

uCore OS(on RISC-V64)实验指导书

Introduction

- [ucore实验指导书](#)
 - [ucore labs 1-8 源码](#)
-

实验总体流程

1. 认真上操作系统的理论课程；
 2. 阅读[ucore实验指导书](#)，并参考其内容完成联系和实验报告；
 3. 在实验环境中完成实验并提交到自己的[github](#)上；
 4. 遇到问题，首先查询手册等其他资料，先自行解决；
 5. 如若不能解决，可在飞书群里提问，可以互相讨论，有助教老师答疑。
-

学习目标与对应手段

1. 掌握OS基本概念：通过上课与学习教材，能理解OS原理与概念；阅读指导书并分析源码，能理解 `lab_codes_answer` 的labs运行结果。
 2. 掌握OS设计实现：在1的基础上，能够通过编程完成 `lab_codes` 的8个lab实验中的基本练习和实验报告。
 3. 掌握OS核心功能：在2的基础上，能够通过编程完成 `lab_codes` 的8个lab实验中的challenge练习。
 4. 掌握OS科学研究：联系老师，加入实验室，开始科研吧。
-

友情提示

1. 课程铺垫——计算机组成原理、C语言、数据结构
 2. 工具掌握——命令行 [shell](#)、软件管理 [apt-get/aptitude](#)、版本管理 [git/github](#)、代码阅读 [understand/VSCode](#)、代码比较 [diff/meld](#)、开发编译调试 [gcc/gdb/make](#)、硬件模拟器 [qemu](#)、md文档编写 [Typora](#)
-

实验报告要求

1. 独立完成；
 2. 用Markdown语言编写；
 3. 报告内容包括但不限于：实验目的、实验内容、实验步骤、实验结果、遇到的问题与解决方法；
 4. 报告编写完需按时发送给助教并上传到自己的github仓库里。
-

维护者

kelee@mail.nankai.edu.cn


如若对本指导书有任何疑问，请联系维护者！

Reference

[ucore step by step](#)

LAB0 : ready~go!

1. 学习与操作系统实验相关的前导知识，其主要包括实验环境、实验步骤、实验工具、uCore历史、RISC-V简介。
2. 配置实验所需要的环境与软件。

 本次实验不需要撰写实验报告，但是需要自己设置好GitHub账号与自己的git连接。并且把自己的初始化后的仓库地址发送给助教老师。

实验目的

1. 了解操作系统开发实验环境
2. 学会使用Ubuntu操作系统
3. 熟悉命令行方式的编译、调试工程
4. 掌握基于硬件模拟器的调试技术
5. 学会使用基本的开发工具
6. 掌握RISCV-64汇编语言

实验内容

1. 阅读实验指导书内容以及涉及的官方文档或者教程。
2. 对于实验指导书未涉及的内容能自行动手查询资料学习。
3. 安装Ubuntu操作系统。
4. 安装推荐的开发工具，若对某些工具情有独钟亦可。
5. 安装依赖包、硬件模拟器、调试工具后尝试进行联合调试。

前导知识

该模块是LAB0的第一个模块，主要就是阅读学习相关的知识，对OS实验有一个基本的了解，看一下我们接下来一个学期到底要做什么，要实现什么样的一个东西，而这个东西又是基于什么基础慢慢搭建起来的。要搭建房子需要各种工具，工欲善其事必先利其器，所以要学会使用主要的开发调试工具，能让我们实验过程变得非常愉快。

最后由于我们是基于RISC-V指令集来完成实验，当然要掌握这门指令集的所有东西啦，幸运的是，你选择的是最简单的指令集，有没有开心一点呢？

 千万不要嫌接下来的前导知识杂没有用哦，答应我，一定要认认真真的看哈~

了解uCore

2006年, MIT的Frans Kaashoek等人参考PDP-11上的UNIX Version 6写了一个可在x86指令集架构上运行的操作系统xv6 (基于MIT License)。

2010年, 清华大学操作系统教学团队参考MIT的教学操作系统xv6, 开发了在x86指令集架构上运行的操作系统ucore, 多年来作为操作系统课程的实验框架使用, 已经成为了大家口中的"祖传实验".

ucore麻雀虽小, 五脏俱全。在不超过5k的代码量中包含虚拟内存管理、进程管理、处理器调度、同步互斥、进程间通信、文件系统等主要内核功能, 充分体现了“小而全”的指导思想。

ucore的运行环境可以是真实的计算机 (包括小型智能设备)。一开始ucore是运行在x86指令集架构上的, 到了如今, x86指令集架构的问题渐渐开始暴露出来。虽然在PC平台上占据绝对主流, 但出于兼容性考虑x86架构仍然保留了许多历史包袱, 用于教学的时候总有些累赘。另一方面, 为了更好的和目前5G、物联网技术的发展衔接, 将ucore移植到RISC-V架构势在必行。

了解RISC-V

发明

RISC发明者是美国加州大学伯克利分校教师David Patterson，RISC-V（拼做risk-five）是第五代精简指令集，也是由David Patterson指导的项目。2010年伯克利大学并行计算实验室(Par Lab)的1位教授和2个研究生想要做一个项目，需要选一种计算机架构来做。当时面临的是选择X86、ARM，还是其他指令集，不管选择哪个都或多或少有些问题，比如授权费价格高昂，不能开源，不能扩展更改等等。所以他们在2010年5月开始规划自己做一个新的、开源的指令集，就是RISC-V。

接着时间到2015年，这个指令集在学术界已经开始出名了，这时为了更好的推动这个指令集在技术和商业上的发展，3位创始人大佬做了下面两件事情。

技术方向，成立RISC-V基金会，维护指令集架构的完整性和非碎片化。

商业方向，成立SiFive公司，推动RISC-V的商业化。

基金会会员

目前加入RISC-V基金会的中国企业和机构有：阿里巴巴、华为、中科院计算所、华米科技、智芯科技、浪潮等。（具体会员名单可以在[这里](#)查看）

特点

设计哲学-简单就是美

1. 无病一身轻——架构的篇幅。目前的“RISC-V架构文档”分为“指令集文档”（[riscv-spec-v2.2.pdf](#)）和“特权架构文档”（[riscv-privileged-v1.10.pdf](#)）。“指令集文档”的篇幅为145页，而“特权架构文档”的篇幅也仅为91页。熟悉体系结构的工程师仅需一至两天便可将

其通读，虽然“RISC-V的手册”还在不断地丰富，但是相比“x86的架构文档”与“ARM的架构文档”，RISC-V的篇幅可以说是极其短小精悍。

2. 能屈能伸——模块化的指令集。RISC-V架构相比其他成熟的商业架构的最大一个不同还在于它是一个模块化的架构。因此，RISC-V架构不仅短小精悍，而且其不同的部分还能以模块化的方式组织在一起，从而试图通过一套统一的架构满足各种不同的应用。这种模块化是x86与ARM架构所不具备的。以ARM的架构为例，ARM的架构分为A、R和M三个系列，分别针对于Application（应用操作系统）、Real-Time（实时）和Embedded（嵌入式）三个领域，彼此之间并不兼容。模块化的RISC-V架构能够使得用户能够灵活选择不同的模块组合，以满足不同的应用场景，可以说是“老少咸宜”
3. 浓缩的都是精华——指令的数量。短小精悍的架构以及模块化的哲学，使得RISC-V架构的指令数目非常的简洁。基本的RISC-V指令数目仅有40多条，加上其他的模块化扩展指令总共几十条指令。

 一定要仔细阅读《RISC-V手册》哦！！

指令集简介

1. 模块化的指令子集。RISC-V的指令集使用模块化的方式进行组织，每一个模块使用一个英文字母来表示。RISC-V最基本也是唯一强制要求实现的指令集部分是由I字母表示的基本整数指令子集，使用该整数指令子集，便能够实现完整的软件编译器。其他的指令子集部分均为可选的模块，具有代表性的模块包括M/A/F/D/C。
2. 规整的指令编码。RISC-V的指令集编码非常的规整，指令所需的通用寄存器的索引（Index）都被放在固定的位置。因此指令译码器（Instruction Decoder）可以非常便捷的译码出寄存器索引然后读取通用寄存器组（Register File，Regfile）。
3. 优雅的压缩指令子集。基本的RISC-V基本整数指令子集（字母I表示）规定的指令长度均为等长的32位，这种等长指令定义使得仅支持整数指令子集的基本RISC-V CPU非常容易设计。但是等长的32位编码指令也会造成代码体积（Code Size）相对较大的问题。为了满足某些对于代码体积要求较高的场景（譬如嵌入式领域），RISC-V定义了一种可选的压缩（Compressed）指令子集，由字母C表示，也可以由RVC表示。RISC-V具有后发优势，从一开始便规划了压缩指令，预留了足够的编码空间，16位长指令与普通的32位长指令可以无缝自由地交织在一起，处理器也没有定义额外的状态。
4. 特权模式。RISC-V架构定义了三种工作模式，又称特权模式（Privileged Mode）：
Machine Mode：机器模式，简称M Mode。Supervisor Mode：监督模式，简称S Mode。User Mode：用户模式，简称U Mode。RISC-V架构定义M Mode为必选模式，另外两种为可选模式。通过不同的模式组合可以实现不同的系统。

5. 自定制指令扩展。除了上述阐述的模块化指令子集的可扩展、可选择，RISC-V架构还有一个非常重要的特性，那就是支持第三方的扩展。用户可以扩展自己的指令子集，RISC-V预留了大量的指令编码空间用于用户的自定义扩展，同时，还定义了四条Custom指令可供用户直接使用，每条Custom指令都有几个比特位的子编码空间预留，因此，用户可以直接使用四条Custom指令扩展出几十条自定义的指令。

其他特点

可配置的通用寄存器组、简洁的存储器访问指令、高效的分支跳转指令、简洁的子程序调用、无条件码执行、无分支延迟槽、简洁的运算指令。



友情链接

[RISC-V基金会](#)

[中国开放指令生态（RISC-V）联盟](#)

[赛昉科技](#)

[终于有人把RISC-V讲明白了](#)

[什么是RISC-V](#)

了解OS实验

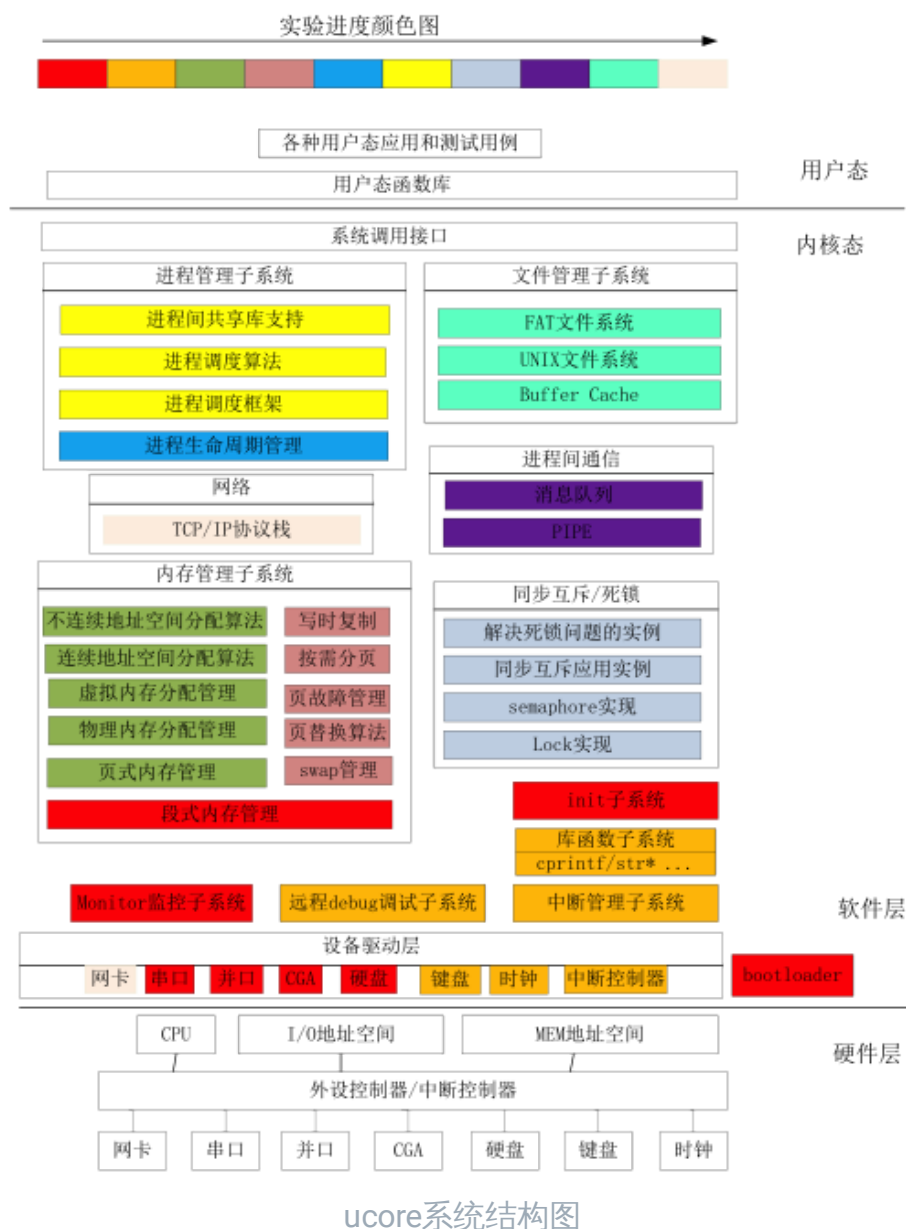
写一个操作系统难吗？别被现在上百万行的Linux和Windows操作系统吓倒。当年Thompson趁他老婆带着小孩度假留他一人在家时，写了UNIX；当年Linus还是一个21岁大学生时完成了Linux雏形。站在这些巨人的肩膀上，我们能否也尝试一下做“巨人”的滋味呢？

实验内容

那我们准备如何一步一步来实现ucore呢？根据一个操作系统的设计实现过程，我们可以有如下的实验步骤：

1. 启动操作系统的bootloader:OpenSBI。了解操作系统启动前的状态和要做的准备工作，了解运行操作系统的硬件支持，操作系统如何加载到内存中，理解两类中断-“外设中断”，“陷阱中断”等；
2. 物理内存管理子系统。用于理解RISC-V分段/分页模式，了解操作系统如何管理物理内存；
3. 虚拟内存管理子系统。通过页表机制和换入换出（swap）机制，以及中断-“故障中断”、缺页故障处理等，实现基于页的内存替换算法；
4. 内核线程子系统。用于了解如何创建相对与用户进程更加简单的内核态线程，如果对内核线程进行动态管理等；
5. 用户进程管理子系统。用于了解用户态进程创建、执行、切换和结束的动态管理过程，了解在用户态通过系统调用得到内核态的内核服务的过程；
6. 处理器调度子系统。用于理解操作系统的调度过程和调度算法；
7. 同步互斥与进程间通信子系统。了解进程间如何进行信息交换和共享，并了解同步互斥的具体实现以及对系统性能的影响，研究死锁产生的原因，以及如何避免死锁；
8. 文件系统。了解文件系统的具体实现，与进程管理等的关系，了解缓存对操作系统IO访问的性能改进，了解虚拟文件系统（VFS）、buffer cache和disk driver之间的关系。

其中每个开发步骤都是建立在上一个步骤之上的，就像搭积木，从一个一个小木块，最终搭出来一个小房子。在搭房子的过程中，完成从理解操作系统原理到实践操作系统设计与实现的探索过程。这个房子最终的建筑架构和建设进度如下图所示：




开发OS实验的步骤

本次OS实验大致可以通过如下过程就可以完成使用：

1. 学习相关理论知识
2. 建立LAB实验环境，采用VMware虚拟机的最简单方式完成
3. 阅读本LAB实验指导书，了解实验要求
4. 下载源码
5. 进入各个OS实验工程目录 例如：`cd labcodes/lab1`
6. 根据实验要求阅读并修改代码

7. 编译源码 例如执行：`make`
8. 如果不通过则返回步骤3
9. 如果实现不正确（即看到步骤6的输出存在不是OK的情况）则返回3
10. 如果实现基本正确（即看到6的输出都是OK）则 `push` 到自己github仓库
11. 编写实验报告，发送到助教邮箱，并 `push` 到github仓库

 可以通过 `make qemu` 让OS实验工程在qemu上运行；可以通过 `make debug` 或 `make debug-nox` 命令实现通过gdb远程调试 OS实验工程。

了解实验环境

在实验中，我们使用的系统环境是Ubuntu18.04。在实验过程中，我们需要了解基于命令行方式的编译、调试、运行操作系统的实验方法。为此，需要了解基本的Linux命令行使用。

✔ 当然现在只是了解一下，具体的操作还是要在安装好虚机以后再来哦~

命令模式的基本结构和概念

Ubuntu是图形界面友好和易操作的Linux发行版，但有时只需执行几条简单的指令就可以完成繁琐的鼠标点击才能完成的操作。Linux的命令行操作模式功能可以实现你需要的所有操作。简单的说，命令行就是基于字符命令的用户界面，也被称为文本操作模式。绝大多数情况下，用户通过输入一行或多行命令直接与计算机互动，来实现对计算机的操作。

如何进入命令模式

假设使用默认的图形界面为GNOME的任意版本Ubuntu Linux。点击鼠标右键->终端，就可以启动名为terminal程序，从而可以在此软件界面中进行命令行操作。打开terminal程序后你首先可能会注意到类似下面的界面：

```
kelee@ubuntu:~$
```

你所看到的这些被称为命令终端提示符，它表示计算机已就绪，正在等待着用户输入操作指令。以我的屏幕画面为例，“kelee”是当前所登录的用户名，“ubuntu”是这台计算机的主机名，“~”表示当前目录。此时输入任何指令按回车之后该指令将会提交到计算机运行，比如你可以输入命令：ls 再按下回车：

```
ls [ENTER]
```

注意：[ENTER]是指输入完ls后按下回车键，而不是叫你输入这个单词，ls这个命令将会列出你当前所在目录里的所有文件和子目录列表。

下面介绍bash shell程序的基本使用方法，它是ubuntu缺省的外壳程序。

常用指令

查询文件列表：(ls)

```
1 kelee@ubuntu:~$ ls
2 Desktop      Downloads      Music          Public  Templates
3 Documents    examples.desktop  Pictures      riscv   Videos
```

ls命令默认状态下将按首字母升序列出你当前文件夹下面的所有内容，但这样直接运行所得到的信息也是比较少的，通常它可以结合以下这些参数运行以查询更多的信息：

```
1 ls /          # 将列出根目录 '/' 下的文件清单.如果给定一个参数，则命令行会把该参数当作
2 ls -l         # 将给你列出一个更详细的文件清单。
3 ls -a         # 将列出包括隐藏文件(以.开头的文件)在内的所有文
```

件。]ls -h # 将以KB/MB/GB的形式给出文件大小,而不是以纯粹的Bytes.

查询当前所在目录：(pwd)

```
1 kelee@ubuntu:~$ pwd
2 /home/kelee
```


进入其他目录：(cd)

```
1 kelee@ubuntu:~$ cd riscv
2 kelee@ubuntu:~/riscv$ pwd
3 /home/kelee/riscv
4 kelee@ubuntu:~/riscv$
```

上面例子中，当前目录原来是/home/kelee,执行cd riscv之后再运行pwd可以发现，当前目录已经改为/riscv了。

在屏幕上输出字符：(echo)

```
1 kelee@ubuntu:~$ echo "Hello World"
2 Hello World
```

这是一个很有用的命令，它可以在屏幕上输入你指定的参数("号中的内容)，当然这里举的这个例子中它没有多大的实际意义，但随着你对Linux指令的不断深入，就会发现它的价值所在。

显示文件内容：cat

```
1 kelee@ubuntu:~$ cat tempfile.txt
2 Roses are red.
3 Violets are blue,
4 and you have the bird-flue!
```

也可以使用less或more来显示比较大的文本文件内容。

复制文件：cp

```
1 kelee@ubuntu:~$ cp tempfile.txt tempfile_copy.txt
```

```
2 kelee@ubuntu:~$ cat tempfile_copy.txt
3 Roses are red.
4 Violets are blue,
5 and you have the bird-flue!
```

移动文件 : mv

```
1 kelee@ubuntu:~$ ls
2 Desktop    Downloads      Music    Public  tempfile_copy.txt  Template
3 Documents  examples.desktop  Pictures  riscv   tempfile.txt       Videos
4 kelee@ubuntu:~$ mv tempfile_copy.txt tempfile_mv.txt
5 kelee@ubuntu:~$ ls
6 Desktop    Downloads      Music    Public  tempfile_mv.txt  Templates
7 Documents  examples.desktop  Pictures  riscv   tempfile.txt     Videos
```

注意：在命令操作时系统基本上不会给你什么提示，当然，绝大多数的命令可以通过加上一个参数-v来要求系统给出执行命令的反馈信息；

```
1 kelee@ubuntu:~$ mv -v tempfile_mv.txt tempfile_mv_v.txt
2 renamed 'tempfile_mv.txt' -> 'tempfile_mv_v.txt'
```

建立一个空文本文件 : touch

```
1 kelee@ubuntu:~$ ls
2 Desktop    Downloads      Music    Public  tempfile_mv_v.txt  Template
3 Documents  examples.desktop  Pictures  riscv   tempfile.txt       Videos
4 kelee@ubuntu:~$ touch file1.txt
5 kelee@ubuntu:~$ ls
6 Desktop    examples.desktop  Pictures  tempfile_mv_v.txt  Videos
7 Documents  file1.txt         Public    tempfile.txt
8 Downloads  Music             riscv     Templates
```

建立一个目录 : mkdir

```

1 kelee@ubuntu:~$ ls
2 Desktop    examples.desktop  Pictures  tempfile_mv_v.txt  Videos
3 Documents  file1.txt         Public    tempfile.txt
4 Downloads  Music             riscv     Templates
5 kelee@ubuntu:~$ mkdir test_dir
6 kelee@ubuntu:~$ ls
7 Desktop    examples.desktop  Pictures  tempfile_mv_v.txt  test_dir
8 Documents  file1.txt         Public    tempfile.txt       Videos
9 Downloads  Music             riscv     Templates

```

删除文件/目录 : rm

```

1 kelee@ubuntu:~$ ls
2 Desktop    examples.desktop  Pictures  tempfile_mv_v.txt  test_dir
3 Documents  file1.txt         Public    tempfile.txt       Videos
4 Downloads  Music             riscv     Templates
5 kelee@ubuntu:~$ rm tempfile_mv_v.txt
6 kelee@ubuntu:~$ ls -p
7 Desktop/   examples.desktop  Pictures/  tempfile.txt  Videos/
8 Documents/ file1.txt         Public/    Templates/
9 Downloads/ Music/           riscv/     test_dir/
10 kelee@ubuntu:~$ rm test_dir
11 rm: cannot remove 'test_dir': Is a directory
12 kelee@ubuntu:~$ rm -R test_dir
13 kelee@ubuntu:~$ ls
14 Desktop    Downloads         file1.txt  Pictures  riscv     Templates
15 Documents  examples.desktop  Music     Public    tempfile.txt  Videos

```

在上面的操作：首先我们通过ls命令查询可知当前目下有两个文件和一个文件夹；

- 1 [1] 你可以用参数 `-p`来让系统显示某一项的类型，比如是文件/文件夹/快捷链接等等；
- 2 [2] 接下来我们用`rm -i`尝试删除文件，`-i`参数是让系统在执行删除操作前输出一条确认提示；
- 3 [3] 当我们尝试用上面的命令去删除一个文件夹时会得到错误的提示，因为删除文件夹必须使用

特别提示：在使用命令操作时，系统假设你很明确自己在做什么，它不会给你太多的提示，比如你执行`rm -Rf /`，它将会删除你硬盘上所有的东西，并且不会给你任何提示，所以，尽量在使用命令时加上`-i`的参数，以让系统在执行前进行一次确认，防止你干一些蠢事。如果你觉得每次都要输入`-i`太麻烦，你可以执行以下的命令，让`-i`成为默认参数：

```
alias rm='rm -i'
```

查询当前进程：ps

```
1 kelee@ubuntu:~$ ps
2      PID TTY          TIME CMD
3      3356 pts/0    00:00:00 bash
4      3659 pts/0    00:00:00 ps
```

这条命令会例出你所启动的所有进程；

```
1 ps -a          #可以例出系统当前运行的所有进程，包括由其他用户启动的进程；
2 ps auxww       #是一条相当人性化的命令，它会例出除一些很特殊进程以外的所有进程，并会以
```

基本命令的介绍就到此为止，你可以访问网络得到更加详细的Linux命令介绍。

控制流程

输入/输出

input用来读取你通过键盘（或其他标准输入设备）输入的信息，output用于在屏幕（或其他标准输出设备）上输出你指定的输出内容.另外还有一些标准的出错提示也是通过这个命令来实现的。通常在遇到操作错误时，系统会自动调用这个命令来输出标准错误提示；

我们能重定向命令中产生的输入和输出流的位置。

重定向

如果你想把命令产生的输出流指向一个文件而不是（默认的）终端，你可以使用如下的语句：

```
1 kelee@ubuntu:~$ ls >file2.txt
2 kelee@ubuntu:~$ cat file2.txt
3 Desktop
4 Documents
5 Downloads
6 examples.desktop
7 file1.txt
8 file2.txt
9 Music
10 Pictures
11 Public
12 riscv
13 tempfile.txt
14 Templates
15 Videos
```

以上例子将创建文件file2.txt如果file2.txt不存在的话。注意：如果file2.txt已经存在，那么上面的命令将复盖文件的内容。如果你想将内容添加到已存在的文件内容的最后，那你可以用下面这个语句：

```
command >> filename
```

示例:

```
1 kelee@ubuntu:~$ ls >>file2.txt
2 kelee@ubuntu:~$ cat file2.txt
3 Desktop
4 Documents
5 Downloads
6 examples.desktop
7 file1.txt
8 file2.txt
9 Music
10 Pictures
11 Public
12 riscv
```

```
13 tempfile.txt
14 Templates
15 Videos
16 Desktop
17 Documents
18 Downloads
19 examples.desktop
20 file1.txt
21 file2.txt
22 Music
23 Pictures
24 Public
25 riscv
26 tempfile.txt
27 Templates
28 Videos
```

在这个例子中，你会发现原有的文件中添加了新的内容。接下来我们会见到另一种重定向方式：我们将把一个文件的内容作为将要执行的命令的输入。以下是这个语句：

```
command < filename
```

示例:

```
1 kelee@ubuntu:~$ sort <file2.txt
2 Desktop
3 Desktop
4 Documents
5 Documents
6 Downloads
7 Downloads
8 examples.desktop
9 examples.desktop
10 file1.txt
11 file1.txt
12 file2.txt
13 file2.txt
14 Music
15 Music
16 Pictures
17 Pictures
18 Public
```

```
19 Public
20 riscv
21 riscv
22 tempfile.txt
23 tempfile.txt
24 Templates
25 Templates
26 Videos
27 Videos
```

管道

Linux的强大之处在于它能把几个简单的命令联合成为复杂的功能，通过键盘上的管道符号'|'完成。现在，我们来排序上面的"grep"命令：

```
1 kelee@ubuntu:~$ grep -i 'D' <file2.txt | sort > result.txt
2 kelee@ubuntu:~$ cat result.txt
3 Desktop
4 Desktop
5 Documents
6 Documents
7 Downloads
8 Downloads
9 examples.desktop
10 examples.desktop
11 Videos
12 Videos
```

搜索 file2.txt 中的d字母，将输出分类并写入分类文件到 result.txt 。有时候用ls列出很多命令的时候很不方便 这时“|”就充分利用到了 ls -l | less 慢慢看吧.

后台进程

CLI 不是系统的串行接口。您可以在执行其他命令时给出系统命令。要启动一个进程到后台，追加一个“&”到命令后面。

```
1 sleep 60 &
2 ls
```

睡眠命令在后台运行，您依然可以与计算机交互。除了不同步启动命令以外，最好把 '&' 理解成 ';'。

如果您有一个命令将占用很多时间，您想把它放入后台运行，也很简单。只要在命令运行时按下 ctrl-z，它就会停止。然后键入 bg 使其转入后台。fg 命令可使其转回前台。

```
1 sleep 60
2 <ctrl-z> # 这表示敲入Ctrl+Z键
3 bg
4 fg
```

最后，您可以使用 ctrl-c 来杀死一个前台进程。

环境变量

特殊变量。PATH, PS1, ...

不显示中文

可通过执行如下命令避免显示乱码中文。在一个shell中，执行：

```
export LANG=""
```

这样在这个shell中，output信息缺省时英文。

获得软件包

命令行获取软件包

Ubuntu 下可以使用 apt-get 命令，apt-get 是一条 Linux 命令行命令，适用于 deb 包管理式的操作系统，主要用于自动从互联网软件库中搜索、安装、升级以及卸载软件或者操作系统。一般需要 root 执行权限，所以一般跟随 sudo 命令，如：

```
sudo apt-get install gcc [ENTER]
```

常见的以及常用的 apt 命令有：

- 1 apt-get install <package>
- 2 下载 <package> 以及所依赖的软件包，同时进行软件包的安装或者升级。
- 3 apt-get remove <package>
- 4 移除 <package> 以及所有依赖的软件包。
- 5 apt-cache search <pattern>
- 6 搜索满足 <pattern> 的软件包。
- 7 apt-cache show/showpkg <package>
- 8 显示软件包 <package> 的完整描述。

例如：

```
1 kelee@ubuntu:~$ apt-cache search aptitude
2 aptitude - terminal-based package manager
3 aptitude-common - architecture independent files for the aptitude package
4 aptitude-doc-en - English manual for aptitude, a terminal-based package ma
5 libcwidget-dev - high-level terminal interface library for C++ (developmen
6 apt-cacher - Caching proxy server for Debian/Ubuntu software repositories
7 apticron - Simple tool to mail about pending package updates - cron versio
8 apticron-systemd - Simple tool to mail about pending package updates - sys
9 aptitude-doc-cs - Czech manual for aptitude, a terminal-based package mana
10 aptitude-doc-es - Spanish manual for aptitude, a terminal-based package ma
11 aptitude-doc-fi - Finnish manual for aptitude, a terminal-based package ma
12 aptitude-doc-fr - French manual for aptitude, a terminal-based package man
13 aptitude-doc-it - Italian manual for aptitude, a terminal-based package ma
14 aptitude-doc-ja - Japanese manual for aptitude, a terminal-based package m
15 aptitude-doc-nl - Dutch manual for aptitude, a terminal-based package mana
16 aptitude-doc-ru - Russian manual for aptitude, a terminal-based package ma
17 aptitude-robot - Automate package choice management
```

```
18 cron-apt - automatic update of packages using apt-get
19 cupt - flexible package manager -- console interface
20 gbrainy - brain teaser game and trainer to have fun and to keep your brain
21 pkgsync - automated package list synchronization
22 wajig - unified package management front-end for Debian
23 kelee@ubuntu:~$
```

配置升级源

Ubuntu的软件包获取依赖升级源，通过Software&Updates->Ubuntu Software->Download from:->Other:->China->mirrors.aliyun.com->Choose Server

查找帮助文件

Ubuntu 下提供 man 命令以完成帮助手册得查询。man 是 manual 的缩写，通过 man 命令可以对 Linux 下常用命令、安装软件、以及C语言常用函数等进行查询，获得相关帮助。

例如：

```
1 kelee@ubuntu:~$ man printf
2 PRINTF(1)                                User Commands                                PRINTF
3
4 NAME
5     printf - format and print data
6
7 SYNOPSIS
8     printf FORMAT [ARGUMENT]...
9     printf OPTION
10
11 DESCRIPTION
12     Print ARGUMENT(s) according to FORMAT, or execute according to OPTI
13
14     --help display this help and exit
15
16     --version
17         output version information and exit
18
19     FORMAT controls the output as in C printf.  Interpreted sequences a
20
21     \"    double quote
```

```
22
23     \\      backslash
24
25 Manual page printf(1) line 1 (press h for help or q to quit)
```

通常可能会用到的帮助文件例如：

```
gcc-doc cpp-doc glibc-doc
```

上述帮助文件可以通过 apt-get 命令或者软件包管理器获得。获得以后可以通过 man 命令进行命令或者参数查询。

了解开发调试基本工具


编辑器

Understand

在OS实验网站上有[Understand](#) Windows版的资源。该软件是一个阅读代码的很好工具，可以可视化的看到各个函数之间的调用关系，可以很好的找到函数、变量的定义，具体的使用方法以及介绍可以参考[该教程](#)。但是就编辑代码来说，不建议使用Understand。

VScode

VScode是很好的项目管理、代码编译器工具，集成了git，并且可以安装各类插件支持各种语言，习惯使用visual studio的同学使用起来会非常习惯，具体的下载安装使用方法，可以参考[该教程](#)，值得说明的是，我们在编译的时候需要其他工具联合编译，因此可以仅仅把VScode当成没有感情的写代码工具，不由它来编译运行，编译运行交给终端。

 VScode 快捷键的使用在Windows和Ubuntu上有些不同哦~

写完了代码别忘了格式化代码鸭~看起来好舒服的！

编译器:GCC

在Ubuntu Linux中的C语言编程主要基于GNU C的语法，通过gcc来编译并生成最终执行文件。GNU汇编（assembler）采用的是AT&T汇编格式，Microsoft 汇编采用Intel格式。

编译简单的 C 程序

C语言经典的入门例子是 Hello World，下面是一示例代码：

```
1  #include <stdio.h>
2  int
3  main(void)
4  {
5      printf("Hello, world!\n");
6      return 0;
7  }
```

我们假定该代码存为文件‘hello.c’。要用 gcc 编译该文件，使用下面的命令：

```
$ gcc -Wall hello.c -o hello
```

该命令将文件‘hello.c’中的代码编译为机器码并存储在可执行文件 ‘hello’中。机器码的文件名是通过 -o 选项指定的。该选项通常作为命令行中的最后一个参数。如果被省略，输出文件默认为 ‘a.out’。

注意到如果当前目录中与可执行文件重名的文件已经存在，它将被复盖。选项 -Wall 开启编译器几乎所有常用的警告——**强烈建议你始终使用该选项**。编译器有很多其他的警告选项，但 -Wall 是最常用的。默认情况下GCC不会产生任何警告信息。当编写 C 或 C++ 程序时编译器警告非常有助于检测程序存在的问题。

本例中，编译器使用了 -Wall 选项而没产生任何警告，因为示例程序是完全合法的。

要运行该程序，输入可执行文件的路径如下：

```
1  $ ./hello
2  Hello, world!
```

这可将执行文件载入内存，并使 CPU 开始执行其包含的指令。路径 ./ 指代当前目录，因此 ./hello 载入并执行当前目录下的可执行文件 ‘hello’。

AT&T汇编基本语法

Ucore中用到的是AT&T格式的汇编，与Intel格式的汇编有一些不同。二者语法上主要有以下几个不同：

```
1      * 寄存器命名原则
2          AT&T: %eax                      Intel: eax
3      * 源/目的操作数顺序
4          AT&T: movl %eax, %ebx           Intel: mov ebx, eax
5      * 常数/立即数的格式
6          AT&T: movl $_value, %ebx       Intel: mov eax, _value
7          把value的地址放入eax寄存器
8          AT&T: movl $0xd00d, %ebx       Intel: mov ebx, 0xd00d
9      * 操作数长度标识
10         AT&T: movw %ax, %bx             Intel: mov bx, ax
11      * 寻址方式
12         AT&T:   immed32(basepointer, indexpointer, indexscale)
13         Intel: [basepointer + indexpointer × indexscale + imm32)
```

如果操作系统工作于保护模式下，用的是32位线性地址，所以在计算地址时不用考虑segment:offset的问题。上式中的地址应为：

$$\text{imm32} + \text{basepointer} + \text{indexpointer} \times \text{indexscale}$$

下面是一些例子：

```
1      * 直接寻址
2          AT&T:  foo                      Intel: [foo]
3          boo是一个全局变量。注意加上$是表示地址引用，不加是表示值引用。对于局部
4
5      * 寄存器间接寻址
6          AT&T: (%eax)                   Intel: [eax]
7
8      * 变址寻址
9          AT&T: _variable(%eax)          Intel: [eax + _variable]
10         AT&T: _array( ,%eax, 4)        Intel: [eax × 4 + _array]
11         AT&T: _array(%ebx, %eax, 8)    Intel: [ebx + eax × 8 + _a
```

GCC基本内联汇编

GCC 提供了两内内联汇编语句（inline asm statements）：基本内联汇编语句（basic inline asm statement）和扩展内联汇编语句（extended inline asm statement）。GCC基本内联汇编很简单，一般是按照下面的格式：

```
asm("statements");
```

例如：

```
asm("nop"); asm("cli");
```

"asm" 和 **"asm"** 的含义是完全一样的。如果有多行汇编，则每一行都要加上 "\n\t"。其中的 "\n" 是换行符，"\t" 是 tab 符，在每条命令的结束加这两个符号，是为了让 gcc 把内联汇编代码翻译成一般的汇编代码时能够保证换行和留有一定的空格。对于基本asm语句，GCC编译出来的汇编代码就是双引号里的内容。例如：

```
1      asm( "pushl %eax\n\t"  
2          "movl $0,%eax\n\t"  
3          "popl %eax"  
4      );
```

实际上gcc在处理汇编时，是要把asm(...)的内容"打印"到汇编文件中，所以格式控制字符是必要的。再例如：

```
1      asm("movl %eax, %ebx");  
2      asm("xorl %ebx, %edx");  
3      asm("movl $0, _boo);
```

在上面的例子中，由于我们在内联汇编中改变了 edx 和 ebx 的值，但是由于 gcc 的特殊的处理方法，即先形成汇编文件，再交给 GAS 去汇编，所以 GAS 并不知道我们已经改变了 edx 和 ebx 的值，如果程序的上下文需要 edx 或 ebx 作其他内存单元或变量的暂存，就会产生没

有预料的多次赋值，引起严重的后果。对于变量 `_boo` 也存在一样的问题。为了解决这个问题，就要用到扩展 GCC 内联汇编语法。

GCC扩展内联汇编

使用GCC扩展内联汇编的例子如下：

```
1  #define read_cr0() ({ \
2  unsigned int __dummy; \
3  __asm__( \
4      "movl %%cr0,%0\n\t" \
5      : "=r" (__dummy)); \
6  __dummy; \
7  })
```

它代表什么含义呢？这需要从其基本格式讲起。GCC扩展内联汇编的基本格式是：

```
1  asm [volatile] ( Assembler Template
2      : Output Operands
3      [ : Input Operands
4      [ : Clobbers ] ])
```

其中，**asm** 表示汇编代码的开始，其后可以跟 **volatile**（这是可选项），其含义是避免“asm”指令被删除、移动或组合，在执行代码时，如果不希望汇编语句被 gcc 优化而改变位置，就需要在 asm 符号后添加 volatile 关键词：`asm volatile(...)`；或者更详细地说明为：**asm volatile(...)**；然后就是小括弧，括弧中的内容是具体的内联汇编指令代码。"" 为汇编指令部分，例如，`"movl %%cr0,%0\n\t"`。数字前加前缀“%”，如 %1，%2 等表示使用寄存器的样板操作数。可以使用的操作数总数取决于具体 CPU 中通用寄存器的数量，如 Intel 可以有 8 个。指令中有几个操作数，就说明有几个变量需要与寄存器结合，由 gcc 在编译时根据后面输出部分和输入部分的约束条件进行相应的处理。由于这些样板操作数的前缀使用了“%”，因此，在用到具体的寄存器时就在前面加**两个“%”**，如 `%%cr0`。输出部分（output operand list），用以规定对输出变量（目标操作数）如何与寄存器结合的约束（constraint），输出部分可以有多个约束，互相以逗号分开。每个约束以“=”开头，接着用一个字母来表示操作数的类型，然后是关于变量结合的约束。例如，上例中：


```
: "=r" (__dummy)
```

“=r”表示相应的目标操作数（指令部分的%0）可以使用任何一个通用寄存器，并且变量__dummy 存放在这个寄存器中，但如果是：

```
: "=m" (__dummy)
```

“=m”就表示相应的目标操作数是存放在内存单元__dummy中。表示约束条件的字母很多，下表给出几个主要的约束字母及其含义：

字母	含义
m, v, o	内存单元
R	任何通用寄存器
Q	寄存器eax, ebx, ecx,edx之一
l, h	直接操作数
E, F	浮点数
G	任意
a, b, c, d	寄存器eax/ax/al, ebx/bx/bl, ecx/cx/cl或edx/dx/dl
S, D	寄存器esi或edi
I	常数（0～31）

输入部分（input operand list）：输入部分与输出部分相似，但没有“=”。如果输入部分一个操作数所要求使用的寄存器，与前面输出部分某个约束所要求的是同一个寄存器，那就把对应操作数的编号（如“1”，“2”等）放在约束条件中。在后面的例子中，可看到这种情况。

修改部分（clobber list,也称 乱码列表）:这部分常常以“memory”为约束条件，以表示操作完成后内存中的内容已有改变，如果原来某个寄存器的内容来自内存，那么现在内存中这个单元的内容已经改变。乱码列表通知编译器，有些寄存器或内存因内联汇编块造成乱码，可隐

式地破坏了条件寄存器的某些位（字段）。注意，指令部分为必选项，而输入部分、输出部分及修改部分为可选项，当输入部分存在，而输出部分不存在时，冒号“:”要保留，当“memory”存在时，三个冒号都要保留，例如

```
#define __cli() __asm__ __volatile__("cli": : : "memory")
```

下面是一个例子：

```
1  int count=1;
2  int value=1;
3  int buf[10];
4  void main()
5  {
6      asm(
7          "cld \n\t"
8          "rep \n\t"
9          "stosl"
10         :
11         : "c" (count), "a" (value) , "D" (buf)
12         );
13 }
```

得到的主要汇编代码为：

```
1  movl count,%ecx
2  movl value,%eax
3  movl buf,%edi
4  #APP
5  cld
6  rep
7  stosl
8  #NO_APP
```

cld,rep,stos这几条语句的功能是向buf中写上count个value值。冒号后的语句指明输入，输出和被改变的寄存器。通过冒号以后的语句，编译器就知道你的指令需要和改变哪些寄存

器，从而可以优化寄存器的分配。其中符号"c"(count)指示要把count的值放入ecx寄存器。类似的还有：

```
1  a eax
2  b ebx
3  c ecx
4  d edx
5  S esi
6  D edi
7  I 常数值, (0 - 31)
8  q,r 动态分配的寄存器
9  g eax,ebx,ecx,edx或内存变量
10 A 把eax和edx合成一个64位的寄存器 (use long longs)
```

也可以让gcc自己选择合适的寄存器。如下面的例子：

```
1  asm("leal (%1,%1,4),%0"
2      : "=r" (x)
3      : "0" (x)
4      );
```

这段代码到的主要汇编代码为：


```
1  movl x,%eax
2  #APP
3  leal (%eax,%eax,4),%eax
4  #NO_APP
5  movl %eax,x
```

几点说明：

- [1] 使用q指示编译器从eax, ebx, ecx, edx分配寄存器。使用r指示编译器从eax, ebx, ecx, edx, esi, edi分配寄存器。
- [2] 不必把编译器分配的寄存器放入改变的寄存器列表，因为寄存器已经记住了它们。
- [3] "="是标示输出寄存器，必须这样用。

- [4] 数字%n的用法：数字表示的寄存器是按照出现和从左到右的顺序映射到用"r"或"q"请求的寄存器。如果要重用"r"或"q"请求的寄存器的话，就可以使用它们。
- [5] 如果强制使用固定的寄存器的话，如不用%1，而用ebx，则：

```
1  asm("leal (%%ebx,%%ebx,4),%0"  
2      : "=r" (x)  
3      : "0" (x)  
4  );
```

 注意要使用两个%,因为一个%的语法已经被%n用掉了。

代码维护

make和Makefile

简介

GNU make(简称make)是一种代码维护工具，在大中型项目中，它将根据程序各个模块的更新情况，自动的维护和生成目标代码。

make命令执行时，需要一个 makefile（或Makefile）文件，以告诉make命令需要怎么样的去编译和链接程序。首先，我们用一个示例来说明makefile的书写规则。以便给大家一个感兴认识。这个示例来源于gnu的make使用手册，在这个示例中，我们的工程有8个c文件，和3个头文件，我们要写一个makefile来告诉make命令如何编译和链接这几个文件。我们的规则是：

- 如果这个工程没有编译过，那么我们的所有c文件都要编译并被链接。
- 如果这个工程的某几个c文件被修改，那么我们只编译被修改的c文件，并链接目标程序。
- 如果这个工程的头文件被改变了，那么我们需要编译引用了这几个头文件的c文件，并链接目标程序。

只要我们的makefile写得够好，所有的这一切，我们只用一个make命令就可以完成，make命令会自动智能地根据当前的文件修改的情况来确定哪些文件需要重编译，从而自己编译所需要的文件和链接目标程序。

makefile的规则

在讲述这个makefile之前，还是让我们先来粗略地看一看makefile的规则。

```
1 target ... : prerequisites ...  
2     command  
3     ...  
4     ...
```

target也就是一个目标文件，可以是object file，也可以是执行文件。还可以是一个标签（label）。prerequisites就是，要生成那个target所需要的文件或是目标。command也就是make需要执行的命令（任意的shell命令）。这是一个文件的依赖关系，也就是说，target这一个或多个的目标文件依赖于prerequisites中的文件，其生成规则定义在command中。如果prerequisites中有一个以上的文件比target文件要新，那么command所定义的命令就会被执行。这就是makefile的规则。也就是makefile中最核心的内容。

可以查看GNU手册，或者查看这份[中文教程](#)。

Git

Git是一个开源的分布式版本控制系统，可以有效、高速地处理从很小到非常大的项目版本管理。Git 是 Linus Torvalds 为了帮助管理 Linux 内核开发而开发的一个开放源码的版本控制软件。Git 与Github不一样哦，Git是工具，而GitHub是可以用Git进行管理的远程仓库。

代码层次

1. 你目录中的文件是第一层
2. 缓存区，每次add之后，当前目录中要追踪的文件会作为一个版本会存放在缓存区。注意不是所有的文件。一般一个文件生成之后，会标记为“未追踪”，但是否对其做版本管理还是要选择的。例如一些编译文件就没有必要追踪。对需要做版本管理的问件，用add添加，不需要的用clean删除。

3. 本地仓库，每次commit之后，缓存区最新的版本就会存放在本地仓库。这里要提及一个HEAD的概念。HEAD是当前的版本指向，每次更新或者回退都会修改HEAD的指向，但对仓库中每一个版本并不会删除。所以即使回退到过去还是有机会回到现在的版本的。
4. 远程仓库，每次push之后，会将本地仓库中HEAD所指向的版本存放到远程仓库

常用命令

命令	功能
git init	在本地的当前目录里初始化git仓库
git status	查看当前仓库的状态
git add -A	增加目录中所有的文件到缓存区
git add file	增加相应文件到缓存区
git commit -m "信息"	将缓存区中更改提交到本地仓库
git log	查看当前版本之前的提交记录
git reflog	查看HEAD的变更记录，包括回退
git branch -b branch_name	建立一个新的分支
git diff	查看当前文件与缓存区文件的差异
git checkout -- file	取消更改，将缓存区的文件提取覆盖当前文件
git reset --hard 版本号	回退到相应版本号，同样也可以回退到未来的版本号
git clean -xf	删除当前目录中所有未追踪的文件
git config --global core.quotePath false	处理中文文件名

与Github链接

首先我们认为你已经有一个github的账户。

然后我们要建立SSH链接。这是一种通讯的加密协议。我先在我的笔记本上计算一对公钥和私钥，将公钥存储在github中，这样本地就可以通过SSH与github展开加密通讯。

建立方法，输入命令

```
ssh-keygen -t rsa -C "your_email@youremail.com" //双引号里面是你的常用邮箱
```

输入之后要输入口令，可以不用输入直接按“enter”一路确认就可以了。然后在账户的根目录（/或者/home/你的账户名，具体取决于你执行上述命令时所采用的账户）查找隐藏目录.ssh/id_rsa.pub文件，将当中内容添加到github中。

这样你就可以通过SSH链接到github中了。但是github作为一个远程仓库，你可以链接这个仓库，并保持同步。但是你不能把本地仓库直接上传到github中去。所以你应该先在github中建立一个对应的仓库，然后再在本地建立一个仓库，将两者进行链接，再去写入文件执行版本管理。所用到的命令有

```
1 git remote add origin git@github.com:<用户名>/<仓库名>.git
2 git pull origin master //因为github建立仓库时会有readme.md文件，先要拷贝一份
3 git push -u origin master //将本地仓库链接到master分支上，你当然可以链接到其他分
4 git push//上传你的本地仓库
```

