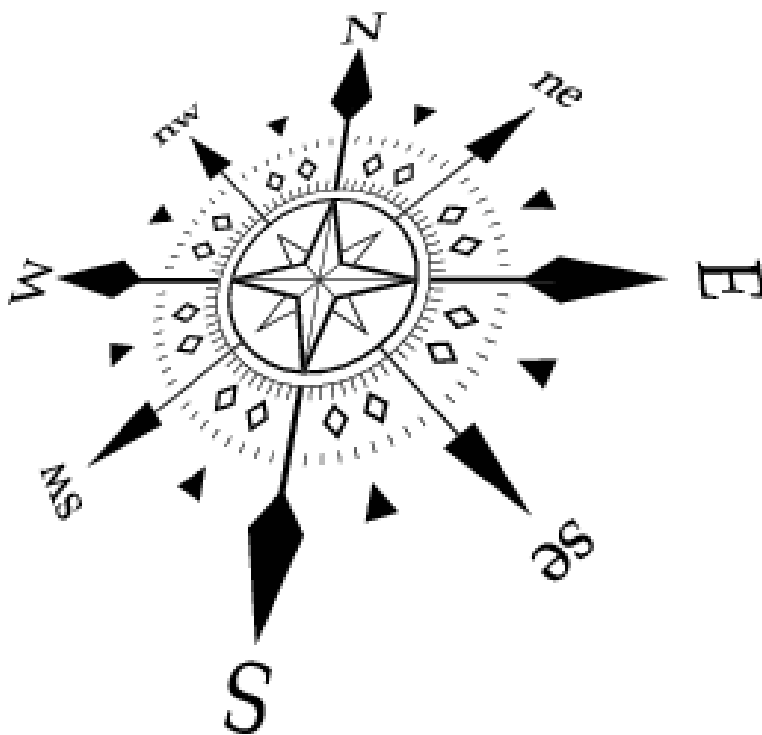

MOS 操作系统实验 指导书

操作系统课程实验任务及相关说明

带你体验自己动手完成一个小操作系统的乐趣



*SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
BEIHANG UNIVERSITY*

2025 年 2 月 27 日

POWERED BY L^AT_EX

引言	1
引言	1
实验内容	2
实验设计	2
实验环境	3
虚拟机平台	4
Git 服务器	4
自动评测	5
0 初识操作系统	6
0.1 实验目的	6
0.2 初识实验	6
0.2.1 了解实验环境	6
0.2.2 通过网站远程访问本实验环境	7
0.2.3 命令行界面 (CLI)	7
0.3 基础操作介绍	8
0.3.1 命令行	8
0.3.2 Linux 基本操作命令	8
0.4 实用工具介绍	11
0.4.1 Vim	11
0.4.2 GCC	12
0.4.3 Makefile	13
0.4.4 ctags	15
0.5 Git 专栏-轻松维护和提交代码	16
0.5.1 Git 是什么?	17
0.5.2 Git 基础指引	18
0.5.3 Git 文件状态	20

0.5.4	Git 三棵树	22
0.5.5	Git 版本回退	23
0.5.6	Git 分支	25
0.5.7	Git 远程仓库与本地	27
0.6	进阶操作	27
0.6.1	Linux 操作补充	27
0.6.2	shell 脚本	31
0.6.3	重定向和管道	33
0.7	实战测试	35
0.8	实验思考	38
1	内核、启动和 printf	39
1.1	实验目的	39
1.2	操作系统的启动	39
1.2.1	QEMU 模拟器	39
1.2.2	QEMU 中的启动流程	40
1.3	Let's hack the kernel!	40
1.3.1	Makefile——内核代码的地图	40
1.3.2	ELF——操作系统内核的本质	42
1.3.3	MIPS 内存布局——内核运行的正确位置	48
1.3.4	Linker Script——控制内核加载地址	50
1.4	从零开始搭建 MOS	52
1.4.1	从 make 开始	52
1.4.2	_start 函数	54
1.5	实战 printf	55
1.6	实验正确结果	57
1.7	任务列表	57
1.8	实验思考	57
2	内存管理	58
2.1	实验目的	58
2.2	4Kc 访存流程概览	60
2.2.1	CPU 发出地址	60
2.2.2	虚拟地址映射到物理地址	60
2.3	内核程序启动	60
2.3.1	mips_detect_memory	61
2.3.2	mips_vm_init 函数	61
2.3.3	mips_vm_init	63
2.4	物理内存管理	64
2.4.1	链表宏	64

2.4.2	页控制块	65
2.4.3	其他相关函数	66
2.4.4	正确结果展示	68
2.5	虚拟内存管理	68
2.5.1	两级页表结构	68
2.5.2	与页表相关的函数	70
2.6	访问内存与 TLB 重填	73
2.6.1	TLB 相关的前置知识	73
2.6.2	TLB 维护流程	76
2.6.3	正确结果展示	79
2.7	Lab2 在 MOS 中的概况	79
2.8	其他体系结构中的内存管理	82
2.9	任务列表	82
2.10	实验思考	82
3	进程与异常	83
3.1	实验目的	83
3.2	进程	83
3.2.1	进程控制块	83
3.2.2	段地址映射	85
3.2.3	进程的标识	85
3.2.4	设置进程控制块	86
3.2.5	加载二进制镜像	89
3.2.6	创建进程	91
3.2.7	进程运行与切换	92
3.2.8	实验正确结果	94
3.3	中断与异常	94
3.3.1	异常的分发	95
3.3.2	异常向量组	97
3.3.3	时钟中断	98
3.3.4	进程调度	100
3.4	Lab3 在 MOS 中的概况	101
3.5	实验正确结果	102
3.6	代码导读	102
3.7	任务列表	103
3.8	实验思考	104
4	系统调用与 fork	105
4.1	实验目的	105
4.2	系统调用 (System Call)	105

4.2.0	用户态与内核态	105
4.2.1	系统调用实例	106
4.2.2	系统调用机制的实现	107
4.2.3	基础系统调用函数	111
4.3	进程间通信机制 (IPC)	113
4.4	Fork	115
4.4.1	初窥 fork	116
4.4.2	写时复制机制	120
4.4.3	fork 的返回值	120
4.4.4	地址空间的准备	121
4.4.5	页写入异常	122
4.4.6	使用用户程序进行测试	126
4.5	实验正确结果	126
4.6	任务列表	127
4.7	实验思考	128
5	文件系统	129
5.1	实验目的	129
5.2	文件系统概述	129
5.2.1	文件系统的设计与实现	130
5.3	IDE 磁盘驱动	131
5.3.1	内存映射 I/O (MMIO)	132
5.3.2	IDE 磁盘	133
5.3.3	驱动程序编写	135
5.4	文件系统结构	138
5.4.1	磁盘文件系统布局	138
5.4.2	文件系统详细结构	140
5.4.3	块缓存	141
5.5	文件系统的用户接口	143
5.5.1	文件描述符	143
5.5.2	文件系统服务	145
5.6	正确结果展示	146
5.6.1	IDE 磁盘交互	147
5.6.2	文件系统测试	147
5.6.3	文件系统服务测试	147
5.7	任务列表	148
5.8	实验思考	148

6 管道与 Shell	149
6.1 实验目的	149
6.2 管道	149
6.2.1 初窥管道	149
6.2.2 mos 中 pipe 的使用与实现	150
6.2.3 管道的读写	151
6.2.4 管道关闭的正确判断	154
6.2.5 相关函数	156
6.3 shell	158
6.3.1 完善 spawn 函数	158
6.3.2 解释 shell 命令	159
6.3.3 相关函数	161
6.4 实验正确结果	164
6.4.1 管道测试	164
6.4.2 shell 测试	164
6.5 任务列表	165
6.6 实验思考	165
A 补充知识	166
A.1 shell 编程易错点说明	166
A.2 真实操作系统的内核及启动详解	167
A.2.1 Bootloader	168
A.3 编译与链接详解	170
A.4 printk 格式具体说明	174
A.5 MIPS 汇编与 C 语言	175
A.5.1 循环与判断	175
A.5.2 函数调用	176
A.5.3 通用寄存器使用约定	178
A.5.4 LEAF、NESTED 和 END	179
A.6 多级页表与页目录自映射	180
A.6.1 MOS 中的页目录自映射应用	180
A.6.2 其他页表机制	181
A.6.3 虚拟内存和磁盘中的 ELF 文件	182

引言

操作系统是计算机系统中软件与硬件联系的纽带，课程内容丰富，既包含操作系统的基础理论，又涉及实际操作系统的设计与实现。操作系统实验设计是操作系统课程实践环节的集中表现，旨在巩固学生理论课学习的概念和原理，同时培养学生的工程实践能力。一些国内外著名大学都非常重视操作系统的实验设计，例如麻省理工学院的 Frans Kaashoek 等设计的 JOS 和 xv6 教学操作系统、哈佛大学的 David A. Holland 等设计的 OS161 操作系统用于实现操作系统实验教学。

我们尝试了 MINIX、Nachos、Linux、Windows 等操作系统实验，发现以 Linux 和 Windows 为基础的实验，由于系统规模庞大，很难让学生建立起完整的操作系统概念，导致专业知识碎片化，不符合系统能力培养目标。此外，操作系统涉及硬件的很多相关知识，本身还包含并发程序设计等比较难理解的概念，因此学生的学习曲线很陡峭，如何让学生由浅入深，平滑地掌握这些知识是实验设计的难点。本书设计实验的基本目标是：一学期内设计实现一个可在实际硬件平台上运行的小型操作系统，该系统具备现代操作系统特征（如虚存管理、多进程等），符合工业标准。

基于系统能力培养的理念和目标希望构建计算机组成原理、操作系统和编译原理等课程的一体化实验体系，因而本书操作系统课程设计采用了计算机组成原理课程中的 MIPS 指令系统（MIPS 4Kc）作为硬件基础，参考 JOS 的设计思路、方法和源代码，实现一个可以在 MIPS 平台上运行的小型操作系统，包括操作系统启动、物理内存管理、虚拟内存管理、进程管理、中断处理、系统调用、文件系统、Shell 等主要操作系统主要功能。为了降低学习难度，采用增量式实验设计思想，每个实验包含的内核代码量（C、汇编、注释）在几百行左右，提供代码框架和代码示例。每个实验可以独立运行和评测，但是后面的实验依赖前面的实验，学生实现的代码从 Lab1 贯穿到 Lab6，最后实现一个完成的小型操作系统。

实验内容

本书设计的操作系统实验分为 6 个实验 (Lab1 ~ Lab6)，目标是在一学期内自主开发一个小型操作系统。各个实验的相互关系如图 1 所示，具体实验内容如下。

1. **内核、启动和 printf**: 通过 PC 启动的实验，掌握硬件的启动过程，理解链接地址、加载地址和重定位的概念，学习如何编写裸机代码。
2. **内存管理**: 理解虚拟内存和物理内存的管理，实现操作系统对虚拟内存空间的管理。
3. **进程与异常**: 通过设置进程控制块和编写进程创建、进程中止和进程调度程序，实现进程管理；编写通用中断分派程序和时钟中断例程，实现中断管理。
4. **系统调用与 fork**: 掌握系统调用的实现方法，理解系统调用的处理流程，实现本实验所需的系统调用。
5. **文件系统**: 通过实现一个简单的、基于磁盘的、微内核方式的文件系统，掌握文件系统的实现方法和层次结构。
6. **管道与 shell**: 实现具有管道，重定向功能的命令解释程序 shell，能够执行一些简单的命令。最后将 6 部分链接起来，使之成为一个能够运行的操作系统。

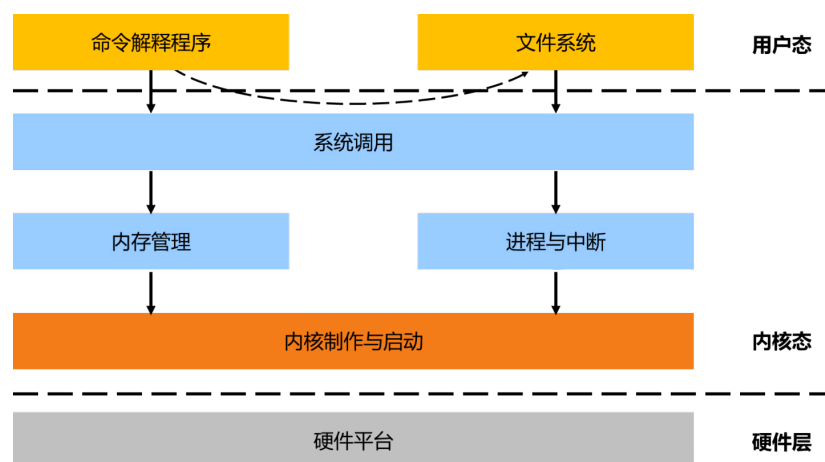


图 1: 实验内容的关系

另外，考虑有些学生对 Linux 系统、GCC 编译器、Makefile 和 git 等工具不熟悉，专门设置了一个 Lab0，主要介绍 Linux、Makefile、git、vi 和仿真器的使用以及基本的 shell 编程等，为后续实验的顺利实施打好基础。

实验设计

由于开发一个实际的操作系统难度大、工作量繁重，为了保证教学效果，在核心能力部分采用微内核结构和增量式设计的原则，因此可以从最基本的硬件管理功能逐步扩充，最后完成一个完整的系统。实验内容的设计满足以下条件。

1. 每个实验可独立运行与测试，便于调试与评测，可获得阶段性成果。
2. 每个实验内容包含相对独立的知识点，并只依赖其前序实验。
3. 基本保证在两周内完成一个实验，这样在一学期内可以完成整个实验。
4. 各个实验提交的代码一直伴随整个实验过程，可以不断改进、完善代码。

整个系统结构如图 2 所示，蓝色部分是每次实验需要新增加的模块，绿色部分是需要修改完善的模块，灰色部分是不用修改的模块。在增量式设计下，可以从基本的功能出发，逐步完善整个系统，从而降低了学习操作系统的难度。

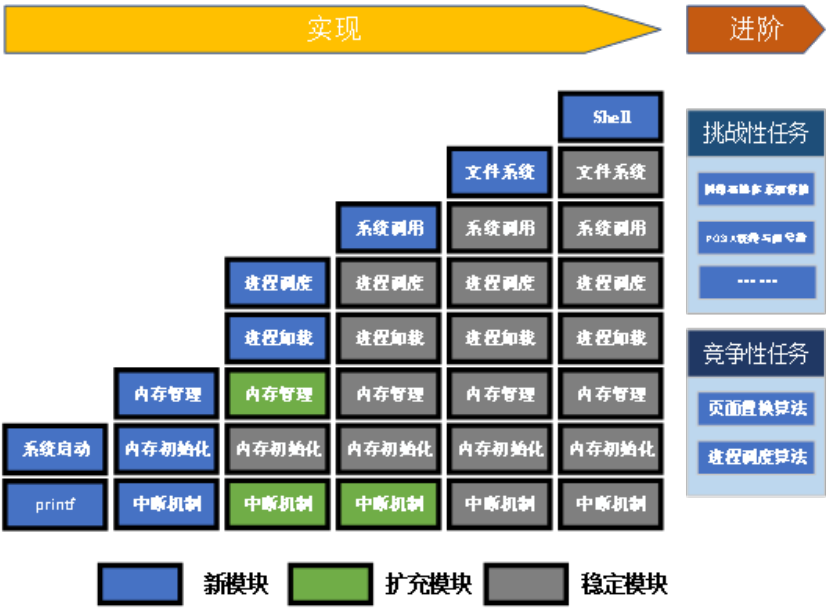


图 2: 增量式实验方法

为了适应不同读者的学习要求，本书的实验采用分层的方式，从基础到复杂逐步实现实验的基本目标。因此，可将实验基本目标分为三个层次：

- 第一层次，掌握基本的系统使用与编程能力：包括Linux、Makefile、git、vi 和仿真器的使用，基本的 shell 编程和使用系统调用编程；
- 第二层次，掌握操作系统核心能力：包括 6 个实验，从操作系统内核构造、内存管理、进程管理、系统调用、文件系统和命令解释程序，构成一个完整的小型操作系统；
- 第三层次，锻炼操作系统提升能力：主要包括若干挑战性任务，学生需要独立在某一方面实现若干新的系统功能。

实验环境

一次实验的整个流程包括，初始代码发布、代码编写、调试运行、代码提交、编译测试以及评分结果的反馈。为了方便学生和教师，我们设计了操作系统实验集成环境，采

用 git 进行版本管理，保证学生之间的代码互不可见，而教师和助教可以方便地查看每位学生的代码。整个环境的结构如图 3 所示。为了满足实验需求，整个系统分为以下几个部分。

- a) 虚拟机平台，包含实验需要的开发环境，例如 Linux 环境、交叉编译器、MIPS 仿真器等。
- b) git 服务器，包括学生各个实验的代码以及相关信息。
- c) 自动评测和反馈，这部分集成在 git 服务器中，后面会详细介绍。

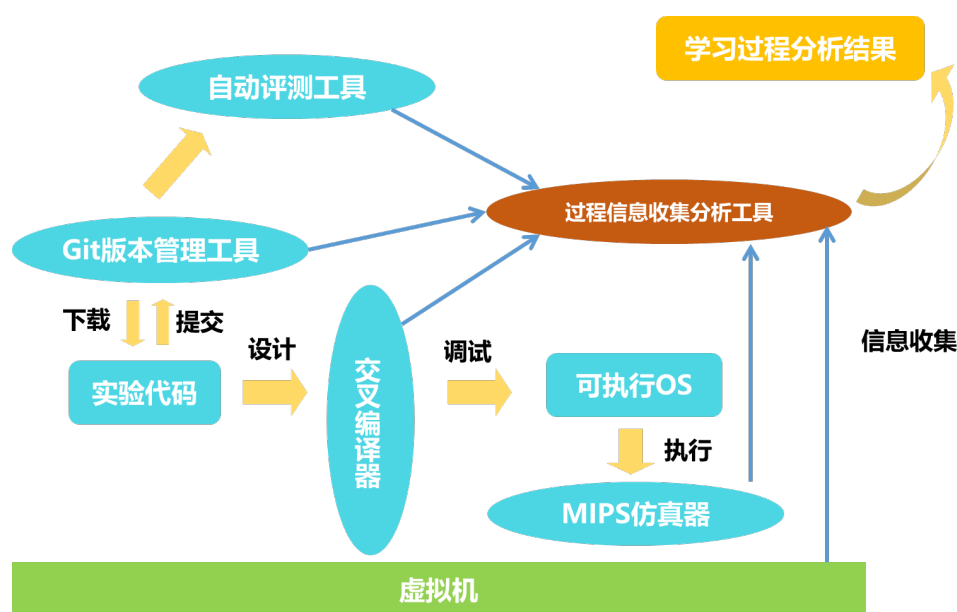


图 3: 操作系统实验集成环境

虚拟机平台

为了方便地收集学习过程数据，同时尽量降低学生端设备的要求，我们提供虚拟机作为实验后端，虚拟机中使用 Linux 系统，方便部署环境，整个实验过程在虚拟机中完成。在虚拟机中，需要部署相应仿真器、编译器、文件编辑器等环境。推荐使用 QEMU 作为仿真器，MIPS 的 gcc 交叉编译器作为编译器，vim 作为文件编辑器。这些工具已经部署在虚拟机环境中，避免了软件版本冲突问题，方便自动评测系统的部署与实施。同时，节省安装实验环境花费的时间。每位学生可以登录到实验系统，完成实验。

Git 服务器

为了保证学生代码安全，同时提供方便的代码版本管理工具，提供一个 git 服务器，每位学生的代码编写过程在虚拟机中完成，同时将代码托管在 git 服务器中，避免了故障导致的损失。同时，实验发布及测试结果反馈均通过 git 服务器完成。在 git 服务器中，每个学生拥有一个独立的代码库，对于每位学生来说，只有自己的代码库是可见的，

每位学生可以随时下载和提交自己代码库中的代码。同时，所有学生的代码库对于助教和教师是可见的，所有的助教和教师都可以下载学生代码。

自动评测

由于学生人数众多，对于学生实验代码的评判是一个繁复、机械化的过程，助教和教师手动评测非常困难。自动评测系统能对学生提交的代码自动给出相应的评分。当学生执行代码提交后，可提交评测。评测系统将获取学生代码，依次完成编译、运行和测试，给出评测结果反馈结果给学生查看。

CHAPTER 0

初识操作系统

0.1 实验目的

1. 认识操作系统实验环境。
2. 掌握操作系统实验所需的基本工具。

在本实验中，需要了解实验环境，熟悉 Linux 操作系统 (Ubuntu)，了解控制终端，掌握一些常用工具并能够脱离可视化界面进行工作。本实验难度不大，重点是熟悉操作系统实验环境的各类工具，为后续实验的开展奠定基础。

0.2 初识实验

“工欲善其事必先利其器”，应对我们的环境和工具有足够的了解，才能顺利开展实验工作。

0.2.1 了解实验环境

实验环境整体配置如下：

- 操作系统：Linux 内核，Ubuntu 操作系统
- 硬件模拟器：QEMU
- 编译器：GCC
- 版本控制：Git

Ubuntu 操作系统是一款开源的 GNU/Linux 操作系统，它基于 Linux 内核实现，是目前较为流行的 Linux 发行版之一。GNU (GNU is Not Unix 的递归缩写) 是一套开源计划，其中包含了三个协议条款，为用户提供了大量开源软件。而人们常说的 Linux，从

严格意义上是指 Linux 内核，基于该内核的操作系统众多，具有免费、可靠、安全、稳定、多平台等特点。

QEMU 是一款硬件模拟器，可以模拟所需硬件环境，例如本实验需要的 MIPS 架构下的 CPU。

GCC 是一套免费、开源的编译器，诞生并服务于 GNU 计划，最初名称为 GNU C Compiler，后来因支持更多编程语言而改名为 GNU Compiler Collection，很多集成开发环境 (Integrated Development Environment, IDE) 的编译器用的就是 GCC 套件，例如 Dev-C++，Code::Blocks 等。本实验将使用基于 MIPS 的 GCC 交叉编译器。

Git 是一款免费、开源的版本控制系统，本书实验将利用它来提供管理、发布、提交、评测等功能。第 0.5 节将会详细介绍 Git 如何使用。



图 0.1: Ubuntu, GNU, Linux

0.2.2 通过网站远程访问本实验环境

为了方便访问，本实验提供通过网站远程访问虚拟机的方法，不需要手动进行复杂的配置。可直接按照下面的流程，进入和上文下相同的界面。

本书实验的虚拟环境的网站是 <http://lab.os.buaa.edu.cn>，在网页的右上角点击进入 Web 终端即可在网页上使用我们提供的虚拟机。

0.2.3 命令行界面 (CLI)

对于目前主流的操作系统，如 Windows、Mac OS、Ubuntu 等均使用图形用户界面 (Graphical user interface, GUI)，但是在操作系统课程中，需要了解命令行界面 (Command line Interface, CLI)。

其实，上文关于“操作系统”的描述并不严谨，用户接触的并不是真正的“Ubuntu 操作系统”，而是它的“壳” (shell)。一般我们将操作系统核心部分称为内核 (kernel)，与其相对的是它最外层的“壳” (shell) 即为命令解释程序，是访问操作系统服务的用户界面。操作系统“壳”有命令行界面 (CLI) 或图形用户界面 (GUI) 两种形式。如 Windows 中的命令提示符 cmd 就是命令行界面，它是纯文本界面，用于接收、解释、执行用户输入的命令，完成相应的功能。

在 Ubuntu 中，命令行界面默认的 shell 是 bash，它也是一款基于 GNU 的免费、开源软件。

0.3 基础操作介绍

0.3.1 命令行

在命令行界面 (Command Line Interface, CLI) 中, 用户或客户端通过单条或连续命令行 (command line) 的形式向程序发出命令, 从而达到与计算机程序进行人机交互的目的。在 Linux 系统中, 命令用于对 Linux 系统进行管理, 其一般格式为: 命令名 [选项] [参数]。其中, 方括号表示可选, 意为可根据需要选用, 例如 `ls -a directory`。shell 命令有两种类型: shell 内建命令和外置程序。

下面介绍 Linux 的基本操作命令, 熟悉此部分内容的读者可跳过此节。

0.3.2 Linux 基本操作命令

进入终端后, 首先会看到光标前的如下内容。

```
1 开始连接到 git@xxx.xxx.xxx.xxx 0.1
2 Welcome to Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-46-generic x86_64)
3
4      * Documentation:  https://help.ubuntu.com
5      * Management:    https://landscape.canonical.com
6      * Support:        https://ubuntu.com/advantage
7 Last login: Wed Jan  4 12:50:58 2023 from 10.134.170.231
8 git@22xxxxxx:~$
```

其中 @ 符号前的是用户名, @ 符号后的是计算机名, 冒号后为当前所在的文件目录 (/ 表示根目录, ~ 表示家目录, 家目录即 /home/<user_name>), 最后 \$ 或 # 分别表示当前用户为普通用户或超级用户 root。然后通过键盘输入命令, 按回车后即可执行相应命令。

要想知道当前目录中包含哪些文件, 可使用 `ls` 命令, 其输出信息可以通过彩色加亮显示, 来区分不同类型的文件, `ls` 是使用率较高的命令, 其详细信息如图所示。一般情况下, 该命令的参数省略则默认显示该目录下所有文件, 所以只需使用 `ls` 即可看到所有非隐藏文件, 若要看到隐藏文件则需要加上 `-a` 选项, 若要查看文件的详细信息则需要加上 `-l` 选项。

```
1  ls
2  用法: ls [选项]... [文件]...
3  选项 (常用):
4  -a          不隐藏任何以 . 开始的项目 (在 Linux 中, 文件名以 . 开头的项目为隐藏项目)
5  -l          列出文件的详细信息并且每行只列出一个文件
```

针对该命令, 一般只会用到 `ls`, 以及带选项的三种形式: `ls -a`、`ls -l` 和 `ls -al`。现在尝试在命令行输入 `ls` 后, 按下回车会出现一个命名为学号的目录, 这就是读者进行实验的目录。如果想尝试创建一个新的空文件, 可以用 `touch` 命令。

```
1  touch
2  用法: touch [选项]... [文件名]...
```

输入 `touch hello_world.c`, 即可在该目录下成功创建一个新的文件, 再输入 `ls`, 就可看到 `hello_world.c` 文件了, 可以试一试 `ls -a` 和 `ls -l` 命令进行操作。关于如何编辑 `hello_world.c` 文件, 将在下一节关于 Vim 使用工具的介绍中展开。

为了通过目录对文件来进行组织和管理, 可以使用 `mkdir` 命令创建文件目录, 该命令的参数为创建的新目录的名称, 如 `mkdir newdir` 可创建一个名为 `newdir` 的目录。

```
1 mkdir
2 用法: mkdir [选项]... 目录...
```

现在输入 `mkdir newdir` 就在目录下创建了一个名为 `newdir` 的目录, 读者可以再使用 `ls` 命令查看是否新增了 `newdir` 目录。在这里可使用 `cd` 命令进入 `newdir` 目录。

```
1 cd
2 用法: cd [选项]... 目录
```

在命令行输入 `cd newdir` 命令, 进入 `newdir` 目录。为了返回上一级目录, 可以使用 `cd ..` 命令。在 Linux 系统里 `..` 表示上一级目录, `.` 表示当前目录, 因此在输入 `cd ..` 后, 返回上一级目录, 输入 `cd .` 进入当前所在的目录。

查看 `cd` 目录时, 命令行发生一些变化, 在命令行的 `>` 的左边, 从 `~` 变成了 `~/newdir`, 这个字符串是指当前所在的目录, `~` 表示所登录用户的家目录, 输入 `pwd` 可查看当前的绝对路径。同时 `cd` 命令也可直接把要跳转到的目录改为绝对路径, 如 `cd /home` 跳转到根目录下的 `home` 这个目录。

Note 0.3.1 在需要键入文件名或目录名时, 可以使用 `Tab` 键补足全名。当有多种补足时, 双击 `Tab` 键可以显示所有可能选项。在屏幕上输入 `cd /h` 然后按下 `Tab`, 就会自动补全为 `cd /home`, 如果输入的是 `cd /`, 再按两次 `Tab`, 会看到所有可选项, 和 `ls` 类似。

输入 `rmdir` 可以删除一个空的目录。

```
1 rmdir
2 用法: rmdir [选项]... 目录...
```

如果目录非空, 则使用 `rm` 命令。`rm` 命令可以删除一个目录中的一个或多个文件或目录, 也可以将某个目录及其下属的所有文件及其子目录均删除掉。对于链接文件, 只是删除整个链接文件, 而原有文件保持不变。

```
1 rm
2 用法: rm [选项]... 文件...
3 选项 (常用):
4 -r          递归删除目录及其内容, 如果不加这个命令, 删除一个有内容的目录
5             会提示不能删
6 -f          强制删除。忽略不存在的文件, 不提示确认
```

此外, `rm` 命令还有 `-i` 选项, 这个选项在使用文件扩展名字符删除多个文件时特别有用。使用这个选项, 系统会要求逐一确定是否要删除。这时, 必须输入 `y` 并按回车键, 才能删除文件。如果仅按回车键或其他字符, 文件不会被删除。与之相对应的就是 `-f` 选项, 其作用是强制删除文件或目录, 并不询问用户。使用该选项并配合 `-r` 选项, 可实现递归强制删除, 强制将指定目录下的所有文件与子目录一并删除, 这肯导致灾难性后果。例如 `rm -rf /` 即可强制递归删除全盘文件, 绝对不要轻易尝试!

Note 0.3.2 使用 `rm` 命令要格外小心，因为一旦删除了一个文件，就无法再恢复它。所以，在删除文件之前，最好再看一下文件的内容，确定是否真要删除。一些用户会在家目录下建一个类似回收站的目录，如果要使用 `rm` 命令，先把要删除的文件移到这个目录里，然后再进行 `rm`，一定时间之后再对回收站里的文件进行删除。

了解以上注意事项后，再来尝试使用 `rm`。读者先回到 `~` 目录下，假设要删除开始创建的 `hello_world.c`，首先在该目录下创建一个回收站目录叫 `.trash`，然后把 `hello_world.c` 拷贝到这个目录中，这里需要使用 `cp` 命令，该命令的第一个参数为源文件路径，命令的第二个参数为目标文件路径。

```
1 cp
2 用法: cp [选项]... 源文件... 目录
3 选项 (常用):
4 -r          递归复制目录及其子目录内的所有内容
```

下面我们介绍移动的命令。移动命令为 `mv`，与 `cp` 的用法相似。命令 `mv hello_world.c ~/.trash/` 就是将 `hello_world.c` 移动到 `~/.trash` 这个目录中。此时使用 `ls` 命令可以看到 `hello_world.c` 文件已被移除。

另外，在 Linux 系统中若要对文件进行重命名操作，使用 `mv oldname newname` 命令即可。

```
1 mv
2 用法: mv [选项]... 源文件... 目录
```

最后介绍回显命令 `echo`，如果输入 `echo hello_world`，就会回显 `hello_world`，这个命令看起来像一个复读机的功能，本书后文会介绍一些它的有趣用法。

以上就是 Linux 系统入门级的部分常用操作命令以及这些命令的常用选项，如果想要查看这些命令的其他功能选项或者新命令的详尽说明，可使用 Linux 下的帮助命令——`man` 命令，通过 `man` 命令可以查看 Linux 中的命令帮助、配置文件帮助和编程帮助等信息。

```
1 man - manual
2 用法: man page
3 e.g.
4 man ls
```

以下为几个常用的快捷键可供参考。

- `Ctrl+C` 终止当前程序的执行
- `Ctrl+Z` 挂起当前程序
- `Ctrl+D` 终止输入（若正在使用 `shell`，则退出当前 `shell`）
- `Ctrl+L` 清屏

其中，如果你写了一个死循环，或者程序执行到一半你不想让它执行了，`Ctrl+C` 是你的很好的选择。`Ctrl+Z` 挂起程序后会显示该程序挂起编号，若想要恢复该程序可以使

用 `fg [job_spec]`, `job_spec` 即为挂起编号, 不输入时默认为最近挂起进程。而如果你写了一个等到识别到 `EOF` 才停止的程序, 你就需要输入 `Ctrl+D` 来当作输入了一个 `EOF`。

对其他内容感兴趣的同学可以自行网络搜索或用 `man` 命令看帮助手册进行学习和了解。以上述内容为基础，接下来讲如何在 Linux 下写代码。

Note 0.3.3 在多数 shell 中，四个方向键也是有各自特定的功能的：← 和 → 可以控制光标的位置，↑ 和 ↓ 可以切换最近使用过的命令

0.4 实用工具介绍

学会了 Linux 基本操作命令，接下来就可以得心应手地使用命令行界面的 Linux 操作系统了。想要使用 Linux 系统完成工作，仅靠命令行还远远不够。在开始动手阅读并修改代码之前，读者还需要掌握一些实用工具的使用方法。这里首先介绍一种常用的文本编辑器：Vim。

0.4.1 Vim

Vim 被誉为编辑器之神，是程序员为程序员设计的编辑器，编辑效率高，十分适合编辑代码。对于习惯了图形化界面文本编辑软件的读者来说，刚接触 Vim 时一定会觉得非常不习惯非常不顺手，所以下面通过创建一个 `helloworld.c`，来让读者熟悉一下 Vim 的基本操作：

- (1) 创建文件：利用之前学到的 `touch` 命令创建 `helloworld.c` 文件（这时，使用 `ls` 命令，可以发现已经在当前目录下创建了 `helloworld.c` 文件）。
- (2) 打开文件：在命令行界面输入 `vim helloworld.c` 打开新建的文件；
- (3) 输入内容：刚打开文件时，无法向其中直接输入代码，需要按“i”键进入插入模式（在底栏出现 `-- INSERT --` 表示已进入插入模式），之后便可以向其中输入 `helloworld` 程序，如下图所示：

```
#include <stdio.h>
int main() {
    printf("Hello World!\n");
    return 0;
}
```

图 0.2: 写入 `helloworld.c` 中的内容

(4) 保存并回到命令行界面：完成文件修改后，按“Esc”回到命令模式，再按下“:”进入底线命令模式，此时可以看到屏幕的左下角出现了一个冒号，输入“w” (write)，并按下回车，文件便得到了保存；再进入底线命令模式，输入“q” (quit) 便可以关闭文件，回到命令行界面（保存和退出的命令，可以如上述分两步完成，也可以由“:wq”一条命令完成，如图）。

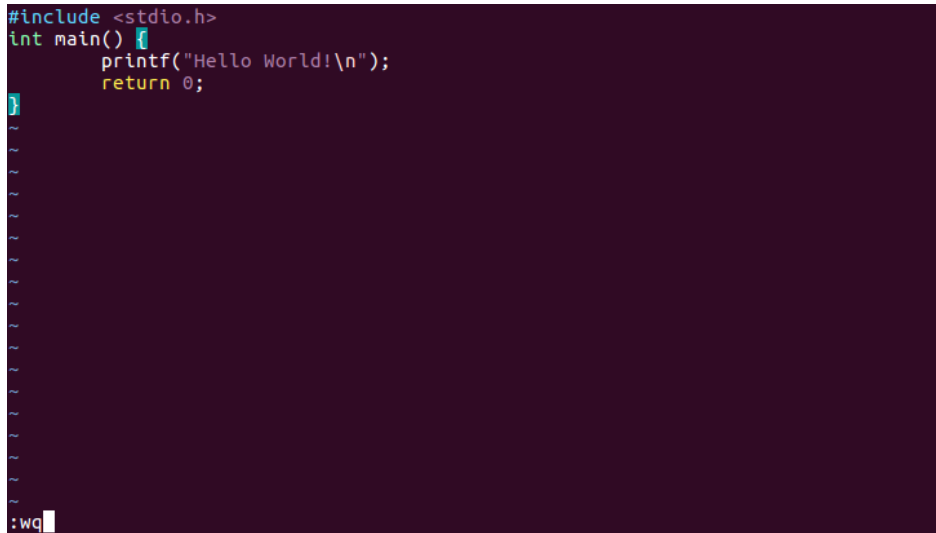


图 0.3: 保存并退出

接着简单了解一下对 Vim 进行一些自定义配置的方法：

```
1 # 首先按如下方法打开（如果原本没有则是新建） ~/.vimrc 文件
2 vim ~/.vimrc
```

在这个文件中写入如下内容：

```
1 set cursorline
```

保存并退出文件后，我们便完成了对 Vim 的配置（上述配置能够让光标所在行下加下划线）。想要了解更多配置参数的同学可以自行在网上搜索。

相关的内容可在网上查询，随用随查即可。

0.4.2 GCC

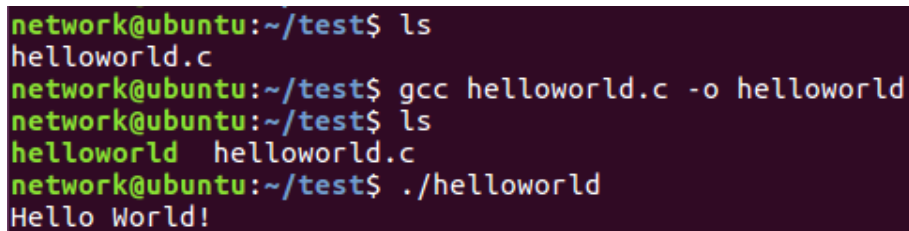
GCC (GNU Compiler Collection, GNU 编译器套件) 包含了著名的 C 语言编译器 gcc (GNU project C and C++ compiler)。我们的实验课程使用 gcc 作为 C 语言编译器。其常用的使用方法如下图所示，可以自己动手实践，写一些简单的 C 代码来编译运行。如果想要同时编译多个文件，可以直接用 -o 选项将多个文件进行编译连接：gcc testfun.c test.c -o test，也可以先使用 -c 选项将每个文件单独编译成 .o 文件，再用 -o 选项将多个 .o 文件进行链接：gcc -c testfun.c && gcc -c test.c && gcc testfun.o test.o -o test，两者等价。

```
1  语法: gcc [选项]... [参数]...
2  选项 (常用):
3  -o          指定生成的输出文件
4  -S          将 C 代码转换为汇编代码
5  -Wall       显示一些警告信息
6  -c          仅执行编译操作, 不进行链接操作
7  -M          列出依赖
8  -I<path>    编译时指定头文件目录, 使用标准库时不需要指定目录, -I 参数可以用相对
    ↪ 路径, 比如头文件在当前目录, 可以用-I. 来指定
9  参数:
10 C 源文件: 指定 C 语言源代码文件
11 e.g.
12
13 $ gcc test.c -o test
14 # 使用 -o 选项生成名为 test 的可执行文件
```

下面, 我们通过编译之前完成的 `helloworld.c` 来熟悉 GCC 的最基本使用方法:

(1) 在命令行中, 使用 `gcc helloworld.c -o helloworld` 命令, 便可以创建由 `helloworld.c` 文件编译成的 `helloworld` 的可执行文件 (使用 `ls` 可以看到目录内出现了 `helloworld` 可执行文件)。

(2) 在命令行中, 输入 `./helloworld` 运行可执行文件, 观察现象。



```
network@ubuntu:~/test$ ls
helloworld.c
network@ubuntu:~/test$ gcc helloworld.c -o helloworld
network@ubuntu:~/test$ ls
helloworld helloworld.c
network@ubuntu:~/test$ ./helloworld
Hello World!
```

图 0.4: GCC 编译可执行文件并运行

0.4.3 Makefile

如果想了解操作系统这样的大型软件, 首先面临一个很大的问题: 这些代码应当从那里开始阅读? 答案是 Makefile。当你不知所措的时候, 从 Makefile 开始往往会是一个不错的选择。那么 `make` 是什么, 而 Makefile 又是什么呢? `make` 工具一般用于维护软件开发的工程项目, 它可以根据时间戳自动判断项目的哪些部分需要重新编译, 每次只重编译必要的部分。`make` 工具会读取 Makefile 文件, 并根据 Makefile 的内容来执行相应的编译操作。

相较于手动编译而言, Makefile 具有更高的便捷性, 可以方便地管理大型项目。而且理论上 Makefile 支持任意语言, 只要其编译器可以通过 `shell` 命令来调用即可。如果一个项目有着复杂的构建流程, Makefile 便能充分展现其优势。为了更为清晰地介绍 Makefile 的基本概念, 下面通过编制一个简单的 Makefile 来说明。假设我们手头有一个 Hello World 程序需要编译。如果我们没有 Makefile, 则需动手编译这个程序, 执行以下命令:

```
1 # 直接使用 gcc 编译 Hello World 程序
2 $ gcc -o hello_world hello_world.c
```

那么, 如果想把它写成 Makefile, 该如何操作呢? Makefile 最基本的格式如下:

```
1 target: dependencies
2     command 1
3     command 2
4     ...
5     command n
```

其中, **target** 是构建 (build) 的目标, 可以是目标文件、可执行文件, 也可以是一个标签。而 **dependencies** 是构建该目标所需的其他文件或其他目标。之后是构建该目标所需执行的命令。有一点尤为需要注意, 每一个命令 (command) 之前必须用一个制表符 (Tab) 缩进。这里必须使用制表符而不能是空格, 否则 **make** 会报错。

通过在 Makefile 中书写显式规则来告诉 **make** 工具文件间的依赖关系: 如果想要构建 **target**, 那么首先要准备好 **dependencies**, 接着执行 **command** 中的命令, 最终完成构建 **target**。在编写完恰当的规则之后即在 shell 中输入 **make target** (**target** 是目标名), 即可执行相应的命令、生成相应的目标。

前面提到, **make** 工具根据时间戳来判断是否需要编译, **make** 只有依赖文件中存在文件的修改时间比目标文件的修改时间晚时 (也就是对依赖文件做了改动), shell 命令才会被执行, 编译生成新的目标文件。

简易 Makefile 内容如下, 之后执行 **make all** 或是 **make** 命令, 即可产生 **hello_world** 可执行文件。

```
1 all: hello_world.c
2     gcc -o hello_world hello_world.c
```

下面, 我们尝试编写一个简单的 Makefile 文件:

(1) 在命令行中, 创建名为 "Makefile" 的文件, 使用 Vim 打开它, 并写入如下内容:

```
1 all: hello_world.c
2     gcc -o hello_world hello_world.c
3 clean:
4     rm -f helloworld
```

其中, 前两行定义了编译 **helloworld.c** 的命令 ("all" 为 **target**, "helloworld.c" 为 **dependencies**, 第二行为编译 **helloworld** 程序的命令); "clean" 后的部分为删除可执行文件的命令。

(2) 保存并回到命令行界面后, 输入 **make clean**, 执行 Makefile 文件中的 **rm helloworld** 命令, 删除了之前编译出的可执行文件; 接着, 输入 **make all**, 执行 Makefile 文件中的 **gcc helloworld.c -o helloworld** 命令, 编译出可执行文件。过程如下图:

```
network@ubuntu:~/test$ make clean
rm helloworld
network@ubuntu:~/test$ ls
helloworld.c  Makefile
network@ubuntu:~/test$ make all
gcc helloworld.c -o helloworld
network@ubuntu:~/test$ ls
helloworld  helloworld.c  Makefile
network@ubuntu:~/test$ ./helloworld
Hello World!
```

图 0.5: make 命令执行过程和效果

在 lab0 阶段，建议自己尝试编写一个简单的 Makefile 文件，体会 make 工具的使用方法。

0.4.4 ctags

ctags 是一个方便 Vim 下代码阅读的工具。这里只进行一些最基础功能的介绍：(1) 为了能够跨目录使用 ctags，我们需要添加 Vim 的相关配置，按照上文所述的方法打开 `.vimrc` 文件，添加以下内容并保存：

```
1 set tags=tags;
2 set autochdir
```

(2) 为了进行测试，我们修改 `helloworld.c` 文件（在其中添加一个函数），并新建一个文件 `ctags test.c`，在其中调用 `helloworld.c` 中新建的函数，如下图：

[illegible]

图 0.6: helloworld.c 文件中的内容

[illegible]

图 0.7: ctags_test.c 文件中的内容

(3) 我们回到命令行界面，执行命令 `ctags -R *`，会发现在该目录下出现了新的文件 `tags`，接下来就可以使用一些 `ctags` 的功能了：

使用 Vim 打开 `ctags_test.c`，将光标移到调用的函数 (`test_ctags`) 上，按下 `Ctrl+]`，便可以跳转到 `helloworld.c` 中的函数定义处；再按下 `Ctrl+T` 或 `Ctrl+O`（有些浏览器 `Ctrl+T` 是新建页面，会出现热键冲突），便可以回到跳转前的位置。

```
#include "helloworld.c"

int main() {
    test_ctags();
    return 0;
}
```

图 0.8: 光标处于函数名上

使用 Vim 打开 `ctags_test.c`, 按 `:` 进入底线命令模式, 再输入 `tag test_ctags`, 也可以跳转到该函数定义的位置。

正式开始操作系统实验后，需要阅读和理解的代码量会增加很多，不同文件之间的函数调用会给阅读代码带来很大的阻力。熟练运用 `ctags` 的相关功能，可以为读者阅读、理解代码提供很大的帮助。

0.5 Git 专栏-轻松维护和提交代码

本书设计的实验通过 Git 版本控制系统进行管理，在这里就来了解一下 Git 相关内容。

0.5.1 Git 是什么?

最初的版本控制是纯手工完成的：修改文件，保存文件副本。如果保存副本时命名比较随意，时间长了就不知道哪个是新的，哪个是旧的了，即使知道新旧，可能也不知道每个版本是什么内容，相对上一版做了哪些修改，当几个版本过去后，很可能就像图 0.9 一样糟糕了。

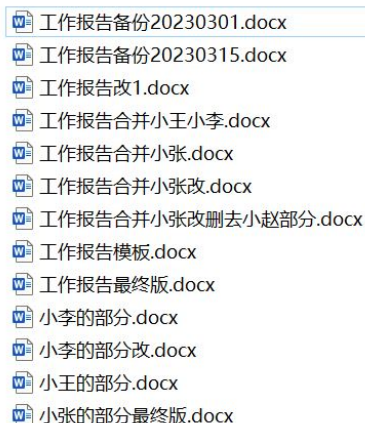


图 0.9: 手工版本控制

在很多情况下，工程项目也往往由多人一起完成的，在项目刚刚开始时，分工，制定计划，埋头苦干，但版本管理可能会让人头疼不已。其本质原因在于每个人都会对目的内容进行改动，结果就可能是 A 把添加完功能的项目打包发给了 B，然后自己继续添加功能。一天后，B 把他修改后的项目包又发给了 A，这时 A 就必须非常清楚发给 B 之后到他发回来的这段时间，自己究竟对哪些地方做了改动，然后还要进行合并，相当困难。

这时会面临一个无法避免的事实：如果每一次小小的改动，开发者之间都要相互通知，那么一些错误的改动将会令我们付出很大的代价：一个错误的改动通知几方同时纠正。如果一次性做了大幅度的修改，那么只有在概览项目的很多文件后才能知道改动在哪里，也才能做合并修改。

由此产生了需求，大家希望：

- 自动帮助记录每次文件的改动，而且最好是有撤回功能，改错了可以轻松撤销。
- 支持多人协作编辑，命令与操作简洁。
- 能够在不同的历史版本中切换。
- 可方便地在软件中查看某次改动。

版本控制系统就是能够解决上述需求的一种系统，而 Git 则是一种先进的分布式版本控制系统。

Git 是由 Linux 的创始人林纳斯·托瓦尔兹 (Linus Torvalds) 创造，最初用于管理自己的 Linux 开发过程。他对于 Git 的解释是：The stupid content tracker (傻瓜内容追踪器)。

Note 0.5.1 版本控制是一种记录若干文件内容变化,以便将来查阅特定版本修订情况的系统。

0.5.2 Git 基础指引

通过前几节的学习,完成如下操作:

1. 回到主目录 `~`, 创建一个名为 `learnGit` 的目录
2. 进入 `learnGit` 目录
3. 输入 `git init`
4. 用 `ls` 命令添加适当参数看看多了什么

可以发现,新建的目录下面多了一个名为 `.git` 的隐藏目录,这个目录就是 Git 版本库,常被称为仓库 (repository)。需要注意的是,实验中不会对 `.git` 目录进行任何直接操作,不要轻易对该目录做任何操作。

`git init` 执行后就拥有了一个仓库。建立的 `learnGit` 目录就是 Git 里的工作区。目前除了 `.git` 版本库目录以外空无一物。

Note 0.5.2 在我们的 MOS 操作系统实验中我们不需要使用到 `git init` 命令,每个人一开始就都有一个名为 `22xxxxxx` (你的学号) 的版本库。

目前,在工作区 `learnGit` 中仅有 Git 版本库,下面新建一个文件 `readme.txt`,内容为“BUAA_OSLAB”。执行以下命令得到该文件添加至版本库:

```
1 $ git add readme.txt
```

注意,执行此命令后,并未真正地把 `readme.txt` 提交到版本库,Git 同其他大多数版本控制系统一样,需要 `add` 之后再执行一次提交操作,提交操作的命令如下:

```
1 $ git commit
```

如果不带任何附加选项,执行后会弹出一个说明窗口,如下所示。我们使用此前学到的 Vim 编辑操作在该说明窗口的最后添加一行说明“**Notes to test**”即为本次提交所附加的说明。

```
1 # 请为您的变更输入提交说明。以 '#' 开始的行将被忽略, 而一个空的提交
2 # 说明将会终止提交。
3 #
4 # 位于分支 master
5 # 您的分支与上游分支 'origin/master' 一致。
6 #
7 # 要提交的变更:
8 #   修改:      readme.txt
9 #
10 Notes to test
```

注意,弹出的窗口中我们**必须**得添加本次 `commit` 的说明,这意味着我们不能提交空白说明,否则我们的提交不会成功。在添加说明之后,使用 Vim 保存退出该说明即可成功提交。

Note 0.5.3 初学者一般不重视 `git commit` 内容的有效性，总是使用说明意义不明的字符串作为说明提交。但以后可能就会发现写一个自己看得懂，别人也能看得懂的提交说明是多么必要。所以为了提高可读性，尽量每次提交都能见名知义，比如“fixed a bug in ...”这样的描述，推荐一条命令：`git commit --amend` 重新书写最后一次提交的说明。

以窗口提交方式是一种更简洁的方式，使用以下命令：

```
1 $ git commit -m [comments]
```

[comments] 格式为“评论内容”，上述的提交过程我们可以简化为下面一条命令

```
1 $ git commit -m "Notes to test"
```

如果提交之后看到类似的提示，就说明提交成功了。

```
1 [master 955db52] Notes to test.  
2 1 file changed, 1 insertion(+), 1 deletion(-)
```

本次提交中可以得到以下提示信息中，后续会详细说明提交提示信息含义。

- 本次提交的分支是 master
- 本次提交的 ID 是 955db52
- 提交说明是 Notes to test
- 共有 1 个文件相比之前发生了变化：1 行的添加与 1 行的删除行为

在实验过程中，提交后可能会出现如下提示信息，说明这是要求设置提交者身份。

```
1 *** Please tell me who you are.  
2  
3 Run  
4  
5 git config --global user.email "you@example.com"  
6 git config --global user.name "Your Name"  
7  
8 # to set your account's default identity.  
9 # Omit --global to set the identity only in this repository.
```

Note 0.5.4 可以用以下两条命令设置用户名和邮箱：

```
git config --global user.email "you@example.com"  
git config --global user.name "Your Name"  
示例：  
git config --global user.email "qianlxc@126.com"  
git config --global user.name "Qian"
```

现在已设置了提交者的信息，提交者信息是为了告知所有负责该项目的人每次提交是由谁提交的，并提供联系方式以进行交流。

0.5.3 Git 文件状态

首先对于 Git 管理的任何一个文件,在本地仓库中都只有四种状态:未跟踪 (untracked)、未修改 (unmodified)、已修改 (modified)、已暂存 (staged):

未跟踪 表示没有跟踪 (add) 某个文件的变化,使用 `git add` 即可跟踪文件。

未修改 表示某文件在跟踪后一直没有改动过或者改动已经被提交。

已修改 表示修改了某个文件,但还没有加入 (add) 到暂存区中。

已暂存 表示把已修改的文件放在下次提交 (commit) 时要保存的清单中。

这里使用一张图来说明文件的四种状态的转换关系:

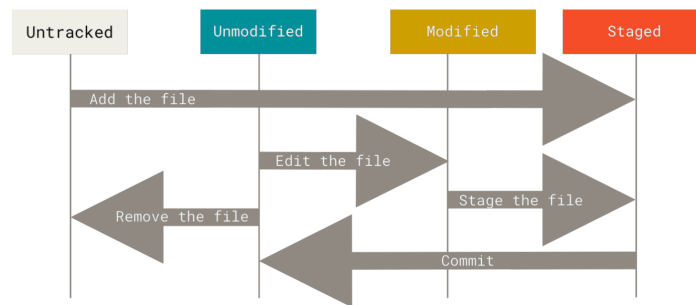


图 0.10: Git 中的四种状态转换关系

完成以下练习以进一步熟悉 Git 的使用方法。

Thinking 0.1 思考下列有关 Git 的问题:

- 在前述已初始化的 `~/learnGit` 目录下,创建一个名为 `README.txt` 的文件。执行命令 `git status > Untracked.txt` (其中的 `>` 为输出重定向,我们将在 0.6.3 中详细介绍)。
- 在 `README.txt` 文件中添加任意文件内容,然后使用 `add` 命令,再执行命令 `git status > Stage.txt`。
- 提交 `README.txt`,并在提交说明里写入自己的学号。
- 执行命令 `cat Untracked.txt` 和 `cat Stage.txt`,对比两次运行的结果,体会 `README.txt` 两次所处位置的不同。
- 修改 `README.txt` 文件,再执行命令 `git status > Modified.txt`。
- 执行命令 `cat Modified.txt`,观察其结果和第一次执行 `add` 命令之前的 `status` 是否一样,并思考原因。

Note 0.5.5 `git status` 是一个查看当前文件状态的有效命令，而 `git log` 则是提交日志，每提交一次，Git 会在提交日志中记录一次。`git log` 将在版本切换时发挥很大的作用。

实施前述实验后，`Untracked.txt`，`Stage.txt` 和 `Modified.txt` 的内容如下。

```

1  # Untracked.txt 的内容如下
2
3  On branch master
4  Untracked files:
5    (use "git add <file>..." to include in what will be committed)
6
7    README.txt
8
9  nothing added to commit but untracked files present (use "git add" to track)
10
11
12 # Stage.txt 的内容如下
13
14 On branch master
15 Changes to be committed:
16   (use "git reset HEAD <file>..." to unstage)
17
18   new file:   README.txt
19
20
21 # Modified.txt 的内容如下
22
23 On branch master
24 Changes not staged for commit:
25   (use "git add <file>..." to update what will be committed)
26   (use "git checkout -- <file>..." to discard changes in working directory)
27
28   modified:   README.txt
29
30 no changes added to commit (use "git add" and/or "git commit -a")

```

通过仔细观察，我们看到第一个文本文件 `Untracked.txt` 中第 2 行是：Untracked files，而第二个文本文件 `Stage.txt` 中第二行内容是：Changes to be committed，而第三个文件 `Modified.txt` 中则是 Changes not staged for commit。

这三种不同的提示分别意味着：在 `README.txt` 新建的时候，其处于为未跟踪状态 (untracked)；在 `README.txt` 中任意添加内容，接着用 `add` 命令之后，文件处于暂存状态 (staged)；在修改 `README.txt` 之后，其处于被修改状态 (modified)。

Note 0.5.6 关于 [思考-Git 的使用 1](#)，实际上是因为 `git add` 命令本身是有多义性的，虽然差别较小但是不同情境下使用依然是有区别。因此需注意：新建文件后要 `git add`，修改文件后也需要 `git add`。

Thinking 0.2 仔细看看[0.10](#)，思考一下箭头中的 `add the file`、`stage the file` 和 `commit` 分别对应的是 Git 里的哪些命令呢？ ■

此时相信读者对 Git 的设计有了初步的认识。下一步就来深入理解一下 Git 里的一些机制。

0.5.4 Git 三棵树

本地仓库由 Git 维护的三棵“树”组成。第一个是工作区，它持有实际文件；第二个是暂存区（Index 有时也称 Stage），它像个暂时存放的区域，临时保存你的改动；最后是 HEAD，指向你最近一次提交后的结果。

Git 的对象库位于 `.git/objects` 中，所有需要实施版本控制的文件会被压缩成二进制文件，压缩后的二进制文件成为一个 Git 对象，保存在 `.git/objects` 目录。Git 计算当前文件内容的哈希值（长度为 40 的字符串），并作为该对象的文件名。

在 `.git` 目录中，文件 `.git/index` 实际上就是一个包含文件索引的目录树，像是一个虚拟的工作区。在这个虚拟工作区的目录树中，记录了文件名、文件的状态信息（时间戳、文件长度等），但是文件的内容并不存储在其中，而是保存在 Git 对象库（`.git/objects`）中，文件索引建立了文件和对象库中对象实体之间的对应。下图展示了工作区、版本库中的暂存区和版本库之间的关系，揭示了不同操作所带来的不同影响。

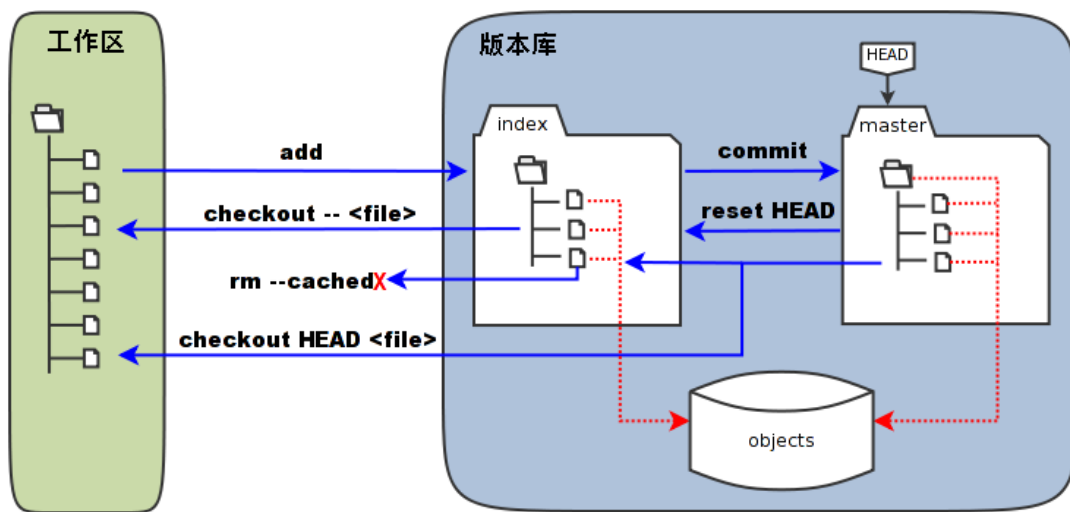


图 0.11: 工作区、暂存区和版本库

- 图中 `objects` 标识的区域为 Git 的对象库，实际位于 `.git/objects` 目录下。
- 图中左侧为工作区，右侧为版本库。在版本库中标记为“index”的区域是暂存区 (stage, index)，标记为 `master` 的是 `master` 分支所代表的目录树。
- 图中我们可以看出此时 `HEAD` 实际是指向 `master` 分支的一个“指针”。所以图示的命令中出现 `HEAD` 的地方可以用 `master` 来替换。
- 当对工作区修改（或新增）的文件执行 `git add` 命令时，暂存区的目录树被更新，同时工作区修改（或新增）的文件内容被写入到对象库中的一个新的对象中，而该

对象的 ID 被记录在暂存区的文件索引中。

- 当执行提交操作 (`git commit`) 时, 会将暂存区的目录树写到版本库 (对象库) 中, `master` 分支会做相应的更新。即 `HEAD` 指向的目录树就是提交时暂存区的目录树。
- 当执行 `git rm --cached <file>` 命令时, 会直接从暂存区删除文件, 工作区则不做出改变。
- 当执行 `git reset HEAD` 命令时, 暂存区的目录树会被重写, 由 `master` 分支指向的目录树所替换, 但是工作区不受影响。
- 当执行 `git checkout -- <file>` 命令时, 会用暂存区指定的文件替换工作区的文件。这个操作很危险, 会清除工作区中未添加到暂存区的改动。
- 当执行 `git checkout HEAD <file>` 命令时, 会用 `HEAD` 指向的 `master` 分支中的指定文件来替换暂存区和以及工作区中的文件。这个命令也是极具危险性的, 因为不但会清除工作区中未提交的改动, 也会清除暂存区中未提交的改动。

在 Git 中引入**暂存区**的概念是 Git 里较难理解却是最有亮点的设计之一, 在这里不再详细介绍其能快速快照与回滚原理。有兴趣的同学不妨去看看 [Pro Git](#) 这本书。

0.5.5 Git 版本回退

在编写代码时, 可能遇到过因错误地删除文件或因一个修改导致程序再也无法运行等情况。Git 允许进行版本回退。首先, 有必要学习一些撤销命令:

`git rm --cached <file>` 这条命令是指从暂存区中删除不再想跟踪的文件, 比如调试用的文件等。

`git checkout -- <file>` 丢弃工作区中的修改, 把暂存区的文件恢复到工作区中。如果在工作区中对多个文件经过多次修改后, 发现编译无法通过了。如果尚未执行 `git add`, 则可使用本命令将工作区恢复成原来的样子。

`git reset HEAD <file>` 撤销暂存区的修改, 把暂存区恢复到执行 `git add` 之前的状态。如果对工作区的修改已经执行了 `git add`, 想要撤销修改, 可以使用本命令, 再对需要恢复的文件使用上一条命令即可。

`git clean <file> -f` 如果你的工作区混入了未知内容, 你没有追踪它, 但是想清除它的话就可以使用本命令, 它可以帮你把未知内容剔除出去。

`git restore <file>` 作用与 `git checkout -- <file>` 相同, 在使用 `git status` 时通常会提示使用该指令来撤销对工作区的修改。

Thinking 0.3 思考下列问题:

1. 代码文件 `print.c` 被错误删除时, 应当使用什么命令将其恢复?
2. 代码文件 `print.c` 被错误删除后, 执行了 `git rm print.c` 命令, 此时应当使用什么命令将其恢复?
3. 无关文件 `hello.txt` 已经被添加到暂存区时, 如何在不删除此文件的前提下将其移出暂存区?

