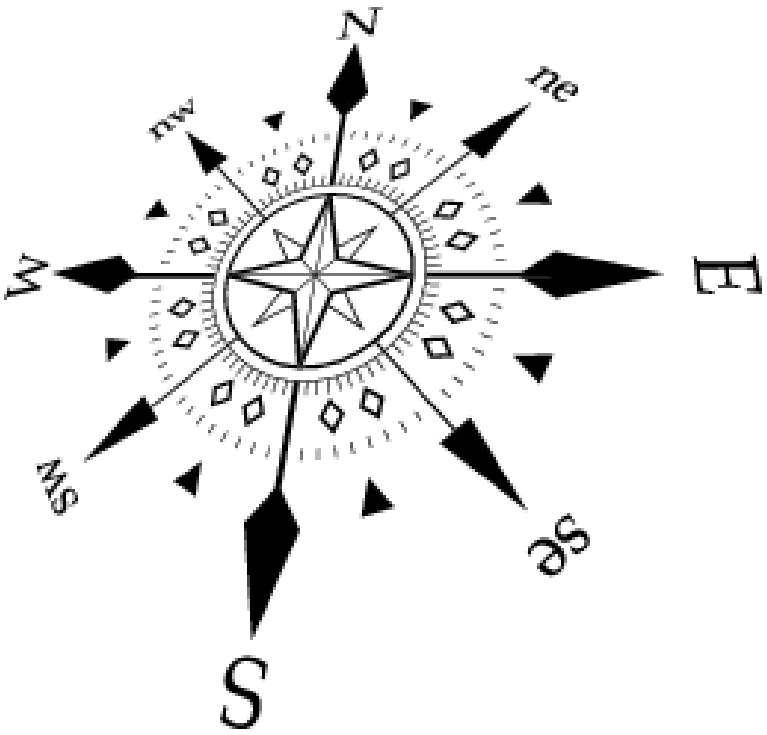
MOS 操作系统实验
指导书

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

2026 年 3 月 6 日

Powered by LaTeX

目录

引言

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

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

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

实验内容

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

1. **内核、启动和 `printf`:** 通过 PC 启动的实验，掌握硬件的启动过程，理解链接地址、加载地址和重定位的概念，学习如何编写裸机代码。

2. **内存管理:** 理解虚拟内存和物理内存的管理，实现操作系统对虚拟内存空间的管理。

3. **进程与异常:** 通过设置进程控制块和编写进程创建、进程中止和进程调度程序，实现进程管理；编写通用中断分派程序和时钟中断例程，实现中断管理。

4. **系统调用与 `fork`:** 掌握系统调用的实现方法，理解系统调用的处理流程，实现本实验所需的系统调用。

5. **文件系统:** 通过实现一个简单的、基于磁盘的、微内核方式的文件系统，掌握文件系统的实现方法和层次结构。

6. **管道与 shell:** 实现具有管道，重定向功能的命令解释程序 shell，能够执行一些简单的命令。最后将 6 部分链接起来，使之成为一个能够运行的操作系统。

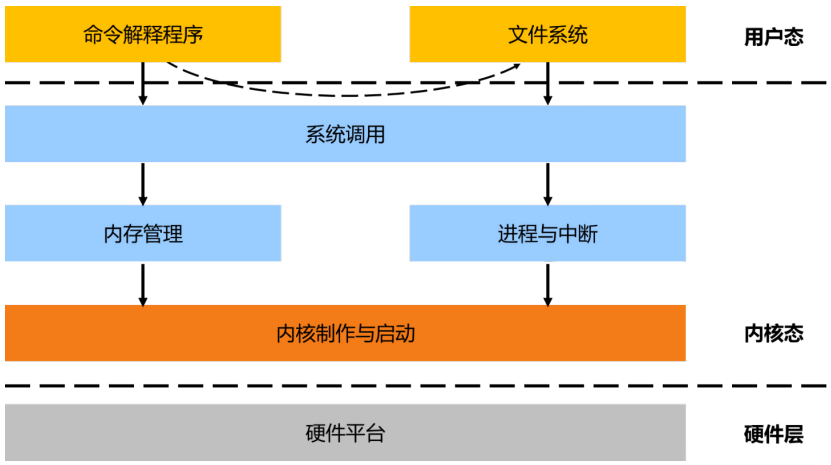


图 1: 实验内容的关系

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

实验设计

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

1. 每个实验可独立运行与测试，便于调试与评测，可获得阶段性成果。

2. 每个实验内容包含相对独立的知识点，并只依赖其前序实验。

3. 基本保证在两周内完成一个实验，这样在一学期内可以完成整个实验。

4. 各个实验提交的代码一直伴随整个实验过程，可以不断改进、完善代码。

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

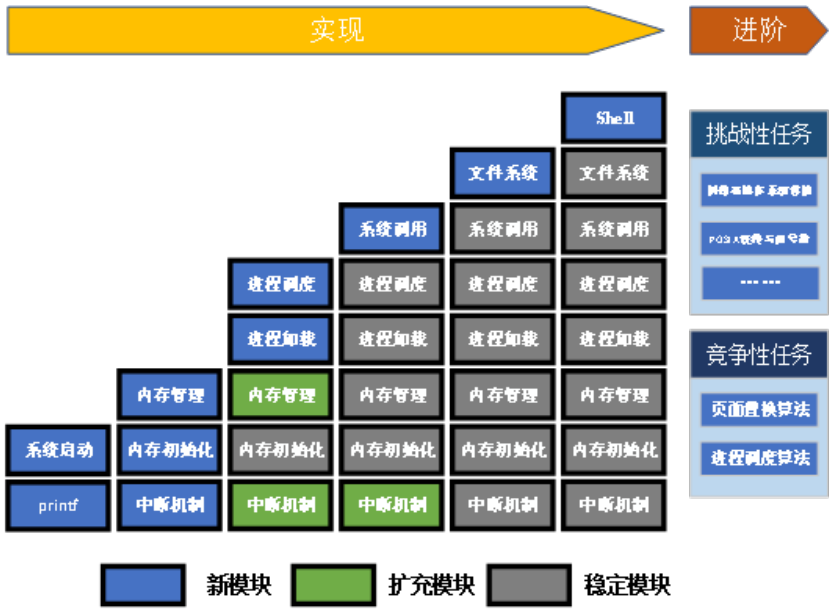


图 2: 增量式实验方法

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

- 第一层次，掌握基本的系统使用与编程能力：包括`Linux`、`Makefile`、`git`、`vi` 和仿真器的使用，基本的 shell 编程和使用系统调用编程；

- 第二层次，掌握操作系统核心能力：包括 6 个实验，从操作系统内核构造、内存管理、进程管理、系统调用、文件系统和命令解释程序，构成一个完整的小型操作系统；

- 第三层次，锻炼操作系统提升能力：主要包括若干挑战性任务，学生需要独立在某一方面实现若干新的系统功能。

实验环境

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

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

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

b) **git 服务器**，包括学生各个实验的代码以及相关信息。

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

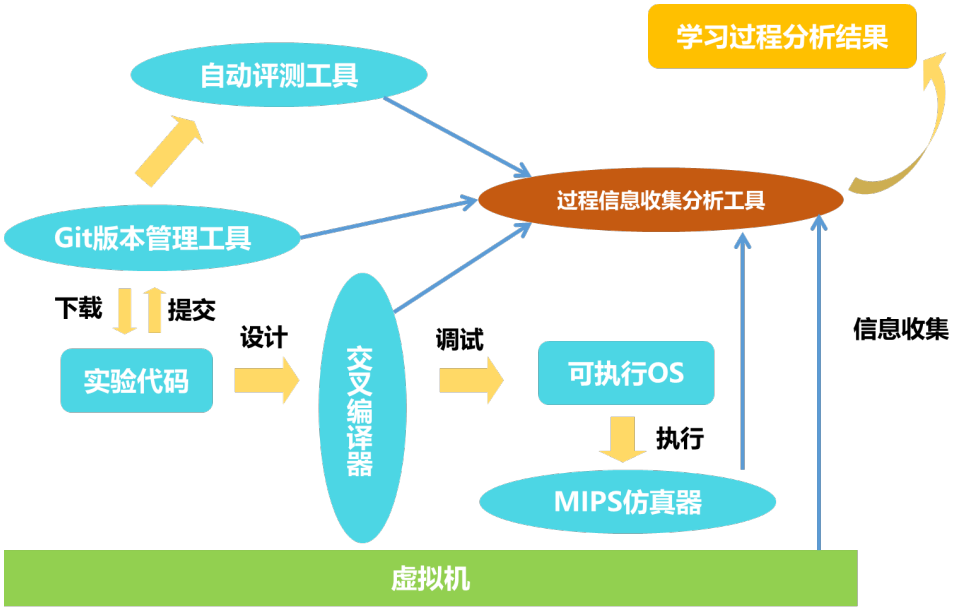


图 3: 操作系统实验集成环境

虚拟机平台

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

Git 服务器

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

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

自动评测

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

CHAPTER 0

初识操作系统

0.1 实验目的

1. 熟悉操作系统实验环境。
2. 掌握操作系统实验所需的基本工具。

在本实验中，我们需要熟悉实验环境，了解 GNU/Linux 操作系统（Ubuntu），并掌握一些常用命令行工具，最终能够脱离图形化界面进行工作。本实验难度不大，重点是熟悉操作系统实验环境的各类工具，为后续实验的开展奠定基础。

0.2 初识实验

“工欲善其事必先利其器”，我们应对环境和工具有足够的了解，才能顺利开始实验工作。

0.2.1 了解实验环境

实验环境主要配置如下：

- 操作系统：Ubuntu
- 编译工具链：GCC + Binutils
- 调试工具链：GDB + QEMU
- 构建工具：Make
- 版本控制工具：Git

Ubuntu 是一款 GNU/Linux 操作系统，是目前较为流行的 Linux 发行版之一。GNU（GNU is Not Unix 的递归缩写）是一个类 Unix 操作系统，包括 GNU 软件包（专门由

GNU 项目发布的程序) 和由第三方发布的自由软件，其计划中的 Hurd 内核仍在继续开发，目前所用的典型内核是 Linux。而人们常说的 Linux 严格意义上是指 Linux 内核，它仅是用于资源分配和硬件管理的核心程序，与不同的软件组合进行定制就构成了不同的 Linux 发行版。目前常见的 Linux 发行版都是 GNU 软件和 Linux 内核的组合，因此该组合叫做 GNU/Linux，通常具有开源、免费、稳定、多平台等特点。

GCC 是 GNU 项目开发的一款编译器套件，最初名称为 GNU C Compiler，后来因支持更多编程语言而改名为 GNU Compiler Collection。很多集成开发环境（Integrated Development Environment, IDE) 的编译器用的就是 GCC 套件，例如 Dev-C++，Code::Blocks 等。而 Binutils 是 GNU 项目开发的一套程序工具，包含汇编器、链接器、反汇编器等工具，常与 GCC 配合使用。本书实验将使用以 MIPS 为目标平台的 GCC + Binutils 交叉编译工具链。

GDB（GNU Project Debugger）是 GNU 项目开发的一款调试器，它能让你在程序执行过程中查看其内部情况，或者在程序崩溃时查看其当时的运行状态。这些程序可以与 GDB 在同一机器上运行（本地)，也可以在另一台机器（或者模拟器）上运行（远程)。GDB 支持多种编程语言和操作系统。本书实验将使用以 MIPS 为目标平台的 GDB 调试在本地和 QEMU 模拟器上运行的程序。

