实验一：初探Linux与环境配置

实验目的

- 学习如何在虚拟机中使用Linux；
- 学习Linux (Ubuntu) 的使用方法；
- 学习并熟练使用若干Linux指令；
- 掌握Linux内核编译方法；
- 掌握使用gdb调试内核的方法及步骤。

实验环境

- 虚拟机：VMware/VituralBox
- 操作系统：Ubuntu 20.04.4 LTS

> 系统的安装形式可以自由选择，双系统，虚拟机都可以，系统版本则推荐使用本文档所用版本。注意：由于Linux各种发行版非常庞杂且存在较大差异，因此本试验在其他Linux发行版可能会存在兼容性问题。如果想使用其他环境（如vlab）或系统（如Arch、WSL等），请根据自己的系统**自行**调整实验步骤以及具体指令，达成实验目标即可，但其中出现的兼容性问题助教**无法**保证能够一定解决。

实验时间安排

> 注：此处为实验发布时的安排计划，请以课程主页和课程群内最新公告为准

- 3.18 晚实验课，讲解实验、检查实验
- 3.25 晚实验课，检查实验
- 4.1 晚及之后实验课，补检查实验

> 补检查分数照常给分，但会**记录**此次检查未按时完成，此记录在最后综合分数时作为一种参考（即：最终分数可能会低于当前分数）。

检查时间、地点：周五晚19: 00~22: 00，电三楼406/408。

如何提问

- 请同学们先阅读《提问的智慧》。**原文链接**
- 提问前，请先**阅读报错信息**、查询在线文档，或百度。**在线文档链接**；
- 在向助教提问时，请详细描述问题，并提供相关指令及相关问题的报错截图；
- 在QQ群内提问时，如遇到长时未收到回复的情况，可能是由于消息太多可能会被刷掉，因此建议在在线文档上提问；
- 如果助教的回复成功地帮你解决了问题，请回复“问题已解决”，并将问题及解答更新到在线文档。这有助于他人解决同样的问题。

为什么要做这个实验

- 为什么要学会使用Linux?
  - Linux的安全性、稳定性更好，性能也更好，配置也更灵活方便，所以常用于服务器和开发环境。实验室和公司的服务器一般也都用Linux；
  - Linux是开源系统，代码修改方便，很多学术成果都基于Linux完成；
  - Windows是闭源系统，代码无法修改，无法进行后续实验。
- 为什么要使用虚拟机?
  - 虚拟机对你的电脑影响最低。双系统若配置不正确，可能导致无法进入Windows，而虚拟机自带的快照功能也可以解决部分误操作带来的问题。
  - 本实验并不禁止其他环境的使用，但考虑其他环境（如WSL）变数太大，比如可能存在兼容性或者其他配置问题，会耽误同学们大量时间浪费在实验内容以外的琐事，因此建议各位同学尽量保持与本试验一致或类似的环境。
- 为什么要学会编译Linux内核?
  - 这是后续实验的基础。在后续实验中，我们会让大家通过阅读Linux源码、修改Linux源码、编写模块等方式理解一个真实的操作系统是怎么工作的。

其他友情提示

- **合理安排时间，强烈不建议在ddl前赶实验。**
- 本课程的实验实践性很强，请各位大胆尝试，适当变通，能完成实验任务即可。
- pdf上文本的复制有时候会丢失或者增加不必要的空格，有时候还会增加不必要的回车，有些指令一行写不下分成两行了，一些同学就漏了第二行。如果出了bug，建议各位先仔细确认自己输入的指令是否正确。要**逐字符**比对。每次输完指令之后，请观察一下指令的输出，检查一下这个输出是不是报错。**请在复制文档上的指令之前先理解一下指令的含义。** 我们在检查实验时会抽查提问指令的含义。
- 如果你想问“为什么PDF的复制容易出现问题”，请参考 **此文章** 。
- 如果同学们遇到了问题，请先查询在线文档。在线文档地址： **链接**

第一部分：在虚拟机下安装Linux系统

> 提示：
>
> - 本部分属于初学者指南。我们不限制环境的使用。你可以使用双系统或其他linux发行版完成实验。虽然理论上影响不大，**但若其他版本的操作系统在后续实验中出现兼容性问题，可能需要你自己解决。**
> - 你可以使用VMware/VirtualBox完成实验。推荐使用Virtualbox，因为它开源且免费。相比之下，VMware的免费版(VMware Workstation Player)不具备快照功能。如果你的虚拟机不慎挂掉，解决起来可能会比较麻烦。但在一些其他课程的实验上，VirtualBox可能有bug。二者的使用方法大同小异，请各位自行权衡。

1.0 若干名词解释

宿主机(host)：主机，即物理机器。

虚拟机：在主机操作系统上运行的一个“子机器”。

Linux发行版：Linux内核与应用软件打包构成的可以使用的操作系统套装。常见的有Ubuntu、Arch、CentOS甚至Android等。

1.1 下载

虚拟机软件（二选一）：

- VMware Workstation Player的下载链接： **https://customerconnect.vmware.com/zh/downloads/details?downloadGroup=WKST-PLAYER-1622&productId=1039&rPId=82555**
- VirtualBox的下载链接： **https://www.virtualbox.org/wiki/Downloads** ，注意根据你的宿主机(host)选取合适的安装包。
- macOS宿主机若想使用其他虚拟机软件，请自行搜索安装教程。

Ubuntu 20.04.4 LTS 安装镜像文件（下载完成之后，你不需要打开镜像文件）：

- 官网链接： **https://releases.ubuntu.com/20.04/ubuntu-20.04.4-desktop-amd64.iso**
- LUG校内镜像，校内下载速度可达几十MB/s： **http://mirrors.ustc.edu.cn/ubuntu-releases/20.04/ubuntu-20.04.4-desktop-amd64.iso**

安装VMware/VirtualBox的步骤较为简单，运行安装程序即可，在此不表。

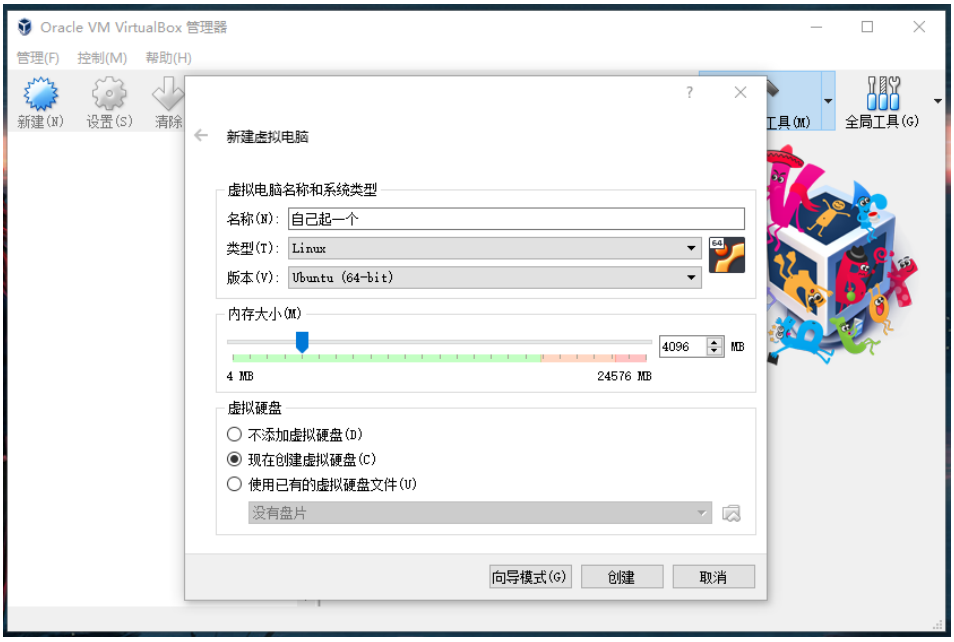
1.2 创建、安装虚拟机 (VirtualBox)

下面介绍VirtualBox创建、安装虚拟机的过程。如果你使用VMware，请直接看下一节。

1.2.1 新建虚拟机

点击“新建”创建虚拟机。设置虚拟机的名称、类型、分配内存等。同时要选择“现在创建虚拟硬盘”。

> 请至少分配2GB以上的内存给虚拟机。同时建议分配至少1/4主机内存给虚拟机。

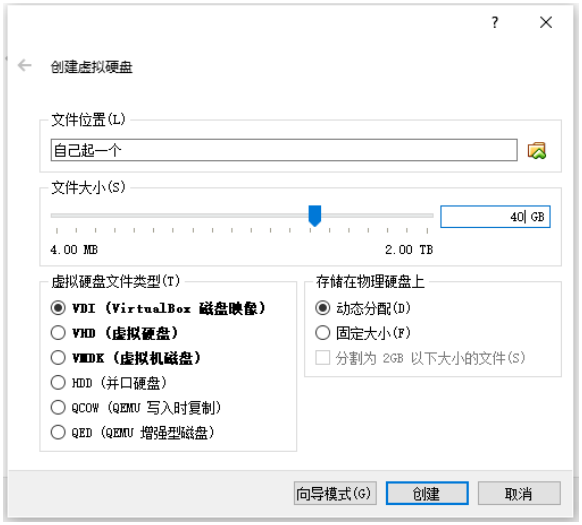


1.2.2 创建虚拟硬盘

- 文件位置：考虑到虚拟磁盘大小动辄几十GB，建议将其放在空间有富余的磁盘分区上。如果你有很多不常用的文件占用大量磁盘空间，可以考虑将其转移到 **睿客云盘** 上保存。
- 文件大小：建议30~40G。我们的实验会占用较多的磁盘空间。20GB **非常紧张**。

> 警告：如果磁盘空间不够，Linux启动会黑屏进不去图形界面，需要在命令模式下删除一些文件后重启才能进入图形界面。一些虚拟机具备“扩展磁盘容量”的功能，但是根据实际测试，发现很多时候反而会让虚拟机直接黑屏。

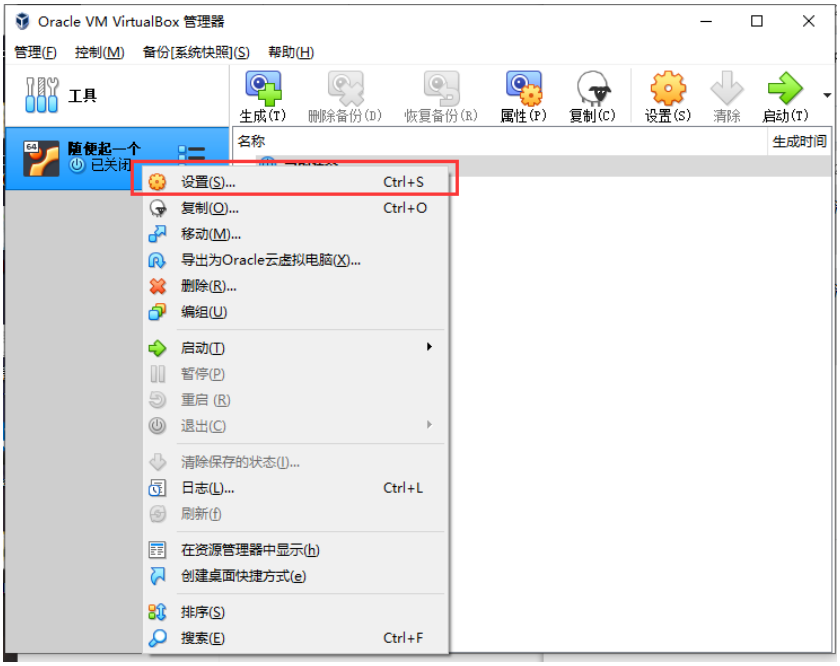
- 虚拟硬盘文件类型：默认即可。
- 动态分配/固定大小：默认即可。
- 最后点击创建。



至此，虚拟机已经创建完毕。点击“启动”即可进入虚拟机。

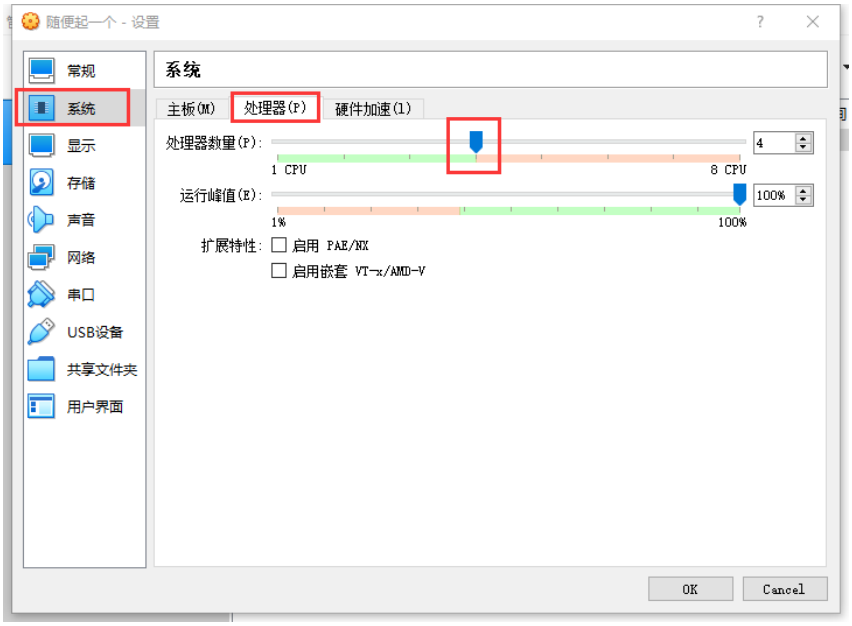
1.2.3 设置CPU数量

选中新建的虚拟机，右键-设置，打开设置界面。



在系统-处理器中将处理器数量调至合适的数量。该选项请根据自己电脑的处理器核数自行调整。

> 为虚拟机分配更多的CPU内核数量有助于提高虚拟机的性能。注意，给虚拟机分配的内核不是被虚拟机独占的。就算为虚拟机分配宿主机相同的内核数量，也毫无问题。



1.2.4 启动、挂载启动盘

然而，此时的虚拟机只是个空壳——硬盘是空的，里面什么都没有。所以我们需要安装操作系统。

启动虚拟机。虚拟机程序发现磁盘是空的，会自动提示我们挂载启动盘。选择我们下载的Ubuntu安装镜像文件作为启动盘。点击“启动”即可进入安装程序。

1.2.5 安装Ubuntu

1. 等一会之后会进入安装界面。你可以在左边把安装界面切成中文，然后点“安装Ubuntu”。

   > 如果在安装时发现“继续”、“后退”、“退出”等按钮在屏幕外，请先按 `Alt+F7` ，然后松开键盘，再移动鼠标以拖动窗口。点击鼠标会使窗口拖动停止。

2. 键盘布局选择Chinese即可。
3. 这里不建议选“安装Ubuntu时下载更新”。因为国内默认的下载源速度较慢，换源之后速度才快。

4. 因为虚拟机的磁盘本来就是空的，所以安装类型选择“清除整个磁盘并安装Ubuntu”。然后点现在安装-继续。

   > 警告：在安装双系统时，不要选这个，否则后果自负。

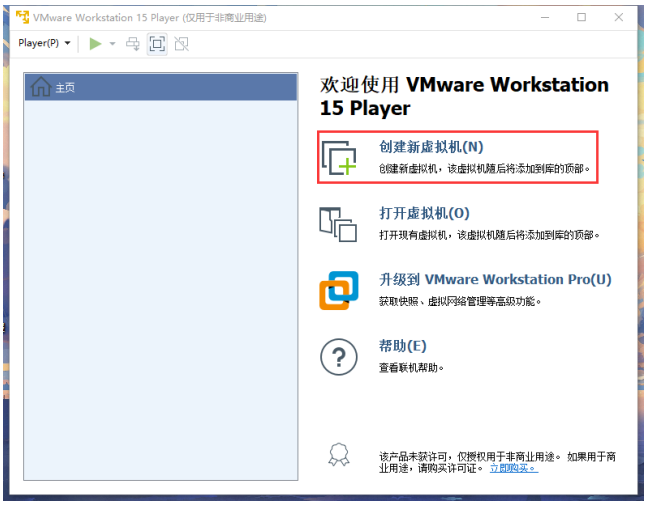
5. 时区位置默认上海即可。
6. 随便编一个姓名、计算机名、用户名，然后设置密码。

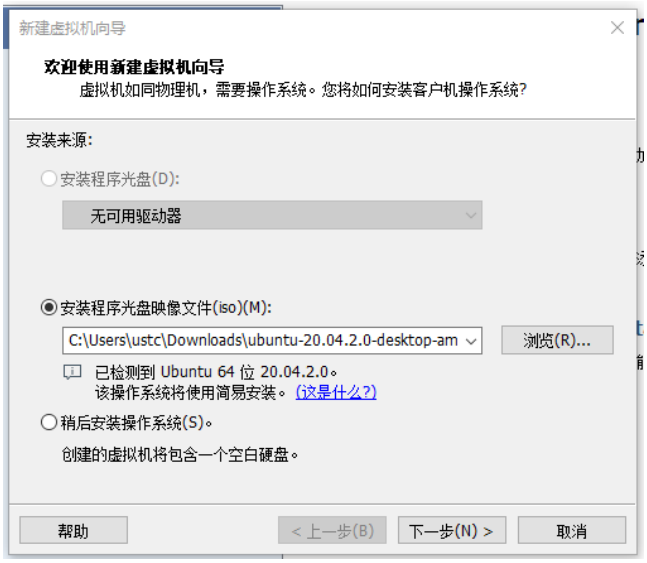
   > 警告：请一定要记住密码。否则会进不去系统。

7. 等安装完即可。安装完成之后系统会提示重启。如果重启之后系统提示需要移除安装光盘，对于VirtualBox，在上方控制-设置-存储里检查“控制器：IDE”里有没有安装镜像即可。如果有，就在右侧“分配光驱”里移除虚拟盘；如果是“没有盘片”，就直接在虚拟机中回车重启。

1.3 创建、安装虚拟机 (VMware)

1. 直接使用下载的Ubuntu镜像文件进行简易安装。





2. 设置计算机名、用户名、密码。

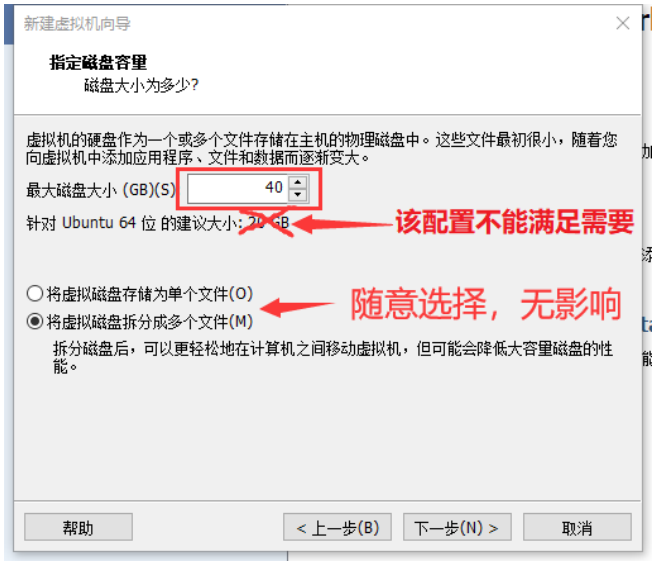
> 警告：请一定要记住密码。否则会进不去系统。

3. 设置虚拟机名称和文件存放位置。

> 考虑到虚拟磁盘大小动辄几十GB，建议将其放在空间有富余的磁盘分区上。

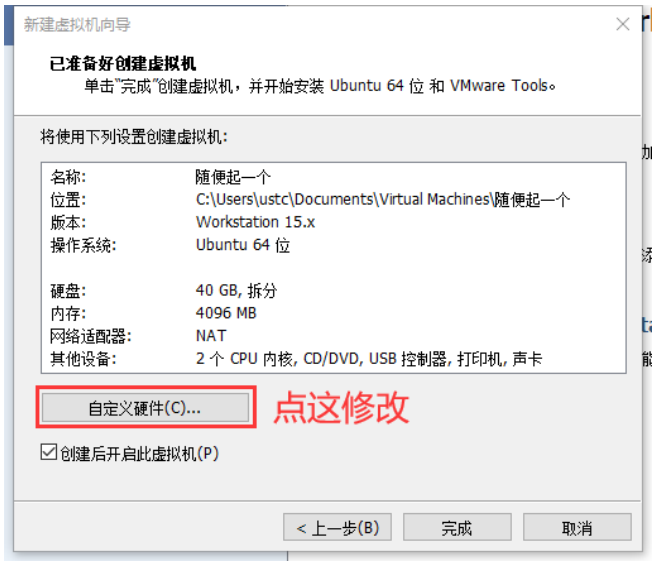
4. 最大磁盘大小：建议30~40G。我们的实验会占用较多的磁盘空间。20GB **非常紧张**。你可以随意选择是否拆分磁盘的选项。如果你有很多不常用的文件占用大量磁盘空间，可以考虑将其转移到**睿客云盘**上保存。

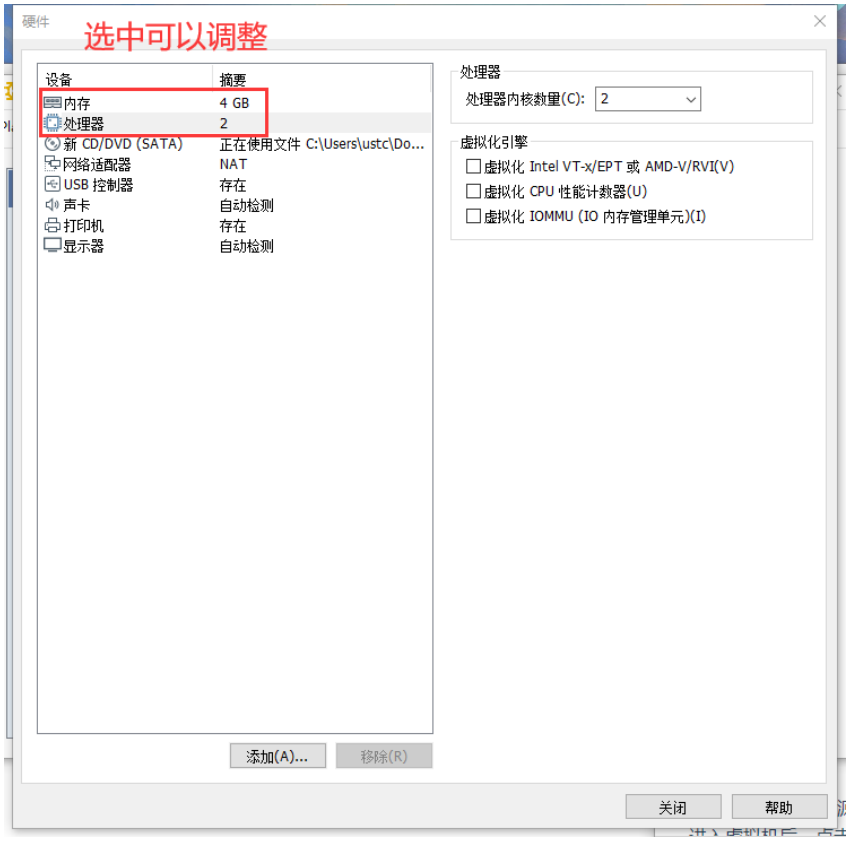
> 警告：如果磁盘空间不够，Linux启动会黑屏进不去图形界面，需要在命令模式下删除一些文件后重启才能进入图形界面。一些虚拟机具备“扩展磁盘容量”的功能，但是根据实际测试，发现很多时候反而会让虚拟机直接黑屏。



5. 可以在“自定义硬件”内自己设置内存、处理器核数等设置。

> 请至少分配2GB以上的内存给虚拟机。同时建议分配至少1/4主机内存给虚拟机。
>
> 为虚拟机分配更多的CPU内核数量有助于提高虚拟机的性能。注意，给虚拟机分配的内核不是被虚拟机独占的。就算为虚拟机分配宿主机相同的内核数量，也毫无问题。





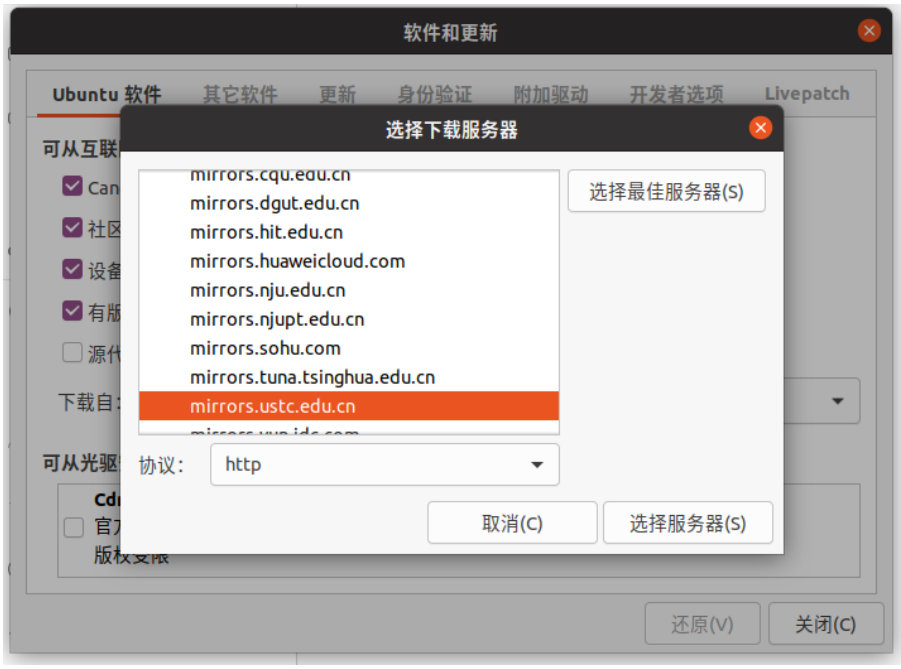
6. 点击“完成”即可自动安装Ubuntu系统。之后就不需要操作了，直接等安装完毕后虚拟机重启就可以愉快地使用Ubuntu了。

1.4 其他必要设置

1.4.1 换源

Ubuntu自带的软件源较慢，这会导致我们安装软件包时花更多的时间下载。所以要更换软件源为科大镜像。进入虚拟机后，点击左下角的进入应用菜单，找到并进入“软件更新器”。进入之后它会检查更新，最后会跳出一个“是否向安装更新”的提示。**不要安装**，并点击“设置”。

更改“Ubuntu”软件选项卡的“下载自”为“其他站点”，在弹出的“选择下载服务器”窗口中选择“中国-mirrors.ustc.edu.cn”。输入密码即可完成修改。



设置之后，如果提示更新软件包缓存，请选择更新，并等待更新结束再安装其他软件包/语言包。如果提示更新系统，也可以放心地选择更新而不必担心用时过长。

1.4.2 设置自动调整分辨率、文件拖放

- 对于VirtualBox，若要设置自动调整分辨率，请安装增强功能。可以参考此链接：https://ywnz.com/linuxjc/2410.html
- 对于VirtualBox，若要实现文件拖放，请安装扩展包。到官网下载页面(https://www.virtualbox.org/wiki/Downloads)找“VirtualBox 6.1.32 Oracle VM VirtualBox Extension Pack”，下载后双击安装即可。然后在上面的“设备”菜单中设置拖放和剪切板为“双向”。

> 虽然不支持将文件拖放到桌面，但你可以尝试打开Ubuntu的文件管理器，并将文件拖放至他处。

- VMware无需执行此步骤。因为VMware会自动安装VMware tools，但是如果发现调整不了虚拟机分辨率、无法共享粘贴板等情况，是自动安装失败（比如网络问题），需要手动安装。请参考此链接：https://blog.csdn.net/qq_64092369/article/details/123050107

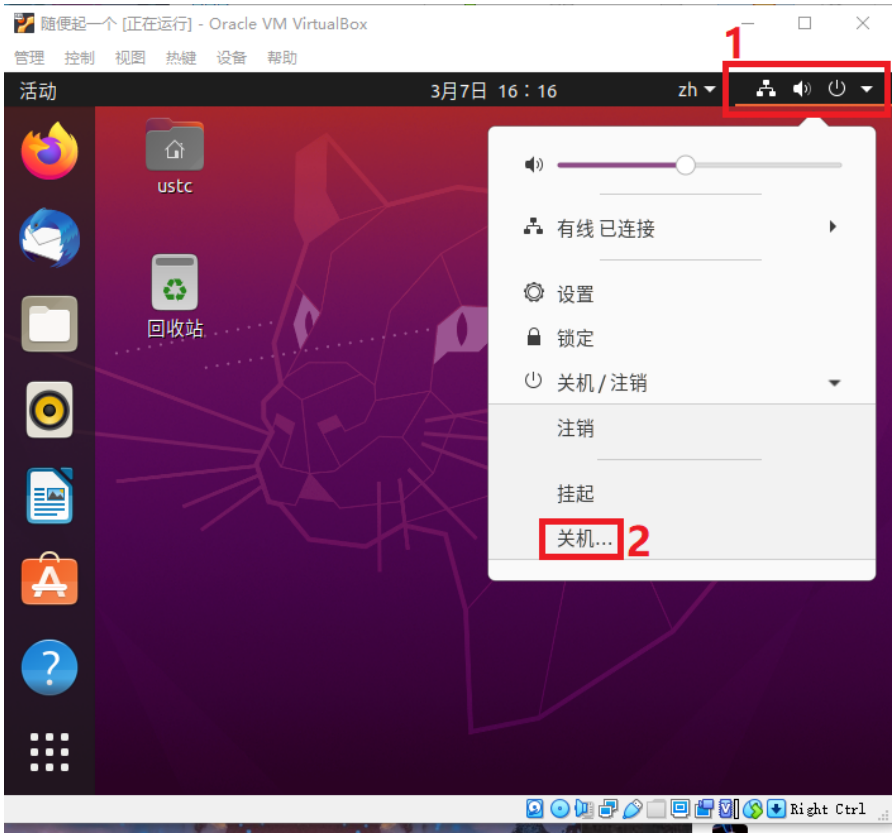
> VirtualBox/VMware的文件拖放经常会出问题，目前并没有通用的解决方案，因此建议用U盘、共享文件夹、睿客云盘之类实现文件中转。

1.4.3 修改语言 (VMware 简易安装)

- 请参考此链接：https://blog.csdn.net/langshi_2011/article/details/78993781
- 请尤其注意链接的第六步。

1.4.4 如何关机

如何关闭Ubuntu：如下图所示，点屏幕右上角-关机。



直接点虚拟机右上角的叉也可以关机。

- “快速休眠”是保存虚拟机状态，下次启动虚拟机时可以直接恢复上次状态；（建议使用）
- “正常关闭”相当于按电脑的电源键关机；
- “强制退出”相当于拔掉电脑电源进行硬关机。

第二部分：初探Linux

> 考虑到很多同学在本次实验之前没有使用过Linux系统，因此今年增加此部分来速成Linux。

2.1 Ubuntu GUI的使用

打开虚拟机进入到Ubuntu之后，即可看到Ubuntu的GUI界面。默认左侧是Dock（类似Windows的任务栏），里面有若干内置软件。左下角是菜单（类似Windows的开始菜单）。考虑到部分同学首次接触Ubuntu，因此建议各位依次点击所有的软件、按钮，以进一步了解Ubuntu并熟悉其中的软件。

屏幕最右上角有几个图标，可以调整音量、网络设置、语言、输入法等，还可以关机。在菜单里找到“设置”，里面可以调整系统设置，如分辨率、壁纸等。Dock里有一个文件夹形状的图标，它是文件管理器，可以像windows一样图形化地浏览文件。

> 提示：如果在执行某个操作时报错文件/文件夹不存在，可以在UI界面内手动复制粘贴文件到目标位置（文件所有者为root用户的除外）。

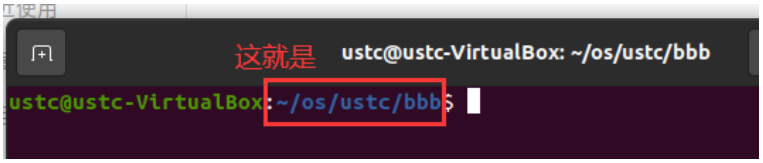
2.2 终端（命令行）的使用

2.2.1 打开终端

在桌面/文件管理器的空白处右键即可出现“在终端打开”按钮，点击此处即可呼出终端进而在终端中执行相关的Linux指令。注意：终端也是有工作位置的。简单而言，在哪个目录下打开终端，命令就会在哪个目录下执行。终端当前目录一般称为“**工作目录**”。

> 举例：`ls`指令可以显示工作目录下的文件。在不同的目录下运行此指令的结果显然是不一样的。

终端会显示工作目录，如图所示：



2.2.2 目录与路径

Linux与Windows不同，Windows一般会分有多个逻辑磁盘，每个逻辑磁盘各有一棵目录树，但Linux只有一个目录树，磁盘可以作为一棵子树**挂载**到目录树的某个节点。在Linux操作系统中，整个目录树的根节点被称为**根目录**。每个用户拥有一个**主目录**，或称**家目录**，类似windows的`C:\Users\用户名`。从根目录看，除root用户之外，每个用户的家目录为`/home/用户名`。

Linux终端里使用的路径分为**绝对路径**和**相对路径**两种。绝对路径指从根目录算起的路径，相对路径指从工作目录算起的路径。其中，以根目录算起的绝对路径以`/`开头，以家目录算起的绝对路径以`~`开头，相对路径不需要`/`或`~`开头。

> 举例：一个名为ustc的用户在他的家目录下创建了一个名为os的目录，在os目录下面又创建了一个名为lab1的目录，则该lab1目录可以表示为：
>
> - `/home/ustc/os/lab1`
> - `~/os/lab1`
> - 如果工作目录在家目录：`os/lab1`
> - 如果工作目录在os目录：`lab1`

一些特殊的目录：

- `.` 代表该目录自身。例：`cd .` 代表原地跳转（`cd` 是切换目录的指令）；
- `..` 代表该目录的父目录。特别地，根目录的父目录也是自身。
- 在Linux里，文件名以 `.` 开头的文件是隐藏文件（或目录），如何显示它们请参考2.3.3.
    - `.` 和 `..` 都是隐藏的目录。

> 举例：以下几个路径是等价的：
>
> - `/home/ustc/os/lab1`
> - `/home/ustc/os/././././/./lab1`
> - `/home/ustc/os/../os/lab1`

2.2.3 运行指令

在终端中输入可执行文件的路径即可。终端所在路径是程序运行时的路径。需要注意的是，如果运行的二进制文件就在工作目录下，需要在文件名前加上 `./` 。

> 提示：与windows可执行文件扩展名为.exe不同，Linux中，可执行文件一般没有扩展名。例外：gcc在不指定输出文件名的情况下，编译出的可执行文件会带有.out的扩展名。但你也不需要管这个.out，直接运行也是一样的。

> 例：有一个可执行文件，其路径为 `~/os/lab1/testprog` 。它在运行时会读取一个相对路径为 `a.txt` 的文件。
>
> - 在家目录下运行：需要执行 `os/lab1/testprog` （或 `~/os/lab1/testprog` 等绝对路径），程序读取的 `a.txt` 在家目录下；
> - 在 `~/os/lab1` 下运行：需要执行 `./testprog` (当然你用绝对路径也无所谓)，程序读取的 `a.txt` 在 `~/os/lab1` 目录下。

相关问题：为什么在运行 `sudo` 、 `man` 等命令时，只需要输入指令名而不需要输入这些指令对应的二进制文件所在的路径?

