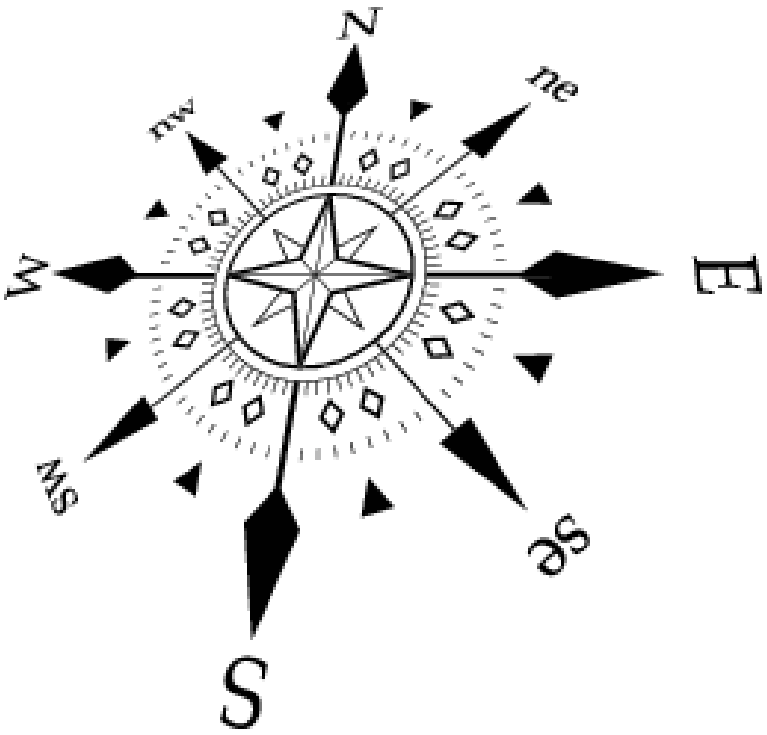
MOS 操作系统实验
指导书

操作系统课程实验任务及相关说明

带你体验自己动手完成一个小操作系统的乐趣



School of Computer Science and Engineering
Beihang University

2022 年 5 月 19 日

Powered by LaTeX

编者序

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

教师寄语

“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，我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“初识操作系统”、“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）等七个部分。

1. 初识操作系统：在只有命令行的环境下，掌握 Linux 的基础命令、常用工具以及 Git 的使用。

2. 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。

3. 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。

4. 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。

5. 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。

6. 文件系统：实现一个基于块设备的文件系统。

7. 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！

目录

引言

操作系统是计算机系统中软件与硬件的纽带，该课程内容丰富，既要讲授关于操作系统的基础理论，又要让学生了解实际操作系统的设计与实现。操作系统实验设计正是该课程实践环节的集中表现，不仅使学生巩固理论学习的概念和原理，同时培养学生的工程实践能力。国内外著名大学都非常重视操作系统实验设计，例如 MIT 的 Frans Kaashoek 等设计的 JOS 和 xv6 教学操作系统、Harvard 大学的 David A. Holland 等设计了 OS161 操作系统用于实现操作系统实验教学。

我们尝试了 MINIX、Nachos、Linux、Windows 等很多操作系统实验，发现以 Linux 和 Windows 为基础的实验，由于系统规模庞大，很难让学生建立起完整的操作系统概念，让学生专业知识碎片化，不符合系统能力培养目标。此外，操作系统涉及很多硬件相关知识，本身包含了并发程序设计等学生比较难理解的概念，因此学生的学习曲线很陡峭，如何让学生由浅入深，平滑地掌握这些知识是一个实验设计的难点。因此，我们实验的基本目标是：一学期内学生设计实现一个在实际硬件平台上运行的小型操作系统，该系统具备现代操作系统特征（虚存管理、多进程等），符合工业标准。

正好北京航空航天大学计算机学院希望将建立计算机组成原理、操作系统和编译原理等课程的一体化实验体系，培养学生的系统能力。我们操作系统课程设计采用了计算机组成原理课程中的 MIPS 指令系统（MIPS R3000）作为硬件基础，参考 JOS 的设计思路、方法和源代码，实现了一个可以在 MIPS 平台上运行的小型操作系统，包括了操作系统启动、物理内存管理、虚拟内存管理、进程管理、中断处理、系统调用、文件系统、Shell 等主要操作系统主要功能。为了降低学生学习的难点，我们采用了增量式实验设计，每个实验包含的内核代码量（C、汇编、注释）在 1000 行左右，为学生提供代码框架和代码示例，学生可以参考这些代码完成实验。每个实验可以独立运行和评测，但是后面的实验依赖前面的实验，学生实现的代码会一直从 lab1 贯穿到 lab6，最后实现一个完成的小型操作系统。

实验内容

操作系统实验内容分解为六个实验，最终让学生在一学期内自主开发一个小型操作系统。实验各个部分的相互关系如图 1 所示，具体实验内容如下：

1. **启动和系统初始化：** 通过 PC 启动的实验，掌握硬件的启动过程，理解链接地址、加载地址和重定位的概念，学习如何编写位置无关代码。
2. **内存管理实验：** 理解虚拟内存和物理内存的管理，实现操作系统对虚拟内存空间的管理。
3. **进程管理：** 通过设置进程控制块和编写进程创建、进程中止和进程调度程序，实现进程管理；编写通用中断分派程序和时钟中断例程，实现中断管理。
4. **系统调用：** 掌握系统调用的实现方法，理解系统调用的处理流程，实现本实验所需的系统调用。
5. **文件系统：** 通过实现一个简单的、基于磁盘的、微内核方式的文件系统，掌握文件系统的实现方法和层次结构。
6. **命令解释程序：** 实现具有管道，重定向功能的 shell，能够执行一些简单的命令。最后通过调试将六部分链接起来，使之成为一个能够运转的操作系统。

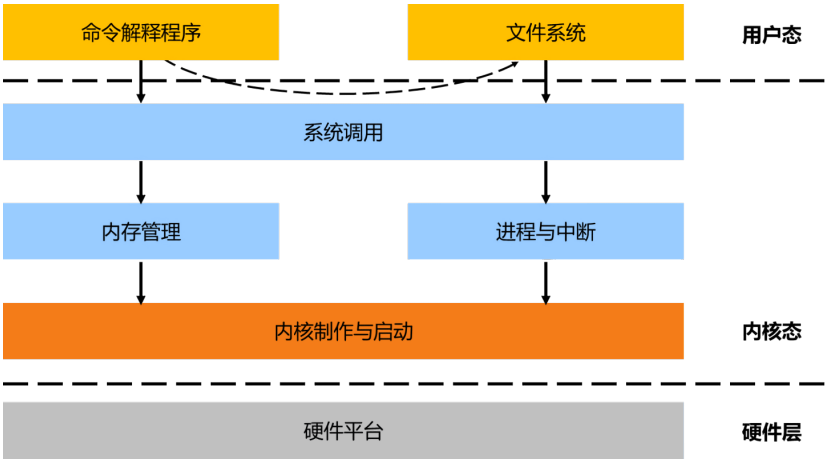


图 1: 实验内容的关系

由于课程开设的学年是大二，有不少学生对 Linux 系统、GCC 编译器、Makefile 和 git 等工具不熟悉，因此，我们增加了一个 lab0，主要介绍 Linux、Makefile、git、vi 和仿真器的使用以及基本的 shell 编程等。

实验设计

由于开发一个实际的操作系统难度和工作量很多，为了保证教学效果，我们在核心能力部分采用了微内核结构和增量式设计的原则，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。实验内容的设计满足以下条件：

1. 每个实验可独立运行与测试；便于调试与评测，让学生获得阶段性成果。
2. 每个实验内容包含相对独立的知识点，并只依赖它的前序实验。
3. 大部分学生可以在 2 周内完成一个实验，这样在一学期内可以完成整个实验。
4. 学生每个实验提交的代码一直伴随整个实验过程，他们可以不断改进完善代码。

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

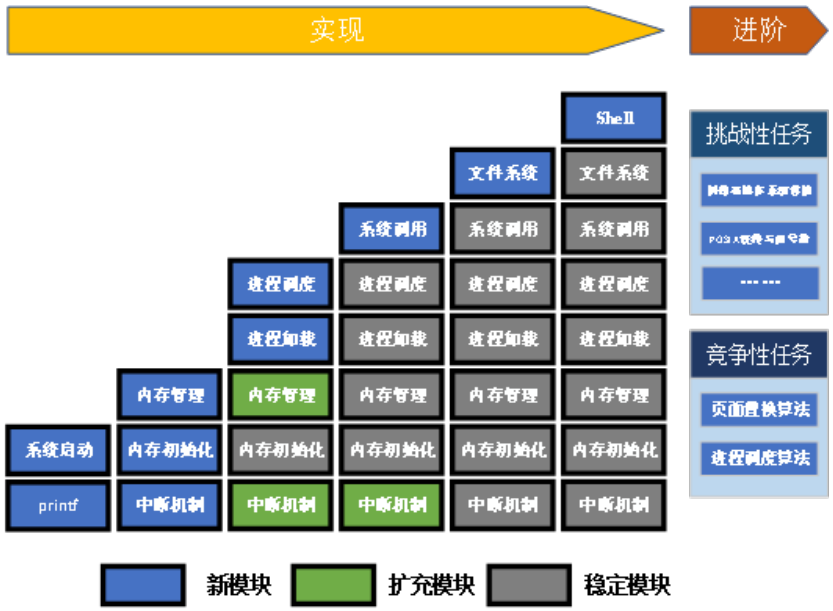


图 2: 增量式实验方法

我们采用了分层的方式，从基础到复杂来实现基本目标。我们将基本目标分为三个层次：

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

实验环境

作为一门实验课程，一次实验课程的整个流程包括，初始代码的发布，代码编写，调试运行，学生代码的提交，编译测试以及评分结果的反馈。为了方便学生和老师，我们设计了操作系统实验集成环境，采用 git 进行版本管理，保证学生之间的代码互不可见，

而老师和助教可以方便地查看每位学生的代码。整个环境的结构如图 3 所示。为了满足实验需求，整个系统分为以下几个部件：

a) **虚拟机平台，**包含实验需要的开发环境，例如 Linux 环境、交叉编译器、MIPS 仿真器等。

b) **Git 服务器，**包括学生各个实验的代码以及相关信息。

c) **自动评测和反馈，**这部分集成在 git 服务器中，下面会详细介绍。

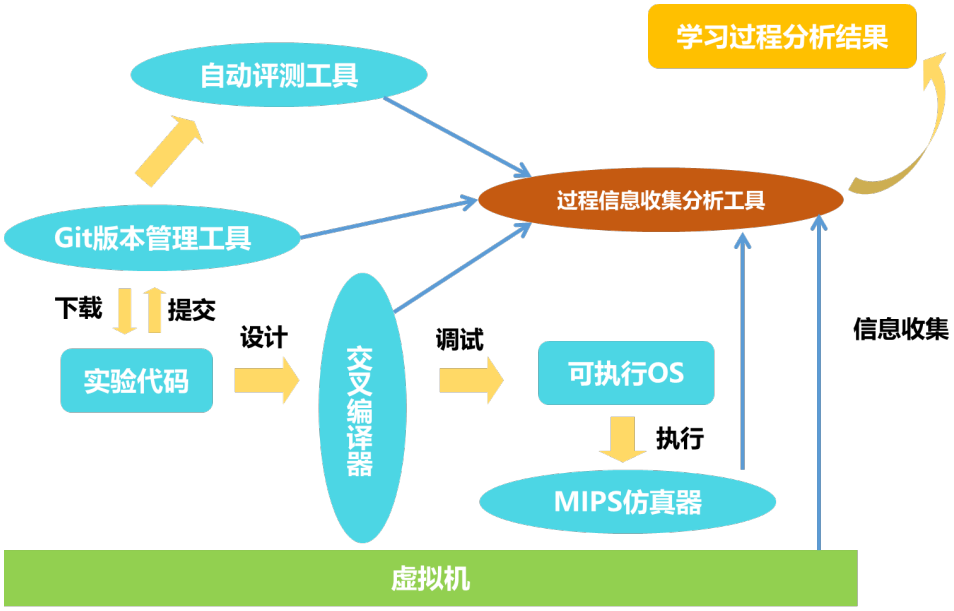


图 3: 操作系统实验集成环境

虚拟机平台

为了学生学习过程的数据收集的方便，同时为了尽量减少学生端的设备要求，我们提供虚拟机作为我们的实验后端，虚拟机中使用 Linux 系统，方便部署环境，学生的整个实验过程在虚拟机中进行。在虚拟机中，需要部署相应仿真器、编译器、文件编辑器等环境。我们推荐使用 Gxemul 作为仿真器，MIPS 的 gcc 交叉编译器作为编译器，vi 作为文件编辑器。这些工具已经部署在我们的虚拟机环境中，避免了软件版本冲突问题，为我们的自动评测系统打下了基础。同时，节约学生安装实验环境花费的时间。每位学生在虚拟机中拥有一个普通权限的 Linux 账号，学生只需要一个 ssh 客户端，就可以登录到我们的系统，完成实验。

Git 服务器

为了保证学生代码安全，同时提供方便的代码版本管理工具，我们提供了一个 git 服务器，每位学生的代码编写过程在虚拟机中完成，同时将代码托管在 git 服务器中，避免不必要的故障带来损失。同时，我们实验的发布，测试结果的反馈均通过 git 服务器

完成。在 git 服务器中，每个学生拥有一个独立的代码库，对于每位学生来说，只有自己的代码库是可见的，每位学生可以随意的下载和提交自己代码库中的代码。同时，所有学生的代码库对于助教和老师是可见的，所有的助教和老师都可以下载学生代码。服务器中允许学生自行建立分支，进行版本管理，但是 labX（X=1……6）这 6 个分支为我们认定的分支，这些分支分别代表实验 1 到实验 6 的最终结果，我们只会评测这些分支中的代码。

自动评测

由于学生人数众多，对于学生实验代码的评判是一个繁复、机械化的过程，通过每个助教和老师手动评测非常困难。该自动评测系统对学生提交的代码自动给出相应的评分。当学生执行代码提交后，触发评测系统。评测系统获取学生最新代码，依次完成编译、运行和测试，最终给出评测结果。最后，通过 git 服务器反馈结果给学生查看。如果学生通过了评测，则直接发布下一次实验的内容，让有能力的同学尽可能多完成实验内容。

CHAPTER 0

初识操作系统

0.1 实验目的

1. 认识操作系统实验环境
2. 掌握操作系统实验所需的基本工具

在本章中，我们需要去了解实验环境，熟悉 Linux 操作系统（Ubuntu），了解控制终端，掌握一些常用工具并能够脱离可视化界面进行工作。本章节难度非常低，旨在让大家熟悉操作系统实验环境的各类工具，为后续实验奠定基础。

0.2 初识实验

所谓工欲善其事必先利其器，我们需要对我们的环境和工具有了足够多的了解才能开始我们的工作，本章内容看似简单，但实则不可忽略，请大家认真对待。对于课程设计这名称，大家上次接触是《计算机组成课程设计》，也相信这门课程为大家带来了一段难忘而珍贵的记忆。就像“计组”一样，我们的课程也需要一定的开发环境，但好消息是，我们的环境约束要少得多，那么接下来熟悉一下我们的实验环境。

0.2.1 了解实验环境

实验环境整体配置如下：

- 操作系统：Linux 虚拟机，Ubuntu
- 硬件模拟器：Gxemul
- 编译器：GCC
- 版本控制：Git

Ubuntu 操作系统，一款开源的 GNU/Linux 操作系统，基于 Linux 内核实现，是目前较为流行的几个 Linux 发行版之一。GNU （GNU is Not Unix 的递归缩写），是一套计划，其中包含了三个协议条款，为我们带来了大量开源软件；Linux：严格意义上指代 Linux 内核，基于该内核的操作系统众多，具有免费、可靠、安全、稳定、多平台等特点。

GXemul，一款计算机架构仿真器，可以模拟所需硬件环境，例如我们需要的 MIPS 架构下的 CPU。

GCC，一套免费、开源的编译器，诞生并服务于 GNU 计划，最初名称为 GNU C Compiler，后支持了更多编程语言而改名为 GNU Compiler Collection，如你所见，简称没变，很多我们熟知的 IDE/集成开发环境的编译器用的便是 GCC 套件，例如 Dev-C++，Code::Blocks 等。我们的实验将使用 mips-C 交叉编译器。

Git，一款免费、开源的版本控制系统，我们的实验将用它为大家提供管理、发布、提交、评测等功能。0.5 小节将会为大家详细介绍 Git 为何物以及如何使用，故不在此赘述。

图 0.1: Ubuntu，GNU，Linux

0.2.2 通过 ssh 远程访问我们的环境

熟悉了实验环境之后，我们不禁产生了疑问：难道这些环境全部需要我们自行安装和配置？这里为大家带来另一个好消息：当然不是，这些环境已经为你们配置好了！

本实验总计 7 个 Lab 完全依赖于远程的数台虚拟机上，最终成果也需要通过这些机器进行提交，所以同学们几乎不用担心个人电脑“带不动”实验环境，你需要的仅仅是一个能够支持 ssh 协议的远程连接工具。对你来说，主要会有两种连接虚拟机的方式，一种是使用本机的 ssh 工具进行连接，另外一种是通过浏览器在实验平台直接进行访问。

由于通过 ssh 进行连接是绝大多数连接虚拟机和服务器的方式，通过浏览器直接访问背后也是的原理也是这个，因此我们先介绍通过 ssh 进行访问的做法。

一般 Liunx 或 Mac OS 等类 Unix 操作系统都会附带 ssh 客户端，即直接在终端使用 ssh 命令便可，Windows 平台一般不自带 ssh，需要下载第三方软件，在这里建议大家使用一款轻量级的，开源的，名为 PuTTY 的小工具，当然也不乏各种功能强大的工具，接触过 git for Windows 或对 Windows 10 有所研究的同学亦可使用 git bash 的以及 Windows 10 的几款 Linux 子系统。如果同学们是在 Mac OS 的终端或 Linux 系统的终端，命令行既可以实现相似的功能。本学期由于我们更新了 ssh 连接的机制，不建议使用 putty 等 ssh 客户端进行连接，直接使用下面介绍的网站的方式进行连接。

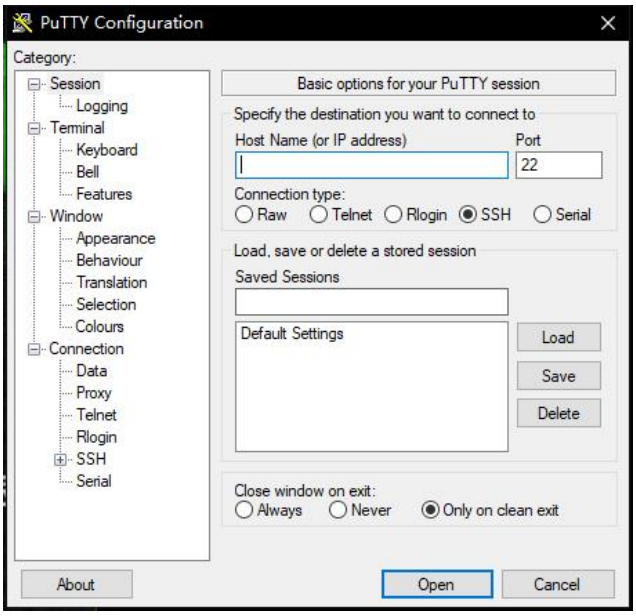


图 0.2: PuTTY 登录界面

Note 0.2.1 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。

0.2.3 通过网站远程访问我们的环境

为了方便同学们的访问，课程组给大家直接提供了网站可以直接访问虚拟机，不需要手动进行那么多复杂的配置。你可以直接按照下面的流程，也可以进到和前文同样的界面。

我们的虚拟环境的网站是`lab.os.buaa.edu.cn`，如果你在校外的话需要先使用 easy-connect 连上北航 vpn 再进行访问。

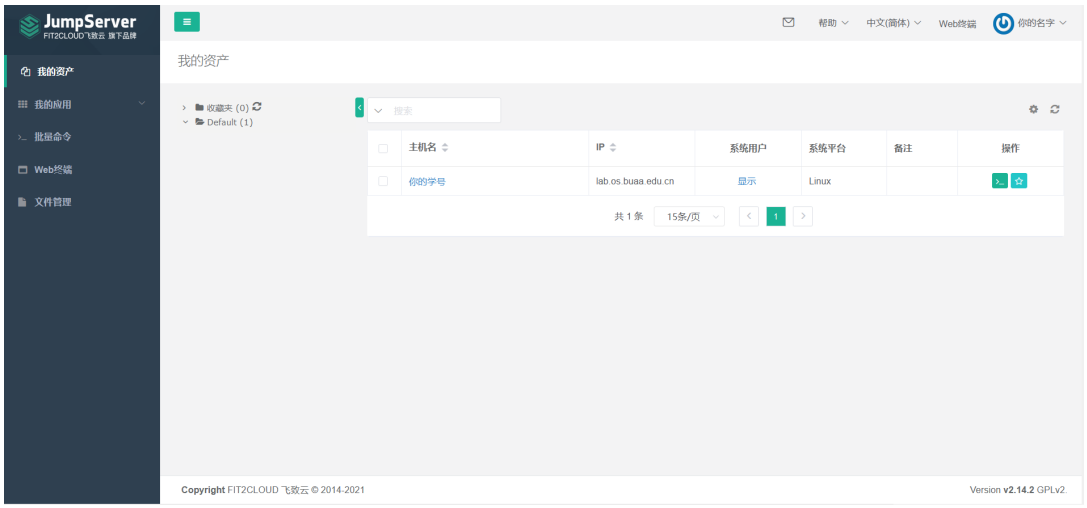


图 0.3: 虚拟环境网站页面

成功进入后，点击“操作”框里左侧的按钮即可进入到属于你的虚拟环境中了。

```
开始连接到 root-in-docker@lab.os.buaa.edu.cn 0.0
Welcome to Ubuntu 20.04.3 LTS (GNU/Linux 5.13.0-30-generic x86_64)

* Documentation:  https://help.ubuntu.com
* Management:     https://landscape.canonical.com
* Support:        https://ubuntu.com/advantage
Last login: Mon Feb 28 16:19:23 2022 from 172.17.0.1
root@20xxxxxx:~#
# 欢迎信息后会出现命令提示符，等待你输入命令。
```

0.2.4 接触 CLI Shell，告别 GUI Shell

当你使用上面提到的方法登录了自己的账号后，可能会有点不知所措并产生疑问：在我面前的这些东西是什么？

简单来说，这就是你们要接触的操作系统 Ubuntu，在这里为从未使用过命令行的同学们解释一下，当你阅读这份文档时，你所使用的一定是一款拥有图形化界面的操作系统，一般来讲是 Windows、Mac OS、Ubuntu 等，而你面前的这个黑黑的东西就是一个没有图形化界面的操作系统，也就是模拟终端，我们为什么剥夺了你使用图形化界面的权力呢，原因有二：1. 为了锻炼同学们在相对图形化恶劣的环境下工作的能力；2. 你们的任务通过终端全部可以实现。

当然上文的描述是不严谨的，你所面对的并不是真正的“Ubuntu 操作系统”，而是它的“壳”。一般的，我们管操作系统核心部分称为内核（英文：Kernel），与其相对的就是它最外层的壳（英文：Shell）。Shell 是用于访问操作系统服务的用户界面。一般而言，操作系统 shell 使用命令行界面（CLI）或图形用户界面（GUI），这个纯文本的界面就是命令行界面，它能接收你发送的命令，如果命令存在于环境中，它便会为你提供相应的功能。

在 Ubuntu 中，我们默认使用的 CLI shell 是 bash，也是一款基于 GNU 的免费、开源软件。

0.3 基础操作介绍

0.3.1 命令行

这一部分介绍的命令行操作是使用 Linux 系统的基础，10 年之后你可能会忘记操作系统课程，但是这一部分的内容那时也会经常和你打交道的。因此在接下来的几个部分，这里我们会一步一步带你在 Linux 系统下写出一个简单的 helloworld 程序，对于没有使用过 Linux 系统的同学，在这里可以打起精神！跟着我们一步一步跟着我们去认识这个系统。等未来如果大家对这些指令有些遗忘，也可以把指导书翻回这里，进行回顾。由于我们讲得是一个完整的故事，在这一部分，你可以在我们要求你进行输入的时候再进行输入，这样可以达到更好的体验效果。

现在，我们要正式与那黑黑的界面交流了，命令行界面 (Command Line Interface，简称 CLI) 中，用户或客户端通过单条或连续命令行的形式向程序发出命令，从而达到与计算机程序进行人机交互的目的。在 Linux 系统中，命令即是对 Linux 系统进行管理的一系列命令，其一般格式为：命令名 [选项] [参数]，其中中括号表示可选，意为可有可无。对于 Linux 系统来说，无论是中央处理器、内存、磁盘驱动器、键盘、鼠标，还是用户等都是文件，Linux 系统的管理命令是它正常运行的核心，与 Windows 的命令提示符（CMD）命令类似。Linux 命令在系统中有两种类型：内置 Shell（外壳）命令和 Linux 命令。

对于刚接触到命令行界面的各位萌新们来说，因为不清楚基本的 Linux 命令而呆呆地望着屏幕上的光标，想要做些什么却又不知所措是再正常不过的事。不过不要着急，万事开头难，学会并可以熟练使用下面介绍的一些基本操作之后，相信你们一定会对命令行界面有一个全新的认识，不再一头雾水。

0.3.2 Linux 基本操作命令

面对一个全新的界面，我们首先要知道当前目录中都有哪些文件，这时就需要使用 ls 命令，其输出信息可以进行彩色加亮显示，以分区不同类型的文件，是使用率较高的命令，其详细信息如下图所示。一般情况下，该命令的参数省略则默认显示该目录下所有文件，所以我们只需使用 ls 即可看到所有非隐藏文件，若要看到隐藏文件则需要加上-a 选项，若要查看文件的详细信息则需要加上-l 选项。

```
ls
用法:ls [选项]... [文件]...
选项（常用）：
-a                  不隐藏任何以. 开始的项目
-l                  每行只列出一个文件
```

看到这么复杂的参数，你可能会一头雾水，不过我们一般只会用到 ls，以及 ls -a 和 ls -l 三种形式。现在你试着在命令行里输入 ls，然后按下回车，这时你可能会发现只有一个命名为自己学号的文件夹，这是未来你进行 os 的文件夹，我们暂时不进入与操作这个文件夹，我们先试着创建一个文件，那么如何创建一个新的文件呢？这时候就要用到 touch 指令了。

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

我们这时输入 touch hello_world.c，这时候就在该目录下成功创建了一个文件，这时候我们再输入 ls，就能看到有一个 hello_world.c 了，这个时候你也可以试一试 ls -a 和 ls -l。怎么对这个 hello_world.c 进行编辑呢，这是下一节使用工具介绍中 vim 会介绍的内容，我们在这里先讲一些别的操作。

我们总不能把所有的文件都存在一个目录下吧，我们肯定需要通过文件夹来进行一些组织管理，这时我们可以使用 mkdir 命令创建文件目录（即 Windows 系统中的文件夹），该命令的参数为创建的新目录的名称，如：mkdir newdir 为创建一个名为 newdir 的目录。

```
mkdir
用法:mkdir [选项]... 目录...
```

我们现在输入 mkdir newdir 就在目录下创建了一个叫 newdir 的文件夹了，你可以再使用 ls 命令，就会发现多了一个蓝色的叫 newdir 的文件夹了。可是现在我们还在登录进来的文件夹，我们该如何进到这个文件夹呢，这时候需要 cd 命令。

```
cd
用法:cd [选项]... 目录...
```

我们在命令行输入 cd newdir 就可以进到刚刚新建的这个 newdir 的文件夹了，使用 ls 命令去看发现又是空空如也，当然你可以继续去创建一些文件和文件夹，这里就留给你继续探索了。但是你可能又遇到了个问题，该怎么返回上一级目录呢？在 Linux 系统里.. 表示上一级目录. 表示当前目录，所以你输入 cd .. 就能返回上一级目录了，如果输 cd . ，就是进入当前所在的目录，也就是什么都不做。

不知道你在刚刚 cd 目录的时候有没有发现命令行发生了一些区别，在命令行的 > 的左边，从~变成了，~/newdir，你可能猜到了，那一小段的字符串意义是当前所在的文件夹，那~指的是什么呢，指的其实就是你所登录用户的目录，在这里你可以试着输入 pwd，查看当前的绝对路径，这里你就知道~所蕴含的含义了，而之前提到的 cd 命令，也可以直接进行把要跳转到的文件夹改为绝对路径，如 cd /home，就跳转到了 home 根目录下 home 这个文件夹了。

Note 0.3.1 在需要键入文件名/目录名时，可以使用 TAB 键补足全名，当有多种补足可能时双击 TAB 键可以显示所有可能选项。你可以试着在屏幕上输入 cd /h 然后按下 TAB，就会自动补全为 cd /home，如果你输入的是 cd /，再按两下 TAB，会给你看你可以的选择，有些像 ls

如果你想删除一个空的目录，rmdir 则是你的选择，这里要注意只有空目录才可以使用 rmdir 命令删除。

```
rmdir
用法:rmdir [选项]... 目录...
```

那么目录非空时怎么办呢？这就需要用到搞破坏者最喜欢的 rm 命令了，rm 命令可以删除一个目录中的一个或多个文件或目录，也可以将某个目录及其下属的所有文件及其子目录均删除掉。对于链接文件，只是删除整个链接文件，而原有文件保持不变。

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

此外，rm 命令可以用-i 选项，这个选项在使用文件扩展名字符删除多个文件时特别有用。使用这个选项，系统会要求你逐一确定是否要删除。这时，必须输入 y 并按 Enter 键，才能删除文件。如果仅按 Enter 键或其他字符，文件不会被删除。与之相对应的就是-f 选项，强制删除文件或目录，不询问用户。使用该选项配合-r 选项，进行递归强制删除，强制将指定目录下的所有文件与子目录一并删除可以达到毁灭性效果。例如：rm -rf / 即可强制递归删除全盘文件，绝对不要轻易尝试！

Note 0.3.2 使用 rm 命令要格外小心，因为一旦删除了一个文件，就无法再恢复它。所以，在删除文件之前，最好再看一下文件的内容，确定是否真要删除。很多优秀的程序员会在目录下建一个文件夹类似回收站，如果要使用 rm 指令，先把要删除的文件移到这个文件夹里，然后再进行 rm，一定时间之后再对回收站里的文件进行删除。

讲了这么多 rm 相关的注意事项，我们现在再来试着使用 rm，我们先回到~目录下，假设我们要删除开始创建的 hello_world.c, 我们先要在该目录下创建一个回收站文件夹叫 trashbin，想想该怎么做？然后把 hello_world.c 拷贝到这个文件夹中，这里需要使用 cp 命令，该命令的第一个参数为源文件路径，命令的第二个参数为目标文件路径。

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

我们这时输入 cp hello_world.c trashbin/，就能够将 hello_world.c 移动到回收站里了，我们在输入 rm hello_world.c, 这时就删掉了 hell_world.c，可以使用 ls 命令看一看是不是被删掉了。

复制加删除其实就是移动，Linux 当然也有移动命令。移动命令为 mv，与 cp 的操作相似。写 mv hello_world.c trashbin/就是将 hello_world.c 移动到 trashbin 这个文件夹了。不过 mv 还有一种更有意思的用法，mv file ../file_mv 就是将当前目录中的 file 文件移动至上一层目录中且重命名为 file_mv。聪明的你应该已经看出来，在 Linux 系统中想要对文件进行重命名操作，使用 mv oldname newname 命令就可以了。

```
mv
用法:mv [选项]... 源文件... 目录
```

最后再介绍一个有趣的小指令，echo，如果你在屏幕上输入 echo hello_world，你

会发现它给你回显了一个 hello_world，看起来是一个复读机的功能，我们后面会用一些它的有趣用法。

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

```
man - manual
用法:man page
e.g.
man ls
```

最后，还有下面几个常用的快捷键介绍给同学们。

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

其中，如果你写了一个死循环，或者程序执行到一半你不想让它执行了，Ctrl+C 是你的很好的选择。Ctrl+Z 挂起程序后会显示该程序挂起编号，若想要恢复该程序可以使用 fg [job_spec] 即可，job_spec 即为挂起编号，不输入时默认为最近挂起进程。而如果你写了一个等到识别到 EOF 才停止的程序，你就需要输入 Ctrl+D 来当作输入了一个 EOF。

对其他内容感兴趣的同学可以自行百度或用 man 命令看帮助手册进行学习和了解。有了这些做基础，下一部分我们就来讲如何在 Linux 下写代码了。

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

0.4 实用工具介绍

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

0.4.1 Vim

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

(1) 创建文件：利用之前学到的 touch 命令创建 helloworld.c 文件（这时，使用 ls 命令，可以发现我们已经在当前目录下创建了 helloworld.c 文件)。

(2) 打开文件：在命令行界面输入 ‘vim helloworld.c‘打开新建的文件；

(3) 输入内容：刚打开文件时，我们无法向其中直接输入代码，需要按”i” 键进入插入模式，之后便可以向其中输入 helloworld 程序，如下图所示：

图 0.4: 写入 helloworld.c 中的内容

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

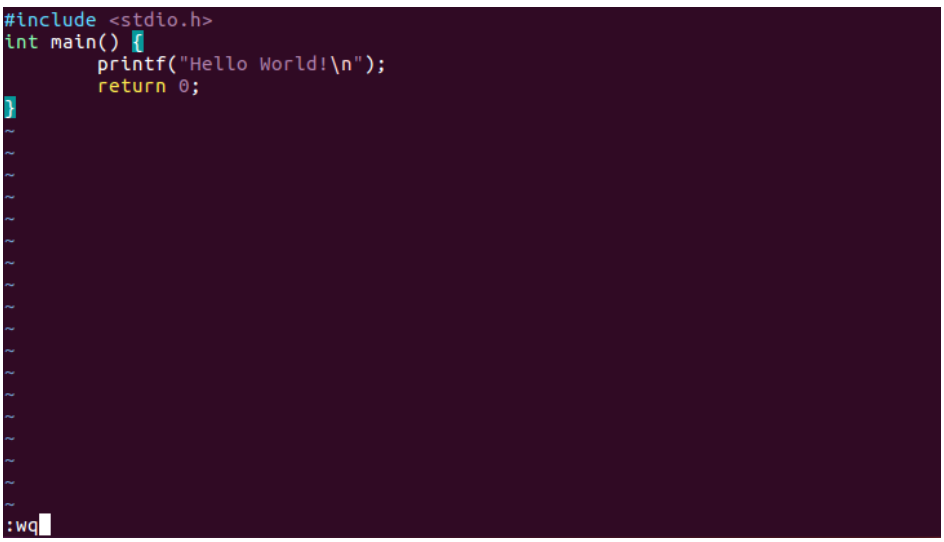


图 0.5: 保存并退出

接着，让我们简单了解一下对 vim 进行一些自定义配置的方法：

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

在这个文件中写入如下内容：

```
set tabstop=4
```

保存并退出文件后，我们便完成了对 vim 的配置（上述配置能够设置 Tab 键宽度为 4 个空格)。想要了解更多配置参数的同学可以自行在网上搜索。

在这里，推荐给大家一篇质量极高的 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，而如果你只想把 Vim 当成记事本用的话，几分钟的学习足矣。其他的内容网上有太多太多的资料，随用随查即可。

图 0.6: Vim 界面及基础介绍

0.4.2 GCC

在没有 IDE 的情况下，使用 GCC 编译器是一种简单快捷生成可执行文件的途径，只需一行命令即可将 C 源文件编译成可执行文件。其常用的使用方法如下图所示，同学们可以自己动手实践，写一些简单的 C 代码来编译运行。如果想要同时编译多个文件，可以直接用-o 选项将多个文件进行编译连接：gcc testfun.c test.c -o test ，也可以先使用-c 选项将每个文件单独编译成.o 文件，再用-o 选项将多个.o 文件进行连接：gcc -c testfun.c -> gcc -c test.c -> gcc testfun.o test.o -o test ，两者等价。

```
语法:gcc [选项]... [参数]...
选项（常用）:
-o                  指定生成的输出文件
-S                  将 C 代码转换为汇编代码
-Wall                  显示最多警告信息
-c                  仅执行编译操作，不进行链接操作
-M                  列出依赖
-include filename 编译时用来包含头文件，功能相当于在代码中使用 #include<filename>
-Ipath                  编译时指定头文件目录，使用标准库时不需要指定目录，-I 参数可以用相对路径，比如头文件在当前目录，可以用-I. 来指定
参数：
C 源文件：指定 C 语言源代码文件
e.g.

$ gcc test.c
# 默认生成名为 a.out 的可执行文件
#Windows 平台为 a.exe

$ gcc test.c -o test
# 使用-o 选项生成名为 test 的可执行文件
#Windows 平台为 test.exe
```

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

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

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

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

图 0.7: gcc 编译可执行文件并运行

0.4.3 Makefile

当我们想要翻开代码大干一番的时候，却面临着一个天大的问题：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。

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，如果我们没有 Makefile，直接动手编译这个程序，我们需要下面这样一个指令

```
# 直接使用 gcc 编译 Hello World 程序
$ 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:

```
target: dependencies
command 1
command 2
...
command n
```

其中，target 是我们构建 (Build) 的目标，可以是真的目标文件、可执行文件，也可以是一个标签。而 dependencies 是构建该目标所需的其它文件或其他目标。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。

我们通过在 makefile 中书写这些显式规则来告诉 make 工具文件间的依赖关系：如果想要构建 target，那么首先要准备好 dependencies，接着执行 command 中的命令，然后 target 就会送到我们手中，有点类似集齐龙珠向神龙许愿。在书写完恰当的规则之后，只需要在 shell 中输入 make target(目标名)，即可执行相应的命令、生成相应的目标。

还记得我们之前提到过 make 工具是根据时间戳来判断是否编译的吗？make 是个聪 (lan) 明 (duo) 的神龙，它不会浪费自己的神力实现相同的愿望，只有依赖文件中存在文件的修改时间比目标文件的修改时间晚时 (也就是对依赖文件做了改动)，shell 命令才被会执行，生成新的目标文件。

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

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

下面，我们尝试编写一个简单的 Makefile 文件：

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

```
all: helloworld.c
	gcc helloworld.c -o helloworld

clean:
	rm helloworld
```

图 0.8: 写入 Makefile 文件中的内容

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

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

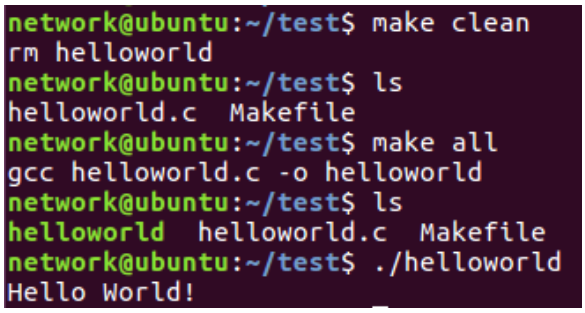


图 0.9: make 命令执行过程和效果

这里为同学们提供几个新手友好的网址为大家服务：

1. http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor
2. http://www.gnu.org/software/make/manual/make.html#Reading-Makefiles
3. https://seisman.github.io/how-to-write-makefile/introduction.html

第一个网址详细描述了如何从一个简单的 Makefile 入手，逐步扩充其功能，最终写出一个相对完善的 Makefile，同学们可以从这一网站入手，仿照其样例，写出一个自己的 Makefile。第二个网址提供了一份完备的 Makefile 语法说明，今后同学们在实验中遇到了未知的 Makefile 语法，均可在这一网站上找到满意的答案。第三个网站则是一份入门级中文教程，同学们可以通过这个网站了解和学习 Makefile 简单的基础知识，对 Makefile 各个重要部分有一个大致的了解。

0.4.4 ctags

ctags（Generate tag files for source code）是 vim 下方便代码阅读的工具。这里只进行一些最基础功能的介绍：（1）为了能够跨文件夹使用 ctags，我们需要添加 vim 的相关配置，按照上文所述的方法打开.vimrc 文件，添加以下内容并保存：

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

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

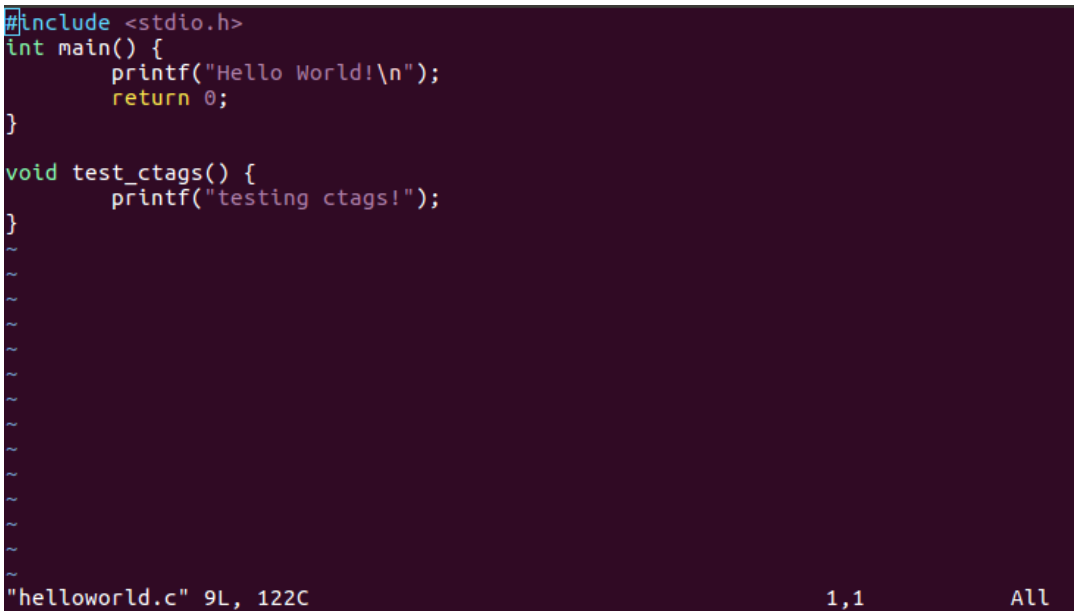


图 0.10: ctags_test.c 文件中的内容

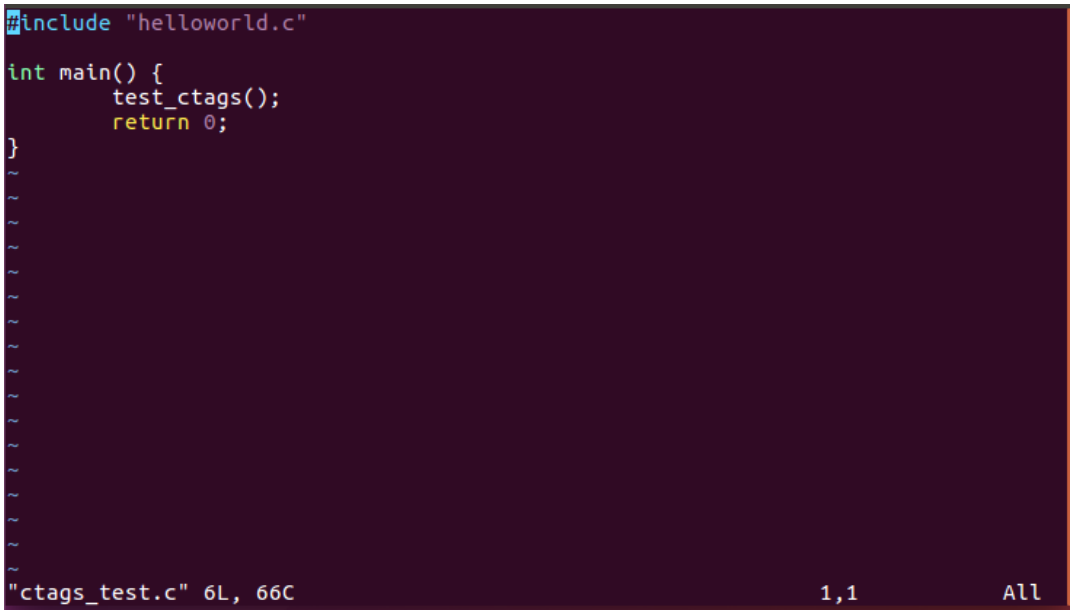


图 0.11: helloworld.c 文件中的内容

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

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

```
#include "helloworld.c"

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

图 0.12: 光标处于函数名上

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

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

0.5 Git 专栏–轻松维护和提交代码

经过上面的一系列的学习之后，你刚刚伸了个懒腰，也许准备喝杯茶休息休息。但突然你意识到了一个问题：之前好像说，有一个叫做 Git 的东西能对我们的学习进行评测？我们的实验是通过 git 版本控制系统进行管理，那么接下来，我们就来了解一下 git 相关的内容。这一部分讲述的内容相对来说比较细致与详尽，很多知识仅作了解。同学们可以着重阅读关于实验代码提交的部分，如果还有不清楚，可以观看我们教程网站的视频加以理解。

0.5.1 Git 是什么?

说到 Git 是什么，我们就得考虑什么是版本控制，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，哪个是老的了，即使知道新旧，可能也不知道每个版本是什么内容，相对上一版作了什么修改了，当几个版本过去后，很可能就是下面的样子了：