还有一种方法不用分两地建库再去链接。你可以只在github上建库，然后clone到本地目录中。

```
git clone git@github.com:<用户名>/<仓库名>.git
```

VScode中使用

因为VScode是一个集成工具可以直接在VScode中使用Git，用VScode打开已经配置好的仓库，VScode就可以自动读取里面的内容，然后当进行修改后可以通过VScode直接commit与push。具体的操作可以参考该[教程](#)。

调试器:GDB

功能

gdb 是功能强大的调试程序，可完成如下的调试任务：

- 设置断点
- 监视程序变量的值
- 程序的单步(step in/step over)执行
- 显示/修改变量的值
- 显示/修改寄存器
- 查看程序的堆栈情况
- 远程调试
- 调试线程

在可以使用 gdb 调试程序之前，必须使用 -g 或 -ggdb编译选项编译源文件。运行 gdb 调试程序时通常使用如下的命令：

```
gdb progname
```

在 gdb 提示符处键入help，将列出命令的分类，主要的分类有：

- aliases：命令别名
- breakpoints：断点定义；
- data：数据查看；
- files：指定并查看文件；
- internals：维护命令；
- running：程序执行；
- stack：调用栈查看；
- status：状态查看；
- tracepoints：跟踪程序执行。

键入 help 后跟命令的分类名，可获得该类命令的详细清单。

常用命令

命令	功能
break FILENAME:NUM	在特定源文件特定行上设置断点
clear FILENAME:NUM	删除设置在特定源文件特定行上的断点
run	运行调试程序
step	单步执行调试程序，不会直接执行函数
next	单步执行调试程序，会直接执行函数
backtrace	显示所有的调用栈帧。该命令可用来显示函数的调用顺序
where continue	继续执行正在调试的程序
display EXPR	每次程序停止后显示表达式的值,表达式由程序定义的变量组成
file FILENAME	装载指定的可执行文件进行调试
help CMDNAME	显示指定调试命令的帮助信息
info break	显示当前断点列表，包括到达断点处的次数等
info files	显示被调试文件的详细信息
info func	显示被调试程序的所有函数名称
info prog	显示被调试程序的执行状态
info local	显示被调试程序当前函数中的局部变量信息
info var	显示被调试程序的所有全局和静态变量名称
kill	终止正在被调试的程序
list	显示被调试程序的源代码
quit	退出 gdb

窗口相关命令

用gdb查看源代码可以用list命令，但是这个不够灵活。可以使用"layout src"命令，或者按Ctrl-X再按A，就会出现一个窗口可以查看源代码。也可以用使用-tui参数，这样进入gdb里面后就能直接打开代码查看窗口。其他代码窗口相关命令：

命令	功能
info win	显示窗口的大小
layout next	切换到下一个布局模式
layout prev	切换到上一个布局模式
layout src	只显示源代码
layout asm	只显示汇编代码
layout split	显示源代码和汇编代码
layout regs	增加寄存器内容显示
focus cmd/src/asm/regs/next/prev	切换当前窗口
refresh	刷新所有窗口
tui reg next	显示下一组寄存器
tui reg system	显示系统寄存器
update	更新源代码窗口和当前执行点
winheight name +/- line	调整name窗口的高度
tabset nchar	设置tab为nchar个字符

示例

下面以一个有错误的例子程序来介绍gdb的使用：

```
1  /*bugging.c*/
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  static char buff [256];
6  static char* string;
7  int main ()
8  {
9      printf ("Please input a string: ");
10     gets (string);
11     printf ("\nYour string is: %s\n", string);
12 }
```

这个程序是接受用户的输入，然后将用户的输入打印出来。该程序使用了一个未经过初始化的字符串地址 `string`，因此，编译并运行之后，将出现 "Segment Fault" 错误：

```
1  $ gcc -o bugging -g  bugging.c
2  $ ./bugging
3  Please input a string: asdf
4  Segmentation fault (core dumped)
```

为了查找该程序中出现的問題，我們利用 `gdb`，并按如下的步骤进行：

[1] 运行 “`gdb bugging`”，加载 `bugging` 可执行文件；

```
$gdb bugging
```

[2] 执行装入的 `bugging` 命令；

```
(gdb) run
```

[3] 使用 `where` 命令查看程序出错的地方；

```
(gdb) where
```

[4] 利用 list 命令查看调用 gets 函数附近的代码；

```
(gdb) list
```

[5] 在 gdb 中，我们在第 11 行处设置断点，看看是否是在第11行出错；

```
(gdb) break 11
```

[6] 程序重新运行到第 11 行处停止，这时程序正常，然后执行单步命令next；

```
(gdb) next
```

[7] 程序确实出错，能够导致 gets 函数出错的因素就是变量 string。重新执行测试程，用 print 命令查看 string 的值；

```
1 (gdb) run
2 (gdb) print string
3 (gdb) $1=0x0
```

[8] 问题在于string指向的是一个无效指针，修改程序，在10行和11行之间增加一条语句“string=buff;”，重新编译程序，然后继续运行，将看到正确的程序运行结果。

了解硬件模拟器

简介

我们有了操作系统的代码，那要在哪里去运行呢。我们当然可以像计算机组成原理一样去烧制一块RISC-v架构的开发板，然后去debug。虽然这样可以，但duck不必，使用模拟器会使我们的实验更加方便。模拟器就是在计算机上通过软件模拟一个RISC-v架构的硬件平台，从而能够运行RISC-v的目标代码。

模拟器有很多，但我们为了方便，选择的是QEMU模拟器，的优点在于，内置了一套OpenSBI固件的实现，可以简化我们的代码。

常用命令

help	查看 qemu 帮助，显示所有支持的命令。
q、quit、exit	退出 qemu。
stop	停止 qemu。
c、cont、continue	连续执行。
x /fmt addr xp /fmt addr	显示内存内容，其中 'x' 为虚地址，'xp' 为实地址。 参数 /fmt i 表示反汇编，缺省参数为前一次参数。
p、print	计算表达式值并显示，例如 \$reg 表示寄存器结果。
memsave addr size file pmemsave addr size file	将内存保存到文件，memsave 为虚地址，pmemsave 为实地址。
breakpoint 相关：	设置、查看以及删除 breakpoint，pc执行到 breakpoint，qemu 停止。（暂时没有此功能）
watchpoint 相关：	设置、查看以及删除 watchpoint, 当 watchpoint 地址内容被修改，停止。（暂时没有此功能）

s、step	单步一条指令，能够跳过断点执行。
r、registers	显示全部寄存器内容。
info 相关操作	查询 qemu 支持的关于系统状态信息的操作。

其他具体的命令格式以及说明，参见 `qemu help` 命令帮助。

配置环境

看了这么多理论知识，终于可以动手开始实验啦。相信经过大二的学习，很多同学都会明白环境的重要性。在这一模块我们需要实现以下东西。

1. 安装好Ubuntu虚拟系统，当然也可以是双系统，但不建议哈。太麻烦了。大家伙对Ubuntu可能不太熟悉，所以指导了大家安装了一些小工具，能够让大家用起来舒服一些。
2. 安装开发工具。这里大家就要小心啦，涉及到编译、运行的东西一定咬谨慎哦。
3. 安装硬件模拟器。我们的系统是需要运行在RISC-v架构的计算机上的，但是可以用软件模拟，为什么要用硬件呢，成本还低。所以要安装好硬件模拟器噢~
4. 我们写出来的代码不一定立马是对的，所以需要调试工具与硬件模拟器一起来联合调试。

安装虚拟环境

安装Ubuntu

1. 首先[下载](#)并安装VMware15 客户端（VMware只支持Windows与Linux，MAC需要自己上网查找虚拟机安装教程）。
2. 到清华镜像站[下载](#)Ubuntu 18.04.3 的镜像文件。
3. 具体的安装步骤可参考这个[教程](#)。



1. 不建议大家使用最新版的Ubuntu系统哦，因为可能会有一些玄学问题。
2. 记得根据【了解实验环境】里面的配置升级源哦。

安装小工具

aptitude

```
sudo apt get install aptitude
```

这个工具是安装软件的一个工具，可以自己解决包依赖问题，之后安装可以直接使用

```
sudo aptitude install $APP ($APP 为要安装的软件名字)
```

gnome-tweaks

```
sudo aptitude install gnome-tweaks
```


有的同学可能会感觉Ubuntu的字体太小，可以安装gnome-tweaks，来调整哦。具体的使用方法，自己研究哈。

搜狗输入法

默认的Ubuntu调中文还是比较麻烦，可以到搜狗输入法官网下载。

v2rayL

在某些时候，我们不得已需要科学上网的时候，可以使用一些工具，Windows的工具很多，但Linux的工具很少，可以前往[这里](#)下载一个工具。具体的安装步骤自行领会。

安装开发工具

设置环境变量

方便起见，可以先在终端里设置一个叫做**RISCV**的环境变量(在bash命令里可以通过**\$RISCV**使用)，作为你安装所有和riscv有关的软件的路径。在 `/etc/profile` 里面写一行

```
export RISCV=/your/path/to/riscv
```

之类的东西就行。后面安装的各个项目最好也放在上面的路径里面。当然需要去创建这个文件夹。

最小的软件开发环境需要：能够编译程序，能够运行程序。开发操作系统这样的系统软件也不例外。

安装VScode

进入其官网进行下载安装即可，与Windows类似。

安装git

```
sudo aptitude install git
```

git下载完后记得初始化噢~并且与自己的github连接起来。

安装编译器

我们使用的计算机都是基于x86架构的。如何把程序编译到riscv64架构的汇编？这需要我们使用“目标语言为riscv64机器码的编译器”，在我们的电脑上进行**交叉编译**。


放心，这里不需要你自己写编译器。我们使用现有的riscv-gcc编译器即可。从

<https://github.com/riscv/riscv-gcc> clone下来，然后在x86架构上编译riscv-gcc编译器为可执行的x86程序，就可以运行它，来把你的程序源代码编译成riscv架构的可执行文件了。这有点像绕口令，但只要有一点编译原理的基础就可以理解。不过，这个riscv-gcc仓库很大，而且自己编译工具链总是一件麻烦的事。

其实，没必要那么麻烦，我们大可以使用别人已经编译好的编译器的可执行文件，也就是所谓的**预编译 (prebuilt)** 工具链，下载下来，放在你喜欢的地方（比如之前定义的**\$RISCV**），配好路径（把编译器的位置加到系统的**PATH**环境变量里），就能在终端使用了。我们推荐使用sifive公司提供的预编译工具链，[下载“GNU Embedded Toolchain”](#)。然后解压到之前的riscv文件夹下，把里面的bin文件夹加入到环境变量。修改完记得运行

```
source /etc/profile 噢。
```

配置好后，在终端输入 `riscv64-unknown-elf-gcc -v` 查看安装的gcc版本, 如果输出一大堆东西且最后一行有 `gcc version 某个数字.某个数字.某个数字`，说明gcc配置成功，否则需要检查一下哪里做错了，比如环境变量**PATH**配置是否正确。一般需要把一个形如 `....bin` 的目录加到**PATH**里。

 可能有人会说，到底该怎么去做嘛，烦人，都没有写完。没有写完的，剩一点点的需要自己去查资料，自己实现哦~

安装硬件模拟器

下载安装QEMU

我们也像安装编译器一样，去sifive的官网[下载](#)Ubuntu系统的riscv-qemu，里面的很多设置都已经设置好了的。在下载完，然后解压到之前的riscv文件夹下，把里面的bin文件夹加入到环境变量。修改完记得重启虚拟机噢。（这次熟练多了吧~）。

使用OpenSBI

新版 Qemu 中内置了 OpenSBI 固件（firmware），它主要负责在操作系统运行前的硬件初始化和加载操作系统的功能。我们使用以下命令尝试运行一下：

```
qemu-system-riscv64 --machine virt --nographic --bios default
```

如果成功的话，就可以看到。

```
1  OpenSBI v0.5 (Oct  9 2019 12:03:04)
2
3  / _ _ \          / _ _ _ | _ \ _ _ |
4  | | | | | _ _ _ _ _ _ _ _ | ( _ _ | | ) | | |
5  | | | | | ' _ \ / _ \ ' _ \ \ _ _ \ | _ < | |
6  | | _ | | | _ ) | _ / | | | _ _ _ ) | | _ | |
7  \ _ _ _ / | . _ / \ _ _ | | | | _ _ _ / | _ _ _ / _ _ _ |
8      | |
9      | _ |
10
11 Platform Name       : QEMU Virt Machine
12 Platform HART Features : RV64ACDFIMSU
13 Platform Max HARTs   : 8
14 Current Hart        : 0
15 Firmware Base       : 0x80000000
16 Firmware Size       : 116 KB
17 Runtime SBI Version  : 0.2
18
19 PMP0: 0x0000000080000000-0x000000008001ffff (A)
```

```
20 PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
```

可以看到我们已经在 `qemu-system-riscv64` 模拟的 `virt machine` 硬件上将 `OpenSBI` 这个固件跑起来了。Qemu 可以使用 `ctrl+a` 再按下 `x` 退出（注意要松开 `ctrl` 再单独按 `x`）。

如果无法正常使用 Qemu，可以尝试下面这个命令。

```
$ sudo sysctl vm.overcommit_memory=1
```

安装调试工具

编译

其实说安装有点不准确，因为之前已经把GDB已经悄悄的装进了环境变量里。其实就是在我们要安装编译器的时候我们只需要打开 lab0 的文件夹，然后打开终端，输入 make ，就可以进行编译了

```
1  $ make
2  + cc kern/init/entry.S
3  + cc kern/init/init.c
4  + cc kern/libs/stdio.c
5  + cc kern/driver/console.c
6  + cc libs/string.c
7  + cc libs/printfmt.c
8  + cc libs/readline.c
9  + cc libs/sbi.c
10 + ld bin/kernel
11 riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img
```

可以看到多了几个文件夹，我们之后再介绍这些。

修改脚本

工具链（之前下载的编译器里面的工具）里提供了相应的gdb工具，要使用gdb，就需要对脚本进行修改。在源代码的文件夹中，找到lab0中的Makefile文件然后找到

```
CC := $(GCCPREFIX)gcc 在后面添加-g。
```

使用GDB进行调试

因为gdb和qemu是两个应用不能直接交流，比较常用的方法是以tcp进行通讯，也就是让qemu在localhost::1234端口上等待。

在 lab0 文件夹下打开终端，运行

```
1 $ qemu-system-riscv64 -S -s -hda ./bin/ucore.img
2 WARNING: Image format was not specified for './bin/ucore.img' and probing
3     Automatically detecting the format is dangerous for raw images, w
4     Specify the 'raw' format explicitly to remove the restrictions.
5 VNC server running on 127.0.0.1:5900xxxxxxxxx qemu-system-riscv64 -S -s -
```

然后在该文件夹下重新打开一个终端，运行

```
1 $ riscv64-unknown-elf-gdb ./bin/kernel
2 GNU gdb (SiFive GDB 8.3.0-2020.04.0) 8.3
3 Copyright (C) 2019 Free Software Foundation, Inc.
4 License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.ht
5 This is free software: you are free to change and redistribute it.
6 There is NO WARRANTY, to the extent permitted by law.
7 Type "show copying" and "show warranty" for details.
8 This GDB was configured as "--host=x86_64-linux-gnu --target=riscv64-unkno
9 Type "show configuration" for configuration details.
10 For bug reporting instructions, please see:
11 <https://github.com/sifive/freedom-tools/issues>.
12 Find the GDB manual and other documentation resources online at:
13 <http://www.gnu.org/software/gdb/documentation/>.
14
15 For help, type "help".
16 Type "apropos word" to search for commands related to "word"...
17 Reading symbols from ./bin/kernel...
18 (gdb)
```

接着连接qemu：

```
1 (gdb) target remote :1234
2 Remote debugging using :1234
3 0x0000000000000100 in ?? ()
```

连接成功输入si就可以进行运行下一条指令，

```
1 (gdb) si
2 0x00000000000001004 in ?? ()
```

这样就代表可以正常运行了噢，具体调试步骤与方法，看一下前面哈。

LAB0.5：最小可执行内核

相对于上百万行的现代操作系统(linux, windows), 几千行的ucore是一只"麻雀"。但这只麻雀依然是一只胖麻雀，我们一眼看不过来几千行的代码。所以，我们要再做简化，先用好刀法，片掉麻雀的血肉, 搞出一个"麻雀骨架"，看得通透，再像组装哪吒一样，把血肉安回去，变成一个活生生的麻雀。

lab0.5是lab1的预备，我们构建一个最小的可执行内核（"麻雀骨架"），它能够进行格式化的输出，然后进入死循环。

实验目的

逐步掌握以下过程：

1. 源码是如何被编译成可执行文件的。
2. 编译成可执行文件后，计算机如何加载操作系统。
3. 加载以后，该从哪里去运行操作系统。
4. 操作系统的输出信息是怎么输出的呢。

实验内容

1. 跟着实验指导书的步伐，阅读框架代码。
2. 结合框架代码，深刻理解RISC-v。
3. 内核的内存布局和入口点设置
4. 通过sbi封装好输入输出函数
5. 借助bootloader:OpenSBI初始化OS，完成练习。
6. 按要求撰写实验报告。

练习

练习1：理解通过make生成执行文件的过程

列出本实验各练习中对应的OS原理的知识点，并说明本实验中的实现部分如何对应和体现了原理中的基本概念和关键知识点。

在此练习中，大家需要通过静态分析代码来了解：

1. lab0.5/Makefile，解释操作系统镜像文件ucore.img是如何一步一步生成的？(需要比较详细地解释Makefile中每一条相关命令和命令参数的含义，以及说明命令导致的结果)
2. 阅读分析lab0.5/tools/kernel.ld 链接脚本，给出其每行含义。
3. 一个被系统认为是符合规范的硬盘主引导扇区的特征是什么？



查看linkscript、装载位置(base address)和对其地址(align))

练习2：分析OpenSBI加载bin格式的OS的过程

- OpenSBI如何读取硬盘扇区的？
- OpenSBI是如何加载bin格式的OS？

内存布局

计算机组成

首先我们回顾计算机的组成:

CPU, 存储设备（粗略地说，包括断电后遗失的内存，和断电后不遗失的硬盘），输入输出设备，总线。

现在我们手里的东西有：QEMU会帮助我们模拟一块riscv64的CPU，一块物理内存，还会借助你的电脑的键盘和显示屏来模拟命令行的输入和输出。虽然QEMU不会真正模拟一堆线缆，但是总线的通信功能也在QEMU内部实现了。

还差什么呢？硬盘。

OpenSBI

我们需要硬盘上的程序和数据。比如崭新的windows电脑里C盘已经被占据的二三十GB空间，除去预装的应用软件，还有一部分是windows操作系统的内核。在插上电源开机之后，就需要运行操作系统的内核，然后由操作系统来管理计算机。

问题在于，操作系统作为一个程序，必须加载到内存里才能执行。而“把操作系统加载到内存里”这件事情，不是操作系统自己能做到的，就好像你不能拽着头发把自己拽离地面。

因此我们可以想象，在操作系统执行之前，必然有一个其他程序执行，他作为“先锋队”，完成“把操作系统加载到内存”这个工作，然后他功成身退，把CPU的控制权交给操作系统。

这个“其他程序”，我们一般称之为bootloader. 很好理解：他负责boot(开机)，还负责load(加载OS到内存里)，所以叫bootloader.

在QEMU模拟的riscv计算机里，我们使用QEMU自带的bootloader: OpenSBI固件。



知识点

在计算机中，**固件(firmware)**是一种特定的计算机软件，它为设备的特定硬件提供低级控制，也可以进一步加载其他软件。固件可以为设备更复杂的软件（如操作系统）提供标准化的操作环境。对于不太复杂的设备，固件可以直接充当设备的完整操作系统，执行所有控制、监视和数据操作功能。在基于 x86 的计算机系统中, BIOS 或 UEFI 是固件；在基于 riscv 的计算机系统中，OpenSBI 是固件。OpenSBI运行在**M态 (M-mode)**，因为固件需要直接访问硬件。

RISCV有四种**特权级 (privilege level)**。

Level	Encoding	全称	简称
0	00	User/Application	U
1	01	Supervisor	S
2	10	Reserved(目前未使用，保留)	
3	11	Machine	M

粗略的分类：

U-mode是用户程序、应用程序的特权级，S-mode是操作系统内核的特权级，M-mode是固件的特权级。

详细内容请自行查阅RISC-v手册。

elf与bin

我们可以想象这样的过程：操作系统的二进制可执行文件被OpenSBI加载到内存中，然后OpenSBI会把CPU的"当前指令指针"(pc, program counter)跳转到内存里的一个位置，开始执行内存中那个位置的指令。

OpenSBI怎样知道把操作系统加载到内存的什么位置？总不能随便选个位置。也许你会觉得可以把操作系统的代码总是加载到固定的位置，比如总是加载到内存地址最高的地方。

问题在于，之后OpenSBI还要把CPU的program counter跳转到一个位置,开始操作系统的执行。如果加载操作系统到内存里的时候随便加载，那么OpenSBI怎么知道把program counter跳转到哪里去呢？难道操作系统的二进制可执行文件需要提供“program counter跳转到哪里”这样的信息？

实际上，操作系统的二进制可执行文件，会指定它自己应该被加载到内存的哪个地址。而OpenSBI会很听话地把二进制可执行文件放到她想去的位置上。但是关于program counter的跳转，OpenSBI是独断专行的，总是会把program counter跳到 `0x80200000` 这个内存地址开始执行, 所以故事(版本1)其实是这样的：

OpenSBI: 操作系统，你到0x8020000等着program counter跳过来执行！

操作系统：好的！请把我加载到xxxxxx这个位置，这样program counter跳过来的时候就不会出问题了。

实际上，二进制程序加载到内存中是一件很精细的工作。一个二进制程序包括很多section, 如text(程序代码)，bss(需要初始化为零的数据)，rodata(只读数据)。二进制程序的每个section都可以指定一个希望被加载到的内存地址。

故事可以是这样的吗？（版本2）

OpenSBI: 操作系统，你到0x8020000等着program counter跳过来执行！

操作系统：好的！请把我的text section加载到A位置，data section加载到B位置，rodata section加载到C位置.....这样program counter跳过来的时候就不会出问题了！

OpenSBI: 你说啥？

两个版本的故事是因为，我们有两种不同的可执行文件格式：`elf` (e是executable的意思，l是linkable的意思，f是format的意思)和 `bin` (binary)。

`elf` 文件([wikipedia: elf](#))比较复杂，包含一个文件头(ELF header), 包含冗余的调试信息，指定程序每个section的内存布局，需要解析program header才能知道各段(section)的信息。如果我们已经有一个完整的操作系统来解析elf文件，那么elf文件可以直接执行。但是对于OpenSBI来说，elf格式还是太复杂了，把操作系统内核的elf文件交给OpenSBI就会发生版本2的悲惨故事。

`bin` 文件就比较简单了，简单地在文件头之后解释自己应该被加载到什么起始位置。
OpenSBI可以理解得很清楚，这就是版本1的故事。

我们举一个例子解释elf和bin文件的区别：初始化为零的一个大数组，在elf文件里是bss数据段的一部分，只需要记住这个数组的起点和终点就可以了，等到加载到内存里的时候分配那一段内存。但是在bin文件里，那个数组有多大，有多少个字节的0，bin文件就要对应有多少个零。所以如果一个程序里声明了一个大全局数组（默认初始化为0），那么可能编译出来的elf文件只有几KB, 而生成bin文件之后却有几MB, 这是很正常的。实际上，可以认为bin文件会把elf文件指定的每段的内存布局都映射到一块线性的数据里，这块线性的数据（或者说程序）加载到内存里就符合elf文件之前指定的布局。

那么我们的任务就明确了：得到内存布局合适的elf文件，然后把它转化成bin文件（这一步通过objcopy实现），然后加载到QEMU里运行（QEMU自带的OpenSBI会干这个活）。下面我们来看如何设置elf文件的内存布局。

链接脚本

gnu工具链中，包含一个链接器 `ld`

如果你很好奇，可以看[linker script的详细语法](#)

链接器的作用是把输入文件(往往是 .o 文件)链接成输出文件(往往是 elf 文件)。一般来说，输入文件和输出文件都有很多 section, 链接脚本(linker script)的作用，就是描述怎样把输入文件的 section 映射到输出文件的 section, 同时规定这些 section 的内存布局。

如果你不提供链接脚本，ld 会使用默认的一个链接脚本，这个默认的链接脚本适合链接出一个能在现有操作系统下运行的应用程序，但是并不适合链接一个操作系统内核。你可以通过 `ld --verbose` 来查看默认的链接脚本。

 要打开代码看噢，这里没有啦！

我们在链接脚本里把程序的入口点定义为 `kern_entry`，那么我们的程序里需要有一个名称为 `kern_entry` 的符号。我们在 `kern/init/entry.S` 编写了一段汇编代码, 作为整个内核的入口点。

```
1  #include <mmu.h>
2  #include <memlayout.h>
3
4      .section .text,"ax",%progbits
5      .globl kern_entry
6  kern_entry:
7      la sp, bootstacktop
8
9      tail kern_init
10     #调用kern_init，这是我们要用C语言编写的一个函数，tail是riscv伪指令，作用相当于调
11     .section .data
12         # .align 2^12
13         .align PGSHIFT
14         .global bootstack
15     bootstack:
16         .space KSTACKSIZE
17         .global bootstacktop
```

里面有很多符号和指令，定义的符号，应该知道去哪找吧（偷偷告诉你吧，在头文件呀！）。关于指令呢，就需要自己去查阅手册咯。如果看不懂汇编代码结构的，自己查资料噢~不要怀着疑惑往下做。会越来越懵。

诶诶，看到这，下一步该干嘛，该去找哪个呢，想一想噢。

哎鸭！调用了唯一的一个函数，当然是去找函数啦。这个函数在干嘛呢？不好奇么。那就往下看。

真正的入口点

我们在 `kern/init/init.c` 编写函数 `kern_init`，作为“真正的”内核入口点。为了让我们能看到一些效果，我们希望它能在命令行进行格式化输出。

如果我们在linux下运行一个C程序，需要格式化输出，那么大一学生都知道我们应该 `#include<stdio.h>`。于是我们在 `kern/init/init.c` 也这么写一句。且慢！linux下，当我们调用C语言标准库的函数时，实际上依赖于 `glibc` 提供的运行时环境，也就是一一定程度上依赖于操作系统提供的支持。可是我们并没有把 `glibc` 移植到ucore里！

怎么办呢？只能自己动手，丰衣足食。QEMU里的OpenSBI固件提供了输入一个字符和输出一个字符的接口，我们一会把这个接口一层层封装起来，提供 `stdio.h` 里的格式化输出函数 `cprintf()` 来使用。这里格式化输出函数的名字不使用原先的 `printf()`，强调这是我们在ucore里重新实现的函数。

接下来就去看看，我们是怎么从OpenSBI的接口一层层封装到格式化输入输出函数的。

从SBI到stdio

OpenSBI作为运行在M态的软件（或者说固件），提供了一些接口供我们编写内核的时候使用。

我们可以通过 `ecall` 指令(environment call)调用OpenSBI。通过寄存器传递给OpenSBI一个“调用编号”，如果编号在 `0-8` 之间，则由OpenSBI进行处理，否则交由我们自己的中断处理程序处理（暂未实现）。有时OpenSBI调用需要像函数调用一样传递参数，这里传递参数的方式也和函数调用一样，按照riscv的函数调用约定(calling convention)把参数放到寄存器里。可以阅读[SBI的详细文档](#)。

✓ 知识点

ecall(environment call)，当我们在 S 态执行这条指令时，会触发一个 `ecall-from-s-mode-exception`，从而进入 M 模式中的中断处理流程（如设置定时器等）；当我们在 U 态执行这条指令时，会触发一个 `ecall-from-u-mode-exception`，从而进入 S 模式中的中断处理流程（常用来进行系统调用）。

关于这个，大三的时候会被好好折磨的噢【坏笑】。

C语言并不能直接调用 `ecall`，需要通过内联汇编来实现。

```
1 // libs/sbi.c
2 #include <sbi.h>
3 #include <defs.h>
4
5 //SBI编号和函数的对应
6 uint64_t SBI_SET_TIMER = 0;
7 uint64_t SBI_CONSOLE_PUTCHAR = 1;
8 uint64_t SBI_CONSOLE_GETCHAR = 2;
9 uint64_t SBI_CLEAR_IPI = 3;
10 uint64_t SBI_SEND_IPI = 4;
11 uint64_t SBI_REMOTE_FENCE_I = 5;
12 uint64_t SBI_REMOTE_SFENCE_VMA = 6;
13 uint64_t SBI_REMOTE_SFENCE_VMA_ASID = 7;
14 uint64_t SBI_SHUTDOWN = 8;
15 //sbi_call函数是我们关注的核心
```

```

16 uint64_t sbi_call(uint64_t sbi_type, uint64_t arg0, uint64_t arg1, uint64_t
17     uint64_t ret_val;
18     __asm__ volatile (
19         "mv x17, %[sbi_type]\n"
20         "mv x10, %[arg0]\n"
21         "mv x11, %[arg1]\n"
22         "mv x12, %[arg2]\n"    //mv操作把参数的数值放到寄存器里
23         "ecall\n"            //参数放好之后，通过ecall，交给OpenSBI来执行
24         "mv %[ret_val], x10"
25         //OpenSBI按照riscv的calling convention,把返回值放到x10寄存器里
26         //我们还需要自己通过内联汇编把返回值拿到我们的变量里
27         : [ret_val] "=r" (ret_val)
28         : [sbi_type] "r" (sbi_type), [arg0] "r" (arg0), [arg1] "r" (arg1),
29         : "memory"
30     );
31     return ret_val;
32 }
33
34 void sbi_console_putchar(unsigned char ch) {
35     sbi_call(SBI_CONSOLE_PUTCHAR, ch, 0, 0); //注意这里ch隐式类型转换为int64_t
36 }
37
38 void sbi_set_timer(unsigned long long stime_value) {
39     sbi_call(SBI_SET_TIMER, stime_value, 0, 0);
40 }

```

知识点

函数调用与calling convention

我们知道，编译器将高级语言源代码翻译成汇编代码。对于汇编语言而言，在最简单的编程模型中，所能够利用的只有指令集中提供的指令、各通用寄存器、CPU 的状态、内存资源。那么，在高级语言中，我们进行一次函数调用，编译器要做哪些工作利用汇编语言来实现这一功能呢？

显然并不是仅用一条指令跳转到被调用函数开头地址就行了。我们还需要考虑：

- 如何传递参数？
- 如何传递返回值？
- 如何保证函数返回后能从我们期望的位置继续执行？

等更多事项。通常编译器按照某种规范去翻译所有的函数调用，这种规范被称为 **calling convention**。值得一提的是，为了实现函数调用，我们需要预先分配一块内存作为 **调用栈**，后面会看到调用栈在函数调用过程中极其重要。你也可以理解为什么第一章刚开始我们就要分配栈了。

可以参考 [riscv calling convention](#)

现在可以输出一个字符了，有了第一个，就会有第二个第三个.....第无数个。

这样我们就可以通过 `sbi_console_putchar()` 来输出一个字符。接下来我们要做的事情就像月饼包装，把它封了一层又一层。

`console.c` 只是简单地封装一下

```
1 // kern/driver/console.c#include <sbi.h>#include <console.h>
2 void cons_putc(int c) { sbi_console_putchar((unsigned char)c); }
```

`stdio.c` 里面实现了一些函数，注意我们已经实现了ucore版本的puts函数: `cputs()`

```
1 // kern/libs/stdio.c
2 #include <console.h>
3 #include <defs.h>
4 #include <stdio.h>
5
6 /* HIGH level console I/O */
7
8 /* *
9  * cputch - writes a single character @c to stdout, and it will
10 * increace the value of counter pointed by @cnt.
11 * */
12 static void cputch(int c, int *cnt) {
13     cons_putc(c);
14     (*cnt)++;
15 }
16 /* cputchar - writes a single character to stdout */
17 void cputchar(int c) { cons_putc(c); }
18
19 int cputs(const char *str) {
```

```

20     int cnt = 0;
21     char c;
22     while ((c = *str++) != '\0') {
23         cputch(c, &cnt);
24     }
25     cputch('\n', &cnt);
26     return cnt;
27 }

```

我们还在 `libs/printfmt.c` 实现了一些复杂的格式化输入输出函数。最后得到的 `cprintf()` 函数仍在 `kern/libs/stdio.c` 定义，功能和C标准库的 `printf()` 基本相同。

可能你注意到我们用到一个头文件 `defs.h`，我们在里面定义了一些有用的宏和类型

```

1  // libs/defs.h
2  #ifndef __LIBS_DEFS_H__
3  #define __LIBS_DEFS_H__
4  ...
5  /* Represents true-or-false values */
6  typedef int bool;
7  /* Explicitly-sized versions of integer types */
8  typedef char int8_t;
9  typedef unsigned char uint8_t;
10 typedef short int16_t;
11 typedef unsigned short uint16_t;
12 typedef int int32_t;
13 typedef unsigned int uint32_t;
14 typedef long long int64_t;
15 typedef unsigned long long uint64_t;
16 ...
17 /* *
18  * Rounding operations (efficient when n is a power of 2)
19  * Round down to the nearest multiple of n
20  * */
21 #define ROUNDDOWN(a, n) ({                                \
22     size_t __a = (size_t)(a);                            \
23     (typeof(a))(__a - __a % (n));                        \
24 })
25 ...
26 #endif

```

`printfmt.c` 还依赖一个头文件 `riscv.h`, 这个头文件主要定义了若干和riscv架构相关的宏, 尤其是将一些内联汇编的代码封装成宏, 使得我们更方便地使用内联汇编来读写寄存器。当然这里我们还没有用到它的强大功能。

```
1 // libs/riscv.h
2 ...
3 #define read_csr(reg) ({ unsigned long __tmp; \
4     asm volatile ("csrr %0, " #reg : "=r"(__tmp)); \
5     __tmp; })
6 //通过内联汇编包装了 csrr 指令为 read_csr() 宏
7 #define write_csr(reg, val) ({ \
8     if (__builtin_constant_p(val) && (unsigned long)(val) < 32) \
9         asm volatile ("csrw " #reg ", %0" :: "i"(val)); \
10    else \
11        asm volatile ("csrw " #reg ", %0" :: "r"(val)); })
12 ...
```

到现在, 我们已经看过了一个最小化的内核的各个部分, 虽然一些部分没有逐行细读, 但我们也知道它在做什么。

是不是感觉好麻烦啊! 输出一个字符都那么麻烦。那是肯定的噢, 可以稍微喘下气, 脑子里回忆一下, 我们是怎么一层一层剥开, 又是如何一层一层包装的。好玩吧!

但一直到现在我们还没进行过编译。下面就把它编译一下跑起来。

编译运行

我们需要：编译所有的源代码，把目标文件链接起来，生成elf文件，生成bin硬盘镜像，用qemu跑起来

这一系列复杂的命令，我们不想每次用到的时候都敲一遍，所以我们使用魔改的祖传 Makefile 。

我们的 Makefile 还依赖 tools/function.mk

在源代码的根目录下 `make qemu`，我们就把ucore跑起来了。

它输出一行 `(THU.CST) os is loading`，然后进入死循环。

关于Makefile的语法, 如果不熟悉, 可以回看LAB0的前导知识。

项目组成与执行流

lab0的项目组成:

```
1  — Makefile
2  |— kern
3     |— debug
4         |— assert.h
5         |— kdebug.c
6         |— kdebug.h
7         |— kmonitor.c
8         |— kmonitor.h
9         |— panic.c
10        |— stab.h
11     |— driver
12         |— clock.c
13         |— clock.h
14         |— console.c
15         |— console.h
16         |— intr.c
17         |— intr.h
18         |— kbdreg.h
19         |— picirq.c
20         |— picirq.h
21     |— init
22         |— entry.S
23         |— init.c
24     |— libs
25         |— readline.c
26         |— stdio.c
27     |— mm
28         |— memlayout.h
29         |— mmu.h
30         |— pmm.c
31         |— pmm.h
32     |— trap
33         |— trap.c
34         |— trap.h
35         |— trapentry.S
36 |— libs
37     |— defs.h
38     |— elf.h
39     |— error.h
40     |— printfmt.c
41     |— riscv.h
42     |— sbi.c
```

```
43 | |— sbi.h
44 | |— stdarg.h
45 | |— stdio.h
46 | |— string.c
47 | |— string.h
48 |— tools
49 |— function.mk
50 |— kernel.ld
```

内核启动

`kern/init/entry.S` : OpenSBI启动之后将要跳转到的一段汇编代码。在这里进行内核栈的分配，然后转入C语言编写的内核初始化函数。

`kern/init/init.c` : C语言编写的内核入口点。主要包含 `kern_init()` 函数，从 `kern/entry.S` 跳转过来完成其他初始化工作。

设备驱动

`kern/driver/console.c(h)` : 在QEMU上模拟的时候，唯一的“设备”是虚拟的控制台，通过OpenSBI接口使用。简单封装了OpenSBI的字符读写接口，向上提供给输入输出库。

库文件

`libs/riscv.h` : 以宏的方式，定义了riscv指令集的寄存器和指令。如果在C语言里使用riscv指令，需要通过内联汇编和寄存器的编号。这个头文件把寄存器编号和内联汇编都封装成宏，使得我们可以用类似函数的方式在C语言里执行一句riscv指令。

`libs/sbi.c(h)` : 封装OpenSBI接口为函数。如果想在C语言里使用OpenSBI提供的接口，需要使用内联汇编。这个头文件把OpenSBI的内联汇编调用封装为函数。

`libs/defs.h` : 定义了一些常用的类型和宏。例如 `bool` 类型 (C语言不自带, 这里 `typedef int bool`)。

`libs/string.c(h)` : 一些对字符数组进行操作的函数, 如 `memset()`, `memcpy()` 等, 类似C语言的 `string.h` 。

`kern/libs/stdio.c` , `libs/readline.c` , `libs/printfmt.c` : 实现了一套标准输入输出, 功能类似于C语言的 `printf()` 和 `getchar()` 。需要内核为输入输出函数提供两个桩函数 (stub): 输出一个字符的函数, 输入一个字符的函数。在这里, 是 `cons_getc()` 和 `cons_putc()` 。

`kern/errors.h` : 定义了一些内核错误类型的宏。

编译、链接脚本

`tools/kernel.ld` : ucore的链接脚本(link script), 告诉链接器如何将目标文件的section组合为可执行文件。

`tools/function.mk` : 定义Makefile中使用的一些函数

`Makefile` : GNU make编译脚本

执行流

最小可执行内核的执行流为:

加电 -> OpenSBI启动 -> 跳转到 `0x80200000` (`kern/init/entry.S`) -> 进入 `kern_init()` 函数 (`kern/init/init.c`) -> 调用 `cprintf()` 输出一行信息 -> 结束

`cprintf()` 函数的执行流为:

接受一个格式化字符串和若干个需要输出的变量作为参数 -> 解析格式化的字符串，把需要输出的各种变量转化为一串字符 -> 调用 `console.c` 提供的字符输出接口依次输出所有字符
(实际上 `console.c` 又封装了 `sbi.c` 向上提供的OpenSBI接口)

LAB1：中断机制

中断 (interrupt) 机制，就是不管CPU现在手里在干啥活，收到“中断”的时候，都先放下来去处理其他事情，处理完其他事情可能再回来干手头的活。

例如，CPU要向磁盘发一个读取数据的请求，由于磁盘速度相对CPU较慢，在“发出请求”到“收到磁盘数据”之间会经过很多时间周期，如果CPU干等着磁盘干活就相当于CPU在磨洋工。因此我们可以让CPU发出读数据的请求后立刻开始干另一件事情。但是，等一段时间之后，磁盘的数据取到了，而CPU在干其他的事情，我们怎么办才能让CPU知道之前发出的磁盘请求已经完成了呢？我们可以让磁盘给CPU一个“中断”，让CPU放下手里的事情来接受磁盘的数据。

再比如，为了保证CPU正在执行的程序不会永远运行下去，我们需要定时检查一下它是否已经运行“超时”。想象有一个程序由于bug进入了死循环，如果CPU一直运行这个程序，那么其他的所有程序都会因为等待CPU资源而无法运行，造成严重的资源浪费。但是检查是否超时，需要CPU执行一段代码，也就是让CPU暂停当前执行的程序。我们不能假设当前执行的程序会主动地定时让出CPU，那么就需要CPU定时“打断”当前程序的执行，去进行一些处理，这通过时钟中断来实现。

从这些描述我们可以看出，中断机制需要软硬件一起来支持。硬件进行中断和异常的发现，然后交给软件来进行处理。回忆一下组成原理课程中学到的各个控制寄存器以及他们的用途（下一小节会进行简单回顾），这些寄存器构成了重要的**硬件/软件接口**。由此，我们也可以得到在一般OS中进行中断处理支持的方法：

- 编写相应的中断处理代码
- 在启动中正确设置控制寄存器
- CPU捕获异常
- 控制转交给相应中断处理代码进行处理
- 返回正在运行的程序

由于中断处理需要进行较高权限的操作，中断处理程序一般处于**内核态**，或者说，处于“比被打断的程序更高的特权级”。注意，在RISC-V里，中断(interrupt)和异常(exception)统称为“trap”。

这次实验就一起来看一下ucore是如何支持中断处理的。

实验目的

1. 了解CPU的中断机制
2. 了解RISC-v架构是如何支持CPU中断的
3. 掌握与软件相关的中断处理
4. 掌握时钟中断管理

实验内容

1. 跟着实验指导书理解lab1框架代码。
2. 阅读RISC-V手册有关中断部分。
3. 完成练习。
4. 撰写并提交实验报告。

练习

练习1：描述处理中断异常的流程

像LAB0.5里的执行流一样描述ucore是如何处理中断异常的。从异常的产生开始。

练习2：对于任何中断，都需要保存所有寄存器吗？为什么？

练习3：触发、捕获、处理异常

编程：在任意位置触发一条非法指令异常（如：mret），在 `kern/trap/trap.c` 的异常处理函数中捕获，并对其进行处理，简单输出异常类型和指令即可。

RISC-V中断相关

寄存器

除了32个通用寄存器之外，RISCV架构还有大量的 **控制状态寄存器 Control and Status Registers(CSRs)**。其中有几个重要的寄存器和中断机制有关。

有些时候，禁止CPU产生中断很有用。（就像你在做重要的事情，如操作系统lab的时候，并不想被打断）。所以，`sstatus` 寄存器(Supervisor Status Register)里面有一个二进制位 `SIE` (supervisor interrupt enable，在RISCV标准里是 2^1 对应的二进制位)，数值为0的时候，如果当程序在S态运行，将禁用全部中断。（对于在U态运行的程序，SIE这个二进制位的数值没有任何意义），`sstatus` 还有一个二进制位 `UIE` (user interrupt enable)可以在置零的时候禁止用户态程序产生中断。

在中断产生后，应该有个**中断处理程序**来处理中断。CPU怎么知道中断处理程序在哪？实际上，RISCV架构有个CSR叫做 `stvec` (Supervisor Trap Vector Base Address Register)，即所谓的“中断向量表基址”。中断向量表的作用就是把不同种类的中断映射到对应的中断处理程序。如果只有一个中断处理程序，那么可以让 `stvec` 直接指向那个中断处理程序的地址。

对于RISCV架构，`stvec` 会把最低位的两个二进制位用来编码一个“模式”，如果是“00”就说明更高的SXLEN-2个二进制位存储的是唯一的**中断处理程序的地址**(SXLEN是 `stval` 寄存器的位数)，如果是“01”说明更高的SXLEN-2个二进制位存储的是**中断向量表基址**，通过不同的异常原因来索引中断向量表。但是怎样用62个二进制位编码一个64位的地址？RISCV架构要求这个地址是四字节对齐的，总是在较高的62位后补两个0。

手册P110

机器和监管者自陷向量 (trap-vector) 基地址寄存器 (`mtvec`和 `stvec`) CSR。他们是位宽为 XLEN的读 /写寄存器，用于保存自陷向量的配置，包括向量基址 (BASE) 和向量模式 (MODE)。BASE域中的值必须按 4字节对齐。MODE = 0 表示所有异常都把 PC设置为 BASE。MODE = 1会在一部中断时将 PC设置为 $(BASE + (E \times cause))$ 。

当我们触发中断进入 S 态进行处理时，以下寄存器会被硬件自动设置，将一些信息提供给中断处理程序：

sepc(supervisor exception program counter)，它会记录触发中断的那条指令的地址；

scause，它会记录中断发生的原因，还会记录该中断是不是一个外部中断；

stval，它会记录一些中断处理所需要的辅助信息，比如指令获取(instruction fetch)、访存、缺页异常，它会记录发生问题的目标地址或者出错的指令记录下来，这样我们在中断处理程序中就知道处理目标了。

特权指令

RISCV支持以下和中断相关的特权指令：

ecall(environment call)，当我们在 S 态执行这条指令时，会触发一个 ecall-from-s-mode-exception，从而进入 M 模式中的中断处理流程（如设置定时器等）；当我们在 U 态执行这条指令时，会触发一个 ecall-from-u-mode-exception，从而进入 S 模式中的中断处理流程（常用来进行系统调用）。

sret，用于 S 态中断返回到 U 态，实际作用为 $pc \leftarrow sepc$ ，回顾**sepc**定义，返回到通过中断进入 S 态之前的地址。

ebreak(environment break)，执行这条指令会触发一个断点中断从而进入中断处理流程。

mret，用于 M 态中断返回到 S 态或 U 态，实际作用为 $pc \leftarrow mepc$ ，回顾**sepc**定义，返回到通过中断进入 M 态之前的地址。（一般不用涉及）



关于上面提及的内容，要去手册里找相关内容，然后看明白！

上下文处理


我们已经知道,在发生中断的时候,CPU会跳到 `stvec` .我们准备采用 `Direct` 模式,也就是只有一个中断处理程序, `stvec` 直接跳到中断处理程序的入口点,那么需要我们对 `stvec` 寄存器做初始化.

