哈尔滨工业大学计算机科学与技术学院

2017年秋季学期《操作系统》实验报告

学生姓名： 马玉坤

学生学号： 1150310618

联系方式： 18845895386

指导老师： 曲明成

二〇一八年一月

# 摘 要

麻省理工学院（Massachusetts Institute of Technology，MIT）的操作系统课程 [1]（Operating System Engineering）是面向本科生的一门操作系统课程，本门课程有 7 个实验。本人基于对本课程前 6 个实验的理解，在本人完成该 6 个实验的基础上，撰写了该报告。

关键词: 6.828; 操作系统; 麻省理工学院; MIT

# 目 录

摘要 **I**

1. **xv6** 实验系统简述 **1**
2. **xv6** 环境配置 **2**
   1. 系统环境 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
   2. 编译环境 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
   3. 安装 QEMU . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
3. 实验过程 **6**
   1. Lab 1: Booting a PC . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
      1. 实验简介 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
      2. 实验目的 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
      3. 实验内容 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
         1. 准备 . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
         2. 第一部分：启动 PC . . . . . . . . . . . . . . . . . . 7
         3. 第二部分：The Boot Loader . . . . . . . . . . . . . . 9
         4. 第三部分：内核 . . . . . . . . . . . . . . . . . . . . 13
   2. Lab 4: Preemptive Multitasking . . . . . . . . . . . . . . . . . . . . . . 18
      1. 实验简介 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
      2. 实验目的 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
      3. 实验内容 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
         1. 准备工作 . . . . . . . . . . . . . . . . . . . . . . . . 19
         2. 第一部分: 多处理器支持及协作式多任务 . . . . . . 19
         3. 第二部分: 写时拷贝 . . . . . . . . . . . . . . . . . . 35
         4. 第三部分: 抢占式多任务及进程间通信 . . . . . . . . 43
4. **xv6** 实验系统分析 **48**
   1. xv6 的系统结构 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
   2. xv6 的引导 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
   3. xv6 的进程与调度分析 . . . . . . . . . . . . . . . . . . . . . . . . . . 48
   4. xv6 的内存管理 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
   5. xv6 的文件系统 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
   6. xv6 的 I/O . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
5. 总结与心得 **49**

参考文献 **50**

附录 **A** 实验代码 **51**

A.1 通用搜索算法 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

# **1 xv6**实验系统简述

xv6[2][3] 是由麻省理工学院 (MIT) 为操作系统工程的课程 (代号 6.828), 开发的一个教学目的的操作系统。xv6 是在 x86 处理器上 (x 即指 x86) 用 ANSI 标准 C 重新实现的 Unix 第六版 (Unix V6，通常直接被称为 V6)。

截至 2017 年 12 月，xv6 的最新版本为 rev10。

xv6 使用 GNU C 编译器编译，并支持 ELF 可执行文件格式。（由于 OS X 不支持 ELF 可执行文件，所以需要交叉编译器[[1]](#footnote-1)）。

xv6 总代码行数只有 9299 行[[2]](#footnote-2)。但是麻雀虽小，五脏俱全，包括了 Unix 的重要概念和组织结构，因此我个人认为特别适用于本科的操作系统教学中。

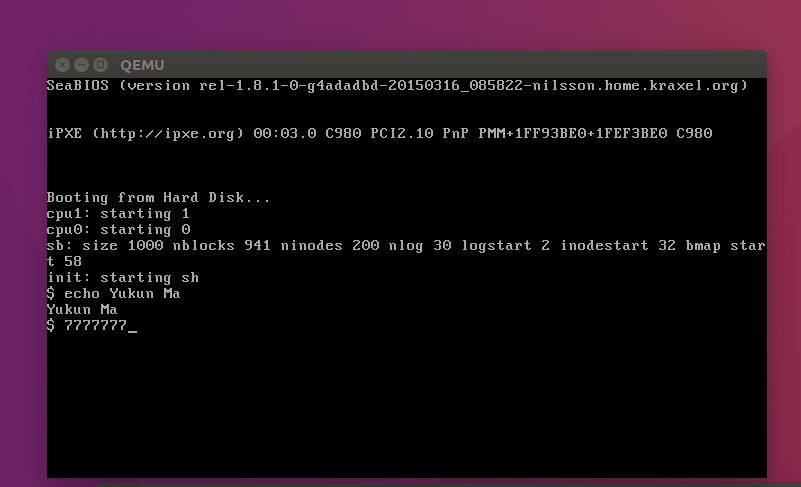


图 1.1 xv6 运行截图

**2 xv6**环境配置

## **2.1** 系统环境

建议使用 32 位 Linux 操作系统，64 位 Linux 操作系统也可以。这里我在 Windows 上用 Virtual Box 安装了 Ubuntu 14.04 32 位版。

