

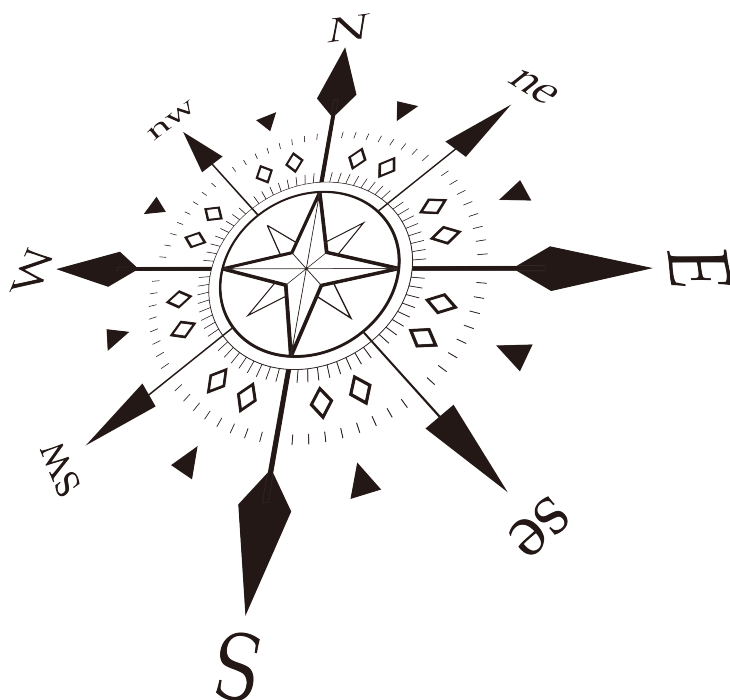
---

# 小操作系统实验 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*

2015

POWERED BY L<sup>A</sup>T<sub>E</sub>X

Die Philosophen haben die Welt nur verschieden interpretiert,  
es kommt aber darauf an, sie zu verändern.

哲学家们只是用不同的方式解释世界，而问题在于改变世界。

——马克思《关于费尔巴哈的提纲》

深夜，幽幽的荧屏，杂乱的代码，诡异的错误，不知所措的我们死死地盯着屏幕，试图解决实验里那些莫名其妙的问题。每当这时，我们都不禁抱怨：代码为什么有这么多错误，注释为什么这么混乱，指导书内容又为何如此鸡肋。抱怨之余我们也幻想了很多假如，相信曾挣扎于操作系统实验的人都曾这么想过：假如代码注释更完善些，假如实验的体系和计组一样成熟，假如...

我们最大的愿望，便是有一本指南针般的指导书。它可能会穿插设置一些小练习让我们更容易上手并带来满足感，它可能会针对性地让我们对某些细节进行深度思考，它可能会从实验角度为我们展现一个真实的操作系统，它可能会让我们着迷于操作系统实验，它优雅而美丽，它平易而近人。假如那样，我们或许能更专注于操作系统本身设计吧。

但我们不希望我们只在心里假如，我们想做些什么。一路走来，我们耗费了太多的时间在不必要的地方，因此，我们不希望这样的经历延续下去，一年又一年，一届又一届，我们想做些改变。

这正是我们重新整理并改写指导书的初衷，而你将要看到的，就是我们为改善操作系统实验而作出的努力。仅以此书献给我们所热爱的学弟学妹们，愿它能带给你一次愉快而奇妙的编写操作系统的体验。

学生的潜力是无穷的，但我们还只是操作系统实验改革的起点。本书还有很多不足之处与不完善之处，但是我们可以向你许诺，不再是大量无用材料的堆砌，不再是混乱无逻辑的排版，除很少部分的转载外，其他内容都是我们日日夜夜呕心沥血的原创。同时，秉着减轻代码难度，加强思考深度的初心，我们创造了许多思考训练。所有的思考训练都是我们精心设计，是世上绝无仅有的。所以我们请求你，希望你多花些时间来看看这本书，自主思考，勇于探索，并鼓励你与伙伴积极交流。

哪天我们若能在路上听到你和小伙伴神采飞扬地讨论操作系统实验中的某个细节，那时或许我们会在心里欣慰地轻叹一声

“为了这帮兔崽子，值了！”

在重编指导书的过程中，我们对于读者的知识掌握情况做了如下的假设。

- 熟练掌握 C 语言，特别是指针、结构体的使用。
- 对于 MIPS 体系结构有基本的了解（正常完成了计算机组成课程实验）。
- 初步了解中断、异常等概念。

指导书有三种特殊标记的段落：Note、Exercise 和 Thinking。

**Note** 表示该段内容是补充说明，或者课外延展，可以选择性地跳过。

**Exercise** 我们需要完成的实验任务，我们将在服务器端对代码是否合格进行检查。

**Thinking** 思考训练，无固定答案，但是需要将你的见解写在实验报告中。注意这些思考训练不同于你以往所做的题目，它由**关键代码**、**见解**、**参考资料**组成。你需要使用参考资料与你认为的关键代码对你的见解进行支撑，为了做到这一点你可能需要做很多功课。这部分内容将作为操作系统实验成绩的重要参考。我们没有规定固定格式，只需要在说明前标明 Thinking 序号即可。

---

## 编者寄语 (按姓名汉语拼音顺序排列)

天道酬勤。

——何涛

行胜于言。

——刘乾

一个人可以走得很快，但一群人可以走得更远，我愿与你们一路同行，分享彼此的经验与知识。

——王鹿鸣

“掌握操作系统原理的最好途径就是自己编写一个操作系统，希望大家都能写出自己的操作系统。”

——王雷

操作系统是计算机系统的一个重要系统软件，也计算机专业教学的重要内容。该课程概念众多、内容抽象，不仅需要讲授操作系统的原理，而且还要通过实验加深对操作系统理解。实验对操作系统课程的学习是至关重要的，掌握操作系统原理的最好途径就是自己编写一个操作系统。

因此，从 1999 年我开始讲授这门课程以来，一直想寻找一个好的操作系统实验环境。我曾经尝试过 Minix、Nachos、Linux、WRK 等很多实验环境，其中还得到了微软亚洲研究院、SUN 中国研究院的帮助，但是一直没有找到合适的实验环境。直到 2007 年我发现了 MIT 的 JOS 系统，并指导学生刘智武在毕业设计期间完成了该系统实验。同时，在操作系统课上选了两个学生尝试完成 JOS 实验，但是效果不太好，由于 x86 启动比较复杂，他们在前两个实验上花费的时间太多了，以至于没时间完成后面的实验。因此，我开始尝试将实验适当简化，并移植到相对简单的 MIPS 上。

正好北航计算机学院在进行教学改革，希望将硬件课与软件课打通，加强学生系统能力的培养。在学院支持下，组织了刘阳、程致远、刘伟、朱沥可等同学，我们参考 Linux 代码，完成了向 MIPS 的移植工作。特别是刘阳同学，不仅编写了代码和手册，还完成了很多组织协调工作。这时候总算有了一个能让学生在第一学期完成的、相对完整的小型操作系统。在推广 MIPS 操作系统实验时，为了保证教学连续性，我们允许学生从 Windows、Linux 和 MIPS 操作系统中选择一个实验完成，并可以分组完成。

2010、2011 和 2012 年选择 MIPS 操作系统实验的同学人数分别为 3%、14%、30%，实验成绩也在逐步上升。在 2013 年的计算机学院实验班、2014 年和 2015 年计算机大班中开始全面推广，并要求每个同学独立完成。在实验教学过程中，我的研究生都当过我的助教，另外还有一些其他老师的研究生和一些本科生志愿者，这些同学共同完成了实验手册的编写、实验代码的完善和实验环境的搭建。这些人包括蔚鹏志、谭成鑫、王刚、王欢、李康、王振、王平、马春雷、师斌、张健、高超、康乔、禹舟健和宗毅等同学，特别是王振和马春雷对完成了大量实验手册完善工作，高超在沃天宇老师和师斌的帮助下独立完成了整个实验环境的搭建，宗毅完成了实验向 QEMU 的部分移植工作。我可能无法把所有人的名字列出来，但由衷地感谢他们！

最后我要感谢刘乾、王鹿鸣和何涛三位同学！他们修改了代码中的错误、加入了大量注释，特别是他们重写了整个实验指导手册。我再次感谢他们！

本实验的目标是在一学期内，以 MIPS 为基础，让学生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。操作系统实验共包括“内核制作与启动”、“内存管理”、“进程与中断”、“系统调用”、“文件系统”（选做）和“shell”（选做）等六个部分。

1. 内核制作与启动：了解计算机在加电之后，如何引导文件，初始化基本硬件设备，通过修改链接脚本，学习把一段程序放在指定的内存地址。
2. 内存管理：完成初始化 MMU，TLB，建立虚拟内存管理机制，并在内存中安排基本的内核数据结构布局。
3. 进程与中断：完成初始化进程运行环境，实现进程创建的基本方法和简单的进程调度算法。
4. 系统调用：进程使用内核服务都是通过系统调用的方式实现。
5. 文件系统：实现一个基于块设备的文件系统。
6. shell 命令解释程序：Shell 功能的实现，给用户提供了访问操作系统的接口。

从实验内容可以看出，现代操作系统基本的几个功能，例如内存空间，进程管理等，都得到实现。通过这些实验，学生能够更加深入的理解操作系统原理及其实现方法，同时也可以在这个基础上实现自己的功能，实现更加复杂的操作系统并完成一些有挑战性任务。

我相信，现在的实验指导手册和代码注释会使同学们在完成复杂操作系统实验时感到一些轻松。感谢这些同学对实验做出的贡献，希望你们不辜负他们的努力，用心完成实验。更希望你们能为实验和实验指导书提出更多的反馈意见！

## MIPS 操作系统实验大事表

---

- 1999 • 尝试 Minix、Nachos、Linux、WRK 等实验环境，还得到了微软亚洲研究院、SUN 中国研究院的帮助，但没有合适的实验环境。
- 2007 • 发现 MIT 的 JOS 实验，指导刘智武在毕业设计期间完成 JOS 实验。
- 2007 • 挑选学生尝试 JOS 实验，但是由于 x86 的启动比较复杂，学生只完成两个实验。开始尝试将实验移植到相对简单的 MIPS 上。
- 2009 • 北航计算机学院教学改革，在学院支持下，组织了刘阳、程致远、刘伟、朱沥可完成了 JOS 到 MIPS 的移植工作。
- 2010 • 选择 MIPS 操作系统实验的同学仅有 3%。
- 2011 • 选择 MIPS 操作实验的同学比例上升到 14%。
- 2012 • 选择的同学增加到了 30%，实验成绩稳步上升。
- 2013 • 在北航计算机学院实验班推广，并要求学生独立完成。
- 2014 • 在北航计算机学院大班中推广，并且在这期间，许多人作为研究生助教与本科生志愿者参与了实验手册的编写、环境搭建与代码的完善。
- 2015 • 有感于实验手册质量欠缺，何涛、王鹿鸣、刘乾同学完成本书的第一版撰写。
- 2016 • 更多的可能，期待你们来书写！

编者序	i
教师寄语	iii
<b>1 内核、Boot 和 printf</b>	<b>1</b>
1.1 实验目的	1
1.2 操作系统的启动	1
1.2.1 内核在哪里?	1
1.2.2 Bootloader	3
1.2.3 gxemul 中的启动流程	4
1.3 Let's hack the kernel!	5
1.3.1 ssh——远程连接到服务器	5
1.3.2 vim 或 nano——阅读并修改代码	6
1.3.3 Makefile——内核代码的地图	6
1.3.4 ELF——深入探究编译与链接	10
1.3.5 MIPS 内存布局——寻找内核的正确位置	15
1.3.6 Linker Script——控制加载地址	17
1.4 MIPS 汇编与 C 语言	20
1.4.1 循环与判断	21
1.4.2 函数调用	21
1.4.3 通用寄存器使用约定	23
1.5 实战 printf	24
1.6 Git——轻松维护和提交代码	25
1.6.1 手工的版本控制	25
1.6.2 Git 基础指引	26
1.6.3 Git 文件四状态	30
1.6.4 Git 三棵树	31
1.6.5 Git 时光机	32

1.6.6	Git 分支 . . . . .	34
1.6.7	Git 远程仓库与本地 . . . . .	37
1.6.8	Git 冲突与解决冲突 . . . . .	38
1.6.9	实验代码提交流程 . . . . .	40
1.7	实验思考 . . . . .	41
<b>2</b>	<b>内存管理</b>	<b>42</b>
2.1	实验目的 . . . . .	42
2.2	MMU 和 TLB . . . . .	42
2.3	MIPS 虚存映射布局 . . . . .	42
2.4	内存管理 . . . . .	43
2.5	物理内存管理 . . . . .	43
2.5.1	内存控制块 . . . . .	43
2.5.2	内存分配和释放 . . . . .	44
2.6	虚拟内存管理 . . . . .	45
2.6.1	两级页表机制 . . . . .	45
2.6.2	地址转换 . . . . .	46
2.6.3	页目录自映射 . . . . .	47
2.6.4	创建页表 . . . . .	48
2.6.5	地址映射 . . . . .	48
2.6.6	page insert and page remove . . . . .	48
2.7	正确结果展示 . . . . .	50
2.8	实验思考 . . . . .	51
2.9	挑战性任务 . . . . .	51
<b>3</b>	<b>进程与异常</b>	<b>52</b>
3.1	实验目的 . . . . .	52
3.2	进程 . . . . .	52
3.2.1	进程控制块 . . . . .	52
3.2.2	设置进程控制块 . . . . .	54
3.2.3	加载二进制镜像 . . . . .	60
3.2.4	创建进程 . . . . .	63
3.2.5	进程运行与切换 . . . . .	64
3.3	中断与异常 . . . . .	66
3.3.1	异常的分发 . . . . .	66
3.3.2	异常向量组 . . . . .	68
3.3.3	时钟中断 . . . . .	69
3.3.4	进程调度 . . . . .	70
3.4	实验正确结果 . . . . .	70
3.5	实验思考 . . . . .	71



3.6 挑战性任务 . . . . .	71
<b>4 系统调用与 fork . . . . .</b>	<b>72</b>
4.1 实验目的 . . . . .	72
4.2 系统调用 (System Call) . . . . .	72
4.2.1 一探到底, 系统调用的来龙去脉 . . . . .	72
4.2.2 系统调用机制的实现 . . . . .	74
4.2.3 基础系统调用函数 . . . . .	77
4.3 进程间通信机制 (Inter-Process Communication) . . . . .	77
4.4 FORK . . . . .	79
4.4.1 初窥 fork . . . . .	79
4.4.2 fork 的结构 . . . . .	81
4.4.3 返回值的秘密 . . . . .	83
4.4.4 父与子 . . . . .	85
<b>5 文件系统 . . . . .</b>	<b>87</b>
5.1 实验目的 . . . . .	87
5.2 背景知识 . . . . .	87
5.2.1 文件系统 . . . . .	87
5.2.2 磁盘 . . . . .	88
5.3 IDE 磁盘 . . . . .	88
5.3.1 内存映射 I/O . . . . .	89
5.3.2 IDE 磁盘操作 . . . . .	89
5.4 文件系统结构 . . . . .	91
5.4.1 磁盘布局 . . . . .	91
5.4.2 块缓存 . . . . .	93
5.4.3 文件系统详细结构 . . . . .	94
5.5 文件系统服务 . . . . .	96
5.6 正确结果展示 . . . . .	96
5.7 实验思考 . . . . .	97
5.8 实验挑战 . . . . .	97
<b>6 管道与 Shell . . . . .</b>	<b>98</b>
6.1 实验目的 . . . . .	98
6.2 管道 . . . . .	98
6.2.1 初窥管道 . . . . .	98
6.2.2 管道的测试 . . . . .	100
6.2.3 管道的读写 . . . . .	102
6.2.4 管道的竞争 . . . . .	104
6.3 shell . . . . .	107
6.4 实验正确结果 . . . . .	108

6.4.1	管道测试	108
6.4.2	shell 测试	108
6.5	实验思考	108
<b>A</b>	<b>操作系统实验环境</b>	<b>109</b>
A.1	实验目的	109
A.2	了解操作系统实验	109
A.3	操作系统实验工具	110
A.3.1	交叉编译器	110
A.3.2	Linux 系统	111
A.4	实验环境配置	112
A.4.1	Linux 操作系统	112
A.4.2	安装交叉编译器	116
A.4.3	安装仿真器	117
A.5	实验思考	118

# CHAPTER 1

## 内核、BOOT 和 PRINTF

### 1.1 实验目的

1. 掌握操作系统实验所需的基本工具
2. 从操作系统角度理解 MIPS 体系结构
3. 掌握操作系统启动的基本流程

在本章中，我们需要阅读并填写部分代码，使得我们的小操作系统可以正常的运行起来。这一章节的难度较为简单，主要是为了让大家熟悉操作系统实验环境中的各类工具，为后续的实验奠定基础。

### 1.2 操作系统的启动

#### 1.2.1 内核在哪里？

通过《计算机组成原理》课的学习，我们知道计算机是由硬件和软件组成的，仅有一个裸机是什么也干不了的；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，我们需要使用操作系统。而操作系统本身也是一种软件，我们可能会有这样的疑问：我们运行的 Hello World 程序是通过操作系统的机制实现的，那么操作系统这种软件又是如何启动起来的呢？

**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”。

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，从而利用硬件的功能为用户进程提供服务。我们知道一个程序要能够运行，其代码必须能够被 CPU 直接访问，所以不能在磁盘上，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。所以直观上想，内核文件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，把操作系统内核放置在这种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 寻址空间的某个区域，这个是在硬件设计上决定的。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，对于内核文件较大的普通操作系统而言显然是不足够的。
2. 如果操作系统内核在 CPU 加电后直接启动，意味着一个计算机硬件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放在名为“bootloader”的程序中来完成。这样做的好处正对应上述的问题：

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。因此需要存储在非易失性存储器中的硬件启动相关指令不需要很多，能够很容易地保存在 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文件从存放它的存储器（比如磁盘）中读到 RAM 中。既然 bootloader 需要将内核镜像文件加载到内存中，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运行多个操作系统了。
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。需要注意的是这样做并不意味着操作系统不依赖硬件。由于操作系统直接管理着计算机所有的硬件资源，要想操作系统完全独立于处理器架构和硬件平台显然是不切实际的。然而使用 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。

### 1.2.2 Bootloader

从操作系统的角度看，boot loader 的总目标就是正确地调用内核来执行。另外，由于 boot loader 的实现依赖于 cpu 的体系结构，因此大多数 boot loader 都分为 stage1 和 stage2 两大部分。

在 stage 1 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 stage 1 直接运行在存放 boot loader 的存储设备上（比如 FLASH）。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，比如有些 flash 程序不可写，即使程序可写的 flash 也有存储空间限制。这就是为什么需要 stage 2 的原因。stage 1 除了初始化基本的硬件设备以外，会为加载 stage 2 准备 RAM 空间，然后将 stage 2 的代码复制到 RAM 空间，并且设置堆栈，最后跳转到 stage 2 的入口函数。

stage 2 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，然后将内核镜像文件从存储器读到 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1.1所示。

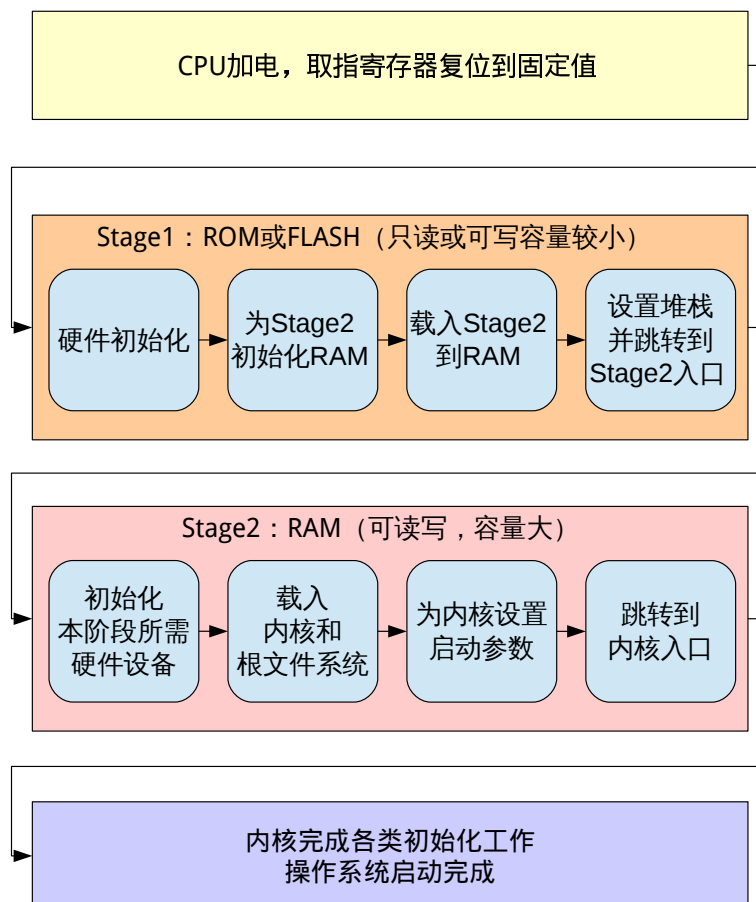


Figure 1.1: 启动的基本步骤

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 1.2.2** GRUB(Grand Unified Bootloader) 是 GNU 项目的一个多操作系统启动程序。简单的说，就是可以用于在有多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中的内核镜像文件加载到内存中，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 stage 1 的工作，而 stage 2 的工作则是在 Grub 中完成的。

**Note 1.2.3** bootloader 有两种操作模式：启动加载模式和下载模式。对于普通用户而言，bootloader 只运行在启动加载模式，就是我们之前讲解的过程。而下载模式仅仅对于开发人员有意义，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上载到内存的。

### 1.2.3 gxemul 中的启动流程

从前面的分析，我们可以看到，操作系统的启动是一个非常复杂的过程。不过，幸运的是，由于我们的小操作系统的目标是在 gxemul 仿真器上运行，这个过程被大大简化了。gxemul 仿真器支持直接加载 elf 格式的内核，也就是说，gxemul 已经提供了 bootloader 全部功能。我们的小操作系统不需要再实现 bootloader 的功能了。换句话说，你可以假定，从我们的小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 C 环境了。全局变量、函数调用等等 C 语言运行所需的功能已经可以正常使用了。

**Note 1.2.4** 如果你以前对于操作系统的概念仅仅停留在很表面的层次上，那么这里你也许会有所疑惑，为什么我们这里要说“正常的 C 环境”？难道还能有“不正常的 C 环境”？我们来举一个例子说明一下：假定我们刚加电，CPU 开始从 ROM 上取指。为了简化，我们假定这台机器上没有 BIOS(Basic Input Output System), bootloader 被直接烧在了 ROM 中 (很多嵌入式环境就是这样做的)。这时，由于内存没有被初始化，整个 bootloader 程序尚处于 ROM 中。程序中的全局变量也仍被储存在 ROM 上。而 ROM 是只读的，所以任何对于全局变量的赋值操作都是不被允许的。可见，此时我们尚不能正常使用 C 语言的一些特性。而当内存被初始化，bootloader 将后续代码载入到内存中后，位于内存中的代码便能完整地使用 C 语言的各类功能了。所以说，内存中的代码拥有了一个正常的 C 环境。

gxemul 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。启动完毕:) 整个启动过程非常简单。这里要注意，之所以简单还有一个原因就在于 gxemul 本身是仿真器，是一种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小操作系统内核了。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 ssh——远程连接到服务器

课程提供的整个实验环境位于远程的一台虚拟机上，最终的成果也需要通过这台机器进行提交。因此，我们面临的第一个问题就是：如何操作这台机器？为了操作这台机器，我们需要通过 SSH 协议远程连接上它。

一般在 Linux 或 Mac 等 Unix 类环境中都会附带 ssh 客户端。我们只需要打开终端，然后执行如下指令

```
1  # username 处填写你的用户名, ip 处填写远程主机的地址
2  $ ssh username@ip
3  # 之后等候片刻会要求你输入密码, 输入的密码不会被显示在屏幕上, 输入完成后按回车即可
4  # 链接后会显示一些欢迎信息, 下面是欢迎信息的一个例子
5  Welcome to Ubuntu 12.04.5 LTS (GNU/Linux 3.13.0-32-generic i686)
6
7  * Documentation:  https://help.ubuntu.com/
8
9  System information as of Tue Aug 11 09:55:40 CST 2015
10 System load:   0.0              Processes:      118
11 Usage of /:   8.7% of 145.55GB  Users logged in: 0
12 Memory usage: 6%              IP address for eth0: 0.0.0.0
13 Swap usage:   0%
14
15 Graph this data and manage this system at:
16   https://landscape.canonical.com/
17
18 0 packages can be updated.
19 0 updates are security updates.
20 # 欢迎信息后会出现命令提示符, 等待你输入命令。
```

Windows 平台下一般不自带 ssh，需要下载第三方软件实现这个功能。这里我们使用 PuTTY 这个小工具实现 ssh 连接。当然，如果你愿意花时间问问度娘（或 Google）的话，你会发现也有很多功能更为强大 ssh 客户端软件。不过，鉴于 PuTTY 小巧简洁且是一个在 MIT 协议下开源的开源软件，我们这里只介绍 PuTTY 的用法。

PuTTY 主界面的截图如图1.2所示。打开 PuTTY 后，在主机名 (Host Name) 处填写主机地址 (ip)，连接方式 (Connection type) 处选择 SSH，端口 (Port) 使用默认值即可。之后点击连接 (Open)。PuTTY 会自动打开一个窗口，按照提示输入用户名和密码即可进入远程主机。



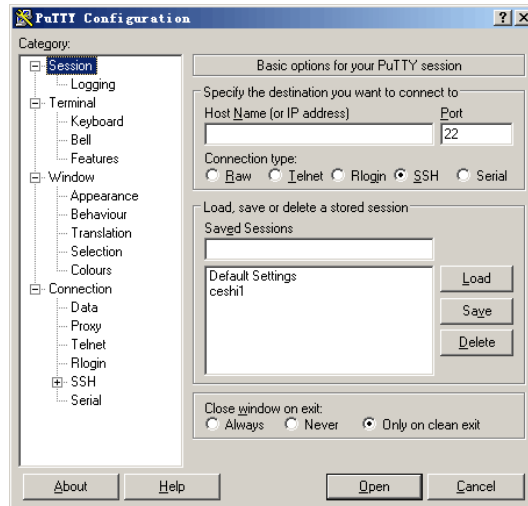


Figure 1.2: PuTTY 界面

**Note 1.3.1** SSH 是 Secure Shell 的缩写。它是一种用于建立安全的远程连接的网络协议。在 Unix 类系统上被广泛采用。除了连接到远程网络，目前 SSH 还有一种较为有趣的用法。当你既想用 Windows，又有时需要 Unix 环境时，可以利用 Windows8 及以上版本自带的 Hyper-V 开启一个 Linux 虚拟机（不必开启图形界面）。之后通过 SSH 客户端连接到本机上的 Linux 虚拟机上，即可获得一个 Unix 环境。甚至可以开启 X11 转发，即可在 Windows 上开启一些 Linux 上的带图形界面的程序，十分方便。

### 1.3.2 vim 或 nano——阅读并修改代码

连接到虚拟机以后，我们就可以开始动手阅读并修改代码了。但当你看到光秃秃的命令行的时候，是否感到一阵心慌？到底该如何打开并编辑一个文件呢？我们先从一个简易的工具入手：nano。

nano 的主界面如图1.3所示。所有基本的操作都被罗列在下面。其中界面上显示的向上的尖状符号 (^) 代表 **Ctrl**，后面的字母就代表相应的快捷键。例如保存是 **Ctrl+O**。nano 较为容易上手，但功能相对有限。如果你需要更为强大的功能，可以使用 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。关于如何入门 Vim，网络上的资料实在太多，这里就不再赘述了。推荐一篇质量极高的 Vim 教程《简明 Vim 练级攻略》(<http://coolshell.cn/articles/5426.html>)。只需要十多分钟的阅读，你就可以掌握 Vim 的所有基本操作。当然，想要完全掌握它需要相当长的时间去练习。不过，如果你只是想把 Vim 当成记事本用的话，那么几分钟的学习足矣。



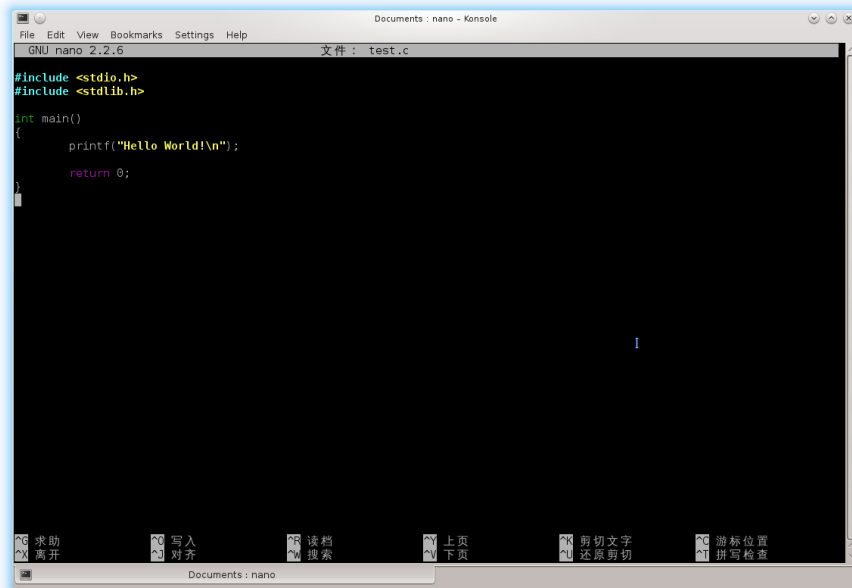


Figure 1.3: nano 界面

### 1.3.3 Makefile——内核代码的地图

当我们学会了 Vim，想要翻开代码大干一番的时候，一个新的问题又迎面而来了：这堆代码应当从何读起？答曰：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。这时，有同学又要问了：什么是 make？什么又是 Makefile 呢？make 工具一般用于维护工程。它可以根据时间戳自动判断项目的哪些部分是需要重新编译的，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。Makefile 类似于大家以前接触过的 VC 工程文件。只不过不像 VC 那样有图形界面，而是直接用类似脚本的方式实现的。

**Note 1.3.2** 相较于 VC 工程而言，Makefile 具有更高的**灵活性**（当然，高灵活性的代价就是学习成本会有所提升，这是必然的），可以方便地管理大型的项目。而且 Makefile 理论上支持**任意的语言**，只要其编译器可以通过 shell 命令来调用。当你的项目可能会混合多种语言，有着复杂的构建流程的时候，Makefile 便能展现出它真正的威力来。

为了使你更为清晰地了解 Makefile 的基本概念，我们来写一个简单的 Makefile。假设我们手头有一个 Hello World 程序需要编译。我们来为它写一个简易的 Makefile。让我们从头开始，如果我没有 Makefile，直接动手编译这个程序，我们需要下面这样一个指令

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

那么，如果我们想把它写成 Makefile，我们应该怎么办呢？makefile 最基本的格式是这样的

```

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

```

其中, target 是我们构建 (Build) 的目标, 而 dependencies 是构建该目标所需的其它文件或其他目标。之后是构建出该目标所需执行的指令。有一点尤为需要注意: **每一个指令 (command) 之前必须有一个 TAB**。这里必须使用 TAB 而不能是空格, 否则 make 会报错。

我们的简易的 Makefile 可以写成如下的样子, 之后执行 make 即可产生 hello\_world 这个可执行文件。

```

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