一种情况是，这是因为这些指令所在的路径（一般是 `/usr/bin` ）被加入到了该用户的**环境变量**中。当终端读取到一个不带路径的命令之后，系统只会在环境变量中搜索，从而方便用户使用。当然，默认被加入环境变量里的路径一般只有一些系统路径，除非自行设置，家目录下面没有目录默认在环境变量中。如感兴趣，修改环境变量的方法可自行搜索了解。

另一种情况是，部分指令是Shell内建指令（如 `cd` ），它们的意义直接由Shell解释，没有对应的二进制文件。此情况可能会在下一实验中详细阐述。

2.2.4 指令及其参数

无论是Linux还是Windows，一条完整的命令都由命令及其参数构成。你们可以通过 `man 指令名` 来自行了解指令语法。一般来说，指令语法里带有'[ ]'的是可选参数，其他是必选参数。不同的参数的顺序一般是可以互换的。下面以 `gcc` （一种编译器）为例，介绍实验文档描述指令的方式，以及如何按需构造一条指令。

本次要用到的 `gcc` 指令的一部分语法是： `gcc [-static] [-o outfile] infile` 。下面是各参数介绍：

参数	含义
-static	静态编译选项（此处参数仅为示例，参数详细含义请自行上网搜索）。
-o outfile	指定输出的可执行文件的文件名为outfile。如果不指定，会输出为a.out。
infile	要编译的gcc文件名。注意绝对路径/相对路径的问题。

- 如果我们编译test.c，不指定输出文件名，命令就只是 `gcc test.c` ；（这种情况下，gcc会自动命名输出文件为a.out）
- 如果我们编译test.c，输出二进制文件名为test，命令就是 `gcc -o test test.c` ；
    - 一般来说参数的顺序是无所谓的。所以使用 `gcc test.c -o test` 也一样能编译。

- 如果我们编译test.c，输出二进制文件名为test，且要使用静态编译，那么构造出的编译指令就是 `gcc -static -o test test.c` 。

2.2.5 终端使用小技巧

- 按键盘的 `↑` `↓` 键可以切换到之前输入过的指令；
- 按键盘的 `Tab` 键可以自动补全。如果按一下Tab之后没反应，说明候选项太多。再按一下Tab可以显示所有候选项。
- 在shell里， `Ctrl+C` 是终止不是复制。复制的快捷键是 `Ctrl+Insert` 或 `Ctrl+Shift+C` ，粘贴的快捷键是 `Shift+Insert` 或 `Ctrl+Shift+V` 。

2.3 Linux常用指令

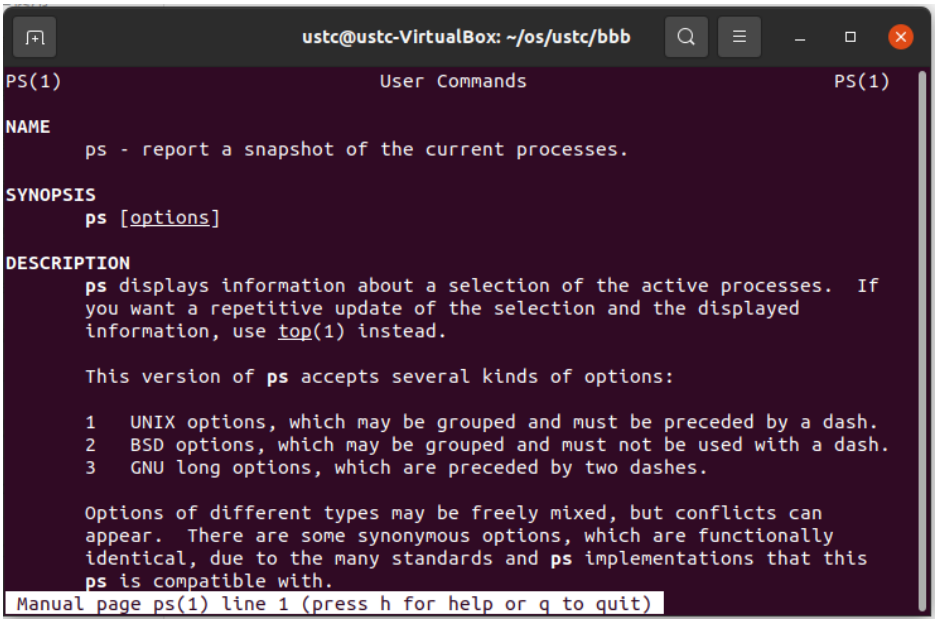
> 注：一些指令的使用方法详见提供的链接。本部分涉及测验考察，测验方式见文档4.2。

2.3.1 man

英文缩写：manual

如果你不知道一条命令的含义，使用 `man xxx` 可以显示该命令的使用手册。

举例： `man ps` 可以显示 `ps` 指令的使用方法。指令输出如下图。按q退出手册。



2.3.2 sudo

TL;DR： `sudo` **= “以管理员模式运行”。** A joke

Linux采用**用户组**的概念实现访问控制，其中，只有root用户组才具备管理系统的权限。在 `sudo` 出现之前，一般用户管理linux系统的方式是，先用 `su` 指令切换到root用户，然后在root用户下进行操作。但是使用 `su` 的缺点之一在于必须要先告知root用户的密码，且这种控制方式不够精细。

为了方便操作，并更加精确地控制权限， `sudo` 用来将root用户的部分权限让渡给普通用户。普通用户只需要输入自己的密码，确认“我是我自己”，就能执行自己拥有的那部分管理员权限。特别地，在Ubuntu下，普通用户默认可以通过 `sudo` 取得root用户的所有权限。若想精确地控制每个用户能用 `sudo` 干什么，可以参阅 `visudo` 。

如果在输入某个指令后，系统提示权限不够(Permission Denied)，那么在指令前加上 `sudo` 一般都能解决问题。

> 注意1：root用户具备很高的权限。一些需要管理员权限才能执行的指令（如，删除整个磁盘上的内容）会破坏系统。所以在使用 `sudo` 时，请务必确认输入的指令没问题。

> 注意2：为了防止旁窥者获知密码长度，linux下输入密码不会在屏幕上给出诸如***的回显。输完密码敲回车就行了。

常见使用方法：`sudo command`

例：以普通用户运行`apt install vim`，会被告知没有权限，因为只有root用户（管理员）才有资格安装软件包。正确的用法是`sudo apt install vim`。

2.3.3 ls

英文全拼：list files

显示特定目录中的文件列表。常见的一部分语法是：`ls [-a] [name]`。参数含义详见**Linux ls命令**。

2.3.4 cd

英文全拼：change directory

切换工作目录。常见的一部分语法是：`cd [name]`。参数含义详见**Linux cd命令**。特别地，常见使用`cd -`来返回上次到达的目录。

2.3.5 pwd

英文全拼：print work directory

输出工作目录的绝对路径。常见的使用方法是：`pwd`。

2.3.6 rm

英文全拼：remove

删除一个文件或目录。常见的一部分语法是：`rm [-rf] name`。参数含义详见**Linux rm命令**。

2.3.7 mv

英文全拼：move

移动一个文件或目录，或重命名文件。常见的一部分语法是：`mv source dest`。参数含义详见**Linux mv命令**。

2.3.8 cp

英文全拼：copy

复制一个文件或目录，或重命名文件。常见的一部分语法是：`cp [-r] source dest`。参数含义详见**Linux cp命令**。

2.3.9 mkdir

英文全拼：make directory

创建目录。常见的一部分语法是：`mkdir [-p] dirName`。

参数含义详见**Linux mkdir命令**。

2.3.10 cat

英文全拼：concatenate

一般用于将文件内容输出到屏幕。常见的使用方法是：`cat fileName`。

2.3.11 kill

用于将指定的信息发送给程序，通常使用此指令来结束进程。强制结束进程的信号编号是9，所以强制结束进程的使用方法是：`kill -9 [进程编号]`。

其他参数的使用方法可以参考 **Linux kill命令**。

2.3.12 ps

英文全拼：process status

用于显示进程状态（任务管理器）。常见的使用方法：

- `ps`：显示**当前用户**在当前终端控制下的进程。考虑到用户通常想看不止当前用户和当前终端下的进程，所以不加参数的用法并不常用。
- `ps aux`：展示所有用户所有进程的详细信息。注意，a前面带一个横线是严格意义上不正确的使用方法。
- `ps -ef`：也是展示所有用户所有进程的详细信息。就输出结果而言和 `ps aux` 无甚差别。

2.3.13 wget

wget是一个在命令行下下载文件的工具。常见的使用方法是：`wget [-O FILE] URL`。

例如，我们要下载 **https://git-scm.com/images/logo@2x.png** 到本地。

1. 直接下载：`wget https://git-scm.com/images/logo@2x.png`，文件名是logo@2x.png。
2. 直接下载并重命名：`wget -O git.png https://git-scm.com/images/logo@2x.png`，文件名是git.png。注：-O中的O表示字母O而不是数字0。

注意，如果发现有同名文件，1所示方法会在文件名后面加上.1的后缀进行区分，而2所示方法会直接覆盖。

2.3.14 tar

用于压缩、解压压缩包。常见使用方法：

- 把某名为source的文件或目录压缩成名为out.tar.gz的gzip格式压缩文件：`tar zcvf out.tar.gz source`
- 解压缩某名为abc.tar.gz的gzip格式压缩文件：`tar zxvf abc.tar.gz`

其他使用方法详见 **Linux tar命令**

2.3.15 包管理器（apt等）

在Linux下，如何安装软件包？每个Linux发行版都会自带一个**包管理器**，类似一个“软件管家”，专门下载免费软件。

不同Linux发行版附带的包管理器是不一样的。如，Debian用apt（Ubuntu是基于Debian的，所以也用apt），ArchLinux用Pacman，CentOS用yum，等等。

apt的常见用法：

- 安装xx包，yy包和zz包：`sudo apt install xx yy zz`
- 删除xx包，yy包和zz包：`sudo apt remove xx yy zz`
- 更新已安装的软件包：`sudo apt upgrade`
- 从软件源处检查系统中的软件包是否有更新：`sudo apt update`

其他使用方法参见 **Linux apt命令**。

> 其他提示：
>
> - 如果报错“`无法获得锁 /var/lib/dpkg/lock.....`”，这是因为系统在同一时刻只能运行一个apt，请耐心等另一边安装/更新完。Ubuntu的软件包管理器也是一个apt。若你确信没有别的apt在运行，可能是因为没安装完包就关掉了终端或apt。请重启Linux或参考 **此链接**。
> - 如果报错“`下列软件包有未满足的依赖关系.....`”或 `没有可用的软件包.....，但是它被其他软件包引用了....`，可能是因为刚换了源，没等包刷新完就关闭窗口，请手动 `sudo apt-get update`。

2.3.16 文字编辑器（vim、gedit等）

Linux系统中，常见的使用命令行的文字编辑器是vim。系统一般并**不**自带vim，需要使用包管理器安装。由于vim的使用方法与我们习惯的的GUI编辑方式有很大差异，所以在这里不详细介绍它的用法。若想了解请参考 https://www.runoob.com/linux/linux-vim.html 。

gedit是Ubuntu使用的Gnome桌面环境**自带**的一款文本编辑器，其使用方法与Windows的记事本(Notepad)大同小异。在这里也不多介绍。在终端里输入 `gedit` 回车即可启动gedit。编辑特定的文件的使用方法是 `gedit 文件名` ，若输入的文件名不存在，将自动创建新文件。在Ubuntu的图形化界面中直接双击文本文件也可以编辑文件。**但需要注意的是，在编辑一些需要root权限的文件时，直接双击文件不能编辑，只能在终端里 `sudo gedit 文件名` 。**

> 在命令行下启动gedit会在终端里报warning，忽略即可。 **参考链接**

如果想使用一些更高级的编辑器，可以考虑安装vscode。安装方法请自行到网上搜索。

如果想安装高级IDE的话（不推荐这么做），只能用CLion等。Visual Studio不支持Linux。

> 在后续的实验中，如果某步骤写着 `vim xxx` ，说明这是让你编辑某文件。编辑文件的方式不仅局限于vim，用gedit等也可。

2.3.17 编译指令（gcc、g++、make等）

我们在编译代码时，需要使用编译器。 `gcc` 和 `g++` 是常见的编译命令。其中:

- `gcc` 会把 `.c` 文件当作C语言进行编译，把 `.cpp` 文件当作C++语言编译。
- `g++` 会把 `.c` 和 `.cpp` 文件都当作C++语言编译。

考虑到复杂工程需要编译的文件数量会很多，此时每次都手输编译命令较为繁琐，为此GNU提供了一个make工具，可以按照一个编写好的Makefile文件来完成编译任务。本课程实验应该不会涉及让学生自行从头编写一个Makefile，但涉及使用 `make` 命令。make的工作原理、Makefile的编写方法可以参考 **这个链接** 。

> 注意：
>
> 1. 这里的“C语言”是C90标准的C语言，**不能使用STL、类、引用、for内定义变量等C++特性。**
> 2. Linux内核是使用上面所述的C语言编写的，而不是C++。
> 3. 在编写代码时，请注意代码文件的扩展名命名，和编译指令的选择。
> 4. `gcc` 和 `g++` 默认是不安装的。如果你想使用，请先使用包管理器安装 `build-essential` 包，里面包括 `gcc` 、 `g++` 等常见编译器，和make工具等。

简单地使用 `gcc` / `g++` 编译的语法是： `gcc [-o outfile] infile` 。下面是指令的各个参数介绍。如需了解更详尽的使用方法，可参考gcc官方手册： http://www.gnu.org/software/gcc/ 。

参数	含义
-o outfile	指定输出的可执行文件的文件名为outfile。如果不指定，会输出为a.out。
infile	要编译的gcc文件名。注意绝对路径/相对路径的问题。

使用 `make` 的方法是：在工程目录下直接运行 `make` 即可。一些Makefile会提供多种编译选项，如删除编译好的二进制文件(`make clean`)、编译不同的文件等。这时候需要根据Makefile来确定不同的编译选项。在后面的实验中，如有此方面的需要，我们将给出使用方法。

2.4 Shell 脚本

考虑到往届在检查实验时，大多数时间都浪费在了敲指令上，今年尝试教学生如何编写Shell脚本。

2.4.1 Shell脚本的编写

Shell脚本类似于Windows的.bat批处理文件。一个最简单的Shell脚本长这样：

```
#!/bin/bash
第一条命令
第二条命令
第三条命令，以此类推
```

其中，第一行的`#!/bin/bash`是一个约定的标记，它告诉系统这个脚本需要什么解释器来执行，即使用哪一种Shell。Ubuntu下默认的shell是bash。脚本一般命名为`xxx.sh`。

2.4.2 脚本的运行

运行Shell脚本有两种方式。

1. 通过执行`sh xxx.sh`来直接调用解释器运行脚本。其中，sh就是我们的Shell。这种方式运行的脚本，不需要在第一行指定解释器信息。
2. 将该脚本视为可执行程序。首先，保存脚本之后，要通过`chmod +x xxx.sh`给脚本赋予执行权限，然后直接`./xxx.sh`运行脚本。

> 注意：脚本的执行和在终端下执行这些语句是完全一致的，依然需要注意绝对路径和相对路径的问题。

举例：首先编译abc.cpp并输出一个名为aabbcc的二进制可执行文件，然后执行aabbcc，最后删掉aabbcc，上述操作可以使用如下脚本来实现：

```
#!/bin/bash
gcc -o aabbcc abc.cpp
./aabbcc
rm aabbcc
```

之后的实验会介绍更多Shell脚本的语法。

2.5 参考资料

- Linux命令大全： https://www.runoob.com/linux/linux-command-manual.html
- Shell教程： https://www.runoob.com/linux/linux-shell.html

第三部分：编译、调试Linux内核

3.1 先导知识

3.1.1 系统内核启动过程

Linux kernel在自身初始化完成之后，需要能够找到并运行第一个用户程序（此程序通常叫做“init”程序）。用户程序存在于文件系统之中，因此，内核必须找到并挂载一个文件系统才可以成功完成系统的引导过程。

在grub中提供了一个选项“root=”用来指定第一个文件系统，但随着硬件的发展，很多情况下这个文件系统也许是存放在USB设备，SCSI设备等等多种多样的设备之上，如果需要正确引导，USB或者SCSI驱动模块首先需要运行起来，可是不巧的是，这些驱动程序也是存放在文件系统里，因此会形成一个悖论。

为解决此问题，Linux kernel提出了一个RAM disk的解决方案，把一些启动所必须的用户程序和驱动模块放在RAM disk中，这个RAM disk看上去和普通的disk一样，有文件系统，有cache，内核启动时，首先把RAM disk挂载起来，等到init程序和一些必要模块运行起来之后，再切换到真正的文件系统之中。

但是，这种RAM disk的方案（下称initrd）虽然解决了问题但并不完美。 比如，disk有cache机制，对于RAM disk来说，这个cache机制就显得很多余且浪费空间；disk需要文件系统，那文件系统（如ext2等）必须被编译进kernel而不能作为模块来使用。

Linux 2.6 kernel提出了一种新的实现机制，即initramfs。顾名思义，initramfs只是一种RAM filesystem而不是disk。initramfs实际是一个cpio归档，启动所需的用户程序和驱动模块被归档成一个文件。因此，不需要cache，也不需要文件系统。

3.1.2 什么是initramfs

initramfs 是一种以 cpio 格式压缩后的 rootfs 文件系统，它通常和 Linux 内核文件一起被打包成boot.img 作为启动镜像。

BootLoader 加载 boot.img，并启动内核之后，内核接着就对 cpio 格式的 initramfs 进行解压，并将解压后得到的 rootfs 加载进内存，最后内核会检查 rootfs 中是否存在 init 可执行文件（该init 文件本质上是一个执行的 shell 脚本），如果存在，就开始执行 init 程序并创建 Linux 系统用户空间 PID 为 1 的进程，然后将磁盘中存放根目录内容的分区真正地挂载到 / 根目录上，最后通过 `exec chroot . /sbin/init` 命令来将 rootfs 中的根目录切换到挂载了实际磁盘分区文件系统中，并执行 /sbin/init 程序来启动系统中的其他进程和服务。

基于ramfs开发的initramfs取代了initrd。

3.1.3 什么是initrd

initrd代指内核启动过程中的一个阶段：临时挂载文件系统，加载硬盘的基础驱动，进而过渡到最终的根文件系统。

initrd也是早期基于ramdisk生成的临时根文件系统的名称。现阶段虽然基于initramfs，但是临时根文件系统也依然存在某些发行版称其为initrd。例如，CentOS 临时根文件系统命名为 initramfs-`uname -r`.img，Ubuntu 临时根文件系统命名为 initrd-`uname -r`.img（uname -r是系统内核版本）。

3.1.4 QEMU

QEMU是一个开源虚拟机。可以在里面运行Linux甚至Windows等操作系统。

本次实验需要在虚拟机中安装QEMU，并使用该QEMU来运行编译好的Linux内核。这么做的原因如下：

- 在Windows下编译LInux源码十分麻烦，且QEMU在Windows下速度很慢；
- 之后的实验会涉及修改Linux源码。如果直接修改Ubuntu的内核，改完代码重新编译之后需要重启才能完成更改，但带GUI的Ubuntu系统启动速度较慢。另外，操作失误可能导致Ubuntu系统损坏无法运行。

3.2 下载、安装

> 为防止表述混乱，本文档指定 `~/oslab` 为本次实验使用的的目录。同学们也可以根据需要在其他目录中完成本次实验。如果在实验中遇到“找不到 `~/oslab` ”的报错，请先创建相关目录。

3.2.1 下载 Linux 内核源码

- 下载Linux内核源码，保存到 `~/oslab/` 目录（提示：使用 `wget` ）。源码地址：
  - https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.9.263.tar.xz
  - 备用链接1： https://od.srpr.cc/acgg0/linux-4.9.263.tar.xz
  - 备用链接2： http://home.ustc.edu.cn/~maohaoyu/linux-4.9.263.tar.xz
- 解压Linux内核源码（提示：使用 `tar` 。xz格式的解压参数是 `Jxvf` ，参数区分大小写）。

3.2.2 下载 busybox

- 下载busybox源码，保存到 `~/oslab/` 目录（提示：使用 `wget` ）。源码地址：
  - https://busybox.net/downloads/busybox-1.32.1.tar.bz2
  - 备用链接1： https://od.srpr.cc/acgg0/busybox-1.32.1.tar.bz2
  - 备用链接2： http://home.ustc.edu.cn/~maohaoyu/busybox-1.32.1.tar.bz2
- 解压busybox源码（提示：使用 `tar` 。bz2格式的解压参数是 `jxvf` ）。

如果3.2.1和3.2.2执行正确的话，你应该能在 `~/oslab/` 目录下看到两个子目录，一个叫 `linux-4.9.263` ，是**Linux内核源码路径**。另一个叫 `busybox-1.32.1` ，是**busybox源码路径**。

3.2.3 安装 qemu 和 Linux 编译依赖库

使用包管理器安装以下包：

- qemu-system-x86 (Ubuntu 20.04以上) 或 qemu (Ubuntu 其他版本)
- git
- build-essential （里面包含了make/gcc/g++等，省去了单独安装的麻烦）
- libelf-dev
- xz-utils
- libssl-dev
- bc
- libncurses5-dev
- libncursesw5-dev

3.2.4 编译 Linux 源码

1. 精简配置：将我们提供的 `.config` 文件下载到Linux内核源码路径（提示：该路径请看3.2.2最后一段的描述，使用 `wget` 下载）。 `.config` 文件地址：
   - http://home.ustc.edu.cn/~maohaoyu/.config
   - 备用链接： https://git.lug.ustc.edu.cn/gloomy/ustc_os/-/raw/master/term2021/lab1/.config

   > 提示：以 `.` 开头的文件是隐藏文件。如果你下载后找不到它们，请参考2.2.2和2.3.3。

2. 内核配置：在Linux内核源码路径下运行 `make menuconfig` 以进行编译设置。本次实验直接选择Save，然后Exit。
3. 编译：在Linux内核源码路径下运行 `make -j $((`nproc`-1))` 以编译内核。作为参考，助教台式机CPU为i5-7500(4核4线程)，使用VitrualBox虚拟机，编译用时5min左右。

   > 提示：

> 1. `nproc` 是个shell内置变量，代表CPU核心数。如果虚拟机分配的cpu数只有1（如Hyper-V默认只分配1核），则需先调整虚拟机分配的核心数。
> 2. 如果你的指令只有 `make -j` ，后面没有加上处理器核数，那么编译器会无限开多线程。因为Linux内核编译十分复杂，这会直接吃满你的系统资源，导致系统崩溃。
> 3. `nproc` 前的 ` 是反引号，位于键盘左上侧。**不**是enter键旁边那个。

若干其他问题及其解决方案：

> 问题1: 编译内核时遇到 make[1]: *** No rule to make target 'debian/canonical-certs.pem', needed by 'certs/x509_certificate_list'. Stop.
> 解决方案：用文本编辑器(vim 或 gedit)打开 PATH-TO-linux-4.9.263/.config文件, 找到并注释掉包含 CONFIG_SYSTEM_TRUSTED_KEY 和 CONFIG_MODULE_SIG_KEY 的两行即可.
> 解决方案链接： **https://unix.stackexchange.com/questions/293642/attempting-to-compile-kernel-yieldsa-certification-error**

> 问题2：make menuconfig时遇到以下错误：Your display is too small to run Menuconfig! It must be at least 19 lines by 80 columns.
> 解决方案：请阅读报错信息并自行解决该问题。

4. 如果编译成功，我们可以在Linux内核源码路径下的 `arch/x86_64/boot/` 下看到一个 `bzImage` 文件，这个文件就是内核镜像文件。

> 若怀疑编译/安装有问题，可以先在Linux内核源码路径下运行 `make clean` 之后从3.2.4.1开始。

3.2.5 编译busybox

1. 在busybox的源码路径下运行 `make menuconfig` 以进行编译设置。修改配置如下：(空格键勾选)

```
Settings ->
  Build Options
   [*] Build static binary（no share libs）
```

2. 编译：在busybox的源码路径下运行 `make -j $((`nproc`-1))` 以编译busybox。本部分的提示与3.2.4.3相同。
3. 安装：在busybox的源码路径下运行 `sudo make install` 。

> 若怀疑编译/安装有问题，可以先在busybox的源码路径下运行 `make clean` 之后从3.2.5.1开始。

3.2.6 制作根文件系统

1. 将工作目录切换为busybox源码目录下的 `_install` 目录。
2. **使用 `sudo` 创建**一个名为 `dev` 的文件夹（提示：参考2.3.9）。
3. 使用以下指令创建 `dev/ram` 和 `dev/console` 两个设备文件：
   - `sudo mknod dev/console c 5 1`
   - `sudo mknod dev/ram b 1 0`
4. **使用 `sudo` 创建**一个名为 `init` 的文件（提示：参考2.3.16）。**使用文本编辑器编辑文件**，文件内容如下：

> 特别提示：这一步**不是**让你在终端里执行这些命令。在复制粘贴时，请注意空格、回车是否正确地保留。
>
> 在命令行下启动gedit会在终端里报warning，忽略即可。 **参考链接**

```
#!/bin/sh
echo "INIT SCRIPT"
mkdir /proc
mkdir /sys
mount -t proc none /proc
mount -t sysfs none /sys
mkdir /tmp
mount -t tmpfs none /tmp
echo -e "\nThis boot took $(cut -d' ' -f1 /proc/uptime) seconds\n"
exec /bin/sh
```

5. 赋予 `init` 执行权限：`sudo chmod +x init`
6. 将x86-busybox下面的内容打包归档成cpio文件，以供Linux内核做initramfs启动执行:

   `find . -print0 | cpio --null -ov --format=newc | gzip -9 > 你们自己指定的cpio文件路径`

   > 注意1：该命令一定要在busybox的 _install 目录下执行。
   >
   > 注意2：每次修改 `_install` ，都要重新执行该命令。
   >
   > 注意3：请自行指定cpio文件名及其路径。示例：`~/oslab/initramfs-busybox-x64.cpio.gz`

3.2.7 运行qemu

我们本次实验需要用到的qemu指令格式：

`qemu-system-x86_64 [-s] [-S] [-kernel ImagePath] [-initrd InitPath] [--append Para] [-nographic]`

参数	含义
-s	在3.3.2介绍
-S	在3.3.2介绍
-kernel ImagePath	指定系统镜像的路径为ImagePath。在本次实验中，该路径为3.2.4.4所述的bzImage文件路径。
-initrd InitPath	指定initramfs(3.2.6生成的cpio文件)的路径为InitPath。在本次实验中，该路径为3.2.6.6所述的cpio文件路径。
--append Para	指定给内核启动赋的参数。
-nographic	设置无图形化界面启动。

在 `--append` 参数中，本次实验需要使用以下子参数：

参数	含义
nokaslr	关闭内核地址随机化，便于调试。
root=/dev/ram	指定启动的第一个文件系统为ramdisk。我们在3.2.6.3创建了 `/dev/ram` 这个设备文件。
init=/init	指定init进程为/init。我们在3.2.6.4创建了init。
console=ttyS0	**对于无图形化界面启动**（如WSL），需要使用本参数指定控制台输出位置。

这些参数之间以空格分隔。

总结(TL;DR)：

> 运行下述指令前，请注意先**理解指令含义**，**注意换行问题**。

> 如果报错找不到文件，请先检查：**是否把pdf的换行和不必要的空格复制进去了？前几步输出的东西是不是在期望的位置上？** 我们不会提示下述指令里哪些空格/换行符是应有的，哪些是不应有的。请自行对着上面的指令含义排查。
> 如果报错Failed to execute /init (error -8)，建议从3.2.5开始重做。若多次重做仍有问题，建议直接删掉解压好的busybox和Linux源码目录，从3.2.4重做。目前已知init文件配置错误、Linux内核编译出现问题都有可能导致本错误。

- 以图形界面，弹出窗口形式运行内核：

```
qemu-system-x86_64 -kernel ~/oslab/linux-4.9.263/arch/x86_64/boot/bzImage -initrd
~/oslab/initramfs-busybox-x64.cpio.gz --append "nokaslr root=/dev/ram init=/init"
```

- Ubuntu 20.04/20.10 环境下如果出现问题，可执行以下指令:

```
qemu-system-x86_64 -kernel ~/oslab/linux-4.9.263/arch/x86/boot/bzImage -initrd
~/oslab/initramfs-busybox-x64.cpio.gz --append "nokaslr root=/dev/ram init=/init"
```

- 如不希望qemu以图形界面启动，希望以无界面形式启动（如WSL），输出重定向到当前shell，使用以下命令:

```
qemu-system-x86_64 -kernel ~/oslab/linux-4.9.263/arch/x86_64/boot/bzImage -initrd
~/oslab/initramfs-busybox-x64.cpio.gz --append "nokaslr root=/dev/ram init=/init
console=ttyS0 " -nographic
```

其他常见问题：

1. 如何检查运行是否正确？

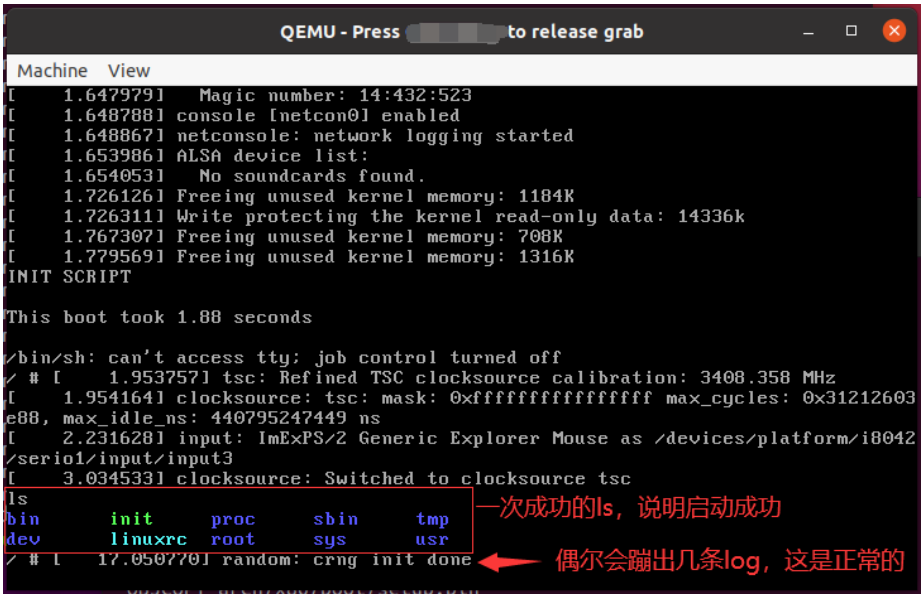
   弹出的黑色窗口就是qemu。因为系统内核在启动的时候会输出一些log，所以qemu界面里偶尔会蹦出一两条log是正常的， **只要这些log不是诸如"kernel panic"之类的报错** 即可。建议尝试输入一条命令，比如在弹出的窗口里面输入 `ls` 回车，如果能够显示相关的文件列表,即说明运行正确。

2. 鼠标不见了，该怎么办？

   请观察窗口上方的标题栏，"Press （请自行观察标题栏上的说明） to release grab"

3. 如何关闭qemu？

   对于图形化界面，直接点击右上角的叉就行了。对于非图形化界面，请先按下Ctrl+A，松开这两个键之后再按X。



3.3 使用 gdb调试内核

gdb是一款命令行下常用的调试工具，可以用来打断点、显示变量的内存地址，以及内存地址中的数据等。使用方法是 `gdb 可执行文件名`，即可在gdb下调试某二进制文件。

一般在使用gcc等编译器编译程序的时候，编译器不会把调试信息放进可执行文件里，进而导致gdb知道某段内存里有内容，但并不知道这些内容是变量a还是变量b。或者，gdb知道运行了若干机器指令，但不知道这些机器指令对应哪些C语言代码。所以，在使用gdb时需要在编译时加入 `-g` 选项，如：`gcc -g -o test test.c` 来将调试信息加入可执行文件。而Linux内核采取了另一种方式：它把符号表独立成了另一个文件，在调试的时候载入符号表文件就可以达到相同的效果。

- gdb里的常用命令

```
r/run                               # 开始执行程序
b/break <location>                  # 在location处添加断点，location可以是代码行数或函数名
b/break <location> if <condition>   # 在location处添加断点，仅当condition条件满足才中断运行
c/continue                          # 继续执行到下一个断点或程序结束
n/next                              # 运行下一行代码，如果遇到函数调用直接跳到调用结束
s/step                              # 运行下一行代码，如果遇到函数调用则进入函数内部逐行执行
ni/nexti                            # 类似next，运行下一行汇编代码（一行c代码可能对应多行汇编代码）
si/stepi                            # 类似step，运行下一行汇编代码
list                                # 显示当前行代码
p/print <expression>                # 查看表达式expression的值
q                                   # 退出gdb
```

3.3.1 安装 gdb

使用包管理器安装名为gdb的包即可。

3.3.2 启动 gdb server

使用3.2.7所述指令运行qemu。但需要加上 `-s` 和 `-S` 两个参数。这两个参数的含义如下：

参数	含义
-s	启动gdb server调试内核，server端口是1234。若不想使用1234端口，则可以使用 `-gdb tcp:xxxx` 来取代此选项。
-S	若使用本参数，在qemu刚运行时，CPU是停止的。你需要在gdb里面使用c来手动开始运行内核。

3.3.3 建立连接

另开一个终端，运行gdb，然后在gdb界面里运行如下命令：

```
target remote:1234   # 建立gdb与gdb server间的连接。这时候我们看到输出带有??，还报了一条warning
                     # 这是因为没有加载符号表，gdb不知道运行的是什么代码。
c                    # 手动开始运行内核。执行完这句后，你应该能发现旁边的qemu开始运行了。
q                    # 退出gdb。
```

3.3.4 重新编译Linux内核使其携带调试信息

刚才因为没有加载符号表，所以gdb不知道运行了什么代码。所以我们要重新编译Linux来使其携带调试信息。

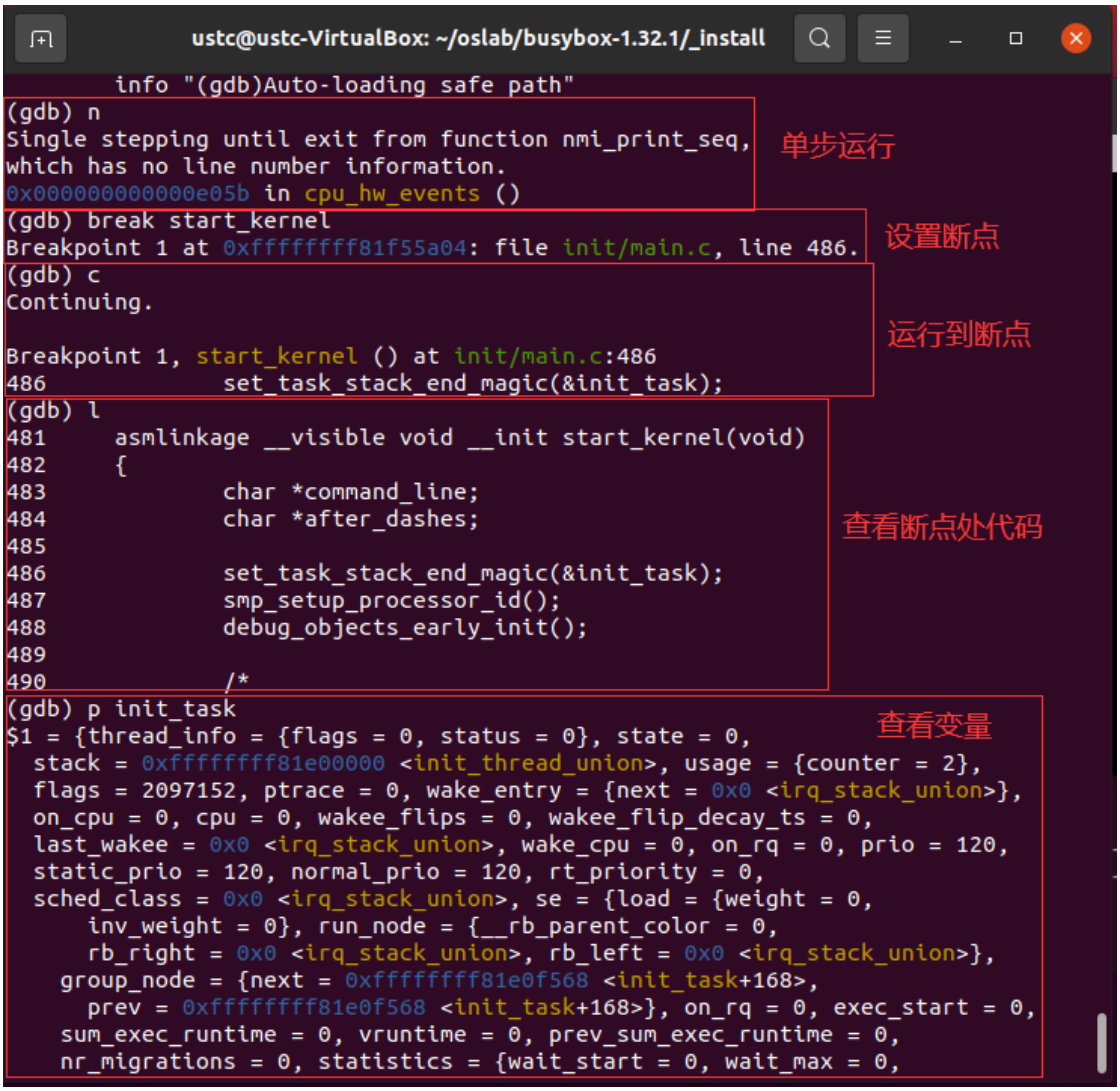
1. 进入Linux源码路径。
2. 执行下列语句(这条语句没有回车，注意文档显示时产生的额外换行问题)：
   `./scripts/config -e DEBUG_INFO -e GDB_SCRIPTS -d DEBUG_INFO_REDUCED -d DEBUG_INFO_SPLIT -d DEBUG_INFO_DWARF4`
3. 重新编译内核。`make -j $((`nproc`-1))`

作为参考，助教台式机CPU为i5-7500(4核4线程)，使用VitrualBox虚拟机，编译用时8.5min左右。

3.3.5 加载符号表、设置断点

1. 重新执行3.3.2.
2. **另开**一个终端，运行gdb，然后在gdb界面里运行如下命令：

```
target remote:1234          # 建立gdb与gdb server间的连接。这时候我们看到输出带有??，
                            # 这是因为没有加载符号表，gdb不知道运行的是什么代码。
file Linux源码路径/vmlinux # 加载符号表。（不要把左边的东西原封不动地复制过来）
n                           # 单步运行。此时可以看到右边不是??，而是具体的函数名了。
break start_kernel          # 设置断点在start_kernel函数。
c                           # 运行到断点。
l                           # 查看断点代码。
p init_task                 # 查看断点处名为"init_task"的变量值。这个变量是个结构体，里面有一大堆成员。
```



第四部分： 检查要求

本次实验无实验报告。

本次实验共10分。

4.1 第一部分检查要求

检查学生是否安装了Linux系统（不局限于Ubuntu）。本部分不计分。

4.2 第二部分检查要求

1. 我们现场随便给出一条本实验文档未涉及的指令，你需要自己使用 `man` 指令阅读其手册简要解释该指令的含义，并简要介绍任意一个参数的含义。本部分共1分。若遇到不认识的英文单词，可以现场搜索。
2. 在助教的电脑上或机房的电脑上，在助教的监督下，使用实验提供的测试程序进行现场测试，每人最多尝试2次，取最高分。**答题时不得打小抄、查阅实验文档。**从题库中抽4题，每题抽4个选项，选项顺序会随机打乱。答题总时间120秒。每题1分。满分4分。为防止同一学生在不同助教处刷次数，每一次尝试都记录成绩。
   题库见提供的csv文件，三个csv都是题库。csv文件里，第一列是题目，第二列是正确答案，其余是干扰项。公布的题库、自测程序和考察时使用的完全相同。csv文件是UTF-8 with BOM格式，可以直接用Excel打开。但Excel可能会将部分选项解释异常，因此建议使用文本编辑器打开。自测程序由Python语言编写，在Linux下，在自测程序目录下执行 `python3 test.py` 即可运行测试程序。Ubuntu自带Python3。若未安装Python3，可用apt自行安装。强烈建议大家在检查之前自测几次，熟悉程序的工作流程。
   为防止替考及背完就忘的情况，在后续实验中，如果发现有同学忘了指令含义，我们可能会考虑让其重做一遍测试题，并按照错误个数扣实验分。

   > 因为往年很多同学实验做到最后还是对Linux指令很不熟悉，所以只能出此下策强迫大家进行记忆。

   > 为保证本评测程序在多种操作系统下均可运行，秒数刷新时会覆盖掉已输入的数字。但这些数字确实是已经输入了的，只是被后续读秒的输出覆盖而不可见。所以，若确信输入正确，直接回车即可。若怀疑输入错误，请长按 `Backspace` 保证之前输入的全被删掉之后再输入新的。

4.3 第三部分检查要求

现场检查能否启动虚拟机并启动gdb调试，即现场执行3.3.5。本部分共5分。本部分检查中，允许学生对照实验文档操作。

- 你需要能够简要解释符号表的作用。若不能解释，本部分减1分。
- 此外，**为提高检查效率，请自己编写一个【shell脚本】启动qemu（即，使用2.4所述方法完成3.3.5的第一步）。** 若不能使用脚本启动qemu，本部分减1分。你不需要使用脚本启动或操作gdb。

实验二 添加Linux系统调用

实验目的

- 学习如何使用Linux系统调用：实现一个简单的shell
- 学习如何添加Linux系统调用：实现一个简单的top

实验环境

- OS: Ubuntu 20.04.4 LTS
- Linux内核版本: 4.9.263

实验时间安排

> 注：此处为实验发布时的安排计划，请以课程主页和课程群内最新公告为准

- 4.1晚实验课，讲解实验 (19:00开始) 及检查
- 4.8晚实验课，检查实验
- 4.15晚实验课，检查实验

> 补检查分数照常给分，但会有标记记录此次检查未按时完成，标记会在最后综合分数时作为一种参考。

友情提示

- 本次实验以实验一为基础。一些步骤（如如何启动qemu运行Linux内核）不会在实验说明中详述。如果有不熟悉的步骤，请复习实验一。
- 请注意单词拼写、大小写，如common、asmlinkage、user等。
- 在本次实验中，如果你写出的代码出现了数组越界，则会出现很多奇怪的错误（如段错误（Segmentation Fault）、输出其他数组的值或一段乱码（注意，烫烫烫是Visual C的特性，Linux下没有烫烫烫）、其他数组的值被改变等）。提问前请先排查是否出现了此类问题。
- 如果同学们遇到了问题，请先查询在线文档。在线文档地址：
  https://docs.qq.com/sheet/DR1dZTnFRTURHc051

第一部分 续·Linux与Shell指令教学

1.1 Linux指令

1.1.1 echo

输出一个字符串。用法：`echo string`。

同样也可以输出一个文件的内容，如`echo file`将文件file的内容以纯文本形式读并输出。

经常在Shell脚本中使用该指令，以打印指定信息。

1.1.2 top

ps 命令可以一次性给出当前系统中进程状态，但使用此方式得到的信息缺乏时效性，并且，如果管理员需要实时监控进程运行情况，就必须不停地执行 ps 命令，这显然是缺乏效率的。

为此，Linux 提供了top命令。top 命令动态地持续监听进程地运行状态，与此同时，该命令还提供了一个交互界面，用户可以根据需要，人性化地定制自己的输出，进而更清楚地了进程的运行状态。直接输入`top`即可使用。

参数	含义
-d 秒数	指定 top 命令每隔几秒更新。默认是 3 秒。
-b	使用批处理模式输出。一般和`-n`选项合用，用于把 top 命令重定向到文件中。
-n 次数	指定 top 命令执行的次数。一般和`-b`选项合用。
-p 进程PID	仅查看指定 ID 的进程。
-u 用户名	只监听某个用户的进程。

在 top 命令的显示窗口中，还可以使用如下按键，进行以下交互操作：

按键 (注意大小写)	含义
? 或 h	显示交互模式的帮助。
P	按照 CPU 的使用率排序，默认就是此选项。
M	按照内存的使用率排序。
N	按照 PID 排序。
k	按照 PID 给予某个进程一个信号。一般用于中止某个进程，信号 9 是强制中止的信号。
q	退出 top 命令。

1.1.3 sleep

`sleep n`可以使当前终端暂停n秒。常见于shell脚本中，用于实现两条指令间的等待。

1.1.4 grep

筛选并高亮指定字符串。尝试在终端下运行 `grep a`，它会等待用户输入。若输入一行不带a的字符串并按回车，它什么都不会输出。若输入多行字符串，其中某些行带a，它会筛选出带a的行，并将该行的a高亮后输出。