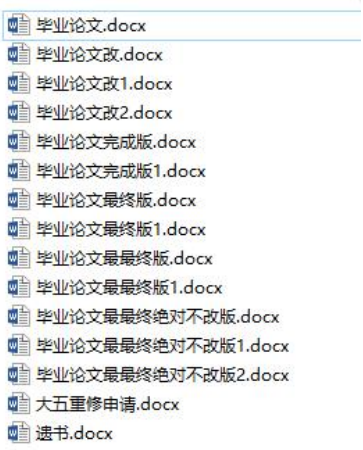


图 0.13: 手工版本控制

当然，有些时候，我们不仅是一个人写文本，那些工程项目也往往不是由一个人负

责。

分工，制定计划，埋头苦干，看起来一切井然有序，后来却只会让人蛋疼不已。本质原因在于每个人都会对项目的内容进行改动，结果最后有了这样一副情形：A 把添加完功能的项目打包发给了 B，然后自己继续添加功能。一天后，B 把他修改后的项目包又发给了 A，这时 A 就必须非常清楚发给 B 之后到他发回来的这段期间，自己究竟对哪里做了改动，然后还要进行合并，相当困难。

这时我们发现了一个无法避免的事实：如果每一次小小的改动，项目负责人之间都要相互通知，那么一些错误的改动将会令我们付出惨痛的代价：一个错误的改动要频繁在几方同时通知纠正。如果一次性改动了大幅度的内容，那么只有概览了项目的多数文件才能知道改动在哪，也才能合并劳动成果。项目只有 10 个文件时还好接受，如果是 40 个，60 个，80 个呢。

于是就产生了需求，我们希望有一款软件：

- 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。

那岂不是美滋滋~

版本控制系统就是这样一种神器的系统。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。

Git 是由 Linux 的缔造者 Linus Torvalds 创造，最开始也是用于管理自己的 Linux 开发过程。他对于 Git 的解释是：The stupid content tracker, 傻瓜内容追踪器。Git 一词本身也是个俚语，大概表示“混帐”。

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

0.5.2 Git 基础指引

看过上面的小案例后，相信好奇心旺盛的各位已经对 Git 充满了兴趣。那么 Git 在实际应用中究竟是怎样的一个系统呢？别着急，我们慢慢来。

先从 Git 最基础的指令讲起，通过前几节的学习，接下来请进行如下操作：

1. 创建一个名为 learnGit 的文件夹
2. 进入 learnGit 目录
3. 输入 git init
4. 用 ls 指令添加适当参数看看多了什么

我们会发现，新建的目录下面多了一个叫.git 的隐藏目录，这个目录就是我们的 Git 版本库，更多的被称为仓库 (repository)。需要注意的是，**在我们的实验中是不会对.git 文件夹进行任何直接操作的，所以不要轻易动这个文件夹中的任何文件**

在 init 执行完后，我们就拥有了一个仓库。我们建立的 learnGit 文件夹就是 Git 里的工作区。目前我们除了.git 版本库目录以外空无一物。

> **Note 0.5.2** 在我们的 MOS 操作系统实验中我们不需要使用到 git init 命令，每个人一开始就都有一个名为 20xxxxxx(你的学号) 的版本库，包含了 lab0 的实验内容。

既然工作区里空荡荡的，那我们来为他加点料：用你已知的方法在工作区中建立一个 readme.txt，内容为“BUAA_OSLAB”

当然这只是创建了一个文件而已，下面我们要将它添加至版本库，执行如下内容

```
$ 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:

```
$ git commit
```

如果不带任何附加选项，执行后会弹出一个说明窗口，如下所示:

```
GNU nano 2.6    文件： /home/13061193/13061193-lab/.git/COMMIT_EDITMSG

Notes to test.
# 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
# 说明将会终止提交。
# 位于分支 master
# 您的分支与上游分支 'origin/master' 一致。
#
# 要提交的变更：
#       修改：         readme.txt
#

[ 已读取 9 行 ]
^G 求助      ^O 写入      ^R 读档      ^Y 上页      ^K 剪切文字  ^C 游标位置
^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 0.5.3** 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。但以后你可能就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“fixedabug

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。

可能这样的窗口提交方式比较繁琐，我们可以采用一种简洁的方式：

```
$ 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

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

如果我们提交之后看到类似的提示就说明我们提交成功了

```
[master 955db52] Notes to test.
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思，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

```
*** Please tell me who you are.

Run

git config --global user.email "you@example.com"
git config --global user.name "Your Name"

# to set your account's default identity.
# 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 0.5.4 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，提交者信息是为了告知所有负责该项目的人每次提交是由谁提交的，并提供联系方式以进行交流

那么做一下这个小练习来快速上手 Git 的使用吧。

Thinking 0.1 思考下列有关 Git 的问题：

- 在/home/20xxxxxx/learnGit（已 init）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？(只有这个问题需要写到课后的实验报告中)

■

Note 0.5.5 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

```
Untracked.txt 的内容如下

# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       README.txt
nothing added to commit but untracked files present (use "git add" to track)

Stage.txt 的内容如下

# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README.txt
```

```
#

Modified.txt 的内容如下

# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   README.txt
#
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。

0.5.3 Git 文件状态

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 0.5.6** 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

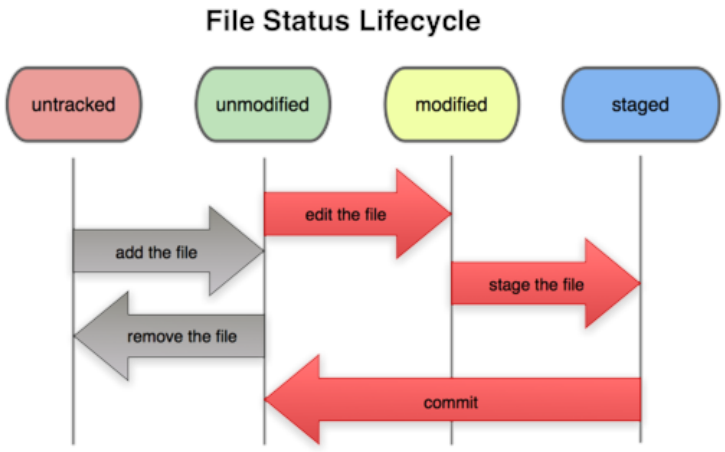


图 0.14: Git 中的四种状态转换关系

Thinking 0.2 仔细看看这张图，思考一下箭头中的 add the file 、stage the file 和 commit 分别对应的是 Git 里的哪些命令呢？ ■

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。

0.5.4 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，希望你能耐着性子仔细理解这张图和不同操作所带来的不同影响。

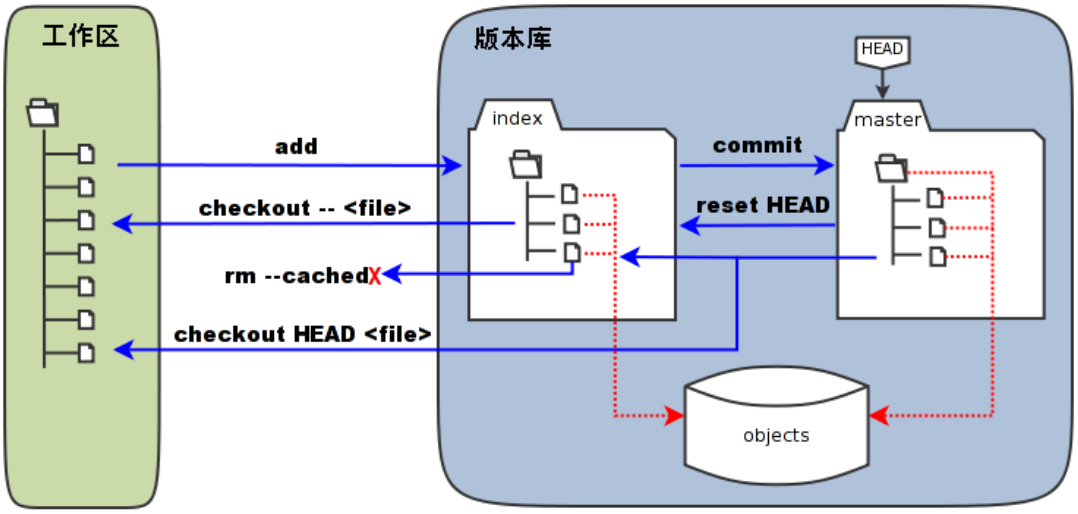


图 0.15: 工作区、暂存区和版本库

- 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。
- 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。

0.5.5 Git 时光机

图 0.16: 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 0.3 (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？
(2) 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？
(3) 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 **Tucao.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 Tucao.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

Thinking 0.4 思考下列有关 Git 的问题：

- 找到在/home/20xxxxxx/learnGit 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[a]。
- 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！
- 这一部分在课后的思考题中简单写一写你的理解即可，毕竟能够进行版本的恢复是使用 git 很重要的一个原因。

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

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！

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

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！

0.5.6 Git 分支

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

```
# 创建一个基于当前分支产生的分支，其名字为 <branch-name>
$ 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[1]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 0.17）。

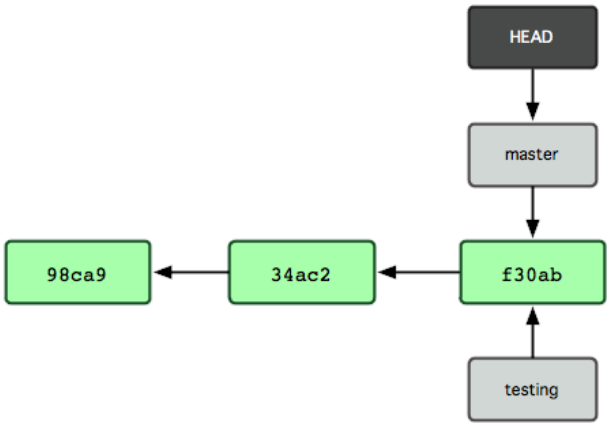


图 0.17: 分支建立后

[1]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

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

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

```
# 查看所有的远程与本地分支
$ git branch -a

# 使用该命令的效果如下
# 前面带 * 的分支是当前分支
lab1
lab1-exam
* lab1-result
master
remotes/origin/HEAD -> origin/master
remotes/origin/lab1
remotes/origin/lab1-exam
remotes/origin/lab1-result
remotes/origin/master
# 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

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

比如这时候我们使用 **git checkout testing**，这样 HEAD 就指向了 testing 分支 (见图0.18)。

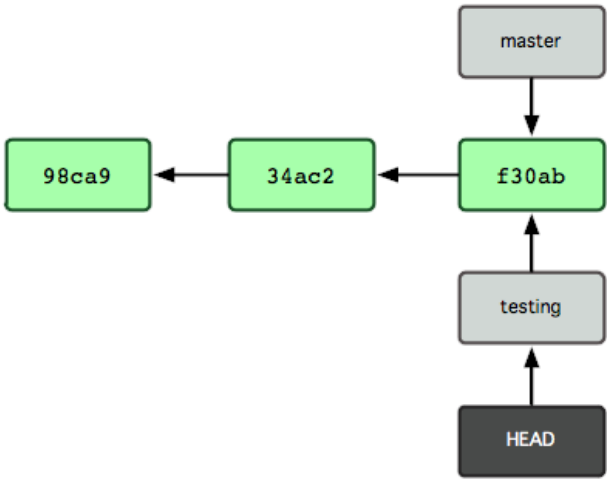


图 0.18: 分支切换后

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

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。

> **Note 0.5.7** 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。

0.5.7 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

```
# git clone 用于从远程仓库克隆一份到本地版本库
$ 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 0.5 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 0.5.8 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，克隆下来就着急忙慌地编译。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。

```
# git push 用于从本地版本库推送到服务器远程仓库
$ git push

# git pull 用于从服务器远程仓库抓取到本地版本库
$ 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

```
# 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。
$ 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。

0.5.8 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

```
中文版：
To git@github.com:20xxxxxx.git
! [rejected]        master -> master (non-fast-forward)
error: 无法推送一些引用到 'git@github.com:20xxxxxx.git'
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。
提示：再次推送前，先与远程变更合并（如 'git pull ...'）。详见
提示：'git push --help' 中的 'Note about fast-forwards' 小节。

英文版：
To git@github.com:20xxxxxx.git
! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'To git@github.com:20xxxxxx.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
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0.19）。

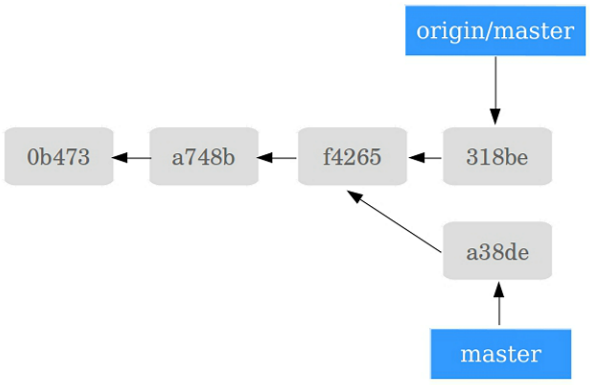


图 0.19: 远程仓库与本地仓库的岔路

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

```
Auto-merging test.txt
CONFLICT (content): Merge conflict in test.txt
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”

```
a123
<<<<<<< HEAD
b789
=======
b45678910
>>>>>>> 6853e5ff961e684d3a6c02d4d06183b5ff330dcc
c
```

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

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

```
error: Your local changes to the following files would be overwritten by merge:
20xxxxxx-lab/readme.txt
Please, commit your changes or stash them before you can merge.
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。

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！

0.5.9 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。

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。

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[2]。如果你希望能学到更厉害的技术，推荐GitHug，这是一个关于 Git 的通关小游戏。开启你快乐的实验之旅吧！^_^

0.6 进阶操作

0.6.1 Linux 操作补充

首先，是两种常用的查找命令：find 和 grep

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

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

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

```
grep - print lines matching a pattern
用法:grep [选项]... PATTERN [FILE]...
选项（常用）：
-a                不忽略二进制数据进行搜索
-i                忽略文件大小写差异
-r                从文件夹递归查找
-n    显示行号
```

tree 指令可以根据文件目录生成文件树，作用类似于 ls。

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

locate 也是查找文件的指令，与 find 的不同之处在于: find 是去硬盘找，locate 只在/var/lib/slocate 资料库中找。locate 的速度比 find 快，它并不是真的查找文件，而是查数据库，所以 locate 的查找并不是实时的，而是以数据库的更新为准，一般是系统自己维护，也可以手工升级数据库。

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

```
locate
用法：locate [选项] 文件名
```

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

```
chmod
用法：chmod 权限设定字串 文件...
权限设定字串格式 ：
[ugoa...][[+-=][rwxX]...][,...]
```

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

此外 chmod 也可以用数字来表示权限，格式为：

```
chmod abc 文件
```

abc 为三个数字，分别表示拥有者，群组，其他人的权限。r=4，w=2，x=1，用这些数字的加和来表示权限。例如 chmod 777 file 和 chmod a=rwx file 效果相同。

diff 命令用于比较文件的差异。

```
diff [选项] 文件 1 文件 2
常用选项
-b 不检查空格字符的不同
-B 不检查空行
-q 仅显示有无差异，不显示详细信息
```

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

```
sed
sed [选项] ‘命令’ 输入文本
选项(常用):
-n: 使用安静模式。在一般 sed 的用法中，输入文本的所有内容都会被输出。
加上 -n 参数后，则只有经过 sed 处理的内容才会被显示。
-e: 进行多项编辑，即对输入行应用多条 sed 命令时使用。
-i: 直接修改读取的档案内容，而不是输出到屏幕。使用时应小心。
命令(常用)：
a ：新增， a 后紧接着\\，在当前行的后面添加一行文本
c ：取代， c 后紧接着\\，用新的文本取代本行的文本
i ：插入， i 后紧接着\\，在当前行的上面插入一行文本
d ：删除，删除当前行的内容
p ：显示，把选择的内容输出。通常 p 会与参数 sed -n 一起使用。
s ：取代，格式为 s/re/string，re 表示正则表达式，string 为字符串，
功能为将正则表达式替换为字符串。
```

举例

```
sed -n '3p' my.txt
# 输出 my.txt 的第三行
sed '2d' my.txt
# 删除 my.txt 文件的第二行
sed '2,$d' my.txt
# 删除 my.txt 文件的第二行到最后一行
sed 's/str1/str2/g' my.txt
```

```
# 在整行范围内把 str1 替换为 str2。
# 如果没有 g 标记，则只有每行第一个匹配的 str1 被替换成 str2
sed -e '4a\str ' -e 's/str/aaa/' my.txt
#-e 选项允许在同一行里执行多条命令。例子的第一条是第四行后添加一个 str，
# 第二个命令是将 str 替换为 aaa。命令的执行顺序对结果有影响。
```

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

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

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

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

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

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

窗格操作

tmux 的窗格（pane）可以做出分屏的效果。

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

窗口操作

每个窗口（window）可以分割成多个窗格（pane）。

- ctrl+b c 创建之后会多出一个窗口

- ctrl+b p 切换到上一个窗口。
- ctrl+b n 切换到下一个窗口。
- ctrl+b 0 切换到 0 号窗口，依此类推，可换成任意窗口序号
- ctrl+b w 列出当前 session 所有串口，通过上、下键切换窗口
- ctrl+b & 关闭当前 window，会给出提示是否关闭当前窗口，按下 y 确认即可。

会话操作

一个会话 (session) 可以包含多个窗口 (window)

- tmux new -s 会话名新建会话
- ctrl+b d 退出会话，回到 shell 的终端环境
- tmux ls 终端环境查看会话列表
- tmux a -t 会话名从终端环境进入会话
- tmux kill-session -t 会话名销毁会话

0.6.2 shell 脚本

在以后的工作中，可能会遇到重复多次用到单条或多条长而复杂命令的情况，初学者可能会想把这些命令保存在一个文件中，以后再打开文件复制粘贴运行，其实大可不必复制粘贴，将文件按照批处理脚本运行即可。简单来说，批处理脚本就是存储了一条或多条命令的文本文件，Linux 系统中有一种简单快速执行批处理文件的方法（类似 Windows 系统中的.bat 批处理脚本）——source 命令。source 命令是 bash 的内置命令，该命令通常用点命令. 来替代。这两个命令都以一个脚本为参数，该脚本将作为当前 Shell 的环境执行，不会启动一个新的子进程，所有在脚本中设置的变量将成为当前 Shell 的一部分。使用方法如下图所示，其命令格式与之前介绍的命令类似，请同学们自己动手举例尝试。

```
source - execute commands in file
用法:source 文件名 [参数]
注：文件应为可执行文件，即为绿色
```

当有很多想要执行的 Linux 指令来完成复杂的工作，或者有一个或一组指令会经常执行时，我们可以通过 shell 脚本来完成。本节我们将学习使用 bash shell。首先用 touch 创建一个文件 my.sh，使用 vim 将其打开，并向其中写入以下内容（别忘了在 vim 里要输入要先按 i 哦）：

```
#!/bin/bash
#balabala
echo "Hello World!"
```

我们自己创建的 shell 脚本一般是不能直接运行的，需要添加运行权限：chmod +x my.sh 添加权限之后，我们可以使用 bash 命令来运行这个文件，把我们的文件作为它的参数：

```
bash my.sh
```

或者使用之前介绍的指令 source:

```
source my.sh
```

这样有些不方便，一种更方便的用法是:

```
./my.sh
```

在脚本中我们通常会加入 #!/bin/bash 到文件首行,以保证我们的脚本默认会使用 bash。第二行的内容是注释，以 # 开头。第三行是命令。

shell 传递参数与函数

我们可以向 shell 脚本传递参数。my2.sh 的内容

```
echo $1
```

执行命令

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

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

- $# 传递的参数个数
- $* 一个字符串显示传递的全部参数

shell 中的函数也用类似的方式传递参数。

```
function 函数名 ()
{
commands
[return int]
}
```

function 或者 () 可以省略其中一个。举例

```
fun(){
echo $1
echo $2
echo "the number of parameters is $#"
}
fun 1 str2
```

shell 流程控制 shell 脚本中也可以使用分支和循环语句。学有余力的同学可以学习一下。

if 的格式:

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

或者写到一行

```
if condition; then command1; command2; ⋯ fi
```

举例

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

条件部分可能会让同学们感到疑惑。中括号包含的条件表达式的-ne 是关系运算符，它们和 c 语言的比较运算符对应如下。

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

条件也可以写 true 或 false。变量除了使用自己定义的以外，还有一个比较常用的是 $?，代表上一个命令的返回值。比如刚执行完 diff 后，若两文件相同 $? 为 0。

实际上 condition 的位置上也是命令，当返回值为 0 时执行。左中括号是指令，$a、-ne、1、] 是指令的选项，关系成立返回 0。true 则是直接返回 0。condition 也可以用 diff file1 file2 来填。

while 语句格式如下

```
while condition
do
commands
done
```

while 语句可以使用 continue 和 break 这两个循环控制语句。

例如创建 10 个目录，名字是 file1 到 file9。

```
a=1
while [ $a -ne 10 ]
do
mkdir file$a
a=$[$a+1]
done
```

有两点请注意：流程控制的内容不可为空。运算符和变量之间要有空格。

除了以上内容，shell 还有 for，case，else 语句，逻辑运算符等语法，内容有兴趣的同学可以自行了解。

Thinking 0.6 执行如下命令, 并查看结果

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

■

Thinking 0.7 使用你知道的方法（包括重定向）创建下图内容的文件（文件命名为 test），将创建该文件的命令序列保存在 command 文件中，并将 test 文件作为批处理文件运行，将运行结果输出至 result 文件中。给出 command 文件和 result 文件的内容，并对最后的结果进行解释说明（可以从 test 文件的内容入手）. 具体实现的过程中思考下列问题: echo echo Shell Start 与 echo 'echo Shell Start' 效果是否有区别; echo echo $c>file1 与 echo 'echo $c>file1' 效果是否有区别. ■

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

图 0.20: 文件内容

0.6.3 重定向和管道

这部分我们将学习如何实现 Linux 命令的输入输出怎样定向到文件，以及如何将多个指令组合实现更强大的功能。shell 使用三种流:

- 标准输入: stdin ，由 0 表示
- 标准输出: stdout，由 1 表示

- 标准错误：stderr，由 2 表示

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

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

管道:

管道符号“|”可以连接命令:

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

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

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

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

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

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

0.6.4 gxemul 的使用

gxemul 是我们运行 MOS 操作系统的仿真器，它可以帮助我们运行和调试 MOS 操作系统。直接输入 gxemul 会显示帮助信息。

gxemul 运行选项:

- -E 仿真机器的类型
- -C 仿真 cpu 的类型
- -M 仿真的内存大小
- -V 进入调试模式

举例:

```
gxemul -E testmips -C R3000 -M 64 vmlinux    # 用 gxemul 运行 vmlinux
gxemul -E testmips -C R3000 -M 64 -V vmlinux
# 以调试模式打开 gxemul，对 vmlinux 进行调试（进入后直接中断，
# 输入 continue 或 step 才会继续运行，在此之前可以进行添加断点等操作）
```

进入 gxemul 后使用 Ctrl-C 可以中断运行。中断后可以进行单步调试，执行如下指令:

- breakpoint add addr 添加断点
- continue 继续执行
- step [n] 向后执行 n 条汇编指令
- lookup name|addr 通过名字或地址查找标识符
- dump [addr [endaddr]] 查询指定地址的内容
- reg [cpuid][,c] 查看寄存器内容，添加”,c”可以查看协处理器
- help 显示各个指令的作用与用法
- quit 退出

(以上中括号表示内容可以没有)

更多 gxemul 相关的信息参考http://gavare.se/gxemul/gxemul-stable/doc/index.html

0.6.5 x86 汇编语言

由于我们实验使用的是 Linux X86_64 平台，所以我们不免的要与 x86 汇编语言进行接触，对其进行深入的了解可以帮助我们更好的阅读实验环境中的代码和输出。在这里给出一个英文文档，对 x86 汇编语言进行了一个大致的介绍，希望大家能够进行阅读。

http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

0.7 实战测试

随着 lab0 学习的结束，下面就要开始通过实战检验水平了，请同学们按照下面的题目要求完成所需操作，最后将工作区 push 至远端以进行评测，评测完成后会返回 lab0 课下测试的成绩，通过课下测试 (>=60 分) 即可参加上机时的课上测试。

Exercise 0.1 lab0 第一道练习题包括以下四题，如果你四道题全部完成且正确，即可获得 50 分。

1、在 lab0 工作区的 src 目录中，存在一个名为 palindrome.c 的文件，使用刚刚学过的工具打开 palindrome.c，使用 c 语言实现判断输入整数 n($1 \leq n \leq 10000$) 是否为回文数的程序 (输入输出部分已经完成)。通过 stdin 每次只输入一个整数 n，若这个数字为回文数则输出 Y，否则输出 N。[注意：正读倒读相同的整数叫回文数]

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

3、在 src/sh_test 目录下，有一个 file 文件和 hello_os.sh 文件。hello_os.sh 是一个未完成的脚本文档，请同学们借助 shell 编程的知识，将其补完，以实现通

过指令 bash hello_os.sh AAA BBB.c，在 hello_os.sh 所处的文件夹新建一个名为 BBB.c 的文件，其内容为 AAA 文件的第 8、32、128、512、1024 行的内容提取 (AAA 文件行数一定超过 1024 行)。[注意: 对于指令 bash hello_os.sh AAA BBB.c，AAA 及 BBB 可为任何合法文件的名称，例如 bash hello_os.sh file hello_os.c，若以有 hello_os.c 文件，则将其原有内容覆盖]

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

要求按照要求完成后，最终提交的文件树图示如图0.21 ■

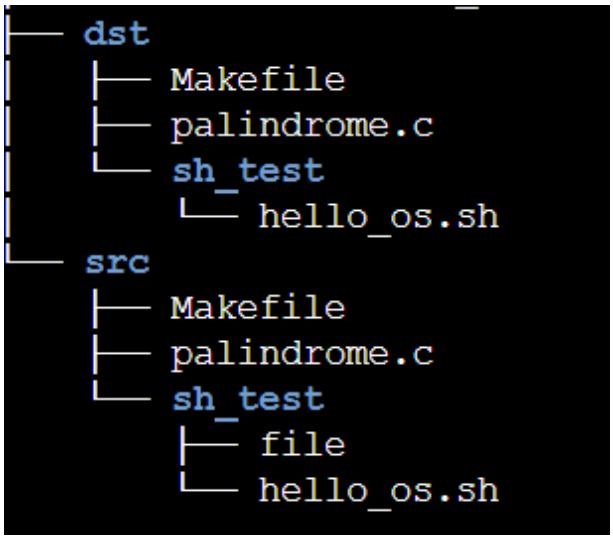


图 0.21: 第一题最终提交的文件树

Exercise 0.2 lab0 第二道练习题包括以下一题，如果你完成且正确，即可获得 12 分。

1、在 lab0 工作区 ray/sh_test1 目录中，含有 100 个子文件夹 file1~file100，还存在一个名为 changefile.sh 的文件，将其补完，以实现通过指令 bash changefile.sh，可以删除该文件夹内 file71~file100 共计 30 个子文件夹，将 file41~file70 共计 30 个子文件夹重命名为 newfile41~newfile70。[注意: 评测时仅检测 changefile.sh 的正确性]

要求按照要求完成后，最终提交的文件树图示如图0.22(file 下标只显示 1~12，newfile 下标只显示 41~55) ■

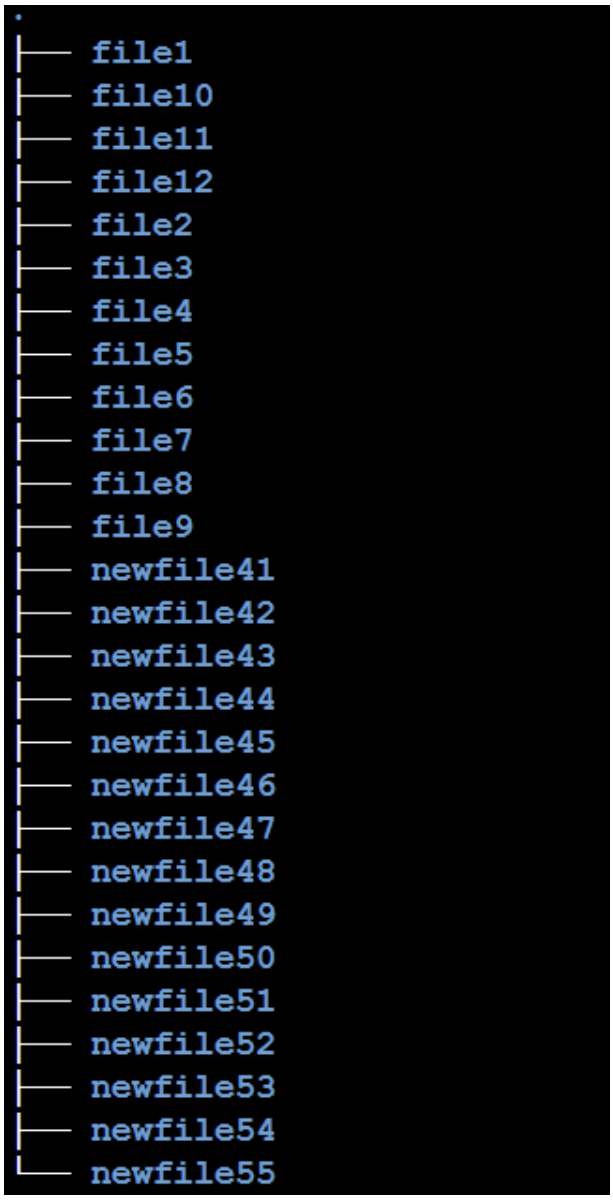


图 0.22: 第二题最终提交的文件树

Exercise 0.3 lab0 第三道练习题包括以下一题，如果你完成且正确，即可获得 12 分。

1、在 lab0 工作区的 ray/sh_test2 目录下，存在一个未补全的 search.sh 文件，将其补完，以实现通过指令 bash search.sh file int result，可以在当前文件夹下生成 result 文件，内容为 file 文件含有 int 字符串所在的行数，即若有多行含有 int 字符串需要全部输出。[注意：对于指令 bash search.sh file int result，file 及 result 可为任何合法文件名称，int 可为任何合法字符串，若已有 result 文件，则将其原有内容覆盖，匹配时大小写不忽略]

要求按照要求完成后，result 内显示样式如图0.23(一个答案占一行)：

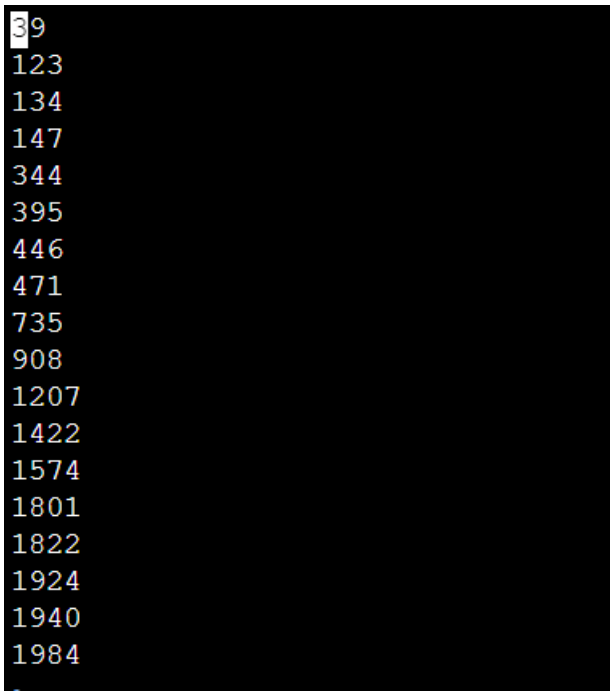


图 0.23: 第三题完成后结果

Exercise 0.4 lab0 第四道练习题包括以下两题，如果你均完成且正确，即可获得 26 分。

1、在 lab0 工作区的 csc/code 目录下，存在 fibo.c、main.c，其中 fibo.c 有点小问题，还有一个未补全的 modify.sh 文件，将其补完，以实现通过指令 bash modify.sh fibo.c char int，可以将 fibo.c 中所有的 char 字符串更改为 int 字符串。[注意：对于指令 bash modify.sh fibo.c char int，fibo.c 可为任何合法文件名，char 及 int 可以是任何字符串，评测时评测 modify.sh 的正确性，而不是检查修改后 fibo.c 的正确性]

2、lab0 工作区的 csc/code/fibo.c 成功更换字段后 (bash modify.sh fibo.c char int)，现已有 csc/Makefile 和 csc/code/Makefile，补全两个 Makefile 文件，要求在 csc 目录下通过指令 make 可在 csc/code 文件夹中生成 fibo.o、main.o，在 csc 文件夹中生成可执行文件 fibo，再输入指令 make clean 后只删除两个.o 文件。[注意：不能修改 fibo.h 和 main.c 文件中的内容，提交的文件中 fibo.c 必须是修改后正确的 fibo.c，可执行文件 fibo 作用是输入一个整数 n(从 stdin 输入 n)，可以输出斐波那契数列前 n 项，每一项之间用空格分开。比如 n=5，输出 1 1 2 3 5]

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

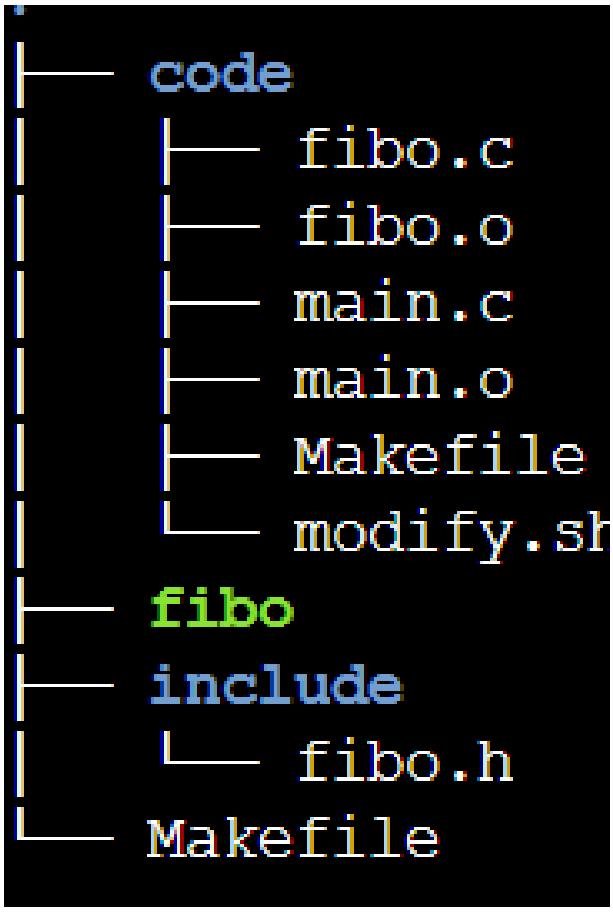


图 0.24: 第四题 make 后文件树

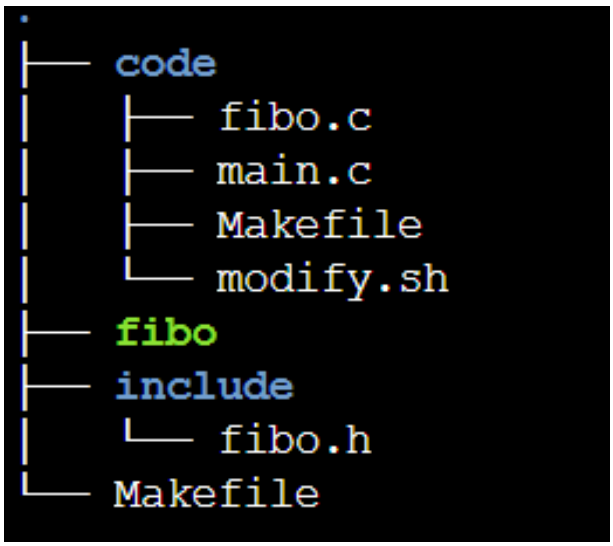


图 0.25: 第四题 make clean 后文件树

图 0.26: 满分提交评测反馈

0.8 实验思考

- 思考-箭头与指令
- 思考-小明的困境
- 思考-克隆命令
- 思考-文件的操作
- 思考-Git 的使用 1
- 思考-Git 的使用 2
- 思考-echo 的使用

实验思考里的内容需要附在实验文档中一起提交哦!

CHAPTER 1

内核、BOOT 和 PRINTF

1.1 实验目的

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

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

1.2 操作系统的启动

1.2.1 内核在哪里?

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。那么在操作系统设计课程中，我们管理的硬件在哪里呢？如果足够细心的话，在前面的内容学习可以知道 GXemul 是一款计算机架构仿真器，在本实验可以模拟我们需要的 CPU 等硬件环境。在编写操作系统之前，我们每个人面前都是一个裸机。对于操作系统课程设计，我们编写代码的平台是 Linux 系统，进行仿真实验的平台是 GXemul。我们编写的所有的代码，在提供的 Linux 平台通过 Makefile 交叉编译产生可执行文件，最后使用 GXemul 对可执行文件运行填充完成的操作系统。

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管理各个硬件，从而利用硬件的功能为用户进程提供服务。启动操作系统，我们就需要将内核代码在计算机结构上运行起来，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。所以直观上可以认识到： 磁盘不能直接访问 内存掉电易失，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服务的工作，从而降低了硬件相关代码和软件相关代码的耦合

度，有助于操作系统的移植。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，bootloader 的总目标就是正确地调用内核来执行。另外，由于 bootloader 的实现依赖于 CPU 的体系结构，因此大多数 bootloader 都分为 stage1 和 stage2 两大部分。

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 bootloader 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。

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，就是可以可以用于在有多个操作系统的机器上，在刚开机的时候选择一个操作系统进行引导。如果安装过 Ubuntu 一类的发行版的话，一开机出现的那个选择系统用的菜单就是 GRUB 提供的。

（这里以 GRUB 为例）BIOS 加载 MBR 中的 GRUB 代码后就把 CPU 交给了 GRUB，GRUB 的工作就是一步一步的加载自身代码，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 的工作则是完全在 GRUB 中完成的。

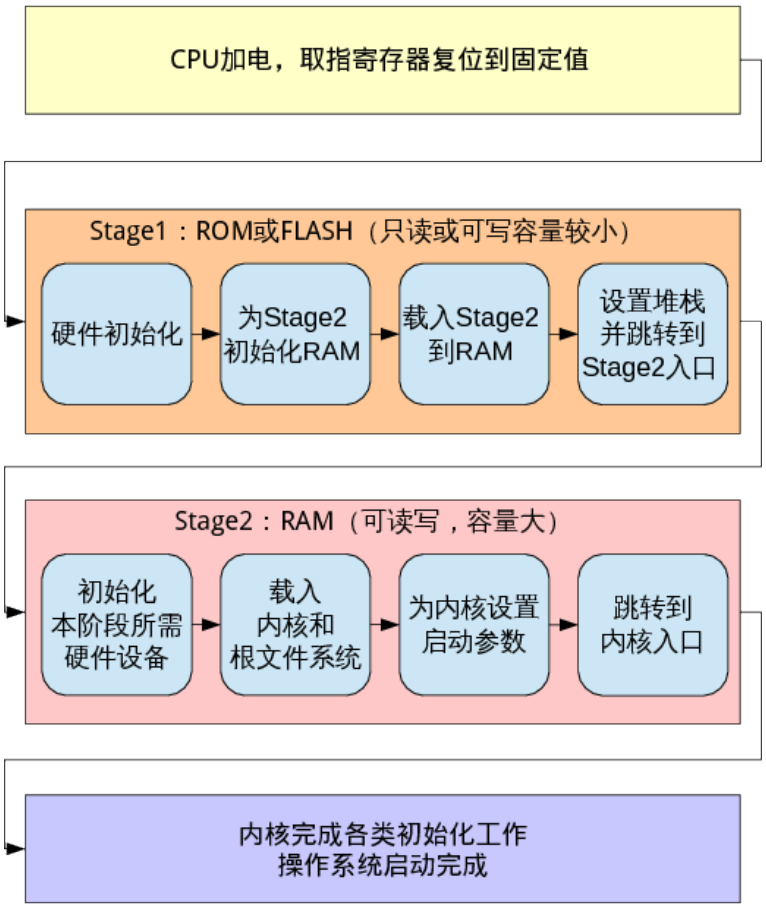

CPU加电，取指寄存器复位到固定值
Stage1：ROM或FLASH（只读或可写容量较小）
硬件初始化
为Stage2
初始化RAM
载入Stage2
到RAM
设置堆栈
并跳转到
Stage2入口
Stage2：RAM（可读写，容量大）
初始化
本阶段所需
硬件设备
载入
内核和
根文件系统
为内核设置
启动参数
跳转到
内核入口
内核完成各类初始化工作
操作系统启动完成

图 1.1: 启动的基本步骤

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 MOS 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 MOS 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 MOS 操作系统的运行第一行代码前，我们就已经拥有一个正常的 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 MOS 操作系统内核了。这一节中，我们将介绍如何修改内核并实现一些自定义的功能。

1.3.1 Makefile——内核代码的地图

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

不过我们看见有一个文件非常熟悉，叫做 Makefile。

相信大家在 lab0 中，已经对 Makefile 有了初步的了解。下面这个 Makefile 文件即为构建我们整个操作系统所用的顶层 Makefile。那么，我们就可以通过浏览这个文件来对整个操作系统的布局产生初步的了解：可以说，Makefile 就像源代码的地图，告诉我们源代码是如何一步一步成为最终的可执行文件的。代码1是实验代码最顶层的 Makefile，通过阅读它我们就能了解代码中很多宏观的东西。(为了方便理解，加入了部分注释)

Listing 1: 顶层 Makefile

```
# Main makefile
#
# Copyright (C) 2007 Beihang University
# Written by Zhu Like ( zlike@cse.buaa.edu.cn )
#

drivers_dir          := drivers
boot_dir          := boot
init_dir          := init
lib_dir                  := lib
tools_dir          := tools
test_dir          :=
# 上面定义了各种文件夹名称
vmlinux_elf          := gxemul/vmlinux
# 这个是我们最终需要生成的 elf 文件
link_script   := $(tools_dir)/scse0_3.lds # 链接用的脚本

modules                  := boot drivers init lib
objects                  := $(boot_dir)/start.o                          \
                                $(init_dir)/main.o                          \
                                $(init_dir)/init.o                          \
                                   $(drivers_dir)/gxconsole/console.o \
                                $(lib_dir)/*.o                                  \
# 定义了需要生成的各种文件

.PHONY: all $(modules) clean

all: $(modules) vmlinux # 我们的“最终目标”

vmlinux: $(modules)
        $(LD) -o $(vmlinux_elf) -N -T $(link_script) $(objects)

## 注意，这里调用了一个叫做 $(LD) 的程序

$(modules):
        $(MAKE) --directory=$@

# 进入各个子文件夹进行 make

clean:
        for d in $(modules);        \
                do                                              \
                        $(MAKE) --directory=$$d clean; \
                done; \
        rm -rf *.o *~ $(vmlinux_elf)

```

```
47 include include.mk
```

如果你以前没有接触过 Makefile 的话，仅仅凭借 lab0 的简单练习获得的知识，可能还不足以顺畅的阅读这份 47 行的 Makefile。不必着急，我们来一行一行地解读它。前 6 行是注释，不必介绍。接下来很开心的看到了我们熟悉的赋值符号。没错，这是 Makefile 中对变量的定义语句。7～23 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)。其中，最值得注意的两个变量分别是 modules 和 objects。modules 定义了内核所包含的所有模块，objects 则表示要编译出内核所依赖的所有.o 文件。19 到 23 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 1.3.1** linker script 是用于指导链接器将多个.o 文件链接成目标可执行文件的脚本。.o 文件、linker script 等内容我们会在下面的小节中细致地讲解，大家这里只要知道这些文件是编译内核所必要的就好。

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

第 28 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 30 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 31 行的指令来产生 vmlinux。我们可以看到，第 31 行调用了链接器将之前构建各模块产生的所有.o 文件在 linker script 的指导下链接到一起，产生最终的 vmlinux 可执行文件。第 35 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 40 到 45 行定义了如何清理所有被构建出来的文件。

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

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

紧接着我们看到了第 47 行有一条 include 命令。看来，这个顶层的 Makefile 还引用了其他的东西。让我们来看看这个文件，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 := 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
```

原来我们的 LD 变量是在这里被定义的 (变量 MAKE 是 Makefile 中的自带变量, 不是我们定义的)。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)，也就是变量 CC。不难看出，这里的 CROSS_COMPILE 变量是在定义编译和链接等指令的前缀，或者说是交叉编译器的具体位置。例如，在我们的实验以及测评环境中，LD 最终调用的指令是“/OSLAB/compiler/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。 ■

Note 1.3.3 可以自己尝试去目录/OSLAB/compiler/usr/bin/看一眼，确信交叉编译路径不是我们随口说的，而是确确实实躺在那里的哦。

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。关于 Makefile 还有许多有趣的知识，同学们可以自行了解学习。

最后，简要总结一下实验代码中其他目录的组织以及其中的重要文件:

- tool 目录下只有 scse0_3.lds 这个 linker script 文件，我们会在下面的小节中详细讲解。
- gxemul 目录存放着 GXemul 仿真器启动内核文件时所需要的配置文件，另外代码编译完成后生成的名为“vmlinux”的内核文件也会放在这个目录下。
- boot 目录中需要注意的是 start.S 文件。这个文件中的 _start 函数是 CPU 控制权被转交给内核后执行的第一个函数，主要工作是初始化硬件相关的设置，为之后操作系统初始化做准备，最后跳转到 init/main.c 文件中定义的 main 函数。
- init 目录中主要有两个代码文件 main.c 和 init.c，其作用是初始化操作系统。通过分析函数调用关系，我们可以发现最终调用的函数是 init.c 文件中的 mips_init() 函数。在本实验中此函数只是简单的打印输出，而在之后的实验中会逐步添加新的内核功能，所以诸如内存初始化等具体方面的初始化函数都会在这里被调用。
- include 目录中存放系统头文件。在本实验中需要用到的头文件是 mmu.h 文件，这个文件中有一张内存布局图，我们在填写 linker script 的时候需要根据这个图来设置相应节的加载地址。

- lib 目录存放一些常用库函数，这里主要存放一些打印的函数。
- driver 目录中存放驱动相关的代码，这里主要存放的是终端输出相关的函数。

1.3.2 ELF——深入探究编译与链接

如果你已经尝试过运行内核，那么你会发现它现在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。请注意我们在本章中阐述的内容一部分关于 Linux 实验环境，即在 Linux 环境下如何模拟我们编写的操作系统，另一部分则是关于我们将要编写的操作系统，在看琐碎的知识点时，请务必注意我们在讨论哪一部分，本章很多概念的混淆或模糊都是由于没有区分开这两部分内容。请务必坚持看完:)

Listing 3: 一个简单的 C 程序