QEMU（Quick Emulator）是一款通用且开源的机器模拟器和虚拟化软件。作为机器模拟器使用时，QEMU 可以在一台机器（例如你的个人电脑）上运行为另一台机器(例如本实验需要的 MIPS 架构 CPU 、串行终端和 IDE 磁盘控制器）编写的程序。

Make 是一款构建自动化工具，它控制从程序的源文件生成可执行文件和其它文件的过程。Make 通过名为 Makefile 的文件来获取构建规则，包括所有构建目标以及它们的构建方法，自动计算依赖关系并进行增量构建。本书实验将使用 GNU Make（GNU 项目的一部分，Make 工具的事实标准）管理整个工作流程。

Git 是一款免费且开源的分布式版本控制系统，旨在快速高效地处理各种规模的项目。本书实验将利用它来提供实验代码的版本控制、发布和提交等功能。第 0.5 节将会详细介绍 Git 的使用方法。

图 0.1: Ubuntu，GNU，Linux

0.2.2 远程访问实验环境

为了方便学习，本书实验配备了配套实验环境，可通过网站远程访问，无需本地再进行复杂配置。

本书实验环境的网站是 http://lab.os.buaa.edu.cn，在右上角点击“Web 终端”按钮即可使用。

0.2.3 实验环境的操作界面

目前主流的桌面操作系统，如 Windows、Mac OS 和 Ubuntu 等主要使用图形用户界面（Graphical User Interface, GUI）。但是在操作系统课程中，需要使用命令行界面（Command Line Interface, CLI）。

GUI 允许用户使用鼠标等输入设备操纵屏幕上的图标或菜单选项，而 CLI 则要求用户通过键盘输入文本或字符命令来与计算机进行交互。如在 Windows 中，资源管理器（explorer）提供 GUI，命令提示符（cmd）则提供 CLI。

虽然 GUI 更为直观和易用，但它需要更多的系统资源，并且难以实现自动化。相比之下，CLI 更为高效和灵活，天然适合批量执行复杂的自动化任务，是开发人员和系统管理员的首选。

0.3 基础操作介绍

0.3.1 shell 命令

严格意义上讲，用户并不直接接触“操作系统”，尤其无法直接操作其“内核”（kernel），而主要与它最外层的“壳”（shell）进行交互。shell 负责解析用户输入的指令，并将指令传递给操作系统进行处理，最后将处理结果反馈给用户。

人们常说的 shell 一般是指 CLI shell。在 Ubuntu 中，默认的 shell 是 Bash（Bourne-Again Shell），它也是 GNU 项目的一部分，在兼容 sh 标准的基础上提供了许多增强功能，是目前最流行的 shell 之一。

shell 命令有两种类型：内建命令和外部命令。内建命令由 shell 本身实现，直接在 shell 中执行；外部命令则指向独立的可执行文件，需要通过 shell 调用来执行。在 Ubuntu 中，最基本的外部命令由核心工具集（Coreutils）提供，它同样是 GNU 项目的一部分。

shell 命令的一般格式为：命令名 [选项] [参数]。其中，方括号内的选项和参数是可选的，用于调整命令的功能和指定操作对象。

下面介绍 Linux 的基本操作，熟悉此部分内容的读者可跳过此节。

0.3.2 Linux 基本操作

进入终端后，首先会看到光标前的如下内容。

```
开始连接到 git@xxx.xxx.xxx.xxx  0.1
Welcome to Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-46-generic x86_64)

        * Documentation:  https://help.ubuntu.com
        * Management:     https://landscape.canonical.com
        * Support:        https://ubuntu.com/advantage
Last login: Wed Jan  4 12:50:58 2023 from 10.134.170.231
git@22xxxxxx:~$
```

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

要想知道当前目录中包含哪些文件，可使用 ls 命令，其输出信息可以通过彩色加亮显示，来区分不同类型的文件，ls 是使用率较高的命令，其详细信息如图所示。若不指定参数，只使用 ls，则默认情况下显示该目录下所有非隐藏文件[1]。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    列出文件的详细信息并且每行只列出一个文件
```

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

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

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

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

```
mkdir
用法: mkdir [选项]... 目录...
```

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

```
cd
用法: cd [选项]... 目录
```

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

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

[1]在 Linux 中，文件名以 . 开头的文件为隐藏文件，目录同理。

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

输入 `rmdir` 可以删除一个空的目录。

```
rmdir
用法:rmdir [选项]... 目录...
```

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

```
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，删除一个有内容的目录
                    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` 命令还有 `-i` 选项，这个选项在使用文件扩展名字符删除多个文件时特别有用。使用这个选项，系统会要求逐一确定是否要删除。这时，必须输入 `Y` 并按回车键，才能删除文件。如果仅按回车键或其他字符，文件不会被删除。与之相对应的就是 `-f` 选项，其作用是强制删除文件或目录，并不询问用户。使用该选项并配合 `-r` 选项，可实现递归强制删除，强制将指定目录下的所有文件与子目录一并删除，这可能导致灾难性后果。例如 `rm -rf /` 即可强制递归删除全盘文件，绝对不要轻易尝试!

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

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

```
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
```

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

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

```
mv
用法: mv [选项]... 源文件... 目录
```

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

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

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

以下为几个常用的快捷键可供参考。

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

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

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

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

0.4 实用工具介绍

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

0.4.1 Vim

Vim 被誉为编辑器之神，是程序员为程序员设计的编辑器，编辑效率高，十分适合编辑代码。对于习惯了图形化界面文本编辑软件的读者来说，刚接触 Vim 时一定会觉

得非常不习惯非常不顺手，所以下面通过创建一个 `helloworld.c`，来让读者熟悉一下 Vim 的基本操作：

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

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

(3) 输入内容：刚打开文件时，无法向其中直接输入代码，需要按“`i`”键进入插入模式（在底栏出现 `-- INSERT --` 表示已进入插入模式），之后便可以向其中输入 `helloworld` 程序，如下图所示：

图 0.2: 写入 `helloworld.c` 中的内容

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

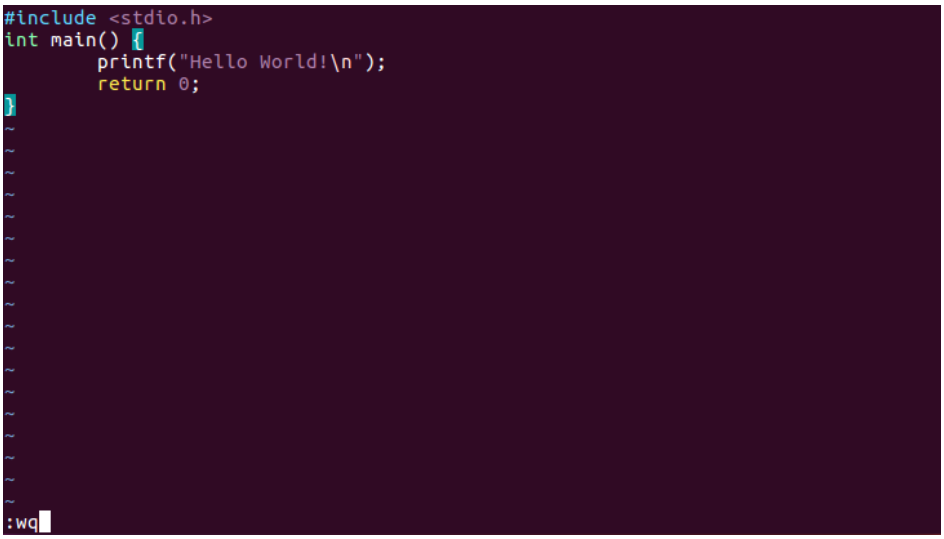


图 0.3: 保存并退出

接着简单了解一下对 Vim 进行一些自定义配置的方法:

```
# 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 cursorline
```

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

相关的内容可在网上查询，随用随查即可。

0.4.2 GCC

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

```
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                  显示一些警告信息
-c                  仅执行编译操作，不进行链接操作
-M                  列出依赖
-I<path>          编译时指定头文件目录，使用标准库时不需要指定目录，-I 参数可以用相对路径，比如头文件在当前目录，可以用-I. 来指定
参数：
```

```
C 源文件：指定 C 语言源代码文件
e.g.

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

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

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

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

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

图 0.4: GCC 编译可执行文件并运行

0.4.3 Makefile

如果想了解操作系统这样的大型软件，首先面临一个很大的问题：这些代码应当从那里开始阅读？答案是 Makefile。当你不知所措的时候，从 Makefile 开始往往会是一个不错的选择。那么 make 是什么，而 Makefile 又是什么呢？

make 是一个起源于 Unix 的构建自动化工具，一般用于维护软件开发的工程项目。它的核心思想是读取一个名为 Makefile 的文件，根据文件中定义的依赖关系和规则来决定需要重新编译或执行哪些命令，以达到某个目标（比如生成可执行文件）。严格来讲，make 本身是一种标准或概念，而不是具体的软件实现。但在一般语境中，make 指的是 GNU Make 是这个最流行、功能最丰富且最被广泛使用的具体实现。

相较于手动编译而言，使用 make 和 Makefile 具有更高的便捷性，可以方便地管理大型项目。而且理论上 Makefile 支持构建任意语言编写的源代码，只要其编译器可以通过 shell 命令来调用即可。如果一个项目有着复杂的构建流程，Makefile 便能充分展现其优势。

为了更为清晰地介绍 Makefile 的基本概念，下面通过编制一个简单的 Makefile 来说明。假设我们手头有一个 Hello World 程序需要编译。如果我们没有 Makefile，则需动手编译这个程序，执行以下命令：

```
# 直接使用 gcc 编译 Hello World 程序
$ gcc -o helloworld helloworld.c
```

那么，如何用 Makefile 表达呢？

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

该 Makefile 表达了什么含义呢？我们将一步一步进行介绍。

Makefile 最基本的格式如下:

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

其中，`target` 是构建 (build) 的目标[2]。而 `dependencies` 是构建该目标所需的其他目标，`dependencies` 可以为空。之后是构建该目标所需执行的命令。有一点尤为需要注意，每一个命令 (command) 之前必须用一个制表符（Tab）缩进。这里必须使用制表符而不能是空格，否则 make 会报错。

通过在 Makefile 中书写显式规则来告诉 make 工具文件间的依赖关系：如果想要构建 `target`，那么首先要准备好 `dependencies`，接着执行 `command` 中的命令 [3]。

在编写完恰当的规则之后即在 shell 中输入 `make target`(`target` 是目标名) [4] ，即可执行相应的命令、生成相应的目标。

前面提到，make 工具根据时间戳来判断是否需要编译，根据目标的不同类型，其具体逻辑如下：

- 若是文件目标，检查是否存在以目标名为文件名的文件
  - 若以目标名为文件名的文件不存在，则认为需要编译
  - 若以目标名为文件名的文件存在，则检查其依赖（若有）
    - 若其依赖都是文件目标，检查其依赖对应的文件的时间戳，若存在一个或多个依赖的时间戳比目标文件新，则认为需要编译
    - 若其依赖含有伪目标，则认为始终需要编译
- 若是伪目标，则认为始终需要编译

有时，我们并不希望目标与文件名绑定，例如，对于如下 Makefile：