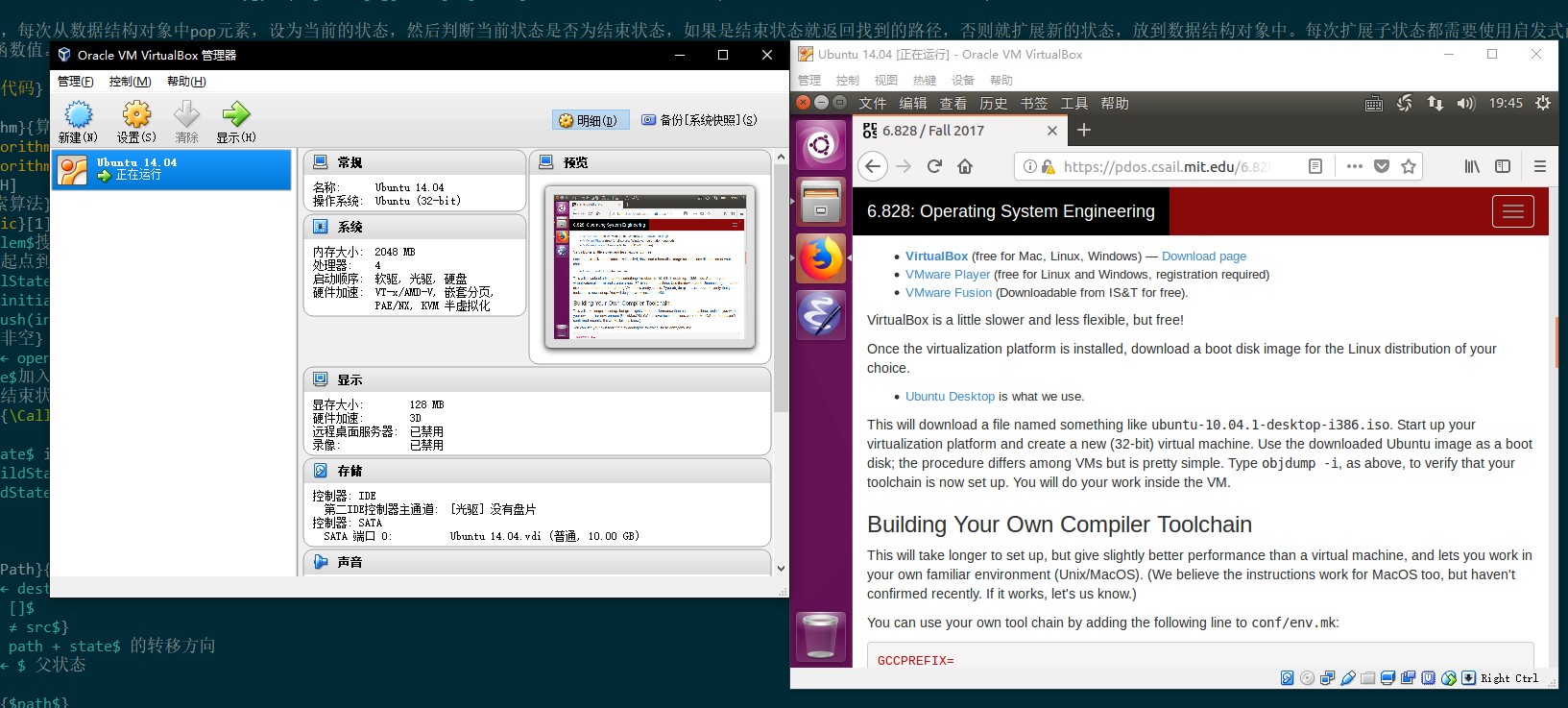


图 2.1 Ubuntu 虚拟机

**2.2** 编译环境

先用以下命令检查编译环境是否装好。

# objdump -i

第二行应该是 **elf32-i386**。

# gcc -m32 -print-libgcc-file-name

这个命令应该打印一些类似于**Thecommandshouldprintsomethinglike/usr/lib/gcc/i486-**

**linux-gnu/version/libgcc.a or /usr/lib/gcc/x86\_64-linux-gnu/version/32/libgcc.a** 的东西。还需要安装 GCC（GNU C Compiler）。

# sudo apt-get install -y build-essential gdb

如果是 64 位系统，还需要：

# sudo apt-get install gcc-multilib

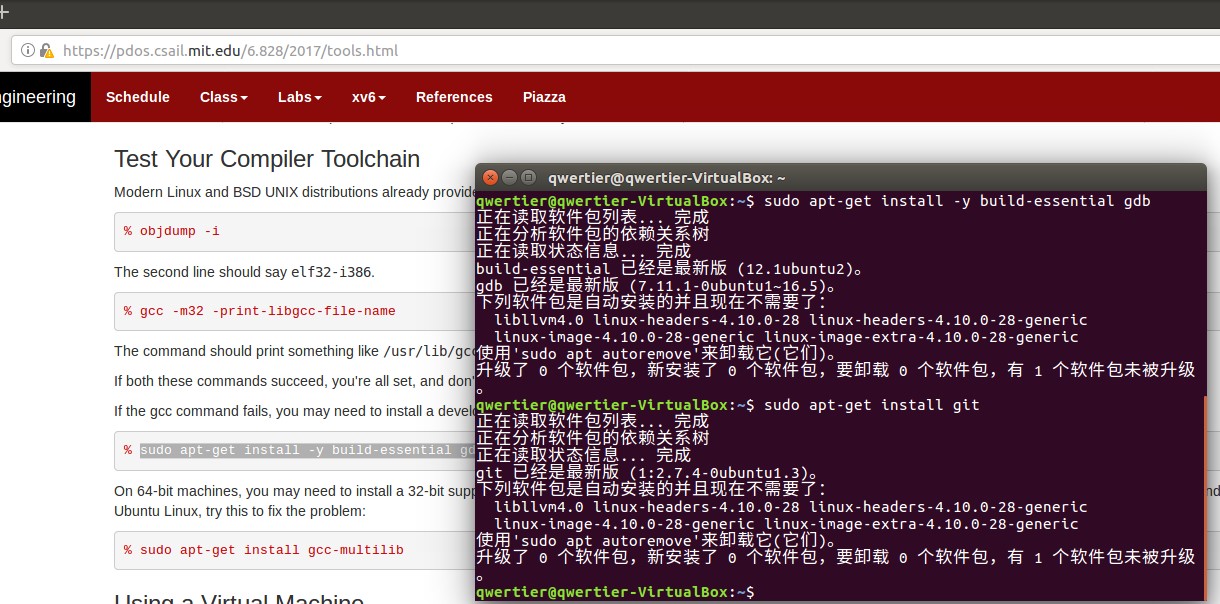


图 2.2 安装所需的软件

**2.3** 安装 **QEMU**

实验需要用到 QEMU 模拟器，但是普通的 QEMU 会出 bug，所以需要修改版

的。如图2.3，先要使用 git clone 下修改版 QEMU。

# git clone https://github.com/geofft/qemu.git

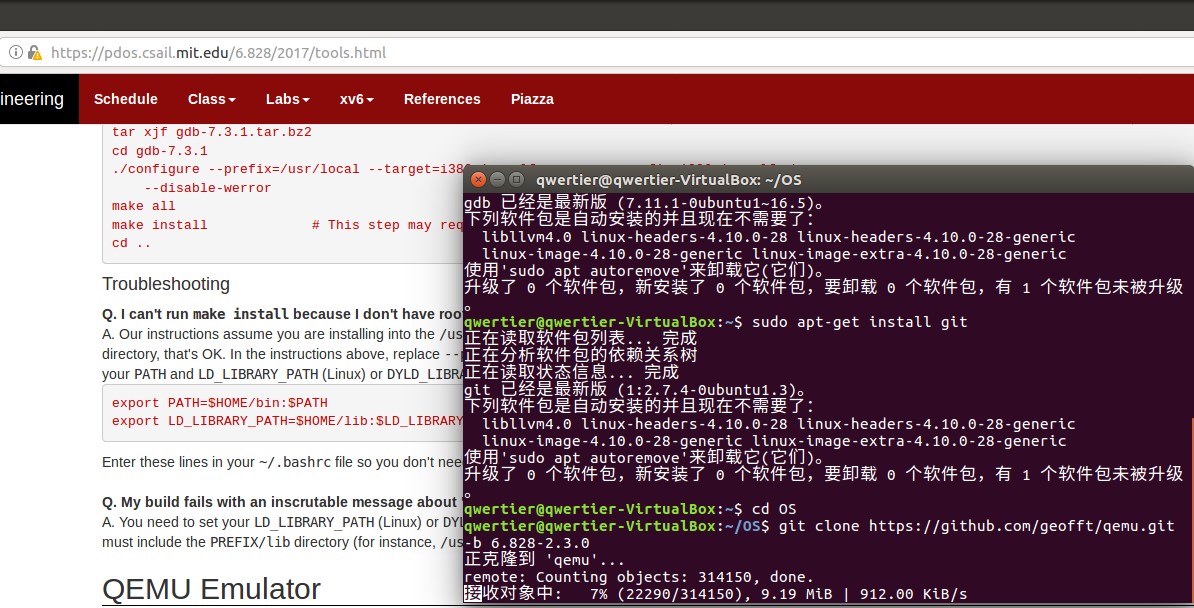


图 2.3 git clone QEMU 的过程

如图2.4，运行进入到 qemu 路径下，使用 configure。

# ./configure --disable-kvm --target-list="i386-softmmu x86\_64-softmmu"

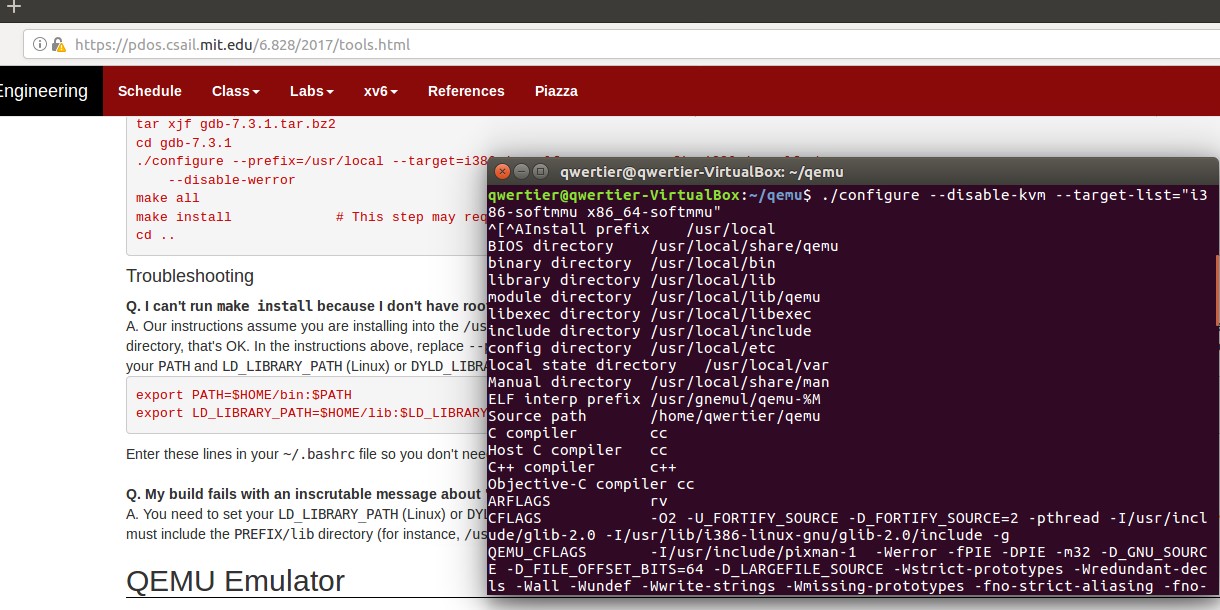


图 2.4 configure QEMU 的过程

如图2.5，使用 make 命令编译 qemu。

# sudo make && sudo make install

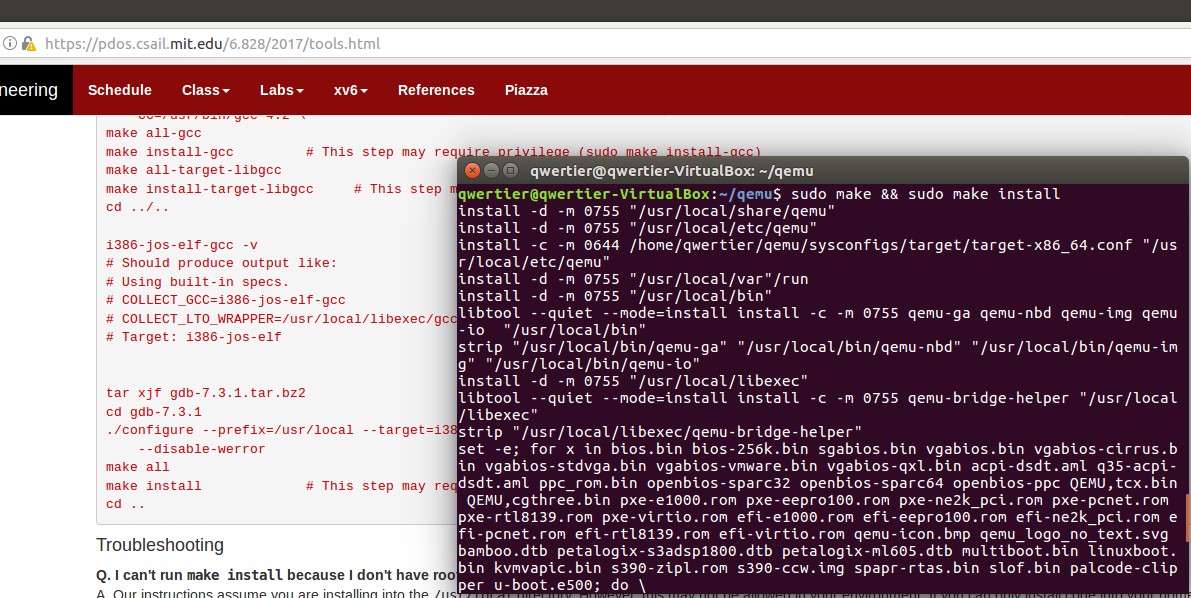


图 2.5 make QEMU 的过程

然后 QEMU 就安装成功了。

# **3** 实验过程

**3.1 Lab 1: Booting a PC**

**3.1.1** 实验简介

本实验被分为三个部分。第一个部分主要集中在熟悉 x86 汇编语言、QEMU 模拟器以及 PC 的启动步骤。第二个部分通过实践来对系统内核的 boot loader 进一步了解。第三部分来深入研究 JOS 的系统内核。

**3.1.2** 实验目的

1. 熟悉（复习）x86 汇编语言，为接下来的工作打好基础
2. 了解开发环境的使用方法，尤其是 QEMU 模拟器的使用
3. 对 PC 启动的过程有更清晰的了解
4. 了解堆栈，清楚函数调用时的栈帧结构

**3.1.3** 实验内容

**3.1.3.1** 准备

装完 QEMU 模拟器之后，如果想做后续的实验，还需要下载 JOS 的残缺的源码（需要我们在后面填充内容）。

# git clone https://pdos.csail.mit.edu/6.828/2017/jos.git lab

# cd lab

进入 lab 目录之后，如图3.1，我们可以看到 lab 目录的大致结构。

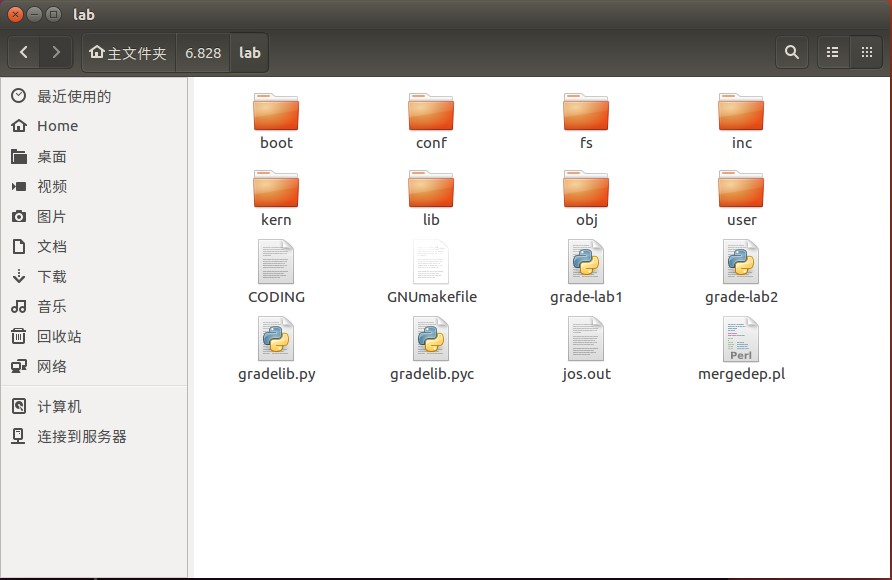


图 3.1 lab 目录的结构

在 lab 目录下运行如下命令：

# make qemu

如图3.2，就能用 QEMU 运行 JOS。

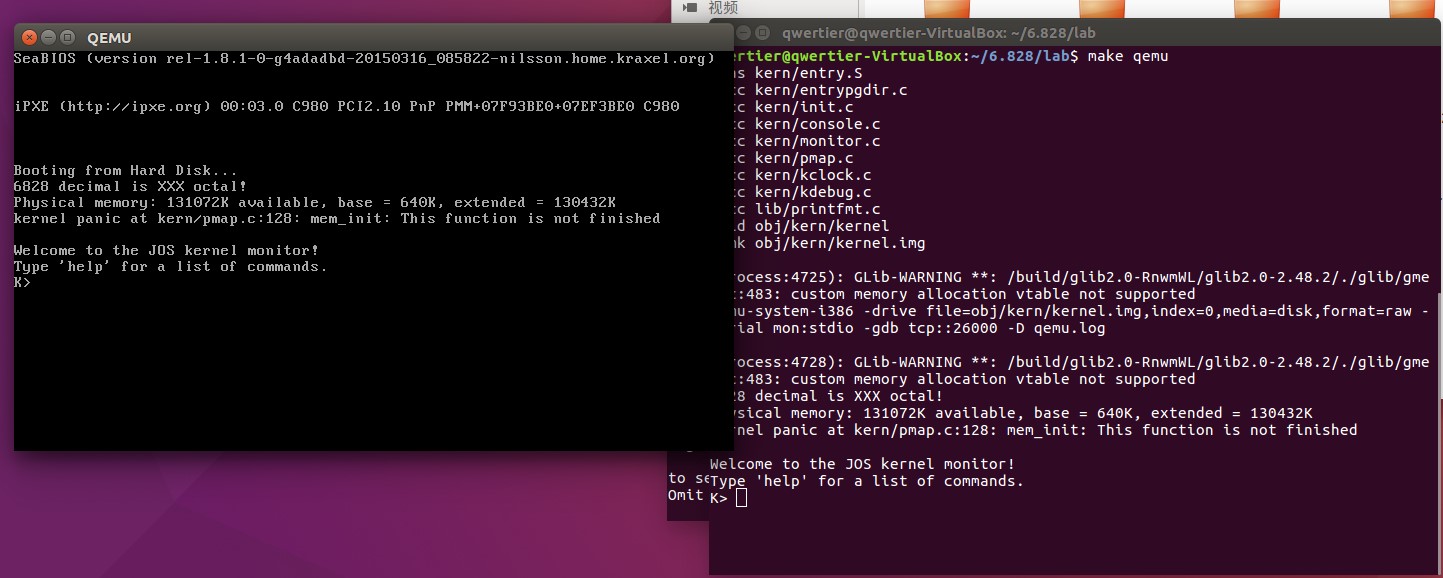


图 3.2 lab 目录的结构

**3.1.3.2** 第一部分：启动 **PC**

**Ex. 1 —** 熟悉汇编语言汇编语言的英文资料可以在[https://pdos.csail.mit.edu/6.828/2017/reference. html](https://pdos.csail.mit.edu/6.828/2017/reference.html)找到。

**Ex. 2 —** 使用 GDB 的单步调试 (si) 语句追踪 ROM BIOS，然后猜测它在做什么。在一个终端中输入 make qemu-gdb ，另一个终端输入 make gdb 。开始调试程

序。

# [f000:fff0] 0xffff0: ljmp $0xf000,$0xe05b

是 GDB 反汇编出的第一条执行指令，这条指令表明了：IBM PC 执行的起始物理地址为 0x000ffff0，PC 的偏移方式为 CS = 0xf000，IP = 0xfff0。