```c
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

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？[1] 我们都学过，C 语言中函数必须有定义才能被调用，那么 printf 的定义在哪里呢？你一定会笑一笑说，别傻了，不就在 stdio.h 中吗？我们在程序开头通过 include 引用了它的。然而事实真的是这样吗？我们来进去看一看 stdio.h 里到底有些什么。

Listing 4: stdio.h 中关于 printf 的内容

```c
/*
 *        ISO C99 Standard: 7.19 Input/output        <stdio.h>
 */

/* Write formatted output to stdout.
```

[1]printf 位于标准库中，而不在我们的 C 代码中。将标准库和我们自己编写的 C 文件编译成一个可执行文件的过程，与将多个 C 文件编译成一个可执行文件的过程相仿。因此，我们通过探究 printf 如何和我们的 C 文件编译到一起，来展示整个过程。

```
   This function is a possible cancellation point and therefore not
   marked with __THROW.  */
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。Linux 命令如下：

```
gcc -E 源代码文件名
```

你可以使用重定向将上述命令输出重定向至文件，以便于观察输出情况。

```
/* 由于原输出太长，这里只能留下很少很少的一部分。 */
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;


typedef signed char __int8_t;
typedef unsigned char __uint8_t;
typedef signed short int __int16_t;
typedef unsigned short int __uint16_t;
typedef signed int __int32_t;
typedef unsigned int __uint32_t;

typedef signed long int __int64_t;
typedef unsigned long int __uint64_t;

extern struct _IO_FILE *stdin;
extern struct _IO_FILE *stdout;
extern struct _IO_FILE *stderr;

extern int printf (const char *__restrict __format, ...);

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

可以看到，C 语言的预处理器将头文件的内容添加到了源文件中，但同时我们也能看到，这里一阶段并没有 printf 这一函数的定义。

之后，我们将 gcc 的-E 选项换为-c 选项，**只编译而不链接**，产生一个同名的.o 目标文件。命令如下：

```
gcc -c 源代码文件名
```

我们对其进行反汇编[2]，反汇编并将结果导出至文本文件的命令如下：

[2]为了便于你重现，我们这里没有选择 MIPS，而选择了在流行的 x86-64 体系结构上进行反汇编。同时，由于 x86-64 的汇编是 CISC 汇编，看起来会更为清晰一些。

```
objdump -DS 要反汇编的目标文件名 > 导出文本文件名
```

main 函数部分的结果如下

```
hello.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   bf 00 00 00 00          mov    $0x0,%edi
   9:   e8 00 00 00 00          callq  e <main+0xe>
   e:   b8 00 00 00 00          mov    $0x0,%eax
  13:   5d                      pop    %rbp
  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 MIPS 指令的特点，e8 后面应该跟的是 printf 的地址。可在这里我们却发现，**本该填写 printf 地址的位置上被填写了一串 0**。那个地址显然不可能是 printf 的地址。也就是说，直到这一步，printf 的具体实现依然不在我们的程序中。

最后，我们允许 gcc 进行链接，也就是**正常地编译**出可执行文件。然后，再用 objdump 进行反汇编。命令如下，其中-o 选项用于指定输出的目标文件名，如果不设置的话默认为 a.out

```
gcc [-o 输出可执行文件名] 源代码文件名
objdump -DS 输出可执行文件名 > 导出文本文件名
```

反汇编结果如下

```
hello:     file format elf64-x86-64


Disassembly of section .init:

00000000004003a8 <_init>:
  4003a8:       48 83 ec 08             sub    $0x8,%rsp
  4003ac:       48 8b 05 0d 05 20 00    mov    0x20050d(%rip),%rax
  4003b3:       48 85 c0                test   %rax,%rax
  4003b6:       74 05                   je     4003bd <_init+0x15>
  4003b8:       e8 43 00 00 00          callq  400400 <__gmon_start__@plt>
  4003bd:       48 83 c4 08             add    $0x8,%rsp
  4003c1:       c3                      retq

Disassembly of section .plt:

00000000004003d0 <puts@plt-0x10>:
  4003d0:       ff 35 fa 04 20 00       pushq  0x2004fa(%rip)
  4003d6:       ff 25 fc 04 20 00       jmpq   *0x2004fc(%rip)
  4003dc:       0f 1f 40 00             nopl   0x0(%rax)

00000000004003e0 <puts@plt>:
  4003e0:       ff 25 fa 04 20 00       jmpq   *0x2004fa(%rip)
  4003e6:       68 00 00 00 00          pushq  $0x0
```

```
  4003eb:       e9 e0 ff ff ff          jmpq   4003d0 <_init+0x28>

00000000004003f0 <__libc_start_main@plt>:
  4003f0:       ff 25 f2 04 20 00       jmpq   *0x2004f2(%rip)
  4003f6:       68 01 00 00 00          pushq  $0x1
  4003fb:       e9 d0 ff ff ff          jmpq   4003d0 <_init+0x28>

0000000000400400 <__gmon_start__@plt>:
  400400:       ff 25 ea 04 20 00       jmpq   *0x2004ea(%rip)
  400406:       68 02 00 00 00          pushq  $0x2
  40040b:       e9 c0 ff ff ff          jmpq   4003d0 <_init+0x28>

Disassembly of section .text:

0000000000400410 <main>:
  400410:       48 83 ec 08             sub    $0x8,%rsp
  400414:       bf a4 05 40 00          mov    $0x4005a4,%edi
  400419:       e8 c2 ff ff ff          callq  4003e0 <puts@plt>
  40041e:       31 c0                   xor    %eax,%eax
  400420:       48 83 c4 08             add    $0x8,%rsp
  400424:       c3                      retq

0000000000400425 <_start>:
  400425:       31 ed                   xor    %ebp,%ebp
  400427:       49 89 d1                mov    %rdx,%r9
  40042a:       5e                      pop    %rsi
  40042b:       48 89 e2                mov    %rsp,%rdx
  40042e:       48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
  400432:       50                      push   %rax
  400433:       54                      push   %rsp
  400434:       49 c7 c0 90 05 40 00    mov    $0x400590,%r8
  40043b:       48 c7 c1 20 05 40 00    mov    $0x400520,%rcx
  400442:       48 c7 c7 10 04 40 00    mov    $0x400410,%rdi
  400449:       e8 a2 ff ff ff          callq  4003f0 <__libc_start_main@plt>
  40044e:       f4                      hlt
  40044f:       90                      nop
```

篇幅所限，余下的部分没法再展示了（大约还有 100 来行）。

当你看到熟悉的“hello world”被展开成如此“臃肿”的代码，可能不忍直视。但是别急，我们还是只把注意力放在主函数中，这一次，我们可以惊喜的看到，主函数里那一句 callq 后面已经不再是一串 0 了。那里已经**被填入了一个地址**。从反汇编代码中我们也可以看到，这个地址就在这个可执行文件里，就在被标记为 puts@plt 的那个位置上。虽然搞不清楚那个东西是什么，但显然那就是我们所调用的 printf 的具体实现了。

由此，我们不难推断，printf 的实现是在**链接 (Link)** 这一步骤中被插入到最终的可执行文件中的。那么，了解这个细节究竟有什么用呢？作为一个库函数，printf 被大量的程序所使用。因此，每次都将其编译一遍实在太浪费时间了。printf 的实现其实早就被编译成了二进制形式。但此时，printf 并未链接到程序中，它的状态与我们利用-c 选项产生的 hello.o 相仿，都还处于未链接的状态。而在编译的最后，链接器 (Linker) 会将所有的目标文件链接在一起，将之前未填写的地址等信息填上，形成最终的可执行文件，这就是链接的过程。

对于拥有多个 c 文件的工程来说，编译器会首先将所有的 c 文件以文件为单位，编译成.o 文件。最后再将所有的.o 文件以及函数库链接在一起，形成最终的可执行文件。

整个过程如图1.2所示。

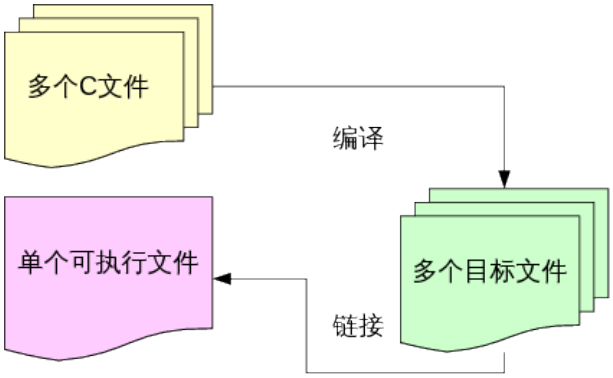


图 1.2: 编译、链接的过程

Thinking 1.1 请查阅并给出前述 objdump 中使用的参数的含义。使用其它体系结构的编译器（如课程平台的 MIPS 交叉编译器）重复上述各步编译过程，观察并在实验报告中提交相应的结果。 ■

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

为了帮助你了解 ELF 文件，下一步我们需要进一步探究它的功能以及格式。

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

你可以使用 file 命令来获得文件的类型，如图1.3所示。

```
15061119@ubuntu:~$ file a.o
a.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
15061119@ubuntu:~$ file a.out
a.out: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically linked, BuildID[sha1]=0xf6dc027bfd1e8cd03b60b65ba2e7856614d0d4b2, not stripped
15061119@ubuntu:~$ file a.so
a.so: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically linked, BuildID[sha1]=0xcbda03781956e54814a01c989c0ef44fb77b2c69, not stripped
```

图 1.3: file 命令

那么，ELF 文件中都包含有什么东西呢？简而言之，就是和程序相关的所有必要信息，下图1.4说明了 ELF 文件的结构：

通过上图我们可以知道，ELF 文件从整体来说包含 5 个部分：

1. ELF Header，包括程序的基本信息，比如体系结构和操作系统，同时也包含了 Section Header Table 和 Program Header Table 相对文件的偏移量 (offset)。

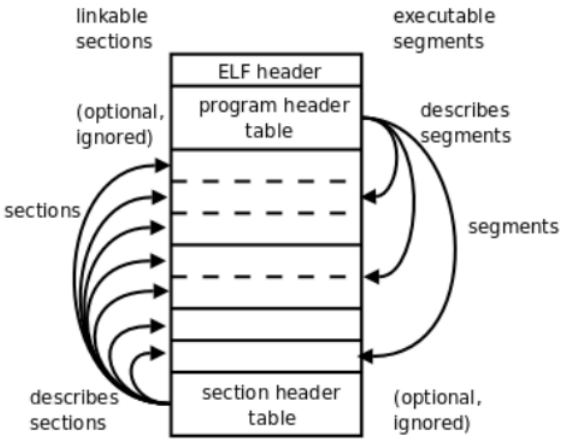


图 1.4: ELF 文件结构

2. Program Header Table，也可以称为 Segment Table，主要包含程序中各个 Segment 的信息，Segment 的信息需要在运行时刻使用。

3. Section Header Table，主要包含各个 Section 的信息，Section 的信息需要在程序编译和链接的时候使用。

4. Segments，就是各个 Segment。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。

5. Sections，就是各个 Section。Section 记录了程序的代码段、数据段等各个段的内容，主要是链接器在链接的过程中需要使用。

观察上图1.4我们可以发现，Program Header Table 和 Section Header Table 指向了同样的地方，这就说明两者所代表的内容是重合的，这意味着什么呢？意味着两者只是同一个东西的不同视图！产生这种情况的原因在于 ELF 文件需要在两种场合使用：

1. 组成可重定位文件，参与可执行文件和可共享文件的链接。

2. 组成可执行文件或者可共享文件，在运行时为加载器提供信息。

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

为了帮助大家进行理解，我们先对这个程序中涉及的三个关键数据结构做一下简要说明，请大家打开./readelf/kerelf.h 这个文件，配合下方给出的注释和说明，仔细阅读其中的代码和注释，然后完成练习。下面的代码都截取自./readelf/kerelf.h 文件

```
1 /* 文件的前面是各种变量类型定义，在此省略 */
2 /* The ELF file header.  This appears at the start of every ELF file.  */
3 /* ELF 文件的文件头。所有的 ELF 文件均以此为起始 */
4 #define EI_NIDENT (16)
5
```

```
typedef struct {
    unsigned char   e_ident[EI_NIDENT];     /* Magic number and other info */
    // 存放魔数以及其他信息
    Elf32_Half      e_type;                 /* Object file type */
    // 文件类型
    Elf32_Half      e_machine;              /* Architecture */
    // 机器架构
    Elf32_Word      e_version;              /* Object file version */
    // 文件版本
    Elf32_Addr      e_entry;                /* Entry point virtual address */
    // 入口点的虚拟地址
    Elf32_Off       e_phoff;                /* Program header table file offset */
    // 程序头表所在处与此文件头的偏移
    Elf32_Off       e_shoff;                /* Section header table file offset */
    // 节头表所在处与此文件头的偏移
    Elf32_Word      e_flags;                /* Processor-specific flags */
    // 针对处理器的标记
    Elf32_Half      e_ehsize;               /* ELF header size in bytes */
    // ELF 文件头的大小（单位为字节）
    Elf32_Half      e_phentsize;            /* Program header table entry size */
    // 程序头表入口大小
    Elf32_Half      e_phnum;                /* Program header table entry count */
    // 程序头表入口数
    Elf32_Half      e_shentsize;            /* Section header table entry size */
    // 节头表入口大小
    Elf32_Half      e_shnum;                /* Section header table entry count */
    // 节头表入口数
    Elf32_Half      e_shstrndx;             /* Section header string table index */
    // 节头字符串编号
} Elf32_Ehdr;

typedef struct
{
  // section name
  Elf32_Word sh_name;
  // section type
  Elf32_Word sh_type;
  // section flags
  Elf32_Word sh_flags;
  // section addr
  Elf32_Addr sh_addr;
  // offset from elf head of this entry
  Elf32_Off  sh_offset;
  // byte size of this section
  Elf32_Word sh_size;
  // link
  Elf32_Word sh_link;
  // extra info
  Elf32_Word sh_info;
  // alignment
  Elf32_Word sh_addralign;
  // entry size
  Elf32_Word sh_entsize;
}Elf32_Shdr;

typedef struct
{
  // segment type
```

```
64    Elf32_Word p_type;
65    // offset from elf file head of this entry
66    Elf32_Off  p_offset;
67    // virtual addr of this segment
68    Elf32_Addr p_vaddr;
69    // physical addr, in linux, this value is meanless and has same value of p_vaddr
70    Elf32_Addr p_paddr;
71    // file size of this segment
72    Elf32_Word p_filesz;
73    // memory size of this segment
74    Elf32_Word p_memsz;
75    // segment flag
76    Elf32_Word p_flags;
77    // alignment
78    Elf32_Word p_align;
79  }Elf32_Phdr;
```

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

Exercise 1.2 阅读./readelf 文件夹中 kerelf.h、readelf.c 以及 main.c 三个文件中的代码，并完成 readelf.c 中缺少的代码，readelf 函数需要输出 ELF 文件的所有 section header 的序号和地址信息，对每个 section header，输出格式为:"%d:0x%x\n"，两个标识符分别代表序号和地址。正确完成 readelf.c 代码之后，在 readelf 文件夹下执行 make 命令，即可生成可执行文件 readelf，它接受文件名作为参数，对 ELF 文件进行解析。 ■

Note 1.3.4 请也阅读 Elf32_Shdr 这个结构体的定义。

关于如何取得后续的节头，可以采用代码中所提示的，先读取节头大小，随后每次累加。也可以利用 C 语言中指针的性质，算出第一个节头的指针之后，当作数组进行访问。

通过刚才的练习，相信你已经对 ELF 文件有了一个比较充分的了解，你可能想进一步解析 ELF 文件来获得更多的信息，不过这个工作已经有人帮你做了。

手抖的同学可能发现，当我们完成 exercise，生成可执行文件 readelf 并且想要运行时，如果不小心忘记打"./"，linux 并没有给我们反馈“command not found”，而是列出了一大堆 help 信息。原来这是因为在 linux 命令中，原本有就一个命令叫做 readelf，它的使用格式是“readelf <option(s)> elf-file(s)”，用来显示一个或者多个 ELF 格式的目标文件的信息。我们可以通过它的选项来控制显示哪些信息。例如，我们执行 readelf -S testELF 命令后，testELF 文件中各个 section 的详细信息将以列表的形式为我们展示出来。我们可以利用 readelf 工具来验证我们自己写的简易版 readelf 输出的结果是否正确，还可以使用“readelf –help”看到该命令各个选项及其对 ELF 文件的解析方式，从

而了解我们想了解的信息。

Thinking 1.2 也许你会发现我们的 readelf 程序是不能解析之前生成的内核文件 (内核文件是可执行文件) 的，而我们刚才介绍的工具 readelf 则可以解析，这是为什么呢？(提示：尝试使用 readelf -h，观察不同) ■

至此，我们 lab1 第一部分的内容已经完成，同学们可以先小小的休息一会儿，认真理解学到的内容，从而更好的进行下一步部分的学习。

现在，我们继续内核的话题。

我们最终生成的内核也是 ELF 格式的，被仿真器载入到内存中。因此，我们暂且只关注 ELF 是如何被载入到内存中，并且被执行的，而不再关心具体的链接细节。也即上文旨在探索关于我们实验环境的编译链接问题。在阅读完上面编译和 ELF 的说明后，应该明确我们 OS 实验课如何编译链接产生实验操作系统的可执行文件并且 GXemul 运行模拟该 ELF 文件。ELF 中有两个相似却不同的概念 segment 和 section，我们之前已经提到过。

我们不妨来看一下，我们之前那个 hello world 程序的各个 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。首先我们使用-l 参数来查看各个 segment 的信息。

```
Elf 文件类型为 EXEC (可执行文件)
入口点 0x400e6e
共有 5 个程序头，开始于偏移量 64

程序头：
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000b33c0 0x00000000000b33c0  R E    200000
  LOAD           0x00000000000b4000 0x00000000006b4000 0x00000000006b4000
                 0x0000000000001cd0 0x0000000000003f48  RW     200000
  NOTE           0x0000000000000158 0x0000000000400158 0x0000000000400158
                 0x0000000000000044 0x0000000000000044  R      4
  TLS            0x00000000000b4000 0x00000000006b4000 0x00000000006b4000
                 0x0000000000000020 0x0000000000000050  R      8
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10

 Section to Segment mapping:
  段节...
   00     .note.ABI-tag .note.gnu.build-id .rela.plt .init .plt .text
   __libc_freeres_fn __libc_thread_freeres_fn .fini .rodata __libc_subfreeres
   __libc_atexit __libc_thread_subfreeres .eh_frame .gcc_except_table
   01     .tdata .init_array .fini_array .jcr .data.rel.ro .got .got.plt .data
   .bss __libc_freeres_ptrs
   02     .note.ABI-tag .note.gnu.build-id
   03     .tdata .tbss
   04
```

这些输出中，我们只需要关注这样几个部分：Offset 代表该段 (segment) 的数据相对于 ELF 文件的偏移。VirtAddr 代表该段最终需要被加载到内存的哪个位置。FileSiz 代表该段的数据在文件中的长度。MemSiz 代表该段的数据在内存中所应当占的大小。表

下方的 Section to Segment mapping 表明每个段各自含有的节。

> **Note 1.3.5** 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，我们又发现了几个重要的问题。

1. 当我们讲加载操作系统内核到正确的 GXemul 模拟物理地址时，讨论的是 Linux 实验环境呢？还是我们编写的操作系统本身呢？
2. 什么叫做正确的位置？到底放在哪里才叫正确。
3. 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三个问题。

1.3.3 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：

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，不通过 MMU。因为转换很简单，我们常常把这些地址称为“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。

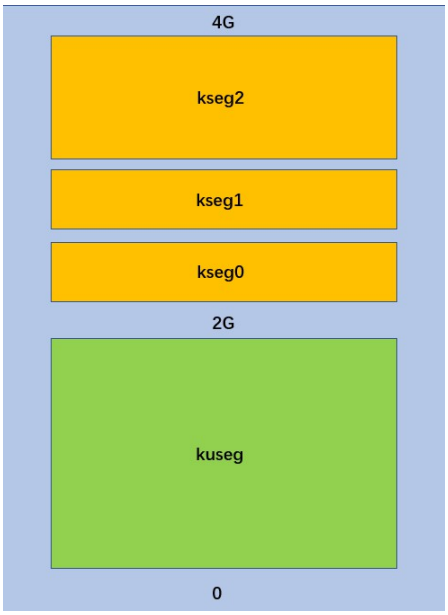


图 1.5: MIPS 内存布局

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，重复映射到了低端 512MB 的物理地址。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，你也许还是很迷茫：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 kseg0 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 MOS 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。

(Tip: 在操作系统实验中，填空的代码往往只是核心部分，而对于实验的理解而言需要阅读的代码有很多，而且在阅读的过程中，希望能够梳理出一个脉络和体系，比如某些代码完成了什么功能，代码间依赖关系是什么？OS 课程需要填空的代码并不多，希望本课程结束之后，学弟学妹们具有读懂一段较长的工程代码的能力。而课上测试将建立在理解代码的基础上。)

> Listing 5: include/mmu.h 中的内存布局图

```
/*
 o     4G ------------>  +------------------------------+------------0x100000000
 o                       |       ...                    |  kseg3
 o                       +------------------------------+------------0xe000 0000
 o                       |       ...                    |  kseg2
 o                       +------------------------------+------------0xc000 0000
 o                       |   Interrupts & Exception     |  kseg1
 o                       +------------------------------+------------0xa000 0000
 o                       |      Invalid memory          |   /|\
 o                       +------------------------------+----|-------Physics Memory Max
 o                       |       ...                    |  kseg0
 o  VPT,KSTACKTOP------> +------------------------------+----|-------0x8040 0000-------end
 o                       |       Kernel Stack           |    | KSTKSIZE            /|\
 o                       +------------------------------+----|------                |
 o                       |       Kernel Text            |    |                    PDMAP
 o      KERNBASE -----> +------------------------------+----|-------0x8001 0000    |
 o                       |   Interrupts & Exception     |   \|/                    \|/
 o      ULIM     -----> +------------------------------+------------0x8000 0000--------
 o                       |         User VPT             |     PDMAP                /|\
 o      UVPT     -----> +------------------------------+------------0x7fc0 0000    |
 o                       |         PAGES                |     PDMAP                |
 o      UPAGES   -----> +------------------------------+------------0x7f80 0000    |
 o                       |         ENVS                 |     PDMAP                |
 o  UTOP,UENVS   -----> +------------------------------+------------0x7f40 0000    |
 o  UXSTACKTOP -/       |     user exception stack     |     BY2PG                |
 o                       +------------------------------+------------0x7f3f f000    |
 o                       |       Invalid memory         |     BY2PG                |
 o      USTACKTOP ----> +------------------------------+------------0x7f3f e000    |
 o                       |     normal user stack        |     BY2PG                |
 o                       +------------------------------+------------0x7f3f d000    |
 a                       |                              |                          |
 a                       ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~                           |
 a                       .                              .                          |
 a                       .                              .                        kuseg
 a                       .                              .                          |
 a                       |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|                          |
 a                       |                              |                          |
 o       UTEXT   -----> +------------------------------+                          |
 o                       |                              |     2 * PDMAP            \|/
 a     0 ------------>   +------------------------------+ ------------------------------
 o
*/
```

相信聪明的你已经发现了内核的正确位置了吧？

1.3.4 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（把编写的操作系统放到模拟硬件的某个物理位置）。之前在分析 ELF 文件时我们曾看到过，编译器在生成 ELF 文件时就已经记录了各节所需要被加载到的位置。同时，我们也发现，最终的可执行文件实际上是由链接器产生的（它将多个目标文件链接产生最终可执行文件）。因此，我们所需要做的，就是控制链接器的链接过程。

接下来，我们就要引入一个神奇的东西：Linker Script。链接器的设计者们在设计链接器的时候面临这样一个问题：不同平台的 ABI(Application Binary Interface) 都不一

样，怎样做才能增加链接器的通用性，使得它能为各个不同的平台生成可执行文件呢？于是，就有了 Linker Script。Linker Script 中记录了各个 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链接脚本，你可以在自己的机器上尝试这一条指令：

```
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 记录了在链接过程中所需要的必要信息。其中最为重要的三个 section 为.text、.data、.bss。这三个 section 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 节相关信息之间的关系。

Listing 6: 用于输出各 section 地址的程序

```
#include <stdio.h>

char msg[]="Hello World!\n";
int count;

int main()
{
    printf("%X\n",msg);
    printf("%X\n",&count);
    printf("%X\n",main);

    return 0;
}
```

该程序的一个可能的输出如下[3]。

```
user@debian ~/Desktop $ ./program
80D4188
80D60A0
8048AAC
```

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

[3]在不同机器上运行，结果也许会有一定的差异

```
共有 29 个节头，从偏移量 0x9c258 开始：

节头：
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 4] .text             PROGBITS        08048140 000140 0620e4 00  AX  0   0 16
  [22] .data             PROGBITS        080d4180 08b180 000f20 00  WA  0   0 32
  [23] .bss              NOBITS          080d50c0 08c0a0 00136c 00  WA  0   0 64
```

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

接下来，我们通过 Linker Script 来尝试调整各节的位置。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：

```
SECTIONS
{
   . = 0x10000;
   .text : { *(.text) }
   . = 0x8000000;
   .data : { *(.data) }
   .bss : { *(.bss) }
}
```

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

```
char msg[]="Hello World!\n";
int count;

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

编译，并查看生产的可执行文件各 section 的信息。

```
user@debian ~/Desktop $ gcc -o test test.c -T test.lds -nostdlib -m32
user@debian ~/Desktop $ readelf -S test
共有 11 个节头，从偏移量 0x2164 开始：

节头：
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 2] .text             PROGBITS        00010000 001000 000018 00  AX  0   0  1
  [ 5] .data             PROGBITS        08000000 002000 00000e 00  WA  0   0  1
  [ 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 文件控制各节被加载到的位置。

Exercise 1.3 填写 tools/scse0_3.lds 中空缺的部分，在 lab1 中，只需要填补.text、.data 和.bss 节，将内核调整到正确的位置上即可。 ■

Note 1.3.6 通过查看内存布局图，同学们应该能找到.text 节的加载地址了，.data 和.bss 只需要紧随其后即可。同学们可以思考一下为什么要这么安排.data 和.bss。注意 lds 文件编辑时“=”两边的空格哦！

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 scirpt 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 节的第一个字节
5. 地址 0 处

阅读实验代码，想一想在我们的实验中，我们是采用何种方式指定程序入口的呢？

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

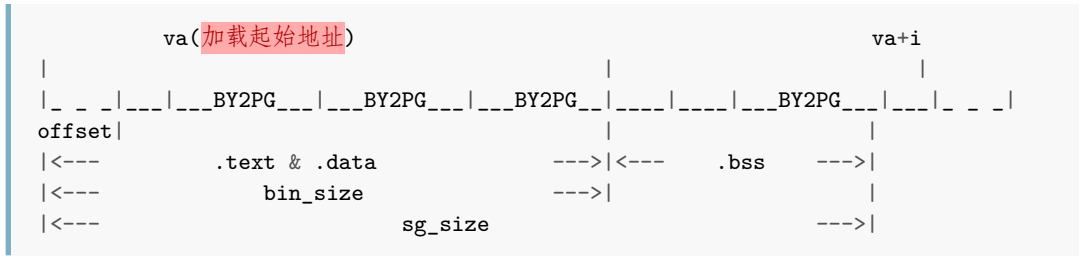
（提示：思考实验中启动过程的两阶段分别由谁执行。） ■

在本章节关于 ELF 的部分中，我们只是使用了仿真器的静态加载功能，并且针对 Elf 的读取解析并不是在我们的小操作系统运行时由我们的操作系统进行读取的。其实，我们的内核，就是一个可执行程序，在这一性质上，它和我们在后续 lab 中需要加载的用户进程并没有什么不同，它们都将自己需要的在内存空间上的排布信息存储在了 ELF 文件中，大体分布可以更形象的表示如下：

```
va = 0x80000000(内核加载起始地址)                    (内核加载终止地址)
|                                                          |                |
|___BY2PG___|___BY2PG___|___BY2PG___|___BY2PG___|_______|___BY2PG___|___|
|                                                          |                |
|<---                        .data & .bss                                --->|
```

其中 BY2PG 代表我们系统中一“页”的大小，在这里需要注意的是，虽然我们静态加载内核的地址保证了我们在进行内核加载时一定是页对齐的，但是内核加载结束时候就不一定是页对齐的了，如果我们要在这个位置加载新的内容，则需要进行是否申请新的页面的考量。

我们在本节进行的内核加载只是最简单，正好对齐的，已知位置的静态加载，在后续的实验中，同学们可能要面对动态的，页不对齐的多段 ELF 加载，这种加载的最坏情况基本可以如下图所示：



这是一个典型的我们需要自己加载的程序段的模样，其中 BY2PG 仍是我们系统中约定的一“页”的大小，va 代表该段要被加载到的虚地址，sg_size 代表该段所需要的的总内存空间的大小,bin_size 则代表该段在 ELF 文件中的大小。

Thinking 1.4 sg_size 和 bin_size 的区别它的开始加载位置并非页对齐，同时 bin_size 的结束位置（va+i）也并非页对齐，最终整个段加载完毕的 sg_size 末尾的位置也并非页对齐，请思考，为了保证页面不冲突（不重复为同一地址申请多个页，以及页上数据尽可能减少冲突），这样一个程序段应该怎样加载内存空间中。

彻底并透彻地理解上图能帮助大家在后续实验中节约很多时间。 ■

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

```c
int fib(int n)
{
    if (n == 0 || n == 1) {
        return 1;
    }
    return fib(n-1) + fib(n-2);
}

int main()
{
```

```
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[4]，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。

[4]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

1.4.2 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。

```
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:       27bdffd8        addiu   sp,sp,-40
3    4000b4:       afbf0020        sw      ra,32(sp)
4    4000b8:       afbe001c        sw      s8,28(sp)
5    4000bc:       afb00018        sw      s0,24(sp)
6    # 中间暂且掠过，只关注一系列 sw 和 lw 操作。
7    400130:       8fbf0020        lw      ra,32(sp)
8    400134:       8fbe001c        lw      s8,28(sp)
9    400138:       8fb00018        lw      s0,24(sp)
10   40013c:       27bd0028        addiu   sp,sp,40
11   400140:       03e00008        jr      ra
12   400144:       00000000        nop
```

我们来回忆一下 C 语言的递归。C 语言递归的过程和栈这种数据结构有着惊人的相似性，函数递归到最底以及返回过程，就好像栈的后入先出。而且每一次递归操作就仿佛将当前函数的所有变量和状态压入了一个栈中，待到返回时再从栈中弹出来，“一切”都保持原样。

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 [5]。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

[5] 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

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。这下同学们应该也不难理解，为什么复杂函数在递归层数过多时会导致程序崩溃，也就是我们常说的“栈溢出”。

Note 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[6]

```
1  lw      $2,40($fp)      # v0 = n;
2  addiu   $2,$2,-1        # v0 = v0 - 1;
3  move    $4,$2           # a0 = v0;      // 即 a0=n-1
4  jal     fib             # v0 = fib(a0);
5  nop                     #
6
7  move    $16,$2          # s0 = v0;
8  lw      $2,40($fp)      # v0 = n;
9  addiu   $2,$2,-2        # v0 = n - 2;
10 move    $4,$2           # a0 = v0;      // 即 a0=n-2
11 jal     fib             # v0 = fib(a0);
12 nop                     #
13
14 addu    $16,$16,$2      # s0 += v0;
15 sw      $16,16($fp)     #
```

我们将汇编所对应的语义用 C 语言标明在右侧。可以看到，调用一个函数就是将参数存放在 a0-a3 寄存器中（我们暂且不关心参数非常多的函数会如何处理），然后使用 jal 指令跳转到相应的函数中。函数的返回值会被保存在 v0-v1 寄存器中。我们通过这两个寄存器的值来获取返回值。

Thinking 1.5 内核入口在什么地方？main 函数在什么地方？我们是怎么让内核进入到想要的 main 函数的呢？又是怎么进行跨文件调用函数的呢？ ■

如果仔细观察了上一个实验中的 tools/scse0_3.lds 这个文件，会发现在其中有 ENTRY(_start) 这一行命令旁边的注释写着，这是为了把入口定为 _start 这个函数。那么，这个函数是什么呢？

我们可以在实验代码的根目录运行一下命令“grep -r _start *”进行文件内容的查找（这个命令在以后会很常用的，特别是想查找函数相互调用关系的时候）。然后发现

[6]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

boot/start.S 中似乎定义了我们的内核入口函数。在 start.S 中，前半部分前半段都是在初始化 CPU。后半段则是我们需要填补的部分。

Note 1.4.2 在阅读 boot/start.S 汇编代码的时候会遇到 LEAF、NESTED 和 END 这三个宏。LEAF 用于定义那些不包含对其他函数调用的函数；NESTED 用于定义那些需要调用其它函数的函数。两者的区别在于对栈帧 (frame) 的处理，LEAF 不需要过多处理堆栈指针。简言之，现阶段只需要知道这两个宏是在汇编代码中用来定义函数的。

END 宏则仅仅是用来结束函数定义的。

阅读 LEAF 等宏定义的时候，我们会发现这些宏的定义是一系列以“.”开头的指令。这些指令不是 MIPS 汇编指令，而是 Assembler directives(参考https://sourceware.org/binutils/docs-2.34/as/Pseudo-Ops.html)，主要是在编译时用来指定目标代码的生成。

Exercise 1.4 完成 boot/start.S 中空缺的部分。设置栈指针，跳转到 main 函数。使用/OSLAB/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)。 ■

Thinking 1.6 查阅《See MIPS Run Linux》一书相关章节，解释 boot/start.S 中下面几行对 CP0 协处理器寄存器进行读写的意义。具体而言，它们分别读/写了哪些寄存器的哪些特定位，从而达到什么目的？

```
1   /* Disable interrupts */
2   mtc0        zero, CP0_STATUS
3
4   ......
5
6   /* disable kernel mode cache */
7   mfc0        t0, CP0_CONFIG
8   and         t0, ~0x7
9   ori         t0, 0x2
10  mtc0        t0, CP0_CONFIG
```

■

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。

Note 1.4.3 main 函数虽然为 c 语言所书写，但是在被编译成汇编之后，其入口点会被翻译为一个标签，类似于：

main:

XXXXXX

想想看汇编程序中，如何调用函数？

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。

表 1.1: MIPS 通用寄存器

寄存器编号	助记符	用途
0	zero	值总是为 0
1	at	（汇编暂存寄存器）一般由汇编器作为临时寄存器使用。
2-3	v0-v1	用于存放表达式的值或函数的整形、指针类型返回值
4-7	a0-a3	用于函数传参。其值在函数调用的过程中不会被保存。若函数参数较多，多出来的参数会采用栈进行传递
8-15	t0-t7	用于存放表达式的值的临时寄存器; 其值在函数调用的过程中不会被保存。
16-23	s0-s7	保存寄存器; 这些寄存器中的值在经过函数调用后不会被改变。
24-25	t8-t9	用于存放表达式的值的临时寄存器; 其值在函数调用的过程中不会被保存。当调用位置无关函数 (position independent function) 时，25 号寄存器必须存放被调用函数的地址。
26-27	k0-k1	仅被操作系统使用。
28	gp	全局指针和内容指针。
29	sp	栈指针。
30	fp 或 s8	保存寄存器（同 s0-s7)。也可用作帧指针。
31	ra	函数返回地址。

其中，只有 16-23 号寄存器和 28-30 号寄存器的值在函数调用的前后是不变的[7]。对于 28 号寄存器有一个特例：当调用位置无关代码 (position independent code) 时，28 号寄存器的值是不被保存的。

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。

[7]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。

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 drivers/gxconsole/console.c。

首先大家先对他们的大致的内容有个了解：

1. drivers/gxconsole/console.c: 这个文件负责往 GXemul 的控制台输出字符，其原理为读写某一个特殊的内存地址
2. lib/printf.c: 此文件中，实现了 printf, 但其所做的，实际上是把输出字符的函数，接受的输出参数给传递给了 lp_Print 这个函数
3. lib/print.c: 此文件中，实现了 lp_Print 函数，这个函数是 printf 函数的真正内核。

接着为了便于理解，我们一起来梳理一下这几个文件之间的关系。

lib/printf.c 中定义了我们的 printf 函数。但是，仔细观察就可以发现，这个 printf 函数实际上基本什么事情都没有做。他只是把接受到的参数，以及 myoutput() 函数指针给传入 lp_Print 这个函数中。

```
1  void printf(char *fmt, ...)
2  {
3      va_list ap;
4      va_start(ap, fmt);
5      lp_Print(myoutput, 0, fmt, ap);
6      va_end(ap);
7  }
```

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

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

stdarg.h 头文件中为处理变长参数表定义了一组宏和变量类型如下：

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

4. va_end(va_list ap)，结束使用变长参数表的宏。

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

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

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

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

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

在所有参数处理完毕后，退出函数前，需要调用 va_end 宏一次以结束变长参数表的使用。

上述变长参数使用方法转述于 Kernighan, Ritchie 的《C 程序设计语言》一书。同学们可以参考该书相应章节和附录，以及这篇博客：https://www.cnblogs.com/qiwu1314/p/9844039.html来了解可变长参数的使用方法。由于 stdarg.h 在不同平台上实现不同，我们不介绍其底层实现。有兴趣的同学可以参考http://www.cplusplus.com/reference/cstdarg/、https://www.cnblogs.com/cpoint/p/3368993.html（这个博客关于传参的内存原理在 Lab4 及后续编译技术课程中可能会有帮助）等资料。

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

```
static void myoutput(void *arg, char *s, int l)
{
    int i;

    // special termination call
    if ((l==1) && (s[0] == '\0')) return;

    for (i=0; i< l; i++) {
        printcharc(s[i]);
        if (s[i] == '\n') printcharc('\n');
    }
}
```

可以发现，他实际上的核心是调用了一个叫做 printcharc 的函数。这个函数在哪呢？我们可以在./drivers/gxconsole/gxconsole.c 下面找到：

```
void printcharc(char ch)
{
        *((volatile unsigned char *) PUTCHAR_ADDRESS) = ch;
}
```

原来，想让控制台输出一个字符，实际上是对某一个内存地址写了一串数据。

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

为了方便大家理解这个比较复杂的函数，我们来给大家简单介绍一下。

首先，我们来看这个宏函数，这个函数先用宏定义来简化了 myoutput 这个函数指针的使用

```
#define                 OUTPUT(arg, s, l)   \
  { if (((l) < 0) || ((l) > LP_MAX_BUF)) { \
       (*output)(arg, (char*)theFatalMsg, sizeof(theFatalMsg)-1); for(;;); \
    } else { \
      (*output)(arg, s, l); \
    } \
  }
```

观察 lp_Print 函数的参数，想想我们调用它时传入的参数，我们可以发现这个宏函数中的函数指针，就是我们之前提到过的 myoutput 函数，而这个宏定义的函数 OUTPUT，其实就是简化了 myoutput 的调用过程。

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

```
    int longFlag;//标记是否为 long 型
    int negFlag;//标记是否为负数
    int width;//标记输出宽度
    int prec;//标记小数精度
    int ladjust;//标记是否左对齐
    char padc;//填充多余位置所用的字符
```

接下来，我们发现整个的函数主体是一个没有终止条件的无限循环，这肯定是不对的，看来我们要填补的地方就是这里了。这个循环中，主要有两个逻辑部分，第一部分：找到格式符%，并分析输出格式; 第二部分，根据格式符分析结果进行输出。

什么叫找到格式并分析输出格式呢？想一想，我们在使用 printf 输出信息时，%ld，%-b，%c 等等会被替换为相应输出格式的变量的值，他们就叫做格式符。在第一部分中，我们要干的就是解析 fmt 格式字符串，如果是不需要转换成变量的字符，那么就直接输出; 如果碰到了格式字符串的结尾，就意味着本次输出结束，停止循环; 但是如果碰到我们熟悉的%，那么就要按照 printf 的规格要求，开始解析格式符，分析输出格式，用上述变量记录下来这次对变量的输出要求，例如是要左对齐还是右对齐，是不是 long 型，是否有输出宽度要求等等，然后进入第二部分。

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

到这里我们 printf 整个流程就梳理的差不多了，细节部分就靠同学们自己思考啦! 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 1.5 阅读相关代码和下面对于函数规格的说明，补全 lib/print.c 中 lp_Print() 函数中两处缺失的部分来实现字符输出。第一处缺失部分：找到% 并分析输出格式；第二处缺失部分：取出参数，输出格式串为"%[flags][width][.precision][length]d" 的情况。 ■

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

下面是我们需要完成的 printf 函数的具体说明，同学们可以参考 cppreference 中有关 C 语言 printf 函数的文档[8]或者标准 C++ 文档[9]，对 printf 函数进行更加详细的了解。

函数原型：

```
1 void printf(const char* fmt, ...)
```

参数 fmt 是和 C 中 printf 类似的格式字符串，除了可以包含一般字符，还可以包含格式符 (format specifiers)，但略去并新添加了一些功能，格式符的原型为：

$$\%[flags][width][.precision][length]specifier$$

其中 specifier 指定了输出变量的类型，参见下表1.2：

表 1.2: Specifiers 说明

Specifier	输出	例子
b	无符号二进制数	110
d D	十进制数	920
o O	无符号八进制数	777
u U	无符号十进制数	920
x	无符号十六进制数，字母小写	1ab
X	无符号十六进制数，字母大写	1AB
c	字符	a
s	字符串	sample

除了 specifier 之外，格式符也可以包含一些其它可选的副格式符 (sub-specifier)，有 flag(表1.3)：

[8] https://en.cppreference.com/w/c/io/fprintf/

[9] http://www.cplusplus.com/reference/cstdio/printf/

表 1.3: flag 说明

flag	描述
-	在给定的宽度 (width) 上左对齐输出，默认为右对齐
0	当输出宽度和指定宽度不同的时候，在空白位置填充 0

和 width(表1.4):

表 1.4: width 说明

width	描述
数字	指定了要打印数字的最小宽度，当这个值大于要输出数字的宽度，则对多出的部分填充空格，但当这个值小于要输出数字的宽度的时候则不会对数字进行截断。

以及 precision(表1.5):

表 1.5: precision 说明

.precision	描述
. 数字	指定了精度，不同标识符下有不同的意义，但在我们实验的版本中这个值只进行计算而没有具体意义，所以不赘述。

另外，还可以使用 length 来修改数据类型的长度，在 C 中我们可以使用 l、ll、h 等，但这里我们只使用 l，参看下表1.6

表 1.6: length 说明

length	Specifier	
	d D	b o O u U x X
l	long int	unsigned long int

1.6 实验正确结果

如果你正确地实现了前面所要求的全部内容，你将在 GXemul 中观察到如下输出，这标志着你顺利完成了 lab1 实验。

```
GXemul 0.4.6    Copyright (C) 2003-2007  Anders Gavare
Read the source code and/or documentation for other Copyright messages.