```
some_file:
	touch some_file

clean:
	rm -f some_file
```

[2]从使用上来讲，可以认为目标分为文件目标和伪（Phony）目标两类，目标默认是文件目标；伪目标将在稍后介绍。

[3]注意即使 `target` 是文件目标，make 工具也并不会在执行 `command` 中的命令后自动创建名为 `target` 的文件，创建文件应当是 `command` 中的命令的职责。

[4]若在调用 make 时不指定目标名，则默认构建 Makefile 中出现的第一个目标。

其中设置 clean 目标的意图是，当我们执行 make clean 时，**总是**可以通过执行其中定义的 rm -f some_file 命令，来清理编译产物 some_file。

根据上面提到的判断是否需要编译的逻辑，若当前目录下有一个名为 clean 的文件，而 clean 目标没有任何依赖，则 make 就会认为该目标的构建已经完成了，且无需更新，导致清理编译产物的命令不被执行。

伪目标即是用来解决这一问题的，若将 clean 设置为伪目标，则 make 不会检查是否存在名为 clean 的文件，也不会根据其时间戳来判断是否需要重新构建，而是始终认为伪目标需要重新构建。

使用.PHONY 可将一个或多个目标定义为伪目标，其语法为.PHONY: targets，其中 targets 可以是一个或多个目标名，以空格分隔。

例如，对于上例，加入.PHONY 定义后，即可实现执行 make clean 时，**总是**可以清理编译产物 some_file。

```
.PHONY: clean

some_file:
	touch some_file

clean:
	rm -f some_file
```

在 Makefile 中还可以使用变量，变量赋值的基本语法为 < 变量名 > := < 变量值 > [5]，可以使用 $(< 变量名 >) 访问变量的值，例如：

```
files := file1 file2
some_file: $(files)
	echo "Look at this variable: " $(files)
	touch some_file

file1:
	touch file1
file2:
	touch file2

.PHONY: clean
clean:
	rm -f file1 file2 some_file
```

第一行定义了 files 变量，其值为 file1 file2；some_file 目标使用 files 变量的值作为依赖，表示其依赖 file1 和 file2 两个目标。

在 Makefile 中，所有变量都是字符串类型，并且给变量赋值时，单引号和双引号没有特殊含义，而是直接作为变量的值的一部分赋值给变量。

例如在下面的例子中，printf '$(a)' 和 printf $(b) 两行实际执行的命令都是 printf 'one two'。

[5]注意这只是基础介绍，上述使用:= 定义的变量是“简单展开变量”(simply expanded)，其它定义方式请参阅GNU make 文档。

```
a := one two
b := 'one two'

all:
        printf '$(a)'
        printf $(b)
```

下面，我们尝试编写一个简单的 Makefile 文件，完成上一节中编写的 `helloworld.c` 源文件的编译，以体会 make 工具的使用方法：

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：

```
CC := gcc

.PHONY: all clean

all: helloworld

helloworld: helloworld.c
        $(CC) -o helloworld helloworld.c

clean:
        rm -f helloworld
```

其中，`CC := gcc` 定义了变量 `CC`[6]，其值为 `gcc`；`.PHONY: all clean` 将 `all`[7] 和 `clean` 定义为伪目标。构建出 `helloworld` 可执行文件，即说明本“项目”构建完了，故 `all` 伪目标依赖 `helloworld` 文件目标。然后定义了编译 `helloworld.c` 的命令；`clean` 后的部分为删除可执行文件的命令。

2. 保存并回到命令行界面后，输入 `make clean`，执行 Makefile 文件中的 `rm helloworld` 命令，删除了之前编译出的可执行文件；接着，输入 `make all`，执行 Makefile 文件中的 `$(CC) helloworld.c -o helloworld` 命令，其中变量引用 `$(CC)` 被替换为变量值 `gcc`，编译出可执行文件。过程如下图：

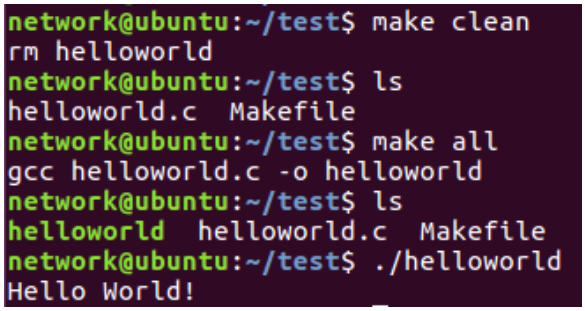


图 0.5: `make` 命令执行过程和效果

在 lab0 阶段，建议自己尝试编写更多的 Makefile 文件，尝试运用相关知识，掌握 make 工具的使用方法。

[6]根据惯例，变量名 `CC` 一般表示 C 编译器。

[7]`all` 目标一般表示构建完整项目，并在 Makefile 中放到第一个目标的位置，这样，当不手动指定目标，而是直接执行 `make` 时，将默认构建整个项目。注意 `all` 这个名称本身并不被 make 工具特殊处理，其表示“构建完整项目”只是一种惯例，而非规定。

0.4.4 ctags

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

```
set tags=tags;
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：

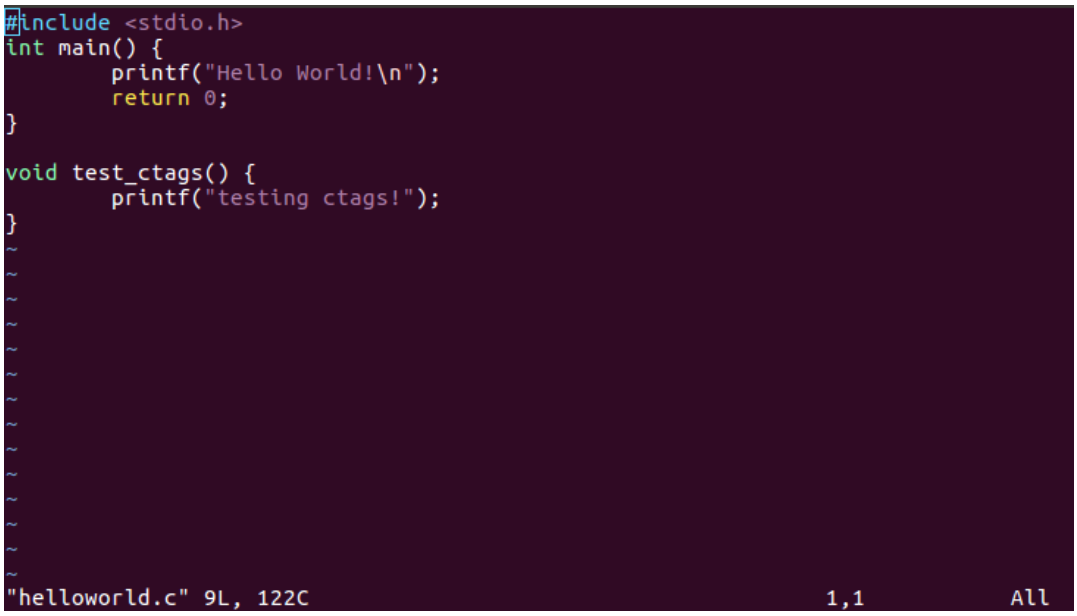


图 0.6: `helloworld.c` 文件中的内容

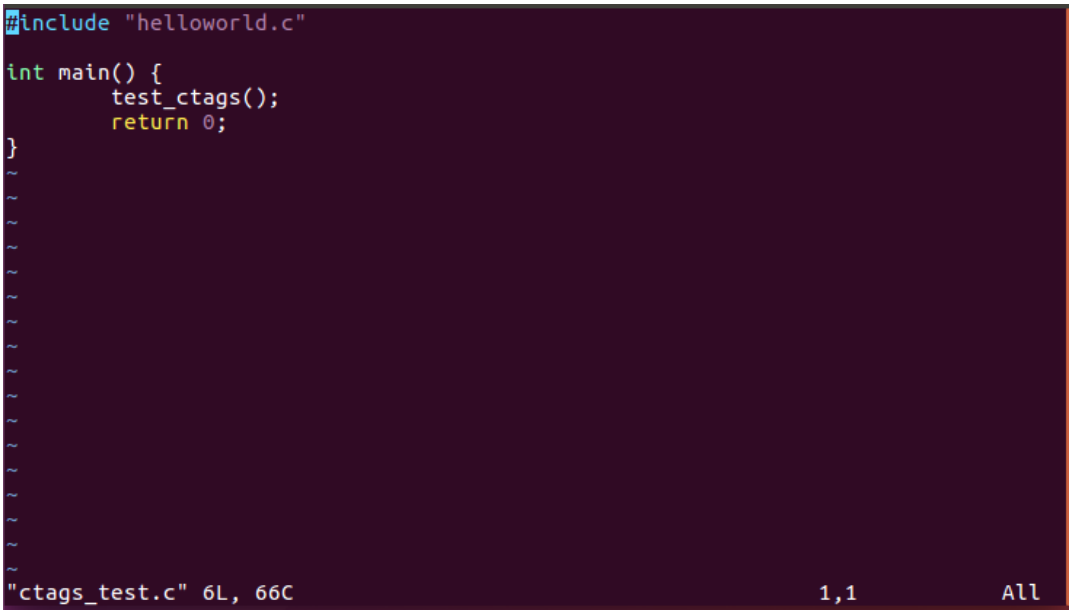


图 0.7: `ctags_test.c` 文件中的内容

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

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

```
#include "helloworld.c"

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

图 0.8: 光标处于函数名上

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

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

0.5 Git 专栏-轻松维护和提交代码

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

0.5.1 Git 是什么?

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

图 0.9: 手工版本控制

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

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

由此产生了需求，大家希望：

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

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

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

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

0.5.2 Git 基础指引

通过前几节的学习，完成如下操作：

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

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

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

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

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

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

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

```
$ git commit
```

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

```
# 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
#
Notes to test
```

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

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

以窗口提交方式是一种更简洁的方式，使用以下命令：

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

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

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

如果提交之后看到类似的提示，就说明提交成功了。

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

本次提交中可以得到以下提示信息中，后续会详细说明提交提示信息的含义。

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

在实验过程中，提交后可能会出现如下提示信息，说明这是要求设置提交者身份。

```
*** 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.
```

Note 0.5.4 可以用以下两条命令设置用户名和邮箱：

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

示例：

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

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

0.5.3 Git 文件状态

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

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

未修改 表示某文件在跟踪后一直没有改动过或者改动已经被提交。

已修改 表示修改了某个文件，但还没有加入（`add`）到暂存区中。

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

这里使用一张图来说明文件的四种状态的转换关系：

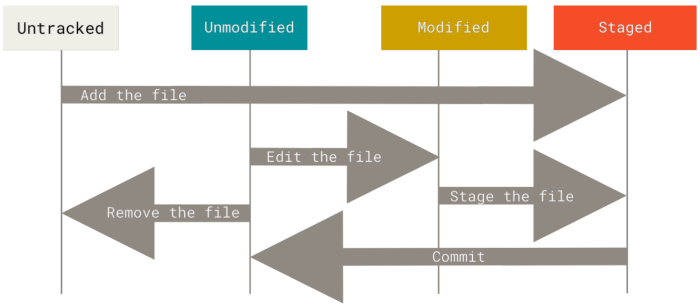


图 0.10: Git 中的四种状态转换关系

完成以下练习以进一步熟悉 Git 的使用方法。

Thinking 0.1 思考下列有关 Git 的问题：

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

■

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

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

```
# 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 Untracked.txt 中第 2 行是：Untracked files，而第二个文本文件 Stage.txt 中第二行内容是：Changes to be committed，而第三个文件 Modified.txt 中则是 Changes not staged for commit。