上下文

中断的处理需要“放下当前的事情但之后还能回来接着之前往下做”，对于CPU来说，实际上只需要把原先的寄存器保存下来，做完其他事情把寄存器恢复回来就可以了。这些寄存器也被叫做CPU的**context(上下文，情境)**。

我们要用汇编实现上下文切换(context switch)机制，这包含两步：

- 保存CPU的寄存器（上下文）到内存中（栈上）
- 从内存中（栈上）恢复CPU的寄存器

 通用寄存器的介绍见中文手册42页。

为了方便我们组织上下文的数据（几十个寄存器），我们定义一个结构体。

```
1 // kern/trap/trap.h
2 #ifndef __KERN_TRAP_TRAP_H__
3 #define __KERN_TRAP_TRAP_H__
4
5 #include <defs.h>
6
7 struct pushregs {
8     uintptr_t zero; // Hard-wired zero
9     uintptr_t ra;    // Return address
10    uintptr_t sp;    // Stack pointer
11    uintptr_t gp;    // Global pointer
12    uintptr_t tp;    // Thread pointer
13    uintptr_t t0;    // Temporary
```

```

14     uintptr_t t1;    // Temporary
15     uintptr_t t2;    // Temporary
16     uintptr_t s0;    // Saved register/frame pointer
17     uintptr_t s1;    // Saved register
18     uintptr_t a0;    // Function argument/return value
19     uintptr_t a1;    // Function argument/return value
20     uintptr_t a2;    // Function argument
21     uintptr_t a3;    // Function argument
22     uintptr_t a4;    // Function argument
23     uintptr_t a5;    // Function argument
24     uintptr_t a6;    // Function argument
25     uintptr_t a7;    // Function argument
26     uintptr_t s2;    // Saved register
27     uintptr_t s3;    // Saved register
28     uintptr_t s4;    // Saved register
29     uintptr_t s5;    // Saved register
30     uintptr_t s6;    // Saved register
31     uintptr_t s7;    // Saved register
32     uintptr_t s8;    // Saved register
33     uintptr_t s9;    // Saved register
34     uintptr_t s10;   // Saved register
35     uintptr_t s11;   // Saved register
36     uintptr_t t3;    // Temporary
37     uintptr_t t4;    // Temporary
38     uintptr_t t5;    // Temporary
39     uintptr_t t6;    // Temporary
40 };
41
42 struct trapframe {
43     struct pushregs gpr;
44     uintptr_t status; //sstatus
45     uintptr_t epc;    //sepc
46     uintptr_t badvaddr; //sbadvaddr
47     uintptr_t cause;  //scause
48 };
49
50 void trap(struct trapframe *tf);

```

C语言里面的结构体，是若干个变量在内存里直线排列。也就是说，一个 `trapFrame` 结构体占据36个 `uintptr_t` 的空间（在64位RISCV架构里我们定义 `uintptr_t` 为64位无符号整数），里面依次排列通用寄存器 `x0` 到 `x31` ,然后依次排列4个和中断相关的CSR, 我们希望中断处理程序能够利用这几个CSR的数值。

保存上下文

我们在理论课上也学到了保存上下文是用汇编语言实现的。首先我们定义一个汇编宏 `SAVE_ALL`，用来保存所有寄存器到栈顶（实际上把一个 `trapFrame` 结构体放到了栈顶）。

```
1  # kern/trap/trapentry.S
2  #include <riscv.h>
3
4  .macro SAVE_ALL #定义汇编宏
5
6  csrw sscratch, sp #保存原先的栈顶指针到sscratch
7
8  addi sp, sp, -36 * REGBYTES #REGBYTES是riscv.h定义的常量，表示一个寄存器占
9  #让栈顶指针向低地址空间延伸 36个寄存器的空间，可以放下一个trapFrame结构体。
10 #除了32个通用寄存器，我们还要保存4个和中断有关的CSR
11
12 #依次保存32个通用寄存器。但栈顶指针需要特殊处理。
13 #因为我们想在trapFrame里保存分配36个REGBYTES之前的sp
14 #也就是保存之前写到sscratch里的sp的值
15 STORE x0, 0*REGBYTES(sp)
16 STORE x1, 1*REGBYTES(sp)
17 STORE x3, 3*REGBYTES(sp)
18 STORE x4, 4*REGBYTES(sp)
19 STORE x5, 5*REGBYTES(sp)
20 STORE x6, 6*REGBYTES(sp)
21 STORE x7, 7*REGBYTES(sp)
22 STORE x8, 8*REGBYTES(sp)
23 STORE x9, 9*REGBYTES(sp)
24 STORE x10, 10*REGBYTES(sp)
25 STORE x11, 11*REGBYTES(sp)
26 STORE x12, 12*REGBYTES(sp)
27 STORE x13, 13*REGBYTES(sp)
28 STORE x14, 14*REGBYTES(sp)
29 STORE x15, 15*REGBYTES(sp)
30 STORE x16, 16*REGBYTES(sp)
31 STORE x17, 17*REGBYTES(sp)
32 STORE x18, 18*REGBYTES(sp)
33 STORE x19, 19*REGBYTES(sp)
34 STORE x20, 20*REGBYTES(sp)
35 STORE x21, 21*REGBYTES(sp)
36 STORE x22, 22*REGBYTES(sp)
37 STORE x23, 23*REGBYTES(sp)
38 STORE x24, 24*REGBYTES(sp)
39 STORE x25, 25*REGBYTES(sp)
40 STORE x26, 26*REGBYTES(sp)
41 STORE x27, 27*REGBYTES(sp)
42 STORE x28, 28*REGBYTES(sp)
```

```

43     STORE x29, 29*REGBYTES(sp)
44     STORE x30, 30*REGBYTES(sp)
45     STORE x31, 31*REGBYTES(sp)
46     # RISC-V不能直接从CSR写到内存，需要csrr把CSR读取到通用寄存器，再从通用寄存器STORE到内存
47     csrrw s0, sscratch, x0
48     csrr s1, sstatus
49     csrr s2, sepc
50     csrr s3, sbadaddr
51     csrr s4, scause
52
53     STORE s0, 2*REGBYTES(sp)
54     STORE s1, 32*REGBYTES(sp)
55     STORE s2, 33*REGBYTES(sp)
56     STORE s3, 34*REGBYTES(sp)
57     STORE s4, 35*REGBYTES(sp)
58     .endm #汇编宏定义结束

```

恢复上下文

然后是恢复上下文的汇编宏，恢复的顺序和当时保存的顺序反过来，先加载两个CSR, 再加载通用寄存器。

```

1  # kern/trap/trapentry.S
2  .macro RESTORE_ALL
3
4  LOAD s1, 32*REGBYTES(sp)
5  LOAD s2, 33*REGBYTES(sp)
6
7  # 注意之前保存的几个CSR并不都需要恢复
8  csrw sstatus, s1
9  csrw sepc, s2
10
11 # 恢复sp之外的通用寄存器，这时候还需要根据sp来确定其他寄存器数值保存的位置
12 LOAD x1, 1*REGBYTES(sp)
13 LOAD x3, 3*REGBYTES(sp)
14 LOAD x4, 4*REGBYTES(sp)
15 LOAD x5, 5*REGBYTES(sp)
16 LOAD x6, 6*REGBYTES(sp)
17 LOAD x7, 7*REGBYTES(sp)
18 LOAD x8, 8*REGBYTES(sp)
19 LOAD x9, 9*REGBYTES(sp)
20 LOAD x10, 10*REGBYTES(sp)
21 LOAD x11, 11*REGBYTES(sp)

```

```

22  LOAD x12, 12*REGBYTES(sp)
23  LOAD x13, 13*REGBYTES(sp)
24  LOAD x14, 14*REGBYTES(sp)
25  LOAD x15, 15*REGBYTES(sp)
26  LOAD x16, 16*REGBYTES(sp)
27  LOAD x17, 17*REGBYTES(sp)
28  LOAD x18, 18*REGBYTES(sp)
29  LOAD x19, 19*REGBYTES(sp)
30  LOAD x20, 20*REGBYTES(sp)
31  LOAD x21, 21*REGBYTES(sp)
32  LOAD x22, 22*REGBYTES(sp)
33  LOAD x23, 23*REGBYTES(sp)
34  LOAD x24, 24*REGBYTES(sp)
35  LOAD x25, 25*REGBYTES(sp)
36  LOAD x26, 26*REGBYTES(sp)
37  LOAD x27, 27*REGBYTES(sp)
38  LOAD x28, 28*REGBYTES(sp)
39  LOAD x29, 29*REGBYTES(sp)
40  LOAD x30, 30*REGBYTES(sp)
41  LOAD x31, 31*REGBYTES(sp)
42  # 最后恢复sp
43  LOAD x2, 2*REGBYTES(sp)
44  .endm

```

中断入口

真正的入口点就是去调用这两个宏定义

```

1      .globl __alltraps
2
3  .align(2) #中断入口点 __alltraps必须四字节对齐
4  __alltraps:
5      SAVE_ALL #保存上下文
6
7      move  a0, sp #传递参数。
8      #按照RISCV calling convention, a0寄存器传递参数给接下来调用的函数trap。
9      #trap是trap.c里面的一个C语言函数，也就是我们的中断处理程序
10     jal trap
11     #trap函数指向完之后，会回到这里向下继续执行__trapret里面的内容，RESTORE_ALL,
12
13     .globl __trapret
14  __trapret:
15     RESTORE_ALL

```



```
16     # return from supervisor call
17     sret
```

我们可以看到， `trapentry.S` 这个中断入口点的作用是保存和恢复上下文，并把上下文包装成结构体送到trap函数那里去。下面我们就去看看trap函数里面做些什么。

中断处理程序

scause

当处理自陷时，cause CSR中被写入一个指示导致自陷的事件的代码。如果自陷由中断引起，则置上中断位。“异常代码”字段包含指示最后一个异常的代码。具体的中断/异常映射关系，见中文手册100页。

初始化

中断处理需要初始化，所以我们在 `init.c` 里调用一些初始化的函数

```
1 // kern/init/init.c
2 #include <trap.h>
3 int kern_init(void) {
4     extern char edata[], end[];
5     memset(edata, 0, end - edata);
6
7     cons_init(); // init the console
8
9     const char *message = "(THU.CST) os is loading ...\n";
10    cprintf("%s\n\n", message);
11
12    print_kerninfo();
13
14    // grade_backtrace();
15
16    //trap.h的函数，初始化中断
17    idt_init(); // init interrupt descriptor table
18
19    //clock.h的函数，初始化时钟中断
20    clock_init();
21    //intr.h的函数，使能中断
22    intr_enable();
23
24    // LAB1: CHALLENGE 1 If you try to do it, uncomment lab1_switch_test()
25    // user/kernel mode switch test
26    // lab1_switch_test();
27    /* do nothing */
28    while (1)
```

```

29         ;
30     }
31     // kern/trap/trap.c
32     void idt_init(void) {
33         extern void __alltraps(void);
34         //约定：若中断前处于S态，sscratch为0
35         //若中断前处于U态，sscratch存储内核栈地址
36         //那么之后就可以通过sscratch的数值判断是内核态产生的中断还是用户态产生的中断
37         //我们现在是内核态所以给sscratch置零
38         write_csr(sscratch, 0);
39         //我们保证__alltraps的地址是四字节对齐的，将__alltraps这个符号的地址直接写到s
40         write_csr(stvec, &__alltraps);
41     }
42     //kern/driver/intr.c
43     #include <intr.h>
44     #include <riscv.h>
45     /* intr_enable - enable irq interrupt, 设置sstatus的Supervisor中断使能位 */
46     void intr_enable(void) { set_csr(sstatus, SSTATUS_SIE); }
47     /* intr_disable - disable irq interrupt */
48     void intr_disable(void) { clear_csr(sstatus, SSTATUS_SIE); }

```

处理

trap.c的中断处理函数trap, 实际上把中断处理,异常处理的工作分发给了

interrupt_handler(), exception_handler(), 这些函数再根据中断或异常的不同类型来处理。

```

1  // kern/trap/trap.c
2  /* trap_dispatch - dispatch based on what type of trap occurred */
3  static inline void trap_dispatch(struct trapframe *tf) {
4      //scause的最高位是1, 说明trap是由中断引起的
5      if ((intptr_t)tf->cause < 0) {
6          // interrupts
7          interrupt_handler(tf);
8      } else {
9          // exceptions
10         exception_handler(tf);
11     }
12 }
13
14 /* *
15  * trap - handles or dispatches an exception/interrupt. if and when trap()

```

```
16  * returns,  
17  * the code in kern/trap/trapentry.S restores the old CPU state saved in t  
18  * trapframe and then uses the ired instruction to return from the excepti  
19  * */  
20  void trap(struct trapframe *tf) { trap_dispatch(tf); }
```

`interrupt_handler()` 和 `exception_handler()` 的实现还比较简单，只是简单地根据 `scause` 的数值更仔细地分了下类，做了一些输出就直接返回了。`switch`里的各种case, 如 `IRQ_U_SOFT`, `CAUSE_USER_ECALL`, 是riscv ISA 标准里规定的。我们在 `riscv.h` 里定义了这些常量。我们接下来主要关注时钟中断的处理。

在这里我们对时钟中断进行了一个简单的处理，即每次触发时钟中断的时候，我们会给一个计数器加一，并且设定好下一次时钟中断。当计数器加到100的时候，我们会输出一个 `100ticks` 表示我们触发了100次时钟中断。通过在模拟器中观察输出我们即刻看到是否正确触发了时钟中断，从而验证我们实现的异常处理机制。

时钟中断


时钟中断需要CPU硬件的支持。CPU以“时钟周期”为工作的基本时间单位，对逻辑门的时序电路进行同步。

我们的“时钟中断”实际上就是“每隔若干个时钟周期执行一次的程序”。

“若干个时钟周期”是多少个？太短了肯定不行。如果时钟中断处理程序需要100个时钟周期执行，而你每50个时钟周期就触发一个时钟中断，那么间隔时间连一个完整的时钟中断程序都跑不完。如果你200个时钟周期就触发一个时钟中断，那么CPU的时间将有一半消耗在时钟中断，开销太大。一般而言，可以设置时钟中断间隔设置为CPU频率的1%，也就是每秒钟触发100次时钟中断，避免开销过大。

我们用到的RISCV对时钟中断的硬件支持包括：

- OpenSBI提供的 `sbi_set_timer()` 接口，可以传入一个时刻，让它在那个时刻触发一次时钟中断
- `rdtime` 伪指令，读取一个叫做 `time` 的CSR的数值，表示CPU启动之后经过的真实时间。在不同硬件平台，时钟频率可能不同。在QEMU上，这个时钟的频率是10MHz, 每过1s, `rdtime` 返回的结果增大 100000000

 在RISCV32和RISCV64架构中，`time` 寄存器都是64位的。

`rdcycle` 伪指令可以读取经过的时钟周期数目，对应一个寄存器 `cycle`

注意，我们需要“每隔若干时间就发生一次时钟中断”，但是OpenSBI提供的接口一次只能设置一个时钟中断事件。我们采用的方式是：一开始只设置一个时钟中断，之后每次发生时钟中断的时候，设置下一次的时钟中断。

在clock.c里面初始化时钟并封装一些接口

```
1 //libs/sbi.c
2
3 //当time寄存器(rdtimes的返回值)为stime_value的时候触发一个时钟中断
```

```

4 void sbi_set_timer(unsigned long long stime_value) {
5     sbi_call(SBI_SET_TIMER, stime_value, 0, 0);
6 }
7
8 // kern/driver/clock.c
9 #include <clock.h>
10 #include <defs.h>
11 #include <sbi.h>
12 #include <stdio.h>
13 #include <riscv.h>
14
15 //volatile告诉编译器这个变量可能在其他地方被瞎改一通，所以编译器不要对这个变量瞎优化
16 volatile size_t ticks;
17
18 //对64位和32位架构，读取time的方法是不同的
19 //32位架构下，需要把64位的time寄存器读到两个32位整数里，然后拼起来形成一个64位整数
20 //64位架构简单的一句rdtime就可以了
21 //__riscv_xlen是gcc定义的一个宏，可以用来区分是32位还是64位。
22 static inline uint64_t get_time(void) { //返回当前时间
23     #if __riscv_xlen == 64
24         uint64_t n;
25         __asm__ __volatile__ ("rdtime %0" : "=r"(n));
26         return n;
27     #else
28         uint32_t lo, hi, tmp;
29         __asm__ __volatile__ (
30             "1:\n"
31             "rdtimeh %0\n"
32             "rdtime %1\n"
33             "rdtimeh %2\n"
34             "bne %0, %2, 1b"
35             : "=&r"(hi), "=&r"(lo), "=&r"(tmp));
36         return ((uint64_t)hi << 32) | lo;
37     #endif
38 }
39
40
41 // Hardcode timebase
42 static uint64_t timebase = 1000000;
43
44 void clock_init(void) {
45     // sie这个CSR可以单独使能/禁用某个来源的中断。默认时钟中断是关闭的
46     // 所以我们要在初始化的时候，使能时钟中断
47     set_csr(sie, MIP_STIP); // enable timer interrupt in sie
48     //设置第一个时钟中断事件
49     clock_set_next_event();
50     // 初始化一个计数器
51     ticks = 0;
52
53     cprintf("++ setup timer interrupts\n");
54 }

```

```

55 //设置时钟中断：timer的数值变为当前时间 + timebase 后，触发一次时钟中断
56 //对于QEMU，timer增加1，过去了10-7 s，也就是100ns
57 void clock_set_next_event(void) { sbi_set_timer(get_time() + timebase); }

```

回来看trap.c里面时钟中断处理的代码, 还是很简单的：每秒100次时钟中断，触发每次时钟中断后设置10ms后触发下一次时钟中断，每触发100次时钟中断（1秒钟）输出一行信息到控制台。

```

1 // kern/trap/trap.c
2 #include<clock.h>
3
4 #define TICK_NUM 100
5 static void print_ticks() {
6     cprintf("%d ticks\n", TICK_NUM);
7 #ifdef DEBUG_GRADE
8     cprintf("End of Test.\n");
9     panic("EOT: kernel seems ok.");
10 #endif
11 }
12
13 void interrupt_handler(struct trapframe *tf) {
14     intptr_t cause = (tf->cause << 1) >> 1;
15     switch (cause) {
16         /* blabla 其他case*/
17         case IRQ_S_TIMER:
18             clock_set_next_event();//发生这次时钟中断的时候，我们要设置下一次时
19             if (++ticks % TICK_NUM == 0) {
20                 print_ticks();
21             }
22             break;
23         /* blabla 其他case*/
24     }

```

现在执行 `make qemu` ,应该能看到打印一行行的 `100 ticks` 。



时钟是属于外部设备了。之所以给大家呈现着一块，是为了能够更好的理解，操作系统与外设如何进行交互。中断来临如何处理。

项目组成与执行流

项目组成

```
1  lab1
2  |— Makefile
3  |— kern
4  |   |— debug
5  |   |   |— assert.h
6  |   |   |— kdebug.c
7  |   |   |— kdebug.h
8  |   |   |— kmonitor.c
9  |   |   |— kmonitor.h
10 |   |   |— panic.c
11 |   |   |— stab.h
12 |   |— driver
13 |   |   |— clock.c
14 |   |   |— clock.h
15 |   |   |— console.c
16 |   |   |— console.h
17 |   |   |— intr.c
18 |   |   |— intr.h
19 |   |— init
20 |   |   |— entry.S
21 |   |   |— init.c
22 |   |— libs
23 |   |   |— stdio.c
24 |   |— mm
25 |   |   |— memlayout.h
26 |   |   |— mmu.h
27 |   |   |— pmm.c
28 |   |   |— pmm.h
29 |   |— trap
30 |   |   |— trap.c
31 |   |   |— trap.h
32 |   |   |— trapentry.S
33 |— lab1.md
34 |— libs
35 |   |— defs.h
36 |   |— error.h
37 |   |— printfmt.c
38 |   |— readline.c
39 |   |— riscv.h
40 |   |— sbi.c
41 |   |— sbi.h
42 |   |— stdarg.h
```



```
43 |   |— stdio.h
44 |   |— string.c
45 |   |— string.h
46 |— readme.md
47 |— tools
48 |   |— function.mk
49 |   |— gdbinit
50 |   |— grade.sh
51 |   |— kernel.ld
52 |   |— sign.c
53 |   |— vector.c
54
55 9 directories, 43 files
```

硬件驱动层

`kern/driver/clock.c(h)` : 通过 `OpenSBI` 的接口, 可以读取当前时间(`rdtime`), 设置时钟事件(`sbi_set_timer`), 是时钟中断必需的硬件支持。

`kern/driver/intr.c(h)` : 中断也需要CPU的硬件支持, 这里提供了设置中断使能位的接口 (其实只封装了一句riscv指令) 。

初始化

`kern/init/init.c` : 需要调用中断机制的初始化函数。

中断处理

`kern/trap/trapentry.S` : 我们把中断入口点设置为这段汇编代码。这段汇编代码把寄存器的数据挪来挪去, 进行上下文切换。

`kern/trap/trap.c(h)` : 分发不同类型的中断给不同的handler, 完成上下文切换之后对中断的具体处理, 例如外设中断要处理外设发来的信息, 时钟中断要触发特定的事件。中断处理

初始化的函数也在这里，主要是把中断向量表(stvec)设置成所有中断都要跳到 `trapentry.S` 进行处理。

执行流

没有啦！需要自己总结，撰写实验报告噢！

LAB2：物理内存管理

如果我们只有物理内存空间，那么我们也可以写程序，但是所有的程序，包括内核，包括用户程序，都在同一个地址空间里，用户程序访问的 `0x80200000` 和内核访问的 `0x80200000` 是同一个地址。这样好不好？如果只有一个程序在运行，那也无所谓。但很多程序使用同一个内存空间，就会有问题：怎样防止程序之间互相干扰，甚至互相搞破坏？比较粗暴的方式就是，我让用户程序访问的 `0x80200000` 和内核访问的 `0x80200000` 不是一个地址。但是我们只有一块内存，为了创造两个不同的地址空间，我们可以引入一个“翻译”机制：程序使用的地址需要经过一步“翻译”才能变成真正的内存的物理地址。这个“翻译”过程，我们用一个“词典”实现—给出翻译之前的地址，可以在词典里查找翻译后的地址。每个程序都有唯一的一本“词典”，而它能使用的内存也就只有他的“词典”所包含的。

“词典”是否对能使用的每个字节都进行翻译？我们可以想象，存储每个字节翻译的结果至少需要一个字节，那么使用1MB的内存将至少需要构造1MB的“词典”，这效率太低了。观察到，一个程序使用内存的数量级通常远大于字节，至少以KB为单位（所以上古时代的人说的是“640K对每个人都够了”而不是“640B对每个人都够了”）。那么我们可以考虑，把连续的很多字节合在一起翻译，让他们翻译前后的数值之差相同，这就是“页”。

实验目的

1. 掌握内存管理相关的概念
2. 掌握内存地址的转换机制
3. 掌握页表的建立和使用方法
4. 掌握物理内存的管理方法

实验内容

1. 阅读手册有关内存管理部分内容。
2. 了解如何发现系统中的物理内存。
3. 学习如何使用页表机制进行物理内存与虚拟内存管理。
4. 自己动手实现页面分配算法。

练习

练习1：如何获取物理内存范围

如果 OS 无法提前知道当前硬件的可用物理内存范围，请问你有何办法让 OS 获取可用物理内存范围？

练习2：实现Best Fit页面分配算法

实现Best Fit页面分配算法，算法的时空复杂度不做要求，能通过测试即可。参考 `kern/mm/default_pmm.c` 对First Fit算法的实现。

地址与页表

物理地址与虚拟地址

RV64支持多种分页方案，但我们只介绍最受欢迎的一种， Sv39 。每个页的大小是4KB，也就是4096个字节。页表就是那个“字典”，里面有程序使用的虚拟页号到实际内存的物理页号的对应关系，但并不是所有的虚拟页都有对应的物理页。虚拟页可能的数目远大于物理页的数目，而且一个程序在运行时，一般不会拥有所有物理页的使用权，而只是将部分物理页在它的页表里进行映射。

在 Sv39 中，定义**物理地址(Physical Address)**有 56位，而**虚拟地址(Virtual Address)**有 39位。实际使用的时候，一个虚拟地址要占用 64位，只有低 39位有效，规定 63-39 位的值必须等于第 38 位的值（类似有符号整数），否则会认为该虚拟地址不合法，在访问时会产生异常。

未被使用的地址位

由于Sv39的虚拟地址比RISC-v64整数寄存器要短，可能你想知道剩下的35位是什么。 Sv39要求地址位 63-39是第 38位的副本。因此有效的虚拟地址是0x0000 0000 0000 0000 -0x0000 003f ffff ffff和 0xffff ffc0 0000 0000-0xffff ffff ffff ffff。这两个区间之间间隔的大小是两个区间长度大小的225倍，看上去似乎浪费了64位寄存器可以表达范围的99.999997%。为什么不充分地利用这额外的 25位空间呢？答案是，随着程序的增长，它们可能会需要大于 512 GiB的虚址空间。而架构师希望再不破坏向后兼容性的前提下增加地址空间。如果我们允许程序在高 25位中存储额外的数据，那么以后就不可能把这些位回收 从而存储更大的地址。像这样允许在未使用的地址位中存储数据的严重错误，在计算机的历史中已经重复出现了多次。

不论是物理地址还是虚拟地址，我们都可以认为，最后12位表示的是页内偏移，也就是这个地址在它所在页帧的什么位置（同一个位置的物理地址和虚拟地址的页内偏移相同）。除了最后12位，前面的部分表示的是物理页号或者虚拟页号。

页表项

很容易理解，我们需要给词典的每个词条约定一个固定的格式（包括每个词条的大小，含义），查起来才方便。

我们的“词典”（页表）存储在内存里，由若干个格式固定的“词条”也就是**页表项（PTE, Page Table Entry）**组成。

一个页表项是用来描述一个虚拟页号如何映射到物理页号的。如果一个虚拟页号通过**某种手段**找到了一个页表项，并通过读取上面的物理页号完成映射，我们称这个虚拟页号**通过该页表项**完成映射。

Sv39的一个页表项占据8字节，结构是这样的：

63-54	53-28	27-19	18-10	9-8	7	6	5	4	3	2	1	0
<i>Reserved</i>	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
10	26	9	9	2	1	1	1	1	1	1	1	1

我们可以看到 Sv39 里面的一个页表项大小为64位 8 字节。其中第 53-10 共 44 位为一个物理页号，表示这个虚拟页号映射到的物理页号。后面的第 9-0位则描述映射的状态信息。

- RSW两位留给 S Mode 的应用程序，我们可以用来进行拓展。
- D，即 Dirty，如果 D=1表示自从上次 D被清零后，有虚拟地址通过这个页表项进行写入。
- A，即 Accessed，如果 A=1表示自从上次 A 被清零后，有虚拟地址通过这个页表项进行读、或者写、或者取指。
- G，即Global，如果G=1表示这个页表项是“全局”的，也就是所有的地址空间（所有的页表）都包含这一项
- U(user)为 111 表示用户态 (U Mode)的程序 可以通过该页表项进行映射。在用户态运行时也只能够通过 U=1的页表项进行虚实地址映射。

注意，S Mode 不一定可以通过 U=1的页表项进行映射。我们需要将 S Mode 的状态寄存器 `sstatus` 上的 **SUM** 位手动设置为 111 才可以做到这一点（通常情况不会把它置1）。否则通过 U=1的页表项进行映射也会报出异常。另外，不论 `sstatus` 的**SUM**位如何取值，S Mode都不允许执行 U=1的页面里包含的指令，这是出于安全的考虑。

- R,W,X为许可位，分别表示是否可读 (Readable)，可写 (Writable)，可执行 (Executable)。
- V表示这个页表项是否合法。如果为 000 表示不合法，此时页表项其他位的值都会被忽略。

以 W这一位为例，如果 W=0 表示不可写，那么如果一条 store 的指令，它通过这个页表项完成了虚拟页号到物理页号的映射，找到了物理地址。但是仍然会报出异常，是因为这个页表项规定如果物理地址是通过它映射得到的，那么不准写入！R,X 也是同样的道理。

根据 R,W,X取值的不同，我们可以分成下面几种类型：

X	W	R	Meaning
0	0	0	指向下一级页表的指针
0	0	1	这一页只读
0	1	0	保留(reserved for future use)
0	1	1	这一页可读可写（不可执行）
1	0	0	这一页可读可执行（不可写）
1	0	1	这一页可读可执行
1	1	0	保留(reserved for future use)
1	1	1	这一页可读可写可执行

“指向下一级页表的指针”暗示我们有多级页表。下面就来看看多级页表是怎么回事。

多级页表

主要矛盾在于：相比于可用的物理内存空间，我们的虚拟地址空间太大，不可能为每个虚拟内存页都分配一个页表项。在Sv39中，虚拟地址有39位，后12位是页内偏移，还有27位可以编码不同的虚拟页号。如果开个大数组Pagetable[]，给 2^{27} 个虚拟页号都分配8字节的页表项，pagetable[vpn]是虚拟页号为vpn的虚拟页的页表项，那就是整整1 GiB的内存。这里面

很多虚拟地址我们没有用到，会有大片大片的页表项的V标志位为0（不合法）。我们不想为那么多非法页表项浪费宝贵的内存空间。

因此，我们可以对页表进行“分级”，变成一个树状结构。也就是把很多页表项组合成一个“大页”，如果这些页表项都非法（没有对应的物理页），那么只需要用一个非法的页表项来覆盖这个大页，而不需要分别建立一大堆非法页表项。很多个大页(megapage)还可以组合起来变成大大页(gigapage!)，继而可以有大大大页(terapage!).....但肯定不是分层越多越好，层数越多开销越大。

Sv39权衡各方面效率，使用三级页表。有4KiB=4096字节的页，大小为2MiB=2²¹字节的大页，和大小为1 GiB的大大页。

原先的一个39位虚拟地址，被我们看成27位的页号和12位的页内偏移。

现在我们把它看成9位的“大大页页号”，9位的“大页页号”（也是大大页内的页内偏移），9位的“页号”（大页的页内偏移），还有12位的页内偏移。这是一个递归的过程，中间的每一级页表映射是类似的。

也就是说，整个Sv39的虚拟内存空间里，有512（2的9次方）个大大页，每个大大页里有512个大页，每个大页里有512个页，每个页里有4096个字节，整个虚拟内存空间里就有512*512*512*4096=512³*4096个字节，是512GiB的地址空间。

那么为啥是512呢？注意，4096/8 = 512，我们恰好可以在一页里放下512个页表项！

我们可以认为，Sv39的多级页表在逻辑上是一棵树，它的每个叶子节点（直接映射4KB的页的页表项）都对应内存的一页，它的每个内部节点都对应512个更低一层的节点，而每个内部节点向更低一层的节点的链接都使用内存里的一页进行存储。

或者说，Sv39页表的根节点占据一页4KiB的内存，存储512个页表项，分别对应512个1 GiB的大大页，其中有些页表项（大大页）是非法的，另一些合法的页表项（大大页）是根节点的儿子，可以通过合法的页表项跳转到一个物理页号，这个物理页对应树中一个“大大页”的节点，里面有512个页表项，每个页表项对应一个2MiB的大页。同样，这些大页可能合法，也可能非法，非法的页表项不对应内存里的页，合法的页表项会跳转到一个物理页号，这个物理页对应树中一个“大页”的节点，里面有512个页表项，每个页表项对应一个4KiB的页，在这里最终完成虚拟页到物理页的映射。

三级和二级页表项不一定要指向下一级页表。我们知道每个一级页表项控制一个虚拟页号，即控制 4KiB 虚拟内存；每个二级页表项则控制 9位虚拟页号，总计控制 $4\text{KiB} \times 2^9 = 2\text{MiB}$ 虚拟内存；每个三级页表项控制 18位虚拟页号，总计控制 $2\text{MiB} \times 2^9 = 1\text{GiB}$ 虚拟内存。我们可以将二级页表项的 R,W,X 设置为不是全 0 的许可要求，那么它将与一级页表项类似，只不过可以映射一个 2MiB 的**大页 (Mega Page)**。同理，也可以将三级页表项看作一个叶子，来映射一个 1GiB 的**大大页(Giga Page)**。

i 这么看来，建立一个虚拟页到物理页的映射，我们需要在三个层级（页，大页，大大页）各自给它分配一个物理页帧，是不是还没把所有物理内存都建立映射，就把所有物理页帧都耗尽了？

事实上这个问题是不存在的。关键点在于，我们要映射的是一段**连续**的虚拟内存**区间**，因此，每连续建立 512 页的映射才会新建一个一级页表，每连续建立 512^2 页的映射才会新建一个二级页表，而三级页表最多只新建一个。因此这样进行映射花费的总物理页帧数,约占物理内存中物理页帧总数的约 $1/512 \approx 0.2\%$ 。

页表基址

在翻译的过程中，我们首先需要知道树状页表的根节点的物理地址（思考：为啥不是“根节点的虚拟地址”？）。

这一般保存在一个特殊寄存器里。对于RISCV架构，是一个叫做 `satp`（Supervisor Address Translation and Protection Register）的CSR。实际上，`satp` 里面存的不是最高级页表的起始物理地址，而是它所在的物理页号。除了物理页号，`satp` 还包含其他信息

63-60	59-44	43-0
MODE	ASID	PPN
4	16	44

MODE表示当前页表的模式

- 0000表示不使用页表，直接使用物理地址，在简单的嵌入式系统里用着很方便。
- 0100表示Sv39页表，也就是我们使用的，虚拟内存空间高达512GiB。
- 0101表示Sv48页表，它和Sv39兼容，可以猜猜它有几层。虚拟内存空间高达256TiB。
- 其他编码保留备用。

ASID（Address Space Identifier 地址空间标识符）域是可选的，它可以用来降低上下文切换的开销。

PPN字段保存了根页表的物理地址，它以 4 KiB的页面大小为单位。通常 M模式的程序在第一次进入 S模式之前会把零写入 satp以禁用分页，然后 S模式的程序在初始化页表以后会再次进行satp寄存器的写操作。

OS 可以在内存中为不同的应用分别建立不同虚实映射的页表，并通过修改寄存器 `satp` 的值指向不同的页表，从而可以修改 CPU 虚实地址映射关系及内存保护的行为。

快表

物理内存的访问速度要比 CPU 的运行速度慢很多，去访问一次物理内存可能需要几百个时钟周期（带来所谓的“冯诺依曼瓶颈”）。如果我们按照页表机制一步步走，将一个虚拟地址转化为物理地址需要访问 333 次物理内存，得到物理地址之后还要再访问一次物理内存，才能读到我们想要的数​​据。这很大程度上降低了效率。

好在，实践表明虚拟地址的访问具有时间局部性和空间局部性。

- 时间局部性是指，被访问过一次的地址很有可能不久的将来再次被访问；
- 空间局部性是指，如果一个地址被访问，则这个地址附近的地址很有可能在不远的将来被访问。

因此，在 CPU 内部，我们使用**快表 (TLB, Translation Lookaside Buffer)** 来记录近期已完成的虚拟页号到物理页号的映射。由于局部性，当我们要做一个映射时，会有很大可能这个映射在近期被完成过，所以我们可以先到 TLB 里面去查一下，如果有的话我们就可以直接完成映射，而不用访问那么多次内存了。

但是，我们如果修改了 `satp` 寄存器，比如将上面的 PPN字段进行了修改，说明我们切换到了一个与先前映射方式完全不同的页表。此时快表里面存储的映射结果就跟不上时代了，

很可能是错误的。这种情况下我们要使用 `sfence.vma` 指令刷新整个 TLB。

同样，我们手动修改一个页表项之后，也修改了映射，但 TLB 并不会自动刷新，我们也需要使用 `sfence.vma` 指令刷新 TLB。如果不加参数的，`sfence.vma` 会刷新整个 TLB。你可以在后面加上一个虚拟地址，这样 `sfence.vma` 只会刷新这个虚拟地址的映射。

物理内存探测

操作系统怎样知道物理内存所在的那段物理地址呢？在 RISC-V 中，这个一般是由 bootloader，即 OpenSBI 来完成的。它来完成对于包括物理内存在内的各外设的扫描，将扫描结果以 DTB(Device Tree Blob) 的格式保存在物理内存中的某个地方。随后 OpenSBI 会将其地址保存在 `a1` 寄存器中，给我们使用。

这个扫描结果描述了所有外设的信息，当中也包括 Qemu 模拟的 RISC-V 计算机中的物理内存。

✔ Qemu 模拟的 RISC-V virt 计算机中的物理内存

通过查看 `virt.c` 的 `virt_memmap[]` 的定义，可以了解到 Qemu 模拟的 RISC-V virt 计算机的详细物理内存布局。可以看到，整个物理内存中有不少内存空洞（即含义为 `unmapped` 的地址空间），也有很多外设特定的地址空间，现在我们看不懂没有关系，后面会慢慢涉及到。目前只需关心最后一块含义为 **DRAM** 的地址空间，这就是 OS 将要管理的 128MB 的内存空间。

起始地址	终止地址	含义
0x0	0x100	QEMU VIRT_DEBUG
0x100	0x1000	unmapped
0x1000	0x12000	QEMU MROM (包括 hard-coded reset vector; device tree)
0x12000	0x100000	unmapped
0x100000	0x101000	QEMU VIRT_TEST
0x101000	0x2000000	unmapped
0x2000000	0x2010000	QEMU VIRT_CLINT
0x2010000	0x3000000	unmapped
0x3000000	0x3010000	QEMU VIRT_PCIE_PIO

0x3010000	0xc000000	unmapped
0xc000000	0x10000000	QEMU VIRT_PLIC
0x10000000	0x10000100	QEMU VIRT_UART0
0x10000100	0x10001000	unmapped
0x10001000	0x10002000	QEMU VIRT_VIRTIO
0x10002000	0x20000000	unmapped
0x20000000	0x24000000	QEMU VIRT_FLASH
0x24000000	0x30000000	unmapped
0x30000000	0x40000000	QEMU VIRT_PCIE_ECAM
0x40000000	0x80000000	QEMU VIRT_PCIE_MMIO
0x80000000	0x88000000	DRAM 缺省 128MB，大小可配置

以页为单位管理物理内存

Page结构体

在获得可用物理内存范围后，系统需要建立相应的数据结构来管理以物理页（按4KB对齐，且大小为4KB的物理内存单元）为最小单位的整个物理内存，以配合后续涉及的分页管理机制。每个物理页可以用一个 Page 数据结构来表示。由于一个物理页需要占用一个Page结构的空间，Page结构在设计时须尽可能小，以减少对内存的占用。Page的定义在

kern/mm/memlayout.h 中。以页为单位的物理内存分配管理的实现在 kern/default_pmm.[ch] 。

为了与以后的分页机制配合，我们首先需要建立对整个计算机的每一个物理页的属性用结构 Page来表示，它包含了映射此物理页的虚拟页个数，描述物理页属性的flags和双向链接各个Page结构的page_link双向链表。

```
1 struct Page {
2     int ref;           // page frame's reference counter
3     uint32_t flags;    // array of flags that describe the status of the page
4     unsigned int property; // the num of free block, used in first fit pm m
5     list_entry_t page_link; // free list link
6 };
```

这里看看Page数据结构的各个成员变量有何具体含义。ref表示这页被页表的引用记数（在“实现分页机制”一节会讲到）。如果这个页被页表引用了，即在某页表中有一个页表项设置了一个虚拟页到这个Page管理的物理页的映射关系，就会把Page的ref加一；反之，若页表项取消，即映射关系解除，就会把Page的ref减一。flags表示此物理页的状态标记，进一步查看 kern/mm/memlayout.h 中的定义，可以看到：

```
1 /* Flags describing the status of a page frame */
2 #define PG_reserved      0           // the page descriptor is reserved
3 #define PG_property      1           // the member 'property' is valid
```

这表示flags目前用到了两个bit表示页目前具有的两种属性，bit 0表示此页是否被保留（reserved），如果是被保留的页，则bit 0会设置为1，且不能放到空闲页链表中，即这样的

页不是空闲页，不能动态分配与释放。比如目前内核代码占用的空间就属于这样“被保留”的页。在本实验中，bit 1表示此页是否是free的，如果设置为1，表示这页是free的，可以被分配；如果设置为0，表示这页已经被分配出去了，不能被再二次分配。另外，本实验这里取的名字PG_property比较不直观，主要是我们可以设计不同的页分配算法（best fit, buddy system等），那么这个PG_property就有不同的含义了。

在本实验中，Page数据结构的成员变量property用来记录某连续内存空闲块的大小（即地址连续的空闲页的个数）。这里需要注意的是用到此成员变量的这个Page比较特殊，是这个连续内存空闲块地址最小的一页（即头一页，Head Page）。连续内存空闲块利用这个页的成员变量property来记录在此块内的空闲页的个数。这里去的名字property也不是很直观，原因与上面类似，在不同的页分配算法中，property有不同的含义。

Page数据结构的成员变量page_link是便于把多个连续内存空闲块链接在一起的双向链表指针（可回顾在lab0实验指导书中有关双向链表数据结构的介绍）。这里需要注意的是用到此成员变量的这个Page比较特殊，是这个连续内存空闲块地址最小的一页（即头一页，Head Page）。连续内存空闲块利用这个页的成员变量page_link来链接比它地址小和大的其他连续内存空闲块。

free_area_t结构体

在初始情况下，也许这个物理内存的空闲物理页都是连续的，这样就形成了一个大的连续内存空闲块。但随着物理页的分配与释放，这个大的连续内存空闲块会分裂为一系列地址不连续的多个小连续内存空闲块，且每个连续内存空闲块内部的物理页是连续的。那么为了有效地管理这些小连续内存空闲块。所有的连续内存空闲块可用一个双向链表管理起来，便于分配和释放，为此定义了一个free_area_t数据结构，包含了一个list_entry结构的双向链表指针和记录当前空闲页的个数的无符号整型变量nr_free。其中的链表指针指向了空闲的物理页。

```
1  /* free_area_t - maintains a doubly linked list to record free (unused) pa
2  typedef struct {
3      list_entry_t free_list;                // the
4      unsigned int nr_free;                  // # of
5  } free_area_t;
```

初始化Page结构体

为了管理物理内存，我们需要在内核里定义一些数据结构，来存储“当前使用了哪些物理页面，哪些物理页面没被使用”这样的信息，使用的是Page结构体。我们将一些Page结构体在内存里排列在内核后面，这要占用一些内存。而摆放这些Page结构体的物理页面，以及内核占用的物理页面，之后都无法再使用了。我们用 `page_init()` 函数给这些管理物理内存的结构体做初始化。

`page_init()` 的代码里，我们调用了一个函数 `init_memmap()`，这我们的另一个结构体 `pmm_manager` 有关。虽然C语言基本上不支持面向对象，但我们可以用类似面向对象的思路，把“物理内存管理”的功能集中给一个结构体。我们甚至可以让函数指针作为结构体的成员，强行在C语言里支持了“成员函数”。可以看到，我们调用的 `init_memmap()` 实际上又调用了 `pmm_manager` 的一个“成员函数”。

`pmm_manager` 提供了各种接口：分配页面，释放页面，查看当前空闲页面数。但是我们好像始终没看见 `pmm_manager` 内部对这些接口的实现，那些接口只是作为函数指针，作为 `pmm_manager` 的一部分，我们需要把那些函数指针变量赋值为真正的函数名称。在这里面我们把 `pmm_manager` 的指针赋值成 `&default_pmm_manager`。

页面分配算法

如果要在ucore中实现连续物理内存分配算法，则需要考虑的事情比较多，相对课本上的物理内存分配算法描述要复杂不少。下面介绍一下如果要实现一个FirstFit内存分配算法的大致流程。原理FirstFit内存分配算法上很简单，但要在ucore中实现，需要充分了解和利用ucore已有的数据结构和相关操作、关键的一些全局变量等。

关键数据结构与变量

`first_fit` 分配算法需要维护一个查找有序（地址按从小到大排列）空闲块（以页为最小单位的连续地址空间）的数据结构，而双向链表是一个很好的选择。

`libs/list.h` 定义了可挂接任意元素的通用双向链表结构和对应的操作，所以需要了解如何使用这个文件提供的各种函数，从而可以完成对双向链表的初始化/插入/删除等。

显然，我们可以通过 `free_area_t` 数据结构来完成对空闲块的管理。而 `default_pmm.c` 中定义的 `free_area` 变量就是干这个事情的。

`kern/mm/pmm.h` 中定义了一个通用的分配算法的函数列表，用 `pmm_manager` 表示。其中 `init` 函数就是用来初始化 `free_area` 变量的，`first_fit` 分配算法可直接重用 `default_init` 函数的实现。`init_memmap` 函数需要根据现有的内存情况构建空闲块列表的初始状态。何时应该执行这个函数呢？

通过分析代码，可以知道：

```
kern_init --> pmm_init-->page_init-->init_memmap--> pmm_manager->init_memmap
```

所以，`default_init_memmap` 需要根据 `page_init` 函数中传递过来的参数（某个连续地址的空闲块的起始页，页个数）来建立一个连续内存空闲块的双向链表。这里有一个假定 `page_init` 函数是按地址从小到大的顺序传来的连续内存空闲块的。链表头是

`free_area.free_list`，链表项是Page数据结构的 `base->page_link`。这样我们就依靠Page数据结构中的成员变量 `page_link` 形成了连续内存空闲块列表。

设计实现

`default_init_memmap` 函数将根据每个物理页帧的情况来建立空闲页链表，且空闲页块应该是根据地址高低形成一个有序链表。根据上述变量的定义，`default_init_memmap` 可大致实现如下：

```
1  default_init_memmap(struct Page *base, size_t n) {
2      struct Page *p = base;
3      for (; p != base + n; p++) {
4          p->flags = p->property = 0;
5          set_page_ref(p, 0);
6      }
7      base->property = n;
8      SetPageProperty(base);
9      nr_free += n;
10     list_add(&free_list, &(base->page_link));
11 }
```

如果要分配一个页，那要考虑哪些呢？这里就需要考虑实现 `default_alloc_pages` 函数，注意参数 `n` 表示要分配 `n` 个页。另外，需要注意实现时尽量多考虑一些边界情况，这样确保软件的鲁棒性。比如

```
1  if (n > nr_free) {
2      return NULL;
3  }
```

这样可以确保分配不会超出范围。也可加一些 `assert` 函数，在有错误出现时，能够迅速发现。比如 `n` 应该大于0，我们就可以加上

```
assert(n > 0);
```

这样在 $n \leq 0$ 的情况下，ucore会迅速报错。firstfit 需要从空闲链表头开始查找最小的地址，通过 list_next 找到下一个空闲块元素，通过 le2page 宏可以由链表元素获得对应的 Page 指针 p。通过 p->property 可以了解此空闲块的大小。如果 $\geq n$ ，这就找到了！如果 $< n$ ，则list_next，继续查找。直到 list_next == &free_list，这表示找完了一遍了。找到后，就要从新组织空闲块，然后把找到的page返回。所以 default_alloc_pages 可大致实现如下：

```
1 static struct Page *
2 default_alloc_pages(size_t n) {
3     if (n > nr_free) {
4         return NULL;
5     }
6     struct Page *page = NULL;
7     list_entry_t *le = &free_list;
8     while ((le = list_next(le)) != &free_list) {
9         struct Page *p = le2page(le, page_link);
10        if (p->property >= n) {
11            page = p;
12            break;
13        }
14    }
15    if (page != NULL) {
16        list_del(&(amp;page->page_link));
17        if (page->property > n) {
18            struct Page *p = page + n;
19            p->property = page->property - n;
20            list_add(&free_list, &(p->page_link));
21        }
22        nr_free -= n;
23        ClearPageProperty(page);
24    }
25    return page;
26 }
```

default_free_pages 函数的实现其实是 default_alloc_pages 的逆过程，不过需要考虑空闲块的合并问题。这里就不再细讲了。自行阅读代码。

① 看完了first fit算法了，就需要自己实现best fit算法了噢，思路都是一样的，依葫芦画瓢。

项目组成与执行流

项目组成

```
1  lab2
2  |— Makefile
3  |— kern
4  |   |— debug
5  |   |   |— assert.h
6  |   |   |— kdebug.c
7  |   |   |— kdebug.h
8  |   |   |— kmonitor.c
9  |   |   |— kmonitor.h
10 |   |   |— panic.c
11 |   |   |— stab.h
12 |   |— driver
13 |   |   |— clock.c
14 |   |   |— clock.h
15 |   |   |— console.c
16 |   |   |— console.h
17 |   |   |— intr.c
18 |   |   |— intr.h
19 |   |— init
20 |   |   |— entry.S
21 |   |   |— init.c
22 |   |— libs
23 |   |   |— stdio.c
24 |   |— mm
25 |   |   |— best_fit_pmm.c
26 |   |   |— best_fit_pmm.h
27 |   |   |— default_pmm.c
28 |   |   |— default_pmm.h
29 |   |   |— memlayout.h
30 |   |   |— mmu.h
31 |   |   |— pmm.c
32 |   |   |— pmm.h
33 |   |— sync
34 |   |   |— sync.h
35 |   |— trap
36 |   |   |— trap.c
37 |   |   |— trap.h
38 |   |   |— trapentry.S
39 |— lab2.md
40 |— libs
41 |   |— atomic.h
42 |   |— defs.h
```

```
43 | |— error.h
44 | |— list.h
45 | |— printfmt.c
46 | |— readline.c
47 | |— riscv.h
48 | |— sbi.c
49 | |— sbi.h
50 | |— stdarg.h
51 | |— stdio.h
52 | |— string.c
53 | |— string.h
54 |— tools
55 |   |— boot.ld
56 |   |— function.mk
57 |   |— gdbinit
58 |   |— grade.sh
59 |   |— kernel.ld
60 |   |— kernel_nopage.ld
61 |   |— sign.c
62 |   |— vector.c
63
64 10 directories, 51 files
```

链表

`libs/list.h` : 定义了通用双向链表结构以及相关的查找、插入等基本操作，这是建立基于链表方法的物理内存管理（以及其他内核功能）的基础。其他有类似双向链表需求的内核功能模块可直接使用 `list.h` 中定义的函数。

物理内存管理器

`kern/mm/pmm.c(h)` 物理内存管理器结构的实现，定义了内存管理相关函数。

默认的页面分配算法

`kern/mm/default_pmm.c(h)` 上一节已经介绍完毕，具体的内存管理函数的实现，还包括实现的检查。

Best Fit 页面分配算法

`kern/mm/best_fit_pmm.c(h)` 这就是需要自己实现啦~疯狂暗示：依葫芦画瓢！甚至是画葫芦。

执行流

结合前面所述自行理解、总结。

LAB3：虚拟内存管理

什么是虚拟内存？简单地说是指程序员或CPU“看到”的内存。但有几点需要注意：

1. 虚拟内存单元不一定有实际的物理内存单元对应，即实际的物理内存单元可能不存在；
2. 如果虚拟内存单元对应有实际的物理内存单元，那二者的地址一般是不相等的；
3. 通过操作系统实现的某种内存映射可建立虚拟内存与物理内存的对应关系，使得程序员或CPU访问的虚拟内存地址会自动转换为一个物理内存地址。

