

Operating Systems

Associate Prof. Yongkun Li

中科大-计算机学院 副教授

<http://staff.ustc.edu.cn/~ykli>

Chapter 0 Course Overview

主讲老师

- 李永坤

- 教育工作背景

- 本科：中科大；博士：香港中文大学
 - 中科大计算机学院副教授

- 研究方向：存储系统

- 文件系统，Key-value系统
 - 内存系统
 - 图计算，云计算/大数据存储等

- 合作者：

- Prof. John C.S. Lui, Prof. Patrick P.C. Lee @CUHK, Prof. Richard Ma @NUS, Prof. Song Jiang @UTA, Prof. Yipeng Zhou @Macquarie U
 - Prof. Qun Huang@PKU, Hong Xie@CQU

- 主页：<http://staff.ustc.edu.cn/~ykli/>

助教信息

- 课程主页: <http://staff.ustc.edu.cn/~ykli/os2022>
- QQ群: **742371585** (进群验证信息"学号+姓名")

刘朕

liuzhenm@mail.ustc.edu.cn

邓龙

ldeng@mail.ustc.edu.cn

王霄阳

wxy1999@mail.ustc.edu.cn

毛浩宇

maohaoyu@mail.ustc.edu.cn

王志强

wzq666@mail.ustc.edu.cn

李昱祁

yuqi_lee@mail.ustc.edu.cn

李卓远

skeleton_man@mail.ustc.edu.cn

课程介绍

- 教室和时间
 - 理论（60）
 - 周一6-7节（14:00-15:35） & 周三3-4节（9:45-11:20）
 - 3C203
 - 实验（40）： 待定
- 前期课程要求
 - C语言
 - 数据结构
- 课件
 - 英文为主
 - 内容主要来源于WONG Tsz Yeung博士的课件和Operating System Concepts 教材

教材和参考书

教材

– <https://www.os-book.com/OS10/index.html>



参考书



课程要求

- 课堂
 - 按时上课
 - 教材+PPT
- 作业
 - 每1-2周一次，每次5-10个题目
 - 严禁抄袭，按时提交
- 实验
 - ~4次
 - 5-15周，地点待定
 - 严禁抄袭，按时提交

成绩考核

- 基本遵循往年比例分配
 - 作业：20%
 - 实验：30%
 - 期末考试（闭卷）：50%

Operating Systems

Associate Prof. Yongkun Li
中科大-计算机学院 副教授
<http://staff.ustc.edu.cn/~ykli>

Chapter 1 Overview of an Operating System

Objectives

- Overview of OS
 - Overview of Computer System: Organization & Architecture
 - What is an OS
 - OS Operation: Interrupt-driven via system call
- Major OS Components
 - Process Management
 - Memory Management
 - Storage Management
- Kernel Data Structures
- Misc: Computing Environments & Open-Sourced OS

What is an Operating System?

- According to your experience...

- Networking;
- Storage;
- Multimedia;
- Gaming;
- What else?



None of the above were about the OS!

Before we talk about OS...

Overview of Computer System

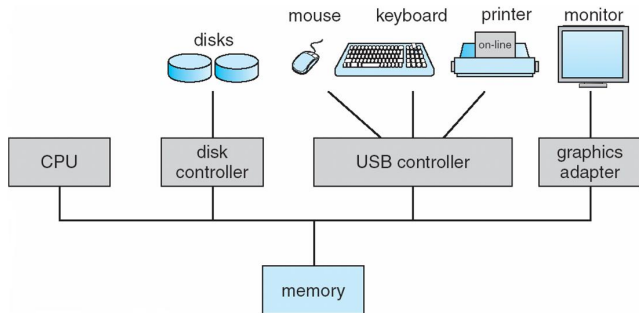
- System Organization

- Storage Structure

- System Architecture

Computer System Organization

- Computer-system organization
 - One or more CPUs, device controllers connect through common bus providing access to shared memory
 - Concurrent execution of CPUs and devices competing for memory cycles



Computer-System Organization

- I/O devices and the CPU can execute concurrently
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer
- CPU moves data from/to main memory to/from local buffers
- Device controller informs CPU that it has finished its operation by causing an **interrupt**

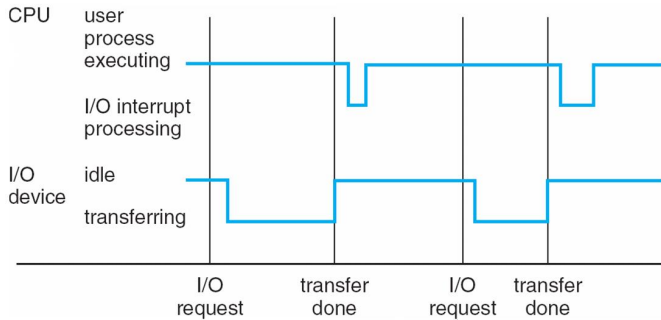
Computer Startup

- **bootstrap program** is loaded at power-up or reboot
 - Typically stored in ROM or EPROM, generally known as **firmware**
 - Initializes all aspects of system
 - Loads operating system kernel into memory and starts execution
- System processes or system daemons
 - Run the entire time the kernel is running
 - On UNIX, the first system process is “init”
- After fully booted, waits for events to occur
 - Signaled by **interrupt**

Interrupt Handling

- Interrupt can be triggered by hardware and software
 - Hardware sends signal to CPU
 - Software executes a special operation: **system call**
- Interrupt procedure
 - CPU stops what is doing
 - Execute the service routine for the interrupt
 - CPU resumes
- Operating system is **interrupt driven**

Interrupt Timeline



Common Functions of Interrupts

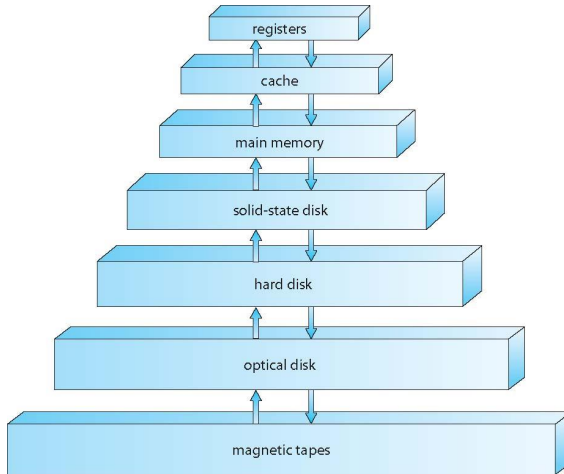
- Each computer design has its own interrupt mechanism
- Interrupt transfers control to the interrupt service routine
 - A table of pointers to interrupt routines, the **interrupt vector**, can be used to provide necessary speed
 - The table of pointers is stored in low memory
- Interrupt architecture must save the address of the interrupted instruction
 - Modern architectures store the return address on system stack

Overview of Computer System

- System Organization
- Storage Structure**
- System Architecture

Storage Structure

- Storage systems organized in hierarchy
 - Speed
 - Cost
 - Volatility



Storage Structure

- Main memory
 - CPU can load instructions only from memory (only large storage media that the CPU can access directly)
 - **Random access**, typically **small size** and **volatile**
 - All forms of memory provide an array of bytes
 - Each byte has its own address
 - Interaction: load & store (memory \leftrightarrow register)
- Instruction-execution cycle
 - Fetch an instruction from memory and store in register
 - Decode instruction (fetch operands if necessary)
 - Store result back to memory

Storage Structure

- Secondary storage – extension of main memory that provides large **nonvolatile** storage capacity
 - Hard disks – rigid metal or glass platters covered with magnetic recording material
 - Disk surface is logically divided into **tracks**, which are subdivided into **sectors**
 - The **disk controller** determines the logical interaction between the device and the computer
 - **Solid-state disks** – faster than hard disks, nonvolatile
 - Various technologies
 - Becoming more popular

Caching

- **Caching** – copying information into faster storage system; main memory can be viewed as a cache for secondary storage
- Faster storage (cache) checked first to determine if information is there
 - If it is, information used directly from the cache (fast)
 - If not, data copied to cache and used there
- Cache smaller than storage being cached
 - Cache management important design problem
 - Cache size and replacement policy
- Important principle, performed at many levels in a computer (in hardware, operating system, software)

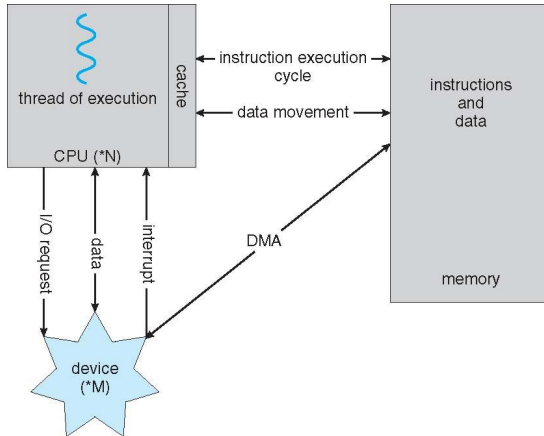
I/O Structure

- Storage is only one of many types of I/O devices
- Device controller
 - More than one device may be attached
 - Local buffer storage & a set of registers
- Device driver: for each device controller to manage I/O, provides uniform interface between controller and kernel
- Interrupt-driven I/O
 - Device driver loads registers within the controller
 - Controller examines the registers to decide what action to take
 - Device controller starts data transfer to its local buffer
 - Informs driver via an interrupt and returns control to OS

Direct Memory Access Structure

- Used for high-speed I/O devices able to transmit information at close to memory speeds
- Device controller transfers **blocks of data** from buffer storage directly to main memory **without CPU intervention**
- Only one interrupt is generated per block, rather than the one interrupt per byte

How a Modern Computer Works



Overview of Computer System

- System Organization
- Storage Structure
- System Architecture**

Computer-System Architecture

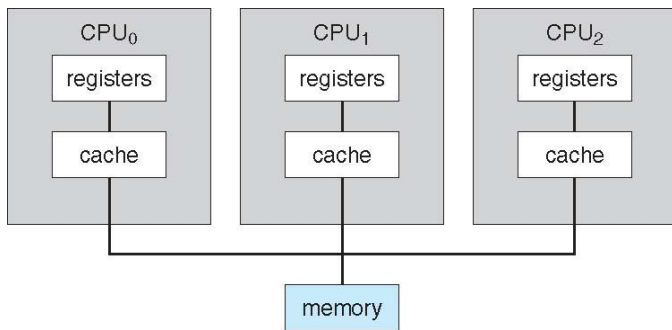
- Most systems use a single general-purpose processor
 - One main CPU capable of executing general-purpose instruction set
- May have special-purpose processors as well
 - Device-specific processors: disk, keyboard, etc...
 - Run a limited instruction set
 - Do not run user processes
 - Managed by OS or built into the hardware

Computer-System Architecture

- **Multiprocessors** systems grow in use and importance
 - Also known as **parallel systems, multicore systems**
- Advantages include:
 - **Increased throughput**
 - **Economy of scale:** share peripherals, mass storage and power supply
 - **Increased reliability** – graceful degradation or fault tolerance
- Two types
 - **Asymmetric Multiprocessing** – each processor is assigned a specific task: boss-worker relationship
 - **Symmetric Multiprocessing (SMP)** – each processor performs all tasks: all processors are peers

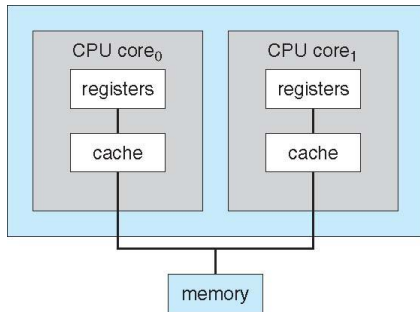
Symmetric Multiprocessing Architecture

- Symmetric Multiprocessing (SMP)
 - Result from hardware or software
 - Adds CPUs to increase computing power
 - Causes non-uniform memory access (NUMA)



Multicore

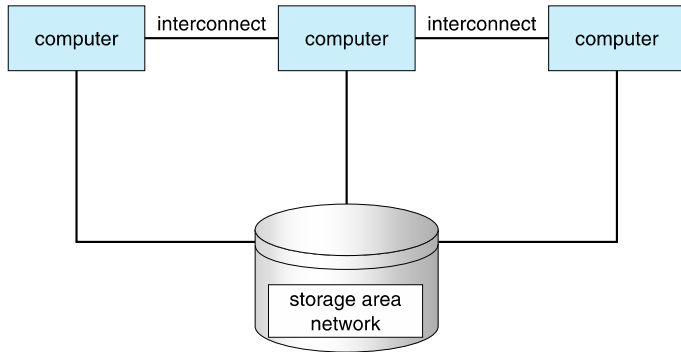
- **Multicore:** include multiple cores on a single chip
- More efficient
 - On-chip communication is faster than between-chip communication
 - Less power
- Dual-core design



Clustered Systems

- Like multiprocessor systems, but multiple systems working together
 - Usually sharing storage via a **storage-area network (SAN)**
 - Provides a **high-availability** service which survives failures
 - **Asymmetric clustering** has one machine in hot-standby mode
 - **Symmetric clustering** has multiple nodes running applications, monitoring each other
 - Some clusters are for **high-performance computing (HPC)**
 - Applications must be written to use **parallelization**
 - Some have **distributed lock manager (DLM)** to avoid conflicting operations

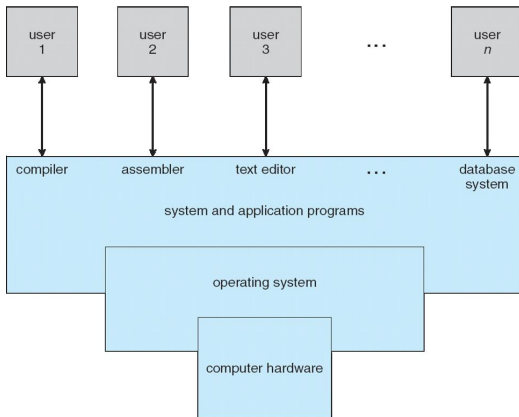
Clustered Systems



What is an Operating System?

Where is the OS?

- Let's start understanding an OS from this question:
Where is it?

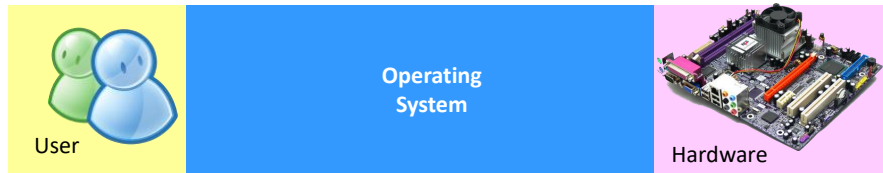


Where is the OS?

- Four components of a computer system
 - **Hardware** – provides basic computing resources (CPU, memory, I/O devices)
 - **Users:** People, machines, other computers
 - **App. programs** – define the ways in which the sys. resources are used to solve the computing problems
 - Word processors, compilers, web browsers, database systems, video games, etc.
 - **Operating system**
 - Controls and coordinates use of hardware among various applications and users

What is an Operating System?

- It stands **between** the hardware and the user.
 - A program that acts as an intermediary between a user of a computer and the computer hardware



- Operating system goals:
 - Execute user programs & make solving user problems easier
 - Make the computer system **convenient** to use
 - Use the computer hardware in an **efficient** manner
 - Design **tradeoff** between convenient and efficiency

What is an Operating System?

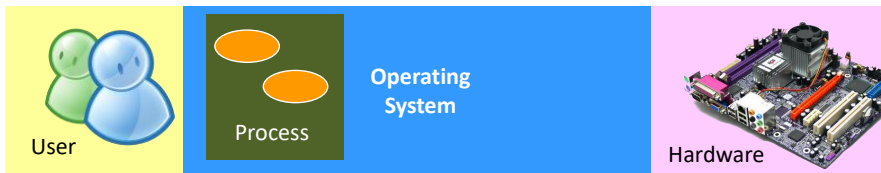
- How good is this design?
 - The user does not have to program the hardware directly.
 - It hides all the troublesome operations of the hardware.

Example. The OS, on one hand, hides the physical system memory away from you. On the other hand, it tells you that there is system memory available when you run your applications.



What is an Operating System?

- **Processes** as the starting point!
 - Whatever programs you run, you create **processes**.
 - i.e., you need processes to open files, utilize system memory, listen to music, etc.
 - So, process lifecycle, process management, and other related issues are essential topics of this course.

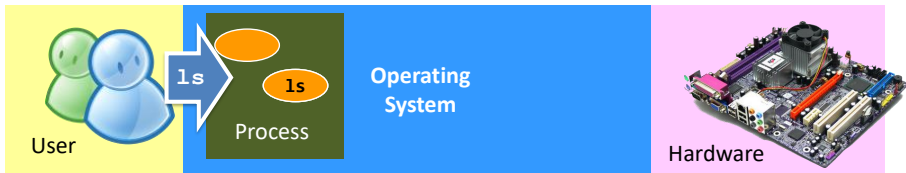


What is an Operating System?

- Example (step 1)

Most commands you type in the shell are the same as starting a new process.

```
$ ls
```



What is an Operating System?

- Example (step 2)

The operating system contains the codes that are needed to work with the file system.

The codes are called the kernel.

\$ ls

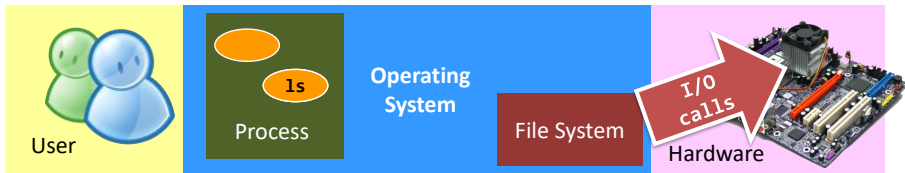


What is an Operating System?

- Example (step 3)

The file system module inside the operating system knows how to work with devices, using device drivers.

\$ ls

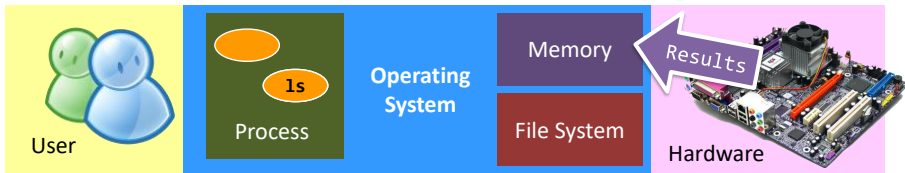


What is an Operating System?

- Example (step 4)

Of course, the operating system will allocate memory for the results.

\$ 1s



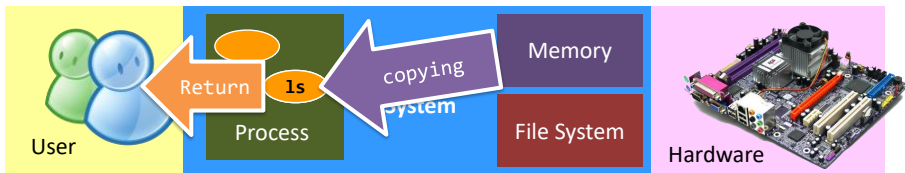
What is an Operating System?

- Example (final step)

The memory management sub-system will copy the result to the memory of the process.

At last, the result returns.

```
$ ls  
.  ..  index.html  
$ _
```



What Operating Systems Do

- **System View**

- OS is a **control program**

- Controls execution of programs to prevent errors and improper use of the computer

- OS is a **resource allocator**

- Manages all resources
- Decides between conflicting requests for efficient and fair resource use

What Operating Systems Do

- Depends on the point of view
- **User View**
 - PC users want convenience, **ease of use** and **good performance**, don't care about resource utilization
 - But shared computer such as mainframe or minicomputer must keep all users happy: maximize **resource utilization**
 - Users of dedicated systems such as workstations have dedicated resources but frequently use shared resources from servers: **tradeoff**
 - Mobile computers are resource poor, optimized for **usability and battery life**
 - Some computers have little or no user interface, such as embedded computers in devices and automobiles

Operating System Definition

- No universally accepted definition
- Simple viewpoint
 - “Everything a vendor ships when you order an operating system” is a good approximation
 - But varies wildly
- Common definition
 - “The one program running at all times on the computer” is the **kernel**.
- Everything else is either
 - a system program (ships with the operating system) , or
 - an application program.

Operating System Definition (Cont.)

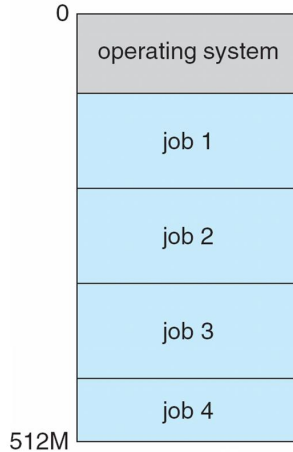
- No universally accepted definition of what is part of the operating system
 - Operating systems grew increasingly sophisticated
 - Microsoft case
- Current Mobile OS
 - Once again the number of features constituting the OS is increasing
 - Core kernel + Middleware
 - Databases, multimedia, graphics, etc...

Operating System Operations

Multiprogramming

- Operating system provides the environments within which programs are executed
 - Single user cannot keep CPU and I/O devices busy at all times
- **Multiprogramming** needed for efficiency: most important aspect of OS
 - Multiprogramming organizes jobs (code and data) so CPU always has one to execute
 - All jobs are initially kept on disk in the job pool, a subset of total jobs in system is kept in memory,
 - One job selected and run via **job scheduling**
 - When it has to wait (for I/O for example), OS switches to another job

Memory Layout for Multi-programmed System



Multitasking

- **Time sharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
 - **Response time** should be < 1 second
- Allow many users to share the computer
 - Each user has at least one program executing in memory
⇒ **process**
- Issues
 - If several jobs ready to run at the same time ⇒ **CPU scheduling**
 - If processes don't fit in memory, **swapping** moves them in and out to run
 - **Virtual memory** allows execution of processes not completely in memory

Interrupt Driven Mechanism

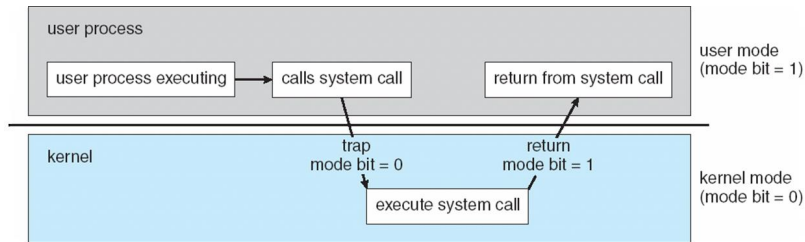
- **Interrupt driven** (hardware and software)
 - Hardware interrupt by one of the devices
 - Software interrupt (**exception** or **trap**):
 - Software error (e.g., division by zero)
 - **Request for operating system service**
 - Other process problems include infinite loop, processes modifying each other or the operating system
 - An interrupt service routine is provided to deal with the interrupt

Dual-mode Operation

- **Dual-mode** operation allows OS to protect itself and other system components
 - **User mode** and **kernel mode**
 - **Mode bit** provided by hardware
 - Provides ability to distinguish when system is running user code or kernel code
 - Some instructions designated as **privileged**, only executable in kernel mode
 - System call changes mode to kernel, return from call resets it to user

Transition from User to Kernel Mode

- At system boot time, the hardware starts in kernel mode
- OS is loaded and starts user application in user mode
- Interrupt occurs, the hardware switches from user mode to kernel mode
- Whenever the OS gains control, it is in kernel mode



System Calls

- Informally, a system call is similar to a function call, but...
 - The function implementation is inside the OS.
 - We name it the **OS kernel**.

```
int add_function(int a, int b) {  
    return (a + b);  
}
```

Function
implementation.

```
int main(void) {  
    int result;  
    result = add_function(a,b);  
    return 0;  
}
```

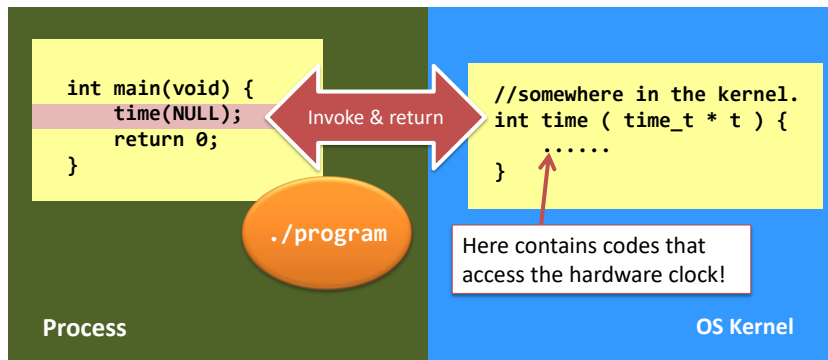
This is a
function call.

```
// this is a dummy example...
```

System Calls

- System calls are the **programming interface** between processes and the OS kernel
 - System calls provide the means for a user program to ask the operating system to perform tasks
- A system call usually takes the form of a trap to a specific location in the interrupt vector, **treated by the hardware as a software interrupt**
- The system call service routine is a part of the OS

Interacting with the OS



System calls

- The system calls are usually
 - **primitive**,
 - **important**, and
 - **fundamental**.
 - e.g., the **time()** system call.
- Roughly speaking, we can categorize system calls as follows:

Process	File System	Memory
Security	Device	

System calls VS Library function calls

- If a call is not system calls, then they are **library calls** (or function calls)!
- Take **fopen()** as an example.
 - **fopen()** invokes the system call **open()**.
 - So, why people invented **fopen()**?
 - Because **open()** is too primitive and is not programmer-friendly!

Library call

```
fopen("hello.txt", "w");
```

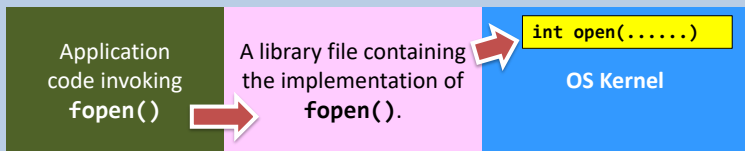
System call

```
open("hello.txt", O_WRONLY | O_CREAT | O_TRUNC, 0666);
```

System calls VS Library function calls

- Library functions are usually compiled and packed inside an object called the **library file**.
 - In windows: DLL – dynamically linked library.
 - In Linux: SO – shared objects.

Big picture



OS Standards

- Who defines the system calls? Functionalities? Arguments? Return values?
- There are standards!

Standards	Full Name	Example OS
POSIX	Portable Operating System Interface	Linux
BSD	Berkeley Software Distribution	Mac OS Darwin
SVR4	System V (five) Release 4	Solaris Unix

Introduction to Operating System Components

Process

Process OR Program?

- A process is not a program!

Let's consider the following two commands

Command A

```
ls -R /
```

Recursively print the directory entries,
starting from the directory '/'

Command B

```
ls -R /home
```

Recursively print the directory entries,
starting from the directory '/home'

Similarity	Difference
Both use the program file <code>"/bin/ls"</code> .	The program arguments are different.
---	The processes' internal status are different, such as running time.

Program != Process

- A process is an **execution instance** of a program.
 - More than one process can execute the same program code
 - Later, you'll find that a process is not bounded to execute just one program!
- A process is active.
 - A process has its **local states** concerning the execution. E.g.,
 - which line of codes it is running;
 - which CPU core (if there are many) it is running on.
 - The local states change over time.
- Commands about processes (and hopefully you've tried them before) – e.g., **ps** & **top**.

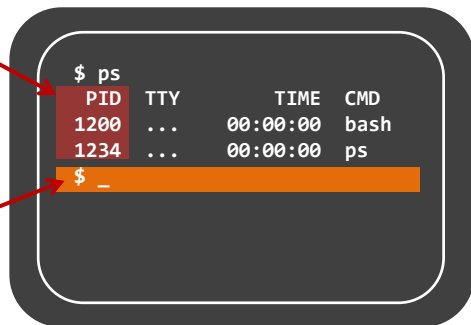
Process-Related Tools

- The tool “**ps**” can report a vast amount of information about every process in the system
 - Try “**ps -ef**”.

This column shows the unique identification number of a process, called **Process ID**, or PID for short.

Hint: you can treat **ps** as the short-form of “**p**rocess **s**tatus”

By the way, this is called **shell**.



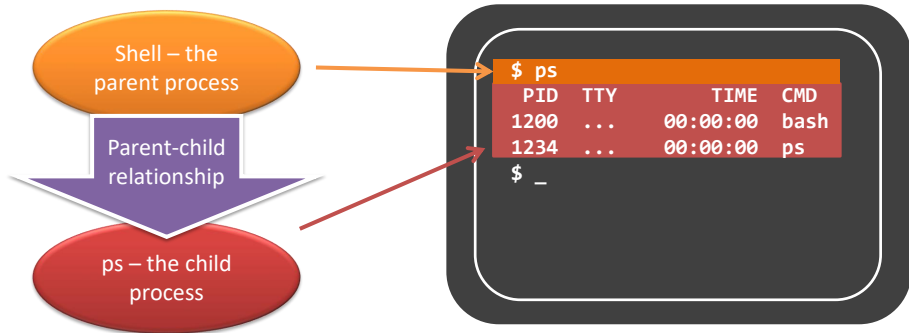
```
$ ps
```

PID	TTY	TIME	CMD
1200	...	00:00:00	bash
1234	...	00:00:00	ps

```
$ _
```

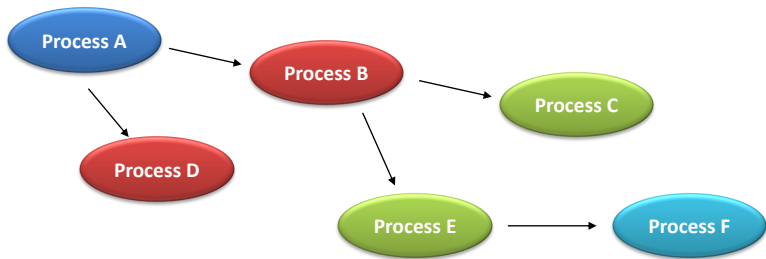
Shell – a process launching pad

- So, what is going on inside that shell?
 - The shell creates a new process, and is called a **child process** of the shell.
- The child process then executes the command “**ps**”.



Process Hierarchy

- Process relationship:
 - A parent process will have its child process.
 - Also, a child process will have its child processes.
 - This form a **tree hierarchy**.



E.g., "Process E" is the shell and "Process F" is "**ps**".

Process Summary

- A process is an execution instance of a program. It is a unit of work within the system.
 - Program is a ***passive entity***, process is an ***active entity***.
- Process needs resources to accomplish its task, process termination requires reclaim of any reusable resources
 - CPU, memory, I/O, files, Initialization data
- Single-threaded process has one **program counter** specifying location of next instruction to execute, multi-threaded process has one program counter per thread
 - Process executes instructions sequentially, one at a time, until completion
- Typically, system has many processes, some user, some operating system running concurrently

Process Management Activities

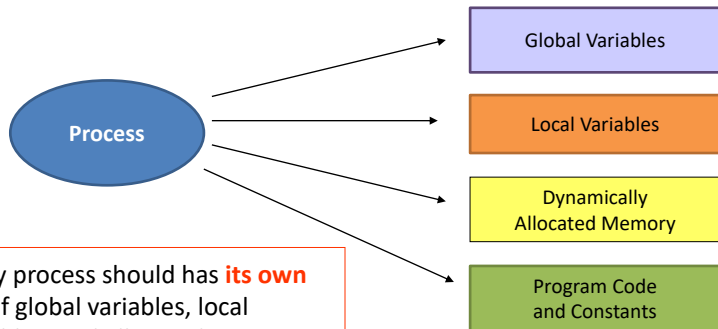
- The operating system is responsible for the following activities in connection with process management:
 - Creating and deleting both user and system processes
 - Suspending and resuming processes
 - Providing mechanisms for **process synchronization**
 - Providing mechanisms for **process communication**
 - Providing mechanisms for **deadlock handling**

Introduction to Operating System Components

Memory

Process' Memory

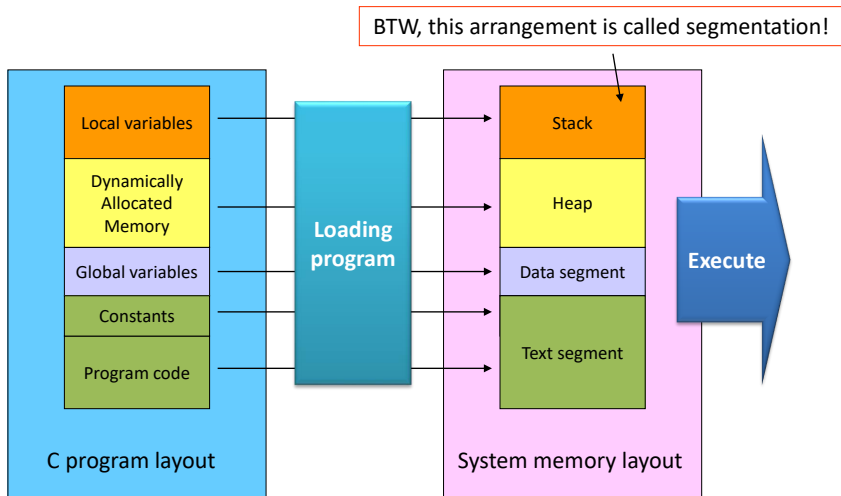
- What are the things that a process has to store?



Every process should have **its own set** of global variables, local variables, and allocated memory.

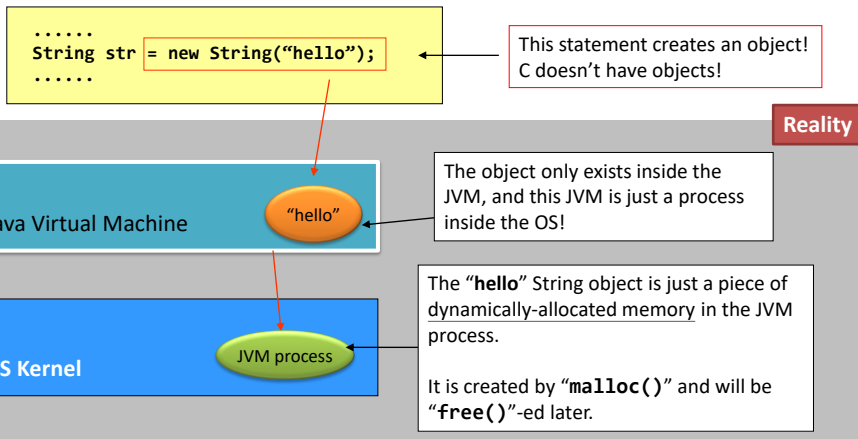
Process' Memory

- OMG...C is too low-level...



Process' Memory

- “*Hey, you’re wrong! Some languages, e.g., Java, do not have the above layout...*”, you asked.



Sidetrack: Pros and Cons in using C

- Cons:

- Some people argued that C is a **bad beginner's programming language**. Now, you can understand why...

Because C requires a programmer to take care of the process-level memory management.

Every programmer needs to know about the low-level memory layout in order for him/her to understand what segmentation fault means!

Every aspect on memory management can be manipulated using C.

Learning **malloc()** exposes you to the heap manipulation. This makes a high-level programming language becoming low-level. Plus, **this exposes you to unpredictable dangers!**

* Disclaimer: choosing which programming language is really a personal choice.

Sidetrack: Pros and Cons in using C

- Pros:
 - Some people argued that C is an **efficient programming language**. Now, you can understand why...

Because C allows a programmer to **manipulate the process-level memory management “directly”**.

That's why many user libraries are implemented using C because of efficiency consideration.

E.g., the Java Virtual Machine is implemented using C!

Most importantly, **C is the only language to interact with the OS directly!**
In other words, the system call interface is written in C.

* Disclaimer: choosing which programming language is really a personal choice.

Memory Hierarchy

- In case that someone doesn't know about the hierarchy below...
 - A program is fetched **from hard disk to main memory**.
 - When executed, instructions in the program are fetched **from the main memory to CPU**.



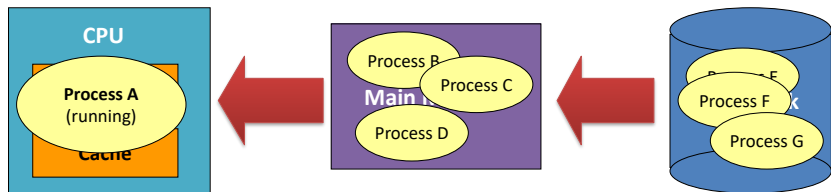
Memory Hierarchy

- However, did you ever need to program those three things when you want to run the program “ls”?
 - Never! Then, who have the jobs done?
 - Of course, **OS**!



Memory Hierarchy

- Typically, there are more than 100 processes running "at the same time".
 - There is only a finite number of CPU cores, depending on how much money you spent.
 - Then, only a finite number of processes can be executed "really at the same time".
 - So, other (non-running) processes are stored at different devices controlled by the OS before they get a chance to run.



Memory Management Summary

- To execute a program
 - All (or part) of the instructions must be in memory
 - All (or part) of the data that is needed by the program must be in memory.
- Memory management determines what is in memory
 - Optimizing CPU utilization and computer response to users
- Memory management activities
 - Keeping track of which parts of memory are currently being used and by whom
 - Deciding which processes (or parts thereof) and data to move into and out of memory
 - Allocating and deallocating memory space as needed

Introduction to Operating System Components

Storage Management

What is a File System?

- A file system, FS, means the way that a storage device is used.
- Have you heard of...
 - FAT16, FAT32, NTFS, Ext3, Ext4, Btrfs?
 - They are all file systems.
 - It is about how a storage device is utilized.

Index

Metadata

Files / Data

What is a File System?

- A file system must record the following things:
 - directories;
 - files;
 - allocated space;
 - free space.
- Think about the consequences if any one of the above is missing...

Two faces of a file system

- The **storage design** of the file system.
 - A file spends most of its time on the disk.
 - So, a file system is about how they are stored.
 - Apart from files, many others things are stored in the disk.
- The **operations** of the file system.
 - A file can be manipulated by processes.
 - So, a file system is also about the operations which manipulate the content stored.

- **A FS is independent of an OS!**
 - If an OS supports a FS, then the OS can do whatever operations over that storage device.
 - Else, the OS doesn't know how to read or update the device's content.

Windows XP supports	Linux supports
NTFS, FAT32, FAT16, ISO9660, Juliet, CIFS	NTFS, FAT32, FAT16, ISO9660, Juliet, CIFS, Ext2, Ext3, etc...

Linux supports far more FS-es than any versions of Windows

File Operations?

- Pop quiz!
 - Guess, what are the fundamental file (not dir) operations?

Open	Read	Write	Close	Rename	Delete
------	------	-------	-------	--------	--------

- Well...creating is not...
 - It is just a special case of opening a file.
- Sorry...copying is not...
 - Do you know how it is implemented through the above operations?
- Sorry...moving is the same as renaming...
 - Except that a file is moving from one disk to another.

Storage Management

- OS provides uniform, logical view of information storage
 - Abstracts physical properties to logical storage unit - **file**
 - Various devices (i.e., disk drive, tape drive)
 - Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)
- File-System management
 - Files usually organized into directories
 - **Access control** to determine who can access what
 - OS activities include
 - Creating and deleting files and directories
 - Primitives to manipulate files and directories
 - Mapping files onto secondary storage
 - Backup files onto stable (non-volatile) storage media

Mass-Storage Management

- Usually disks used to store data that does not fit in main memory or data that must be kept for a long period of time
- Proper management is of central importance
 - Entire speed of computer operation hinges on disk subsystem and its algorithms
- OS activities
 - Free-space management
 - Storage allocation
 - Disk scheduling
- Some storage need not be fast
 - Tertiary storage includes optical storage, magnetic tape
 - Still must be managed – by OS or applications

Performance of Various Levels of Storage

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Kernel Data Structures

Kernel Data Structures

Lists, Trees, Hash Map and Bitmaps

Kernel Data Structures

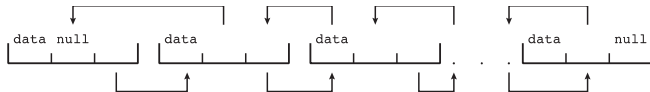
- Many similar to standard programming data structures

- **Lists**

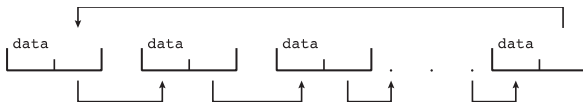
- Singly linked list



- Doubly linked list



- Circularly linked list



Kernel Data Structures

- **Stack**

- Last in first out (LIFO)
- Widely used when invoking function calls

- **Queue**

- First in first out (FIFO)
- Widely used in job scheduling

Kernel Data Structures

- **Trees**

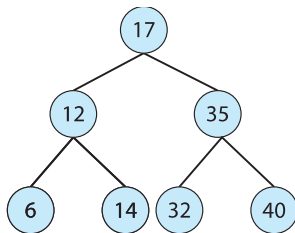
- **Binary tree**

- **Binary search tree**: left \leq right

- Worse-case search performance is $O(n)$

- **Balanced binary search tree**

- Worse-case search performance is $O(\lg n)$

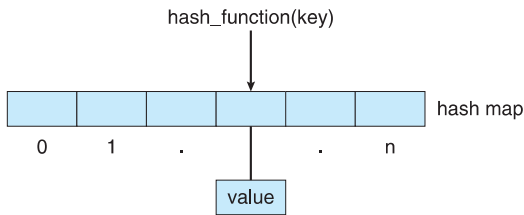


Kernel Data Structures

- **Hash function**

- Takes data as input, performs numeric operation on the data, and returns a numeric value
- Retrieve data: $O(1)$
- Hash collision

- **Hash function** can create a **hash map**



Kernel Data Structures

- **Bitmap** – string of n binary digits representing the status of n items
- Pros:
 - Space efficiency
- Example: used to indicate the availability of disk blocks
- Linux data structures defined in ***include*** files
`<linux/list.h>`, `<linux/kfifo.h>`,
`<linux/rbtree.h>`

MISC

Protection and Security, Computing Environments and Open-sourced OS

Protection and Security

- **Protection** – any mechanism for **controlling access** of processes or users to resources defined by the OS
- **Security** – defense of the system against internal and external **attacks**
 - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first **distinguish among users**, to determine who can do what
 - User identities (**user IDs**, security IDs) include name and associated number, one per user, determine access control
 - Group identifier (**group ID**) allows set of users to be defined and controls managed, then also associated with each process, file
 - **Privilege escalation** allows user to change to effective ID with more rights

Computing Environments - Traditional

- **Stand-alone** general purpose machines
- Blurred as most systems interconnect with others (i.e., the Internet)
 - **Portals** provide web access to internal systems
 - **Network computers** (**thin clients**) are like Web terminals
 - Mobile computers interconnect via **wireless networks**
- **Networking becoming ubiquitous** – even home systems use **firewalls** to protect home computers from Internet attacks

Computing Environments - Mobile

- Handheld smartphones, tablets, etc
- What is the **functional difference** between them and a “traditional” laptop?
 - Extra feature – more OS features (GPS, gyroscope)
 - Allows new types of apps like ***augmented reality***
 - Use IEEE 802.11 wireless, or cellular data networks for connectivity
- Leaders are **Apple iOS** and **Google Android**

Computing Environments – Distributed

- Distributed computing
 - Collection of separate, possibly heterogeneous, systems **networked together**
 - **Network** is a communication path, **TCP/IP** most common
 - **Local Area Network (LAN)**
 - **Wide Area Network (WAN)**
 - **Metropolitan Area Network (MAN)**
 - **Personal Area Network (PAN)**
 - **Network Operating System** provides features between systems across network
 - Communication scheme allows systems to exchange messages
 - Illusion of a single system

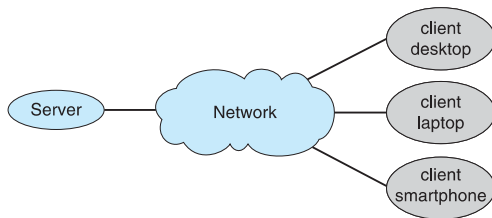
Computing Environments – Client-Server

Client-Server Computing

Dumb terminals supplanted by smart PCs

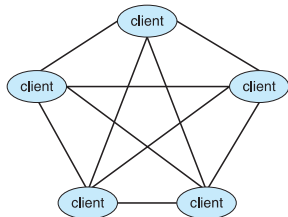
Many systems act as **servers**, responding to requests generated by **clients**

- ▶ **Compute-server system** provides an interface to client to request services (i.e., database)
- ▶ **File-server system** provides interface for clients to store and retrieve files



Computing Environments - Peer-to-Peer

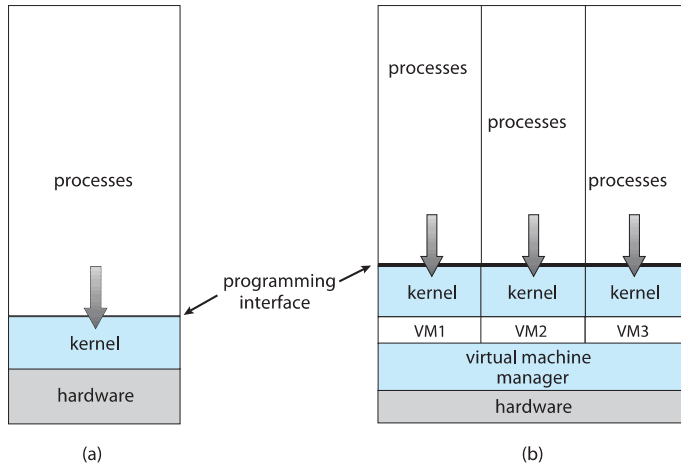
- Another model of distributed system, does not distinguish clients and servers
 - Instead all nodes are considered peers
 - May each act as client, server or both
 - Node must join P2P network
 - Registers its service with **central lookup service** on network, or
 - **Broadcast** request for service and respond to requests for service via **discovery protocol**
 - Examples include BitTorrent



Computing Environments - Virtualization

- Allows OSes to run applications within other OSes
- **Emulation** used when source CPU type is different from target type (i.e. PowerPC to Intel x86)
 - Generally slowest method
 - Every machine-level instruction must be translated
- **Virtualization** – OS natively compiled for CPU, running **guest** OSes also natively compiled
 - **Running multiple VMs** allows many users to run tasks on a system designed for a single user
 - **VMM** (Virtual Machine Manager) provides virtualization services

Computing Environments - Virtualization

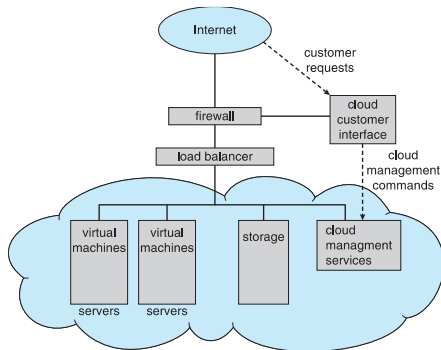


Computing Environments – Cloud Computing

- Delivers computing, storage, even apps as a service across a network
- **Logical extension of virtualization** because it uses virtualization as the base for its functionality.
 - Amazon **EC2** has thousands of servers, millions of virtual machines, petabytes of storage available across the Internet, **pay based on usage**
- Many types
 - **Public cloud** – available via Internet to anyone willing to pay
 - **Private cloud** – run by a company for the company's own use
 - **Hybrid cloud** – includes both public and private cloud components
 - Software as a Service (**SaaS**) – one or more applications available via the Internet (i.e., word processor)
 - Platform as a Service (**PaaS**) – software stack ready for application use via the Internet (i.e., a database server)
 - Infrastructure as a Service (**IaaS**) – servers or storage available over Internet (i.e., storage available for backup use)

Computing Environments – Cloud Computing

- Cloud computing environments composed of traditional OSes, plus VMMs, plus cloud management tools
 - Internet connectivity requires security like firewalls
 - **Load balancers** spread traffic across multiple applications



Computing Environments – Real-Time Embedded Systems

- Real-time embedded systems: most prevalent form of computers
 - Car engines, robots, DVDs, etc.
- Real-time OS has well-defined fixed time constraints
 - Processing **must** be done within constraint
 - Correct operation only if constraints met
- Many other special computing environments as well
 - Some have OSES, some perform tasks without an OS

Open-Source Operating Systems

- Operating systems made available in source-code format rather than just binary **closed-source**
- Started by **Free Software Foundation (FSF)**, which has “copyleft” **GNU Public License (GPL)**
- Examples include **GNU/Linux** and **BSD UNIX** (including core of **Mac OS X**)
- Can use VMM like VMware Player (Free on Windows), Virtualbox (open source and free on many platforms - <http://www.virtualbox.com>)
 - Use to run guest operating systems for exploration

Summary

- OS Overview
 - OS Concept
 - Multiprogramming & Multitasking
 - Dual Mode & System Call
- OS Components
 - Process Management
 - Memory Management
 - Storage Management
- Computer System Organization & Architecture
 - Interrupt

End of Chapter 1

Operating Systems

Associate Prof. Yongkun Li

中科大-计算机学院 副教授

<http://staff.ustc.edu.cn/~ykli>

Chapter 2 Operating System Structures

Objectives

- Operating System Services
 - User Operating System Interface
 - System Calls
- Operating System Structure
- Operating System Design and Implementation
- MISC: Debugging, Generation & System Boot

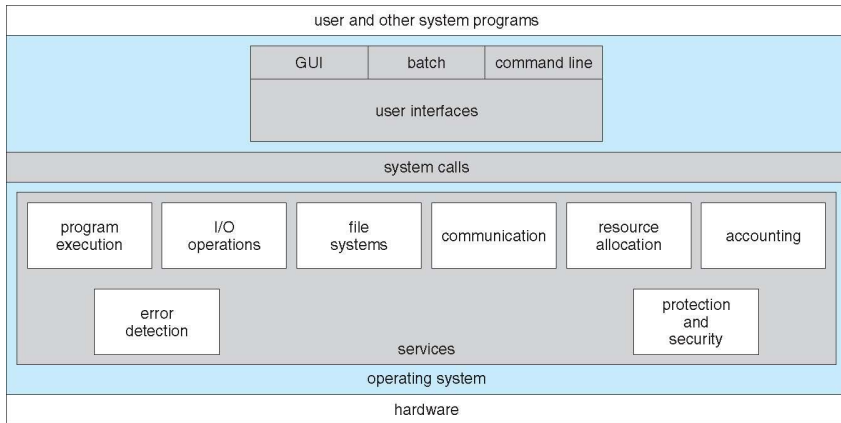
Operating System Services

Services Overview, User Interface

Operating System Services

- Operating systems provide
 - an environment for **execution** of programs and
 - **services** to programs and users
- Services may differ from one OS to another
- What are the common classes?
 - **Convenience** of the user
 - **Efficiency** of the system

Overview of Operating System Services



OS Services for Helping Users

- **Program execution**
 - Load a program into memory
 - Run the program
 - End execution
 - either normally or
 - abnormally (indicating error)

OS Services for Helping Users

- **I/O operations** - A running program may require I/O, which may involve a file or an I/O device
 - Common I/Os: read, write, etc.
 - Special functions: recording CD/DVD
- Notes: Users usually cannot control I/O devices directly, so OS provides a mean to do I/O
 - Mainly for efficiency and protection

OS Services for Helping Users

- **File-system manipulation** - The file system is of particular interest
 - OS provides a variety of file systems
- Major services
 - read and write files and directories
 - create and delete files and directories
 - search for a given file
 - list file Information
 - permission management: allow/deny access

OS Services for Helping Users

- **Communications:** information exchange between processes
 - Processes on the same computer
 - Processes between computers over a network
- Implementations
 - **Shared memory**
 - Two or more processes read/write to a shared section of mem.
 - **Message passing**
 - Packets of information are moved between processes by OS

OS Services for Helping Users

- **Error detection** – OS needs to be constantly aware of possible errors
- **Error types**
 - CPU
 - memory hardware: memory error, power failure, etc.
 - I/O devices: parity error, connection failure, etc.
 - user program: arithmetic overflow, access illegal mem.
- **Error handling**
 - Ensure correct and consistent computing
 - Halt the system, terminate an error-causing process etc.

OS Services for Ensuring Efficiency

- Systems with multiple users can gain efficiency by sharing the computer resources
- **Resource allocation**
 - Resources must be allocated to each user/job
 - **Resource types** - CPU cycles, main memory, file storage, I/O devices
 - Special allocation code may be required, e.g., CPU scheduling routines depend on
 - Speed of the CPU, jobs, number of registers, etc.

OS Services for Ensuring Efficiency

- **Accounting** - To keep track of
 - which users use how much and what kinds of resources
- **Usage**
 - Accounting for **billing** users
 - Accumulating **usage statistics**, can be used for
 - Reconfiguration of the system
 - Improvement of the efficiency

OS Services for Ensuring Efficiency

- **Protection and security**
 - Concurrent processes should not interfere w/ each other
 - Control the use of computer
- **Protection**
 - Ensure that all access to system resources is **controlled**
- **Security**
 - **User authentication** by password to gain access
 - Extends to **defending external I/O devices** from invalid access attempts

OS Services for Helping Users

- **User interface** - Almost all operating systems have a user interface (**UI**).
 - Three forms
 - **Command-Line (CLI)**
 - Shell command
 - **Batch**
 - Shell script
 - **Graphics User Interface (GUI)**
 - Windows system

User Operating System Interface - CLI

- Command line interface or command interpreter
 - Allows direct command entry
 - Included in the kernel or treated as a special program
- Sometimes multiple flavors implemented – **shells**
 - Linux: multiple shells (C shell, Korn Shell etc.)
 - Third-party shell or free user-written shell
 - Most shells provide similar functionality (personal preference)

Bourne Shell Command Interpreter

```
Default
New Info Close Execute Bookmarks

PBG-Mac-Pro:~ pbg$ w
15:24 up 56 mins, 2 users, load averages: 1.51 1.53 1.65
USER      TTY      FROM          LOGIN@  IDLE WHAT
pbg       console -              14:34   50 -
pbg       s000    -              15:05   - w
PBG-Mac-Pro:~ pbg$ iostat 5
            disk0      disk1      disk10      cpu      load average
      KB/t tps MB/s    KB/t tps MB/s    KB/t tps MB/s  us sy id  1m  5m  15m
      33.75 343 11.30    64.31 14  0.88    39.67 0  0.02 11  5 84  1.51 1.53 1.65
      5.27 320  1.65     0.00 0  0.00     0.00 0  0.00  4  2 94  1.39 1.51 1.65
      4.28 329  1.37     0.00 0  0.00     0.00 0  0.00  5  3 92  1.44 1.51 1.65
AC
PBG-Mac-Pro:~ pbg$ ls
Applications          Music                  WebEx
Applications (Parallels)  Pando Packages       config.log
Desktop               Pictures              getsmartdata.txt
Documents             Public                imp
Downloads             Sites                 log
Dropbox               Thumbs.db             panda-dist
Library               Virtual Machines      prob.txt
Movies                Volumes               scripts
PBG-Mac-Pro:~ pbg$ pwd
/Users/pbg
PBG-Mac-Pro:~ pbg$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: icmp_seq=0 ttl=64 time=2.257 ms
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=1.262 ms
AC
--- 192.168.1.1 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 1.262/1.760/2.257/0.498 ms
PBG-Mac-Pro:~ pbg$
```

User Operating System Interface - CLI

- **Main function** of CLI
 - Get and execute the next user-specified command
 - Many commands manipulate files
- Two ways of **implementing commands**
 - The command interpreter itself contains the code
 - Jump to a section of its code & make appropriate system call
 - Number of commands determines the size of CLI
 - Implements commands through system program (UNIX)
 - CLI does not understand the command
 - Use the command to identify a file to be loaded into memory and executed
 - Exp: rm file.txt (search for file rm, load into memory and exe w/ file.txt)
 - Add new commands easily

User Operating System Interface - GUI

- User-friendly graphical user interface
 - Mouse-based window-and-menu system (desktop metaphor)
 - Icons represent files, programs, actions, etc
 - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a folder))
 - Invented at Xerox PARC in early 1970s
- Many systems now include both CLI and GUI interfaces
 - Microsoft Windows is GUI with CLI “command” shell
 - Apple Mac OS X is “Aqua” GUI interface with UNIX kernel
 - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

Touchscreen Interfaces

- Touchscreen devices require new interfaces
 - Mouse not possible or not desired
 - Actions and selection based on **gestures**
 - Virtual keyboard for text entry
 - Voice commands



Choices of Interfaces

- Personal preference
- CLI: **more efficient, easier for repetitive tasks**
 - System administrator
 - Power users who have deep knowledge of a system
 - Shell scripts
- GUI: user-friendly
- The design and implementation of user interface is **not a direct function** of the OS

System Call

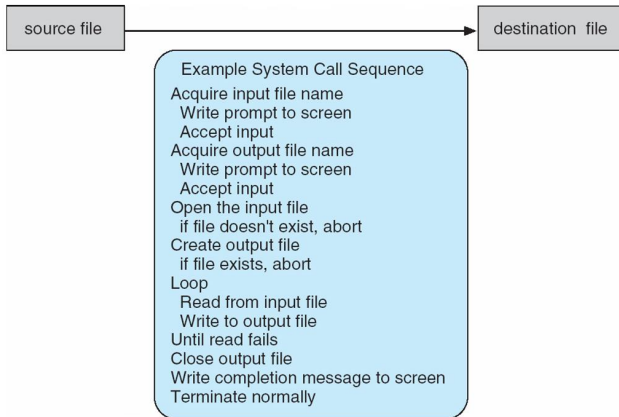
Usage, Implementation, Types

System Calls

- **Programming interface** to the services provided by the OS
- Implementation language
 - Typically written in a high-level language (C or C++)
 - Certain low-level tasks (direct hardware access) are written using assembly language
- **Example** of using system call
 - Read data from a file and copy to another file
 - `open()` + `read()` + `write()`?

Example of System Calls

- System call sequence to copy the contents of one file to another file



System Call

- Simple programs may make heavy use of the OS
 - A system executes thousands of system calls per second
 - Not user-friendly
- Each OS has its own name for each system call
 - This course/textbook uses generic examples

System Call

- How to use?
 - Mostly accessed by programs via a high-level **API** rather than direct system call use
- Why prefer API rather than invoking system call?
 - **Easy of use**
 - Simple programs may make heavy use of the OS
 - **Program portability**
 - Compile and run on any system that supports the same API

- **Application Programming Interface (API)**
 - A set of functions that are available to application programmers

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

#include <unistd.h>		
ssize_t	read(int fd, void *buf, size_t count)	
return value	function name	parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns `-1`.

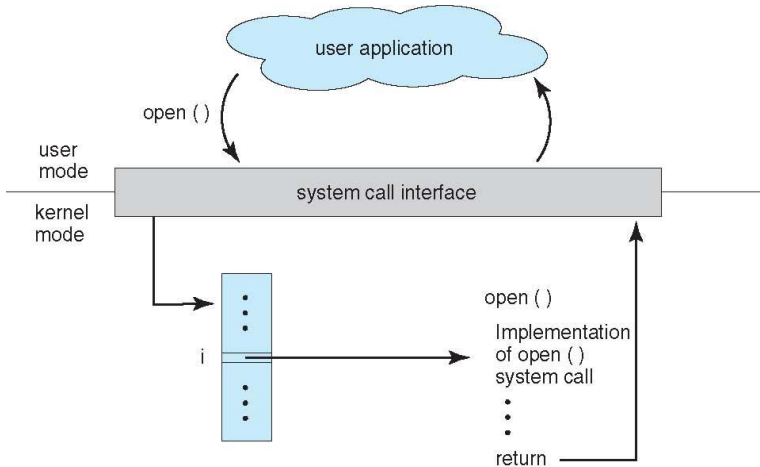
API

- **Application Programming Interface (API)**
 - A set of functions that are available to application programmers
- Three most common APIs
 - Win32 API for Windows
 - POSIX API for POSIX-based systems
 - including virtually all versions of UNIX, Linux, and Mac OS X
 - Java API for the Java virtual machine (JVM)
- How to use API?
 - Via a library of code provided by OS
 - Libc: UNIX/LINUX with C language

System Call Implementation

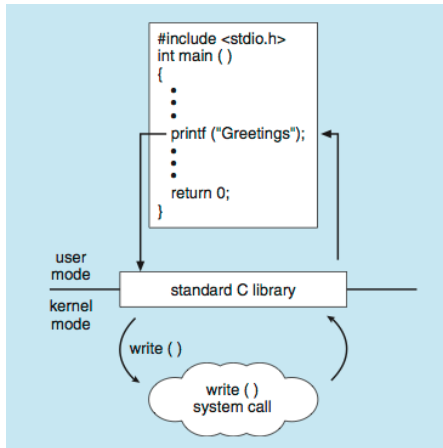
- Who invokes system call: **System call interface**
 - Provided by the **run-time support system**, which is
 - a set of functions built into libraries within a compiler
- How?
 - intercepts function calls in the API
 - invokes necessary system calls
- Implementation
 - Typically, a number associated with each system call
 - System-call interface maintains a table indexed according to the numbers

API – System Call – OS Relationship



Standard C Library Example

- C program invoking `printf()` library call, which calls `write()` system call



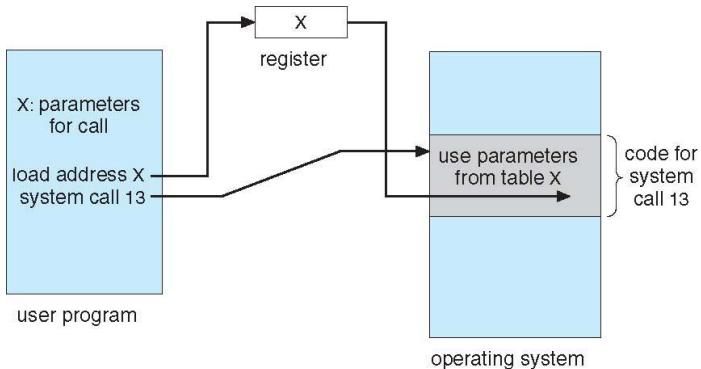
Implementation Benefits

- The caller needs to know nothing about
 - how the system call is implemented
 - what it does during execution
 - Just needs to obey API and understand what OS will do as a result call
- Most details of OS interface are hidden from programmer by API
 - Managed by run-time support library

System Call Parameter Passing

- More information is required than simply the identity of desired system call
 - Parameters: file, address and length of buffer
- Three methods to **pass parameters** to the OS
 - Simplest: pass the parameters in **registers**
 - In some cases, may be more parameters than registers
 - Table-based
 - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
 - This approach taken by Linux and Solaris
 - Stack-based
 - Parameters are placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system

Parameter Passing via Table



Types of System Calls

- Six major categories
 - Process control
 - File manipulation
 - Device manipulation
 - Information maintenance
 - Communications
 - Protection

Types of System Calls

- Process control
 - `end()`, `abort()`
 - Halt a running program normally or abnormally
 - Transfer control to invoking command interpreter
 - Memory dump & error message
 - Written to disk and examined by debugger
 - Respond to error: alert window (GUI system) or terminate the entire job (batch system)
 - Error level: normal termination (level 0)

Types of System Calls

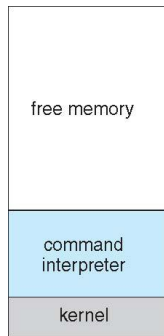
- Process control
 - `end()`, `abort()`
 - `load()`, `execute()`
 - Where to return?
 - Return to existing program: save mem. image
 - Both programs continue concurrently: multiprogram
 - `create_process()`, `terminate_process()`
 - `get_process_attributes()`, `set_process_attributes()`
 - Job's priority, maximum allowable execution time, etc

Types of System Calls

- Process control
 - `end()`, `abort()`
 - `load()`, `execute()`
 - `create_process()`, `terminate_process()`
 - `get_process_attributes()`, `set_process_attributes()`
 - `wait_time()`
 - `wait_event()`, `signal_event()`
 - `acquire_lock()`, `release_lock()`

Example of Process Control: MS-DOS

- Single-tasking
- Shell invoked when system booted
- Simple method to run program
 - No process created
- Single memory space
- Loads program into memory, overwriting all but the kernel
- Program exit -> shell reloaded



(a)

At system startup

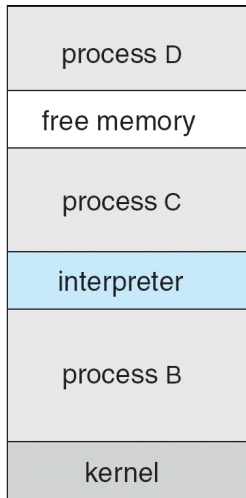


(b)

running a program

Example of Process Control: FreeBSD

- Unix variant
- Multitasking
- User login -> invoke user's choice of shell
- Shell executes `fork()` system call to create process
 - Executes `exec()` to load program into process
 - Shell waits for process to terminate or continues with user commands
- Process exits with:
 - code = 0 – no error
 - code > 0 – error code



Types of System Calls

- File management
 - create file, delete file
 - open, close file
 - read, write, reposition
 - get and set file attributes
- Device management: physical/virtual devices
 - request device, release device
 - read, write, reposition
 - get device attributes, set device attributes
 - logically attach or detach devices

Types of System Calls

- Information maintenance
 - Get time or date, set time or date
 - Get system data, set system data
 - Num. of current users, os version, amount of free mem. & disk
 - Debugging
 - Dump memory
 - Single-step execution
 - Time profile: timer interrupt
 - The amount of time that the program executes at a particular location

Types of System Calls

- Communications

- Message-passing model

- Host name, IP, process name
 - `Get_hostid()`, `get_processid()`, `open_connection()`, `close_connection()`, `accept_connection()`, `read_message()`, `write_message()`
 - Useful for exchanging smaller amounts of data

- Shared-memory model

- Remove the normal restriction of preventing one process from accessing another process's memory
 - Create and gain access to shared mem. region
 - `shared_memory_create()`, `shared_memory_attach()`
 - Threads: memory is shared by default
 - Efficient and convenient, having protection and synchronization issues

Types of System Calls

- Protection
 - Control access to resources
 - All computer systems must be concerned
 - Permission setting
 - `get_permission()`, `set_permission()`
 - Allow/deny access to certain resources
 - `allow_user()`, `deny_user()`

Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

<https://www.kernel.org/doc/man-pages/>
<http://man7.org/linux/man-pages/>

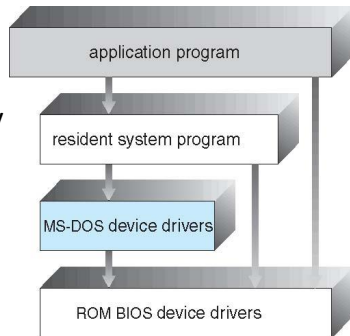
Operating System Structures

Operating System Structure

- General-purpose OS is a very large program
- Various ways to structure ones
 - Simple structure – MS-DOS
 - Monolithic-- UNIX
 - Layered – an abstraction
 - Microkernel –Mach
 - Modules
 - Hybrid system – most OSes

Simple Structure -- MS-DOS

- MS-DOS – written to provide the most functionality in the **least space**
 - Do not have well-defined structures
 - Not divided into modules
 - Its interfaces and levels of functionality are not well separated
 - Application programs can access basic I/O routines
 - Vulnerable to errant programs
 - Limited by hardware



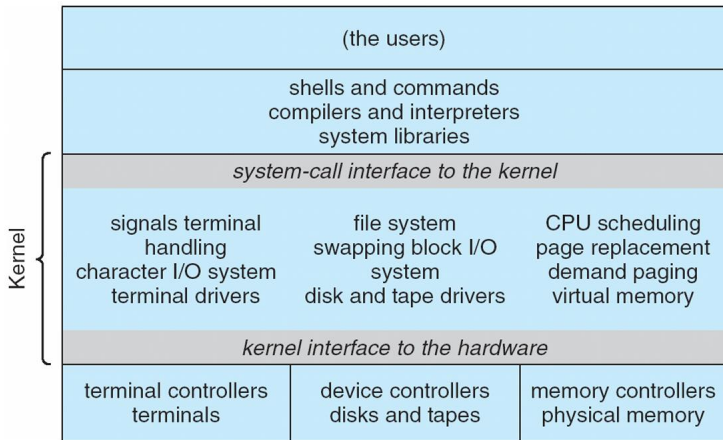
Monolithic Structure -- UNIX

- UNIX

- The original UNIX operating system had **limited structuring**, it consists of **two separable parts**
 - Systems programs
 - The kernel
 - Consists of **everything** below the system-call interface and above the physical hardware
 - A series of interfaces and device drivers
- **Monolithic structure**: combine all functionality in one level
 - File system, CPU scheduling, memory management, and other operating-system functions
 - Difficult to implement and maintain
 - Performance advantage

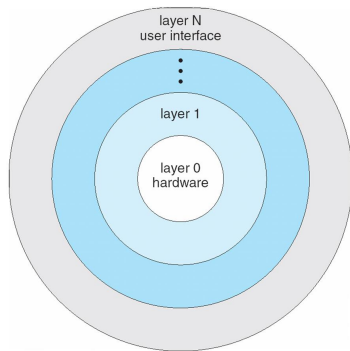
Traditional UNIX System Structure

- Beyond simple but not fully layered



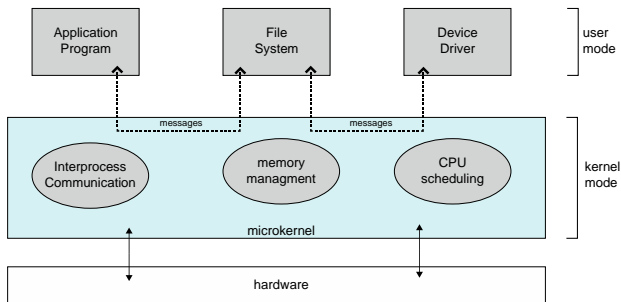
Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers
 - The bottom layer (0), is the hardware; the highest layer (N) is the user interface
- Implementation
 - Each layer is an implementation of an abstract object made up of data and operations
- Advantages
 - Simple to construct and debug
 - Hides the existence of DS, Ops, hardware from upper layers
- Challenges
 - How to define various layers?
 - Efficiency problem
 - I/O->memory manage->CPU scheduling->hardware



Microkernel System Structure

- Moves as much from the kernel into user space as possible
 - Provides minimal process, memory management and communication
 - **Mach**: example of **microkernel** (developed by CMU in mid-1980s)
 - Mac OS X kernel (**Darwin**) partly based on Mach
- Main function
 - Communication between client program and services (also in user space)
 - Provided through **message passing**

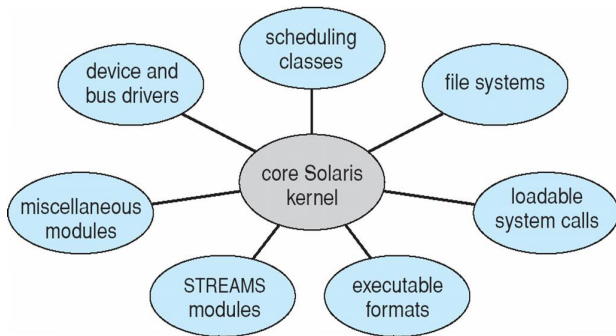


Microkernel System Structure

- Moves as much from the kernel into user space as possible
 - Provides minimal process, memory management and communication
 - **Mach**: example of **microkernel** (developed by CMU in mid-1980s)
 - Mac OS X kernel (**Darwin**) partly based on Mach
- Main function
 - Communication between client program and services (also in user space)
 - Provided through **message passing**
- Benefits
 - Easier to extend a microkernel: add services to user space, no changes to kernel
 - Easier to port the operating system to new architectures
 - More reliable & more secure (less code is running in kernel mode)
- Detriments
 - Performance overhead of user space to kernel space communication

Modules

- Many modern operating systems implement **loadable kernel modules**
 - The Kernel has a set of core components
 - Links in additional services via modules (boot time or run time)
 - Common in most modern OSes



Modules

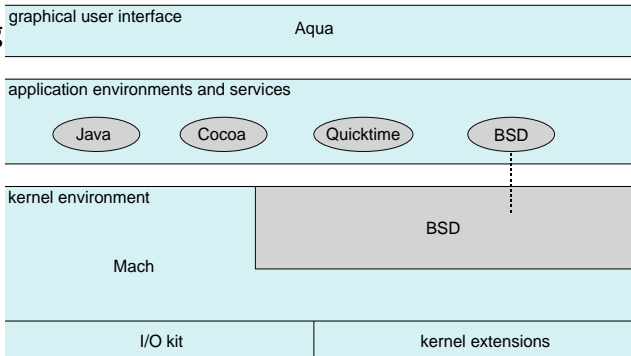
- Many modern operating systems implement **loadable kernel modules**
 - The Kernel has a set of core components
 - Links in additional services via modules (boot time or run time)
 - Common in most modern OSes
- Similar to layered system
 - Any module can call any other model
 - **More flexible**
- Similar to the microkernel
 - Primary module has only core functions
 - No need to invoke message passing
 - **More efficient**

Hybrid Systems

- Most modern operating systems combine different structures, resulting in **hybrid systems**
 - Why? Address performance, security, usability needs
- Examples
 - Linux kernel
 - Monolithic: single address space (**for efficient performance**)
 - Modular: dynamic loading of functionality
 - Windows
 - Mostly monolithic, plus microkernel for different subsystem personalities (running in user-mode), also support loadable kernel module
 - Apple Mac OS X
 - Mach microkernel, BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**)

Mac OS X Structure

- Layered system: user interface + application environment & services + kernel (Mach+BSD UNIX)
- Mach Microkernel
 - Memory management
 - inter-process communication
 - Thread scheduling
- BSD UNIX
 - CLI
 - POSIX API
 - Networking
 - File system



iOS

- Apple mobile OS for *iPhone, iPad*
 - Structured on Mac OS X
 - Added functionality
 - Does not run OS X applications natively
 - Also runs on different CPU architecture (ARM vs. Intel)

Cocoa Touch

Media Services

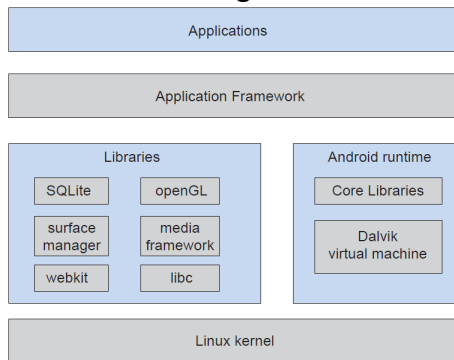
Core Services

Core OS

- Structure
 - **Cocoa Touch** is Objective-C API for developing apps
 - **Media services** layer for graphics, audio, video
 - **Core services** provides cloud computing, databases
 - **Core operating system**, based on Mac OS X kernel

Android

- Developed by Open Handset Alliance (mostly Google)
 - Similar stack to IOS
 - Open Source
- Based on Linux kernel
 - Provides process, memory, device-driver management
- Optimization
 - Adds power management



Operating System Design and Implementation

Operating System Design and Implementation

- Design and Implementation of OS not “solvable”, but some approaches have proven successful
- First problem: **Design goals and specifications**
 - Affected by choice of hardware, type of system (batch, time-sharing, single/multiple users, distributed, real-time, etc)
 - User goals
 - Convenient to use, easy to learn, reliable, safe, and fast
 - System goals
 - Easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
 - **No unique solution to the problem of defining the requirements**

Operating System Design and Implementation

- Important principle to separate
 - **Mechanism**: *How* to do it?
 - **Policy**: *What* will be done?
- Examples
 - Timer mechanism (for CPU protection)
 - Policy decision: How long the timer is to be set?
 - Priority mechanism (in job scheduling)
 - Policy: I/O-intensive programs have higher priority than CPU-intensive ones or vice versa
- Benefits: maximum flexibility
 - Change policy without changing mechanism

OS Implementation

- Much variation
 - Early OSes in assembly language
 - Now C, C++
- Actually usually a mix of languages
 - Main body in C
 - Lowest levels in assembly
 - Systems programs in C, C++, scripting languages
- Pros and cons
 - Code can be written faster, easier to understand/debug
 - More high-level language, easier to **port** to other hardware
 - Slower & increased storage requirement

Implementation

- Performance?
 - Major performance improvements: better data structures and algorithms
 - How about developing excellent assembly-language code in OS implementation?
 - Modern compiler is well optimized
 - A small amount of the code is critical to performance, easy to do specialized optimization
 - Interrupt handler
 - I/O manager
 - Memory manager
 - CPU scheduler

MISC

Debugging, Generation, Booting

Operating-System Debugging

- **Failure analysis**
 - **log files**: written with error information when process fails
 - **core dump**: a capture of the memory of the processes
 - **crash dump**: memory state when OS crashes
- **Performance tuning**
 - ***Trace listings*** of system behavior
 - **Interactive tools**: top displays resource usage of processes
- Kernighan's Law
 - “**Debugging is twice as hard as writing the code in the first place.** Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

Operating System Generation

- Operating systems are designed to run on any of **a class of machines**
 - The system must be **configured or generated** for each specific computer site
- **SYSGEN** program obtains information concerning the specific configuration of the hardware system
 - Read from file, ask the operator or probe
 - Generation methods
 - Modify source code and completely recompile
 - Select modules from precompiled library and link together

System Boot

- System booting on most computer systems
 - Bootstrap program (residing in ROM) locates the kernel, loads it into memory, and starts it
 - ROM needs no initialization, cannot be easily infected by virus
 - Diagnostics to determine machine state
 - Initialization: CPU registers, device controllers, memory
 - Some use two-step process: a simple bootstrap loader fetches a more complex bootstrap program, which loads kernel (large OSES)
 - Some store the entire OS in ROM (Mobile OS)
- Common bootstrap loader allows selection of kernel from multiple disks, versions, kernel options (**GRUB**)

Summary

- Operating system services
- System calls
 - Relationship between system call and API
- Operating system structures
 - Modular is important
 - Generally adopt a hybrid approach
- Design principles
 - Separate policy from mechanism

Summary of Part I (Ch1 & Ch2)

- | | |
|---|--|
| <ul style="list-style-type: none">• OS Overview<ul style="list-style-type: none">– OS Functionality– Multiprogramming & Multitasking• OS Operations<ul style="list-style-type: none">– Dual Mode & System Call• OS Components<ul style="list-style-type: none">– Process Management– Memory Management– Storage Management• Computing Environment | <ul style="list-style-type: none">• Ch2 OS Structure<ul style="list-style-type: none">– Operating system services– System calls– Operating system structures– Design principles• Process management<ul style="list-style-type: none">– Concept, scheduling, operation, communication, synchronization• Memory management<ul style="list-style-type: none">– Main memory, virtual mem• Storage management<ul style="list-style-type: none">– Storage, FS, I/O |
|---|--|

End of Chapter 2

Operating Systems

Associate Prof. Yongkun Li

中科大-计算机学院 副教授

<http://staff.ustc.edu.cn/~ykli>

Chapter 3 Process Concepts & Operations

Outline

- Process Concept
 - Program vs process
 - Process in memory & PCB
 - Process state
- Processes Operations
 - Process creation, program execution, process termination
 - UNIX example: `fork()`, `exec*()`, `wait()`

What is a process?

Informally, a process is a program in execution.



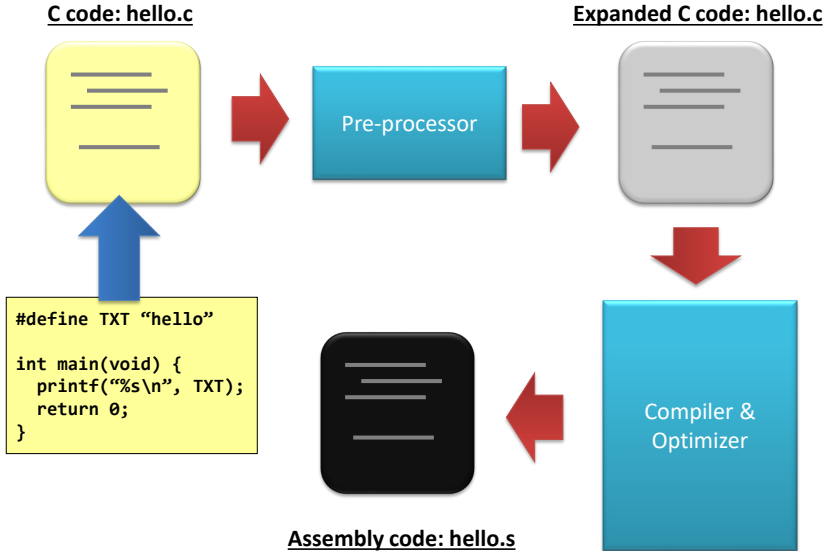
What is a program?



What is a program?

- What is a program?
 - A program is a just **a piece of code**.
- But, which code do you mean?
 - High-level language code: C or C++?
 - Low-level language code: assembly code?
 - Not-yet an executable: object code?
 - Executable: machine code?

Flow of building a program (1 of 2)



(Still...1 of 2) Pre-processor

- The pre-processor expands:
 - **#define**, **#include**, **#ifdef**, **#ifndef**, **#endif**, etc.
 - Try: **"gcc -E hello.c"**



(Still...1 of 2) Pre-processor

- Another example: **the macro!**

```
#define SWAP(a,b) { int c; c = a; a = b; b = c; }
```

```
int main(void) {  
    int i = 10, j = 20;  
    printf("before swap: i = %d, j = %d\n", i, j);  
    SWAP(i, j);  
    printf("after swap: i = %d, j = %d\n", i, j);  
}
```



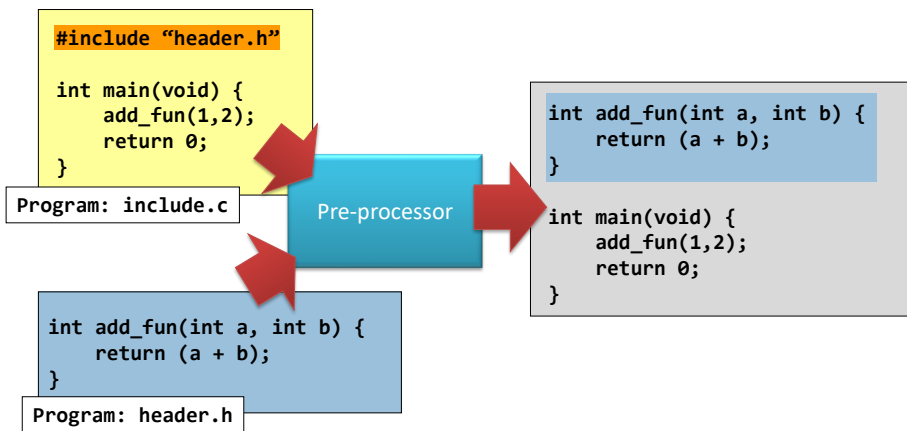
Pre-processor



```
int main(void) {  
    int i = 10, j = 20;  
    printf("before swap: i = %d, j = %d\n", i, j);  
    { int c; c = i; i = j; j = c; };  
    printf("after swap: i = %d, j = %d\n", i, j);  
}
```


(Still...1 of 2) Pre-processor

- How about: #include?



(Still...1 of 2) Compiler and Optimizer

- The compiler performs:
 - Syntax checking and analyzing;
 - If there is no syntax error, construct intermediate codes, i.e., assembly codes;
- The optimizer optimizes codes
 - **It improves stupid codes!**
 - Check the parameter of gcc

“-0” means to optimize.

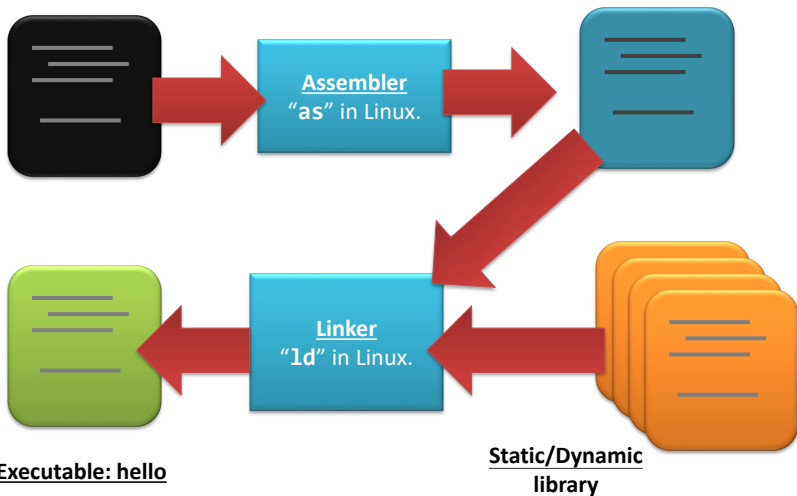
The number followed is the optimization level. Max is level 3, i.e., “-03”. Default is level is “-01”.

“-00”: means no optimization.

Flow of building a program (2 of 2)

Assembly code: hello.s

Object code: hello.o

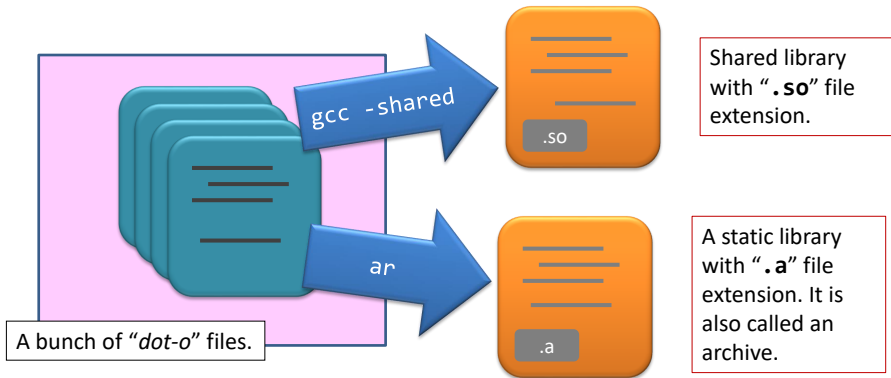


(Still...2 of 2) Assembler and Linker

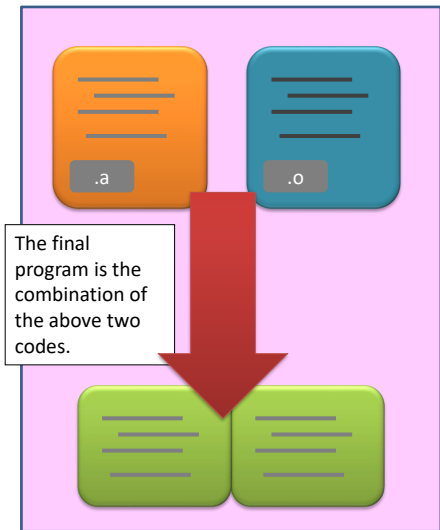
- The assembler assembles “**hello.s**” and generates an object code “**hello.o**”
 - A step closer to machine code
 - Try: “**as hello.s -o hello.o**”
- The linker puts together all object files as well as the libraries
 - There are two kinds of libraries: **statically-linked** and **dynamically-linked** ones

Sidetrack: Library files

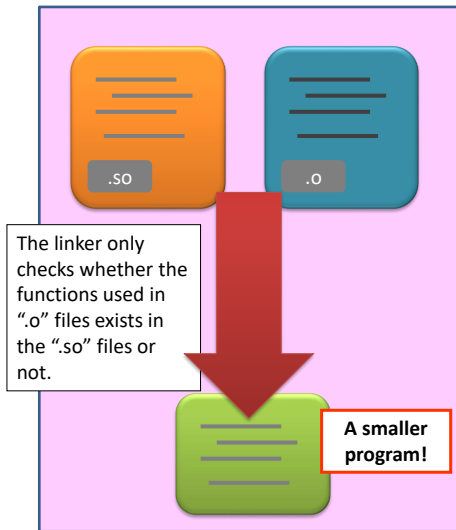
- A library file is...
 - just a bunch of function implementations.
 - for the linker to look for the function(s) that the target C program needs.



Sidetrack: Library files



Linking with static library file.



Linking with dynamic library file.

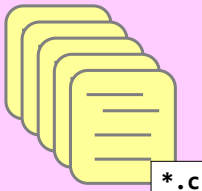
How to compile multiple files?

- **gcc** by default hides all the intermediate steps.
 - Executable: “**gcc -o hello hello.c**” generates “**hello**” directly.
 - Object code: “**gcc -c hello.c**” generates “**hello.o**” directly.
- How about working with multiple files?

How to compile multiple files?

Remember, below shows one of the solution.

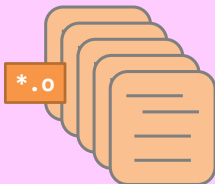
Step 1.



Prepare all the source files.
Important: there must be one and only one file containing the main function.

Step 2.

```
$ gcc -c code.c  
.....
```



Compile them into object codes one by one.

Step 3.

```
$ gcc -o prog *.o
```



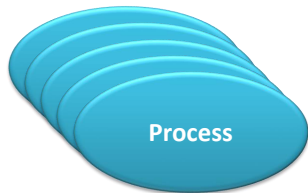
prog

Construct the program together with all the object codes.

Conclusion on “*what is a program?*”

- A program is just an executable file!
 - It is static;
 - It may be associated with dynamically-linked files;
 - “*.so” in Linux and “*.dll” in Windows.
- It may be compiled from more than one file

What is a process?

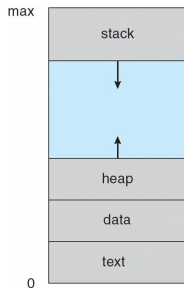


Process in Memory

- A process is a program in execution
 - A program (an executable file) becomes process when it is **loaded into memory**
 - **Active**
- Process in memory
 - Text section
 - Stack
 - Heap
 - Data section
 - Program counter
 - Contents of registers

Process in Memory

- Text section
 - Program code
- Data section
 - Global variables
- Stack
 - Temporary data (function parameters, return addresses, local variables)
- Heap
 - Dynamically allocated memory during process run time
- Program counter and contents of registers

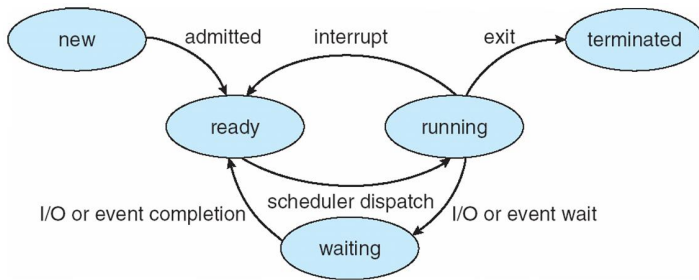


Process State

- As a process executes, it changes **state**, which is defined in part by the **current activity**
 - **new**: The process is being created
 - **running**: Instructions are being executed
 - **waiting**: The process is waiting for some event to occur
 - I/O completion or reception of a signal
 - **ready**: The process is waiting to be assigned to a processor
 - **terminated**: The process has finished execution

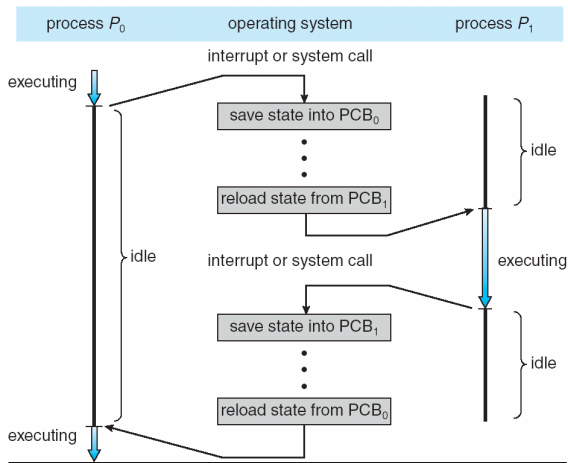
Diagram of Process State

- State diagram



- Only one process can be running on any processor at any instant
- Many processes may be ready or waiting

How to switch processes?



Example: CPU switch from process to process

How to locate/represent a process?

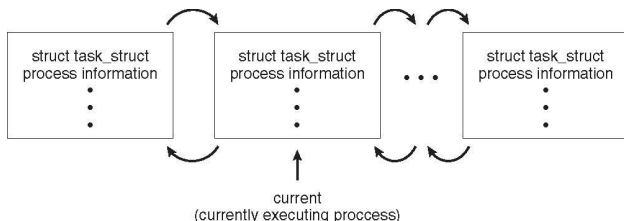
- Process control block (PCB) or task control block
 - Process state (running, waiting, etc)
 - Program counter
 - location of next instruction to execute
 - CPU registers
 - contents of all process-centric registers
 - CPU scheduling information
 - priorities, scheduling queue pointers
 - Memory-management information
 - memory allocated to the process
 - I/O status information
 - I/O devices allocated to process, list of open files
 - Accounting information
 - CPU used, clock time elapsed since start, time limits



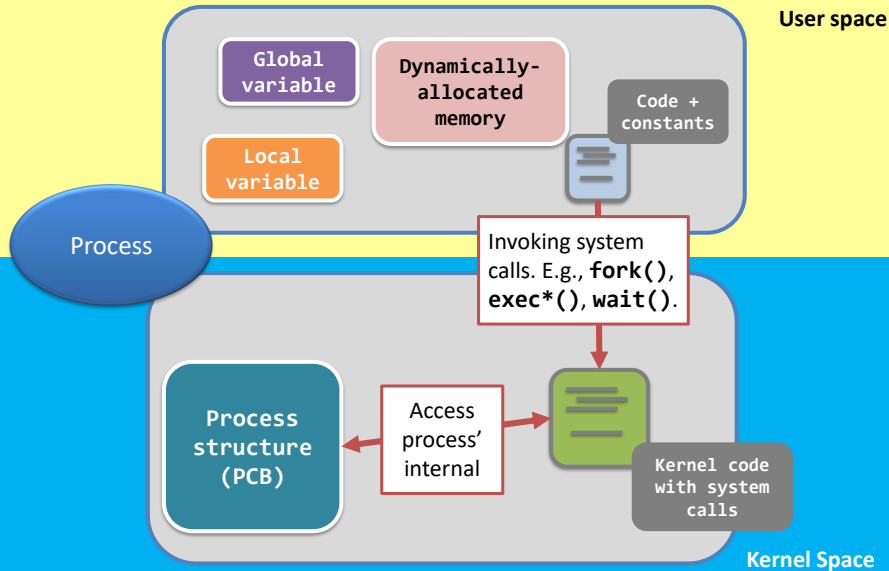
Process Data Structure in Linux

- Represented by C structure `task_struct`
 - `<linux/sched.h>`

```
pid t_pid; /* process identifier */
long state; /* state of the process */
struct sched_entity se; /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



Relationship between Process Data & PCB



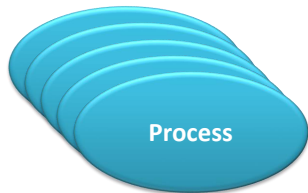
Conclusion on “*what is a process?*”

- A process is a program **in execution**
 - process (active entity) != program (static entity)
 - Why active?
 - A program counter specifying the next instruction to execute + a set of associated resources
- Only one process can be running on any processor at any instant

Conclusion on “*what is a process?*”

- Two processes maybe associated with the same program (Two users are running the same program)
 - Example
 - The same user invokes two copies of the web browser
 - Separate execution sequences
 - The text section may be equivalent
 - The data, heap, and stack sections vary
- A process can be an execution environment for other code
 - Java programming environment
 - java Program (java runs JVM as a process)

Process Operations



Process Operations

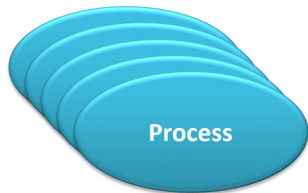
- Process
 - It associates with all the files opened by that process.
 - It attaches to all the memory that is allocated for it.
 - It contains every accounting information,
 - running time, current memory usage, who owns the process, etc.
- You couldn't operate any things without processes.

Process Operations

- System must provide mechanisms for:
 - process **identification**
 - process **creation**
 - program **execution**
 - process **termination**
- Some basic and important system calls
 - **getpid()**
 - **fork()**
 - **exec*()**
 - **wait()**
 - **exit()**

Process Operations

- process identification



Process identification

- How can we identify processes?
 - Each process is given an unique ID number, and is called the **process ID**, or the **PID**.
 - The system call, **getpid()**, prints the PID of the calling process.

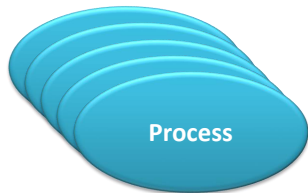
```
#include <stdio.h>    // printf()
#include <unistd.h>    // getpid()

int main(void) {
    printf("My PID is %d\n", getpid() );
}
```

```
$ ./getpid
My PID is 1234
$ ./getpid
My PID is 1235
$ ./getpid
My PID is 1237
```

Process Operations

- process identification
- process creation



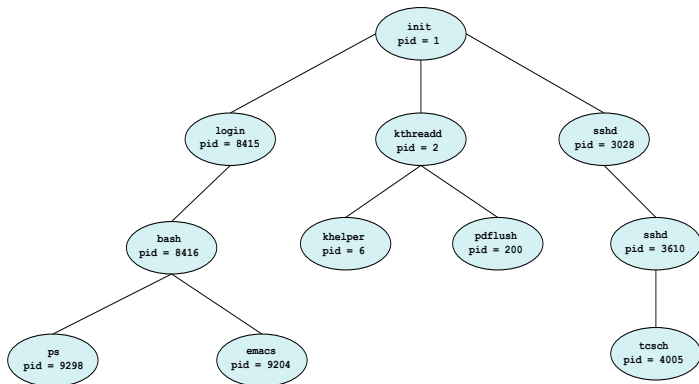
Process Creation

- A process may create several new processes
 - **Parent** process: the creating process
 - **Children** processes: the new processes
- The first process
 - The kernel, while it is booting up, creates the first process – **init**.
 - The “**init**” process:
 - has **PID = 1**, and
 - is running the program code “**/sbin/init**”.
 - Its first task is to **create more processes...**

Process Creation

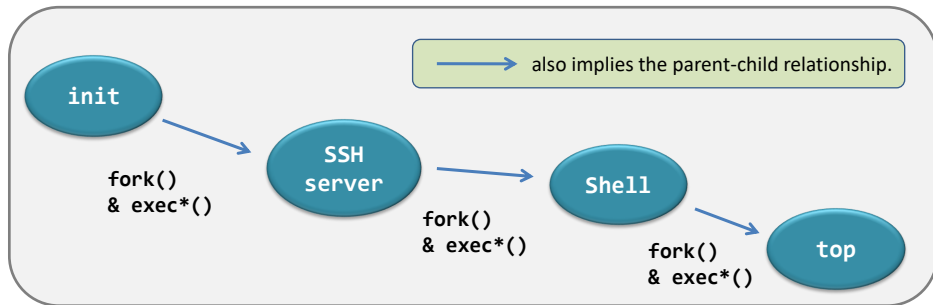
- Tree hierarchy

- Each of the new process may in turn create other processes, and form a tree hierarchy



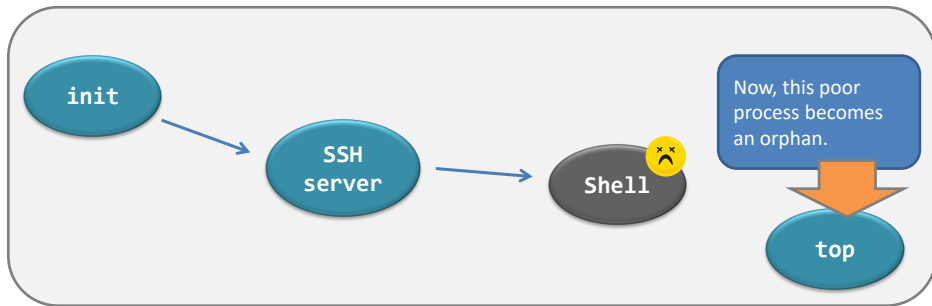
Process blossoming

- You can view the tree with the command:
 - “**ps tree**”; or
 - “**ps tree -A**” for ASCII-character-only display.



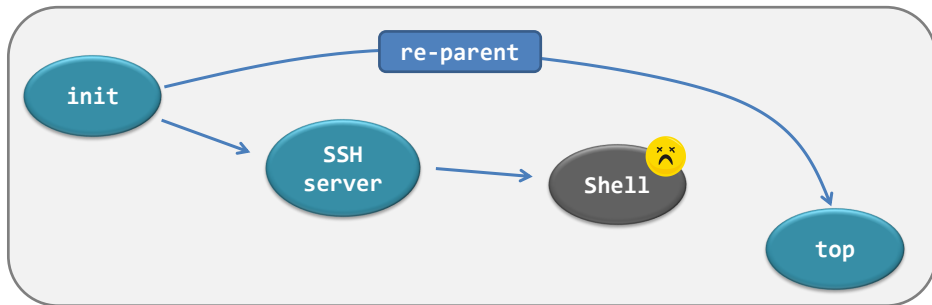
Process blossoming...with orphans?

- However, termination can happen, at any time and in any place...
 - All the resources are deallocated to OS when a process terminates
 - A process may become an orphan when its parent terminated
 - An orphan turns the hierarchy from a **tree** into a **forest**!
 - Plus, no one would know the termination of the orphan.



Process blossoming...with re-parent!

- In Linux...
 - We have the **re-parent operation**.
 - The “**init**” process will become the step-mother of all orphans.
- Well...Windows maintains a *forest-like* hierarchy.



A short summary

- **Observation 1**

- The processes in Linux is always organized as a tree.
- Because of the re-parent operation, there is always **only one process tree**.

- **Observation 2**

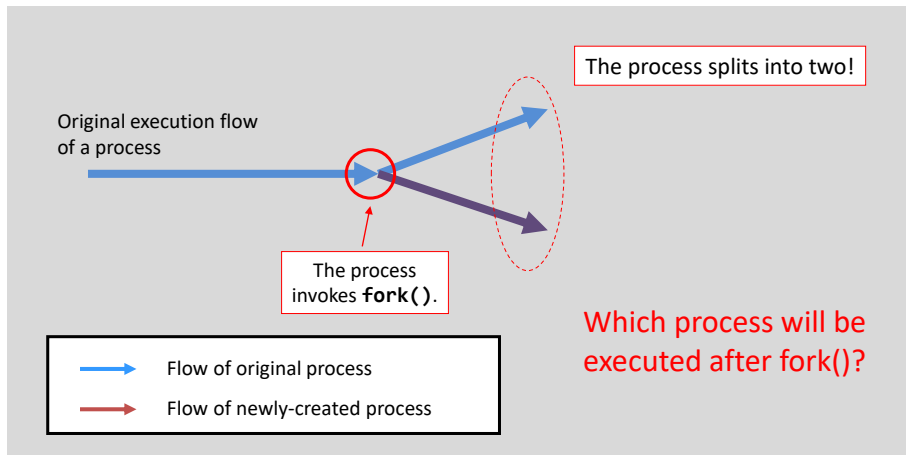
- The re-parent operation allows processes running **without the need of a parent terminal**.
- Thus, the **background jobs** survive even though the hosting terminal is closed.

Relationship between Parent and Child

- Resource sharing options
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- Execution options
 - Parent and children execute concurrently
 - Parent waits until children terminate
- Address space options
 - Child is a duplicate of parent
 - Child has a new program loaded into it
- We focus on UNIX examples to illustrate

Process creation

- To create a process, we use the system call **fork()**



Process creation – **fork()** system call

- So, how do **fork()** and the processes behave?

```
$ ./fork_example_1
Ready (PID=1234)
My PID is 1234
My PID is 1235
$ _
```

PID 1234

Process 1234 is the original process, and we call it the **parent process**.

PID 1235

```
int main(void) {
    printf("Ready (PID = %d)\n", getpid());
    fork();
    printf("My PID is %d\n", getpid() );
    return 0;
}
```

Process 1235 is created by the **fork()** system call, and we call it the **child process**.

Why is this line of code executed twice?

Process creation – **fork()** system call

- So, how do **fork()** and the processes behave?


```
int main(void) {  
    printf("Ready (PID = %d)\n", getpid());  
    fork();  
    printf("My PID is %d\n", getpid() );  
    return 0;  
}
```

What do we know so far?

- Both the parent and the child execute **the same program before and after fork()**.
- The child process starts its execution **at the location that fork() is returned**, *not from the beginning of the program*.

Process creation – **fork()** system call

One more example




```
1 int main(void) {
2     int result;
3     printf("before fork ...\n");
4     result = fork();
5     printf("result = %d.\n", result);
6
7     if(result == 0) {
8         printf("I'm the child.\n");
9         printf("My PID is %d\n", getpid());
10    }
11    else {
12        printf("I'm the parent.\n");
13        printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17 }
```

```
$ ./fork_example_2
before fork ...
```

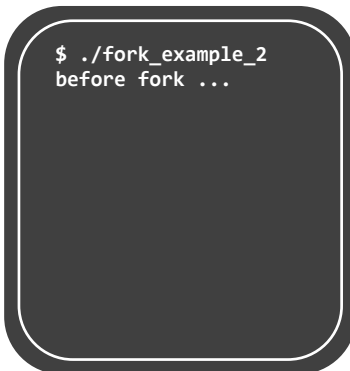
PID 1234

Process creation – **fork()** system call

One more example



```
1 int main(void) {
2     int result;
3     printf("before fork ...\n");
4     result = fork();
5     printf("result = %d.\n", result);
6
7     if(result == 0) {
8         printf("I'm the child.\n");
9         printf("My PID is %d\n", getpid());
10    }
11    else {
12        printf("I'm the parent.\n");
13        printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17 }
```



```
$ ./fork_example_2
before fork ...
```

PID 1234

fork()

PID 1235

Process creation – **fork()** system call


Assumption

Let there be only **ONE CPU**. Then...

- Only one process is allowed to be executed at one time.
- However, we can't predict which process will be chosen by the OS.
- By the time, this mechanism is called **process scheduling**.

In this example, we assume that the parent, PID 1234, runs first, after the **fork()** call.

Process creation – **fork()** system call



```
1 int main(void) {
2     int result;
3     printf("before fork ...\n");
4     result = fork();
5     printf("result = %d.\n", result);
6
7     if(result == 0) {
8         printf("I'm the child.\n");
9         printf("My PID is %d\n", getpid());
10    }
11    else {
12        printf("I'm the parent.\n");
13        printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17 }
```

```
$ ./fork_example_2
before fork ...
result = 1235
```

Important

For parent, the return value of **fork()** is the PID of the created child.

PID 1234
(running)

PID 1235
(waiting)

Process creation – **fork()** system call

```
1 int main(void) {  
2     int result;  
3     printf("before fork ...\n");  
4     result = fork();  
5     printf("result = %d.\n", result);  
6  
7     if(result == 0) {  
8         printf("I'm the child.\n");  
9         printf("My PID is %d\n", getpid());  
10    }  
11    else {  
12        printf("I'm the parent.\n");  
13        printf("My PID is %d\n", getpid());  
14    }  
15  
16    printf("program terminated.\n");  
17 }
```

```
$ ./fork_example_2  
before fork ...  
result = 1235  
I'm the parent.  
My PID is 1234  
program terminated.
```

PID 1234
(dead)



PID 1235
(waiting)

Process creation – **fork()** system call

```
1  int main(void) {  
2      int result;  
3      printf("before fork ...\n");  
4      result = fork();  
5      printf("result = %d.\n", result);  
6  
7      if(result == 0) {  
8          printf("I'm the child.\n");  
9          printf("My PID is %d\n", getpid());  
10     }  
11     else {  
12         printf("I'm the parent.\n");  
13         printf("My PID is %d\n", getpid());  
14     }  
15  
16     printf("program terminated.\n");  
17 }
```

```
$ ./fork_example_2  
before fork ...  
result = 1235  
I'm the parent.  
My PID is 1234  
program terminated.  
result = 0
```

Important

For child, the return value of **fork()** is 0.

PID 1234
(dead)



PID 1235
(running)

Process creation – **fork()** system call

```
1 int main(void) {  
2     int result;  
3     printf("before fork ...\n");  
4     result = fork();  
5     printf("result = %d.\n", result);  
6  
7     if(result == 0) {  
8         printf("I'm the child.\n");  
9         printf("My PID is %d\n", getpid());  
10    }  
11    else {  
12        printf("I'm the parent.\n");  
13        printf("My PID is %d\n", getpid());  
14    }  
15  
16    printf("program terminated.\n");  
17 }
```

```
$ ./fork_example_2  
before fork ...  
result = 1235  
I'm the parent.  
My PID is 1234  
program terminated.  
result = 0  
I'm the child.  
My PID is 1235  
program terminated.  
$ _
```

PID 1234
(dead)



PID 1235
(dead)



Process creation – **fork()** system call

- **fork()** behaves like “*cell division*”.
 - It creates the child process by **cloning** from the parent process, including...

Cloned items	Descriptions
Program code [File & Memory]	They are sharing the same piece of code.
Memory	Including local variables, global variables, and dynamically allocated memory.
Opened files [Kernel's internal]	If the parent has opened a file “A”, then the child will also have file “A” opened automatically.
Program counter [CPU register]	That's why they both execute from the same line of code after fork() returns.

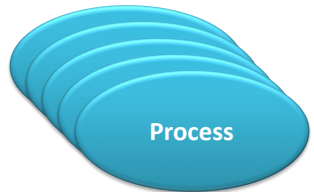
Process creation – **fork()** system call

- However...
 - **fork()** does not clone the following...
 - Note: they are all data inside the memory of kernel.

Distinct items	Parent	Child
Return value of fork()	PID of the child process.	0
PID	Unchanged.	Different, not necessarily be "Parent PID + 1"
Parent process	Unchanged.	Doesn't have the same parent as that of the parent process.
Running time	Cumulated.	Just created, so should be 0.

Process Operations

- process identification
- process creation
- **program execution**



`fork()` can only duplicate...

- `fork()` is rather **boring**...
 - If a process can only duplicate itself and always runs the same program, then...
 - how can we execute other programs?
- We want **CHANGE!**
 - Meet the **`exec()`** system call family.

Program execution

- **exec1()** – a member of the **exec** system call family (and the family has 6 members).

```
int main(void) {  
    printf("before exec1 ...\n");  
    exec1("/bin/ls", "/bin/ls", NULL);  
    printf("after exec1 ...\n");  
    return 0;  
}
```

```
$ ./exec_example  
before exec1 ...
```

Arguments of the exec1() call

1st argument: the program name, **"/bin/ls"** in the example.

2nd argument: 1st argument to the program.

3rd argument: indicate the end of the list of arguments.

Program execution

- Example #1: run the command **"/bin/ls"**

```
exec1("/bin/ls", "/bin/ls", NULL);
```

Argument Order	Value in above example	Description
1	"/bin/ls"	The file that the programmer wants to execute.
2	"/bin/ls"	When the process switches to "/bin/ls" , this string is the first program argument .
3	NULL	This states the end of the program argument list.

Program execution


- Example #2: run the command **"/bin/ls -l"**

```
exec1("/bin/ls", "/bin/ls", "-l", NULL);
```

Argument Order	Value in above example	Description
1	"/bin/ls"	The file that the programmer wants to execute.
2	"/bin/ls"	When the process switches to "/bin/ls" , this string is the first program argument .
3	"-l"	When the process switches to "/bin/ls" , this string is the second program argument .
4	NULL	This states the end of the program argument list.

Program execution

- **exec1()** – a member of the **exec** system call family (and the family has 6 members).

```
int main(void) {  
    printf("before exec1 ...\n");  
     exec1("/bin/ls", "/bin/ls", NULL);  
    printf("after exec1 ...\n");  
    return 0;  
}
```


```
$ ./exec_example  
before exec1 ...
```

What is the output?

The same as the output of running "ls" in the shell.

Program execution


- **exec1()** – a member of the **exec** system call family (and the family has 6 members).

```
int main(void) {  
    printf("before exec1 ...\n");  
     exec1("/bin/ls", "/bin/ls", NULL);  
    printf("after exec1 ...\n");  
    return 0;  
}
```

```
$ ./exec_example  
before exec1 ...  
exec_example  
exec_example.c
```

Program execution

- **exec1()** – a member of the **exec** system call family (and the family has 6 members).

```
int main(void) {  
    printf("before exec1 ...\n");  
     exec1("/bin/ls", "/bin/ls", NULL);  
    printf("after exec1 ...\n");  
    return 0;  
}
```

```
$ ./exec_example  
before exec1 ...  
exec_example  
exec_example.c
```

GUESS:
What happens next?

Program execution

- **exec1()** – a member of the **exec** system call family (and the family has 6 members).

```
int main(void) {  
    printf("before exec1 ...\n");  
    exec1("/bin/ls", "/bin/ls", NULL);  
    printf("after exec1 ...\n");  
    return 0;  
}
```

WHAT?!
The shell prompt appears!

```
$ ./exec_example  
before exec1 ...  
exec_example  
exec_example.c  
$ _
```

The output says:

- (1) The gray code block **is not reached!**
- (2) The process is **terminated!**

WHY IS THAT?!

Program execution

- The **exec** system call family is not simply a function that “invokes” a command.

```
int main(void) {  
    printf("before execl ...\n");  
    execl("/bin/ls", "/bin/ls", NULL);  
    printf("after execl ...\n");  
    return 0;  
}
```

Originally, the process is executing the program “**exec_example**”.



Program execution

- The **exec** system call family is not simply a function that “invokes” a command.

```
int main(void) {  
    printf("before execl ...\n");  
    execl("/bin/ls", "/bin/ls", NULL);  
    printf("after execl ...\n");  
    return 0;  
}
```

The execl() call changes the execution from “**exec_example**” to “**/bin/ls**”

```
/* The program “ls” */  
int main(int argc, char ** argv)  
{  
    .....  
    exit(0);  
}
```

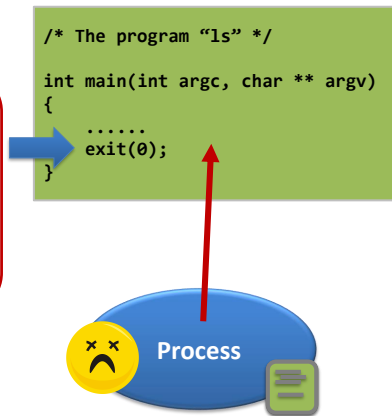


Program execution

- The **exec** system call family is not simply a function that “invokes” a command.

The “**return**” or the “**exit()**” statement in “**/bin/ls**” will terminate the process...

Therefore, it is certain that the process cannot go back to the old program!

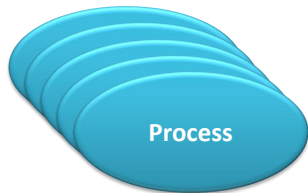


Program execution - observation

- The process is changing the code that is executing and **never returns to the original code.**
 - The last two lines of codes are therefore not executed.
- The process that calls any one of the member of the exec system call family will **throw away** many things, e.g.,
 - Memory: local variables, global variables, and dynamically allocated memory;
 - Register value: e.g., the program counter;
- But, the process will **preserve** something, including:
 - PID;
 - Process relationship;
 - Running time, etc.

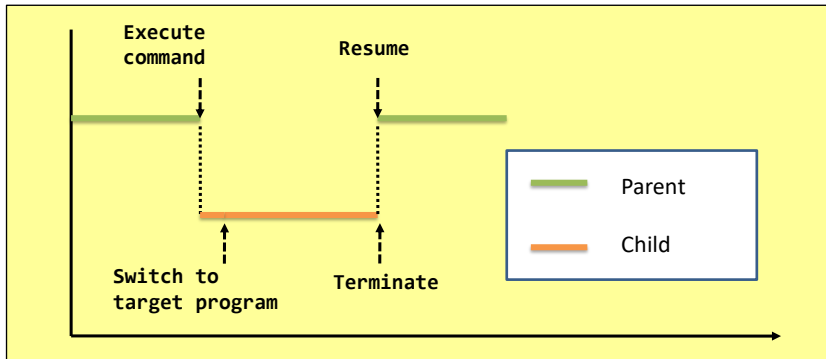
Process Operations

- process identification
- process creation
- program execution
- **`fork()` + `exec*()` = ?**



When **fork()** meets **exec*()**...

- The mix can become:
 - A shell,
 - The **system()** library call, etc...



fork() + exec*() = system()?

```
1 int system_test(const char *cmd_str) {
2     if(cmd_str == -1)
3         return -1;
4     if(fork() == 0) {
5         execl(cmd_str, cmd_str, NULL);
6         fprintf(stderr,
7             "%s: command not found\n", cmd_str);
8         exit(-1);
9     }
10    return 0;
11 }
12 int main(void) {
13     printf("before...\n\n");
14     system_test("/bin/ls");
15     printf("\nafter...\n");
16     return 0;
17 }
```

Is this the
only result?

```
$ ./system_implement_1
before...

system_implement_1
system_implement_1.c

after...
$ _
```

fork() + exec*() = system()?!

```
1 int system_test(const char *cmd_str) {
2     if(cmd_str == -1)
3         return -1;
4     if(fork() == 0) {
5         execl(cmd_str, cmd_str, NULL);
6         fprintf(stderr,
7             "%s: command not found\n", cmd_str);
8         exit(-1);
9     }
10    return 0;
11 }
12 int main(void) {
13     printf("before...\n\n");
14     system_test("/bin/ls");
15     printf("\nafter...\n");
16     return 0;
17 }
```

Some strange cases happened when the program is executed repeatedly!! Why?

```
$ ./system_implement_1
before...

after...
system_implement_1
system_implement_1.c
$ _
```

fork() + exec*() = system()...



```
1 int system_test(const char *cmd_str) {
2     if(cmd_str == -1)
3         return -1;
4     if(fork() == 0) {
5         execl(cmd_str, cmd_str, NULL);
6         fprintf(stderr,
7             "%s: command not found\n", cmd_str);
8         exit(-1);
9     }
10    return 0;
11 }
12 int main(void) {
13     printf("before...\n\n");
14     system_test("/bin/ls");
15     printf("\nafter...\n");
16     return 0;
17 }
```

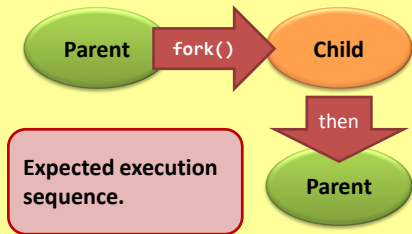
Let's re-color the program!

- Parent process
- Child process
- Both processes

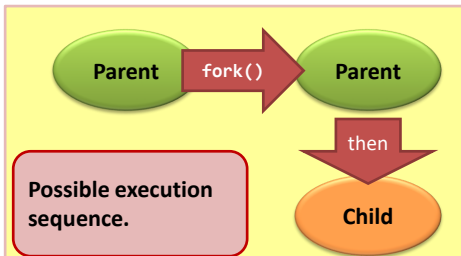
```
$ ./system_implement_1
before...

after...
system_implement_1
system_implement_1.c
$ _
```

`fork()` + `exec*()` = `system()`...



```
$ ./system_implement_1
before...
system_implement_1
System_implement_1.c
after...
$ _
```



```
$ ./system_implement_1
before...
after...
system_implement_1
System_implement_1.c
$ _
```


fork() + exec*()

Is it enough?

`fork()` + `exec*()` = `system()`...



- Don't forget that we're trying to implement a **system()**-compatible function...
 - It is very weird to allow different execution orders.
- How to let the child to execute first?
 - But...we can't control the **process scheduling** of the OS to this extent.
- Then, our problem becomes...
 - How to **suspend** the execution of the parent process?
 - How to **wake** the parent up after the child is terminated?

fork()+ exec*() + wait() = system()

```
1 int system_test(const char *cmd_str) {
2     if(cmd_str == -1)
3         return -1;
4     if(fork() == 0) {
5         execl("/bin/sh", "/bin/sh",
6             "-c", cmd_str, NULL);
7         fprintf(stderr,
8             "%s: command not found\n", cmd_str);
9         exit(-1);
10    }
11    wait(NULL);
12    return 0;
13 }
14
15 int main(void) {
16     printf("before...\n\n");
17     system_test("/bin/ls");
18     printf("\nafter...\n");
19     return 0;
20 }
```

What is the
output now?

fork()+ exec*() + wait() = system()

```
1 int system_test(const char *cmd_str) {
2     if(cmd_str == -1)
3         return -1;
4     if(fork() == 0) {
5         execl("/bin/sh", "/bin/sh",
6             "-c", cmd_str, NULL);
7         fprintf(stderr,
8             "%s: command not found\n", cmd_str);
9         exit(-1);
10    }
11    wait(NULL);
12    return 0;
13 }
14
15 int main(void) {
16     printf("before...\n\n");
17     system_test("/bin/ls");
18     printf("\nafter...\n");
19     return 0;
20 }
```

The parent is
suspended until
the child
terminates

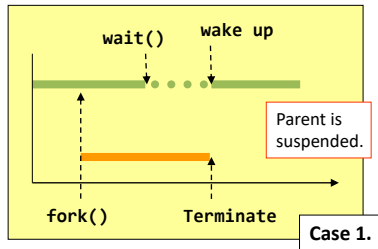
```
$ ./system_implement_2
before...
```

```
system_implement_2
System_implement_2.c
```

```
after...
$ _
```

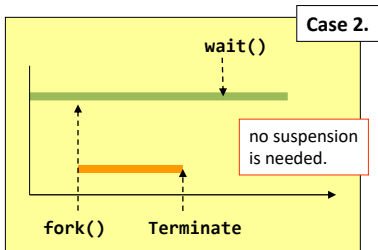
wait() – properties explained

- The **wait()** system call **suspend** the calling parent process (Case 1).
- When to wake up?
 - **wait()** returns and wakes up the calling process when the one of its child processes changes from **running to terminated**.



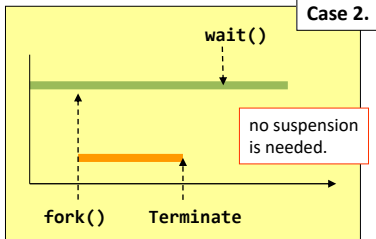
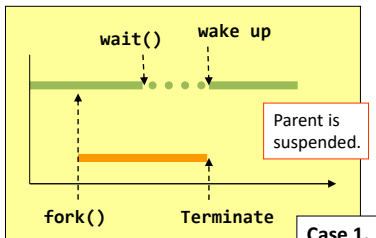
`wait()` – properties explained

- What happens if
 - There were no running children;
 - There were no children;
- `wait()` does not suspend the calling process (Case 2)



wait() – summary

- The **wait()** system call **suspend** the calling parent process (Case 1).
- **wait()** returns and wakes up the calling process when the one of its child processes changes from **running to terminated**.
- **wait()** does not suspend the calling process (Case 2) if
 - There were no running children;
 - There were no children;



More powerful **wait()**?

- Limitation of **wait()**?
 - waits for any one of the children
 - Detect child termination only
- How to wait for a particular process?
 - **waitpid()**

wait() VS waitpid()

wait()	waitpid()
Wait for any one of the children.	Depending on the parameters, waitpid() will wait for a particular child only.
Detect child termination only.	Depending on the parameters, waitpid() <u>can detect child's status changing</u> : -from running to suspended, and -from suspended to running.

For more details, you must read the man pages of **wait()** and **waitpid()**.

Summary of Process Operations

- A process is created by **cloning**
 - **fork()** is the system call that clones processes
 - Cloning is copying
 - What are inherited?
 - What are not?
 - Metaphor of father-son relationship
 - **wait()** can be used to suspend the parent process, so as to guarantee the expected execution sequence
- Program execution is fundamental, but not trivial
 - A process is the place that hosts a program and run it
 - **exec()** system call family changes the program that a process is running.
 - A process can run more than one program...
 - as long as there is a set of programs that keeps on calling the **exec** system call family.

Summary of Ch3

- Concepts
 - Process data in memory
 - PCB
- Operations
 - `fork()`, `exec*()`, `wait()`
 - Just introduced how they could be used to create processes and execute programs
 - How about the internal working of these system calls?
 - How does the kernel behaves when calling these system calls?

End of Chapter 3

Operating Systems

Associate Prof. Yongkun Li

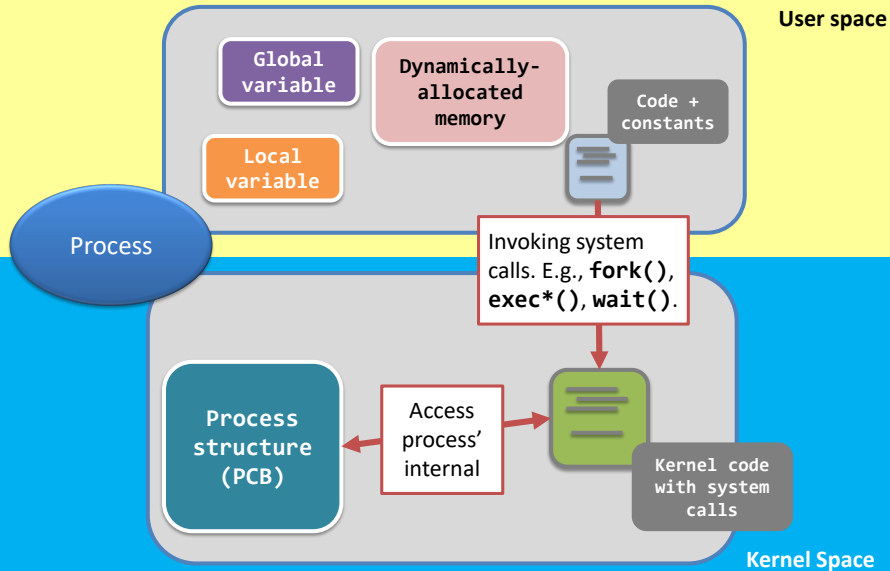
中科大-计算机学院 副教授

<http://staff.ustc.edu.cn/~ykli>

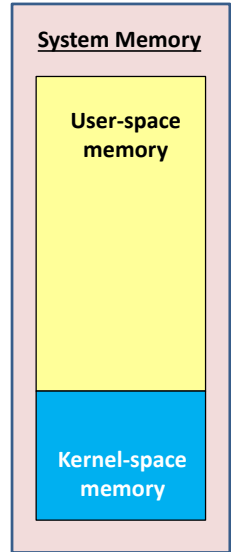
Ch3 - Process Operations

-from kernel's perspective

Process in Memory

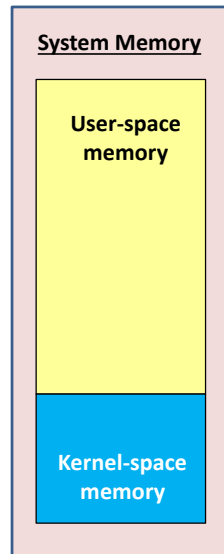


Kernel-space VS User-space



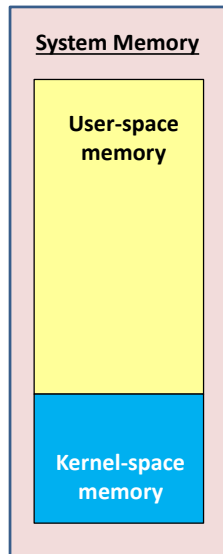
Kernel-space VS User-space

	Kernel-space memory	User-space memory
Storing what		
Accessed by whom		



Kernel-space VS User-space

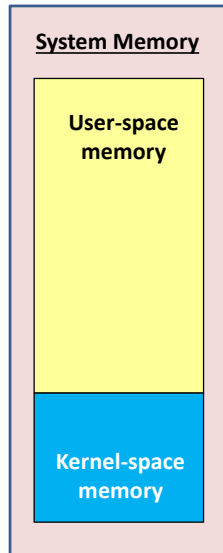
	Kernel-space memory	User-space memory
Storing what	Kernel data structure Kernel code Device drivers	Process' memory Program code of the process
Accessed by whom		



Kernel-space VS User-space

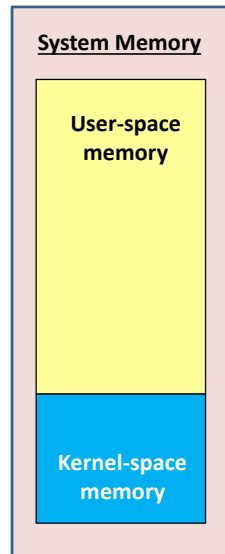
	Kernel-space memory	User-space memory
Storing what	Kernel data structure Kernel code Device drivers	Process' memory. Program code of the process
Accessed by whom	Kernel code	User program code + kernel code

The kernel is invincible!



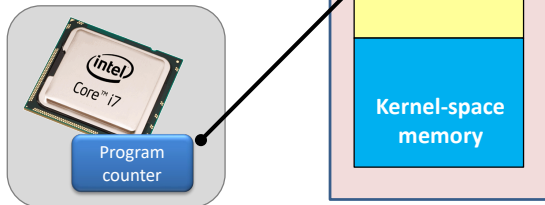
Process is going back and forth...

- A process will switch its execution from user space to kernel space
- **How?**
 - through invoking system call



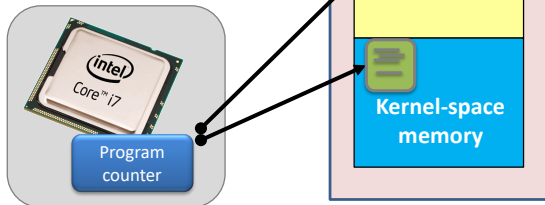
Process is going back and forth...

- Example
 - Say, the CPU is running a program code of a process
 - Where is the code?
 - **User-space memory**
 - Recall the process structure in memory
 - Where should the program counter point to?



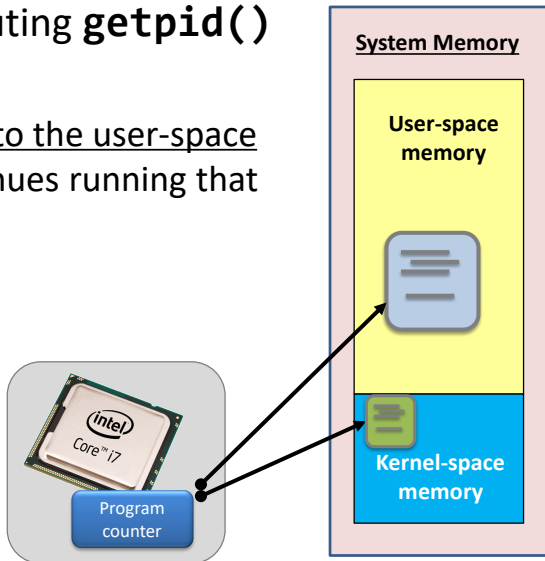
Process is going back and forth...

- What happens...
 - When the process is calling the system call `“getpid()”`
- Where to get the PID
 - PCB (in kernel-space memory)
- The CPU switches from the user-space to the kernel-space, and reads the PID



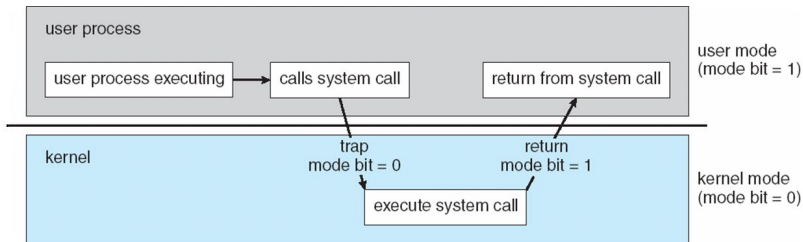
Process is going back and forth...

- After finished executing **getpid()**
 - What happens?
 - CPU switches back to the user-space memory, and continues running that program code



User Mode & Kernel Mode

- Remember this?



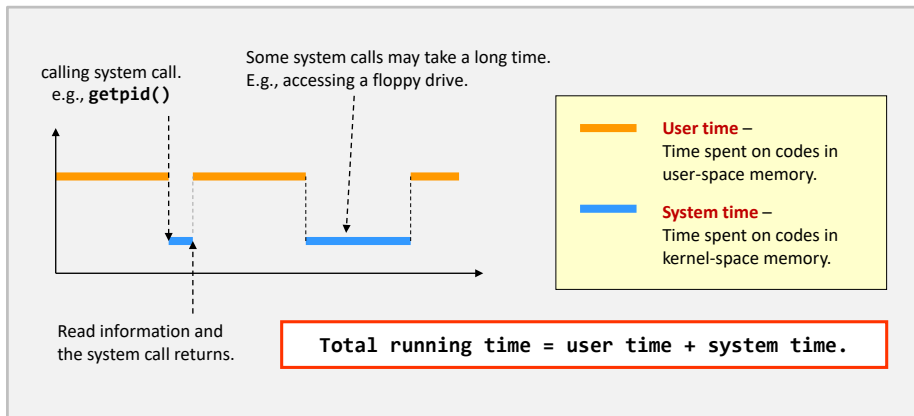
Another question: How much time was spent in each part?

User time VS System time

- So, not just the memory, but also the **execution of a process** is also divided into two parts.
 - User time and system time

User time VS System time

- So, not just the memory, but also the **execution of a process** is also divided into two parts.
 - User time and system time



User time VS System time – example 1

- Let's tell the difference...with the tool “**time**”.