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

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

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

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

0.5.4 Git 三棵树

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

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

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

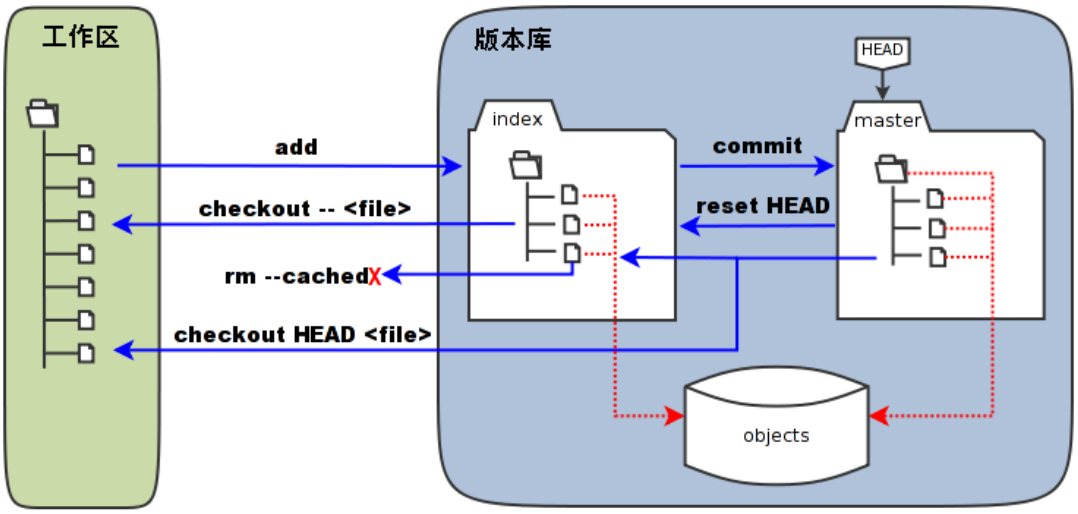


图 0.11: 工作区、暂存区和版本库

- 图中 objects 标识的区域为 Git 的对象库，实际位于 .git/objects 目录下。
- 图中左侧为工作区，右侧为版本库。在版本库中标记为”index” 的区域是暂存区（stage, index），标记为 master 的是 master 分支所代表的目录树。
- 图中我们可以看出此时 HEAD 实际是指向 master 分支的一个“指针”。所以图示的命令中出现 HEAD 的地方可以用 master 来替换。
- 当对工作区修改（或新增）的文件执行 git add 命令时，暂存区的目录树被更新，同时工作区修改（或新增）的文件内容被写入到对象库中的一个新的对象中，而该对象的 ID 被记录在暂存区的文件索引中。
- 当执行提交操作（git commit）时，会将暂存区的目录树写到版本库（对象库）中，master 分支会做相应的更新。即 HEAD 指向的目录树就是提交时暂存区的目录树。
- 当执行 git rm --cached <file> 命令时，会直接从暂存区删除文件，工作区则不做出改变。
- 当执行 git reset HEAD 命令时，暂存区的目录树会被重写，由 master 分支指向的目录树所替换，但是工作区不受影响。
- 当执行 git checkout -- <file> 命令时，会用暂存区指定的文件替换工作区的文件。这个操作很危险，会清除工作区中未添加到暂存区的改动。

- 当执行 `git checkout HEAD <file>` 命令时，会用 `HEAD` 指向的 `master` 分支中的指定文件来替换暂存区和以及工作区中的文件。这个命令也是极具危险性的，因为不但会清除工作区中未提交的改动，也会清除暂存区中未提交的改动。

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

0.5.5 Git 版本回退

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

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

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

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

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

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

Thinking 0.3 思考下列问题：

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

■

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

下面介绍 `git reset` 的更多用法。

```
git reset --hard
```

为了体会 reset 命令的作用，下面先做一个小练习：

Thinking 0.4 思考下列有关 Git 的问题：

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

■

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

使用这条命令可以进行版本回退或者切换到任何一个版本。它有两种用法：第一种是使用 HEAD 类似形式，如果想退回上个版本就用 HEAD^，上上个版本的话就用 HEAD^^，要是回退到前 50 个版本则可使用 HEAD~50 来代替；第二种就是使用 hash 值，使用 hash 值可以在不同版本之间任意切换，足见 hash 值的强大。

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

0.5.6 Git 分支

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

```
# 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>
```

这条命令将会在实验课考试进行的时候用到。其功能相当于把当前分支的内容拷贝一份到新的分支里去，然后我们在新的分支上做测试功能的添加即可，不会影响实验分支的效果等。假如我们当前在 master[8]分支下已经有过三次提交记录，这时我们使用 `git branch` 命令新建了一个分支为 testing（参考图 0.12）。

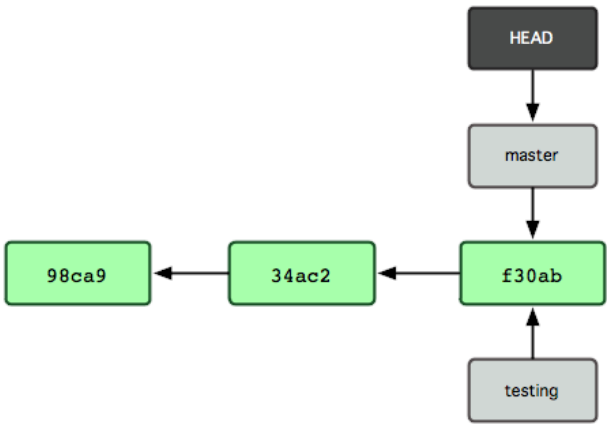


图 0.12: 分支建立后

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

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

想查看所有的远程分支和本地分支，只需要加上 `-a` 选项即可

```
# 查看所有的远程与本地的所有分支
$ 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。

[8]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

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

```
# 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0.13)。

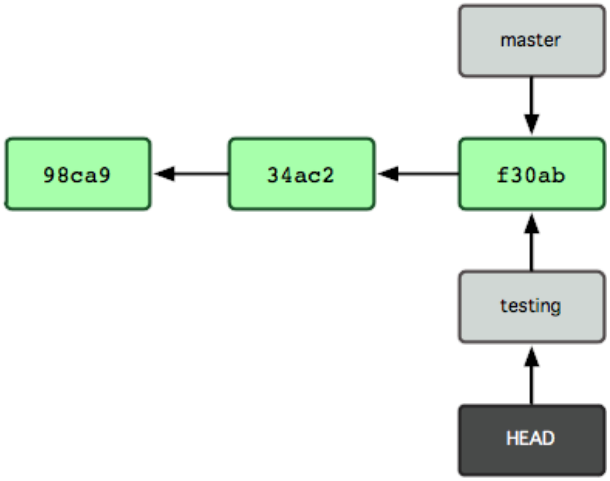


图 0.13: 分支切换后

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

我们之前所介绍的这些命令只是在本地进行操作的，其中必须掌握

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

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

0.5.7 Git 远程仓库与本地

下面介绍两条跟远程仓库有关的命令，其作用很简单，但要用好却是比较难。

```
# 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]
```

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

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

如果你还想进一步学习 Git 的知识，可以查看教程[9] 和游玩 GitHug

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                从目录递归查找
-n                显示行号
```

tree 命令可以根据文件目录生成文件树，作用类似于 ls。

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

```
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
```

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 ugo 文件
```

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

下面是一些 chmod 的使用示例：

```
# 为 run.sh 的拥有者添加执行权限
chmod u+x run.sh

# 修改 run.sh 的权限为 rwxr-xr-x（对应二进制 111b = 7, 101b = 5）
chmod 755 run.sh

# 移除所有人对 run.sh 的写权限
chmod -w run.sh
```

diff 命令用于比较文件的差异。

```
diff [选项] file1 file2
选项（常用）：
-b                不检查空白字符的不同
-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            安静模式，只显示经过 sed 处理的内容。否则显示输入文本的所有内容。
-i            直接修改读取的档案内容，而不是输出到屏幕。否则，只输出不编辑。
命令（常用）：
[行号] a\ [内容] 新增，在行号后新增一行相应内容。行号可以是“数字”，在这一行之后新增，
              也可以是“起始行，终止行”，在其中的每一行后新增。
              当不写行号时，在每一行之后新增。使用 $ 表示最后一行。后面的命令同理。
[行号] c\ [内容] 取代。用内容取代相应行的文本。
[行号] i\ [内容] 插入。在当前行的上面插入一行文本。
```

```
[行号] d        删除当前行的内容。
[行号] p        输出选择的内容。通常与选项-n 一起使用。
s/re/string/    将 re（正则表达式）匹配的内容替换为 string。
```

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

下面是一些 sed 的使用示例：

```
# 输出 my.txt 的第三行
sed -n '3p' my.txt

# 删除 my.txt 文件的第二行
sed '2d' my.txt

# 删除 my.txt 文件的第二行到最后一行（$ 符号表示到最末尾）
sed '2,$d' 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 's/str1/str2/g' 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。
sed -e '4a\str ' -e 's/str/aaa/' my.txt
```

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

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

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

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

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

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

1. 窗格操作

   tmux 的窗格（pane）可以做出分屏的效果。

   - Ctrl+B % 垂直分屏 (组合键之后按一个百分号)，用一条垂线把当前窗口分成左右两屏。
   - Ctrl+B ” 水平分屏 (组合键之后按一个双引号)，用一条水平线把当前窗口分成上下两屏。

- Ctrl+B O 依次切换当前窗口下的各个窗格。
- Ctrl+B Up|Down|Left|Right 根据按箭方向选择切换到某个窗格。
- Ctrl+B Space (空格键) 对当前窗口下的所有窗格重新排列布局，每按一次，换一种样式。
- Ctrl+B Z 最大化当前窗格。再按一次后恢复。
- Ctrl+B X 关闭当前使用中的窗格，操作之后会给出是否关闭的提示，按 `y` 确认即关闭。
- Ctrl+B [进入当前窗格的翻页模式。进入翻页模式后，可使用 `PageUp/PageDown` 键进行向上/向下翻页操作，按 `Esc` 键即可退出翻页模式。

2. 窗口操作

   每个窗口（window）可以分割成多个窗格（pane）。

   - Ctrl+B C 创建之后会多出一个窗口
   - Ctrl+B P 切换到上一个窗口。
   - Ctrl+B N 切换到下一个窗口。
   - Ctrl+B 0 切换到 `0` 号窗口，依此类推，可换成任意窗口序号
   - Ctrl+B W 列出当前 session 所有串口，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。

3. 会话操作

   一个会话（session）可以包含多个窗口（window）

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

0.6.2 shell 脚本

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

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

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

我们可以使用 bash 来运行这个脚本:

```
bash my.sh
```

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

```
./my.sh
```

来运行。

在修改权限之前, 我们自己创建的 shell 脚本一般是不能直接运行的, 需要用 chmod +x my.sh 添加运行权限: 在脚本中我们通常会加入 #!/bin/bash 到文件首行, 以保证直接执行我们的脚本时使用 bash 作为解释器。[10] 第二行的内容是注释, 以 # 开头。第三行是命令。

参数与函数

我们可以向 shell 脚本传递参数。文件 my2.sh 的内容如下:

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

执行命令

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

则 shell 会执行 echo msg 这条命令。

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

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

shell 中的函数也用类似的方式传递参数。

[10] 这一行被称为 Shebang。Shebang 是出现在脚本文件第一行, 由 #! 开头的字符序列, 其语法为 #!interpreter [optional-one-arg-only], 其中 interpreter 指定了执行脚本文件的解释器, optional-one-arg-only 是可选参数, 可用于向解释器传递一个参数。在类 Unix 操作系统中, 当包含 Shebang 的文本文件被当作可执行文件执行时, 程序加载器将执行 interpreter 指定的解释器, 并将脚本文件名作为第一个参数传递给解释器。例如, #!/usr/bin/env python3 将使用 Python 解释脚本内容, 因而可在文本文件中编写 Python 代码, 并像一般的可执行文件那样使用./< 脚本文件名 > 的方式执行。使用/usr/bin/env 作为解释器, 并将 python3 作为参数传递, 可查找并执行 PATH 环境变量中的 python3 可执行文件, 而不是硬编码 python3 的绝对路径, 以配合虚拟环境使用。

```
function < 函数名 > () {
        <commands>
}
```

function 或者 () 可以省略其中一个。例如：

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

# 调用函数 fun
fun 1 str2
```

流程控制语句

shell 脚本中也可以使用分支和循环语句：

if 的格式：

```
if <condition>
then
        <command1>
        <command2>
        # ...
fi
```

或者写到一行

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

例如（注意等号前后不能有空格，左中括号后和右中括号前一定需要空格）：

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

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

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

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

while 语句格式如下