```

理解了 Makefile 最基本的概念后, 我们来看一下我们的小操作系统的最顶层的 Makefile。由于 Makefile 是用于指导程序如何被构建的, 因此, 通过阅读 Makefile, 我们就可以理解源代码被构建为可执行文件的过程。这一过程可以给我们一些阅读代码的提示, 可以说, Makefile 就像源代码的地图, 告诉你源代码是如何一步一步成为最终的可执行文件的。代码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。

Listing 1: 顶层 Makefile

```

1  # Main makefile
2  #
3  # Copyright (C) 2007 Beihang University
4  # Written by Zhu Like ( zlike@cse.buaa.edu.cn )
5  #
6
7  drivers_dir      := drivers
8  boot_dir         := boot
9  init_dir         := init
10 lib_dir          := lib
11 tools_dir        := tools
12 vmlinux_elf       := gxemul/vmlinux
13
14 link_script      := $(tools_dir)/scse0_3.lds
15
16 modules          := boot drivers init lib
17 objects          := $(boot_dir)/start.o \
18                    $(init_dir)/main.o \
19                    $(init_dir)/init.o \
20                    $(drivers_dir)/gxconsole/console.o \
21                    $(lib_dir)/*.o
22
23 .PHONY: all $(modules) clean
24
25 all: $(modules) vmlinux
26
27 vmlinux: $(modules)

```

```

28     $(LD) -o $(vmlinux_elf) -N -T $(link_script) $(objects)
29
30 $(modules):
31     $(MAKE) --directory=$@
32
33 clean:
34     for d in $(modules); \
35     do \
36         $(MAKE) --directory=$$d clean; \
37     done; \
38     rm -rf *.o *~ $(vmlinux_elf)
39
40 include include.mk

```

如果你以前没有接触过 Makefile 的话，突然看到这份 40 行的 Makefile 可能会有些无奈，完全看不懂啊。不必着急，我们来一行一行地解读它。前 6 行是注释，你懂得。7 ~ 21 行定义了一些变量，包括各个子目录的相对路径，最终的可执行文件的路径 (`vmlinux_elf`)，linker script 的位置 (`link_script`)。值得注意的两个是 `modules` 定义了内核所包含的所有模块，而 `objects` 则表示要编译出内核所依赖的所有 `.o` 文件。17 到 21 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。这种写法一般用于提高文件的可读性。可以把本该写在同一行的东西分布在多行中，使得文件更容易被人类阅读。

**Note 1.3.3** linker script 是用于指导连接器将多个 `.o` 文件连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。

23 行的 `.PHONY` 表明列在其后的规则不受修改时间的约束。也就是说，一旦该规则被调用，一定保证它被执行。第 25 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所有的模块以及 `vmlinux`。那么 `vmlinux` 是如何被构建的呢？紧接着的 27 行定义了，`vmlinux` 的构建依赖于所有的模块。在构建完所有模块后，将执行第 28 行的指令来产生 `vmlinux`。我们可以看到，第 28 行调用了连接器将之前构建各模块产生的所有 `.o` 文件在 linker script 的指导下连接到一起，产生最终的 `vmlinux` 可执行文件。第 30 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 33 到 38 行定义了如何清理所有被构建出来的文件。

**Note 1.3.4** 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，我们会发现还有几个关键的变量没有定义。是的，就是 `LD`、`MAKE` 等出现在编译指令中的变量。紧接着我们看到了第 40 行有一条 `include` 命令。看来，这个顶层的 Makefile 还引用了其他的東西。显然这些未定义的变量，是被定义在了这个被 `include` 的文件中。被引用的文件如代码 2 所示。

Listing 2: include.mk

```

1  # Common includes in Makefile
2  #
3  # Copyright (C) 2007 Beihang University
4  # Written By Zhu Like ( zlike@cse.buaa.edu.cn )
5
6
7  CROSS_COMPILE := /opt/eldk/usr/bin/mips_4KC-
8  CC             := $(CROSS_COMPILE)gcc
9  CFLAGS         := -O -G 0 -mno-abicalls -fno-builtin -Wa,-xgot -Wall -fPIC -Werror
10 LD             := $(CROSS_COMPILE)ld

```

在该文件中,我们看到了一个非常熟悉的关键词——Cross Compile(交叉编译)。不难看出,这里的 CROSS\_COMPILE 变量是在定义编译和连接等指令的前缀,或者说是交叉编译器的具体位置。例如,LD 最终调用的指令是“/opt/eldk/usr/bin/mips\_4KC-ld”。通过修改该变量,就可以方便地设定交叉编译工具链的位置。

**Exercise 1.1** 请修改 include.mk 文件,使交叉编译器的路径正确。之后执行 make 指令,如果配置一切正确,则会在 gxemul 目录下生成 vmlinux 的内核文件。 ■

至此,我们就可以大致掌握阅读 Makefile 的方法了。善于运用 make 的功能可以给你带来很多惊喜哦:) 提示:可以试着使用一下 make clean。如果你觉得每次用 gxemul 运行内核都需要打很长的指令这件事很麻烦,那么可以尝试在 Makefile 中添加运行内核这一功能,使得通过 make 就能自动运行内核。

### 1.3.4 ELF——深入探究编译与链接

如果你已经尝试过运行内核,那么你会发现它是根本运行不起来的。因为我们还有一些重要的步骤没有做。但是在做这些之前,我们不得不补充一些重要的,但又有些琐碎的知识。在这里,我们将掀开可执行文件的神秘面纱,进一步去了解一段代码是如何从编译一步一步变成一个可执行文件以及可执行文件又是如何被执行的。

在一切开始之前,请你先泡好一杯茶,慢慢地、耐心地读下去。这一部分的知识对于后面十分重要,但又十分冗长。我们会尽量说得轻松活泼一些,但由于知识本身的琐碎以及不连贯,所以阅读体验并不会很好。请务必坚持看完:)

Listing 3: 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？<sup>1</sup> 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？我们来进去看一看 stdio.h 里到底有什么。

Listing 4: 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4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。

```

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;

```

<sup>1</sup>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。

```

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 `.o` 目标文件。我们对其进行反汇编<sup>2</sup>，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>
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。根据我们在《计算机组成》课程中学习 MIPS 跳转指令的经验，`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` 进行连接，也就是正常地编译出可执行文件。然后，再用 `objdump` 进行反汇编。

```

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>:

```

<sup>2</sup>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。

```

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 000000000400425 <_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）。当你看到这段代码的时候，心头一定有一大群草泥马呼啸而过。什么鬼！我们原本那只可爱的 Hello World 怎么变成了这副鬼样子！

别急，我们还是只把注意力放在主函数中，这一次，我们可以看到，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。虽然高不清楚那货是什么，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**连接 (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` 并未连接到程序中，它的状态与我们利用 `-c`



选项产生的 `hello.o` 相仿，都还处于未连接的状态。而在编译的最后，连接器 (Linker) 会将所有的目标文件连接在一起，将之前未填写的地址等信息填上，形成最终的可执行文件，这就是连接的过程。

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` 文件以及函数库连接在一起，形成最终的可执行文件。整个过程如图1.4所示。

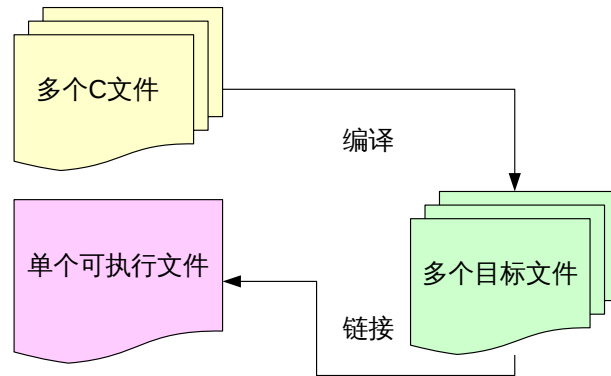


Figure 1.4: 编译、连接的过程

接下来，我们提出我们的下一个问题：**连接器通过哪些信息来连接多个目标文件呢？**答案就在于在目标文件（也就是我们通过 `-c` 选项生成的 `.o` 文件）。在目标文件中，记录了代码各个段的具体信息。连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。

我们最终生成的内核是 ELF 格式的，被模拟器载入到内核中。因此，我们暂且只关注 ELF 是如何被载入到内核中，并且被执行的，而不再关心具体的连接细节。ELF 中有两个相似却不同的概念 `segment` 和 `section`。`section` 记录了程序的代码段、数据段等各个段的内容，主要是连接器在连接的过程中需要使用。而 `segment` 则记录了每一段数据（包括代码等内容）需要被载入到哪里，记录了用于指导应用程序加载的各类信息。

我们不妨来看一下，我们之前那个 `hello world` 程序的各个 `segment` 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。

```

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          Flags   Align
8   LOAD          0x0000000000000000 0x0000000000400000 0x0000000000400000
9   LOAD          0x000000000000b33c0 0x000000000000b33c0 R E    200000
10  LOAD          0x000000000000b4000 0x0000000000006b4000 0x0000000000006b4000
11  LOAD          0x00000000000001cd0 0x00000000000003f48 RW     200000
12  NOTE          0x00000000000000158 0x000000000000400158 0x000000000000400158
13  NOTE          0x00000000000000044 0x00000000000000044 R      4
14  TLS           0x000000000000b4000 0x0000000000006b4000 0x0000000000006b4000
  
```



```

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 .rela.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。

**Note 1.3.5** 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，Gxemul 仿真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。我们的内核之所以不能够正常运行，显然是因为我们内核所处的地址是不正确的。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，我们又发现了两个重要的问题。

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掉这两个问题。

### 1.3.5 MIPS 内存布局——寻找内核的正确位置

在这一节中，我们来解决关于内核应该被放在何处的的问题。在 32 位的 MIPS CPU 中，程序地址空间会被划分为 4 个大区域。如图1.5所示。

从硬件角度讲，这四个区域的情况如下：

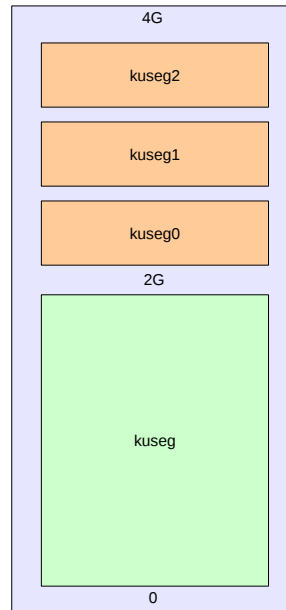


Figure 1.5: MIPS 内存布局

1. User Space(kuseg) 0x00000000-0x7FFFFFFF(2G): 这些是用户模式下可用的地址。在使用这些地址的时候, 程序会通过 MMU 映射到实际的物理地址上。
2. Kernel Space Unmapped Cached(kseg0) 0x80000000-0x9FFFFFFF(512MB): 只需要将地址的高 3 位清零, 这些地址就被转换为物理地址。也就是说, 逻辑地址到物理地址的映射关系是硬件直接确定的, 不通过 MMU。因为转换很简单, 我们常常把这些地址成为“无需转换的”。一般情况下, 都是通过 cache 对这段区域的地址进行访问。
3. Kernel Space Unmapped Uncached(kseg1) 0xA0000000-0xBFFFFFFF(512MB): 这一段地址的转换规则与前者相似, 通过将高 3 位清零的方法将地址映射为相应的物理地址, 然后映射到物理内存中 512MB 大小的低字段。需要注意的是, kseg1 不通过 cache 进行存取。
4. Kernel Space Mapped Cached(kseg2) 0xC0000000-0xFFFFFFFF(1GB): 这段地址只能在内核态下使用并且需要 MMU 的转换。

看到这里, 你也许又蔫儿了, 还是完全不知道该把内核放在哪里呀! 这里, 我们再提供一个提示: 需要通过 MMU 映射访问的地址得在建立虚拟内存机制后才能正常使用, 是由操作系统所管理的。因此, 在载入内核时, 我们只能选用不需要通过 MMU 的内存空间, 因为此时尚未建立虚存机制。好了, 满足这一条件的只有 kseg0 和 kseg1 了。而 kseg1 是不经过 cache 的, 一般用于访问外部设备。所以, 我们的内核只能放在 seg0 了。

那么具体放在哪里呢? 这时, 我们就需要仔细阅读代码了。在 include/mmu.h 里有我们的小操作系统内核完整的内存布局图 (代码5所示), 在之后的实验中, 善用它可以带来意料之外的惊喜。

Listing 5: include/mmu.h 中的内存布局图

```

1  /*
2  o    4G -----> +-----+-----0x100000000
3  o    /      ...      / kseg3
4  o    +-----+-----0xe000 0000
5  o    /      ...      / kseg2
6  o    +-----+-----0xc000 0000
7  o    /  Interrupts & Exception / kseg1
8  o    +-----+-----0xa000 0000
9  o    /    Invalid memory      / /\
10 o    +-----+-----Physics Memory Max
11 o    /      ...      / kseg0
12 o  VPT,KSTACKTOP-----> +-----+-----0x8040 0000-----end
13 o    /    Kernel Stack      / / KSTKSIZE      /\
14 o    +-----+-----+-----+
15 o    /    Kernel Text      / /      PDMAP
16 o    KERNBASE -----> +-----+-----0x8001 0000 /
17 o    /  Interrupts & Exception / \/\
18 o    ULIM -----> +-----+-----0x8000 0000-----
19 o    /    User VPT      / PDMAP      /\
20 o    UVPT -----> +-----+-----0x7fc0 0000 /
21 o    /    PAGES      / PDMAP      /
22 o    UPAGES -----> +-----+-----0x7f80 0000 /
23 o    /    ENV$      / PDMAP      /
24 o    UTOP,UENVS -----> +-----+-----0x7f40 0000 /
25 o    UXSTACKTOP -/      /    user exception stack / BY2PG      /
26 o    +-----+-----+-----0x7f3f f000 /
27 o    /    Invalid memory      / BY2PG      /
28 o    USTACKTOP -----> +-----+-----0x7f3f e000 /
29 o    /    normal user stack      / BY2PG      /
30 o    +-----+-----+-----0x7f3f d000 /
31 a    /      /
32 a    ~~~~~
33 a    .      .      /
34 a    .      .      kuseg
35 a    .      .      /
36 a    /~~~~~+
37 a    /      /
38 o    UTEXT -----> +-----+
39 o    /      /      2 * PDMAP      \/\
40 a    0 -----> +-----+
41 o
42 */

```

相信聪明的你已经发现了内核的正确位置了吧？

### 1.3.6 Linker Script——控制加载地址

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 OK 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段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连接器产生的（它将多个目标文件连接产生最终可执行文件）。因此，我们所需要做的，就是控制连接器的连接过程。

接下来，我们就要引入一个神奇的东西：Linker Script。连接器的设计者们在设计连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，怎样做才能增加连接器的通用性，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 section 应该如何映射到 segment，以及各个 segment 应该被加载到的位置。下面的指令可以输出默认的连接脚本，你可以在自己的机器上尝试这一条指令：

```
1 ld --verbose
```

这里，我们再补充一下关于 ELF 文件中 section 的概念。在链接过程中，目标文件被看成 section 的集合，并使用 section header table 来描述各个 section 的组织。换句话说，section 记录了在连接过程中所需要的必要信息。其中最为重要的三个段为 .text、.data、.bss。这三种段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6所示）。观察其运行结果与 ELF 文件中记录的 .text、.data 和 .bss 段相关信息之间的关系。

Listing 6: 用于输出各 section 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<sup>3</sup>。

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

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

<sup>3</sup>在不同机器上运行，结果也许会有一些的差异

```

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` 段包含了可执行文件中的代码，`.data` 段包含了需要被初始化的全局变量和静态变量，而`.bss` 段包含了无需初始化的全局变量和静态变量

接下来，我们通过 Linker Script 来尝试调整各段的位置。这里，我们选用 GNU LD 官方帮助文档上的例子 (<https://www.sourceware.org/binutils/docs/Ld/Simple-Example.html#Simple-Example>) 该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，它根据输出段的大小增长。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 section 的其实地址。“\*”是一个通配符，匹配所有的相应的段。例如“`.bss:{*(.bss)}`”表示将所有输入文件中的`.bss` 段（右边的`.bss`）都放到输出的`.bss` 段（左边的`.bss`）中。为了能够编译通过（这个脚本过于简单，难以用于连接真正的程序），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 section 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           00010024  001024  00000a 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，生成的程序各个 section 的位置就被调整到了我们所指定的地址上。segment 是由 section 组合而成的，section 的地址被调整了，那么最终 segment 的地址也会相应地被调整。至此，我们就了解了如何通过 lds 文件控制各段被加载到的位置。

**Exercise 1.2** 填写 tools/scse0\_3.lds 中空缺的部分，将内核调整到正确的位置上。■

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称为 entry point，我们在 linker script 中可以通过 ENTRY(function name) 指令来设置程序入口。linker 中程序入口的设置方法有以下五种：

1. 使用 ld 命令时，通过参数 “-e” 设置
2. 在 linker script 中使用 ENTRY() 指令指定了程序入口
3. 如果定义 start，则 start 就是程序入口
4. .text 段的第一个字节
5. 地址 0 处

在我们的实验中，采用了其中的第二种方式指定了程序入口。

## 1.4 MIPS 汇编与 C 语言

在这一节中，我们将简单介绍 MIPS 汇编，以及常见的 C 语言语法与汇编的对应关系。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。为了能够实现这些，我们需要了解 C 语言与汇编之间千丝万缕的联系。

我们以代码 7 为例，介绍典型的 C 语言中的语句对应的汇编代码。

Listing 7: 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 }
```

### 1.4.1 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7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<sup>4</sup>，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 代码对应会原来的 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。

### 1.4.2 函数调用

这里需要区分函数的调用方和被调用方来分别分析。我们选用样例程序中的 `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。

我们还是先将整个函数调用过程用高级语言来表示一下。

<sup>4</sup>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  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？

```

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。在经过了《数据结构》这门课程的学习后，你一定发现了，递归的过程和栈这种数据结构有着惊人的相似性。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。

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<sup>5</sup>。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，编译器真的为我们维护了一个栈。

**Note 1.4.1** `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

<sup>5</sup>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



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些什么? 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<sup>6</sup>

```

1  lw      $2,40($fp)      # v0 = n;
2  addiu   $2,$2,-1        # v0 = a0 - 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;
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。

**Exercise 1.3** 完成 boot/start.S 中空缺的部分。设置栈指针, 跳转到 main 函数。使用 `gxemul -E testmips -C R3000 -M 64 elf-file` 运行 (其中 elf-file 是你编译生成的 vmlinux 文件的路径)。

在调用 main 函数之前, 我们需要将 sp 寄存器设置到内核栈空间的位置上。具体的地址可以从 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.S 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。

### 1.4.3 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1.1所示。

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<sup>7</sup>。对

<sup>6</sup>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

<sup>7</sup>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。

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	kt0-kt1	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。

Table 1.1: MIPS 通用寄存器

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 Gxemul 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。

## 1.5 实战 `printf`

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 `printf` 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。

要弄懂系统如何将一个字符输出到终端当中, 需要阅读以下三个文件: `lib/printf.c`, `lib/print.c` 和 `driver/gxemul/console.c`。相信阅读这些代码对于聪明的你来说, 完全不是问题, 我们就不再赘述了。

**Exercise 1.4** 阅读相关代码，并补全 lib/print.c 中 lp\_Print() 函数中缺失的部分来实现字符输出。 ■

当你刚看到 lp\_Print() 的代码时，也许会手忙脚乱。不知如何读起，更不清楚该如何补全它。这主要是由于，我们对于 printf 要实现的具体功能没有完整的认识。这里我们给出一些提示，printf 的 format specifier(也就是它的第一个参数) 的格式如下：

`%[flags][width][.precision][length]specifier`

更详尽的描述，请大家查询 C 语言标准库的相关文档<sup>8</sup>。完整的描述较为复杂，我们只需实现我们所有能够支持的部分即可。例如，内核中是无法使用浮点数的，因此所有和浮点输出相关的东西都不必要实现。

## 1.6 Git——轻松维护和提交代码

在完成了这次实验后，你刚刚伸了个懒腰。突然，一个问题浮现在了你的脑海中：如何提交刚刚完成的实验代码呢？整个实验的代码采用了 git 版本控制系统进行管理，所有与代码管理相关的操作全部通过 git 完成。在本章最后的部分，我们就来了解一下 git 相关的内容。

### 1.6.1 手工的版本控制

最原始的版本控制是纯手工的版本控制：修改文件，保存文件副本。有时候偷懒省事，保存副本时命名比较随意，时间长了就不知道哪个是新的，哪个是老的，即使知道新旧，可能也不知道每个版本是什么内容，相对上一版作了什么修改了，当几个版本过去后，很可能就是下面的样子了：

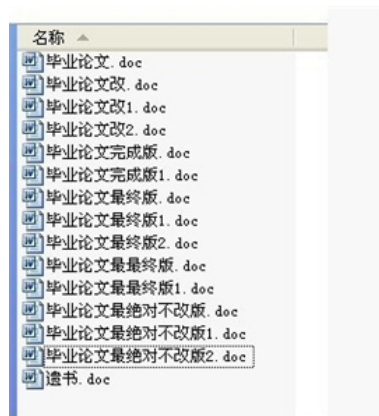


Figure 1.6: 手工版本控制效果图

当然，有些时候，我们不仅是一个人写论文，很可能是多个人同时写一篇 paper。

<sup>8</sup>推荐 cplusplus，这个网站给出了很多的样例 <http://www.cplusplus.com/reference/cstdio/printf/>

分工，制定计划，埋头苦干，看起来一切都井然有序，后来却只会让人蛋疼不已。本质原因在于每个人都会对书的内容进行改动，结果最后成了这样的情形：我把我修订的最新版电子版发邮件给他，然后，我继续修改书的内容。一天后，他再把电子书传给我。每到此时，我就必须想清楚，发给他之后到我收到他的文件期间，我在哪里作了哪些改动，还得把我的改动和他的部分合并，真困难。

到这时我们发现一个无法避免的事实：如果每一次小小的改动都要通知对方，那么一些错误的改动将会令我们付出很大的代价：一个错误的改动要频繁在两方同时通知纠正。但如果一次性改动大幅度的内容，尤其是在不连续的段落中删删改改，将变成只有阅读完整篇论文才能知道对方在哪里改动过，才能把两个人的劳动成果合并。论文只有 10 页时还可以接受，但如果变成 20 页，50 页，100 页呢？

后来我们就想如果能有这样一个软件：

- 自动帮我记录每次文件的改动，而且最好是有后悔药的功能，改错了一个东西，我可以轻松撤销。
- 还得有多人协作编辑不费力的好处，有着简洁的指令与操作。
- 最好能像时光机一样穿越回以前，而且不但能穿越回去，还能在不满意的时候穿越回来！
- 如果想查看某次改动，只需要在软件里瞄一眼就可以。

那岂不是十分方便！

版本控制系统就是这样一种神奇的系统。而 git，则是目前世界上最先进的分布式版本控制系统，没有之一。

**Note 1.6.1** 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。

### 1.6.2 Git 基础指引

看过那个小故事后，相信大家对 git 这个世界上目前最先进的分布式版本控制系统已经产生了很强的兴趣。简单地说，Git 究竟是怎样的一个系统呢？别着急，我们慢慢来。

先从 git 的最基础的指令讲起

```
1 $ git init
```

使用命令

```
1 $ sudo mkdir learnGit
```

使用 `cd learnGit` 进入后，执行 `git init` 创建新的 git 仓库。这时候细心的同学会发现在我们的新文件夹下其实已经有东西了，我们可以使用

```
1 $ ls -a
```

来观察，我们看到我们的文件夹下多出一个名叫.git 的目录，这个隐藏的.git 目录就是 Git 版本库，更多时候我们称之为仓库 (repository)。仓库是用于跟踪管理版本库的，在我们的实验中不会对.git 文件夹下的文件进行任何操作，所以不要对该文件夹中的任何文件进行任何手工修改！

在刚刚我们提到的 `init` 命令完成之后，我们就已经有了一个仓库。我们之前所建立的 `learnGit` 这个文件夹就是 Git 里的工作区。目前我们的工作区除了包含一个隐藏的.git 版本库目录外空无一物。

**Note 1.6.2** 在我们的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 `1406xxxx-lab` 的版本库，包含了 `lab1` 的实验内容。

我们的工作区现在空荡荡的，我们来为它加点料。我们使用下面的指令创建一个新的文本文件

```
1 | $ echo "BUAA_OSLAB" > readme.txt
```

这一步只是创建一个文本文件而已，为了将它添加到版本库，我们需要执行下面的命令

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

注意，到这里还没有结束，你可能会想，那我既然都把 `readme.txt` 加入了，难道不是已经提交到版本库了吗？但事实就是这样，Git——同其他大多数版本控制系统一样，`add` 之后需要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的话，`git commit` 后会弹出一个说明窗口，如下所示

```
1 | GNU nano 2.2.6    文件： /home/13061193/13061193-lab/.git/COMMIT_EDITMSG
2 |
3 | Notes to test.
4 | # 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
5 | # 说明将会终止提交。
6 | # 位于分支 master
7 | # 您的分支与上游分支 'origin/master' 一致。
8 | #
9 | # 要提交的变更：
10 | #     修改：      readme.txt
11 | #
12 |
13 | [ 已读取 9 行 ]
14 | ^G 求助      ^O 写入      ^R 读档      ^Y 上页      ^K 剪切文字  ^C 光标位置
15 | ^X 离开      ^J 对齐      ^W 搜索      ^V 下页      ^U 还原剪切  ^T 拼写检查
```

在上面里书写的 **Notes to test.** 是我们本次提交所附加的说明。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。而且在添加评论之后，可以按提示按键来成功保存。

**Note 1.6.3** 初学者一般不太重视 git commit 内容的有效性，总是使用无意义的字符串作为说明提交。但以后你可能会发现自己写了一个自己看得懂，别人也能看得懂提交说明是多么庆幸。所以尽量让你的每次提交显得有意义，比如“fixed a bug in ...”这样的描述，顺便推荐一条命令：git commit -amend，这条命令可以重新书写你最后一次 commit 的说明。

这样窗口提交的方式比较繁琐，我们可以采取一种较为简洁的方式

```
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指令

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

如果我们在提交后看到类似提示就说明我们提交成功了

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

从我们本次提交中我们可以得到以下信息，可能现在你还不能完全理解这些信息代表的意义，但是没关系，之后我们会讲解

- 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

但是在我们实验中，第一次提交可不会这么一帆风顺，我们第一次提交往往会出现下面的提示

```
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.
```

相信大家从第一句也能推测出，这是要求我们设置提交者身份的。我们设置身份有什么作用呢？别急，等你设置成功了我们再详谈。

**Note 1.6.4** 从上面我们也知道了，我们可以用

```
git config --global user.email "you@example.com"
```

```
git config --global user.name "Your Name"
```

这两条命令设置我们的名字和邮箱，在我们的实验中对这两个没有什么要求，大家随性设置就好，给个示例：

```
git config --global user.email "qianlxc@126.com"
```



```
git config --global user.name "Qian"
```

现在你已设置了提交者的信息，那么做一下这个小练习来快速上手 Git 的使用吧

- Exercise 1.5**
- 在 `/home/1406xxxx/` 目录下创建一个名为 `README.txt` 的文件。这时使用 `git status > Untracked.txt`。
  - 在 `README.txt` 文件中随便写点什么，然后使用刚刚学到的 `add` 命令，再使用 `git status > Stage.txt`。
  - 之后使用上面学到的 Git 提交有关的知识把 `README.txt` 提交，并在提交说明里写入自己的学号。
  - 使用 `cat Untracked.txt` 和 `cat Stage.txt`，对比一下两次的结果，体会一下 `README.txt` 两次所处位置的不同。
  - 修改 `README.txt` 文件，再使用 `git status > Modified.txt`。
  - 使用 `cat Modified.txt`，观察它和第一次 `add` 之前的 `status` 一样吗，思考一下为什么？

**Note 1.6.5** `git status` 是一个查看当前文件状态的有效指令，而 `git log` 则是提交日志，每 `commit` 一次，Git 会在提交日志中记录一次。`git log` 将在我们后面乘坐时光机时发挥很大的作用。