关于上面那些撤销命令, 可在你不慎删除不应删除内容时再行查阅, 当然更推荐使用 `git status` 来看当前状态下 Git 的推荐命令。现阶段主要掌握 `git add` 和 `git commit` 的用法。当然, 一定要慎用撤销命令。否则撤销之后如何撤除撤销命令将是一件难事。

下面介绍 `git reset` 的更多用法。

```
1 git reset --hard
```

为了体会 `reset` 命令的作用, 下面先做一个小练习:

Thinking 0.4 思考下列有关 Git 的问题:

- 找到在 `/home/22xxxxxx/learnGit` 下刚刚创建的 `README.txt` 文件, 若不存在则新建该文件。
- 在文件里加入 `Testing 1`, `git add`, `git commit`, 提交说明记为 1。
- 模仿上述做法, 把 1 分别改为 2 和 3, 再提交两次。
- 使用 `git log` 命令查看提交日志, 看是否已经有三次提交, 记下提交说明为 3 的哈希值^a。
- 进行版本回退。执行命令 `git reset --hard HEAD^` 后, 再执行 `git log`, 观察其变化。
- 找到提交说明为 1 的哈希值, 执行命令 `git reset --hard <hash>` 后, 再执行 `git log`, 观察其变化。
- 现在已经回到了旧版本, 为了再次回到新版本, 执行 `git reset --hard <hash>`, 再执行 `git log`, 观察其变化。

^a使用 `git log` 命令时, 在 `commit` 标识符后的一长串数字和字母组成的字符串

使用这条命令可以进行版本回退或者切换到任何一个版本。它有两种用法: 第一种是使用 `HEAD` 类似形式, 如果想退回上个版本就用 `HEAD^`, 上上个版本的话就用

HEAD^^，要是回退到前 50 个版本则可使用 HEAD~50 来代替；第二种就是使用 hash 值，使用 hash 值可以在不同版本之间任意切换，足见 hash 值的强大。

必须注意，`--hard` 是 `git reset` 命令唯一的危险用法，它也是 Git 会真正地销毁数据的几个操作之一。其他任何形式的 `git reset` 调用都可以轻松撤销，但是 `--hard` 选项不能，因为它强制覆盖了工作目录中的文件。若该文件还未提交，Git 会覆盖它从而导致无法恢复。

0.5.6 Git 分支

如果你还有印象的话，我们之前提到过分支这个概念，那么分支是个什么东西呢？分支就是科幻电影里面的平行宇宙，不同的分支间不会互相影响。或许当你正在电脑前努力学习操作系统的时候，另一个你正在另一个平行宇宙里努力学习面向对象。使用分支意味着你可以从开发主线上分离开来，然后在不影响主线的时候继续工作。在我们实验中也会多次使用到分支的概念。首先我们来讲一条创建分支的命令。

```
1 # 创建一个基于当前分支产生的分支，其名字为 <branch-name>
2 $ git branch <branch-name>
```

这条命令将会在实验课考试进行的时候用到。其功能相当于把当前分支的内容拷贝一份到新的分支里去，然后我们在新的分支上做测试功能的添加即可，不会影响实验分支的效果等。假如我们当前在 master¹分支下已经有过三次提交记录，这时我们使用 `git branch` 命令新建了一个分支为 `testing`（参考图 0.12）。

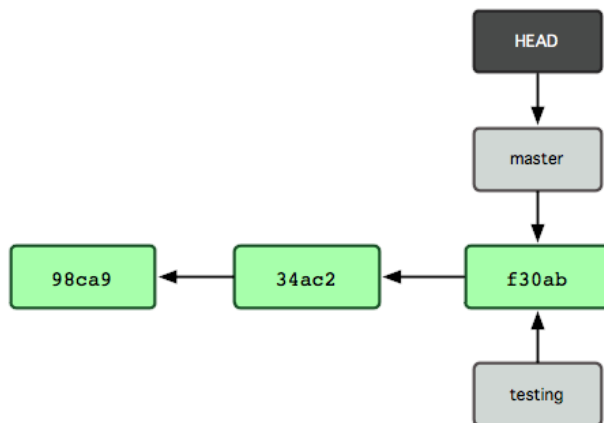


图 0.12: 分支建立后

删除一个分支也很简单，只要加上 `-d` 选项（`-D` 是强制删除）即可，就像这样

```
1 # 强制删除一个指定的分支
2 $ git branch -D <branch-name>
```

想查看所有的远程分支和本地分支，只需要加上 `-a` 选项即可

¹master 分支是我们的主分支，一个仓库初始化时自动建立的默认分支


```

1  # 查看所有的远程与本地的所有分支
2  $ git branch -a
3
4  # 使用该命令的效果如下
5  # 前面带 * 的分支是当前分支
6  lab1
7  lab1-exam
8  * lab1-result
9  master
10 remotes/origin/HEAD -> origin/master
11 remotes/origin/lab1
12 remotes/origin/lab1-exam
13 remotes/origin/lab1-result
14 remotes/origin/master
15 # 带 remotes 是远程分支，在后面提到远程仓库的时候我们会知道

```

我们建立了分支并不代表会自动切换到分支，那么，Git 是如何知道你当前在哪个分支上工作的呢？其实答案也很简单，它保存着一个名为 HEAD 的特别指针。在 Git 中，它是一个指向你正在工作中的本地分支的指针，可以将 HEAD 想象为当前分支的别名。运行 `git branch` 命令，仅仅是建立了一个新的分支，但不会自动切换到这个分支中去，所以在这个例子中，我们依然还在 master 分支里工作。

那么我们如何切换到另一个分支去呢，这时候我们就要用到这个我们在实验中更常见的分支命令了

```

1  # 切换到 <branch-name> 代表的分支，这时候 HEAD 游标指向新的分支
2  $ git checkout <branch-name>

```

比如这时候我们使用 `git checkout testing`，这样 HEAD 就指向了 testing 分支（见图0.13）。

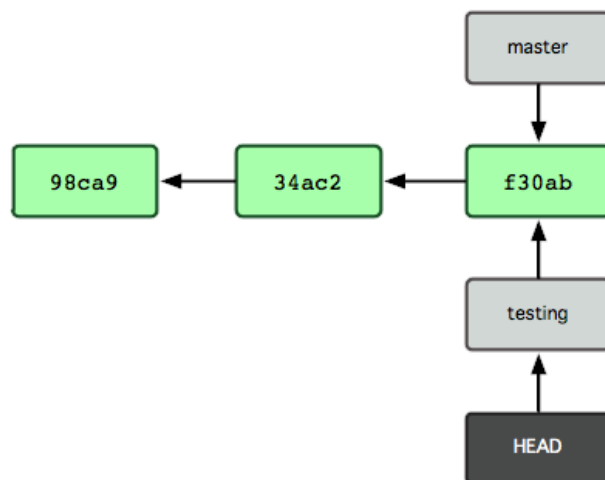


图 0.13: 分支切换后

这时候你会发现你的工作区就是 testing 分支下的工作目录，而且在 testing 分支下的修改，添加与提交不会对 master 分支产生任何影响。

我们之前所介绍的这些命令只是在本地进行操作的，其中必须掌握

1. git add

2. `git commit`
3. `git branch`
4. `git checkout`

其余命令可以临时查阅，当然掌握这些 Git 操作的益处你现在也许体会不出来，但当你们小团队哪天一起做项目的时候，你就会体会到掌握这么多 Git 的知识是件多么幸福的事情了。之前我们所有的操作都是在本地版本库上操作的，下面我们要介绍的是一组和远程仓库有关的命令。这组命令是最容易出错的，所以一定要认真学习。

0.5.7 Git 远程仓库与本地

下面介绍两条跟远程仓库有关的命令，其作用很简单，但要用好却是比较难。

```
1 # git push 用于从本地版本库推送到服务器远程仓库
2 $ git push
3
4 # git pull 用于从服务器远程仓库抓取到本地版本库
5 $ git pull
```

`git push` 只是将本地版本库里已经 `commit` 的部分同步到服务器上去，不包括暂存区里存放的内容。在我们实验中除了还可能会加些选项使用

```
1 # origin 在我们实验里是固定的，以后就明白了。branch 是指本地分支的名称。
2 $ git push origin [branch]
```

这条命令可以将我们本地创建的分支推送到远程仓库中，在远程仓库建立一个同名的本地追踪的远程分支。

`git pull` 是条更新用的命令，如果助教老师在服务器端发布了新的分支，下发了新的代码或者进行了一些改动的话，我们就需要使用 `git pull` 来让本地版本库与远程仓库保持同步。

如果你还想进一步学习 Git 的知识，可以查看教程²和游玩 [GitHub](#)

0.6 进阶操作

0.6.1 Linux 操作补充

首先，是两种常用的查找命令：`find` 和 `grep`

使用 `find` 命令并加上 `-name` 选项可以在当前目录下递归地查找符合参数所示文件名的文件，并将文件的路径输出至屏幕上。

```
1 find - search for files in a directory hierarchy
2 用法: find -name 文件名
```

²推荐廖雪峰老师的网站：<http://www.liaoxuefeng.com/>

grep 是一种强大的文本搜索工具，它能使用正则表达式搜索文本，并把匹配的行打印出来。简单来说，**grep** 命令可以从文件中查找包含 **pattern** 部分字符串的行，并将该文件的路径和该行输出至屏幕。当你需要在整个项目目录中查找某个函数名、变量名等特定文本的时候，**grep** 将是你手头一个强有力的工具。

```
1  grep - print lines matching a pattern
2  用法: grep [选项]... PATTERN [FILE]...
3  选项 (常用):
4  -a          不忽略二进制数据进行搜索
5  -i          忽略文件大小写差异
6  -r          从目录递归查找
7  -n          显示行号
```

tree 命令可以根据文件目录生成文件树，作用类似于 **ls**。

```
1  tree
2  用法: tree [选项] [目录名]
3  选项 (常用):
4  -a          列出全部文件
5  -d          只列出目录
```

Linux 的文件调用权限分为三级: 文件拥有者、群组、其他。利用 **chmod** 可以藉以控制文件如何被他人所调用。

```
1  chmod
2  用法: chmod 权限设定字符串 文件...
3  权限设定字符串格式:
4  [ugoa...][[+|=] [rwxX]...][,...]
```

其中: **u** 表示该文件的拥有者, **g** 表示与该文件的拥有者属于同一个群组, **o** 表示其他以外的人, **a** 表示这三者皆是。+ 表示增加权限、- 表示取消权限、= 表示唯一设定权限。**r** 表示可读取, **w** 表示可写入, **x** 表示可执行, **X** 表示只有当该文件是个子目录或者该文件已经被设定过为可执行。

此外 **chmod** 也可以用数字来表示权限, 格式为:

```
1  chmod ugo 文件
```

ugo 分别为一个三位的二进制数在十进制下的数值, 这三个数字分别表示拥有者, 群组, 其他人的权限。三位二进制从高位到低位分别表示 **rwX** 的权限是否打开。

下面是一些 **chmod** 的使用示例:

```
1  # 为 run.sh 的拥有者添加执行权限
2  chmod u+x run.sh
3
4  # 修改 run.sh 的权限为 rwxr-xr-x (对应二进制 111b = 7, 101b = 5)
5  chmod 755 run.sh
6
7  # 移除所有人对 run.sh 的写权限
8  chmod -w run.sh
```

diff 命令用于比较文件的差异。

```

1 diff [选项] file1 file2
2 选项 (常用):
3 -b          不检查空白字符的不同
4 -B          不检查空行
5 -q          仅显示有无差异, 不显示详细信息

```

`sed` 是一个文件处理工具, 可以将数据行进行替换、删除、新增、选取等特定工作。

```

1 sed
2 sed [选项] 命令 输入文本
3 选项 (常用):
4 -n          安静模式, 只显示经过 sed 处理的内容。否则显示输入文本的所有内容。
5 -i          直接修改读取的档案内容, 而不是输出到屏幕。否则, 只输出不编辑。
6 命令 (常用):
7 [行号] a\ [内容] 新增, 在行号后新增一行相应内容。行号可以是“数字”, 在这一行之后新增,
8                也可以是“起始行, 终止行”, 在其中的每一行后新增。
9                当不写行号时, 在每一行之后新增。使用 $ 表示最后一行。后面的命令同理。
10 [行号] c\ [内容] 取代。用内容取代相应行的文本。
11 [行号] i\ [内容] 插入。在当前行的上面插入一行文本。
12 [行号] d      删除当前行的内容。
13 [行号] p      输出选择的内容。通常与选项 -n 一起使用。
14 s/re/string  将 re (正则表达式) 匹配的内容替换为 string。

```

`sed` 中正则表达式的相关语法可以查阅 [sed 文档](#)。`sed` 等工具中的正则表达式语法和 Java 等语言中不完全相同, 请注意区分。

下面是一些 `sed` 的使用示例:

```

1 # 输出 my.txt 的第三行
2 sed -n '3p' my.txt
3
4 # 删除 my.txt 文件的第二行
5 sed '2d' my.txt
6
7 # 删除 my.txt 文件的第二行到最后一行 ($ 符号表示到最末尾)
8 sed '2,$d' my.txt
9
10 # 在整行范围内把 str1 替换为 str2
11 # 如果没有 g 标记, 则只有每行第一个匹配的 str1 被替换成 str2
12 sed 's/str1/str2/g' my.txt
13
14 # -e 选项允许在同一行里执行多条命令。例子的第一条是第四行后添加一个 str,
15 # 第二个命令是将 str 替换为 aaa。命令的执行顺序对结果有影响。
16 sed -e '4a\str ' -e 's/str/aaa/' my.txt

```

`awk` 是一种处理文本文件的语言, 是一个强大的文本分析工具。这里只举几个简单的例子, 学有余力的同学可以自行深入学习。

```

1 awk '$1>2 {print $1,$3}' my.txt

```

这个命令的格式为 `awk 'pattern action' file`, `pattern` 为条件, `action` 为命令, `file` 为文件。命令中出现的 `$n` 代表每一行中用空格分隔后的第 `n` 项。所以该命令的意义是输出文件 `my.txt` 中所有第一项大于 2 的行的第一项和第三项。

```
1 awk -F, '{print $2}' my.txt
```

`-F` 选项用来指定用于分隔的字符，默认是空格。所以该命令的 `$n` 就是用 “,” 分隔的第 n 项了。

`tmux` 是一个优秀的终端复用软件，可用于在一个终端窗口中运行多个终端会话。窗格、窗口和会话是 `tmux` 的三个基本概念，一个会话可以包含多个窗口，一个窗口可以分割为多个窗格。突然中断退出后 `tmux` 仍会保持会话，通过进入会话可以直接从之前的环境开始工作。

1. 窗格操作

`tmux` 的窗格 (pane) 可以做出分屏的效果。

- `Ctrl+B %` 垂直分屏 (组合键之后按一个百分号)，用一条垂线把当前窗口分成左右两屏。
- `Ctrl+B "` 水平分屏 (组合键之后按一个双引号)，用一条水平线把当前窗口分成上下两屏。
- `Ctrl+B O` 依次切换当前窗口下的各个窗格。
- `Ctrl+B Up|Down|Left|Right` 根据按箭方向选择切换到某个窗格。
- `Ctrl+B Space` (空格键) 对当前窗口下的所有窗格重新排列布局，每按一次，换一种样式。
- `Ctrl+B Z` 最大化当前窗格。再按一次后恢复。
- `Ctrl+B X` 关闭当前使用中的窗格，操作之后会给出是否关闭的提示，按 `y` 确认即关闭。

2. 窗口操作

每个窗口 (window) 可以分割成多个窗格 (pane)。

- `Ctrl+B C` 创建之后会多出一个窗口
- `Ctrl+B P` 切换到上一个窗口。
- `Ctrl+B N` 切换到下一个窗口。
- `Ctrl+B 0` 切换到 0 号窗口，依此类推，可换成任意窗口序号
- `Ctrl+B W` 列出当前 session 所有窗口，通过上、下键切换窗口
- `Ctrl+B &` 关闭当前 window，会给出提示是否关闭当前窗口，按下 `y` 确认即可。

3. 会话操作

一个会话 (session) 可以包含多个窗口 (window)

- `tmux new -s 会话名` 新建会话
- `Ctrl+B D` 退出会话，回到 shell 的终端环境

- `tmux ls` 终端环境查看会话列表
- `tmux a -t` 会话名从终端环境进入会话
- `tmux kill-session -t` 会话名销毁会话

0.6.2 shell 脚本

在以后的工作中，可能会遇到重复多次用到单条或多条长而复杂命令的情况，初学者可能会想把这些命令保存在一个文件中，以后再打开文件复制粘贴运行，其实大可不必复制粘贴，将文件按照批处理脚本运行即可。简单来说，批处理脚本就是存储了一条或多条命令的文本文件。当有很多想要执行的 Linux 命令来完成复杂的工作，或者有一个或一组命令会经常执行时，我们可以通过 shell 脚本来完成。本节我们将学习使用 bash shell。

首先执行 `vim my.sh` 创建并打开一个文件 `my.sh`，使用 Vim 将其打开，并向其中写入以下内容（别忘了在 Vim 里要输入要先按 `i` 进入插入模式）：

```
1  #!/bin/bash
2  # balabala
3  echo "Hello World!"
```

我们可以使用 `bash` 来运行这个脚本：

```
1  bash my.sh
```

另外，我们也可以通过命令 `chmod +x my.sh` 来为脚本添加执行权限，然后使用

```
1  ./my.sh
```

来运行。

在修改权限之前，我们自己创建的 shell 脚本一般是不能直接运行的，需要用 `chmod +x my.sh` 添加运行权限：在脚本中我们通常会加入 `#!/bin/bash` 到文件首行，以保证直接执行我们的脚本时使用 `bash` 作为解释器。第二行的内容是注释，以 `#` 开头。第三行是命令。

参数与函数

我们可以向 shell 脚本传递参数。文件 `my2.sh` 的内容如下：

```
1  #!/bin/bash
2  echo $1
```

执行命令

```
1  ./my2.sh msg
```

则 shell 会执行 `echo msg` 这条命令。

`$n` 就代表第几个参数，而 `$0` 也就是命令，在例子中就是 `./my2.sh`。除此之外还有一些可能有用的符号组合

- `$#` 传递的参数个数
- `$*` 一个字符串显示传递的全部参数
- `$?` 获取前一个命令的返回值

shell 中的函数也用类似的方式传递参数。

```
1 function < 函数名 > () {
2     <commands>
3 }
```

`function` 或者 `()` 可以省略其中一个。例如：

```
1 # 定义函数 fun
2 fun() {
3     echo "$1"
4     echo "$2"
5     echo "the number of parameters is $#"
```

流程控制语句

shell 脚本中也可以使用分支和循环语句：

`if` 的格式：

```
1 if <condition>
2 then
3     <command1>
4     <command2>
5     # ...
6 fi
```

或者写到一行

```
1 if condition; then command1; command2; ... fi
```

例如（注意等号前后不能有空格，左中括号后和右中括号前一定需要空格）：

```
1 a=1
2 if [ $a -ne 1 ]; then echo ok; fi
```

条件部分可能会让同学们感到疑惑，实际上 `<condition>` 的位置上也是命令，一条命令的返回值为 0 时表示其成功执行，作为条件时则视为成立。可以使用特殊变量 `$?` 获取前一个命令的返回值，比如刚执行完 `diff <file1> <file2>` 后，若两文件内容相同，则 `$?` 为 0。

左中括号 `[` 是一种常用作条件的命令，其参数是一个条件表达式和末尾的 `]`，在上例中为 `$a`、`-ne`、`1` 和 `]`。该命令在关系成立时返回 0，其完整形式为 `test` 命令，可以在终端中使用 `man test` 查看其详细用法。此外，`true` 命令能够直接返回 0，`!` 命令能够反转参数中命令的返回值，也经常在条件中使用，例如 `! diff <file1> <file2>` 在两文件内容不同时返回 0。

条件表达式中的 `-ne` 是一种关系运算符，它们和 C 语言的比较运算符对应如下：

```
-eq  == (equal)
-ne  != (not equal)
-gt  > (greater than)
-lt  < (less than)
-ge  >= (greater or equal)
-le  <= (less or equal)
```

`while` 语句格式如下

```
1 while <condition>
2 do
3     <commands>
4 done
```

`while` 语句体中可以使用 `continue` 和 `break` 这两个循环控制语句。

例如创建 9 个目录，名字是 `file1` 到 `file9`。

```
1 a=1
2 while [ $a -ne 10 ]
3 do
4     mkdir file$a
5     a=$((a+1))
6 done
```

除了以上内容，shell 还有 `for`、`case` 语句，`else` 子句，以及逻辑运算符等语法，对这些内容有兴趣的同学可以自行了解。对于部分 shell 编程中容易让人误解的点，可以参考 [附录中的 shell 编程易错点说明](#)。

0.6.3 重定向和管道

这部分我们将学习如何实现 Linux 命令的输入输出怎样定向到文件，以及如何将多个命令组合实现更强大的功能。Linux 定义了三种流：

- 标准输入：`stdin`，由 0 表示
- 标准输出：`stdout`，由 1 表示
- 标准错误：`stderr`，由 2 表示

重定向和管道可以重定向以上的流。`>` 可以重定向命令的标准输出到文件。例如：`ls / > filename` 可以将根目录下的文件名输出到当前目录下的 `filename` 中。与之类似的，还有重定向追加输出 `>>`，将 `>>` 前命令的输出追加输出到 `>>` 后指定的文件中；以及重定向输入 `<`，将 `<` 后指定的文件中的数据输入到 `<` 前的命令中去，同学们可以自己动手实践一下。`"2>>"` 可以将标准错误重定向。三种流可以同时重定向，举例：

```
1 command < input.txt 1>output.txt 2>err.txt
```

管道：

管道符号 `|` 可以连接命令：

```
1 command1 | command2 | command3 | ...
```

以上内容是将 `command1` 的 `stdout` 发给 `command2` 的 `stdin`，`command2` 的 `stdout` 发给 `command3` 的 `stdin`，依此类推。例如：

```
1 cat my.sh | grep "Hello"
```

上述命令的功能为将 `my.sh` 的内容输出给 `grep` 命令, `grep` 在其中查找字符串。

```
1 cat < my.sh | grep "Hello" > output.txt
```

上述命令重定向和管道混合使用, 功能为将 `my.sh` 的内容作为 `cat` 命令参数, `cat` 命令 `stdout` 发给 `grep` 命令的 `stdin`, `grep` 在其中查找字符串, 最后将结果输出到 `output.txt`。

Thinking 0.5 执行如下命令, 并查看结果

- `echo first`
- `echo second > output.txt`
- `echo third > output.txt`
- `echo forth >> output.txt`

Thinking 0.6 使用你知道的方法 (包括重定向) 创建下图内容的文件 (文件命名为 `test`), 将创建该文件的命令序列保存在 `command` 文件中, 并将 `test` 文件作为批处理文件运行, 将运行结果输出至 `result` 文件中。给出 `command` 文件和 `result` 文件的内容, 并对最后的结果进行解释说明 (可以从 `test` 文件的内容入手)。具体实现的过程中思考下列问题: `echo echo Shell Start` 与 `echo `echo Shell Start`` 效果是否有区别; `echo echo $c>file1` 与 `echo `echo $c>file1`` 效果是否有区别。

```
echo Shell Start...
echo set a = 1
a=1
echo set b = 2
b=2
echo set c = a+b
c=${a+$b}
echo c = $c
echo save c to ./file1
echo $c>file1
echo save b to ./file2
echo $b>file2
echo save a to ./file3
echo $a>file3
echo save file1 file2 file3 to file4
cat file1>file4
cat file2>>file4
cat file3>>file4
echo save file4 to ./result
cat file4>>result
```

图 0.14: 文件内容

0.7 实战测试

随着 Lab0 学习的结束，下面就要开始通过实战检验水平了，请同学们按照下面的题目要求完成所需操作。

Exercise 0.1 Lab0 第一道练习题包括以下四题，如果你四道题全部完成且正确，即可获得 50 分。

1、在 Lab0 工作区的 `src` 目录中，存在一个名为 `palindrome.c` 的文件，使用刚刚学过的工具打开 `palindrome.c`，使用 c 语言实现判断输入整数 $n(1 \leq n \leq 10000)$ 是否为回文数的程序（输入输出部分已经完成）。通过 `stdin` 每次只输入一个整数 n ，若这个数字为回文数则输出 Y，否则输出 N。[注意：正读倒读相同的整数叫回文数]

2、在 `src` 目录下，存在一个未补全的 `Makefile` 文件，借助刚刚掌握的 `Makefile` 知识，将其补全，以实现通过 `make` 命令触发 `src` 目录下的 `palindrome.c` 文件的编译链接的功能，生成的可执行文件命名为 `palindrome`。

3、在 `src/sh_test` 目录下，有一个 `file` 文件和 `hello_os.sh` 文件。`hello_os.sh` 是一个未完成的脚本文档，请同学们借助 shell 编程的知识，将其补完，以实现通过命令 `bash hello_os.sh AAA BBB`，在 `hello_os.sh` 所处的目录新建一个名为 `BBB` 的文件，其内容为 `AAA` 文件的第 8、32、128、512、1024 行的内容提取（`AAA` 文件行数一定超过 1024 行）。[注意：对于命令 `bash hello_os.sh AAA BBB`，`AAA` 及 `BBB` 可为任何合法文件的名称，例如 `bash hello_os.sh file hello_os.c`，若已有 `hello_os.c` 文件，则将其原有内容覆盖]

4、补全后的 `palindrome.c`、`Makefile`、`hello_os.sh` 依次复制到路径 `dst/palindrome.c`，`dst/Makefile`，`dst/sh_test/hello_os.sh` [注意：文件名和路径必须与题目要求相同]

要求按照要求完成后，最终提交的文件树图示如下

```

1 |-- dst
2 |   |-- Makefile
3 |   |-- palindrome.c
4 |   `-- sh_test
5 |       |-- hello_os.sh
6 |-- src
7 |   |-- Makefile
8 |   |-- palindrome.c
9 |   `-- sh_test
10 |       |-- file
11 |       `-- hello_os.sh

```

第一题最终提交的文件树

Exercise 0.2 Lab0 第二道练习题包括以下一题，如果你完成且正确，即可获得 12 分。

1、在 Lab0 工作区 `ray/sh_test1` 目录中，含有 100 个子目录 `file1~file100`，还存在一个名为 `changeFile.sh` 的文件，将其补完，以实现通过命令 `bash changeFile.sh`，可以删除该目录内 `file71~file100` 共计 30 个子目录，将 `file41~file70` 共计 30 个子目录重命名为 `newfile41~newfile70`。[注意：评测时仅检测 `changeFile.sh` 的正确性]

要求按照要求完成后，最终提交的文件树图示如下（`file` 下标只显示 1~12，`newfile` 下标只显示 41~55）

```
1  |-- sh_test1
2      |-- file1
3      |-- file10
4      |-- file11
5      |-- file12
6      |-- file2
7      |-- file3
8      |-- file4
9      |-- file5
10     |-- file6
11     |-- file7
12     |-- file8
13     |-- file9
14     |-- newfile41
15     |-- newfile42
16     |-- newfile43
17     |-- newfile44
18     |-- newfile45
19     |-- newfile46
20     |-- newfile47
21     |-- newfile48
22     |-- newfile49
23     |-- newfile50
24     |-- newfile51
25     |-- newfile52
26     |-- newfile53
27     |-- newfile54
28     |-- newfile55
29  |-- changefile.sh
```

第二题最终提交的文件树

Exercise 0.3 Lab0 第三道练习题包括以下一题，如果你完成且正确，即可获得 12 分。

1、在 Lab0 工作区的 `ray/sh_test2` 目录下，存在一个未补全的 `search.sh` 文件，将其补完，以实现通过命令 `bash search.sh file int result`，可以在当前目录下生成 `result` 文件，内容为 `file` 文件含有 `int` 字符串所在的行数，即若有多行含有 `int` 字符串需要全部输出。[注意：对于命令 `bash search.sh file int result`，`file` 及 `result` 可为任何合法文件名称，`int` 可为任何合法字符串，若已有 `result` 文件，则将其原有内容覆盖，匹配时大小写不忽略]

要求按照要求完成后，`result` 内显示样式如下（一个答案占一行）：

```

1 39
2 123
3 134
4 147
5 344
6 395
7 446
8 471
9 735
10 908
11 1207
12 1422
13 1574
14 1801
15 1822
16 1924
17 1940
18 1984

```

第三题完成后结果

Exercise 0.4 Lab0 第四道练习题包括以下两题，如果你均完成且正确，即可获得 26 分。

1、在 Lab0 工作区的 `csc/code` 目录下，存在 `fibo.c`、`main.c`，其中 `fibo.c` 有点小问题，还有一个未补全的 `modify.sh` 文件，将其补全，以实现通过命令 `bash modify.sh fibo.c char int`，可以将 `fibo.c` 中所有的 `char` 字符串更改为 `int` 字符串。[注意：对于命令 `bash modify.sh fibo.c char int`，`fibo.c` 可为任何合法文件名，`char` 及 `int` 可以是任何字符串，评测时评测 `modify.sh` 的正确性，而不是检查修改后 `fibo.c` 的正确性]

2、Lab0 工作区的 `csc/code/fibo.c` 成功更换字段后 (`bash modify.sh fibo.c char int`)，现已有 `csc/Makefile` 和 `csc/code/Makefile`，补全两个 `Makefile` 文件，要求在 `csc` 目录下通过命令 `make` 可在 `csc/code` 目录中生成 `fibo.o`、`main.o`，在 `csc` 目录中生成可执行文件 `fibo`，再输入命令 `make clean` 后只删除两个 `.o` 文件。[注意：不能修改 `fibo.h` 和 `main.c` 文件中的内容，提交的文件中 `fibo.c` 必须是修改后正确的 `fibo.c`，可执行文件 `fibo` 作用是输入一个整数 `n` (从 `stdin` 输入 `n`)，可以输出斐波那契数列前 `n` 项，每一项之间用空格分开。比如 `n=5`，输出 `1 1 2 3 5`]

要求成功使用脚本文件 `modify.sh` 修改 `fibo.c`，实现使用 `make` 命令可以生成 `.o` 文件和可执行文件，再使用命令 `make clean` 可以将 `.o` 文件删除，但保留 `fibo` 和 `.c` 文件。最终提交时文件中 `fibo` 和 `.o` 文件可有可无。

```

1 |-- code
2 | |-- Makefile
3 | |-- fibo.c
4 | |-- fibo.o
5 | |-- main.c
6 | |-- main.o
7 | `-- modify.sh
8 |-- fibo
9 |-- include
10 | `-- fibo.h
11 `-- Makefile

```

第四题 `make` 后文件树

```
1 |-- code
2 |   |-- Makefile
3 |   |-- fibo.c
4 |   |-- main.c
5 |   `-- modify.sh
6 |-- fibo
7 |-- include
8 |   `-- fibo.h
9 `-- Makefile
```

第四题 `make clean` 后文件树

0.8 实验思考

- 思考-Git 的使用 1
- 思考-箭头与命令
- 思考-Git 的一些场景
- 思考-Git 的使用 2
- 思考-echo 的使用
- 思考-文件的操作

1.1 实验目的

1. 从操作系统角度理解 MIPS 体系结构
2. 掌握操作系统启动的基本流程
3. 掌握 ELF 文件的结构和功能
4. 完成 `printk` 函数的编写

在本章中，需要阅读并填写部分代码，使得 MOS 操作系统可以正常的运行起来。这一章节的难度较为简单。

1.2 操作系统的启动

1.2.1 QEMU 模拟器

计算机是由硬件和软件组成，如果仅有一个裸机什么工作也无法完成。另一方面，软件也必须运行在硬件之上才能实现其价值。由此可见，硬件和软件是相互依存、密不可分的。为了能较好的管理计算机系统的硬件资源，需要使用操作系统。

那么在我们的操作系统实验中，需要管理的硬件在哪里呢？通过前面的学习，可以知道 QEMU 是一款计算机模拟器，在本实验中可以模拟 CPU 等硬件环境。总的来说，在操作系统课程实验中，编写代码的环境是 Linux 系统，进行实验的硬件仿真平台是 QEMU 模拟器。实验编写的操作系统代码在 Linux 环境中通过 Makefile 来组织，通过交叉编译产生可执行文件，最后使用 QEMU 模拟器运行该可执行文件，实现 MOS 操作系统的运行。

Note 1.2.1 操作系统的启动英文称作“boot”。这个词是 bootstrap 的缩写，意思是鞋带（靴子上的那种）。之所以将操作系统启动称为 boot，源自于一个英文的成语“pull oneself up by one's bootstraps”，直译过来就是用自己的鞋带把自己提起来。操作系统启动的过程正是这样一个极度纠结的过程。硬件是在软件的控制下执行的，而当刚上电的时候，存储设备上的软件又需要由硬件载入到内存中去执行。可是没有软件的控制，谁来指挥硬件去载入软件？因此，就产生了一个类似于鸡生蛋，蛋生鸡一样的问题。硬件需要软件控制，软件又依赖硬件载入。就好像要用自己的鞋带把自己提起来一样。早期的工程师们在这一问

题上消耗了大量的精力。所以，他们后来将“启动”这一纠结的过程称为“boot”。

真实操作系统内核的存储位置以及启动流程请同学们阅读[附录中的真实操作系统的内核及启动详解](#)了解，关于 QEMU 中操作系统的启动流程请同学们继续阅读下一章。

1.2.2 QEMU 中的启动流程

操作系统的启动是一个非常复杂的过程。不过，由于 MOS 操作系统的目标是在 QEMU 模拟器上运行，这个过程被大大简化了。QEMU 模拟器支持直接加载 ELF 格式的内核，也就是说，QEMU 已经提供了 bootloader 的引导（启动）功能。MOS 操作系统不需要再实现 bootloader 的功能。在 MOS 操作系统的运行第一行代码前，我们就已经拥有一个正常的程序运行环境，内存和一些外围设备都可以正常使用。

QEMU 支持加载 ELF 格式内核，所以启动流程被简化为加载内核到内存，之后跳转到内核的入口，启动就完成了。这里要注意，之所以简单还有一个原因就在于 QEMU 本身是模拟器，是一种模拟硬件的软件而不是真正的硬件，所以就不需要面对传统的 bootloader 面对的那种非常纠结的情况了。

1.3 Let's hack the kernel!

接下来，我们就要开始来折腾我们的 MOS 操作系统内核了。这一节中，我们将介绍如何修改内核并实现一些自定义的功能。

1.3.1 Makefile——内核代码的地图

当我们使用 `ls` 命令看看都有哪些实验代码时，会发现似乎文件目录有点多，各个不同的目录名称大致说明了他们各自的功用，但是挨个文件进行浏览还是有点难度。

不过我们看见有一个文件非常熟悉，叫做 `Makefile`。

相信大家在 lab0 中，已经对 `Makefile` 有了初步的了解。下面这个 `Makefile` 文件即为构建我们整个操作系统所用的顶层 `Makefile`。那么，我们就可以通过浏览这个文件来对整个操作系统的布局产生初步的了解：可以说，`Makefile` 就像源代码的地图，告诉我们源代码是如何一步一步成为最终的可执行文件的。代码 1.3.1 是实验代码最顶层的 `Makefile`，通过阅读它我们就能了解代码中很多宏观的东西。（为了方便理解，简化了部分内容并加入了一些注释）

顶层 Makefile

```

1  include include.mk
2
3  target_dir    := target          # MOS 构建目标所在目录
4  mos_elf       := $(target_dir)/mos # 最终需要生成的 ELF 可执行文件
5  user_disk     := $(target_dir)/fs.img # MOS 文件系统使用的磁盘镜像文件
6  link_script   := kernel.lds
7
8  modules       := lib init kern   # 需要生成的子模块
9  objects       := $(addsuffix /*.o, $(modules)) # 遍历需要生成的目标文件
10
11 QEMU_FLAGS    := -cpu 4Kc -m 64 -nographic -M malta \
12                $(shell [ -f '$(user_disk)' ] && \
13                echo '-drive id=id0,file=$(user_disk),if=id0,format=raw') \

```

```

14         -no-reboot # QEMU 运行参数
15
16 .PHONY: all $(modules) clean
17
18 all: $(mos_elf) # 我们的“最终目标”
19
20 $(mos_elf): $(modules) # 调用链接器 $(LD) 链接所有目标文件
21     $(LD) $(LDFLAGS) -o $(mos_elf) -N -T $(link_script) $(objects)
22
23 $(modules): # 进入各个子目录进行 make
24     $(MAKE) --directory=$@
25
26 clean:
27     for d in $(modules); do \
28         $(MAKE) --directory=$$d clean; \
29     done; \
30     rm -rf *.o *~ $(mos_elf)
31
32 run:
33     $(QEMU) $(QEMU_FLAGS) -kernel $(mos_elf)

```

如果你以前没有接触过 `Makefile` 的话，仅仅凭借 lab0 的简单练习获得的知识，可能还不足以顺畅的阅读这份 `Makefile`。不必着急，我们来一行一行地解读它。在 3 行 - 11 行可以看到我们熟悉的赋值符号，这是 `Makefile` 中对变量的定义语句，它们定义了各个子模块的目录名 `modules`、最终生成的内核可执行文件的路径 `mos_elf` 和 linker script 的位置等。其中，最值得注意的两个变量分别是 `modules` 和 `objects`。`modules` 定义了内核所包含的所有模块，`objects` 则表示要编译出内核所依赖的所有目标文件 (`*.o`)。11 行 - 13 行行末的斜杠代表这一行没有结束，下一行的内容和这一行是连在一起的。这种写法可以把本该写在同一行的东西分布在多行中，使得文件更具有可读性。

Note 1.3.1 linker script 是用于指导链接器将多个 `.o` 文件链接成目标可执行文件的脚本。`.o` 文件、linker script 等内容我们会在下面的小节中细致地讲解，大家这里只要知道这些文件是编译内核所必要的就好。

16 行的 `.PHONY` 表明列在其后的目标不受修改时间的约束。也就是说，一旦该规则被调用，无视 `make` 工具编译时有关时间戳的性质，无论依赖文件是否被修改，一定保证它被执行。

第 18 行定义 `all` 这一规则的依赖。`all` 代表整个项目，由此我们可以知道，构建整个项目依赖于构建好内核可执行文件 `$(mos_elf)`。那么 `$(mos_elf)` 是如何被构建的呢？紧接着的 20 行定义了，`$(mos_elf)` 的构建依赖于所有的模块。在构建完所有模块后，将执行第 21 行的命令来产生 `$(mos_elf)`。我们可以看到，第 21 行调用了链接器将之前构建各模块产生的所有 `.o` 文件在 linker script 的指导下链接到一起，产生最终的 `mos` 可执行文件。第 23 行定义了每个模块的构建方法为调用对应模块目录下的 `Makefile`。最后的 26 到 30 行定义了如何清理所有被构建出来的文件。

Note 1.3.2 一般在写 `Makefile` 时，习惯将第一个规则命名为 `all`，也就是构建整个项目的意思。如果调用 `make` 时没有指定目标，`make` 会自动执行第一个目标，所以把 `all` 放在第一个目标的位置上，可以使得 `make` 命令默认构建整个项目，较为方便。

读到这里，有一点需要注意，我们在编译命令中使用了 `CC`、`LD`、`MAKE` 等变量，但是我们似乎从来没有定义过这个变量。那么这个变量定义在哪呢？

我们看到第 1 行有一条 `include` 命令。看来，这个顶层的 `Makefile` 还引用了其他的東西。让我们来看看这个文件，被引用的文件如代码 1.3.1 所示。


```

                                include.mk

1  CROSS_COMPILE := mips-linux-gnu-
2  CC             := $(CROSS_COMPILE)gcc
3  CFLAGS         += --std=gnu99 -EL -G 0 -mno-abicalls -fno-pic \
4                  -ffreestanding -fno-stack-protector -fno-builtin \
5                  -Wa,-xgot -Wall -mxgot -mno-fix-r4000 -march=4kc
6  LD             := $(CROSS_COMPILE)ld
7  LDFLAGS        += -EL -G 0 -static -n -nostdlib --fatal-warnings

```

原来我们的 `LD` 变量是在这里被定义的（变量 `MAKE` 是 `Makefile` 中预定义的变量，不是我们定义的）。在该文件中，我们看到了一个关键词——Cross Compile（交叉编译）。这里的 `CROSS_COMPILE` 变量是实际使用的编译器和链接器等工具的前缀，或者说是交叉编译器的具体位置。例如，在我们的实验环境中，`LD` 最终调用的链接器是“`mips-linux-gnu-ld`”。通过修改该变量，就可以方便地设定交叉编译使用的工具链。

至此，我们就可以大致掌握阅读 `Makefile` 的方法了。善于运用 `make` 的功能可以提高开发效率。提示：可以试着使用一下 `make clean`。

执行 `make` 命令，如果配置正确，则会在 `target` 目录下生成内核镜像文件 `mos`。

最后，简要总结一下实验代码中其他目录的组织以及其中的重要文件：

- 根目录下还存在 `kernel.lds` 这个 linker script 文件，我们会在下面的小节中详细讲解。
- `init` 目录中主要有两个代码文件 `start.S` 和 `init.c`，其作用是初始化内核。`start.S` 文件中的 `_start` 函数是 CPU 控制权被转交给内核后执行的第一个函数，主要工作是初始化 CPU 和栈指针，为之后的内核初始化做准备，最后跳转到 `init.c` 文件中定义的 `mips_init` 函数。在本章中 `mips_init` 函数只是简单的打印输出，而在之后的实验中会逐步添加新的内核功能，内核中各模块的初始化函数都会在这里被调用。
- `include` 目录中存放系统头文件。在本章中需要用到的头文件是 `mmu.h` 文件，这个文件中有一张内存布局图，我们在填写 linker script 的时候需要根据这个图来设置相应节的加载地址。
- `lib` 目录存放一些常用库函数，本章中主要存放用于格式化输出的函数。
- `kern` 目录中存放内核的主体代码，本章中主要存放的是终端输出相关的函数。
- `tests` 目录中存放公开的测试用例，我们在进行本地测试时会用到它。

1.3.2 ELF——操作系统内核的本质

通过阅读 `Makefile` 文件，我们大致把握了 MOS 操作系统的代码组织结构。在这一小节里，我们来进一步探究 MOS 代码最终编译链接得到“操作系统内核”究竟是什么。

我们知道源代码文件需要经过编译和链接两个阶段，才能变成完整的可执行文件来运行。在编译阶段，每个源文件被翻译成二进制指令，得到**目标文件**，也就是我们通过 `-c` 编译选项生成的 `.o` 文件。然而，由于完整程序的地址布局尚未确定，目标文件中的跳转等指令中的目标地址还是空的。在链接阶段，链接器会将所有的目标文件链接在一起，并填写具体的地址等信息，形成最终的**可执行文件**。

Thinking 1.1 在阅读 [附录中的编译链接详解](#) 以及本章内容后，尝试分别使用实验环境中的原生 x86 工具链 (gcc、ld、readelf、objdump 等) 和 MIPS 交叉编译工具链 (带有 mips-linux-gnu- 前缀，如 mips-linux-gnu-gcc、mips-linux-gnu-ld)，重复其中的编译和解析过程，观察相应的结果，并解释其中向 objdump 传入的参数含义。 ■

接下来，我们提出我们的下一个问题：**链接器通过哪些信息来链接多个目标文件呢？** 答案就在于在目标文件的结构中。在目标文件中，记录了代码各个段的具体信息。链接器通过这些信息来将目标文件链接到一起。而 ELF (Executable and Linkable Format) 正是 Unix 上常用的一种目标文件格式。其实，不仅仅是目标文件，可执行文件也是使用 ELF 格式记录的。这一点通过 ELF 的全称也可以看出来。

我们前面编译出来的 MOS 内核本质上也是一个 ELF 文件。为了进一步了解编译出的内核文件，我们需要进一步探究 ELF 文件的功能以及格式。

ELF 是一种用于可执行文件、目标文件和库的文件格式。ELF 格式是 UNIX 系统实验室作为 ABI(Application Binary Interface) 而开发和发布的，现在早已经是 Linux 下的标准格式了。我们在之前曾经看见过的 .o 文件就是 ELF 所包含的三种文件类型中的一种，称为可重定位 (relocatable) 文件，其它两种文件类型分别是可执行 (executable) 文件和共享对象 (shared object) 文件，这两种文件都需要链接器对可重定位文件进行处理才能生成。

你可以使用 `file` 命令来获得文件的类型，如下所示：

```
1 git@21xxxxxx:~/test$ file a.o
2 a.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
3 git@21xxxxxx:~/test$ file a.out
4 a.out: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked,
5 interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=440d864c365dbaa240f8fb8584e7
6 49c3816da34b, for GNU/Linux 3.2.0, not stripped
7 git@21xxxxxx:~/test$ file a.so
8 a.so: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked,
9 BuildID[sha1]=158029301e80e6a5b650a429d5c0bc5516dfbeba, not stripped
```

那么，ELF 文件中都包含有什么东西呢？简而言之，就是和程序相关的所有必要信息，下图 1.1 说明了 ELF 文件的结构：

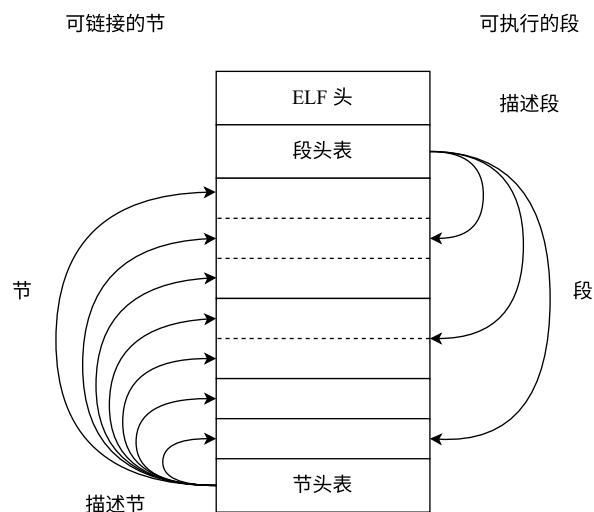


图 1.1: ELF 文件结构

通过上图我们可以知道，ELF 文件从整体来说包含 5 个部分：

1. ELF 头，包括程序的基本信息，比如体系结构和操作系统，同时也包含了节头表和段头表相对文件的偏移量 (offset)。
2. 段头表 (或程序头表, program header table), 主要包含程序中各个段 (segment) 的信息, 段的信息需要在运行时刻使用。
3. 节头表 (section header table), 主要包含程序中各个节 (section) 的信息, 节的信息需要在程序编译和链接的时候使用。
4. 段头表中的每一个表项, 记录了该段数据载入内存时的目标位置等, 记录了用于指导应用程序加载的各类信息。
5. 节头表中的每一个表项, 记录了该节程序的代码段、数据段等各个段的内容, 主要是链接器在链接的过程中需要使用。

观察图 1.1, 我们可以发现, 段头表和节头表指向了同样的地方, 这意味着两者只是程序数据的两种视图:

1. 组成可重定位文件, 参与可执行文件和可共享文件的链接。此时使用节头表。
2. 组成可执行文件或者可共享文件, 在运行时为加载器提供信息。此时使用段头表。

我们已经了解了 ELF 文件的大体结构以及相应功能, 现在, 我们需要同学们自己动手, 阅读一个简易的对 32-bit little-endian ELF 文件的解析程序, 然后完成部分代码, 来了解 ELF 文件各个部分的详细结构。

为了帮助大家进行理解, 我们先对这个程序中涉及的三个关键数据结构做一下简要说明, 请你切换到 `tools/readelf` 目录, 阅读 `elf.h` 这个头文件, 结合下方说明, 仔细阅读其中的代码和注释, 然后完成练习。下面的代码都截取自 `tools/readelf/elf.h` 文件:

```

1
2  /* 文件的前面是各种变量类型定义, 在此省略 */
3  /* The ELF file header.  This appears at the start of every ELF file.  */
4  /* ELF 文件的文件头。所有的 ELF 文件均以此为起始 */
5  #define EI_NIDENT (16)
6
7  typedef struct {
8      unsigned char    e_ident[EI_NIDENT]; /* Magic number and other info */
9      /* 存放魔数以及其他信息
10     Elf32_Half        e_type;              /* Object file type */
11     /* 文件类型
12     Elf32_Half        e_machine;          /* Architecture */
13     /* 机器架构
14     Elf32_Word        e_version;          /* Object file version */
15     /* 文件版本
16     Elf32_Addr        e_entry;            /* Entry point virtual address */
17     /* 入口点的虚拟地址
18     Elf32_Off         e_phoff;            /* Program header table file offset */
19     /* 程序头表所在处与此文件头的偏移
20     Elf32_Off         e_shoff;            /* Section header table file offset */
21     /* 节头表所在处与此文件头的偏移
22     Elf32_Word        e_flags;            /* Processor-specific flags */
23     /* 针对处理器的标记
24     Elf32_Half        e_ehsize;           /* ELF header size in bytes */
25     /* ELF 文件头的大小 (单位为字节)
26     Elf32_Half        e_phentsize;       /* Program header table entry size */
27     /* 程序头表表项大小
28     Elf32_Half        e_phnum;           /* Program header table entry count */

```

```

29      // 程序头表项数
30      Elf32_Half      e_shentsize;          /* Section header table entry size */
31      // 节头表项大小
32      Elf32_Half      e_shnum;              /* Section header table entry count */
33      // 节头表项数
34      Elf32_Half      e_shstrndx;          /* Section header string table index */
35      // 节头字符串编号
36  } Elf32_Ehdr;
37
38  typedef struct {
39      // section name
40      Elf32_Word sh_name;
41      // section type
42      Elf32_Word sh_type;
43      // section flags
44      Elf32_Word sh_flags;
45      // section addr
46      Elf32_Addr sh_addr;
47      // offset from elf head of this entry
48      Elf32_Off  sh_offset;
49      // byte size of this section
50      Elf32_Word sh_size;
51      // link
52      Elf32_Word sh_link;
53      // extra info
54      Elf32_Word sh_info;
55      // alignment
56      Elf32_Word sh_addralign;
57      // entry size
58      Elf32_Word sh_entsize;
59  } Elf32_Shdr;
60
61  typedef struct {
62      // segment type
63      Elf32_Word p_type;
64      // offset from elf file head of this entry
65      Elf32_Off  p_offset;
66      // virtual addr of this segment
67      Elf32_Addr p_vaddr;
68      // physical addr, this value is meaningless in linux and has same value of p_vaddr
69      Elf32_Addr p_paddr;
70      // file size of this segment
71      Elf32_Word p_filesz;
72      // memory size of this segment
73      Elf32_Word p_memsz;
74      // segment flag
75      Elf32_Word p_flags;
76      // alignment
77      Elf32_Word p_align;
78  } Elf32_Phdr;

```

通过阅读代码可以发现，原来 ELF 的文件头，就是一个存了关于 ELF 文件信息的结构体。首先，结构体中存储了 ELF 的魔数，以验证这是一个有效的 ELF。当我们验证了这是个 ELF 文件之后，便可以通过 ELF 头中提供的信息，进一步地解析 ELF 文件了。在 ELF 头中，提供了节头表的入口偏移，这个是什么意思呢？简单来说，假设 `binary` 为 ELF 的文件头地址，`shoff` 为入口偏移，那么 `binary + shoff` 即为节头表第一项的地址。

Exercise 1.1 阅读 `tools/readelf` 目录下的 `elf.h`、`readelf.c` 和 `main.c` 文件，并补全 `readelf.c` 中缺少的代码。`readelf` 函数需要输出 ELF 文件中所有节头中的地址信息，对于每个节头，输出格式为 `"%d:0x%x\n"`，其中的 `%d` 和 `%x` 分别代表序号和地址。

正确完成 `readelf.c` 之后，在 `tools/readelf` 目录下执行 `make` 命令，即可生成可执行文件 `readelf`，它接受文件名作为参数，对 ELF 文件进行解析。可以执行 `make hello` 生成测试用的 ELF 文件 `hello`，然后运行 `./readelf hello` 来测试 `readelf`。 ■

Note 1.3.3 请阅读 `Elf32_Shdr` 这个结构体的定义。

遍历每一个节头的方法是：先读取节头的大小，随后以指向第一个节头的指针（即节头表第一项的地址）为基地址，不断累加得到每个节头的地址。

通过刚才的练习，相信你已经对 ELF 文件的结构有了初步的了解。你可能想进一步解析 ELF 文件来获得更多的信息，不过这个工作已经在系统中完成了。在我们完成练习 1.1 的过程中，生成可执行文件 `readelf` 后想要运行时，如果不小心忘记打 `“./”`，系统并不会给我们反馈 `“command not found”`，而是会列出了一些帮助信息，这是因为有一个随编译工具链提供的工具也名为 `readelf`，它的用法是 `readelf [option(s)] <elf-file(s)>`，用来解析一个或者多个 ELF 文件的信息。我们可以通过它的选项来控制显示哪些信息。例如，我们执行 `readelf -S hello` 命令后，`hello` 文件中各个节的详细信息将以列表的形式为我们展示出来。我们可以利用 `readelf` 工具来验证我们自己写的简易版 `readelf` 输出的结果是否正确，还可以使用 `readelf --help` 看到该命令各个选项及其对 ELF 文件的解析方式，从而了解我们想了解的信息。

在下面的思考题中，利用 `readelf` 工具，我们可以进一步解析先前编译出来的 `target` 文件夹下的操作系统内核 ELF 文件。需要注意：`tools` 目录下的程序是运行在 Linux 宿主机上的。它们不属于 MOS 内核代码，也不会被编译到 MOS 内核中。

Thinking 1.2 思考下述问题：

- 尝试使用我们编写的 `readelf` 程序，解析之前在 `target` 目录下生成的内核 ELF 文件。
- 也许你会发现我们编写的 `readelf` 程序是不能解析 `readelf` 文件本身的，而我们刚才介绍的系统工具 `readelf` 则可以解析，这是为什么呢？（提示：尝试使用 `readelf -h`，并阅读 `tools/readelf` 目录下的 `Makefile`，观察 `readelf` 与 `hello` 的不同） ■

现在，我们继续内核的话题。

我们最终生成的内核也是 ELF 格式的，被模拟器载入到内存中。因此，我们暂且只关注 ELF 是如何被载入到内存中，并且被执行的，而不再关心具体的链接细节。也即上文旨在探索关于我们实验环境的编译链接问题。在阅读完上面编译和 ELF 的说明后，应该明确实验中如何编译链接产生实验操作系统的 ELF 格式文件，也应该了解 QEMU 可以运行 ELF 格式的内核。ELF 中有两个相似却不同的概念，即“段”和“节”，我们之前已经提到过。

我们不妨来看一下，我们之前那个 `hello world` 程序的各个段长什么样子。`readelf` 工具可以方便地解析出 ELF 文件的内容，这里我们使用它来分析我们的程序。首先我们使用 `-l` 参数来查看各个段的信息。

```
1 Elf 文件类型为 EXEC（可执行文件）
2 入口点 0x400e6e
```

```

3  共有 5 个程序头，开始于偏移量 64
4
5  程序头：
6      Type                Offset                VirtAddr                PhysAddr
7      FileSiz             MemSiz
8      LOAD                0x0000000000000000 0x0000000000400000 0x0000000000400000
9      0x00000000000b33c0 0x00000000000b33c0 R E    200000
10     LOAD                0x00000000000b4000 0x00000000000b4000 0x00000000000b4000
11     0x0000000000001cd0 0x0000000000003f48 RW     200000
12     NOTE                0x000000000000158 0x0000000000400158 0x0000000000400158
13     0x0000000000000044 0x0000000000000044 R      4
14     TLS                 0x00000000000b4000 0x00000000000b4000 0x00000000000b4000
15     0x0000000000000020 0x0000000000000050 R      8
16     GNU_STACK            0x0000000000000000 0x0000000000000000 0x0000000000000000
17     0x0000000000000000 0x0000000000000000 RW     10
18
19  Section to Segment mapping:
20  段节...
21  00      .note.ABI-tag .note.gnu.build-id .rel.plt .init .plt .text
22  __libc_freeres_fn __libc_thread_freeres_fn .fini .rodata __libc_subfreeres
23  __libc_atexit __libc_thread_subfreeres .eh_frame .gcc_except_table
24  01      .tdata .init_array .fini_array .jcr .data.rel.ro .got .got.plt .data
25  .bss __libc_freeres_ptrs
26  02      .note.ABI-tag .note.gnu.build-id
27  03      .tdata .tbss
28  04

```

这些输出中，我们只需要关注这样几个部分：**Offset** 代表该段（segment）的数据相对于 ELF 文件的偏移。**VirtAddr** 代表该段最终需要被加载到内存的哪个位置。**FileSiz** 代表该段的数据在文件中的长度。**MemSiz** 代表该段的数据在内存中所应当占的大小。表下方的 **Section to Segment mapping** 表明每个段各自含有的节。

Note 1.3.4 **MemSiz** 永远大于等于 **FileSiz**。若 **MemSiz** 大于 **FileSiz**，则操作系统在加载程序的时候，会首先将文件中记录的数据加载到对应的 **VirtAddr** 处。之后，向内存中填 0，直到该段在内存中的大小达到 **MemSiz** 为止。那么为什么 **MemSiz** 有时候会大于 **FileSiz** 呢？这里举这样一个例子：C 语言中未初始化的全局变量，我们需要为其分配内存，但它又不需要被初始化成特定数据。因此，在可执行文件中也只记录它需要占用内存 (**MemSiz**)，但在文件中却没有相应的数据（因为它并不需要初始化成特定数据）。故而在这种情况下，**MemSiz** 会大于 **FileSiz**。这也解释了，为什么 C 语言中全局变量会有默认值 0。这是因为操作系统在加载时将所有未初始化的全局变量所占的内存统一填了 0。

VirtAddr 是我们尤为需要注意的。由于它的存在，我们就不难推测，QEMU 模拟器在加载我们的内核时，是按照内核这一可执行文件中所记录的地址，将我们内核中的代码、数据等加载到相应位置。并将 CPU 的控制权交给内核。如果我们内核所处的地址是不正确的，我们的内核将不能够正常运行。换句话说，**只要我们能够将内核加载到正确的位置上，我们的内核就应该可以运行起来。**

思考到这里，我们又发现了几个重要的问题。

1. 什么叫做正确的位置？到底放在哪里才叫正确。
2. 哪个段被加载到哪里是记录在编译器编译出来的 ELF 文件里的，我们怎么才能修改它呢？

在接下来的小节中，我们将一点一点解决这两个问题。

1.3.3 MIPS 内存布局——内核运行的正确位置

在计算机中，启动并运行一个操作系统前，我们需要先将该操作系统内核加载到内存中（参见附录 A.2 真实操作系统的内核及启动详解）。到现在为止，我们已经得到了编译出的操作系统内核文件 `target/mos`，也能够使用 `readelf` 工具去解析内核文件，就差加载这个内核文件了。在这一小节中，我们来解决关于内核应该被放在哪里的问题。在下一小节中，我们再探讨如何加载内核到正确的位置上。

首先我们先来复习地址的概念。

程序中使用的地址与处理器真正发往总线的访存地址往往是不同的，程序中使用的地址一般称作**虚拟地址**（virtual address）、**程序地址**（program address）或者**逻辑地址**，而处理器发往总线的访存地址则称为**物理地址**（physical address）。这种地址转换大部分通过一个在处理器中的叫做 MMU（Memory Management Unit）的硬件模块完成。全部虚拟地址构成了**虚拟地址空间**。对于 32 位处理器，虚拟地址空间的大小一般为 4 GB。

本实验中，MIPS 体系结构的虚拟地址空间大小为 4 GB，并规定了这 4 GB 该如何进行划分。在 MIPS 体系结构中，虚拟地址空间会被划分为 4 个大区域，如图 1.2 所示。

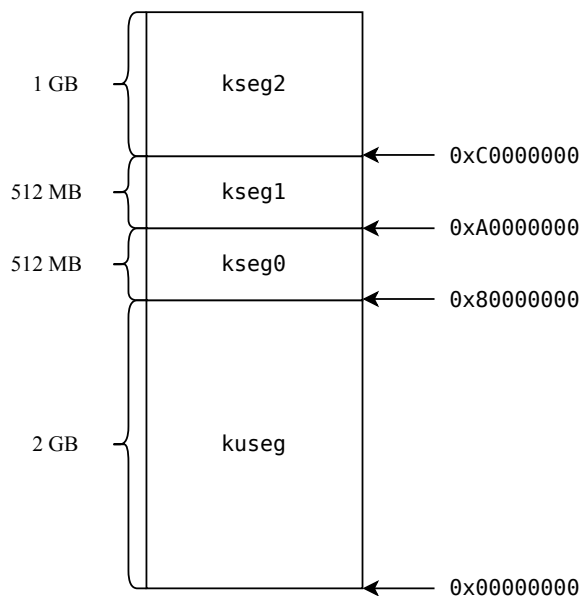


图 1.2: MIPS 内存布局

从硬件角度讲，这四个区域的情况如下：

1. kuseg 0x00000000-0x7FFFFFFF(2 GB): 这一段是用户态下唯一可用的地址空间（内核态下也可使用这段地址空间），大小为 2 GB，也就是 MIPS 约定的用户内存空间。需要通过 MMU（Memory Management Unit）中的 TLB 进行虚拟地址到物理地址的变换。对这段地址的存取都会通过 cache。
2. kseg0 0x80000000-0x9FFFFFFF(512 MB): 这一段是内核态下可用的地址，MMU 将地址的最高位清零（& 0x7fffffff）就得到物理地址用于访存。也就是说，这段的虚拟地址被连续地映射到物理地址的低 512 MB 空间。对这段地址的存取都会通过 cache。
3. kseg1 0xA0000000-0xBFFFFFFF(512 MB): 与 kseg0 类似，这段地址也是内核态下可用的地址，MMU 将虚拟地址的高三位清零（& 0x1fffffff）就得到物理地址用于访存。这段虚拟地址也被连续地映射到物理地址的低 512 MB 空间。但是对这段地址的存取不通过 cache，往往在这段地址上使用 MMIO（Memory-Mapped I/O）技术来访问外设。

4. kseg2 0xC0000000-0xFFFFFFFF(1 GB): 这段地址只能在内核态下使用并且需要 MMU 中 TLB 将虚拟地址转换为物理地址。对这段地址的存取都会通过 cache。

TLB 需要操作系统进行配置管理, 因此在载入内核时, 我们不能选用需要通过 TLB 转换的虚拟地址空间。这样内核就只能放在 kseg0 或 kseg1 了。而 kseg1 是不经过 cache 的, 一般来说, 利用 MMIO 访问外设时才会使用 kseg1。因此我们将内核的 `.text`、`.data`、`.bss` 段都放到 kseg0 中。至于上文提到 kseg0 在 cache 未初始化前不能使用, 但这里我们又将内核放在 kseg0 段, 这是因为在真实的系统中, 运行在 kseg1 中的 bootloader 在载入内核前会进行 cache 初始化工作。

在 `include/mmu.h` 里有我们的 MOS 操作系统内核完整的内存布局图 (代码1.3.3所示), 其中 KERNBASE 是内核镜像的起始虚拟地址。

include/mmu.h 中的内存布局图

```

1  /*
2  o  4G -----> +-----0x10000000
3  o  /      ...      / kseg2
4  o  KSEG2 -----> +-----0xc000 0000
5  o  /      Devices      / kseg1
6  o  KSEG1 -----> +-----0xa000 0000
7  o  /      Invalid Memory      /\
8  o  +-----+-----Physical Memory Max
9  o  /      ...      / kseg0
10 o  KSTACKTOP-----> +-----0x8040 0000-----
11 o  /      Kernel Stack      / | KSTKSIZE      /\
12 o  +-----+-----|
13 o  /      Kernel Text      / |
14 o  KERNBASE -----> +-----0x8002 0000 |
15 o  /      Exception Entry      / \/\
16 o  ULIM -----> +-----0x8000 0000-----
17 o  /      User VPT      / PDMAP      /\
18 o  UVPT -----> +-----0x7fc0 0000 |
19 o  /      pages      / PDMAP      |
20 o  UPAGES -----> +-----0x7f80 0000 |
21 o  /      envs      / PDMAP      |
22 o  UTOP,UENVS -----> +-----0x7f40 0000 |
23 o  UXSTACKTOP -/      /      User exception stack      / PTMAP      |
24 o  +-----+-----0x7f3f f000 |
25 o  /      /      PTMAP      |
26 o  USTACKTOP -----> +-----0x7f3f e000 |
27 o  /      Normal user stack      / PTMAP      |
28 o  +-----+-----0x7f3f d000 |
29 a  /      /
30 a  ~~~~~
31 a  .      .      |
32 a  .      .      kuseg
33 a  .      .      |
34 a  /~~~~~|
35 a  /      /
36 o  UTEXT -----> +-----0x0040 0000 |
37 o  /      reserved for COW      / PTMAP      |
38 o  UCOW -----> +-----0x003f f000 |
39 o  /      reversed for temporary      / PTMAP      |
40 o  UTEMP -----> +-----0x003f e000 |

```

```

41 | o          /      invalid memory      /          \
42 | a      0 -----> +-----+-----+-----+
43 | o
44 | */