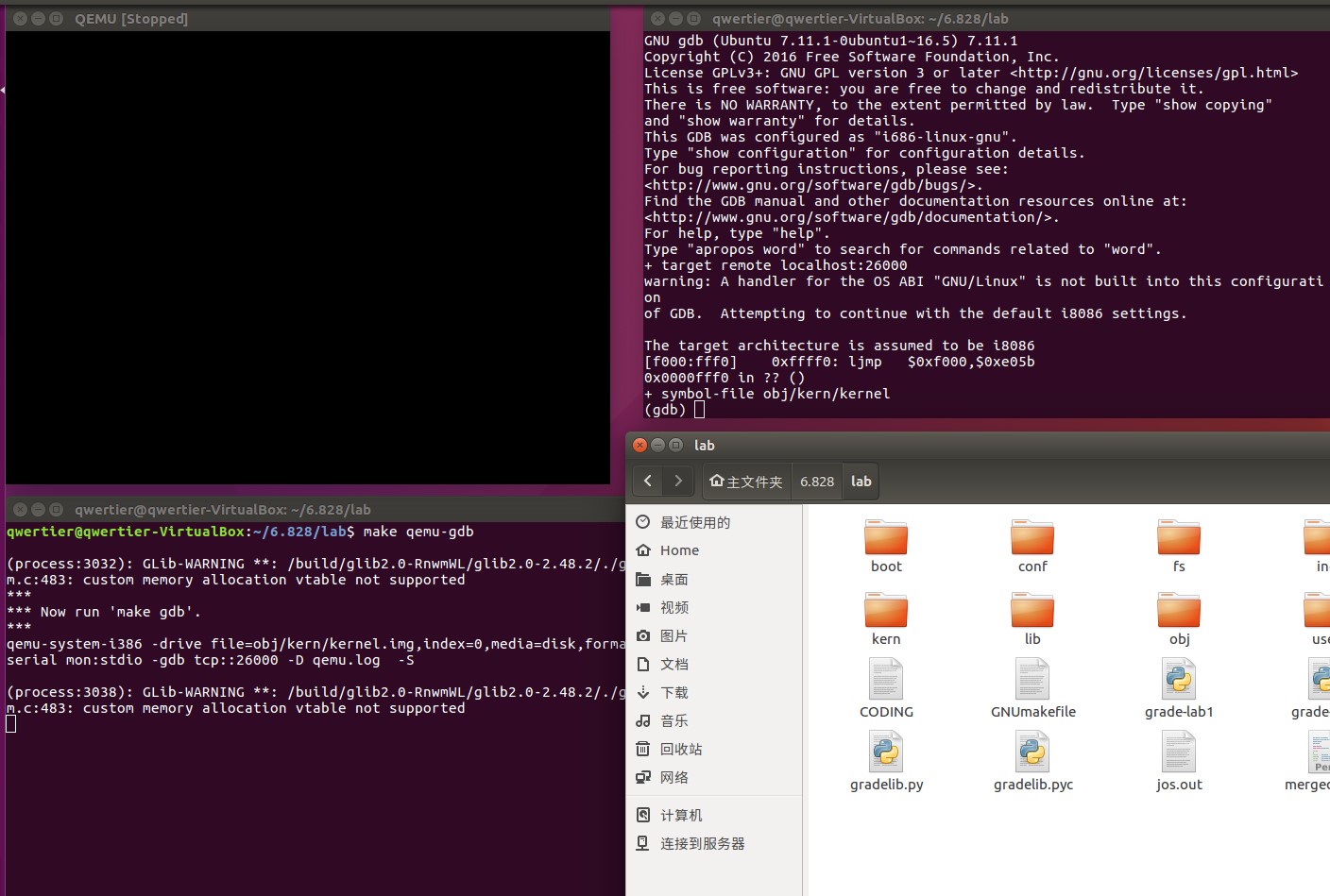


图 3.3 使用 gdb 调试第一条指令

第一条指令执行的是 jmp 指令，跳转到段地址 CS = 0xf000，IP = 0xe05b。QEMU 模拟了 8088 处理器的启动，当启动电源，BIOS 最先控制机器，这时还没有其他程序执行，之后处理器进入实模式也就是设置 CS 为 0xf000，IP 为 0xfff0。在启动电源也就是实模式时，地址转译根据这个公式工作：物理地址 = 16 \* 段地址 + 偏移量。所以 PC 中 CS 为 0xf000 IP 为 0xfff0 的物理地址为：

16∗0*xf*000+0*xfff*0 (3.1)

|  |  |
| --- | --- |
| =0*xf*0000+0*xfff*0 | (3.2) |
| =0*xffff*0 | (3.3) |

0xffff0 在 BIOS (0x100000) 的结束地址之前。

当 BIOS 开始运行时，它会建立中断描述表[[3]](#footnote-3)然后初始化众多设备（例如显示器）。这就是当你从 QEMU 中看到”Starting SeaBIOS” 的时候。

当所有的其他设备都初始化完成了，BIOS 会继续找可以启动的设备（例如硬盘、CD-ROM）。最后它会找到可启动盘，接着 BIOS 从启动盘中读出 boot loader 然后转移控制权。

**3.1.3.3** 第二部分：**The Boot Loader**

**Ex. 3 —** 在<https://pdos.csail.mit.edu/6.828/2017/labguide.html>熟悉一下GDB 的指令。然后在 0x7c00 处设一个断点，执行到断点。进入 boot/boot.S，参考源代码和反汇编代码 *obj/boot/boot.asm* 来跟踪。追踪到 bootmain 函数中，而且还要具体追踪到 readsect() 子函数里面。找出和 readsect()c 语言程序的每一条语句所对应的汇编指令，回到 bootmain()，然后找出把内核文件从磁盘读取到内存的那个 for 循环所对应的汇编语句。找出当循环结束后会执行哪条语句，在那里设置断点，继续运行到断点，然后运行完所有的剩下的语句。

**$[0:7c2d] => 0x7c2d: ljmp $0x8,$0x7c32** 这条指令之后，也就是 boot.S 中的 **ljmp $PROT\_MODE\_CSEG, $protcseg** ，地址符号就变成 0x7c32 了。

首先查看 boot.S 文件，在开头可以看到如下代码，cld 是串操作指令，用来操

作方向标志位 DF，使 DF=0。 start:

.code16 cli

cld *# String operations increment*

如以下代码，**inb $0x64,%al** 把0*x*64端口 (8042 键盘控制器) 的状态写入 al 中（inb 代表 IO 端口读）, 之后 **testb $0x2,%al** 判断 al 的第二位是否为 0，不为 0 就循环执行 seta20.1。这里第二位代表输入缓冲区是否满了。接着0*xd*1放入0*x*64端口。最后将0*xdf* 放入 0x60 端口，代表开启 A20 地址线了。

*# Enable A20:*

*# For backwards compatibility with the earliest PCs, physical*

*# address line 20 is tied low, so that addresses higher than*

*# 1MB wrap around to zero by default. This code undoes this.* seta20.1:

|  |  |
| --- | --- |
| inb $0x64,%al testb $0x2,%al jnz seta20.1 | *# Wait for not busy* |
| movb $0xd1,%al outb %al,$0x64  seta20.2: | *# 0xd1 -> port 0x64* |
| inb $0x64,%al testb $0x2,%al jnz seta20.2 | *# Wait for not busy* |
| movb $0xdf,%al | *# 0xdf -> port 0x60* |

outb %al,$0x60 通过如下代码，开始调用 *bootmain* 函数。

*# Set up the stack pointer and call into C.* movl $start, %esp call bootmain

如下代码所示，在 bootmain 函数中，调用 readseg 函数。该函数有三个参数，

分别是物理地址、页大小和偏移量。

*# read 1st page off disk* readseg((uint32\_t) ELFHDR, SECTSIZE\*8, 0)*;* push $0x0 push $0x1000 push $0x10000 call 7cdc <readseg> 如下指令用于加载程序段。

mov 0x1001c,%eax movzwl 0x1002c,%esi lea 0x10000(%eax),%ebx shl $0x5,%esi add %ebx,%esi

**Ex. 4 —** 了解 C 语言中关于指针的知识（已经学过了）为了理解 boot/main.c，需要了解 ELF 二进制文件。编译并链接比如 JOS 内核这样的 C 程序，编译器会将源文件 (.c) 转为包含汇编指令的目标文件 (.o)。接着链接器把所有的目标文件组合成一个单独的二进制镜像（binary image），比如 obj/kern/kernel，这种文件就是 ELF(是可执行可链接形式的缩写)。

当前只需要知道，可执行的 ELF 文件由带有加载信息的头，多个程序段表组成。每个程序段表是一个连续代码块或者数据，它们要被加载到内存具体地址中。 boot loader 不修改源码和数据，直接加载到内存中并运行。

ELF 开头是固定长度的 ELF 头，之后是一个可变长度的程序头，它列出了需要加载的程序段。ELF 头的定义在 inc/elf.h 中。主要学习以下 3 个程序段：

•.text: 程序执行指令

•.rodata: 只读数据，比如 ASCII 字符串

•.data: 存放程序初始化的数据段，比如有初始值的全局变量。使用以下命令可以查看 *obj/kern/kernel* 文件的 ELF 头的相关信息，结果如图3.4。

# objdump -h obj/kern/kernel

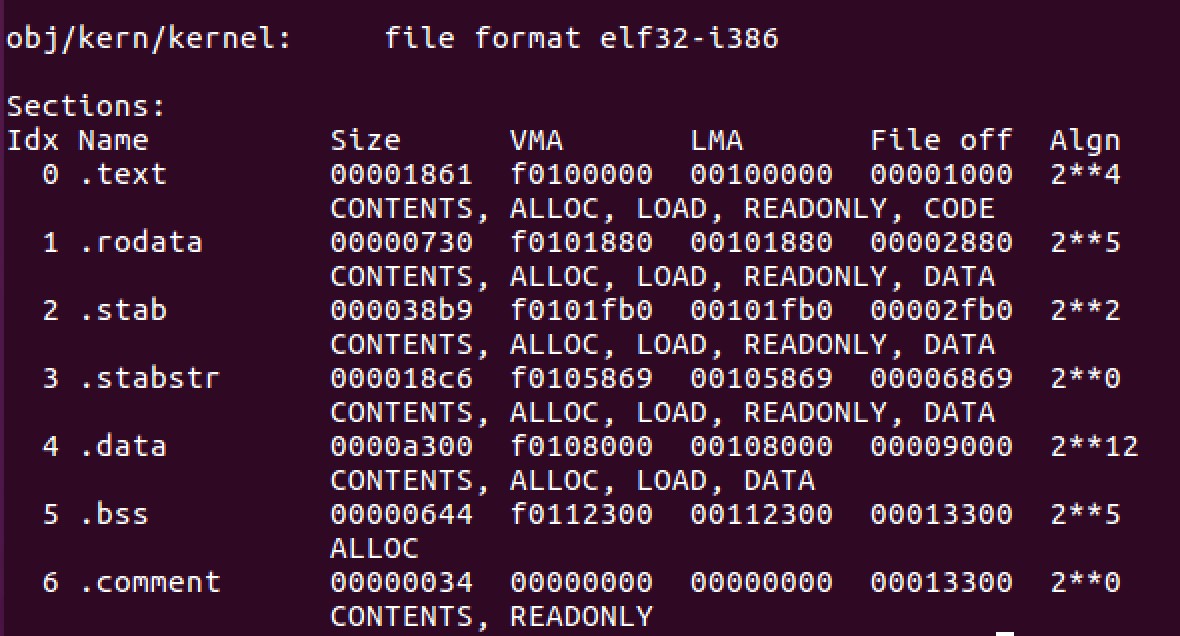


图 3.4 *obj/kern/kernel* 文件的 ELF 头

使用以下命令可以查看 *obj/boot/boot.out* 文件的 ELF 头的相关信息，结果如

图3.5所示。

# objdump -h obj/boot/boot.out

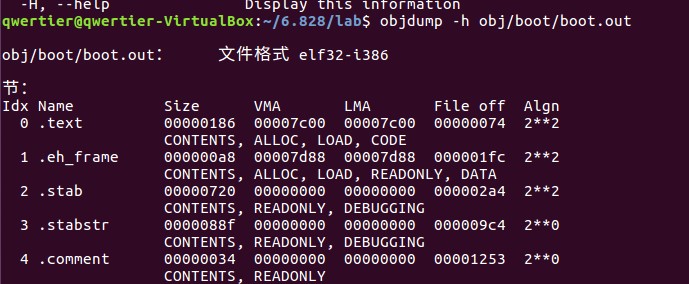


图 3.5 *obj/boot/boot.out* 文件的 ELF 头

**Ex. 5 —** 修改 *boot/Makefrag*，使得 boot loader 的加载地址出错，找到第一条出错的指令。[[4]](#footnote-4)查看 *boot/Makefrag* 文件，文件内容如下所示。可以发现**-Ttext** 后面的参数就

是入口地址。如果把这个值修改为0*x*8*C*00，保存后回到 lab1 文件夹下进行 make。

$(OBJDIR)/boot/boot: **$(**BOOT\_OBJS**)**

@echo + ld boot/boot

**$(**V**)$(**LD**) $(**LDFLAGS**)** -N -e start -Ttext 0x7C00 -o $@.out $^

**$(**V**)$(**OBJDUMP**)** -S $@.out >$@.asm

**$(**V**)$(**OBJCOPY**)** -S -O binary -j .text $@.out $@

**$(**V**)**perl boot/sign.pl **$(**OBJDIR**)**/boot/boot

如下代码，查看 *obj/boot/boot.asm* 会发现起始地址从原来的 00007c00 变为 00008c00。虽然此时在 0x7c00 处打断点然后运行时正常的，但是继续 si 以后会

在 **[0:7c2d] => 0x7c2d: ljmp $0x8,$0x8c32** 出循环，同时 qemu 端口出现了错误。因为不能 ljmp 到 $0x7c32 而是调到了 $0x8c32，所以无法执行正确的指令。查看 boot.asm 可以知道上面这个指令是 **ljmp $PROT\_MODE\_CSEG, $protcseg**，是为

|  |  |  |
| --- | --- | --- |
| 了进入 32 位模式的。 | |  |
| .set CR0\_PE\_ON,  .globl start start: | 0x1 | *# protected mode enable flag* |
| .code16 |  | *# Assemble for 16-bit mode* |
| cli |  | *# Disable interrupts* |
| 8c00: fa |  | cli |
| cld |  | *# String operations increment* |

8c01: fc cld

**Ex. 6 —** 使用 GDB 的 **x**[[5]](#footnote-5)命令可以可以查看内存信息。重启 QEMU，当 BIOS 进入 boot loader 时检查 0x00100000 处的 8 个字节，然后在 boot loader 进入内核时再看一次。为什么两者有不同？在第二个端点处到底有什么？