1.1.5 wc

英文全拼：wordcount

常见用法：`wc [-lw] [filename]`，用于统计字数。

参数	含义
-l	统计行数。
-w	统计字数。
filename	统计指定文件的行数/字数。若不指定，会从标准输入(C/C++的stdin/scanf/cin)处读取数据。

1.2 Shell指令

1.2.1 管道符 |

问题：我们想统计Linux下进程的数量，应该如何解决？

一个很麻烦的解决方法是：先 `ps aux` 输出所有进程的列表，然后复制它的输出，执行 `wc -l`，将之前 `ps` 输出的内容粘贴到标准输入，传递给 `wc` 以统计 `ps` 输出的行数，即进程数量。有没有方法可以免去复制粘贴的麻烦？

管道符 `|` 可以将前面一个命令的标准输出（对应C/C++的stdout/printf/cout）传递给下一个命令，作为它的**标准输入**（对应C/C++的stdin/scanf/cin）。

- 例1：在终端中运行 `ps aux | wc -l` 可以显示所有进程的数量。
- 例2：先打开Linux下的firefox，然后在终端运行 `ps aux | grep firefox` 可以显示所有进程名含firefox的进程。
- 例3：接例2，在终端运行 `ps aux | grep firefox | wc -l` 可以显示所有进程名含firefox的进程的数量。

> 思考：`wc -l` 的作用是统计输出的行数。所以 `ps aux | wc -l` 统计出的数字真的是进程数量吗？

1.2.2 重定向符 >/>>/<

重定向符 `>`，`>>`：它可以把前面一个命令的标准输出保存到文件内。如果文件不存在，则会创建文件。`>` 表示覆盖文件，`>>` 表示追加文件。

- 举例：`ps -aux > ps.txt` 可以把当前运行的所有进程信息输出到ps.txt。ps.txt的原有内容会被覆盖。
- 举例：`ps -aux >> ps.txt` 可以把当前运行的所有进程信息追加写到ps.txt。

重定向符 `<`：可以将 `<` 前的指令的标准输入重定向为 `<` 后文件的内容。

- 举例：`wc -l < ps.txt` 可以把ps.txt中记录的信息作为命令的输入，即统计ps.txt中的行数。

1.2.3 分隔符 ;

子命令：每行命令可能由若干个子命令组成，各子命令由 `;` 分隔，这些子命令会被按序依次执行。

如：`ps -a; pwd; ls -a` 表示：先打印当前用户运行的进程，然后打印当前shell所在的目录，最后显示当前目录下所有文件。

第二部分 实现一个Linux Shell

> 注意：
>
> - 本部分的主要目的是锻炼大家**使用** Linux系统调用编写程序的能力，不会涉及编写一个完整的系统调用。
> - 本部分主要是代码填空，费力的部分已由助教完成，代码量不大，大家不必恐慌。**需要大家完成的代码已经用注释 `TODO:` 标记，可以通过搜索轻松找到**，使用支持TODO高亮编辑器（如vscode装TODO highlight插件）的同学也可以通过高亮找到要添加内容的地方。

2.1 Shell的基本原理

我们在各种操作系统中会遇到各式各样的用于输入命令、基于文字与系统交互的终端，Windows下的cmd和powershell、Unix下的sh、bash和zsh等等，这些都是不同的shell，我们可以看到它们的共同点——基于“命令”(command) 与系统交互，完成相应的工作。

Shell处理命令的方式很好理解，每条输入的“命令”其实都对应着一个可执行文件，例如我们输入 `top` 时，**shell程序会创建一个新进程并在新进程中运行可执行文件。具体而言，shell程序在特定的文件夹中搜索名为"top"的可执行文件，搜索到之后运行该可执行文件，并将进程的输出定向到指定的位置 (如终端设备)**，这一过程也是我们本次实验要实现的内容。具体来讲：

- **创建新进程：**用到我们课上讲过的 `fork()` 。
- **运行可执行文件：**使用课上讲的 `exec()` 系列函数。
  - exec开头的函数有很多，如 `execvp` 、 `execl` 等，可以使用man查看不同exec函数的区别及使用方法，最后选取合适的函数。
  - exec中需要指定可执行文件，因为用户的命令给出的只有文件名本身，没有给出文件的所在位置，所以在系统中通常会定义一个环境变量PATH来记录一组文件夹，所有命令对应的可执行文件通常存在于某个文件夹下，找不到则返回不存在。可以使用 `export -p PATH` 来查看你现在使用的shell都会去哪些文件夹下查找可执行程序。
- **结果输出：**命令默认会从STDIN_FILENO (默认值为0) 中读输入，并输出到STDOUT_FILENO (默认值为1) 中。在shell中也会有其他的输入来源和输出目标，如将一个命令的输出作为另一个命令的输入，在这种场合下就需要进行进程间的通信，比如采用我们课上讲过的管道pipe，执行两个命令的进程一个在写端输出内容，一个在读端读取输入。此外，命令还可以设置从文件输入和输出到文件，比如将执行命令程序中的STDOUT_FILENO使用文件描述符覆盖后，命令的执行结果就输出到文件中。

2.2 Shell内建指令

`cd` 命令可以用来切换shell的当前目录。但需要指出的是，不同于其他命令（比如 `ls` ，我们可以在/bin下面找到一个名为 `ls` 的可执行文件）， `cd` 命令其实是一个shell内置指令。由于子进程无法修改父进程的参数，所以若不使用内建命令而是fork出一个子进程并且在子进程中exec一个 `cd` 程序，因为子进程执行结束后会回到了父shell环境，而父shell的路径根本没有被改变，最终无法得到期望的结果。同理，不仅是 `cd` ，**改变当前shell的参数（如 `source` 命令、 `exit` 命令）基本都是由shell内建命令实现的** 。

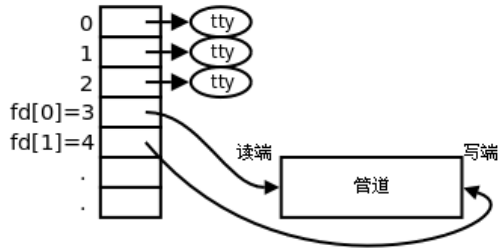
2.3 有关管道的背景知识

“一切皆文件”是Unix/Linux的基本哲学之一。普通文件、目录、I/O设备等在Unix/Linux都被当做文件来对待。虽然他们的类型不同，但是linux系统为它们提供了一套统一的操作接口，即文件的open/read/write/close等。当我们调用Linux的系统调用打开一个文件时，系统会返回一个文件描述符，每个文件描述符与一个打开的文件唯一对应。 之后我们可以通过文件描述符来对文件进行操作。管道也是一样，我们可以通过类似文件的read/write操作来对管道进行读写。为便于理解，本次实验使用匿名管道。匿名管道具有以下特点：

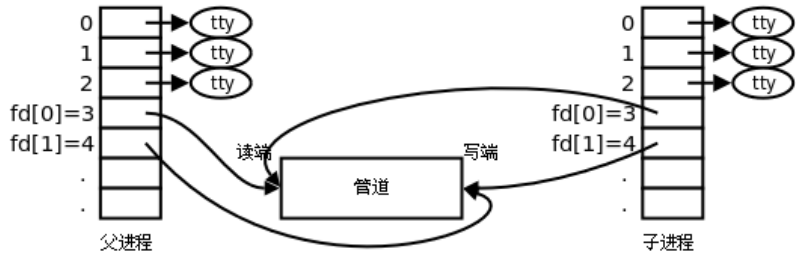
1. 只能用于父子进程等有血缘关系的进程；
2. 写端不关闭，并且不写，读端读完，继续等待，此时阻塞，直到有数据写入才继续（就好比你的C程序在scanf，但你一直什么都不输入，程序会停住）；
    - 尤其地，假如一条管道有多个写端，那么只有在所有写端都关闭之后（管道的引用数降为0），读端才会解除阻塞状态。
3. 读端不关闭，并且不读，写端写满管道buffer，此时阻塞，直到管道有空位才继续；
4. 读端关闭，写端在写，那么写进程收到信号SIGPIPE，通常导致进程异常中止；
5. 写端关闭，读端在读，那么读端在读完之后再读会返回0；
6. 匿名管道的通信通常是一次性的，如果需要反复通信，可以使用命名管道。

一般来说，匿名管道的使用方法是：

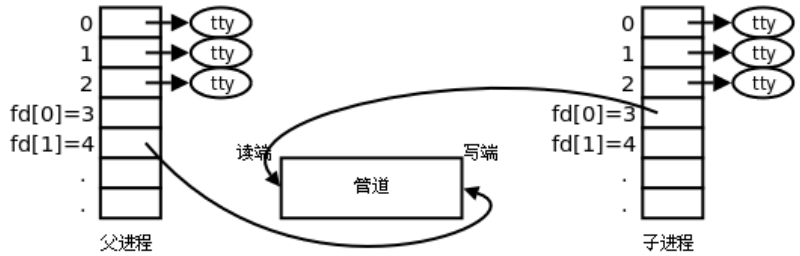
1. 父进程创建管道



2. 父进程fork出子进程



3. 父进程关闭fd[0]，子进程关闭fd[1]



- 首先，父进程调用pipe函数创建一个匿名管道。pipe的函数原型是 `int pipe(int pipefd[2])` 。我们传入一个长为2的一维数组 `pipefd` ，Linux会将 `pipefd[0]` 设为**读端**的文件描述符，并将 `pipefd[1]` 设为**写端**的文件描述符。（注：此时管道已被打开，相当于已调用了 `open` 函数打开文件）但是需要注意：此时管道的读端和写端接在了同一个进程上。如果你此时往 `pipefd[1]` 里写入数据，这些数据可以从 `pipefd[0]` 里读出来。不过这种“我传我自己”(原地tp) 通常没什么意义，我们接下来要把管道应用于进程通信。
- 其次，使用 `fork` 函数创建一个子进程。 `fork` 完成之后，数组 `pipefd` 也会被复制。此时，子进程也拥有了对管道的控制权。若目的是父进程向子进程发送数据，那么父进程就是写端，子进程就是读端。我们应该把父进程的读端关闭，把子进程的写端关闭，进而便于数据从父进程流向子进程。

    - 如果不关闭子进程的写端，子进程会一直等待（参考2.3.2）。
- 因为匿名管道是单向的，所以如果想实现从子进程向父进程发送数据，就得另开一个管道。
- 父子进程调用 `write` 函数和 `read` 函数写入、读出数据。
    - `write` 函数的原型是： `ssize_t write(int fd, const void * buf, size_t count);`
    - `read` 函数的原型是： `ssize_t read(int fd, void * buf, size_t count);`
    - 如果你要向管道里读写数据，那么这里的 `fd` 就是上面的 `pipefd[0]` 或 `pipefd[1]` 。
    - 这两个函数的使用方法在此不多赘述。如有疑问，可以百度。

> **注意：如果你的数组越了界，或在read/write的时候count的值比buf的大小更大，则会出现很多奇怪的错误（如段错误（Segmentation Fault）、其他数组的值被改变、输出其他数组的值或一段乱码（注意，烫烫烫是Visual C的特性，Linux下没有烫烫烫）等）。提问前请先排查是否出现了此类问题。**

2.4 输入/输出的重定向

> 注：本节描述的是如何将一个程序产生的标准输出转移到其他非标准输出的地方，不特指'>'和'>>'符号。

在使用 '|', '>', '>>', '<' 时，我们需要将程序输出的内容重定向为文件输出或其他程序的输入。在本次实验中，为方便起见，我们将shell作为重定向的中转站。

- 当出现前三种符号时，我们需要把前一指令的标准输出重定向为管道，让父进程（即shell）截获它的标准输出，然后由shell决定将前一指令的输出转发到下一进程、文件或标准输出（即屏幕）。
- 当出现'|' 和 '<'时，我们需要把后一指令的标准输入重定向为管道，让父进程（即shell）把前一进程被截获的标准输出/指定文件读出的内容通过管道发给后一进程。

重定向使用 `dup2` 系统调用完成。其原型为： `int dup2(int oldfd, int newfd);` 。该函数相当于将 `newfd` 标识符变成 `oldfd` 的一个拷贝，与 `newfd` 相关的输入/输出都会重定向到 `oldfd` 中。如果 `newfd` 之前已被打开，则先将其关闭。举例：下述程序在屏幕上没有输出，而在文件输出"hello!goodbye!"。屏幕上不会出现"hello!"和"goodbye"。

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

int main(void)
{
    int fd;

    fd = open("./test.txt", O_RDWR | O_CREAT | O_TRUNC, 0666);

    // 此代码运行在dup2之前，本应显示到屏幕，但实际上是暂时放到屏幕输出缓冲区
    printf("hello!");

    // 程序默认是输出到STDOUT_FILENO即1中，现在我们让1重定向到fd。
    // 也就是将标准输出的内容重定向到文件，因此屏幕输出缓冲区以及后面的printf
    // 语句本应输出到屏幕（即标准输出，fd为STDOUT_FILENO）的内容重定向到文件中。
    dup2(fd, 1);

    printf("goodbye!\n");
    return 0;
}
```

> 易混淆的地方：向文件/管道内写入数据实际是程序的一个输出过程。

2.5 一些shell命令的例子和结果分析

> 本部分旨在帮助想要进一步理解真实shell行为的同学，希望下述示例及结果分析可以为同学们带来启发。

这一节中，我们给出一些shell命令的例子，并分析命令的输出结果，让大家能够更直观的理解shell的运行。**我们并不要求实验中实现的shell行为和真实的shell完全一致，但你自己需要能够清楚地解释自己实现的处理逻辑。**但在实现shell时，可以参考真实shell的这些行为。这些实验都是在bash下测试的，大家也可以自己在Ubuntu等的终端中重复这些实验。在这一节的代码中，每行开头为 `$` 的，是输入的shell命令，剩下的行是shell的输出结果。如：

```
$ cd /bin
$ pwd
/bin
```

表示在shell中了先运行了 `cd /bin` ，然后运行了 `pwd` ，第一条命令没有任何输出，第二条命令输出为 `/bin` 。 `pwd` 命令会显示shell的当前目录，我们先用 `cd` 命令进入了 `/bin` 目录，所以 `pwd` 输出 `/bin` 。

2.5.1 多个子命令

由 `;` 分隔的多个子命令，和在多行中依次运行这些子命令效果相同。

```
$ cd /bin ; pwd
/bin
$ echo hello ; echo my ; echo shell
hello
my
shell
```

`echo` 会将它的参数打印到标准输出，如 `echo hello world` 会输出 `hello world` 。这个实验用 `;` 分隔了三个shell命令，结果和在三行中分别运行这些命令一致。

2.5.2 管道符

这个实验展示了shell处理管道时的行为，相同的命令在管道中行为可能和单独运行时不同。

```
$ echo hello | echo my | echo shell
shell
```

如上，虽然管道中的上一条命令的输出被重定向至下一条命令的输入，但是因为 `echo` 命令本身不接受输入，所以前两个 `echo` 的结果不会显示。如果实验检查中，你提供了这样的测试样例，是不能证明正确实现了管道的。

```
$ cd /bin ; pwd
/bin
$ cd /etc | pwd
/bin
```

如上， `cd /etc` 并没有改变shell的当前目录，这是因为**管道中的内置命令也是在新的子进程中运行的**，所以不会改变当前进程（shell）的状态。而在**不包含管道的命令中，内置命令在shell父进程中运行，外部命令在子进程中运行**，所以，不包含管道的内置命令能够改变当前进程（shell）的状态。可以用 `type` 命令检查命令是否是内部命令。

```
$ type cd
cd is a shell builtin
$ type cat
cat is hashed (/bin/cat)
```

说明 `cd` 是内置命令，而 `cat` 则是调用 `/bin/cat` 的外置命令。接下来我们测试管道中命令的运行顺序。

```
$ sleep 0.02 | sleep 0.02 | sleep 0.02 | ps | grep sleep
15840 tty1     00:00:00 sleep
$ sleep 0.02 | sleep 0.02 | sleep 0.02 | ps | grep sleep
15845 tty1     00:00:00 sleep
15846 tty1     00:00:00 sleep
15847 tty1     00:00:00 sleep
```

可以看到，运行了两次相同的命令，却得到了不同的结果，多次运行该命令，每次输出的sleep行数不同，从0行到3行都有可能（根据电脑速度和核数，可能需要将sleep后面的数字增大或缩小来重复该实验）。这说明**管道中各个命令是并行执行的**，`ps`命令运行时，前面的`sleep`命令可能执行结束，也有可能仍在执行。

2.5.3 重定向

```
$ cd /tmp ; mkdir test ; cd test
$ echo hello > a >> b > c
$ ls
a  b  c
$ cat a
$ cat b
$ cat c
hello
```

当一个命令中出现多个输出重定向时，虽然所有文件都会被建立，但是只有最后一个文件会真正被写入命令的输出。

```
$ cd /tmp ; mkdir test ; cd test
$ echo hello > out | grep hello
$ cat out
hello
```

当输出重定向和管道符同时使用时，命令会将结果输出到文件中，而管道中的下一个命令将接收不到任何字符。

```
$ cd /tmp ; mkdir test ; cd test
$ > out2 echo hello ; cat out2
hello
```

重定向符可以写在命令前。

2.6 总结：本次实验中shell执行命令的流程

- 第一步：打印命令提示符（类似shell ->）。
- 第二步：把分隔符`;`连接的各条命令分割开。（多命令选做内容）
- 第三步：对于单条命令，把管道符`|`连接的各部分分割开。
- 第四步：如果命令为单一命令没有管道，先根据命令设置标准输入和标准输出的重定向（重定向选做内容）；再检查是否是shell内置指令：是则处理内置指令后进入下一个循环；如果不是，则fork出一个子进程，然后在fork出的子进程中exec运行命令，等待运行结束。（如果不fork直接exec，会怎么样？）
- 第五步：如果只有一个管道，创建一个管道，并将子进程1的**标准输出重定向到管道写端**，然后fork出一个子进程，根据命令重新设置标准输入和标准输出的重定向（重定向选做内容），在子进程中先检查是否为内置指令，是则处理内置指令，否则exec运行命令；子进程2的**标准输入重定向到管道读端**，同子进程1的运行思路。（注：2.4的样例分析中我们得出，管道的多个命令之间，虽然某个命令可能会因为等待前一个进程的输出而阻塞，但整体是没有顺序执行的，即并发执行。所以我们为了让多个内置指令可以并发，需要在fork出子进程后才执行内置指令）

- 第六步：如果有多个管道，参考第三步，n个进程创建n-1个管道，每次将子进程的**标准输出重定向到管道写端**，父进程保存**对应管道的读端**（上一个子进程向管道写入的内容），并使得下一个进程的**标准输入重定向到保存的读端**，直到最后一个进程使用标准输出将结果打印到终端。（多管道选做内容）
- 第七步：根据第二步结果确定是否有剩余命令未执行，如果有，返回第三步执行（多命令选做内容）；否则进入下一步。（分隔符多命令和管道连接的多命令实现方式上有什么区别？为什么？）
- 第八步：打印新的命令提示符，进入下一轮循环。

2.7 任务目标

代码填空实现一个shell。这个shell不要求在qemu下运行。功能包括：

必做部分

- 实现运行单条命令（非shell内置命令）。
- 支持一条命令中有单个管道符 `|` 。
- 实现exit, cd两个shell内置指令。
- 在shell上能显示当前所在的目录。如：{ustc}shell: [/home/ustc/exp2/] > （后面是用户的输入）

二选一部分

- 实现子命令符 `;` 和重定向符 `>` , `>>` , `<` 。
- 支持一条命令中有多个管道符 `|` 。如果你确信你的多管道功能可以正常实现单管道功能，代码填空里的单管道可以不做。

其他说明

- 我们使用的是Linux系统调用，请不要尝试在Windows下运行自己写的shell程序。那是不可能的。
- 需要自行设计测试样例以验证你的实现是正确的。如测试单管道时使用 `ps aux | wc -l` ，与自带的shell输出结果进行比较。
- 不限制分隔符、管道符、重定向符的符号优先级。你可以参考我们代码框架中实现、提示的优先级。
- 我们提供了本次实验使用的系统调用API的范围，请同学们自行查询它们的使用方法。你在实验中可能会用到它们中的一部分。 你也可以使用不属于本表的系统调用。
    - read/write
    - open/close
    - pipe
    - dup/dup2
    - getpid/getcwd
    - fork/vfork/clone/wait/exec
    - mkdir/rmdir/chdir
    - exit/kill
    - shutdown/reboot
    - chmod/chown
- 如果你想问如何编译、运行自己写的代码，请参考lab1的2.3.17和2.2.3。
- 得分细则见文档4.2。

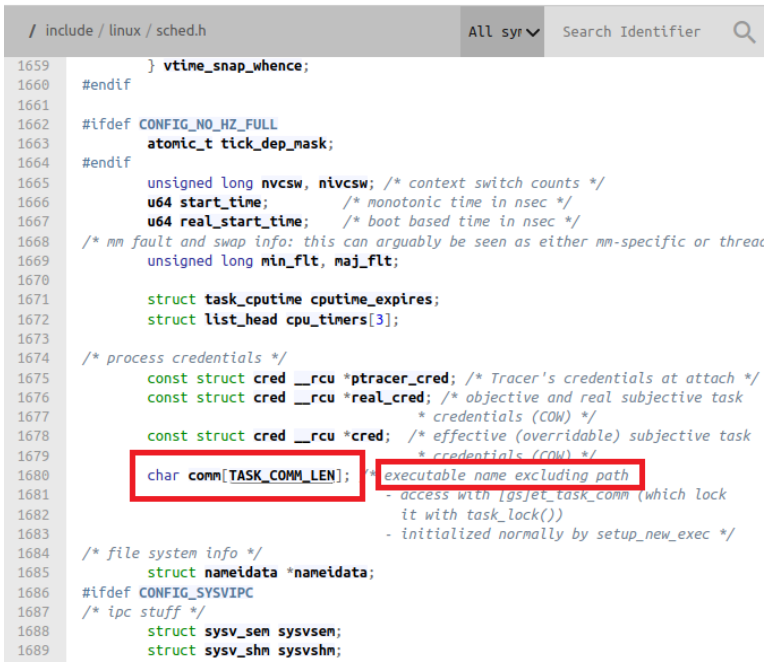
第三部分 编写系统调用实现一个Linux top

3.0 如何阅读Linux源码

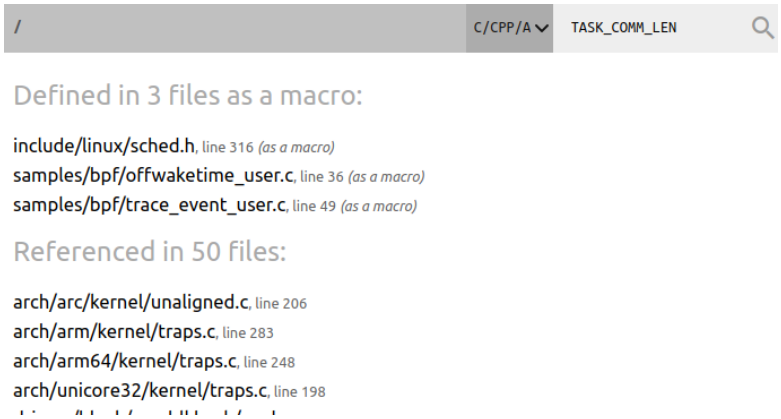
因为直接在VSCode、Visual Studio等本地软件中阅读Linux源码会让电脑运行速度变得很慢，所以我们建议使用在线阅读。一个在线阅读站是 https://elixir.bootlin.com/linux/v4.9.263/source 。左栏可以选择内核源码版本，右边是目录树或代码，右上角有搜索框。下面举例描述一次查阅Linux源码的过程。

假定我们要查询Linux中进程名最长有多长。首先，我们知道进程在内存中的数据结构是结构体 `task_struct` ，所以我们首先查找 `task_struct` 在哪。搜索发现在 `include/linux/sched.h` 。（下面那个从名字来看像个附属模块，所以大概可以确定结构体的声明应该在 `sched.h` 。）

进入 `sched.h` 查看 `task_struct` 的声明。我们不难猜到，进程名应该是个一维char数组。所以我们开始找 `char` 数组。看了几十个成员之后，我们找到了 `char comm[TASK_COMM_LEN]` 这个东西。右边注释写着 `executable name excluding path` ，应该就是这个了。所以它的长度就应该是 `TASK_COMM_LEN` 。



于是我们点击一下这个 `TASK_COMM_LEN` ，找到这个宏在哪定义。发现定义也在 `sched.h` 。点开就能看到宏定义了。至于它的长度到底是多少，留作习题，给大家自己实践查询。



3.1 若干名词解释

3.1.1 用户空间、内核空间

参考PPT。

3.1.2 系统调用

系统调用 (System call, Syscall)是操作系统提供给**用户程序访问内核空间的合法接口**。

系统调用**运行于内核空间，可以被用户空间调用**，是内核空间和用户空间划分的关键所在，它保证了两个空间必要的联系。从用户空间来看，系统调用是一组统一的抽象接口，用户程序**无需考虑接口下面是什么**；从内核空间来看，用户程序不能直接进行敏感操作，所有对内核的操作都必须通过功能受限的接口间接完成，保证了**内核空间的稳定和安全**。

> 我们平时的编程中**为什么没有意识到系统调用的存在**？这是因为应用程序现在一般**通过应用编程接口** (API)**来间接使用系统调用**，比如我们如果想要C程序打印内容到终端，只需要使用C的标准库函数API `printf()` 就可以了，而不是使用系统调用 `write()` 将内容写到终端的输出文件（`STDOUT_FILENO`）中。

3.1.3 glibc

glibc是GNU发布的c运行库。glibc是linux系统中最底层的api，几乎其它任何运行库都会依赖于glibc。glibc除了封装linux操作系统所提供的系统服务外，它本身也提供了许多其它一些必要功能服务的实现，其内容包罗万象。glibc内含的档案群分散于系统的目录树中，像支架一般撑起整个操作系统。

`stdio.h`，`malloc.h`，`unistd.h` 等都是glibc实现的封装。

3.2 系统调用是如何执行的

1. 调用应用程序调用**库函数（API）**
   Linux使用的开源标准C运行库glibc（GNU libc）有一个头文件 `unistd.h`，其中声明了很多**封装好的**系统调用函数，如 `read/write`。这些API会调用实际的系统调用。
2. API将**系统调用号存入**EAX，然后**触发软中断**使系统进入内核空间。
   32位x86机器使用汇编代码 `int $0x80` 触发中断。具体如何触发软中断可以去参考Linux的实现源码（如 `arch/x86/entry/entry_32.S`）。因为这部分涉及触发中断的汇编指令，这部分代码是用汇编语言写的。
3. 内核的中断处理函数根据系统调用号，调用对应的**内核函数**（也就是课上讲的的系统调用）。前面的 `read/write`，实际上就是调用了内核函数 `sys_read/sys_write`。

> 总结：添加系统调用需要**注册中断处理函数和对应的调用号**，内核才能找到这个系统调用，并执行对应的内核函数。

4. 内核函数完成相应功能，将返回值存入EAX，返回到中断处理函数；

> 注1：系统执行相应功能，调用的是C代码编写的函数，而不是汇编代码，减轻了实现系统调用的负担。系统调用的函数定义在 `include/linux/syscalls.h` 中。因为汇编代码到C代码的参数传递是通过栈实现的，所以，所有系统调用的函数前面都使用了 `asmlinkage` 宏，它意味着编译时限制只使用栈来传递参数。如果不这么做，用汇编代码调用C函数时可能会产生异常。
> 注2：用户空间和内核空间各自的地址是不能直接互相访问的，需要借助函数拷贝来实现数据传递，后面会说明。

5. 中断处理函数返回到API中；
6. API将EAX返回给应用程序。

> 注：当glibc库没有封装某个系统调用时，我们就没办法通过使用封装好的API来调用该系统调用，而使用 `int $0x80` 触发中断又需要进行汇编代码的编写，这两种方法都不适合我们在添加过新系统调用后再编写测试代码去调用。因此我们后面采用的是第三种方法，使用glibc提供的 `syscall` 库函数，这个库函数只需要传入调用号和对应内核函数要用到的参数，就可以直接调用我们新写的系统调用。具体详见 3.4.1.

3.3 添加系统调用的流程

> Linux 4.9的文档 `linux-4.9.263/Documentation/adding-syscalls.txt` 中有说明如何添加一个系统调用。本实验文档假设所有同学使用的平台都是x86，采用的是x86平台的系统调用添加方法。其他平台的同学可以参考上述文档给出的其他平台系统调用添加方法。

在实现系统调用之前，要先考虑好添加系统调用的函数原型，确定函数名称、要传入的参数个数和类型、返回值的意义。在实际设计中，还要考虑加入系统调用的必要性。

我们这里以一个统计系统中进程个数的系统调用 `ps_counter(int *num)` 作为演示，num是返回值对应的地址。

> 因为系统调用的返回值表示系统调用的完成状态（0是正常，其他值是异常，代表错误编号），所以不建议用返回值传递信息。

本节的各步是没有严格顺序的，但在编译前都要完成：

3.3.1 注册系统调用

内核的汇编代码会在 `arch/x86/include/generated/asm/syscalls_64.h` 中查找调用号。为便于添加系统调用，x86平台提供了一个专门用来注册系统调用的文件 `/arch/x86/entry/syscalls/syscall_64.tbl` 。在编译时，脚本 `arch/x86/entry/syscalls/syscalltbl.sh` 会被运行，将上述 `syscall_64.tbl` 文件中登记过的系统调用都生成到前面的 `syscalls_64.h` 文件中。因此我们需要修改 `syscall_64.tbl` 。

打开 `linux源码路径/arch/x86/entry/syscalls/syscall_64.tbl` 。

```
# linux-4.9.263/arch/x86/entry/syscalls/syscall_64.tbl
# ... 前面还有，但没展示
329 common  pkey_mprotect       sys_pkey_mprotect
330 common  pkey_alloc          sys_pkey_alloc
331 common  pkey_free           sys_pkey_free
# 以上是系统自带的，不是我写的，下面这个是
332 common  ps_counter          sys_ps_counter
```

这个文件是**x86平台下的系统调用注册表**，记录了很多数字与函数名的映射关系。我们在这个文件中注册系统调用。
每一行从左到右各字段的含义，及填写方法是：

- 系统调用号：请为你的系统调用找一个没有用过的数字作为调用号（记住调用号，后面要用）
- 调用类型：本次实验请选择common（含义为：x86_64和x32都适用）
- 系统调用名xxx
- 内核函数名sys_xxx

> 提示：为保持代码美观，每一列用制表符排得很整齐。但经本人测试，无论你是用制表符还是用空格分隔，无论你用多少空格或制表符，只要你把每一列分开了，无论是否整齐，对这一步的完成都没有影响。

3.3.2 声明内核函数原型

打开 `linux-4.9.263/include/linux/syscalls.h`，里面是对于系统调用函数原型的定义，在最后面加上我们创建的新的系统调用函数原型，格式为 `asmlinkage long sys_xxx(...)` 。注意，如果传入了用户空间的地址，需要加入 `__user` 宏来说明。

```
asmlinkage long sys_mlock2(unsigned long start, size_t len, int flags);
asmlinkage long sys_pkey_mprotect(unsigned long start, size_t len,
              unsigned long prot, int pkey);
asmlinkage long sys_pkey_alloc(unsigned long flags, unsigned long init_val);
asmlinkage long sys_pkey_free(int pkey);
//这里是我们新增的系统调用
asmlinkage long sys_ps_counter(int __user * num);

#endif
```

3.3.3 实现内核函数

你可以在 `linux-4.9.264/kernel/sys.c` 代码的最后添加你自己的函数，这个文件中有很多已经实现的系统调用的函数作为参考。我们给出了一段示例代码：

```
SYSCALL_DEFINE1(ps_counter, int __user *, num){
    struct task_struct* task;
    int counter = 0;
    printk("[Syscall] ps_counter\n");
    for_each_process(task){
        counter ++;
    }
    copy_to_user(num, &counter, sizeof(int));
    return 0;
}
```

以下是这段代码的解释：

1. 使用宏 `SYSCALL_DEFINEx` 来简化实现的过程，其中x代表参数的个数。传入宏的参数为：调用名(不带sys_)、以及每个参数的类型、名称（注意这里输入时为两项）。

   以我们现在的 `sys_ps_counter` 为例。因为使用了一个参数，所以定义语句是 `SYSCALL_DEFINE1(ps_counter, int *, num)` 。如果该系统调用有两个参数，那就应该是 `SYSCALL_DEFINE2(ps_counter, 参数1的类型, 参数1的名称, 参数2的类型, 参数2的名称)` ，以此类推。

   > 无法通过 `SYSCALL_DEFINEx` 定义二维数组（如 `char (*p)[50]` ）为参数。你可以尝试定义一个，然后结合编译器的报错信息分析该宏的实现原理，并分析不能定义的原因。

2. `printk()` 是内核中实现的print函数，和 `printf()` 的使用方式相同，`printk()` 会将信息写入到内核日志中。尤其地，在qemu下调试内核时，系统日志会直接打印到屏幕上，所以我们可以直接在屏幕上看到 `printk` 打印出的内容。`printf()` 是使用了C的标准库函数的时候才能使用的，而内核中无法使用标准库函数。你不能 `#include<stdio.h>` ，自然不能用 `printf()` 。

   > 在使用 `printk` 时，行首输出的时间等信息是去不掉的。我们会用此方法检查ps的输出是否是用户态。

3. 为了获取当前的所有进程，我们使用了宏函数 `for_each_process(p)` （定义在 `include/linux/sched.h` 中），遍历当前所有任务的信息结构体 `task_struct` （同样定义在 `include/linux/sched.h` 中），并将地址赋值给参数 `p` 。后面实验内容中我们会进一步用到 `task_struct` 中的成员变量来获取更多相关信息，注意我们暂时不讨论多线程的场景，每一个 `task_struct` 对应的可以认为就是一个进程。
4. 前面说到，系统调用是在内核空间中执行，所以如果我们要将在内核空间获取的值传递给用户空间，需要使用函数 `copy_to_user(void __user *to, const void *from, unsigned long n)` 来完成从内核空间到用户空间的复制。 其中， `from` 和 `to` 是复制的来源和目标地址， `n` 是复制的大小，我们这里就是 `sizeof(int)` 。

3.4 测试

> 若想使用VS Code调试Linux内核，可以参考此文章： **https://zhuanlan.zhihu.com/p/105069730** ，直接从“VS Code配置”一章阅读即可。实际操作中，你可能还需要在VS Code中安装"C/C++"和"C/C++ Extension Pack"插件。

3.4.1 编写测试代码

在你的Linux环境下（不是打开qemu后弹出的那个）编写测试代码。我们在这里命名其为 `get_ps_num.c` 。新增的系统调用可以使用 `long int syscall (long int sysno, ...)` 来触发， `sysno` 就是我们前面添加的调用号，后面跟着要传入的参数，下面是一个简单的测试代码样例：

```
#include<stdio.h>
#include<unistd.h>
#include<sys/syscall.h>
int main(void)
{
    int result;
    syscall(332, &result);
    printf("process number is %d\n",result);
    return 0;
}
```

> 提示：
>
> - 这里的测试代码是用户态代码，不是内核态代码，所以可以使用 `printf` 。
> - 为使用系统调用号调用系统调用，需要使用 `sys/syscall.h` 内的 `syscall` 函数。
> - 我们推荐使用C语言而不是C++，因为C++在下一步(3.4.2)的静态编译时容易出现问题。使用C语言时需要注意C和C++之间的语法差异。

3.4.2 编译

使用gcc编译器编译 `get_ps_num.c` 。**本次实验需要使用静态编译（ `-static` 选项）** 。要用到的gcc指令原型是： `gcc [-static] [-o outfile] infile` 。

3.4.3 运行测试程序

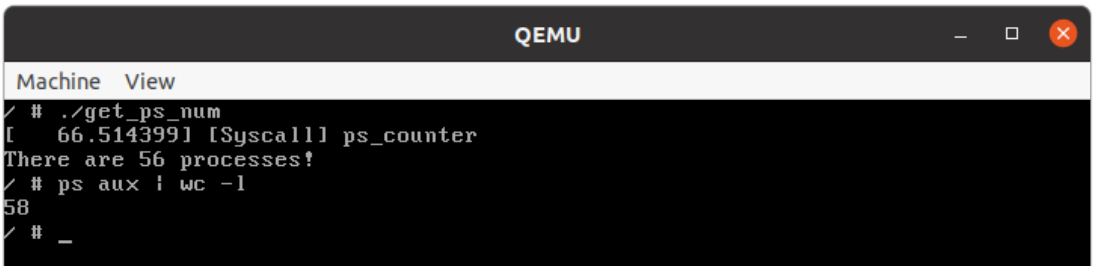
1. 将3.4.2编译出的可执行文件 `get_ps_num` 放到 `busybox-1.32.1/_install` 下面，重新制作initramfs文件（重新执行Lab1的3.2.6.6的find操作），这样我们才能在qemu中看见编译好的 `get_ps_num` 可执行文件。

   > 注意：因为_install文件夹只有root用户才有修改权限，所以复制文件应该在命令行下使用 `sudo` 。

2. 重新编译linux源码（重新执行Lab1的3.2.4.3，无需重新make menuconfig）
3. 运行qemu（重新执行Lab1的3.2.7，我们在这里不要求使用gdb调试）

4. 在qemu中运行 `get_ps_num` 程序，得到当前的进程数量（这个进程数量是包括 `get_ps_num` 程序本身的，下图中，除了 `get_ps_num` 本身外，还有55个进程）。
5. 验证：使用 `ps aux | wc -l` 获取当前进程数。但这个输出的结果与上一步中得到的不同。请自己思考数字不同的原因。

> 在实际操作中，系统中的进程数量可能不是55。只要与 `ps aux | wc -l` 得到的结果匹配即可。

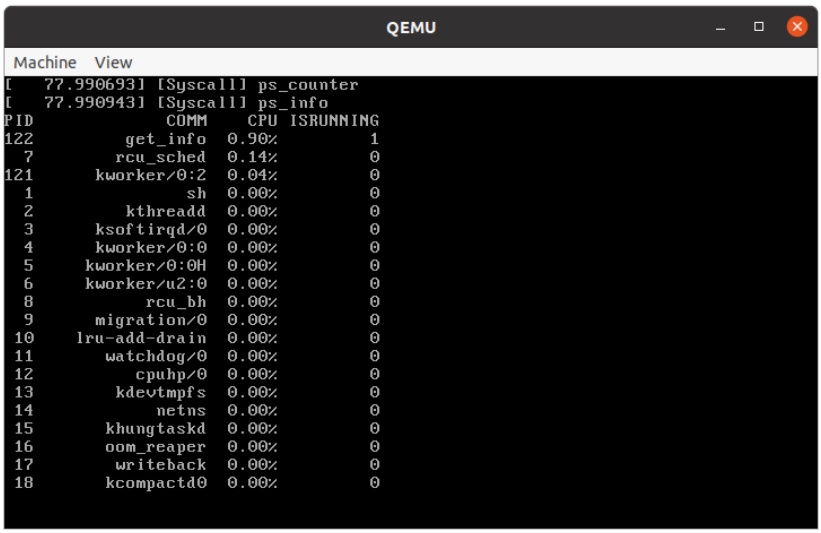


3.5 任务目标

- 在linux4.9下创建适当的（可以是一个或多个）系统调用。利用新实现的系统调用，实现一个linux4.9下的进程状态信息统计程序，参考命令 `top`：