```

1.3.4 Linker Script——控制内核加载地址

在发现了内核的正确位置后，我们只需要想办法让内核被加载到那里就可以了。之前在分析 ELF 文件时我们曾看到过，编译器在生成 ELF 文件时就已经记录了各节所需要被加载到的位置。同时，我们也发现，最终的可执行文件实际上是由链接器产生的（它将多个目标文件链接产生最终可执行文件）。因此，我们所需要做的，就是控制链接器的链接过程。

接下来，我们就要引入一个神奇的东西：Linker Script。链接器的设计者们在设计链接器的时候面临这样一个问题：不同平台的 ABI (Application Binary Interface) 都不一样，怎样做才能增加链接器的通用性，使得它能为各个不同的平台生成可执行文件呢？于是，就有了 Linker Script。Linker Script 中记录了各个节应该如何映射到段，以及各个段应该被加载到的位置。下面的命令可以输出默认的链接脚本，你可以在自己的机器上尝试这一条命令：

```
1 ld --verbose
```

这里，我们再补充一下关于 ELF 文件中节 (section) 的概念。在链接过程中，目标文件被看成节的集合，并使用节头表来描述各个节的组织。换句话说，节记录了在链接过程中所需要的必要信息。其中最为重要的三个节为 `.text`、`.data`、`.bss`。这三个节的意义是必须要掌握的：

`.text` 保存可执行文件的操作指令。

`.data` 保存已初始化的全局变量和静态变量。

`.bss` 保存未初始化的全局变量和静态变量。

以上的描述也许会显得比较抽象，这里我们来做一个实验。我们编写一个用于输出代码、全局已初始化变量和全局未初始化变量地址的代码（如代码 1.3.4 所示）。观察其运行结果与 ELF 文件中记录的 `.text`、`.data` 和 `.bss` 节相关信息之间的关系。

用于输出各个节地址的程序

```

1  #include <stdio.h>
2
3  char msg[]="Hello World!\n";
4  int count;
5
6  int main()
7  {
8      printf("%X\n",msg);
9      printf("%X\n",&count);
10     printf("%X\n",main);
11
12     return 0;
13 }

```

该程序的一个可能的输出如下¹。

¹在不同机器上运行，结果也许会有有一定的差异


```

1 user@debian ~/Desktop $ ./program
2 80D4188
3 80D60A0
4 8048AAC

```

我们再看看 ELF 文件中记录的各个节的相关信息（为了突出重点，这里只保留我们所关心的节）。

```

1 共有 29 个节头，从偏移量 0x9c258 开始：
2
3 节头：
4 [Nr] Name                Type                Addr      Off      Size    ES Flg Lk Inf Al
5 [ 4] .text                PROGBITS           08048140  000140  0620e4  00  AX  0  0 16
6 [22] .data                PROGBITS           080d4180  08b180  000f20  00  WA  0  0 32
7 [23] .bss                 NOBITS             080d50c0  08c0a0  00136c  00  WA  0  0 64

```

对比二者，我们就可以清晰的知道 `.text` 包含了可执行文件中的代码，`.data` 包含了需要被初始化的全局变量和静态变量，而 `.bss` 包含了未初始化的全局变量和静态变量

接下来，我们通过 Linker Script 来尝试调整各节的位置。这里，我们选用 [GNU LD 官方帮助文档上的例子](#)，例子的完整代码如下所示：

```

1 SECTIONS
2 {
3     . = 0x10000;
4     .text : { *(.text) }
5     . = 0x8000000;
6     .data : { *(.data) }
7     .bss : { *(.bss) }
8 }

```

在第三行的“.”是一个特殊符号，用来做定位计数器，它根据输出节的大小增长。在 `SECTIONS` 命令开始的时候，它的值为 0。通过设置“.”即可设置接下来的节的起始地址。“*”是一个通配符，匹配所有的相应的节。例如“`.bss : {*(.bss)}`”表示将所有输入文件中的 `.bss` 节（右边的 `.bss`）都放到输出的 `.bss` 节（左边的 `.bss`）中。为了能够编译通过（这个脚本过于简单，难以用于链接真正的程序），我们将原来的实验代码简化如下

```

1 char msg[]="Hello World!\n";
2 int count;
3
4 int main()
5 {
6     return 0;
7 }

```

编译，并查看生产的可执行文件各个节的信息。

```

1 user@debian ~/Desktop $ gcc -o test test.c -T test.lds -nostdlib -m32
2 user@debian ~/Desktop $ readelf -S test
3 共有 11 个节头，从偏移量 0x2164 开始：
4
5 节头：
6 [Nr] Name                Type                Addr      Off      Size    ES Flg Lk Inf Al
7 [ 2] .text                PROGBITS           00010000  001000  000018  00  AX  0  0  1
8 [ 5] .data                PROGBITS           08000000  002000  00000e  00  WA  0  0  1
9 [ 6] .bss                 NOBITS             08000010  00200e  000004  00  WA  0  0  4

```

可以看到，在使用了我们自定义的 Linker Script 以后，生成的程序中，各个节的位置就被调整到了我们所指定的地址上。段是由节组合而成的，节的地址被调整了，那么最终段的地址也会相应地被调整。至此，我们就了解了如何通过 Linker Script 控制各节被加载到的位置。

Note 1.3.5 通过查看内存布局图，同学们应该能找到 `.text` 节的加载地址了，`.data` 和 `.bss` 只需要紧随其后即可。同学们可以思考一下为什么要这么安排 `.data` 和 `.bss`。注意 Linker Script 文件编辑时“=”两边的空格哦！

Exercise 1.2 填写 `kernel.lds` 中空缺的部分，在 Lab1 中，只需要填补 `.text`、`.data` 和 `.bss` 节，将内核调整到正确的位置上即可。 ■

再补充一点：关于链接后的程序从何处开始执行。程序执行的第一条指令的地址称为入口地址 (entrypoint)。我们的实验就在 `kernel.lds` 中通过 `ENTRY(_start)` 来设置程序入口为 `_start`。

Thinking 1.3 在理论课上我们了解到，MIPS 体系结构上电时，启动入口地址为 `0xBFC00000` (其实启动入口地址是根据具体型号而定的，由硬件逻辑确定，也有可能不是这个地址，但一定是一个确定的地址)，但实验操作系统的内核入口并没有放在上电启动地址，而是按照内存布局图放置。思考为什么这样放置内核还能保证内核入口被正确跳转到？
(提示：思考实验中启动过程的两阶段分别由谁执行。) ■

1.4 从零开始搭建 MOS

1.4.1 从 make 开始

在前面的教程中，我们介绍了内核的构建。我们需要先进行编译，然后进行链接，才能得到我们的 MOS 内核。

现在，我们的起点是，在命令行中输入 `make`，输入 `make` 后，具体都发生了什么呢？在前面的章节中我们也简单介绍了我们实验中的 `Makefile`，现在让我们来仔细的阅读 `Makefile` 的代码²。

```
1 targets := $(mos_elf)
2
3 all: $(targets)
4
5 $(modules):
6     $(MAKE) --directory=$@
7
8 $(mos_elf): $(modules)
9     $(LD) $(LDFLAGS) -o $(mos_elf) -N -T $(link_script) $(objects)
```

在控制台执行 `make` 后，`make` 会开始构建 `Makefile` 中的第一个目标 `all`，这时会先构建 `all` 目标的依赖项，也就是 `$(targets)`。而 `targets` 中指定了 `$(mos_elf)`，于是随后会开始构建 `$(mos_elf)` 的依赖项，也就是 `$(modules)` 中的各个目录。

在后面为 `$(modules)` 中的目标定义的配方中，不再有任何依赖项，所以 `make` 开始执行配方，对 `$(modules)` 中的每个目录执行一次 `$(MAKE) --directory=$@`。这里的 `$@` 会被展开为当前目标的名称，在这里就是 `$(modules)` 中的模块名，如 `kern` 等。

²代码中形如 `$(...)` 的变量我们均可以在 `Makefile` 及其依赖的 `include.mk` 中找到

这时，我们便看到了形如 `make[1]: Entering directory '/home/git/xxxxxxx/init'` 的输出，这是在切换工作目录，而 `$(MAKE) --directory=$@` 的执行效果与手动切换到模块目录再调用 `make` 是一致的³。

`mos` 的构建会在完成了所有 `$(modules)` 目标的构建后开始。从代码中可以看出，这里执行了 `$(LD) -o $(mos_elf) -N -T $(link_script) $(objects)`。

在这句指令中，首先是 `$(LD)`，指我们调用了链接器。随后使用的 `-o -T` 参数，可以通过 `ld --help` 查看参数的含义⁴。这句命令的作用是，使用 `$(link_script)` 将 `$(objects)` 链接，输出到 `$(mos_elf)` 位置。

接下来，让我们具体看一看 `$(modules)` 包含的内容。

```

1  modules                := lib init kern
2  targets                := $(mos_elf)
3
4  lab-ge = $(shell [ "$(echo $(lab)_ | cut -f1 -d_)" -ge $(1) ] && echo true)
5
6  ifeq ($(call lab-ge,3),true)
7      user_modules      += user/bare
8  endif
9
10 ifeq ($(call lab-ge,4),true)
11     user_modules      += user
12 endif
13
14 ifeq ($(call lab-ge,5),true)
15     user_modules      += fs
16     targets            += fs-image
17 endif
18
19 objects                := $(addsuffix /*.o, $(modules)) $(addsuffix /*.x, $(user_modules))
20 modules                += $(user_modules)

```

这里，我们可以看到，`$(modules)` 包含了 `lib`、`init` 和 `kern` 三个构建目标，分别对应操作系统的一些依赖库，初始化代码和内核代码。这三个目标的构建都通过前面提到的对 `$(modules)` 的构建来完成，具体来说，`make` 会分别进入这几个目录调用对应的 `Makefile` 完成相应的组件的构建。

而 `$(user_modules)` 则是一个可选的构建目标，它的构建取决于 `lab` 变量的值。具体而言，`$(user_modules)` 默认是空的，但会通过 `lab-ge` 函数判断 `lab` 的值是否大于等于某个值，以此决定是否将某个目标加入 `$(user_modules)` 中。比如，当 `lab` 的值大于等于 5 时，就会将文件系统的代码加入构建。在这部分代码中，`ifeq` 的含义是比较其后指定的两个字符串是否相等，`call` 的含义是调用一个函数，`shell` 的含义是执行一个 shell 命令，`cut` 则是将输入按给定的字符分割。简单来说，这里通过执行一个 shell 命令来将 `lab` 的值中第一个下划线前的内容提取出来并判断是否大于等于给定值，从而决定是否将对应目标加入构建。

最终，我们将组合好的 `$(modules)` 和 `$(user_modules)` 的内容对应的生成文件赋值给 `$(objects)`，用于指定所有依赖的目标文件。在这之后，再将 `$(user_modules)` 加入 `$(modules)`，用于指定所有依赖的构建目标。注意到这里先设置了 `$(objects)`，再设置 `$(modules)`，这是因为用户程序和内核程序的链接方式不同，所以我们需要将用户程序和内核程序的目标文件分为不同的后缀保存，再设置 `$(modules)` 为所有需要依赖的构建目标。这样就可以正常的对各个目标进行构建和生成了。

³对于每个目录内部的 `Makefile`，流程大致相同，对于每个目标，先构建所有依赖项，再执行目标的配方中的每条命令。

⁴这里建议运行我们实验使用的链接器，可以在 `include.mk` 中查看我们使用的链接器的位置。

至此，我们的内核便成功的生成了，我们可以在 `$(mos_elf)` 下查看到我们的内核文件了。这时我们便可以输入命令，运行我们的内核了。

Note 1.4.1 到这里，我们在这一节开始提出的第一个问题就可以得到解答了。内核的入口在哪里呢？让我们回去看一下我们的内核是如何生成的。我们是使用的 `$(link_script)` 来生成的内核。仔细观察 `kernel.lds` 这个文件，我们会发现在其中有 `ENTRY(_start)` 这一行命令，这就是内核的入口了。那么 `_start` 具体是什么呢？我们会在下一个小节具体介绍。

1.4.2 `_start` 函数

理解了 `Makefile` 的工作流程之后，我们再来看 `_start` 函数，下面是 `_start` 函数的代码，我们逐行进行阅读。

```

1  EXPORT(_start)
2  .set at
3  .set reorder
4      /* disable interrupts */
5      mtc0    zero, CP0_STATUS
6
7      /* hint: you can reference the memory layout in include/mmu.h */
8      /* set up the kernel stack */
9      /* Exercise 1.2: Your code here. (1/2) */
10
11     /* jump to mips_init */
12     /* Exercise 1.2: Your code here. (2/2) */

```

Note 1.4.2 本段代码中的 `EXPORT` 是一个宏，它将 `_start` 函数导出为一个符号，使得链接器可以找到它。可以简单的理解为，它实现了一种在汇编语言中的函数定义。关于这些宏的具体实现和功能，可以参阅补充知识部分对 MIPS 汇编和 C 语言部分的介绍。

首先是 `_start` 函数的声明，随后 2、3 行的 `.set` 允许汇编器使用 `at` 寄存器，也允许对接下来的代码进行重排序，第 5 行禁用了外部中断。

接下来需要我们填写的就是比较重要的部分了。我们需要首先将 `sp` 寄存器设置到内核栈空间的位置上，随后跳转到 `mips_init` 函数。内核栈空间的地址可以从 `include/mmu.h` 中看到。这里做一个提醒，请注意栈的增长方向。设置完栈指针后，我们就具备了执行 C 语言代码的条件，因此，接下来的工作就可以交给 C 代码来完成了。所以，在 `_start` 的最后，我们调用 C 代码的主函数，正式进入内核的 C 语言部分。

Exercise 1.3 完成 `init/start.S` 中空缺的部分。设置栈指针，跳转到 `mips_init` 函数。执行命令 `make run` 运行仿真，或使用命令 `make dbg_run` 在调试模式下运行 QEMU。

Note 1.4.3 `mips_init` 函数虽然为 C 语言所书写，但是在被编译成汇编之后，其入口点成为一个符号。

`mips_init` 函数在什么地方？我们是怎么让内核进入到想要的 `mips_init` 函数的呢？又是怎么进行跨文件调用函数的呢？通过 `grep` 搜索，我们可以在 `init/init.c` 中发现 `mips_init` 函数。

我们在汇编程序中，是通过 `jal` 或 `j` 等跳转指令跳转到函数符号对应的地址，也就是指令的存储地址来实现调用函数的。在链接时，链接器会对目标文件中的符号进行**重定位**，使得跳转指令中的地址指向正确的函数，从而实现跨文件的函数调用，因此我们在代码中也可以跨文件使用这些符号。

至此，我们 Lab1 第一部分的内容已经完成，同学们可以稍作休息，认真理解学到的内容。

1.5 实战 printk

了解了这么多的内容后，我们来进行一番实战，在内核中实现一个 `printk` 函数。在平时使用中，你可能觉得 `printf` 函数是由语言本身提供的，其实不是，`printf` 全都是由 C 语言的标准库提供的。而 C 语言的标准库是建立在操作系统基础之上的。所以，当我们开发操作系统的时候，我们就会发现，我们失去了 C 语言标准库的支持。我们需要用到的几乎所有东西，都需要我们自己来实现。

要弄懂内核如何将信息输出到控制台，需要阅读以下三个文件：`kern/printk.c`、`lib/print.c` 和 `kern/machine.c`。

首先大家先对它们的大致内容有个了解：

1. `kern/machine.c`: 这个文件负责往 QEMU 的控制台输出字符，其原理为读写某一个特殊的内存地址。
2. `kern/printk.c`: 此文件中，实现了 `printk`，但其所做的，实际上是把输出字符的函数，接受的输出参数给传递给了 `vprintfmt` 这个函数。
3. `lib/print.c`: 此文件中，实现了 `vprintfmt` 函数，其实现了格式化输出的主体逻辑。

接着为了便于理解，我们一起来梳理一下这几个文件之间的关系。

`kern/printk.c` 中定义了我们的 `printk` 函数。仔细观察就可以发现，这个函数并没有直接实现输出，它只是把接受到的参数，以及 `outputk` 函数指针传入 `vprintfmt` 这个函数中。

```
1 void printk(const char *fmt, ...) {
2     va_list ap;
3     va_start(ap, fmt);
4     vprintfmt(outputk, NULL, fmt, ap);
5     va_end(ap);
6 }
```

同学们可能对 `va_list`、`va_start`、`va_end` 这三个语句以及函数参数中三个点比较陌生。想一想，我们使用 `printk` 时为什么可以有时只输出一个字符串，有时又可以一次输出好多变量呢？实际上，这是 C 语言函数可变长参数的功劳，接下来我们简单介绍一下变长参数的使用方法。

简单来讲，当函数参数列表末尾有省略号时，该函数即有变长的参数表。由于需要定位变长参数表的起始位置，函数需要含有至少一个固定参数，且变长参数必须在参数表的末尾。

`stdarg.h` 头文件中为处理变长参数表定义了一组宏和变量类型如下：

1. `va_list`，变长参数表的变量类型；
2. `va_start(va_list ap, lastarg)`，用于初始化变长参数表的宏；
3. `va_arg(va_list ap, 类型)`，用于取变长参数表下一个参数的宏；
4. `va_end(va_list ap)`，结束使用变长参数表的宏。

在带变长参数表的函数内使用变长参数表前，需要先声明一个类型为 `va_list` 的变量 `ap`，然后用 `va_start` 宏进行一次初始化：

```
1  va_list ap;
2  va_start(ap, lastarg);
```

其中 `lastarg` 为该函数最后一个命名的形式参数。在初始化后，每次可以使用 `va_arg` 宏获取一个形式参数，该宏也会同时修改 `ap` 使得下次被调用将返回当前获取参数的下一个参数。例如

```
1  int num;
2  num = va_arg(ap, int);
```

`va_arg` 的第二个参数为这次获取的参数的类型，如上述代码就从参数列表中取出一个 `int` 型变量。

在所有参数处理完毕后，退出函数前，需要调用 `va_end` 宏以结束变长参数表的使用。

同学们可以参考 <https://zh.cppreference.com/w/c/variadic> 来具体了解变长参数的内容。

然后我们来看看 `outputk` 这个函数，这个函数实际上是用来输出一个字符串的：

```
1  void outputk(void *data, const char *buf, size_t len) {
2      for (int i = 0; i < len; i++) {
3          putcharc(buf[i]);
4      }
5  }
```

可以发现，它实际上调用了—个叫做 `putcharc` 的函数。我们可以在 `kern/machine.c` 下面找到它的定义：

```
1  void putchar(char ch) {
2      .....
3      *((volatile uint8_t *) (KSEG1 + MALTA_SERIAL_DATA)) = ch;
4  }
```

原来，想让控制台输出一个字符，实际上是对某一个内存地址写了一个字节。

看起来输出字符的函数一切正常，那为什么我们还不能使用 `printk` 呢？还记得 `printk` 函数实际上只是把相关信息传入了 `vprintfmt` 函数里面吗？而这个函数现在有一部分代码缺失了，需要你来帮忙补全。

为了方便大家理解这个比较复杂的函数，我们来给大家简单介绍一下。

首先，`vprintfmt` 中定义了一些需要使用到的变量。有几个变量我们需要重点了解其含义：

```
1  int width;      // 标记输出宽度
2  int long_flag;  // 标记是否为 long 型
3  int neg_flag;   // 标记是否为负数
4  int ladjust;    // 标记是否左对齐
5  char padc;      // 填充多余位置所用的字符
```

接下来，我们发现 `vprintfmt` 函数的主体是一个没有终止条件的无限循环，这肯定是不对的，这就是我们需要填补的地方。这个循环中，主要有两个逻辑部分，第一部分：找到格式符 `%`，并分析输出格式；第二部分，根据格式符分析结果进行输出。

什么叫找到格式并分析输出格式呢？想一想，我们在使用 `printf` 输出信息时，`%ld`，`%-b`，`%c` 等等会被替换为相应输出格式的变量的值，他们就叫做格式符。在第一部分中，我们要干的

就是解析 `fmt` 格式字符串，如果是不需要转换成变量的字符，那么就直接输出；如果碰到了格式字符串的结尾，就意味着本次输出结束，停止循环；但是如果碰到我们熟悉的 `%`，那么就要按照 `printf` 的规格要求，开始解析格式符，分析输出格式，用上述变量记录下来这次对变量的输出要求，例如是要左对齐还是右对齐，是不是 `long` 型，是否有输出宽度要求等等，然后进入第二部分。

记录完输出格式，在第二部分中，我们需要做的就是按照格式输出相应的变量的值了。这部分逻辑就比较简单了，先根据格式符进入相应的输出分支，然后取出变长参数中下一个参数，按照输出格式输出这个变量，输出完成后，又继续回到循环开头，重复第一部分，直到整个格式字符串被解析和输出完成。

Exercise 1.4 阅读相关代码和下面对于函数规格的说明，补全 `lib/print.c` 中 `vprintfmt()` 函数中两处缺失的部分来实现字符输出。第一处缺失部分：找到 `%` 并分析输出格式；第二处缺失部分：取出参数，输出格式串为 `%[flags][width][length]<specifier>` 的情况。具体格式详见 [printf 格式具体说明](#)。 ■

Note 1.5.1 这个函数非常重要，希望大家即使评测得到满分，也要多加测试。否则可能出现一些诡异的情况下函数出错，造成后续 lab 输出出错而导致评测结果错误，无法过关。

1.6 实验正确结果

如果你正确地实现了前面所要求的全部内容，你将在 QEMU 中观察到如下输出，这标志着你顺利完成了 Lab1 实验。

```
1 | init.c:      mips_init() is called
```

1.7 任务列表

- Exercise-完成 `readelf.c`
- Exercise-填写 `kernel.lds` 中空缺的部分
- Exercise-完成 `init/start.S`
- Exercise-完成 `vprintfmt()` 函数

1.8 实验思考

- 思考-`objdump` 参数和其它体系结构编译链接过程
- 思考-`readelf` 程序的问题
- 思考-内核入口地址的问题

2.1 实验目的

- 了解 MIPS 4Kc 的访存流程与内存映射布局
- 掌握与实现物理内存的管理方法（链表法）
- 掌握与实现虚拟内存的管理方法（两级页表）
- 掌握 TLB 清除与重填的流程

在 MOS 的设计中，两级页表作为一种内核数据结构放置于内存中，在用户程序需要访问虚拟地址时（通过中断机制）内核负责将对应的页表项填入 TLB。在 Lab2 中，我们的核心任务是管理两级页表与填写 TLB。

MOS 通过两级页表保存和描述虚拟地址与物理地址的映射关系，通过 TLB 加速虚拟地址到物理地址的转换。在本次实验中，我们将实现页表的初始化、页表项的填写、TLB 的清除与重填等功能。

在 Lab2 实现之后，一次 CPU 访存应当经过如下流程

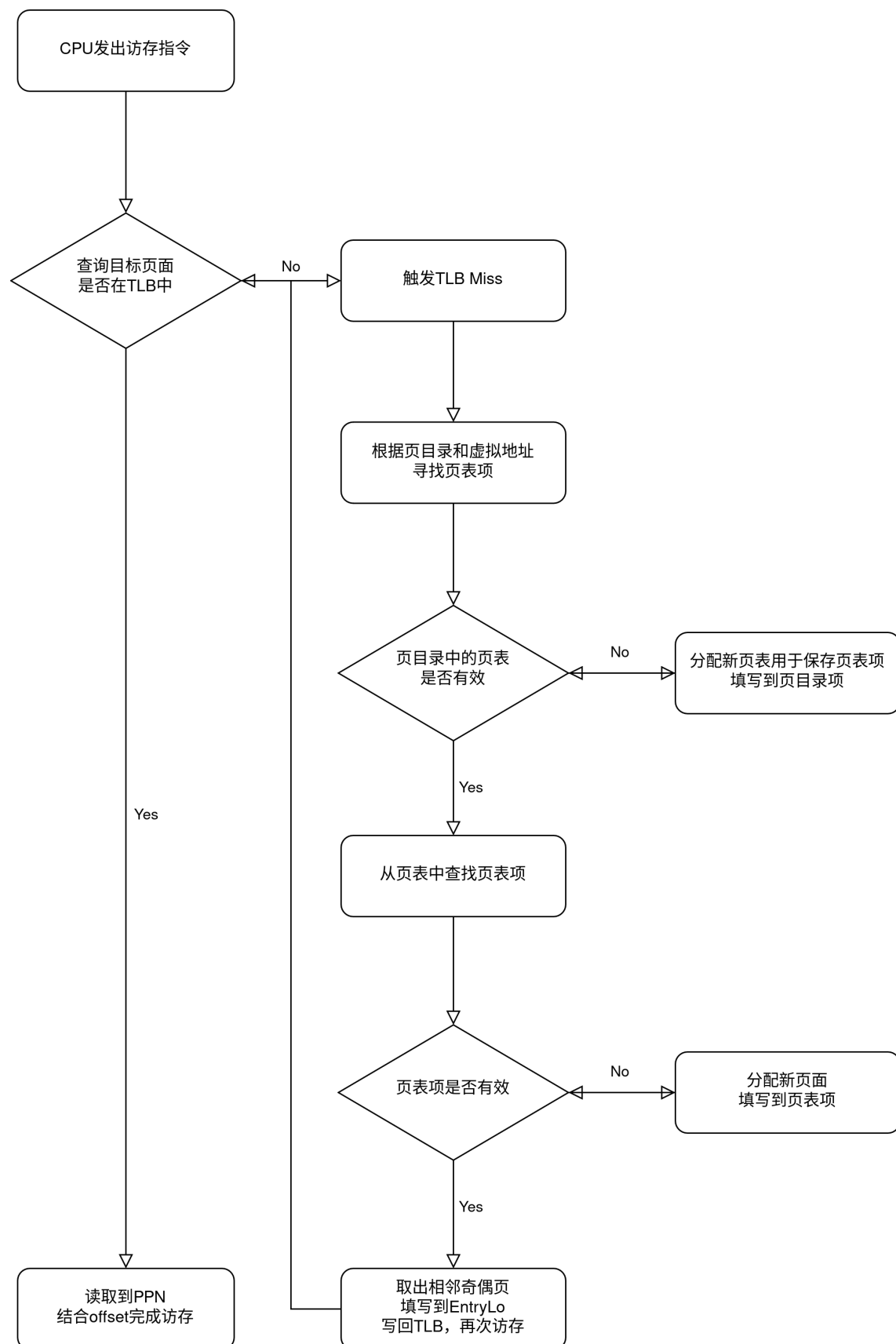


图 2.1: CPU 访存流程

2.2 4Kc 访存流程概览

MIPS 4Kc 是在后续所有实验中所采用的 CPU，了解其各种特性有助于同学们对实验的理解。下面将简略介绍 4Kc 的访存流程。

2.2.1 CPU 发出地址

CPU 运行程序时通过访存指令发出访存请求，进行内存读写操作。在计算机组成原理等硬件实验中，CPU 通常直接发送物理地址，这是为了简化内存操作，让大家关注 CPU 内部的计算与控制逻辑。而在实际程序中，访存、跳转等指令以及用于取指的 PC 寄存器中的访存目标地址都是**虚拟地址**。我们编写的 C 程序中也经常通过对指针解引用来进行访存，其中指针的值也会被视为**虚拟地址**，经过编译后生成相应的访存指令。

Thinking 2.1 请根据上述说明，回答问题：在编写的 C 程序中，指针变量中存储的地址被视为虚拟地址，还是物理地址？MIPS 汇编程序中 `lw` 和 `sw` 指令使用的地址被视为虚拟地址，还是物理地址？

2.2.2 虚拟地址映射到物理地址

在 4Kc 上，软件访存的虚拟地址会先被 MMU 硬件映射到物理地址，随后使用物理地址来访问内存或其他外设。与本实验相关的映射与寻址规则（内存布局）如下：

- 若虚拟地址处于 `0x80000000~0x9fffffff` (`kseg0`)，则将虚拟地址的**最高位置 0** 得到物理地址，通过 cache 访存。**这一部分用于存放内核代码与数据。**
- 若虚拟地址处于 `0xa0000000~0xbfffffff` (`kseg1`)，则将虚拟地址的**最高 3 位置 0** 得到物理地址，不通过 cache 访存。**这一部分可以用于访问外设。**
- 若虚拟地址处于 `0x00000000~0x7fffffff` (`kuseg`)，则需要**通过 TLB 转换成物理地址**，再通过 cache 访存。**这一部分用于存放用户程序代码与数据。**

在 4Kc 中，使用 MMU 来完成上述地址映射，MMU 采用**硬件 TLB** 来完成地址映射。TLB 需要由**软件进行填写**，即操作系统内核负责维护 TLB 中的数据。所有对低 2GB 空间 (`kuseg`) 的内存访问操作都需要经过 TLB。

2.3 内核程序启动

Lab1 已经完成了跳转到 `mips_init` 函数，该函数实现位于 `init/init.c` 中。`mips_init(u_int argc, char **argv, char **penv, u_int ram_low_size)` 函数需要分别调用如下三个函数：

- `mips_detect_memory(u_int _memsize)`，作用是探测硬件可用内存，并对一些和内存管理相关的变量进行初始化。
- `mips_vm_init()`，作用是为内存管理机制作准备，建立一些用于管理的数据结构。
- `page_init()`，实现位于 `kern/pmap.c` 中，作用是初始化 `pages` 数组中的 `Page` 结构体以及空闲链表。这个函数的具体功能会在后面详细描述。

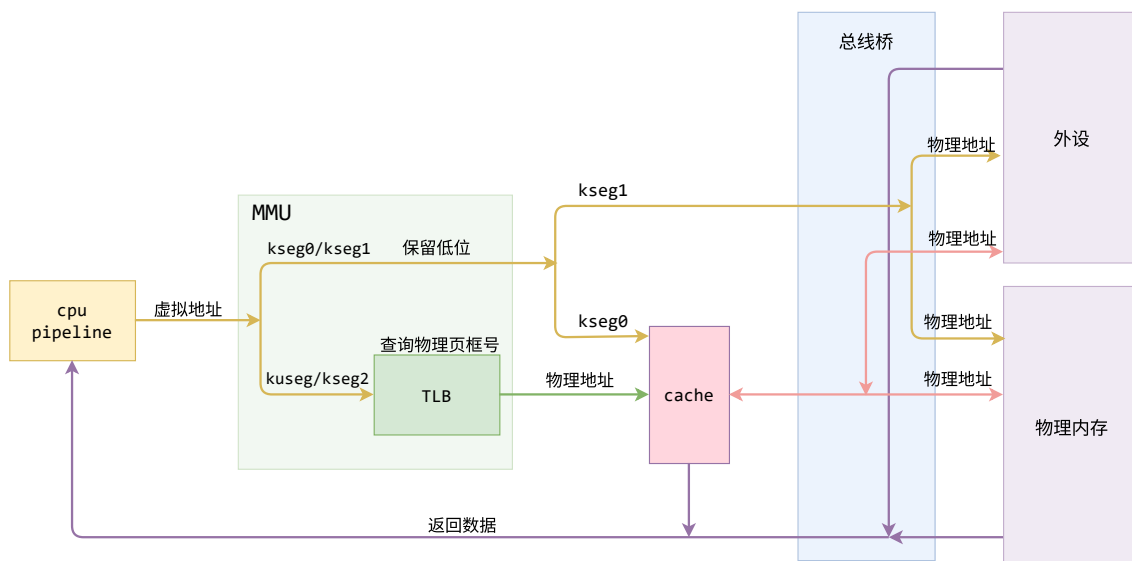


图 2.2: cpu-tlb-memory 关系

Note 2.3.1 `mips_init` 函数中的参数 `u_int argc`, `char **argv`, `char **penv`, `u_int ram_low_size` 是由 `bootloader` 传递给内核的。在 `bootloader` 加载内核，跳转到内核入口处之前，会按照 MIPS ABI 的调用约定，将这些参数填入 `a0-a3` 寄存器供内核使用。其中传递的 `ram_low_size` 参数，指定了硬件可用内存大小。

在接下来的小节中，将重点介绍前两个函数。

2.3.1 mips_detect_memory

`mips_detect_memory()`，实现位于 `kern/pmap.c`，作用是探测硬件可用内存，并对一些和内存管理相关的变量进行初始化。在“开机”之后，操作系统首先会探测硬件的可用内存。`bootloader` 实际上已经帮助我们完成了这件事情，在启动时，系统可用物理内存大小参数通过 `ram_low_size` 被传入，我们只需要直接使用即可。

在该函数中初始化的变量包括：

- `memsize`，表示总物理内存对应的字节数。
- `npage`，表示总物理页数。

Exercise 2.1 请参考代码注释，补全 `mips_detect_memory` 函数。

在实验中，指定了硬件可用内存大小 `memsize`，请你用内存大小 `memsize` 完成总物理页数 `npage` 的初始化。

2.3.2 mips_vm_init 函数

`mips_vm_init()`，实现位于 `kern/pmap.c` 中。在探测完可用内存后，将开始建立内存管理机制。

为了建立起内存管理机制，还需要用到 `alloc` 函数，它同样位于 `kern/pmap.c`。

在没有页式内存管理机制时，操作系统也需要建立一些数据结构来管理内存，这就会涉及到内存空间的分配。`alloc` 函数的功能就是用于分配内存空间（在建立页式内存管理机制之前使用）。

alloc 的实现代码如下:

```

1  static void *alloc(u_int n, u_int align, int clear) {
2      extern char end[];
3      u_long allocated_mem;
4      /* Initialize `freemem` if this is the first time. The first virtual address
5       * that the linker did *not* assign to any kernel code or global variables. */
6      if (freemem == 0) {
7          freemem = (u_long)end; // end
8      }
9      /* Step 1: Round up `freemem` up to be aligned properly */
10     freemem = ROUND(freemem, align);
11     /* Step 2: Save current value of `freemem` as allocated chunk. */
12     allocated_mem = freemem;
13     /* Step 3: Increase `freemem` to record allocation. */
14     freemem = freemem + n;
15     // Panic if we're out of memory.
16     if (PADDR(freemem) >= memsize) {
17         panic("out of memory");
18     }
19     /* Step 4: Clear allocated chunk if parameter `clear` is set. */
20     if (clear) {
21         memset((void *)allocated_mem, 0, n);
22     }
23     /* Step 5: return allocated chunk. */
24     return (void *)allocated_mem;
25 }

```

这段代码的作用是分配 `n` 字节的空间并返回初始的虚拟地址, 同时将地址按 `align` 字节对齐 (保证 `align` 可以整除初始虚拟地址), 若 `clear` 为真, 则将对内存空间的值清零, 否则不清零。

Note 2.3.2 看到这里大家可能有一个疑问: 现在还没有建立内存管理机制, 那么内核是如何操作内存的呢?

考虑 `kseg0` 段的性质, 该段上的虚拟地址被线性映射到物理地址, 操作系统通过访问该段的地址来直接操作物理内存, 从而逐步建立内存管理机制。例如当写虚拟地址 `0x80012340` 时, 由于 `kseg0` 段的性质, 事实上在写物理地址 `0x12340`。

可以回顾前面的[虚拟地址映射到物理地址](#)小节, 其中有提到, 在 `kuseg` 段的虚拟地址才需要通过 MMU 的转换获得物理地址后访存。在 Lab2 中所做的内存管理工作, 正是使将来在 `kuseg` 段的用户程序能够正常工作。

下面通过逐行的解释来了解该函数的实现:

- `extern char end[]` 是一个定义在该文件之外的变量, 并不在一个 C 代码文件中, 而是在 Lab1 填写的 `kernel.lds` 中:

```

1      . = 0x80400000;
2      end = . ;

```

也就是说该变量对应虚拟地址 `0x80400000`, 在建立内存管理机制时, 本实验都是通过 `kseg0` 来访问内存。根据映射规则, `0x80400000` 对应的物理地址是 `0x400000`。

在物理地址 `0x400000` 的前面, 存放着操作系统内核的代码和定义的全局变量或数组 (还额外保留了一些空间)。接下来将从物理地址 `0x400000` 开始分配物理内存, 用于建立管理内存的数据结构。

- `u_long allocated_mem` 是用于存放“已分配的物理内存空间的首地址”的变量。
- `if (freemem == 0) {...}` 语句中, `freemem` 是一个全局变量, 初始值为 0, 第一次调用该函数将其赋值为 `end`。

这个变量的意义是小于 `freemem` 对应物理地址的物理内存都被分配了。

Note 2.3.3 例如, 当初始化 `freemem = 0x80400000` 时, 表示物理地址 `0x400000` 之前的物理内存空间 `[0x0, 0x400000)` 都被分配了。

- `freemem = ROUND(freemem, align)`, `ROUND(a,n)` 是一个定义在 `include/types.h` 的宏, 作用是返回 $\left\lceil \frac{a}{n} \right\rceil n$ (将 `a` 按 `n` 向上对齐), 要求 `n` 必须是 2 的非负整数次幂。在此处 `align` 也必须是一个非负整数次幂。这行代码的含义即找到 `freemem` 之上最小的、按 `align` 对齐的初始虚拟地址, 中间未用到的地址空间全部放弃。实际上是找到了一段空闲的、起始地址与 `align` 对齐的内存空间。

Note 2.3.4 与之相对的, `include/types.h` 还有另一个宏 `ROUNDDOWN(a, n)`, 它的作用是返回 $\left\lfloor \frac{a}{n} \right\rfloor n$ (将 `a` 按 `n` 向下对齐), 同样要求 `n` 必须是 2 的非负整数次幂。

- `allocated_mem = freemem`, 将 `allocated_mem` 赋值为“将要分配的存储空间的起始地址”。例如若 `allocated_mem = 0x80409000`, 意义为将要分配的存储空间的物理地址为 `0x409000`。
- `freemem = freemem + n`, 在前面 `freemem` 已经被设置为“将要分配的存储空间的起始地址”, 因此从此处向后长度为 `n` 的这一段空间即将被分配出去, 分配后需要将“已分配的存储空间的末尾地址”赋值给“将要分配的存储空间的起始地址”。
- `if (PADDR(freemem) >= maxpa) {...}`, 在此处的 `PADDR(x)` 是一个返回虚拟地址 `x` 所对应物理地址的宏, 它定义在 `include/mmu.h`, 该宏要求 `x` 必须是 `kseg0` 中的虚拟地址, 这部分的虚拟地址只需要将最高位清零就可以得到 `kseg0` 段虚拟地址对应的物理地址。这段代码的含义是检查分配的空间是否超出了最大物理地址, 若是则产生报错。

Note 2.3.5 与之相对的, `include/mmu.h` 还有另一个宏 `KADDR(x)`, 作用是返回物理地址 `x` 所位于 `kseg0` 的虚拟地址, 原理同上。

- `if (clear) {...}`, 如果 `clear` 为真, 则使用 `memset` 函数将这一部分内存清零, `memset` 函数的实现位于 `lib/string.c`, 实现方法类似于 `memcpy` 函数。
- `return (void *)allocated_mem`, 最后将初始虚拟地址返回。

2.3.3 mips_vm_init

接着, 回到 `mips_vm_init()` 函数。在函数中, 将完成内存管理数据结构的空间分配。

```
1 void mips_vm_init() {
2     pages = (struct Page *)alloc(npages * sizeof(struct Page), PAGE_SIZE, 1);
3 }
```

`mips_vm_init()` 这个函数使用 `alloc` 函数为物理内存管理所用到的 `Page` 结构体按页分配物理内存, 设 `npages` 个 `Page` 结构体的大小为 `n`, 一页的大小为 `m`, 由上述函数分析可知分配的大小为 $\left\lceil \frac{n}{m} \right\rceil m$ 。

2.4 物理内存管理

MOS 中的内存管理使用页式内存管理，采用链表法管理空闲物理页框。

在实验中，内存管理的代码位于 `kern/pmap.c` 中。函数的声明位于 `include/pmap.h` 中。

2.4.1 链表宏

在后续的 Lab 中，也有一些地方需要用到链表，因此 MOS 中使用宏对链表的操作进行了封装。这部分功能非常有用，设计技巧应用广泛，大家需要仔细阅读代码，深入理解。

链表宏的定义位于 `include/queue.h`，其实现了双向链表功能，下面将对一些主要的宏进行解释：

- `LIST_HEAD(name, type)`，创建一个名称为 `name` 链表的头部结构体，包含一个指向 `type` 类型结构体的指针，这个指针可以指向链表的首个元素。
- `LIST_ENTRY(type)`，作为一个特殊的类型出现，例如可以进行如下的定义：

```
1 LIST_ENTRY(Page) a;
```

它的本质是一个链表项，包括指向下一个元素的指针 `le_next`，以及指向前一个元素链表项 `le_next` 的指针 `le_prev`。`le_prev` 是一个指针的指针，它的作用是当删除一个元素时，更改前一个元素链表项的 `le_next`。

- `LIST_EMPTY(head)`，判断 `head` 指针指向的头部结构体对应的链表是否为空。
- `LIST_FIRST(head)`，将返回 `head` 对应的链表的首个元素。
- `LIST_INIT(head)`，将 `head` 对应的链表初始化。
- `LIST_NEXT(elm, field)`，返回指针 `elm` 指向的元素在对应链表中的下一个元素的指针。`elm` 指针指向的结构体需要包含一个名为 `field` 的字段，类型是一个链表项 `LIST_ENTRY(type)`，下面出现的 `field` 含义均和此相同。
- `LIST_INSERT_AFTER(listelm, elm, field)`，将 `elm` 插到已有元素 `listelm` 之后。
- `LIST_INSERT_BEFORE(listelm, elm, field)`，将 `elm` 插到已有元素 `listelm` 之前。
- `LIST_INSERT_HEAD(head, elm, field)`，将 `elm` 插到 `head` 对应链表的头部。
- `LIST_REMOVE(elm, field)`，将 `elm` 从对应链表中删除。

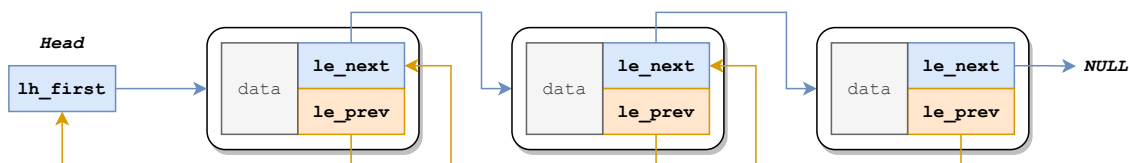


图 2.3: macro list

C++ 中可以使用 `std::stack<T>` 定义一个类型为 `T` 的栈，Java 中可以使用 `HashMap<K,V>` 定义一个键类型为 `K` 且值类型为 `V` 的哈希表。这种模式称为泛型，C 语言并没有泛型的语法，因此需要通过宏另辟蹊径来实现泛型。

Exercise 2.2 完成 `include/queue.h` 中空缺的函数 `LIST_INSERT_AFTER`。

其功能是将一个元素插入到已有元素之后，可以仿照 `LIST_INSERT_BEFORE` 函数来实现。

Thinking 2.2 请思考下述两个问题：

- 从可重用性的角度，阐述用宏来实现链表的好处。
- 查看实验环境中的 `/usr/include/sys/queue.h`，了解其中单向链表与循环链表的实现，比较它们与本实验中使用的双向链表，分析三者插入与删除操作上的性能差异。

2.4.2 页控制块

接着回到物理内存管理，MOS 中维护了 `npage` 个页控制块，也就是 `Page` 结构体。每一个页控制块对应一页的物理内存，MOS 用这个结构体来按页管理物理内存的分配。

Note 2.4.1 `npage` 个 `Page` 和 `npage` 个物理页面一一顺序对应。具体来说，用一个数组存放这些 `Page` 结构体，首个 `Page` 的地址为 `P`，则 `P[i]` 对应从 0 开始计数的第 `i` 个物理页面。`Page` 与其对应的物理页面地址的转换可以使用 `include/pmap.h` 中的 `page2pa` 和 `pa2page` 这两个函数。

Thinking 2.3 请阅读 `include/queue.h` 以及 `include/pmap.h`，将 `Page_list` 的结构梳理清楚，选择正确的展开结构。

```

1  A:
2  struct Page_list{
3      struct {
4          struct {
5              struct Page *le_next;
6              struct Page *le_prev;
7          } pp_link;
8          u_short pp_ref;
9      }* lh_first;
10 }

```

```

1  B:
2  struct Page_list{
3      struct {
4          struct {
5              struct Page *le_next;
6              struct Page **le_prev;
7          } pp_link;
8          u_short pp_ref;
9      } lh_first;
10 }

```

```

1  C:
2  struct Page_list{
3      struct {
4          struct {
5              struct Page *le_next;
6              struct Page **le_prev;
7          } pp_link;
8          u_short pp_ref;
9      } * lh_first;
10 }

```

将空闲物理页对应的 `Page` 结构体全部插入一个链表中,该链表被称为空闲链表,即 `page_free_list`。

当一个进程需要分配内存时,就需要将空闲链表头部的页控制块对应的那一页物理内存分配出去,同时将该页控制块从空闲链表中删去。

当一页物理内存被使用完毕(准确来说,引用次数为 0)时,将其对应的页控制块重新插入到空闲链表的头部。

首先分析 `Page` 结构体,它的定义位于 `include/pmap.h` 中:

```

1  typedef LIST_ENTRY(Page) Page_LIST_entry_t;
2
3  struct Page {
4      Page_LIST_entry_t pp_link; /* free list link */
5
6      // Ref is the count of pointers (usually in page table entries)
7      // to this page. This only holds for pages allocated using
8      // page_alloc. Pages allocated at boot time using pmap.c's "alloc"
9      // do not have valid reference count fields.
10
11      u_short pp_ref;
12 };

```

- `Page_LIST_entry_t` 定义为 `LIST_ENTRY(Page)`, 实际上是一个指向链表项结构体的指针类型, 因此 `pp_link` 即为对应的链表项。
- `pp_ref` 对应这一页物理内存被引用的次数, 它等于有多少虚拟页映射到该物理页。

Note 2.4.2 `pp_ref` 的作用是记录有多少个虚拟页映射到该物理页, 当 `pp_ref` 为 0 时, 表示该物理页没有被引用, 可以被分配出去。

在页式内存管理中, 一个物理页可能被多个虚拟页映射, 这样可以实现内存共享, 例如多个进程共享相同的代码段。

例如在 `fork` 过程中, 子进程的代码段可以与父进程共享, 这样可以节省内存空间, 此时 `pp_ref` 会对应增加, 而 `COW` 页在复制时, 会通过 `unmap` 函数将 `pp_ref` 减少, 以助于系统及时回收内存, 具体内容将会在 Lab4 中详细介绍。

2.4.3 其他相关函数

在 `include/pmap.h` 中可以看到若干个以 `page_` 开头的函数。其中, 有关物理内存管理的函数有 4 个, 分别用来初始化物理页面管理、分配物理页面、减少物理页面引用、回收物理页面到空闲页面链表。

它们的实现均位于 `kern/pmap.c` 中:

- `page_init()`, 它执行了以下操作:

1. 首先利用链表相关宏初始化 `page_free_list`。
2. 将目前已被操作系统内核使用的空间的地址标记进行页对齐。
3. 接着将已使用空间对应的所有物理页面的页控制块的引用次数全部标为 1。
4. 最后将剩下的物理页面的引用次数全部标为 0，并将它们对应的页控制块插入到 `page_free_list`。

启动过程首先初始化了空闲页面链表，用于存储“没有被使用”的页控制块。需要申请内存空间时，就可以从没有被使用的页面中分配一页来使用。

第 2 步需要完成对齐操作，是因为在后续分配内存的时候，都是以页为单位进行分配，所以物理地址需要按页对齐。例如 `0x1000` 到 `0x1fff` 为一页，如果原 `freemem` 的值位于这两个地址之间（确切地说是 `(0x1000, 0x1fff]`），那么此页中 `freemem` 前的空间“已分配”，剩余空间也应该算作已分配的空间。因此为了后续能够正确地按页分配存储空间，这里需要对 `freemem` 进行对齐。

Note 2.4.3 同时，在该函数中也是 `freemem` 变量最后一次被使用。在 `mips_vm_init` 函数执行完毕后，`alloc` 函数就不会再被调用了，在此之后的“分配空间”操作通过 `page_alloc` 函数完成，该函数会在后面进行介绍。

接下来，将使用过的页面标记上引用次数，表示该页面被引用 1 次（可以通过已建立的页控制块数组访问）。将未被使用过的页面的页控制块加入空闲页面链表，供后续分配。

Exercise 2.3 完成 `page_init` 函数。

请按照函数中的注释提示，完成上述操作。此外，这里也给出一些提示：

1. 使用链表初始化宏 `LIST_INIT`。
2. 将 `freemem` 按照 `PAGE_SIZE` 进行对齐（使用 `ROUND` 宏为 `freemem` 赋值）。
3. 将 `freemem` 以下页面对应的页控制块中的 `pp_ref` 标为 1。
4. 将其它页面对应的页控制块中的 `pp_ref` 标为 0 并使用 `LIST_INSERT_HEAD` 将其插入空闲链表。

- `page_alloc(struct Page **pp)`，它的作用是将 `page_free_list` 空闲链表头部页控制块对应的物理页面分配出去，将其从空闲链表中移除，并清空此页中的数据，最后将 `pp` 指向的空间赋值为这个页控制块的地址。

Exercise 2.4 完成 `page_alloc` 函数。

在 `page_init` 函数运行完毕后，在 MOS 中如果想申请存储空间，都需要通过这个函数来申请分配。该函数的逻辑简单来说，可以表述为：

1. 如果空闲链表没有可用页，返回异常返回值。
2. 如果空闲链表有可用的页，取出链表头部的一页；初始化后，将该页对应的页控制块的地址放到调用者指定的地方。

填空时，你可能需要使用链表宏 `LIST_EMPTY` 与函数 `page2kva`。

- `page_decref(struct Page *pp)`，作用是令 `pp` 对应页控制块的引用次数减少 1，如果引

用次数为 0 则会调用 `page_free` 函数将对应物理页面重新设置为空闲页面。

- `page_free(struct Page *pp)`, 它的作用是将 `pp` 指向的页控制块重新插入到 `page_free_list` 中。此外需要先确保 `pp` 指向的页控制块对应的物理页面引用次数为 0。

Note 2.4.4 `page_decref` 和 `page_free` 函数关系密切。当且仅当用户程序主动放弃某个页面、或用户程序执行完毕退出回收所有页面时, 会调用 `page_decref` 来减少页面的引用, 当页面引用被减为 0 时会回收该页面。

Exercise 2.5 完成 `page_free` 函数。

提示: 使用链表宏 `LIST_INSERT_HEAD`, 将页结构体插入空闲页结构体链表。 ■

2.4.4 正确结果展示

执行 `make test lab=2_1 && make run` 编译运行内核, 显示如下信息即通过物理内存管理的单元测试:

```
1 The number in address temp is 1000
2 physical_memory_manage_check() succeeded
3 The number in address temp is 1000
4 physical_memory_manage_check_strong() succeeded
```

2.5 虚拟内存管理

前面已经提到过, MOS 中用 `PADDR` 与 `KADDR` 这两个宏可以对位于 `kseg0` 的虚拟地址和对应的物理地址进行转换。

但是, 对于位于 `kuseg` 的虚拟地址, MOS 中采用两级页表结构对其进行地址转换。

2.5.1 两级页表结构

第一级表称为页目录 (Page Directory), 第二级表称为页表 (Page Table)。为避免歧义, 下面用 **一级页表**指代 Page Directory, **二级页表**指代 Page Table。

两级页表机制相比单级页表机制, 将虚拟页号进一步分为了两部分。

具体来说, 对于一个 32 位的虚存地址, 从低到高从 0 开始编号, 其 31-22 位表示的是一级页表项的偏移量, 21-12 位表示的是二级页表项的偏移量, 11-0 位表示的是页内偏移量。

`include/mmu.h` 中提供了两个宏以快速获取偏移量, `PDX(va)` 可以获取虚拟地址 `va` 的 31-22 位, `PTX(va)` 可以获取虚拟地址 `va` 的 21-12 位。

访问虚拟地址时, 先通过一级页表基地址和一级页表项的偏移量, 找到对应的一级页表项, 得到对应的二级页表的物理页号, 再根据二级页表项的偏移量找到所需的二级页表项, 进而得到该虚拟地址对应的物理页号。

图解流程如下:

MIPS 4Kc 发出的地址均为虚拟地址, 因此如果程序想访问某个物理地址, 需要通过映射到该物理地址的虚拟地址来访问。对页表进行操作时硬件处于内核态, 因此使用宏 `KADDR` 获得其位于 `kseg0` 中的虚拟地址即可完成转换。

在 MOS 中, 无论是一级页表还是二级页表, 它们的结构都一样, 只不过每个页表项记录的物理页号含义有所不同。每个页表均由 1024 个页表项组成, 每个页表项由 32 位组成, 包括 20 位物理页号以及 12 位标志位。其中, 12 位标志位包含高 6 位硬件标志位与低 6 位软件标志位。

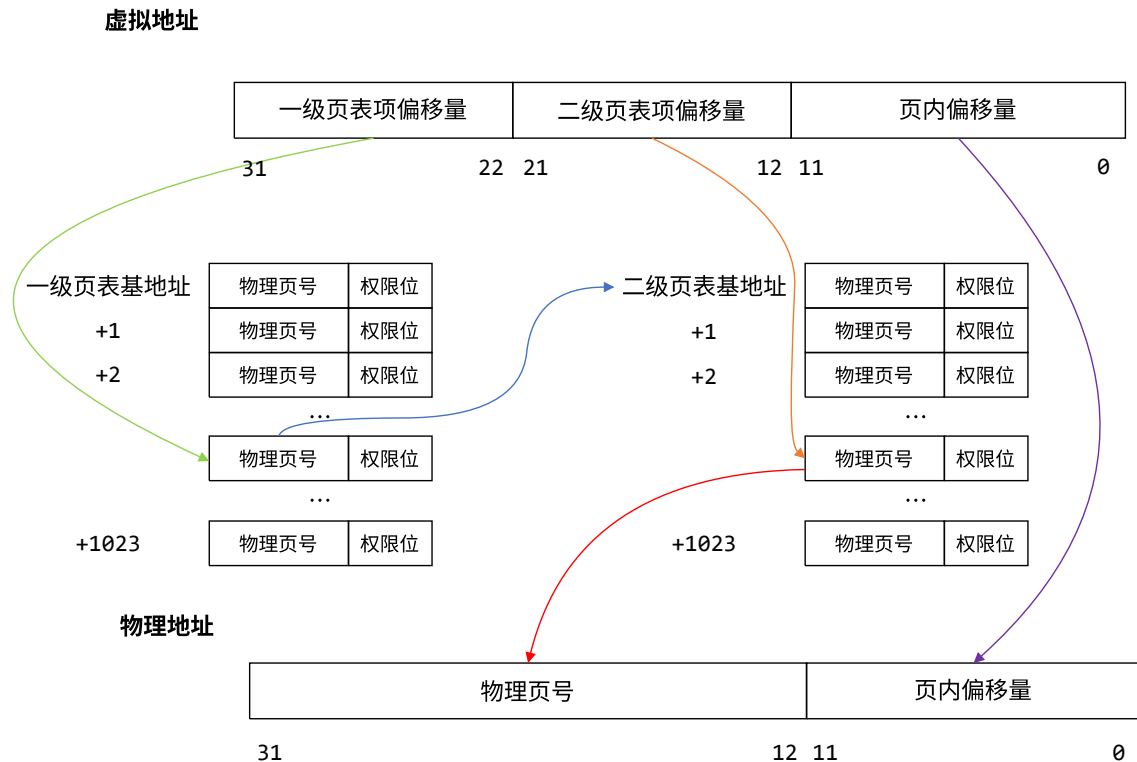


图 2.4: 两级页表结构的地址变换机制

高 6 位硬件标志位用于存入 `EntryLo` 寄存器中, 供硬件 (详见 2.6.1) 使用 (例如标志位 `PTE_V`、`PTE_D` 就分别对应 `EntryLo` 中的 `V`、`D` 标志位)。低 6 位软件标志位不会被存入 TLB 中, 仅供软件使用 (例如标志位 `PTE_COW`、`PTE_LIBRARY` 不对应任何硬件标志位, 仅在页表的部分操作中被操作系统软件利用)。当页表项需要借助 `EntryLo` 寄存器填入 TLB 时, 页表项会被右移 6 位, 移出所有软件标志位, 仅将高 20 位物理页号以及 6 位硬件标志位填入 TLB 使用。每个页表所占的空间为 4KB, 恰好为一个物理页面的大小。

由于一个页表项可以恰好由一个 32 位整型来表示, 因此可以使用 `Pde` 来表示一级页表项类型, 用 `Pte` 来表示二级页表项类型, 这两者的本质都是 `u_long` 类型, 它们的类型别名定义位于 `include/mmu.h`。

例如, 设 `pgdir` 是一个 `Pde *` 类型的指针, 表示一个一级页表的基地址, 那么使用 `pgdir + i` 即可得到偏移量为 `i` 的一级页表项 (页目录项) 地址。

这里再复习一下 TLB 的功能。MOS 中, 通过页表进行地址变换时, 硬件只会查询 TLB, 如果查找失败, 就会触发 TLB Miss (TLB 缺失) 异常, 对应的异常中断处理程序就会对 TLB 进行重填。我们会在 TLB 重填部分的内容中设置一些练习, 所以大家需要了解 MIPS 虚拟地址映射的原理。需要特别注意的是, Lab2 中**没有开启**异常处理功能, 我们仅对 TLB 重填过程进行模拟测试。这部分 TLB 相关的详细内容会在下面介绍。

使用 `tlb_invalidate` 函数可以实现删除特定虚拟地址的映射, 每当页表被修改, 就需要调用该函数以保证下次访问相应虚拟地址时一定触发 TLB 重填, 进而保证访存的正确性。

除此之外, 一般的操作系统中, 当物理页全部被映射 (所有内存空间均被占用), 此时还需要申请新的物理页, 那么就需要将一些在内存中的物理页置换到硬盘中, 选择哪个物理页的算法就称为页面置换算法, 例如 FIFO 算法和 LRU 算法。

然而在 MOS 中, 这一过程的实现被简化, 一旦物理页全部被分配, 进行新的物理页分配时并不会进行任何的页面置换, 而是直接返回错误, 即在对应 `page_alloc` 函数中返回 `-E_NO_MEM`。

2.5.2 与页表相关的函数

下面介绍几个与页表相关的函数。

Note 2.5.1 前面一节中，我们了解到：页表项的 12 位标志位规范和 `EntryLo` 寄存器的规范相同 (见`EntryLo 寄存器`)，起到了访问物理页面时的权限控制作用。

实验中用宏来表示页表项的权限位。注意，这些宏与 `EntryLo` 寄存器中的位存在一定命名差异，比如，`EntryLo` 中的 `D`，实际对应着本实验的页表项权限位 `PTE_D`。

在后续实验中，会进一步了解这些页表项权限，所以这里只介绍一些常用权限位的具体含义。

PTE_V 有效位，若某页表项的有效位为 1，则该页表项有效，其中高 20 位就是对应的物理页号。

PTE_D 可写位，若某页表项的可写位为 1，则允许经由该页表项对物理页进行写操作。

PTE_G 全局位，若某页表项的全局位为 1，则 TLB 仅通过虚页号匹配表项，而不匹配 ASID，将在 Lab3 中用于映射 `pages` 和 `envs` 到用户空间。本 Lab 中可以忽略。

PTE_C_CACHEABLE 可缓存位，配置对应页面的访问属性为可缓存。通常对于所有物理页面，都将其配置为可缓存，以允许 CPU 使用 cache 加速对这些页面的访存请求。

PTE_COW 写时复制位，将在 Lab4 中用到，用于实现 `fork` 的写时复制机制。本 Lab 中可以忽略。

PTE_LIBRARY 共享页面位，将在 Lab6 中用到，用于实现管道机制。本 Lab 中可以忽略。

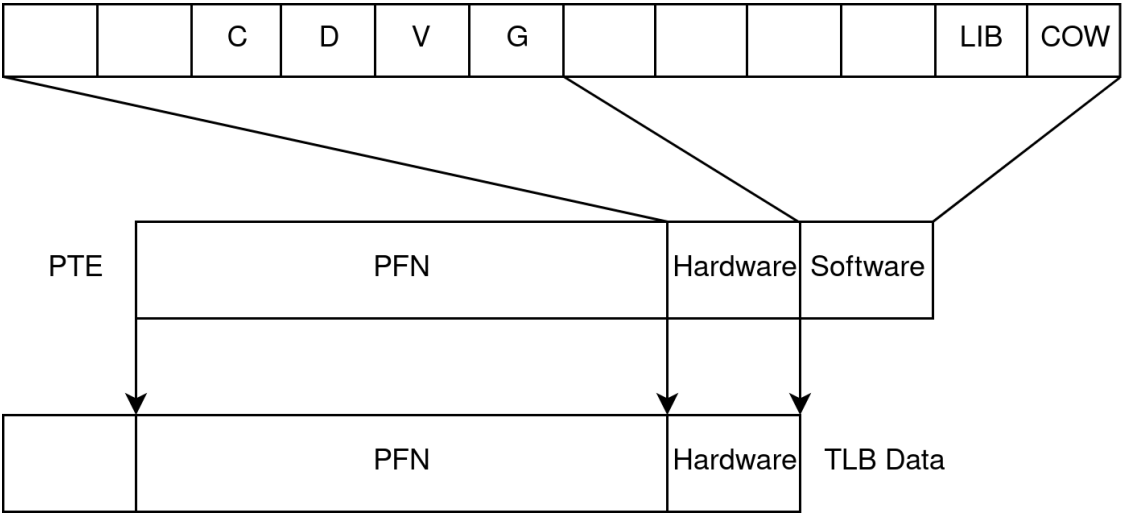
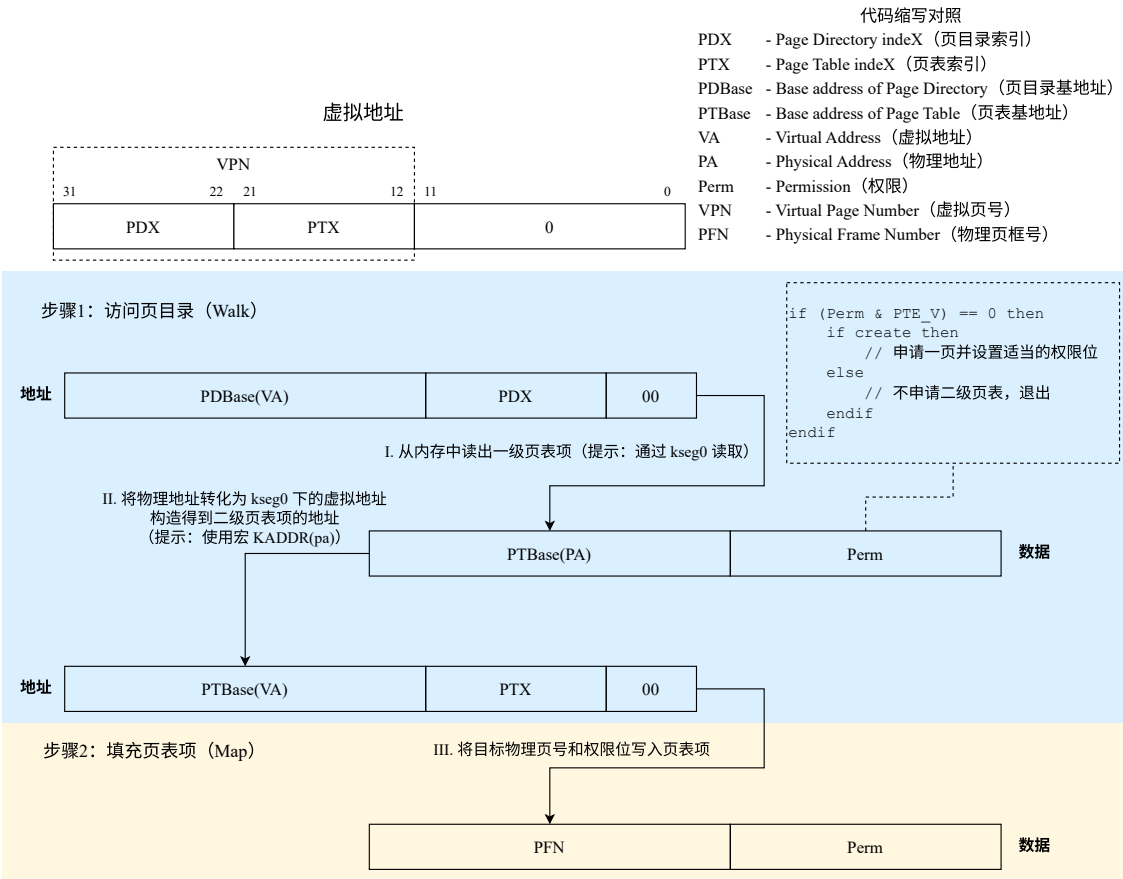