相信你做过上述实验后，心里还是会有些疑惑，没关系，我们来一起看一下我们刚才得到的 `Untracked.txt`，`Stage.txt` 和 `Modified.txt` 的内容

```
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  nothing added to commit but untracked files present (use "git add" to track)
9
10 Stage.txt 的内容如下
11
12 # On branch master
13 # Changes to be committed:
14 #   (use "git reset HEAD <file>..." to unstage)
15 #
16 #       new file:   README.txt
17 #
18
19 Modified.txt 的内容如下
20
21 # On branch master
22 # Changes not staged for commit:
```

```
23 # (use "git add <file>..." to update what will be committed)
24 # (use "git checkout -- <file>..." to discard changes in working directory)
25 #
26 #       modified:   README.txt
27 #
28 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中第 2 行是：Untracked files，而第二个文本文件中第二行内容是：Changes to be committed，而第三个则是 Changes not staged for commit。这三种不同的提示意味着什么，需要你通过后面的学习找到答案，答案就在不远处。

我们开始时已经介绍了 Git 中的工作区的概念，接下来的内容就是 Git 中的最核心的概念，为了能自如运用 Git 中的命令，你一定要仔细学习。

### 1.6.3 Git 文件四状态

首先对于任何一个文件，在 Git 内都只有四种状态：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

**Note 1.6.6** 关于刚才的 exercise 中的思考，实际上是因为 git add 指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。我们现在只需要记住：新建文件后要 git add，修改文件后也需要 git add。

我们使用一张图来说明文件的四种状态的转换关系

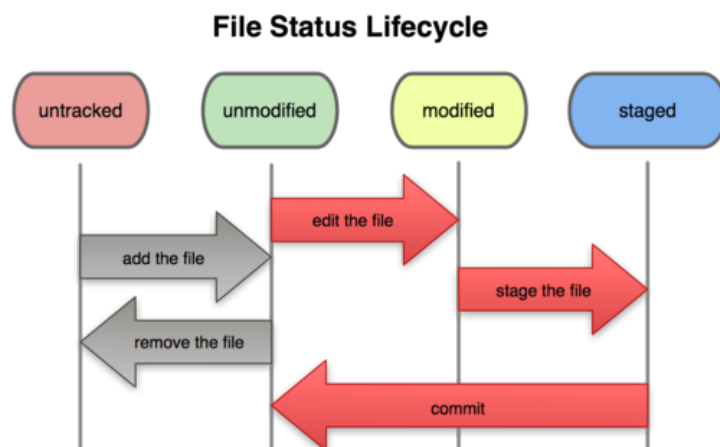


Figure 1.7: Git 中的四种状态转换关系



**Exercise 1.6** 仔细看看这张图，思考一下红箭头里的 add the file、stage the file 和 commit 分别对应的是 Git 里的哪些命令呢？

看到这里，相信你对 Git 的设计有了初步的认识。下一步我们就来深入理解一下 Git 里的一些机制，从而让我们可以一次上手，终身难忘。

#### 1.6.4 Git 三棵树

我们的本地仓库由 git 维护的三棵“树”组成。第一个是我们的工作区，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。

在我们的 .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。下面这个图展示了工作区、版本库中的暂存区和版本库之间的关系<sup>9</sup>，希望你能耐着性子仔细理解这张图 and 不同操作所带来的不同影响。

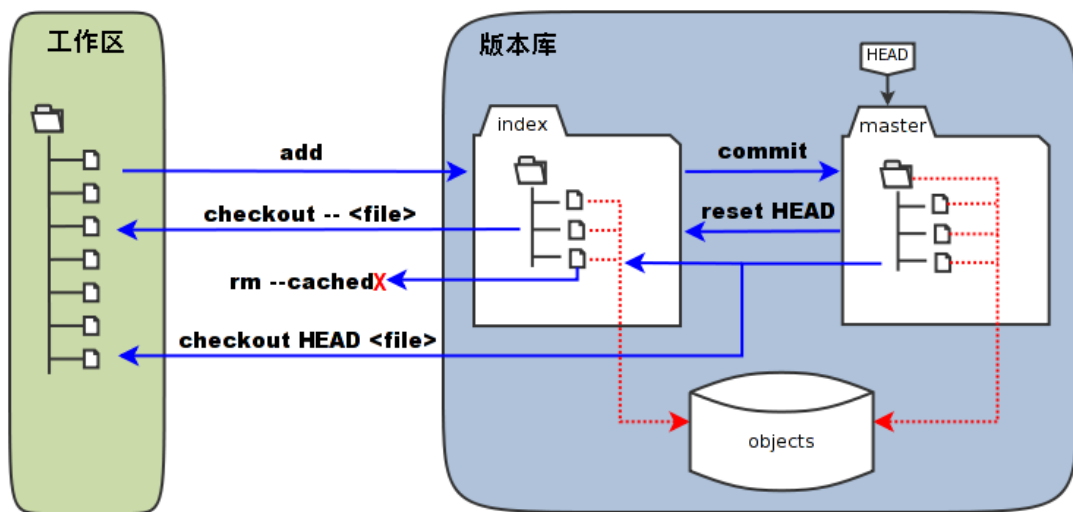


Figure 1.8: 工作区、暂存区和版本库

- 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“游标”。所以图示的命令中出现 HEAD 的地方可以用 master 来替换。

<sup>9</sup>这张图转载自网站<http://www.worldhello.net/2010/11/30/2166.html>

- 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 master 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，被 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。

我们在下载软件的时候常常会我们在考虑暂存区和版本库的关系的时候，可以粗略地认为暂存区是开发版，而版本库可以认为是稳定版，而 commit 其实就是将稳定版版本升到当前开发版的一个操作。

Git 中引入的**暂存区**的概念可以说是 Git 里最难理解但是却是最有亮点的设计之一，我们在这里不再详细介绍其能快速快照与回滚的原理，如果有兴趣的同学不妨去看看[Pro Git](#)这本书。

### 1.6.5 Git 时光机



Figure 1.9: 多啦 A 梦的时光机

我们都知道多啦 A 梦的时光机能穿越时空回到过去，而在我们神奇的 Git 里，也有堪称时光机的指令哦！在学习之前，我们先学习一下已经大致了解的一些伪·时光机指令，比如下面这些

**git rm --cached <file>** 这条指令是指从暂存区中删去一些我们不想跟踪的文件，比如我们自己调试用的文件等。

**git checkout - <file>** 如果我们在工作区改呀改，把一堆文件改得乱七八糟的，发现编译不过了！别急，如果我们还没 **git add**，就能使用这条命令，把它变回曾经美妙的样子。

**git reset HEAD <file>** 刚才提到，如果没有 **git add** 把修改放入暂存区的话，我们可以使用 **checkout** 命令，那么如果我们不慎已经 **git add** 加入了怎么办呢？那就需要这条指令来帮助我们了！这条指令可以让我们的**暂存区**焕然一新。再对同一个文件使用楼上那条指令，哈哈，世界清静了。

**git clean <file> -f** 如果你的工作区这时候混入了奇怪的东西，你没有追踪它，但是想清除它的话就可以使用这条指令，它可以帮你把奇怪的东西剔除出去。

好了，学了这么多，我们来利用自己的知识帮助小明摆脱困境吧。

**Thinking 1.1**

- 深夜，小明在做操作系统实验。困意一阵阵袭来，小明睡倒在了键盘上。等到小明早上醒来的时候，他惊恐地发现，他把一个重要的代码文件 `printf.c` 删除掉了。苦恼的小明向你求助，你该怎样帮他把代码文件恢复呢？
- 正在小明苦恼的时候，小红主动请缨帮小明解决问题。小红很爽快地在键盘上敲下了 `git rm printf.c`，这下事情更复杂了，现在你又该如何处理才能弥补小红的过错呢？
- 处理完代码文件，你正打算去找小明说他的文件已经恢复了，但突然发现小明的仓库里有一个叫 **Tucaao.txt**，你好奇地打开一看，发现是吐槽操作系统实验的，且该文件已经被添加到暂存区了，面对这样的情况，你该如何设置才能使 `Tucaao.txt` 在不从工作区删除的情况下不会被 `git commit` 指令提交到版本库？

关于上面那些撤销指令，等到你哪天突然不小心犯错的时候再来查阅即可，当然更推荐你使用 `git status` 来看当前状态下 Git 的推荐指令。我们现阶段先掌握好 `add` 和 `commit` 的用法即可。当然，**一定要慎用撤销指令**。虽然说 Git 理论上没有不能穿越的时空，但是需要我们功力深厚，掌握许多奇技淫巧，否则撤销之后如何撤除撤销指令将是一件难事。

介绍完上面三条撤销指令，我们来介绍真正的时光机指令

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

为了体会它的作用，我们做个小练习试一下

**Exercise 1.7** • 找到我们在/home/1406xxxx/下刚刚创建的 README.txt，没有的话就新建一个。

- 在文件里加入 **Testing 1**，add，commit，提交说明写 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 命令查看一下提交日志，看是否已经有三次提交了？记下提交说明为 3 的哈希值<sup>a</sup>。
- 开动时光机！使用 git reset --hard HEAD~，现在再使用 git log，看看什么没了？
- 找到提交说明为 1 的哈希值，使用 git reset --hard <Hash-code>，再使用 git log，看看什么没了？
- 现在我们已经回到过去了，为了再次回到未来，使用 git reset --hard <Hash-code>，再使用 git log，我胡汉三又回来了！

<sup>a</sup>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

这条指令就是我们可前进，可后退，还可以随意篡改“历史”的时光机是也。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~，上上个的话就用 HEAD^^，当然要是退 50 次的话写不了那么多~，可以使用 HEAD~50 来代替。第二种就是使用我们神器 Hash 值，用 Hash 值不仅可以回到过去，还可以“回到未来”。Hash 值在手，天下任我走！

现在我们已经学会了一大杀器，其正式的名字其实叫做**版本回退**。我们再来学个 Git 里同样被称为**必杀级特性**的神奇性质！

### 1.6.6 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同时继续工作。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指令

```
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>
```

这条指令往往会在我们进行周一小测的时候用到。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 master<sup>10</sup>分支下已经有过三次提交记录，这时我们使用 branch 命令新建了一个分支为 testing（参考图 1.10）。

<sup>10</sup>master 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

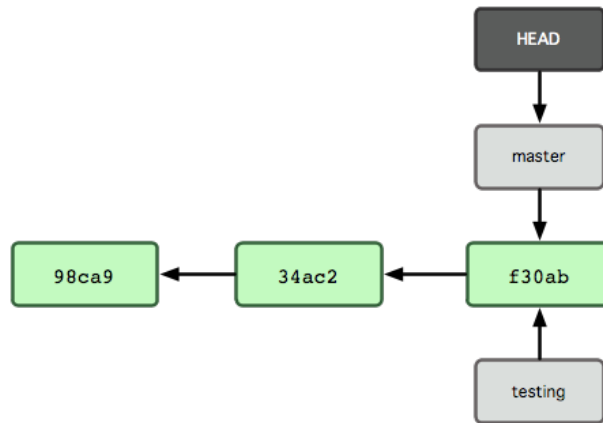


Figure 1.10: 分支建立后

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

```

想查看分支情况以及当前所在分支，只需要加上 `-a` 选项即可

```

1 # 查看所有的远程与本地分支
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指令了

```

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1.11)。

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。

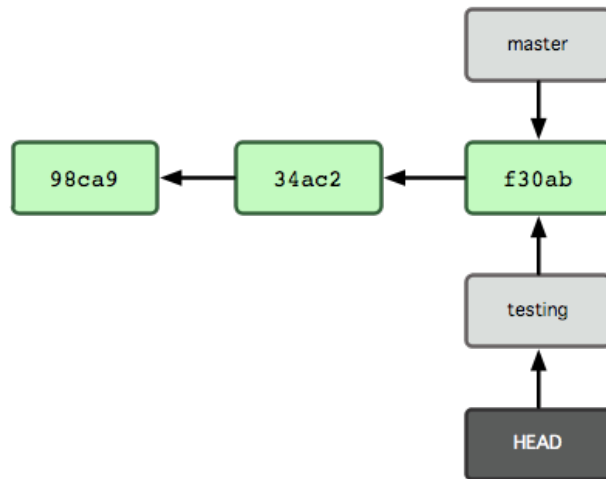


Figure 1.11: 分支切换后

在我们的操作系统实验中，有以下几种分支：

**labx** 这是我们提交实验代码的分支，这个分支不需要我们手动创建。当写好代码提交到服务器上后，在该次实验结束后，使用后面提到的更新指令可获取到新的实验分支，到时只需要使用 `git checkout labx` 即可进行新的实验。

**labx-exam** 这是我们周一小测实验的分支，每次需要使用 `git branch` 指令将刚完成的实验分支拷贝一份到 `labx-exam` 分支下，并进行小测代码的填写。

**labx-result** 这是我们每次实验结果的分支，每次的实验结果将会在该分支工作区的 `log` 文件夹下，数字越大代表检测的时间越近。测试下方 Summary : Number (in 100)，只要 `Number >= 60` 即算作通过本次实验。

**Note 1.6.7** 每次实验虽然是 60 算实验通过，但是 Summary 最好是 100。因为每次新实验的代码是你刚完成的实验代码以及一些新的要填充的文件组成的，前面实验的错误可能会在后面的实验中变成不小的坑。当然 Summary 为 100 也不代表实验一定全部正确，尽可能多花点时间理解与修改。

我们之前所介绍的这些指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`

其余指令可以临时查阅，当然掌握对你益处现在体会不出来，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指令。这组指令是最容易出错的，所以你一定要认真学习。



### 1.6.7 Git 远程仓库与本地

在我们的实验中，我们设立了几台服务器主机作为大家的远程仓库。那么远程仓库是什么呢？远程仓库其实和你本地版本库结构是一致的，只不过远程仓库是在服务器上的仓库，而本地仓库是在本地的。实验中我们每次对代码有所修改时，最后都需要在实验截止时间之前提交到服务器上，我们以服务器上的远程仓库里的代码为评测标准哦。我们先介绍一条我们实验中比较常用的一条命令

```
1 # git clone 用于从远程仓库克隆一份到本地版本库
2 $ git clone git@ip: 学号 -lab
```

从名字也能很容易理解这条指令的含义所在，我们就是使用 clone 指令而把服务器上的远程仓库拷贝到本地版本库里。这是一条很重要的指令，以后我们会经常使用。包括前期检查我们是否成功地提交到服务器上，以及后期使用 Git 为开源社区做贡献时都需要。但是初学者在使用这条命令的时候可能会遇到一个问题，那么来仔细思考一下下面的问题

**Thinking 1.2** 思考下面四个描述，你觉得哪些正确，哪些错误，请给出你参考的资料或实验证据。

1. 克隆时所有分支均被克隆，但只有 HEAD 指向的分支被检出。
2. 克隆出的工作区中执行 git log、git status、git checkout、git commit 等操作不会去访问远程版本库。
3. 克隆时只有远程版本库 HEAD 指向的分支被克隆。
4. 克隆后工作区的默认分支处于 master 分支。

**Note 1.6.8** 检出某分支指的是在该分支有对应的本地分支，使用 git checkout 后会在本地检出一个同名分支自动跟踪远程分支。比如现在本地空无一物，远程有一个名为 os 的分支，我们使用 git checkout os 即可在本地建立一个跟远程分支同名，自动追踪远程分支的 os 分支，并且在 os 分支下 push 时会默认提交到远程分支 os 上。

初学者最容易犯的一个错误是，在检查自己是否提交到服务器上时，克隆下来就着急慌地编译。大侠莫慌，看清楚分支再编译。我们克隆下来时默认处于 master 分支，但很可惜实验的代码是不会在 master 分支上测试的，所以我们要先使用 git checkout 检出对应的 labx 分支，再进行测试。

下面再介绍两条跟远程仓库有关的指令，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]
```

这条指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。比如我们实验小测时要在本地先建立一个 labx-exam 的分支，在提交完成后，我们要使用 **git push origin labx-exam** 在服务器上建立一个同名远程分支，这样服务器才可以通过检测该分支的代码来检测你的代码是否正确。

git pull 是条更新用的指令，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。

### 1.6.8 Git 冲突与解决冲突

这两条指令含义注释里也写得清楚，但是还是很容易出问题。新手使用 push 时，容易出现的大问题会是这样的

```
1 | 中文版：
2 | To git@github.com:1406xxxx.git
3 | ! [rejected]      master -> master (non-fast-forward)
4 | error: 无法推送一些引用到 'git@github.com:1406xxxx.git'
5 | 提示：更新被拒绝，因为您当前分支的最新提交落后于其对应的远程分支。
6 | 提示：再次推送前，先与远程变更合并（如 'git pull ...'）。详见
7 | 提示：'git push --help' 中的 'Note about fast-forwards' 小节。
8 |
9 | 英文版：
10 | To git@github.com:1406xxxx.git
11 | ! [rejected]      master -> master (non-fast-forward)
12 | error: failed to push some refs to 'To git@github.com:1406xxxx.git'
13 | hint: Updates were rejected because the tip of your current branch is behind
14 | hint: its remote counterpart. Integrate the remote changes (e.g.
15 | hint: 'git pull ...') before pushing again.
16 | hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

你的提示可能是英文的，但这并不妨碍问题的发生，这个问题是因为什么而产生的呢？我们来分析一下，想象你在公司和在家操作同一个分支，在公司你对一个文件进行了修改，然后进行了提交。回了家又对同样的文件做了不同的修改，在家中使用 push 同步到远程分支了。但等你回到公司再 push 的时候就会发现一个严重的问题：现在远程仓库和本地仓库已经分离开变成两条岔路了（见图1.12）。

这样的话远程仓库可就为难了，你在公司的提交有效，在家里的提交也有效，你又不想浪费劳动成果，想让远程仓库把你的提交全部接受，那么我们怎么样才能解决这个问题呢？这时候就要请出我们的 **git pull** 指令了！

你此时可能会产生一个很大的疑问，在 push 之前，使用 git pull 轻轻一挥，难道问题就能全部解决？答案当然是否定的，我们不能指望 Git 帮我们把文件中的修改全部妥善合并，但是 Git 为我们提供了另一种机制帮我们能快速定位有冲突（conflict）的文件，这时候我们使用 git pull，你可能会看到有下面这样的提示



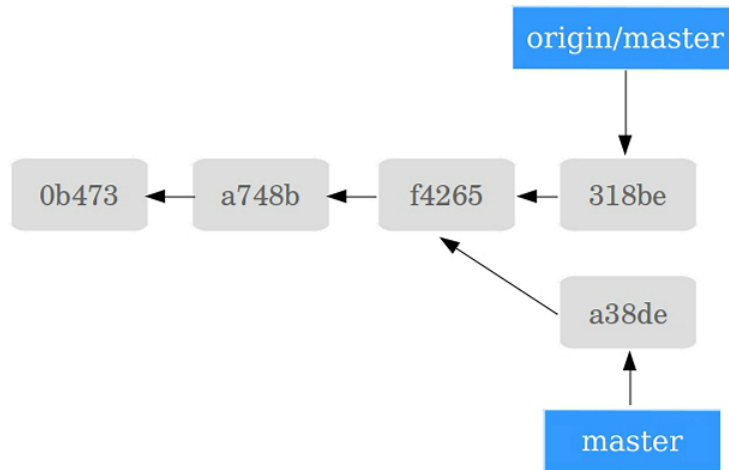


Figure 1.12: 远程仓库与本地仓库的岔路

```

1 Auto-merging test.txt
2 CONFLICT (content): Merge conflict in test.txt
3 Automatic merge failed; fix conflicts and then commit the result.

```

有冲突的文件中往往包含一部分类似如下的奇怪代码，我们打开 test.txt，发现这样一些“乱码”

```

1 a123
2 <<<<<<< HEAD
3 b789
4 =====
5 b45678910
6 >>>>>> 6853e5ff961e684d3a6c02d4d06183b5ff330dcc
7 c

```

冲突标记<<<<<<< 与 ===== 之间的内容是你在家里的修改，===== 与>>>>>>> 之间的内容是你公司的修改。

要解决冲突也很简单：编辑冲突文件，将其中冲突的内容手工合理合并一下就可以了，当然记得在文件中解决了冲突之后要重新 add 该文件并 commit。大声告诉我，是不是非常简单？

然而世间并没有那么多简单的事情，如果你足够不幸，你可能在 git pull 的时候也会遇到不小的问题，问题可能是这样的

```

1 error: Your local changes to the following files would be overwritten by merge:
2     1406xxxx-lab/readme.txt
3 Please, commit your changes or stash them before you can merge.
4 Aborting

```

其实提示已经比较清楚了，这里我们只需要把我们之前的所有修改全部提交(commit)即可，提交之后再 git pull 就好。当然，有更高级的用法是这样的，不推荐大家现在学习，如果你已经熟悉了 Git 的基础操作，那么可以阅读[git stash 解决 git pull 冲突](#)

不要觉得这一节的冲突一节不需要学习，因为你可能会想：我现在怎么可能在公司和家里同时修改文件呢！但是要注意，在远程仓库编辑的不止你一个人，还有助教老师，

助教老师一旦修改一些东西都有可能产生冲突，所以你一定要认真学会这一节的内容。当然实践是最好的老师，我们再来实践一下

**Exercise 1.8** 仔细回顾一下上面这些指令，然后完成下面的任务

- 在 `/home/1406xxxx/1406xxxx-lab` 下新建分支，名字为 `Test`
- 切换到 `Test` 分支，添加一份 `readme.txt`，内容写入自己的学号
- 将文件提交到本地版本库，然后建立相应的远程分支。

到这里 Git 教程基本就算是结束了，能看完这么长的教程也真是辛苦你了，奉送一下实验代码提交流程的简明教程，希望你可以快速上手，终身难忘！

### 1.6.9 实验代码提交流程

**modify** 写代码。

**git add & git commit <modified-file>** 提交到本地版本库。

**git pull** 从服务器拉回本地版本库，并解决服务器版本库与本地代码的冲突。

**git add & git commit <conflict-file>** 将远程库与本地代码合并结果提交到本地版本库。

**git push** 将本地版本库推到服务器。

**mkdir test & cd test & git clone** 建立一个额外的文件夹来测试服务器上的代码是否正确。

而我们在一次实验结束，新的实验代码下发时，一般是按照以下流程的来开启新的实验之旅。

**git add & git commit** 如果当前分支的暂存区还有东西的话，先提交。

**git pull** 这一步很重要！要先确保服务器上的更新全部同步到本地版本库！

**git checkout labx** 检出新实验分支并进行实验。

谨记，一定要勤使用 **git pull**，这条指令很重要！有事没事，同步一下！

感谢你看这篇长长的 Git 教程到现在，希望你能快乐地使用 Git，若有不会勤查教程<sup>11</sup>。如果你希望能学到更厉害的技术，推荐 [GitHub](https://github.com)，这是一个关于 Git 的通关小游戏。开启你快乐的实验之旅吧！^\_^

<sup>11</sup>推荐廖雪峰老师的网站: <http://www.liaoxuefeng.com/>

## 1.7 实验思考

- 思考 -小明的困境
- 思考 -克隆命令

请认真做练习，然后把实验思考里的内容附在实验文档中一起提交!

## CHAPTER 2

## 内存管理

### 2.1 实验目的

1. 了解 MIPS 内存映射布局
2. 掌握使用空闲链表的管理物理内存的方法
3. 建立页表，实现分页式虚存管理
4. 实现内存分配和释放的函数

本次实验中，我们需要掌握 MIPS 页式内存管理机制，需要使用一些数据结构来记录内存的使用情况，并实现内存分配和释放的相关函数，完成物理内存管理和虚拟内存管理。

### 2.2 MMU 和 TLB

首先介绍两个与内存管理有关的概念：MMU 和 TLB。

**Note 2.2.1** MMU，即内存管理单元（memory-management unit）。MMU 是 CPU 中用来管理虚拟存储器、物理存储器的控制线路，同时也负责虚拟地址映射为物理地址，以及提供硬件机制的内存访问授权。

**Note 2.2.2** TLB，即后备缓冲（translation lookaside buffer）。TLB 是将程序使用的地址（虚拟地址）翻译成物理地址（访问内存的地址）的硬件。

### 2.3 MIPS 虚存映射布局

32 位的 MIPS CPU 最大寻址空间为 4GB(2<sup>32</sup> 字节)，这 4GB 虚存空间被划分为四个部分：

1. kuseg (TLB-mapped cacheable user space, 0x00000000 ~ 0x7fffffff): 这一段是用户模式下可用的地址, 大小为 2G, 也就是 MIPS 约定的用户内存空间。需要通过 MMU 进行虚拟地址到物理地址的转换。
2. kseg0 (direct-mapped cached kernel space, 0x80000000 ~ 0x9fffffff): 这一段是内核地址, 其内存虚存地址到物理内存地址的映射转换不通过 MMU, 使用时只需要将地址的最高位清零 (& 0x7fffffff), 这些地址就被转换为物理地址。也就是说, 这段逻辑地址被连续地映射到物理内存的低端 512M 空间。对这段地址的存取都会通过高速缓存 (cached)。通常在没有 MMU 的系统中, 这段空间用于存放大多数程序和数据。对于有 MMU 的系统, 操作系统的内核会存放在这个区域。
3. kseg1 (direct-mapped uncached kernel space, 0xa0000000 ~ 0xbfffffff): 与 kseg0 类似, 这段地址也是内核地址, 将虚拟地址的高 3 位清零 (& 0x1fffffff), 就可以转换到物理地址, 这段逻辑地址也是被连续地映射到物理内存的低端 512M 空间。但是 kseg1 不使用缓存 (uncached), 访问速度比较慢, 但对硬件 I/O 寄存器来说, 也就不存在 Cache 一致性的问题了, 这段内存通常被映射到 I/O 寄存器, 用来实现对外设的访问。
4. kseg2 (TLB-mapped cacheable kernel space, 0xc0000000 ~ 0xffffffff): 这段地址只能在内核态下使用, 并且需要 MMU 的转换。

## 2.4 内存管理

在第一实验中, 我们将内核加载到内存中的 kseg0 区域 (0x80010000), 成功启动并跳转到 init/main.c 中的 main 函数开始运行, 现在我们需要在 main 函数中调用定义在 init/init.c 中的 mips\_init() 函数, 并进一步通过

1. mips\_detect\_memory();
2. mips\_vm\_init();
3. page\_init();

这三个函数来实现内存管理的相关数据结构的初始化。

## 2.5 物理内存管理

### 2.5.1 内存控制块

在 MIPS CPU 中, 地址转换以 4KB 大小为单位, 称为页。我们使用 Page 结构体来作为记录一页内存的相关信息的数据结构:

```
1 typedef LIST_ENTRY(Page) Page_LIST_entry_t;
2
3 struct Page {
4     Page_LIST_entry_t pp_link; /* free list link */
```

```

5     u_short pp_ref;
6 };

```

其中, `pp_ref` 用来记录这一物理页面的引用次数, `pp_link` 是当前节点指向链表中下一个节点的指针, 其类型为 `LIST_ENTRY(Page)`。我们在 `include/queue.h` 中定义了一系列的宏函数来简化对链表的操作。请阅读这些宏函数的代码, 理解它们的原理和巧妙之处。

**Thinking 2.1** 我们注意到我们把宏函数的函数体写成了 `do { // ... } while(0)` 的形式, 而不是仅仅写成形如 `{ // ... }` 的语句块, 这样的写法好处是什么? ■

在 `include/pmap.h` 中, 我们使用 `LIST_HEAD` 宏来定义了一个结构体类型 `Page_list`, 在 `mm/pmap.c` 中, 创建了一个该类型的变量 `page_free_list` 来以链表的形式表示所有的空闲物理内存:

```

1 LIST_HEAD(Page_list, Page);
2
3 static struct Page_list page_free_list; /* Free list of physical pages */

```

### 2.5.2 内存分配和释放

首先, 我们需要注意在 `mm/pmap.c` 中定义的和内存相关的全局变量:

```

1 u_long maxpa;          /* Maximum physical address */
2 u_long npage;          /* Amount of memory(in pages) */
3 u_long basemem;        /* Amount of base memory(in bytes) */
4 u_long extmem;         /* Amount of extended memory(in bytes) */

```

**Exercise 2.1** 我们需要在 `mips_detect_memory()` 函数中初始化这几个全局变量, 以确定内核可用的物理内存的大小和范围。根据代码注释中的提示, 完成 `mips_detect_memory()` 函数。 ■

在操作系统刚刚启动时, 我们还没有建立起有效的数据结构来管理所有的物理内存, 因此, 出于最基本的内存管理的需求, 我们需要实现一个函数来分配指定字节的物理内存。这一功能由 `mm/pmap.c` 中的 `alloc` 函数来实现。

```

1 static void *alloc(u_int n, u_int align, int clear);

```

`alloc` 函数能够按照参数 `align` 进行对齐, 然后分配 `n` 字节大小的物理内存, 并根据参数 `clear` 的设定决定是否将新分配的内存全部清零, 并最终返回新分配的内存的首地址。

有了分配物理内存的功能后, 接下来我们需要给操作系统内核必须的数据结构 – 页表 (`pgdir`)、内存控制块数组 (`pages`) 和进程控制块数组 (`envs`) 分配所需的物理内存。`mips_vm_init()` 函数实现了这一功能, 并且完成了相关的虚拟内存与物理内存之间的映射。

完成上述工作后, 我们便可以通过在 `mips_init()` 函数中调用 `page_init()` 函数将余下的物理内存块加入到空闲链表中。

**Exercise 2.2** 完成 `page_init` 函数，使用 `include/queue.h` 中定义的宏函数将未分配的物理页加入到空闲链表 `page_free_list` 中去。思考如何区分已分配的内存块和未分配的内存块，并注意内核可用的物理内存上限。

有了记录物理内存使用情况的链表之后，我们就可以不再像之前的 `alloc` 函数那样按字节为单位进行内存的分配，而是可以以页为单位进行物理内存的分配与释放。`page_alloc` 函数用来从空闲链表中分配一页物理内存，而 `page_free` 函数则用于将一页之前分配的内存重新加入到空闲链表中。

**Exercise 2.3** 完成 `mm/pmap.c` 中的 `page_alloc` 和 `page_free` 函数，基于空闲内存链表 `page_free_list`，以页为单位进行物理内存的管理。

至此，我们的内核已经能够按照分页的方式对物理内存进行管理。

2.6 虚拟内存管理

我们通过建立两级页表来进行虚拟内存的管理，在此基础上，我们将实现在页表中根据虚拟地址在页表中查找对应的物理地址，将一段虚存地址映射到一段的物理地址，然后实现虚存的管理与释放，最后为内核建立起一套虚存管理系统。

2.6.1 两级页表机制

我们的操作系统内核采取二级页表结构，如图2.1所示：

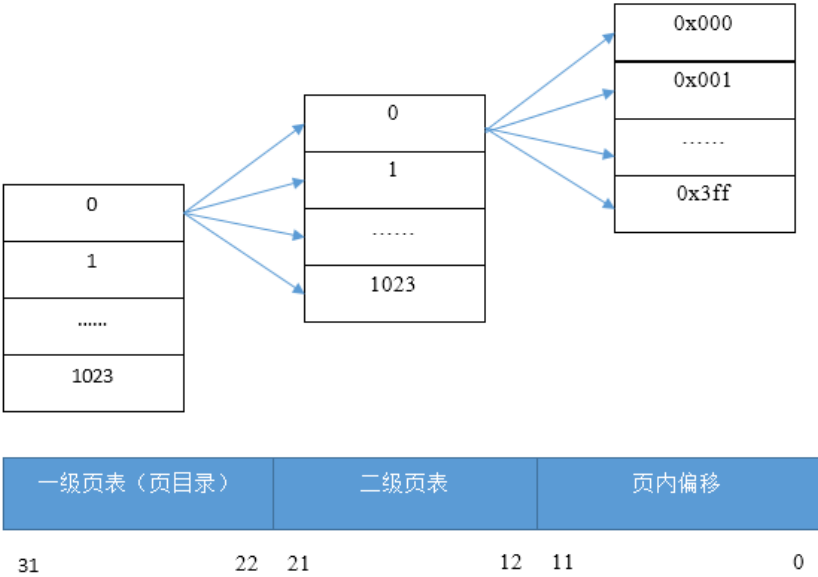


Figure 2.1: 二级页表结构示意图

第一级表称为页目录 (page directory)，一共 1024 个页目录项，每个页目录项 32 位 (4 Byte)，页目录项存储的值为其对应的二级页表入口的物理地址。整个页目录存放

在一个页面 (4KB) 中, 也就是我们在 `mips_vm_init` 函数中为其分配了相应的物理内存。第二级表称为页表 (page table), 每一张页表有 1024 个页表项, 每个页表项 32 位 (4 Byte), 页表项存储的是对应页面的页框号 (20 位) 以及标志位 (12 位)。每张页表占用一个页面大小 (4KB) 的内存空间。

对于一个 32 位的虚存地址, 其 31-22 位表示的是页目录项的索引, 21-12 位表示的是页表项的索引, 11-0 位表示的是该地址在该页面内的偏移。

### 2.6.2 地址转换

对于操作系统来说, 虚拟地址与物理地址之间的转换是内存管理中非常重要的内容。在这一部分, 我们将详细探讨咱们的内核是如何进行地址转换的。

首先从较为简单的形式开始。在前面的实验中, 我们通过设置 `lds` 文件让操作系统内核加载到内存的 `0x80010000` 位置, 在上文我们对 MIPS 存储器映射布局的介绍中我们知道, 这一地址对应的是 `kseg0` 区域, 这一部分的地址转换不通过 MMU 进行。我们也称这一部分虚拟地址为内核虚拟地址。从虚拟地址到物理地址的转换只需要清掉最高位的零即可, 反过来, 将对应范围内的物理地址转换到内核虚拟地址, 也只需要将最高位设置为 1 即可。我们在 `include/mmu.h` 中定义了 `PADDR` 和 `KADDR` 两个宏来实验这一功能:

```

1 // translates from kernel virtual address to physical address.
2 #define PADDR(kva) \
3     ({ \
4         u_long a = (u_long) (kva); \
5         if (a < ULIM) \
6             panic("PADDR called with invalid kva %08lx", a);\
7         a - ULIM; \
8     })
9
10 // translates from physical address to kernel virtual address.
11 #define KADDR(pa) \
12     ({ \
13         u_long ppn = PPN(pa); \
14         if (ppn >= npage) \
15             panic("KADDR called with invalid pa %08lx", (u_long)pa);\
16         (pa) + ULIM; \
17     })

```

在 `PADDR` 中, 我们使用了一个宏 `ULIM`, 这个宏定义在 `include/mmu.h` 中, 其值为 `0x80000000`。对于小于 `0x80000000` 的虚拟地址值, 显然不可能是内核区域的虚拟地址。在 `KADDR` 中, 一个合理的物理地址的物理页框号显然不能大于我们在 `mm/pmap.c` 中所定义的物理内存总页数 `npage` 的值。