```
$ time ./time_example
```

```
real    0m0.003s
user    0m0.003s
sys     0m0.000s
$ _
```

Time elapsed when “./time_example” terminates.

The user time of “./time_example” measured when the process is on CPU.

The system time of “./time_example” measured when the process is on CPU.

Why comment this line???

```
int main(void) {
    int x = 0;
    for(i = 1; i <= 100000; i++) {
        x = x + i;
        // printf("x = %d\n", x);
    }
    return 0;
}
```

Commented on purpose.

User time VS System time – example 1

- Let's tell the difference...with the tool “**time**”.

```
$ time ./time_example
```

```
real    0m0.003s
user    0m0.003s
sys     0m0.000s
$ _
```

```
int main(void) {
    int x = 0;
    for(i = 1; i <= 100000; i++) {
        x = x + i;
        // printf("x = %d\n", x);
    }
    return 0;
}
```

Commented on purpose.

```
$ time ./time_example
```

```
real    0m0.677s
user    0m0.032s
sys     0m0.227s
$ _
```

```
int main(void) {
    int x = 0;
    for(i = 1; i <= 100000; i++) {
        x = x + i;
        printf("x = %d\n", x);
    }
    return 0;
}
```

Comment released.

See? Accessing hardware costs the process more time.

User time VS System time – example 2

- What is the difference of the two programs?

```
#define MAX 1000000

int main(void) {
    int i;
    for(i = 0; i < MAX; i++)
        printf("x\n");
    return 0;
}
```

```
#define MAX 1000000

int main(void) {
    int i;
    for(i = 0; i < MAX / 5 ; i++)
        printf("x\nx\nx\nx\nx\n");
    return 0;
}
```

Lessons learned: When writing a program, you must consider both the user time and the system time

User time VS System time – short summary

- The user time and the system time together define the **performance** of an application
 - System call plays a major role in **performance**.
 - **Blocking system call**: some system calls even stop your process until the data is available.
- Programmers should pay attention to system performance
 - Reading a file byte-by-byte
 - Reading a file block-by-block, where the size of a block is 4,096 bytes

User space and Kernel space

User time and system time



Working of system calls

- `fork()`;
- `exec*()`;
- `wait()` + `exit()`;



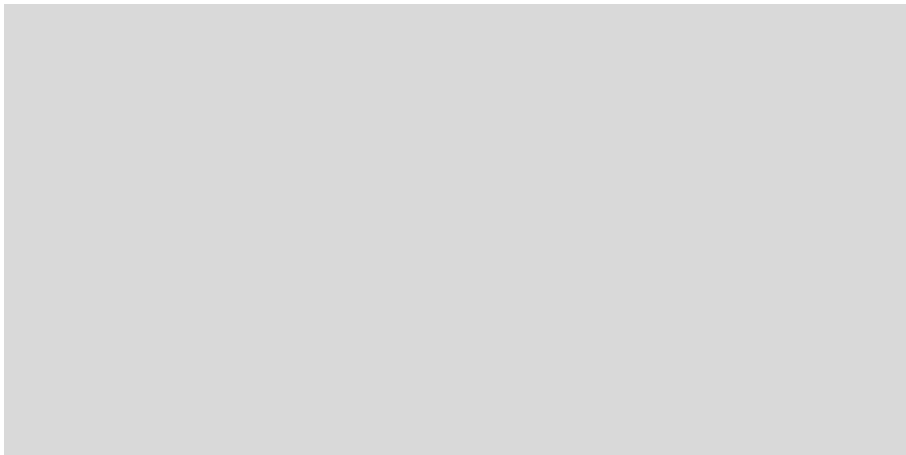
Working of system calls

- **fork();**
- `exec*();`
- `wait() + exit();`



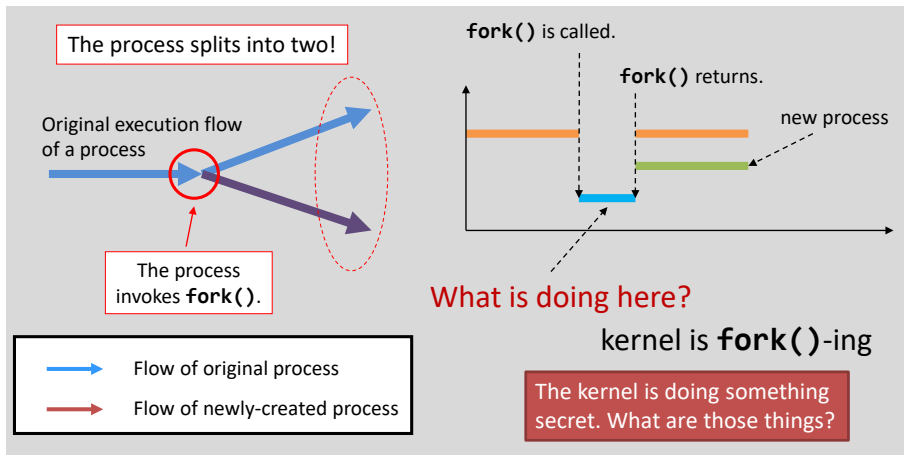
fork()

- From a **programmer's view**, **fork()** behaves like the following:



fork()

- From a programmer's view, **fork()** behaves like the following:



fork()

- From the Kernel's view...

Guess: What will be modified?

Process creation – **fork()** system call

Recall

- **fork()** behaves like “*cell division*”.
 - It creates the child process by **cloning** from the parent process, including...

Cloned items	Descriptions
Program counter [CPU register]	That's why they both execute from the same line of code after fork() returns.
Program code [File & Memory]	They are sharing the same piece of code.
Memory	Including local variables, global variables, and dynamically allocated memory.
Opened files [Kernel's internal]	If the parent has opened a file “A”, then the child will also have file “A” opened automatically.

Process creation – **fork()** system call

Recall

- However...
 - **fork()** does not clone the following...
 - Note: they are all data inside the **memory of kernel**.

Distinct items	Parent	Child
Return value of fork()	PID of the child process.	0
PID	Unchanged.	Different, not necessarily be "Parent PID + 1"
Parent process	Unchanged.	Doesn't have the same parent as that of the parent process.
Running time	Cumulated.	Just created, so should be 0.

fork() in action – the start...

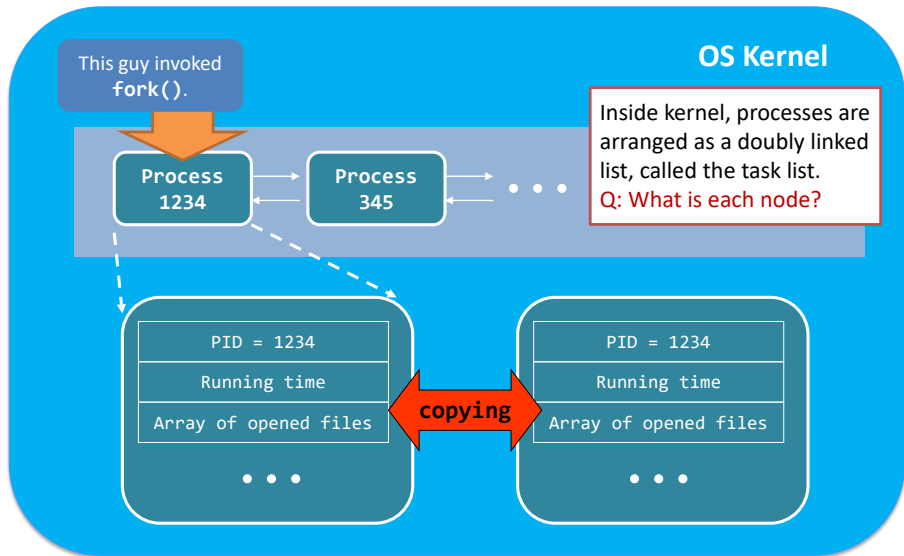
OS Kernel



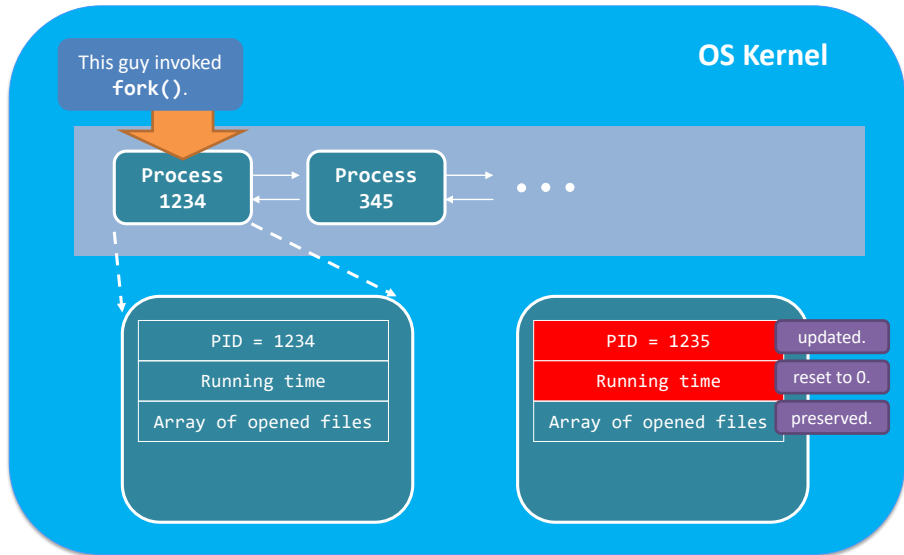
Inside kernel, processes are arranged as a **doubly linked list**, called the task list.

Q: What is each node?

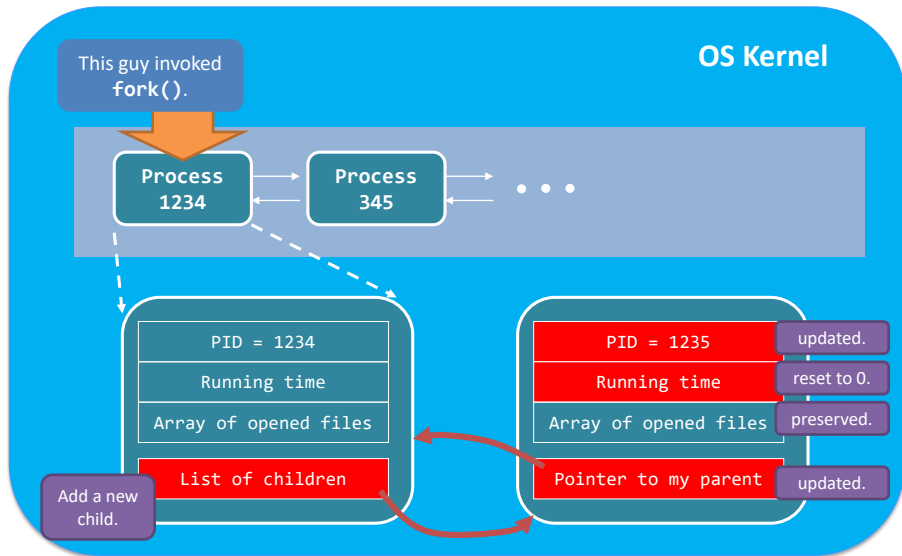
fork() in action – the start...



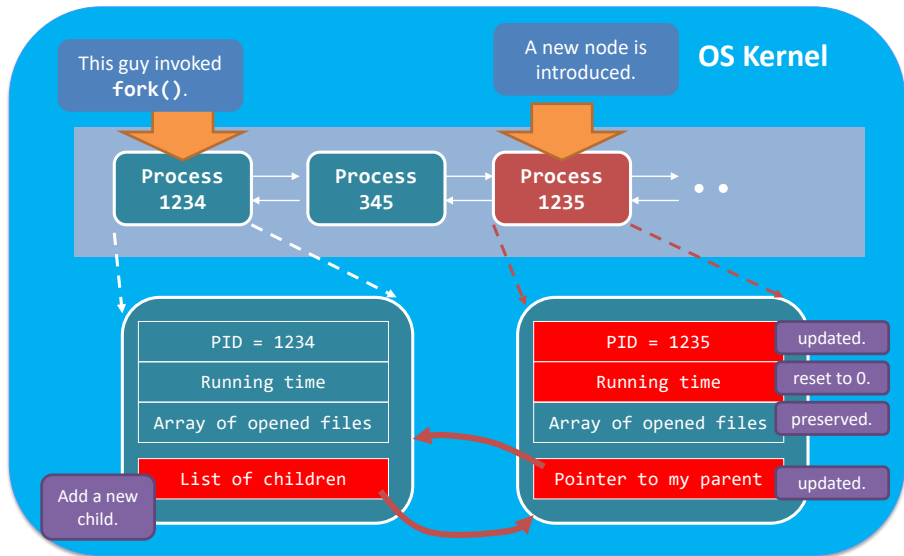
fork() in action – kernel-space update



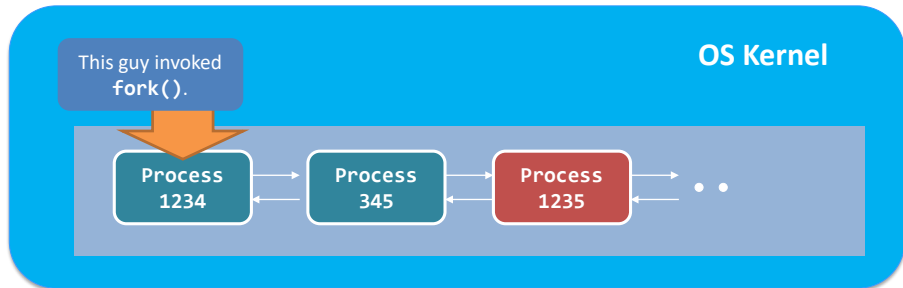
fork() in action – kernel-space update



fork() in action – kernel-space update

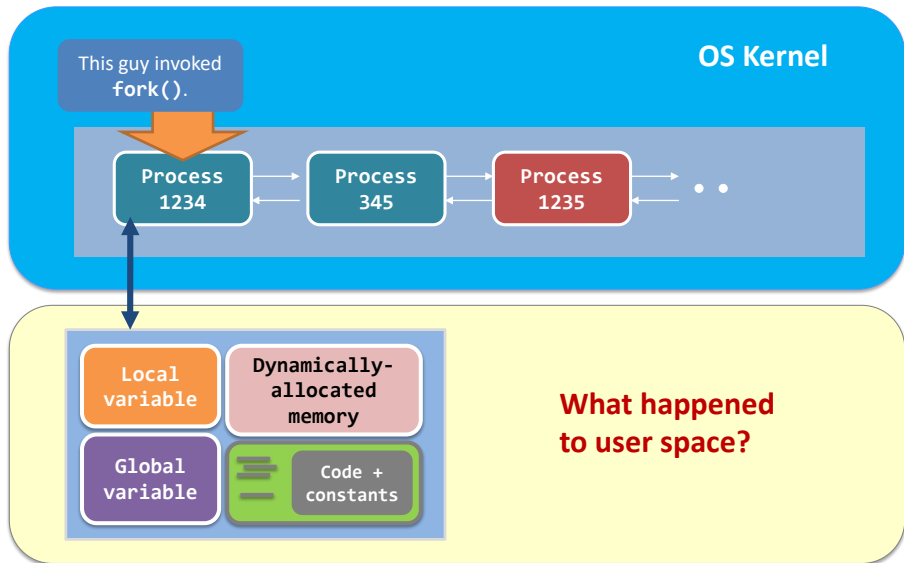


fork() in action – user-space update

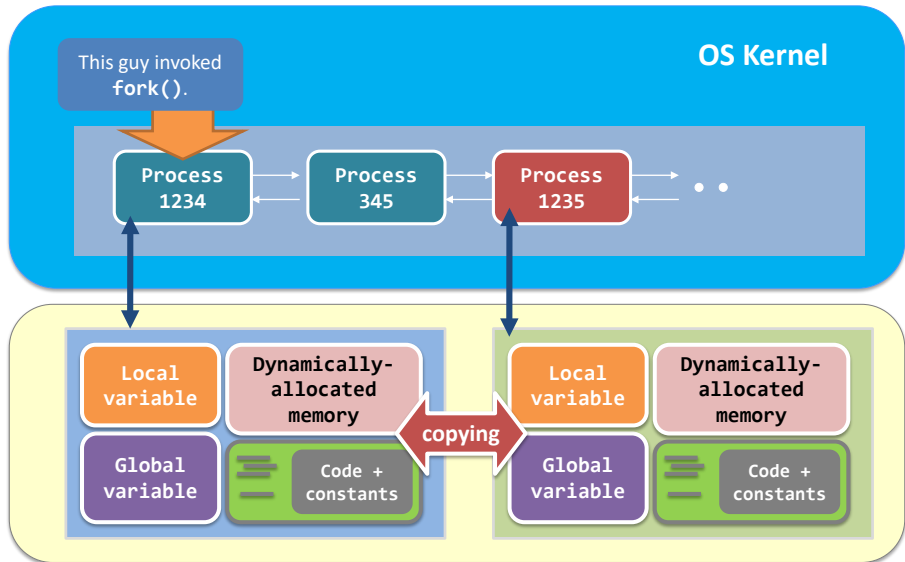


**What happened
to user space?**

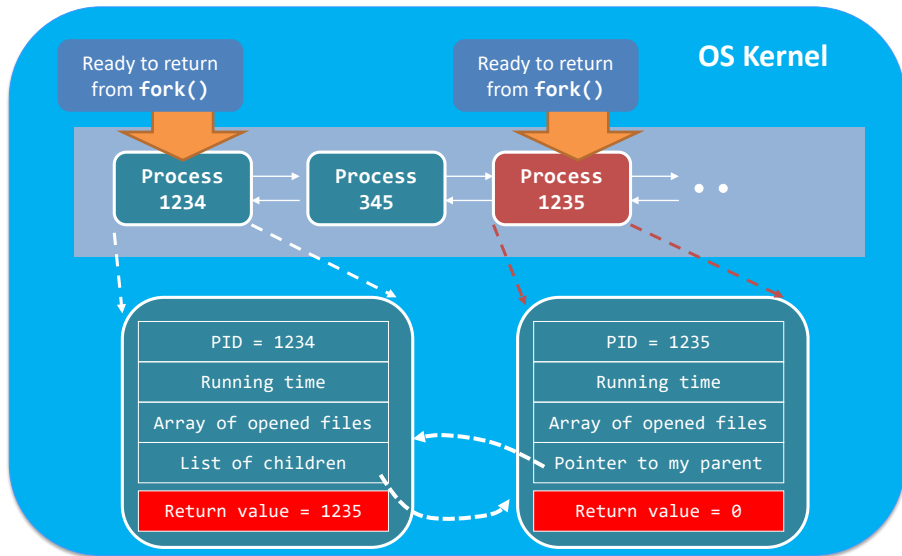
fork() in action – user-space update



fork() in action – user-space update



fork() in action – finish



fork() in action – array of opened files?

- After **fork()**
 - The child process share a set of opened files
- What are the array of opened files?

fork() in action – array of opened files?

- Array of opened files contains:

Array Index	Description
0	Standard Input Stream; FILE *stdin;
1	Standard Output Stream; FILE *stdout;
2	Standard Error Stream; FILE *stderr;
3 or beyond	Storing the files you opened, e.g., fopen() , open() , etc.

- That's why a parent process **shares the same terminal output stream** as the child process!

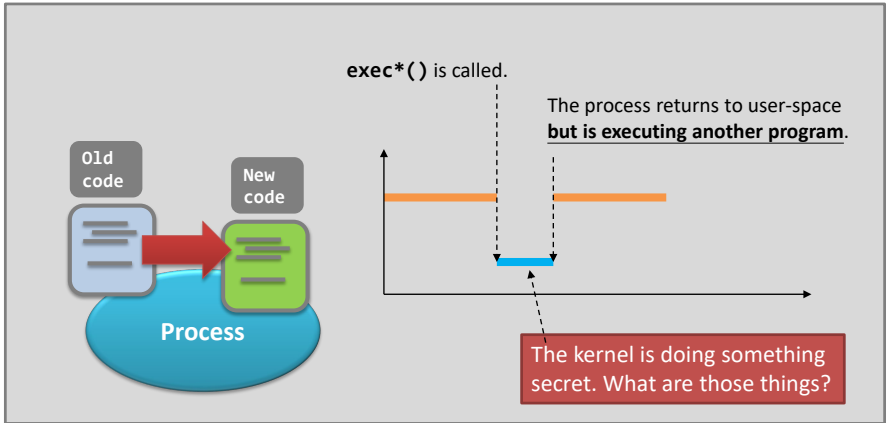
Working of system calls

- `fork()`;
- `exec*()`;

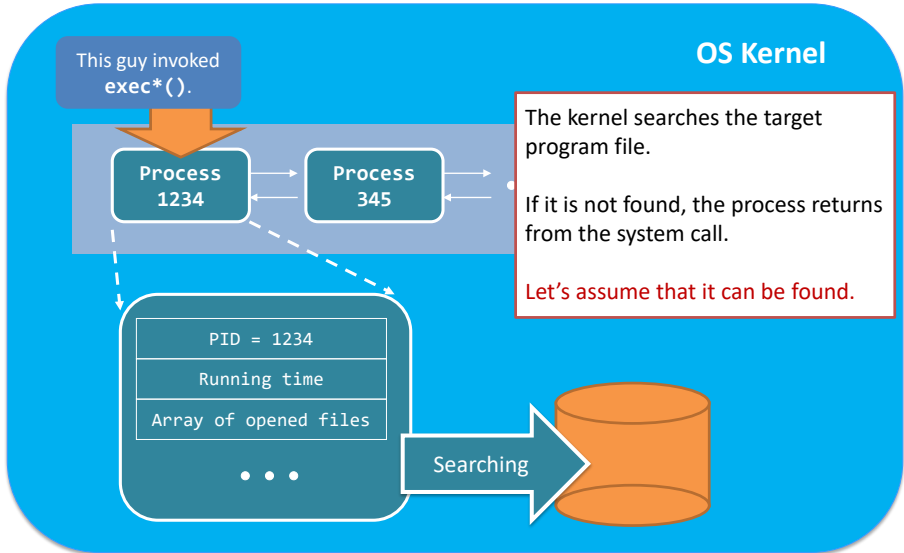


exec*()

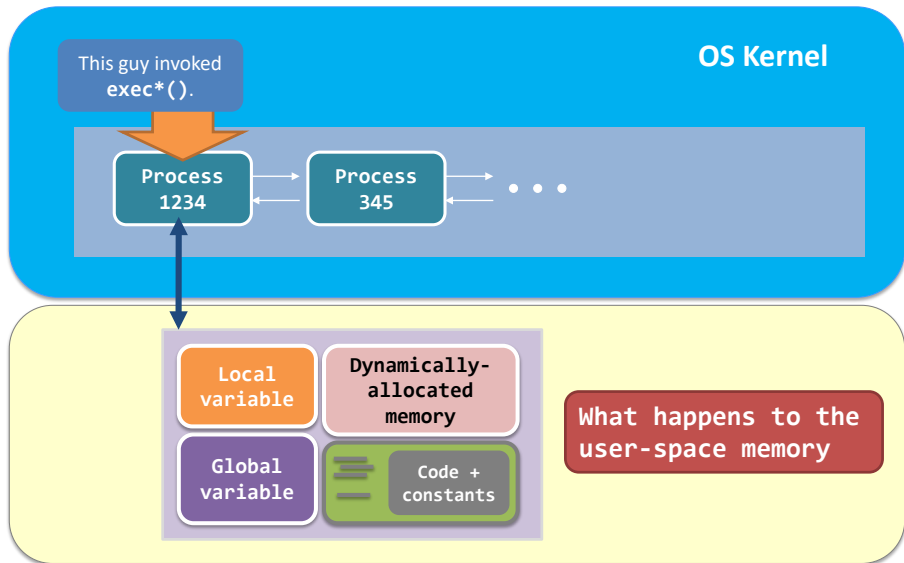
- How about the **exec*()** call family?
e.g., `execl("/bin/ls", "/bin/ls", NULL);`



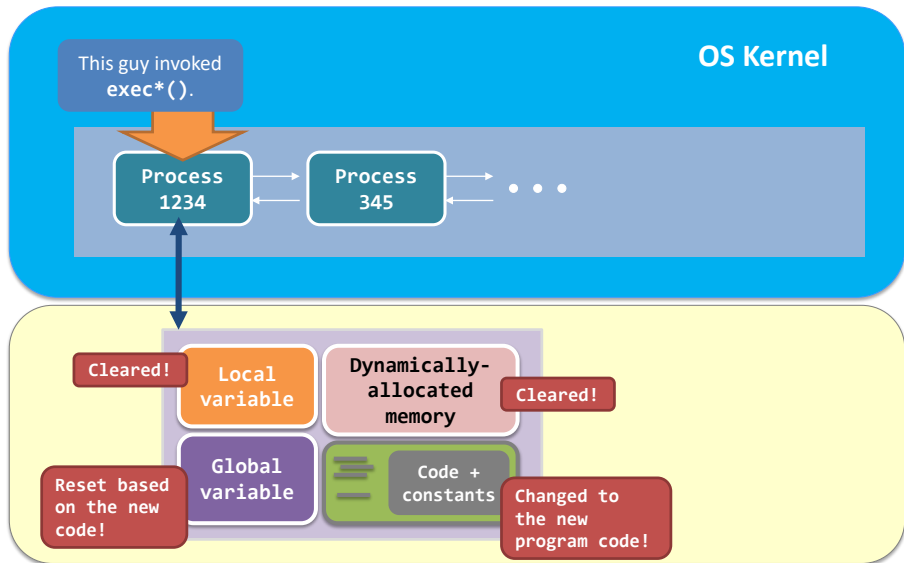
exec*() in action – the start...



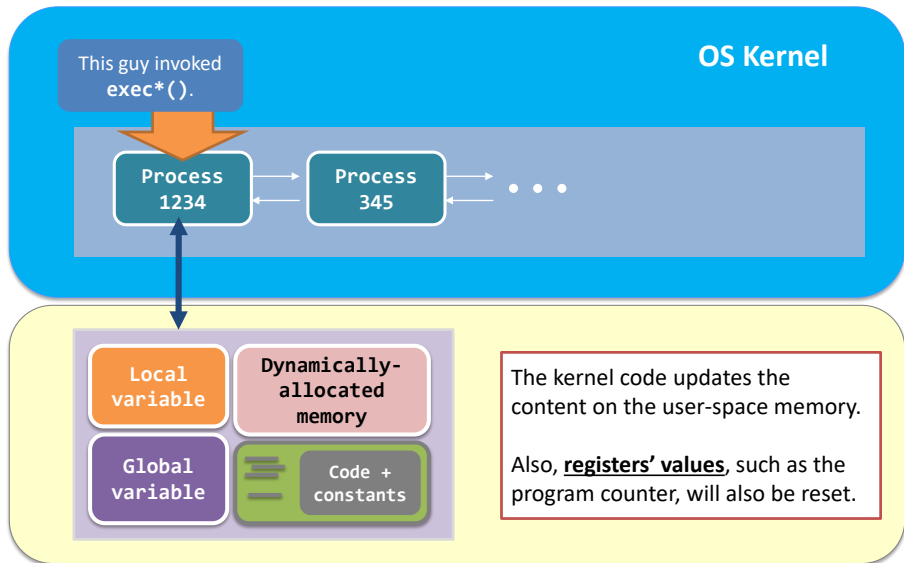
exec*() in action – the end



exec*() in action – the end



exec*() in action – the end



Working of system calls

- `fork()`;
- `exec*()`;
- `wait() + exit()`;



Recall the example

```
1 int system_test(const char *cmd_str) {
2     if(cmd_str == -1)
3         return -1;
4     if(fork() == 0) {
5         execl("/bin/sh", "/bin/sh",
6             "-c", cmd_str, NULL);
7         fprintf(stderr,
8             "%s: command not found\n", cmd_str);
9         exit(-1);
10    }
11    wait(NULL);
12    return 0;
13 }
14
15 int main(void) {
16     printf("before...\n\n");
17     system_test("/bin/ls");
18     printf("\nafter...\n");
19     return 0;
20 }
```

The parent is
suspended until
the child
terminates

```
$ ./system_implement_2
before...
```

```
system_implement_2
System_implement_2.c
```

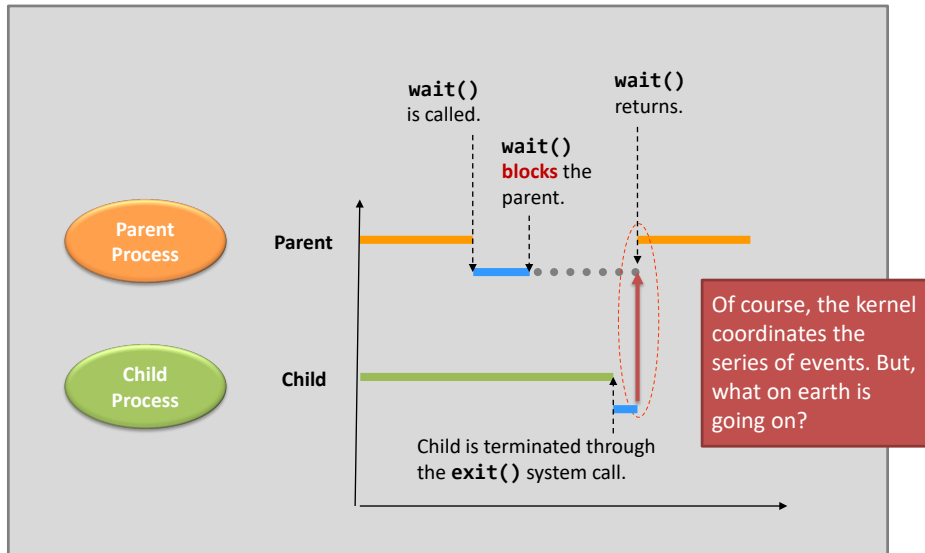
```
after...
```

```
$ _
```


`wait()`

- **`wait()`** system call
 - Suspend the parent process
 - Wake up when one child process terminates
- How to terminate the child process
 - Through the **`exit()`** system call
- **`wait()`** and **`exit()`** – they come together!

`wait()` and `exit()` – Time Analysis

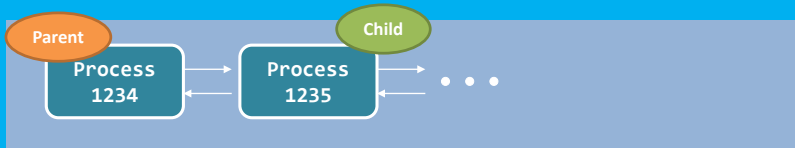


Guess...

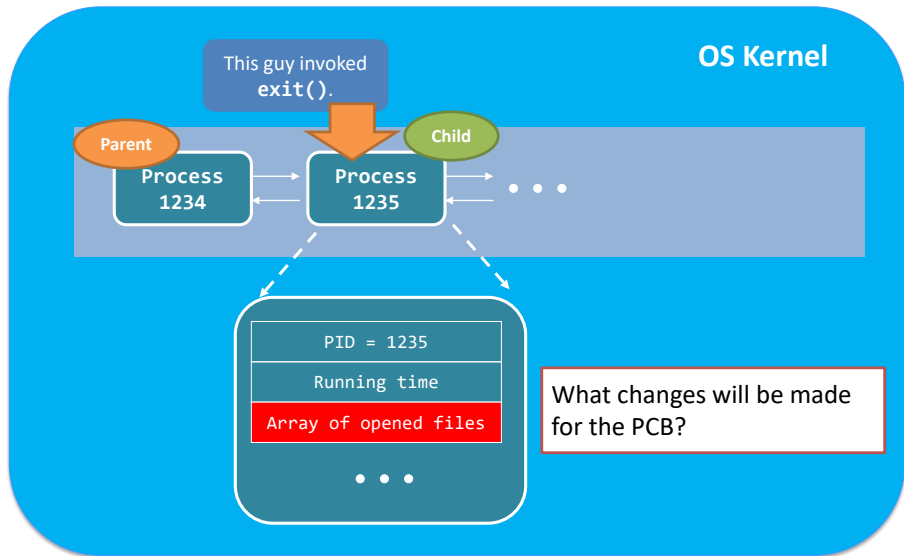
- What is going on inside kernel?
 - Child: **exit()**
 - Process data + PCB
 - Parent: **wait()**
 - Process data + PCB

`wait()` and `exit()` – child side

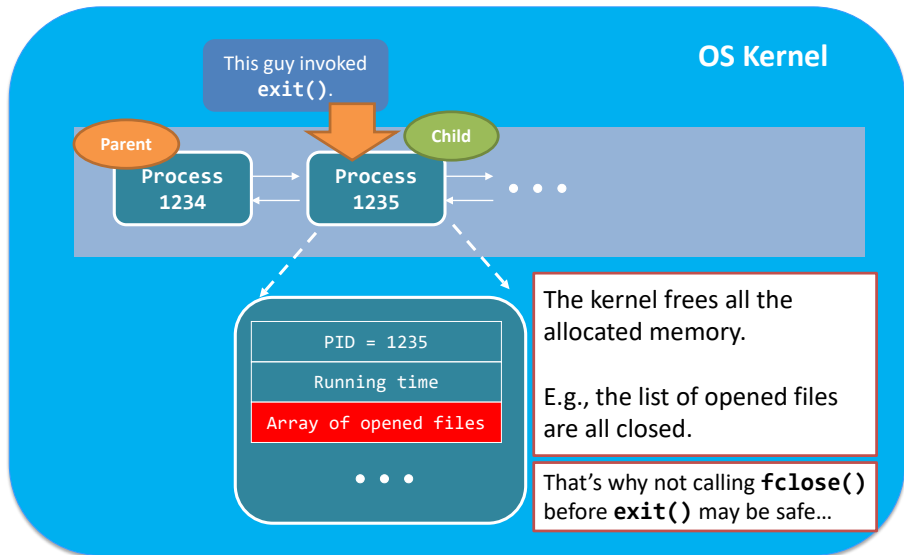
OS Kernel



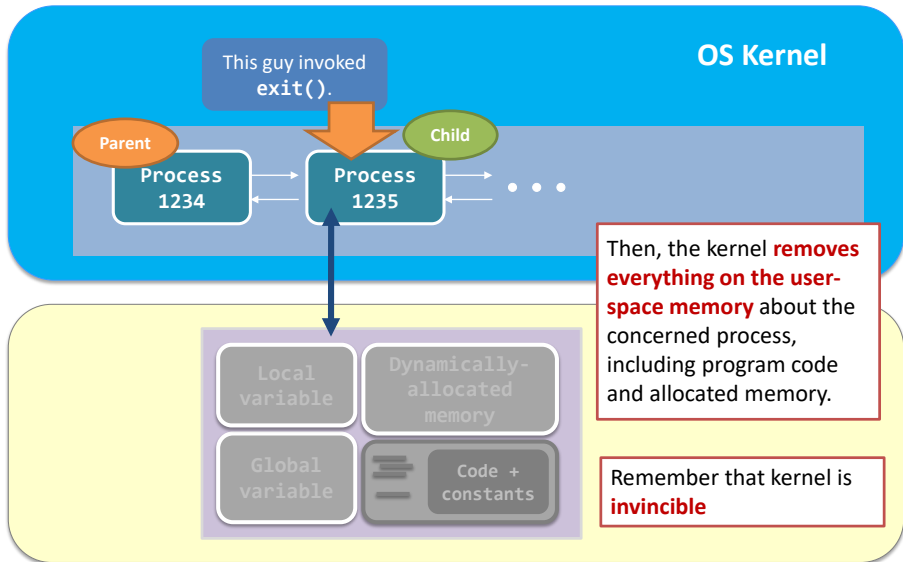
`wait()` and `exit()` – child side



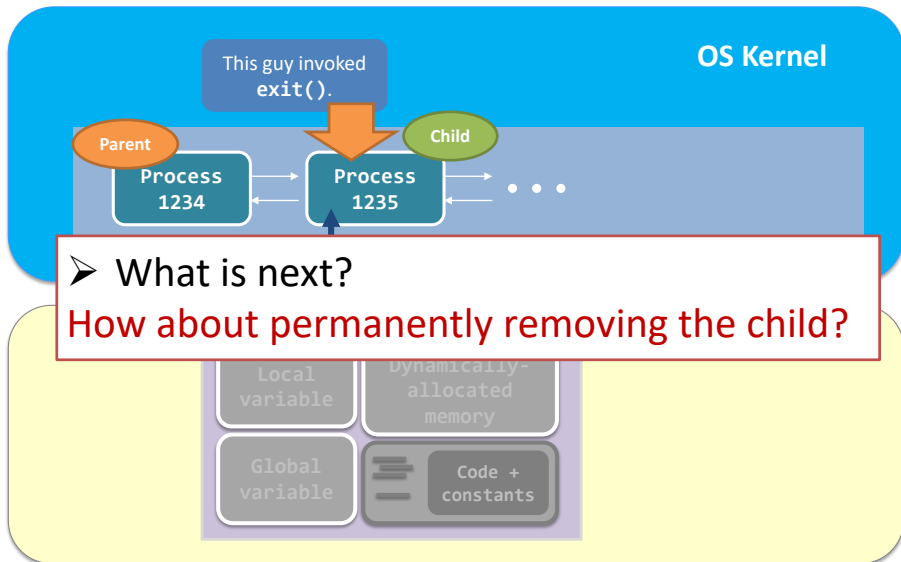
`wait()` and `exit()` – child side



wait() and exit() – child side



`wait()` and `exit()` – child side



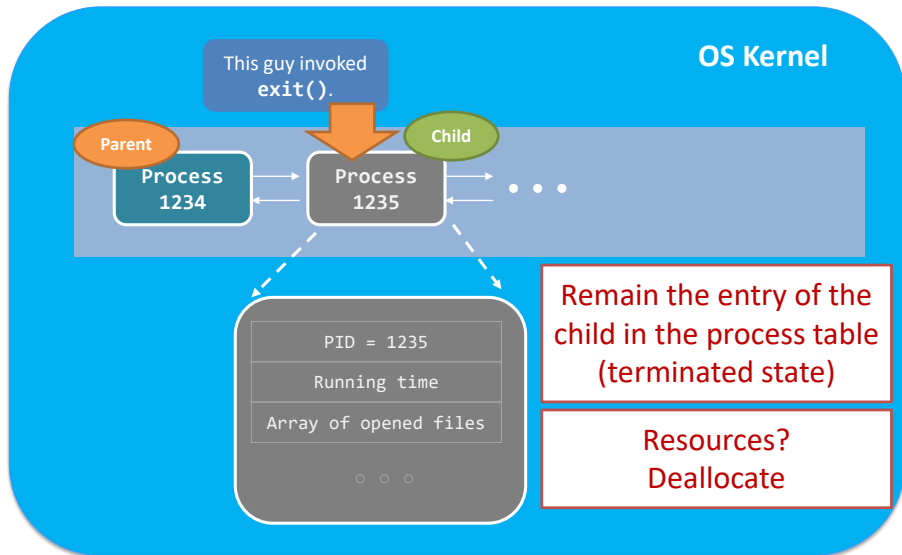
`wait()` and `exit()` – child side

OS Kernel

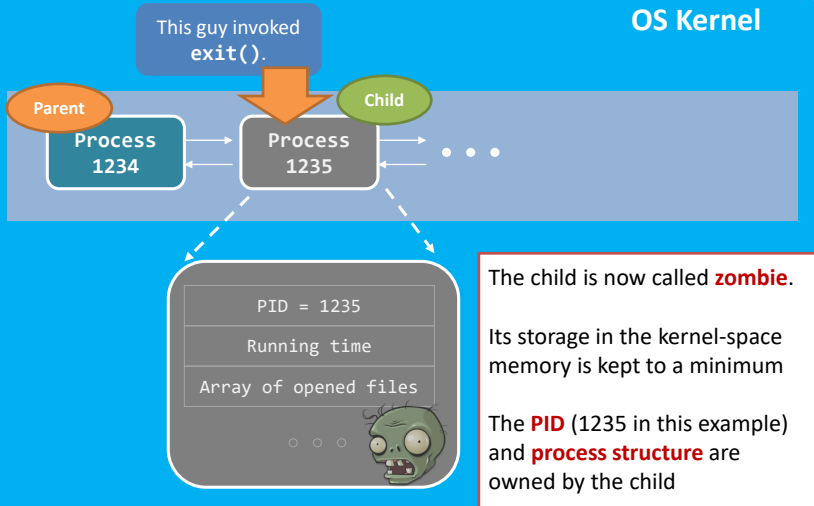


Removed from the process table immediately?
Not really! Why?

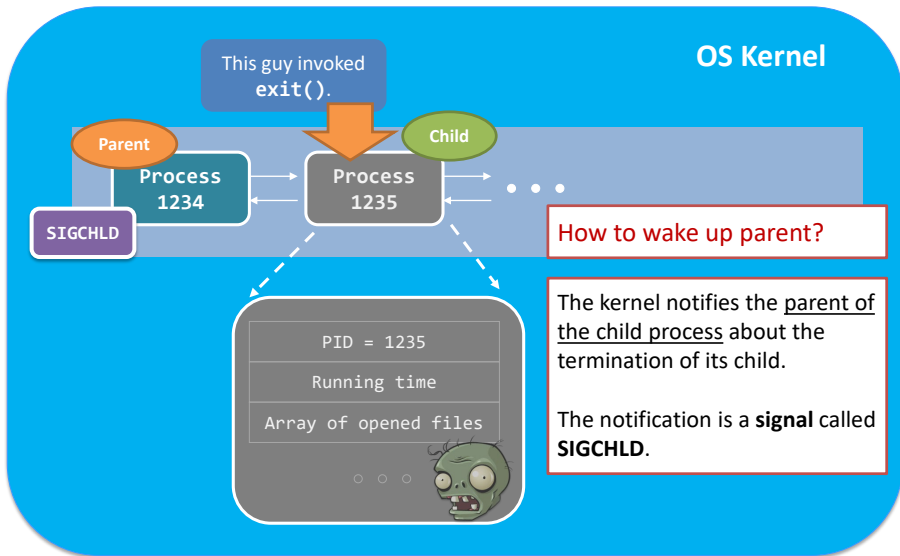
`wait()` and `exit()` – child side



wait() and exit() – child side



wait() and exit() – child side



Signal

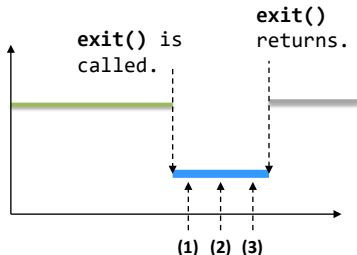
- What is signal?
 - A software interrupt
 - It takes steps as in the hardware interrupt
- Two kinds of signals
 - Generated from user space
 - **Ctrl+C**, **kill()** system call, etc.
 - Generated from kernel and CPU
 - Segmentation fault (**SIGSEGV**), Floating point exception (**SIGFPE**), child process termination (**SIGCHLD**), etc.
- Signal is very hard to master, will be skipped in this course
 - Reference: Advanced Programming Environment in UNIX
 - Linux manpage

A short summary for `exit()`

Step (1) Clean up most of the allocated kernel-space memory.

Step (2) Clean up all user-space memory.

Step (3) Notify the parent with SIGCHLD.



Although the child is still in the system, it is no longer running. There is no program code!!!

It turns into a mindless zombie...

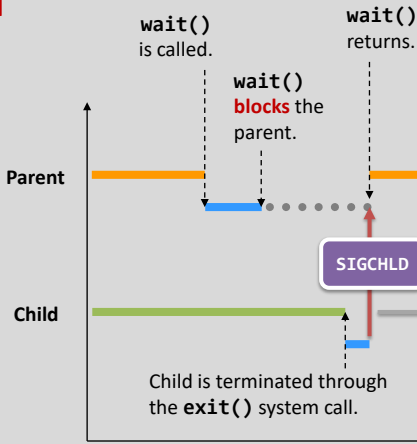
You cannot kill a zombie process, as it is already dead. Then how to eliminate it?

`wait()` and `exit()` – they come together!

How to proceed
with `wait()`?

Parent
Process

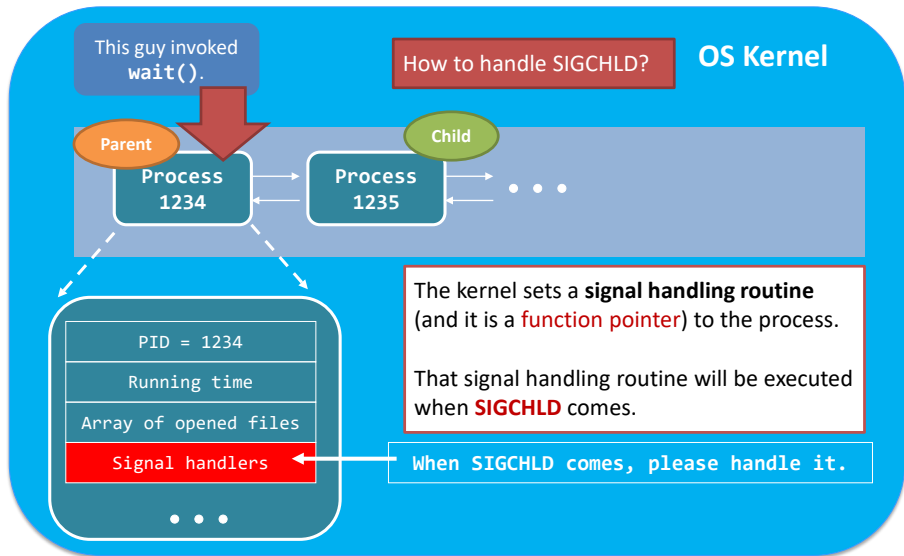
Child
Process



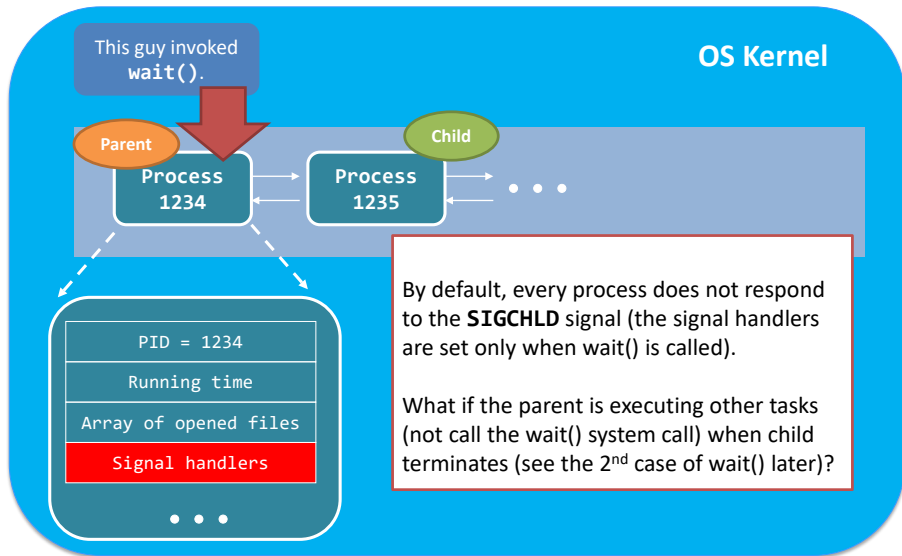
Now, it is trivial to see that **SIGCHLD** signal is the trick!

But, how to handle SIGCHLD?

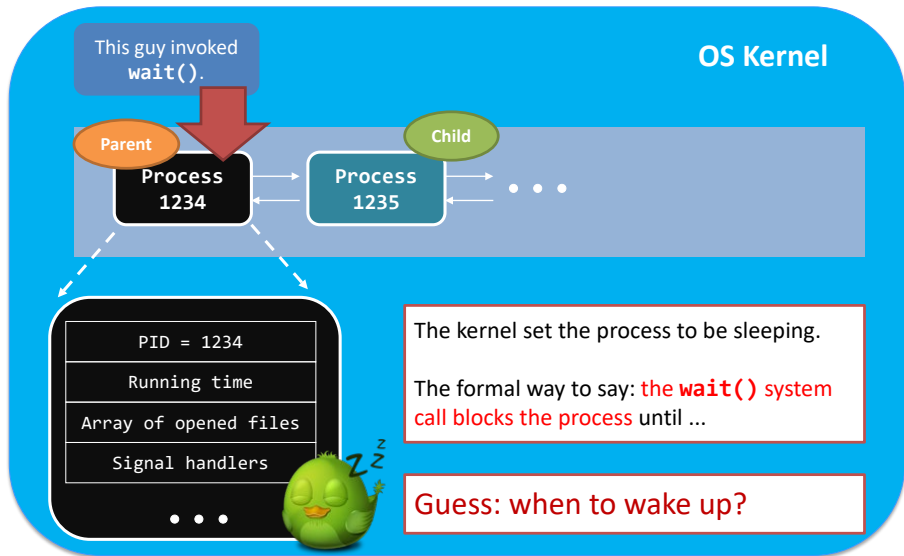
wait() and exit() – parent side



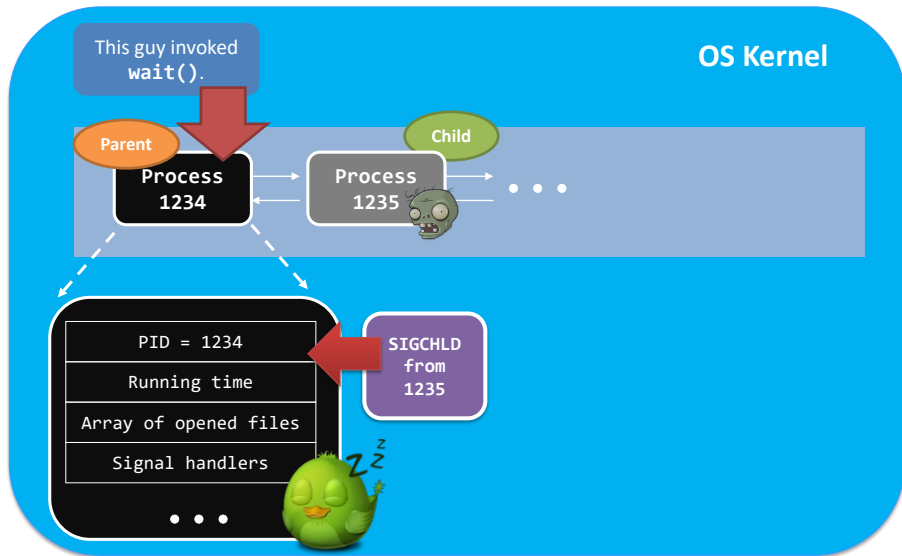
wait() and exit() – parent side



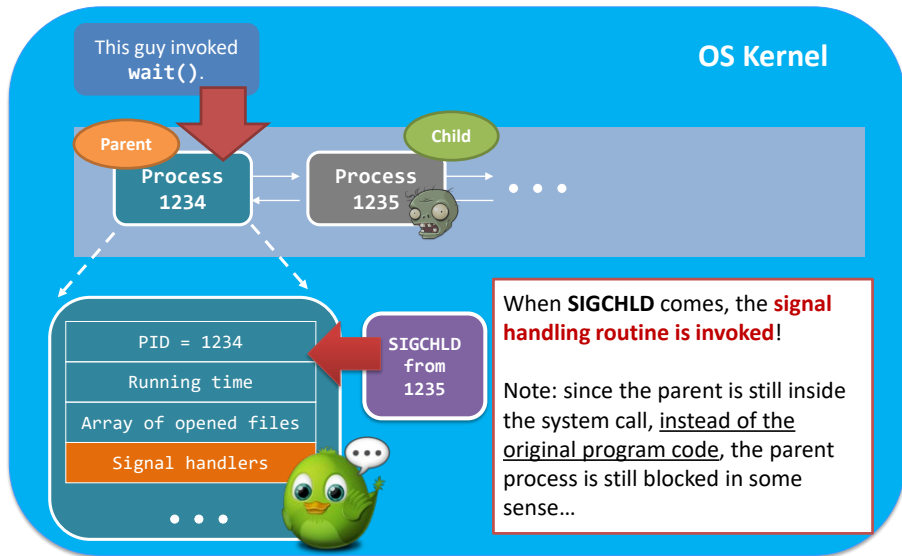
wait() and exit() – parent side



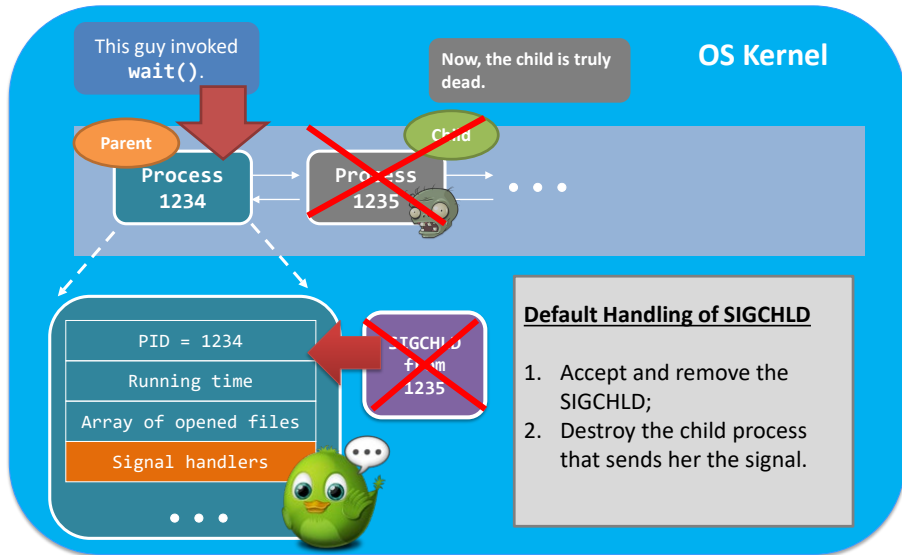
wait() and exit() – parent side



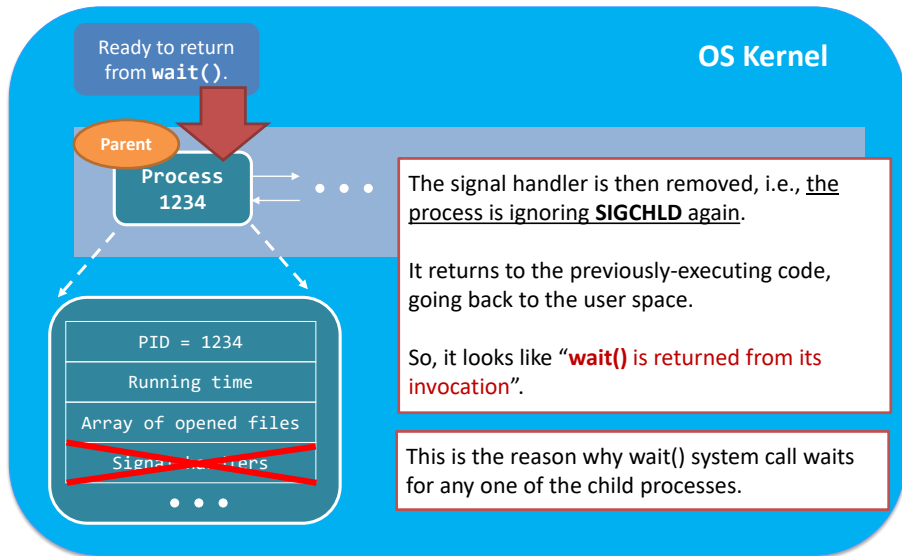
wait() and exit() – parent side



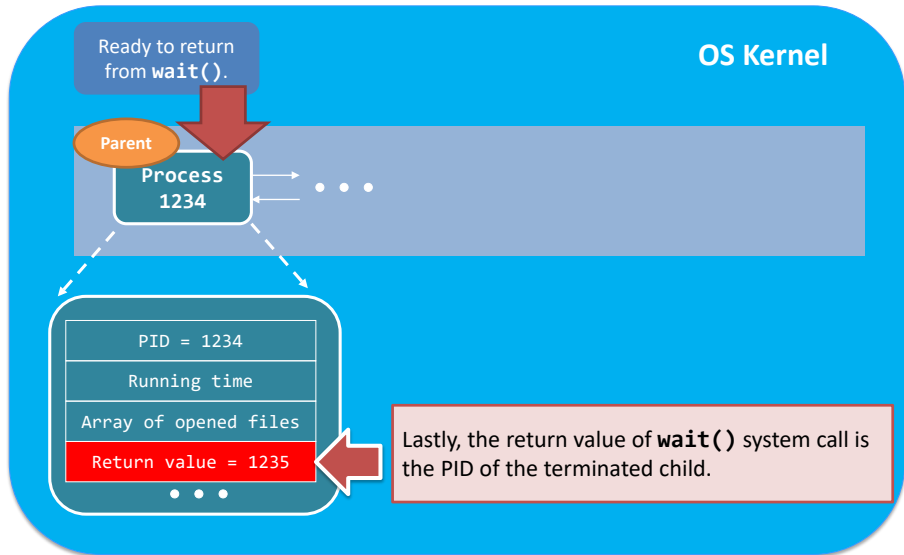
wait() and exit() – parent side



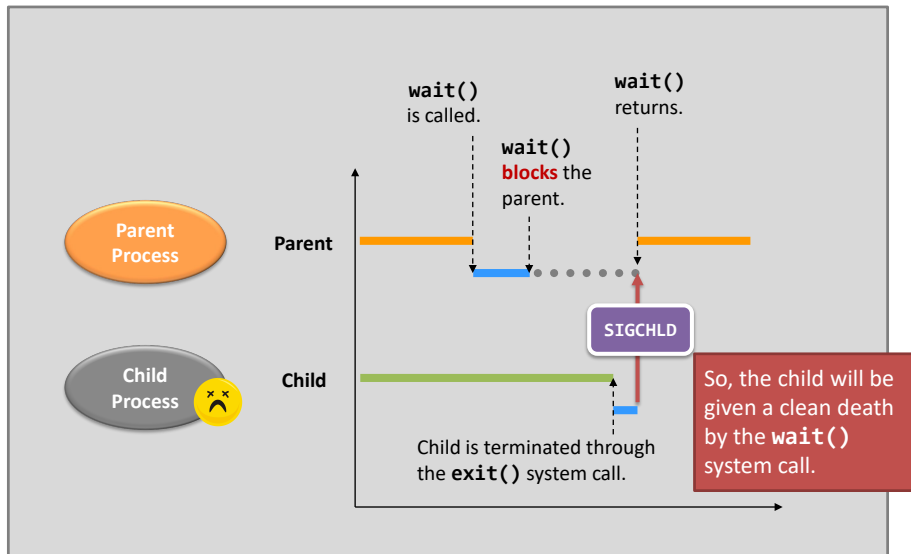
wait() and exit() – parent side



`wait()` and `exit()` – parent side

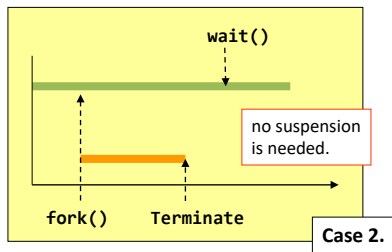
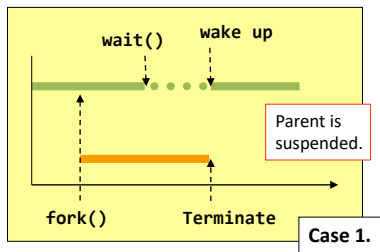


`wait()` and `exit()` – parent side



Is it done?

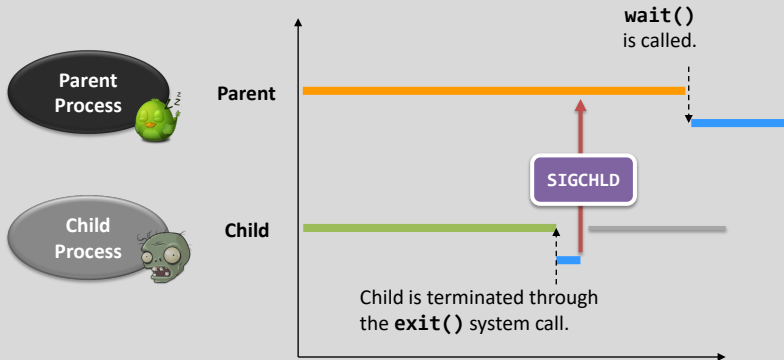
- How about `wait()` is called after the child already terminated?
 - Remember the case 2 (which is safe)



`wait()` and `exit()` – parent side

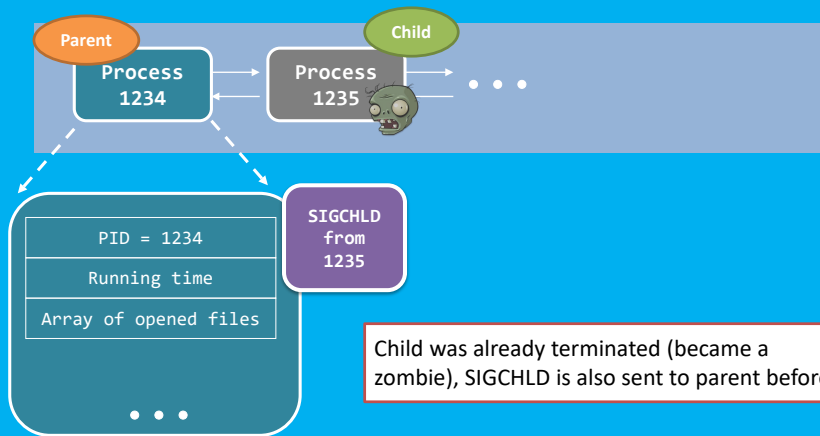
Case 2.

What is going on inside the kernel?

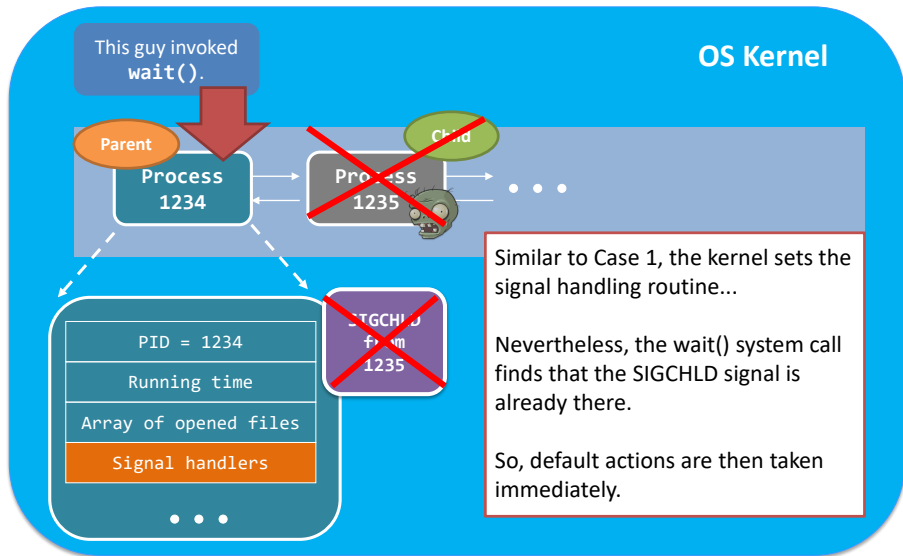


wait() and exit() – parent side

OS Kernel

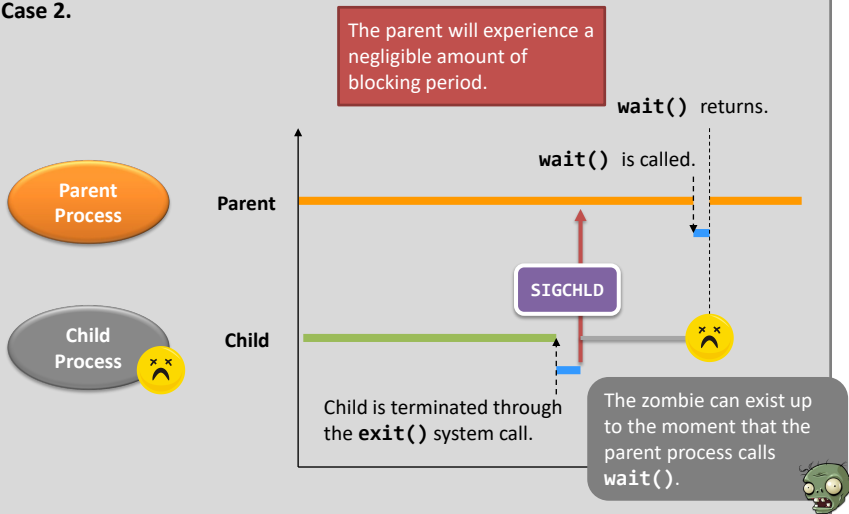


wait() and exit() – parent side



`wait()` and `exit()` – parent side

Case 2.



Orphans (zombies)

- What would happen if a parent did not invoke `wait()` and terminated?
 - Remember the `reparent` operation in Linux?
- `init` is the new parent, and it **periodically** invokes `wait()`

wait() and **exit()** – short summary

- A process is turned into a zombie when...
 - The process calls **exit()**.
 - The process returns from **main()**.
 - The process terminates abnormally.
 - You know, the kernel knows that the process is terminated abnormally. Hence, the kernel invokes **exit()** by itself.
- Remember why **exec*()** does not return to its calling process in previous example...

wait() and **exit()** – short summary

- **wait()** is to reap zombie child processes
 - You should never leave any zombies in the system.
- Linux will label zombie processes as “<**defunct**>”.
 - To look for them: **ps aux | grep defunct**
- Learn **waitpid()** by yourself...

wait() and exit() – Example

```
1 int main(void)
2 {
3     int pid;
4     if( (pid = fork()) ) {
5         printf("Look at the status of the process %d\n", pid);
6         while( getchar() != '\n' );
7         wait(NULL);
8         printf("Look again!\n");
9         while( getchar() != '\n' );
10    }
11    return 0;
12 }
```

What is the purpose of this program?

wait() and exit() – Example

```
1 int main(void)
2 {
3     int pid;
4     if( (pid = fork()) ) {
5         printf("Look at the status of the process %d\n", pid);
6         while( getchar() != '\n' );
7         wait(NULL);
8         printf("Look again!\n");
9         while( getchar() != '\n' );
10    }
11    return 0;
12 }
```

This program requires you to type “enter” twice before the process terminates.

You are expected to see **the status of the child process changes** between the 1st and the 2nd “enter”.

Working of system calls

- `fork()`;
- `exec*()`;
- `wait()` + `exit()`;
- **importance/fun in knowing the above things?**

The role of **wait()** in the OS...

- Why calling **wait()** is important
 - It is not about process execution/suspension...
 - It is about **system resource management**.
- Think about it:
 - A zombie takes up a PID;
 - The total number of PIDs are limited;
 - Read the limit: "**cat /proc/sys/kernel/pid_max**"
 - **What will happen if we don't clean up the zombies?**

When `wait()` is absent...

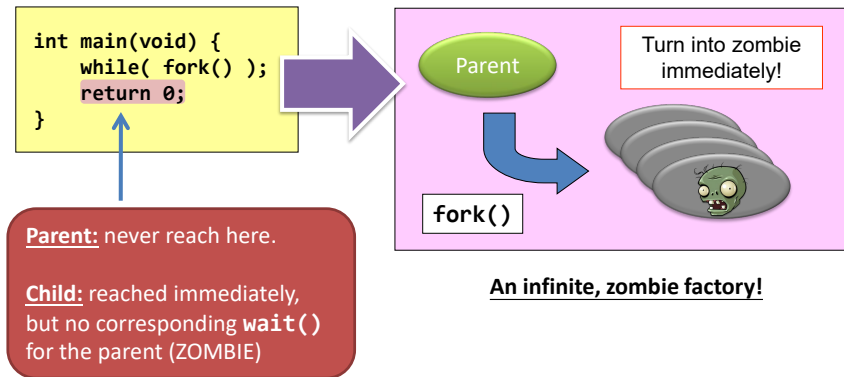
- What is the result of this program?
 - Do not try to know the result by running it

```
int main(void) {  
    while( fork() );  
    return 0;  
}
```

Think about what will be
happened to both parent
and child processes?

When `wait()` is absent...

- Don't try this...



Summary

- Process concept
 - Process vs program
 - User-space memory + PCB
- Process operations
 - Creation, program execution, termination
 - The internal workings of
 - **fork()**
 - **exec*()**
 - **wait()+exit()**: come together
- Calling **wait()** is important

Operating Systems

Associate Prof. Yongkun Li

中科大-计算机学院 副教授

<http://staff.ustc.edu.cn/~ykli>

Ch4 Threads

Chapter 4: Threads

- Thread Concepts
 - Why use threads
 - Structure in Memory
 - Benefits and Challenges
 - Thread Models
- Programming
 - Basic Programming: Pthreads Library
 - Implicit Threading: Thread Pools & OpenMP

Multi-threading

- Motivation

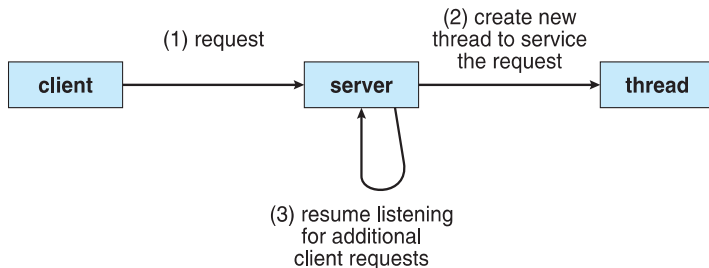


Motivation - Application Side

- Most software applications are multithreaded, each application is implemented as a process with several threads of control
 - Web browser
 - displays images, retrieve data from network
 - Word processor
 - display graphics, respond to keystrokes, spelling & grammar checking

Motivation - Application Side

- Most software applications are multithreaded
 - Web browser
 - Word processor
 - Similar tasks in a single application (web server)
 - Accept client requests, service the requests
 - Usually serve thousands of clients



Motivation – Application Side

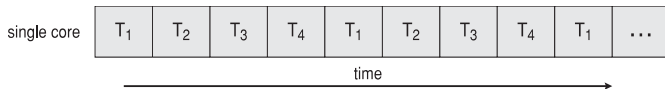
- Why not create a process for each task?
 - Process creation is
 - Heavy-weighted
 - Resource intensive
- Still remember what kinds of data are included in a process...
 - Text, data, stack, heap in user-space memory
 - PCB in kernel-space memory
- Many of the data can be shared between multiple tasks within an application

Motivation – System Side

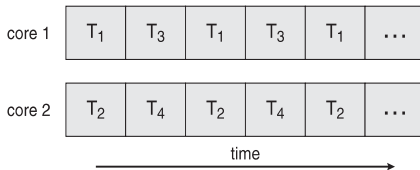
- Modern computers usually contain multicores
 - But, each processor can run only one process at a time
 - CPU is not fully utilized
- How to improve the efficiency?
 - Assign one task to each core
 - Real parallelism (not just concurrency with interleaving on single-core system)

Concurrency vs. Parallelism

Concurrent execution on single-core system:



Parallel execution on a multi-core system:

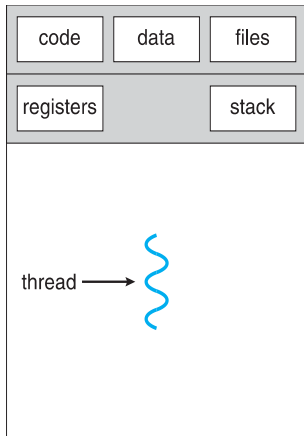


Multi-threading

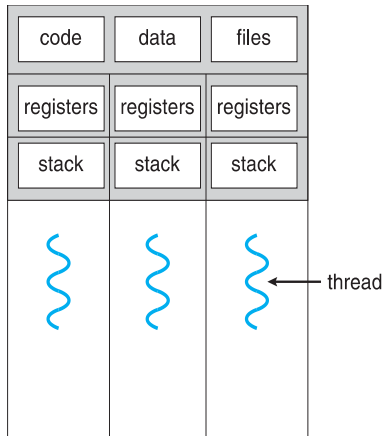
- Motivation
- Thread Concept



High-level Idea



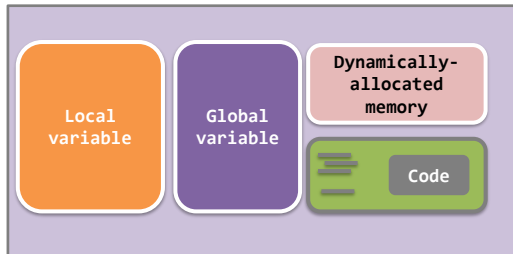
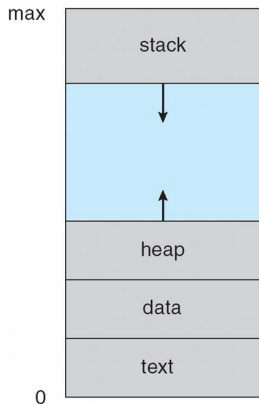
single-threaded process



multithreaded process

Recall: Process in Memory

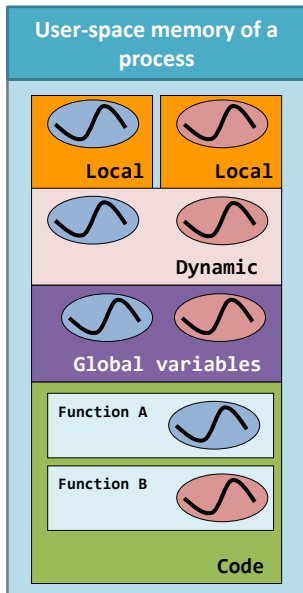
- User-space memory of Process A



Multi-thread – internals

Code

- All threads **share** the same code.
- A thread starts with **one specific function**.
 - We name it the **thread function**
 - Functions A & B in the diagram
- The thread function can invoke other functions or system calls
- But, a thread could **never return to the caller of the thread function.**

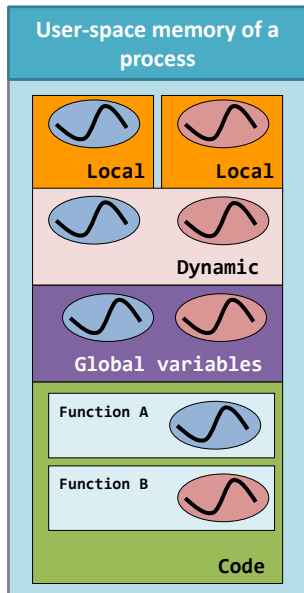


Multi-thread – internals

Dynamically allocated memory

Global variables

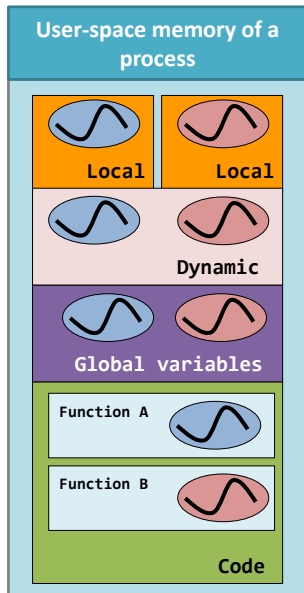
- All threads **share** the same global variable zone and the same dynamically allocated memory
- All threads can read from and write to both areas



Multi-thread – internals

Local variables

- Each thread has its own memory range for the local variables
- So, the stack is the private zone for each stack

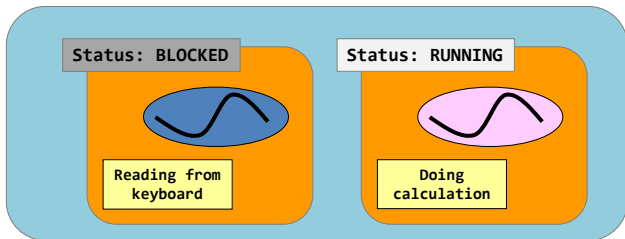


Benefits of Multi-thread

- **Responsiveness and multi-tasking**

- Multi-threading design allows an application to do **parallel tasks simultaneously**
- Example: Although a thread is blocked, the process can still depend on another thread to do other things!
- Especially important for interactive applications (user interface)

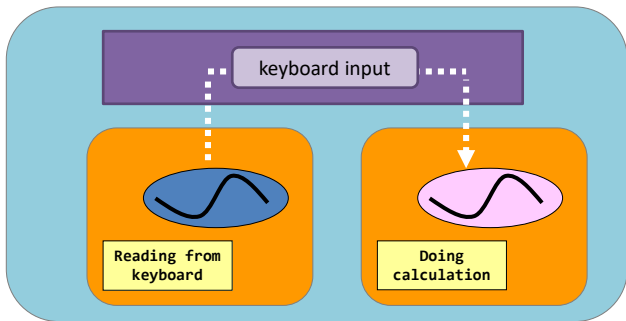
It'd be nice to assign **one thread** for **one blocking system/library call**.



Benefits of Multi-thread

- **Ease in data sharing**, can be done using:
 - global variables, and
 - dynamically allocated memory.
- Processes share resources via shared memory or message passing, which must be explicitly arranged by the programmer

Of course, this leads to the **mutual exclusion** & the **synchronization** problems (will be talked in later chapters)



Benefits of Multi-thread

- **Economy**

- Allocating memory and resources for process creation is costly, dozens of times slower than creating threads
- Context-switch between processes is also costly, several times of slower

- **Scalability**

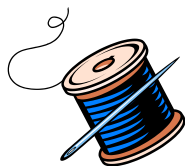
- Threads may be running in parallel on different cores

Programming Challenges

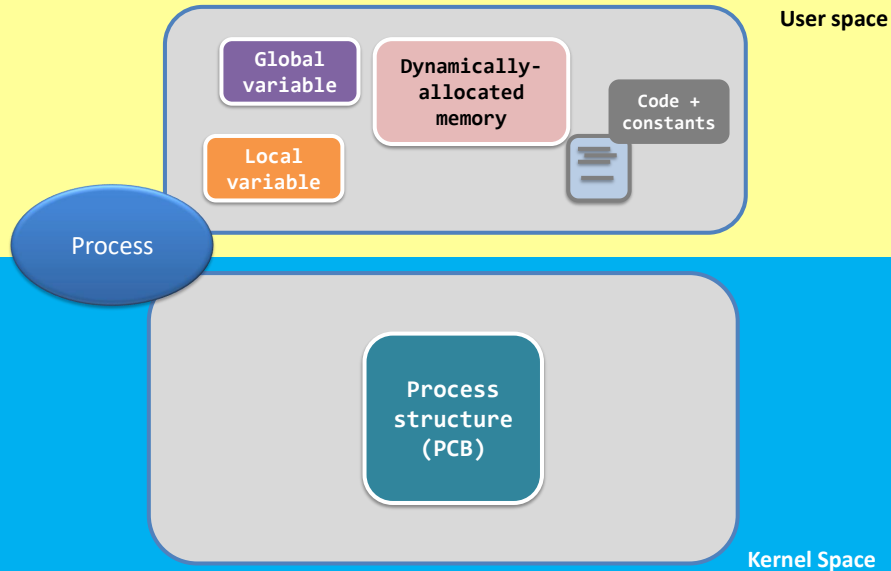
- **Identifying tasks**
 - Divide separate and concurrent tasks
- **Balance**
 - Tasks should perform equal work of equal value
- **Data splitting**
 - Data must be divided to run on separate cores
- **Data dependency**
 - Synchronization is needed
- **Testing and debugging**

Multi-threading

- Motivation
- Thread Concept
- Thread Models



Recall Process Structure



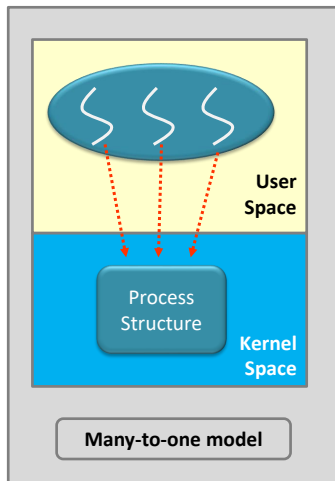
Similarly...

- Thread should also include
 - Data/resources in user-space memory
 - Structure in kernel
- How to provide thread support?
 - User thread
 - Implement in user space
 - Kernel thread
 - Supported and managed by kernel
- Thread models (relationship between user/kernel thread)
 - Many-to-one
 - One-to-one
 - Many-to-many

Thread models

- **Many-to-One Model**

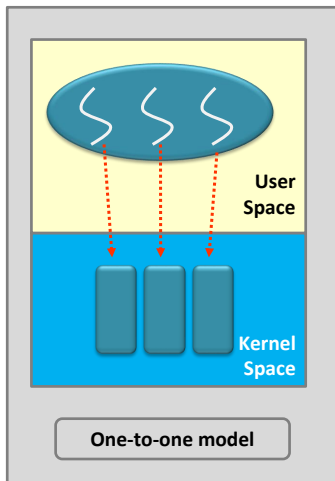
- All the threads are mapped to one process structure in the kernel.
- Merit
 - Easy for the kernel to implement.
- Drawback
 - When a blocking system call is called, all the threads will be blocked
- **Example.** Old UNIX & green thread in some programming languages.



Thread models

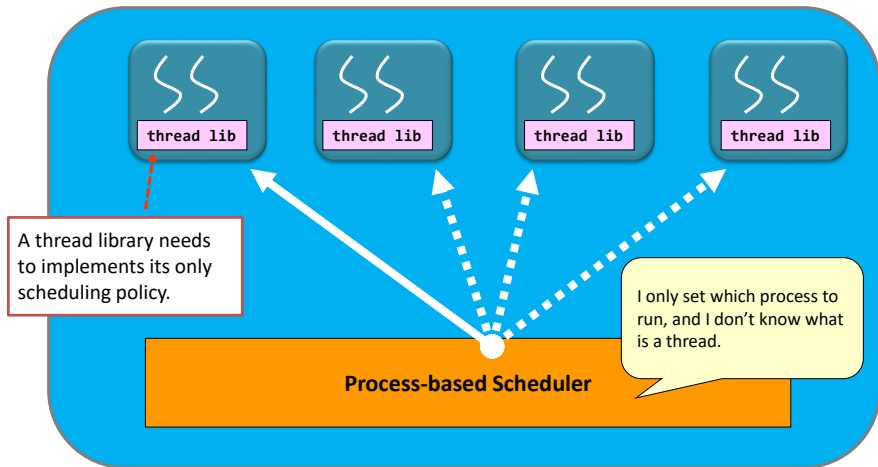
• One-to-One Model

- Each thread is mapped to a process or a thread structure
- Merit:
 - Calling blocking system calls only block those calling threads
 - A high degree of concurrency
- Drawback:
 - Cannot create too many threads as it is restricted by the size of the kernel memory
- **Example.** Linux and Windows follow this thread model



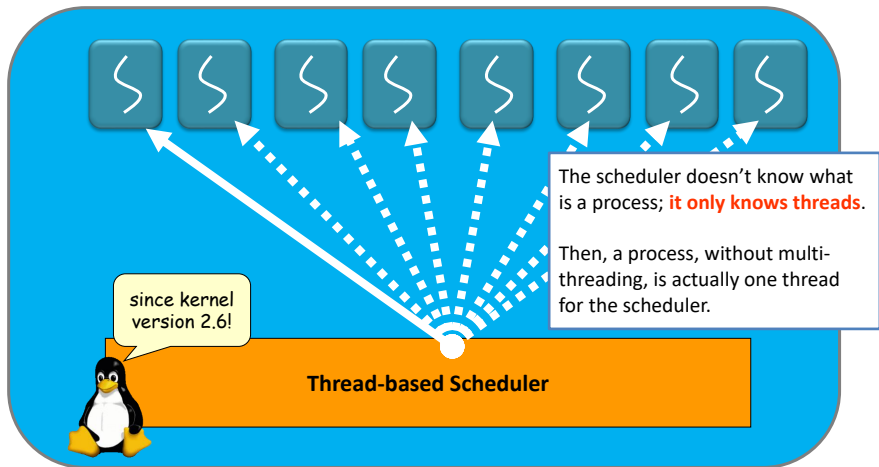
Scheduling – why & who cares?

- If a scheduler only interests in **processes**...



Scheduling – why & who cares?

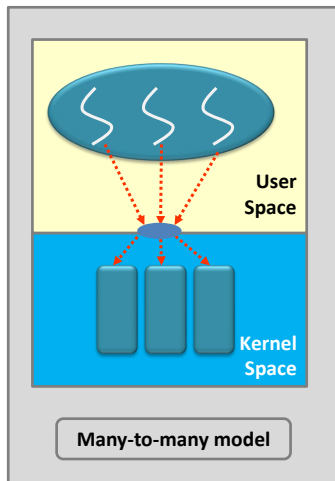
- If a scheduler only interests in **threads**...



Thread models

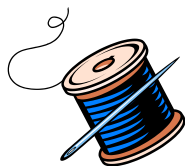
- **Many-to-many Model**

- Multiple threads are mapped to multiple structures (group mapping)
- Merit:
 - Create as many threads as necessary
 - Also have a high degree of concurrency



Multi-threading

- Motivation
- Thread Concept
- Thread Models
- Basic Programming



Thread Libraries

- A thread library provides the programmer with an **API** for creating and managing threads
 - Two ways of implementation: User-level or kernel-level
- Three main thread libraries
 - POSIX Pthreads (user-level or kernel-level)
 - Windows (kernel-level)
 - Java (implemented using Windows API or Pthreads)

Creating Multiple Threads

- Asynchronous threading
 - Parent resumes execution after creating a child
 - Parent and child execute **concurrently**
 - Each thread runs independently
 - Little data sharing
- Synchronous threading
 - Fork-join strategy: Parent waits for children to terminate
 - Significant data sharing

The Pthreads Library

- **Pthreads**: POSIX standard defining an API for thread creation and synchronization.
 - Specification, not implementation
- How to use Pthreads?

	Process	Thread
Creation	<code>fork()</code>	<code>pthread_create()</code>
I.D. Type	PID, an integer	“pthread_t”, a structure
Who am I?	<code>getpid()</code>	<code>pthread_self()</code>
Termination	<code>exit()</code>	<code>pthread_exit()</code>
Wait for child termination	<code>wait()</code> or <code>waitpid()</code>	<code>pthread_join()</code>
Kill?	<code>kill()</code>	<code>pthread_kill()</code>

ISSUE 1: Thread Creation

Thread creation – `pthread_create()`

Thread Function

```
1 void * hello( void *input ) {  
2     printf("%s\n", (char *) input);  
3     pthread_exit(NULL);  
4 }
```

Main Function

```
5 int main(void) {  
6     pthread_t tid;  
7     pthread_create(&tid, NULL, hello, "hello world");  
8     pthread_join(tid, NULL);  
9     return 0;  
10 }
```

Thread creation – pthread_create()

Thread Function

```
1 void * hello( void *input ) {  
2     printf(“%s\n”, (char *) input);  
3     pthread_exit(NULL);  
4 }
```

Main Function

```
5 int main(void) {  
6     pthread_t tid;  
7     pthread_create(&tid, NULL, hello, “hello world”);  
8     pthread_join(tid, NULL);  
9     return 0;  
10 }
```

At the beginning,
there is only one
thread running: **the
main thread.**



Main Thread

Thread creation – `pthread_create()`

Thread Function

```
1 void * hello( void *input ) {  
2     printf(“%s\n”, (char *) input);  
3     pthread_exit(NULL);  
4 }
```

Main Function

```
5 int main(void) {  
6     pthread_t tid;  
7     pthread_create(&tid, NULL, hello, “hello world”);  
8     pthread_join(tid, NULL);  
9     return 0;  
10 }
```

The hello thread is created!

It is running *“together”* with the main thread.



Main Thread

`pthread_create()`



Hello Thread

Thread creation – `pthread_create()`

Thread Function

```
1 void * hello( void *input ) {  
2     printf("%s\n", (char *) input);  
3     pthread_exit(NULL);  
4 }
```

The `pthread_create()` function allows one argument to be passed to the thread function.

Main Function

```
5 int main(void) {  
6     pthread_t tid;  
7     pthread_create(&tid, NULL, hello, "hello world");  
8     pthread_join(tid, NULL);  
9     return 0;  
10 }
```

This sets the thread function of the to-be-created thread as: `hello()`.

Remember: A thread starts with **one specific function (thread function)**

Thread creation – `pthread_create()`

Thread Function

```
1 void * hello( void *input ) {  
2     printf(“%s\n”, (char *) input);  
3     pthread_exit(NULL);  
4 }
```

Main Function

```
5 int main(void) {  
6     pthread_t tid;  
7     pthread_create(&tid, NULL, hello, “hello world”);  
8     pthread_join(tid, NULL);  
9     return 0;  
10 }
```

Remember `wait()`
and `waitpid()`?

`pthread_join()`
performs similarly.



Blocked

Main Thread



Hello Thread

Thread creation – pthread_create()

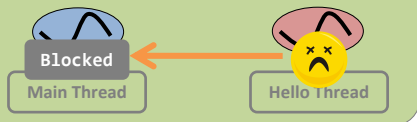
Thread Function

```
1 void * hello( void *input ) {  
2     printf(“%s\n”, (char *) input);  
3     pthread_exit(NULL);  
4 }
```

Main Function

```
5 int main(void) {  
6     pthread_t tid;  
7     pthread_create(&tid, NULL, hello, “hello world”);  
8     pthread_join(tid, NULL);  
9     return 0;  
10 }
```

Termination of the target thread causes **pthread_join()** to return.



ISSUE 2: Passing parameters

Thread creation – passing parameter

Thread Function

```
1 void * do_your_job( void *input ) {  
2     printf("child = %d\n", *( (int *) input) );  
3     *((int *) input) = 20;  
4     printf("child = %d\n", *( (int *) input) );  
5     pthread_exit(NULL);  
6 }
```

Main Function

```
7 int main(void) {  
8     pthread_t tid;  
9     int input = 10;  
10    printf("main = %d\n", input);  
11    pthread_create(&tid, NULL, do_your_job, &input);  
12    pthread_join(tid, NULL);  
13    printf("main = %d\n", input);  
14    return 0;  
15 }
```

Guess: What is the output?

```
$ ./pthread_evil_1  
main = 10  
child = 10  
child = 20  
main = 20  
$
```



Each thread has a separated stack.

Why do we have such results?

Thread creation – passing parameter

Well, we all know that the local variable “**input**” is in the stack for the main thread.

```
1 void * do_your_job( void *input ) {
2     printf("child = %d\n", *(int *) input);
3     *((int *) input) = 20;
4     printf("child = %d\n", *(int *) input);
5     pthread_exit(NULL);
6 }

7 int main(void) {
8     pthread_t tid;
9     int input = 10;
10    printf("main = %d\n", input);
11    pthread_create(&tid, NULL, do_your_job, &input);
12    pthread_join(tid, NULL);
13    printf("main = %d\n", input);
13    return 0;
14 }
```

Local
(main thread)

Dynamic

Global

Code

Thread creation – passing parameter

Yet...the stack for the new thread is not on another process, but is on the same piece of user-space memory as the main thread.

```
1 void * do_your_job( void *input ) {  
2     printf("child = %d\n", *( (int *) input) );  
3     *((int *) input) = 20;  
4     printf("child = %d\n", *( (int *) input) );  
5     pthread_exit(NULL);  
6 }  
  
7 int main(void) {  
8     pthread_t tid;  
9     int input = 10;  
10    printf("main = %d\n", input);  
11    pthread_create(&tid, NULL, do_your_job, &input);  
12    pthread_join(tid, NULL);  
13    printf("main = %d\n", input);  
13    return 0;  
14 }
```

Local
(new thread)

Local
(main thread)

Dynamic

Global

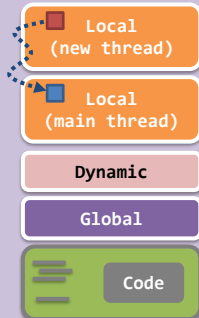
Code

Thread creation – passing parameter

The `pthread_create()` function only passes an **address** to the new thread. Worse, the address is pointing to a variable in the stack of the main thread!

```
1 void * do_your_job( void *input ) {
2     printf("child = %d\n", *( (int *) input) );
3     *((int *) input) = 20;
4     printf("child = %d\n", *( (int *) input) );
5     pthread_exit(NULL);
6 }

7 int main(void) {
8     pthread_t tid;
9     int input = 10;
10    printf("main = %d\n", input);
11    pthread_create(&tid, NULL, do_your_job, &input);
12    pthread_join(tid, NULL);
13    printf("main = %d\n", input);
13    return 0;
14 }
```



Thread creation – passing parameter

Therefore, the new thread can change the value in the main thread, and vice versa.

```
1 void * do_your_job( void *input ) {  
2     printf("child = %d\n", *( (int *) input) );  
3     *((int *) input) = 20;  
4     printf("child = %d\n", *( (int *) input) );  
5     pthread_exit(NULL);  
6 }  
  
7 int main(void) {  
8     pthread_t tid;  
9     int input = 10;  
10    printf("main = %d\n", input);  
11    pthread_create(&tid, NULL, do_your_job, &input);  
12    pthread_join(tid, NULL);  
13    printf("main = %d\n, input);  
13    return 0;  
14 }
```

Local
(new thread)

Local
(main thread)

Dynamic

Global

Code

ISSUE 3: Multiple Threads

Thread creation – multiple threads


Thread Function

```
1 void * do_your_job(void *input) {  
2     int id = *((int *) input);  
3     printf("My ID number = %d\n", id);  
4     pthread_exit(NULL);  
5 }
```

Main Function

```
6 int main(void) {  
7     int i;  
8     pthread_t tid[5];  
9  
10    for(i = 0; i < 5; i++)  
11        pthread_create(&tid[i], NULL, do_your_job, &i);  
12    for(i = 0; i < 5; i++)  
13        pthread_join(tid[i], NULL);  
14    return 0;  
15 }
```

Waiting on several threads: enclose pthread_join() within a for loop



ISSUE 4: Return Value

Thread termination – passing return value

Thread Function

```
1 void * do_your_job(void *input) {  
2     int *output = (int *) malloc(sizeof(int));  
3     srand(time(NULL));  
4     *output = ((rand() % 10) + 1) * (*(int *) input);  
5     pthread_exit(output);  
6 }
```

`void pthread_exit(void *return_value);`

Together with termination, a pointer to a **global variable or a piece of dynamically allocated memory** is returned to the main thread.

Main Function

```
7 int main(void) {  
8     pthread_t tid;  
9     int input = 10, *output;  
10    pthread_create(&tid, NULL, do_your_job, &input);  
11    pthread_join(tid, (void **) &output);  
12    return 0;  
13 }
```

Using pass-by-reference, a pointer to the result is received in the main thread.

Other Libraries

- For Windows threads and Java threads, you can refer to the textbook if you are interested in.

Multi-threading

- Motivation
- Thread Concept
- Thread Models
- Basic Programming
- Implicit Threading



Implicit Threading

- Applications are containing hundreds or even thousands of threads
 - Program correctness is more difficult with explicit threads
- How to address the programming difficulties?
 - Transfer the creation and management of threading from programmers to compilers and run-time libraries
 - Implicit threading
- We will introduce two methods
 - Thread Pools
 - OpenMP

Thread Pools

- Problems with multithreaded servers
 - Time required to create threads, which will be discarded once completed their work
 - Unlimited threads could exhaust the system resources
- How to solve?
 - Thread pool
 - Idea
 - Create a number of threads in a pool where they wait for work
 - Procedure
 - Awakens a thread if necessary
 - Returns to the pool after completion
 - Waits until one becomes free if the pool contains no available thread

Thread Pools

- Advantages
 - Usually slightly faster to service a request with an existing thread than create a new thread
 - Allows the number of threads in the application(s) to be bound to the size of the pool

OpenMP

- Provides support for parallel programming in shared-memory environments
- Set of compiler directives and an API for C, C++, FORTRAN
- Identifies **parallel regions** – blocks of code that can run in parallel

Parallel for loop

```
#pragma omp parallel for
for(i=0;i<N;i++) {
    c[i] = a[i] + b[i];
}
```

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    /* sequential code */

    #pragma omp parallel
    {
        printf("I am a parallel region.");
    }

    /* sequential code */

    return 0;
}
```

When OpenMP encounters the directive, it creates as many threads as there are processing cores

Multi-threading

- Motivation
- Thread Concept
- Thread Models
- Basic Programming
- Implicit Threading
- Threading Issues



Semantics of `fork()` and `exec()`

- Two key system calls for processes: **fork**, **exec**
- **fork()**: Some UNIX systems have two versions
 - The new process duplicates all threads, or
 - Duplicates only the thread that invoked **fork()**
- **exec()**: usually works as normal
 - Replace the running process - including **all threads**

Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred
 - Synchronous signal and asynchronous signal
 - Default handler or user-defined handler
- Where should a signal be delivered in multi-threaded program?
 - Deliver the signal to the thread to which the signal applies
 - Deliver the signal to every thread in the process
 - Deliver the signal to certain threads in the process
 - Assign a specific thread to receive all signals for the process
- Deliver a signal to a specified thread with Pthread
 - `pthread_kill(pthread_t tid, int signal)`

Thread Cancellation

- Terminating a thread before it has finished
 - Why needed?
 - Example: Close a browser when multiple threads are loading images
- Two general approaches
 - **Asynchronous cancellation** terminates the target thread immediately
 - Problem: Troublesome when canceling a thread which is updating data shared by other threads
 - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled (can be canceled safely)

Thread Cancellation (Cont.) - Pthreads

- Pthreads code example
 - **pthread_cancel()**
 - Indicates only a request

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

- Three cancellation modes

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- Default: deferred
 - Cancellation occurs only when it reaches a cancellation point, can be established by **pthread_testcancel()**

Thread-Local Storage

- Some applications, each thread may need its own copy of certain data
 - Transaction processing system: service each transaction (with a unique identifier) in a thread
 - How about local variables?
 - Visible only during a single function invocation
- Thread-local storage (TLS) allows each thread to have its own copy of data
 - TLS is **visible across function invocations**
 - Similar to **static** data
 - TLS data are unique to each thread

Summary of Threads

- Virtually all modern OSes support multi-threading
 - A thread is a basic unit of CPU utilization
 - Each comprises a thread ID, a program counter, a register set, and a stack
 - All threads within a process share code section, data section, other resources like open files and signals
- You should **take great care** when writing multi-threaded programs
- You also have to take care of (will be talked later):
 - Mutual exclusion and
 - Synchronization

End of Chapter 4

Operating Systems

Associate Prof. Yongkun Li

中科大-计算机学院 副教授

<http://staff.ustc.edu.cn/~ykli>

Ch5

Process Communication & Synchronization

Story so far...

- Process concept + operations
 - Programmer's perspective + kernel's perspective
- Thread
 - Lightweight process
- We mainly talked about the stuffs related to a single process/thread, what if multiple processes exist...

Processes

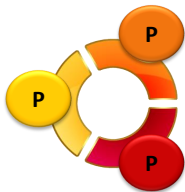
- The processes within a system may be
 - ***independent*** or
 - Independent process cannot affect or be affected by other processes
 - ***cooperating***
 - Cooperating process can affect or be affected by other processes
- Note: Any process that shares data with others is a cooperating process

Cooperating Processes

- Why we need cooperating processes
 - Information sharing
 - e.g., shared file
 - Computation speedup
 - executing subtasks in parallel
 - Modularity
 - dividing system functions into separate processes
 - Convenience

Inter-process communication (IPC)

- What and how?



Interprocess Communication

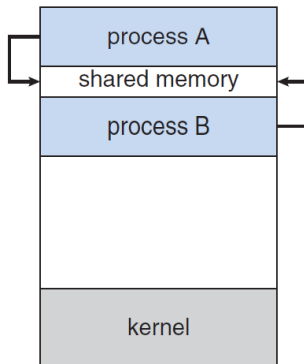
- IPC: used for exchanging data between processes
- Cooperating processes need
 - **interprocess communication (IPC)** for exchanging data
- Paradigm for cooperating processes
 - **Producer-consumer problem**, useful metaphor for many applications (abstracted problem model)
 - **producer** process produces information that is consumed by a **consumer** process
 - At least one producer and one consumer

Two models

- Two (abstracted) models of IPC

- Shared memory**

- Establish a shared memory region, read/write to shared region
 - Accesses are treated as routine memory accesses
 - Faster

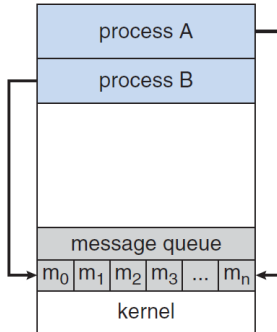


Two models

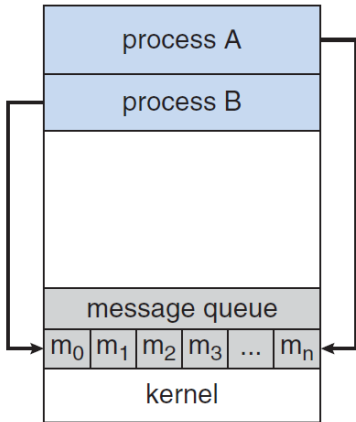
- Two (abstracted) models of IPC

- **Message passing**

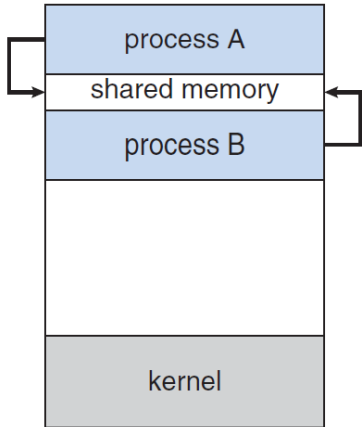
- Exchange message
- Require kernel intervention
- Easier to implement in distributed system



Communications Models



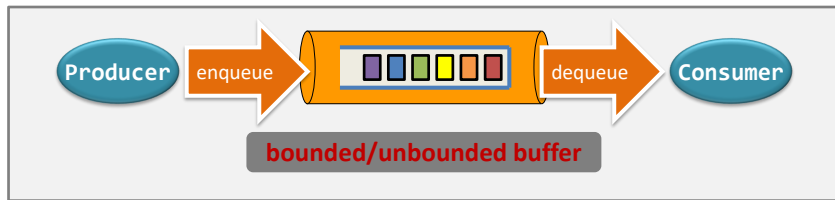
Message passing



Shared memory

Producer-Consumer Problem

- Shared memory solution
 - A buffer is needed to allow processes to run concurrently



A buffer	<ul style="list-style-type: none">-It is a shared object;-It is a queue (imagine that it is an array implementation of queue).
A producer process	<ul style="list-style-type: none">-It produces a unit of data, and-writes that a piece of data to the tail of the buffer at one time.
A consumer process	<ul style="list-style-type: none">-It removes a unit of data from the head of the bounded buffer at one time.

Producer-Consumer Problem

- Focus on bounded buffer: what are the requirements?

Producer-consumer requirement #1

When the producer wants to
(a) put a new item in the buffer, but
(b) **the buffer is already full...**

Then,

- (1) **The producer should be suspended**, and
- (2) **The consumer should wake the producer up** after she has dequeued an item.

Producer-consumer requirement #2

When the consumer wants to
(a) consumes an item from the buffer, but
(b) **the buffer is empty...**

Then,

- (1) **The consumer should be suspended**, and
- (2) **The producer should wake the consumer up** after she has enqueued an item.

Producer-consumer solution (shared mem)

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Shared memory by producer
& consumer processes



out (consumer) in (producer)

**Only allows BUFFER_SIZE-1
items at the same time. Why?**

```
item next_produced;
while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

Producer

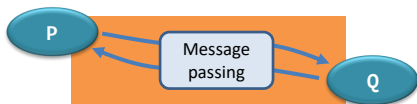
```
item next_consumed;
while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item */
}
```

Consumer

Message Passing

- Communicating processes may reside on different computers connected by a network
- IPC facility provides two operations:
 - **send**(message) + **receive**(message)
- If processes *P* and *Q* wish to communicate
 - Establish a **communication link** between them
 - Exchange messages via send/receive



Message Passing (Cont.)

- Implementation issues (logical):
 - Naming: Direct/indirect communication
 - Synchronization: Synchronous/asynchronous
 - Buffering

Naming

- How to refer to each other?
- **Direct communication**: explicitly name each other
 - Operations (symmetry)
 - **send** (Q , *message*) – send a message to process Q
 - **receive**(P , *message*) – receive a message from process P
 - Properties of communication link
 - Links are established automatically (every pair can establish)
 - A link is associated with exactly one pair of processes
 - Between each pair, there exists exactly one link
 - Disadvantage: limited modularity (hard-coding)

Naming

- How to refer to each other?
- **Indirect communication**: sent to and received from mailboxes (ports)
 - Operations
 - **send** (*A, message*) – send a message to mailbox A
 - **receive**(*A, message*) – receive a message from mailbox A
 - Properties of communication link
 - A link is established between a pair of processes only if both members have a shared mailbox
 - A link may be associated with more than two processes
 - Between each pair, a number of different links may exist

Issues of Indirect Communication

- ISSUE1: Who receives the message when multiple processes are associated with one link?

- Who gets the message?



- Policies

- Allow a link to be associated with at most two processes
 - Allow only one process at a time to execute a receive operation
 - Allow the system to select arbitrarily the receiver (based on an algorithm). Sender is notified who the receiver was.

- ISSUE2: Who owns the mailbox?

- The process (ownership may be passed)
 - The OS (need a method to create, send/receive, delete)

Synchronization

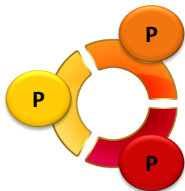
- How to implement send/receive?
 - **Blocking** is considered **synchronous**
 - **Blocking send** - the sender is blocked until the msg is received
 - **Blocking receive** - the receiver is blocked until a msg is available
 - **Non-blocking** is considered **asynchronous**
 - **Non-blocking send** - the sender sends the message and resumes
 - **Non-blocking receive** - the receiver receives a valid msg or null
- Different combinations are possible
 - When both send and receive are blocking, we have a *rendezvous* between the processes.
 - Other combinations need *buffering*.

Buffering

- Different combinations are possible
 - When both send and receive are blocking, we have a *rendezvous* between the processes.
 - Other combinations need *buffering*.
- Messages reside in a temporary queue, which can be implemented in three ways
 - **Zero capacity** – no messages are queued on a link, sender must wait for receiver (no buffering)
 - **Bounded capacity** – finite length of n messages, sender must wait if link is full
 - **Unbounded capacity** – infinite length, sender never waits

Inter-process communication (IPC)

- What and how?
- POSIX shared memory



POSIX Shared Memory

- POSIX shared memory is organized using memory-mapped file
 - Associate the region of shared memory with a file
- Illustrate with the producer-consumer problem
 - Producer
 - Consumer

POSIX Shared Memory

- Producer

- Create a shared-memory object

- `shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);`

Name of the shared memory object

Create the object if it does not exist

Open for reading & writing

Directory permissions

POSIX Shared Memory

- Producer

- Create a shared-memory object

- `shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);`

- Configure object size

- `ftruncate(shm_fd, SIZE);`

File descriptor for the shared mem. Obj.

Size of the shared-memory object

POSIX Shared Memory

- Producer

- Create a shared-memory object

- `shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);`

- Configure object size

- `ftruncate(shm_fd, SIZE);`

- Establish a memory-mapped file containing the object

- `ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);`

Allows writing to the object
(only writing is necessary for producer)

Changes to the shared-memory object will
be visible to all processes sharing the object

POSIX Shared Memory

- Consumer

- Open the shared-memory object

- `shm_fd = shm_open(name, O_RDONLY, 0666);`

Open for read only

POSIX Shared Memory

- Consumer

- Open the shared-memory object

- `shm_fd = shm_open(name, O_RDONLY, 0666);`

- Memory map the object

- `ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);`

Allows reading to the object
(only reading is necessary for consumer)

POSIX Shared Memory

- Consumer
 - Open the shared-memory object
 - `shm_fd = shm_open(name, O_RDONLY, 0666);`
 - Memory map the object
 - `ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);`
 - Remove the shared memory object
 - `shm_unlink(name);`

POSIX Shared Memory – Complete Solution

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
```

Producer

```
int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* strings written to shared memory */
    const char *message_0 = "Hello";
    const char *message_1 = "World!";

    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory object */
    void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr, "%s", message_0);
    ptr += strlen(message_0);
    sprintf(ptr, "%s", message_1);
    ptr += strlen(message_1);

    return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
```

Consumer

```
int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory object */
    void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s", (char *)ptr);

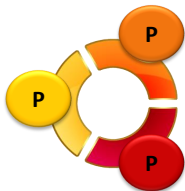
    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

Direct access to the shared memory region

Inter-process communication (IPC)

- What and how?
- POSIX shared memory
- Sockets

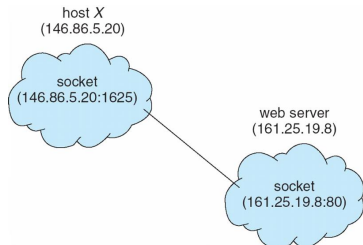


Sockets

- A **socket** is defined as an endpoint for communication (over a network)
 - A pair of processes employ a pair of sockets
 - A socket is identified by an **IP address** and a **port** number
 - All ports below 1024 are used for standard services
 - telnet server listens to port 23
 - FTP server listens to port 21
 - HTTP server listens to port 80

Sockets

- Socket uses a client-server architecture
 - Server waits for incoming client requests by listening to a specific port
 - Accepts a connection from the client socket to complete the connection
- All connections must be unique
 - Establishing a new connection on the same host needs another port (>1024)
- Special IP address 127.0.0.1 (**loopback**) refers to itself
 - Allow a client and server on the same host to communicate using the TCP/IP protocol



Example in Java

- Three types of sockets
 - Connection-oriented (TCP), Connectionless (UDP), Multicast** – data can be sent to multiple recipients

```
import java.net.*;
import java.io.*;

public class DateServer
{
    public static void main(String[] args) {
        try {
            ServerSocket sock = new ServerSocket(6013);
            /* now listen for connections */
            while (true) {
                Socket client = sock.accept();

                PrintWriter pout = new
                    PrintWriter(client.getOutputStream(), true);

                /* write the Date to the socket */
                pout.println(new java.util.Date().toString());

                /* close the socket and resume */
                /* listening for connections */
                client.close();
            }
        } catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

```
import java.net.*;
import java.io.*;

public class DateClient
{
    public static void main(String[] args) {
        try {
            /* make connection to server socket */
            Socket sock = new Socket("127.0.0.1", 6013);

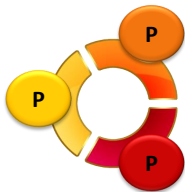
            InputStream in = sock.getInputStream();
            BufferedReader bin = new
                BufferedReader(new InputStreamReader(in));

            /* read the date from the socket */
            String line;
            while ( (line = bin.readLine()) != null)
                System.out.println(line);

            /* close the socket connection*/
            sock.close();
        } catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

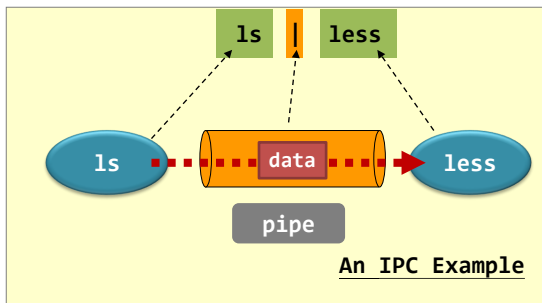
Inter-process communication (IPC)

- What and how?
- POSIX shared memory
- Sockets
- Pipes



What is pipe?

- Pipe is a **shared object**.
 - Using pipe is a way to realize IPC.
 - Acts as a conduit allowing two processes to communicate.



Pipes

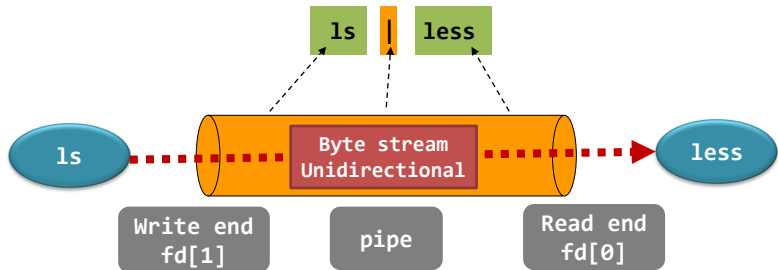
- Four issues:
 - Is the communication unidirectional or bidirectional?
 - In the case of two-way communication, is it half or full-duplex?
 - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
 - Can the pipes be used over a network?
- Two common pipes
 - Ordinary pipes and named pipes

Ordinary Pipes

- Ordinary pipes (no name in file system)
 - Ordinary pipes are used only for related processes (**parent-child relationship**)
 - Processes must reside on the same machine
 - Ordinary pipes are **unidirectional** (one-way communication)
 - Ceases to exist after communication has finished
- Ordinary pipes allow communication in standard producer-consumer style
 - Producer writes to one end (**write-end**)
 - Consumer reads from the other end (**read-end**)

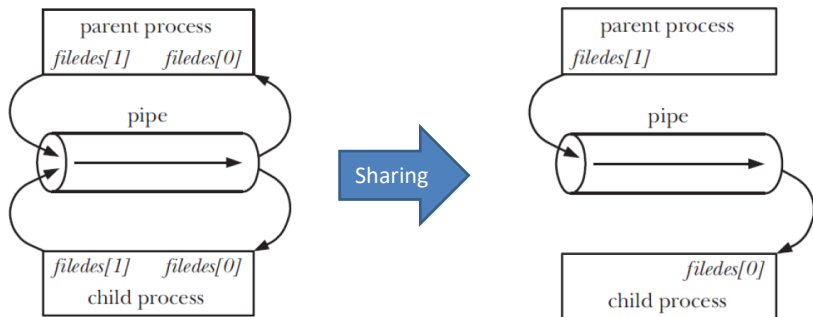
UNIX Pipe

- UNIX treats a pipe as a special file (child inherits it from parent)
 - Create: `pipe(int fd[]);`
 - `fd[0]`: read end
 - `fd[1]`: write end
 - Access: Ordinary `read()` and `write()` system calls



UNIX Pipe

- Pipes are anonymous (no name in file system), then how to share?
 - **fork()** duplicates parent's file descriptors
 - Parent and child use each end of the pipe



UNIX Pipe

```
/* fork a child process */  
pid = fork();
```

Create a child process

```
if (pid < 0) { /* error occurred */  
    fprintf(stderr, "Fork Failed");  
    return 1;  
}
```

```
if (pid > 0) { /* parent process */  
    /* close the unused end of the pipe */  
    close(fd[READ_END]);  
  
    /* write to the pipe */  
    write(fd[WRITE_END], write_msg, strlen(write_msg)+1)  
  
    /* close the write end of the pipe */  
    close(fd[WRITE_END]);  
}
```

Parent process
Use the write end only

```
else { /* child process */  
    /* close the unused end of the pipe */  
    close(fd[WRITE_END]);  
  
    /* read from the pipe */  
    read(fd[READ_END], read_msg, BUFFER_SIZE);  
    printf("read %s", read_msg);  
  
    /* close the read end of the pipe */  
    close(fd[READ_END]);  
}
```

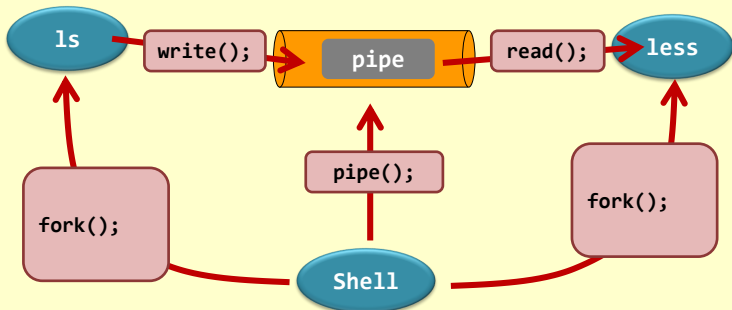
unidirectional (one-
way communication)

Child process
Use the read end only

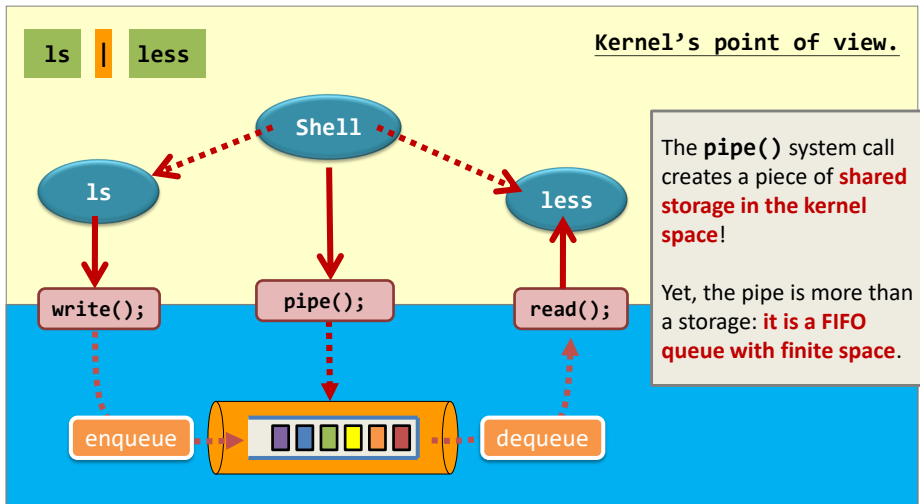
Pipe - Shell Example

ls | less

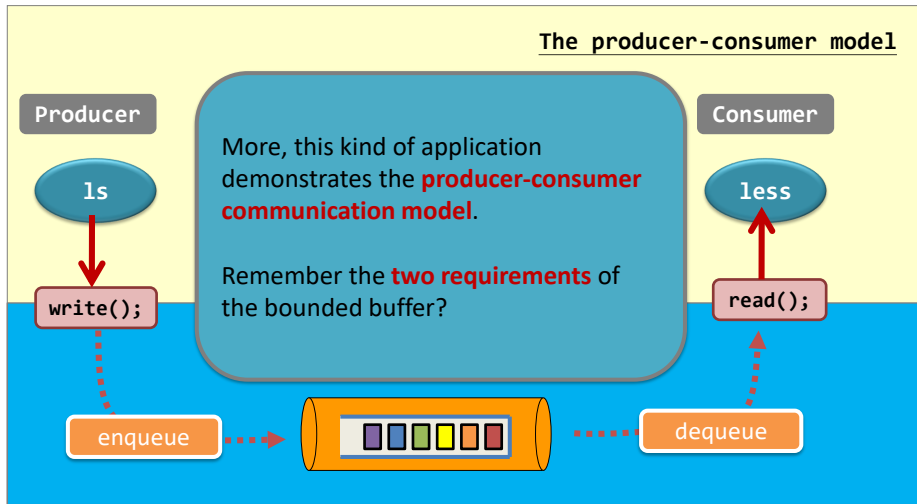
Programmer's point of view.



Pipe – Shell Example



Pipe – Shell Example



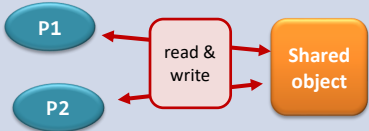
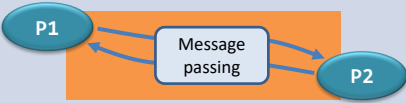
Named Pipes

- Named pipes (pipe with name in file system)
 - No parent-child relationship is necessary (processes must reside on the same machine)
 - Several processes can use the named pipe for communication (may have several writers)
 - Continue to exist until it is explicitly deleted
 - Communication is bidirectional (still half-duplex)
- Named pipes are referred to as **FIFOs** in UNIX
 - Treated as typical files
 - `mkfifo()`, `open()`, `read()`, `write()`, `close()`

Story so far...

- Interprocess communication (IPC)
 - Necessary for cooperating processes
 - Producer-consumer model
- IPC models
 - Shared memory & message passing
- IPC schemes
 - Shared memory
 - Ordinary pipes (parent-child processes)
 - FIFOs (processes on the same machine)
 - Sockets (intermachine communication)
- More: Michael Kerrisk, “The Linux Programming Interface” (<http://www.man7.org/tlpi/>)

IPC models – another point of view

Shared Objects	Message Passing
 <p>The diagram shows two processes, P1 and P2, represented as blue ovals. They are connected to a central red box labeled 'read & write' by red arrows. This box is then connected to an orange box labeled 'Shared object' by red arrows. This illustrates a shared resource that both processes interact with through a common interface.</p>	 <p>The diagram shows two processes, P1 and P2, represented as blue ovals. They are connected to a central white box labeled 'Message passing' by blue arrows. This box is set against an orange rectangular background. This illustrates a communication channel where data is explicitly sent and received between processes.</p>
<p><u>Challenge.</u> Coordination can only be done by detecting the status of the shared object. <i>E.g., is the pipe empty / full?</i></p>	<p><u>Challenge.</u> Coordination relies on the reliability and the efficiency of the communication medium (and protocol).</p>
<p>E.g., pipes, shared memory, and regular files.</p>	<p>E.g., socket programming, message passing interface (MPI) library.</p>

Operating Systems

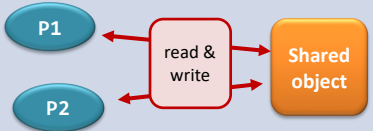
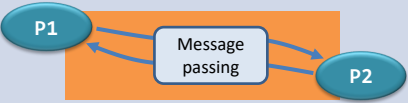
Associate Prof. Yongkun Li

中科大-计算机学院 副教授

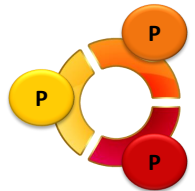
<http://staff.ustc.edu.cn/~ykli>

Ch5 Process Communication & Synchronization -Part 2

Summary on IPC models – another point of view

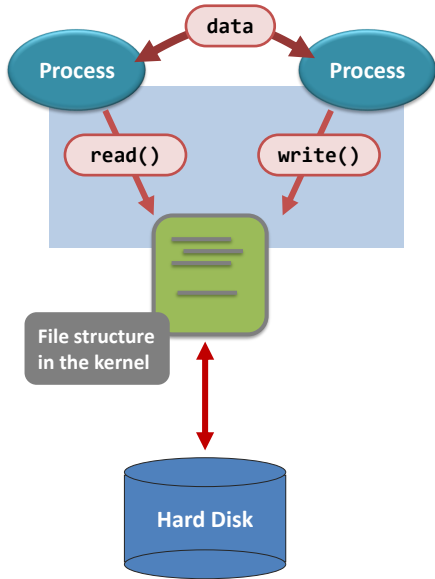
Shared Objects	Message Passing
 <p>The diagram shows two processes, P1 and P2, represented as blue ovals. They are connected to a central red box labeled 'read & write' by red arrows. This box is then connected to an orange box labeled 'Shared object' by red arrows. This illustrates a shared resource that both processes can access directly.</p>	 <p>The diagram shows two processes, P1 and P2, represented as blue ovals. They are connected to a central white box labeled 'Message passing' by blue arrows. This box is set against an orange background. This illustrates communication through a dedicated channel.</p>
<p><u>Challenge.</u> Coordination can only be done by detecting the status of the shared object. <i>E.g., is the pipe empty / full?</i></p>	<p><u>Challenge.</u> Coordination relies on the reliability and the efficiency of the communication medium (and protocol).</p>
<p>E.g., pipes, shared memory, and regular files.</p>	<p>E.g., socket programming, message passing interface (MPI) library.</p>

IPC problem: Race condition



Evil source: the shared objects

- Pipe is implemented with the thought that **there may be more than one process accessing it "at the same time"**
- For shared memory and files, **concurrent access may yield unpredictable outcomes**



Understanding the problem...

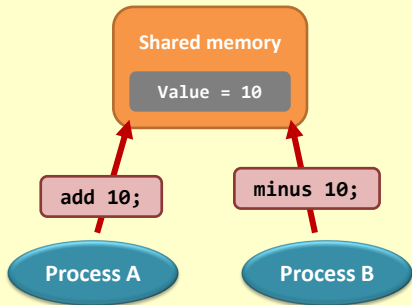
High-level language for Program A

```
1  attach to the shared memory X;  
2  add 10 to X;  
3  exit;
```

High-level language for Program B

```
1  attach to the shared memory X;  
2  minus 10 to X;  
3  exit;
```

The Scenario



Guess what the final result should be?

It may be 10, 0 or 20, can you believe it?

Understanding the problem...

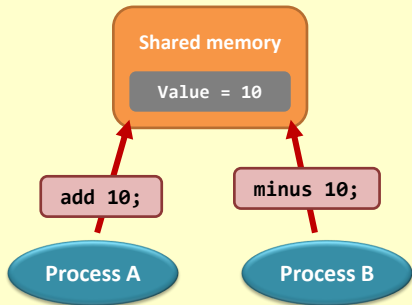
High-level language for Program A

```
1 attach to the shared memory X;  
2 add 10 to X;  
3 exit;
```

High-level language for Program B