Simple setup...
    net: simulating 10.0.0.0/8 (max outgoing: TCP=100, UDP=100)
```

```
        simulated gateway: 10.0.0.254 (60:50:40:30:20:10)
            using nameserver 202.112.128.51
    machine "default":
        memory: 64 MB
        cpu0: R3000 (I+D = 4+4 KB)
        machine: MIPS test machine
        loading gxemul/vmlinux
        starting cpu0 at 0x80010000
--------------------------------------------------------------------------------

main.c: main is start ...

init.c: mips_init() is called

panic at init.c:24: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

1.7 如何退出 GXemul

1. 按 ctrl+c, 以中断模拟
2. 输入 quit 以退出仿真器

一定要区分好“退出仿真器”，和“把仿真器挂在后台”这两件事。仿真器是相当占用系统资源的。

可能有同学不小心按下 ctrl+z 把仿真器挂到后台，或是不确定自己有没有把挂起的进程关掉，可以通过 jobs 命令观察后台有多少正在运行的作业。如果发现有多个仿真器正在运行的话，可以使用 fg 把 GXemul 作业调至前台然后使用 ctrl+z 停掉，也可以使用 kill 命令直接杀死。同学们可以自行了解一下 Linux 后台进程管理的知识。

另外有一个小 tips，从 lab1 起，git add –all 之前一定要先 make clean，清空编译结果，这样可以最大限度降低评测机的存储压力。

1.8 任务列表

- **Exercise-修改 include.mk 文件**
- **Exercise-完成 readelf.c**
- **Exercise-填写 tools/scse0_3.lds 中空缺的部分**
- **Exercise-完成 boot/start.S**
- **Exercise-完成 lp_Print() 函数**

1.9 实验思考

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

- **思考-ELF 中典型段在内存中的表现形式**
- **思考-跨文件调用函数的问题**
- **思考-boot/start.S 中对 CP0 操作的意义**

CHAPTER 2

内存管理

Note 2.0.1 建议同学们同时阅读 R3000 文档 Chapter 6。

2.1 实验目的

- 了解 MIPS-R3000 的访存流程与内存映射布局
- 掌握与实现物理内存的管理方法（链表法）
- 掌握与实现虚拟内存的管理方法（两级页表）
- 掌握 TLB 清除与重填的流程

Note 2.1.1 在我们的实验中，将两级页表放置于内存中 (内核数据结构)，在需要访问虚拟地址时（通过中断机制）将两级页表填入 TLB。由于在 Lab2 中，尚未建立中断异常的处理程序，因此我们着重于**维护两级页表**，而根据两级页表来填写 TLB 的例程则在 `lib/genex.S` 中直接给出。

2.2 R3000 访存流程概览

R3000 是我们在后续所有 Lab 中采用的 MIPS CPU，了解其各种特性有助于同学们对实验的理解。下面将带大家了解 R3000 的访存流程。

2.2.1 CPU 发出地址

CPU 运行程序时，会发出地址，进行内存读写操作。在计组课设中，CPU 直接发送物理地址，这是为了简化内存操作，让大家关注 CPU 内部的计算与控制逻辑。而在操作系统课设中，**R3000 CPU 只会发出虚拟地址**。

Thinking 2.1 请你根据上述说明，回答问题：在我们编写的 C 程序中，指针变量中存储的地址是虚拟地址还是物理地址？MIPS 汇编程序中 `lw`, `sw` 使用的是虚拟地址还是物理地址？ ■

2.2.2 虚拟地址映射到物理地址

在 R3000 上，**虚拟地址映射到物理地址**，随后使用物理地址来访存。与我们实验相关的映射与寻址规则（内存布局）如下：

- 若虚拟地址处于 `0x80000000~0x9fffffff` (`kseg0`)，则将虚拟地址的**最高位**置 0 得到物理地址，通过 cache 访存。**这一部分用于存放内核代码与数据结构**。
- 若虚拟地址处于 `0xa0000000~0xbfffffff` (`kseg1`)，则将虚拟地址的**最高 3 位**置 0 得到物理地址，不通过 cache 访存。**这一部分用于映射外设**。
- 若虚拟地址处于 `0x00000000~0x7fffffff` (`kuseg`)，则需要**通过 TLB 来获取物理地址**，通过 cache 访存。

在 R3000 中，使用 MMU 来完成上述转换，而 MMU 中是采用**硬件 TLB** 来完成地址映射的。在我们采用的 R3000 中，我们需要通过**软件填写 TLB** 来构建地址映射。所有的在低 2GB 空间的访存操作都需要经过 TLB。

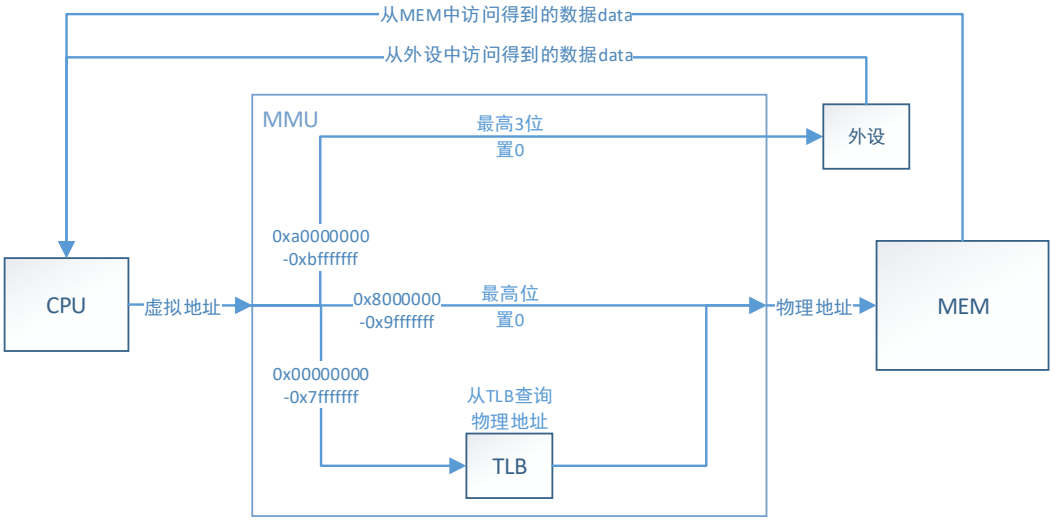


图 2.1: cpu-tlb-memory 关系

2.3 内核程序启动

lab1 中我们已经来到了 `main` 函数，因此下面将从 `main` 函数开始对 lab2 内存管理相关内容的流程进行介绍。

可以看到，在 main 函数中，调用了 mips_init() 这一函数，它的定义位于 include/pmap.h 中，实现位于 init/init.c 中，这个函数按顺序分别调用了如下五个函数：

- mips_detect_memory()，实现位于 mm/pmap.c，它的作用是对一些和内存管理相关的变量进行初始化，包括：
  - maxpa，它的含义是物理地址的最大值 +1，即 $[0, \text{maxpa} - 1]$ 范围内所有整数所组成的集合等于物理地址的集合。
  - basemem，表示物理内存对应的字节数。
  - npage，表示总物理页数。
  - extmem，表示扩展内存的大小，本实验中不涉及扩展内存，直接设为 0 即可。

> **Exercise 2.1** 完成 mips_detect_memory 函数。
>
> MOS 中，物理内存的大小为 64MB，页的大小为 4KB。 ■

- mips_vm_init()，实现位于 mm/pmap.c 中。

  这个函数调用了同样位于 mm/pmap.c 的 alloc 函数，我们先来看看它：

```
1  static void *alloc(u_int n, u_int align, int clear)
2  {
3      extern char end[];
4      u_long alloced_mem;
5
6      /* Initialize `freemem` if this is the first time.
7        * The first virtual address that the linker did *not* assign
8        * to any kernel code or global variables.
9        */
10     if (freemem == 0) {
11         freemem = (u_long)end;
12     }
13
14     /* Step 1: Round up `freemem` up to be aligned properly */
15     freemem = ROUND(freemem, align);
16
17     /* Step 2: Save current value of `freemem` as allocated chunk. */
18     alloced_mem = freemem;
19
20     /* Step 3: Increase `freemem` to record allocation. */
21     freemem = freemem + n;
22
23     /* Check if we're out of memory. If we are, PANIC !! */
24     if (PADDR(freemem) >= maxpa) {
25         panic("out of memory\n");
26         return (void *)-E_NO_MEM;
27     }
28
29     /* Step 4: Clear allocated chunk if parameter `clear` is set. */
30     if (clear) {
31         bzero((void *)alloced_mem, n);
32     }
33
34     /* Step 5: return allocated chunk. */
```

```
35          return (void *)alloced_mem;
36      }
```

这段代码的作用是分配 n 字节的空间并返回初始虚拟地址，同时保证 align 可以整除初始虚拟地址，若 clear 为真则将对应空间的值清零，否则不清零。

我们来一行行解释这个函数的实现原理：

- extern char end[]; 显然是一个定义在该文件之外的变量，它并不在一个 C 代码文件中，而是在 lab1 填写的 tools/scse0_3.lds 中：

  ```
  1       . = 0x80400000;
  2       end = . ;
  ```

  也就是说它对应虚拟地址 0x80400000，根据映射规则，对应的物理地址是 0x00400000，因此我们所管理的物理地址区间为 $[0x00400000, maxpa - 1]$。同时，这段物理地址区间和虚拟地址区间 $[0x80400000, 0x83ffffff]$ 一一对应，因此当虚拟地址被分配出去，代表对应的物理地址也被分配了出去。

- freemem 作为一个全局变量，初始值为 0，因此第一次调用该函数将其赋为 end，freemem 的含义是 $[0x80400000, freemem - 1]$ 的虚拟地址都已经被分配了。

- Step 1，ROUND(a, n) 是一个定义在 include/types.h 的宏，它的作用是返回 $\left\lceil \frac{a}{n} \right\rceil n$，要求 n 必须是 2 的非负整数次幂，因此 align 也必须是一个非负整数次幂。这行代码的含义即找到最小的符合条件的初始虚拟地址，中间未用到的地址空间全部放弃。

  > **Note 2.3.1** 与之相对的，include/types.h 还有另一个宏 ROUNDDOWN(a, n)，它的作用是返回 $\left\lfloor \frac{a}{n} \right\rfloor n$，同样要求 n 必须是 2 的非负整数次幂。

- Step 2，将 alloced_mem 设为初始虚拟地址。

- Step 3，将 freemem 增加 n，表示这一段都被使用了。

  PADDR(x) 是一个返回虚拟地址 x 所对应物理地址的宏，它定义在 include/mmu.h，这里要求 x 必须是 kseg0 的虚拟地址，这部分的虚拟地址只需要将二进制下最高位清零就可以得到对应的物理地址。这段代码的含义即检查分配的空间是否超出了最大物理地址，若是则产生报错。

  > **Note 2.3.2** 与之相对的，include/mmu.h 还有另一个宏 KADDR(x)，它的作用是返回物理地址 x 所位于 kseg0 的虚拟地址，原理同上。

- Step 4，如果 clear 为真，则使用 bzero 函数将这一部分清零，bzero 函数的实现位于 init/init.c，实现方法类似于 bcopy 函数。

- Step 5，最后将初始虚拟地址返回即可。

接着，我们回到 mips_vm_init() 函数，它依次实现了如下三个功能：

  - 使用 alloc 函数为内核一级页表分配 1 页物理内存，并将 boot_pgdir 设为对应的初始虚拟地址。
  - 使用 alloc 函数为物理内存管理所用到的 Page 结构体按页分配物理内存，设 npage 个 Page 结构体的大小为 n，一页的大小为 m，由上述函数分析可知分配的大小为 $\left\lceil \frac{n}{m} \right\rceil m$。同时将分配的空间映射到上述的内核页表中，对应的起始虚拟地址为 UPAGES。
  - 为进程管理所用到的 Env 结构体按页分配物理内存，设 NENV 是最大进程数，NENV 个 Env 结构体的大小为 n，一页的大小为 m，由上述函数分析可知分配的大小为 $\left\lceil \frac{n}{m} \right\rceil m$。同时将分配的空间映射到上述的内核页表中，对应的起始虚拟地址为 UENVS。

    这一部分涉及 lab3 知识，不过可以看出它和上一条几乎是一样的，仅有的区别在于结构体不同和对应结构体的数量不同。

    上述映射用到了 boot_map_segment 函数，它的具体作用会在后面详细叙述。

- page_init()，实现位于 mm/pmap.c 中，它的作用是初始化 Pages 结构体以及空闲链表。这个函数的具体作用会在后面详细叙述。
- physical_memory_manage_check()，是一个用于检测你所填写代码是否正确的函数，与整体启动流程无关。
- page_check()，也是一个用于检测你所填写代码是否正确的函数，与整体启动流程无关。

2.4 物理内存管理

MOS 中的内存管理使用页式内存管理，采用链表链表法管理空闲物理页框。

在后续的 lab 中，也有一些地方需要用到链表，因此 MOS 中使用宏对链表的操作进行了封装。

2.4.1 链表宏

链表宏的定义位于 include/queue.h，其实现的是 **双向链表**，下面将对一些主要的宏进行解释:

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

```
1 LIST_ENTRY(int) a;
```

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

- `LIST_EMPTY(head)`，判断头部结构体 `head` 对应的链表是否为空。
- `LIST_FIRST(head)`，将返回头部结构体 `head` 对应的链表的首个元素。
- `LIST_INIT(head)` 是将头部结构体 `head` 对应的链表初始化，等价于将首个元素清空。
- `LIST_NEXT(elm, field)`，结构体 `elm` 包含的名为 `field` 的数据，类型是一个链表项 `LIST_ENTRY(type)`，返回其指向的下一个元素。下面出现的 `field` 含义均和此相同。
- `LIST_INSERT_AFTER(listelm, elm, field)`，将 `elm` 插到已有元素 `listelm` 之后。
- `LIST_INSERT_BEFORE(listelm, elm, field)`，将 `elm` 插到已有元素 `listelm` 之前。
- `LIST_INSERT_HEAD(head, elm, field)`，将 `elm` 插到头部结构体 `head` 对应链表的头部。
- `LIST_INSERT_TAIL(head, elm, field)`，将 `elm` 插到头部结构体 `head` 对应链表的尾部。
- `LIST_REMOVE(elm, field)`，将 `elm` 从对应链表中删除。

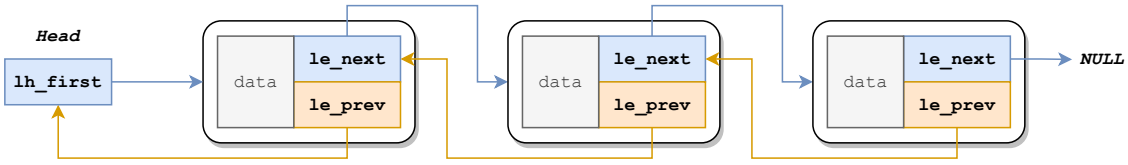


图 2.2: macro list

C++ 中可以使用 `stack<T>` 定义一个类型为 `T` 的栈，Java 中可以使用 `HashMap<K,V>` 定义一个键类型为 `K` 且值类型为 `V` 的哈希表。这种模式称为泛型，C 语言并不支持泛型，因此需要通过宏另辟蹊径来实现「泛型」。

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

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

`LIST_INSERT_TAIL` 的功能是将一个元素插入链表尾部，由于没有记录链表的尾指针，因此该函数的实现和 `LIST_INSERT_HEAD` 有很大区别，请注意区分。 ■

Thinking 2.2 请你思考下述两个问题：

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

■

2.4.2 内存控制块

接着我们回到物理内存管理，MOS 中维护 `npage` 个内存控制块，也就是 `Page` 结构体。每一个内存控制块对应一页的物理内存，注意并不是这个结构体就是对应的物理内存，而是用这个结构体来管理这块内存的分配。

Note 2.4.1 `npage` 个 `Page` 和 `npage` 个物理页面一一顺序对应，具体来说，`npage` 个 `Page` 的起始地址为 `pages`，则 `pages[i]` 对应从 0 开始计数的第 i 个物理页面。两者的转换可以使用 `include/pmap.h` 中的 `page2pa` 和 `pa2page` 这两个函数。

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

```
A:
struct Page_list{
    struct {
        struct {
            struct Page *le_next;
            struct Page **le_prev;
        }* pp_link;
        u_short pp_ref;
    }* lh_first;
}
```

```
B:
struct Page_list{
    struct {
        struct {
            struct Page *le_next;
            struct Page **le_prev;
        } pp_link;
        u_short pp_ref;
    } lh_first;
}
```

```
C:
struct Page_list{
    struct {
        struct {
            struct Page *le_next;
            struct Page **le_prev;
        } pp_link;
        u_short pp_ref;
    }* lh_first;
}
```

将这些结构体全部插入一个链表中,这个链表被称为空闲链表,它对应 page_free_list。

当一个进程需要被分配内存时，将链表头部的内存控制块对应的那一页物理内存分配出去，同时将该内存控制块从空闲链表的头部删去。

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

首先我们来看 Page 结构体，它的定义位于 include/pmap.h 中:

```
typedef LIST_ENTRY(Page) Page_LIST_entry_t;

struct Page {
  Page_LIST_entry_t pp_link;        /* free list link */

  // Ref is the count of pointers (usually in page table entries)
  // to this page.  This only holds for pages allocated using
  // page_alloc.  Pages allocated at boot time using pmap.c's "alloc"
  // do not have valid reference count fields.

  u_short pp_ref;
};
```

- Page_LIST_entry_t 定义为 LIST_ENTRY(Page)，因此 pp_link 即为对应的链表项。
- pp_ref 对应这一页物理内存被引用的次数，它等于有多少虚拟页映射到该物理页。

2.4.3 相关函数

接着介绍 4 个相关的函数，它们的实现均位于 mm/pmap.c 中:

- page_init()，在启动过程中提到过，它实现了以下三个功能:
  1. 首先利用链表相关宏初始化 page_free_list。
  2. 接着将 mips_vm_init() 中用到的空间对应的物理页面的内存控制块的引用次数全部标为 1。
  3. 最后将剩下的物理页面的引用次数全部标为 0，并将它们对应的内存控制块插入到 page_free_list。

Exercise 2.3 完成 `page_init` 函数。

请你借助函数中的注释提示，完成上述三个功能。此外，这里也给出一些提示：

1. 使用链表初始化宏 `LIST_INIT`。
2. 将 `freemem` 按照 `BY2PG` 进行对齐。
3. 将 `freemem` 以下页面对应的结构体中的 `pp_ref` 标为 1。
4. 将其它页面对应的结构体中的 `pp_ref` 标为 0 并使用 `LIST_INSERT_HEAD` 将其插入空闲链表。

■

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

Exercise 2.4 完成 `page_alloc` 函数。

一种供参考的框架如下：

```
int page_alloc(struct Page **pp) {
  struct Page *tmp;
  if (/* I. `page_free_list` is empty */) return -E_NO_MEM;
  // negative return value indicates exception.

  tmp = /* II. the first item in `page_free_list` */;

  /* III. remove this page from the list */;

  bzero(/* IV. kernel virtual address of this page */, BY2PG);

  *pp = tmp;
  return 0;
  // zero indicates success.
}
```

对框架中四处代码填空的提示：

1. 使用链表宏 `LIST_EMPTY`。
2. 使用链表宏 `LIST_FIRST`。
3. 使用链表宏 `LIST_REMOVE`。
4. 使用函数 `page2kva`。

■

- `page_decref(struct Page *pp)`，它的作用是令 `pp` 对应内存控制块的引用次数减少 1，如果引用次数为 0 则会调用下面的 `page_free` 函数将对应物理页面重新设置为空闲页面。

- page_free(struct Page *pp)，它的作用是判断 pp 指向内存控制块对应的物理页面引用次数是否为 0，若是则该物理页面为空闲页面，将其对应的内存控制块重新插入到 page_free_list。

Exercise 2.5 完成 page_free 函数。

一种供参考的框架如下：

```
void page_free(struct Page *pp) {
    if (pp->pp_ref == 0) {
        /* I. insert this item into `page_free_list` */
        return;
    } else if (pp->pp_ref > 0) return; // in use

    panic("pp_ref is less then 0");
}
```

提示：使用链表宏 LIST_INSERT_HEAD。 ■

2.4.4 正确结果展示

在在完成物理内存管理后，在 init/init.c 中将 page_check() 注释掉，并需要保证 physical_memory_manage_check() 没有被注释。然后编译运行内核，显示如下信息即通过物理内存管理的单元测试：

```
main.c: main is start ...
init.c: mips_init() is called
Physical memory: 65536K available, base = 65536K, extended = 0K
to memory 80401000 for struct page directory.
to memory 80431000 for struct Pages.
pmap.c:  mips vm init success
The number in address temp is 1000
physical_memory_manage_check() succeeded
panic at init.c:42: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

其中，最后一行的数字 42 在不同的代码中是不一样的。

2.5 虚拟内存管理

MOS 中，采用 PADDR 与 KADDR 这两个宏就可以对位于 kseg0 的虚拟地址和对应的物理地址进行转换。它们已经在前面有所提到过。

对于位于 kuseg 的虚拟地址，MOS 中采用两级页表结构对其进行管理。

2.5.1 两级页表结构

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

两级页表机制相比计组理论中学习的单级页表机制，将虚拟页号进一步分为了两部分。

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

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

访存时，先通过一级页表基地址和一级页表项的偏移量，找到对应的一级页表项，得到对应的二级页表基地址的物理页号，再根据二级页表项的偏移量找到所需的二级页表项，进而得到对应物理页的物理页号。

图解流程如下：

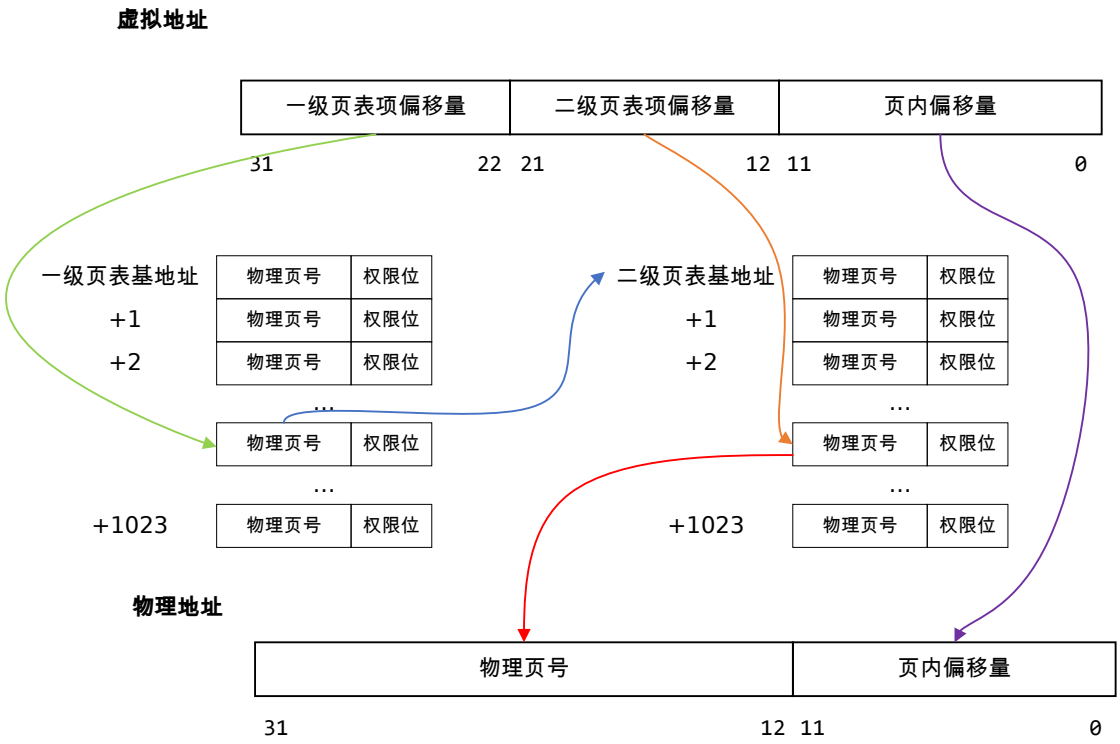


图 2.3: 两级页表结构的地址变换机制

CPU 发出的地址均为虚拟地址，因此获取相关物理地址后，需要转换为虚拟地址再访问。对页表进行操作时处于内核态，因此使用宏 `KADDR` 即可。

存储上，无论是一级页表还是二级页表，它们的结构都是一样的，只不过每个页表项记录的物理页号含义有所不同。每个页表均由 1024 个页表项组成，每个页表项由 32 位组成，包括 20 位物理页号以及 12 位标志位。关于标志位的相关定义，其实前面已经提到过了，因为每个页表项最后都会填入 TLB 中，因此它的规范和 EntryLo 寄存器的规范相同。每个页表所占的空间为 4KB，恰好为一个物理页面的大小。

可以发现，一个页表项可以恰好由一个 32 位整型来表示，因此我们使用 `Pde` 来表示一个一级页表项，用 `Pte` 来表示一个二级页表项，这两者的本质都是 `u_long`，它们对应的 `typedef` 位于 `include/mmu.h`。

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

这里再复习一下 TLB 的功能。MOS 中，通过页表进行地址变换时，硬件只会查询 TLB，如果查找失败，就会触发 TLB 中断，对应的异常中断处理就会对 TLB 进行重填。需要特别注意的是，lab2 里面 **没有开启**这个功能，因此上述过程你无法测试出来，但你一定要知道它的原理。这一部分的详细内容会在下一部分进行介绍。

使用 `tlb_invalidate` 函数可以实现删除特定虚拟地址的映射，每当页表被修改，就需要调用该函数以保证下次访问该虚拟地址时诱发 TLB 重填以保证访存的正确性。

除此之外，一般的操作系统中，当物理页框全部被映射，此时有新的虚拟页框需要映射到物理页框，那么就需要将一些物理页框置换到硬盘中，选择哪个物理页框的算法就称为页面置换算法，例如我们熟悉的 FIFO 算法和 LRU 算法。

然而在我们的 MOS 中，对这一过程进行了简化，一旦物理页框全部被分配，进行新的映射时并不会进行任何的页面置换，而是直接返回错误，这对应 `page_alloc` 函数中返回的 `-E_NO_MEM`。

2.5.2 相关函数

下面我们介绍 7 个与页表相关的函数，其中：

- 以 `boot` 为前缀的函数名表示**该函数是在系统启动、初始化时调用的**。
- 不以 `boot` 为前缀的函数名表示**该函数是在进程的生命周期中被调用的，用于操作进程的页表**。

启动时的二级页表检索函数

`static Pte *boot_pgdir_walk(Pde *pgdir, u_long va, int create)`，它返回一级页表基地址 `pgdir` 对应的两级页表结构中，`va` 这个虚拟地址所在的二级页表项，如果 `create` 不为 0 且对应的二级页表不存在则会使用 `alloc` 函数分配一页物理内存用于存放。这里之所以使用 `alloc` 而不用 `page_alloc` 是因为这个函数是在内核启动过程中使用的，此时还没有建立好物理内存管理机制。

该函数的图解流程如下图中的蓝色部分：(注意，此图适用于 walk 和 map/insert 系列函数，图中蓝色部分是 walk，黄色部分是 map/insert)

Exercise 2.6 完成 `boot_pgdir_walk` 函数。

一种供参考的框架如下：

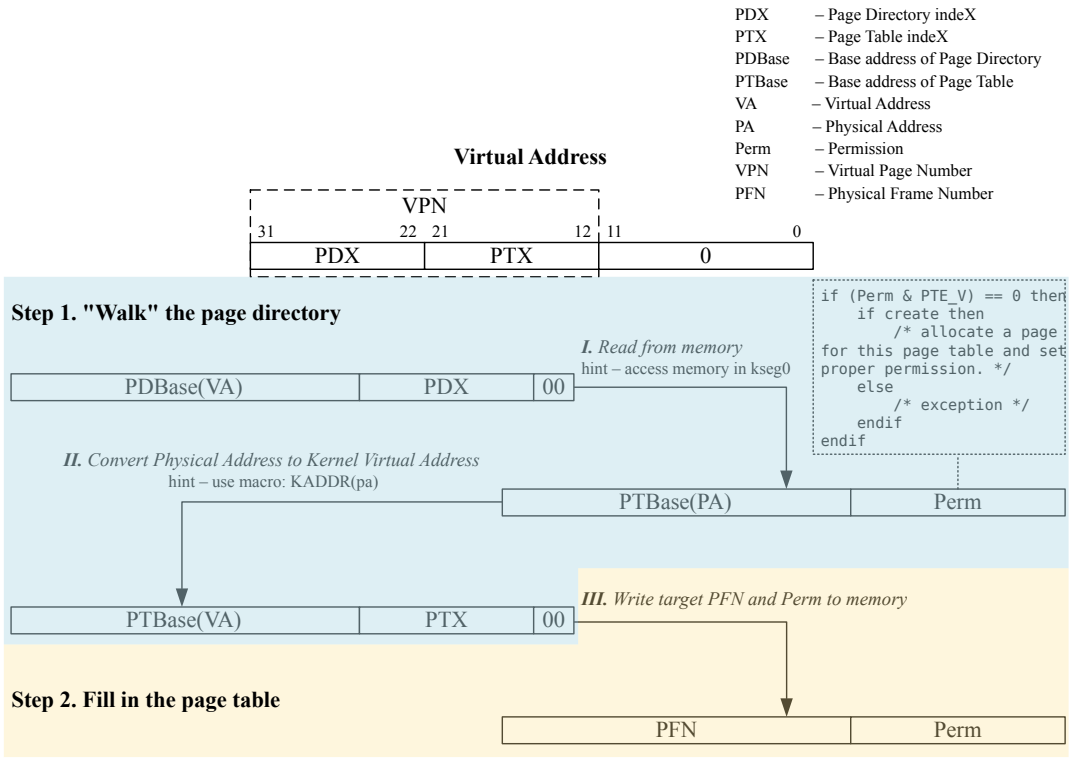


图 2.4: walk & map(insert) 系列函数图解流程

```c
static Pte *boot_pgdir_walk(Pde *pgdir, u_long va, int create) {
    Pde *pgdir_entry = pgdir + PDX(va); // 对应上图中步骤 I

    // check whether the page table exists
    if ((*pgdir_entry & PTE_V) == 0) {
        if (create) {
            /**
            * use `alloc` to allocate a page for the page table
            * set permission: `PTE_V | PTE_R`
            * hint: `PTE_V` <==> valid ; `PTE_R` <==> writable
            */
        } else return 0; // exception
    }

    // return the address of entry of page table
    return ((Pte *)(/* II. Kernel Virtual Address of PTBase */)) + PTX(va);
}
```

对框架中两处代码填空的提示：

1. 使用 `alloc` 函数分配内存，权限位设置为 `PTE_V | PTE_R`。
2. 使用宏 `KADDR` 和 `PTE_ADDR`。

启动时区间地址映射函数

`void boot_map_segment(Pde *pgdir, u_long va, u_long size, u_long pa, int perm)`，它的作用是将一级页表基地址 `pgdir` 对应的两级页表结构做区间地址映射，将虚拟地址区间 $[va, va + size - 1]$ 映射到物理地址区间 $[pa, pa + size - 1]$，因为是按页映射，要求 `size` 必须是页面大小的整数倍。同时为相关页表项的权限为设置为 `perm`。这个函数调用了 `boot_pgdir_walk`，它也是在内核启动过程中使用的，前面我们提到过它为 `Page` 结构体和 `Env` 结构体进行了映射。

Exercise 2.7 完成 `boot_map_segment` 函数。

一种供参考的框架如下：

```
void boot_map_segment(Pde *pgdir,
                      u_long va, u_long size,
                      u_long pa, int perm) {
    int i;
    Pte *pgtable_entry;
    for (i = 0, size = ROUND(size, BY2PG); i < size; i += BY2PG) {
        /* Step 1. use `boot_pgdir_walk` to "walk" the page directory */
        pgtable_entry = boot_pgdir_walk(
            pgdir,
            va + i,
            1 /* create if entry of page directory not exists yet */
        );
        /* Step 2. fill in the page table */
        *pgtable_entry = (/* III. Physical Frame Address of `pa + i` */)
                         | perm | PTE_V;
    }
}
```

提示：需要填写的部分为物理页框号 `pa + i`，不需要使用宏进行转换。 ■

Thinking 2.4 请你寻找上述两个 `boot_*` 函数在何处被调用。 ■

运行时的二级页表检索函数

`int pgdir_walk(Pde *pgdir, u_long va, int create, Pte **ppte)`，这个函数是 `boot_pgdir_walk` 的非启动版本，只不过后者是在内核启动过程，不允许失败，而该函数是在普通运行过程，如果空间不够允许失败，因此将返回值变为 `int`，若为 0 代表执行成功否则为一个失败码，同时将原本的返回值 `Pte *` 放到 `ppte` 所指的空间上。此外，因为该函数的调用在启动之后，`create` 不为 0 且对应的二级页表不存在时，它使用 `page_alloc` 而不使用 `alloc` 进行物理页面的分配。

Exercise 2.8 完成 `pgdir_walk` 函数。

一种供参考的框架如下：

```c
int pgdir_walk(Pde *pgdir, u_long va, int create, Pte **ppte) {
    Pde *pgdir_entry = pgdir + PDX(va);
    struct Page *page;
    int ret;

    // check whether the page table exists
    if ((*pgdir_entry & PTE_V) == 0) {
        if (create) {
            if ((ret = page_alloc(&page)) < 0) return ret;
            *pgdir_entry = (/* Physical Address of `page` */)
                           | PTE_V | PTE_R;
        } else {
            *ppte = 0;
            return 0;
        }
    }
    *ppte = ((Pte *)(/*Kernel Virtual Address of PTBase */)) + PTX(va);
    return 0;
}
```

对框架中两处代码填空的提示：

1. 使用宏 `page2pa`。
2. 使用宏 `KADDR` 和 `PTE_ADDR`。
3. 注意，这里可能会在页目录表项无效且 `create` 为真时，使用 `page_alloc` 创建一个一级页表，此时应维护申请得到的物理页的 `pp_ref` 字段。

■

增加地址映射函数

`int page_insert(Pde *pgdir, struct Page *pp, u_long va, u_int perm)`，这个函数的作用是将一级页表基地址 `pgdir` 对应的两级页表结构中 `va` 这一虚拟地址映射到内存控制块 `pp` 对应的物理页面，并将页表项权限为设置为 `perm`。

Exercise 2.9 完成 `page_insert` 函数。

一种供参考的框架如下：

```c
int page_insert(Pde *pgdir, struct Page *pp, u_long va, u_int perm) {
    Pte *pgtable_entry;
    int ret;

    perm = perm | PTE_V;

    // Step 0. check whether `va` is already mapping to `pa`
    pgdir_walk(pgdir, va, 0 /* for check */, &pgtable_entry);
    if (pgtable_entry != 0 && (*pgtable_entry & PTE_V) != 0) {
        // check whether `va` is mapping to another physical frame
        if (pa2page(*pgtable_entry) != pp) {
            page_remove(pgdir, va); // unmap it!
        } else {
            tlb_invalidate(pgdir, va);              // <~~
            *pgtable_entry = page2pa(pp) | perm;    // update the permission
            return 0;
        }
    }
    tlb_invalidate(pgdir, va);                      // <~~
    /* Step 1. use `pgdir_walk` to "walk" the page directory */
    if ((ret = pgdir_walk(pgdir, va, 1, &pgtable_entry)) < 0)
        return ret; // exception
    /* Step 2. fill in the page table */
    *pgtable_entry = (/*Physical Frame Address of `pp` */) | perm;
    pp->pp_ref++;
    return 0;
}
```

提示：使用宏 page2pa。

寻找映射的物理地址函数

`struct Page * page_lookup(Pde *pgdir, u_long va, Pte **ppte)`，它的作用是返回一级页表基地址 `pgdir` 对应的两级页表结构中 `va` 这一虚拟地址映射对应的物理页面对应的内存控制块，同时将 `ppte` 指向的空间设为对应的二级页表项地址。

取消地址映射函数

`void page_remove(Pde *pgdir, u_long va)`，它的作用是删除一级页表基地址 `pgdir` 对应的两级页表结构中 `va` 这一虚拟地址对物理地址的映射，如果存在这样的映射，那么对应物理页面的引用次数会减少。

被动地址映射函数

`void pageout(int va, int context)`，一级页表基地址 `context` 对应的两级页表结构中 `va` 新增这一虚拟地址的映射，对应的物理页面通过函数 `page_alloc` 获取而不特殊指定，这对应其「被动」的性质。该函数的具体调用流程将在下一部分进行介绍。

2.6 访存与 TLB 重填

2.6.1 TLB 相关的前置知识

计组理论课中，同学们学习了 TLB 的组成与功能，但没有深入了解 TLB 的填充方法。下面将先介绍一些 R3000 的体系结构知识。

内存管理相关的 CP0 寄存器：

寄存器序号	寄存器名	用途
8	BadVaddr	保存引发地址异常的虚拟地址
10、2	EntryHi、EntryLo	所有读写 TLB 的操作都要通过这两个寄存器，详情请看下一小节
0	Index	TLB 读写相关需要用到该寄存器
1	Random	随机填写 TLB 表项时需要用到该寄存器

TLB 是一个硬件

每一个 TLB 表项都有 64 位，其中高 32 位是 Key，低 32 位是 Data。

EntryHi、EntryLo

EntryHi、EntryLo 都是 CP0 中的寄存器，**他们只是分别对应到 TLB 的 Key 与 Data，并不是 TLB 本身**。

EntryHi、EntryLo 的位结构如下：

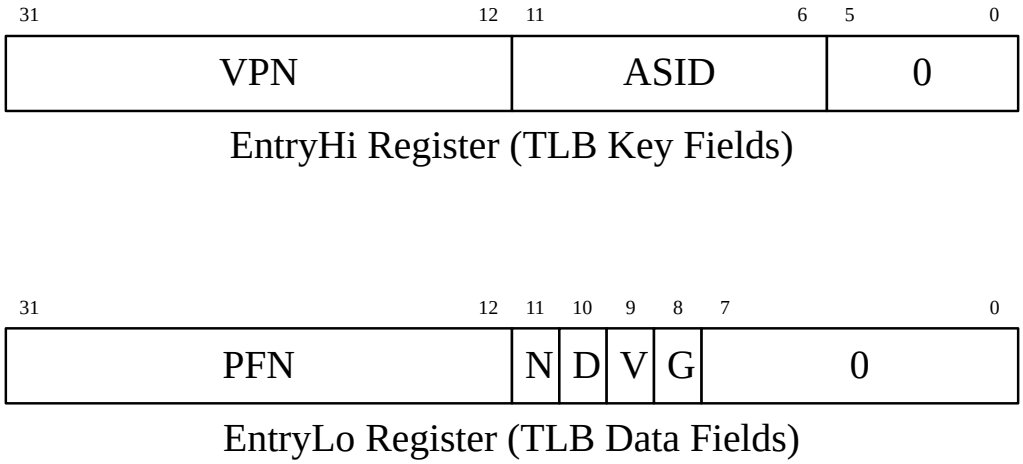


图 2.5: EntryHi & EntryLo

- Key（EntryHi）：

  - VPN：Virtual Page Number
    - **当 TLB 缺失**（CPU 发出虚拟地址，欲在 TLB 中查找物理地址但未查到）时，**EntryHi 中的 VPN 自动（由硬件）填充为对应虚拟地址的虚页号**。
    - 当需要填充或检索 TLB 表项时，软件需要将 VPN 段填充为对应的虚拟地址。
  - ASID：Address Space IDentifier
    - 用于区分不同的地址空间。**查找 TLB 表项时，除了需要提供 VPN，还需要提供 ASID**（同一虚拟地址在不同的地址空间中通常映射到不同的物理地址）。

- Data（EntryLo）：
  - PFN：Physical Frame Number
    - **软件通过填写 PFN，随后使用 TLB 写指令，才将此时的 Key 与 Data 写入 TLB 中**。
  - N：Non-cachable。当该位置高时，后续的物理地址访存将不通过 cache。
  - D：Dirty。事实上是**可写位**。当该位置高时，该地址可写；否则**任何写操作**都将引发 TLB 异常。
  - V：Valid。如果该位为低，则**任何访问该地址**的操作都将引发 TLB 异常。
  - G：Global。

Note 2.6.1 TLB 事实上构建了一个映射 $< \text{VPN}, \text{ASID} > \xrightarrow{\text{TLB}} < \text{PFN}, \text{N}, \text{D}, \text{V}, \text{G} >$。

Thinking 2.5 请你思考下述两个问题：

- 请阅读上面有关 R3000-TLB 的叙述，从虚拟内存的实现角度，阐述 ASID 的必要性
- 请阅读《IDT R30xx Family Software Reference Manual》的 Chapter 6，结合 ASID 段的位数，说明 R3000 中可容纳不同的地址空间的最大数量

■

TLB 相关指令

- `tlbr`：以 Index 寄存器中的值为索引，读出 TLB 中对应的表项到 EntryHi 与 EntryLo。
- `tlbwi`：以 Index 寄存器中的值为索引，将此时 EntryHi 与 EntryLo 的值写到索引指定的 TLB 表项中。

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

（内核）软件操作 TLB 的流程

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

1. 填写 CP0 寄存器。
2. 使用 TLB 相关指令。

2.6.2 TLB 重填

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

如上文所述，我们所使用的 MIPS-R3000 的 MMU 硬件中只有 TLB，在用户地址空间访存时，虚拟地址到物理地址的转化均通过 TLB 进行，即只有当当前虚拟地址对应虚拟页号在 TLB 中时，才能找到对应的物理地址；因此访存时，首先在 TLB 中查询相应的页号，如果能够查询到，则可取得物理地址，如果不能查询到，则产生 TLB 中断，跳转到异常处理程序中，对 TLB 进行重填。

TLB 的重填过程由 `do_refill` 函数（lib/genex.S）完成，相关流程我们在介绍两级页表时已经有所了解，但当时并没有涉及 TLB 部分，加入 TLB 后，整个流程大致如下:

1. **确定此时的一级页表基地址**: `mCONTEXT` 中存储了当前进程一级页表基地址位于 `kseg0` 的虚拟地址;

   > **Note 2.6.2** 通过自映射相关知识, 可以计算出对于每个进程而言, `0x7fdff000` 这一虚拟地址也同样映射到该进程的一级页表基地址，但是重填时处于内核态, 如果使用 `0x7fdff000` 则还需要额外确定当前属于哪一个进程，使用位于 `kseg0` 的虚拟地址可以通过映射规则直接确定物理地址。

2. 从 BadVaddr 中取出引发 TLB 缺失的虚拟地址, 并确定其对应的一级页表偏移量 (高 10 位);

3. 根据一级页表偏移量, 从一级页表中取出对应的表项：此时取出的表项由**二级页表基地址的物理地址与权限位组成**;

4. 判定权限位: 若权限位显示该表项无效（无 `PTE_V` ）, 则调用 `page_out` ，随后回到第一步;

5. 确定引发 TLB 缺失的虚拟地址对应的二级页表偏移量（中间 10 位），与先前取得的二级页表基地址的物理地址共同确认二级页表项的物理地址;

6. **将二级页表项物理地址转为位于 `kseg0` 的虚拟地址**（高位补 `1`），随后页表中取出对应的表项：**此时取出的表项由物理地址与权限位组成**;

7. 判定权限位: 若权限位显示该表项无效（无 `PTE_V`），则调用 `page_out` ，随后回到第一步;（`PTE_COW` 为写时复制权限位，将在 lab4 中用到，此时将所有页表项该位视为 0 即可）

8. 将物理地址存入 EntryLo , 并**调用 `tlbwr` 将此时的 EntryHi 与 EntryLo 写入到 TLB 中**（EntryHi 中保留了虚拟地址相关信息）。

其中第 4 步与第 7 步均可能调用 **`page_out` 函数**（mm/pmap.c），**处理页表中找不到对应表项的异常**, 流程大致如下:

1. 若 TLB 缺失时，`mCONTEXT` 不位于 `kseg0`, 则说明系统出现了故障。

2. 若引发 TLB 缺失的虚拟地址不合法（如访问了过低或过高的地址），则说明系统故障。

3. 否则可以为此虚拟地址申请一个物理页面（`page_alloc`），并将虚拟地址映射到该物理页面（`page_insert`）。(被动扩充内存)

Note 2.6.3 这里的两个函数都只进行了大致的解释, 详细的实现可以参阅源代码

经过以上分析，`tlb_invalidate` 的重要性更得以体现——如果页表内容变化而 TLB 未更新，则可能访问到错误的物理页面。同时中断、异常处理等过程将是之后实验的重点，现在可暂且将其理解为代码的跳转，此处只需明白 R3000 中代码在访问内存时的处理过程即可。

Exercise 2.10 完成 `tlb_out` 函数。该函数根据传入的参数（TLB 的 Key）找到对应的 TLB 表项，并将其清空。

具体来说，你需要在两个位置插入两条指令，其中一个位置为 `tlbp`，另一个位置为 `tlbwi`。

因流水线设计架构原因，`tlbp` 指令的前后都应各插入一个 `nop` 以解决数据冒险。 ■

Thinking 2.6 请你思考下述两个问题:

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

■

2.6.3 正确结果展示

在完成虚拟内存管理后,在 init/init.c 中,注释掉 `physical_memory_manage_check()`, 并保证 `page_check()` 没有被注释。然后编译运行内核，显示如下信息即通过虚拟内存管理的单元测试:

```
main.c: main is start ...
init.c: mips_init() is called
Physical memory: 65536K available, base = 65536K, extended = 0K
to memory 80401000 for struct page directory.
to memory 80431000 for struct Pages.
pmap.c:  mips vm init success
va2pa(boot_pgdir, 0x0) is 3ffe000
page2pa(pp1) is 3ffe000
start page_insert
pp2->pp_ref 0
end page_insert
page_check() succeeded!
panic at init.c:42: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