```
while <condition>
do
        <commands>
done
```

while 语句体中可以使用 continue 和 break 这两个循环控制语句。
例如创建 9 个目录，名字是 file1 到 file9。

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

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

0.6.3 重定向和管道

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

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

重定向和管道可以重定向以上的流。> 可以重定向命令的标准输出到文件。例如：ls / > filename 可以将根目录下的文件名输出到当前目录下的 filename 中。与之类似的，还有重定向追加输出 >>，将 >> 前命令的输出追加输出到 >> 后指定的文件中去；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。例如：

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

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

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

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

Thinking 0.5 执行如下命令, 并查看结果

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

■

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

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

图 0.14: 文件内容

0.7 实战测试

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

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

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

回文数则输出 Y，否则输出 N。[注意：正读倒读相同的整数叫回文数]

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

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

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

要求按照要求完成后，最终提交的文件树图示如下

```
|-- dst
|   |-- Makefile
|   |-- palindrome.c
|   `-- sh_test
|       `-- hello_os.sh
`-- src
    |-- Makefile
    |-- palindrome.c
    `-- sh_test
        |-- file
        `-- hello_os.sh
```

第一题最终提交的文件树

■

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

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

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

```
`-- sh_test1
    |-- file1
    |-- file10
    |-- file11
    |-- file12
    |-- file2
    |-- file3
    |-- file4
    |-- file5
    |-- file6
    |-- file7
    |-- file8
    |-- file9
    |-- newfile41
    |-- newfile42
    |-- newfile43
    |-- newfile44
    |-- newfile45
    |-- newfile46
    |-- newfile47
    |-- newfile48
    |-- newfile49
    |-- newfile50
    |-- newfile51
    |-- newfile52
    |-- newfile53
    |-- newfile54
    |-- newfile55
    `-- changefile.sh
```

第二题最终提交的文件树

■

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

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

要求按照要求完成后，`result` 内显示样式如下 (一个答案占一行)：

```
39
123
134
147
344
395
446
471
735
908
1207
1422
1574
1801
1822
1924
1940
1984
```

第三题完成后结果

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

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

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

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

```
|-- code
|   |-- Makefile
|   |-- fibo.c
|   |-- fibo.o
|   |-- main.c
|   |-- main.o
|   `-- modify.sh
|-- fibo
|-- include
|   `-- fibo.h
`-- Makefile
```

第四题 `make` 后文件树

```
|-- code
|   |-- Makefile
|   |-- fibo.c
|   |-- main.c
|   `-- modify.sh
|-- fibo
|-- include
|   `-- fibo.h
`-- Makefile
```

第四题 `make clean` 后文件树

0.8 实验思考

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

附录 A 补充知识

在本章中，我们将介绍一些操作系统实验的补充知识，有助于我们理解并实现操作系统。

A.1 shell 编程易错点说明

语句与命令的区分

对于有良好的代码格式规范的同学，一般会习惯于在赋值语句的等号两边加上空格，而在 shell 编程中我们赋值语句的等号两边都不能存在空格，这可能会让部分同学感到困惑。这种看似与大多编程语言的规范都不同的使用方式其实是从 shell 自身的功能出发的所要求的。在 shell 出现之时，目的就是让用户能够通过命令与我们的操作系统进行交互，而 shell 中一条命令的基本格式为 `<command> <arg0> <arg1> ...`。因此如果我们写出 `a = 1` 这样的“赋值语句”，它将会被 shell 解析为调用一个名为 `a` 的命令，并给它传入参数 `=` 和 `1`，从而会得到报错 `command not found: a`（当我们的环境中不存在 `a` 这个可执行程序时）。因此在 shell 中一条赋值语句只能写作 `a=1` 这样的等号两边都不能有空格的形式。

命令中括号的用法

在 shell 中不同的括号用法可能会对初学者造成不小的困扰，因此下面对括号的基本用法进行简单的介绍。

- 美元符与单小括号、反引号

  使用美元符加上单小括号用于隔离并执行命令，并将其输出替换至原处。例如下述 shell 程序中，使用 `$(diff file1 file2)` 得到 `diff` 比较的输出并替换至原处，从而变为 `"<diff output>"` 也即得到 `diff` 的输出结果并作为一个字符串，判断该字符串是否为空即可判断 `file1` 与 `file2` 是否相同。在该命令中的 `$(diff file1 file2)` 改为使用反引号 `` `diff file1 file2` `` 也有着相同的效果。

```
#!/bin/bash

if [ -z "$(diff file1 file2)" ]; then
        echo "same"
else
        echo "different"
```

```
fi
```

- 双小括号

  双小括号用于算术表达式（当然你也可以在其中进行赋值），例如：

```
#!/bin/bash

a=1
b=2
c=$((a + b))
echo $c         # 3

((a = 10))
echo $a         # 10

((b++))
echo $b         # 3

a=$((a + 2))
echo $a         # 12

echo $((a * b + c))         # 39
```

- 单中括号

  单中括号与 `test` 相同，用于判断条件是否成立。大家通过 `ls /bin` 能发现其中存在一个名为 `[` 的可执行程序，事实上这个 `[` 就是我们写在判断条件中的 `[`。这也就是为什么在判断条件 `[ <condition> ]` 时一定需要有空格，例如条件 `[ $a -ne 2 ]`，实际上是将 `a`, `-ne`, `2`, `]` 作为参数传给可执行程序 `[`，而不是表面上看起来像是用中括号包裹了判断语句 `$a -ne 2`。

- 双中括号

  双中括号同样用于判断条件，但相比单中括号提供了更为丰富的一些功能。例如在双中括号中可以直接使用逻辑运算：`[[ $a -gt 1 && $a -lt 100 ]]`，而使用单中括号只能写为 `[ $a -gt 1 ] && [ $a -lt 100 ]`

A.2 真实操作系统的内核及启动详解

操作系统最重要的部分是操作系统内核，因为内核需要直接与硬件交互来管理各个硬件，从而利用硬件的功能为用户进程提供服务。为了启动操作系统，就需要将内核程序在计算机上运行起来。一个程序要能够运行，其必须能够被 CPU 直接访问，所以不能放在磁盘上，因为 CPU 无法直接访问磁盘；另一方面，内存 RAM 是易失性存储器，掉电后将丢失全部数据，所以不可能将内核代码保存在内存中。所以直观上可以认识到：磁盘不能直接访问，并且内存掉电易失，因此，内核有可能放置的位置只能是 CPU 能够直接访问的非易失性存储器——ROM 或 FLASH 中。

但是，直接把操作系统内核放置在这样的非易失存储器上会有一些问题：

1. 这种 CPU 能直接访问的非易失性存储器的存储空间一般会映射到 CPU 可寻址空间的某个区域，这个是在硬件设计决定的。显然这个区域的大小是有限的，如果功能比较简单的操作系统还能够放在其中，对于较大的普通操作系统显然不够。
2. 如果操作系统内核在 CPU 加电后直接启动，意味着一个计算机上只能启动一个操作系统，这样的限制显然不是我们所希望的。

3. 把特定硬件相关的代码全部放在操作系统中也不利于操作系统的移植工作。

基于上述考虑，设计人员一般都会将硬件初始化的相关工作作为“bootloader”程序放在非易失存储器中，而将操作系统内核放在磁盘中。这样的做法可有效解决上述的问题：

1. 将硬件初始化的相关工作从操作系统中抽出放在 bootloader 中实现，意味着通过这种方式实现了硬件启动和软件启动的分离。因此需要存储的硬件启动相关指令不需要很多，能够很容易地保存在容量较小的 ROM 或 FLASH 中。

2. bootloader 在硬件初始化完后，需要为软件启动（即操作系统内核的功能）做相应的准备，比如需要将内核镜像从存放它的存储器（比如磁盘）中读到 RAM 中。既然 bootloader 需要将内核镜像加载到内存中，那么它就能选择使用哪一个内核镜像进行加载，即实现多重开机的功能。使用 bootloader 后，我们就能够在一个硬件上选择运行不同的操作系统了。

3. bootloader 主要负责硬件启动相关工作，同时操作系统内核则能够专注于软件启动以及对用户提供服务的工作，从而降低了硬件相关代码和软件相关代码的耦合度，有助于操作系统的移植。使用 bootloader 更清晰地划分了硬件启动和软件启动的边界，使操作系统与硬件交互的抽象层次提高了，从而简化了操作系统的开发和移植工作。

A.2.1 Bootloader

从操作系统的角度看，bootloader 的目标就是正确地找到内核并加载执行。另外，由于 bootloader 的实现依赖于 CPU 的体系结构，因此大多数 bootloader 都分为 stage1 和 stage2 两个部分。

在 stage1 时，此时需要初始化硬件设备，包括 watchdog timer、中断、时钟、内存等。需要注意的一个细节是，此时内存 RAM 尚未初始化完成，因而 stage1 运行的 bootloader 程序直接从非易失存储器上（比如 ROM 或 FLASH）加载。由于当前阶段不能在内存 RAM 中运行，其自身运行会受诸多限制，比如某些非易失存储器（ROM）不可写，即使程序可写的 FLASH 也有存储空间限制。这就是为什么需要 stage2 的原因。所以，stage1 除了初始化基本的硬件设备以外，会为加载 stage2 准备 RAM 空间，然后将 stage2 的代码复制到 RAM 空间，并且设置堆栈，最后跳转到 stage2 的入口函数。

stage2 运行在 RAM 中，此时有足够的运行环境从而可以用 C 语言来实现较为复杂的功能。这一阶段的工作包括，初始化这一阶段需要使用的硬件设备以及其他功能，然后将内核镜像从存储器读到 RAM 中，并为内核设置启动参数，最后将 CPU 指令寄存器的内容设置为内核入口函数的地址，即可将控制权从 bootloader 转交给操作系统内核。

从 CPU 上电到操作系统内核被加载的整个启动的步骤如图A.1所示。

需要注意的是，以上 bootloader 的两个工作阶段只是从功能上论述内核加载的过程，在具体实现上不同的系统可能有所差别，而且对于不同的硬件环境也会有些不同。在我们常见的 x86 PC 的启动过程中，首先执行的是 BIOS 中的代码，主要完成硬件初始化相关的工作，然后 BIOS 会从 MBR（master boot record，开机硬盘的第一个扇区）中读取开机信息。在 Linux 中常说的 GRUB 和 LILO 这两种开机管理程序就是保存在 MBR 中。

> **Note A.2.1** GRUB(GRand Unified Bootloader) 是 GNU 项目的一个多操作系统启动程序。简单的说，就是可以用于有多个操作系统的机器上，在刚开机的时候选择一个操作系统进行引导。如果安装过 Ubuntu 一类的发行版的话，一开机出现的那个选择系统用的菜单就是 GRUB 提供的。

(这里以 GRUB 为例)BIOS 加载 MBR 中的 GRUB 代码后就把 CPU 交给了 GRUB，GRUB 的工作就是一步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

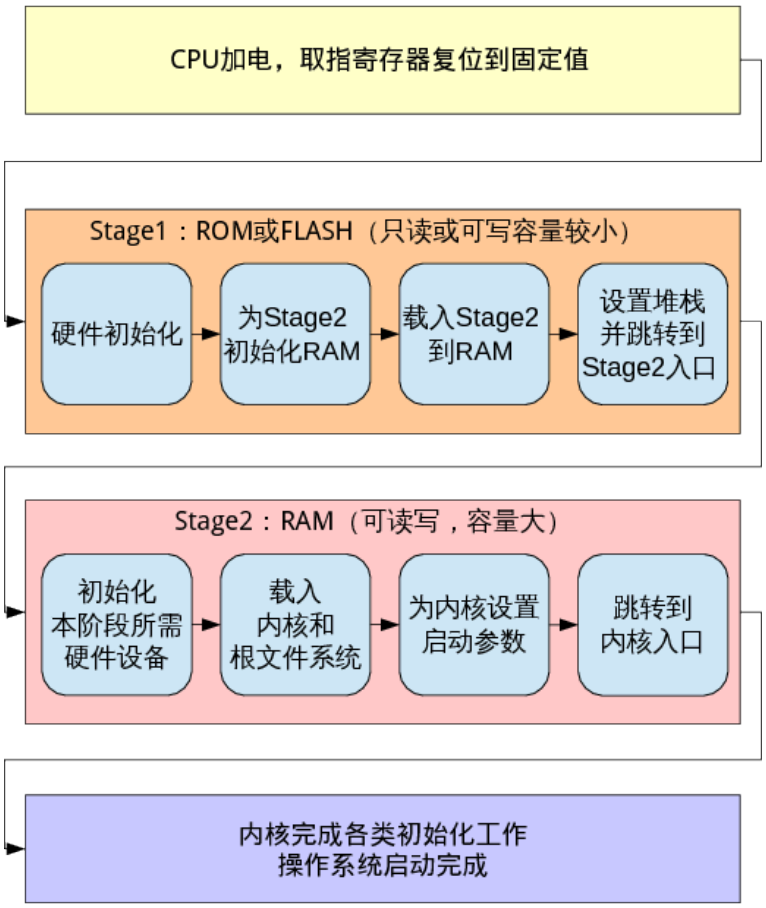

CPU加电，取指寄存器复位到固定值
Stage1：ROM或FLASH（只读或可写容量较小）
硬件初始化
为Stage2
初始化RAM
载入Stage2
到RAM
设置堆栈
并跳转到
Stage2入口
Stage2：RAM（可读写，容量大）
初始化
本阶段所需
硬件设备
载入
内核和
根文件系统
为内核设置
启动参数
跳转到
内核入口
内核完成各类初始化工作
操作系统启动完成

图 A.1: 启动的基本步骤

文件加载到内存中，并将 CPU 控制权转交给操作系统内核。这样看来，其实 BIOS 和 GRUB 的前一部分构成了前述 stage1 的工作，而 stage2 的工作则是完全在 GRUB 中完成的。

> **Note A.2.2** bootloader 有两种操作模式：启动加载模式和下载模式。区别是前者是通过本地设备中的内核镜像文件启动操作系统的，而后者是通过串口或以太网等通信手段将远端的内核镜像下载到内存。

A.3 编译与链接详解

一个简单的 C 程序