在 0x7c00 处设置断点，然后运行到断点处，使用 x/8x 0x100000 可以看到如图3.6

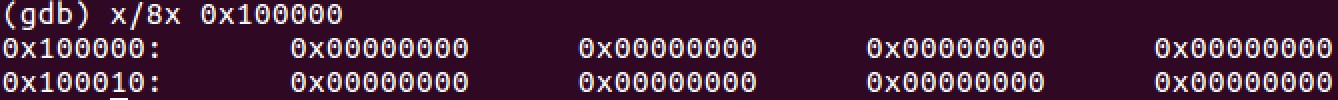


图 3.6 内存截图

根据之前的看到程序入口点是 0x10000c ，所以在 0x10000c 处打断点运行，如图3.7同样可以看到

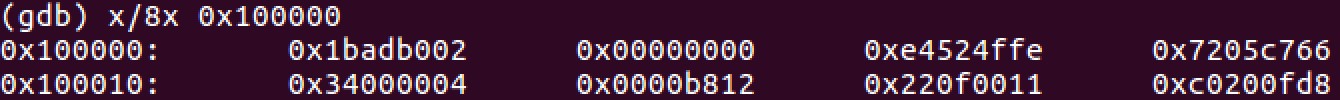


图 3.7 内存截图

0*x*100000处存放的其实就是程序指令段，也就是说 bootmain 函数会把内核的程序段送到内存0*x*100000处。

## **3.1.3.4** 第三部分：内核

boot loader 的链接地址和加载地址是一样的，然而 kernel 的链接地址和加载地址有些差异。查看 kern/kernel.ld 可以发现内核地址在 0xF0100000。

操作系统内核通常被链接并且运行在非常高的虚拟地址，比如文件里看到的

0xf0100000，为了让处理器虚拟地址空间的低地址部分给用户程序使用。

许多机器没有地址为 0xf0100000 的物理内存，所以内核不能放在那儿。因此使用处理器内存管理硬件将虚拟地址 0xf0100000 (内核希望运行的链接地址) 映射到物理地址 0x00100000 (boot loader 加载内核后所放的物理地址)。尽管内核虚拟地址很高，但加载进物理地址位于 1MB 的地方仅仅高于 BIOS 的 ROM。这需要

PC 至少有 1MB 的物理内存。

在下一个lab，会映射物理地址空间底部256MB，也就是0x00000000到0x0fffffff，到虚拟地址 0xf0000000 0xffffffff。所以 JOS 只使用物理内存开始的 256MB。

目前，只是映射了物理内存开始的 4MB，使用手写的静态初始化页目录和也表在 kern/entrypgdir.c。当 kern/entry.S 设置 CR0\_PG 标记，存储器引用就变为虚拟地址，即存储器引用是由虚拟存储器硬件转换为物理地址的虚拟地址。entry\_pgdir 将虚拟地址 0xf0000000 0xf0400000 转换为物理地址 0x00000000 0x00400000，虚拟地址 0x00000000 0x00400000 也转换为物理地址 0x00000000 0x00400000。任何不在这两个范围内的虚拟地址会导致硬件异常。

**Ex. 7 —** 追踪JOS内核并停在movl%eax, %cr0。查看内存0x00100000和0xf0100000。接着使用 stepi 来看上面两个地址里内容的变化。若注释了 kern/entry.S 的 movl %eax, %cr0, 查看第一个出现问题的指令是什么。查看kern/entry.S发现\_start是ELF入口点，exercise5提到了入口点是0x0010000c.

所以在 0x0010000c 处打断点。

# (gdb) b \*0x0010000c

注释 **movl %eax, %cr0** 后，make clean 之后重新编译，再运行。一步步 si 后出现了问题。



图 3.8 GDB 截图

在 0x10002a 处的 jmp 指令，要跳到 0xf010002c 处，然而因为没有分页管理，

不会进行虚拟地址映射到物理地址的转化，如图3.9，在另一个窗口可以看到错误信息，访问地址超出内存。

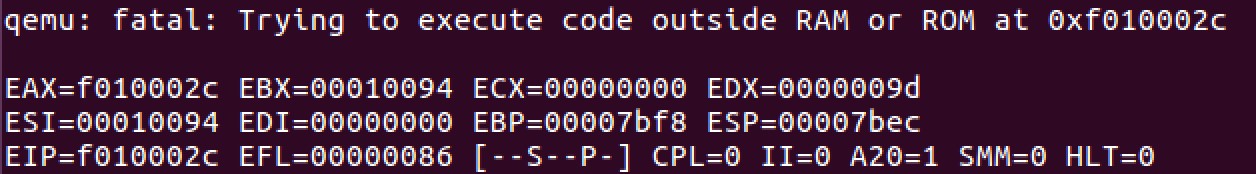


图 3.9 错误截图

**Ex. 8 —** 完成指定输出”%o” 格式字符串的代码。

要去实现%o 的格式化输出。在 lib/printfmt.c 可以看到要填写的地方。参考上

面 case ’u’ 的写法。 **case** 'o': num = getuint(&ap, lflag); base = 8; **goto** number;

修改完以后保存，make clean 之后运行，会发现启动以后，如图3.10，qemu 里 JOS 启动时会出现这样一行字。

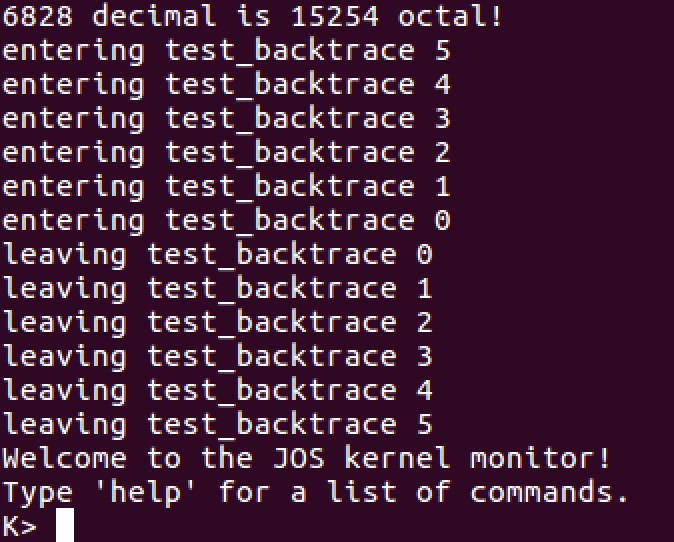


图 3.10 错误截图

**Ex. 9 —** 研究内核是在哪初始化堆栈，找出堆栈存放在内存的位置。内核是如何保存一块空间给堆栈的？堆栈指针指向这块区域的哪儿？看了几个文件以后，发现在 kern/entry.S 中提到了设置堆指针和栈指针。

*# Clear the frame pointer register (EBP)*

*# so that once we get into debugging C code,*

*# stack backtraces will be terminated properly.*

movl $0x0,%ebp *# nuke frame pointer*

*# Set the stack pointer* movl $(bootstacktop),%esp

为了查看堆的位置，所以要使用gdb，同样还是b\*0x10000c打断点进入entry。si 一步步执行，在 0x10002d: jmp \*%eax 之后，下一条指令变为 0xf010002f <relocated>: mov $0x0,%ebp。其实地址应该还是 0x10002f，所以这里的 0xf010002f 是因为开启的虚拟地址。

通过gdb发现0xf0100034<relocated+5>: mov$0xf0110000,%esp，也就是说%esp

也就是 bootstacktop 的值为 0xf0110000。其中 kern/entry.S 的 KSTKSIZE 应该就是堆栈的大小，通过跳转，发现在 inc/memlayout.h 里提到了堆栈。

*// Kernel stack.*

*#define KSTACKTOP KERNBASE*

*#define KSTKSIZE (8\*PGSIZE) // size of a kernel stack #define KSTKGAP (8\*PGSIZE) // size of a kernel stack guard*

**Ex. 10 —** 研究 obj/kern/kernel.asm 中 test\_backtrace 向堆栈里压入的信息。

使用单步调试和 info registers 来查看 esp 和 ebp 的变化。运行 test\_backtrace 前寄存器如下所示，

esp 0xf010ffdc 0xf010ffdc

ebp 0xf010fff8 0xf010fff8

现在需要实现 mon\_backtrace() 这个函数，需要显示 ebp，eip 和 args。ebp 是基

址指针，eip 是返回指令指针。简单实现 backtrace，实现效果如下：

Stack backtrace: ebp f0109e58 eip f0100a62 args 00000001 f0109e80 f0109e98 f0100ed2 00000031

ebp f0109ed8 eip f01000d6 args 00000000 00000000 f0100058 f0109f28 00000061 ... 代码：

**int** mon\_backtrace(**int** argc, **char** \*\*argv, **struct** Trapframe \*tf) {

*// Your code here.*

**int** i, j; **uint32\_t** \*ebp = (**uint32\_t**\*)read\_ebp(); **uint32\_t** eip; **struct** Eipdebuginfo info; cprintf("Stack backtrace:**\n**"); **while** (ebp != NULL) { eip = ebp[1]; cprintf(" ebp %08x eip %08x args", ebp, eip); debuginfo\_eip(eip, &info); **uint32\_t** \*args = ebp + 2; **for** (j = 0; j < 5; j ++) {

cprintf(" %08x", \*(args + j));

} cprintf("**\n**"); cprintf(" %s:%d: ", info.eip\_file, info.eip\_line); **for** (i = 0; i < info.eip\_fn\_namelen; i++) cprintf("%c", info.eip\_fn\_name[i]);

cprintf("+%d**\n**", eip-info.eip\_fn\_addr);

ebp = (**uint32\_t**\*)(ebp[0]);

} **return** 0;

}

**Ex. 11 —** 修改上面实现的backtrace，要显示详细的函数地址。可以使用kern/kdebug.c 的 debuginfo\_eip()。在查看debuginfo\_eip时发现其中有一段代码需要填写。这段代码是填写eip\_line。这里用到了写好的二分查找。

*// Your code here.* stab\_binsearch(stabs, &lline, &rline, N\_SLINE, addr); **if** (lline <= rline) {

info->eip\_line = stabs[lline].n\_desc;

} **else** { info->eip\_line = 0; **return** -1;

}

**Ex. 12 —** 将 backtrace 嵌入终端中，使其可以被调用。

只需要修改 kern/monitor.c 的这个部分：

**static struct** Command commands[] = {

{ "help", "Display this list of commands", mon\_help },

{ "kerninfo", "Display information about the kernel", mon\_kerninfo },

{ "backtrace", "Display a listing of function call frames", mon\_backtrace}

};

至此，Lab1 正式完成。

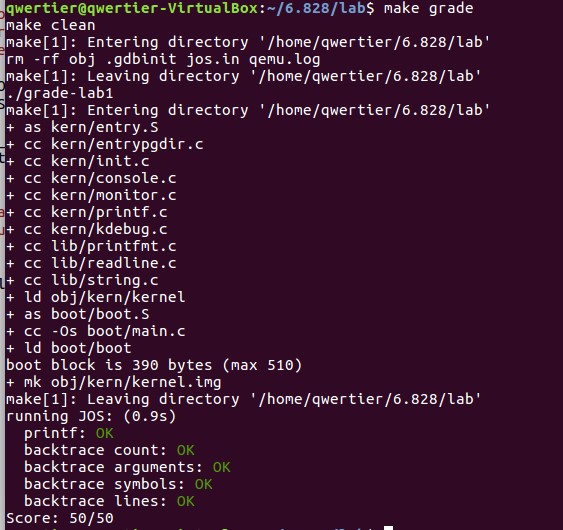


图 3.11 实验 1 完成图

**3.2 Lab 4: Preemptive Multitasking**

**3.2.1** 实验简介

在本实验中，你将会实现抢占式多进程管理。需要哈工大操作系统课第五章的知识。

在第一部分里，需要向 JOS 添加多核支持，实现转轮调度，并添加基础的系统调用（用于添加或删除运行上下文环境，申请或对应内存）。

在第二部分里，需要实现 Unix 风格的 fork() 函数，支持从用户态环境建立一个自己的拷贝。

最后在第三部分里，需要添加对进程间通信 (IPC) 的支持，允许不同的用户态环境减间的交流并同步。还需要添加对硬件时钟中断和抢占的支持。

**3.2.2** 实验目的

* 了解并实现抢占式多进程管理
* 了解并实现转轮法调度算法

**3.2.3** 实验内容

**3.2.3.1** 准备工作

需要用 git commit 把 Lab3 的成果提交，然后在 lab3 分支上新建分支 lab4，并合并远程分支 lab4。

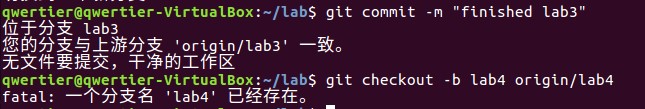


图 3.12 使用 git 的截图

## **3.2.3.2** 第一部分**:** 多处理器支持[[6]](#footnote-6)及协作式多任务[[7]](#footnote-7)

协作式多进程（Cooperative）指的是：进程主动（定期或者空闲时候）让出 CPU 的执行时间给其他进程来执行，这种模式非常依赖于程序的设计，如果程序写的很差，进程长时间占用 CPU 的执行时间（大量计算密集型的操作或者等待外设等），这会让其他进程没有机会执行，从而造成整个操作系统失去响应。

协作式多任务在早期的操作系统中使用得比较广泛（比如 Windows 9x、Classic

Mac OS 等等）。

我们将使 JOS 支持“对称多处理”（SMP），这是一种多处理器模型，其中所有的 CPU 都可以访问系统资源，例如内存和 I / O 总线。虽然所有的 CPU 在功能上与 SMP 完全相同，但在引导过程中，它们可以分为两种类型：引导处理器（BSP）负责初始化系统和引导操作系统; 并且只有在操作系统启动并运行之后，应用处理器（AP）才由 BSP 激活。BSP 的处理器是由硬件和 BIOS 决定的。到目前为止，所有现有的 JOS 代码已经在 BSP 上运行。