地址 `3ffe000` 和最后一行的数字 `42` 是不固定的。

注意，此处的单元测试仅针对虚拟内存管理。由于还未建立中断异常处理机制，所以没有对 TLB 部分进行测试，同学们可以在 Lab3 时再检验此处的正确性。

2.7 Lab2 在 MOS 中的概况

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

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

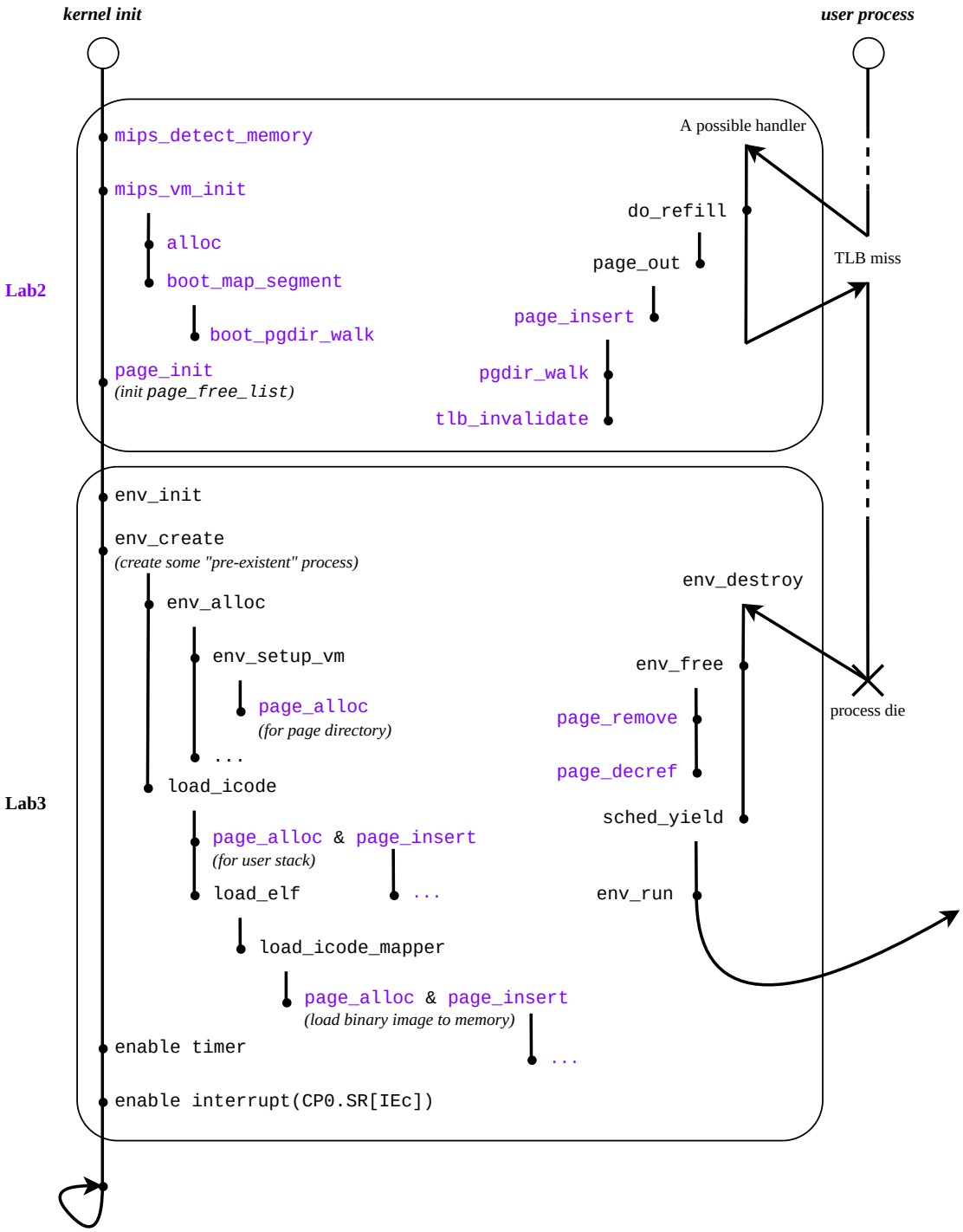


图 2.6: Lab2 in MOS

2.8 多级页表与页目录自映射

2.8.1 MOS 中的页表自映射应用

在 Lab2 中，我们实现了内存管理，建立了两级页表机制。

试想这样一个问题：如果页表和页目录没有被映射到进程的地址空间，而一个进程的 4GB 地址空间均映射物理内存的话，那么就需要 4MB 来存放页表（1024 个页表），4KB 来存放页目录；而在 MOS 中，页表和页目录都在进程的地址空间中得到映射，这意味着在 1024 个页表中，**有一个页表所对应的 4MB 空间就是这 1024 个页表占用的 4MB 空间**。这一个特殊的页表就是页目录，它的 1024 个表项映射到这 1024 个页表。因此只需要 4MB 的空间即可容纳页表和页目录。

而 MOS 中，将页表和页目录映射到了用户空间中的 `0x7fc00000-0x80000000`（共 4MB）区域，这意味着 MOS 中允许在用户态下访问当前进程的页表和页目录，这一特性将在后续的 Lab 中用到。

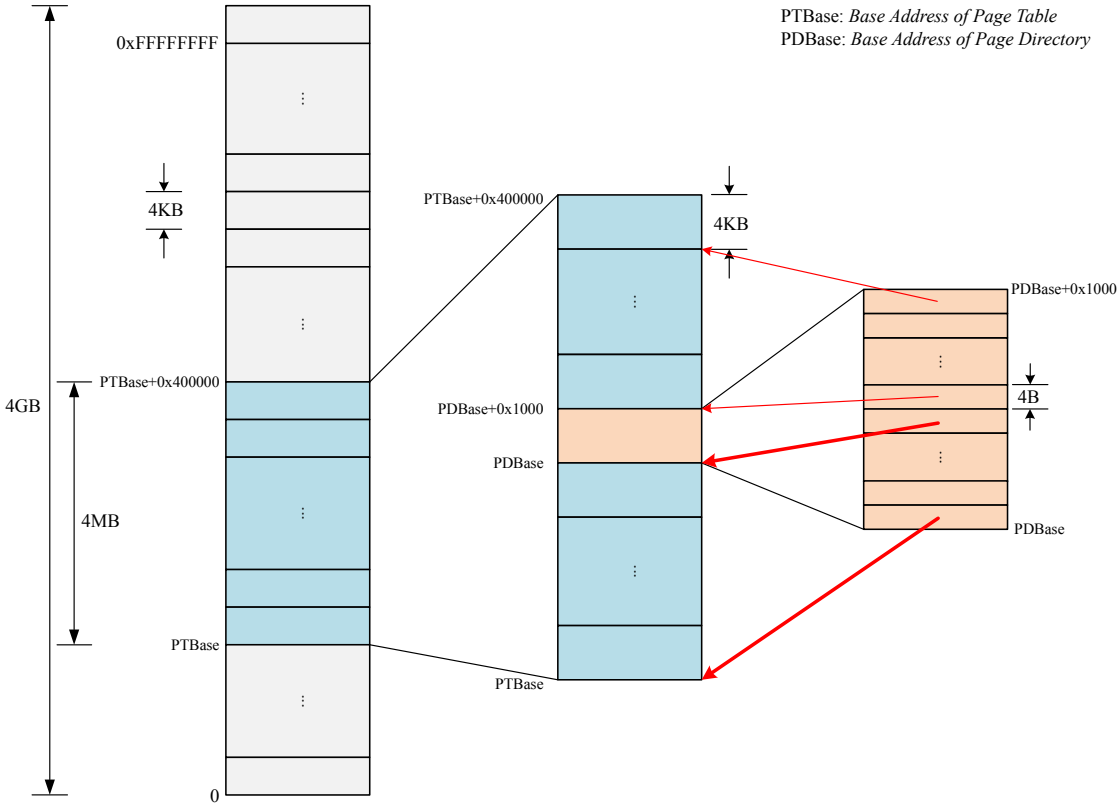


图 2.7: 页目录自映射

下面我们根据页目录自映射的性质，求出 MOS 中页目录的基地址：

`0x7fc00000-0x80000000` 这 4MB 空间的起始位置（也就是第一个二级页表的基地址）对应着页目录的第一个页目录项。同时由于 1M 个页表项和 4GB 地址空间是线性映射的，不难算出 `0x7fc00000` 这一个地址对应的应该是第 `0x7fc00000 >> 12` 个页表项（这一个页表项也就是第一个页目录项）。由于一个页表项占 4B 空间，因此第 `0x7fc00000 >> 12` 个页表项相对于页表基地址的偏移为 `(0x7fc00000 >> 12) * 4`，即 `0x1ff000`。

最终即可得到页目录基地址为 `0x7fdff000`。

2.8.2 其他页表机制

在其他系统中，还会使用三级页表等更多级的页表机制。请同学们结合理论课所学，查阅相关资料，回答下述思考题。

Thinking 2.7 在现代的 64 位系统中，提供了 64 位的字长，但实际上不是 64 位页式存储系统。假设在 64 位系统中采用三级页表机制，页面大小 4KB。由于 64 位系统中字长为 8B，且页目录也占用一页，因此页目录中有 512 个页目录项，因此每级页表都需要 9 位。因此在 64 位系统下，总共需要 $3 \times 9 + 12 = 39$ 位就可以实现三级页表机制，并不需要 64 位。

现考虑上述 39 位的三级页式存储系统，虚拟地址空间为 512 GB，若记三级页表的基地址为 $\mathrm{PT}_{\mathrm{base}}$，请你计算：

- 三级页表页目录的基地址
- 映射到页目录自身的页目录项（自映射）

■

2.9 其他体系结构中的内存管理

本实验在 MIPS 体系结构上构建操作系统，因此内存管理机制与 MIPS 特性相耦合。在其他的一些体系结构中，内存管理可能会有很大的差别：有一些体系结构直接使用硬件机制来填写 TLB，而不是本实验中依赖软件 `do_refill` 函数来填写 TLB。

Thinking 2.8 任选下述二者之一回答：

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

■

2.10 任务列表

- Exercise-完成 `mips_detect_memory` 函数
- Exercise-完成 `queue.h`
- Exercise-完成 `page_init` 函数
- Exercise-完成 `page_alloc` 函数

- Exercise-完成 page_free 函数
- Exercise-完成 boot_pgdir_walk 函数
- Exercise-实现 boot_map_segment 函数
- Exercise-完成 pgdir_walk 函数
- Exercise-完成 page_insert 函数
- Exercise-完成 tlb_out 函数

2.11 实验思考

- 思考-程序代码中的地址
- 思考-链表宏
- 思考-Page_list 结构体
- 思考-两个 boot_* 的调用点
- 思考-地址空间
- 思考-tlb_invalidate
- 思考-三级页表自映射
- 思考-其他体系结构的内存管理

CHAPTER 3

进程与异常

3.1 实验目的

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

在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，并建立一个简单的用户进程，加载一个程序镜像到指定的内存空间，然后让它运行起来。

同时，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。每个进程都是一个实体，有其自己的地址空间，通常包括代码段、数据段和堆栈。程序是一个没有生命的实体，只有被处理器赋予生命时，它才能成为一个活动的实体，而执行中的程序，就是我们所说的进程。

3.2.1 进程控制块

进程控制块 (PCB) 是系统专门设置用来管理进程的数据结构，它可以记录进程的外部特征，描述进程的运动变化过程。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 8: 进程控制块

```
struct Env {
    struct Trapframe env_tf;        // Saved registers
    LIST_ENTRY(Env) env_link;       // Free LIST_ENTRY
    u_int env_id;                   // Unique environment identifier
    u_int env_parent_id;            // env_id of this env's parent
    u_int env_status;               // Status of the environment
    Pde  *env_pgdir;                // Kernel virtual address of page dir
    u_int env_cr3;
    LIST_ENTRY(Env) env_sched_link;
    u_int env_pri;
};
```

为了集中注意力在关键的地方，我们暂时不对后续实验用到的域做介绍。下面是 **lab3** 相关部分的简单说明：

- env_tf：Trapframe 结构体的定义在 **include/trap.h** 中，在发生进程调度，或当陷入内核时，会将当时的进程上下文环境保存在 env_tf 变量中。
- env_link：env_link 的机制类似于 lab2 中的 pp_link，使用它和 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，它是进程独一无二的标识符。
- env_parent_id：在之后的实验中，我们将了解到进程是可以被其他进程创建的，创建本进程的进程称为父进程。此变量记录父进程的进程 id，进程之间通过此关联可以形成一棵进程树。
- env_status：该变量只能有以下三种取值：
  - 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，处于该状态的进程需要在一定条件下变成就绪状态从而被 CPU 调度。（比如因进程通信阻塞时变为 ENV_NOT_RUNNABLE，收到信息后变回 ENV_RUNNABLE）
  - ENV_RUNNABLE：该进程处于**执行状态或就绪状态**，即其可能是正在运行的，也可能正在等待被调度。
- env_pgdir：这个变量保存了该进程页目录的内核虚拟地址。
- env_cr3：这个变量保存了该进程页目录的物理地址。
- env_sched_link：这个变量用来构造调度队列。
- env_pri：这个变量保存了该进程的优先级。

在我们的实验中，存放进程控制块的物理内存在系统启动后就要被分配好，也就是 **envs** 数组。

我们已经实现了这部分工作，它位于 mm/pmap.c 中的 mips_vm_init 函数中，但你仍需理解其实现过程。

Exercise 3.1 完成下述任务：

- 阅读 mips_vm_init 函数中为 envs 数组分配空间部分的代码。
- envs 数组包含 NENV 个 Env 结构体成员，可以参考 pmap.c 中的 **pages** 数组空间的分配方式。
- 除了要为数组 envs 分配空间外，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（参考 **./include/mmu.h** ）。

注：本次实验**不需要**填写这个部分，但是请仔细阅读相关代码。 ■

当然，有了存储进程控制块信息的 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 lib/env.c 中的 env_init 函数，注意链表插入的顺序，顺序要求已在注释中给出。 ■

Note 3.2.2 这里我们规定链表的插入顺序，是为了方便进行单元测试，再次提醒不要自由发挥。

3.2.2 进程的标识

你的电脑中经常有很多进程同时存在，每个进程执行不同的任务，它们之间也经常需要相互协作、通信，那操作系统是如何识别每个进程呢？

在上一部分中，我们提到了 struct Env 进程控制块中的 env_id 这个域，它是每个进程独一无二的标识符，需要在进程创建的时候就被赋予。

你可以在 env.c 文件中找到一个叫做 mkenvid 的函数，它的作用就是生成一个新的进程 id。

你应该已经注意到 mkenvid 中调用的 asid_alloc 函数，这个函数的作用是为新创建的进程分配一个异于当前所有未被释放的进程的 ASID。或许你会有所疑问，这个 ASID 是什么？为什么要与其他的进程不同呢？为什么不能简单的通过自增来避免冲突呢？

要解答这些问题，还需回到 TLB 的讨论：

Note 3.2.3 根据 lab2 的学习我们得知进程是通过页表来访问内存的，而不同的进程的同一个虚拟地址可能会映射到不同的物理地址。

为了实现这个功能，TLB 中除了存储页表的映射信息之外，还会存储进程的标识编号，作为 Key 的一部分，用于保证查到的页面映射属于当前进程，而这个编号就是 ASID。显然，不同进程的虚拟地址是可以对应相同 VPN 的，而如果 ASID 也不具备唯

一标识性，就与 TLB Field 的唯一性要求相矛盾了。因此，直到进程被销毁或 TLB 被清空时，才可以把这个 ASID 分配给其他进程。

到此，前两个问题就得到了解答，而为了解答第三个问题，我们需要更加深入地了解 TLB 的结构。我们采用的模拟器模拟的 CPU 型号是 MIPS R3000，其 TLB 结构的 Key Fields（也就是 EntryHi 寄存器）中应用到了 ASID：

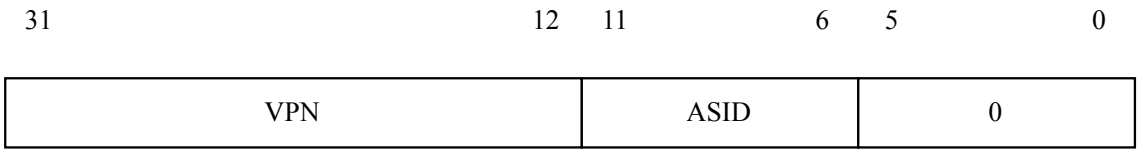


图 3.1: TLB 对 ASID 的规定

可以发现，其中 ASID 部分只占据了 6-11 共 6 个 bit，所以如果单纯的通过自增的方式来分配 ASID 的话，很快就会发生溢出，导致 ASID 重复。(事实上，在 MOS 系统的早期版本我们就采用了这种简易的实现策略，这导致系统在创建进程过多时死机)

为了解决这个问题，我们采用限制同时运行的进程个数的方法来防止 ASID 重复。具体实现是通过位图法管理可用的 64 个 ASID，如果当 ASID 耗尽时仍要创建进程，系统会 panic。同学们可以参考 env.c 中 asid_alloc 函数的实现来理解。

Note 3.2.4 实际的 Linux 系统中通过另一种方式解决了这个问题，使得同时运行的进程数不受到 ASID 个数的限制，感兴趣的同学可以在课下了解一下 Linux 的实现机制。

如果我们现在已经知道一个进程的 id，那么如何才能找到该 id 对应的进程控制块呢？

Exercise 3.3 仔细阅读注释，完成 lib/env.c 中的 envid2env 函数，实现通过一个 env 的 id 获取该 id 对应的进程控制块的功能。 ■

Thinking 3.1 思考 envid2env 函数：为什么 envid2env 中需要判断 e->env_id != envid 的情况？如果没有这步判断会发生什么情况？ ■

3.2.3 设置进程控制块

做完上面那个小练习后，我们就可以开始利用**空闲进程链表 env_free_list** 创建进程了。下面我们就来具体讲讲你应该如何创建一个进程[1]。

进程创建的流程如下：

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

[1]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。

第二步“纯手工打造”打造一个进程。在这种创建方式下，由于没有模板进程，所以进程拥有的所有信息都是手工设置的。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 的信息还没法跑起来，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 此时 PCB 已经被填写了很多东西，不再是一张白纸，把它从空闲链表里摘出，就可以投入使用了。

当然，第二步的信息设置是本次实验的关键，那么下面让我们来结合注释看看这段代码

Listing 9: 进程创建

```
/* Overview:
 *  Allocate and Initialize a new environment.
 *  On success, the new environment is stored in *new.
 *
 * Pre-Condition:
 *  If the new Env doesn't have parent, parent_id should be zero.
 *  env_init has been called before this function.
 *
 * Post-Condition:
 *  return 0 on success, and set appropriate values of the new Env.
 *  return -E_NO_FREE_ENV on error, if no free env.
 *
 * Hints:
 *  You may use these functions and macro definitions:
 *      LIST_FIRST,LIST_REMOVE, mkenvid (Not All)
 *  You should set some states of Env:
 *      id , status , the sp register, CPU status , parent_id
 *      (the value of PC should NOT be set in env_alloc)
 */
/*** exercise 3.5 ***/
int
env_alloc(struct Env **new, u_int parent_id)
{
    int r;
    struct Env *e;

    /* Step 1: Get a new Env from env_free_list*/


    /* Step 2: Call a certain function (has been completed just now)
         *   to init kernel memory layout for this new Env.
     * The function mainly maps the kernel address to this new Env address. */


    /* Step 3: Initialize every field of new Env with appropriate values.*/


```

```
38      /* Step 4: Focus on initializing the sp register
39           *   and cp0_status of env_tf field, located at this new Env. */
40      e->env_tf.cp0_status = 0x10001004;
41
42
43      /* Step 5: Remove the new Env from env_free_list. */
44
45
46  }
```

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

Listing 10: 地址空间初始化

```
1   /* Overview:
2    *  Initialize the kernel virtual memory layout for 'e'.
3    *  Allocate a page directory, set e->env_pgdir and e->env_cr3 accordingly,
4    *    and initialize the kernel portion of the new env's address space.
5    *  DO NOT map anything into the user portion of the env's virtual address space.
6    */
7   /*** exercise 3.4 ***/
8   static int
9   env_setup_vm(struct Env *e)
10  {
11
12      int i, r;
13      struct Page *p = NULL;
14      Pde *pgdir;
15
16      /* Step 1: Allocate a page for the page directory
17       *   using a function you completed in the lab2 and add its pp_ref.
18       *   pgdir is the page directory of Env e, assign value for it. */
19      if (      ) {
20          panic("env_setup_vm - page alloc error\n");
21          return r;
22      }
23
24      /* Step 2: Zero pgdir's field before UTOP. */
25
26
27      /* Step 3: Copy kernel's boot_pgdir to pgdir. */
28
29      /* Hint:
30       *  The VA space of all envs is identical above UTOP
31       *  (except at UVPT, which we've set below).
32       *  See ./include/mmu.h for layout.
33       *  Can you use boot_pgdir as a template?
34       */
35
36
37      /* UVPT maps the env's own page table, with read-only permission.*/
38      e->env_pgdir[PDX(UVPT)]  = e->env_cr3 | PTE_V;
39      return 0;
40  }
```

在你动手开始完成 env_setup_vm 之前，为了便于理解你需要在这个函数中所做的

事情，请先阅读以下提示:

在我们的实验中，虚拟地址 ULIM 以上的地方，kseg0 和 kseg1 两部分内存的访问不经过 TLB，它们不归属于某一个进程，而是由内核管理的。在这里，操作系统将一些内核的数据暴露到用户空间，使得进程不需要切换到内核态就能访问，这也是 MOS 的微内核设计。我们将在 lab4 和 lab6 中用到此机制。而这里我们要暴露的空间是 UTOP 以上 ULIM 以下的部分，也就是把这部分内存对应的内核页表拷贝到进程页表中。

Exercise 3.4 仔细阅读注释，填写 env_setup_vm 函数 ■

Thinking 3.2 结合 include/mmu.h 中的地址空间布局，思考 env_setup_vm 函数:

- UTOP 和 ULIM 的含义分别是什么，UTOP 和 ULIM 之间的区域与 UTOP 以下的区域相比有什么区别?
- 请结合系统自映射机制解释代码中`pgdir[PDX(UVPT)]=env_cr3`的含义。
- 谈谈自己对进程中物理地址和虚拟地址的理解。

■

在上述的思考完成后，那么我们就可以直接在 **env_alloc** 第二步使用该函数了。现在来解决一下刚才的问题，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，因此我们必须直接在代码中给出。下面介绍它的具体作用。

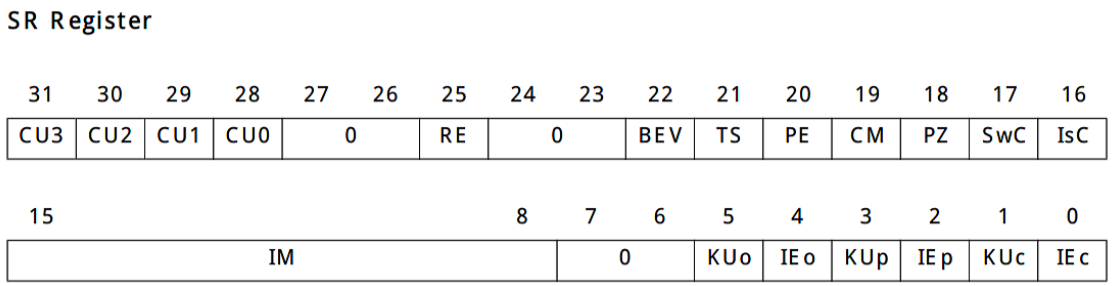


图 3.2: R3000 的 SR 寄存器示意图

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，表示允许在用户模式下使用 CP0 寄存器。

第 12bit 设置为 1，表示 4 号中断可以被响应。

下面的内容更加重要:

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[2]。

而每当 rfe 指令调用的时候，就会进行上面操作的**逆操作**。我们现在先不管为何，但是已经知道，下面这一段代码 (位于 lib/env_asm.S 中) **是每个进程在每一次被调度时都会执行的**，所以就一定会执行`rfe`这条指令。

```
lw   k0,TF_STATUS(k0)        # 恢复 CP0_STATUS 寄存器
mtc0 k0,CP0_STATUS
j    k1
rfe
```

现在你可能就懂了为何我们 status 后六位是设置为`000100`了。当运行进程前，运行上述代码到`rfe`的时候 (rfe 处于延迟槽中)，就会将 KUp 和 IEp 拷贝回 KUc 和 IEc，令 status 为 `000001`，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了。

> **Note 3.2.5** 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.

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 3.5** 根据上面的提示与代码注释，填写 **env_ alloc** 函数。 ■

3.2.4 加载二进制镜像

在进程创建第三点曾提到，我们需要为**新进程的程序**分配空间来容纳程序代码。那么下面我需要有两个函数来一起完成这个任务

Listing 11: 加载镜像映射

```
/* Overview:
 *   This is a call back function for kernel's elf loader.
 * Elf loader extracts each segment of the given binary image.
 * Then the loader calls this function to map each segment
 * at correct virtual address.
 *
```

[2]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。

```
 *   `bin_size` is the size of `bin`. `sgsize` is the
 * segment size in memory.
 *
 * Pre-Condition:
 *   bin can't be NULL.
 *   Hint: va may be NOT aligned with 4KB.
 *
 * Post-Condition:
 *   return 0 on success, otherwise < 0.
 */
/*** exercise 3.6 ***/
static int load_icode_mapper(u_long va, u_int32_t sgsize,
       u_char *bin, u_int32_t bin_size, void *user_data)
{
    struct Env *env = (struct Env *)user_data;
    struct Page *p = NULL;
    u_long i;
    int r;
    u_long offset = va - ROUNDDOWN(va, BY2PG);

    /* Step 1: load all content of bin into memory. */
    for (i = 0; i < bin_size; i += BY2PG) {
        /* Hint: You should alloc a new page. */
    }
    /* Step 2: alloc pages to reach `sgsize` when `bin_size` < `sgsize`.
     * hint: variable `i` has the value of `bin_size` now! */
    while (i < sgsize) {


    }
    return 0;
}
```

在 lab1 中我们曾详细学习了 ELF 文件，它的出现场合有两种，一是组成可重定位文件，二是组成可执行文件或可被共享的对象文件。在这里 ELF 文件以后者的形式出现，用于在内存中构建进程映像。

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。我们已经写好了用于解析 ELF 文件的代码中的大部分内容 (位于 lib/kernel_elfloader.c 中)，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：

```
// binary 为整个待加载的 ELF 文件。size 为该 ELF 文件的大小。
// 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
int load_elf(u_char *binary, int size, u_long *entry_point, void *user_data,
             int (*map)(u_long va, u_int32_t sgsize,
                        u_char *bin, u_int32_t bin_size, void *user_data))
```

Note 3.2.6 在 lab3 阶段，我们还没有实现文件系统，因此无法直接操作磁盘中的 ELF 文件。在这里我们已经将 ELF 文件内容转化成了 C 数组的形式 (如 init/code_a.c 文件)，这样可以通过编译到内核中完成加载。

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，并由它完成单个 segment 的加载过程。

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11所示。load_elf() 函数会从 ELF 文件文件中解析出每个 segment 的四个信息：va(该段需要被加载到的虚地址)、sgsize(该段在内存中的大小)、bin(该段在 ELF 文件中的起始位置)、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为填入这段内容新分配的页面大小，即 alloc 了新的页表但没能填满，那么余下的部分用 0 来填充。

这个函数是至关重要的，它很可能成为未来大量 TOO LOW 的元凶。因此建议你静下心来，认真考虑对齐的问题，保证在应有位置完成 ELF 加载后既不会破坏原有页面也不会引入新的多余内容。在这里，我们给出一点提示，下图展示的是“最糟糕”的情况。

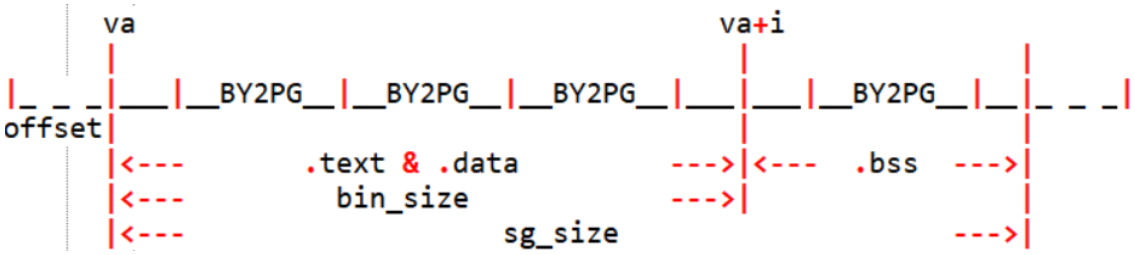


图 3.3: 每个 segment 的加载地址布局

load_elf() 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。但你会发现，我们并没有真正解释其中 user_data 这个参数的作用。这样设计又是为了什么呢？这个问题我们决定留给你来思考。

Thinking 3.3 找到 user_data 这一参数的来源，思考它的作用。没有这个参数可不可以？为什么？(可以尝试说明实际的应用场景，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 3.6 通过上面补充的知识与注释，填充 **load_icode_mapper** 函数。 ■

Thinking 3.4 结合 load_icode_mapper 的参数以及二进制镜像的大小，考虑该函数可能会面临哪几种复制的情况？你是否都考虑到了？ ■

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 12: 完整加载镜像

```
/* Overview:
 *  Sets up the the initial stack and program binary for a user process.
 *  This function loads the complete binary image by using elf loader,
 *  into the environment's user memory. The entry point of the binary image
 *  is given by the elf loader. And this function maps one page for the
 *  program's initial stack at virtual address USTACKTOP - BY2PG.
 *
 * Hints:
 *  All mapping permissions are read/write including text segment.
 *  You may use these :
 *      page_alloc, page_insert, page2kva , e->env_pgdir and load_elf.
 */
/*** exercise 3.7 ***/
static void
load_icode(struct Env *e, u_char *binary, u_int size)
{
    /* Hint:
     *  You must figure out which permissions you'll need
     *  for the different mappings you create.
     *  Remember that the binary image is an a.out format image,
     *  which contains both text and data.
     */
    struct Page *p = NULL;
    u_long entry_point;
    u_long r;
    u_long perm;

    /* Step 1: alloc a page. */


    /* Step 2: Use appropriate perm to set initial stack for new Env. */
    /* Hint: Should the user-stack be writable? */


    /* Step 3: load the binary using elf loader. */


    /* Step 4: Set CPU's PC register as appropriate value. */
    e->env_tf.pc = entry_point;
}
```

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 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，就不再赘述了。

Exercise 3.7 根据 ELF 文件加载内容与注释的步骤提示，填充 **load_elf** 函数和 **load_icode** 函数。 ■

这里的`e->env_tf.pc`是什么呢？这个字段指示了进程要恢复运行时 pc 应恢复到的位置。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 3.5 思考上面这一段话，并根据自己在 **lab2** 中的理解，回答：

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

■

思考完这一点后，下面我们可以真正创建进程了。

3.2.5 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 3.8 根据提示，完成 **env_create** 函数与 **env_create_priority** 函数的填写。 ■

当然提到创建进程，我们还需要提到两个封装好的宏命令

```
#define ENV_CREATE_PRIORITY(x, y) \
{ \
    extern u_char binary_##x##_start[];\
    extern u_int binary_##x##_size; \
    env_create_priority(binary_##x##_start, \
        (u_int)binary_##x##_size, y); \
}
```

```
#define ENV_CREATE(x) \
{ \
    extern u_char binary_##x##_start[];\
    extern u_int binary_##x##_size; \
    env_create(binary_##x##_start, \
        (u_int)binary_##x##_size); \
}
```

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，例如下面这段代码[3]

[3]这个例子出自http://www.cnblogs.com/hnrainll/archive/2012/08/15/2640558.html，想深入了解的同学可以参考这篇博客

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

最后，你需要在init/init.c中增加下面两句代码，来初始化创建两个进程。这里的 user_A 和 user_B 用于变量命名，以 user_A 为例，经过 ENV_CREATE 宏的拼接后，得到 binary_user_A_start 数组和 binary_user_A_size 变量，我们可以在之前提到的 init/code_a.c 文件中可以找到它们的定义。

```c
ENV_CREATE_PRIORITY(user_A, 2);
ENV_CREATE_PRIORITY(user_B, 1);
```

Exercise 3.9 根据注释与理解，将上述两条进程创建命令加入 **init/init.c** 中。 ■

3.2.6 进程运行与切换

Listing 13: 进程的运行

```c
extern void env_pop_tf(struct Trapframe *tf, int id);
extern void lcontext(u_int contxt);

/* Overview:
 *  Restore the register values in the Trapframe with env_pop_tf,
 *  and switch the context from 'curenv' to 'e'.
 *
 * Post-Condition:
 *  Set 'e' as the curenv running environment.
 *
 * Hints:
 *  You may use these functions:
 *      env_pop_tf , lcontext.
 */
/*** exercise 3.10 ***/
void
env_run(struct Env *e)
{
    /* Step 1: save register state of curenv. */
    /* Hint: if there is an environment running,
     *   you should switch the context and save the registers.
     *   You can imitate env_destroy() 's behaviors.*/


    /* Step 2: Set 'curenv' to the new environment. */


    /* Step 3: Use lcontext() to switch to its address space. */


    /* Step 4: Use env_pop_tf() to restore the environment's
```

```
32          *   environment   registers and return to user mode.
33          *
34          * Hint: You should use GET_ENV_ASID there. Think why?
35          *   (read <see mips run linux>, page 135-144)
36          */
37
38  }
```

刚刚说到的 load_icode 是为数不多的代码理解难度较大的函数之一，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：

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

> **Note 3.2.7** 进程上下文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，顾名思义，就是当前进程停下工作，让出 CPU 来运行另外的进程。那么要理解进程切换，我们就要知道进程切换时系统需要做些什么。Alt+Tab 可以吗？当然不可以。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？你可能会想到下面两种需要保存的状态：

进程本身的信息

进程周围的环境信息

事实上，进程本身的信息无非就是进程控制块中那些字段，包括 **env_id,env_parent_id,env_pgdir,env_cr3...**

这些在进程切换后还保留在原本的进程控制块中，并不会改变，因此不需要保存。而会变的实际上是进程周围的环境信息，这才是需要保存的内容。也就是 env_tf 中的进程上下文。

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。第一步注释还说到，让我们参考 `env_destroy` ，其实就是把 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。

还有一点很关键，我们需要将 env_tf.pc 设置为上下文的 epc 值。

Thinking 3.6 请查阅相关资料解释，上面提到的 epc 是什么？为什么要将`env_tf.pc`设置为 epc 呢？ ■

总结以上说明，我们不难看出 env_run 的执行流程：

1. 保存当前进程的上下文信息，设置当前进程上下文中的 pc 为 epc。
2. 切换 curenv 为即将运行的进程。
3. 调用 lcontext 函数，设置全局变量 mCONTEXT 为当前进程页目录地址，这个值将在 TLB 重填时用到。
4. 调用 env_pop_tf 函数，恢复现场、异常返回。

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

Thinking 3.7 关于 TIMESTACK，请思考以下问题：

- 操作系统在何时将什么内容存到了 TIMESTACK 区域
- TIMESTACK 和 env_asm.S 中所定义的 KERNEL_SP 的含义有何不同

■

Exercise 3.10 根据注释与讲解，填充完成 **env_run** 函数。 ■

通过填写上述代码，大家应该已经发现，在进程切换时，我们会将寄存器的值保存在 TIMESTACK 对应的页面。但是在 lab2 初始化页表时，这一页也被设置为了“空闲”状态，所以此物理页面可能会被进程占用导致问题。现在需要同学们修改 **page_init** 函数来保证这种问题不会发生。

Exercise 3.11 修改 **page_init** 函数，防止 TIMESTACK 所在页面被分配出去。 ■

至此，第一部分工作已经完成。在 **init/init.c** 中注释掉 **trap_init** 和 **kclock_init** 两个函数，就可以进行此部分单元测试了。

3.2.7 实验正确结果

如果你按流程做完第一部分，并且做的结果正确，gxemul 中会输出以下结果。

```
1 main.c: main is start ...
2
3 init.c: mips_init() is called
4
5 Physical memory: 65536K available, base = 65536K, extended = 0K
6
7 to memory 80401000 for struct page directory.
8
```

```
to memory 80431000 for struct Pages.

pmap.c:  mips vm init success

pe0->env_id 1024

pe1->env_id 3073

pe2->env_id 5122

env_init() work well!

envid2env() work well!

pe1->env_pgdir 83ffe000

pe1->env_cr3 3ffe000

env_setup_vm passed!

pe2`s sp register 7f3fe000

[00000000] free env 00001402

[00000000] free env 00000c01

[00000000] free env 00000400

env_check() succeeded!

text & data segment load right!

bss segment load right!

[00000000] free env 00000400

load_icode_check() succeeded!

panic at init.c:29: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

3.3 中断与异常

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

在这里先对 P7 的知识进行简单的回顾。CPU 不仅仅有我们常见的 32 个通用寄存器，还有功能广泛的协处理器，而中断/异常部分就用到了其中的控制寄存器 CP0。当然，这不是某一个寄存器，而是一组寄存器，这里用到的是编号为 12,13,14 的三个 CP0 寄存器。下面的表格列出了这三个寄存器的具体功能。

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

Note 3.3.1 我们实验里认为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。

SR 寄存器：图3.2(在设置进程控制块部分给出) 是 MIPS R3000 中 Status Register 寄存器，15-8 位为中断屏蔽位，每一位代表一个不同的中断活动，其中 15-10 位使能外部中断源，9-8 位是 Cause 寄存器软件可写的中断位。

Cause 寄存器：图3.4是 MIPS R3000 中 Cause 寄存器。其中保存着 CPU 中哪一些中断或者异常已经发生。15-8 位保存着哪一些中断发生了，其中 15-10 位来自硬件，9-8 位可以由软件写入，当 SR 寄存器中相同位允许中断（为 1）时，Cause 寄存器这一位活动就会导致中断。6-2 位（ExcCode），记录发生了什么异常。

Cause Register

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

图 3.4: Cause 寄存器

在《See MIPS Run Linux》的第五章中介绍到，MIPS CPU 处理一个异常时大致要完成四项工作：

1 设置 EPC 指向异常结束时重新返回的地址。

2 设置 SR 位，强制 CPU 进入内核态（行驶更高级的特权）并禁止中断。

3 设置 Cause 寄存器，用于记录异常发生的原因。

4 CPU 开始从异常入口位置取指，此后一切交给软件处理。

而这句“一切交给软件处理”，就是我们当前任务的开始。
我们可以通过图3.5理解异常的产生与返回过程。

3.3.1 异常的分发

当发生异常时，处理器会进入一个用于分发异常的程序，这个程序的作用就是检测发生了哪种异常，并调用相应的异常处理程序。一般来说，异常分发程序会被要求放在

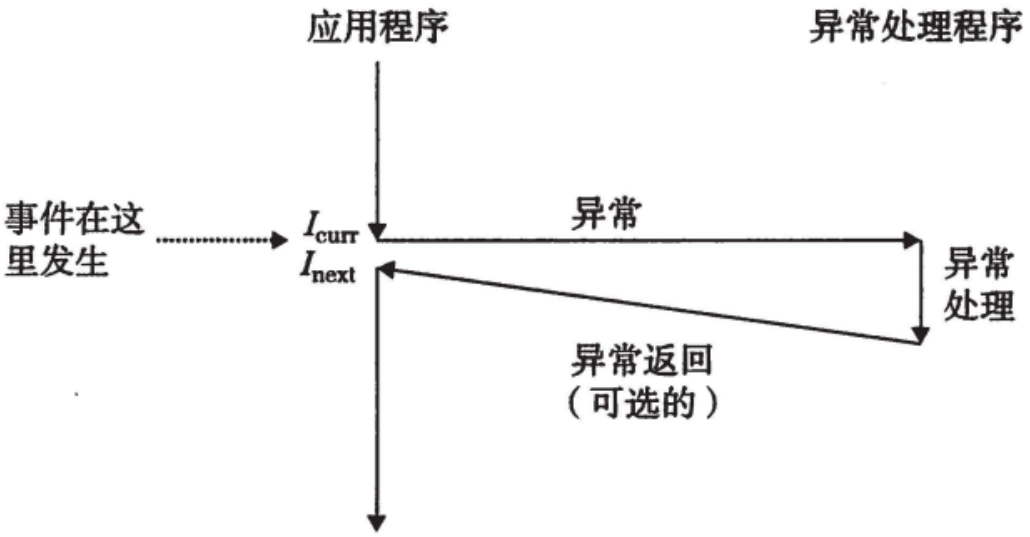


图 3.5: 异常处理图示

固定的某个物理地址上（根据处理器的区别有所不同），以保证处理器能在检测到异常时正确地跳转到那里。这个分发程序可以认为是操作系统的一部分。

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

```
.section .text.exc_vec3
NESTED(except_vec3, 0, sp)
     .set noat
     .set noreorder
  1:
     mfc0 k1,CP0_CAUSE
     la   k0,exception_handlers
     andi k1,0x7c
     addu k0,k1
     lw   k0,(k0)
     NOP
     jr   k0
     nop
END(except_vec3)
     .set at
```

Exercise 3.12 将异常分发代码填入 **boot/start.S** 合适的部分。 ■

这段异常分发代码的作用流程如下：

1. 将 CP0_CAUSE 寄存器的内容拷贝到 k1 寄存器中。
2. 将 execption_handlers 基地址拷贝到 k0。
3. 取得 CP0_CAUSE 中的 2~6 位，也就是对应的异常码，这是区别不同异常的重要标志。

4. 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。

5. 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。

.text.exc_vec3 段需要被链接器放到特定的位置，在 R3000 中这一段是要求放到地址 0x80000080 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，就会自动跳转到地址 0x80000080 处，开始执行。

下面我们在 **tools/scse0_3.lds** 中适当位置增加如下代码，即将.text.exc_vec3 放到 0x80000080 处, 来为操作系统增加异常分发功能。

```
. = 0x80000080;
.except_vec3 : {
    *(.text.exc_vec3)
}
```

Exercise 3.13 将 **tools/scse0_3.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

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

下面我们来看一下 lib/traps.c 中的 trap_init() 函数，来了解异常向量组里存放了什么。

```
extern void handle_int();
extern void handle_reserved();
extern void handle_tlb();
extern void handle_sys();
extern void handle_mod();
unsigned long exception_handlers[32];

void trap_init()
{
    int i;

    for (i = 0; i < 32; i++) {
        set_except_vector(i, handle_reserved);
    }

    set_except_vector(0, handle_int);
    set_except_vector(1, handle_mod);
    set_except_vector(2, handle_tlb);
    set_except_vector(3, handle_tlb);
    set_except_vector(8, handle_sys);
}
void *set_except_vector(int n, void *addr)
{
    unsigned long handler = (unsigned long)addr;
    unsigned long old_handler = exception_handlers[n];
```

```
26      exception_handlers[n] = handler;
27      return (void *)old_handler;
28  }
```

实际上，这个函数实现了对全局变量 exception_handlers[32] 数组初始化的工作，即通过把相应处理函数的地址填到对应数组项中，初始化了如下异常：

0 号异常 的处理函数为 handle_int，表示中断，由时钟中断、控制台中断等中断造成

1 号异常 的处理函数为 handle_mod，表示存储异常，进行存储操作时该页被标记为只读

2 号异常 的处理函数为 handle_tlb，TLB 异常，TLB 中没有和程序地址匹配的有效入口

3 号异常 的处理函数为 handle_tlb，TLB 异常，TLB 失效，且未处于异常模式 (用于提高处理效率)

8 号异常 的处理函数为 handle_sys，系统调用，陷入内核，执行了 syscall 指令

Thinking 3.8 试找出上述 5 个异常处理函数的具体实现位置。 ■

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，我们主要是使用 0 号异常，即中断异常的处理函数。而我们接下来要做的，就是产生时钟中断。

3.3.3 时钟中断

在前面的介绍中我们已经知道 Cause 寄存器中有 8 个独立的中断位。其中 6 位来自外部，另外 2 位是由软件进行读写，且不同中断处理起来也会有差异。所以在完成这一部分内容之前，我们首先来介绍一下从 CPU 到操作系统中关于中断处理的流程。

1 将当前 PC 地址存入 CP0 中的 EPC 寄存器。

2 将 IEc,KUc 拷贝至 KUp 和 IEp 中，同时将 IEc 置为 0，表示关闭全局中断使能，将 KUc 置 1，表示处于内核态。

3 在 Cause 寄存器中，保存 ExcCode 段。由于此处是中断异常，对应的异常码即为 0。

4 PC 转入异常分发程序入口。

5 通过异常分发，判断出当前异常为中断异常，随后进入相应的中断处理程序。在 MOS 中即对应 handle_int 函数。

6 在中断处理程序中进一步判断 CP0_CAUSE 寄存器中是由几号中断位引发的中断，然后进入不同中断对应的中断服务函数。

7 中断处理完成，将 EPC 的值取出到 PC 中，恢复 SR 中相应的中断使能，继续执行。

以上流程中 1-4 以及第 7 步是由 CPU 完成的，真正需要我们完成的只有 5-6 步，而且在这一部分我们只需要完成外设中断中的时钟中断。

下面我们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，还要了解 gxemul 是如何模拟时钟中断的。kclock_init 函数完成了时钟的初始化，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 0xc8，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，0xc8 则表示 1 秒钟中断 200 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 中判断`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。

在 handle_int 函数中出现了一个新的宏 SAVE_ALL，该宏在后面的 lab 中也会使用，为了让大家更好的理解，请大家移步 include/stackframe.h 中，我们一起来看一下这个宏都做了些什么。

顾名思义，这个函数被用于保存一些内容。但具体是在保存些什么呢？让我们通过分析其运行流程来了解。

前四行其首先取出了 SR 寄存器的值，然后利用移位等操作判断第 28 位的值，根据前面的讲述可以知道，也即判断当前是否处于用户模式下。

接下来将当前运行栈的地址保存到 k0 中；然后调用 get_sp 宏，根据中断异常的种类判断需要保存的位置，并分配一定的空间；将之前的运行栈地址与 2 号寄存器 $v0 先保存起来，以便后面放心使用 sp 寄存器与 v0 寄存器。剩下的部分，不言而喻。

Exercise 3.14 通过上面的描述，补充 lib/kclock.c 中的 **kclock_init** 函数。 ■

Thinking 3.9 阅读 kclock_asm.S 和 genex.S 两个文件，并尝试说出 set_timer 和 timer_irq 函数中每行汇编代码的作用 ■

3.3.4 进程调度

handle_int 函数的最后跳转到了 sched_ yield 函数。这个函数在`lib/sched.c`中所定义，它就是我们本次实验最后要写的调度函数。调度的算法很简单，就是时间片轮转的算法。env 中的优先级在这里起到了作用，我们规定其数值表示进程每次运行的时间片数量。不过寻找就绪状态进程不是简单遍历进程链表，而是用两个链表存储所有参与调度进程。当进程被创建时，我们要将其插入第一个进程调度链表的头部。调用 sched_yield 函数时，先判断当前时间片是否用完。如果用完，将其插入另一个进程调度链表的尾部。之后判断当前进程调度链表是否为空。如果为空，切换到另一个进程调度链表。

根据调度队列的用途，我们需要在创建进程时将该 env 插入到第一个队列中。相应的，我们需要在其被销毁（env_destroy）时，将其从队列中移除。

Exercise 3.15 根据注释，完成 **sched_yield** 函数的补充，并根据调度需求对 env.c 中的部分函数进行修改，使得进程能够被正确调度。

提示 1：使用静态变量来存储当前进程剩余执行次数、当前进程、当前正在遍历的队列等。

提示 2：调度队列中并不一定只存在状态为 ENV_RUNNABLE 的进程 ■

Thinking 3.10 阅读相关代码，思考操作系统是怎么根据时钟周期切换进程的。 ■

至此，我们的实验三就算是圆满完成了。

3.4 实验正确结果

取消第一部分的注释并运行，如果你按流程做完本实验全部内容，并且做的结果正确，gxemul 中会输出以下结果。

```
1  main.c: main is start ...
2
3  init.c: mips_init() is called
4
5  Physical memory: 65536K available, base = 65536K, extended = 0K
6
7  to memory 80401000 for struct page directory.
8
9  to memory 80431000 for struct Pages.
10
11 pmap.c:  mips vm init success
12
```

```
pe0->env_id 1024

pe1->env_id 3073

pe2->env_id 5122

env_init() work well!

envid2env() work well!

pe1->env_pgdir 83ffe000

pe1->env_cr3 3ffe000

env_setup_vm passed!

pe2`s sp register 7f3fe000

[00000000] free env 00001402

[00000000] free env 00000c01

[00000000] free env 00000400

env_check() ucceeded!

text & data segment load right!

bss segment load right!