图 2.5: walk & map(insert) PTE 与 TLB Data 映射关系

二级页表检索函数

`int pgdir_walk(Pde *pgdir, u_long va, int create, Pte **ppte)`, 该函数将一级页表基地址 `pgdir` 对应的两级页表结构中 `va` 虚拟地址所在的二级页表项的指针存储在 `ppte` 指向的空间上。如果 `create` 不为 0 且对应的二级页表不存在，则会使用 `page_alloc` 函数分配一页物理内存用于存放二级页表，如果分配失败则返回错误码。

该函数的图解流程如图2.6中的蓝色部分（注意，此图适用于 walk 和 map、insert 系列函数，图中蓝色部分是 walk，黄色部分是 map、insert）。



```

1  int page_insert(Pde *pgdir, u_int asid, struct Page *pp, u_long va, u_int perm) {
2      Pte *pte;
3
4      // 由于需要将 pp 对应页映射到 va, 首先检查是否 va 已经存在了映射?
5      pgdir_walk(pgdir, va, 0, &pte);
6
7      // 如果已经存在了映射, 该映射是否有效?
8      if (pte != 0 && (*pte & PTE_V) != 0) {
9          // 如果存在的映射是有效的, 那现有映射的页和需要插入的页是否一样?
10         if (pa2page(*pte) != pp) {
11             // 如果不一样, 那么先移除现有的映射, 后续再插入新的页。
12             page_remove(pgdir, asid, va);
13         } else {
14             // 如果一样, 就只需要更新一下映射的权限位。
15             // 但先要将 TLB 中缓存的页表项删掉, 然后更新内存中的
16             // 页表项。这样下次加载 va 所在页时, TLB 会重新从页
17             // 表中加载这一页表项。插入完成后函数立即返回。
18             tlb_invalidate(asid, va);
19             *pte = page2pa(pp) | perm | PTE_C_CACHEABLE | PTE_V;
20             return 0;
21         }
22     }
23
24     /*TODO : */
25
26     return 0;
27 }

```

Exercise 2.7 完成 page_insert 函数 (补全 TODO 部分)。

寻找映射的物理地址函数

struct Page * page_lookup(Pde *pgdir, u_long va, Pte **pppte), 作用是返回一级页表基地址 pgdir 对应的两级页表结构中虚拟地址 va 映射的物理页面的页控制块, 同时将 pppte 指向的空间设为对应的二级页表项地址。

```

1  struct Page *page_lookup(Pde *pgdir, u_long va, Pte **pppte) {
2      struct Page *pp;
3      Pte *pte;
4
5      // 首先寻找是否存在这一页表项?
6      pgdir_walk(pgdir, va, 0, &pte);
7
8      // 如果不存在这一页表项, 或页表项无效,
9      // 返回空指针, 代表没有查找到
10     if (pte == NULL || (*pte & PTE_V) == 0) {
11         return NULL;
12     }
13
14     // 如果存在有效的页表项, 则获取这一有效
15     // 页面的页控制块的指针
16     pp = pa2page(*pte);
17
18     // 如果调用方需要这一页表项的地址
19     // (传入了用于传递页表项地址的空间的地址)
20     // 则将页表项地址传递

```

```

21     if (ppte) {
22         *ppte = pte;
23     }
24
25     // 返回查找的页面的页控制块地址
26     return pp;
27 }

```

取消地址映射函数

`void page_remove(Pde *pgdir, u_int asid, u_long va)`, 作用是删除一级页表基地址 `pgdir` 对应的两级页表结构中虚拟地址 `va` 对物理地址的映射。如果存在这样的映射, 那么对应物理页面的引用次数会减少一次。

```

1  void page_remove(Pde *pgdir, u_int asid, u_long va) {
2      Pte *pte;
3      struct Page *pp;
4
5      // 查找 va 对应的页控制块
6      pp = page_lookup(pgdir, va, &pte);
7
8      // 如果查找失败, 说明不存在这一映射,
9      // 不需要取消, 直接返回
10     if (pp == NULL) {
11         return;
12     }
13
14     // 如果查找成功, 则解除其被 va 的映射。
15     page_decref(pp);
16
17     // 将对应的页表项清空
18     *pte = 0;
19
20     // 清空该映射在 TLB 中的缓存
21     tlb_invalidate(asid, va);
22     return;
23 }

```

2.6 访问内存与 TLB 重填

2.6.1 TLB 相关的前置知识

计算机组成原理理论课程中, 会详细介绍 TLB 的功能和硬件实现, 但对 TLB 的填充方法却较少涉及。为了便于理解, 下面先介绍一些 4Kc 的体系结构知识。

4Kc 中与内存管理相关的 CP0 寄存器:

寄存器序号	寄存器名	用途
8	BadVaddr	保存引发地址异常的虚拟地址
10、2、3	EntryHi、EntryLo0、EntryLo1	所有读写 TLB 的操作都要通过这三个寄存器，详见下一小节
0	Index	TLB 读写相关需要用到该寄存器
1	Random	随机填写 TLB 表项时需要用到该寄存器

TLB 组成

每个 TLB 表项都有两个组成部分，包括一组 Key 和两组 Data。

EntryHi、EntryLo0、EntryLo1

EntryHi、EntryLo0、EntryLo1 都是 CP0 中的寄存器，他们只是分别对应到 TLB 的 Key 与两组 Data，并不是 TLB 本身。

其中 EntryLo0、EntryLo1 拥有完全相同的位结构，EntryLo0 存储 Key 对应的偶页面而 EntryLo1 存储 Key 对应的奇页。

Note 2.6.1 4Kc 中的 TLB 采用奇偶页的设计，即使用 VPN 中的高 19 位与 ASID 作为 Key，一次查找到两个 Data（一对相邻页面的两个页表项），并用 VPN 中的最低 1 位在两个 Data 中选择命中的结果。因此在对 TLB 进行维护（无效化、重填）时，除了维护目标页面，同时还需要维护目标页面的邻居页面。

EntryHi、EntryLo0、EntryLo1 的位结构如下：

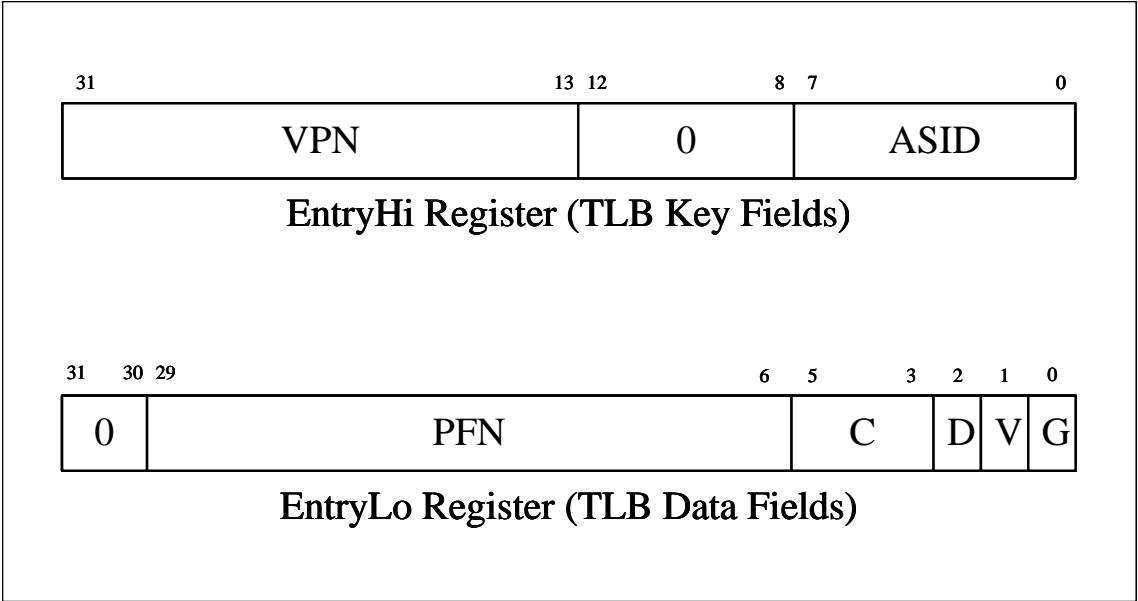


图 2.7: EntryHi & EntryLo

- Key (EntryHi):

- VPN: Virtual Page Number
 - * 当 TLB 缺失 (CPU 发出虚拟地址, TLB 查找对应物理地址但未查到) 时, EntryHi 中的 VPN 自动 (由硬件) 填充为对应虚拟地址的虚页号。
 - * 当需要填充或检索 TLB 表项时, 软件需要将 VPN 段填充为对应的虚拟地址。
- ASID: Address Space Identifier
 - * 用于区分不同的地址空间。查找 TLB 表项时, 除了需要提供 VPN, 还需要提供 ASID (同一虚拟地址在不同的地址空间中通常映射到不同的物理地址)。
- Data (EntryLo):
 - PFN: Physical Frame Number
 - * 软件通过填写 PFN, 接着使用 TLB 写指令, 才可以将此时 EntryHi 中的 Key 与 EntryLo 中的 Data 写入 TLB。
 - C: Cache Coherency Attributes 标识页面的 cache 访问属性。PTE_C_CACHEABLE 与 PTE_C_UNCACHEABLE 宏可用于填充该段。
 - D: Dirty。事实上是**可写位**。当该位为 1 时, 对应的页可写; 否则对相应页的任何写操作都将引发 TLB 异常。
 - V: Valid。如果该位为 0, 则任何访问对应页的操作都将引发 TLB 异常。
 - G: Global。如果该位为 1, 则 CPU 发出的虚拟地址只需要与该表项的 VPN 匹配, 即可与此 TLB 项匹配成功 (不需要检查 ASID 是否匹配)。

Note 2.6.2 TLB 事实上构建了一个映射 $\langle \text{VPN}, \text{ASID} \rangle \xrightarrow{\text{TLB}} \langle \text{PFN}, \text{N}, \text{D}, \text{V}, \text{G} \rangle$ 。

Note 2.6.3 TLB 组成细节

在 4Kc 中, 一个完整的 TLB 由 Joint TLB (JTLB)、Instruction micro TLB (ITLB) 和 Data micro TLB (DTLB) 三部分组成。其中 JTLB 是 TLB 的主要组成单元, 我们的操作系统也只需要关注并维护 JTLB 和内核页表的一致性; ITLB 与 DTLB 则由硬件维护它们与 JTLB 之间的一致性, 我们无需过多关注。

在 4Kc 硬件设计上下文中, 如果没有特殊说明, 指导书提到的 TLB 均指 JTLB。

Thinking 2.4 请思考下面两个问题:

- 请阅读上面有关 TLB 的描述, 从虚拟内存和多进程操作系统的实现角度, 阐述 ASID 的必要性。
- 请阅读 MIPS 4Kc 文档《MIPS32® 4K™ Processor Core Family Software User's Manual》的 Section 3.3.1 与 Section 3.4, 结合 ASID 段的位数, 说明 4Kc 中可容纳不同的地址空间的最大数量。

TLB 相关指令

- tlbwr: 以 Index 寄存器中的值为索引, 读出 TLB 中对应的表项到 EntryHi 与 EntryLo0、EntryLo1。

- **tlbwi**: 以 **Index** 寄存器中的值为索引, 将此时 **EntryHi** 与 **EntryLo0**、**EntryLo1** 的值写到索引指定的 TLB 表项中。
- **tlbwr**: 将 **EntryHi** 与 **EntryLo0**、**EntryLo1** 的数据随机写到一个 TLB 表项中 (此处使用 **Random** 寄存器来“随机”指定表项, **Random** 寄存器本质上是一个不停运行的循环计数器)。
- **tlbp**: 根据 **EntryHi** 中的 Key (包含 VPN 与 ASID), 查找 TLB 中与之对应的表项, 并将表项的索引存入 **Index** 寄存器 (若未找到匹配项, 则 **Index** 最高位被置 1)。

(内核) 软件操作 TLB 的流程

软件必须经过 CP0 与 TLB 交互, 因此软件操作 TLB 的流程总是分为两步:

1. 填写 CP0 寄存器。
2. 使用 TLB 相关指令。

2.6.2 TLB 维护流程

通过之前的实验, 大家可能仍然对软件访问内存的过程有所疑惑, 这是由于还未涉及到用户进程相关的内容, 所有代码、数据的虚拟地址均在 **kseg0** 段, 无需通过页表的翻译便可直接获得其物理地址。因此本次实验中所完成的代码, 大多是为之后的实验提供接口, 本次实验中的一些内存管理功能只作为独立的函数存在。但由于内存访问将是之后实验中很重要的内容, 在此次实验结束时, 有必要将用户进程访问内存的流程解释清楚, 既能帮助之后的实验, 也能加深对本次实验的理解。

本实验所使用的 MIPS 4Kc 的 MMU 硬件中只有 TLB, 在用户地址空间访存时, 虚拟地址到物理地址的转换均通过 TLB 进行。访问需要经过转换的虚拟内存地址时, 首先要使用虚拟页号和当前进程的 ASID 在 TLB 中查询该地址对应的物理页号, 如果虚页号和 ASID 组成的 Key 在 TLB 中存在对应的 TLB 表项 (或虚页号在 TLB 中存在对应的 TLB 表项且表项权限位中的 G 位为 1) 时, 则可取得物理地址; 如果不能查询到, 则产生 TLB Miss 异常, 系统跳转到异常处理程序, 在内核的两级页表结构中找到对应的物理地址, 对 TLB 进行重填。

操作系统可以修改页表中虚拟地址映射的物理页号或映射的权限位。如果 TLB 中已暂存了页表中某一虚拟地址对应的页表项内容, 之后操作系统更新了该页表项, 但没有更新 TLB, 则访问该虚拟地址时实际可能会访问到错误的物理页面。所以我们需要维护 TLB 的表项, 使得当 TLB 能够查询到虚拟地址相应的页号时, 取得的物理页号和权限信息与实际在内核页表中对应的数据一致。

具体来说, 维护 TLB 的流程如下:

1. 更新页表中虚拟地址对应的页表项的同时, 将 TLB 中对应的旧表项无效化
2. 在下一访问该虚拟地址时, 硬件会触发 TLB 重填异常, 此时操作系统对 TLB 进行重填

TLB 旧表项无效化

我们通过位于 **kern/tlbex.c** 中的 **tlb_invalidate** 函数实现删除特定虚拟地址在 TLB 中的旧表项, 函数的主要逻辑依靠位于 **kern/tlb_asm.S** 中的 **tlb_out** 函数。当 **tlb_out** 被 **tlb_invalidate** 调用时, **a0** 寄存器中存放着传入的参数, 其值为旧表项的 Key (由虚拟页号和 ASID 组成)。我们使用 **mtc0** 指令将 Key 写入 **EntryHi**, 随后使用 **tlbp** 指令, 根据 **EntryHi** 中

的 Key 查找对应的旧表项，将表项的索引存入 Index。如果索引值大于等于 0（即 TLB 中存在 Key 对应的表项），我们向 EntryHi 和 EntryLo0、EntryLo1 中写入 0，随后使用 tlbwi 指令，将 EntryHi 和 EntryLo0、EntryLo1 中的值写入索引指定的表项。此时旧表项的 Key 和 Data 被清零，实现将其无效化。

Exercise 2.8 完成 kern/tlb_asm.S 中的 tlb_out 函数。该函数根据传入的参数（TLB 的 Key）找到对应的 TLB 表项，并将其清空。

具体来说，需要在两个位置插入两条指令，其中一个位置为 tlbpi，另一个位置为 tlbwi。因流水线设计架构原因，tlbpi 指令的前后都应各插入一个 nop 以解决数据冒险。 ■

Thinking 2.5 请回答下述三个问题：

- tlb_invalidate 和 tlb_out 的调用关系？
- 请用一句话概括 tlb_invalidate 的作用。
- 逐行解释 tlb_out 中的汇编代码。

TLB 重填

TLB 的重填过程由 kern/tlb_asm.S 中的 do_tlb_refill 函数完成。由于 4Kc 中存在的奇偶页设计，该过程需重填触发异常的页面，及其邻居页面。将两个页面对应的页表项先写入 EntryLo 寄存器，再填入 TLB。

```

1  NESTED(do_tlb_refill, 0, zero)
2  .....
3  addi    sp, sp, -24 /* Allocate stack. */
4  sw      ra, 20(sp) /* [sp + 20] - [sp + 23] store return address. */
5  addi    a0, sp, 12 /* [sp + 12] - [sp + 19] store return value. */
6  jal     _do_tlb_refill /* (Pte *, u_int, u_int) */
7  lw      a0, 12(sp) /* Return value 0 - Even page table entry */
8  lw      a1, 16(sp) /* Return value 1 - Odd page table entry */
9  lw      ra, 20(sp) /* Return address */
10 addi    sp, sp, 24 /* Deallocate stack */
11 mtc0    a0, CPO_ENTRYLO0 /* Even page table entry */
12 mtc0    a1, CPO_ENTRYLO1 /* Odd page table entry */
13 nop
14 /* Hint: use 'tlbwr' to write CPO.EntryHi/Lo into a random tlb entry. */
15 tlbwr
16 jr      ra
17 END(do_tlb_refill)

```

TLB 重填的相关流程在介绍两级页表时已经有所了解，具体的 do_tlb_refill 流程大致如下：

1. 从 BadVAddr 中取出引发 TLB 缺失的虚拟地址。
2. 从 EntryHi 的 0 - 7 位取出当前进程的 ASID。在 Lab3 的代码中，会在进程切换时修改 EntryHi 中的 ASID，以标识访存所在的地址空间。
3. 先在栈上为返回地址、待填入 TLB 的页表项以及函数参数传递预留空间，并存入返回地址。以存储奇偶页表项的地址、触发异常的虚拟地址和 ASID 为参数，调用 _do_tlb_refill

函数。该函数是 TLB 重填过程的核心，其功能是根据虚拟地址和 ASID 查找页表，将对应的奇偶页表项写回其第一个参数所指定的地址。

4. 将页表项存入 EntryLo0、EntryLo1，并执行 tlbwr 将此时的 EntryHi 与 EntryLo0、EntryLo1 写入到 TLB 中（在发生 TLB 缺失时，EntryHi 已经由硬件写入了虚拟页号等信息，无需修改）。

我们将操作页表的逻辑使用 C 语言封装在 kern/tlbex.c 中的 _do_tlb_refill() 函数中：

```

1 void _do_tlb_refill(Pte *pte, u_int va, u_int asid) {
2     tlb_invalidate(asid, va);
3     Pte *ppte;
4
5     /* 提示：
6      尝试在循环中调用 'page_lookup' 以查找虚拟地址 va
7      在当前进程页表中对应的页表项 *ppte'
8      如果 'page_lookup' 返回 'NULL'，表明 *ppte 找不到，使用 'passive_alloc'
9      为 va 所在的虚拟页面分配物理页面，
10     直至 'page_lookup' 返回不为 'NULL' 则退出循环。
11
12     你可以在调用函数时，使用全局变量 cur_pgdir 作为其中一个实参。
13     */
14
15     ppte = (Pte *)((u_long)ppte & ~0x7);
16     pte[0] = ppte[0] >> 6;
17     pte[1] = ppte[1] >> 6;
18 }

```

TLB 中重填时的对应关系：

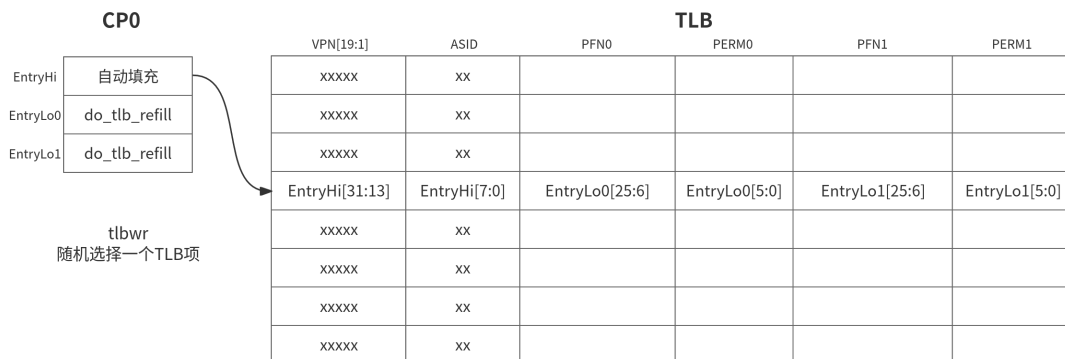


图 2.8: TLB refill

cur_pgdir 是一个在 kern/pmap.c 定义的全局变量，其中存储了当前进程一级页表基地址位于 kseg0 的虚拟地址。

Note 2.6.4 在 Lab3 中，内核会在进程切换时更改 cur_pgdir 的值，使之始终指向当前进程的一级页表基地址。

当 page_lookup() 函数在页表中找不到对应表项时，我们调用 passive_alloc() 函数进行处理。若该虚拟地址合法，我们可以为此虚拟地址申请一个物理页面 (page_alloc)，并将虚拟地址映射到该物理页面 (page_insert)，即进行被动页面分配。

Exercise 2.9 完成 `kern/tlbex.c` 中的 `_do_tlb_refill` 函数。 ■

Exercise 2.10 完成 `kern/tlb_asm.S` 中的 `do_tlb_refill` 函数。 ■

Note 2.6.5 在之后的 Lab 中，运行 MOS 时容易出现的 `too low` 等错误信息就是访问了一个过低的地址导致的。此时应该检查代码中是否存在访问非法内存（如空指针、野指针）的操作，或者忘记将物理地址转化为 `kseg0` 内核虚拟地址的问题。

中断、异常处理等过程将是之后实验的重点，现在可暂且将其理解为代码的跳转，到本节实验为止只需明白 4Kc 中软件代码在访问内存时的处理过程即可。

2.6.3 正确结果展示

在完成虚拟内存管理后，执行 `make test lab=2_2 && make run` 编译运行内核，显示如下信息即通过虚拟内存管理的单元测试（其中地址 `3ffe000` 在不同实现下的值可能不同）：

```
1  va2pa(boot_pgdir, 0x0) is 3ffe000
2  page2pa(pp1) is 3ffe000
3  start page_insert
4  pp2->pp_ref 0
5  end page_insert
6  page_check() succeeded!
7  va2pa(boot_pgdir, 0x0) is 3ffe000
8  page2pa(pp1) is 3ffe000
9  start page_insert
10 end page_insert
11 page_check_strong() succeeded!
```

在完成 TLB 重填练习后，执行 `make test lab=2_3 && make run` 编译运行内核，显示如下信息即通过 TLB 重填的单元测试：

```
1  tlb_refill_check() begin!
2  test point 1 ok
3  test point 2 ok
4  tlb_refill_check() succeed!
```

2.7 Lab2 在 MOS 中的概况

下图展示了 Lab2 中填写的内容在 MOS 系统中的作用（紫色部分就是 Lab2 为系统提供的功能）。图中每一根竖线都代表一个执行流程，竖线上的点代表一个步骤（或函数调用），点的旁边会注明该步骤（或函数）的名称。如果某一个步骤（或函数）包含（或调用）若干个重要的子步骤（或函数），则在该步骤（或函数）的名称下以“竖线 + 点”的形式表达。比如：`passive_alloc` 按顺序调用了 `page_alloc` 和 `page_insert` 这两个函数，而 `page_insert` 又调用了 `tlb_invalidate` 这个函数。

在图中，左侧主干是内核初始化的流程，内核初始化完毕后陷入死循环。同学们完成 Lab3 后可以知道，第一次时钟中断来临后用户进程开始运作，而在用户进程运行过程中，会遇到中断和异常（如图中右侧主干）而陷入内核，调用相应的中断异常处理函数。

Thinking 2.6 请结合 Lab2 开始的 CPU 访存流程与下图中的 Lab2 用户函数部分，尝试将函数调用与 CPU 访存流程对应起来，思考函数调用与 CPU 访存流程的关系。 ■

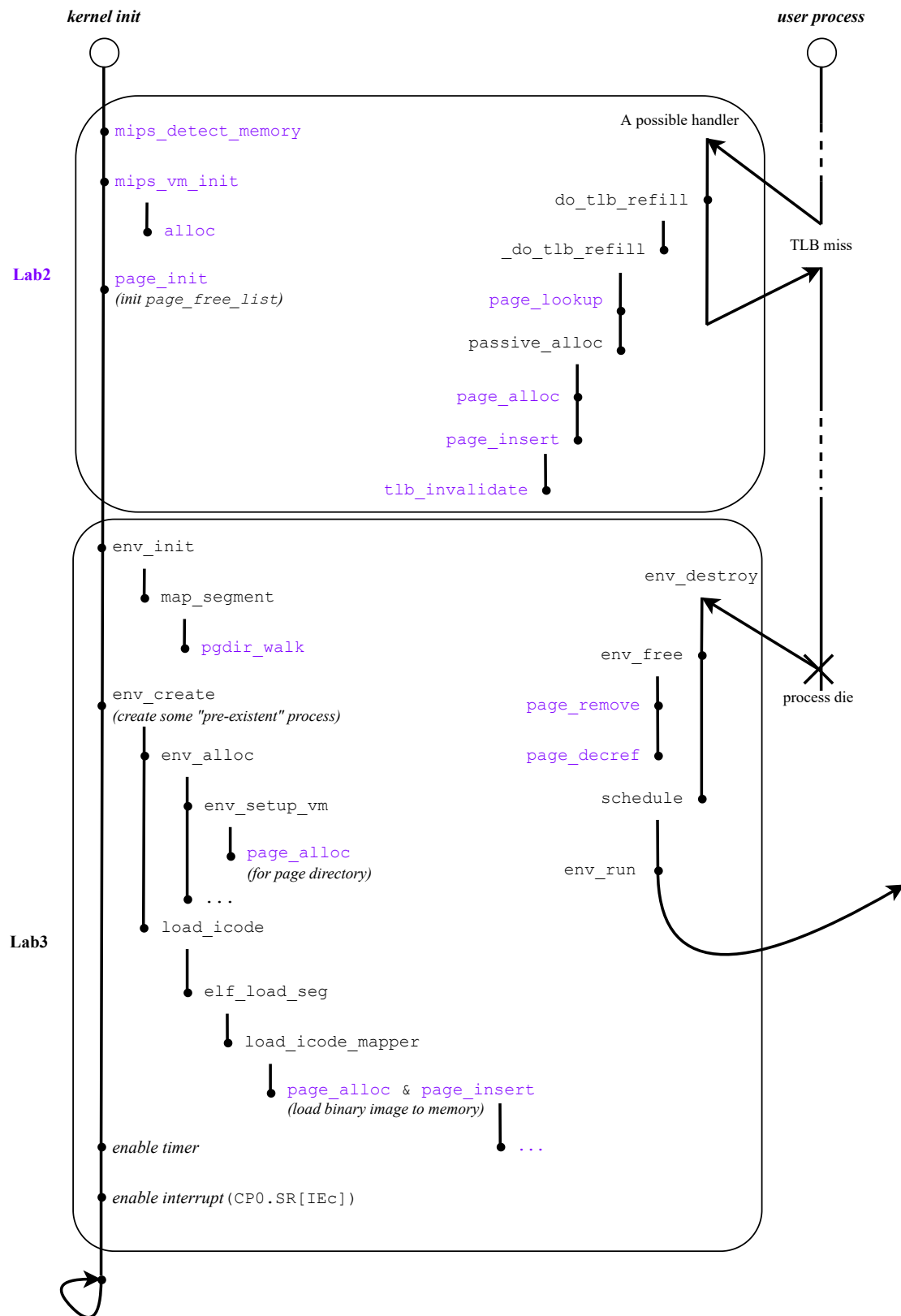


图 2.9: Lab2 in MOS

2.8 其他体系结构中的内存管理

本实验在 MIPS 体系结构上构建操作系统，因此内存管理机制与 MIPS 特性耦合紧密。在其他的一些体系结构中，内存管理可能会和 MIPS 有很大的差别。有一些体系结构有着复杂的 MMU，直接使用硬件机制来填写 TLB，而不像本实验中依赖软件代码来填写 TLB。

Thinking 2.7 从下述三个问题中任选其一回答：

- 简单了解并叙述 X86 体系结构中的内存管理机制，比较 X86 和 MIPS 在内存管理上的区别。
- 简单了解并叙述 RISC-V 中的内存管理机制，比较 RISC-V 与 MIPS 在内存管理上的区别。
- 简单了解并叙述 LoongArch 中的内存管理机制，比较 LoongArch 与 MIPS 在内存管理上的区别。

2.9 任务列表

- Exercise-完成 `mips_detect_memory` 函数
- Exercise-完成 `queue.h`
- Exercise-完成 `page_init` 函数
- Exercise-完成 `page_alloc` 函数
- Exercise-完成 `page_free` 函数
- Exercise-完成 `pgdir_walk` 函数
- Exercise-完成 `page_insert` 函数
- Exercise-完成 `tlb_out` 函数
- Exercise-完成 `_do_tlb_refill` 函数
- Exercise-完成 `do_tlb_refill` 函数

2.10 实验思考

- 思考-程序代码中的地址
- 思考-链表宏
- 思考-`Page_list` 结构体
- 思考-地址空间
- 思考-`tlb_invalidate`
- 思考-三级页表自映射
- 思考-函数调用与 CPU 访存流程
- 思考-其他体系结构的内存管理

CHAPTER 3

进程与异常

3.1 实验目的

1. 创建一个进程并成功运行
2. 实现时钟中断，通过时钟中断内核可以再次获得执行权
3. 实现进程调度，创建两个进程，并且通过时钟中断切换进程执行

在 Lab3 中将运行一个用户模式的进程。

本实验需要使用数据结构进程控制块 `Env` 来跟踪用户进程，并建立一个简单的用户进程，加载一个程序镜像到指定的内存空间，然后让它运行起来。

同时，实验实现的 MIPS 内核具有处理异常的能力。

3.2 进程

Note 3.2.1 由于没有实现线程，本实验中进程既是基本的分配单元，也是基本的执行单元。每个进程都是一个实体，有其自己的地址空间，通常包括代码段、数据段和堆栈。程序是一个没有生命的实体，只有被操作系统赋予生命时，它才能成为一个活动的实体，而执行中的程序，就是进程。

3.2.1 进程控制块

进程控制块 (PCB) 是系统专门设置用来管理进程的数据结构，它可以记录进程的外部特征，描述进程的变化过程。系统利用 PCB 来控制和管理进程，所以 **PCB 是系统感知进程存在的唯一标志。进程与 PCB 是一一对应的**。在 MOS 中，PCB 由一个 `Env` 结构体实现，主要包含如下一些信息：

进程控制块

```
1 struct Env {  
2     struct Trapframe env_tf;    // saved context (registers) before switching  
3     LIST_ENTRY(Env) env_link;    // intrusive entry in 'env_free_list'
```

```

4      u_int env_id;           // unique environment identifier
5      u_int env_parent_id;    // env_id of this env's parent
6      u_int env_status;       // status of this env
7      Pde *env_pgdir;         // page directory
8      TAILQ_ENTRY(Env) env_sched_link; // intrusive entry in 'env_sched_list'
9      u_int env_pri;          // schedule priority
10 };

```

为了将注意力集中在关键的地方，暂时不对后续实验用到的域做介绍。下面是 **Lab3** 相关部分的简单说明：

- **env_tf**：用于保存当前进程的上下文信息对应的结构体 **Trapframe**，该结构体的定义在 **include/trap.h** 中，在发生进程调度，或当陷入内核时，会将当时的进程上下文环境保存在 **env_tf** 变量中。
- **env_link**：**env_link** 的机制类似于 Lab2 中的 **pp_link**，使用它来构造空闲进程链表 **env_free_list**。
- **env_id**：每个进程的 **env_id** 都不一样，它是进程独一无二的标识符。
- **env_parent_id**：在之后的实验中，我们将了解到进程是可以被其他进程创建的，创建本进程的进程称为父进程。此变量记录父进程的进程 id，进程之间通过此关联可以形成一棵进程树。
- **env_status**：该字段只能有以下三种取值：
 - **ENV_FREE**：表明该进程控制块处于**空闲状态**，其没有被任何进程使用，即该进程控制块处于进程空闲链表中。
 - **ENV_NOT_RUNNABLE**：表明该进程处于**阻塞状态**，处于该状态的进程需要在一定条件下变成就绪状态从而被 CPU 调度。(比如因进程通信阻塞时变为 **ENV_NOT_RUNNABLE**，收到信息后变回 **ENV_RUNNABLE**)
 - **ENV_RUNNABLE**：该进程处于**执行状态或就绪状态**，即其可能是正在运行的，也可能正在等待被调度。
- **env_pgdir**：这个字段保存了该进程页目录的内核虚拟地址。
- **env_sched_link**：这个字段用来构造调度队列 **env_sched_list**;
- **env_pri**：这个字段保存了该进程的优先级。

在实验中，存放进程控制块的物理内存存在系统启动后就已经分配好，就是 **envs** 数组。

和 Lab2 对页控制块数组的管理类似，我们使用链表管理进程控制块数组。**struct Env** 中的链表项共涉及调度队列 **env_sched_list** 和空闲队列 **env_free_list** 两个队列：

- **env_sched_list**：为了便于管理已经被分配的进程控制块和对进程的调度，我们用链表将这些控制块串联起来，形成 **env_sched_list**。在进程创建时需要为其分配进程控制块并加入 **env_sched_list**，在进程被释放时需要将其对应的进程控制块从 **env_sched_list** 移出。
- **env_free_list**：为了快速分配空闲的进程控制块，我们需要像 Lab2 一样将空闲的 **Env** 控制块按照链表形式串联起来，形成 **env_free_list**。一开始所有进程控制块都是空闲的，所以要把它们都串联到 **env_free_list** 上去。

其中 `env_free_list` 使用了 Lab2 中涉及的 `LIST` 结构, 而 `env_sched_list` 则使用了相似但不同的另一种结构 `TAILQ`, 它同样定义在 `include/queue.h` 中, 实现了一个双端队列, 既支持在头部插入和取出, 也支持在尾部插入和取出。你需要使用以 `TAILQ` 开头的宏来操作该队列。

Exercise 3.1 完成 `env_init` 函数。

实现 `Env` 控制块的空闲队列和调度队列的初始化功能。请注意, 你需要按倒序将所有控制块插入到空闲链表的头部, 使得编号更小的进程控制块被优先分配。

Note 3.2.2 这里规定的链表插入顺序, 是为了方便进行单元测试, 请按规定的次序实现。

3.2.2 段地址映射

在 `env_init` 函数的最后, 使用 `page_alloc` 函数为模板页表 `base_pgdir` 分配了一页物理内存, 将其转换为内核虚拟地址, 并使用 `map_segment` 函数在该页表中将内核数组 `pages` 和 `envs` 映射到了用户空间的 `UPAGES` 和 `UENVS` 处。在之后的 `env_setup_vm` 函数中, 我们会将这部分模板页表复制到每个进程的页表中。

段地址映射函数 `void map_segment(Pde *pgdir, u_int asid, u_long pa, u_long va, u_long size, u_int perm)`, 功能是在一级页表基地址 `pgdir` 对应的两级页表结构中做段地址映射, 将虚拟地址段 `[va, va+size)` 映射到物理地址段 `[pa, pa+size)`, 因为是按页映射, 要求 `size` 必须是页面大小的整数倍。同时为相关页表项的权限为设置为 `perm`。它在这里的作用是将内核中的 `Page` 和 `Env` 数据结构映射到用户地址, 以供用户程序读取。

Exercise 3.2 请你结合 `env_init` 中的使用方式, 完成 `map_segment` 函数。

3.2.3 进程的标识

现代计算机系统中经常有很多进程同时存在, 每个进程执行不同的任务, 它们之间也经常需要相互协作、通信, 那么操作系统是如何识别每个进程呢?

在本实验中我们依靠进程标识符来实现这一点。`struct Env` 进程控制块中的 `env_id` 域, 是每个进程独一无二的标识符, 需要在进程创建的时候就被赋予。

`env.c` 文件中实现了一个叫做 `mkenvid` 的函数, 作用就是生成一个新的进程 `env_id`。

MOS 系统的进程控制块, 还有一个 `env_asid` 域记录进程的 `ASID`, 这是进程虚拟地址空间的标识。

`env_id` 已经可以唯一标识进程, 包括进程虚拟地址空间, 为什么 MOS 还需要引入额外的域作为虚拟地址空间的标识?

系统中并发执行多个拥有不同虚拟地址空间的进程, 具有不同的页表。而 CPU 的 MMU 使用 TLB 缓存虚拟地址映射关系, 不同页表拥有不同虚拟地址映射, 显然, 不同进程的虚拟地址可以对应相同的虚拟页号。当 CPU 切换页表, TLB 中仍可能缓存有之前页表的虚拟地址映射关系, 这些映射关系可能与当前页表所描述的不一致, 可以将其称为无效映射关系。为了避免 TLB 缓存的无效映射关系在页表切换后导致错误的地址翻译发生, 早期操作系统实现在 CPU 每次切换页表时, 无效化所有 TLB 表项。

但这种实现会导致频繁的 TLB Miss, 影响处理器性能。现代的 CPU 及操作系统, 采用 `ASID` 解决上述问题。`ASID` 用于标识虚拟地址空间, 同时并发执行的多个进程具有不同 `ASID` 以方便 TLB 标识其虚拟地址空间。

正如 Lab 2 中提到, TLB 实质上构建了一个 $\langle \text{VPN}, \text{ASID} \rangle \xrightarrow{\text{TLB}} \langle \text{PFN}, \text{N}, \text{D}, \text{V}, \text{G} \rangle$ 的映射。这里的 VPN 表示虚拟页号, PFN 表示物理页框号。TLB 存储进程的 ASID, 作为 Key 的一部分, 用于区别不同的虚拟地址空间中的映射。相当于每一个进程, 都有自己虚拟地址空间 (使用 ASID 标识) 下的一套独立 TLB 缓存, 每次切换页表后, 操作系统不必再清空所有 TLB 表项。

由于 ASID 也具有唯一标识性, 在实验中直到进程被销毁或 TLB 被清空时, 其 ASID 才可以被分配给其他进程。MOS 实验模拟 CPU 的型号是 MIPS 4Kc, 其 TLB 结构的 Key Fields (也就是 EntryHi 寄存器) 中用到了 ASID, 如图 2.7 所示。

可以看出, 其中 ASID 部分只占据了 0-7 共 8 个 bit, 也就是说 ASID 资源是有限的, 需要使用一定的资源管理方法来分配、回收 ASID。MOS 实验采用了位图法管理 256 个可用的 ASID, 如果 ASID 耗尽时仍要创建进程, 内核会发生崩溃 (panic)。可以参考 env.c 中 asid_alloc 函数的代码来理解实现过程。

Note 3.2.3 实际的 Linux 系统中通过 ASID 分代机制来使得同时运行的进程数不受硬件 ASID 位数的限制, 感兴趣的同学可以在课下了解一下 Linux 的实现机制。

3.2.4 设置进程控制块

完成练习 3.1 后, 可以开始利用空闲进程链表 env_free_list 创建进程。下面具体介绍如何创建一个进程。

进程创建的流程如下:

第一步 申请一个空闲的 PCB (也就是 Env 结构体), 从 env_free_list 中索取一个空闲 PCB 块, 这时候的 PCB 就像张白纸一样。

第二步 “纯手工打造” 打造一个进程。在这种创建方式下, 由于没有模板进程, 所以进程拥有的所有信息都是手工设置。而进程的信息又都存放于进程控制块中, 所以需要手工初始化进程控制块。

第三步 进程光有 PCB 的信息还没法跑起来, 每个进程都有独立的地址空间。所以, 要为新进程初始化页目录。

第四步 此时 PCB 已经被填写了很多东西, 不再是一张白纸, 把它从空闲链表里摘出, 就可以使用。

Note 3.2.4 用户栈是在使用时被动态分配的。

第二步的信息设置是本次实验的关键, 那么下面结合注释看看这段代码

进程创建

```

1  int env_alloc(struct Env **new, u_int parent_id) {
2      int r;
3      struct Env *e;
4
5      /* Step 1: Get a free Env from 'env_free_list' */
6
7      /* Step 2: Call a 'env_setup_vm' to initialize the user address space
8       * for this new Env.
9       */
10

```

```

11  /* Step 3: Initialize these fields for the new Env with appropriate values:
12  *   'env_user_tlb_mod_entry' (lab4), 'env_runs' (lab6),
13  *   'env_id' (lab3), 'env_asid' (lab3), 'env_parent_id' (lab3)
14  *
15  * Hint:
16  *   Use 'asid_alloc' to allocate a free asid.
17  *   Use 'mkenvid' to allocate a free envid.
18  */
19  e->env_user_tlb_mod_entry = 0; // for lab4
20  e->env_runs = 0; // for lab6
21
22  /* Step 4: Initialize the sp and 'cp0_status' in 'e->env_tf'.
23  *   Set the EXL bit to ensure that the processor remains in kernel mode
24  *   during context recovery. Additionally, set UM to 1 so that when ERET
25  *   unsets EXL, the processor transitions to user mode.
26  *   Timer interrupt will be enabled by setting (STATUS_IM7 | STATUS_IE).
27  */
28  e->env_tf.cp0_status = STATUS_IM7 | STATUS_IE | STATUS_EXL | STATUS_UM;
29  // Reserve space for 'argc' and 'argv'.
30  e->env_tf.regs[29] = USTACKTOP - sizeof(int) - sizeof(char **);
31
32  /* Step 5: Remove the new Env from env_free_list. */
33
34  *new = e;
35  return 0;
36  }

```

env.c 中的 `env_setup_vm` 函数就是在第二步中要使用的函数，该函数的作用是初始化新进程的地址空间，这部分任务是 Lab3 的难点之一。

地址空间初始化

```

1  /* Overview:
2  *   Initialize the user address space for 'e'.
3  */
4  static int env_setup_vm(struct Env *e) {
5      /* Step 1:
6      *   Allocate a page for the page directory with 'page_alloc'.
7      *   Increase its 'pp_ref' and assign its kernel address to 'e->env_pgdir'.
8      *
9      * Hint:
10     *   Use 'page2kva' to get the kernel address of a specified physical page.
11     */
12     struct Page *p;
13     try(page_alloc(&p));
14     /* Exercise 3.3: Your code here. */
15
16     /* Step 2: Copy the template page directory 'base_pgdir' to 'e->env_pgdir'. */
17     /* Hint:
18     *   As a result, the address space of all envs is identical in [UTOP, UVPT).
19     *   See include/mmu.h for layout.
20     */
21     memcpy(e->env_pgdir + PDX(UTOP), base_pgdir + PDX(UTOP),
22           sizeof(Pde) * (PDX(UVPT) - PDX(UTOP)));
23
24     /* Step 3: Map its own page table at 'UVPT' with readonly permission.
25     *   As a result, user programs can read its page table through 'UVPT' */

```

```

26     e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_V;
27     return 0;
28 }

```

在开始完成 `env_setup_vm` 之前，为了解这个函数中实现的功能，请先阅读以下提示：

在实验中，虚拟地址 `ULIM` 是 `kseg0` 与 `kuseg` 的分界线，是系统给用户进程分配的最高地址。`ULIM` 以上的地方，`kseg0` 和 `kseg1` 两部分内存的访问不经过 TLB，这部分内存由内核管理、所有进程共享。在 MOS 操作系统特意将一些内核的数据暴露到用户空间，使得进程不需要切换到内核态就能访问，这是 MOS 特有的设计。在 Lab4 和 Lab6 中将用到此机制。而这里我们要暴露是 `UTOP` 往上到 `UVPT` 之间所有进程共享的只读空间，也就是把这部分内存对应的内核页表 `base_pgdir` 拷贝到进程页表中。从 `UVPT` 往上到 `ULIM` 之间则是进程自己的页表。

Exercise 3.3 完成 `env_setup_vm` 函数。

仔细阅读前文的提示理解一个进程虚拟地址空间的分布，根据注释完成函数，实现初始化一个新进程地址空间的功能。

Thinking 3.1 这里为什么要拷贝内核页表 `base_pgdir`？拷贝之后的目标页需不需要修改引用次数？

Thinking 3.2 请结合 MOS 中的页目录自映射应用解释代码中 `e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_V` 的含义。

在练习 3.3 完成后，我们就可以直接在 `env_alloc` 第二步使用 `env_setup_vm` 了。我们需要初始化的字段已经在 `env_alloc` 函数的注释中给出，这个函数的重点在于我们已经给出的这个赋值 `e->env_tf.cp0_status = STATUS_IM7 | STATUS_IE | STATUS_EXL | STATUS_UM`。这个赋值很重要，因此我们必须直接在代码中给出。下面介绍它的具体作用。

Figure 6-12 Status Register Format

31	28	27	26	25	24	23	22	21	20	19	18	17	16	15			8	7	6	5	4	3	2	1	0
CU3..CU0	RP	FR	RE	MX	PX	BEV	TS	SR	NMI	0	Impl	IM7..IM0					KX	SX	UX	UM	R0	ERL	EXL	IE	
																				KSU					

图 3.1: 4Kc 的 Status 寄存器示意图

图 3.1 是我们 MIPS 4Kc 里的 Status 寄存器示意图，就是我们在 `env_tf` 里的 `cp0_status`。

`IE` 位表示中断是否开启，为 1 表示开启，否则不开启。我们需要将 `IE` 及 `IM7` 设置为 1，表示中断使能，且 7 号中断（时钟中断）可以被响应。

此外，4Kc 中 Status 寄存器的 `EXL` 以及 `UM` 表示了处理器当前的运行状态。

当且仅当 `EXL` 被设置为 0 且 `UM` 被设置为 1 时，处理器处于用户模式，其它所有情况下，处理器均处于内核模式下。每当异常发生的时候，`EXL` 会被自动设置为 1，并由异常处理程序负责后续处理。

而每当执行 `eret` 指令时，`EXL` 会被自动设置为 0。我们现在先不管为何，但是已经知道，下面这一段代码是每个进程在每一次被调度时都会执行的，其中 `RESTORE_ALL` 是一个宏，用于完成对处理器寄存器状态的恢复，`TF` 中记录的 Status 寄存器值将会在 `RESTORE_ALL` 中被写入处理器的 Status 寄存器。在寄存器恢复全部完成后，`eret` 这条指令会被执行。


```

1  RESTORE_ALL
2  eret

```

现在你可能就明白为何我们要设置 Status 寄存器中的 UM 和 EXL 位了。如果不设置 Status 寄存器中的 EXL 位，在 RESTORE_ALL 中恢复 Status 寄存器后，处理器将立即进入用户模式，不再处于特权模式。此时，再访问内核地址段，执行特权指令，处理器会立刻陷入异常，不符合我们的预期。当运行进程前，运行上述代码到 eret 的时候，就会将 EXL 设置为 0。此时 Status 中的 UM、IE 均已被设置为 1，表示在用户模式下且开启中断。之后第一个进程成功以用户模式运行，这时操作系统也可以正常响应中断了。

当然我们从注释也能看出，第四步除了需要设置 cp0_status 以外，还需要设置栈指针。在 MIPS 中，栈寄存器是第 29 号寄存器，注意这里的栈是用户栈，不是内核栈。

Exercise 3.4 完成 env_alloc 函数。

env_alloc 函数实现了申请并初始化一个进程控制块的功能。这里给出如下提示：

1. 回忆 Lab2 中的链表宏 LIST_FIRST、LIST_REMOVE，实现在 env_free_list 中申请空闲进程控制块。
2. 用 env_setup_vm 初始化新进程的地址空间。
3. 仔细阅读前文中对与 Lab3 相关的域的介绍，思考相关域的恰当赋值。

3.2.5 加载二进制镜像

在这一小节中，我们需要将程序加载到新进程的地址空间中。

在 Lab1 中我们曾详细学习了 ELF 文件，它的类型有三种，一是可重定位文件，二是可执行文件，三是可被共享的对象文件。本节中的程序是指可执行文件。

要想正确加载一个 ELF 文件到内存，只需将 ELF 文件中所有需要加载的程序段 (program segment) 加载到对应的虚拟地址上即可。我们已经写好了用于解析 ELF 文件的代码中的大部分内容，你可以直接调用相应函数获取 ELF 文件的各项信息，并完成加载过程。相关函数和类型的声明如下：

```

1  // lib/elfloader.c
2  const Elf32_Ehdr *elf_from(const void *binary, size_t size);
3  int elf_load_seg(Elf32_Phdr *ph, const void *bin,
4    elf_mapper_t map_page, void *data);
5
6  // kern/env.c
7  static void load_icode(struct Env *e, const void *binary, size_t size);
8  static int load_icode_mapper(void *data, u_long va, size_t offset,
9    u_int perm, const void *src, size_t len);

```

Note 3.2.5 在 Lab3 阶段，我们还没有实现文件系统，因此无法直接操作磁盘中的 ELF 文件。在这里我们已经将 ELF 文件内容转化成了 C 数组的形式（可以在 make 后查看 user/bare/loop.b.c 文件），这样可以通过编译到内核中完成加载。

load_icode 函数负责加载可执行文件 binary 到进程 e 的内存中。它调用的 elf_from 函数完成了解析 ELF 文件头的部分，elf_load_seg 负责将 ELF 文件的一个 segment 加载到内存。

为了达到这一目标, `elf_load_seg` 的最后两个参数用于接受一个自定义的回调函数 `map_page`, 以及需要传递给回调函数的额外参数 `data`。每当 `elf_load_seg` 函数解析到一个需要加载到内存中的页面, 会将有关的信息作为参数传递给回调函数, 并由它完成单个页面的加载过程, 而这里 `load_icode_mapper` 就是 `map_page` 的具体实现。

`load_icode` 函数会从 ELF 文件中解析出每个 segment 的段头 `ph`, 以及其数据在内存中的起始位置 `bin`, 再由 `elf_load_seg` 函数将参数指定的程序段 (program segment) 加载到进程的地址空间中。

`elf_load_seg` 函数会从 `ph` 中获取 `va` (该段需要被加载到的虚地址)、`sgsize` (该段在内存中的大小)、`bin_size` (该段在文件中的大小) 和 `perm` (该段被加载时的页面权限), 并根据这些信息完成以下两个步骤:

第一步 加载该段的所有数据 (`bin`) 中的所有内容到内存 (`va`)。

第二步 如果该段在文件中的内容的大小达不到为填入这段内容新分配的页面大小, 即分配了新的页面但没能填满 (如 `.bss` 区域), 那么余下的部分用 0 来填充。

如图 3.2 所示, 段内的内存布局可能较为复杂, 需加载到的虚拟地址 `va`、该段占据的内存长度 `sg_size` 以及需要拷贝的数据长度 `bin_size` 都可能不是页对齐的。`elf_load_seg` 会正确处理这些地址的页面偏移, 对于每个需要加载的页面, 用对齐后的地址 `va` 以及该页的其他信息调用回调函数 `map_page`, 由回调函数完成单页的加载。这样的设计允许 `elf_load_seg` 只关心 ELF 段的结构, 而不用处理与具体操作系统相关的页面加载过程。

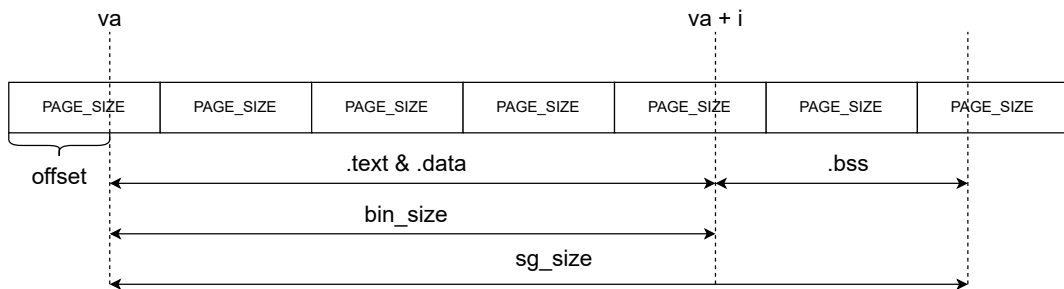


图 3.2: 每个 segment 的加载地址布局

为了简化实现难度, 你只需要完成作为回调函数的 `load_icode_mapper` 函数, 其中需要分配所需的物理页面, 并在页表中建立映射。若 `src` 非空, 你还需要将该处的 ELF 数据拷贝到物理页面中。

Exercise 3.5 完成 `kern/env.c` 中的 `load_icode_mapper` 函数。

提示: 可能使用到的函数有 `page_alloc`, `page_insert`, `memcpy`。

Thinking 3.3 `elf_load_seg` 以函数指针的形式, 接受外部自定义的回调函数 `map_page`。请你找到与之相关的 `data` 这一参数在此处的来源, 并思考它的作用。没有这个参数可不可以? 为什么?

Thinking 3.4 结合 `elf_load_seg` 的参数和实现, 考虑该函数需要处理哪些页面加载的情况。

现在我们已经完成了加载 ELF 镜像的核心函数，那么下面我们完成这个函数后，就能真正实现把 ELF 加载到进程的任务了。

完整加载镜像

```

1 static void load_icode(struct Env *e, const void *binary, size_t size) {
2     /* Step 1: Use 'elf_from' to parse an ELF header from 'binary'. */
3     const Elf32_Ehdr *ehdr = elf_from(binary, size);
4     if (!ehdr) {
5         panic("bad elf at %x", binary);
6     }
7
8     /* Step 2: Load the segments using 'ELF_FOREACH_PHDR_OFF' and 'elf_load_seg'.
9      * As a loader, we just care about loadable segments, so parse only program
10     * headers here.
11     */
12     size_t ph_off;
13     ELF_FOREACH_PHDR_OFF (ph_off, ehdr) {
14         Elf32_Phdr *ph = (Elf32_Phdr *) (binary + ph_off);
15         if (ph->p_type == PT_LOAD) {
16             // 'elf_load_seg' is defined in lib/elfloader.c
17             // 'load_icode_mapper' defines the way in which a page in this segment
18             // should be mapped.
19             panic_on(elf_load_seg(ph, binary + ph->p_offset, load_icode_mapper, e));
20         }
21     }
22
23     /* Step 3: Set 'e->env_tf.cp0_epc' to 'ehdr->e_entry'. */
24     /* Exercise 3.6: Your code here. */
25
26 }

```

这个函数通过调用 `elf_load_seg` 函数来将 ELF 文件真正加载到内存中，将 `load_icode_mapper` 这个函数作为参数传入。

Exercise 3.6 根据注释的提示，完成 `kern/env.c` 中的 `load_icode` 函数。

这里的 `env_tf.cp0_epc` 字段指示了进程恢复运行时 PC 应恢复到的位置。我们要运行的进程的代码段预先被载入到了内存中，且程序入口为 `e_entry`，当我们运行进程时，CPU 将自动从 PC 所指的位置开始执行二进制码。

Thinking 3.5 思考上面这一段话，并根据自己在 **Lab2** 中的理解，回答：

- 你认为这里的 `env_tf.cp0_epc` 存储的是物理地址还是虚拟地址？

思考完这一点后，下面我们可以真正创建进程了。

3.2.6 创建进程

这里需要指出，“创建进程”是指在操作系统内核初始化时直接创建进程，而不是在通过 `fork` 等系统调用来创建进程。在 **Lab4** 中将介绍 `fork` 这一种进程创建的方式。创建进程的过程很简单，就是实现对上述个别函数的封装，分配一个新的 `Env` 结构体，设置进程控制块，并将程序载入到目标进程的地址空间即可完成。

Exercise 3.7 完成 `env_create` 函数。

根据提示，理解并恰当使用前面实现的函数，完成 `kern/env.c` 中 `env_create` 函数的填写，实现创建一个新进程的功能。 ■

当然提到创建进程，我们还需要提到两个封装好的宏命令

```
1  #define ENV_CREATE_PRIORITY(x, y) \
2  { \
3      extern u_char binary_###x##_start[];\
4      extern u_int binary_###x##_size; \
5      env_create(binary_###x##_start, \
6                  (u_int)binary_###x##_size, y); \
7  }
```

```
1  #define ENV_CREATE(x) \
2  { \
3      extern u_char binary_###x##_start[];\
4      extern u_int binary_###x##_size; \
5      env_create(binary_###x##_start, \
6                  (u_int)binary_###x##_size, 1); \
7  }
```

最后，你需要在 `init/init.c` 中增加下面两句代码，在内核初始化时创建两个进程。

```
1  ENV_CREATE_PRIORITY(user_bare_loop, 1);
2  ENV_CREATE_PRIORITY(user_bare_loop, 2);
```

这里的 `user_bare_loop` 用于变量命名，对应的用户程序位于 `user/bare/loop.S`。经过 `ENV_CREATE` 宏的拼接后，得到内核中的 `binary_user_bare_loop_start` 数组和 `binary_user_bare_loop_size` 变量，我们可以在 `make` 时构建出的 `user/bare/loop.b.c` 文件中找到它们的定义。

在创建进程前，记得调用 `env_init` 初始化进程管理。

3.2.7 进程运行与切换

进程的运行

```
1  /* Overview:
2   *   Switch CPU context to the specified env 'e'.
3   *
4   * Post-Condition:
5   *   Set 'e' as the current running env 'curenv'.
6   *
7   * Hints:
8   *   You may use these functions: 'env_pop_tf'.
9   */
10 void env_run(struct Env *e) {
11     assert(e->env_status == ENV_RUNNABLE);
12     /* Step 1:
13      *   If 'curenv' is NULL, this is the first time through.
14      *   If not, we are switching from a previous env, so save its context into
15      *   'curenv->env_tf' first.
16      */
17     if (curenv) {
18         curenv->env_tf = *((struct Trapframe *)KSTACKTOP - 1);
```

```

19     }
20
21     /* Step 2: Change 'curenv' to 'e'. */
22     curenv = e;
23     curenv->env_runs++; // lab6
24
25     /* Step 3: Change 'cur_pgdir' to 'curenv->env_pgdir', switching to its
26     * address space.
27     */
28
29     /* Step 4: Use 'env_pop_tf' to restore the curenv's saved context (registers)
30     * and return/go to user mode.
31     *
32     * Hint:
33     *   You should use 'curenv->env_asid' here.
34     *   'env_pop_tf' is a 'noreturn' function: it restores PC from 'cp0_epc' thus
35     * not returning to the kernel caller, making 'env_run' a 'noreturn' function
36     * as well.
37     */
38 }

```

`env_run` 是进程运行使用的基本函数，它包括两部分：

- 保存当前进程上下文 (如果当前没有运行的进程就跳过这一步)
- 恢复要启动的进程的上下文，然后运行该进程。

Note 3.2.6 进程上下文就是进程执行时所有寄存器的状态。具体来说就是代码中的 `Trapframe`。

其实我们这里运行一个新进程往往意味着是进程切换，而不是单纯的进程运行。进程切换，顾名思义，就是当前进程停下工作，让出 CPU 来运行另外的进程。那么要理解进程切换，我们就要知道进程切换时系统需要做些什么。实际上进程切换的时候，为了保证下一次进入这个进程的时候我们不会再“从头来过”，而是有记忆地从离开的地方继续往后走，我们要保存一些信息，那么，需要保存什么信息呢？事实上，我们只需要保存进程的上下文信息，包括通用寄存器、HI、LO 和 CP0 中的 Status, EPC, Cause 和 BadVAddr 寄存器。进程控制块除了 `env_tf` 其他的字段在进程切换后还保留在原本的进程控制块中，并不会改变，因此不需要保存。

在 Lab3 中，我们在本实验里的寄存器状态保存的地方是 `KSTACKTOP` 以下的一个 `sizeof(TrapFrame)` 大小的区域中。

```
curenv->env_tf = *((struct Trapframe *)KSTACKTOP - 1)
```

中的 `curenv->env_tf` 就是当前进程的上下文所存放的区域。我们将把 `KSTACKTOP` 之下的 `Trapframe` 拷贝到当前进程的 `env_tf` 中，以达到保存进程上下文的效果。

总结以上说明，我们不难看出 `env_run` 的执行流程：

1. 保存当前进程的上下文信息。
2. 切换 `curenv` 为即将运行的进程。
3. 设置全局变量 `cur_pgdir` 为当前进程页目录地址，在 TLB 重填时将用到该全局变量。
4. 调用 `env_pop_tf` 函数，恢复现场、异常返回。

这里用到的 `env_pop_tf` 是定义在 `kern/env_asm.S` 中的一个汇编函数。这个函数也呼应了我们前文提到的，进程每次被调度运行前一定会执行的 `eret` 汇编指令。

Exercise 3.8 完成 `env_run`。

仔细阅读前文讲解，并根据注释填写 `kern/env.c` 中的 `env_run` 函数。

至此，第一部分工作已经完成。

3.2.8 实验正确结果

执行 `make test lab=3_1 && make run` 编译运行内核，显示如下信息即通过 `env_init` 的单元测试：

```

1  pe0->env_id 2048
2  pe1->env_id 4097
3  pe2->env_id 6146
4  env_init() work well!
5  pe1->env_pgdir 83ffb000
6  env_setup_vm passed!
7  pe2`s sp register 7f3fdff8
8  [00000000] free env 00001802
9  [00000000] free env 00001001
10 [00000000] free env 00000800
11 env_check() succeeded!
```

执行 `make test lab=3_2 && make run` 编译运行内核，显示如下信息即通过 `load_icode` 的单元测试：

```

1  testing load_icode for icode_check
2  segment check: 401030 - 4011b0 (384)
3  segment check: 402000 - 402fbc (4028)
4  segment check: 402fbc - 403f8c (4048)
5  load_icode test for icode_check passed!
```

3.3 中断与异常

此部分将与计算机组成原理课设 P7 有较大联系，不过即使没有到过 P7 也不用担心，CO 和 OS 负责的部分相当于一个大流程的两个阶段，就像做计算机组成原理课时只需要知道硬件完成任务后放心交给软件，现在操作系统也只需要放心地从硬件处拿到东西继续处理。

在这里先对 P7 的知识进行简单的回顾。CPU 不仅仅有我们常见的 32 个通用寄存器，还有功能广泛的协处理器，而中断/异常部分就用到了其中的协处理器 CP0。下面的表格介绍了编号为 12, 13, 14 的三个 CP0 寄存器的具体功能。

寄存器助记符	CP0 寄存器编号	描述
Status	12	状态寄存器，包括中断引脚使能，其他 CPU 模式等位域
Cause	13	记录导致异常的原因
EPC	14	异常结束后程序恢复执行的位置

Note 3.3.1 我们实验里认为中断是异常的一种，并且是仅有的一种异步异常。

Status 寄存器：图3.1(在设置进程控制块部分给出) 是 MIPS 4Kc 中 Status Register 寄存器，15-8 位为中断屏蔽位，每一位代表一个不同的中断活动，其中 15-10 位使能硬件中断源，9-8 位是 Cause 寄存器软件可写的中断位。