```
1 attach to the shared memory X;  
2 minus 10 to X;  
3 exit;
```

The Scenario



Remember the flow of executing a program and the system hierarchy?

Understanding the problem...

High-level language for Program A

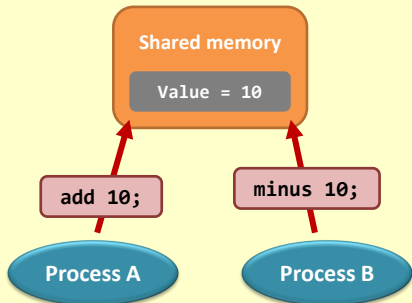
```
1  attach to the shared memory X;  
2  add 10 to X;  
3  exit;
```

This operation
is not atomic

Partial low-level language for Program A

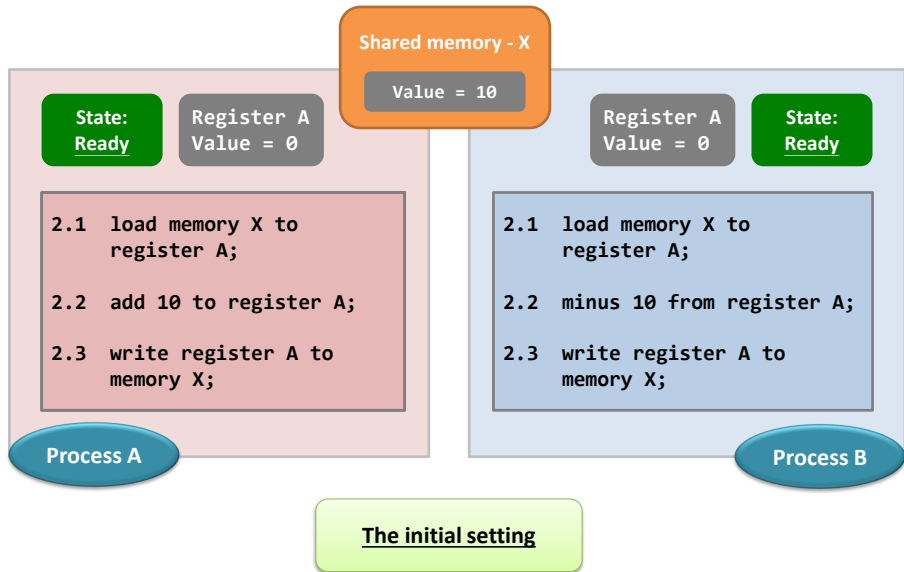
```
1  attach to the shared memory X;  
.....  
2.1 load memory X to register A;  
2.2 add 10 to register A;  
2.3 write register A to memory X;  
.....  
3  exit;
```

The Scenario



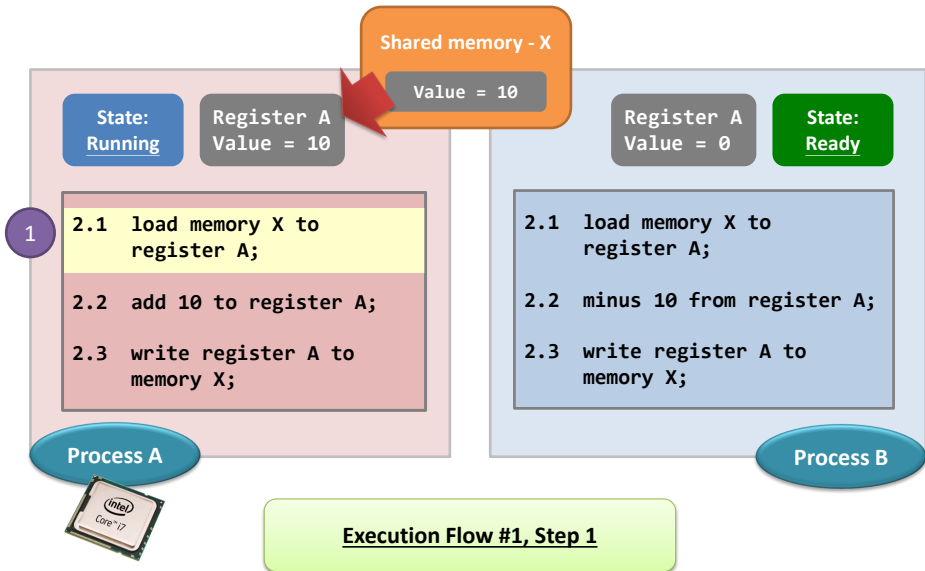
Guess what? This code block is evil!

Understanding the problem...

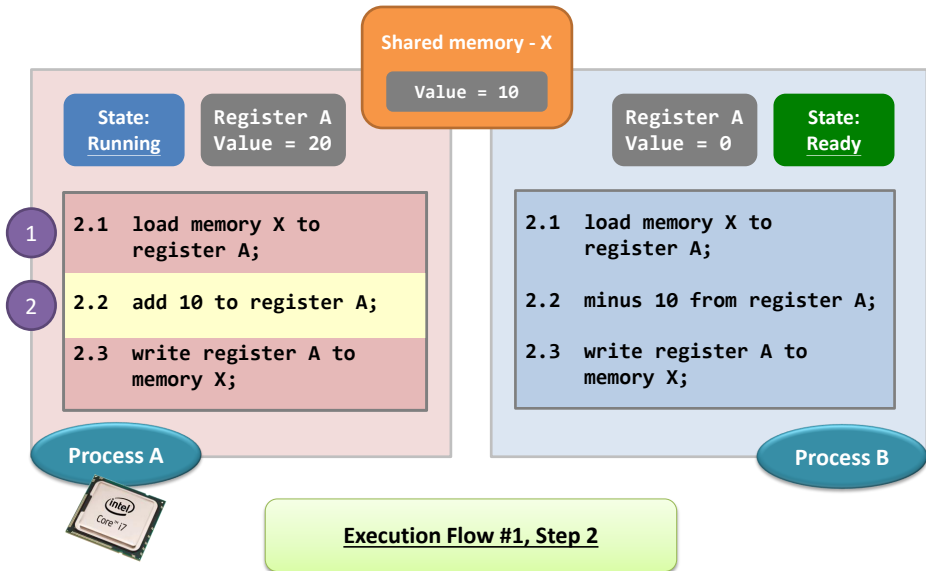


Execution Flow #1

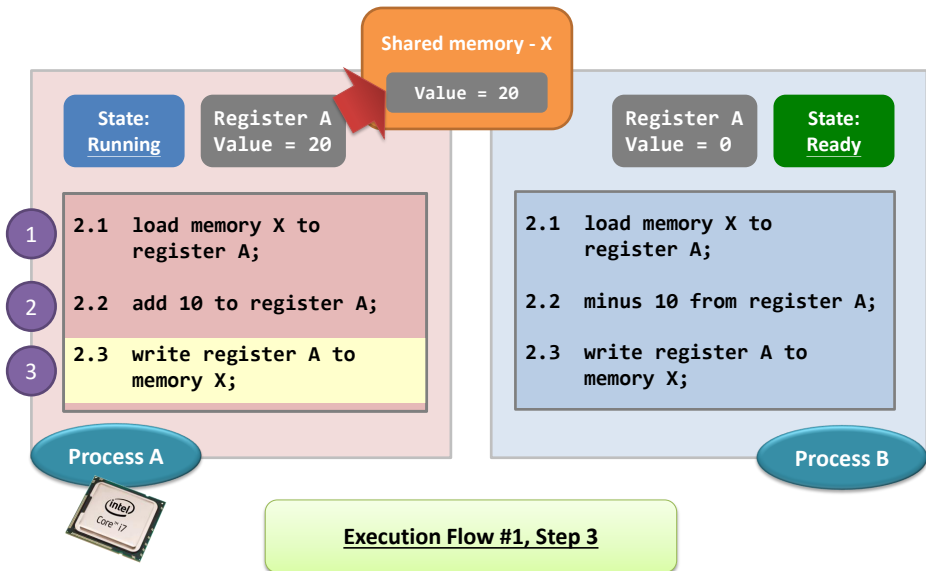
Problem not yet arise...



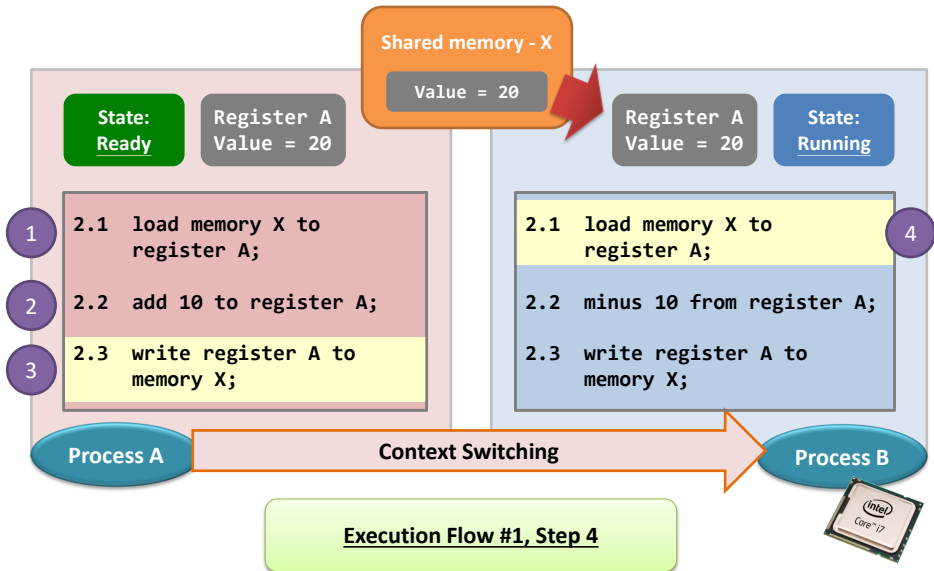
Problem not yet arise...



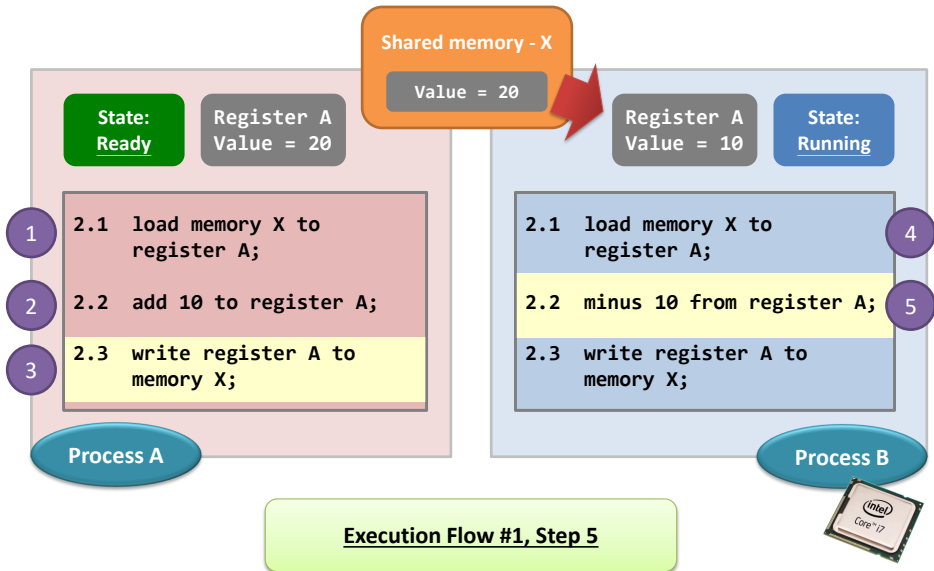
Problem not yet arise...



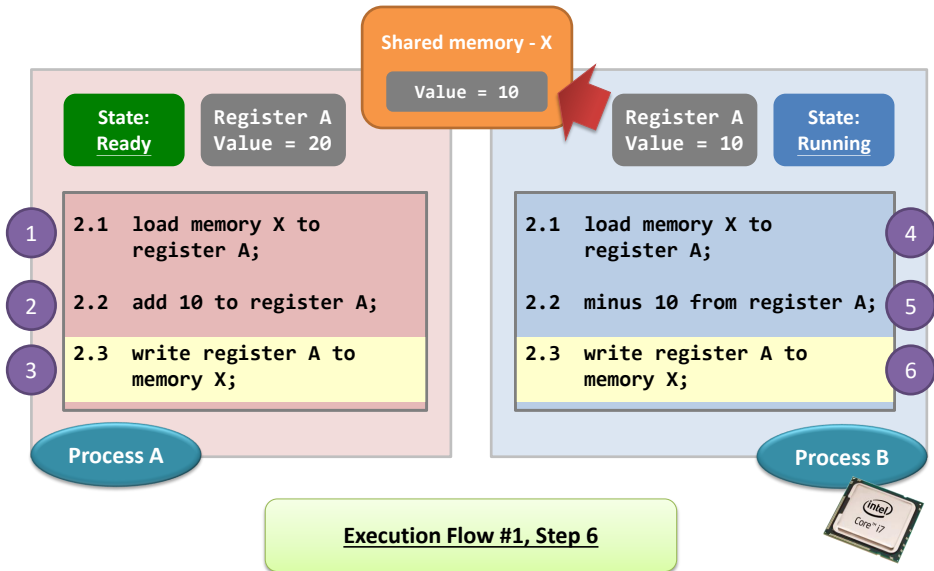
Problem not yet arise...



Problem not yet arise...

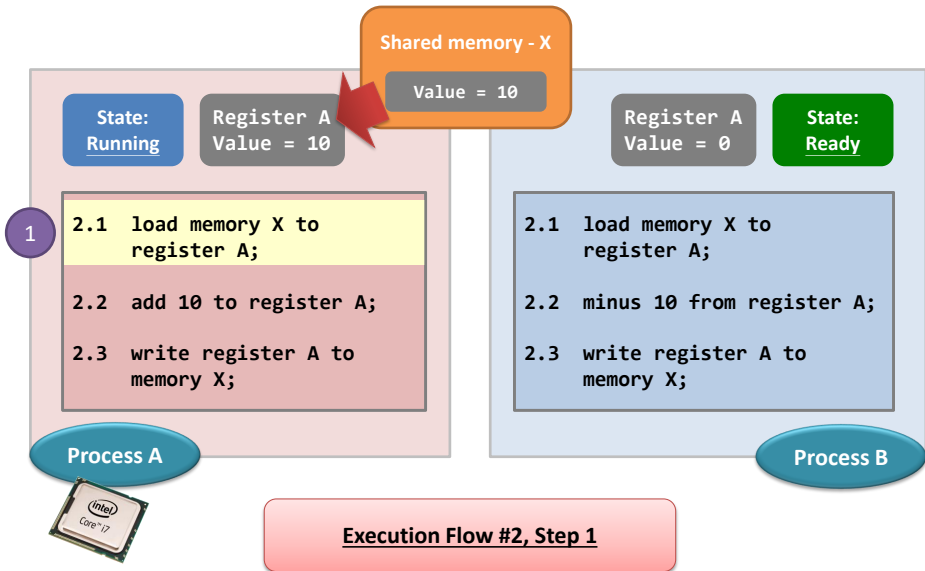


Problem not yet arise...

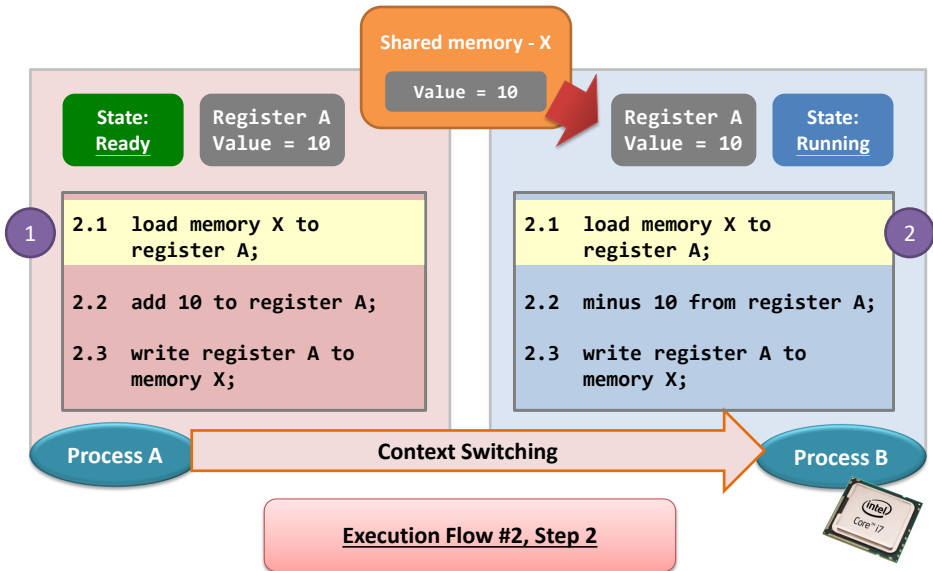


Execution Flow #2

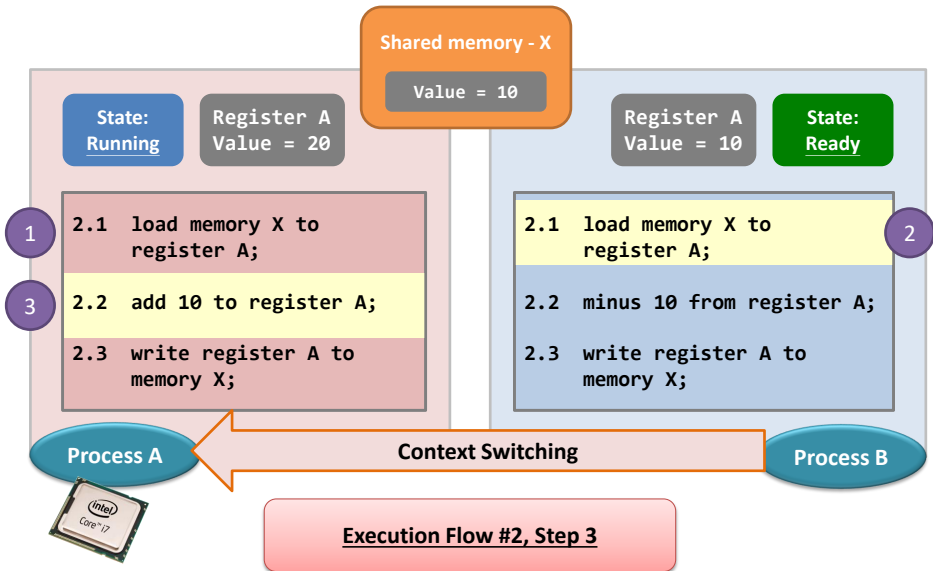
Problem arise...



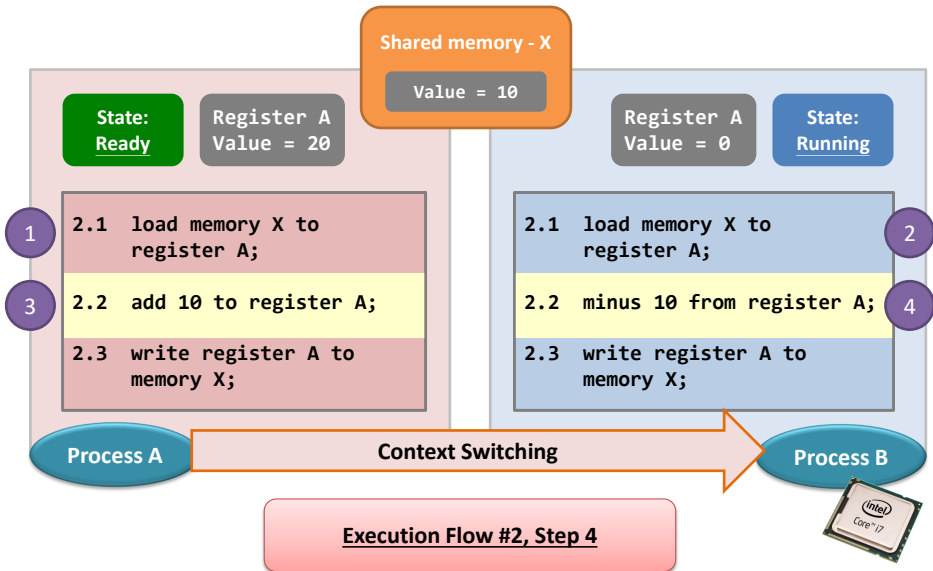
Problem arise...



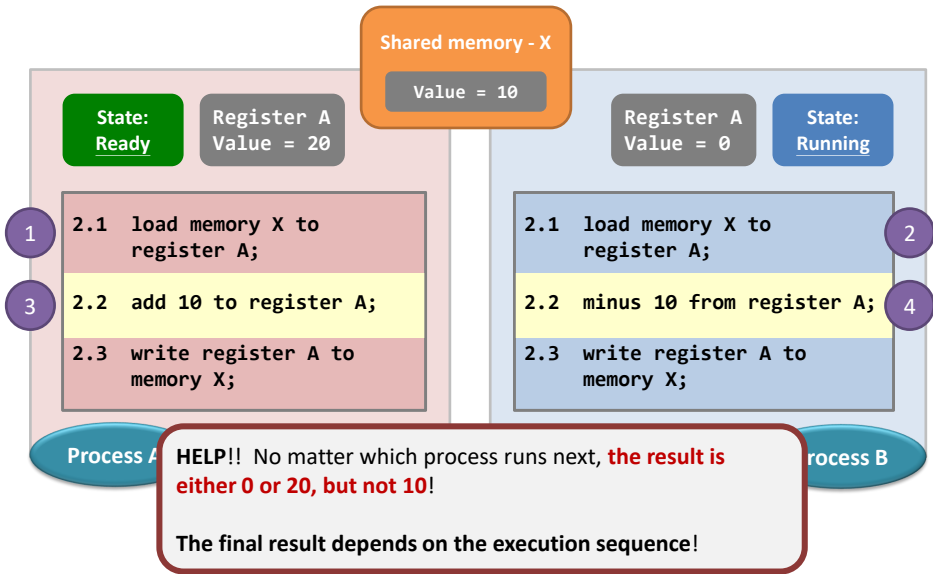
Problem arise...



Problem arise...



Problem arise...

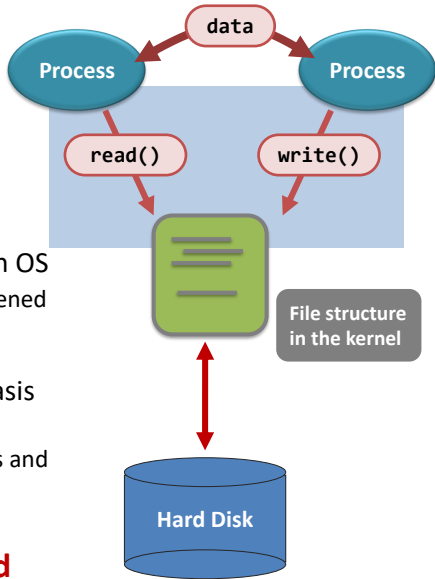


Race condition – the curse

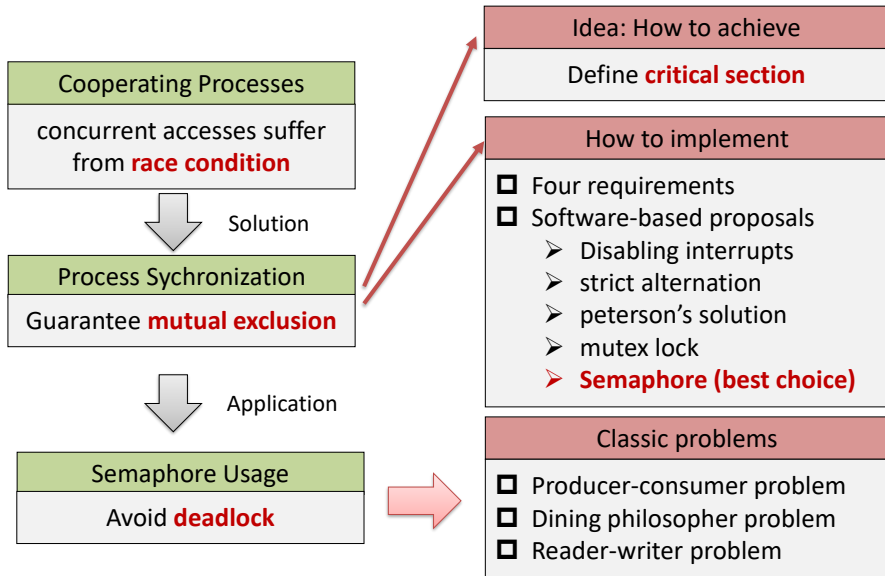
- The above scenario is called the **race condition**.
- A **race condition** means
 - the outcome of an execution depends on a particular order in which the shared resource is accessed.
- Remember: race condition is always a bad thing and debugging race condition has no fun at all!
 - It may end up ...
 - 99% of the executions are fine.
 - 1% of the executions are problematic.

Race condition – the curse

- For shared memory and files, **concurrent access may yield unpredictable outcomes**
 - Race condition
- Common situation
 - Resource sharing occurs frequently in OS
 - EXP: Kernel DS maintaining a list of opened files, maintaining memory allocation, process lists...
 - Multicore brings an increased emphasis on multithreading
 - Multiple threads share global variables and dynamically allocated memory
- **Process synchronization is needed**

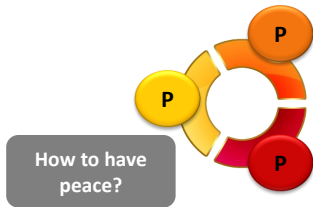


Topics in Process Synchronization

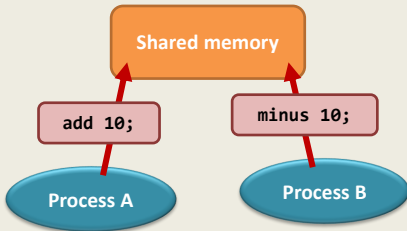


Inter-process communication (IPC)

- Mutual exclusion
 - what & how to achieve?



Mutual Exclusion



Two processes playing with the same shared memory is dangerous.

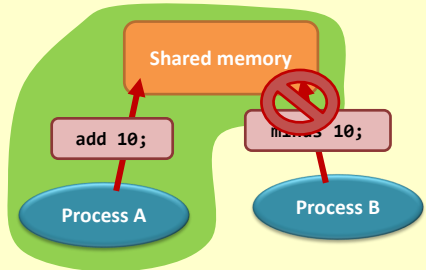
We will face the curse - **race condition**.

The solution can be simple:

When I'm playing with the shared memory, no one could touch it.

This is called **mutual exclusion**.

A set of processes would not have the problem of race condition *if mutual exclusion is guaranteed*.

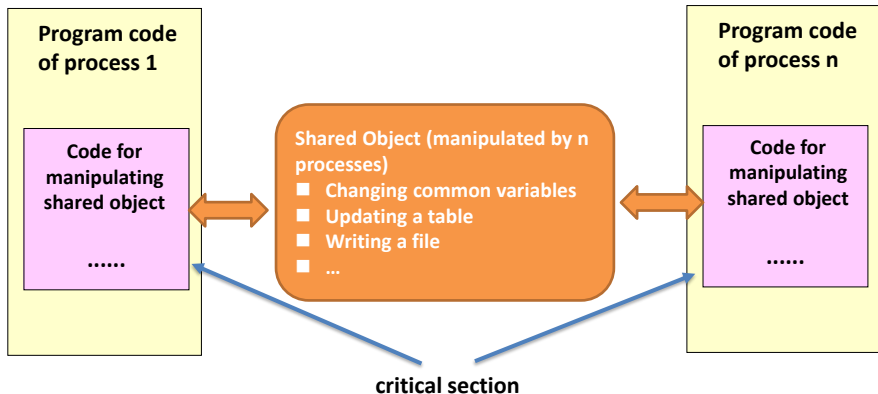


How to realize mutual exclusion?

- Kernel
 - Preemptive kernels and nonpreemptive kernels
 - Allows (not allow) a process to be preempted while it is running in kernel mode
 - A nonpreemptive kernel is essentially free from race conditions on kernel data structures, and also easy to design (especially for SMP architecture)
 - Why would anyone favor a preemptive kernel
 - More responsive
 - More suitable for real-time programming

Mutual Exclusion

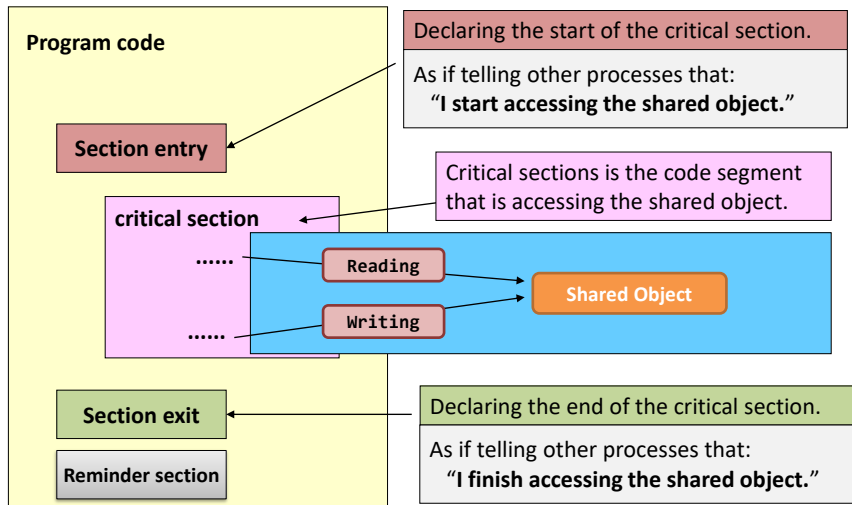
- More generally, how to realize?



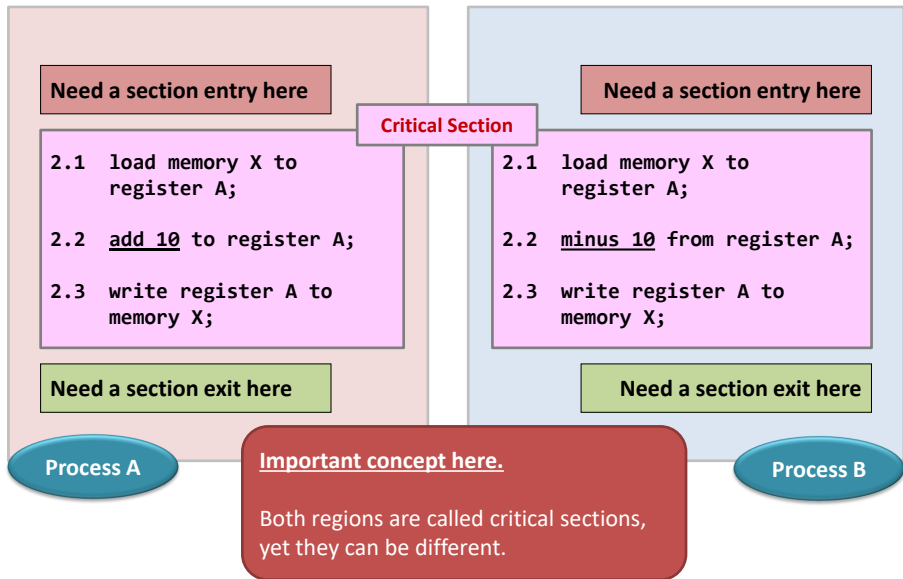
Solution: To guarantee that when one process is executing in its critical section, no other process is allowed execute in its critical section.

Critical Section – General Structure

To guarantee that when one process is executing in its critical section, no other process is allowed execute in its critical section.



Critical Section – Example



Summary...for the content so far...

- **Race condition** is a problem.
 - It makes a concurrent program producing **unpredictable** results if you are using shared objects as the communication medium.
 - The outcome of the computation **totally depends on the execution sequences** of the processes involved.
- **Mutual exclusion** is a requirement.
 - If it could be achieved, then the problem of the race condition would be gone.
 - Mutual exclusion hinders the performance of parallel computations.

Summary...for the content so far...

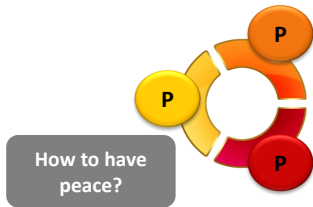
- **Defining critical sections** is a solution.
 - They are code segments that access shared objects.
 - Critical section must be **as tight as possible**.
 - Well, you can declare the entire code of a program to be a big critical section.
 - But, the program will be a very high chance to block other processes or to be blocked by other processes.
 - Note that **one critical section** can be designed for **accessing more than one shared objects**.

Summary...for the content so far...

- **Implementing section entry and exit** is a challenge.
 - The entry and the exit are **the core parts that guarantee mutual exclusion**, but not the critical section.
 - Unless they are correctly implemented, race condition would appear.

Inter-process communication (IPC)

- Mutual exclusion:
 - how to achieve?
 - how to implement?
(section entry and exit)



Entry and exit implementation - requirements

- **Requirement #1: Mutual Exclusion**. No two processes could be simultaneously inside their critical sections.

Implication: when one process is inside its critical section, any attempts to go inside the critical sections by other processes are not allowed.

- **Requirement #2**. Each process is executing at a nonzero speed, but no assumptions should be made about the relative speed of the processes and the number of CPUs.

Implication: the solution **cannot depend on the time spent inside the critical section**, and the solution cannot assume the number of CPUs in the system.

Entry and exit implementation - requirements

- **Requirement #3: progress**. No process running outside its critical section should block other processes.

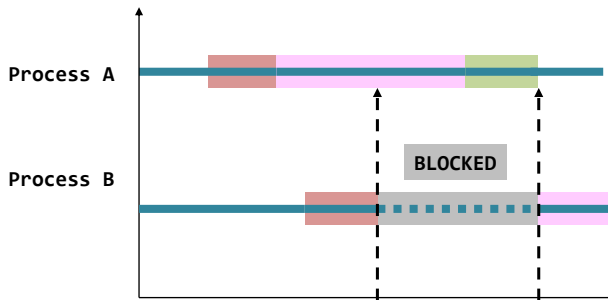
Implication: Only processes that are **not executing in their remainder sections** can participate in deciding which will enter its critical section.

- **Requirement #4: Bounded waiting**. No process would have to wait forever in order to enter its critical section.





Implication: There exists a bound or limit on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section (no processes should be **starved to death**).

A typical mutual exclusion scenario

Remember, it is always the entry blocks other processes, but not the critical section.



Keys

-  Critical section entry
-  Inside Critical section
-  Critical section exit
-  Shared object (if any)

We will be using this coloring scheme throughout this part.

B tries to enter its critical section but A is in its critical section.

A leaves its critical section and B resumes execution accordingly.

Mutual Exclusion Implementation

- Challenges of Implementing **section entry** & **exit**
 - Both operations must be atomic
 - Also need to satisfy the above requirements
 - Performance consideration
- Hardware solution
 - Rely on atomic instructions
 - `test_and_set()`
 - `compare_and_swap`

Example: test_and_set()

- Definition

```
boolean test_and_set(boolean *target) {  
    boolean rv = *target;  
    *target = true;  
  
    return rv;  
}
```

- Mutual exclusion implementation

```
do {  
    while (test_and_set(&lock))  
        ; /* do nothing */  
  
    /* critical section */  
  
    lock = false;  
  
    /* remainder section */  
} while (true);
```


Example: compare_and_swap()

- Definition

```
int compare_and_swap(int *value, int expected, int new_value) {  
    int temp = *value;  
  
    if (*value == expected)  
        *value = new_value;  
  
    return temp;  
}
```

- Mutual exclusion implementation

How to satisfy
bounded waiting?

```
do {  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
  
    /* critical section */  
  
    lock = 0;  
  
    /* remainder section */  
} while (true);
```

Enhanced version

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;

    /* critical section */

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;

    if (j == i)
        lock = false;
    else
        waiting[j] = false;

    /* remainder section */
} while (true);
```

lock is initialized as false



Proposal #1 – disabling interrupt.

- **Method**

- Similar idea as nonpreemptive kernels
- To **disable context switching** when the process is inside the critical section.

- **Effect**

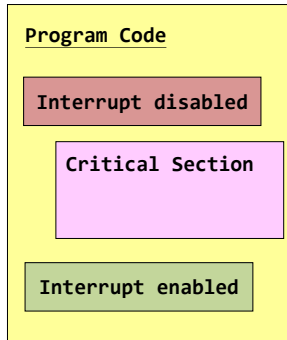
- When a process is in its critical section, no other processes could be able to run.

- **Implementation**

- A new system call should be provided.

- **Correctness?**

- **Correct**, but it is not an *attractive* solution.
- Not as feasible in a multiprocessor environment
- Performance issue (may sacrifice concurrency)



Proposal #2: Mutex Locks

- **Idea**

- A process must acquire the lock before entering a critical section, and release the lock when it exits the critical section
- Using a new shared object to detect the status of other processes, and “**lock**” the shared object

Shared object: “available” (lock)

```
1  acquire(){
2      while(!available)
3          ; /* busy waiting */
4      available = false;
5  }
```

```
1  release(){
2      available = true;
3  }
```

Proposal #2: Mutex Locks

- **Implementation**

- Calls to acquire and release locks must be performed **atomically**
- Often use hardware instructions

- **Issue**

- Busy waiting: Waste CPU resource
 - **Spinlock**

- **Applications**

- Multiprocessor system
 - When locks are expected to be held for short times

Note that: all processes run the following same code.

Program Code

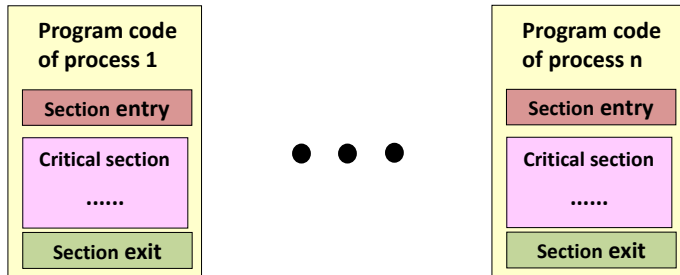
```
acquire();
```

```
Critical Section
```

```
release();
```

Other software-based solutions

- Aim
 - To decide which process could go into its critical section

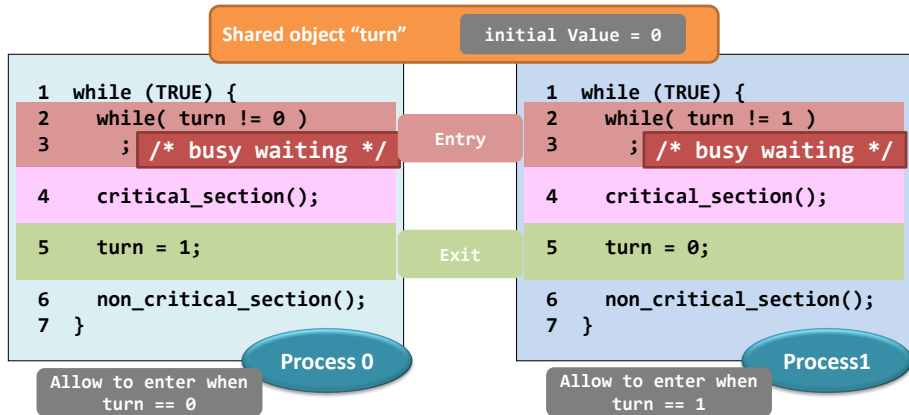


- Key Issues
 - Detect the status of processes (section entry)
 - Need other shared variables
 - Atomicity of section entry and exit

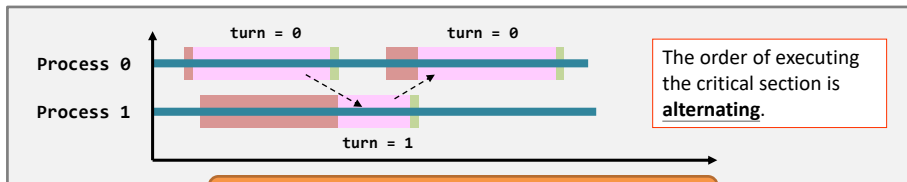
Proposal #3: Strict alternation

• Method

- Using a new shared object to detect the status of other processes



Proposal #3: Strict alternation



Shared object "turn"

initial Value = 0

```
1 while (TRUE) {  
2   while( turn != 0 )  
3     ; /* busy waiting */  
4   critical_section();  
5   turn = 1;  
6   non_critical_section();  
7 }
```

Process 0

```
1 while (TRUE) {  
2   while( turn != 1 )  
3     ; /* busy waiting */  
4   critical_section();  
5   turn = 0;  
6   non_critical_section();  
7 }
```

Process1

Proposal #3: Strict alternation - Cons

- Strict alternation seems good, yet, it is **inefficient**.
 - Busy waiting wastes CPU resources.
- In addition, the alternating order is **too strict**.
 - What if Process 0 wants to enter the critical section **twice in a row**? **NO WAY!**
 - Violate any requirement?

Requirement #3. No process running outside its critical section should block other processes.

Proposal #4: Peterson's solution

- How to improve the strict alternation proposal?
 - The Peterson's solution
- Highlights:
 - Share two data items
 - **int turn;** //whose turn to enter its critical section
 - **Boolean interested[2];** //if a process wants to enter
 - Processes would act as a gentleman: if you want to enter, I'll let you first
 - No alternation is there

Proposal #4: Peterson's solution

Shared object: "turn" &
"interested[2]"

```
1  int turn;                                /* who can enter critical section */
2  int interested[2] = {FALSE,FALSE};      /* wants to enter critical section*/
3
4  void enter_region( int process ) {       /* process is 0 or 1 */
5      int other;                          /* number of the other process */
6      other = 1-process;                  /* other is 1 or 0 */
7      interested[process] = TRUE;         /* want to enter critical section */
8      turn = other;
9      while ( turn == other &&
              interested[other] == TRUE )
10         ; /* busy waiting */
11 }
12
13 void leave_region( int process ) {       /* process: who is leaving */
14     interested[process] = FALSE;        /* I just left critical region */
15 }
```

Entry

Exit

Proposal #4: Peterson's solution

```
1  int turn;
2  int interested[2] = {FALSE,FALSE};
3
4  void enter_region( int process ) {
5      int other;
6      other = 1-process;
7      interested[process] = TRUE;
8      turn = other;
9      while ( turn == other &&
              interested[other] == TRUE
10          ;    /* busy waiting */
11      }
12
13  void leave_region( int process ) {
14      interested[process] = FALSE;
15  }
```

Line 8 therefore makes the other one the turn to run.

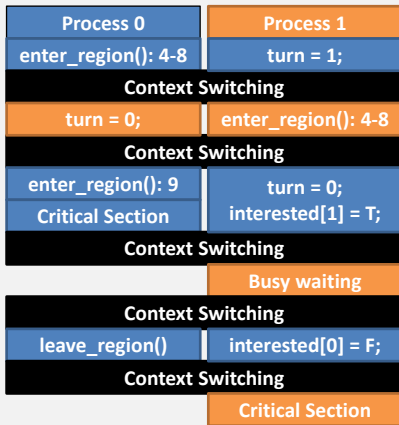
Of course, the process is willing to wait when she wants to enter the critical section.

"I'm a gentleman!"

The process always let another process to enter the critical region first although she wants to enter too.

Proposal #4: Peterson's solution

```
1  int turn;
2  int interested[2] = {FALSE,FALSE};
3
4  void enter_region( int process ) {
5      int other;
6      other = 1-process;
7      interested[process] = TRUE;
8      turn = other;
9      while ( turn == other &&
10             interested[other] == TRUE )
11          ; /* busy waiting */
12
13 void leave_region( int process ) {
14     interested[process] = FALSE;
15 }
```

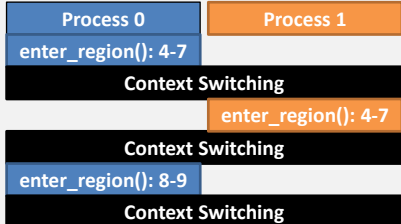


and the story goes on...

Can you show that the requirements are satisfied?

Proposal #4: Peterson's solution

```
1  int turn;
2  int interested[2] = {FALSE,FALSE};
3
4  void enter_region( int process ) {
5      int other;
6      other = 1-process;
7      interested[process] = TRUE;
8      turn = other;
9      while ( turn == other &&
10             interested[other] == TRUE )
11          ; /* busy waiting */
12
13 void leave_region( int process ) {
14     interested[process] = FALSE;
15 }
```

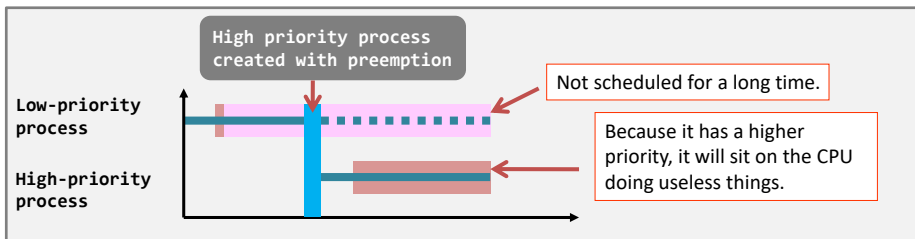


Can you complete the flow?
(what is the difference?)

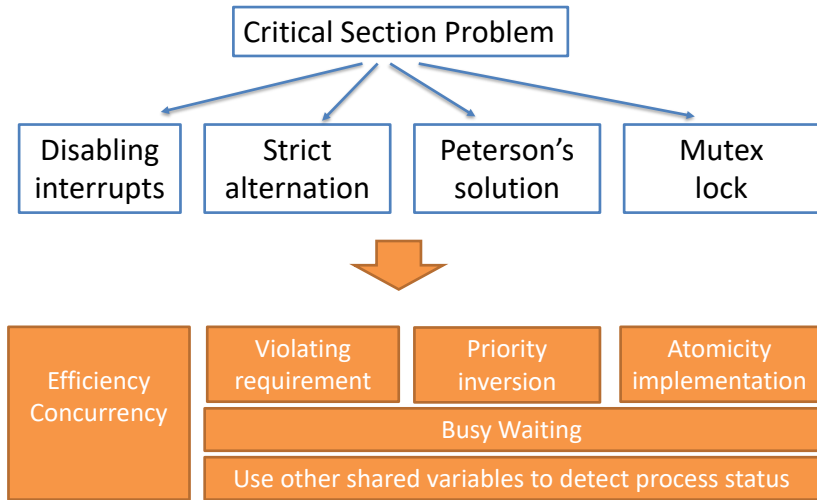
Can both processes progress?

Proposal #4: Peterson's solution – issues

- Busy waiting has its own problem...
 - An apparent problem: wasting CPU time.
 - A hidden, serious problem: **priority inversion problem**.
 - A low priority process is inside the critical region, but ...
 - A high priority process wants to enter the critical region.
 - Then, the high priority process will perform busy waiting for a long time or even forever.

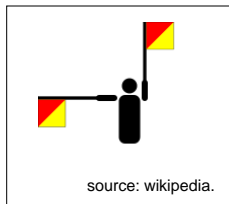


Story so far...



Final proposal: Semaphore

- In real life, semaphore is a flag signaling system.
 - It tells a train driver (or a plane pilot) when to stop and when to proceed.
- When it comes to programming...
 - A semaphore is a data type.
 - You can imagine that it is **an integer** (but it is certainly not an integer when it comes to real implementation).

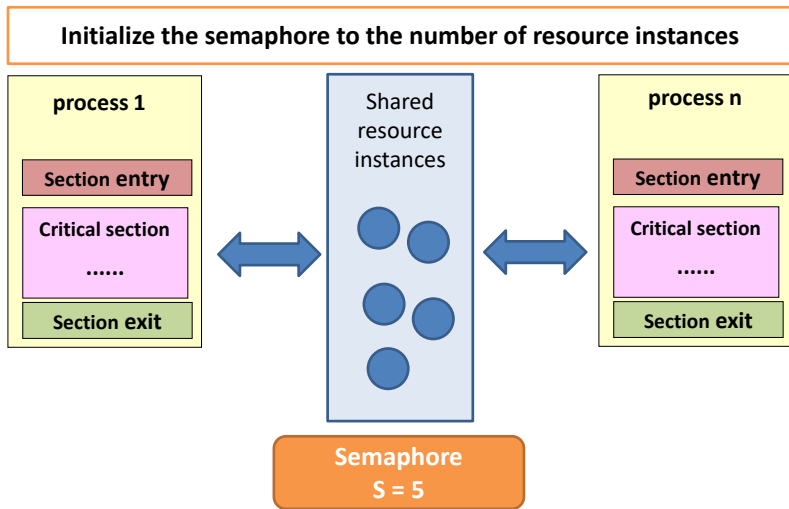


Final proposal: Semaphore

- Semaphore is a data type (**additional shared object**)
 - Accessed only through two standard **atomic** operations
 - **down()**: originally termed P (from Dutch *proberen*, “to test”), **wait()** in textbook
 - Decrementing the count
 - **up()**: originally termed V (from *verhogen*, “to increment”), **signal()** in textbook
 - Incrementing the count
- Two types
 - **Binary semaphore**: 0 or 1 (similar to mutex lock)
 - **Counting semaphore**: control finite number of resources

Final proposal: Semaphore

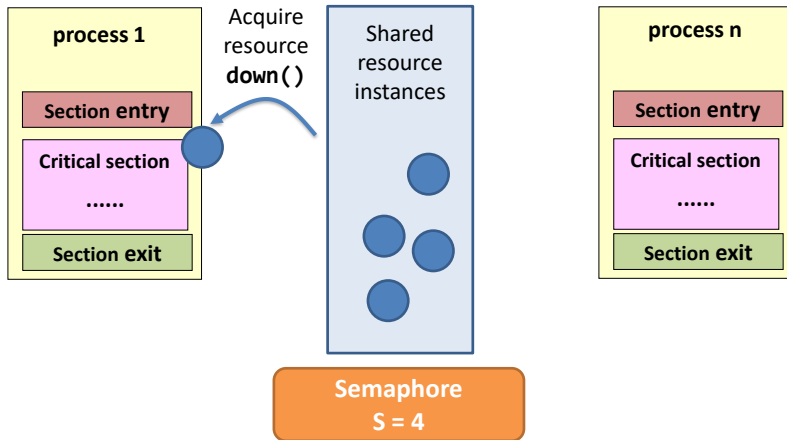
- Idea



Final proposal: Semaphore

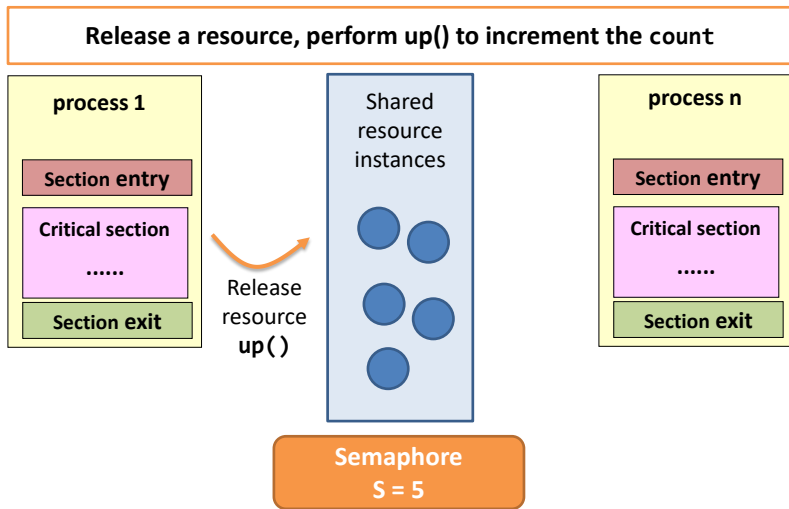
- Idea

Wish to use a resource, perform `down()` to decrement the count



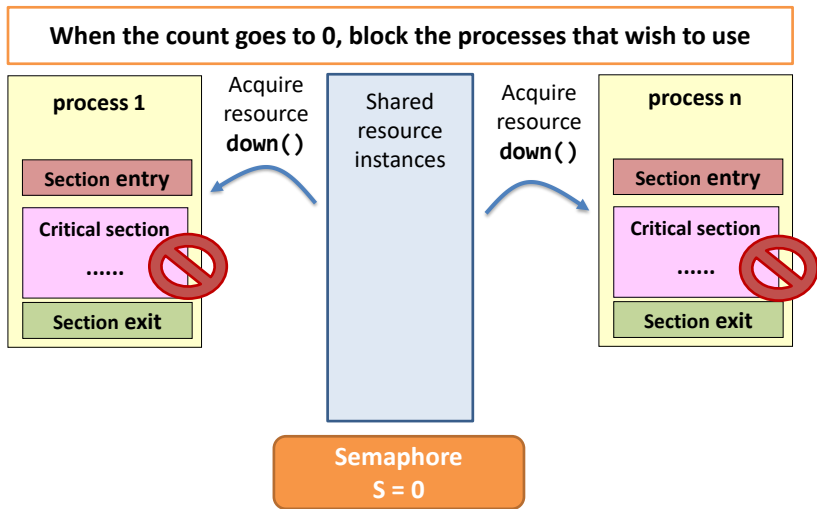
Final proposal: Semaphore

- Idea



Final proposal: Semaphore

- Idea



Semaphore – Simple Implementation

Data Type definition

```
typedef int semaphore;
```

Counting Semaphore: initialized to be the number of resources available

Section Entry: down()

```
1 void down(semaphore *s) {  
2  
3     while ( *s == 0 ) {  
4  
5         ;//busy wait  
6  
7     }  
8     *s = *s - 1;  
9  
10 }
```

Section Exit: up()

```
1 void up(semaphore *s) {  
2  
3  
4  
5     *s = *s + 1;  
6  
7 }
```

Semaphore – Address busy waiting

Data Type definition

```
typedef int semaphore;
```

First issue: Busy waiting

Solution: block the process instead of busy waiting (place the process into a waiting queue)

Section Entry: down()

```
1 void down(semaphore *s) {  
2  
3     while ( *s == 0 ) {  
4  
5         special_sleep();  
6  
7     }  
8     *s = *s - 1;  
9  
10 }
```

Section Exit: up()

```
1 void up(semaphore *s) {  
2  
3     if ( *s == 0 )  
4         special_wakeup();  
5     *s = *s + 1;  
6  
7 }
```


Semaphore – Address busy waiting

Data Type definition

```
typedef int semaphore;
```

First issue: Busy waiting

Solution: block the process instead of busy waiting (place the process into a waiting queue)

```
typedef struct{  
    int value;  
    struct process * list;  
}semaphore;
```

Note

Implementation: The waiting queue may be associated with the semaphore, so a semaphore is not just an integer

Semaphore – Atomicity

Data Type definition

```
typedef int semaphore;
```

Section Entry: down()

```
1 void down(semaphore *s) {  
2  
3     while ( *s == 0 ) {  
4  
5         special_sleep();  
6  
7     }  
8     *s = *s - 1;  
9  
10 }
```

Second issue: Atomicity (both operations must be atomic)

Solution: Disabling interrupts

Section Exit: up()

```
1 void up(semaphore *s) {  
2  
3     if ( *s == 0 )  
4         special_wakeup();  
5     *s = *s + 1;  
6  
7 }
```

Semaphore – Atomicity

Data Type definition

```
typedef int semaphore;
```

Section Entry: down()

```
1 void down(semaphore *s) {
2     disable_interrupt();
3     while ( *s == 0 ) {
4         enable_interrupt();
5         special_sleep();
6         disable_interrupt();
7     }
8     *s = *s - 1;
9     enable_interrupt();
10 }
```

Second issue: Atomicity (both operations must be atomic)

Solution: Disabling interrupts

Also, only one process can invoke “**disable_interrupt()**”. Later processes would be blocked until “**enable_interrupt()**” is called.

Section Exit: up()

```
1 void up(semaphore *s) {
2     disable_interrupt();
3     if ( *s == 0 )
4         special_wakeup();
5     *s = *s + 1;
6     enable_interrupt();
7 }
```

Semaphore – The code

Data Type definition

```
typedef int semaphore;
```

Why need these two statements?

Disabling interrupts may sacrifice concurrency, so it is essential to keep the critical section as short as possible

Section Entry: down()

```
1 void down(semaphore *s) {  
2     disable_interrupt();  
3     while ( *s == 0 ) {  
4         enable_interrupt();  
5         special_sleep();  
6         disable_interrupt();  
7     }  
8     *s = *s - 1;  
9     enable_interrupt();  
10 }
```

Section Exit: up()

```
1 void up(semaphore *s) {  
2     disable_interrupt();  
3     if ( *s == 0 )  
4         special_wakeup();  
5     *s = *s + 1;  
6     enable_interrupt();  
7 }
```

Semaphore – details

Process 1234

down(X)

Section Entry: down()

```
1 void down(semaphore *s) {  
2     disable_interrupt();  
3     while ( *s == 0 ) {  
4         enable_interrupt();  
5         special_sleep();  
6         disable_interrupt();  
7     }  
8     *s = *s - 1;  
9     enable_interrupt();  
10 }
```

Suppose that process 1234 is willing to access the shared resource (enter its critical section), but no resource is available

Semaphore X
Value = 0

1234

Waiting List

Semaphore – details

Process 1357

up(X)

Section Exit: up()

```
1 void up(semaphore *s) {  
2     disable_interrupt();  
3     if ( *s == 0 )  
4         special_wakeup();  
5     *s = *s + 1;  
6     enable_interrupt();  
7 }
```

Process 1234

wakeup

Process 2468

wakeup

Semaphore X
Value = 1

1234

2468

Waiting List

Semaphore – details

Process 1234

Process 2468

down(X)

down(X)

Note that it is impossible for **two blocked processes to get out of the down() simultaneously.**

Why?

Only one process can invoke **disable_interrupt()**

Only one process can manipulate this shared variable

Section Entry: down()

```
1 void down(semaphore *s) {  
2     disable_interrupt();  
3     while ( *s == 0 ) {  
4         enable_interrupt();  
5         special_sleep();  
6         disable_interrupt();  
7     }  
8     *s = *s - 1;  
9     enable_interrupt();  
10 }
```

here

Semaphore – details

Process 1234

down(X)

Process 2468

down(X)

Note that it is impossible for **two processes to get out of the down() simultaneously.**

Why?

Whether which process can get out of **down()** is **the business of the scheduler.**

Section Entry: down()

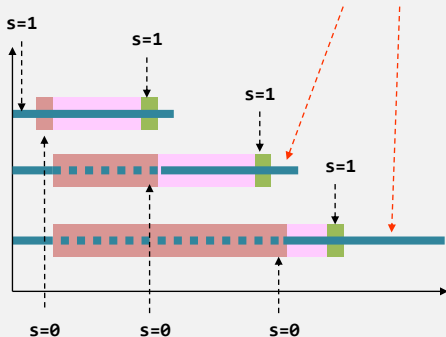
```
1 void down(semaphore *s) {  
2     disable_interrupt();  
3     while ( *s == 0 ) {  
4         enable_interrupt();  
5         special_sleep();  
6         disable_interrupt();  
7     }  
8     *s = *s - 1;  
9     enable_interrupt();  
10 }
```

here

Semaphore – in action

- Add them together...

Either one of the processes can enter the critical section when the first process calls “up(s)”.



```
semaphore *s;  
*s = 1;      /* initial value */
```

```
1 while(TRUE) {  
2     down(s);  
3     critical_section();  
4     up(s);  
5 }
```

entry

exit

Summary...on semaphore

- More on semaphore...it demonstrates an important kind of operations – **atomic operations**.

Definition of atomic operation

- Either none of the instructions of an atomic operation were completed, or
- All instructions of an atomic operation are completed.

- In other words, the entire **up()** and **down()** are indivisible.
 - If it returns, the change must have been made;
 - If it is aborted, no change would be made.

Summary...on critical section problem

- What happened is just the implementation of mutual exclusion (section entry and section exit).

	Comments
Disabling interrupts	Time consuming for multiprocessor systems, sacrifices concurrency.
Strict alternation	Not a good one, busy waiting & violating one mutual exclusion requirement.
Peterson's solution	Busy waiting & has a potential " <i>priority inversion problem</i> ".
Mutex lock	Busy waiting, often relies on hardware instructions.
Semaphore	BEST CHOICE.

Story so far...

- Cooperating processes
 - IPC mechanisms (shared memory, pipes, FIFOs, sockets)
 - Race condition
- Synchronization
 - Mutual exclusion
 - Critical section problem
 - Disabling interrupts, strict alternation, Peterson's solution, mutex lock, semaphore
- What is next?
 - How to use semaphore to solve classic IPC problems
 - Deadlock

Operating Systems

Associate Prof. Yongkun Li

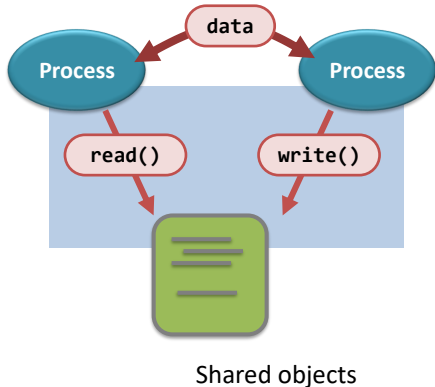
中科大-计算机学院 副教授

<http://staff.ustc.edu.cn/~ykli>

Ch5 Process Communication & Synchronization -Part 3

Story so far...

- For shared memory and files, concurrent access may yield unpredictable outcomes
 - Race condition
- To avoid race condition, **mutual exclusion** must be guaranteed
 - Critical section
 - Implementations (entry/exit)
 - Hardware instructions
 - Disabling interrupts
 - Strict alternation
 - Peterson's solution
 - Mutex lock
 - **Semaphore**



Semaphore Usage

- Semaphore can be used for
 - Mutual exclusion (binary semaphore)
 - Process synchronization (counting semaphore may be needed)
- How to do **process synchronization** w/ semaphore?
 - Mutual exclusion + coordination (multiple semaphores)
 - Careless design may lead to other issues
 - Deadlock

Deadlock

- ❑ Concept
- ❑ Necessary conditions
- ❑ Characterization
- ❑ Solutions

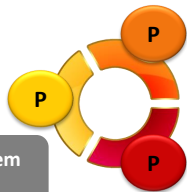
Classic problems

- ❑ Producer-consumer problem
- ❑ Dining philosopher problem
- ❑ Reader-writer problem

The Deadlock Problem

Classic IPC problems

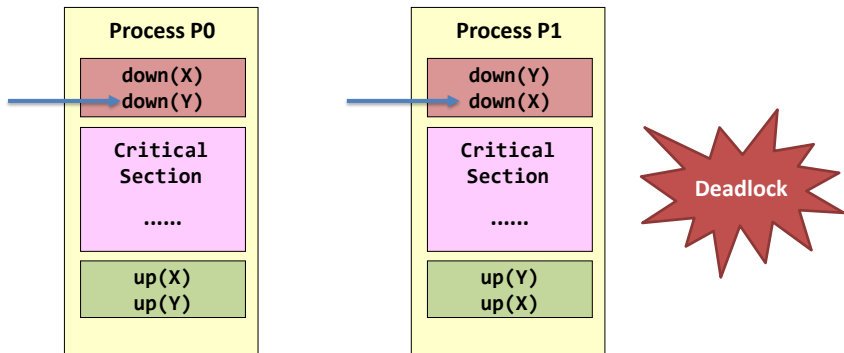
- Producer-consumer problem
- Dining philosopher problem
- Reader-writer problem



Let's teach them
not to fight.

Deadlock Example

- Problems when using semaphore



Scenario: P0 must wait until P1 executes **up(Y)**, P1 must wait until P0 executes **up(X)**

Deadlock Requirements

- **Requirement #1: Mutual Exclusion.**
 - Only one process at a time can use a resource
- **Requirement #2. Hold and wait.**
 - A process must be holding at least one resource and waiting to acquire additional resources held by other processes

Deadlock Requirements

- **Requirement #3: No preemption.**

- A resource can be released only voluntarily by the process holding it after that process has completed its task



- **Requirement #4. Circular wait.**

- There exists a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that P_0 waits for P_1 , P_1 waits for P_2 , ..., P_{n-1} waits for P_n , P_n waits for P_0

How to Handle Deadlocks

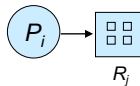
- Deadlock characterization: Deadlocks can be described using **resource-allocation graph**

– Set V is partitioned into two types:

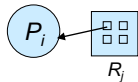
- $P = \{P_1, P_2, \dots, P_n\}$: processes 
- $R = \{R_1, R_2, \dots, R_m\}$: all resource types (each type may have multiple instances) 

– Set E

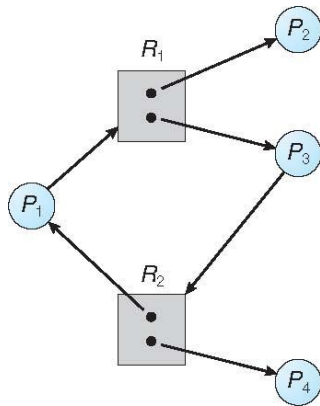
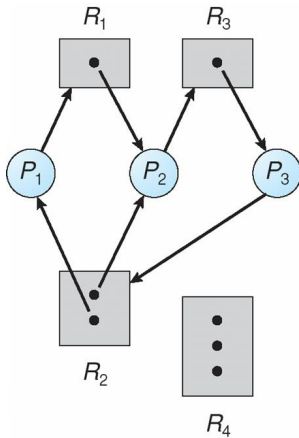
- request edge** – directed edge $P_i \rightarrow R_j$



- assignment edge** – directed edge $R_j \rightarrow P_i$



Examples

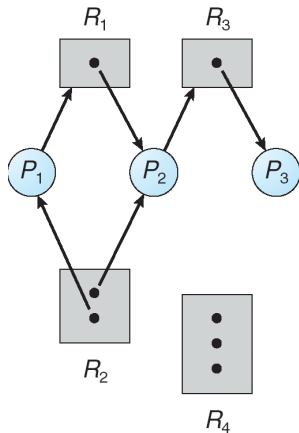


How to Handle Deadlocks

- **Detect** deadlock and recover

- Case 1: Each resource has one instance

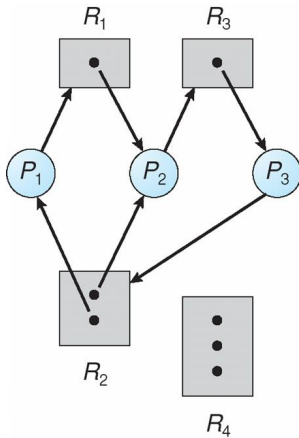
- Resource-allocation graph: detect the existence of a cycle



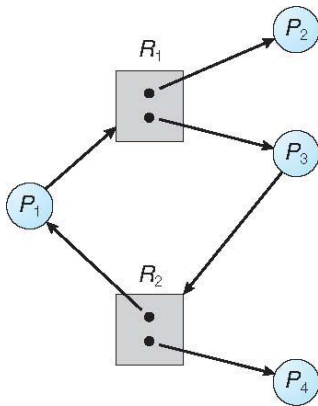
No cycles
No deadlock

Contains a cycle
Case 1: only one instance per resource
Case 2: several instances per resource type: possible deadlock

Examples



Deadlock



No deadlock

How to Handle Deadlocks

- **Detect** deadlock and recover

- Case 2: Each resource has multiple instances

- Matrix method: four data structures

- Existing (total) resources (m types): (E_1, E_2, \dots, E_m)

- Available resources: (A_1, A_2, \dots, A_m)

- Allocation matrix: $\begin{bmatrix} C_{11} & \cdots & C_{1m} \\ \vdots & \ddots & \vdots \\ C_{n1} & \cdots & C_{nm} \end{bmatrix}$ (C_{ij} : # of type- j resources held by process i)

- Request matrix: $\begin{bmatrix} R_{11} & \cdots & R_{1m} \\ \vdots & \ddots & \vdots \\ R_{n1} & \cdots & R_{nm} \end{bmatrix}$ (R_{ij} : # of type- j resources requested by process i)

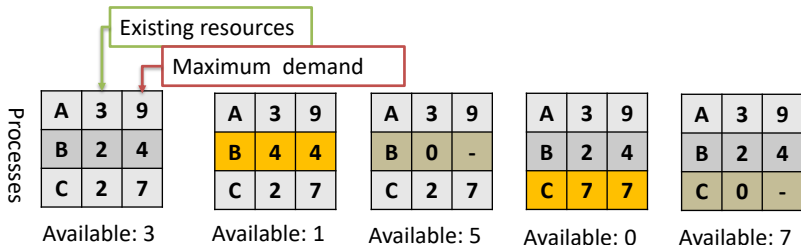
➤ Repeatedly check P_i s.t. $R_i \leq A$? (P_i can be satisfied)

✓ Yes: $A = A + C_i$ (release resources)

✓ No: End (remaining processes are deadlocked)

How to Handle Deadlocks

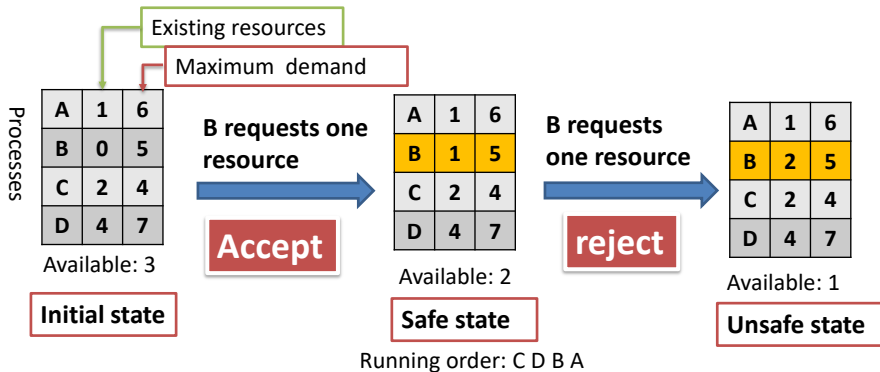
- **Prevent/avoid** deadlocks: Banker's algorithm
 - Idea: check system state defined by (E, A, C, R)
 - **Safe state**: exist one running sequence to guarantee that all processes' demand can be satisfied



- **Unsafe state**: Not exist any sequence to guarantee the demand
 - It is not deadlock (it can still run for some time/processes may release some resources)

How to Handle Deadlocks

- **Prevent/avoid** deadlocks: Banker's algorithm
 - For each request: safe (accept), unsafe (reject)



The algorithm can also be extended to the case of multiple resources, but it needs to know the demand

How to Handle Deadlocks

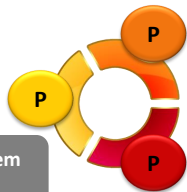
- **Ignore** the problem and pretend that deadlocks never occur (stop functioning and restart manually)
 - 鸵鸟算法（假装没发生）
 - Used by most operating systems, including UNIX and windows
 - Deadlocks occur infrequently, avoiding/detecting it is expensive
- A deadlock-free solution does not eliminate **starvation**

The Deadlock Problem

Classic IPC problems

- Producer-consumer problem
- Dining philosopher problem
- Reader-writer problem

Let's teach them
not to fight.



What are the problems?

- All the IPC classical problems use **semaphores** to fulfill the synchronization requirements.

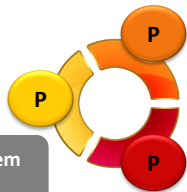
	Properties	Examples
Producer-Consumer Problem	Two classes of processes: <u>producer</u> and <u>consumer</u> ; At least one producer and one consumer.	FIFO buffer, such as pipe.
Dining Philosophy Problem	They are all running the same program; At least two processes.	Cross-road traffic control.
Reader-Writer Problem	Two classes of processes: <u>reader</u> and <u>writer</u> . No limit on the number of the processes of each class.	Database.

The Deadlock Problem

Classic IPC problems

- **Producer-consumer problem**
- Dining philosopher problem
- Reader-writer problem

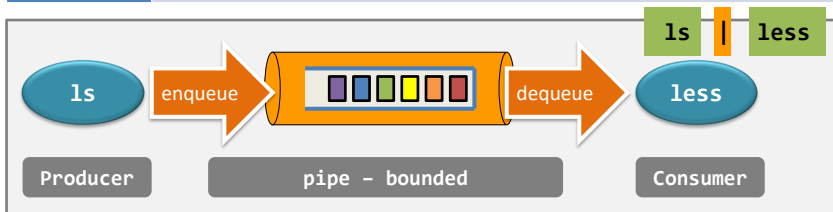
Let's teach them
not to fight.



Producer-consumer problem – recall

- Also known as the **bounded-buffer problem**.

A bounded buffer	<ul style="list-style-type: none">-It is a shared object;-Its size is bounded, say N slots.-It is a queue (imagine that it is an array implementation of queue).
A producer process	<ul style="list-style-type: none">-It produces a unit of data, and-writes that a piece of data to the tail of the buffer at one time.
A consumer process	<ul style="list-style-type: none">-It removes a unit of data from the head of the bounded buffer at one time.



Producer-consumer problem – recall

Producer-consumer requirement #1

- When the producer wants to
- (a) put a new item in the buffer, but
 - (b) **the buffer is already full...**

Then,

- (1) **The producer should be suspended**, and
- (2) **The consumer should wake the producer up** after she has dequeued an item.

Producer-consumer requirement #2

- When the consumer wants to
- (a) consumes an item from the buffer, but
 - (b) **the buffer is empty...**

Then,

- (1) **The consumer should be suspended**, and
- (2) **The producer should wake the consumer up** after she has enqueued an item.

Producer-consumer problem

- Pipe is working fine. Is it enough?
 - What if we cannot use pipes?
 - Say, there are 2 producers and 2 consumers without any parent-child relationships?
 - Then, **the kernel can't protect you with a pipe.**
- In the following, we revisit the producer-consumer problem with the use of shared objects and semaphores, instead of pipe.

Design – Semaphores

- **ISSUE #1: Mutual Exclusion.**

Solution: one binary semaphore (mutex)

- **ISSUE #2: Synchronization (coordination).**

- Remember the two requirements:
 - Insert an item when it is not FULL
 - Consume an item when it is not EMPTY
- Can we use a binary semaphore?

Solution: two counting semaphores (full & empty)

Producer-consumer problem – solution

Note

The functions “insert_item()” and “remove_item()” are accessing the bounded buffer (codes in critical section).

The size of the bounded buffer is “N”.

Producer function

```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6
7         insert_item(item);
8
9     }
10 }
11 }
12 }
```

Consumer Function

```
1 void consumer(void) {
2     int item;
3
4     while(TRUE) {
5
6         item = remove_item();
7
8         consume_item(item);
9     }
10 }
11 }
12 }
```

Producer-consumer problem – solution

Note

Mutual exclusion requirement

Synchronization requirement

Shared object

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

Producer function

```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6
7         insert_item(item);
8
9     }
10 }
11 }
12 }
```

Consumer Function

```
1 void consumer(void) {
2     int item;
3
4     while(TRUE) {
5
6         item = remove_item();
7
8         consume_item(item);
9     }
10 }
11 }
12 }
```

Producer-consumer problem – Understanding

Why we need three semaphores, “empty”, “full”, “mutex”?

Shared object

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

Producer function

```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6
7         insert_item(item);
8
9     }
10 }
11 }
12 }
```

Consumer Function

```
1 void consumer(void) {
2     int item;
3
4     while(TRUE) {
5
6         item = remove_item();
7
8         consume_item(item);
9     }
10 }
11 }
12 }
```

Producer-consumer problem – Understanding

Why we need three semaphores, “empty”, “full”, “mutex”?

mutex:

What is its purpose?

Why is the initial value of mutex 1?

Shared object

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

Producer function

```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6
7         down(&mutex);
8         insert_item(item);
9         up(&mutex);
10    }
11 }
12 }
```

Consumer Function

```
1 void consumer(void) {
2     int item;
3
4     while(TRUE) {
5
6         down(&mutex);
7         item = remove_item();
8         up(&mutex);
9
10        consume_item(item);
11    }
12 }
```

Producer-consumer problem – Understanding

Why we need three semaphores, “empty”, “full”, “mutex”?

mutex:

what is its purpose?

Why is the initial value of mutex 1?

Shared object

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

Producer function

```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6
7         down(&mutex);
8         insert_item(item);
9         up(&mutex);
10    }
11 }
12 }
```

The “**mutex**” stands for mutual exclusion.

- **down()** and **up()** statements are the entry and the exit of the critical section, respectively.

What is the meaning of the initial value 1?

Producer-consumer problem – Understanding

Why we need three semaphores, “empty”, “full”, “mutex”?

How about “full” and “empty”?

Shared object

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

Producer function

```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6         down(&empty);
7         down(&mutex);
8         insert_item(item);
9         up(&mutex);
10        up(&full);
11    }
12 }
```

Consumer Function

```
1 void consumer(void) {
2     int item;
3
4     while(TRUE) {
5         down(&full);
6         down(&mutex);
7         item = remove_item();
8         up(&mutex);
9         up(&empty);
10        consume_item(item);
11    }
12 }
```

Producer-consumer problem – Understanding

- The two variables are not for mutual exclusion, but for **process synchronization**.
 - “Process synchronization” means **to coordinate** the set of processes so as to produce meaningful output.

Producer function

```
1 void producer(void) {  
2     int item;  
3  
4     while(TRUE) {  
5         item = produce_item();  
6         down(&empty);  
7         down(&mutex);  
8         insert_item(item);  
9         up(&mutex);  
10        up(&full);  
11    }  
12 }
```

Consumer Function

```
1 void consumer(void) {  
2     int item;  
3  
4     while(TRUE) {  
5         down(&full);  
6         down(&mutex);  
7         item = remove_item();  
8         up(&mutex);  
9         up(&empty);  
10        consume_item(item);  
11    }  
12 }
```

Producer-consumer problem – Understanding

For “empty”,

- Its initial value is N;
- It decrements by 1 in each iteration.
- When it reaches 0, the producers sleeps.

So, does it sound like one of the requirements?

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

The consumer wakes the producer up when it finds “empty” is 0.

Producer function

```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6         down(&empty);
7         down(&mutex);
8         insert_item(item);
9         up(&mutex);
10        up(&full);
11    }
12 }
```

Consumer Function

```
1 void consumer(void) {
2     int item;
3
4     while(TRUE) {
5         down(&full);
6         down(&mutex);
7         item = remove_item();
8         up(&mutex);
9         up(&empty);
10        consume_item(item);
11    }
12 }
```

Producer-consumer problem – Understanding

- Semaphore can be more than mutual exclusion!

empty	It represents the number of empty slots.
full	It represents the number of occupied slots.

Producer function

```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6         down(&empty);
7         down(&mutex);
8         insert_item(item);
9         up(&mutex);
10        up(&full);
11    }
12 }
```

Consumer Function

```
1 void consumer(void) {
2     int item;
3
4     while(TRUE) {
5         down(&full);
6         down(&mutex);
7         item = remove_item();
8         up(&mutex);
9         up(&empty);
10        consume_item(item);
11    }
12 }
```

Producer-consumer problem – question

Question.

Can we swap Lines 6 & 7 of the producer?

Let us simulate what will happen with the modified code!

Shared object

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

Producer function

```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6*      down(&mutex);
7*      down(&empty);
8         insert_item(item);
9         up(&mutex);
10        up(&full);
11    }
12 }
```

Consumer Function

```
1 void consumer(void) {
2     int item;
3
4     while(TRUE) {
5         down(&full);
6         down(&mutex);
7         item = remove_item();
8         up(&mutex);
9         up(&empty);
10        consume_item(item);
11    }
12 }
```

Producer-consumer problem – question

Producer

running until Line
10

Consumer

We are showing the value of the semaphores before the producer is suspended.

mutex = 1

empty = 0

full = N

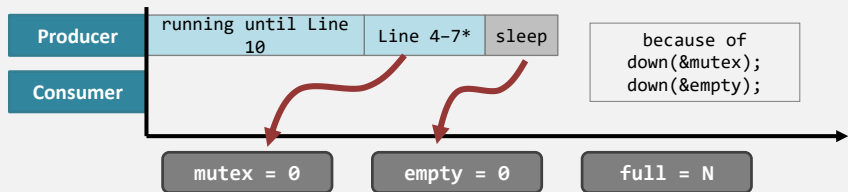
Producer function

```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6*      down(&mutex);
7*      down(&empty);
8         insert_item(item);
9         up(&mutex);
10        up(&full);
11    }
12 }
```

Consumer Function

```
1 void consumer(void) {
2     int item;
3
4     while(TRUE) {
5         down(&full);
6         down(&mutex);
7         item = remove_item();
8         up(&mutex);
9         up(&empty);
10        consume_item(item);
11    }
12 }
```

Producer-consumer problem – question



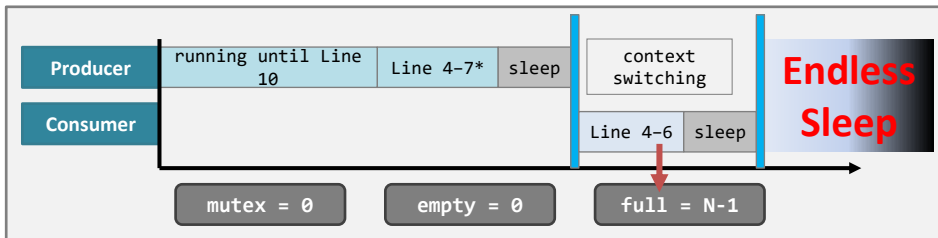
Producer function

```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6*      down(&mutex);
7*      down(&empty);
8         insert_item(item);
9         up(&mutex);
10        up(&full);
11    }
12 }
```

Consumer Function

```
1 void consumer(void) {
2     int item;
3
4     while(TRUE) {
5         down(&full);
6         down(&mutex);
7         item = remove_item();
8         up(&mutex);
9         up(&empty);
10        consume_item(item);
11    }
12 }
```

Producer-consumer problem – question



Producer function

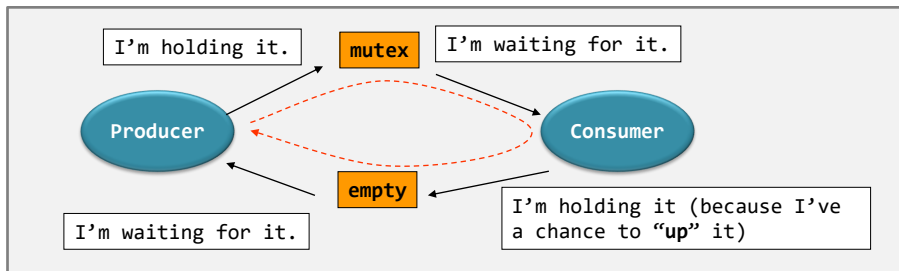
```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6*      down(&mutex);
7*      down(&empty);
8         insert_item(item);
9         up(&mutex);
10        up(&full);
11    }
12 }
```

Consumer Function

```
1 void consumer(void) {
2     int item;
3
4     while(TRUE) {
5         down(&full);
6         down(&mutex);
7         item = remove_item();
8         up(&mutex);
9         up(&empty);
10        consume_item(item);
11    }
12 }
```


Producer-consumer problem

- **Deadlock** happens when a **circular wait** appears
 - The producer is waiting for the consumer to “up()” the “**empty**” semaphore, and
 - the consumer is waiting for the producer to “up()” the “**mutex**” semaphore.

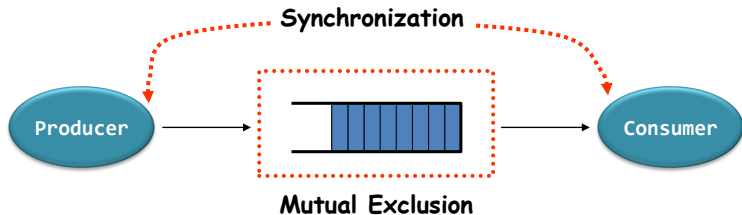


Producer-consumer problem

- **Deadlock** happens when a **circular wait** appears
 - The producer is waiting for the consumer to “**up()**” the “**empty**” semaphore, and
 - the consumer is waiting for the producer to “**up()**” the “**mutex**” semaphore.
- **No progress could be made by all processes + All processes are blocked.**
 - **Implication:** careless implementation of the producer-consumer solution can be disastrous.

Summary on producer-consumer problem

- The problem can be divided into two sub-problems.
 - Mutual exclusion.
 - The buffer is a shared object. Mutual exclusion is needed.
 - Synchronization.
 - Because the buffer's size is bounded, coordination is needed.



Summary on producer-consumer problem

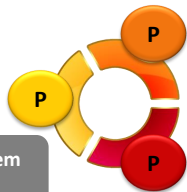
- How to guarantee mutual exclusion?
 - A **binary semaphore** is used as the entry and the exit of the critical sections.
- How to achieve synchronization?
 - Two semaphores are used as **counters** to monitor the status of the buffer.
 - Two semaphores are needed because the two suspension conditions are different.

The Deadlock Problem

Classic IPC problems

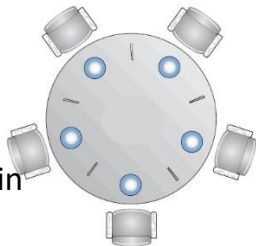
- Producer-consumer problem
- Dining philosopher problem
- Reader-writer problem

Let's teach them
not to fight.

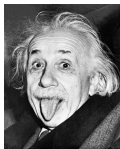


Dining philosopher – introduction

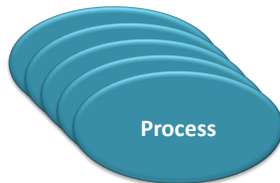
- 5 philosophers, 5 plates of spaghetti, and 5 chopsticks.
- The jobs of each philosopher are
 - to think and
 - to eat: They **need exactly two chopsticks** in order to eat the spaghetti.
- Question: how to construct a synchronization protocol such that
 - they will not result in any **deadlocking scenarios**, and
 - they will not be **starved to death**



Dining philosopher – introduction



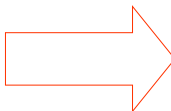
Philosophers



Chopsticks



Spaghetti

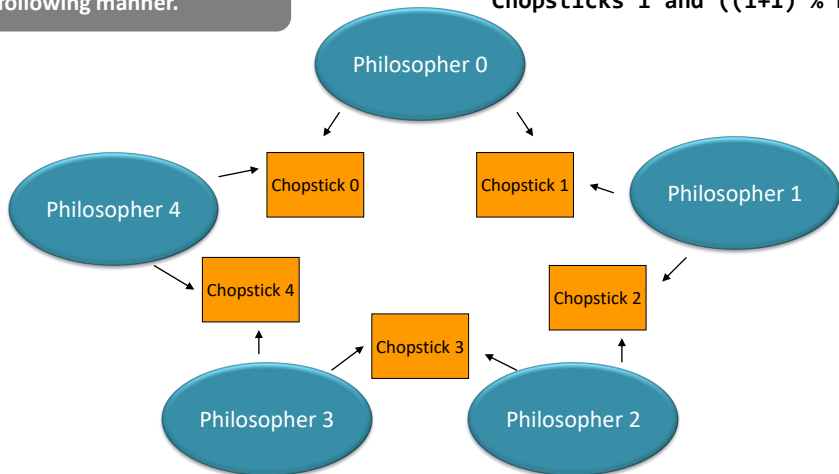


Consider to have
infinite supply.

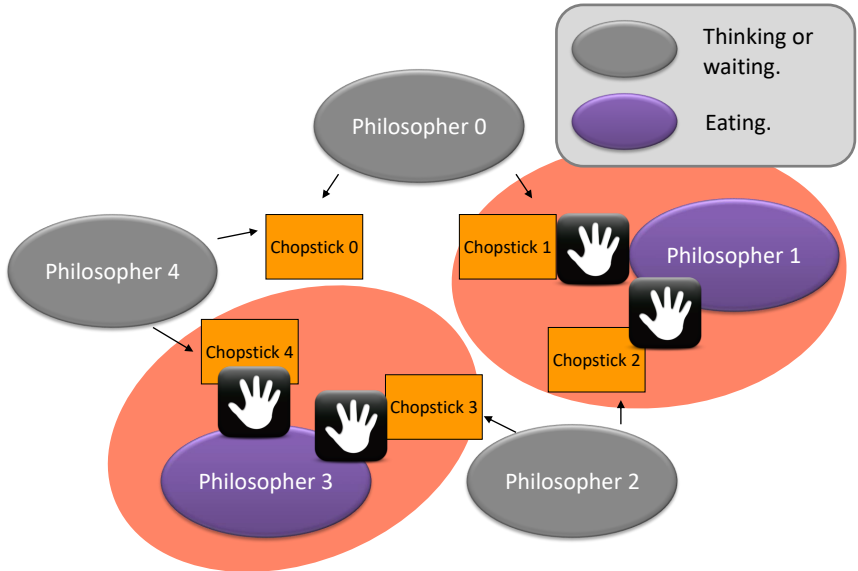
Dining philosopher – introduction

The chopsticks are arranged in the following manner.

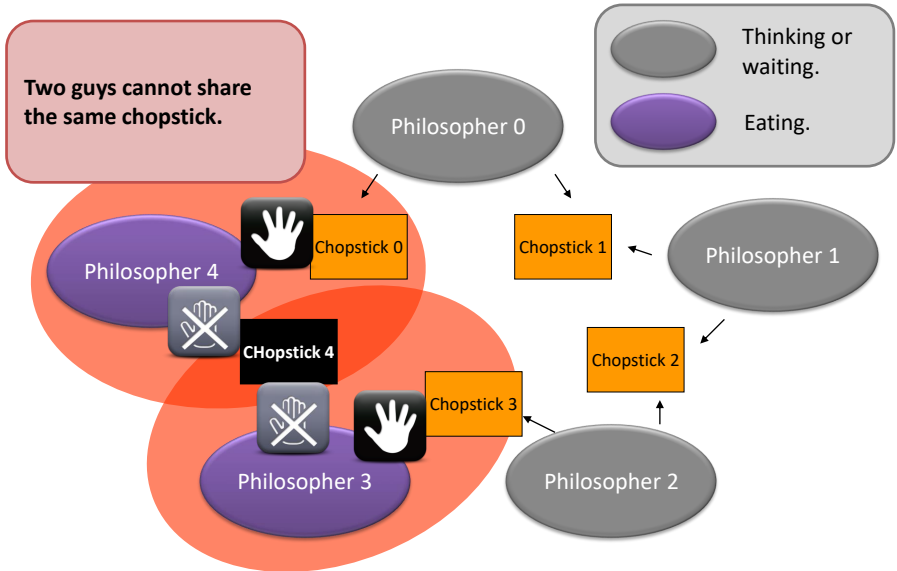
Philosopher i needs
Chopsticks i and $((i+1) \% N)$;



Dining philosopher – introduction



Dining philosopher – introduction



Dining philosopher – requirement #1

- **Mutual exclusion**

- What if there is no mutual exclusion?

- Then: while you're eating, the two men besides you will and must **steal all your chopsticks!**

- Let's propose the following solution:

- When you are hungry, you have to check if anyone is using the chopstick that you need.
 - If yes, you have to wait.
 - If no, **seize both chopsticks.**
 - After eating, put down all your chopsticks.

Dining philosopher – meeting requirement #1?

Shared object

```
#define N 5  
semaphore chop[N];
```

A quick question: what should be initial values?

Helper Functions

```
void take(int i) {  
    down(&chop[i]);  
}
```

```
void put(int i) {  
    up(&chop[i]);  
}
```

Section
Entry

Critical
Section

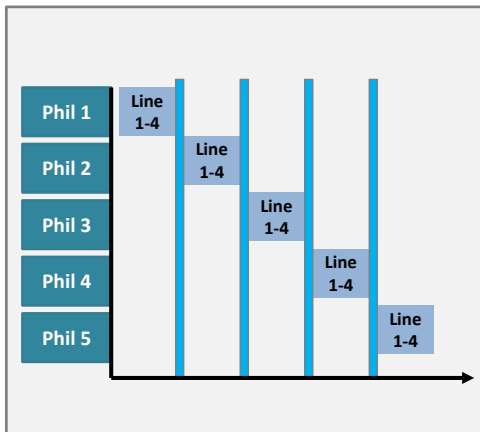
Section
Exit

Main Function

```
1 void philosopher(int i) {  
2     while (TRUE) {  
3         think();  
4         take(i);  
5         take((i+1) % N);  
6         eat();  
7         put(i);  
8         put((i+1) % N);  
9     }  
10 }
```

Dining philosopher – meeting requirement #1?

Final Destination: Deadlock!



Main Function

```
1 void philosopher(int i) {  
2     while (TRUE) {  
3         think();  
4         take(i);  
5         take((i+1) % N);  
6         eat();  
7         put(i);  
8         put((i+1) % N);  
9     }  
10 }
```

Dining philosopher – requirement #2

- **Synchronization**

- Should avoid any **potential deadlocking execution order**.

- How about the following suggestions:

- First, a philosopher **takes a chopstick**.
 - If a philosopher finds that he cannot take the second one, then he should **put down the first chopstick**.
 - Then, the philosopher **goes to sleep** for a while.
 - Again, the philosopher tries to get both chopsticks until both ones are seized.

Dining philosopher – meeting requirement #2?

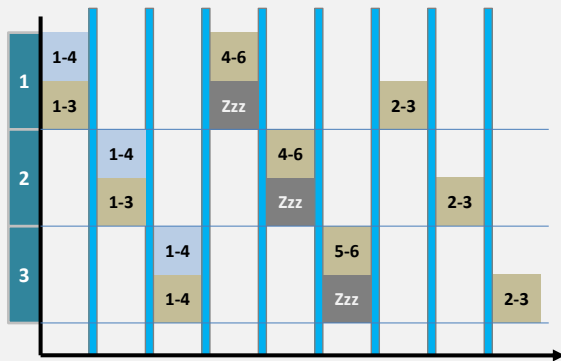
The code: meeting requirement #2?

```
1 void philosopher(int i) {  
2     while (TRUE) {  
3         think();  
4         take(i);  
5         eat();  
6         up(&chop[i]);  
7         up(&chop[(i+1)%N]);  
8     }  
9 }
```

```
1 void take(int i) {  
2     while(TRUE) {  
3         down(&chop[i]);  
4         if (isUsed((i+1)%N)) {  
5             up(&chop[i]);  
6             sleep(1);  
7         }  
8         else {  
9             down(&chop[(i+1)%N]);  
10            break;  
11        }  
12    }  
13 }
```

Dining philosopher – meeting requirement #2?

Potential Problem: Philosophers are all busy but no progress were made!



Assume $N = 3$ (because the space is limited)

```
1 void take(int i) {  
2     while(TRUE) {  
3         down(&chop[i]);  
4         if (isUsed((i+1)%N)) {  
5             up(&chop[i]);  
6             sleep(1);  
7         }  
8         else {  
9             down(&chop[(i+1)%N]);  
10            break;  
11        }  
12    }  
13 }
```

```
1 void philosopher(int i) {  
2     while (TRUE) {  
3         think();  
4         take(i);  
5         eat();  
6         up(&chop[i]);  
7         up(&chop[(i+1)%N]);  
8     }  
9 }
```


Dining philosopher – before the final solution.

- Before we present the final solution, let's see what are the problems that we have.

Problems

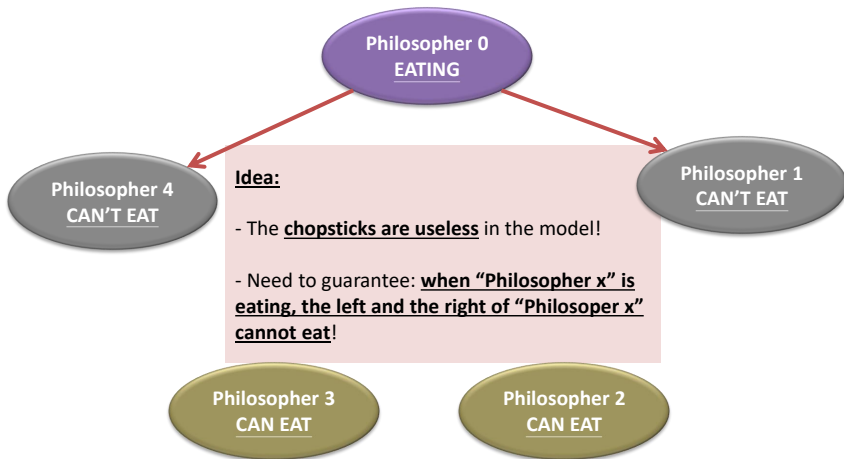
Model a chopstick as a semaphore is intuitive, but is not working.

The problem is that we are afraid to “**down()**”, as that may lead to a deadlock.

Using `sleep()` to avoid deadlock is effective, yet bringing another problem.

We can always create an execution order that keeps all the philosophers busy, but without useful output.

Dining philosopher – before the final solution.



Dining philosopher – the final solution.

Shared object

```
#define N 5
#define LEFT  ((i+N-1) % N)
#define RIGHT  ((i+1) % N)

int state[N];
semaphore mutex = 1;
semaphore s[N];
```

Main function

```
1 void philosopher(int i) {
2     think();
3     take(i);
4     eat();
5     put(i);
6 }
```

Section entry

```
1 void take(int i) {
2     down(&mutex);
3     state[i] = HUNGRY;
4     test(i);
5     up(&mutex);
6     down(&s[i]);
7 }
```

Section exit

```
1 void put(int i) {
2     down(&mutex);
3     state[i] = THINKING;
4     test(LEFT);
5     test(RIGHT);
6     up(&mutex);
7 }
```

I will explain the
code later.

Extremely important helper function

```
1 void test(int i) {
2     if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
3         state[i] = EATING;
4         up(&s[i]);
5     }
6 }
```

Dining philosopher – the final solution.

Shared object

```
#define N 5
#define LEFT  ((i+N-1) % N)
#define RIGHT ((i+1) % N)

int state[N];
semaphore mutex = 1;
semaphore s[N];
```

Going “left” and “right” in a circular manner.

The states of the philosophers, including “EATING”, “THINKING”, and “HUNGRY”.

Remember, this is shared array.

To guarantee mutual exclusive access to the “**state[N]**” array.

Guess:

What is the meaning of the semaphore `s[N]`?

To fulfill the synchronization requirement.

Question. What are the initial values of the “`s[N]`” array?

Dining philosopher – the final solution.

Shared object

```
#define N 5
#define LEFT ((i+N-1) % N)
#define RIGHT ((i+1) % N)

int state[N];
semaphore mutex = 1;
semaphore s[N];
```

Section entry

```
1 void take(int i) {
2     down(&mutex);
3     state[i] = HUNGRY;
4     test(i);
5     up(&mutex);
6     down(&s[i]);
7 }
```

Question. What are they doing?

If both chopsticks are available,
I eat. Else, I sleep.

Extremely important helper function

```
1 void test(int i) {
2     if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
3         state[i] = EATING;
4         up(&s[i]);
5     }
6 }
```

If they are eating, I can't be eating.

Dining philosopher – the final solution.

Try to let the one on the **left of the caller** to eat.

Try to let the one on the **right of the caller** to eat.

Section exit

```
1 void put(int i) {  
2     down(&mutex);  
3     state[i] = THINKING;  
4     test(LEFT);  
5     test(RIGHT);  
6     up(&mutex);  
7 }
```

Extremely important helper function

```
1 void test(int i) {  
2     if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {  
3         state[i] = EATING;  
4         up(&s[i]);  
5     }  
6 }
```

Wake up the one who can eat!

Dining philosopher – the final solution.

An illustration: How can
Philosopher 1 start eating?

Philosopher 0
THINKING

Philosopher 4
THINKING

Note: no chopsticks objects
will be shown in this
illustration because we
don't need them now.

Philosopher 1
THINKING

Philosopher 3
THINKING

Philosopher 2
THINKING

Dining philosopher – the final solution.

Section entry

```
1 void take(int i) {  
2     down(&mutex);  
3     state[i] = HUNGRY;  
4     test(i);  
5     up(&mutex);  
6     down(&s[i]);  
7 }
```

Call take();

Philosopher 0
HUNGRY

To LEFT:
are you "EATING"?

To RIGHT:
are you "EATING"?

Philosopher 4
THINKING

Philosopher 1
THINKING

Philosopher 3
THINKING

Philosopher 2
THINKING

Dining philosopher – the final solution.

Section entry

```
1 void take(int i) {  
2     down(&mutex);  
3     state[i] = HUNGRY;  
4     test(i);  
5     up(&mutex);  
6     down(&s[i]);  
7 }
```

Call `take()`;

Philosopher 0
HUNGRY

To LEFT:
are you "EATING"?

Philosopher 4
THINKING

To RIGHT:
are you "EATING"?

Philosopher 1
THINKING

Calling `take()`.
but, it is blocked.

Why?

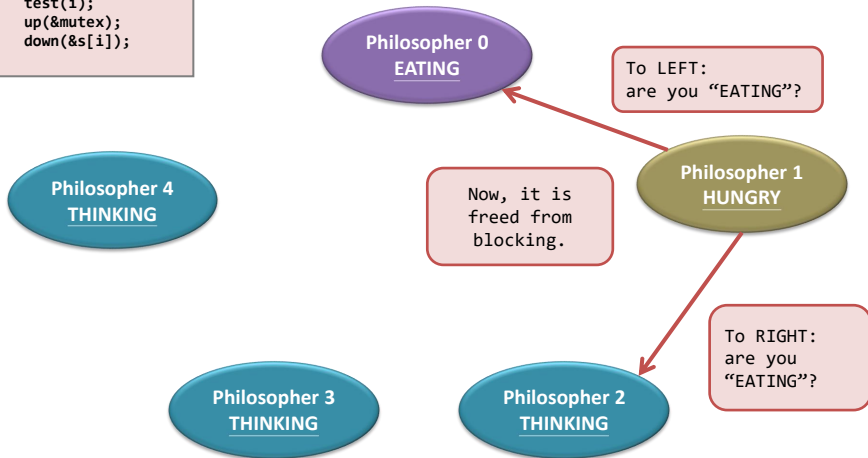
Philosopher 3
THINKING

Philosopher 2
THINKING

Dining philosopher – the final solution.

Section entry

```
1 void take(int i) {  
2     down(&mutex);  
3     state[i] = HUNGRY;  
4     test(i);  
5     up(&mutex);  
6     down(&s[i]);  
7 }
```



Dining philosopher – the final solution.

Section entry

```
1 void take(int i) {  
2     down(&mutex);  
3     state[i] = HUNGRY;  
4     test(i);  
5     up(&mutex);  
6     down(&s[i]);  
7 }
```

Philosopher 0
EATING

Philosopher 4
THINKING

Philosopher 1
HUNGRY

To RIGHT:
are you
"EATING"?

To LEFT:
are you
"EATING"?

Blocked;
because of
`down(&s[1]);`

Philosopher 3
HUNGRY

Philosopher 2
THINKING

Dining philosopher – the final solution.

Section entry

```
1 void take(int i) {  
2     down(&mutex);  
3     state[i] = HUNGRY;  
4     test(i);  
5     up(&mutex);  
6     down(&s[i]);  
7 }
```

Philosopher 0
EATING

Philosopher 4
THINKING

Philosopher 1
HUNGRY

Blocked;
because of
`down(&s[1]);`

Philosopher 3
EATING

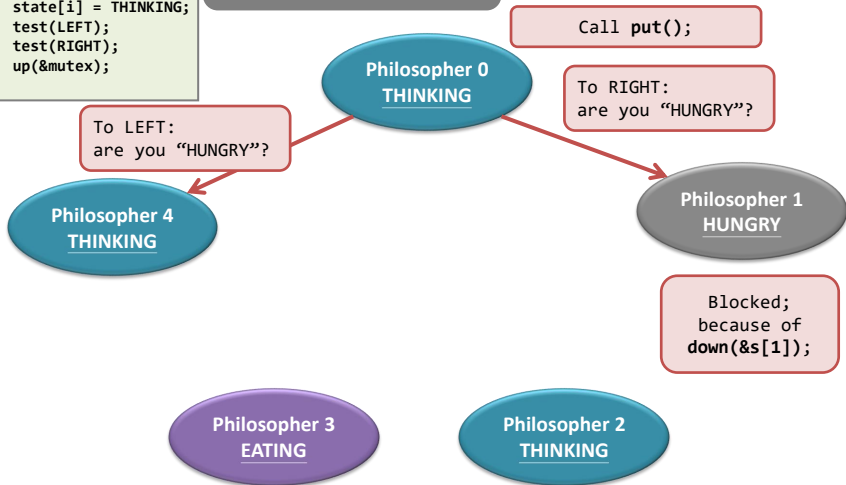
Philosopher 2
THINKING

Dining philosopher – the final solution.

Section exit

```
1 void put(int i) {  
2   down(&mutex);  
3   state[i] = THINKING;  
4   test(LEFT);  
5   test(RIGHT);  
6   up(&mutex);  
7 }
```

An illustration: How can
Philosopher 1 start eating?

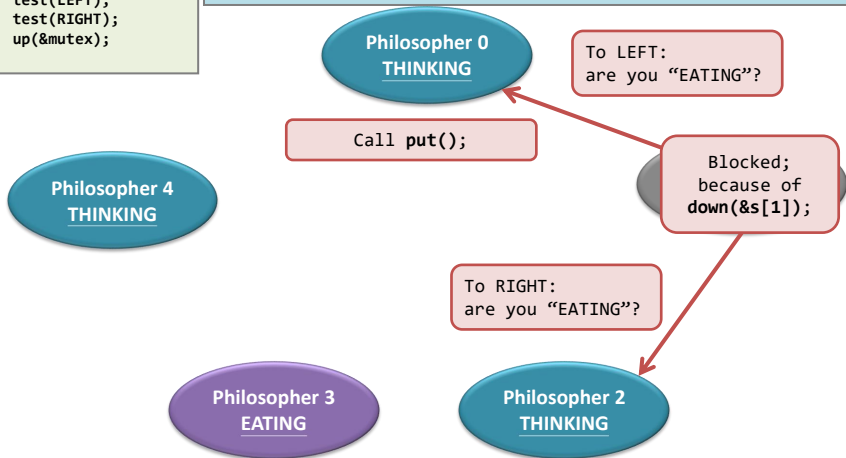


Dining philosopher – the final solution.

Section exit

```
1 void put(int i) {  
2   down(&mutex);  
3   state[i] = THINKING;  
4   test(LEFT);  
5   test(RIGHT);  
6   up(&mutex);  
7 }
```

```
1 void test(int i) {  
2   if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {  
3     state[i] = EATING;  
4     up(&s[i]);  
5   }  
6 }
```



Dining philosopher – the final solution.

Section exit

```
1 void put(int i) {  
2   down(&mutex);  
3   state[i] = THINKING;  
4   test(LEFT);  
5   test(RIGHT);  
6   up(&mutex);  
7 }
```

```
1 void test(int i) {  
2   if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {  
3     state[i] = EATING;  
4     up(&s[i]);  
5   }  
6 }
```

Philosopher 0
THINKING

Call put();

Blocked;
because of
`down(&s[1]);`

Remove your
blocked state by
calling `up(&s[1]);`

Philosopher 4
THINKING

Philosopher 3
EATING

Philosopher 2
THINKING

Dining philosopher – the final solution.

Section entry

```
1 void take(int i) {  
2     down(&mutex);  
3     state[i] = HUNGRY;  
4     test(i);  
5     up(&mutex);  
6     down(&s[i]);  
7 }
```

Philosopher 0
THINKING

Philosopher 4
THINKING

Philosopher 1
EATING

Eventually...

Philosopher 3
EATING

Philosopher 2
THINKING

Dining philosopher - summary

- What is the shared object in the final solution?
 - How to guarantee the mutual exclusion

Section entry	Section exit
<pre>1 void take(int i) { 2 down(&mutex); 3 state[i] = HUNGRY; 4 test(i); 5 up(&mutex); 6 down(&s[i]); 7 }</pre>	<pre>1 void put(int i) { 2 down(&mutex); 3 state[i] = THINKING; 4 test(LEFT); 5 test(RIGHT); 6 up(&mutex); 7 }</pre>

Dining philosopher - summary

- Think:
 - Why the semaphore $s[N]$ is needed
 - How to set its initial value

Section entry

```
1 void take(int i) {  
2     down(&mutex);  
3     state[i] = HUNGRY;  
4     test(i);  
5     up(&mutex);  
6     down(&s[i]);  
7 }
```

Extremely important helper function

```
1 void test(int i) {  
2     if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {  
3         state[i] = EATING;  
4         up(&s[i]);  
5     }  
6 }
```

Dining philosopher - summary

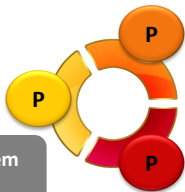
- Solution to IPC problem can be difficult to comprehend.
 - Usually, intuitive methods failed.
 - Depending on time, e.g., `sleep(1)`, does not guarantee a useful solution.
- As a matter of fact, dining philosopher **is not restricted to 5 philosophers**.

The Deadlock Problem

Classic IPC problems

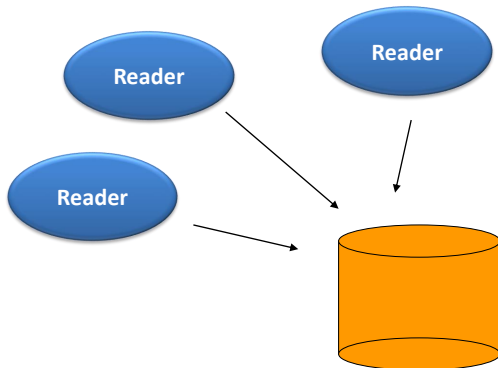
- Producer-consumer problem
- Dining philosopher problem
- Reader-writer problem

Let's teach them
not to fight.



Reader-writer problem – introduction

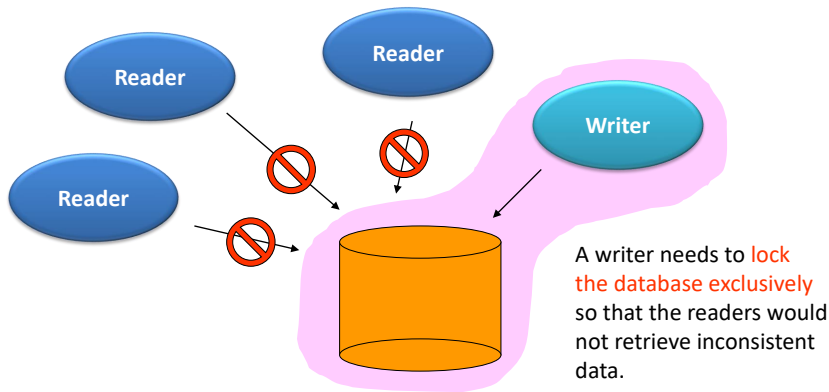
- It is a concurrent database problem.



Readers are allowed to read the content of the database concurrently.

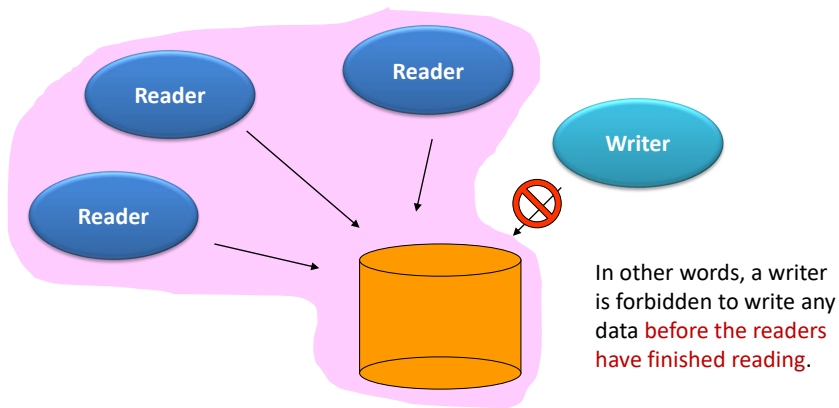
Reader-writer problem – introduction

- It is a concurrent database problem.



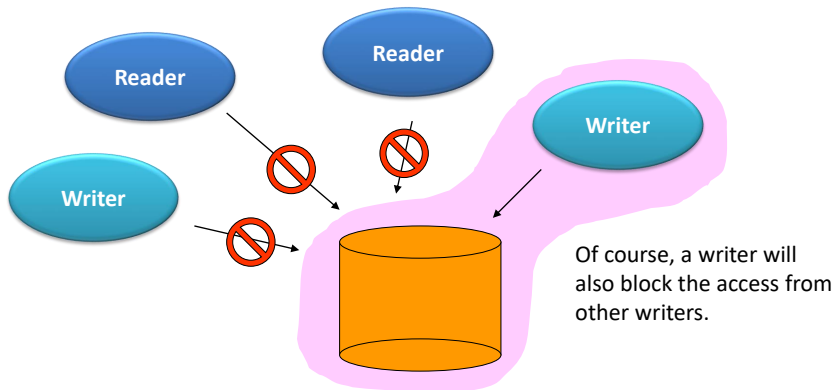
Reader-writer problem – introduction

- It is a concurrent database problem.



Reader-writer problem – introduction

- It is a concurrent database problem.

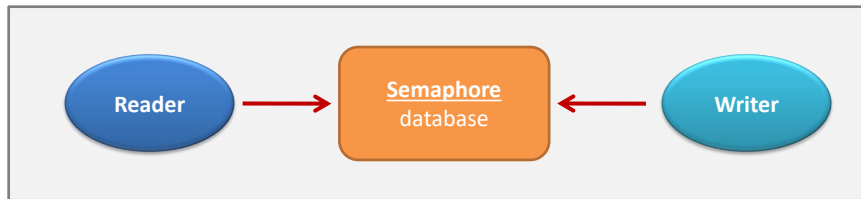


Reader-writer problem – subproblems

- A mutual exclusion problem.
 - The database is a shared object.
- A synchronization problem.
 - **Rule 1.** While a reader is reading, other readers is allowed to read the database.
 - **Rule 2.** While a reader is reading, no writers is allowed to write to the database.
 - **Rule 3.** While a writer is writing, no writers and readers are allowed to access the database.
- A concurrency problem.
 - **Simultaneous access for multiple readers** is allowed and must be guaranteed.

Reader-writer problem – solution outline

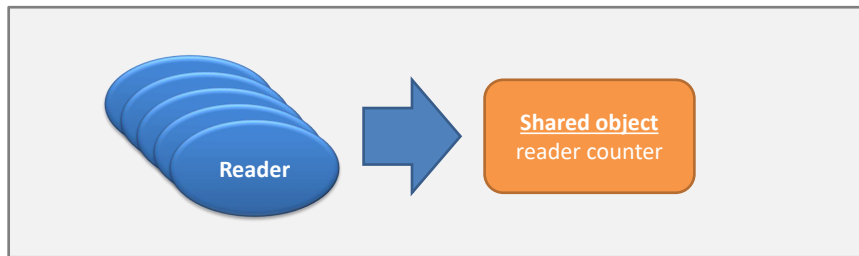
- **Mutual exclusion**: relate the readers and the writers to one semaphore.
 - This guarantees **no readers and writers** could proceed to their critical sections at the same time.
 - This also guarantees **no two writers** could proceed to their critical sections at the same time.



Reader-writer problem – solution outline

- Readers' concurrency

- The **first reader coming** to the system “**down()**” the “**database**” semaphore.
- The **last reader leaving** the system “**up()**” the “**database**” semaphore.



Reader-writer problem – final solution

Shared object

```
semaphore db    = 1;
semaphore mutex = 1;
int read_count  = 0;
```

Writer function

```
1 void writer(void) {
2   while(TRUE) {
```

Section Entry

```
   prepare_write();
   down(&db);
```

Critical Section

```
   write_database();
```

Section Exit

```
   up(&db);
```

```
7   }
8 }
```

Reader Function

```
1 void reader(void) {
2   while(TRUE) {
```

Section Entry

```
   down(&mutex);
   read_count++;
5   if(read_count == 1)
6     down(&db);
7   up(&mutex);
```

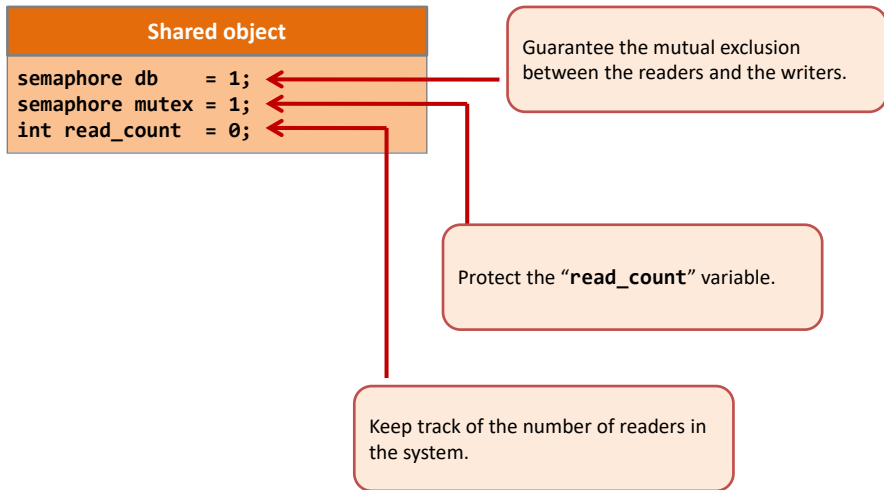
Critical Section

```
   read_database();
```

Section Exit

```
   down(&mutex);
   read_count--;
11  if(read_count == 0)
12    up(&db);
13  up(&mutex);
14  process_data();
15  }
16 }
```

Reader-writer problem – final solution



Reader-writer problem – final solution

Shared object

```
semaphore db    = 1;  
semaphore mutex = 1;  
int read_count  = 0;
```

Writer function

```
1 void writer(void) {  
2   while(TRUE) {  
   Section Entry  prepare_write();  
                  down(&db);  
   Critical Section write_database();  
   Section Exit   up(&db);  
7   }  
8   }
```

The writer is allowed to enter its critical section when no other process is in its critical section (protected by the “**db**” semaphore)

Reader-writer problem – final solution

Shared object

```
semaphore db    = 1;  
semaphore mutex = 1;  
int read_count  = 0;
```

The first reader “**down()**” the “**db**” semaphore so that no writers would be allowed to enter their critical sections.

The last reader “**up()**” the “**db**” semaphore so as to let the writers to enter their critical section.

Reader Function

```
1 void reader(void) {  
2     while(TRUE) {  
3         down(&mutex);  
4         read_count++;  
5         if(read_count == 1)  
6             down(&db);  
7         up(&mutex);  
8         read_database();  
9         down(&mutex);  
10        read_count--;  
11        if(read_count == 0)  
12            up(&db);  
13        up(&mutex);  
14        process_data();  
15    }  
16 }
```

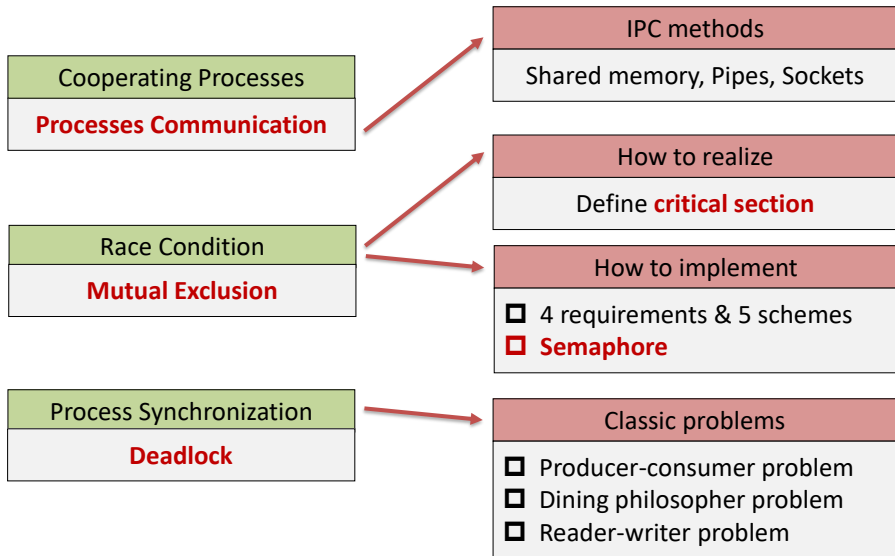
Reader-writer problem – summary

- This solution does not limit the number of readers and the writers admitted to the system.
 - A realistic database needs this property.
- This solution gives readers a higher priority over the writers.
 - Whenever there are readers, writers must be blocked, not the other way round.
- **What if a writer should be given a higher priority?**

Summary on IPC problems

- The problems have the following properties in common:
 - Multiple processes;
 - Shared and limited resources;
 - Processes have to be synchronized in order to generate useful output;
- The synchronization algorithms have the following requirements in common:
 - Guarantee mutual exclusion;
 - Uphold the correct synchronization among processes;
 - Deadlock-free.

Summary on Ch5



Operating Systems

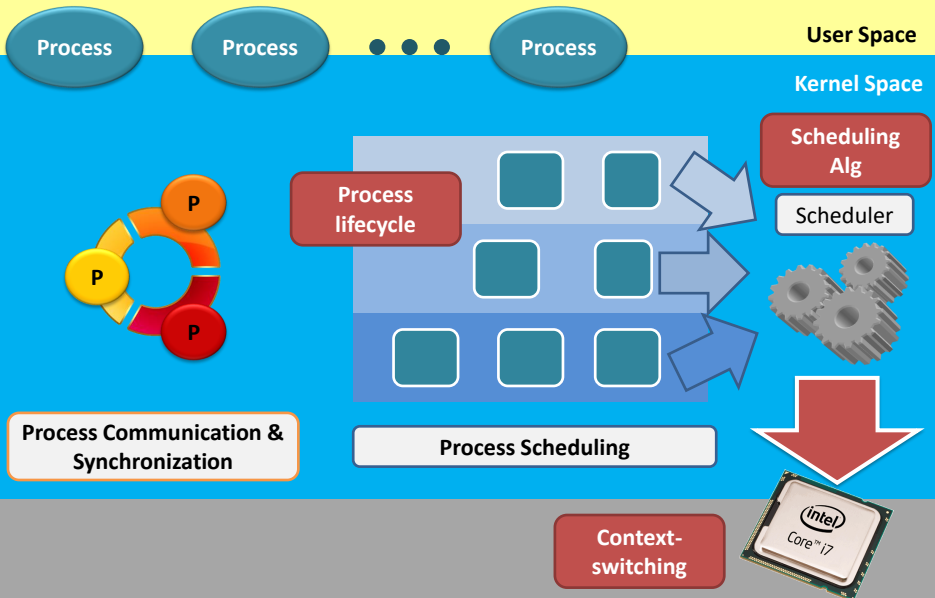
Associate Prof. Yongkun Li

中科大-计算机学院 副教授

<http://staff.ustc.edu.cn/~ykli>

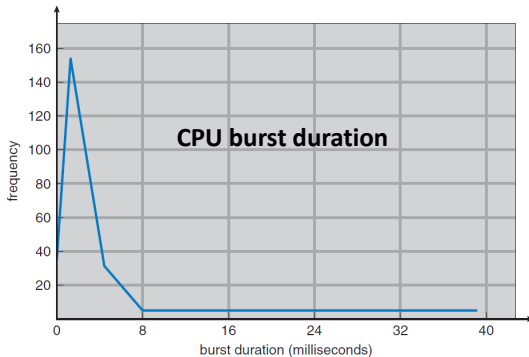
Ch6 Process Scheduling

Outline



Why scheduling is needed

- Process execution
 - Consists of a cycle of CPU execution and I/O wait
 - CPU burst + I/O burst



Why scheduling is needed

Question. How to improve CPU utilization (CPU is much faster than I/O)?

Question. How to improve system responsiveness (interactive applications)?



Multiprogramming

Multitasking

A system may contain many processes which are at different states (ready for running, waiting for I/O)

Scheduling is required because the number of computing resource – the CPU – is **limited**.

Topics

- Process lifecycle
- Process scheduling
 - Context switching
 - Scheduling criteria
 - Scheduling algorithms
 - Applications/Scenarios



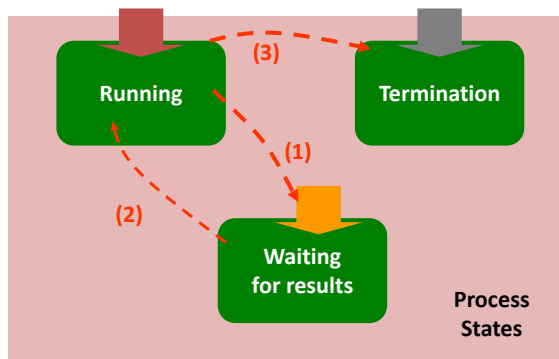
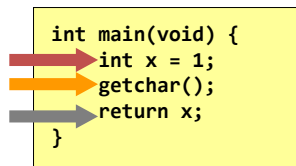
Topics

- **Process lifecycle**
- Process scheduling
 - Context switching
 - Scheduling criteria
 - Scheduling algorithms
 - Applications/Scenarios

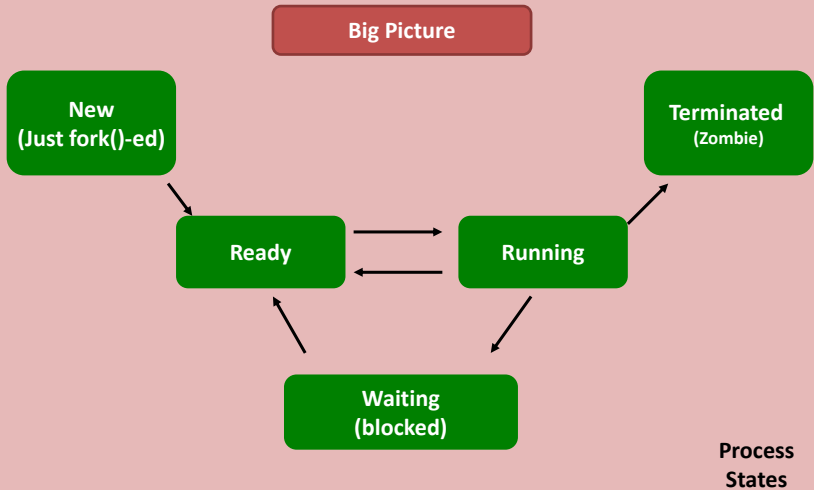


Programmer's point of view...

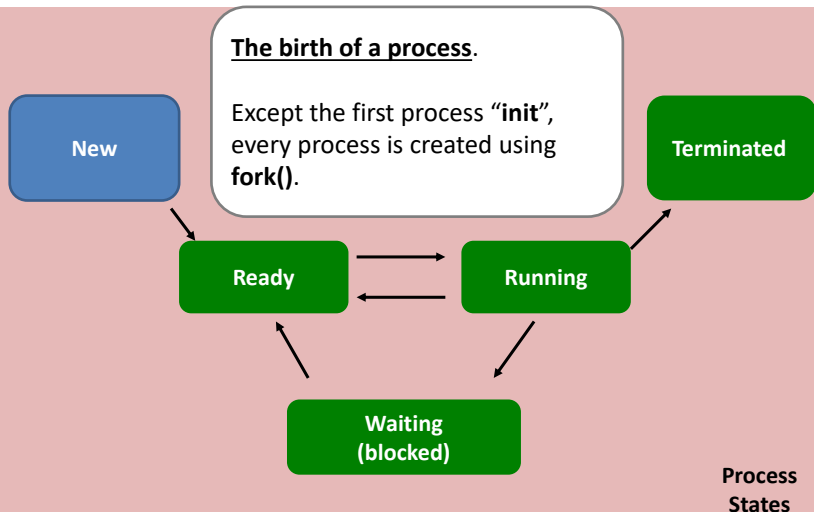
- This is how a fresh programmer looks at a process' life cycle.



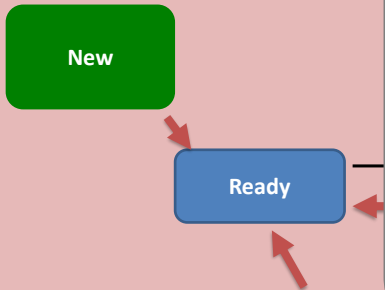
Kernel's point of view...



Kernel's point of view...



Kernel's point of view...



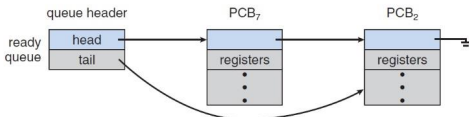
The process is ready.

It means it is **ready to run but is not running**.

A process may become “**ready**” after...

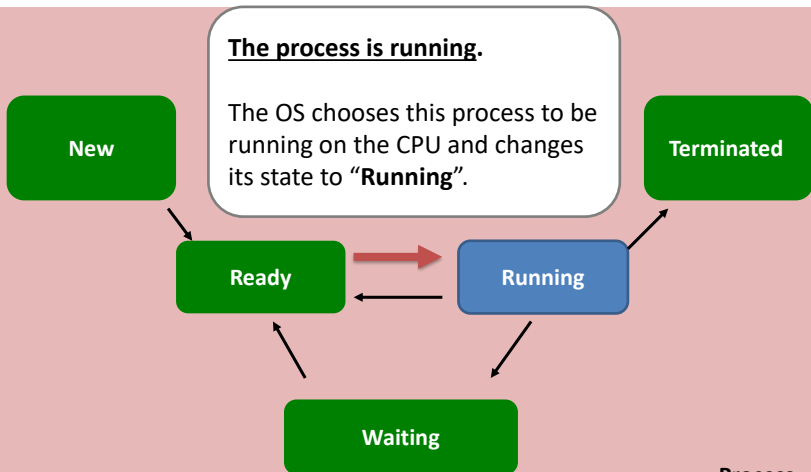
- it is just created by **fork()**;
- it has been running on the CPU for some time and the OS chooses another process to run;
- returning from blocked states.

All ready processes are kept on a list called **ready queue**



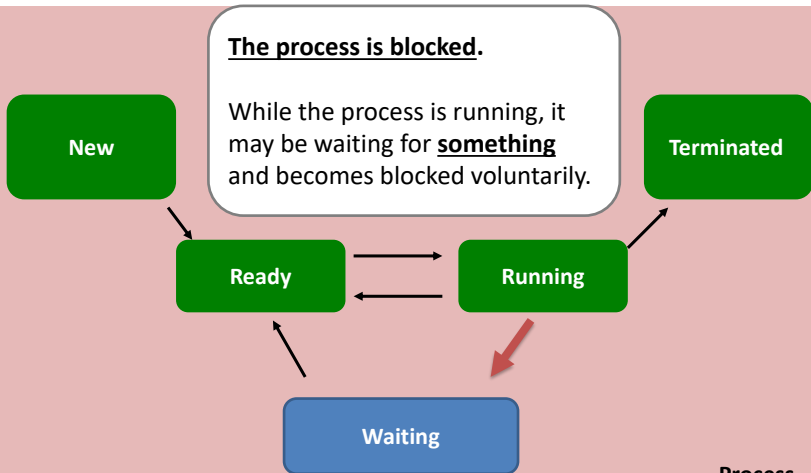
**Process
States**

Kernel's point of view...



Process
States

Kernel's point of view...



Process
States

Kernel's point of view...

Example. Reading a file.

Sometimes, the process has to wait for the response from the device and, therefore, it is **blocked**.

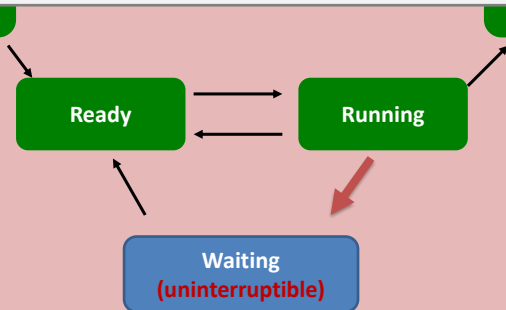
Nevertheless, this blocking state is **interruptible**. E.g., “**Ctrl + C**” can get the process out of the waiting state (but goes to termination state instead).



Process
States

Kernel's point of view...

Sometimes, a process needs to wait for a resource but it doesn't want to be disturbed while it is waiting. In other words, **the process wants that resource very much**. Then, the process status is set to the **uninterruptible** status.

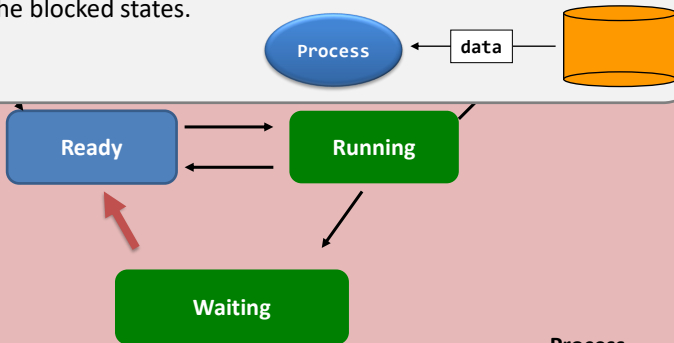


Process
States

Kernel's point of view...

Return back to ready.

When response arrives, the status of the process changes back to **Ready**.
from any one of the blocked states.



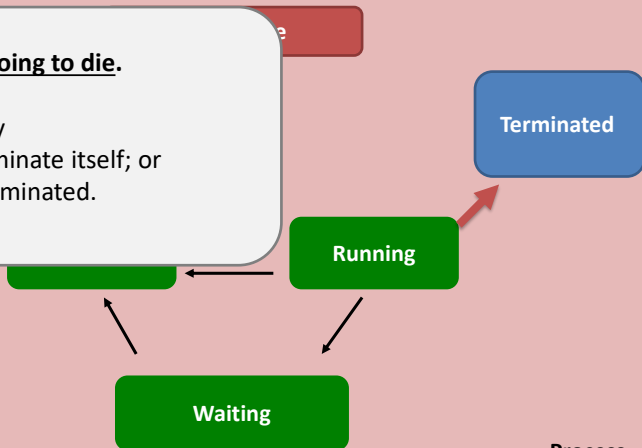
Process
States

Kernel's point of view...

The process is going to die.