接下来, 我们讨论如何通过二级页表进行虚拟地址到物理地址的转换。

首先, 我们可以通过 `PDX(va)` 来获得一个虚拟地址对应的页目录索引, 然后直接以凭借索引在页目录中得到对应的二级页表的基址 (物理地址), 然后把这个物理地址转化为内核虚拟地址 (`KADDR`), 之后, 通过 `PTX(va)` 获得这个虚存地址对应的页表索引, 然后就可以从页表中得到对应的页面的物理地址。整个转换的过程如图2.2 所示:



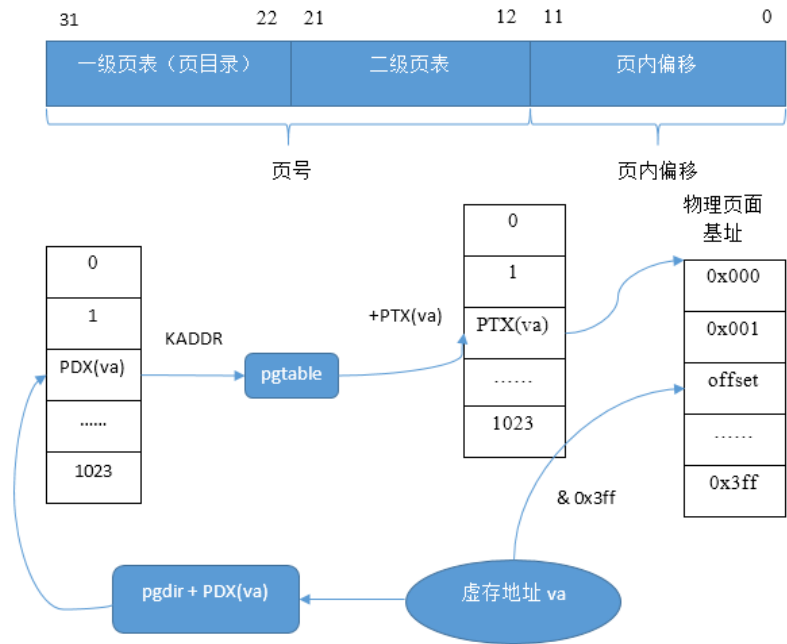


Figure 2.2: 地址转换过程

2.6.3 页目录自映射

上文中我们讲二级页表结构的时候提到，要映射整个 4G 地址空间，一共需要 1024 个页表和 1 个页目录的。一个页表占用 4KB 空间，页目录也占用 4KB 空间，也就是说，整个二级页表结构将占用 4MB+4KB 的存储空间， $1024 \times 4KB + 1 \times 4KB = 4MB + 4KB$ 。

在 include/mmu.h 中的内存布局图里有这样的内容：

```
1  /**
2   *      ULIM      -----> +-----+-----+-----0x8000 0000-----+
3   *      /          User VPT          /      PDMAP          /\
4   *      UVPT      -----> +-----+-----+-----0x7fc0 0000      /
5   */
```

不难计算出 UVPT(0x7fc00000) 到 ULIM(0x80000000) 之间的空间只有 4MB，这一区域就是进程的页表的位置，于是我们不禁想问：页目录所占用的 4KB 内存空间在哪儿？

**答案就在于页目录的自映射机制！**

如果页表和页目录没有被映射到进程的地址空间中，而一个进程的 4GB 地址空间又都映射了物理内存的话，那么就确实需要 1024 个物理页 (4MB) 来存放页表，和另外 1 个物理页 (4KB) 来存放页目录，也就是需要 (4M+4K) 的物理内存。但是页表也被映射到了进程的地址空间中，也就意味着 1024 个页表中，有一个页表所对应的 4M 空间正是这 1024 个页表所占用的 4M 内存，这个页表的 1024 的页表项存储了 1024 个物理地址，分别是 1024 个页表的物理地址。而在二级页表结构中，页目录对应着二级页表，1024 个页目录项存储的也是全部 1024 个页表的物理地址。也就是说，一个页表的内容和页目录的内容是完全一样的，正是这种完全相同，使得将 1024 个页表加 1 个页目录

映射到地址空间只需要 4M 的地址空间，其中的一个页表和页目录完全重合了。

接下来我们会想到这样一个问题：那么，这个与页目录重合的页表，也就是页目录究竟在哪儿呢？

我们知道，这 4M 空间的起始位置也就是第一个二级页表对应着页目录的第一个页目录项，同时，由于 1M 个页表项和 4G 地址空间是线性映射，不难算出  $0x7fc00000$  这一地址对应的应该是第  $(0x7fc00000 \gg 12)$  个页表项，这个页表项也就是第一个页目录项。一个页表项 32 位，占用 4 个字节的内存，因此，其相对于页表起始地址  $0x7fc00000$  的偏移为  $(0x7fc00000 \gg 12) * 4 = 0x1ff000$ ，于是得到地址为  $0x7fc00000 + 0x1ff000 = 0x7fdff000$ 。也就是说，页目录的虚存地址为  $0x7fdff000$ 。

**Thinking 2.2** 了解了二级页表页目录自映射的原理之后，我们知道，Win2k 内核的虚存管理也是采用了二级页表的形式，其页表所占的 4M 空间对应的虚存起始地址为  $0xC0000000$ ，那么，它的页目录的起始地址是多少呢？ ■

#### 2.6.4 创建页表

将虚拟地址转换为物理地址的过程中，如果这个虚拟地址所对应的二级页表不存在，有时，我们可能需要为这个虚拟地址创建一个新的页表。我们需要申请一页物理内存，然后将他的物理地址赋值给对应的页目录项，最后设置好页目录项的权限位即可。

我们的内核在 `mm/pmap.c` 中分定义了 `boot_pgdir_walk` 和 `pgdir_walk` 两个函数来实现地址转换和页表的创建，这两个函数的区别仅仅在于当虚存地址所对应的页表的页面不存在的时候，分配策略的不同和使用的内存分配函数的不同。前者用于内核刚刚启动的时候，这一部分内存通常用于存放内存控制块和进程控制块等数据结构，相关的页表和内存映射必须建立，否则操作系统内核无法正常执行后续的工作。然而用户程序的内存申请却不是必须满足的，当可用的物理内存不足时，内存分配允许出现错误。`boot_pgdir_walk` 被调用时，还没有建立起空闲链表来管理物理内存，因此，直接使用 `alloc` 函数以字节为单位进行物理内存的分配，而 `pgdir_walk` 则在空闲链表初始化之后发挥功能，因此，直接使用 `page_alloc` 函数从空闲链表中以页为单位进行内存的申请。

上文中介绍了页目录自映射的相关知识，我们了解到页目录其实也是一个页表。我们知道，在咱们实验使用的内核中，一个页表指向的物理页面存在的标志是页表项存储的值的 `PTE_V` 标志位被置为 1。因此，在将页表的物理地址赋值给页目录项时，我们还为页目录项设置权限位。

**Exercise 2.4** 完成 `mm/pmap.c` 中的 `boot_pgdir_walk` 和 `pgdir_walk` 函数，实现虚拟地址到物理地址的转换以及创建页表的功能。 ■

#### 2.6.5 地址映射

**Exercise 2.5** 实现 `mm/pmap.c` 中的 `boot_map_segment` 函数，实现将制定的物理内存与虚拟内存建立起映射的功能。 ■

#### 2.6.6 page insert and page remove

```

1 // Overview:
2 // Map the physical page 'pp' at virtual address 'va'.
3 // The permissions (the low 12 bits) of the page table entry
4 // should be set to 'perm|PTE_V'.
5 //
6 // Post-Condition:
7 // Return 0 on success
8 // Return -E_NO_MEM, if page table couldn't be allocated
9 //
10 // Hint:
11 // If there is already a page mapped at `va`, call page_remove()
12 // to release this mapping. The `pp_ref` should be incremented
13 // if the insertion succeeds.
14 int
15 page_insert(Pde *pgdir, struct Page *pp, u_long va, u_int perm)
16 {
17     u_int PERM;
18     Pte *pgtable_entry;
19     PERM = perm | PTE_V;
20
21     /* Step 1: Get corresponding page table entry. */
22     pgdir_walk(pgdir, va, 0, &pgtable_entry);
23
24     if (pgtable_entry != 0 && (*pgtable_entry & PTE_V) != 0) {
25         if (pa2page(*pgtable_entry) != pp) {
26             page_remove(pgdir, va);
27         }
28         else{
29             tlb_invalidate(pgdir, va);
30             *pgtable_entry = (page2pa(pp) | PERM);
31             return 0;
32         }
33     }
34
35     /* Step 2: Update TLB. */
36     tlb_invalidate(pgdir, va);
37
38     /* Step 3: Do check, re-get page table entry to validate
39      * the insertion. */
40     if (pgdir_walk(pgdir, va, 1, &pgtable_entry) != 0) {
41         return -E_NO_MEM;    // panic ("page insert failed .\n");
42     }
43
44     *pgtable_entry = (page2pa(pp) | PERM);
45     pp->pp_ref++;
46     return 0;
47 }

```

这个函数虽然我们并没有填，但是**非常重要**！这个函数将在 lab3 和 lab4 中被反复用到，这个函数将 va 虚拟地址和其要对应的物理页 pp 的映射关系以 perm 的权限设置加入页目录。我们大概讲一下函数的执行流程与执行要点。

**流程大致如下：**先判断 va 是否有对应的页表项：如果页表项有效（或者叫 va 是否已经有了映射的物理地址）的话，则去判断这个物理地址是不是我们要插入的那个物理地址，如果不是，那么就把该物理地址移除掉；如果是的话，则修改权限，放到 tlb 中。

**有一个值得指出的要点：**我们能看到，只要对页表的内容修改，都必须 tlb\_invalidate 来让 tlb 更新，否则后面紧接着对内存的访问很有可能出错。

可以说 `tlb_invalidate` 函数是它的一个核心子函数, 这个函数实际上又是由 `tlb_out` 汇编函数组成的。

Listing 8: TLB 汇编函数

```

1  #include <asm/regdef.h>
2  #include <asm/cp0regdef.h>
3  #include <asm/asm.h>
4
5  LEAF(tlb_out)
6      nop
7      mfc0      k1,CPO_ENTRYHI
8      mtc0      a0,CPO_ENTRYHI
9      nop
10     tlbp
11     nop
12     nop
13     nop
14     nop
15     mfc0      k0,CPO_INDEX
16     bltz      k0,NOFOUND
17     nop
18     mtc0      zero,CPO_ENTRYHI
19     mtc0      zero,CPO_ENTRYLO0
20     nop
21     tlbbwi
22 NOFOUND:
23
24     mtc0      k1,CPO_ENTRYHI
25
26     j         ra
27     nop
28 END(tlb_out)

```

这个汇编函数相对其他汇编函数来说相对简单, 那么留给你一个思考问题。

**Thinking 2.3** 思考一下 `tlb_out` 汇编函数, 结合代码阐述一下跳转到 **NOFOUND** 的流程?

## 2.7 正确结果展示

实验二做完之后, 正确的结果应该是这样的:

```

1  main.c: main is start ...
2  init.c: mips_init() is called
3  Physical memory: 65536K available, base = 65536K, extended = 0K
4  to memory 80401000 for struct page directory.
5  to memory 80431000 for struct Pages.
6  mips_vm_init:boot_pgdir is 80400000
7  pmap.c: mips vm init success
8  start page_insert
9  va2pa(boot_pgdir, 0x0) is 3ffe000
10 page2pa(pp1) is 3ffe000
11 pp2->pp_ref 0

```

```
12  end page_insert
13  page_check() succeeded!
14  panic at init.c:55: ~~~~~
```

最后一行的数字 55 是不固定的。

## 2.8 实验思考

- 思考 -使用 `do-while(0)` 语句的好处
- 思考 -自映射机制页目录地址的计算
- 思考 -`NOFOUND` 的奥妙

## 2.9 挑战性任务

我们的虚存管理系统分配和回收页面粒度的内存，问题是如果我们需要支持真正的 I/O 设备，需要分配超过 4KB 的物理连续的内存；如果我们需要用户态使用，为了提高 CPU 效率，我们需要分配大小超过 4MB 的超级页。修改虚存管理，使得可以一次申请 2 的指数大小的内存，你可以自行设置大小的上限。确保，在需要的时候，将大的存储单元分为小的存储单元；如果可能，将小的存储单元合并为大的存储单元。

## CHAPTER 3

### 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

在本次实验中你将运行一个用户模式的进程。你需要使用数据结构进程控制块 `Env` 来跟踪用户进程。通过建立一个简单的用户进程，加载一个程序镜像到进程控制块中，并让它运行起来。同时，你的 MIPS 内核将拥有处理异常的能力。

### 3.2 进程

**Note 3.2.1** 进程既是基本的分配单元，也是基本的执行单元。第一，进程是一个实体。每一个进程都有它自己的地址空间，一般情况下，包括代码段、数据段和堆栈。第二，进程是一个“执行中的程序”。程序是一个没有生命的实体，只有处理器赋予程序生命时，它才能成为一个活动的实体，我们称其为进程。

#### 3.2.1 进程控制块

这次实验是关于进程和异常的，那么我们首先来结合代码看看进程控制块是个什么东西。

进程控制块 (PCB) 是系统为了管理进程设置的一个专门的数据结构，用它来记录进程的外部特征，描述进程的运动变化过程。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。** 通常 PCB 应包含如下一些信息：

Listing 9: 进程控制块

```

1  struct Env {
2      struct Trapframe env_tf;           // Saved registers
3      LIST_ENTRY(Env) env_link;         // Free LIST_ENTRY
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 the environment
7      Pde *env_pgdir;                  // Kernel virtual address of page dir
8      u_int env_cr3;
9  };

```

为了集中你的注意力在关键的地方，我们暂时不介绍其他实验所需介绍的内容。下面是 **lab3** 中需要用到的这些域的一些简单说明：

- `env_tf` : `Trapframe` 结构体的定义在 `include/trap.h` 中，`env_tf` 的作用就是在进程因为时间片用光不再运行时，将其当时的进程上下文环境保存在 `env_tf` 变量中。当从用户模式切换到内核模式时，内核也会保存进程上下文，因此进程返回时上下文可以从中恢复。
- `env_link` : `env_link` 的机制类似于实验二中的 `pp_link`，使用它和 `env_free_list` 来构造空闲进程链表。
- `env_id` : 每个进程的 `env_id` 都不一样，`env_id` 是进程独一无二的标识符。
- `env_parent_id` : 该变量存储了创建本进程的进程 `id`。这样进程之间通过父子进程之间的关联可以形成一颗进程树。
- `env_status` : 该变量只能在以下三个值中进行取值：
  - `ENV_FREE` : 表明该进程是不活动的，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，处于该状态的进程往往在等待一定的条件才可以变为就绪状态从而被 CPU 调度。
  - `ENV_RUNNABLE` : 表明该进程处于就绪状态，正在等待被调度，但处于 `RUNNABLE` 状态的进程可以是正在运行的，也可能不在运行中。
- `env_pgdir` : 这个变量保存了该进程页目录的虚拟地址。
- `env_cr3` : 这个变量保存了该进程页目录的物理地址。

这里值得强调的一点就是 **`ENV_RUNNABLE` 状态并不代表进程一定在运行中，它也可能正处于调度队列中。**而我们的进程调度也只会调度处于 `RUNNABLE` 状态的进程。

既然我们知道了进程控制块，我们就来认识一下进程控制块数组 `envs`。在我们的实验中，存放进程控制块的物理内存存在系统启动后就要被分配好，并且这块内存不可被换出。所以需要在系统启动后就要为 `envs` 数组分配内存，下面你就要完成这个重任



**Exercise 3.1** • 修改 `pmap.c/mips_vm_init` 函数来为 `envs` 数组分配空间。

- `envs` 数组包含 `NENV` 个 `Env` 结构体成员，你可以参考 `pmap.c` 中已经写过的 `pages` 数组空间的分配方式。
- 除了要为数组 `envs` 分配空间外，你还需要使用 `pmap.c` 中你填写过的一个内核态函数为其进行段映射，`envs` 数组应该被 `UENVS` 区域映射，你可以参考 `./include/mmu.h`。

当然我们光有了容纳进程控制块存储信息的”房子”`envs` 还不够，我们还需要像 lab2 一样将空闲的 `env` 控制块按照链表形式”串”起来，便于后续分配 `ENV` 结构体对象，形成 `env_free_list`。一开始我们的所有进程控制块都是空闲的，所以我们要把它们都”串”到 `env_free_list` 上去。

**Exercise 3.2** 仔细阅读注释，填写 `env_init` 函数，注意要按照逆序插入。

在填写完 `env_init` 函数后，我们对于 `envs` 的操作暂时就告一段落了，不过我们还有一个小问题没解决，注释里说明了要逆序插入，但是为什么呢？我们需要你来仔细思考，注释中已经给出了提示。

**Thinking 3.1** 为什么我们在构造空闲进程链表时使用了逆序插入的方式？

### 3.2.2 设置进程控制块

做完上面那个小练习后，那么我们就可以开始利用空闲进程链表 `env_free_list` 创建进程来玩了。下面我们就来具体讲讲你应该如何创建一个进程<sup>1</sup>。

进程创建的流程如下：

**第一步** 申请一个空闲的 PCB，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，由于没有进程存在，所以没有办法”生”进程，所以进程拥有的所有信息都是强制设置的。而进程的信息又都存放于进程控制块中，所以我们需要手工初始化进程控制块。

**第三步** 进程光有 PCB 那点信息还没法跑起来，我们知道，每个进程都有独立的地址空间。所以，我们要为新进程分配资源，为新进程的程序和数据以及用户栈分配必要的内存空间。

**第四步** 这时候 PCB 已经被涂涂画画了很多东西，不再是一张白纸了，所以我们还要把它从空闲链表里除名。

当然，第二步的信息设置是实验的关键，那么下面让我们来结合注释看看这段代码

<sup>1</sup>这里我们创建进程是指系统创建进程，不是指 `fork` 等进程”生”进程。我们将在 lab4 中接触另一种进程创建的方式。



Listing 10: 进程创建

```

1  /* Overview:
2   * Allocates and Initializes a new environment.
3   * On success, the new environment is stored in *new.
4   *
5   * Pre-Condition:
6   * If this don't have parent, parent_id should be zero.
7   * env_init has been called before this function.
8   *
9   * Post-Condition:
10  * return 0 on success, and set appropriate values for Env new.
11  * return -E_NO_FREE_ENV on error, if no free env.
12  *
13  * Hints:
14  * You may use these functions and defines:
15  *     LIST_FIRST, LIST_REMOVE, mkenvid (Not All)
16  * You should set some states of Env:
17  *     id , status , the sp register, CPU status , parent_id
18  *     (the value of PC should NOT be set in env_alloc)
19  */
20
21  int
22  env_alloc(struct Env **new, u_int parent_id)
23  {
24      int r;
25      struct Env *e;
26
27      /*Step 1: Get a new Env from env_free_list*/
28
29
30      /*Step 2: Call some function(has been implemented)
31       * to intial kernel memory layout for this new Env.
32       * this function mainly map the kernel address
33       * to this new Env address. */
34
35
36      /*Step 3: Initial every field of new Env to appropriate value*/
37
38
39      /**Your Question Here**/
40      /*Step 4: focus on initializing env_tf structure, located at this new Env.
41       * especially the sp register,CPU status. */
42      e->env_tf.cp0_status = 0x10001004;
43
44
45      /*Step 5: Remove the new Env from Env free list*/
46
47
48  }
```

相信你一眼看到第三条注释的时候一定会吐槽：“什么鬼，什么叫合适的值啊”。淡定，先别着急吐槽，先花半分钟的时间看一下第二条注释。

下面这个函数就是你在第二步中要使用的函数。在用之前，我们希望你仔细看注释，并对这个函数进行一些思考。

Listing 11: 地址空间初始化

```

1  /* Overview:
2   * Initialize the kernel virtual memory layout for environment e.
3   * Allocate a page directory, set e->env_pgdir and e->env_cr3
4   * accordingly, and initialize the kernel portion of the
5   * new environment's address space.
6   * Do NOT map anything into the user portion of the environment's
7   * virtual address space.
8   */
9  static int
10 env_setup_vm(struct Env *e)
11 {
12
13     int i, r;
14     struct Page *p = NULL;
15     Pde *pgdir;
16
17     /*Step 1: Allocate a page for the page directory and add its reference.
18      *pgdir is the page directory of Env e. */
19     if ((r = page_alloc(&p)) < 0) {
20         panic("env_setup_vm - page_alloc error\n");
21         return r;
22     }
23     p->pp_ref++;
24     pgdir = (Pde *)page2kva(p);
25
26     /*Step 2: Zero pgdir's field before UTOP. */
27     for (i = 0; i < PDX(UTOP); i++) {
28         pgdir[i] = 0;
29     }
30
31     /*Step 3: Copy kernel's boot_pgdir to pgdir. */
32
33     /* Hint:
34      * The VA space of all envs is identical above UTOP
35      * (except at VPT and UVPT, which we've set below).
36      * See ./include/mmu.h for layout.
37      * Why you use boot_pgdir as a template?
38      */
39     for (i = PDX(UTOP); i <= PDX(~0); i++) {
40         pgdir[i] = boot_pgdir[i];
41     }
42     e->env_pgdir = pgdir;
43     e->env_cr3 = PADDR(pgdir);
44
45     /*Step 4: VPT and UVPT map the env's own page table, with
46      *different permissions. */
47     e->env_pgdir[PDX(VPT)] = e->env_cr3;
48     e->env_pgdir[PDX(UVPT)] = e->env_cr3 | PTE_V | PTE_R;
49     return 0;
50 }

```

请在仔细看完上述函数的注释和其中的 Hint 后，来思考一下下面这些为你准备的问题

**Thinking 3.2** 思考 `env_setup_vm` 函数:

- 第三点注释中的问题: 为什么我们要执行 `pgdir[i] = boot_pgdir[i]` 这个赋值操作? 换种说法, 我们为什么要使用 `boot_pgdir` 作为一部分模板? (提示: mips 虚拟空间布局)
- `UTOP` 和 `ULIM` 的含义分别是什么, 在 `UTOP` 到 `ULIM` 的区域与其他用户区相比有什么最大的区别?
- (选做) 我们为什么要让 `pgdir[PDX(UVPT)] = env_cr3`? (提示: 结合系统自映射机制)

其实第三点注释问题本身要思考起来是有比较大跨度的, 所以我们直接在这里给出笔者在反复思考与理解后所获得的心得, 鼓励你在写文档时可以进行更多思考与更深层的理解: )。

在我们的实验中, 对于不同的进程而言, 虚拟地址 `ULIM` 以上的地方, 虚拟地址到物理地址的映射关系都是一样的。因为这 2G 虚拟空间, 不是由哪个进程管理的, 而是由内核管理的! 如果你仔细思索这句话, 可能会产生疑惑: “那为什么不能该内核管的时候让内核进程去管, 该普通进程管的时候给普通进程去管, 非要混在一个地址空间里搞来搞去的呢?” 这种想法是很好的, 但是问题来了, 在我们这种布局模式下, 有严格意义上的内核进程吗?

答案当然是否定的, 这里我们要再讲清楚几个概念, 方可解决这个问题。

首先来科普下虚拟空间的分配模式。我们知道, 每一个进程都有 4G 的逻辑地址可以访问, 我们所熟知的系统不管是 Linux 还是 Windows 系统, 都可以支持 3G/1G 模式或者 2G/2G 模式。3G/1G 模式即满 32 位的进程地址空间中, 用户态占 3G, 内核态占 1G。这些情况在进入内核态的时候叫做陷入内核, 因为**即使进入了内核态, 还处在同一个地址空间中, 并不切换 CR3 寄存器**。但是! 还有一种模式是 4G/4G 模式, 内核单独占有一个 4G 的地址空间, 所有的用户进程独享自己的 4G 地址空间, 这种模式下, 在进入内核态的时候, 叫做切换到内核, **因为需要切换 CR3 寄存器**, 所以进入了不同的地址空间!

**Note 3.2.2** 用户态和内核态的概念相信大家也不陌生, 内核态即计组实验中所提到的特权态, 用户态就是非特权态。mips 汇编中使用一些特权指令如 `mtc0`、`mfc0`、`syscall` 等都会陷入特权态 (内核态)。

而我们这次实验, 根据 `./include/mmu.h` 里面的布局来说, 我们是 2G/2G 模式, 用户态占用 2G, 内核态占用 2G。接下来, 也是最容易产生混淆的地方, 进程从用户态提升到内核态的过程, 操作系统中发生了什么?

是从当前进程切换成一个所谓的“内核进程”来管理一切了吗? 不! 还是一样的配方, 还是一样的进程! 改变的只是进程的权限! 我们打一个比方, 你所在的班级里没有

班长，任何人都可以以合理的理由到老师那申请当临时班长。你是班里的成员吗？当然是的。某天你申请当临时班长，申请成功了，那么现在你还是班里的成员吗？当然还是的。那么你前后有什么区别呢？是权限的变化。可能你之前和其他成员是完全平等，互不干涉的。那么现在你可以根据花名册点名，你可以安排班里的成员做些事情，你可以开班长会议等等。那么我们能说你是班长吗？不能，因为你并不是永久的班长。但能说拥有成为班长的资格吗？当然可以，这种**成为临时班长的资格**，我们可以粗略地认为它就是内核态的精髓所在。

而在操作系统中，每个完整的进程都拥有这种成为临时内核的资格，即所有的进程都可以发出申请变成内核态下运行的进程。内核态下进程可访问的资源更多，更加自由。在之后我们会提到一种“申请”的方式，就叫做“系统调用”。

那么你现在应该能够理解为什么我们要将内核才能使用的虚页表为每个进程都拷贝一份了，在 2G/2G 这种布局模式下，每个进程都会有 **2G 内核态** 的虚拟地址，以便临时变身为“内核”行使大权。但是，在变身器使用之前，就算是奥特曼，一样也只能访问自己的那 2G 用户态的虚拟地址。

那么这种微妙的关系应该类似于下面这种：（图中灰色代表不可用，白色代表可用）

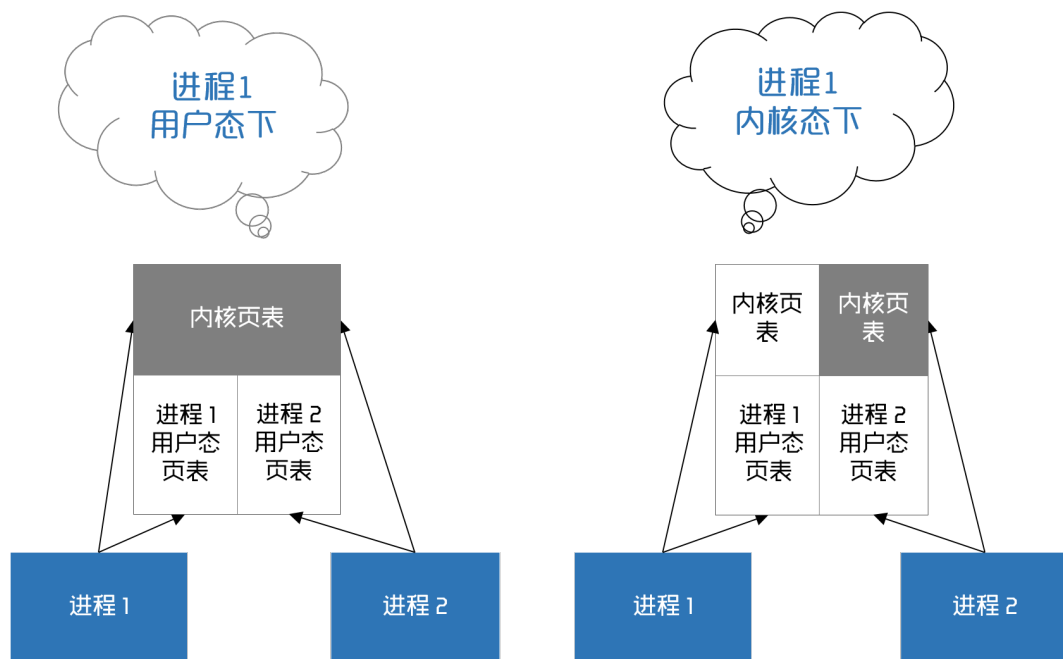


Figure 3.1: 进程页表与内核页表的关系

在上述的思考完成后，那么我们就可以直接在 `env__alloc` 第二步直接使用该函数了。现在来解决一下刚才的吐槽，第三点所说的合适的值是什么？我们要设定哪些变量的值呢？

我们要设定的变量其实在 `env__alloc` 函数的提示中已经说的很清楚了，至于其合适的值，相信仔细的你从函数的前面长长的注释里一定能获得足够的信息。当然我要讲的重点不在这里，重点在我们已经给出的这个设置 `e->env_tf.cp0_status = 0x10001004;`;

这个设置很重要，很重要，很重要，重要到我们必须直接在代码中给出。为什么说

它重要，各位看官且听我娓娓道来。

### SR Register

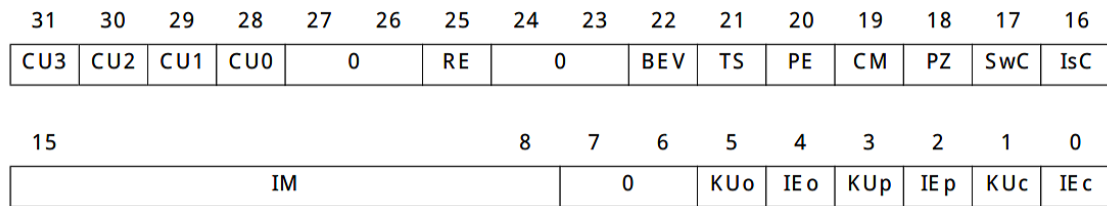


Figure 3.2: R3000 的 SR 寄存器示意图

图3.2是我们 MIPS R3000 里的 SR(status register) 寄存器示意图，就是我们在 `env_tf` 里的 `cp0_status`。

第 28bit 设置为 1，表示处于用户模式下。

第 12bit 设置为 1，表示 4 号中断可以被响应。

这些都是小 case，下面讲的才是重点！

R3000 的 SR 寄存器的低六位是一个二重栈的结构。KUo 和 IEo 是一组，每当中断发生的时候，硬件自动会将 KUp 和 IEp 的数值拷贝到这里；KUp 和 IEp 是一组，当中断发生的时候，硬件会把 KUc 和 IEc 的数值拷贝到这里。

其中 KU 表示是否位于内核模式下，为 1 表示位于内核模式下；IE 表示中断是否开启，为 1 表示开启，否则不开启<sup>2</sup>。

而每当 `rfe` 指令调用的时候，就会进行上面操作的逆操作。下面这一段代码在运行第一个进程前是一定要执行的，所以就一定会执行 `rfe` 这条指令。

```

1  lw  k0,TF_STATUS(k0)      # 恢复 CP0_STATUS 寄存器
2  mtc0 k0,CP0_STATUS
3  j   k1
4  rfe

```

现在你可能就懂了为何我们 `status` 后六位是设置为 `000100b` 了。当运行第一个进程前，运行上述代码到 `rfe` 的时候，就会将 KUp 和 IEp 拷贝回 KUc 和 IEc，令 `status` 为 `000001b`，最后两位 KUc,IEc 为 `[0,1]`，表示开启了中断。之后第一个进程成功运行，这时操作系统也可以正常响应中断，Nice！

**Note 3.2.3** 关于 MIPS R3000 版本 SR 寄存器功能的英文原文描述：The status register is one of the more important registers. The register has several fields. The current Kernel/User (KUc) flag states whether the CPU is in kernel mode. The current Interrupt Enable (IEc) flag states whether external interrupts are turned on. If cleared then external interrupts are ignored until the flag is set again. In an exception these flags are copied to previous Kernel/User and Interrupt Enable (KUp and IEp) and then cleared so the system moves to a kernel mode with external interrupts disabled. The Return From Exception instruction writes the previous flags to the current flags.