```
  PID   PR  S  %CPU  COMMAND
142420   20  S   6.0  netease-cloud-m
146353   20  S   4.0  chrome
  2100   20  S   3.3  pulseaudio
  2542   20  S   3.3  deepin-system-m
   786   20  S   2.7  Xorg
  1742   20  S   1.3  kwin_x11
  1822   20  S   1.3  dde-dock
```

- 输出的信息需要包括：
  - 进程的PID
  - 进程的COMMAND（进程名）
  - 进程是否处于running状态
  - 进程的CPU占用率
- 每一秒刷新一次信息。输出的信息需要按CPU占用率排序。输出前20行即可，无需输出在两次刷新间隔之间新建/消失的进程。程序需要一直运行，无需考虑你写的 `top` 程序如何关闭。
- 添加的系统调用在被调用时需要 `printk` 出自己的系统调用名（如 `ps_counter` ）
- 具体评分规则见4.3。 我们不限制大家创建系统调用的数量、功能，也不限制测试程序的实现。能完成实验内容即可。
- 下面是示例输出。进程信息是随时间变化的，状态值、排序并不固定。



3.6 任务提示

- 前面示例中 `for_each_process` 获取到的 `task_struct` 结构体保存有对应进程的很多信息，如:

成员变量	成员含义	其他说明
`char comm[TASK_COMM_LEN]`	进程名	
`pid_t pid`	pid	
`volatile long state`	进程的状态，如运行、停止、僵尸进程等	进程的状态定义于 `include/linux/sched.h` 的197～244行。Linux使用位掩码(bitmask)实现多种状态的叠加，若想了解请自行搜索。
`struct sched_entity se`	该进程的调度实体，里面记录了很多与进程调度相关的信息	`sched_entity` 结构体中的 `sum_exec_runtime` 成员定义进程从进程开始时算起的实际运行时间。它的单位是纳秒(ns)。

- 内核空间的成员变量（如 `task_struct` 结构体及其成员）要借助 `copy_to_user(void __user *to, const void *from, unsigned long n)` 函数复制到用户空间的变量中，才可以在用户空间访问内容。
- 如果需要将用户空间的变量复制到内核空间，用 `copy_from_user(void *to, const void __user *from, unsigned long n)`
- 在你的top程序（不是内核代码）中，使用 `<unistd.h>` 中的 `sleep` 函数可以使程序等待，使用 `<stdlib.h>` 中的 `system("clear")` 可以清屏。
- 进程在一段物理时间内的CPU占用率的计算方法：进程在这段时间内的实际运行时间/物理时间长度。
- **如果访问数组越界，会出现很多奇怪的错误（如程序运行时报malloc错误、段错误、输出其他数组的值或一段乱码（注意，烫烫烫是Visual C的特性，Linux下没有烫烫烫）、其他数组的值被改变等）。提问前请先排查是否出现了此类问题。**
- 一些常见错误：
  - 拼写错误，如 `asmlkage` 、 `commom` 、 `__uesr` 、中文逗号
  - 语法错误，如 `sizeof(16 * int)`
  - C标准问题，如C语言声明结构体、算 `sizeof` 时应该用 `struct xxx` 而不是直接 `xxx`

  上述错误均可以通过阅读编译器报错信息查出。

第四部分 实验评分标准

本次实验共10分，无实验报告。评分方式为按点给分，某个点做出来即可拿到对应的分数，不同的得分点之间没有约束关系，比如本实验中最难的按cpu使用率排序和多管道、重定向这些都没做，但是其他的基础内容都做出来了，也是可以拿到8分的。

4.1 知识问答 (2')

我们会使用随机数发生器随机地从下面题库中抽三道题，每回答正确一题得一分，满分2分。也就是说，允许答错一题。请按照自己的理解答题，答题时不准念实验文档或念提前准备的答案稿。
下面是题库：

1. 解释 `wc` 和 `grep` 指令的含义。
2. 解释 `ps aux | grep firefox | wc -l` 的含义。
3. `echo aaa | echo bbb | echo ccc` 是否适合做shell实验中管道符的检查用例？说明原因。
4. 对于匿名管道，如果写端不关闭，并且不写，读端会怎样？
5. 对于匿名管道，如果读端关闭，但写端仍尝试写入，写端会怎样？
6. 假如使用匿名管道从父进程向子进程传输数据，这时子进程不写数据，为什么子进程要关闭管道的写端？
7. fork之后，是管道从一分为二，变成两根管道了吗？如果不是，复制的是什么？
8. 解释系统调用 `dup2` 的作用。
9. 什么是shell内置指令，为什么不能fork一个子进程然后 `exec cd` ？
10. 为什么 `ps aux | wc -l` 得出的结果比 `get_ps_num` 多2？
11. 进程名的最大长度是多少？这个长度在哪定义？
12. `task_struct` 在Linux源码的哪个文件中定义？
13. 为什么无法通过 `SYSCALL_DEFINEx` 定义二维数组（如 `char (*p)[50]` ）为参数？
14. 在修改内核代码的时候，能用 `printf` 调试吗？如果不能，应该用什么调试？
15. `read()`、`write()`、`dup2()` 都能直接调用。现在我们已经写好了一个名为ps_counter的系统调用。为什么我们不能在测试代码中直接调 `ps_counter()` 来调用系统调用？

大家可以自己思考或互相讨论这些题目的答案（但不得在本课程群内公布答案）。助教不会公布这些题目的答案。

4.2 “实现一个Shell”部分检查标准 (4')

- 支持基本的单条命令运行、支持两条命令间的管道 `|` 、内建命令（只要求 `cd` / `exit` ），得1分。
- 选做：
  - 支持多条命令间的管道 `|` 操作，得2分。
  - 支持重定向符 `>` 、 `>>` 、 `<` 和分号 `;` ，得2分。
- 若两个选做都做，另加1分。
- 你需要流畅地说明你是如何实现实验要求的（即，现场讲解代码）。本部分共1分。若未能完成任何一个功能，则该部分分数无效。

> 注：实验检查时，请**自行准备**可以验证相应功能支持的测试样例。若我们认为测试样例不能验证程序是否有误，则该部分不计分。

4.3 “编写系统调用”部分检查标准 (4')

- 你的程序需要在【用户态】每秒打印一次各进程的CPU占用统计，并按CPU占用统计排序。本部分共1分。
- 你的程序需要在【用户态】每秒打印一次各进程的PID、进程名、进程是否处于running状态。本部分共1分。
- 你需要在现场阅读Linux源码，展示pid的数据类型是什么（即，透过多层 `typedef` 找到真正的pid数据类型）。本部分共1分。
- 你需要流畅地展示你修改了哪些文件的代码，允许对着实验文档描述（即，现场讲解代码）。本部分共1分。若未能完成任何一个功能，则该部分分数无效。
- 不准在内核态直接 `printk` 进程信息。否则会被视为该部分未完成。
- 不准在用户态直接调用ps/top等工具，或读取procfs获取进程信息，否则会被视为该部分未完成。

实验3.1 动态内存分配器malloc的实现

版本1.1

> **说明：这是Lab3的第一部分。下周还会放出第二部分。本次实验不接受分部分检查，请两部分做完之后一起检查。**

实验目的

- 使用显式空闲链表实现一个64位堆内存分配器
  - 实现两种基本搜索算法
  - 实现堆内存的动态扩容
  - 实现实时的分配器内存使用情况统计
- 学会以动态链接库的形式制作库并使用
- 体会系统实验数据的测量和分析过程

实验环境

- OS: Ubuntu 20.04.4 LTS
- 无需在QEMU下调试

实验时间安排

> 注：此处为实验发布时的安排计划，请以课程主页和课程群内最新公告为准

- 4.22晚实验课，讲解实验第一部分
- 4.29晚实验课，讲解实验第二部分及检查
- 5.6晚实验课，检查实验
- 5.13晚实验课，检查实验

> 补检查分数照常给分，但会有标记记录此次检查未按时完成，标记会在最后综合分数时作为一种参考。

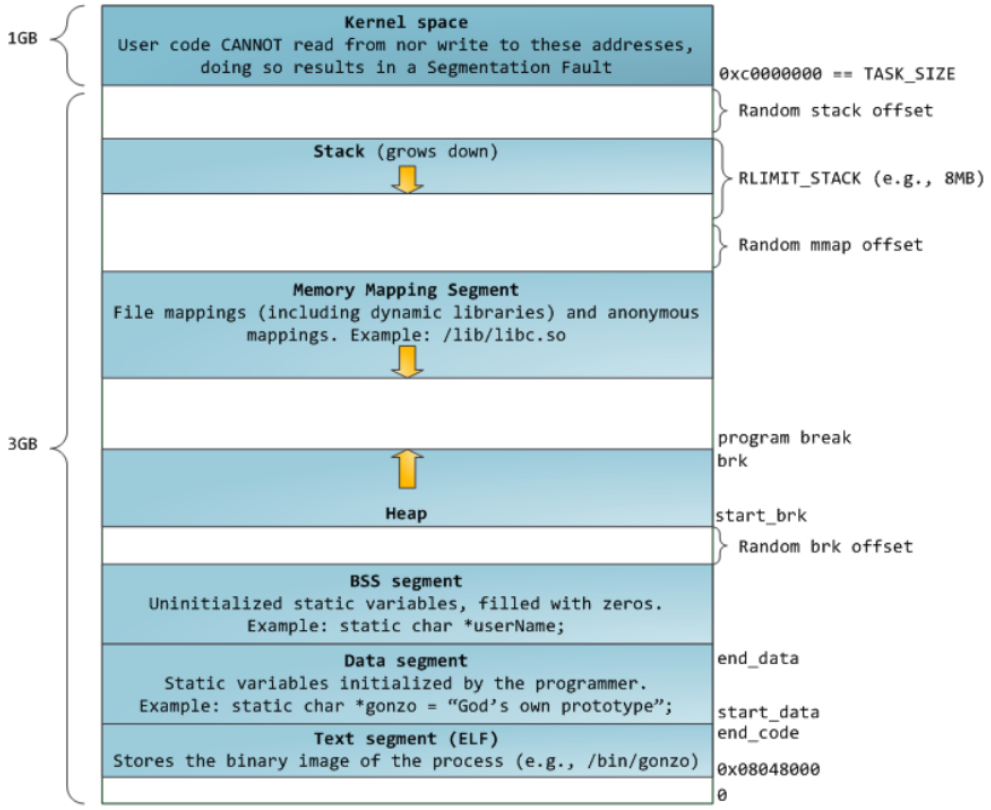
友情提示

- **本实验难度较大，目的是为了拉开区分度，将同学们的平时投入体现在期末总评之中，同时锻炼较为优秀的同学的系统设计能力与代码能力。本实验在设计时的目标并不是让所有同学都能轻易拿到满分。**
- 如果同学们遇到了问题，请先查询在线文档。在线文档地址：
  https://docs.qq.com/sheet/DR1dZTnFRTURHc051

第一部分 内存分配器的基本概念与原理

1.1 内存分配的原理

我们在课上学到了内存分配的底层系统调用`brk()`，它可以让你的应用程序堆空间向上增长一定的范围，从而分配更多的内存。在linux内核中，除了`brk()`，还有一种很常用的、同样向内核申请内存的系统调用`mmap()`，它可以在堆和栈之间未被分配的内存空间中，以用户定义或者内核自己决定的方式从一个特定的地址开始分配一块内存，而非依靠堆的增长来分配。如下图所示，32位操作系统中的堆、栈和mmap各自从不同的起始地址开始在不同的区域增长、分配。（顺便提一句，这里有个random offset是为了不让恶意用户猜到起始地址，并通过缓冲区溢出等方式非法获取你的内存）。



那么为什么有了`brk()`，还需要有`mmap()`呢？不难理解，mmap是一种更为自由的分配方式，尤其是在回收已分配的内存空间时。如果brk先增长了2GB然后又增长了4KB，只要这4KB还在占用，2GB的内存空间就算是不用了，也无法被即时的回收，因为brk只有从堆顶向下移动的方式来回收，而被4KB的使用“卡住”。而mmap中的内存块是单独存在的，释放也是经由专有接口`munmap()`来释放，所以在mmap的2GB内存用完之后，可以直接调用`munmap()`还给内核。

mmap虽然更为灵活，但也带来了更长的分配延迟，因为mmap需要搜索空闲空间，确认是否可以分配，而brk直接增长堆顶指针就可以了。这里就出现了一对trade-off：mmap更灵活，brk更快。

> 注：在我们学到按需调页之后会知道，这里分配的其实只是虚拟内存空间，在实际访问时才会触发缺页中断实际分配内存。

1.2 内存分配器的作用

> 内核提供了分配内存的接口，用户可以直接调用，为什么还需要用户使用内存分配器呢？

前面学到，系统调用的过程需要从用户态进入内核态，处理完成后再回到用户态，这种用户态和内核态的切换是有明显开销的。因此，在我们经常会用到的内存申请这一操作中，如果每次内存申请都去调用内存分配，就类似于我们每次买书都去找出版社买。系统工程师们给出的解决方案就是在用户态提供一个内存分配器，它一次“批发”一定量的内存，然后用户调用的时候作“零售”，如果“存货”不够了就再去“进货”，相当于一个书店的作用。

内存分配器的作用不仅仅在于减少系统调用开销。买到的书看完了可以卖给书店，从而再转手给其他人，出版社一般不做这个。即，内存分配器可以把用户释放掉的内存，先临时保存，如果有新的内存申请，正好可以用掉这块内存，就再分配出去，这样既避免了回收内存给内核的开销，又减少了向内核申请内存的开销。

因此，编写应用程序时，使用一个内存分配器来保证时间和空间两方面的高效性，是很有必要的。

1.3 如何设计一个内存分配器

> 分配器并不是全世界一套的，甚至每个应用都可以自己实现一个适合自己的内存分配器。不同分配器的设计有哪些相同点，差别又在哪里？

最常用的内存分配器莫过于同学们大一就接触到的C语言中的 `malloc()` ，这个malloc其实只是glibc中的实现方式，它有一个更唯一的名字——ptmalloc。除了ptmalloc，比较著名的内存分配器还有Facebook的jemalloc、Google的tcmalloc和Microsoft的mimalloc，应用在各种小到个人应用、大到企业业务的场景下。

那么，内存分配器该如何设计呢？我们先从内存分配时延最低的思路出发。

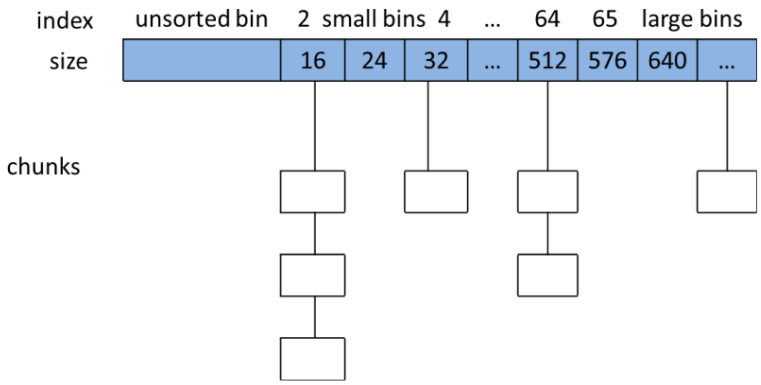
内存分配器批发了一大块内存，如何更快的响应用户的请求呢？最简单的就是一次向内核申请一大块内存，类似于使用堆的方式做分配，用户需要多少，增加多少，然后返回增加的指针，在大块内存快用完时，再向内核申请新的。这样时延的问题保证了“最优”。

但是我们再考虑用户释放内存的重用，就会发现，堆只向上增长，是没考虑重用的。怎么办呢？我们必然要维护一个记录有空闲空间信息的数据结构，来保证可以在每次分配的时候都能用到那些释放了的内存。

维护这个数据结构，最简单的是使用一个空闲链表，插入释放的内存，删除要分配的内存（数组插入和删除开销更大），但是这样我们时延的最优就被牺牲了，遍历链表的开销是很大的，就算我们只是找到第一个可以放得下的空闲块（first-fit），最差时也要遍历整个链表。

截止到这里，我们的设计已经是一个“可以工作”的阶段了，书店的“仓库”使用堆来管理，“书架”使用链表来管理。本次实验的内容也是实现到这个程度，因为进一步的优化伴随着更为复杂的设计与大量的代码工程。但是我们可以简单了解一下，真正投入使用的内存分配器，又有哪些更进一步的优化方法：

- **ptmalloc** 使用了多级链表来管理空闲空间以减少搜索的开销，16Byte一个链表，24Byte一个链表，以此类推；



- **jemalloc** 使用buddy tree减少内存分配中的碎片；
- **tcmalloc** 使用slab分配器和线程级cache来进一步降低分配的时延。

总的来讲，设计内存分配器主要围绕分配时延、内存空间利用率（内存碎片量、或者说内存浪费程度）以及多线程下的分配带宽三点性能来讨论，我们本次实验中会关注到前面两项最基础的性能指标，即单线程下的分配时延与分配器内存空间利用率，并引导大家发现first-fit和best-fit两种算法在两方面的优劣，理解其中的trade-off，即内存分配中也是No Silver Bullet的。

1.4 本实验中的内存分配器流程

1. 堆空间初始化，使用sbrk从内核申请5MB的空间，堆指针指向最低位置；
2. 内存分配器初始化，堆空间中取4KB空间（堆指针向上增加4KB）加入到空闲链表；
3. 用户调用malloc函数申请 `request_size` 大小的内存；
4. 搜索空闲链表是否有符合条件的块（first-fit best-fit），如果找到则转到6；
5. 没有符合条件的块，则内存分配器向堆空间申请max(4KB, request_size)大小的内存，做一次尝试合并（查看地址相邻的前后的块是否也是空闲的），加入空闲链表，并作为符合条件的块返回；
6. 对符合条件的块作处理，若该块分配过 `request_size` 大小的内存后还剩余较多内存（大于 `MIN_BLK_SIZE` ），则需先分割出空闲部分，加入空闲链表，然后将分配出去的块从空闲链表移除。

参考资料

- 《Computer Systems: A Programmer's Perspective 3rd》
- 《Glibc 内存管理——Ptmalloc2源码分析》（如果想了解ptmalloc是如何设计实现的，非常推荐阅读）
- 《Understanding the Linux Virtual Memory Manager》（深入理解计算机系统虚拟内存管理）
- 《Malloc tutorial》

第二部分 代码与实现流程

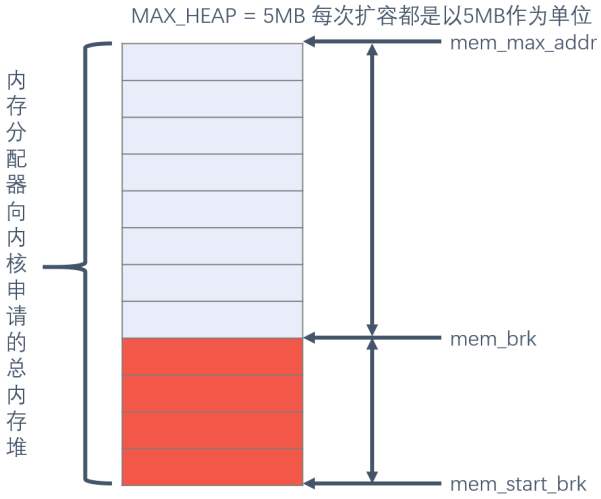
2.1 内存分配器与内核的交互模块

首先，我们将内存分配器调用 `sbrk()` 向内核申请内存空间的过程分离为一个单独的模块，它负责完成初始化时第一次sbrk申请内存，以及后续可用内存不够时再次调用 `sbrk()` 申请内存。我们将这部分功能放在文件 `memlib.c` 中。

> 在理论课程的学习中，我们了解到内存分配器可以通过调用 `brk()` 向内核申请空间。而实际上相关的系统调用有两个，分别是 `brk()` 和 `sbrk()` ，二者的声明如下：
>
> ```
> int brk( const void *addr );
> void* sbrk ( intptr_t incr );
> ```
>
> 这两个函数的主要作用，都是扩展堆的上界brk，且都是是系统调用函数。区别在于：
>
> - `brk()` 的参数 `addr` 为设置的新的 `brk` 上界地址；
> - `sbrk()` 的参数 `incr` 为需要申请的内存的大小。
>
> 同学们可以自行通过 `man` 查看二者更详细的介绍。本次实验中，我们要求大家使用 `sbrk()` 来完成堆的扩展。它的返回值是堆上界brk的旧值，也即新增内存空间的开始位置，因此第一次调用 `sbrk()` 时，它的返回值可以作为堆空间的起始地址。

主要变量如下所示，这里的 `mem_start_brk` 对应了**1.1**节图中Heap的start_brk， `mem_brk` 对应了同一张图中的brk。

变量	作用
`mem_start_brk`	记录内存块的第一个字节地址
`mem_brk`	记录已分配内存的最后一个字节的末尾地址
`mem_max_addr`	记录当前内存块的最大可用地址



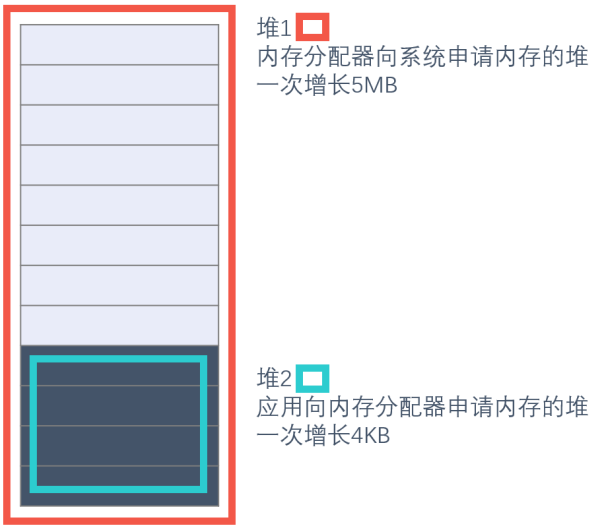
要说明的一点：我们这里的设计中存在两个堆：

- 第一个堆是内存分配器向内核申请的**堆1**，这个堆1每次增长5MB，更改 `mem_max_addr` ；
- 第二个堆是内存分配器分配内存时，将堆1看作一个没分配过的**堆2**，每次以分配内存或者至少为4KB单位的增长堆2，更改 `mem_brk` ，然后堆2分配出去的进入内存分配器的空闲链表，最后被应用使用。

这样做看上去很冗余，很“套娃”，但是我们这么做主要有以下几点：

1. 堆1的设计是为了减少系统调用次数，一次就向内核申请5MB的空间。

2. 堆2的设计是为了快速增长内存，如果我们发现空闲链表中分不下了，就会从堆2来做分配。
3. 功能的隔离，堆1是内存分配器和系统内核之间用的，堆2是内存分配器和应用之间用的，即内核--堆1-->内存分配器--堆2-->应用。



在本部分中，需要补充两个主要的功能函数，也就是初始化堆1和分配堆2的两个函数：

1. 堆1的初始化 `mem_init()`

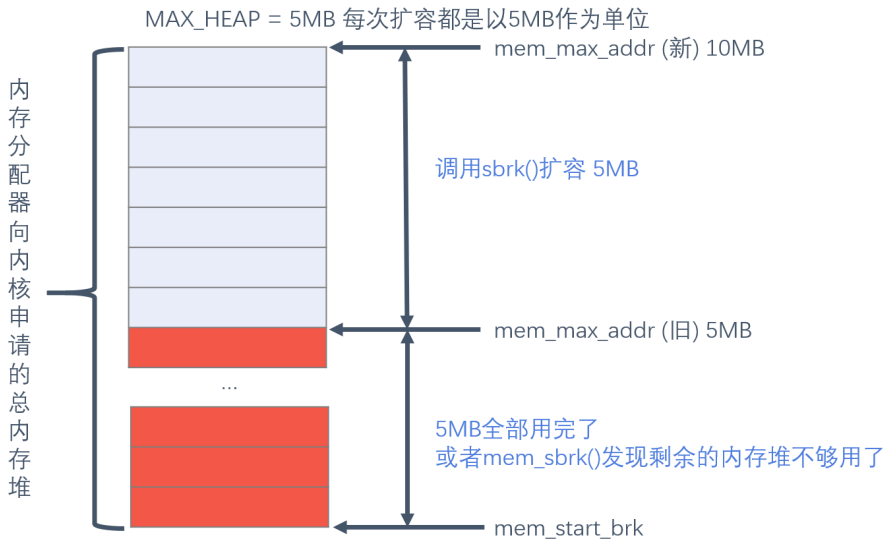
   比较简单，调用 `sbrk()` 获取5MB内存块，并记录对应的 `mem_start_brk` 、 `mem_max_addr` 和 `mem_brk` 。

2. 堆2的分配 `mem_sbrk()`

   首先要记录旧的 `mem_brk` ，作为分配首地址返回。

   然后分两种情况：如果加上incr之后 `mem_brk` 的值超过了当前的 `mem_max_addr` ，则证明堆1不够用了，需要再次调用 `sbrk()` 来增加堆1的大小，并修改相关变量，然后增加 `mem_brk` 。如果没有超过，就直接给 `mem_brk` 增加一个incr，增加堆2，即分配堆2。

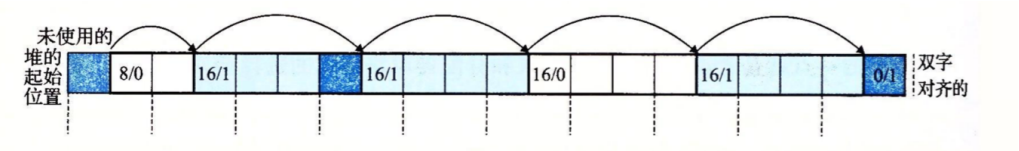
   最后返回旧的 `mem_brk` 。



以上为与内核交互的 `memlib.c` ，后面我们将围绕内存分配器实际分配和回收应用内存的 `mm.c` 展开讲解。

2.2 内存分配器空闲块管理——隐式空闲链表

与老师在课上讲授的内容相似，实际代码中，隐式空闲链表将堆中的内存块按地址顺序串成一个链表，以块头的块大小数据作为计算下一块地址的依据。接受到内存分配请求时，分配器遍历该链表来找到合适的空闲内存块并返回，这意味着分配块的时间是与内存空间的块数成线性关系。当找不到合适的空闲内存块时(如:堆内存不足,或没有大小足够的空闲内存块)，向堆顶扩展更多的内存。隐式空闲链表如下图所示：



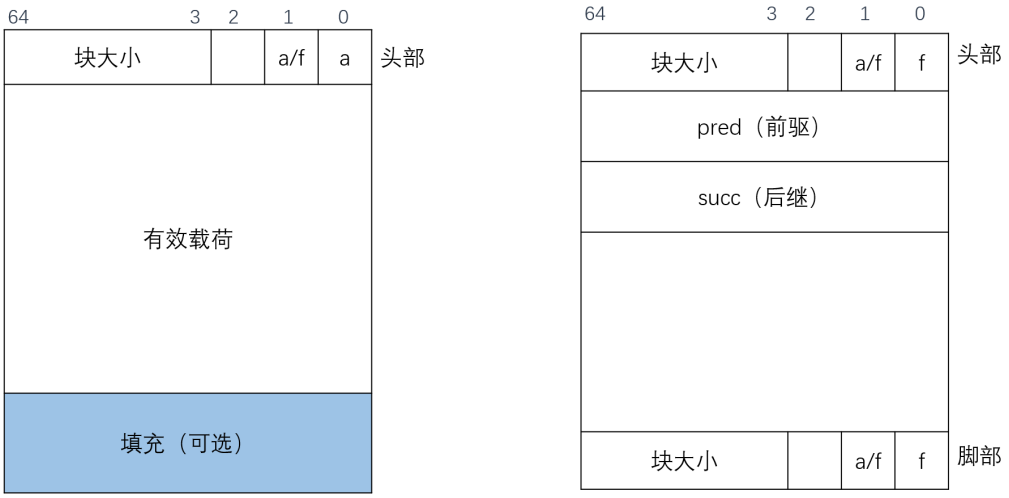
图中淡蓝色部分为已分配块，深蓝色为填充块（为了内存双字对齐），数字为块头部。

隐式空闲链表为我们提供了一种简单的分配方式。但是，在隐式空闲链表方案中：块分配时间复杂度与堆中块的总数呈线性关系。这在实际中是不能接受的。所以我们本次实验中主要设计和实现一种由双向链表组织的显式空闲链表方案。

2.3 内存分配器空闲块管理——显式空闲链表

2.3.1 块的格式

实际实现中通常将空闲块组织成某种形式的显式数据结构（如，链表）。由于空闲块的空间是不用的，所以实现链表的指针可以存放在空闲块的主体里。例如，将堆组织成一个双向的空闲链表，在每个空闲块中，都包含一个 `pred` （前驱）和 `succ` （后继）指针，分配块和空闲块的格式如下图所示：



对比隐式空闲链表，显式双向空闲链表的方式使适配算法的搜索时间由**块总数的线性时间**减少到**空闲块数量的线性时间**，因为它不需要搜索整个堆，而只是需要搜索空闲链表即可。

如上图所示，与隐式空闲链表相比，分配块和空闲块的格式都有变化。

- 首先，分配块没有了脚部，这可以优化空间利用率。回想前面的介绍，当进行块合并时，只有当前块的前面邻居块是空闲的情况下，才会使用到前邻居块的脚部。如果我们把前面邻居块的已分配/空闲信息位保存在当前块头部中未使用的低位中（比如第1位中），那么已分配的块就不需要脚部了。但是，一定注意：空闲块仍然需要脚部，因为脚部需要在合并时用到。
- 其次，空闲块中多了 `pred` （前驱）和 `succ` （后继）指针。正是由于空闲块中多了这两个指针，再加上头部、脚部的大小，所以最小的块大小为**4个字**。

下面详细说明一下分配块和空闲块的格式：

- 分配块：
  - 由头部、有效载荷部分、可选的填充部分组成。其中最重要的是头部的信息：
    - 头部大小为一个字(64 bits)，
    - 其中第3-64位存储该块大小的高位。（因为按字对齐，所以低三位都0）

        - 第0位的值表示该块是否已分配，0表示未分配（空闲块），1表示已分配（分配块）。
        - 第1位的值表示该块**前面的邻居块**是否已分配，0表示前邻居未分配，1表示前邻居已分配。
- 空闲块：
    - 由头部、前驱、后继、其余空闲部分、脚部组成。
        - 头部、脚部的信息与分配块的头部信息格式一样。
        - 前驱表示在空闲链表中前一个空闲块的地址。后继表示在空闲链表中后一个空闲块的地址。前驱和后继是组成空闲链表的关键。

2.3.2 分配器初始化与扩容堆2操作

首先，分配器实现会涉及大量的指针操作，包括从地址取值/赋值，查找块头/块脚地址等。为了方便操作以及保障操作性能，我们在 `mm.c` 中定义了如下宏操作。

宏	意义
`WSIZE`	字长
`DSIZE`	双字长
`CHUNKSIZE`	内存分配器扩容最小单元
`PACK`	块大小和分配位结合返回一个值
`GET` / `PUT`	对指针p指向的位置取值/赋值
`HDRP` / `FTRP`	返回bp指向块的头/脚部指针
`PREV_BLKP` / `NEXT_BLKP`	返回与bp相邻的上一/下一块
`GET_PRED` / `GET_SUCC`	返回与空闲块bp相连的上一个/下一个空闲块

```
/* Basic constants and macros */
#define WSIZE 8             /* Word and header/fonter size(bytes) */
#define DSIZE 16            /* Double word size(bytes) */
#define CHUNKSIZE (1 << 12)  /* Extend heap by this amount (bytes) */
#define MAX(x, y) ((x) > (y) ? (x) : (y))

/* Pack a size and allocated bit into a word */
#define PACK(size, prev_alloc, alloc) ((size) & ~(1<<1) | (prev_alloc << 1) & ~(1) | (alloc))
#define PACK_PREV_ALLOC(val, prev_alloc) ((val) & ~(1<<1) | (prev_alloc << 1))
#define PACK_ALLOC(val, alloc) ((val) | (alloc))

/* Read and write a word at address p */
#define GET(p) (*(unsigned long *)(p))
#define PUT(p, val) (*(unsigned long *)(p) = (val))

/* Read the size and allocated fields from address p */
#define GET_SIZE(p) (GET(p) & ~0x7)
#define GET_ALLOC(p) (GET(p) & 0x1)

/* Given block ptr bp, compute address of it's header and footer */
#define HDRP(bp) ((char *)(bp)-WSIZE)
#define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)

/* Given free block ptr bp, compute address of its previous and next free block */
#define GET_PRED(bp) (GET(bp))
#define SET_PRED(bp, val) (PUT(bp, val))
#define GET_SUCC(bp) (GET(bp + WSIZE))
#define SET_SUCC(bp, val) (PUT(bp + WSIZE, val))
```

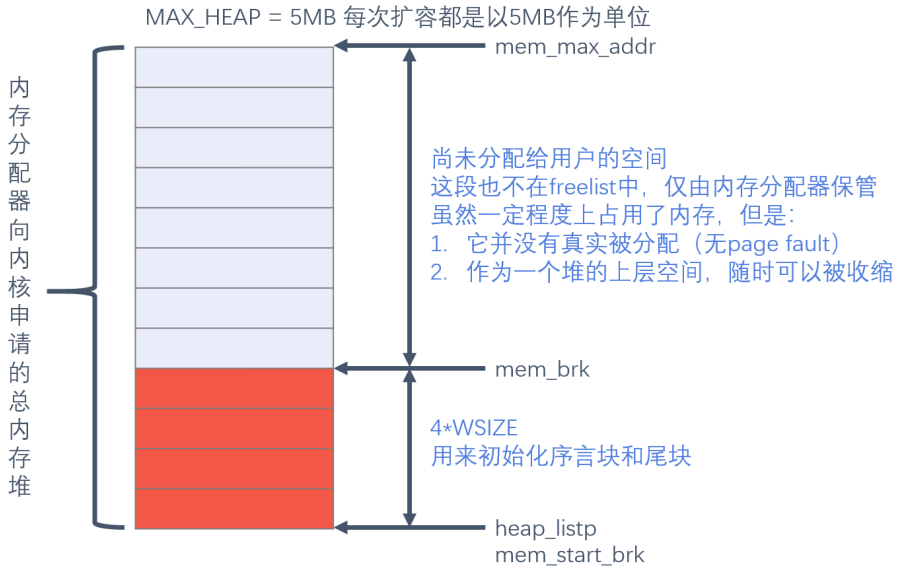
```
/* Given block ptr bp, compute address of next and previous blocks */
#define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(((char *)(bp)-WSIZE)))
#define PREV_BLKP(bp) ((char *)(bp)-GET_SIZE(((char *)(bp)-DSIZE)))
/* Used for the sp place() function */
#define MIN_BLK_SIZE (2 * DSIZE)
```

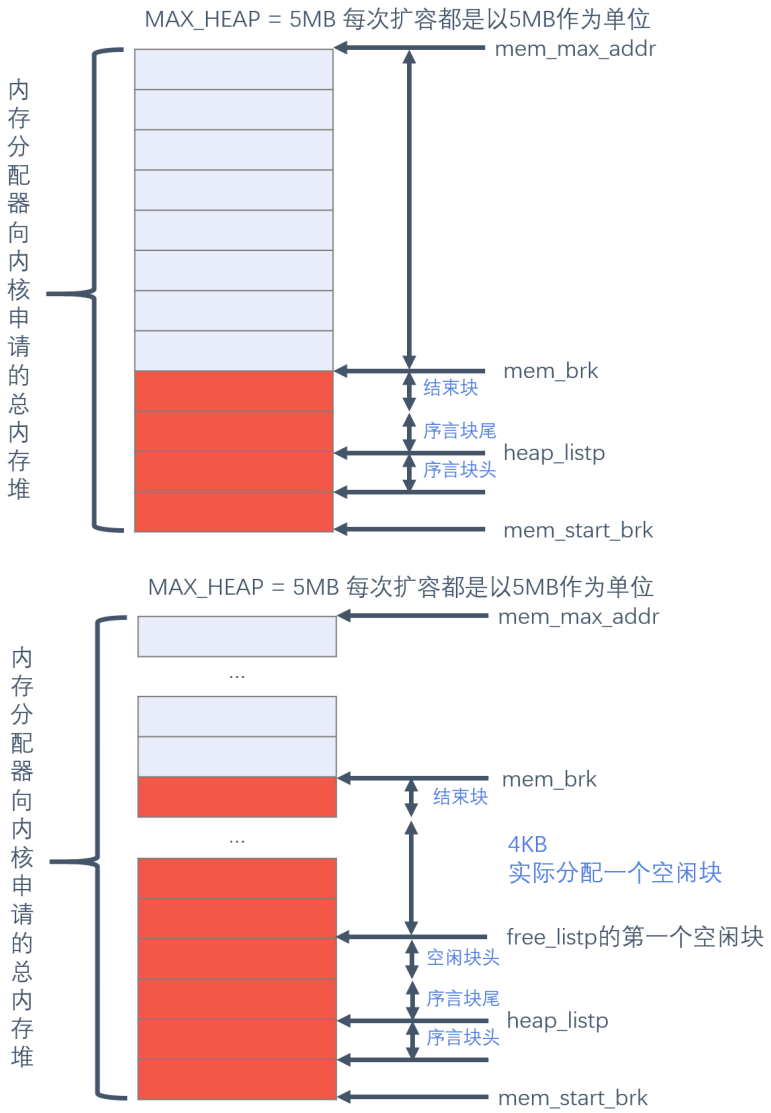
(1) 分配器初始化

在开始调用 `malloc()` 分配内存前，需要先调用 `mm_init()` 函数初始化分配器。其主要工作是分配初始堆内存，分配序言块和尾块，以及初始化空闲链表，如下代码所示。

```
/*
 * mm_init - initialize the malloc package.
 */
int mm_init(void)
{
    /* 首先通过mem_sbrk请求4个字的内存(模拟sbrk) */
    if ((heap_listp = mem_sbrk(4 * WSIZE)) == (void *)-1)
        return -1;
    /*  这四个字分别作为填充块（为了对齐）
     *  序言块头/脚部，尾块
     *  并将heap_listp指针指向序言块使其作为链表的第一个节点
     */
    PUT(heap_listp, 0);
    PUT(heap_listp + (1 * WSIZE), PACK(DSIZE, 1));
    PUT(heap_listp + (2 * WSIZE), PACK(DSIZE, 1));
    PUT(heap_listp + (3 * WSIZE), PACK(0, 1));
    heap_listp += (2 * WSIZE);
    /*
     * 调用extend_heap函数向系统申请一个CHUNKSIZE的内存作为堆的初始内存
     */
    if (extend_heap(CHUNKSIZE / WSIZE) == NULL)
        return -1;
    return 0;
}
```

初始化的流程图如下图所示。





(2) 堆2的扩容

当brk管理的堆2内内存无法满足申请要求 (即经过查找，发现不存在满足当前申请内存大小的空闲块）时，分配器会向总内存堆1申请内存，进行内存扩容。涉及到堆1的操作都在堆1的 `mem_sbrk()` 中完成，所以我们这里主要讲堆1给堆2增长过内存后，堆2如何管理这些新增内存，如何与已有块合并、插入到空闲链表。

```
/* extend_heap函数是对mem_sbrk的一层封装，接收的参数是要分配的字数，
 * 在堆初始化以及malloc找不到合适内存块时使用。
 * 它首先对请求大小进行地址对齐，然后调用mem_sbrk获取空间
 */
static void *extend_heap(size_t words)
{
    char *bp;
    size_t size;
    /* Allocate an even number of words to maintain alignment */
    size = (words % 2) ? (words + 1) * WSIZE : words * WSIZE;
    if ((long)(bp = mem_sbrk(size)) == -1)
        return NULL;
    /* Initialize free block header/footer and the epilogue header */
    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));

    /* Coalesce if the previous block was free */
```

```
    return coalesce(bp);
}
```

> 注意：这里实际操作是将扩展前的尾块作为了新空闲块的头块，然后在新的堆末尾分配一个新的尾块。

堆2内存中第一个字是一个为了内存对齐的填充字。填充字后面紧跟一个特殊的序言块，它是一个16字节的已分配块， 只由一个头部和一个脚部组成。序言块是在初始化时创建的，并且永不释放。序言块后面是普通块。堆的最后一个字 是一个特殊的结尾块，它是一个有效大小为0的已分配块，只由一个头部组成。

序言块和结尾块的作用是消除空闲块合并时的边界检查，在后续代码中可以看到这两个块的用途。

2.3.3 链表管理

在显式链表管理方案下，分配器维护一个指针 `heap_listp` 指向堆中的第一个内存块，也即序言块；另外，分配器还维护了另一个指针 `free_listp` 指向堆中的第一个空闲内存块。

- 访问空闲链表：当我们拥有某空闲块的地址 `bp` ，那么要想访问前一空闲块/后一空闲块，就可调用2.3.2中的 `GET_PREV` 和 `GET_SUCC` 宏获取其基地址，这相当于双向链表中的 `prev()` 和 `next()` 成员函数。
- 增删空闲链表：我们提供了 `add_to_free_list` 和 `delete_from_free_list` 两个接口，用来向双向链表中增加/删除空闲块。

2.3.4 分配块

按照代码执行流程，我们先介绍 `mm_malloc()` 函数，向堆申请 `size` 大小的内存并返回指针。