那么这个“虚拟”的作用或意义在哪里体现呢？在操作系统中，虚拟内存其实包含多个虚拟层次，在不同的层次体现了不同的作用。首先，在有了分页机制后，程序员或CPU“看到”的地址已经不是实际的物理地址了，这已经有一层虚拟化，我们可简称为内存地址虚拟化。有了内存地址虚拟化，我们就可以通过设置页表项来限定软件运行时的访问空间，确保软件运行不越界，完成内存访问保护的功能。

通过内存地址虚拟化，可以使得软件在没有访问某虚拟内存地址时不分配具体的物理内存，而只有在实际访问某虚拟内存地址时，操作系统再动态地分配物理内存，建立虚拟内存到物理内存的页映射关系，这种技术称为按需分页（demand paging）。把不经常访问的数据所占的内存空间临时写到硬盘上，这样可以腾出更多的空闲内存空间给经常访问的数据；当CPU访问到不经常访问的数据时，再把这些数据从硬盘读入到内存中，这种技术称为页换入换出（page swap in/out）。这种内存管理技术给了程序员更大的内存“空间”，从而可以让更多的程序在内存中并发运行。

实验目的

1. 了解虚拟内存的Page Fault异常处理实现
2. 掌握页替换算法在操作系统中的实现

实验内容

1. 阅读理论课有关页面置换相关的内容。
2. 阅读PageFault异常处理和FIFO页面替换算法的实现。
3. 自己动手实现页面置换算法。

练习

练习1：简述页面从换入到换出过程

描述FIFO页面置换算法下，一个页面从被换入到被换出的过程中，会经过代码里哪些函数/宏的处理（或者说，需要调用哪些函数/宏），并用简单的一两句话描述每个函数在过程中做了什么。我们认为只要函数原型不同，就算两个不同的函数。要求指出对执行过程有实际影响,删去后会导致输出结果不同的函数（例如assert）而不是cprintf这样的函数。

练习2：理解get_pte函数

1. `get_pte()` 函数中有两段形式类似的代码，结合sv32，sv39，sv48的异同，解释这两段代码为什么如此相像？
 2. 目前`get_pte()`函数将页表项的查找和页表项的分配合并在一个函数里，你认为这种写法好吗？有没有必要把两个功能拆开？
-

练习3：实现Clock页替换算法

在我们给出的框架上，填写代码，实现 Clock页替换算法，比较它和FIFO算法的不同。

页面置换

操作系统为何要进行页面置换呢？这是由于操作系统给用户态的应用程序提供了一个虚拟的“大容量”内存空间，而实际的物理内存空间又没有那么大。所以操作系统就“瞒着”应用程序，只把应用程序中“常用”的数据和代码放在物理内存中，而不常用的数据和代码放在了硬盘这样的存储介质上。如果应用程序访问的是“常用”的数据和代码，那么操作系统已经放置在内存中了，不会出现什么问题。但当应用程序访问它认为应该在内存中的数据和代码时，如果这些数据或代码不在内存中，则根据上一小节介绍，会产生页访问异常。这时，操作系统必须能够应对这种页访问异常，即尽快把应用程序当前需要的数据或代码放到内存中来，然后重新执行应用程序产生异常的访存指令。如果在把硬盘中对应的数据或代码调入内存前，操作系统发现物理内存已经没有空闲空间了，这时操作系统必须把它认为“不常用”的页换出到磁盘上去，以腾出内存空闲空间给应用程序所需的数据或代码。

页面置换算法

操作系统迟早会碰到没有内存空闲空间而必须要置换出内存中某个“不常用”的页的情况。如何判断内存中哪些是“常用”的页，哪些是“不常用”的页，把“常用”的页保持在内存中，在物理内存空闲空间不够的情况下，把“不常用”的页置换到硬盘上就是页替换算法着重考虑的问题。容易理解，一个好的页替换算法会导致页访问异常次数少，也就意味着访问硬盘的次数也少，从而使得应用程序执行的效率就高。本次实验涉及的页替换算法（包括扩展练习）：

- 先进先出(First In First Out, FIFO)页替换算法：该算法总是淘汰最先进入内存的页，即选择在内存中驻留时间最久的页予以淘汰。只需把一个应用程序在执行过程中已调入内存的页按先后次序链接成一个队列，队列头指向内存中驻留时间最久的页，队列尾指向最近被调入内存的页。这样需要淘汰页时，从队列头很容易查找到需要淘汰的页。FIFO 算法只是在应用程序按线性顺序访问地址空间时效果才好，否则效率不高。因为那些常被访问的页，往往在内存中也停留得最久，结果它们因变“老”而不得被置换出去。FIFO 算法的另一个缺点是，它有一种异常现象（Belady 现象），即在增加放置页的物理页帧的情况下，反而使页访问异常次数增多。
- 最久未使用(least recently used, LRU)算法：利用局部性，通过过去的访问情况预测未来的访问情况，我们可以认为最近还被访问过的页面将来被访问的可能性大，而很久没访问过的页面将来不太可能被访问。于是我们比较当前内存里的页面最近一次被访问的时间，把上一次访问时间离现在最久的页面置换出去。

- 时钟（Clock）页替换算法：是 LRU 算法的一种近似实现。时钟页替换算法把各个页面组织成环形链表的形式，类似于一个钟的表面。然后把一个指针（简称当前指针）指向最老的那个页面，即最先进来的那个页面。另外，时钟算法需要在页表项（PTE）中设置了一位访问位来表示此页表项对应的页当前是否被访问过。当该页被访问时，CPU 中的 MMU 硬件将把访问位置“1”。当操作系统需要淘汰页时，对当前指针指向的页所对应的页表项进行查询，如果访问位为“0”，则淘汰该页，如果该页被写过，则还要把它换出到硬盘上；如果访问位为“1”，则将该页表项的此位置“0”，继续访问下一个页。该算法近似地体现了 LRU 的思想，且易于实现，开销少，需要硬件支持来设置访问位。时钟页替换算法在本质上与 FIFO 算法是类似的，不同之处是在时钟页替换算法中跳过了访问位为 1 的页。
- 改进的时钟（Enhanced Clock）页替换算法：在时钟置换算法中，淘汰一个页面时只考虑了页面是否被访问过，但在实际情况中，还应考虑被淘汰的页面是否被修改过。因为淘汰修改过的页面还需要写回硬盘，使得其置换代价大于未修改过的页面，所以优先淘汰没有修改的页，减少磁盘操作次数。改进的时钟置换算法除了考虑页面的访问情况，还需考虑页面的修改情况。即该算法不但希望淘汰的页面是最近未使用的页，而且还希望被淘汰的页是在主存驻留期间其页面内容未被修改过的。这需要为每一页的对应页表项内容中增加一位引用位和一位修改位。当该页被访问时，CPU 中的 MMU 硬件将把访问位置“1”。当该页被“写”时，CPU 中的 MMU 硬件将把修改位置“1”。这样这两位就存在四种可能的组合情况：（0，0）表示最近未被引用也未被修改，首先选择此页淘汰；（0，1）最近未被使用，但被修改，其次选择；（1，0）最近使用而未修改，再次选择；（1，1）最近使用且修改，最后选择。该算法与时钟算法相比，可进一步减少磁盘的 I/O 操作次数，但为了查找到一个尽可能适合淘汰的页面，可能需要经过多次扫描，增加了算法本身的执行开销。

虚假的硬盘

在 QEMU 里实际上并没有真正模拟“硬盘”。为了实现“页面置换”的效果，我们采取的措施是，从内核的静态存储(static)区里面分出一块内存，声称这块存储区域是“硬盘”，然后包裹一下给出“硬盘 IO”的接口。思考一下，内存和硬盘，除了一个掉电后数据易失一个不易失，一个访问快一个访问慢，其实并没有本质的区别。对于我们的页面置换算法来说，也不要求硬盘上存多余页面的交换空间能够“不易失”，反正这些页面存在内存里的时候就是易失的。理论上，我们完全可以把一块机械硬盘加以改造，写好驱动之后，插到主板的内存插槽上作为内存条使用，当然性能就别想了。（如果半导体工业没有发明出成本和访问速率都介于寄存器和硬盘之间的 ram，我们将不得不这么做！）那么我们就把 QEMU 模拟出来的一块 ram 叫做“硬盘”，用作页面置换时的交换区，完全没有问题。你可能会觉得，这样折腾一通，我们总共

能使用的页面数并没有增加，原先能直接在内存里使用的一些页面变成了“硬盘”，只是在自娱自乐。确实，我们在这里只是想介绍页面置换的原理，并不关心实际性能。

这一部分在 `driver/ide.h` `driver/ide.c` `fs/fs.h` `fs/swapfs.h` `fs/swapfs.c` 实现。

`fs` 就是file system,我们这里其实并没有“文件”的概念，这个模块称作 `fs` 只是说明它是“硬盘”和内核之间的接口。

`ide` 在这里不是integrated development environment的意思，而是Integrated Drive Electronics的意思，表示的是一种标准的硬盘接口。这里写的东西和Integrated Drive Electronics并不相关，这个命名是ucore的历史遗留。

```
1 // kern/driver/ide.c
2 /*
3  *#include"s
4  */
5
6 void ide_init(void) {}
7
8 #define MAX_IDE 2
9 #define MAX_DISK_NSECS 56
10 static char ide[MAX_DISK_NSECS * SECTSIZE];
11
12 bool ide_device_valid(unsigned short ideno) { return ideno < MAX_IDE; }
13
14 size_t ide_device_size(unsigned short ideno) { return MAX_DISK_NSECS; }
15
16 int ide_read_secs(unsigned short ideno, uint32_t secno, void *dst,
17                  size_t nsecs) {
18     //ideno: 假设挂载了多块磁盘，选择哪一块磁盘 这里我们其实只有一块“磁盘”，这个参
19     int iobase = secno * SECTSIZE;
20     memcpy(dst, &ide[iobase], nsecs * SECTSIZE);
21     return 0;
22 }
23
24 int ide_write_secs(unsigned short ideno, uint32_t secno, const void *src,
25                   size_t nsecs) {
26     int iobase = secno * SECTSIZE;
27     memcpy(&ide[iobase], src, nsecs * SECTSIZE);
28     return 0;
29 }
```



可以看到，我们这里所谓的“硬盘IO”，只是在内存里用 `memcpy` 把数据复制来复制去。同时为了逼真地模仿磁盘，我们只允许以磁盘扇区为数据传输的基本单位，也就是一次传输的数据必须是512字节的倍数，并且必须对齐。

PageFault

出现

一般应用程序的对虚拟内存的“需求”与物理内存空间的“供给”没有直接的对应关系，ucore是通过 `page fault` 异常处理来间接完成 这二者之间的衔接。

当我们引入了虚拟内存，就意味着虚拟内存的空间可以远远大于物理内存，意味着程序可以访问“不对应物理内存页帧的虚拟内存地址”，这时CPU应当抛出 `Page Fault` 这个异常。

 在操作系统的设计中，一个基本的原则是：并非所有的物理页都可以交换出去的，只有映射到用户空间且被用户程序直接访问的页面才能被交换，而被内核直接使用的内核空间的页面不能被换出。这里面的原因是什么呢？操作系统是执行的关键代码，需要保证运行的高效性和实时性，如果在操作系统执行过程中，发生了缺页现象，则操作系统不得不等很长时间（硬盘的访问速度比内存的访问速度慢 2~3 个数量级），这将导致整个系统运行低效。而且，不难想象，处理缺页过程所用到的内核代码或者数据如果被换出，整个内核都面临崩溃的危险。

但在实验三实现的 ucore 中，我们只是实现了换入换出机制，还没有设计用户态执行的程序，所以我们在实验三中仅仅通过执行 `check_swap` 函数在内核中分配一些页，模拟对这些页的访问，然后通过 `do_pgfault` 来调用 `swap_map_swappable` 函数来查询这些页的访问情况并间接调用相关函数，换出“不常用”的页到磁盘上。

处理

`pgfault_handler`

回想一下，我们处理异常的时候，是在 `kern/trap/trap.c` 的 `exception_handler()` 函数里进行的。按照 `scause` 寄存器对异常的分类里，有 `CAUSE_LOAD_PAGE_FAULT` 和

CAUSE_STORE_PAGE_FAULT 两个case。之前我们并没有真正对异常进行处理，只是简单输出一下就返回了。现在我们要真正进行Page Fault的处理。

```
1 // kern/trap/trap.c
2 static inline void print_pgfault(struct trapframe *tf) {
3     cprintf("page fault at 0x%08x: %c/%c\n", tf->badvaddr,
4             trap_in_kernel(tf) ? 'K' : 'U',
5             tf->cause == CAUSE_STORE_PAGE_FAULT ? 'W' : 'R');
6 }
7
8 static int pgfault_handler(struct trapframe *tf) {
9     extern struct mm_struct *check_mm_struct;
10    print_pgfault(tf);
11    if (check_mm_struct != NULL) {
12        return do_pgfault(check_mm_struct, tf->cause, tf->badvaddr);
13    }
14    panic("unhandled page fault.\n");
15 }
16
17 void exception_handler(struct trapframe *tf) {
18     int ret;
19     switch (tf->cause) {
20         /* .... other cases */
21         case CAUSE_FETCH_PAGE_FAULT: // 取指令时发生的Page Fault先不处理
22             cprintf("Instruction page fault\n");
23             break;
24         case CAUSE_LOAD_PAGE_FAULT:
25             cprintf("Load page fault\n");
26             if ((ret = pgfault_handler(tf)) != 0) {
27                 print_trapframe(tf);
28                 panic("handle pgfault failed. %e\n", ret);
29             }
30             break;
31         case CAUSE_STORE_PAGE_FAULT:
32             cprintf("Store/AMO page fault\n");
33             if ((ret = pgfault_handler(tf)) != 0) { //do_pgfault()页面置换成
34                 print_trapframe(tf);
35                 panic("handle pgfault failed. %e\n", ret);
36             }
37             break;
38         default:
39             print_trapframe(tf);
40             break;
41     }
42 }
```

这里的异常处理程序，把Page Fault分发给 `kern/mm/vmm.c` 的 `do_pgfault()` 函数，尝试进行页面置换。接下来我们处理多级页表。之前的初始页表占据一个页的物理内存，只有一个页表项是有用的，映射了一个大大页(Giga Page)。

地址转换

之前我们物理页帧管理有个功能没有实现，那就是动态的内存分配。管理虚拟内存的数据结构（页表）需要有空间进行存储，而我们又没有给它预先分配内存（也无法预先分配，因为事先不确定我们的页表需要分配多少内存），就需要有 `malloc/free` 的接口来分配释放内存。我们在这里顺便看看 `pmm.h` 里对物理页面和虚拟地址，物理地址进行转换的一些函数。

```
1 // kern/mm/pmm.c
2 void *kmalloc(size_t n) { //分配至少n个连续的字节，这里实现得不精细，占用的只能是
3     void *ptr = NULL;
4     struct Page *base = NULL;
5     assert(n > 0 && n < 1024 * 0124);
6     int num_pages = (n + PGSIZE - 1) / PGSIZE; //向上取整到整数个页
7     base = alloc_pages(num_pages);
8     assert(base != NULL); //如果分配失败就直接panic
9     ptr = page2kva(base); //分配的内存的起始位置（虚拟地址），
10    //page2kva，就是page_to_kernel_virtual_address
11    return ptr;
12 }
13
14 void kfree(void *ptr, size_t n) { //从某个位置开始释放n个字节
15     assert(n > 0 && n < 1024 * 0124);
16     assert(ptr != NULL);
17     struct Page *base = NULL;
18     int num_pages = (n + PGSIZE - 1) / PGSIZE;
19     /*计算num_pages和kmalloc里一样，
20     但是如果程序员写错了呢？调用kfree的时候传入的n和调用kmalloc传入的n不一样？
21     就像你平时在windows/linux写C语言一样，会出各种奇奇怪怪的bug。
22     */
23     base = kva2page(ptr); //kernel_virtual_address_to_page
24     free_pages(base, num_pages);
25 }
26 // kern/mm/pmm.h
27 /*
28 KADDR, PADDR进行的是物理地址和虚拟地址的互换
29 由于我们在ucore里实现的页表映射很简单，所有物理地址和虚拟地址的偏移值相同，
30 所以这两个宏本质上只是做了一步加法/减法，额外还做了一些合法性检查。
31 */
32 /* *
```

```

33  * PADDR - takes a kernel virtual address (an address that points above
34  * KERNBASE),
35  * where the machine's maximum 256MB of physical memory is mapped and returns
36  * the
37  * corresponding physical address. It panics if you pass it a non-kernel
38  * virtual address.
39  * */
40  #define PADDR(kva) \
41      ({ \
42          uintptr_t __m_kva = (uintptr_t)(kva); \
43          if (__m_kva < KERNBASE) { \
44              panic("PADDR called with invalid kva %08lx", __m_kva); \
45          } \
46          __m_kva - va_pa_offset; \
47      })
48
49  /* *
50  * KADDR - takes a physical address and returns the corresponding kernel v
51  * address. It panics if you pass an invalid physical address.
52  * */
53  #define KADDR(pa) \
54      ({ \
55          uintptr_t __m_pa = (pa); \
56          size_t __m_ppn = PPN(__m_pa); \
57          if (__m_ppn >= npage) { \
58              panic("KADDR called with invalid pa %08lx", __m_pa); \
59          } \
60          (void *)(__m_pa + va_pa_offset); \
61      })
62
63  extern struct Page *pages;
64  extern size_t npage;
65  extern const size_t nbase;
66  extern uint_t va_pa_offset;
67
68  /*
69  我们曾经在内存里分配了一堆连续的Page结构体，来管理物理页面。可以把它们看作一个结构体
70  pages指针是这个数组的起始地址，减一下，加上一个基准值nbase，就可以得到正确的物理页号
71  pages指针和nbase基准值我们都在其他地方做了正确的初始化
72  */
73  static inline ppn_t page2ppn(struct Page *page) { return page - pages + nbase; }
74  /*
75  指向某个Page结构体的指针，对应一个物理页面，也对应一个起始的物理地址。
76  左移若干位就可以从物理页号得到页面的起始物理地址。
77  */
78  static inline uintptr_t page2pa(struct Page *page) {
79      return page2ppn(page) << PGSHIFT;
80  }
81  /*
82  倒过来，从物理页面的地址得到所在的物理页面。实际上是得到管理这个物理页面的Page结构体
83  */

```

```

84 static inline struct Page *pa2page(uintptr_t pa) {
85     if (PPN(pa) >= npage) {
86         panic("pa2page called with invalid pa");
87     }
88     return &pages[PPN(pa) - nbase]; //把pages指针当作数组使用
89 }
90
91 static inline void *page2kva(struct Page *page) { return KADDR(page2pa(page)); }
92
93 static inline struct Page *kva2page(void *kva) { return pa2page(PADDR(kva)); }
94
95 //从页表项得到对应的页，这里用到了 PTE_ADDR(pte)宏，对页表项做操作，在mmu.h里
96 static inline struct Page *pte2page(pte_t pte) {
97     if (!(pte & PTE_V)) {
98         panic("pte2page called with invalid pte");
99     }
100    return pa2page(PTE_ADDR(pte));
101 }
102 //PDE(Page Directory Entry)指的是不在叶节点的页表项（指向低一级页表的页表项）
103 static inline struct Page *pde2page(pde_t pde) { //PDE_ADDR这个宏和PTE_ADDR类似
104     return pa2page(PDE_ADDR(pde));
105 }

```

使用多级页表

我们需要把页表放在内存里，并且需要有办法修改页表，比如在页表里增加一个页面的映射或者删除某个页面的映射。

最主要的是两个接口：

`page_insert()`，在页表里建立一个映射

`page_remove()`，在页表里删除一个映射

这些我们都在 `kern/mm/pmm.c` 里面编写。

我们来看 `page_insert()`, `page_remove()` 的实现。注意它们都要调用两个对页表项进行操作的函数：`get_pte()` 和 `page_remove_pte()`

```
1  int page_insert(pde_t *pgdir, struct Page *page, uintptr_t la, uint32_t perm) {
2      //pgdir是页表基址(satp)，page对应物理页面，la是虚拟地址
3      pte_t *ptep = get_pte(pgdir, la, 1);
4      //先找到对应页表项的位置，如果原先不存在，get_pte()会分配页表项的内存
5      if (ptep == NULL) {
6          return -E_NO_MEM;
7      }
8      page_ref_inc(page); //指向这个物理页面的虚拟地址增加了一个
9      if (*ptep & PTE_V) { //原先存在映射
10         struct Page *p = pte2page(*ptep);
11         if (p == page) { //如果这个映射原先就有
12             page_ref_dec(page);
13         } else { //如果原先这个虚拟地址映射到其他物理页面，那么需要删除映射
14             page_remove_pte(pgdir, la, ptep);
15         }
16     }
17     *ptep = pte_create(page2ppn(page), PTE_V | perm); //构造页表项
18     tlb_invalidate(pgdir, la); //页表改变之后要刷新TLB
19     return 0;
20 }
21 void page_remove(pde_t *pgdir, uintptr_t la) {
22     pte_t *ptep = get_pte(pgdir, la, 0); //找到页表项所在位置
23     if (ptep != NULL) {
24         page_remove_pte(pgdir, la, ptep); //删除这个页表项的映射
25     }
26 }
```

```

27 //删除一个页表项以及它的映射
28 static inline void page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep
29     if (*ptep & PTE_V) { //(1) check if this page table entry is valid
30         struct Page *page = pte2page(*ptep); //(2) find corresponding pag
31         page_ref_dec(page); //(3) decrease page reference
32         if (page_ref(page) == 0) {
33             //(4) and free this page when page reference reaches 0
34             free_page(page);
35         }
36         *ptep = 0; //(5) clear page table entry
37         tlb_invalidate(pgdir, la); //(6) flush tlb
38     }
39 }
40 //寻找(有必要的时候分配)一个页表项
41 pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create) {
42     /* LAB2 EXERCISE 2: YOUR CODE
43     *
44     * If you need to visit a physical address, please use KADDR()
45     * please read pmm.h for useful macros
46     *
47     * Maybe you want help comment, BELOW comments can help you finish the
48     *
49     * Some Useful MACROs and DEFINES, you can use them in below implement
50     * MACROs or Functions:
51     *   PDX(la) = the index of page directory entry of VIRTUAL ADDRESS la
52     *   KADDR(pa) : takes a physical address and returns the corresponding
53     *   kernel virtual address.
54     *   set_page_ref(page,1) : means the page be referenced by one time
55     *   page2pa(page): get the physical address of memory which this (str
56     *   Page *) page manages
57     *   struct Page * alloc_page() : allocation a page
58     *   memset(void *s, char c, size_t n) : sets the first n bytes of the
59     *   memory area pointed by s
60     *
61     *                                     to the specified value c.
62     *   DEFINES:
63     *   PTE_P           0x001           // page table/directory e
64     *   flags bit : Present
65     *   PTE_W           0x002           // page table/directory e
66     *   flags bit : Writeable
67     *   PTE_U           0x004           // page table/directory e
68     *   flags bit : User can access
69     */
70     pde_t *pdep1 = &pgdir[PDX1(la)]; //找到对应的Giga Page
71     if (!(*pdep1 & PTE_V)) { //如果下一级页表不存在，那就给它分配一页，创造新页表
72         struct Page *page;
73         if (!create || (page = alloc_page()) == NULL) {
74             return NULL;
75         }
76         set_page_ref(page, 1);
77         uintptr_t pa = page2pa(page);
78         memset(KADDR(pa), 0, PGSIZE);

```



```

78         //我们现在在虚拟地址空间中，所以要转化为KADDR再memset.
79         //不管页表怎么构造，我们确保物理地址和虚拟地址的偏移量始终相同，那么就可以用
80         *pdep1 = pte_create(page2ppn(page), PTE_U | PTE_V); //注意这里R,W,X全
81     }
82     pde_t *pdep0 = &((pde_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(la)]; //再下一级
83     //这里的逻辑和前面完全一致，页表不存在就现在分配一个
84     if (!(*pdep0 & PTE_V)) {
85         struct Page *page;
86         if (!create || (page = alloc_page()) == NULL) {
87             return NULL;
88         }
89         set_page_ref(page, 1);
90         uintptr_t pa = page2pa(page);
91         memset(KADDR(pa), 0, PGSIZE);
92         *pdep0 = pte_create(page2ppn(page), PTE_U | PTE_V);
93     }
94     //找到输入的虚拟地址la对应的页表项的地址(可能是刚刚分配的)
95     return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(la)];
96 }

```

页面置换机制

结构体

现在来看看ucore页面置换机制的实现。

页面置换机制中，我们需要维护一些“不在内存当中但是也许会用到”的页，它们存储在磁盘的交换区里，也有对应的虚拟地址，但是因为它们不在内存里，在页表里并没有对它们的虚拟地址进行映射。但是在发生Page Fault之后，会把访问到的页放到内存里，这时也许会把其他页扔出去，来给要用的页腾出地方。页面置换算法的核心任务，主要就是确定“把谁扔出去”。

页表里的信息只是“当前哪些数据在内存条里以及它们物理地址和虚拟地址的对应关系”，这里我们需要一些页表之外的数据结构来维护当前页表里没映射的页。也就是这些信息：有哪些虚拟地址对应的页当前在磁盘上，分别在磁盘上的哪个位置。有哪些虚拟地址对应的页面当前放在内存里。这两类页面（内存上的，磁盘上的）会相互转换（换入/换出内存），所以我们将这两类页面一起维护，也就是维护“所有可用的虚拟地址/虚拟页的集合”（不论当前这个虚拟地址对应的页在内存上还是在硬盘上）。之后我们将要实现进程机制，对于不同的进程，可用的虚拟地址（虚拟页）的集合常常是不一样的，因此每个进程需要一个页表，也需要一个数据结构来维护“所有可用的虚拟地址”。

我们在vmm.h定义两个结构体（vmm：virtual memory management）。

`vma_struct` 结构体描述一段连续的虚拟地址，从 `vm_start` 到 `vm_end`。通过包含一个 `list_entry_t` 成员，我们可以把同一个页表对应的多个 `vma_struct` 结构体串成一个链表，在链表里把它们按照区间的起始点进行排序。

`vm_flags` 表示的是一段虚拟地址对应的权限（可读，可写，可执行等），这个权限在页表项里也要进行对应的设置。

我们注意到，每个页表（每个虚拟地址空间）可能包含多个 `vma_struct`，也就是多个访问权限可能不同的，不相交连续地址区间。我们用 `mm_struct` 结构体把一个页表对应的信息组合起来，包括 `vma_struct` 链表的首指针，对应的页表在内存里的指针，`vma_struct` 链表的元素个数。

```

1 // kern/mm/vmm.h
2 //pre define
3 struct mm_struct;
4
5 // the virtual continuous memory area(vma), [vm_start, vm_end),
6 // addr belong to a vma means vma.vm_start<= addr <vma.vm_end
7 struct vma_struct {
8     struct mm_struct *vm_mm; // the set of vma using the same PDT
9     uintptr_t vm_start;      // start addr of vma
10    uintptr_t vm_end;        // end addr of vma, not include the vm_end it
11    uint_t vm_flags;         // flags of vma
12    list_entry_t list_link;  // linear list link which sorted by start add
13 };
14
15 #define le2vma(le, member) \
16     to_struct((le), struct vma_struct, member)
17
18 #define VM_READ            0x00000001
19 #define VM_WRITE           0x00000002
20 #define VM_EXEC            0x00000004
21
22 // the control struct for a set of vma using the same Page Table
23 struct mm_struct {
24     list_entry_t mmap_list; // linear list link which sorted by sta
25     struct vma_struct *mmap_cache; // current accessed vma, used for speed
26     pde_t *pgdir;           // the Page Table of these vma
27     int map_count;          // the count of these vma
28     void *sm_priv;          // the private data for swap manager
29 };

```

函数

create

我们需要为 `vma_struct` 和 `mm_struct` 定义和实现一些接口：包括它们的构造函数，以及如何把新的 `vma_struct` 插入到 `mm_struct` 对应的链表里。注意这两个结构体占用的内存空间需要用 `kmalloc()` 函数动态分配。

```

1 // kern/mm/vmm.c

```

```

2 // mm_create - alloc a mm_struct & initialize it.
3 struct mm_struct *
4 mm_create(void) {
5     struct mm_struct *mm = kmalloc(sizeof(struct mm_struct));
6
7     if (mm != NULL) {
8         list_init(&(mm->mmap_list));
9         mm->mmap_cache = NULL;
10        mm->pgdir = NULL;
11        mm->map_count = 0;
12
13        if (swap_init_ok) swap_init_mm(mm); //我们接下来解释页面置换的初始化
14        else mm->sm_priv = NULL;
15    }
16    return mm;
17 }
18
19 // vma_create - alloc a vma_struct & initialize it. (addr range: vm_start~
20 struct vma_struct *
21 vma_create(uintptr_t vm_start, uintptr_t vm_end, uint_t vm_flags) {
22     struct vma_struct *vma = kmalloc(sizeof(struct vma_struct));
23     if (vma != NULL) {
24         vma->vm_start = vm_start;
25         vma->vm_end = vm_end;
26         vma->vm_flags = vm_flags;
27     }
28     return vma;
29 }

```

check_vma_overlap

在插入一个新的 `vma_struct` 之前，我们要保证它和原有的区间都不重合。

```

1 // kern/mm/vmm.c
2 // check_vma_overlap - check if vma1 overlaps vma2 ?
3 static inline void
4 check_vma_overlap(struct vma_struct *prev, struct vma_struct *next) {
5     assert(prev->vm_start < prev->vm_end);
6     assert(prev->vm_end <= next->vm_start);
7     assert(next->vm_start < next->vm_end); // next 是我们想插入的区间，这里顺便
8 }

```

insert_vma_struct & find_vma

我们可以插入一个新的 `vma_struct` ,也可以查找某个虚拟地址对应的 `vma_struct` 是否存在

```
1 // kern/mm/vmm.c
2
3 // insert_vma_struct -insert vma in mm's list link
4 void
5 insert_vma_struct(struct mm_struct *mm, struct vma_struct *vma) {
6     assert(vma->vm_start < vma->vm_end);
7     list_entry_t *list = &(mm->mmap_list);
8     list_entry_t *le_prev = list, *le_next;
9
10    list_entry_t *le = list;
11    while ((le = list_next(le)) != list) {
12        struct vma_struct *mmap_prev = le2vma(le, list_link);
13        if (mmap_prev->vm_start > vma->vm_start) {
14            break;
15        }
16        le_prev = le;
17    }
18    //保证插入后所有vma_struct按照区间左端点有序排列
19    le_next = list_next(le_prev);
20
21    /* check overlap */
22    if (le_prev != list) {
23        check_vma_overlap(le2vma(le_prev, list_link), vma);
24    }
25    if (le_next != list) {
26        check_vma_overlap(vma, le2vma(le_next, list_link));
27    }
28
29    vma->vm_mm = mm;
30    list_add_after(le_prev, &(vma->list_link));
31
32    mm->map_count ++; //计数器
33 }
34
35 // find_vma - find a vma (vma->vm_start <= addr <= vma->vm_end)
36 //如果返回NULL,说明查询的虚拟地址不存在/不合法,既不对应内存里的某个页,也不对应硬盘
37 struct vma_struct *
38 find_vma(struct mm_struct *mm, uintptr_t addr) {
39     struct vma_struct *vma = NULL;
40     if (mm != NULL) {
41         vma = mm->mmap_cache;
42         if (!(vma != NULL && vma->vm_start <= addr && vma->vm_end > addr))
43             bool found = 0;
```

```

44         list_entry_t *list = &(mm->mmap_list), *le = list;
45         while ((le = list_next(le)) != list) {
46             vma = le2vma(le, list_link);
47             if (vma->vm_start<=addr && addr < vma->vm_end) {
48                 found = 1;
49                 break;
50             }
51         }
52         if (!found) {
53             vma = NULL;
54         }
55     }
56     if (vma != NULL) {
57         mm->mmap_cache = vma;
58     }
59 }
60 return vma;
61 }

```

pgfault_handler

考虑当发生Page Fault的时候我们怎么做。回顾异常处理程序，我们的 `trapFrame` 传递了 `badvaddr` 给 `do_pgfault()` 函数。这实际上是 `stval` 这个寄存器的数值（在旧版的RISC-V标准里叫做 `sbadvaddr`），这个寄存器存储一些关于异常的数据，对于 `PageFault` 它存储的是访问出错的虚拟地址。

```

1  // kern/trap/trap.c
2  static int pgfault_handler(struct trapframe *tf) {
3      extern struct mm_struct *check_mm_struct; //当前使用的mm_struct的指针，在v
4      print_pgfault(tf);
5      if (check_mm_struct != NULL) {
6          return do_pgfault(check_mm_struct, tf->cause, tf->badvaddr);
7      }
8      panic("unhandled page fault.\n");
9  }
10 // kern/mm/vmm.c
11 struct mm_struct *check_mm_struct;
12
13 // check_pgfault - check correctness of pgfault handler
14 static void
15 check_pgfault(void) {
16     /* ..... */
17     check_mm_struct = mm_create();
18     /* ..... */

```

do_pgfault()

do_pgfault() 函数在 vmm.c 定义，是页面置换机制的核心。如果可行，我们要对页表做对应的修改，加入对应的页表项，并把硬盘上的数据换进内存。这时可能要把内存里的一个页换出去。

```

1 // kern/mm/vmm.c
2 int do_pgfault(struct mm_struct *mm, uint_t error_code, uintptr_t addr) {
3     //addr: 访问出错的虚拟地址
4     int ret = -E_INVAL;
5     //try to find a vma which include addr
6     struct vma_struct *vma = find_vma(mm, addr);
7     //我们首先要做的就是判断在mm_struct里判断这个虚拟地址是否可用
8     pgfault_num++;
9     //If the addr is not in the range of a mm's vma?
10    if (vma == NULL || vma->vm_start > addr) {
11        cprintf("not valid addr %x, and can not find it in vma\n", addr);
12        goto failed;
13    }
14
15    /* IF (write an existed addr ) OR
16     *   (write an non_existed addr && addr is writable) OR
17     *   (read an non_existed addr && addr is readable)
18     * THEN
19     *   continue process
20     */
21    uint32_t perm = PTE_U;
22    if (vma->vm_flags & VM_WRITE) {
23        perm |= (PTE_R | PTE_W);
24    }
25    addr = ROUNDDOWN(addr, PGSIZE); //按照页面大小把地址对齐
26
27    ret = -E_NO_MEM;
28
29    pte_t *ptep=NULL;
30
31    ptep = get_pte(mm->pgdir, addr, 1); // (1) try to find a pte, if pte's
32                                         //PT(Page Table) isn't existed, t
33                                         //create a PT.
34    if (*ptep == 0) {
35        if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) {
36            cprintf("pgdir_alloc_page in do_pgfault failed\n");

```

```

37         goto failed;
38     }
39 } else {
40     /*
41     * Now we think this pte is a swap entry, we should load data from
42     * to a page with phy addr,
43     * and map the phy addr with logical addr, trigger swap manager to
44     * the access situation of this page.
45     *
46     * swap_in(mm, addr, &page) : alloc a memory page, then according
47     * the swap entry in PTE for addr, find the addr of disk page, read
48     * content of disk page into this memory page
49     * page_insert : build the map of phy addr of an Page with the
50     * swap_map_swappable : set the page swappable
51     */
52     if (swap_init_ok) {
53         struct Page *page = NULL;
54         //在swap_in()函数执行完之后，page保存换入的物理页面。
55         //swap_in()函数里面可能把内存里原有的页面换出去
56         swap_in(mm, addr, &page); //(1) According to the mm AND addr,
57                                     //to load the content of right disk
58                                     //into the memory which page manage
59         page_insert(mm->pgdir, page, addr, perm); //更新页表，插入新的页表
60         //(2) According to the mm, addr AND page,
61         // setup the map of phy addr <---> virtual addr
62         swap_map_swappable(mm, addr, page, 1); //(3) make the page swappable
63         //标记这个页面将来是可以再换出的
64         page->pra_vaddr = addr;
65     } else {
66         cprintf("no swap_init_ok but ptep is %x, failed\n", *ptep);
67         goto failed;
68     }
69 }
70
71 ret = 0;
72 failed:
73     return ret;
74 }

```


FIFO置换算法

所谓FIFO(First in, First out)页面置换算法，相当简单，就是把所有页面排在一个队列里，每次换入页面的时候，把队列里最靠前（最早被换入）的页面置换出去。

换出页面的时机相对复杂一些，针对不同的策略有不同的时机。ucore 目前大致有两种策略，即积极换出策略和消极换出策略。积极换出策略是指操作系统周期性地（或在系统不忙的时候）主动把某些认为“不常用”的页换出到硬盘上，从而确保系统中总有一定数量的空闲页存在，这样当需要空闲页时，基本上能够及时满足需求；消极换出策略是指，只是当试图得到空闲页时，发现当前没有空闲的物理页可供分配，这时才开始查找“不常用”页面，并把一个或多个这样的页换出到硬盘上。

alloc_pages

目前的框架支持第二种情况，在 `alloc_pages()` 里面，没有空闲的物理页时，尝试换出页面到硬盘上。

```
1 // kern/mm/pmm.c
2 // alloc_pages - call pmm->alloc_pages to allocate a continuous n*PAGESIZE
3 struct Page *alloc_pages(size_t n) {
4     struct Page *page = NULL;
5     bool intr_flag;
6
7     while (1) {
8         local_intr_save(intr_flag);
9         { page = pmm_manager->alloc_pages(n); }
10        local_intr_restore(intr_flag);
11        //如果有足够的物理页面，就不必换出其他页面
12        //如果n>1，说明希望分配多个连续的页面，但是我们换出页面的时候并不能换出连续
13        //swap_init_ok标志是否成功初始化了
14        if (page != NULL || n > 1 || swap_init_ok == 0) break;
15
16        extern struct mm_struct *check_mm_struct;
17        swap_out(check_mm_struct, n, 0); //调用页面置换的“换出页面”接口。这里必
18    }
19    return page;
20 }
```

swap_manager

类似 `pmm_manager` ,我们定义 `swap_manager` ,组合页面置换需要的一些函数接口。

```
1 // kern/mm/swap.h
2
3 struct swap_manager
4 {
5     const char *name;
6     /* Global initialization for the swap manager */
7     int (*init)          (void);
8     /* Initialize the priv data inside mm_struct */
9     int (*init_mm)       (struct mm_struct *mm);
10    /* Called when tick interrupt occurred */
11    int (*tick_event)     (struct mm_struct *mm);
12    /* Called when map a swappable page into the mm_struct */
13    int (*map_swappable)  (struct mm_struct *mm, uintptr_t addr, struct
14    /* When a page is marked as shared, this routine is called to
15    * delete the addr entry from the swap manager */
16    int (*set_unswappable) (struct mm_struct *mm, uintptr_t addr);
17    /* Try to swap out a page, return then victim */
18    int (*swap_out_victim) (struct mm_struct *mm, struct Page **ptr_page,
19    /* check the page replacement algorithm */
20    int (*check_swap)(void);
21 };
```

swap_in/out

我们来看 `swap_in()` , `swap_out()` 如何换入/换出一个页面.注意我们对物理页面的 `Page` 结构体做了一些改动。

```
1 // kern/mm/memlayout.h
2 struct Page {
3     int ref;                // page frame's reference counter
4     uint_t flags;           // array of flags that describe the stat
5     unsigned int property;  // the num of free block, used in firs
6     list_entry_t page_link; // free list link
7     list_entry_t pra_page_link; // used for pra (page replace algorithm
```

```

8     uintptr_t pra_vaddr;           // used for pra (page replace algorithm)
9 };
10
11 // kern/mm/swap.c
12 int swap_in(struct mm_struct *mm, uintptr_t addr, struct Page **ptr_result)
13 {
14     struct Page *result = alloc_page(); // 这里alloc_page()内部可能调用swap_out
15     // 找到对应的一个物理页面
16     assert(result != NULL);
17
18     pte_t *ptep = get_pte(mm->pgdir, addr, 0); // 找到/构建对应的页表项
19     // 将物理地址映射到虚拟地址是在swap_in()退出之后，调用page_insert()完成的
20     int r;
21     if ((r = swapfs_read(*ptep, result)) != 0) // 将数据从硬盘读到内存
22     {
23         assert(r != 0);
24     }
25     cprintf("swap_in: load disk swap entry %d with swap_page in vaddr 0x%x\n", addr,
26             *ptr_result = result);
27     return 0;
28 }
29 int swap_out(struct mm_struct *mm, int n, int in_tick)
30 {
31     int i;
32     for (i = 0; i != n; ++i)
33     {
34         uintptr_t v;
35         struct Page *page;
36         int r = sm->swap_out_victim(mm, &page, in_tick); // 调用页面置换算法
37         // r=0表示成功找到了可以换出去的页面
38         // 要换出去的物理页面存在page里
39         if (r != 0) {
40             cprintf("i %d, swap_out: call swap_out_victim failed\n", i);
41             break;
42         }
43
44         cprintf("SWAP: choose victim page 0x%08x\n", page);
45
46         v = page->pra_vaddr; // 可以获取物理页面对应的虚拟地址
47         pte_t *ptep = get_pte(mm->pgdir, v, 0);
48         assert((*ptep & PTE_V) != 0);
49
50         if (swapfs_write((page->pra_vaddr / PGSIZE + 1) << 8, page) != 0) {
51             // 尝试把要换出的物理页面写到硬盘上的交换区，返回值不为0说明失败
52             cprintf("SWAP: failed to save\n");
53             sm->map_swappable(mm, v, page, 0);
54             continue;
55         }
56         else {
57             // 成功换出
58             cprintf("swap_out: i %d, store page in vaddr 0x%x to disk\n", i, v);

```

```

59         *ptep = (page->pra_vaddr/PGSIZE+1)<<8;
60         free_page(page);
61     }
62     //由于页表改变了，需要刷新TLB
63     //思考： swap_in()的时候插入新的页表项之后在哪里刷新了TLB?
64     tlb_invalidate(mm->pgdir, v);
65 }
66 return i;
67 }

```

swap_init

kern/mm/swap.c 里其他的接口基本都是简单调用 `swap_manager` 的具体实现。值得一提的是 `swap_init()` 初始化里做的工作。

```

1  // kern/mm/swap.c
2  static struct swap_manager *sm;
3  int swap_init(void)
4  {
5      swapfs_init();
6
7      // Since the IDE is faked, it can only store 7 pages at most to pass
8      if (!(7 <= max_swap_offset &&
9          max_swap_offset < MAX_SWAP_OFFSET_LIMIT)) {
10         panic("bad max_swap_offset %08x.\n", max_swap_offset);
11     }
12
13     sm = &swap_manager_fifo;//use first in first out Page Replacement Alg
14     int r = sm->init();
15
16     if (r == 0)
17     {
18         swap_init_ok = 1;
19         cprintf("SWAP: manager = %s\n", sm->name);
20         check_swap();
21     }
22
23     return r;
24 }
25
26 int swap_init_mm(struct mm_struct *mm)
27 {

```

```

28     return sm->init_mm(mm);
29 }
30
31 int swap_tick_event(struct mm_struct *mm)
32 {
33     return sm->tick_event(mm);
34 }
35
36 int swap_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *
37 {
38     return sm->map_swappable(mm, addr, page, swap_in);
39 }
40
41 int swap_set_unswappable(struct mm_struct *mm, uintptr_t addr)
42 {
43     return sm->set_unswappable(mm, addr);
44 }

```

swap_fifo.h

kern/mm/swap_fifo.h 完成了FIFO置换算法最终的具体实现。我们所做的就是维护了一个队列（用链表实现）。

```

1  // kern/mm/swap_fifo.h
2  #ifndef __KERN_MM_SWAP_FIFO_H__
3  #define __KERN_MM_SWAP_FIFO_H__
4
5  #include <swap.h>
6  extern struct swap_manager swap_manager_fifo;
7
8  #endif
9  // kern/mm/swap_fifo.c
10 /* Details of FIFO PRA
11  * (1) Prepare: In order to implement FIFO PRA, we should manage all swapp
12  * link these pages into pra_list_head according the time order. At first
13  * be familiar to the struct list in list.h. struct list is a simple doubl
14  * implementation. You should know howto USE: list_init, list_add(list_add
15  * list_add_before, list_del, list_next, list_prev. Another tricky method
16  * a general list struct to a special struct (such as struct page). You c
17  * le2page (in memlayout.h), (in future labs: le2vma (in vmm.h), le2proc (
18  */
19

```

```

20 list_entry_t pra_list_head;
21 /*
22  * (2) _fifo_init_mm: init pra_list_head and let mm->sm_priv point to the
23  *      Now, From the memory control struct mm_struct, we can access FIFO P
24  */
25 static int
26 _fifo_init_mm(struct mm_struct *mm)
27 {
28     list_init(&pra_list_head);
29     mm->sm_priv = &pra_list_head;
30     return 0;
31 }
32 /*
33  * (3)_fifo_map_swappable: According FIFO PRA, we should link the most rec
34  */
35 static int
36 _fifo_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *pag
37 {
38     list_entry_t *head=(list_entry_t*) mm->sm_priv;
39     list_entry_t *entry=&(page->pra_page_link);
40
41     assert(entry != NULL && head != NULL);
42     //record the page access situation
43     //(1)link the most recent arrival page at the back of the pra_list_head
44     list_add(head, entry);
45     return 0;
46 }
47 /*
48  * (4)_fifo_swap_out_victim: According FIFO PRA, we should unlink the ea
49  *      then set the addr of addr of this page to ptr_page.
50  */
51 static int
52 _fifo_swap_out_victim(struct mm_struct *mm, struct Page ** ptr_page, int i
53 {
54     list_entry_t *head=(list_entry_t*) mm->sm_priv;
55     assert(head != NULL);
56     assert(in_tick==0);
57     /* Select the victim */
58     //(1) unlink the earliest arrival page in front of pra_list_head qe
59     //(2) set the addr of addr of this page to ptr_page
60     list_entry_t* entry = list_prev(head);
61     if (entry != head) {
62         list_del(entry);
63         *ptr_page = le2page(entry, pra_page_link);
64     } else {
65         *ptr_page = NULL;
66     }
67     return 0;
68 }
69 static int _fifo_init(void)//初始化的时候什么都不做
70 {

```

```

71     return 0;
72 }
73
74 static int _fifo_set_unswappable(struct mm_struct *mm, uintptr_t addr)
75 {
76     return 0;
77 }
78
79 static int _fifo_tick_event(struct mm_struct *mm) //时钟中断的时候什么都不做
80 { return 0; }
81
82
83 struct swap_manager swap_manager_fifo =
84 {
85     .name           = "fifo swap manager",
86     .init           = &_fifo_init,
87     .init_mm        = &_fifo_init_mm,
88     .tick_event      = &_fifo_tick_event,
89     .map_swappable   = &_fifo_map_swappable,
90     .set_unswappable = &_fifo_set_unswappable,
91     .swap_out_victim = &_fifo_swap_out_victim,
92     .check_swap      = &_fifo_check_swap,
93 };

```

我们通过 `_fifo_check_swap()` , `check_swap()` , `check_vma_struct()` , `check_pgfault()` 等接口对页面置换机制进行了简单的测试。

项目组成与执行流

项目组成

```
1  lab3
2  |— Makefile
3  |— kern
4  |   |— debug
5  |   |   |— assert.h
6  |   |   |— kdebug.c
7  |   |   |— kdebug.h
8  |   |   |— kmonitor.c
9  |   |   |— kmonitor.h
10 |   |   |— panic.c
11 |   |   |— stab.h
12 |   |— driver
13 |   |   |— clock.c
14 |   |   |— clock.h
15 |   |   |— console.c
16 |   |   |— console.h
17 |   |   |— ide.c
18 |   |   |— ide.h
19 |   |   |— intr.c
20 |   |   |— intr.h
21 |   |— fs
22 |   |   |— fs.h
23 |   |   |— swapfs.c
24 |   |   |— swapfs.h
25 |   |— init
26 |   |   |— entry.S
27 |   |   |— init.c
28 |   |— libs
29 |   |   |— stdio.c
30 |   |— mm
31 |   |   |— default_pmm.c
32 |   |   |— default_pmm.h
33 |   |   |— memlayout.h
34 |   |   |— mmu.h
35 |   |   |— pmm.c
36 |   |   |— pmm.h
37 |   |   |— swap.c
38 |   |   |— swap.h
39 |   |   |— swap_clock.c
40 |   |   |— swap_clock.h
41 |   |   |— swap_fifo.c
42 |   |   |— swap_fifo.h
```



```
43 | | | | vmm.c
44 | | | | vmm.h
45 | | | | sync
46 | | | | | sync.h
47 | | | | trap
48 | | | | | trap.c
49 | | | | | trap.h
50 | | | | | trapentry.S
51 | | lab3.md
52 | | libs
53 | | | | atomic.h
54 | | | | defs.h
55 | | | | error.h
56 | | | | list.h
57 | | | | printfmt.c
58 | | | | rand.c
59 | | | | readline.c
60 | | | | riscv.h
61 | | | | sbi.h
62 | | | | stdarg.h
63 | | | | stdio.h
64 | | | | stdlib.h
65 | | | | string.c
66 | | | | string.h
67 | | tools
68 | | | | boot.ld
69 | | | | function.mk
70 | | | | gdbinit
71 | | | | grade.sh
72 | | | | kernel.ld
73 | | | | sign.c
74 | | | | vector.c
75
76 11 directories, 62 files
```