Cause 寄存器：图3.3是 MIPS 4Kc 中 **Cause** 寄存器。其中保存着 CPU 中哪一些中断或者异常已经发生。15-8 位保存着哪一些中断发生了，其中 15-10 位来自硬件，9-8 位可以由软件写入，当 **Status** 寄存器中相同位允许中断（为 1）时，**Cause** 寄存器这一位活动就会导致中断。6-2 位（**ExcCode**），记录发生了什么异常。

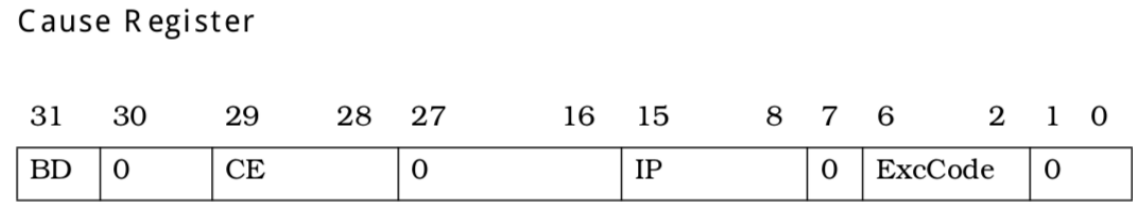


图 3.3: Cause 寄存器

MIPS CPU 处理一个异常时大致要完成四项工作：

- 1 设置 **EPC** 指向从异常返回的地址。
- 2 设置 **EXL** 位，强制 CPU 进入内核态（行使更高级的特权）并禁止中断。
- 3 设置 **Cause** 寄存器，用于记录异常发生的原因。
- 4 CPU 开始从异常入口位置取指，此后一切交给软件处理。

而这句“一切交给软件处理”，就是我们当前任务的开始。
我们可以通过图3.4理解异常的产生与返回过程。

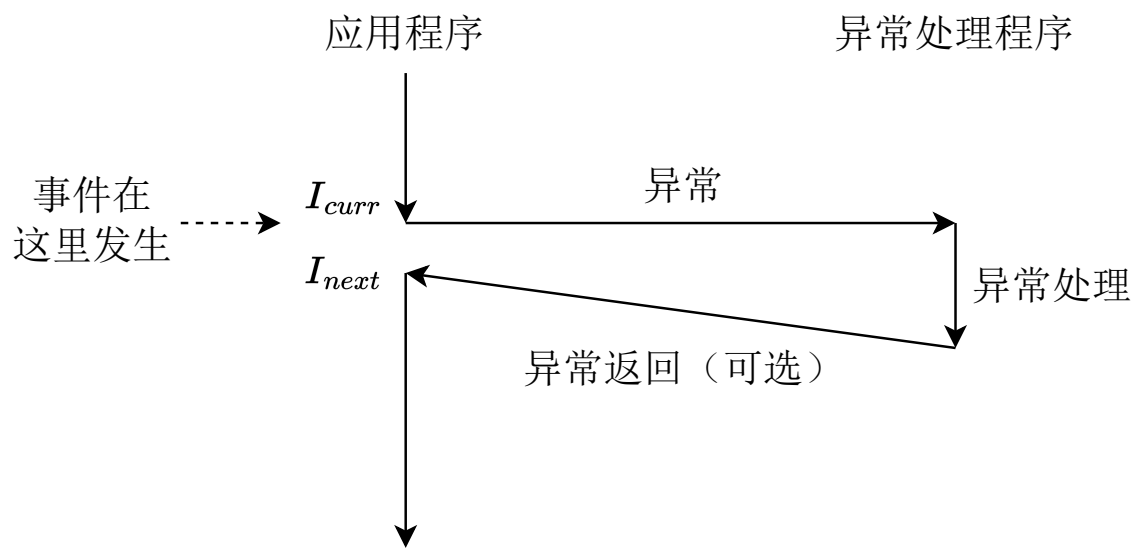


图 3.4: 异常处理图示

3.3.1 异常的分发

当发生异常时，处理器会进入一个用于分发异常的程序，这个程序的作用就是检测发生了哪种异常，并调用相应的异常处理程序。一般来说，异常分发程序会被要求放在固定的某个物理地址上（根据处理器的区别有所不同），以保证处理器能在检测到异常时正确地跳转到那里。这个分发程序可以认为是操作系统的一部分。

下述代码就是异常分发代码，我们先将下面代码填充到 `entry.S` 中，然后分析其功能。

```

1  .section .text.exc_gen_entry
2  exc_gen_entry:
3      SAVE_ALL
4      mfc0    t0, CPO_STATUS
5      and     t0, t0, ~(STATUS_UM | STATUS_EXL | STATUS_IE)
6      mtc0    t0, CPO_STATUS
7      mfc0    t0, CPO_CAUSE
8      andi    t0, 0x7c
9      lw      t0, exception_handlers(t0)
10     jr      t0

```

Exercise 3.9 补充 kern/entry.S。

理解异常分发代码，并将异常分发代码填至 `kern/entry.S` 恰当的部分。

这段异常分发代码的作用流程如下：

1. 使用 `SAVE_ALL` 宏将当前上下文保存到内核的异常栈中。
2. 清除 Status 寄存器中的 UM、EXL、IE 位，以保持处理器处于内核态（UM==0）、关闭中断且允许嵌套异常。
3. 将 Cause 寄存器的内容拷贝到 `t0` 寄存器中。
4. 取得 Cause 寄存器中的 2~6 位，也就是对应的异常码，这是区别不同异常的重要标志。
5. 以得到的异常码作为索引在 `exception_handlers` 数组中找到对应的中断处理函数，后文中会有涉及。
6. 跳转到对应的中断处理函数中，从而响应了异常，并将异常交给了对应的异常处理函数去处理。

这里出现了一个新的宏 `SAVE_ALL`，该宏在后面的 Lab 中也会使用，用于将当前的 CPU 现场（上下文）保存到内核的异常栈中，我们可以在 `include/stackframe.h` 文件中查看这个宏的定义。

在 `SAVE_ALL` 的过程中，我们首先检查待处理的异常是否是在内核态被触发的（异常重入）。我们将 STATUS 寄存器的值读入 `k0` 寄存器，并检查其 UM 位是否为 0，如果是 0 则代表待处理异常是在内核态被触发的，是异常重入的情况。对于异常重入的情况，`sp` 寄存器储存的栈指针已指向内核异常栈，而对于非异常重入的情况，我们需要先将 `sp` 寄存器指向内核异常栈。借助 MIPS 中的延迟槽机制，我们在完成异常重入判断的跳转指令延迟槽中，将 `sp` 寄存器的值复制到 `k0` 寄存器，实现了对 `sp` 寄存器的保存。

`.text.exc_gen_entry` 段和 `.text.tlb_miss_entry` 段需要被链接器放到特定的位置。在 4Kc 中，这两个段分别要求放到地址 `0x80000180` 和 `0x80000000` 处，它们是异常处理程序的入口地址。在我们的系统中，CPU 发生异常（除了用户态地址的 TLB Miss 异常）后，就会自动跳转到地址 `0x80000180` 处；发生用户态地址的 TLB Miss 异常时，会自动跳转到地址 `0x80000000` 处。开始执行。

下面我们在 `kernel.lds` 中增加如下代码，即将 `.text.exc_gen_entry` 放到 `0x80000180` 处，`.text.tlb_miss_entry` 放到 `0x80000000` 处，来为操作系统增加异常分发功能。

```

1      . = 0x80000000;
2      .tlb_miss_entry : {
3          *(.text.tlb_miss_entry)
4      }
5
6      . = 0x80000180;
7      .exc_gen_entry : {
8          *(.text.exc_gen_entry)
9      }

```

Exercise 3.10 补全 kernel.lds。

根据前文讲解将 kernel.lds 代码补全使得异常发生后可以跳到异常分发代码。 ■

3.3.2 异常向量组

异常分发程序通过 `exception_handlers` 数组定位中断处理程序，而 `exception_handlers` 就称作异常向量组。

下面我们来看一下 `kern/traps.c` 中的 `exception_handlers` 数组，来了解异常向量组里存放了什么。

```

1  extern void handle_int(void);
2  extern void handle_tlb(void);
3  extern void handle_sys(void);
4  extern void handle_mod(void);
5  extern void handle_reserved(void);
6
7  void (*exception_handlers[32])(void) = {
8      [0 ... 31] = handle_reserved,
9      [0] = handle_int,
10     [1] = handle_mod,
11     [2 ... 3] = handle_tlb,
12     [8] = handle_sys,
13 };

```

上述代码使用了 GNU C 的扩展语法 `[first ... last] = value` 来对数组某个区间上的元素赋成同一个值。举例来说，

```
int arr[5] = { [0 ... 3] = 1, [2 ... 4] = 2 };
```

这一句等价于下面的代码：

```
int arr[5] = { 1, 1, 2, 2, 2 };
```

通过把相应处理函数的地址填到对应数组项中，我们初始化了如下异常：

- 0 号异常** 的处理函数为 `handle_int`，表示中断，由时钟中断、控制台中断等中断造成
- 1 号异常** 的处理函数为 `handle_mod`，表示存储异常，进行存储操作时该页被标记为只读
- 2 号异常** 的处理函数为 `handle_tlb`，表示 TLB load 异常
- 3 号异常** 的处理函数为 `handle_tlb`，表示 TLB store 异常
- 8 号异常** 的处理函数为 `handle_sys`，表示系统调用，用户进程通过执行 `syscall` 指令陷入内核

Thinking 3.6 试找出 0、1、2、3 号异常处理函数的具体实现位置。8 号异常（系统调用）涉及的 `do_syscall()` 函数将在 Lab4 中实现。 ■

一旦初始化结束，有异常产生，那么其对应的处理函数就会得到执行。而我们在本 Lab 中，主要使用 0 号异常，即中断异常的处理函数（对于 2、3 号异常，在 Lab2 的相关内容中有所介绍）。而我们接下来要做的，就是产生并处理时钟中断，利用时钟中断进行抢占式进程调度。

3.3.3 时钟中断

在前面的介绍中我们已经知道 `Cause` 寄存器中有 8 个独立的中断位。其中 6 位是硬件中断，另外 2 位是软件中断，且不同中断处理起来也会有差异。所以在完成这一部分内容之前，我们首先来介绍一下中断处理的流程。

- 1 通过异常分发，判断出当前异常为中断异常，随后进入相应的中断处理程序。在 MOS 中即对应 `handle_int` 函数。
- 2 在中断处理程序中进一步判断 `Cause` 寄存器中是由几号中断位引发的中断，然后进入不同中断对应的中断服务函数。
- 3 中断处理完成，通过 `ret_from_exception` 函数恢复现场，继续执行。

上面涉及到的函数定义在 `kern/genex.S` 中。

```

                                kern/genex.S
1  #include <asm/asm.h>
2  #include <drivers/dev_rtc.h>
3  #include <stackframe.h>
4
5  .macro BUILD_HANDLER exception handler
6  NESTED(handle_\exception, TF_SIZE, zero)
7      move    a0, sp
8      jal     \handler
9      j       ret_from_exception
10 END(handle_\exception)
11 .endm
12
13 .text
14
15 FEXPORT(ret_from_exception)
16     RESTORE_SOME
17     lw       k0, TF_EPC(sp)
18     lw       sp, TF_REG29(sp) /* Deallocate stack */
19 .set noreorder
20     jr       k0
21     rfe
22 .set reorder
23
24 NESTED(handle_int, TF_SIZE, zero)
25     mfc0     t0, CP0_CAUSE
26     mfc0     t2, CP0_STATUS
27     and      t0, t2
28     andi     t1, t0, STATUS_IM4
29     bnez     t1, timer_irq

```



```

30          // TODO: handle other irqs
31 timer_irq:
32     sw      zero, (KSEG1 | DEV_RTC_ADDRESS | DEV_RTC_INTERRUPT_ACK)
33     li      a0, 0
34     j      schedule
35 END(handle_int)
36
37 BUILD_HANDLER tlb do_tlb_refill
38
39 BUILD_HANDLER reserved do_reserved

```

下面我们来简单介绍一下时钟中断的概念，为什么 MOS 系统需要时钟中断。

时钟中断与 MOS 系统的时间片轮转调度算法是紧密相关的。时间片轮转调度是一种进程调度算法，每个进程被分配一个时间段，称作它的时间片，即该进程允许运行的时间。如果在时间片结束时进程还在运行，则该进程将挂起，切换到另一个进程运行。如果该进程在时间片结束前阻塞或者结束，则立即切换到另一个进程运行。

MOS 是如何知晓一个进程的时间片结束的呢？就是通过硬件定时器产生的时钟中断。在 MOS 中，时间片的长度是用时钟中断衡量的。比如设定某个进程的时间片的长度为 200 倍的 `TIMER_INTERVAL`（时钟中断间隔），那么当 MOS 记录到该进程的执行中发生了 200 个时钟中断时，MOS 就知晓该进程的时间片结束了。

当时钟中断产生时，当前运行的进程被挂起，MOS 需要在调度队列中选取一个合适的进程运行。

4KC 中的 CP0 内置了一个可产生中断的 Timer，MOS 即使用这个内置的 Timer 产生时钟中断。`include/kclock.h` 中的 `RESET_KCLOCK` 宏完成了对 CP0 中 Timer 的配置。

具体来说 CP0 中存在两个用于控制此内置 Timer 的寄存器，即 Count 寄存器与 Compare 寄存器。其中，Count 寄存器会按照某种仅与处理器流水线频率相关的频率不断自增，而 Compare 寄存器维持不变。当 Count 寄存器的值与 Compare 寄存器的值相等且非 0 时，时钟中断会被立即触发。

`RESET_KCLOCK` 宏将 Count 寄存器清零并将 Compare 寄存器配置为我们所期望的计时器周期数，这就对 Timer 完成了配置。在设定个时钟周期后，时钟中断将被触发。

MOS 中，时钟中断的初始化发生在调度执行每一个进程之前。从代码角度，就是在 `env_pop_tf` 中调用了宏 `RESET_KCLOCK`，随后又在宏 `RESTORE_ALL` 中恢复了 Status 寄存器，开启了中断。

一旦时钟中断产生，就会触发 4KC 硬件的异常中断处理流程。系统将 PC 指向 `0x80000180`，跳转到 `.text.exc_gen_entry` 代码段执行。对于时钟引起的中断，通过 `.text.exc_gen_entry` 代码段的分发，最终会调用 `handle_int` 函数进行处理。

`handle_int` 函数根据 Cause 寄存器的值判断是否是 Timer 对应的 7 号中断位引发的时钟中断，如果是，则执行中断服务函数 `timer_irq`，跳转到 `schedule` 中执行。这个函数就是我们将要补充的调度函数。

Exercise 3.11 补充 `RESET_KCLOCK` 宏。

通过上面的描述，补充 `include/kclock.h` 中的 `RESET_KCLOCK` 宏。

Thinking 3.7 阅读 `entry.S`、`genex.S` 和 `env_asm.S` 这几个文件，并尝试说出时钟中断在哪些时候开启，在哪些时候关闭。

3.3.4 进程调度

`handle_int` 函数的最后跳转到了 `schedule` 函数。这个函数在 `kern/sched.c` 中所定义, 它就是我们本次实验最后要写的调度函数。调度的算法很简单, 就是时间片轮转的算法。上文提到, MOS 中的时间片的长度是用时钟中断衡量的, 即时间片长度被量化为 $N \times \text{TIMER_INTERVAL}$ 。具体的, `env` 中的优先级即为这里的 `N`, 规定了该进程的时间片长度。不过寻找就绪状态进程不是简单遍历所有进程, 而是用一个**调度链表**存储所有就绪(可运行)的进程, 即一个进程在调度链表中当且仅当这个进程是就绪(`ENV_RUNNABLE`)状态的。当内核创建新进程时, 将其插入调度链表的头部; 在其不再就绪(被阻塞)或退出时, 将其从调度链表中移除。

调度函数 `schedule` 被调用时, 当前正在运行的进程被存储在全局变量 `curenv` 中(在第一个进程被调度前为 `NULL`), 其剩余的时间片长度被存储在静态变量 `count` 中。我们考虑是否需要进行**进程切换**, 包括以下几种情况:

- 尚未调度过任何进程 (`curenv` 为空指针);
- 当前进程已经用完了时间片;
- 当前进程不再就绪(如被阻塞或退出);
- `yield` 参数指定必须发生切换。

无需进行切换时, 我们只需要将剩余时间片长度 `count` 减去 1, 然后调用 `env_run` 函数, 继续运行当前进程 `curenv`。在发生切换的情况下, 我们还需要判断当前进程是否仍然就绪, 如果是则将其移动到调度链表的尾部。之后, 我们选中调度链表首部的进程来调度运行, 将剩余时间片长度设置为其优先级。

Note 3.3.2 另一种合理的实现是不判断当前进程的运行状态, 直接选中调度队列首部的进程来调度, 同时立即将其移动到调度队列尾部。

此外, 这里也给出如下提示:

1. 静态变量 `count` 中应存储当前进程剩余的执行次数;
2. 调度队列中**存在且只存在**所有就绪(状态为 `ENV_RUNNABLE`)的进程, 其中也需要包括正在运行的进程;
3. 调度函数 `schedule` 以及其中逐级调用的 `env_run`、`env_pop_tf` 和 `ret_from_exception` 函数都是**不返回**(`noreturn`)的函数, 被调用后会从内核跳转到被调度进程的用户程序中执行。在 MIPS 中通常使用 `j` 指令而非 `jal` 调用不返回的函数, 因为它们不会再返回到其调用者。

Exercise 3.12 完成 `schedule` 函数。

根据注释, 填写 `kern/sched.c` 中的 `schedule` 函数实现切换进程的功能, 使得进程能够被正确调度。 ■

Thinking 3.8 阅读相关代码, 思考操作系统是怎么根据时钟中断切换进程的。 ■

至此, 我们的 Lab3 就算是圆满完成了。

3.4 Lab3 在 MOS 中的概况

下图展示了 Lab3 中填写的内容在 MOS 系统中的作用，带有斜体字注释的部分是 Lab3 为系统提供的功能。图中的步骤（或函数调用）表达方式同 Lab2。

在图中，左侧主干是内核初始化的流程（包括创建进程的步骤），右侧是用户进程的运行流程（用户进程中未展开的详细流程用点线表示），中间是异常向量组中与 Lab3 相关的异常处理流程，其中不同的折线表示不同异常的发生和处理流程。

内核初始化完毕后陷入死循环，等待第一次时钟中断来临，通过异常处理来调度已经创建好的用户进程运行，如图中由左侧主干引出的 Timer Interrupt 虚线所示，虚线中省略的过程与右侧 Timer Interrupt 处的实线完全等价。在用户进程运行过程中，也会遇到中断和异常，如图中右侧主干中引出的不同实线所示，包括两种 TLB 中断 TLB Load Miss 和 TLB Store Miss，以及时钟中断 Timer Interrupt，在时钟中断发生后操作系统实现了进程切换。

图中根据箭头方向连接的实线是 PC 的跳转过程。

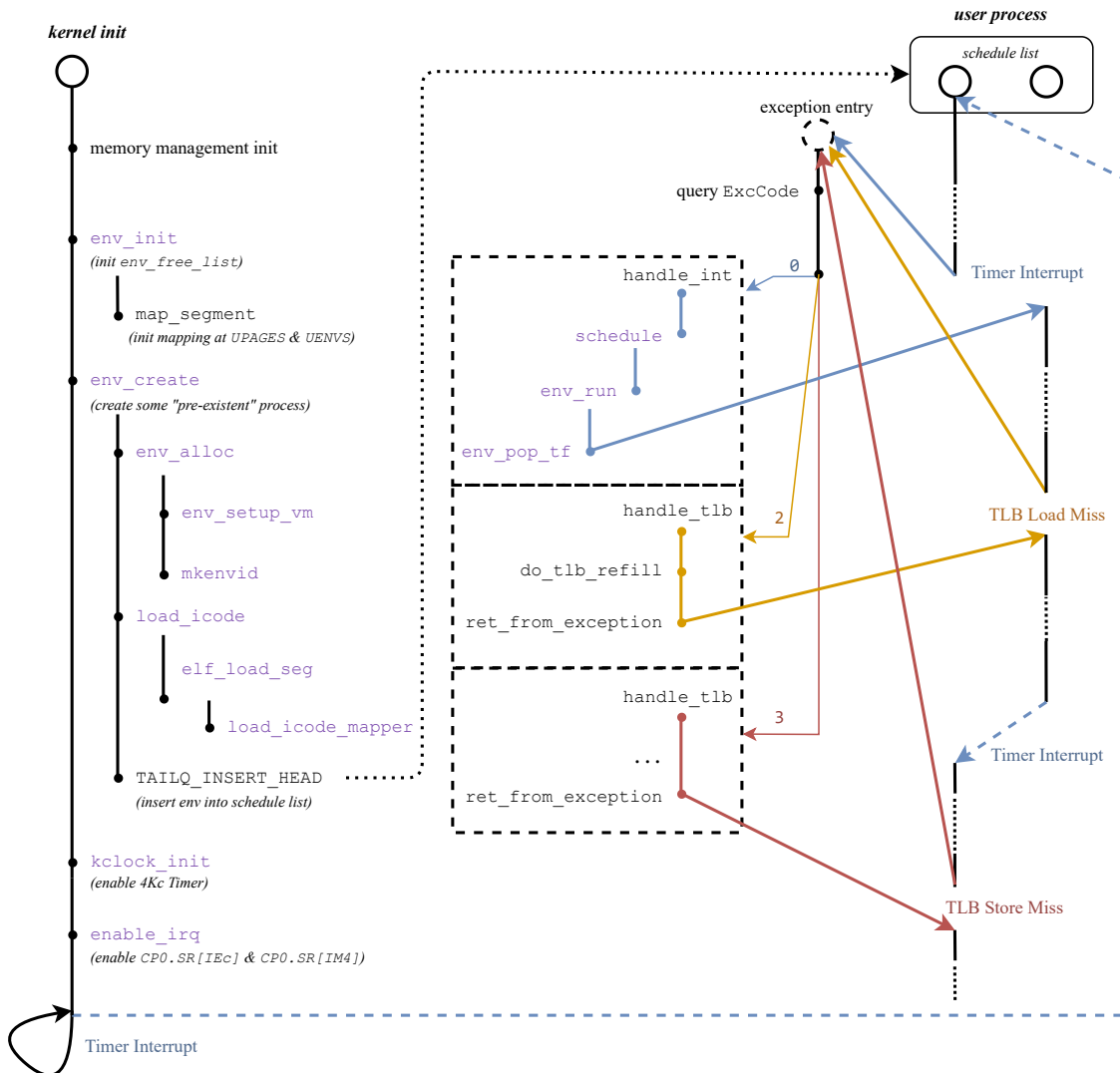


图 3.5: Lab3 in MOS

3.5 实验正确结果

执行 `CFLAGS=-DMOS_SCHED_MAX_TICKS=100 make` 或 `make test lab=3_3` 构建内核，再执行 `make run`，显示如下信息即通过 Lab3 的测试：

```

1      0: 00001001
2      1: 00001001
3      2: 00000800
4      3: 00001001
5      ...
6     96: 00001001
7     97: 00001001
8     98: 00000800
9     99: 00001001
10    100: 00001001
11    101: ticks exceeded the limit 100

```

即观察到进程 1001 与进程 800 交替运行，且前者的数量是后者的两倍。

3.6 代码导读

为了帮助同学们理清 Lab3 代码逻辑和执行流程，我们在这里给出代码导读部分。

Lab3 在 Lab2 基础上增加了两个代码段，叫做 `.tlb_miss_entry` 和 `.exc_gen_entry`，其代码定义在 `kern/entry.S`，实现了分发处理异常的过程，在**异常的分发**小节有所详述，这里不再赘述。

下面将对 Lab3 中的代码进行详细的分析：

1. `exception_handlers` 这个数组在前面的中断处理程序中有所涉及，可参看前面 `.text.exc_gen_entry` 代码部分。主要初始化 0 号异常的处理函数为 `handle_int`，1 号异常处理函数为 `handle_mod`，2 号异常处理函数为 `handle_tlb`，3 号异常处理函数为 `handle_tlb`。初始化结束后，若有异常产生，那么其对应的处理函数就会得到执行。

2. `env_create`

(1) 分配进程控制块

`env_alloc` 函数从空闲链表中分配一个空闲控制块，并进行相应的初始化工作。初始化过程中调用的 `env_setup_vm` 函数，主要做了为进程创建并初始化了页表，也就是为进程创建并初始化了虚拟地址空间。

(2) 调用 `load_icode` 将程序加载到新创建的地址空间中。

3. `handle_int`

根据 `Cause` 寄存器判断是否为 7 号中断（即时钟中断），并根据 `Status` 寄存器判断 7 号中断是否开启。如果上述两判断都为是，则调用 `schedule` 函数。`schedule` 函数进行进程调度，并调用 `env_run` 来运行进程。

4. `env_run`

(1) 将正在执行的进程（如果有）的现场保存到对应的进程控制块中。

(2) 选择一个可以运行的进程，恢复该进程上次被挂起时候的现场。这个过程主要通过 `env_pop_tf` 来完成，而 `env_pop_tf` 调用了 `ret_from_exception` 来完成从异常返回的过程。`env_pop_tf` 函数的部分内容如下：

```

1      mtc0 a1,CP0_ENTRYHI      # 设置 EntryHi 寄存器的 ASID 域
2      move sp, a0              # 设置 sp 寄存器, 配合 RESTORE_ALL 宏
3      RESET_KCLOCK            # 重设时钟中断
4      # j ret_from_exception 为方便观察, 这里展开 ret_from_exception 的代码
5      RESTORE_ALL              # 恢复进程上下文
6      eret

```

我们在 `env.c` 中将进程控制快的 `env_tf.cp0_status` 初始化为 `STATUS_IM7 | STATUS_IE | STATUS_EXL | STATUS_UM`。这样设置是为了让 MOS 可以正常响应时钟中断, 从而可以在时钟中断发生时, 由异常分发程序调用 `handle_int` 函数, 经过 `timer_irq`、`schedule`、`env_run` 来完成进程调度

5. TLB Miss 异常何时产生、如何被处理?

从上面的分析看, 操作系统在时钟中断的驱动下, 通过时间片轮转算法实现进程的并发执行。不过如果不处理 TLB Miss 异常, 系统也不能正常运行。

因为硬件在取数据或者取指令的时候, 会先发出一个所需数据所在的虚拟地址给 MMU, 由 MMU 将这个虚拟地址转换为对应的物理地址, 再以转换后的物理地址执行访存操作。

而 4Kc 的 MMU, 对于 KUSEG 段的地址翻译, 仅由 TLB 完成。正常情况下, TLB 命中虚拟地址, 访存过程完全由硬件完成, 操作系统并不参与其中。但是当 TLB 在转换的时候发现 TLB 中还没有对应于该虚拟地址的映射项目, 那么此时就会产生一个 TLB Miss 异常。硬件会打断无法继续进行的访存操作, 陷入内核态并跳转执行操作系统中对应的异常处理程序 (`tlb_miss_entry`), 由操作系统查找页表, 填补缺失的 TLB 项。之后再从异常返回, 重新执行刚刚被打断的访存。

TLB Miss 对应的异常处理函数是 `kern/genex.S` 中的 `handle_tlb`, 通过宏 `BUILD_HANDLER` 包装了 `do_tlb_refill` 函数来完成 TLB 重填。在 Lab2 中我们已经学习了 [TLB 的基本结构](#), 简单来说就是对于不同进程的同一个虚拟地址, 结合 ASID 和虚拟地址可以定位到不同的物理地址。下面介绍 TLB Miss 异常发生后硬件 (4Kc CPU) 和软件 (MOS 系统) 分别做的工作:

硬件: 在发生 TLB Miss 异常的时候, 4Kc CPU 会把引发 TLB Miss 的虚拟地址填入到 `BadVAddr` 寄存器中、虚页号填入到 `EntryHi` 寄存器的 `VPN` 域中, 将 `Cause` 寄存器中的 `ExcCode` 域填写为 `TLBL` (读请求 TLB Miss) 或 `TLBS` (写请求 TLB Miss)。

软件: 从 `BadVAddr` 寄存器中获取引发 TLB Miss 的虚拟地址, 接着在 `cur_pgdir` 中查找该虚拟地址对应的物理地址与权限位, 然后将物理页面号和权限位填入到 `EntryLo` 寄存器的 `PFN` 域和权限位中, 再使用 `tlbwr` (TLB Write entry selected by Random) 将 `EntryHi` 和 `EntryLo` 寄存器中的 `VPN`、`PFN`、`ASID`、权限位等随机地写入到 TLB 中, 最后调用 `ret_from_exception` 从异常返回。

3.7 任务列表

- Exercise-完成 `env_init` 函数
- Exercise-实现 `map_segment` 函数
- Exercise-完成 `env_setup_vm` 函数
- Exercise-完成 `env_alloc` 函数

- Exercise-完成 `load_icode_mapper` 函数
- Exercise-完成 `load_icode` 函数
- Exercise-完成 `env_create`
- Exercise-完成 `env_run` 函数
- Exercise-完成 `entry.S`
- Exercise-补全 `kernel.lds`
- Exercise-完成 `RESET_KCLOCK` 宏
- Exercise-完成 `schedule` 函数

3.8 实验思考

- 思考-对页表拷贝目的和过程的理解
- 思考-对物理地址和虚拟地址的理解
- 思考-data 的作用
- 思考-`elf_load_seg` 的不同情况
- 思考-EPC 的含义
- 思考-异常处理函数的实现位置
- 思考-时钟的设置
- 思考-进程的调度

4.1 实验目的

1. 掌握系统调用的概念及流程
2. 实现进程间通信机制
3. 实现 `fork` 函数
4. 掌握页写入异常的处理流程

在用户态下，用户进程不能访问系统的内核空间，也就是说它一般不能存取内核使用的内存数据，也不能调用内核函数，这一点是由体系结构保证的。然而，用户进程在特定的场景下往往需要执行一些只能由内核完成的操作，如操作硬件、动态分配内存，以及与其他进程进行通信等。允许在内核态执行用户程序提供的代码显然是不安全的，因此操作系统设计了一系列内核空间中的函数，当用户进程需要进行这些操作时，会引发特定的异常以陷入内核态，由内核调用对应的函数，从而安全地为用户进程提供受限的系统级操作，我们把这种机制称为**系统调用**。

在 Lab4 中，我们需要实现上述的系统调用机制，并在此基础上实现进程间通信（IPC）机制和一个重要的进程创建机制 `fork`。在 `fork` 部分的实验中，我们会介绍一种被称为写时复制（COW）的特性，以及与其相关的页写入异常处理。

4.2 系统调用 (System Call)

本节中，我们着重讨论系统调用的作用，并完成其实现。

4.2.0 用户态与内核态

Note 4.2.1 In kernel mode, the CPU may perform any operation allowed by its architecture; any instruction may be executed, any I/O operation initiated, any area of memory accessed, and so on. In the other CPU modes, certain restrictions on CPU operations are enforced by the hardware. Typically, certain instructions are not permitted (especially those—including I/O operations—that could alter the global state of the machine), some memory areas cannot be accessed, etc. User-mode capabilities of the CPU are typically a subset of those available in kernel mode, but in some cases, such as hardware emula-

tion of non-native architectures, they may be significantly different from those available in standard kernel mode.

—“CPU modes”, Wikipedia.

Note 4.2.2 A modern computer operating system usually segregates virtual memory into user space and kernel space. Primarily, this separation serves to provide memory protection and hardware protection from malicious or errant software behaviour.

Kernel space is strictly reserved for running a privileged operating system kernel, kernel extensions, and most device drivers. In contrast, user space is the memory area where application software and some drivers execute.

—“User space and kernel space”, Wikipedia.

在之前各个 Lab 的学习中，相信大家对用户态、用户空间、内核态等概念已经不陌生了。随着 MOS 操作系统功能的不断完善，我们也需要扩充实现完整的用户态机制，这就包括操作系统中用户进程与内核进行通信的关键机制——系统调用。

我们首先回顾以下几组概念：

1. 用户态 和 内核态 (也称用户模式 和 内核模式)：它们是 CPU 运行的两种状态。根据 Lab3 的说明，在 MOS 操作系统实验使用的仿真 4Kc CPU 中，该状态由 CP0 SR 寄存器中 UM(User Mode) 位和 EXL(Exception Level) 位的值标志。
2. 用户空间 和 内核空间：它们是虚拟内存 (进程的地址空间) 中的两部分区域。根据 Lab2 的说明，MOS 中的用户空间包括 kuseg，而内核空间主要包括 kseg0 和 kseg1。每个进程的用户空间通常通过页表映射到不同的物理页，而内核空间则直接映射到固定的物理页¹以及外部硬件设备。CPU 在内核态下可以访问任何内存区域，对物理内存等硬件设备有完整的控制权，而在用户态下则只能访问用户空间。
3. (用户) 进程 和 内核：进程是资源分配与调度的基本单位，拥有独立的地址空间，而内核负责管理系统资源和调度进程，使进程能够并发运行。与前两对概念不同，进程和内核并不是对立的存在，可以认为内核是存在于所有进程地址空间中的一段代码。

4.2.1 系统调用实例

puts() 函数是 C 标准库中用于输出一个字符串的函数，我们以其在 Linux 下的执行过程为例，展示其完整的调用过程如下：

Step 1	用户调用 puts 函数
Step 2	在一系列的函数调用后，最终调用了 write 函数
Step 3	write 函数在寄存器中设置了对应的系统调用号以及相应的参数，并执行了 syscall 指令
Step 4	进入内核态，内核中相应的函数或服务被执行
Step 5	回到用户态的 write 函数中，将结果从相关的寄存器中取回，并返回
Step 6	再次经过一系列的返回过程后，回到了 puts 函数中
Step 7	puts 函数返回

¹这里忽略了 kseg2 的情况。

在 MIPS 中, `syscall` 是一条用于执行系统调用的自陷指令, 它使得进程陷入到内核的异常处理程序中, 由内核根据系统调用时的上下文执行相应的内核函数, 完成相应的功能, 并最终返回到 `syscall` 的下一条指令。从以上步骤, 我们可以看到:

1. 存在一些只能由内核来完成的操作 (如读写设备、创建进程、IO 等)。
2. C 标准库中一些函数的实现须依赖于操作系统 (如我们所探究的 `puts` 函数)。
3. 通过执行 `syscall` 指令, 用户进程可以陷入到内核态, 请求内核提供的服务。
4. 通过系统调用陷入到内核态时, 需要在用户态与内核态之间进行数据传递与保护。

综合以上内容, 我们可以知道, 内核将自己所能提供的服务以系统调用的方式提供给用户空间, 以供用户程序完成一些特殊的系统级操作。由于用户程序只能将服务相关的参数交予操作系统由操作系统执行, 这样就保证了所有的特殊操作受操作系统掌控, 而这些操作实际执行时也将由内核进行重重检查, 因此系统调用保证了系统的安全性。

进一步, 由于直接使用这些系统调用较为麻烦, 于是产生了一系列用户空间的 API 定义, 如 POSIX 和 C 标准库等, 它们在系统调用的基础上, 实现更多更高级的常用功能, 使得用户在编写程序时不用再处理这些繁琐而复杂的底层操作, 而是直接通过调用高层次的 API 就能实现各种功能。通过这样巧妙的层次划分, 使得程序更为灵活, 也具有了更好的可移植性。对于用户程序来说, 只要自己所依赖的 API 不变, 无论底层的系统调用如何变化, 都不会对自己造成影响, 更易于在不同的系统间移植。整个结构如表 4.1 所示。

表 4.1: API、系统调用层次结构

用户程序 User Program	
应用程序编程接口 API	POSIX, C Standard Library, etc.
系统调用	<code>read, write, etc.</code>

4.2.2 系统调用机制的实现

发现了系统调用的本质之后, 我们就要着手在我们的 MOS 操作系统中实现一套系统调用机制了。为了使得后面的实现思路更清晰, 这里我们先回顾一下 Lab3 里面中断异常处理的行为:

1. 处理器跳转到异常分发代码处
2. 进入异常分发程序, 根据 `cause` 寄存器值判断异常类型并跳转到对应的处理程序
3. 处理异常, 并返回

大家一定还记得异常分发向量组中的 8 号异常, 它就是我们的操作系统处理系统调用时的异常。我们观察已有代码并跟随用户态 `user/lib/debugf.c` 中的 `debugf` 函数来学习其具体流程:

1. `debugf` 函数内部的逻辑可分为两部分, 一部分负责将参数解析为字符串, 一部分负责将字符串输出 (`debug_output` 函数)
2. `debug_output` 函数调用了用户空间的 `syscall_*` 函数
3. `syscall_*` 函数调用了 `msyscall` 函数, 系统由此陷入内核态
4. 内核态中将异常分发到 `handle_sys` 函数, 将系统调用所需要的信息 (在此处是需要输出的字符串 `s`) 传递入内核

5. 内核取得信息，执行对应的内核空间的系统调用函数 (`sys_*`)
6. 系统调用完成，并返回用户态，同时将返回值“传递”回用户态
7. 从系统调用函数返回，回到用户程序 `debugf` 调用处

c

按照如上流程阅读代码的过程中，相信你已经发现了系统调用的流程（如图4.1所示）。

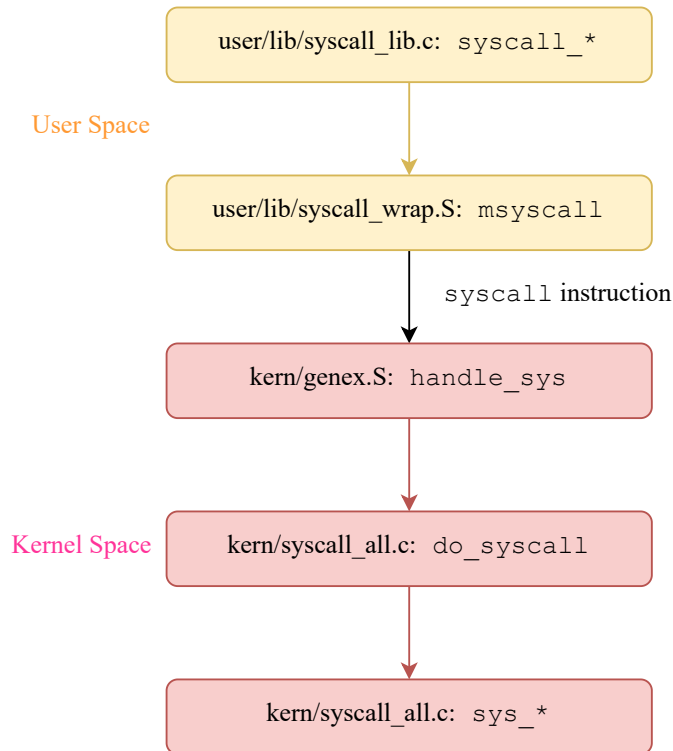


图 4.1: syscall 过程流程图

在用户空间的程序中，我们定义了许多函数，以 `debugf` 函数为例，这一函数实际上并不是最接近内核的函数，它最后会调用一个名为 `syscall_print_cons` 的函数，这个函数在 `user/lib/syscall_lib.c` 中。

下面我们来引入**系统调用号**。

系统调用号的概念：

1. 在我们的 MOS 操作系统实验中，这些 `syscall_*` 的函数与内核中的系统调用函数 (`sys_*` 的函数) 是一一对应的。
 - (a) `syscall_*` 的函数是我们在用户空间中最接近的内核的函数，在用户态调用。
 - (b) `sys_*` 的函数是内核中系统调用的具体实现部分，在内核态执行。
2. 在所有 `syscall_*` 的函数的实现中，都调用了 `msyscall` 函数。
3. `msyscall` 函数的第一个参数是一个与调用名相似的宏。例：`SYS_print_cons`。
4. 这个参数就是**系统调用号**，定义于 `include/syscall.h`。

系统调用号的作用：

1. 类似于不同异常类型对应不同异常号，系统调用号是内核区分不同系统调用的唯一依据。

`msyscall` 函数的参数:

`msyscall` 一共有 6 个参数, 除系统调用号之外 `msyscall` 函数还有 5 个参数, 这些参数是系统调用时需要传递给内核的参数, 因为最多参数的系统调用所需要的参数数量(`syscall_mem_map` 函数需要 5 个参数)。

进一步的问题是, 这些参数究竟是如何从用户态传递入内核态的呢? 这里就需要用 MIPS 的调用规范来说明了。这里引入**栈帧**。

栈帧的概念:

1. 在 MIPS 的调用规范中, 进入函数体时会通过对栈指针做减法 (压栈) 的方式为该函数自身的**局部变量**、**返回地址**、**调用函数的参数**分配存储空间。
2. 而在函数调用结束之后会对栈指针做加法 (弹栈) 来释放这部分空间。
3. 这部分空间称为**栈帧** (stack frame)。

栈帧的使用方法:

1. 调用方: 在自身栈帧的底部预留被调用函数的参数存储空间。
2. 被调用方: 从调用方的栈帧中读取参数。

根据 MIPS 寄存器使用规范, 让我们来学习一下栈帧的实现细节:

1. 寄存器 `$a0-$a3` 用于存放函数调用的前四个参数。
2. 其余的参数存放在栈中。
 - (a) 但是前四个参数还需要在栈上预留空间, 就像它们需要从栈上传递一样。

一个在 MOS 中的例子:

1. `msyscall` 函数一共有 6 个参数。
 - (a) 前 4 个参数会被 `syscall_*` 的函数分别存入 `$a0-$a3` 寄存器 (寄存器传参的部分) 同时栈帧底部保留 16 字节的空间 (不要求存入参数的值)。
 - (b) 后 2 个参数只会被存入在预留空间之上的 8 字节空间内 (没有寄存器传参)。
2. 总共 24 字节的空间用于参数传递。
3. C 代码中的这些调用过程会由编译器自动编译为汇编代码。
4. 而我们在内核中则需要显式地从保存的用户上下文中获取函数的参数值。

详情请见图4.2:

当 `user/lib/syscall_lib.c` 中定义的用户包装函数 `syscall_*` 调用 `msyscall` 时, 根据以上的调用规范, 需要传递给内核的系统调用号以及其他参数已经被合理安置。接下来我们只需要编写用户空间中的 `msyscall` 函数:

1. 这个叶函数没有局部变量。
2. 也就是说这个函数不需要分配栈帧。
3. 只需执行自陷指令 `syscall` 来陷入内核态并在处理结束后正常返回即可。
4. 请注意, `syscall` 指令是不允许在延迟槽中使用的。

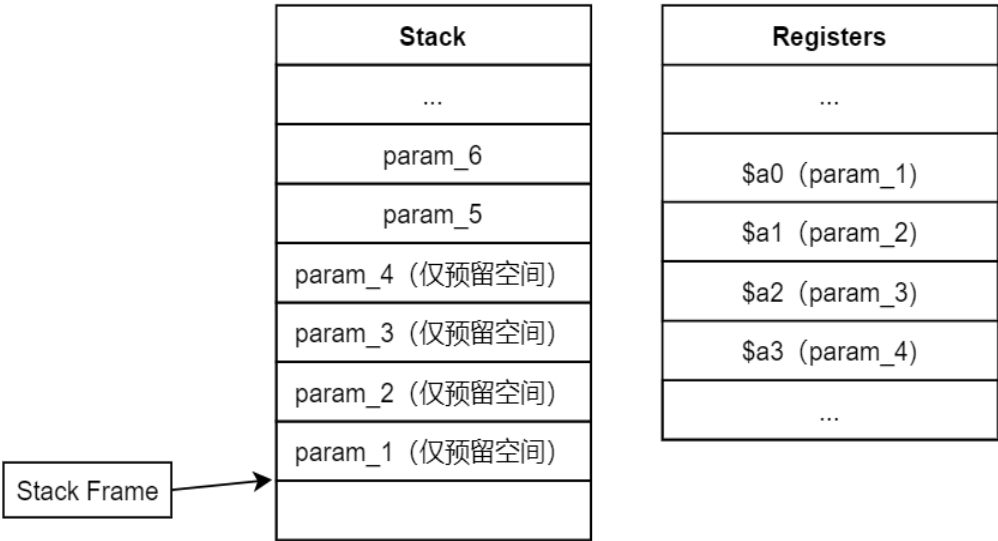


图 4.2: 寄存器传参示意图

Exercise 4.1 填写 `user/lib/syscall_wrap.S` 中的 `msyscall` 函数，使得用户部分的系统调用机制可以正常工作。

在通过 `syscall` 指令陷入内核态后，处理器将 `PC` 寄存器指向一个内核中固定的异常处理入口。在异常向量表中，系统调用这一异常类型的处理入口为 `handle_sys` 函数，它是在 `kern/genex.S` 中定义的对 `do_syscall` 函数的包装，我们需要首先在 `kern/syscall_all.c` 中实现 `do_syscall` 函数。

关于陷入内核态，有一些点需要注意，具体如下：

- 陷入内核态的操作并不是从一个函数跳转到了另一个函数，代码使用的栈指针 `$sp` 是内核空间中的栈指针。
- 系统从用户态切换到内核态后，内核首先需要将原用户进程的运行现场保存到内核空间。
 - (a) 在 `kern/entry.S` 中通过 `SAVE_ALL` 宏完成。
- 随后的栈指针则指向保存的 `Trapframe`，因此我们正是借助这个保存的结构体来获取用户态中传递过来的值。
 - (a) 例如：用户态下 `$a0` 寄存器的值保存在内核栈的 `TF_REG4(sp)` 处。
- 当内核在 `handle_` 开头的包装函数中调用实际进行异常处理的 C 函数时，这个栈指针将作为参数传递给这个 C 函数。
- 因此我们可以在 C 语言中通过这个 `struct Trapframe *` 来获取用户态现场中的参数。

内核的系统调用处理程序

```
1 void do_syscall(struct Trapframe *tf) {
2     int (*func)(u_int, u_int, u_int, u_int, u_int);
3     int sysno = tf->regs[4];
```

```

4      if (sysno < 0 || sysno >= MAX_SYSNO) {
5          tf->regs[2] = -E_NO_SYS;
6          return;
7      }
8
9      /* Step 1: Add the EPC in 'tf' by a word (size of an instruction). */
10     /* Exercise 4.2: Your code here. (1/4) */
11
12     /* Step 2: Use 'sysno' to get 'func' from 'syscall_table'. */
13     /* Exercise 4.2: Your code here. (2/4) */
14
15     /* Step 3: First 3 args are stored at $a1, $a2, $a3. */
16     u_int arg1 = tf->regs[5];
17     u_int arg2 = tf->regs[6];
18     u_int arg3 = tf->regs[7];
19
20     /* Step 4: Last 2 args are stored in stack at [$sp + 16 bytes], [$sp + 20 bytes] */
21     u_int arg4, arg5;
22     /* Exercise 4.2: Your code here. (3/4) */
23
24     /* Step 5: Invoke 'func' with retrieved arguments and store its return value to $v0 in 'tf'. */
25     /* Exercise 4.2: Your code here. (4/4) */
26
27 }
28

```

Thinking 4.1 思考并回答下面的问题：

- 内核在保存现场的时候是如何避免破坏通用寄存器的？
- 系统陷入内核调用后可以直接从当时的 \$a0-\$a3 参数寄存器中得到用户调用 `msyscall` 留下的信息吗？
- 我们是怎么做到让 `sys` 开头的函数“认为”我们提供了和用户调用 `msyscall` 时同样的参数的？
- 内核处理系统调用的过程对 `Trapframe` 做了哪些更改？这种修改对应的用户态的变化是什么？

Exercise 4.2 根据 `kern/syscall_all.c` 中的提示，完成 `do_syscall` 函数，使得内核部分的系统调用机制可以正常工作。

做完这一步，整个系统调用的机制已经可以正常工作，接下来我们要来实现几个具体的系统调用。

4.2.3 基础系统调用函数

在了解系统调用机制后，接下来我们实现几个系统调用。`kern/syscall_all.c` 中定义了一系列系统调用，它们就是 MOS 系统的基础系统调用，后续的 IPC 与 `fork` 机制都以这些系统调用作为支撑。

`sys_mem_alloc` 函数：

1. 这个函数的主要功能是分配内存。

- (a) 通过这个系统调用，用户程序可以给该程序所允许的虚拟内存空间**显式地**分配实际的物理内存。
 - (b) 对于我们程序员的视角而言，是我们编写的程序在内存中申请了一片空间。
 - (c) 对于操作系统内核来说，是一个进程请求将其运行空间中的某段地址与实际物理内存进行映射。
 - i. 此处请同学们回顾进程虚拟页面映射机制、物理内存申请机制。
 - ii. 我们将通过 `page_alloc`, `page_insert` 来实现内存分配和映射页面。
 - iii. 完成时请注意检查虚拟地址的合法性。
 - iv. 向系统调用传入的虚拟地址应当是有效的用户地址，否则不进行内存分配操作，返回错误代码-`E_INVALID`。
 - (d) 从而可以通过该虚拟页面来对物理内存进行存取访问。
2. 接受一个进程的标识符 (`envid`) 作为参数，来确定发出请求的进程。
 3. 通过调用 `envid2env` 函数获得对应的进程控制块。

Exercise 4.3 实现 `kern/env.c` 中的 `envid2env` 函数。

实现通过一个进程的 `id` 获取该进程控制块的功能。提示：可以利用 `include/env.h` 中的宏函数 `ENVX()`，用于获取目标 `Env` 块在 `env` 数组中的下标。

Thinking 4.2 思考 `envid2env` 函数：为什么 `envid2env` 中需要判断 `e->env_id != envid` 的情况？如果没有这步判断会发生什么情况？

Exercise 4.4 实现 `kern/syscall_all.c` 中的 `int sys_mem_alloc(u_int envid, u_int va, u_int perm)` 函数。

`sys_mem_map` 函数：

1. 这个函数的作用是将源进程地址空间中的相应内存映射到目标进程的相应地址空间的相应虚拟内存中去。
 - (a) 此时两者共享一页物理内存。
2. 这个函数有如下操作逻辑。
 - (a) 找到需要操作的两个进程。
 - (b) 获取源进程的虚拟页面对应的实际物理页面。
 - (c) 将该物理页面与目标进程的相应地址完成映射。
 - (d) 完成时请注意检查虚拟地址的合法性。
 - i. 传入的虚拟地址应当是有效的用户地址。
 - ii. 否则不进行页面映射操作，返回错误代码-`E_INVALID`，表示传入非法 (`Invalid`) 用户地址。
3. 这个函数可能会用到如下函数。
 - (a) `page_insert`。
 - (b) `page_lookup`。

Exercise 4.5 实现 kern/syscall_all.c 中的 `int sys_mem_map(u_int srcid, u_int srcva, u_int dstid, u_int dstva, u_int perm)` 函数。 ■

关于内存还有一个函数：`sys_mem_unmap`，这个系统调用的功能是解除某个进程地址空间虚拟内存和物理内存之间的映射关系。可能用到的函数：`page_remove`。

Exercise 4.6 实现 kern/syscall_all.c 中的 `int sys_mem_unmap(u_int envuid, u_int va)` 函数。 ■

除了与内存相关的函数外，另外一个常用的系统调用函数是 `sys_yield`。这个函数的功能是实现用户进程对 CPU 的放弃，从而调度其他的进程。可以利用我们之前已经编写好的函数 `schedule`。

Exercise 4.7 实现 kern/syscall_all.c 中的 `void sys_yield(void)` 函数。 ■

至此，我们能够进一步理解进程与内核间的关系并非对立：在内核处理进程发起的系统调用时，我们并没有切换地址空间（页目录地址），也不需要保存进程上下文（Trapframe）保存到进程控制块中，只是切换到内核态下，执行了一些内核代码。可以说，处理系统调用时的内核仍然是代表当前进程的，这也是系统调用、TLB 缺失等同步异常与时钟中断等异步异常的本质区别，

实现系统调用后，我们已经可以编写并运行用户程序，利用系统调用让用户程序在控制台上输出文本了。

4.3 进程间通信机制 (IPC)

进程间通信机制 (IPC) 是微内核最重要的机制之一。

Note 4.3.1 上世纪末，微内核设计逐渐成为了一个热点。微内核设计主张将传统操作系统中的设备驱动、文件系统等可在用户空间实现的功能，移出内核，作为普通的用户程序来实现。这样，即使它们崩溃，也不会影响到整个系统的稳定。其他应用程序通过进程间通信来请求文件系统等相关服务。因此，在微内核中 IPC 是一个十分重要的机制。

IPC 机制的实现远远没有我们想象得那样神秘，特别是在我们这个被简化了的 MOS 操作系统中。IPC 机制的实现使得我们系统中的进程之间拥有了相互传递消息的能力，为后续实现 `fork`、文件系统服务、管道和 `shell` 均有着极大的帮助。根据之前的讨论，我们能够确定的是：

- IPC 的目的是使两个进程之间可以通信
- IPC 需要通过系统调用来实现
- IPC 还与进程的数据、页面等信息有关

所谓通信，最直观的一种理解就是交换数据。假如我们能够将一个进程有能力将数据传递给另一个进程，那么进程之间自然具有了相互通信的能力。由于这些进程的地址空间之间是相互独立的，要想传递数据，我们就需要想办法把一个地址空间中的东西传给另一个地址空间。

我们知道，所有的进程都共享同一个内核空间（主要为 `kseg0`）。因此，想要在不同空间之间交换数据，我们就可以借助于内核空间来实现。发送方进程可以将数据以系统调用的形式存放在进程控制块中，接收方进程同样以系统调用的方式在进程控制块中找到对应的数据，读取并返回。


```

1  struct Env {
2      // lab 4 IPC
3      u_int env_ipc_value;           // data value sent to us
4      u_int env_ipc_from;           // envid of the sender
5      u_int env_ipc_recving;        // env is blocked receiving
6      u_int env_ipc_dstva;          // va at which to map received page
7      u_int env_ipc_perm;           // perm of page mapping received
8  };

```

在进程控制块中我们看到了我们想要的内容：

env_ipc_value	进程传递的具体数值
env_ipc_from	发送方的进程 ID
env_ipc_recving	1: 等待接受数据中; 0: 不可接受数据
env_ipc_dstva	接收到的页面需要与自身的哪个虚拟页面完成映射
env_ipc_perm	传递的页面的权限位设置

知道了这些，我们就不难实现 IPC 机制了。请结合下方讲解完成练习：

Exercise 4.8 实现 kern/syscall_all.c 中的 `int sys_ipc_recv(u_int dstva)` 函数和 `int sys_ipc_try_send(u_int envid, u_int value, u_int srcva, u_int perm)` 函数。请注意在修改进程控制块的状态后，应同步维护调度队列。

`sys_ipc_recv(u_int dstva)` 函数用于接受消息。在该函数中：

1. 首先要将自身的 `env_ipc_recving` 设置为 1，表明该进程准备接受发送方的消息
2. 之后给 `env_ipc_dstva` 赋值，表明自己要接受到的页面与 `dstva` 完成映射
3. 阻塞当前进程，即把当前进程的状态置为不可运行 (`ENV_NOT_RUNNABLE`)
4. 最后放弃 CPU（调用相关函数重新进行调度），等待发送方将数据发送过来

`sys_ipc_try_send(u_int envid, u_int value, u_int srcva, u_int perm)` 函数用于发送消息：

1. 根据 `envid` 找到相应进程，如果指定进程为可接收状态（考虑 `env_ipc_recving`），则发送成功
2. 否则，函数返回 `-E_IPC_NOT_RECV`，表示目标进程未处于接受状态
3. 清除接收进程的接收状态，将相应数据填入进程控制块，传递物理页面的映射关系
4. 修改进程控制块中的进程状态，使接受数据的进程可继续运行 (`ENV_RUNNABLE`)

IPC 的大致流程总结如图 4.3 所示：

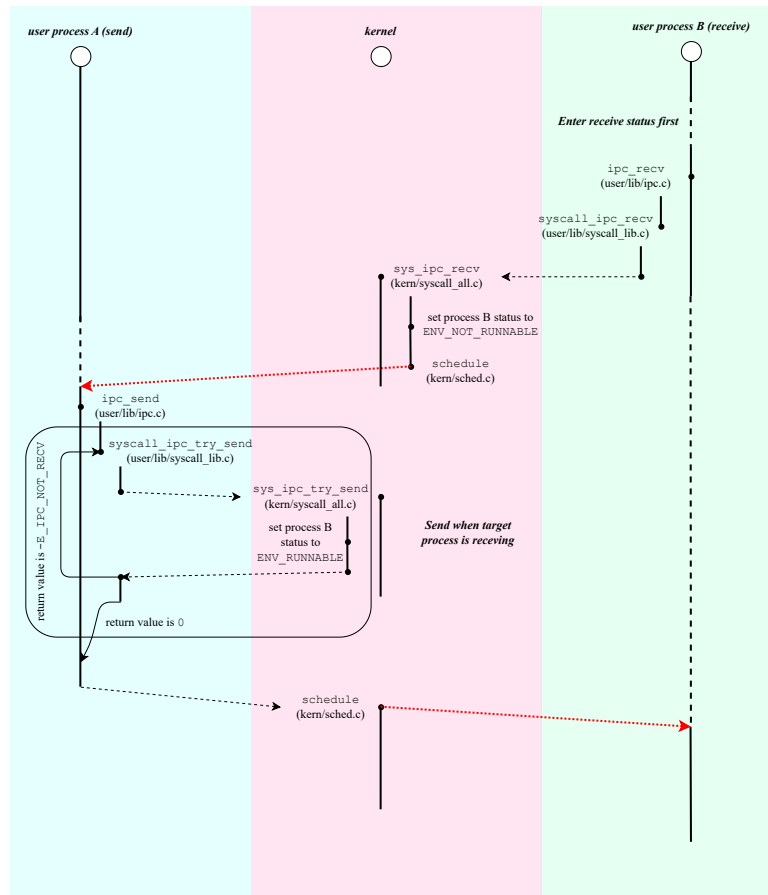


图 4.3: IPC 流程图

值得一提的是，由于在我们的用户程序中，会大量使用 `srcva` 为 0 的调用来表示只传 `value` 值，而不需要传递物理页面，换句话说，当 `srcva` 不为 0 时，我们才建立两个进程的页面映射关系。因此在编写相关函数时也需要注意此种情况。

Thinking 4.3 思考下面的问题，并对这个问题谈谈你的理解：请回顾 `kern/env.c` 文件中 `mkenvid()` 函数的实现，该函数不会返回 0，请结合系统调用和 IPC 部分的实现与 `envid2env()` 函数的行为进行解释。

4.4 Fork

在 Lab3 我们曾提到过，内核通过 `env_create` 函数创建一个进程。但如果要让一个进程创建一个进程，我们就需要基于系统调用，引入新的 `fork` 机制了。

4.4.1 初窥 fork

fork 函数简介：

1. fork 这个函数名有如下内涵。
 - (a) 直接的翻译是叉子的意思，而在操作系统中是表示分叉的意思。
 - (b) 就好像一条河流动着，遇到一个分叉口，分成两条河一样。
 - (c) **fork** 就是那个分叉口。
2. 一个进程在调用 **fork()** 函数后，将从此分叉成为两个进程运行。
 - (a) 其中新产生的进程称为原进程的**子进程**。
 - i. 子进程开始运行时的大部分上下文状态与原进程相同
 - A. 包括程序和 **fork** 运行时的现场（包括通用寄存器和程序计数器 PC 等）。
 - ii. 在子进程中，**fork()** 调用的返回值为 0。
 - (b) 其中旧的进程称为子进程的**父进程**。
 - i. 在父进程中，**fork()** 调用的返回值为子进程的 **env_id**。
 - A. **env_id** 一定大于 0。
 - B. **fork** 失败的情况下，子进程不会被创建，且父进程将得到小于 0 的返回值。

fork 在父子进程中产生不同返回值这一特性，让我们能够在代码中调用 **fork** 后判断当前在父进程还是子进程中，以执行不同的后续逻辑，也使父进程能够与子进程进行通信。

你可能会想，**fork** 执行完为什么不直接生成一个空白的进程块，生成一个几乎和父进程一模一样的子进程有什么用呢？事实上，**fork** 是 Linux 操作系统中创建新进程最主要的方式。相比独立开始运行的两个进程，父子进程间的通信要方便的多。因为子进程中仍能读取原属于父进程的部分数据，父进程也可以根据 **fork** 返回的子进程 **id**，通过调用其他系统接口控制其行为。

在 Linux 中，与 **fork** 经常一起使用的，是名为 **exec** 的一系列系统调用。它会使进程抛弃现有的程序和运行现场，执行一个新的程序。若在进程中调用 **exec**，进程的地址空间（以及在内存中持有的所有数据）都将被重置，新程序的二进制镜像将被加载到其代码段，从而让一个从头运行的全新进程取而代之。**fork** 的一种常见应用就被称作 **fork-exec**，指在 **fork** 出的子进程中调用 **exec**，从而在创建出的新进程中运行另一个程序。

为了让读者对 **fork** 的认识不只是停留在理论层面，我们下面来做一个小实验，复制到您的 Linux 环境下运行一下吧。

fork_test.c

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  int main() {
5      int var = 1;
6      long pid;
7      printf("Before fork, var = %d.\n", var);
8      pid = fork();
9      printf("After fork, var = %d.\n", var);
10     if (pid == 0) {
11         var = 2;
12         sleep(3);
```

```

13         printf("child got %ld, var = %d", pid, var);
14     } else {
15         sleep(2);
16         printf("parent got %ld, var = %d", pid, var);
17     }
18     printf(", pid: %ld\n", (long) getpid());
19     return 0;
20 }

```

使用 `gcc fork_test.c && ./a.out` 运行一下，你得到的输出应该如下所示（pid 可能不同）：