```c
#include <stdio.h>

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

我们以代码A.3为例，讲述我们这个冗长的故事。我们首先探究这样一个问题：**含有多个 C 文件的工程是如何编译成一个可执行文件的？**

这段代码相信你非常熟悉了，不知你有没有注意到过这样一个小细节：puts 函数的定义在哪里？[1]

我们都学过，C 语言中函数必须有声明（函数原型）才能被调用，并且若要能生成可执行文件，编译器必须能找到所有被调用了的函数的定义（具体实现）。那么 puts 的声明和具体实现在哪里呢？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。

stdio.h 中关于 puts 的内容

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

/* Write a string, followed by a newline, to stdout.

   This function is a possible cancellation point and therefore not
   marked with __THROW.  */
extern int puts (const char *__s);
```

在代码A.3中，我们展示了从当前系统的 stdio.h 中摘录出的与 puts 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 puts 的定义。

或者说，并没有 puts 的具体实现。可没有具体的实现，我们究竟是如何调用 puts 的呢？没有实现代码，编译器怎么知道应该如何生成二进制代码呢？

[1] puts 是 C 标准定义的标准库函数，而不是我们自己的代码定义实现的。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 puts 如何和我们的 C 文件编译到一起，来展示整个过程。

我们来一步一步探究，puts 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 puts (const char *__s);

int main()
{
    puts("Hello World!");
    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，这一阶段处理后，仍然只有 puts 函数的声明，而没有定义。

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 -g -c 源代码文件名
```

其中-g 选项表示在生成的二进制文件中附加调试信息，便于分析汇编代码时与源代码相对应。

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：

```
objdump --section=.text --disassemble=main --source \
要反汇编的目标文件名 > 导出文本文件名
```

其中--section=.text 表示仅处理.text 节的内容；--disassemble=main 表示仅反汇编 main 符号的代码；--source 表示显示汇编代码与源代码的对应关系。

[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。

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

Disassembly of section .text:

0000000000000000 <main>:
#include <stdio.h>

int main()
{
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
    puts("Hello World!");
   4:   48 8d 05 00 00 00 00    lea    0x0(%rip),%rax        # b <main+0xb>
   b:   48 89 c7                mov    %rax,%rdi
   e:   e8 00 00 00 00          call   13 <main+0x13>
    return 0;
  13:   b8 00 00 00 00          mov    $0x0,%eax
}
  18:   5d                      pop    %rbp
  19:   c3                      ret
```

我们只需要注意中间那句 `call` 即可，这一句是调用函数的指令。对照左侧的机器码，其中 `e8` 是 `call` 指令的操作码。根据 x86 指令的特点，`e8` 后面应该跟的是 `puts` 的地址。可在这里我们却发现，**本该填写 `puts` 地址的位置上被填写了一串 0**。那个地址显然不可能是 `puts` 的地址。也就是说，直到这一步，`puts` 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`；`-static` 表示进行静态链接，若不指定，则现代的系统上可能默认使用动态链接。使用静态链接时，生成的二进制文件中即含有所有需要使用的函数的代码，便于我们观察分析；而使用动态链接时，部分函数可能在运行时才由动态链接器进行链接，不利于我们分析。

```
gcc -g -static [-o 输出可执行文件名] 源代码文件名
objdump --disassemble --source 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:

0000000000401000 <_init>:
  401000:  f3 0f 1e fa             endbr64
  401004:  48 83 ec 08             sub    $0x8,%rsp
  401008:  48 c7 c0 00 00 00 00    mov    $0x0,%rax
  40100f:  48 85 c0                test   %rax,%rax
  401012:  74 02                   je     401016 <_init+0x16>
  401014:  ff d0                   call   *%rax
  401016:  48 83 c4 08             add    $0x8,%rsp
  40101a:  c3                      ret

Disassembly of section .plt:

0000000000401020 <.plt>:
  401020:  ff 25 da 6f 0a 00       jmp    *0xa6fda(%rip)
  401026:  66 90                   xchg   %ax,%ax
  401028:  ff 25 da 6f 0a 00       jmp    *0xa6fda(%rip)
```

```
...

Disassembly of section .text:

...

0000000000402dc0 <_start>:
  402dc0:  f3 0f 1e fa             endbr64
  402dc4:  31 ed                   xor    %ebp,%ebp
  402dc6:  49 89 d1                mov    %rdx,%r9
  402dc9:  5e                      pop    %rsi
  402dca:  48 89 e2                mov    %rsp,%rdx
  402dcd:  48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
  402dd1:  50                      push   %rax
  402dd2:  54                      push   %rsp
  402dd3:  45 31 c0                xor    %r8d,%r8d
  402dd6:  31 c9                   xor    %ecx,%ecx
  402dd8:  48 c7 c7 e5 2e 40 00    mov    $0x402ee5,%rdi
  402ddf:  67 e8 3b 26 00 00       addr32 call 405420 <__libc_start_main>
  402de5:  f4                      hlt
  402de6:  66 2e 0f 1f 84 00 00    cs nopw 0x0(%rax,%rax,1)
  402ded:  00 00 00

...

0000000000402ee5 <main>:
#include <stdio.h>

int main()
{
  402ee5:  55                      push   %rbp
  402ee6:  48 89 e5                mov    %rsp,%rbp
    puts("Hello World!");
  402ee9:  48 8d 05 20 a1 07 00    lea    0x7a120(%rip),%rax
  402ef0:  48 89 c7                mov    %rax,%rdi
  402ef3:  e8 18 31 00 00          call   406010 <_IO_puts>
    return 0;
  402ef8:  b8 00 00 00 00          mov    $0x0,%eax
}
  402efd:  5d                      pop    %rbp
  402efe:  c3                      ret
  402eff:  90                      nop

...

0000000000406010 <_IO_puts>:
  406010:  f3 0f 1e fa             endbr64
  406014:  55                      push   %rbp
  406015:  48 89 e5                mov    %rsp,%rbp
  406018:  41 55                   push   %r13
  40601a:  49 89 fd                mov    %rdi,%r13
  40601d:  41 54                   push   %r12
  40601f:  53                      push   %rbx
  406020:  48 83 ec 18             sub    $0x18,%rsp
  406024:  e8 77 b0 ff ff          call   4010a0 <_init+0xa0>

...
```

篇幅所限，只保留了关键部分的代码。

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，我们可以惊喜地看到，主函数里那一句 call 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 _IO_puts 的那个位置上。其中包含我们所调用的 puts 的具体实现。

由此，我们不难推断，puts 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，而不是直接以源代码形式和我们编写的其它源代码一起编译。那么，这个细节究竟有什么用呢？

作为一个库函数，puts 被大量的程序所使用。因此，每次都将其从源代码编译一遍实在太浪费时间了。puts 的实现其实早就被编译成了二进制形式。

但此时，puts 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。

对于含有多个.c 文件的工程来说，编译器会首先将所有的.c 文件以文件为单位，编译成.o 文件。最后再将所有的.o 文件以及函数库链接在一起，形成最终的可执行文件。整个过程如图A.2所示。

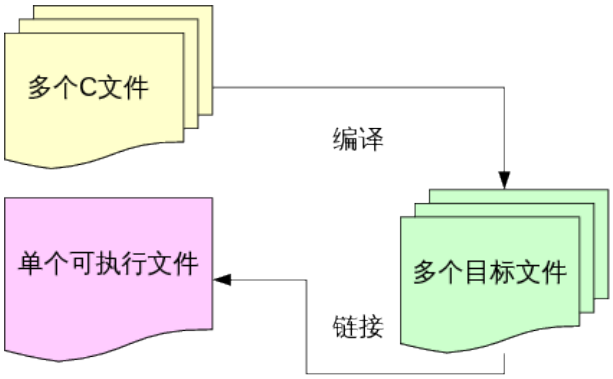


图 A.2: 编译、链接的过程

A.4 printk 格式具体说明

下面是我们需要完成的 printk 函数的具体说明，同学们可以参考 cppreference 中有关 C 语言 printf 函数的文档[3]或者 C++ 文档[4]，对 printf 函数进行更加详细的了解。

函数原型：

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

参数 fmt 除了可以包含直接输出的字符，还可以包含格式符（format specifiers），类似 printf 中的格式字符串，但简化并新增了一些功能。格式符的原型为：