页面定义

`kern/mm/memlayout.h` : 修改了 `struct Page` , 增加了两项 `pra_*` 成员结构, 其中 `pra_page_link` 可以用来建立描述各个页访问情况 (比如根据访问先后) 的链表。在本实验中会涉及使用这两个成员结构, 以及 `le2page` 等宏。

虚拟内存信息

`kern/mm/vmm.[ch]` : `vmm.h` 描述了 `mm_struct` , `vma_struct` 等表述可访问的虚存地址访问的一些信息, 下面会进一步详细讲解。 `vmm.c` 涉及 `mm,vma` 结构数据的创建/销毁/查找/插入等函数, 这些函数在 `check_vma` 、 `check_vmm` 等中被使用, 理解即可。而 `page fault` 处理相关的 `do_pgfault` 函数是本次实验需要涉及完成的。

替换算法框架

`kern/mm/swap.[ch]` : 定义了实现页替换算法类框架 `struct swap_manager` 。 `swap.c` 包含了对此页替换算法类框架的初始化、页换入/换出等各种函数实现。重点是要理解何时调用 `swap_out` 和 `swap_in` 函数。和如何在此框架下连接具体的页替换算法实现。

FIFO算法

`kern/mm/swap_fifo.[ch]` : 演示的算法实现。

Clock算法

`kern/mm/swap_clock.[ch]` : 需要自己实现的算法, 有注释提示。

执行流

结合前面所述自行理解、总结。

LAB4：进程管理

在lab2和lab3中，我们已经将物理内存纳入了掌控，并且实现了虚拟内存的机制，使得我们可以建立一些真正操作系统级别的抽象。在本章和下一章当中，我们要实现操作系统当中非常重要的一个部分：进程管理。我们主要分成了两个部分来实现，在本章中我们会实现内核进程的管理，在下一章再实现用户进程的管理。

内核进程和用户进程有什么区别呢？首先，内核进程运行于内核态，而用户进程一般处于用户态，只有在需要系统调用时才会进入内核态。其次，内核进程不需要很复杂的内存管理，共用整个内核内存空间。这是因为内核进程往往用来完成很多和操作系统有关的任务，操作系统应当信任内核进程；而用户进程由用户提供，为了避免恶意的用户影响操作系统以及其他进程的运行状态，需要对于地址空间进行隔离。

本次实验主要介绍ucore如何实现内核进程。

实验目的

1. 了解内核进程创建、执行的管理过程
2. 了解内核线程的切换和基本调度过程

实验内容

1. 阅读线程、进程相关知识。
2. 阅读框架代码，深刻理解内核进程创建、切换过程。
3. 自己动手参与创建一个内核线程。

练习

练习1：分配并初始化一个进程控制块

`alloc_proc` 函数（位于 `kern/process/proc.c` 中）负责分配并返回一个新的 `struct proc_struct` 结构，用于存储新建立的内核线程的管理信息。`ucore` 需要对这个结构进行最基本的初始化，你需要完成这个初始化过程。

- ✔ 在 `alloc_proc` 函数的实现中，需要初始化的 `proc_struct` 结构中的成员变量至少包括：
`state/pid/runs/kstack/need_resched/parent/mm/context/tf/cr3/flags/name`。

请在实验报告中简要说明你的设计实现过程。并回答如下问题：

- 请说明 `proc_struct` 中 `struct context context` 和 `struct trapframe *tf` 成员变量含义和在本实验中的作用是啥？（提示：通过看代码和编程调试可以判断出来）

练习2：为新创建的内核线程分配资源

创建一个内核线程需要分配和设置好很多资源。`kernel_thread` 函数通过调用 `do_fork` 函数完成具体内核线程的创建工作。`do_kernel` 函数会调用 `alloc_proc` 函数来分配并初始化一个进程控制块，但 `alloc_proc` 只是找到了一小块内存用以记录进程的必要信息，并没有实际分配这些资源。`ucore` 一般通过 `do_fork` 实际创建新的内核线程。`do_fork` 的作用是，创建当前内核线程的一个副本，它们的执行上下文、代码、数据都一样，但是存储位置不同。在这个过程中，需要给新内核线程分配资源，并且复制原进程的状态。你需要完成在 `kern/process/proc.c` 中的 `do_fork` 函数中的处理过程。它的大致执行步骤包括：

- 调用 `alloc_proc`，首先获得一块用户信息块。
- 为进程分配一个内核栈。
- 复制原进程的内存管理信息到新进程（但内核线程不必做此事）
- 复制原进程上下文到新进程

- 将新进程添加到进程列表
- 唤醒新进程
- 返回新进程号

请在实验报告中简要说明你的设计实现过程。请回答如下问题：

- 请说明 ucore 是否做到给每个新 fork 的线程一个唯一的 id？请说明你的分析和理由。

练习3：理解proc_run函数和调用的函数如何完成进程切换的

请在实验报告中简要说明你对 proc_run 函数的分析。并回答如下问题：

- 在本实验的执行过程中，创建且运行了几个内核线程？
- 语句 `local_intr_save(intr_flag);...local_intr_restore(intr_flag);` 在这里有何作用？请说明理由

进程与线程

进程与线程

在操作系统中，我们经常谈到的两个概念就是进程与线程的概念。这两个概念虽然有许多相似的地方，但也有很多的不同。

我们平时编写的源代码，经过编译器编译就变成了可执行文件，我们管这一类文件叫做**程序**。而当一个程序被用户或操作系统启动，分配资源，装载进内存开始执行后，它就成为了一个**进程**。进程与程序之间最大的不同在于进程是一个“正在运行”的实体，而程序只是一个不动的文件。进程包含程序的内容，也就是它的静态的代码部分，也包括一些在运行时在可以体现出来的信息，比如堆栈，寄存器等数据，这些组成了进程“正在运行”的特性。

如果我们只关注于那些“正在运行”的部分，我们就从进程当中剥离出来了**线程**。一个进程可以对应一个线程，也可以对应很多线程。这些线程之间往往具有相同的代码，共享一块内存，但是却有不同CPU执行状态。相比于线程，进程更多的作为一个资源管理的实体（因为操作系统分配网络等资源时往往是基于进程的），这样线程就作为可以被调度的最小单元，给了调度器更多的调度可能。

为什么需要进程

进程的一个重要特点在于其可以调度。在我们操作系统启动的时候，操作系统相当是一个初始的进程。之后，操作系统会创建不同的进程负责不同的任务。用户可以通过命令行启动进程，从而使用计算机。想想如果没有进程会怎么样？所有的代码可能需要在操作系统编译的时候就打包在一块，安装软件将变成一件非常难的事情，这显然对于用户使用计算机是不利的。

另一方面，从2000年开始，CPU越来越多的使用多核心的设计。这主要是因为芯片设计师们发现在一个核心上提高主频变得越来越难（这其中有许多原因，相信组成原理课上已经有过介绍），所以采用多个核心，将利用多核性能的任务交给了程序员。在这种环境下，操作系统也需要进行相应的调整，以适应这种多核的趋势。使用进程的概念有助于各个进程同时的利用CPU的各个核心，这是单进程系统往往做不到的。

但是，多进程的引入其实远早于多核心处理器。在计算机的远古时代，存在许多“巨无霸”计算机。但是，如果只让这些计算机服务于一个用户，有时候又有点浪费。有没有可能让一个计算机服务于多个用户呢（哪怕只有一个核心）？分时操作系统解决了这个问题，就是通过时间片轮转的方法使得多个用户可以“同时”使用计算资源。这个时候，引入进程的概念，成为操作系统调度的单元就显得十分必要了。

综合以上可以看出，操作系统的确离不开进程管理。在下一节，我们会介绍ucore中与进程相关的数据结构，看一看如果定义一个进程。

相关结构体

在实现和进程相关的代码之前，我们先来实现一些数据结构来帮助我们对于进程进行管理。在本章实现的进程管理模型中，我们主要维护两个数据结构：进程控制块和进程上下文。进程控制块维护进程的各个信息，包括内存映射，进程名等等。进程上下文里面保存了和进程运行状态相关的各个寄存器的值，这些是为了之后恢复进程运行状态用的。下面我们来看一看这两个数据结构。

进程控制块

我们在ucore中使用结构体 `struct proc_struct` 来保存和进程相关的控制信息。

`struct proc_struct` 内部结构如下：

```
1  struct proc_struct {
2      enum proc_state state;           // Process state
3      int pid;                         // Process ID
4      int runs;                        // the running times of Process
5      uintptr_t kstack;                // Process kernel stack
6      volatile bool need_resched;      // bool value: need to be resched
7      struct proc_struct *parent;      // the parent process
8      struct mm_struct *mm;            // Process's memory management
9      struct context context;          // Switch here to run process
10     struct trapframe *tf;             // Trap frame for current interrupt
11     uintptr_t cr3;                   // CR3 register: the base address of kernel
12     uint32_t flags;                  // Process flag
13     char name[PROC_NAME_LEN + 1];    // Process name
14     list_entry_t list_link;           // Process link list
15     list_entry_t hash_link;          // Process hash list
16 };
```

这里面值得我们关注的主要有以下几个成员变量：

- `parent`：里面保存了进程的父进程的指针。在内核中，只有内核创建的idle进程没有父进程，其他进程都有父进程。进程的父子关系组成了一棵进程树，这种父子关系有利于维护父进程对于子进程的一些特殊操作。

- `mm` : 这里面保存了内存管理的信息, 包括内存映射, 虚存管理等内容。具体内在实现可以参考之前的章节。
 - `context` : `context` 中保存了进程执行的上下文, 也就是几个关键的寄存器的值。这些寄存器的值用于在进程切换中还原之前进程的运行状态 (进程切换的详细过程在后面会介绍)。切换过程的实现在 `kern/process/switch.S` 。
 - `tf` : `tf` 里保存了进程的中断帧。当进程从用户空间跳进内核空间的时候, 进程的执行状态被保存在了中断帧中 (注意这里需要保存的执行状态数量不同于上下文切换)。系统调用可能会改变用户寄存器的值, 我们可以通过调整中断帧来使得系统调用返回特定的值。
 - `cr3` : `cr3` 寄存器是x86架构的特殊寄存器, 用来保存页表所在的基址。出于legacy的原因, 我们这里仍然保留了这个名字, 但其值仍然是页表基址所在的位置。
-

进程上下文

进程上下文使用结构体 `struct context` 保存, 其中包含了 `ra` , `sp` , `s0~s11` 共14个寄存器。

可能感兴趣的同学就会问了, 为什么我们不需要保存所有的寄存器呢? 这里我们巧妙地利用了编译器对于函数的处理。我们知道寄存器可以分为调用者保存 (caller-saved) 寄存器和被调用者保存 (callee-saved) 寄存器。因为线程切换在一个函数当中 (我们下一小节就会看到), 所以编译器会自动帮助我们生成保存和恢复调用者保存寄存器的代码, 在实际的进程切换过程中我们只需要保存被调用者保存寄存器就好啦!

进程模块初始化

创建idle进程

进程初始化的函数定义在文件 `kern/process/proc.c` 中的 `proc_init`。进程模块的初始化主要分为两步，首先创建第0个内核进程，`idle`。

```
1 // kern/process/proc.c
2 // proc_init - set up the first kernel thread idleproc "idle" by itself and
3 //             - create the second kernel thread init_main
4 void
5 proc_init(void) {
6     int i;
7
8     list_init(&proc_list); // 进程链表
9     for (i = 0; i < HASH_LIST_SIZE; i++) {
10         list_init(hash_list + i);
11     }
12
13     if ((idleproc = alloc_proc()) == NULL) { // 分配"第0个"进程 idle
14         panic("cannot alloc idleproc.\n");
15     }
16
17     idleproc->pid = 0;
18     idleproc->state = PROC_RUNNABLE;
19     idleproc->kstack = (uintptr_t)bootstack;
20     idleproc->need_resched = 1;
21     set_proc_name(idleproc, "idle");
22     nr_process++;
23     // 全局变量current保存当前正在执行的进程
24     current = idleproc;
25
26     int pid = kernel_thread(init_main, "Hello world!!", 0);
27     if (pid <= 0) {
28         panic("create init_main failed.\n");
29     }
30
31     initproc = find_proc(pid);
32     set_proc_name(initproc, "init");
33
34     assert(idleproc != NULL && idleproc->pid == 0);
35     assert(initproc != NULL && initproc->pid == 1);
36 }
```

在进程模块初始化时，首先需要初始化进程链表。进程链表就是把所有进程控制块串联起来的数据结构，可以记录和追踪每一个进程。然后，调用 `proc_alloc` 函数来为第一个进程分配其进程控制块。当我们的操作系统开始运行的时候，其实它已经可以被视作一个进程了。但是我们还没有为他设计好进程控制块，也就没法进行管理。`proc_alloc` 函数会使用 `kmalloc` 分配一段空间来保存进程控制块，并且设定一些初值告诉我们这个进程目前还在初始化中。

在分配完空间后，我们对于 `idle` 进程的控制块进行一定的初始化：

```
1  idleproc->pid = 0;
2  idleproc->state = PROC_RUNNABLE;
3  idleproc->kstack = (uintptr_t)bootstack;
4  idleproc->need_resched = 1;
5  set_proc_name(idleproc, "idle");
6  nr_process ++;
```

从这里开始，`idle` 进程具有了合法的进程编号，`0`。我们把 `idle` 进程的状态设置为 `RUNNABLE`，表示其可以执行。因为这是第一个内核进程，所以我们可以直接将 `ucore` 的启动栈分配给他。需要注意的是，后面再分配新进程时我们需要为其分配一个栈，而不能再使用启动栈了。我们再把 `idle` 进程标志为需要调度，这样一旦 `idle` 进程开始执行，马上就可以让调度器调度另一个进程进行执行。

创建内核进程

接下来我们对于第一个真正的内核进程进行初始化（因为 `idle` 进程仅仅算是“继承了”`ucore` 的运行）。我们的目标是使用新的内核进程进行一下内核初始化的工作，但在这章我们先仅仅让它输出一个 `Hello World`，证明我们的内核进程实现的没有问题。下面是创建内核进程的代码：

```
1  int
2  kernel_thread(int (*fn)(void *), void *arg, uint32_t clone_flags) {
3      struct trapframe tf;
4      memset(&tf, 0, sizeof(struct trapframe));
5  }
```

```

6     tf.gpr.s0 = (uintptr_t)fn;
7     tf.gpr.s1 = (uintptr_t)arg;
8     tf.status = (read_csr(sstatus) | SSTATUS_SPP | SSTATUS_SPIE) & ~SSTATUS_SIE;
9     tf.epc = (uintptr_t)kernel_thread_entry;
10    return do_fork(clone_flags | CLONE_VM, 0, &tf);
11 }

```

我们将寄存器 `s0` 和 `s1` 分别设置为需要进程执行的函数和相关参数列表，之后设置了 `status` 寄存器使得进程切换后处于中断使能的状态。我们还设置了 `epc` 使其指向 `kernel_thread_entry`，这是进程执行的入口函数。最后，调用 `do_fork` 函数把当前的进程复制一份。

`do_fork` 函数内部主要进行了如下操作：

1. 分配并初始化进程控制块（`alloc_proc` 函数）
2. 分配并初始化内核栈（`setup_stack` 函数）
3. 根据 `clone_flags` 决定是复制还是共享内存管理系统（`copy_mm` 函数）
4. 设置进程的中断帧和上下文（`copy_thread` 函数）
5. 把设置好的进程加入链表
6. 将新建的进程设为就绪态
7. 将返回值设为线程id

如果执行失败，则需要调用相应的错误处理函数释放空间。更多的实现细节可以参考代码，在练习中也会有更多的涉及。

在这里我们需要尤其关注 `copy_thread` 函数：

```

1  static void
2  copy_thread(struct proc_struct *proc, uintptr_t esp, struct trapframe *tf)
3  {
4      proc->tf = (struct trapframe *) (proc->kstack + KSTACKSIZE - sizeof(struct trapframe));
5      *(proc->tf) = *tf;
6
7      // Set a0 to 0 so a child process knows it's just forked
8      proc->tf->gpr.a0 = 0;
9      proc->tf->gpr.sp = (esp == 0) ? (uintptr_t)proc->tf : esp;
10
11     proc->context.ra = (uintptr_t)forkret;
12     proc->context.sp = (uintptr_t)(proc->tf);
13 }

```

在这里我们首先在上面分配的内核栈上分配出一片空间来保存 `trapframe`。然后，我们将 `trapframe` 中的 `a0` 寄存器（返回值）设置为0，说明这个进程是一个子进程。之后我们将上下文中的 `ra` 设置为了 `forkret` 函数的入口，并且把 `trapframe` 放在上下文的栈顶。

进程切换

在ucore完成初始化后，ucore内核就没事做了，于是进入了“idle”的状态（这也是我们给第0个进程起名叫 idle 的原因）。这是，它会陷入死循环，不断检查自己是否需要调度：

```
1 void
2 cpu_idle(void) {
3     while (1) {
4         if (current->need_resched) {
5             schedule();
6             .....
```

因为我们之前在初始化中把 idle 进程的 need_resched 设为了 1，所以其总会调用 schedule 函数来检查是否有进程可以调度。我们已经初始化了另外一个内核进程，所以调度器发现了这个进程，并且准备调度到这个进程。

我们实现的 schedule 函数非常的简单：当需要调度的时候，把当前的进程放在队尾，从队列中取出第一个可以运行的进程，切换到它运行。这就是FIFO调度算法。schedule 函数会调用 proc_run 来唤醒选定的进程。proc_run 函数内部如下：

```
1 void
2 proc_run(struct proc_struct *proc) {
3     if (proc != current) {
4         bool intr_flag;
5         struct proc_struct *prev = current, *next = proc;
6         local_intr_save(intr_flag);
7         {
8             current = proc;
9             lcr3(next->cr3);
10            switch_to(&(prev->context), &(next->context));
11        }
12        local_intr_restore(intr_flag);
13    }
14 }
```

函数中主要进行了三个操作：

1. 将当前运行的进程设置为要切换过去的进程
2. 将页表换成新进程的页表
3. 使用 `switch_to` 切换到新进程

`switch_to` 函数如下：

```
1  .text
2  # void switch_to(struct proc_struct* from, struct proc_struct* to)
3  .globl switch_to
4  switch_to:
5      # save from's registers
6      STORE ra, 0*REGBYTES(a0)
7      STORE sp, 1*REGBYTES(a0)
8      STORE s0, 2*REGBYTES(a0)
9      STORE s1, 3*REGBYTES(a0)
10     STORE s2, 4*REGBYTES(a0)
11     STORE s3, 5*REGBYTES(a0)
12     STORE s4, 6*REGBYTES(a0)
13     STORE s5, 7*REGBYTES(a0)
14     STORE s6, 8*REGBYTES(a0)
15     STORE s7, 9*REGBYTES(a0)
16     STORE s8, 10*REGBYTES(a0)
17     STORE s9, 11*REGBYTES(a0)
18     STORE s10, 12*REGBYTES(a0)
19     STORE s11, 13*REGBYTES(a0)
20
21     # restore to's registers
22     LOAD ra, 0*REGBYTES(a1)
23     LOAD sp, 1*REGBYTES(a1)
24     LOAD s0, 2*REGBYTES(a1)
25     LOAD s1, 3*REGBYTES(a1)
26     LOAD s2, 4*REGBYTES(a1)
27     LOAD s3, 5*REGBYTES(a1)
28     LOAD s4, 6*REGBYTES(a1)
29     LOAD s5, 7*REGBYTES(a1)
30     LOAD s6, 8*REGBYTES(a1)
31     LOAD s7, 9*REGBYTES(a1)
32     LOAD s8, 10*REGBYTES(a1)
33     LOAD s9, 11*REGBYTES(a1)
34     LOAD s10, 12*REGBYTES(a1)
35     LOAD s11, 13*REGBYTES(a1)
36
37     ret
```

可以看出这段代码就是将需要保存的寄存器进行保存和调换。在之前我们也已经谈到过了，这里只需要调换被调用者保存寄存器即可。由于我们在初始化时把上下文的 `ra` 寄存器设定成了 `forkret` 函数的入口，所以这里会返回到 `forkret` 函数，进一步进入到 `forkrets`。`forkrets` 函数很短：

```
1     .globl forkrets
2 forkrets:
3     # set stack to this new process's trapframe
4     move sp, a0
5     j __trapret
```

这里把传进来的参数，也就是进程的中断帧放在了 `sp`，这样在 `__trapret` 中就可以直接从中断帧里面恢复所有的寄存器啦！我们在初始化的时候对于中断帧做了一点手脚，`epc` 寄存器指向的是 `kernel_thread_entry`，`s0` 寄存器里放的是新进程要执行的函数，`s1` 寄存器里放的是传给函数的参数。在 `kernel_thread_entry` 函数中：

```
1 .text
2 .globl kernel_thread_entry
3 kernel_thread_entry:      # void kernel_thread(void)
4     move a0, s1
5     jalr s0
6
7     jal do_exit
```

我们把参数放在了 `a0` 寄存器，并跳转到 `s0` 执行我们指定的函数！这样，一个进程的初始化就完成了。至此，我们实现了基本的进程管理，并且成功创建并切换到了我们的第一个内核进程。

项目组成与执行流

项目组成

```
1 lab4
2 |— Makefile
3 |— kern
4 |   |— debug
5 |       |— assert.h
6 |       |— kdebug.c
7 |       |— kdebug.h
8 |       |— kmonitor.c
9 |       |— kmonitor.h
10 |      |— panic.c
11 |      |— stab.h
12 |   |— driver
13 |       |— clock.c
14 |       |— clock.h
15 |       |— console.c
16 |       |— console.h
17 |       |— ide.c
18 |       |— ide.h
19 |       |— intr.c
20 |       |— intr.h
21 |       |— kbdreg.h
22 |       |— picirq.c
23 |       |— picirq.h
24 |   |— fs
25 |       |— fs.h
26 |       |— swapfs.c
27 |       |— swapfs.h
28 |   |— init
29 |       |— entry.S
30 |       |— init.c
31 |   |— libs
32 |       |— readline.c
33 |       |— stdio.c
34 |   |— mm
35 |       |— default_pmm.c
36 |       |— default_pmm.h
37 |       |— kmalloc.c
38 |       |— kmalloc.h
39 |       |— memlayout.h
40 |       |— mmu.h
41 |       |— pmm.c
42 |       |— pmm.h
```

```
43 | | | | swap.c
44 | | | | swap.h
45 | | | | swap_fifo.c
46 | | | | swap_fifo.h
47 | | | | vmm.c
48 | | | | vmm.h
49 | | | process
50 | | | | entry.S
51 | | | | proc.c
52 | | | | proc.h
53 | | | | switch.S
54 | | | schedule
55 | | | | sched.c
56 | | | | sched.h
57 | | | sync
58 | | | | sync.h
59 | | | trap
60 | | | | trap.c
61 | | | | trap.h
62 | | | | trapentry.S
63 | | lab4.md
64 | | libs
65 | | | atomic.h
66 | | | defs.h
67 | | | elf.h
68 | | | error.h
69 | | | hash.c
70 | | | list.h
71 | | | printfmt.c
72 | | | rand.c
73 | | | riscv.h
74 | | | sbi.h
75 | | | stdarg.h
76 | | | stdio.h
77 | | | stdlib.h
78 | | | string.c
79 | | | string.h
80 | | tools
81 | | | boot.ld
82 | | | function.mk
83 | | | gdbinit
84 | | | grade.sh
85 | | | kernel.ld
86 | | | sign.c
87 | | | vector.c
88
89 13 directories, 73 files
```

进程管理

`kern/process/proc.[ch]` : 新增: 实现进程、线程相关功能, 包括: 创建进程/线程, 初始化进程/线程, 处理进程/线程退出等功能

`kern/process/entry.S` : 新增: 内核线程入口函数 `kernel_thread_entry` 的实现

`kern/process/switch.S` : 新增: 上下文切换, 利用堆栈保存、恢复进程上下文

进程系统初始化

`kern/init/init.c` : 修改: 完成进程系统初始化, 并在内核初始化后切入idle进程

进程调度

`kern/schedule/sched.[ch]` : 新增: 实现FIFO策略的进程调度

执行流

结合前面所述自行理解、总结。

LAB5：用户程序

之前我们已经实现了内存的管理和内核进程的建立。但是那都是在内核态。

接下来我们将在用户态运行一些程序。

用户程序，也就是我们在计算机系课程里一直在写的那些程序，到底怎样在操作系统上跑起来？

首先需要编译器把用户程序的源代码编译为可以在CPU执行的目标程序，这个目标程序里，既要有执行的代码，又要有关于内存分配的一些信息，告诉我们应该怎样为这个程序分配内存。

我们先不考虑怎样在ucore里运行编译器，只考虑ucore如何把编译好的用户程序运行起来。这需要给它分配一些内存，把程序代码加载进来，建立一个进程，然后通过调度让这个用户进程开始执行。

实验目的

1. 掌握第一个用户进程创建过程
2. 了解系统调用框架的实现机制
3. 了解用户进程是如何被管理的

实验内容

1. 构造出第一个用户进程。
2. 学会使用系统调用来运行不同的应用程序。
3. 完成对用户进程的执行过程的基本管理。

练习

练习1：解释 `do_fork()` 与 `alloc_proc()`

为了正常运行lab5, 需要对lab4已有的函数进行一些改动. 我们在lab5的代码框架里标注了两个有改动的函数, 一处是 `do_fork()`, 一处是 `alloc_proc()`, 请对每处改动解释一下如果lab5的代码里不做这个改动, 那么现在或者将来会出什么bug, 或者说会影响哪些功能。

练习2：编写 `load_icode` 函数

`do_execve`函数调用 `load_icode` (位于 `kern/process/proc.c` 中) 来加载并解析一个处于内存中的 ELF 执行文件格式的应用程序, 建立相应的用户内存空间来放置应用程序的代码段、数据段等, 且要设置好 `proc_struct` 结构中的成员变量 `trapframe` 中的内容, 确保在执行此进程后, 能够从应用程序设定的起始执行地址开始执行。需设置正确的 `trapframe` 内容。

请在实验报告中简要说明你的设计实现过程。

请在实验报告中描述当创建一个用户态进程并加载了应用程序后, CPU 是如何让这个应用程序最终在用户态执行起来的。即这个用户态进程被 `ucore` 选择占用 CPU 执行 (RUNNING 态) 到具体执行应用程序第一条指令的整个经过。

练习3：填写 `copy_range` 函数

创建子进程的函数 `do_fork` 在执行中将拷贝当前进程 (即父进程) 的用户内存地址空间中的合法内容到新进程中 (子进程), 完成内存资源的复制。具体是通过 `copy_range` 函数 (位于 `kern/mm/pmm.c` 中) 实现的, 请补充 `copy_range` 的实现, 确保能够正确执行。

请在实验报告中简要说明如何设计实现“Copy on Write 机制”, 给出概要设计, 鼓励给出详细设计。

- ① Copy-on-write (简称 COW) 的基本概念是指如果有多个使用者对一个资源 A (比如内存块) 进行读操作, 则每个使用者只需获得一个指向同一个资源 A 的指针, 就可以该资源了。若某使用者需要对这个资源 A 进行写操作, 系统会对该资源进行拷贝操作, 从而使得该“写操作”使用者获得一个该资源 A 的“私有”拷贝—资源 B, 可对资源 B 进行写操作。该“写操作”使用者对资源 B 的改变对于其他的使用者而言是不可见的, 因为其他使用者看到的还是资源 A。

用户进程

系统调用

操作系统应当提供给用户程序一些接口，让用户程序使用操作系统提供的服务。这些接口就是**系统调用**。用户程序在用户态运行(U mode), 系统调用在内核态执行(S mode)。这里有一个CPU的特权级切换的过程, 要用到 `ecall` 指令从U mode进入S mode。想想我们之前用 `ecall` 做过什么？在S mode调用OpenSBI提供的M mode接口。当时我们用 `ecall` 进入了M mode, 剩下的事情就交给OpenSBI来完成，然后我们收到OpenSBI返回的结果。

现在我们用 `ecall` 从U mode进入S mode之后，对应的处理需要我们编写内核系统调用的代码来完成。

另外，我们总不能让用户程序里直接调用 `ecall` 。通常我们会把这样的系统调用操作封装成一个个的函数，作为“标准库”提供给用户使用。例如在linux里，写一个C程序时使用 `printf()` 函数进行输出, 实际上是要进行 `write()` 的系统调用，通过内核把输出打印到命令行或其他地方。

对于用户进程的管理，有四个系统调用比较重要。

`sys_fork()`：把当前的进程复制一份，创建一个子进程，原先的进程是父进程。接下来两个进程都会收到 `sys_fork()` 的返回值，如果返回0说明当前位于子进程中，返回一个非0的值（子进程的PID）说明当前位于父进程中。然后就可以根据返回值的不同，在两个进程里进行不同的处理。

`sys_exec()`：在当前的进程下，停止原先正在运行的程序，开始执行一个新程序。PID不变，但是内存空间要重新分配，执行的机器代码发生了改变。我们可以用 `fork()` 和 `exec()` 配合，在当前程序不停止的情况下，开始执行另一个程序。

`sys_exit()`：退出当前的进程。

`sys_wait()`：挂起当前的进程，等到特定条件满足的时候再继续执行。

从内核线程到用户进程

在实验四中设计实现了进程控制块，并实现了内核线程的创建和简单的调度执行。但实验四中没有在用户态执行用户进程的管理机制，既无法体现用户进程的地址空间，以及用户进程间地址空间隔离的保护机制，不支持进程执行过程的用户态和核心态之间的切换，且没有用户进程的完整状态变化的生命周期。其实没有实现的原因是内核线程不需要这些功能。那内核线程相对于用户态线程有何特点呢？

但其实我们已经在实验四中看到了内核线程，内核线程的管理实现相对是简单的，其特点是直接使用操作系统（比如 ucore）在初始化中建立的内核虚拟内存地址空间，不同的内核线程之间可以通过调度器实现线程间的切换，达到分时使用 CPU 的目的。由于内核虚拟内存空间是一一映射计算机系统的物理空间的，这使得可用空间的大小不会超过物理空间大小，所以操作系统程序员编写内核线程时，需要考虑到有限的地址空间，需要保证各个内核线程在执行过程中不会破坏操作系统的正常运行。这样在实现内核线程管理时，不必考虑涉及与进程相关的虚拟内存管理中的缺页处理、按需分页、写时复制、页换入换出等功能。如果在内核线程执行过程中出现了访存错误异常或内存不够的情况，就认为操作系统出现错误了，操作系统将直接宕机。在 ucore 中，就是调用 `panic()` 函数，进入内核调试监控器

```
kernel_debug_monitor。
```

内核线程管理思想相对简单，但编写内核线程对程序员的要求很高。从理论上讲（理想情况），如果程序员都是能够编写操作系统级别的“高手”，能够勤俭和高效地使用计算机系统资源，且这些“高手”都为他人着想，具有奉献精神，在别的应用需要计算机资源的时候，能够从大局出发，从整个系统的执行效率出发，让出自己占用的资源，那这些“高手”编写出来的程序直接作为内核线程运行即可，也就没有用户进程存在的必要了。

但现实与理论的差距是巨大的，能编写操作系统的程序员是极少数的，与当前的应用程序员相比，估计大约差了 3~4 个数量级。如果还要求编写操作系统的程序员考虑其他未知程序员的未知需求，那这样的程序员估计可以成为是编程界的“上帝”了。

从应用程序编写和运行的角度看，既然程序员都不是“上帝”，操作系统程序员就需要给应用程序员编写的程序提供一个既“宽松”又“严格”的执行环境，让对内存大小和 CPU 使用时间等资源的限制没有仔细考虑的应用程序都能在操作系统中正常运行，且即使程序太可靠，也只能破坏自己，而不能破坏其他运行程序和整个系统。“严格”就是安全性保证，即应用程序执行不会破坏在内存中存在的其他应用程序和操作系统的内存空间等独占的资源；“宽松”就算是方便性支持，即提供给应用程序尽量丰富的服务功能和一个远大于物理内存空间的虚拟地址空间，使得应用程序在执行过程中不必考虑很多繁琐的细节（比如如何初始化 PCI 总线和外设等，如何管理物理内存等）。

让用户进程正常运行的用户环境

在操作系统原理的介绍中，一般提到进程的概念其实主要是指用户进程。从操作系统的设计和实现的角度看，其实用户进程是指一个应用程序在操作系统提供的一个用户环境中的一次执行过程。这里的重点是用户环境。用户环境有啥功能？用户环境指的是什么？

从功能上看，操作系统提供的这个用户环境有两方面的特点。一方面与存储空间相关，即限制用户进程可以访问的物理地址空间，且让各个用户进程之间的物理内存空间访问不重叠，这样可以保证不同用户进程之间不能相互破坏各自的内存空间，利用虚拟内存的功能（页换入换出）。给用户进程提供了远大于实际物理内存空间的虚拟内存空间。

另一方面与执行指令相关，即限制用户进程可执行的指令，不能让用户进程执行特权指令（比如修改页表起始地址），从而保证用户进程无法破坏系统。但如果不能执行特权指令，则很多功能（比如访问磁盘等）无法实现，所以需要某种机制，让操作系统完成需要特权指令才能做的各种服务功能，给用户进程一个“服务窗口”，用户进程可以通过这个“窗口”向操作系统提出服务请求，由操作系统来帮助用户进程完成需要特权指令才能做的各种服务。另外，还要有一个“中断窗口”，让用户进程不主动放弃使用 CPU 时，操作系统能够通过这个“中断窗口”强制让用户进程放弃使用 CPU，从而让其他用户进程有机会执行。

基于功能分析，我们就可以把这个用户环境定义为如下组成部分：

- 建立用户虚拟空间的页表和支持页换入换出机制的用户内存访问错误异常服务例程：提供地址隔离和超过物理空间大小的虚存空间。
- 应用程序执行的用户态 CPU 特权级：在用户态 CPU 特权级，应用程序只能执行一般指令，如果执行特权指令，结果不是无效就是产生“执行非法指令”异常；
- 系统调用机制：给用户进程提供“服务窗口”；
- 中断响应机制：给用户进程设置“中断窗口”，这样产生中断后，当前执行的用户进程将被强制打断，CPU 控制权将被操作系统的中断服务例程使用。

用户态进程的执行过程分析

在这个环境下运行的进程就是用户进程。那如果用户进程由于某种原因下面进入内核态后，那在内核态执行的是什么呢？还是用户进程吗？首先分析一下用户进程这样会进入内核态呢？回顾一下 lab1，就可以知道当产生外设中断、CPU 执行异常（比如访存错误）、陷入（系统调用），用户进程就会切换到内核中的操作系统中来。表面上看，到内核态后，操作

系统取得了 CPU 控制权，所以现在执行的应该是操作系统代码，由于此时 CPU 处于核心态特权级，所以操作系统的执行过程就应该是内核进程了。这样理解忽略了操作系统的具体实现。如果考虑操作系统的具体实现，应该如果来理解进程呢？

从进程控制块的角度看，如果执行了进程执行现场（上下文）的切换，就认为到另外一个进程执行了，及进程的分界点设定在执行进程切换的前后。到底切换了什么呢？其实只是切换了进程的页表和相关硬件寄存器，这些信息都保存在进程控制块中的相关域中。所以，我们可以把执行应用程序的代码一直到执行操作系统中的进程切换处为止都认为是一个应用程序的执行过程（其中有操作系统的部分代码执行过过程）即进程。因为在这个过程中，没有更换到另外一个进程控制块的进程的页表和相关硬件寄存器。

从指令执行的角度看，如果再仔细分析一下操作系统这个软件的特点并细化一下进入内核原因，就可以看出进一步进行划分。操作系统的主要功能是给上层应用提供服务，管理整个计算机系统资源。所以操作系统虽然是一个软件，但其实是一个基于事件的软件，这里操作系统需要响应的事件包括三类：外设中断、CPU 执行异常（比如访存错误）、陷入（系统调用）。如果用户进程通过系统调用要求操作系统提供服务，那么从用户进程的角度看，操作系统就是一个特殊的软件库（比如相对于用户态的 libc 库，操作系统可看作是内核态的 libc 库），完成用户进程的需求，从执行逻辑上看，是用户进程“主观”执行的一部分，即用户进程“知道”操作系统要做的事情。那么在这种情况下，进程的代码空间包括用户态的执行程序和内核态响应用户进程通过系统调用而在核心特权态执行服务请求的操作系统代码，为此这种情况下的进程的内存虚拟空间也包括两部分：用户态的虚地址空间和核心态的虚地址空间。但如果此时发生的事件是外设中断和 CPU 执行异常，虽然 CPU 控制权也转入到操作系统中的中断服务例程，但这些内核执行代码执行过程是用户进程“不知道”的，是另外一段执行逻辑。那么在这种情况下，实际上是执行了两段目标不同的执行程序，一个是代表应用程序的用户进程，一个是代表中断服务例程处理外设中断和 CPU 执行异常的内核线程。这个用户进程和内核线程在产生中断或异常的时候，CPU 硬件就完成了它们之间的指令流切换。

用户进程的运行状态分析

用户进程在其执行过程中会存在很多种不同的执行状态，根据操作系统原理，一个用户进程一般的运行状态有五种：创建（new）态、就绪（ready）态、运行（running）态、等待（blocked）态、退出（exit）态。各个状态之间会由于发生了某事件而进行状态转换。

但在用户进程的执行过程中，具体在哪个时间段处于上述状态的呢？上述状态是如何转变的呢？首先，我们看创建（new）态，操作系统完成进程的创建工作，而体现进程存在的就是

进程控制块，所以一旦操作系统创建了进程控制块，则可以认为此时进程就已经存在了，但由于进程能够运行的各种资源还没准备好，所以此时的进程处于创建（new）态。创建了进程控制块后，进程并不能就执行了，还需准备好各种资源，如果把进程执行所需要的虚拟内存空间，执行代码，要处理的数据等都准备好了，则此时进程已经可以执行了，但还没有被操作系统调度，需要等待操作系统选择这个进程执行，于是把这个做好“执行准备”的进程放入到一个队列中，并可以认为此时进程处于就绪（ready）态。当操作系统的调度器从就绪进程队列中选择一个就绪进程后，通过执行进程切换，就让这个被选上的就绪进程执行了，此时进程就处于运行（running）态了。到了运行态后，会出现三种事件。如果进程需要等待某个事件（比如主动睡眠 10 秒钟，或进程访问某个内存空间，但此内存空间被换出到硬盘 swap 分区中了，进程不得不等待操作系统把缓慢的硬盘上的数据重新读回到内存中），那么操作系统会把 CPU 给其他进程执行，并把进程状态从运行（running）态转换为等待（blocked）态。如果用户进程的应用程序逻辑流程执行结束了，那么操作系统会把 CPU 给其他进程执行，并把进程状态从运行（running）态转换为退出（exit）态，并准备回收用户进程占用的各种资源，当把表示整个进程存在的进程控制块也回收了，这进程就不存在了。在这整个回收过程中，进程都处于退出（exit）态。

2 考虑到在内存中存在多个处于就绪态的用户进程，但只有一个 CPU，所以为了公平起见，每个就绪态进程都只有有限的时间片段，当一个运行态的进程用完了它的时间片段后，操作系统会剥夺此进程的 CPU 使用权，并把此进程状态从运行（running）态转换为就绪（ready）态，最后把 CPU 给其他进程执行。如果某个处于等待（blocked）态的进程所等待的事件产生了（比如睡眠时间到，或需要访问的数据已经从硬盘换入到内存中），则操作系统会通过把等待此事件的进程状态从等待（blocked）态转到就绪（ready）态。这样进程的整个状态转换形成了一个有限状态自动机。

用户程序

应用程序的组成与编译

我们首先来看一个应用程序，这里我们假定是hello应用程序，在 `user/hello.c` 中实现，代码如下：

```
1  #include <stdio.h>
2  #include <ulib.h>
3
4  int main(void) {
5      cprintf("Hello world!!.\n");
6      cprintf("I am process %d.\n", getpid());
7      cprintf("hello pass.\n");
8      return 0;
9  }
```

hello应用程序只是输出一些字符串，并通过系统调用 `sys_getpid`（在 `getpid` 函数中调用）输出代表hello应用程序执行的用户进程的进程标识- `pid`。

首先，我们需要了解ucore操作系统如何能够找到hello应用程序。这需要分析ucore和hello是如何编译的。修改Makefile，把第六行注释掉。然后在本实验源码目录下执行make，可得到如下输出：

```
1  + cc user/hello.c
2  riscv64-unknown-elf-gcc -Iuser/ -mcmmodel=medany -O2 -std=gnu99 -Wno-unused
3
4  riscv64-unknown-elf-ld -m elf64lriscv -nostdlib --gc-sections -T tools/use
5
6  + ld bin/kernel
7  riscv64-unknown-elf-ld -m elf64lriscv -nostdlib --gc-sections -T tools/ker
8  riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img
```

从中可以看出，hello应用程序不仅仅是 `hello.c`，还包含了支持hello应用程序的用户态库：

- `user/libs/initcode.S` : 所有应用程序的起始用户态执行地址“`_start`”，调整了EBP和ESP后，调用 `umain` 函数。
- `user/libs/umain.c` : 实现了 `umain` 函数，这是所有应用程序执行的第一个C函数，它将调用应用程序的 `main` 函数，并在 `main` 函数结束后调用 `exit` 函数，而 `exit` 函数最终将调用 `sys_exit` 系统调用，让操作系统回收进程资源。
- `user/libs/ulib.[ch]` : 实现了最小的C函数库，除了一些与系统调用无关的函数，其他函数是对访问系统调用的包装。
- `user/libs/syscall.[ch]` : 用户层发出系统调用的具体实现。
- `user/libs/stdio.c` : 实现 `cprintf` 函数，通过系统调用 `sys_putc` 来完成字符输出。
- `user/libs/panic.c` : 实现 `__panic/__warn` 函数，通过系统调用 `sys_exit` 完成用户进程退出。

除了这些用户态库函数实现外，还有一些 `libs/*.ch` 是操作系统内核和应用程序共用的函数实现。这些用户库函数其实在本质上与UNIX系统中的标准libc没有区别，只是实现得很简单，但hello应用程序的正确执行离不开这些库函数。

[!NOTE|style:flat]

`libs/.ch`、`user/libs/.ch`、`user/*.ch`的源码中没有任何特权指令。

在make的最后一步执行了一个ld命令，把hello应用程序的执行码 `obj/__user_hello.out` 连接在了 `ucore kernel` 的末尾。且ld命令会在kernel中会把 `__user_hello.out` 的位置和大小记录在全局变量 `_binary_obj___user_hello_out_start` 和

`_binary_obj___user_hello_out_size` 中，这样这个hello用户程序就能够和ucore内核一起被 OpenSBI加载到内存里中，并且通过这两个全局变量定位hello用户程序执行码的起始位置和大小。而到了与文件系统相关的实验后，ucore会提供一个简单的文件系统，那时所有的用户程序就都不再用这种方法进行加载了，而可以用大家熟悉的文件方式进行加载了。

用户进程的虚拟地址空间

在tools/user.ld描述了用户程序的用户虚拟空间的执行入口虚拟地址：

```
1  SECTIONS {
2      /* Load programs at this address: "." means the current address */
3      . = 0x800020;
```

在tools/kernel.ld描述了操作系统的内核虚拟空间的起始入口虚拟地址：

```
1  BASE_ADDRESS = 0xFFFFFFFFC0200000;
2
3  SECTIONS
4  {
5      /* Load the kernel at this address: "." means the current address */
6      . = BASE_ADDRESS;
```

这样ucore把用户进程的虚拟地址空间分了两块，一块与内核线程一样，是所有用户进程都共享的内核虚拟地址空间，映射到同样的物理内存空间中，这样在物理内存中只需放置一份内核代码，使得用户进程从用户态进入核心态时，内核代码可以统一应对不同的内核程序；另外一块是用户虚拟地址空间，虽然虚拟地址范围一样，但映射到不同且没有交集的物理内存空间中。这样当ucore把用户进程的执行代码（即应用程序的执行代码）和数据（即应用程序的全局变量等）放到用户虚拟地址空间中时，确保了各个进程不会“非法”访问到其他进程的物理内存空间。

创建并执行用户进程

我们在 `proc_init()` 函数里初始化进程的时候, 认为启动时运行的 `ucore` 程序, 是一个内核进程("第0个"内核进程), 并将其初始化为 `idleproc` 进程。然后我们新建了一个内核进程执行 `init_main()` 函数。

我们比较 lab4 和 lab5 的 `init_main()` 有何不同。

```
1 // kern/process/proc.c (lab4)
2 static int init_main(void *arg) {
3     cprintf("this initproc, pid = %d, name = \"%s\"\n", current->pid, get_
4     cprintf("To U: \"%s\".\n", (const char *)arg);
5     cprintf("To U: \"en.., Bye, Bye. :)\"\n");
6     return 0;
7 }
8
9 // kern/process/proc.c (lab5)
10 static int init_main(void *arg) {
11     size_t nr_free_pages_store = nr_free_pages();
12     size_t kernel_allocated_store = kallocated();
13
14     int pid = kernel_thread(user_main, NULL, 0);
15     if (pid <= 0) {
16         panic("create user_main failed.\n");
17     }
18
19     while (do_wait(0, NULL) == 0) {
20         schedule();
21     }
22
23     cprintf("all user-mode processes have quit.\n");
24     assert(initproc->cptr == NULL && initproc->yptr == NULL && initproc->o
25     assert(nr_process == 2);
26     assert(list_next(&proc_list) == &(initproc->list_link));
27     assert(list_prev(&proc_list) == &(initproc->list_link));
28
29     cprintf("init check memory pass.\n");
30     return 0;
31 }
```

注意到, lab5 新建了一个内核进程, 执行函数 `user_main()`, 这个内核进程里我们将要开始执行用户进程。

`do_wait(0, NULL)` 等待子进程退出，也就是等待 `user_main()` 退出。

我们来看 `user_main()` 和 `do_wait()` 里做了什么

```
1 // kern/process/proc.c
2 #define __KERNEL_EXECVE(name, binary, size) ({
3     cprintf("kernel_execve: pid = %d, name = \"%s\".\n",
4             current->pid, name);
5     kernel_execve(name, binary, (size_t)(size));
6 })
7
8 #define KERNEL_EXECVE(x) ({
9     extern unsigned char _binary_obj__user_##x##_out_start[],
10     _binary_obj__user_##x##_out_size[];
11     __KERNEL_EXECVE(#x, _binary_obj__user_##x##_out_start,
12     _binary_obj__user_##x##_out_size);
13 })
14
15 #define __KERNEL_EXECVE2(x, xstart, xsize) ({
16     extern unsigned char xstart[], xsize[];
17     __KERNEL_EXECVE(#x, xstart, (size_t)xsize);
18 })
19
20 #define KERNEL_EXECVE2(x, xstart, xsize) __KERNEL_EXECVE2(x, xstart, xsize)
21
22 // user_main - kernel thread used to exec a user program
23 static int
24 user_main(void *arg) {
25     #ifdef TEST
26         KERNEL_EXECVE2(TEST, TESTSTART, TESTSIZE);
27     #else
28         KERNEL_EXECVE(exit);
29     #endif
30     panic("user_main execve failed.\n");
31 }
```

于是，我们在 `user_main()` 所做的，就是执行了

```
kern_execve("exit", _binary_obj__user_exit_out_start, _binary_obj__user_exit_out_size);
```

这么一个函数。

如果你熟悉 `execve()` 函数，或许已经猜到这里我们做了什么。

实际上，就是加载了存储在这个位置的程序 `exit` 并在 `user_main` 这个进程里开始执行。这时 `user_main` 就从内核进程变成了用户进程。我们在下一节介绍 `kern_execve()` 的实现。