<sup>2</sup>我们的实验是不支持中断嵌套的，所以内核态时是不可以开启中断的。

当然我们从注释也能看出，第四步除了需要设置`cp0status`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 3.3** 根据上面的提示与代码注释，填写 `env__alloc` 函数。 ■

### 3.2.3 加载二进制镜像

我们继续来完成我们的实验。下面这个函数还是蛮难填的呢，所以大家一定要紧跟我的步伐，我们慢慢来分析一下这个函数。

我们在进程创建第三点曾提到过，我们需要为**新进程的程序**分配空间来容纳程序代码。那么下面我需要有二个函数来一起完成这个任务

Listing 12: 加载镜像映射

```

1  /* Overview:
2   *   This is a call back function for kernel's elf loader.
3   *   Elf loader extracts each segment of the given binary image.
4   *   Then the loader calls this function to map each segment
5   *   at correct virtual address.
6   *
7   *   `bin_size` is the size of `bin`. `sgsize` is the
8   *   segment size in memory.
9   *
10  * Pre-Condition:
11  *   va aligned 4KB and bin can't be NULL.
12  *
13  * Post-Condition:
14  *   return 0 on success, otherwise < 0.
15  */
16  static int load_icode_mapper(u_long va, u_int32_t sgsz,
17                               u_char *bin, u_int32_t bin_size, void *user_data)
18  {
19      struct Env *env = (struct Env *)user_data;
20      struct Page *p = NULL;
21      u_long i;
22      int r;
23
24      /*Step 1: load all content of bin into memory. */
25      for (i = 0; i < bin_size; i += BY2PG) {
26          /* Hint: You should alloc a page and increase the reference count of it. */
27
28
29      }
30      /*Step 2: alloc pages to reach `sgsz` when `bin_size` < `sgsz`. */
31      while (i < sgsz) {
32
33
34      }
35      return 0;
36  }

```

为了完成这个函数，我们接下来再补充一点关于 ELF 的知识。

前面在讲解内核加载的时候，我们曾简要说明过 ELF 的加载过程。这里，我们再做一些补充说明。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 segment 加载到对应的虚地址上即可。为了降低难度，我们已经写好了用于解析 ELF 文件的代码 (lib/kernel\_elfloader.c)，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。该函数的原型如下：

```
1 // binary 为整个待加载的 ELF 文件。size 为该 ELF 文件的大小。
2 // entry_point 是一个 u_long 变量的地址（相当于引用），解析出的入口地址会被存入到该位置
3 int load_elf(u_char *binary, int size, u_long *entry_point, void *user_data,
4             int (*map)(u_long va, u_int32_t sgsz,
5                       u_char *bin, u_int32_t bin_size, void *user_data))
```

我们来着重解释一下 load\_elf() 函数的设计，以及最后两个参数的作用。为了让你有机会完成加载可执行文件到内存的过程，load\_elf() 函数只完成了解析 ELF 文件的部分，而把将 ELF 文件的各个 segment 加载到内存的工作留给了你。为了达到这一目标，load\_elf() 的最后两个参数用于接受一个你的自定义函数以及你想传递给你的自定义函数的额外参数。每当 load\_elf() 函数解析到一个需要加载的 segment，会将 ELF 文件里与加载有关的信息作为参数传递给你的自定义函数。你的自定义函数完成加载单个 segment 的过程。

为了进一步简化你的理解难度，我们已经为你定义好了这个“自定义函数”的框架。如代码12所示。load\_elf() 函数会从 ELF 文件中解析出每个 segment 的四个信息：va(该段需要被加载到的虚地址)、sgsz(该段在内存中的大小)、bin(该段在 ELF 文件中的内容)、bin\_size(该段在文件中的大小)，并将这些信息传给我们的“自定义函数”。

接下来，你只需要完成以下两个步骤：

**第一步** 加载该段在 ELF 文件中的所有内容到内存。

**第二步** 如果该段在文件中的内容的大小达不到该段在内存中所应有的大小，那么余下的部分用 0 来填充。

也许机灵的你发现了一个很无语的情况：我们并没有真正解释清楚 user\_data 这个参数的作用。最后一个参数是一个函数指针，用于将我们的自定义函数传入进去。但这个诡异的 user\_data 到底是做什么用的呢？这样设计又是为了什么呢？很不幸，这个问题我们决定留给你来思考。

**Thinking 3.3** 思考 user\_data 这个参数的作用。没有这个参数可不可以？为什么？（如果你能说明哪些应用场景中可能会应用这种设计就更好了。可以举一个实际的库中的例子） ■

思考完这一点，我们就可以进入这一小节的练习部分了。

**Exercise 3.4** 通过上面补充的知识与注释，填充 load\_\_ icode\_\_ mapper 函数。 ■

现在我们已经完成了补充部分最难的一个函数，那么下面我们完成这个函数后，就能真正实现把二进制镜像加载进内存的任务了。



Listing 13: 完整加载镜像

```

1  /* Overview:
2   * Sets up the the initial stack and program binary for a user process.
3   * This function loads the complete binary image by using elf loader,
4   * into the environment's user memory. The entry point of the binary image
5   * is given by the elf loader. And this function maps one page for the
6   * program's initial stack at virtual address USTACKTOP - BY2PG.
7   *
8   * Hints:
9   * All mappings are read/write including those of the text segment.
10  * You may use these :
11  *     page_alloc, page_insert, page2kva , e->env_pgdir and load_elf.
12  */
13  static void
14  load_icode(struct Env *e, u_char *binary, u_int size)
15  {
16      /* Hint:
17       * You must figure out which permissions you'll need
18       * for the different mappings you create.
19       * Remember that the binary image is an a.out format image,
20       * which contains both text and data.
21       */
22      struct Page *p = NULL;
23      u_long entry_point;
24      u_long r;
25      u_long perm;
26
27      /*Step 1: alloc a page. */
28
29      /*Step 2: Use appropriate perm to set initial stack for new Env. */
30      /*Hint: The user-stack should be writable? */
31
32      /*Step 3:load the binary by using elf loader. */
33
34      /***Your Question Here***/
35      /*Step 4:Set CPU's PC register as appropriate value. */
36      e->env_tf.pc = entry_point;
37  }

```

现在我们来根据注释一步一步完成这个函数。在第二步我们要用第一步申请的页面来初始化一个进程的栈，根据注释你应当可以轻松完成。这里我们只讲第三步的注释所代表的内容，其余你可以根据注释中的提示来完成。

第三步通过调用我们预先为你准备好的 `load_elf()` 函数来将 ELF 文件真正加载到内存中。这里仅做一点提醒：请将 `load_icode_mapper()` 这个函数作为参数传入到 `load_elf()` 中。其余的参数在前面已经解释过了，就不再赘述了。

**Exercise 3.5** 通过补充的 ELF 知识与注释，填充 `load_icode` 函数。 ■

这里的 `e->env_tf.pc` 是什么呢？就是在我们计组中反复强调的甚为重要的 PC。它指示着进程当前指令所处的位置。你应该知道，冯诺依曼体系结构的一大特点就是：程序预存储，计算机自动执行。我们要运行的进程的代码段预先被载入到了 `entry_point`



为起点的内存中，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 3.4** 思考上面这一段话，并根据自己在 **lab2** 中的理解，回答：

- 我们这里出现的“指令位置”的概念，你认为该概念是针对虚拟空间，还是物理内存所定义的呢？
- 你觉得 `entry_point` 其值对于每个进程是否一样？该如何理解这种统一或不同？
- 从布局图中找到你认为最有可能是 `entry_point` 的值。

思考完这一点后，下面我们来准备准备可以真正创建进程了。

### 3.2.4 创建进程

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，并将二进制代码载入到对应地址空间即可完成。好好思考上面的函数，我们需要用到哪些函数来做这几件事？

**Exercise 3.6** 根据提示，完成 `env_create` 函数的填写。

当然提到创建进程，我们还需要提到一个封装好的宏命令

```
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); \
7  }
```

这个宏里的语法大家可能以前没有见过，这里解释一下 `##` 代表拼接，例如<sup>3</sup>

```
1  #define CONS(a,b) int(a##e##b)
2  int main()
3  {
4      printf("%d\n", CONS(2,3)); // 2e3 输出:2000
5      return 0;
6  }
```

好，那么现在我们就得手工在我们的 `init/init.c` 里面加两句话来初始化创建两个进程

```
1  ENV_CREATE(user_A);
2  ENV_CREATE(user_B);
```

这两句话加在哪里呢？那就需要你翻代码来寻找啦～

<sup>3</sup>这个例子是转载的，出处为<http://www.cnblogs.com/hnrain11/archive/2012/08/15/2640558.html>，想深入了解的同学可以参考这篇博客

**Exercise 3.7** 根据注释与理解，将上述两条进程创建命令加入 `init/init.c` 中。 ■

做完这些，是不是迫不及待地想要跑个进程看看能否成功？等我们完成下面这个函数，就可以开始第一部分的自我测试了！

### 3.2.5 进程运行与切换

Listing 14: 进程的运行

```

1  extern void env_pop_tf(struct Trapframe *tf, int id);
2  extern void lcontext(u_int context);
3
4  /* Overview:
5   * Restores the register values in the Trapframe with the
6   * env_pop_tf, and context switch from curenv to env e.
7   *
8   * Post-Condition:
9   * Set 'e' as the curenv running environment.
10  *
11  * Hints:
12  * You may use these functions:
13  *     env_pop_tf and lcontext.
14  */
15  void
16  env_run(struct Env *e)
17  {
18      /*Step 1: save register state of curenv. */
19      /* Hint: if there is a environment running, you should do
20       * context switch. You can imitate env_destroy() 's behaviors.*/
21
22
23      /*Step 2: Set 'curenv' to the new environment. */
24
25
26      /*Step 3: Use lcontext() to switch to its address space. */
27
28
29      /*Step 4: Use env_pop_tf() to restore the environment's
30       * environment registers and drop into user mode in the
31       * the environment.
32       */
33      /* Hint: You should use GET_ENV_ASID there. Think why? */
34
35  }
```

刚刚说到的 `load_icode` 是为数不多的坑函数之一，`env_run` 也是，而且其实按程度来讲可能更甚一筹。

`env_run`，是进程运行使用的基本函数，它包括两部分：

- 一部分是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)
- 另一部分就是恢复要启动的进程的上下文，然后运行该进程。

**Note 3.2.4** 进程上下文说来就是一个环境，相对于进程而言，就是进程执行时的环境。具体来说就是各个变量和数据，包括所有的寄存器变量、内存信息等。

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，人如其名，就是当前进程停下工作，让出 CPU 处理器来运行另外的进程。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？键盘告诉我们：Alt+Tab。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？理所当然地想想，你可能会想到下面两种需要保存的状态：

### 进程本身的状态

### 进程周围的环境状态

那么我们先解决一个问题，进程本身的状态需要记录吗？进程本身的状态无非就是进程块里面那几个东西，包括

`env__id, env__parent__id, env__pgdir, env__cr3...`

或许你会有所疑问，`env__tf` 不算是进程本身的状态吗？按笔者的理解来说，是不算的。`env__tf` 保存的是进程上下文，相当于我们的第二点，进程周围的环境状态。我们仔细思索一下，就能发现，进程本身的状态在进程切换的时候是不会变化的。（我们不会去毁灭一个进程块，进程块跟我们又没仇。）会变的也是需要我们保存的实际上是进程的环境信息。

这样或许你就能明白 `run` 代码中的第一句注释的含义了：/\*Step 1: save register state of curenv. \*/

那么你可能会想，进程运行到某个时刻，它的上下文——所谓的 CPU 的寄存器在哪呢？我们又该如何保存？在 `lab3` 中，我们在本实验里的寄存器状态保存的地方是 `TIMESTACK` 区域。

```
struct Trapframe *old;
```

```
old = (struct Trapframe *) (TIMESTACK - sizeof(struct Trapframe));
```

这个 `old` 就是当前进程的上下文所存放的区域。那么第一步注释还说到，让我们参考 `env__destroy`，其实就是让我们把 `old` 区域的东西拷贝到当前进程的 `env__tf` 中，以达到保存进程上下文的效果。那么我们还有一点很关键，就是当我们返回到该进程执行的时候，应该从哪条指令开始执行？即当前进程上下文中的 `pc` 应该设为什么值？这留给聪明的你去思考。

**Thinking 3.5** 思考一下，要保存的进程上下文中的 `env__tf.pc` 的值应该设置为多少？为什么要这样设置？ ■

思考完上面的，我们沿着注释再一路向下，后面好像没有什么很难的地方了。根据提示也完全能够做出来。但是我们还有一点坑没填，我们忽略了 `env__pop__tf` 函数。还记得我们之前曾说的那个一定会在第一个进程运行前执行的 `rfe` 吗？既然提到它了，

那么请仔细地看看这个函数，并且仔细观察，你或许能看出什么关于 TIMESTACK 的端倪。TIMESTACK 问题可能将是你在本实验中需要思考时间最久的问题，希望你能和小伙伴积极交流，努力寻找实验代码来支撑你的看法与观点，鼓励提出新奇的想法！

**Thinking 3.6** 思考 TIMESTACK 的含义，并找出相关语句与证明来回答以下关于 TIMESTACK 的问题：

- 请给出一个你认为合适的 TIMESTACK 的定义
- 请为你的定义在实验中找到合适的代码段作为证据 (请对代码段进行分析)
- 思考 TIMESTACK 和第 18 行的 KERNEL\_SP 的含义有何不同

**Exercise 3.8** 根据补充说明，填充完成 `env__run` 函数。

至此，我们第一部分的工作已经完成了！第二部分的代码量很少，但是不可或缺！休息一下，我们继续奋斗！

### 3.3 中断与异常

之前我们在学习计组的时候已经学习了异常和中断的概念，所以这里我们不再将概念作为主要介绍内容。

**Note 3.3.1** 我们实验里认为凡是引起控制流突变的都叫做异常，而中断仅仅是异常的一种，并且是仅有的一种异步异常。

我们可以通过一个简单的图来认识一下异常的产生与返回（见图3.3）。

#### 3.3.1 异常的分发

每当发生异常的时候，一般来说，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，这个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。

代码3.3.1就是异常分发代码，我们先将下面代码填充到我们的`start.S`的开头，然后我们分析一下。

```
1  .section .text.exc_vec3
2  NESTED(exception_vec3, 0, sp)
3      .set noat
4      .set noreorder
5  1:
6      mfc0 k1, CP0_CAUSE
7      la    k0, exception_handlers
8      andi k1, 0x7c
```

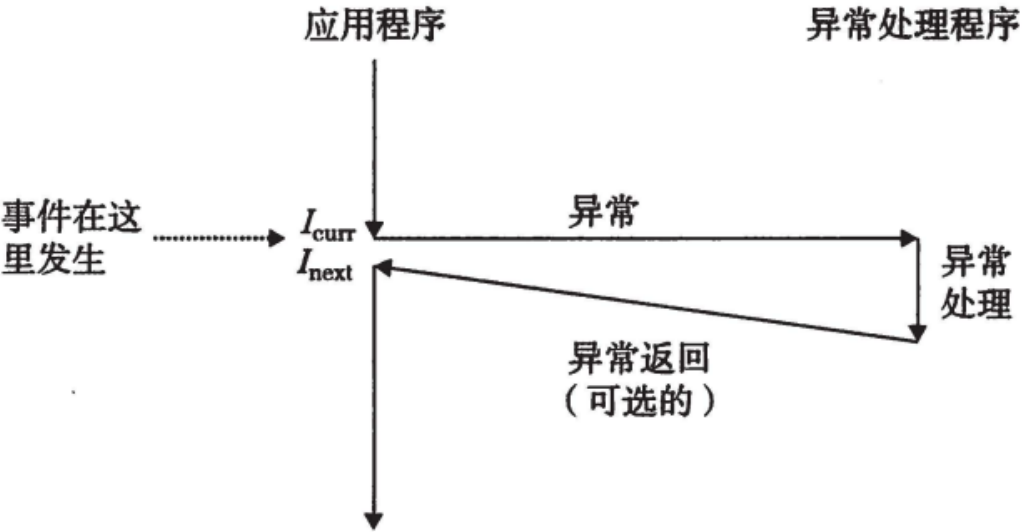


Figure 3.3: 异常处理图示

```
9      addu k0,k1
10     lw   k0,(k0)
11     NOP
12     jr   k0
13     nop
14     END(exception_vec3)
15     .set at
```

**Exercise 3.9** 将异常分发代码填入 `init/start.S` 合适的部分。 ■

这段异常分发代码的作用流程简述如下：

- 1. 取得异常码，这是区别不同异常的重要标志。
- 2. 以得到的异常码作为索引去 `exception_handlers` 数组中找到对应的中断处理函数，后文中会有涉及。
- 3. 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

Cause Register

31	30	29	28	27	16	15	8	7	6	2	1	0
BD	0	CE		0		IP		0	ExcCode		0	

Figure 3.4: CR 寄存器

图3.4是 MIPS3000 中 Cause Register 寄存器。其中保存着 CPU 中哪一些中断或者异常已经发生。bit2 bit6 保存着异常码，也就是根据异常码来识别具体哪一个异常

发生了。bit8 bit15 保存着哪一些中断发生了。其他的一些位含义在此不涉及，可参看 MIPS 开发手册。

这个 `.text.exec_vec3` 段将通过链接器放到特定的位置，在 R3000 中要求是放到 `0x80000080` 地址处，这个地址处存放的是异常处理程序的入口地址。一旦 CPU 发生异常，就会自动跳转到 `0x80000080` 地址处，开始执行，下面我们将 `.text.exec_vec3` 放到该位置，一旦异常发生，就会引起该段代码的执行，而该段代码就是一个分发处理异常的过程。

所以我们要在我们的 `lds` 中增加这么一段，从而可以让 R3000 出现异常时自动跳转到异常分发代码处。

```
1  . = 0x80000080;
2  .except_vec3 : {
3      *(.text.exc_vec3)
4  }
```

**Exercise 3.10** 将 `lds` 代码补全使得异常后可以跳到异常分发代码。 ■

### 3.3.2 异常向量组

我们刚才提到了异常的分发，要寻找到 `exception_handlers` 数组中的中断处理函数，而 `exception_handlers` 就是所谓的异常向量组。

下面我们跟随 `trap_init(lib/traps.c)` 函数来看一下，异常向量组里存放的是些什么？

```
1  extern void handle_int();
2  extern void handle_reserved();
3  extern void handle_tlb();
4  extern void handle_sys();
5  extern void handle_mod();
6  unsigned long exception_handlers[32];
7
8  void trap_init()
9  {
10     int i;
11
12     for (i = 0; i < 32; i++) {
13         set_except_vector(i, handle_reserved);
14     }
15
16     set_except_vector(0, handle_int);
17     set_except_vector(1, handle_mod);
18     set_except_vector(2, handle_tlb);
19     set_except_vector(3, handle_tlb);
20     set_except_vector(8, handle_sys);
21 }
22 void *set_except_vector(int n, void *addr)
23 {
24     unsigned long handler = (unsigned long)addr;
25     unsigned long old_handler = exception_handlers[n];
26     exception_handlers[n] = handler;
27     return (void *)old_handler;
28 }
```

实际上呢，这个函数实现了对全局变量 `exception_handlers[32]` 数组初始化的工作，其实就是把相应的处理函数的地址填到对应数组项中。主要初始化

- 0 号异常 的处理函数为 `handle_int`,
- 1 号异常 的处理函数为 `handle_mod`,
- 2 号异常 的处理函数为 `handle_tlb`,
- 3 号异常 的处理函数为 `handle_tlb`,
- 8 号异常 的处理函数为 `handle_sys`,

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。而在我们的实验中，我们主要是要使用 0 号异常，即中断异常的处理函数。因为我们接下来要做的，就是要产生时钟中断。

### 3.3.3 时钟中断

希望你还没有忘记在计组实验中所接触到的**定时器**这个概念。或许你当时对定时器的作用会有疑惑，为了防止你继续迷糊不清，我们下面来简单介绍一下时钟中断的概念。

时钟中断和操作系统的时间片轮转算法是紧密相关的。时间片轮转调度是一种很公平的算法。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。那么 CPU 是如何知晓一个进程的时间片结束的呢？就是通过定时器产生的时钟中断。当时钟中断产生时，当前运行的进程被挂起，我们需要在调度队列中选取一个合适的进程运行。如何“选取”，就要涉及到进程的调度了。

要产生时钟中断，我们不仅要了解中断的产生与处理。我们还要了解 `gxemul` 是如何模拟时钟中断的。初始化时钟主要是在 `kclock_init` 函数中，该函数主要调用 `set_timer` 函数，完成如下操作：

- 首先向 `0xb5000100` 位置写入 1，其中 `0xb5000000` 是模拟器 (`gxemul`) 映射实时钟的位置。偏移量为 `0x100` 表示来设置实时钟中断的频率，1 则表示 1 秒钟中断 1 次，如果写入 0，表示关闭实时钟。实时钟对于 R3000 来说绑定到了 4 号中断上，故这段代码其实主要用来触发了 4 号中断。注意这里的中断号和异常号是不一样的概念，我们实验的异常包括中断。
- 一旦实时钟中断产生，就会触发 MIPS 中断，从而 MIPS 将 PC 指向 `0x80000080`，从而跳转到 `.text.exc_vec3` 代码段执行。对于实时钟引起的中断，通过 `text.exc_vec3` 代码段的分发，最终会调用 `handle_int` 函数来处理实时钟中断。
- 在 `handle_int` 判断 `CP0_CAUSE` 寄存器是不是对应的 4 号中断位引发的中断，如果是，则执行中断服务函数 `timer_irq`。
- 在 `timer_irq` 里直接跳转到 `sched_yield` 中执行。而这个函数就是我们将要补充的调度函数。



以上就是我们时钟中断的产生与中断处理流程，我们在这里要完成下面的任务以顺利产生时钟中断。

**Exercise 3.11** 通过上面的描述，补充 `kclock_init` 函数。 ■

### 3.3.4 进程调度

通过上面的描述，我们知道了，其实在时钟中断产生时，最终是调用了 `sched_yield` 函数来进行进程的调度，这个函数在 `lib/sched.c` 中所定义。这个函数就是我们本次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，没有优先级，根据注释就可以轻松写出。

**Exercise 3.12** 根据注释，完成 `sched_yield` 函数的补充。 ■

至此，我们的实验三就算是圆满完成了。

## 3.4 实验正确结果

如果你按流程做下来并且做的结果正确的话，你运行之后应该会出现这样的结果

```

1  init.c: mips_init() is called
2
3  Physical memory: 65536K available, base = 65536K, extended = 0K
4
5  to memory 80401000 for struct page directory.
6
7  to memory 80431000 for struct Pages.
8
9  mips_vm_init:boot_pgdir is 80400000
10
11 pmap.c: mips vm init success
12
13 panic at init.c:27: ~~~~~
14
15  2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
16  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
17  2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
18  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
19  2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
20  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

```

当然不会这么整齐，且没有换行，只是交替输出 2 和 1 而已～当然还不能放过你，你还需要再思考一部分内容

**Thinking 3.7** 思考一下你的调度程序，这种调度方式由于某种不可避免的缺陷而造成对进程的不公平。

- 这种不公平是如何产生的？
- 如果实验确定只运行两个进程，你如何改进可以降低这种不公平？

### 3.5 实验思考

- 思考 `-init` 的逆序插入
- 思考 `-地址空间初始化`
- 思考 `-user_data` 的作用
- 思考 `-位置` 的含义
- 思考 `-进程上下文的 PC 值`
- 思考 `-TIMESTACK` 的含义
- 思考 `-不公的调度`

### 3.6 挑战性任务

我们实验中使用的调度算法是很简单的，那么现在希望你可以向内核中添加自己的调度策略，比如带优先级的时间片轮转算法。修改调度算法，并利用测试的现象说明调度算法是正确的。如果你做了这部分的挑战性任务，请将代码提交到分支 **lab3-challenge** 并推送到服务器上，并在实验文档中写入新的调度策略对于结果的影响并说明为什么，可以自定义测试程序。

## CHAPTER 4

---

## 系统调用与 FORK

### 4.1 实验目的

1. 掌握系统调用的概念及流程
2. 实现进程间通讯机制
3. 实现 fork 函数

在系统中真正被所有进程都使用的内核通信方式是系统调用，例如当进程请求内核服务时，就使用的是系统调用。一般情况下，进程是不能够存取系统内核的，它不能存取内核使用的内存段，也不能调用内核函数，CPU 的硬件结构保证了这一点，只有系统调用是一个例外。进程使用寄存器中适当的值跳转到内核中事先定义好的代码中执行（当然，这些代码是只读的）。在这一节的实验中，我们需要实现系统调用机制，并再此基础上实现进程间通信（ipc）机制和 fork。

### 4.2 系统调用 (System Call)

本节中，我们着重讨论系统调用的作用，并完成实现相关的内容。

#### 4.2.1 一探到底，系统调用的来龙去脉

说起系统调用，你冒出的第一个问题一定是：系统调用到底长什么样子？为了一探究竟，我们选择一个即为简单的程序作为实验对象。在这个程序中，我们通过 puts 来输出一个字符串到终端。

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

**Note 4.2.1** 如果你还记得 C 语言课上关于标准输出的相关知识的话，你一定知道在 C 语言中，终端被抽象为了标准输出文件 stdout。通过向标准输出文件写东西，就可以输出内容到屏幕。而向文件写入内容是通过 write 系统调用完成的。因此，我们选择通过观察 puts 函数，来探究系统调用的奥秘。

我们通过 GDB 进行单步调试，逐步深入到函数之中，观察 puts 具体的调用过程<sup>1</sup>。运行 GDB，将断点设置在 puts 这条语句上，并通过 stepi 指令<sup>2</sup> 单步进入到函数中。当程序到达 write 函数时停下，因为 write 正是 Linux 的一条系统调用。我们打印出此时的函数调用栈，可以看出，C 标准库中的 puts 函数实际上通过了很多层函数调用，最终调用到了底层的 write 函数进行真正的屏幕打印操作。

```

1 (gdb)
2 0x00007ffff7b1b4e0 in write () from /lib64/libc.so.6
3 (gdb) backtrace
4 #0 0x00007ffff7b1b4e0 in write () from /lib64/libc.so.6
5 #1 0x00007ffff7ab340f in _IO_file_write () from /lib64/libc.so.6
6 #2 0x00007ffff7ab2aa3 in ?? () from /lib64/libc.so.6
7 #3 0x00007ffff7ab4299 in _IO_do_write () from /lib64/libc.so.6
8 #4 0x00007ffff7ab462b in _IO_file_overflow () from /lib64/libc.so.6
9 #5 0x00007ffff7ab5361 in _IO_default_xsputn () from /lib64/libc.so.6
10 #6 0x00007ffff7ab3992 in _IO_file_xsputn () from /lib64/libc.so.6
11 #7 0x00007ffff7aaa4ef in puts () from /lib64/libc.so.6
12 #8 0x000000000400564 in main () at test.c:4

```

通过 gdb 显示的信息，我们可以看到，这个 write() 函数是在 libc.so 这个动态链接库中的，也就是说，它仍然是 C 库中的函数，而不是内核中的函数。因此，该 write() 函数依旧是个用户空间函数。为了彻底揭开这个函数的秘密，我们对其进行反汇编。

```

1 (gdb) disassemble 0x00007ffff7b1b4e0
2 Dump of assembler code for function write:
3 => 0x00007ffff7b1b4e0 <+0>:    cmpl    $0x0,0x2bf759(%rip)          # 0x7ffff7ddac40
4      0x00007ffff7b1b4e7 <+7>:    jne     0x7ffff7b1b4f9 <write+25>
5      0x00007ffff7b1b4e9 <+9>:    mov     $0x1,%eax
6      0x00007ffff7b1b4ee <+14>:   syscall
7      0x00007ffff7b1b4f0 <+16>:   cmp     $0xffffffffffffffff001,%rax
8      0x00007ffff7b1b4f6 <+22>:   jae     0x7ffff7b1b529 <write+73>
9      0x00007ffff7b1b4f8 <+24>:   retq
10 End of assembler dump.