The process may

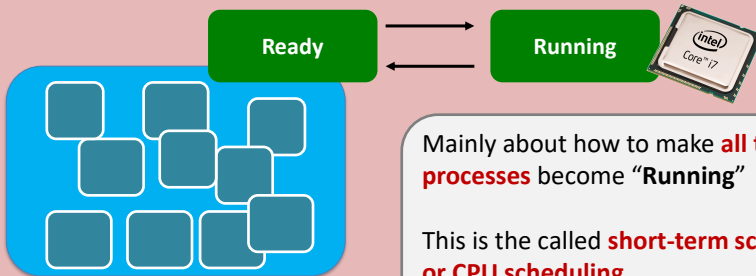
- choose to terminate itself; or
- force to be terminated.



Process
States

What is scheduling?

So, what is process scheduling?



Triggering Events

- When process scheduling happens:

A new process is created.

When “**fork()**” is invoked and returns successfully.

Then, whether the parent or the child is scheduled is up to the scheduler's decision.

An existing process is terminated.

The CPU is freed. The scheduler should choose another process to run.

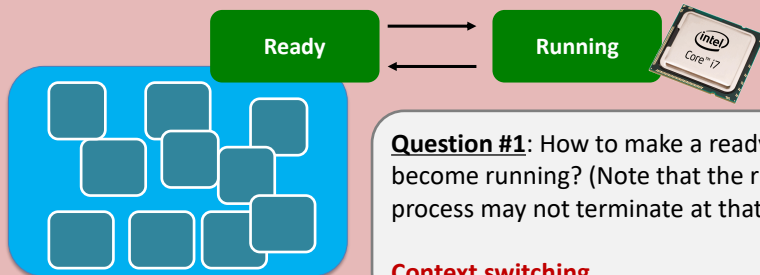
A process waits for I/O.

The CPU is freed. The scheduler should choose another process to run.

A process finishes waiting for I/O.

The interrupt handling routine **makes a scheduling request**, if necessary.

Key Issues



Question #2: How to decide which process should be running?

Scheduling criteria & scheduling algorithms

Question #3: How to design scheduling in a real/specific system?

Multiprocessor system, real-time system, algorithm evaluation

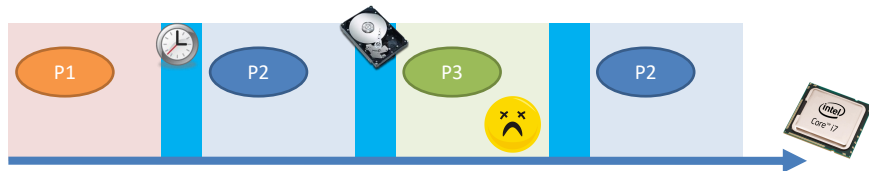
Topics

- Process lifecycle
- **Process scheduling**
 - **Context switching**
 - Scheduling criteria
 - Scheduling algorithms
 - Applications/Scenarios



What is context switching?

- Before we can jump into the process scheduling topic, we have to understand what “**context switching**” is.



Scheduling is the procedure that decides which process to run next.

Context switching is the actual switching procedure, from one process to another.



Timer interrupt.



Hardware interrupt.

Switching from one process to another.

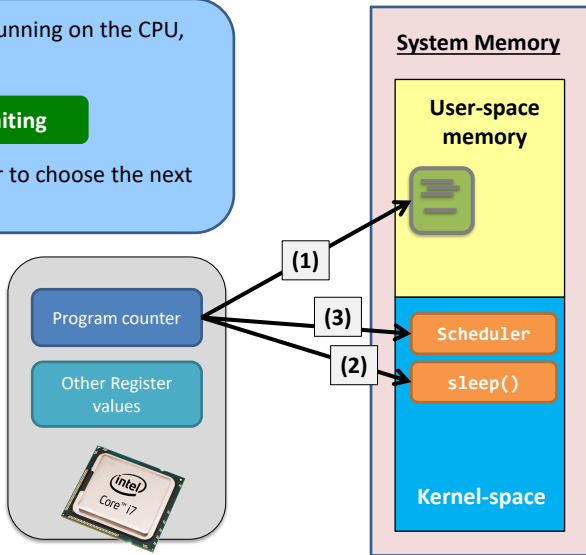
Suppose this process gives up running on the CPU, e.g., calling `sleep()`. Then:

Running



Waiting

Now, it is time for the scheduler to choose the next process to run.



Switching from one process to another.

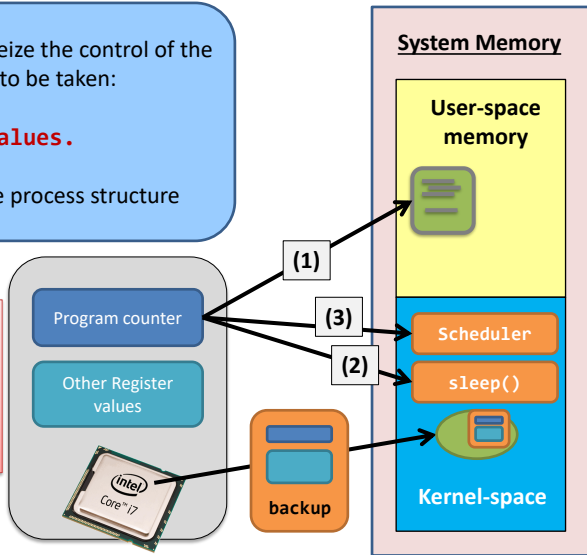
But, before the scheduler can seize the control of the CPU, a very important step has to be taken:

Backup all registers' values.

The backup will be stored in the process structure

The context of a process

The union of the user-space memory and the registers' values of the process



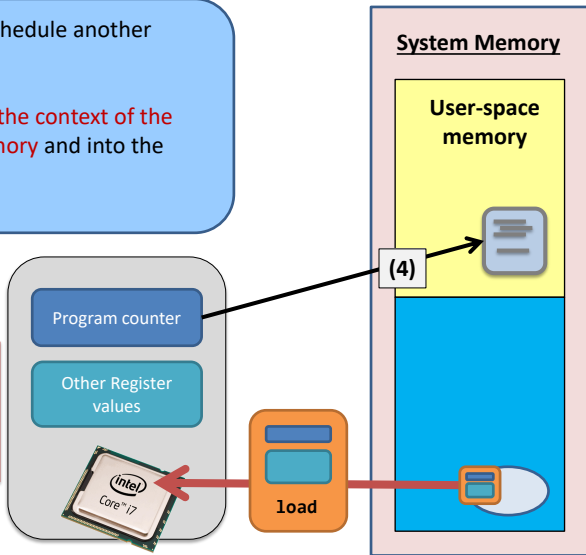
Switching from one process to another.

Say, the scheduler decides to schedule another process.

Then, the scheduler has to **load the context of the new process into the main memory** and into the CPU.

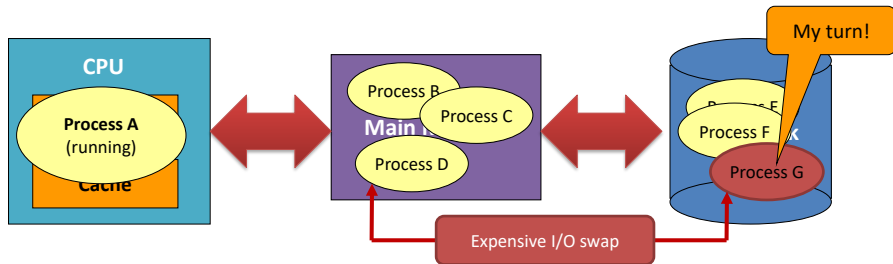
We call the entire operation:

context switching



Context switching has a price to pay...

- However, context switching may be expensive...
 - Even worse, the target process may be currently stored in the hard disk.
- So, **minimizing the number of context switching** may help boosting system performance.



Topics

- Process lifecycle
- **Process scheduling**
 - Context switching
 - **Scheduling criteria**
 - Scheduling algorithms
 - Applications/Scenarios



Scheduling Criteria

- How to choose which algorithm to use in a particular situation?

Types

Preemptive

Nonpreemptive

Application

Multiprocessor

Real-time sys

Algorithm Properties

CPU utilization

Throughput

Turnaround time

Waiting time

Response time

Application requirements and algorithm properties may vary significantly

Classes of process scheduling

- Non-preemptive scheduling.

What is it?	<p>When a process is chosen by the scheduler, the process would never leave the scheduler until...</p> <ul style="list-style-type: none">-the process voluntarily waits for I/O, or-the process voluntarily releases the CPU, e.g., exit().
What is the catch?	<p>If the process is <u>purely CPU-bound</u>, it will seize the CPU from the time it is chosen until it terminates.</p>
Pros	<p>Good for systems that emphasize the time in finishing tasks.</p> <ul style="list-style-type: none">- Because the task is running without others' interruption.
Cons	<p>Bad for nowadays systems in which user experience and multi-tasking are the primary goals.</p>
Where can I find it?	<p>Nowhere...but it could be found back in the mainframe computers in 1960s.</p>

Classes of process scheduling

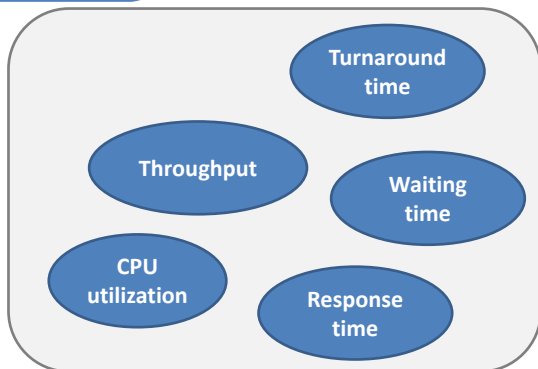
- Preemptive scheduling.

What is it?	<p>When a process is chosen by the scheduler, the process would never leave the scheduler until...</p> <ul style="list-style-type: none">-the process voluntarily waits for I/O, or-the process voluntarily releases the CPU, e.g., <code>exit()</code>.-particular kinds of interrupts and events are detected.
What is the catch?	<p>If that particular event is the periodic clock interrupt, then you can have a time-sharing system.</p>
Pros	<p>Good for systems that emphasize interactiveness.</p> <ul style="list-style-type: none">- Because every task will receive attentions from the CPU.
Cons	<p>Bad for systems that emphasize the time in finishing tasks.</p>
Where can I find it?	<p>Everywhere! This is the design of nowadays systems.</p>

Performance measures

In algorithm design:

What factors/performance measures should be carefully considered?



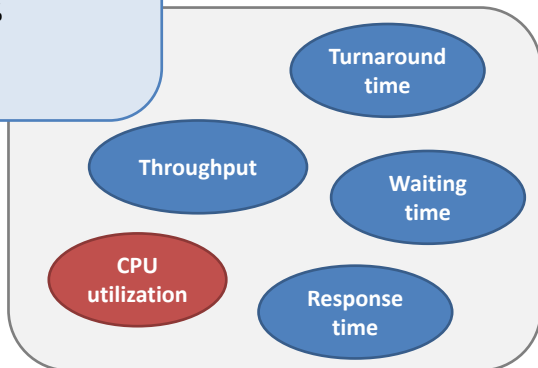
Performance measures

CPU utilization.

We want to keep CPU as busy as possible.

Theoretically, can range from 0-100%, but in real system, range from 40%-90%

The higher the better

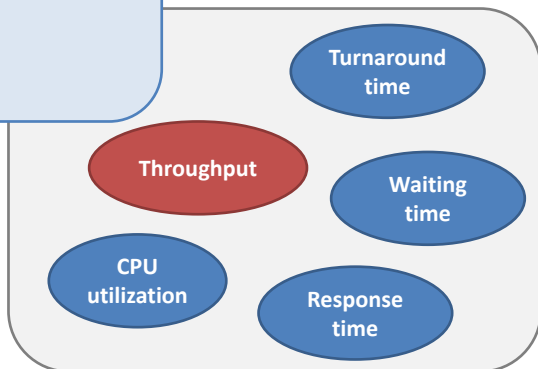


Performance measures

Throughput.

Number of processes that are completed per time unit

The higher the better

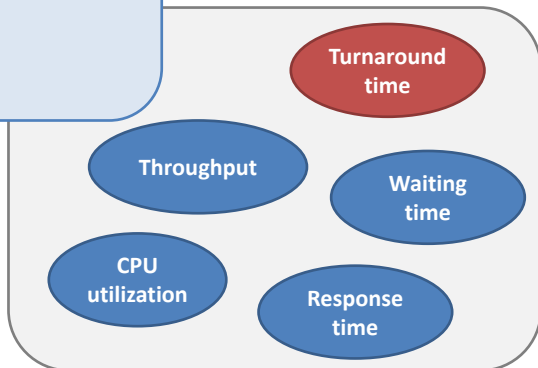


Performance measures

Turnaround time.

Time to execute the process: interval from the time of submission to the time of completion (total running time + waiting time + doing I/O)

The lower the better

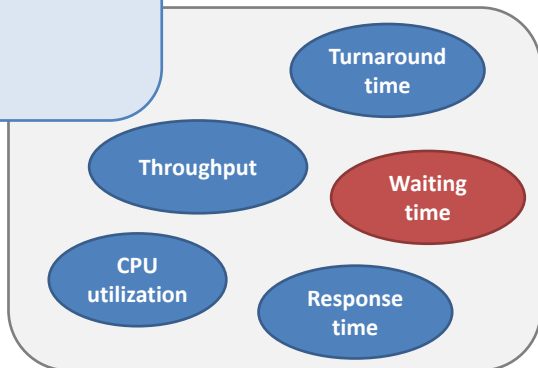


Performance measures

Waiting time.

The time spent waiting in the ready queue

The lower the better

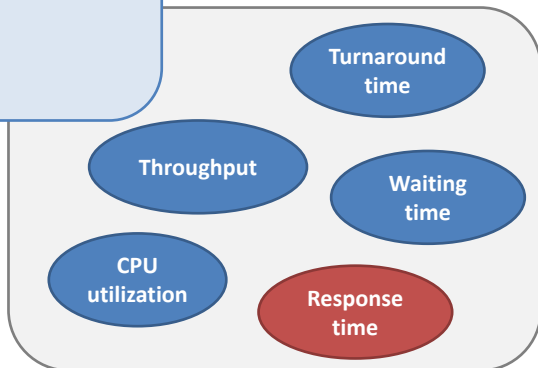


Performance measures

Response time.

The time from the submission of a request until the first response is produced (useful measure for interactive systems)

The lower the better



Challenge

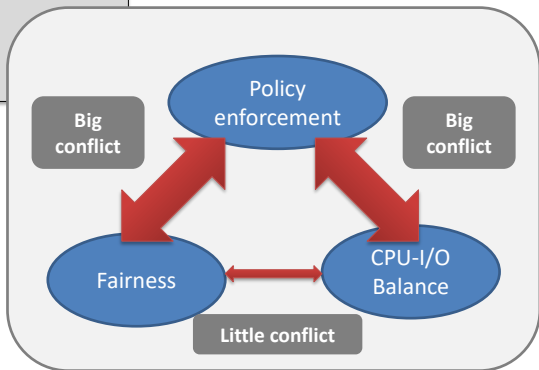
Question:

Can we optimize all the above measures simultaneously?

Usually can not!

**Design
Tradeoff**

**Common
goal**



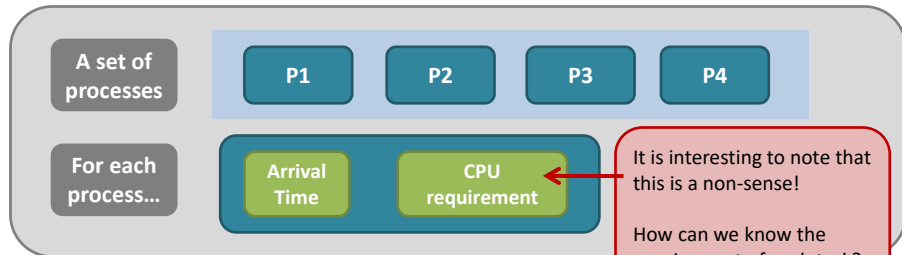
Topics

- Process lifecycle
- **Process scheduling**
 - Context switching
 - Scheduling criteria
 - **Scheduling algorithms**
 - Applications/Scenarios



Scheduling algorithms

- Inputs to the algorithms.



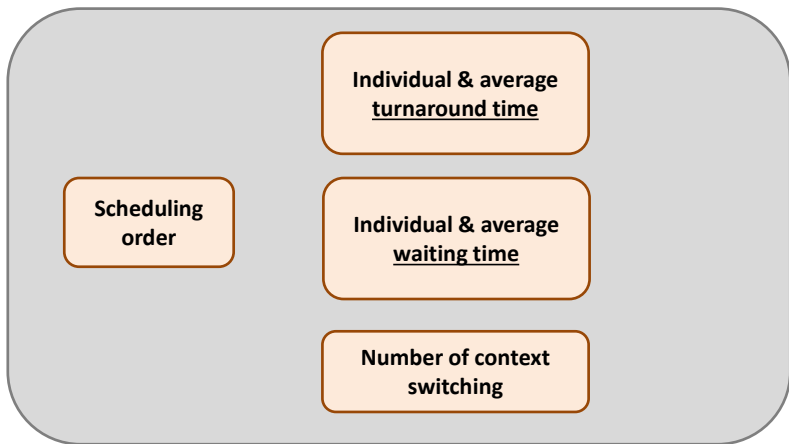
Online VS Offline

An **offline scheduling algorithm** assumes that you know all the processes submitted to the system before hand. But, an **online scheduling algorithm** does not have such an assumption.

Yet, every real scheduler has to work in an “**online scenario**”. So, we have to think in an “online” way...

Scheduling algorithms

- **Outputs of the algorithms.**



Different algorithms

Algorithms	Preemptive?	Target System
First-come, first-served or First-in, First-out (FIFO)	No.	Out-of-date
Shortest-job-first (SJF)	Can be both.	Out-of-date
Round-robin (RR)	Yes.	Modern
Priority scheduling	Yes.	Modern
Priority scheduling with multiple queues.	The real implementation!	

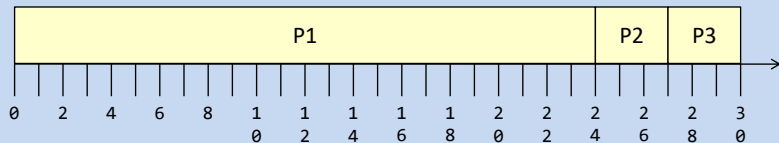
First-come, first-served scheduling

- Example 1.

Gantt Chart

No preemption

Output



Waiting time: P1 = 0; P2 = 23; P3 = 25;

Average waiting time = $(0+23+25)/3 = 16$;

Turnaround time: P1 = 24; P2 = 26; P3 = 28;

Average turnaround time = $(24+26+28)/3 = 26$;

Task	Arrival Time	CPU Req.
P1	0	24
P2	1	3
P3	2	3

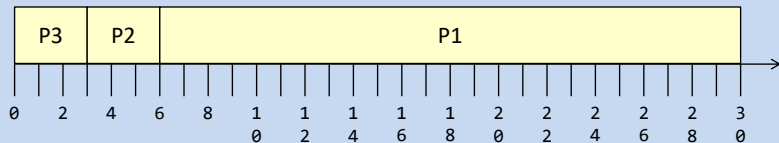
Input

First-come, first-served scheduling

- Example 2.

Gantt Chart

Output



Waiting time: P1 = 4; P2 = 2; P3 = 0;

Average waiting time = $(4+2+0)/3 = 2$;
(which is 16 in the previous case)

Turnaround time: P1 = 28; P2 = 5; P3 = 3;

Average turnaround time = $(28+5+3)/3 = 12$;
(which is 26 in the previous case)

Task	Arrival Time	CPU Req.
P3	0	3
P2	1	3
P1	2	24

Input order
changed

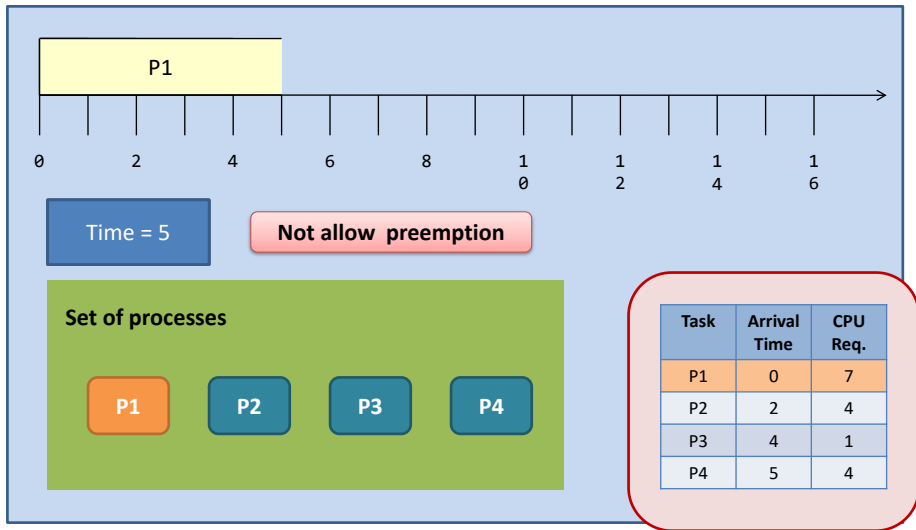
First-come, first-served scheduling

- A short summary:
 - FIFO scheduling is sensitive to the input.
 - The average waiting time is often long. Think about the scenario (convoy effect):
 - Someone is standing before you in the queue in KFC, and
 - you find that he/she is ordering the **bucket chicken meal** (P1 in example 1)!!!!
 - So, two people (P2 and P3) are unhappy while only P1 is happy.
 - Can we do something about this?

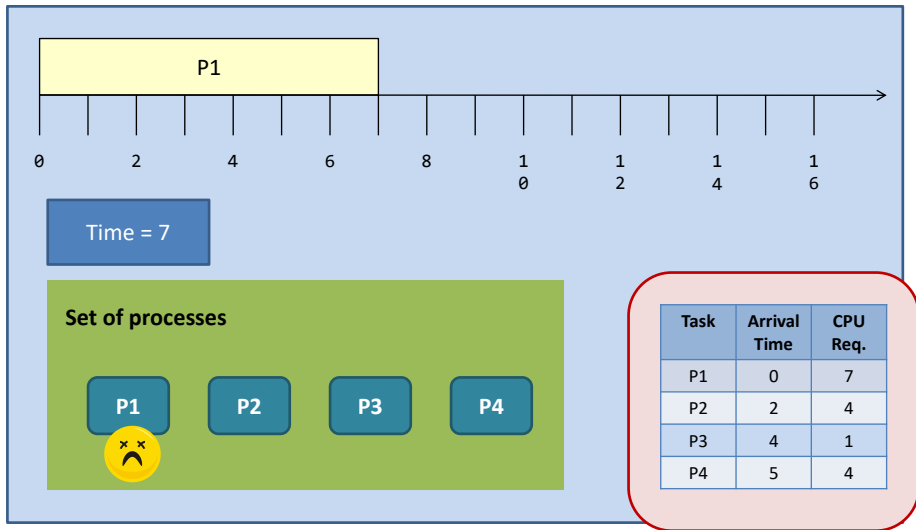
Different algorithms

Algorithms	Preemptive?	Target System
First-come, first-served or First-in, First-out (FIFO)	No.	Out-of-date
Shortest-job-first (SJF)	Can be both.	Out-of-date
Round-robin (RR)	Yes.	Modern
Priority scheduling	Yes.	Modern
Priority scheduling with multiple queues.	The real implementation!	

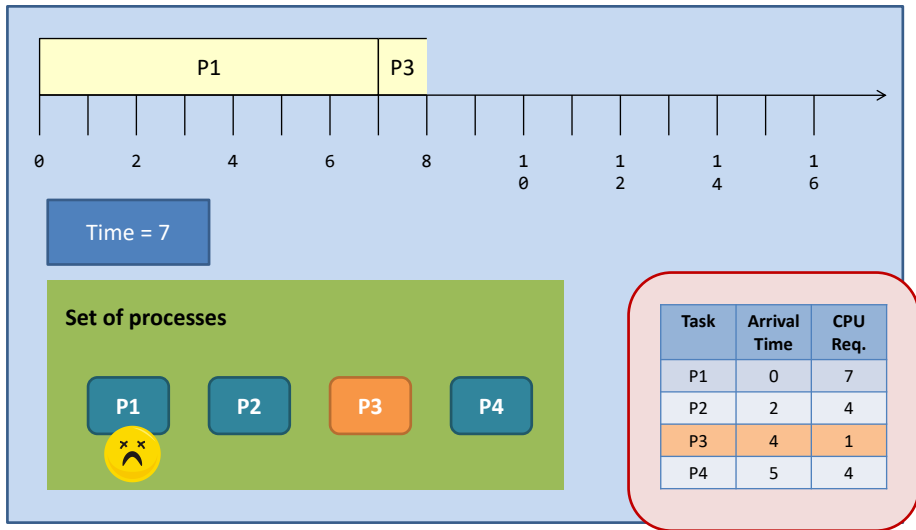
Non-preemptive SJF



Non-preemptive SJF

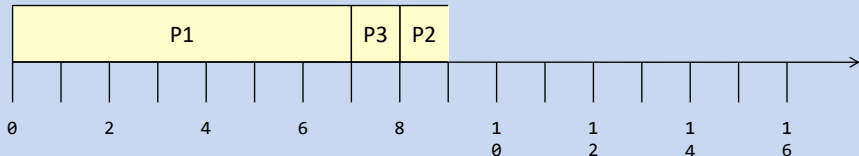


Non-preemptive SJF



Non-preemptive SJF

In this example, we use **FIFO** to break the tie.



Time = 8

Set of processes

P1



P2

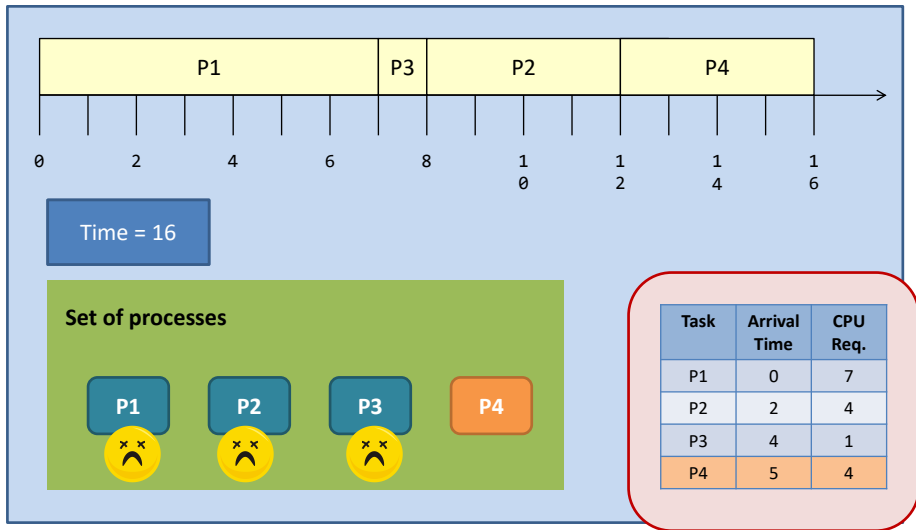
P3



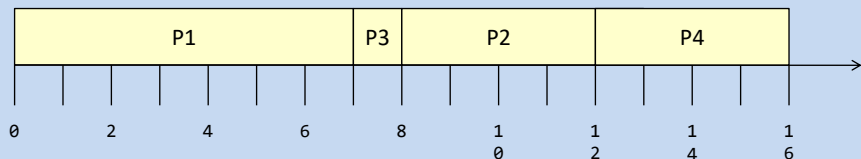
P4

Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Non-preemptive SJF



Non-preemptive SJF



Waiting time:

$P1 = 0; P2 = 6; P3 = 3; P4 = 7;$

$Average = (0 + 6 + 3 + 7) / 4 = 4.$

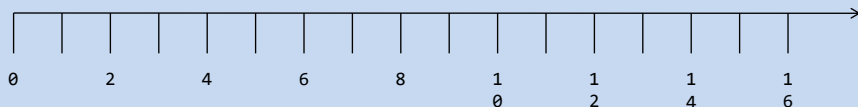
Turnaround time:

$P1 = 7; P2 = 10; P3 = 4; P4 = 11;$

$Average = (7 + 10 + 4 + 11) / 4 = 8.$

Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Preemptive SJF



Rules for preemptive scheduling

(for this example only)

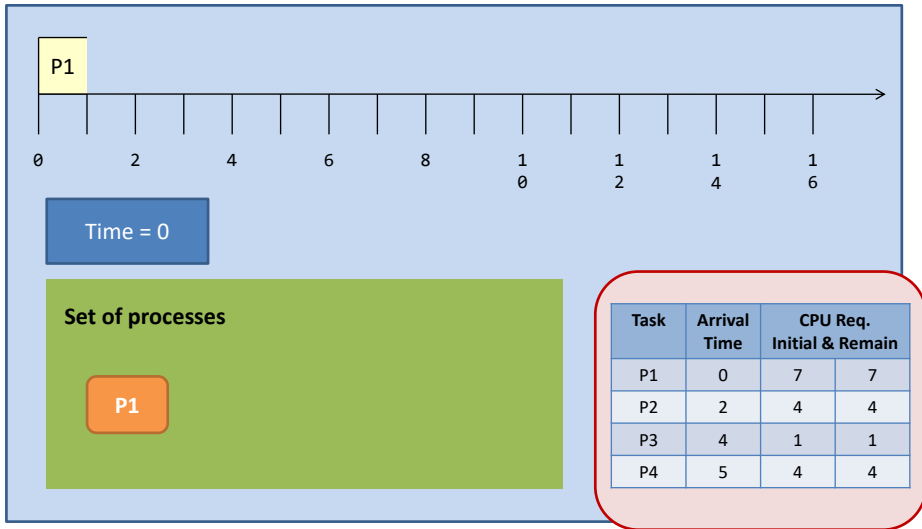
-Preemption happens when a new process arrives at the system.

-Then, the scheduler steps in and selects the next task based on **their remaining CPU requirements**.

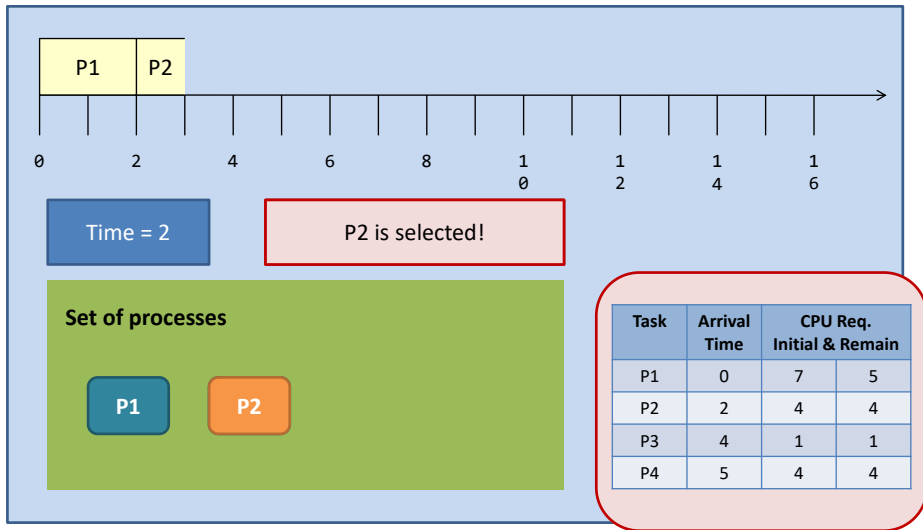
Shortest-remaining-time-first

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	7
P2	2	4	4
P3	4	1	1
P4	5	4	4

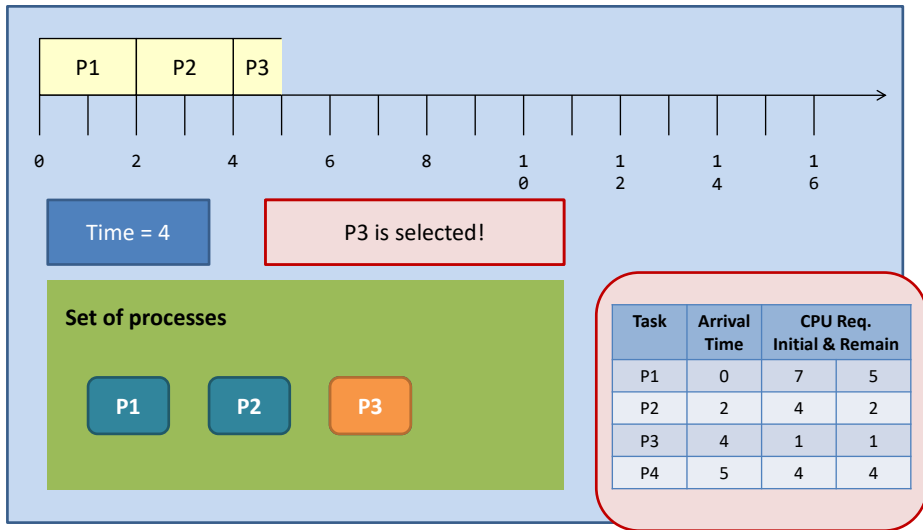
Preemptive SJF



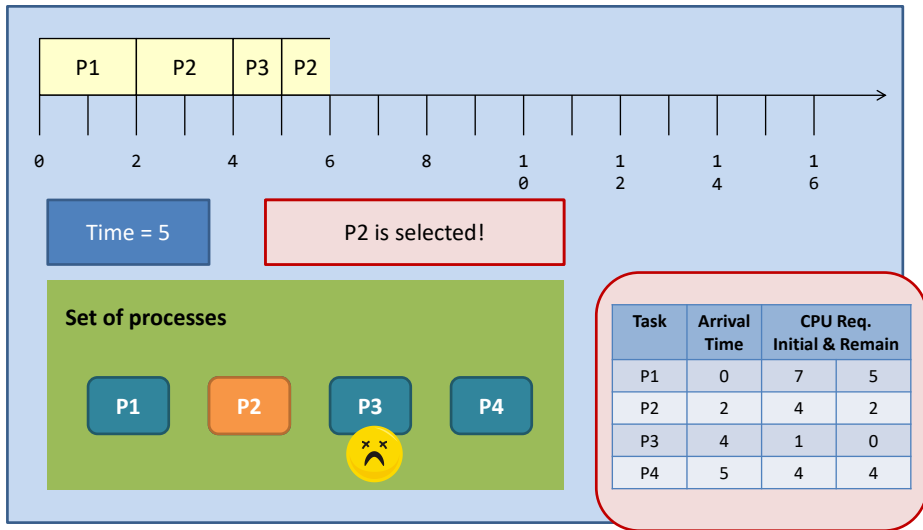
Preemptive SJF



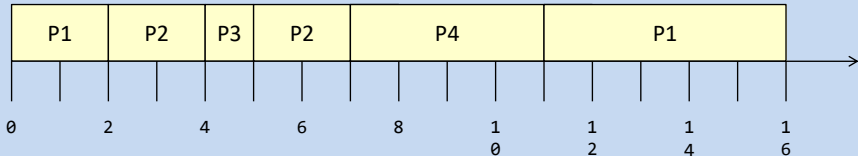
Preemptive SJF



Preemptive SJF

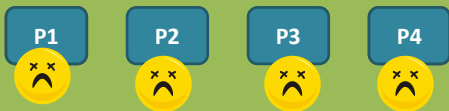


Preemptive SJF



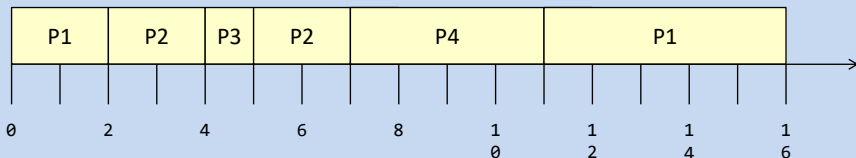
Time = 16

Set of processes



Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	0
P2	2	4	0
P3	4	1	0
P4	5	4	0

Preemptive SJF



Waiting time:

$P1 = 9; P2 = 1; P3 = 0; P4 = 2;$

$Average = (9 + 1 + 0 + 2) / 4 = 3.$

Turnaround time:

$P1 = 16; P2 = 5; P3 = 1; P4 = 6;$


$Average = (16 + 5 + 1 + 6) / 4 = 7.$

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	0
P2	2	4	0
P3	4	1	0
P4	5	4	0

SJF: Short summary

	Non-preemptive SJF	Preemptive SJF
Average waiting time	4	3 (smallest)
Average turnaround time	8	7 (smallest)
# of context switching	3 (smallest)	5

The waiting time and the turnaround time decrease at the expense of the **increased number of context switching**.




Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

SJF: Short summary

	Non-preemptive SJF	Preemptive SJF
Average waiting time	4	3 (smallest)
Average turnaround time	8	7 (smallest)
# of context switching	3 (smallest)	5

SJF is provably optimal in that it gives the **minimum average waiting time**

Challenge: How to know the length of the next CPU request?



Task	Arrival Time	CPU Req.
P1	0	7
P2	2	4
P3	4	1
P4	5	4

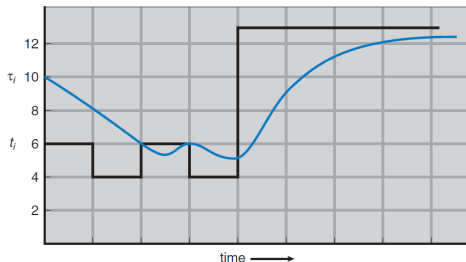
SJF: Short summary

Challenge: How to know the length of the next CPU request?

Solution: Prediction (by expecting that the next CPU burst will be similar in length to the previous ones)

General approach
exponential average

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n.$$



CPU burst (t_i)	6	4	6	4	13	13	13	...	
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Predicted
value

Most recent information

Different algorithms

Algorithms	Preemptive?	Target System
First-come, first-served or First-in, First-out (FIFO)	No.	Out-of-date
Shortest-job-first (SJF)	Can be both.	Out-of-date
Round-robin (RR)	Yes.	Modern
Priority scheduling	Yes.	Modern
Priority scheduling with multiple queues.	The real implementation!	

Round-robin

- Round-Robin (RR) scheduling is preemptive.
 - Every process is given a **quantum**, or the amount of time allowed to execute.
 - When the quantum of a process is **used up** (i.e., 0), the process releases the CPU and **this is the preemption**.
 - Then, the scheduler steps in and it chooses **the next process which has a non-zero quantum** to run.
- Processes are running one-by-one, like a circular queue.
 - Designed specially for time-sharing systems

Round-robin

Rules for Round-Robin

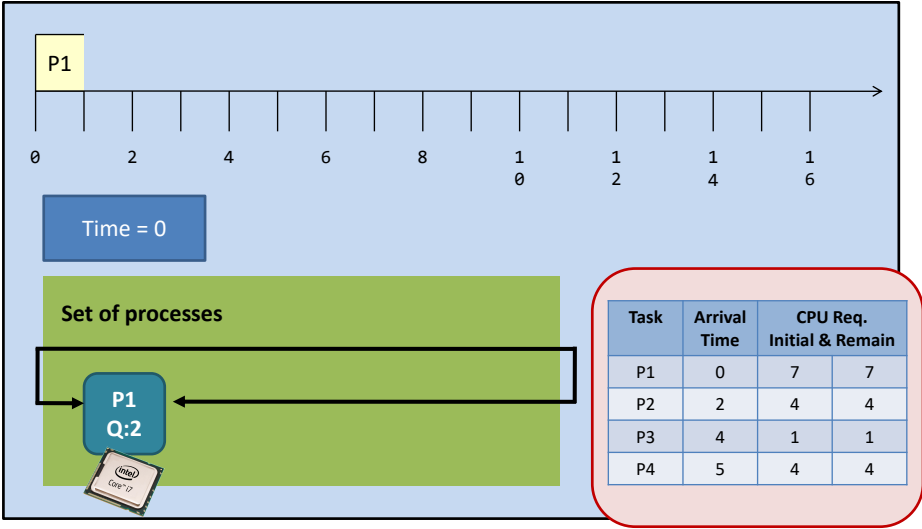
(for this example only)

-The quantum of every process is fixed and is 2 units.

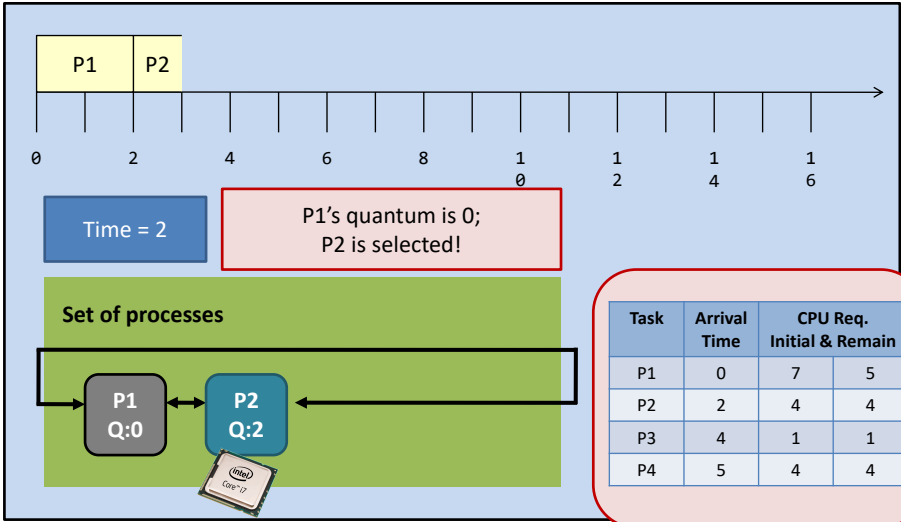
-The process queue is sorted according the processes' arrival time, in an ascending order.
(This rule allows us to break tie.)

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	7
P2	2	4	4
P3	4	1	1
P4	5	4	4

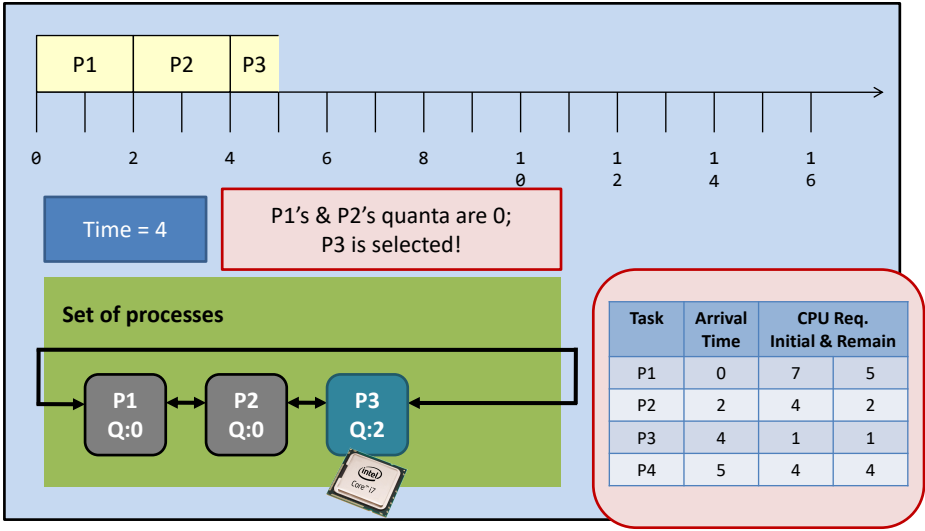
Round-robin



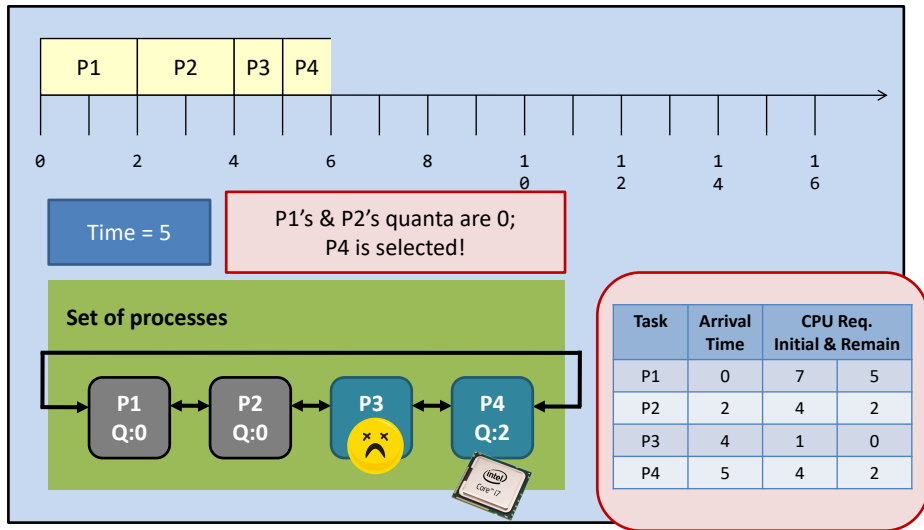
Round-robin



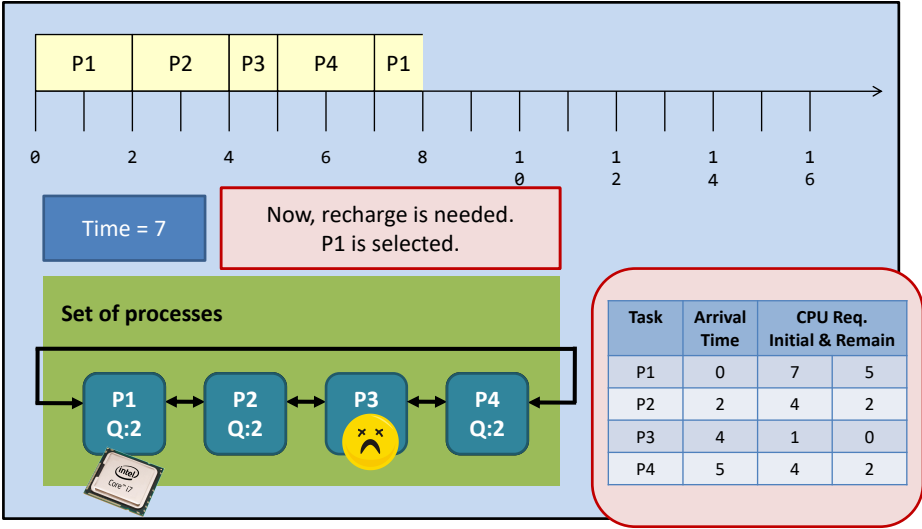
Round-robin



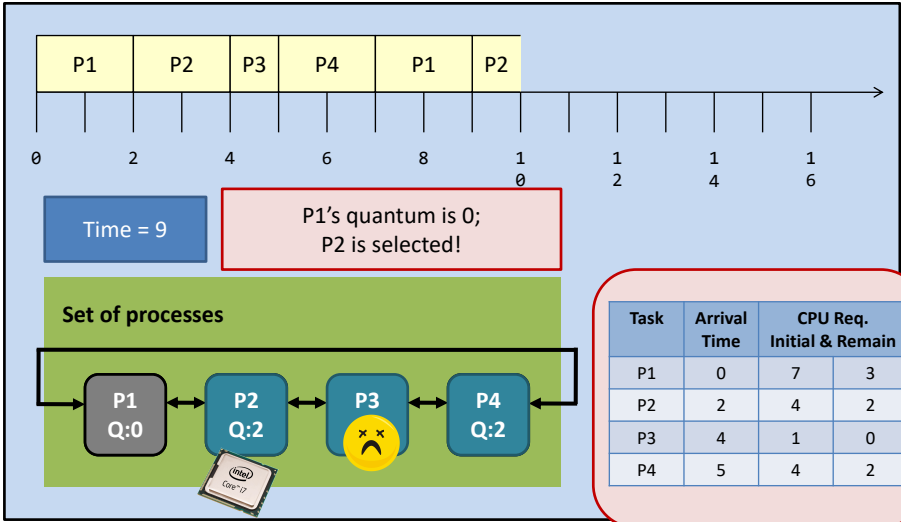
Round-robin



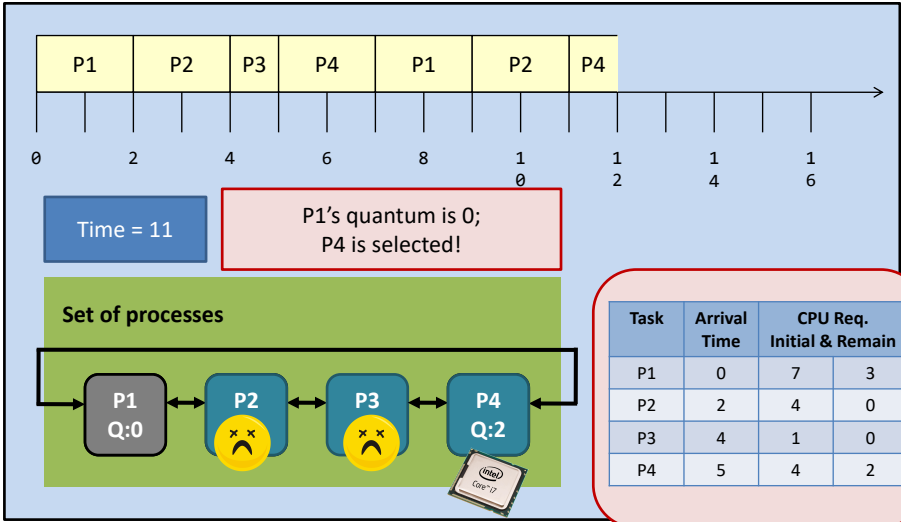
Round-robin



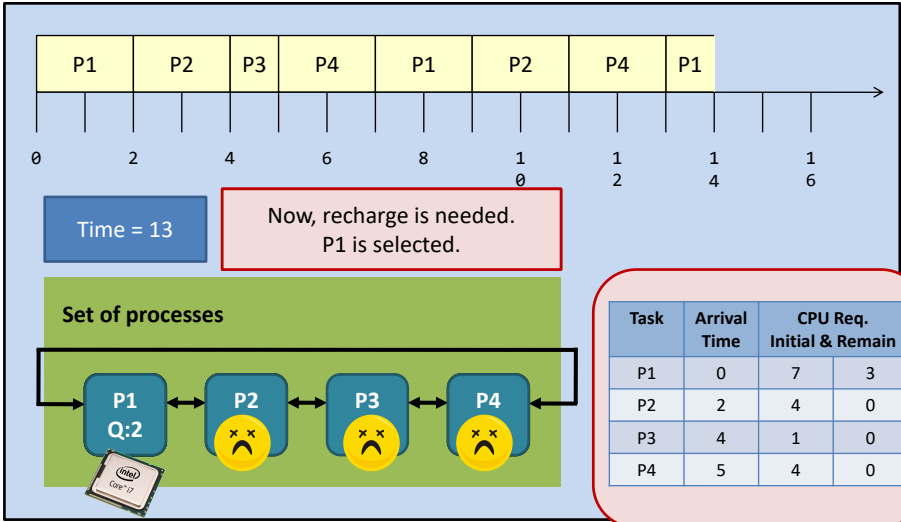
Round-robin



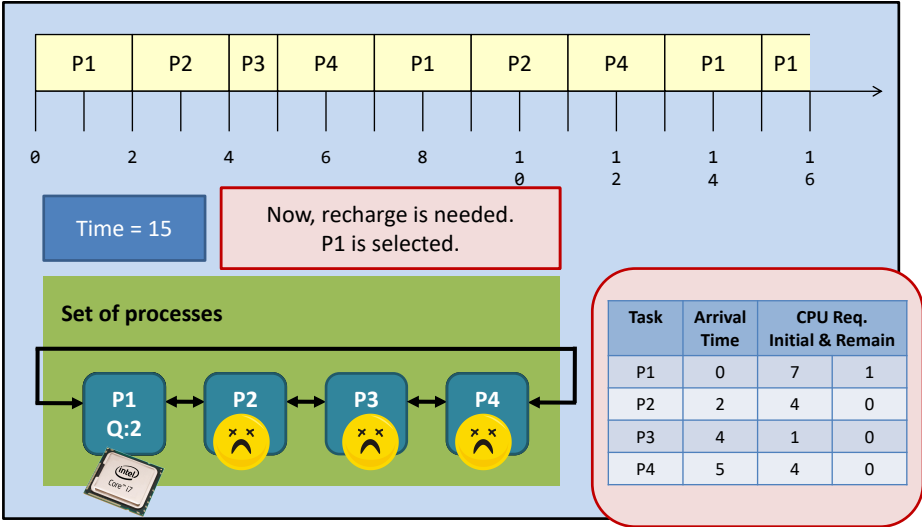
Round-robin



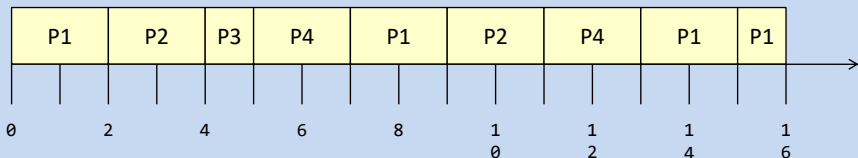
Round-robin



Round-robin



Round-robin



Waiting time:

$P1 = 9; P2 = 5; P3 = 0; P4 = 4;$

$Average = (9 + 5 + 0 + 4) / 4 = 4.5$

Turnaround time:

$P1 = 16; P2 = 9; P3 = 1; P4 = 8;$

$Average = (16 + 9 + 1 + 8) / 4 = 8.5$

Task	Arrival Time	CPU Req. Initial & Remain	
P1	0	7	0
P2	2	4	0
P3	4	1	0
P4	5	4	0

RR VS SJF

	Non-preemptive SJF	Preemptive SJF	RR
Average waiting time	4	3	4.5 (largest)
Average turnaround time	8	7	8.5 (largest)
# of context switching	3	5	8 (largest)



So, the RR algorithm gets all the bad! Why do we still need it?

The responsiveness of the processes is great under the RR algorithm. E.g., you won't feel a job is "frozen" because every job is on the CPU from time to time!

Round-robin

Issue for Round-Robin

-How to set the size of the time quantum?

-**Too large:** FCFS

-**Too small:** frequent context switch

-**In practice:** 10-100ms

-**A rule of thumb:** 80% CPU burst should be shorter than the time quantum

Observations on RR

- Modified versions of round-robin are implemented in (nearly) every modern OS.
 - Users run a lot of **interactive jobs** on modern OS-es.
 - Users' priority list:
 - Number one - Responsiveness;
 - Number two - Efficiency;
 - In other words, “ordinary users” expect a fast GUI response than an efficient scheduler running behind.
- With the round-robin deployed, the scheduling **looks like random**.
 - It also looks like “*fair to all processes*”.

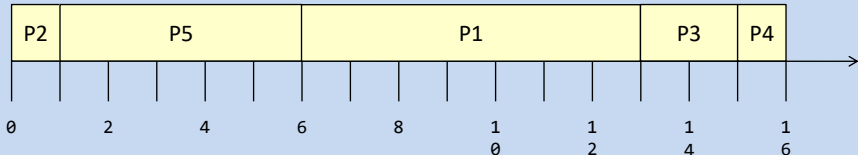
Different algorithms

Algorithms	Preemptive?	Target System
First-come, first-served or First-in, First-out (FIFO)	No.	Out-of-date
Shortest-job-first (SJF)	Can be both.	Out-of-date
Round-robin (RR)	Yes.	Modern
Priority scheduling	Yes.	Modern
Priority scheduling with multiple queues.	The real implementation!	

Priority Scheduling

- Some basics:
 - A task is given a priority (and is usually an integer).
 - A scheduler selects the next process based on the priority.
 - ***A typical practice***: the highest priority is always chosen.
 - Special case: SJF, FCFS (equal priority)
- How to define priority
 - Internally: time limits, memory requirements, number of open files, CPU burst and I/O burst...
 - Externally: process importance, paid funds...

Priority Scheduling



Assumption:

- All arrive at time 0
- Low numbers represent high priority

Problem: Indefinite blocking or starvation

Solution: Aging (gradually increase the priority of waiting processes)

Task	CPU Burst	Priority
P1	7	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Different algorithms

Algorithms	Preemptive?	Target System
First-come, first-served or First-in, First-out (FIFO)	No.	Out-of-date
Shortest-job-first (SJF)	Can be both.	Out-of-date
Round-robin (RR)	Yes.	Modern
Priority scheduling	Yes.	Modern
Priority scheduling with multiple queues.	The real implementation!	

Multilevel queue scheduling

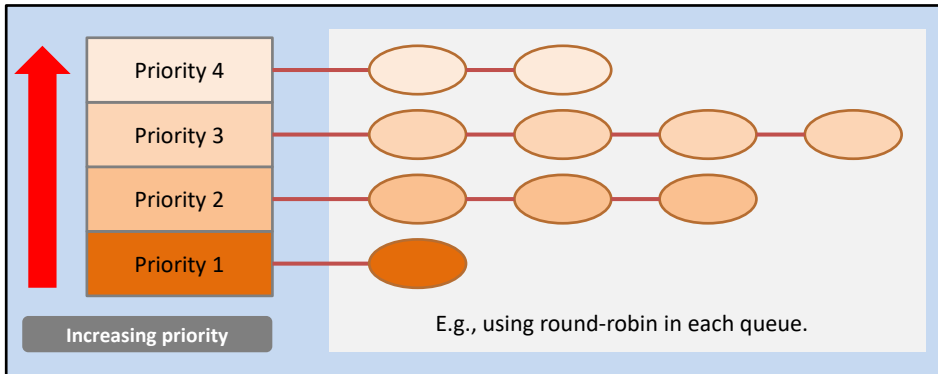
- **Definitions.**

- It is still a priority scheduler.
- But, at each priority class, **different schedulers** may be deployed.
- Eg: Foreground processes and background processes

Priority class 5	Non-preemptive, FIFO	<div>Just an example.</div> <div>The processes are permanently assigned to one queue</div> <div>Fixed-priority preemptive scheduling among queues</div>
Priority class 4	Non-preemptive, SJF	
Priority class 3	RR with quantum = 10 units.	
Priority class 2	RR with quantum = 20 units.	
Priority class 1	RR with quantum = 40 units.	

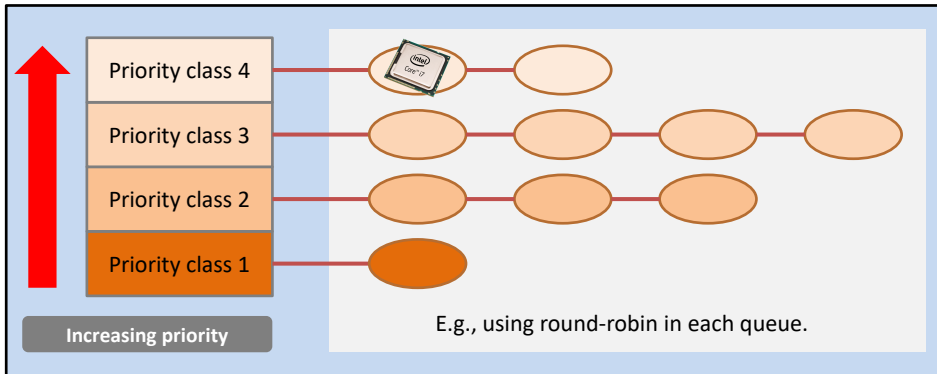
Multilevel queue scheduling– an example

- **Properties:** process is assigned a fix priority when they are submitted to the system.



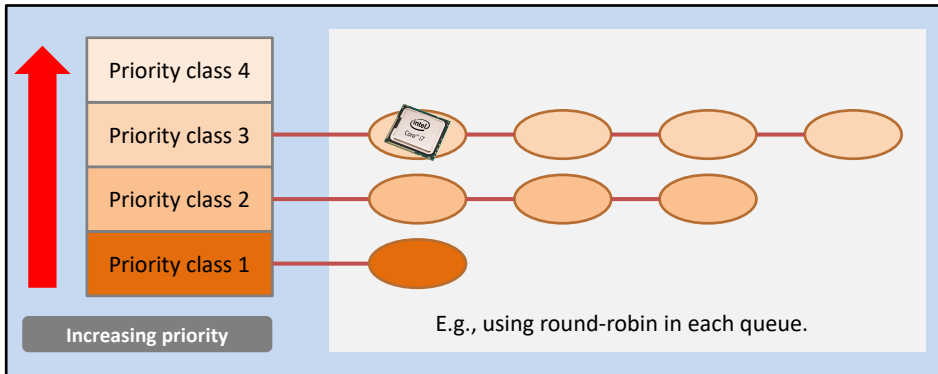
Multilevel queue scheduling– an example

- The highest priority class will be selected.
 - To prevent high-priority tasks from running indefinitely.
 - The tasks with a higher priority should be short-lived, but important;



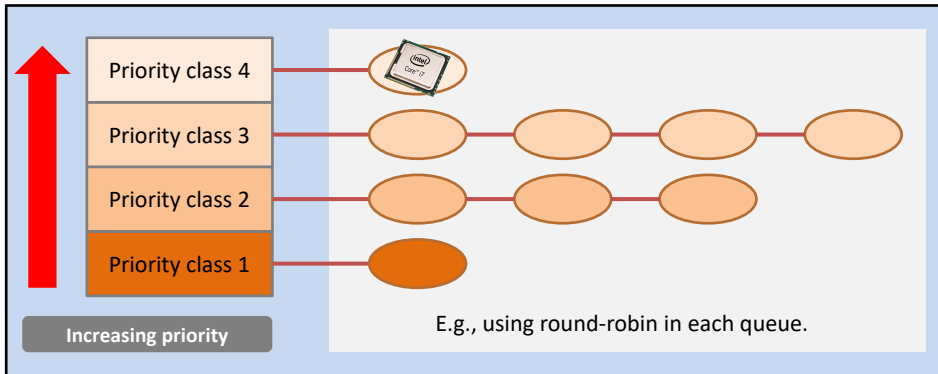
Multilevel queue scheduling– an example

- Lower priority classes will be scheduled only when the upper priority classes has no tasks.



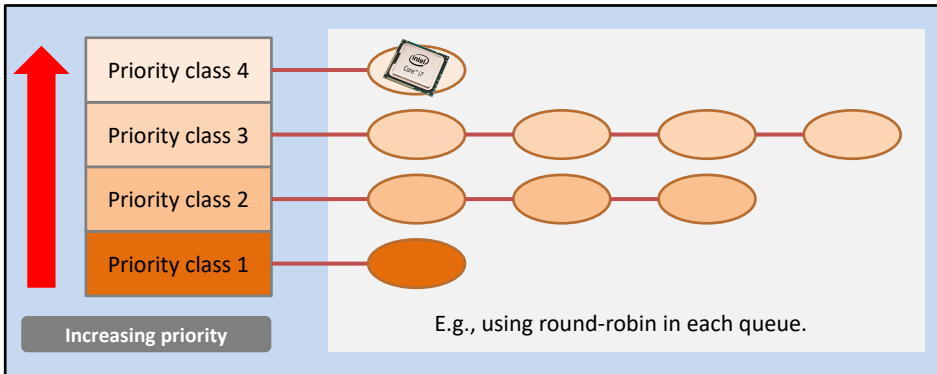
Multilevel queue scheduling– an example

- Of course, it is a good design to have a high-priority task preempting a low-priority task.
(conditioned that the high-priority task is short-lived.)



Multilevel queue scheduling– an example

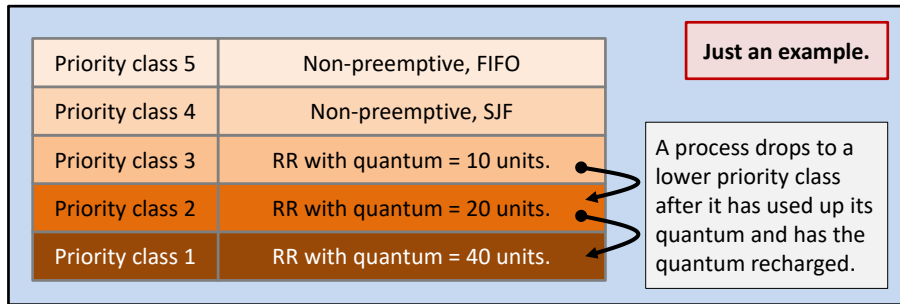
- Any problem?
 - Fixed priority
 - Indefinite blocking or starvation



Multilevel feedback queue scheduling

- **How to improve the previous scheme?**

- Allows a process to move between queues (dynamic priority).
- Why needed?
 - Eg.: Separate processes according to their CPU bursts.

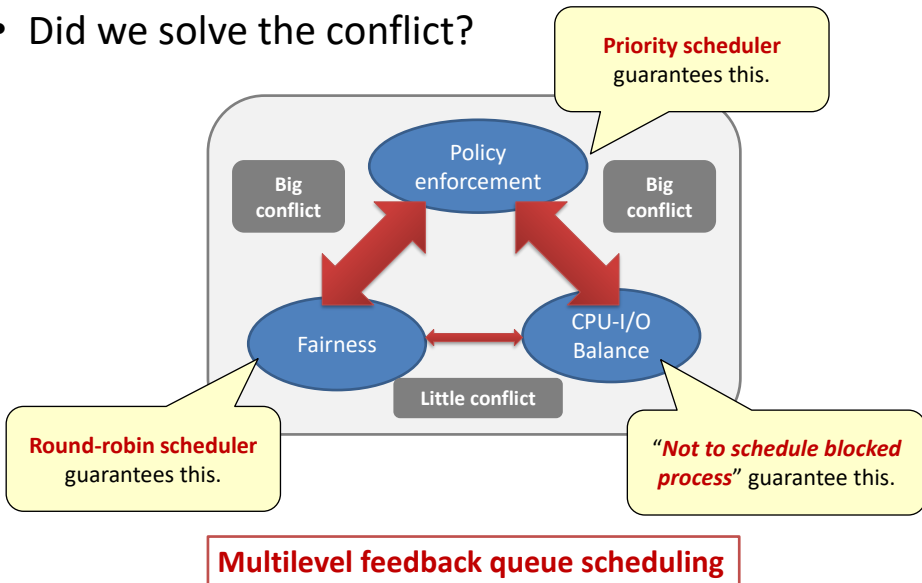


Multilevel feedback queue scheduling

- How to design (factors)?
 - Number of queues
 - Scheduling algorithm for each queue
 - Method for determining when to upgrade/downgrade a process
 - Method for determining which queue a process will enter
- Most general, but also most complex
 - Can be configured to match a specific system

Summary

- Did we solve the conflict?



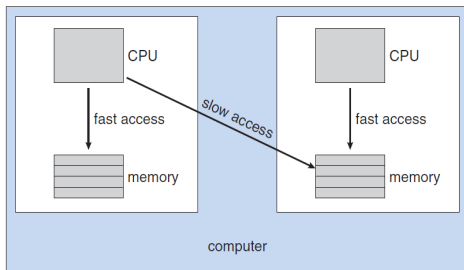
- **Applications/Scenarios**
 - Multiple processors
 - Real-time systems
 - Example: Linux scheduler
 - Algorithm evaluation



- **Applications/Scenarios**
 - **Multiple processors**
 - Real-time systems
 - Example: Linux scheduler
 - Algorithm evaluation



Scheduling Issues with SMP



SMP: Each processor may have its private queue of ready processes

Scheduling between processors

Process migration: Invalidating the cache of the first processor and repopulating the cache of the second processor)

Process migration is costly

Processor Affinity

Attempt to keep a process running on the same processor

Soft/hard affinity

NUMA

CPU scheduler and memory-placement algorithms work together

Load balancing

Push migration: a specific task periodically check the status & rebalance

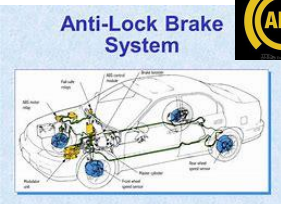
Pull migration: an idle processor pulls a waiting task from busy processor

No absolute rule concerning what policy is best

- **Applications/Scenarios**
 - Multiple processors
 - **Real-time systems**
 - Example: Linux scheduler
 - Algorithm evaluation



Real-time CPU Scheduling



Antilock brake system: Latency requirement: 3-5 ms

Hard real-time systems: A task must be served by its deadline (otherwise, expired as no service at all)

Soft real-time systems: Critical processes will be given preference over noncritical processes (no guarantee)

Responsiveness: Respond immediately to a real-time process as soon as it requires the CPU

Support priority-based alg. with preemption

Interrupt latency (minimize or bounded):

- ✓ Determining interrupt type and save the state of the current process
- ✓ Minimize the time interrupts may be disabled

Dispatch latency:

- ✓ Time required by dispatcher (preemption running process and release resources of low-priority proc).
- ✓ Most effective way is to use preemptive kernel

Real-time CPU Scheduling Algorithms

Rate monotonic scheduling

Assumption: Processes require CPU at constant periods: processing time t and period p (rate $1/p$)

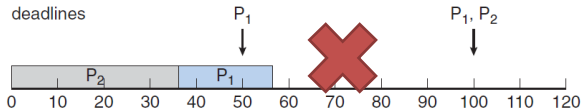
Each process is assigned a priority proportional to its rate, and schedule processes with a static priority policy with preemption (**fixed priority**)

Example

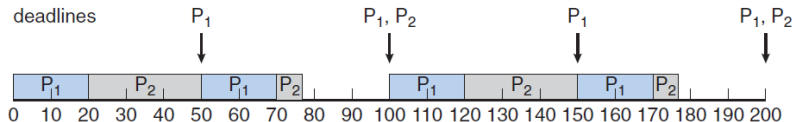
P1: $p_1=50, t_1=20$

P2: $p_2=100, t_2=35$

deadlines



deadlines



Real-time CPU Scheduling Algorithms

Rate monotonic scheduling

Any problem?

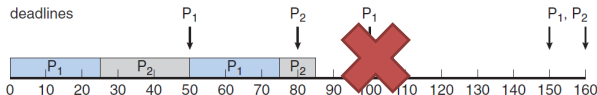
Processes require CPU at constant periods: processing time t and period p (rate $1/p$)

Each process is assigned a priority proportional to its rate, and schedule processes with a static priority policy with preemption (**fixed priority**)

Example

P1: $p_1=50$, $t_1=25$

P2: $p_2=80$, $t_2=35$



Can not guarantee that a set of processes can be scheduled

Real-time CPU Scheduling Algorithms

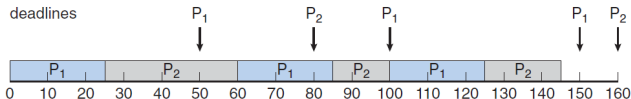
Earliest-deadline-first scheduling (EDF)

Dynamically assigns priorities according to deadline (the earlier the deadline, the higher the priority)

Example

P1: $p1=50$, $t1=25$

P2: $p2=80$, $t2=35$



EDF does not require the processes to be periodic, nor require a constant CPU time per burst

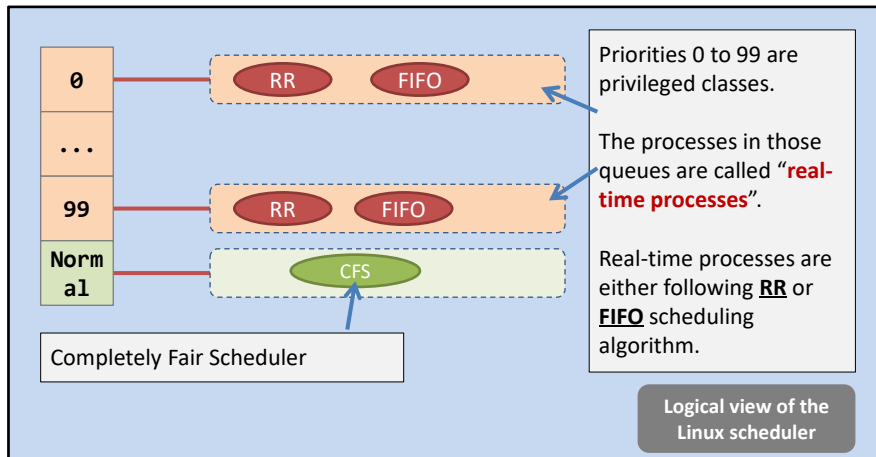
EDF requires the announcement of deadlines

- **Applications/Scenarios**
 - Multiple processors
 - Real-time systems
 - **Example: Linux scheduler**
 - Algorithm evaluation



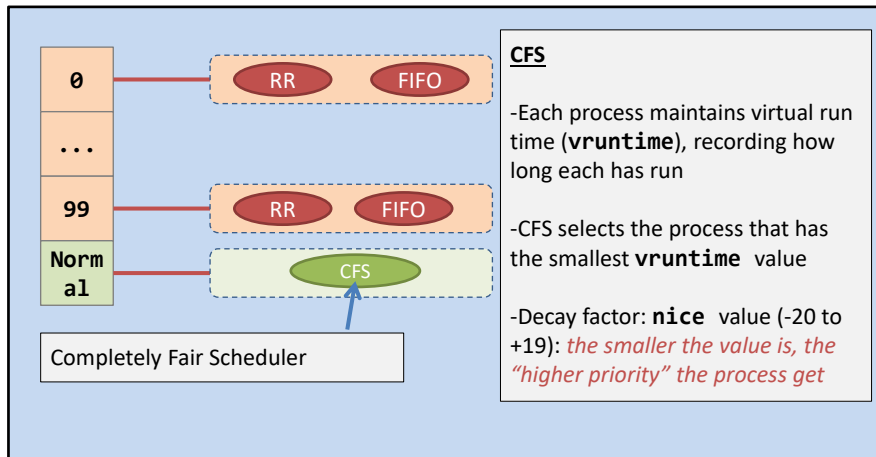
Linux Scheduler

- A multiple queue, (kind of) static priority scheduler.



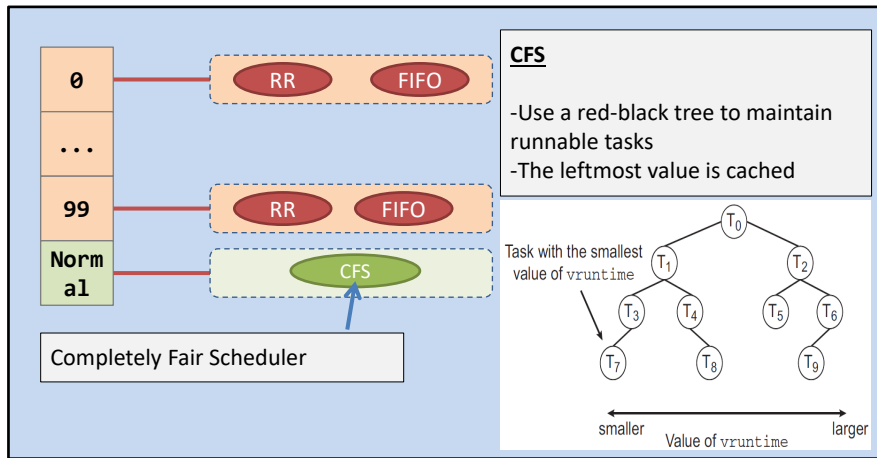
Linux Scheduler

- A multiple queue, (kind of) static priority scheduler.



Linux Scheduler

- A multiple queue, (kind of) static priority scheduler.



- **Applications/Scenarios**
 - Multiple processors
 - Real-time systems
 - Example: Linux scheduler
 - **Algorithm evaluation**



How to select/evaluate a scheduling algorithm?

How to select a scheduling alg? (many algorithms with different parameters and properties)

Step 1: Define a criteria or the importance of various measures (application dependent)

Step 2: Design/Select an algorithm to satisfy the requirements. How to guarantee?

Evaluate Algorithms

Deterministic modeling

Simple and fast

Demonstration examples

Queueing modeling

Queueing network analysis

Distribution of CPU and I/O burst (Poisson arrival)

Little's law: $n = \lambda \times W$

Simulation & Implementation

Trace driven

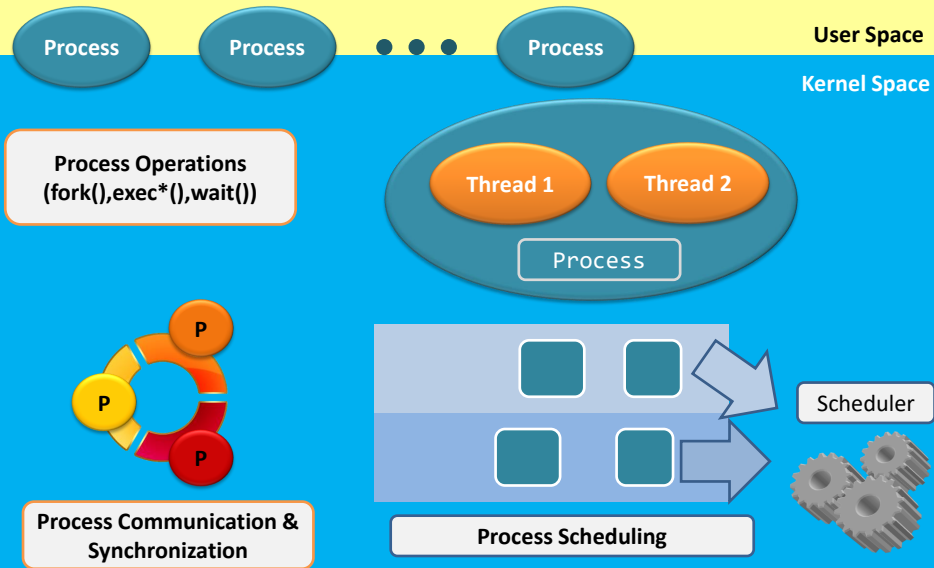
High cost (coding/debugging...)

Hard to understand the full design space

Summary on scheduling

- So, you may ask:
 - “What is the best scheduling algorithm?”
 - “What is the standard scheduling algorithm?”
- There is **no best or standard** algorithm because of, *at least*, the following reasons:
 - No one could predict how many clock ticks does a process requires.
 - On modern OS-es, processes are submitted online.
 - Conflicting criterias

Summary on part 2



Operating Systems

Associate Prof. Yongkun Li

中科大-计算机学院 副教授

<http://staff.ustc.edu.cn/~ykli>

Ch7

Memory Management from a Programmer's Perspective

Why we need memory management

- The running program code requires memory
 - Because the CPU needs to fetch the instructions from the memory for execution
- We must keep several processes in memory
 - Improve both CPU utilization and responsiveness
 - Multiprogramming

It is required to efficiently manage the memory

Topics in Ch7

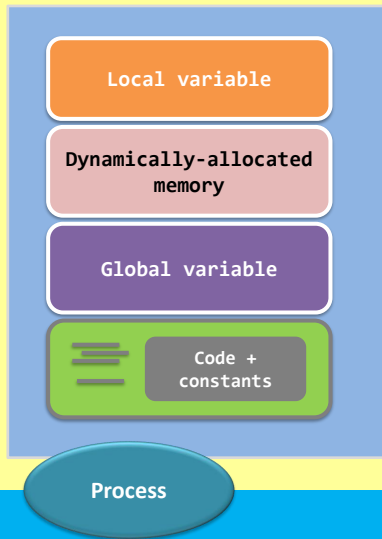
From a programmer's perspective: user-space memory management

- What is the address space of a process?
- How are the program code and data stored in memory?
- How to allocate/free memory (`malloc()` + `free()`)?
- How much memory can be used in a program?
- What are segmentation and segmentation fault?

From the kernel's perspective: How to manage the memory

- What is virtual memory?
- How to realize address mapping (paging)?
- How to support very large programs (demand paging)?
- How to do page replacement?
- What is TLB?
- What is memory-mapped file?

Part 1: User-space memory



Do you remember this?

- Content of a process (in user-space memory)

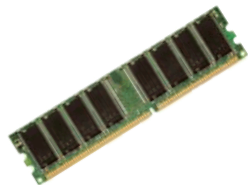
How does each part use the memory?

- From a programmer's perspective

Let's forget about the kernel for a moment. We are going to explore the **user-space memory** first.

User-space memory management

- **Address space;**
- Code & constants;
- Data segment;
- Stack;
- Heap;
- Segmentation fault;



Address space

How does a programmer look at the memory space?

- An array of bytes?
- Memory of a process is divided into segments
- This way of arranging memory is called **segmentation**

Stack - Local variables

Heap - Dynamically allocated memory

Data Segment & BSS - Global and static variables

Code + Constant

Address space

```
int main(void) {  
    int *malloc_ptr = malloc(4);  
    char *constant_ptr = "hello";  
  
    printf("Local variable = %15p\n", &malloc_ptr);  
    printf("malloc() space = %15p\n", malloc_ptr);  
    printf("Global variable = %15p\n", &global_int);  
    printf("Code & constant = %15p\n", constant_ptr);  
  
    return 0;  
}
```

\$./addr

Local variable = 0xbfa8938c
malloc() space = 0x915c008
Global variable = 0x804a020
Code & constant = 0x8048550
\$ _

Note

The addresses are not necessarily the same in different processes

What is the process address space?

Increasing
address

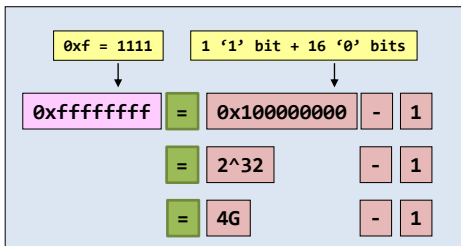
Stack - Local variables

Heap - Dynamically
allocated memory

Data Segment & BSS -
Global and static
variables

Code + Constant

Address space



In a 32-bit system,

- One address maps to one byte.
- The maximum amount of memory in a process is **4GB**.

Note

- This is the so called logical address space
- **Each process has its own address space**, and it can reside in any part of the physical memory

How large is the address space?

Increasing
address

Stack - Local variables

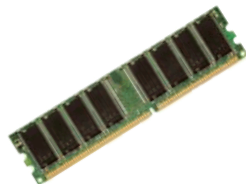
Heap - Dynamically
allocated memory

Data Segment & BSS -
Global and static
variables

Code + Constant

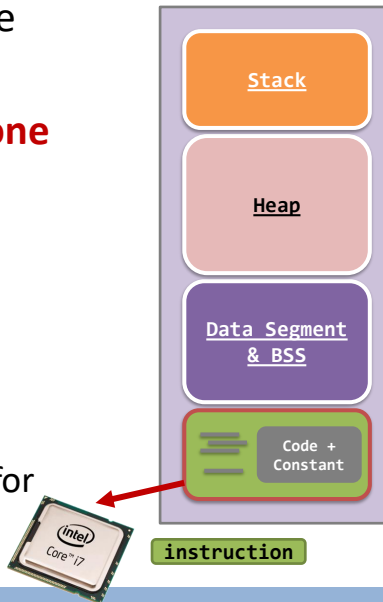
User-space memory management

- Address space;
- **Code & constants;**
- Data segment;
- Stack;
- Heap;
- Segmentation fault;



Program code & constants

- A program is an executable file
- A process is **not bounded to one program code**.
 - Remember `exec*()` family?
- The program code requires memory space because...
 - The CPU needs to fetch the instructions from the memory for execution.



Program code & constants

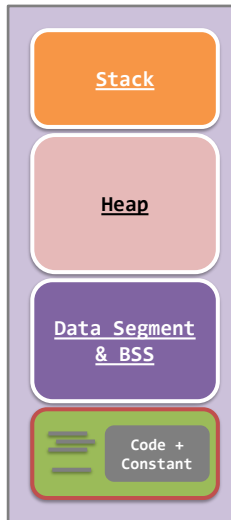
```
1 int main(void) {  
2     char *string = "hello";  
3     printf("\nhello\"      = %p\n", "hello");  
4     printf("String pointer = %p\n", string);  
5     string[4] = '\\0';  
6     printf("Go to %s\n", string);  
7     return 0;  
8 }
```

- Question #1.** What are the printouts from Line 3 & 4?

"hello" = 0x8048520
String pointer = 0x8048520

- Question #2.** What is the printout from Line 6?

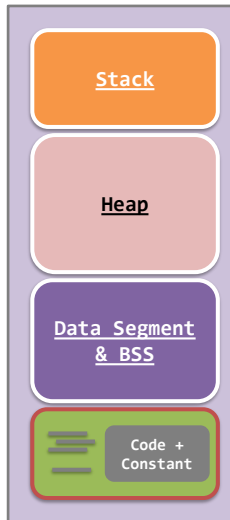
Segmentation fault



Program code & constants

```
1 int main(void) {  
2     char *string = "hello";  
3     printf("\nhello\" = %p\n", "hello");  
4     printf("String pointer = %p\n", string);  
5     string[4] = '\\0';  
6     printf("Go to %s\n", string);  
7     return 0;  
8 }
```

- Constants are stored in code segment.
 - The memory for constants is decided by the program code
 - Accessing of constants are done using addresses (or pointers).
- Codes and constants are both **read-only**.



User-space memory management

- Address space;
- Code & constants;
- **Data segment;**
- Stack;
- Heap;
- Segmentation fault;



Data Segment & BSS – properties

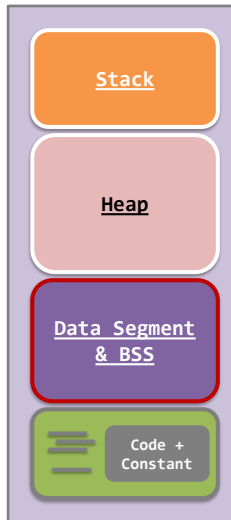
```
int global_int = 10;
int main(void) {
    int local_int = 10;
    static int static_int = 10;
    printf("local_int addr = %p\n", &local_int );
    printf("static_int addr = %p\n", &static_int );
    printf("global_int addr = %p\n", &global_int );
    return 0;
}
```

```
$ ./global_vs_static
local_int addr = 0xbf8bb8ac
static_int addr = 0x804a018
global_int addr = 0x804a014
$_
```

They are stored next to each other.

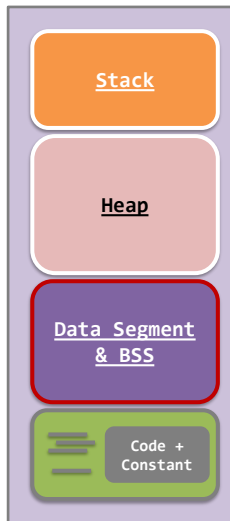
This implies that they are **in the same segment!**

Note: A static variable is treated as the same as a global variable!



Data Segment & BSS – properties

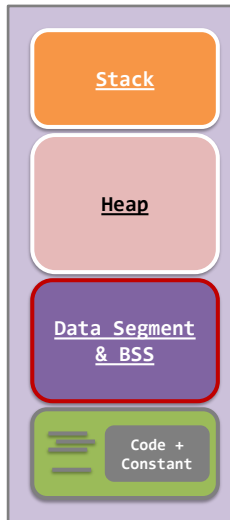
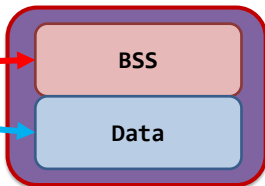
- Data
 - Containing **initialized** global and static variables.
- BSS (**B**lock **S**tarted by **S**ymbol)
 - Containing **uninitialized** global and static variables.



Data Segment & BSS – locations

```
1 int global_bss;  
2 int global_data = 10;  
3 int main(void) {  
4     static int static_bss;  
5     static int static_data = 10;  
6     printf("global bss = %p\n", &global_bss );  
7     printf("static bss = %p\n", &static_bss );  
8     printf("global data = %p\n", &global_data );  
9     printf("static data = %p\n", &static_data );  
10 }
```

```
$ ./data_vs_bss  
global bss = 0x804a028  
static bss = 0x804a024  
global data = 0x804a014  
static data = 0x804a018  
$_
```



Data Segment & BSS – sizes

```
char a[1000000] = {10};
```

```
int main(void) {  
    return 0;  
}
```

Program: data_large.c

```
char a[100] = {10};
```

```
int main(void) {  
    return 0;  
}
```

Program: data_small.c

No optimization.

```
$ gcc -O0 -o data_large data_large.c
```

```
$ gcc -O0 -o data_small data_small.c
```

```
$ ls -l data_small data_large
```

Guess! Which one is large?

What is the difference between data and BSS?

Data Segment & BSS – sizes

```
char a[1000000] = {10};  
  
int main(void) {  
    return 0;  
}
```

Program: data_large.c

```
char a[100] = {10};  
  
int main(void) {  
    return 0;  
}
```

Program: data_small.c

```
$ gcc -O0 -o data_large data_large.c  
$ gcc -O0 -o data_small data_small.c  
  
$ ls -l data_small data_large  
-rwxr-xr-x ... 1004816 ... data_large  
-rwxr-xr-x ... 4916 ... data_small  
$_
```

Wow!

The data segment has the required space already allocated.

Data Segment & BSS – sizes

```
char a[1000000];
```

```
int main(void) {  
    return 0;  
}
```

Program: bss_large.c

```
char a[100];
```

```
int main(void) {  
    return 0;  
}
```

Program: bss_small.c

```
$ gcc -O0 -o bss_large bss_large.c
```

```
$ gcc -O0 -o bss_small bss_small.c
```

```
$ ls -l bss_small bss_large
```

```
-rwxr-xr-x ... 4775... bss_large
```

```
-rwxr-xr-x ... 4775... bss_small
```

```
$ _
```

Same size!

To the program, BSS is just a bunch of symbols.
The space is not yet allocated.

The space will be allocated to the process once it starts executing.

This is why BSS is called “*Block Started by Symbol*”.

Data Segment & BSS – limits

How large is the data segment?

```
$ ulimit -a
core file size      (blocks, -c) 0
data seg size       (kbytes, -d) unlimited
.....
$ _
```

In Linux, “**ulimit**” is a built-in command in “**/bin/bash**”.

It sets or gets the system limitations in the current shell.

Does the “unlimited” mean that you can define a global array with large enough size?

Data Segment & BSS – limits

```
#define ONE_MEG (1024 * 1024)

char a[1024 * ONE_MEG];

int main(void) {
    memset(a, 0, sizeof(a));
    printf("1GB OK\n");
}
```

```
#define ONE_MEG (1024 * 1024)

char a[2048 * ONE_MEG];

int main(void) {
    memset(a, 0, sizeof(a));
    printf("2GB OK\n");
}
```

```
$ gcc -Wall -O0 global_2gb.c -o global_2gb
global_2gb.c:6: warning: integer overflow in expression
global_2gb.c:6: error: size of array 'a' is negative
$ _
```

The size of an array is a 32-bit **signed integer**, no matter 32-bit or 64-bit systems. Therefore...

Data Segment & BSS – limits

```
#define ONE_MEG (1024 * 1024)

char a[1024 * ONE_MEG];
char b[1024 * ONE_MEG];
char c[1024 * ONE_MEG];
char d[1024 * ONE_MEG];

int main(void) {
    memset(a, 0, sizeof(a));
    printf("1GB OK\n");
    memset(b, 0, sizeof(b));
    printf("2GB OK\n");
    memset(c, 0, sizeof(c));
    printf("3GB OK\n");
    memset(d, 0, sizeof(d));
    printf("4GB OK\n");
}
```

Program: global_4gb.c

Segmentation fault
why?

On a **32-bit** Linux system, the user-space **addressing space is around 3GB**.

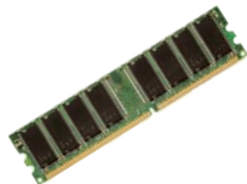
The kernel reserves 1GB addressing space.

Data Segment & BSS – summary

- Remember, “**global variable == static variables**”.
 - Only the **compiler cares** about the difference!
- Everything in a computer has a limit!
 - Different systems have different limits: 32-bit VS 64-bit.
 - Your job is to adapt to such limits.
 - On a **32-bit** Linux system, the user-space **addressing space is around 3GB**.

User-space memory management

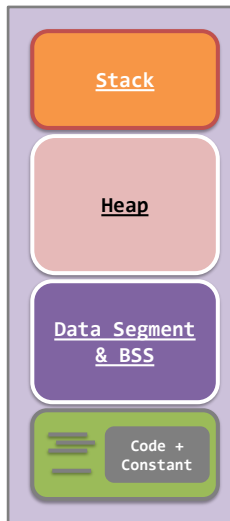
- Address space;
- Code & constants;
- Data segment;
- **Stack;**
- Heap;
- Segmentation fault;



Stack – properties

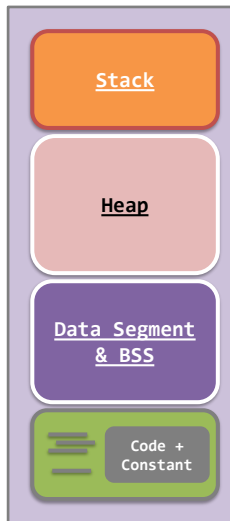
- The stack contains:
 - all the local variables,
 - all function parameters,
 - program arguments, and
 - environment variables.

How are the data stored and what is the size limit?

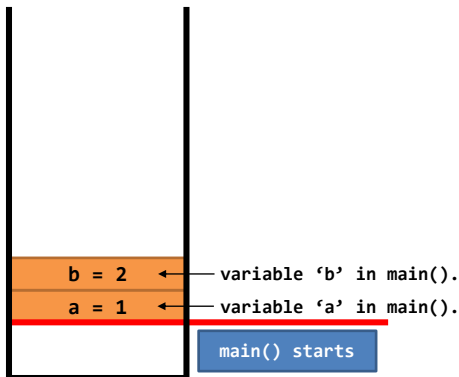


Stack – properties

- Stack: FILO
- When a function is called, the local variables are allocated in the stack.
- When a function returns, the local variables are deallocated from the stack.



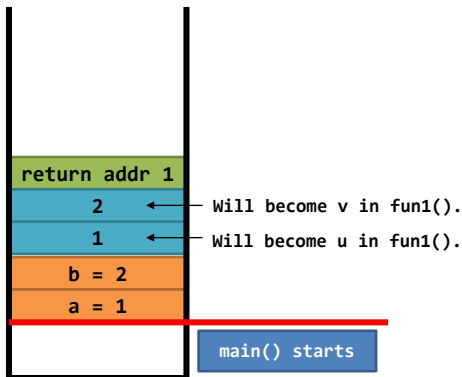
Stack – push & pop mechanisms



```
int fun2(int x, int y) {  
    int c = 10;  
    return (x + y + c);  
}  
  
int fun1(int u, int v) {  
    return fun2(v, u);  
}  
  
int main(void) {  
    int a = 1, b = 2;  
    b = fun1(a, b);  
    return 0;  
}
```

Stack – push & pop mechanisms

Calling function “**fun1()**” starts.
It is the beginning of the call, and the CPU has not switched to **fun1()** yet.



```
int fun2(int x, int y) {  
    int c = 10;  
    return (x + y + c);  
}
```

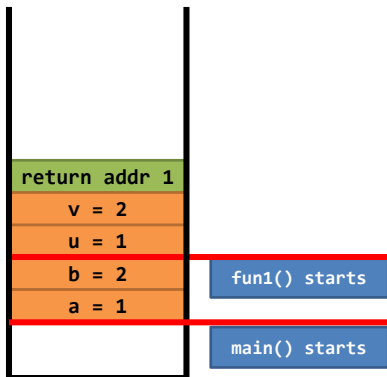
```
int fun1(int u, int v) {  
    return fun2(v, u);  
}
```

```
int main(void) {  
    int a = 1, b = 2;  
    b = fun1(a, b);  
    return 0;  
}
```

“return addr 1”
is approx. here.

Stack – push & pop mechanisms

Calling function “**fun1()**” takes place. The CPU has switched to **fun1()**.



```
int fun2(int x, int y) {  
    int c = 10;  
    return (x + y + c);  
}
```

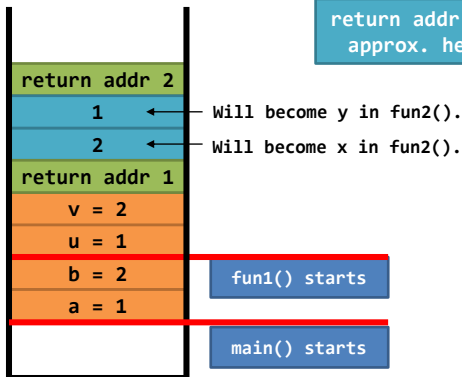
```
int fun1(int u, int v) {  
    return fun2(v, u);  
}
```

```
int main(void) {  
    int a = 1, b = 2;  
    b = fun1(a, b);  
    return 0;  
}
```


Stack – push & pop mechanisms

Calling function “**fun2()**” starts.

It is the beginning of the call, and the CPU has not switched to **fun2()** yet.



return addr 2 is approx. here.

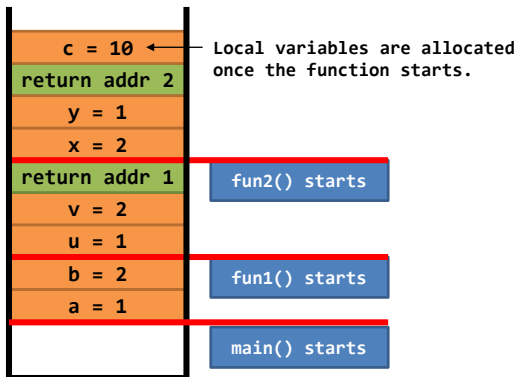
```
int fun2(int x, int y) {  
    int c = 10;  
    return (x + y + c);  
}
```

```
int fun1(int u, int v) {  
    return fun2(v, u);  
}
```

```
int main(void) {  
    int a = 1, b = 2;  
    b = fun1(a, b);  
    return 0;  
}
```

Stack – push & pop mechanisms

Calling function “**fun2()**” takes place. The CPU has switched to **fun2()**.



```
int fun2(int x, int y) {  
    int c = 10;  
    return (x + y + c);  
}
```

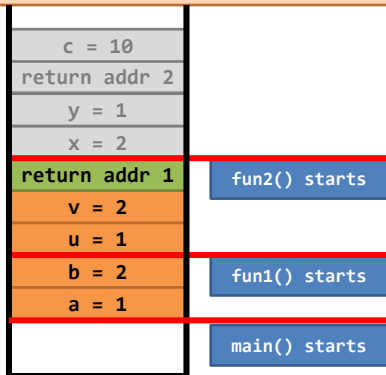
```
int fun1(int u, int v) {  
    return fun2(v, u);  
}
```

```
int main(void) {  
    int a = 1, b = 2;  
    b = fun1(a, b);  
    return 0;  
}
```

Stack – push & pop mechanisms

“Return” takes place.

- (1) Return value is written to the EAX register.
- (2) Stack **shrinks**.
- (3) CPU jumps back to **fun1()**.



```
int fun2(int x, int y) {  
    int c = 10;  
    return (x + y + c);  
}
```

```
int fun1(int u, int v) {  
    return fun2(v, u);  
}
```

```
int main(void) {  
    int a = 1, b = 2;  
    b = fun1(a, b);  
    return 0;  
}
```



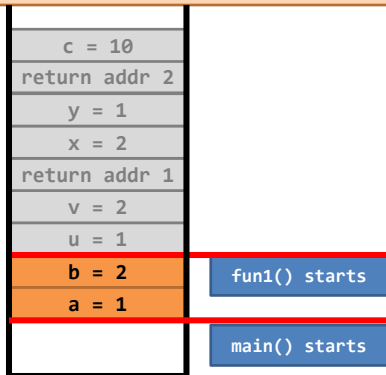
EAX: 13



Stack – push & pop mechanisms

“Return” takes place.

- (1) Return value is written to the EAX register.
- (2) Stack **shrinks**.
- (3) CPU jumps back to `main()`.



```
int fun2(int x, int y) {  
    int c = 10;  
    return (x + y + c);  
}
```

```
int fun1(int u, int v) {  
    return fun2(v, u);  
}
```

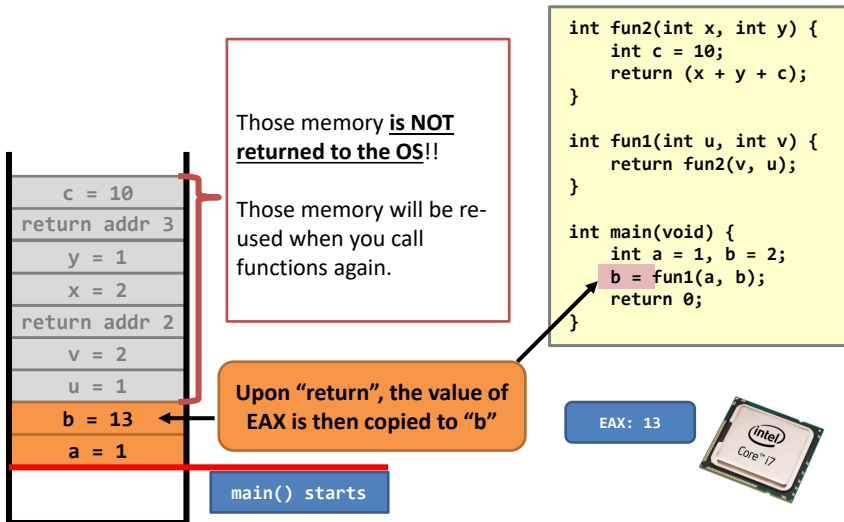
```
int main(void) {  
    int a = 1, b = 2;  
    b = fun1(a, b);  
    return 0;  
}
```



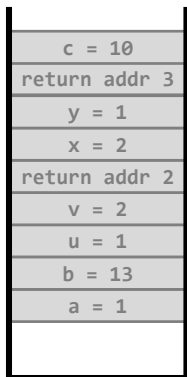
EAX: 13



Stack – push & pop mechanisms



Stack – push & pop mechanisms



Eventually, the main function reaches “**return 0**”.

This takes the CPU pointing to the C library.

Inside the C library, we will eventually reach the system call **exit()**.

```
int fun2(int x, int y) {  
    int c = 10;  
    return (x + y + c);  
}
```

```
int fun1(int u, int v) {  
    return fun2(v, u);  
}
```

```
int main(void) {  
    int a = 1, b = 2;  
    b = fun1(a, b);  
    return 0;  
}
```

EAX: 0



Stack – limits

```
$ ulimit -a
core file size      (blocks, -c) 0
data seg size       (kbytes, -d) unlimited
.....
stack size          (kbytes, -s) 8192 ←
.....
$ _
```

So, the limit is:
 $8192 \times 1024 = 8\text{MB}$.

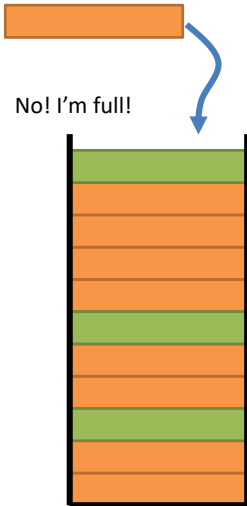
Can you define a local array larger than the limit?

**Segmentation
fault**

```
$ ulimit -a
core file size      (blocks, -c) 0
data seg size       (kbytes, -d) unlimited
.....
stack size          (kbytes, -s) 8192
.....
$ ulimit -s 81920 ←
```

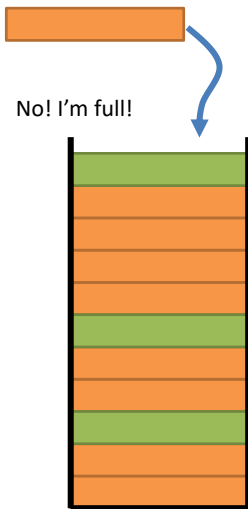
Now, the limit is:
 $81920 \times 1024 = 80\text{MB}$.

Stack – summary



- What if it is a chain of endless recursive function calls?
- What will happen?
 - **Exception caught by the CPU!**
 - **Stack overflow exception!**
 - **Program terminated!**

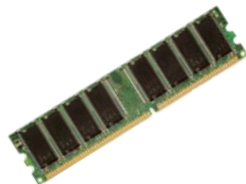
Stack – summary



- *“I really need to play with recursions.”* Any workaround?
 - Minimize the number of arguments
 - Minimize the number of local variables
 - Minimize the number of calls
 - Use global variables
- Note: A function can ask the CPU **to read and to write anywhere in the stack**, not just the “zone” belonging to the running function!
 - Isn't it horrible (profitable and fun)?

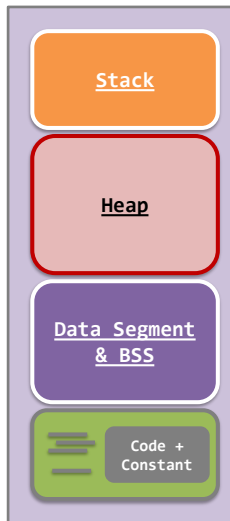
User-space memory management

- Address space;
- Code & constants;
- Data segment;
- Stack;
- **Heap;**
- Segmentation fault;



Dynamically allocated memory – properties

- Its name tells you its nature:
 - The dynamically allocated memory is called the **heap**.
 - Don't mix it up with the binary heap;
 - It has nothing to do with the binary heap.
 - **Dynamic**: not defined at compile time.
 - **Allocation**: only when you ask for memory, you would be allocated the memory.

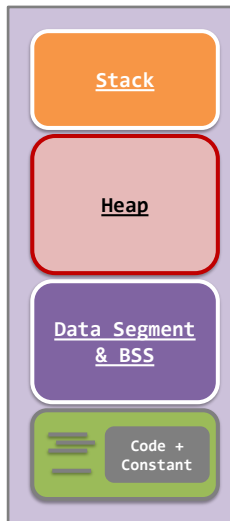


Dynamically allocated memory – properties

- Lecturers of a programming course would tell you the following:

- “***malloc()***” is a function that allocates memory for you.
- “***free()***” is a function that gives up a piece of memory that is produced by previous “***malloc()***” call.

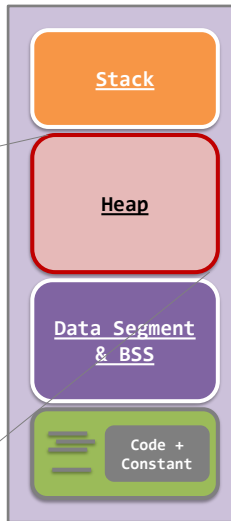
- The lecturer of the OS course is **to define and to defy** what you know about the **malloc()** and **free()** library functions.



malloc()

When a program just starts running, the entire heap space is unallocated, or empty.

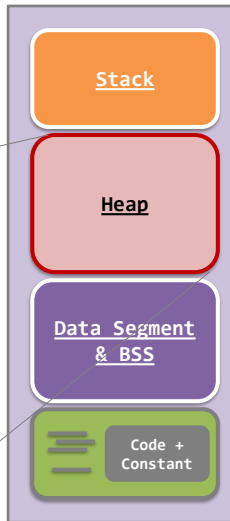
An empty heap.



malloc()

When “**malloc()**” is called, the “**brk()**” system call is invoked accordingly.

“**brk()**” allocates the space required by “**malloc()**”. But, it doesn't care how “**malloc()**” uses the space.

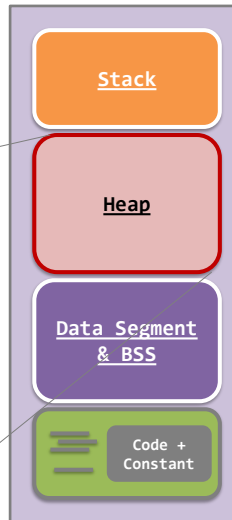
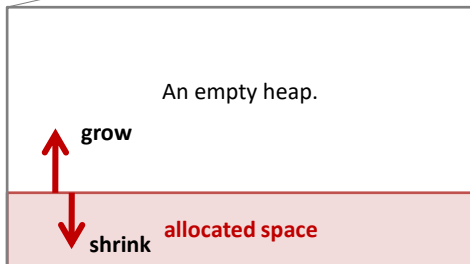


malloc()

The allocated space growing or shrinking depends on the further actions of the process. That means the “**brk()**” system call can **grow or shrink** the allocated area.

In **malloc()**, the library call just invoke **brk()** for growing the heap space.

The **free()** call may shrink the heap space.



malloc()

```
int main(void) {  
    char *ptr1, *ptr2;  
    ptr1 = (char *)malloc(16);  
    ptr2 = (char *)malloc(16);  
  
    printf("Distance between ptr1 and  
           ptr2 - ptr1);  
    return 0;  
}
```

The return value of **malloc()** is of type "**void ***", which means it is just a memory address only, and can be of any data types.

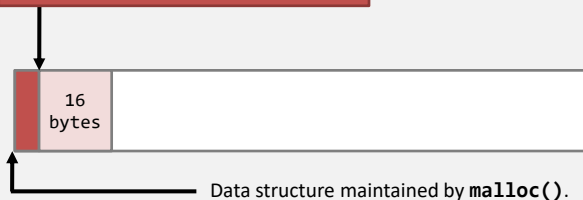
Such a memory address is the starting address of a piece of memory of 16 bytes ("16" is the request of **malloc()** call).

Heap

malloc()

```
int main(void) {  
    char *ptr1, *ptr2;  
    ptr1 = (char *)malloc(16);  
    ptr2 = (char *)malloc(16);  
  
    printf("Distance between ptr1 and ptr2: %d bytes\n",  
          ptr2 - ptr1);  
    return 0;  
}
```

Address returned by 1st malloc() call.

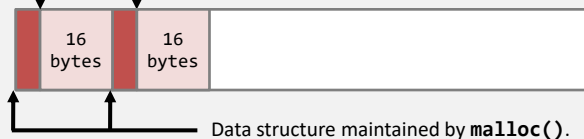


malloc()

```
int main(void) {  
    char *ptr1, *ptr2;  
    ptr1 = (char *)malloc(16);  
    ptr2 = (char *)malloc(16);  
  
    printf("Distance between ptr1 and ptr2: %d bytes\n",  
          ptr2 - ptr1);  
    return 0;  
}
```

Address returned by 1st malloc() call.

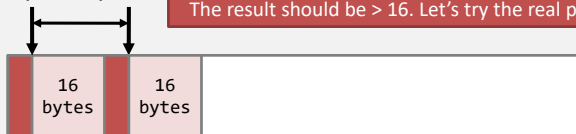
Address returned by 2nd malloc() call.



malloc()

```
int main(void) {  
    char *ptr1, *ptr2;  
    ptr1 = malloc(16);  
    ptr2 = malloc(16);  
  
    printf("Distance between ptr1 and ptr2: %d bytes\n",  
           ptr2 - ptr1);  
    return 0;  
}
```

ptr2 - ptr1

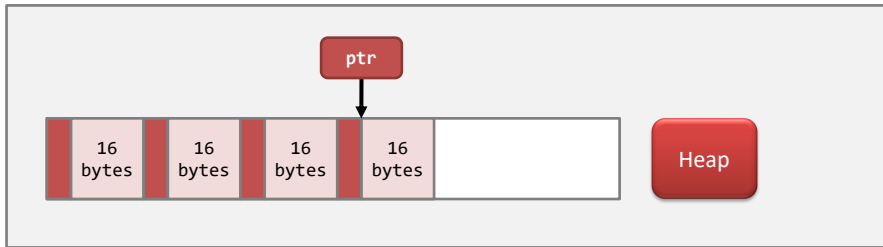


The result should be > 16. Let's try the real program!

Heap

free()

- “**free()**” seems to be the opposite to “**malloc()**”:
 - It de-allocates any allocated memory.
 - When a program calls “**free(ptr)**”, then the address “**ptr**” must be the start of a piece of memory obtained by a previous “**malloc()**” call.



free() – case #1

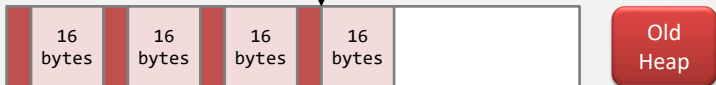
- Case #1: de-allocating the last block.

This is accomplished by calling **brk()** system call. This heap has become smaller.



ptr

The last block is not needed.



free() – case #2

- Case #2: de-allocating an intermediate block.

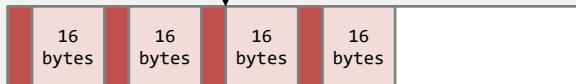
Calling `brk()` system call without using your brain is not acceptable!



New
Heap

ptr

We don't want an intermediate block.



Old
Heap

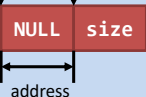
free() – case #2

- Case #2: de-allocating an intermediate block.

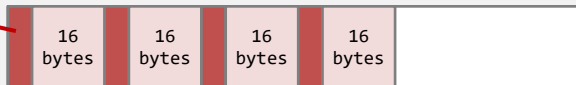
Here comes the role of the data structure created by `malloc()`!

This pointer is used for creating a linked list of de-allocated block.

This size record the size of de-allocated block.



32-bit system: $4+4 = 8$ bytes
64-bit system: $8+8 = 16$ bytes



Heap

free() – case #2

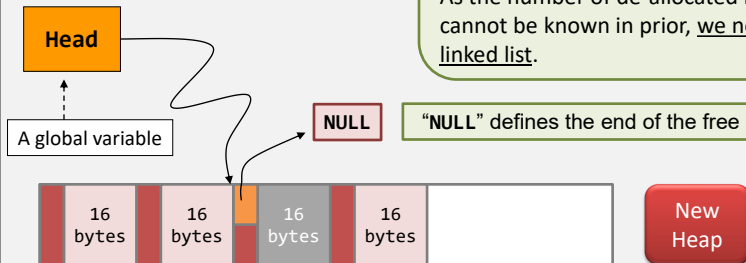
- Case #2: de-allocating an intermediate block.

The “**Head**” variable is a pointer acting as the **start of the list of the free blocks**.

We have to keep the de-allocated blocks because they cannot be returned to the system.

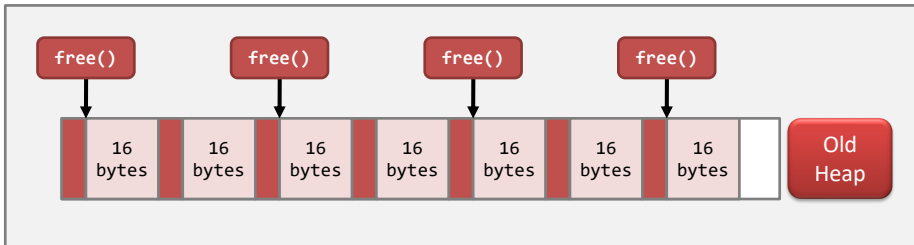
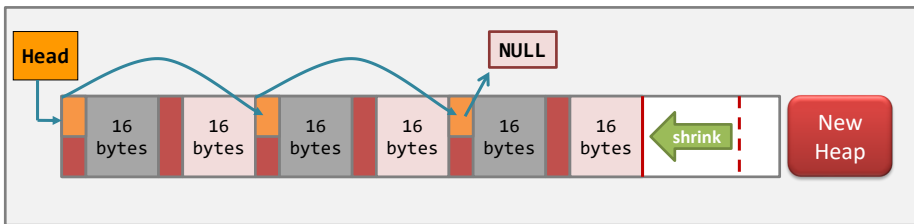
As the number of de-allocated blocks cannot be known in prior, we need a linked list.

“**NULL**” defines the end of the free list.



free() – case #2

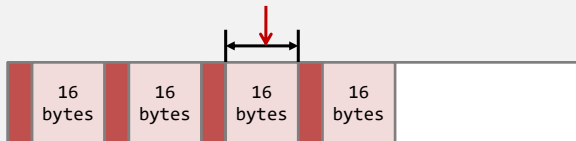
- Case #2: another example.



free() – cautions

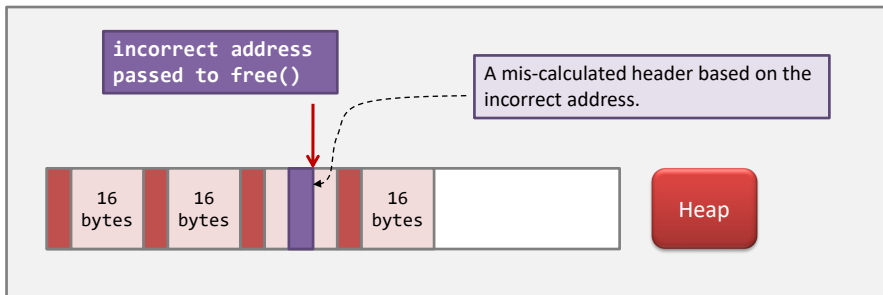
- The calling program is **assumed** to be carefully written.
 - After **malloc()** has been invoked, the program should read and write inside the requested area only.
 - Now, you know why you'd **have troubles** when you write data outside the allocated space.

You can only play within this zone. Please behave!
Note: be careful of the **consequences** of misbehaves...



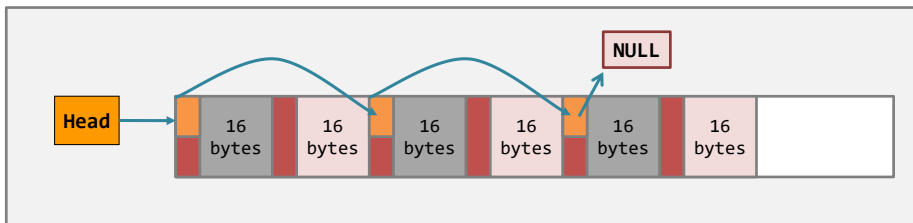
free() – cautions

- The calling program is **assumed** to be carefully written.
 - When **free()** is called, the program should provide **free()** with the correct address...
 - i.e., the address previously returned by a **malloc()** call.



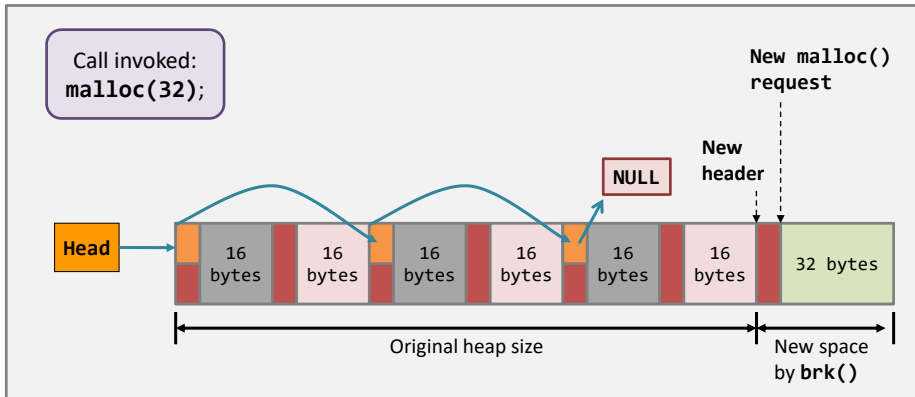
When `malloc()` meets free blocks...

- Problem: whether to use the free blocks or not?
 - *Is there any free block that is large enough to satisfy the need of the `malloc()` call?*



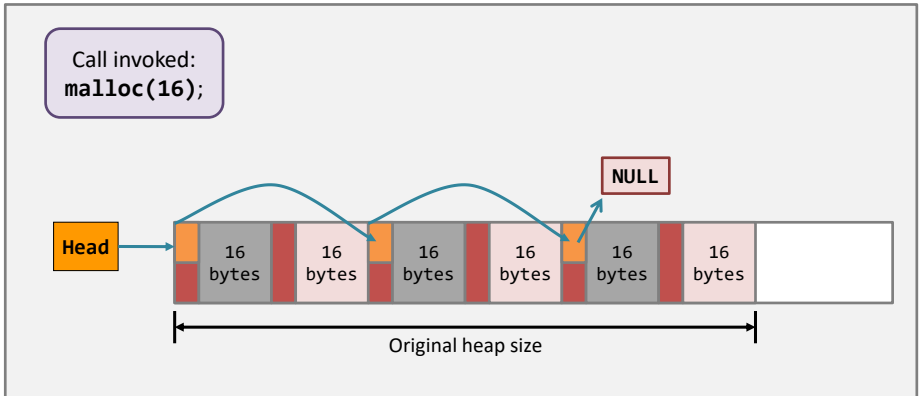
When `malloc()` meets free blocks...case #1

- Case #1: if there is **no suitable free block**...
 - then, the `malloc()` function should call `brk()` system call...in order to claim more heap space.



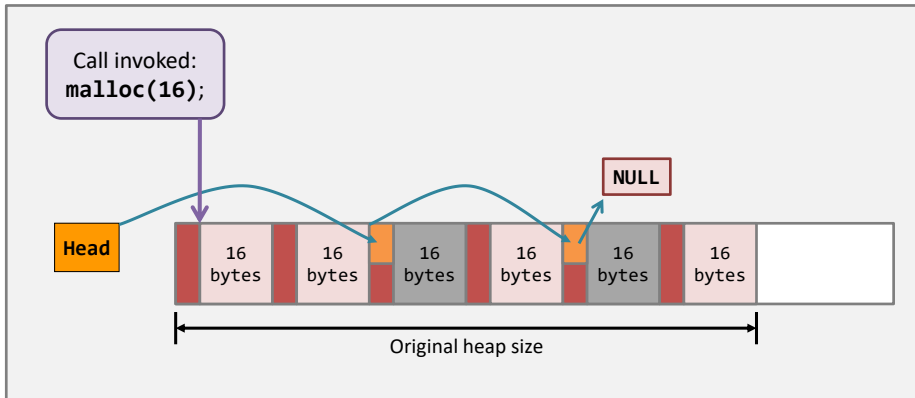
When `malloc()` meets free blocks...case #2

- Case #2: if there is a suitable free block



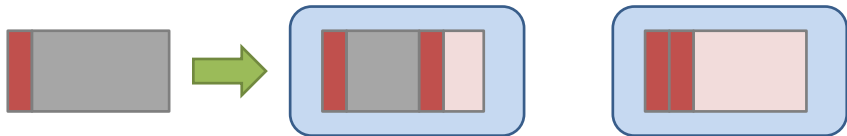
When `malloc()` meets free blocks...case #2

- Case #2: if there is a suitable free block
 - the `malloc()` function should reuse that free block.



When `malloc()` meets free blocks...

- There can be other cases:
 - A `malloc()` request that takes a partial block;
 - A `malloc()` request that takes a partial block, but leaving no space in the previously free block.



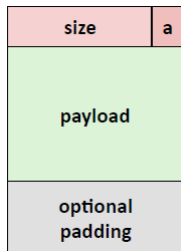
- We will skip those subtle cases...
 - It boils to implementation only...
 - You already have the **big picture** about `malloc()` and `free()`.

When `malloc()` meets free blocks...

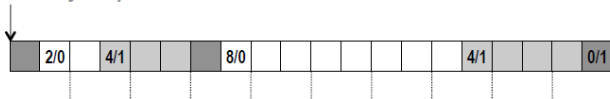
- Now, let us look at some implementations...

Implicit free list

- Needs two information for each block
 - size & is_allocated



Start of heap



free



allocated

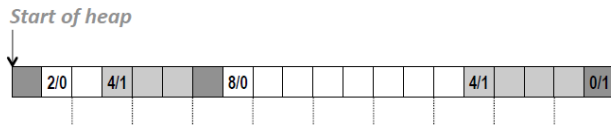


Allocated & unused

How about memory allocation and free?

Implicit free list

- Allocation: May need linear time search



First fit: allocate the first hole that is big enough (fast)

Next fit: similar to first fit, but start where previous search finishes

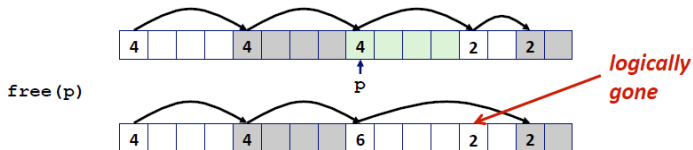
Best fit: allocate the smallest hole that is big enough (helps fragmentation, larger search time)

Worst fit: allocate the largest hole

– Allocate the whole block or splitting

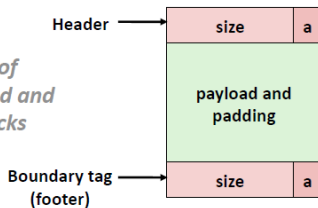
Implicit free list

- Free: Coalescing
 - Coalescing with next block: easy



- How about coalescing with previous block?
 - [Knuth 73] Add a boundary tag in the footer

*Format of
allocated and
free blocks*



Implicit free list

- Constant time coalescing w/ boundary tag (4 cases)

m1	1
m1	1
n	1
n	1
m2	1
m2	1



m1	1
m1	1
n	0
n	0
m2	1
m2	1

m1	1
m1	1
n	1
n	1
m2	0
m2	0



m1	1
m1	1
n+m2	0
n+m2	0

m1	0
m1	0
n	1
n	1
m2	1
m2	1



n+m1	0
n+m1	0
m2	1
m2	1

m1	0
m1	0
n	1
n	1
m2	0
m2	0



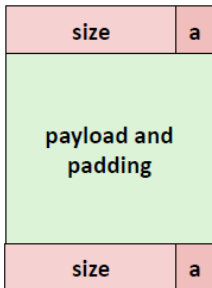
n+m1+m2	0
n+m1+m2	0

Implicit free list: summary

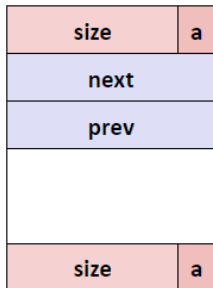
- May not be used in practical malloc() and free() implementations
 - High memory allocation cost
- Some ideas are still useful and important
 - Splitting available blocks
 - Boundary tag

Explicit free list

Allocated (as before)



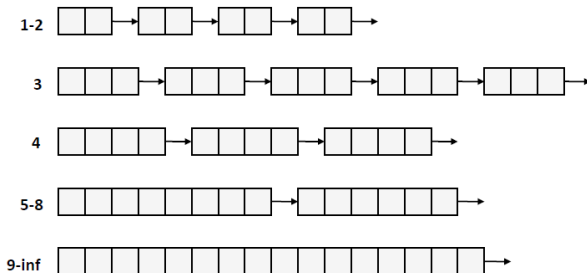
Free



- Track only free blocks (LIFO or address-ordered)
- Block splitting is useful in allocation
- Boundary tag is still useful in coalescing

Segregated free list

- Segregated free list (分离空闲链表)
 - Different free lists for different size classes



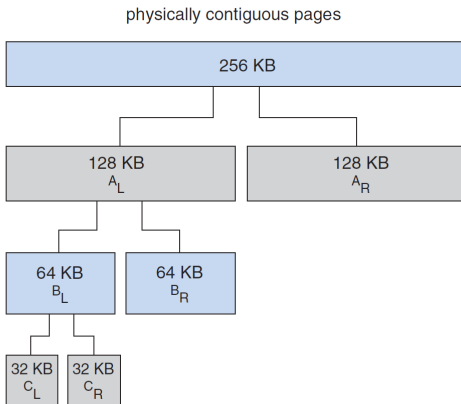
– Allocation

- Search appropriate list (larger size)
- Found and split
- Not found: search next

Approximates best -fit

Segregated free list

- Special example
 - Buddy system (power-of-two block size)



Issues raised by malloc() and free()

- The kernel knows how much memory should be given to the heap.
 - When you call **brk()**, the kernel tries to find the memory for you.
- Then...one natural question...
 - Is it possible to **run out of memory (OOM)**?

Out of memory?

- Try this!

```
#define ONE_MEG 1024 * 1024

int main(void) {
    void *ptr;
    int counter = 0;

    while(1) {
        ptr = malloc(ONE_MEG);
        if(!ptr)
            break;
        counter++;
        printf("Allocated %d MB\n", counter);
    }

    return 0;
}
```

Is it safe to run this program on a 32-bit machine?

What is the output?

```
Allocated 3052 MB
Allocated 3053 MB
Allocated 3054 MB
Allocated 3055 MB
linux2:/uac/rsr/ykli>
```

Out of memory?

- On 32-bit Linux, why does the OOM generator stop at around 3055MB?
- Still remember what we said when we are talking about data segment?
 - Every 32-bit Linux system has an **addressable memory space** of 4G-1 bytes.
 - The kernel reserves 1GB addressing space.

Out of memory?

- Try this! Yet another OOM Generator!

```
#define ONE_MEG 1024 * 1024
char global[1024 * ONE_MEG];
int main(void) {
    void *ptr;
    int counter = 0;
    char local[8000 * 1024];
    while(1) {
        ptr = malloc(ONE_MEG);
        if(!ptr)
            break;
        counter++;
        printf("Allocated %d MB\n", counter);
    }

    return 0;
}
```

Yet, what is the output?

```
Allocated 3044 MB
Allocated 3045 MB
Allocated 3046 MB
Allocated 3047 MB
linux2:/uac/rshr/ykli>
```

Real OOM!

Explanation is in Part 2.

```
#define ONE_MEG 1024 * 1024

int main(void) {
    void *ptr;
    int counter = 0;

    while(1) {
        ptr = malloc(ONE_MEG);
        if(!ptr)
            break;
        memset(ptr, 0, ONE_MEG);
        counter++;
        printf("Allocated %d MB\n", counter);
    }

    return 0;
}
```

Warning #1. Don't run this program on any department's machines.

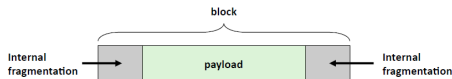
Warning #2. Don't run this program when you have important tasks running at the same time.

Lazy allocation

That is why previous programs run very fast.

Other Issues

- External fragmentation
 - The heap memory looks like a map with many holes
 - It is the source of inefficiency because of the **unavoidable search for suitable space**
 - The memory wasted because **external fragmentation is inevitable**
- Internal fragmentation
 - Payload is smaller than allocated block size
 - Padding for alignment
 - Placement policy
 - Allocate a big block for small request



User-space memory management

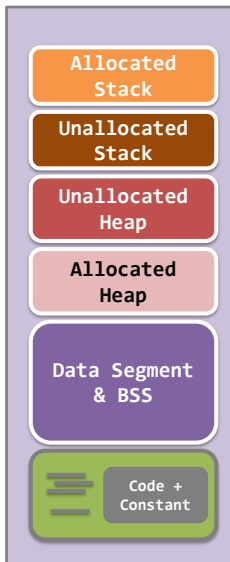
- Address space;
- Code & constants;
- Data segment;
- Stack;
- Heap;
- **Segmentation fault;**



What is segmentation fault?

- Someone must have told you:
 - When you are accessing a piece of memory that is not allowed to be accessed, then the OS returns you an error called – segmentation fault.
- As a matter of fact, how many ways are there to generate a segmentation fault?

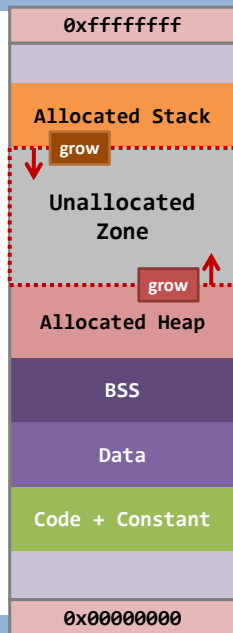
What is segmentation fault?