我们在 `user` 目录下存储了一些用户程序，在编译的时候放到生成的镜像里。

系统调用

系统调用，是用户态(U mode)的程序获取内核态 (S mode)服务的方法，所以需要在用户态和内核态都加入对应的支持和处理。我们也可以认为用户态只是提供一个调用的接口，真正的处理都在内核态进行。

系统调用转发

首先我们在头文件里定义一些系统调用的编号。

```
1 // libs/unistd.h
2 #ifndef __LIBS_UNISTD_H__
3 #define __LIBS_UNISTD_H__
4
5 #define T_SYSCALL          0x80
6
7 /* syscall number */
8 #define SYS_exit           1
9 #define SYS_fork           2
10 #define SYS_wait           3
11 #define SYS_exec           4
12 #define SYS_clone          5
13 #define SYS_yield          10
14 #define SYS_sleep          11
15 #define SYS_kill           12
16 #define SYS_gettime        17
17 #define SYS_getpid         18
18 #define SYS_brk            19
19 #define SYS_mmap           20
20 #define SYS_munmap         21
21 #define SYS_shmem          22
22 #define SYS_putc           30
23 #define SYS_pgdir          31
24
25 /* SYS_fork flags */
26 #define CLONE_VM            0x00000100 // set if VM shared between proces
27 #define CLONE_THREAD        0x00000200 // thread group
28
29 #endif /* !__LIBS_UNISTD_H__ */
```

我们注意在用户态进行系统调用的核心操作是，通过内联汇编进行 `ecall` 环境调用。这将产生一个trap, 进入S mode进行异常处理。

```
1 // user/libs/syscall.c
2 #include <defs.h>
3 #include <unistd.h>
4 #include <stdarg.h>
5 #include <syscall.h>
6 #define MAX_ARGS 5
7 static inline int syscall(int num, ...) {
8     //va_list, va_start, va_arg都是C语言处理参数个数不定的函数的宏
9     //在stdarg.h里定义
10    va_list ap; //ap: 参数列表(此时未初始化)
11    va_start(ap, num); //初始化参数列表, 从num开始
12    //First, va_start initializes the list of variable arguments as a va_l
13    uint64_t a[MAX_ARGS];
14    int i, ret;
15    for (i = 0; i < MAX_ARGS; i++) { //把参数依次取出
16        /*Subsequent executions of va_arg yield the values of the addit
17        in the same order as passed to the function.*/
18        a[i] = va_arg(ap, uint64_t);
19    }
20    va_end(ap); //Finally, va_end shall be executed before the function re
21    asm volatile (
22        "ld a0, %1\n"
23        "ld a1, %2\n"
24        "ld a2, %3\n"
25        "ld a3, %4\n"
26        "ld a4, %5\n"
27        "ld a5, %6\n"
28        "ecall\n"
29        "sd a0, %0"
30        : "=m" (ret)
31        : "m"(num), "m"(a[0]), "m"(a[1]), "m"(a[2]), "m"(a[3]), "m"(a[4])
32        : "memory");
33    //num存到a0寄存器, a[0]存到a1寄存器
34    //ecall的返回值存到ret
35    return ret;
36 }
37 int sys_exit(int error_code) { return syscall(SYS_exit, error_code); }
38 int sys_fork(void) { return syscall(SYS_fork); }
39 int sys_wait(int pid, int *store) { return syscall(SYS_wait, pid, store); }
40 int sys_yield(void) { return syscall(SYS_yield); }
41 int sys_kill(int pid) { return syscall(SYS_kill, pid); }
42 int sys_getpid(void) { return syscall(SYS_getpid); }
43 int sys_putc(int c) { return syscall(SYS_putc, c); }
```

我们下面看看trap.c是如何转发这个系统调用的。

```
1 // kern/trap/trap.c
2 void exception_handler(struct trapframe *tf) {
3     int ret;
4     switch (tf->cause) { //通过中断帧里 scause寄存器的数值，判断出当前是来自USER
5         case CAUSE_USER_ECALL:
6             //cprintf("Environment call from U-mode\n");
7             tf->epc += 4;
8             //sepc寄存器是产生异常的指令的位置，在异常处理结束后，会回到sepc的位置
9             //对于ecall，我们希望sepc寄存器要指向产生异常的指令(ecall)的下一条指令
10            //否则就会回到ecall执行再执行一次ecall，无限循环
11            syscall(); // 进行系统调用处理
12            break;
13        /*other cases .... */
14    }
15 }
16 // kern/syscall/syscall.c
17 #include <unistd.h>
18 #include <proc.h>
19 #include <syscall.h>
20 #include <trap.h>
21 #include <stdio.h>
22 #include <pmm.h>
23 #include <assert.h>
24 //这里把系统调用进一步转发给proc.c的do_exit(), do_fork()等函数
25 static int sys_exit(uint64_t arg[]) {
26     int error_code = (int)arg[0];
27     return do_exit(error_code);
28 }
29 static int sys_fork(uint64_t arg[]) {
30     struct trapframe *tf = current->tf;
31     uintptr_t stack = tf->gpr.sp;
32     return do_fork(0, stack, tf);
33 }
34 static int sys_wait(uint64_t arg[]) {
35     int pid = (int)arg[0];
36     int *store = (int *)arg[1];
37     return do_wait(pid, store);
38 }
39 static int sys_exec(uint64_t arg[]) {
40     const char *name = (const char *)arg[0];
41     size_t len = (size_t)arg[1];
42     unsigned char *binary = (unsigned char *)arg[2];
43     size_t size = (size_t)arg[3];
44     //用户态调用的exec(), 归根结底是do_execve()
45     return do_execve(name, len, binary, size);
46 }
```



```

47 static int sys_yield(uint64_t arg[]) {
48     return do_yield();
49 }
50 static int sys_kill(uint64_t arg[]) {
51     int pid = (int)arg[0];
52     return do_kill(pid);
53 }
54 static int sys_getpid(uint64_t arg[]) {
55     return current->pid;
56 }
57 static int sys_putc(uint64_t arg[]) {
58     int c = (int)arg[0];
59     cputchar(c);
60     return 0;
61 }
62 //这里定义了函数指针的数组syscalls，把每个系统调用编号的下标上初始化为对应的函数指针
63 static int (*syscalls[])(uint64_t arg[]) = {
64     [SYS_exit]          sys_exit,
65     [SYS_fork]          sys_fork,
66     [SYS_wait]          sys_wait,
67     [SYS_exec]          sys_exec,
68     [SYS_yield]         sys_yield,
69     [SYS_kill]          sys_kill,
70     [SYS_getpid]        sys_getpid,
71     [SYS_putc]          sys_putc,
72 };
73
74 #define NUM_SYSCALLS      ((sizeof(syscalls)) / (sizeof(syscalls[0])))
75
76 void syscall(void) {
77     struct trapframe *tf = current->tf;
78     uint64_t arg[5];
79     int num = tf->gpr.a0;//a0寄存器保存了系统调用编号
80     if (num >= 0 && num < NUM_SYSCALLS) { //防止syscalls[num]下标越界
81         if (syscalls[num] != NULL) {
82             arg[0] = tf->gpr.a1;
83             arg[1] = tf->gpr.a2;
84             arg[2] = tf->gpr.a3;
85             arg[3] = tf->gpr.a4;
86             arg[4] = tf->gpr.a5;
87             tf->gpr.a0 = syscalls[num](arg);
88             //把寄存器里的参数取出来，转发给系统调用编号对应的函数进行处理
89             return ;
90         }
91     }
92     //如果执行到这里，说明传入的系统调用编号还没有被实现，就崩掉了。
93     print_trapframe(tf);
94     panic("undefined syscall %d, pid = %d, name = %s.\n",
95         num, current->pid, current->name);
96 }

```

这样我们就完成了系统调用的转发。接下来就是在 `do_exit()`, `do_execve()` 等函数中进行具体处理了。

do_execve()

我们看看 `do_execve()` 函数

```
1 // kern/mm/vmm.c
2 bool user_mem_check(struct mm_struct *mm, uintptr_t addr, size_t len, bool write) {
3     //检查从addr开始长为len的一段内存能否被用户态程序访问
4     if (mm != NULL) {
5         if (!USER_ACCESS(addr, addr + len)) {
6             return 0;
7         }
8         struct vma_struct *vma;
9         uintptr_t start = addr, end = addr + len;
10        while (start < end) {
11            if ((vma = find_vma(mm, start)) == NULL || start < vma->vm_start)
12                return 0;
13            if (!(vma->vm_flags & ((write) ? VM_WRITE : VM_READ))) {
14                return 0;
15            }
16            if (write && (vma->vm_flags & VM_STACK)) {
17                if (start < vma->vm_start + PGSIZE) { //check stack start
18                    return 0;
19                }
20            }
21            start = vma->vm_end;
22        }
23        return 1;
24    }
25    return KERN_ACCESS(addr, addr + len);
26 }
27
28 // kern/process/proc.c
29 // do_execve - call exit_mmap(mm)&put_pgdir(mm) to reclaim memory space of
30 //             - call load_icode to setup new memory space according binary
31 int do_execve(const char *name, size_t len, unsigned char *binary, size_t len) {
32     struct mm_struct *mm = current->mm;
33     if (!user_mem_check(mm, (uintptr_t)name, len, 0)) { //检查name的内存空间
34         return -E_INVALID;
```

```

35     }
36     if (len > PROC_NAME_LEN) { //进程名字的长度有上限 PROC_NAME_LEN, 在proc.h
37         len = PROC_NAME_LEN;
38     }
39     char local_name[PROC_NAME_LEN + 1];
40     memset(local_name, 0, sizeof(local_name));
41     memcpy(local_name, name, len);
42
43     if (mm != NULL) {
44         cputs("mm != NULL");
45         lcr3(boot_cr3);
46         if (mm_count_dec(mm) == 0) {
47             exit_mmap(mm);
48             put_pgdir(mm);
49             mm_destroy(mm); //把进程当前占用的内存释放，之后重新分配内存
50         }
51         current->mm = NULL;
52     }
53     //把新的程序加载到当前进程里的工作都在load_icode()函数里完成
54     int ret;
55     if ((ret = load_icode(binary, size)) != 0) {
56         goto execve_exit; //返回不为0，则加载失败
57     }
58     set_proc_name(current, local_name);
59     //如果set_proc_name的实现不变，为什么不能直接set_proc_name(current, name)?
60     return 0;
61
62 execve_exit:
63     do_exit(ret);
64     panic("already exit: %e.\n", ret);
65 }

```

kernel_execve()

那么我们如何实现 kernel_execve() 函数？

能否直接调用 do_execve() ？

```

1 // kern/process/proc.c
2 static int kernel_execve(const char *name, unsigned char *binary, size_t s
3     int64_t ret=0, len = strlen(name);
4     ret = do_execve(name, len, binary, size);

```

```

5     cprintf("ret = %d\n", ret);
6     return ret;
7 }

```

很不幸。这么做行不通。 `do_execve()` `load_icode()` 里面只是构建了用户程序运行的上下文，但是并没有完成切换。上下文切换实际上要借助中断处理的返回来完成。直接调用 `do_execve()` 是无法完成上下文切换的。如果是在用户态调用 `exec()`，系统调用的 `ecall` 产生的中断返回时，就可以完成上下文切换。

由于目前我们在S mode下，所以不能通过 `ecall` 来产生中断。我们这里采取一个取巧的办法，用 `ebreak` 产生断点中断进行处理，通过设置 `a7` 寄存器的值为10说明这不是一个普通的断点中断，而是要转发到 `syscall()`，这样用一个不是特别优雅的方式，实现了在内核态使用系统调用。

```

1 // kern/process/proc.c
2 // kernel_execve - do SYS_exec syscall to exec a user program called by us
3 static int kernel_execve(const char *name, unsigned char *binary, size_t s
4     int64_t ret=0, len = strlen(name);
5     asm volatile(
6         "li a0, %1\n"
7         "lw a1, %2\n"
8         "lw a2, %3\n"
9         "lw a3, %4\n"
10        "lw a4, %5\n"
11        "li a7, 10\n"
12        "ebreak\n"
13        "sw a0, %0\n"
14        : "=m"(ret)
15        : "i"(SYS_exec), "m"(name), "m"(len), "m"(binary), "m"(size)
16        : "memory"); //这里内联汇编的格式，和用户态调用ecall的格式类似，只是ecall
17    cprintf("ret = %d\n", ret);
18    return ret;
19 }
20 // kern/trap/trap.c
21 void exception_handler(struct trapframe *tf) {
22     int ret;
23     switch (tf->cause) {
24         case CAUSE_BREAKPOINT:
25             cprintf("Breakpoint\n");
26             if(tf->gpr.a7 == 10){
27                 tf->epc += 4; //注意返回时要执行ebreak的下一条指令
28                 syscall();
29             }

```

```
30         break;
31         /* other cases ... */
32     }
33 }
```

注意我们需要让CPU进入U mode执行 `do_execve()` 加载的用户程序。进行系统调用 `sys_exec` 之后，我们在trap返回的时候调用了 `sret` 指令，这时只要 `sstatus` 寄存器的 `SPP` 二进制位为0，就会切换到U mode，但 `SPP` 存储的是“进入trap之前来自什么特权级”，也就是说我们这里ebreak之后 `SPP` 的数值为1，`sret`之后会回到S mode在内核态执行用户程序。所以 `load_icode()` 函数在构造新进程的时候，会把 `SSTATUS_SPP` 设置为0，使得 `sret` 的时候能回到U mode。

用户进程的退出和等待

退出

在进程执行完工作后，需要退出，释放资源，正我们在 `do_execve` 函数末尾看到的一样，退出进程是调用 `do_exit` 函数来实现的。

```
1  execve_exit:
2      do_exit(ret);
3      panic("already exit: %e.\n", ret);
```

```
1  // do_exit - called by sys_exit
2  //  1. call exit_mmap & put_pgdir & mm_destroy to free the almost all mem
3  //  2. set process' state as PROC_ZOMBIE, then call wakeup_proc(parent) t
4  //  3. call scheduler to switch to other process
5  int
6  do_exit(int error_code) {
7      if (current == idleproc) {
8          panic("idleproc exit.\n");
9      }
10     if (current == initproc) {
11         panic("initproc exit.\n");
12     }
13     struct mm_struct *mm = current->mm;
14     if (mm != NULL) {
15         lcr3(boot_cr3);
16         if (mm_count_dec(mm) == 0) {
17             exit_mmap(mm);
18             put_pgdir(mm);
19             mm_destroy(mm);
20         }
21         current->mm = NULL;
22     }
23     current->state = PROC_ZOMBIE;
24     current->exit_code = error_code;
25     bool intr_flag;
26     struct proc_struct *proc;
27     local_intr_save(intr_flag);
28     {
29         proc = current->parent;
30         if (proc->wait_state == WT_CHILD) {
31             wakeup_proc(proc);
```

```

32     }
33     while (current->cptr != NULL) {
34         proc = current->cptr;
35         current->cptr = proc->optr;
36
37         proc->yptr = NULL;
38         if ((proc->optr = initproc->cptr) != NULL) {
39             initproc->cptr->yptr = proc;
40         }
41         proc->parent = initproc;
42         initproc->cptr = proc;
43         if (proc->state == PROC_ZOMBIE) {
44             if (initproc->wait_state == WT_CHILD) {
45                 wakeup_proc(initproc);
46             }
47         }
48     }
49 }
50 local_intr_restore(intr_flag);
51 schedule();
52 panic("do_exit will not return!! %d.\n", current->pid);
53 }

```

1. 如果是内核线程则不需要回收空间
2. 如果是用户进程，就开始回收，首先执行 `lcr3(boot_cr3)`；切换到内核的页表上，这样用户进程就只能在内核的虚拟地址空间上执行，因为内核权限高。如果当前进程的被调用数减一后等于0，那么就没有其他进程在使用了，就可以进行回收，先回收内存资源，调用 `exit_mmap` 函数释放 `mm` 中的 `vma` 描述的进程合法空间中实际分配的内存，然后把对应的页表项内容清空，最后把页表项和页目录表清空。然后调用 `put_pgdir` 函数释放页目录表所占用的内存。最后调用 `mm_destroy` 释放 `vma` 与 `mm` 的内存。把 `mm` 置为 `NULL`，表示与当前进程相关的用户虚拟内存空间和对应的内存管理成员变量所占的内核虚拟内存空间已经回收完毕；
3. 设置进程的状态为 `PROC_ZOMBIE` 表示该进程要死了，等待父进程来回收资源，回收内核栈和进程控制块。当前进程的退出码为 `error_code` 表示该进程已经不能被调度。
4. 如果当前进程的父进程处于等待子进程的状态，则唤醒父进程让父进程回收资源。
5. 如果该进程还有子进程，那么就指向第一个孩子，把后面的孩子全部置为空，然后把孩子过继给内核线程 `initproc`，把子进程插入到 `initproc` 的孩子链表中，如果某个子进程的状态时要死的状态，并且 `initproc` 的状态时等待孩子的状态，则唤醒 `initproc` 来回收子进程的资源。
6. 然后开启中断，执行 `schedule` 函数，选择新的进程执行

等待

那么父进程如何完成对子进程的最后回收工作呢？这要求父进程要执行 `wait` 用户函数或 `wait_pid` 用户函数，这两个函数的区别是，`wait` 函数等待任意子进程的结束通知，而 `wait_pid` 函数等待进程id号为pid的子进程结束通知。这两个函数最终访问 `sys_wait` 系统调用接口让ucore来完成对子进程的最后回收工作，即回收子进程的内核栈和进程控制块所占内存空间，具体流程如下：

```
1 // do_wait - wait one OR any children with PROC_ZOMBIE state, and free mem
2 //           - proc struct of this child.
3 // NOTE: only after do_wait function, all resources of the child proces ar
4 int
5 do_wait(int pid, int *code_store) {
6     struct mm_struct *mm = current->mm;
7     if (code_store != NULL) {
8         if (!user_mem_check(mm, (uintptr_t)code_store, sizeof(int), 1)) {
9             return -E_INVAL;
10        }
11    }
12
13    struct proc_struct *proc;
14    bool intr_flag, haskid;
15    repeat:
16        haskid = 0;
17        if (pid != 0) {
18            proc = find_proc(pid);
19            if (proc != NULL && proc->parent == current) {
20                haskid = 1;
21                if (proc->state == PROC_ZOMBIE) {
22                    goto found;
23                }
24            }
25        }
26        else {
27            proc = current->cptr;
28            for (; proc != NULL; proc = proc->optr) {
29                haskid = 1;
30                if (proc->state == PROC_ZOMBIE) {
31                    goto found;
32                }
33            }
34        }
35        if (haskid) {
36            current->state = PROC_SLEEPING;
37            current->wait_state = WT_CHILD;
```



```

38     schedule();
39     if (current->flags & PF_EXITING) {
40         do_exit(-E_KILLED);
41     }
42     goto repeat;
43 }
44 return -E_BAD_PROC;
45
46 found:
47 if (proc == idleproc || proc == initproc) {
48     panic("wait idleproc or initproc.\n");
49 }
50 if (code_store != NULL) {
51     *code_store = proc->exit_code;
52 }
53 local_intr_save(intr_flag);
54 {
55     unhash_proc(proc);
56     remove_links(proc);
57 }
58 local_intr_restore(intr_flag);
59 put_kstack(proc);
60 kfree(proc);
61 return 0;
62 }

```

1. 首先进行检查
2. 若 `pid` 等于0，就去找对应的孩子进程，否则就任意的一个快死的孩子进程。如果此子进程的执行状态不为 `PROC_ZOMBIE`，表明此子进程还没有退出，则当前进程只好设置自己的执行状态为 `PROC_SLEEPING`，睡眠原因为 `WT_CHILD`（即等待子进程退出），调用 `schedule()` 函数选择新的进程执行，自己睡眠等待，如果被唤醒，则重复该步骤执行；
3. 如果此子进程的执行状态为 `PROC_ZOMBIE`，表明此子进程处于退出状态，需要当前进程（即子进程的父进程）完成对子进程的最终回收工作，即首先把子进程控制块从两个进程队列 `proc_list` 和 `hash_list` 中删除，并释放子进程的内核堆栈和进程控制块。自此，子进程才彻底地结束了它的执行过程，消除了它所占用的所有资源。

项目组成与执行流

项目组成

```
1 lab5
2 |— Makefile
3 |— boot
4 |   |— asm.h
5 |   |— bootasm.S
6 |   |— bootmain.c
7 |— kern
8 |   |— debug
9 |       |— assert.h
10 |       |— kdebug.c
11 |       |— kdebug.h
12 |       |— kmonitor.c
13 |       |— kmonitor.h
14 |       |— panic.c
15 |       |— stab.h
16 |   |— driver
17 |       |— clock.c
18 |       |— clock.h
19 |       |— console.c
20 |       |— console.h
21 |       |— ide.c
22 |       |— ide.h
23 |       |— intr.c
24 |       |— intr.h
25 |       |— kbdreg.h
26 |       |— picirq.c
27 |       |— picirq.h
28 |   |— fs
29 |       |— fs.h
30 |       |— swapfs.c
31 |       |— swapfs.h
32 |   |— init
33 |       |— entry.S
34 |       |— init.c
35 |   |— libs
36 |       |— readline.c
37 |       |— stdio.c
38 |   |— mm
39 |       |— default_pmm.c
40 |       |— default_pmm.h
41 |       |— kmalloc.c
42 |       |— kmalloc.h
```

```
43 | | | |— memlayout.h
44 | | | |— mmu.h
45 | | | |— pmm.c
46 | | | |— pmm.h
47 | | | |— swap.c
48 | | | |— swap.h
49 | | | |— swap_fifo.c
50 | | | |— swap_fifo.h
51 | | | |— vmm.c
52 | | | |— vmm.h
53 | | |— process
54 | | | |— entry.S
55 | | | |— proc.c
56 | | | |— proc.h
57 | | | |— switch.S
58 | | |— schedule
59 | | | |— sched.c
60 | | | |— sched.h
61 | | |— sync
62 | | | |— sync.h
63 | | |— syscall
64 | | | |— syscall.c
65 | | | |— syscall.h
66 | | |— trap
67 | | | |— trap.c
68 | | | |— trap.h
69 | | | |— trapentry.S
70 |— lab5.md
71 |— libs
72 | |— atomic.h
73 | |— defs.h
74 | |— elf.h
75 | |— error.h
76 | |— hash.c
77 | |— list.h
78 | |— printfmt.c
79 | |— rand.c
80 | |— riscv.h
81 | |— sbi.h
82 | |— stdarg.h
83 | |— stdio.h
84 | |— stdlib.h
85 | |— string.c
86 | |— string.h
87 | |— unistd.h
88 |— tools
89 | |— boot.ld
90 | |— function.mk
91 | |— gdbinit
92 | |— grade.sh
93 | |— kernel.ld
```

```
94 |   |— sign.c
95 |   |— user.ld
96 |   |— vector.c
97 |   └─ user
98 |       |— badarg.c
99 |       |— badsegment.c
100 |       |— divzero.c
101 |       |— exit.c
102 |       |— faultread.c
103 |       |— faultreadkernel.c
104 |       |— forktest.c
105 |       |— forktree.c
106 |       |— hello.c
107 |       |— libs
108 |       |   |— initcode.S
109 |       |   |— panic.c
110 |       |   |— stdio.c
111 |       |   |— syscall.c
112 |       |   |— syscall.h
113 |       |   |— ulib.c
114 |       |   |— ulib.h
115 |       |   └─ umain.c
116 |       |— pgdir.c
117 |       |— softint.c
118 |       |— spin.c
119 |       |— testbss.c
120 |       |— waitkill.c
121 |       └─ yield.c
122
123 17 directories, 103 files
```

用户进程内存管理

kern/mm/pmm.[ch] : 添加了用于进程退出 (do_exit) 的内存资源回收的 page_remove_pte 、 unmap_range 、 exit_range 函数和用于创建子进程 (do_fork) 中拷贝父进程内存空间的 copy_range 函数, 修改了 pgdir_alloc_page 函数

kern/mm/vmm.[ch] : 修改: 扩展了 mm_struct 数据结构, 增加了一系列函数

- mm_map / dup_mmap / exit_mmap : 设定/取消/复制/删除用户进程的合法内存空间
- copy_from_user / copy_to_user : 用户内存空间内容与内核内存空间内容的相互拷贝的实现

- `user_mem_check` : 搜索vma链表, 检查是否是一个合法的用户空间范围
-

用户进程管理

`kern/process/proc.[ch]` : 扩展了 `proc_struct` 数据结构。增加或修改了一系列函数

- `setup_pgdir/put_pgdir` : 创建并设置/释放页目录表
 - `copy_mm` : 复制用户进程的内存空间和设置相关内存管理（如页表等）信息
 - `do_exit` : 释放进程自身所占内存空间和相关内存管理（如页表等）信息所占空间, 唤醒父进程, 好让父进程收了自己, 让调度器切换到其他进程
 - `load_icode` : 被 `do_execve` 调用, 完成加载放在内存中的执行程序到进程空间, 这涉及到对页表等的修改, 分配用户栈
 - `do_execve` : 先回收自身所占用户空间, 然后调用 `load_icode`, 用新的程序覆盖内存空间, 形成一个执行新程序的新进程
 - `do_yield` : 让调度器执行一次选择新进程的过程
 - `do_wait` : 父进程等待子进程, 并在得到子进程的退出消息后, 彻底回收子进程所占的资源（比如子进程的内核栈和进程控制块）
 - `do_kill` : 给一个进程设置 `PF_EXITING` 标志（“kill”信息, 即要它死掉）, 这样在trap函数中, 将根据此标志, 让进程退出
 - `KERNEL_EXECVE/__KERNEL_EXECVE/__KERNEL_EXECVE2` : 被`user_main`调用, 执行一用户进程
-

执行流

结合前面所述自行理解、总结。

LAB6：进程调度

在前两章中，我们已经分别实现了内核进程和用户进程，并且让他们正确运行了起来。我们同时也实现了一个简单的调度算法，FIFO调度算法，来对我们的进程进行调度。但是，单单如此就够了吗？显然，我们可以让ucore支持更加丰富的调度算法，从而满足各方面的调度需求。在本章里，我们要在调度框架的基础上实现各种各样的调度算法。

实验目的

1. 理解操作系统的调度管理机制
2. 熟悉ucore的系统调度器框架
3. 熟悉Round-Robin调度算法
4. 熟悉Stride Scheduling调度算法

实验内容

1. 掌握理论课有关调度算法的内容。
2. 阅读框架代码，掌握进程调度的主要流程。
3. 在框架的RR调度算法的基础上，实现Stride调度算法。

练习

练习1：比较函数

比较一个在lab5和lab6都有,但是实现不同的函数,说说为什么要做这个改动,不做这个改动会出什么问题。

✔ 如 `kern/schedule/sched.c` 里的函数。你也可以找个其他地方做了改动的函数。

练习2：分析 `sched_class`

理解并分析 `sched_class` 中各个函数指针的用法，并描述ucore如何通过Round Robin算法来调度**两个进程**，并解释`sched_class`里的**每个**函数（函数指针）是怎么被调用的。

练习3：二选一

1. 简要说明如何设计实现“多级反馈队列调度算法”，给出概要设计，鼓励给出详细设计
2. 简要证明/说明（不必特别严谨，但应当能够“说服你自己”），为什么Stride算法中，经过足够多的时间片之后，每个进程分配到的时间片数目和优先级成正比。

练习4：实现Stride调度算法

在 `kern/schedule/default_sched_stride.c` 填写 `Stride` 调度算法实现。

✔ 你需要调试 `proc.c` 中的一个bug，才能使调度算法正常运行。

进程状态

在本次实验中，我们在 `init/init.c` 中加入了对 `sched_init` 函数的调用。这个函数主要完成调度器和特定调度算法的绑定。初始化后，我们在调度函数中就可以使用相应的接口了。这也是在C语言环境下对于面向对象编程模式的一种模仿。这样之后，我们只需要关注于实现调度类的接口即可，操作系统也同样不关心调度类具体的实现，方便了新调度算法的开发。

在ucore中，进程有如下几个状态：

- `PROC_UNINIT` ：这个状态表示进程刚刚被分配相应的进程控制块，但还没有初始化，需要进一步的初始化才能进入 `PROC_RUNNABLE` 的状态。
- `PROC_SLEEPING` ：这个状态表示进程正在等待某个事件的发生，通常由于等待锁的释放，或者主动交出CPU资源（`do_sleep`）。这个状态下的进程是不会被调度的。
- `PROC_RUNNABLE` ：这个状态表示进程已经准备好要执行了，只需要操作系统给他分配相应的CPU资源就可以运行。
- `PROC_ZOMBIE` ：这个状态表示进程已经退出，相应的资源被回收（大部分），`almost dead`。

一个进程的生命周期一般由如下过程组成：

1. 刚刚开始初始化，进程处在 `PROC_UNINIT` 的状态 2. 进程已经完成初始化，时刻准备执行，进入 `PROC_RUNNABLE` 状态 3. 在调度的时候，调度器选中该进程进行执行，进程处在 `running` 的状态 4.(1) 正在运行的进程由于 `wait` 等系统调用被阻塞，进入 `PROC_SLEEPING`，等待相应的资源或者信号。 4.(2) 另一种可能是正在运行的进程被外部中断打断，此时进程变为 `PROC_RUNNABLE` 状态，等待下次被调用 5. 等待的事件发生，进程又变成 `PROC_RUNNABLE` 状态 6. 重复3~6，直到进程执行完毕，通过 `exit` 进入 `PROC_ZOMBIE` 状态，由父进程对他的资源进行回收，释放进程控制块。至此，这个进程的生命周期彻底结束。

再次认识进程切换

我们在第四章已经简单了解过了在内核启动过程中进程切换的过程，在这一节我们再来重新回顾一下这些内容，并且深入讨论下其中的几个细节。

首先我们需要考虑的是，什么时候可以进行进程的切换。这里可以主要分为下面几种情况：

- 1. 进程主动放弃当前的CPU资源，比如显式调用 `wait` 或 `sleep` 通知操作系统当前进程需要等待
- 2. 进程想要获取的资源当前不可用，比如尝试获得未被释放的锁，或进行磁盘操作的时候
- 3. 进程由于外部中断被打断进入内核态，内核发现某些条件满足（比如当前进程时间片用尽），进行进程切换

在我们实现的ucore中，内核进程是**不可抢占**的。这也就意味着当内核执行的时候，另一个内核进程不可以夺走它的CPU资源。但这是不是就意味着一个内核进程执行的时候，它就会一直执行到结束呢？虽说内核进程不可以抢占，但是它可以主动放弃自己占有的CPU资源。如果不这样设计的话，内核当中很有可能出现各种死锁导致内核崩溃。

另一方面，内核不能相信用户进程不会无限执行下去，所以需要提手段在用户进程执行的时候打断他，比如时钟中断。在ucore的实现中，用户进程可以随时被打断进入内核，操作系统会检查当前进程是否需要调度，从而把运行的机会交给别的进程。

由于内核进程是不可抢占的，所以我们在内核中有许多地方使用了显式的函数调用来进行调度，主要有以下几个地方：

函数	原因
<code>proc.c/do_exit</code>	用户进程退出，放弃CPU资源
<code>proc.c/do_wait</code>	用户进程等待，放弃CPU资源
<code>proc.c/init_main</code>	<code>init</code> 线程会等待所有的用户线程执行完毕，之后调用 <code>kswapd</code> 内核线程回收内存资源
<code>proc.c/cpu_idle</code>	<code>idle</code> 线程等待处于就绪态的线程，如果有就调用 <code>schedule</code>
<code>sync.c/lock</code>	如果获取锁失败就进入等待

当用户进程A发生中断或系统调用之后，首先其中断帧会被保存，CPU进入内核态，执行中断处理函数。在执行完毕中断处理函数后，操作系统检查当前进程是否需要调度，如果需要，就把当前的进程状态保存，switch到另一个进程B中。注意在执行上面的操作的时候，进程A处于内核态，类似的，调度后我们到达的是进程B的内核态。进程B从系统调用中返回，继续执行。如果进程B在中断或系统调用中被调度，控制权可能转交给进程A的内核态，这样进程A从内核态返回后就可以继续执行之前的代码了。

调度算法框架

结构体

调度算法框架实现为一个结构体，其中保存了各个函数指针。通过实现这些函数指针即可实现各个调度算法。结构体的定义如下：

```
1  struct sched_class {
2      // 调度类的名字
3      const char *name;
4      // 初始化run queue
5      void (*init)(struct run_queue *rq);
6      // 把进程放进run queue，这个是run queue的维护函数
7      void (*enqueue)(struct run_queue *rq, struct proc_struct *proc);
8      // 把进程取出run queue
9      void (*dequeue)(struct run_queue *rq, struct proc_struct *proc);
10     // 选择下一个要执行的进程
11     struct proc_struct *(*pick_next)(struct run_queue *rq);
12     // 每次时钟中断调用
13     void (*proc_tick)(struct run_queue *rq, struct proc_struct *proc);
14 };
```

所有的进程被组织成一个 `run_queue` 数据结构。这个数据结构虽然没有保存在调度类中，但是是由调度类来管理的。目前ucore仅支持单个CPU核心，所以只有一个全局的 `run_queue`。

我们在进程控制块中也记录了一些和调度有关的信息：

```
1  struct proc_struct {
2      // ...
3      // 表示这个进程是否需要调度
4      volatile bool need_resched;
5      // run queue的指针
6      struct run_queue *rq;
7      // 与这个进程相关的run queue表项
8      list_entry_t run_link;
9      // 这个进程剩下的时间片
10     int time_slice;
11     // 以下几个都和Stride调度算法实现有关
```

```

12     // 这个进程在优先队列中对应的项
13     skew_heap_entry_t lab6_run_pool;
14     // 该进程的Stride值
15     uint32_t lab6_stride;
16     // 该进程的优先级
17     uint32_t lab6_priority;
18 };

```

前面的几个成员变量的含义都比较直接，最后面的几个的含义可以参见Stride调度算法。这也是本次lab的实验内容。

结构体 `run_queue` 实现了运行队列，其内部结构如下：

```

1  struct run_queue {
2      // 保存着链表头指针
3      list_entry_t run_list;
4      // 运行队列中的线程数
5      unsigned int proc_num;
6      // 最大的时间片大小
7      int max_time_slice;
8      // Stride调度算法中的优先队列
9      skew_heap_entry_t *lab6_run_pool;
10 };

```

RR算法

有了这些基础，我们就来实现一个最简单的调度算法：Round-Robin调度算法，也叫时间片轮转调度算法。

时间片轮转调度算法非常简单。它为每一个进程维护了一个最大运行时间片。当一个进程运行够了其最大运行时间片那么长的时间后，调度器会把它标记为需要调度，并且把它的进程控制块放在队尾，重置其时间片。这种调度算法保证了公平性，每个进程都有均等的机会使用CPU，但是没有区分不同进程的优先级（这个也就是在Stride算法中需要考虑的问题）。下面我们来实现以下时间片轮转算法相对应的调度器借口吧！

首先是 `enqueue` 操作。RR算法直接把需要入队的进程放在调度队列的尾端，并且如果这个进程的剩余时间片为0（刚刚用完时间片被收回CPU），则需要把它的剩余时间片设为最大时间片。具体的实现如下：

```
1 static void
2 RR_enqueue(struct run_queue *rq, struct proc_struct *proc) {
3     assert(list_empty(&(proc->run_link)));
4     list_add_before(&(rq->run_list), &(proc->run_link));
5     if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
6         proc->time_slice = rq->max_time_slice;
7     }
8     proc->rq = rq;
9     rq->proc_num ++;
10 }
```

`dequeue` 操作非常普通，将相应的项从队列中删除即可：

```
1 static void
2 RR_dequeue(struct run_queue *rq, struct proc_struct *proc) {
3     assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
4     list_del_init(&(proc->run_link));
5     rq->proc_num --;
6 }
```

`pick_next` 选取队列头的表项，用 `le2proc` 函数获得对应的进程控制块，返回：

```
1 static struct proc_struct *
2 RR_pick_next(struct run_queue *rq) {
3     list_entry_t *le = list_next(&(rq->run_list));
4     if (le != &(rq->run_list)) {
5         return le2proc(le, run_link);
6     }
7     return NULL;
8 }
```

`proc_tick` 函数在每一次时钟中断调用。在这里，我们需要对当前正在运行的进程的剩余时间片减一。如果在减一后，其剩余时间片为0，那么我们就把这个进程标记为“需要调度”，这样在中断处理完之后内核判断进程是否需要调度的时候就会把它进行调度：

```
1 static void
2 RR_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
3     if (proc->time_slice > 0) {
4         proc->time_slice --;
5     }
6     if (proc->time_slice == 0) {
7         proc->need_resched = 1;
8     }
9 }
```

至此我们就实现完了和时间片轮转算法相关的所有重要接口。类似于RR算法，我们也可以参照这个方法实现自己的调度算法。本次实验中需要同学们自己实现Stride调度算法。


Stride算法

考察 `round-robin` 调度器，在假设所有进程都充分使用了其拥有的 CPU 时间资源的情况下，所有进程得到的 CPU 时间应该是相等的。但是有时候我们希望调度器能够更智能地为每个进程分配合理的 CPU 资源。假设我们为不同的进程分配不同的优先级，则我们有可能希望每个进程得到的时间资源与他们的优先级成正比关系。Stride调度是基于这种想法的一个较为典型和简单的算法。除了简单易于实现以外，它还有如下的特点：

- 可控性：如我们之前所希望的，可以证明 Stride Scheduling对进程的调度次数正比于其优先级。
- 确定性：在不考虑计时器事件的情况下，整个调度机制都是可预知和重现的。该算法的基本思想可以考虑如下：
 1. 为每个runnable的进程设置一个当前状态stride，表示该进程当前的调度权。另外定义其对应的pass值，表示对应进程在调度后，stride 需要进行的累加值。
 2. 每次需要调度时，从当前 runnable 态的进程中选择 stride最小的进程调度。
 3. 对于获得调度的进程P，将对应的stride加上其对应的步长pass（只与进程的优先权有关系）。
 4. 在一段固定的时间之后，回到 2.步骤，重新调度当前stride最小的进程。

可以证明，如果令 $P.\text{pass} = \text{BigStride} / P.\text{priority}$ 其中 $P.\text{priority}$ 表示进程的优先权（大于1），而 BigStride 表示一个预先定义的大常数，则该调度方案为每个进程分配的时间将与其优先级成正比。证明过程我们在这里略去，有兴趣的同学可以在网上查找相关资料。将该调度器应用到 ucore 的调度器框架中来，则需要将调度器接口实现如下：

- `init` :
 - 初始化调度器类的信息（如果有的话）。
 - 初始化当前的运行队列为一个空的容器结构。（比如和RR调度算法一样，初始化为一个有序列表）
- `enqueue`
 - 初始化刚进入运行队列的进程 `proc` 的 `stride` 属性。
 - 将 `proc` 插入放入运行队列中去（注意：这里并不要求放置在队列头部）。
- `dequeue`
 - 从运行队列中删除相应的元素。
- `pick_next`
 - 扫描整个运行队列，返回其中 `stride` 值最小的对应进程。
 - 更新对应进程的 `stride` 值，即
$$\text{pass} = \text{BIG_STRIDE} / P \rightarrow \text{priority}; P \rightarrow \text{stride} += \text{pass}。$$
- `proc_tick` :
 - 检测当前进程是否已用完分配的时间片。如果时间片用完，应该正确设置进程结构的相关标记来引起进程切换。
 - 一个 `process` 最多可以连续运行 `rq.max_time_slice` 个时间片。

 在具体实现时，有一个需要注意的地方：`stride` 属性的溢出问题，在之前的实现里面我们并没有考虑 `stride` 的数值范围，而这个值在理论上是不断增加的，在 `stride` 溢出以后，基于 `stride` 的比较可能会出现错误。比如假设当前存在两个进程 A 和 B，`stride` 属性采用 16 位无符号整数进行存储。当前队列中元素如下（假设当前运行的进程已经被重新放置进运行队列中）：

A.stride(实际值)	A.stride(理论值)	A.pass(=BigStride/A.priority)
65534	65534	100

B.stride(实际值)	B.stride(理论值)	B.pass(=BigStride/B.priority)
65535	65535	50

此时应该选择A作为调度的进程，而在一轮调度后，队列将如下：

A.stride(实际值)	A.stride(理论值)	A.pass(=BigStride/A.priority)
98	65634	100
B.stride(实际值)	B.stride(理论值)	B.pass(=BigStride/B.priority)
65535	65535	50

可以看到由于溢出的出现，进程间stride的理论比较和实际比较结果出现了偏差。我们首先在理论上分析这个问题：令 $PASS_MAX$ 为当前所有进程里最大的步进值。则我们可以证明如下结论：对每次Stride调度器的调度步骤中，有其最大的步进值 $STRIDE_MAX$ 和最小的步进值 $STRIDE_MIN$ 之差：

$$STRIDE_MAX - STRIDE_MIN \leq PASS_MAX$$

有了该结论，在加上之前对优先级有 $Priority > 1$ 限制，我们有

$STRIDE_MAX - STRIDE_MIN \leq BIG_STRIDE$ ，于是我们只要将BigStride取在某个范围之内，即可保证对于任意两个 Stride 之差都会在机器整数表示的范围之内。而我们可以通过其与0的比较结构，来得到两个Stride的大小关系。在上例中，虽然在直接的数值表示上 $98 < 65535$ ，但是 $98 - 65535$ 的结果用带符号的 16位整数表示的结果为99,与理论值之差相等。所以在这个意义下 $98 > 65535$ 。基于这种特殊考虑的比较方法，即便Stride有可能溢出，我们仍能够得到理论上的当前最小Stride，并做出正确的调度决定。

项目组成与执行流

项目组成

```
1 lab6
2 |— Makefile
3 |— kern
4 |   |— debug
5 |       |— assert.h
6 |       |— kdebug.c
7 |       |— kdebug.h
8 |       |— kmonitor.c
9 |       |— kmonitor.h
10 |      |— panic.c
11 |      |— stab.h
12 |   |— driver
13 |       |— clock.c
14 |       |— clock.h
15 |       |— console.c
16 |       |— console.h
17 |       |— ide.c
18 |       |— ide.h
19 |       |— intr.c
20 |       |— intr.h
21 |       |— kbdreg.h
22 |       |— picirq.c
23 |       |— picirq.h
24 |   |— fs
25 |       |— fs.h
26 |       |— swapfs.c
27 |       |— swapfs.h
28 |   |— init
29 |       |— entry.S
30 |       |— init.c
31 |   |— libs
32 |       |— readline.c
33 |       |— stdio.c
34 |   |— mm
35 |       |— default_pmm.c
36 |       |— default_pmm.h
37 |       |— kmalloc.c
38 |       |— kmalloc.h
39 |       |— memlayout.h
40 |       |— mmu.h
41 |       |— pmm.c
42 |       |— pmm.h
```

```
43 | | | swap.c
44 | | | swap.h
45 | | | swap_fifo.c
46 | | | swap_fifo.h
47 | | | vmm.c
48 | | | vmm.h
49 | | process
50 | | | entry.S
51 | | | proc.c
52 | | | proc.h
53 | | | switch.S
54 | | schedule
55 | | | default_sched.h
56 | | | default_sched_c
57 | | | default_sched_stride.c
58 | | | sched.c
59 | | | sched.h
60 | | sync
61 | | | sync.h
62 | | syscall
63 | | | syscall.c
64 | | | syscall.h
65 | | trap
66 | | | trap.c
67 | | | trap.h
68 | | | trapentry.S
69 | | libs
70 | | | atomic.h
71 | | | defs.h
72 | | | elf.h
73 | | | error.h
74 | | | hash.c
75 | | | list.h
76 | | | printfmt.c
77 | | | rand.c
78 | | | riscv.h
79 | | | sbi.h
80 | | | skew_heap.h
81 | | | stdarg.h
82 | | | stdio.h
83 | | | stdlib.h
84 | | | string.c
85 | | | string.h
86 | | | unistd.h
87 | | tools
88 | | | boot.ld
89 | | | function.mk
90 | | | gdbinit
91 | | | grade.sh
92 | | | kernel.ld
93 | | | sign.c
```

```
94 |   |— user.ld
95 |   |— vector.c
96 |— user
97 |   |— badarg.c
98 |   |— badsegment.c
99 |   |— divzero.c
100 |   |— exit.c
101 |   |— faultread.c
102 |   |— faultreadkernel.c
103 |   |— forktest.c
104 |   |— forktree.c
105 |   |— hello.c
106 |   |— libs
107 |       |— initcode.S
108 |       |— panic.c
109 |       |— stdio.c
110 |       |— syscall.c
111 |       |— syscall.h
112 |       |— ulib.c
113 |       |— ulib.h
114 |       |— umain.c
115 |   |— matrix.c
116 |   |— pgdir.c
117 |   |— priority.c
118 |   |— softint.c
119 |   |— spin.c
120 |   |— testbss.c
121 |   |— waitkill.c
122 |   |— yield.c
123
124 16 directories, 105 files
```

优先队列

`libs/skew_heap.h` : 提供了基本的优先队列数据结构，为本次实验提供了抽象数据结构方面的支持。

调度器框架

`kern/schedule/sched.[ch]` : 定义了 ucore 的调度器框架，其中包括相关的数据结构（包括调度器的接口和运行队列的结构），和具体的运行时机制。

RR算法

`kern/schedule/default_sched.[ch]` : 具体的 round-robin 算法，在本次实验中你需要了解其实现。

Stride算法

`kern/schedule/default_sched_stride.c` : Stride Scheduling调度器的基本框架，在此次实验中你需要填充其中的空白部分以实现一个完整的 Stride 调度器。

执行流

结合前面所述自行理解、总结。

LAB7：同步互斥

在之前的几章中我们已经实现了进程以及调度算法，可以让多个进程并发的执行。在现实的系统当中，有许多多线程的系统都需要协同的完成某一项任务。但是在协同的过程中，存在许多资源共享的问题，比如对一个文件读写的并发访问等等。这些问题需要我们提供一些同步互斥的机制来让程序可以有序的、无冲突的完成他们的工作，这也是这一章内我们要解决的问题。

我们最终实现的目标是解决“哲学家就餐问题”。“哲学家就餐问题”是一个非常有名的同步互斥问题：有五个哲学家围成一圈吃饭，每两个哲学家中间有一根筷子。每个需要就餐的哲学家需要两根筷子才可以就餐。哲学家处于两种状态之间：思考和饥饿。当哲学家处于思考的状态时，哲学家便无欲无求；而当哲学家处于饥饿状态时，他必须通过就餐来解决饥饿，重新回到思考的状态。如何让这5个哲学家可以不发生死锁的把这一顿饭吃完就是我们要解决的目标。

实验目的

1. 理解操作系统的同步互斥的设计实现
2. 掌握在ucore中信号量机制的具体实现
3. 理解管程机制，在ucore中增加基于管程的条件变量的支持
4. 了解经典进程同步问题，并能使用同步机制解决进程同步问题

实验内容

1. 阅读教材与实验指导书。
2. 分析信号量与条件变量的实现。
3. 设计为用户态进程提供信号量机制与条件变量机制的方案。

练习

练习1：说明不会出现死锁的原因

1. 证明/说明为什么我们给出的信号量实现的哲学家问题不会出现死锁。不必特别严谨，但要能说服你自己/助教？
 2. 证明/说明为什么我们给出的条件变量实现的哲学家问题不会出现死锁。不必特别严谨，但要能说服你自己/助教？
-

练习2：设计方案

1. 给出为用户态进程/线程提供信号量机制的设计方案，并比较说明给内核级提供信号量机制的异同。
 2. 给出为用户态进程/线程提供条件变量机制的设计方案，并比较说明给内核级提供条件变量机制的异同。
-

练习3：信号量实现条件变量

能否不基于信号量机制来完成条件变量？如果不能，请给出理由，如果能，请给出设计说明和具体实现。

练习4：禁用中断

kern/sync/sem.c的信号量实现中，出现了这样的暂时禁用中断的代码：

```
1 static __noinline uint32_t __down(semaphore_t *sem, uint32_t wait_state) {
2     bool intr_flag;
3     local_intr_save(intr_flag);
4     /*some code*/
5     local_intr_restore(intr_flag);
```

我们其实从正确的框架里删除了两句 `local_intr_restore(intr_flag)` 和一句 `local_intr_save(intr_flag);`，请你找出是哪里删掉了，并举出删掉之后不能正常工作的一种情况。