```

通过 gdb 的反汇编功能，我们可以看到，这个函数最终执行了 syscall 这个极为特殊的指令。从它的名字我们就能够猜出它的用途，它使得进程陷入到内核态中，执行内核中的相应函数，完成相应的功能。在系统调用返回后，用户空间的相关函数会将系统调用的结果，通过一系列的过程，最终返回给用户程序。

由此我们可以看到，系统调用实际上是操作系统和用户空间的一组接口。用户空间的程序通过系统调用来访问操作系统的一些服务，谋求操作系统提供必要的帮助。

在进行了上面的一系列探究后，我们将我们的发现罗列出来，整理一下我们的思路：

<sup>1</sup>这里为了方便大家在自己的机器上重现，我们选用了 Linux X86\_64 平台作为实验平台

<sup>2</sup>为了加快调试进程，可以尝试 stepi N 指令，N 的位置填写任意数字均可。这样每次会执行 N 条机器指令。笔者使用的是 stepi 10。

- 存在一些只能由操作系统来完成的操作（如读写设备、创建进程等）。
- 用户程序要实现一些功能（比如执行另一个程序、读写文件），必须依赖操作系统的帮助。
- C 标准库中的一些函数的实现必须依赖于操作系统（如我们所探究的 `puts` 函数）。
- 通过执行 `syscall` 指令，我们可以陷入到内核态，请求操作系统的一些服务。
- 直接使用操作系统的功能是很繁复的（每次都需要设置必要的寄存器，并执行 `syscall` 指令）

之后，我们再来整理一下调用 C 标注库中的 `puts` 函数的过程中发生了什么：

1. 调用 `puts` 函数
2. 在一系列的函数调用后，最终调用了 `write` 函数。
3. `write` 函数设置了为寄存器设置了相应的值，并执行了 `syscall` 指令。
4. 进入内核态，内核中相应的函数或服务被执行。
5. 回到用户态的 `write` 函数中，将系统调用的结果从相关的寄存器中取回，并返回。
6. 再次经过一系列的返回过程后，回到了 `put` 函数中。
7. `puts` 函数返回。

综合上面这些内容，相信你一定已经发现了其中的巧妙之处。操作系统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。用户程序即可通过操作系统来完成一些特殊的操作。同时，所有的特殊操作就全部在操作系统的掌控之中了（因为用户程序只能通过由操作系统提供的系统调用来完成这些操作，所以操作系统可以确保用户不破坏系统的稳定）。而直接使用这些系统调用较为麻烦，于是由产生了用户空间的一系列 API，如 POSIX、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，使得程序更易于在不同的系统间移植。整个结构如图4.1所示。

#### 4.2.2 系统调用机制的实现

在发现了系统调用的本质之后，我们就要着手在我们的小操作系统中实现一套系统调用机制了。不过，不要着急，为了使得后面的思路更清晰，我们先来整理一下系统调用的流程：

1. 调用库函数

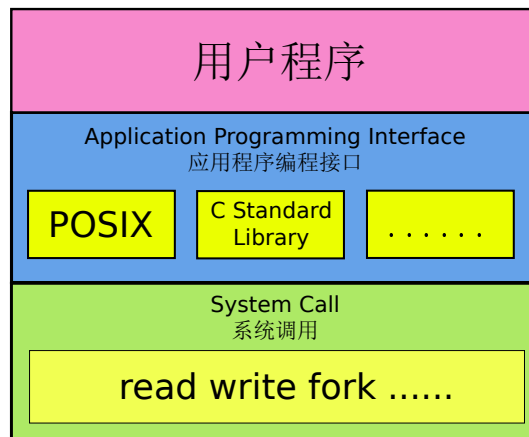


Figure 4.1: API、系统调用层次结构

2. 调用一个用户空间的系统调用函数（用于封装设置寄存器、传参等繁复的操作）。
3. 内核空间系统调用函数被执行。
4. 从相应寄存器中收集返回值，并返回到库函数中。
5. 库函数返回，回到用户程序

也就是说，为了实现系统调用机制并实现我们自己的系统调用，我们需要找到内核空间中的系统调用函数，以及对应的用户空间系统调用函数。不难发现，内核空间的系统调用实现在 `lib/syscall_all.c` 中。在这段代码中，有大量的以 `sys` 开头的函数，这些函数便是内核空间的系统调用的实现。

之后，我们再看用户空间的 `user/syscall_lib.c`，发现了一堆以 `syscall` 开头的函数，每个函数都根据传入的参数，调用了 `msyscall` 函数。看到这里，你想到了什么？`msyscall` 中一定是实际执行设置寄存器、执行 `syscall` 指令一类的底层操作的函数。我们看到的这些 `syscall` 开头的函数为用户封装了这些繁复的过程。

这些 `syscall` 函数已经被预先实现好了，我们来看看 `msyscall` 函数。这个函数位于 `user/syscall_wrap.S`。`msyscall` 需要实现将必要的参数压栈，并执行 `syscall` 指令陷入内核态。

**Exercise 4.1** 填写 `msyscall`，使得系统调用机制可以正常工作。 ■

看到这个 exercise 你是不是觉得有些无语？完全没有可以参考的信息啊！我们从何知道这个 `msyscall` 如何填写呢？仔细思考一下，`msyscall` 的主要工作是设置栈、寄存器等，为后面的系统调用设置好参数。也就是说，想知道 `msyscall` 需要做些什么，我们只需看一下后面内核中的相关代码是如何使用这些参数的就 OK 了。知道了内核用了什么，自然也就知道我们应当设置些什么。内核处理系统调用的中断处理程序位于 `lib/syscall.S` 中，相应的注释已经预先加在了代码中，如代码15所示。

Listing 15: 系统调用服务程序

```

1  NESTED(handle_sys,TF_SIZE, sp)
2
3  SAVE_ALL
4  CLI
5
6  nop
7  .set at
8  lw t1, TF_EPC(sp)
9  lw v0, TF_REG2(sp)
10 # 下面这段是在通过系统调用号来查找对应的系统调用的入口地址
11 # v0 里是系统调用号。
12 subu v0, v0, __SYSCALL_BASE
13 sltiu t0, v0, __NR_SYSCALLS+1
14
15 addiu t1, 4
16 sw t1, TF_EPC(sp)
17 beqz t0, illegal_syscall//undef
18 nop
19 sll t0, v0, 2
20 la t1, sys_call_table
21 addu t1, t0
22 lw t2, (t1)
23 beqz t2, illegal_syscall//undef
24 nop
25 # 这里提取了之前用户空间的栈指针的位置。
26 lw t0, TF_REG29(sp)
27
28 # 这里是核心，这段代码将之前存在栈中的参数载入到了 t1-t7 这七个寄存器中。
29 lw t1, (t0)
30 lw t3, 4(t0)
31 lw t4, 8(t0)
32 lw t5, 12(t0)
33 lw t6, 16(t0)
34 lw t7, 20(t0)
35
36 # 为内核中系统调用函数的参数分配栈空间
37 subu sp, 20
38
39 # 把参数存在当前的栈上。
40 sw t1, 0(sp)
41 sw t3, 4(sp)
42 sw t4, 8(sp)
43 sw t5, 12(sp)
44 sw t6, 16(sp)
45 sw t7, 20(sp)
46
47 # 把前四个参数移入 a0~a3。
48 # 前四个参数在 a0~a3, 后面的参数在栈上，这是 MIPS 的 ABI 标准的要求。
49 move a0, t1
50 move a1, t3
51 move a2, t4
52 move a3, t5
53
54 # 跳转到系统调用函数入口地址

```



```

55  jalr    t2
56  nop
57
58  # 释放栈空间
59  addu    sp, 20
60
61  # 将返回值保存在进程的运行环境的 v0 寄存器中。
62  # 这样返回用户态时，用户就可以获得返回值了。
63  sw      v0, TF_REG2(sp)
64
65  j        ret_from_exception # extern?
66  nop
67
68  illegal_syscall: j illegal_syscall
69                  nop

```

通过上面的代码，我们可以知道，所有的参数都应当按照顺序放入栈中。根据 MIPS 的 ABI 标准，前 4 个参数通过 a0-a3 寄存器传递，后面的参数通过栈传递。值得注意的是，标准要求程序必须分配足够容纳所有参数的栈空间。也就是说，尽管在函数调用的过程中，前四个参数是通过寄存器来传递的，但程序依旧需要为其分配栈空间。也就是说，GCC 在生成调用 `msyscall` 函数的代码时已经按照标准为我们开了一个足够大的栈，我们在 `msyscall` 中只需将前四个参数压入到栈中即可，无需再自行分配栈空间。

完成 `msyscall` 函数后，我们的小内核的系统调用机制便可以正常工作了。

### 4.2.3 基础系统调用函数

在系统调用机制搞定之后，我们自然是要弄几个系统调用玩一玩了。我们实现些什么系统调用呢？打开 `lib/syscall_all.c`，可以看到琳琅满目的系统调用函数等着我们去填写。这些系统调用都是基础的系统调用，不论是之后的 IPC 还是 `fork`，都需要这些基础的系统调用作为支撑。

首先我们看向 `sys_mem_alloc`。

[未填完的坑][FIX ME]

## 4.3 进程间通信机制 (Inter-Process Communication)

在系统调用机制搞定之后，我们自然是要弄几个系统调用玩一玩了。作为一个微内核系统，我们要来实现个什么系统调用呢？没错，当然是 IPC 了。IPC 可是微内核最重要的机制之一了。

**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讯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。

接下来进入正题，IPC 机制远远没有我们想象得那样神秘，特别是在我们这个被极度简化了的小操作系统中。根据之前的讨论，我们能够得知这样几个细节：

- IPC 的目的是使两个进程之间可以通讯。
- IPC 需要通过系统调用来实现

所谓通讯，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讯的能力。那么，要实现交换数据，我们所面临的最大的问题是什么呢？没错，问题就在于**各个进程的地址空间是相互独立的**。相信你在实现内存管理的时候已经深刻体会到了这一点，每个进程都有各自的地址空间，这些地址空间之间是相互独立的。因此，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。

想要让两个完全独立的地址空间之间发生联系，最好的方式是什么呢？对，我们要去找一找它们是否存在共享的部分。虽然地址空间本身独立，但是有些地址也许被映射到了同一物理内存上。如果你之前详细地看过进程的页表建立的部分的话，想必你已经找到线索了。是的，线索就在 `env_setup_vm()` 这个函数里面。

```

1  static int
2  env_setup_vm(struct Env *e)
3  {
4      //略去的无关代码
5
6      for (i = PDX(UTOP); i <= PDX(~0); i++) {
7          pgdir[i] = boot_pgdir[i];
8      }
9      e->env_pgdir = pgdir;
10     e->env_cr3 = PADDR(pgdir);
11
12     //略去的无关代码
13 }
```

如果你之前认真思考了这个函数的话会发现，所有的进程都共享了内核所在的 2G 空间。对于任意的进程，这 2G 都是一样的。因此，想要在不同空间之间交换数据，我们就需要借助于内核的空间来实现。那么，我们把要传递的消息放在哪里比较好呢？恩，发送和接受消息和进程有关，消息都是由一个进程发送给另一个进程的。内核里什么地方和进程最相关呢？啊哈！进程控制块！

```

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  };
```

果然，我们看到了我们想要的东西，`env_ipc_value` 用于存放需要发给当前进程的数据。`env_ipc_dstva` 则说明了接收到的页需要被映射到哪个虚地址上。知道了这些，我们就不难实现 IPC 机制了。只需要做做赋值，填充下对应的域，映射下该映射的页之类的就好了。

**Exercise 4.2** 实现 lib/syscall\_all.c 中的 void sys\_ipc\_recv(int sysno, u\_int dstva) 函数和 int sys\_ipc\_can\_send(int sysno, u\_int envid, u\_int value, u\_int srcva, u\_int perm) 函数。 ■

sys\_ipc\_recv(int sysno, u\_int dstva) 函数首先要将 env\_ipc\_recving 设置为 1, 表明该进程准备接受其它进程的消息了。之后阻塞当前进程, 即将当前进程的状态置为不可运行。之后放弃 CPU (调用相关函数重新进行调度)。

int sys\_ipc\_can\_send(int sysno, u\_int envid, u\_int value, u\_int srcva, u\_int perm) 函数用于发送消息。如果指定进程为可接收状态, 则发送成功, 清除接收进程的接收状态, 使其可运行, 返回 0, 否则, 返回 \_E\_IPC\_NOT\_RECV。

讲完 IPC 后, 我们来利用前面已经实现的基础系统调用来实现一个非常有趣的函数: fork。

## 4.4 FORK

在 Lab3 我们曾提到过, env\_alloc 是内核产生一个进程。但如果想让一个进程创建一个进程, 就像是父亲与儿子那样, 我们就需要使用到 fork 了。那么 fork 究竟是什么呢?

### 4.4.1 初窥 fork

fork, 直观意象是叉子的意思。在我们这里更像是分叉的意思, 就好像一条河流动着, 遇到一个分叉口, 分成两条河一样, fork 就是那个分叉口。在操作系统中, 在某个进程中调用 fork() 之后, 将会以此为分叉分成两个进程运行。新的进程在开始运行时有着和旧进程**绝大部分相同的信息**, 而且在新的进程中 fork 依旧有一个返回值, 只是该返回值为 0。在旧进程, 也就是所谓的父进程中, fork 的返回值是子进程的 env\_id, 是大于 0 的。在父子进程中有不同的返回值的特性, 可以让我们在使用 fork 后很好地区分父子进程, 从而安排不同的工作。

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? 换成创建一个空白的进程多简单! 按笔者的理解, 这是因为:

- 与不相干的两个进程相比, 父子进程间的通信要方便的多。因为 fork 虽然没法造成进程间的统治关系<sup>3</sup>, 但是因为在子进程中记录了父进程的一些信息, 父进程也可以很方便地对子进程进行一些管理等。
- 当然还有一个可能的原因在于安全与稳定, 尤其是关于操作权限方面。对这方面有兴趣的同学可以查看链接<sup>4</sup> 探索一下。

fork 之后父子进程就分道扬镳, 互相独立了。而和 fork “针锋相对” 却又经常 “纠缠不清” 的, 是名为 exec 系列的系统调用。它会 “勾引” 子进程抛弃现有的一切, 另起

<sup>3</sup>这是因为进程之间是并发的, 在操作系统看来, 父子进程之间更像是兄弟关系。

<sup>4</sup><http://www.jbxue.com/shouce/apache2.2/mod/prefork.html>

炉灶。若在子进程中执行 `exec`，完成后子进程从父进程那拷贝来的东西就全部消失了。取而代之的是一个全新的进程，就像太乙真人用莲藕为哪吒重塑了一个肉身一样。`exec` 系列系统调用我们将会作为一个挑战性任务放在后面来实现，暂时不做过多介绍。

为了让你对 `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，复制到你的 `linux` 环境下运行一下吧。

Listing 16: 理解 `fork`

```
1  #include<stdio.h>
2  #include<sys/types.h>
3
4  int main(void){
5      int var;
6      pid_t pid;
7      printf("Before fork.\n");
8      pid = fork();
9      printf("After fork.\n");
10     if(pid==0){
11         printf("son.");
12     }else{
13         sleep(2);
14         printf("father.");
15     }
16     printf("pid:%d\n",getpid());
17     return 0;
18 }
```

使用 `gcc fork_test.c`，然后 `./a.out` 运行一下，你得到的正常的结果应该如下所示：

```
1  Before fork.
2  After fork.
3  After fork.
4  son.pid:16903 (数字不一定一样)
5  father.pid:16902
```

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：

- 在 `fork` 之前的代码段只有父进程会执行。
- 在 `fork` 之后的代码段父子进程都会执行。
- `fork` 在不同的进程中返回值不一样，在父进程中返回值不为 0，在子进程中返回值为 0。
- 父进程和子进程虽然很多信息相同，但他们的 `env_id` 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绝大多数信息和状态作为模板而雕琢出来的。即使是以父进程为模板，父子进程也还是有很多不同的地方，不知细心的你从刚才的小实验中能否看出父子进程有哪些地方是明显不一样的吗？

**Thinking 4.1** 思考下面的问题，并对这两个问题谈谈你的理解：

- 子进程完全按照 `fork()` 之后父进程的代码执行，说明了什么？
- 但是子进程却没有执行 `fork()` 之前父进程的代码，又说明了什么？

#### 4.4.2 fork 的结构

通过使用初步了解 `fork` 后，先不着急实现它。俗话说“兵马未动，粮草先行”，我们先来了解一下关于 `fork` 的底层细节。根据维基百科的描述，在 `fork` 时，父进程会为子进程分配独立的地址空间。但是分配独立的虚拟空间并不意味着一定会分配额外的物理内存：父子进程用的是相同的物理空间。子进程的代码段、数据段、堆栈都是指向父进程的物理空间，也就是说，虽然两者的虚拟空间是不同的，但是他们所对应的物理空间是同一个。

**Note 4.4.1** Wiki Fork: In Unix systems equipped with virtual memory support (practically all modern variants), the `fork` operation creates a separate address space for the child. The child process has an exact copy of all the memory segments of the parent process, though if copy-on-write semantics are implemented, the physical memory need not be actually copied. Instead, virtual memory pages in both processes may refer to the same pages of physical memory until one of them writes to such a page: then it is copied. This optimization is important in the common case where `fork` is used in conjunction with `exec` to execute a new program: typically, the child process performs only a small set of actions before it ceases execution of its program in favour of the program to be started, and it requires very few, if any, of its parent's data structures.

那你可能就有问题了：既然上文提到了父子进程之间是独立的，而现在又说共享物理内存，这不是矛盾吗？按照共享物理内存的说法，那岂不是变成了“父教子从，子不得不从”？

这两种说法实际上不矛盾，因为父子进程共享物理内存是有前提条件的：共享的物理内存不会被任一进程修改。那么，对于那些父进程或子进程修改的内存我们又该如何处理呢？这里我们引入一个新的概念——写时复制（copy on write，简称 COW）。写时复制，通俗来讲，就是当父子进程中有**更改**相应段的行为发生时，再为子进程相应的段分配物理空间，而**子进程的代码段继续共享父进程的物理空间**（两者的代码完全相同）。

**Note 4.4.2** 如果在 `fork` 之后在子进程中执行了 `exec`，由于这时和父进程要执行的代码完全不同，子进程的代码段也会分配单独的物理空间。

更深层次地讲，COW 就是父进程和子进程平时共享物理页面，写时复制物理页面。为了能够在物理页被修改时及时处理，只要是进程可写的页面，就要**通过设置权限位 `PTE_COW` 的方式**被保护起来。无论父进程还是子进程何时试图写一个被保护的物理页，就会产生一个异常。异常发生后，操作系统检测该物理页是否为 copy on write

的页面，如果是的话将原页复制一份，然后重新映射到导致中断的虚拟地址上，之后重新执行指令。下面这张图较为生动地展示了这一过程：

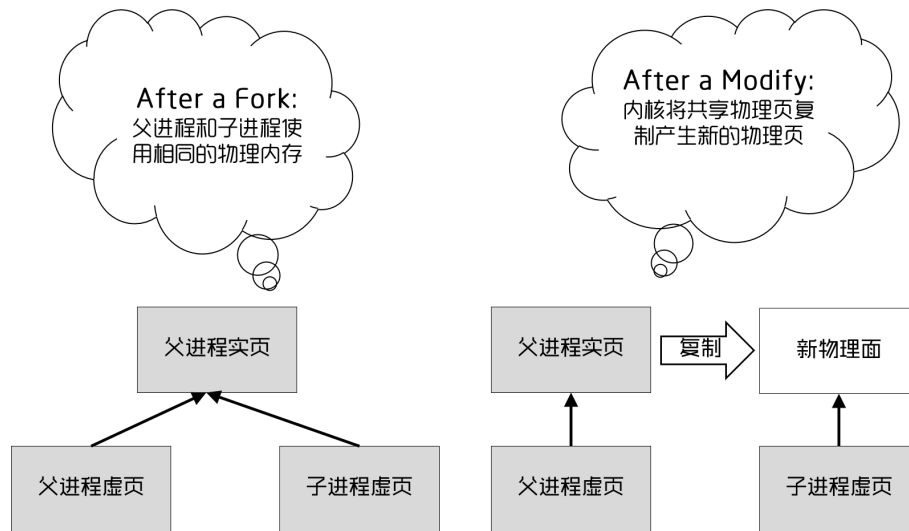


Figure 4.2: fork 的 Copy-On-Write 机制

**Note 4.4.3** 早期的 Unix 系统对于 fork 采取的策略是：直接把父进程所有的资源复制给新创建的进程。这种实现过于简单，并且效率非常低。因为它拷贝的内存也许是需要父子进程共享的，当然更糟的情况是，如果新进程打算通过 exec 执行一个新的映像，那么所有的拷贝都将前功尽弃。

在我们的实验中，fork 呢，主要由以下几个子函数和一些我们之前填过的系统调用构成：

**pgfault** 还记得之前的 cow 吗？没错，这个函数就是为了解决这个问题而存在的。在我们的实验中，我们通过 **pgfault** 函数来对共享页保护引起的异常进行处理。

**syscall\_env\_alloc** 这个系统调用是 fork 两个返回值的关键所在，fork 是通过判断这个系统调用的返回值是否为 0 从而判断当前执行 fork 的是父进程还是子进程。

**duppage** 这个函数就是用来给共享的物理页添加保护权限位的。在 fork 中我们要通过遍历用户空间来寻找“需要添加保护”的页——即那些可能会被进程修改的物理页。

小红：“咦，不科学啊。fork 的两个返回值为啥是 `sys_env_alloc` 的功劳？不是说子进程只执行 fork 之后的代码吗？”

小绿：“你还别不信，还真的就是 `sys_env_alloc` 的功劳。我们前面是提到了子进程执行 fork 之后的代码，实则不准确：因为在 fork 内部呀，就要用 `sys_env_alloc` 的两个返回值区分开父子进程，好安排他们在返回之后执行不同的任务呀！你想想，虽然子进程在被创建出来就已经有：

**健康的身体** `env_alloc` 给予了 PCB 控制块



**良好的家境** `sys_env_alloc` 为它拷贝了合适的 CPU 环境

**自己的房子** `env_alloc` 还给予了它栈区

但是它还缺少一个避风港 (UXSTACK——用户异常栈), 更重要的是, 子进程只有在得到父进程的允许才能独立生活 (进程是否能够开始被调度是要由父进程拍板决定的)。所以, 在 `fork` 内部得让父进程为子进程把剩下的生活安顿好, 才能让它独立啊!”

我们在这里又看到了一个新的概念, 异常栈。那么它和正常栈有什么区别呢? 你应该知道, 一般的用户进程运行时, 会有自己的用户栈 (以 `USTACKTOP` 为栈顶)。但是当发生页中断后, 我们是不能直接在用户栈上处理页中断的。为了区别于用户的正常栈, 内核将重新启动该进程并使其在另外一个栈 (以 `UXSTACKTOOP` 为栈顶)——用户异常栈上运行用户已经向内核注册过的的中断处理程序, 该中断处理程序通过系统调用来处理页中断。

在讲完上述这些后, 不知道你对 `fork` 的具体流程是否弄清楚了呢? 下面就让我们来具体地一个函数一个函数地攻破它吧!

#### 4.4.3 返回值的秘密

刚刚接触 `fork` 这个函数, 相信你最感兴趣的可能不是别的, 而是 `fork` 的两个返回值。而我们刚才也发现, `fork` 的两个返回值实际上是由 `sys_env_alloc` 所引起的, 究其根本, 秘密在 `sys_env_alloc` 身上。那么, 我们首先来思考一个问题:

**Thinking 4.2** 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 ■

以这个问题作为我们的开题小菜, 结合刚才未让你填写的系统调用 `sys_env_alloc` 函数来说明 `fork` 的两个返回值。

Listing 17: `alloc`——`fork` 之魂

```

1  /* Overview:
2     *       Allocate a new environment.
3     *
4     * Pre-Condition:
5     * The new child is left as env_alloc created it, except that
6     * status is set to ENV_NOT_RUNNABLE and the register set is copied
7     * from the current environment.
8     *
9     * Post-Condition:
10    *       In the child, the register set is tweaked so sys_env_alloc returns 0.
11    *       Returns env_id of new environment, or < 0 on error.
12    */
13 int sys_env_alloc(void)

```



```

14 {
15     // Your code here.
16     int r;
17     struct Env *e;
18 }

```

这个系统调用就是为了新建一个进程。从系统调用的名字上来看我们也能发现，实际上这个系统调用和 `env_alloc` 的功能很像。但是我们又提到 `fork` 是一种不完全复制，所以它除了建造外，还需要用一些当前进程的信息作为模版来填充新的进程。那么还需要复制些什么？

**CPU 环境** 要复制一份当前进程的运行环境到子进程的 `env_tf` 控制块里。

**PC** 刚才那些知识综合一下，你应该也能推断出子进程实际上是从 `sys_env_alloc` 返回的地方作为起点，执行父进程的代码。所以子进程的 PC 值应该被设置为 `syscall_env_alloc` 返回后的地址。

**返回值有关** 看到 `syscall_env_alloc` 提示你应该能明白，我们需要调整某个寄存器的值以让 `syscall_env_alloc` 在子进程中可以返回 0。

**进程状态** 我们当然不能让子进程在 `syscall_env_alloc` 返回后就直接进入调度，因为这时候它还没有做好充分的准备，所以我们需要设定不能让它被加入调度队列。

了解到这些信息，相信你可以写出一个合格的 `syscall_env_alloc` 系统调用了，那么下面我们就把它填充完整。

**Exercise 4.3** 填写 `./lib/syscall_all.c` 中的函数 `sys_env_alloc`，可以不填返回值。 ■

我们刚才提到了子进程好像是从 `sys_env_alloc` 返回之后的地方开始执行的，那么，证据呢？

证据其实离我们一点都不远——在刚才的 `fork` 内部我们讲到需要区分父子进程，所以需要判断 `sys_env_alloc` 的返回值，所以会执行这样结构的语句：

```

1     envuid = sys_env_alloc();
2     //在父进程中
3     if(envuid!=0)
4         ...
5     //在子进程中
6     else
7         ...

```

那么现在的问题就是在不同的进程中，为什么 `envuid` 会有两个值，一个为 0，一个非 0？

实际上是这样的：在父进程运行到 `envuid = sys_env_alloc` 这句话时，我们把它拆分成两步，其实是如下的一个过程：

```

1     # 假设返回值存放的寄存器为 R
2     sys_env_alloc() -> R
3     R -> envuid

```

首先运行 `sys_env_alloc()` 函数，它的返回值被放在了 R 寄存器中。下一步将 R 寄存器中的值赋给 `envid`。我们再返回来看一下前面所述“子进程的 PC 值应该被设置为 `sys_env_alloc` 返回后的地址”。所以当子进程开始被调度时，子进程运行的第一条指令实际上是上述代码中的 `R -> env`，又因为我们之前在父进程中已经设置子进程的进程控制块的 R 寄存器值为 0，所以在子进程中，`env` 的值是 0！而在父进程中，因为父进程完整地执行完了 `sys_env_alloc`，所以其 R 寄存器存放的正是 `sys_env_alloc` 的返回值。

更直观一些，请看图4.3:

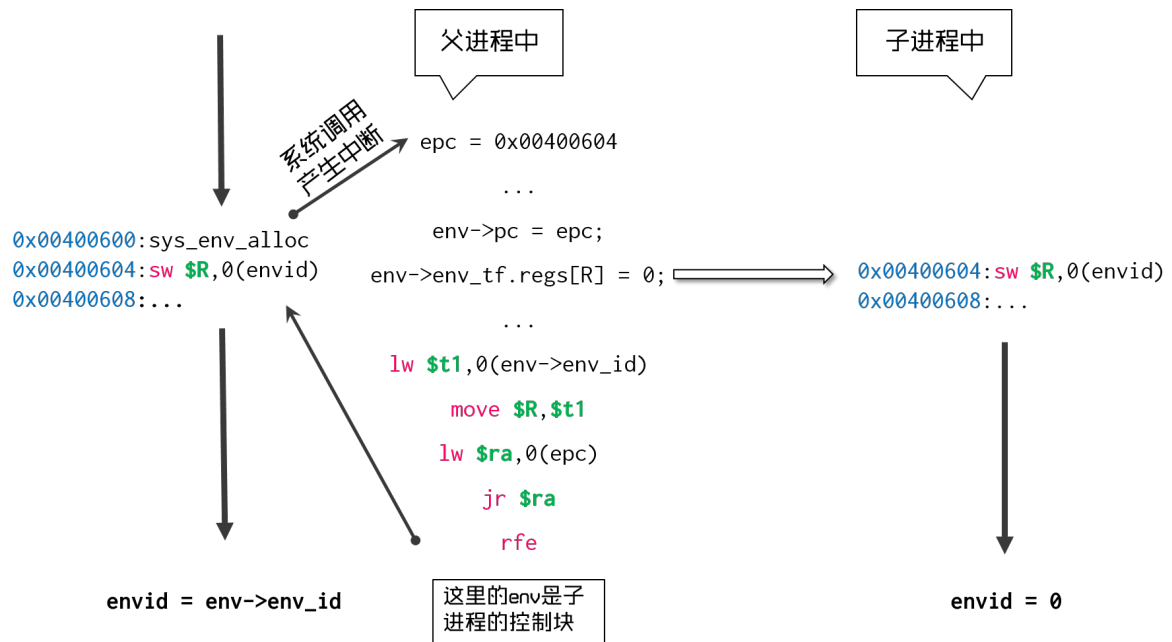


Figure 4.3: 两个返回值

**Exercise 4.4** 补充./lib/syscall\_all.c 中的函数 `sys_env_alloc` 的返回值。 ■

返回值的秘密我们讲完了，那么接下来该做些什么呢？当然是填写 `fork` 啦，根据上文我们知道，`fork` 内部要通过 `syscall_env_alloc()` 的返回值来区分父子进程，那么在父进程中和在子进程中分别要完成什么任务呢？

#### 4.4.4 父与子

`fork` 里的 `extern struct Env *env;`，这个 `env` 可是大有来历：它是源自外部函数 `./user/libos.c` 里的一个全局变量。找到 `entry.S` 仔细观察一下就可以发现：在整个实验体系中，`_start` 叶函数执行完毕后会跳转到 `libmain` 执行，真正的入口函数是从 `libmain` 开始的。`libmain` 中使用 `env` 指向当前的进程以便之后进程通信的需要，然后才开始执行我们所使用的测试函数中的 `umain` 内容。所以，为了使得 `env` 始终指向当前进程，在子进程中我们需要进行必要的设置。

**Exercise 4.5** 补充./user/fork.c 中的函数 fork 中关于 sys\_env\_alloc 的部分和“子进程”执行的部分。 ■

那父进程需要做些什么呢？我们在刚才提到过了父进程的任务：父进程要帮子进程搭建一个 UXSTACK，然后为了能让这个错误栈正确地投入使用，还得帮子进程注册错误处理函数。最后父进程恋恋不舍地把子进程的状态更改为 RUNNABLE，子进程就可以参与调度了——“你走吧，终究还是要出去闯荡的”。这时候父进程任务做完，就可以松一口气了。

难道真的能松口气了？不，父进程这一切之前漏了最重要的一步：没把子进程跟父进程共享的物理页保护起来。所以，我们还需要遍历父进程的所有用户空间页，然后找到可写的页，让它在父进程和子进程中同时被 PTE\_COW 权限保护起来！我们前面提到了，保护页用的函数是啥？对，就是 duppage 函数。

### duppage

在 duppage 函数中，唯一需要强调的一点是要对不同权限的页有着不同的处理方式。

- 在父进程中可写的或者是被打上 PTE\_COW 标记的页，需要在子进程中为其加上 PTE\_COW 的标志。
- 在父进程中只读的页，按照相同的权限给予进程就好。

**Exercise 4.6** 结合注释，补充 user/fork.c 中的函数 duppage。 ■

### pgfault

我们前面提到了写时复制。如果说 duppage 是负责触发“写时”的，那 pgfault 就是负责“保护”的。pgfault 很简单，按下述步骤执行即可：

1. 判断页是否为 copy-on-write，是则进行下一步；
2. 分配一个新的内存页到临时位置，将要复制的内容拷贝到刚刚分配的页中；
3. 将临时位置上的内容映射到指定地址 va，然后解除临时位置对内存的映射；

**Exercise 4.7** 结合注释，补充 user/fork.c 中的函数 pgfault。 ■

### fork

## 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. 掌握并实现文件系统的基本操作。
4. 了解微内核的基本设计思想和结构。

## 5.2 背景知识

### 5.2.1 文件系统

计算机的文件系统是一种存储和组织计算机数据的方法，它使得对其访问和查找变得容易，文件系统使用文件和树形目录的抽象逻辑概念代替了硬盘和光盘等物理设备使用数据块的概念，用户使用文件系统来保存数据不必关心数据实际保存在硬盘（或者光盘）的地址为多少的数据块上，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那个块地址没有被使用，硬盘上的存储空间管理（分配和释放）功能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。

文件系统通常使用硬盘和光盘这样的存储设备，并维护文件在设备中的物理位置。但是，实际上文件系统也可能仅仅是一种访问数据的界面而已，实际的数据是通过网络协议（如 NFS、SMB、9P 等）提供的或者内存上，甚至可能根本没有对应的文件（如 proc 文件系统）。

**Thinking 5.1** 查阅资料，了解 Linux/Unix 的 /proc 文件系统是什么？有什么作用？Windows 操作系统又是如何实现这些功能的？proc 文件系统这样的设计有什么好处？

### 5.2.2 磁盘

接下来简单介绍一下与磁盘相关的基本知识。磁盘相关的几个基本概念：

1. 扇区 (Sector): 磁盘盘片被划分成很多扇形的区域, 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。
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。

典型的磁盘的基本结构如图5.1所示：

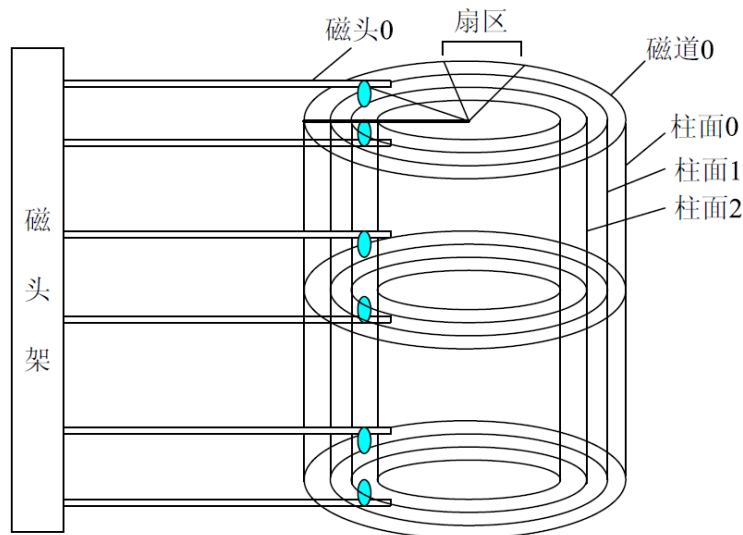


Figure 5.1: 磁盘结构示意图

## 5.3 IDE 磁盘

**Note 5.3.1** IDE 的英文全称为 “Integrated Drive Electronics”，即 “电子集成驱动器”，是目前最主流的硬盘接口，也是光储类设备的主要接口。IDE 接口，也称之为 ATA 接口。

在我们的操作系统实验中，我们使用的 Gxemul 模拟器提供了一个 IDE 仿真设备，我们需要此基础上实现我们的文件系统，接下来，我们将了解一些读写 IDE 磁盘的基础知识。

### 5.3.1 内存映射 I/O

在第二次试验中，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**，我们使用 I/O 端口来访问 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。这些硬件 I/O 寄存器被映射到指定的内存空间，例如，在 Gxemul 中，console 设备被映射到 **0x10000000**，simulated IDE disk 被映射到 **0x13000000**，等等。更详细的关于 Gxemul 的仿真设备的说明，可以参考 [Gxemul Experimental Devices](#)。

驱动程序访问的是 IO 空间，与一般我们说的内存空间是不同的。外设的 IO 空间地址是系统启动后才知道，CPU 通常并没有为这些已知的外设 I/O 内存资源的物理地址预定义虚拟地址范围，所以驱动程序并不能直接通过物理地址访问 I/O 内存资源，而必须将它们映射到内核虚拟地址空间内然后才能根据映射所得到的核心虚拟地址范围，通过访存指令访问这些 I/O 内存资源。

咱们的操作系统内核中，将物理内存转换为内核虚拟地址，可以使用 `KADDR` 宏，也就是将物理地址加上 `ULIM` 的值 (**0x80000000**)。

例如，Gxemul 提供的 console 设备的地址：**0x10000000**，设备寄存器映射如表 5.1 所示：

Table 5.1: Gxemul Console 内存映射

Offset	Effect
0x00	Read: <code>getchar()</code> (non-blocking; returns 0 if no char was available)
	Write: <code>putchar(ch)</code>
0x10	Read or write: <code>halt()</code>
	(Useful for exiting the emulator.)