```
%[flags][width][length]<specifier>
```

其中 specifier 指定了输出变量的类型，参见下表A.1：

[3] https://zh.cppreference.com/w/c/io/fprintf

[4] https://www.cplusplus.com/reference/cstdio/printf

表 A.1: Specifiers 说明

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

除了 specifier 之外，格式符也可以包含一些其它可选的副格式符 (sub-specifier)，有 flag(表A.2)：

表 A.2: flag 说明

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

和 width(表A.3)：

表 A.3: width 说明

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

另外，还可以使用 length 来修改数据类型的长度，在 C 中我们可以使用 `l`、`ll`、`h` 等，但这里我们只使用 `l`，参看下表A.4

表 A.4: length 说明

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

A.5 MIPS 汇编与 C 语言

在操作系统编程中，不可避免地要接触到汇编语言。我们经常需要从 C 语言中调用一些汇编语言写成的函数，或者反过来，在汇编中跳转到 C 函数。为了帮助同学更好的了解 c 语言于汇编之间的联系，我们在附录的这一节中补充了 MIPS 汇编的相关知识，介绍了常见的 C 语言语法与汇编的对应关系。我们给出样例代码A.5，介绍典型的 C 语言中的语句对应的汇编代码。

样例程序

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

int main()
{
    int i;
    int sum = 0;
    for (i = 0; i < 10; ++i) {
        sum += fib(i);
    }

    return 0;
}
```

A.5.1 循环与判断

这里你可能会问了，样例代码里只有循环啊！哪里有什么判断语句呀？事实上，由于 MIPS 汇编中没有循环这样的高级结构，所有的循环均是采用判断加跳转语句实现的，所以我们将循环语句和判断语句合并在一起进行分析。我们分析代码的第一步，就是要将循环等高级结构，用**判断加跳转**的方式替代。例如，代码A.5第 13-15 行的循环语句，其最终的实现可能就如下面的 C 代码所展示的那样。

```c
      i = 0;
      goto CHECK;
FOR:  sum += fib(i);
      ++i;
CHECK:if (i < 10) goto FOR;
```

将样例程序编译[5]，我们观察其反汇编代码。对照汇编代码和我们刚才所分析出来的 C 代码。我们基本就能够看出来其间的对应关系。这里，我们将对应的 C 代码标记在反汇编代码右侧。

```
  400158:       sw      zero,16(s8)           #       sum = 0;
  40015c:       sw      zero,20(s8)           #       i = 0;
  400160:       j       400190 <main+0x48>    #       goto CHECK;
  400164:       nop                           # --------------------
  400168:       lw      a0,20(s8)             #  FOR:
  40016c:       jal     4000b0 <fib>          #
  400170:       nop                           #
```

[5]为了生成更简单的汇编代码，我们采用了`-nostdlib -static -mno-abicalls` 这三个编译参数

```
  400174:       move    v1,v0                   #       sum += fib(i);
  400178:       lw      v0,16(s8)               #
  40017c:       addu    v0,v0,v1                #
  400180:       sw      v0,16(s8)               #
  400184:       lw      v0,20(s8)               # --------------------
  400188:       addiu   v0,v0,1                 #       ++i;
  40018c:       sw      v0,20(s8)               # --------------------
  400190:       lw      v0,20(s8)               # CHECK:
  400194:       slti    v0,v0,10                #       if (i < 10)
  400198:       bnez    v0,400168 <main+0x20> #           goto FOR;
  40019c:       nop
```

再将右边的 C 代码对应回原来的 C 代码，我们就能够大致知道每一条汇编语句所对应的原始的 C 代码是什么了。可以看出，判断和循环主要采用 `slt`、`slti` 判断两数间的大小关系，再结合 b 类型指令根据对应条件跳转。以这些指令为突破口，我们就能大致识别出循环结构、分支结构了。

A.5.2 函数调用

Note A.5.1 注意区分函数的调用方和被调用方来分析函数调用。

这里选用样例程序中的 `fib` 这个函数来观察函数调用相关的内容。这个函数是一个递归函数，因此，它函数调用过程的调用者，同时也是被调用者。我们可以从中观察到如何调用一个函数，以及一个被调用的函数应当做些什么工作。

我们先将整个函数调用过程用高级语言来表示一下。

```
int fib(int n)
{
        if (n == 0) goto BRANCH;
        if (n != 1) goto BRANCH2;
BRANCH: v0 = 1;
        goto RETURN;
BRANCH2:v0 = fib(n-1) + fib(n-2);
RETURN: return v0;
}
```

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？

```
004000b0 <fib>:
  4000b0:       27bdffd8        addiu   sp,sp,-40
  4000b4:       afbf0020        sw      ra,32(sp)
  4000b8:       afbe001c        sw      s8,28(sp)
  4000bc:       afb00018        sw      s0,24(sp)
  # 中间暂且掠过，只关注一系列 sw 和 lw 操作。
  400130:       8fbf0020        lw      ra,32(sp)
  400134:       8fbe001c        lw      s8,28(sp)
  400138:       8fb00018        lw      s0,24(sp)
  40013c:       27bd0028        addiu   sp,sp,40
  400140:       03e00008        jr      ra
  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。[6]

[6]这里“压入栈的状态”通常称为“栈帧”，栈帧中保存了该函数的返回地址和局部变量。

好了，回忆起了这个细节，我们再来看看汇编代码。在函数的开头，编译器为我们添加了一组 `sw` 操作，将所有当前函数所需要用到的寄存器原有的值全部保存到了内存中[7]。而在函数返回之前，编译器又加入了一组 `lw` 操作，将值被改变的寄存器全部恢复为原有的值。

我们惊奇地发现：编译器在函数调用的前后为我们添加了一组压栈 (push) 和弹栈 (pop) 的操作，为我们保存了函数的当前状态。函数的开始，编译器首先**减小 `sp` 指针的值，为栈分配空间**。并将需要保存的值放置在栈中。当函数将要返回时，编译器再**增加 `sp` 指针的值，释放栈空间**。同时，恢复之前被保存的寄存器原有的值。这就是为何 C 语言的函数调用和栈有着很大的相似性的原因：在函数调用过程中，编译器的确为我们维护了一个栈。这下同学们应该也不难理解，为什么复杂函数在递归层数过多时会导致程序崩溃，也就是我们常说的“栈溢出”。

Note A.5.2 `ra` 寄存器存放了函数的返回地址。使得被调用的函数结束时得以返回到调用者调用它的地方。但你有没有想过，我们其实可以将这个返回点设置为别的函数的入口，使得该函数在返回时直接进入另一个函数中，而不是回到调用者哪里？一个函数调用了另一个函数，而返回时，返回到第三个函数中，是不是也是一种很有价值的编程模型呢？如果你对此感兴趣，可以了解一下函数式编程中的 Continuations 的概念 (推荐 Functional Programming For The Rest Of Us 这篇文章)，在很多新近流行起来的语言中，都引入了类似的想法。

在我们看到了一个函数作为被调用者做了哪些工作后，我们再来看看，作为函数的调用者需要做些什么？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[8]

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

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

A.5.3 通用寄存器使用约定

为了和编译器等程序相互配合，我们需要遵循一些使用约定。这些规定与硬件无关，硬件并不关心寄存器具体被用于什么用途。这些规定是为了让不同的软件之间得以协同工作而制定的。

[7]其实这样说并不准确，后面我们会看到，有些寄存器的值是由调用者负责保存的，有些是由被调用者保存的。但这里为了理解方便，我们姑且认为被调用的函数保存了调用者的所有状态吧

[8]为了方便你了解自己手写汇编时应当怎样写，我们这一次采用汇编代码，而不是反汇编代码。这里注意，`fp` 和上面反汇编出的 `s8` 其实是同一个寄存器，只是有两个不同的名字而已

MIPS 中一共有 32 个通用寄存器 (General Purpose Registers)，其用途如表A.5所示。

表 A.5: MIPS 通用寄存器

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

其中，只有 16-23 号寄存器和 28-30 号寄存器的值在函数调用的前后是不变的[9]。对于 28 号寄存器有一个特例：当调用位置无关代码 (position independent code) 时，28 号寄存器的值是不被保存的。

除了这些通用寄存器之外，还有一个特殊的寄存器：PC 寄存器。这个寄存器中储存了当前要执行的指令的地址。当你在 QEMU 仿真器上调试内核时，可以留意一下这个寄存器。通过 PC 的值，我们就能够知道当前内核在执行的代码是哪一条，或者触发中断的代码是哪一条等等。

A.5.4 LEAF、NESTED 和 END

在我们的实验中，可以在 init/start.S 里找到 LEAF(_start) 这样的代码，其中的 LEAF 其实是一个使用.macro 定义的宏。那么首先让我们来了解一下 LEAF、NESTED 和 END 这三个宏吧。

> **Note A.5.3** 阅读 LEAF 等宏定义的时候，我们会发现这些宏的定义是一系列以. 开头的指令。这些指令不是 MIPS 汇编指令，而是 Assembler directives(参考 https://ftp.gnu.org/old-gnu/Manuals/gas-2.9.1/html_chapter/as_7.html)，它们的主要作用是向汇编器提供细节信息，以辅助生成目标代码。

[9]请注意，这里的不变并不意味着它们的值在函数调用的过程中不能被改变。只是指它们的值在函数调用后和函数调用前是一致的。

通过 grep 进行查找，发现可以在 include/asm/asm.h 中发现这三个宏定义。

首先让我们来看 LEAF 和 NESTED 的宏定义，两个宏的内容基本一致，仅在最后一行有区别。

```
 /*
 * LEAF - declare leaf routine
 */
#define LEAF(symbol)                \
  .globl symbol;                    \
  .align 2;                         \
  .type symbol, @function;          \
  .ent symbol;                      \
  symbol:                           \
  .frame sp, 0, ra

/*
 * NESTED - declare nested routine entry point
 */
#define NESTED(symbol, framesize, rpc)        \
  .globl symbol;                              \
  .align 2;                                   \
  .type symbol, @function;                    \
  .ent symbol;                                \
  symbol:                                     \
  .frame sp, framesize, rpc
```

下面我们逐行来理解这些宏定义。

第一行是对 LEAF 宏的定义，后面括号中的 symbol 类似于函数的参数，在宏定义中的作用类似，编译时在宏中会将 symbol 替换为实际传入的文本。

第二行中，.globl 的作用是“使标签对链接器可见”，这样即使在其它文件中也可以引用到 symbol 标签，从而使得其它文件中可以调用我们使用宏定义声明的函数。

第三行中，.align 的作用是“使下面的数据进行地址对齐”，这一行语句使得下面的 symbol 标签按 4 Byte 进行对齐（参数 x 代表以 2^x 字节对齐），从而使得我们可以使用 jal 指令跳转到这个函数（末尾拼接两位 0）。

第四行中，.type 的作用是设置 symbol 标签的类别，在这里我们设置了 symbol 标签为函数标签。

第五行中，.ent 的作用是标记每个函数的开头，需要与.end 配对使用。这些标记使得可以在 Debug 时查看调用链。

第六行的开头便是 symbol 标签了，后面的.frame 的用法如下。