From illustration to reality...

Forget about the illustration, the memory in a process is separated into **segments**.

So, when you visit a segment in an illegal way, then...**segmentation fault**.



How to “segmentation fault”?

Read	0xffffffff	Write
YES	Unusable	YES
NO	Allocated Stack	NO
YES	Unallocated Zone	YES
NO	Allocated Heap	NO
NO	BSS	NO
NO	Data	NO
NO	Code + Constant	YES
YES	Unusable	YES
	0x00000000	

How to “segmentation fault”?

Read	0xffffffff	Write
YES	Unusable	YES
NO	Allocated Stack	NO
YES	Unallocated Zone	YES
NO	Allocated Heap	NO
NO	BSS	NO
NO	Data	NO
NO	Code + Constant	YES
YES	Unusable	YES
	0x00000000	

Now, we can understand:

```
char *ptr = NULL;  
char c = *ptr;
```

will generates

Segmentation fault

NULL = 0x00000000



*ptr is reading

Summary of segmentation fault

- When you have a **so-called address** (maybe it is just a random sequence of 4 bytes), one of the following cases happens:

See if you have luck...

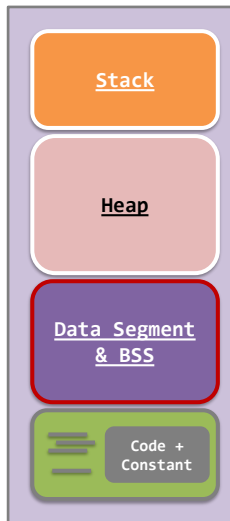
	Read-only segments	Allocated segments	Unused or unallocated segments
Reading	No problem	No problem	Segmentation fault
Writing	Segmentation fault	No problem	Segmentation fault

Summary of segmentation fault

- Now, you know what is a segmentation fault, and the cause is always **carelessness**!
 - Now, you know why “**free()**” sometimes give you segmentation fault...
 - because **you corrupt the list of free blocks**!
 - Now, you know why “**malloc()**”-ing a space that is smaller than required is ok...
 - because **you are overwriting the neighboring blocks**!

Summary of part 1

- Memory of a process is divided into segments (**segmentation**):
 - codes and constants;
 - global and static variables;
 - allocated memory (or heap);
 - local variables (or stack);
- When you access a memory that is not allowed, then the OS returns you **segmentation fault**
- **Every process' segments are independent and distinct.**



Summary of part 1

- The dynamically allocated memory is not as simple as you learned before.
 - Allocating large memory blocks is not efficient; instead, **allocating small memory blocks** can make use of the **holes** in the heap memory efficiently.
 - Keep calling **malloc()** without calling **free()** is dangerous...
 - because there is no garbage collector in C or the OS...
 - **OOM error awaits you!**

End of part 1

Operating Systems

Associate Prof. Yongkun Li

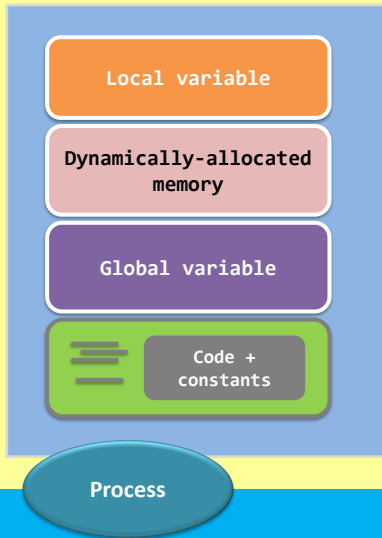
中科大-计算机学院 副教授

<http://staff.ustc.edu.cn/~ykli>

Ch7, part 2

Memory Management from the Kernel's Perspective: Virtual Memory Support

Memory management



How to use the addresses to access the memory device?

How do multiple process share the same physical memory device?

How to support large process?

How does the CPU read what it wants from the memory device?

.....

The kernel and the hardware are doing lots of managements...

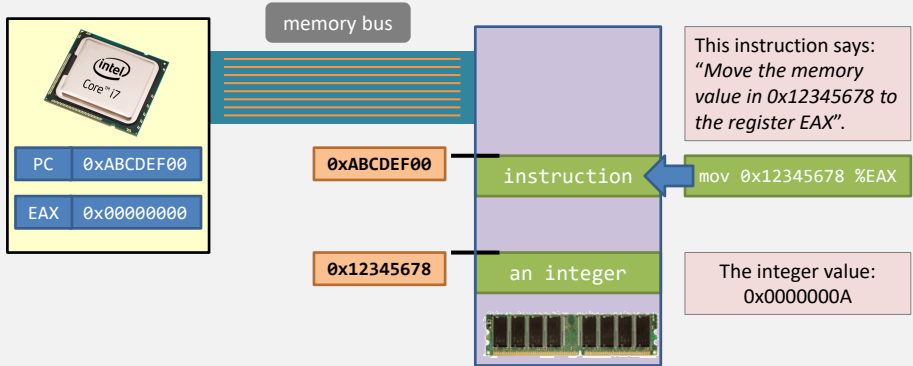
Memory Management

- **Virtual memory;**
- MMU implementation & paging;
- Demand paging;
- Page replacement algorithms;
- Allocation of frames;



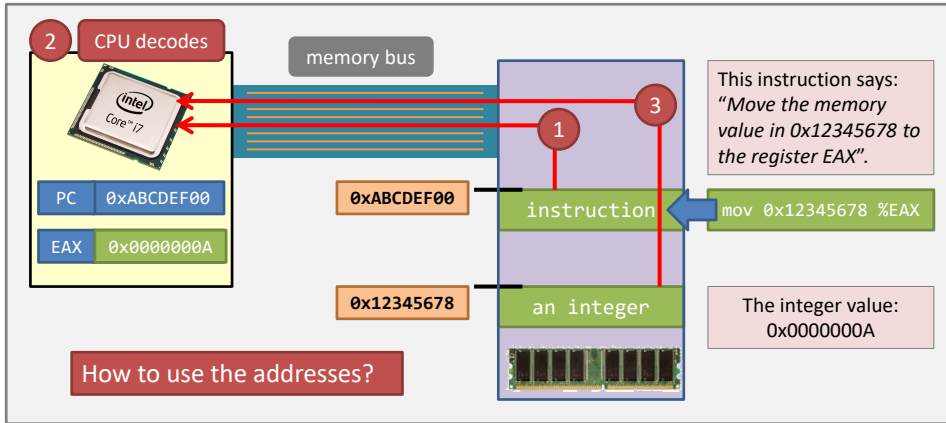
CPU working – illustration that you may know

- Let's review the “fetch-decode-execute” cycle!



CPU working – illustration that you may know

- Let's review the “fetch-decode-execute” cycle!



"You've been living in a dream world, Neo"

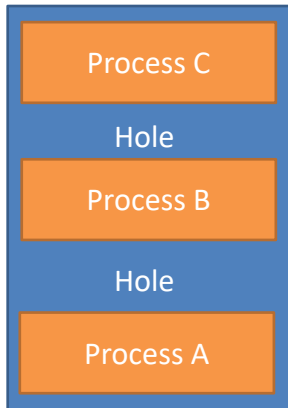
```
int main(void) {  
    int pid;  
    pid = fork();  
    printf("PID %d: %p.\n", getpid(), &pid);  
    if(pid)  
        wait(NULL);  
    return 0;  
}
```

```
$ ./same_addr  
PID 1234: 0xbfe85e0c.  
PID 1235: 0xbfe85e0c.  
$_
```

- Can you guess the result?
 - Two **different processes**, the **same variable name**, carry **different values**
 - Use the **same address**! (What? How COME?!)
- Well, what is the meaning of a memory address?!
 - Logical address: **virtual memory**
 - Address translation needed (logical/virtual->physical)
 - Why we use virtual memory??

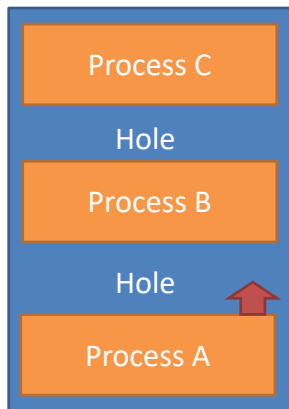
CPU working ... contiguous allocation?

- Each process is contained in a single section of mem



CPU working ... contiguous allocation?

- Problem #1...



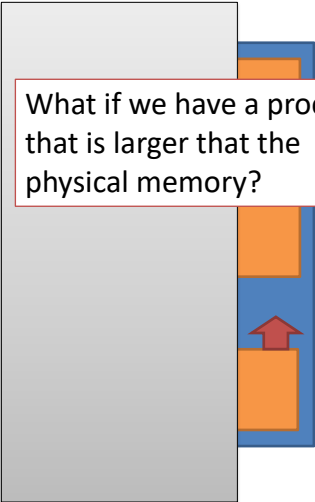
We also know that a process' memory can grow.

So, does a process **always** have a chance to grow to reach its need?

memory growth
e.g., because of **brk()** calls

CPU working ... contiguous allocation?

- Problem #2...



The diagram shows a large grey rectangle representing physical memory. To its right, there is a blue vertical bar representing a process's address space. Inside the blue bar, there are several orange rectangles of different sizes, representing memory segments. A red arrow points upwards from one orange segment to another, indicating a shift or movement of memory segments. A text box is overlaid on the grey rectangle.

What if we have a process that is larger than the physical memory?

We are not talking about the program's size, but the process' size!

What the CPU (or OS) can do is to give up running ...

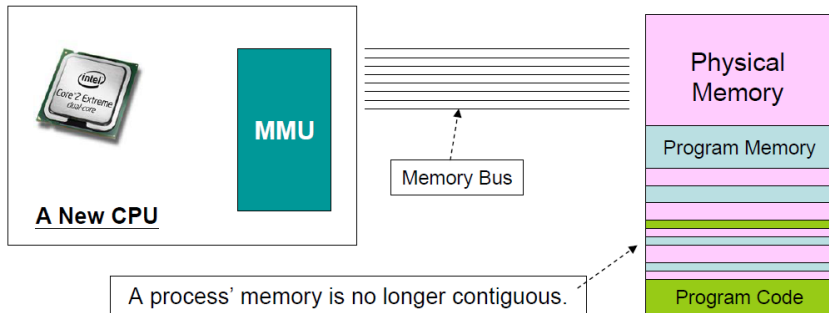
So, we need to have the CPU design that can understand processes so that:

(1) the address space is no longer required to be contiguous.

(2) it allows a process to have a size beyond the physical memory.

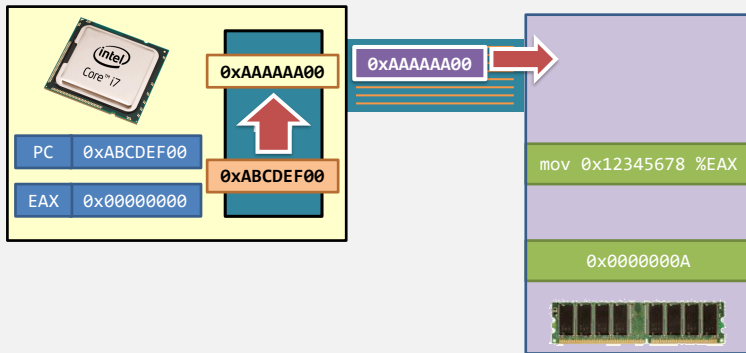
Virtual memory support in modern CPUs

- The new design of the CPU includes a new module: the **memory management unit (MMU)**.
 - MMU is designed to perform address translation.
 - The **MMU** is an **on-CPU device**.



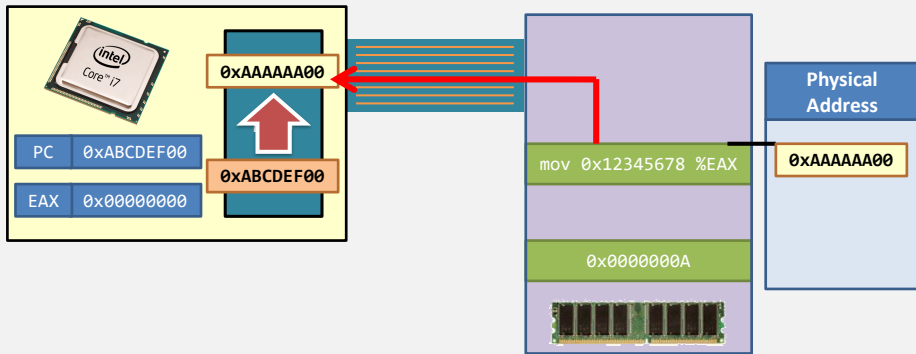
Virtual memory – how does it work?

- Step 1. When CPU wants to fetch an instruction, the virtual address is sent to MMU and is translated into a physical address.



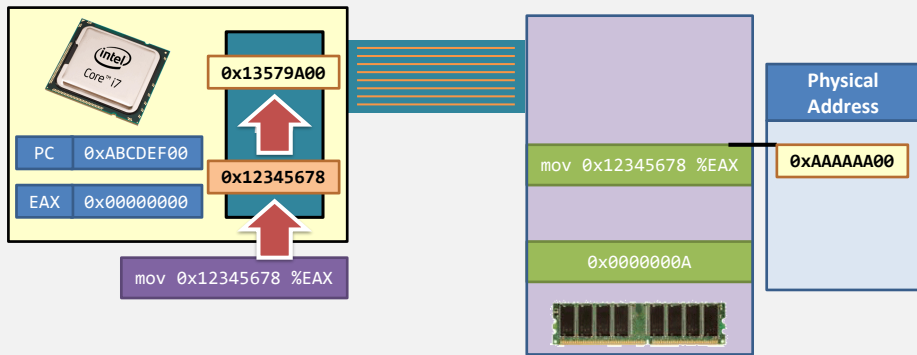
Virtual memory – how does it work?

- Step 2. The memory returns the instruction addressed in physical address.



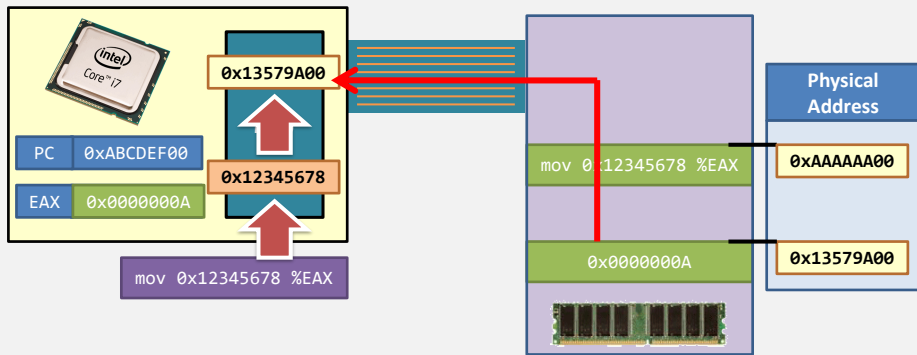
Virtual memory – how does it work?

- Step 3. The CPU decodes the instruction.
 - An instruction **always stores virtual addresses**, but not physical addresses.



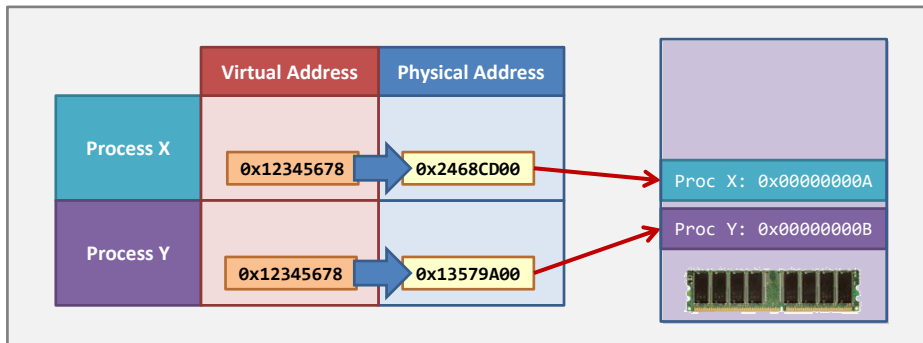
Virtual memory – how does it work?

- Step 4. With the help of the MMU, the target memory is retrieved.



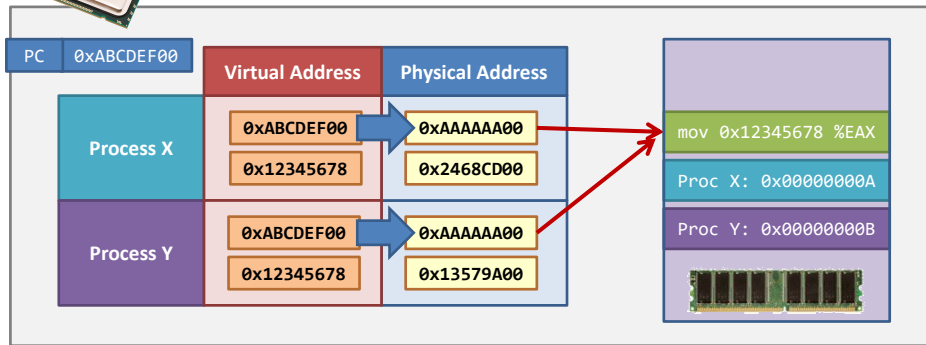
Virtual memory – What is the good?

- **Merit 1.** Different processes use the same virtual addresses, they may be **translated to different physical addresses**.
 - Recall the “**pid**” variable in the example using **fork()**.
 - The address translation helps the CPU to **retrieve data in a non-contiguous layout** (the process address space is contiguous).



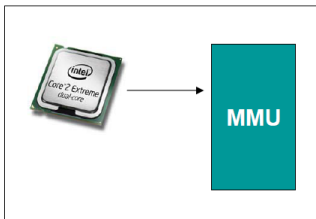
Virtual memory – What is the good?

- **Merit 2. Memory sharing** can be implemented!
 - This is how threads share memory!
 - This is how different processes share codes! (HOW?)

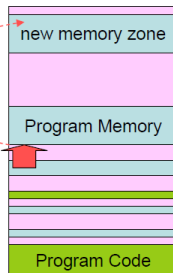


Virtual memory – What is the good?

- **Merit 3.** **Memory growth** can be implemented!
 - When the memory of a process grows, the newly-allocated memory is not required to be contiguous



The growth of the process' memory is not bounded by **the availability of the holes** any more.



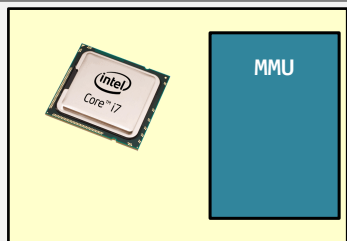
Memory Management

- Virtual memory;
- MMU implementation & paging;
- Demand paging;
- Page replacement algorithms;
- Allocation of frames;



MMU implementation

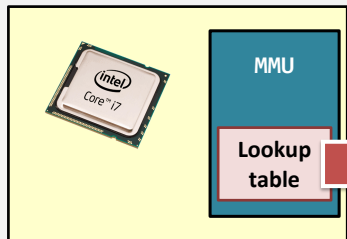
- How to implement the MMU?
 - How to efficiently translate from virtual address to physical address?
 - Translation is needed for every process



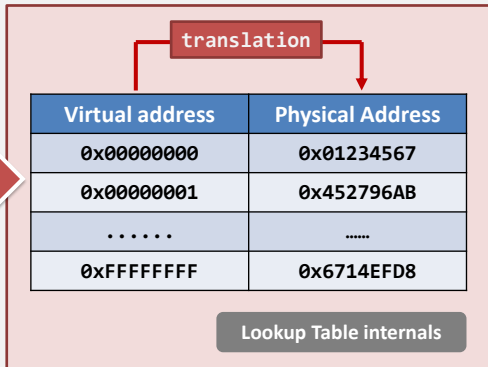
	Virtual Address	Physical Address
Process X	0xABCDEF00	0xAAAAAA00
	0x12345678	0x2468CD00
Process Y	0xABCDEF00	0xAAAAAA00
	0x12345678	0x13579A00

MMU implementation – a translation table

- So, can translation be done by a **lookup table**?
 - Remember, every process needs its own lookup table.
(Do you remember the reason?)



What is the problem with this method?



MMU implementation – a translation table

- Then, how large is the lookup table?

How many addresses are there?

2^{32}

How large is an address?

4 bytes

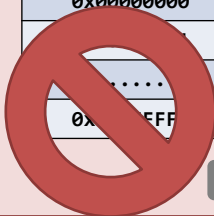
Size of the lookup table =

Number of addresses
x Size of an address

$2^{32} \times 4 \text{ bytes} = 16 \text{ Gbytes}$

Only this column is stored.

Virtual address	Physical Address
0x00000000	0x01234567
.....	0x452796AB
.....
0xFFFFFFFF	0x6714EFD8



Lookup Table internals

MMU implementation – a translation table

- Then, how large is the lookup table?

How many addresses are there?

2^{32}

How large is an address?

4 bytes

Size of the lookup table =

Number of addresses
x Size of an address

$2^{32} \times 4 \text{ bytes} = 16 \text{ Gbytes}$

Can we reduce the table size?

Note. Every address in a CPU is always of 4 bytes.

The only choice is to reduce the number of addresses

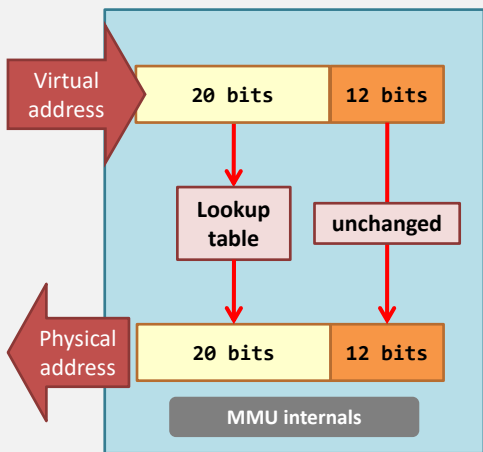
MMU implementation – a partial lookup table

Size of the lookup table =

Number of addresses
x Size of an address

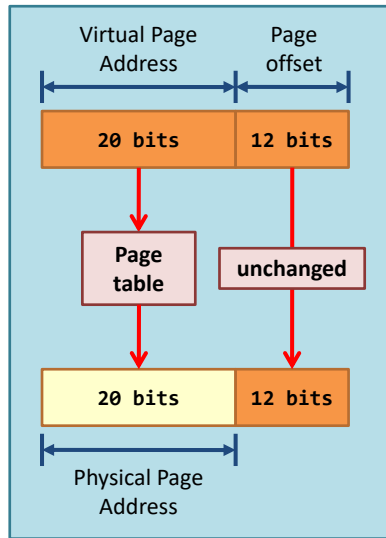
$2^{20} \times 4 \text{ bytes} = 4 \text{ Mbytes}$

Note. Every address in a CPU is always of 4 bytes although you only use 20 bits.

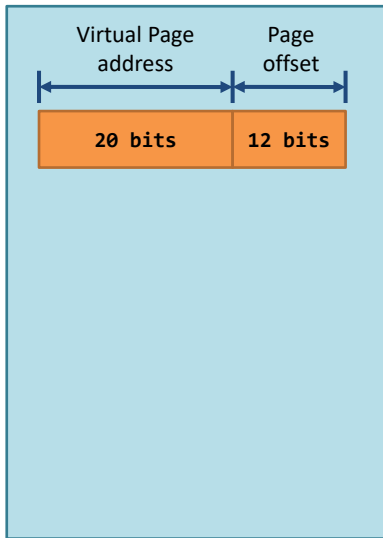
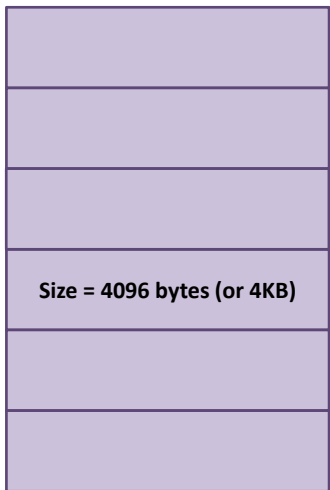


MMU implementation – paging

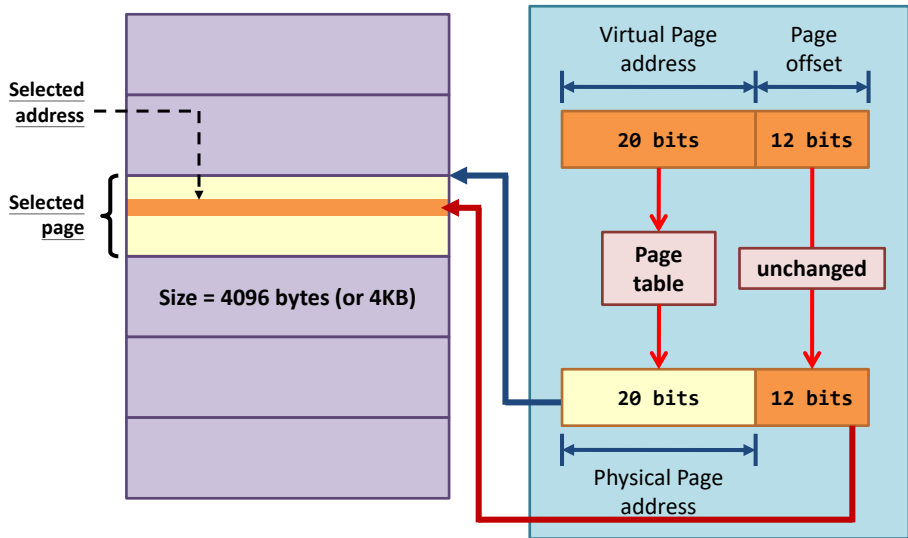
- This technique is called **paging**.
 - This partitions the memory into fixed blocks called **pages**.
 - The lookup table inside the MMU is now called the **page table**.



Paging - properties

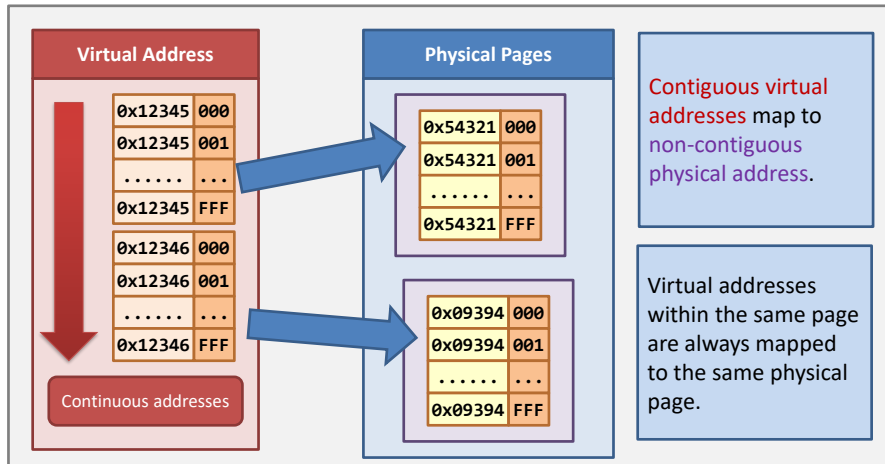


Paging - properties



Paging - properties

- Adjacent virtual pages are not guaranteed to be mapped to adjacent physical pages.



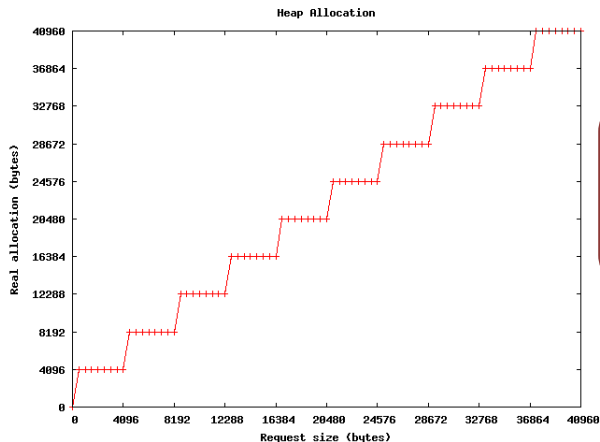
Paging – memory allocation

- How to do memory allocation with paging

```
1 char *prev_ptr = NULL;
2 char *ptr = NULL;
3
4 void handler(int sig) {
5     printf("Page size = %d bytes\n",
6           (int) (ptr - prev_ptr));
7     exit(0);
8 }
9 int main(int argc, char **argv) {
10     char c;
11     signal(SIGSEGV, handler);
12     prev_ptr = ptr = sbrk(0); // find the heap's start.
13     sbrk(1);                  // increase heap by 1 byte?
14     while(1)
15         c = *(++ptr);
16 }
```

Paging – memory allocation

- A page is the basic unit of memory allocation.



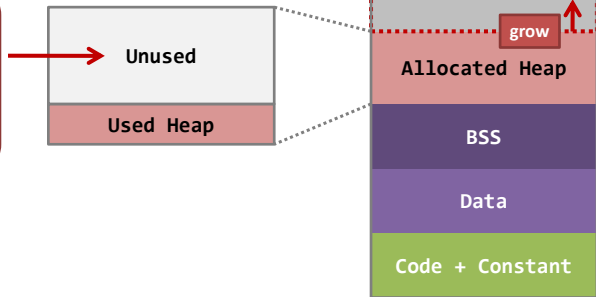
The allocation is in a page-by-page manner.

The same case for the growth of the stack.

Paging – memory allocation

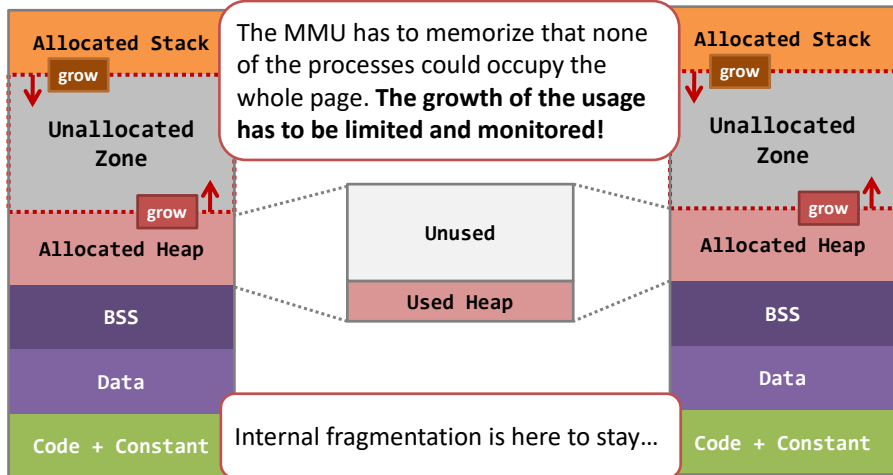
- Problem???
- The minimum allocation unit is 4,096 bytes.
- But, the process cannot use that much.
- So, the rest of the page is unused.

Internal fragmentation
means space is avoidably
wasted when allocation is
done in a page-by-page
manner.

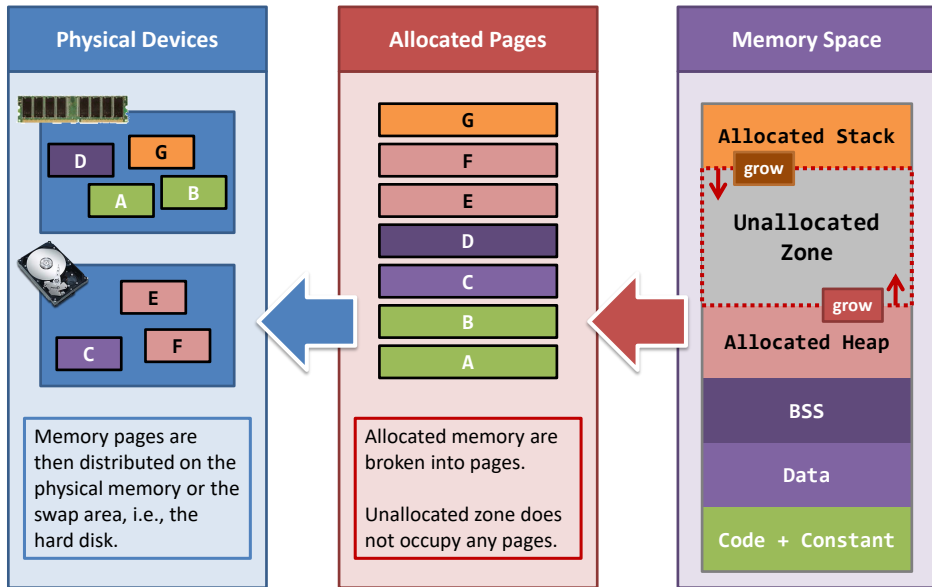


Paging – internal fragmentation

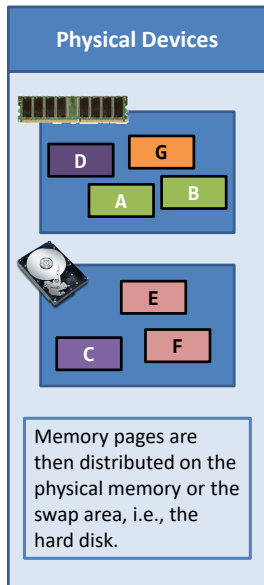
How about letting another process to use the “unused space”?



Paging – putting it together



Paging – page table design



- So, next waves of questions are:
 - Who can tell which virtual page is allocated?
 - Who can tell which page is on which device?
- Those questions can be answered by the design of the **page table**.

Paging – page table design

- How to design the page table?
 - First of all, which information need to be maintained?
 - Mapping from virtual pages to physical pages (called frames)
 - Permission information
 - Where is the page (in memory or not)
 - Second...
 - Each process needs one page table

Paging – page table design

Page Table of Process A			
Virtual Page #	Permission	Valid-invalid bit	Frame #
A	rwX-	1	0
B	NIL	0	NIL
C	r--s	1	2
D	NIL	0	NIL
...

The physical memory is just **an array of frames**.
The size of a frame is 4KB.

This row means the virtual page “A” is mapped to the physical frame “0”.

This row, with **NIL**, means the virtual page “D” is **not allocated**.

Remember, the entire 4G memory zone is usually not fully utilized.

For the sake of convenience, we don't use addresses here. Also, this column **is not stored in the page table**.

Paging – page table design

Page Table of Process A			
Virtual Page #	Permission	Valid-invalid bit	Frame #
A	rwx-	1	0
B	NIL	0	NIL
C	r--s	1	2
D	NIL	0	NIL
...

This bit is to tell the CPU whether this row is valid or not.

If the row is invalid, it means that the virtual page is not in the memory.

Note. This is not the same as an unallocated page.

1 - valid, in memory.
0 - invalid, not in memory.

Paging – page table design

Page Table of Process A			
Virtual Page #	Permission	Valid-invalid bit	
A	rwx-	1	
B	NIL	0	
C	r--s	1	
D	NIL	0	
...	



s – means sharable.

How does the CPU check if you can write to a memory zone?

When a virtual address is translated to an **unallocated frame**...

OR

When you write to **read-only pages**...

OR

When you try to execute a non-executable pages...



SEGMENTATION FAULT!!

Paging – page table design

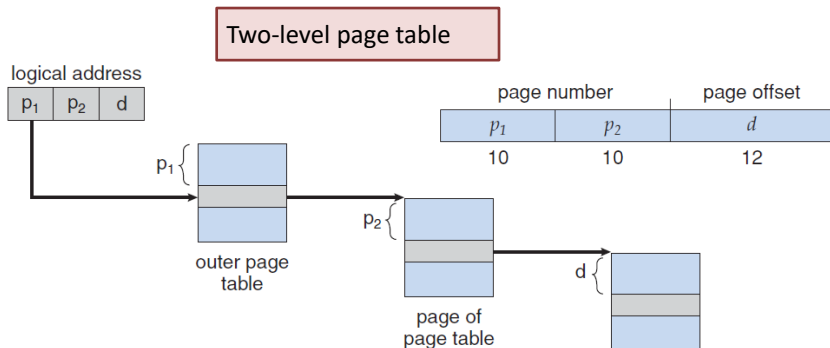
- Other design issues

How to store the page table if it is large (structure of page table)?

How to improve memory access performance (page table look incurs large overhead)?
Caching: Translation lookaside buffer (TLB)

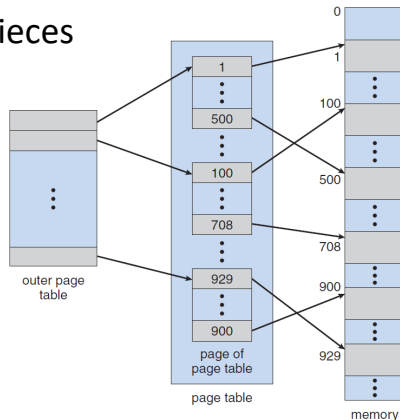
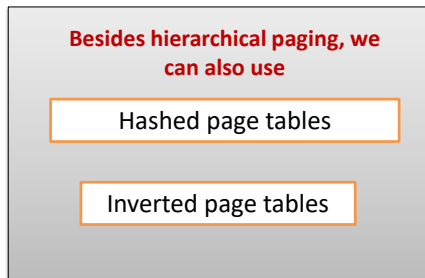
Paging – page table structure

- The page table may be large...multiple MBs
 - We would not want to allocate the page table contiguously in memory, how?
 - Divide the page table into pieces



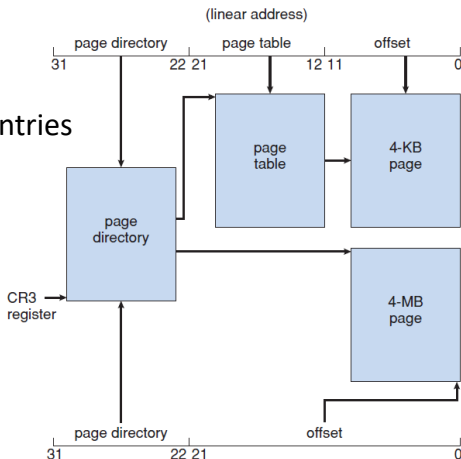
Paging – page table structure

- The page table may be large...multiple MBs
 - We would not want to allocate the page table contiguously in memory, how?
 - Divide the page table into pieces



Paging – Performance Boost

- Memory access requires to look up page table
 - This overhead is even larger with multi-level page tables
 - Any solution?
 - (1) large pages**
 - Reduce the page table entries
 - Cons?
 - Internal fragmentation
 - Deduplication



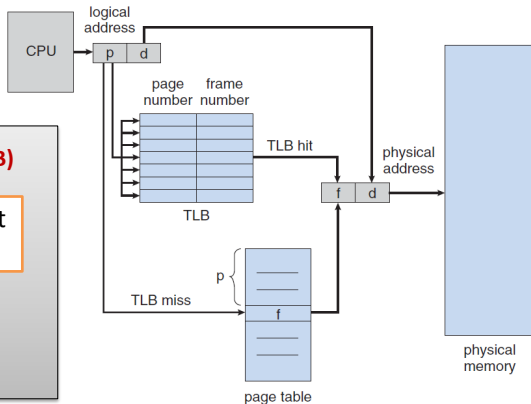
Paging – Performance Boost

- Memory access requires to look up page table
 - This overhead is even larger with multi-level page tables
 - Any solution?
 - (2) Caching**

Translation lookaside buffer (TLB)

The search in TLB is fast: Part of the instruction pipeline

The size of TLB is small:
e.g., 32-1024 entries



Paging – Performance Boost

- Memory access requires to look up page table
 - This overhead is even larger with multi-level page tables
 - Any solution?
 - (2) Caching**

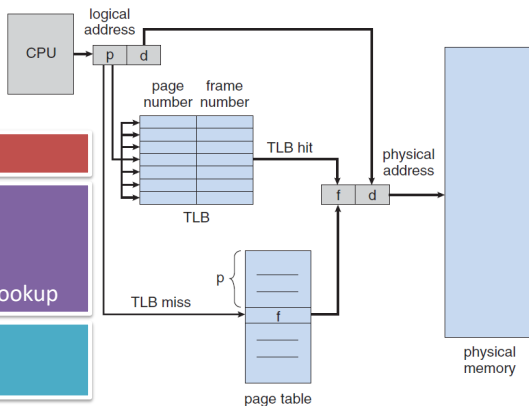
Effective memory-access time

Example:

- Hit ratio: 80%
- Mem access time: 100 ns
- One mem access for page table lookup

Effective mem-access time is

$$0.8 * 100 + 0.2 * (100 + 100) = 120 \text{ ns}$$

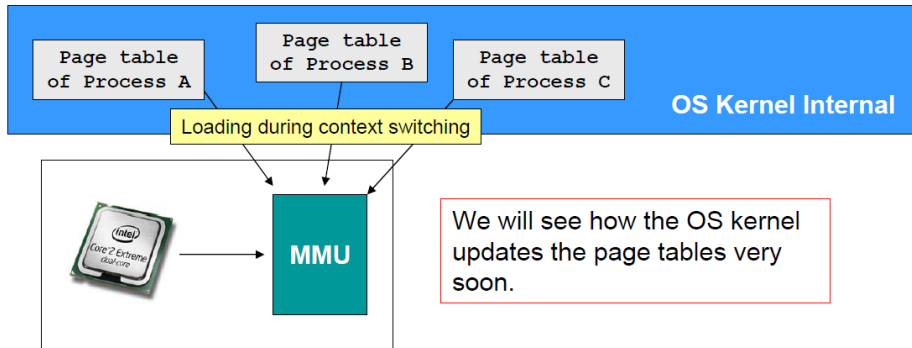


Paging – summary

- Virtual memory (VM) is just a table-lookup implementation. The specials about VM are:
 - The table-lookup is implemented inside the CPU, i.e., a hardware solution.
 - **Each process should have its own page table.**

Paging – summary

- How about the OS?
 - The OS stores and manages the page tables of all processes.



Paging – summary

- We talked about **segmentation** in part 1...
 - Address mapping can also be done in segments
 - Also permits physical address space of a process to be non-contiguous
 - But usually incurs severe **fragmentation** in both memory and backing store
- **Paging is used in most operating systems**
 - Hybrid scheme is also possible

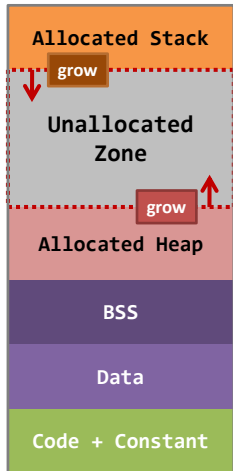
Memory Management

- Virtual memory;
- MMU implementation & paging;
- **Demand paging;**
- Page replacement algorithms;
- Allocation of frames;



Memory / page allocation?

- The stack and the heap will grow:
 - (1) calling **brk()**, i.e., the **heap** grows;
 - (2) calling nested function calls, i.e., the **stack** grows;
- The question is...
 - Will the memory be immediately allocated for you when you call **malloc()**?



Remember the OOM generator?

```
#define ONE_MEG 1024 * 1024

int main(void) {
    void *ptr;
    int counter = 0;

    while(1) {
        ptr = malloc(ONE_MEG);
        if(!ptr)
            break;
        counter++;
        printf("Allocated %d MB\n", counter);
    }

    return 0;
}
```

```
Allocated 3052 MB
Allocated 3053 MB
Allocated 3054 MB
Allocated 3055 MB
linux2:/uac/rsr/ykli> █
```

This program runs very fast,
why?

Memory / allocation – demand paging

- The reality is: allocation is done in a **lazy** way!
 - The system only **says** that the memory is allocated.
 - Yet, it is **not really allocated** until you access it.

```
1  #define BUF_SIZE  512 * 1024
2  void re() {
3      char buf[BUF_SIZE];
4      while( getchar() != '\n' );
5      memset(buf, 0, sizeof(buf));
6      while( getchar() != '\n' );
7      re();
8  }
9
10 int main(void) {
11     re();
12     return 0;
13 }
```

This statement does not involve any memory access.

So, the virtual address space is allocated, but **the page is not allocated yet**.

This statement really accesses the "allocated" memory.

So, this statement really **asks the system** to allocate memory.

Memory / allocation – demand paging

- How about the heap?

```
1  #define ONE_MEG (1024 * 1024)
2  #define COUNT  1024
3
4  int main(void) {
5      int i;
6      char *ptr[COUNT];
7      for(i = 0; i < COUNT; i++)
8          ptr[i] = malloc(ONE_MEG);
9
10     for(i = 0; i < COUNT; i++) {
11         while(getchar() != '\n');
12         memset(ptr[i], 0, ONE_MEG);
13     }
14 }
```

grow_heap.c

As a matter of fact, `malloc()` does not involve any memory allocation, only involving the allocation of the virtual address page.

So, this loop is only for enlarging the virtual page allocation.

This statement really accesses the “allocated” memory.

So, this statement really **asks the system** to allocate memory.

This lazy way is called **demand paging**, but how does it work?

Demand paging – illustration.

Assumption: 1 process only.

- Let's consider the “**grow_heap.c**” example.
 - Suppose that a process initially has 4 page frames.
 - We are now in the **memset()** for-loop in Lines 10 - 13.

OS kernel

Virtual page #	Bit	Frame #
A	1	0
...
D	1	3
E	0	NIL
...



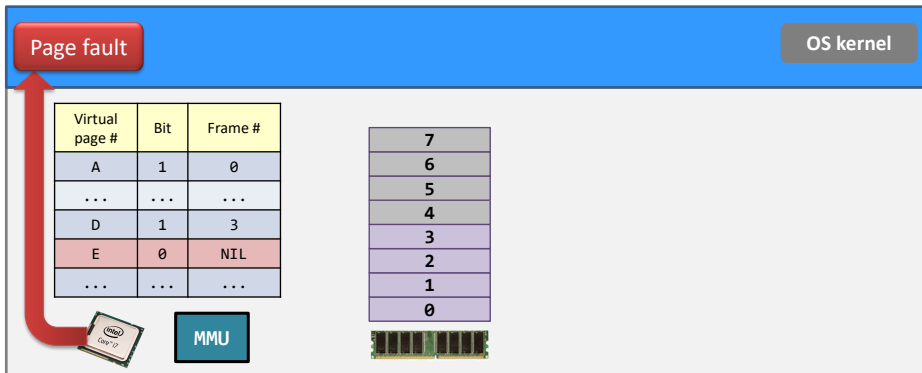
7
6
5
4
3
2
1
0



Demand paging – illustration.

Assumption: 1 process only.

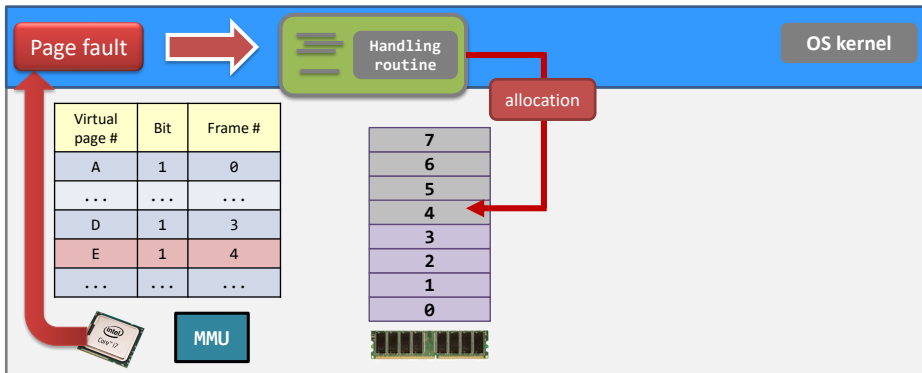
- When **memset()** runs,
 - the MMU finds that a **virtual page involved is invalid**,
 - the CPU then generates an interrupt called **page fault**.



Demand paging – illustration.

Assumption: 1 process only.

- The **page fault handling routine** is running:
 - The kernel knows the page allocation for all processes.
 - It allocates a memory page for that request.
 - Last, the **page table entry** for Page E is updated.



Demand paging – illustration.

Assumption: 1 process only.

- The routine finishes...and
- the **memset()** statement **is restarted**.
 - Then, no page fault will be generated until the next unallocated page is encountered.

OS kernel

Virtual page #	Bit	Frame #
A	1	0
...
D	1	3
E	1	4
...

OK



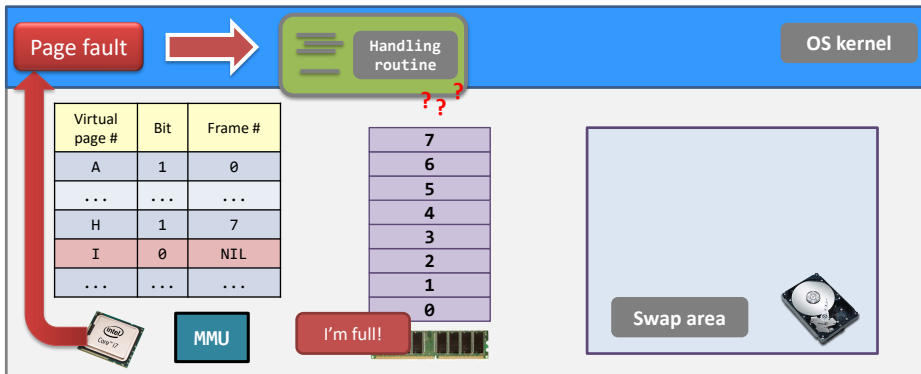
7
6
5
4
3
2
1
0



Demand paging – illustration.

Assumption: 1 process only.

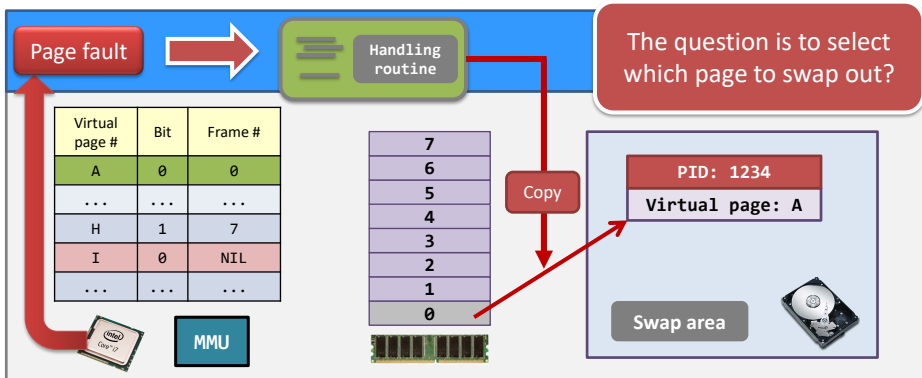
- So, how about the case when the routine finds that **all frames are allocated**?
 - Then, we need the help of the **swap area**.



Demand paging – illustration.

Assumption: 1 process only.

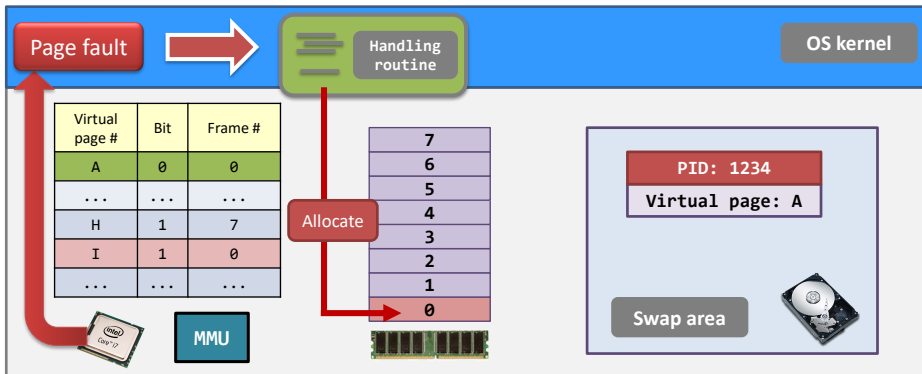
- Using the swap area:
 - Step (1) Select a **victim virtual page** and copy the victim to the swap area.
 - Now, Frame 0 is a free frame and the bit for Page A is 0.



Demand paging – illustration.

Assumption: 1 process only.

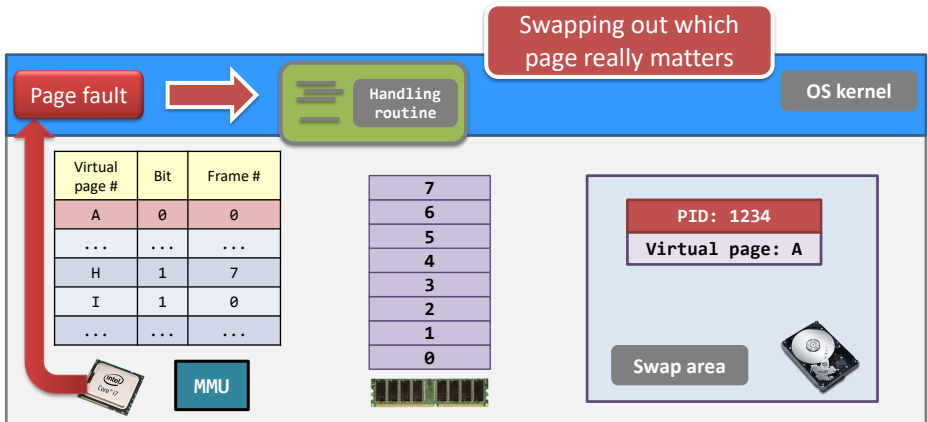
- Using the swap area:
 - Step (2) Allocate the free frame to the new frame allocation request.
 - Now, Page I takes Frame 0.



Demand paging – illustration.

Assumption: 1 process only.

- How about **virtual page A** is accessed again?
 - Of course, a page fault is generated, and
 - steps similar to the previous case takes place.



OOM generator

- Now, you should understand why this OOM generator run very fast.

```
#define ONE_MEG  1024 * 1024

int main(void) {
    void *ptr;
    int counter = 0;

    while(1) {
        ptr = malloc(ONE_MEG);
        if(!ptr)
            break;
        counter++;
        printf("Allocated %d MB\n", counter);
    }

    return 0;
}
```

The memory page frames are not really allocated (demand paging).

[It is only for enlarging the virtual page allocation.](#)

Real OOM – code

```
#define ONE_MEG 1024 * 1024

int main(void) {
    void *ptr;
    int counter = 0;

    while(1) {
        ptr = malloc(ONE_MEG);
        if(!ptr)
            break;
        memset(ptr, 0, ONE_MEG);
        counter++;
        printf("Allocated %d MB\n", counter);
    }

    return 0;
}
```

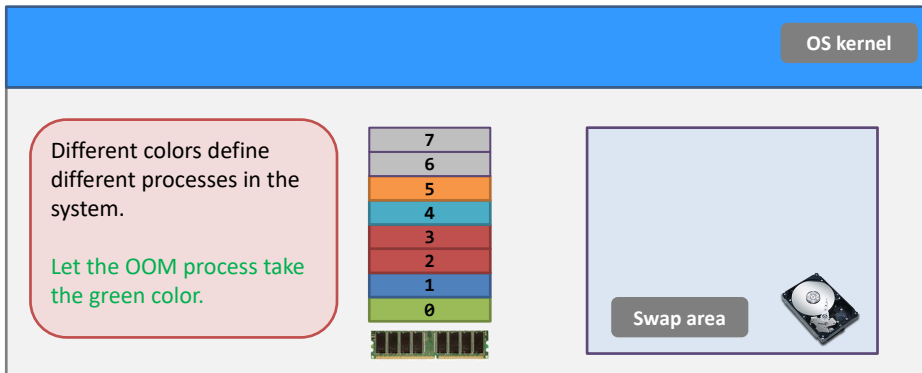
Warning #1. Don't run this program on any department's machines.

Warning #2. Don't run this program when you have important tasks running at the same time.

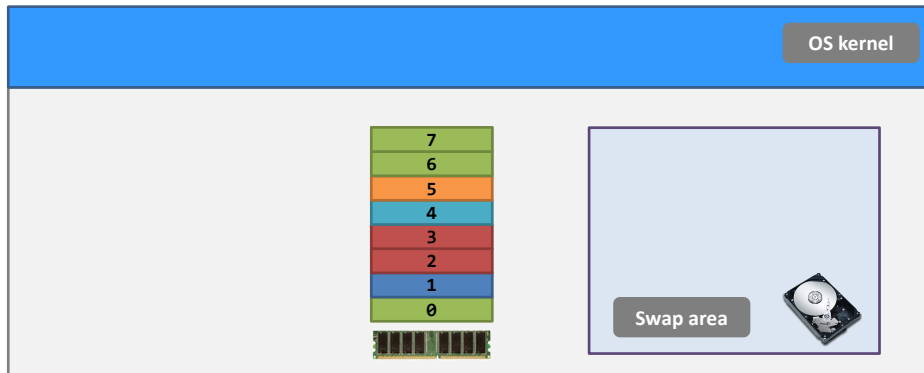
How does this program “eat” your memory?

What is the consequence after running this program?

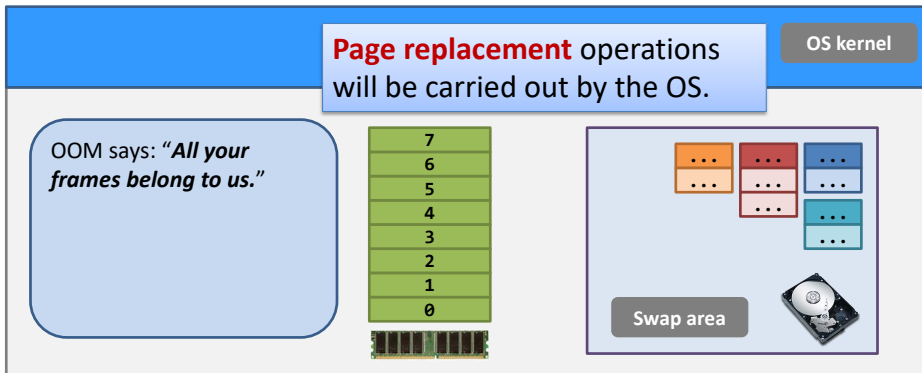
- So, what will happen when the real OOM program is running?
 - Suppose the OOM program has just started with **only one page allocated**. (For illustration only!)



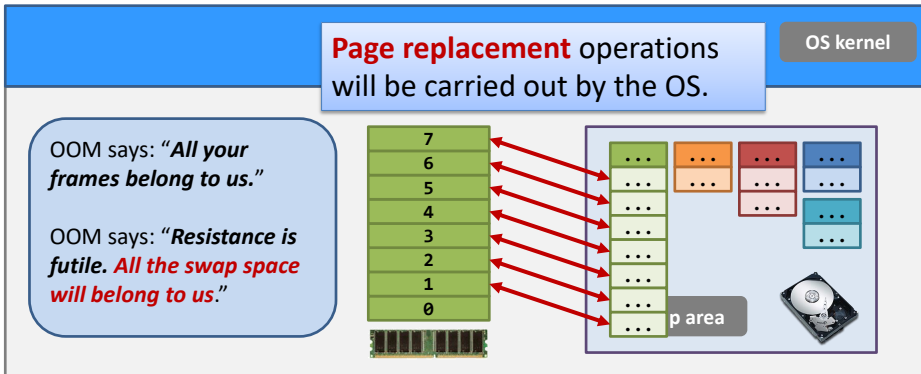
- OOM is running...1st stage.
 - The **free memory frames** are the first zone that the process has conquered.
 - All other processes could hardly allocate pages.



- OOM is running...2nd stage.
 - Occupied memory frames are the next zone that the process conquers (no unused frames).
 - **Disk activity flies high!**

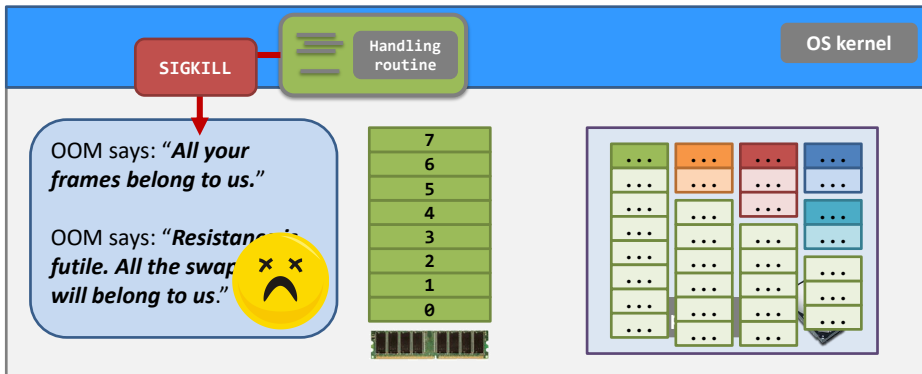


- OOM is running... **3rd stage**.
 - The previously-conquered frames are swapping to the swap area.
 - **Disk activity flies high!**



Real OOM – illustration

- OOM is running...Final stage.
 - The page fault handling routine finds that:
 - **No free space left in the swap area!**
 - **Decided to kill the OOM process!**

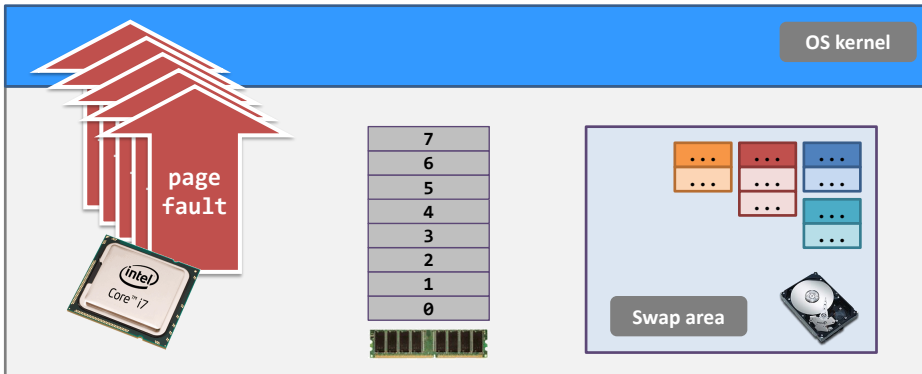


Real OOM – illustration

- OOM has died, but... Painful aftermath.

- **Lots of page faults! Why?**

- It is because other processes need to take back the frames!
 - **Disk activity flies high again**, but will go down eventually.



Demand paging - Issues

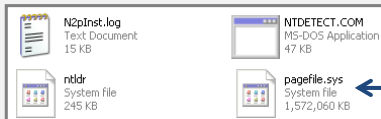
- Swap area
 - Where is it?
 - How large is it?
- Can we run a really large process (e.g., bigger than physical memory)?
 - How large is it at most?
- How about `fork()` and `exec*()`?
 - Can they be clever?

Swap area – location

- The swap area is usually **a space reserved** in a permanent storage device.

Linux needs a separate partition and it is called the **swap partition**.

```
$ sudo fdisk /dev/sda
.....
Command (m for help): p
.....
/dev/sda1 ..... Linux
/dev/sda2 ..... Linux swap / Solaris
Command (m for help): _
```



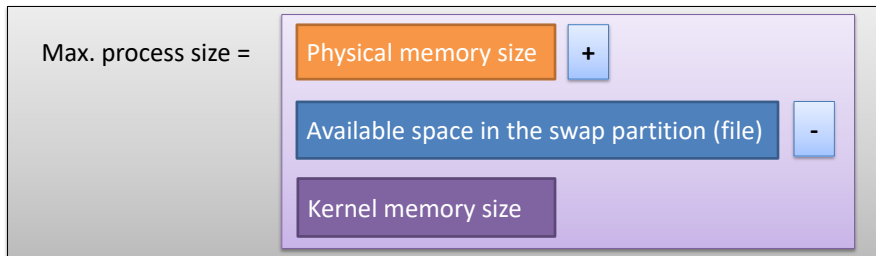
Windows hides a file “**pagefile.sys**”, which is the swap area, in one of the drives.

Swap area - size

- How large should the swap space be?
 - It should be at least the **same** as the size of the physical memory, so that ...
 - when a really large process wants to take all the memory...
 - all the pages on the physical memory can find a place to hide.
 - An old rule said that “***swap should be twice the size of the physical memory***”.
 - But, I can't find the reasons anymore, and this rule does not hold nowadays because we now have too much RAM!

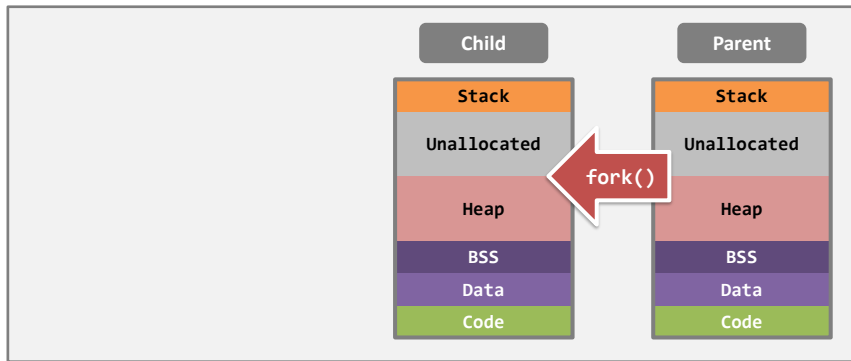
How about running large programs

- When a process is larger than the physical memory, is it able to run?
 - No need to load all data in memory...Demand paging
 - Generates **page fault** to allocate physical page frames
 - Trigger **page replacement** if there is no unused frames
- How large is a process that a system can support



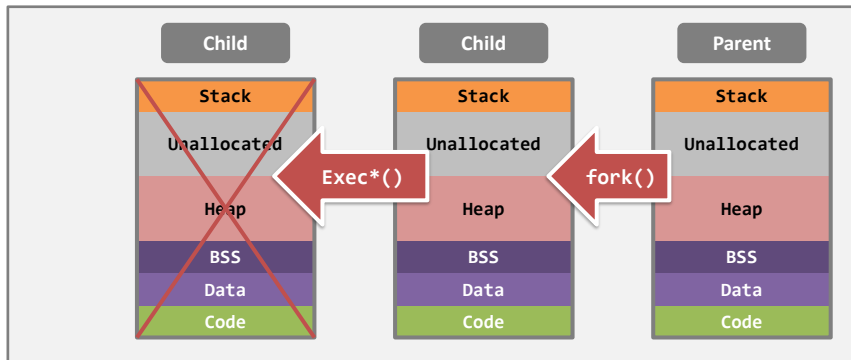
How about `fork()` & `exec()`

- What we have learned about the `fork()` system call is...**duplication!**
 - The parent process and the child process **are identical** from the userspace memory point of view.



How about **fork()** & **exec()**

- What does duplication mean? Allocate new pages for the child process?
 - If yes...then consider **exec*()** system call as well...
 - Isn't it stupid?



How about **fork()** & **exec()**

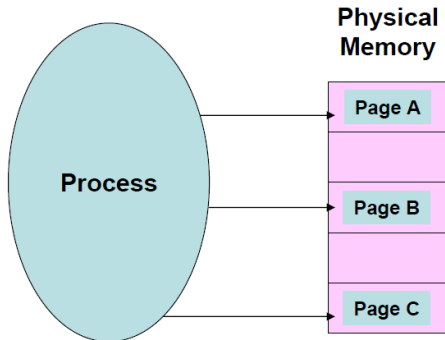
- Can we have a clever design with demand paging?
 - A technique called **copy-on-write** is implemented

Copy-on-write technique allows the parent and the child processes to **share** pages after the **fork()** system call is invoked.

A new, separated page will be **copied and modified** only when one of the processes **wants to write** on a shared page.

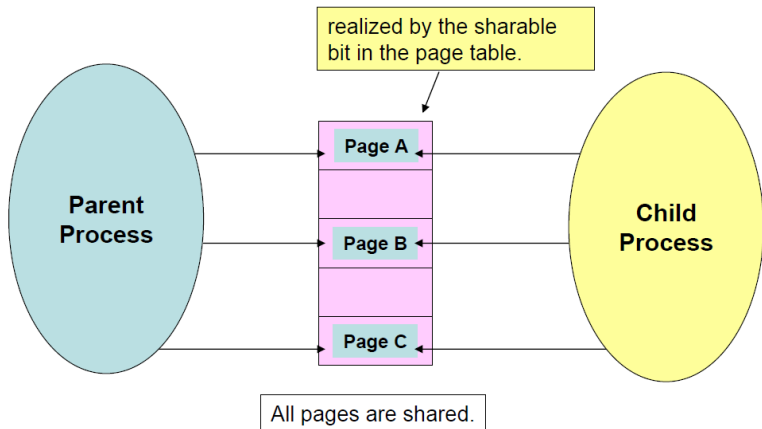
Copy-on-Write (COW)

- Before **fork()** ...



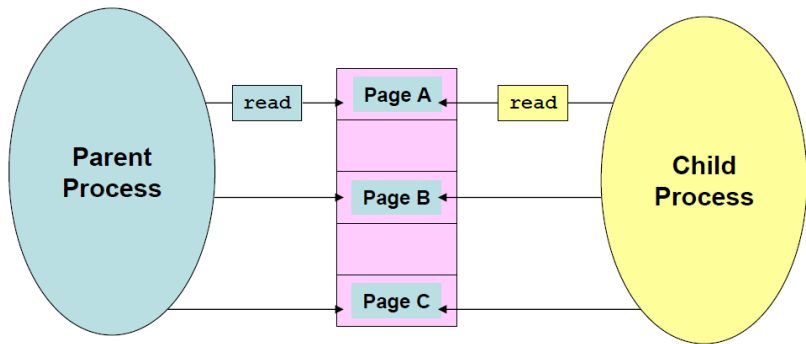
Copy-on-Write (COW)

- Right after **fork()** in invoked ...



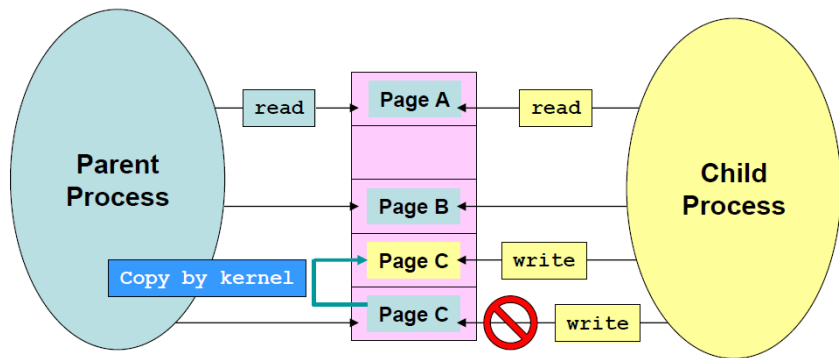
Copy-on-Write (COW)

- When both processes read the pages...



Copy-on-Write (COW)

- When one of the processes write to a shared page...



Demand paging - performance

- Demand paging can significantly affect performance
 - Service the page fault interrupt
 - Read in the page
 - Restart the instruction/process
- How to characterize?
 - Effective access time
 - $(1 - p) \times ma + p \times \text{page fault time}$
 - ma : memory access time (10-200ns)
 - p : prob. of a page fault
 - page fault time : ms

Example

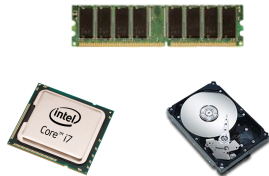
- *ma*: 200ns, *page fault time*: 8ms
- 1/1000 page fault probability
 - Effective access time: $(1 - p)200ns + p \times 8ms = 8.2\mu s$
- To allow 10% performance degradation only
 - $(1 - p)200ns + p \times 8ms < 220ns$
 - $p < 0.0000025$
- Thus, page fault rate must be low

Summary of demand paging

- Demand paging enables **over-commitment**
 - Large process can be supported
 - Concurrent running of multiple processes is also supported
- One key issue is...
 - How to select victim pages to swap out?
 - Page-replacement algorithm

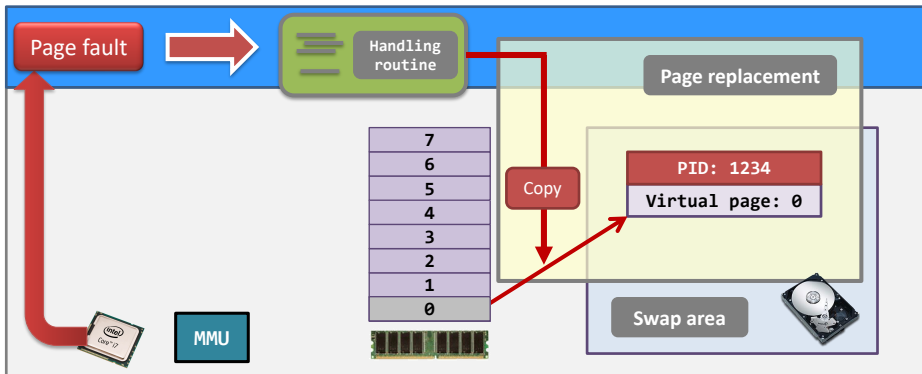
Memory Management

- Virtual memory;
- MMU implementation & paging;
- Demand paging;
- **Page replacement algorithms;**
- Allocation of frames;



Page replacement – introduction

- Remember the page replacement operation?
 - It is the job of the kernel to **find a victim page** in the physical memory, and...
 - write the victim page** to the swap space.

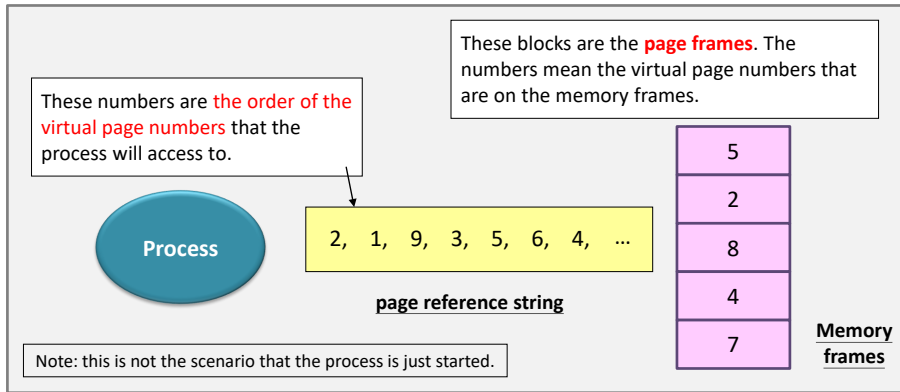


Page replacement – introduction

- Replacing a page involves disk accesses, therefore a page fault is **slow and expensive**!
 - Key issue: which page should be swapped out?
 - Page replacement algorithms should minimize further page faults.
- In the following, we introduce four algorithms:
 - Optimal;
 - First-in first-out (FIFO);
 - Least recently used (LRU);
 - Second-chance algorithm

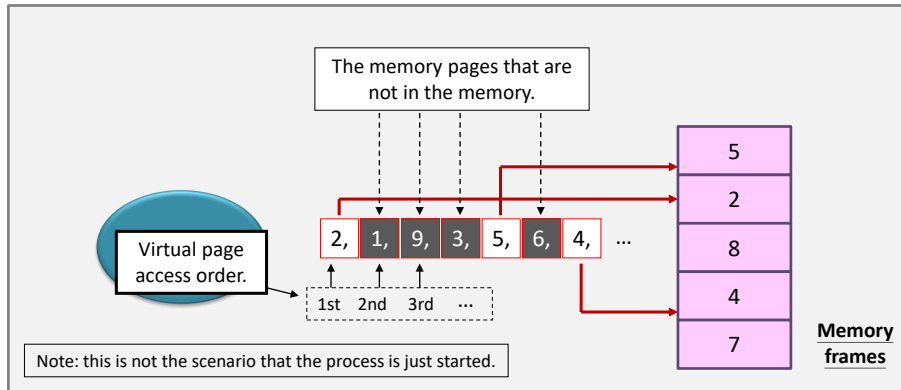
Page replacement – algorithm

- Imagine that you are the kernel...
 - you have a process just started to run;
 - the process' memory is larger than the physical memory;
 - assume that all the pages are in the swap space.



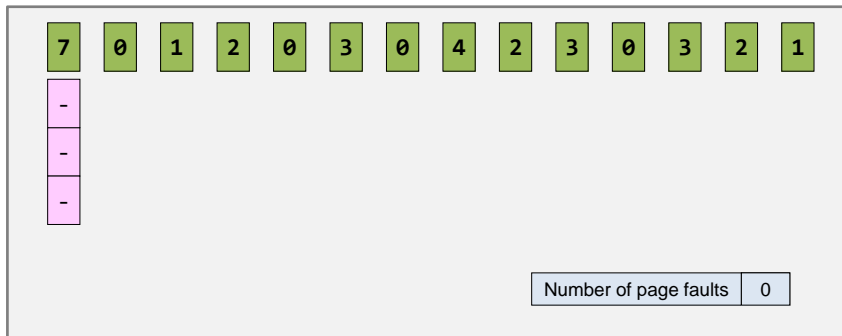
Page replacement – algorithm

- Imagine that you are the kernel...
 - you have a process just started to run;
 - the process' memory is larger than the physical memory;
 - assume that all the pages are in the swap space.



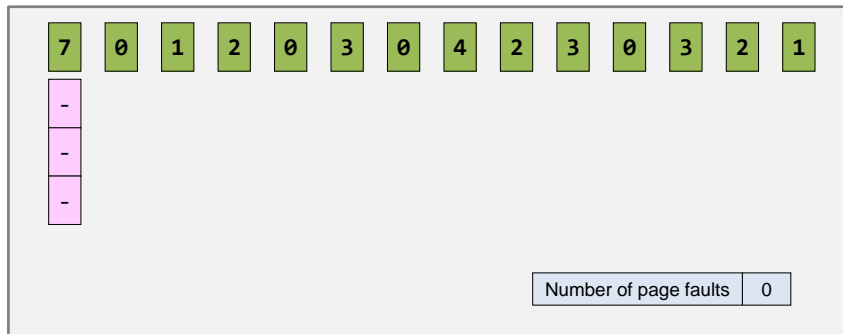
Page replacement – when an algorithm starts

- Initial condition
 - Let all the frames be empty.



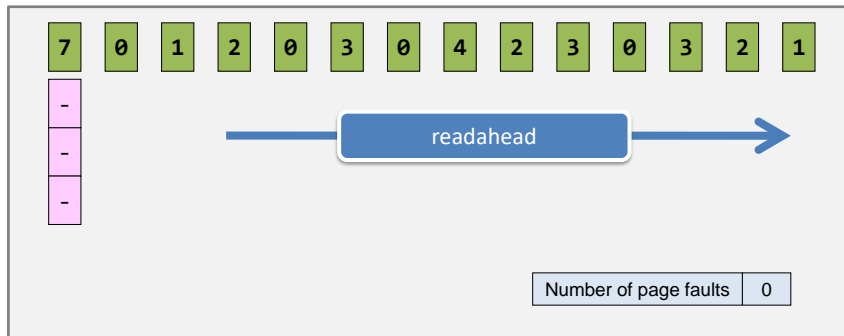
Page replacement – optimal algorithm

- What is the best algorithm?
 - Do not worry about the implementation at this moment.



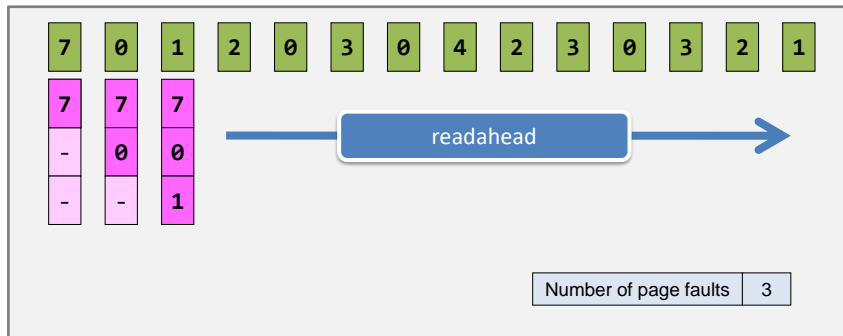
Page replacement – optimal algorithm

- If I **know the future**, then I know how to do better.
 - That means I can optimize the result if the page reference string is given in advance.
 - That's why the algorithm is called “optimal”.



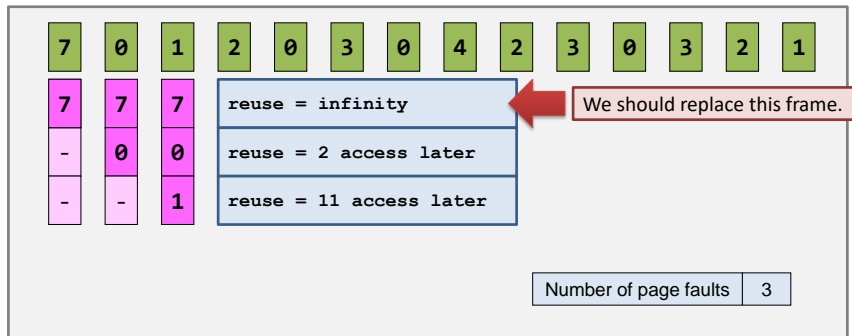
Page replacement – optimal algorithm

- If I **know the future**, then I know how to do better.
 - The first page request will cause a **page fault**.
 - Because there are free frames, no replacement is needed.



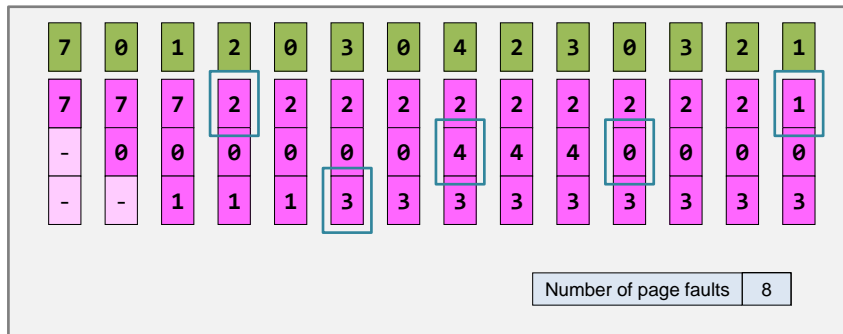
Page replacement – optimal algorithm

- Replace strategy:
 - To replace the page that **will not be used for the longest period of time**.



Page replacement – optimal algorithm

- The story goes on...
 - But, do you think that this is a **non-sense**?
 - Of course, this is to give you a sense that **how close** an algorithm is from the optimal.

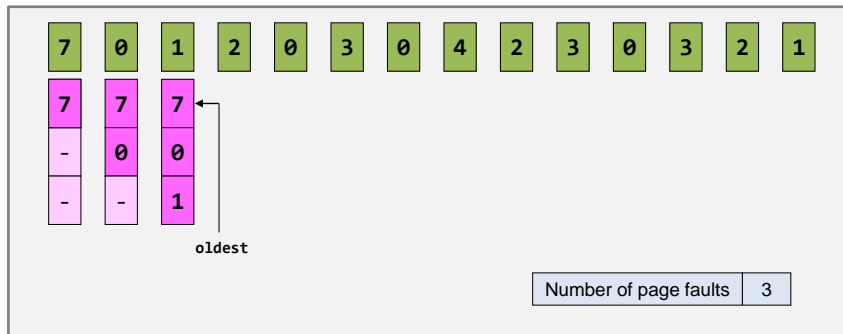


Page replacement – Problem of the optimal algorithm

- Unfortunately, you never know the future...
 - It is not practical to implement such an algorithm
 - Is there any easy-to-implement algorithm?
 - You have already learnt process scheduling
- FIFO: the **first page being swapped into** the frames will be the **first page being swapped out**.
 - The victim page will always be the oldest page.
 - The age of a page is counted by the time period that it is stored in the memory.

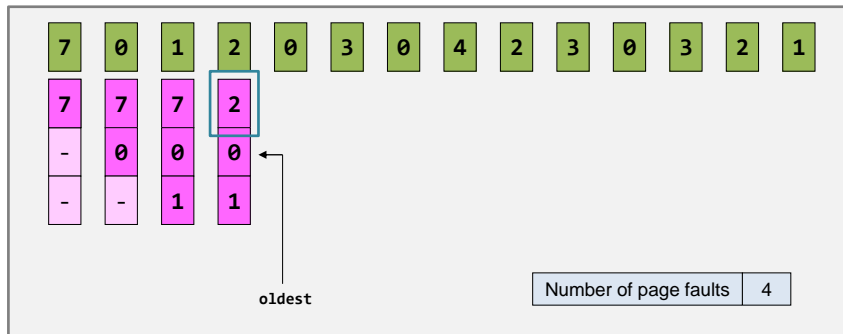
Page replacement – FIFO algorithm

- When there is no free frames,
 - The **FIFO page replacement algorithm** will choose the oldest page to be the victim.



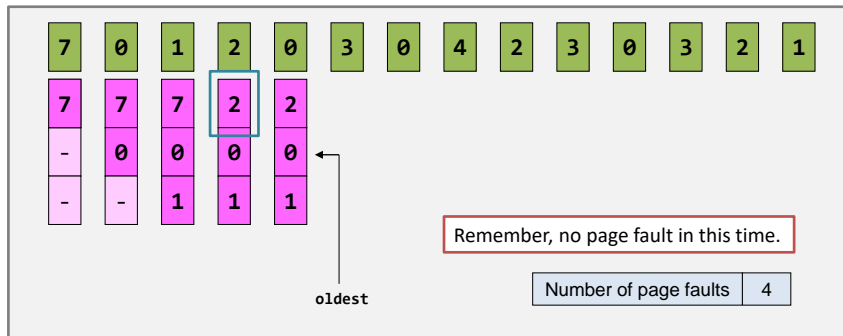
Page replacement – FIFO algorithm

- When there is no free frames,
 - The **FIFO page replacement algorithm** will choose the oldest page to be the victim.
 - Of course, the oldest page changes.



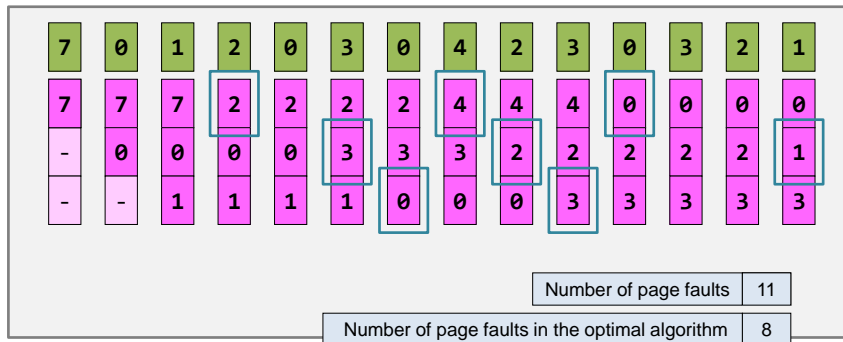
Page replacement – FIFO algorithm

- When a memory reference can be found in the memory, will the age of that frame be changed?
 - NO! The frame storing “page 0” is still the oldest frame.



Page replacement – FIFO algorithm

- The story goes on...
 - Seems that there is **no intelligence** in this method...
 - Pages which will be accessed again are swapped out

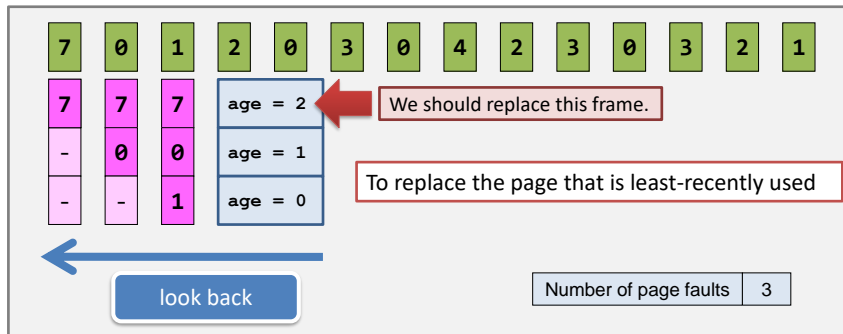


Page replacement – LRU algorithm

- Can we do better?
 - Still remember the locality rule?
 - Recently accessed pages may be accessed again in near future
 - Why not swap out the pages which are not accessed recently
 - This is the **least-recently-used** (LRU) page replacement.

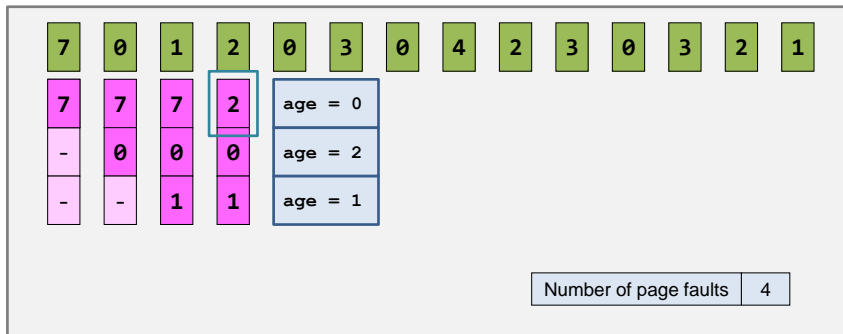
Page replacement – LRU algorithm

- Strategy:
 - Attach every frame with an age, which is an integer.
 - When a page is just accessed,
 - no matter that page is originally on a frame or not, **set its age to be 0**.
 - Other frames' ages are incremented by 1.



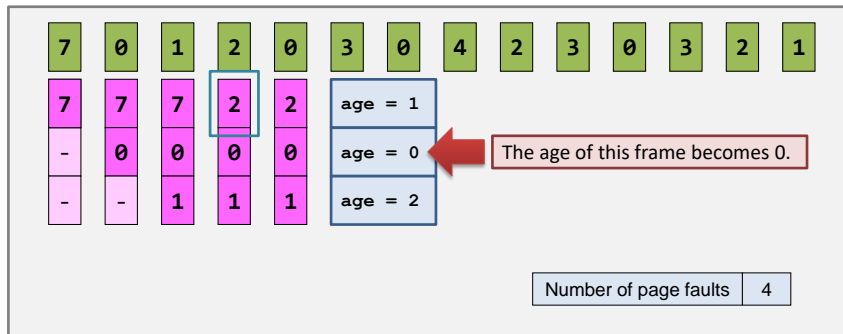
Page replacement – LRU algorithm

- Strategy:
 - Attach every frame with an age, which is an integer.
 - When a page is just accessed,
 - no matter that page is originally on a frame or not, **set its age to be 0**.
 - Other frames' ages are incremented by 1.



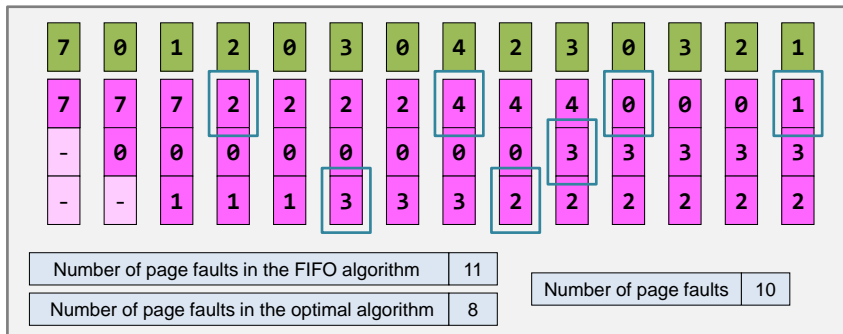
Page replacement – LRU algorithm

- Strategy:
 - Attach every frame with an age, which is an integer.
 - When a page is just accessed,
 - no matter that page is originally on a frame or not, **set its age to be 0**.
 - Other frames' ages are incremented by 1.



Page replacement – LRU algorithm

- Strategy:
 - Attach every frame with an age, which is an integer.
 - When a page is just accessed,
 - no matter that page is originally on a frame or not, **set its age to be 0**.
 - Other frames' ages are incremented by 1.

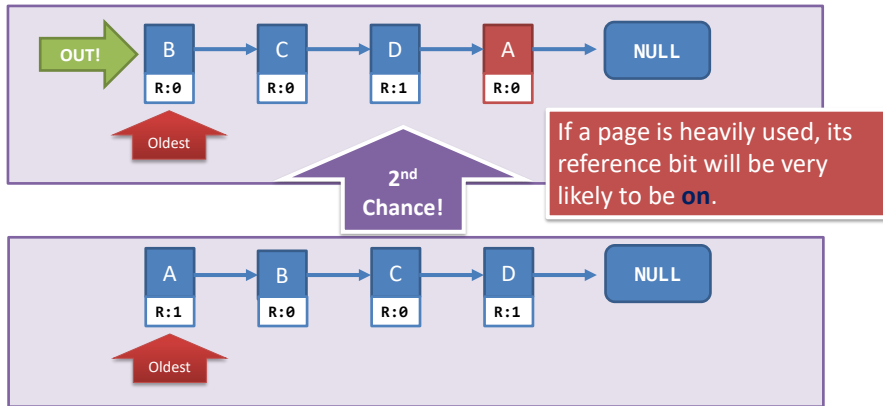


Page replacement – LRU algorithm

- The performance of LRU is considered to be good, but **how to implement** the LRU algorithm efficiently
 - Counters: requires to update counter and search the table to find the page to evict
 - Stack: implement with doubly linked list (pointer update)
- Common case in many systems
 - A reference bit for each page (set by hardware)
 - LRU approximation: **Second-chance algorithm**

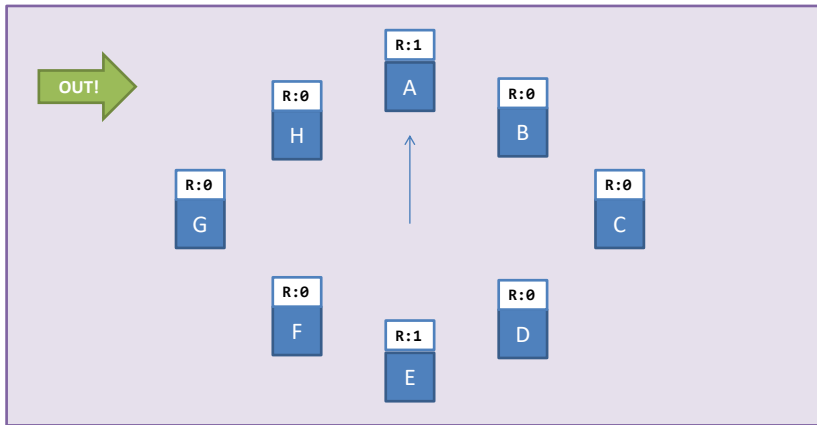
Page replacement – LRU approximation

- Second-chance algorithm
 - Basic: FIFO
 - Give the page a second chance if its reference bit is on



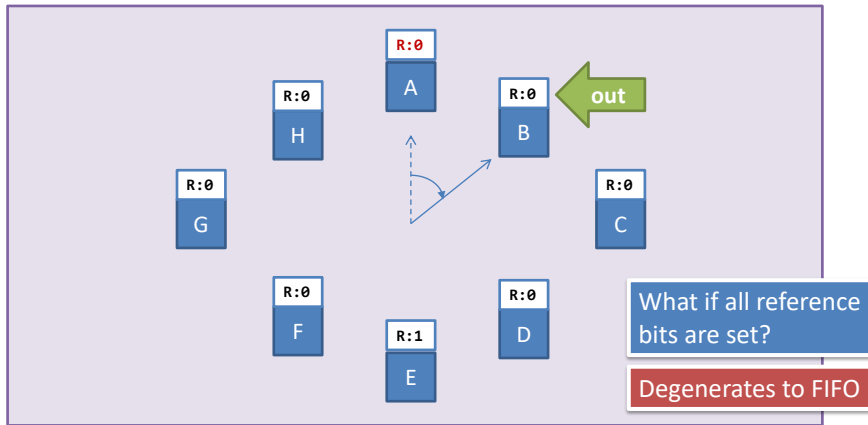
Page replacement – LRU approximation

- Clock is the efficient implementation of the 2nd chance algorithm (**circular queue**).



Page replacement – LRU approximation

- Clock is the efficient implementation of the 2nd chance algorithm (**circular queue**).

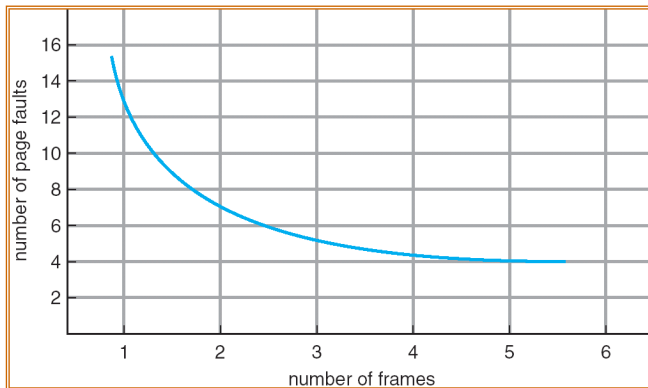


Page replacement – performance

- Number of page frames VS Performance.
 - Increasing the number of page frames implies increasing the amount of the physical memory.
- So, it is natural to think that:
 - I have more memory...and more frames...
 - Then, my system **must be faster** than before!
 - Therefore, the number of **page faults must be fewer** than before, given the same page reference string.

Page replacement – performance

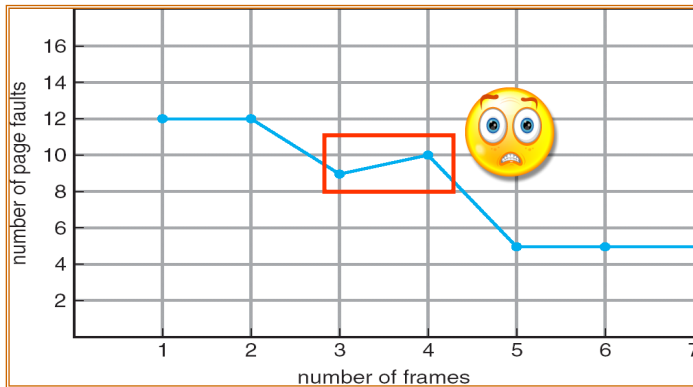
- Your expectation:



Page replacement – performance

- The reality may be:

This is called Belady's anomaly



Page replacement – performance

- Try the following:
 - all page frames are initially empty;
 - use **FIFO** page replacement algorithm;
 - use the number of frames: 3, 4, and 5.
 - The page reference string is:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Page replacement – performance

- Belady's anomaly exists for some algorithms
 - Both optimal and LRU do not suffer from it
- **Stack algorithms:** never exhibit Belady's anomaly
 - Feature: The set of pages in memory for n frames is always a **subset** of the set of pages in memory for $n + 1$ frames
 - Example: LRU
 - The n most recently referenced pages will still be the most recently referenced pages when the number of frames increases

Memory Management

- Virtual memory;
- MMU implementation & paging;
- Demand paging;
- Page replacement algorithms;
- **Allocation of frames;**



Allocation for user processes

- Free-frame list
 - Demand paging and page replacement
- Constrains
 - Limit on number of frames
 - Upper bound: total available frames
 - Lower bound: has a minimum number
 - Performance consideration (limit page-fault rate)
 - Defined by computer architecture (instructions)
 - Process will be suspended if the number of allocated frames falls below the minimum requirement
 - Global / local allocation (replacement)

Allocation algorithm

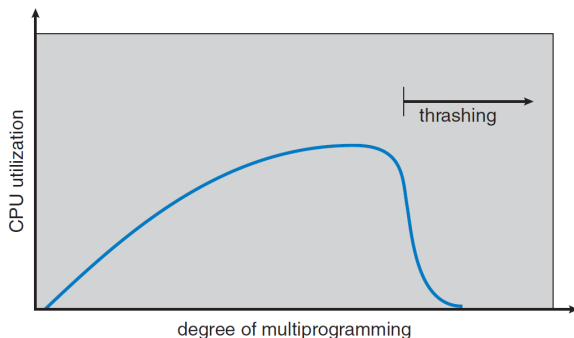
- Equal allocation
 - m frames among n processes
 - $\frac{m}{n}$ frames for each process
 - Memory waste
- Proportional allocation
 - Size of process p_i is s_i , then allocate
 - $a_i = \frac{s_i}{\sum s_i} \times m$
- Priority-based scheme
 - Ratio depends on both process size and priority

Issues - Thrashing

- If a process does not have enough frames – number of frames required to support pages in active use
 - Frequent page fault
 - Replace a page that will be needed again right away
 - This is called thrashing
 - Spend more time paging than executing

Issues - Thrashing

- Example: Multiprogramming + global page replacement
 - Increase CPU utilization (increase degree of multiprogramming)
 - Frequent page fault (queue up for paging, reduce CPU utilization, increase degree of multiprogramming)



Issues - Thrashing

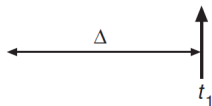
- How to address?
 - Local replacement/priority replacement
 - Will not cause other processes to thrash
 - Still not fully solve this problem
 - Increase average time for a page fault
 - longer queue for the paging device
 - longer effective access time even for non-thrashing processes

Issues - Thrashing

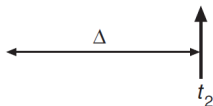
- How to address?
 - Provide as many frames as needed
 - Use working-set strategy to estimate needed frames
 - Working set: the set of pages in the recent Δ page references

page reference table

... 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...



$WS(t_1) = \{1, 2, 5, 6, 7\}$

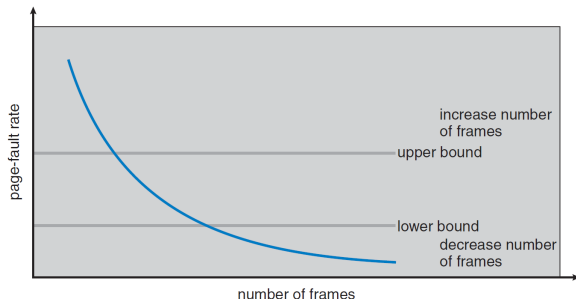


$WS(t_2) = \{3, 4\}$

$\sum WSS_i > m$: thrashing may occur

Issues - Thrashing

- How to address?
 - Provide as many frames as needed
 - Use working-set strategy to estimate needed frames
 - Working set: the set of pages in the recent Δ page references
 - Use page-fault frequency

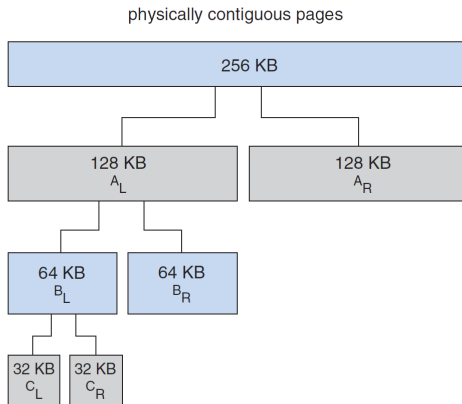


Allocation for kernel memory

- Kernel memory allocation requirement
 - Features
 - Varying (small) size requirement: different data structures
 - Contiguous requirement (certain hardware devices interact with physical memory)
 - Paging: Internal fragmentation
- Buddy system + Slab allocation

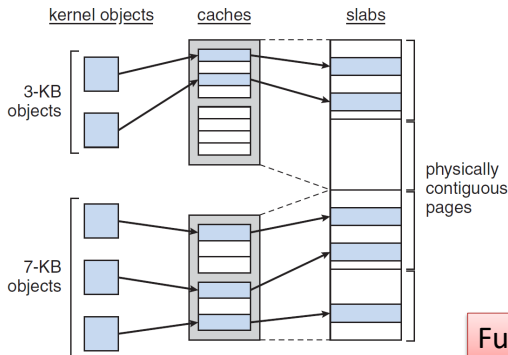
Buddy system

- Allocate memory from a fixed-size segment
 - Power-of-2 allocator (11 orders)
 - Advantage: coalescing



Slab allocation

- Allocate memory for small objects (limit fragmentation)
 - Slab: one/more contiguous pages
 - Cache: one/more slabs
 - A separate cache for each unique kernel data structure



Reduce fragmentation

Fast allocation
(caching benefit)

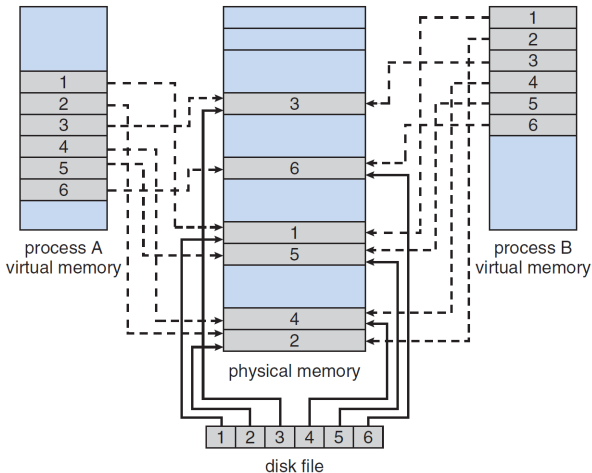
Further reading: SLOB/SLUB

Memory mapped file

- Ordinary file access
 - `open()`, `read()`, `write()`
 - System call + disk access
- Memory mapped file
 - Memory mapping a file: associate a part of the virtual address space with the file
 - File access
 - Initial access to file: demand paging
 - Subsequent reads/writes: routine memory accesses
 - Improves performance
 - Refer to `mmap(2)` system call

Memory mapped file

- Also allow multiple processes to map the same file



Summary

- We have introduced...
 - Segmentation
 - Paging + page table
 - Demand paging + COW + page replacement algorithms
 - Allocation of frames
 - User process
 - Thrashing
 - Kernel memory (buddy + slab)
 - Memory-mapped file
- More...
 - **malloc()** is not that simple: refer to “glibc malloc”
 - Other page-replacement algorithms

Hope you enjoyed the OS course!

Operating Systems

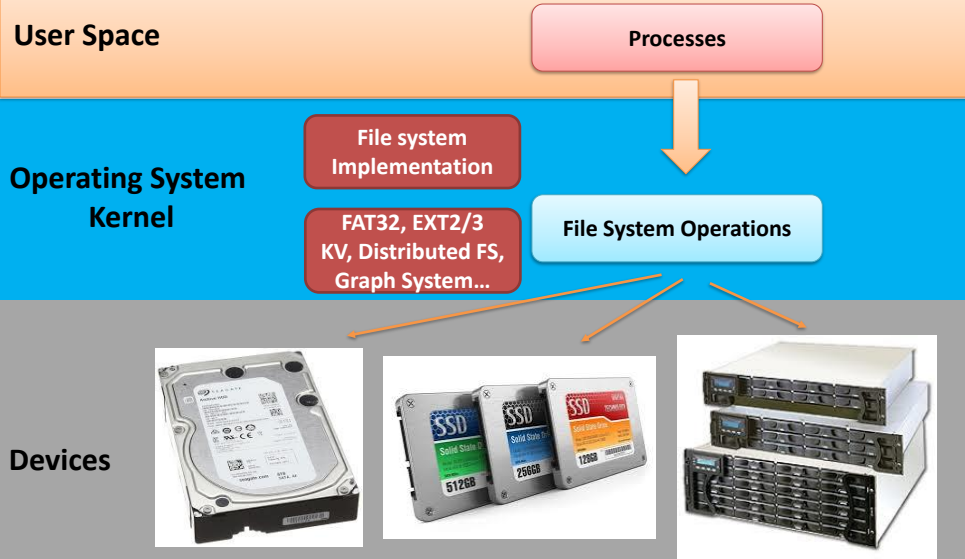
Associate Prof. Yongkun Li

中科大-计算机学院 副教授

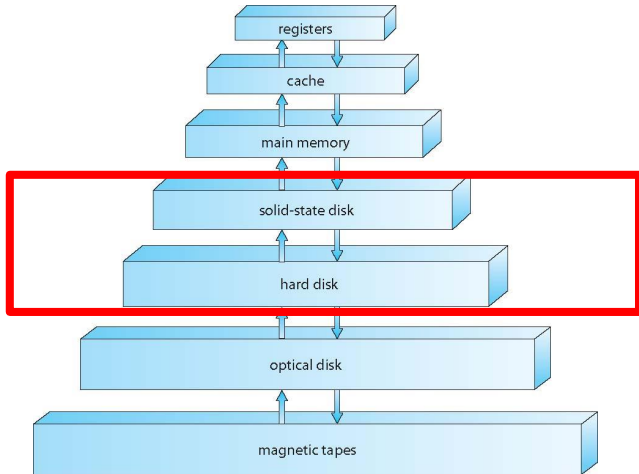
<http://staff.ustc.edu.cn/~ykli>

Chapter 8 Mass Storage

Topics in Part 3 (Storage Management)



Storage Hierarchy



Topics (Mass Storage)



Disk Structure

Disk Scheduling



SSD Structure

SSD Features/Issues



RAID

Erasure Coding

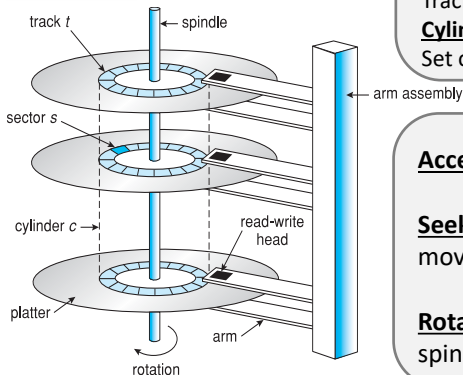
Research Problems

Topics

- **Disk structure**
- Disk scheduling
- Solid-state drives (SSDs)
- RAID
- Erasure coding



Hard Disk Structure – Physical view



Physical address (cylinder, track, sector)

Track:

The surface of a platter is divided into tracks

Sector:

Track is divided into sectors (512B data + ECC)

Cylinder:

Set of tracks that are at one arm position

Access: Seek + Rotate

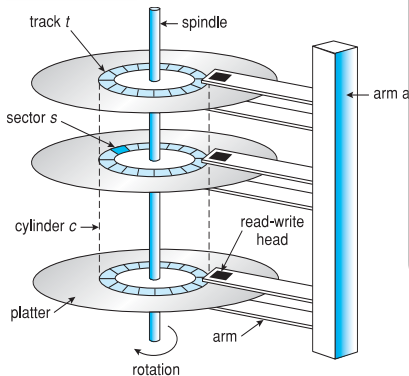
Seek time:

move disk arm to desired cylinder

Rotational latency:

spin at 5400/7200/10K/15K RPM

Hard Disk Structure – Physical view



Constant liner velocity (CLV)

- Uniform density of bits per track, outer track hold more sectors
- Variable rotation speed to keep the same rate of data moving
- CD-ROM/DVD-ROM

Constant angular velocity (CAV)

- Constant rotation speed
- Higher density of bits in inner tracks
- Hard disks

Hard Disk Structure – Logical view



How to use?

Large 1-D arrays of logical blocks (usually 512 bytes)

Address mapping

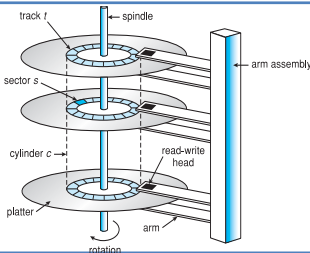
Logical block number -> (cylinder #, track #, sector #)

Disk management is required

- Disks are prone to failures: defective sectors are common (bad blocks)
 - ✓ Need to handle defective sectors: bad block management
- Disk formatting

Disk Management

Bad Block Management



- ✓ Maintain a list of bad blocks (initialized during low-level formatting) and preserve an amount of spare sectors
- ✓ **Sector sparing/forwarding:** replace a bad sector logically with one spare sector
 - Problem: invalidate disk scheduling algorithm
 - Solution: spare sectors in each cylinder + spare cylinder
- ✓ **Sector slipping:** remap to the next sector (data movement is needed)

Disk Management

Disk Formatting

Step 1: Low-level formatting/physical formatting

- ✓ Divide into sectors so disk controller can read/write
- ✓ Fills the disk with a special data structure for each sector (data area(512B), header and trailer (sector number & ECC))
 - The controller automatically does the ECC processing whenever a sector is read/written
- ✓ Done at factory, used for testing and initializing (e.g., the mapping). It is also possible to set the sector size (256B, 512B, 1K, 4K)

Disk Management

Disk Formatting

Step 2: How to use disks to hold files after shipment?

➤ Choice 1: File system

- ✓ Partition into one or more groups of cylinders (each as a separate disk)
- ✓ Logical formatting: creating a FS by storing the initial FS data structures
- ✓ I/O optimization: Disk I/O (via blocks) & file system I/O (via clusters), why?
 - More sequential access, fewer random access

➤ Choice 2: Raw disk

- ✓ Use disk partition as a large sequential array of logical blocks, without FS
- ✓ Raw I/O: bypass all FS services (buffer cache, prefetching...), be able to control exact disk location

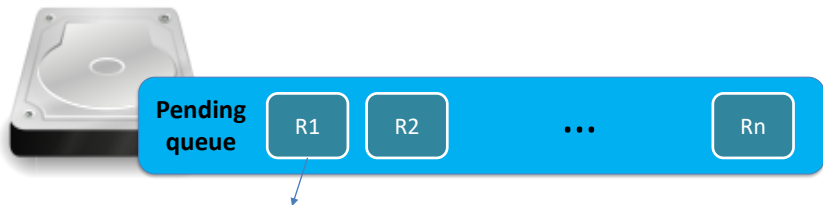
Topics

- Disk structure
- **Disk scheduling**
- Solid-state drives (SSDs)
- RAID & Erasure codes
- Problems with EC



Why needed?

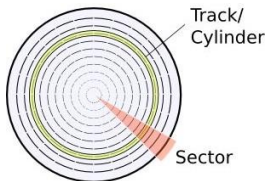
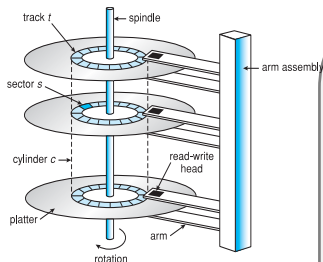
- Requests are placed in the queue of pending requests for that drive if the drive/controller is busy



Read/write, disk address, memory address,
number of sectors to be transferred

Request ordering significantly affects the access performance (seek + rotate), so scheduling is needed

What is disk scheduling



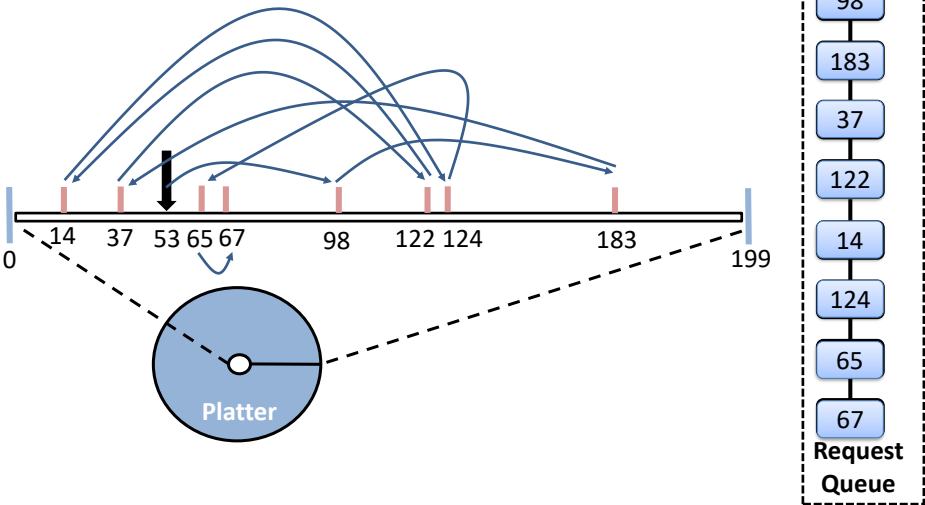
- I/O access procedure
 - Seek: move the head to the desired cylinder
 - Rotate: spin to the target sector on the track
- Disk scheduling
 - Choose the next request in the pending queue to service so as to minimize the **seek time**
- Scheduling algorithms

FCFS Scheduling

- First-come, first-served (FCFS)
 - Intrinsically fair, but does not provide the fastest service

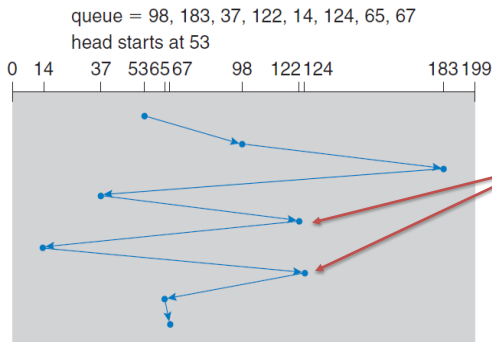
FCFS Scheduling

- First-come, first-served (FCFS)



FCFS Scheduling

- Scheduling diagram



Total head movement
(640 cylinders)

Wild swing is very common
E.g.: 122 to 14, then to 124

How to reduce the head movement?

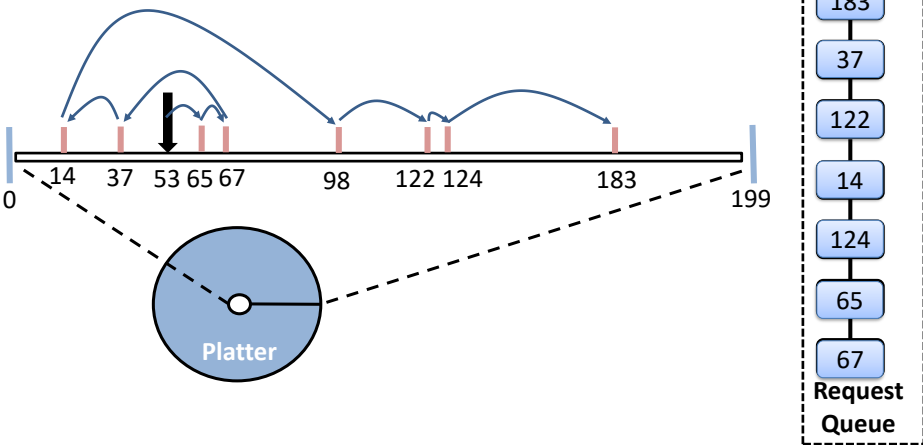
Handle nearby requests first

SSTF Scheduling

- Shortest seek time first (SSTF)
 - Choose the request with the least seek time
 - Choose the request closest to the current head position

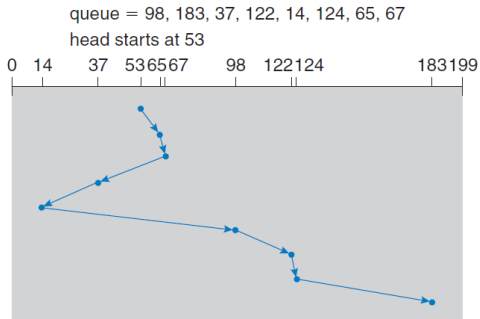
SSTF Scheduling

- Shortest seek time first (SSTF)



SSTF Scheduling

- Scheduling diagram



Total head movement: 236 cylinders (it is 640 for FCFS)

Essentially a form of SJF scheduling

It is not optimal

The sequence of 53-37-14-65... could reduce the head movement to 208

It may cause starvation

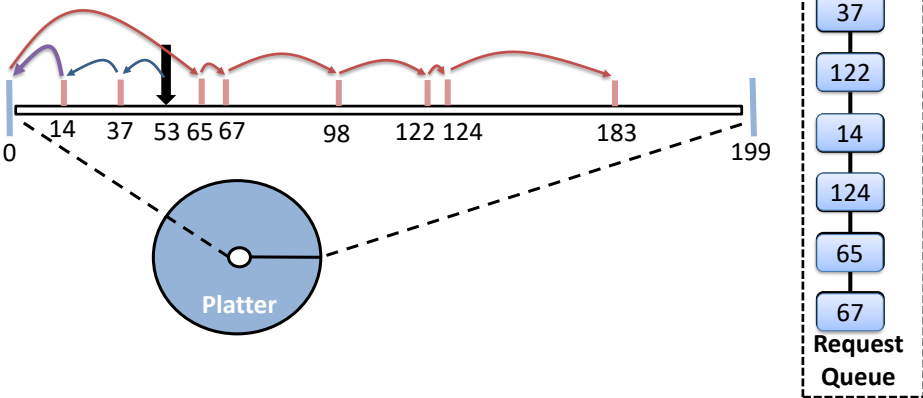
SCAN Scheduling

- Scan back and forth
 - Starts at one end, moves toward the other end
 - Service the requests as it reaches each cylinder
 - Reverse the direction
 - **Elevator algorithm**

SCAN Scheduling

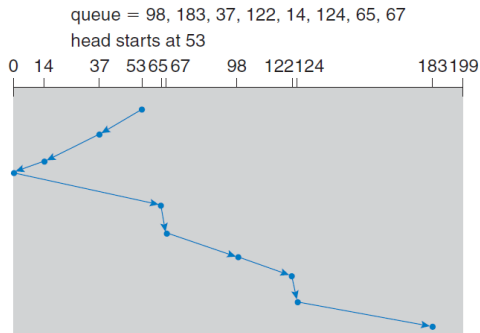
- Scan back and forth

Suppose the head is moving from 53 to 0



SCAN Scheduling

- Scheduling diagram



Any problem?

Assume a uniform request distribution

The heaviest density of requests is at the other end of the disk

They need to wait for a long time

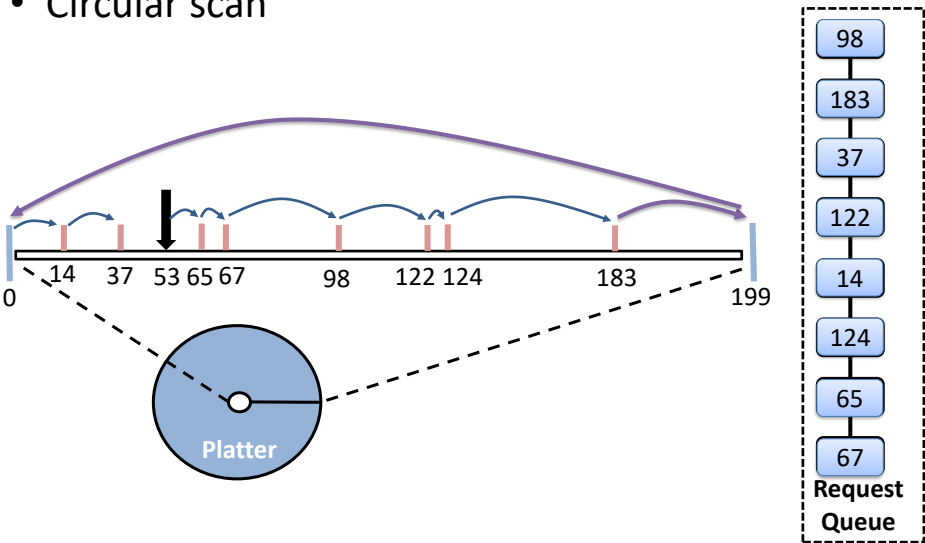
Can we do something about this?

C-SCAN Scheduling

- Circular Scan back and forth
 - A variant of SCAN: immediately return when reaches the end
 - Aim for providing a more uniform wait time

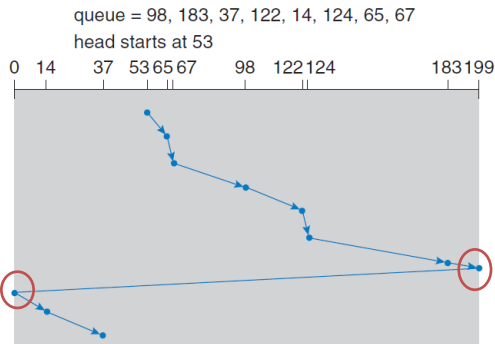
C-SCAN Scheduling

- Circular scan



C-SCAN Scheduling

- Scheduling diagram



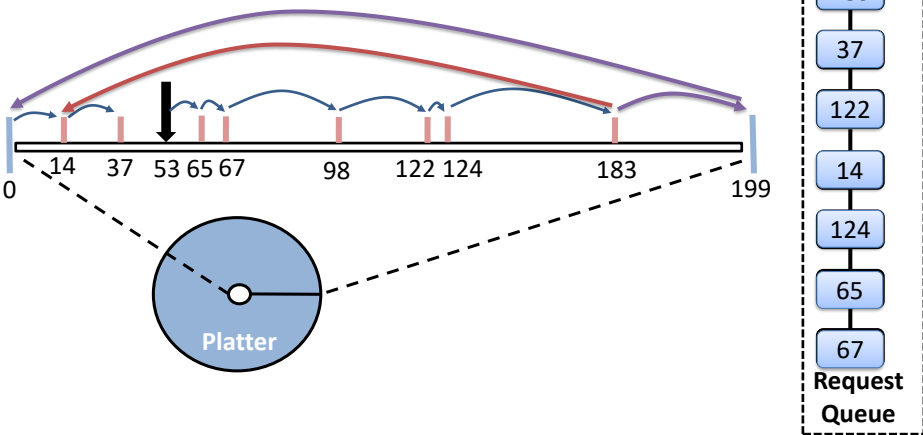
Unnecessary

No need to move across the full width of the disk, but only need to reach the final request

Improved SCAN and C-SCAN: LOOK and C-LOOK

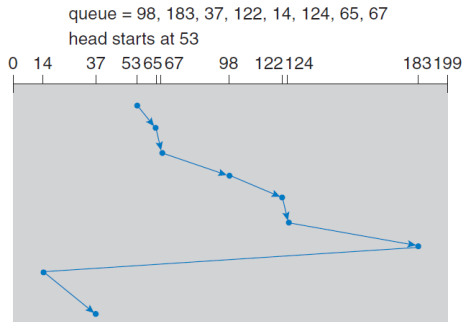
C-LOOK Scheduling

- Goes only as far as the final request
 - Look for a request before moving



C-LOOK Scheduling

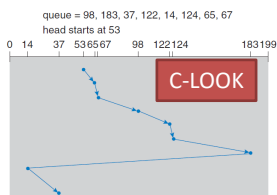
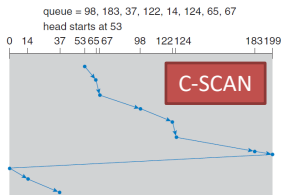
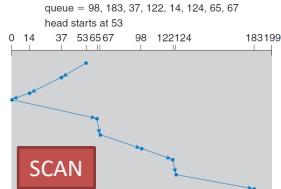
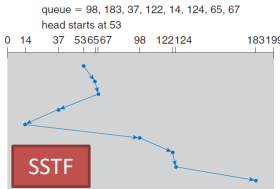
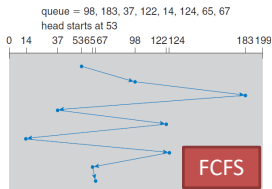
- Scheduling diagram



Look for a request before continuing to move in a given direction

Fewer head movements than SCAN/C-SCAN

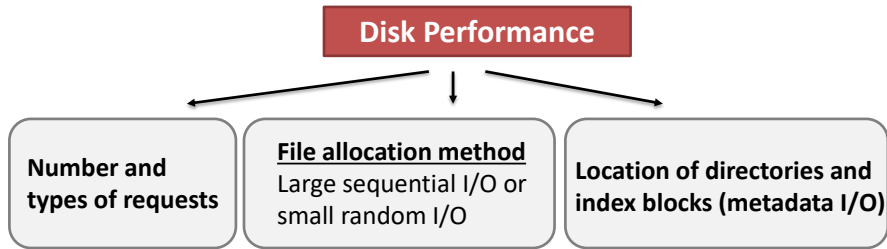
Summary of scheduling algorithms



SSTF outperforms FCFS, but may suffer from starvation

SCAN and C-SCAN perform better for heavy load systems, and they are less likely to cause starvation

Selection of a scheduling algorithm



Implementing scheduling in OS is necessary to satisfy other constraints (e.g., priority defined by OS)

Write disk scheduling as a separate module of the OS
Can be easily replaced with different alg. (default: SSTF/LOOK).