```
/*
 * mm_malloc - Allocate a block by incrementing the brk pointer.
 *     Always allocate a block whose size is a multiple of the alignment.
 */
void *mm_malloc(size_t size)
{
    size_t newsize;         /* Adjusted block size */
    size_t extend_size;     /* Amount to extend head if not fit */
    char *bp;
    /* Ignore spurious requesets */
    if (size == 0)
        return NULL;

    /* Adjust block size to include overhead and alignment reqs. */
    newsize = ALIGN(size) + DSIZE;

    /* Search the free list for a fit */
    if ((bp = find_fit(newsize)) != NULL)
    {
        place(bp, newsize);
        return bp;
    }
    /* no fit found. Get more memory and place the block */
    extend_size = MAX(newsize, CHUNKSIZE);
    if ((bp = extend_heap(extend_size / WSIZE)) == NULL)
    {
        return NULL;
    }
    place(bp, newsize);
    return bp;
}
```

首先将申请内存大小加上块头/尾部大小并进行对齐，然后调用 `find_fit()` 函数(想想怎么实现)从内存块链表中找到合适的块，如果成功找到则调用 `place()` 函数判断是否需要对该块作分割操作。

如果查找失败则向系统请求分配更多堆内存。为了避免频繁请求，一次最少申请 `CHUNKSIZE` 大小的内存。

2.3.5 分配策略

当我们调用 `malloc()` 发出一个内存分配请求时，分配器首先需要搜索堆中的内存块找到一个足够大的空闲块并返回。具体选择哪一个内存块由分配策略决定。主要有两种：

- 首次适配。从头开始搜索链表，找到第一个大小合适的空闲内存块便返回。
- 最佳适配。搜索整个链表，返回满足需求的最小的空闲块。

本次实验需要补充 `find_fit_first()` 和 `find_fit_best()` 两个函数的实现。从名字就可以看出，两个函数分别按照首次适配和最佳适配策略查找合适的空闲块。函数输入为申请空间大小 (指用户申请大小+分配块padding大小)，返回值为分配块的地址。（不需要对块内的具体数据进行更改）

2.3.6 放置块与分割空闲块

本次实验需要补充 `place()` 函数的实现，它对应了我们的放置块部分。

当分配器找到一个合适的空闲块后，便会调用 `place()` 函数进行放置，将空闲块格式改为分配块格式。同时，如果空闲块大小大于请求的内存大小，则需要分割该空闲块，避免内存浪费。

具体步骤为：

- 修改空闲块头部，将空闲块从空闲链表中删除，将大小改为分配的大小，并标记该块为已分配。
- 为多余的内存添加一个块头部，记录其大小并标记为未分配，正确设置 `PREV_ALLOC` 域，并将其加入空闲链表，使其成为一个新的空闲内存块。
- 返回分配的块指针。

> 在函数的实现过程中，你还需要思考：**如果空闲块多余的空间不足以构成一个空闲块，那么还能将其放回空闲链表吗？分配后，邻居块的 `PREV_ALLOC` 域应该如何修改？**

2.3.7 释放块

当调用 `mm_free()` 释放某个块后，如果该块相邻有其他的空闲块，则需要调用 `coalesce()` 函数，将这些块合并成一个大的空闲块，避免出现“假碎片”现象（多个小空闲块相邻，无法满足大块内存分配请求）。另外，上一节提到的修改 `PREV_ALLOC` 域的操作已经在 `mm_free()` 函数中实现。

```
/*
 * mm_free - Freeing a block does nothing.
 */
void mm_free(void *bp)
{
    size_t size = GET_SIZE(HDRP(bp));
    size_t prev_alloc = GET_PREV_ALLOC(HDRP(bp));
    void *head_next_bp = NULL;

    PUT(HDRP(bp), PACK(size, prev_alloc, 0));
    PUT(FTRP(bp), PACK(size, prev_alloc, 0));

     /*notify next_block, i am free*/
    head_next_bp = HDRP(NEXT_BLKP(bp));
    PUT(head_next_bp, PACK_PREV_ALLOC(GET(head_next_bp), 0));

    coalesce(bp);
}
```

> 思考：在 `mm_free()` 中实际上只修改了下一块**头部**信息中的 `PREV_ALLOC` 值，之后就调用了 `coalesce` 函数。为什么下一块**尾部**（若存在）中的 `PREV_ALLOC` 值在这里可以不做修改？（提示：和 `coalesce` 中合并空闲块的操作相关）

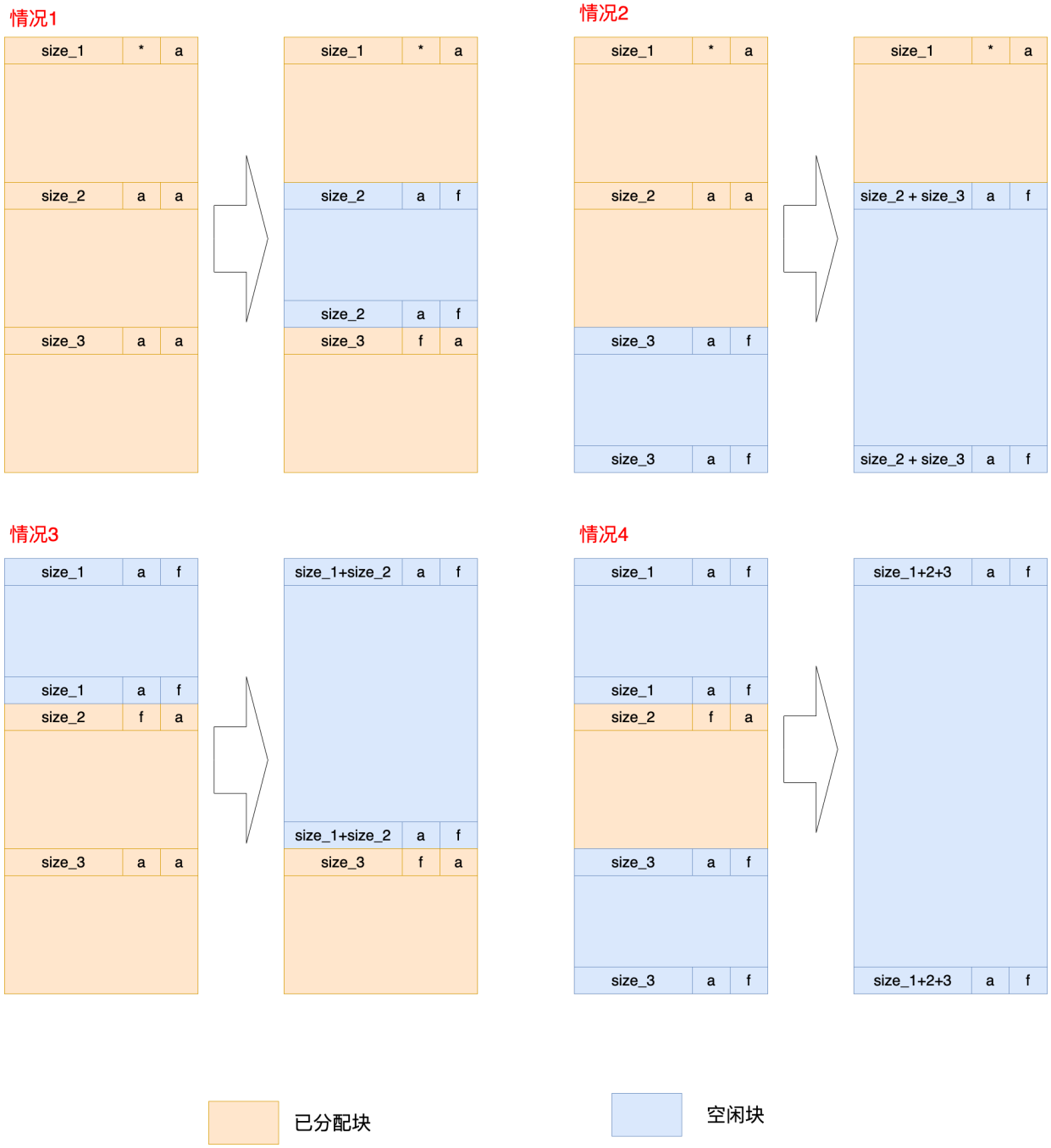
2.3.8 合并

上一节提到的合并操作在`coalesce()`函数中实现。同时，本次实验需要同学们完成`coalesce()`函数的编写。这里我们介绍一下合并的策略。

释放当前内存块时，根据相邻块的的分配状态，有如下四种不同情况：

1. 前面的块和后面的块都已分配；
2. 前面的块已分配，后面的块空闲；
3. 前面的块空闲，后面的块已分配；
4. 前后块都空闲。

以下为这四种情况的合并前后示意图，图示里中间大小为`size_2`的块即为当前释放的块：



图中块头部（尾部）中的信息从左到右依次为：块大小，前一块分配情况，当前块分配情况（a即alloc，表示已分配；f即free，表示空闲）

合并的流程如下所述：

- `coalesce()`函数首先从前一块的脚部后后一块的头部获取它们的分配状态。
- 然后根据前文所述的4种不同情况作相应处理，最后返回合并后的指针。

- 合并的过程中，要从空闲链表中删除合并前的空闲块并且插入合并后的空闲块。（`bp`一开始就不在空闲链表中，所以不需要删除它）
- 由于序言块和尾块的存在，不需要考虑边界条件，进行合并操作的块一定不会触及堆底和堆顶，因此不需要检查合并块位置。

> 提示：空闲块加入空闲链表，可以通过 `add_to_freelist()` 来完成；从空闲链表中删除块，可以通过 `delete_from_freelist()` 来完成。这两个函数已经被实现好，你无需改动。

```
static void *coalesce(void *bp)
{
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));
    /* 第一种情况 */
    if (prev_alloc && next_alloc)
    {
        /*  待补充 */
    }
    /* 第二种情况 */
    else if (prev_alloc && !next_alloc)
    {
        /*  待补充 */
    }
    /* 第三种情况 */
    else if (!prev_alloc && next_alloc)
    {
        /*  待补充 */
    }
    /* 第四种情况 */
    else
    {
        /*  待补充 */
    }
    return bp;
}
```

至此，一个显式链表管理方式的堆内存分配器实现完成。

第三部分 内存分配器的使用与测试

3.1 使用Makefile编译导出malloc动态库

上一次实验中，我们简单介绍了C标准库：它不仅完成了对基础系统调用的封装，还提供了许多必要功能服务的实现，比如动态内存分配。这样一来，平常我们编写C程序时，只需要声明头文件 `stdlib.h`，即可调用C标准库函数 `malloc()` 和 `free()` 进行堆内存的申请与释放，而无需关心 `malloc()` 和 `free()` 的底层实现。

现在，假设我们期望将自己设计的内存分配器进行推广使用，一个比较理想的方案是把分配器编译成为动态链接库的形式。之后，内存分配器的使用者只需要获取该动态库，即可在代码中使用 `mm_malloc()` 和 `mm_free()` 进行堆内存的动态分配/释放。

我们在 malloclab 目录下已经给出了所需的 Makefile，该Makefile无需改动：

```
CC = gcc -g -fPIC
CFLAGS = -Wall

all: libmem.so

libmem.so: memlib.o mm.o
    $(CC) $(CFLAGS) -shared -o libmem.so mm.o memlib.o

memlib.o: memlib.c memlib.h
mm.o: mm.c mm.h memlib.h

clean:
    rm -f *~ *.o libmem.so
```

其中：

- `libmem.so` 是整个编译的 target；
- `.so` 文件（shared object，共享目标）是linux系统下的动态链接库文件；
- 编译时，gcc的 `-shared` 参数表明产生共享库（动态库）。

在 malloclab 目录下执行 `make` 命令，即可编译得到动态链接库 `libmem.so`。

3.2 基于动态链接库的malloc库调用方法介绍

现在我们已经得到了内存分配器对应的动态链接库。之后，我们可以在自己的程序中引用头文件 `mm.h` 和 `memlib.h`，并调用函数 `mm_malloc` 和 `mm_free` 来进行内存的动态申请/释放，正如你之前声明头文件 `stdlib.h` 并调用 `malloc()` 和 `free()` 一样。我们提供的测试程序是 trace 目录下的 `workload.cc`，其中声明头文件、调用 `mm_malloc` / `mm_free` 的过程比较简单，不再赘述。

值得注意的是，malloc 和 free 的可执行程序存放在C标准库 libc 中，编译器会自动完成对 `libc.so` 的链接；而3.1中得到的动态库 `libmem.so` 无法被自动链接，需要我们显式地指定编译器完成对它的链接。同样在 trace 目录下，我们提供了编译+运行 `workload.cc` 的脚本 `run.sh`，内容如下：

```
#! /bin/bash

TASKPTH=$PWD
MALLOCPATH=/YOUR/PATH/TO/YOUR/malloclab/ # 需要修改为你的libmem.so所在目录
export LD_LIBRARY_PATH=$MALLOCPATH:$LD_LIBRARY_PATH
cd $MALLOCPATH; make clean; make
cd $TASKPATH
g++ workload.cc -o workload -I$MALLOCPATH -L$MALLOCPATH -lmem -lpthread
./workload
```

该脚本的含义如下：

- 第4行，请将你编译得到的 `libmem.so` 所在目录路径赋给变量 `MALLOCPATH`；
- 第5行将 `MALLOCPATH` （即 `libmem.so` 的路径）添加进环境变量 `LD_LIBRARY_PATH` 中。这是因为系统默认的动态链接库搜索路径为 `/lib` 和 `/usr/lib`，而我们想调用默认路径外的库 `libmem.so`，此时就需要在 `LD_LIBRARY_PATH` 中指明该库的搜索路径；
- 第3、6、7行是为了方便大家debug，每次跑workload之前都会重新编译一遍你的 `libmem.so`，这样你在修改libmem代码后直接在workload目录下运行 `sh run.sh` 就可以直接编译（我们提问的时候可能会问你这几句）；
- 第8行是编译 `workload.cc` 并运行可执行文件 workload 的命令：
  - 其中 g++ 命令的参数 `-I` 和 `-L` 分别指明了编译器优先寻找头文件和库文件的路径，这里我们指定的路径是 `$MALLOCPATH`，即优先在 `libmem.so` 所在目录下寻找；
  - 3.1中编译得到的动态链接库是 `libmem.so`，其中 `.so` 是后缀， `lib` 是动态链接库标准前缀，因此其实际名称可认为是mem；在 g++ 命令中，通过 `-lname` 的形式指定需要链接的动态库名称（name），因此我们链接 libmem.so 所用的参数为 `-lmem`。

为使脚本正常运行，你需要且只需要修改脚本第四行。

运行该脚本即可编译并运行 workload 测试程序，并且将数据输出到 trace 目录下的 mem_util.csv 文件中。

3.3 测试trace介绍

在介绍完如何将内存分配器引入到代码中，并编译运行后，我们来研究一下到底是一个什么样的程序在使用我们的分配器。

首先这是一个C++代码，而我们的分配器是C代码，所以这里也算是简单给出了一个C++调用C的例子，我们的C分配器在 `mm.h` 和 `memlib.h` 头文件中做了C++代码的适配，因为C++编译会 `#define __cplusplus`，所以这些头文件会增加 `extern "C"` 的前缀，从而可以在C++代码中使用。

```
#include <stdio.h>

#ifdef __cplusplus
extern "C" {
#endif

extern double get_utilization();
extern int mm_init (void);
extern void *mm_malloc (size_t size);
extern void mm_free (void *ptr);
extern void *mm_realloc(void *ptr, size_t size);
extern size_t user_malloc_size ;
extern size_t heap_size ;

#ifdef __cplusplus
}
```

```
#endif
```

workload中定义了一种核心的数据结构即 `workload_base`，是存储字符串数组头指针的一个数组 `addr`，即 `addr[i]` 保存了第 `i` 个字符串：

- 字符串可能会有16种随机长度值，从12到1024不等（见 `workload_size`）
- 每个字符串在生成时会调用 `malloc()`，并作随机字符串生成填充
- 上限有 `MAX_ITEMS` 个字符串生成。

workload首先会初始化 `workload_base`，调用 `workload_create()`，动态分配一个指针数组。然后进入一个插入-相邻交换-随机读-删除的循环：

- 插入为指针数组的每个空项malloc一个随机字符串，达到100%的指针数组使用率；
- 从第一个开始，每个都和后一个作指针交换；
- 作一个zipfian分布（带有冷热特征的随机访问模型，类似80%的读集中在20%的区域）的随机读；
- 最后free掉其中80%的项；
- 下一次循环再填充到100%。

如此循环，使得我们的内存分配器可以在反复的malloc和free中测试分配时间和内存的使用量。

这里的随机都是固定的随机，即所有人的随机数种子是一样的，所以你和其他同学跑出来的空间利用率应该是差不多的，运行时间可能要看下机器性能。

关于内存使用率和分配时间，你可能要改 `workload.cc` 中的部分代码：

- 每个loop的总运行时间已经在代码中给出，每次循环测一次，你可以修改 `workload_run()` 中的时间统计和输出格式代码为你自己想展示的形式。
- 内存使用率需要大家在分配器的代码中实现，我们定义好了两个全局变量，`user_malloc_size` 和 `heap_size`，和一个 `get_utilization()` 函数，这些是分配器代码中的内容，但是都可以workload.cc中直接引用。`user_malloc_size` 和 `heap_size` 分别代表用户申请的内存和已分配的内存，进一步的讲，用户申请的内存是用户当前需要多少的内存，已分配的内存是当前堆2的内存大小。`get_utilization()` 函数用于计算内存使用率，即用户在用的 `user_malloc_size` 内存分配器消耗掉的 `heap_size`。为什么不是堆1的size呢？因为堆1中没分给堆2的那部分，是没有实际分配的，随时可以回收，也就可以不算作消耗掉的空间。

> 你会发现有一个没有用到的 `monitor_run()`，这个函数的目的是每隔一段时间就统计一次内存使用率，比如以1s为单位，然后输出到文件中。如果你对内存使用率随时间变化情况感兴趣，可以使用这个函数得到数据，并作图分析。

3.4 内存使用率和分配时间

在跑通workload之后，我们就可以基于这个workload进行内存分配器的性能测试了，需要大家对于workload运行过程中内存分配器的内存使用情况和分配时间做统计，以比较best-fit和first-fit的区别。

3.4.1 用户申请量的统计

用户已经申请的内存量即 `user_malloc_size`，这个值可以在调用 `mm_malloc` 和 `mm_free` 时进行修改。需要注意的细节有：

- 我们可能一次分配出比申请量更大的空闲块，即 `find_fit_*`（注：`*` 表示任意内容都可匹配，这里意为两种fit算法中任意一种代入都成立）找到的空闲块有富余且无法被分割。此时可以认为额外分配的大小也在用户申请的大小中，以便与 `mm_free` 中直接释放掉整块内存对应；
- 已分配块的头部大小不应被计算在 `user_malloc_size` 中。分配器中元数据的内存开销也被视为降低了内存使用率。

3.4.2 分配器占用量的统计

分配器占用的内存量即 `heap_size` 。如前文所述，内存分配器的底层维护了两个堆结构（堆1和堆2）。我们在计算分配器的内存占用量时，只用考虑堆2占用的内存即可。这是因为：

- 堆1调用系统调用 `sbrk` 后，只是增加了其虚拟地址的空间；
- 而根据操作系统按需调页的实现原理，堆1比堆2多出来的这部分虚拟地址所在虚拟页，还没有被实际映射到物理页，因此实际上也就没有占用物理内存；
- 堆2中的内存已经被划分为分配块/空闲块，且写入有信息。因此堆2中的内存全部都是实际占用的内存，把它作为我们的 `heap_size` 是合理的。

3.4.3 分配时间

workload源码中给出了测量整个loop时间的代码实现，你可以参考这部分代码来测量一下insert的 `gen_random_string()` 部分中malloc所花费的时间。

第四部分 实验内容与评分标准

4.1 实验内容

- **补全** `mm.c` 和 `memlib.c` 文件中的**部分函数**。待补全的函数如下：
  - `void mem_init(void)`
    - 调用 `sbrk` ，初始化 `mem_start_brk` 、 `mem_brk` 、以及 `mem_max_addr` 。
    - 此处初始增长空间大小为 `MAX_HEAP` 。
  - `void *mem_sbrk(int incr)`
    - 模拟堆空间的增长。
    - 参数 `incr` 表示上层函数请求的空间大小。
    - 返回新分配空间的基址。
  - `static void *find_fit_first(size_t asize)`
    - 针对某个内存分配请求，该函数在空闲链表中执行首次适配搜索。
    - 参数 `asize` 表示请求块的大小。 返回值为满足要求的空闲块的地址。
    - 若返回值为 `NULL` ，表示当前堆块中没有满足要求的空闲块。
  - `static void *find_fit_best(size_t asize)`
    - 针对某个内存分配请求，该函数在空闲链表中执行首次适配搜索。
    - 参数 `asize` 表示请求块的大小。 返回值为满足要求的空闲块的地址。
    - 若返回值为 `NULL` ，表示当前堆块中没有满足要求的空闲块。
  - `static void place(void* bp, size_t asize)`
    - 在空闲块中分配一个块给用户。
    - 参数 `bp` 表示以定位的空闲块指针， `asize` 为待分配块的总空间要求。
    - 对于分配后的剩余空闲空间，需要转换为新的空闲块。
  - `static void*_ _coalesce(void* bp)`
    - 释放空闲块时，判断相邻块是否空闲，合并空闲块。
    - 根据相邻块的的分配状态，共有四种不同情况待补充，具体参见合并步骤这一节。
- 为你的内存分配器添加实时统计内存使用情况和分配时间的功能：
  - `user_malloc_size` —— 用户需要的内存量。
  - `heap_size` —— 内存分配器实际使用的内存量。
  - `double get_utilization()` ——返回 `user_malloc_size` 除以 `heap_size` 的值。

4.2 “实现内存分配器”部分评分标准(4')

本次实验满分10分，无实验报告。本次实验文档主要叙述了其中的“实现内存分配器”部分，满分为4分。

- 运行workload（已经写好）测试你的内存分配器，围绕时间和空间两点来分析first-fit和best-fit的性能差异：
  - 你需要向助教展示两种不同放置策略下的程序运行过程以及结果，并流畅说明代码内容与实现思路。
  - 你的程序应当能完整测试我们给出的workload，输出内存分配时间和内存使用率。
  - 你需要根据两种策略输出的内存分配时间和内存使用率，总结两种策略在时间和空间利用率上的优劣。
- 如果你能正确完成上面三个要求，那么得到满分4分。扣分项如下：
  - 无法逻辑通顺地讲解自己的代码 (-1')
  - 无法在初始堆大小5MB的设置下完成workload测试 (-1')
  - 最终的策略比较结论与真实情况不符 (-1')

V1.1 更新日志

文档更新

更新**2.1**节，补充对于 `sbrk()` 和 `brk()` 的简单介绍；

更新**2.3.2**节，修改序言块的长度为16字节、修改 `PACK_PREV_ALLOC` 与 `PACK` 两个宏定义；

更新**2.3.5**节与**2.3.6**节，补充管理各块头尾 `PREV_ALLOC` 域的相关提示；

更新**2.3.7**节，补充了一个思考问题；

更新**2.3.8**节，补充对于 `freelist` 相关增删操作的提示，更新了4种情况的图示、补充合并过程中的空闲链表管理提示；

更新**3.2**节，修改 `run.sh` 的语法错误。

代码更新

更新 `trace/run.sh`：修改 `TASKPATH` 变量名，去掉多余的空格；

更新 `malloclab/mm.c`：修改第28，29两行的宏定义。

实验3.2 Linux进程内存信息统计

版本v1.01（紧急更新）

> 本版本是临时版本，之后还会根据修改建议推出正式的修改版。

实验目的

- 学习如何添加Linux模块
- 学习使用sysfs虚拟文件系统
- 学习和掌握内存管理中的基本知识：遍历vma、页面冷热统计、页表遍历，dump 进程的数据段和代码段

实验环境

- OS: Ubuntu 18.04.4 LTS或者Ubuntu 20.04.4
- Linux内核版本: 5.9.0或者5.13.40

> 注：实验代码中所用的函数在不同的内核版本中的实现略有差别，为此我们提供了两种解决策略。第一种（**推荐**）是依据内核版本替换所使用的函数，详情见下文。一般来说，对本次实验而言，5以上的版本在操作上没有差别。第二种是固定所用的内核版本为5.9.0，固定系统内核版本方法请参考附录“编译替换内核版本”这一节。

实验时间安排

> 注：此处为实验发布时的安排计划，请以课程主页和课程群内最新公告为准

- 4.29晚实验课，讲解实验及检查
- 5.6晚实验课，检查实验
- 5.13晚实验课，检查实验
- 5.13日及之后，补检查实验

> 补检查分数照常给分，但会有标记记录此次检查未按时完成，标记会在最后综合分数时作为一种参考。

友情提示

- 本次实验以实验一为基础。一些步骤（如如何启动qemu运行Linux内核）不会在实验说明中详述。如果有不熟悉的步骤，请复习实验一。
- 在本次实验中，如果你写出的代码出现了数组越界，则会出现很多奇怪的错误（如段错误（Segmentation Fault）、输出其他数组的值或一段乱码（注意，烫烫烫是Visual C的特性，Linux下没有烫烫烫）、其他数组的值被改变等）。提问前请先排查是否出现了此类问题。
- 如果同学们遇到了问题，请先查询在线文档。在线文档地址：
  https://docs.qq.com/sheet/DR1dZTnFRTURHc051
- **建议同学们在拿到代码之后先尝试直接编译，如果遇到问题再尝试替换内核版本。一般来说Linux 5以上版本都无需替换内核版本。**

警告

本实验的一部分操作可能会对Linux系统产生一些可逆的破坏性。若内核模块的代码不正确，可能会导致：

现象	解决方案
内核模块无法卸载。	重启linux。
内核模块卸载后再重新载入模块时，命令一直卡住。	重启linux。
死循环导致Linux死机，或内核日志过长，或内存爆满。	重启linux。
虚拟机报错：客户机已禁用CPU。请关闭或重置虚拟机。	重启虚拟机。
Linux磁盘爆满，重启后卡在/dev/sda1:clean,.../...files,.../...blocks进不去图形界面。	按ctrl+alt+F3进入文本模式，删除一些无用内容以清理磁盘。
Ubuntu报错：“检测到系统程序出现问题”，但关闭提示窗口后系统仍能正常运行。	删除系统内核错误报告。**参考链接**

大部分情况均可通过表中所述解决方案解决。但考虑到重启系统、修复故障可能会花费一定时间，**建议使用快照保存系统状态**。但请注意，恢复快照前请保证自己已写的代码已被备份。

第一部分 内核模块介绍

1.1 什么是Linux内核模块

> 课程中我们学习到内核按照种类来划分主要分为**宏内核**、**微内核**和**混合内核**，具体定义和解释参考课程PPT。Linux 内核模块作为 Linux 内核的扩展手段，可以在运行时动态加载和卸载。

Linux是宏内核结构，所有内容都集成在一起， 效率很高，但可扩展性和可维护性较差，而模块化机制可弥补这一缺陷。所谓的模块化，就是各个部分以模块的形式进行组织，可以根据需要对指定的模块进行编译，然后安装到内核中即可。该种方式的优势是不需要预先将无用功能都编译进内核，尤其是各种驱动（通常来说，不同型号的硬件，对应的驱动不同）。如果为了顾全所有的硬件，而把所有的驱动都编译进内核，内核的体积会变得非常庞大，同时，在需要添加新硬件或者升级设备驱动时，必须重新构建内核。

可加载内核模块（Loadable Kernel Module，LKM）可以根据需要在系统运行时动态加载模块，扩充内核的功能，也可以在不需要时卸载模块。这样，对于模块的维护以及系统的使用就更加简单方便。用户可根据需要在Linux内核源码树之外开发并编译新的模块，进而修改内核功能，不必重新全部编译整个内核，只需要编译相应模块即可。同时，模块目标代码一旦被加载重定位到内核，其作用域和静态链接的代码完全等价。
目前内核模块存在两种加载方式：

- 静态加载：在内核启动过程中加载，例如：**KSM模块**
- 动态加载：在内核运行的过程中随时加载。例如：各种驱动代码。

1.2 内核模块的组成

> 基本结构：
> **头文件 -> 初始化函数 -> 清除函数 -> 引导内核的模块入口 -> 引导内核的模块出口 -> 模块许可证 -> 编译安装**

1.2.1 需要引用的基础头文件

> 注：因为内核编程和用户层编程所用的库函数不一样，故头文件也不同。
>
> - 内核头文件的位置：`/usr/src/linux-5.9/include/`。引用方法是`#include <linux/xxx.h>`。
> - 用户层头文件的位置:`/usr/include/`。引用方法是`#include <xxx.h>`。

编写内核代码所用到的头文件包含在内核代码树的`include/`及其子目录中。`module.h`，`kernel.h`，`init.h`，这三个头文件全部包含在`/include/linux/`中。这三个头文件以预处理指令的形式写在模块源代码的首部：

```
// 必备头函数
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
```

在编译模块源文件之前，由预处理程序对预处理指令进行预处理，然后由这些头文件的内容和其他部分一起组成一个完整的，可以用来编译的最后的源程序，然后由编译程序对该源程序正式进行编译，才得到目标程序。内核模块代码编译后得到目标文件后缀为.ko。

头文件`module.h`包含了对模块的结构定义以及模块的版本控制，可装载模块需要的大量符号和函数定义。

头文件`init.h`包含了两个非常重要的宏`__init`和`__exit`。

- 宏`__init`用于将一些函数标记为“初始化”函数。内核可以将此作为一个提示，即该函数仅在初始化阶段使用，并在初始化阶段之后释放使用的内存资源。**模块被装载之后**，模块装载器就会将初始化函数扔掉，这样可将该函数占用的资源释放出来。其使用方法详见2.2.2.

- 宏 `__exit` 的用法和 `__init` 一样，它的作用是标记该段代码仅用于模块卸载（编译器将把该函数放在特殊的ELF段中）。即被标记为 `__exit` 的函数只能在模块被卸载时调用。其使用方法详见2.2.3.

  头文件 `kernel.h` 包含了内核常用的API，如， `printk()` 就在 `kernel.h` 中声明。

1.2.2 初始化函数

初始化函数的定义如下：

```
static int __init name_function(void)
{
    /* 模块要实现的功能 */
    return 0;
}
module_init(name_function);
```

模块功能函数是在模块被装入内核后调用的，也就是在模块的代码被装入内核内存后，才调用模块功能函数。

> **注意：** `__init` 标记只是一个可选项，并不是写所有模块代码都要加 `__init` 。但是在测试我们自己写的模块时，最好加上 `__init` 。因为我们在写一个模块功能函数的时候，可能这个函数里面有定义的变量，当调用这个函数的时候，就要为变量分配内存空间，但注意，此时分配给变量的内存，是在内核空间分配的，也就是说分配的是内核内存。所以说如果只是想要测试一下模块的功能，并不需要让模块常驻内核内存，那就应该在执行完函数后，将当初分配给变量的内存释放。为了达到这个效果，只需要把这个函数标记为 `__init` 属性。

1.2.3 清除函数

清除函数的定义如下：

```
static void __exit name_function(void)
{
    /* 这里是清除代码*/
}
module_exit(name_function);
```

`__exit` 标记该段代码仅用于模块卸载，被标记为 `__exit` 的函数只能在模块被卸载或者系统关闭时调用。如果一个模块未定义清除函数，则内核不允许卸载该模块。

1.2.4 初始化入口点

源码定义如下：

```
/**
 * module_init() - driver initialization entry point
 * @x: function to be run at kernel boot time or module insertion
 *
 * module_init() will either be called during do_initcalls (if
 * builtin) or at module insertion time (if a module).  There can only
 * be one per module.
 */
#define module_init(x)  __initcall(x);
```

`module_init()` 是驱动程序初始化入口点。每一个普通的c程序都有一个 `main` 函数作为入口。而在内核中，则是 `module_init()` 来负责。在内核引导时运行的函数，或者在 `do_initcalls` 期间调用 `module_init()` ，或者在模块插入时(如果是模块)调用 `module_init()` 。每个模块只能有一个。

1.2.5 初始化出口点

源码定义如下：

```
/**
 * module_exit() - driver exit entry point
 * @x: function to be run when driver is removed
 *
 * module_exit() will wrap the driver clean-up code
 * with cleanup_module() when used with rmmod when
 * the driver is a module.  If the driver is statically
 * compiled into the kernel, module_exit() has no effect.
 * There can only be one per module.
 */
 # define module_exit(x)    __exitcall(x);
```

`module_exit()` 是驱动程序出口点。当驱动程序被删除时运行的函数。当驱动程序是一个模块时，`module_exit()` 将使用 `cleanup_module()` 包装驱动程序清理代码。如果驱动程序被静态编译到内核中，则 `module_exit()` 没有作用。每个模块只能有一个。

1.2.6 许可证

编写内核模块，需要添加模块许可证。如果没有添加模块许可证，会收到内核被污染的警告：`module license unspecified taints kernel`
内核被污染可能会导致驱动程序的一些系统调用无法使用。

```
// 该模块的LICENSE
MODULE_LICENSE("GPL");
// 该模块的作者
MODULE_AUTHOR("OS2022");
// 该模块的说明
MODULE_DESCRIPTION("test");
```

若添加模块许可证之后仍然报错，可能是因为编译时系统时间早于模块.c文件的修改时间。

1.2.7 参数

有时在安装模块的时候需要传递一些信息给模块，可以使用以下方式传递：

```
// 需要加上该头文件
#include <linux/moduleparam.h>

module_param(name, type, param);
// name为安装以及使用时的参数名字，type为类型，param为对应的sysfs的权限

module_param_string(name, string, len, param);
// name为外部名字，string为内部名字

module_param_array(name, type, nump, param)
// nump用于存放数组项数
```

使用的方式为，在安装模块的指定对应的参数及其值即可，如 `sudo insmod xxx.ko name=xxx`

1.2.8 如何使用内核未导出的变量

参考： **如何使用Linux内核中没有被导出的变量或函数?**
建议使用第三种方法，不需要重新编译内核，在后面的实验中，很多函数不能直接使用，如：
`page_referenced`， `follow_page`等。这里以 `follow_page` 函数为例：

```
// 声明在   /include/linux/mm.h
struct page *follow_page(struct vm_area_struct *vma, unsigned long address,
             unsigned int foll_flags);
```

使用方法：

```
// 全局为follow_page定义别名
typedef typeof(follow_page)* my_follow_page;

// 需要使用函数的地方
my_follow_page mfollow_page;

// 在命令行中使用 sudo cat /proc/kallsyms | grep follow_page
// 使用该方法获取follow_page的内核地址
// ffffffffa34739a0，内核重启地址可能发生变化
mfollow_page = (my_follow_page)0xffffffffa34739a0;
```

若输出的地址为0，是因为系统对内核地址做了保护。解决方案请参考 **链接1** 和 **链接2** 。

> 警告：每次重启系统之后，都要重新获取函数的内核地址，并在源码中修改地址，再重新编译。若不这么做，会产生段错误(Segmentation Fault)，这会导致模块无法卸载，需要重启系统。
>
> 注意：在不同的内核版本中不一定可以查询到follow_page函数的地址，有的版本只提供了follow_page_mask的地址（如Linux 4.9.263）。follow_page_mask是follow_page的具体实现，参数与follow_page略有差别，在实验代码中我们也提供了相关的调用方法。同学们在遇到无法找到follow_page函数的地址时，可以尝试使用follow_page_mask函数。

1.3 编译、加载内核模块

在本部分中，我们将向各位演示加载一个模块的全部流程。作为示例，我们使用附件提供的 `hello_world.c` 。该模块的功能是接收一个名为 `loop` 的参数，然后向内核日志中打印 `loop` 次"hello world!"。

1.3.1 基本知识

1. Linux内核日志

使用 `printk` 打印出的Linux内核日志存放在 `/var/log/` 中。
使用 `dmesg` 打印日志，使用 `sudo dmesg -C` 清空日志。

> 提示：
>
> 1. 每条内核日志的左侧记录有日志的打印时间。
> 2. 内核日志是行缓冲的。若打印的输出末尾没有\n，有可能打印不出来。 **参考链接**
> 3. 若执行 `dmesg` 之后系统卡死，可能是因为写出死循环，导致内核日志过长，输出时内存爆满，建议重启。

2. 模块相关指令

`insmod [模块文件][参数名称=参数值]`	**加载模块**
`lsmod`	显示已加载的模块
`modinfo 模块名`	查看模块信息
`rmmod 模块名`	卸载模块

> 参考资料：
>
> - **内核日志及printk结构分析**
> - **dmesg命令**

1.3.2 编辑模块代码、编译模块

操作流程：

1. 创建一个名为mymod的目录，并进入该目录。
2. 编写模块代码与Makefile。作为示例，这里使用我们提供的 `helloworld.c` 和 `Makefile` 。
3. 使用 `make` 编译（参考Lab1的2.3.17）。编译出的模块二进制文件名默认扩展名为 `.ko` 。在示例中，文件名是 `helloworld.ko` 。

> 注：
>
> 1. **linux内核模块签名**
> 2. 本次实验不涉及自己编写Makefile，可以直接使用助教提供的Makefile编译自己写的模块。Makefile中各指令的含义详见Makefile文件内的注释。当需要修改要编译的模块时，只需要修改obj-m后面的值。如，若要编译 `test.c` ，可将其改为 `obj-m += test.o` ，得到的模块二进制文件为 `test.ko` 。
> 3. 若不使用我们给的Makefile而直接用 `gcc helloworld.c` ，会报诸如“无法读取modules.order”的错。

1.3.3 加载模块

使用指令 `sudo insmod 模块二进制文件名 参数=值` 加载模块。在示例中，若想打印"hello world!"三次，加载指令应当为 `sudo insmod helloworld.ko loop=3` 。

> 提示：
>
> 1. 若 `insmod` 时报错"loading out-of-tree module taints kernel"的warning，可以忽略。
> 2. 若 `insmod` 时报错"insmod:ERROR:could not insert module xxx.ko.File existed"，说明该模块已被加载了，请先卸载模块再加载模块。**若无法卸载模块，请重启Linux。目前助教尚未获知除重启外方便地强制卸载模块的方法。**
> 3. 若双系统在 `insmod` 时报错"Operation not permitted"，可能是因为BIOS设置问题，请进入BIOS并将secure mode设置为disable。
> 4. 若模块加载后死机，可能是因为写出了死循环，建议重启。

1.3.4 检查模块是否正常工作

1. 使用 `lsmod | grep helloworld` 检查模块 `helloworld` 是否挂载。
2. 使用 `sudo dmesg` 查看内核日志，检查"hello world!"是否在内核日志中被打印了三次。

> 若 `lsmod` 时报错kmod_module_get_holders:could not open.....: no such file or directory ,建议在家目录下创建一个与（不含 `.c` 扩展名的）源码文件名相同的文件夹，将源码文件移动到该处目录下编译运行。

1.3.5 卸载模块

使用指令 `sudo rmmod helloworld` 卸载模块。

> 注意： 模块和终端不是绑定的，关掉一个终端不会卸载模块。如果卸载模块时卡死，即使你关掉终端，这个模块还是没有卸载成功。

第二部分 本实验涉及的其他内核功能

> 本部分所涉及内容已在代码框架中实现，各位无需更改。若不感兴趣，可以不了解其实现，只了解如何向sysfs内读写数据即可。

2.1 内核线程kthread

使用下列函数（宏）需要引入 `#include <kthread.h>` 。

2.1.1 创建并运行内核线程

```
kthread_run(threadfn, data, namefmt, ...)
```

- 参数 `threadfn` ：线程要执行的函数。其类型为 `int (*threadfn)(void *data)` ，即：该函数接受一个 `void *` 指针作为接受的参数，返回值为 `int` ；
- 参数 `data` ：线程函数接受的参数，其类型为 `void *` ；
- 参数 `namefmt` ：线程名。
- 返回值：线程对应的 `task_struct` 指针。

2.1.2 停止内核线程