在 SMP 系统中，每个 CPU 都有一个附带的本地 APIC（LAPIC）单元。LAPIC 单元负责在整个系统中提供中断。LAPIC 还为连接的 CPU 提供唯一的标识符。在这个实验中，我们利用了 LAPIC 单元的下列基本功能（在 kern / lapic.c 中）：

* 读取 LAPIC 标识符（APIC ID），以告知我们的代码当前正在哪个 CPU 上运行（请参阅 cpunum()）。
* 发送 STARTUP 从 BSP 到的 AP 间中断（IPI）带来的其他 CPU。
* 在 C 部分，我们编程 LAPIC 的内置定时器来触发时钟中断，以支持抢先式多任务处理（请参阅参考资料 apic\_init()）。

**Ex. 1 —** 实现 kern/pmap.c 文件中的 mmio\_map\_region 函数。想要看它是如何被用到的，需要看 kern/lapic.c 中的 lapic\_init。 **void** \* mmio\_map\_region(physaddr\_t pa, **size\_t** size) {

*// Where to start the next region. Initially, this is the // beginning of the MMIO region. Because this is static, its // (just like nextfree in boot\_alloc).* **static uintptr\_t** base = MMIOBASE;

*// Reserve size bytes of virtual memory starting at base and*

*// map physical pages [pa,pa+size) to virtual addresses*

*// [base,base+size). Since this is device memory and not*

*// regular DRAM, you'll have to tell the CPU that it isn't*

*// safe to cache access to this memory. Luckily, the page*

*// tables provide bits for this purpose; simply create the*

*// mapping with PTE\_PCD|PTE\_PWT (cache-disable and*

*// write-through) in addition to PTE\_W. (If you're interested*

*// in more details on this, see section 10.5 of IA32 volume*

*// 3A.)*

*//*

*// Be sure to round size up to a multiple of PGSIZE and to*

*// handle if this reservation would overflow MMIOLIM (it's // okay to simply panic if this happens).*

*//*

*// Hint: The staff solution uses boot\_map\_region.*

*//*

*// Your code here:* **int** nextbase = ROUNDUP(base + size, PGSIZE); **if** (nextbase >= MMIOLIM) { panic("nextbase >= MMIOLIM");

}

boot\_map\_region(kern\_pgdir, base, size, pa, PTE\_PCD | PTE\_PWT | PTE\_W);

**void** \*ret = (**void** \*)base; base = nextbase;

**return** ret;

*// panic("mmio\_map\_region not implemented");*

}

在启动 AP 之前，BSP 应首先收集有关多处理器系统的信息，例如 CPU 的总数，APIC ID 和 LAPIC 单元的 MMIO 地址。kern / mpconfig.c 中的 mp\_init() 函数通过读取驻留在 BIOS 的内存区域中的 MP 配置表来检索此信息。

**Ex. 2 —** 阅读 kern/init.c 中的 boot\_aps() 和 mp\_main() 函数，以及 kern/mpentry.S 文件。确保你理解了 AP 启动时的控制流。修改你在 kern/pmap.c 中的 page\_init() 的实现来防止添加 MPENTRY\_PADDR 位置的页到 free list，以让我们可以安全的拷贝并在那个物理地址执行 AP 运行代码。你的代码应该通过 check\_page\_free\_list() 测试，但是可能通不过 check\_kern\_pgdir() 测试，这个我们稍后会修正。

*// Setup code for APs* **void** mp\_main(**void**) {

*// We are in high EIP now, safe to switch to kern\_pgdir* lcr3(PADDR(kern\_pgdir)); cprintf("SMP: CPU %d starting**\n**", cpunum());

lapic\_init(); env\_init\_percpu(); trap\_init\_percpu(); xchg(&thiscpu->cpu\_status, CPU\_STARTED); *// tell boot\_aps() we're up*

*// Now that we have finished some basic setup, call sched\_yield()*

*// to start running processes on this CPU. But make sure that // only one CPU can enter the scheduler at a time!*

*//*

*// Your code here:*

lock\_kernel(); *//* 锁定内核 sched\_yield(); *//* 调用轮换调度算法

*// Remove this after you finish Exercise 4*

*// for (;;);*

}

编写多处理器操作系统时，区分 CPU 私有状态[[8]](#footnote-8)以及全局状态[[9]](#footnote-9)。需要了解如

下部分：

•CPU 独立内核栈[[10]](#footnote-10)

•Per-CPU TSS(Task State Segment) 和 TSS 描述符

•Per-CPU 当前环境指针

•Per-CPU 系统寄存器

**Ex. 3 —** 修改 kern/pmap.c 中的 mem\_init\_mp() 函数，来映射在 KSTACKTOP 位置的 CPU 独立栈。每个栈大小是 KSTKSIZE 字节加上 KSTKGAP 字节。代码需要通过 check\_kern\_pgdir() 测试。

**void** mem\_init\_mp(**void**) {

*// Map per-CPU stacks starting at KSTACKTOP, for up to 'NCPU' CPUs.*

*//*

*// For CPU i, use the physical memory that 'percpu\_kstacks[i]' refers*

*// to as its kernel stack. CPU i's kernel stack grows down from virtual*

*// address kstacktop\_i = KSTACKTOP - i \* (KSTKSIZE + KSTKGAP), and is*

*// divided into two pieces, just like the single stack you set up in // mem\_init:*

*// \* [kstacktop\_i - KSTKSIZE, kstacktop\_i)*

*// -- backed by physical memory*

*// \* [kstacktop\_i - (KSTKSIZE + KSTKGAP), kstacktop\_i - KSTKSIZE)*

*// -- not backed; so if the kernel overflows its stack, // it will fault rather than overwrite another CPU's stack.*

*// Known as a "guard page".*

*// Permissions: kernel RW, user NONE*

*//*

*// LAB 4: Your code here:*

**int** i; **for** (i=0; i<NCPU; i++) { boot\_map\_region(kern\_pgdir,

KSTACKTOP - i \* (KSTKSIZE + KSTKGAP) - KSTKSIZE, *//* 起始地址

KSTKSIZE, *//* 内存块大小 PADDR(percpu\_kstacks[i]),

PTE\_W|PTE\_P);

}

}

**Ex. 4 —** kern/trap.c 文件中 trap\_init\_percpu() 的代码初始化了 TSS 和 TSS 描述符。

在 Lab3 中可以正常工作，但是在其他 CPU 上运行时就会出问题。修改这些代码已让它能运行在所有 CPU 上。 **void** trap\_init\_percpu(**void**) {

*// The example code here sets up the Task State Segment (TSS) and*

*// the TSS descriptor for CPU 0. But it is incorrect if we are // running on other CPUs because each CPU has its own kernel stack.*

*// Fix the code so that it works for all CPUs.*

*//*

*// Hints:*

|  |  |
| --- | --- |
| *//* | *- The macro "thiscpu" always refers to the current CPU's* |
| *//* | *struct CpuInfo;* |
| *//* | *- The ID of the current CPU is given by cpunum() or* |
| *//* | *thiscpu->cpu\_id;* |
| *//* | *- Use "thiscpu->cpu\_ts" as the TSS for the current CPU,* |
| *//* | *rather than the global "ts" variable;* |
| *//* | *- Use gdt[(GD\_TSS0 >> 3) + i] for CPU i's TSS descriptor;* |
| *//* | *- You mapped the per-CPU kernel stacks in mem\_init\_mp()* |

*//*

*// ltr sets a 'busy' flag in the TSS selector, so if you*

*// accidentally load the same TSS on more than one CPU, you'll*

*// get a triple fault. If you set up an individual CPU's TSS // wrong, you may not get a fault until you try to return from // user space on that CPU.*

*//*

*// LAB 4: Your code here:*

*// Setup a TSS so that we get the right stack // when we trap to the kernel.* thiscpu->cpu\_ts.ts\_esp0 = KSTACKTOP - cpunum() \* (KSTKSIZE + KSTKGAP); thiscpu->cpu\_ts.ts\_ss0 = GD\_KD;

*// Initialize the TSS slot of the gdt.*

gdt[(GD\_TSS0 >> 3) + cpunum()] = SEG16(STS\_T32A,

(**uint32\_t**) &thiscpu->cpu\_ts, **sizeof**(**struct** Taskstate) - 1, 0);

gdt[(GD\_TSS0 >> 3) + cpunum()].sd\_s = 0;

*// Load the TSS selector (like other segment selectors, the*

*// bottom three bits are special; we leave them 0)*

*//ltr(GD\_TSS0 + cpunum()\*sizeof(struct Segdesc));* ltr(((GD\_TSS0 >> 3) + cpunum()) << 3);

*// Load the IDT* lidt(&idt\_pd);

}

如图3.13，四个 CPU 都正常初始化。

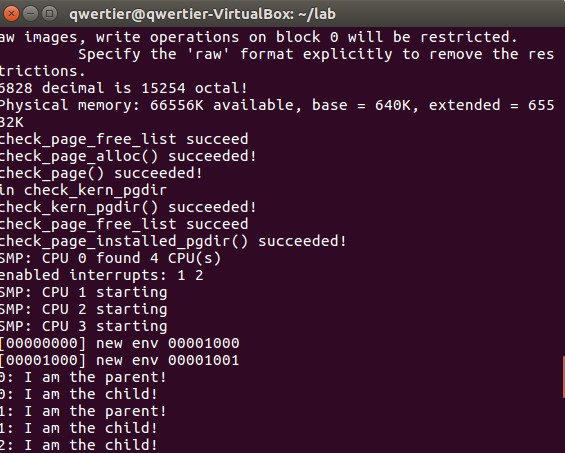


图 3.13 QEMU 启动截图

我们当前的代码在 mp\_main() 函数中初始化 AP 之后就不会再运行了。在让 AP 继续运行前，我们首先需要解决竞争问题当多个 CPU 运行同一个内核时。最简单的方式是使用一个大内核锁。这个大内核锁是一个全局锁会作用于当进程进入内核模式时，而且在返回用户模式时会被释放。在这个模型下，用户模式下的进程可以在当前运行在任何可行的 CPU 上，但是不能有任何一个进程运行在内核模式中。

应该在四个位置应用大内核锁：

•在 i386\_init()BSP 唤醒其他 CPU 之前获取锁。

•在 mp\_main() 初始化 AP 后获取锁，然后调用 sched\_yield() 该 AP 启动运行环境。

•在 trap() 从用户模式中获取锁时。要确定陷阱是发生在用户模式还是在内核模式下，请检查低位 tf\_cs。

•在切换到用户模式之前 env\_run()，立即释放锁定。不要太早或太晚，否则会遇到冲突。

**Ex. 5 —** 如上所述，应用大内核锁，通过调用 lock\_kernel() 并 unlock\_kernel() 在适当的位置。

只需要在 i386\_init(), mp\_main(), trap() 这几个函数中添加 lock\_kernel()，在 env\_run() 中添加 unlock\_kernel() 即可。

因为前三个函数都是进入内核态，而最后的 env\_run() 是从内核态返回用户态。

**Ex. 6 —** 在/kern/sched.c文件的sched\_yield()中实现轮转调度，不要忘记修改syscall() 发送 sys\_yield()。确保在 mp\_main() 中调用 sched\_yield()。并修改 kern/inic.c 来创建所有运行程序 user/yield.c。

JOS 的轮转调度算法如下：

1.函数 sched\_yield() 在新的 kern/sched.c 中，负责选择一个进程来运行。envs[] 以循环的方式按顺序搜索数组，在刚才运行的环境之后（或者在数组的开始处，如果没有以前运行的环境）开始，选择状态为 ENV\_RUNNABLE（见 inc/ env.h）的进程，并调用 env\_run() 进入该进程。

2.sched\_yield() 决不能同时在两个 CPU 上运行相同的进程。它可以告诉一个环境当前正在某个 CPU（可能是当前的 CPU）上运行，因为该进程的状态将会

是 ENV\_RUNNING。

3.已经实现了一个新的系统调用 sys\_yield()，进程可以调用内核的 sched\_yield() 功能，从而自动放弃 CPU 到不同的进程。修改 kern/sched.c 的 sched\_yield() 函数：

**void**

sched\_yield(**void**)

{ **struct** Env \*idle;

*// Implement simple round-robin scheduling.*

*//*

*// Search through 'envs' for an ENV\_RUNNABLE environment in // circular fashion starting just after the env this CPU was*

*// last running. Switch to the first such environment found.*

*//*

*// If no envs are runnable, but the environment previously*

*// running on this CPU is still ENV\_RUNNING, it's okay to // choose that environment.*

*//*

*// Never choose an environment that's currently running on*

*// another CPU (env\_status == ENV\_RUNNING). If there are // no runnable environments, simply drop through to the code // below to halt the cpu. // LAB 4: Your code here.*

**uint32\_t** envid = curenv? ENVX(thiscpu->cpu\_env->env\_id): -1; **uint32\_t** first\_eid = (++envid) % NENV; **uint32\_t** next\_envid; **int** i;

*// case:* 进程可运行 **for** (i = 0; i < NENV; i++) {

next\_envid = (first\_eid+i) % NENV; **if** (envs[next\_envid].env\_status == ENV\_RUNNABLE) {

env\_run(&envs[next\_envid]); **break**;