Topics

- Disk structure
- Disk scheduling
- **Solid-state drives (SSDs)**
- RAID
- Erasure coding



- **Solid-state drives (SSDs)**
 - **SSD architecture**
 - SSD operations
 - Flash translation layer



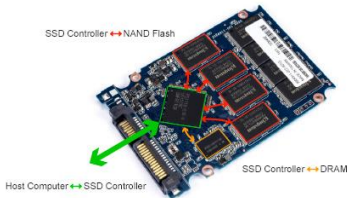
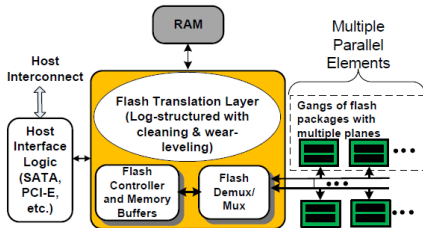
SSDs are widely used



Advantages of flash-based SSDs: non-volatility, shock resistance, high speed and low energy consumption;

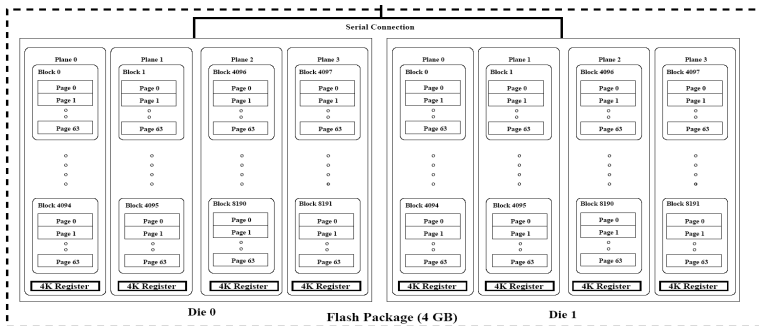
SSD Architecture

- SSD components
 - Multiple flash packages, controller, RAM



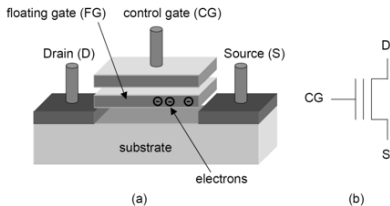
Flash Package

- Package > die/chip > plane > block > page



Samsung K9XXG08UXM (SLC) (2 dies, 4 planes, 2048 blocks, 64 pages)

Flash Cell



(a) Floating gate memory cell and (b) its schematic symbol

- Each cell stores one bit (or multiple bits)
- Program operation can only change the value from 1 to 0 (erase operation changes the value from 0 to 1)
 - **No overwritten**
- The floating gate becomes thinner as the cell undergoes more program-erase cycles
 - **Decreasing reliability**

Flash Types

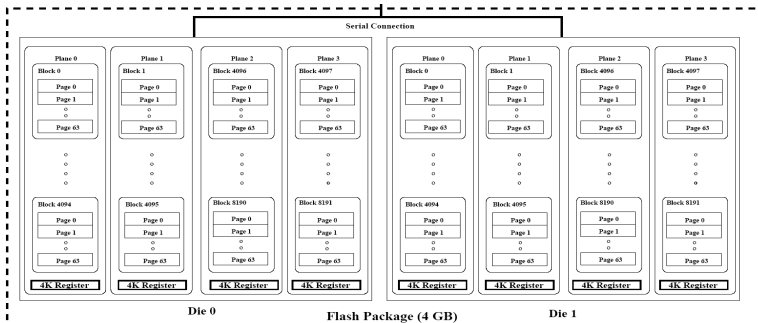
- NAND flash and NOR flash
 - NAND flash: denser capacity, only allow access in units of pages, faster erase operation
 - Most SSD products are based on NAND flash
- NAND flash: SLC and MLC
 - SLC: each cell stores one bit
 - Longer life time, lower access latency, higher cost
 - MLC: each cell stores two (or three) bits
 - Higher capacity

- **Solid-state drives (SSDs)**
 - SSD architecture
 - SSD operations
 - Flash translation layer



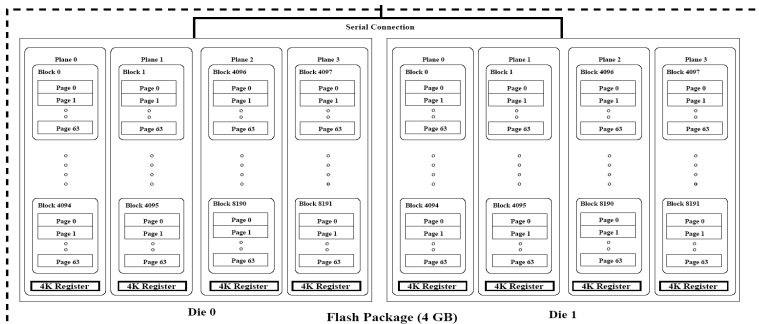
Read

- Read: in unit of pages (4KB)



Write

- Write: in unit of pages (4KB)



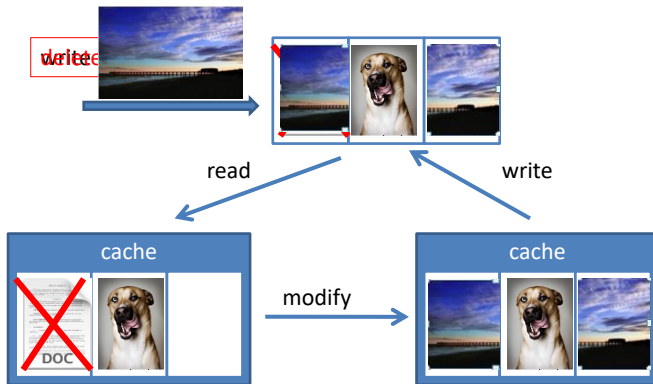
Erase

- Erase
 - In unit of blocks (64/128 pages)
 - Change all bits to 1
 - Much slower than read/write: 1.5ms
- Each block can only tolerate limited number of P/E cycles
 - SLC: 100K, MLC: 10K, TLC (several K to several hundred)
- The number of maximum P/E cycles decreases when
 - More bits are stored in one cell
 - The feature size of flash cell decreases (72nm, 34nm, 25nm)

Overwrite & Delete

- Delete
 - Simply mark the page as invalid
- Overwrite/update
 - Does not support in-place overwrite
 - Data can only be programmed to clean pages
- How about read-modify-write?

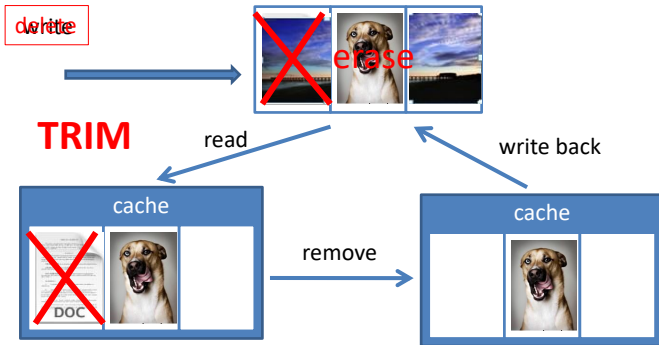
Read-Modify-Write



RMW may require a lot of read and write operations, so it is very slow

Trim

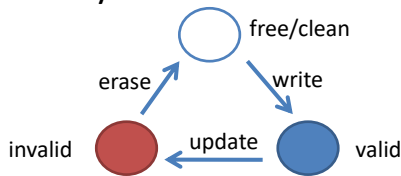
- Improve write performance degraded by RMW
 - The OS also sends a TRIM command to SSD after delete pages
 - Requires both OS and SSD to support



TRIM avoids slow RMW operation during write, so it increases write performance

Software layer in controller

- How to further improve write performance?
 - Address mapping is needed
- Page states
 - Garbage collection is also necessary
- Flash translation layer

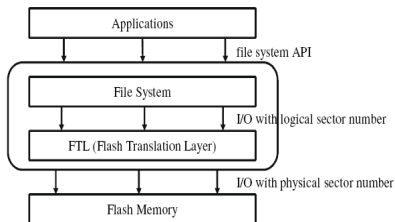


- **Solid-state drives (SSDs)**
 - SSD architecture
 - SSD operations
 - Flash translation layer
 -



Flash Translation Layer

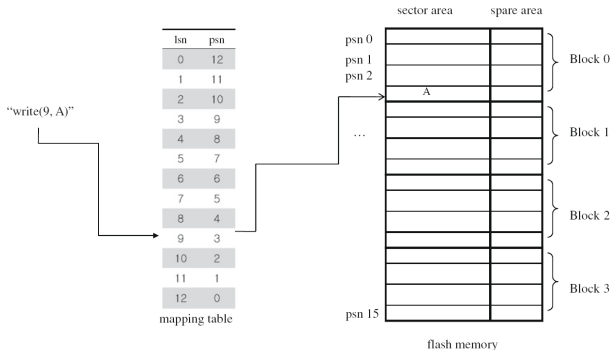
- Three functionalities
 - Address mapping
 - Garbage collection
 - Wear-leveling



Address Mapping

- Sector mapping
- Block mapping
- Hybrid mapping
- Log-structured mapping

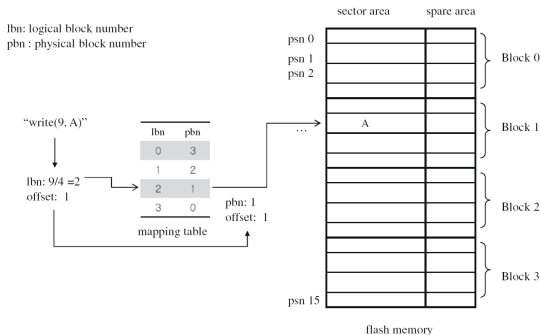
Sector Mapping



Mapping table is large: requires a large amount of RAM

Block Mapping

- The logical sector offset is the same with the physical sector offset

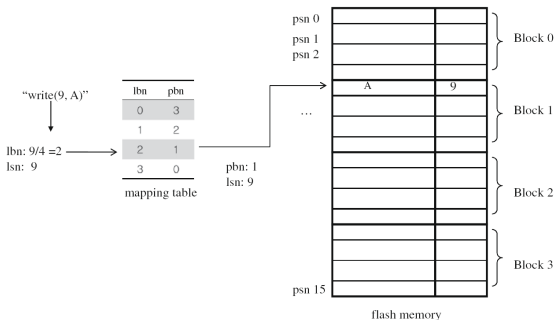


Smaller mapping table

If the FS issues writes with identical lsn, many erases

Hybrid Mapping

- First use block mapping, then use sector mapping in each block

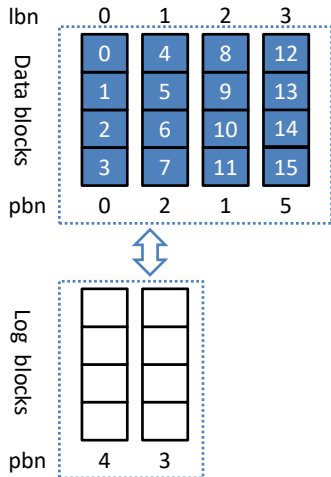


Small mapping table

Avoid a lot of erase operations

Longer time to identify the location of a page

Log-structured Mapping

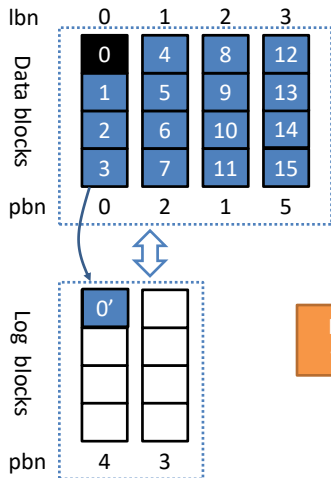


Data blocks: block mapping
Log blocks: sector mapping

In Flash

In RAM

Log-structured Mapping



Data blocks: block mapping
Log blocks: sector mapping

Multiple
variants

(lbn, pbn)

(0,0)
(1,2)
(2,1)
(3,5)

BMT

(lsn, (pbn, off))

(0, (4,0))

SMT

In Flash

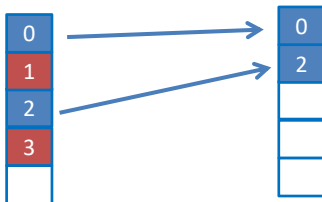
In RAM

Short summary

- The performance of address mapping is workload dependent
 - Block mapping is suitable for sequential workloads
 - Sector mapping is suitable for random workloads
 - Log-structured mapping is suitable for workloads with large sequential and small random requests
- **Tradeoff exists**

Garbage Collection

- Due to the existence of invalid pages, GC must be called to reclaim storage
 - Choose a candidate block
 - Write valid pages to another free block
 - Erase the original block

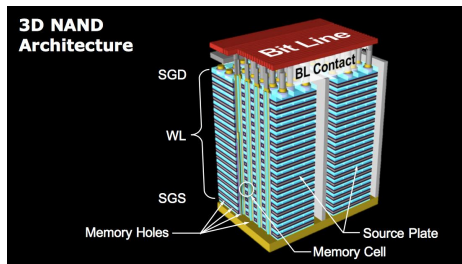


Design Issues of GC Algorithms

- **Tradeoff** in GC design
 - Efficiency: minimize writes
 - Wear-leveling: erase every block as even as possible
 - Tradeoff
 - GC is considered together with wear-leveling
- Algorithms
 - Greedy, random, and their variants
 - Hot/cold identification

Other Technologies

- 3D NAND flash



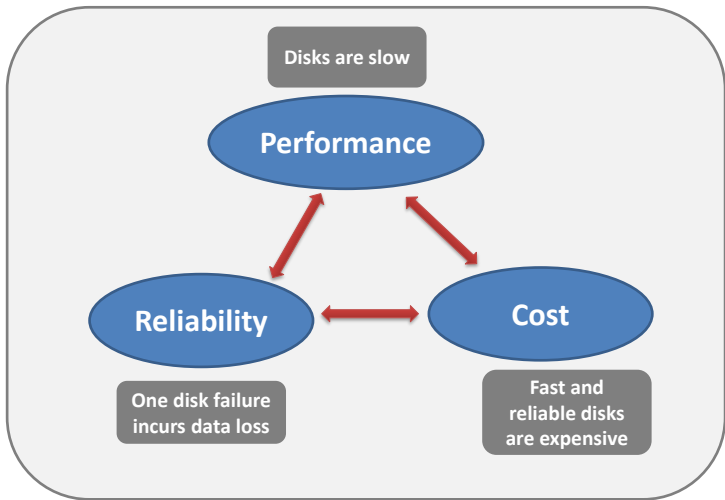
- Non-volatile memory (NVRAM)
 - PCM, STTRAM, ReRAM, etc...
 - Byte-addressable and non-volatile
 - 3D XPoint

Topics

- Disk structure
- Disk scheduling
- Solid-state drives (SSDs)
- **RAID & Erasure codes**
- Problems with EC



RAID Motivation

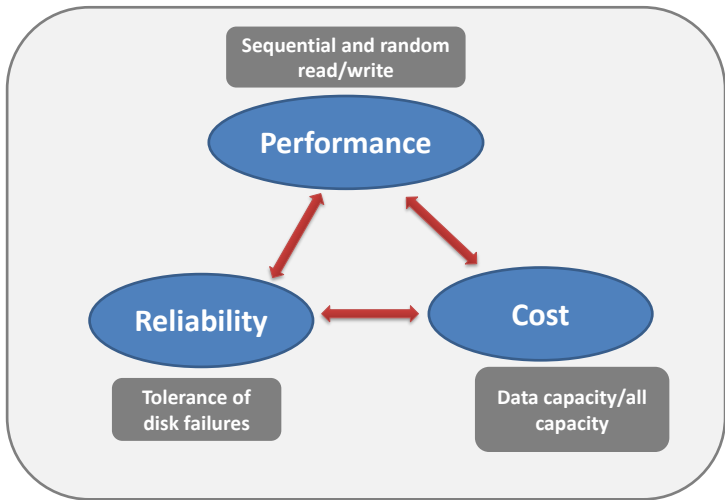


RAID Introduction

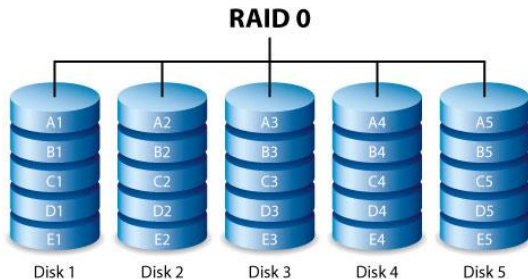
RAID: Redundant Array of Inexpensive (independent) Disks

- ✓ In the past
 - Combine small and cheap disks as a **cost-effective** alternative to large and expensive disks
- ✓ Nowadays
 - Higher performance
 - Higher reliability via redundant data
 - Larger storage capacity
- ✓ Many different levels of RAID systems
 - Different levels of redundancy, capacity, cost...

RAID Evaluation



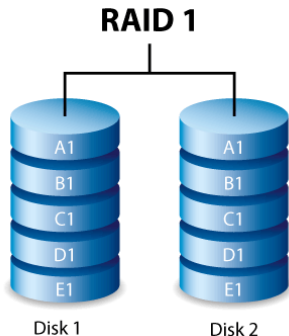
RAID 0



- Block-level striping, no redundancy
- Provides higher data-transfer rate
- Does not improve reliability. **Once a disk fails, data loss may happen (MTTF: mean time to failure)**

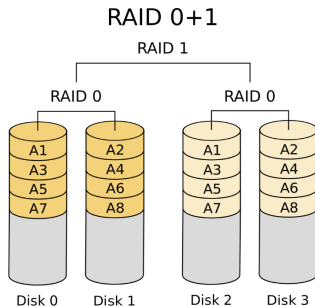
RAID 1

- How to improve reliability?
- Data mirroring (RAID1)
 - ✓ Two copies of the data are held on two physical disks, and the data is always identical.
 - ✓ Replication
- **High storage cost**
 - ✓ Twice as many disks are required to store the same data when compared to RAID 0.
 - ✓ Even worse storage efficiency with more copies



Combinations

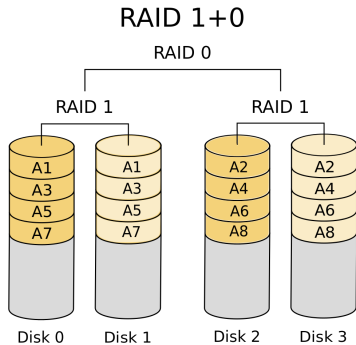
- RAID 0 provides reliability and RAID 1 provides reliability
- RAID 0+1 (RAID01)
 - ✓ First data striping
 - ✓ Then data mirroring



Same storage
cost as RAID 1

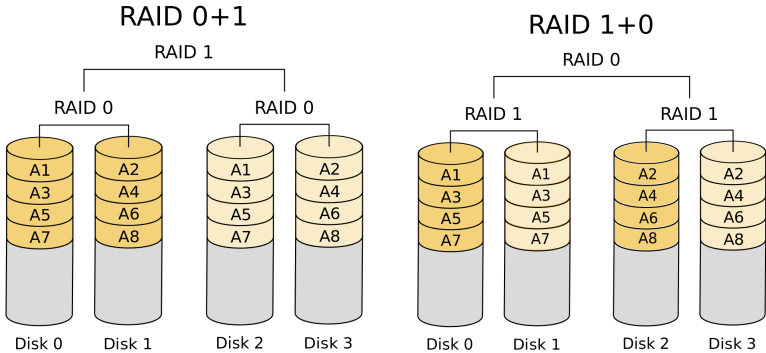
Combinations

- RAID 0 provides reliability and RAID 1 provides reliability
- RAID 0+1 (RAID01)
 - ✓ First data striping
 - ✓ Then data mirroring
- RAID 1+0 (RAID10)
 - ✓ First data mirroring
 - ✓ Then data striping



Same storage cost

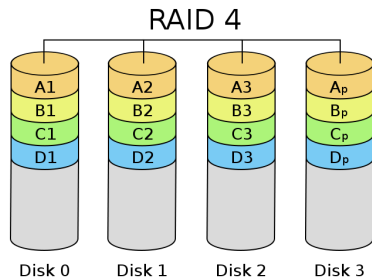
RAID01 vs RAID10



Both suffer from high storage cost

RAID 4

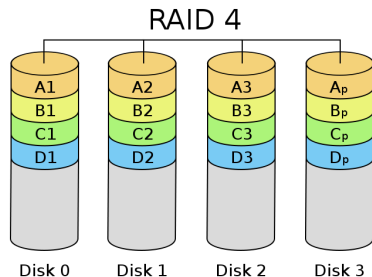
- Balance the tradeoff between reliability and storage cost?
 - Redundancy with parities
- **Parity generation:** Each parity block is the XOR value of the corresponding data disks
- **Block-level data striping**
 - Data and parity blocks are distributed across disks
 - Dedicated parity disk
- Any problem?



$$A_p = A1 \otimes A2 \otimes A3$$

How to update data

- Suppose A1 will be updated to A1'
 - Both A1 and A_p need to be updated
 - Read-modify-write (RMW)

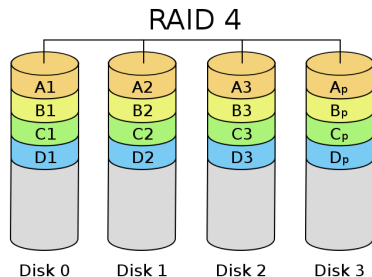


RMW: $A'_p = A_p \otimes A1 \otimes A1'$

$$A_p' = A1 \otimes A2 \otimes A3 \otimes A1 \otimes A1' \\ = A2 \otimes A3 \otimes A1'$$

How to update data

- Suppose A1 will be updated to A1'
 - Both A1 and A_p need to be updated
 - Read-modify-write (RMW)
- How about updating both A1 and A2 simultaneously?
 - RMW?
 - Read-reconstruct-write (RRW)
- Selection of RMW/RRW

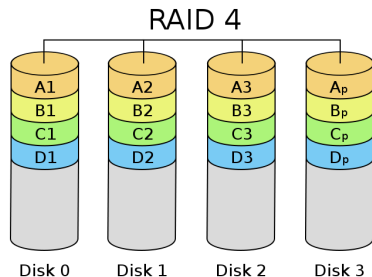


$$\text{RRW: } A'_p = A_3 \otimes A_1' \otimes A_2'$$

Both RMW and RRW incur extra reads and writes

Problems of RAID 4

- Problems of RAID 4
- Disk bandwidth are not fully utilized
 - Parity disk will not be accessed under normal mode
- Parity disk may become the bottleneck
 - E.g., updating A1, B2, C3

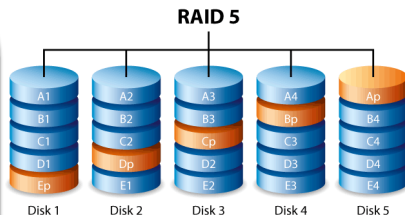


Read: A1, B2, C3, **Ap, Bp, Cp**

Write: A1' B2', C3', **Ap', Bp', Cp'**

RAID 5

- Similar to RAID 4
 - One parity per stripe
- Key difference
 - Uniform parity distribution
- RAID 5 is an ideal combination of
 - good performance
 - good fault tolerance
 - high capacity
 - storage efficiency



$$A_P = A_1 \oplus A_2 \oplus A_3 \oplus A_4$$

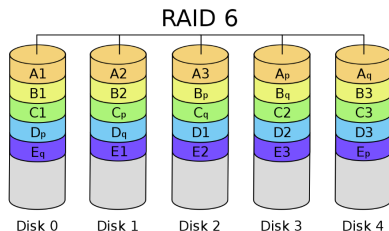
\vdots

$$E_P = E_1 \oplus E_2 \oplus E_3 \oplus E_4$$

Parity update overhead still exist

RAID 6

- How to tolerate more disk failures?
- RAID-6 protects against two disk failures by maintaining two parities
- Encoding/decoding operations:
 - Based on **Galois field**



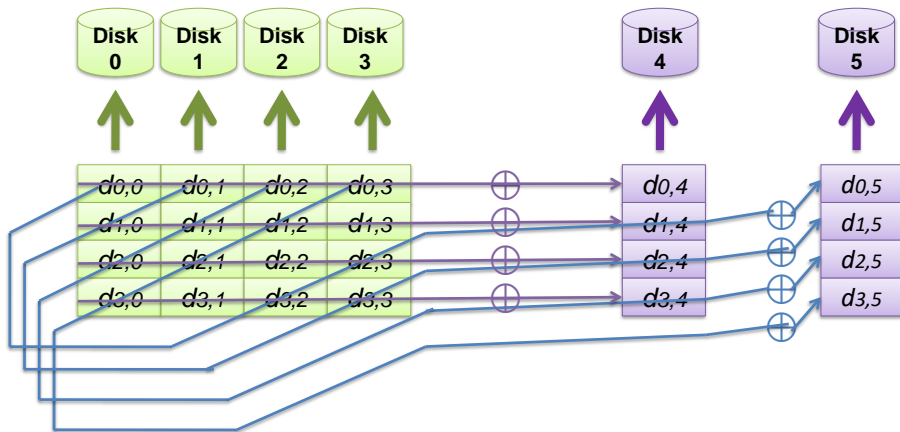
$$A_p = A_1 \oplus A_2 \oplus A_3 \oplus A_4$$

$$A_q = c^0 A_1 \oplus c^1 A_2 \oplus c^2 A_3 \oplus c^3 A_4$$

**Parity update overhead
becomes larger**

RDP Code

- An RDP code example with 6 disks

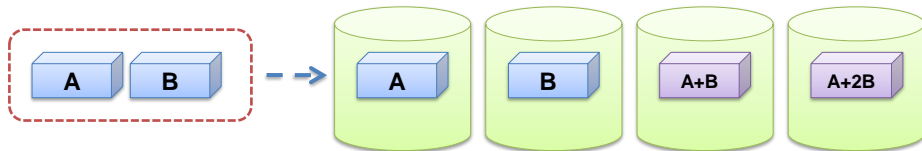


Erasure Codes

➤ Erasure codes

- Different redundancy levels
 - **2-fault tolerant**: RDP, EVENODD, X-Code
 - **3-fault tolerant**: STAR
 - **General-fault tolerant**: Cauchy Reed-Solomon (CRS)

➤ Generate **m** code blocks from **k** data blocks, so as to tolerate any **m** disk failures



Summary on Erasure Codes

- The motivation to introduce erasure codes in large-scale storage systems

The need to reduce the tremendous cost of storage

- In practice, erasure codes have seen widely deployment
 - Google File System [Ford, OSDI'10]
 - Windows Azure Storage [Huang, ATC'12]
 - Facebook [Borthakur, Hadoop User Group Meeting 2010]
 - ...

Topics

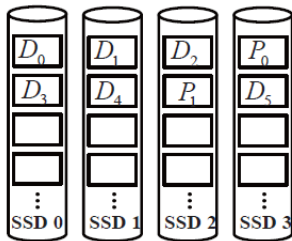
- Problems with RAID/EC
 - Optimizing parity updates
 - Recovery
 - Asynchronous coding
 - ...



SSD RAID

- RAID provides device-level fault tolerance
 - Each stripe contains data and parity

- Limitation: Parity updates
 - Update data -> update parity
 - Update D_1 to D_1'
 - RMW: $P'_0 = P_0 \oplus D_1 \oplus D_1'$
 - RRW: $P'_0 = D_0 \oplus D_1' \oplus D_2$
 - Extra I/Os and GC



Parity chunks:

$$P_0 = D_0 \oplus D_1 \oplus D_2$$

$$P_1 = D_3 \oplus D_4 \oplus D_5$$

- SSD RAID
 - Parity update influences both performance and endurance

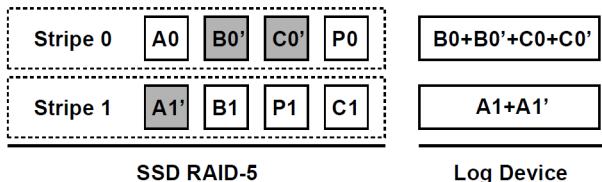
Design tradeoff

- Design trade-off in SSD RAID arrays
 - RAID improves reliability
 - Parity updates incur extra I/Os and GC operations
 - Degrade performance and endurance

How to address the parity update overhead?

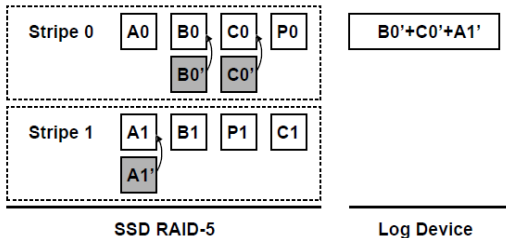
Parity Logging

- Original Parity logging
 - Incoming reqs: $\{A_0, B_0, C_0\}, \{A_1, B_1, C_1\}, \{B_0', C_0', A_1'\}$



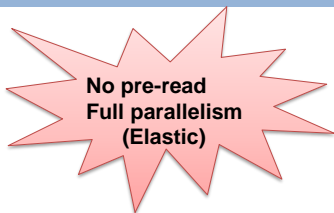
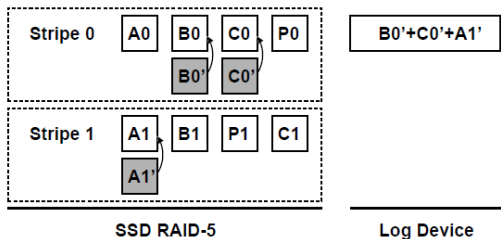
- Drawbacks
 - **Pre-read**: Extra reads
 - **Per-stripe basis**: Extra log chunks; Partial parallelism

Our solution: **New RAID Design via Elastic Parity Logging (EPLOG)**



**No pre-read
Full parallelism
(Elastic)**

EPLOG



- **Benefits of EPLOG**

- General RAID
- High endurance: Reduce parity writes to SSDs
- High performance: Reduce extra I/Os
- Low-cost deployment: Commodity hardware

- ✓ Yongkun Li, Helen H. W. Chan, Patrick P. C. Lee, and Yinlong Xu. "Elastic Parity Logging for SSD RAID Arrays." IEEE/IFIP DSN (Regular paper), Toulouse, France, June 2016.
- ✓ Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. "Elastic Parity Logging for SSD RAID Arrays: Design, Analysis, and Implementation." IEEE TPDS, volume: 29, issue: 10, Oct. 2018.

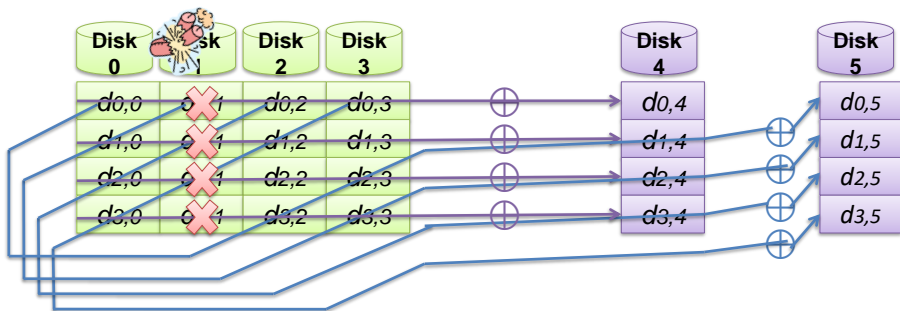
Topics

- **Problems with RAID/EC**
 - Optimizing parity updates
 - **Recovery**
 - Asynchronous coding
 - ...



Failure Recovery Problem

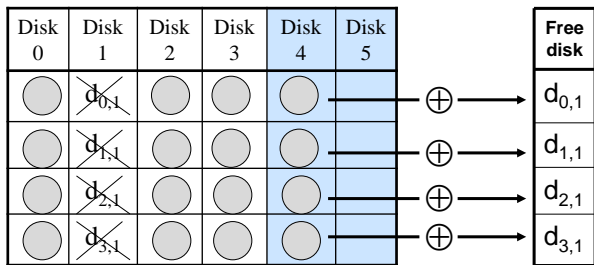
- Recovering disk failures is necessary
 - Preserve the required redundancy level
- Single-disk failure recovery
 - Single-disk failure occurs more frequently than a concurrent multi-disk failure



Suppose Disk 1 fails. How do we recover Disk 1 efficiently?

Optimize Recovery Performance

- Traditional method: only use row blocks for repair.







⑩ Example: $d_{0,1} = d_{0,0} \oplus d_{0,2} \oplus d_{0,3} \oplus d_{0,4}$





⑩ Need read $(p-1)^2 = 16$ blocks

Optimize Recovery Performance

- Recovery choices: row blocks or diagonal blocks

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
	$d_{0,1}$				
	$d_{1,1}$				
	$d_{2,1}$				
	$d_{3,1}$				

Repair $d_{0,1}$ from row blocks

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
	$d_{0,1}$				
	$d_{1,1}$				
	$d_{2,1}$				
	$d_{3,1}$				

Repair $d_{0,1}$ from diagonal blocks

Single Disk Failure Hybrid Recovery

- Recover Disk 1.

Failure blocks

Recover choices

$d_{0,1}$

diagonal

$d_{1,1}$













diagonal

$d_{2,1}$

row

$d_{3,1}$

row

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
	$d_{0,1}$				
	$d_{1,1}$				
	$d_{2,1}$				
	$d_{3,1}$				



Duplicate data block

- ❖ The four blocks are repeated twice.
- ❖ Result: Need read $16-4=12 < 16$ block for recovery.

Xiang, L., Xu, Y., Lui, J., Chang, Q. "Optimal recovery of single disk failure in RDP code storage systems". *ACM SIGMETRICS 2010*.

Hybrid Recovery

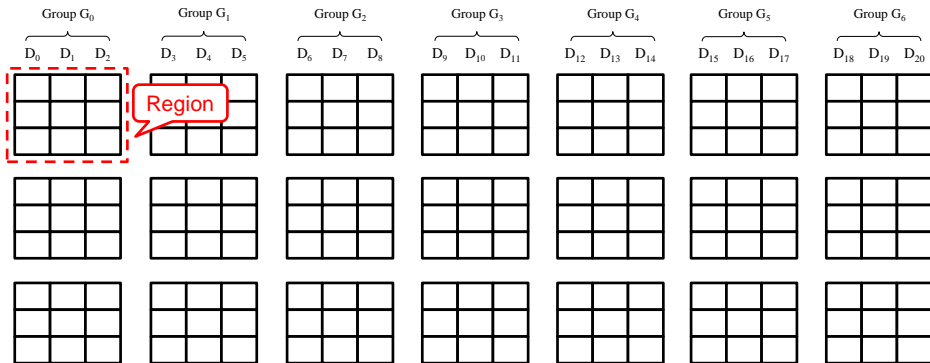
Previous approach leverages the code property,
but not change the code

Alternative approach

Can we design new codes which benefit the
recovery performance?

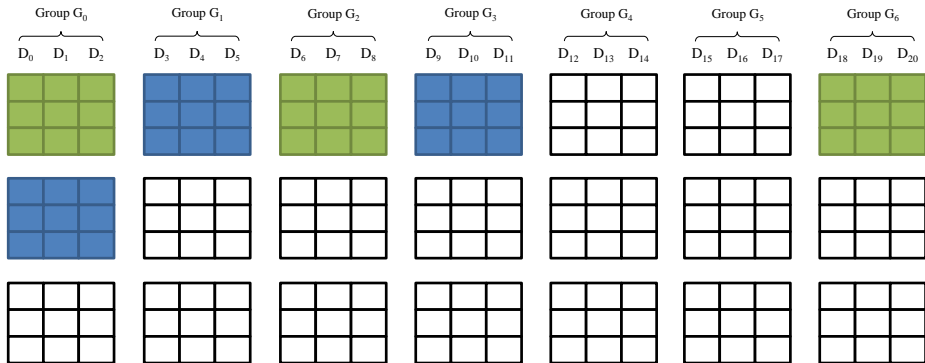
Yes! Our solution: OI-RAID

OI-RAID: An Example



- Divide 21 disks into 7 **groups**
- Divide each disk into 9 **storage units**
- Form a **region** with every 3×3 storage unit array in a group

OI-RAID: An Example



➤ Group regions into **region sets** based on BIBD

➤ (7,7,3,3,1)-BIBD:

Tuple T0: 0, 2, 6 Tuple T1: 0, 1, 3 Tuple T2: 1, 2, 4 Tuple T3: 2, 3, 5
 Tuple T4: 3, 4, 6 Tuple T5: 0, 4, 5 Tuple T6: 1, 5, 6

OI-RAID: An Example

Group G ₀			Group G ₁			Group G ₂			Group G ₃			Group G ₄			Group G ₅			Group G ₆		
D ₀	D ₁	D ₂	D ₃	D ₄	D ₅	D ₆	D ₇	D ₈	D ₉	D ₁₀	D ₁₁	D ₁₂	D ₁₃	D ₁₄	D ₁₅	D ₁₆	D ₁₇	D ₁₈	D ₁₉	D ₂₀
0	1	2	6	7	8	0	1	2	6	7	8	12	13	14	18	19	20	0	1	2
3	4	5	11	9	10	5	3	4	10	11	9	16	17	15	22	23	21	4	5	3
6	7	8	12	13	14	12	13	14	18	19	20	24	25	26	30	31	32	24	25	26
9	10	11	15	16	17	17	15	16	23	21	22	29	27	28	34	35	33	28	29	27
30	31	32	36	37	38	18	19	20	24	25	26	30	31	32	36	37	38	36	37	38
33	34	35	39	40	41	21	22	23	27	28	29	35	33	34	41	39	40	40	41	39

➤ **Inner layer code:**

- RAID5 within a region along the diagonal line


0	1	2
3	4	5

➤ **Outer layer code**

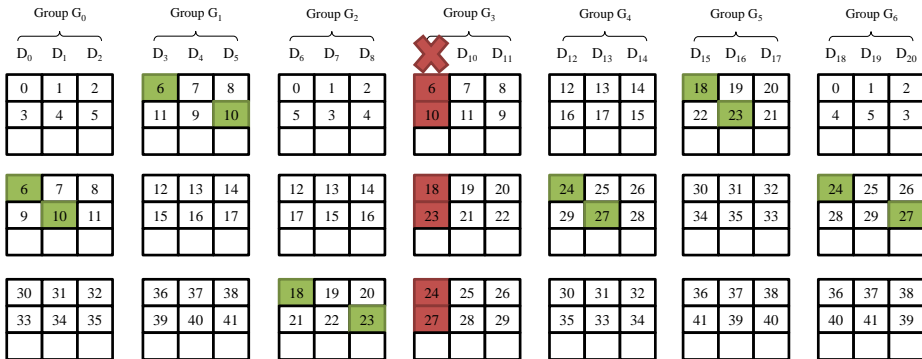
- RAID5 within a region set

The two layer code makes OI-RAID tolerate **three** arbitrary disk failures

Single Failure Recovery

Group G ₀			Group G ₁			Group G ₂			Group G ₃			Group G ₄			Group G ₅			Group G ₆		
D ₀	D ₁	D ₂	D ₃	D ₄	D ₅	D ₆	D ₇	D ₈	 D ₁₀	D ₁₁		D ₁₂	D ₁₃	D ₁₄	D ₁₅	D ₁₆	D ₁₇	D ₁₈	D ₁₉	D ₂₀
0	1	2	6	7	8	0	1	2	6	7	8	12	13	14	18	19	20	0	1	2
3	4	5	11	9	10	5	3	4	10	11	9	16	17	15	22	23	21	4	5	3
6	7	8	12	13	14	12	13	14	18	19	20	24	25	26	30	31	32	24	25	26
9	10	11	15	16	17	17	15	16	23	21	22	29	27	28	34	35	33	28	29	27
30	31	32	36	37	38	18	19	20	24	25	26	30	31	32	36	37	38	36	37	38
33	34	35	39	40	41	21	22	23	27	28	29	35	33	34	41	39	40	40	41	39

Single Failure Recovery



- To rebuild the 6 failed data units in disk D₉
 - OI-RAID reads only **one** unit from each surviving disk

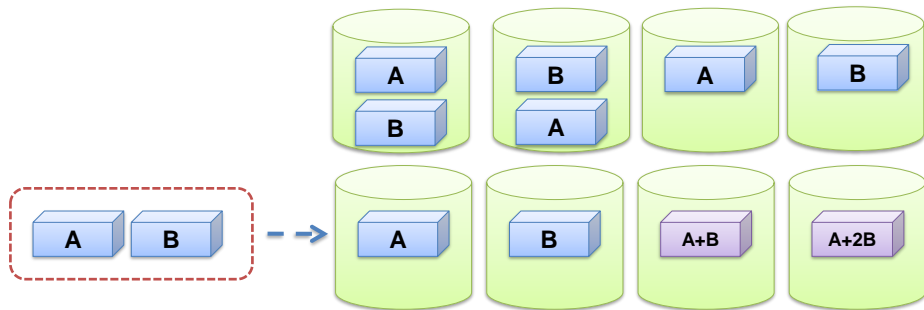
- ✓ Neng Wang, Yinlong Xu, Yongkun Li, and Si Wu. "OI-RAID: A Two-layer RAID Architecture Towards Fast Recovery and High Reliability." IEEE/IFIP **DSN**, Toulouse, France, June 2016.
- ✓ Yongkun Li, Neng Wang, Chengjin Tian, Si Wu, Yueming Zhang, Yinlong Xu. "A Hierarchical RAID Architecture Towards Fast Recovery and High Reliability." IEEE TPDS, 29(4) , pp. 734 - 747, April 2018.

Topics

- **Problems with RAID/EC**
 - Optimizing parity updates
 - Recovery
 - **Asynchronous coding**
 - ...

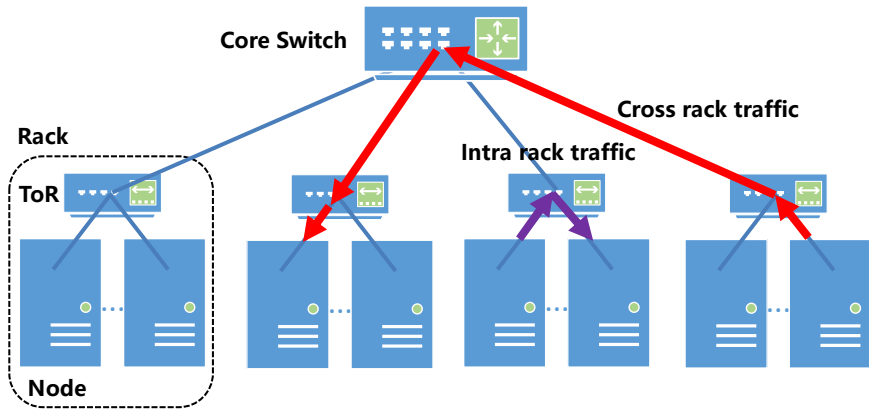


Replication vs. Erasure Coding



- Replication has better **read throughput** while erasure coding has smaller **storage overhead**
- **In practice:**
 - Data will be frequently accessed in a short time
 - Replication to erasure coding

Clustered File System

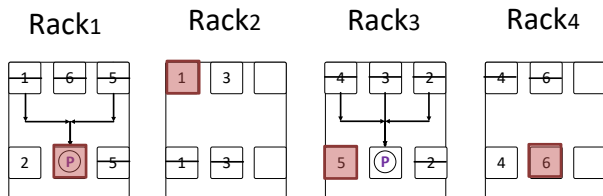


Cross-rack access typically takes *longer time*

Dynamic Stripe Construction

How to reduce/eliminate cross-rack traffic?

Our solution: Dynamic strip construction



- Encoding speed increases by up to 81%
- Improve frontend map-reduce tasks by 16.4%

Shuzhan Wei, **Yongkun Li***, Yinlong Xu, and Si Wu. "DSC: Dynamic Stripe Construction for Asynchronous Encoding in Clustered File System." IEEE **INFOCOM**, Atlanta, USA, May 2017.

Many Other Problems

- Recovery in heterogeneous systems
- Storage system scaling
- Hybrid system design
 - HDD+SSD
 - NVRAM
- Leveraging SSDs in various systems
- Data consistency
- ...

Summary of Ch8

Disk Structure

Disk Scheduling

- ✓ Cylinder, Track, Sector: CLV, CAV
- ✓ Access time
- ✓ FCFS, SSTF, SCAN/C-SCAN, LOOK/C-LOOK

SSD Structure

SSD Features/Issues

- ✓ Structure and features
- ✓ Operations (read/write/erase/GC)

RAID

Erasure Coding

Research Problems

- ✓ RAID structures (RAID0, 1, 4, 5, 6)
- ✓ Parity update

Operating Systems

Associate Prof. Yongkun Li

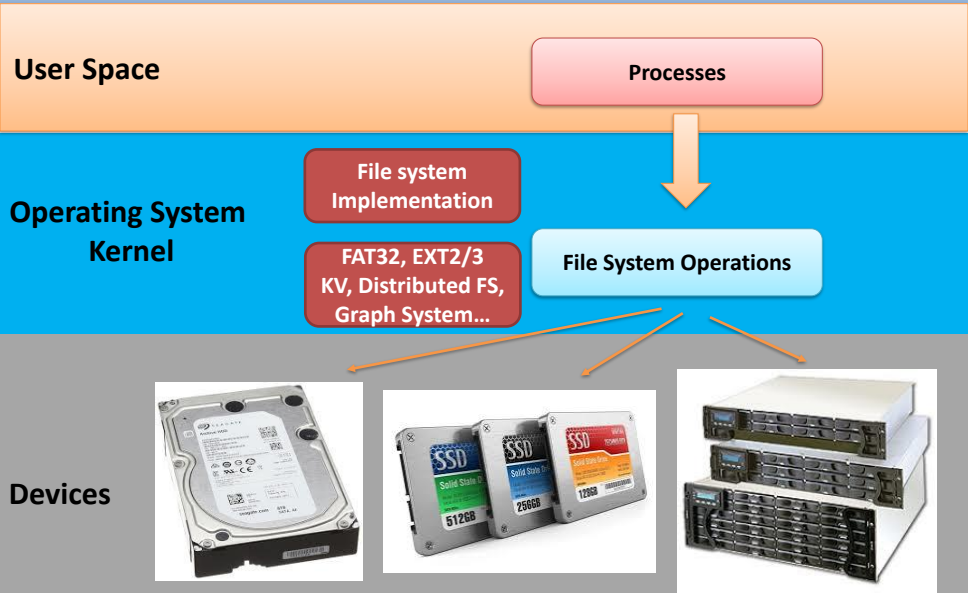
中科大-计算机学院 副教授

<http://staff.ustc.edu.cn/~ykli>

Chapter 9, part 1

File Systems – Programmer Perspective

Story so far...

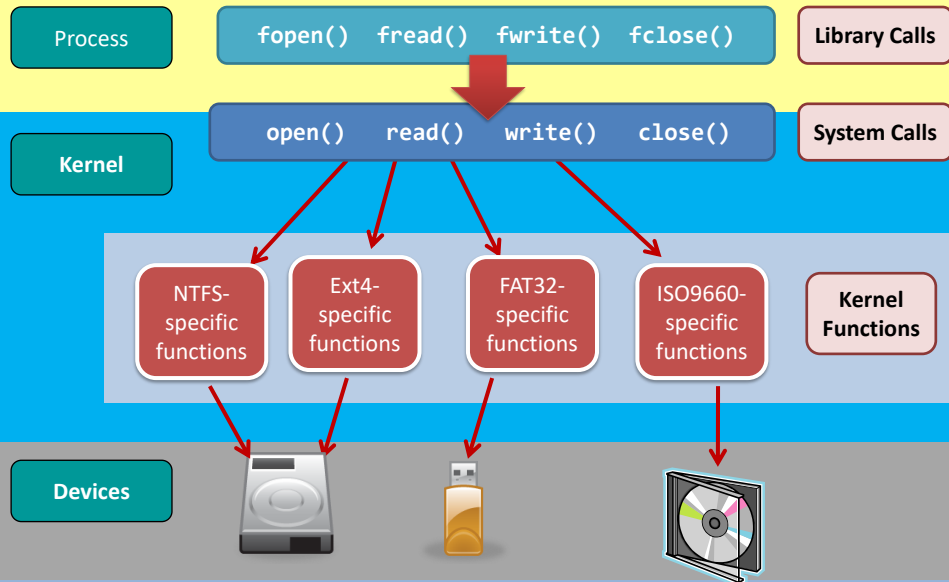


Outline

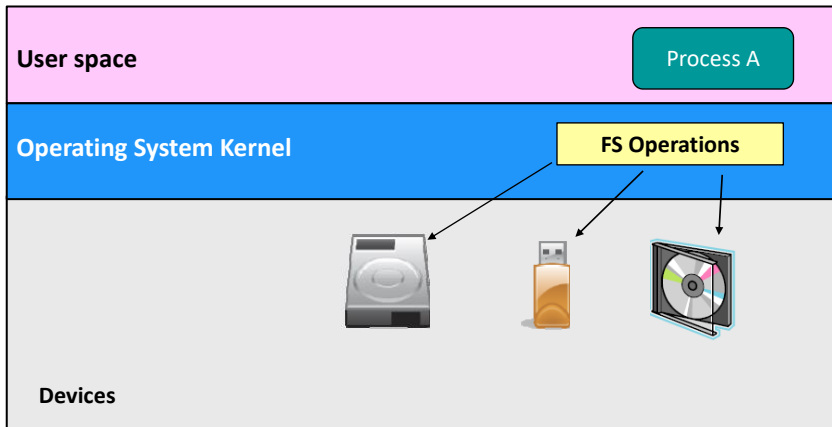
- File system introduction
- What are stored inside a storage device?
 - File
 - Directory
 - Interfaces/Operations
- How are the data stored?
 - File system layout

File system introduction

Introduction

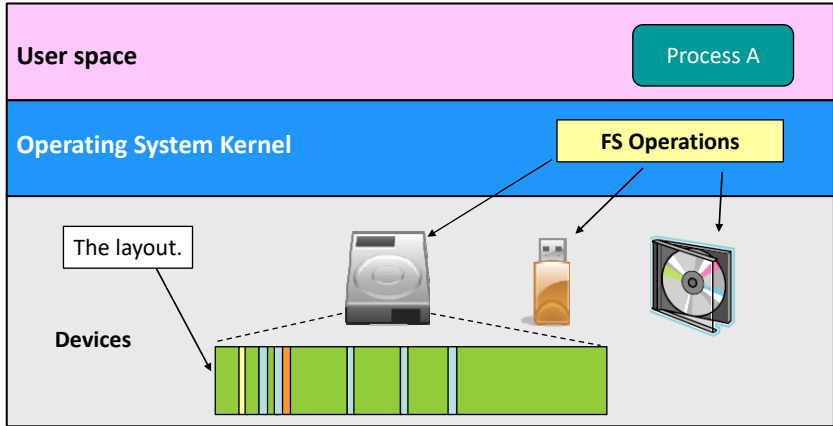


Introduction



- To understand what a file system (FS) is, we follow two different, but related directions:
 - **Layout & Operations.**

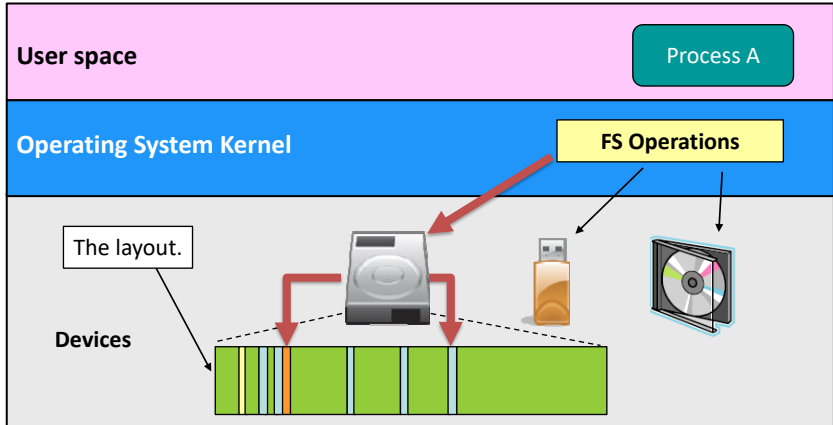
Introduction



Every FS has an unique layout on the storage device. The layout defines:

- **What** are the things stored in the device.
- **Where** the stored things are.

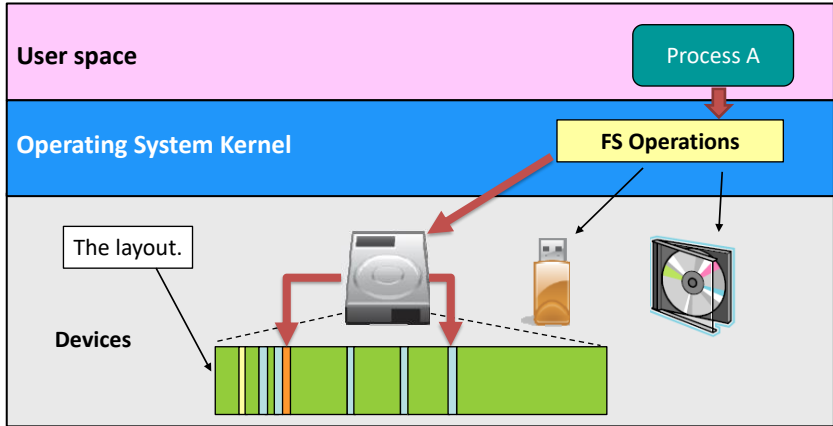
Introduction



The set of FS operations defines how the OS should work with the FS layout.

In other words, **OS knows the FS layout** and works with that layout.

Introduction



The process uses **system calls**, which then invoke the FS operations, to access the storage device.

Introduction

- Ask yourself:
 - OS = FS?
 - **Correct answer: OS \neq FS**
 - **An OS supports a FS**

- An OS can support more than one FS.
- A FS can be read by more than one OS.

Introduction

- Ask yourself:
 - Storage Device = FS?
 - **Correct answer: Storage Device \neq FS.**

- A FS must be stored on a device.
 - But, a device may or may not contain any FS.
 - Some storage devices can host more than one FS.

- A storage device is only a dummy container.
 - It doesn't know and doesn't need to know what FS-es are stored inside it.
 - The OS instructs the storage device how the data should be stored.

Outline of topics

- There are **two basic things** that are stored inside a storage device, and are common to all existing file systems.

What are they?

- They are **Files** and **Directories**.
- We will learn what they are and some basic operations of them.

Outline of topics

- There are **two basic things** that are stored inside a storage device, and are common to all existing file systems.

How does a FS store data into the disk?

- That is, the **layout** of file systems.
- The layout affects many things:
 - The speed in operating on the file systems;
 - The reliability in using the file systems;
 - The allocation and de-allocation of disk spaces.

Outline of topics

- Other topics

- We will look into the details of FAT32 and Ext2/3 file systems.
- Case studies: key-value systems, distributed file systems, graph storage systems

Part1: FS - Programmer Perspective

- File
- Operations
- Directory

- Why do we need files?
 - Storing information in memory is good because **memory is fast**.
 - However, memory vanishes after process termination.
- File provides a long-term information storage.
 - It is **persistent** and survives after process termination.
 - File is also a **shared object** for processes to access concurrently.

File

- What is a file?
 - A uniform logical view of stored information provided by OS.
 - **OS perspective**: A file is a logical storage unit (a sequence of logical records), it is an abstract data type
 - **User perspective**: the smallest allotment of logical secondary storage

- File type (executable, object, source code, text, multimedia, archive...)
- File attributes
- File operations

File – what are going to be stored?

- E.g., a text file.

```
h e l l o _ w o r l d '\n'
```

test.txt

What can we find out in this example?

Content?	Content of the file
Filename?	Content of its parent directory
File size?	Attribute of the file

When a file is named, it becomes independent of the process, the user, and even the system

File Attributes

- Typical file attributes

Name	Human-readable form
Identifier	Unique tag (a number which identifies the file within the FS)
Type	Text file, source file, executable file...
Location	Pointer to a device and to the location of the file on the device
Size	Number of bytes, words, or blocks
Time, date	Creation, last modification, last use...
Protection	Access control information (read/write/execute)

You can try the command “ls -l”

File Attributes

- Typical file attributes

Name	Human-readable form
Identifier	Unique tag (a number which identifies the file within the FS)
Type	Text file, source file, executable file...
Location	Pointer to a device and to the location of the file on the device
Size	Number of bytes, words, or blocks
Time, date	Creation, last modification, last use...
Protection	Access control information (read/write/execute)

Some new systems also support extended file attributes (e.g., checksum)

File Attributes

- File attributes are FS dependent.
 - Not OS dependent.

The design of FAT32 does not include any security ingredients.

Common Attributes	FAT32	NTFS	Ext2/3/4
Name	✓	✓	✓
Size	✓	✓	✓
Permission		✓	✓
Owner		✓	✓
Access, creation, modification time	✓	✓	✓

File Permissions

- E.g., in Unix system

-rw-rw-r--	1	pbg	staff	31200	Sep 3 08:30	intro.ps
drwx-----	5	pbg	staff	512	Jul 8 09:33	private/
drwxrwxr-x	2	pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2	jwg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1	pbg	staff	9423	Feb 24 2012	program.c
-rwxr-xr-x	1	pbg	staff	20471	Feb 24 2012	program
drwx--x--x	4	tag	faculty	512	Jul 31 10:31	lib/
drwx-----	3	pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3	pbg	staff	512	Jul 8 09:35	test/

First field: File/director

2nd /3rd /4th fields (3 bits each): controls read/write/execute for the file owner/file's group/others (e.g., 111:7,110:6)

What is the meaning of the permission 775/664?

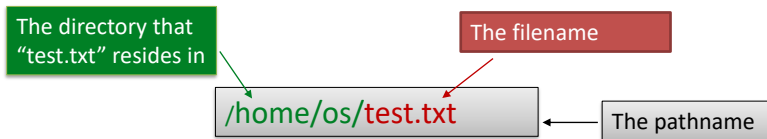
Writing attributes?

- Can you change those attributes directly?

Common Attributes	Way to change them?	
	Command?	Syscall?
Name	<code>mv</code>	<code>rename()</code>
Size	Too many tools to update files' contents	<code>write()</code> , <code>truncate()</code> , etc.
Permission	<code>chmod</code>	<code>chmod()</code>
Owner	<code>chown</code>	<code>chown()</code>
Access, creation, modification time	<code>touch</code>	<code>utime()</code>

Pathname vs Filename

- A file can be referred to by its name, then how to achieve this?



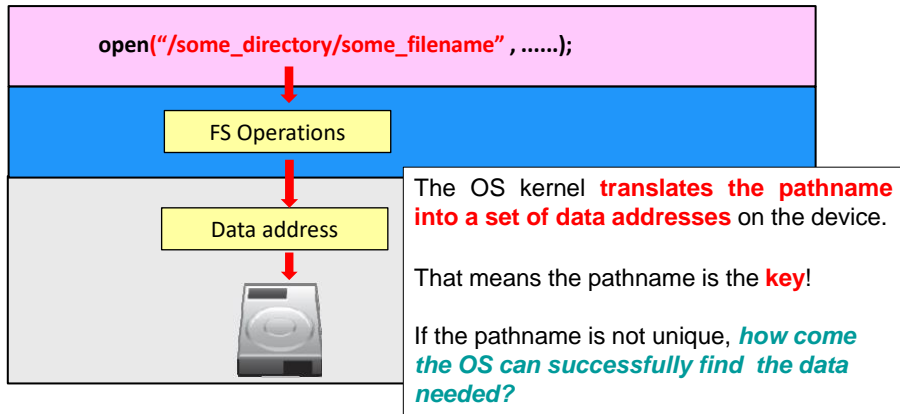
The pathname is **unique** within the entire file system.

The filename is **not unique** within the entire file system.

The filename is only **unique within the directory** that it resides.

Pathname vs Filename

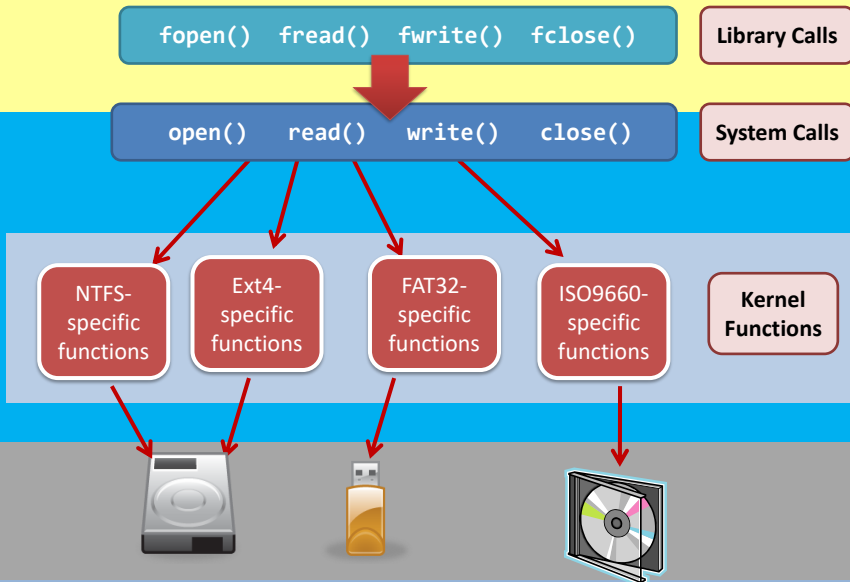
- Why do we need to consider **uniqueness**?



Part1: FS - Programmer Perspective

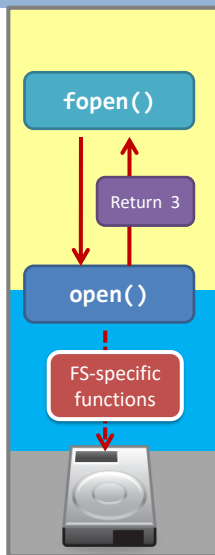
- File
- Operations
- Directory

Overview



File Open – Example


- What is **fopen()**?
 - First thing first, **fopen()** calls **open()**.
 - `FILE *fopen(const char *filename, const char *mode)`
- What is the type “**FILE**”?
 - “**FILE**”: a structure defined in “**stdio.h**”.
 - **fopen()** creates memory for the “**FILE**” structure.
 - Fact: occupying space in the area of dynamically allocated memory, i.e., `malloc()`



What is inside the “**FILE**” structure?

- There is a lot of helpful data in **FILE**:
 - Two important things: the **file descriptor** and **a buffer**!

```
int main(void) {  
    printf("fd of stdin  = %d\n", fileno(stdin) );  
    printf("fd of stdout = %d\n", fileno(stdout) );  
    printf("fd of stderr = %d\n", fileno(stderr) );  
}
```



fileno() returns the file descriptor of the FILE structure.

The type of **stdin**, **stdout**, and **stderr** is “**FILE ***”

```
$ ./fileno  
fd of stdin  = 0  
fd of stdout = 1  
fd of stderr = 2  
$ _
```

File operations

- The operating system should provide...

Create

Allocate space, add an entry in the directory

Write

Filename, file content (write pointer)

Read

Filename, mem location (read pointer)

Reposition

File seek (not involve actual I/O), **required for random accesses**

Delete

Release space, and erase directory entry

Truncate

Keeps attributes only

File operations

- Many operations involve searching the directory for locating the file (read/write/reposition...) – Can we avoid this content searching???

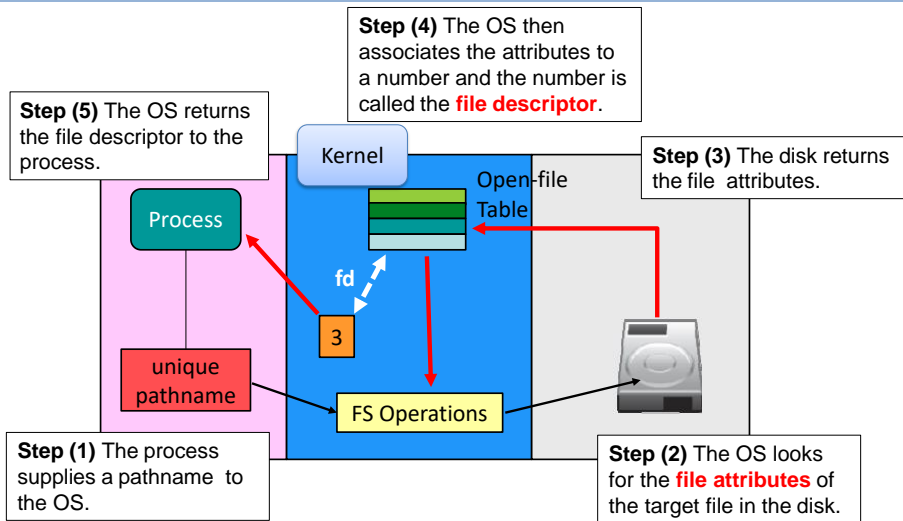
Open-file table

An `open()` system call is provided, and it is called before a file is first used

OS keeps a table containing information about all open files (per-process and system-wide table)

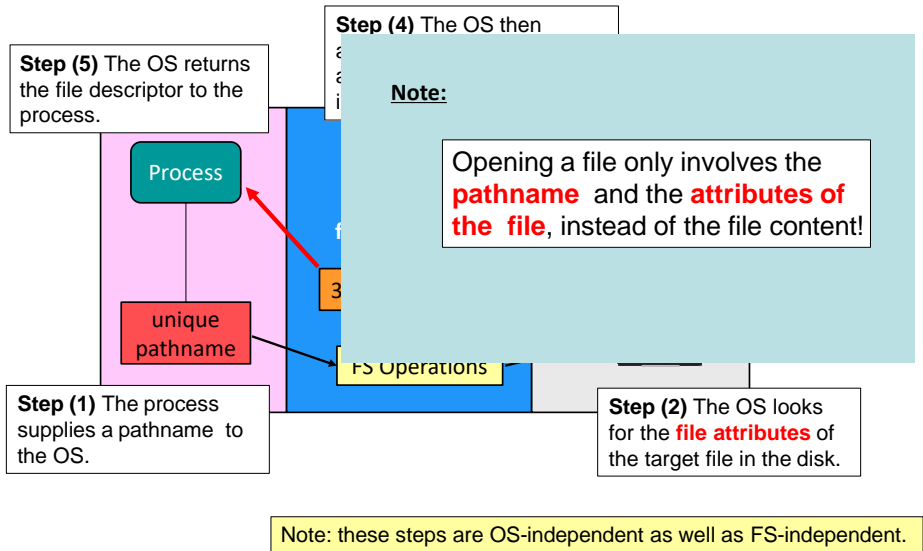
The file will be closed when it is no longer being actively used, using `close()` system call

The Truth of Opening a File



Note: these steps are OS-independent as well as FS-independent.

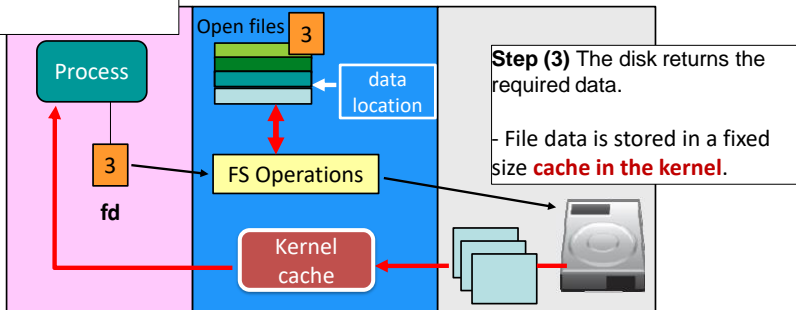
The Truth of Opening a File



How to read from open files

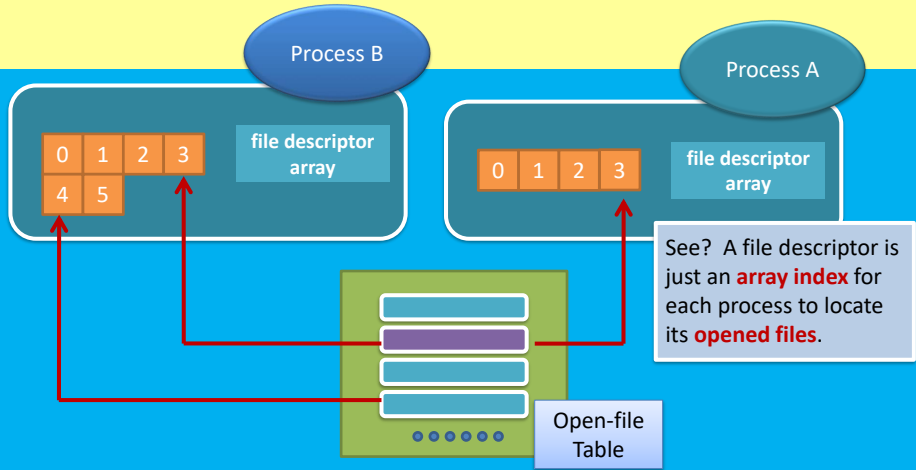
Step (1) The process supplies a file descriptor to the OS.

Step (2) The OS reads the file attributes and uses the stored attributes to **locate the required data**.



Step (4) The OS fills the **buffer provided by the process** with the data. **Write data to the userspace buffer**.

What is a file descriptor?



Although a file is opened by two different processes, the kernel uses **one structure to maintain it!**



How about read and write (**read()**
and **write()** system calls)?

read() & write()

Library calls that eventually invoke the read() system call	Library calls that eventually invoke the write() system call
scanf(), fscanf()	printf(), fprintf()
getchar(), fgetc()	putchar(), fputc()
gets(), fgets()	puts(), fputs()
fread()	fwrite()

- You know, I/O-related calls will invoke system calls.

```
int read ( int fd, void *buffer, int bytes_to_read )
```

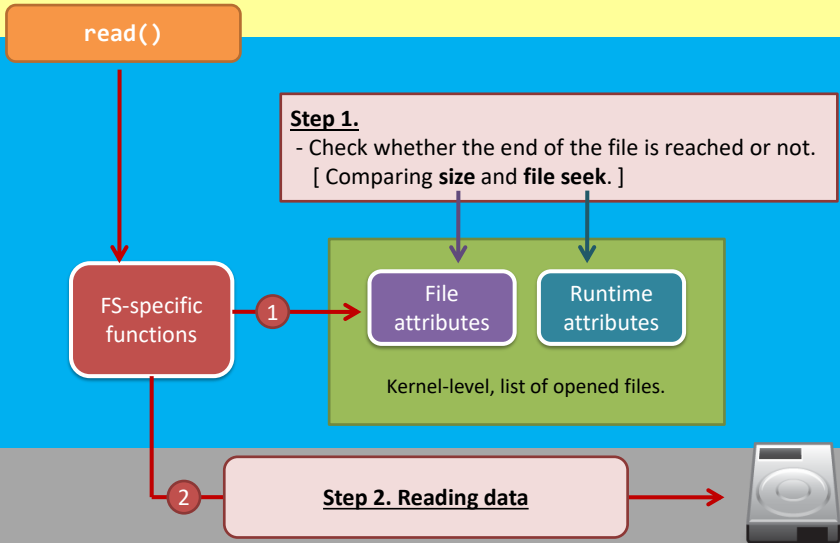
From file to buffer.

```
int write ( int fd, void *buffer, int bytes_to_write )
```

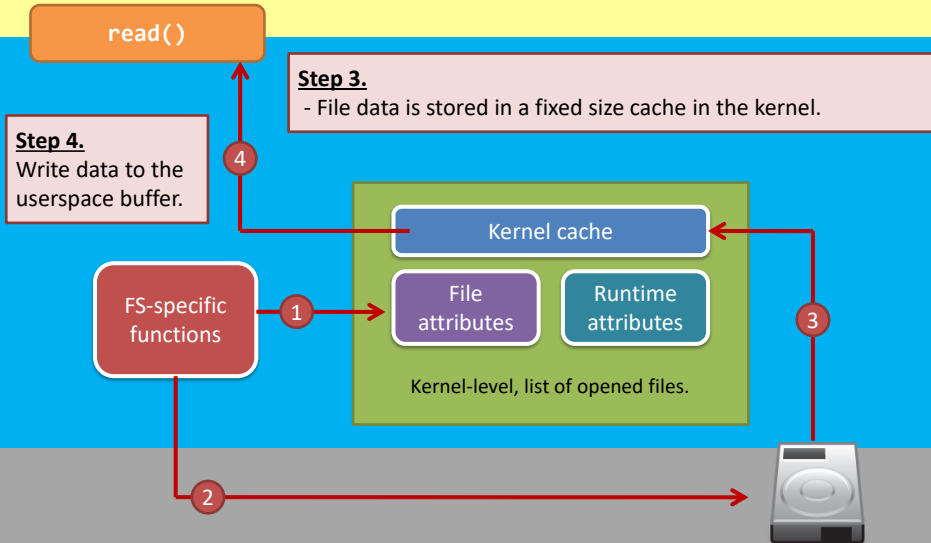
From buffer to file.

Note: I modified the function prototypes.

read() system call



read() system call



write() system call

write()

Step 1.

Write data to the kernel buffer.

1

Step 3.

The call returns.

3

2

Kernel cache

2

File
attributes

Runtime
attributes

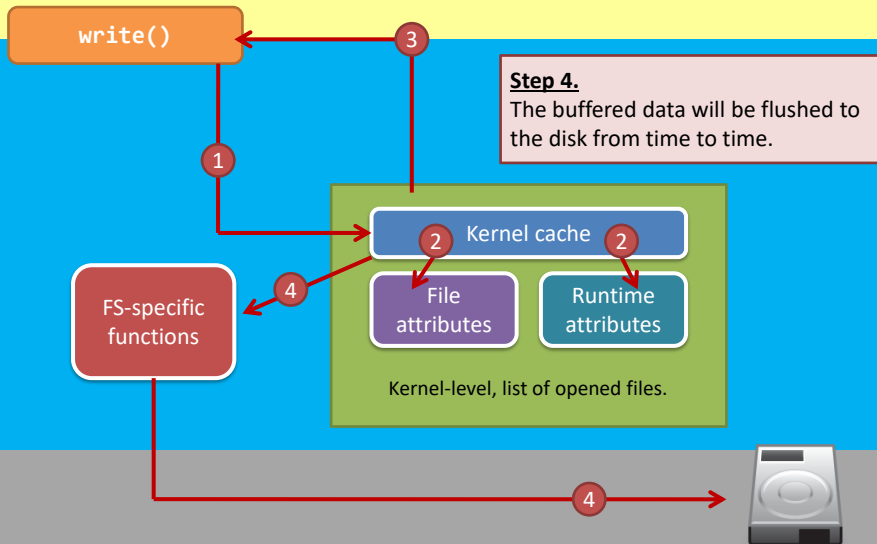
Kernel-level, list of opened files.

Step 2.

According to the data length,
(1) change in file size, if any, and
(2) change in the file seek.



write() system call

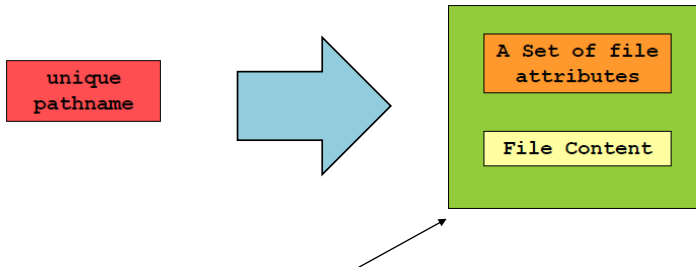


The kernel buffer cache implies...

- Performance
 - Increase reading performance?
 - Increase writing performance?
- Problem
 - Can you answer me why **you cannot press the reset button?**
 - Can you answer me **why you need to press the “eject” button before removing USB drives?**

Short Summary

- Every file has its unique pathname.
 - Its pathname leads you to its attributes and the file content.



A file has **two** important components! Plus, there are usually stored **separately**.

Short Summary

- We only introduce the read/write flow:
 - File writing involves **disk space allocation**; but...
 - The allocation of disk space is highly related to the design of the layout of the FS.
 - Also, the same case for the de-allocation of the disk space...

Part1: FS - Programmer Perspective

- File
- Operations
- **Directory**

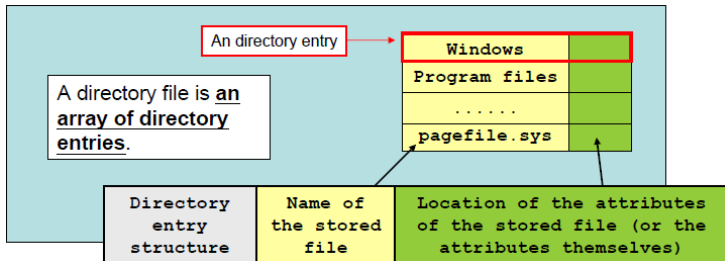
Directory

- A directory is **a file**.
 - Then, does it imply that it has **file attributes** and **file content**?

Answer: Sure

Answer: FS dependent

- How does a directory file look like?



Directory Traversal Process

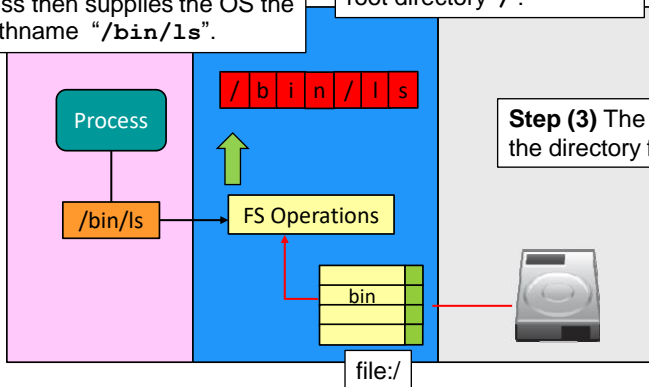
- How to locate a file using pathname?

Step (1) Suppose that the process wants to open the file `"/bin/ls"`.

The process then supplies the OS the unique pathname `"/bin/ls"`.

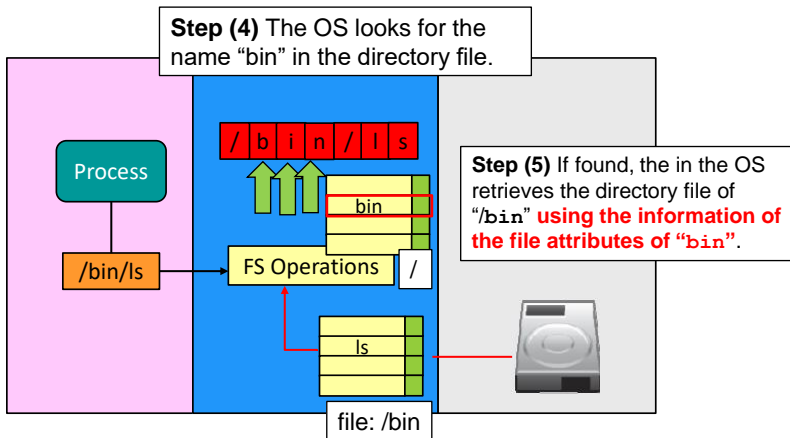
Step (2) The OS retrieves the directory file of the root directory `'/'`.

Step (3) The disk returns the directory file.



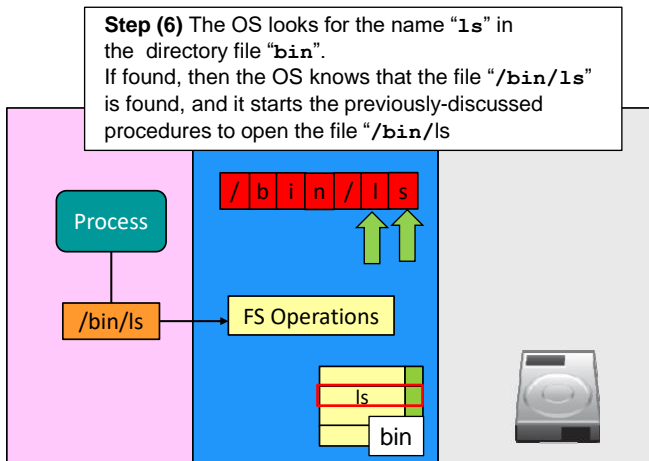
Directory Traversal Process

- How to locate a file using pathname?



Directory Traversal Process

- How to locate a file using pathname?



Short Summary

- A directory file records all the files including directories that are belonging to it.
 - So, do you understand “/bin/ls” now?
 - **Locate the directory file of the target directory** and to print contents out.
- Locating a file requires the **directory traversal process**:
 - open a file;
 - listing the content of a directory.

File Creation and Directory

- According to your experience, **what is the file creation?**
 - E.g., creating a file named “`test.txt`”?
 - “`touch test.txt`”?
 - “`vim test.txt`”, then type “`:wq`”?
 - “`cp [some filename] test.txt`”?
- The truth is:
File creation == Update of the directory file

File Creation and Directory

- If I type “**touch text.txt**” and “**text.txt**” does not exist, what will happen to the Directory file?

Note: “**touch text.txt**” will only create the directory entry, and there is **no allocation for the file content**.

Directory file: “/home/os”

score_sheet.xls	
midterm_marks.xls	
final_exam_paper.pdf	
.....	



score_sheet.xls	
midterm_marks.xls	
final_exam_paper.pdf	
.....	
text.txt	

A new directory entry is created.

File Deletion and Directory

- **Removing** a file is the reverse of the creation process.
 - Note that we are not ready to talk about de-allocation of the file content yet.

Directory file: `"/home/os"`

score_sheet.xls	
midterm_marks.xls	
final_exam_paper.pdf	
.....	
text.txt	



score_sheet.xls	
midterm_marks.xls	
final_exam_paper.pdf	
.....	

Updating directory file

- When/how to update a directory file?

Creating a directory file	syscall - <code>mkdir()</code> ; Example program - <code>mkdir</code> .
Add an entry to the directory file	syscall - <code>open()</code> , <code>creat()</code> ; Example program - <code>cp</code> , <code>mv</code> , etc.
Remove an entry to the directory file	syscall - <code>unlink()</code> ; Example program - <code>rm</code> .
Remove a directory file	syscall - <code>rmdir()</code> ; Example program - <code>rmdir</code> .

Summary of part 1

- In this part, we have an introduction to FS
 - File and directory
 - The truth about the calls that we usually use,
 - We learned: The **content** of a file is not the only entity, but also the file **attributes**.
- In the next part, we will go into the disk:
 - How and where to store the file attributes?
 - How and where to store the data?
 - How to manage a disk?

Operating Systems

Associate Prof. Yongkun Li

中科大-计算机学院 副教授

<http://staff.ustc.edu.cn/~ykli>

Chapter 9, part2 File System Layout

Outline

operations

Questions.

- Can I read back what I've written?
- Can I get back free space when I remove a file?
- How much space is consumed when I create a 1GB file?



You're given a disk of 1TB space. How to utilize it?

File content &
attributes

Directory

Allocated
Space

Free
Space

Things need to be stored.

Outline

- We briefly introduce the evolution of the file system layout:
 - From a dummy way to advanced ways.
 - The pros and cons are covered.
- We begin to look at some details of the FAT file system and EXT file system

How to store data?

- Consider the following case:
 - You are going to design the layout of a FS.
 - You are given the freedom to choose the locations to store files, including directory files.
 - How will you organize the data?



0

100GB

How to store data?

- Some (basic) rules are required:
 - Every data written to the device must be able to be retrieved.
 - Would you use the FS that will lose data randomly?
 - Every FS operation should be done as efficient as possible.
 - Would you use the FS if it takes a minute to retrieve several bytes of data?
 - When a file is removed, the FS should free the corresponding space.
 - Would you use the FS if it cannot free any occupied space?



0

100GB

File System Layout

Trial 1.0

The Contiguous Allocation

Trial 1.0 – the basics

- Just like a book!



Table of content

Chapter 1	p.1
Chapter 2	p.2
Chapter 3	p.10

Book VS Trial #1

Book	Trial #1
Chapter	Filename
Starting Page	Starting Address
NIL	Ending Address



Trial 1.0 – the basics

- Just like a book!

Suppose we have 3 files to store

rock.mp3
sweet.jpg
same.exe

We do not consider the directory structure at this moment

Book VS Trial #1	
Book	Trial #1
Chapter	Filename
Starting Page	Starting Address
NIL	Ending Address

Like a book, we need to some space to store the **table of content**, which records the filename and the (starting and ending) addresses of the file content.



Trial 1.0 – the basics

- Just like a book!

The table of content!

Book VS Trial #1	
Book	Trial #1
Chapter	Filename
Starting Page	Starting Address
NIL	Ending Address

Filename	Starting Address	Ending Address
rock.mp3	0	2000
sweet.jpg	2001	3456
game.exe	5000	5678

File attributes



Trial 1.0 – the basics

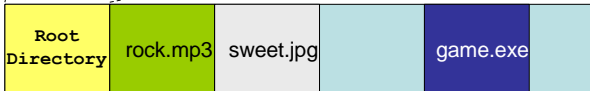
- Just like a book!

The table of content!

Contiguous allocation is very similar to the way we write a book. It starts with the table of content, which we call the root directory.

Filename	Starting Address	Ending Address
rock.mp3	0	2000
sweet.jpg	2001	3456
game.exe	5000	5678

File attributes



Trial 1.0 – the basics

You can locate files easily (with a directory structure).

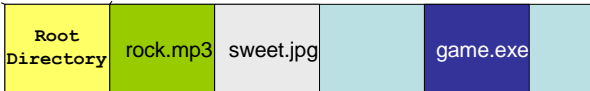
But, can you locate the **allocated space** and the **free space** in a short period of time?

Filename	Starting Address	Ending Address
rock.mp3	0	2000
sweet.jpg	2001	3456
game.exe	5000	5678

Free space is here.

But, it needs an $O(n)$ search, where n is the total number of files.

What if the disk is large and the files are small?



Trial 1.0 – the basics

File deletion is easy! Space de-allocation is the same as updating the root directory!

Yet, how about file creation?



Filename	Starting Address	Ending Address
rock.mp3	0	2000
sweet.jpg	2001	3456
game.exe	5000	5678



Filename	Starting Address	Ending Address
rock.mp3	0	2000
game.exe	5000	5678



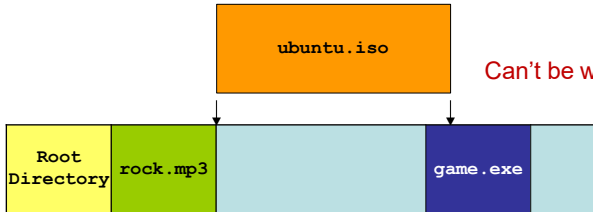
Trial 1.0 – the bad #1

- Suppose we need to write a new, but large file?

Really BAD! We have enough space, but there is no holes that I can satisfy the request. The name of the problem is called:

External Fragmentation

Any solution?



Trial 1.0 – the bad #1

- The **defragmentation process** may help.

Filename	Starting Address	Ending Address
rock.mp3	0	2000
game.exe	5000	5678



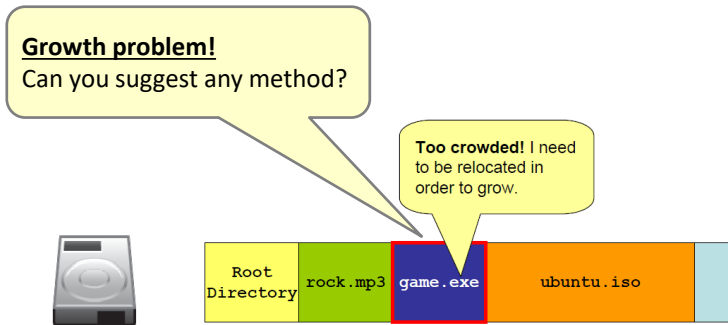
Filename	Starting Address	Ending Address
rock.mp3	0	2000
game.exe	2001	2679
ubuntu.iso	2680	6000

Very expensive (think about the disk structure)



Trial 1.0 – the bad #2

- Comment:
 - Also, the **growth** problem...there is no space for files to grow.



Trial 1.0 – the reality

- This kind of file systems has a name called the **contiguous allocation**.
- This kind of file system is not totally useless...
 - The suitable storage device is something that is...
 - **read-only** (just like a book)

Trial 1.0 – the reality

- Can you think of any real life example?
 - Hint #1: better not grow any files.
 - Hint #2: OK to delete files.
 - Hint #3: better not add any files; or just add to the tail.
- ISO9660.



File System Layout

Trial 2.0

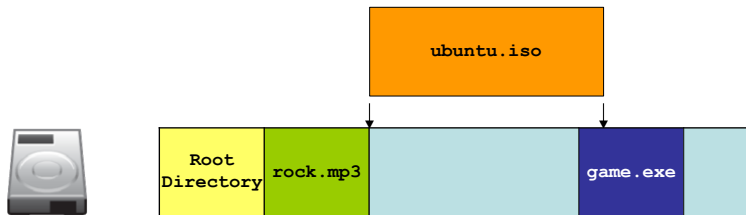
The Linked List Allocation

From Trial 1.0 to Trial 2.0...

- Lessons learned from Trial 1.0:
 - **File Size Growth:**
 - Can we let every file to grow without paying an experience overhead?
 - **External fragmentation:**
 - Can we reduce its damage?
- One goal
 - **To avoid allocating space in a contiguous manner!**

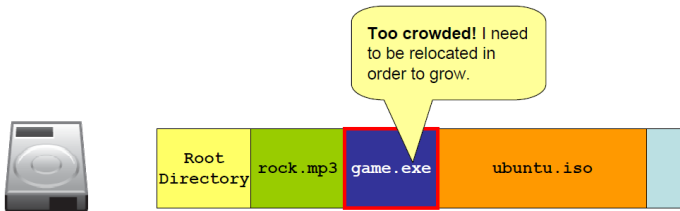
Trial 2.0 – the basics

- How?
 - The first undesirable case in trial 1.0 is to **write a large file** (as it may fail or need defragmentation)
 - So, can we write small files/units only?
 - For large files, let us break them into small pieces...



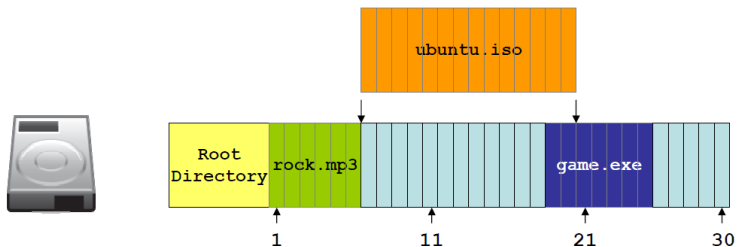
Trial 2.0 – the basics

- How?
 - The second undesirable case in trial 1.0 is when **file grows** (as it needs reallocation)
 - So, how can we support dynamic growth?
 - Let's borrow the idea from the linked list...



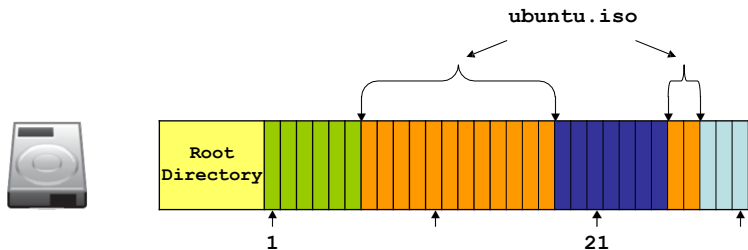
Trial 2.0 – the basics

- Linked list allocation...
 - Step (1): Chop the storage device into **equal-sized blocks**.



Trial 2.0 – the basics

- Linked list allocation...
 - Step (2): Fill the new file into the empty space in a **block-by-block** manner.



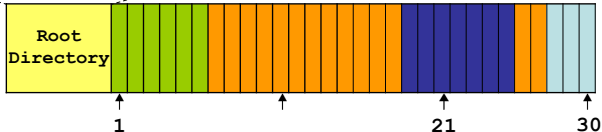
Trial 2.0 – the basics

- Linked list allocation...
 - Step (3): The root directory...
 - becomes strange/complicated.

Filename	Sequence of Block #	Sequence of Block #
rock.mp3	1-6	NULL
game.exe	19-25	NULL
ubuntu.iso	7-18	26-27

Since a directory file is an array, it is difficult to **pretend** to be a linked list....

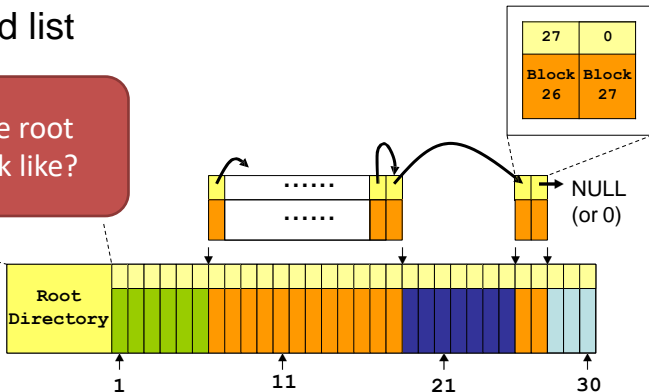
Can we have a better solution to optimize the directory?



Trial 2.1 – the linked list

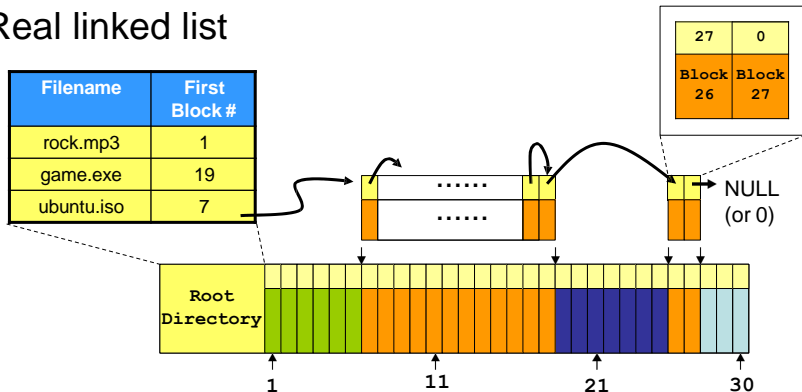
- Let's borrow 4 bytes from each block.
 - To write **the block # of the next block** into the first 4 bytes of each block.
 - Real linked list

How does the root directory look like?



Trial 2.1 – the linked list

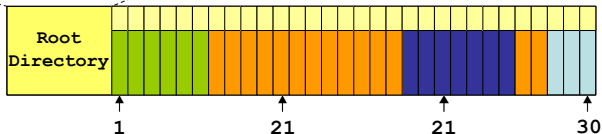
- Let's borrow 4 bytes from each block.
 - To write **the block # of the next block** into the first 4 bytes of each block.
 - Real linked list



Trial 2.1 – the file size

- Note that we need the **file size stored in the root directory** because...
 - The last block of a file **may not be fully filled**.

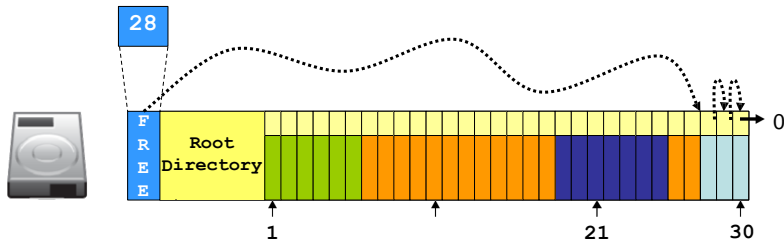
Filename	First Block #	File Size
rock.mp3	1	600M
game.exe	19	2000M
ubuntu.iso	7	700M



Trial 2.1 – the free space

- One more thing: **free space management**.
 - Extra data is needed to maintain a free list.

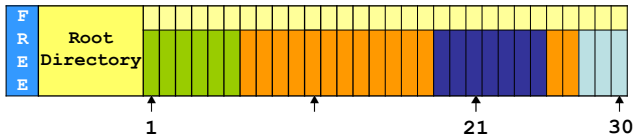
We can also maintain the free blocks as a linked list, too.



Trial 2.1 – the good

- Pros:

External fragmentation problem is solved.	Files can grow and shrink freely.	Free block management is easy to implement.
---	-----------------------------------	---



Trial 2.1 – the bad #1

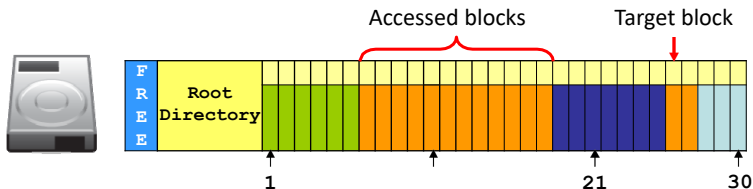
- Cons:

- **Random access performance problem.**

- The random access mode is to access a file at random locations.

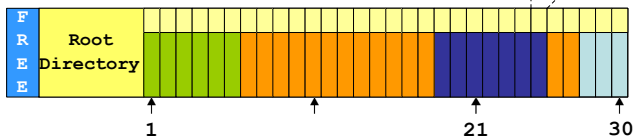
- The OS needs to access a series of blocks before it can access an arbitrary block.

- Worst case: **$O(n)$ number of I/O accesses**, where n is the number of blocks of the file.



Trial 2.1 – the bad #2

- Cons (recall why we record file size?):
 - **Internal Fragmentation.**
 - A file is not always a multiple of the block size
 - The last block of a file may not be fill completely.
 - This empty space will be wasted since no other files can be allowed to fill such space.



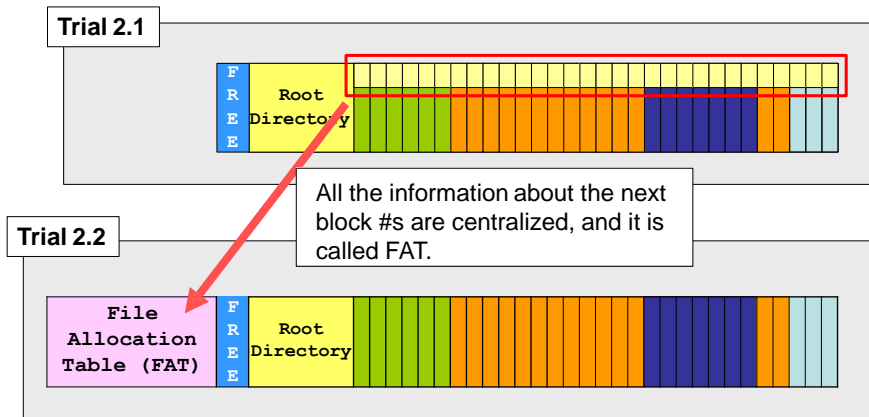
From Trial 2.1 to Trial 2.2

- Can we further improve?
 - We know that **the internal fragmentation problem is here to stay.**
 - How about the random access problem?
 - We are very wrong at the very beginning...decentralized next block location

The information about the next block should be centralized

Trial 2.2 – the FAT

- The only difference between 2.1 and 2.2...

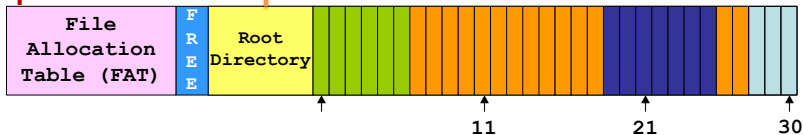


Trial 2.2 – the FAT implementation

Task: read “ubuntu.iso” sequentially.

Filename	First Block #
rock.mp3	1
game.exe	19
ubuntu.iso	7

Block #	1	...	6	7	...	18	19	...	25	26	27	28	29	30
Next Block #	2	...	0	8	...	26	20	...	0	27	0	29	30	0



Trial 2.2 – the FAT

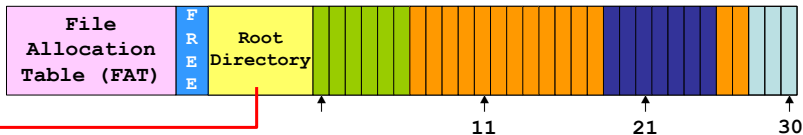
Task: read “ubuntu.iso” sequentially.

Step (1). Look for the first block # of the file.

Step
(1)

Filename	First Block #
rock.mp3	1
game.exe	19
ubuntu.iso	7

Block #	1	...	6	7	...	18	19	...	25	26	27	28	29	30
Next Block #	2	...	0	8	...	26	20	...	0	27	0	29	30	0



Trial 2.2 – the FAT

Task: read “ubuntu.iso” sequentially.

Step
(1)

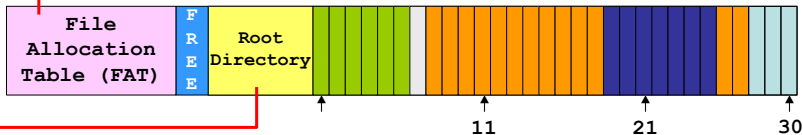
Filename	First Block #
rock.mp3	1
game.exe	19
ubuntu.iso	7

Step (2). Read the file allocation table to determine the location of the next block.

The next block of 7 is 8.

Step
(2)

Block #	1	...	6	7	...	18	19	...	25	26	27	28	29	30
Next Block #	2	...	0	8	...	26	20	...	0	27	0	29	30	0



Trial 2.2 – the FAT

Task: read “ubuntu.iso” sequentially.

Step (2). Read the file allocation table to determine the location of the next block.

Step
(1)

Filename	First Block #
rock.mp3	1
game.exe	19
ubuntu.iso	7

Note that the next block is not necessarily the adjacent one.

Step
(2)

Block #	1	...	6	7	...	18	19	...	25	26	27	28	29	30
Next Block #	2	...	0	8	...	26	20	...	0	27	0	29	30	0



Trial 2.2 – the FAT

Task: read “ubuntu.iso” sequentially.

Step
(1)

Filename	First Block #
rock.mp3	1
game.exe	19
ubuntu.iso	7

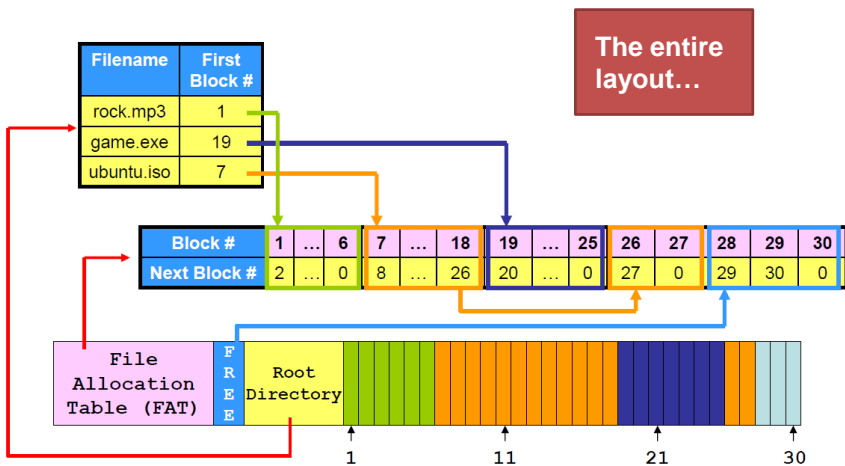
Step (3). The process stops until the block with the “next block # = 0”.

Step
(2)

Block #	1	...	6	7	...	18	19	...	25	26	27	28	29	30
Next Block #	2	...	0	8	...	26	20	...	0	27	0	29	30	0



Trial 2.2 – the FAT



Trial 2.2 – the lookup

- A point to look into:
 - Centralizing the data does not mean that the random access problem will be gone automatically, unless...
 - the file allocation table is presented as **an array**.

File Allocation Table

Block #	1	...	6	7	...	18	19	...	25	26	27	28	29	30
Next Block #	2	...	0	8	...	26	20	...	0	27	0	29	30	0

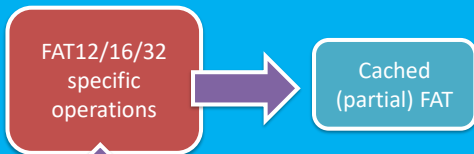
I know the
starting
position.

I know
the width.

So, going to an arbitrary location
is as simple as **doing a pointer
addition operation**.

The random access problem can be eased by keeping a **cached
version of FAT** inside the kernel.

Trial 2.2 – the lookup



If this table is partially kept on the cache, then **extra I/O requests** will be generated in locating the next block #.

Block #	1	...	6	7	...	18	19	...	25	26	27	28	29	30
Next Block #	2	...	0	8	...	26	20	...	0	27	0	29	30	0

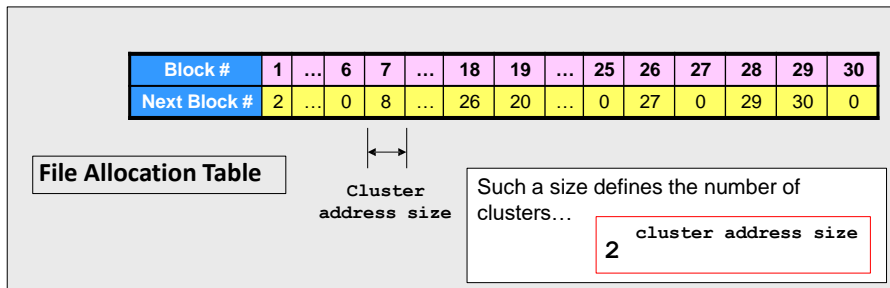
File Allocation Table (FAT)



- **Every file system** supported by MSDOS and the Windows family is implementing the linked list allocation.
- The file systems are:
 - The FAT family: FAT12, FAT16, and FAT32;
 - The New Technology File System: NTFS.

FATs Brief Introduction

- What is the meaning of the numbers (12/16/32)?
 - A block is named a **cluster**.
 - The main difference among all the versions of FAT FS-es is the **cluster address size**.



FATs Brief Introduction

- Cluster address sizes

File System	FAT12	FAT16	FAT32
Cluster address length	12 bits	16 bits	32 bits (28?)
Number of clusters	4K	64K	256M

- The larger the cluster address size is, the larger **the size of the file allocation table**.
- The larger the cluster size is, the larger **the size of the disk partition** is.

We will look into more details of FAT32 in later lectures

Summary of Trial 2.2

- Is FAT a perfect solution...
 - Tradeoff: **trade space for performance**
 - The entire FAT has to be stored in memory so that...
 - the performance of looking up of an arbitrary block is satisfactory.
- Can we have a solution that stands in middle?
 - Not store the entire set of block locations in mem...
 - I don't need an extremely high performance in block lookups.

File System Layout

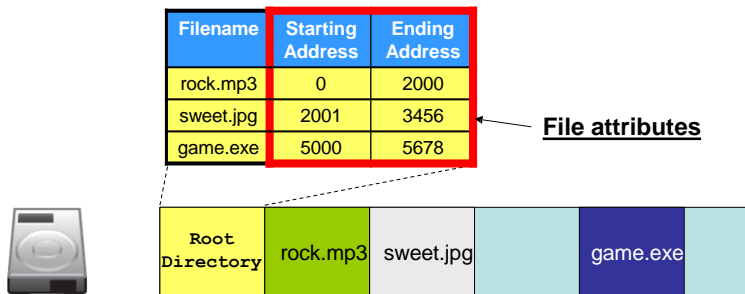
Trial 3.0

The Index-Node Allocation

Back to Trial 1.0-2.2

- File system layout: how to store file and directory
 - 1.0: Contiguous allocation (**just like a book**)

Two key problems: External fragmentation + file growth

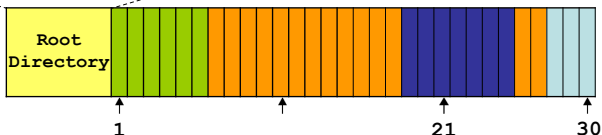


Back to Trial 1.0-2.2

- File system layout: how to store file and directory
 - 2.0: Linked-list allocation: **blocking**

Key problem: complicated root directory

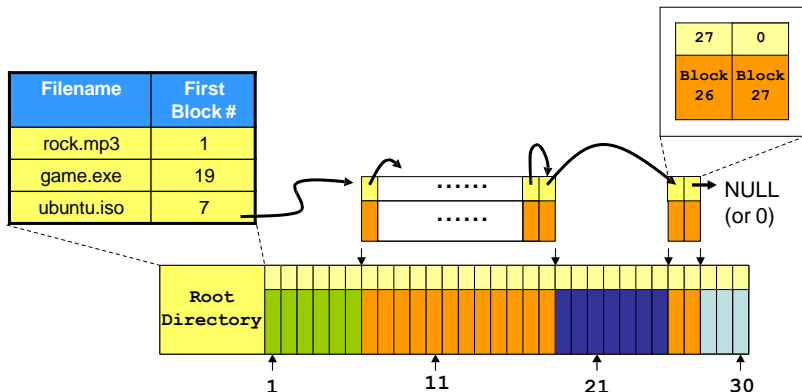
Filename	Sequence of Block #	Sequence of Block #
rock.mp3	1-6	NULL
game.exe	19-25	NULL
ubuntu.iso	7-18	26-27



Back to Trial 1.0-2.2

- File system layout: how to store file and directory
 - 2.1: Linked-list allocation: **blocking + linked list**

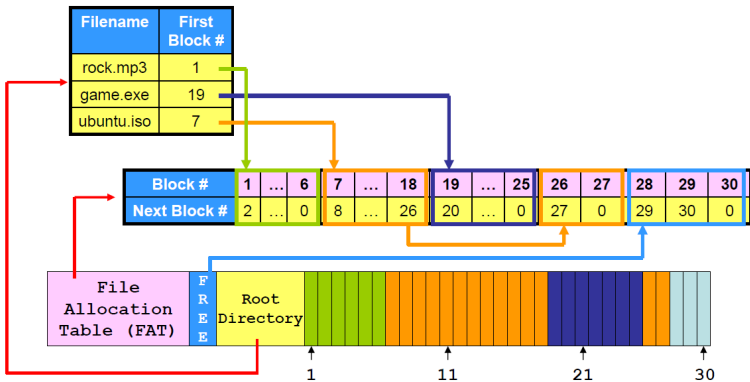
Key problem: random access problem



Back to Trial 1.0-2.2

- File system layout: how to store file and directory
 - 2.2: Linked-list allocation: **centralized next-block #** (FAT)

Requirement: FAT Caching

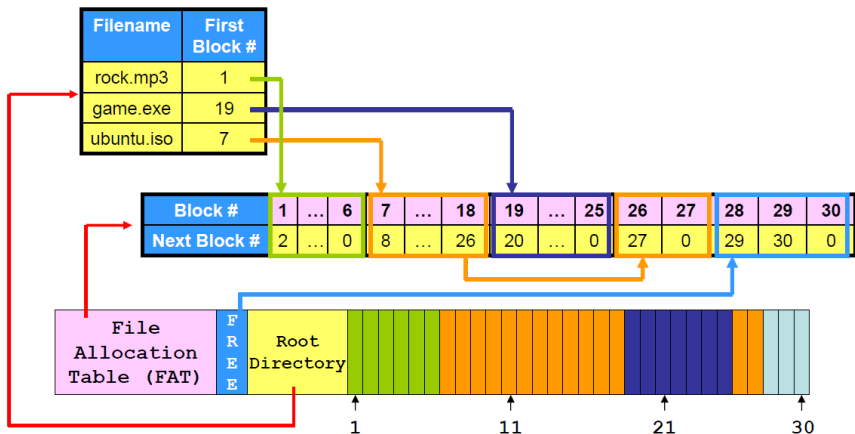


Trial 2.2 - FAT

- FAT provides a good performance in all aspects
 - File creation, file growth/shrink, file deletion ...
 - Random access performance...but requires to
 - **cache the FAT**
- Balance the tradeoff between Performance and memory space
 - **Partial caching**
 - **How?**

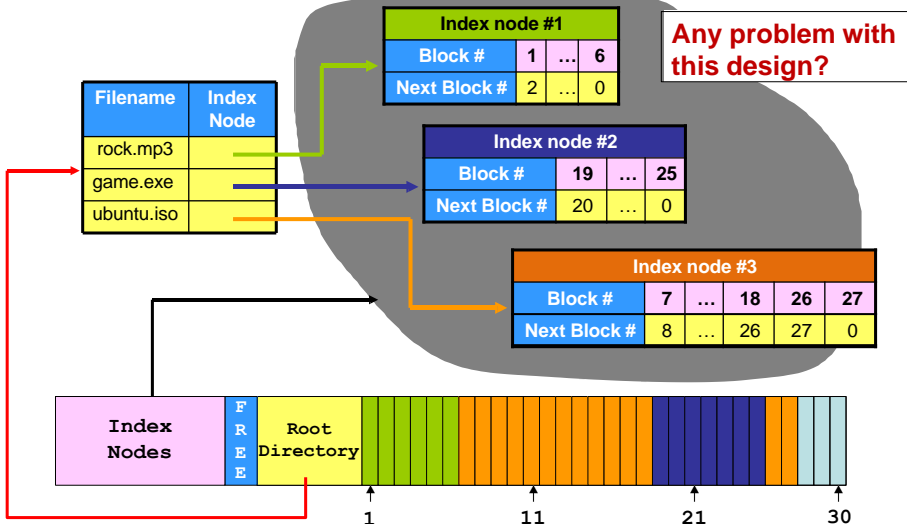
Trial 2.2 - FAT

We are going to **break the FAT into pieces**... Trial 3.0

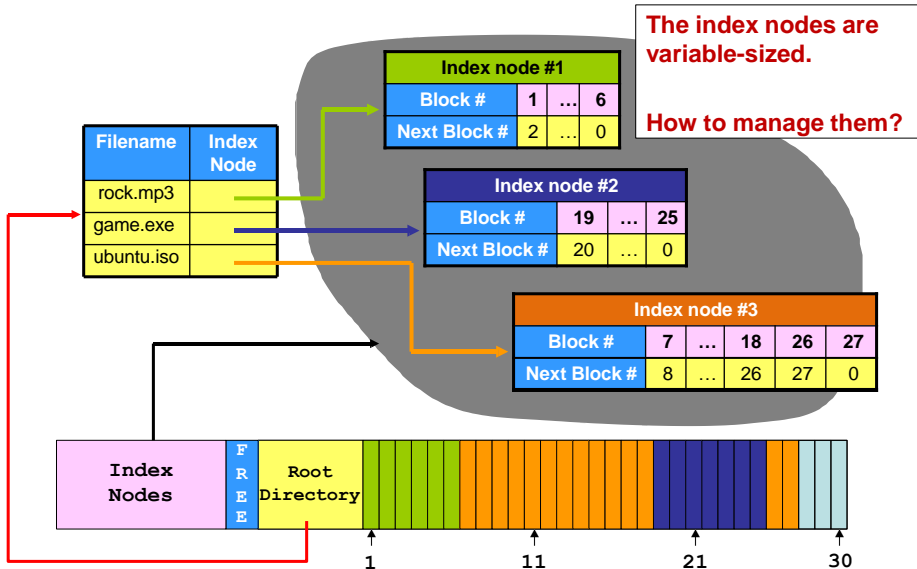


Trial 3.0 – the beginning

Block #	1	...	6	7	...	18	19	...	25	26	27	28	29	30
Next Block #	2	...	0	8	...	26	20	...	0	27	0	29	30	0



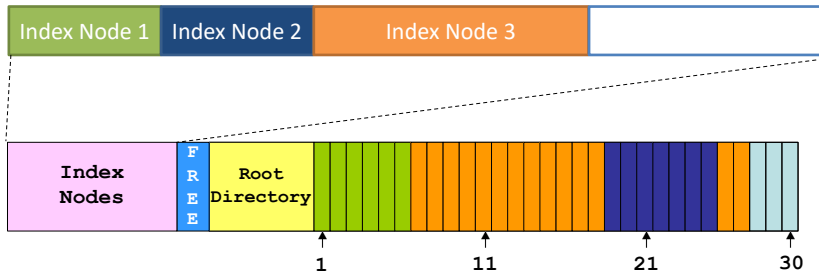
Trial 3.0 – the beginning



Trial 3.0 – the beginning

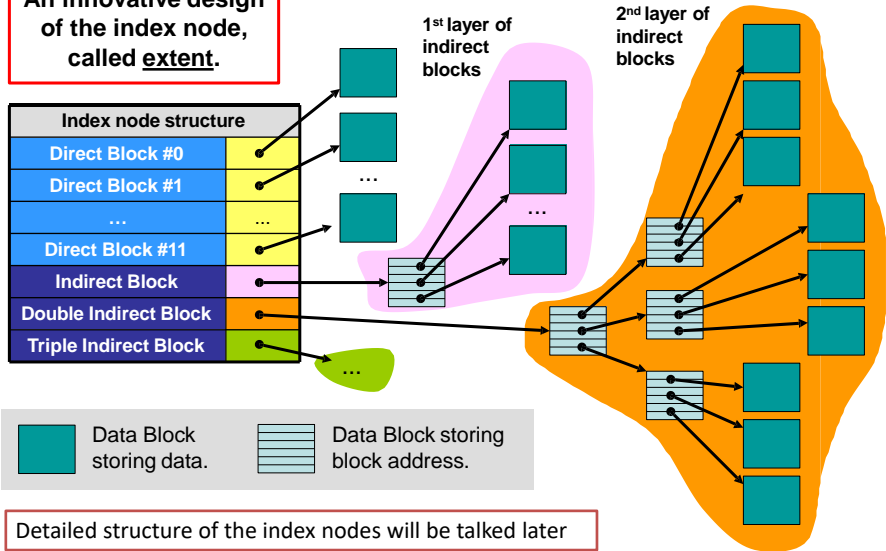
- Problems with variable-sized index nodes
 - How to locate an index node?
 - How to support file growth...size of index nodes depends on file size

Fix-sized index nodes are preferable, how to achieve?



Trial 3.0 – the heart

An innovative design
of the index node,
called extent.



Trial 3.0 – the two kinds of blocks

Indirect block

Stores an array of **block addresses**.

An address may point to either a data block or another indirect block.

However, in a block, **all** the addresses are either pointing to indirect blocks or data blocks.

Data block

Stores file data.

Keys



Indirect blocks that point to indirect blocks

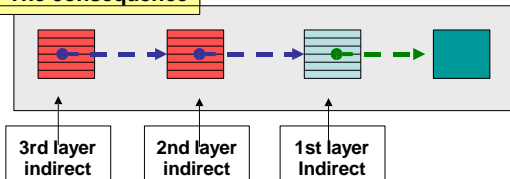


Indirect blocks that point to data blocks



Data blocks

The consequence



Where are the (indirect) blocks stored?



Trial 3.0 – the file size

How large files can be supported?

Number of direct blocks	12
Number of indirect blocks	1
Number of double indirect blocks	1
Number of triple indirect blocks	1
Block size	2^x bytes
Address length	4 bytes



" $2^x / 4 = 2^{x-2}$ "
addresses

File size = number of data blocks * block size

$$12 \times 2^x +$$

$$2^{x-2} * 2^x = 2^{2x-2} +$$

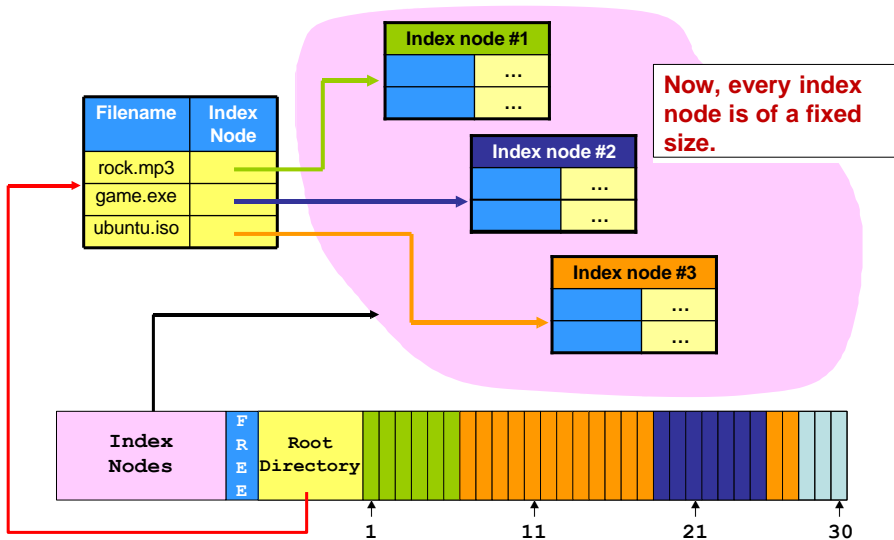
$$2^{x-2} * 2^{x-2} * 2^x = 2^{3x-4} +$$

$$2^{4x-6}$$

The dominating factor.

Block size	File size
1024 bytes = 2^{10}	approx. 16 Gbytes
4096 bytes = 2^{12}	approx. 4 Tbytes

Trial 3.0 – the final design



Trial 3.0 – the final design

Layout & read process

Filename	Index Node #
rock.mp3	1
game.exe	2
ubuntu.iso	3

Now, this column stores the **index node #**.

Inside the index node table ...

Index node #1		Index node #2		...		Index node #n-1	

Searching the index nodes **using the index node #**.

It is arranged as **an array**. So, looking up an index node will be fast.



Trial 3.0

- How about the tradeoff between performance and memory usage?
 - Partial caching is easy
- Any overhead of Trial 3.0?
 - The index-node allocation uses more storage:
 - **to trade for a larger file size (with fixed-size index nodes).**
 - The indirect blocks are the **extra things**.

Trial 3.0 – Storage Overhead

- The indirect blocks are the **extra things**.

File Size	$12 \times 2^x + 2^{2x-2}$	~4M (x=12)
# of Indirect Blocks	$(2^{x-2})^0$	1 block

File Size	$12 \times 2^x + 2^{2x-2} + 2^{3x-4}$	~4G (x=12)
# of Indirect Blocks	$(2^{x-2})^0 + (2^{x-2})^0 + (2^{x-2})^1$	~1K blocks

Trial 3.0 – Storage Overhead

- The indirect blocks are the **extra things**.

$\sim 4T \ (x=12)$	
File Size	$12 \times 2^x + 2^{2x-2} + 2^{3x-4} + 2^{4x-6}$
# of Indirect Blocks	$(2^{x-2})^0 + (2^{x-2})^0 + (2^{x-2})^1 + (2^{x-2})^0 + (2^{x-2})^1 + (2^{x-2})^2$
$\sim 1M \text{ blocks}$	

Trial 3.0 – Storage Overhead

- The indirect blocks are the **extra things**.
 - Max. number of indirect blocks depends on
 - Block size
 - File size

$$(2^{x-2})^0 + (2^{x-2})^1 + (2^{x-2})^2$$

Block size	Max. # of indirect blocks	Max. Extra Size involved
1024 bytes = 2^{10}	approx. 2^{16}	approx. 256 Mbytes
4096 bytes = 2^{12}	approx. 2^{20}	approx. 4 Gbytes

Remember, they are not static and they grow/shrink with the file size.

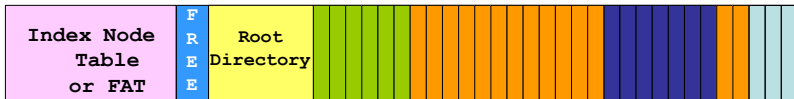


- FSes in UNIX and Linux use the index-node allocation method.
 - The **Ext2/3/4** file systems.
 - The index node is called **inode** in those systems.
 - Ext4 uses extent, not indirect blocks
 - We will discuss the details of Ext file system later.

From Trial 1.0 to Trial 3.0...

- We studied what are the possible ways to store data in the storage device.
 - The things stored are usually:

<u>Root directory</u> Hey, where are the sub-directories? Still remember the directory traversal	<u>File attributes</u> Except the file size and the locations of the data blocks, where and what are the other attributes ?
<u>Free space management</u> Actually, we didn't cover that much...	<u>Data block management</u> The FAT, the extents, the table of content.

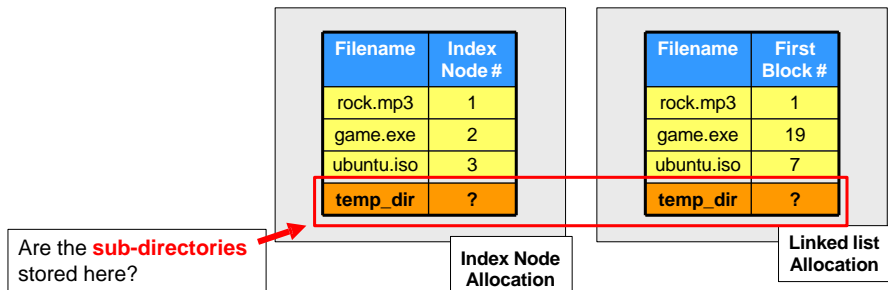


File System Layout

Root Directory and
Sub-directories

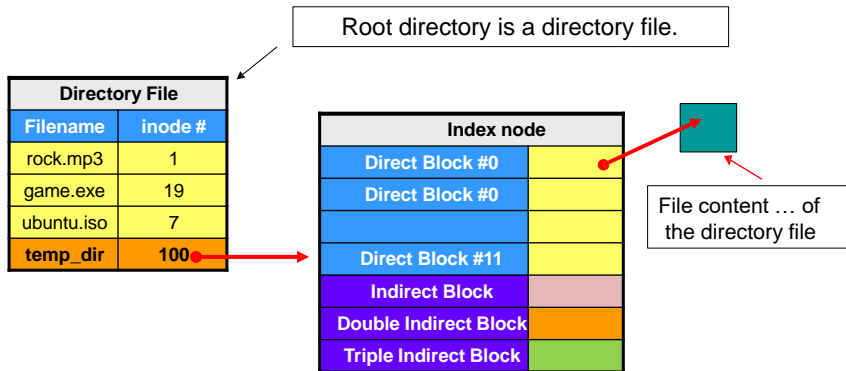
Root directory

- We know that the root directory is vital.
 - However, we have sub-directories...
 - Where are they?



Sub-directories?

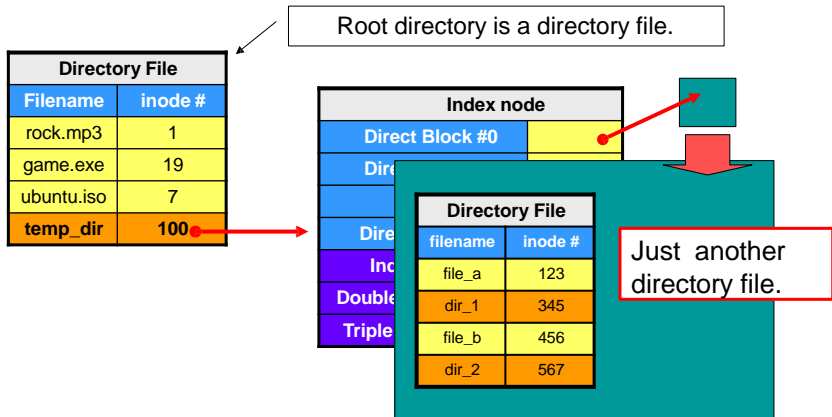
- Let's take the index-node allocation as an example...



Directory is also a file, so it has an inode too

Sub-directories?

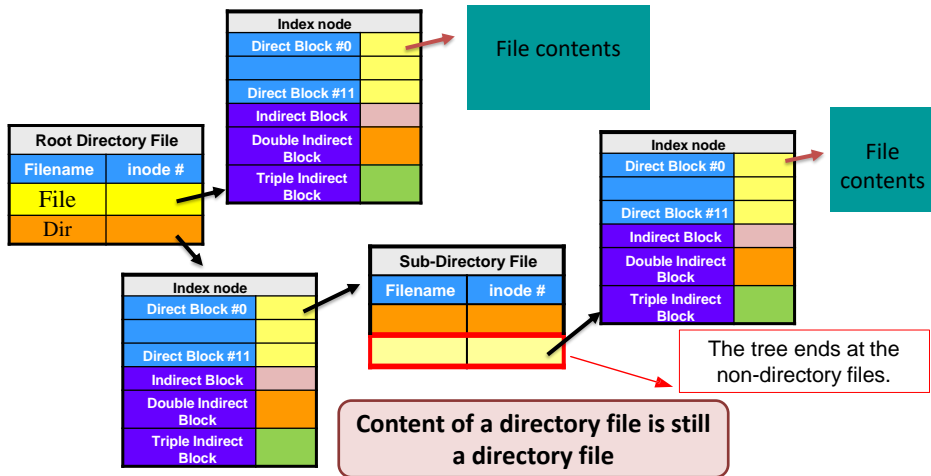
- Let's take the index-node allocation as an example...



See, each directory entry keeps the **address** of the file attributes, not the attributes themselves (how about FAT file systems?)

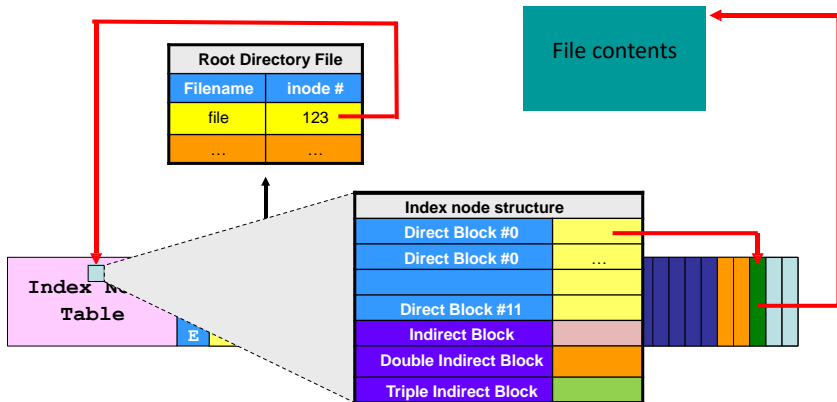
Traversing directory structure...

- Let's take index-node allocation as an example...



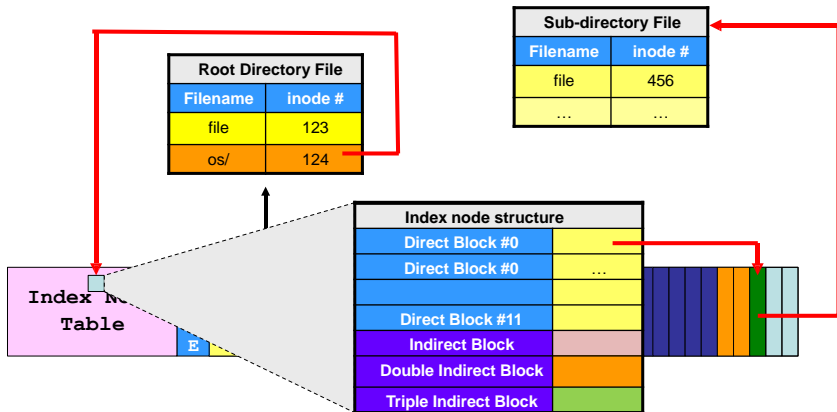
Traversing directory structure...

- Work together with the layout
 - Let's still take index-node allocation as an example...
 - E.g.: `/file`



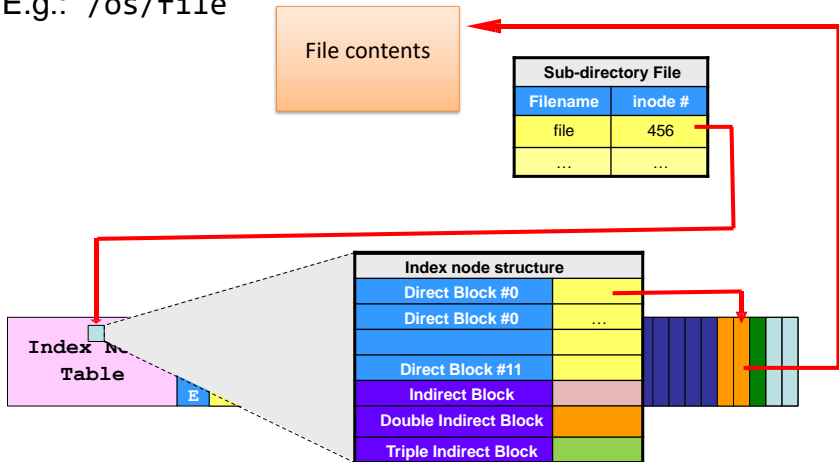
Traversing directory structure...

- Work together with the layout
 - Let's still take index-node allocation as an example...
 - E.g.: “/os/file”



Traversing directory structure...

- Work together with the layout
 - Let's still take index-node allocation as an example...
 - E.g.: `/os/file`

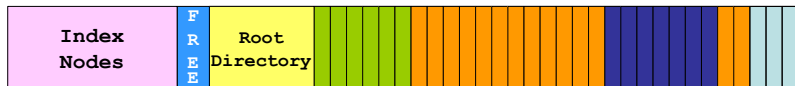


File System Layout

File system information
and partitioning

Storage layout

- What are stored on disk?
 - Root directory, index nodes/FAT, data blocks, free space information...
 - Others?
 - E.g., How do we know where the root directory is?
 - Where is the first inode?
 - File system information



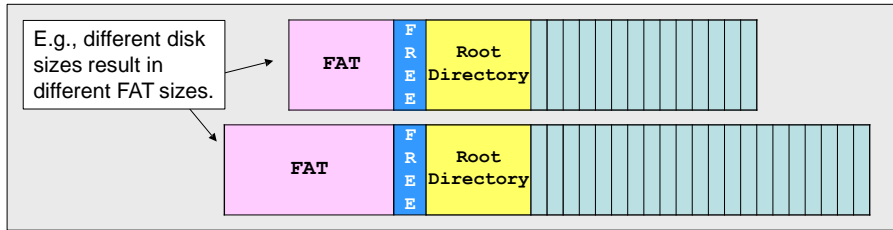
File System Information

- It is a set of important, FS-specific data...