```
int kthread_stop(struct task_struct *k);
```

- 参数 `k` ：内核线程的 `task_struct` 结构体。
- 返回值：内核函数的结束状态。

该函数会设置一个标志位，内核线程可以通过调用 `kthread_should_stop()` 函数查看自己是否应当停止

2.1.3 延时

将内核线程挂起等待，涉及等待队列、 `wait_event` 系列函数。可以参考： **等待队列**

在我们提供的代码中，为了将内核线程停止一段时间（类似于sleep），使用了 `schedule_timeout_interruptible` 函数。其用法参见： **内核线程** 。

2.1.4 系统中与时间相关的变量（宏）

变量（宏）名	含义
`HZ`	Linux内核每隔固定周期都会发生时钟中断， HZ 代表系统在1s中发生时钟中断的次数。
`TICK_NSEC` ，`TICK_NSEC`	`HZ` 的倒数，表示两次时钟中断的时间间隔（单位分别为纳秒、微秒）
`jiffies`	记录自系统启动以来发生的时钟中断总数。因为一秒内时钟中断的次数等于 `HZ` ，所以 `jiffies` 一秒内增加的值就是 `HZ` 。

Linux还提供了 `msecs_to_jiffies` 等函数实现不同时间单位的转换。使用上述变量和函数需要 `#include<linux/jiffies.h>` 。

2.2 sysfs文件系统

2.2.1 基本介绍

- sysfs 是 Linux 内核中一种基于内存的虚拟文件系统，用于向用户空间导出内核对象并且能对其进行读写。
- 一个**sysfs的设计原则是：一个属性文件只做一件事情**，sysfs 属性文件一般只有一个值，直接读取或者写入。
- sysfs被映射在 `/sys/` 目录下。

2.2.2 使用sysfs

下面假定我们要在 `/sys/kernel/mm/` 下创建一个名为 `ktest` 的属性集合，该属性集下有 `pid` 和 `func` 两个属性。这样，我们可以通过读取或修改 `/sys/kernel/mm/ktest/pid` 、 `/sys/kernel/mm/ktest/func` 两个文件来控制模块。

> 提示：使用下列函数需要 `#include <linux/sysfs.h>` 和 `#include <kobject.h>` 。

1. 设置属性的读方法(show)。每一个属性文件都要设置。当我们cat 这个属性文件时，会自动调用此方法。

参数 `kobj` 和 `attr` 表示挂载点（后面会提到）和属性列表， `buf` 表示映射到sysfs的内存地址。向这个内存地址内写入字符串，就能从sysfs内读出相同的字符串。下面的示例代码表示将一个全局变量 `pid` 打印到 `buf` 内：

```
static ssize_t pid_show(struct kobject *kobj, struct kobj_attribute *attr,
                        char *buf)
{
    return sprintf(buf, "%u\n", pid);
}
```

2. 设置属性的写方法(store)。每一个属性文件都要设置。当我们echo值到这个节点时，调用此方法。

```
static ssize_t pid_store(struct kobject *kobj,
                         struct kobj_attribute *attr,
                         const char *buf, size_t count){
    // do something
}
```

3. 创建属性。每一个属性都要执行相同操作。在提供的代码里，我们使用了一个 `KPAP_ATTR` 宏简化此步骤。

```
static struct kobj_attribute pid_attr = __ATTR(_name, 0644, pid_show, pid_store)
```

4. 设置属性集

```
// 创建属性集内的属性列表。
static struct attribute* ktest_attrs[] = {
    &pid_attr.attr,
    &func_attr.attr,
    NULL,
};

// 设置属性集名称及其中的属性。
static struct attribute_group ktest_attr_group = {
    .attrs = ktest_attrs,
    .name = "ktest",
};
```

5. 创建属性集

```
// 创建属性集，如果文件已存在，会报错。
// 前一个参数可以简单理解为挂载位置，后一个参数是属性集结构体。
int sysfs_create_group(struct kobject *kobj,
               const struct attribute_group *grp)
// 用例：mm_kobj的挂载点是/sys/kernel/mm/。
sysfs_create_group(mm_kobj, &ktest_attr_group)
```

6. 删除属性集

```
// 删除属性集，并删除属性文件。
// 前一个参数可以简单理解为挂载位置，后一个参数是属性集结构体。
void sysfs_remove_group(struct kobject *kobj,
            const struct attribute_group *grp)
// 用例：mm_kobj的挂载点是/sys/kernel/mm/。
sysfs_remove_group(mm_kobj, &ktest_attr_group)
```

2.2.3 sysfs文件系统简单示例

我们提供了 `sysfs_test.c` 作为示例文件，该示例通过sysfs文件系统控制模块输入，主要控制指标包括：模块是否开启、模块启动函数、模块扫描周期数及模块扫描时间间隔。

下面给出了示例代码的操作与输出示例：

> 在这一节的代码中，每行开头为$的，是输入的shell命令，剩下的行是shell的输出结果。

```
# 清空缓存
$ sudo dmesg -C
# 编译并加载模块
$ sudo make
$ sudo insmod sysfs_test.ko

# 查看模块是否加载成功
$ lsmod | grep sysfs_test

# 结果
sysfs_test                 16384  0

# 查看sysfs文件是否创建成功
$ ll /sys/kernel/mm/sysfs_test/

#结果
total 0
drwxr-xr-x 2 root root    0 4月  29 14:10 ./
drwxr-xr-x 8 root root    0 4月  29 14:10 ../
-rw-r--r-- 1 root root 4096 4月  29 14:10 cycle
-rw-r--r-- 1 root root 4096 4月  29 14:10 func
-rw-r--r-- 1 root root 4096 4月  29 14:10 run
-rw-r--r-- 1 root root 4096 4月  29 14:10 sleep_millisec

# 测试
# 开启print_hello
$ sudo sh -c "echo 1 > /sys/kernel/mm/sysfs_test/func"
$ sudo sh -c "echo 4 > /sys/kernel/mm/sysfs_test/cycle"
$ sudo sh -c "echo 1 > /sys/kernel/mm/sysfs_test/run"

#查看结果
$ dmesg
[52801.318328] hello world!
[52801.318342] hello world!
[52801.318343] hello world!
[52801.318344] hello world!

# 停止
$ sudo sh -c "echo 0 > /sys/kernel/mm/sysfs_test/run"

# 卸载模块并查看sysfs是否正常卸载
```

```
$ sudo rmmod sysfs_test
$ ll /sys/kernel/mm/

#结果
total 0
drwxr-xr-x  7 root root 0 May  3 05:52 ./
drwxr-xr-x 14 root root 0 May  3 05:52 ../
drwxr-xr-x  4 root root 0 May  3 05:52 hugepages/
drwxr-xr-x  2 root root 0 May  3 05:52 ksm/
drwxr-xr-x  2 root root 0 May  3 06:51 page_idle/
drwxr-xr-x  2 root root 0 May  3 06:51 swap/
drwxr-xr-x  3 root root 0 May  3 06:51 transparent_hugepage/
```

第三部分 内存管理

这一节中，我们给出一些内存管理中所必须的知识，帮助大家了解Linux系统内存管理子模块。主要涉及内存的基本管理单位页面（这里主要涉及4K大小的页面，暂时不考虑大页），进程虚拟空间的管理单位vma，系统如何判断页面非活跃以及页表的软件层次遍历。

> 前期准备工作（关闭透明大页）：

```
sudo sh -c "echo never > /sys/kernel/mm/transparent_hugepage/enabled"
sudo sh -c "echo never > /sys/kernel/mm/transparent_hugepage/defrag"
```

3.1 程序内存空间总述

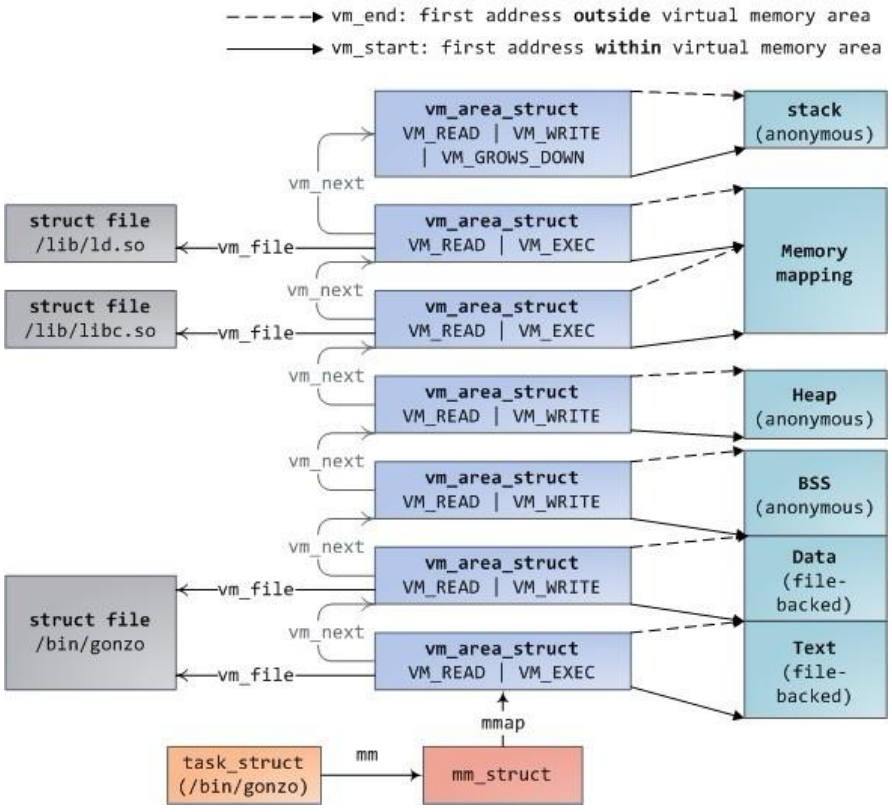
在Linux系统中，用户的内存空间可以分为以下几个层次：

- 段；
- VMA；
- 页。

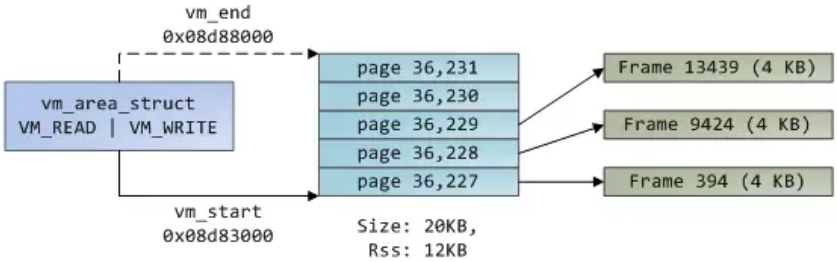
其中，如课程PPT(ch7_part1.pdf, P6)所述，用户的内存空间被划分为多个段，包括堆栈、数据段、代码段等。然而这种分段方式存在一些问题：

- 为节省内存空间，动态链接库采用的是延迟加载机制，即程序在运行的过程中在被调用的时候才会被加载。如果进程的内存空间简单地按段分配，那么无法实现更细粒度的内存管理；
- 当程序要将数据写入文件时，若直接写入磁盘，开销将会很大，需要在内存进行缓冲以加快读写速度。
- 内存是按需调页的。当一个程序申请若干内存空间时，不需要真的给它分配那么多空间，可以等程序访问对应内存时再申请物理页。因此，内核也需要管理程序已经申请的虚拟空间。

为此，如下图所述，Linux将整个用户地址空间分为若干个**VMA（Virtual Memory Area，虚拟内存空间）**，每一个段可能对应多个VMA。有的VMA与文件相关联，里面的页称为文件页；有的与文件无关，里面的页称为匿名页。



一个VMA内有多个页，如下图所示。



3.2 内存描述符

每个进程的 `task_struct` 内记录有一个内存描述符成员 `struct mm_struct *mm` ，记录该进程的全部内存信息。结构体 `mm_struct` 在 `include/linux/mm.h` 中定义。下面是该结构体的一些重要成员：

```
struct mm_struct {
    // 省略部分域展示
    /* 该进程的VMA链表 */
    struct vm_area_struct *mmap;
    /* 该进程的VMA红黑树根节点 */
    struct rb_node vm_rb;
    /* 代码段和数据段的起始地址和结束地址 */
    unsigned long start_code, end_code, start_data, end_data;
    /* 堆的起始、结束地址；用户态栈的起始地址 */
    unsigned long start_brk, brk, start_stack;
    /* 进程在应用程序启动时传递参数字符串的起始、结束地址；环境变量的起始、结束地址 */
    unsigned long arg_start, arg_end, env_start, env_end;
}
```

3.3 VMA

用户进程的虚拟地址空间包含了若干区域，这些区域的分布方式是特定于体系结构的，不过所有的方式都包含下列成分：

- 可执行文件的二进制代码，也就是程序的代码段；
- 存储全局变量的数据段；
- 用于保存局部变量和实现函数调用的栈；
- 环境变量和命令行参数；
- 程序使用的动态库的代码；
- 用于映射文件内容的区域。

由此可以看到进程的虚拟内存空间会被分成若干不同的区域，每个区域都有其相关的属性和用途，一个合法地址总是落在某个区域当中的，这些区域也不会重叠。在linux内核中，这样的区域被称之为虚拟内存区域(virtual memory areas)，简称vma。一个vma就是一块连续的线性地址空间的抽象，它拥有自身的权限(可读，可写，可执行等等)，每一个虚拟内存区域都由一个相关的 `struct vm_area_struct` 结构来描述：

```
struct vm_area_struct {
    struct mm_struct * vm_mm;   /* 所属的内存描述符 */
    unsigned long vm_start;     /* vma的起始地址 */
    unsigned long vm_end;       /* vma的结束地址 */

    /* 该vma的在一个进程的vma链表中的前驱vma和后驱vma指针，链表中的vma都是按地址来排序的*/
    struct vm_area_struct *vm_next, *vm_prev;

    struct rb_node vm_rb;       /* 红黑树中对应的节点 */
```

```
    /*anno_vma_node和annon_vma用于管理源自匿名映射的共享页*/
    struct list_head anon_vma_node; /* Serialized by anon_vma->lock */
    struct anon_vma *anon_vma;  /* Serialized by page_table_lock */

    /* Information about our backing store: */
    struct file * vm_file;          /* 映射的文件，没有则为NULL */
};
```

进程的若干个vma区域都得按一定的形式组织在一起，这些vma都包含在进程的内存描述符(`mm_struct`)中，它们主要以两种方式进行组织，一种是链表的方式（按虚拟地址大小的顺序）链接，对应于 `mm_struct` 中的 `mmap` 链表头，一种是红黑树方式，对应于 `mm_struct` 中的 `mm_rb` 根节点，和内核其他地方一样，链表用于遍历，红黑树用于查找。

3.4 页、页框

在Linux中，进程中的（逻辑）内存块被称为页（Page），物理内存中的内存块被称为页框（Frame）。一般来说，页与页框的大小相等。Linux 通常把 RAM 按每 4KB划分为一个页框。

利用页框机制有助于灵活分配内存地址，因为分配时不必要求必须有大块的连续内存，系统可以离散寻找空闲页凑出所需要的内存供进程使用。为了能够在保存连续页的同时，又能满足对启动内存需求的分配， 页框在分配上使用了**伙伴系统**。

在内核中，每一个物理页框对应一个**页描述符**(`struct page`)，所有的页描述符存放在一个**mem_map**（定义在 `mm/memory.c` ）线性数组之中。 每一个页框的页框号（ `physical address >> PAGE_SHIFT` ）是页描述符在 `mem_map` 数组的位置(下标)。

> 页框号（PFN）就是内存地址除以4K的值。 `PAGE_SHIFT` 定义在 `arch/所用架构/asm/page_types.h` 中，x86架构下其值为12。

在Linux内核中可以使用 `follow_page` 函数通过虚拟地址获取页描述符：

```
// include/linux/mm.h
// 获取某虚拟地址信息对应的页面结构体struct page
// 其中vma表示虚拟内存区域，address表示地址。
// 为获取一个页，请将foll_flags参数设置为宏FOLL_GET。
static inline struct page *follow_page(struct vm_area_struct *vma,
        unsigned long address, unsigned int foll_flags)
```

> 注意：
>
> 1. `follow_page` 函数未被内核导出，其使用方法请参考1.2.8。（我们提供的脚本已经帮你干了这件事）
> 2. 在Linux 4.20版本（不含）之前， `follow_page` 函数无法使用，需要使用 `follow_page_mask` 函数。其使用方法请参考 https://elixir.bootlin.com/linux/v4.19.240/source/include/linux/mm.h#L2587 ，即：定义一个无用的 `int` 型变量作为其最后一个参数。
> 3. `follow_page` 有时会返回NULL，这是正常的，原因是按需调页。**请务必使用宏 `IS_ERR_OR_NULL(page)` 检查page是否正确。**

页描述符定义在 `include/linux/mm.h` 中。下面是它的一些重要成员：

```
// 仅列出部分内容
struct page {
  /**
     * 一组标志，
     *    - 也对页框所在的管理区进行编号
     *    - 描述页框的状态：如 页被锁定，IO错误， 被访问，被修改位于活动页，位于slab 等。
     */
    page_flags_t flags;
  /**
```

```
     * 页框的引用计数。
     *   - 小于0表示没有人使用。
     *   - 大于0表示使用人数目
     */
    atomic_t _count;
  /**
     * 页框中的页表项数目（没有则为-1）
     *      -1:      表示没有页表项引用该页框。
     *      0:       表明页是非共享的。
     *      >0:      表示页是共享的。
     */
    atomic_t _mapcount;
  /**
     * 可用于正在使用页的内核成分（如在缓冲页的情况下，它是一个缓冲器头指针。）
     * 如果页是空闲的，则该字段由伙伴系统使用。
     * 当用于伙伴系统时，如果该页是一个2^k的空闲页块的第一个页，那么它的值就是k.
     * 这样，伙伴系统可以查找相邻的伙伴，以确定是否可以将空闲块合并成2^(k+1)大小的空闲块。
     */
    unsigned long private;
  /**
     * 当页被插入页高速缓存时使用或者当页属于匿名页时使用）。
     *      如果mapping字段为空，则该页属于交换高速缓存(swap cache)。
     *      如果mapping字段不为空，且最低位为1，表示该页为匿名页。同时该字段中存放的是指向anon_vma描述符的指
针。
     *      如果mapping字段不为空，且最低位为0，表示该页为映射页。同时该字段指向对应文件的address_space对象。
     */
    struct address_space *mapping;
  /**
     * 作为不同的含义被几种内核成分使用。
     *  - 在页磁盘映象或匿名区中表示存放在页框中的数据的位置。不是页内偏移量，而是该页面相对于文件起始位置，以页面
为大小的偏移量
     *  - 或者它存放在一个换出页标志符。表示所有者的地址空间中以页大小为单位的偏移量，也就是磁盘映像中页中数据的位
置 page->index是区域内的页索引或是页的线性地址除以PAGE_SIZE
     */
    pgoff_t index;
  struct list_head lru; // 包含页的最近最少使用的双向链表的指针， 用于伙伴系统。
#if defined(WANT_PAGE_VIRTUAL)
  /**
     * 如果进行了内存映射，就是内核虚拟地址。对存在高端内存的系统来说有意义。
     * 否则是NULL
     */
    void *virtual;
#endif /* WANT_PAGE_VIRTUAL */
}
```

3.5 页框管理

页框管理是Linux系统的基本功能，主要负责维护RAM资源，完成系统对RAM资源请求的分配。

页面回收算法： Linux 中的页面回收是基于 LRU（least recently used，即最近最少使用) 算法的。LRU 算法基于这样一个事实，过去一段时间内频繁使用的页面，在不久的将来很可能会被再次访问到。反过来说，已经很久没有访问过的页面在未来较短的时间内也不会被频繁访问到。因此，在物理内存不够用的情况下，这样的页面成为被换出的最佳候选者。

LRU 算法的基本原理很简单，为每个物理页面绑定一个计数器，用以标识该页面的访问频度。操作系统内核进行页面回收的时候就可以根据页面的计数器的值来确定要回收哪些页面。然而，在硬件上提供这种支持的体系结构很少，Linux 操作系统没有办法依靠这样一种页计数器去跟踪每个页面的访问情况，所以，Linux 在页表项中增加了一个 Accessed 位，当页面被访问到的时候，该位就会被硬件自动置位。该位被置位表示该页面还很年轻，不能被换出去。此后，在系统的运行过程中，该页面的年龄会被操作系统更改。在 Linux 中，相关的操作主要是基于两个 LRU 链表以及两个标识页面状态的标志符。

在 Linux 中，操作系统对 LRU 的实现主要是基于一对双向链表：active 链表和 inactive 链表，这两个链表是 Linux 操作系统进行页面回收所依赖的关键数据结构，每个内存区域都存在一对这样的链表。顾名思义，那些经常被访问的处于活跃状态的页面会被放在 active 链表上，而那些虽然可能关联到一个或者多个进程，但是并不经常使用的页面则会被放到 inactive 链表上。页面会在这两个双向链表中移动，操作系统会根据页面的活跃程度来判断应该把页面放到哪个链表上。页面可能会从 active 链表上被转移到 inactive 链表上，也可能从 inactive 链表上被转移到 active 链表上，但是，这种转移并不是每次页面访问都会发生，页面的这种转移发生的间隔有可能比较长。那些最近最少使用的页面会被逐个放到 inactive 链表的尾部。进行页面回收的时候，Linux 操作系统会从 inactive 链表的尾部开始进行回收。简单说，就是针对匿名页和文件页都拆分成一个活跃，一个不活跃的链表。Linux 引入了两个页面标志符 **PG_active** 和 **PG_referenced** 用于标识页面的活跃程度，从而决定如何在两个链表之间移动页面。PG_active 用于表示页面当前是否是活跃的，如果该位被置位，则表示该页面是活跃的。

PG_referenced 用于表示页面最近是否被访问过，每次页面被访问，该位都会被置位。Linux 必须同时使用这两个标志符来判断页面的活跃程度，假如只是用一个标志符，在页面被访问时，置位该标志符，之后该页面一直处于活跃状态，如果操作系统不清除该标志位，那么即使之后很长一段时间内该页面都没有或很少被访问过，该页面也还是处于活跃状态。为了能够有效清除该标志位，需要有定时器的支持以便于在超时时间之后该标志位可以自动被清除。然而，很多 Linux 支持的体系结构并不能提供这样的硬件支持，所以 Linux 中使用两个标志符来判断页面的活跃程度。

```
// include/linux/rmap.h
// 获取page引用计数，不为0意味着活跃
// 注意其中的memcg参数，在不同的内核版本中略有区别。
// 在Linux 5.9中可以memcg可以使用page->mem_cgroup得到
// 而在Linux 5.13中需要使用(struct mem_cgroup *)(page->memcg_data)得到
// 各位同学在使用本函数时请注意自己所用的内核版本。
int page_referenced(struct page *, int is_locked,
            struct mem_cgroup *memcg, unsigned long *vm_flags);
```

Linux 中实现在 LRU 链表之间移动页面的关键函数如下所示：

- `mark_page_accessed()`：当一个页面被访问时，该函数会被调用以修改 `PG_active` 和 `PG_referenced`。
- `page_referenced()`：当操作系统进行页面回收时，每扫描到一个页面，就会调用该函数设置页面的 `PG_referenced` 位。如果一个页面的 `PG_referenced` 位被置位，但是在一定时间内该页面没有被再次访问，那么该页面的 `PG_referenced` 位会被清除。
- `activate_page()`：该函数将页面放到 `active` 链表上去。
- `shrink_active_list()`：该函数将页面移动到 `inactive` 链表上去。

> 建议阅读资料：**Linux物理内存分配**

3.6 页表

用来将虚拟地址映射到物理地址的数据结构称为页表。实现两个地址空间的关联最容易的方式是使用数组：对虚拟地址空间中的每一页，都分配一个数组项，指向与之关联的页帧。但这样做会产生很多问题：

1. IA-32体系结构使用4KB大小的页，在虚拟地址空间为4GB的前提下，页表项数可达100万项，其大小为4MB。在64位体系结构下，这个问题会变得更加糟糕。
2. 每个进程都需要一套页表。如果系统的进程数量较多，会导致系统中大量的内存都用来保存页表。
3. 很多情况下进程用不满全部的4GB虚拟内存空间。直接分配一个较大的页表会产生空间浪费。

为此，Linux采用**多级页表**管理进程的内存空间。为同时支持适用于32位和64位的系统，Linux采用了通用的分页模型。Linux在2.6.10版本中采用了三级分页模型，从2.6.11开始采用了四级分页模型，从4.12开始采用了五级分页模型。

在多级页表模型下，页分为页目录表、页表和页三类：

- 页目录表是一个特殊的页，记录该地址对应的子目录或页表的物理内存基址；
- 页表也是一个特殊的页，记录该地址对应的页的物理内存基址。

单元	描述
页全局目录	Page Global Directory (PGD)
页上级目录	Page Upper Directory (PUD)
页中间目录	Page Middle Directory (PMD)
页表	Page Table (PT)
页内偏移	Page Offset

下图形象地展示了x86-64架构下四级页表的结构。在x86-64架构下，内存地址是64位的。每个4KB大小的页可以存放2^9个地址，所以在虚拟地址中，每级页表都分了9位。虽然每个页表（或页目录表）项长达64位，但因为其记录的是一个页（页表、页目录表本质也是一个页）的基址，而页地址永远4KB对齐，所以这64位地址的低12位永远为0。因为物理地址是64位（8Bytes）对齐的，所以在这低12位里面低3位也永远为0，中间的9位就是页表（或页目录表）项索引（可以通俗地理解为偏移量）。

这样的话，以下图为例，假如我们在PGD表内查到了该虚拟地址的PUD物理地址，可以用该PUD的物理基址+虚拟地址的30~38位左移3位 得到PUD表项的地址，这个地址里面存放的数据（也就是表项）就是该虚拟地址的PMD物理基址。

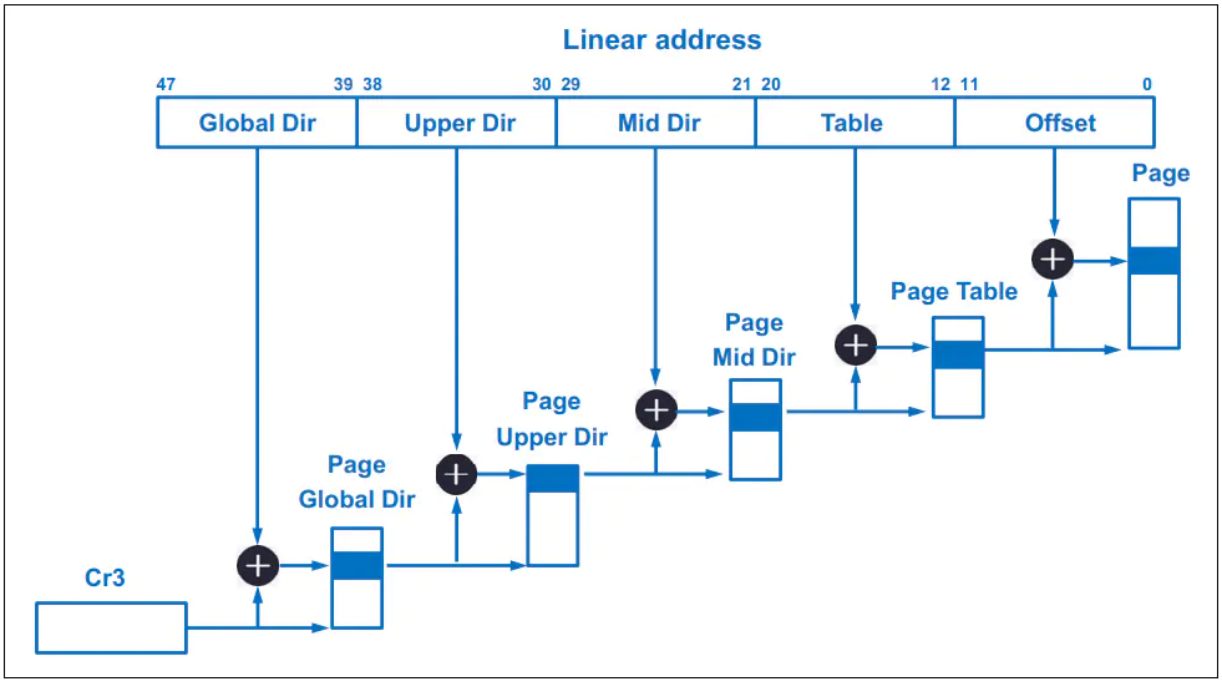


Figure 1　Structure of the 4-level paging

解释上图：Linux内核通过如下流程完成虚拟地址到物理地址的转换（以64位机器、四级页表为例）：

1. 从CR3寄存器中（也可以从 `mm_struct->pgd_t` ）读取PGD（页全局目录）所在物理页面的物理地址（即所谓的页目录基址）。
2. 获取内存地址的PGD（页全局目录）项索引（即：虚拟地址的39~47位），然后查PGD表，找到该地址对应的PGD表项。该表项即为该虚拟地址所属的页上级目录（PUD）的物理基址。
3. 获取内存地址的PUD（页上级目录）项索引（即：虚拟地址的30~38位），然后查上步获取的PUD表，找到该地址对应的PMD表项。该表项即为该虚拟地址所属的页中间目录（PMD）的物理基址。

4. 获取内存地址的PMD（页中间目录）项索引（即：虚拟地址的21~29位），然后查上步获取的PMD表，找到该地址对应的PMD表项。该表项即为该虚拟地址所属的页表（PT）的物理基址。
5. 获取内存地址的页表项（PTE）索引（即：虚拟地址的12~20位），然后查上步获取的页表，找到该物理页面（即页框）的物理基址（即页框编号，PFN）。
6. 获取页面的物理基址后，加上0~11位的offset，就可以得到最终的物理地址，进而访存。

页表转换宏：

```
/*描述各级页表中的页表项*/
typedef struct { pteval_t pte; } pte_t;
typedef struct { pmdval_t pmd; } pmd_t;
typedef struct { pudval_t pud; } pud_t;
typedef struct { pgdval_t pgd; } pgd_t;

/*  将页表项类型转换成无符号类型 */
#define pte_val(x)  ((x).pte)
#define pmd_val(x)  ((x).pmd)
#define pud_val(x)  ((x).pud)
#define pgd_val(x)  ((x).pgd)

/*  将无符号类型转换成页表项类型 */
#define __pte(x)    ((pte_t) { (x) } )
#define __pmd(x)    ((pmd_t) { (x) } )
#define __pud(x)    ((pud_t) { (x) } )
#define __pgd(x)    ((pgd_t) { (x) } )

/* 获取页表项的索引值 */
#define pgd_index(addr)     (((addr) >> PGDIR_SHIFT) & (PTRS_PER_PGD - 1))
#define pud_index(addr)     (((addr) >> PUD_SHIFT) & (PTRS_PER_PUD - 1))
#define pmd_index(addr)     (((addr) >> PMD_SHIFT) & (PTRS_PER_PMD - 1))
#define pte_index(addr)     (((addr) >> PAGE_SHIFT) & (PTRS_PER_PTE - 1))

/*  获取页表中entry的偏移值 */
#define pgd_offset(mm, addr)    (pgd_offset_raw((mm)->pgd, (addr)))
#define pgd_offset_k(addr)  pgd_offset(&init_mm, addr)
#define pud_offset_phys(dir, addr)  (pgd_page_paddr(*(dir)) + pud_index(addr) * sizeof(pud_t))
#define pud_offset(dir, addr)       ((pud_t *)__va(pud_offset_phys((dir), (addr))))
#define pmd_offset_phys(dir, addr)  (pud_page_paddr(*(dir)) + pmd_index(addr) * sizeof(pmd_t))
#define pmd_offset(dir, addr)       ((pmd_t *)__va(pmd_offset_phys((dir), (addr))))
#define pte_offset_phys(dir,addr)   (pmd_page_paddr(READ_ONCE(*(dir))) + pte_index(addr) *
sizeof(pte_t))
#define pte_offset_kernel(dir,addr) ((pte_t *)__va(pte_offset_phys((dir), (addr))))
#define pte_page(pte)   pfn_to_page(pte_pfn(pte))

/* 检查页（目录）表项是否为空 */
static inline int pgd_none(pgd_t pgd);
static inline int pud_none(pud_t pud);
static inline int pmd_none(pmd_t pmd);
static inline int pte_none(pte_t pte);

/* 检查页（目录）表项是否可用 */
static inline int pgd_bad(pgd_t pgd);
static inline int pud_bad(pud_t pud);
static inline int pmd_bad(pmd_t pmd);
static inline int pte_bad(pte_t pte);
```

> 提示：Linux在4.12开始采用了五级分页模型，在pgd和pud之间加了一个p4d。因此，`pud_offset` 接受的第一个参数类型应当为 `p4d_t *` 。但我们的程序实际用不到这一级，所有的虚拟地址这段都是0，所以可以将获得的 `pgd_t *` 类型变量强制转换为 `p4d_t *` 。

请通过自行阅读源码，或阅读下列参考资料了解各转换宏的使用方法。

参考资料：

- **Linux kernel内存管理**
- **Linux分页机制**
- **Linux中的页表实现**
- **深入理解计算机系统-之-内存寻址（五）-页式存储管理**，详细讲解了传统的页式存储管理机制；
- **深入理解计算机系统-之-内存寻址（六）-linux中的分页机制**，详细的讲解了Linux内核分页机制的实现机制。

第四部分 实验流程及要求

实验的第二部分分为以下几个部分：

- 统计进程VMA数目
- 页面冷热统计
- 页表遍历
- 进程信息获取

本次实验需要使用sysfs文件系统实现自动化模块ktest，主要是指通过sysfs完成模块参数的输入和输出。要求：

文件	功能
/sys/kernel/mm/ktest/func	选择功能
/sys/kernel/mm/ktest/sleep_millisecs	扫描时间间隔
/sys/kernel/mm/ktest/run	是否开启模块功能
/sys/kernel/mm/ktest/pid	选择哪个进程
/sys/kernel/mm/ktest/vma	对应进程的vma总数

> 注：
>
> 1. 提示：你们需要完成我们提供的 `ktest.c` 中的代码填空。
> 2. 为了简化代码，本次实验均在系统仅存在小页的环境下完成，需关闭系统透明大页设置。（参考本文档第三部分的开头内容）
> 3. 在测试时，下文的“测试程序”特指使用我们已经提供了的 `workload.cc` 编译运行产生的进程。

4.0 测试程序、代码框架

4.0.1测试程序

我们提供了workload.cc程序，同学们可以编译并运行（./run.sh）该应用程序，然后通过完成4.1~4.5来监控该进程的相关信息。该测试程序与lab3.1提供的测试程序**不同**：这里直接用 `malloc` 分配空间，而不是利用内存分配器。

> 注：编译workload.cc的方式为： `g++ -std=c++11 workload.cc -o [exe file name] -lpthread`

4.0.2 代码框架

为完成本实验，你需要完成 `ktest.c` 的代码填空。

我们已经在 `ktest.c` 中实现了代码框架，你们无需修改代码框架。已实现的功能有：

- sysfs的输入与输出。程序会自动地从sysfs读取内容并存放入全局变量内，也会自动地将需要的变量输出到sysfs中。
- 使用内核线程实现了每5秒扫描一次sysfs，读取pid、func和sleep_milliseconds。程序会自动地根据读取到的pid找到进程的 `task_struct` 结构体并放入 `my_task_info` 内。
    - 需要指定pid为4.0.1所述的测试程序。但我们已经在4.7内提供了一个自动化脚本，它会自动向sysfs内写入pid。所以这部分无需操作。
    - 需要指定func为下述要完成的实验功能。但我们已经在4.7内提供了一个自动化脚本，它会接受参数并自动向sysfs内写入func并完成测试。所以这部分也无需操作。
    - sleep_milliseconds使用模块程序默认的5秒即可。这个参数代表我们的程序每5秒扫描一次并输出一次。
- 代码框架提供了大量注释，如有需要可以阅读。

4.0.3 如何进行简单的编译

1. 解压提供的"lab3 part2-进程内存信息统计.zip"。
2. 编译时文件路径不能带空格。所以请重命名解压缩后的文件夹。
3. 为 `run_expr.sh` 添加执行权限： `sudo chmod +x run_expr.sh`
4. 执行自动化脚本进行测试运行： `sudo ./run_expr.sh 0` ，其中0是参数，选择使用哪个func。参数为0表示测试func，每5秒 `printk` 一次helloworld，你可以另开一个终端并通过1.3.1介绍的方法读取。
5. 使用1.3.5介绍的方法卸载模块，然后用 `Ctrl + C` 退出脚本。

4.1 func = 1： 每隔5s统计测试程序的vma数量

1. 需要的文件

- 控制： /sys/kernel/mm/ktest/func：选择功能
- 控制： /sys/kernel/mm/ktest/pid：选择进程
- 输出： /sys/kernel/mm/ktest/vma：对应进程的VMA数量。预期值在39左右。你可以通过 `cat /sys/kernel/mm/ktest/vma` 获取输出。

2. 任务提示

- 遍历vma需要使用到 `mm_struct` 描述符中的 `mmap` 成员；

4.2 func = 2： 每隔5s统计测试程序的页面冷热信息

1. 需要的文件

- 控制：/sys/kernel/mm/ktest/func：选择功能
- 控制： /sys/kernel/mm/ktest/pid：选择进程
- 输出： 输出页面冷热信息到文件中，方便进行图形绘制工作。**输出要求：输出页面的十进制物理地址，且输出之间用“,”（英文逗号）连接，每扫描一次所有页面执行一次换行，以便直接使用我们提供的画图脚本进行画图。**

2. 任务提示

- 流程：遍历vma，再遍历vma中的虚拟地址（间隔： `PAGE_SIZE` -- 4K），使用 `follow_page` 函数通过虚拟地址获取页面结构体 `struct page` ，然后通过 `page_referenced` 函数判断页面最近是否被引用过。内核态进行文件读写可以参考 **这里** 。

3. 按照如下命令来安装画图脚本所用的库：

```
sudo apt install python3
sudo apt install python3-pip
python3 -m pip install matplotlib
```

4.3 func = 3： 遍历测试程序的页表，并打印所有页面物理号

1. 需要的文件

- 控制：/sys/kernel/mm/ktest/func：选择功能；
- 控制： /sys/kernel/mm/ktest/pid：选择进程
- 输出： 输出到文件中，虚拟地址和物理页号对应。

> **注：这里需要遍历多级页表得到物理地址，不可以使用page_to_pfn(struct page)函数直接得到一个page对应的物理地址，但是可以使用page_to_pfn来检查自己得到的结果是否正确。**

2. 任务提示

- 流程：遍历vma，再遍历vma中的虚拟地址（间隔：PAGE_SIZE -- 4K），通过虚拟地址进行解析，得到对应的物理页号（详见3.6）。

4.4 func = 4： dump测试程序的数据段

1. 需要的文件

- 控制：/sys/kernel/mm/ktest/func：选择功能；
- 控制： /sys/kernel/mm/ktest/pid：选择进程；
- 输出： 输出到文件中。

2. 任务提示

- 流程：首先根据 pid 获取对应的 `mm_struct`，获得里面的 `start_data` 和 `end_data` 地址，之后将对应的数据通过 `kernel_write` 写入到文件中即可。 `kernel_write` 函数的用法：