[00000000] free env 00000400

load_icode_check() succeeded!

panic at init.c:29: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

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
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 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
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
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 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
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
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 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，最后一部分只是 2 和 1 的交替输出，没有换行，也不会像示例这样整齐，但 1 的个数约为 2 的两倍。

3.5 代码导读

为了帮助同学们理清 lab3 代码逻辑和执行流程，我们在这里给出代码导读部分。

系统入口点仍然是 _start，不过 lab3 在 lab2 基础上增加了一个代码段，名字叫做.text.exc_vec3，该段代码实际上是一个分发处理异常的过程，对于该段的代码在**异常的分发**小节有所详述，这里不再赘述。

进入入口后很快，CPU 会执行到 main 函数入口处，紧接着进行一系列的初始化操作。对于本实验来说，重点引入了 trap_init，env_create 以及 kclock_init 三个函数。下面进行详细介绍。

1. trap_init:

   对exception_handlers[32] 数组进行初始化，这个数组在前面的中断处理程序中有所涉及，可参看前面.text.exc_vec3 代码部分。主要初始化 0 号异常的处理函数为 handle_int，1 号异常处理函数为 handle_mod，2 号异常处理函数为 handle_tlb，3 号异常处理函数为 handle_tlb，8 号异常处理函数为 handle_sys。初始化结束后，若有异常产生，那么其对应的处理函数就会得到执行。

2. env_create: 创建一个进程，主要分两大部分

   (1) 分配进程控制块

   env_alloc 函数从空闲链表中分配一个空闲控制块，并进行相应的初始化工作，重点就是 PC 和 SP 的正确初始化。env_setup_vm 函数的主要工作是：为进程创建一个页表，在该系统中每一个进程都有自己独立的页表，所建立的这个页表主要在缺页中断或者 TLB 中断中被服务程序用来查询，或者对于具有 MMU 单元的系统，供 MMU 来进行查询。

   (2) 载入执行程序将代码拷贝到新创建的进程的地址空间，这个过程实际上就是一个内存的拷贝，通过 bcopy 完成。

3. kclock_init:

   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 0xc8，其中 0xb5000000 是模拟器 (gxemul) 映射实时钟的位置，而偏移量为 0x100 表示来设置实时钟中断的频率，0xc8 则表示 1 秒钟中断 200 次，如果写入 0，表示关闭实时钟。而实时钟对于 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。一旦实时钟中断产生，就会触发中断异常，将 PC 指向 0x80000080，从而跳转到.text.exc_vec3 代码段进行异常分发。

4. 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

5. timer_irq:

   首先写 0xb5000110 地址响应时钟中断，之后跳转到 sched_yield 中执行。而 sched_yield 函数会调用引起进程切换的函数来完成进程的切换。注意:这里是第一次进行进程切换，请务必保证：kclock_init 函数在 env_create 函数之后调用

6. sched_yield:

   引发进程切换，主要分为两大块:

(1) 将正在执行的进程（如果有）的现场保存到对应的进程控制块中。

(2) 选择一个可以运行的进程，恢复该进程上次被挂起时候的现场（对于新创建的进程，创建的时候仅仅初始化了 PC 和 SP 也可以看作一个现场）这个过程主要通过 env_pop_tf 来完成：该函数其他部分代码比较容易看懂，下面主要看一下关键的数条代码：

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

我们知道在 env.c 中我们对进程中的 env_tf.CP0_STATUS 初始化为 0x10001004，前面提到，这样设置主要是为了让我们的操作系统可以正常对中断（lab3 中主要是时钟中断）进行响应，从而可以正常调用 handle_int 函数，经过 timer_irq、sched_yield 完成进程切换

7. TLB 中断何时被调用？

从上面的分析看，操作系统在实时钟的驱动下，通过时间片轮转算法实现进程的并发执行，不过如果没有 TLB 中断，真的可以正确的运行吗，当然不行。

因为每当硬件在取数据或者取指令的时候，都会发出一个所需数据所在的虚拟地址，TLB 就是将这个虚拟地址转换为对应的物理地址，进而才能够驱动内存取得正确的所期望的数据。但是当 TLB 在转换的时候发现没有对应于该虚拟地址的项，那么此时就会产生一个 TLB 中断。

TLB 对应的中断处理函数是 handle_tlb，通过宏映射到了 do_refill 函数上：这个函数完成 TLB 的填充，在 lab2 中我们已经学习了 TLB 的基本结构，简单来说就是对于不同进程的同一个虚拟地址，结合 ASID 和虚拟地址可以定位到不同的物理地址（这里仍建议花几分钟时间翻回前面巩固一下），下面重点介绍 TLB 缺失处理的过程。

硬件为我们做了什么？在发生 TLB 缺失的时候，会把引发 TLB 缺失的虚拟地址填入到 BadVAddr Register 中，这个寄存器具体的含义请参看 MIPS 手册。接着触发一个 TLB 缺失异常。

我们需要做什么？从 BadVAddr 寄存器中获取使 TLB 缺失的虚拟地址，接着拿着这个虚拟地址通过手动查询页表（mContext 所指向的），找到这个页面所对应的物理页面号，并将这个页面号填入到 PFN 中，也就是 EntryLo 寄存器，填写好之后，tlbwr 指令就会将填入的内容填入到具体的某一项 TLB 表项中。

8. handle_sys:

虽然 lab3 中没有用到系统调用，但我们在这里还是简单介绍一下，相关机制在 lab4 中会用到。系统调用是通过在用户态执行 syscall 指令来触发的，一旦触发，根据上面的介绍，会调用.text.exc_vec3 代码段的代码进行异常分发，最终调用 handle_sys 函数。

这个函数实质上也是一个分发函数，首先这个函数需要从用户态拷贝参数到内核中，然后根据第一个参数（是一个系统调用号）来作为一个数组的索引，取得该索引项所对应的那一项的值，其中这个数组就是sys_call_table(系统调用表)，数组里面存放的每一项都是位于内核中的系统调用服务函数的入口地址。一旦找到对应的入口地址，则跳转到该入口处进行代码的执行。

3.6 任务列表

- Exercise-完成 mips_vm_init 函数
- Exercise-完成 env_init 函数
- Exercise-完成 envid2env 函数
- Exercise-完成 env_setup_vm 函数
- Exercise-完成 env_alloc 函数
- Exercise-完成 load_icode_mapper 函数
- Exercise-完成 load_elf 和 load_icode 函数
- Exercise-完成 env_create 和 env_create_priority 函数
- Exercise-补全 init.c
- Exercise-完成 env_run 函数
- Exercise-修改 page_init 函数
- Exercise-完成 start.S
- Exercise-补全 scse0_3.lds
- Exercise-完成 kclock_init 函数
- Exercise-完成 sched_yield 函数

3.7 实验思考

- 思考-envid2env 的实现
- 思考-地址空间初始化
- 思考-user_data 的作用
- 思考-load-icode 的不同情况
- 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-异常处理函数的实现位置
- 思考-时钟的设置
- 思考-进程的调度

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
4. 掌握页写入异常的处理流程

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

在 lab4 中，我们需要实现上述的系统调用机制，并在此基础上实现进程间通信(IPC)机制和一个重要的进程创建机制 fork。在 fork 部分的实验中，我们会介绍一种被称为写时复制（COW）的特性，以及与其相关的页写入异常处理。

4.2 系统调用 (System Call)

本节中，我们着重讨论系统调用的作用，并完成其实现。

4.2.0 用户态与内核态

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

—"CPU modes", Wikipedia.

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

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

—"User space and kernel space", Wikipedia.

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

我们首先回顾以下几组概念：

1. 用户态 和内核态 (也称用户模式 和内核模式)：它们是 CPU 运行的两种状态。根据 lab3 的说明，在 MOS 操作系统实验使用的仿真 R3000 CPU 中，该状态由 CP0 SR 寄存器中 KUc 位的值标志。

2. 用户空间 和内核空间：它们是虚拟内存（进程的地址空间）中的两部分区域。根据 lab2 的说明，MOS 中的用户空间包括 kuseg，而内核空间主要包括 kseg0 和 kseg1。每个进程的用户空间通常通过页表映射到不同的物理页，而内核空间则直接映射到固定的物理页[1]以及外部硬件设备。CPU 在内核态下可以访问任何内存区域，对物理内存等硬件设备有完整的控制权，而在用户态下则只能访问用户空间。

3. (用户) 进程 和内核：进程是资源分配与调度的基本单位，拥有独立的地址空间，而内核负责管理系统资源和调度进程，使进程能够并发运行。与前两对概念不同，进程和内核并不是对立的存在，可以认为内核是存在于所有进程地址空间中的一段代码。

[1]这里忽略了 kseg2 的情况。

事实上，由于仿真器 gxemul 的实现与 IDT R30xx 手册存在的差异，lab3 中并没有使用真正的用户态。在 gxemul 的实现中，KUc 为 1 表示 CPU 处于用户态，此时若进程试图访问内核空间，则会触发 TLB 异常，这在 MOS 中通常导致内核输出 TOO LOW 并 panic [2]，而在 Linux 中进程通常会收到 SIGSEGV 信号，输出“段错误”(segmentation fault) 等信息并退出。

由于 lab3 使用的进程仍在内核态运行，程序可以在不使用系统调用的情况下，直接读写内核空间的硬件地址，从而向控制台输出文本。为了让进程被调度后严格在用户态下运行，我们需要修改进程控制块中保存的 SR 寄存器的初始状态。结合 `env_pop_tf` 函数的实现，我们知道内核开始调度一个进程时，首先恢复其进程上下文，然后使用 `rfe` 指令进入用户态，因此我们不能直接设置 KUc 位的值。

Exercise 4.0 修改 lib/env.c 中的 `env_alloc()` 函数，使在新创建的进程控制块中，`env_tf.cp0_status` 的值为 `0x1000100c`。 ■

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：系统调用到底是什么意思？为了一探究竟，我们选择一个极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。

```c
#include <stdio.h>

int main()
{
        puts("Hello World!\n");
        return 0;
}
```

在 C 语言的用户程序运行环境中，终端通常被抽象为标准输出流，即文件描述符 (fd) 为 1 的 stdout 文件。通过该文件写入数据，就可以输出文本到终端，而类 Unix 操作系统下写入文件是通过 write 系统调用完成的。因此，我们选择观察 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[3]。

首先，在跳板机上编写如上的 hello.c 代码，并使用 `gcc -g` 将其编译为名为 hello 的可执行文件。随后，在当前目录下执行 `gdb hello` 命令，运行 GDB 对该文件进行调试。使用 list 指令可以查看当前文件的源码，我们的目标 puts 函数在文件内第五行。接着使用 break 5 指令将断点设置在 puts 这条语句上，并通过 step 指令[4] 单步进入到函数中。不断调试，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。由于跳板机中的动态链接库不同，同学们在机器上尝试时，看到的调用栈可能与以下结果有差异。

[2]可参考 mm/pmap.c 中的 pageout 函数，以及 lib/genex.S 中的 do_refill 函数。

[3]为了方便大家重现，我们选用了 Linux x86_64 平台进行演示，不同环境下的显示的调用栈可能不同。

[4]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。

```
(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 0x0000000000400564 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（限于篇幅，下只给出该函数反汇编结果的一部分）。需要注意的是，同学们在机器上调试时，寻找到的是/lib64/ld-linux-x86-64.so.2 下的库文件，此时所谓的 write() 函数为 ___GI____libc_write()，可以直接使用 disassemble 命令，对其进行反汇编。

```
(gdb) disassemble __GI___libc_write
Dump of assembler code for function __GI___libc_write:
   0x00007f54bddcd1d0 <+0>:     endbr64
=> 0x00007f54bddcd1d4 <+4>:     mov    %fs:0x18,%eax
   0x00007f54bddcd1dc <+12>:    test   %eax,%eax
   0x00007f54bddcd1de <+14>:    jne    0x7f54bddcd1f0 <__GI___libc_write+32>
   0x00007f54bddcd1e0 <+16>:    mov    $0x1,%eax
   0x00007f54bddcd1e5 <+21>:    syscall
   0x00007f54bddcd1e7 <+23>:    cmp    $0xfffffffffffff000,%rax
   0x00007f54bddcd1ed <+29>:    ja     0x7f54bddcd240 <__GI___libc_write+112>
   0x00007f54bddcd1ef <+31>:    retq
   0x00007f54bddcd1f0 <+32>:    sub    $0x28,%rsp
```

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。在系统调用完成后，用户空间的相关函数会将系统调用的结果，通过一系列的返回过程，最终反馈给用户程序。（以上部分的实验是基于 x86_64 架构下的工具链进行编译调试的，目的是让同学们对系统调用的过程有大致的认识，与我们实现的 MIPS 架构下的操作系统没有直接关系）

由此我们了解到，系统调用实际上是操作系统为用户态提供的一组接口，进程在用户态下通过系统调用可以访问内核提供的文件系统等服务。

在进行了上面的一系列探究后，我们先整理一下调用 C 标准库中的 puts 函数的过程中发生了什么：

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

总结一下我们的收获:

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

综合上面这些内容，相信你一定已经发现了其中的巧妙之处———内核将自己所能够提供的服务以系统调用的方式提供给用户空间，以供用户程序完成一些特殊的系统级操作。这样一来，所有的特殊操作就全部在操作系统的掌控之中了，因为用户程序只能将服务相关的参数交予操作系统，而实际完成需要特权的操作是由内核经过重重检查后执行的，所以系统调用可以确保系统的安全性。

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

表 4.1: API、系统调用层次结构

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

4.2.2 系统调用机制的实现

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

1. 处理器跳转到异常分发代码处

2. 进入异常分发程序，根据 cause 寄存器值判断异常类型并跳转到对应的处理程序

3. 处理异常，并返回

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

1. `writef` 函数内部的逻辑可分为两部分，一部分负责将参数解析为字符串，一部分负责将字符串输出（`user_myoutput` 函数)

2. `user_myoutput` 函数调用了用户空间的 `syscall_*` 函数

3. `syscall_*` 函数调用了 `msyscall` 函数，猜想系统由此陷入内核态

4. 内核态中将异常分发到 `handle_sys` 函数，将系统调用所需要的信息 (在此处是需要输出的字符 ch) 传递入内核

5. 内核取得信息，执行对应的内核空间的系统调用函数（`sys_*`）

6. 系统调用完成，并返回用户态，同时将返回值“传递”回用户态

7. 从系统调用函数返回，回到用户程序 `writef` 调用处

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

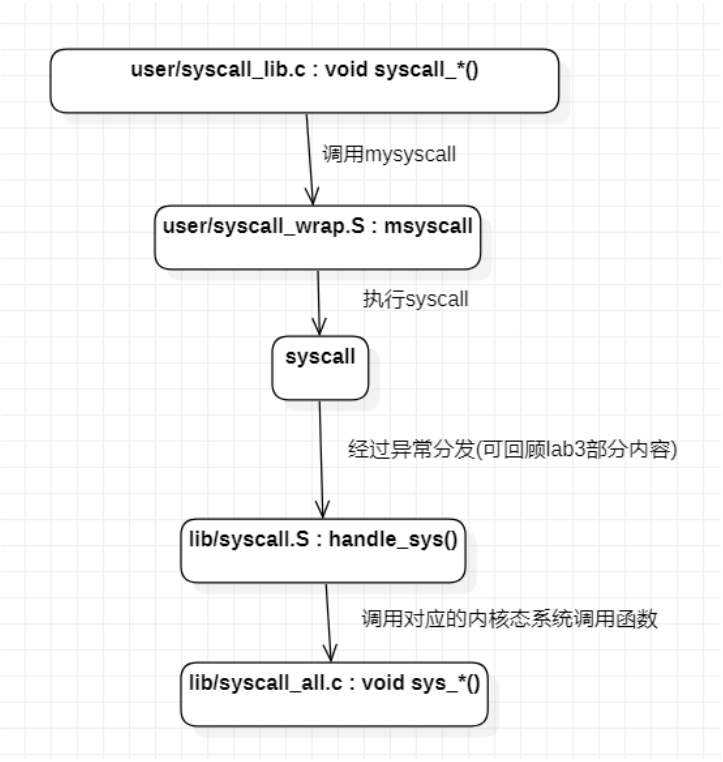


图 4.1: syscall 过程流程图

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

实际上，在我们的 MOS 操作系统实验中，这些 `syscall_*` 的函数与内核中的系统调用函数（`sys_*` 的函数）是一一对应的：`syscall_*` 的函数是我们在用户空间中最接近的内核的也是最原子的函数，而 `sys_*` 的函数是内核中系统调用的具体实现部分。`syscall_*` 的函数的实现中，它们毫无例外都调用了 msyscall 函数，而且函数的第一个参数都是一个与调用名相似的宏（如 `SYS_putchar`），在我们的 MOS 操作系统实验中把这个参数称为**系统调用号**（请在代码中找到这个宏的定义位置，了解系统调用号的排布规则）

类似于不同异常类型对应不同异常号，系统调用号是内核区分这究竟是何种系统调用的唯一依据。除此之外 msyscall 函数还有 5 个参数，这些参数是系统调用时需要传递给内核的参数。而为了方便传递参数，我们采用的是最多参数的系统调用所需要的参数数量（`syscall_mem_map` 函数需要 5 个参数）。

进一步的问题是，这些参数究竟是如何从用户态传递入内核态的呢？这里就需要用 MIPS 的调用规范来说明这件事情了。我们把函数体中不存在函数调用语句的函数称为**叶函数**，自然如果函数体中存在函数调用语句，那么该函数称为**非叶函数**。

我们站在 C 语言的视角举一个简化的例子：

```c
#include <stdio.h>
int g(int x, int y, int z)
{
  int u = x + y + z;
  return u;
}
int f()
{
  int b = 0, c = 0, d = 0;
  int a = g(b, c, d);
  return 0;
}
```

不难看出，代码中的 f 函数即为非叶函数，g 函数为叶函数；回顾内存栈的模型，在 f 函数即将调用 g 函数时，首先需要将 g 所需的参数 b，c，d 的值（或值的地址）压入栈中，此时再进行 jal g 跳转到 g 函数体时，栈中已经保存了对应参数的值（或值的地址），g 函数运行结束后，需要将已经使用完毕的参数从栈中弹出，恢复初始状态。

严格的讲，在 MIPS 的调用规范中，进入函数体时会通过对栈指针做减法（压栈）的方式为该函数自身的**局部变量、返回地址、调用函数的参数**分配存储空间（叶函数没有后两者），在函数调用结束之后会对栈指针做加法（弹栈）来释放这部分空间，我们把这部分空间称为**栈帧 (Stack Frame)** [5]。调用方是在自身的栈帧的底部预留被调用函数的参数存储空间（被调用方 g 从调用方 f 的栈帧中取得参数）。

进一步，MIPS 寄存器使用规范中指出，寄存器 $a0-$a3 用于存放函数调用的前四个参数（但在栈中仍然需要为其预留空间），剩余的参数仅存放在栈中。以我们的 MOS

[5]注意，此处描述的规范与 main 函数的命令行传参不同

操作系统为例，msyscall 函数一共有 6 个参数，前 4 个参数会被 `syscall_*` 的函数分别存入 $a0-$a3 寄存器（寄存器传参的部分）同时栈帧底部保留 16 字节的空间（不要求存入参数的值），后 2 个参数只会被存入在预留空间之上的 8 字节空间内（没有寄存器传参），于是总共 24 字节的空间用于参数传递。这些过程在 C 代码中会由编译器自动编译为 mips 代码，但是我们的 `handle_sys` 函数是以汇编的形式编写的，需要手动在内核中以汇编的方式显式地把函数的参数值“转移”到内核空间中。详情请见图4.2:

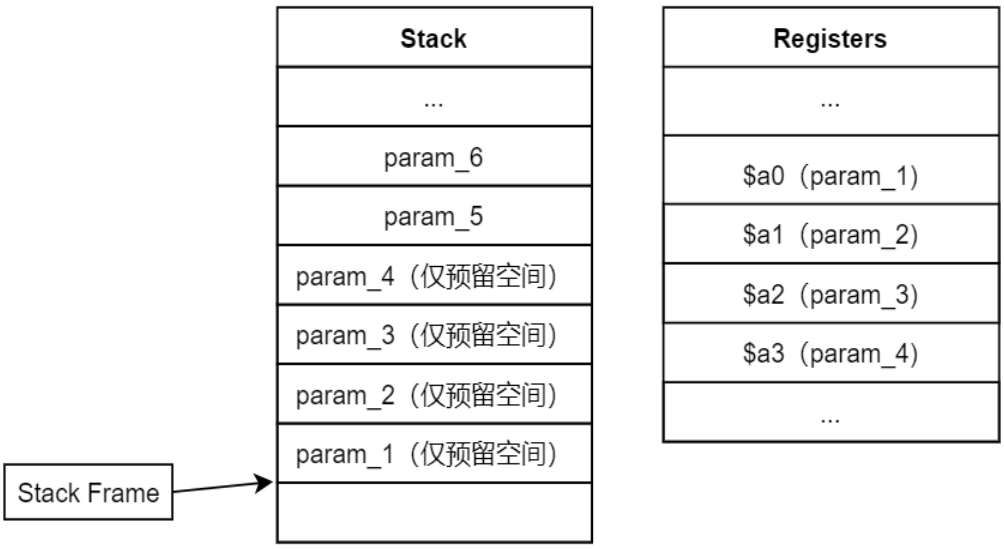


图 4.2: 寄存器传参示意图

既然参数的位置已经被合理安置，那么接下来我们需要编写用户空间中的 msyscall 函数，这个叶函数没有局部变量，也就是说这个函数不需要分配栈帧，我们只需要执行自陷指令 syscall 来陷入内核态并保证处理结束后函数能正常返回即可。请注意不要将 syscall 指令置于跳转指令的延迟槽中，这可以简化内核中的后续处理。

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

在通过 syscall 指令陷入内核态后，处理器将 PC 寄存器指向一个内核中固定的异常处理入口。在初始化异常向量表时，`trap_init` 函数中将系统调用这一异常类型的处理入口设置为 `handle_sys` 函数，我们需要在 lib/syscall.S 中实现该函数。

需要注意的是，陷入内核态的操作并不是从一个函数跳转到了另一个函数，此处的栈指针 $sp 是内核空间的栈指针，系统从用户态切换到内核态后，**内核首先需要将原用户进程的运行现场保存到内核空间**（其保存的结构与结构体`struct Trapframe`等同，请寻找完成这部分功能的代码实现），栈指针指向这个结构体的起始位置，因此我们正是借助这个保存的结构体来获取用户态中传递过来的值（例如：用户态下 $a0 寄存器的值保存在了当前栈下的 TF_REG4(sp) 处）。你可以参照 include/trap.h 的宏，使用 lw 指令取得保存现场的一些寄存器的值。

Listing 14: 内核的系统调用处理程序

```
NESTED(handle_sys,TF_SIZE, sp)
    SAVE_ALL                            /* 用于保存所有寄存器的汇编宏 */
    CLI                                 /* 用于屏蔽中断位的设置的汇编宏 */
    nop
    .set at                             /* 恢复 $at 寄存器的使用 */

    /* TODO: 将 Trapframe 的 EPC 寄存器取出，计算一个合理的值存回 Trapframe 中 */

    /* TODO: 将系统调用号“复制”入寄存器 $a0 */

    addiu   a0, a0, -__SYSCALL_BASE     /* a0 <- “相对”系统调用号 */
    sll     t0, a0, 2                   /* t0 <- 相对系统调用号 * 4 */
    la      t1, sys_call_table          /* t1 <- 系统调用函数的入口表基地址 */
    addu    t1, t1, t0                  /* t1 <- 特定系统调用函数入口表项地址 */
    lw      t2, 0(t1)                   /* t2 <- 特定系统调用函数入口函数地址 */

    lw      t0, TF_REG29(sp)            /* t0 <- 用户态的栈指针 */
    lw      t3, 16(t0)                  /* t3 <- msyscall 的第 5 个参数 */
    lw      t4, 20(t0)                  /* t4 <- msyscall 的第 6 个参数 */

    /* TODO: 在当前栈指针分配 6 个参数的存储空间，并将 6 个参数安置到期望的位置 */


    jalr    t2                          /* 调用 sys_* 函数 */
    nop

    /* TODO: 恢复栈指针到分配前的状态 */

    sw      v0, TF_REG2(sp)             /* 将 $v0 中的 sys_* 函数返回值存入 Trapframe */

    j       ret_from_exception          /* 从异常中返回（恢复现场） */
    nop
END(handle_sys)

sys_call_table:                         /* 系统调用函数的入口表 */
.align 2
    .word sys_putchar
    .word sys_getenvid
    .word sys_yield
    .word sys_env_destroy
    .word sys_set_pgfault_handler
    .word sys_mem_alloc
    .word sys_mem_map
    .word sys_mem_unmap
    .word sys_env_alloc
    .word sys_set_env_status
    .word sys_set_trapframe
    .word sys_panic
    .word sys_ipc_can_send
    .word sys_ipc_recv
    .word sys_cgetc
    /* 每一个整字都将初值设定为对应 sys_* 函数的地址 */
    /* 在此处增加内核系统调用的入口地址 */
```

Thinking 4.1 思考并回答下面的问题：

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

■

Exercise 4.2 按照 lib/syscall.S 中的提示，完成 `handle_sys` 函数，使得内核部分的系统调用机制可以正常工作。 ■

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

4.2.3 基础系统调用函数

在系统调用机制搞定之后，我们可以实现几个系统调用。我们实现些什么系统调用呢？打开 lib/syscall_all.c，可以看到目不暇接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`。这个函数的主要功能是分配内存，简单的说，用户程序可以通过这个系统调用给该程序所允许的虚拟内存空间内存**显式地**分配实际的物理内存 (是不是很像 malloc 的内核态函数？)。对于我们程序员的视角而言，是我们编写的程序在内存中申请了一片空间；而对于操作系统内核来说，是一个进程请求将其运行空间中的某段地址与实际物理内存进行映射，从而可以通过该虚拟页面来对物理内存进行存取访问。那么内核通过什么来确定发出请求的进程是哪一个？又是如何完成分配与映射页面的？此处请回顾进程虚拟页面映射机制、物理内存申请机制。可能用到的函数：`page_alloc`, `page_insert`。完成时请注意对页面位置和权限位的相关判断。

Exercise 4.3 实现 lib/syscall_all.c 中的 int sys_mem_alloc(int sysno,u_int envid, u_int va, u_int perm) 函数 ■

其次是 `sys_mem_map`。这个函数的参数很多，但是意义很直接：将源进程地址空间中的相应内存映射到目标进程的相应地址空间的相应虚拟内存中去。换句话说，此时两者共享一页物理内存。那么具体的逻辑为：首先找到需要操作的两个进程，其次获取源进程的虚拟页面对应的实际物理页面，最后将该物理页面与目标进程的相应地址完成映

射即可。可能用到的函数：`page_alloc`，`page_insert`，`page_lookup`。完成时请注意对页面位置和权限位的相关判断。

Exercise 4.4 实现 lib/syscall_all.c 中的 int sys_mem_map(int sysno,u_int srcid, u_int srcva, u_int dstid, u_int dstva, u_int perm) 函数 ■

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

Exercise 4.5 实现 lib/syscall_all.c 中的 int sys_mem_unmap(int sysno,u_int envid, u_int va) 函数 ■

除了与内存相关的函数外，另外一个常用的系统调用函数是 `sys_yield`。这个函数的功能主要就在于实现用户进程对 CPU 的放弃，从而调度其他的进程。可以利用我们之前已经编写好的函数 `sched_yield`，另外为了通过我们之前编写的进程切换机制保存现场，这里需要先在 KERNEL_SP 和 TIMESTACK 上做一点准备工作，可以回顾 lab3 的部分内容，思考一下在保存进程现场时二者的区别———时钟中断时保存在 TIMESTACK；系统调用时保存在 KERNEL_SP；而 env_run 时默认保存在 TIMESTACK。

Exercise 4.6 实现 lib/syscall_all.c 中的 void sys_yield(void) 函数 ■

ps. 可能大家注意到了，在此我们的系统调用函数并没使用到它的第一个参数 sysno，在这里，sysno 作为系统调用号被传入，现在起的更多只是一个“占位”的作用，能和之前用户层面的系统调用函数参数传递顺序相匹配。

至此，我们能够进一步理解，进程与内核间的关系并非对立：在内核处理进程发起的系统调用时，我们并没有切换 CPU 的地址空间（页目录地址），也不需要将进程上下文 (Trapframe) 保存到进程控制块中，只是切换到内核态下，执行了一些内核代码。可以说，处理系统调用时的内核仍然是代表当前进程的，这也是系统调用等中断与时钟中断的本质区别，也是我们引入 KERNEL_SP 和 TIMESTACK 两种机制来保存进程上下文的一个原因。你也可以结合这一点，理解内核中 `ret_from_exception` 和 `env_pop_tf` 这两个用于返回用户态的汇编函数间的区别。

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

4.3 进程间通信机制 (IPC)

进程间通信机制 (IPC) 是微内核最重要的机制之一。

Note 4.3.1 上世纪末，微内核设计逐渐成为了一个热点。微内核设计主张将传统操作系统中的设备驱动、文件系统等可在用户空间实现的功能，移出内核，作为普通的用户程序来实现。这样，即使它们崩溃，也不会影响到整个系统的稳定。其他应用程序通过进程间通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 MOS 操作系统中。IPC 机制的实现使得我们系统中的进程之间拥有了相互传递消息的能力，为后续实现 fork、文件系统服务、管道与 shell 等均有着极大的帮助。根据之前的讨论，我们能够确定的是:

- IPC 的目的是使两个进程之间可以通讯
- IPC 需要通过系统调用来实现
- IPC 还与进程的数据、页面等信息有关

所谓通信，最直观的一种理解就是交换数据。假如我们能够将让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。但是，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，同一个虚拟地址但却可能在不同进程下对应不同的物理页面，自然对应的值就不同。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？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。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 }
```

从这个函数的实现中可以发现，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，我们就可以借助于内核的空间来实现。也就是说，发送方进程可以将数据以系统调用的形式存放在内核空间中，接收方进程同样以系统调用的方式在内核找到对应的数据，读取并返回。

那么，我们把要传递的消息放在哪里比较好呢？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？——进程控制块！

```
struct Env {
    // Lab 4 IPC
    u_int env_ipc_value;            // data value sent to us
    u_int env_ipc_from;             // envid of the sender
    u_int env_ipc_recving;          // env is blocked receiving
    u_int env_ipc_dstva;            // va at which to map received page
    u_int env_ipc_perm;             // perm of page mapping received
};
```

在进程控制块中我们看到了我们想要的内容：

env_ipc_value	进程传递的具体数值
env_ipc_from	发送方的进程 ID
env_ipc_recving	1：等待接受数据中；0：不可接受数据
env_ipc_dstva	接收到的页面需要与自身的哪个虚拟页面完成映射
env_ipc_perm	传递的页面的权限位设置

知道了这些，我们就不难实现 IPC 机制了。请结合下方讲解完成练习：

Exercise 4.7 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)` 函数用于接受消息。在该函数中：

1. 首先要将自身的 env_ipc_recving 设置为 1，表明该进程准备接受发送方的消息
2. 之后给 env_ipc_dstva 赋值，表明自己要将接受到的页面与 dstva 完成映射
3. 阻塞当前进程，即把当前进程的状态置为不可运行（ENV_NOT_RUNNABLE）
4. 最后放弃 CPU（调用相关函数重新进行调度），安心等待发送方将数据发送过来

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：

1. 根据 envid 找到相应进程，如果指定进程为可接收状态 (考虑 env_ipc_recving)，则发送成功
2. 否则，函数返回-E_IPC_NOT_RECV，表示目标进程未处于接受状态
3. 清除接收进程的接收状态，将相应数据填入进程控制块，传递物理页面的映射关系
4. 修改进程控制块中的进程状态，使接受数据的进程可继续运行 (ENV_RUNNABLE)

IPC 的大致流程总结如图 4.3 所示：

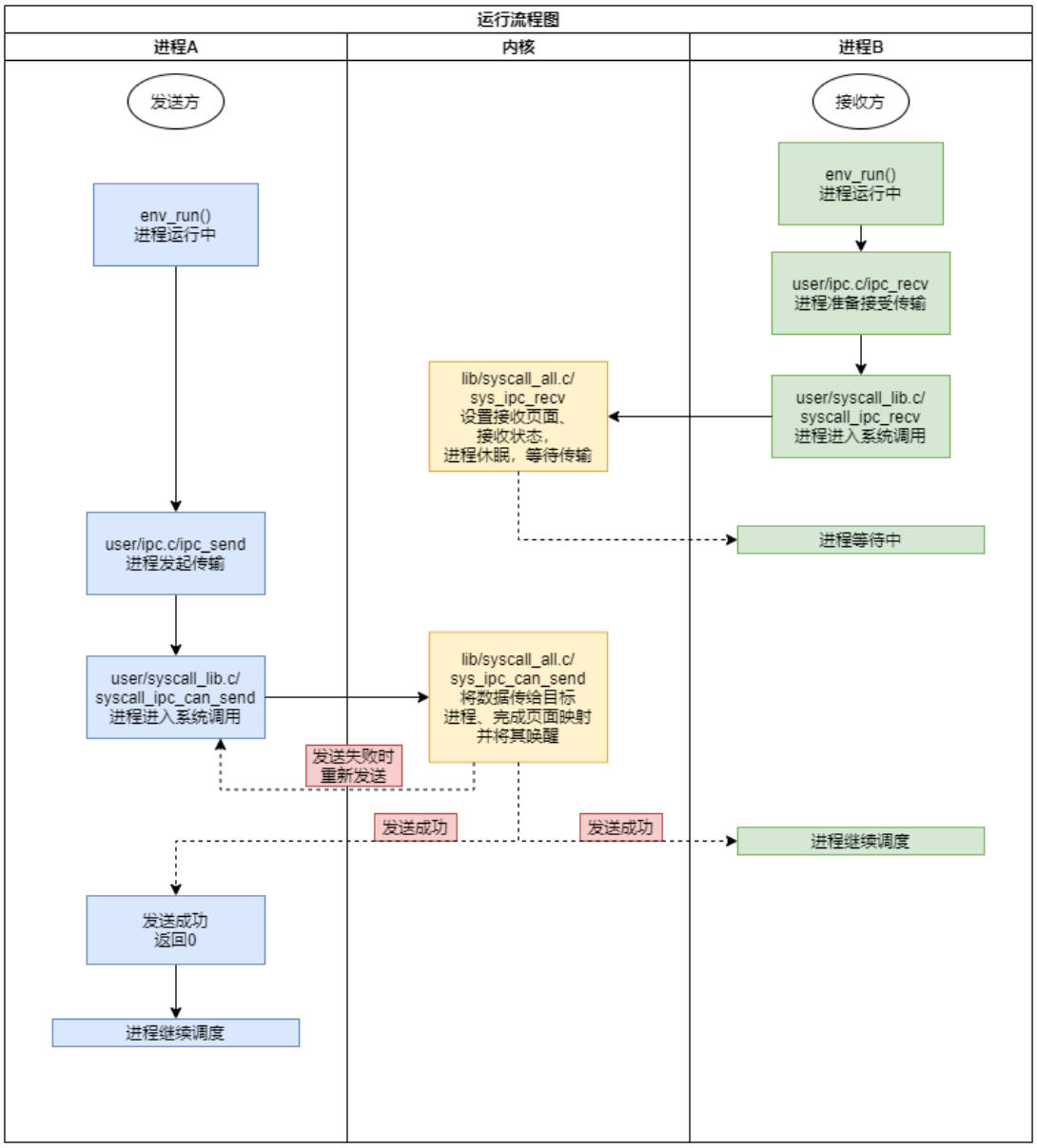


图 4.3: IPC 流程图

值得一提的是，由于在我们的用户程序中，会大量使用 srcva 为 0 的调用来表示只传 value 值，而不需要传递物理页面，换句话说，当 srcva 不为 0 时，我们才建立两个进程的页面映射关系。因此在编写相关函数时也需要注意此种情况。

Thinking 4.2 思考下面的问题，并对这个问题谈谈你的理解：请回顾 lib/env.c 文件中 `mkenvid()` 函数的实现，该函数不会返回 0，请结合系统调用和 IPC 部分的实现与 `envid2env()` 函数的行为进行解释。 ■

4.4 Fork

在 lab3 我们曾提到过，内核通过 `env_alloc` 函数创建一个进程。但如果要让一个进程创建一个进程，就像是父亲与孩子那样，我们就需要基于系统调用，引入新的 fork 机制了。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，而在操作系统中更像是分叉的意思，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，一个进程在调用 fork() 函数后，将从此分叉成为两个进程运行，其中新产生的进程称为原进程的**子进程**。对于操作系统，子进程开始运行时的大部分上下文状态与原进程相同，包括程序镜像、通用寄存器和程序计数器 PC 等。在新的进程中，这一 fork() 调用的返回值为 0，而在旧进程，也就是所谓的父进程中，同一调用的返回值是子进程的进程 ID（MOS 中的 env_id），且一定大于 0[6]。fork 在父子进程中产生不同返回值这一特性，让我们能够在代码中调用 fork 后判断当前在父进程还是子进程中，以执行不同的后续逻辑，也使父进程能够与子进程进行通信。

你可能会想，fork 执行完为什么不直接生成一个空白的进程块，生成一个几乎和父进程一模一样的子进程有什么用呢？事实上，fork 是 Linux 操作系统中创建新进程最主要的方式。按笔者的理解，这是因为相比独立开始运行的两个进程，父子进程间的通信要方便的多。因为 fork 虽然不会构成进程间的统治关系[7]，但子进程中仍能读取原属于父进程的部分数据，父进程也可以根据 fork 返回的子进程 id，通过调用其他系统接口控制其行为。

与 fork 经常“纠缠不清”的，是名为 exec 的一系列系统调用。它会使进程抛弃现有的“一切”，另起炉灶执行新的程序。若在进程中调用 exec，进程的地址空间（以及在内存中持有的所有数据）都将被重置，新程序的二进制镜像将被加载到其代码段，从而让一个从头运行的全新进程取而代之，就像太乙真人用莲藕为哪吒重塑了一个肉身一样。fork 的一种常见应用就被称作 fork-exec，指在 fork 出的子进程中调用 exec，从而在创建出的新进程中运行另一个程序。我们会将 exec 的具体实现作为后续的一个挑战性任务，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 15: 理解 fork

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main() {
```

[6]fork 失败的情况下，子进程不会被创建，且父进程将得到小于 0 的返回值。

[7]默认情况下，父进程退出后子进程不会被强制杀死。在操作系统看来，父子进程之间更像是兄弟关系。

```
5           int var = 1;
6           long pid;
7           printf("Before fork, var = %d.\n", var);
8           pid = fork();
9           printf("After fork, var = %d.\n", var);
10          if (pid == 0) {
11                  var = 2;
12                  sleep(3);
13                  printf("child got %ld, var = %d", pid, var);
14          } else {
15                  sleep(2);
16                  printf("parent got %ld, var = %d", pid, var);
17          }
18          printf(", pid: %ld\n", (long) getpid());
19          return 0;
20  }
```

使用 `gcc fork_test.c && ./a.out` 运行一下，你得到的输出应该如下所示（pid 可能不同）：

```
1  Before fork, var = 1.
2  After fork, var = 1.
3  After fork, var = 1.
4  parent got 16903, var = 1, pid: 16902
5  child got 0, var = 2, pid: 16903
```

我们从这段简短的代码里可以获取到很多的信息，比如以下几点：

- 只有父进程会执行 fork 之前的代码段。
- 父子进程同时开始执行 fork 之后的代码段。
- fork 在不同的进程中返回值不一样，在子进程中返回值为 0，在父进程中返回值不为 0，而为子进程的 pid（Linux 中进程专属的 id，类似于 MOS 中的 envid）。
- 父进程和子进程虽然很多信息相同，但他们的进程控制块是不同的。

从上面的小实验我们也能看出来，子进程实际上就是按父进程的代码段等内存数据，以及进程上下文等状态作为模板而雕琢出来的。但即使如此，父子进程也还是有很多不同的地方，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不一样的？

Thinking 4.3 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？

■

Thinking 4.4 关于 fork 函数的两个返回值，下面说法正确的是：

A、fork 在父进程中被调用两次，产生两个返回值

B、fork 在两个进程中分别被调用一次，产生两个不同的返回值

C、fork 只在父进程中被调用了一次，在两个进程中各产生一个返回值
D、fork 只在子进程中被调用了一次，在两个进程中各产生一个返回值 ■

首先，我们简要概括一下整个 fork 实现过程中可能需要阅读或实现的文件，包括：

- **lib/syscall_all.c**：`sys_env_alloc` 函数，`sys_set_env_status` 函数，`sys_set_pgfault_handler` 函数是我们这次需要完成的函数。
- **lib/traps.c**：`page_fault_handler` 函数负责完成写时复制处理前的相关设置，也是我们这次需要完成的函数。
- **user/fork.c**：`fork` 函数是我们这次实验的重点函数，我们将分多个步骤来完成这个函数。
- **user/fork.c**：`pgfault` 函数是写时复制处理的函数，也是 `page_fault_handler` 后续会调用到的函数，负责对 PTE_COW 标志的页面进行处理，是我们这次需要完成的主要函数之一。
- **user/fork.c**：`duppage` 函数是父进程对子进程页面空间进行映射以及相关标志设置的函数，是我们这次需要完成的主要函数之一。
- **user/pgfault.c**：`set_pgfault_handler` 函数是父进程为子进程设置页错误处理函数的函数，是我们这次需要了解的函数之一。
- **user/entry.S**：用户进程的入口，里面实现了 `__asm_pgfault_handler` 函数，也是 `page_fault_handler` 后续会调用到的函数，是我们这次需要了解的函数之一。
- **user/libos.c**：用户进程入口的 C 语言部分，负责完成执行用户程序 `umain` 前后的准备和清理工作，是我们这次需要了解的函数之一。
- **lib/genex.S**：该文件实现了 MOS 的中断处理流程，虽然不是我们需要实现的重点，但是建议课下多多阅读，理解中断处理的流程。

本实验中 MOS 系统的 fork 函数流程大致如下图所示，其中的大部分函数也是这次本次实验的任务，会在后续详细介绍。请注意，图中提到的缺页中断不是虚拟内存管理中涉及的页缺失异常，而是指本节后续将介绍的页写入异常。

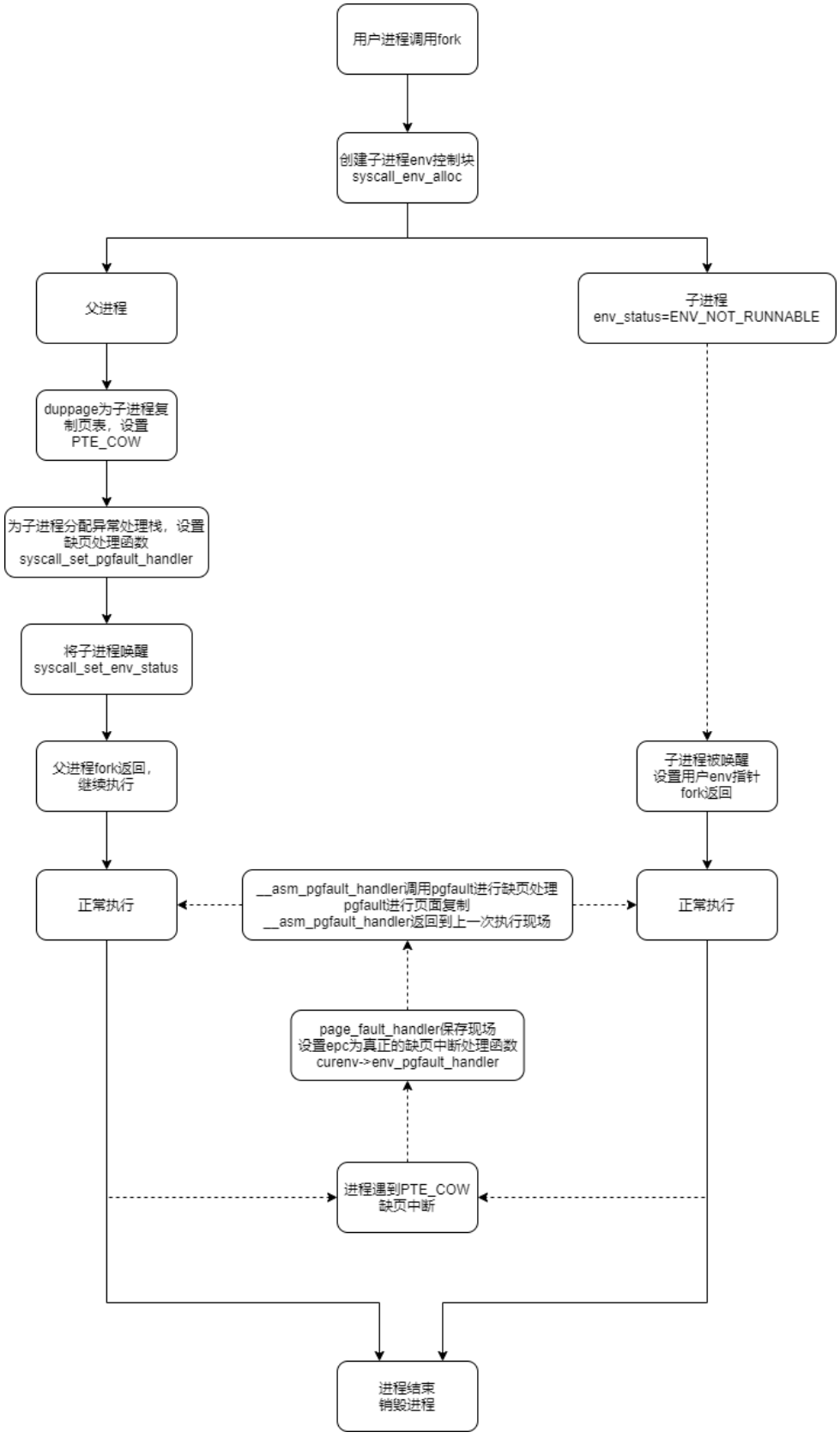

用户进程调用fork
创建子进程env控制块
syscall_env_alloc
父进程
子进程
env_status=ENV_NOT_RUNNABLE
duppage为子进程复制页表，设置PTE_COW
为子进程分配异常处理栈，设置缺页处理函数
syscall_set_pgfault_handler
将子进程唤醒
syscall_set_env_status
父进程fork返回，继续执行
子进程被唤醒
设置用户env指针
fork返回
正常执行
__asm_pgfault_handler调用pgfault进行缺页处理
pgfault进行页面复制
__asm_pgfault_handler返回到上一次执行现场
正常执行
page_fault_handler保存现场
设置epc为真正的缺页中断处理函数
curenv->env_pgfault_handler
进程遇到PTE_COW
缺页中断
进程结束
销毁进程

图 4.4: fork 流程图

4.4.2 写时复制机制

在初步了解 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 的内部细节。在 fork 时，操作系统会为新进程分配独立的虚拟地址空间，但分配独立的地址空间并不意味着一定会分配额外的物理内存。实际上，刚创建好的子进程使用的仍然是其父进程使用的物理内存，子进程地址空间中的代码段、数据段、堆栈等都被映射到父进程中相同区段对应的页面，这也是子进程能“复刻”父进程的状态往后执行的一个原因。也就是说，虽然两者的地址空间（页目录和页表）是不同的，但是它们此时还对应相同的物理内存。

> **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：既然父子进程需要独立并发运行，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，父子进程执行不同逻辑时对相同的内存进行读写，岂不是会造成数据冲突？

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）。COW 类似于一种对虚拟页的保护机制，通俗来讲就是在 fork 后的父子进程中有**修改**内存（一般是数据段或栈）的行为发生时，内核会捕获到一种**页写入异常**，并在异常处理时为**修改内存的进程**的地址空间中相应地址分配新的物理页面。一般来说，子进程的代码段仍会共享父进程的物理空间，两者的程序镜像也完全相同[8]。在这样的保护下，用户程序可以在行为上认为 fork 时父进程中内存的状态被完整复制到了子进程中，此后父子进程可以独立操作各自的内存。

在我们的 MOS 操作系统实验中，进程调用 fork 时，其所有的可写入的内存页面，都需要通过设置页表项标志位 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页面，都会产生一个页写入异常，而在其处理函数中，操作系统会进行**写时复制**，把该页面重新映射到一个新分配的物理页中，并将原物理页中的内容复制过来，同时取消虚拟页的这一标志位。其实现会在后文详细介绍。

[8]如果进程调用了 exec，其代码段也可能被修改，并被映射到新的物理内存。

> **Note 4.4.2** 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 执行一个新的程序镜像，那么所有的拷贝都将前功尽弃。

4.4.3 返回值的秘密

小红：“咦，不科学啊。fork 的两个返回值为啥是系统调用 `syscall_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，还真的就是系统调用 `syscall_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，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。你想想，虽然子进程在被创建出来就已经有了进程控制块和进程上下文，但是子进程是否能够开始被调度是要由父进程决定的。”

在我们的 MOS 操作系统实验中，需要强调的一点是我们实现的 fork 是一个用户态函数，fork 函数中需要若干个“原子的”系统调用来完成所期望的功能。其中最核心的一个系统调用就是新进程的创建 `syscall_env_alloc`。

在 fork 的实现中，我们是通过判断 `syscall_env_alloc` 的返回值来决定 fork 的返回值以及后续动作，所以会有类似这样结构的代码：