Examples of FS-Specific Data
How large is a block?
How many allocated blocks are there?
How many free blocks are there?
Where is the root directory?
Where is the allocation information, e.g., FAT & inode table?
How large is the allocation information?

File System Information

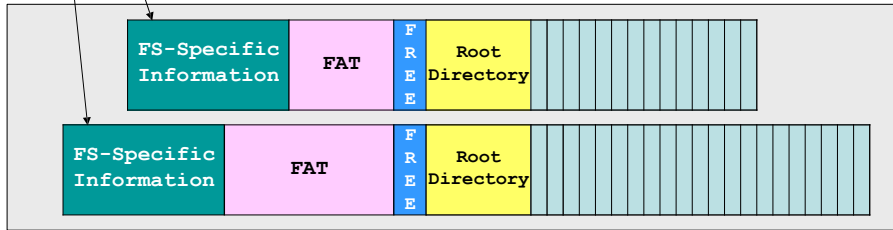
- It is a set of important, FS-specific data...
 - Can we **hardcode** those information in the kernel code...
 - **No!!!** Because different storage devices have different needs.



File System Information

- It is a set of important, FS-specific data...
 - Solution:** The workaround is to save those information on the device.

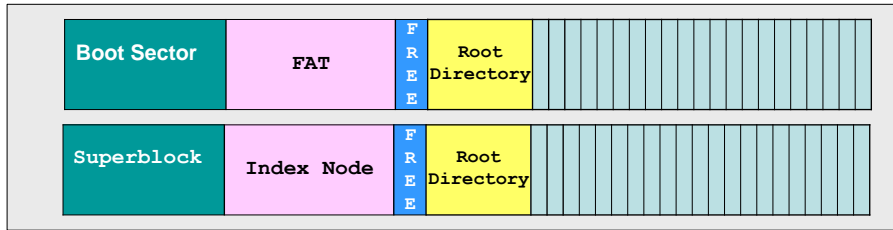
Each device should has its own copy of information.



File System Information

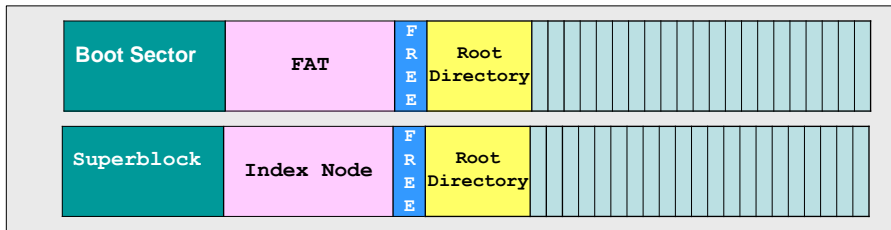
- It is a set of important, FS-specific data...
 - Solution:** The workaround is to save those information on the device.

In FAT* & NTFS	Boot Sector
In Ext*	Superblock



Story so far...

- We talked about the file system layout
 - FAT and index node



Only one file system can be stored in a disk?

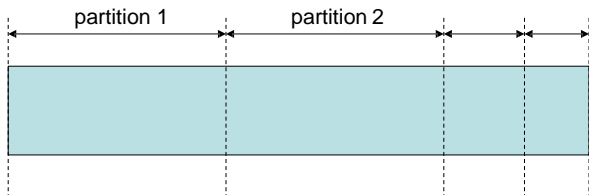
No!

What is the problem with a very large file system?

Large FAT

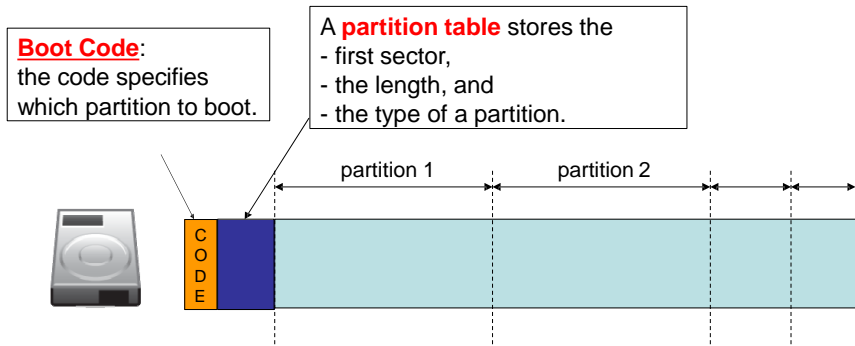
Disk partitions

- Partitioning is needed to
 - limit the file system size
 - support multiple file systems on a single disk

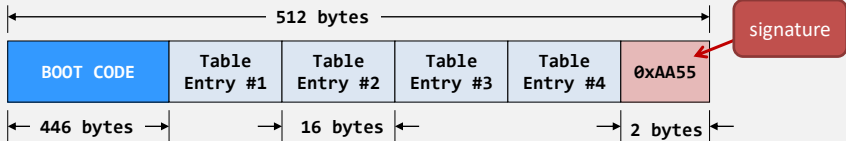


Disk partitions

- What is a disk partition?
 - A disk partition is a logical space...
 - A file system must be stored in a partition.
 - An operating system must be hosted in a partition.



Master boot record (MBR)...



The range of a partition is described by the: (offset, length) tuple.

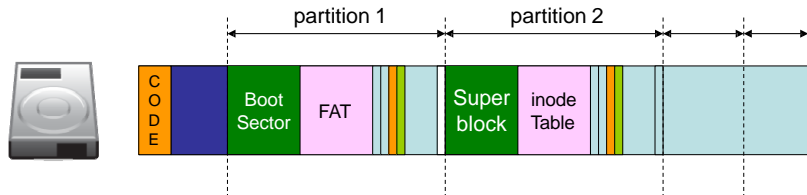
Partition Table Entry	
Bytes	Description
0-0	Bootable flag; 0x80 means bootable.
1-3	Starting CHS address
4-4	Partition type http://www.datarecovery.com/hexcodes.asp
5-7	Ending CHS address
8-11	Starting LBA address (measured in # of sectors)
12-15	Sizes in sectors

Disk partitions - summary

- Benefits of partitioning:
 - **Performance**
 - A smaller file system is more efficient!
 - Think about FAT32.
 - **Multi-booting**
 - You can have a Windows XP + Linux + Mac installed on a single hard disk (not using VMware).
 - **Data management**
 - You can have one logical drive to store movies, one logical drive to store the OS-related files, etc.

Final view of a disk storage space

- Final view of disk layout



- Now, do you know what is meant by “formatting” a disk?
 - Create and initialize a file system!
 - In Windows, we have “**format.exe**”.
 - In Linux, we have “**mkfs.ext2**”, “**mkfs.ext3**”, etc.

Summary of part2

- We have looked into many details about different file system layouts:
 - Contiguous allocation;
 - Linked list allocation; and
 - Index-node allocation.
- We also show the complete view of disk space
 - File system specific information & disk partition
- Linked list allocation and index-node allocation are the main streams but not the only way to implement modern file systems.

So far, we have learnt:

What are stored on disk

File: content + attributes

Directory: Directory file

How to access them?

File operations: open(), read(), write()

Directory lookup: Directory traversal

How are the files stored on disk?

File system layout: Contiguous/linked-list (FAT)/index-node allocation

Topics not covered:

Only the attributes of file name and locations are covered, how about other attributes? Free space management?

We'll look into some **real implementations** (FAT32 + EXT2/3/4)

Operating Systems

Associate Prof. Yongkun Li

中科大-计算机学院 副教授

<http://staff.ustc.edu.cn/~ykli>

Ch10, part 1 Details of FAT32

Story so far...

What are stored on disk

File: content + attributes

Directory: Directory file

How to access them?

File operations: open(), read(), write()

Directory lookup: Directory traversal

How are the files stored on disk?

File system layout

Contiguous allocation
linked-list allocation (FAT*)
index-node allocation (EXT*)

Topics in Ch10

- Case study

Details of FAT32

File attributes and directory entries, file operations

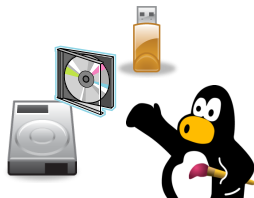
Details of Ext2/3/4

Detailed layout, detailed inode structure (file attributes), FS operations...

Details of FAT32

- **Introduction**
- Directory and File Attributes
- File Operations
 - Read files
 - Write files
 - Delete files
 - Recover deleted files

Microsoft Extensible Firmware Initiative FAT32 File System Specification (FAT: General Overview of On-Disk Format), Version 1.03, December 6, 2000, hardware white papers @ Microsoft Corporation.

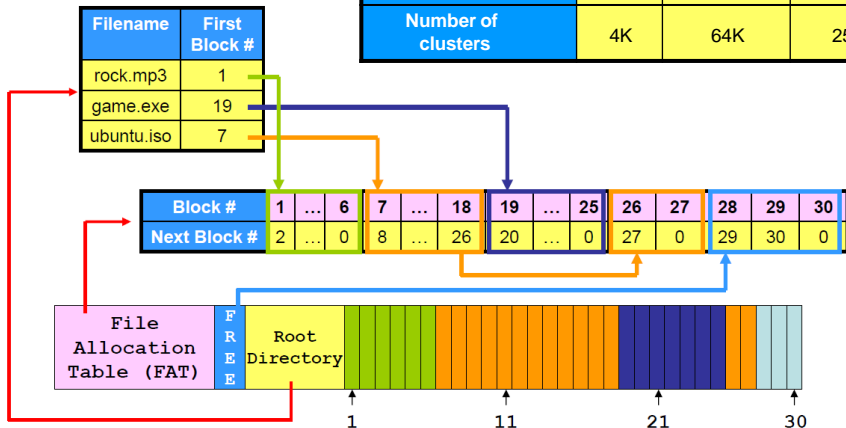


Recall on FAT allocation

- The layout

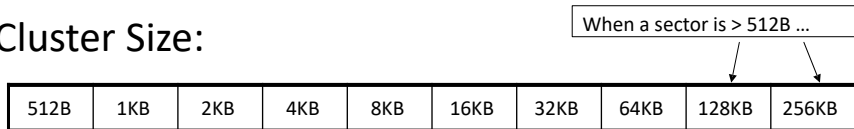
A block is named a **cluster**.

File System	FAT12	FAT16	FAT32
Cluster addr length	12 bits	16 bits	32 bits (28?)
Number of clusters	4K	64K	256M



Trivia

- Cluster Size:



- Try typing “**help format**” in the command prompt in Windows.

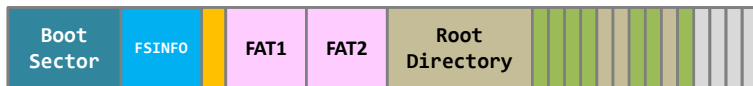
- Calculating the maximum partition size

- with the cluster size = 32KB...

$$(32 \times 2^{10}) \times 2^{28} = 2^{43} (8TB)$$

Typical layout of a FAT32 partition

	Propose	Size
Boot sector	Store FS-specific parameters	1 sector, 512 bytes
FSINFO	Free-space management	1 sector, 512 bytes
Reserved sectors	Don't ask me, ask Micro\$oft!	Variable, can be changed during format.
FAT (2 pieces)	A robust design : if "FAT 1" is corrupted or containing bad sectors, then "FAT 2" can act as a backup.	Variable, depends on disk size and cluster size.
Root directory	Start of the directory tree.	At least one cluster, depend on the number of director entries.



Typical layout of a FAT32 partition

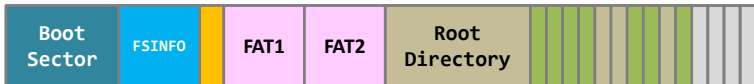
```
$ sudo mkfs.vfat -F32 /dev/ram0  
mkfs.fat 3.0.28 (2015-05-16)  
.....  
$ sudo dosfsck -v /dev/ram0
```

Format the disk, “-F32” means FAT32.

Read the information stored in the boot sector.

Running “**dosfsck**”, *DOS file system check*, on a FAT32 FS.

This program reads details from the **Boot Sector**.



Typical layout of a FAT32 partition

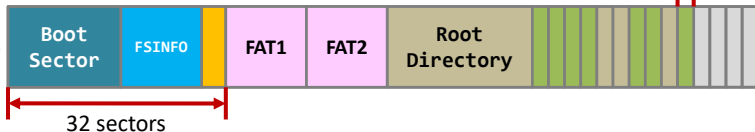
```
$ sudo mkfs.vfat -F32 /dev/ram0
mkfs.fat 3.0.28 (2015-05-16)
.....
$ sudo dosfsck -v /dev/ram0
fsck.fat 3.0.28 (2015-05-16)

Checking we can access the last sector of the filesystem
Boot sector contents:
System ID "mkfs.fat"
Media byte 0xf8 (hard disk)
  512 bytes per logical sector
  512 bytes per cluster
  32 reserved sectors
First FAT starts at byte 16384 (sector 32)
  2 FATs, 32 bit entries
  516608 bytes per FAT (= 1009 sectors)
Root directory start at cluster 2 (arbitrary size)
Data area starts at byte 1049600 (sector 2050)
  129022 data clusters (66059264 bytes)
.....
```

Details of the **Boot Sector**

The boot sector says:
A cluster is made of 1 sector.

One cluster size: 512
bytes in this case



Typical layout of a FAT32 partition

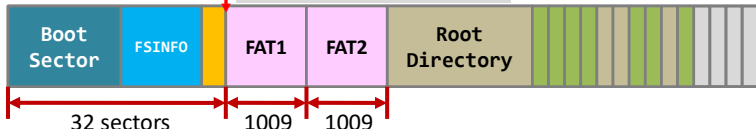
```
$ sudo mkfs.vfat -F32 /dev/ram0
mkfs.fat 3.0.28 (2015-05-16)
.....
$ sudo dosfsck -v /dev/ram0
fsck.fat 3.0.28 (2015-05-16)

Checking we can access the last sector of the filesystem
Boot sector contents:
System ID "mkdosfs"
Media byte 0xf8 (hard disk)
    512 bytes per logical sector
    512 bytes per cluster
    32 reserved sectors
First FAT starts at byte 16384 (sector 32)
    2 FATs, 32 bit entries
    516608 bytes per FAT (= 1009 sectors)
Root directory start at cluster 2 (arbitrary size)
Data area starts at byte 1049600 (sector 2050)
    129022 data clusters (66059264 bytes)
.....
```

The boot sector says:
2 FATs and each of them is of
size **516,608 bytes**.

Number of FATs and the
length of each entry in a FAT.

Good! No slack space between
reserved sectors of the first FAT.



Typical layout of a FAT32 partition

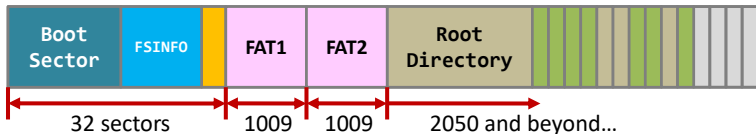
```
$ sudo mkfs.vfat -F32 /dev/ram0
mkfs.fat 3.0.28 (2015-05-16)
.....
$ sudo dosfsck -v /dev/ram0
fsck.fat 3.0.28 (2015-05-16)

Checking we can access the last sector of the filesystem
Boot sector contents:
System ID "mkdosfs"
Media byte 0xf8 (hard disk)
      512 bytes per logical sector
      512 bytes per cluster
      32 reserved sectors
First FAT starts at byte 16384 (sector 32)
      2 FATs, 32 bit entries
      516608 bytes per FAT (= 1009 sectors)
Root directory start at cluster 2 (arbitrary size)
Data area starts at byte 1049600 (sector 2050)
      129022 data clusters (66059264 bytes)
.....
```

The first data cluster is **Cluster #2** and it is usually, not always, the root directory.

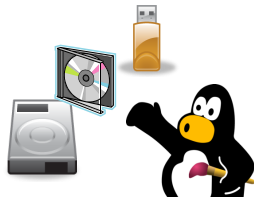
Cluster #0 & #1 are reserved.

$$32 + 1009 \times 2 = 2050$$



Details of FAT32

- Introduction
- **Directory and File Attributes**
- File Operations
 - Read files
 - Write files
 - Delete files
 - Recover deleted files



Directory Traversal

Step (1) Read the directory file of the root directory starting from **Cluster #2**.

"C:\windows" starts from Cluster #123.

```
c:\> dir c:\windows
```

```
.....
```

```
06/13/2012  2,033,216  explorer.exe
```

```
08/04/2015   169,120  notepad.exe
```

```
.....
```

```
c:\> _
```

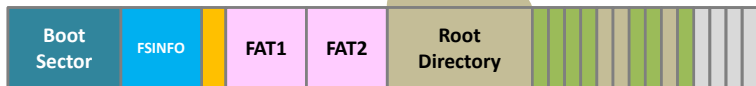
How does this work?

Cluster #2		
Filename	Attributes	Cluster #
.	?
..	?
.....
windows	123

A directory entry

Check this out by yourself.

Whether those two directory entries exist or not.



Directory Traversal

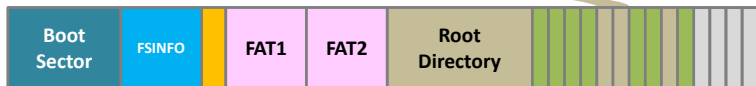
Step (2) Read the directory file of the "C:\windows" starting from **Cluster #123**.

```
c:\> dir c:\windows
.....
06/13/2012  2,033,216  explorer.exe
08/04/2015   169,120  notepad.exe
.....
c:\> _
```

How does this work?

Cluster #123		
Filename	Attributes	Cluster #
.	?
..	?
.....
notepad.exe	456

But, where are the information, e.g., file size, modification time, etc?



Directory entry

- Directory entry is just a structure.

Bytes	Description
0-0	1 st character of the filename (0x00 or 0xe5 means unallocated)
1-10	7+3 characters of filename + extension.
11-11	File attributes (e.g., read only, hidden)
12-12	Reserved.
13-19	Creation and access time information.
20-21	High 2 bytes of the first cluster address (0 for FAT16 and FAT12).
22-25	Written time information.
26-27	Low 2 bytes of first cluster address.
28-31	File size.



what?

Filename	Attributes	Cluster #
explorer.exe	32



How?

0	e	x	p	l	o	r	e	r	7
8	e	x	e	15
16	00	00	23
24	20	00	00	C4	0F	00	31

Note. This is the 8+3 naming convention.

8 characters for name +
3 characters for file extension

Directory entry

- Directory entry is just a structure.

Bytes	Description
0-0	1 st character of the filename (0x00 or 0xe5 means unallocated)
1-10	7+3 characters of filename + extension.
11-11	File attributes (e.g., read only, hidden)
12-12	Reserved.
13-19	Creation and access time information.
20-21	High 2 bytes of the first cluster address (0 for FAT16 and FAT12).
22-25	Written time information.
26-27	Low 2 bytes of first cluster address.
28-31	File size.



what?

Filename	Attributes	Cluster #
explorer.exe	32



How?

0	e	x	p	l	o	r	e	r	7
8	e	x	e	15
16	00	00	23
24	20	00	00	C4	0F	00	31

How to calculate the first cluster address?

Directory entry

- Directory entry is just a structure.

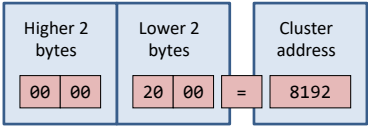
Bytes	Description
0-0	1 st character of the filename (0x00 or 0xe5 means unallocated)
1-10	7+3 characters of filename + extension.
11-11	File attributes (e.g., read only, hidden)
12-12	Reserved.
13-19	Creation and access time information.
20-21	High 2 bytes of the first cluster address (0 for FAT16 and FAT12).
22-25	Written time information.
26-27	Low 2 bytes of first cluster address.
28-31	File size.

what?

Filename	Attributes	Cluster #
explorer.exe	32

How?

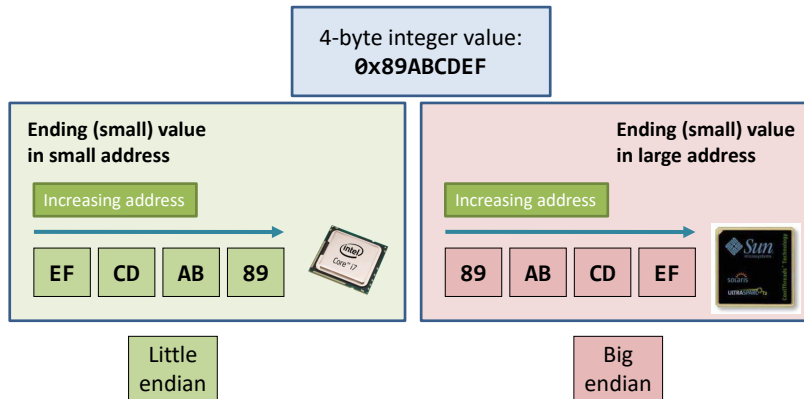
0	e	x	p	l	o	r	e	r	7
8	e	x	e	15
16	00	00	23
24	20	00	00	C4	0F	00	31



It is not 32, why?

Big Endian vs Little Endian

- Endian-ness is about **byte ordering**.
 - It means the way that a machine (we mean the entire computer architecture) orders the bytes.



Big Endian vs Little Endian

- Directory entry is just a structure.

Bytes	Description
0-0	1 st character of the filename (0x00 or 0xe5 means unallocated)
1-10	7+3 characters of filename + extension.
11-11	File attributes (e.g., read only, hidden)
12-12	Reserved.
13-19	Creation and access time information.
20-21	High 2 bytes of the first cluster address (0 for FAT16 and FAT12).
22-25	Written time information.
26-27	Low 2 bytes of first cluster address.
28-31	File size.

what?

Filename	Attributes	Cluster #
explorer.exe	32

How?

0	e	x	p	l	o	r	e	r	7
8	e	x	e	15
16	00	00	23
24	20	00	00	C4	0F	00	31

Big
endian

00	00	20	00	=	8192
----	----	----	----	---	------

Little
endian

00	00	00	20	=	32
----	----	----	----	---	----

The [FAT](#) is defined to use little-endian byte ordering, as its original implementation was on the Intel x86 platform

The file size...

Bytes	Description
0-0	1 st character of the filename (0x00 or 0xe5 means unallocated)
1-10	7+3 characters of filename + extension.
11-11	File attributes (e.g., read only, hidden)
12-12	Reserved.
13-19	Creation and access time information.
20-21	High 2 bytes of the first cluster address (0 for FAT16 and FAT12).
22-25	Written time information.
26-27	Low 2 bytes of first cluster address.
28-31	File size.

what?

Filename	Attributes	Cluster #
explorer.exe	32

How?

0	e	x	p	l	o	r	e	r	7
8	e	x	e	15
16	00	00	23
24	20	00	00	C4	0F	00	31

So, what is the largest size of a file?

4G – 1 bytes

Directory entry

- Any problem with this design?

Bytes	Description
0-0	1 st character of the filename (0x00 or 0xe5 means unallocated)
1-10	7+3 characters of filename + extension.
11-11	File attributes (e.g., read only, hidden)
12-12	Reserved.
13-19	Creation and access time information.
20-21	High 2 bytes of the first cluster address (0 for FAT16 and FAT12).
22-25	Written time information.
26-27	Low 2 bytes of first cluster address.
28-31	File size.

Note. This is the 8+3 naming convention.

8 characters for name +
3 characters for file extension

Example:

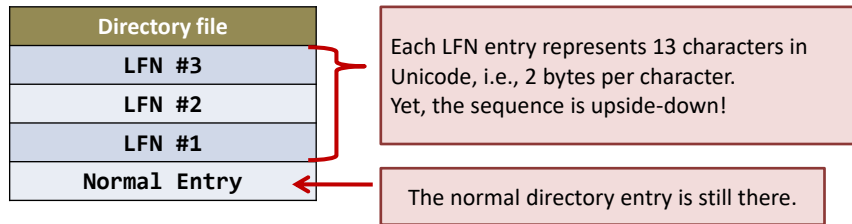
How to store the file:

"I_love_the_operating_system_course.txt"

How to store long
filename?

FAT series – LFN directory entry

- LFN: Long File Name.
 - In FAT32, the 8+3 naming convention is removed by...
 - Adding more entries to represent the filename



FAT series – LFN directory entry

Normal entry

Bytes	Description
0-0	1 st character of the filename (0x00 or 0xe5 means unallocated)
1-10	7+3 characters of filename + extension.
11-11	File attributes (e.g., read only, hidden)
12-12	Reserved.
13-19	Creation and access time information.
20-21	High 2 bytes of the first cluster address (0 for FAT16 and FAT12).
22-25	Written time information.
26-27	Low 2 bytes of first cluster address.
28-31	File size.

LFN entry

Bytes	Description
0-0	Sequence Number
1-10	File name characters (5 characters in Unicode)
11-11	File attributes - always 0x0F
12-12	Reserved.
13-13	Checksum
14-25	File name characters (6 characters in Unicode)
26-27	Reserved
28-31	File name characters (2 characters in Unicode)

FAT series – LFN directory entry

- Filename:
“I_love_the_operating_system_course.txt”.

Byte 11 is always 0x0F to indicate that is a LFN.

LFN #3	436d 005f 0063 006f 0075 000f 0040 7200 Cm._.c.o.u...@r. 7300 6500 2e00 7400 7800 0000 7400 0000 s.e...t.x...t...
LFN #2	0265 0072 0061 0074 0069 000f 0040 6e00 .e.r.a.t.i...@n. 6700 5f00 7300 7900 7300 0000 7400 6500 g._.s.y.s...t.e.
LFN #1	0149 005f 006c 006f 0076 000f 0040 6500 .I._.l.o.v...@e. 5f00 7400 6800 6500 5f00 0000 6f00 7000 _t.h.e._...o.p.
Normal	495f 4c4f 5645 7e31 5458 5420 0064 b99e I_LOVE~1TXT .d.. 773d 773d 0000 b99e 773d 0000 0000 0000 W=W=....W=.....

FAT series – LFN directory entry

This is the sequence number, and they are arranged in descending order.

The terminating directory entry has the sequence number **OR-ed with 0x40**.

Directory file

LFN #3: "m_cou" "rse.tx" "t"
LFN #2: "erati" "ng_sys" "te"
LFN #1: "I_lov" "e_the_" "op"
Normal Entry

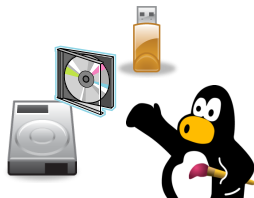
LFN #3	43	6d 005f 0063 006f 0075 000f 0040 7200	Cm._.c.o.u...@r.
		7300 6500 2e00 7400 7800 0000 7400 0000	s.e...t.x...t...
LFN #2	02	65 0072 0061 0074 0069 000f 0040 6e00	.e.r.a.t.i...@n.
		6700 5f00 7300 7900 7300 0000 7400 6500	g._.s.y.s...t.e.
LFN #1	01	49 005f 006c 006f 0076 000f 0040 6500	.I._.l.o.v...@e.
		5f00 7400 6800 6500 5f00 0000 6f00 7000	_.t.h.e._...o.p.
Normal		495f 4c4f 5645 7e31 5458 5420 0064 b99e	I_LOVE~1TXT .d..
		773d 773d 0000 b99e 773d 0000 0000 0000	W=W=....W=.....

FAT series – directory entry: a short summary

- A directory is an extremely important part of a FAT-like file system.
 - It stores the **start of the content**, i.e., the start cluster number.
 - It stores the **end of the content**, i.e., the file size; without the file size, how can you know when you should stop reading a cluster?
 - It stores **all file attributes**.

Details of FAT32

- Introduction
- Directory and File Attributes
- File Operations
 - **Read files**
 - Write files
 - Delete files
 - Recover deleted files



How to read a file?

Task: read "C:\windows\explorer.exe" sequentially.

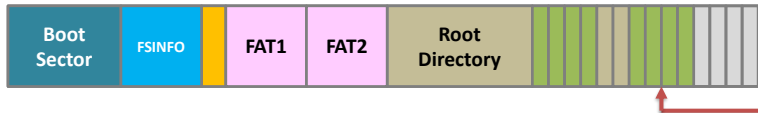
Suppose we already read out the directory entry...

You know the process of directory traversal, right?

Filename	Attributes	Cluster #
explorer.exe	32

Step 1. Read the content from Cluster #32.

Note. The **file size** may also help determine if the last cluster is reached (remember where it is stored?)



How to read a file?

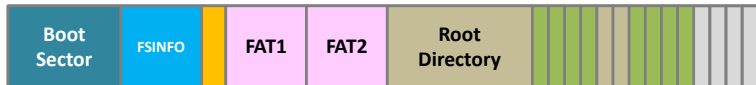
Task: read "C:\windows\explorer.exe" sequentially.

0	...
1	...
...	...
32	33
33	EOF
34	0
35	0

Filename	Attributes	Cluster #
explorer.exe	32

Step 1. Read the content from Cluster #32.
Note. The **file size** may also help determining if the last cluster is reached.

Step 2. Look for the next cluster and it is Cluster #33 (from the **FAT** table)



How to read a file?

Task: read "C:\windows\explorer.exe" sequentially.

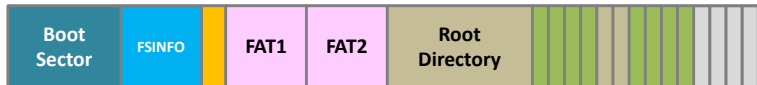
0	...
1	...
...	...
32	33
33	EOF
34	0
35	0

Filename	Attributes	Cluster #
explorer.exe	32

Step 3. Since the FAT has marked "EOF", we have reached the last cluster.

Note. The file size help determine **how many bytes to read** from the last cluster.

FAT entry structure??
Remember: 28bits are used to represent cluster number for FAT32



How to read a file?

Task: read "C:\windows\explorer.exe" sequentially.

0	...
1	...
...	...
32	33
33	EOF
34	0
35	0

Damaged = 0x0fffff7

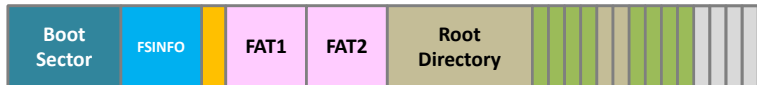
EOF >= 0x0fffff8

Unallocated = 0x0

Filename	Attributes	Cluster #
explorer.exe	32

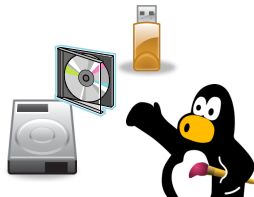
Step 3. Since the FAT has marked "EOF", we have reached the last cluster.

Note. The file size help determine **how many bytes to read** from the last cluster.



Details of FAT32

- Introduction
- Directory and File Attributes
- File Operations
 - Read files
 - **Write files**
 - Delete files
 - Recover deleted files



How to write a file?

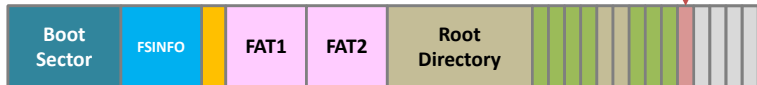
Task: append data to "C:\windows\explorer.exe".

0	...
1	...
...	...
32	33
33	EOF
34	0
35	0

Filename	Attributes	Cluster #
explorer.exe	32

Step 1. Locate the last cluster.

Step 2. Start writing to the non-full cluster.



How to write a file?

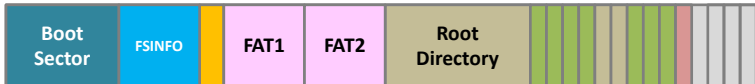
Task: append data to "C:\windows\explorer.exe".

0	...
1	...
...	...
32	33
33	EOF
34	0
35	0

Filename	Attributes	Cluster #
explorer.exe	32

Step 3. Allocate the next cluster through FSINFO.

What is stored in FSINFO?
How to allocate?



How to write a file?

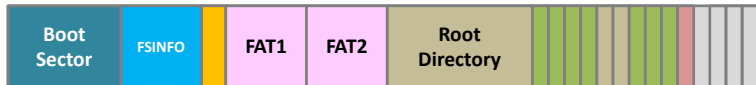
Task: append data to "C:\windows\explorer.exe".

0	...
1	...
...	...
32	33
33	EOF
34	0
35	0

FSINFO	
# of free clusters	4
Next free cluster #	34

Filename	Attributes	Cluster #
explorer.exe	32

Step 3. Allocate the next cluster through FSINFO.



How to write a file?

Task: append data to "C:\windows\explorer.exe".

0	...
1	...
...	...
32	33
33	34
34	EOF
35	0

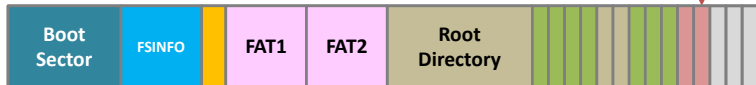
FSINFO	
# of free clusters	3
Next free cluster #	35

Filename	Attributes	Cluster #
explorer.exe	32

Step 3. Allocate the next cluster through FSINFO.

Step 4. Update the FATs and FSINFO.

Step 5. When write finishes, update the file size.



How to write a file?

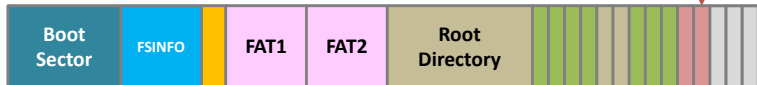
Task: append data to "C:\windows\explorer.exe".

0	...
1	...
...	...
32	33
33	34
34	EOF
35	0

FSINFO	
# of free clusters	3
Next free cluster #	35

Filename	Attributes	Cluster #
explorer.exe	32

Q: How to obtain the next free cluster?



How to write a file?

Task: append data to "C:\windows\explorer.exe".

0	...
1	...
...	...
32	33
33	34
34	EOF
35	0

Filename	Attributes	Cluster #
explorer.exe	32

The search for the next free cluster is a **circular, next-available** search.

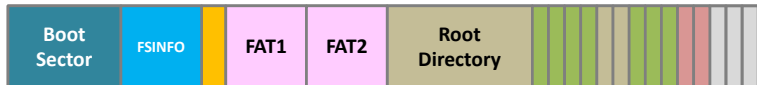
Why implementing next-available?

Principle of locality

Why circular?

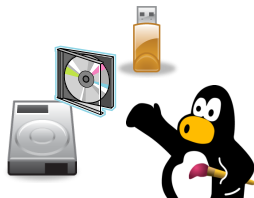
To find out every free block

FSINFO	
# of free clusters	3
Next free cluster #	35



Details of FAT32

- Introduction
- Directory and File Attributes
- File Operations
 - Read files
 - Write files
 - **Delete files**
 - Recover deleted files



How to delete a file?

Task: delete "C:\windows\explorer.exe".

0	...
1	...
...	...
32	33
33	34
34	EOF
35	0



0	...
1	...
...	...
32	0
33	0
34	0
35	0

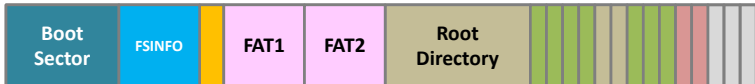
Filename	Attributes	Cluster #
explorer.exe	32

Step 1. De-allocate all the blocks involved. Update FSINFO and FATs.



FSINFO	
# of free clusters	3
Next free cluster #	35

FSINFO	
# of free clusters	6
Next free cluster #	32

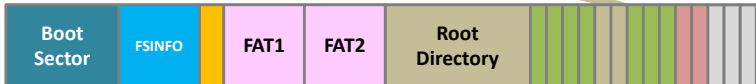


How to delete a file?

Task: delete "C:\windows\explorer.exe".

How about the directory entry

Cluster #123		
Filename	Attributes	Cluster #
.	?
..	?
explorer.exe	32
notepad.exe	456



How to delete a file?

Task: delete "C:\windows\explorer.exe".

Bytes	Description
0-0	1 st character of the filename (0x00 or 0xE5 means unallocated)

The first character becomes "0xE5".

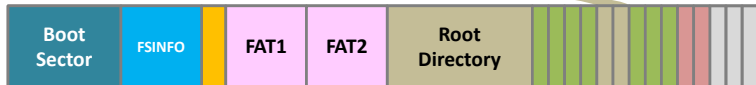
Cluster #123		
Filename	Attributes	Cluster #
.	?
..	?
_explorer.exe	32
notepad.exe	456

How about the directory entry

Step 2. Change the first byte of the directory entry to 0xE5.

LFN entries also receive the same treatment.

That's the end of deletion!



Really delete a file?

- Can you see that: **the file is not really removed from the FS layout?**
 - Perform a search in all the free space. Then, you will find all deleted file contents.
- “*Deleted data*” persists until the de-allocated clusters **are reused**.
 - This is an issue between performance (during deletion) and security.
- Any way(s) to delete a file **securely**?

How to delete a file “securely”?



Mac OS X Secure Disk Erase

Secure Erase Options

These options specify how to erase the selected disk or volume to prevent disk recovery applications from recovering it.

Note: Secure Erase overwrites data accessible to Mac OS X. Certain types of media may retain data that Disk Utility cannot erase.



This option meets the US Department of Defense (DOD) 5220-22 M standard for securely erasing magnetic media. It erases the information used to access your files and writes over the data 7 times.



Cancel

OK

Brute Force?

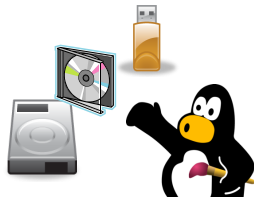
<http://www.ohgizmo.com/2009/06/01/manual-hard-drive-destroyer-looks-like-fun/>

What will the research community tell you?

<http://cdn.computerscience1.net/2006/fall/lectures/8/articles8.pdf>

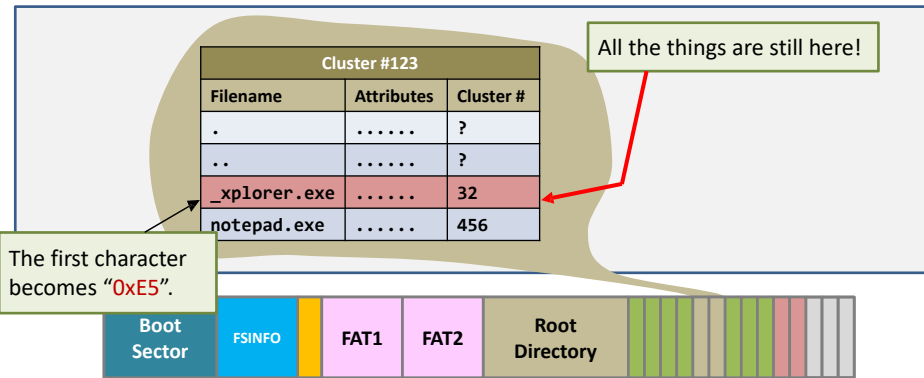
Details of FAT32

- Introduction
- Directory and File Attributes
- File Operations
 - Read files
 - Write files
 - Delete files
 - **Recover deleted files**



How to “rescue” a deleted file?

- If you’re really care about the deleted file, then...
 - **PULL THE POWER PLUG AT ONCE!**
 - Pulling the power plug stops the target clusters from being over-written.



How to “rescue” a deleted file?

- If you're really care about the deleted file, then...
 - **PULL THE POWER PLUG AT ONCE!**
 - Pulling the power plug stops the target clusters from being over-written.

Principle of “rescue” deleted file

Data persists unless the sectors are reallocated and overwritten.

File size ≤ 1
cluster

Because **the first cluster address** is still readable, the recovery is having a very high successful rate.

Note that filenames with the same postfix may also be found.

How to “rescue” a deleted file?

- If you’re really care about the deleted file, then...
 - **PULL THE POWER PLUG AT ONCE!**
 - Pulling the power plug stops the target clusters from being over-written.

Principle of “rescue” deleted file

Data persists unless the sectors are reallocated and overwritten.

File size > 1
cluster

It is still possible as the clusters of a file are likely to be contiguously allocated.

The next-available search provides a hint in looking for deleted blocks.

If not, you’d better have the **checksum** and **the exact file size** beforehand, so that you can use a **brute-force method** to recover the file.

How to “rescue” a deleted file?

- What if the value of the 32nd cluster is not 0?

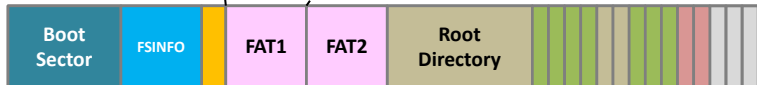
It is hard to find them out without some hints.

The use of checksum may be a good hint...

0	...
1	...
...	...
32	0
33	0
34	0
35	0

_xplorer.exe	32
--------------	-------	----

The first cluster is the one that we can be sure of...



FAT series – conclusion

- It is a “nice” file system:
 - Space efficient: 4 bytes overhead (FAT entry) per data cluster.
- Deletion problem:
 - This is a **lazy yet fast** implementation.
 - Need extra protection for deleted data.
- Deployment:
 - It is everywhere: SD cards, USB drives, disks...

Operating Systems

Associate Prof. Yongkun Li

中科大-计算机学院 副教授

<http://staff.ustc.edu.cn/~ykli>

Ch10, part2

Details of Ext2/3 File System

Trivia

- **Extended File System (Ext2/3/4)**
 - Follow index-node allocation
 - Primary FS for Linux distribution
 - Ext4 was merged in the Linux 2.6.28 and released in 2008
 - Backward-compatible
 - For simplicity, we focus on Ext2/3
 - Features of Ext2/3/4
 - https://ext4.wiki.kernel.org/index.php/Main_Page
 - <http://e2fsprogs.sourceforge.net/ext2.html>

Details of Ext2/3

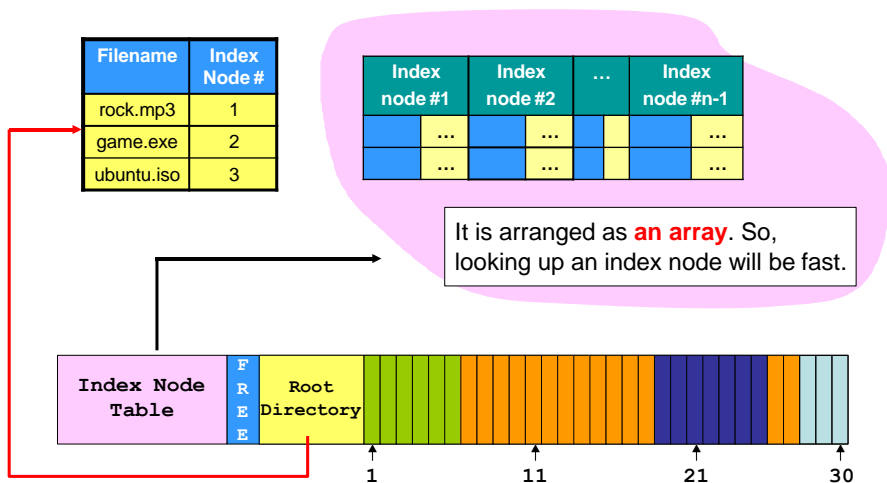
- Layout
- Inode and directory structure
- Link file
- Buffer cache
- Journaling
- VFS

Details of Ext2/3

- **Layout**
- Inode and directory structure
- Link file
- Buffer cache
- Journaling
- VFS

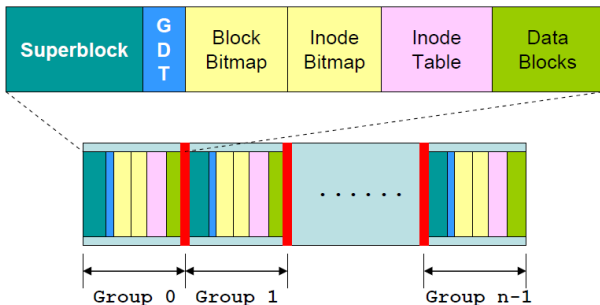
Index-node allocation

- Ext2/3 file systems follow the index-node allocation



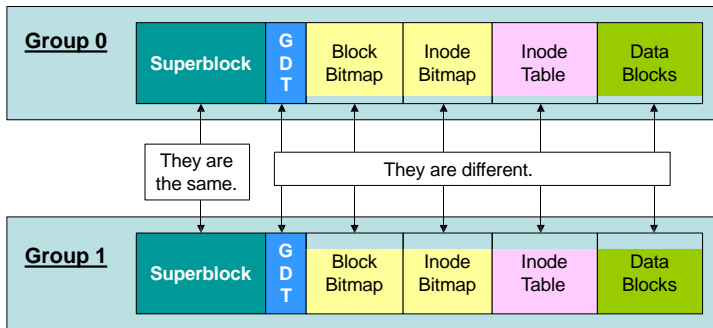
Specific Layout

- The file system is not that simple...
 - it is divided into groups, and ...
 - every group has the same structure.



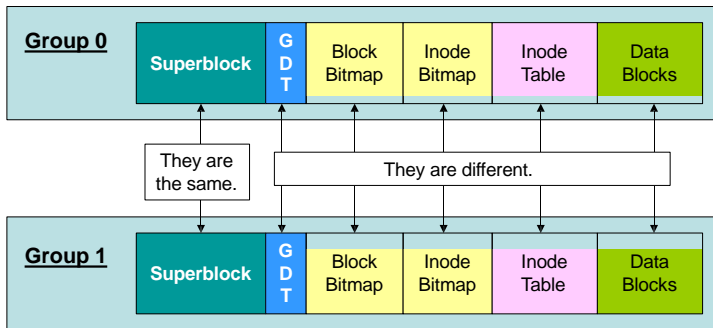
Specific Layout

- The file system is not that simple...
 - it is divided into groups, and ...
 - every group has the same structure.



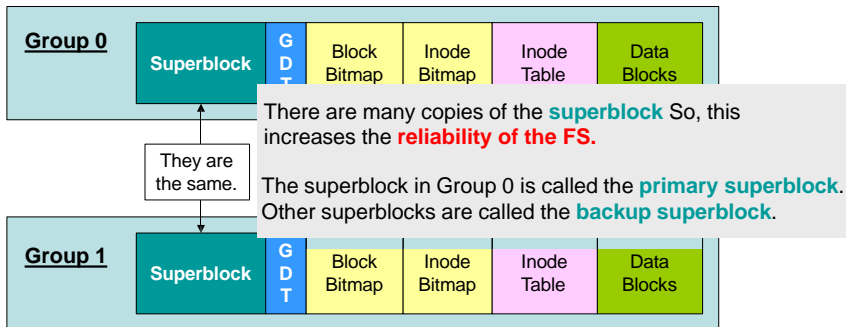
Specific Layout

- Why doing so?
 - This is for **reliability and performance**.



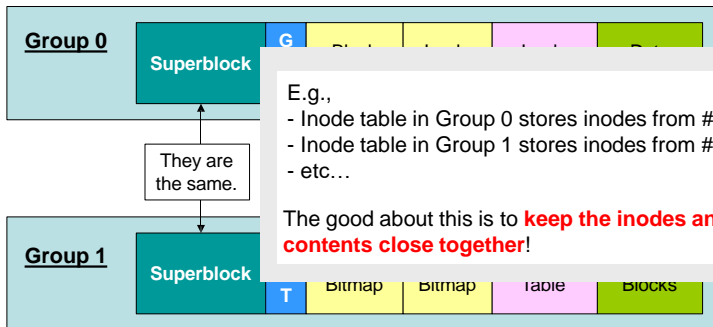
Specific Layout

- Why doing so?
 - For **reliability**...



Specific Layout

- Why doing so?
 - For **performance...**



E.g.,

- Inode table in Group 0 stores inodes from #1 to #100;
- Inode table in Group 1 stores inodes from #101 to #200;
- etc...

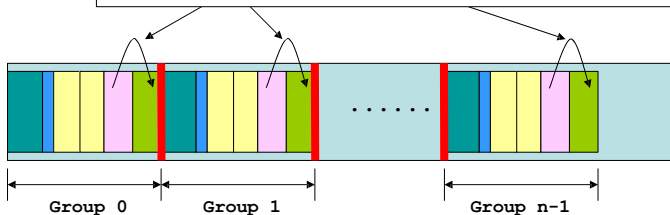
The good about this is to **keep the inodes and the file contents close together!**

Specific Layout

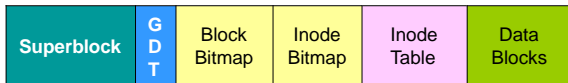
- Why doing so?
 - For **performance...**

The inodes in a particular group will *usually* refer to the data blocks in the same group.

So, this keeps them close together in a physical sense. The storage device may be able to locate the data in a faster manner. (Remember the **principle of locality**?)



Layout in Each Group

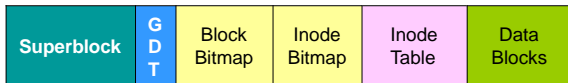


Superblock	Stores FS specific data.
------------	--------------------------



Total number of inodes in the system.
Total number of blocks in the system.
Number of reserved blocks
Total number of free blocks.
Total number of free inodes.
Location of the first block.
The size of a block.

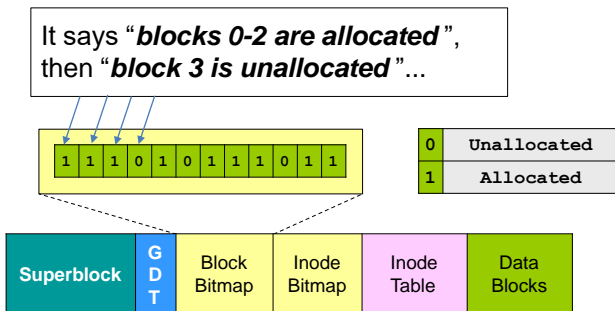
Layout in Each Group



Superblock	Stores FS specific data. E.g., the total number of blocks, etc.
GDT – Group Descriptor Table	It stores: <ul style="list-style-type: none">-The starting block numbers of the block bitmap, the inode bitmap, and the inode table.- Free block count, free inode count, etc...
Inode Table	An array of inodes ordered by the inode #.
Data Blocks	An array of blocks that stored files.
Block Bitmap	A bit string that represents if a block is allocated or not.
Inode Bitmap	A bit string that represents if an inode is allocated or not.

Layout in Each Group

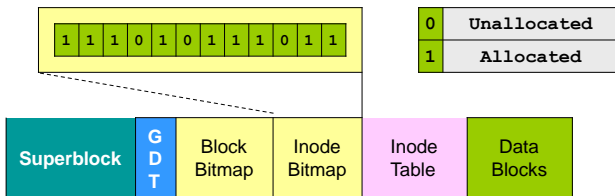
- What is a **block bitmap**?
 - A sequence of bits indicates **the allocation of the blocks**.



Layout in Each Group

- Then, what is an **inode bitmap**?
 - A sequence of bits indicates **the allocation of the inodes**.
 - This implies that...

The **number of files** in the file system is fixed!

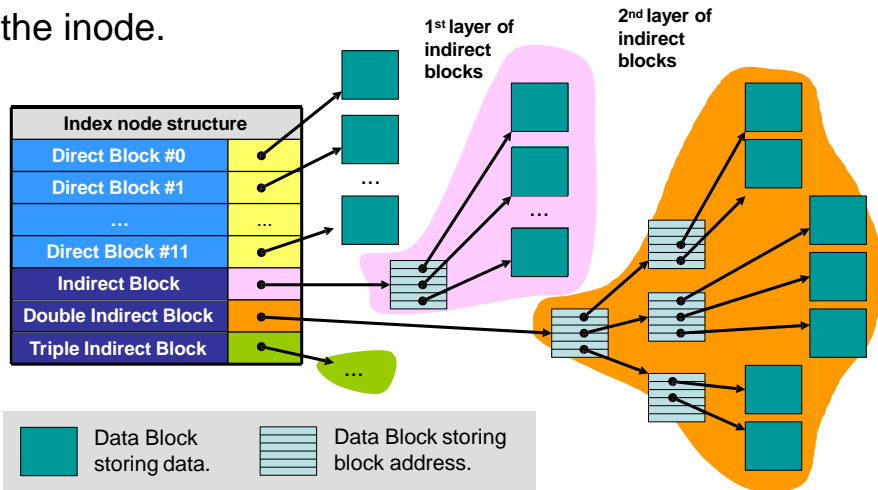


Details of Ext2/3

- Layout
- **Inode and directory structure**
- Link file
- Buffer cache
- Journaling
- VFS

Inode Structure

- We know that...
 - The locations of the data blocks of a file are stored in the inode.



Inode Structure

Inode Structure (128 bytes long)	
Bytes	Value
0-1	File type and permission
2-3	User ID
4-7	Lower 32 bits of file sizes in bytes
8-23	Time information
24-25	Group ID
26-27	Link count
...	...
40-87	12 direct data block pointers
88-91	Single indirect block pointer
92-95	Double indirect block pointer
96-99	Triple Indirect block pointer
...	...
108-111	Upper 32 bits of file sizes in bytes

What are stored in inode besides block addresses?

An inode is the structure that stores every information about a file.

The locations of the data blocks

More details: https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout#Inode_Table

Inode Structure

Inode Structure (128 bytes long)	
Bytes	Value
0-1	File type and permission
2-3	User ID
4-7	Lower 32 bits of file sizes in bytes
8-23	Time information
24-25	Group ID
26-27	Link count
...	...
40-87	12 direct data block pointers
88-91	Single indirect block pointer
92-95	Double indirect block pointer
96-99	Triple Indirect block pointer
...	...
108-111	Upper 32 bits of file sizes in bytes

What is the maximum file size supported?

$$2^{64} - 1$$

$$= 16 \times 2^{30} \text{ Gbytes} - 1 \text{ byte}$$

Is this really the case?

Remember the dominating factor: $2^{4 \times 6}$

Block size	File size
1024B = 2^{10}	~16 Gbytes
4096B = 2^{12}	~4 Tbytes

Inode Structure

Inode Structure (128 bytes long)	
Bytes	Value
0-1	File type and permission
2-3	User ID
4-7	Lower 32 bits of file sizes in bytes
8-23	Time information
24-25	Group ID
26-27	Link count
...	...
40-87	12 direct data block pointers
88-91	Single indirect block pointer
92-95	Double indirect block pointer
96-99	Triple Indirect block pointer
...	...
108-111	Upper 32 bits of file sizes in bytes

What is link count?

We will talk about it later

Where is the file name?

Let us take a look at the directory structure

Directory Structure

Filename	Index Node #
rock.mp3	1
game.exe	2
ubuntu.iso	3

The directory entry stores the **file name** and **the inode #**.

```
1  int main(void) {
2      DIR * dir;
3      struct dirent *entry;
4
5      di
6
7      wh
8
9
10     }
11
12     cl
13     re
14 }
```

```
struct dirent {
    ino_t      d_ino;        // inode number
    off_t      d_off;        // offset to the next dirent
    unsigned short d_reclen; // record length
    unsigned char d_type;    // file type
    char *      d_name;      // file name
}
```

Directory Structure

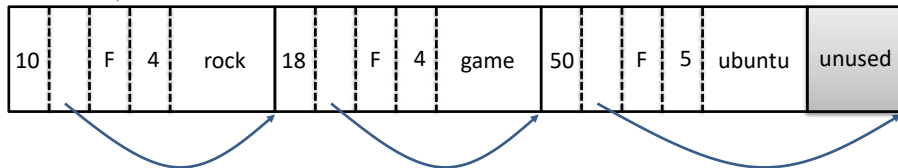
inode number

Entry size

Type

File name length

A Linux directory with three files



```
struct dirent {  
    ino_t      d_ino;      // inode number  
    off_t      d_off;      // offset to the next dirent  
    unsigned short d_reclen; // record length  
    unsigned char d_type;   // file type  
    char *      d_name;    // file name  
}
```


Directory Structure

inode number

Entry size

Type

File name length

A Linux directory with three files

10	F	4	rock	18	F	4	game	50	F	5	ubuntu	unused
----	---	---	------	----	---	---	------	----	---	---	--------	--------



After game has been removed

10	F	4	rock	unused	50	F	5	ubuntu	unused
----	---	---	------	--------	----	---	---	--------	--------

Accessing Directory File

- How to access directory file?

Note: `opendir()`, `readdir()`, and `closedir()` are library function calls.

```
1  int main(void) {  
2      DIR * dir;  
3      struct dirent *entry;  
4  
5      dir = opendir("/");  
6  
7      while ( (entry = readdir(dir)) != NULL) {  
8          // print the directory name  
9          printf("%s\n", entry->d_name);  
10     }  
11  
12     closedir(dir);  
13     return 0;  
14 }
```

Open the directory file.

Read the directory entries one by one until there is not further entries.

Close the directory file.

Details of Ext2/3

- Layout
- Inode and directory structure
- **Link file**
- Buffer cache
- Journaling
- VFS

Link File

- Can we allow a file to have multiple names and be accessed by several paths?
- How to create shortcuts?

Example use in Linux

```
# ls /dir1/12.jpg
12.jpg
# ln /dir1/12.jpg /my_link
# _
```

```
# ls /dir1/12.jpg
12.jpg
# ln -s /dir1/12.jpg /my_link
# _
```

These are called **hard link** and **symbolic link**

Link File – what is a hard link?

- A **hard link is a directory entry** pointing to an existing file.
 - **No new file content is created!**

```
# ls /dir1/12.jpg
12.jpg
# ln /dir1/12.jpg /my_link
# _
```

A new directory entry
is created.

Directory: /dir1

Inode #	...	Filename
123
2
5,086	...	12.jpg

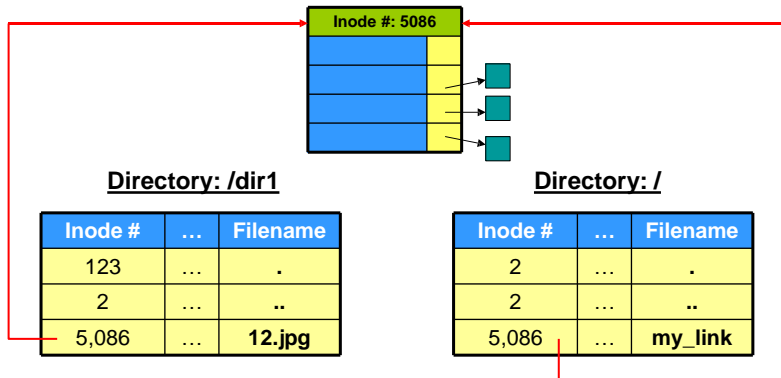
Directory: /

Inode #	...	Filename
2
2
5,086	...	my_link

Link File – what is a hard link?

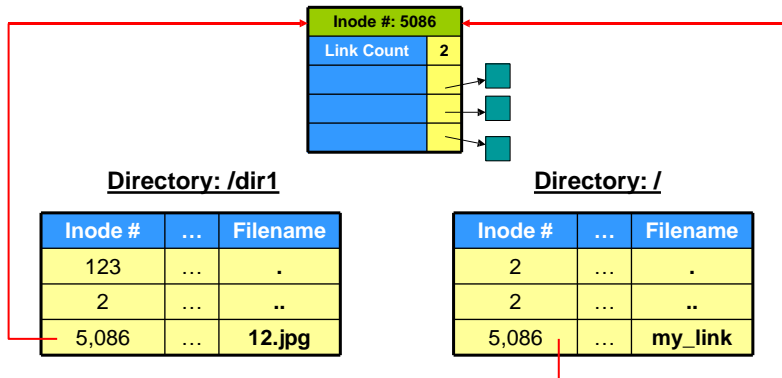
- Conceptually speaking, this **creates a file with two pathnames.**

How to maintain this info.



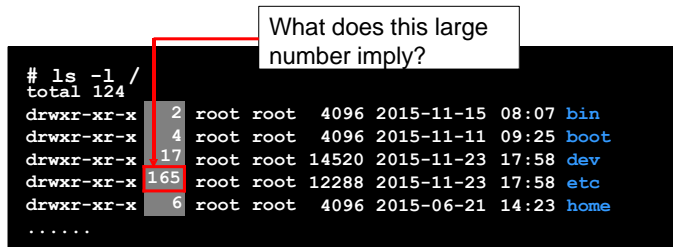
Link File – what is a link count?

- There is a field called **link count** in an inode.
 - It stores the **number of directory entries** pointing to the inode.



Link File – showing the link counts

- Special hard links
 - The directory “.” is a hard link to itself.
 - The directory “..” is a hard link to the parent directory.



A terminal window showing the command `# ls -l /` and its output. The output lists several directories with their permissions, link counts, owners, sizes, timestamps, and names. The `/etc` directory is highlighted with a red box around its link count of 165. A red line connects this box to a callout box containing the text "What does this large number imply?".

Permissions	Link Count	Owner	Group	Size	Timestamp	Name
drwxr-xr-x	2	root	root	4096	2015-11-15 08:07	bin
drwxr-xr-x	4	root	root	4096	2015-11-11 09:25	boot
drwxr-xr-x	17	root	root	14520	2015-11-23 17:58	dev
drwxr-xr-x	165	root	root	12288	2015-11-23 17:58	etc
drwxr-xr-x	6	root	root	4096	2015-06-21 14:23	home

This implies “/etc” has a lot of sub-directories.

Link File – showing the link counts

- Special hard links
 - The directory “.” is a hard link to itself.
 - The directory “..” is a hard link to the parent directory.
- What is the value of the link count, if
 - A **file** is created
 - A **directory** is created

Link File – showing the link counts

- When a regular file is created, the link count is **always 1**

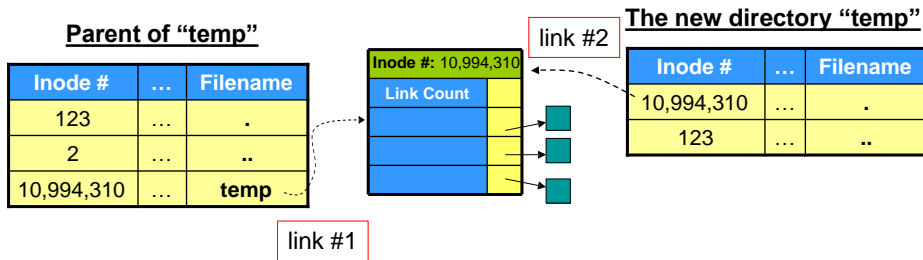
```
# stat Makefile
File: `Makefile'
Size: 4552          Blocks: 16          IO Block: 4096   regular file
Device: 801h/2049d Inode: 30669       Links: 1
.....
```

- When a directory is created, **the initial link count is always 2**

```
# mkdir temp
# stat temp
File: `temp'
Size: 4096          Blocks: 8          IO Block: 4096   directory
Device: 804h/2052d Inode: 10994310    Links: 2
.....
```

Why it is 2

Link File – showing the link counts



- When a directory is created, **the initial link count is always 2**. Why?

```
# mkdir temp
# stat temp
  File: `temp'
  Size: 4096          Blocks: 8          IO Block: 4096   directory
Device: 804h/2052d   Inode: 10994310    Links: 2
.....
```

Link File – showing the link counts

Parent of “temp”

Inode #	...	Filename
123
2
10,994,310	...	temp

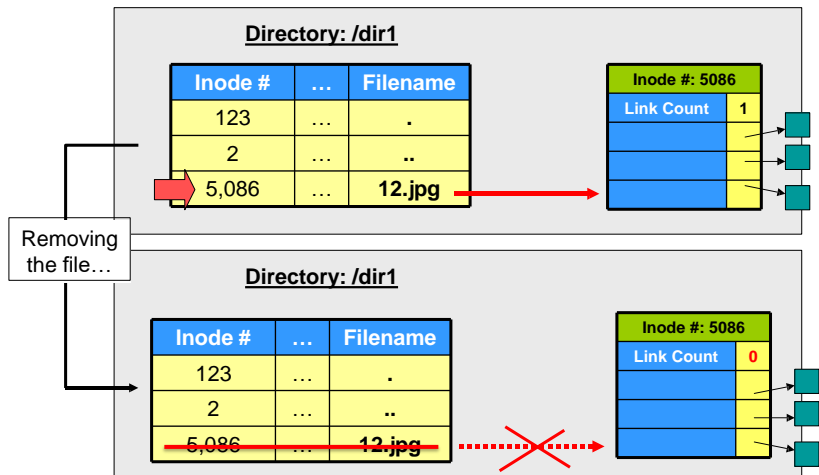
The new directory “temp”

Inode #	...	Filename
10,994,310
123

- The hosting directory of the newly creating directory will have its **link count increased by 1**.

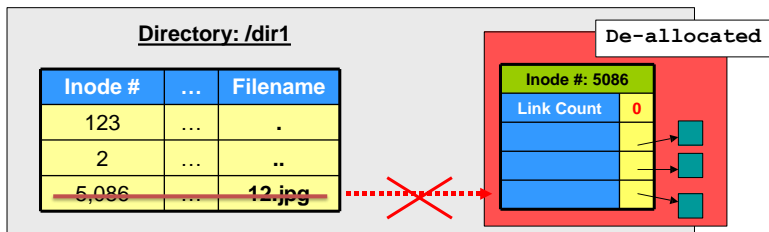
Link File – decrementing the link count?

- How about removing a file?



Link File – decrementing the link count?

- How about removing a file?
 - The system call that removing a file is, therefore, called **unlink()**.
 - The **unlink()** system call is to decrement the link count by **exactly one**.
 - When the **link count == 0**, the **data blocks** and the **inode** will all be de-allocated by the kernel.



Link File – decrementing the link count?

- Back to the previous hard link example...

Directory: /dir1

Inode #	...	Filename
123
2
5,086	...	12.jpg

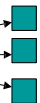

```
# ls /dir1/12.jpg
12.jpg
# ln /dir1/12.jpg /my_link
```

Directory: /

Inode #	...	Filename
2
2
5,086	...	my_link

Inode #: 5086

Link Count	2



Link File – decrementing the link count?

- Back to the previous hard link example...

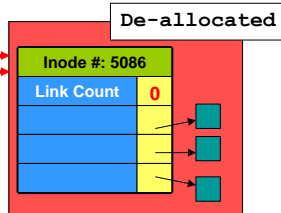
Directory: /dir1

Inode #	...	Filename
123
2
5,086	...	12.jpg

```
# ls /dir1/12.jpg
12.jpg
# ln /dir1/12.jpg /my_link
# rm /dir/12.jpg
# rm /my_link
```

Directory: /

Inode #	...	Filename
2
2
5,086	...	my_link



Link File – what is a symbolic link?

- A symbolic link **is a file**.
 - Unlike the hard link, **a new inode is created** for each symbolic link.
 - It stores the pathname (shortcut)

```
# ls /dir1/12.jpg
12.jpg
# ln -s /dir1/12.jpg /my_link
# ls -l /mylink
/mylink -> /dir1/12.jpg
#
```

A new directory entry is created.

Directory: /dir1

Inode #	...	Filename
123
2
5,086	...	12.jpg

Directory: /

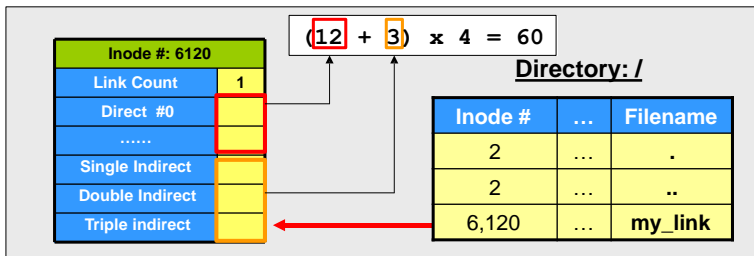
Inode #	...	Filename
2
2
6,120	...	my_link

Another
inode



Link File – what is a symbolic link?

- How to store the target path?
 - If the pathname is less than 60 characters
 - It is stored in the **12 direct block and the 3 indirect block pointers.**
 - Else, **one extra data block** is allocated



Short summary

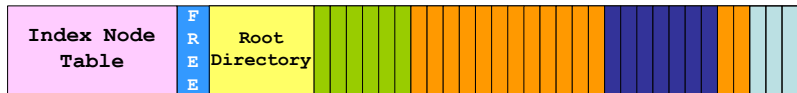
- Hard link
 - **A directory entry** pointing to an existing file
 - They point to the same inode (no new file content)
 - A file with two pathname
 - Remove file == unlink (link count - 1)
 - Examples: dot/dot dot
- Symbolic link
 - **A file with a new inode**
 - Stores the target pathname
 - Shortcuts

Details of Ext2/3

- Layout
- Inode and directory structure
- Link file
- **Buffer cache**
- Journaling
- VFS

File system performance

- Recall the read/write process
 - Directory traversal
 - Reading inode
 - Data blocks



How to improve file system performance?

Kernel Buffer Cache

- Kernel Buffer Cache
 - The kernel will keep a set of copies of the read/written data blocks.
 - The space that stores those blocks are called the **buffer cache**.
 - It is used for **reducing the time** in accessing those blocks in the near future
- Why effective?
 - **Principle of locality**

Kernel Buffer Cache

- What need to be cached?
 - Data blocks, directory file, inode?
 - All of them can benefit from caching



Kernel Buffer Cache

- Three types of buffer caches!

Page Cache	It buffers the data blocks of an opened file.
Directory entry (dcache) cache	Directory entry is stored in the kernel.
Inode cache	The content of an inode is stored in the kernel temporary.

Remember, those cached data is stored in the kernel even though the **corresponding file is closed!**

By the way, the cache is managed under the LRU algorithm.

Kernel Buffer Cache

Read/write mode with kernel buffer cache

Mode	Description
Reading mode	When a process reads a file, the data will be cached automatically. E.g., Readahead system call

Ways	Descriptions
System call	<code>ssize_t readahead(int fd, off64_t offset, size_t count);</code> A <u>blocking system call</u> that stores requested range of data into the kernel page caches Later <code>read()</code> calls over the range will not block .

Readahead

- How does it work?
 - When a file reading operation is requesting for **Block x**, there is a **chance** that **Block x+1** will also be needed.
 - Such a chance depends on:
 - The file reading mode: sequential access or random access.
 - The file reading history: whether the process **prefers** reading sequentially or not.
 - If such a chance is high, then reading a series of continuous blocks will **reduce the number of disk accesses**. Why?
 - Because the disk head is not always stopped at your desired locations.
 - Because a mechanical disk is good at reading sequential data.
 - How about SSD?

Kernel Buffer Cache

Read/write mode with kernel buffer cache

How about write?

Mode	Description
Write-through mode	<p>Both the <u>on-disk</u> and the <u>cached</u> copies update together.</p> <p>E.g., The write() system call will not return until the on-disk copy is written.</p>
Write-back mode	<p>When a piece of data is going to be written to a file, the cached copy is updated first. The update of the on-disk copy is delayed.</p> <p><u>On-demand writing dirty blocks back.</u></p> <p>Command: sync System calls: sync(), fsync()</p>

Details of Ext2/3

- Layout
- Inode and directory structure
- Link file
- Buffer cache
- **Journaling**
- VFS

File System Consistency

- Think about caching...tradeoff?
 - System inconsistency exists
 - Power failure, pressing reset button accidentally; etc.
- Disk only provides
 - **atomic** write of **one sector at a time**
- A write may require modifying several sectors
 - How to atomically update file system from one consistent state to another?

The **file system journal** is the current, state-of-the-art practice.

Example: Journaling File System

You write down all the tasks assigned to you into a log book.



Task list:

- 1) Buy boss a DC.
- 2) Pick up boss' friend.
- 3) Drive his friend back to his home.
- 4) Buy boss a coffee when I return.



Your boss orders you to do a set of tasks!

Example: Journaling File System



Task list:

- 1) ~~Buy boss a DC.~~
- 2) Pick up boss' friend.
- 3) Drive his friend back to his home.
- 4) Buy boss a coffee when I return.

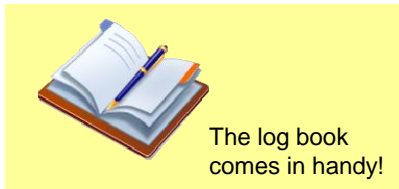
You cross out a task when it is completed.



Example: Journaling File System

Unfortunately, a car accident happens!

You lost all your memory!!



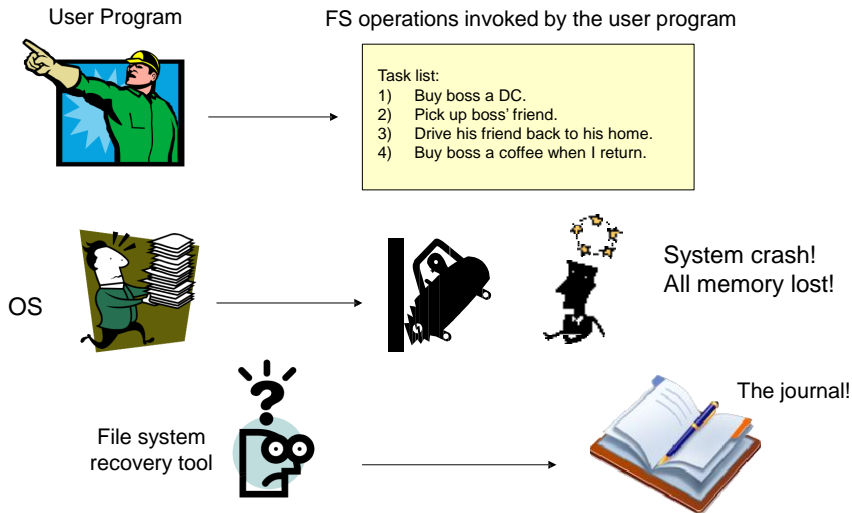
The log book comes in handy!



Your boss sends your colleague to finish your job. But, **he doesn't know about your progress.**

Worse, your boss has **forgotten** what are the tasks given to you!

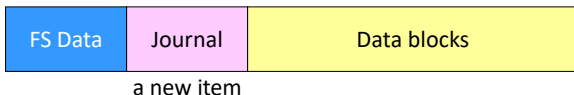
Example: Journaling File System



What is journal?



- A journal is the log book for the file system.
 - It is kept inside the file system, i.e., inside the disk.



- In database: **Write-ahead logging**
- In file systems: **Journaling**
 - Applications: Linux ext3 and ext4, Windows NTFS

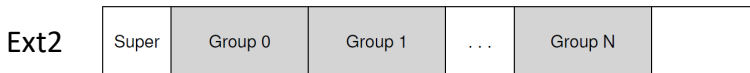
Basic idea: when updating the disk, before overwriting the structures in place, first write down **a little note** describing what you are about to do

What is journal?

- In order to make use of the journal:
 - A set of file system operations becomes an atomic **transaction**.
 - Either all operations are completed successfully, or
 - no operation is completed.
 - A transaction marks all the changes that **will be done** to the FS.
 - Every transaction is written to the journal.

Journaling in Linux ext3

- How does Linux ext3 incorporate the journaling?
 - Most of on-disk structures are identical to Linux ext2
 - The new key structure is the journal itself
 - It occupies some small amount of space within the partition or on another device

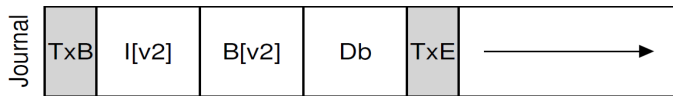


Data Journaling

- How to do journaling?
- **Task:** update inode ($I[v2]$), bitmap ($B[v2]$), and data block (Db) to disk
 - Metadata + data
- **Strategy: Data journaling**
 - Write all data (metadata+data) to journal
 - Before writing them to their final disk locations, we first write them to log (a.k.a. journal)
 - An available mode with the Linux ext3 file system

Data Journaling

- **Journal layout:**



- TxB: Transaction begin block

- It contains some kind of **transaction identifier (TID)**

- TxE: Transaction end block

- Marker of the end of this transaction
- It also contain the TID

- **Checkpoint**

- Overwrite the old structures in the file system after the transaction being safely on disk

Data Journaling

- Operation sequence:
 - **Journal write**
 - Write the transaction to log and wait for these writes to complete
 - TxB, all pending data, metadata updates, TxE
 - **Checkpoint**
 - Write the pending metadata and data updates to their final locations
- Any problem with this flow?
 - What if crash occurs during the writes to journal

Data Journaling

- We need to write the set of blocks (TxB, I[v2], B[v2], Db, TxE)
 - **Issue one block at a time**
 - It is slow because of waiting for each to complete
 - **Issue all blocks at once**
 - Five writes -> a single sequential write: Faster way
 - However, it is unsafe...
 - The disk internally may perform scheduling and complete small pieces of the big write **in any order**

Data Journaling

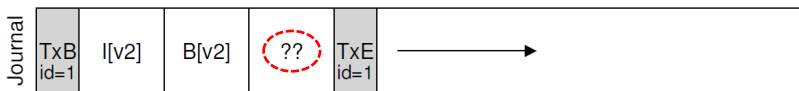
- **Issue all blocks at once**

- **Suppose:** disk internally

- (1) writes TxB, I[v2], B[v2], TxE and later
- (2) writes Db

- When crash occurs during the writes to journal

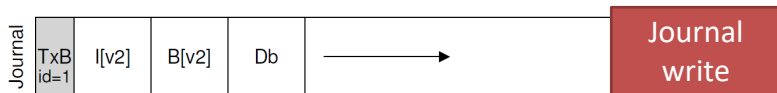
- If the disk loses power between (1) and (2)



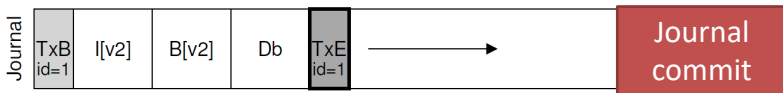
Problem: Transaction looks like a valid transaction, but
the file system can't look at the fourth block and know it is wrong

Data Journaling

- How to solve this problem?
 - Issue transactional write in two steps
 - **First step:** writes all blocks **except the TxE block** to journal



- **Second step:** file system issues the write of the TxE



Make sure the write of TxE is atomic

Data Journaling

- Operation sequence:
 - **Journal write**
 - Write the contents of the transaction (including TxB, metadata, and data)
 - **Journal commit**
 - metadata, and data (including TxE)
 - **Checkpoint**
 - Write the contents of the update to their on-disk locations

The write order must be guaranteed

Data Journaling

- How to do recovery?
 - Case 1: crash happens **before journal commit**

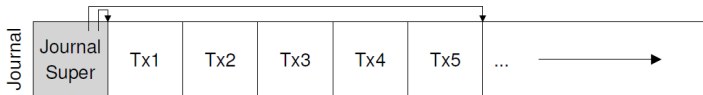
Easy! Skip the pending update

- Case 2: crash happens **after journal commit, but before checkpoint**

Replay transactions in order. Called **redo logging**

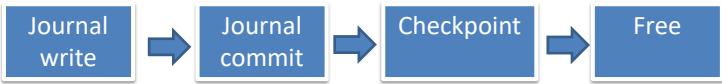
Data Journaling

- The log is of finite size
 - What problems may arise if it is full?
 - Long time to replay
 - Unable to append new transactions
- Manage as a **circular log**
 - **Free** space after checkpointing



Data Journaling

- Write sequence

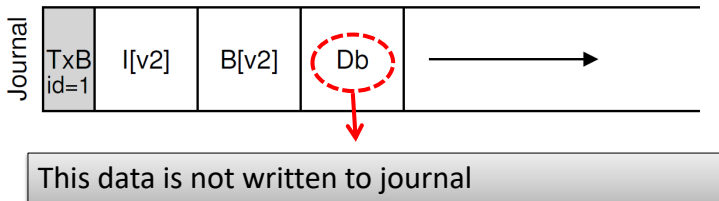


- Data Journaling Timeline

TxB	Journal Contents		TxE	File System	
	(metadata)	(data)		Metadata	Data
issue	issue	issue			
complete	complete				
		complete			
			issue		
			complete		
				issue	issue
				complete	complete

Metadata Journaling

- Any problem with data journaling?
 - Write every Db to disk **twice**
 - Commit to log (journal file)
 - Checkpoint to on-disk location
- How to avoid writing twice?
 - **Metadata journaling**: Logging metadata only

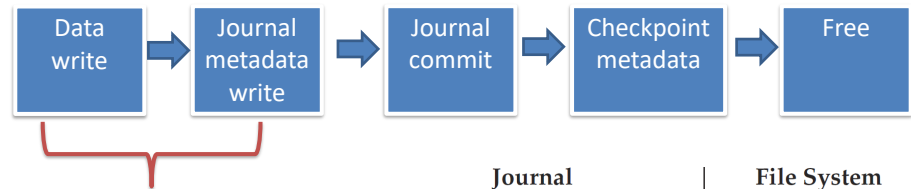


Metadata Journaling

- **Write-back mode**: no order restriction (data/journal)
 - How about data is written to disk after journal commit?
 - File system is consistent (from the perspective of metadata)
 - Metadata points to **garbage data**
- **Ordered mode**
 - Data is written to file system **before** journal commit
 - Rule:
 - *Write the pointed-to object before the object that points to it*
 - Core of crash consistency
 - Widely deployed by Ext3, NTFS, etc.

Metadata Journaling

- Write sequence



The two writes can be issued in parallel

TxB	Journal Contents (metadata)	TxE	File System	
			Metadata	Data
issue	issue			issue
				complete
complete	complete			
- - - - -	- - - - -	issue		
- - - - -	- - - - -	complete		
			issue	
			complete	

Summary on journal

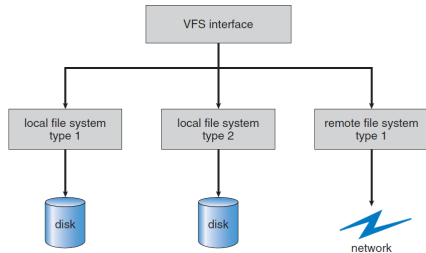
- Working principle:
 - All the changes to the FS are **written to the journal first**, including:
 - the changes in the metadata, i.e., information other than the file content. E.g., the inodes, the directory entries, etc.
 - the file data (depends on data journaling/metadata journaling)
 - Then, the system call returns to the user process.
 - Meanwhile, **the entries in the journal are replayed** and the changes are reflected to the actual file system.

Details of Ext2/3

- Layout
- Inode and directory structure
- Link file
- Buffer cache
- Journaling
- **VFS**

Virtual File System (VFS)

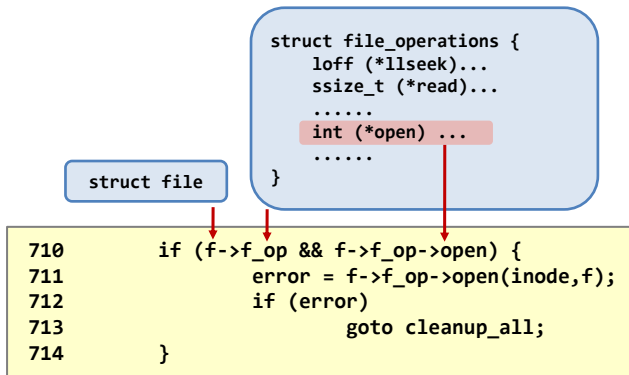
- Old days: “the” file system
- Nowadays: many fs types and instances co-exist



VFS: an FS abstraction layer

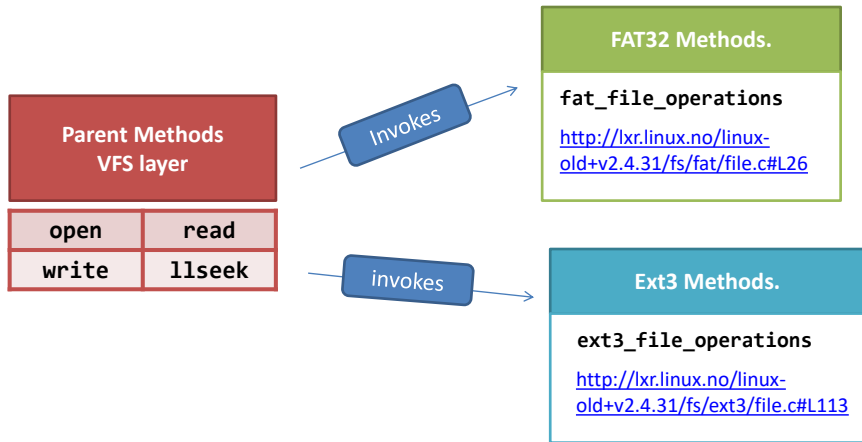
- Transparently and uniformly supports multiple FSES
- A VFS specifies an interface
- A specific FS implements this interface

- Let's look into the implementation of **open()**.



<http://lxr.linux.no/linux-old+v2.4.31/fs/open.c>

- For each file system, they have their own set of file operations.



- So, the beauty in such design is that:
 - The caller, i.e. the VFS layer, doesn't need to care about nor hard-coding which FS you are working on.

```
error = f->f_op->open(inode, f);
```

The only things that require hard-coding are:

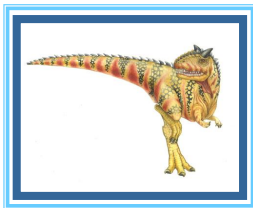
- The definition of the file operations.
- The assignment of file operation structures for each FS.

- A follow-up question is:
 - What if a FS does not support a particular subset of operations?
 - E.g., FAT32 does not need to implement **chmod()**!
 - Solution?
 - Simple! Using NULL pointers!
 - When a NULL pointer to a file is detected, returning an error or proceed without any changes.

Summary

- Ext* file systems are the primary FS for Linux
 - They follow the index-node allocation
 - We talked about...
 - Detailed layout (grouping, bitmaps)
 - Inode structure
 - Directory structure
 - Link file (hard link and symbolic link)
 - Kernel buffer cache and readahead
 - Journaling (data journaling, metadata journaling)
 - VFS

I/O Systems





Overview

I/O management is a major component of operating system design

- Important aspect of computer operation

- I/O devices vary greatly

- Various methods to control them

- Performance management

Ports, busses, device controllers connect to various devices

Device drivers encapsulate device details

- Present uniform device-access interface to I/O subsystem





I/O Hardware

Incredible variety of I/O devices

Storage

Transmission

Human-interface

Common concepts

Port – connection point for device

Bus - **daisy chain** or shared direct access

- ▶ **PCI** bus common in PCs and servers, PCI Express (**PCle**)
- ▶ **expansion bus** connects relatively slow devices

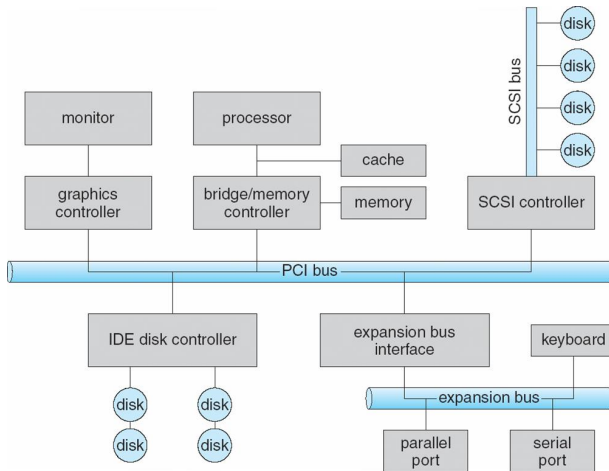
Controller (**host adapter**) – electronics that operate port, bus, device

- ▶ Sometimes integrated
- ▶ Sometimes separate circuit board (host adapter)





A Typical PC Bus Structure





I/O Hardware

How to control devices?

Devices usually have registers where device driver places commands, addresses, and data to write, or read data from registers after command execution

Data-in register, data-out register, status register, control register

How to communicate with controller?

Devices have addresses, used by

- ▶ Direct I/O instructions
- ▶ **Memory-mapped I/O**
 - Device data and command registers mapped to processor address space





Device I/O Port Locations on PCs (partial)

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)





Polling (轮询)

For **each byte** of I/O

1. Read busy bit from status register until 0
2. Host sets read or write bit and if write copies data into data-out register
3. Host sets command-ready bit
4. Controller sets busy bit, executes transfer
5. Controller clears busy bit, error bit, command-ready bit when transfer done

Step 1 is **busy-wait** cycle to wait for I/O from device

Reasonable if device is fast

But inefficient if device is slow





Interrupts (中断)

CPU **Interrupt-request line** triggered by I/O device

Two lines:

- ▶ **Maskable (可屏蔽)** and **nonmaskable (非屏蔽) interrupt**

Checked by processor after each instruction

Interrupt handler receives interrupts

Interrupt vector (中断向量) to dispatch interrupt to correct handler

Context switch at start and end

Based on priority, some are **nonmaskable**

Interrupt chaining if more than one device at same interrupt number





Intel Pentium Processor Event-Vector Table

非屏蔽中断

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts





Interrupts (Cont.)

Interrupt mechanism also used for **exceptions** (异常)

Terminate process, crash system due to hardware error

Page fault

executes when memory access error

System call

executes via **software interrupt** or **trap** to trigger kernel to execute request





Direct Memory Access

Used to avoid **programmed I/O** (one byte at a time) (**程序控制I/O**) for large data movement

Requires **DMA** controller

Bypasses CPU to transfer data directly between device & memory

How to work?

OS writes DMA command block into memory

- ▶ Source and destination addresses
- ▶ Read or write mode
- ▶ Count of bytes

Writes location of command block to DMA controller, then CPU can continue to execute other tasks

DMA controller masters bus and does the transmission without CPU

- ▶ DMA-request and DMA acknowledge between DMA controller and device controller





Application I/O Interface

Devices vary in many dimensions

Character-stream or **block**

Sequential or **random-access**

Synchronous or **asynchronous**

Sharable or **dedicated**

Speed of operation

read-write, read only, write only

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read-write	CD-ROM graphics controller disk

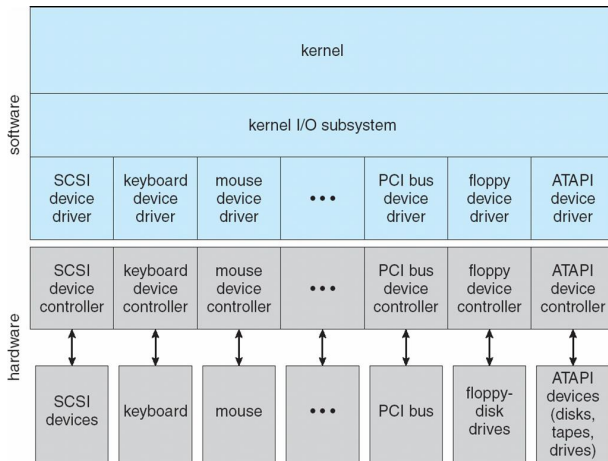
How to provide a standard and uniform I/O interface?

Abstraction, encapsulation, layering (抽象, 封装, 分层)





A Kernel I/O Structure





I/O Devices

Block devices include disk drives

Commands include **read**, **write**, **seek**

Raw I/O, **direct I/O**, or file-system access

Memory-mapped file access possible

- ▶ File mapped to virtual memory and clusters brought via demand paging

DMA

Character devices include keyboards, mice, serial ports

Commands include `get()`, `put()`

Network devices

socket interface





Clocks and Timers

Functionalities of hardware clock and timer

- Get current time

- Get elapsed time

- Timer

Programmable interval timer (可编程间隔定时器) used for timings, periodic interrupts

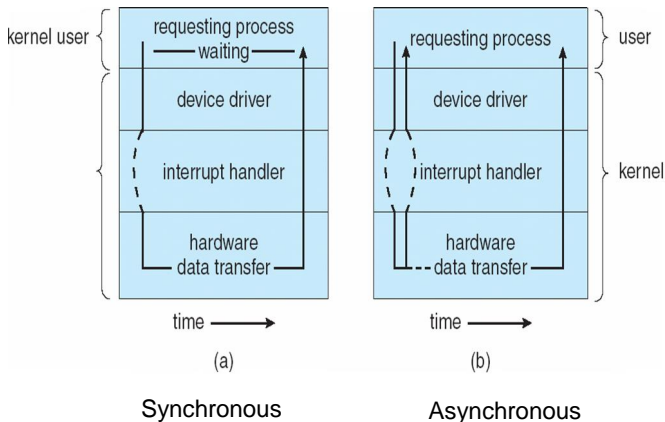
- Process scheduler: interrupt when time quantum is zero

- I/O subsystem: periodic flush





Two I/O Methods





Kernel I/O Subsystem

Kernel I/O subsystem provides many services

I/O scheduling

- Maintain a per-device queue

- Re-ordering the requests

- Average waiting time, fairness, etc.

Buffering - store data in memory while transferring between devices

- To cope with device speed mismatch

- To cope with device transfer size mismatch

- To maintain “copy semantics” (e.g., copy from application’s buffer to kernel buffer)





Kernel I/O Subsystem

Caching - faster device holding copy of data

- Always just a copy

- Key to performance

- Sometimes combined with buffering

Spooling - hold output for a device

- If device can serve only one request at a time, e.g., Printing

Error handling and **I/O protection**

- OS can recover from disk read error, device unavailable, transient write failures

- All I/O instructions defined to be privileged

Power management, etc.





Summary

I/O hardware

Port, bus, controller

Polling, interrupt, DMA

Application I/O interface

block devices, character devices, network devices, clock and timer

Kernel I/O subsystem

Services



Hardware White Paper

Designing Hardware for Microsoft® Operating Systems

Microsoft Extensible Firmware Initiative FAT32 File System Specification

FAT: General Overview of On-Disk Format

Version 1.03, December 6, 2000
Microsoft Corporation

The FAT (File Allocation Table) file system has its origins in the late 1970s and early 1980s and was the file system supported by the Microsoft® MS-DOS® operating system. It was originally developed as a simple file system suitable for floppy disk drives less than 500K in size. Over time it has been enhanced to support larger and larger media. Currently there are three FAT file system types: FAT12, FAT16 and FAT32. The basic difference in these FAT sub types, and the reason for the names, is the size, in bits, of the entries in the actual FAT structure on the disk. There are 12 bits in a FAT12 FAT entry, 16 bits in a FAT16 FAT entry and 32 bits in a FAT32 FAT entry.

Contents

Notational Conventions in this Document	7
General Comments (Applicable to FAT File System All Types)	7
Boot Sector and BPB	7
FAT Data Structure	13
FAT Type Determination	14
FAT Volume Initialization	19
FAT32 FSInfo Sector Structure and Backup Boot Sector	21
FAT Directory Structure	22
FAT Long Directory Entries	25
Name Limits and Character Sets	29
Name Matching In Short & Long Names	30
Naming Conventions and Long Names	30
Effect of Long Directory Entries on Down Level Versions of FAT	32
Validating The Contents of a Directory	32
Other Notes Relating to FAT Directories	33

Microsoft, MS-DOS, Windows, and Windows NT are trademarks or registered trademarks of Microsoft Corporation in the United States and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners.

© 2000 Microsoft Corporation. All rights reserved.

Microsoft Extensible Firmware Initiative FAT32 File System Specification

IMPORTANT-READ CAREFULLY: This Microsoft Agreement ("Agreement") is a legal agreement between you (either an individual or a single entity) and Microsoft Corporation ("Microsoft") for the version of the Microsoft specification identified above which you are about to download ("Specification"). BY DOWNLOADING, COPYING OR OTHERWISE USING THE SPECIFICATION, YOU AGREE TO BE BOUND BY THE TERMS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO THE TERMS OF THIS AGREEMENT, DO NOT DOWNLOAD, COPY, OR USE THE SPECIFICATION.

The Specification is owned by Microsoft or its suppliers and is protected by copyright laws and international copyright treaties, as well as other intellectual property laws and treaties.

1. LIMITED LICENSE AND COVENANT NOT TO SUE.

(a) Provided that you comply with all terms and conditions of this Agreement and subject to the limitations in Sections 1(c) - (f) below, Microsoft grants to you the following non-exclusive, worldwide, royalty-free, non-transferable, non-sublicenseable license under any copyrights owned or licensable by Microsoft without payment of consideration to unaffiliated third parties, to reproduce the Specification solely for the purposes of creating portions of products which comply with the Specification in unmodified form.

(b) Provided that you comply with all terms and conditions of this Agreement and subject to the limitations in Sections 1(c) - (f) below, Microsoft grants to you the following non-exclusive, worldwide, royalty-free, non-transferable, non-sublicenseable, reciprocal limited covenant not to sue under its Necessary Claims solely to make, have made, use, import, and directly and indirectly, offer to sell, sell and otherwise distribute and dispose of portions of products which comply with the Specification in unmodified form.

For purposes of sections (a) and (b) above, the Specification is "unmodified" if there are no changes, additions or extensions to the Specification, and "Necessary Claims" means claims of a patent or patent application which are (1) owned or licensable by Microsoft without payment of consideration to an unaffiliated third party, and (2) have an effective filing date on or before December 31, 2010, that must be infringed in order to make a portion(s) of a product that complies with the Specification. Necessary Claims does not include claims relating to semiconductor manufacturing technology or microprocessor circuits or claims not required to be infringed in complying with the Specification (even if in the same patent as Necessary Claims).

(c) The foregoing covenant not to sue shall not extend to any part or function of a product which (i) is not required to comply with the Specification in unmodified form, or (ii) to which there was a commercially reasonable alternative to infringing a Necessary Claim.

(d) Each of the license and the covenant not to sue described above shall be unavailable to you and shall terminate immediately if you or any of your Affiliates (collectively "Covenantant Party") "Initiates" any action for patent infringement against: (x) Microsoft or any of its Affiliates (collectively "Granting Party"), (y) any customers or distributors of the Granting Party, or other recipients of a covenant not to sue with respect to the Specification from the Granting Party ("Covenantees"); or (z) any customers or distributors of Covenantees (all parties identified in (y) and (z) collectively referred to as "Customers"), which action is based on a conformant implementation of the Specification. As used herein, "Affiliate" means any entity which directly or indirectly controls, is controlled by, or is under common control with a party; and control shall mean the power, whether direct or indirect, to direct or cause the direction of the management or policies of any entity whether through the ownership of voting securities, by contract or otherwise. "Initiates" means that a Covenantant Party is the first (as between the Granting Party and the Covenantant Party) to file or institute any legal or administrative claim or action for patent infringement against the Granting Party or any of the Customers. "Initiates" includes any situation in which a Covenantant Party files or initiates a legal or administrative claim or action for patent

infringement solely as a counterclaim or equivalent in response to a Granting Party first filing or instituting a legal or administrative patent infringement claim against such Covenantant Party.

(e) Each of the license and the covenant not to sue described above shall not extend to your use of any portion of the Specification for any purpose other than (a) to create portions of an operating system (i) only as necessary to adapt such operating system so that it can directly interact with a firmware implementation of the Extensible Firmware Initiative Specification v. 1.0 ("EFI Specification"); (ii) only as necessary to emulate an implementation of the EFI Specification; and (b) to create firmware, applications, utilities and/or drivers that will be used and/or licensed for only the following purposes: (i) to install, repair and maintain hardware, firmware and portions of operating system software which are utilized in the boot process; (ii) to provide to an operating system runtime services that are specified in the EFI Specification; (iii) to diagnose and correct failures in the hardware, firmware or operating system software; (iv) to query for identification of a computer system (whether by serial numbers, asset tags, user or otherwise); (v) to perform inventory of a computer system; and (vi) to manufacture, install and setup any hardware, firmware or operating system software.

(f) Microsoft reserves all other rights it may have in the Specification and any intellectual property therein. The furnishing of this document does not give you any license or covenant not to sue with respect to any other Microsoft patents, trademarks, copyrights or other intellectual property rights.

2. ADDITIONAL LIMITATIONS AND OBLIGATIONS.

(a) The foregoing license and covenant not to sue is applicable only to the version of the Specification which you are about to download. It does not apply to any additional versions or extensions to the Specification.

(b) Without prejudice to any other rights, Microsoft may terminate this Agreement if you fail to comply with the terms and conditions of this Agreement. In such event you must destroy all copies of the Specification.

3. **INTELLECTUAL PROPERTY RIGHTS.** All ownership, title and intellectual property rights in and to the Specification are owned by Microsoft or its suppliers.

4. **U.S. GOVERNMENT RIGHTS.** Any Specification provided to the U.S. Government pursuant to solicitations issued on or after December 1, 1995 is provided with the commercial rights and restrictions described elsewhere herein. Any Specification provided to the U.S. Government pursuant to solicitations issued prior to December 1, 1995 is provided with RESTRICTED RIGHTS as provided for in FAR, 48 CFR 52.227-14 (JUNE 1987) or DFAR, 48 CFR 252-227-7013 (OCT 1988), as applicable.

5. **EXPORT RESTRICTIONS.** Export of the Specification, any part thereof, or any process or service that is the direct product of the Specification (the foregoing collectively referred to as the "Restricted Components") from the United States is regulated by the Export Administration Regulations (EAR, 15 CFR 730-744) of the U.S. Commerce Department, Bureau of Export Administration ("BXA"). You agree to comply with the EAR in the export or re-export of the Restricted Components (i) to any country to which the U.S. has embargoed or restricted the export of goods or services, which currently include, but are not necessarily limited to Cuba, Iran, Iraq, Libya, North Korea, Sudan, Syria and the Federal Republic of Yugoslavia (including Serbia, but not Montenegro), or to any national of any such country, wherever located, who intends to transmit or transport the Restricted Components back to such country; (ii) to any person or entity who you know or have reason to know will utilize the Restricted Components in the design, development or production of nuclear, chemical or biological weapons; or (iii) to any person or entity who has been prohibited from participating in U.S. export transactions by any federal agency of the U.S. government. You warrant and represent that neither the BXA nor any other U.S. federal agency has suspended, revoked or denied your export privileges. For additional information see <http://www.microsoft.com/exporting>.

6. **DISCLAIMER OF WARRANTIES.** To the maximum extent permitted by applicable law, Microsoft and its suppliers provide the Specification (and all intellectual property therein) and any (if any) support services related to the Specification ("Support Services") AS IS AND WITH ALL FAULTS, and hereby disclaim all warranties and conditions, either express, implied or statutory, including, but not limited to, any (if any) implied warranties or conditions of merchantability, of fitness for a particular purpose, of lack of viruses, of accuracy or completeness of responses, of results, and of lack of negligence or lack of workmanlike effort, all with regard to the Specification, any intellectual property therein and the provision of or failure to provide Support Services. ALSO, THERE IS NO WARRANTY OR CONDITION OF TITLE, QUIET ENJOYMENT, QUIET POSSESSION, CORRESPONDENCE TO DESCRIPTION OR NON-INFRINGEMENT, WITH REGARD TO THE SPECIFICATION AND ANY INTELLECTUAL PROPERTY THEREIN. THE ENTIRE RISK AS TO THE QUALITY OF OR ARISING OUT OF USE OR PERFORMANCE OF THE SPECIFICATION, ANY INTELLECTUAL PROPERTY THEREIN, AND SUPPORT SERVICES, IF ANY, REMAINS WITH YOU.

7. **EXCLUSION OF INCIDENTAL, CONSEQUENTIAL AND CERTAIN OTHER DAMAGES.** To the maximum extent permitted by applicable law, in no event shall Microsoft or its suppliers be liable for any special, incidental, indirect, or consequential damages whatsoever (including, but not limited to, damages for loss of profits or confidential or other information, for business interruption, for personal injury, for loss of privacy, for failure to meet any duty including of good faith or of reasonable care, for negligence, and for any other pecuniary or other loss whatsoever) arising out of or in any way related to the use of or inability to use the SPECIFICATION, ANY INTELLECTUAL PROPERTY THEREIN, the provision of or failure to provide Support Services, or otherwise under or in connection with any provision of this AGREEMENT, even in the event of the fault, tort (including negligence), strict liability, breach of contract or breach of warranty of Microsoft or any supplier, and even if Microsoft or any supplier has been advised of the possibility of such damages.

8. **LIMITATION OF LIABILITY AND REMEDIES.** Notwithstanding any damages that you might incur for any reason whatsoever (including, without limitation, all damages referenced above and all direct or general damages), the entire liability of Microsoft and any of its suppliers under any provision of this Agreement and your exclusive remedy for all of the foregoing shall be limited to the greater of the amount actually paid by you for the Specification or U.S.\$5.00. The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails its essential purpose.

9. **APPLICABLE LAW.** If you acquired this Specification in the United States, this Agreement is governed by the laws of the State of Washington. If you acquired this Specification in Canada, unless expressly prohibited by local law, this Agreement is governed by the laws in force in the Province of Ontario, Canada; and, in respect of any dispute which may arise hereunder, you consent to the jurisdiction of the federal and provincial courts sitting in Toronto, Ontario. If this Specification was acquired outside the United States, then local law may apply.

10. **QUESTIONS.** Should you have any questions concerning this Agreement, or if you desire to contact Microsoft for any reason, please contact the Microsoft subsidiary serving your country, or write: Microsoft Sales Information Center/One Microsoft Way/Redmond, WA 98052-6399.

11. **ENTIRE AGREEMENT.** This Agreement is the entire agreement between you and Microsoft relating to the Specification and the Support Services (if any) and they supersede all prior or contemporaneous oral or written communications, proposals and representations with respect to the Specification or any other subject matter covered by this Agreement. To the extent the terms of any Microsoft policies or programs for Support Services conflict with the terms of this Agreement, the terms of this Agreement shall control.

Si vous avez acquis votre produit Microsoft au CANADA, la garantie limitée suivante vous concerne :

RENONCIATION AUX GARANTIES. Dans toute la mesure permise par la législation en vigueur, Microsoft et ses fournisseurs fournissent la Spécification (et à toute propriété intellectuelle dans celle-ci) et tous (selon le cas) les services d'assistance liés à la Spécification ("Services d'assistance") TELS QUELS ET AVEC TOUS LEURS DÉFAUTS, et par les présentes excluent toute garantie ou condition, expresse ou implicite, légale ou conventionnelle, écrite ou verbale, y compris, mais sans limitation, toute (selon le cas) garantie ou condition implicite ou légale de qualité marchande, de conformité à un usage particulier, d'absence de virus, d'exactitude et d'intégralité des réponses, de résultats, d'efforts techniques et professionnels et d'absence de négligence, le tout relativement à la Spécification, à toute propriété intellectuelle dans celle-ci et à la prestation ou à la non-prestation des Services d'assistance. DE PLUS, IL N'Y A AUCUNE GARANTIE ET CONDITION DE TITRE, DE JOUISSANCE PAISIBLE, DE POSSESSION PAISIBLE, DE SIMILARITÉ À LA DESCRIPTION ET D'ABSENCE DE CONTREFAÇON RELATIVEMENT À LA SPÉCIFICATION ET À TOUTE PROPRIÉTÉ INTELLECTUELLE DANS CELLE-CI. VOUS SUPPORTEZ TOUS LES RISQUES DÉCOULANT DE L'UTILISATION ET DE LA PERFORMANCE DE LA SPÉCIFICATION ET DE TOUTE PROPRIÉTÉ INTELLECTUELLE DANS CELLE-CI ET CEUX DÉCOULANT DES SERVICES D'ASSISTANCE (S'IL Y A LIEU).

EXCLUSION DES DOMMAGES INDIRECTS, ACCESSOIRES ET AUTRES. Dans toute la mesure permise par la législation en vigueur, Microsoft et ses fournisseurs ne sont en aucun cas responsables de tout dommage spécial, indirect, accessoire, moral ou exemplaire quel qu'il soit (y compris, mais sans limitation, les dommages entraînés par la perte de bénéfices ou la perte d'information confidentielle ou autre, l'interruption des affaires, les préjudices corporels, la perte de confidentialité, le défaut de remplir toute obligation y compris les obligations de bonne foi et de diligence raisonnable, la négligence et toute autre perte pécuniaire ou autre perte de quelque nature que ce soit) découlant de, ou de toute autre manière lié à, l'utilisation ou l'impossibilité d'utiliser la Spécification, toute propriété intellectuelle dans celle-ci, la prestation ou la non-prestation des Services d'assistance ou autrement en vertu de ou relativement à toute disposition de cette convention, que ce soit en cas de faute, de délit (y compris la négligence), de responsabilité stricte, de manquement à un contrat ou de manquement à une garantie de Microsoft ou de l'un de ses fournisseurs, et ce, même si Microsoft ou l'un de ses fournisseurs a été avisé de la possibilité de tels dommages.

LIMITATION DE RESPONSABILITÉ ET RECOURS. Malgré tout dommage que vous pourriez encourir pour quelque raison que ce soit (y compris, mais sans limitation, tous les dommages mentionnés ci-dessus et tous les dommages directs et généraux), la seule responsabilité de Microsoft et de ses fournisseurs en vertu de toute disposition de cette convention et votre unique recours en regard de tout ce qui précède sont limités au plus élevé des montants suivants: soit (a) le montant que vous avez payé pour la Spécification, soit (b) un montant équivalent à cinq dollars U.S. (5,00 \$ U.S.). Les limitations, exclusions et renoncements ci-dessus s'appliquent dans toute la mesure permise par la législation en vigueur, et ce même si leur application a pour effet de priver un recours de son essence.

DROITS LIMITÉS DU GOUVERNEMENT AMÉRICAIN

Tout Produit Logiciel fourni au gouvernement américain conformément à des demandes émises le ou après le 1^{er} décembre 1995 est offert avec les restrictions et droits commerciaux décrits ailleurs dans la présente convention. Tout Produit Logiciel fourni au gouvernement américain conformément à des demandes émises avant le 1^{er} décembre 1995 est offert avec des DROITS LIMITÉS tels que prévus dans le FAR, 48CFR 52.227-14 (juin 1987) ou dans le FAR, 48CFR 252.227-7013 (octobre 1988), tels qu'applicables.

Sauf lorsqu'expressément prohibé par la législation locale, la présente convention est régie par les lois en vigueur dans la province d'Ontario, Canada. Pour tout différend qui pourrait découler des présentes, vous acceptez la compétence des tribunaux fédéraux et provinciaux siégeant à Toronto, Ontario.

Si vous avez des questions concernant cette convention ou si vous désirez communiquer avec Microsoft pour quelque raison que ce soit, veuillez contacter la succursale Microsoft desservant votre pays, ou écrire à: Microsoft Sales Information Center, One Microsoft Way, Redmond, Washington 98052-6399.

Notational Conventions in this Document

Numbers that have the characters “0x” at the beginning of them are hexadecimal (base 16) numbers.

Any numbers that do not have the characters “0x” at the beginning are decimal (base 10) numbers.

The code fragments in this document are written in the ‘C’ programming language. Strict typing and syntax are not adhered to.

There are several code fragments in this document that freely mix 32-bit and 16-bit data elements. It is assumed that you are a programmer who understands how to properly type such operations so that data is not lost due to truncation of 32-bit values to 16-bit values. Also take note that all data types are UNSIGNED. Do not do FAT computations with signed integer types, because the computations will be wrong on some FAT volumes.

General Comments (Applicable to FAT File System All Types)

All of the FAT file systems were originally developed for the IBM PC machine architecture. The importance of this is that FAT file system on disk data structure is all “little endian.” If we look at one 32-bit FAT entry stored on disk as a series of four 8-bit bytes—the first being byte[0] and the last being byte[4]—here is where the 32 bits numbered 00 through 31 are (00 being the least significant bit):

```
byte[3]      3 3 2 2 2 2 2 2
              1 0 9 8 7 6 5 4

byte[2]      2 2 2 2 1 1 1 1
              3 2 1 0 9 8 7 6

byte[1]      1 1 1 1 1 1 0 0
              5 4 3 2 1 0 9 8

byte[0]      0 0 0 0 0 0 0 0
              7 6 5 4 3 2 1 0
```

This is important if your machine is a “big endian” machine, because you will have to translate between big and little endian as you move data to and from the disk.

A FAT file system volume is composed of four basic regions, which are laid out in this order on the volume:

- 0 – Reserved Region
- 1 – FAT Region
- 2 – Root Directory Region (doesn’t exist on FAT32 volumes)
- 3 – File and Directory Data Region

Boot Sector and BPB

The first important data structure on a FAT volume is called the BPB (BIOS Parameter Block), which is located in the first sector of the volume in the Reserved Region. This sector is sometimes called the “boot sector” or the “reserved sector” or the “0th sector,” but the important fact is simply that it is the first sector of the volume.

This is the first thing about the FAT file system that sometimes causes confusion. In MS-DOS version 1.x, there was not a BPB in the boot sector. In this first version of the FAT file system, there were only two different formats, the one for single-sided and the one for double-sided 360K 5.25-inch

floppy disks. The determination of which type was on the disk was done by looking at the first byte of the FAT (the low 8 bits of FAT[0]).

This type of media determination was superseded in MS-DOS version 2.x by putting a BPB in the boot sector, and the old style of media determination (done by looking at the first byte of the FAT) was no longer supported. All FAT volumes must have a BPB in the boot sector.

This brings us to the second point of confusion relating to FAT volume determination: What exactly does a BPB look like? The BPB in the boot sector defined for MS-DOS 2.x only allowed for a FAT volume with strictly less than 65,536 sectors (32 MB worth of 512-byte sectors). This limitation was due to the fact that the “total sectors” field was only a 16-bit field. This limitation was addressed by MS-DOS 3.x, where the BPB was modified to include a new 32-bit field for the total sectors value.

The next BPB change occurred with the Microsoft Windows 95 operating system, specifically OEM Service Release 2 (OSR2), where the FAT32 type was introduced. FAT16 was limited by the maximum size of the FAT and the maximum valid cluster size to no more than a 2 GB volume if the disk had 512-byte sectors. FAT32 addressed this limitation on the amount of disk space that one FAT volume could occupy so that disks larger than 2 GB only had to have one partition defined.

The FAT32 BPB exactly matches the FAT12/FAT16 BPB up to and including the BPB_TotSec32 field. They differ starting at offset 36, depending on whether the media type is FAT12/FAT16 or FAT32 (see discussion below for determining FAT type). The relevant point here is that the BPB in the boot sector of a FAT volume should always be one that has all of the new BPB fields for either the FAT12/FAT16 or FAT32 BPB type. Doing it this way ensures the maximum compatibility of the FAT volume and ensures that all FAT file system drivers will understand and support the volume properly, because it always contains all of the currently defined fields.

NOTE: In the following description, all the fields whose names start with BPB_ are part of the BPB. All the fields whose names start with BS_ are part of the boot sector and not really part of the BPB. The following shows the start of sector 0 of a FAT volume, which contains the BPB:

Boot Sector and BPB Structure

Name	Offset (byte)	Size (bytes)	Description
BS_jmpBoot	0	3	<p>Jump instruction to boot code. This field has two allowed forms:</p> <p>jmpBoot[0] = 0xEB, jmpBoot[1] = 0x??, jmpBoot[2] = 0x90 and jmpBoot[0] = 0xE9, jmpBoot[1] = 0x??, jmpBoot[2] = 0x??</p> <p>0x?? indicates that any 8-bit value is allowed in that byte. What this forms is a three-byte Intel x86 unconditional branch (jump) instruction that jumps to the start of the operating system bootstrap code. This code typically occupies the rest of sector 0 of the volume following the BPB and possibly other sectors. Either of these forms is acceptable. JmpBoot[0] = 0xEB is the more frequently used format.</p>
BS_OEMName	3	8	<p>"MSWIN4.1" There are many misconceptions about this field. It is only a name string. Microsoft operating systems don't pay any attention to this field. Some FAT drivers do. This is the reason that the indicated string, "MSWIN4.1", is the recommended setting, because it is the setting least likely to cause compatibility problems. If you want to put something else in here, that is your option, but the result may be that some FAT drivers might not recognize the volume. Typically this is some indication of what system formatted the volume.</p>
BPB_BytsPerSec	11	2	<p>Count of bytes per sector. This value may take on only the following values: 512, 1024, 2048 or 4096. If maximum compatibility with old implementations is desired, only the value 512 should be used. There is a lot of FAT code in the world that is basically "hard wired" to 512 bytes per sector and doesn't bother to check this field to make sure it is 512. Microsoft operating systems will properly support 1024, 2048, and 4096.</p> <p>Note: Do not misinterpret these statements about maximum compatibility. If the media being recorded has a physical sector size N, you must use N and this must still be less than or equal to 4096. Maximum compatibility is achieved by only using media with specific sector sizes.</p>
BPB_SecPerClus	13	1	<p>Number of sectors per allocation unit. This value must be a power of 2 that is greater than 0. The legal values are 1, 2, 4, 8, 16, 32, 64, and 128. Note however, that a value should never be used that results in a "bytes per cluster" value ($\text{BPB_BytsPerSec} * \text{BPB_SecPerClus}$) greater than 32K ($32 * 1024$). There is a misconception that values greater than this are OK. Values that cause a cluster size greater than 32K bytes do not work properly; do not try to define one. Some versions of some systems allow 64K bytes per cluster value. Many application setup programs will not work correctly on such a FAT volume.</p>
BPB_RsvdSecCnt	14	2	<p>Number of reserved sectors in the Reserved region of the volume starting at the first sector of the volume. This field must not be 0. For FAT12 and FAT16 volumes, this value should never be anything other than 1. For FAT32 volumes, this value is typically 32. There is a lot of FAT code in the world "hard wired" to 1 reserved sector for FAT12 and FAT16 volumes and that doesn't bother to check this field to make sure it is 1. Microsoft operating systems will properly support any non-zero value in this field.</p>

BPB_NumFATs	16	1	<p>The count of FAT data structures on the volume. This field should always contain the value 2 for any FAT volume of any type. Although any value greater than or equal to 1 is perfectly valid, many software programs and a few operating systems' FAT file system drivers may not function properly if the value is something other than 2. All Microsoft file system drivers will support a value other than 2, but it is still highly recommended that no value other than 2 be used in this field.</p> <p>The reason the standard value for this field is 2 is to provide redundancy for the FAT data structure so that if a sector goes bad in one of the FATs, that data is not lost because it is duplicated in the other FAT. On non-disk-based media, such as FLASH memory cards, where such redundancy is a useless feature, a value of 1 may be used to save the space that a second copy of the FAT uses, but some FAT file system drivers might not recognize such a volume properly.</p>
BPB_RootEntCnt	17	2	For FAT12 and FAT16 volumes, this field contains the count of 32-byte directory entries in the root directory. For FAT32 volumes, this field must be set to 0. For FAT12 and FAT16 volumes, this value should always specify a count that when multiplied by 32 results in an even multiple of BPB_BytsPerSec. For maximum compatibility, FAT16 volumes should use the value 512.
BPB_TotSec16	19	2	This field is the old 16-bit total count of sectors on the volume. This count includes the count of all sectors in all four regions of the volume. This field can be 0; if it is 0, then BPB_TotSec32 must be non-zero. For FAT32 volumes, this field must be 0. For FAT12 and FAT16 volumes, this field contains the sector count, and BPB_TotSec32 is 0 if the total sector count "fits" (is less than 0x10000).
BPB_Media	21	1	0xF8 is the standard value for "fixed" (non-removable) media. For removable media, 0xF0 is frequently used. The legal values for this field are 0xF0, 0xF8, 0xF9, 0xFA, 0xFB, 0xFC, 0xFD, 0xFE, and 0xFF. The only other important point is that whatever value is put in here must also be put in the low byte of the FAT[0] entry. This dates back to the old MS-DOS 1.x media determination noted earlier and is no longer usually used for anything.
BPB_FATSz16	22	2	This field is the FAT12/FAT16 16-bit count of sectors occupied by ONE FAT. On FAT32 volumes this field must be 0, and BPB_FATSz32 contains the FAT size count.
BPB_SecPerTrk	24	2	Sectors per track for interrupt 0x13. This field is only relevant for media that have a geometry (volume is broken down into tracks by multiple heads and cylinders) and are visible on interrupt 0x13. This field contains the "sectors per track" geometry value.
BPB_NumHeads	26	2	Number of heads for interrupt 0x13. This field is relevant as discussed earlier for BPB_SecPerTrk. This field contains the one-based "count of heads". For example, on a 1.44 MB 3.5-inch floppy drive this value is 2.
BPB_HiddSec	28	4	Count of hidden sectors preceding the partition that contains this FAT volume. This field is generally only relevant for media visible on interrupt 0x13. This field should always be zero on media that are not partitioned. Exactly what value is appropriate is operating system specific.
BPB_TotSec32	32	4	This field is the new 32-bit total count of sectors on the volume. This count includes the count of all sectors in all four regions of the volume. This field can be 0; if it is 0, then BPB_TotSec16 must be non-zero. For FAT32 volumes, this field must be non-zero. For FAT12/FAT16 volumes, this field contains the sector count if BPB_TotSec16 is 0 (count is greater than or equal to 0x10000).

At this point, the BPB/boot sector for FAT12 and FAT16 differs from the BPB/boot sector for FAT32. The first table shows the structure for FAT12 and FAT16 starting at offset 36 of the boot sector.

Fat12 and Fat16 Structure Starting at Offset 36

Name	Offset (byte)	Size (bytes)	Description
BS_DrvNum	36	1	Int 0x13 drive number (e.g. 0x80). This field supports MS-DOS bootstrap and is set to the INT 0x13 drive number of the media (0x00 for floppy disks, 0x80 for hard disks). NOTE: This field is actually operating system specific.
BS_Reserved1	37	1	Reserved (used by Windows NT). Code that formats FAT volumes should always set this byte to 0.
BS_BootSig	38	1	Extended boot signature (0x29). This is a signature byte that indicates that the following three fields in the boot sector are present.
BS_VolID	39	4	Volume serial number. This field, together with BS_VolLab, supports volume tracking on removable media. These values allow FAT file system drivers to detect that the wrong disk is inserted in a removable drive. This ID is usually generated by simply combining the current date and time into a 32-bit value.
BS_VolLab	43	11	Volume label. This field matches the 11-byte volume label recorded in the root directory. NOTE: FAT file system drivers should make sure that they update this field when the volume label file in the root directory has its name changed or created. The setting for this field when there is no volume label is the string " NO NAME ".
BS_FilSysType	54	8	One of the strings " FAT12 ", " FAT16 ", or " FAT ". NOTE: Many people think that the string in this field has something to do with the determination of what type of FAT—FAT12, FAT16, or FAT32—that the volume has. This is not true. You will note from its name that this field is not actually part of the BPB. This string is informational only and is not used by Microsoft file system drivers to determine FAT type, because it is frequently not set correctly or is not present. See the FAT Type Determination section of this document. This string should be set based on the FAT type though, because some non-Microsoft FAT file system drivers do look at it.

Here is the structure for FAT32 starting at offset 36 of the boot sector.

FAT32 Structure Starting at Offset 36

Name	Offset (byte)	Size (bytes)	Description
BPB_FATSz32	36	4	This field is only defined for FAT32 media and does not exist on FAT12 and FAT16 media. This field is the FAT32 32-bit count of sectors occupied by ONE FAT. BPB_FATSz16 must be 0.
BPB_ExtFlags	40	2	This field is only defined for FAT32 media and does not exist on FAT12 and FAT16 media. Bits 0-3 -- Zero-based number of active FAT. Only valid if mirroring is disabled. Bits 4-6 -- Reserved. Bit 7 -- 0 means the FAT is mirrored at runtime into all FATs. -- 1 means only one FAT is active; it is the one referenced in bits 0-3. Bits 8-15 -- Reserved.
BPB_FSVer	42	2	This field is only defined for FAT32 media and does not exist on FAT12 and FAT16 media. High byte is major revision number. Low byte is minor revision number. This is the version number of the FAT32 volume. This supports the ability to extend the FAT32 media type in the future without worrying about old FAT32 drivers mounting the volume. This document defines the version to 0.0. If this field is non-zero, back-level Windows versions will not mount the volume. NOTE: Disk utilities should respect this field and not operate on volumes with a higher major or minor version number than that for which they were designed. FAT32 file system drivers must check this field and not mount the volume if it does not contain a version number that was defined at the time the driver was written.
BPB_RootClus	44	4	This field is only defined for FAT32 media and does not exist on FAT12 and FAT16 media. This is set to the cluster number of the first cluster of the root directory, usually 2 but not required to be 2. NOTE: Disk utilities that change the location of the root directory should make every effort to place the first cluster of the root directory in the first non-bad cluster on the drive (i.e., in cluster 2, unless it's marked bad). This is specified so that disk repair utilities can easily find the root directory if this field accidentally gets zeroed.
BPB_FSInfo	48	2	This field is only defined for FAT32 media and does not exist on FAT12 and FAT16 media. Sector number of FSINFO structure in the reserved area of the FAT32 volume. Usually 1. NOTE: There will be a copy of the FSINFO structure in BackupBoot, but only the copy pointed to by this field will be kept up to date (i.e., both the primary and backup boot record will point to the same FSINFO sector).
BPB_BkBootSec	50	2	This field is only defined for FAT32 media and does not exist on FAT12 and FAT16 media. If non-zero, indicates the sector number in the reserved area of the volume of a copy of the boot record. Usually 6. No value other than 6 is recommended.
BPB_Reserved	52	12	This field is only defined for FAT32 media and does not exist on FAT12 and FAT16 media. Reserved for future expansion. Code that formats FAT32 volumes should always set all of the bytes of this field to 0.
BS_DrvNum	64	1	This field has the same definition as it does for FAT12 and FAT16 media. The only difference for FAT32 media is that the field is at a different offset in the boot sector.
BS_Reserved1	65	1	This field has the same definition as it does for FAT12 and FAT16 media. The only difference for FAT32 media is that the field is at a different offset in the boot sector.

BS_BootSig	66	1	This field has the same definition as it does for FAT12 and FAT16 media. The only difference for FAT32 media is that the field is at a different offset in the boot sector.
BS_VolID	67	4	This field has the same definition as it does for FAT12 and FAT16 media. The only difference for FAT32 media is that the field is at a different offset in the boot sector.
BS_VolLab	71	11	This field has the same definition as it does for FAT12 and FAT16 media. The only difference for FAT32 media is that the field is at a different offset in the boot sector.
BS_FilSysType	82	8	Always set to the string " FAT32 ". Please see the note for this field in the FAT12/FAT16 section earlier. This field has nothing to do with FAT type determination.

There is one other important note about Sector 0 of a FAT volume. If we consider the contents of the sector as a byte array, it must be true that sector[510] equals 0x55, and sector[511] equals 0xAA.

NOTE: Many FAT documents mistakenly say that this 0xAA55 signature occupies the “last 2 bytes of the boot sector”. This statement is correct if — and only if — BPB_BytsPerSec is 512. If BPB_BytsPerSec is greater than 512, the offsets of these signature bytes do not change (although it is perfectly OK for the last two bytes at the end of the boot sector to also contain this signature).

Check your assumptions about the value in the BPB_TotSec16/32 field. Assume we have a disk or partition of size in sectors DskSz. If the BPB_TotSec field (either BPB_TotSec16 or BPB_TotSec32 — whichever is non-zero) is *less than or equal to* DskSz, there is nothing whatsoever wrong with the FAT volume. In fact, it is not at all unusual to have a BPB_TotSec16/32 value that is slightly smaller than DskSz. It is also perfectly OK for the BPB_TotSec16/32 value to be considerably smaller than DskSz.

All this means is that disk space is being wasted. It does not by itself mean that the FAT volume is damaged in some way. However, if BPB_TotSec16/32 is *larger* than DskSz, the volume is seriously damaged or malformed because it extends past the end of the media or overlaps data that follows it on the disk. Treating a volume for which the BPB_TotSec16/32 value is “too large” for the media or partition as valid can lead to catastrophic data loss.

FAT Data Structure

The next data structure that is important is the FAT itself. What this data structure does is define a singly linked list of the “extents” (clusters) of a file. Note at this point that a FAT directory or file container is nothing but a regular file that has a special attribute indicating it is a directory. The only other special thing about a directory is that the data or contents of the “file” is a series of 32-byte FAT directory entries (see discussion below). In all other respects, a directory is just like a file. The FAT maps the data region of the volume by cluster number. The first data cluster is cluster 2.

The first sector of cluster 2 (the data region of the disk) is computed using the BPB fields for the volume as follows. First, we determine the count of sectors occupied by the root directory:

```
RootDirSectors = ((BPB_RootEntCnt * 32) + (BPB_BytsPerSec - 1)) / BPB_BytsPerSec;
```

Note that on a FAT32 volume the BPB_RootEntCnt value is always 0, so on a FAT32 volume RootDirSectors is always 0. The 32 in the above is the size of one FAT directory entry in bytes. Note also that this computation rounds *up*.

The start of the data region, the first sector of cluster 2, is computed as follows:

```

If(BPB_FATSz16 != 0)
    FATSz = BPB_FATSz16;
Else
    FATSz = BPB_FATSz32;

FirstDataSector = BPB_ReservedSecCnt + (BPB_NumFATs * FATSz) + RootDirSectors;

```

NOTE: This sector number is relative to the first sector of the volume that contains the BPB (the sector that contains the BPB is sector number 0). This does not necessarily map directly onto the drive, because sector 0 of the volume is not necessarily sector 0 of the drive due to partitioning.

Given any valid data cluster number *N*, the sector number of the first sector of that cluster (again relative to sector 0 of the FAT volume) is computed as follows:

```

FirstSectorOfCluster = ((N - 2) * BPB_SecPerClus) + FirstDataSector;

```

NOTE: Because BPB_SecPerClus is restricted to powers of 2 (1,2,4,8,16,32,...), this means that division and multiplication by BPB_SecPerClus can actually be performed via SHIFT operations on 2s complement architectures that are usually faster instructions than MULT and DIV instructions. On current Intel X86 processors, this is largely irrelevant though because the MULT and DIV machine instructions are heavily optimized for multiplication and division by powers of 2.

FAT Type Determination

There is considerable confusion over exactly how this works, which leads to many “off by 1”, “off by 2”, “off by 10”, and “massively off” errors. It is really quite simple how this works. The FAT type—one of FAT12, FAT16, or FAT32—is determined by the count of clusters on the volume and *nothing* else.

Please read everything in this section carefully, all of the words are important. For example, note that the statement was “count of clusters.” This is not the same thing as “maximum valid cluster number,” because the first data cluster is 2 and not 0 or 1.

To begin, let’s discuss exactly how the “count of clusters” value is determined. This is all done using the BPB fields for the volume. First, we determine the count of sectors occupied by the root directory as noted earlier.

```

RootDirSectors = ((BPB_RootEntCnt * 32) + (BPB_BytsPerSec - 1)) / BPB_BytsPerSec;

```

Note that on a FAT32 volume, the BPB_RootEntCnt value is always 0; so on a FAT32 volume, RootDirSectors is always 0.