}

}

*// case:* 进程在运行

**if** (curenv && curenv->env\_status == ENV\_RUNNING) {

env\_run(curenv);

}

*// sched\_halt never returns* sched\_halt();

}

修改 kern/inic 文件的 i386\_init() 函数：

**void** i386\_init(**void**) { **extern char** edata[], end[];

*// Before doing anything else, complete the ELF loading process.*

*// Clear the uninitialized global data (BSS) section of our program.*

*// This ensures that all static/global variables start out zero.* memset(edata, 0, end - edata);

*// Initialize the console.*

*// Can't call cprintf until after we do this!* cons\_init();

cprintf("6828 decimal is %o octal!**\n**", 6828);

*// Lab 2 memory management initialization functions* mem\_init();

*// Lab 3 user environment initialization functions* env\_init(); trap\_init();

*// Lab 4 multiprocessor initialization functions* mp\_init(); lapic\_init();

*// Lab 4 multitasking initialization functions* pic\_init();

*// Acquire the big kernel lock before waking up APs // Your code here:* lock\_kernel();

*// Starting non-boot CPUs* boot\_aps();

*#if defined(TEST)*

*// Don't touch -- used by grading script!* ENV\_CREATE(TEST, ENV\_TYPE\_USER);

*#else*

*// Touch all you want.*

*//ENV\_CREATE(user\_primes, ENV\_TYPE\_USER);*

ENV\_CREATE(user\_dumbfork, ENV\_TYPE\_USER);

*#endif // TEST\**

*// Schedule and run the first user environment!*

sched\_yield();

}

尽管内核现在可以在多个用户级环境之间运行和切换，但仍然局限于运行内核最初设置的环境。现在将实现必要的 JOS 系统调用，以允许用户环境创建并启动其他新用户环境。

Unix 提供 fork() 系统调用作为其进程创建原语。Unix fork() 复制调用进程（父进程）的整个地址空间来创建一个新进程（子进程）。用户空间中两个观察值之间唯一的区别是它们的进程 ID 和父进程 ID（由 getpid 和返回 getppid）。在父 fork() 进程中，返回子进程 ID，而在子进程中，fork() 返回 0. 默认情况下，每个进程都有自己的私有地址空间，进程对内存的修改对另一个进程是不可见的。

现在需要为创建新的用户模式环境提供一组不同的 JOS 系统调用。有了这些系统调用 fork()，除了其他类型的环境创建之外，您将能够完全在用户空间中实现类 Unix 。新的系统调用你将为 JOS 写入如下：

1.sys\_exofork：这个系统调用创建了一个几乎空白的新环境：在它的地址空间的用户部分没有任何东西被映射，它不能运行。新环境在 sys\_exofork 调用时将具有与父环境相同的注册状态。在父代中，sys\_exofork 将返回 envid\_t 新创建的环境（如果环境分配失败，则返回负面的错误代码）。然而，在小孩，它将返回 0.（由于孩子开始标记为不可运行，sys\_exofork 实际上不会返回到孩子，直到父母明确允许这一点，通过标记可运行的孩子....）

2.sys\_env\_set\_status：将指定环境的状态设置为ENV\_RUNNABLE或ENV\_NOT\_RUNNABLE。

一旦地址空间和寄存器状态完全初始化，这个系统调用通常用于标记一个准备运行的新环境。

3.sys\_page\_alloc：分配一页物理内存并将其映射到给定环境的地址空间中的给定虚拟地址。

4.sys\_page\_map：将页面映射（不是页面内容！）从一个环境的地址空间复制到另一个环境的地址空间，留下一个内存共享安排，以便新映射和旧映射都指向同一页的物理内存。

5.sys\_page\_unmap：取消映射在给定环境中给定虚拟地址映射的页面。

**Ex. 7 —** 在 kern / syscall.c 中实现上述系统调用，并确保 syscall（）调用它们。将需要在 kern/pmap.c 和 kern/env.c 中使用各种函数，特别是 envid2env()。现在，只要你调用 envid2env()，在 checkperm 参数中传递 1。确保你检查任何无效的系统调用参数，-E\_INVAL 在这种情况下返回。用 user/dumbfork 测试你的 JOS 内核，并确保它的工作。

*// Allocate a new environment.*

*// Returns envid of new environment, or < 0 on error. Errors are:*

*//^^I-E\_NO\_FREE\_ENV if no free environment is available.*

*//^^I-E\_NO\_MEM on memory exhaustion.*

**static** envid\_t sys\_exofork(**void**) {

*// Create the new environment with env\_alloc(), from kern/env.c.*

*// It should be left as env\_alloc created it, except that*

*// status is set to ENV\_NOT\_RUNNABLE, and the register set is copied*

*// from the current environment -- but tweaked so sys\_exofork // will appear to return 0.*

*// LAB 4: Your code here.*

**struct** Env \*env; **int** rtn = env\_alloc(&env, curenv->env\_id); **if** (rtn < 0) **return** rtn;

env->env\_status = ENV\_NOT\_RUNNABLE; memmove(&env->env\_tf, &curenv->env\_tf, **sizeof**(**struct** Trapframe)); env->env\_tf.tf\_regs.reg\_eax = 0;

**return** env->env\_id;

}

*// Set envid's env\_status to status, which must be ENV\_RUNNABLE // or ENV\_NOT\_RUNNABLE.*

*//*

*// Returns 0 on success, < 0 on error. Errors are:*

*//^^I-E\_BAD\_ENV if environment envid doesn't currently exist, //^^I^^Ior the caller doesn't have permission to change envid. //^^I-E\_INVAL if status is not a valid status for an environment.* **static int** sys\_env\_set\_status(envid\_t envid, **int** status) {

*// Hint: Use the 'envid2env' function from kern/env.c to translate an // envid to a struct Env.*

*// You should set envid2env's third argument to 1, which will*

*// check whether the current environment has permission to set*

*// envid's status.*

*// LAB 4: Your code here.*

**struct** Env \*env; **int** rtn;

rtn = envid2env(envid, &env, 1); **if** (rtn < 0) {

cprintf("sys\_env\_set\_status: envid2env**\n**"); **return** rtn;

}

**if** (status != ENV\_RUNNABLE && status != ENV\_NOT\_RUNNABLE) { cprintf("sys\_env\_set\_status: invalid status**\n**"); **return** -E\_INVAL;

}

env->env\_status = status; **return** 0;

*//panic("sys\_env\_set\_status not implemented");*

}

*// Allocate a page of memory and map it at 'va' with permission // 'perm' in the address space of 'envid'.*

*// The page's contents are set to 0.*

*// If a page is already mapped at 'va', that page is unmapped as a // side effect.*

*//*

*// perm -- PTE\_U | PTE\_P must be set, PTE\_AVAIL | PTE\_W may or may not be set, // but no other bits may be set. See PTE\_SYSCALL in inc/mmu.h.*

*//*

*// Return 0 on success, < 0 on error. Errors are:*

*//^^I-E\_BAD\_ENV if environment envid doesn't currently exist, //^^I^^Ior the caller doesn't have permission to change envid.*

*//^^I-E\_INVAL if va >= UTOP, or va is not page-aligned.*

*//^^I-E\_INVAL if perm is inappropriate (see above). //^^I-E\_NO\_MEM if there's no memory to allocate the new page, //^^I^^Ior to allocate any necessary page tables.*

**static int**

sys\_page\_alloc(envid\_t envid, **void** \*va, **int** perm) {

*// Hint: This function is a wrapper around page\_alloc() and // page\_insert() from kern/pmap.c.*

*// Most of the new code you write should be to check the // parameters for correctness.*

*// If page\_insert() fails, remember to free the page you // allocated!*

*// LAB 4: Your code here.*

**struct** Env \*env; **int** rtn;

rtn = envid2env(envid, &env, 1); **if** (rtn < 0) {

cprintf("sys\_page\_alloc: envid2env**\n**"); **return** rtn;

}

**if** ((**uint32\_t**)va >= UTOP || (**uint32\_t**)va % PGSIZE != 0) { cprintf("sys\_page\_alloc: invalid boundary**\n**"); **return** -E\_INVAL;

}

**if** ((perm & (PTE\_U | PTE\_P)) != (PTE\_U | PTE\_P)) { cprintf("sys\_page\_alloc: PTE\_U | PTE\_P are not set**\n**"); **return** -E\_INVAL;

}

**if** ((perm & ~PTE\_SYSCALL) != 0) { cprintf("sys\_page\_alloc: invalid perm**\n**"); **return** -E\_INVAL;

}

**struct** PageInfo \*pp = page\_alloc(ALLOC\_ZERO); **if** (!pp) {

cprintf("sys\_page\_alloc: page\_alloc**\n**"); **return** -E\_NO\_MEM;

}

rtn = page\_insert(env->env\_pgdir, pp, va, perm);

**if** (rtn < 0) {

page\_free(pp); cprintf("sys\_page\_alloc: page\_insert**\n**"); **return** rtn;

}

**return** 0;

*//panic("sys\_page\_alloc not implemented");*

}

*// Map the page of memory at 'srcva' in srcenvid's address space*

*// at 'dstva' in dstenvid's address space with permission 'perm'.*

*// Perm has the same restrictions as in sys\_page\_alloc, except*

*// that it also must not grant write access to a read-only // page.*

*//*

*// Return 0 on success, < 0 on error. Errors are:*

*//^^I-E\_BAD\_ENV if srcenvid and/or dstenvid doesn't currently exist, //^^I^^Ior the caller doesn't have permission to change one of them.*

*//^^I-E\_INVAL if srcva >= UTOP or srcva is not page-aligned, //^^I^^Ior dstva >= UTOP or dstva is not page-aligned.*

*//^^I-E\_INVAL is srcva is not mapped in srcenvid's address space.*

*//^^I-E\_INVAL if perm is inappropriate (see sys\_page\_alloc). //^^I-E\_INVAL if (perm & PTE\_W), but srcva is read-only in srcenvid's //^^I^^Iaddress space.*

*//^^I-E\_NO\_MEM if there's no memory to allocate any necessary page tables.*

**static int** sys\_page\_map(envid\_t srcenvid, **void** \*srcva,

envid\_t dstenvid, **void** \*dstva, **int** perm) {

*// Hint: This function is a wrapper around page\_lookup() and // page\_insert() from kern/pmap.c.*

*// Again, most of the new code you write should be to check the // parameters for correctness.*

*// Use the third argument to page\_lookup() to // check the current permissions on the page.*

*// LAB 4: Your code here.* **struct** Env \*se, \*de;

**if** (envid2env(srcenvid, &se, 1)

|| envid2env(dstenvid, &de, 1)) { cprintf("sys\_page\_map: E\_BAD\_ENV**\n**"); **return** -E\_BAD\_ENV;

}

**if** ((**uint32\_t**)srcva >= UTOP

|| (**uint32\_t**)dstva >= UTOP

|| (**uint32\_t**)srcva % PGSIZE != 0

|| (**uint32\_t**)dstva % PGSIZE != 0) { cprintf("sys\_page\_map: invalid boundary or va size, %d, %d**\n**", (**uint32\_t**)srcva, (**uint return** -E\_INVAL;

}

**if** ((perm & (PTE\_U | PTE\_P)) != (PTE\_U | PTE\_P)) { cprintf("sys\_page\_map: PTE\_U | PTE\_P are not set**\n**"); **return** -E\_INVAL;

}

*/\* if ((perm & ~PTE\_SYSCALL) != 0) { cprintf("sys\_page\_map: invalid perm\n"); return -E\_INVAL;*

*} \*/*

pte\_t \*pte; **struct** PageInfo \*pp = page\_lookup(se->env\_pgdir, srcva, &pte); **if** (!pp) { cprintf("sys\_page\_map: map not found**\n**"); **return** -E\_INVAL;

}

**if** ((perm & PTE\_W) && !(\*pte & PTE\_W)) { cprintf("sys\_page\_map: invalid PTE\_W**\n**"); **return** -E\_INVAL;

}

**int** rtn = page\_insert(de->env\_pgdir, pp, dstva, perm); **if** (rtn < 0) {

cprintf("sys\_page\_map: page\_insert**\n**");

**return** rtn;

}

**return** 0;

*// panic("sys\_page\_map not implemented");*

}

*// Unmap the page of memory at 'va' in the address space of 'envid'.*

*// If no page is mapped, the function silently succeeds.*

*//*

*// Return 0 on success, < 0 on error. Errors are:*

*//^^I-E\_BAD\_ENV if environment envid doesn't currently exist, //^^I^^Ior the caller doesn't have permission to change envid.*

*//^^I-E\_INVAL if va >= UTOP, or va is not page-aligned.* **static int** sys\_page\_unmap(envid\_t envid, **void** \*va)

{

*// Hint: This function is a wrapper around page\_remove().*

*// LAB 4: Your code here.*

**struct** Env \*env; **int** rtn;

rtn = envid2env(envid, &env, 1); **if** (rtn < 0) {

cprintf("sys\_page\_unmap: envid2env**\n**"); **return** rtn;

}

**if** ((**uint32\_t**)va >= UTOP || (**uint32\_t**)va % PGSIZE != 0) { cprintf("sys\_page\_unmap: invalid boundary**\n**"); **return** -E\_INVAL;

}

page\_remove(env->env\_pgdir, va);