同步互斥的基本概念

虽然我们经常把同步和互斥放在一起说，但是这两个词是两个概念。**同步**指的是进程间的执行需要按照某种先后顺序，即访问是有序的。**互斥**指的是对于某些共享资源的访问不能同时进行，同一时间只能有一定数量的进程进行。这两种情况基本构成了我们在多线程执行中遇到的各种问题。

与同步互斥相关的另一个概念是**临界区**。临界区指的是进程的一段代码，其特征要求了同一时间段只能有一个进程执行，否则就有可能出现问题。一般而言，进程处理临界区的思路是设计一个协议，不同的进程遵守这个相同的协议来进行临界区的协调。在进入临界区前，进程请求进入的许可，这段代码称为**进入区**；退出临界区时，进程应该通过协议告知别的进程自己已经使用完临界区，这段代码称为**退出区**；临界区其他的部分称为**剩余区**。我们本章解决的问题，就是在进入区和退出区为进程提供同步互斥的机制。

为了提供同步互斥机制，操作系统有多种实现方法，包括时钟中断管理，屏蔽使能中断，等待队列，信号量，管程等等。下面我们来分别看一看上面提到的部分机制：

- 时钟中断管理：在第一个lab中我们已经实现了时钟中断。通过时钟中断，操作系统可以提供基于时间节点的事件。通过时钟中断，操作系统可以提供任意长度的等待唤醒机制，由此可以给应用程序机会来实现更加复杂的自定义调度操作。
- 屏蔽使能中断：这部分主要是处理内核内的同步互斥问题。因为内核在执行的过程中可能会被打断，我们实现的ucore也是不可抢占的系统，所以可以在操作系统进行某些需要同步互斥的操作的时候先禁用中断，等执行完之后再使能。这样保证了操作系统在执行临界区的时候不会被打断，也就实现了同步互斥。在ucore中经常可以看到下面这样的代码：

```
1  .....
2  local_intr_save(intr_flag);
3  {
4      临界区代码
5  }
6  local_intr_restore(intr_flag);
7  .....
```

这段代码就是使用屏蔽使能中断处理内核同步互斥问题的例子。

- 等待队列：等待队列是操作系统提供的一种事件机制。一些进程可能会在执行的过程中等待某些特定事件的发生，这个时候进程进入睡眠状态。操作系统维护一个等待队列，把这个进程放进他等待的事件的等待队列中。当对应的事件发生之后，操作系统就唤醒相应等待队列中的进程。这也是ucore内部实现信号量的机制。

信号量

介绍

信号量（semaphore）是一种同步互斥的实现。semaphore一词来源于荷兰语，原来是指火车信号灯。想象一些火车要进站，火车站里有n个站台，那么同一时间只能有n辆火车进站装卸货物。当火车站里已经有了n辆火车，信号灯应该通知后面的火车不能进站了。当有火车出站之后，信号灯应该告诉后面的火车可以进站。

这个问题放在操作系统的语境下就是有一个共享资源只能支持n个线程并行的访问，信号量统计目前有多少进程正在访问，当同时访问的进程数小于n时就可以让新的线程进入，当同时访问的进程数为n时想要访问的进程就需要等待。

在信号量中，一般用两种操作来刻画申请资源和释放资源：P操作申请一份资源，如果申请不到则等待；V操作释放一份资源，如果此时有进程正在等待，则唤醒该进程。在ucore中，我们使用 `down` 函数实现P操作，`up` 函数实现V操作。

结构体

首先是信号量结构体的定义：

```
1 typedef struct {
2     int value;
3     wait_queue_t wait_queue;
4 } semaphore_t;
```

其中的 `value` 表示信号量的值，其正值表示当前可用的资源数量，负值表示正在等待资源的进程数量。`wait_queue` 即为这个信号量相对应的等待队列。

P操作

`down` 函数实现的是P操作。首先关闭中断，然后判断信号量的值是否为正，如果是正值说明进程可以获得信号量，将信号量的值减一，打开中断然后函数返回即可。否则表示无法获取信号量，将自己的进程保存进等待队列，打开中断，调用 `schedule` 函数进行调度。等到V操作唤醒进程的时候，其会回到调用 `schedule` 函数后面，将自身从等待队列中删除（此过程需要关闭中断）并返回即可。具体的实现如下：

```
1 static __noinline uint32_t __down(semaphore_t *sem, uint32_t wait_state) {
2     bool intr_flag;
3     local_intr_save(intr_flag);
4     if (sem->value > 0) {
5         sem->value --;
6         local_intr_restore(intr_flag);
7         return 0;
8     }
9     wait_t __wait, *wait = &__wait;
10    wait_current_set(&(sem->wait_queue), wait, wait_state);
11    local_intr_restore(intr_flag);
12
13    schedule();
14
15    local_intr_save(intr_flag);
16    wait_current_del(&(sem->wait_queue), wait);
17    local_intr_restore(intr_flag);
18
19    if (wait->wakeup_flags != wait_state) {
20        return wait->wakeup_flags;
21    }
22    return 0;
23 }
```

V操作

`up` 函数实现了V操作。首先关闭中断，如果释放的信号量没有进程正在等待，那么将信号量的值加一，打开中断直接返回即可。如果有进程正在等待，那么唤醒这个进程，把它放进就绪队列，把信号量的值加一，打开中断并返回。具体的实现如下：

```
1 static __noinline void __up(semaphore_t *sem, uint32_t wait_state) {
2     bool intr_flag;
```

```
3     local_intr_save(intr_flag);
4     {
5         wait_t *wait;
6         if ((wait = wait_queue_first(&(sem->wait_queue))) == NULL) {
7             sem->value ++;
8         }
9         else {
10            assert(wait->proc->wait_state == wait_state);
11            wakeup_wait(&(sem->wait_queue), wait, wait_state, 1);
12        }
13    }
14    local_intr_restore(intr_flag);
15 }
```


条件变量与管程

介绍

我们已经有了基本的同步互斥机制实现，下面让我们用这些机制来实现一个管程，从而解决哲学家就餐问题。

为什么要使用管程呢？引入管程相当于将底层的同步互斥机制封装了起来，对外提供已经经过同步的接口供进程使用，大大降低了并行进程开发的门槛。管程主要由四个部分组成：

- 管程内部的共享变量
- 管程内部的条件变量
- 管程内部并发执行的进程
- 对局部于管程内部的共享数据设置初始值的语句

由此可见，管程把需要互斥访问的变量直接包装了起来，对共享变量的访问只能通过管程提供的相应接口，方便了多进程的编写。但是管程只有同步互斥是不够的，可能需要条件变量。条件变量类似于信号量，只不过在信号量中进程等待某一个资源可用，而条件变量中进程等待条件变量相应的资源为真。

结构体

条件变量的结构体如下：

```
1  typedef struct condvar{
2      // 信号量
3      semaphore_t sem;
4      // 正在等待的线程数
5      int count;
6      // 自己属于哪一个管程
7      monitor_t * owner;
8  } condvar_t;
```

等待唤醒

我们主要需要实现两个函数：`wait` 函数，等待某一个条件；`signal` 函数，提醒某一个条件已经达成。具体实现比较简单，可以参考代码如下：

```
1 // wait
2 cv.count++;
3 if(monitor.next_count > 0)
4     sem_signal(monitor.next);
5 else
6     sem_signal(monitor.mutex);
7 sem_wait(cv.sem);
8 cv.count -- ;
```

```
1 // signal
2 if( cv.count > 0) {
3     monitor.next_count ++;
4     sem_signal(cv.sem);
5     sem_wait(monitor.next);
6     monitor.next_count -- ;
7 }
```

实现

管程的内部实现如下所示：

```
1 typedef struct monitor{
2     // 保证管程互斥访问的信号量
3     semaphore_t mutex;
4     // 里面放着正在等待进入管程执行的进程
5     semaphore_t next;
6     // 正在等待进入管程的进程数
7     int next_count;
8     // 条件变量
9     condvar_t *cv;
10 } monitor_t;
```

条件变量 `cv` 被设置时，会使得当前在管程内的进程等待条件变量而睡眠，其他进程进入管程执行。当 `cv` 被唤醒的时候，之前等待这个条件变量的进程也会被唤醒，进入管程执行。由于管程内部只能由一个条件变量，所以通过设置 `next` 来维护下一个要运行的进程是哪一个。

使用了管程，我们的哲学家就餐问题可以被实现为如下：

```
1  monitor dp
2  {
3      enum {THINKING, HUNGRY, EATING} state[5];
4      condition self[5];
5
6      void pickup(int i) {
7          state[i] = HUNGRY;
8          test(i);
9          if (state[i] != EATING)
10             self[i].wait_cv();
11     }
12
13     void putdown(int i) {
14         state[i] = THINKING;
15         test((i + 4) % 5);
16         test((i + 1) % 5);
17     }
18
19     void test(int i) {
20         if ((state[(i + 4) % 5] != EATING) &&
21             (state[i] == HUNGRY) &&
22             (state[(i + 1) % 5] != EATING)) {
23             state[i] = EATING;
24             self[i].signal_cv();
25         }
26     }
27
28     initialization code() {
29         for (int i = 0; i < 5; i++)
30             state[i] = THINKING;
31     }
32 }
```

项目组成与执行流

项目组成

```
1  lab7
2  |— Makefile
3  |— kern
4  |   |— debug
5  |   |   |— assert.h
6  |   |   |— kdebug.c
7  |   |   |— kdebug.h
8  |   |   |— kmonitor.c
9  |   |   |— kmonitor.h
10 |   |   |— panic.c
11 |   |   |— stab.h
12 |   |— driver
13 |   |   |— clock.c
14 |   |   |— clock.h
15 |   |   |— console.c
16 |   |   |— console.h
17 |   |   |— ide.c
18 |   |   |— ide.h
19 |   |   |— intr.c
20 |   |   |— intr.h
21 |   |   |— kbdreg.h
22 |   |   |— picirq.c
23 |   |   |— picirq.h
24 |   |— fs
25 |   |   |— fs.h
26 |   |   |— swapfs.c
27 |   |   |— swapfs.h
28 |   |— init
29 |   |   |— entry.S
30 |   |   |— init.c
31 |   |— libs
32 |   |   |— readline.c
33 |   |   |— stdio.c
34 |   |— mm
35 |   |   |— default_pmm.c
36 |   |   |— default_pmm.h
37 |   |   |— kmalloc.c
38 |   |   |— kmalloc.h
39 |   |   |— memlayout.h
40 |   |   |— mmu.h
41 |   |   |— pmm.c
42 |   |   |— pmm.h
```

```

43 | | | |— swap.c
44 | | | |— swap.h
45 | | | |— swap_fifo.c
46 | | | |— swap_fifo.h
47 | | | |— vmm.c
48 | | | |— vmm.h
49 | | |— process
50 | | | |— entry.S
51 | | | |— proc.c
52 | | | |— proc.h
53 | | | |— switch.S
54 | | |— schedule
55 | | | |— default_sched.h
56 | | | |— default_sched_c
57 | | | |— default_sched_stride.c
58 | | | |— sched.c
59 | | | |— sched.h
60 | | |— sync
61 | | | |— check_sync.c
62 | | | |— monitor.c
63 | | | |— monitor.h
64 | | | |— sem.c
65 | | | |— sem.h
66 | | | |— sync.h
67 | | | |— wait.c
68 | | | |— wait.h
69 | | |— syscall
70 | | | |— syscall.c
71 | | | |— syscall.h
72 | | |— trap
73 | | | |— trap.c
74 | | | |— trap.h
75 | | | |— trapentry.S
76 |— lab5.md
77 |— libs
78 | |— atomic.h
79 | |— defs.h
80 | |— elf.h
81 | |— error.h
82 | |— hash.c
83 | |— list.h
84 | |— printfmt.c
85 | |— rand.c
86 | |— riscv.h
87 | |— sbi.h
88 | |— skew_heap.h
89 | |— stdarg.h
90 | |— stdio.h
91 | |— stdlib.h
92 | |— string.c
93 | |— string.h

```

```
94 |   └─ unistd.h
95 | └─ tools
96 |   └─ boot.ld
97 |   └─ function.mk
98 |   └─ gdbinit
99 |   └─ grade.sh
100 |   └─ kernel.ld
101 |   └─ sign.c
102 |   └─ user.ld
103 |   └─ vector.c
104 | └─ user
105 |   └─ badarg.c
106 |   └─ badsegment.c
107 |   └─ divzero.c
108 |   └─ exit.c
109 |   └─ faultread.c
110 |   └─ faultreadkernel.c
111 |   └─ forktest.c
112 |   └─ forktree.c
113 |   └─ hello.c
114 |   └─ libs
115 |     └─ initcode.S
116 |     └─ panic.c
117 |     └─ stdio.c
118 |     └─ syscall.c
119 |     └─ syscall.h
120 |     └─ ulib.c
121 |     └─ ulib.h
122 |     └─ umain.c
123 |   └─ matrix.c
124 |   └─ pgdir.c
125 |   └─ priority.c
126 |   └─ sleep.c
127 |   └─ sleepkill.c
128 |   └─ softint.c
129 |   └─ spin.c
130 |   └─ testbss.c
131 |   └─ waitkill.c
132 |   └─ yield.c
133 |
134 | 16 directories, 115 files
```

等待队列

`kern/sync/wait.[ch]` : 定义了等待队列 `wait_queue` 结构和等待entry的wait结构以及在此之上的函数，这是ucore中的信号量semaphore机制和条件变量机制的基础。

信号量

`kern/sync/sem.[ch]` : 定义并实现了ucore中内核级信号量相关的数据结构和函数。

条件变量与管程

`kern/sync/monitor.[ch]` : 基于管程的条件变量的实现程序。

执行流

结合前面所述自行理解、总结。

LAB8：文件系统

我们来到了最后一个lab。**文件系统(file system)**，指的是操作系统中管理（硬盘上）持久存储数据的模块。

实验目的

1. 了解基本的文件系统系统调用的实现方法；
2. 了解一个基于索引节点组织方式的Simple FS文件系统的设计与实现；
3. 了解文件系统抽象层-VFS的设计与实现；

实验内容

1. 通过分析了解ucore文件系统的总体架构设计，完善读写文件操作。
2. 重新实现基于文件系统的执行程序机制。
3. 完成执行存储在磁盘上的文件和实现文件读写等功能。

练习

练习1：填写sfs_io_nolock()函数

首先了解打开文件的处理流程，然后参考本实验后续的文件读写操作的过程分析，填写在 `kern/fs/sfs/sfs_inode.c` 中的 `sfs_io_nolock()` 函数，实现读文件中数据的代码。

练习2：填写load_icode()函数

改写 `proc.c` 中的 `load_icode` 函数和其他相关函数，实现基于文件系统的执行程序机制。执行：`make qemu`。如果能看看到 `sh` 用户程序的执行界面，则基本成功了。如果在 `sh` 用户界面上可以执行“`ls`”、“`hello`”等其他放置在 `sfs` 文件系统下的其他执行程序，则可以认为本实验基本成功。

练习3：管道 (Pipe)机制

如果要在 `ucore` 里加入 UNIX 的管道 (Pipe) 机制，至少需要定义哪些数据结构和接口？（接口给出语义即可，不必具体实现。数据结构的设计应当给出一个(或多个) 具体的 C 语言 `struct` 定义。你的设计应当体现出对可能出现的同步互斥问题的处理。

文件系统介绍

为什么需要文件/文件系统

能够"持久存储数据的设备",可能包括: 机械硬盘(HDD, hard disk drive), 固态硬盘(solid-state storage device), 光盘（加上它的驱动器），软盘，U盘，甚至磁带或纸带等等。一般来说，每个设备上都会连出很多针脚（这些针脚很可能符合某个特定协议，如USB协议，SATA协议），它们都可以按照事先约定的接口/协议把数据从设备中读到内存里，或者把内存里的数据按照一定的格式储存到设备里。

我们希望做到的第一件事情，就是把以上种种存储设备当作“同样的设备”进行使用。不管机械硬盘的扇区有多少个，或者一块固态硬盘里有多少存储单元，我们希望都能用同样的接口进行使用，提供“把硬盘的第a到第b个字节读出来”和“把内存里这些内容写到硬盘的从第a个字节开始的位置”的接口，在使用时只会感受到不同设备速度的不同。否则，我们还需要自己处理什么SATA协议，NVMe协议啥的。处理具体设备，具体协议并向上提供简单接口的软件，我们叫做**设备驱动(device driver)**，简称**驱动**。

文件的概念在我们脑子里根深蒂固，但“文件”其实也是一种抽象。理论上，只要提供了上面提到的读写两个接口，我们就可以进行编程，而并不需要什么文件的概念。只要你自己在小本本上记住，你的某些数据存储在硬盘上从某个地址开始的多么长的位置，以及硬盘上哪些位置是没有被占用的。编程的时候，如果用到/修改硬盘上的数据，就把小本本上记录的位置给硬编码到程序里。

但这很不灵活！如果你的小本本丢了，你就再也无法使用硬盘里的数据了，因为你不知道那些数据都是谁跟谁。另外，你也可能一不小心修改到无关的数据。最后，这个小本本经常需要修改，你很容易出错。

显然，我们应该把这个小本本交给计算机来保存和自动维护。这就是 **文件系统**。

我们把一段逻辑上相关联的数据看作一个整体，叫做**文件**。

除了硬盘，我们还可以把其他设备（如标准输入，标准输出）看成是文件，由文件系统进行统一的管理。之前我们模拟过一个很简单的文件系统，用来存放内存置换出的页面（参见 lab3 页面置换）。现在我们要来实现功能更强大也更复杂的文件系统。

虚拟文件系统

我们可以实现一个**虚拟文件系统 (virtual filesystem, VFS)**，作为操作系统和更具体的文件系统之间的接口。所谓“具体文件系统”，更接近具体设备和文件系统的内部实现，而“虚拟文件系统”更接近用户使用的接口。

电脑上本地的硬盘和远程的云盘(例如清华云盘，OneDrive) 的具体文件管理方式很可能不同，但是操作系统可以让我们用相同的操作访问这两个地方的文件，可以认为用到了虚拟文件系统。

例如，远程的云盘OneDrive，在Windows资源管理器(可以看作是虚拟文件系统提供给用户的接口)里，看起来和本地磁盘上的文件夹一模一样，也可以做和本地的文件夹相同的操作，复制，粘贴，打开。虽然背后有着网络传输，可能速度会慢，但是接口一致。（理论上可以在OneDrive的远程存储服务器上使用linux操作系统）。

在Linux系统中，如 `/floppy` 是一块MS-DOS文件系统的软盘的挂载点，`/tmp/test` 是Ext2文件系统的一个目录，我们执行 `cp /floppy/TEST /tmp/test`，进行目录的拷贝，相当于执行下面这段代码：

```
1  inf = open("/floppy/TEST", O_RDONLY, 0);
2  outf = open("/tmp/test", O_WRONLY|O_CREAT|O_TRUNC, 0600);
3  do {
4      i = read(inf, buf, 4096);
5      write(outf, buf, i);
6  } while (i);
7  close(outf);
8  close(inf);
```

对于不同文件系统的目录，我们可以使用相同的open, read, write, close接口，好像它们在一个文件系统里一样。这是虚拟系统的功能。

UNIX文件系统

ucore 的文件系统模型和传统的 UNIX文件系统类似。

UNIX 文件中的内容可理解为是一段有序的字节，占用磁盘上可能连续或不连续的一些空间（实际占用的空间可能比你存储的数据要多）。每个文件都有一个方便应用程序识别的文件名（也可以称作路径path），另外有一个文件系统内部使用的编号（用户程序不知道这个底层编号）。你可以对着一个文件又读又写（写的太多的时候可能会自动给文件分配更多的硬盘存储空间），也可以创建或删除文件。

目录(directory)是特殊的文件，一个目录里包含若干其他文件或目录。

在 UNIX 中，文件系统可以被安装在一个特定的文件路径位置，这个位置就是**挂载点 (mount point)**。所有的已安装文件系统都作为根文件系统树中的叶子出现在系统中。比如当你把U盘插进来，系统检测到U盘之后，会给U盘的文件系统一个挂载点，这个挂载点是原先的UNIX操作系统的叶子，但也可以认为是U盘文件系统的根节点。

UNIX的文件系统中，有一个**通用文件模型 (Common File Model)**，所有具体的文件系统（不管是Ext4, ZFS还是FAT），都需要提供通用文件模型所约定的行为。我们可以认为，通用文件模型是面向对象的，由若干对象(Object)组成，每个对象有成员属性和函数接口。类UNIX系统的内核一般使用C语言实现，“成员函数”一般体现为函数指针。

通用文件模型定义了一些对象：

- **超级块(superblock)**：存储整个文件系统的相关信息。对于磁盘上的文件系统，对应磁盘里的**文件系统控制块(filesystem control block)**
- **索引节点 (inode)**：存储关于某个文件的元数据信息（如访问控制权限、大小、拥有者、创建时间、数据内容等等），通常对应磁盘上的**文件控制块 (file control block)**。每个索引节点有一个编号，唯一确定文件系统里的一个文件。
- **文件(file)**：这里定义的file object不是指磁盘上的一个“文件”，而是指一个进程和它打开的一个文件之间的关系，这个对象存储在内核态的内存中，仅当某个进程打开某个文件的时候才存在。
- **目录项 (dentry)**：维护从“目录里的某一项”到“对应的文件”的链接/指针。一个目录也是一个文件，包含若干个子目录和其他文件。从某个子目录、文件的名称，对应到具体的文件/子目录的地址(或者索引节点inode)的链接，通过**目录项(dentry)**来描述。

上述抽象概念形成了 UNIX 文件系统的逻辑数据结构，并需要通过一个具体文件系统的架构，把上述信息映射并储存到磁盘介质上，从而在具体文件系统的磁盘布局（即数据在磁盘上的物理组织）上体现出上述抽象概念。比如文件元数据信息存储在磁盘块中的索引节点上。当文件被载入内存时，内核需要使用磁盘块中的索引点来构造内存中的索引节点。又比如dentry对象在磁盘上不存在，但是当目录包含的某一项（可能是子目录或文件）的信

息被载入到内存时，内核会构建对应的dentry对象，如 `/tmp/test` 这个路径，在解析的过程中，内核为根目录 `/` 创建一个dentry对象，为根目录的成员 `tmp` 构建一个dentry对象，为 `/tmp` 目录的成员 `test` 也构建一个dentry对象。

ucore 文件系统总体介绍

我们将在ucore里用虚拟文件系统管理三类设备：

- 硬盘，我们管理硬盘的具体文件系统是Simple File System（地位和Ext2等文件系统相同）
- 标准输出（控制台输出），只能写不能读
- 标准输入（键盘输入），只能读不能写

其中，标准输入和标准输出都是比较简单的设备。管理硬盘的Simple File System相对而言比较复杂。

我们的“硬盘”依然需要通过用一块内存来模拟。

lab8的Makefile和之前不同，我们分三段构建内核镜像。

- `sfs.img`: 一块符合SFS文件系统的硬盘，里面存储编译好的用户程序
- `swap.img`: 一段初始化为0的硬盘交换区
- `kernel objects`: ucore内核代码的目标文件

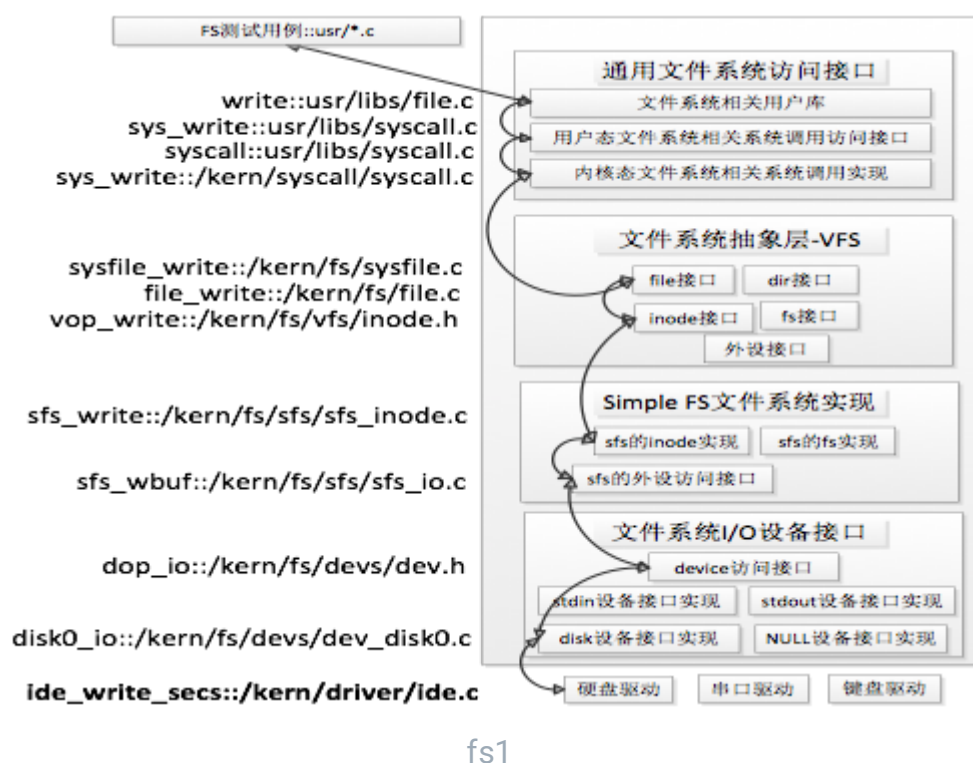
这三部分共同组成`ucore.img`, 加载到QEMU里运行。ucore代码中，我们通过链接时添加的首尾符号，把 `swap.img` 和 `sfs.img` 两段“硬盘”（实际上对应两段内存空间）找出来，然后作为“硬盘”进行管理。

注意，我们要在ucore内核开始执行之前，构造好“一块符合SFS文件系统的硬盘”，这就得另外写个程序做这个事情。这个程序就是 `tools/mksfs.c`。它有500多行，如果感兴趣的话可以通过它了解Simple File System的结构。

ucore 模仿了 UNIX 的文件系统设计，ucore 的文件系统架构主要由四部分组成：

- 通用文件系统访问接口层：该层提供了一个从用户空间到文件系统的标准访问接口。这一层访问接口让应用程序能够通过一个简单的接口获得 ucore 内核的文件系统服务。
- 文件系统抽象层：向上提供一个一致的接口给内核其他部分（文件系统相关的系统调用实现模块和其他内核功能模块）访问。向下提供一个同样的抽象函数指针列表和数据结构屏蔽不同文件系统的实现细节。
- Simple FS 文件系统层：一个基于索引方式的简单文件系统实例。向上通过各种具体函数实现以对应文件系统抽象层提出的抽象函数。向下访问外设接口
- 外设接口层：向上提供 device 访问接口屏蔽不同硬件细节。向下实现访问各种具体设备驱动接口，比如 disk 设备接口/串口设备接口/键盘设备接口等。

对照上面的层次我们再大致介绍一下文件系统的访问处理过程，加深对文件系统的总体理解。假如应用程序操作文件（打开/创建/删除/读写），首先需要通过文件系统的通用文件系统访问接口给用户空间提供的访问接口进入文件系统内部，接着由文件系统抽象层把访问请求转发给某一具体文件系统（比如 SFS 文件系统），具体文件系统（Simple FS 文件系统层）把应用程序的访问请求转化为对磁盘上的 block 的处理请求，并通过外设接口层交给磁盘驱动例程来完成具体的磁盘操作。结合用户态写文件函数 write 的整个执行过程，我们可以比较清楚地看出 ucore 文件系统架构的层次和依赖关系。

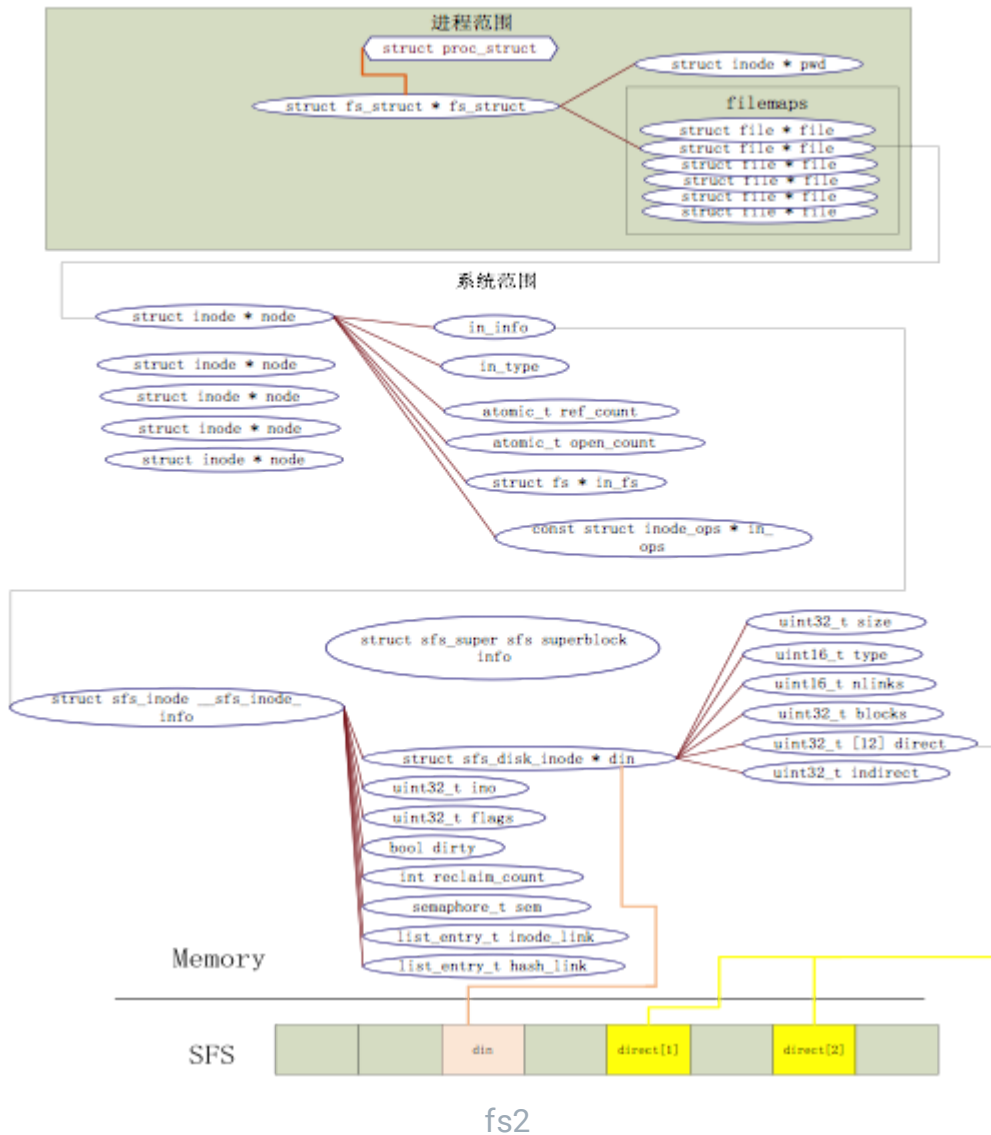


ucore 文件系统总体结构

从 ucore 操作系统不同的角度来看，ucore 中的文件系统架构包含四类主要的数据结构, 它们分别是：

- 超级块（SuperBlock），它主要从文件系统的全局角度描述特定文件系统的全局信息。它的作用范围是整个 OS 空间。
- 索引节点（inode）：它主要从文件系统的单个文件的角度它描述了文件的各种属性和数据所在位置。它的作用范围是整个 OS 空间。
- 目录项（dentry）：它主要从文件的文件路径的角度描述了文件路径中的一个特定的目录项（注：一系列目录项形成目录/文件路径）。它的作用范围是整个 OS 空间。对于 SFS 而言，inode(具体为 struct sfs_disk_inode)对应于物理磁盘上的具体对象，dentry（具体为 struct sfs_disk_entry）是一个内存实体，其中的 ino 成员指向对应的 inode number，另外一个成员是 file name(文件名)。
- 文件（file），它主要从进程的角度描述了一个进程在访问文件时需要了解的文件标识，文件读写的位置，文件引用情况等信息。它的作用范围是某一具体进程。

如果一个用户进程打开了一个文件，那么在 ucore 中涉及的相关数据结构和关系如下图所示：



文件系统抽象层VFS

文件系统抽象层是把不同文件系统的对外共性接口提取出来，形成一个函数指针数组，这样，通用文件系统访问接口层只需访问文件系统抽象层，而不需关心具体文件系统的实现细节和接口。

file&dir接口

file&dir 接口层定义了进程在内核中直接访问的文件相关信息，这定义在 file 数据结构中，具体描述如下：

```
1 // kern/fs/file.h
2 struct file {
3     enum {
4         FD_NONE, FD_INIT, FD_OPENED, FD_CLOSED,
5     } status;           //访问文件的执行状态
6     bool readable;       //文件是否可读
7     bool writable;       //文件是否可写
8     int fd;              //文件在filemap中的索引值
9     off_t pos;           //访问文件的当前位置
10    struct inode *node;   //该文件对应的内存inode指针
11    int open_count;       //打开此文件的次数
12 };
```

而在 kern/process/proc.h 中的 proc_struct 结构中加入了描述了进程访问文件的数据接口 files_struct，其数据结构定义如下：

```
1 // kern/fs/fs.h
2 struct files_struct {
3     struct inode *pwd;   //进程当前执行目录的内存inode指针
4     struct file *fd_array; //进程打开文件的数组
5     atomic_t files_count; //访问此文件的线程个数
6     semaphore_t files_sem; //确保对进程控制块中fs_struct的互斥访问
7 };
```

当创建一个进程后，该进程的 `files_struct` 将会被初始化或复制父进程的 `files_struct`。当用户进程打开一个文件时，将从 `fd_array` 数组中取得一个空闲 `file` 项，然后会把此 `file` 的成员变量 `node` 指针指向一个代表此文件的 `inode` 的起始地址。

inode接口

`index node` 是位于内存的索引节点，它是 VFS 结构中的重要数据结构，因为它实际负责把不同文件系统的特定索引节点信息（甚至不能算是一个索引节点）统一封装起来，避免了进程直接访问具体文件系统。其定义如下：

```
1 // kern/vfs/inode.h
2 struct inode {
3     union {                                //包含不同文件系统特定inode信息的
4         struct device __device_info;        //设备文件系统内存inode信息
5         struct sfs_inode __sfs_inode_info;  //SFS文件系统内存inode信息
6     } in_info;
7     enum {
8         inode_type_device_info = 0x1234,
9         inode_type_sfs_inode_info,
10    } in_type;                             //此inode所属文件系统类型
11    atomic_t ref_count;                    //此inode的引用计数
12    atomic_t open_count;                  //打开此inode对应文件的个数
13    struct fs *in_fs;                     //抽象的文件系统，包含访问文件系统的函数
14    const struct inode_ops *in_ops;       //抽象的inode操作，包含访问inode的函数
15 };
```

在 `inode` 中，有一成员变量为 `in_ops`，这是对此 `inode` 的操作函数指针列表，其数据结构定义如下：

```
1 struct inode_ops {
2     unsigned long vop_magic;
3     int (*vop_open)(struct inode *node, uint32_t open_flags);
4     int (*vop_close)(struct inode *node);
5     int (*vop_read)(struct inode *node, struct iobuf *iob);
6     int (*vop_write)(struct inode *node, struct iobuf *iob);
7     int (*vop_getdirent)(struct inode *node, struct iobuf *iob);
8     int (*vop_create)(struct inode *node, const char *name, bool excl, struct
9     int (*vop_lookup)(struct inode *node, char *path, struct inode **node_store);
```

```
10 .....
11 };
```

参照上面对 SFS 中的索引节点操作函数的说明，可以看出 `inode_ops` 是对常规文件、目录、设备文件所有操作的一个抽象函数表示。对于某一具体的文件系统上的文件或目录，只需实现相关的函数，就可以被用户进程访问具体的文件了，且用户进程无需了解具体文件系统的实现细节。

open系统调用的执行过程

下面我们通过打开文件的系统调用 `open()` 的执行过程, 看看文件系统的不同层次是如何交互的。

首先，经过 `syscall.c` 的处理之后，进入内核态，执行 `sysfile_open()` 函数

```
1 // kern/fs/sysfile.c
2 /* sysfile_open - open file */
3 int sysfile_open(const char *__path, uint32_t open_flags) {
4     int ret;
5     char *path;
6     if ((ret = copy_path(&path, __path)) != 0) {
7         return ret;
8     }
9     ret = file_open(path, open_flags);
10    kfree(path);
11    return ret;
12 }
```

可以看到，`sysfile_open` 把路径复制了一份，然后调用了 `file_open`，`file_open` 调用了 `vfs_open`，使用了VFS的接口。

```
1 // kern/fs/file.c
2 // open file
3 int file_open(char *path, uint32_t open_flags) {
4     bool readable = 0, writable = 0;
```

```

5     switch (open_flags & O_ACCMODE) { //解析 open_flags
6     case O_RDONLY: readable = 1; break;
7     case O_WRONLY: writable = 1; break;
8     case O_RDWR:
9         readable = writable = 1;
10        break;
11    default:
12        return -E_INVAL;
13    }
14    int ret;
15    struct file *file;
16    if ((ret = fd_array_alloc(NO_FD, &file)) != 0) { //在当前进程分配file de
17        return ret;
18    }
19    struct inode *node;
20    if ((ret = vfs_open(path, open_flags, &node)) != 0) { //打开文件的工作在
21        fd_array_free(file); //打开失败，释放file descriptor
22        return ret;
23    }
24    file->pos = 0;
25    if (open_flags & O_APPEND) {
26        struct stat __stat, *stat = &__stat;
27        if ((ret = vop_fstat(node, stat)) != 0) {
28            vfs_close(node);
29            fd_array_free(file);
30            return ret;
31        }
32        file->pos = stat->st_size; //追加写模式，设置当前位置为文件尾
33    }
34    file->node = node;
35    file->readable = readable;
36    file->writable = writable;
37    fd_array_open(file); //设置该文件的状态为“打开”
38    return file->fd;
39 }
40 // fs_array_alloc - allocate a free file item (with FD_NONE status) in ope
41 static int fd_array_alloc(int fd, struct file **file_store) {
42     struct file *file = get_fd_array();
43     if (fd == NO_FD) {
44         for (fd = 0; fd < FILES_STRUCT_NENTRY; fd ++, file ++) {
45             if (file->status == FD_NONE) {
46                 goto found;
47             }
48         }
49         return -E_MAX_OPEN;
50     }
51     else {
52         if (testfd(fd)) {
53             file += fd;
54             if (file->status == FD_NONE) {
55                 goto found;

```

```

56         }
57         return -E_BUSY;
58     }
59     return -E_INVAL;
60 }
61 found:
62     assert(fopen_count(file) == 0);
63     file->status = FD_INIT, file->node = NULL;
64     *file_store = file;
65     return 0;
66 }
67
68 void fd_array_open(struct file *file) {
69     assert(file->status == FD_INIT && file->node != NULL);
70     file->status = FD_OPENED; //设置状态为“打开”
71     fopen_count_inc(file); //增加文件的“打开计数”
72 }

```

`vfs_open` 是一个比较复杂的函数，这里我们使用的打开文件的flags, 基本是参照linux，如果希望详细了解，可以阅读[linux manual: open](#)。

```

1 // kern/fs/vfs/vfsfile.c
2
3 // open file in vfs, get/create inode for file with filename path.
4 int vfs_open(char *path, uint32_t open_flags, struct inode **node_store) {
5     bool can_write = 0;
6     // 解析open_flags并做合法性检查
7     switch (open_flags & O_ACCMODE) {
8     case O_RDONLY:
9         break;
10    case O_WRONLY:
11    case O_RDWR:
12        can_write = 1;
13        break;
14    default:
15        return -E_INVAL;
16    }
17
18    if (open_flags & O_TRUNC) {
19        if (!can_write) {
20            return -E_INVAL;
21        }
22    }
23    /*
24    linux manual
25    O_TRUNC

```

```

26         If the file already exists and is a regular file and the
27         access mode allows writing (i.e., is O_RDWR or O_WRONLY) it
28         will be truncated to length 0. If the file is a FIFO or ter
29         minal device file, the O_TRUNC flag is ignored. Otherwise,
30         the effect of O_TRUNC is unspecified.
31     */
32     int ret;
33     struct inode *node;
34     bool excl = (open_flags & O_EXCL) != 0;
35     bool create = (open_flags & O_CREAT) != 0;
36     ret = vfs_lookup(path, &node); // vfs_lookup根据路径构造inode
37
38     if (ret != 0) { //要打开的文件还不存在，可能出错，也可能需要创建新文件
39         if (ret == -16 && (create)) {
40             char *name;
41             struct inode *dir;
42             if ((ret = vfs_lookup_parent(path, &dir, &name)) != 0) {
43                 return ret; //需要在已经存在的目录下创建文件，目录不存在，则出错
44             }
45             ret = vop_create(dir, name, excl, &node); //创建新文件
46         } else return ret;
47     } else if (excl && create) {
48         return -E_EXISTS;
49     }
50     /*
51         linux manual
52         O_EXCL Ensure that this call creates the file: if this flag
53         specified in conjunction with O_CREAT, and pathname already
54         exists, then open() fails with the error EEXIST.
55     */
56     assert(node != NULL);
57
58     if ((ret = vop_open(node, open_flags)) != 0) {
59         vop_ref_dec(node);
60         return ret;
61     }
62
63     vop_open_inc(node);
64     if (open_flags & O_TRUNC || create) {
65         if ((ret = vop_truncate(node, 0)) != 0) {
66             vop_open_dec(node);
67             vop_ref_dec(node);
68             return ret;
69         }
70     }
71     *node_store = node;
72     return 0;
73 }

```


我们看看 `vfs_look_up` 的实现

```
1  /*
2   * get_device- Common code to pull the device name, if any, off the front
3   *               path and choose the inode to begin the name lookup relative
4   */
5
6  static int get_device(char *path, char **subpath, struct inode **node_store)
7  {
8      int i, slash = -1, colon = -1;
9      for (i = 0; path[i] != '\0'; i++) {
10         if (path[i] == ':') { colon = i; break; }
11         if (path[i] == '/') { slash = i; break; }
12     }
13     if (colon < 0 && slash != 0) {
14         /* *
15          * No colon before a slash, so no device name specified, and the slash
16          * or is also absent, so this is a relative path or just a bare filename
17          * the current directory, and use the whole thing as the subpath.
18          */
19         *subpath = path;
20         return vfs_get_curdir(node_store); //把当前目录的inode存到node_store
21     }
22     if (colon > 0) {
23         /* device:path - get root of device's filesystem */
24         path[colon] = '\0';
25
26         /* device:/path - skip slash, treat as device:path */
27         while (path[++ colon] == '/');
28         *subpath = path + colon;
29         return vfs_get_root(path, node_store);
30     }
31
32     /* *
33      * we have either /path or :path
34      * /path is a path relative to the root of the "boot filesystem"
35      * :path is a path relative to the root of the current filesystem
36      */
37     int ret;
38     if (*path == '/') {
39         if ((ret = vfs_get_bootfs(node_store)) != 0) {
40             return ret;
41         }
42     }
43     else {
44         assert(*path == ':');
45         struct inode *node;
46         if ((ret = vfs_get_curdir(&node)) != 0) {
```

```

47     }
48     /* The current directory may not be a device, so it must have a fs
49     assert(node->in_fs != NULL);
50     *node_store = fsop_get_root(node->in_fs);
51     vop_ref_dec(node);
52 }
53
54 /* ///... or :/... */
55 while (*(++ path) == '/');
56 *subpath = path;
57 return 0;
58 }
59
60 /*
61  * vfs_lookup - get the inode according to the path filename
62  */
63 int vfs_lookup(char *path, struct inode **node_store) {
64     int ret;
65     struct inode *node;
66     if ((ret = get_device(path, &path, &node)) != 0) {
67         return ret;
68     }
69     if (*path != '\\0') {
70         ret = vop_lookup(node, path, node_store);
71         vop_ref_dec(node);
72         return ret;
73     }
74     *node_store = node;
75     return 0;
76 }
77
78 /*
79  * vfs_lookup_parent - Name-to-vnode translation.
80  * (In BSD, both of these are subsumed by namei().)
81  */
82 int vfs_lookup_parent(char *path, struct inode **node_store, char **endp){
83     int ret;
84     struct inode *node;
85     if ((ret = get_device(path, &path, &node)) != 0) {
86         return ret;
87     }
88     *endp = path;
89     *node_store = node;
90     return 0;
91 }

```

我们注意到，这个流程中，有大量以 `vop` 开头的函数，它们都通过一些宏和函数的转发，最后变成对 `inode` 结构体里的 `inode_ops` 结构体的“成员函数”（实际上是函数指针）的调

用。对于SFS文件系统的 `inode` 来说，会变成对sfs文件系统的具体操作。sfs的这些具体接口的实现较为繁琐，可以在 `kern/fs/sfs/sfs_inode.c` 具体查看。我们的练习要求在 `kern/fs/sfs/sfs_io.c` 填写一个函数。

```
1 // kern/fs/sfs/sfs_inode.c
2
3 // The sfs specific DIR operations correspond to the abstract operations o
4 static const struct inode_ops sfs_node_dirops = {
5     .vop_magic                = VOP_MAGIC,
6     .vop_open                 = sfs_opendir,
7     .vop_close                = sfs_close,
8     .vop_fstat                = sfs_fstat,
9     .vop_fsync                = sfs_fsync,
10    .vop_namefile              = sfs_namefile,
11    .vop_getdirententry        = sfs_getdirententry,
12    .vop_reclaim               = sfs_reclaim,
13    .vop_gettype               = sfs_gettype,
14    .vop_lookup                = sfs_lookup,
15 };
16 /// The sfs specific FILE operations correspond to the abstract operations
17 static const struct inode_ops sfs_node_fileops = {
18     .vop_magic                = VOP_MAGIC,
19     .vop_open                 = sfs_openfile,
20     .vop_close                = sfs_close,
21     .vop_read                 = sfs_read,
22     .vop_write                = sfs_write,
23     .vop_fstat                = sfs_fstat,
24     .vop_fsync                = sfs_fsync,
25     .vop_reclaim               = sfs_reclaim,
26     .vop_gettype               = sfs_gettype,
27     .vop_tryseek              = sfs_tryseek,
28     .vop_truncate             = sfs_truncfile,
29 };
```

硬盘文件系统SFS

介绍

通常文件系统中，磁盘的使用是以扇区（Sector）为单位的，但是为了实现简便，SFS 中以 block（4K，与内存 page 大小相等）为基本单位。

SFS 文件系统的布局如下表所示。

superblock	root-dir inode	freemap	inode、File Data、Dir Data Blocks
超级块	根目录索引节点	空闲块映射	目录和文件的数据和索引节点

第 0 个块（4K）是超级块（superblock），它包含了关于文件系统的所有关键参数，当计算机被启动或文件系统被首次接触时，超级块的内容就会被装入内存。其定义如下：

```
1 struct sfs_super {
2     uint32_t magic;                /* magic number, should be 0x2f8dbe2a */
3     uint32_t blocks;               /* # of blocks in fs */
4     uint32_t unused_blocks;        /* # of unused blocks */
5     char info[SFS_MAX_INFO_LEN + 1]; /* information for sfs */
6 };
```

可以看到，包含一个成员变量魔数 magic，其值为 0x2f8dbe2a，内核通过它来检查磁盘镜像是否是合法的 SFS img；成员变量 blocks 记录了 SFS 中所有 block 的数量，即 img 的大小；成员变量 unused_block 记录了 SFS 中还没有被使用的 block 的数量；成员变量 info 包含了字符串"simple file system"。

第 1 个块放了一个 root-dir 的 inode，用来记录根目录的相关信息。有关 inode 还将在后续部分介绍。这里只要理解 root-dir 是 SFS 文件系统的根结点，通过这个 root-dir 的 inode 信息就可以定位并查找到根目录下的所有文件信息。

从第 2 个块开始，根据 SFS 中所有块的数量，用 1 个 bit 来表示一个块的占用和未被占用的情况。这个区域称为 SFS 的 freemap 区域，这将占用若干个块空间。为了更好地记录和管

理 freemap 区域，专门提供了两个文件 kern/fs/sfs/bitmap.[ch]来完成根据一个块号查找或设置对应的 bit 位的值。

最后在剩余的磁盘空间中，存放了所有其他目录和文件的 inode 信息和内容数据信息。需要注意的是虽然 inode 的大小小于一个块的大小（4096B），但为了实现简单，每个 inode 都占用一个完整的 block。

在 sfs_fs.c 文件中的 sfs_do_mount 函数中，完成了加载位于硬盘上的 SFS 文件系统的超级块 superblock 和 freemap 的工作。这样，在内存中就有了 SFS 文件系统的全局信息。

"魔数"是怎样工作的?