Next, we determine the count of sectors in the data region of the volume:

```

If(BPB_FATSz16 != 0)
    FATSz = BPB_FATSz16;
Else
    FATSz = BPB_FATSz32;

If(BPB_TotSec16 != 0)
    TotSec = BPB_TotSec16;
Else
    TotSec = BPB_TotSec32;

DataSec = TotSec - (BPB_ReservedSecCnt + (BPB_NumFATs * FATSz) + RootDirSectors);

```

Now we determine the count of clusters:

```
CountofClusters = DataSec / BPB_SecPerClus;
```

Please note that this computation rounds *down*.

Now we can determine the FAT type. *Please note carefully or you will commit an off-by-one error!*

In the following example, when it says $<$, it does not mean \leq . Note also that the numbers are correct. The first number for FAT12 is 4085; the second number for FAT16 is 65525. These numbers and the ' $<$ ' signs are not wrong.

```
if(CountofClusters < 4085) {
    /* Volume is FAT12 */
} else if(CountofClusters < 65525) {
    /* Volume is FAT16 */
} else {
    /* Volume is FAT32 */
}
```

This is the one and only way that FAT type is determined. There is no such thing as a FAT12 volume that has more than 4084 clusters. There is no such thing as a FAT16 volume that has less than 4085 clusters or more than 65,524 clusters. There is no such thing as a FAT32 volume that has less than 65,525 clusters. If you try to make a FAT volume that violates this rule, Microsoft operating systems will not handle them correctly because they will think the volume has a different type of FAT than what you think it does.

NOTE: As is noted numerous times earlier, the world is full of FAT code that is wrong. There is a lot of FAT type code that is off by 1 or 2 or 8 or 10 or 16. For this reason, it is highly recommended that if you are formatting a FAT volume which has maximum compatibility with all existing FAT code, then you should avoid making volumes of any type that have close to 4,085 or 65,525 clusters. Stay at least 16 clusters on each side away from these cut-over cluster counts.

Note also that the CountofClusters value is exactly that—the *count* of data clusters starting at cluster 2. The maximum valid cluster number for the volume is CountofClusters + 1, and the “count of clusters including the two reserved clusters” is CountofClusters + 2.

There is one more important computation related to the FAT. Given any valid cluster number N , where in the FAT(s) is the entry for that cluster number? The only FAT type for which this is complex is FAT12. For FAT16 and FAT32, the computation is simple:

```
if(BPB_FATSz16 != 0)
    FATSz = BPB_FATSz16;
else
    FATSz = BPB_FATSz32;

if(FATType == FAT16)
    FATOffset = N * 2;
else if (FATType == FAT32)
    FATOffset = N * 4;

ThisFATSecNum = BPB_ReservedSecCnt + (FATOffset / BPB_BytsPerSec);
ThisFATEntOffset = REM(FATOffset / BPB_BytsPerSec);
```

REM(...) is the remainder operator. That means the remainder after division of FATOffset by BPB_BytsPerSec. ThisFATSecNum is the sector number of the FAT sector that contains the entry for cluster N in the first FAT. If you want the sector number in the second FAT, you add FATSz to ThisFATSecNum; for the third FAT, you add 2*FATSz, and so on.

You now read sector number `ThisFATSecNum` (remember this is a sector number relative to sector 0 of the FAT volume). Assume this is read into an 8-bit byte array named `SecBuff`. Also assume that the type `WORD` is a 16-bit unsigned and that the type `DWORD` is a 32-bit unsigned.

```

If (FATType == FAT16)
    FAT16ClusterEntryVal = *((WORD *) &SecBuff[ThisFATEntOffset]);
Else
    FAT32ClusterEntryVal = *((DWORD *) &SecBuff[ThisFATEntOffset]) & 0x0FFFFFFF;

```

Fetches the contents of that cluster. To set the contents of this same cluster you do the following:

```

If (FATType == FAT16)
    *((WORD *) &SecBuff[ThisFATEntOffset]) = FAT16ClusterEntryVal;
Else
    {
        FAT32ClusterEntryVal = FAT32ClusterEntryVal & 0x0FFFFFFF;
        *((DWORD *) &SecBuff[ThisFATEntOffset]) =
            (*((DWORD *) &SecBuff[ThisFATEntOffset]) & 0xF0000000) |
            *((DWORD *) &SecBuff[ThisFATEntOffset]);
        *((DWORD *) &SecBuff[ThisFATEntOffset]) =
            (*((DWORD *) &SecBuff[ThisFATEntOffset]) | FAT32ClusterEntryVal);
    }

```

Note how the FAT32 code above works. A FAT32 FAT entry is actually only a 28-bit entry. The high 4 bits of a FAT32 FAT entry are reserved. The only time that the high 4 bits of FAT32 FAT entries should ever be changed is when the volume is formatted, at which time the whole 32-bit FAT entry should be zeroed, including the high 4 bits.

A bit more explanation is in order here, because this point about FAT32 FAT entries seems to cause a great deal of confusion. Basically 32-bit FAT entries are not really 32-bit values; they are only 28-bit values. For example, all of these 32-bit cluster entry values: `0x10000000`, `0xF0000000`, and `0x00000000` all indicate that the cluster is FREE, because you ignore the high 4 bits when you read the cluster entry value. If the 32-bit free cluster value is currently `0x30000000` and you want to mark this cluster as bad by storing the value `0x0FFFFFF7` in it. Then the 32-bit entry will contain the value `0x3FFFFFF7` when you are done, because you must preserve the high 4 bits when you write in the `0x0FFFFFF7` bad cluster mark.

Take note that because the `BPB_BytesPerSec` value is always divisible by 2 and 4, you never have to worry about a FAT16 or FAT32 FAT entry spanning over a sector boundary (this is not true of FAT12).

The code for FAT12 is more complicated because there are 1.5 bytes (12-bits) per FAT entry.

```

If (FATType == FAT12)
    FATOffset = N * (N / 2);
/* Multiply by 1.5 without using floating point, the divide by 2 rounds DOWN */
ThisFATSecNum = BPR_HeadSecCnt + (FATOffset / BPR_BytesPerSec);
ThisFATEntOffset = REM(FATOffset / BPR_BytesPerSec);

```

We now have to check for the sector boundary case:

```

If(ThisFATEntOffset == (BPR_BytesPerSec - 1)) {
    /* This cluster spans a sector boundary in the FAT */
    /* There are a number of strategies to handling this. The */
    /* easiest is to always load FAT sectors into memory */
    /* in pairs if the volume is FAT12 (if you want to load */
    /* FAT sector N, you also load FAT sector N+1 immediately */
    /* following it in memory unless sector N is the last FAT */
    /* sector). It is assumed that this is the strategy used here */
    /* which makes this if test for a sector boundary span */
    /* unnecessary. */
}

```

We now access the FAT entry as a WORD just as we do for FAT16, but if the cluster number is EVEN, we only want the low 12-bits of the 16-bits we fetch; and if the cluster number is ODD, we only want the high 12-bits of the 16-bits we fetch.

```

FAT12ClusterEntryVal = *((WORD *) &SecBuff[ThisFATEntOffset]);
if(N & 0x0001)
    FAT12ClusterEntryVal = FAT12ClusterEntryVal >> 4; /* Cluster number is ODD */
else
    FAT12ClusterEntryVal = FAT12ClusterEntryVal & 0x0FFF; /* Cluster number is EVEN */

```

Fetches the contents of that cluster. To set the contents of this same cluster you do the following:

```

if(N & 0x0001) {
    FAT12ClusterEntryVal = FAT12ClusterEntryVal << 4; /* Cluster number is ODD */
    *((WORD *) &SecBuff[ThisFATEntOffset]) =
        *((WORD *) &SecBuff[ThisFATEntOffset]) & 0x000F;
} else {
    FAT12ClusterEntryVal = FAT12ClusterEntryVal & 0x0FFF; /* Cluster number is EVEN */
    *((WORD *) &SecBuff[ThisFATEntOffset]) =
        *((WORD *) &SecBuff[ThisFATEntOffset]) & 0xF000;
}
*((WORD *) &SecBuff[ThisFATEntOffset]) =
    *((WORD *) &SecBuff[ThisFATEntOffset]) | FAT12ClusterEntryVal;

```

NOTE: It is assumed that the >> operator shifts a bit value of 0 into the high 4 bits and that the << operator shifts a bit value of 0 into the low 4 bits.

The way the data of a file is associated with the file is as follows. In the directory entry, the cluster number of the first cluster of the file is recorded. The first cluster (extent) of the file is the data associated with this first cluster number, and the location of that data on the volume is computed from the cluster number as described earlier (computation of FirstSectorOfCluster).

Note that a zero-length file—a file that has no data allocated to it—has a first cluster number of 0 placed in its directory entry. This cluster location in the FAT (see earlier computation of ThisFATSecNum and ThisFATEntOffset) contains either an EOC mark (End Of Clusterchain) or the cluster number of the next cluster of the file. The EOC value is FAT type dependant (assume FATContent is the contents of the cluster entry in the FAT being checked to see whether it is an EOC mark):

```

IsEOF = FALSE;
if(FATType == FAT12) {
    if(FATContent >= 0x0FFF)
        IsEOF = TRUE;
    } else if(FATType == FAT16) {
    if(FATContent >= 0xFFFF)
        IsEOF = TRUE;
    } else if (FATType == FAT32) {
    if(FATContent >= 0xFFFFFFFF)
        IsEOF = TRUE;
    }
}

```

Note that the cluster number whose cluster entry in the FAT contains the EOC mark is allocated to the file and is also the last cluster allocated to the file. Microsoft operating system FAT drivers use the EOC value 0x0FFF for FAT12, 0xFFFF for FAT16, and 0x0FFFFFFF for FAT32 when they set the contents of a cluster to the EOC mark. There are various disk utilities for Microsoft operating systems that use a different value, however.

There is also a special “BAD CLUSTER” mark. Any cluster that contains the “BAD CLUSTER” value in its FAT entry is a cluster that should not be placed on the free list because it is prone to disk errors. The “BAD CLUSTER” value is 0x0FF7 for FAT12, 0xFFFF for FAT16, and 0x0FFFFFFF for FAT32. The other relevant note here is that these bad clusters are also lost clusters—clusters that appear to be allocated because they contain a non-zero value but which are not part of any files allocation chain. Disk repair utilities must recognize lost clusters that contain this special value as bad clusters and not change the content of the cluster entry.

NOTE: It is not possible for the bad cluster mark to be an allocatable cluster number on FAT12 and FAT16 volumes, but it is feasible for 0x0FFFFFF7 to be an allocatable cluster number on FAT32 volumes. To avoid possible confusion by disk utilities, no FAT32 volume should ever be configured such that 0x0FFFFFF7 is an allocatable cluster number.

The list of free clusters in the FAT is nothing more than the list of all clusters that contain the value 0 in their FAT cluster entry. Note that this value must be fetched as described earlier as for any other FAT entry that is not free. This list of free clusters is not stored anywhere on the volume; it must be computed when the volume is mounted by scanning the FAT for entries that contain the value 0. On FAT32 volumes, the BPB_FSInfo sector *may* contain a valid count of free clusters on the volume. See the documentation of the FAT32 FSInfo sector.

What are the two reserved clusters at the start of the FAT for? The first reserved cluster, FAT[0], contains the BPB_Media byte value in its low 8 bits, and all other bits are set to 1. For example, if the BPB_Media value is 0xF8, for FAT12 FAT[0] = 0x0FF8, for FAT16 FAT[0] = 0xFFFF, and for FAT32 FAT[0] = 0x0FFFFFF8. The second reserved cluster, FAT[1], is set by FORMAT to the EOC mark. On FAT12 volumes, it is not used and is simply always contains an EOC mark. For FAT16 and FAT32, the file system driver may use the high two bits of the FAT[1] entry for dirty volume flags (all other bits, are always left set to 1). Note that the bit location is different for FAT16 and FAT32, because they are the high 2 bits of the entry.

For FAT16:

```
ClnShutBitMask    = 0x8000;
HrdErrBitMask     = 0x4000;
```

For FAT32:

```
ClnShutBitMask    = 0x08000000;
HrdErrBitMask     = 0x04000000;
```

- Bit ClnShutBitMask – If bit is 1, volume is “clean”.
If bit is 0, volume is “dirty”. This indicates that the file system driver did not Dismount the volume properly the last time it had the volume mounted. It would be a good idea to run a Chkdsk/Scandisk disk repair utility on it, because it may be damaged.
- Bit HrdErrBitMask – If this bit is 1, no disk read/write errors were encountered.
If this bit is 0, the file system driver encountered a disk I/O error on the Volume the last time it was mounted, which is an indicator that some sectors may have gone bad on the volume. It would be a good idea to run a Chkdsk/Scandisk disk repair utility that does surface analysis on it to look for new bad sectors.

Here are two more important notes about the FAT region of a FAT volume:

1. The last sector of the FAT is not necessarily all part of the FAT. The FAT stops at the cluster number in the last FAT sector that corresponds to the entry for cluster number `CountOfClusters + 1` (see the `CountOfClusters` computation earlier), and this entry is not necessarily at the end of the last FAT sector. FAT code should not make any assumptions about what the contents of the last FAT sector are after the `CountOfClusters + 1` entry. FAT format code should zero the bytes after this entry though.
2. The `BPB_FATSz16` (`BPB_FATSz32` for FAT32 volumes) value *may* be bigger than it needs to be. In other words, there may be totally unused FAT sectors at the end of each FAT in the FAT region of the volume. For this reason, the last sector of the FAT is always computed using the `CountOfClusters + 1` value, never from the `BPB_FATSz16/32` value. FAT code should not make any assumptions about what the contents of these “extra” FAT sectors are. FAT format code should zero the contents of these extra FAT sectors though.

FAT Volume Initialization

At this point, the careful reader should have one very interesting question. Given that the FAT type (FAT12, FAT16, or FAT32) is dependant on the number of clusters—and that the sectors available in the data area of a FAT volume is dependant on the size of the FAT—when handed an unformatted volume that does not yet have a BPB, how do you determine all this and compute the proper values to put in `BPB_SecPerClus` and either `BPB_FATSz16` or `BPB_FATSz32`? The way Microsoft operating systems do this is with a fixed value, several tables, and a clever piece of arithmetic.

Microsoft operating systems only do FAT12 on floppy disks. Because there is a limited number of floppy formats that all have a fixed size, this is done with a simple table:

“If it is a floppy of this type, then the BPB looks like this.”

There is no dynamic computation for FAT12. For the FAT12 formats, all the computation for `BPB_SecPerClus` and `BPB_FATSz16` was worked out by hand on a piece of paper and recorded in the table (being careful of course that the resultant cluster count was always less than 4085). If your media is larger than 4 MB, do not bother with FAT12. Use smaller `BPB_SecPerClus` values so that the volume will be FAT16.

The rest of this section is totally specific to drives that have 512 bytes per sector. You cannot use these tables, or the clever arithmetic, with drives that have a different sector size. The “fixed value” is simply a volume size that is the “FAT16 to FAT32 cutover value”. Any volume size smaller than this is FAT16 and any volume of this size or larger is FAT32. For Windows, this value is 512 MB. Any FAT volume smaller than 512 MB is FAT16, and any FAT volume of 512 MB or larger is FAT32.

Please don’t draw an incorrect conclusion here.

There are many FAT16 volumes out there that are larger than 512 MB. There are various ways to force the format to be FAT16 rather than the default of FAT32, and there is a great deal of code that implements different limits. All we are talking about here is the *default* cutover value for MS-DOS and Windows on volumes that have not yet been formatted. There are two tables—one is for FAT16 and the other is for FAT32. An entry in these tables is selected based on the size of the volume in 512 byte sectors (the value that will go in `BPB_TotSec16` or `BPB_TotSec32`), and the value that this table sets is the `BPB_SecPerClus` value.

```

struct DSKSTOSECPCERCLUS {
    DWORD   DiskSize;
    BYTE    SecPerClusVal;
};

/*
* This is the table for FAT16 drives. NOTE that this table includes
* * entries for disk sizes larger than 512 MB even though typically
* * only the entries for disks >= 512 MB in size are used.
* * The way this table is accessed is to look for the first entry
* * in the table for which the disk size is less than or equal
* * to the DiskSize field in that table entry. For this table to
* * work properly BPB_RsvdSecCnt must be 1, BPB_RunFATs
* * must be 2, and BPB_RootEntCnt must be 512. Any of these values
* * being different may require the first table entries DiskSize value
* * to be changed otherwise the cluster count may be too low for FAT16.
*/
DSKSTOSECPCERCLUS DskTableFAT16 [] = {
    { 8400, 0}, /* disks up to 4.1 MB, the 0 value for SecPerClusVal trips an error */
    { 32680, 2}, /* disks up to 16 MB, 1k cluster */
    { 262144, 4}, /* disks up to 128 MB, 2k cluster */
    { 524288, 8}, /* disks up to 256 MB, 4k cluster */
    { 1048576, 16}, /* disks up to 512 MB, 8k cluster */
    /* The entries after this point are not used unless FAT16 is forced */
    { 2097152, 32}, /* disks up to 1 GB, 16k cluster */
    { 4194304, 64}, /* disks up to 2 GB, 32k cluster */
    { 0xFFFFFFFF, 0} /* any disk greater than 2GB, 0 value for SecPerClusVal trips an error */
};

/*
* This is the table for FAT32 drives. NOTE that this table includes
* * entries for disk sizes smaller than 512 MB even though typically
* * only the entries for disks >= 512 MB in size are used.
* * The way this table is accessed is to look for the first entry
* * in the table for which the disk size is less than or equal
* * to the DiskSize field in that table entry. For this table to
* * work properly BPB_RsvdSecCnt must be 32, and BPB_RunFATs
* * must be 2. Any of these values being different may require the first
* * table entries DiskSize value to be changed otherwise the cluster count
* * may be too low for FAT32.
*/
DSKSTOSECPCERCLUS DskTableFAT32 [] = {
    { 65536, 0}, /* disks up to 32.5 MB, the 0 value for SecPerClusVal trips an error */
    { 532480, 1}, /* disks up to 260 MB, .5k cluster */
    { 16777216, 8}, /* disks up to 8 GB, 4k cluster */
    { 33554432, 16}, /* disks up to 16 GB, 8k cluster */
    { 67108864, 32}, /* disks up to 32 GB, 16k cluster */
    { 0xFFFFFFFF, 64} /* disks greater than 32GB, 32k cluster */
};

```

So given a disk size and a FAT type of FAT16 or FAT32, we now have a BPB_SecPerClus value. The only thing we have left is to compute how many sectors the FAT takes up so that we can set BPB_FATSz16 or BPB_FATSz32. Note that at this point we assume that BPB_RootEntCnt, BPB_RsvdSecCnt, and BPB_NumFATs are appropriately set. We also assume that DskSize is the size of the volume that we are either going to put in BPB_TotSec32 or BPB_TotSec16.


```

RootDirSectors = ((BPB_RootEntCnt * 32) + (BPB_BytsPerSec - 1)) / BPB_BytsPerSec;
TmpVal1 = DiskSize - (BPB_ReservedSecCnt + RootDirSectors);
TmpVal2 = (256 * BPB_SecPerClus) + BPB_NumFAts;
if (FATType == FAT32)
    TmpVal2 = TmpVal2 / 2;
FATSz = (TmpVal1 + (TmpVal2 - 1)) / TmpVal2;
if (FATType == FAT32) {
    BPB_FATSz16 = 0;
    BPB_FATSz32 = FATSz;
} else {
    BPB_FATSz16 = LOWORD(FATSz);
    /* there is no BPB_FATSz32 in a FAT16 BPB */
}

```

Do not spend too much time trying to figure out why this math works. The basis for the computation is complicated; the important point is that this is how Microsoft operating systems do it, and it works. Note, however, that this math does not work perfectly. It will occasionally set a FATSz that is up to 2 sectors too large for FAT16, and occasionally up to 8 sectors too large for FAT32. It will never compute a FATSz value that is too small, however. Because it is OK to have a FATSz that is too large, at the expense of wasting a few sectors, the fact that this computation is surprisingly simple more than makes up for it being off in a safe way in some cases.

FAT32 FSInfo Sector Structure and Backup Boot Sector

On a FAT32 volume, the FAT can be a large data structure, unlike on FAT16 where it is limited to a maximum of 128K worth of sectors and FAT12 where it is limited to a maximum of 6K worth of sectors. For this reason, a provision is made to store the “last known” free cluster count on the FAT32 volume so that it does not have to be computed as soon as an API call is made to ask how much free space there is on the volume (like at the end of a directory listing). The FSInfo sector number is the value in the BPB_FSInfo field; for Microsoft operating systems it is always set to 1. Here is the structure of the FSInfo sector:

FAT32 FSInfo Sector Structure and Backup Boot Sector

Name	Offset (byte)	Size (bytes)	Description
FSI_LeadSig	0	4	Value 0x41615252. This lead signature is used to validate that this is in fact an FSInfo sector.
FSI_Reserved1	4	480	This field is currently reserved for future expansion. FAT32 format code should always initialize all bytes of this field to 0. Bytes in this field must currently never be used.
FSI_StrucSig	484	4	Value 0x61417272. Another signature that is more localized in the sector to the location of the fields that are used.
FSI_Free_Count	488	4	Contains the last known free cluster count on the volume. If the value is 0xFFFFFFFF, then the free count is unknown and must be computed. Any other value can be used, but is not necessarily correct. It should be range checked at least to make sure it is <= volume cluster count.
FSI_Nxt_Free	492	4	This is a hint for the FAT driver. It indicates the cluster number at which the driver should start looking for free clusters. Because a FAT32 FAT is large, it can be rather time consuming if there are a lot of allocated clusters at the start of the FAT and the driver starts looking for a free cluster starting at cluster 2. Typically this value is set to the last cluster number that the driver allocated. If the value is 0xFFFFFFFF, then there is no hint and the driver should start looking at cluster 2. Any other value can be used, but should be checked first to make sure it is a valid cluster number for the volume.
FSI_Reserved2	496	12	This field is currently reserved for future expansion. FAT32 format code should always initialize all bytes of this field to 0. Bytes in this field must currently never be used.

FSI_TrailSig	508	4	Value 0xAA550000. This trail signature is used to validate that this is in fact an FSInfo sector. Note that the high 2 bytes of this value—which go into the bytes at offsets 510 and 511—match the signature bytes used at the same offsets in sector 0.
--------------	-----	---	---

Another feature on FAT32 volumes that is not present on FAT16/FAT12 is the BPB_BkBootSec field. FAT16/FAT12 volumes can be totally lost if the contents of sector 0 of the volume are overwritten or sector 0 goes bad and cannot be read. This is a “single point of failure” for FAT16 and FAT12 volumes. The BPB_BkBootSec field reduces the severity of this problem for FAT32 volumes, because starting at that sector number on the volume—6—there is a backup copy of the boot sector information including the volume’s BPB.

In the case where the sector 0 information has been accidentally overwritten, all a disk repair utility has to do is restore the boot sector(s) from the backup copy. In the case where sector 0 goes bad, this allows the volume to be mounted so that the user can access data before replacing the disk.

This second case—sector 0 goes bad—is the reason why no value other than 6 should ever be placed in the BPB_BkBootSec field. If sector 0 is unreadable, various operating systems are “hard wired” to check for backup boot sector(s) starting at sector 6 of the FAT32 volume. Note that starting at the BPB_BkBootSec sector is a *complete* boot record. The Microsoft FAT32 “boot sector” is actually three 512-byte sectors long. There is a copy of all three of these sectors starting at the BPB_BkBootSec sector. A copy of the FSInfo sector is also there, even though the BPB_FSInfo field in this backup boot sector is set to the same value as is stored in the sector 0 BPB.

NOTE: All 3 of these sectors have the 0xAA55 signature in sector offsets 510 and 511, just like the first boot sector does (see the earlier discussion at the end of the BPB structure description).

FAT Directory Structure

We will first talk about short directory entries and ignore long directory entries for the moment.

A FAT directory is nothing but a “file” composed of a linear list of 32-byte structures. The only special directory, which must always be present, is the root directory. For FAT12 and FAT16 media, the root directory is located in a fixed location on the disk immediately following the last FAT and is of a fixed size in sectors computed from the BPB_RootEntCnt value (see computations for RootDirSectors earlier in this document). For FAT12 and FAT16 media, the first sector of the root directory is sector number relative to the first sector of the FAT volume:

```
FirstRootDirSecNum = BPB_RsvdSecCnt + (BPB_NumFats * BPB_FATSz16);
```

For FAT32, the root directory can be of variable size and is a cluster chain, just like any other directory is. The first cluster of the root directory on a FAT32 volume is stored in BPB_RootClus. Unlike other directories, the root directory itself on any FAT type does not have any date or time stamps, does not have a file name (other than the implied file name “\”), and does not contain “.” and “..” files as the first two directory entries in the directory. The only other special aspect of the root directory is that it is the only directory on the FAT volume for which it is valid to have a file that has only the ATTR_VOLUME_ID attribute bit set (see below).

FAT 32 Byte Directory Entry Structure

Name	Offset (byte)	Size (bytes)	Description
DIR_Name	0	11	Short name.
DIR_Attr	11	1	File attributes: ATTR_READ_ONLY 0x01 ATTR_HIDDEN 0x02 ATTR_SYSTEM 0x04 ATTR_VOLUME_ID 0x08 ATTR_DIRECTORY 0x10 ATTR_ARCHIVE 0x20 ATTR_LONG_NAME ATTR_READ_ONLY ATTR_HIDDEN ATTR_SYSTEM ATTR_VOLUME_ID The upper two bits of the attribute byte are reserved and should always be set to 0 when a file is created and never modified or looked at after that.
DIR_NTRes	12	1	Reserved for use by Windows NT. Set value to 0 when a file is created and never modify or look at it after that.
DIR_CrtTimeTenth	13	1	Millisecond stamp at file creation time. This field actually contains a count of tenths of a second. The granularity of the seconds part of DIR_CrtTime is 2 seconds so this field is a count of tenths of a second and its valid value range is 0-199 inclusive.
DIR_CrtTime	14	2	Time file was created.
DIR_CriDate	16	2	Date file was created.
DIR_LstAccDate	18	2	Last access date. Note that there is no last access time, only a date. This is the date of last read or write. In the case of a write, this should be set to the same date as DIR_WrtDate.
DIR_FstClusHI	20	2	High word of this entry's first cluster number (always 0 for a FAT12 or FAT16 volume).
DIR_WrtTime	22	2	Time of last write. Note that file creation is considered a write.
DIR_WrtDate	24	2	Date of last write. Note that file creation is considered a write.
DIR_FstClusLO	26	2	Low word of this entry's first cluster number.
DIR_FileSize	28	4	32-bit DWORD holding this file's size in bytes.

DIR_Name[0]

Special notes about the first byte (DIR_Name[0]) of a FAT directory entry:

- If DIR_Name[0] == 0xE5, then the directory entry is free (there is no file or directory name in this entry).
- If DIR_Name[0] == 0x00, then the directory entry is free (same as for 0xE5), and there are no allocated directory entries after this one (all of the DIR_Name[0] bytes in all of the entries after this one are also set to 0).

The special 0 value, rather than the 0xE5 value, indicates to FAT file system driver code that the rest of the entries in this directory do not need to be examined because they are all free.

- If DIR_Name[0] == 0x05, then the actual file name character for this byte is 0xE5. 0xE5 is actually a valid KANJI lead byte value for the character set used in Japan. The special 0x05 value is used so that this special file name case for Japan can be handled properly and not cause FAT file system code to think that the entry is free.

The `DIR_Name` field is actually broken into two parts: the 8-character main part of the name, and the 3-character extension. These two parts are “trailing space padded” with bytes of 0x20.

`DIR_Name[0]` may not equal 0x20. There is an implied ‘.’ character between the main part of the name and the extension part of the name that is not present in `DIR_Name`. Lower case characters are not allowed in `DIR_Name` (what these characters are is country specific).

The following characters are not legal in any bytes of `DIR_Name`:

- Values less than 0x20 except for the special case of 0x05 in `DIR_Name[0]` described above.
- 0x22, 0x2A, 0x2B, 0x2C, 0x2E, 0x2F, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F, 0x5B, 0x5C, 0x5D, and 0x7C.

Here are some examples of how a user-entered name maps into `DIR_Name`:

```
*foo.bar"      -> "FOO  BAR"
*FOO.BAR"      -> "FOO  BAR"
*Foo.Bar"      -> "FOO  BAR"
*foo"          -> "FOO  "
*foo."         -> "FOO  "
*PICKLE.A"     -> "PICKLE A "
*prettybig.big" -> "PRETTYBIG"
*.big"         -> illegal, DIR_Name[0] cannot be 0x20
```

In FAT directories all names are unique. Look at the first three examples earlier. Those different names all refer to the same file, and there can only be one file with `DIR_Name` set to “FOO BAR” in any directory.

`DIR_Attr` specifies attributes of the file:

<code>ATTR_READ_ONLY</code>	Indicates that writes to the file should fail.
<code>ATTR_HIDDEN</code>	Indicates that normal directory listings should not show this file.
<code>ATTR_SYSTEM</code>	Indicates that this is an operating system file.
<code>ATTR_VOLUME_ID</code>	There should only be one “file” on the volume that has this attribute set, and that file must be in the root directory. This name of this file is actually the label for the volume. <code>DIR_FstClusHI</code> and <code>DIR_FstClusLO</code> must always be 0 for the volume label (no data clusters are allocated to the volume label file).
<code>ATTR_DIRECTORY</code>	Indicates that this file is actually a container for other files.
<code>ATTR_ARCHIVE</code>	This attribute supports backup utilities. This bit is set by the FAT file system driver when a file is created, renamed, or written to. Backup utilities may use this attribute to indicate which files on the volume have been modified since the last time that a backup was performed.

Note that the `ATTR_LONG_NAME` attribute bit combination indicates that the “file” is actually part of the long name entry for some other file. See the next section for more information on this attribute combination.

When a directory is created, a file with the `ATTR_DIRECTORY` bit set in its `DIR_Attr` field, you set its `DIR_FileSize` to 0. `DIR_FileSize` is not used and is always 0 on a file with the `ATTR_DIRECTORY` attribute (directories are sized by simply following their cluster chains to the EOC mark). One cluster is allocated to the directory (unless it is the root directory on a FAT16/FAT12 volume), and you set `DIR_FstClusLO` and `DIR_FstClusHI` to that cluster number and place an EOC mark in that clusters entry in the FAT. Next, you initialize all bytes of that cluster to 0. If the directory is the root directory, you are done (there are no dot or *dotdot* entries in the root directory). If the directory is not the root directory, you need to create two special entries in the first two 32-byte

directory entries of the directory (the first two 32 byte entries in the data region of the cluster you just allocated).

The first directory entry has DIR_Name set to:

```

"

```

The second has DIR_Name set to:

```

"..

```

These are called the *dot* and *dotdot* entries. The DIR_FileSize field on both entries is set to 0, and all of the date and time fields in both of these entries are set to the same values as they were in the directory entry for the directory that you just created. You now set DIR_FstClusLO and DIR_FstClusHI for the *dot* entry (the first entry) to the same values you put in those fields for the directories directory entry (the cluster number of the cluster that contains the *dot* and *dotdot* entries).

Finally, you set DIR_FstClusLO and DIR_FstClusHI for the *dotdot* entry (the second entry) to the first cluster number of the directory in which you just created the directory (value is 0 if this directory is the root directory even for FAT32 volumes).

Here is the summary for the *dot* and *dotdot* entries:

- The *dot* entry is a directory that points to itself.
- The *dotdot* entry points to the starting cluster of the parent of this directory (which is 0 if this directory's parent is the root directory).

Date and Time Formats

Many FAT file systems do not support Date/Time other than DIR_WrtTime and DIR_WrtDate. For this reason, DIR_CrtTimeMil, DIR_CrtTime, DIR_CrtDate, and DIR_LstAccDate are actually optional fields. DIR_WrtTime and DIR_WrtDate *must* be supported, however. If the other date and time fields are not supported, they should be set to 0 on file create and ignored on other file operations.

Date Format. A FAT directory entry date stamp is a 16-bit field that is basically a date relative to the MS-DOS epoch of 01/01/1980. Here is the format (bit 0 is the LSB of the 16-bit word, bit 15 is the MSB of the 16-bit word):

Bits 0–4: Day of month, valid value range 1–31 inclusive.

Bits 5–8: Month of year, 1 = January, valid value range 1–12 inclusive.

Bits 9–15: Count of years from 1980, valid value range 0–127 inclusive (1980–2107).

Time Format. A FAT directory entry time stamp is a 16-bit field that has a granularity of 2 seconds. Here is the format (bit 0 is the LSB of the 16-bit word, bit 15 is the MSB of the 16-bit word).

Bits 0–4: 2-second count, valid value range 0–29 inclusive (0 – 58 seconds).

Bits 5–10: Minutes, valid value range 0–59 inclusive.

Bits 11–15: Hours, valid value range 0–23 inclusive.

The valid time range is from Midnight 00:00:00 to 23:59:58.

FAT Long Directory Entries

In adding long directory entries to the FAT file system it was crucial that their addition to the FAT file system's existing design:

- Be essentially transparent on earlier versions of MS-DOS. The primary goal being that existing MS-DOS APIs on previous versions of MS-DOS/Windows do not easily “find” long directory entries. The only MS-DOS APIs that can “find” long directory entries are the FCB-based-find APIs when used with a full meta-character matching pattern (i.e. *, ?) and full attribute matching bits (i.e. matching attributes are FFh). On post-Windows 95 versions of MS-DOS/Windows, no MS-DOS API can accidentally “find” a single long directory entry.
- Be located in close physical proximity, on the media, to the short directory entries they are associated with. As will be evident, long directory entries are immediately contiguous to the short directory entry they are associated with and their existence imposes an unnoticeable performance impact on the file system.
- If detected by disk maintenance utilities, they do not jeopardize the integrity of existing file data. Disk maintenance utilities typically do not use MS-DOS APIs to access on-media file-system-specific data structures. Rather they read physical or logical sector information from the disk and judge for themselves what the directory entries contain. Based on the heuristics employed in the utilities, the utility may take various steps to “repair” what it perceives to be “damaged” file-system-specific data structures. Long directory entries were added to the FAT file system in such a way as to not cause the loss of file data if a disk containing long directory entries was “repaired” by a pre-Windows 95-compatible disk utility on a previous version of MS-DOS/Windows.

In order to meet the goals of locality-of-access and transparency, the long directory entry is defined as a short directory entry with a special attribute. As described previously, a long directory entry is just a regular directory entry in which the attribute field has a value of:

```
ATTR_LONG_NAME      ATTR_READ_ONLY |
                   ATTR_HIDDEN |
                   ATTR_SYSTEM |
                   ATTR_VOLUME_ID
```

A mask for determining whether an entry is a long-name sub-component should also be defined:

```
ATTR_LONG_NAME_MASK ATTR_READ_ONLY |
                   ATTR_HIDDEN |
                   ATTR_SYSTEM |
                   ATTR_VOLUME_ID |
                   ATTR_DIRECTORY |
                   ATTR_ARCHIVE
```

When such a directory entry is encountered it is given special treatment by the file system. It is treated as part of a set of directory entries that are associated with a single short directory entry. Each long directory entry has the following structure:

FAT Long Directory Entry Structure

Name	Offset (byte)	Size (bytes)	Description
LDIR_Old	0	1	The order of this entry in the sequence of long dir entries associated with the short dir entry at the end of the long dir set. If masked with 0x40 (LAST_LONG_ENTRY), this indicates the entry is the last long dir entry in a set of long dir entries. All valid sets of long dir entries must begin with an entry having this mask.
LDIR_Name1	1	10	Characters 1-5 of the long-name sub-component in this dir entry.
LDIR_Attr	11	1	Attributes - must be ATTR_LONG_NAME

LDIR_Type	12	1	If zero, indicates a directory entry that is a sub-component of a long name. NOTE: Other values reserved for future extensions. Non-zero implies other dirent types.
LDIR_Chksum	13	1	Checksum of name in the short dir entry at the end of the long dir set.
LDIR_Name2	14	12	Characters 6-11 of the long-name sub-component in this dir entry.
LDIR_FuClusLO	26	2	Must be ZERO. This is an artifact of the FAT "first cluster" and must be zero for compatibility with existing disk utilities. It's meaningless in the context of a long dir entry.
LDIR_Name3	28	4	Characters 12-13 of the long-name sub-component in this dir entry.

Organization and Association of Short & Long Directory Entries

A set of long entries is always associated with a short entry that they always immediately precede. Long entries are paired with short entries for one reason: only short directory entries are visible to previous versions of MS-DOS/Windows. Without a short entry to accompany it, a long directory entry would be completely invisible on previous versions of MS-DOS/Windows. A long entry never legally exists all by itself. If long entries are found without being paired with a valid short entry, they are termed *orphans*. The following figure depicts a set of *n* long directory entries associated with its single short entry.

Long entries always immediately precede and are physically contiguous with, the short entry they are associated with. The file system makes a few other checks to ensure that a set of long entries is actually associated with a short entry.

Sequence Of Long Directory Entries

Entry	Ordinal
Nth Long entry	LAST_LONG_ENTRY (0x40) N
... Additional Long Entries	...
1 st Long entry	1
Short Entry Associated With Preceding Long Entries	(not applicable)

First, every member of a set of long entries is uniquely numbered and the last member of the set is *ord* with a flag indicating that it is, in fact, the last member of the set. The LDIR_Ord field is used to make this determination. The first member of a set has an LDIR_Ord value of one. The *n*th long member of the set has a value of (*n* OR LAST_LONG_ENTRY). Note that the LDIR_Ord field cannot have values of 0xE5 or 0x00. These values have always been used by the file system to indicate a "free" directory entry, or the "last" directory entry in a cluster. Values for LDIR_Ord do not take on these two values over their range. Values for LDIR_Ord must run from 1 to (*n* OR LAST_LONG_ENTRY). If they do not, the long entries are "damaged" and are treated as orphans by the file system.

Second, an 8-bit checksum is computed on the name contained in the short directory entry at the time the short and long directory entries are created. All 11 characters of the name in the short entry are used in the checksum calculation. The check sum is placed in every long entry. If any of the check sums in the set of long entries do not agree with the computed checksum of the name contained in the short entry, then the long entries are treated as orphans. This can occur if a disk containing long and short entries is taken to a previous version of MS-DOS/Windows and only the short name of a file or directory with a long entries is renamed.

The algorithm, implemented in C, for computing the checksum is:

```

//-----
// ChkSum()
// Returns an unsigned byte checksum computed on an unsigned byte
// array. The array must be 11 bytes long and is assumed to contain
// a name stored in the format of a MS-DOS directory entry.
// Passed: pFcbName Pointer to an unsigned byte array assumed to be
// 11 bytes long.
// Returns: Sum An 8-bit unsigned checksum of the array pointed
// to by pFcbName.
//-----
unsigned char ChkSum(unsigned char *pFcbName)
{
    short FcbNameLen;
    unsigned char Sum;

    Sum = 0;
    for (FcbNameLen=11; FcbNameLen!=0; FcbNameLen--) {
        // NOTE: The operation is an unsigned char rotate right.
        Sum = ((Sum & 1) ? 0x80 : 0) + (Sum >> 1) + *pFcbName++;
    }
    return (Sum);
}

```

As a consequence of this pairing, the short directory entry serves as the structure that contains fields like: last access date, creation time, creation date, first cluster, and size. It also holds a name that is visible on previous versions of MS-DOS/Windows. The long directory entries are free to contain new information and need not replicate information already available in the short entry. Principally, the long entries contain the long name of a file. The name contained in a short entry which is associated with a set of long entries is termed the *alias name*, or simply *alias*, of the file.

Storage of a Long-Name Within Long Directory Entries

A long name can consist of more characters than can fit in a single long directory entry. When this occurs the name is stored in more than one long entry. In any event, the name fields themselves within the long entries are disjoint. The following example is provided to illustrate how a long name is stored across several long directory entries. Names are also NUL terminated and padded with 0xFFFF characters in order to detect corruption of long name fields by errant disk utilities. A name that fits exactly in a n long directory entries (i.e. is an integer multiple of 13) is not NUL terminated and not padded with 0xFFFFs.

Suppose a file is created with the name: "The quick brown fox". The following example illustrates how the name is packed into long and short directory entries. Most fields in the directory entries are also filled in as well.

2nd long entry (and last)	→	42h	w	n	.	f	o	0Fh	00h	chk- sum	x
		0000h	FFFFh	FFFFh	FFFFh	FFFFh	0000h	FFFFh	FFFFh		
1st long entry	→	01h	T		h	e		q	0Fh	00h	chk- sum
			i	c		k		b	0000h	r	o
Short entry	→	T	H	E	Q	U	I	~	I	F	O
		Created Date	Last Access Date	0000h	Last Modified Time	Last Modified Date	First Cluster	File Size			

The heuristics used to "auto-generate" a short name from a long name are explained in a later section.

Name Limits and Character Sets

Short Directory Entries

Short names are limited to 8 characters followed by an optional period (.) and extension of up to 3 characters. The total path length of a short name cannot exceed 80 characters (64 char path + 3 drive letter + 12 for 8.3 name + NUL) including the trailing NUL. The characters may be any combination of letters, digits, or characters with code point values greater than 127. The following special characters are also allowed:

\$ % ' - _ @ ~ ! () { } ^ # &

Names are stored in a short directory entry in the OEM code page that the system is configured for at the time the directory entry is created. Short directory entries remain in OEM for compatibility with previous versions of MS-DOS/Windows. OEM characters are single 8-bit characters or can be DBCS character pairs for certain code pages.

Short names passed to the file system are always converted to upper case and their original case value is lost. One problem that is generally true of most OEM code pages is that they map lower to upper case extended characters in a non-unique fashion. That is, they map multiple extended characters to a single upper case character. This creates problems because it does not preserve the information that the extended character provides. This mapping also prevents the creation of some file names that would normally differ, but because of the mapping to upper case they become the same file name.

Long Directory Entries

Long names are limited to 255 characters, not including the trailing NUL. The total path length of a long name cannot exceed 260 characters, including the trailing NUL. The characters may be any combination of those defined for short names with the addition of the period (.) character used multiple times within the long name. A space is also a valid character in a long name as it always has been for a short name. However, in short names it typically is not used. The following six special characters are now allowed in a long name. They are not legal in a short name.

+ , ; = []

Embedded spaces within a long name are allowed. Leading and trailing spaces in a long name are ignored.

Leading and embedded periods are allowed in a name and are stored in the long name. Trailing periods are ignored.

Long names are stored in long directory entries in UNICODE. UNICODE characters are 16-bit characters. It is not possible to store UNICODE in short directory entries since the names stored there are 8-bit characters or DBCS characters.

Long names passed to the file system are not converted to upper case and their original case value is preserved. UNICODE solves the case mapping problem prevalent in some OEM code pages by always providing a translation for lower case characters to a single, unique upper case character.

Name Matching In Short & Long Names

The names contained in the set of all short directory entries are termed the "short name space". The names contained in the set of all long directory entries are termed the "long name space". Together, they form a single unified name space in which no duplicate names can exist. That is: any name within a specific directory, whether it is a short name or a long name, can occur only once in the name space. Furthermore, although the case of a name is preserved in a long name, no two names can have the same name although the names on the media actually differ by case. That is names like "foobar" cannot be created if there is already a short entry with a name of "FOOBAR" or a long name with a name of "FooBar".

All types of search operations within the file system (i.e. find, open, create, delete, rename) are case-insensitive. An open of "FOOBAR" will open either "FooBar" or "foobar" if one or the other exists. A find using "FOOBAR" as a pattern will find the same files mentioned. The same rules are also true for extended characters that are accented.

A short name search operation checks only the names of the short directory entries for a match. A long name search operation checks both the long and short directory entries. As the file system traverses a directory, it caches the long-name sub-components contained in long directory entries. As soon as a short directory entry is encountered that is associated with the cached long name, the long name search operation will check the cached long name first and then the short name for a match.

When a character on the media, whether it is stored in the OEM character set or in UNICODE, cannot be translated into the appropriate character in the OEM or ANSI code page, it is always "translated" to the "_" (underscore) character as it is returned to the user – it is NOT modified on the disk. This character is the same in all OEM code pages and ANSI.

Naming Conventions and Long Names

An API allows the caller to specify the long name to be assigned to a file or directory. They do not allow the caller to independently specify the short name. The reason for this prohibition is that the short and long names are considered to be a single unified name space. As should be obvious the file system's name space does not support duplicate names. In other words, a long name for a file may not contain the same name, ignoring case, as the short name in a different file. This restriction is intended to prevent confusion among users, and applications, regarding the proper name of a file or directory. To make this restriction transparent, whenever a long name is created and the no matching long name exists, the short name is automatically generated from the long name in such a way that it does not collide with an existing short name.

The technique chosen to auto-generate short names from long names is modeled after Windows NT. Auto-generated short names are composed of the *basis-name* and an optional *numeric-tail*.

The Basis-Name Generation Algorithm

The *basis-name* generation algorithm is outlined below. This is a *sample* algorithm and serves to illustrate how short names can be auto-generated from long names. An implementation *should* follow this basic sequence of steps.

1. The UNICODE name passed to the file system is converted to upper case.
2. The upper cased UNICODE name is converted to OEM.
 if (the uppercased UNICODE glyph does not exist as an OEM glyph in the OEM code page)
 or (the OEM glyph is invalid in an 8.3 name)
 {

- ```

 Replace the glyph to an OEM '_' (underscore) character.
 Set a "lossy conversion" flag.
 }

3. Strip all leading and embedded spaces from the long name.

4. Strip all leading periods from the long name.

5. While (not at end of the long name)
 and (char is not a period)
 and (total chars copied < 8)
 {
 Copy characters into primary portion of the basis name
 }

6. Insert a dot at the end of the primary components of the basis-name iff the basis name has an
 extension after the last period in the name.

7. Scan for the last embedded period in the long name.
 If (the last embedded period was found)
 {
 While (not at end of the long name)
 and (total chars copied < 3)
 {
 Copy characters into extension portion of the basis name
 }
 }

```

Proceed to *numeric-tail* generation.

### The Numeric-Tail Generation Algorithm

```

If (a "lossy conversion" was not flagged)
and (the long name fits within the 8.3 naming conventions)
and (the basis-name does not collide with any existing short name)
{
 The short name is only the basis-name without the numeric tail.
}
else
{
 Insert a numeric-tail "-n" to the end of the primary name such that the value of the "-n" is
 chosen so that the
 name thus formed does not collide with any existing short name and that the primary name does
 not exceed eight characters in length.
}

```

The "-n" string can range from "-1" to "-999999". The number "n" is chosen so that it is the next number in a sequence of files with similar basis-names. For example, assume the following short names existed: LETTER-1.DOC and LETTER-2.DOC. As expected, the next auto-generated name of name of this type would be LETTER-3.DOC. Assume the following short names existed: LETTER-1.DOC, LETTER-3.DOC. Again, the next auto-generated name of name of this type would be LETTER-2.DOC. However, one *absolutely cannot* count on this behavior. In a directory with a very large mix of names of this type, the selection algorithm is optimized for speed and may select another "n" based on the characteristics of short names that end in "-n" and have *similar* leading name patterns.

## Effect of Long Directory Entries on Down Level Versions of FAT

The support of long names is most important on the hard disk, however it will be supported on removable media as well. The implementation provides support for long names without breaking compatibility with the existing FAT format. A disk can be read by a down level system without any compatibility problems. An existing disk does not go through a conversion process before it can start using long names. All of the current files remain unmodified. The long name directory entries are added when a long name is created. The addition of a long name to an existing file may require the 8.3 directory entry to be moved if the required adjacent directory entries are not available.

The long name entries are as hidden as hidden or system files are on a down level system. This is enough to keep the casual user from causing problems. The user can copy the files off using the 8.3 name, and put new files on without any side effects

The interesting part of this is what happens when the disk is taken to a down level FAT system and the directory is changed. This can affect the long name entries since the down level system ignores these long names and will not ensure they are properly associated with the 8.3 names.

A down level system will only see the long name entries when searching for a label. On a down level system, the volume label will be incorrectly reported if the true volume label does not come before all of the long name entries in the root directory. This is because the long name entries also have the volume label bit set. This is unfortunate, but is not a critical problem.

If an attempt is made to remove the volume label, one of the long name directory entries may be deleted. This would be a rare occurrence. It is easily detected on an aware system. The long name entry will no longer be a valid file entry, since one or more of the long entries is marked as deleted. If the deleted entry is reused, then the attribute byte will not have the proper value for a long name entry.

If a file is renamed on a down level system, then only the short name will be renamed. The long name will not be affected. Since the long and short names must be kept consistent across the name space, it is desirable to have the long name become invalid as a result of this rename. The checksum of the 8.3 name that is kept in the long name directory provides the ability to detect this type of change. This checksum will be checked to validate the long name before it is used. Rename will cause problems only if the renamed 8.3 file name happens to have the same checksum. The checksum algorithm chosen has a relatively flat distribution across the short name space.

This rename of the 8.3 name must also not conflict with any of the long names. Otherwise a down level system could create a short name in one file that matches a long name, when case is ignored, in a different file. To prevent this, the automatic creation of an 8.3 name from a long name, that has an 8.3 format, will directly map the long name to the 8.3 name by converting the characters to upper case.

If the file is deleted, then the long name is simply orphaned. If a new file is created, the long name may be incorrectly associated with the new file name. As in the case of a rename the checksum of the 8.3 name will help prevent this incorrect association.

## Validating The Contents of a Directory

These guidelines are provided so that disk maintenance utilities can verify individual directory entries for 'correctness' while maintaining compatibility with future enhancements to the directory structure.

1. *DO NOT* look at the content of directory entry fields marked 'reserved' and assume that, if they are any value other than zero, that they are 'bad'.
2. *DO NOT* reset the content of directory entry fields marked *reserved* to zero when they contain non-zero values (under the assumption that they are "bad"). Directory entry fields are designated

*reserved*, rather than *must-be-zero*. They should be ignored by your application. These fields are intended for future extensions of the file system. By ignoring them an utility can continue to run on future versions of the operating system.

3. *DO* use the `A_LONG` attribute *first* when determining whether a directory entry is a long directory entry or a short directory entry. The following algorithm is the correct algorithm for making this determination:

```
if (((LDIR_attr & ATTR_LONG_NAME_MASK) == ATTR_LONG_NAME) && (LDIR_Ord != 0xE5))
{
 /* Found an active long name sub-component. */
}
```

4. *DO* use bits 4 and 3 of a short entry *together* when determining what type of short directory entry is being inspected. The following algorithm is the correct algorithm for making this determination:

```
if (((LDIR_attr & ATTR_LONG_NAME_MASK) != ATTR_LONG_NAME) && (LDIR_Ord != 0xE5))
{
 if ((DIR_Attr & (ATTR_DIRECTORY | ATTR_VOLUME_ID)) == 0x00)
 /* Found a file. */
 else if ((DIR_Attr & (ATTR_DIRECTORY | ATTR_VOLUME_ID)) == ATTR_DIRECTORY)
 /* Found a directory. */
 else if ((DIR_Attr & (ATTR_DIRECTORY | ATTR_VOLUME_ID)) == ATTR_VOLUME_ID)
 /* Found a volume label. */
 else
 /* Found an invalid directory entry. */
}
```

5. *DO NOT* assume that a non-zero value in the "type" field indicates a bad directory entry. Do not force the "type" field to zero.
6. Use the "checksum" field as a value to validate the directory entry. The "first cluster" field is currently being set to zero, though this might change in future.

## Other Notes Relating to FAT Directories

- Long File Name directory entries are identical on all FAT types. See the preceding sections for details.
- `DIR_FileSize` is a 32-bit field. For FAT32 volumes, your FAT file system driver must not allow a cluster chain to be created that is longer than 0x100000000 bytes, and the last byte of the last cluster in a chain that long cannot be allocated to the file. This must be done so that no file has a file size > 0xFFFFFFFF bytes. This is a fundamental limit of all FAT file systems. The maximum allowed file size on a FAT volume is 0xFFFFFFFF (4,294,967,295) bytes.
- Similarly, a FAT file system driver must not allow a directory (a file that is actually a container for other files) to be larger than  $65,536 * 32$  (2,097,152) bytes.

**NOTE:** This limit does *not* apply to the number of files in the directory. This limit is on the size of the directory itself and has nothing to do with the content of the directory. There are two reasons for this limit:

1. Because FAT directories are not sorted or indexed, it is a bad idea to create huge directories; otherwise, operations like creating a new entry (which requires every allocated directory entry to be checked to verify that the name doesn't already exist in the directory) become very slow.

2. There are many FAT file system drivers and disk utilities, including Microsoft's, that expect to be able to count the entries in a directory using a 16-bit WORD variable. For this reason, directories cannot have more than 16-bits worth of entries.

## Crash Consistency: FSCK and Journaling

As we've seen thus far, the file system manages a set of data structures to implement the expected abstractions: files, directories, and all of the other metadata needed to support the basic abstraction that we expect from a file system. Unlike most data structures (for example, those found in memory of a running program), file system data structures must **persist**, i.e., they must survive over the long haul, stored on devices that retain data despite power loss (such as hard disks or flash-based SSDs).

One major challenge faced by a file system is how to update persistent data structures despite the presence of a **power loss** or **system crash**. Specifically, what happens if, right in the middle of updating on-disk structures, someone trips over the power cord and the machine loses power? Or the operating system encounters a bug and crashes? Because of power losses and crashes, updating a persistent data structure can be quite tricky, and leads to a new and interesting problem in file system implementation, known as the **crash-consistency problem**.

This problem is quite simple to understand. Imagine you have to update two on-disk structures, *A* and *B*, in order to complete a particular operation. Because the disk only services a single request at a time, one of these requests will reach the disk first (either *A* or *B*). If the system crashes or loses power after one write completes, the on-disk structure will be left in an **inconsistent** state. And thus, we have a problem that all file systems need to solve:

### THE CRUX: HOW TO UPDATE THE DISK DESPITE CRASHES

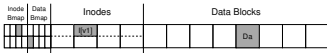
The system may crash or lose power between any two writes, and thus the on-disk state may only partially get updated. After the crash, the system boots and wishes to mount the file system again (in order to access files and such). Given that crashes can occur at arbitrary points in time, how do we ensure the file system keeps the on-disk image in a reasonable state?

In this chapter, we'll describe this problem in more detail, and look at some methods file systems have used to overcome it. We'll begin by examining the approach taken by older file systems, known as **fsck** or the **file system checker**. We'll then turn our attention to another approach, known as **journaling** (also known as **write-ahead logging**), a technique which adds a little bit of overhead to each write but recovers more quickly from crashes or power losses. We will discuss the basic machinery of journaling, including a few different flavors of journaling that Linux ext3 [T98,PAA05] (a relatively modern journaling file system) implements.

## 4.2.1 A Detailed Example

To kick off our investigation of journaling, let's look at an example. We'll need to use a **workload** that updates on-disk structures in some way. Assume here that the workload is simple: the append of a single data block to an existing file. The append is accomplished by opening the file, calling `lseek()` to move the file offset to the end of the file, and then issuing a single 4KB write to the file before closing it.

Let's also assume we are using standard simple file system structures on the disk, similar to file systems we have seen before. This tiny example includes an **inode bitmap** (with just 8 bits, one per inode), a **data bitmap** (also 8 bits, one per data block), inodes (8 total, numbered 0 to 7, and spread across four blocks), and data blocks (8 total, numbered 0 to 7). Here is a diagram of this file system:



If you look at the structures in the picture, you can see that a single inode is allocated (inode number 2), which is marked in the inode bitmap, and a single allocated data block (data block 4), also marked in the data bitmap. The inode is denoted `[v1]`, as it is the first version of this inode; it will soon be updated (due to the workload described above).

Let's peek inside this simplified inode too. Inside of `[v1]`, we see:

```
owner : ramzi
permissions : read-write
size : 1
pointer : 4
pointer : null
pointer : null
pointer : null
```

In this simplified inode, the `size` of the file is 1 (it has one block allocated), the first direct pointer points to block 4 (the first data block of the file, `Da`), and all three other direct pointers are set to `null` (indicating



that they are not used). Of course, real inodes have many more fields; see previous chapters for more information.

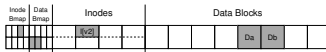
When we append to the file, we are adding a new data block to it, and thus must update three on-disk structures: the inode (which must point to the new block as well as have a bigger size due to the append), the new data block Db, and a new version of the data bitmap (call it B[v2]) to indicate that the new data block has been allocated.

Thus, in the memory of the system, we have three blocks which we must write to disk. The updated inode (inode version 2, or I[v2] for short) now looks like this:

```
owner : remzi
permissions : read-write
size : 2
pointer : 4
pointer : 5
pointer : null
pointer : null
```

The updated data bitmap (B[v2]) now looks like this: 00001100. Finally, there is the data block (Db), which is just filled with whatever it is users put into files. Stolen music perhaps?

What we would like is for the final on-disk image of the file system to look like this:



To achieve this transition, the file system must perform three separate writes to the disk, one each for the inode (I[v2]), bitmap (B[v2]), and data block (Db). Note that these writes usually don't happen immediately when the user issues a `write()` system call; rather, the dirty inode, bitmap, and new data will sit in main memory (in the **page cache** or **buffer cache**) for some time first; then, when the file system finally decides to write them to disk (after say 5 seconds or 30 seconds), the file system will issue the requisite write requests to the disk. Unfortunately, a crash may occur and thus interfere with these updates to the disk. In particular, if a crash happens after one or two of these writes have taken place, but not all three, the file system could be left in a funny state.

### Crash Scenarios

To understand the problem better, let's look at some example crash scenarios. Imagine only a single write succeeds; there are thus three possible outcomes, which we list here:

- **Just the data block (Db) is written to disk.** In this case, the data is on disk, but there is no inode that points to it and no bitmap that even says the block is allocated. Thus, it is as if the write never occurred. This case is not a problem at all, from the perspective of file-system crash consistency<sup>1</sup>.
- **Just the updated inode (I[v2]) is written to disk.** In this case, the inode points to the disk address (5) where Db was about to be written, but Db has not yet been written there. Thus, if we trust that pointer, we will read **garbage** data from the disk (the old contents of disk address 5).

Further, we have a new problem, which we call a **file-system inconsistency**. The on-disk bitmap is telling us that data block 5 has not been allocated, but the inode is saying that it has. The disagreement between the bitmap and the inode is an inconsistency in the data structures of the file system; to use the file system, we must somehow resolve this problem (more on that below).

- **Just the updated bitmap (B[v2]) is written to disk.** In this case, the bitmap indicates that block 5 is allocated, but there is no inode that points to it. Thus the file system is inconsistent again; if left unresolved, this write would result in a **space leak**, as block 5 would never be used by the file system.

There are also three more crash scenarios in this attempt to write three blocks to disk. In these cases, two writes succeed and the last one fails:

- **The inode (I[v2]) and bitmap (B[v2]) are written to disk, but not data (Db).** In this case, the file system metadata is completely consistent: the inode has a pointer to block 5, the bitmap indicates that 5 is in use, and thus everything looks OK from the perspective of the file system's metadata. But there is one problem: 5 has garbage in it again.
- **The inode (I[v2]) and the data block (Db) are written, but not the bitmap (B[v2]).** In this case, we have the inode pointing to the correct data on disk, but again have an inconsistency between the inode and the old version of the bitmap (B1). Thus, we once again need to resolve the problem before using the file system.
- **The bitmap (B[v2]) and data block (Db) are written, but not the inode (I[v2]).** In this case, we again have an inconsistency between the inode and the data bitmap. However, even though the block was written and the bitmap indicates its usage, we have no idea which file it belongs to, as no inode points to the file.

<sup>1</sup>However, it might be a problem for the user, who just lost some data!

### The Crash Consistency Problem

Hopefully, from these crash scenarios, you can see the many problems that can occur to our on-disk file system image because of crashes: we can have inconsistency in file system data structures; we can have space leaks; we can return garbage data to a user; and so forth. What we'd like to do ideally is move the file system from one consistent state (e.g., before the file got appended to) to another **atomically** (e.g., after the inode, bitmap, and new data block have been written to disk). Unfortunately, we can't do this easily because the disk only commits one write at a time, and crashes or power loss may occur between these updates. We call this general problem the **crash-consistency problem** (we could also call it the **consistent-update problem**).

#### 42.2 Solution #1: The File System Checker

Early file systems took a simple approach to crash consistency. Basically, they decided to let inconsistencies happen and then fix them later (when rebooting). A classic example of this lazy approach is found in a tool that does this: **fsck**<sup>2</sup>. *fsck* is a UNIX tool for finding such inconsistencies and repairing them [M86]; similar tools to check and repair a disk partition exist on different systems. Note that such an approach can't fix all problems; consider, for example, the case above where the file system looks consistent but the inode points to garbage data. The only real goal is to make sure the file system metadata is internally consistent.

The tool *fsck* operates in a number of phases, as summarized in McKusick and Kowalski's paper [MK96]. It is run *before* the file system is mounted and made available (*fsck* assumes that no other file-system activity is on-going while it runs); once finished, the on-disk file system should be consistent and thus can be made accessible to users.

Here is a basic summary of what *fsck* does:

- **Superblock:** *fsck* first checks if the superblock looks reasonable, mostly doing sanity checks such as making sure the file system size is greater than the number of blocks that have been allocated. Usually the goal of these sanity checks is to find a suspect (corrupt) superblock; in this case, the system (or administrator) may decide to use an alternate copy of the superblock.
- **Free blocks:** Next, *fsck* scans the inodes, indirect blocks, double indirect blocks, etc., to build an understanding of which blocks are currently allocated within the file system. It uses this knowledge to produce a correct version of the allocation bitmaps; thus, if there is any inconsistency between bitmaps and inodes, it is resolved by trusting the information within the inodes. The same type of check is performed for all the inodes, making sure that all inodes that look like they are in use are marked as such in the inode bitmaps.

<sup>2</sup>Pronounced either "eff-ess-see-kay", "eff-ess-check", or, if you don't like the tool, "eff-suck". Yes, serious professional people use this term.

- **Inode state:** Each inode is checked for corruption or other problems. For example, `fsck` makes sure that each allocated inode has a valid type field (e.g., regular file, directory, symbolic link, etc.). If there are problems with the inode fields that are not easily fixed, the inode is considered suspect and cleared by `fsck`; the inode bitmap is correspondingly updated.
- **Inode links:** `fsck` also verifies the link count of each allocated inode. As you may recall, the link count indicates the number of different directories that contain a reference (i.e., a link) to this particular file. To verify the link count, `fsck` scans through the entire directory tree, starting at the root directory, and builds its own link counts for every file and directory in the file system. If there is a mismatch between the newly-calculated count and that found within an inode, corrective action must be taken, usually by fixing the count within the inode. If an allocated inode is discovered but no directory refers to it, it is moved to the `lost+found` directory.
- **Duplicates:** `fsck` also checks for duplicate pointers, i.e., cases where two different inodes refer to the same block. If one inode is obviously bad, it may be cleared. Alternately, the pointed-to block could be copied, thus giving each inode its own copy as desired.
- **Bad blocks:** A check for bad block pointers is also performed while scanning through the list of all pointers. A pointer is considered “bad” if it obviously points to something outside its valid range, e.g., it has an address that refers to a block greater than the partition size. In this case, `fsck` can’t do anything too intelligent; it just removes (clears) the pointer from the inode or indirect block.
- **Directory checks:** `fsck` does not understand the contents of user files; however, directories hold specifically formatted information created by the file system itself. Thus, `fsck` performs additional integrity checks on the contents of each directory, making sure that “.” and “..” are the first entries, that each inode referred to in a directory entry is allocated, and ensuring that no directory is linked to more than once in the entire hierarchy.

As you can see, building a working `fsck` requires intricate knowledge of the file system; making sure such a piece of code works correctly in all cases can be challenging [G+08]. However, `fsck` (and similar approaches) have a bigger and perhaps more fundamental problem: they are *too slow*. With a very large disk volume, scanning the entire disk to find all the allocated blocks and read the entire directory tree may take many minutes or hours. Performance of `fsck`, as disks grew in capacity and RAID5s grew in popularity, became prohibitive (despite recent advances [M+13]).

At a higher level, the basic premise of `fsck` seems just a tad irrational. Consider our example above, where just three blocks are written to the disk; it is incredibly expensive to scan the entire disk to fix problems that occurred during an update of just three blocks. This situation is akin to dropping your keys on the floor in your bedroom, and then com-

mencing a *search-the-entire-house-for-keys* recovery algorithm, starting in the basement and working your way through every room. It works but is wasteful. Thus, as disks (and RAID)s grew, researchers and practitioners started to look for other solutions.

#### 42.3 Solution #2: Journaling (or Write-Ahead Logging)

Probably the most popular solution to the consistent update problem is to steal an idea from the world of database management systems. That idea, known as **write-ahead logging**, was invented to address exactly this type of problem. In file systems, we usually call write-ahead logging **journaling** for historical reasons. The first file system to do this was Cedar [H87], though many modern file systems use the idea, including Linux ext3 and ext4, reiserfs, IBM's JFS, SGI's XFS, and Windows NTFS.

The basic idea is as follows. When updating the disk, before overwriting the structures in place, first write down a little note (somewhere else on the disk, in a well-known location) describing what you are about to do. Writing this note is the “write ahead” part, and we write it to a structure that we organize as a “log”; hence, write-ahead logging.

By writing the note to disk, you are guaranteeing that if a crash takes places during the update (overwrite) of the structures you are updating, you can go back and look at the note you made and try again; thus, you will know exactly what to fix (and how to fix it) after a crash, instead of having to scan the entire disk. By design, journaling thus adds a bit of work during updates to greatly reduce the amount of work required during recovery.

We'll now describe how **Linux ext3**, a popular journaling file system, incorporates journaling into the file system. Most of the on-disk structures are identical to **Linux ext2**, e.g., the disk is divided into block groups, and each block group has an inode and data bitmap as well as inodes and data blocks. The new key structure is the journal itself, which occupies some small amount of space within the partition or on another device. Thus, an ext2 file system (without journaling) looks like this:

|       |         |         |     |         |
|-------|---------|---------|-----|---------|
| Super | Group 0 | Group 1 | ... | Group N |
|-------|---------|---------|-----|---------|

Assuming the journal is placed within the same file system image (though sometimes it is placed on a separate device, or as a file within the file system), an ext3 file system with a journal looks like this:

|       |         |         |         |     |         |
|-------|---------|---------|---------|-----|---------|
| Super | Journal | Group 0 | Group 1 | ... | Group N |
|-------|---------|---------|---------|-----|---------|

The real difference is just the presence of the journal, and of course, how it is used.

### Data Journaling

Let's look at a simple example to understand how **data journaling** works. Data journaling is available as a mode with the Linux ext3 file system, from which much of this discussion is based.

Say we have our canonical update again, where we wish to write the inode (I[v2]), bitmap (B[v2]), and data block (Db) to disk again. Before writing them to their final disk locations, we are now first going to write them to the log (a.k.a. journal). This is what this will look like in the log:



You can see we have written five blocks here. The transaction begin (TxB) tells us about this update, including information about the pending update to the file system (e.g., the final addresses of the blocks I[v2], B[v2], and Db), as well as some kind of **transaction identifier (TID)**. The middle three blocks just contain the exact contents of the blocks themselves; this is known as **physical logging** as we are putting the exact physical contents of the update in the journal (an alternate idea, **logical logging**, puts a more compact logical representation of the update in the journal, e.g., "this update wishes to append data block Db to file X", which is a little more complex but can save space in the log and perhaps improve performance). The final block (TxE) is a marker of the end of this transaction, and will also contain the TID.

Once this transaction is safely on disk, we are ready to overwrite the old structures in the file system; this process is called **checkpointing**. Thus, to **checkpoint** the file system (i.e., bring it up to date with the pending update in the journal), we issue the writes I[v2], B[v2], and Db to their disk locations as seen above; if these writes complete successfully, we have successfully checkpointed the file system and are basically done. Thus, our initial sequence of operations:

1. **Journal write:** Write the transaction, including a transaction-begin block, all pending data and metadata updates, and a transaction-end block, to the log; wait for these writes to complete.
2. **Checkpoint:** Write the pending metadata and data updates to their final locations in the file system.

In our example, we would write TxB, I[v2], B[v2], Db, and TxE to the journal first. When these writes complete, we would complete the update by checkpointing I[v2], B[v2], and Db, to their final locations on disk.

Things get a little trickier when a crash occurs during the writes to the journal. Here, we are trying to write the set of blocks in the transaction (e.g., TxB, I[v2], B[v2], Db, TxE) to disk. One simple way to do this would be to issue each one at a time, waiting for each to complete, and then issuing the next. However, this is slow. Ideally, we'd like to issue

## ASIDE: FORCING WRITES TO DISK

To enforce ordering between two disk writes, modern file systems have to take a few extra precautions. In olden times, forcing ordering between two writes, *A* and *B*, was easy: just issue the write of *A* to the disk, wait for the disk to interrupt the OS when the write is complete, and then issue the write of *B*.

Things got slightly more complex due to the increased use of write caches within disks. With write buffering enabled (sometimes called **immediate reporting**), a disk will inform the OS the write is complete when it simply has been placed in the disk's memory cache, and has not yet reached disk. If the OS then issues a subsequent write, it is not guaranteed to reach the disk after previous writes; thus ordering between writes is not preserved. One solution is to disable write buffering. However, more modern systems take extra precautions and issue explicit **write barriers**; such a barrier, when it completes, guarantees that all writes issued before the barrier will reach disk before any writes issued after the barrier.

All of this machinery requires a great deal of trust in the correct operation of the disk. Unfortunately, recent research shows that some disk manufacturers, in an effort to deliver "higher performing" disks, explicitly ignore write-barrier requests, thus making the disks seemingly run faster but at the risk of incorrect operation [C+13, R+11]. As Kahan said, the fast almost always beats out the slow, even if the fast is wrong.

all five block writes at once, as this would turn five writes into a single sequential write and thus be faster. However, this is unsafe, for the following reason: given such a big write, the disk internally may perform scheduling and complete small pieces of the big write in any order. Thus, the disk internally may (1) write TxB, I[v2], B[v2], and TxE and only later (2) write Db. Unfortunately, if the disk loses power between (1) and (2), this is what ends up on disk:



Why is this a problem? Well, the transaction looks like a valid transaction (it has a begin and an end with matching sequence numbers). Further, the file system can't look at that fourth block and know it is wrong; after all, it is arbitrary user data. Thus, if the system now reboots and runs recovery, it will replay this transaction, and ignorantly copy the contents of the garbage block '??' to the location where Db is supposed to live. This is bad for arbitrary user data in a file; it is much worse if it happens to a critical piece of file system, such as the superblock, which could render the file system unmountable.

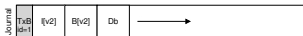
#### ASIDE: OPTIMIZING LOG WRITES

You may have noticed a particular inefficiency of writing to the log. Namely, the file system first has to write out the transaction-begin block and contents of the transaction; only after these writes complete can the file system send the transaction-end block to disk. The performance impact is clear, if you think about how a disk works: usually an extra rotation is incurred (think about why).

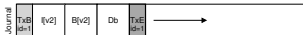
One of our former graduate students, Vijayan Prabhakaran, had a simple idea to fix this problem [P+05]. When writing a transaction to the journal, include a checksum of the contents of the journal in the begin and end blocks. Doing so enables the file system to write the entire transaction at once, without incurring a wait; if, during recovery, the file system sees a mismatch in the computed checksum versus the stored checksum in the transaction, it can conclude that a crash occurred during the write of the transaction and thus discard the file-system update. Thus, with a small tweak in the write protocol and recovery system, a file system can achieve faster common-case performance; on top of that, the system is slightly more reliable, as any reads from the journal are now protected by a checksum.

This simple fix was attractive enough to gain the notice of Linux file system developers, who then incorporated it into the next generation Linux file system, called (you guessed it!) **Linux ext4**. It now ships on millions of machines worldwide, including the Android handheld platform. Thus, every time you write to disk on many Linux-based systems, a little code developed at Wisconsin makes your system a little faster and more reliable.

To avoid this problem, the file system issues the transactional write in two steps. First, it writes all blocks except the TxE block to the journal, issuing these writes all at once. When these writes complete, the journal will look something like this (assuming our append workload again):



When those writes complete, the file system issues the write of the TxE block, thus leaving the journal in this final, safe state:



An important aspect of this process is the atomicity guarantee provided by the disk. It turns out that the disk guarantees that any 512-byte



write will either happen or not (and never be half-written); thus, to make sure the write of TxE is atomic, one should make it a single 512-byte block. Thus, our current protocol to update the file system, with each of its three phases labeled:

1. **Journal write:** Write the contents of the transaction (including TxB, metadata, and data) to the log; wait for these writes to complete.
2. **Journal commit:** Write the transaction commit block (containing TxE) to the log; wait for write to complete; transaction is said to be **committed**.
3. **Checkpoint:** Write the contents of the update (metadata and data) to their final on-disk locations.

### Recovery

Let's now understand how a file system can use the contents of the journal to **recover** from a crash. A crash may happen at any time during this sequence of updates. If the crash happens before the transaction is written safely to the log (i.e., before Step 2 above completes), then our job is easy: the pending update is simply skipped. If the crash happens after the transaction has committed to the log, but before the checkpoint is complete, the file system can **recover** the update as follows. When the system boots, the file system recovery process will scan the log and look for transactions that have committed to the disk; these transactions are thus **replayed** (in order), with the file system again attempting to write out the blocks in the transaction to their final on-disk locations. This form of logging is one of the simplest forms there is, and is called **redo logging**. By recovering the committed transactions in the journal, the file system ensures that the on-disk structures are consistent, and thus can proceed by mounting the file system and readying itself for new requests.

Note that it is fine for a crash to happen at any point during checkpointing, even after some of the updates to the final locations of the blocks have completed. In the worst case, some of these updates are simply performed again during recovery. Because recovery is a rare operation (only taking place after an unexpected system crash), a few redundant writes are nothing to worry about<sup>3</sup>.

### Batching Log Updates

You might have noticed that the basic protocol could add a lot of extra disk traffic. For example, imagine we create two files in a row, called `file1` and `file2`, in the same directory. To create one file, one has to update a number of on-disk structures, minimally including: the inode bitmap (to allocate a new inode), the newly-created inode of the file, the

<sup>3</sup>Unless you worry about everything, in which case we can't help you. Stop worrying so much, it is unhealthy! But now you're probably worried about over-worrying.

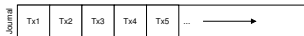
data block of the parent directory containing the new directory entry, as well as the parent directory inode (which now has a new modification time). With journaling, we logically commit all of this information to the journal for each of our two file creations; because the files are in the same directory, and assuming they even have inodes within the same inode block, this means that if we're not careful, we'll end up writing these same blocks over and over.

To remedy this problem, some file systems do not commit each update to disk one at a time (e.g., Linux ext3); rather, one can buffer all updates into a global transaction. In our example above, when the two files are created, the file system just marks the in-memory inode bitmap, inodes of the files, directory data, and directory inode as dirty, and adds them to the list of blocks that form the current transaction. When it is finally time to write these blocks to disk (say, after a timeout of 5 seconds), this single global transaction is committed containing all of the updates described above. Thus, by buffering updates, a file system can avoid excessive write traffic to disk in many cases.

### Making The Log Finite

We thus have arrived at a basic protocol for updating file-system on-disk structures. The file system buffers updates in memory for some time; when it is finally time to write to disk, the file system first carefully writes out the details of the transaction to the journal (a.k.a. write-ahead log); after the transaction is complete, the file system checkpoints those blocks to their final locations on disk.

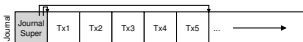
However, the log is of a finite size. If we keep adding transactions to it (as in this figure), it will soon fill. What do you think happens then?



Two problems arise when the log becomes full. The first is simpler, but less critical: the larger the log, the longer recovery will take, as the recovery process must replay all the transactions within the log (in order) to recover. The second is more of an issue: when the log is full (or nearly full), no further transactions can be committed to the disk, thus making the file system "less than useful" (i.e., useless).

To address these problems, journaling file systems treat the log as a circular data structure, re-using it over and over; this is why the journal is sometimes referred to as a **circular log**. To do so, the file system must take action some time after a checkpoint. Specifically, once a transaction has been checkpointed, the file system should free the space it was occupying within the journal, allowing the log space to be reused. There are many ways to achieve this end; for example, you could simply mark the

oldest and newest non-checkpointed transactions in the log in a **journal superblock**; all other space is free. Here is a graphical depiction:



In the journal superblock (not to be confused with the main file system superblock), the journaling system records enough information to know which transactions have not yet been checkpointed, and thus reduces recovery time as well as enables re-use of the log in a circular fashion. And thus we add another step to our basic protocol:

1. **Journal write:** Write the contents of the transaction (containing TxB and the contents of the update) to the log; wait for these writes to complete.
2. **Journal commit:** Write the transaction commit block (containing TxE) to the log; wait for the write to complete; the transaction is now committed.
3. **Checkpoint:** Write the contents of the update to their final locations within the file system.
4. **Free:** Some time later, mark the transaction free in the journal by updating the journal superblock.

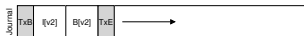
Thus we have our final data journaling protocol. But there is still a problem: we are writing each data block to the disk *twice*, which is a heavy cost to pay, especially for something as rare as a system crash. Can you figure out a way to retain consistency without writing data twice?

### Metadata Journaling

Although recovery is now fast (scanning the journal and replaying a few transactions as opposed to scanning the entire disk), normal operation of the file system is slower than we might desire. In particular, for each write to disk, we are now also writing to the journal first, thus doubling write traffic; this doubling is especially painful during sequential write workloads, which now will proceed at half the peak write bandwidth of the drive. Further, between writes to the journal and writes to the main file system, there is a costly seek, which adds noticeable overhead for some workloads.

Because of the high cost of writing every data block to disk twice, people have tried a few different things in order to speed up performance. For example, the mode of journaling we described above is often called **data journaling** (as in Linux ext3), as it journals all user data (in addition to the metadata of the file system). A simpler (and more common) form of journaling is sometimes called **ordered journaling** (or just **metadata**

journaling), and it is nearly the same, except that user data is *not* written to the journal. Thus, when performing the same update as above, the following information would be written to the journal:



The data block Db, previously written to the log, would instead be written to the file system proper, avoiding the extra write; given that most I/O traffic to the disk is data, not writing data twice substantially reduces the I/O load of journaling. The modification does raise an interesting question, though: when should we write data blocks to disk?

Let's again consider our example append of a file to understand the problem better. The update consists of three blocks: I[v2], B[v2], and Db. The first two are both metadata and will be logged and then checkpointed; the latter will only be written once to the file system. When should we write Db to disk? Does it matter?

As it turns out, the ordering of the data write does matter for metadata-only journaling. For example, what if we write Db to disk *after* the transaction (containing I[v2] and B[v2]) completes? Unfortunately, this approach has a problem: the file system is consistent but I[v2] may end up pointing to garbage data. Specifically, consider the case where I[v2] and B[v2] are written but Db did not make it to disk. The file system will then try to recover. Because Db is *not* in the log, the file system will replay writes to I[v2] and B[v2], and produce a consistent file system (from the perspective of file-system metadata). However, I[v2] will be pointing to garbage data, i.e., at whatever was in the slot where Db was headed.

To ensure this situation does not arise, some file systems (e.g., Linux ext3) write data blocks (of regular files) to the disk *first*, before related metadata is written to disk. Specifically, the protocol is as follows:

1. **Data write:** Write data to final location; wait for completion (the wait is optional; see below for details).
2. **Journal metadata write:** Write the begin block and metadata to the log; wait for writes to complete.
3. **Journal commit:** Write the transaction commit block (containing TxE) to the log; wait for the write to complete; the transaction (including data) is now **committed**.
4. **Checkpoint metadata:** Write the contents of the metadata update to their final locations within the file system.
5. **Free:** Later, mark the transaction free in journal superblock.

By forcing the data write first, a file system can guarantee that a pointer will never point to garbage. Indeed, this rule of “write the pointed-to object before the object that points to it” is at the core of crash consistency, and is exploited even further by other crash consistency schemes [GP94] (see below for details).

In most systems, metadata journaling (akin to ordered journaling of ext3) is more popular than full data journaling. For example, Windows NTFS and SGI's XFS both use some form of metadata journaling. Linux ext3 gives you the option of choosing either data, ordered, or unordered modes (in unordered mode, data can be written at any time). All of these modes keep metadata consistent; they vary in their semantics for data.

Finally, note that forcing the data write to complete (Step 1) before issuing writes to the journal (Step 2) is not required for correctness, as indicated in the protocol above. Specifically, it would be fine to issue data writes as well as the transaction-begin block and metadata to the journal; the only real requirement is that Steps 1 and 2 complete before the issuing of the journal commit block (Step 3).

### Tricky Case: Block Reuse

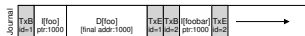
There are some interesting corner cases that make journaling more tricky, and thus are worth discussing. A number of them revolve around block reuse; as Stephen Tweedie (one of the main forces behind ext3) said:

"What's the hideous part of the entire system? ... It's deleting files. Everything to do with delete is hairy. Everything to do with delete... you have nightmares around what happens if blocks get deleted and then reallocated." [T00]

The particular example Tweedie gives is as follows. Suppose you are using some form of metadata journaling (and thus data blocks for files are not journaled). Let's say you have a directory called `foo`. The user adds an entry to `foo` (say by creating a file), and thus the contents of `foo` (because directories are considered metadata) are written to the log; assume the location of the `foo` directory data is block 1000. The log thus contains something like this:



At this point, the user deletes everything in the directory as well as the directory itself, freeing up block 1000 for reuse. Finally, the user creates a new file (say `foobar`), which ends up reusing the same block (1000) that used to belong to `foo`. The inode of `foobar` is committed to disk, as is its data; note, however, because metadata journaling is in use, only the inode of `foobar` is committed to the journal; the newly-written data in block 1000 in the file `foobar` is not journaled.



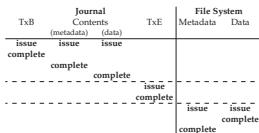


Figure 42.1: Data Journaling Timeline

Now assume a crash occurs and all of this information is still in the log. During replay, the recovery process simply replays everything in the log, including the write of directory data in block 1000; the replay thus overwrites the user data of current file `foobar` with old directory contents! Clearly this is not a correct recovery action, and certainly it will be a surprise to the user when reading the file `foobar`.

There are a number of solutions to this problem. One could, for example, never reuse blocks until the delete of said blocks is checkpointed out of the journal. What Linux ext3 does instead is to add a new type of record to the journal, known as a **revoke** record. In the case above, deleting the directory would cause a revoke record to be written to the journal. When replaying the journal, the system first scans for such revoke records; any such revoked data is never replayed, thus avoiding the problem mentioned above.

### Wrapping Up Journaling: A Timeline

Before ending our discussion of journaling, we summarize the protocols we have discussed with timelines depicting each of them. Figure 42.1 shows the protocol when journaling data as well as metadata, whereas Figure 42.2 shows the protocol when journaling only metadata.

In each figure, time increases in the downward direction, and each row in the figure shows the logical time that a write can be issued or might complete. For example, in the data journaling protocol (Figure 42.1), the writes of the transaction begin block (TxB) and the contents of the transaction can logically be issued at the same time, and thus can be completed in any order; however, the write to the transaction end block (TxE) must not be issued until said previous writes complete. Similarly, the checkpointing writes to data and metadata blocks cannot begin until the transaction end block has committed. Horizontal dashed lines show where write-ordering requirements must be obeyed.

A similar timeline is shown for the metadata journaling protocol. Note that the data write can logically be issued at the same time as the writes

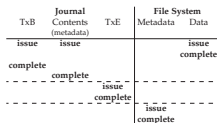


Figure 42.2: Metadata Journaling Timeline

to the transaction begin and the contents of the journal; however, it must be issued and complete before the transaction end has been issued.

Finally, note that the time of completion marked for each write in the timelines is arbitrary. In a real system, completion time is determined by the I/O subsystem, which may reorder writes to improve performance. The only guarantees about ordering that we have are those that must be enforced for protocol correctness (and are shown via the horizontal dashed lines in the figures).

#### 42.4 Solution #3: Other Approaches

We've thus far described two options in keeping file system metadata consistent: a lazy approach based on `fsck`, and a more active approach known as journaling. However, these are not the only two approaches. One such approach, known as Soft Updates [GP94], was introduced by Ganger and Fatt. This approach carefully orders all writes to the file system to ensure that the on-disk structures are never left in an inconsistent state. For example, by writing a pointed-to data block to disk *before* the inode that points to it, we can ensure that the inode never points to garbage; similar rules can be derived for all the structures of the file system. Implementing Soft Updates can be a challenge, however; whereas the journaling layer described above can be implemented with relatively little knowledge of the exact file system structures, Soft Updates requires intricate knowledge of each file system data structure and thus adds a fair amount of complexity to the system.

Another approach is known as **copy-on-write** (yes, **COW**), and is used in a number of popular file systems, including Sun's ZFS [B07]. This technique never overwrites files or directories in place; rather, it places new updates to previously unused locations on disk. After a number of updates are completed, COW file systems flip the root structure of the file system to include pointers to the newly updated structures. Doing so makes keeping the file system consistent straightforward. We'll be learning more about this technique when we discuss the log-structured file system (LFS) in a future chapter; LFS is an early example of a COW.

Another approach is one we just developed here at Wisconsin. In this technique, entitled **backpointer-based consistency** (or **BBC**), no ordering is enforced between writes. To achieve consistency, an additional **back pointer** is added to every block in the system; for example, each data block has a reference to the inode to which it belongs. When accessing a file, the file system can determine if the file is consistent by checking if the forward pointer (e.g., the address in the inode or direct block) points to a block that refers back to it. If so, everything must have safely reached disk and thus the file is consistent; if not, the file is inconsistent, and an error is returned. By adding back pointers to the file system, a new form of lazy crash consistency can be attained [C+12].

Finally, we also have explored techniques to reduce the number of times a journal protocol has to wait for disk writes to complete. Entitled **optimistic crash consistency** [C+13], this new approach issues as many writes to disk as possible and uses a generalized form of the **transaction checksum** [P+05], as well as a few other techniques, to detect inconsistencies should they arise. For some workloads, these optimistic techniques can improve performance by an order of magnitude. However, to truly function well, a slightly different disk interface is required [C+13].

## 42.5 Summary

We have introduced the problem of crash consistency, and discussed various approaches to attacking this problem. The older approach of building a file system checker works but is likely too slow to recover on modern systems. Thus, many file systems now use journaling. Journaling reduces recovery time from  $O(\text{size-of-the-disk-volume})$  to  $O(\text{size-of-the-log})$ , thus speeding recovery substantially after a crash and restart. For this reason, many modern file systems use journaling. We have also seen that journaling can come in many different forms; the most commonly used is ordered metadata journaling, which reduces the amount of traffic to the journal while still preserving reasonable consistency guarantees for both file system metadata as well as user data.



## References

- [B07] "ZFS: The Last Word in File Systems"  
Jeff Bonwick and Bill Moore  
Available: <http://www.ostep.org/Citations/zfs.Last.pdf>  
*ZFS uses copy-on-write and journaling, actually, as in some cases, logging writes to disk will perform better.*
- [C+12] "Consistency Without Ordering"  
Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau  
FAST '12, San Jose, California  
*A recent paper of ours about a new form of crash consistency based on back pointers. Read it for the exciting details!*
- [C+13] "Optimistic Crash Consistency"  
Vijay Chidambaram, Thanu S. Pillai, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau  
SOCP '13, Nemacon Woodlands Resort, PA, November 2013  
*Our work on a more optimistic and higher performance journaling protocol. For workloads that call `fsync(1)` a lot, performance can be greatly improved.*
- [CP94] "Metadata Update Performance in File Systems"  
Gregory R. Ganger and Yale N. Patt  
OSDI '94  
*A clever paper about using careful ordering of writes as the main way to achieve consistency. Implemented later in BSD-based systems.*
- [G+08] "SQCK: A Declarative File System Checker"  
Haryadi S. Gunawi, Abhishek Rajinwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau  
OSDI '08, San Diego, California  
*Our own paper on a new and better way to build a file system checker using SQL queries. We also show some problems with the existing checker, finding numerous bugs and odd behaviors, a direct result of the complexity of `fsck`.*
- [H87] "Reimplementing the Cedar File System Using Logging and Group Commit"  
Robert Hagmann  
SOCP '87, Austin, Texas, November 1987  
*The first work (that we know of) that applied write-ahead logging (a.k.a. journaling) to a file system.*
- [M+13] "ffick: The Fast File System Checker"  
Ao Ma, Chris Drago, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau  
FAST '13, San Jose, California, February 2013  
*A recent paper of ours detailing how to make `fsck` an order of magnitude faster. Some of the ideas have already been incorporated into the BSD file system checker [MK96] and are deployed today.*
- [MK96] "Fck - The UNIX File System Check Program"  
Marshall Kirk McKusick and T. J. Kowalski  
Revised in 1996  
*Describes the first comprehensive file-system checking tool, the eponymous `fsck`. Written by some of the same people who brought you FFS.*
- [MJLF84] "A Fast File System for UNIX"  
Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry  
ACM Transactions on Computing Systems.  
August 1984, Volume 2:3  
*You already know enough about FFS, right? But yeah, it is OK to reference papers like this more than once in a book.*

[P+05] "IRON File Systems"

Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

SOSP '05, Brighton, England, October 2005

*A paper mostly focused on studying how file systems react to disk failures. Towards the end, we introduce a transaction checksum to speed up logging, which was eventually adopted into Linux ext4.*

[PAA05] "Analysis and Evolution of Journaling File Systems"

Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

USENIX '05, Anaheim, California, April 2005

*An early paper we wrote analyzing how journaling file systems work.*

[R+11] "Corrected Cache Eviction and Discreet-Mode Journaling"

Abhishek Rajimwale, Vijay Chidambaram, Deepak Ramamurthi,

Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

DSN '11, Hong Kong, China, June 2011

*Our own paper on the problem of disks that buffer writes in a memory cache instead of forcing them to disk, even when explicitly told not to do that! Our solution to overcome this problem, if you want. A to be written to disk before B, first write A, then send a lot of "dummy" writes to disk, hopefully causing A to be forced to disk to make room for them in the cache. A neat if impractical solution.*

[T98] "Journaling the Linux ext2fs File System"

Stephen C. Tweedie

The Fourth Annual Linux Expo, May 1998

*Tweedie did much of the heavy lifting in adding journaling to the Linux ext2 file system; the result, not surprisingly, is called ext3. Some nice design decisions include the strong focus on backwards compatibility, e.g., you can just add a journaling file to an existing ext2 file system and then mount it as an ext3 file system.*

[T00] "EXT3, Journaling Filesystem"

Stephen Tweedie

Talk at the Ottawa Linux Symposium, July 2000

[ols.trans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html](http://ols.trans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html)

*A transcript of a talk given by Tweedie on ext3.*

[T01] "The Linux ext2 File System"

Theodore Ts'o, June, 2001.

Available: <http://c2fsprogs.sourceforge.net/ext2.html>

*A simple Linux file system based on the ideas found in FFS. For a while it was quite heavily used; now it is really just in the kernel as an example of a simple file system.*

# Key-Value and Graph Storage

**李永坤**

博士、副教授

中科大计算机学院

<http://staff.ustc.edu.cn/~ykli/>

# Key-Value (KV) Storage

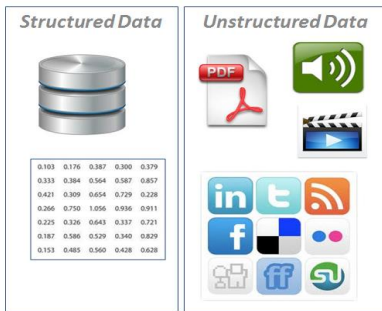
# Why key-value stores?

- The amount of data is growing exponentially
  - Facebook adds billions of new content every day
  - Hundreds of billions of e-mail messages are sent worldwide every day
  - It is estimated that the total volume of global data will reach 40ZB in 2020



# Why key-value stores?

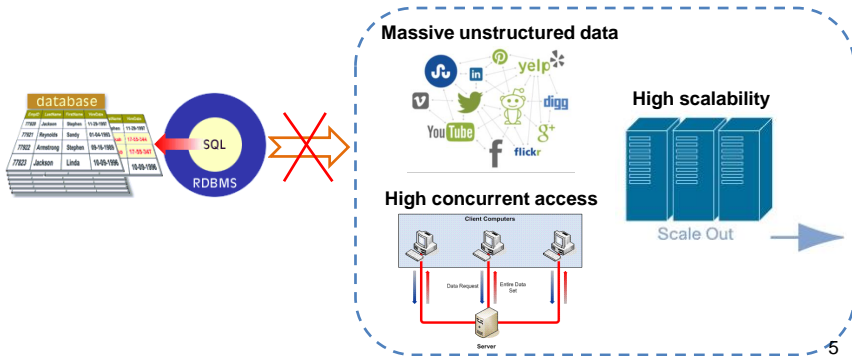
- Data format and storage requirement
  - **Unstructured data** is very common
  - In web application, unstructured data requires efficient write, query and scan service support



# Why key-value stores?

## ➤ The RDBMS is facing challenges

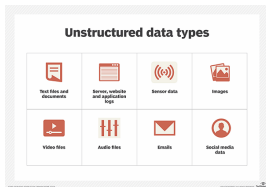
- RDBMS can't meet demand:
  - Management of massive unstructured data
  - High concurrent access to data
  - High scalability and high availability



# Why key-value stores?

- The file system is facing challenges
  - Both file system scalability and directory tree management face new challenges

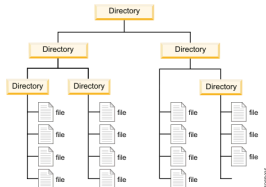
Massive unstructured small data



High overhead & Bad scalability



The management mode of the **directory tree**





# What are key-value stores

## ➤ KV stores

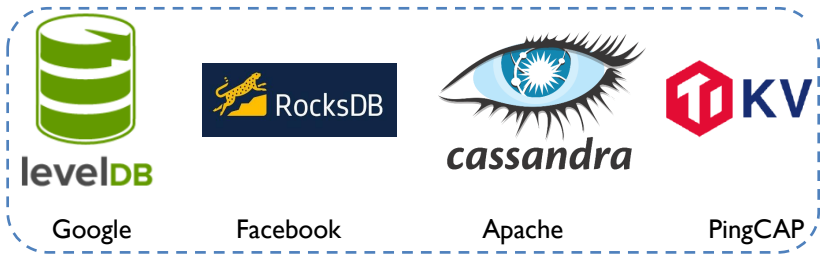
- A new storage architecture
- A flexible type of NoSQL database
- A data storage paradigm

| KEY | VALUE            |
|-----|------------------|
| K1  | AAA, BBB, CCC    |
| K2  | AAA, BBB         |
| K3  | AAA, DDD         |
| K4  | AAA,2,01/01/2015 |
| K5  | 3,ZZZ,5623       |



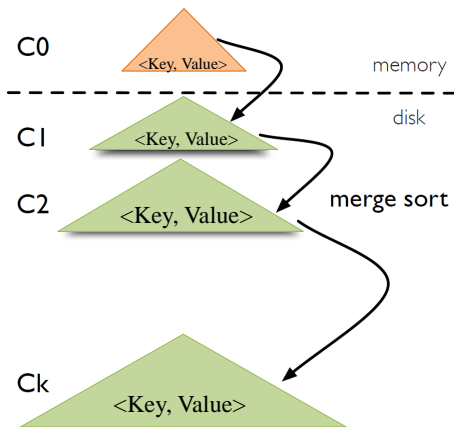
# What are key-value stores

- LSM-tree based KV stores are most common
  - optimize for write intensive workloads
  - widely deployed
    - BigTable and LevelDB at Google
    - HBase, Cassandra and RocksDB at FaceBook



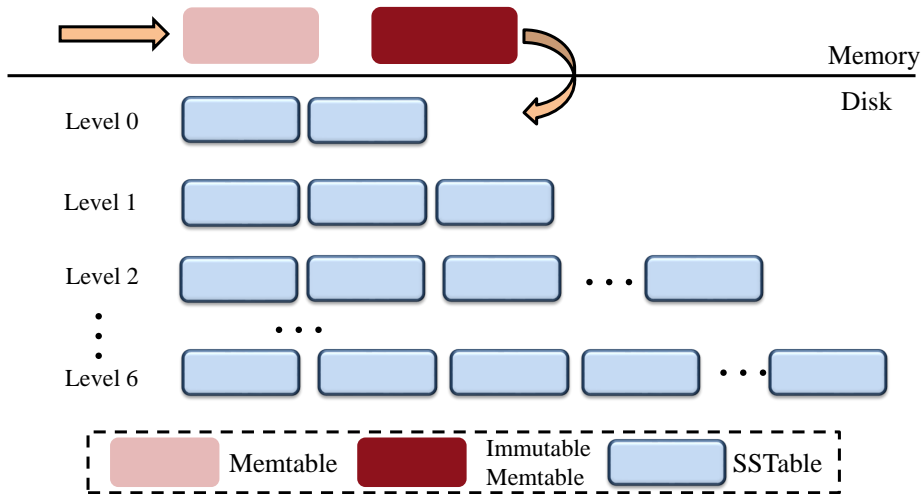
# LSM-tree

## ➤ LSM-tree structure



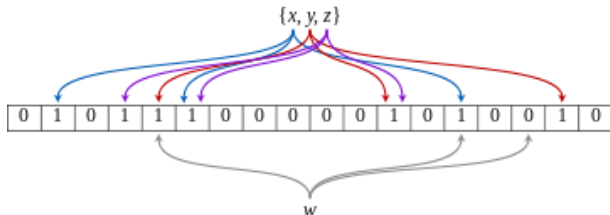
- ✓ It buffers and sorts data in  $C_0$ , then writes into  $C_1$  on disk sequentially
- ✓ When  $C_i$  is full, it merges with  $C_{i+1}$ , then writes into  $C_{i+1}$

# Read/Write Process



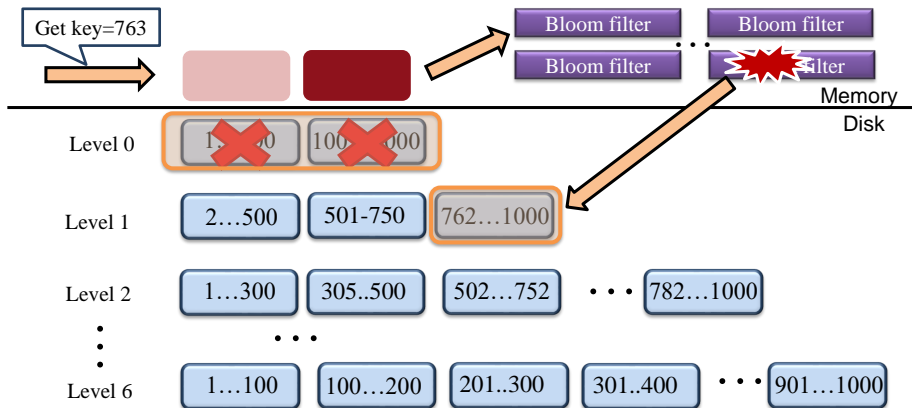
# Bloom Filter

- How to quickly determine the existence of a kv pair in each SSTable?
- Key comparison is slow
  - Bloom filter



Bloom filters have false positive with rate  $(1 - e^{-\frac{k}{b}})^k$ ,  
minimized to be **0.6185<sup>b</sup>** when  $k = \ln 2 \times b$

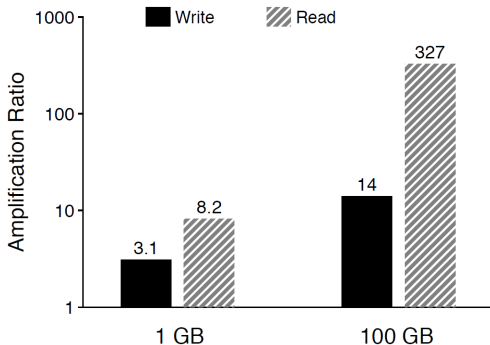
# With the Help of Bloom Filter



False positive incurs extra I/O requests

# I/O Amplification in LSM-tree

## ➤ RA/ WA



(database sizes 1 GB and 100 GB. Key size 16 B and value size 1 KB)

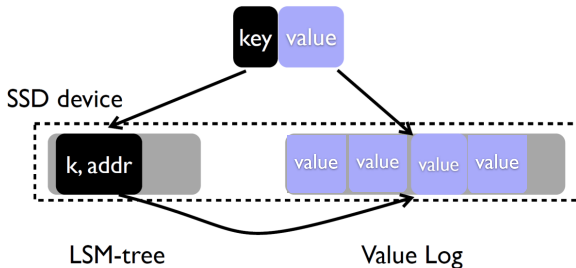
*(From Wiskey @ FAST '16)*



# Key-Value Separation

## ➤ Wisckey (FAST'16)

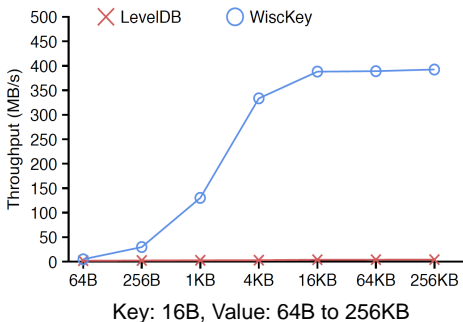
- Separates values from keys
- Values are stored in a separate log file
- Keys are stored in an LSM-tree with a addr pointer



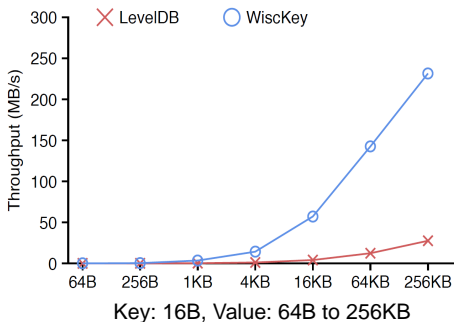
# Evaluation

## ➤ Compared with LeveDB

**Random Load**



**Random Lookup**



**It improves a lot, especially when values are large.**

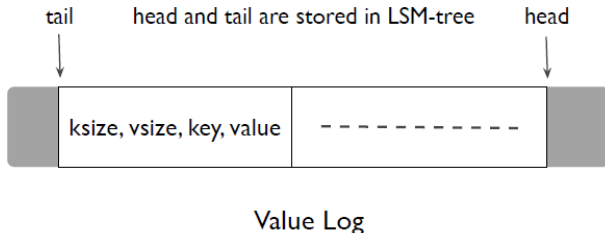
# Other Related Works

## ➤ Related works

- [1]VLDB '10 FlashStore
- [2]SIGMOD '11 SkippyStash
- [3]SOSP '11 SILT
- [4]MSST '12 BloomStore(Bloom-Filter based memory-efficient)
- [5]SIGMOD '12 bLSM
- [6]EuroSys '14 LOCS(on open-channel SSD)
- [7]ATC '15 LSM-trie
- [8]MSST'15 Atlas(Baidu's kv store)
- [9]FAST '16 WiscKey
- [10]ATC '17 TRIAD
- [11]SOSP '17 PebblesDB(Fragmented LSM Trees)
- [12]ATC '17 HiKV (hybrid index on DRAM-NVM)
- [13]CIDR '17 Optimize Space Amplification in RocksDB
- [14]SIGMOD '18 Dostoevsky (balanced performance)
- [15]FAST '19 GearDB (on hard drive)
- [16]FAST '19 SLM-DB (B+tree index on single level LSM Tree)
- [17]SOSP '19 KVell (on NVMe SSD)
- [18]ATC '20 MatrixKV
- [19]FAST '20 HotRing (hash table in memory)
- [20]FAST '21 SpanDB (on NVMe SSD)
- [21]FAST '21 REMIX (range index)
- [22]VLDB '21 Viper (hash table for persistent memory)
- [23]ATC '21 DiffKV
- [24]FAST '22 DEPART

# HashKV

- Limitations of **circular log** in key-value separation



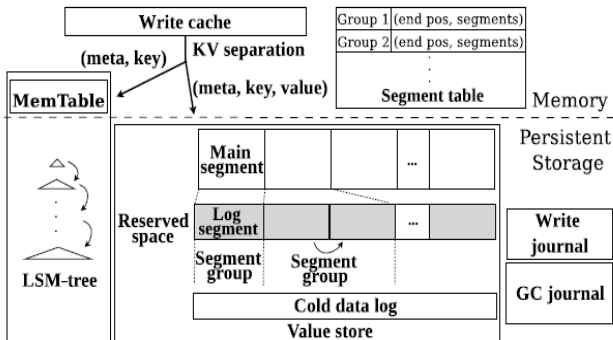
- Large GC overhead

- Data movements: need to write back valid KV
- Valid KV identification: need to access LSM-tree

# HashKV

## ➤ Core idea

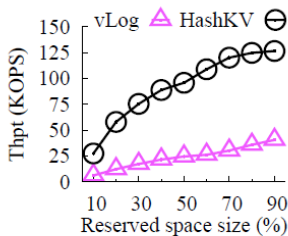
- Hash-based data grouping
- Dynamic reserved space allocation



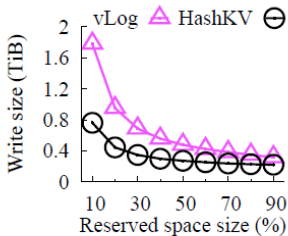
Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, Yinlong Xu. "HashKV: Enabling Efficient Updates in KV Storage via Hashing". USENIX ATC 2018.

# HashKV

- HashKV achieves 3.1-4.7x throughput of vLog and reduces the write size by 30.1-57.3%



(a) Throughput



(b) Total write size

# ElasticBF

## ➤ False positive of BF

- $0.6185^b$  (b: bits-per-key)

| Bits-per-key        | 2bits | 3bits | 4bits | 5bits | 6bits |
|---------------------|-------|-------|-------|-------|-------|
| False positive rate | 40%   | 23.7% | 14.7% | 9.2%  | 5.6%  |

## ➤ Reducing False Positive Rate

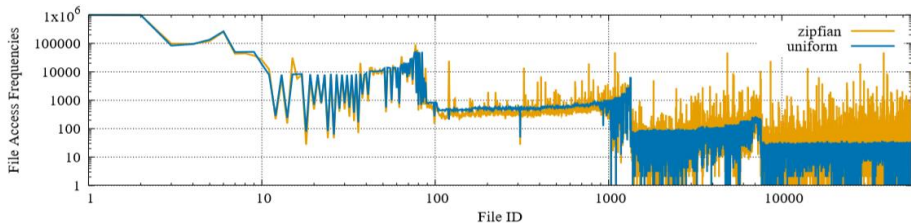
- Increase the bits-per-key used by all Bloom filters
- Large memory space overhead

| Size | Database size | Bits-per-key | Memory cost |
|------|---------------|--------------|-------------|
| 100B | 10TB(Level7)  | 8            | 100GB       |

With limited memory space, how to reduce extra I/O requests caused by false positive of Bloom filter so as to improve read performance?

# ElasticBF: Access Locality

| Key-value pair size | Size of database | Benchmark | Number of read requests |
|---------------------|------------------|-----------|-------------------------|
| 1KB                 | 100GB            | YCSB[1]   | 1 million               |



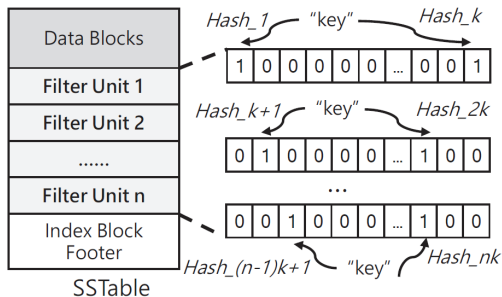
- Access frequency of SSTables in low levels are higher
- Unevenness of access frequency is very common in the same level



# ElasticBF

## ➤ Main idea

- Hot SSTables
  - Allocate more bits per key to reduce false positive rate
- Cold SSTables
  - Allocate fewer bits per key to save memory space



Non-exist as long as one filter unit gives negative return

Separability

$$(0.6185^{b/n})^n = 0.6185^b$$

# ElasticBF

## ➤ Key issues/challenges

- How to design an adjusting rule to determine the most appropriate number of filter units for each SSTable?

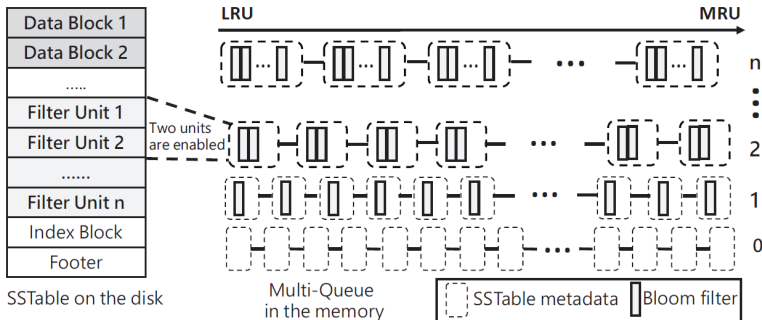
Minimize extra I/Os with hotness awareness

- How to realize a dynamic adjustment with small overhead?

Maintain a MQ in memory

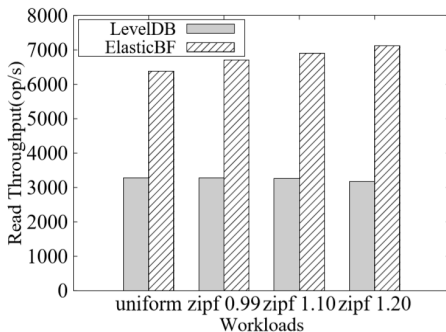
# Dynamic Adjustment

## ➤ Multiple LRU lists in memory



# Experiment Results

## ➤ Different workloads



1.84x-2.24x

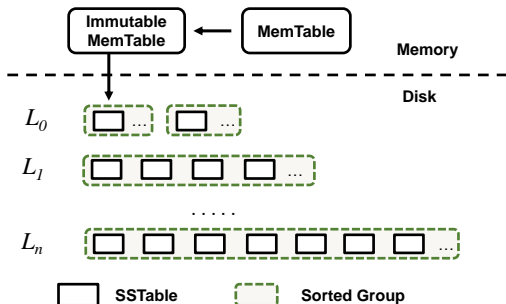
|           | uniform | zipf 0.99 | zipf 1.10 | zipf 1.20 |
|-----------|---------|-----------|-----------|-----------|
| LevelDB   | 1525595 | 1585605   | 1634752   | 1667947   |
| ElasticBF | 628225  | 578553    | 550658    | 545345    |

Table 1: Number of I/Os for data access

The number of I/O requests for data access is greatly reduced

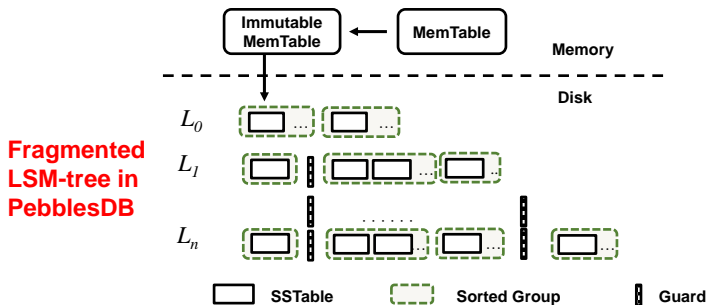
# DiffKV: Basics

- Store keys and values together
- Keys and values are fully sorted in each level
  - Compaction across levels → high I/O amplifications



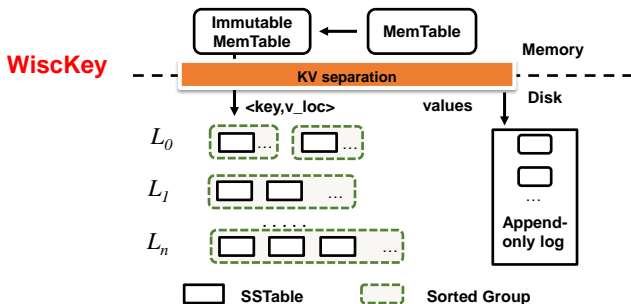
# Relaxing Fully-Sorted Ordering

- Each level is not necessarily fully sorted by keys
- e.g., PebblesDB [SOSP'17], Dostoevsky [SIGMOD'18], etc.
  - Support efficient writes, but sacrifice reads and scans



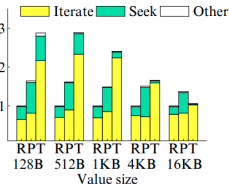
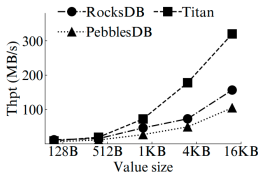
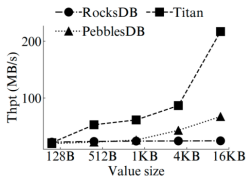
# KV Separation

- Store keys and values separately
- e.g., WiscKey, HashKV, Titan, Bourbon, etc.
  - Support efficient writes and reads, but have **poor** scan performance



# Trade-off Analysis

- Are the optimizations suitable for all conditions?
- Relax fully-sorted ordering
    - Efficient in small-to-medium values
  - KV separation
    - Suitable for large values



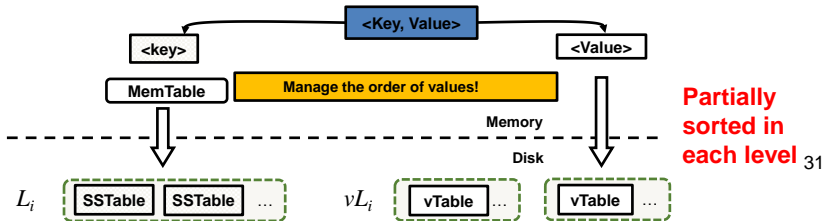
**Trade-offs between reads/writes and scans**



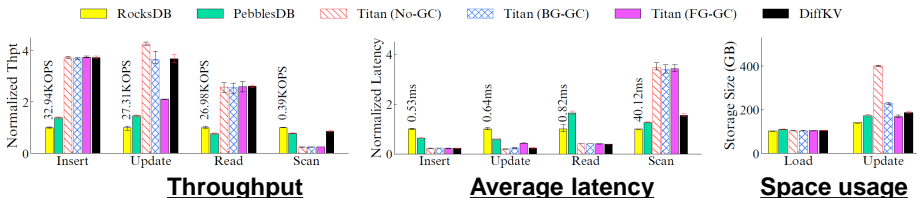
# DiffKV

## ➤ Decouple keys and values

- **vTree**: a multiple-level tree; each level has multiple sorted groups
- Values in a level are *not* fully sorted and have overlapped key ranges



# Microbenchmarks of DiffKV



- Compared to RocksDB and PebblesDB
  - 2.7-3.8x inserts; 2.3-3.7x updates; 2.6-3.4x reads
  - Comparable scan performance
- Compared to Titan
  - 3.2x scans; up to 1.7x updates; 43.2% lower scan latency
- DiffKV has acceptable space usage

# Summary on KV

- Key-value stores are common
  - LSM-tree is the basic structure
  - Large read and write amplifications
- Research efforts
  - New architectures to reduce read/write amplification
  - I/O scheduling and optimizations
  - Leverage new hardware: NVRAM/SSD
  - Application-specific design/optimization
  - Distributed KV stores
  - ...

# Graph Systems

# Graphs are common



Web Graph

Online Social Network

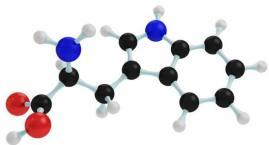


Music / Movie



Online shops

# Graphs are common



Protein Molecular Network



City Traffic Network

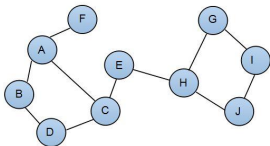


Router Network

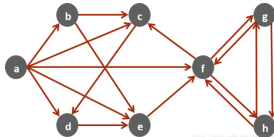
# Graph Structure

## ➤ Graph

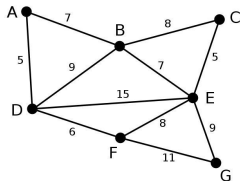
- A set of vertices and edges



Undirected Graph



Directed Graph



Weighted Undirected Graph

# Large scale of graph data

## ➤ Space requirement

| Dataset    | Type           | #Nodes | #Edges | Size |
|------------|----------------|--------|--------|------|
| Twitter    | Social Network | 53M    | 2B     | 15G  |
| Friendster | Social Network | 68M    | 2.6B   | 20G  |
| Gsh        | Web Graph      | 986M   | 33.6B  | 252G |
| UK         | Web Graph      | 778M   | 46B    | 270G |
| Clueweb    | Web Graph      | 978M   | 42.6B  | 336G |
| EU         | Web Graph      | 1071M  | 92B    | 683G |
| Kron31     | Graph500       | 2B     | 1T     | 8T   |

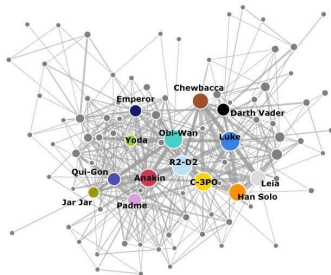
The whole graph can't fit in the memory of a single-PC



# Characteristics for graph data

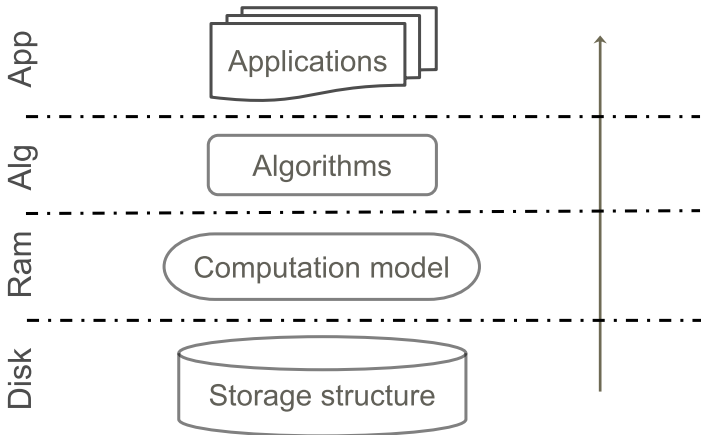
## ➤ Real life graphs

- Imbalanced degree distribution
  - Power law distribution
- Complicated structure
  - Poor locality

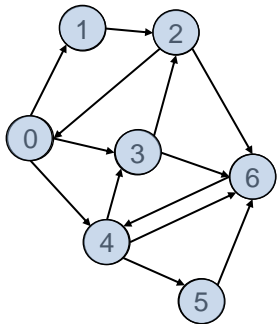


# Graph Processing System

How to develop efficient computation systems for large-scale graphs?



# Graphs in General File System



BFS

|        |
|--------|
| (0, 1) |
| (0, 3) |
| (0, 4) |
| (1, 2) |
| (2, 0) |
| (2, 6) |
| (3, 2) |
| (3, 6) |
| (4, 5) |
| (5, 6) |
| (6, 4) |

Need a lot of  
random disk  
accesses

**Performance  
bottleneck**

Edge list is stored in a file

# Graph Processing System

## ➤ GraphChi

- The first single-PC graph process system

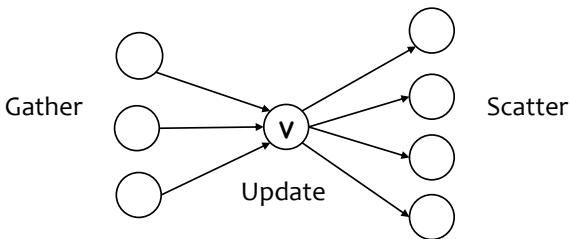
**Compute on graphs with billions of edges,  
in *a reasonable time*, on a single PC.**

Aapo Kyrola, Guy Blelloch, Carlos Guestrin. **GraphChi: Large-Scale Graph Computation on Just a PC.** USENIX OSDI 2012.

# GraphChi

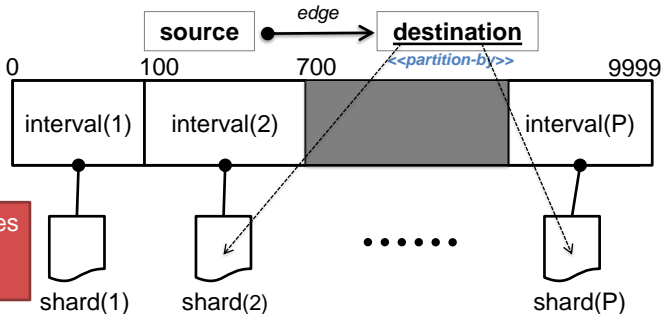
## ➤ Computation model

- Vertex-centric programming
  - “Think like a vertex”
  - Each edge and vertex is associated with a value
- **Iteration-based computation**



# GraphChi

- PSW : intervals and shards
  - Vertices are numbered from 0 to  $n-1$ 
    - **P** intervals
    - **sub-graph** = interval of vertices

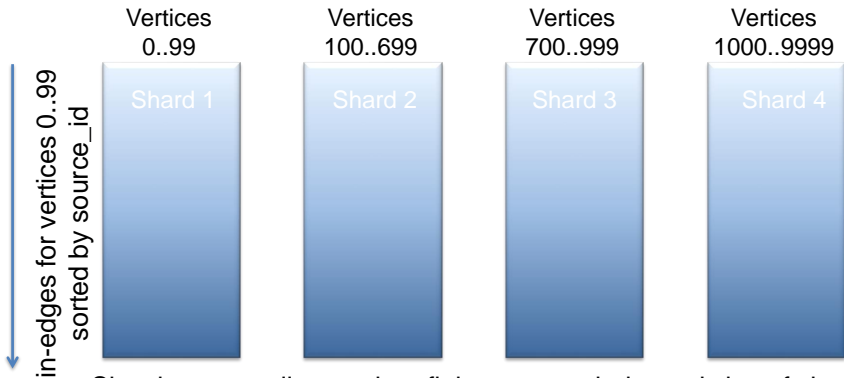


In shards, edges sorted by source node.

# GraphChi

## ➤ Layout

- Shard: in-edges for **interval** of vertices; sorted by source-id

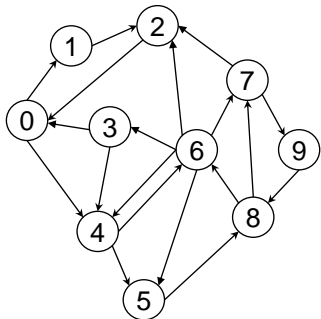


Shards are small enough to fit in memory; balanced size of shards

# GraphChi

## ➤ Layout

- Shard: in-edges for **interval** of vertices; sorted by source-id



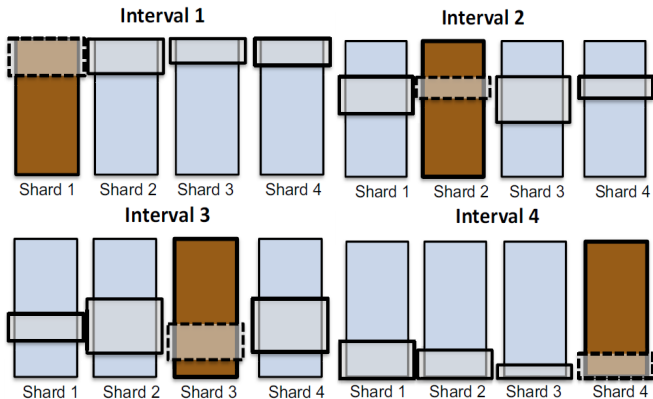
Example Graph

|                          | Shard 1 | Shard 2 | Shard 3 |
|--------------------------|---------|---------|---------|
| sorted by source_id<br>↓ | <0, 1>  | <0, 4>  | <4, 6>  |
|                          | <1, 2>  | <3, 4>  | <5, 8>  |
|                          | <2, 0>  | <4, 5>  | <6, 7>  |
|                          | <3, 0>  | <6, 3>  | <7, 9>  |
|                          | <6, 2>  | <6, 4>  | <8, 6>  |
|                          | <7, 2>  | <6, 5>  | <8, 7>  |
|                          |         |         | <9, 8>  |



# GraphChi

## ➤ Parallel Sliding Windows



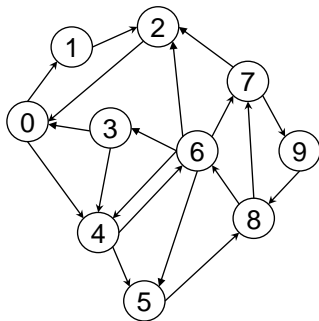
# GraphChi

- Other design details/implementations
  - Refer to the paper and source code
  - C++ implementation: 8,000 lines of code
    - Java-implementation also available

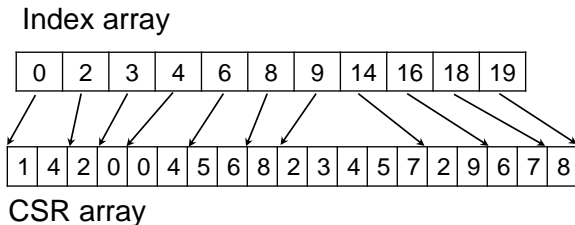
Source code and examples:  
<http://github.com/graphchi>

# CSR Format

- Highly efficient data structure to store graph
- Index array: store the offset in CSR array
  - CSR array: store the out-neighbors of vertices

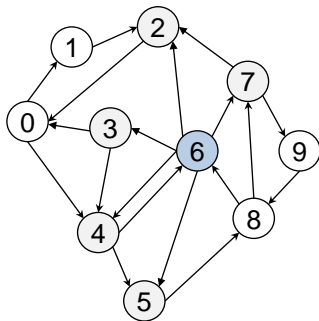


Example Graph

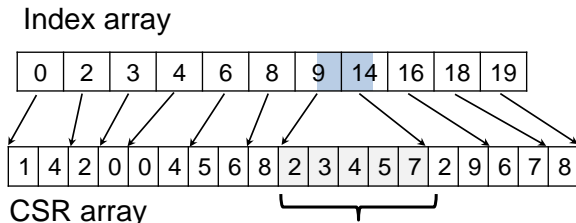


# CSR Format

- Highly efficient data structure to store graph
- High efficiency to access out-neighbors (BFS, RW)
  - E.g., access the neighbors of vertex 6



Example Graph

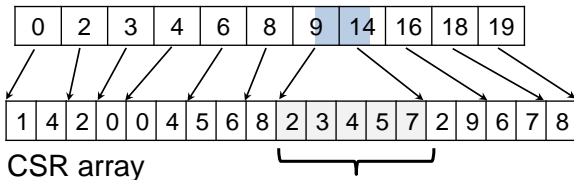


Sequentially read the out neighbors of vertex 6

# Limitations

## ➤ How to support dynamic graphs?

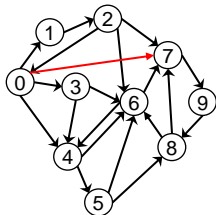
Index array



Sequentially read the out neighbors of vertex 6

## ➤ How to store attributes?

# Example

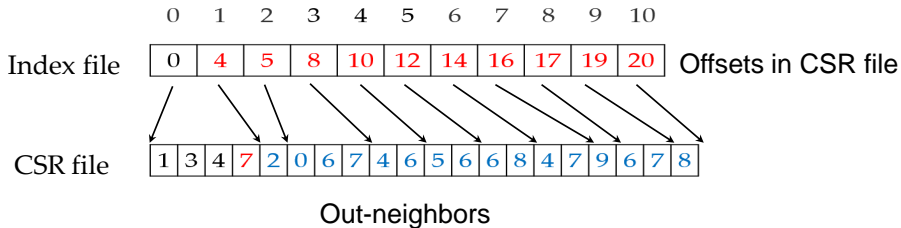


Add a new edge (0, 7) to CSR

Rewrite most part of CSR

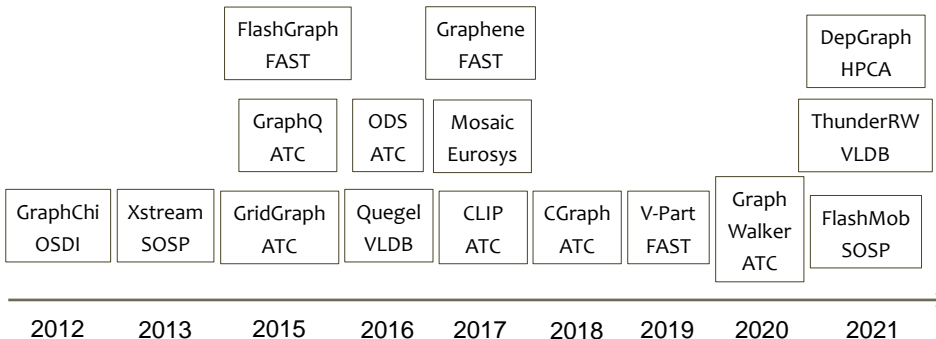
Hard to support dynamic graphs

How to support fast query on dynamic graph?



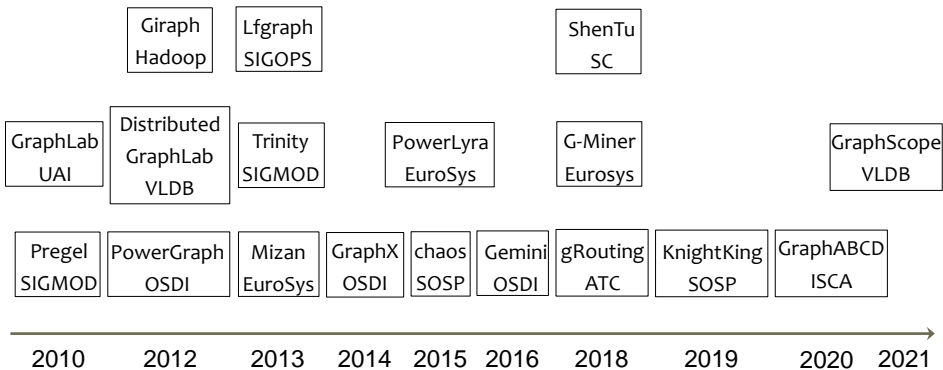
# Related Works

## ➤ Single machine graph processing systems



# Related Works

## ➤ Distributed graph processing systems





# Summary on graph systems

- Graphs become extremely large
  - Data must be kept on disk or in clusters
- Graph systems have specific features
  - Random access
  - Computation models
  - Application requirements
- A large body of works...

# Thanks!