```

1 Before fork, var = 1.
2 After fork, var = 1.
3 After fork, var = 1.
4 parent got 16903, var = 1, pid: 16902
5 child got 0, var = 2, pid: 16903

```

我们从这段简短的代码里可以获取到很多的信息，比如以下几点：

- `fork` 之前只有父进程存在。
- `fork` 之后，父子进程同时开始执行 `fork` 之后的代码段。
- `fork` 在不同的进程中返回值不一样，在子进程中返回值为 0，在父进程中返回值不为 0，而为子进程的 pid（Linux 中进程专属的 id，类似于 MOS 中的 `envid`）。
- 父进程和子进程虽然很多信息相同，但他们的进程控制块是不同的。

从上面的小实验我们也能看出来，子进程实际上就是按父进程的代码段等内存数据，以及进程上下文等状态作为模板而雕琢出来的。但即使如此，父子进程也还是有很多不同的地方。

Thinking 4.4 关于 `fork` 函数的两个返回值，下面说法正确的是：

- A、`fork` 在父进程中被调用两次，产生两个返回值
- B、`fork` 在两个进程中分别被调用一次，产生两个不同的返回值
- C、`fork` 只在父进程中被调用了一次，在两个进程中各产生一个返回值
- D、`fork` 只在子进程中被调用了一次，在两个进程中各产生一个返回值

我们简要概括一下整个 `fork` 实现过程中可能需要阅读或实现的文件，包括：

- `kern/syscall_all.c`: `sys_exofork` 函数, `sys_set_env_status` 函数, `sys_set_tlb_mod_entry` 函数是我们这次需要完成的函数。
- `kern/tlbex.c`: `do_tlb_mod` 函数负责完成写时复制处理前的相关设置，也是我们这次需要完成的函数。
- `user/lib/fork.c`: `fork` 函数是我们这次实验的重点函数，我们将分多个步骤来完成这个函数。
- `user/lib/fork.c`: `cow_entry` 函数是写时复制处理的函数，也是内核会从 `do_tlb_mod` 返回到的函数，负责对带有 `PTE_COW` 标志的页面进行处理，是我们这次需要完成的主要函数之一。
- `user/lib/fork.c`: `duppage` 函数是父进程对子进程页面空间进行映射以及相关标志设置的函数，是我们这次需要完成的主要函数之一。

- **user/lib/entry.S**: 用户进程的入口，是我们这次需要了解的函数之一。
- **user/lib/libos.c**: 用户进程入口的 C 语言部分，负责完成执行用户程序 **main** 前后的准备和清理工作，是我们这次需要了解的函数之一。
- **kern/genex.S**: 该文件实现了 MOS 的异常处理流程，虽然不是我们需要实现的重点，但是建议读者认真阅读，理解中断处理的流程。

本实验中 MOS 系统的 **fork** 函数流程大致如图 4.4 所示，其中的大部分函数也是这次本次实验的任务，会在后续详细介绍。

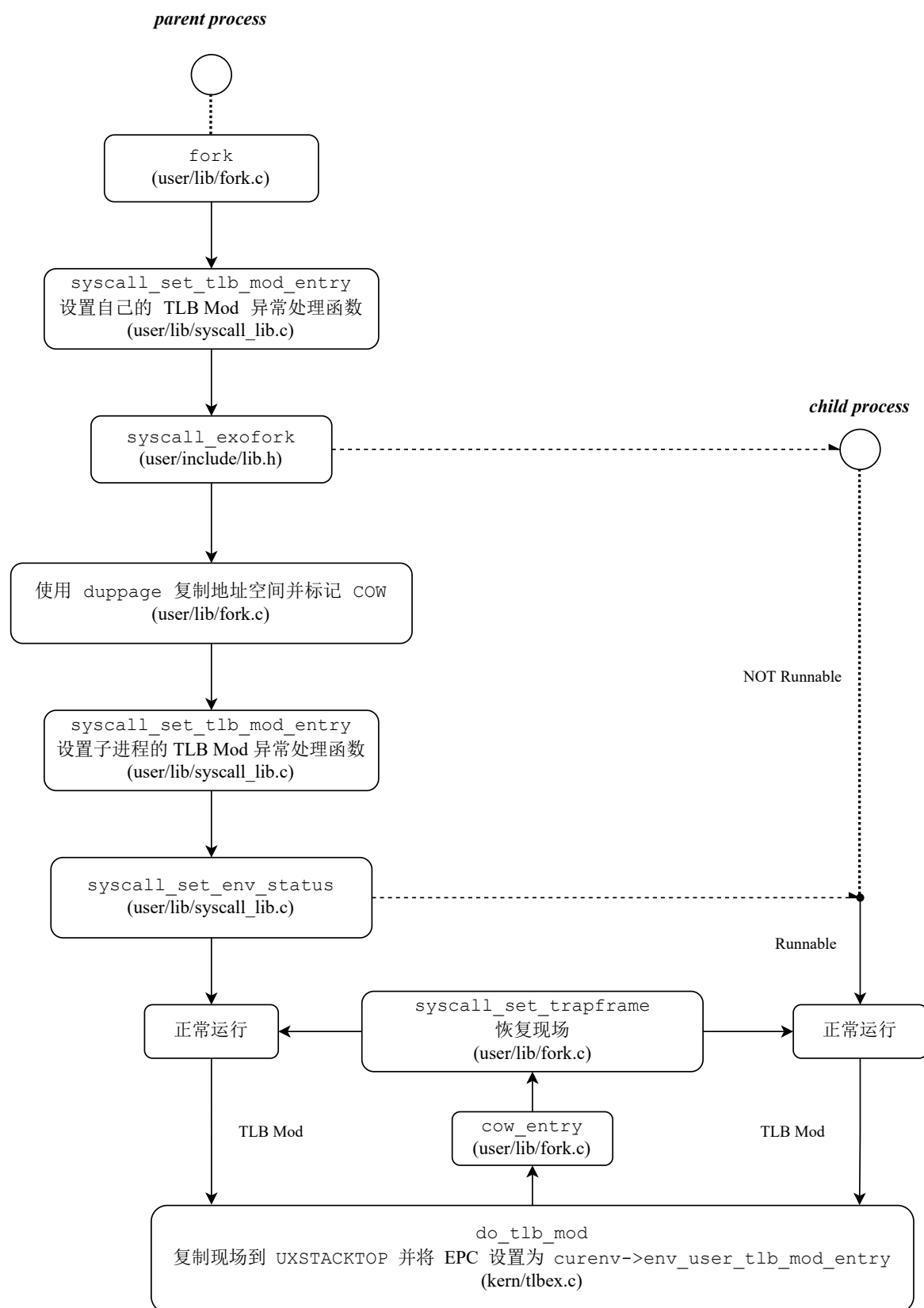


图 4.4: fork 流程图

4.4.2 写时复制机制

前文中，我们知道了在调用 `fork` 后，子进程会继承父进程地址空间中的代码段和数据段等内容。由于在 `fork` 后，父子进程将成为相互独立的两个进程，因此两个进程对于其内存的修改应该是互不影响的。

如果我们在 `fork` 时将父进程地址空间中的内容全部复制到新的物理页，将会消耗大量的物理内存。而这些物理内存中，如代码段部分，父子进程通常不会对其进行写入。对于这样的页面，我们希望能够避免对它们进行复制，从而可以节省物理内存。

为了父子进程能够共用尽可能多的物理内存，我们希望引入一种**写时复制**（Copy-on-write, COW）机制：

1. 在 `fork` 时，只需将地址空间中的所有可写页标记为写时复制页面。
2. 根据标记，在父进程或子进程对写时复制页面进行写入时，能够产生一种**异常**。
3. 操作系统处理异常如下。
 - (a) 为当前进程试图写入的虚拟地址分配新的物理页面。
 - (b) 新的页面复制原页面的内容。
 - (c) 返回用户程序。
4. 处理完成后即可对新分配的物理页面进行写入。
5. 这种机制使得我们可以最大限度的节省物理内存，减少了不必要的复制。

为了实现写时复制机制，我们需要借助硬件异常来实现。在 MIPS 中，当程序尝试写入的虚拟页对应的 TLB 项没有 `PTE_D` 标志时，会触发 TLB Mod 异常，使系统陷入到内核中。由于对应的异常处理函数是由内核设置的，因此我们可以使用它来实现写时复制机制。

我们可以将写时复制页面的 `PTE_D` 标志置为 0。当进程读这个页面时，不会出现问题。但当进程尝试写这个页面时，由于 `PTE_D` 为 0，所以会触发 TLB Mod 异常。此时，我们就可以在异常处理函数中，将虚拟页映射到一个新的物理页，然后将旧的物理页的内容复制到新的物理页中。这样，两个进程的虚拟页就会各自映射到不同的物理页了。之后进程再对这个虚拟页进行修改，就不会有任何问题了。

同时，为了区分真正的“只读”页面和“写时复制”页面，我们需要利用 TLB 项中的软件保留位，引入新的标志位 `PTE_COW`。在 TLB Mod 的异常处理函数中，如果触发该异常的页面的 `PTE_COW` 为 1，我们就需要进行上述的写时复制处理，即分配一页新的物理页，将写时复制页面的内容拷贝到这一物理页，然后映射给尝试写入该页面的进程。

在我们的 MOS 操作系统实验中，进程调用 `fork` 时，需要对其所有的可写入的内存页面，设置页表项标志位 `PTE_COW` 并取消可写位 `PTE_D`，以实现写时复制保护。在这样的保护下，用户程序可以在逻辑上认为 `fork` 时父进程中内存的状态被完整复制到了子进程中，此后父子进程可以独立操作各自的内存。

4.4.3 `fork` 的返回值

在我们的 MOS 操作系统实验中，需要强调的一点是我们实现的 `fork` 是一个用户态函数，`fork` 函数中需要若干个原子的系统调用来完成所期望的功能。其中最核心的一个系统调用就是新进程的创建 `syscall_exofork`。

在 `fork` 的实现中，我们是通过判断 `syscall_exofork` 的返回值来决定 `fork` 的返回值以及后续动作，所以会有类似这样结构的代码：

```

1   envid = syscall_exofork();
2   if (envid == 0) {
3       // 子进程
4       ...
5   } else {
6       // 父进程
7       ...
8   }

```

既然 `fork` 的目的是使得父子进程处于几乎相同的运行状态，我们可以认为在返回用户态时，父子进程应该经历了同样的恢复运行现场的过程，只不过对于父进程是从系统调用中返回时恢复现场，而对于子进程则是在进程被调度时恢复现场。在现场恢复后，父子进程都会从内核返回到 `msyscall` 函数中，而它们的现场中存储的返回值（即 `$v0` 寄存器的值）是不同的。这一返回值随后再被返回到 `syscall_exofork` 和 `fork` 函数，使 `fork` 函数也能区分两者。

为了实现这一特性，你可能需要先实现 `sys_exofork` 的几个任务，在它分配一个新的进程控制块后，还需要用一些当前进程的信息作为模版来填充这个控制块：

运行现场 要复制一份当前进程的运行现场（进程上下文）`Trapframe` 到子进程的进程控制块中。

返回值有关 这个系统调用本身是需要一个返回值的，我们希望系统调用在内核态返回的 `envid` 只传递给父进程，对于子进程，则需要将其现场中的 `v0` 寄存器修改为 0。

进程状态 我们当然不能让子进程在父进程的 `syscall_exofork` 返回后就直接被调度，因为这时候它还没有做好充分的准备，所以我们需要将它的状态设为 `ENV_NOT_RUNNABLE` 且避免它被加入调度队列。

其他信息 观察 `Env` 结构体的结构，思考下还有哪些字段需要进行初始化，这些字段的初始值应该是继承自父进程还是使用新的值，如果这些字段没有初始化会有什么后果（提示：`env_pri`）。

Exercise 4.9 请根据上述步骤以及代码中的注释提示，填写 `kern/syscall_all.c` 中的 `sys_exofork` 函数。 ■

在解决完返回值的问题之后，父与子就能够分别各自从 `syscall_exofork` 中返回。

4.4.4 地址空间的准备

MOS 允许进程访问自身的进程控制块，而在 `user/lib/libos.c` 的实现中，用户程序在运行时入口会将一个用户空间中的指针变量 `struct Env *env` 指向当前进程的控制块。对于 `fork` 后的子进程，它具有了一个与父亲不同的进程控制块，因此在子进程第一次被调度的时候（当然这时还是在 `fork` 函数中）需要对 `env` 指针进行更新，使其仍指向当前进程的控制块。这一更新过程与运行时入口对 `env` 指针的初始化过程相同，具体步骤如下：

1. 通过一个系统调用来取得自己的 `envid`，因为对于子进程而言 `syscall_exofork` 返回的是一个 0 值。
2. 根据获得的 `envid`，计算对应的进程控制块的下标，将对应的进程控制块的指针赋给 `env`。

此外，父进程还需要将地址空间中需要与子进程共享的页面映射给子进程，这需要我们遍历父进程的大部分用户空间页，并使用将要实现的 `duppage` 函数来完成这一过程。`duppage` 时，对于可以写入的页面的页表项，在父进程和子进程都需要加上 `PTE_COW` 标志位，同时取消 `PTE_D` 标志位，以实现写时复制保护。

Thinking 4.5 我们并不应该对所有的用户空间页都使用 `duppage` 进行映射。那么究竟哪些用户空间页应该映射，哪些不应该呢？请结合 `kern/env.c` 中 `env_init` 函数进行的页面映射、`include/mmu.h` 里的内存布局图以及本章的后续描述进行思考。 ■

Thinking 4.6 在遍历地址空间存取页表项时你需要使用到 `vpd` 和 `vpt` 这两个指针，请参考 `user/include/lib.h` 中的相关定义，思考并回答这几个问题：

- `vpt` 和 `vpd` 的作用是什么？怎样使用它们？
- 从实现的角度谈一下为什么进程能够通过这种方式来存取自身的页表？
- 它们是如何体现自映射设计的？
- 进程能够通过这种方式来修改自己的页表项吗？

在 `duppage` 函数中，唯一需要强调的一点是，要对具有不同权限位的页使用不同的方式进行处理。你可能会遇到这几种情况：

只读页面 对于不具有 `PTE_D` 权限位的页面，按照相同权限（只读）映射给子进程即可。

写时复制页面 即具有 `PTE_COW` 权限位的页面。这类页面是之前的 `fork` 时 `duppage` 的结果，且在本次 `fork` 前必然未被写入过。

共享页面 即具有 `PTE_LIBRARY` 权限位的页面。这类页面需要保持共享可写的状态，即在父子进程中映射到相同的物理页，使对其进行修改的结果相互可见。在文件系统部分的实验中，我们会使用到这样的页面。

可写页面 即具有 `PTE_D` 权限位，且不符合以上特殊情况的页面。这类页面需要在父进程和子进程的页表项中都使用 `PTE_COW` 权限位进行保护。

Exercise 4.10 结合代码注释以及上述提示，填写 `user/lib/fork.c` 中的 `duppage` 函数。 ■

Note 4.4.1 在用户态实现的 `fork` 并不是一个原子的过程，所以会出现一段时间（也就是在 `duppage` 之前的时间）我们没有来得及为栈所在的页面设置写时复制的保护机制，在这一段时间内对栈的修改（比如发生了其他的函数调用），会将非叶函数 `syscall_exofork` 函数调用的栈帧中的返回地址覆盖。这一问题对于父进程来说是理所当然的，然而对于子进程来说，这个覆盖导致的后果则是在从 `syscall_exofork` 返回时跳转到一个不可预知的位置造成 `panic`。当然你现在看到的代码已经通过一个优雅的办法来修补这个问题：与其他系统调用函数不同，`syscall_exofork` 是一个内联（inline）的函数，也就是说这个函数并不会被编译为一个函数，而是直接内联展开在 `fork` 函数内。所以 `syscall_exofork` 的栈帧就不存在了，`msyscall` 函数直接返回到了 `fork` 函数内，如此这个问题就解决了。

在完成写时复制的保护机制后，还不能让子进程处于能被调度的状态，因为作为父亲它还有其他的责任——为写时复制特性的页写入异常处理做好准备。

4.4.5 页写入异常

内核在捕获到一个常规的缺页中断（TLB 缺失异常）时会陷入异常，跳转到异常处理函数 `handle_tlb` 中，这一汇编函数的实现在 `kern/genex.S` 中，通过调用 `do_tlb_refill` 函数，在

页表中进行查找，将物理地址填入 TLB 并返回到用户程序中的异常地址，再次执行访存指令。

前文中我们提到了写时复制 (COW) 特性，这种特性也是依赖于异常处理的。当用户程序写入一个在 TLB 中被标记为不可写入 (无 PTE_D) 的页面时，MIPS 会陷入**页写入异常** (TLB Mod)，我们在异常向量组中为其注册了一个处理函数 `handle_mod`，这一函数会跳转到 `kern/tlbex.c` 中的 `do_tlb_mod` 函数中，这个函数正是处理页写入异常的内核函数。对于需要写时复制 (COW) 的页面，我们只需取消其 PTE_D 标记，即可在它们被写入时触发 `do_tlb_mod` 中的处理逻辑。

你可能会发现，`do_tlb_mod` 函数似乎并没有进行页面复制等 COW 的处理操作。事实上，我们的 MOS 操作系统按照微内核的设计理念，尽可能地将功能实现在用户空间中，其中也包括了页写入异常的处理，因此主要的处理过程是在用户态下完成的。

如果需要在用户态下完成页写入异常的处理，是不能直接使用正常情况下的用户栈的（因为发生页写入异常的也可能是正常栈的页面），所以用户进程就需要一个单独的栈来执行处理程序，我们把这个栈称作**异常处理栈**，它的栈顶对应的是内存布局中的 `UXSTACKTOP`。此外，内核还需要知晓进程自身的处理函数所在地址，它的地址存在于进程控制块的 `env_user_tlb_mod_entry` 域中，这个地址也需要事先由父进程通过系统调用设置。

因此，概括一下上述内容，在我们的 MOS 操作系统中，处理页写入异常的大致流程可以概括为：

1. 用户进程触发页写入异常，陷入到内核中的 `handle_mod`，再跳转到 `do_tlb_mod` 函数。
2. `do_tlb_mod` 函数负责将当前现场保存在异常处理栈中，并设置 `a0` 和 `EPC` 寄存器的值，使得从异常恢复后能够以异常处理栈中保存的现场 (`Trapframe`) 为参数，跳转到 `env_user_tlb_mod_entry` 域存储的**用户异常处理函数**的地址。
3. 从异常恢复到用户态，跳转到用户异常处理函数中，由用户程序完成写时复制等自定义处理。

处理 COW 时，我们需要注册的页写入异常用户处理函数是 `fork.c` 中定义的 `cow_entry` 函数。这个函数进行写时复制处理之后，使用系统调用 `syscall_set_trapframe` 恢复事先保存好的现场，其中也包括 `sp` 和 `PC` 寄存器的值，使得用户程序恢复执行。

Exercise 4.11 根据上述提示以及代码注释，完成 `kern/tlbex.c` 中的 `do_tlb_mod` 函数，设置好保存的现场中 `EPC` 寄存器的值。 ■

Thinking 4.7 在 `do_tlb_mod` 函数中，你可能注意到了有一个向异常处理栈复制 `Trapframe` 运行现场的过程，请思考并回答这几个问题：

- 这里实现了一个支持类似于“异常重入”的机制，而在什么时候会出现这种“异常重入”？
- 内核为什么需要将异常的现场 `Trapframe` 复制到用户空间？ ■

让我们回到 `fork` 函数，在调用 `syscall_exofork` 之前，我们需要使用 `syscall_set_tlb_mod_entry` 函数来注册自身的页写入异常处理函数，也就是我们上文提到的 `env_user_tlb_mod_entry` 域指向的用户处理函数。这里需要通过系统调用告知内核自身的处理程序是 `cow_entry`。你还需要完成内核中的系统调用处理函数 `sys_set_tlb_mod_entry`，将进程控制块的 `env_user_tlb_mod_entry` 域设为传入的参数。

Exercise 4.12 完成 kern/syscall_all.c 中的 sys_set_tlb_mod_entry 函数。 ■

我们现在知道了页写入异常处理时会返回到用户空间的 cow_entry 函数，我们再来看这个函数会做些什么。

```
1 static void __attribute__((noreturn)) cow_entry(struct Trapframe *tf) {
2     u_int va = tf->cp0_badvaddr;
3     // ...
4     int r = syscall_set_trapframe(0, tf);
5     user_panic("syscall_set_trapframe returned %d", r);
6 }
```

从内核返回后，此时的第一个参数是由内核设置的，处于异常处理栈中，且指向一个由内核复制好的 Trapframe 结构体的底部。该函数从 Trapframe 中读取了 cp0_badvaddr 字段的值，这个值也正是 CPU 设置的发生页写入异常的地址 va，我们可以根据这个地址进行写时复制处理。在函数的最后，使用系统调用 syscall_set_trapframe 恢复了保存的现场。

Thinking 4.8 在用户态处理页写入异常，相比于在内核态处理有什么优势？ ■

说到这里，我们就要来实现 cow_entry 中真正进行处理的部分了。你需要完成这些任务：

1. 根据 vpt 中 va 所在页的页表项，判断其标志位是否包含 PTE_COW，是则进行下一步，否则调用 user_panic() 报错。
2. 分配一个新的临时物理页到临时地址 UCOW，使用 memcpy 将 va 页的数据拷贝到刚刚分配的页中。
3. 将发生页写入异常的地址 va 映射到临时页面上，注意设定好对应的页面标志位（即去除 PTE_COW 并恢复 PTE_D），然后解除临时地址 UCOW 的内存映射。

Exercise 4.13 填写 user/lib/fork.c 中的 cow_entry 函数。 ■

Thinking 4.9 请思考并回答以下几个问题：

- 为什么需要将 syscall_set_tlb_mod_entry 的调用放置在 syscall_exofork 之前？
- 如果放置在写时复制保护机制完成之后会有怎样的效果？

父进程还需要使用 syscall_set_tlb_mod_entry，设置子进程的页写入异常处理函数为 cow_entry。最后，父进程通过系统调用 syscall_set_env_status 设置子进程为可以运行的状态。在内核中实现

sys_set_env_status 函数时，不仅需要设置进程控制块的 env_status 域，还需要在 env_status 从 ENV_NOT_RUNNABLE 转换为 ENV_RUNNABLE 时将控制块加入到可调度进程的链表中，反之则从链表中移除。

Exercise 4.14 填写 kern/syscall_all.c 中的 sys_set_env_status 函数。 ■

说到这里我们需要整理一下思路，fork 中父进程在 syscall_exofork 后还需要做的事情有：

1. 遍历父进程地址空间，进行 `duppage`。
2. 设置子进程的异常处理函数，确保页写入异常可以被正常处理。
3. 设置子进程的 `env_status`，允许其被调度。

最后再将子进程的 `envid` 返回，`fork` 函数就完成了。

Exercise 4.15 填写 `user/lib/fork.c` 中的 `fork` 函数。

图 4.5 是页写入异常处理流程图，可作为原理理解和代码填写时的参考。

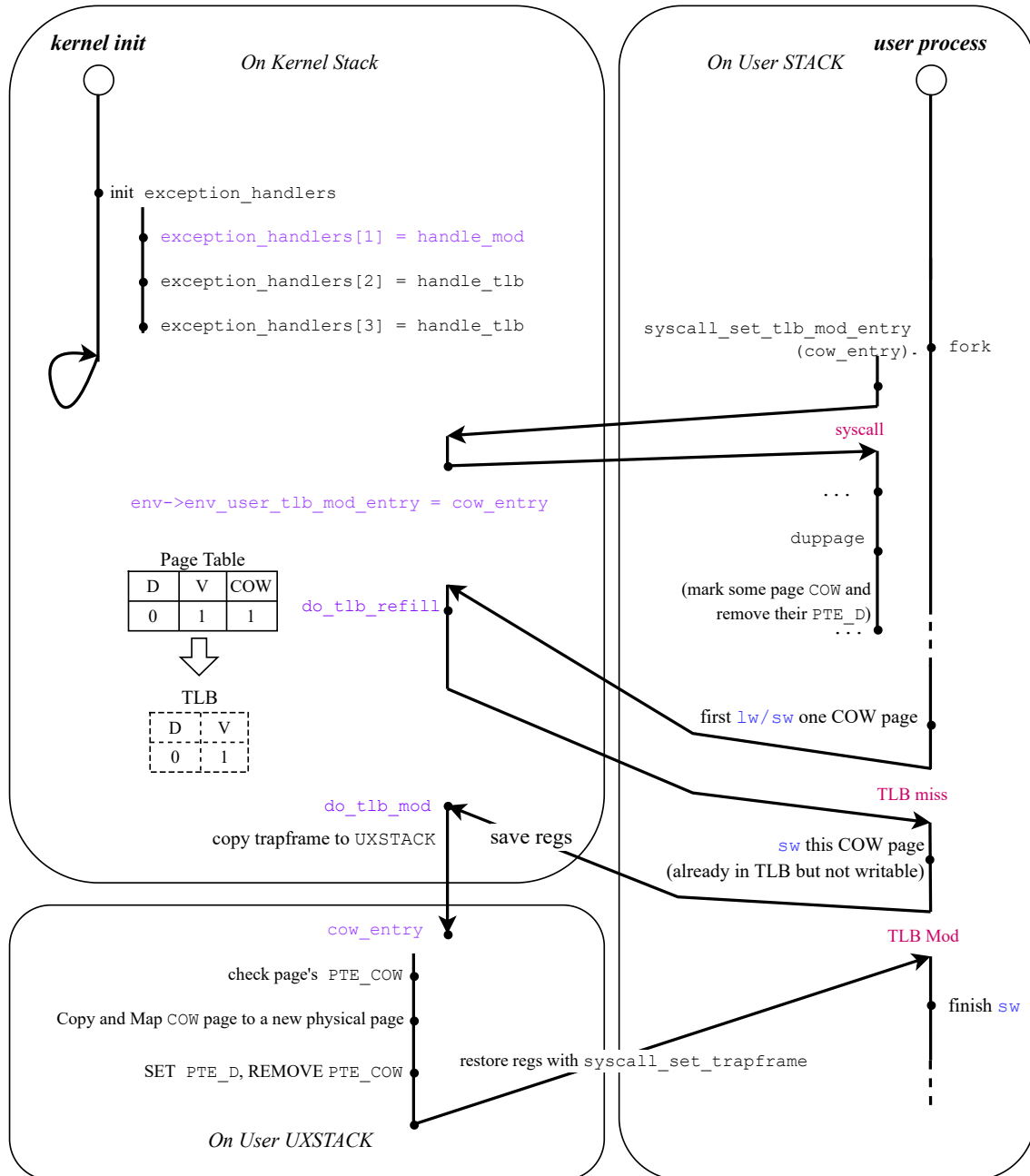


图 4.5: 页写入异常处理流程图

4.4.6 使用用户程序进行测试

您可以参照 `user` 目录下的 `tltest.c`、`fktest.c`、`pingpong.c` 等文件，编写自己的用户程序，测试系统调用、IPC 和 `fork` 等功能：

1. 在 `user` 目录下创建 `xxx.c`，加入 `#include <lib.h>`，并编写自己的测试逻辑
2. 为 `user/include.mk` 中的构建目标 `INITAPPS` 加上 `xxx.x`
3. 在 `init/init.c` 中用 `ENV_CREATE` 或者 `ENV_CREATE_PRIORITY` 创建用户进程，参数为 `user_xxx`
4. `make && make run`，即可观察到 `xxx.c` 的运行结果

4.5 实验正确结果

本次实验下有多个测试程序，最简单的单元测试是 `envid2env_check()`，在完成 `envid2env` 后，将 `envid2env_check()` 加入 `init/init.c` 即可测试，也可以使用 `make test lab=4_1` 直接构建测试。其参考输出如下：

```
1  envid2env() work well!
```

用户态的第一个测试程序是 `user/tltest.c`，在系统调用部分完成后，将 `ENV_CREATE(user_tltest)` 加入 `init/init.c`，使用 `make && make run` 即可测试，参考输出如下：

```
1  Smashing some kernel codes...
2  If your implementation is correct, you may see unknown exception here:
3  ...
4  panic at traps.c:24 (do_reserved): Unknown ExcCode 5
```

完成 `fork` 后，单独测试 `fork` 的程序是 `user/fktest.c`，使用方法同上。其参考输出如下：

```
1  this is father: a:1
2  this is father: a:1
3  this is father: a:1
4  ...
5      this is child :a:2
6      this is child :a:2
7      this is child :a:2
8  ...
9      this is child2 :a:3
10     this is child2 :a:3
11     this is child2 :a:3
```

其中三种文本段应当交替出现且永不停止。

另一个测试程序 `user/pingpong.c` 主要测试 `fork` 和进程间通信，参考输出如下：

```
1  @@@@send 0 from 800 to 1001
2  1001 am waiting.....
3  800 am waiting.....
4  1001 got 0 from 800
5
6  @@@@send 1 from 1001 to 800
7  1001 am waiting.....
8  800 got 1 from 1001
9
```

```

10  @@@@send 2 from 800 to 1001
11  800 am waiting.....
12  1001 got 2 from 800
13
14  @@@@send 3 from 1001 to 800
15  1001 am waiting.....
16  800 got 3 from 1001
17
18  @@@@send 4 from 800 to 1001
19  800 am waiting.....
20  1001 got 4 from 800
21
22  @@@@send 5 from 1001 to 800
23  1001 am waiting.....
24  800 got 5 from 1001
25
26  @@@@send 6 from 800 to 1001
27  800 am waiting.....
28  1001 got 6 from 800
29
30  @@@@send 7 from 1001 to 800
31  1001 am waiting.....
32  800 got 7 from 1001
33
34  @@@@send 8 from 800 to 1001
35  800 am waiting.....
36  1001 got 8 from 800
37
38  @@@@send 9 from 1001 to 800
39  1001 am waiting.....
40  800 got 9 from 1001
41
42  @@@@send 10 from 800 to 1001
43  [00000800] destroying 00000800
44  [00000800] free env 00000800
45  i am killed ...
46  1001 got 10 from 800
47  [00001001] destroying 00001001
48  [00001001] free env 00001001
49  i am killed ...

```

4.6 任务列表

- Exercise-完成 `msyscall` 函数
- Exercise-完成 `do_syscall` 函数
- Exercise-完成 `envid2env` 函数
- Exercise-实现 `sys_mem_alloc` 函数
- Exercise-实现 `sys_mem_map` 函数
- Exercise-实现 `sys_mem_unmap` 函数
- Exercise-实现 `sys_yield` 函数
- Exercise-实现 `sys_ipc_recv` 函数和 `sys_ipc_try_send` 函数
- Exercise-填写 `sys_exofork` 函数

- Exercise-填写 `duppage` 函数
- Exercise-完成 `do_tlb_mod` 函数
- Exercise-完成 `sys_set_tlb_mod_entry` 函数
- Exercise-完成 `cow_entry` 函数
- Exercise-填写 `sys_set_env_status` 函数
- Exercise-填写 `fork` 函数

4.7 实验思考

- 思考-系统调用的实现
- 思考-`envid2env` 的实现
- 思考-`mkenvid` 函数细节
- 思考-`fork` 的返回结果
- 思考-用户空间的保护
- 思考-`vpt` 的使用
- 思考-页写入异常-内核处理
- 思考-页写入异常-用户处理-1
- 思考-页写入异常-用户处理-2

5.1 实验目的

1. 了解文件系统的基本概念和作用。
2. 了解普通磁盘的基本结构和读写方式。
3. 了解实现设备驱动的方法。
4. 掌握并实现文件系统服务的基本操作。
5. 了解微内核的基本设计思想和结构。

在之前的实验中，我们把所有的程序和数据都存放在内存中。然而内存空间的大小是有限的，而且内存中的数据存在易失性问题。因此有些数据必须保存在磁盘、光盘等外部存储设备上，这些外部存储设备能够长期保存大量的数据，且便于将数据装载到不同进程的内存空间进行共享。为了便于管理和访问存放在外部存储设备上的数据，在操作系统中引入了文件系统。在文件系统中，文件是数据存储和访问的基本单位。对于用户而言，文件系统可以屏蔽访问外存数据的复杂性。

5.2 文件系统概述

计算机文件系统是一种存储和组织数据的方法，便于访问和查找数据。文件系统使用文件和树型目录的逻辑抽象屏蔽了底层硬盘和光盘等物理设备基于数据块进行存储和访问的复杂性。用户不必关心数据实际保存在硬盘（或者光盘）的哪个数据块上，只需要记住这个文件的所属目录和文件名。在写入新数据之前，用户不必关心硬盘上的哪个块是空闲的，硬盘上的存储空间管理（分配和释放）由文件系统自动完成，用户只需要记住数据被写入到了哪个文件中即可。

常见的文件系统通常基于硬盘和光盘等块存储设备，并维护文件在设备中的物理位置；而此外的某些文件系统仅仅是一种访问数据的接口，实际的数据在内存中或者通过网络协议（如 NFS、SMB、FTP 等）提供。

广义上，一切带标识的、在逻辑上有完整意义的字节序列都可以称为“文件”。文件系统将外部设备中的资源抽象为文件，从而可以统一管理外部设备，实现对数据的存储、组织、访问和修改等操作。在本实验中，我们拟实现一个精简的文件系统，其中需要对三种设备进行统一管理，即文件设备（file，即狭义的“文件”）、控制台（console）和管道（pipe）。其中，后两者将在下一个实验“管道与 Shell”中进行使用。

5.2.1 文件系统的设计与实现

在本次实验中，我们将要实现一个简单但结构完整的文件系统。整个文件系统包括以下几个部分：

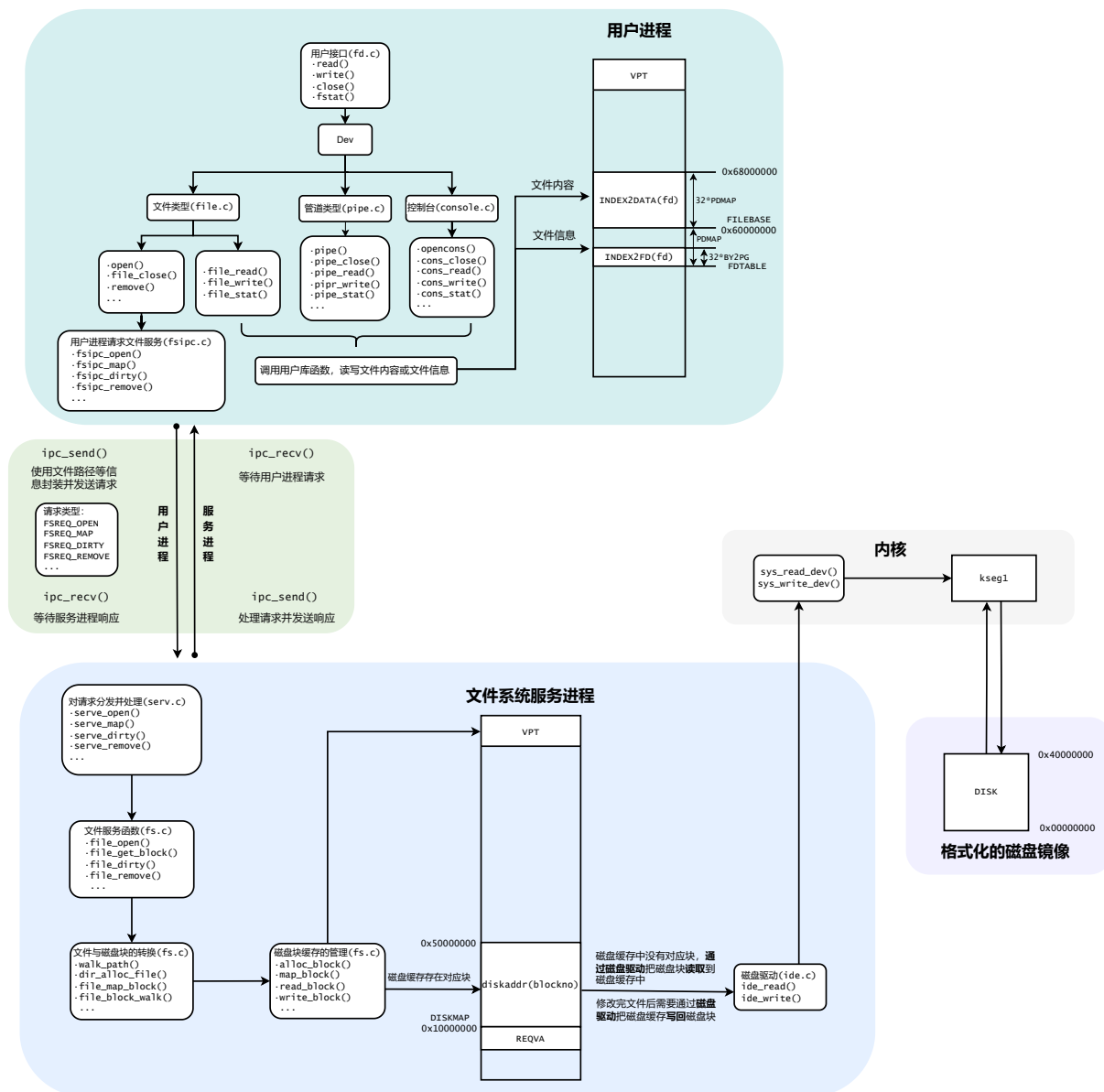


图 5.1: 文件系统总览图

1. **外部存储设备驱动** 通常，我们需要按一定顺序读写设备寄存器，来实现对外部设备的操作。为了将这种操作转化为具有通用、明确语义的接口，必须实现相应的驱动程序。在本部分，我们将实现 IDE 磁盘的用户态驱动程序，该驱动程序将通过系统调用的方式陷入内核，对磁盘镜像进行读写操作。

Note 5.2.1 IDE (Integrated Drive Electronics, 集成驱动器电子设备) 是过去主流的硬盘接口。

2. **文件系统结构** 在本部分，我们会实现模拟磁盘的驱动程序以及磁盘上和操作系统中的文件系统结构，并实现文件系统操作的相关函数。在我们的操作系统中，`fs` 目录下存放的是

文件系统服务程序的代码；而 `user/lib` 目录中的 `file.c`、`fd.c`、`fsipc.c` 等文件则存放了文件系统的用户库，这部分代码会被一同链接到用户程序中，允许用户程序调用其中的函数来操作文件系统。文件系统服务进程和其他用户进程之间使用 Lab4 中实现的 IPC 机制进行通信。

Note 5.2.2 MOS 中的文件系统服务进程实际上也是一个运行在用户态下的进程；不加说明时，下文中提到的“用户程序”通常指其他调用文件系统服务进程提供的 IPC 接口进行文件操作的程序，不包括由操作系统提供的文件系统服务程序。

3. **文件系统的用户接口** 在本部分，要求为用户提供接口和机制使得用户程序能够使用文件系统，这主要通过一个用户态的文件系统服务来实现。同时，我们将引入文件描述符等结构来抽象地表示一个进程所打开的文件，而不必关心文件实际的物理表示。

整个文件系统所涉及的代码文件比较多，此处我们概要介绍一些核心代码文件的主要功能，希望有助于理解文件系统实现的总体框架。这些代码实现在三个目录中：

- **tools** 目录中存放的是构建时辅助工具的代码。在之前的 Lab 中，我们实现了解析 ELF 文件的 `readelf` 工具；本 Lab 中，我们将在其中实现 `fsformat` 工具，并借助它来创建磁盘镜像。请注意，**tools** 目录下的代码仅用于 MOS 的构建，在宿主 Linux 环境（而非 MIPS 模拟器）中运行，也不会被编译进 MOS 的内核、用户库或用户程序中。
- **fs** 目录中存放的是文件系统服务进程的代码。我们在 `fs.c` 中实现文件系统的基本功能函数，在 `ide.c` 中通过系统调用与磁盘镜像进行交互。该进程的主干函数在 `serv.c` 中，通过 IPC 通信与用户进程 `user/lib/fsipc.c` 内的通信函数进行交互。
- **user/lib** 目录下存放了用户程序的库函数。在本 Lab 中，该目录下的 `fsipc.c` 实现了与文件系统服务进程的交互，`file.c` 实现了文件系统的用户接口；`fd.c` 中实现了文件描述符，允许用户程序使用统一的接口，抽象地操作磁盘文件系统中的文件，以及控制台和管道等虚拟的文件。

整个文件系统体现了 MOS 的微内核设计，包括下面三个部分：

1. 将传统操作系统的文件系统移出内核，使用用户态的文件系统服务程序以及一系列用户库来实现。即使它们崩溃，也不会影响到整个内核的稳定。其他用户进程通过进程间通信 (IPC) 来请求文件系统的相关服务。因此，在微内核中进程间通信 (IPC) 是一个十分重要的机制。
2. 操作系统将一些内核数据暴露到用户空间，使得进程不需要切换到内核态就能访问。MOS 将进程页表映射到用户空间，此处文件系统服务进程访问自身进程页表即可判断磁盘缓存中是否存在对应块。
3. 将传统操作系统的设备驱动移出内核，作为用户程序来实现。微内核体现在，内核在此过程中仅提供读写设备物理地址的系统调用。

接下来我们一一详细解读这些部分的实现。

5.3 IDE 磁盘驱动

为了在磁盘等外部设备上实现文件系统，我们必须为这些外部设备编写驱动程序。实际上，MOS 已经在 `kern/machine.c` 中实现了一个简单的控制台驱动程序，其中在内核态下使用了内存映射 I/O (MMIO) 技术，通过控制台实现了字符的输入输出。

本次要实现的硬盘驱动程序与已经实现的串口驱动，都采用 MMIO 技术编写驱动，不同之处在于，我们需要驱动的物理设备——IDE 磁盘功能更加复杂，并且本次要编写的驱动程序完全运行在用户空间。

下面我们首先介绍内存映射 I/O，之后再了解 IDE 磁盘的结构和操作，最后介绍磁盘驱动程序的编写。

5.3.1 内存映射 I/O (MMIO)

在“内存管理”实验中，我们已经了解了 MIPS 存储器地址映射的基本内容。几乎每一种外设都是通过读写设备上的寄存器来进行数据通信，外设寄存器也称为 **I/O 端口**，主要用来访问 I/O 设备。外设寄存器通常包括控制寄存器、状态寄存器和数据寄存器，这些寄存器被映射到指定的**物理地址**空间。例如，在 MALTA 中，console 设备被映射到 0x180003F8，IDE 控制器被映射到 0x180001F0，等等。

实验中使用的 MIPS 体系结构并没有复杂的 I/O 端口的概念，而是统一使用内存映射 I/O 的模型。在 MIPS 的内核地址空间中（kseg0 和 kseg1 段）实现了硬件级别的物理地址和内核虚拟地址的转换机制，其中，对 kseg1 段地址的读写不经过 MMU 映射，且不使用高速缓存，这正是外部设备驱动所需要的。由于我们是在模拟器上运行操作系统，I/O 设备的物理地址是完全固定的，因此我们可以通过简单地读写某些固定的内核虚拟地址来实现驱动程序的功能。

在之前的实验中，我们曾经使用 KADDR 宏把一个物理地址转换为 kseg0 段的内核虚拟地址，实际上是给物理地址加上 kseg0 的偏移值（即 0x80000000）。而在编写设备驱动的时候，我们需要将物理地址转换为 kseg1 段的内核虚拟地址，也就是给物理地址加上 kseg1 的偏移值（0xA0000000）。

Thinking 5.1 如果通过 kseg0 读写设备，那么对于设备的写入会缓存到 Cache 中。这是一种**错误**的行为，在实际编写代码的时候这么做会引发不可预知的问题。请思考：这么做会引发什么问题？对于不同种类的设备（如我们提到的串口设备和 IDE 磁盘）的操作会有差异吗？可以从缓存的性质和缓存更新的策略来考虑。 ■

以我们编写完成的串口设备驱动为例，MALTA 提供的 console 设备是一个典型的 NS16550 设备，其基地址为 0x180003F8，设备寄存器映射如表 5.1 所示。

表 5.1: MALTA NS16550 设备内存映射

偏移	效果
0x00	读：非阻塞的读取一个字符，如果没有未读取的字符则返回上次读取的结果
	写：输出一个字符 ch
0x05	读：获取设备状态，返回 1 说明设备上有未读取的数据就绪可用

现在，我们通过往内存的 (0x180003F8+0xA0000000) 地址写入字符，就能在 shell 中看到对应的输出。

kern/machine.c 中的 printcharc 函数的实现如下所示：

```

1 void printcharc(char ch) {
2     *((volatile char *) (KSEG1 + MALTA_SERIAL_DATA)) = ch;
3 }

```

这个函数首先通过 KSEG1 + MALTA_SERIAL_DATA 定位到了字符需要被写入到的地址，为了对这个地址的数据进行操作需要将其强制转化为类型 char 的指针。注意，此处的类型 char 需要

使用 `volatile` 限定。因为我们对于 I/O 设备的操作，需要直接操作设备，如果不使用 `volatile`，编译器可能会对该地址的数据都访问做出优化，包括让这个访问通过 Cache，这样对于 I/O 设备的地址的访问将不能直接操作到设备导致错误。转化为指针之后，通过解引用，可以对这个地址的数据进行修改。

而在本次实验中，我们需要编写的 IDE 磁盘驱动程序位于用户空间，用户态进程若是直接读写内核虚拟地址将会由处理器引发一个地址错误 (ADEL/S)。所以对于设备的读写必须通过系统调用来实现。这里我们引入了 `sys_write_dev` 和 `sys_read_dev` 两个系统调用来实现设备的读写操作。这两个系统调用以用户虚拟地址、设备的物理地址和读写的长度（按字节计数），作为参数，在内核空间中完成 I/O 操作。

Exercise 5.1 请根据 `kern/syscall_all.c` 中的说明，完成 `sys_write_dev` 函数以及 `sys_read_dev` 函数的实现。

编写这两个系统调用时需要注意物理地址与内核虚拟地址之间的转换。

同时还要检查物理地址的有效性，在实验中允许访问的地址范围为：

console: [0x180003F8, 0x18000418), disk: [0x180001F0, 0x180001F8)，当出现越界时，应返回指定的错误码。

Exercise 5.2 在 `user/lib/syscall_lib.c` 中完成用户态的相应系统调用的接口。

5.3.2 IDE 磁盘

在我们的实验中，MALTA 提供的“磁盘”是一个 Intel PIIX4 IDE 控制器，其上连接 IDE 磁盘供操作系统使用。具体关于此 IDE 设备的信息，可以参考 [Intel 82371AB PCI ISA IDE Xcelerator \(PIIX4\)](#) (第 192 页 9.2 节)。在此基础上，我们就可以实现 MOS 的文件系统了。接下来，我们将了解一些读写 IDE 磁盘的基础知识。

磁盘的物理结构

下面简单介绍与磁盘相关的基本知识，首先是几个基本概念：

1. 扇区 (sector)：磁盘盘片被划分成很多扇形的区域，这些区域叫做扇区。扇区是磁盘执行读写操作的单位，一般是 512 字节。扇区的大小是一个磁盘的硬件属性。
2. 磁道 (track)：盘片上以盘片中心为圆心，不同半径的同心圆。
3. 柱面 (cylinder)：硬盘中，不同盘片相同半径的磁道所组成的圆柱面。
4. 磁头 (head)：每个磁盘有两个面，每个面都有一个磁头。当对磁盘进行读写操作时，磁头在盘片上快速移动。

典型磁盘的基本结构如图 5.2 所示：

IDE 磁盘操作

前文中我们提到过，扇区 (sector) 是磁盘读写的基本单位，MALTA 上的 PIIX4 也提供了对扇区进行操作的基本方法。通过读写 PIIX4 的特定寄存器，我们可以实现以扇区为最小单元的读写。MALTA 平台上的 PIIX4 磁盘控制器基地址为 0x180001F0，其 I/O 相关寄存器相对于该地址的偏移和对应的功能如表 5.2 所示。

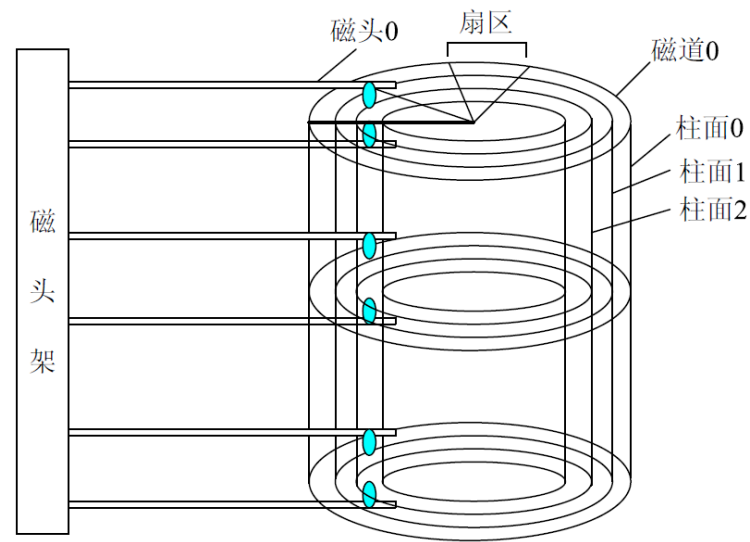


图 5.2: 磁盘结构示意图

表 5.2: PIIX4 I/O 关键寄存器映射

偏移	寄存器功能	数据位宽
0x0	读/写：向磁盘中读/写数据，从 0 字节开始逐个读出/写入	4 字节
0x1	读：设备错误信息；写：设置 IDE 命令的特定参数（实验中不涉及）	1 字节
0x2	写：设置一次需要操作的扇区数量	1 字节
0x3	写：设置目标扇区号的 [7:0] 位 (LBAL)	1 字节
0x4	写：设置目标扇区号的 [15:8] 位 (LBAM)	1 字节
0x5	写：设置目标扇区号的 [23:16] 位 (LBAH)	1 字节
0x6	写：设置目标扇区号的 [27:24] 位，配置扇区寻址模式 (CHS/LBA)，设置要操作的磁盘编号	1 字节
0x7	读：获取设备状态；写：配置设备工作状态	1 字节

在 MOS 中，我们可以挂载两块 IDE 磁盘（磁盘编号为 0 和 1），但实际上我们只使用到了磁盘编号为 0 的一块磁盘。

对于磁盘寻址，我们可以按照柱面-磁头-扇区（Cylinder-Head-Sector, CHS）的方式来定位一个扇区。这种寻址方式符合磁盘的物理结构，但是显然三个参数导致了寻址的复杂性。在该模式下，PIIX4 I/O 的寄存器映射含义与表 5.2 中不完全相同，具体可以参考前文给出的文档。

由于 CHS 模式下不方便进行寻址，因此在实验中我们采用逻辑块寻址（Logical Block Addressing, LBA）的方式来进行扇区寻址。在 LBA 模式下，IDE 设备将磁盘看作一个线性的字节序列，每个扇区都有一个唯一的编号，只需要设置目标扇区编号，就可以完成磁盘的寻址。在我们的实验中，扇区编号有 28 位，因此最多可以寻址 2^{28} 个扇区，即 128 GB 的磁盘空间。该模式下，磁盘操作参数和寄存器映射的关系如图 5.3 所示。

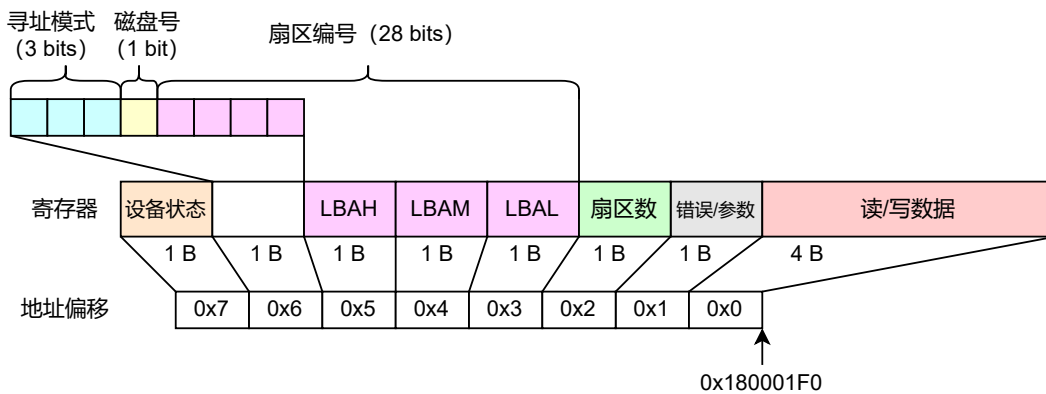


图 5.3: LBA 模式下参数和寄存器映射关系图

5.3.3 驱动程序编写

通过对 `printcharc` 函数的实现的分析，我们已经掌握了 I/O 操作的基本方法，那么，读写 IDE 磁盘的相关代码也就不难理解了。

当我们在磁盘的指定位置读取或写入一个扇区时，需要调用 `read_sector` 函数将磁盘中对扇区的数据读到设备缓冲区中，或 `write_sector` 函数来缓冲区中的数据写入磁盘。注意，与控制台串口相同，磁盘操作中所有的地址操作都需要将物理地址转换成虚拟地址。此处设备基地址对应的 `kseg1` 的内核虚拟地址是 `0xB80001F0`。具体读写磁盘的流程如图 5.4 所示。

在磁盘读写的流程中，我们需要反复检查 IDE 设备是否已经就绪。这是由于 IDE 外设一般不能立即完成数据操作，需要 CPU 检查 IDE 状态并等待操作完成。为此我们构建了一个检查 IDE 状态的帮手函数 `wait_ide`，用于等待 IDE 上的操作就绪。

```

1  u_int wait_ide() {
2      u_int flag;
3      while (1) {
4          flag = *((volatile u_char*)(MALTA_IDE_STATUS + 0xA0000000));
5          if ((flag & MALTA_IDE_BUSY) == 0) {
6              break;
7          }
8      }
9      return flag;
10 }

```

在 IDE 设备就绪后，我们就可以对其进行读写操作了。

首先，设置操作扇区的数目，这里我们只操作一个扇区，因此设置为 1。

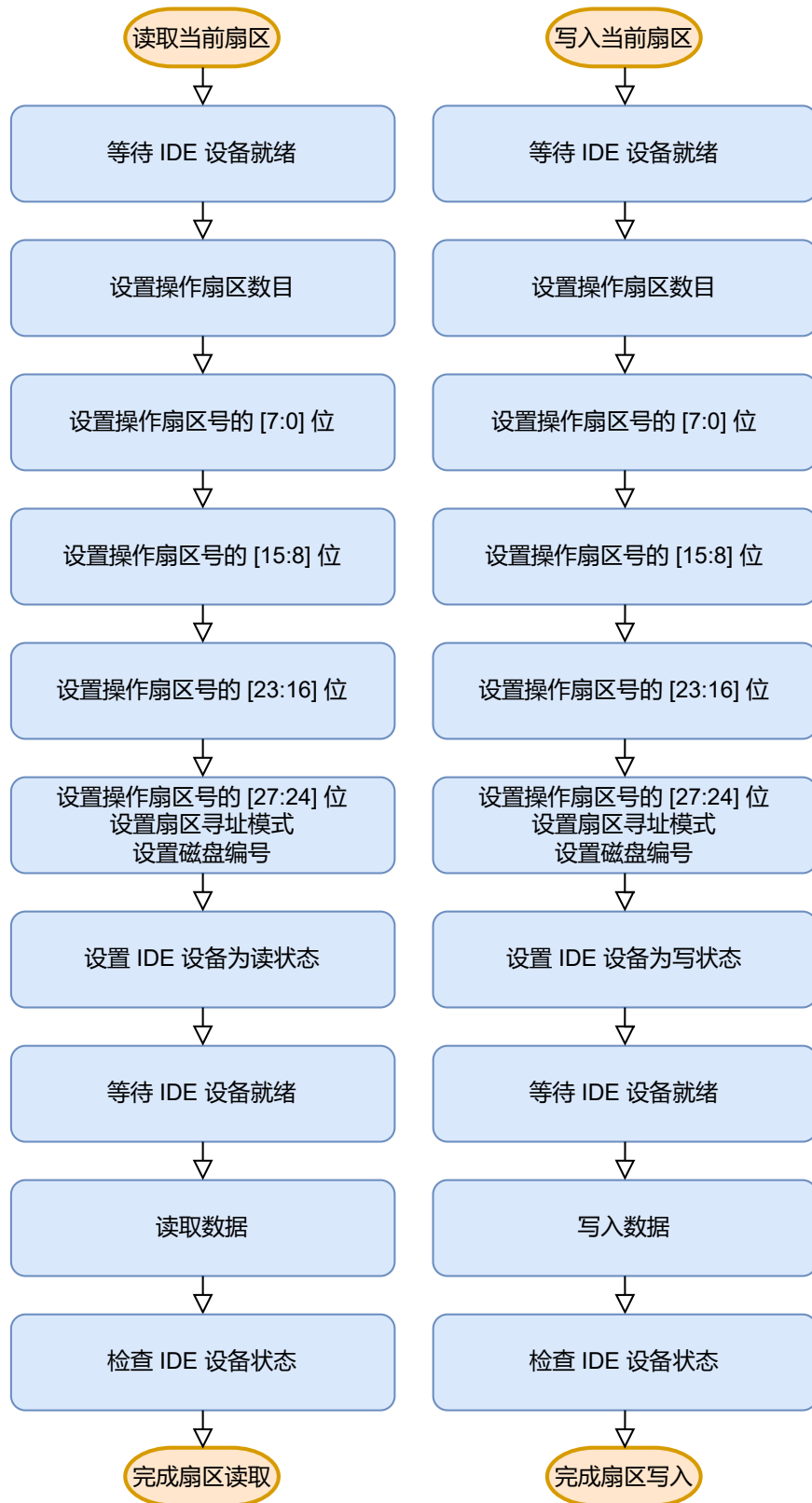


图 5.4: 硬盘扇区读取、写入流程图

接下来，我们依次设置操作扇区号。本实验中使用的 IDE 设备无法一次性写入操作扇区号，因此需要单独设置扇区号的各位。特别地，在设置操作扇区号的 [27:24] 位时，还需要同时设置扇区寻址模式和磁盘编号，因此需要通过位运算将各值组合，并一齐写入对应地址。

完成一系列设置后，再次等待 IDE 设备准备就绪，并在此之后，通过系统调用读取或写入扇区。由于本实验中使用的 IDE 设备每次仅能读取或写入 4 字节，因此需要通过一个循环完成整个扇区的读取或写入，即连续向相同的地址读取或写入 4 字节。

最后，检查 IDE 设备状态，确认扇区读取和写入成功。

具体地，对于读取扇区，我们可以实现 `read_sector` 函数，读取 `diskno` 号磁盘上的 `secno` 号扇区到 `dst` 指向的地址。

```

1  void read_sector(int diskno, int secno, void* dst)
2  {
3      wait_ide();
4
5      *((volatile u_char*)(MALTA_IDE_NSECT + 0xA0000000)) = 1;
6      *((volatile u_char*)(MALTA_IDE_LBAL + 0xA0000000)) = secno & 0xff;
7      *((volatile u_char*)(MALTA_IDE_LBAM + 0xA0000000)) = (secno >> 8) & 0xff;
8      *((volatile u_char*)(MALTA_IDE_LBAH + 0xA0000000)) = (secno >> 16) & 0xff;
9      *((volatile u_char*)(MALTA_IDE_DEVICE + 0xA0000000)) =
10     ((secno >> 24) & 0x0f) | MALTA_IDE_LBA | (diskno << 4);
11     *((volatile u_char*)(MALTA_IDE_STATUS + 0xA0000000)) = MALTA_IDE_CMD_PIO_READ;
12     wait_ide();
13
14     for(int i = 0 ; i < SECT_SIZE; i += 4) {
15         memcpy(dst + i, (void*)(MALTA_IDE_DATA + 0xA0000000), 4);
16     }
17 }

```

类似的，我们也可以实现 `write_sector` 函数，将 `src` 指向的一个扇区的数据写入到 `diskno` 号磁盘的 `secno` 号扇区。

```

1  void write_sector(int diskno, int secno, void* src)
2  {
3      wait_ide();
4
5      *((volatile u_char*)(MALTA_IDE_NSECT + 0xA0000000)) = 1;
6      *((volatile u_char*)(MALTA_IDE_LBAL + 0xA0000000)) = secno & 0xff;
7      *((volatile u_char*)(MALTA_IDE_LBAM + 0xA0000000)) = (secno >> 8) & 0xff;
8      *((volatile u_char*)(MALTA_IDE_LBAH + 0xA0000000)) = (secno >> 16) & 0xff;
9      *((volatile u_char*)(MALTA_IDE_DEVICE + 0xA0000000)) =
10     ((secno >> 24) & 0x0f) | MALTA_IDE_LBA | (diskno << 4);
11     *((volatile u_char*)(MALTA_IDE_STATUS + 0xA0000000)) = MALTA_IDE_CMD_PIO_WRITE;
12     wait_ide();
13
14     for(int i = 0 ; i < SECT_SIZE; i += 4) {
15         memcpy((void*)(MALTA_IDE_DATA + 0xA0000000), src + i, 4);
16     }
17 }

```

Exercise 5.3 参考以上介绍以及内核态磁盘驱动实现，使用系统调用完成 `fs/ide.c` 中的 `ide_read` 和 `ide_write` 函数，实现用户态下对磁盘的读写操作。教程中展示的驱动函数只能运行于内核态，你需要通过系统调用的方式来实现其对应的用户态版本。例如，`wait_ide` 函数的用户态版本为 `fs/ide.c` 中的 `wait_ide_ready`。 ■

实现了磁盘驱动, 我们可以尝试对磁盘进行读写测试。QEMU 允许通过启动参数来指定 IDE 磁盘镜像。编译生成的磁盘镜像文件位于 `target/fs.img`, 我们可以在运行命令中加上 `-drive id=ide0,file=target/fs.img,if=ide,format=raw` 挂载该磁盘镜像。实验中我们挂载了两个磁盘镜像, 其中一个是我们编译生成的磁盘镜像, 另一个是空的磁盘镜像, 完整命令如下:

```

1  qemu-system-mipsel \
2    -cpu 4Kc \
3    -m 64 -nographic \
4    -M malta \
5    -drive id=ide0,file=target/fs.img,if=ide,format=raw \
6    -drive id=ide1,file=target/empty.img,if=ide,format=raw \
7    -no-reboot \
8    -kernel target/mos

```

我们通过 Makefile 简化了这个过程, 可以直接运行 `make run` 来执行上述命令。

5.4 文件系统结构

实现了 IDE 磁盘的驱动, 我们就有了在磁盘上实现文件系统的基础。接下来我们设计整个文件系统的结构, 并在磁盘和操作系统中分别实现对应的结构。

5.4.1 磁盘文件系统布局

磁盘空间的基本布局如图 5.5 所示。

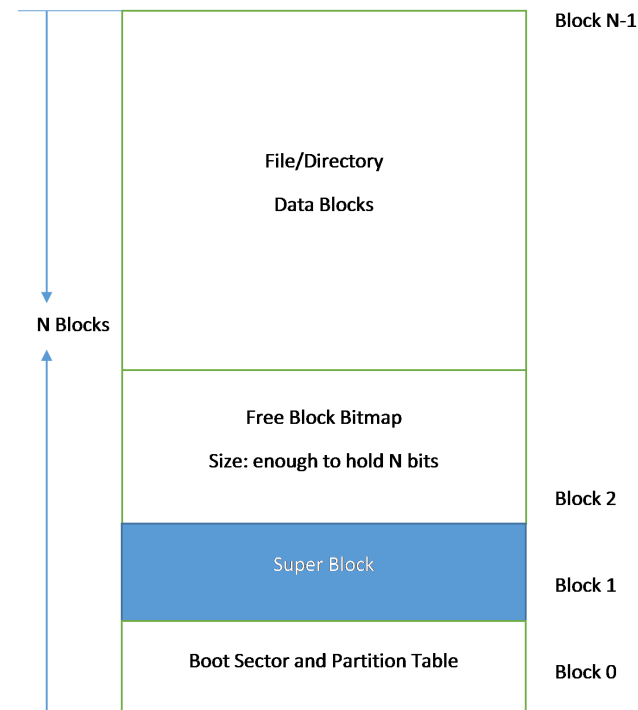


图 5.5: 磁盘空间布局示意图

图中出现的 Block 是磁盘块。不同于扇区, 磁盘块是一个虚拟概念, 是操作系统与磁盘交互的最小单位; 操作系统将相邻的扇区组合在一起, 形成磁盘块进行整体操作, 减小了因扇区过多的

带来的寻址困难；磁盘块的大小由操作系统决定，一般由 2 的幂次个扇区构成。而扇区是真实存在的，是磁盘读写的基本单位，与操作系统无关。

从图 5.5 中可以看到，MOS 操作系统把磁盘最开始的一个磁盘块（4096 字节）当作引导扇区和分区表使用。接下来的一个磁盘块作为超级块（Super Block），用来描述文件系统的基本信息，如 Magic Number、磁盘大小以及根目录的位置。

MOS 操作系统中超级块的结构如下：

```
1 struct Super {
2     u_int s_magic;      // Magic number: FS_MAGIC
3     u_int s_nblocks;    // Total number of blocks on disk
4     struct File s_root; // Root directory node
5 };
```

其中每个域的意义如下：

- `s_magic`：魔数，为一个常量，用于标识该文件系统。
- `s_nblocks`：记录本文件系统有多少个磁盘块，在本文件系统中为 1024。
- `s_root`：根目录，其 `f_type` 为 `FTYPE_DIR`，`f_name` 为 “/”。

通常采用两种数据结构来管理可用的资源：链表和位图。在 Lab2 “内存管理” 和 Lab3 “进程与异常” 实验中，我们使用了链表来管理空闲内存资源和进程控制块。在文件系统中，我们将使用位图 (Bitmap) 法来管理空闲的磁盘资源，用一个二进制位 bit 标识磁盘中的一个磁盘块的使用情况（1 表示空闲，0 表示占用）。

这里我们参考 `tools/fsformat.c` 来介绍文件系统标记空闲块的机制。`tools/fsformat.c` 是用于创建符合我们定义的文件系统镜像的工具，可以将多个文件按照内核所定义的文件系统写入到磁盘镜像中。通过观察头文件和 `tools/Makefile`，我们可以看出，`tools/fsformat.c` 的编译过程与其他文件有所不同，其使用的是 Linux 下的 gcc 编译器，而非交叉编译器。编译生成的 `fsformat` 运行于 Linux 宿主机上，专门用于创建磁盘镜像文件。生成的镜像文件 `fs.img` 可以模拟与真实的磁盘文件设备的交互。

在写入文件之前，`tools/fsformat.c` 的 `init_disk` 函数，将所有的块都标为空闲块：