```
envid = syscall_env_alloc();
if (envid == 0) {
    // 子进程
    ...
}
else {
    // 父进程
    ...
}
```

既然 fork 的目的是使得父子进程处于几乎相同的运行状态，我们可以认为在返回用户态时，父子进程应该经历了同样的恢复运行现场的过程，只不过对于父进程是从系统调用中返回时恢复现场，而对于子进程则是在进程被调度时恢复现场。在现场恢复后，父子进程都会从内核返回到 msyscall 函数中，而它们的现场中存储的返回值（即 v0 寄存器的值）是不同的。这一返回值随后再被返回到 `syscall_env_alloc` 和 fork 函数，使 fork 函数也能区分两者。

为了实现这一特性，你可能需要先实现 `sys_env_alloc` 的几个任务，在它分配一个新的进程控制块后，还需要用一些当前进程的信息作为模版来填充这个控制块：

运行现场 要复制一份当前进程的运行现场（进程上下文）Trapframe 到子进程的进程控制块中。

程序计数器 子进程的现场中的程序计数器（PC）应该被设置为从内核态返回后的地址，也就是使它陷入异常的 syscall 指令的后一条指令的地址。由于我们之前完成的任务，这个值已经保存于 Trapframe 中。

返回值有关 这个系统调用本身是需要一个返回值的，我们希望系统调用在内核态返回的 envid 只传递给父进程，对于子进程则需要对它的保存的现场 Trapframe 进行一个修改，从而在恢复现场时用 0 覆盖系统调用原来的返回值。

进程状态 我们当然不能让子进程在父进程的 `syscall_env_alloc` 返回后就直接被调度，因为这时候它还没有做好充分的准备，所以我们需要避免它被加入调度队列。

其他信息 观察 Env 结构体的结构，思考下还有哪些字段需要进行初始化，这些字段的初始值应该是继承自父进程还是使用新的值，如果这些字段没有初始化会有什么后果（提示：env_pri）。

Exercise 4.8 请根据上述步骤以及代码中的注释提示，填写 lib/syscall_all.c 中的 `sys_env_alloc` 函数。 ■

在解决完返回值的问题之后，父与子就能够分别走上各自的旅途了。

4.4.4 父子各自的旅途

MOS 允许进程访问自身的进程控制块，而在 user/libos.c 的实现中，用户程序在运行时入口会将一个用户空间中的指针变量 `struct Env *env` 指向当前进程的控制块。对于 fork 后的子进程，它具有了一个与父亲不同的进程控制块，因此在子进程第一次被调度的时候（当然这时还是在 fork 函数中）需要对 env 指针进行更新，使其仍指向当前进程的控制块。这一更新过程与运行时入口对 env 指针的初始化过程相同，具体步骤如下：

1. 通过一个系统调用来取得自己的 envid，因为对于子进程而言 `syscall_env_alloc` 返回的是一个 0 值。
2. 根据获得的 envid，计算对应的进程控制块的下标，将对应的进程控制块的指针赋给 env。

做完上面步骤，当子进程醒来时，就可以从 fork 函数中正常返回，开始自己的旅途了。

Exercise 4.9 按照上述提示，在 user/fork.c 中的 fork 函数中填写关于 `sys_env_alloc` 的部分和随后子进程需要执行的部分。 ■

当然只完成子进程部分，子进程还不能正常跑起来，因为父进程在子进程醒来之前还需要做更多的准备，这些准备中最重要的一步是将父进程地址空间中**需要与子进程共享的页面**映射给子进程，这需要我们遍历父进程的**大部分用户空间页**，并使用将要实现的 `duppage` 函数来完成这一过程。`duppage` 时，对于可以写入的页面的页表项，**在父进程和子进程都需要**加以 PTE_COW 标志位保护起来。

Thinking 4.5 我们并不应该对所有的用户空间页都使用 `duppage` 进行映射。那么究竟哪些用户空间页应该映射，哪些不应该呢？请结合本章的后续描述、mm/pmap.c 中 `mips_vm_init` 函数进行的页面映射以及 include/mmu.h 里的内存布局图进行思考。 ■

Thinking 4.6 在遍历地址空间存取页表项时你需要使用到 vpd 和 vpt 这两个“指针的指针”，请参考 user/entry.S 和 include/mmu.h 中的相关实现，思考并回答这几个问题：

- vpt 和 vpd 的作用是什么？怎样使用它们？
- 从实现的角度谈一下为什么进程能够通过这种方式来存取自身的页表？
- 它们是如何体现自映射设计的？
- 进程能够通过这种方式来修改自己的页表项吗？

■

在 `duppage` 函数中，唯一需要强调的一点是，要对具有不同权限位的页使用不同的方式进行处理。你可能会遇到这几种情况：

只读页面 对于不具有 PTE_R 权限位的页面，按照相同权限（只读）映射给子进程即可。

写时复制页面 即具有 PTE_COW 权限位的页面。这类页面是之前的 fork 时 `duppage` 的结果，且在本次 fork 前必然未被写入过。

共享页面 即具有 PTE_LIBRARY 权限位的页面。这类页面需要保持共享可写的状态，即在父子进程中映射到相同的物理页，使对其进行修改的结果相互可见。在文件系统部分的实验中，我们会使用到这样的页面。

可写页面 即具有 PTE_R 权限位，且不符合以上特殊情况的页面。这类页面需要在父进程和子进程的页表项中都使用 PTE_COW 权限位进行保护。

Exercise 4.10 结合代码注释以及上述提示，填写 user/fork.c 中的 `duppage` 函数。 ■

Note 4.4.3 在用户态实现的 fork 并不是一个原子的过程，所以会出现一段时间（也就是在 `duppage` 之前的时间）我们没有来得及为堆栈所在的页面设置写时复制的保护机制，在这一段时间内对堆栈的修改（比如发生了其他的函数调用），会将非叶函数 `syscall_env_alloc` 函数调用的栈帧中的返回地址覆盖。这一问题对于父进程来说是理所当然的，然而对于子进程来说，这个覆盖导致的后果则是在从 `syscall_env_alloc` 返回时跳转到一个不可预知的位置造成 panic。当然你现在看到的代码已经通过一个优雅的办法来修补这个 bug：与其他系统调用函数不同，

`syscall_env_alloc` 是一个内联 (inline) 的函数，也就是说这个函数并不会被编译为一个函数，而是直接内联展开在 fork 函数内。所以 `syscall_env_alloc` 的栈帧就不存在了，`msyscall` 函数直接返回到了 fork 函数内，如此这个 bug 就解决了。

在完成写时复制的保护机制后，还不能让子进程处于能被调度的状态，因为作为父亲它还有其他的责任——为写时复制特性的**页写入异常**处理做好准备。

4.4.5 页写入异常

内核在捕获到一个常规的缺页中断（page fault）时（在 MOS 中这个情况特指页缺失），会进入到一个在 `trap_init` 中“注册”的 `handle_tlb` 的内核处理函数中，这一汇编函数的实现在 lib/genex.S 中，化名为一个叫 `do_refill` 的函数。如果物理页面在页表中存在，则会将其填入 TLB 并返回异常地址再次执行内存存取的指令。如果物理页面不存在，则会触发一个一般意义的缺页错误，并跳转到 mm/pmap.c 中的 `pageout` 函数中。如果存取地址是合法的用户空间地址，内核会为对应地址分配并映射一个物理页面（被动地分配页面）来解决缺页的问题。

前文中我们提到了写时复制(COW)特性，这种特性也是依赖于异常处理的。CPU 的**页写入异常**会在用户进程写入被标记为 PTE_COW 的页面时产生，我们在 `trap_init` 中为其注册了一个处理函数——`handle_mod`，这一函数会跳转到 lib/traps.c 的 `page_fault_handler` 函数中，这个函数正是处理写时复制特性的内核函数。

Note 4.4.4 由于历史原因，MOS 操作系统的源代码中也使用 page fault 代指这里的页写入异常，但这种异常与之前的缺页中断（页缺失异常）是两种不同的 TLB 异常。具体来说，R3000 的页写入异常会在尝试写入页表项中不带有 dirty bit（表示页面可写入的权限位，即代码中的 PTE_R）的页时产生，而页缺失异常则在尝试访问的页表项中不带有 valid bit（有效位，即 PTE_V）时产生。MOS 操作系统的实现巧妙地利用了一个硬件保留的权限位作为 PTE_COW，并在内核进行 TLB 重填时将标记为 PTE_COW 的页表项中的 dirty bit 置零，因此用户程序在处理时可认为这种页写入异常在且仅在写入 PTE_COW 页面时产生。

你可能会发现，这个函数似乎并没有做任何的页面复制操作。事实上，我们的 MOS 操作系统按照微内核的设计理念，尽可能地将功能实现在用户空间中，其中也包括了页写入异常的处理，因此主要的处理过程是在用户态下完成的。

如果需要在用户态下完成页面复制等处理过程，是不能直接使用正常情况下的进程堆栈的（因为发生页写入异常的也可能是正常堆栈的页面），所以用户进程就需要一个单独的堆栈来执行处理程序，我们把这个堆栈称作 **异常处理栈**，它的栈顶对应的是内存布局中的 UXSTACKTOP。父进程需要为自身以及子进程的异常处理栈映射物理页面。此外，内核还需要知晓进程自身的处理函数所在地址，它的地址存在于进程控制块的 env_pgfault_handler 域中，这个地址也需要事先由父进程通过系统调用设置。

因此，概括一下上述内容，在我们的 MOS 操作系统中，处理页写入异常的大致流程可以概括为：

1. 用户进程触发页写入异常，跳转到 `handle_mod` 函数，再跳转到 `page_fault_handler` 函数。

2. `page_fault_handler` 函数负责将当前现场保存在异常处理栈中，并设置 epc 寄存器的值，使得从中断恢复后能够跳转到 env_pgfault_handler 域存储的异常处理函数的地址。

3. 退出中断，跳转到异常处理函数中，这个函数首先跳转到 `pgfault` 函数（定义在 fork.c 中）进行写时复制处理，之后恢复事先保存好的现场，并恢复 sp 寄存器的值，使得子进程恢复执行。

关于上文提到的异常处理函数，在下文中会做具体介绍。

Exercise 4.11 根据上述提示以及代码注释，完成 lib/traps.c 中的 `page_fault_handler` 函数，设置好异常处理栈以及 epc 寄存器的值。 ■

Thinking 4.7 `page_fault_handler` 函数中，你可能注意到了一个向异常处理栈复制 Trapframe 运行现场的过程，请思考并回答这几个问题：

- 这里实现了一个支持类似于“中断重入”的机制，而在什么时候会出现这种“中断重入”？
- 内核为什么需要将异常的现场 Trapframe 复制到用户空间？

■

让我们回到 fork 函数，在调用 `syscall_env_alloc` 之前，有另一个提示——使用 `set_pgfault_handler` 函数来注册页写入异常处理函数，也就是我们上文总提到的 env_pgfault_handler 域指向的异常处理函数。

```
1  void set_pgfault_handler(void (*fn)(u_int va))
2  {
3    if (__pgfault_handler == 0) {
4      if (syscall_mem_alloc(0, UXSTACKTOP - BY2PG, PTE_V | PTE_R) < 0 ||
5        syscall_set_pgfault_handler(0, __asm_pgfault_handler, UXSTACKTOP) < 0) {
6        writef("cannot set pgfault handler\n");
7        return;
8      }
9    }
10   __pgfault_handler = fn;
11 }
```

上面的 `set_pgfault_handler` 函数中，进程**为自身**分配映射了异常处理栈，同时也用系统调用告知内核自身的处理程序是 `__asm_pgfault_handler`（在 entry.S 定义），随后内核也需要将进程控制块的 env_pgfault_handler 域设为它。在函数的最后，将 entry.S 中的字 ___pgfault_handler 赋值为 fn。这里需要你完成的是内核中的系统调用，它需要设置进程控制块中的两个域。

Exercise 4.12 完成 lib/syscall_all.c 中的 `sys_set_pgfault_handler` 函数。 ■

我们现在知道了页写入异常处理会返回到 entry.S 中的 `__asm_pgfault_handler` 函数，我们再来看这个函数会做些什么。

```
__asm_pgfault_handler:
lw      a0, TF_BADVADDR(sp)
lw      t1, __pgfault_handler
jalr    t1
nop

lw      v1, TF_LO(sp)
mtlo    v1
lw      v0, TF_HI(sp)
lw      v1, TF_EPC(sp)
mthi    v0
mtc0    v1, CP0_EPC
lw      $31, TF_REG31(sp)

lw      $1, TF_REG1(sp)
lw      k0, TF_EPC(sp)
jr      k0
lw      sp, TF_REG29(sp)
```

从内核返回后，此时的栈指针是由内核设置的，处于异常处理栈中，且指向一个由内核复制好的 Trapframe 结构体的底部。该函数通过宏定义的偏移量 TF_BADVADDR，用 lw 指令读取了 Trapframe 中的 cp0_badvaddr 字段的值，这个值也正是 CPU 设置的发生页写入异常的地址。该函数将这个地址作为参数去调用了 ___pgfault_handler 这个变量内存储的函数指针，其指向的函数就是“真正”进行处理的函数。随后就是一段用于恢复现场的汇编，最后使用 MIPS 的延时槽特性，在跳转的同时恢复了正常的栈指针。

Thinking 4.8 到这里我们大概知道了这是一个由用户程序处理并由用户程序自身来恢复运行现场的过程，请思考并回答以下几个问题：

- 在用户态处理页写入异常，相比于在内核态处理有什么优势？
- 从通用寄存器的用途角度讨论，在可能被中断的用户态下进行现场的恢复，要如何做到不破坏现场中的通用寄存器？

■

说到这里，我们就要来实现真正进行处理的函数：user/fork.c 中的 `pgfault` 函数了，`pgfault` 需要完成这些任务：

1. 判断页是否为 COW 的页面，是则进行下一步，否则报错
2. 分配一个新的临时物理页到临时位置，将要复制的内容拷贝到刚刚分配的页中（临时页面位置可以自定义，观察 mmu.h 的地址分配查看哪个地址没有被用到，思考这个临时位置可以定在哪）

3. 将发生页写入异常的地址映射到临时页面上，注意设定好对应的页面权限位，然后解除临时位置的内存映射

Exercise 4.13 填写 user/fork.c 中的 `pgfault` 函数。 ■

这里的 `pgfault` 也正是父进程在 fork 中使用 `set_pgfault_handler` 函数注册的处理函数。

Thinking 4.9 请思考并回答以下几个问题：

- 为什么需要将 `set_pgfault_handler` 的调用放置在 `syscall_env_alloc` 之前？
- 如果放置在写时复制保护机制完成之后会有怎样的效果？
- 子进程是否需要对在 entry.S 定义的字 ___pgfault_handler 赋值？

■

父进程还需要通过类似 `set_pgfault_handler` 的方式，用若干系统调用分配子进程的异常处理栈，并设置其处理函数为 `__asm_pgfault_handler`。最后，父进程通过系统调用 `syscall_set_env_status` 设置子进程为可以运行的状态。在内核中实现 `sys_set_env_status` 函数时，不仅需要设置进程控制块的 env_status 域，还需要在 env_status 被设为 RUNNABLE 时将控制块加入到可调度进程的链表中。

Exercise 4.14 填写 lib/syscall_all.c 中的 `sys_set_env_status` 函数。 ■

说到这里我们需要整理一下思路，fork 中父进程在 `syscall_env_alloc` 后还需要做的事情有：

1. 遍历父进程地址空间，进行 duppage。
2. 为子进程分配异常处理栈。
3. 设置子进程的异常处理函数，确保页写入异常可以被正常处理。
4. 设置子进程的运行状态

最后再将子进程的 envid 返回，fork 函数就大功告成了！

Exercise 4.15 填写 user/fork.c 中的 fork 函数中关于父进程执行的部分。 ■

4.4.6 使用用户程序进行测试

至此，我们的 lab4 实验已经基本完成，接下来就一起来愉快地调试吧！

你可以参照 user 目录下的 tltest.c、fktest.c、pingpong.c 等文件，编写自己的用户程序，测试系统调用、IPC 和 fork 等功能：

1. 在 user 目录下创建 *xxx*.c，加入 `#include "lib.h"`，并编写自己的测试逻辑
2. 为 user/Makefile 中的构建目标 all 加上 *xxx*.x 和 *xxx*.b
3. 在 init/init.c 中用 `ENV_CREATE` 或者 `ENV_CREATE_PRIORITY` 创建用户进程，参数为 `user_`*`xxx`*
4. make 并运行 gxemul，即可观察到 *xx*.c 的运行结果

4.5 实验正确结果

本次实验下发了三个官方测试程序，最简单的测试程序是 user/tltest.c，在系统调用部分完成后，将 `ENV_CREATE(user_tltest)` 加入 init/init.c 即可测试。其参考输出如下：

```
main.c:        main is start ...

init.c:        mips_init() is called

Physical memory: 65536K available, base = 65536K, extended = 0K

to memory 80401000 for struct page directory.

to memory 80431000 for struct Pages.

pmap.c:         mips vm init success

pageout:        @@@___0x7f3fe000___@@@  ins a page

pageout:        @@@___0x406000___@@@  ins a page

Smashing some kernel codes...

If your implementation is correct, you may see some TOO LOW here:

panic at pmap.c:680: ^^^^^^TOO LOW^^^^^^^^^
```

当系统调用与 fork 写完后，单独测试 fork 的程序是 user/fktest.c，使用方法同上，其参考输出如下：

```
  main.c: main is start ...

  init.c: mips_init() is called

  Physical memory: 65536K available, base = 65536K, extended = 0K

  to memory 80401000 for struct page directory.

  to memory 80431000 for struct Pages.

  mips_vm_init:boot_pgdir is 80400000

  pmap.c:  mips vm init success

  panic at init.c:31: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
  pageout:  @@@___0x7f3fe000___@@@  ins a page
  this is father: a:1
  this is father: a:1
  this is father: a:1
  this is father: a:1
  this is father: a:1
  this is father: a:1
  this is father: a:1
  this is father: a:1
  this is father: a:1
  this is father: a:1
    child :a:2
    this is child :a:2
    this is child :a:2
        this is child2 :a:3
        this is child2 :a:3
        this is child2 :a:3
        this is child2 :a:3
  this is father: a:1
  this is father: a:1
  this is father: a:1
  this is father: a:1
  this is father: a:1
    this is child :a:2
    this is child :a:2
    this is child :a:2
```

另一个测试程序 user/pingpong.c 主要测试 fork 和进程间通信，参考输出如下：

```
main.c:        main is start ...

init.c:        mips_init() is called

Physical memory: 65536K available, base = 65536K, extended = 0K

```

```
to memory 80401000 for struct page directory.

to memory 80431000 for struct Pages.

pmap.c:         mips vm init success

pageout:        @@@___0x7f3fe000___@@@  ins a page

pageout:        @@@___0x407000___@@@  ins a page


@@@@@send 0 from 400 to c01

c01 am waiting.....

400 am waiting.....

c01 got 0 from 400


@@@@@send 1 from c01 to 400

c01 am waiting.....

400 got 1 from c01


@@@@@send 2 from 400 to c01

400 am waiting.....

c01 got 2 from 400


@@@@@send 3 from c01 to 400

c01 am waiting.....

400 got 3 from c01


@@@@@send 4 from 400 to c01

400 am waiting.....

c01 got 4 from 400


@@@@@send 5 from c01 to 400

c01 am waiting.....

```

```
400 got 5 from c01


@@@@@send 6 from 400 to c01

400 am waiting.....

c01 got 6 from 400


@@@@@send 7 from c01 to 400

c01 am waiting.....

400 got 7 from c01


@@@@@send 8 from 400 to c01

400 am waiting.....

c01 got 8 from 400


@@@@@send 9 from c01 to 400

c01 am waiting.....

400 got 9 from c01


@@@@@send 10 from 400 to c01

[00000400] destroying 00000400

[00000400] free env 00000400

i am killed ...

c01 got 10 from 400

[00000c01] destroying 00000c01

[00000c01] free env 00000c01

i am killed ...
```

此外，我们还在 gxemul 目录下下发了 vmlinux-tltest、vmlinux-fktest 和 vmlinux-pingpong 三个测试镜像，可以使用 gxemul 仿真运行，以观察参考实现下用户程序的运行效果。

4.6 任务列表

- Exercise-修改 cp0_status
- Exercise-完成 msyscall 函数
- Exercise-完成 handle_sys 函数
- Exercise-实现 sys_mem_alloc 函数
- Exercise-实现 sys_mem_map 函数
- Exercise-实现 sys_mem_unmap 函数
- Exercise-实现 sys_yield 函数
- Exercise-实现 sys_ipc_recv 函数和 sys_ipc_can_send 函数
- Exercise-填写 sys_env_alloc 函数
- Exercise-填写 fork 函数中关于 sys_env_alloc 的部分和 “子进程” 执行的部分
- Exercise-填写 duppage 函数
- Exercise-完成 page_fault_handler 函数
- Exercise-完成 sys_set_pgfault_handler 函数
- Exercise-填写 pgfault 函数
- Exercise-填写 sys_set_env_status 函数
- Exercise-填写 fork 函数中关于 “父进程” 执行的部分

4.7 实验思考

- 思考-系统调用的实现
- 思考-mkenvid 函数细节
- 思考-不同的进程代码执行
- 思考-fork 的返回结果
- 思考-用户空间的保护
- 思考-vpt 的使用
- 思考-页写入异常-内核处理
- 思考-页写入异常-用户处理-1
- 思考-页写入异常-用户处理-2

CHAPTER 5

文件系统

5.1 实验目的

1. 了解文件系统的基本概念和作用。
2. 了解普通磁盘的基本结构和读写方式。
3. 了解实现设备驱动的方法。
4. 掌握并实现文件系统服务的基本操作。
5. 了解微内核的基本设计思想和结构。

在之前的实验中，所有的程序和数据都存放在内存中。然而内存空间的大小是有限的，且内存中的数据具有易失性。因此有些数据必须保存在磁盘、光盘等外部存储设备上。这些存储设备能够长期地保存大量的数据，且可以方便地将数据装载到不同进程的内存空间进行共享。为了便于管理存放在外部存储设备上的数据，我们在操作系统中引入了文件系统。文件系统将文件作为数据存储和访问的单位，可看作是对用户数据的逻辑抽象。对于用户而言，文件系统可以屏蔽访问外存上数据的复杂性。

5.2 文件系统概述

计算机文件系统是一种存储和组织数据的方法，它使得对数据的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，用户不必关心数据实际保存在硬盘（或者光盘）的哪个数据块上，只需要记住这个文件的所属目录和文件名。在写入新数据之前，用户不必关心硬盘上的哪个块没有被使用，硬盘上的存储空间管理（分配和释放）由文件系统自动完成，用户只需要记住数据被写入到了哪个文件中。

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，实际的数据在内存中或

者通过网络协议（如 NFS、SMB、9P 等）提供，甚至可能根本没有对应的文件（如 proc 文件系统）。

文件系统对文件进行管理，普遍意义上的“文件”指的是 txt、pdf 等格式的文件；而广义上，一切带标识的、在逻辑上有完整意义的字节序列都可以称为“文件”。文件系统将外部设备抽象为文件，从而可以统一管理外部设备，实现对数据的存储、组织、访问和获取等操作。在我们实现的精简的文件系统中，需要对三种设备进行统一地管理，即文件设备（file，即狭义的“文件”）、控制台（console）和管道（pipe）。后两者将出现在 lab6 中。

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 文件系统的设计有哪些好处和不足？ ■

5.2.1 磁盘文件系统

磁盘文件系统是一种利用存储设备来保存计算机文件的文件系统，最常用的数据存储设备是磁盘驱动器，可以直接或者间接地连接到计算机上。严格来说，磁盘文件系统和操作系统中使用的文件系统不一定相同，如我们可以在 Linux 中挂载使用 Ext4、FAT32 等多种文件系统的磁盘驱动器，但是 Linux 中运行的程序访问这些文件系统的界面都是 Linux 的 VFS 文件系统。

5.2.2 用户空间文件系统

在以 Linux 为代表的宏内核操作系统中，文件系统是内核的一部分。文件系统作为内核资源的索引发挥了重要的定位内核资源的作用，重要的 mmap，ioctl，read，write 操作都依赖文件系统实现。与此相对的是众多微内核操作系统中使用的用户空间文件系统，其特点是文件系统在用户空间中实现，通过特殊的系统调用接口或者通用机制为其他用户程序提供服务。与此概念相关的还有用户态驱动程序。

5.2.3 文件系统的设计与实现

在本次实验中，我们将要实现一个简单但结构完整的文件系统。整个文件系统包括以下几个部分：

1. **外部存储设备驱动** 通常，外部设备的操作需要通过按照一定操作序列读写特定的寄存器来实现。为了将这种操作转化为具有通用、明确语义的接口，必须实现相应的驱动程序。在本部分，我们将实现 IDE 磁盘的用户态驱动程序。

2. **文件系统结构** 在本部分，我们实现磁盘上和操作系统中的文件系统结构，并通过驱动程序实现文件系统操作相关函数。

3. **文件系统的用户接口** 在本部分，我们提供接口和机制使得用户程序能够使用文件系统，这主要通过一个用户态的文件系统服务来实现。同时，我们引入了文件描述符等结构使操作系统和用户程序可以抽象地操作文件而忽略其实际的物理表示。

整个文件系统所涉及的代码文件众多，此处介绍部分代码文件的主要功能来为大家建立初步的印象。我们将通过 fs/fsformat.c 来创建磁盘镜像，在 fs/fs.c 中实现文件系统的基本功能函数，文件系统进程通过 fs/ide.c 与磁盘镜像进行交互，其进程主要运行在 fs/serv.c 上，并在 fs/serv.c 中通过 IPC 通信与用户进程 user/fsipc.c 内的通信函数进行交互；用户进程在 user/file.c 中实现用户接口，并在 user/fd.c 中引入文件描述符，抽象地操作文件、管道等内容。(注：fs.h 头文件存在于 fs/fs.h 和 include/fs.h 中，其内容并不相同)

> **Note 5.2.1** IDE 的英文全称为“Integrated Drive Electronics”，即“集成电子驱动器”，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。

接下来我们一一详细解读这些部分的实现。

5.3 IDE 磁盘驱动

为了在磁盘等外部设备上实现文件系统，我们必须为这些外部设备编写驱动程序。实际上，MOS 操作系统中已经实现了一个简单的驱动程序，那就是位于 driver 目录下的串口通信驱动程序。在这个驱动程序中我们使用了内存映射 I/O(MMIO) 技术编写驱动。

本次要实现的硬盘驱动程序与已经实现的串口驱动，都采用 MMIO 技术编写驱动，不同之处在于，我们需要驱动的物理设备——IDE 磁盘功能更加复杂，并且本次要编写的驱动程序完全运行在用户空间。

本部分我们将首先学习内存映射 I/O，之后了解 IDE 磁盘的结构和操作，最后学习磁盘驱动程序的编写。

5.3.1 内存映射 I/O (MMIO)

在 lab2 实验中，我们已经了解了 MIPS 存储器地址映射的基本内容。几乎每一种外设都是通过读写设备上的寄存器来进行数据通信，外设寄存器也称为 **I/O 端口**，用来访问 I/O 设备。外设寄存器通常包括控制寄存器、状态寄存器和数据寄存器。这些硬件 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 I/O 空间，与一般我们说的内存空间是不同的。外设的 I/O 地址空间是系统启动后才确定（实际上，这个工作是由 BIOS 完成后告知操作系统的)。通常的体系结构（如 x86）没有为这些外设 I/O 空间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访问 I/O 虚拟地址空间，因此**必须首先将它们映射到内核虚地址空间**，驱动程序才能基于虚拟地址及访存指令来实现对 IO 设备的访问。

幸运的是，实验中使用的 MIPS 体系结构并没有复杂的 I/O 端口的概念，而是统一使用内存映射 I/O 的模型。MIPS 的地址空间中，其在内核地址空间中（kseg0 和 kseg1

段）实现了硬件级别的物理地址和内核虚拟地址的转换机制，其中，对 kseg1 段地址的读写不经过 MMU 映射、不经过高速缓存的特性，正是外部设备驱动所需要的。由于我们是在模拟器上运行操作系统，I/O 设备的物理地址是完全固定的，因此我们可以通过简单地读写某些固定的内核虚拟地址来实现驱动程序的功能。

在之前的实验中，我们曾经使用 KADDR 宏把一个物理地址转换为 kseg0 段的内核虚拟地址，实际上是给物理地址加上 ULIM 的值（即0x80000000）。而在编写设备驱动的时候，我们需要将物理地址转换为 kseg1 段的内核虚拟地址，给物理地址加上 kseg1 的偏移值 (0xA0000000)。

Thinking 5.2 如果通过 kseg0 读写设备，那么对于设备的写入会缓存到 Cache 中。这是一种**错误**的行为，在实际编写代码的时候这么做会引发不可预知的问题。请思考：这么做这会引发什么问题？对于不同种类的设备（如我们提到的串口设备和 IDE 磁盘）的操作会有差异吗？可以从缓存的性质和缓存更新的策略来考虑。 ■

以我们编写完成的串口设备驱动为例，Gxemul 提供的 console 设备的地址为 0x10000000，设备寄存器映射如表5.1所示：

表 5.1: Gxemul Console 内存映射

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

现在，我们通过往内存的 (0x10000000+0xA0000000) 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 }
```

而在本次实验中，我们需要编写 IDE 磁盘的驱动完全位于用户空间，用户态进程若是直接读写内核虚拟地址将会由处理器引发一个地址错误（ADEL/S）。所以我们对于设备的读写必须通过系统调用来实现。这里我们引入了 sys_write_dev 和 sys_read_dev 两个系统调用来实现设备的读写操作。这两个系统调用以用户虚拟地址，设备的物理地址和读写的长度（按字节计数）作为参数，在内核空间中完成 I/O 操作。

Exercise 5.1 请根据 lib/syscall_all.c 中的说明，完成 `sys_write_dev` 函数和 `sys_read_dev` 函数的实现，并且在 user/lib.h, user/syscall_lib.c 中完成用户态的相应系统调用的接口。

编写这两个系统调用时需要注意物理地址与内核虚拟地址之间的转换。

同时还要检查物理地址的有效性，在实验中允许访问的地址范围为：console: [0x10000000, 0x10000020)，disk: [0x13000000, 0x13004200)，rtc: [0x15000000, 0x15000200)，当出现越界时，应返回指定的错误码。 ■

5.3.2 IDE 磁盘

在 MOS 操作系统实验中，Gxemul 模拟器提供的“磁盘”是一个 IDE 仿真设备，需要在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。

磁盘的物理结构

我们首先简单介绍一下与磁盘相关的基本知识。首先是几个基本概念：

1. 扇区 (Sector)：磁盘盘片被划分成很多扇形的区域，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：

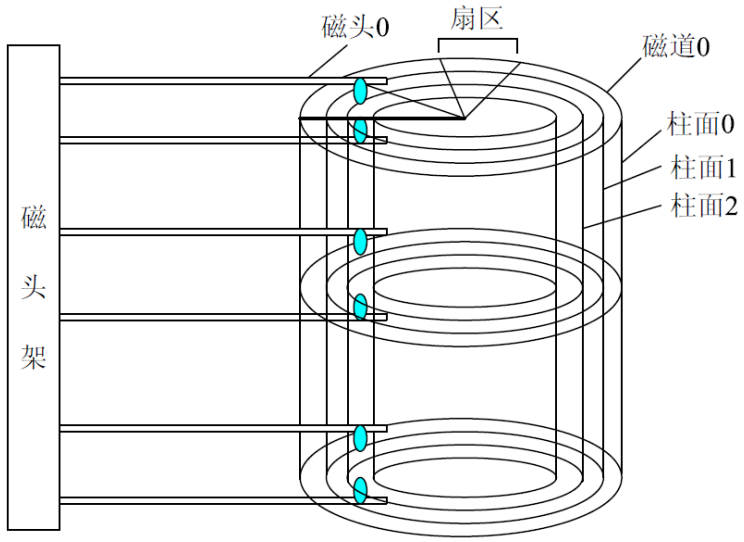


图 5.1: 磁盘结构示意图

IDE 磁盘操作

前文中我们提到过，扇区 (Sector) 是磁盘读写的基本单位，Gxemul 也提供了对扇区进行操作的基本方法。对于 Gxemul 提供的模拟 IDE 磁盘（Simulated IDE disk），我们可以把它当作真实的磁盘去读写数据，通过读写特定位置实现数据的读写以及查看读写是否成功。(特定位置和缓冲区的偏移量在表5.2中给出)。

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：

表 5.2: Gxemul 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.

5.3.3 驱动程序编写

通过对`printcharc` 函数的实现的分析，我们已经掌握了 I/O 操作的基本方法，那么，读写 IDE 磁盘的相关代码也就不难理解了。我们以从硬盘上读取一些扇区为例，先了解一下**内核态的驱动**是如何编写的：

`read_sector` 函数：

```
1  extern int read_sector(int diskno, int offset);
```

```
1  # read sector at specified offset from the beginning of the disk image.
2  LEAF(read_sector)
3      sw  a0, 0xB3000010  # select the IDE id.
4      sw  a1, 0xB3000000  # offset.
5      li  t0, 0
6      sb  t0, 0xB3000020  # start read.
7      lw  v0, 0xB3000030
8      nop
9      jr  ra
10     nop
11 END(read_sector)
```

当需要从磁盘的指定位置读取一个 sector 时，我们需要调用 `read_sector` 函数来将磁盘中对应 sector 的数据读到设备缓冲区中。注意，**所有的地址操作都需要将物理地址转换成虚拟地址**。此处设备基地址对应的 kseg1 的内核虚拟地址是 0xB3000000。

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 0xB3000010 处，这样就表示我们将使用编号为 $a0 的磁盘。在本实验中，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 0xB3000000 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(表5.2)，向内存 0xB3000020 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。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 (offset 0x4000-0x41ff) 中拷贝到目的位置。至此，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 0xB3000020 处写入 1 来启动操作，并从 0xB3000030 处获取写磁盘操作的返回值。

相应地，**用户态磁盘驱动**使用系统调用代替直接对内存空间的读写，从而完成寄存器配置和数据拷贝等功能。

Exercise 5.2 参考内核态驱动，使用系统调用完成 fs/ide.c 中的 `ide_write` 函数，以及 `ide_read` 函数，实现对磁盘的读写操作。 ■

实现了磁盘驱动，我们可以尝试对磁盘进行读写测试。在此之前，需要通过修改运行命令将磁盘镜像挂载到 MOS 操作系统上。编译生成的磁盘镜像文件位于 gxemul/fs.img，在运行命令上加上-d gxemul/fs.img 即可挂载该磁盘镜像，而使用-d x:gxemul/fs.img 可以指定该磁盘镜像的磁盘 ID 为 x。完整的运行命令如下：

Note 5.3.1 /OSLAB/gxemul -E testmips -C R3000 -M 64 -d gxemul/fs.img elf-file (其中 elf-file 是你编译生成的 vmlinux 文件的路径)。

5.4 文件系统结构

实现了 IDE 磁盘的驱动，我们就有了在磁盘上实现文件系统的基础。接下来我们设计整个文件系统的结构，并在磁盘和操作系统中分别实现对应的结构。

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)。MOS 操作系统的文件系统采用了类似的设计，但需要注意与 Unix/Linux 操作系统的区别。

5.4.1 磁盘文件系统布局

磁盘空间的基本布局如图5.2所示。

图中出现的 Block 是磁盘块。不同于扇区 Sector，磁盘块是一个虚拟概念，是操作系统与磁盘交互的最小单位；操作系统将相邻的扇区组合在一起，形成磁盘块进行整体

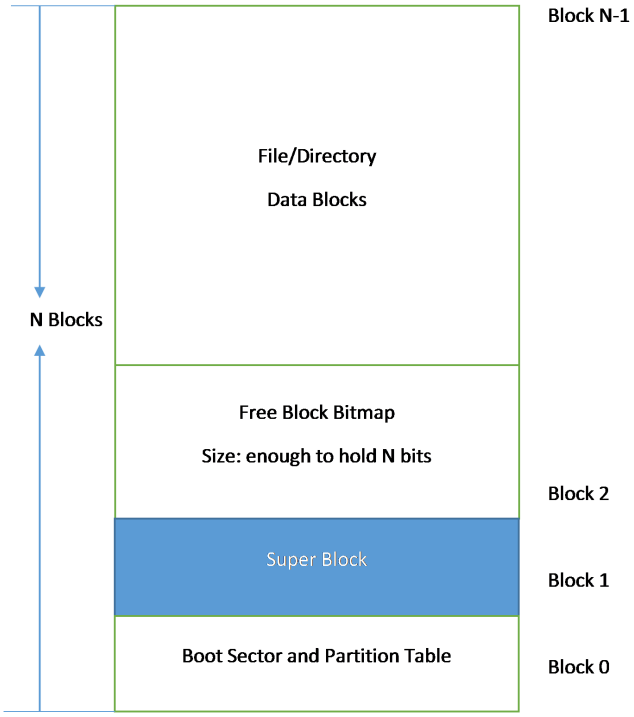


图 5.2: 磁盘空间布局示意图

操作，减小了因扇区过多带来的寻址困难；磁盘块的大小由操作系统决定，一般由 2 的 n 次方个扇区构成。而扇区是真实存在的，是磁盘读写的基本单位，与操作系统无关。

从图5.2中可以看到，MOS 操作系统把磁盘最开始的一个磁盘块 (4096 字节) 当作引导扇区和分区表使用。接下来的一个磁盘块作为超级块 (Super Block)，用来描述文件系统的基本信息，如 Magic Number、磁盘大小以及根目录的位置。

> **Note 5.4.2** 在真实的文件系统中，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。

MOS 操作系统中超级块的结构如下:

```
1 struct Super {
2     u_int s_magic;      // Magic number: FS_MAGIC
3     u_int s_nblocks;    // Total number of blocks on disk
4     struct File s_root; // Root directory node
5 };
```

其中每个域的意义如下:

- `s_magic`: 魔数，用于识别该文件系统，为一个常量。
- `s_nblocks`: 记录本文件系统有多少个磁盘块，本文件系统为 1024。
- `s_root`为根目录，其`f_type`为`FTYPE_DIR`，`f_name`为"/"。

通常采用两种数据结构来管理可用的资源：链表和位图。在 lab2 和 lab3 实验中，我们使用了链表来管理空闲内存资源和进程控制块。在文件系统中，我们将使用位图

(Bitmap) 法来管理空闲的磁盘资源，用一个二进制位 bit 标识磁盘中的每个磁盘块的使用情况（实验中，1 表示空闲）。

这里我们参考 tools/fsformat 表述文件系统标记空闲块的机制。tools/fsformat 是用于创建符合我们定义的文件系统结构的工具，用于将多个文件按照内核所定义的文件系统写入到磁盘镜像中。在写入文件之前，fs/fsformat.c 的`init_disk`函数，将所有的块都标为空闲块：

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

nbitblock 表示记录整个磁盘上所有块的使用信息，需要多少个磁盘块来存储位图。紧接着，我们使用 memset 将位图中的每一个字节 (Byte) 都设成 0xff ，即将所有位图块的每一位都设为 1 ，表示这一块磁盘处于空闲状态。如果位图还有剩余，不能将最后一块位图块中靠后的一部分内容标记为空闲，因为这些位所对应的磁盘块并不存在，不可被使用。因此，将所有的位图块的每一位都置为 1 之后，还需要根据实际情况，将位图不存在的部分设为 0。

相应地，在 MOS 操作系统中，文件系统也需要根据位图来判断和标记磁盘的使用情况。fs/fs.c 中的 `block_is_free` 函数就用来通过位图中的特定位来判断指定的磁盘块是否被占用。

```
int block_is_free(u_int blockno)
{
    if (super == 0 || blockno >= super->s_nblocks) {
        return 0;
    }
    if (bitmap[blockno / 32] & (1 << (blockno % 32))) {
        return 1;
    }
    return 0;
}
```

Exercise 5.3 文件系统需要负责维护磁盘块的申请和释放，在回收一个磁盘块时，需要更改位图中的标志位。如果要将一个磁盘块设置为 free，只需要将位图中对应的位的值设置为 1 即可。请完成 fs/fs.c 中的 `free_block` 函数，实现这一功能。同时思考为什么参数 `blockno` 的值不能为 0 ？

```
// Overview:
//  Mark a block as free in the bitmap.
void
free_block(u_int blockno)
{
    // Step 1: Check if the parameter `blockno` is valid (`blockno` can't be zero).

    // Step 2: Update the flag bit in bitmap.

}
```

5.4.2 文件系统详细结构

操作系统要想管理一类资源，就得有相应的数据结构。对于描述和管理文件来说，一般使用文件控制块（File 结构体）。其定义如下：

```
// file control blocks, defined in include/fs.h
struct File {
    u_char f_name[MAXNAMELEN];  // filename
    u_int f_size;           // file size in bytes
    u_int f_type;           // file type
    u_int f_direct[NDIRECT];
    u_int f_indirect;
    struct File *f_dir;
    u_char f_pad[BY2FILE - MAXNAMELEN - 4 - 4 - NDIRECT * 4 - 4 - 4];
};
```