```
// 将 buf[0..count - 1] 里面的内容输出到 file 对应的文件中，
// 其中，file 对应的文件初始偏移是 offset
int kernel_write(struct file *file, char *buf, size_t count, loff_t *offset);
```

- `mm_struct` 中的成员变量已在3.2讲解。

4.5 func = 5： dump测试程序的代码段

1. 需要的文件

- 控制：/sys/kernel/mm/ktest/func：选择功能；
- 控制： /sys/kernel/mm/ktest/pid：选择进程；
- 输出： 输出到文件中。

2. 任务提示

- 流程：首先根据 pid 获取对应的 `mm_struct`，获得里面的 `start_code` 和 `end_code` 地址，之后将对应的数据通过 `kernel_write` 写入到文件中即可。

4.6 任务评分规则

满分6分，包括：

1. 第一任务(func = 1)完成，得1分。
2. 第二任务(func = 2)完成，得1分。
3. 第三任务(func = 3)完成，得1分。
4. 第四任务(func = 4)完成，得1分。
5. 第五任务(func = 5)完成，得1分。
6. 代码讲解，得1分。

> 注：实验检查时会要求先确认运行结果，再简单解释一下如何实现。

参考文献：

1. **linux进程地址空间--vma的基本操作**
2. **Linux内存管理篇之页框管理**

4.7 自动化运行实验脚本

为了方便学生开展实验，我们提供了自动化运行代码的脚本。其工作过程如下：

1. 禁用大页。
2. 更新代码中涉及 `follow_page` 和 `page_referenced` 地址的相应行，这样即使上述两个函数的地址发生了变化，也能及时的进行地址修改从而避免出错。
3. 加载编写的内核模块。
4. 编译workload.cc。
5. 运行workload。
6. 运行编写的内核模块。
7. 等待workload运行结束。
8. 画图或者保持生成的数据段/代码段等结果。

自动化运行实验脚本代码保存在压缩包内 `run_expr.sh` 。

4.8 实验材料的目录结构

- lab2.zip
    - trace目录：测试程序
    - helloworld目录：一个简易的打印helloworld的模块代码。
    - sysfs目录：2.2.3描述的一个简易的sysfs模块代码。
    - run_expr.sh：4.7描述的自动化运行脚本。
    - ktest.c：本实验的代码框架。
    - draw.py：4.2的画图脚本。
    - active_page_log.txt：画图脚本的参考标准输入。你们需要能够跑出自己的active_page_log.txt。受采样时间的影响，自己跑出log的可以和示例不一致，但画出的图应该与示例接近。直接运行 `python3 draw.py` 就可以读取log并输出图像。
    - Makefile。

附1 实验第一部分选做

在实验第一部分中，我们实现了一个64位内存分配器，并且对其内存使用效率进行了测试与分析。我们会发现在调用过free后，内存的使用率会降到一个很低的水平，这是因为虽然用户调用free还给了内存分配器，但是内存分配器并没有还给内核，依然处于占用状态。在实际的分配器中，对**空闲内存的回收**是提高内存使用率、在必要时将空闲内存归还给内核的有效手段。因此，我们给出一个相关的选做题目如下：

实现分配器的内存回收机制，具体要求：

- 该回收机制必须支持按页回收分配器中空闲内存
    - 回收可以发生在任何空闲页上（而非仅堆尾的空闲页），因此用 `sbrk()` 构造的连续地址空间堆无法满足该要求。你需要将 `sbrk()` 获取内存的方式改为使用 `mmap()` ，然后使用 `munmap()` 以4KB页为单位对齐回收（参考第一部分文档中1.1节）；
    - 回收的粒度是4KB页，也即只有地址按4KB对齐的整块页全部空闲时，你才可以将这块页归还给操作系统。也可以简单理解为， `mmap()` 返回地址开始，每4KB是一个页，然后以页为单位去判断是否可以回收。
- 该回收机制必须支持异步回收：
    - 最终需要呈现出一个内存回收调用接口（比如一个名为 `garbage_collection()` 的函数），这个接口会去扫描空闲链表，查找可回收页并通过适当的系统调用函数进行回收。之后修改负责分配器初始化的 `mm_init()` 函数，在其中额外启动一个内存回收守护线程，该线程每隔固定时间调用一次上述内存回收接口；
    - 该内存回收守护线程在异步执行回收任务的过程中，可能与 `mm_malloc()` 或 `mm_free()` 产生竞争。你需要通过实现页粒度的锁管理，确保分配器的申请/释放功能，与内存回收功能可以在保证一定效率的前提下并发执行（你不能对整个空闲链表全局加锁，因为这样产生的效果与同步回收几乎没有区别）
- 该回收机制需要额外测试：
    - 修改原先内存使用率的统计方法，以确保实现回收机制后，仍能统计到正确的内存使用率；
    - 定量分析出加入回收机制后内存使用率的提升；
    - 展示内存分配器在时间性能上的变化（给出测试结果即可）。

实现上述全部功能和要求，并在实验检查时，向助教流畅讲述代码逻辑及展示测试效果，**可以得到额外的1'加分**。本选做题目较为开放，因此无具体评分细则，由助教在检查时酌情判断。

附2 编译替换内核版本

若本机的Linux版本不能满足实验需要，可以考虑更改Linux内核版本（**这在本次实验中并不是必须的**）。下面是修改内核版本的流程：

注意，在替换内核版本之前，请做好文件备份。虚拟机可以通过生成系统快照的方式进行快捷备份。

附2.1 为什么要替换内核版本

- 为什么要替换内核版本?

模块在编译时会从你的系统中寻找linux头文件，而不同版本的内核头文件不一定相同。模块在运行时会从你的系统中寻找依赖库，而不同版本的内核依赖库也不一定相同。所以，在一个版本下编译运行成功的模块不一定能在其他版本下正常运行。所以，为了方便检查，我们将本次实验的内核版本设置为 5.9。当然，你也可以在其他内核下进行实验，不过不保证助教给的代码可以在其他内核上正常运行。

- 那为什么不用之前已经配置好的qemu环境?

这是因为需要在本地环境编译打包进_install目录下，操作会很麻烦。所以建议本地操作。

附2.2 查看本系统内核版本

```
$ uname -r
5.4.0-42-generic
```

附2.3 编译本实验所用内核版本

该部分过程已经在实验一中讲解，此处只是简单列出操作，编译环境安装步骤在此省略。

1. 进入 `/usr/src` 目录（建议内核代码放在此处）
2. 使用 `wget` 下载实验所需特定内核代码。内核代码地址链接：
   https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.9.tar.xz

> 校内镜像链接： https://mirrors.ustc.edu.cn/kernel.org/linux/kernel/v5.x/linux-5.9.tar.xz

3. 使用 `tar` 指令解压。解压 `.xz` 文件的参数为 `-Jxvf` 。
4. 安装依赖库：使用包管理器安装 `flex` 和 `bison` 两个包。
5. 进入源代码根目录（ `cd /usr/src/linux-5.9` ）
6. 编译配置选择

如果直接 `sudo make menuconfig` 不修改内核编译配置，直接save，然后exit（具体配置含义见附录），会导致编译时间过长（兼容各种环境），且生成的的内核代码占用磁盘空间将近20G。为避免此情况的发生，可以将原始内核配置移动到需编译内核源码目录下，操作如下：

```
sudo cp /boot/config-`uname -r` /usr/src/linux-5.9/.config
sudo make clean
sudo make oldconfig // 出现提示一直Enter就可以了
sudo make localmodconfig // 出现提示一直Enter就可以了
```

7. 编译内核：sudo make -j $((`nproc` -1))
8. 安装内核模块： `sudo make modules_install`
9. 安装内核： `sudo make install`

执行完成后，可以检查一下 `/boot` 目录下是否生成了以下文件：

- config-5.9.0
- initrd.img-5.9.0
- System.map-5.9.0
- vmlinuz-5.9.0

若没有生成以上文件，请检查 `make` 操作后输出的相关信息并重新编译。

附2.4 配置 GRUB 引导文件

1. 修改默认的 grub 配置文件

```
sudo vim /etc/default/grub
```

2. 找到对应的两行配置文件并且修改为如下内容：

```
# GRUB_TIMEOUT_STYLE=hidden
GRUB_TIMEOUT=3
```

3. 更新内核目录项

```
sudo update-grub
```

4. 重启系统后选择 `Advanced options for Ubuntu`，并且选择 5.9 内核启动，系统启动成功后，请使用 `uname -r` 命令检查一下是否成功切换到 5.9 内核。

附3 更新日志

- 增加了如何删除系统内核错误报告的链接；
- 增加了“一般来说Linux 5以上版本都无需替换内核版本”的提示；
- 1.3.3添加了应对insmod时报错Operation not permitted的解决方案；
- 修正2.2.2.6的笔误，该处笔误将remove写成了create；
- 3.1增加了文件页、匿名页的介绍；将“一个VMA内有多个页表”修改为“一个VMA内有多个页”；
- 将3.4的“ `follow_page` 有时会返回NULL”提至外面提醒，并添加“使用 `IS_ERR_OR_NULL` 检查”的提示；
- 将3.5对 `kernel_write` 函数的介绍移动到4.4；
- 重写3.6；
- 在4.0增加了代码框架的介绍；
- 4.1增加提示：可以通过 `cat /sys/kernel/mm/ktest/vma` 获取输出。

实验四 FAT文件系统的实现 Part1

> 本文档对一部分内容进行了修正。修正日志详见Part2文档的第一部分。

实验目标

- 熟悉文件系统的基本功能与工作原理（理论基础）
- 熟悉FAT16的存储结构，利用FUSE实现一个FAT文件系统：
  - 文件目录与文件的读（只读文件系统，基础）
  - 文件的创建与删除（基础）
  - 文件目录的创建与删除（进阶）
  - 文件的写（进阶）
- 在实现fat16文件系统的基础上，引入文件系统的重要问题研究（进阶，选做）
  - 如何做文件系统的错误修复——三备份与纠删码
  - 如何做文件系统的日志恢复——元数据日志

实验环境

- VMware / VirtualBox
- OS: Ubuntu 20.04.4 LTS

实验时间安排

注：此处为实验发布时的安排计划，请以课程主页和课程群内最新公告为准

- 5月13日晚实验课，发布、讲解基础部分及检查实验
- 5月20日晚实验课，发布、讲解进阶部分及检查实验
- 5月27日晚实验课，检查实验
- 6月3日晚实验课，实验补检查

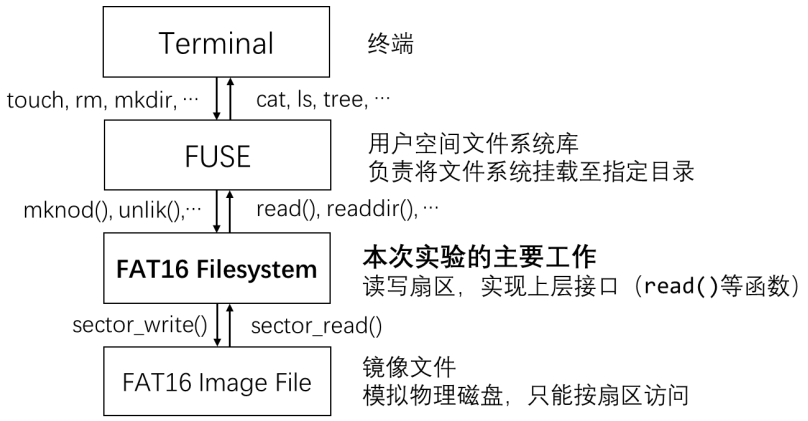
友情提示

- **本次实验总工作量较大，请尽早开始实验。**
- **本次实验有600余行注释，以解决实验文档对于代码实现的提示量不足的问题，代码实现过程中务必注意看注释！看注释！看注释！**
- **实验多个任务难度依次递增，靠前的任务代码提示较为详尽，你可以相对容易地获得这部分分数。**
- **实验多个任务之间较为独立，你也可以选择只完成靠后的任务。**
- **本实验文档较为详尽，是为了尽可能提前解决大家实验中可能遇到的问题。你并不一定需要完全阅读所有实验文档才能完成实验、获得分数。**

第一部分 实验环境

1.1 实验框架概述

在本次实验中，我们将实现简单的FAT16文件系统，来读写一个FAT16镜像文件。本次实验的大体框架如下图。



首先，实验文件中提供了一个已经格式化为FAT16格式的镜像 `fat16_test.img` ，这个文件就是本次实验的“硬盘”，这块“硬盘”中预先存储了一些目录和文件，需要我们通过实现的文件系统去读取它们。

为了隐藏文件系统与操作系统的交互细节，将实验重点聚焦在文件系统的格式和实现上。本实验采用FUSE库以便在用户空间实现文件系统。FUSE库能够帮助处理文件系统和内核交互的部分，将终端命令行对文件系统的访问，转化成对 `fuse` 提供的文件系统接口的调用。所以**本次实验中，只需要实现相应的 文件系统接口** ，就可以在终端中像访问其它目录一样访问实验中提供的“磁盘”。

在实验提供的 `simple_fat16.c` 文件中，定义了所有需要完成的接口，包括 `readdir` 、 `read` 、 `mknod` 等共8个函数，这些函数的具体含义将在实验内容部分详细的进行介绍。同学们只需要根据FAT16文件系统的结构，实验要求和实验代码中的提示，**补全 `simple_fat16.c` 中对应的函数**，实现相应的文件系统功能，即可获得相应的分数。

值得注意的是，磁盘是一种块设备，任何对磁盘的访问都是以扇区为单位的。为了模拟这种访问，实验提供的代码中，抽象了 `sector_read` 和 `sector_write` 函数，分别用于模拟读扇区和写扇区。因此，本次实验**所有对镜像文件的读写操作，最终都是需要通过调用 `sector_read` 和 `sector_write` 两个函数完成。**

1.2 实验流程概述

本次实验中，同学们需要完成以下步骤：

1. 安装并测试 `libfuse` 库。
2. 补充 `simple_fat16_part1.c` 中空缺的一个或多个函数代码（代码中用TODO进行了标注）。
3. 编译并运行代码，在终端中访问文件系统，测试对应功能。

后续实验文档将会给出了完成每一个步骤的所需的流程和背景知识。**如果同学们在实验中遇到问题，请先查阅实验文档的相应章节和 在线文档 。**

1.3 FUSE环境配置

FUSE（Filesystem in Userspace，用户态文件系统）是一个实现在用户空间的文件系统框架，通过FUSE内核模块的支持，使用者只需要根据FUSE提供的接口并实现具体的文件操作就可以实现一个用户态的、自定义的文件系统。 如果感兴趣，同学们可以在本文的附录中了解更多关于FUSE的信息。

1.3.1 配置FUSE环境

我们此处可以选择两种方式安装libfuse：

1. **方法一：**使用 Ubuntu的包管理器直接安装，**推荐大家使用这种方式**，较为简单快捷。

```
sudo apt install libfuse-dev libfuse2
```

2. **方法二：**下载源码并编译安装。

```
$ sudo apt install pkg-config pkgconf
$ wget -O libfuze-2.9.5.zip https://codeload.github.com/libfuse/libfuse/zip/fuse_2_9_5
unzip libfuze-2.9.5.zip
$ cd libfuse-fuse_2_9_5/
$ ./makeconf.sh
$ ./configure --prefix=/usr
$ make -j4
$ sudo make install
```

1.3.2 测试FUSE

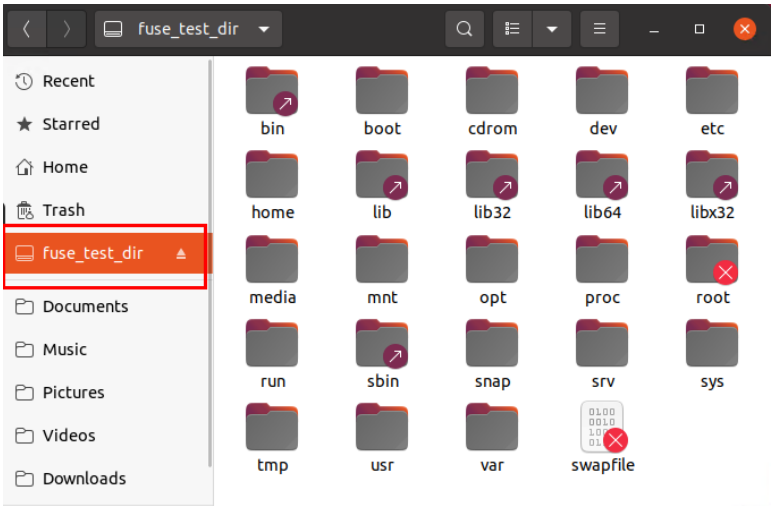
如果使用方法一安装 `libfuse`，**可以跳过本步骤**，或者单独下载 fusexmp.c 编译测试：

```
mkdir example; cd example
wget https://raw.githubusercontent.com/libfuse/libfuse/fuse_2_9_bugfix/example/fusexmp.c
gcc -o ./fusexmp fusexmp.c `pkg-config fuse --cflags --libs`
mkdir fuse_test_dir
./fusexmp -d fuse_test_dir
```

如果使用方法二编译安装 `libfuse`，可以通过 `libfuse-fuse_2_9_5/example` 下的fusexmp进行测试：

```
cd example
mkdir fuse_test_dir
./fusexmp -d fuse_test_dir
```

这时候在文件管理器中打开fuse_test_dir，可以看到当前Linux系统的根目录 `/` 被挂载到这个目录下，如下图：



结束测试可以直接在当前终端中Ctrl + C结束程序，或者在新的终端中输入：

```
fusermount -u fuse_test_dir
```

> **提示**：当执行用户自己实现的fuse程序时，如果出现错误（"fuse: bad mount point ...Transport endpoint is not connected"），可通过执行上面这条命令卸载对应的文件夹来解决。

第二部分 FAT文件系统初识

FAT（File Allocation Table）是“文件分配表”的意思。顾名思义，就是用来记录文件在磁盘中所处位置的表格。FAT对于磁盘的使用是非常重要的，假若丢失文件分配表，那么磁盘上的数据就会因无法定位而不能使用。不同的操作系统所使用的文件系统不尽相同，在个人计算机上常用的操作系统中，MS-DOS 6.x及以下版本使用FAT16。操作系统根据整个磁盘空间所需要的簇的数量来确定使用多大的FAT。其中簇是磁盘空间的配置单位，就象图书馆内一格一格的书架一样。FAT16使用了16位的空间来表示FAT表项（或者簇号），故称之为FAT16。

从上述描述中我们够得知，一个FAT16分区或者磁盘最多能够使用的簇数是2^16=65536个，因此簇大小是一个变化值，通常与分区大小有关，计算方式为：(磁盘大小/簇个数)向上按2的幂取整。在FAT16文件系统中，由于兼容性等原因，簇大小通常不超过32K，这也是FAT分区容量不能超过2GB的原因。

2.1 专有名词

名词	使用场景	释义
簇(cluster)	文件系统	文件的最小空间分配单元，通常为若干个扇区，每个文件最小将占用一个簇。
扇区(sector)	磁盘管理	磁盘上的磁道被等分为若干个弧段，这些弧段被称为扇区，硬盘的读写以扇区为基本单位。

2.2 磁盘布局

<table>
<tr><td colspan="8">一个FAT分区或磁盘的结构布局</td></tr>
<tr><td>内容</td><td>引导扇区</td><td>文件系统信息扇区</td><td>额外的保留空间</td><td>文件分配表(FAT表1)</td><td>文件分配表2(FAT表2)</td><td>根目录(Root Directory)</td><td>数据区</td></tr>
<tr><td>大小(Bytes)</td><td colspan="3">保留扇区数 * 扇区大小</td><td>FAT扇区数 * 扇区大小</td><td>FAT扇区数 * 扇区大小</td><td>根目录条目数 * 文件条目大小</td><td>剩下的磁盘空间</td></tr>
</table>

一个FAT文件系统包括四个不同的部分：

- **保留扇区**。位于磁盘最开始的位置。 第一个保留扇区是**引导扇区**（分区启动记录）。引导扇区中通常包括了操作系统启动所需的一些信息和代码，也包含了一些**文件系统需要的元数据信息**，例如簇的大小，保留扇区的数目等，都保存在引导扇区中，所以需要特别关注，我们将在下面详细介绍。保留扇区还包括文件系统信息扇区）（仅FAT32使用）和额外的保留扇区，本次实验无需用到这些扇区。
    - 保留扇区包括一个称为基本输入输出参数块的区域（包括一些基本的文件系统信息尤其是它的类型和其它指向其它扇区的指针），通常包括操作系统的启动调用代码。保留扇区的总数记录在引导扇区中的一个参数中。引导扇区中的重要信息可以被DOS和OS/2中称为驱动器参数块的操作系统结构访问。
- **FAT区域**。它包含有两份文件分配表，这是出于系统冗余考虑，尽管它很少使用。它是分区信息的映射表，指示簇是如何存储的。本实验中两个FAT表都要修改。
- **根目录区域**。它是在根目录中存储文件和目录信息的目录表。在FAT32下它可以存在分区中的任何位置，但是在FAT16中它永远紧随FAT区域之后。根目录区域的大小一定是 **整数个扇区大小** 。
- **数据区域**。这是实际的文件和目录数据存储的区域，它占据了分区的绝大部分。通过简单地在FAT中添加文件链接的个数可以任意增加文件大小和子目录个数（只要有空簇存在）。然而需要注意的是每个簇只能被一个文件占有：如果在32KB大小的簇中有一个1KB大小的文件，那么31KB的空间就浪费掉了。

2.3 引导扇区

引导扇区包含很多条目，但我们只需关注和文件系统有关的部分，这里列出如下：

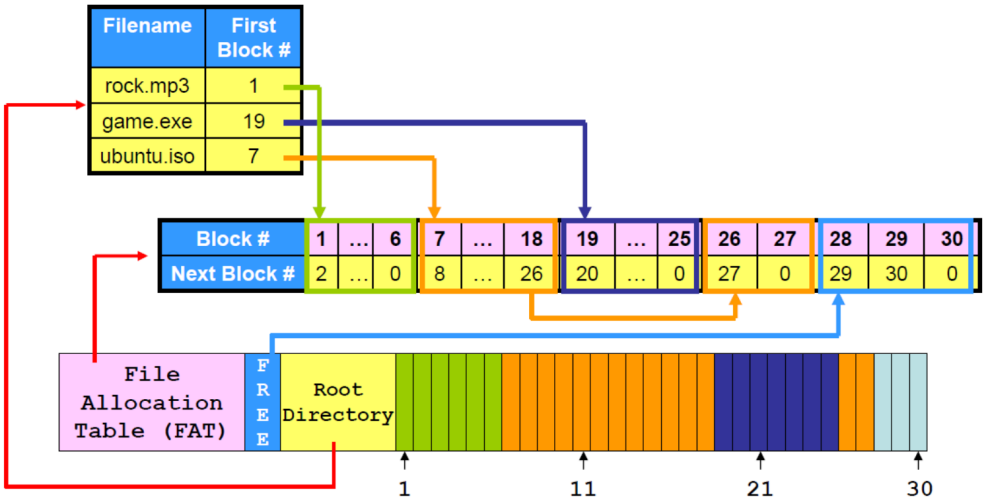
名称	偏移(字节)	长度(字节)	说明
...	...	...	...
BPB_BytsPerSec	0x0b	2	**每个扇区的字节数。基本输入输出系统参数块从这里开始。**
BPB_SecPerClus	0x0d	1	**每簇扇区数**
BPB_RsvdSecCnt	0x0e	2	**保留扇区数（包括引导扇区）**
BPB_NumFATS	0x10	1	**文件分配表数目，FAT16文件系统中为0x02,FAT2作为FAT1的冗余**
BPB_RootEntCnt	0x11	2	**最大根目录条目个数。一定是16的倍数，因为必须保证根目录区域对齐完整扇区。**
...	...	...	...
BPB_FATSz16	0x16	2	**每个文件分配表的扇区数（FAT16专用）**
...	...	...	...

完成实验仅需理解上述条目。引导扇区的完整结构详见附录。

通过这些条目，我们可以计算出一些偏移量，你可以结合2.4的布局示意图理解这些偏移量：

- FAT1偏移地址：保留扇区之后就是FAT1。因此可以得到，FAT1的偏移地址就是 `BPB_RsvdSecCnt` * `BPB_BytsPerSec`
- 根目录偏移地址： FAT1表后两个FAT表地址就是根目录区，即FAT1偏移地址 + `BPB_NumFATS` *`BPB_FATSz16`* `BPB_BytsPerSec` 。
- 数据区偏移地址：根目录偏移地址+根目录大小，即根目录偏移地址+ `BPB_RootEntCnt` * `BYTES_PER_DIR`

2.4 文件存储

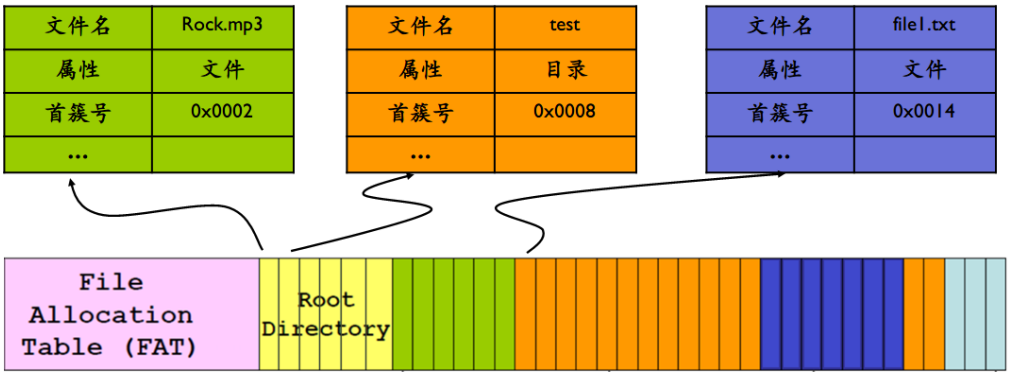


在FAT文件系统中，文件分为两个部分，第一个部分为目录项，记录该文件的文件名，拓展名，以及属性，首簇号等元数据信息（文件夹也是一种文件，它的目录项和普通文件结构相同）；第二部分为实际存储内容，若为文件，则存储文件内容，若为文件夹，则存储子文件夹的文件目录项。

- 例一，文件处于Root目录下 (假设该文件路径为/rock.mp3)
  若文件处于Root目录下，那么该文件的目录项将会存储在 `Root Directory` 区域中，其中记录了该文件的文件名，拓展名，以及第一个文件簇的地址。
  FAT Table中存储了所有可用的簇，通过簇地址查找使用情况，我们能够得知该文件的下一个簇地址，若下一个簇地址为END标记符，则表示该簇为最后一个簇。通过查询FAT Table，我们能够得知该文件所有簇号。
- 情况二，文件处于Root目录下子文件夹中 (假设该文件路径为/test/file1.txt)，那么可知test目录项在 `Root Directory` 中，而rock.mp3的目录项将会存储在test文件的数据区域。

2.4.1 目录项

文件目录是文件的元数据信息，存储在 `Root directory` 以及数据区中，下图是两个文件/rock.mp3和/test/file1.txt的文件目录项存储示例：



每个文件目录项的大小为0x20字节（32字节），每一项的结构如下：

名称	偏移(字节)	长度(字节)	说明
DIR_Name	0x00	11	**文件名。前8个字节为文件名，后3个为拓展名。注意第一位取值会有特殊含义，参见本表后的说明。**
DIR_Attr	0x0B	1	**文件属性，取值为0x10表示为目录，0x20表示为文件**
DIR_NTRes	0x0C	1	保留
DIR_CrtTimeTenth	0x0D	1	保留(FAT32中用作创建时间，精确到10ms))
DIR_CrtTime	0x0E	2	保留(FAT32中用作创建时间，精确到2s)
DIR_CrtDate	0x10	2	保留(FAT32中用作创建日期)
DIR_LstAccDate	0x12	2	保留(FAT32中用作最近访问日期)
DIR_FstClusHI	0x14	2	保留(FAT32用作第一个簇的两个高字节)
DIR_WrtTime	0x16	2	文件最近修改时间
DIR_WrtDate	0x18	2	文件最近修改日期
DIR_FstClusLO	0x1A	2	**文件首簇号(FAT32用作第一个簇的两个低字节)**
DIR_FileSize	0x1C	4	**文件大小**

目录项的状态：若一个目录项0x00偏移处取值为**0x00**，则表示**目录项为空**，取值为**0xE5**则表示曾被使用但目前**已删除。**

2.4.2 文件分配表(FAT表)

FAT表是由FAT表项构成的，我们把FAT表项简称为FAT项。每个FAT项的大小有12位，16位，32位，三种情况，对应的分别FAT12，FAT16，FAT32文件系统。 本次实验中，我们需要实现FAT16文件系统，故**每个FAT项大小为16bit，即2字节。**每个FAT项都有一个固定的编号，这个编号是从0开始。

FAT表的前两个FAT项有专门的用途：0号FAT项通常用来存放分区所在的介质类型，例如硬盘的介质类型为“F8”，那么硬盘上分区FAT表第一个FAT项就是以"F8"开始，1号FAT项则用来存储文件系统的肮脏标志，表明文件系统被非法卸载或者磁盘表面存在错误。

分区的数据区每一个簇都会映射到FAT表中的唯一一个FAT项。因为0号FAT项与1号FAT项已经被系统占用，无法与数据区的簇形成映射，所以从2号FAT项开始跟数据区中的第一个簇映射，正因为如此，数据区中的第一个簇的编号为2，这也是没有0号簇与1号簇的原因。

分区格式化后，用户文件以簇为单位存放在数据区中，一个文件至少占用一个簇。当一个文件占用多个簇时，这些簇的簇号不一定是连续的，但这些簇号在存储该文件时就确定了顺序，即每一个文件都有其特定的“簇号链”。在分区上的每一个可用的簇在FAT中有且只有一个映射FAT项，通过在对应簇号的FAT项内填入“FAT项值”来表明数据区中的该簇是已占用，空闲或者是坏簇三种状态之一，FAT项具体取值及含义见下表：

簇取值	对应含义
0x0000	空闲簇
0x0001	保留簇
0x0002 - 0xFFEF	被占用的簇；指向下一个簇
0xFFF0 - 0xFFF6	保留值
0xFFF7	坏簇
0xFFF8 - 0xFFFF	文件最后一个簇，在本次实验中，为了方便起见，我们均使用**0xFFFF**作为最后一个簇的标志。

第三部分 实验任务

3.0 代码框架

3.0.1 文件说明

在编写自己的代码之前，建议同学们先大概阅读下已经给出的代码框架，了解代码的执行流程以及封装好的函数的功能及使用方法。本实验涉及的代码文件主要有：

- fat16.h：文件系统数据结构和函数的定义
- simple_fat16_part1.c：基础部分的代码，补充挖空的代码即可完成；
- simple_fat16_part2.c：进阶部分的代码，目前只有接口定义，会在下周发布含挖空和更多提示的代码，有能力的同学也可以尝试自主完成。
- fat16.img：FAT16格式镜像文件。

3.0.2 功能函数

为了减少代码实现中的工作量，框架中提供了一系列功能函数来辅助完成（simple_fat16_part1.c 16~618行）。我们在代码的注释中已经具体给出了每个函数的功能和输入输出含义，请各位同学认真阅读。

3.0.3 编译、运行

1. 在代码目录下执行 `make` （Makefile已经写好），并确认没有error（可能会存在warning，可以忽略）；
    - Makefile会将simple_fat16_part1.c和simple_fat16_part2.c编译成一个二进制文件。不完成TODO不影响成功编译。
2. 创建一个空文件夹 `fat_dir` 用于挂载镜像文件；
3. 然后使用命令 `./simple_fat16 -s -d fat_dir` 运行程序。我们提供的代码框架会打开名为全局变量 `FAT_FILE_NAME` 的镜像文件挂载。我们实验检查使用fat16.img，你也可以选择挂载其他的镜像。

> `-s` 参数代表 `single-thread` ，会强制FUSE使用单线程模式运行，一次只能有一个线程访问挂载的文件系统，这避免了数据争用带来的Bug，也简化了我们的调试。

3.1 任务一：实现FAT文件系统读操作

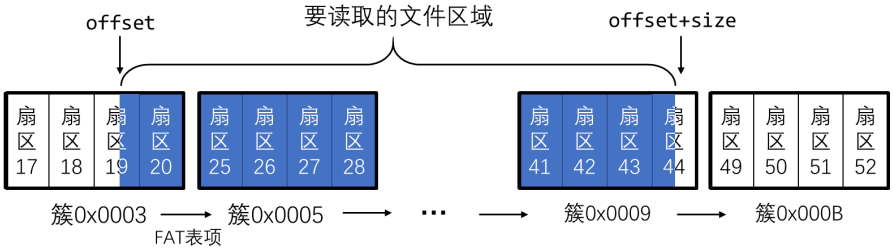
3.1.1 读目录

文件系统需要有获取文件目录结构的能力，从而保证 `ls` 和 `tree` 命令的实现。在我们的代码框架中，对应要实现的是 `int fat16_readdir(const char *path, void *buffer, fuse_fill_dir_t filler, off_t offset, struct fuse_file_info *fi)` ，该函数对于给定的路径path，首先找到对应目录的数据区，然后读出其中每条目录项的信息，通过传入的filler函数记录到buffer中，直到遇到0x00的结束项。
在读目录对应簇的时候，需要注意两点：

1. 一个扇区读完如果并没有读到0x00，那么就需要继续读下一个扇区。
2. 一个簇的扇区如果都读完了，需要找到下一个簇继续读。
3. 请跳过LFN（长文件名）项。该问题详见Part2.pdf的1.3。

3.1.2 读文件

当调用`cat` / `head` / `tail`等读文件的命令时，就会调用到读文件的接口，即`int fat16_read(const char *path, char *buffer, size_t size, off_t offset, struct fuse_file_info *fi)`。读文件的实现逻辑与读目录基本相同，也需要注意跨扇区与跨簇的问题。



read()的读取范围

如图，读的起始地址和结束地址可能位于某个扇区的中间，即除了读覆盖的第一个扇区和最后一个扇区以外，中间的扇区都是全部读出来的，而第一个扇区只读出起始地址之后的内容，最后一个扇区只需要读结束地址之前的内容。所以，需要小心地计算要读取的第一个簇和扇区，以及最后一个簇和扇区，并正确处理他们的读取范围。

此外，读文件可能出现越界的问题，我们这里不做特殊要求，检测到越界返回0或者读到文件结束返回都可以。

3.1.3 实验要求

完成simple fat16中关于读文件的部分。

- **待补全部分**：完成simple_fat16.c文件中的TODO。**我们推荐在BEGIN和END间填充代码。**主要包括以下功能：
  1. `fat16_readdir` 遍历给定目录下的所有文件名，返回给fuse；
  2. `fat16_read` 从给定文件中读取一定长度的数据，返回给fuse。

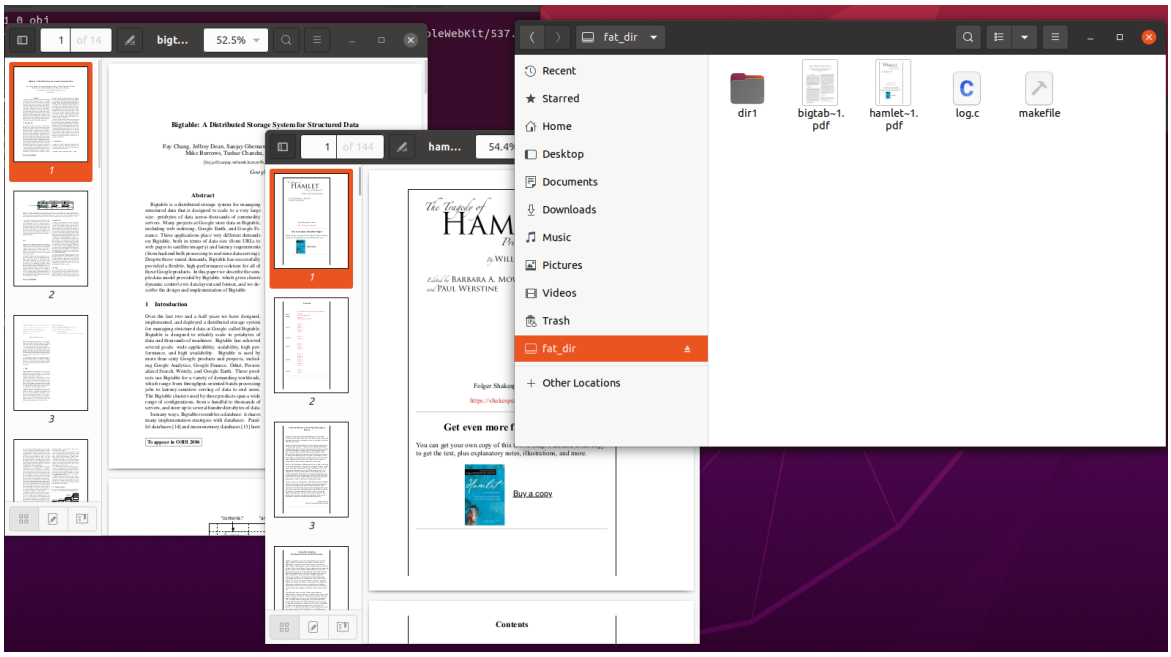
完成实验要求的代码能够从另一个shell中进入该目录，并执行tree和tail等命令得到正确的结果，如下图：

1. 能够运行tree/ls命令查看文件目录结构。输出示例：

```
blahaj@blahaj-VirtualBox:~/lab4-code-complete/fat_dir$ tree
.
├── bigtab~1.pdf
├── dir1
│   └── dir2
│       └── dir3
│           └── test.c
├── hamlet~1.pdf
├── log.c
└── makefile
```

2. 能够运行cat/head/tail命令查看目录下文件内容，或在文件管理器GUI下直接打开文件。
   - 我们在镜像中提供了若干文件。其中，一些文件大小较小，不存在跨簇/跨扇区的情况，一些文件较大，会出现跨簇/扇区的情况，大家可以据此分析代码错误所在。

挂载后用文件管理器GUI打开文件的示例：



3.2 任务二：实现FAT文件系统创建/删除文件操作

在实现了支持读操作的FAT 16文件系统基础上，实现创建文件和删除文件的功能。

3.2.1 创建文件

也就是需要保证文件系统支持 `touch` 命令，具体来讲，创建文件需要完成以下工作：

- 找到新建文件的父目录，有可能是根目录或者是子目录，根目录只有固定大小，子目录可以有多个簇；
- 在父目录中找到可用的entry，即未用过或用过但已删除的entry；
- 将新建文件的信息填入该entry，我们这里需要补充的是填入文件名、文件属性、首簇号以及文件大小。

因为创建文件会涉及到一些扩容相关的事情，但是扩容目前并不要求解决，会在进阶任务中进一步完成，这里解释两点可能会出现的疑问：

1. 创建文件会占用一个目录表项，FAT 16文件系统根目录大小是固定的，也就意味着最多只能有固定数量的表项可用。但是子目录和正常文件一样存放在数据区，这意味着子目录空间是可变的。如果子目录当前簇用完了，可以通过申请新的簇来增加子目录存放表项数量，这个扩容的操作在进阶任务中做进一步讨论，故完成本部分任务时，可以暂不考虑目录用满的解决方案。
2. 在创建文件时，可以传入 `0xffff` 作为新文件的首簇号，并将文件大小定为0。扩容同样在进阶任务中完成，此处不用担心文件后面写入的问题。

3.2.2 删除文件

也就是需要保证文件系统支持 `rm` 命令。删除一个文件，不需要改变文件具体的内容，而是在文件目录中更新对应的表项，文件系统在遍历时就会跳过被删掉的目录项，并在新文件创建时，允许覆盖掉对应的目录项。删除文件的思路大致分为两步：

- 释放该文件使用的簇（遍历簇，修改对应的FAT表项使其标记为空闲）
- 在父目录中释放该文件的entry（和创建文件类似）

代码流程上和创建文件相似，此处不做过多说明。

3.2.3 实验要求

同学们需要补全下面4个函数：

1. `fat16_mknod` 在给定path路径下创建新文件。

   > 在实现 `mknod` 时，你需要能正确识别出LFN项是非空的目录项，不将它们用于创建新文件、文件夹。（一般来说，这条不需要特别实现，因为LFN项的首个字节不会为 `0x00` 或 `0xe5` 等代表未使用或已删除条目字符）。LFN相关问题详见Part2.pdf的1.3。

2. `fat16_unlink` 删除给定path路径下对应的文件。
3. `dir_entry_create` 创建新目录表项。
4. `free_cluster` 释放簇号对应的簇，只需修改FAT对应表项，并返回下一个簇的簇号。

函数的主体框架已经给出，同学们需要实现每个函数中的TODO标记部分。**我们推荐在BEGIN和END间填充代码。**调试过程与任务一相同。

检查要求：需要能够创建文件；需要能够删除涉及跨簇的文件。

> 关于 `touch` 命令：`touch` 命令本身会检查文件是否存在，存在则不会创建文件。`touch` 的基本作用是更新文件的修改时间，创建新文件其实是副作用，但我们目前只使用这个副作用。所以，`touch` 一个已存在的文件时不会触发 `fat16_mknod` 函数。

3.3 任务三：实现FAT文件系统创建/删除目录操作

> 本部分是进阶内容，挖过空的代码将于第二周发布。有能力的同学可以【先行完整地】自己完成两个函数。

实现 FAT16 文件夹创建/删除操作，主要包括如下两个函数的实现：