现在，我们通过往内存的 (**0x10000000+0x80000000**) 地址写入字符，就能在 shell 中看到对应的输出。`drivers/gxconsole/console.c` 中的 `printcharc` 函数的实现如下所示：

```

1 void printcharc(char ch)
2 {
3     *((volatile unsigned char *) PUTCHAR_ADDRESS) = ch;
4 }
```

### 5.3.2 IDE 磁盘操作

磁盘的访问跟 Console 类似，也是将要用到的磁盘设备的 I/O 寄存器映射到制定的虚拟地址。前文中我们提到过，扇区 (Sector) 是磁盘读写的基本单位，Gxemul 也提供了对扇区进行操作的基本方法。

Gxemul 提供的 Simulated IDE disk 的地址是 **0x13000000**，I/O 寄存器相对于 **0x13000000** 的偏移和对应的功能如表 5.2 所示：

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 disk 的相关代码也就不难理解了。以从硬盘上读取一些 Sectors 为例，涉

Table 5.2: Gexmul IDE disk I/O 寄存器映射

Offset	Effect
0x0000	Write: Set the offset (in bytes) from the beginning of the disk image. This offset will be used for the next read/write operation.
0x0008	Write: Set the high 32 bits of the offset (in bytes). (*)
0x0010	Write: Select the IDE ID to be used in the next read/write operation.
0x0020	Write: Start a read or write operation. (Writing 0 means a Read operation, a 1 means a Write operation.)
0x0030	Read: Get status of the last operation. (Status 0 means failure, non-zero means success.)
0x4000-0x41ff	Read/Write: 512 bytes data buffer.

及到的函数有 `fs/ide.c` 中的 `ide_read` 函数和 `fs/ide_asm.S` 中的 `read_sector` 函数。具体代码如下：

`ide_read` 函数：

```

1  extern int read_sector(int diskno, int offset);
2
3  void ide_read(u_int diskno, u_int secno, void *dst, u_int nsecs)
4  {
5      // 0x200: the size of a sector: 512 bytes.
6      int offset_begin = secno * 0x200;
7      int offset_end = offset_begin + nsecs * 0x200;
8      int offset = 0;
9
10     while (offset_begin + offset < offset_end) {
11         if (read_sector(diskno, offset_begin + offset)) {
12             // copy data from disk buffer(512 bytes, a sector) to destination array.
13             user_bcopy((void *)0x93004000, dst + offset, 0x200);
14             offset += 0x200;
15         } else {
16             // error occurred, then panic.
17             user_panic("disk I/O error");
18         }
19     }
20 }
```

`read_sector` 函数：

我们来分析这两段代码。当需要从磁盘的指定位置读取一个 sector 时，需要调用 `read_sector` 函数来将磁盘中对应 sector 的数据读到设备缓冲区中。所有的地址操作都需要将物理地址转换成内核虚拟地址。根据 Gxemul 提供的与 ide disk 相关的数据表格，首先，设置 IDE disk 的 ID，从 `read_sector` 函数的声明 `extern int read_sector(int diskno, int offset);` 中可以看出，`diskno` 是第一个参数，对应的就是 `$a0` 寄存器的值，因此，将其写入到 `0x93000010` 处，这样就表示我们将使用编号为 `$a0` 的磁盘。我们的试验中，只使用了一块 simulated IDE disk，因此，这个值应该为 0。接下来，将相对于磁盘起始位置的 `offset` 写入到 `0x93000000` 位置，



表示在距离磁盘起始处 `offset` 的位置开始进行磁盘操作。然后，根据 Gxemul 的 data sheet，向内存 `0x93000020` 处写入 0 来开始读磁盘（如果是写磁盘，则写入 1）。最后，将磁盘操作的状态码放入 `$v0` 中，作为结果返回给 `ide_read` 函数。在 `ide_read` 函数中，我们通过判断 `read_sector` 函数的返回值，就可以知道读取磁盘的操作是否成功。如果成功，将这个 sector 的数据 (512 bytes) 从设备缓冲区 (`0x4000-0x41ff`) 中拷贝到目的位置 (此处的 `void *dst`)。至此，我们就完成了对磁盘的读操作。

写磁盘的操作与读磁盘的一个区别在于写磁盘需要先将要写入对应 sector 的 512 bytes 的数据放入设备缓冲中，然后向地址 `0x93000020` 处写入 1 来启动操作，并从 `0x93000030` 处获取写磁盘操作的返回值。

**Exercise 5.1** 参考 `ide_read` 函数和 `read_sector` 函数的实现方式，实现 `fs/ide.c` 中的 `ide_write` 函数，以及 `fs/ide_asm.S` 中的 `write_sector` 函数，实现对磁盘的写操作。 ■

## 5.4 文件系统结构

**Note 5.4.1** Unix/Linux 操作系统一般将磁盘分成两个区域：inode 区域和 data 区域。inode 区域用来保存文件的状态属性，以及指向数据块的指针。data 区域用来存放文件的内容和目录的元信息 (包含的文件)。我们实验使用的操作系统内核的文件系统也采用类似的设计。

### 5.4.1 磁盘布局

磁盘空间的基本布局如图5.2所示。

从图中可以看到磁盘最开始的一个扇区 (512 字节) 被当成是启动扇区和分区表使用。接下来的一个扇区用作超级块 (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。

**Note 5.4.2** 在真真实的文件系统中，一般会维护多个超级块，通过复制分散到不同的磁盘分区中，以防止超级块的损坏造成整个磁盘无法使用。

我们的操作系统中超级块的结构:

```
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 };
```

超级块的内容包括一个 magic number，磁盘中 block 的个数，以及根目录。

操作系统有两种常用的方式来管理资源：空闲链表和位图。在第二次试验和第三次试验中，我们使用了空闲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。具体来说，我们使用一个二进制位来表示磁盘上的一个块 (Block) 是否使用。

接下来，我们来学习如何使用 bitmap 来标识磁盘中的所有块的使用情况。

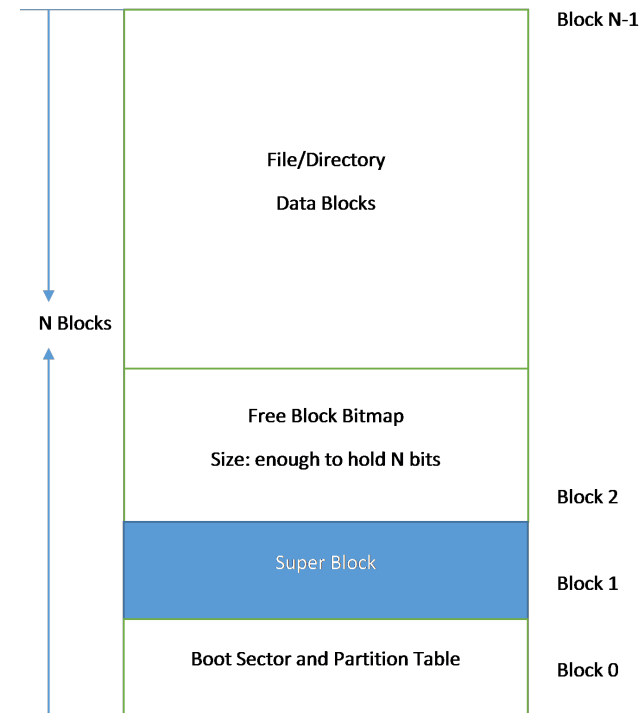


Figure 5.2: 磁盘空间布局示意图

fs/fsformat 用于将多个文件按照我们的内核所定义的文件系统写入到磁盘镜像中。在写入文件之前，我们将所有的块全部都标为空闲块：

```

1  nbitblock = (nblock + BIT2BLK - 1) / BIT2BLK;
2  for(i = 0; i < nbitblock; ++i) {
3      memset(disk[2+i].data, 0xff, nblock/8);
4  }
5  if(nblock != nbitblock * BY2BLK) {
6      diff = nblock % BY2BLK / 8;
7      memset(disk[2+(nbitblock-1)].data+diff, 0x00, BY2BLK - diff);
8  }
```

nbitblock 表示要记录整个磁盘上所有块的使用信息，我们需要多少个块来存储位图。紧接着，我们将所有位图块的每一位都标记为 0x1，表示这一块磁盘处于空闲状态。需要格外注意的是，如果 bitmap 还有剩余，我们不能将最后一块位图块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，不可被使用。因此，在 fs/fsformat.c 中的 init\_disk 函数中，在将所有的位图块都置为 0x1 之后，还需要根据实际情况，将多出来的位图标记为 0x0。

在我们的操作系统内核中，文件系统也需要根据 bitmap 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 bitmap 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 5.2** 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 bitmap 中的标志位。如果要将一个磁盘块设置为 free，只需要将 bitmap 中对应的 bit 的值设置为 0x0 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
4  free_block(u_int blockno)
5  {
6      // Step 1: Check if the parameter `blockno` is valid (`blockno` can't be zero).
7
8      // Step 2: Update the flag bit in bitmap.
9
10 }
```

### 5.4.2 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。我们实验使用的内核中，文件系统是一个用户进程，一个进程可以拥有 4G 的虚拟内存空间，将 `DISKMAP` 到 `DISKMAP+DISKMAX` 这一段虚存地址空间 (`0x10000000-0xcfffff`) 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/fs.h` 中：

```

1  #define DISKMAP    0x10000000
2  #define DISKMAX    0xc0000000

```

**Thinking 5.2** 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，我们采用这样一种设计：结构如图 5.3。

**Exercise 5.3** `fs/fs.c` 中的 `diskaddr` 函数用来计算指定磁盘块对应的虚存地址。完成 `diskaddr` 函数，根据一个块的序号 (block number)，计算这一磁盘块对应的 512 bytes 虚存的起始地址。(提示：`fs/fs.h` 中的宏 `DISKMAP` 和 `DISKMAX` 定义了磁盘映射虚存的地址空间)。

当我们把一个磁盘块 (block) 中的内容载入到内存中时，我们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。

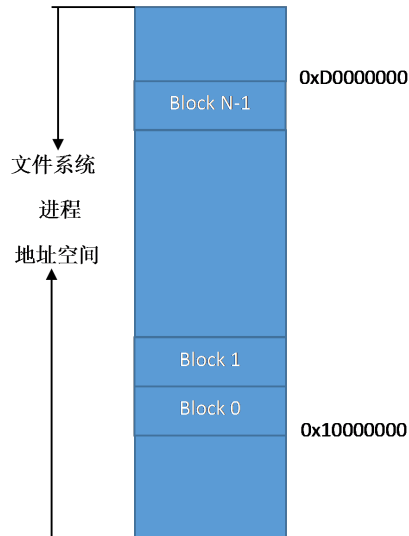


Figure 5.3: 块缓存示意图

**Exercise 5.4** 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。对应地，完成 `unmap_block` 函数，用于接触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都入到内存中，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。

### 5.4.3 文件系统详细结构

在操作系统的学习中，我们多次提到，操作系统要想管理一类资源，就得有相应的数据结构。我们使用文件控制块来描述和管理文件。文件在磁盘上的组织形式如图5.4所示：

对应的文件控制块的定义：

```

1 // file control blocks, defined in include/fs.h
2 struct File {
3     u_char f_name[MAXNAMELEN]; // filename
4     u_int f_size;                // file size in bytes
5     u_int f_type;                // file type
6     u_int f_direct[NDIRECT];
7     u_int f_indirect;
8     struct File *f_dir;
9     u_char f_pad[BY2FILE - MAXNAMELEN - 4 - 4 - NDIRECT * 4 - 4 - 4];
10 };

```

我们使用的操作系统内核中，文件名的最大长度为 `MAXNAMELEN`(128)，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

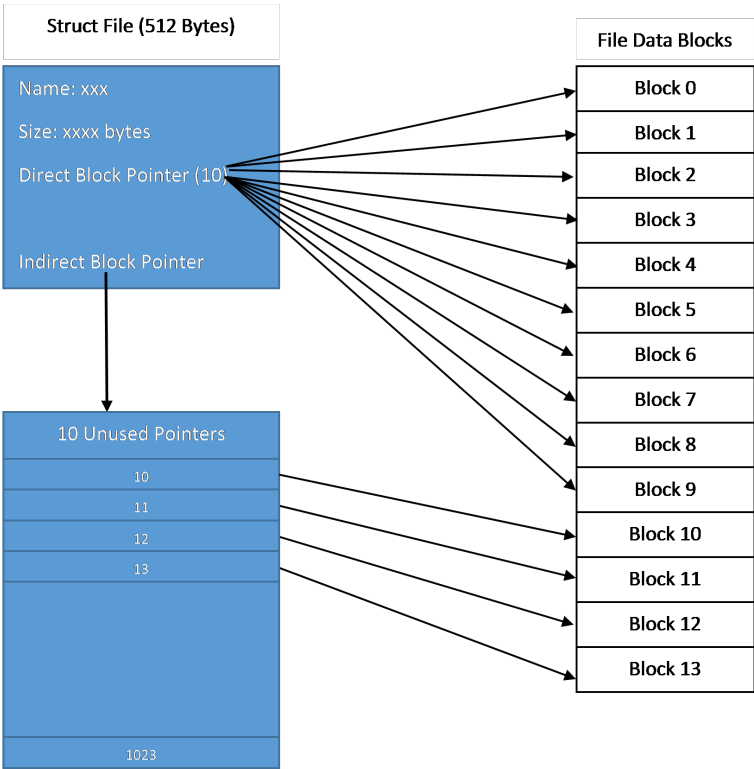


Figure 5.4: 文件控制块

小大于 40KB 时，就需要用到间接指针。File 结构体中有一个域为 `f_indirect`，只想一个间接磁盘块，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。文件控制块的结构如图5.4。

**Note 5.4.3** 我们的文件系统中文件控制块只是用了一级间接指针域，也只有一个。而在真是的文件系统中，对了支持更大的文件，通常会使用多个间接磁盘块，或使用多级间接磁盘块。我们使用的操作系统内核在这一点上做了极大的简化。

**Thinking 5.3** 一个 Block 最多存储 1024 个指向其他磁盘块的指针，试计算，我们的文件系统支持的单个文件的最大大小为多大？ ■

为了更加细致地了解文件系统的内部结构，我们使用 C 语言程序 (`fs/fsformat.c`) 来模拟对磁盘的操作，掌握如何将文件和文件夹按照文件系统的格式写入磁盘，我们也正是通过 `fsformat` 程序来创建一个磁盘文件 `fs/fs.img` 供内核使用。请阅读 `fs/fsformat.c` 中的代码，掌握文件系统结构的具体细节。感兴趣的同学，可以参考 `write_file` 函数的实现，完成 `write_directory` 函数，实现将一个指定目录下的文件按照目录结构写入到 `fs/fs.img` 的根目录下。

## 5.5 文件系统服务

我们实验使用的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，以服务的形式通过 IPC 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_recv` 与 `fs_serv` 进行交互，完成操作。在文件系统服务进程的初始化函数中，首先调用了 `serv_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，`serve` 函数被调用，文件系统服务开始运行，等待其他程序的请求。

**Thinking 5.4** 阅读 `serve` 函数的代码，我们注意到函数中包含了一个死循环 `for (;;) {...}`，为什么这段代码不会导致整个内核进入 `panic` 状态？

文件系统支持的请求类型定义在 `include/fs.h` 中，包含以下几种：

```
1  #define FSREQ_OPEN      1
2  #define FSREQ_MAP      2
3  #define FSREQ_SET_SIZE  3
4  #define FSREQ_CLOSE    4
5  #define FSREQ_DIRTY    5
6  #define FSREQ_REMOVE   6
7  #define FSREQ_SYNC     7
```

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对文件系统的操作（文件的增、删、改、查等），将结果重新通过 IPC 反馈给用户程序。

**Exercise 5.5** 文件 `user/fsipc.c` 中定义了请求文件系统时用到的 IPC 操作，`user/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 `user/fsipc.c` 中的 `fsipc_remove` 函数、`user/file.c` 中的 `remove` 函数，以及 `fs/serv.c` 中的 `serve_remove` 函数，实现删除指定路径的文件的功能。

## 5.6 正确结果展示

在 `init/init.c` 中启动一个 `idle` 进程和文件系统服务进程：

```
1  ENV_CREATE(user_idle);
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。

```
1  FS is running
2  FS can do I/O
3  superbblock is good
4  diskno: 0
5  diskno: 0
```

```
6 read_bitmap is good
7 diskno: 0
8 alloc_block is good
9 file_open is good
10 file_get_block is good
11 file_flush is good
12 file_truncate is good
13 diskno: 0
14 file rewrite is good
```

## 5.7 实验思考

- Unix /proc 文件系统
- 磁盘最大体积
- 文件系统支持的单个文件的最大体积
- 文件系统服务进程运行机制

## 5.8 实验挑战



# CHAPTER 6

## 管道与 SHELL

### 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 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

通俗来讲，管道就像家里的自来水管：一端用于注入水，一端用于放出水，且水只能在一个方向上流动，而不能双向流动，所以说管道是典型的单向通信。管道又叫做匿名管道，只能用在具有公共祖先的进程之间使用，通常使用在父子进程之间通信。

在 Unix 中，管道由 pipe 函数创建，函数原型如下：

```
1 #include<unistd.h>
2
3 int pipe(int fd[2]); 成功返回0，否则返回-1；
4
5 参数fd返回两个文件描述符，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，同样，我们先来做实验亲身体会一下<sup>1</sup>

<sup>1</sup>实验代码参考 <http://pubs.opengroup.org/onlinepubs/9699919799/functions/pipe.html>

Listing 18: 管道示例

```

1  #include <stdlib.h>
2  #include <unistd.h>
3
4  int fildes[2];
5  /* buf size is 100 */
6  char buf[100];
7  int status;
8
9  int main(){
10
11     status = pipe(fildes);
12
13     if (status == -1 ) {
14         /* an error occurred */
15         printf("error\n");
16     }
17
18
19     switch (fork()) {
20     case -1: /* Handle error */
21         break;
22
23
24     case 0: /* Child - reads from pipe */
25         close(fildes[1]); /* Write end is unused */
26         read(fildes[0], buf, 100); /* Get data from pipe */
27         printf("child-process read:%s",buf); /* Print the data */
28         close(fildes[0]); /* Finished with pipe */
29         exit(EXIT_SUCCESS);
30
31
32     default: /* Parent - writes to pipe */
33         close(fildes[0]); /* Read end is unused */
34         write(fildes[1], "Hello world\n", 12); /* Write data on pipe */
35         close(fildes[1]); /* Child will see EOF */
36         exit(EXIT_SUCCESS);
37     }
38 }

```

示例代码实现了从父进程向子进程发送消息“Hello,world”，并且在子进程中打印到屏幕上。它演示了管道在父子进程之间通信的基本用法：在 `pipe` 函数之后，调用 `fork` 来产生一个子进程，之后在父子进程中执行不同的操作。在示例代码中，父进程操作写端，而子进程操作读端。同时，示例代码也为我们演示了使用 `pipe` 系统调用的习惯：`fork` 之后，进程在开始读或写管道之前都会关掉不会用到的管道端。

从本质上说，管道是一种只在内存中的文件。在 UNIX 中使用 `pipe` 系统调用时，进程中会打开两个新的文件描述符：一个只读端和一个只写端，而这两个文件描述符都映射到了同一片内存区域。但这样建立的管道的两端都在同一进程中，而且构建出的管道两端是两个匿名的文件描述符，这就让其他进程无法连接该管道。在 `fork` 的配合下，才能在父子进程间建立起进程间通信管道，这也是匿名管道只能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 管道的测试

我们下面就来填充函数实现匿名管道的功能。思考刚才的代码样例，要实现匿名管道，至少需要有两个功能：管道读取、管道写入。

要想实现管道，首先我们来看看本次实验我们将如何测试。lab6 关于管道的测试有两个，分别是user/testpipe.c与user/testpiperace.c。

首先我们来观察 testpipe 的内容

Listing 19: testpipe 测试

```

1  #include "lib.h"
2
3
4  char *msg =
5      "Now is the time for all good men to come to the aid of their party.";
6
7  void
8  umain(void)
9  {
10     char buf[100];
11     int i, pid, p[2];
12
13     if ((i = pipe(p)) < 0) {
14         user_panic("pipe: %e", i);
15     }
16
17     if ((pid = fork()) < 0) {
18         user_panic("fork: %e", i);
19     }
20
21     if (pid == 0) {
22         writef("[%08x] pipereadeof close %d\n", env->env_id, p[1]);
23         close(p[1]);
24         writef("[%08x] pipereadeof readn %d\n", env->env_id, p[0]);
25         i = readn(p[0], buf, sizeof buf - 1);
26
27         if (i < 0) {
28             user_panic("read: %e", i);
29         }
30
31         buf[i] = 0;
32
33         if (strcmp(buf, msg) == 0) {
34             writef("\npipe read closed properly\n");
35         } else {
36             writef("\ngot %d bytes: %s\n", i, buf);
37         }
38
39         exit();
40     } else {

```

```

41         writef("[%08x] pipereadeof close %d\n", env->env_id, p[0]);
42         close(p[0]);
43         writef("[%08x] pipereadeof write %d\n", env->env_id, p[1]);
44
45         if ((i = write(p[1], msg, strlen(msg))) != strlen(msg)) {
46             user_panic("write: %e", i);
47         }
48
49         close(p[1]);
50     }
51
52     wait(pid);
53
54     if ((i = pipe(p)) < 0) {
55         user_panic("pipe: %e", i);
56     }
57
58     if ((pid = fork()) < 0) {
59         user_panic("fork: %e", i);
60     }
61
62     if (pid == 0) {
63         close(p[0]);
64
65         for (;;) {
66             writef(".");
67
68             if (write(p[1], "x", 1) != 1) {
69                 break;
70             }
71         }
72
73         writef("\npipe write closed properly\n");
74     }
75
76     close(p[0]);
77     close(p[1]);
78     wait(pid);
79
80     writef("pipe tests passed\n");
81 }

```

实际上可以看出，测试文件使用 pipe 的流程和示例代码是一致的。先使用函数 `pipe(int p[2])` 创建了管道，读端的文件控制块编号<sup>2</sup>为 `p[0]`，写端的文件控制块编号为 `p[1]`。之后使用 `fork()` 创建子进程，注意这时父子进程使用 `p[0]` 和 `p[1]` 访问到的内存区域一致。之后子进程关闭了 `p[1]`，从 `p[0]` 读；父进程关闭了 `p[0]`，从 `p[1]` 写入管道。

lab4 的实验中，我们的 fork 实现是完全遵循 Copy-On-Write 原则的，即对于所有用户态的地址空间都进行了 `PTE_COW` 的设置。但实际上写时复制并不完全适用，至少在我们当前情景下是不允许写时拷贝。为什么呢？我们来看看 pipe 函数中的关键部分就能知晓答案：

```

1     int
2     pipe(int pfd[2])
3     {

```

<sup>2</sup>文件控制块编号是 int 型，user/fd.c 中 `num2fd` 函数可通过它定位文件控制块的地址。

```

4      int r, va;
5      struct Fd *fd0, *fd1;
6
7      if ((r = fd_alloc(&fd0)) < 0
8          || (r = syscall_mem_alloc(0, (u_int)fd0, PTE_V|PTE_R|PTE_LIBRARY)) < 0)
9          goto err;
10
11     if ((r = fd_alloc(&fd1)) < 0
12         || (r = syscall_mem_alloc(0, (u_int)fd1, PTE_V|PTE_R|PTE_LIBRARY)) < 0)
13         goto err1;
14
15     va = fd2data(fd0);
16     if ((r = syscall_mem_alloc(0, va, PTE_V|PTE_R|PTE_LIBRARY)) < 0)
17         goto err2;
18     if ((r = syscall_mem_map(0, va, 0, fd2data(fd1), PTE_V|PTE_R|PTE_LIBRARY)) < 0)
19         goto err3;
20
21     ...
22 }

```

在 pipe 中，首先分配两个文件描述符并为其分配空间，然后将一个管道作为这两个文件描述符数据区的第一页数据，从而使得这两个文件描述符能够共享一个管道的数据缓冲区。

**Exercise 6.1** 仔细观察 pipe 中新出现的权限位 PTE\_LIBRARY，根据上述提示修改 fork 系统调用，使得管道缓冲区是父子进程共享的，不设置为写时复制的模式。 ■

下面我们使用一张图来表示父子进程与管道的数据缓冲区的关系：

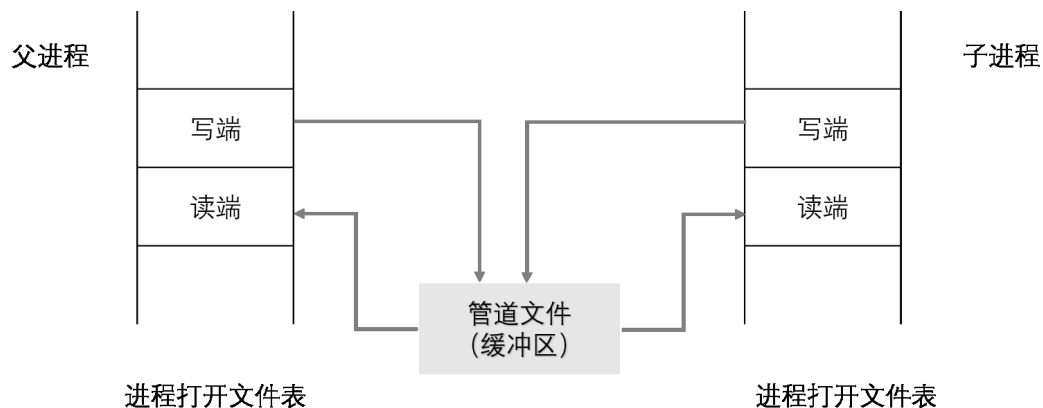


Figure 6.1: 父子进程与管道缓冲区

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下图：

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/pipe.c` 中轻松地找到 Pipe 结构体的定义，它的定义如下：

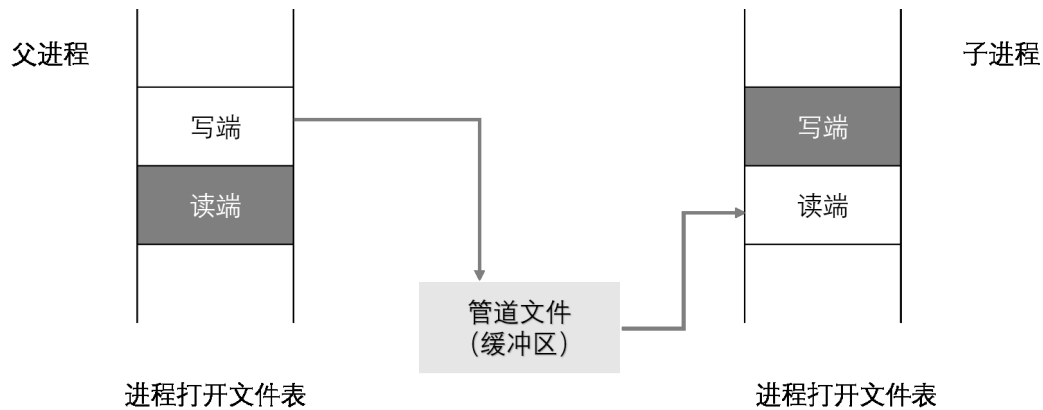


Figure 6.2: 关闭不使用的端口后

```

1  struct Pipe {
2      u_int p_rpos;           // read position
3      u_int p_wpos;           // write position
4      u_char p_buf[BY2PIPE]; // 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协调读写。一个管道有 `BY2PIPE`(32Byte) 大小的缓冲区。

这个只有 `BY2PIPE` 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%BY2PIPE`。

读者在从管道读取数据时, 要将 `p_buf[p_rpos%BY2PIPE]` 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%BY2PIPE]`, 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 < BY2PIPE` 时方可运行, 否则要一直挂起。

上面这些还不足以使得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, 我们必须得知道管道的另一端是否已经关闭。不论是在读者还是在写者中, 我们都需要对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, 则切换到其他进程运行。

**Note 6.2.1** Unix : If all file descriptors referring to the write end of a pipe have been closed, then an attempt to `read(2)` from the pipe will see end-of-file (`read(2)` will return 0) link : <http://linux.die.net/man/7/pipe>

那么我们该如何知晓管道的另一端是否已经关闭了呢？这时就要用到我们的 `static int _pipeisclosed(struct Fd *fd, struct Pipe *p)` 函数。而这个函数的核心，就是下面我们要讲的恒成立等式了。

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。既然管道数据缓冲区 `h` 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，就如同上图所示。`pageref` 函数能得到页的引用次数，所以实际上有下面这个等式成立：

$$\text{pageref}(\text{rfd}) + \text{pageref}(\text{wfd}) = \text{pageref}(\text{pipe})$$

**Note 6.2.2** 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` 的实现实际上就是查询虚页 `P` 对应的实际物理页，然后返回其 `pp_ref` 变量的值。

这个等式对我们而言有什么用呢？假设我们现在在运行读者进程，而进行管道写入的进程都已经结束了，那么此时就应该有：`pageref(wfd) = 0`。所以就有 `pageref(rfd) = pageref(pipe)`。所以我们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 6.2** 根据上述提示与代码中的注释，填写 `user/pipe.c` 中的 `piperead`、`pipewrite`、`_pipeisclosed` 函数并通过 `testpipe` 的测试。 ■