```
1     nbitblock = (NBLOCK + BLOCK_SIZE_BIT - 1) / BLOCK_SIZE_BIT;
2     for (i = 0; i < nbitblock; ++i) {
3         disk[2 + i].type = BLOCK_BMAP;
4     }
5     for (i = 0; i < nbitblock; ++i) {
6         memset(disk[2 + i].data, 0xff, BLOCK_SIZE);
7     }
8     if (NBLOCK != nbitblock * BLOCK_SIZE_BIT) {
9         diff = NBLOCK % BLOCK_SIZE_BIT / 8;
10        memset(disk[2 + (nbitblock - 1)].data + diff, 0x00, BLOCK_SIZE - diff);
11    }
```

`nbitblock` 表示为了用位图标识整个磁盘上所有块的使用情况所需要的磁盘块（bitblock，位图块）的数量。紧接着，我们使用 `memset` 将位图中的每一个字节（Byte）都设成 `0xff`，即将所有位图块的每一位都设为 1，表示磁盘块处于空闲状态。如果位图还有剩余，不能将最后一块位图块中靠后的一部分内容标记为空闲，因为这些位所对应的磁盘块并不存在，是不可使用的。因此，将所有的位图块的每一位都置为 1 之后，还需要根据实际情况，将位图不存在的部分设为 0。

相应地，在 MOS 操作系统中，文件系统也需要根据位图来判断和标记磁盘的使用情况。`fs/fs.c` 中的 `block_is_free` 函数就用来通过位图中的特定位来判断指定的磁盘块是否被占用。

```

1  int block_is_free(u_int blockno)
2  {
3      if (super == 0 || blockno >= super->s_nblocks) {
4          return 0;
5      }
6      if (bitmap[blockno / 32] & (1 << (blockno % 32))) {
7          return 1;
8      }
9      return 0;
10 }

```

Exercise 5.4 文件系统需要负责维护磁盘块的申请和释放，在回收一个磁盘块时，需要更改位图中的标志位。如果要将一个磁盘块设置为 **free**，只需要将位图中对应位的值设置为 1 即可。请完成 **fs/fs.c** 中的 **free_block** 函数，实现这一功能。同时思考为什么参数 **blockno** 的值不能为 0？

```

1  // Overview:
2  // Mark a block as free in the bitmap.
3  void free_block(u_int blockno) {
4      // You can refer to the function 'block_is_free' above.
5      // Step 1: If 'blockno' is invalid (0 or >= the number of blocks in 'super'), return.
6
7      // Step 2: Set the flag bit of 'blockno' in 'bitmap'.
8      // Hint: Use bit operations to update the bitmap, such as b[n / W] |= 1 << (n % W).
9  }

```

5.4.2 文件系统详细结构

操作系统要想管理一类资源，就得有相应的数据结构。对于描述和管理文件来说，一般使用文件控制块（File 结构体）。其定义如下：

```

1  struct File {
2      char f_name[MAXNAMELEN]; // filename
3      uint32_t f_size;          // file size in bytes
4      uint32_t f_type;          // file type
5      uint32_t f_direct[NDIRECT];
6      uint32_t f_indirect;
7
8      struct File *f_dir; // the pointer to the dir where this file is in, valid only in memory.
9      char f_pad[FILE_STRUCT_SIZE - MAXNAMELEN - (3 + NDIRECT) * 4 - sizeof(void *)];
10 } __attribute__((aligned(4), packed));

```

结合文件控制块的示意图 5.6，我们对各个域进行解读：**f_name** 为文件名称，文件名的最大长度 **MAXNAMELEN** 值为 128。**f_size** 为文件的大小，单位为字节。**f_type** 为文件类型，有普通文件 (**FTYPE_REG**) 和目录 (**FTYPE_DIR**) 两种。**f_direct[NDIRECT]** 为文件的直接指针，每个文件控制块设有 10 个直接指针，用来记录文件的数据块在磁盘上的位置。每个磁盘块的大小为 4KB，也就是说，这十个直接指针能够表示最大 40KB 的文件，而当文件的大小大于 40KB 时，就需要用到间接指针。**f_indirect** 指向一个间接磁盘块，用来存储指向文件内容的磁盘块的指针。为了简化计算，我们不使用间接磁盘块的前十个指针。**f_dir** 指向文件所属的文件目录。**f_pad** 则是为了让整数个文件结构体占用一个磁盘块，填充结构体中剩下的字节。

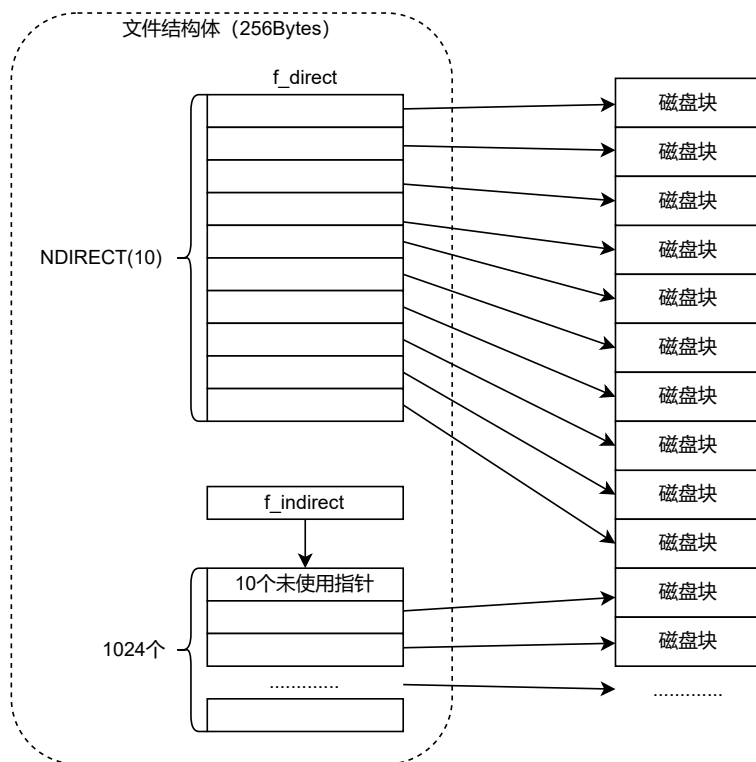


图 5.6: 文件控制块

Note 5.4.1 我们的文件系统中的文件控制块只使用了一级间接指针域，也只有一个。而在实际的文件系统中，为了支持更大的文件，通常会使用多个间接磁盘块，或使用多级间接磁盘块。MOS 操作系统内核在这一点上做了极大的简化。

对于普通的文件，其指向的磁盘块存储着文件内容，而对于目录文件来说，其指向的磁盘块存储着该目录下各个文件对应的文件控制块。当我们要查找某个文件时，首先从超级块中读取根目录的文件控制块，然后沿着目标路径，挨个查看当前目录所包含的文件是否与下一级目标文件同名，如此便能查找到最终的目标文件。

Thinking 5.2 查找代码中的相关定义，试回答一个磁盘块中最多能存储多少个文件控制块？一个目录下最多能有多少个文件？我们的文件系统支持的单个文件最大为多大？ ■

请阅读 `tools/fsformat.c` 和 `fs/Makefile`，掌握如何将文件和目录按照文件系统的格式写入磁盘，了解文件系统结构的具体细节，学会添加自定义文件进入磁盘镜像。（`fsformat.c` 中的主函数十分灵活，可以通过修改命令行参数来生成不同的镜像文件）

Exercise 5.5 参照文件系统的设计，完成 `fsformat.c` 中的 `create_file` 函数，并阅读 `write_directory` 函数（代码已在源文件中给出，不作为考查点），实现将一个文件或指定目录下的文件按照目录结构写入到 `target/fs.img` 的功能。 ■

5.4.3 块缓存

块缓存指的是借助虚拟内存来实现磁盘块缓存的设计。由于对于硬盘 I/O 操作一次只能读取少量数据，同时频繁读写磁盘会拖慢系统，故常用块缓存将数据暂存于内存，减少磁盘访问次数，从而加快数据读取。MOS 操作系统中，文件系统服务是一个用户进程（将在下文介绍），一

个进程可以拥有 4GB 的虚拟内存空间，将 DISKMAP 到 DISKMAP+DISKMAX 这一段虚存地址空间 (0x10000000-0x4FFFFFFF) 作为缓冲区，当磁盘读入内存时，用来映射相关的页。DISKMAP 和 DISKMAX 的值定义在 fs/serv.h 中：

```
1  #define DISKMAP    0x10000000
2  #define DISKMAX    0x40000000
```

Thinking 5.3 请思考，在满足磁盘块缓存的设计的前提下，我们实验使用的内核支持的最大磁盘大小是多少？

为了建立起磁盘地址空间和进程虚存地址空间之间的缓存映射，我们采用的设计如图 5.7 所示。

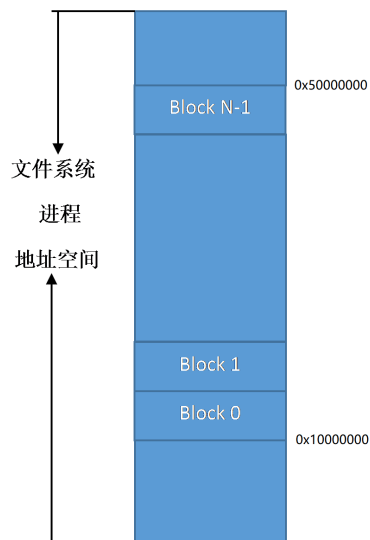


图 5.7: 块缓存示意图

Exercise 5.6 fs/fs.c 中的 disk_addr 函数用来计算指定磁盘块对应的虚存地址。完成 disk_addr 函数，根据一个块的序号 (block number)，计算这一磁盘块对应的虚存的起始地址。(提示：fs/serv.h 中的宏 DISKMAP 和 DISKMAX 定义了磁盘映射虚存的地址空间)。

当把一个磁盘块中的内容载入到内存中时，需要为之分配对应的物理内存；当结束使用这一磁盘块时，需要释放对应的物理内存以回收操作系统资源。fs/fs.c 中的 map_block 函数和 unmap_block 函数实现了这一功能。

Exercise 5.7 实现 map_block 函数，检查指定的磁盘块是否已经映射到内存，如果没有，分配一页内存来保存磁盘上的数据。相应地，完成 unmap_block 函数，用于解除磁盘块和物理内存之间的映射关系，回收内存。(提示：注意磁盘虚拟内存地址空间和磁盘块之间的对应关系)。

read_block 函数和 write_block 函数用于读写磁盘块。read_block 函数将指定编号的磁盘块读入到内存中，首先检查这块磁盘块是否已经在内存中，如果不在，先分配一页物理内存，然后调用 ide_read 函数来读取磁盘上的数据到对应的虚存地址处。

file_get_block 函数用于将某个指定的文件指向的磁盘块读入内存。其主要分为 2 个步

骤：首先为即将读入内存的磁盘块分配物理内存，然后使用 `read_block` 函数将磁盘内容以块为单位读入内存中的相应位置。这两个步骤对应的函数都借助了系统调用来完成。

在完成块缓存部分之后我们就可以实现文件系统中的一些文件操作了。

Exercise 5.8 补全 `dir_lookup` 函数，查找某个目录下是否存在指定的文件。
(使用 `file_get_block` 函数)

这里我们给出了文件系统结构中部分函数可能的调用参考（图 5.8），希望同学们认真理解每个文件、函数的作用和之间的关系。

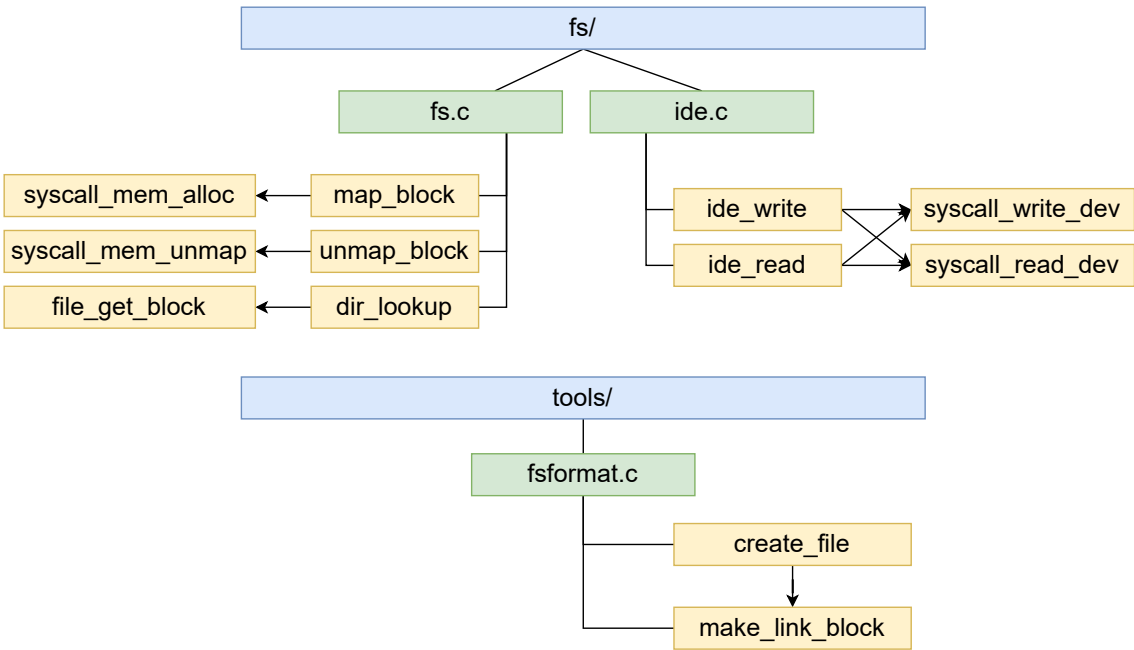


图 5.8: fs/下部分函数调用关系参考

Thinking 5.4 在本实验中，`fs/serv.h`、`user/include/fs.h` 等文件中出现了许多宏定义，试列举你认为较为重要的宏定义，同时进行解释，并描述其主要应用之处。

5.5 文件系统的用户接口

在文件系统建立之后，还需要向用户提供相关的使用接口。MOS 操作系统内核符合一个典型的微内核的设计，文件系统属于用户态进程，以服务的形式供其他进程调用。这个过程中，不仅涉及了不同进程之间通信的问题，也涉及了文件系统如何隔离底层的文件系统实现，抽象地表示一个文件的问题。首先，我们引入文件描述符（file descriptor）作为用户程序管理、操作文件的基础。

5.5.1 文件描述符

Note 5.5.1 `fd` 即 File Descriptor，是系统给用户提供的整数，用于其在描述符表 (Descriptor Table) 中进行索引。我们在作为操作系统的使用者进行文件 I/O 编程时，使用 `open` 在描述符表的指定位置存放被打开文件的信息；使用 `close` 将描述符表中指定位置的文件信息释放；在 `write` 和 `read` 时修改描述符表指定位置的文件信息。这里的“指定位置”即文件

描述符 `fd`。在这一环节中，你将作为操作系统的设计者，体会和参与构建文件描述符背后的精巧结构。

当用户进程试图打开一个文件时，文件系统服务进程需要一个文件描述符来存储文件的基本信息和用户进程中关于文件的状态；同时，文件描述符也起到描述用户对于文件操作的作用。当用户进程向文件系统发送打开文件的请求时，文件系统进程会将这些基本信息记录在内存中，然后由操作系统将用户进程请求的地址映射到同一个存储了文件描述符的物理页上（此部分代码位于 `serv.c` 的 `serve_open` 函数内），因此一个文件描述符至少需要独占一页的空间。当用户进程获取了文件大小等基本信息后，再次向文件系统发送请求将文件内容映射到指定内存空间中。

阅读 `user/lib/file.c` 时，我们会发现很多函数中都会将一个 `struct Fd *` 型的指针转换为 `struct Filefd *` 型的指针，这是基于 C 语言中关于指针的强制类型转换来实现的。该类型转换并不改变指针的值，仅改变程序对地址处数据的解释方式两个结构体的定义如下：

```

1  struct Fd {
2      u_int fd_dev_id;
3      u_int fd_offset;
4      u_int fd_omode;
5  };
6
7  struct Filefd {
8      struct Fd f_fd;
9      u_int f_fileid;
10     struct File f_file;
11 };

```

对于类型为 `Fd` 的一段内存，该段内存开头被解释为三个 `u_int` 类型的数据。将其强制转换为 `Filefd` 类型之后，由于 `Filefd` 第一个成员是 `Fd` 类型，对这段内存开始部分的解释依旧可以是三个 `u_int` 类型的数据（通过 `f_fd` 成员即可访问到）。同时第二个成员开始的内存将会按照 `Filefd` 的定义进行解释。若将 `Filefd` 类型强制转换为 `Fd` 将会遮盖掉不同细分类别的文件描述符的不同，有利于统一操作。

Exercise 5.9 请完成 `user/lib/file.c` 中的 `open` 函数。（提示：若成功打开文件，则该函数返回文件描述符的编号）。 ■

当要读取一个大文件中间的一小部分内容时，一个简单的做法是从头开始查找，但这样的开销很大；此外，在多次读写同一文件描述符期间，我们也希望能够从文件中前一次读写完毕的位置开始，继续读写数据。因此，文件描述符中需要维护一个指针，以帮助我们在文件中定位；在实现 `read`、`write` 和 `seek` 等操作时，也需要更新该指针的值，

Exercise 5.10 参考 `user/lib/fd.c` 中的 `write` 函数，完成 `read` 函数。 ■

Thinking 5.5 在 Lab4 “系统调用与 `fork`” 的实验中我们实现了极为重要的 `fork` 函数。那么 `fork` 前后的父子进程是否会共享文件描述符和定位指针呢？请在完成上述练习的基础上编写一个程序进行验证。 ■

Thinking 5.6 请解释 `File`, `Fd`, `Filefd` 结构体及其各个域的作用。比如各个结构体会在哪些过程中被使用，是否对应磁盘上的物理实体还是单纯的内存数据等。说明形式自定，要求简洁明了，可大致勾勒出文件系统数据结构与物理实体的对应关系与设计框架。

5.5.2 文件系统服务

MOS 操作系统中的文件系统服务通过 IPC 的形式供其他进程调用，进行文件读写操作。具体来说，在内核开始运行时，就启动了文件系统服务进程 `ENV_CREATE(fs_serv)`，用户进程需要进行文件操作时，使用 `ipc_send`、`ipc_rcv` 与 `fs_serv` 进行交互，完成操作。`fs/serv.c` 中服务进程的主函数首先调用了 `serve_init` 函数准备好全局的文件打开记录表 `opentab`，然后调用 `fs_init` 函数来初始化文件系统。`fs_init` 函数首先通过读取超级块的内容获知磁盘的基本信息，然后检查磁盘是否能够正常读写，最后调用 `read_bitmap` 函数检查磁盘块上的位图是否正确。执行完文件系统的初始化后，调用 `serve` 函数，文件系统服务开始运行，等待其他程序的请求。

图 5.9 以 UML 时序图的形式在宏观层面上展示了一个用户进程请求文件系统服务的过程（以 `open` 为例）。其中 `user_env` 所加载的程序不仅可以是实验源码已给出的 `fstest.c`，也可以是其他以 `main` 为入口函数的用户程序，你可以通过这种方式对文件系统服务进行测试。其中 IPC 系统调用的细节请参考 Lab4 的相关内容。（三种颜色不仅区分三个不同的程序，也表示进程执行的代码在我们的操作系统被载入内存前所处的文件位置）

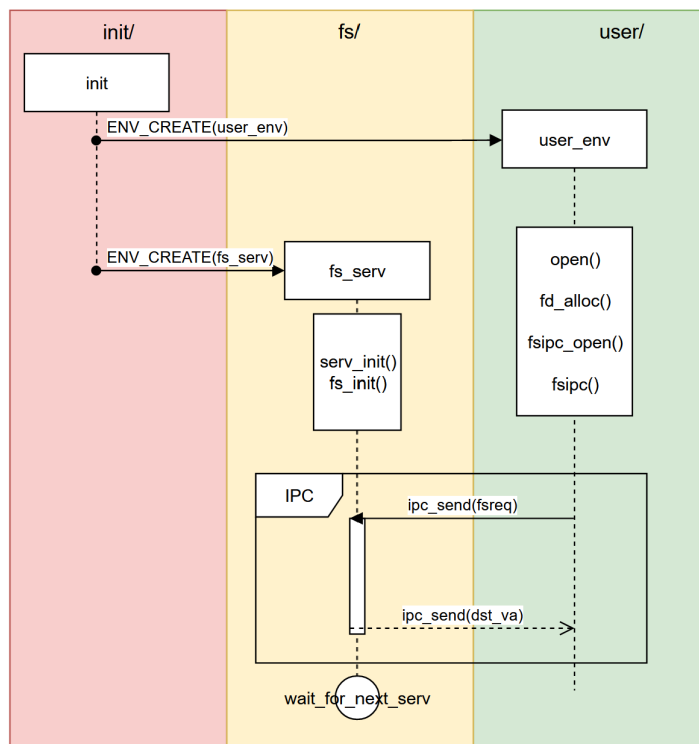


图 5.9: 文件系统服务时序图

Thinking 5.7 图 5.9 中有多种不同形式的箭头，请解释这些不同箭头的差别，并思考我们的操作系统是如何实现对应类型的进程间通信的。

文件系统支持的请求类型定义在 `user/include/fsreq.h` 中，包含以下几种：

```

1  enum {
2      FSREQ_OPEN,
3      FSREQ_MAP,
4      FSREQ_SET_SIZE,
5      FSREQ_CLOSE,
6      FSREQ_DIRTY,
7      FSREQ_REMOVE,
8      FSREQ_SYNC,
9      MAX_FSREQNO,
10 };

```

用户程序在发出文件系统操作请求时，将请求的内容放在对应的结构体中进行消息的传递，`fs_serv` 进程收到其他进行的 IPC 请求后，IPC 传递的消息包含了请求的类型和其他必要的参数，根据请求的类型执行相应的文件操作（文件的增、删、改、查等），将结果重新通过 IPC 反馈给用户程序。

文件 `user/lib/fsipc.c` 中定义了请求文件系统时用到的 IPC 操作，`user/lib/file.c` 文件中定义了用户程序读写、创建、删除和修改文件的接口。完成以下练习，实现删除指定路径的文件的功能。

Exercise 5.11 完成 `fs/serv.c` 中的 `serve_remove` 函数。

Exercise 5.12 完成 `user/lib/fsipc.c` 中的 `fsipc_remove` 函数。

Exercise 5.13 完成 `user/lib/file.c` 中的 `remove` 函数。

这里我们给出了文件系统的用户接口中部分函数可能的调用参考（图 5.10），希望同学们体会函数之间的调用关系、理解文件系统中用户接口的实现过程。

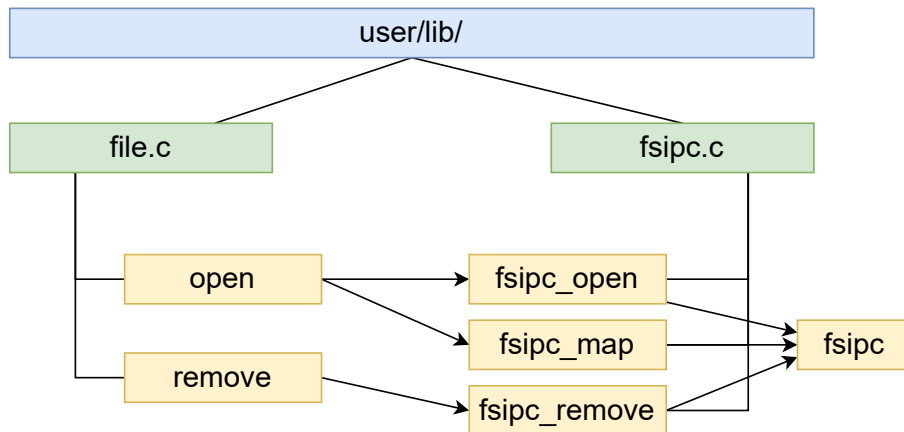


图 5.10: `user/lib` 下部分函数调用关系参考

5.6 正确结果展示

Note 5.6.1 使用 `make run` 运行。

本实验有三个阶段性测试可以检验你实现的文件系统的正确性。

5.6.1 IDE 磁盘交互

完成了 Exercise 5.1-5.2 后, 在 `init/init.c` 中仅启动 `devtst` 进程, 将对 Console 和 IDE 磁盘交互进行测试。

```
1 ENV_CREATE(user_devtst);
```

也可以使用 `make test lab=5_1` 直接构建。`make run` 后, 输入一个字符串 `01234567890123` 并按下回车, 正确结果如下:

```
1 devtst begin
2 01234567890123
3 syscall_read_dev is good
4 end of devtst
5 dev address is ok
```

5.6.2 文件系统测试

完成了 Exercise 5.3-5.7 后, 仅启动文件系统服务进程, 该进程在正式开始服务前, 会执行 `fs/check.c` 中的单元测试。

```
1 ENV_CREATE(fs_serv);
```

```
1 FS is running
2 superblock is good
3 read_bitmap is good
```

5.6.3 文件系统服务测试

完成全部 Exercise 后, 启动一个 `fstest` 进程和文件系统服务进程:

```
1 ENV_CREATE(user_fstest);
2 ENV_CREATE(fs_serv);
```

就能开始对文件系统的检测, 运行文件系统服务, 等待应用程序的请求。注意: 我们此时必须将文件系统进程作为第二个进程启动, 其原因是我们在 `user/lib/fsipc.c` 中定义的文件系统 IPC 请求的目标指定为第二个创建的进程。

```
1 FS is running
2 superblock is good
3 read_bitmap is good
4 serve_open 00000800 ffff000 0x2
5 open is good
6 read is good
7 serve_open 00000800 ffff000 0x0
8 open again: OK
9 read again: OK
10 file rewrite is good
11 serve_open 00000800 ffff000 0x0
12 file remove: OK
```

5.7 任务列表

- 完成 `sys_write_dev` 和 `sys_read_dev`
- 完成 `syscall_write_dev` 和 `syscall_read_dev`
- 完成 `fs/ide.c`
- 完成 `free_block`
- 完成 `create_file`
- 完成 `disk_addr`
- 完成 `map_block` 和 `unmap_block`
- 完成 `dir_lookup`
- 完成 `open`
- 完成 `read`
- 完成 `serve_remove`
- 完成 `fsipc_remove`
- 完成 `remove`

5.8 实验思考

- 设备操作与高速缓存
- 单个文件的最大体积、一个磁盘块最多存储的文件控制块及一个目录最多子文件
- 磁盘最大容量
- 文件系统进程中宏定义理解
- 文件描述符与 `fork` 函数
- 文件系统用户接口中的结构体
- 解释时序图，思考进程间通信

6.1 实验目的

1. 掌握管道的原理与底层细节
2. 实现管道的读写
3. 复述管道竞争情景
4. 实现基本 shell
5. 实现 shell 中涉及管道的部分

6.2 管道

在 Lab4 中，我们已经学习过一种进程间通信 (IPC, Inter-Process Communication) 的方式——共享内存。而今天我们要学的管道，其实也是进程间通信的一种方式。

6.2.1 初窥管道

管道是一种典型的进程间单向通信的方式。管道分有名管道和匿名管道两种，匿名管道只能在具有公共祖先的进程之间使用，且通常使用在父子进程之间。在 MOS 中，我们要实现匿名管道。本章后续所称的管道，没有特别说明，均指匿名管道。

在 Unix 中，管道由 `int pipe(int fd[2])` 函数创建，成功创建管道返回 0，参数中的 `fd` 用来保存读写端的文件描述符，`fd[0]` 对应读端，`fd[1]` 对应写端。

为了更好地理解管道实现的原理，我们先来做一个小实验体会一下¹

test_pipe.c

```
1 #include <stdlib.h>
2 #include <unistd.h>
3
4 int fildes[2];
```

¹实验代码参考 <http://pubs.opengroup.org/onlinepubs/9699919799/functions/pipe.html>

```

5  char buf[100];
6  int status;
7
8  int main(){
9
10     status = pipe(fildes);
11
12     if (status == -1 ) {
13         printf("error\n");
14     }
15
16
17     switch (fork()) {
18     case -1:
19         break;
20
21
22     case 0: /* 子进程 - 作为管道的读者 */
23         close(fildes[1]); /* 关闭不用的写端 */
24         read(fildes[0], buf, 100); /* 从管道中读数据 */
25         printf("child-process read:%s",buf); /* 打印读到的数据 */
26         close(fildes[0]); /* 读取结束, 关闭读端 */
27         exit(EXIT_SUCCESS);
28
29
30     default: /* 父进程 - 作为管道的写者 */
31         close(fildes[0]); /* 关闭不用的读端 */
32         write(fildes[1], "Hello world\n", 12); /* 向管道中写数据 */
33         close(fildes[1]); /* 写入结束, 关闭写端 */
34         exit(EXIT_SUCCESS);
35     }
36 }

```

你可以将示例代码复制到 Linux 环境, 使用命令 `gcc test_pipe.c && ./a.out` 来执行。

示例代码实现了从父进程向管道中写入消息 `Hello,world`, 子进程从管道中读出数据并打印到屏幕上。它演示了管道在父子进程之间通信的基本用法: 父进程在 `pipe` 函数之后, 调用 `fork` 来产生一个子进程, 之后在父子进程中各自执行不同的操作: 关掉自己不会用到的管道端, 然后进行相应的读写操作。在示例代码中, 父进程操作写端, 而子进程操作读端。

从本质上说, 管道是一种只存在于内存中的文件。在 UNIX 以及 MOS 中, 父进程调用 `pipe` 函数时, 会打开两个新的文件描述符: 一个表示只读端, 另一个表示只写端, 两个描述符都映射到了同一片内存区域。在 `fork` 的配合下, 子进程复制父进程的两个文件描述符, 从而在父子进程间形成了四个 (父子各拥有一读一写) 指向同一片内存区域的文件描述符, 父子进程可根据需要关掉自己不用的一个, 从而实现父子进程间的单向通信管道, 这也是匿名管道只能用在具有亲缘关系的进程间通信的原因。

Thinking 6.1 示例代码中, 父进程操作管道的写端, 子进程操作管道的读端。如果现在想让父进程作为“读者”, 代码应当如何修改? ■

6.2.2 mos 中 pipe 的使用与实现

在 MOS 中, 管道的使用和 UNIX 系统的示例代码逻辑相同。MOS 中使用管道的示例代码为 `user/testpipe.c`。

示例代码中先使用函数 `pipe(int p[2])` 创建了管道, 读端的文件描述符 (fd) 编号为 `p[0]`, 写端的文件描述符编号为 `p[1]`。之后使用 `fork()` 创建子进程, 注意这时父子进程使用自己的

p[0] 和 p[1] 访问到的内存区域是一致的。之后子进程关闭了自己的 p[1], 从 p[0] 读; 父进程关闭了自己的 p[0], 从 p[1] 写入管道。

在管道读写的过程中, 我们需要保证写端对管道的写入对读端可见, 也就是保证父子进程通过管道访问的内存相同。下面我们通过对 user/lib/pipe.c 中的 pipe 函数的具体实现来介绍如何实现这一需求。

```

1  int
2  pipe(int pfd[2])
3  {
4      int r, va;
5      struct Fd *fd0, *fd1;
6
7      // allocate the file descriptor table entries
8      if ((r = fd_alloc(&fd0)) < 0 ||
9          (r = syscall_mem_alloc(0, fd0, PTE_D | PTE_LIBRARY)) < 0) {
10         goto err;
11     }
12
13     if ((r = fd_alloc(&fd1)) < 0 ||
14         (r = syscall_mem_alloc(0, fd1, PTE_D | PTE_LIBRARY)) < 0) {
15         goto err1;
16     }
17
18     // allocate the pipe structure as first data page in both
19     va = fd2data(fd0);
20     if ((r = syscall_mem_alloc(0, (void *)va, PTE_D | PTE_LIBRARY)) < 0) {
21         goto err2;
22     }
23     if ((r = syscall_mem_map(0, (void *)va, 0,
24         (void *)fd2data(fd1), PTE_D | PTE_LIBRARY)) <
25         0) {
26         goto err3;
27     }
28
29     ...
30 }

```

在 pipe 中, 首先分配两个文件描述符 fd0 和 fd1 并为其分配空间, 然后给 fd0 对应的虚拟地址分配一页物理内存, 再将 fd1 对应的虚拟地址映射到这一页物理内存。

我们曾在 Lab4 填写 duppage 函数时介绍过这类页面, 即“共享页面”。共享页面是具有权限位 PTE_LIBRARY 的页面, 需要保持共享可写的状态, 使得父子进程对其进行修改的结果相互可见。当父子进程试图写共享页面时, 直接在该页面上进行写操作即可。

仔细观察 pipe 中出现的权限位 PTE_LIBRARY, 根据上述提示检查你在 Lab4 实现的 user/lib/fork.c 中的 duppage 函数, 保证它能根据父页面的不同权限位, 为子页面设置对应的权限位, 以此区分实现 Lab4 所需的写时复制机制或者 Lab6 中管道所需的共享页面机制。

下面我们使用图 6.1 来表示父子进程与管道的数据缓冲区的关系:

实际上, 在父子进程中各自 close 掉不再使用的端口后, 父子进程与管道缓冲区的关系如图 6.2。

下面我们来讲一下 struct Pipe, 并开始着手填写操作管道端的函数。

6.2.3 管道的读写

我们可以在 user/lib/pipe.c 中找到 Pipe 结构体的定义, 它的定义如下:

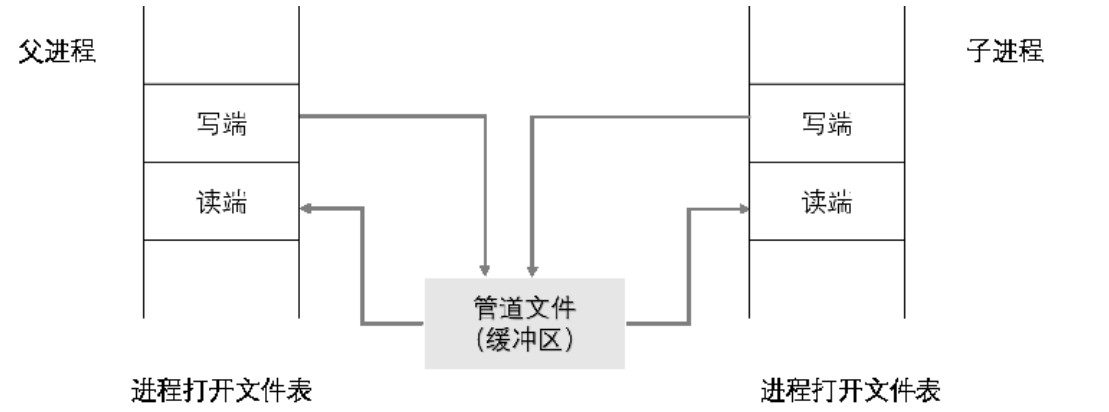


图 6.1: 父子进程与管道缓冲区

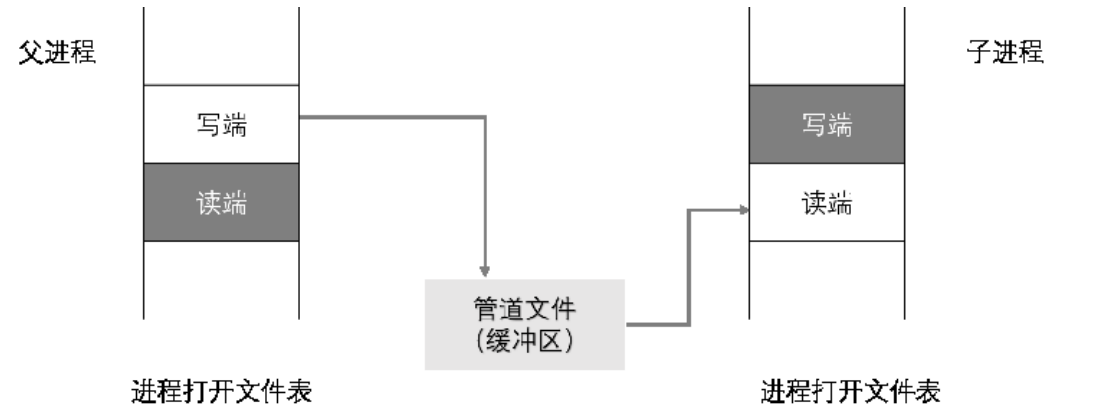


图 6.2: 关闭不使用的端口后

```

1 struct Pipe {
2     u_int p_rpos;           // read position
3     u_int p_wpos;           // write position
4     u_char p_buf[PIPE_SIZE]; // data buffer
5 };

```

在 Pipe 结构体中, `p_rpos` 给出了下一个将从管道读的数据的位置, 而 `p_wpos` 给出了下一个将要向管道写的数据的位置。只有读者可以更新 `p_rpos`, 同样, 只有写者可以更新 `p_wpos`, 读者和写者通过这两个变量的值进行读写的协调。

一个管道有 `PIPE_SIZE`(32 Byte) 大小的缓冲区。这个 `PIPE_SIZE` 大小的缓冲区发挥的作用类似于环形缓冲区, 所以下一个要读或写的位置 `i` 实际上是 `i%PIPE_SIZE`。

读者在从管道读取数据时, 要将 `p_buf[p_rpos%PIPE_SIZE]` 中的数据拷贝走, 然后读指针自增 1。但是需要注意的是, 管道的缓冲区此时可能还没有被写入数据。所以如果管道数据为空, 即当 `p_rpos >= p_wpos` 时, 应该进程切换到写者运行。

类似于读者, 写者在向管道写入数据时, 也是将数据存入 `p_buf[p_wpos%PIPE_SIZE]`, 然后写指针自增 1。需要注意管道的缓冲区可能出现满溢的情况, 所以写者必须得在 `p_wpos - p_rpos < PIPE_SIZE` 时方可运行, 否则要一直挂起。

上面这些还不能保证读者写者一定能顺利完成管道操作。假设这样的情景: 管道写端已经全部关闭, 读者读到缓冲区有效数据的末尾, 此时有 `p_rpos = p_wpos`。按照上面的做法, 我们这里应当切换到写者运行。但写者进程已经结束, 进程切换就造成了死循环, 这时候读者进程如何知道应当退出了呢?

为了解决上面提出的问题, 我们必须知道管道的另一端是否已经关闭。不论是读者还是写者进程, 我们都需要对管道另一端的状态进行判断: 当出现缓冲区空或满的情况时, 要根据另一端是否关闭来判断是否要返回。如果另一端已经关闭, 进程返回 0 即可; 如果没有关闭, 则切换进程运行。

Note 6.2.1 管道的关闭涉及到以下几个函数: `fd.c` 中的 `close`, `fd_close` 以及 `pipe.c` 中的 `pipe_close`。

Note 6.2.2 如果管道的写端相关的所有的文件描述符都已经关闭, 那么管道读端将会读到文件结尾并返回 0。

在 MOS 中, 我们使用 `static int _pipe_is_closed(struct Fd *fd, struct Pipe *p)` 函数来判断管道的另一端是否已经关闭。这个函数的核心, 就是下面要讲的恒成立等式。

在之前的图 6.2 中我们没有明确画出文件描述符所占的页, 但实际上, 对于每一个匿名管道而言, 我们分配了三页空间: 一页是读数据的文件描述符 `rfd`, 一页是写数据的文件描述符 `wfd`, 剩下一页是被两个文件描述符共享的管道数据缓冲区 `pipe`。既然管道数据缓冲区是被两个文件描述符所共享的, 我们很直观地就能得到一个结论: 如果有 1 个读者, 1 个写者, 那么管道将被引用 2 次 (正如图 6.2 所示)。 `pageref` 函数能得到页的引用次数, 所以有下面这个等式成立:

$$\text{pageref}(\text{rfd}) + \text{pageref}(\text{wfd}) = \text{pageref}(\text{pipe})$$

Note 6.2.3 内核会对 `pages` 数组成员维护一个页引用变量 `pp_ref` 来记录指向该物理页的虚页数量。 `pageref` 实际上就是查询虚拟地址对应的实际物理页, 然后返回其 `pp_ref` 变量的值。

我们将借助这个恒等式判断管道另一端是否已经关闭。假设现在读者进程正在运行, 而管道写者进程已经结束了, 那么此时就应该有: `pageref(wfd) = 0`。所以就有 `pageref(rfd) =`

`pageref(pipe)`。因此,只要判断这个等式是否成立就可以得知写端是否关闭。对写者来说同理。

Exercise 6.1 根据上述提示与代码中的注释,填写 `user/lib/pipe.c` 中的 `pipe_read`、`pipe_write`、`_pipe_is_closed` 函数并通过 `testpipe` 的测试。 ■

Note 6.2.4 注意在本次实验中由于文件系统服务所在进程已经默认为 1 号进程 (起始进程为 0 号进程),在测试时想启用文件系统需要注意 `ENV_CREATE(fs_serv)` 在 `init.c` 中的位置。

6.2.4 管道关闭的正确判断

MOS 操作系统采用的是时间片轮转调度的进程调度算法,有关这点,你应在 Lab3 中就深有体会了。这种抢占式的进程管理意味着,用户进程随时可能会被打断。

当然,如果进程间是孤立的,随时打断也没有关系。但当多个进程共享同一个变量时,不同的进程执行顺序有可能产生完全不同的结果,造成运行结果的不确定性。而进程通信需要共享同一块内存 (不论是管道还是共享内存),所以我们要对进程中共享变量的读写操作有足够高的警惕。

因此,因为管道本身的共享性质,在当前这种不加锁控制的情况下,无法保证 `_pipe_is_closed` 用于管道另一端关闭的判断一定返回正确的结果。

进程通过 `pipe_close` 函数来关闭管道的端口,该函数的实质是通过两次系统调用 `unmap` 解除文件描述符 `fd` 和数据缓存区 `pipe` 的映射。但是由于进程切换的存在,并不能保证两次系统调用可以在同一进程时间片内被执行,两次系统调用之间可能因为进程切换而被打断。所以,`fd` 和对 `pipe` 的 `pp_ref` 也不能保证同步被写入,这将影响我们判断管道是否关闭的正确性。

Note 6.2.5 在我们的 MOS 操作系统中,只有 `syscall_` 开头的系统调用函数是原子操作,其他所有包括 `fork` 这些函数在运行时都是可能会被随时打断

结合下文的代码,考虑以下场景:

```

1   pipe(p);
2   if(fork() == 0 ){
3       close(p[1]);
4       read(p[0],buf,sizeof buf);
5   }else{
6       close(p[0]);
7       write(p[1],"Hello",5);
8   }
```

- 假设 `fork` 结束后,子进程先执行。时钟中断产生在 `close(p[1])` 与 `read` 之间,父进程开始执行。
- 父进程在 `close(p[0])` 过程中,已经解除了 `p[0]` 对 `pipe` 的映射 (`unmap`),还没有来得及解除对 `p[0]` 的映射。假设这时时钟中断产生,进程调度后子进程接着执行。
- 注意此时各个页的引用情况: `pageref(p[0]) = 2` (因为父进程还没有解除对 `p[0]` 的映射),而 `pageref(p[1]) = 1` (因为子进程已经关闭了 `p[1]`)。但注意,此时 `pipe` 的 `pageref` 是 2,子进程中 `p[0]` 引用了 `pipe`,同时父进程中 `p[0]` 刚解除对 `pipe` 的映射,所以在父进程中也只有 `p[1]` 引用了 `pipe`。
- 子进程执行 `read`,`read` 中首先判断写者是否关闭。比较 `pageref(pipe)` 与 `pageref(p[0])` 之后发现它们都是 2,说明写端已经关闭,于是子进程退出。

Thinking 6.2 上面这种不同步修改 `pp_ref` 而导致的进程竞争问题在 `user/lib/fd.c` 中的 `dup` 函数中也存在。请结合代码模仿上述情景，分析一下我们的 `dup` 函数中为什么会出现预想之外的情况？

从上述场景中可以发现，由于时间中断发生在父进程 `unmap pipe` 和 `unmap fd` 之间，子进程根据 `pageref(pipe) == pageref(p[0])` 这一等式错误判断了写端已经关闭。同时，我们也可发现，父进程 `pipe_close` 的运行逻辑是首先解除 `pipe` 的映射，再解除 `fd` 的映射。但若我们在 `pipe_close` 函数中调整 `unmap` 的顺序，即首先解除 `fd` 的映射，再解除 `pipe` 的映射，可以避免上述场景的错误情况，接下来，我们将解释调整顺序后可行的原因。

Thinking 6.3 阅读上述材料并思考：为什么系统调用一定是原子操作呢？如果你觉得不是所有的系统调用都是原子操作，请给出反例。希望能结合相关代码进行分析说明。

`_pipe_is_closed` 函数返回正确结果的条件其实只是：

- 写端关闭当且仅当 `pageref(p[0]) == pageref(pipe)`;
- 读端关闭当且仅当 `pageref(p[1]) == pageref(pipe)`;

比如说第一个条件，写端关闭时，当然有 `pageref(p[0]) == pageref(pipe)`。但是由于进程切换的存在，我们无法确保当 `pageref(p[0]) == pageref(pipe)` 时，写端关闭。正面如果不好解决问题，我们可以考虑从其逆否命题着手，即我们要确保：当写端没有关闭的时候，`pageref(p[0]) != pageref(pipe)`。

我们考虑之前那个预想之外的情景，它出现的最关键原因在于：`pipe` 的引用次数总比 `fd` 要高。当管道的 `close` 进行到一半时，若先解除 `pipe` 的映射，再解除 `fd` 的映射，就会使得 `pipe` 引用次数的 -1 先于 `fd`。这就导致在两个 `unmap` 的间隙，会出现 `pageref(pipe) == pageref(fd)` 的情况。若调换 `fd` 和 `pipe` 在 `close` 中的 `unmap` 顺序，使得 `fd` 引用次数的 -1 先于 `pipe`。在两个 `unmap` 的间隙，`pageref(pipe) > pageref(fd)` 仍成立，即使此时发生中断，也不会影响判断管道是否关闭的正确性。

Thinking 6.4 仔细阅读上面这段话，并思考下列问题

- 按照上述说法控制 `pipe_close` 中 `fd` 和 `pipe unmap` 的顺序，是否可以解决上述场景的进程竞争问题？给出你的分析过程。
- 我们只分析了 `close` 时的情形，在 `fd.c` 中有一个 `dup` 函数，用于复制文件描述符。试想，如果要复制的文件描述符指向一个管道，那么是否会出现与 `close` 类似的问题？请模仿上述材料写写你的理解。

根据上面的描述我们其实已经能够得出一个结论：控制 `fd` 与 `pipe` 的 `map/unmap` 的顺序可以解决进程竞争导致非同步写入的问题。

那么下面根据你所思考的内容进行实践吧：

Exercise 6.2 修改 `user/lib/pipe.c` 中 `pipe_close` 函数中的 `unmap` 顺序以避免上述情景中的进程竞争情况。

提示：当前函数中的 `unmap` 操作会使 `pipe/fd` 的引用次数 -1。

Exercise 6.3 修改 `user/lib/fd.c` 中 `dup` 函数中的 `map` 顺序以避免上述情景中的进程竞争情况。

提示：当前函数中的 `map` 操作会使 `fd/pipe` 的引用次数 +1。

我们通过控制修改 `pp_ref` 的前后顺序避免了非同步写入的问题，但是我们还解决第二个问题：读取 `pp_ref` 的同步问题。

同样是上面 6.2.4 节的代码，我们思考下面的情景：

- 假设 `fork` 结束后，子进程先执行。执行完 `close(p[1])` 后，执行 `read`，要从 `p[0]` 读取数据。但由于此时管道数据缓冲区为空，所以 `read` 函数要判断父进程中的写端是否关闭，进入到 `_pipe_is_closed` 函数，`pageref(fd)` 值为 2(父进程和子进程都打开了 `p[0]`)，此时时钟中断产生。
- 内核切换到父进程执行，父进程 `close(p[0])`，之后向管道缓冲区写数据。要写的数据较多，假设写到一半时钟中断产生，进程调度后切换到子进程运行。
- 子进程继续运行，获取到 `pageref(pipe)` 值为 2(父进程打开了 `p[1]`，子进程打开了 `p[0]`)，引用值相等，于是认为父进程的写端已经关闭，子进程退出。

`fd` 是一个父子进程共享的变量，但子进程中的 `pageref(fd)` 没有随父进程对 `fd` 的修改而同步，这就造成了子进程读到的 `pageref(fd)` 成为了“脏数据”。为了保证读的同步性，子进程应当重新读取 `pageref(fd)` 和 `pageref(pipe)`，并且要在确认两次读取之间进程没有切换后，才能返回正确的结果。为了实现这一点，我们要使用到之前一直都没用到的变量：`env_runs`。

`env_runs` 记录了一个进程 `env_run` 的次数，这样我们就可以根据某个操作 `do()` 前后进程 `env_runs` 值是否相等，来判断在 `do()` 中进程是否发生了切换。

根据上面的表述，修改 `_pipe_is_closed` 函数，使得它满足“同步读”的要求。注意 `env_runs` 变量是需要维护的。

6.2.5 相关函数

下面我们介绍 6 个与管道相关的函数。

创建管道函数

`user/lib/pipe.c` 中的 `int pipe(int pfd[2])`。

这个函数的作用是创建一个管道。大致可以分为三步：

- 首先，创建并分配两个文件描述符 `fd0`、`fd1`，并为这两个文件描述符自身分配相应的空间。
- 然后给 `fd0` 对应的数据区域分配一页空间，并将 `fd1` 对应的数据区域映射到相同的物理页，这一页的内容为 `pipe` 结构体。
- 最后，将作为读端的 `fd0` 的权限设置为只读，作为写端的 `fd1` 的权限设置为只写，并通过函数的传入参数将这两个文件描述符的编号返回。

值得注意的是，目前我们一共分配了三个页面，权限位均需要包含 `PTE_LIBRARY`，这是因为这些页面的数据对于父子进程是共享的，修改页面内容时不触发写时复制，这样才能顺利的在父子进程之间传递信息。

两个查询管道是否关闭的函数

user/lib/pipe.c 中的 `static int _pipe_is_closed(struct Fd *fd, struct Pipe *p)` 与 `int pipe_is_closed(int fdnum)`。

这两个函数的作用是判断管道是否已经被关闭,其中,函数的主要逻辑存在于 `_pipe_is_closed` 中, `pipe_is_closed` 是对于 `_pipe_is_closed` 的重新封装。

通过创建管道时,分配的页面以及映射关系可知,文件描述符所在页面 `rfd`、`wfd` 被读端写端各自映射一次,管道页面 `pipe` 被读端写端同时映射,被映射两次,因此则有如下等式

$$\text{pageref}(\text{rfd}) + \text{pageref}(\text{wfd}) = \text{pageref}(\text{pipe})$$

当管道某一端被关闭后,这一端映射的页面将被解除,假设当前调用此函数的进程为读进程,写端关闭,则 `pageref(wfd)=0`,当前进程为写进程时类似,因此,当管道另一端关闭时,以下等式成立:

$$\text{pageref}(\text{fd}) = \text{pageref}(\text{pipe})$$

需要注意的是,在 `_pipe_is_closed` 中,我们需要利用两次 `pageref` 函数分别获取 `fd` 和 `pipe` 对应页面的引用数,由于这两次操作不是原子的,之间可能被时间片调度打断,导致 `pageref(fd)` 和 `pageref(pipe)` 非同步获取,先获取的数据失效。为了解决这个问题,我们需要确保两次获取 `pageref` 时进程没有切换,也就是两次获取 `pageref` 前后的 `env->env_run` 变量没有变化时,再根据等式判断管道另一端是否被关闭。若等式成立,则说明管道另一端已经关闭,返回 1,否则返回 0。

读管道函数

user/lib/pipe.c 中的 `static int pipe_read(struct Fd *fd, void *vbuf, u_int n, u_int offset)`。

这个函数的作用是从 `fd` 对应的管道数据缓冲区中,读取至多 `n` 字节到 `vbuf` 对应的虚拟地址中,并返回本次读到的字节数。

读取的过程中会遇到两类可能的情况:

- 缓冲区不为空: 则按顺序读取缓冲区中的内容,直到缓冲区为空或达到读取上限,并返回读到的字节数。
- 缓冲区为空: 则使用 `_pipe_is_closed` 函数查询管道的写端是否已经关闭,若已经关闭,则说明读入完成,函数返回 0; 若没有关闭,则使用 `syscall_yield()` 进行等待,直到管道关闭或缓冲区不为空。

写管道函数

user/lib/pipe.c 中的 `static int pipe_write(struct Fd *fd, const void *vbuf, u_int n, u_int offset)`。

与读管道函数类似,这个函数的作用是从 `vbuf` 对应的虚拟地址,向 `fd` 对应的管道数据缓冲区中写入 `n` 字节,并返回本次写入的字节数。

此处的参数 `n` 与读取时有所不同,表示需要写入恰好 `n` 个字节,而非缓冲区的长度上限。因此,在完成全部 `n` 个字节的写入之前,若管道缓冲区已满,则需要使用 `syscall_yield()` 进行等待,而非提前返回,直到缓冲区不满或管道关闭。

关闭管道函数

user/lib/pipe.c 中的 `static int pipe_close(struct Fd *fd)`。

这个函数的作用是关闭管道的一端。将待关闭的一端对应的文件描述符传入，并 `unmap` 文件描述符自身的页面以及 `unmap` 文件描述符对应的数据页面——`pipe` 结构体，完成关闭操作。

6.3 shell

在计算机科学中，shell 是指“为用户提供操作界面”的软件（命令解析器）。它接收用户命令，然后调用相应的应用程序。基本上 shell 分两大类：

一是图形界面 shell（Graphical User Interface shell 即 GUI shell）。例如：应用最为广泛的是微软 Windows 系列操作系统的 Windows Explorer，也包括广为人知的 Linux 操作系统的 XWindow Manager（BlackBox 和 FluxBox），以及功能更强大的 CDE、GNOME、KDE 和 XFCE 等。

二是命令行式 shell（Command Line Interface shell，即 CLI shell），也就是我们 MOS 操作系统最后即将实现的 shell 模式。

6.3.1 完善 spawn 函数

`spawn` 的作用是帮助我们调用文件系统中的可执行文件并执行。

`spawn` 的流程可以分解如下：

- 从文件系统打开对应的文件（二进制 ELF，在我们的 OS 里是 *.b）；
- 申请新的进程控制块；
- 将目标程序加载到子进程的地址空间中，并为它们分配物理页面；
- 为子进程初始化地址空间。对于栈空间，由于 `spawn` 需要将命令行参数传递给用户程序，所以要将参数也写入用户栈中；
- 设置子进程的寄存器（栈指针 `sp` 和用户程序入口 `EPC`）；
- 将父进程的共享页面映射到子进程的地址空间中；
- 这些都做完后，设置子进程可执行。

Thinking 6.5 思考以下三个问题。

- 认真回看 Lab5 文件系统相关代码，弄清打开文件的过程。
- 回顾 Lab1 与 Lab3，思考如何读取并加载 ELF 文件。
- 在 Lab1 中我们介绍了 `data text bss` 段及它们的含义，`data` 段存放初始化过的全局变量，`bss` 段存放未初始化的全局变量。关于 `memsize` 和 `filesize`，我们在 Note 1.3.4 中也解释了它们的含义与特点。关于 Note 1.3.4，注意其中关于“`bss` 段并不在文件中占数据”表述的含义。回顾 Lab3 并思考：`elf_load_seg()` 和 `load_icode_mapper()` 函数是如何确保加载 ELF 文件时，`bss` 段数据被正确加载进虚拟内存空间。`bss` 段在 ELF 中并不占空间，但 ELF 加载进内存后，`bss` 段的数据占据了空间，并且初始值都是 0。请回顾 `elf_load_seg()` 和 `load_icode_mapper()` 的实现，思考这一点是如何实现的？

下面给出一些对于上述问题的提示，以便大家更好地把握加载内核进程和加载用户进程的区别与联系，类比完成 `spawn` 函数。

关于第一个问题，在 Lab3 中我们创建进程，并且通过 `ENV_CREATE(...)` 在内核态加载了初始进程，而我们的 `spawn` 函数则是通过和文件系统交互，取得文件描述块，进而找到 ELF 在“硬盘”中的位置，进而读取。

关于第二个问题，各位已经在 Lab3 中填写了 `load_icode` 函数，实现了 ELF 可执行文件中读取数据并加载到内存空间，其中通过调用 `elf_load_seg` 函数来加载各个程序段。在 Lab3 中我们要填写 `load_icode_mapper` 回调函数，在内核态下加载 ELF 数据到内存空间；相应地，在 Lab6 中 `spawn` 函数也需要在用户态下使用系统调用为 ELF 数据分配空间。

附录中有一个实例，可以帮助你再次吸收理解 6.3.1 Lab1、Lab3 联动的内容，探讨 `bss` 段在虚拟内存和磁盘 ELF 文件中是否占据空间这一问题。

关于如何为子进程初始化栈空间，请仔细阅读 `init_stack` 函数。

因为我们无法直接操作子进程的栈空间，所以该函数首先将需要准备的参数填充到本进程的 `UTEMP` 这个页面处，然后将 `UTEMP` 映射到子进程的栈空间中。

首先将 `argc` 个字符串填到栈上，并且不要忘记在每个字符串的末尾要加上 `'\0'` 表示结束，然后将 `argc+1` 个指针填到栈上，第 `argc+1` 个指针指的是一个空字符串表示参数的结束。

最后将 `argc` 和 `argv` 填到栈上，`argv` 将指向那 `argc+1` 个字符指针。

这里给出一张 `spawn` 准备的栈空间的示意图。

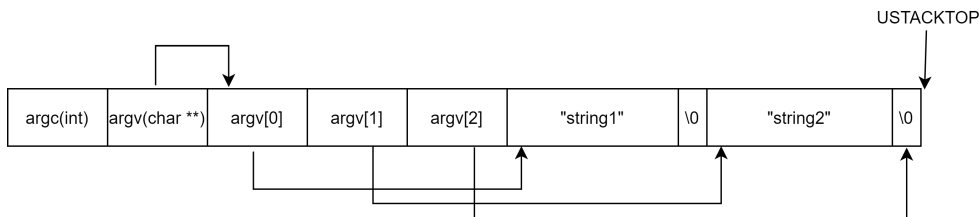


图 6.3: 子进程栈空间示意图

Exercise 6.4 根据以上描述以及注释，补充完成 `user/lib/spawn.c` 中的 `int spawn(char *prog, char **argv)`。

6.3.2 解释 shell 命令

在 Lab5 中我们实现了文件系统，Lab6 中我们在 `user` 目录下提供了 `ls.c`、`cat.c`、`echo.c` 等几个用户程序，模拟了 Linux 下的同名命令，而 Shell 程序 `sh.c` 调用了上面提到的 `spawn` 函数，其能够读取相应的可执行文件，并加载到新进程中运行。请阅读 `fs/Makefile` 与 Exercise 5.5，思考我们是如何将这些用户程序编译得到的可执行文件“烧录”到 MOS 的文件系统中的。

接下来，我们需要在 shell 进程里实现对管道和重定向的解释功能。解释 shell 命令时：

1. 如果碰到重定向符号 `'<'` 或者 `'>'`，则读下一个单词，打开这个单词所代表的文件，然后将其复制给标准输入或者标准输出。
2. 如果碰到管道符号 `'|'`，则首先需要建立管道 `pipe`，然后 `fork`。
 - 对于父进程，需要将管道的写者复制给标准输出，然后关闭父进程的读者和写者，运行 `'|'` 左边的命令，获得输出，然后等待子进程运行。
 - 对于子进程，将管道的读者复制给标准输入，从管道中读取数据，然后关闭子进程的读者和写者，继续读下一个单词。

在这里可以举一个使用管道符号的例子来方便大家理解，在 Lab0 中，我们介绍了 Linux 中的管道命令 `cat my.sh | grep "Hello"` 这就是使用管道的例子，`cat my.sh` 命令会将 `my.sh` 中的内容原文写入到管道，而 `grep "Hello"` 命令作为子进程执行，将管道中的内容读出并作为 `grep` 的输入，进行查找字符串操作并将查找结果输出到标准输出。

Exercise 6.5 根据以上描述，补充完成 `user/sh.c` 中的 `void parsecmd(char **argv, int *rightpipe)`，实现对重定向和管道的解释功能。

提示：注意考虑重定向失败的情况（当文件不能正确打开时，重定向符号前的命令不应执行）。

在 `spawn` 函数中通过设置 `PTE_LIBRARY` 权限位，将父进程所有的共享页面映射给了子进程。想一下，进程空间中的哪些内存是共享内存？

在进程空间中，文件、管道、控制台以及文件描述符都是以共享页面的方式存在的。有几处通过 `spawn` 产生新进程的位置。

- 如图 6.4，内核启动的进程 `user/icode.b` 调用了 `spawn`，创建了 `init.b` 进程。`init.b` 进程先打开控制台（console）作为 0 和 1 号文件描述符（fd），也就是进程的标准输入和输出，然后调用 `spawn` 创建了 `sh.b` 进程，也就是我们的 shell。通过共享页面（`PTE_LIBRARY`）机制，`fork` 和 `spawn` 创建的子进程继承了父进程持有的 fd，所以 shell 进程仍能通过标准输入输出操作控制台，与用户进行交互。

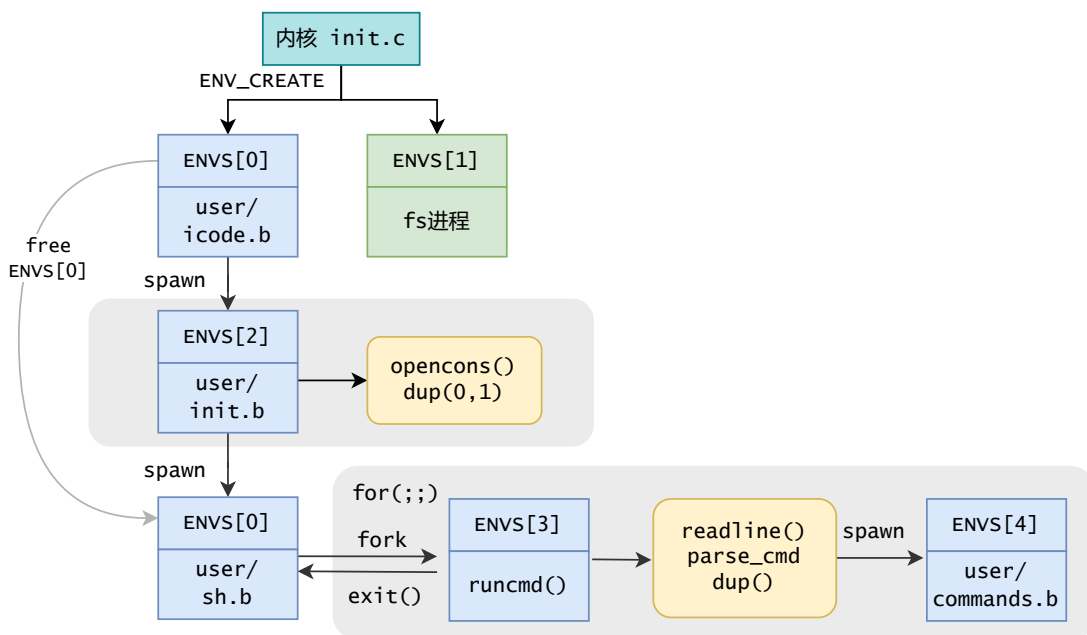


图 6.4: shell 启动执行过程

Thinking 6.6 通过阅读代码空白段的注释我们知道，将标准输入或输出定向到文件，需要将其 `dup` 到 0 或 1 号文件描述符（fd）。那么问题来了：在哪步，0 和 1 被“安排”为标准输入和标准输出？请分析代码执行流程，给出答案。

- 子 shell 进程负责解析命令行命令，并通过 `spawn` 生成可执行程序进程（对应 `*.b` 文件）。在解析命令行的命令时，子 shell 会将重定向的文件及管道等 `dup` 到子 shell 的标准输入

或输出，然后 `spawn` 时将标准输入和输出通过共享内存映射给可执行程序，所以可执行程序可以从控制台、文件和管道等位置输入和输出数据。

Thinking 6.7 在 shell 中执行的命令分为内置命令和外部命令。在执行内置命令时 shell 不需要 `fork` 一个子 shell，如 Linux 系统中的 `cd` 命令。在执行外部命令时 shell 需要 `fork` 一个子 shell，然后子 shell 去执行这条命令。

据此判断，在 MOS 中我们用到的 shell 命令是内置命令还是外部命令？请思考为什么 Linux 的 `cd` 命令是内部命令而不是外部命令？ ■

Thinking 6.8 在你的 shell 中输入命令 `ls.b | cat.b > motd`。

- 请问你可以在你的 shell 中观察到几次 `spawn`？分别对应哪个进程？
- 请问你可以在你的 shell 中观察到几次进程销毁？分别对应哪个进程？

6.3.3 相关函数

下面我们介绍 8 个与 `shell` 相关的函数。

初始化栈空间函数

`user/lib/spawn.c` 中的 `int init_stack(u_int child, char **argv, u_int *init_esp)`。

这个函数的作用是初始化子进程的栈空间，达到向子进程的主函数传递参数的目的。由于父进程无法直接操作子进程的栈空间，因此需要将参数填充到当前进程的 `TMPPAGE` 页面处，将 `TMPPAGE` 映射到子进程的栈空间中。具体参数填充方式可以参考图 6.3。

`spawn` 函数

`user/lib/spawn.c` 中的 `int spawn(char *prog, char **argv)`。

这个函数与 `fork` 函数类似，其最终效果都是产生一个子进程，不过与 `fork` 函数不同的是，`spawn` 函数产生的子进程不再执行与父进程相同的程序，而是装载新的 ELF 文件，执行新的程序。

`spawn` 函数的大致流程如下：

1. 使用文件系统提供的 `open` 函数打开即将装载的 ELF 文件 `prog`。
2. 使用系统调用 `syscall_exofork` 函数为子进程申请一个进程控制块。
3. 使用 `init_stack` 函数为子进程初始化栈空间，将需要传递的参数 `argv` 传入子进程。
4. 使用 `elf_load_seg` 将 ELF 文件的各个段加载进子进程。
5. 设置子进程的运行现场寄存器，将 `tf->cp0_epc` 设置为程序入口点，`tf->regs[29]` 设置为装载参数后的栈顶指针，从而在子进程被唤醒时以正确的状态开始运行。
6. 将父进程的共享页面映射给子进程，与 `fork` 不同的是，这里只映射共享页面。
7. 使用系统调用 `syscall_set_env_status` 唤醒子进程。

其中，第 1 步、第 3 步和第 4 步是相比 `fork` 函数新增的部分，第 5 步和第 6 步与 `fork` 略有不同，第 2 步和第 7 步与 `fork` 中的步骤几乎一致。

sh.c 中的主函数

user/sh.c 中的 `int main(int argc, char **argv)`。

与 `pipe.c` 这样的用户库不同, `sh.c` 是一个完整的用户程序, 也就是 `shell`, 其主函数即为启动 `shell` 进程时第一步进入的函数。函数的主体是一个死循环, 循环中大致流程如下:

- 调用 `readline` 读入用户输入的命令。
- `fork` 出一个子进程。
- 子进程执行用户的命令, 执行结束后子进程结束。父进程在此等待子进程。
- 父进程等待子进程结束后, 返回循环开始, 读入用户的下一个命令。

命令读入函数

user/sh.c 中的 `void readline(char *buf, u_int n)`。

这个函数的作用是从标准输入 (控制台), 读入一行用户读入的命令, 保存在 `char* buf` 中。

命令解析函数

user/sh.c 中的 `int _gettoken(char *s, char **p1, char **p2)` 和 `int gettoken(char *s, char **p1)`。

`gettoken` 函数接收 `readline` 读入的命令字符串作为传入参数 `char* s`。这个两个函数的作用是将命令字符串分割, 提取命令中基本单元——特殊符号或单词, 并过滤空白字符。其中, `gettoken` 是对于 `_gettoken` 的封装, 调用 `gettoken` 时, 每次调用返回上一次调用解析到的基本单元, 并解析下一个基本单元。

特殊符号包括重定向符号 `<` 和 `>`、管道符号 `|`、命令分隔符号; 和 `&`。

cmd 命令解析与执行函数

user/sh.c 中的 `void parsecmd(char **argv, int *rightpipe)` 和 user/sh.c 中的 `void runcmd(char *s)`。

`void parsecmd(char **argv, int *rightpipe)` 这个函数的作用是解析用户输入的命令。通过调用 `gettoken`, 将命令解析成单个词法单元, 保存在 `argv` 中, 并在遇到重定向、管道等特殊符号时做相应的特殊处理。当解析完一条命令包含的所有词法单元后, 结束 `parsecmd` 函数, 进入 `runcmd` 函数的命令执行阶段。特别地, 当命令中包含管道操作时, 则需要创建管道并 `fork` 出一个子 `shell`。子进程调用 `parsecmd` 函数继续解析管道右侧的命令, 父进程返回 `runcmd` 函数执行管道左侧命令, 此时 `rightpipe` 记录子进程的进程号也就是解析并执行管道右侧命令的进程号, 用于管道左侧命令进程等待管道右侧命令进程执行完毕后再释放。

`void runcmd(char *s)` 这个函数的作用是执行解析出的每一条命令。通过 `argv` 判断命令的种类, 调用 `spawn` 产生子进程并装载命令对应的 ELF 文件, 子进程执行命令, 父进程在此处等待子进程结束后, 结束进程。随后, 上文提到的在 `main` 循环结尾处等待的进程, 也就是此处父进程的父进程, 结束等待, 返回循环开始, 读入新一行命令。具体进程间的层次关系可以参照图6.5。

shell 相关函数调用关系

见图 6.5。

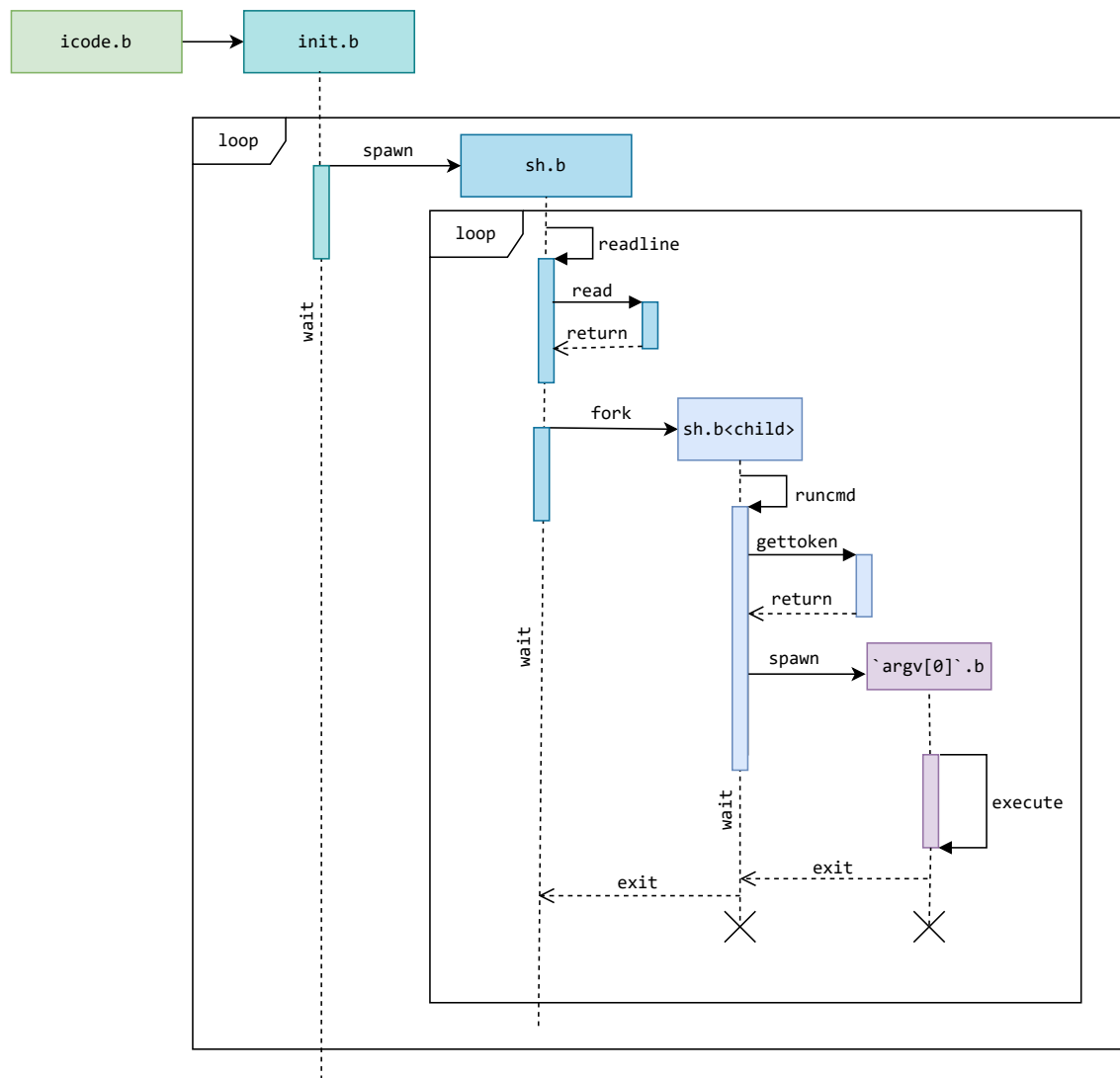


图 6.5: shell 相关函数调用关系

6.4 实验正确结果

本节将通过一些测试用例评测管道和 Shell 的正确性，同时给出了测试结果的截图，同学们可以自行核对。

6.4.1 管道测试

管道测试对应的测试点为 `tests/lab6_1`，测试中有三个文件，分别是 `user/testpipe.c`、`user/testpiperace.c` 和 `user/testtelibrary.c`，以合适的次序建好进程后，在 `testpipe` 的测试中若出现两次 `pipe tests passed` 即说明测试通过。

`testpipe` 本地测试部分运行结果如下。