```
int fat16_mkdir(const char *path, mode_t mode);
int fat16_rmdir(const char *path);
```

这两个函数分别负责目录的创建及删除功能。FUSE库会将来自终端的 `mkdir` 命令转为函数 `fat16_mkdir` 的执行，将 `rmdir` 命令转为函数 `fat16_rmdir` 的执行。

3.3.1 创建文件夹

`fat16_mkdir` 的实现思路：

1. 找到新建文件夹的父目录；
2. 在父目录中找到可用的entry；
3. 将新建文件夹的信息填入该entry；

   这里的实现思路看起来与 `fat16_mknod` 完全一致（事实上，找到父目录以及在父目录中找到可用entry两部分的思路确实一致）。但是，我们在填入entry一步还需要其他操作。

4. 在FAT表中申请一个空闲簇，将其作为新建文件夹的初始簇，并在该簇中插入 `.` 和 `..` 两个目录项。

   这一操作是有其必要性的。第一，课程内容讲过目录文件内前两条entry分别是该文件本身和父目录文件的符号链接，它们在目录文件创建时就已经存在；第二，本次实验中不要求分配新簇，如果不在创建时就分配簇的话，会导致无法在该文件夹中创建文件，这是不可容忍的bug。

   但由于本次实验不涉及链接操作，所以不需要把 `.` 和 `..` 链接到对应目录，FUSE会在涉及这些目录时自动识别。填写它们的entry时，按照实际涉及的两个目录信息填写参数。

3.3.2 删除文件夹

`fat16_rmdir` 的实现思路：

1. 通过 `path` 获取到待删除目录对应的 `DIR_ENTRY` 所在位置；
2. 将 `DIR_ENTRY` 中 `DIR_Name[0]` 置为 `0xe5`，表示该目录已删除；
3. 返回 `0` 表示正常结束，否则表示异常。

注：`fat16_rmdir` 不需要删除待删目录下的子目录与文件。

3.4 任务四：实现FAT文件系统写操作

> 本部分是进阶内容，挖过空的代码将于第二周发布。有能力的同学可以【先行完整地】自己完成两个函数。

实现 FAT16 文件的写操作，主要包括如下两个函数的实现：

```
int fat16_write(const char *path, const char *data, size_t size, off_t offset, struct
fuse_file_info *fi);
int fat16_truncate(const char *path, off_t size);
```

函数解释说明：

- `fat16_write`：将 `data[0..size-1]` 的数据写入到 `path` 对应的，且偏移为 `offset` 的文件中；
- `fat16_truncate`：将 `path` 对应的文件大小设置为 `size`，如果 `size` 过大，初始化数据为 `0x00`。

`fat16_write` 函数的实现思路如下：

1. 通过 path 获取到对应的目录项，即 `DIR_ENTRY` 结构体，通过调用 `find_root` 即可；
2. 比较新写入的数据和文件原来的数据所需的簇数量，如果新写入的数据所需的簇数量大于原来所需的簇数量，则需要为文件扩容簇数量，具体流程可以在 FAT 表项中查找未使用的簇项，然后将其链接到原来文件的末尾；
3. 按需扩容后，只需要实现写文件操作即可，通过 `DIR.FstClusLo` 获取第一个簇号，之后可以通过链接依次遍历，直到达到 `offset` 对应的簇即可，写的流程可以参考3.1.2读流程中的说明（如写区间开头块和结束块的讨论）；
4. 实现目录项的更新，通过 `find_root` 可以获取到目录项对应的偏移量，更新对应的目录项数据写入即可；
5. 返回对应的写入数据字节数。

`fat16_truncate` 函数的实现思路如下：

1. 通过文件路径 path 获取到对应的目录项，即 `DIR_ENTRY` 结构体，通过调用 `find_root` 即可；
2. 比较新数据大小（size）所需簇数量 n1 和文件原来的数据所需的簇数量 n2，如果新写入的数据所需的簇数量大于原来所需的簇数量，则需要为文件扩容簇数量，具体流程可以在 FAT 表项中查找未使用的簇项，然后将其链接到原来文件的末尾：
    1. n1 = n2：此时不需要任何操作；
    2. n1 > n2：此时需要扩容文件大小，可以参照 `fat_write` 实现思路；
    3. n1 < n2：此时需要截断文件，遍历文件的簇号，直到第 n1 次，此时将对应的内容改为 `0XFFFF`，表示该簇是文件结尾。
3. 按需写入，比较新文件的大小 new_size（就是参数中的 `size`）和原来的文件大小old_size（`Dir.DIR_FileSize`）：
    1. new_size > old_size：需要写入 `0x00` 到末尾的 `new_size - old_size` 个字节处；
    2. new_size <= old_size：不需要进行任何操作；
4. 实现目录项的更新，通过 `find_root` 可以获取到目录项对应的偏移量，更新对应的目录项数据写入即可；
5. 返回 0 表示正常结束，否则表示异常。

第四部分 检查内容 & 评分标准

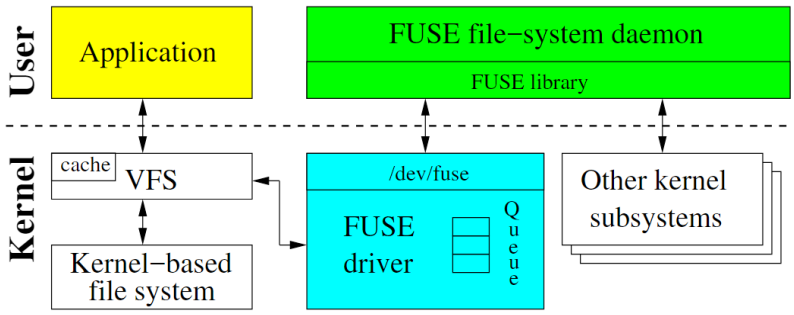
1. 任务一：实现FAT文件系统读操作（**满分2分**）
    - 能够运行 `tree` ， `ls` 命令查看文件目录结构 （**1分**）
    - 能够用过 `cat` / `head` / `tail` 命令，或直接打开文件（如直接打开pdf）查看目录下文件内容（**1分**）
2. 任务二：实现FAT文件系统创建/删除文件操作（**满分2分**）
    - 能够运行 `touch` 命令创建新文件，且保证文件属性的正确填写（**1分**）
    - 能够运行 `rm` 命令删除已有文件，且保证簇的正确释放（**1分**）
3. 任务三：实现FAT文件系统创建/删除文件夹操作（**满分2分**）
    - 能够运行 `mkdir` 命令创建新目录，且新目录中默认具有 `.` 和 `..` 两项（**1分**）
    - 能够运行 `rm -r` 命令删除已存在的目录（**1分**）
4. 任务四：实现FAT文件系统写操作（**满分2分**）
    - 能够使用 `echo` 命令和重定向符 `>` ， `>>` 正确进行文件写入（**2分**）
5. 知识问答（**满分2分**）
    - 回答从知识问答一节中随机抽取的三道问题，每回答正确一题得一分。回答对两题即可得到满分（**2分**）

知识问答

会同进阶部分的文档一并发布。

附录

1. FUSE简介



- FUSE主要由三部分组成：FUSE内核模块、用户空间库libfuse以及挂载工具fusermount：
    1. fuse内核模块：实现了和VFS的对接，实现了一个能被用户空间进程打开的设备。
    2. fuse库libfuse：负责和内核空间通信，接收来自/dev/fuse的请求，并将其转化为一系列的函数调用，将结果写回到/dev/fuse；提供的函数可以对fuse文件系统进行挂载卸载、从linux内核读取请求以及发送响应到内核。
    3. 挂载工具：实现对用户态文件系统的挂载。
- 更多详细内容可参考 **这个文章**。

2. FAT 16 文件系统中的时间

```
struct tm{
    int tm_sec; //代表目前秒数，正常范围为0-59，但允许至61秒
    int tm_min; //代表目前分数，范围0-59
    int tm_hour;  //从午夜算起的时数，范围为0-23
    int tm_mday;  //目前月份的日数，范围01-31
    int tm_mon;  //代表目前月份，从一月算起，范围从0-11
    int tm_year;  //从1900 年算起至今的年数
    int tm_wday;  //一星期的日数，从星期一算起，范围为0-6
    int tm_yday;  //从今年1月1日算起至今的天数，范围为0-365
    int tm_isdst;  //日光节约时间的旗标，也就是夏令时
};

// 用timer的值来填充tm结构。timer的值被分解为tm结构，并用本地时区表示。
struct tm *localtime(const time_t *timer)

// 偏移地址0x16：长度2，表示时间=小时*2048+分钟*32+秒/2。
// 也就是：0x16地址的0~4位是以2秒为单位的量值，
//         0x16地址的5~7位和0x17地址的0~2位是分钟，
//         0x17地址的3~7位是小时。

// 偏移地址0x18：长度为2，表示日期=(年份-1980)*512+月份*32+日。
// 也就是：0x18地址的0~4位是日期数，
//         0x18地址5~7位和0x19地址的0位是月份，
//         0x19地址的1~7位为年号。
```

3. 引导扇区的完整结构

这里列出了引导扇区的完整结构，但本实验只需要关注加粗的几个字段。

名称	偏移(字节)	长度(字节)	说明
BS_jmpBoot	0x00	3	跳转指令（跳过开头一段区域）
BS_OEMName	0x03	8	OEM名称，Windows操作系统没有针对这个字段做特殊的逻辑，理论上说这个字段只是一个标识字符串，但是有一些FAT驱动程序可能依赖这个字段的指定值。常见值是"MSWIN4.1"和"FrLdr1.0"
BPB_BytsPerSec	0x0b	2	**每个扇区的字节数。基本输入输出系统参数块从这里开始。**
BPB_SecPerClus	0x0d	1	**每簇扇区数**
BPB_RsvdSecCnt	0x0e	2	**保留扇区数（包括主引导区）**
BPB_NumFATS	0x10	1	**文件分配表数目，FAT16文件系统中为0x02,FAT2作为FAT1的冗余**
BPB_RootEntCnt	0x11	2	**最大根目录条目个数。一定是16的倍数，因为必须保证根目录区域对齐完整扇区。**
BPB_TotSec16	0x13	2	总扇区数（如果是0，就使用偏移0x20处的4字节值）
BPB_Media	0x15	1	介质描述：F8表示为硬盘，F0表示为软盘
BPB_FATSz16	0x16	2	**每个文件分配表的扇区数（FAT16专用）**
BPB_SecPerTrk	0x18	2	每磁道的扇区数
BPB_NumHeads	0x1a	2	磁头数
BPB_HiddSec	0x1c	4	隐藏扇区数
BPB_TotSec32	0x20	4	总扇区数（如果0x13处不为0，则该处应为0）
BS_DrvNum	0x24	1	物理驱动器号，指定系统从哪里引导。
BS_Reserved1	0x25	1	保留字段。这个字段设计时被用来指定引导扇区所在的
BS_BootSig	0x26	1	扩展引导标记（Extended Boot Signature）
BS_VollID	0x27	4	卷序列号，例如0
BS_VollLab	0x2B	11	卷标，例如"NO NAME "
BS_FilSysType	0x36	8	文件系统类型，例如"FAT16"
Reserved2	0x3E	448	引导程序
Signature_word	0x01FE	2	引导区结束标记

参考资料

- **FAT文件系统实现**
- **文件分配表**
- **fat32文件系统示例**

实验四 FAT文件系统的实现 Part2

> 提示：本文档涉及一些对Part1问题的更正。所以只要你计划完成Part1，就建议你阅读本文档的部分内容。

实验目标

- 熟悉文件系统的基本功能与工作原理（理论基础）
- 熟悉FAT16的存储结构，利用FUSE实现一个FAT文件系统：
  - 文件目录与文件的读（只读文件系统，基础）
  - 文件的创建与删除（基础）
  - 文件目录的创建与删除（进阶）
  - 文件的写（进阶）
- 在实现fat16文件系统的基础上，引入文件系统的重要问题研究（进阶，选做）
  - 如何做文件系统的错误修复——三备份与纠删码
  - 如何做文件系统的日志恢复——元数据日志

实验环境

- VMware / VirtualBox
- OS: Ubuntu 20.04.4 LTS

实验时间安排

注：此处为实验发布时的安排计划，请以课程主页和课程群内最新公告为准

- 5月13日晚实验课，讲解Part1及检查实验
- 5月20日晚实验课，讲解Part2检查实验
- 5月27日晚实验课，检查实验
- 6月3日晚实验课，实验补检查

友情提示

- **本次实验总工作量较大，请尽早开始实验。**
- **本次实验有600余行注释，以解决实验文档对于代码实现的提示量不足的问题，代码实现过程中务必注意看注释！看注释！看注释！**
- 实验多个任务难度依次递增，靠前的任务代码提示较为详尽，你可以相对容易地获得这部分分数。
- 实验多个任务之间较为独立，你也可以选择只完成靠后的任务。
- **本实验文档较为详尽，是为了尽可能提前解决大家实验中可能遇到的问题。你并不一定需要完全阅读所有实验文档才能完成实验、获得分数。**

第一部分：对Part1的补充说明

1.0 关于代码版本的说明

我们在Part2中发布了完整的实验代码和文档包 `lab4-all.zip` ，由于其中的 `simple_fat16_part1.c` 相比于Part1发布的版本稍有修改，我们建议你根据目前的实验情况，选择以下一种方式处理你的代码：

- 如果你**还未开始进行实验**：请进行忽略之前发布的 `lab4-part1.zip` ，**使用 `lab4-all.zip` 中的文件完成实验即可。**
- 如果你**已经开始了Part1的实验**：请使用 `lab4-all.zip` 中的 `simple_fat16_part2.c` 替换Part1发布的同名文件，**并按照1.1节的方法，修正 `simple_fat16_part1.c` 中的计算错误。**

1.1 修正RootOffset的计算错误

若你还未开始进行实验，你可以忽略这一节的内容，使用5月20日发布的 `lab4-all.zip` 中的代码完成实验即可。

在5月13日发布的 `simple_fat16_part1.c` 的第293行，函数 `pre_init_fat16` 的倒数第5行，对 `fat16_ins->RootOffset` （即根目录区域在镜像中的偏移量）的计算出现了错误，误将第一个运算符由 `+` 误打为 `*` 。
这会导致使用以下字段和函数时可能出现错误：

- `fat16_ins->RootOffset`
- `fat16_ins->DataOffset`
- `find_root` 函数的 `offset_dir` 输出参数

我们在Part2的代码中，添加了 `get_fat16_ins_fix` 函数修正该错误，所以在Part2中使用这些字段和函数不会出现问题。但由于这些字段和函数，特别是 `find_root` 函数在实验中大量使用，为防止该错误导致程序出现难以检查的问题，我们强烈建议你**按如下方式手动修正 `simple_fat16_part1.c` 的代码：**

- 在 `simple_fat16_part1.c` 中，查找 `fat16_ins->RootOffset =` ，定位到错误代码（在没有添加代码的文件中，该错误行在293行，函数 `pre_init_fat16` 的倒数第5行），如下图红色行所示。
- 将该行代码中第一个 `*` 改为 `+` ，修改后的代码如下图绿色行所示。
- 你也可以自行阅读改行代码，并对比Part1文档的2.3节，理解该行代码，并检查代码的正确性。

```
291  292      fat16_ins->FatOffset =  fat16_ins->Bpb.BPB_RsvdSecCnt * fat16_ins->Bpb.BPB_BytsPerSec;
292  293      fat16_ins->FatSize = fat16_ins->Bpb.BPB_BytsPerSec * fat16_ins->Bpb.BPB_FATSz16;
293       -   fat16_ins->RootOffset = fat16_ins->FatOffset * fat16_ins->FatSize * fat16_ins->Bpb.BPB_NumFATS;
     294  +   fat16_ins->RootOffset = fat16_ins->FatOffset + fat16_ins->FatSize * fat16_ins->Bpb.BPB_NumFATS;
294  295      fat16_ins->ClusterSize = fat16_ins->Bpb.BPB_BytsPerSec * fat16_ins->Bpb.BPB_SecPerClus;
295  296      fat16_ins->DataOffset = fat16_ins->RootOffset + fat16_ins->Bpb.BPB_RootEntCnt * BYTES_PER_DIR;
```

1.2 强制FUSE使用单线程模式运行

在Part1的3.0.4节，我们给出如下命令用于运行我们的程序，将我们的文件系统挂载到 `fat_dir` 目录下：

```
./simple_fat16 -d fat_dir
```

该命令会导致FUSE使用默认的多线程模式运行，在一般情况下，这不会导致什么问题，但由于我们提供的代码并不保证线程安全，当发生对文件系统并发访问时，有可能因为数据争用而导致错误。例如，某些pdf阅读器为了加速渲染等过程，或由多个线程同时对读取文件，这可能会导致我们的文件系统出现偶然的读取错误，使得pdf阅读器显示不完全、出现乱码或者因发生错误而崩溃。这些错误因为数据争用的不确定性而难以复现。

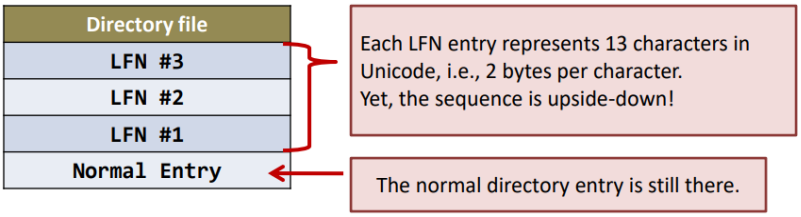
为了避免该问题，**我们推荐使用单线程模式运行FUSE，即使用下列命令运行我们的程序**：

```
./simple_fat16 -s -d fat_dir
```

`-s` 参数代表 `single-thread`，会强制FUSE使用单线程模式运行，一次只能有一个线程访问挂载的文件系统，这避免了数据争用带来的Bug，也简化了我们的调试。

1.3 忽略长文件名项（LFN entries）

在实现任务一的 `readdir` 函数时，一些程序可能会读取到文件名类似 `A1`，`Am`，`B0` 的目录项。如果检查该目录项的 `DIR_Attr` 字段，会发现这些目录项的属性均为 `0x0F`。这些项是FAT文件系统的长文件名项（LFN entries），用于存储文件的长文件名。长文件名可以占据一个或多个目录项的位置，在文件的普通的目录项之前，如图所示：



我们提供的镜像中，所有文件和目录都有对应的长文件名项（包括文件名没有超过8+3字节的文件）。本次实验，我们不要求对LFN进行处理，但是程序要能正确忽略这些LFN项，即，在实现 `readdir`，`mknod`，`rmdir` 等功能时，要主动忽略所有 `Dir_Attr` 为 `0x0F` 的目录项。即：

- 在实现 `readdir` 时，你应该跳过所有LFN项，不将它们读出。
- 在实现 `mknod`、`mkdir` 时，你能正确识别出LFN项是非空的目录项，不将它们用于创建新文件、文件夹。（一般来说，这条不需要特别实现，因为LFN项的首个字节不会为 `0x00` 或 `0xe5` 等代表未使用或已删除条目字符）。
- 在实现 `rmdir`，`unlink` 时，你不需要将文件或目录对应的LFN项删除，忽略它们即可。
- 由于上一条，在实现 `rmdir` 中，判断目录是否为空时，你需要跳过所有LFN项，也就是说，即使被删除的目录的目录项中下仍有LFN项，只有没有正常的目录项存在，你就可以认为该目录为空。

有兴趣的同学也可以思考如何正确读出文件的长文件名和创建LFN。如我们提供的镜像文件中pdf文件，全名是 `hamlet_PDF_FolgerShakespeare.pdf`，这个名称（包括大小写），完整的保存在了LFN中。

1.4 其它补充说明

1.4.1 安装 `tree` 指令

在测试读目录时，若出现找不到 `tree` 命令的情况，请使用包管理器手动安装 `tree` 命令。（不知道什么是包管理器？请回顾Lab1的实验文档。）

1.4.2 POSIX错误代码

本次实验中要求实现的大部分文件系统接口，出现错误时都要求返回POSIX错误代码的负值。你其实可以在错误时返回任意负数，但是这会导致终端显示的错误信息出现问题。如，假设你在出现所有错误时都返回 `-1`，那终端给出的错误提示将永远是 `Operation not permitted` 或 `权限不够`。所以我们推荐你在程序中返回正确的POSIX错误代码。

POSIX错误代码已经以宏定义的形式设置在 `error.h` 头文件中，fuse已经包含了这个头文件，你可以直接使用。错误的宏名和对应的意义可以在**这个链接**中找到，你也可以运行 `man errno` 或直接到 `error.h` 中查看这些宏定义的名字。这里给出一些可能用到的错误：

- `ENOENT` 文件或目录不存在
- `ENOSPC` 空间不足
- `EISDIR` 路径是目录
- `ENOTDIR` 路径不是目录
- `ENOTEMPTY` 目录非空
- `EINVAL` 参数非法
- `EIO` 读写错误
- `EBUSY` 设备或路径被占用（用于 `rmdir` 删除根目录时）
- ……

我们【**不要求，但推荐**】你准确使用这些错误代码，你可以按照自己理解随意使用。

例如，当你的程序执行 `readdir` 时，发现参数的 `path` 不存在，可以返回 `-ENOENT`；发现 `path` 是一个文件（自然没法“读目录”），可以返回 `EISDIR`。

第二部分：实验任务

2.0 前置任务：FAT中空闲簇的分配

在完成本部分实验前，我们建议实现 FAT 文件系统的空闲簇分配，这会极大简化在任务三和任务四的代码复杂度。空闲簇分配的使用场景如下：

- 在 FAT 中创建文件夹的时候，需要申请一个空闲簇，作为新建文件夹的起始簇，存储 `.` 和 `..` 目录项
- 在 FAT 中写文件的时候，如果写入量过大，需要申请额外的空间以防止写数据丢失问题

你也可以不实现本节中的函数，使用自己的逻辑完成任务三和任务四。

2.0.1 空闲簇分配

为实现空闲簇的分配，，我们提前定义了如下函数：

```
WORD alloc_clusters(FAT16 *fat16_ins, uint32_t n);
```

该函数作用是，在文件系统中分配n个未被使用的簇，将它们连接起来，返回首个分配的簇号。如图，如果我们需要分配四个簇，并且在FAT表中找到了 `0x06`，`0x07`，`0x18`，`0x27` 等四个空簇。我们需要将这四个空簇的表项连接在一起，并返回 `0x0006`：

分配前：

Cluster #	…	0x02	…	0x06	0x07	…	0x18	…	0x27	…
Next Cluster #	…	0x03	…	0x00	0x00	…	0x00	…	0x00	…

分配后：

Cluster #	…	0x02	…	0x06	0x07	…	0x18	…	0x27	…
Next Cluster #	…	0x03	…	0x07	0x18	…	0x19	…	0xFFFF	…

CLUSTER_END

该函数实现思路如下：

1. 扫描FAT表，找到n个空闲的簇（空闲簇的FAT表项为 `0x0000`）
2. 若找不到n个空闲簇，直接返回 `CLUSTER_END` 作为错误提示。注意，找不到n个簇时，不应修改任何FAT表项。
3. 依次清零n个簇，这需要将0写入每个簇的所有扇区
4. 依次修改n个簇的FAT表项，将每个簇通过FAT表项指向下一个簇，第n个簇的FAT表项应该指向 `CLUSTER_END`
5. 返回第1个簇的簇号

如果你不喜欢该函数的定义，你也可以选择不实现该函数，自行定义函数或用别的方式完成空闲簇分配的任务。

2.0.2 修改FAT表项

在实现 `alloc_clusters` 的过程中，需要修改多个簇的FAT表项，为了简化代码，我们建议实现 `write_fat_entry` 函数，用于修改单个簇对应的FAT表项。同样，你可以选择不实现这个函数，用自己的方式完成实验。

```
int write_fat_entry(FAT16 *fat16_ins, WORD clusterN, WORD data);
```

该函数作用是，将 `clusterN` 对应的FAT表项修改为 `data` 。这个函数和Part1中给出的 `fat_entry_by_cluster` 函数相似，后者作用为读出 `clusterN` 对应的表项，所以你可以参考 `fat_entry_by_cluster` 函数来实现该函数。该函数的实现思路如下：

1. 计算出 `ClusterN` 这个簇对应的FAT表项所在的扇区号和偏移量。
2. 读取所在扇区，将前述偏移量位置的FAT表项修改为 `data` ，然后写回该扇区。

2.1 任务三：实现FAT文件系统创建/删除文件夹操作

2.1.1 创建文件夹

该部分的主要任务是实现以下函数：

```
int fat16_mkdir(const char *path, mode_t mode);
```

`fat16_mkdir` 函数的功能是，创建 `path` 指定的新目录，注意传入的 `path` 必须保证父目录存在，如父目录不存在，直接返回 `-ENOENT` （文件不存在即可）。 `mode` 参数指定了创建文件夹时的一些选项，**在本次实验中，我们可以忽略 `mode` 参数。**

当在fuse挂载的目录中调用 `mkdir` 等命令时，fuse会将设计的文件系统操作转换为对函数 `fat16_mkdir` 的调用。所以我们只需实现 `fat16_mkdir` 函数，就能支持文件夹的创建功能。
`fat16_mkdir` 的实现思路：

1. 找到新建文件的父目录；
2. 在父目录中找到可用的entry；
3. 将新建文件夹的信息填入该entry：

   这里的实现思路看起来与 `fat16_mknod` 完全一致（事实上，找到父目录以及在父目录中找到可用entry两部分的思路确实一致）。但是，我们在填入entry一步还需要其他操作：
4. 在FAT表中申请一个空闲簇，将其作为新建文件夹的初始簇，并在该簇中插入 `.` 和 `..` 两个目录项：

   这一操作是有其必要性的。第一，在课上你已经学过，目录文件内前两条entry分别是该文件本身和父目录文件的符号链接，它们在目录文件创建时就已经存在；第二，我们本次实验中不要求在目录的目录项空间不足时自动拓展新簇，如果不在创建时就分配簇的话，会导致无法在该文件夹中创建文件和子文件夹，这是不可容忍的bug。如果你实现了前置任务中的 `alloc_clusters` 函数，你可以在此处使用它，简单地为你分配一个空簇，你只需要正确调用 `dir_entry_create` 就可以将这个簇连接到新目录下。

   但是，由于本次实验不涉及链接操作，所以你不需要把 `.` 和 `..` 链接到对应目录，FUSE会在涉及这些目录时自动识别。填写它们的entry时，按照实际涉及的两个目录信息填写参数。（目录的文件大小设置为0即可。）

2.1.2 删除文件夹

该部分的主要任务是实现以下函数：

```
int fat16_rmdir(const char *path);
```

`fat16_rmdir` 函数需要实现删除 `path` 对应的目录的功能，注意，传入的 `path` 对应的目录必须存在，且为空目录，否则应该直接返回错误。另外，文件系统的根目录 `/` 无法被删除。

在终端中运行 `rmdir` 指令时，fuse会将对应的文件系统操作转换为对 `fat16_rmdir` 的调用。而当在终端中调用 `rm -r` 时， `rm` 命令会依次递归的删除目录下的文件和子文件夹（转换为fuse中的 `unlink` 和 `rmdir` 操作），将目录清空后，在删除目录本身。这就是为什么 `fat16_rmdir` 要求 `path` 对应的目录为空，而使用 `rm -r` 时，却不需要保证文件夹为空。

`fat16_rmdir` 的实现思路如下：

1. 找到目录，确认目录为空。

2. 释放目录占用的所有簇。
3. 删除目录在父目录中的目录项。

实现思路中的第一步和 `readdir` 有一些相似，而第2、3步则类似 `unlink` 。你可以参考Part1中的这两个函数来实现文件夹删除。

2.2 任务四：实现FAT文件系统写操作

为了支持 FAT16 文件的写操作，需要实现如下两个函数：

```
int fat16_write(const char *path, const char *data, size_t size, off_t offset, struct
fuse_file_info *fi);
int fat16_truncate(const char *path, off_t size);
```

函数解释说明：

- `fat16_write` ：将 `data[0..size-1]` 的数据写入到 `path` 对应的文件的偏移为 `offset` 的位置，如果写入的数据超出了文件末尾，则应相应增大文件大小。 `fi` 参数在本次实验中可忽略。
- `fat16_truncate` ：将 `path` 对应的文件大小设置为 `size` 。如果 `size` 比原文件的大小要小，则从末尾处截断文件；如果 `size` 大于原文件的大小，那么将新增加部分的数据初始化为 `0` 。

当我们在终端中使用类似 `echo hello >> file` 的命令，通过 `>>` 重定向符向文件末尾追加写时，只会调用 `fat16_write` 函数。若我们使用 `>` 重定向符，如运行 `echo hello > file` 时，则会先调用 `fat16_truncate` 将文件大小设置为0，再调用 `write` 向文件中写入。所以为了支持文件写，两个函数都需要实现。我们也可以使用 `truncate` 命令来直接测试 `fat16_truncate` 的功能。

2.2.1 文件写入

首先看一下 `fat16_write` 的实现思路：

1. 通过 `path` 获取到对应的目录项，即 `DIR_ENTRY` 结构体，通过调用 `find_root` 即可。
2. 比较新写入的数据和文件原来的数据所需的簇数量，如果新写入的数据所需的簇数量大于原来所需的簇数量，则需要为文件扩容簇数量。即在FAT表中查找未分配的簇，并链接在文件末尾。如果你完成了前置任务中的 `alloc_clusters` ，你可以在此处使用它，为你分配你所需要的簇数量，你只需要正确地将分配好的簇链连接在你文件的末尾即可。（如果文件原来没有任何簇，你要把簇链连接在目录项的 `DIR.FstClusLo` ，否则，连接在原文件簇链的末尾。）
3. 按需扩容后，只需要实现写文件操作即可，通过 `DIR.FstClusLo` 获取第一个簇号，之后可以通过链接依次遍历，找到要写入簇的位置，并在读取相应的扇区，修改扇区的正确位置，并写回扇区。（这个过程和 `read` 的实现很类似。）
4. 实现目录项的更新，通过 `find_root` 可以获取到目录项对应的偏移量，更新对应的目录项数据写入即可；
5. 返回成功写入的数据字节数。

`fat16_write` 函数的流程有些复杂，为了简化代码逻辑，在提供的代码中，我们将这一过程拆分成了以下多个函数：

```
// 要实现的函数本身，完成流程的第一步，并正确调用write_file
int fat16_write(const char *path, const char *data, size_t size, off_t offset,
                struct fuse_file_info *fi);

// 写入文件的主要功能在此实现
// 先调用file_new_cluster完成第二步
// 然后循环调用write_to_cluster_at_offset实现第三步
// 最后更新目录项并返回正确的值
int write_file(FAT16 *fat16_ins, DIR_ENTRY *Dir, off_t offset_dir, const void *buff,
               off_t offset, size_t length);

// 完成第三步，调用alloc_clusters，并把新分配的簇链接在文件末尾
```

```c
int file_new_cluster(FAT16 *fat16_ins, DIR_ENTRY *Dir, WORD last_cluster, DWORD count);

// 将数据写入指定簇的对应偏移量，用于简化第三步的实现。
// 所有写入扇区的细节被封装在这个函数中，write_file函数只需要考虑簇一级即可。
size_t write_to_cluster_at_offset(FAT16 *fat16_ins, WORD clusterN, off_t offset,
                                  const BYTE* data, size_t size);

// 辅助函数，找到文件最后一个簇，并计算文件当前簇的数目。
// 在计算文件是否需要分配新簇，和将新簇连接至文件末尾时很有用。
WORD file_last_cluster(FAT16 *fat16_ins, DIR_ENTRY *Dir, int64_t *count);
```

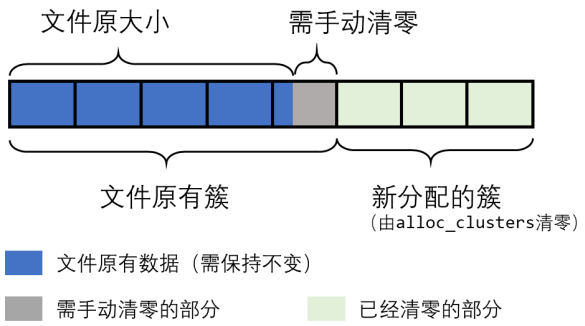
你可以在我们提供的代码中找到这些函数的详细功能即参数说明。你可以通过正确补充这些函数来实现文件写入的功能。当然，和前面的任务一样，你也可以不实现这些函数，而通过自己喜欢的逻辑实现 `fat16_write` 。

2.2.2 文件截断/拓展

`truncate` 意为“截断”，但实际上这个函数也能实现文件拓展，所有下文我们也用“截断”一词统称改变文件大小的操作。

接下来是 `fat16_truncate` 的实现思路：

1. 通过 path 获取到对应的目录项，即 `DIR_ENTRY` 结构体，通过调用 `find_root` 即可
2. 计算并比较原文件拥有的簇数量 `n1` ，和文件截断后需要的新簇数量 `n2` 。通过比较 `n1` 和 `n2` 的大小，来判断文件是否需要新增或释放簇：
    1. `n1 == n2` ：此时不需要任何操作
    2. `n1 < n2` ：此时需要扩容文件大小，可以参照 `fat_write` 实现思路。
    3. `n1 > n2` ：此时需要截断文件，找到文件的第 `n2` 个簇，更改FAT表项，将此处改为文件末尾，并释放后续所有簇。（释放可使用Part1中的 `free_clusters` 函数。）
3. 按需写入，比较新文件的大小 `new_size` （就是参数中的 `size` ） 和原来的文件大小 `old_size` ：
    1. `new_size > old_size` ：需要从 `old_size` 位置开始，将后续文件数据清空为 `0x00` 。如果你正确实现了 `alloc_clusters` ，那么新分配的簇应该已经清空，你只需要将文件原末尾，至原文件的最后一个簇的结尾的一小段手动清零即可。（如图所示，你也可用自己的方式清零。）



    2. `new_size <= old_size` ：不需要进行任何操作
3. 实现目录项的更新，通过 `find_root` 可以获取到目录项对应的偏移量，更新对应的目录项数据写入即可
4. 返回 0 表示正常结束，否则表示异常

2.3 选做：实现FAT文件系统的容错与日志机制

2.3.1 实现文件系统的三备份容错机制（1'）

存储介质在使用过程中有概率会出现不可预测的错误事件，比如单比特反转，在特定场景下会产生严重的影响（某个参数的量级改变）。因此现代计算机系统中会考虑各种技术中的出错现象并设计一定的方案来保证出错后系统的正常运行，如我们课上学到的RAID组织方式下，通常会有冗余或者纠删码来提供出错检测及出错后的修复能力。本次实验中我们可以尝试实现一种最易于理解的纠错设计——三备份冗余，即一个磁盘上的内容，总共保存三份副本，这样当任意一个副本中出现错误时，根据少数服从多数的原则，将两个没有错误的副本中的内容作为正确内容，对于出错副本做修复。一般三份副本会存在三块独立的存储设备上，因此在同一处同时发生两次或三次错误的可能性较低，在一定程度上保证了磁盘的容错能力。

本部分的实现思路为：

- 将原有的img文件复制额外复制两份，在初始化时读入三份img文件
- 维护一个三副本的扇区写接口，每次写入时同时对三份img文件写（视为一个原子操作，但是这里暂时不要求多线程，单线程下能正常运行即可）
- 维护一个三副本的扇区读接口，每次读入三份扇区并做比较，如果有不同则进行修复，最后返回三份一致的扇区内容（同样视为一个原子操作）

本部分的检查要求：人工干扰单个磁盘的内容后（如某个位反转），文件系统可以正确修复该错误。为了验证此功能，你需要先找到一种干扰方式，使得三份img中只有两份可以读出正确内容，而被干扰的img读出的内容存在问题（或者可能直接无法工作），然后在支持三备份的文件系统中挂载三份img后，可以正确读出内容，并修复了存在问题的那个img，保证之后单独读时三份都正确。

2.3.2 实现文件系统的日志机制 (1')

磁盘作为非易失存储，具有掉电后依然保持有数据的特征。但是若设备在持久化存储过程中掉电，则会出现错误，如涉及到多个扇区写的原子操作，在写完一部分扇区后设备断电，那么重新通电时，已写完的部分就会成为错误的信息，且整个原子操作并未实际完成。为了确保掉电后可以正确恢复磁盘内的数据组织，现代文件系统如ext4通常会采用日志来记录每次磁盘操作，并在掉电恢复时先读磁盘操作，如果有尚未完成的操作则尝试恢复或是直接丢弃，从而保证磁盘数据的正确性。

考虑到代码实现的难度，我们对于日志的实现要求做了简化，完成本部分选做只需要额外生成一个log文件，log文件中存储有每次原子操作的记录，并分为begin-end两步，在原子操作开始时记录原子操作对应的参数，并在结束时再加入一条结束记录。

定义fuse的一个operation操作为一个原子操作，我们需要在这部分保证各种操作的原子性，保证多线程下fuse文件系统依然可以保证正确的运行顺序（不会在一个写操作执行的同时执行另一个写操作），之后围绕每个原子操作，做log的记录。在保证原子性的前提下，log的记录中不会出现一个原子操作过程中发生其他原子操作的情况。

本部分的检查要求：需要在多线程下正确打开pdf文件，并且产生的log中没有违反原子性的记录。log中的格式可以自行设计，需要至少有操作名、操作传入的参数，操作的begin-end记录。传入参数含写入数据的可以只记录前30个字符。

第三部分：知识问答

我们将从下列题目随机挑选 3 道，答对 2 题即可获得该部分满分：

1. FAT16 文件的目录名等元数据存储在哪？数据存储在哪，是怎么组织的？
2. FAT16 系统中需要两个 FAT 表的目的是什么？
3. FAT16 中的根目录和子目录在磁盘存储上有什么区别？
4. FAT16 的文件名存储格式是怎样的，如何转化成实际用户看到的文件名？
5. 不考虑LFN，FAT16 文件系统中支持的最大文件名长度是多少，为什么？
6. FAT16 文件系统的最大单文件的大小是多少，受限于什么因素？
7. FAT16 文件系统分配空间的基本单位是什么，文件系统读写磁盘的基本单位是什么？
8. FAT16 文件系统的根目录数量达到最大值的时候，再次创建目录时，会自动分配空闲簇吗？
9. FAT16 文件系统某个文件夹下的所有目录项都是紧密排列的吗？
10. FAT16文件系统的布局是什么样的？分为哪几个部分，它们的位置在哪？

第四部分 检查内容 & 评分标准

1. 任务一：实现FAT文件系统读操作（**满分2分**）
    - 能够运行 `tree` ， `ls` 命令查看文件目录结构 （**1分**）
    - 能够用过 `cat` / `head` / `tail` 命令，或直接打开文件（如直接打开pdf）查看目录下文件内容（**1分**）
2. 任务二：实现FAT文件系统创建/删除文件操作（**满分2分**）
    - 能够运行 `touch` 命令创建新文件，且保证文件属性的正确填写（**1分**）
    - 能够运行 `rm` 命令删除已有文件，且保证簇的正确释放（**1分**）
3. 任务三：实现FAT文件系统创建/删除文件夹操作（**满分2分**）
    - 能够运行 `mkdir` 命令创建新目录，且新目录中默认具有 `.` 和 `..` 两项（**1分**）
    - 能够运行 `rmdir` 和 `rm -r` 命令删除已存在的目录（**1分**）
4. 任务四：实现FAT文件系统写操作（**满分2分**）
    - 能够使用 `echo` 命令和重定向符 `>` ， `>>` 正确进行文件写入（**2分**）
5. 知识问答（**满分2分**）
    - 回答从知识问答一章中随机抽取的三道问题，每回答正确一题得一分。回答对两题即可得到满分（**2分**）
6. 选做：实现三备份和日志机制（**额外加分，满分2分**）
    - 请参考2.3.1和2.3.2中的检查标准，你需要自行设计方法，展示你的成果，并说明你的实现方案。