**Note 6.2.3** 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 管道的竞争

我们的小操作系统采用的是时间片轮转调度的进程调度算法，这点你应该在 lab3 中就深有体会了。这种抢占式的进程管理就意味着，用户进程随时有可能会被打断。

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，执行同一段代码，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（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。

实际上，因为管道本身的共享性质，所以在管道中有一系列的竞争情况。在当前这种不加锁控制的情况下，我们无法保证 `_pipeisclosed` 用于管道另一端关闭的判断一定返回正确的结果。

我们重新看之前写的 `_pipeisclosed` 函数。在这个函数中我们对 `pageref(fd structure)` 与 `pageref(pipe)` 行了等价关系的判断。假如不考虑进程竞争，不论是在读者还是写者进程中，我们会认为：



- 对 fd 和对 pipe 的 pp\_ref 的写入是同步的。
- 对 fd 和对 pipe 的 pp\_ref 的读取是同步的。

但现在我们处于进程竞争、执行顺序不定的情景下，上述两种情况现在都会出现不同步的现象。想想看，如果在下面这种场景下，我们前面提到的等式6.2.3还是恒成立的吗：

```

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  }
```

- 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]) 中，p[0] 已经解除了对 pipe 的映射 (unmap)，还没有来得及解除对 p[0] 的映射，时钟中断产生，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/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？

那看到这里你有可能问：在 close 中，既然问题出现在两次 unmap 之间，那么我们为什么不能使两次 unmap 统一起来是一个原子操作呢？要注意，在我们的小操作系统中，只有 syscall\_ 开头的**系统调用函数**是原子操作，其他所有包括 fork 这些函数都是可能会被打断的。一次系统调用只能 unmap 一页，所以我们是不能保持两次 unmap 为一个原子操作的。那是不是一定要两次 unmap 是原子操作才能使得 \_pipeisclosed 一定返回正确结果呢？

**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。

答案当然是否定的，\_pipeisclosed 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)`。所以我们要解决的实际上是当 **`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，即要满足：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 的引用次数的 -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。那么若调换 fd 和 pipe 在 close 中的 unmap 顺序，能否解决这个问题呢？

**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 **pipeclose** 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，那么对于 dup 中出现的情况又该如何解决？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 的顺序可以解决上述情景中出现的进程竞争问题。

那么下面根据你所思考的内容进行实践吧：

**Exercise 6.3** 修改 user/pipe.c 中的 pipeclose 与 user/fd.c 中的 dup 函数以避免上述情景中的进程竞争情况。

我们通过控制修改 pp\_ref 的前后顺序避免了“写数据”导致的错觉，但是我们还解决第二个问题：读取 pp\_ref 的同步问题。

同样是上面的代码6.2.4，我们思考下面的情景：

- 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 `__pipeisclosed` 函数，`pageref(fd)` 值为 2(父进程和子进程都打开了 p[0])，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。要写的的数据较多，写到一半时钟中断产生，内核切换到子进程运行。
- 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。

上述现象出现的根源在哪里呢？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。

**Exercise 6.4** 根据上面的表述，修改 `_pipeisclosed` 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。 ■

## 6.3 shell

shell 本质上也是一个用户进程。它解释 shell 命令的工作是通过创建并运行子进程来完成的：对于每个 shell 命令，都有一个对应的可执行文件来完成该命令所要完成的工作，shell 需要根据所得到的命令来创建执行相应可执行文件的子进程，从而完成命令的解释工作并得到结果。

为了能够使用 shell，首先需要使我们的操作系统响应键盘的输入，让 shell 能够获得用户输入的命令。我们已经使用汇编完成了 `sys_cgetc` 函数，你可以在 `lib/getc.S` 中看到它的具体实现。但是光有 `sys_cgetc` 函数还不够，你需要增加系统调用 `syscall_cgetc` 来获取键盘的输入。

**Exercise 6.5** 模仿现有的系统调用，增加系统调用 `syscall_cgetc`。（提示：`sys_cgetc` 函数不需要传入参数） ■

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。

**Exercise 6.6** 根据以上描述，补充完成 `user/sh.c` 中的 `void runcmd(char *s)`。 ■

## 6.4 实验正确结果

### 6.4.1 管道测试

管道测试有两个文件，分别是 `user/testpipe.c` 和 `user/testpiperace.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。在 `testpiperace` 的测试中应当出现 `race didn't happen` 是正确的。

### 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现象：

```
1 .....
2
3 ::
4
5 ::          Super Shell  V0.0.0_1          ::
6
7 ::
8
9 .....
10  \ $
```

使用不同的命令会有不同的效果：

- 输入 `ls.b`，会显示一些文件和文件夹；
- 输入 `cat.b`，会有回显现象出现；
- 输入 `ls.b | cat.b`，和 `ls.b` 的现象应当一致；

## 6.5 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 -进程上下文的 PC 值
- 思考 -TIMESTACK 的含义
- 思考 -不公的调度

## A.1 实验目的

1. 了解操作系统实验的意义
2. 掌握操作系统实验环境搭建
3. 了解实验用到的开发工具

在本章中，我们介绍了一般的 linux 环境下，操作系统开发的环境搭建的过程，这个过程对不同的操作系统开发，有一定的普适性。但目前整个实验环境在虚拟机上已经提供，如果无意在本地搭建环境且认为自己足够了解实验意义和相关工具的话，可以跳过这个章节。

## A.2 了解操作系统实验

关于操作系统的教程可以说是数不胜数，但是对于一个从来没有写过或者参与过操作系统开发的人来说，这些书读起来总觉得有隔阂，没有一个感性的认识。这其中的根本原因，在于初学者一开始就面对一个完整的操作系统，或者是面对前人积累了几十年的一系列理论成果（比如经典的页面替换算法）。而这些书的作者无论多擅长写作，或者写的代码多么优秀，要一个初学者理清其中的头绪都是相当困难的。

操作系统教程中往往注重一些理论知识的讲解，对具体实现的细节描述不够。初学者要是真的想自己动手写一个操作系统，会发现理论书籍一下子变得毫无用武之地。初学者甚至连如何将一个已经写好的操作系统原型加载到内存中都要花费很长时间，更不要说自己写出一个比较完善的操作系统。因此，要对操作系统有一个全面的理解，不仅要精读操作系统理论书籍，更要亲自动手编写代码，只有理论联合实际才能够完全掌握操作系统的精髓所在。

很多国际顶尖的高校为了操作系统的教学，编写了专门的操作系统。JOS 是由麻省理工学院教学专用的操作系统，麻省理工学院的操作系统课程之前一直用的是 JOS 操

作系统，该系统也是在很多学生的努力下完善起来的。这个操作系统实验是 MIT 公开课中的一个课程，搭建了一个基础的操作系统框架，让学生一步一步地实现操作系统中的内存管理、中断和异常处理、进程的创建和调度、SMP 支持、文件系统等功能，对于理解操作系统的实现非常有帮助。做完这个实验之后，可以对操作系统的实现有一个整体的、又不失细节的理解。这个操作系统还有一个特点就是它是一个微内核的系统，如果想要对比宏内核（比如 Linux）和微内核，也是一个很不错的选择。Harvard 大学的 David A.Holland 等也设计了 OS161 操作系统用于实现操作系统实验教学。因此，我们可以站在巨人的肩膀上，参考他们的设计思路、方法和源代码，尝试完成一个可以在 mips 上运行的小型操作系统。“麻雀虽小，五脏俱全”，在完成这个基于 mips 架构的小操作系统过程中，我们可以更好的理解操作系统启动、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 等主要操作系统内核功能，每个实验包含的内核代码量（C、汇编、注释）在 1000 行左右，充分体现了“小而全”的指导思想。

这个基于 mips 的小操作系统是运行在 mips 体系结构上的，而我们平常使用的都是基于 x86 体系结构的计算机，所以为了调试和开发的方便，我们采用 mips 硬件模拟器 GXEMUL。实验的开发环境是 GNU 的 gcc、gas 等工具。整个实验的运行和开发环境都在 Linux 中。

那么，我们如何来一步步地实现这个基于 mips 的小操作系统呢？根据一个操作系统和设计实现过程，可以有如下的实验步骤。

1. 启动操作系统的 bootloader，用于了解操作系统启动前的状态和准备工作，了解运行操作系统的硬件支持，操作系统如何加载到内存中，理解中断等。
2. 物理内存和虚拟内存管理子系统，理解分段和分页，了解操作系统如何管理物理内存和虚拟内存。
3. 进程管理和中断管理，了解进程的创建、执行、切换和结束，了解中断的完整过程。
4. 系统调用，了解系统调用的实现过程。
5. 文件系统，了解文件系统的具体实现，与进程管理等关系。
6. Shell，了解 Shell 的实现过程。

其中每个开发步骤都是建立在上一个步骤之上，一步一步最终形成一个比较完善的小操作系统，如图A.1所示。

## A.3 操作系统实验工具

### A.3.1 交叉编译器

首先，整个实验是建立在 MIPS 上的，通过之前的计算机组成等课程的学习我们知道，不同类型的 CPU 有不同的 ISA。我们本地常使用的是 Intel 的 X86 指令集，或

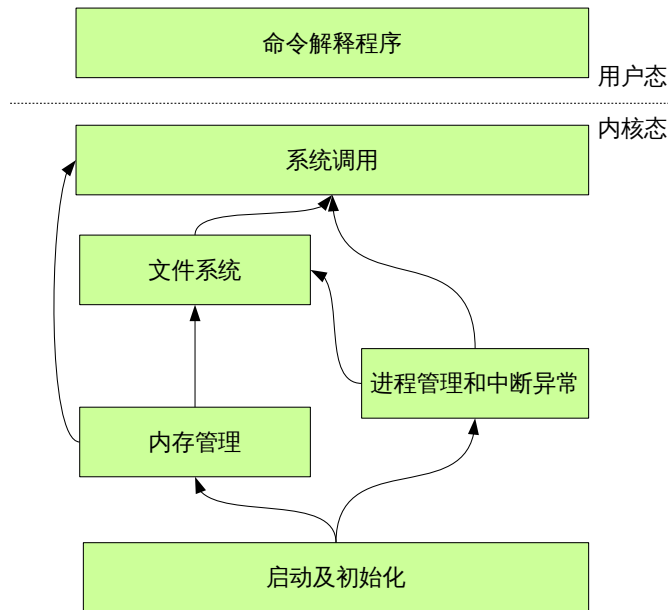


Figure A.1: 六个实验间的关系

AMD64(EMT64) 等指令集。而我们的小操作系统的目标机是 MIPS 指令集的。但我们需要使用交叉编译器来完成编译过程。我们的交叉编译器运行于 x86 平台上，但编译产生的二进制文件却是在 mips 平台上运行的。如图A.2 所示，编译器所运行的平台与其编译出来的程序的平台不同，因此叫做交叉编译器。

**Note A.3.1** 严格意义上来说，所谓平台包含了两个概念：体系结构 (Architecture)、操作系统 (Operating System)。一个体系结构上，可以运行多种不同的操作系统。而同一个操作系统，也可以在不同体系结构上运行。举例来说，我们常说的 x86 Linux 平台实际上是 Intel x86 体系结构和 Linux for x86 操作系统的统称；而 x86 WinNT 平台实际上是 Intel x86 体系结构和 Windows NT for x86 操作系统的简称。

交叉编译器通常用于解决目标平台因性能不足等原因难以直接在其上开发的问题。举个例子，假如我们需要为一个 500MHz 的 ARM 开发程序，因为其性能、工具、环境等原因，我们很难直接在这个 ARM 的板子上进行开发。因此，我们可以选择一个性能强劲、工具齐全的 x86 PC 机上完成程序，再通过交叉编译器编译为 ARM 指令集的程序。通过这样的方式，我们就能够轻松地开发 ARM 程序了。

### A.3.2 Linux 系统

Unix 是一个经典的操作系统。目前操作系统中的很多设计思想以及算法均是源自 Unix 的。但 Unix 在商业化后，我们很难以相对廉价的方式直接接触 Unix。目前相对较为完善的自由的 Unix 衍生版仅有 FreeBSD 等寥寥几个，其上的软件生态环境等等十分有限。不过，幸运的是，Linux 为我们提供了一个相对良好的接触 Unix 思想的机会。Linux 是一个自由的类 Unix(Unix-like) 系统，兼容 POSIX 标准，为我们的实验提供了一个良好的环境。近些年 Linux 发展迅猛，其上的软件环境等也十分丰富，这一点相较于 BSD 家族来说要好很多。我们实验中采用了 GNU 的工具，这一套工具主要是



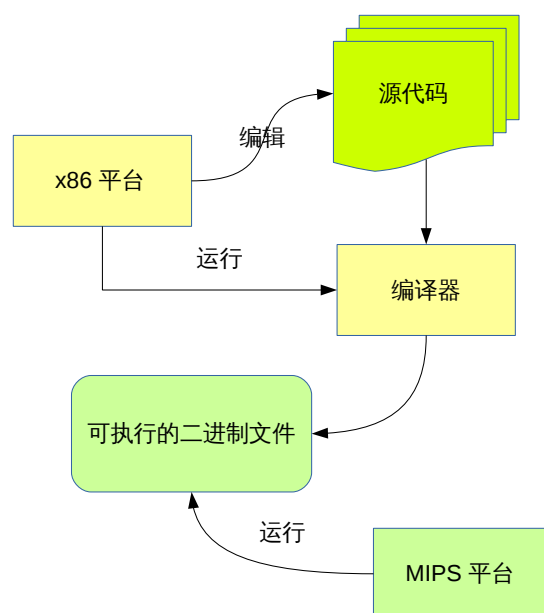


Figure A.2: 交叉编译

为 Unix 类系统编写的。同时，我们的小操作系统中有很多 Unix 风格的设计，因此，采用 Linux 平台做实验更有利于大家理解实验中的很多内容。实验为大家提供的虚拟机上的环境是一个无图形界面的 Ubuntu 12.04，大家通过 ssh 远程连接到虚拟机上去做实验。通过命令行界面来进行全部的操作。很多同学以前没有太接触过这种操作方式，这一点需要大家慢慢熟悉。

**Note A.3.2** 事实上，Linux 仅仅是一个操作系统内核。一般一个完整的基于 Linux 的操作系统还包含 GNU 的各类工具软件，图形环境，应用软件等等其他一系列的软件。但习惯上，将以 Linux 为内核的操作系统简称为 Linux。由于大家长期以来习惯在 Linux 内核上使用大量的 GNU 软件，所以，Richard Stallman 认为将这类操作系统称为“GNU/Linux”更为恰当。

## A.4 实验环境配置

### A.4.1 Linux 操作系统

由于所有的实验都是在 Linux 下完成的，所以需要 Linux 操作系统。我们选用设计上更为用户友好的 Ubuntu 系统。有两种可供参考的安装方式，一是通过虚拟机来安装 Linux 系统，二是直接安装 Linux 系统。这里考虑到大部分同学对于 Linux 不同熟悉，我们介绍虚拟机安装的方法。

首先，我们需要下载操作系统的镜像。为了下载速度考虑，我们选择从中国科学与技术大学的镜像站<http://mirrors.ustc.edu.cn/>上下载。点击该页面右侧的**获取安装镜像**，出现如图A.3所示的界面，版本选择与实验环境一致的 12.04。

接下来，我们下载虚拟机软件。虚拟机相当于一台虚拟的机器，虚拟机上的操作就好像是在操作另一台机器，对本机不会造成影响，是相对安全的一种体验 Linux 的



Figure A.3: 获取安装镜像

方案。这里我们选择采用 VirtualBox 虚拟机。可以去官网<https://www.virtualbox.org/wiki/Downloads>下载，也可以去百度上自行寻找国内的较快的下载地址。当然，你也可以选择其他的虚拟机软件，如 VMware（性能更佳，但是是专有软件，你懂得），具体操作相仿。但 Virtualbox 是自由软件，所以出于版权方面的考虑，这里以 VirtualBox 为例。

安装 VirtualBox（相信安装这种小事一定难不住你）后，打开它，看到如图A.4所示的界面。

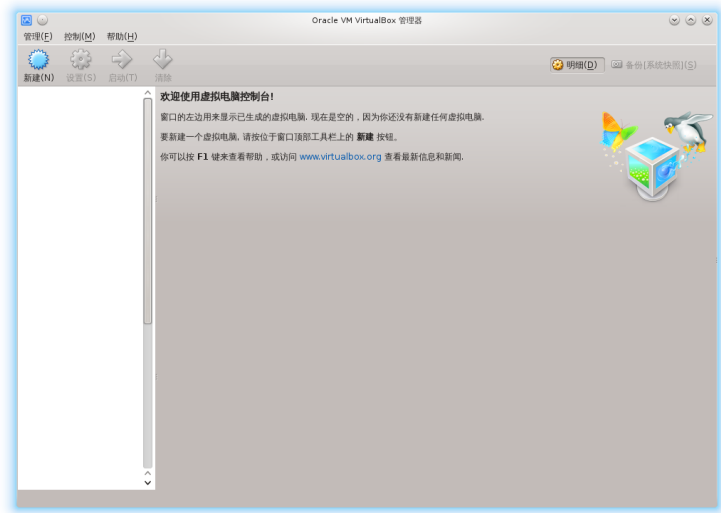


Figure A.4: VirtualBox 虚拟机界面

点击**新建**，出现如图A.5所示的对话框。类型选择 **Linux**，版本选择 **Ubuntu(64bit)**（这里如果你下载的时候选择的是 i386 版本则选择 **Ubuntu(32bit)**）。点击**下一步**，内

存选择 **1024MB**。下一步，选择**现在创建虚拟硬盘**，后面的设置如无特殊需求不必改动，一直下一步即可。直到选择磁盘大小处，建议选择 **20GB**，同时选定一个至少有 20GB 空间的位置放置磁盘镜像文件。

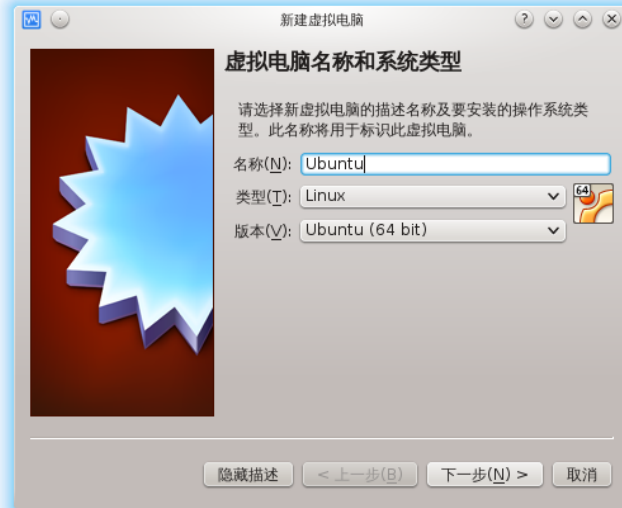


Figure A.5: 新建虚拟机

之后选中我们刚才建立的虚拟机，点**设置**，**存储**，点选**光盘**加号形状图标，添加**虚拟光驱**，如图A.6然后**选择磁盘**，选中之前下载好的 Ubuntu 的 ISO 文件。

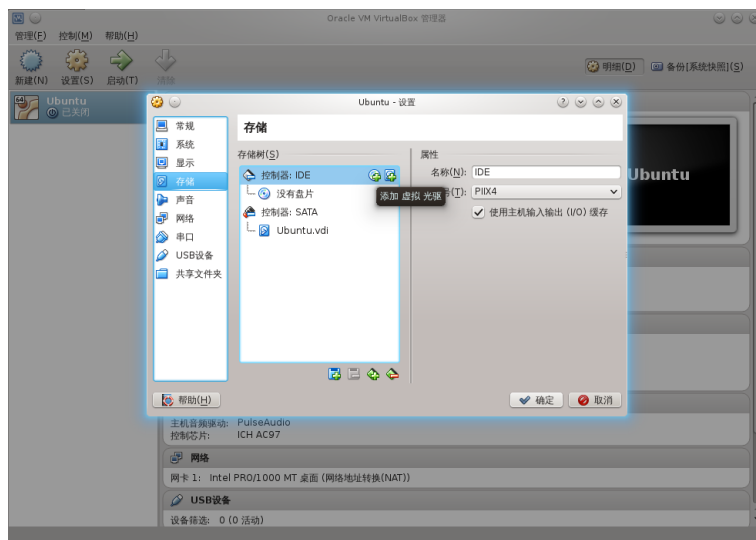


Figure A.6: 选择启动镜像

选中虚拟机，点击**启动**，弹出虚拟机画面，启动后进入 Ubuntu 的安装界面（见图A.7）。左侧可以选择语言，这里我们选择 **English**，并点击 **Install Ubuntu**。（当然，按理说选择中文也是没问题的，但这里为了避免任何不必要的麻烦，我们选择了英文）。

之后由于我们没有改镜像源等设置，为了安装速度考虑，不要选择 **Download**

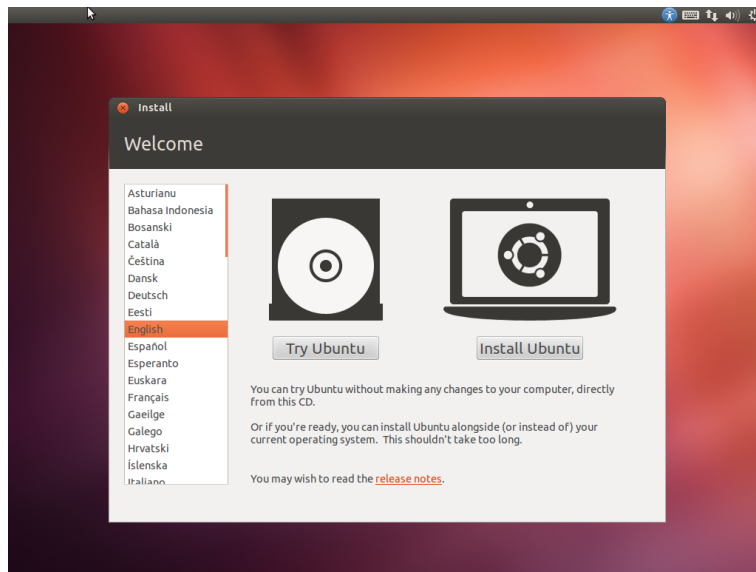


Figure A.7: 选择启动镜像

updates while installing。以免 Ubuntu 从缓慢的官方服务器下载更新。**Install this third-party software** 选项与我们实验无关，你可以自行选择是否勾选。下一步选择 **Erase disk and install Ubuntu**。由于我们是在虚拟机上安装，且使用了新建的虚拟磁盘，所以自然可以大胆地让 Ubuntu 清空整个磁盘并安装。下一步一般不需要做改动，点击 **Install Now** 就好。之后输入密码，等待安装结束就好。

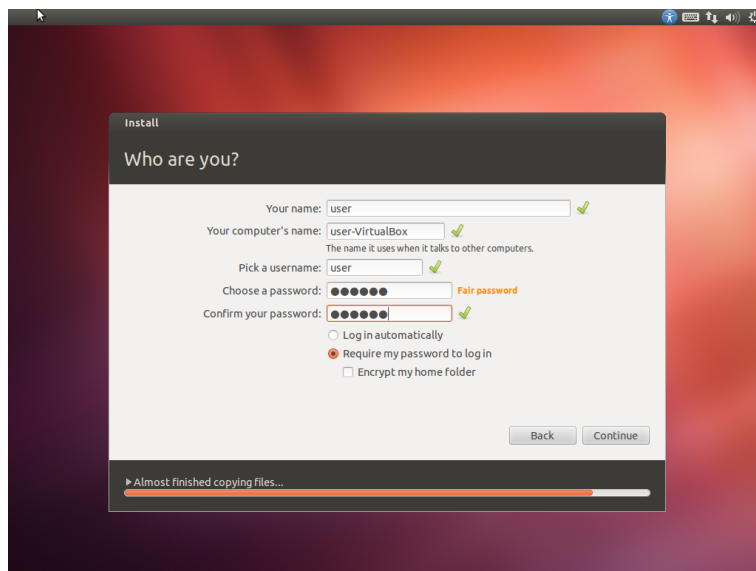


Figure A.8: Ubuntu 安装中新建用户的界面

安装结束后点击 **Restart** 重启。重启时注意，安装镜像关闭时一般会提示你取出光盘并按回车键。光盘镜像正常情况下会被自动弹出。一般来说，你需要做的仅仅是按下回车键并等待重启。这里注意，有时候由于种种原因它不会输出这句提示，遇到这种情况就需要发动你的第六感了:) 感觉系统不再动了以后敲一个回车就好。

重启完成后登入 Ubuntu 界面，接下来我们安装增强功能。点击虚拟机的**设备**，然

后选择**安装增强功能**。之后会 Ubuntu 会选择弹出对话框询问是否自动运行，如图A.9所示。点击 **Run** 后会弹出对话框要求输入密码，输入密码后会启动一个命令行，开始执行安装脚本，看到 **Press Return to close this window** 后按回车键结束安装。手动重启虚拟机，重启后虚拟机的分辨率会变成相对正常的状态，说明增强功能已经成功安装。

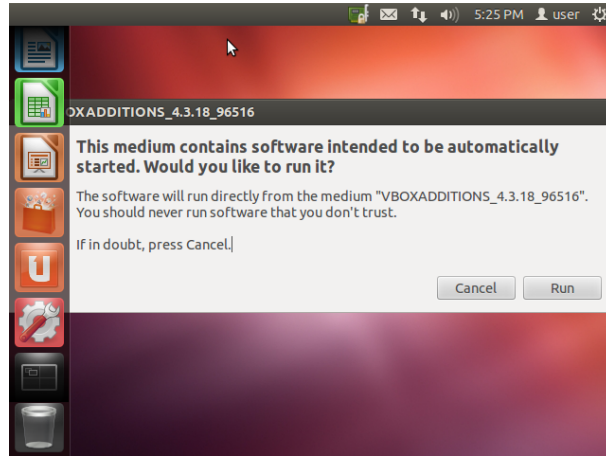


Figure A.9: Ubuntu 安装中新建用户的界面

至此，一个 Ubuntu 环境就搭建完成了。你可以在上面学习并熟悉 Linux 的相关指令，并编写我们的实验代码。

**Exercise A.1** 安装一个 Linux 环境，并尝试学会使用 ls、cat、cd、cp、mv 这五条指令

#### A.4.2 安装交叉编译器

交叉编译器的下载可以访问<http://ftp.sunet.se/pub/Linux/distributions/eldk/4.1/mips-linux-x86/iso/> 到其中下载 **mips-2007-01-21.iso** 文件。这里下载网速较慢，建议大家下载完一份后相互拷贝，一般课程中心也会提供相应的文件。

在 Ubuntu 下，按住 Ctrl+Alt+t 快捷键，可以打开一个终端，我们在终端中

```

1  # 建立一个用于挂载 ISO 镜像的目录
2  sudo mkdir /mnt/mipsiso
3  # 挂载 iso 文件
4  sudo mount -o loop mips-2007-01-21.iso /mnt/mipsiso
5  # 切换到/mnt/mipsiso 文件夹中
6  cd /mnt/mipsiso
7  # 安装 32 位运行库（仅在 64 位系统上需要执行此步骤）
8  sudo apt-get install ia32-libs
9  # 运行安装脚本
10 sudo ./install -d /opt/eldk

```

执行完上面的指令后，检查/opt/eldk/usr/bin 下是否有以 mips\_4KC 开头的一系列工具。打开命令行，尝试运行其中的 gcc，如果看到如下输出，则说明一切 OK ~

```

1  # 注意, 此时需要位于/opt/eldk/usr/bin 目录下
2  $ ./mips_4KC-gcc
3  Reading specs from /opt/eldk/usr/bin/./lib/gcc/mips-linux/4.0.0/specs
4  Target: mips-linux
5  Configured with: /opt/eldk/build/mips-2007-01-21/work/usr/src/denx/
6  BUILD/crosstool-0.35/build/gcc-4.0.0-glibc-2.3.5-eldk/mips-linux/
7  gcc-4.0.0/configure --target=mips-linux --host=i686-host_pc-linux-gnu
8  --prefix=/var/tmp/eldk.PyrfxY/usr/crosstool/gcc-4.0.0-glibc-2.3.5-eldk/
9  /mips-linux --with-headers=/var/tmp/eldk.PyrfxY/usr/crosstool/gcc-4.0.
10 0-glibc-2.3.5-eldk/mips-linux/mips-linux/include --with-local-prefix=
11 /var/tmp/eldk.PyrfxY/usr/crosstool/gcc-4.0.0-glibc-2.3.5-eldk/
12 mips-linux/mips-linux --disable-nls --enable-threads=posix
13 --enable-symvers=gnu --enable-languages=c,c++ --enable-shared
14 --enable-c99 --enable-long-long --enable-__cxa_atexit
15 Thread model: posix
16 gcc version 4.0.0 (DENX ELDK 4.1 4.0.0)

```

### A.4.3 安装仿真器

由于实验的操作系统内核是运行在 mips 体系结构上的, 而我们平常使用的是基于 x86 体系结构的 PC, 所以需要使用仿真器来仿真一个 mips 体系结构的环境, 来让我们的操作系统内核运行。在这个实验中我们使用的是 gxemul 仿真器。

我们需要从<http://gxemul.sourceforge.net/src/>下载 gxemul-0.4.6.tar.gz。这里需要注意, 一定要下载 0.4 系列的 gxemul, 而不是最新的 0.6 系列, 以免发生和实验提供的文件发生不兼容的现象。调出终端, 切换到 gxemul-0.4.6.tar.gz 所在目录, 执行以下指令

```

1  # 解压 gxemul-0.4.6.tar.gz
2  tar -zxvf gxemul-0.4.6.tar.gz
3  # 进入 gxemul 文件夹
4  cd gxemul-0.4.6
5  # 配置并编译安装
6  ./configure
7  make
8  sudo make install
9  sudo cp gxemul /usr/local/bin

```

在命令行中执行 gxemul 指令, 看到如下输出则说明 gxemul 安装正确。

```

1  $ gxemul
2  GXemul 0.4.6    Copyright (C) 2003-2007  Anders Gavare
3  Read the source code and/or documentation for other Copyright messages.
4
5  usage: gxemul [machine, other, and general options] [file [...]]
6         or gxemul [general options] @configfile
7         or gxemul [userland, other, and general options] file [args ...]
8
9  Run  gxemul -h  for help on command line options.
10
11 No filename given. Aborting.

```

**Exercise A.2** 安装交叉编译器及 gxemul, 并确认其正常运行。 ■

## A.5 实验思考

**Thinking A.1** 思考以下几条指令有何作用？

- `ls -l`
- `mv test1.c test2.c`
- `cp test1.c test2.c`
- `cd ..`

我们的整个实验环境全部是运行于 Linux 基础上的，且所提供的虚拟机是没有安装图形界面的，需要通过 ssh 远程连接访问。实验的所有操作全部在命令行下完成。因此，在开始实验前，你需要掌握最基本的一些命令，以保证你**可以在命令行下生存**

**Thinking A.2** 思考 `grep` 指令的用法，例如如何查找项目中所有的宏？如何查找指定的函数名？

`grep` 指令可以在文件中**匹配特定的文本**。支持递归匹配（即查找当前目录及子目录下所有文件）、正则表达式等一系列有用的功能。当你需要在整个项目目录中查找某个函数名、变量名等等的时候，`grep` 将是你手头一个强有力的工具。这里给一个提示，`grep` 的 `-a`、`-i`、`-r` 三个选项使用率较高，可以着重了解一下。

**Thinking A.3** 思考 `gcc` 的 `-Werror` 和 `-Wall` 两个参数在项目开发中的意义。

编译器警告很多时候可以帮你发现一些代码上的错误。善用编译器警告可以大大加快你调试的进度。