```
[00001802] pipecreate
.....
pipe write closed properly
pipe tests passed
[00004005] destroying 00004005
[00004005] free env 00004005
i am killed ...
pipe tests passed
[00001802] destroying 00001802
[00001802] free env 00001802
i am killed ...
```

在 `testpiperace` 中，子进程多次查询管道是否关闭。而父进程不停地执行 `dup` 函数，`dup` 操作主要包含两个操作，分别是关闭映射目标原有的管道 (`close`) 和将旧文件映射到新文件 (包括文件描述符和文件内容)。如果这两个操作中有任意一个操作产生竞争，将可能导致子进程认为写入端关闭。

在 `testpiperace` 的测试中应当出现 `race didn't happen` 是正确的。`testpiperace` 本地测试部分运行结果如下。

```
race didn't happen
[00002003] destroying 00002003
[00002003] free env 00002003
i am killed ...
```

在 `user/testptelibrary.c` 的测试中，如果 `fork` 和 `spawn` 对于共享页面的处理正确，测试才能通过。

6.4.2 shell 测试

在 `init/init.c` 中按照如下顺序依次启动 shell 和文件服务：

```
1 ENV_CREATE(user_icode);
2 ENV_CREATE(fs_serv);
```

如果正常会看到如下效果。

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::                                                                ::
::                               MOS Shell 2023                    ::
::                                                                ::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

使用不同的命令会有不同的效果：

- 执行 `ls.b`, 会显示一些文件和目录名。

```
$ ls.b
testarg.b cat.b pingpong.b testbss.b newmotd testpiperace.b testpipe.b
motd init.b num.b testfdsharing.b ls.b echo.b sh.b halt.b testptelibrary.b
```

- 执行 `cat.b` 并输入任意字符, 会有回显现象出现, 按下 `Ctrl-D` 可结束输入。
- 执行 `cat.b motd`, 会显示文件 `motd` 中的内容。

```
$ cat.b
aabbcc
```

```
$ cat.b motd
This is /motd, the message of the day.

Welcome to the MOS kernel, now with a file system!
```

- 执行 `ls.b | cat.b`, 和 `ls.b` 的现象应当一致。

```
$ ls.b | cat.b
testarg.b cat.b pingpong.b testbss.b newmotd testpiperace.b testpipe.b
motd init.b num.b testfdsharing.b ls.b echo.b sh.b halt.b testptelibrary.b
```

6.5 任务列表

- 填写 `pipe_read`, `pipe_write`, `__pipe_is_closed` 函数
- 修改 `pipe_close` 函数
- 修改 `dup` 函数
- 完成 `spawn` 函数
- 完成 `parsecmd` 函数

6.6 实验思考

- 思考-父进程为读者
- 思考-`dup` 中的进程竞争
- 思考-原子操作
- 思考-解决进程竞争
- 思考-如何实现加载 `bss` 段
- 思考-内置命令与外部命令
- 思考-标准输入和标准输出
- 思考-解释命令执行的现象

附录 A

补充知识

在本章中，我们将介绍一些操作系统实验的补充知识，有助于我们理解并实现操作系统。

A.1 shell 编程易错点说明

语句与命令的区分

对于有良好的代码格式规范的同学，一般会习惯于在赋值语句的等号两边加上空格，而在 shell 编程中我们赋值语句的等号两边都不能存在空格，这可能会让部分同学感到困惑。这种看似与大多编程语言的规范都不同的使用方式其实是从 shell 自身的功能出发的所要求的。在 shell 出现之时，目的就是让用户能够通过命令与我们的操作系统进行交互，而 shell 中一条命令的基本格式为 `<command> <arg0> <arg1> ...`。因此如果我们写出 `a = 1` 这样的“赋值语句”，它将会被 shell 解析为调用一个名为 `a` 的命令，并给它传入参数 `=` 和 `1`，从而会得到报错 `command not found: a`（当我们的环境中不存在 `a` 这个可执行程序时）。因此在 shell 中一条赋值语句只能写作 `a=1` 这样的等号两边都不能有空格的形式。

命令中括号的用法

在 shell 中不同的括号用法可能会对初学者造成不小的困扰，因此下面对括号的基本用法进行简单的介绍。

- 美元符与单小括号、反引号

使用美元符加上单小括号用于隔离并执行命令，并将其输出替换至原处。例如下述 shell 程序中，使用 `$(diff file1 file2)` 得到 `diff` 比较的输出并替换至原处，从而变为 `"<diff output>"` 也即得到 `diff` 的输出结果并作为一个字符串，判断该字符串是否为空即可判断 `file1` 与 `file2` 是否相同。在该命令中的 `$(diff file1 file2)` 改为使用反引号 ``diff file1 file2`` 也有着相同的效果。

```
1  #!/bin/bash
2
3  if [ -z "$(diff file1 file2)" ]; then
4      echo "same"
5  else
6      echo "different"
```

7 | `fi`

- 双小括号

双小括号用于算术表达式（当然你也可以在其中进行赋值），例如：

```

1  #!/bin/bash
2
3  a=1
4  b=2
5  c=$((a + b))
6  echo $c          # 3
7
8  ((a = 10))
9  echo $a          # 10
10
11 ((b++))
12 echo $b          # 3
13
14 a=$((a + 2))
15 echo $a          # 12
16
17 echo $((a * b + c))      # 39

```

- 单中括号

单中括号与 `test` 相同，用于判断条件是否成立。大家通过 `ls /bin` 能发现其中存在一个名为 `[` 的可执行程序，事实上这个 `[` 就是我们写在判断条件中的 `[`。这也就是为什么在判断条件 `[<condition>]` 时一定需要有空格，例如条件 `[$a -ne 2]`，实际上是将 `a`、`-ne`、`2`、`]` 作为参数传给可执行程序 `[`，而不是表面上看起来像是用中括号包裹了判断语句 `$a -ne 2`。

- 双中括号

双中括号同样用于判断条件，但相比单中括号提供了更为丰富的一些功能。例如在双中括号中可以直接使用逻辑运算：`[[$a -gt 1 && $a -lt 100]]`，而使用单中括号只能写为 `[$a -gt 1] && [$a -lt 100]`

A.2 真实操作系统的内核及启动详解

操作系统最重要的部分是操作系统内核，因为内核需要直接与硬件交互来管理各个硬件，从而利用硬件的功能为用户进程提供服务。为了启动操作系统，就需要将内核程序在计算机上运行起来。一个程序要能够运行，其必须能够被 CPU 直接访问，所以不能放在磁盘上，因为 CPU 无法直接访问磁盘；另一方面，内存 RAM 是易失性存储器，掉电后将丢失全部数据，所以不可能将内核代码保存在内存中。所以直观上可以认识到：磁盘不能直接访问，并且内存掉电易失，因此，内核有可能放置的位置只能是 CPU 能够直接访问的非易失性存储器——ROM 或 FLASH 中。

但是，直接把操作系统内核放置在这样的非易失存储器上会有一些问题：

1. 这种 CPU 能直接访问的非易失性存储器的存储空间一般会映射到 CPU 可寻址空间的某个区域，这个是在硬件设计决定的。显然这个区域的大小是有限的，如果功能比较简单的操作系统还能够放在其中，对于较大的普通操作系统显然不够。
2. 如果操作系统内核在 CPU 加电后直接启动，意味着一个计算机上只能启动一个操作系统，这样的限制显然不是我们所希望的。

3. 把特定硬件相关的代码全部放在操作系统中也不利于操作系统的移植工作。

基于上述考虑,设计人员一般都会将硬件初始化的相关工作作为“bootloader”程序放在非易失存储器中,而将操作系统内核放在磁盘中。这样的做法可有效解决上述的问题:

1. 将硬件初始化的相关工作从操作系统中抽出放在 bootloader 中实现,意味着通过这种方式实现了硬件启动和软件启动的分离。因此需要存储的硬件启动相关指令不需要很多,能够很容易地保存在容量较小的 ROM 或 FLASH 中。
2. bootloader 在硬件初始化完后,需要为软件启动(即操作系统内核的功能)做相应的准备,比如需要将内核镜像从存放它的存储器(比如磁盘)中读到 RAM 中。既然 bootloader 需要将内核镜像加载到内存中,那么它就能选择使用哪一个内核镜像进行加载,即实现多重开机的功能。使用 bootloader 后,我们就能够在同一个硬件上选择运行不同的操作系统了。
3. bootloader 主要负责硬件启动相关工作,同时操作系统内核则能够专注于软件启动以及对用户提供服务的工作,从而降低了硬件相关代码和软件相关代码的耦合度,有助于操作系统的移植。使用 bootloader 更清晰地划分了硬件启动和软件启动的边界,使操作系统与硬件交互的抽象层次提高了,从而简化了操作系统的开发和移植工作。

A.2.1 Bootloader

从操作系统的角度看,bootloader 的目标就是正确地找到内核并加载执行。另外,由于 bootloader 的实现依赖于 CPU 的体系结构,因此大多数 bootloader 都分为 stage1 和 stage2 两个部分。

在 stage1 时,此时需要初始化硬件设备,包括 watchdog timer、中断、时钟、内存等。需要注意的一个细节是,此时内存 RAM 尚未初始化完成,因而 stage1 运行的 bootloader 程序直接从非易失存储器上(比如 ROM 或 FLASH)加载。由于当前阶段不能在内存 RAM 中运行,其自身运行会受诸多限制,比如某些非易失存储器(ROM)不可写,即使程序可写的 FLASH 也有存储空间限制。这就是为什么需要 stage2 的原因。所以,stage1 除了初始化基本的硬件设备以外,会为加载 stage2 准备 RAM 空间,然后将 stage2 的代码复制到 RAM 空间,并且设置堆栈,最后跳转到 stage2 的入口函数。

stage2 运行在 RAM 中,此时有足够的运行环境从而可以用 C 语言来实现较为复杂的功能。这一阶段的工作包括,初始化这一阶段需要使用的硬件设备以及其他功能,然后将内核镜像从存储器读到 RAM 中,并为内核设置启动参数,最后将 CPU 指令寄存器的内容设置为内核入口函数的地址,即可将控制权从 bootloader 转交给操作系统内核。

从 CPU 上电到操作系统内核被加载的整个启动的步骤如图 A.1 所示。

需要注意的是,以上 bootloader 的两个工作阶段只是从功能上论述内核加载的过程,在具体实现上不同的系统可能有所差别,而且对于不同的硬件环境也会有些不同。在我们常见的 x86 PC 的启动过程中,首先执行的是 BIOS 中的代码,主要完成硬件初始化相关的工作,然后 BIOS 会从 MBR (master boot record, 开机硬盘的第一个扇区) 中读取开机信息。在 Linux 中常说的 GRUB 和 LILO 这两种开机管理程序就是保存在 MBR 中。

Note A.2.1 GRUB (GRand Unified Bootloader) 是 GNU 项目的一个多操作系统启动程序。简单的说,就是可以用于有多个操作系统的机器上,在刚开机的时候选择一个操作系统进行引导。如果安装过 Ubuntu 一类的发行版的话,一开机出现的那个选择系统用的菜单就是 GRUB 提供的。

(这里以 GRUB 为例)BIOS 加载 MBR 中的 GRUB 代码后就把 CPU 交给了 GRUB, GRUB 的工作就是一步步的加载自身代码,从而识别文件系统,然后就能够将文件系统内的内核镜像

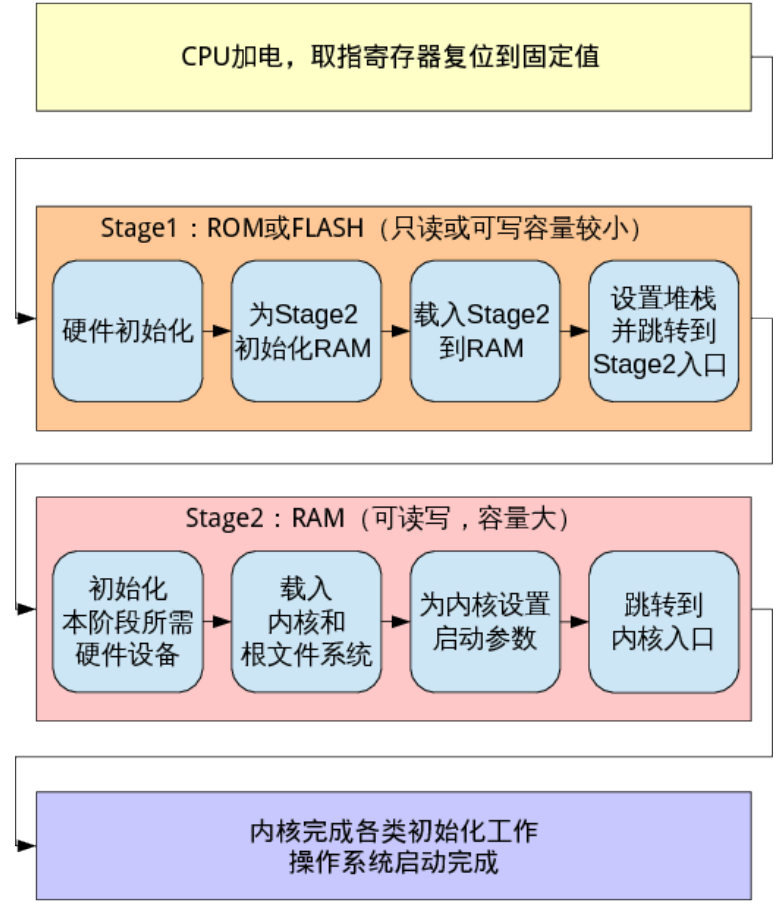


图 A.1: 启动的基本步骤

文件加载到内存中，并将 CPU 控制权转交给操作系统内核。这样看来，其实 BIOS 和 GRUB 的前一部分构成了前述 stage1 的工作，而 stage2 的工作则是完全在 GRUB 中完成的。

Note A.2.2 bootloader 有两种操作模式：启动加载模式和下载模式。区别是前者是通过本地设备中的内核镜像文件启动操作系统的，而后者是通过串口或以太网等通信手段将远端的内核镜像下载到内存。

A.3 编译与链接详解

一个简单的 C 程序

```

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

我们以代码A.3为例，讲述我们这个冗长的故事。我们首先探究这样一个问题：**含有多个 C 文件的工程是如何编译成一个可执行文件的？**

这段代码相信你非常熟悉了，不知你有没有注意到过这样一个小细节：`printf` 的定义在哪里？¹我们都学过，C 语言中函数必须有定义才能被调用，那么 `printf` 的定义在哪里呢？你一定会笑一笑说，别傻了，不就在 `stdio.h` 中吗？我们在程序开头通过 `include` 引用了它的。然而事实真的是这样吗？我们进去看一看 `stdio.h` 里到底有些什么。

stdio.h 中关于 printf 的内容

```

1  /*
2   *      ISO C99 Standard: 7.19 Input/output      <stdio.h>
3   */
4
5  /* Write formatted output to stdout.
6
7   This function is a possible cancellation point and therefore not
8   marked with __THROW. */
9  extern int printf (const char *__restrict __format, ...);
```

在代码A.3中，我们展示了从当前系统的 `stdio.h` 中摘录出的与 `printf` 相关的部分。可以看到，我们所引用的 `stdio.h` 中只有声明，但并没有 `printf` 的定义。或者说，并没有 `printf` 的具体实现。可没有具体的实现，我们究竟是如何调用 `printf` 的呢？我们怎么能够调用一个没有实现的函数呢？

我们来一步一步探究，`printf` 的实现究竟被放在了哪里，又究竟是在何时被插入到我们的程序中的。首先，我们要求编译器**只进行预处理（通过-E 选项）**，而不编译。Linux 命令如下：

¹`printf` 位于标准库中，而不在我们的 C 代码中。将标准库和我们自己编写的 C 文件编译成一个可执行文件的过程，与将多个 C 文件编译成一个可执行文件的过程相仿。因此，我们通过探究 `printf` 如何和我们的 C 文件编译到一起，来展示整个过程。

```
1 gcc -E 源代码文件名
```

你可以使用重定向将上述命令输出重定向至文件，以便于观察输出情况。

```
1  /* 由于原输出太长，这里只能留下很少很少的一部分。 */
2  typedef unsigned char __u_char;
3  typedef unsigned short int __u_short;
4  typedef unsigned int __u_int;
5  typedef unsigned long int __u_long;
6
7
8  typedef signed char __int8_t;
9  typedef unsigned char __uint8_t;
10 typedef signed short int __int16_t;
11 typedef unsigned short int __uint16_t;
12 typedef signed int __int32_t;
13 typedef unsigned int __uint32_t;
14
15 typedef signed long int __int64_t;
16 typedef unsigned long int __uint64_t;
17
18 extern struct _IO_FILE *stdin;
19 extern struct _IO_FILE *stdout;
20 extern struct _IO_FILE *stderr;
21
22 extern int printf (const char *__restrict __format, ...);
23
24 int main()
25 {
26     printf("Hello World!\n");
27     return 0;
28 }
```

可以看到，C 语言的预处理器将头文件的内容添加到了源文件中，但同时我们也能看到，这里一阶段并没有 `printf` 这一函数的定义。

之后，我们将 `gcc` 的 `-E` 选项换为 `-c` 选项，只编译而不链接，产生一个同名的 `.o` 目标文件。命令如下：

```
1 gcc -c 源代码文件名
```

我们对其进行反汇编²，反汇编并将结果导出至文本文件的命令如下：

```
1 objdump -DS 要反汇编的目标文件名 > 导出文本文件名
```

`main` 函数部分的结果如下

```
1 hello.o:      file format elf64-x86-64
2
3 Disassembly of section .text:
4
5 0000000000000000 <main>:
6   0:  55                push    %rbp
7   1:  48 89 e5          mov     %rsp,%rbp
8   4:  bf 00 00 00 00    mov     $0x0,%edi
9   9:  e8 00 00 00 00    callq   e <main+0xe>
```

²为了便于你重现，我们这里没有选择 MIPS，而选择了在流行的 x86-64 体系结构上进行反汇编。同时，由于 x86-64 的汇编是 CISC 汇编，看起来会更为清晰一些。

```

10      e:  b8 00 00 00 00      mov    $0x0,%eax
11      13:  5d                      pop    %rbp
12      14:  c3                      retq

```

我们只需要注意中间那句 `callq` 即可，这一句是调用函数的指令。对照左侧的机器码，其中 `e8` 是 `call` 指令的操作码。根据 x86 指令的特点，`e8` 后面应该跟的是 `printf` 的地址。可在这里我们却发现，**本该填写 `printf` 地址的位置上被填写了一串 0**。那个地址显然不可能是 `printf` 的地址。也就是说，直到这一步，`printf` 的具体实现依然不在我们的程序中。

最后，我们允许 `gcc` 进行链接，也就是**正常地编译**出可执行文件。然后，再用 `objdump` 进行反汇编。命令如下，其中 `-o` 选项用于指定输出的目标文件名，如果不设置的话默认为 `a.out`

```

1  gcc [-o 输出可执行文件名] 源代码文件名
2  objdump -DS 输出可执行文件名 > 导出文本文件名

```

反汇编结果如下

```

1  hello:      file format elf64-x86-64
2
3
4  Disassembly of section .init:
5
6  0000000004003a8 <_init>:
7      4003a8:  48 83 ec 08      sub    $0x8,%rsp
8      4003ac:  48 8b 05 0d 05 20 00  mov    0x20050d(%rip),%rax
9      4003b3:  48 85 c0          test   %rax,%rax
10     4003b6:  74 05           je     4003bd <_init+0x15>
11     4003b8:  e8 43 00 00 00   callq 400400 <__gmon_start__@plt>
12     4003bd:  48 83 c4 08      add    $0x8,%rsp
13     4003c1:  c3              retq
14
15  Disassembly of section .plt:
16
17  0000000004003d0 <puts@plt-0x10>:
18     4003d0:  ff 35 fa 04 20 00  pushq  0x2004fa(%rip)
19     4003d6:  ff 25 fc 04 20 00  jmpq   *0x2004fc(%rip)
20     4003dc:  0f 1f 40 00      nopl   0x0(%rax)
21
22  0000000004003e0 <puts@plt>:
23     4003e0:  ff 25 fa 04 20 00  jmpq   *0x2004fa(%rip)
24     4003e6:  68 00 00 00 00    pushq  $0x0
25     4003eb:  e9 e0 ff ff ff    jmpq   4003d0 <_init+0x28>
26
27  0000000004003f0 <__libc_start_main@plt>:
28     4003f0:  ff 25 f2 04 20 00  jmpq   *0x2004f2(%rip)
29     4003f6:  68 01 00 00 00    pushq  $0x1
30     4003fb:  e9 d0 ff ff ff    jmpq   4003d0 <_init+0x28>
31
32  000000000400400 <__gmon_start__@plt>:
33     400400:  ff 25 ea 04 20 00  jmpq   *0x2004ea(%rip)
34     400406:  68 02 00 00 00    pushq  $0x2
35     40040b:  e9 c0 ff ff ff    jmpq   4003d0 <_init+0x28>
36
37  Disassembly of section .text:
38
39  000000000400410 <main>:
40     400410:  48 83 ec 08      sub    $0x8,%rsp
41     400414:  bf a4 05 40 00    mov    $0x4005a4,%edi

```

```

42 400419: e8 c2 ff ff ff callq 4003e0 <puts@plt>
43 40041e: 31 c0 xor %eax,%eax
44 400420: 48 83 c4 08 add $0x8,%rsp
45 400424: c3 retq
46
47 0000000000400425 <_start>:
48 400425: 31 ed xor %ebp,%ebp
49 400427: 49 89 d1 mov %rdx,%r9
50 40042a: 5e pop %rsi
51 40042b: 48 89 e2 mov %rsp,%rdx
52 40042e: 48 83 e4 f0 and $0xfffffffffffffff0,%rsp
53 400432: 50 push %rax
54 400433: 54 push %rsp
55 400434: 49 c7 c0 90 05 40 00 mov $0x400590,%r8
56 40043b: 48 c7 c1 20 05 40 00 mov $0x400520,%rcx
57 400442: 48 c7 c7 10 04 40 00 mov $0x400410,%rdi
58 400449: e8 a2 ff ff ff callq 4003f0 <__libc_start_main@plt>
59 40044e: f4 hlt
60 40044f: 90 nop

```

篇幅所限，余下的部分没法再展示了（大约还有 100 来行）。

当你看到熟悉的“hello world”被展开成如此“臃肿”的代码，可能不忍直视。但是别急，我们还是只把注意力放在主函数中，这一次，我们可以惊喜的看到，主函数里那句 `callq` 后面已经不再是一串 0 了。那里已经被填入了一个地址。从反汇编代码中我们也可以看到，这个地址就在这个可执行文件里，就在被标记为 `puts@plt` 的那个位置上。虽然搞不清楚那个东西是什么，但显然那就是我们所调用的 `printf` 的具体实现了。

由此，我们不难推断，`printf` 的实现是在链接（Link）这一步骤中被插入到最终的可执行文件中的。那么，了解这个细节究竟有什么用呢？作为一个库函数，`printf` 被大量的程序所使用。因此，每次都将其编译一遍实在太浪费时间了。`printf` 的实现其实早就被编译成了二进制形式。但此时，`printf` 并未链接到程序中，它的状态与我们利用 `-c` 选项产生的 `hello.o` 相仿，都还处于未链接的状态。而在编译的最后，链接器（Linker）会将所有的目标文件链接在一起，将之前未填写的地址等信息填上，形成最终的可执行文件，这就是链接的过程。

对于拥有多个 `c` 文件的工程来说，编译器会首先将所有的 `c` 文件以文件为单位，编译成 `.o` 文件。最后再将所有的 `.o` 文件以及函数库链接在一起，形成最终的可执行文件。整个过程如图 A.2 所示。

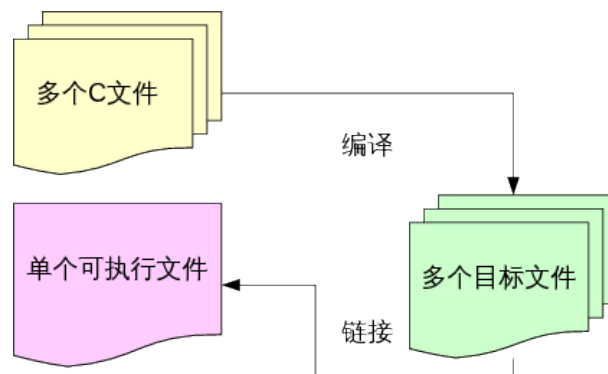


图 A.2: 编译、链接的过程

A.4 printf 格式具体说明

下面是我们需要完成的 `printf` 函数的具体说明，同学们可以参考 `cppreference` 中有关 C 语言 `printf` 函数的文档³或者 C++ 文档⁴，对 `printf` 函数进行更加详细的了解。

函数原型：

```
1 void printf(const char* fmt, ...)
```

参数 `fmt` 除了可以包含直接输出的字符，还可以包含格式符 (format specifiers)，类似 `printf` 中的格式字符串，但简化并新增了一些功能。格式符的原型为：

%[flags] [width] [length] <specifier>

其中 `specifier` 指定了输出变量的类型，参见下表 A.1：

表 A.1: Specifiers 说明

Specifier	输出	例子
b	无符号二进制数	110
d D	十进制数	920
o O	无符号八进制数	777
u U	无符号十进制数	920
x	无符号十六进制数，字母小写	1ab
X	无符号十六进制数，字母大写	1AB
c	字符	a
s	字符串	sample

除了 `specifier` 之外，格式符也可以包含一些其它可选的副格式符 (sub-specifier)，有 `flag`(表 A.2)：

表 A.2: flag 说明

flag	描述
-	在给定的宽度 (width) 上左对齐输出，默认为右对齐
0	当输出宽度和指定宽度不同的时候，在空白位置填充 0

和 `width`(表 A.3)：

表 A.3: width 说明

width	描述
数字	指定了要打印数字的最小宽度，当这个值大于要输出数字的宽度，则对多出的部分填充空格，但当这个值小于要输出数字的宽度的时候则不会对数字进行截断。

³<https://zh.cppreference.com/w/c/io/fprintf>

⁴<https://www.cplusplus.com/reference/cstdio/printf>

另外，还可以使用 `length` 来修改数据类型的长度，在 C 中我们可以使用 `l`、`ll`、`h` 等，但这里我们只使用 `l`，参看下表A.4

表 A.4: `length` 说明

length	Specifier	
	d D	b o O u U x X
l	long int	unsigned long int

A.5 MIPS 汇编与 C 语言

在操作系统编程中，不可避免地要接触到汇编语言。我们经常需要从 C 语言中调用一些汇编语言写成的函数，或者反过来，在汇编中跳转到 C 函数。为了帮助同学更好的了解 c 语言于汇编之间的联系，我们在附录的这一节中补充了 MIPS 汇编的相关知识，介绍了常见的 C 语言语法与汇编的对应关系。我们给出样例代码A.5，介绍典型的 C 语言中的语句对应的汇编代码。

样例程序

```
1  int fib(int n)
2  {
3      if (n == 0 || n == 1) {
4          return 1;
5      }
6      return fib(n-1) + fib(n-2);
7  }
8
9  int main()
10 {
11     int i;
12     int sum = 0;
13     for (i = 0; i < 10; ++i) {
14         sum += fib(i);
15     }
16
17     return 0;
18 }
```

A.5.1 循环与判断

这里你可能会问了，样例代码里只有循环啊！哪里有什么判断语句呀？事实上，由于 MIPS 汇编中没有循环这样的高级结构，所有的循环均是采用判断加跳转语句实现的，所以我们将循环语句和判断语句合并在一起进行分析。我们分析代码的第一步，就是要将循环等高级结构，用**判断加跳转**的方式替代。例如，代码A.5第 13-15 行的循环语句，其最终的实现可能就如下面的 C 代码所展示的那样。

```
1      i = 0;
2      goto CHECK;
3  FOR:  sum += fib(i);
4      ++i;
```

```
5 CHECK: if (i < 10) goto FOR;
```

将样例程序编译⁵，我们观察其反汇编代码。对照汇编代码和我们刚才所分析出来的 C 代码。我们基本就能够看出来其间的对应关系。这里，我们将对应的 C 代码标记在反汇编代码右侧。

```

1  400158:    sw      zero,16(s8)      #      sum = 0;
2  40015c:    sw      zero,20(s8)      #      i = 0;
3  400160:    j       400190 <main+0x48> #      goto CHECK;
4  400164:    nop                                # -----
5  400168:    lw      a0,20(s8)        #  FOR:
6  40016c:    jal     4000b0 <fib>      #
7  400170:    nop                                #
8  400174:    move    v1,v0            #      sum += fib(i);
9  400178:    lw      v0,16(s8)        #
10 40017c:    addu    v0,v0,v1         #
11 400180:    sw      v0,16(s8)        #
12 400184:    lw      v0,20(s8)        # -----
13 400188:    addiu   v0,v0,1         #      ++i;
14 40018c:    sw      v0,20(s8)        # -----
15 400190:    lw      v0,20(s8)        # CHECK:
16 400194:    slti    v0,v0,10         #      if (i < 10)
17 400198:    bnez    v0,400168 <main+0x20> #      goto FOR;
18 40019c:    nop
```

再将右边的 C 代码对应回原来的 C 代码，我们就能够大致知道每一条汇编语句所对应的原始的 C 代码是什么了。可以看出，判断和循环主要采用 `slt`、`slti` 判断两数间的大小关系，再结合 `b` 类型指令根据对应条件跳转。以这些指令为突破口，我们就能大致识别出循环结构、分支结构了。

A.5.2 函数调用

Note A.5.1 注意区分函数的调用方和被调用方来分析函数调用。

这里选用样例程序中的 `fib` 这个函数来观察函数调用相关的内容。这个函数是一个递归函数，因此，它函数调用过程的调用者，同时也是被调用者。我们可以从中观察到如何调用一个函数，以及一个被调用的函数应当做些什么工作。

我们先将整个函数调用过程用高级语言来表示一下。

```

1  int fib(int n)
2  {
3      if (n == 0) goto BRANCH;
4      if (n != 1) goto BRANCH2;
5  BRANCH: v0 = 1;
6          goto RETURN;
7  BRANCH2: v0 = fib(n-1) + fib(n-2);
8  RETURN: return v0;
9  }
```

然而，之后在分析汇编代码的时候，我们会发现有很多 C 语言中没有表示出来的东西。例如，在函数开头，有一大串的 `sw`，结尾处又有一大串的 `lw`。这些东西究竟是在做些什么呢？

⁵为了生成更简单的汇编代码，我们采用了 `-nostdlib -static -mno-abicalls` 这三个编译参数


```

1  004000b0 <fib>:
2  4000b0:      27bdfdd8      addiu    sp,sp,-40
3  4000b4:      afbf0020      sw        ra,32(sp)
4  4000b8:      afbe001c      sw        s8,28(sp)
5  4000bc:      afb00018      sw        s0,24(sp)
6  # 中间暂且掠过, 只关注一系列 sw 和 lw 操作。
7  400130:      8fbf0020      lw        ra,32(sp)
8  400134:      8fbe001c      lw        s8,28(sp)
9  400138:      8fb00018      lw        s0,24(sp)
10 40013c:      27bd0028      addiu    sp,sp,40
11 400140:      03e00008      jr        ra
12 400144:      00000000      nop

```

我们来回忆一下 C 语言的递归。C 语言递归的过程和栈这种数据结构有着惊人的相似性，函数递归到底以及返回过程，就好像栈的后入先出。而且每一次递归操作就仿佛将当前函数的所有变量和状态压入了一个栈中，待到返回时再从栈中弹出来，“一切”都保持原样。⁶

好了，回忆起了这个细节，我们再来看看汇编代码。在函数的开头，编译器为我们添加了一组 `sw` 操作，将所有当前函数所需要用到的寄存器原有的值全部保存到了内存中⁷。而在函数返回之前，编译器又加入了一组 `lw` 操作，将值被改变的寄存器全部恢复为原有的值。

我们惊奇地发现：编译器在函数调用的前后为我们添加了一组压栈 (`push`) 和弹栈 (`pop`) 的操作，为我们保存了函数的当前状态。函数的开始，编译器首先减小 `sp` 指针的值，为栈分配空间。并将需要保存的值放置在栈中。当函数将要返回时，编译器再增加 `sp` 指针的值，释放栈空间。同时，恢复之前被保存的寄存器原有的值。这就是为何 C 语言的函数调用和栈有着很大的相似性的原因：在函数调用过程中，编译器的确为我们维护了一个栈。这下同学们应该也不难理解，为什么复杂函数在递归层数过多时会导致程序崩溃，也就是我们常说的“栈溢出”。

Note A.5.2 `ra` 寄存器存放了函数的返回地址。使得被调用的函数结束时得以返回到调用者调用它的地方。但你有没有想过，我们其实可以将这个返回点设置为别的函数的入口，使得该函数在返回时直接进入另一个函数中，而不是回到调用者哪里？一个函数调用了另一个函数，而返回时，返回到第三个函数中，是不是也是一种很有价值的编程模型呢？如果你对此感兴趣，可以了解一下函数式编程中的 `Continuations` 的概念 (推荐 [Functional Programming For The Rest Of Us](#) 这篇文章)，在很多新近流行起来的语言中，都引入了类似的想法。

在我们看到了一个函数作为被调用者做了哪些工作后，我们再来看看，作为函数的调用者需要做什么？如何调用一个函数？如何传递参数？又如何获取返回值？让我们来看一下，`fib` 函数调用 `fib(n-1)` 和 `fib(n-2)` 时，编译器为我们生成的汇编代码⁸

```

1  lw      $2,40($fp)      # v0 = n;
2  addiu   $2,$2,-1        # v0 = v0 - 1;
3  move    $4,$2           # a0 = v0; // 即 a0=n-1
4  jal     fib             # v0 = fib(a0);
5  nop                                #
6
7  move    $16,$2          # s0 = v0;

```

⁶这里“压入栈的状态”通常称为“栈帧”，栈帧中保存了该函数的返回地址和局部变量。

⁷其实这样说并不准确，后面我们会看到，有些寄存器的值是由调用者负责保存的，有些是由被调用者保存的。但这里为了理解方便，我们姑且认为被调用的函数保存了调用者的所有状态吧

⁸为了方便你了解自己手写汇编时应当怎样写，我们这一次采用汇编代码，而不是反汇编代码。这里注意，`fp` 和上面反汇编出的 `s8` 其实是同一个寄存器，只是有两个不同的名字而已

```

8  lw      $2,40($fp)      # v0 = n;
9  addiu   $2,$2,-2        # v0 = n - 2;
10 move    $4,$2           # a0 = v0;      // 即 a0=n-2
11 jal     fib             # v0 = fib(a0);
12 nop
13
14 addu     $16,$16,$2      # s0 += v0;
15 sw      $16,16($fp)     #

```

我们将汇编所对应的语义用 C 语言标明在右侧。可以看到，调用一个函数就是将参数存放在 `a0-a3` 寄存器中（我们暂且不关心参数非常多的函数会如何处理），然后使用 `jal` 指令跳转到相应的函数中。函数的返回值会被保存在 `v0-v1` 寄存器中。我们通过这两个寄存器的值来获取返回值。

A.5.3 通用寄存器使用约定

为了和编译器等程序相互配合，我们需要遵循一些使用约定。这些规定与硬件无关，硬件并不关心寄存器具体被用于什么用途。这些规定是为了让不同的软件之间得以协同工作而制定的。MIPS 中一共有 32 个通用寄存器 (General Purpose Registers)，其用途如表 A.5 所示。

表 A.5: MIPS 通用寄存器

寄存器编号	助记符	用途
0	zero	值总是为 0
1	at	（汇编暂存寄存器）一般由汇编器作为临时寄存器使用。
2-3	v0-v1	用于存放表达式的值或函数的整形、指针类型返回值
4-7	a0-a3	用于函数传参。其值在函数调用的过程中不会被保存。若函数参数较多，多出来的参数会采用栈进行传递
8-15	t0-t7	用于存放表达式的值的临时寄存器；其值在函数调用的过程中不会被保存。
16-23	s0-s7	保存寄存器；这些寄存器中的值在经过函数调用后不会被改变。
24-25	t8-t9	用于存放表达式的值的临时寄存器；其值在函数调用的过程中不会被保存。当调用位置无关函数 (position independent function) 时，25 号寄存器必须存放被调用函数的地址。
26-27	k0-k1	仅被操作系统使用。
28	gp	全局指针和内容指针。
29	sp	栈指针。
30	fp 或 s8	保存寄存器（同 s0-s7）。也可用作帧指针。
31	ra	函数返回地址。

其中，只有 16-23 号寄存器和 28-30 号寄存器的值在函数调用的前后是不变的⁹。对于 28 号寄存器有一个特例：当调用位置无关代码 (position independent code) 时，28 号寄存器的值是

⁹请注意，这里的不变并不意味着它们的值在函数调用的过程中不能被改变。只是指它们的值在函数调用后和函数调用前是一致的。

不被保存的。

除了这些通用寄存器之外，还有一个特殊的寄存器：PC 寄存器。这个寄存器中储存了当前要执行的指令的地址。当你在 QEMU 仿真器上调试内核时，可以留意一下这个寄存器。通过 PC 的值，我们就能够知道当前内核在执行的代码是哪一条，或者触发中断的代码是哪一条等等。

A.5.4 LEAF、NESTED 和 END

在我们的实验中，可以在 `init/start.S` 里找到 `LEAF(_start)` 这样的代码，其中的 `LEAF` 其实是一个使用 `.macro` 定义的宏。那么首先让我们来了解一下 `LEAF`、`NESTED` 和 `END` 这三个宏吧。

Note A.5.3 阅读 `LEAF` 等宏定义的时候，我们会发现这些宏的定义是一系列以 `.` 开头的指令。这些指令不是 MIPS 汇编指令，而是 Assembler directives(参考 https://ftp.gnu.org/old-gnu/Manuals/gas-2.9.1/html_chapter/as_7.html)，它们的主要作用是向汇编器提供细节信息，以辅助生成目标代码。

通过 `grep` 进行查找，发现可以在 `include/asm/asm.h` 中发现这三个宏定义。

首先让我们来看 `LEAF` 和 `NESTED` 的宏定义，两个宏的内容基本一致，仅在最后一行有区别。

```

1      /*
2      * LEAF - declare leaf routine
3      */
4      #define LEAF(symbol)          \
5      .globl symbol;                \
6      .align 2;                     \
7      .type symbol, @function;      \
8      .ent symbol;                  \
9      symbol:                       \
10     .frame sp, 0, ra
11
12     /*
13     * NESTED - declare nested routine entry point
14     */
15     #define NESTED(symbol, framesize, rpc) \
16     .globl symbol;                      \
17     .align 2;                           \
18     .type symbol, @function;            \
19     .ent symbol;                        \
20     symbol:                             \
21     .frame sp, framesize, rpc

```

下面我们逐行来理解这些宏定义。

第一行是对 `LEAF` 宏的定义，后面括号中的 `symbol` 类似于函数的参数，在宏定义中的作用类似，编译时在宏中会将 `symbol` 替换为实际传入的文本。

第二行中，`.globl` 的作用是“使标签对链接器可见”，这样即使在其它文件中也可以引用到 `symbol` 标签，从而使得其它文件中可以调用我们使用宏定义声明的函数。

第三行中，`.align` 的作用是“使下面的数据进行地址对齐”，这一行语句使得下面的 `symbol` 标签按 4 Byte 进行对齐（参数 `x` 代表以 2^x 字节对齐），从而使得我们可以使用 `jal` 指令跳转到这个函数（末尾拼接两位 0）。

第四行中，`.type` 的作用是设置 `symbol` 标签的类别，在这里我们设置了 `symbol` 标签为函数标签。

第五行中，`.ent` 的作用是标记每个函数的开头，需要与 `.end` 配对使用。这些标记使得可以在 Debug 时查看调用链。

第六行的开头便是 `symbol` 标签了，后面的 `.frame` 的用法如下。

```
1 .frame framereg, framesize, returnreg
```

第一个参数 `framereg` 是用于访问栈帧的寄存器，通常我们使用栈寄存器 `sp`，在创建栈帧时，`sp` 寄存器自减，栈空间向低地址增长一段内存用于栈帧的储存。

第二个参数 `framesize` 是栈帧占据的存储空间大小。

第三个参数 `returnreg` 是存储函数执行完的返回地址的寄存器。通常我们传入 `$0` 表示返回地址储存在栈帧空间中，有时在不需要返回的函数中我们会传入 `$ra` 表示返回地址存储在 `$ra` 寄存器中 (`$31`)。

Note A.5.4 在 `.frame` 的使用时，出现了概念“栈帧”，每个栈帧对应着一个未运行完的函数。栈帧中保存了该函数的返回地址和局部变量。在上一节的“函数调用”小节中有所介绍。

通常来说，栈帧的作用包括但不限于：存储函数的返回地址、存储调用方的临时变量与中间结果、向被调用方传递参数。

通过对比，我们可以发现 `LEAF` 宏和 `NESTED` 宏的区别就在于 `LEAF` 宏定义的函数在被调用时没有分配栈帧的空间记录自己的“运行状态”，`NESTED` 宏在被调用时分配了栈帧的空间用于记录自己的“运行状态”。

下面让我们来看 `END` 宏。

```
1 #define END(function) \
2     .end    function; \
3     .size   function,.-function
```

第一行是对 `END` 宏的定义，与上面 `LEAF` 与 `NESTED` 类似。

第二行的 `.end` 是为了与先前 `LEAF` 或 `NESTED` 声明中的 `.ent` 配对，标记了 `symbol` 函数的结束。

第三行的 `.size` 是标记了 `function` 符号占用的存储空间大小，将 `function` 符号占用的空间大小设置为 `.-function`，`.` 代表了当前地址，当前位置的地址减去 `function` 标签处的地址即可计算出符号占用的空间大小。

A.6 多级页表与页目录自映射

A.6.1 MOS 中的页目录自映射应用

在 Lab2 中，实现了内存管理，建立了两级页表机制。

页表的主要作用是维护虚页面到物理页面之间的映射关系，通常存放在内存中。操作系统对于页表的访问也是通过虚地址来进行。这也就意味着，页表同时也维护了自身所处的虚页面到实际物理页面之间的映射关系。

试想这样一个问题：如何在虚拟存储空间维护页表和页目录？下面介绍一下 MOS 中采用的“自映射”，即将页表和页目录映射到进程地址空间的实现方式。在两级页表中，一个进程的 4GB 地址空间均映射物理内存的话，那么就需要 4MB 来存放页表（1024 个页表），4KB 来存放页目录；如果页表和页目录都在进程的地址空间中得到映射，这意味着在 1024 个页表中，有一个页

表所对应的 4MB 空间就是这 1024 个页表占用的 4MB 空间。这一个特殊的页表就是页目录，它的 1024 个表项映射到这 1024 个页表。因此只需要 4MB 的空间即可容纳页表和页目录。

而 MOS 中，将页表和页目录映射到了用户空间中的 $0x7fc00000-0x80000000$ (共 4MB) 区域，这意味着 MOS 中允许在用户态下通过 UVPT 访问当前进程的页表和页目录。

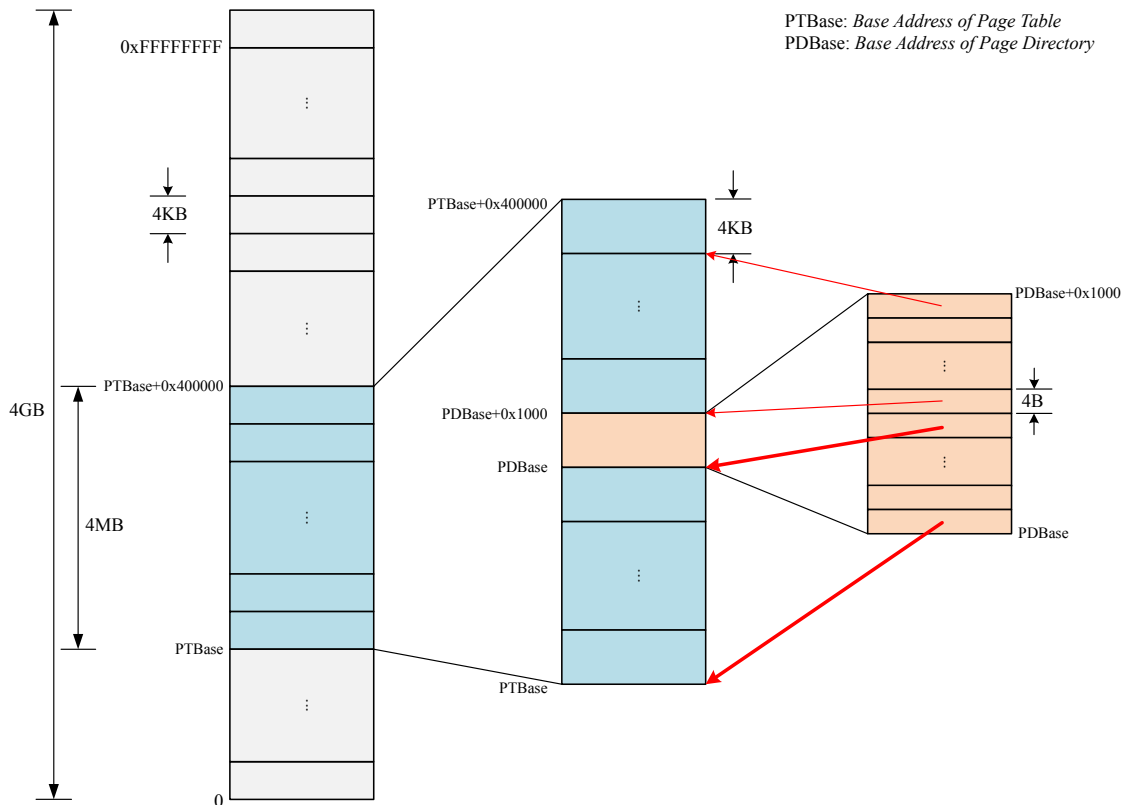


图 A.3: 页目录自映射

下面根据自映射的性质，计算出 MOS 中页目录的基地址：

$0x7fc00000-0x80000000$ 这 4MB 空间的起始位置（也就是第一个二级页表的基地址）对应着页目录的第一个页目录项。同时由于 1M 个页表项和 4GB 地址空间是线性映射的，不难算出 $0x7fc00000$ 这一个地址对应的应该是第 $0x7fc00000 \gg 12$ 个页表项（这一个页表项也就是第一个页目录项）。由于一个页表项占 4B 空间，因此第 $0x7fc00000 \gg 12$ 个页表项相对于页表基地址的偏移为 $(0x7fc00000 \gg 12) * 4$ ，即 $0x1ff000$ 。最终即可得到页目录基地址为 $0x7fdff000$ 。

A.6.2 其他页表机制

在其他系统中，还会使用三级页表等更多级的页表机制。请结合操作系统理论课所学知识，查阅相关资料，回答下述思考题。

Thinking A.1 在现代的 64 位系统中，提供了 64 位的字长，但实际上不是 64 位页式存储系统。假设在 64 位系统中采用三级页表机制，页面大小 4KB。由于 64 位系统中字长为 8B，且页目录也占用一页，因此页目录中有 512 个页目录项，因此每级页表都需要 9 位。因此在 64 位系统下，总共需要 $3 \times 9 + 12 = 39$ 位就可以实现三级页表机制，并不需要 64 位。

现考虑上述 39 位的三级页式存储系统，虚拟地址空间为 512 GB，若三级页表的基地址为 PT_{base} ，请计算：

- 三级页表页目录的基地址。
- 映射到页目录自身的页目录项（自映射）。

A.6.3 虚拟内存和磁盘中的 ELF 文件

我们将通过一个实例，探讨 bss 段在虚拟内存和磁盘 ELF 文件中是否占据空间这一问题。

testbss 测试

```

1  #include <lib.h>
2  #define ARRAYSIZE 0x1000000
3  int bigarray[ARRAYSIZE] = {0};
4  int main(int argc, char **argv) {
5      int i;
6
7      debugf("Making sure bss works right... \n");
8      for (i = 0; i < ARRAYSIZE; i++) {
9          if (bigarray[i] != 0) {
10             user_panic("bigarray[%d] isn't cleared!\n", i);
11         }
12     }
13     for (i = 0; i < ARRAYSIZE; i++) {
14         bigarray[i] = i;
15     }
16     for (i = 0; i < ARRAYSIZE; i++) {
17         if (bigarray[i] != i) {
18             user_panic("bigarray[%d] didn't hold its value!\n", i);
19         }
20     }
21     debugf("Bss is good\n");
22     return 0;
23 }
```

可以参考 Lab4 中 pingpong 和 fktest 的编译与加载流程，完成 testbss 的测试：在 init/init.c 中创建相应的初始进程，观察相应的实验现象。如果能正确运行，则说明我们 Lab3 完成的 load_icode 系列函数正确地完成了在内核态中加载 ELF 时 bss 段的初始化。

正确结果展示：

```

1  Making sure bss works right...
2  Bss is good
```

验证了 bss 段被 load_icode 系列函数正确初始化后，我们可以对 user/testbss.c 的全局数组 bigarray 做以下三种不同的初始化处理，分别 make clean && make 后对三次的 bss.b 进行命令解析：

- 执行 size user/testbss.b 命令，size 命令的前四列结果默认为十进制显示，分别代表 text 段、data 段、bss 段、各段之和的大小，展示虚拟内存空间的分配信息。分析比较三次结果的 data 段与 bss 段大小，思考原因。

- 执行 `ls -l user/testbss.b` 命令，查看二进制文件的详细信息，找到结果中的第五列也就是月份前的那列数字，它表示文件在磁盘中占据的大小，分析比较三次结果并思考原因。

正确结果展示：

- `testbss.c` 第三行为 `int bigarray[ARRAYSIZE]`，即不对全局变量初始化时：

```

1      text    data    bss      dec    hex    filename
2      14280   13851   40964   69095   10de7   user/testbss.b
3
4      -rwxrwxr-x 1 git git 41761 MMM dd HH:mm user/testbss.b

```

- `testbss.c` 第三行为 `int bigarray[ARRAYSIZE]={0}`，即全局变量初始化为 0 时：

```

1      text    data    bss      dec    hex    filename
2      14280   13851   40964   69095   10de7   user/testbss.b
3
4      -rwxrwxr-x 1 git git 41761 MMM dd HH:mm user/testbss.b

```

- `testbss.c` 第三行为 `int bigarray[ARRAYSIZE]={1}`，即对全局变量做初始化时：

```

1      text    data    bss      dec    hex    filename
2      14280   54811     4    69095   10de7   user/testbss.b
3
4      -rwxrwxr-x 1 git git 82721 MMM dd HH:mm user/testbss.b

```

观察比对结果，我们可以得出以下三点：

- 当第三次全局变量初始化时，相较于第一次不初始化的结果，`data` 段增加了 40960 字节，`bss` 段减少了 40960 字节。`bigarray` 数组的大小为 10240 个 `int`，也就是 $10240 \times 4 \text{ Byte} = 40960 \text{ Byte}$ 。说明在虚拟内存布局中，初始化的全局变量保存在 `data` 段，未初始化的全局变量保存在 `bss` 段。
- 当第二次全局变量初始化为 0 时，与第一次不初始化的结果完全一致。说明对于未初始化和初始化为 0 的全局变量，二者在用户空间地址中的数据分配的存放位置相同，均在 `bss` 段。
- 第三次初始化全局变量时，ELF 文件在磁盘的大小为 82721 字节。而全局变量保存在 `bss` 段时，ELF 文件在磁盘的大小为 41761 字节。 $82721 \text{ Byte} - 41761 \text{ Byte} = 40960 \text{ Byte}$ ，说明 `bss` 段不在磁盘文件占据空间。

`bss` 段 (Block Started Symbol, 意为“以符号开始的块”)，存放全局未初始化/初始化为 0、静态未初始化/初始化为 0 的变量，只是简单维护地址空间中开始和结束的地址，在实际运行对内存区域有效地清零即可。

Note A.6.1 第三次 `testbss.c` 中似乎没有未初始化的全局变量，`bss` 段的 4 字节存放的什么变量呢？

我们可以使用 `objdump -t user/testbss.b | grep .bss` 命令反汇编查看变量名及其位置，将反汇编的结果经过管道，筛选显示包含“.bss”的内容。

可以找到位于 `.bss` 段的一条结果：`00412000 g 0 .bss 00000004 env。`

`env` 变量并不在 `testbss.c` 中定义。我们打开 `user/Makefile`，查看 `testbss.b` 的依赖文件，`entry.o` 由 `user/lib/entry.S` 编译而来。`user/lib/entry.S` 的 `_start`：后跳转到 `libmain()` 函数。`libmain` 函数位于 `user/lib/libos.c` 中，它为未初始化的 `env` 变

量赋值后跳转到 `main()` 函数，也就是我们自己在 `user` 目录下编写的测试文件的主函数。
`env` 变量在 `user/lib/libos.c` 中被定义。