```
  .frame framereg, framesize, returnreg
```

第一个参数 framereg 是用于访问栈帧的寄存器，通常我们使用栈寄存器 sp，在创建栈帧时，sp 寄存器自减，栈空间向低地址增长一段内存用于栈帧的储存。

第二个参数 framesize 是栈帧占据的存储空间大小。

第三个参数 returnreg 是存储函数执行完的返回地址的寄存器。通常我们传入 $0 表示返回地址储存在栈帧空间中，有时在不需要返回的函数中我们会传入 $ra 表示返回地址存储在 $ra 寄存器中（$31）。

Note A.5.4 在.frame 的使用时，出现了概念“栈帧”，每个栈帧对应着一个未运行完的函数。栈帧中保存了该函数的返回地址和局部变量。在上一节的“函数调用”小节中有所介绍。

通常来说，栈帧的作用包括但不限于：存储函数的返回地址、存储调用方的临时变量与中间结果、向被调用方传递参数。

通过对比，我们可以发现 LEAF 宏和 NESTED 宏的区别就在于 LEAF 宏定义的函数在被调用时没有分配栈帧的空间记录自己的“运行状态”，NESTED 宏在被调用时分配了栈帧的空间用于记录自己的“运行状态”。

下面让我们来看 END 宏。

```
1  #define END(function)                    \
2          .end    function;                \
3          .size   function,.-function
```

第一行是对 END 宏的定义，与上面 LEAF 与 NESTED 类似。

第二行的.end 是为了与先前 LEAF 或 NESTED 声明中的.ent 配对，标记了 symbol 函数的结束。

第三行的.size 是标记了 function 符号占用的存储空间大小，将 function 符号占用的空间大小设置为 .-function，. 代表了当前地址，当前位置的地址减去 function 标签处的地址即可计算出符号占用的空间大小。

A.6 多级页表与页目录自映射

A.6.1 MOS 中的页目录自映射应用

在 Lab2 中，实现了内存管理，建立了两级页表机制。

页表的主要作用是维护虚页面到物理页面之间的映射关系，通常存放在内存中。操作系统对于页表的访问也是通过虚地址来进行。这也就意味着，页表同时也维护了自身所处的虚页面到实际物理页面之间的映射关系。

试想这样一个问题：如何在虚拟存储空间维护页表和页目录？下面介绍一下 MOS 中采用的“自映射”，即将页表和页目录映射到进程地址空间的实现方式。在两级页表中，一个进程的 4GB 地址空间均映射物理内存的话，那么就需要 4MB 来存放页表（1024 个页表），4KB 来存放页目录；如果页表和页目录都在进程的地址空间中得到映射，这意味着在 1024 个页表中，**有一个页表所对应的 4MB 空间就是这 1024 个页表占用的 4MB 空间**。这一个特殊的页表就是页目录，它的 1024 个表项映射到这 1024 个页表。因此只需要 4MB 的空间即可容纳页表和页目录。

而 MOS 中，将页表和页目录映射到了用户空间中的 0x7fc00000-0x80000000（共 4MB）区域，这意味着 MOS 中允许在用户态下通过 UVPT 访问当前进程的页表和页目录。

下面根据自映射的性质，计算出 MOS 中页目录的基地址：

0x7fc00000-0x80000000 这 4MB 空间的起始位置（也就是第一个二级页表的基地址）对应着页目录的第一个页目录项。同时由于 1M 个页表项和 4GB 地址空间是线性映射的，不难算出 0x7fc00000 这一个地址对应的应该是第 0x7fc00000 >> 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。

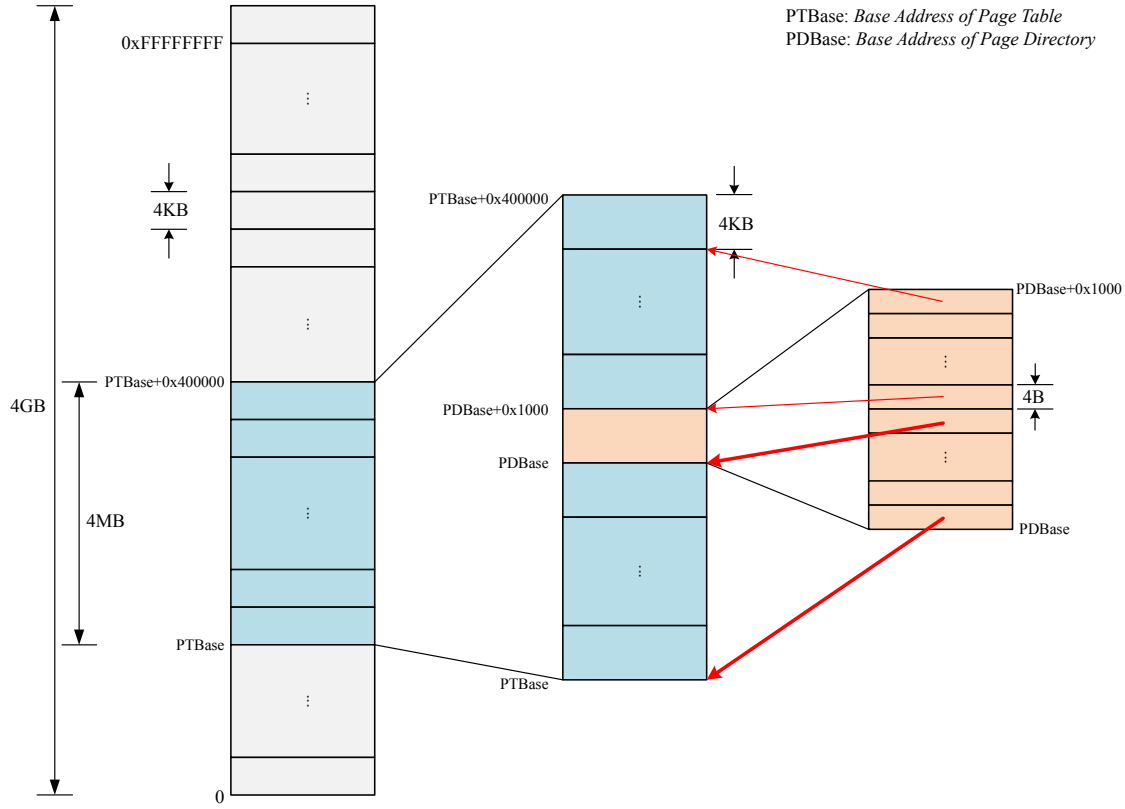


图 A.3: 页目录自映射

A.6.2 其他页表机制

在其他系统中，还会使用三级页表等更多级的页表机制。请结合操作系统理论课所学知识，查阅相关资料，回答下述思考题。

Thinking A.1 在现代的 64 位系统中，提供了 64 位的字长，但实际上不是 64 位页式存储系统。假设在 64 位系统中采用三级页表机制，页面大小 4KB。由于 64 位系统中字长为 8B，且页目录也占用一页，因此页目录中有 512 个页目录项，因此每级页表都需要 9 位。因此在 64 位系统下，总共需要 $3 \times 9 + 12 = 39$ 位就可以实现三级页表机制，并不需要 64 位。

现考虑上述 39 位的三级页式存储系统，虚拟地址空间为 512 GB，若三级页表的基地址为 $\mathrm{PT}_{\mathrm{base}}$，请计算：

- 三级页表页目录的基地址。
- 映射到页目录自身的页目录项（自映射）。

■

A.6.3 虚拟内存和磁盘中的 ELF 文件

我们将通过一个实例，探讨 `bss` 段在虚拟内存和磁盘 ELF 文件中是否占据空间这一问题。

> `testbss` 测试

```c
#include <lib.h>
#define ARRAYSIZE 0x1000000
int bigarray[ARRAYSIZE] = {0};
int main(int argc, char **argv) {
        int i;

        debugf("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);
                }
        }
        debugf("Bss is good\n");
        return 0;
}
```

可以参考 Lab4 中 `pingpong` 和 `fktest` 的编译与加载流程，完成 `testbss` 的测试：在 `init/init.c` 中创建相应的初始进程，观察相应的实验现象。如果能正确运行，则说明我们 Lab3 完成的 `load_icode` 系列函数正确地完成了在内核态中加载 ELF 时 `bss` 段的初始化。

正确结果展示：

```
Making sure bss works right...
Bss is good
```

验证了 `bss` 段被 `load_icode` 系列函数正确初始化后，我们可以对 `user/testbss.c` 的全局数组 `bigarray` 做以下三种不同的初始化处理，分别 `make clean && make` 后对三次的 `bss.b` 进行命令解析：

- 执行 `size user/testbss.b` 命令，`size` 命令的前四列结果默认为十进制显示，分别代表 `text` 段、`data` 段、`bss` 段、各段之和的大小，展示虚拟内存空间的分配信息。分析比较三次结果的 `data` 段与 `bss` 段大小，思考原因。
- 执行 `ls -l user/testbss.b` 命令，查看二进制文件的详细信息，找到结果中的第五列也就是月份前的那列数字，它表示文件在磁盘中占据的大小，分析比较三次结果并思考原因。

正确结果展示：

- `testbss.c` 第三行为 `int bigarray[ARRAYSIZE]` ，即不对全局变量初始化时：

```
   text    data     bss     dec     hex filename
  14280   13851   40964   69095   10de7 user/testbss.b

  -rwxrwxr-x 1 git git 41761 MMM dd HH:mm user/testbss.b
```

- `testbss.c` 第三行为 `int bigarray[ARRAYSIZE]={0}` ，即全局变量初始化为 0 时：

```
   text    data     bss     dec     hex   filename
  14280   13851   40964   69095   10de7   user/testbss.b

  -rwxrwxr-x 1 git git 41761 MMM dd HH:mm user/testbss.b
```

- testbss.c 第三行为 int bigarray[ARRAYSIZE]={1} ，即对全局变量做初始化时：

```
   text    data   bss      dec     hex   filename
  14280   54811     4    69095   10de7   user/testbss.b

  -rwxrwxr-x 1 git git 82721 MMM dd HH:mm user/testbss.b
```

观察比对结果，我们可以得出以下三点：

- 当第三次全局变量初始化时，相较于第一次不初始化的结果，data 段增加了 40960 字节，bss 段减少了 40960 字节。bigarray 数组的大小为 10240 个 int，也就是 10240*4 Byte=40960 Byte。说明在虚拟内存布局中，初始化的全局变量保存在 data 段，未初始化的全局变量保存在 bss 段。
- 当第二次全局变量初始化为 0 时，与第一次不初始化的结果完全一致。说明对于未初始化和初始化为 0 的全局变量，二者在用户空间地址中的数据分配的存放位置相同，均在 bss 段。
- 第三次初始化全局变量时，ELF 文件在磁盘的大小为 82721 字节。而全局变量保存在 bss 段时，ELF 文件在磁盘的大小为 41761 字节。82721Byte-41761Byte=40960Byte，说明 bss 段不在磁盘文件占据空间。

bss 段 (Block Started Symbol，意为“以符号开始的块”)，存放全局未初始化/初始化为 0、静态未初始化/初始化为 0 的变量，只是简单维护地址空间中开始和结束的地址，在实际运行对内存区域有效地清零即可。

Note A.6.1 第三次 testbss.c 中似乎没有未初始化的全局变量，bss 段的 4 字节存放的什么变量呢？

我们可以使用 objdump -t user/testbss.b | grep .bss 命令反汇编查看变量名及其位置，将反汇编的结果经过管道，筛选显示包含“.bss”的内容。

可以找到位于 .bss 段的一条结果：00412000 g O .bss 00000004 env。

env 变量并不在 testbss.c 中定义。我们打开 user/Makefile，查看 testbss.b 的依赖文件，entry.o 由 user/lib/entry.S 编译而来。user/lib/entry.S 的 _start: 后跳转到 libmain() 函数。libmain 函数位于 user/lib/libos.c 中，它为未初始化的 env 变量赋值后跳转到 main() 函数，也就是我们自己在 user 目录下编写的测试文件的主函数。env 变量在 user/lib/libos.c 中被定义。