结合文件控制块的示意图5.3，我们对各个域进行解读：`f_name`为文件名称，文件名的最大长度 MAXNAMELEN 值为 128。`f_size`为文件的大小，单位为字节。`f_type`为文件类型，有普通文件 (`FTYPE_REG`) 和文件夹 (`FTYPE_DIR`) 两种。`f_direct[NDIRECT]`为文件的直接指针，每个文件控制块设有 10 个直接指针，用来记录文件的数据块在磁盘上的位置。每个磁盘块的大小为 4KB，也就是说，这十个直接指针能够表示最大 40KB 的文件，而当文件的大小大于 40KB 时，就需要用到间接指针。`f_indirect`指向一个间接磁盘块，用来存储指向文件内容的磁盘块的指针。为了简化计算，我们不使用间接磁盘块的前十个指针。`f_dir`指向文件所属的文件目录。`f_pad`则是为了让整数个文件结构体占用一个磁盘块，填充结构体中剩下的字节。

Thinking 5.3 比较 MOS 操作系统的文件控制块和 Unix/Linux 操作系统的 inode 及相关概念，试述二者的不同之处。 ■

Note 5.4.3 我们的文件系统中的文件控制块只使用了一级间接指针域，也只有一个。而在真实的文件系统中，为了支持更大的文件，通常会使用多个间接磁盘块，或使用多级间接磁盘块。MOS 操作系统内核在这一点上做了极大的简化。

对于普通的文件，其指向的磁盘块存储着文件内容，而对于目录文件来说，其指向的磁盘块存储着该目录下各个文件对应的的文件控制块。当我们要查找某个文件时，首先从超级块中读取根目录的文件控制块，然后沿着目标路径，挨个查看当前目录所包含的文件是否与下一级目标文件同名，如此便能查找到最终的目标文件。

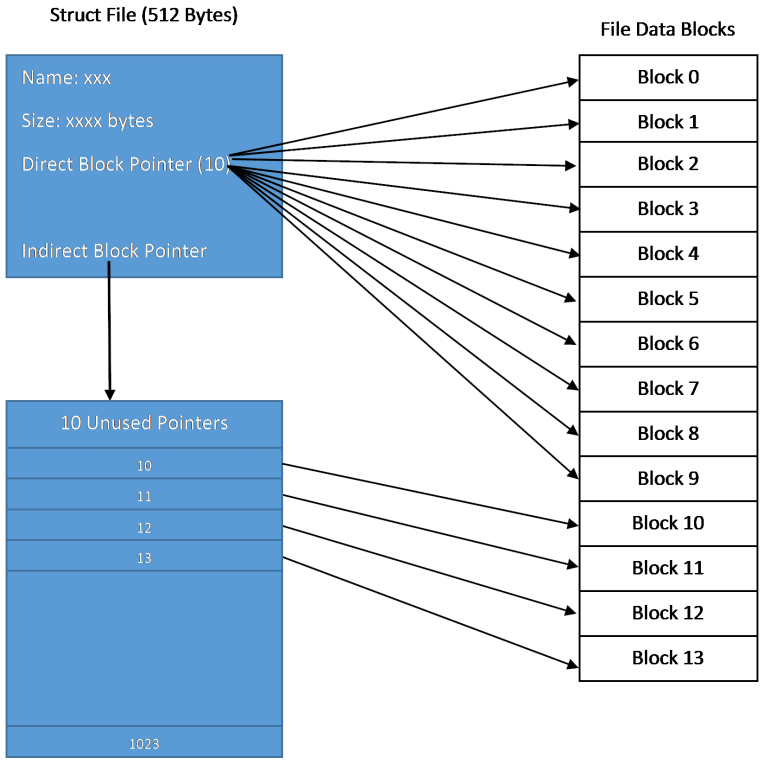


图 5.3: 文件控制块

Thinking 5.4 查找代码中的相关定义，试回答一个磁盘块中最多能存储多少个文件控制块？一个目录下最多能有多少个文件？我们的文件系统支持的单个文件最大为多大？ ■

为了更加细致地了解文件系统的内部结构，我们通过 fsformat（由 fs/fsformat.c 编译而成）程序来创建一个磁盘镜像文件 gxemul/fs.img。通过观察头文件和 fs/Makefile，我们可以看出，fs/fsformat.c 的编译过程与其他文件有所不同，其使用的是 Linux 下的 gcc 编译器，而非 mips_4KC-gcc 交叉编译器。编译生成的 fsformat 独立于 MOS 操作系统，专门用于创建磁盘镜像文件。生成的镜像文件 fs.img 可以模拟与真实的磁盘文件设备的交互。请阅读 fs/fsformat.c 和 fs/Makefile，掌握如何将文件和文件夹按照文件系统的格式写入磁盘，了解文件系统结构的具体细节，学会添加自定义文件进入磁盘镜像。(fsformat.c 中的主函数十分灵活，可以通过修改命令行参数来生成不同的镜像文件)

Exercise 5.4 参照文件系统的设计，完成 fsformat.c 中的 `create_file`函数，并按照个人兴趣完成 `write_directory` 函数（不作为考查点），实现将一个文件或指定目录下的文件按照目录结构写入到 fs/fs.img 的根目录下的功能。

在实现的过程中，你可以将你的实现同我们给出的参考可执行文件 tools/fsformat 进行对比。具体来讲，可以通过 Linux 提供的 xxd 命令将两个 fsformat 产生的二进制镜像转化为可阅读的文本文件，手工进行查看或使用 diff 等工具进行对比。 ■

5.4.3 块缓存

块缓存指的是借助虚拟内存来实现磁盘块缓存的设计。MOS 操作系统中，文件系统服务是一个用户进程（将在下文介绍），一个进程可以拥有 4GB 的虚拟内存空间，将 `DISKMAP` 到 `DISKMAP+DISKMAX` 这一段虚存地址空间 (0x10000000-0x4fffffff) 作为缓冲区，当磁盘读入内存时，用来映射相关的页。`DISKMAP` 和 `DISKMAX` 的值定义在 fs/fs.h 中：

```
1 #define DISKMAP   0x10000000
2 #define DISKMAX   0x40000000
```

Thinking 5.5 请思考，在满足磁盘块缓存的设计的前提下，我们实验使用的内核支持的最大磁盘大小是多少？ ■

为了建立起磁盘地址空间和进程虚存地址空间之间的缓存映射，我们采用的设计如图5.4所示。

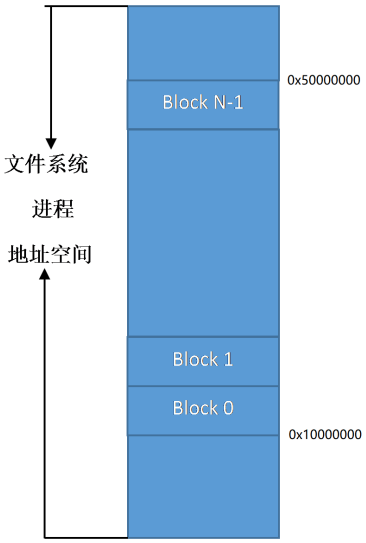


图 5.4: 块缓存示意图

Exercise 5.5 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)。 ■

Thinking 5.6 如果将 DISKMAX 改成 0xC0000000, 超过用户空间，我们的文件系统还能正常工作吗？为什么？ ■

当把一个磁盘块中的内容载入到内存中时，需要为之分配对应的物理内存；当结束使用这一磁盘块时，需要释放对应的物理内存以回收操作系统资源。fs/fs.c 中的 `map_block` 函数和 `unmap_block` 函数实现了这一功能。

Exercise 5.6 实现 `map_block` 函数，检查指定的磁盘块是否已经映射到内存，如果没有，分配一页内存来保存磁盘上的数据。相应地，完成 `unmap_block` 函数，用于解除磁盘块和物理内存之间的映射关系，回收内存。(提示：注意磁盘虚拟内存地址空间和磁盘块之间的对应关系)。 ■

`read_block` 函数和 `write_block` 函数用于读写磁盘块。`read_block` 函数将指定编号的磁盘块读入到内存中，首先检查这块磁盘块是否已经在内存中，如果不在，先分配一页物理内存，然后调用 `ide_read` 函数来读取磁盘上的数据到对应的虚存地址处。

`file_get_block`函数用于将某个指定的文件指向的磁盘块读入内存。其主要分为 2 个步骤：首先为即将读入内存的磁盘块分配物理内存，然后使用`read_block`函数将磁盘内容以块为单位读入内存中的相应位置。这两个步骤对应的函数都借助了系统调用来完成。

在完成块缓存部分之后我们就可以实现文件系统中的一些文件操作了。其中

Exercise 5.7 补全 `dir_lookup` 函数，查找某个目录下是否存在指定的文件。（使用`file_get_block`函数） ■

这里我们给出了文件系统结构中部分函数可能的调用参考 (图5.5)，希望同学们仔细理解每个文件、函数的作用和之间的关系。

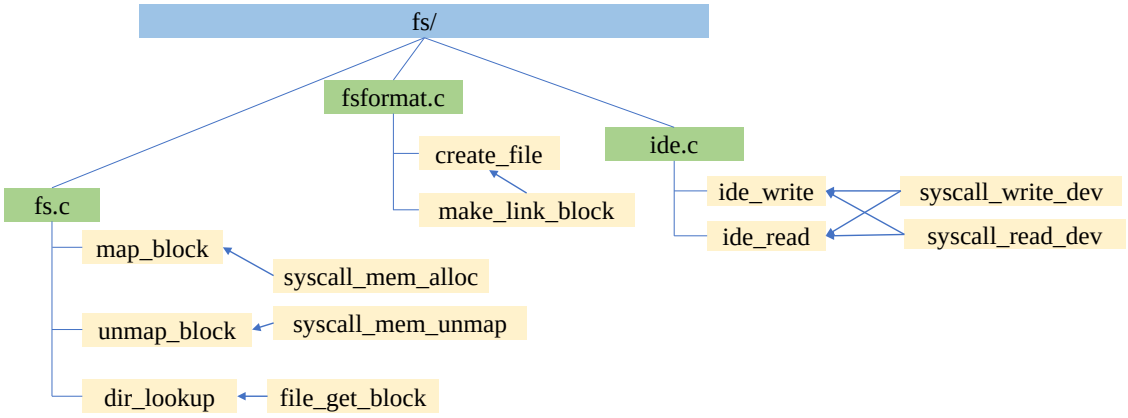


图 5.5: fs/下部分函数调用关系参考

Thinking 5.7 在 lab5 中，fs/fs.h、include/fs.h 等文件中出现了许多宏定义，试列举你认为较为重要的宏定义，并进行解释，写出其主要应用之处。 ■

5.5 文件系统的用户接口

文件系统建立之后，需要向用户提供相关的接口使用。MOS 操作系统内核符合一个典型的微内核的设计，文件系统属于用户态进程，以服务的形式供其他进程调用。这个过程中，不仅涉及了不同进程之间通信的问题，也涉及了文件系统如何隔离底层的文件系统实现，抽象地表示一个文件的问题。首先，我们引入文件描述符（file descriptor）

作为用户程序管理、操作文件资源的方式。

5.5.1 文件描述符

Note 5.5.1 UNIX/LINUX 操作系统中的文件描述符：fd 即 File Discripter，是 UNIX/LINUX 系统给用户提供的 0-255 的整数，用于其在描述符表 (Descriptor Table) 中进行索引。我们在作为操作系统的使用者进行文件 I/O 编程时，使用 `open` 在描述符表的指定位置存放被打开文件的信息；使用 `close` 将描述符表中指定位置的文件信息释放；在 `write` 和 `read` 时修改描述符表指定位置的文件信息。这里的“指定位置”即文件描述符 fd。在这一环节中，你将作为操作系统的设计者，体会和参与构建文件描述符背后的精巧结构。

当用户进程试图打开一个文件时，需要一个文件描述符来存储文件的基本信息和用户进程中关于文件的状态；同时，文件描述符也起到描述用户对于文件操作的作用。当用户进程向文件系统发送打开文件的请求时，文件系统进程会将这些基本信息记录在内存中，然后由操作系统将用户进程请求的地址映射到同一个物理页上，因此一个文件描述符至少需要独占一页的空间。当用户进程获取了文件大小等基本信息后，再次向文件系统发送请求将文件内容映射到指定内存空间中。

Thinking 5.8 阅读 user/file.c ，你会发现很多函数中都会将一个 struct Fd * 型的指针转换为 struct Filefd * 型的指针，请解释为什么这样的转换可行。 ■

Exercise 5.8 完成 user/file.c 中的 `open` 函数。(提示：若成功打开文件，则该函数返回文件描述符的编号)。 ■

当要读取一个大文件中间的一小部分内容时，从头读到尾是极为浪费的，因此需要一个指针帮助我们在文件中定位，在 C 语言中拥有类似功能的函数是 `fseek` 。而在读写期间，每次读写也会更新该指针的值。请自行查阅 C 语言有关文件操作的函数，理解相关概念。

Exercise 5.9 参考 user/fd.c 中的 `write` 函数，完成 `read` 函数。 ■

Thinking 5.9 在 lab4 的实验中我们实现了极为重要的 fork 函数。那么 fork 前后的父子进程是否会共享文件描述符和定位指针呢？请在完成上述练习的基础上编写一个程序进行验证。 ■

Thinking 5.10 请解释 Fd, Filefd, Open 结构体及其各个域的作用。比如各个结构体会在哪些过程中被使用，是否对应磁盘上的物理实体还是单纯的内存数据等。说明形式自定，要求简洁明了，可大致勾勒出文件系统数据结构与物理实体的对应关系与设计框架。 ■

5.5.2 文件系统服务

MOS 操作系统中的文件系统服务通过 IPC 的形式供其他进程调用，进行文件读写操作。具体来说，在内核开始运行时，就启动了文件系统服务进程 `ENV_CREATE(fs_serv)`，用户进程需要进行文件操作时，使用 `ipc_send/ipc_recv` 与 `fs_serv` 进行交互，完成操作。在文件系统服务进程的主函数 serv.c/umain 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。

下图以 UML 时序图的形式在宏观层面上展示了一个用户进程请求文件系统服务的过程 (以`open`为例)。其中 `user_env` 所加载的程序不仅可以是实验源码已给出的 `fstest.c`，也可以是你自己创建的一个以 `u_main` 为入口函数的源码，你可以通过这种方式对自己的文件系统服务进行测试。其中 IPC 系统调用的细节请参考 lab5 的相关内容。(三种颜色不仅区分三个不同的进程，也表示进程执行的代码在我们的操作系统被载入内存前所处的文件位置)

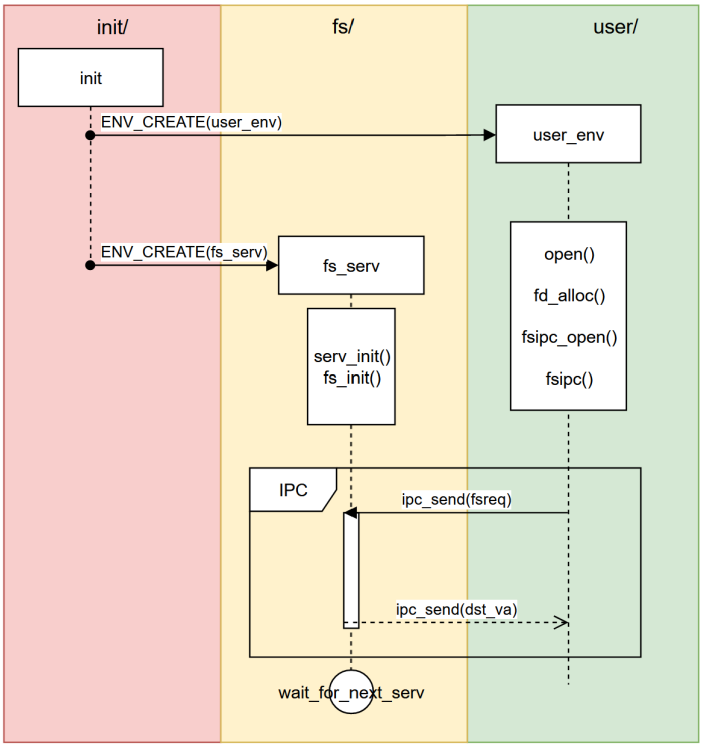


图 5.6: 文件系统服务时序图

Thinking 5.11 上图中有多种不同形式的箭头，请结合 UML 时序图的规范，解释这些不同箭头的差别，并思考我们的操作系统是如何实现对应类型的进程间通信的。■

Thinking 5.12 阅读serv.c/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：

```
#define FSREQ_OPEN      1
#define FSREQ_MAP       2
#define FSREQ_SET_SIZE  3
#define FSREQ_CLOSE     4
#define FSREQ_DIRTY     5
#define FSREQ_REMOVE    6
#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文件操作（文件的增、删、改、查等），将结果重新通过 IPC 反馈给用户程序。

Exercise 5.10 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。 ■

这里我们给出了文件系统的用户接口中部分函数可能的调用参考 (图5.7)，希望同学们体会函数之间的调用关系、理解文件系统中用户接口的实现过程。

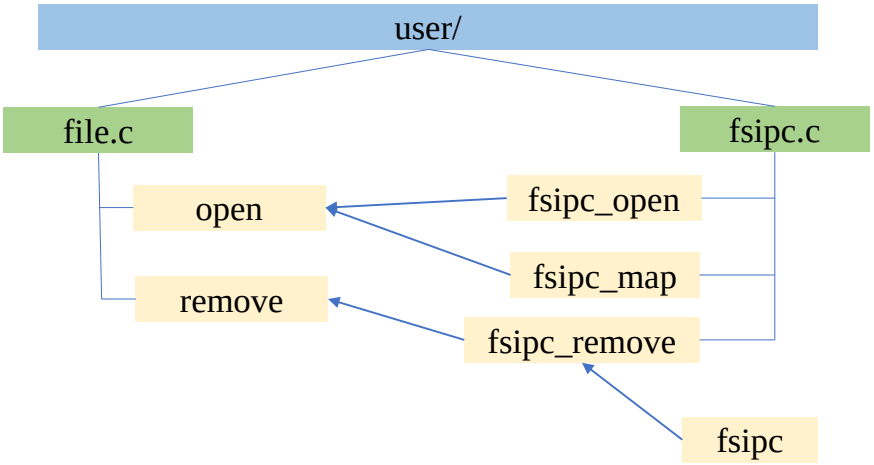


图 5.7: user/下部分函数调用关系参考

5.6 正确结果展示

Note 5.6.1 使用 /OSLAB/gxemul -E testmips -C R3000 -M 64 -d gxemul/fs.img elf-file 运行 (其中 elf-file 是你编译生成的 vmlinux 文件的路径)。

本实验有三个阶段性测试可以检验你实现的文件系统的正确性。

5.6.1 IDE 磁盘交互

完成了 Exercise 5.1-5.2 后，在 init/init.c 中仅启动 devtst 进程，将对 console 和 IDE 磁盘交互进行测试。

```
1     ENV_CREATE(user_devtst);
```

运行 gxemul 后，输入一个字符串'teststring'。

```
1   cons test, please input a string:
2
3   teststring
4   end of devtst
5   IDE test
6   dev address is ok
```

5.6.2 文件系统测试

完成了 Exercise 5.3-5.7 后，在 init/init.c 中仅启动文件系统服务进程，该进程在正式开始服务前，会进行 fs/test.c 中测试。

```
1   ENV_CREATE(fs_serv);
```

```
1   FS is running
2   FS can do I/O
3   superblock 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
```

5.6.3 文件系统服务测试

完成全部 Exercise 后，在 init/init.c 中启动一个 fstest 进程和文件系统服务进程：

```
1     ENV_CREATE(user_fstest);
2     ENV_CREATE(fs_serv);
```

就能开始对文件系统的检测，运行文件系统服务，等待应用程序的请求。注意：我们此时必须将文件系统进程作为第二个进程启动，其原因是我们在 user/fsipc.c 中定义的文件系统 ipc 请求的目标指定为第二个创建的进程。

```
FS is running
FS can do I/O
superblock is good
diskno: 0
diskno: 0
read_bitmap is good
diskno: 0
alloc_block is good
file_open is good
file_get_block is good
file_flush is good
file_truncate is good
diskno: 0
file rewrite is good
serve_open 00000800 ffff000 0x2
open is good
read is good
diskno: 0
serve_open 00000800 ffff000 0x0
open again: OK
read again: OK
file rewrite is good
file remove: OK
```

5.7 任务列表

- 完成 sys_write_dev 和 sys_read_dev
- 完成 fs/ide.c
- 完成 free_block
- 完成 create_file
- 完成 diskaddr
- 完成 map_block 和 unmap_block
- 完成 dir_lookup
- 完成 open
- 完成 read
- 实现删除指定路径的文件的功能

5.8 实验思考

- Unix /proc 文件系统
- 设备操作与高速缓存
- 文件控制块与 inode 对比

- 单个文件的最大体积、一个磁盘块最多存储的文件控制块及一个目录最多子文件
- 磁盘最大容量
- 磁盘映射与用户空间
- 文件系统进程中宏定义理解
- struct Fd * 到 struct Filefd * 的转换
- 文件描述符与 fork 函数
- 文件系统用户接口中的结构体
- 解释时序图，思考进程间通信
- 文件系统服务进程运行机制

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. 实现基本 shell
5. 实现 shell 中涉及管道的部分

6.2 管道

在 lab4 中，我们已经学习过一种进程间通信 (IPC,Inter-Process Communication) 的方式——共享内存。而今天我们要学的管道，其实也是进程间通信的一种方式。

6.2.1 初窥管道

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：

```
#include<unistd.h>

int pipe(int fd[2]); // 成功返回 0，否则返回-1；

// 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，我们先来做实验亲自体会一下[1]

[1]实验代码参考 http://pubs.opengroup.org/onlinepubs/9699919799/functions/pipe.html

Listing 16: 管道示例

```c
#include <stdlib.h>
#include <unistd.h>

int fildes[2];
/* buf size is 100 */
char buf[100];
int status;

int main(){

    status = pipe(fildes);

    if (status == -1 ) {
        /* an error occurred */
        printf("error\n");
    }


    switch (fork()) {
    case -1: /* Handle error */
        break;


    case 0:  /* Child - reads from pipe */
        close(fildes[1]);                       /* Write end is unused */
        read(fildes[0], buf, 100);              /* Get data from pipe */
        printf("child-process read:%s",buf);    /* Print the data */
        close(fildes[0]);                       /* Finished with pipe */
        exit(EXIT_SUCCESS);


    default:  /* Parent - writes to pipe */
        close(fildes[0]);                       /* Read end is unused */
        write(fildes[1], "Hello world\n", 12);  /* Write data on pipe */
        close(fildes[1]);                       /* Child will see EOF */
        exit(EXIT_SUCCESS);
    }
}
```

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 17: 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[2]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：

[2]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。

```
int
pipe(int pfd[2])
{
    int r, va;
    struct Fd *fd0, *fd1;

    if ((r = fd_alloc(&fd0)) < 0
    ||  (r = syscall_mem_alloc(0, (u_int)fd0, PTE_V|PTE_R|PTE_LIBRARY)) < 0)
    goto err;

    if ((r = fd_alloc(&fd1)) < 0
    ||  (r = syscall_mem_alloc(0, (u_int)fd1, PTE_V|PTE_R|PTE_LIBRARY)) < 0)
    goto err1;

    va = fd2data(fd0);
    if ((r = syscall_mem_alloc(0, va, PTE_V|PTE_R|PTE_LIBRARY)) < 0)
    goto err2;
    if ((r = syscall_mem_map(0, va, 0, fd2data(fd1), PTE_V|PTE_R|PTE_LIBRARY)) < 0)
    goto err3;

    ...
}
```

在 pipe 中，首先分配两个文件描述符 fd0,fd1 并为其分配空间，然后给 fd0 对应的虚拟地址分配一页物理内存，再将 fd1 对应的虚拟地址映射到相同的物理内存。这一页上存的就是我们后面要讲的 pipe 结构体，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。Hint: 修改 fork.c 中的 duppage 函数 ■

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：

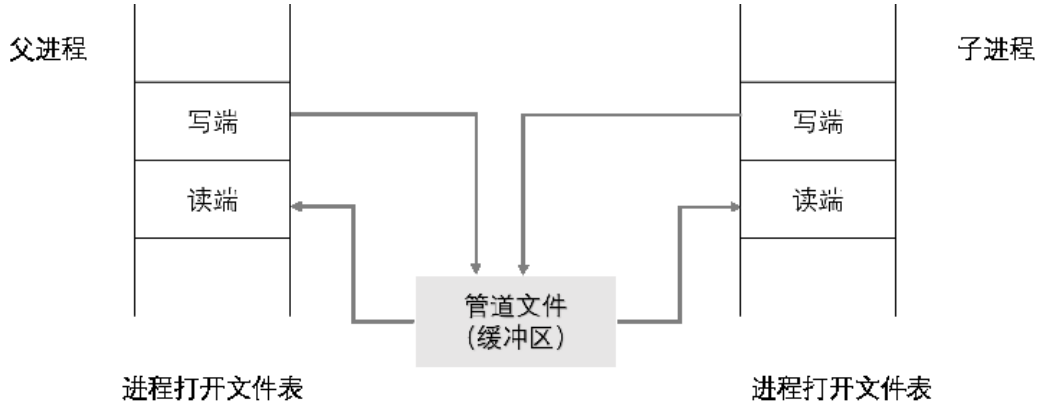


图 6.1: 父子进程与管道缓冲区

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。

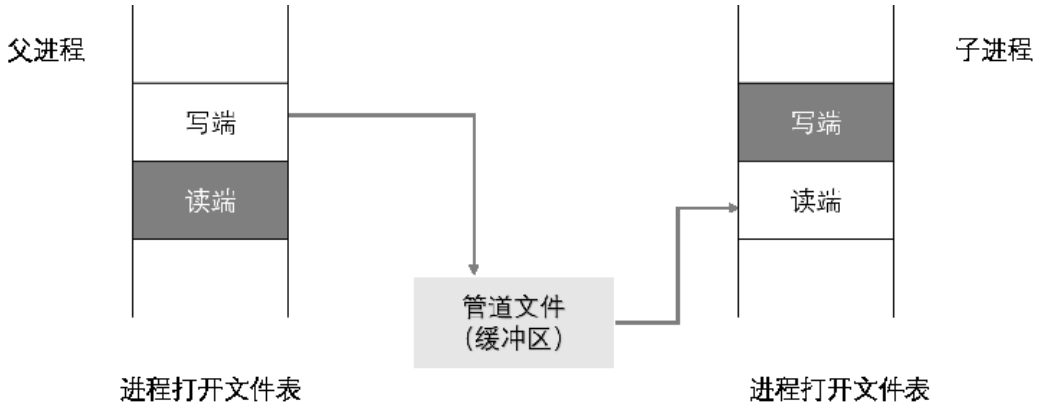


图 6.2: 关闭不使用的端口后

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：

```
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 给出了下一个将要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(32 Byte) 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 管道的关闭涉及到以下几个函数：fd.c 中的 close,fd_close 以及 pipe.c 中的 pipeclose。

Note 6.2.2 如果管道的写端相关的所有的文件描述符都已经关闭，那么管道读端将会读到文件结尾并返回 0。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：

pageref(rfd) + pageref(wfd) = pageref(pipe)

Note 6.2.3 内核会对 pages 数组成员维护一个页引用变量 pp_ref 来记录指向该物理页的虚页数量。pageref 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 6.2.4 注意在本次实验中由于文件系统服务所在进程已经默认为 1 号进程 (起始进程为 0 号进程)，在测试时想启用文件系统需要注意 ENV_CREATE(fs_serv) 在 init.c 中的位置。

6.2.4 管道的竞争

我们的 MOS 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。

回顾`_pipeisclosed`函数。在这个函数中我们对 `pageref(fd)` 与 `pageref(pipe)` 进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:

```
    pipe(p);
    if(fork() == 0 ){
        close(p[1]);
        read(p[0],buf,sizeof buf);
    }else{
        close(p[0]);
        write(p[1],"Hello",5);
    }
```

- 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会问：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 MOS 操作系统中，只有 syscall_ 开头的**系统调用函数**是原子操作，其他所有包括 fork 这些函数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)。但是由于进程切换的存在，我们无法确保 **当 pageref(p[0]) == pageref(pipe) 时，写端关闭**。正面如果不好解决问题，我们可以考虑从其逆否命题着手，即我们要确保：当写端没有关闭的时候，pageref(p[0]) ≠ pageref(pipe)。

我们考虑之前那个预想之外的情景，它出现的最关键原因在于：pipe 的引用次数总比 fd 要高。当管道的 close 进行到一半时，**若先解除 pipe 的映射，再解除 fd 的映射**，就会使得 pipe 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，在 fd.c 中有一个 dup 函数，用于复制文件内容。试想，如果要复制的是一个管道，那么是否会出现与 close 类似的问题？请模仿上述材料写写你的理解。

■

根据上面的描述我们其实已经能够得出一个结论：控制 fd 与 pipe 的 map/unmap 的顺序可以解决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 pipeclose 函数中的 unmap 顺序与 user/fd.c 中 dup 函数中的 map 顺序以避免上述情景中的进程竞争情况。 ■

6.2.5 管道的同步

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，但是我们还得解决第二个问题：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。

上述现象的根源是什么？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

首先恭喜屏幕前的你能够读到这里，你的 OS 大厦即将落成，但所谓“行百里者半九十”，也许你此时会觉得整个系统的代码过多，OS 的大厦摇摇欲坠，每一次 DEBUG 都因找不到问题出处而心力憔悴。当你满怀希望地 make run，却发现屏幕上无限循环的 page_out 或一行简洁的 user_panic，是否让你欲哭无泪。要知道，坚持就是胜利！我们能做的，就是尽可能地引导大家正确思考，并“正确地”实现接下来的部分–shell。

什么是 shell？在计算机科学中，Shell 俗称壳（用来区别于核），是指“为使用者提供操作界面”的软件（命令解析器）。它接收用户命令，然后调用相应的应用程序。基本上 shell 分两大类：

一是图形界面 shell（Graphical User Interface shell 即 GUI shell）。例如：应用最为广泛的 Windows Explorer （微软的 windows 系列操作系统），还有也包括广为人知的 Linux shell，其中 linux shell 包括 X window manager (BlackBox 和 FluxBox)，以及功能更强大的 CDE、GNOME、KDE、XFCE。

二：命令行式 shell（Command Line Interface shell ，即 CLI shell），也就是我们 MOS 操作系统最后即将实现的 shell 模式。

常见的 shell 命令在 lab0 已经介绍过了，这里就不赘述，接下来让我们一步一步揭开 shell 背后的神秘面纱。

6.3.1 完善 spawn 函数

spawn 的汉译为“产卵”，其作用是帮助我们调用文件系统中的可执行文件并执行。

spawn 的流程可以分解如下：

- 从文件系统打开对应的文件 (2 进制 ELF，在我们的 OS 里是 *.b)。
- 申请新的进程描述符;
- 将目标程序加载到子进程的地址空间中，并为它们分配物理页面;
- 为子进程初始化堆、栈空间，并设置栈顶指针，以及重定向、管道的文件描述符，对于栈空间，因为我们的调用可能是有参数的，所以要将参数也安排进用户栈中。
- 设置子进程的寄存器 (栈寄存器 sp 设置为 esp。程序入口地址 pc 设置为 UTEXT)
- 将父进程的共享页面映射到子进程的地址空间中。
- 这些都做完后，设置子进程可执行。

在动手填写 spawn 函数前，我们希望你:

- 认真回看 lab5 文件系统相关代码，弄清打开文件的过程。
- 思考如何读取 elf 文件或者说如何知道二进制文件中 text 段

关于后一个问题，各位已经在 lab3 中填写了 load_icode 函数，实现了 elf 中读取数据并写入内存空间。而 text 段的位置，可以借助 readelf 的帮助（这个命令各位应该不陌生)。

为了确保各位的大楼经得起反复折腾，我们来探讨下面的问题，以检验大家的基本功，以及前几个 lab 是否做得 “到位”:

(如果不到位，请回炉重看一遍并加以理解)

- 在 lab1 中我们介绍了 data text bss 段及它们的含义，data 存放初始化过的全局变量，bss 存放未初始化的全局变量。关于 memsize 和 filesize ，我们在 Note 1.3.5中也解释了它们的含义与特点。关于 1.3.5，注意其中关于 “bss 并不在文件中占数据” 的表述。
- 在 lab3 中我们创建进程，并且在内核态加载了初始进程（ENV_CREATE(...) ），而我们的 spawn 函数则是通过和文件系统交互，取得文件描述块，进而找到 elf 在 “硬盘” 中的位置，进而读取。

在 lab3 中我们要填写 load_icode_mapper 函数，分为两部分填写，第二部分则是处理 msize 和 fsize 不相等时的情况。那么问题来了:

Thinking 6.5 bss 在 ELF 中并不占空间，但 ELF 加载进内存后，bss 段的数据占据了空间，并且初始值都是 0。请回答你设计的函数是如何实现上面这点的? ■

如果你回看了上面提到的指导书内容，答案应该是 “显而易见的”。如果还不太理解，说明你需要回炉重看一遍。

关于如何为子进程初始化栈空间，请仔细阅读 `init_stack` 函数。

因为我们无法直接操作子进程的栈空间，所以该函数首先将需要准备的参数填充到本进程的 TMPPAGE 这个页面处，然后将 TMPPAGE 映射到子进程的栈空间中。

首先将 argc 个字符串填到栈上，并且不要忘记在每个字符串的末尾要加上 '\0' 表示结束，然后将 argc+1 个指针填到栈上，第 argc+1 个指针指的是一个空字符串表示参数的结束。

最后将 argc 和 argv 填到栈上，argv 将指向那 argc+1 个字符指针。

这里给出一张 spawn 准备的栈空间的示意图。

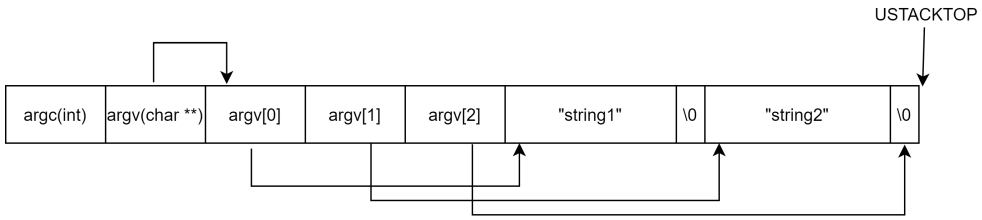


图 6.3: 子进程栈空间示意图

6.3.2 解释 shell 命令

接下来，我们将通过一个实例，再次吸收理解上面 lab1、lab3 联动的内容。

```
#include "lib.h"
#define ARRAYSIZE (1024*10)
int bigarray[ARRAYSIZE]={0};
void
umain(int argc, char **argv)
{
    int i;

    writef("Making sure bss works right...\n");
    for(i = 0;i < ARRAYSIZE; i++)
        if(bigarray[i]!=0)
            user_panic("bigarray[%d] isn't cleared!\n",i);
    for (i = 0; i < ARRAYSIZE; i++)
        bigarray[i] = i;
    for (i = 0; i < ARRAYSIZE; i++)
        if (bigarray[i] != i)
            user_panic("bigarray[%d] didn't hold its value!\n", i);
    writef("Yes, good. Now doing a wild write off the end...\n");
    bigarray[ARRAYSIZE+1024] = 0;
    userpanic("SHOULD HAVE TRAPPED!!!");
}
```

在 user/文件夹下创建上面的文件，并在 Makefile 中添加相应信息，使得生成相应的.b 文件，在 init/init.c 中创建相应的初始进程，观察相应的实验现象。(具体可以参考 lab4 中 pingpong 和 fktest 是如何添加到 makefile 中的) 如果能正确运行，则说明我们的 load_icode 系列函数正确地保证了 bss 段的初始化。

使用 readelf 命令解析我们的 testbss.b 文件，看看 bss 段的大小并分析其原因。学习并使用 size 命令，看看我们的 testbss.b 文件的大小。

修改代码，将数组初始化为 array[SIZE]=0; 重新编译（记得常 make clean），再次分析 bss 段。再次使用 size 命令。

再次修改代码，将数组初始化为 array[SIZE]=1; 再次重新编译，再次分析。

在 Lab5 中我们实现了文件系统，lab6 的 shell 部分我们提供了几个可执行二进制文件，模拟 linux 的命令：ls.b cat.b .....。上面提到的 spawn 函数实现方法，是打开相应的文件并执行。请你思考我们是如何将文件 “烧录” 到 fs.img 中的（阅读 fs/Makefile）。如果你并没有看懂我这段话，请回看 Exercise 5.4

你可以尝试将生成的 testbss.b 加载进 fs.img 中

这节 “补课” 下课以后，大家再去补全 spawn 函数，相信能如鱼得水了。

Exercise 6.5 根据以上描述以及注释，补充完成 user/spawn.c 中的 `int spawn(char *prog, char **argv)`。 ■

Thinking 6.6 为什么我们的 *.b 的 text 段偏移值都是一样的，为固定值？ ■

Thinking 6.7 在 shell 中执行的命令分为内置命令和外部命令。在执行内置命令时 shell 不需要 fork 一个子 shell，如 Linux 系统中的 cd 指令。在执行外部命令时 shell 需要 fork 一个子 shell，然后子 shell 去执行这条命令。

据此判断，在 MOS 中我们用到的 shell 命令是内置命令还是外部命令？请思考为什么 Linux 的 cd 指令是内部指令而不是外部指令？ ■

接下来，我们需要在 shell 进程里实现对管道和重定向的解释功能。解释 shell 命令时：

1. 如果碰到重定向符号‘<’或者’>’，则读下一个单词，打开这个单词所代表的文件，然后将其复制给标准输入或者标准输出。
2. 如果碰到管道符号‘|’，则首先需要建立管道 pipe，然后 fork。
   - 对于父进程，需要将管道的写者复制给标准输出，然后关闭父进程的读者和写者，运行‘|’左边的命令，获得输出，然后等待子进程运行。
   - 对于子进程，将管道的读者复制给标准输入，从管道中读取数据，然后关闭子进程的读者和写者，继续读下一个单词。

在这里可以举一个使用管道符号的例子来方便大家理解，相信大家都使用过 linux 中的 ps 指令，也就是最基本的查看进程的命令，而直接使用 ps 会看到所有的进程，为了更方便的追踪某个进程，我们通常使用 ps aux|grep xxx 这条指令，这就是使用管道的例子，ps aux 命令会将所有的进程按格式输出，而 grep xxx 命令作为子进程执行，所有的进程作为他的输入，最后的输出将会筛选出含有 xxx 字符串的进程展示在屏幕上。

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

通过阅读代码空白段的注释我们知道，将文件复制给标准输入或输出，需要我们将其 dup 到 0 或 1 号文件描述符 (fd). 那么问题来了：

Thinking 6.8 在哪步，0 和 1 被 “安排” 为标准输入和标准输出？请分析代码执行流程，给出答案。 ■

在 `spawn` 函数中标记为 Share memory 一段的作用。这个函数将父进程所有的共享页面映射给了子进程。

想一下，进程空间中的哪些内存是共享内存？

在进程空间中，文件、管道、控制台以及文件描述符都是以共享页面的方式存在的。有几处通过 spawn 产生新进程的位置。

- init.b 进程通过 spawn 生成 shell 进程。init.b 申请了 console（控制台）作为标准输入输出，而这个 console 就是通过共享页面映射给 shell 进程，使得 shell 进程可以通过控制台与用户交互。
- 子 shell 进程负责解析命令行命令，并通过 spawn 生成可执行程序进程（对应 *.b 文件）。在解析命令行的命令时，子 shell 会将重定向的文件及管道等 dup 到子 shell 的标准输入或输出，然后 spawn 时将标准输入和输出通过共享内存映射给可执行程序，所以可执行程序可以从控制台、文件和管道等位置输入和输出数据。

Note 6.3.1 我们的测试进程从 user/icode 开始执行，里面调用了 spawn(init.b), 在完成了 spawn 后，创建了 init.b 进程。init.b 进程调用 spawn(sh.b)，创建了 sh.b 进程，也就是我们的 shell。

Thinking 6.9 在你的 shell 中输入指令 ls.b | cat.b > motd。

- 请问你可以在你的 shell 中观察到几次 spawn？分别对应哪个进程？
- 请问你可以在你的 shell 中观察到几次进程销毁？分别对应哪个进程？

■

6.4 实验正确结果

6.4.1 管道测试

管道测试有三个文件,分别是 user/testpipe.c 、user/testpiperace.c 和 user/testelibrary.c ，以合适的次序建好进程后，在 testpipe 的测试中若出现两次 **pipe tests passed** 即说明测试通过。

testpipe 本地测试部分运行结果如图:

在 testpiperace 的测试中应当出现 race didn’t happen 是正确的。

在 testpiperace 中，子进程多次查询管道是否关闭。而父进程不停地执行 dup 函数，dup 操作主要包含两个操作，分别是关闭管道（close）和将旧文件映射到新的文件描述符。如果这两个操作中有任意一个操作产生竞争，将可能导致子进程认为写入端关闭。

testpiperace 本地测试部分运行结果如图:

图 6.4: 管道测试 1

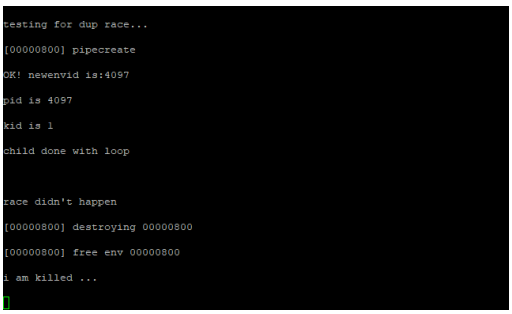


图 6.5: 管道测试 2

在 user/testptelibrary.c 的测试中，如果 fork 和 spawn 对于共享页面的处理均可得当，即可说明测试正确。

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：

```
        ENV_CREATE(user_icode);
        ENV_CREATE(fs_serv);
```

如果正常会看到如下现象：

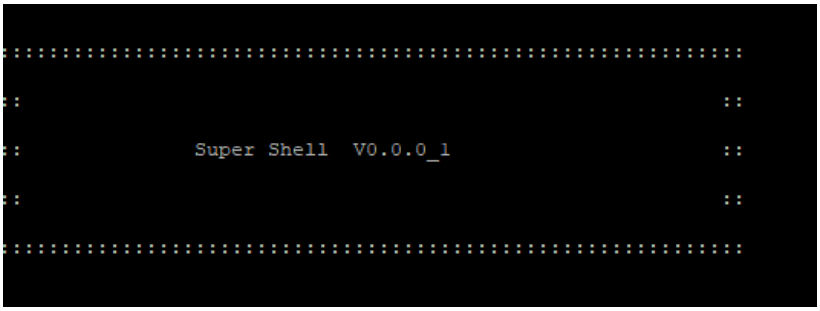


图 6.6: shell 展示界面

> **Note 6.4.1** Shell 部分评测提交，最好注释掉 writef(“:::···supershell···”) 部分内容。

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;

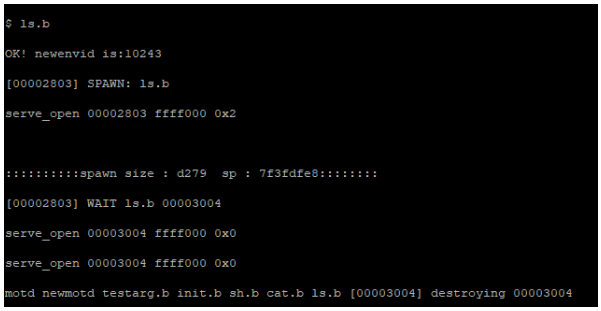


图 6.7: ls 结果

- 输入 cat.b，会有回显现象出现;

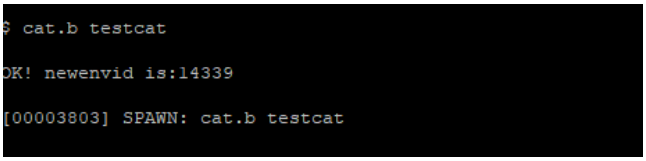


图 6.8: cat 结果

- 输入 ls.b | cat.b, 和 ls.b 的现象应当一致;

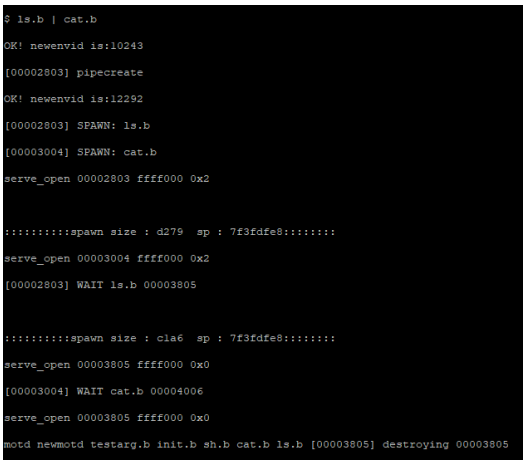


图 6.9: lscat 结果

Note 6.4.2 课程网站上有对于测试文件的解析视频，大家可以移步网站观看。

6.5 任务列表

- **修改 duppage 函数**

- 填写 piperead, pipewrite, _pipeisclosed 函数
- 修改 pipeclose 和 dup 函数
- 修改 _pipeisclosed 函数
- 完成 spawn 函数
- 完成 runcmd 函数

6.6 实验思考

- 思考-父进程为读者
- 思考-dup 中的进程竞争
- 思考-原子操作
- 思考-解决进程竞争
- 思考-如何实现加载 bss 段
- 思考-为什么.b 文件的 text 段偏移值为固定值
- 思考-内置命令与外部命令
- 思考-标准输入和标准输出
- 思考-解释指令执行的现象
- 思考-解释 shell 的等待关系