我们经常需要检查某个文件/某块磁盘是否符合我们需要的格式。一般会按照这个文件的完整格式，进行一次全面的分析。

在一个较早的版本，UNIX的可执行文件格式最开头包含一条PDP-11平台上的跳转指令，使得在PDP-11硬件平台上能够正常运行，而在其他平台上，这条指令就是“魔数”（magic number），只能用作文件类型的标识。

Java类文件（编译到字节码）以十六进制0xCAFEBAE 开头

JPEG图片文件以0xFFD8开头，0xFFD9结尾

PDF文件以“%PDF”的ASCII码开头，十六进制25 50 44 46

进行这样的约定之后，我们发现，如果文件开头的“魔数”不符合要求，那么这个文件的格式一定不对。这让我们立刻发现文件损坏或者搞错文件类型的情况。由于不同类型的文件有不同的魔数，当你把JPEG文件当作PDF打开的时候，立即就会出现异常。

下面是一个摇滚乐队和巧克力豆的故事，有助于你理解魔数的作用。

美国著名重金属摇滚乐队Van Halen的演出合同中有此一条：演出后台必须提供M&M巧克力豆，但是绝对不许出现棕色豆。如有违反，根据合同，乐队可以取消演出。实际情形中乐队甚至会借此发飙，砸后台，主办方也只好承担所有经济损失。这一条款长期被媒体用来作为摇滚乐队耍大牌的典型例子，有传言指某次由于主唱在后台发现了棕色M&M豆，大发其飙地砸了后台，造成损失高达八万五千

美元（当时是八十年代，八万五千还是不少钱）。Van Halen乐队对此从不回应。

多年以后，主唱David Lee Roth 在自传中揭示了这一无厘头条款的来由：Van Halen 乐队在当时是把大型摇滚现场演唱会推向高校及二／三线地区的先锋，由于常常会遇到没有处理过这种大场面的承办者，因此合同里有大量条款来确认演出承办者把场地，器材，工作人员安排等等细节都严格按照要求准备好。合同里有成章成章的技术细节，包括场地的承重要求，各类出入口的宽度，电源要求，以至于插座的数量和插座之间的间隔。因此，乐队把棕色豆条款夹带在合同里，以确认承办方是否“仔仔细细阅读了所有条款”。David说：“如果我在后台的M&M里找到棕色豆，我就会立马知道承办方（十有八九）是没好好读完全部技术要求，我们肯定会碰上技术问题。某些技术问题绝对会毁了这场演出，甚至害死人。”

回到上文，八万五千美元的损失是怎么来的？某次在某大学体育场办演唱会，主唱来到后台，发现了棕色M&M豆，当即发飙，砸了后台化妆室，财物损坏大概值一万二。但实际上更糟糕的是，主办方没有细读演出演出场地的承重要求，结果整个舞台压垮（似乎是压穿）了体育场地面，损失高达八万多。

事后媒体的报道是，由于主唱看到棕色M&M豆后发飙砸了后台，造成高达八万五的损失...

索引节点

在 SFS 文件系统中，需要记录文件内容的存储位置以及文件名与文件内容的对应关系。sfs_disk_inode 记录了文件或目录的内容存储的索引信息，该数据结构在硬盘里储存，需要时读入内存（从磁盘读进来的是一段连续的字节，我们将这段连续的字节强制转换成 sfs_disk_inode 结构体；同样，写入的时候换一个方向强制转换）。sfs_disk_entry 表示一个目录中的一个文件或目录，包含该项所对应 inode 的位置和文件名，同样也在硬盘里储存，需要时读入内存。

磁盘索引节点

SFS 中的磁盘索引节点代表了一个实际位于磁盘上的文件。首先我们看看在硬盘上的索引节点的内容：

```

1 // kern/fs/sfs/sfs.hc
2 /*inode (on disk)*/
3 struct sfs_disk_inode {
4     uint32_t size; //如果inode表示常规文件，则size表示文件的大小
5     uint16_t type; //inode的文件类型
6     uint16_t nlinks; //此inode的硬链接数
7     uint32_t blocks; //此inode的数据块数的个数
8     uint32_t direct[SFS_NDIRECT]; //此inode的直接数据块索引值 (0表示无效索引)
9     uint32_t indirect; //此inode的一级间接数据块索引值
10 };

```

通过上表可以看出，如果 inode 表示的是文件，则成员变量 direct[] 直接指向了保存文件内容数据的数据块索引值。indirect 间接指向了保存文件内容数据的数据块，indirect 指向的是间接数据块（indirect block），此数据块实际存放的全部是数据块索引，这些数据块索引指向的数据块才被用来存放文件内容数据。

默认的，ucore 里 SFS_NDIRECT 是 12，即直接索引的数据页大小为 $12 \times 4k = 48k$ ；当使用一级间接数据块索引时，ucore 支持最大的文件大小为 $12 \times 4k + 1024 \times 4k = 48k + 4m$ 。数据索引表内，0 表示一个无效的索引，inode 里 blocks 表示该文件或者目录占用的磁盘的 block 的个数。indirect 为 0 时，表示不使用一级索引块。（因为 block 0 用来保存 super block，它不可能被其他任何文件或目录使用，所以这么设计也是合理的）。

对于普通文件，索引值指向的 block 中保存的是文件中的数据。而对于目录，索引值指向的数据保存的是目录下所有的文件名以及对应的索引节点所在的索引块（磁盘块）所形成的数组。数据结构如下：

```

1 // kern/fs/sfs/sfs.h
2 /* file entry (on disk) */
3 struct sfs_disk_entry {
4     uint32_t ino; //索引节点所占数据块索引值
5     char name[SFS_MAX_FNAME_LEN + 1]; //文件名
6 };

```

操作系统中，每个文件系统下的 inode 都应该分配唯一的 inode 编号。SFS 下，为了实现的简便（偷懒），每个 inode 直接用他所在的磁盘 block 的编号作为 inode 编号。比如，root block 的 inode 编号为 1；每个 sfs_disk_entry 数据结构中，name 表示目录下文件或文件夹

的名称，ino 表示磁盘 block 编号，通过读取该 block 的数据，能够得到相应的文件或文件夹的 inode。ino 为 0 时，表示一个无效的 entry。

此外，和 inode 相似，每个 `sfs_disk_entry` 也占用一个 block。

内存中的索引节点

```
1 // kern/fs/sfs/sfs.h
2 /* inode for sfs */
3 struct sfs_inode {
4     struct sfs_disk_inode *din;           /* on-disk inode */
5     uint32_t ino;                         /* inode number */
6     uint32_t flags;                       /* inode flags */
7     bool dirty;                           /* true if inode modified */
8     int reclaim_count;                    /* kill inode if it hits zero */
9     semaphore_t sem;                     /* semaphore for din */
10    list_entry_t inode_link;               /* entry for linked-list in */
11    list_entry_t hash_link;               /* entry for hash linked-li
12 };
```

可以看到 SFS 中的内存 inode 包含了 SFS 的硬盘 inode 信息，而且还增加了其他一些信息，这属于是便于进行判断是否改写、互斥操作、回收和快速地定位等作用。需要注意，一个内存 inode 是在打开一个文件后才创建的，如果关机则相关信息都会消失。而硬盘 inode 的内容是保存在硬盘中的，只是在进程需要时才被读入到内存中，用于访问文件或目录的具体内容数据

为了方便实现上面提到的多级数据的访问以及目录中 entry 的操作，对 inode SFS 实现了一些辅助的函数：

(在 `kern/fs/sfs/sfs_inode.c` 实现)

1. `sfs_bmap_load_nolock`：将对应 `sfs_inode` 的第 `index` 个索引指向的 block 的索引值取出存到相应的指针指向的单元 (`ino_store`)。该函数只接受 `index <= inode->blocks` 的参数。当 `index == inode->blocks` 时，该函数理解为需要为 inode 增长一个 block。并标记 inode 为 dirty（所有对 inode 数据的修改都要做这样的操作，这样，当 inode 不再使用的时候，sfs 能够保证 inode 数据能够被写回到磁盘）。`sfs_bmap_load_nolock` 调用的 `sfs_bmap_get_nolock` 来完成相应的操作，阅读 `sfs_bmap_get_nolock`，了解他是如何工作的。（`sfs_bmap_get_nolock` 只由 `sfs_bmap_load_nolock` 调用）

2. `sfs_bmap_truncate_nolock`：将多级数据索引表的最后一个 entry 释放掉。他可以认为是 `sfs_bmap_load_nolock` 中，`index == inode->blocks` 的逆操作。当一个文件或目录被删除时，sfs 会循环调用该函数直到 `inode->blocks` 减为 0，释放所有的数据页。函数通过 `sfs_bmap_free_nolock` 来实现，他应该是 `sfs_bmap_get_nolock` 的逆操作。和 `sfs_bmap_get_nolock` 一样，调用 `sfs_bmap_free_nolock` 也要格外小心。
3. `sfs_dirent_read_nolock`：将目录的第 slot 个 entry 读取到指定的内存空间。他通过上面提到的函数来完成。
4. `sfs_dirent_search_nolock`：是常用的查找函数。他在目录下查找 name，并且返回相应的搜索结果（文件或文件夹）的 inode 的编号（也是磁盘编号），和相应的 entry 在该目录的 index 编号以及目录下的数据页是否有空闲的 entry。（SFS 实现里文件的数据页是连续的，不存在任何空洞；而对于目录，数据页不是连续的，当某个 entry 删除的时候，SFS 通过设置 `entry->ino` 为 0 将该 entry 所在的 block 标记为 free，在需要添加新 entry 的时候，SFS 优先使用这些 free 的 entry，其次才会去在数据页尾追加新的 entry。

注意，这些后缀为 `nolock` 的函数，只能在已经获得相应 inode 的 semaphore 才能调用。

Inode 的文件操作函数

```
1 // kern/fs/sfs/sfs_inode.c
2 static const struct inode_ops sfs_node_fileops = {
3     .vop_magic           = VOP_MAGIC,
4     .vop_open            = sfs_openfile,
5     .vop_close           = sfs_close,
6     .vop_read            = sfs_read,
7     .vop_write           = sfs_write,
8     .....
9 };
```

上述 `sfs_openfile`、`sfs_close`、`sfs_read` 和 `sfs_write` 分别对应用户进程发出的 `open`、`close`、`read`、`write` 操作。其中 `sfs_openfile` 不用做什么事；`sfs_close` 需要把对文件的修改内容写回到硬盘上，这样确保硬盘上的文件内容数据是最新的；`sfs_read` 和 `sfs_write` 函数都调用了函数 `sfs_io`，并最终通过访问硬盘驱动来完成对文件内容数据的读写。

Inode 的目录操作函数

```

1 // kern/fs/sfs/sfs_inode.c
2 static const struct inode_ops sfs_node_dirops = {
3     .vop_magic          = VOP_MAGIC,
4     .vop_open           = sfs_opendir,
5     .vop_close          = sfs_close,
6     .vop_getdirent      = sfs_getdirent,
7     .vop_lookup         = sfs_lookup,
8     .....
9 };

```

对于目录操作而言，由于目录也是一种文件，所以 `sfs_opendir`、`sfs_close` 对应用户进程发出的 `open`、`close` 函数。相对于 `sfs_open`，`sfs_opendir` 只是完成一些 `open` 函数传递的参数判断，没做其他更多的事情。目录的 `close` 操作与文件的 `close` 操作完全一致。由于目录的内容数据与文件的内容数据不同，所以读出目录的内容数据的函数是 `sfs_getdirent()`，其主要工作是获取目录下的文件 `inode` 信息。

这里用到的 `inode_ops` 结构体，在 `kern/fs/vfs/inode.h` 定义，作用是：把关于 `inode` 的操作接口，集中在一个结构体里，通过这个结构体，我们可以把 Simple File System 的接口（如 `sfs_openfile()`）提供给上层的 VFS 使用。可以想象我们除了 Simple File System，还在另一块磁盘上使用完全不同的文件系统 Complex File System，显然 `vop_open()`、`vop_read()` 这些接口的实现都要不一样了。对于同一个文件系统这些接口都是一样的，所以我们可以提供“属于 SFS 的文件的 `inode_ops` 结构体”，“属于 CFS 的文件的 `inode_ops` 结构体”。

下面的注释里详细解释了每个接口的用途。当然，不必现在就详细了解每一个接口。

```

1 // kern/fs/vfs/inode.h
2 struct inode_ops {
3     unsigned long vop_magic;
4     int (*vop_open)(struct inode *node, uint32_t open_flags);
5     int (*vop_close)(struct inode *node);
6     int (*vop_read)(struct inode *node, struct iobuf *iob);
7     int (*vop_write)(struct inode *node, struct iobuf *iob);
8     int (*vop_fstat)(struct inode *node, struct stat *stat);
9     int (*vop_fsync)(struct inode *node);
10    int (*vop_namefile)(struct inode *node, struct iobuf *iob);
11    int (*vop_getdirent)(struct inode *node, struct iobuf *iob);
12    int (*vop_reclaim)(struct inode *node);
13    int (*vop_gettype)(struct inode *node, uint32_t *type_store);
14    int (*vop_tryseek)(struct inode *node, off_t pos);
15    int (*vop_truncate)(struct inode *node, off_t len);

```

```

16     int (*vop_create)(struct inode *node, const char *name, bool excl, str
17     int (*vop_lookup)(struct inode *node, char *path, struct inode **node_
18     int (*vop_ioctl)(struct inode *node, int op, void *data);
19 };
20
21 /*
22  * Abstract operations on a inode.
23  *
24  * These are used in the form VOP_FOO(inode, args), which are macros
25  * that expands to inode->inode_ops->vop_foo(inode, args). The operations
26  * "foo" are:
27  *
28  *     vop_open          - Called on open() of a file. Can be used to
29  *                         reject illegal or undesired open modes. Note that
30  *                         various operations can be performed without the
31  *                         file actually being opened.
32  *                         The inode need not look at O_CREAT, O_EXCL, or
33  *                         O_TRUNC, as these are handled in the VFS layer.
34  *
35  *                         VOP_EACHOPEN should not be called directly from
36  *                         above the VFS layer - use vfs_open() to open inode
37  *                         This maintains the open count so VOP_LASTCLOSE can
38  *                         be called at the right time.
39  *
40  *     vop_close          - To be called on *last* close() of a file.
41  *
42  *                         VOP_LASTCLOSE should not be called directly from
43  *                         above the VFS layer - use vfs_close() to close
44  *                         inodes opened with vfs_open().
45  *
46  *     vop_reclaim        - Called when inode is no longer in use. Note that
47  *                         this may be substantially after vop_lastclose is
48  *                         called.
49  *
50  * *****
51  *
52  *     vop_read           - Read data from file to uio, at offset specified
53  *                         in the uio, updating uio_resid to reflect the
54  *                         amount read, and updating uio_offset to match.
55  *                         Not allowed on directories or symlinks.
56  *
57  *     vop_getdirent      - Read a single filename from a directory into a
58  *                         uio, choosing what name based on the offset
59  *                         field in the uio, and updating that field.
60  *                         Unlike with I/O on regular files, the value of
61  *                         the offset field is not interpreted outside
62  *                         the filesystem and thus need not be a byte
63  *                         count. However, the uio_resid field should be
64  *                         handled in the normal fashion.
65  *                         On non-directory objects, return ENOTDIR.
66  *

```

```

67 *      vop_write      - Write data from uio to file at offset specified
68 *                      in the uio, updating uio_resid to reflect the
69 *                      amount written, and updating uio_offset to match.
70 *                      Not allowed on directories or symlinks.
71 *
72 *      vop_ioctl      - Perform ioctl operation OP on file using data
73 *                      DATA. The interpretation of the data is specific
74 *                      to each ioctl.
75 *
76 *      vop_fstat      -Return info about a file. The pointer is a
77 *                      pointer to struct stat; see stat.h.
78 *
79 *      vop_gettype     - Return type of file. The values for file types
80 *                      are in sfs.h.
81 *
82 *      vop_tryseek     - Check if seeking to the specified position within
83 *                      the file is legal. (For instance, all seeks
84 *                      are illegal on serial port devices, and seeks
85 *                      past EOF on files whose sizes are fixed may be
86 *                      as well.)
87 *
88 *      vop_fsync       - Force any dirty buffers associated with this file
89 *                      to stable storage.
90 *
91 *      vop_truncate    - Forcibly set size of file to the length passed
92 *                      in, discarding any excess blocks.
93 *
94 *      vop_namefile    - Compute pathname relative to filesystem root
95 *                      of the file and copy to the specified io buffer.
96 *                      Need not work on objects that are not
97 *                      directories.
98 *
99 *****
100 *
101 *      vop_creat       - Create a regular file named NAME in the passed
102 *                      directory DIR. If boolean EXCL is true, fail if
103 *                      the file already exists; otherwise, use the
104 *                      existing file if there is one. Hand back the
105 *                      inode for the file as per vop_lookup.
106 *
107 *****
108 *
109 *      vop_lookup      - Parse PATHNAME relative to the passed directory
110 *                      DIR, and hand back the inode for the file it
111 *                      refers to. May destroy PATHNAME. Should increment
112 *                      refcount on inode handed back.
113 */

```

设备即文件

在本实验中，为了统一地访问设备(device)，我们可以把一个设备看成一个文件，通过访问文件的接口来访问设备。目前实现了 stdin 设备文件、stdout 设备文件、disk0 设备。stdin 设备就是键盘，stdout 设备就是控制台终端的文本显示，而 disk0 设备是承载 SFS 文件系统的磁盘设备。下面看看 ucore 是如何让用户把设备看成文件来访问。

设备的定义

为了表示一个设备，需要有对应的数据结构，ucore 为此定义了 struct device，如下：

可以认为 struct device 是一个比较抽象的“设备”的定义。一个具体设备，只要实现了 d_open() 打开设备，d_close() 关闭设备，d_io() (读写该设备，write参数是true/false 决定是读还是写)，d_ioctl() (input/output control)四个函数接口，就可以被文件系统使用了。

```
1 // kern/fs/devs/dev.h
2 /*
3  * Filesystem-namespace-accessible device.
4  * d_io is for both reads and writes; the iobuf will indicates the direction
5  */
6 struct device {
7     size_t d_blocks;
8     size_t d_blocksize;
9     int (*d_open)(struct device *dev, uint32_t open_flags);
10    int (*d_close)(struct device *dev);
11    int (*d_io)(struct device *dev, struct iobuf *iob, bool write);
12    int (*d_ioctl)(struct device *dev, int op, void *data);
13 };
14
15 #define dop_open(dev, open_flags) ((dev)->d_open(dev, open_flags))
16 #define dop_close(dev) ((dev)->d_close(dev))
17 #define dop_io(dev, iob, write) ((dev)->d_io(dev, iob, write))
18 #define dop_ioctl(dev, op, data) ((dev)->d_ioctl(dev, op, data))
```

这个数据结构能够支持对块设备（比如磁盘）、字符设备（比如键盘）的表示，完成对设备的基本操作。

但这个设备描述没有与文件系统以及表示一个文件的 inode 数据结构建立关系，为此，还需要另外一个数据结构把 device 和 inode 联通起来，这就是 `vfs_dev_t` 数据结构。

利用 `vfs_dev_t` 数据结构，就可以让文件系统通过一个链接 `vfs_dev_t` 结构的双向链表找到 device 对应的 inode 数据结构，一个 inode 节点的成员变量 `in_type` 的值是 0x1234，则此 inode 的成员变量 `in_info` 将成为一个 device 结构。这样 inode 就和一个设备建立了联系，这个 inode 就是一个设备文件。

```
1 // kern/fs/vfs/vfsdev.c
2 // device info entry in vdev_list
3 typedef struct {
4     const char *devname;
5     struct inode *devnode;
6     struct fs *fs;
7     bool mountable;
8     list_entry_t vdev_link;
9 } vfs_dev_t;
10 #define le2vdev(le, member) \
11     to_struct((le), vfs_dev_t, member) //为了使用链表定义的宏，做到现在应该对它
12
13 static list_entry_t vdev_list; // device info list in vfs layer
14 static semaphore_t vdev_list_sem; // 互斥访问的semaphore
15 static void lock_vdev_list(void) {
16     down(&vdev_list_sem);
17 }
18 static void unlock_vdev_list(void) {
19     up(&vdev_list_sem);
20 }
```

ucore 虚拟文件系统为了把这些设备链接在一起，还定义了一个设备链表，即双向链表 `vdev_list`，这样通过访问此链表，可以找到 ucore 能够访问的所有设备文件。

注意这里的 `vdev_list` 对应一个 `vdev_list_sem`。在文件系统中，互斥访问非常重要，所以我们将看到很多的 `semaphore`。

我们使用 `iobuf` 结构体传递一个IO请求（要写入设备的数据当前所在内存的位置和长度/从设备读取的数据需要存储到的位置）

```

1 struct iobuf {
2     void *io_base;    // the base addr of buffer (used for Rd/Wr)
3     off_t io_offset;  // current Rd/Wr position in buffer, will have been
4     size_t io_len;    // the length of buffer (used for Rd/Wr)
5     size_t io_resid;  // current resident length need to Rd/Wr, will have
6 };

```

注意设备文件的inode也有一个 `inode_ops` 成员, 提供设备文件应具备的接口。

```

1 // kern/fs/devs/dev.c
2 /*
3  * Function table for device inodes.
4  */
5 static const struct inode_ops dev_node_ops = {
6     .vop_magic          = VOP_MAGIC,
7     .vop_open           = dev_open,
8     .vop_close          = dev_close,
9     .vop_read           = dev_read,
10    .vop_write           = dev_write,
11    .vop_fstat           = dev_fstat,
12    .vop_ioctl           = dev_ioctl,
13    .vop_gettype         = dev_gettype,
14    .vop_tryseek         = dev_tryseek,
15    .vop_lookup           = dev_lookup,
16 };

```

stdin设备

trap.c改变了对 `stdin` 的处理, 将 `stdin` 作为一个设备(也是一个文件), 通过 `sys_read()` 接口读取标准输入的数据。

注意, 既然我们把 `stdin`, `stdout` 看作文件, 那么也需要先打开文件, 才能进行读写。在执行用户程序之前, 我们先执行了 `umain.c` 建立一个运行时环境, 这里主要做的工作, 就是让程序能够使用 `stdin`, `stdout`。

```

1 // user/libs/file.c
2 //这是用户态程序可以使用的“系统库函数”，从文件fd读取len个字节到base这个位置。
3 //当fd = 0的时候，表示从stdin读取
4 int read(int fd, void *base, size_t len) {
5     return sys_read(fd, base, len);
6 }
7 // user/libs/umain.c
8 int main(int argc, char *argv[]);
9
10 static int initfd(int fd2, const char *path, uint32_t open_flags) {
11     int fd1, ret;
12     if ((fd1 = open(path, open_flags)) < 0) {
13         return fd1;
14     } //我们希望文件描述符是fd2，但是分配的fd1如果不等于fd2，就需要做一些处理
15     if (fd1 != fd2) {
16         close(fd1);
17         ret = dup2(fd1, fd2); //通过sys_dup让两个文件描述符指向同一个文件
18         close(fd1);
19     }
20     return ret;
21 }
22
23 void umain(int argc, char *argv[]) {
24     int fd;
25     if ((fd = initfd(0, "stdin:", O_RDONLY)) < 0) {
26         warn("open <stdin> failed: %e.\n", fd);
27     } //0用于描述stdin，这里因为第一个被打开，所以stdin会分配到0
28     if ((fd = initfd(1, "stdout:", O_WRONLY)) < 0) {
29         warn("open <stdout> failed: %e.\n", fd);
30     } //1用于描述stdout
31     int ret = main(argc, argv); //真正的“用户程序”
32     exit(ret);
33 }

```

这里我们需要把命令行的输入转换成一个文件，于是需要一个缓冲区：把已经在命令行输入，但还没有被读取的数据放在缓冲区里。这里遇到一个问题：每当控制台输入一个字符，我们都要及时把它放到 `stdin` 的缓冲区里。一般来说，应当有键盘的外设中断来提醒我们。但是我们在QEMU里收不到这个中断，于是采取一个措施：借助时钟中断，每次时钟中断检查是否有新的字符，这效率比较低，不过也还可以接受。

```

1 // kern/trap/trap.c
2 void interrupt_handler(struct trapframe *tf) {
3     intptr_t cause = (tf->cause << 1) >> 1;
4     switch (cause) {

```



```

5      /*...*/
6      case IRQ_S_TIMER:
7          clock_set_next_event();
8
9          if (++ticks % TICK_NUM == 0 && current) {
10             // print_ticks();
11             current->need_resched = 1;
12         }
13         run_timer_list();
14         //按理说用户程序看到的stdin是“只读”的
15         //但是，一个文件，只往外读，不往里写，是不是会导致数据“不守恒”？
16         //我们在这里就是把控制台输入的数据“写到”stdin里(实际上是写到一个缓冲区
17         //这里的cons_getc()并不一定能返回一个字符,可以认为是轮询
18         //如果cons_getc()返回0，那么dev_stdin_write()函数什么都不做
19         dev_stdin_write(cons_getc());
20         break;
21     }
22 }
23 // kern/driver/console.c
24
25 #define CONSBUFSIZE 512
26
27 static struct {
28     uint8_t buf[CONSBUFSIZE];
29     uint32_t rpos;
30     uint32_t wpos; //控制台的输入缓冲区是一个队列
31 } cons;
32
33 /* *
34  * cons_intr - called by device interrupt routines to feed input
35  * characters into the circular console input buffer.
36  * */
37 void cons_intr(int (*proc)(void)) {
38     int c;
39     while ((c = (*proc)()) != -1) {
40         if (c != 0) {
41             cons.buf[cons.wpos++] = c;
42             if (cons.wpos == CONSBUFSIZE) {
43                 cons.wpos = 0;
44             }
45         }
46     }
47 }
48 /* serial_proc_data - get data from serial port */
49 int serial_proc_data(void) {
50     int c = sbi_console_getchar();
51     if (c < 0) {
52         return -1;
53     }
54     if (c == 127) {
55         c = '\b';

```

```

56     }
57     return c;
58 }
59
60 /* serial_intr - try to feed input characters from serial port */
61 void serial_intr(void) {
62     cons_intr(serial_proc_data);
63 }
64 /* *
65  * cons_getc - return the next input character from console,
66  * or 0 if none waiting.
67  * */
68 int cons_getc(void) {
69     int c = 0;
70     bool intr_flag;
71     local_intr_save(intr_flag);
72     {
73         // poll for any pending input characters,
74         // so that this function works even when interrupts are disabled
75         // (e.g., when called from the kernel monitor).
76         serial_intr();
77
78         // grab the next character from the input buffer.
79         if (cons.rpos != cons.wpos) {
80             c = cons.buf[cons.rpos++];
81             if (cons.rpos == CONSBUFSIZE) {
82                 cons.rpos = 0;
83             }
84         }
85     }
86     local_intr_restore(intr_flag);
87     return c;
88 }

```

我们来看 `stdin` 设备的实现:

```

1 // kern/fs/devs/dev_stdin.c
2
3 #define STDIN_BUFSIZE          4096
4 static char stdin_buffer[STDIN_BUFSIZE];
5 //这里又有一个stdin设备的缓冲区，能否和之前console的缓冲区合并？
6 static off_t p_rpos, p_wpos;
7 static wait_queue_t __wait_queue, *wait_queue = &__wait_queue;
8
9 void dev_stdin_write(char c) { //把其他地方的字符写到stdin缓冲区，准备被读取
10     bool intr_flag;

```

```

11     if (c != '\0') {
12         local_intr_save(intr_flag); //禁用中断
13         {
14             stdin_buffer[p_wpos % STDIN_BUFSIZE] = c;
15             if (p_wpos - p_rpos < STDIN_BUFSIZE) {
16                 p_wpos ++;
17             }
18             if (!wait_queue_empty(wait_queue)) {
19                 wakeup_queue(wait_queue, WT_KBD, 1);
20                 //若当前有进程在等待字符输入，则进行唤醒
21             }
22         }
23         local_intr_restore(intr_flag);
24     }
25 }
26 static int dev_stdin_read(char *buf, size_t len) { //读取len个字符
27     int ret = 0;
28     bool intr_flag;
29     local_intr_save(intr_flag);
30     {
31         for (; ret < len; ret ++, p_rpos ++) {
32             try_again:
33             if (p_rpos < p_wpos) { //当前队列非空
34                 *buf ++ = stdin_buffer[p_rpos % STDIN_BUFSIZE];
35             }
36             else {
37                 //希望读取字符，但是当前没有字符，那么进行等待
38                 wait_t __wait, *wait = &__wait;
39                 wait_current_set(wait_queue, wait, WT_KBD);
40                 local_intr_restore(intr_flag);
41
42                 schedule();
43
44                 local_intr_save(intr_flag);
45                 wait_current_del(wait_queue, wait);
46                 if (wait->wakeup_flags == WT_KBD) {
47                     goto try_again; //再次尝试
48                 }
49                 break;
50             }
51         }
52     }
53     local_intr_restore(intr_flag);
54     return ret;
55 }
56 static int stdin_io(struct device *dev, struct iobuf *iob, bool write) {
57     //对应struct device 的d_io()
58     if (!write) {
59         int ret;
60         if ((ret = dev_stdin_read(iob->io_base, iob->io_resid)) > 0) {
61             iob->io_resid -= ret;

```

```

62     }
63     return ret;
64 }
65 return -E_INVAL;
66 }

```

disk0设备

封装了一下 `ramdisk` 的接口，每次读取或者写入若干个block。

```

1  // kern/fs/devs/dev_disk0.c
2  static char *disk0_buffer;
3  static semaphore_t disk0_sem;
4
5  static void
6  lock_disk0(void) {
7      down(&(disk0_sem));
8  }
9
10 static void
11 unlock_disk0(void) {
12     up(&(disk0_sem));
13 }
14
15 static void disk0_read_blks_nolock(uint32_t blkno, uint32_t nblks) {
16     int ret;
17     uint32_t sectno = blkno * DISK0_BLK_NSECT, nsecs = nblks * DISK0_BLK_NSECT;
18     if ((ret = ide_read_secs(DISK0_DEV_NO, sectno, disk0_buffer, nsecs)) != 0)
19         panic("disk0: read blkno = %d (sectno = %d), nblks = %d (nsecs = %d)",
20               blkno, sectno, nblks, nsecs, ret);
21 }
22 }
23
24 static void disk0_write_blks_nolock(uint32_t blkno, uint32_t nblks) {
25     int ret;
26     uint32_t sectno = blkno * DISK0_BLK_NSECT, nsecs = nblks * DISK0_BLK_NSECT;
27     if ((ret = ide_write_secs(DISK0_DEV_NO, sectno, disk0_buffer, nsecs)) != 0)
28         panic("disk0: write blkno = %d (sectno = %d), nblks = %d (nsecs = %d)",
29               blkno, sectno, nblks, nsecs, ret);
30 }
31 }
32
33 static int disk0_io(struct device *dev, struct iobuf *iob, bool write) {

```

```

34     off_t offset = iob->io_offset;
35     size_t resid = iob->io_resid;
36     uint32_t blkno = offset / DISK0_BLKSIZE;
37     uint32_t nblks = resid / DISK0_BLKSIZE;
38
39     /* don't allow I/O that isn't block-aligned */
40     if ((offset % DISK0_BLKSIZE) != 0 || (resid % DISK0_BLKSIZE) != 0) {
41         return -E_INVAL;
42     }
43
44     /* don't allow I/O past the end of disk0 */
45     if (blkno + nblks > dev->d_blocks) {
46         return -E_INVAL;
47     }
48
49     /* read/write nothing ? */
50     if (nblks == 0) {
51         return 0;
52     }
53
54     lock_disk0();
55     while (resid != 0) {
56         size_t copied, alen = DISK0_BUFSIZE;
57         if (write) {
58             iobuf_move(iob, disk0_buffer, alen, 0, &copied);
59             assert(copied != 0 && copied <= resid && copied % DISK0_BLKSIZE == 0);
60             nblks = copied / DISK0_BLKSIZE;
61             disk0_write_blks_nolock(blkno, nblks);
62         }
63         else {
64             if (alen > resid) {
65                 alen = resid;
66             }
67             nblks = alen / DISK0_BLKSIZE;
68             disk0_read_blks_nolock(blkno, nblks);
69             iobuf_move(iob, disk0_buffer, alen, 1, &copied);
70             assert(copied == alen && copied % DISK0_BLKSIZE == 0);
71         }
72         resid -= copied, blkno += nblks;
73     }
74     unlock_disk0();
75     return 0;
76 }

```

这些设备的实现看起来比较复杂，实际上属于比较麻烦的设备驱动的部分。

从中断到终端

可以说，我们的操作系统之旅，从zhong duan(中断)开始，也在zhong duan(终端)结束。

我们的终端需要实现这样的功能: 根据输入的程序名称, 从文件系统里加载对应的程序并执行。我们采取 `fork()` `exec()` 的方式来加载执行程序，`exec()` 的一系列接口都需要重写来使用文件系统。以 `do_execve()` 为例，

以前的函数原型从内存的某个位置加载程序

```
int do_execve(const char *name, size_t len, unsigned char *binary, size_t si
```

现在则调用文件系统接口加载程序：

```
1 // kern/process/proc.c
2 // do_execve - call exit_mmap(mm)&put_pgdir(mm) to reclaim memory space of
3 //             - call load_icode to setup new memory space accroding binary
4 int
5 do_execve(const char *name, int argc, const char **argv) {
6     static_assert(EXEC_MAX_ARG_LEN >= FS_MAX_FPATH_LEN);
7     struct mm_struct *mm = current->mm;
8     if (!(argc >= 1 && argc <= EXEC_MAX_ARG_NUM)) {
9         return -E_INVAL;
10    }
11
12    char local_name[PROC_NAME_LEN + 1];
13    memset(local_name, 0, sizeof(local_name));
14
15    char *kargv[EXEC_MAX_ARG_NUM];
16    const char *path;
17
18    int ret = -E_INVAL;
19
20    lock_mm(mm);
21    if (name == NULL) {
22        snprintf(local_name, sizeof(local_name), "<null> %d", current->pid);
23    }
24    else {
25        if (!copy_string(mm, local_name, name, sizeof(local_name))) {
26            unlock_mm(mm);
```

```

27         return ret;
28     }
29 }
30 if ((ret = copy_kargv(mm, argc, kargv, argv)) != 0) {
31     unlock_mm(mm);
32     return ret;
33 }
34 path = argv[0];
35 unlock_mm(mm);
36 files_closeall(current->filesp);
37
38 /* sysfile_open will check the first argument path, thus we have to us
39 int fd;
40 if ((ret = fd = sysfile_open(path, O_RDONLY)) < 0) {
41     goto execve_exit;
42 }
43 if (mm != NULL) {
44     lcr3(boot_cr3);
45     if (mm_count_dec(mm) == 0) {
46         exit_mmap(mm);
47         put_pgdir(mm);
48         mm_destroy(mm);
49     }
50     current->mm = NULL;
51 }
52 ret= -E_NO_MEM;;
53 if ((ret = load_icode(fd, argc, kargv)) != 0) {
54     goto execve_exit;
55 }
56 put_kargv(argc, kargv);
57 set_proc_name(current, local_name);
58 return 0;
59
60 execve_exit:
61     put_kargv(argc, kargv);
62     do_exit(ret);
63     panic("already exit: %e.\n", ret);
64 }

```

我们还要看一下终端程序的实现。可以发现终端程序需要对命令进行词法和语法分析。

```

1 // user/sh.c
2 #include <ulib.h>
3 #include <stdio.h>
4 #include <string.h>
5 #include <dir.h>

```

```

6  #include <file.h>
7  #include <error.h>
8  #include <unistd.h>
9
10 #define printf(...)      fprintf(1, __VA_ARGS__)
11 #define putc(c)          printf("%c", c)
12
13 #define BUFSIZE          4096
14 #define WHITESPACE      " \t\r\n"
15 #define SYMBOLS          "<|>&;"
16
17 char shcwd[BUFSIZE];
18
19 int
20 gettoken(char **p1, char **p2) {
21     char *s;
22     if ((s = *p1) == NULL) {
23         return 0;
24     }
25     while (strchr(WHITESPACE, *s) != NULL) {
26         *s ++ = '\0';
27     }
28     if (*s == '\0') {
29         return 0;
30     }
31
32     *p2 = s;
33     int token = 'w';
34     if (strchr(SYMBOLS, *s) != NULL) {
35         token = *s, *s ++ = '\0';
36     }
37     else {
38         bool flag = 0;
39         while (*s != '\0' && (flag || strchr(WHITESPACE SYMBOLS, *s) == NU
40             if (*s == '"') {
41                 *s = ' ', flag = !flag;
42             }
43             s ++;
44         }
45     }
46     *p1 = (*s != '\0' ? s : NULL);
47     return token;
48 }
49
50 char * readline(const char *prompt) {
51     static char buffer[BUFSIZE];
52     if (prompt != NULL) {
53         printf("%s", prompt);
54     }
55     int ret, i = 0;
56     while (1) {

```



```

57     char c;
58     if ((ret = read(0, &c, sizeof(char))) < 0) {
59         return NULL;
60     }
61     else if (ret == 0) {
62         if (i > 0) {
63             buffer[i] = '\0';
64             break;
65         }
66         return NULL;
67     }
68
69     if (c == 3) {
70         return NULL;
71     }
72     else if (c >= ' ' && i < BUFSIZE - 1) {
73         putc(c);
74         buffer[i++] = c;
75     }
76     else if (c == '\b' && i > 0) {
77         putc(c);
78         i--;
79     }
80     else if (c == '\n' || c == '\r') {
81         putc(c);
82         buffer[i] = '\0';
83         break;
84     }
85 }
86 return buffer;
87 }
88
89 void
90 usage(void) {
91     printf("usage: sh [command-file]\n");
92 }
93
94 int
95 reopen(int fd2, const char *filename, uint32_t open_flags) {
96     int ret, fd1;
97     close(fd2);
98     if ((ret = open(filename, open_flags)) >= 0 && ret != fd2) {
99         close(fd2);
100         fd1 = ret, ret = dup2(fd1, fd2);
101         close(fd1);
102     }
103     return ret < 0 ? ret : 0;
104 }
105
106 int
107 testfile(const char *name) {

```

```

108     int ret;
109     if ((ret = open(name, O_RDONLY)) < 0) {
110         return ret;
111     }
112     close(ret);
113     return 0;
114 }
115
116 int
117 runcmd(char *cmd) {
118     static char argv0[BUFSIZE];
119     static const char *argv[EXEC_MAX_ARG_NUM + 1]; // must be static!
120     char *t;
121     int argc, token, ret, p[2];
122 again:
123     argc = 0;
124     while (1) {
125         switch (token = gettoken(&cmd, &t)) {
126             case 'w':
127                 if (argc == EXEC_MAX_ARG_NUM) {
128                     printf("sh error: too many arguments\n");
129                     return -1;
130                 }
131                 argv[argc++] = t;
132                 break;
133             case '<':
134                 if (gettoken(&cmd, &t) != 'w') {
135                     printf("sh error: syntax error: < not followed by word\n");
136                     return -1;
137                 }
138                 if ((ret = reopen(0, t, O_RDONLY)) != 0) {
139                     return ret;
140                 }
141                 break;
142             case '>':
143                 if (gettoken(&cmd, &t) != 'w') {
144                     printf("sh error: syntax error: > not followed by word\n");
145                     return -1;
146                 }
147                 if ((ret = reopen(1, t, O_RDWR | O_TRUNC | O_CREAT)) != 0) {
148                     return ret;
149                 }
150                 break;
151             case '|':
152                 // if ((ret = pipe(p)) != 0) {
153                 //     return ret;
154                 // }
155                 if ((ret = fork()) == 0) {
156                     close(0);
157                     if ((ret = dup2(p[0], 0)) < 0) {
158                         return ret;

```

```

159         }
160         close(p[0]), close(p[1]);
161         goto again;
162     }
163     else {
164         if (ret < 0) {
165             return ret;
166         }
167         close(1);
168         if ((ret = dup2(p[1], 1)) < 0) {
169             return ret;
170         }
171         close(p[0]), close(p[1]);
172         goto runit;
173     }
174     break;
175 case 0:
176     goto runit;
177 case ';':
178     if ((ret = fork()) == 0) {
179         goto runit;
180     }
181     else {
182         if (ret < 0) {
183             return ret;
184         }
185         waitpid(ret, NULL);
186         goto again;
187     }
188     break;
189 default:
190     printf("sh error: bad return %d from gettoken\n", token);
191     return -1;
192 }
193 }
194
195 runit:
196     if (argc == 0) {
197         return 0;
198     }
199     else if (strcmp(argv[0], "cd") == 0) {
200         if (argc != 2) {
201             return -1;
202         }
203         strcpy(shcwd, argv[1]);
204         return 0;
205     }
206     if ((ret = testfile(argv[0])) != 0) {
207         if (ret != -E_NOENT) {
208             return ret;
209         }

```

```

210     snprintf(argv0, sizeof(argv0), "%s", argv[0]);
211     argv[0] = argv0;
212 }
213 argv[argc] = NULL;
214 return __exec(argv[0], argv);
215 }
216
217 int main(int argc, char **argv) {
218     cputs("user sh is running!!!");
219     int ret, interactive = 1;
220     if (argc == 2) {
221         if ((ret = reopen(0, argv[1], O_RDONLY)) != 0) {
222             return ret;
223         }
224         interactive = 0;
225     }
226     else if (argc > 2) {
227         usage();
228         return -1;
229     }
230     //shcwd = malloc(BUFSIZE);
231     assert(shcwd != NULL);
232
233     char *buffer;
234     while ((buffer = readline((interactive) ? "$ " : NULL)) != NULL) {
235         shcwd[0] = '\0';
236         int pid;
237         if ((pid = fork()) == 0) {
238             ret = runcmd(buffer);
239             exit(ret);
240         }
241         assert(pid >= 0);
242         if (waitpid(pid, &ret) == 0) {
243             if (ret == 0 && shcwd[0] != '\0') {
244                 ret = 0;
245             }
246             if (ret != 0) {
247                 printf("error: %d - %e\n", ret, ret);
248             }
249         }
250     }
251     return 0;
252 }

```

如果我们能够把终端运行起来，并能输入命令执行用户程序，就说明程序运行正常。

项目组成与执行流

项目组成

```
1  lab8
2  |— Makefile
3  |— disk0
4  |   |— badarg
5  |   |— badsegment
6  |   |— divzero
7  |   |— exit
8  |   |— faultread
9  |   |— faultreadkernel
10 |   |— forktest
11 |   |— forktree
12 |   |— hello
13 |   |— matrix
14 |   |— pgdir
15 |   |— priority
16 |   |— sh
17 |   |— sleep
18 |   |— sleepkill
19 |   |— softint
20 |   |— spin
21 |   |— testbss
22 |   |— waitkill
23 |   |— yield
24 |— giveitapy.pyq
25 |— kern
26 |   |— debug
27 |       |— assert.h
28 |       |— kdebug.c
29 |       |— kdebug.h
30 |       |— kmonitor.c
31 |       |— kmonitor.h
32 |       |— panic.c
33 |       |— stab.h
34 |   |— driver
35 |       |— clock.c
36 |       |— clock.h
37 |       |— console.c
38 |       |— console.h
39 |       |— ide.c
40 |       |— ide.h
41 |       |— intr.c
42 |       |— intr.h
```

```
43 | | | |— kbdreg.h
44 | | | |— picirq.c
45 | | | |— picirq.h
46 | | | |— ramdisk.c
47 | | | |— ramdisk.h
48 | |— fs
49 | | |— devs
50 | | | |— dev.c
51 | | | |— dev.h
52 | | | |— dev_disk0.c
53 | | | |— dev_stdin.c
54 | | | |— dev_stdout.c
55 | | |— file.c
56 | | |— file.h
57 | | |— fs.c
58 | | |— fs.h
59 | | |— iobuf.c
60 | | |— iobuf.h
61 | | |— sfs
62 | | | |— bitmap.c
63 | | | |— bitmap.h
64 | | | |— sfs.c
65 | | | |— sfs.h
66 | | | |— sfs_fs.c
67 | | | |— sfs_inode.c
68 | | | |— sfs_io.c
69 | | | |— sfs_lock.c
70 | | |— swap
71 | | | |— swapfs.c
72 | | | |— swapfs.h
73 | | |— sysfile.c
74 | | |— sysfile.h
75 | |— vfs
76 | | |— inode.c
77 | | |— inode.h
78 | | |— vfs.c
79 | | |— vfs.h
80 | | |— vfsdev.c
81 | | |— vfsfile.c
82 | | |— vfslookup.c
83 | | |— vfspath.c
84 |— init
85 | |— entry.S
86 | |— init.c
87 |— libs
88 | |— readline.c
89 | |— stdio.c
90 | |— string.c
91 |— mm
92 | |— default_pmm.c
93 | |— default_pmm.h
```

```

94 | | | | — kmalloc.c
95 | | | | — kmalloc.h
96 | | | | — memlayout.h
97 | | | | — mmu.h
98 | | | | — pmm.c
99 | | | | — pmm.h
100 | | | | — swap.c
101 | | | | — swap.h
102 | | | | — swap_fifo.c
103 | | | | — swap_fifo.h
104 | | | | — vmm.c
105 | | | | — vmm.h
106 | | — process
107 | | | — entry.S
108 | | | — proc.c
109 | | | — proc.h
110 | | | — switch.S
111 | | — schedule
112 | | | — default_sched.h
113 | | | — default_sched.c
114 | | | — default_sched_stride.c
115 | | | — sched.c
116 | | | — sched.h
117 | | — sync
118 | | | — check_sync.c
119 | | | — monitor.c
120 | | | — monitor.h
121 | | | — sem.c
122 | | | — sem.h
123 | | | — sync.h
124 | | | — wait.c
125 | | | — wait.h
126 | | — syscall
127 | | | — syscall.c
128 | | | — syscall.h
129 | | — trap
130 | | | — trap.c
131 | | | — trap.h
132 | | | — trapentry.S
133 | — lab5.md
134 | — libs
135 | | — atomic.h
136 | | — defs.h
137 | | — dirent.h
138 | | — elf.h
139 | | — error.h
140 | | — hash.c
141 | | — list.h
142 | | — printfmt.c
143 | | — rand.c
144 | | — riscv.h

```

```
145 | | sbi.h
146 | | skew_heap.h
147 | | stat.h
148 | | stdarg.h
149 | | stdio.h
150 | | stdlib.h
151 | | string.c
152 | | string.h
153 | | unistd.h
154 | — tools
155 | | boot.ld
156 | | function.mk
157 | | gdbinit
158 | | grade.sh
159 | | kernel.ld
160 | | mksfs.c
161 | | sign.c
162 | | user.ld
163 | | vector.c
164 | — user
165 | | badarg.c
166 | | badsegment.c
167 | | divzero.c
168 | | exit.c
169 | | faultread.c
170 | | faultreadkernel.c
171 | | forktest.c
172 | | forktree.c
173 | | hello.c
174 | | libs
175 | | | dir.c
176 | | | dir.h
177 | | | file.c
178 | | | file.h
179 | | | initcode.S
180 | | | lock.h
181 | | | panic.c
182 | | | stdio.c
183 | | | syscall.c
184 | | | syscall.h
185 | | | ulib.c
186 | | | ulib.h
187 | | | umain.c
188 | | matrix.c
189 | | pgdir.c
190 | | priority.c
191 | | sh.c
192 | | sleep.c
193 | | sleepkill.c
194 | | softint.c
195 | | spin.c
```



```
196 |— testbss.c
197 |— waitkill.c
198 |— yield.c
199
200 21 directories, 176 files
```

通过文件系统接口

`user/libs/file.[ch]|dir.[ch]|syscall.c` : 与文件系统操作相关的用户库实行 ;

`kern/syscall.[ch]` : 文件中包含文件系统相关的内核态系统调用接口 ;

`kern/fs/sysfile.[ch]|file.[ch]` : 通用文件系统接口和实行 ;

文件系统抽象层-VFS

`kern/fs/vfs/*.ch` : 虚拟文件系统接口与实现

simple FS文件系统

`kern/fs/sfs/*.ch` : SimpleFS文件系统实现

文件系统的硬盘IO接口

`kern/fs/devs/dev.[ch]|dev_disk0.c` : disk0硬盘设备提供给文件系统的I/O访问接口和实现

执行流

结合前面所述自行理解、总结。