**return** 0;

*// panic("sys\_page\_unmap not implemented");*

}

## **3.2.3.3** 第二部分**:** 写时拷贝[[11]](#footnote-11)

如前所述，Unix 提供 fork() 系统调用作为其主要进程创建原语。该 fork() 系统

调用将调用进程的地址空间（父）创建一个新的进程（孩子）。

xv6 Unix fork() 通过将父页面中的所有数据复制到为子项分配的新页面中来实现。这基本上是一样的方法 dumbfork()。父母的地址空间复制到孩子是操作中最耗时的部分 fork()。

为了处理自己的页面错误，用户环境将需要用 JOS 内核注册页面错误处理程序入口点。用户环境通过新的 sys\_env\_set\_pgfault\_upcall 系统调用注册页面错误入口点。我们在 Env 结构中增加了一个新成员 env\_pgfault\_upcall 来记录这个信息。

**Ex. 8 —** 实现 sys\_env\_set\_pgfault\_upcall 系统调用。

*// Set the page fault upcall for 'envid' by modifying the corresponding struct*

*// Env's 'env\_pgfault\_upcall' field. When 'envid' causes a page fault, the*

*// kernel will push a fault record onto the exception stack, then branch to // 'func'.*

*//*

*// Returns 0 on success, < 0 on error. Errors are:*

*//^^I-E\_BAD\_ENV if environment envid doesn't currently exist, //^^I^^Ior the caller doesn't have permission to change envid.* **static int** sys\_env\_set\_pgfault\_upcall(envid\_t envid, **void** \*func)

{

*// LAB 4: Your code here.*

**struct** Env \*env; **int** rtn;

rtn = envid2env(envid, &env, 1); **if** (rtn < 0) {

cprintf("sys\_env\_set\_status: envid2env**\n**"); **return** rtn;

}

env->env\_pgfault\_upcall = func; **return** 0;

*// panic("sys\_env\_set\_pgfault\_upcall not implemented");*

}

在正常执行期间，JOS 中的用户环境将运行在普通用户堆栈上：它的 ESP 寄存器开始指向USTACKTOP，并且它所推送的堆栈数据驻留在页面之间USTACKTOPPGSIZE（USTACKTOP-1 包含）。然而，当在用户模式下发生页面错误时，内核将重新启动在不同堆栈上运行指定的用户级页面错误处理程序的用户环境，即用户异常堆栈。从本质上讲，我们将使 JOS 内核代表用户环境自动执行“堆栈切换”，就好像 x86 处理器在从用户模式向内核模式转换时已经代表 JOS 执行堆栈切换一样。

现在需要更改 kern / trap.c 中的页面错误处理代码，以处理来自用户模式的页面错误，如下所示。我们将会在用户模式下的 trap-time 产生故障时调用。

如果没有注册页面错误处理程序，那么 JOS 内核像以前那样用消息破坏用户环境。否则，内核设置异常堆栈上的陷阱框架，看起来像一个 struct UTrapframe（见 inc/ trap.h）：

<– UXSTACKTOP trap-time esp

trap-time eflags trap-time eip

trap-time eax start of struct PushRegs trap-time ecx trap-time edx trap-time ebx trap-time esp trap-time ebp

trap-time esi

trap-time edi end of struct PushRegs tf\_err (error code) fault\_va <– %esp when handler is run

然后内核安排用户环境继续执行，在这个堆栈帧上运行异常堆栈上的页面错误处理程序; 你必须弄清楚如何做到这一点。该 fault\_va 是导致页面错误的虚拟地址。

如果用户环境在发生异常时已经在用户异常堆栈上运行，则页面错误处理程序本身出错。在这种情况下，你应该开始新的堆栈框架，tf->tf\_esp 而不是在当前的 UXSTACKTOP。你应该先推一个空的 32 位字，然后再是 struct UTrapframe。

为了测试是否 tf->tf\_esp 已经是用户异常堆栈上，检查它是否在之间的范围

UXSTACKTOP-PGSIZE 和 UXSTACKTOP-1。

**Ex. 9 —** 实现 kern/trap.c 中的 page\_fault\_handler，来处理用户模式的分页错误。写入异常堆栈时一定要采取适当的预防措施。（如果用户环境在异常堆栈上空间不足，会发生什么？）

**void** page\_fault\_handler(**struct** Trapframe \*tf)

{

*//cprintf("in the page\_fault\_handler\n");* **uint32\_t** fault\_va;

*// Read processor's CR2 register to find the faulting address* fault\_va = rcr2();

*// Handle kernel-mode page faults.*

*// LAB 3: Your code here.* **if** ((tf->tf\_cs & 0x03) == 0) {

print\_trapframe(tf);

panic("kernel page fault"); }

*// We've already handled kernel-mode exceptions, so if we get here, // the page fault happened in user mode.*

*// Call the environment's page fault upcall, if one exists. Set up a*

*// page fault stack frame on the user exception stack (below // UXSTACKTOP), then branch to curenv->env\_pgfault\_upcall.*

*//*

*// The page fault upcall might cause another page fault, in which case*

*// we branch to the page fault upcall recursively, pushing another // page fault stack frame on top of the user exception stack.*

*//*

*// The trap handler needs one word of scratch space at the top of the*

*// trap-time stack in order to return. In the non-recursive case, we*

*// don't have to worry about this because the top of the regular user*

*// stack is free. In the recursive case, this means we have to leave*

*// an extra word between the current top of the exception stack and // the new stack frame because the exception stack \_is\_ the trap-time // stack.*

*//*

*// If there's no page fault upcall, the environment didn't allocate a // page for its exception stack or can't write to it, or the exception*

*// stack overflows, then destroy the environment that caused the fault.*

*// Note that the grade script assumes you will first check for the page*

*// fault upcall and print the "user fault va" message below if there is*

*// none. The remaining three checks can be combined into a single test.*

*//*

*// Hints:*

*// user\_mem\_assert() and env\_run() are useful here.*

*// To change what the user environment runs, modify 'curenv->env\_tf' // (the 'tf' variable points at 'curenv->env\_tf').*

*// LAB 4: Your code here.*

**do** {

**if** (!curenv->env\_pgfault\_upcall) { cprintf("[%08x] not set env\_pgfault\_upcall**\n**", curenv->env\_id); **break**; }

**if** (tf->tf\_esp > USTACKTOP && tf->tf\_esp < UXSTACKTOP - PGSIZE) { cprintf("[%08x] exception stack overflow**\n**", curenv->env\_id); **break**; }

**struct** UTrapframe utf; utf.utf\_fault\_va = fault\_va; utf.utf\_err = tf->tf\_err; utf.utf\_regs = tf->tf\_regs; utf.utf\_eip = tf->tf\_eip; utf.utf\_eflags = tf->tf\_eflags; utf.utf\_esp = tf->tf\_esp;

**void** \*dststack;

**if** (tf->tf\_esp < UXSTACKTOP && tf->tf\_esp >= UXSTACKTOP - PGSIZE) { dststack = (**void** \*) (tf->tf\_esp - **sizeof**(**struct** UTrapframe) - 4);

} **else** { dststack = (**void** \*) (UXSTACKTOP - **sizeof**(**struct** UTrapframe));

}

user\_mem\_assert(curenv, dststack, 0, PTE\_U | PTE\_W | PTE\_P); memmove(dststack, &utf, **sizeof**(**struct** UTrapframe)); tf->tf\_eip = (**uintptr\_t**) curenv->env\_pgfault\_upcall; tf->tf\_esp = (**uintptr\_t**) dststack;

env\_run(curenv);

} **while**(0);

*// Destroy the environment that caused the fault.*

cprintf("[%08x] user fault va %08x ip %08x**\n**", curenv->env\_id, fault\_va, tf->tf\_eip); print\_trapframe(tf); env\_destroy(curenv);

}

接下来，需要实现汇编程序，该程序将负责调用 C 页面错误处理程序，并在原始错误指令处恢复执行。这个汇编程序将使用内核注册的处理程序sys\_env\_set\_pgfault\_upcall()。

**Ex. 10 —** 实现 lib/pfentry.S 中的 \_pgfault\_upcall。有趣的部分是返回到导致页面错误的用户代码中的原始点。你会直接返回，而不必回头检查内核。困难的部分是同时切换堆栈和重新加载 EIP。

*# Restore the trap-time registers. After you do this, you # can no longer modify any general-purpose registers.*

*# LAB 4: Your code here.*

movl 48(%esp), %eax subl $4, %eax movl %eax, 48(%esp)

movl 40(%esp), %ebx movl %ebx, (%eax)

addl $8, %esp popal addl $4, %esp

*# Restore eflags from the stack. After you do this, you can*

*# no longer use arithmetic operations or anything else that # modifies eflags. # LAB 4: Your code here.*

popf

*# Switch back to the adjusted trap-time stack.*

*# LAB 4: Your code here.*

movl (%esp), %esp *#popl %esp*

*# Return to re-execute the instruction that faulted.*

*# LAB 4: Your code here.* ret

**Ex. 11 —** 完成 lib/pgfault.c 中的 set\_pgfault\_handler()。

**void** set\_pgfault\_handler(**void** (\*handler)(**struct** UTrapframe \*utf))

{ **int** r;

**if** (\_pgfault\_handler == 0) {

*// First time through!*

*// LAB 4: Your code here.*

**if** ((r = sys\_page\_alloc(0, (**void** \*)ROUNDDOWN(UXSTACKTOP - 1, PGSIZE), PTE\_P|PTE\_U panic("set\_pgfault\_handler error: %e", r);

}

**if** ((r = sys\_env\_set\_pgfault\_upcall(0, \_pgfault\_upcall)) < 0) { panic("sys\_env\_set\_pgfault\_upcall error: %e", r);

}

*//panic("set\_pgfault\_handler not implemented");*

}

*// Save handler pointer for assembly to call.*

\_pgfault\_handler = handler;

}

我们在在 lib/fork.c 中为你提供了一个框架 fork()。类似于 dumbfork()，fork() 应该创建一个新的进程，然后扫描父进程的整个地址空间，并在子进程中设置相应的页面映射。关键区别在于，dumbfork() 复制页面时，fork() 只会复制页面映射。

fork() 只有当其中一个进程尝试写入时，才会复制每个页面。 fork() 的大致过程如下：

1.父进程使用pgfault()为C语言层面的错误处理程序，使用上面实现的set\_pgfault\_handler()。

2.父进程调用 sys\_exofork() 来建立子进程。

3.对于父进程的每一个可写的页或者写入时复制的页，父进程会调用 duppage()，来让页面映射到父进程的地址空间中并且重新映射该页面到自己的地址空间中。然而，异常堆栈不是以这种方式重新映射的。相反，您需要在子项中为异常堆栈分配一个新页面。由于页面错误处理程序将执行实际的复制，并且页面错误处理程序在异常堆栈上运行，所以异常堆栈不能写入时复制。fork() 还需要处理当前不能写入或写入时复制的页面。

4.父进程为子进程设置用户页错误的入口（跟自己的一样）。

5.此时子进程可以运行，所以父进程设置子进程为可运行。

每当其中一个进程写入尚未写入的写入时复制页面时，就会出现页面错误。以下是用户页面错误处理程序的控制流程：

1.内核传播页面错误 \_pgfault\_upcall，它调用 fork() 的 pgfault() 处理程序。

2.pgfault() 检查错误是否写入（FEC\_WR 在错误代码中检查），并确认页面的

PTE 已被标记 PTE\_COW。如果不是，就抛出异常。

3.pgfault() 分配映射到临时位置的新页面并将错误页面的内容复制到其中。然后，故障处理程序将新页面映射到适当的地址，具有读取/写入权限，而不是旧的只读映射。

用户级的 lib/fork.c 代码必须参考上面几个操作的环境页表（例如，页面的 PTE 被标记 PTE\_COW）。内核 UVPT 完全为此目的映射环境的页表。它使用了一个巧妙的映射技巧，使得查找用户代码的 PTE 变得容易。lib/entry.S 设置 uvpt，uvpd 以便您可以在 lib/fork.c 中轻松查找页面表信息。

**Ex. 12 —** 在 lib/fork.c 中实现 fork(),duppage() 以及 pgfault() fork() 函数用于建立子进程，代码如下：

envid\_t fork(**void**) { envid\_t envid; **uintptr\_t** addr;

set\_pgfault\_handler(&pgfault); envid = sys\_exofork(); **if**(envid < 0)

panic("fork: sys\_exofork failed"); **if**(envid == 0){ thisenv = &envs[ENVX(sys\_getenvid())]; **return** 0;

}

**for** (addr = 0; addr < USTACKTOP; addr += PGSIZE) {

**if** ((uvpd[PDX(addr)] & PTE\_P) && (uvpt[PGNUM(addr)] & PTE\_P)) duppage(envid, PGNUM(addr));

}

**if** (sys\_page\_alloc(envid, (**void** \*) (UXSTACKTOP - PGSIZE),

PTE\_P | PTE\_U | PTE\_W) < 0)

panic("fork: sys\_page\_alloc failed");

**extern void** \_pgfault\_upcall(); sys\_env\_set\_pgfault\_upcall(envid, \_pgfault\_upcall); **if** (sys\_env\_set\_status(envid, ENV\_RUNNABLE) != 0) panic("fork: sys\_env\_set\_status");

**return** envid;

}

duppage() 函数用于将父进程的写时复制内存映射到子进程的地址空间，代码

如下：

**static int** duppage(envid\_t envid, **unsigned** pn) { **int** r; **void** \*addr = (**void** \*)(pn \* PGSIZE); **if** ((uvpt[pn] & PTE\_W) || (uvpt[pn] & PTE\_COW)) { **if** ((r = sys\_page\_map(0, addr, envid, addr, PTE\_P | PTE\_U | PTE\_COW)) < 0) panic("duppage: %e", r);

**if** ((r = sys\_page\_map(0, addr, 0, addr, PTE\_P | PTE\_U | PTE\_COW)) < 0) panic("duppage: %e", r);

} **else if** ((r = sys\_page\_map(0, addr, envid, addr, PTE\_P | PTE\_U)) < 0) panic("duppage: %e", r);

**return** 0;

}

pgfault() 函数用于处理页面错误，代码如下：

**static void** pgfault(**struct** UTrapframe \*utf) { **void** \*addr = (**void** \*) utf->utf\_fault\_va; **uint32\_t** err = utf->utf\_err; **int** r; pte\_t pte = ((pte\_t \*) uvpt)[PGNUM(addr)]; **if**(!( (err & FEC\_WR) != 0 && (pte & PTE\_COW)!=0 )){ panic("pgfault: not write and not a COW page"); **return**;

} addr = ROUNDDOWN(addr, PGSIZE); envid\_t envid = sys\_getenvid(); **if** ((r = sys\_page\_alloc(envid, PFTEMP, PTE\_P | PTE\_U | PTE\_W)) < 0) panic("pgfault: %e", r);

memcpy(PFTEMP, addr, PGSIZE); **if** ((r = sys\_page\_map(envid, PFTEMP, envid, addr, PTE\_P | PTE\_U | PTE\_W)) < 0) panic("pgfault: %e", r);

**if** ((r = sys\_page\_unmap(envid, PFTEMP)) < 0) panic("pgfault: %e", r);

**return**;

}

## **3.2.3.4** 第三部分**:** 抢占式多任务[[12]](#footnote-12)及进程间通信[[13]](#footnote-13)

抢占式多进程指的是操作系统把 CPU 的时间划分成一个一个的时间片（毫秒

级），抢占式多任务保证每个进程都能获得一个 CPU 时间片来执行。

现代操作系统都支持抢占式多任务，包括 Windows、macOS、Linux（包括 Android）和 iOS。外部中断（即硬件中断）被称为 IRQ。有 16 个可能的 IRQ，编号为 0 到 15。

从 IRQ 到 IDT 条目的映射不是固定的。picirq.c 中的 pic\_init 映射 IRQ 0-15 到 IDT 条目 IRQ\_OFFSET 到 IRQ\_OFFSET+15。

在 JOS 中，与 xv6 Unix 相比，我们做了一个关键的简化。在内核中，外部设备中断始终是禁用的（并且像 xv6 一样，在用户空间中启用）。外部中断由寄存器的 FL\_IF 标志位控制%eflags（参见 inc/mmu.h）。当该位置 1 时，外部中断使能。虽然可以通过多种方式对位进行修改，但是由于我们的简化，我们将在%eflags 进入和退出用户模式时单独通过保存和恢复寄存器的过程来处理它。

因此必须确保 FL\_IF 在用户环境中设置该标志，以便在中断到达时将其传递

到处理器，并由中断代码进行处理。

**Ex. 13 —** 修改 kern/trapentry.S 和 kern/trap.c 以初始化所述 IDT 中的相应条目，并为 0 到 15 的 IRQ 设置处理程序。然后修改代码中在 kern/env.c 中的 env\_alloc()，以确保用户环境总是在启用中断的情况下运行。再取消 sched\_halt（）中的 sti 指令的注释，以便空闲的 CPU 解除屏蔽中断。修改 kern/trap.c 文件中的 trap\_init() 函数的此位置，扩充设置中断向量表。

**for** (**int** i = 0; i < 256; i++) {

SETGATE(idt[i], 0, GD\_KT, trap\_handlers[i], 0);

}

完成之后，Jos 就能进行时钟中断了。

**Ex. 14 —** 修改内核的 trap\_dispatch() 函数，以便 sched\_yield() 在发生时钟中断时调用查找和运行不同的环境。

处理时钟中断。防止进程死循环一直霸占 cpu，所以需要在时钟中断处，重新调度进程。

**if** (tf->tf\_trapno == IRQ\_OFFSET + IRQ\_TIMER) { lapic\_eoi();

sched\_yield(); **return**;

}

（技术上 JOS 是“环境间通信”或“IEC”，但其他人都称之为 IPC，所以我们将使用标准术语。）

我们一直专注于操作系统的隔离方面，它提供了每个程序都拥有一台机器的错觉。操作系统的另一个重要的服务是允许程序在他们想要的时候相互通信。让程序与其他程序交互可以是相当强大的。Unix 管道模型是典型的例子。

进程间通信有很多模型。即使在今天，仍然存在关于哪种模型最好的争论。我们不会进入这个辩论。相反，我们将实现一个简单的 IPC 机制。

执行两个系统调用，sys\_ipc\_recv 并且 sys\_ipc\_try\_send。然后实现两个库包装

ipc\_recv 和 ipc\_send。

用户环境可以使用 JOS 的 IPC 机制相互发送的“消息”由两个部分组成：单个 32 位值和可选的单页映射。允许进程在消息中传递页面映射提供了一种传输更多数据的有效方式，而不是将其放入单个 32 位整数中，并且还允许进程轻松地设置共享内存布置。

要接收消息，进程调用 sys\_ipc\_recv。这个系统调用取消了当前进程的调度，并且在收到消息之前不会再运行它。当进程等待接收消息时，任何其他进程都可以向其发送消息，而不仅仅是特定的进程，而不仅仅是具有与接收进程的父/子布置的进程。换句话说，在 A 部分中实现的权限检查不适用于 IPC，因为 IPC 系统调用是经过精心设计的以便“安全”：进程不会仅仅通过发送消息就会导致其他进程发生故障。

要尝试发送一个值，进程将调用 sys\_ipc\_try\_send 接收者的进程 ID 和要发送的值。如果指定的进程实际上正在接收（它已经调用 sys\_ipc\_recv 并且还没有获得值），则发送递送消息并返回 0. 否则发送返回-E\_IPC\_NOT\_RECV 以指示目标进程当前不期望接收值。

ipc\_recv 用户空间中的库函数将负责调用 sys\_ipc\_recv，然后在当前进程中查找有关接收值的信息 struct Env。

同样，一个库函数 ipc\_send 将负责重复调用，sys\_ipc\_try\_send 直到发送成功。

当进程 sys\_ipc\_recv 使用有效 dstva 参数调用（下面 UTOP）时，进程表明它愿意接收页面映射。如果发送者发送一个页面，那么该页面应该被映射 dstva 到接收者的地址空间中。如果接收方已经有了一个映射的页面 dstva，那么这个前一个页面就是未映射的。

当进程 sys\_ipc\_try\_send 使用有效的 srcva（下面 UTOP）调用时，这意味着发送者想要发送当前映射 srcva 到接收者的页面，并有权限 perm。在一个成功的 IPC 之后，发送者保持其在页面的原始映射 srcva 到其地址空间，但是接收者也在 dstva 接收者的地址空间中获得接收者最初指定的相同物理页面的映射。结果这个页面在发送者和接收者之间被共享。

如果发送者或接收者不指示应该传送页面，则不传送页面。在任何 IPC 之后，

内核将 env\_ipc\_perm 接收器 Env 结构中的新字段设置为接收到的页面的权限，如果没有收到页面，则为零。

**Ex. 15 —** 实现 kern/syscall.c 中的 sys\_ipc\_recv 和 sys\_ipc\_try\_send。然后在 lib/ipc.c 中实现 ipc\_recv 和 ipc\_send 函数。 **static int** sys\_ipc\_recv(**void** \*dstva)

{

*// LAB 4: Your code here.* **if** (dstva < (**void** \*)UTOP &&

(**uint32\_t**)dstva % PGSIZE) { *//* 地址错误 **return** -E\_INVAL;

}

curenv->env\_ipc\_dstva = dstva; curenv->env\_ipc\_perm = 0;

*//curenv->env\_tf.tf\_regs.reg\_eax = 0;*

curenv->env\_ipc\_recving = 1; *//* 正在接收信息

curenv->env\_status = ENV\_NOT\_RUNNABLE; *//* 挂起进程直到接收到消息

**return** 0;

} **static int** sys\_ipc\_try\_send(envid\_t envid, **uint32\_t** value, **void** \*srcva, **unsigned** perm) {

*// LAB 4: Your code here.* **int** r; **struct** Env \*env;

**if** ((r = envid2env(envid, &env, 0)) < 0) { **return** r;

}

**if** (!env->env\_ipc\_recving) { **return** -E\_IPC\_NOT\_RECV;

}

**if** ((**uint32\_t**)srcva < UTOP && (**uint32\_t**)srcva % PGSIZE != 0) { *//* 地址设置错误 cprintf("sys\_ipc\_try\_send: invalid boundary**\n**"); **return** -E\_INVAL;

}

**if** ((**uint32\_t**)srcva < UTOP &&

!(perm & PTE\_U) &&

!(perm & PTE\_P) &&

(perm & ~PTE\_SYSCALL)) { cprintf("sys\_ipc\_try\_send: invalid perm**\n**"); **return** -E\_INVAL;

}

**if** ((**uint32\_t**)srcva < UTOP && (**uint32\_t**)env->env\_ipc\_dstva < UTOP) {

**struct** PageInfo \*pp = page\_lookup(curenv->env\_pgdir, srcva, 0); **if** (!pp) { **return** -E\_INVAL;

}

r = page\_insert(env->env\_pgdir, pp, env->env\_ipc\_dstva, perm);

**if** (r < 0) {

cprintf("sys\_ipc\_try\_send: page\_insert %d %e**\n**", r, r);

}

env->env\_ipc\_perm = perm;

}

env->env\_ipc\_from = curenv->env\_id; env->env\_ipc\_value = value; env->env\_ipc\_recving = 0; env->env\_tf.tf\_regs.reg\_eax = 0; env->env\_status = ENV\_RUNNABLE;

**return** 0;

}

**int32\_t** ipc\_recv(envid\_t \*from\_env\_store, **void** \*pg, **int** \*perm\_store) {

*// LAB 4: Your code here.* **if** (pg == NULL) { pg = (**void** \*)UTOP;

}

**int** r = sys\_ipc\_recv(pg); **if** (from\_env\_store) \*from\_env\_store = (r == 0)? thisenv->env\_ipc\_from: 0; **if** (perm\_store) \*perm\_store = (r == 0)? thisenv->env\_ipc\_perm: 0;

**if** (r) **return** r;

**return** thisenv->env\_ipc\_value;

}

**void** ipc\_send(envid\_t to\_env, **uint32\_t** val, **void** \*pg, **int** perm) {

*// LAB 4: Your code here.* **if** (pg == NULL) { pg = (**void** \*)UTOP;

}

**int** r; **while** ((r = sys\_ipc\_try\_send(to\_env, val, pg, perm)) < 0) { **if** (r != -E\_IPC\_NOT\_RECV) panic("ipc\_send: sys\_ipc\_send, %d, %e", r, r);

sys\_yield();

}

}

1. **xv6**实验系统分析
   1. **xv6** 的系统结构
   2. **xv6** 的引导

## **4.3 xv6** 的进程与调度分析

**4.4 xv6** 的内存管理

**4.5 xv6** 的文件系统

**4.6 xv6** 的 **I/O**

# **5** 总结与心得

1. 基于测试用例
2. 实验过程精致
3. 环境配置简单，依赖少，只需要 GCC 和 QEMU

完成该实验之后，经过反思，我们首先认为这个实验过程十分有趣，通过补充编写吃豆人这个游戏，来学习人工智能导论的基础知识。加州伯克利分校 [**?**] 的这种寓教于乐的方式很值得我们学习。其次，我们对人工智能导论的学习也产生了更浓厚的兴趣，在接下来的时间中，我们会更认真学习人工智能导论课程的知识。

# 参考文献

1. M. I. of Technology. 6.828 / fall 2017. [https://pdos.csail.mit.edu/6.828/ 2017/](https://pdos.csail.mit.edu/6.828/2017/). Accessed December 25, 2017.
2. M. I. of Technology. Documentation of xv6. [https://pdos.csail.mit.edu/6. 828/2017/xv6/book-rev10.pdf](https://pdos.csail.mit.edu/6.828/2017/xv6/book-rev10.pdf). Accessed December 25, 2017.
3. M. I. of Technology. mit-pdos/xv6-public: xv6 os. [https://github.com/ mit-pdos/xv6-public](https://github.com/mit-pdos/xv6-public). Accessed December 25, 2017.

附录**A** 实验代码

## **A.1** 通用搜索算法

asd

1. cross-compiler [↑](#footnote-ref-1)
2. <https://pdos.csail.mit.edu/6.828/2017/xv6/xv6-rev10.pdf> [↑](#footnote-ref-2)
3. Interrupt Descriptor Table [↑](#footnote-ref-3)
4. 做完之后需要使用 **make clean** 命令之后再 make。 [↑](#footnote-ref-4)
5. 参考<https://sourceware.org/gdb/current/onlinedocs/gdb/Memory.html> [↑](#footnote-ref-5)
6. Multiprocessor Support [↑](#footnote-ref-6)
7. Cooperative Multitasking [↑](#footnote-ref-7)
8. per-CPU state [↑](#footnote-ref-8)
9. global state [↑](#footnote-ref-9)
10. Per-CPU kernel stack [↑](#footnote-ref-10)
11. Copy-on-Write Fork [↑](#footnote-ref-11)
12. Preemptive Multitasking [↑](#footnote-ref-12)
13. Inter-Process communication (IPC) [↑](#footnote-ref-13)