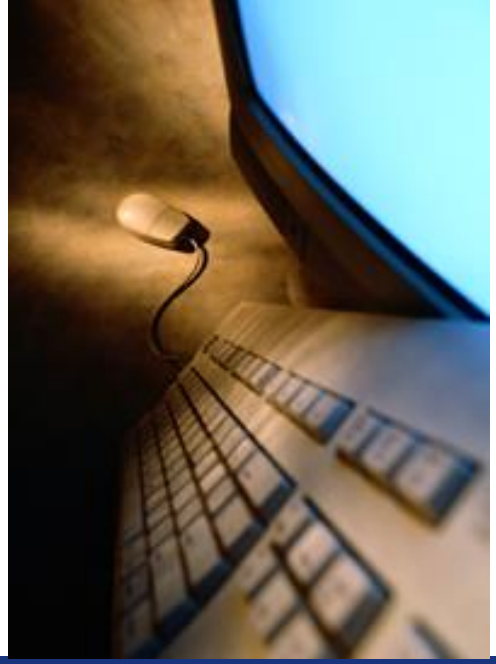
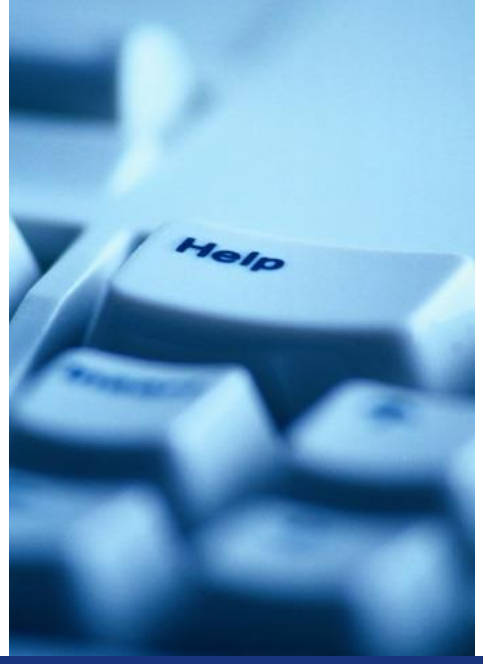



操作系统实验

Labs of Operating Systems

陈鹏飞
计算机学院
2021-03-02

日程

1. 实验目标
2. 实验安排
3. 实验环境
4. 实验考核
5. 实验参考

实验目标

课程目标

独立设计并实现面向特定CPU架构的原型操作系统！！！

```
; A simple boot sector
loop:
    jmp loop

times 510-($-$$) db 0
dw 0xaa55
```

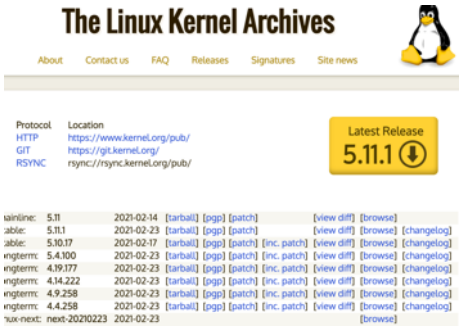
几行

```
[org 0x7c00] ; bootloader offset
    mov bp, 0x9000 ; set the stack
    mov sp, bp

    mov bx, MSG_REAL_MODE
    call print ; This will be written after the BIOS messages

    call switch_to_pm
    jmp $ ; this will actually never be executed

%include "../05-bootsector-functions-strings/boot_sect_print.asm"
%include "../09-32bit-gdt/32bit-gdt.asm"
%include "../08-32bit-print/32bit-print.asm"
%include "32bit-switch.asm"

[bits 32]
BEGIN_PM: ; after the switch we will get here
    mov ebx, MSG_PROT_MODE
    call print_string_pm ; Note that this will be written at the
    jmp $

MSG_REAL_MODE db "Started in 16-bit real mode", 0
MSG_PROT_MODE db "Loaded 32-bit protected mode", 0

; bootsector
times 510-($-$$) db 0
dw 0xaa55
```

几十行

```
8578 void
8579 readseg(uchar* pa, uint count, uint offset)
8580 {
8581   uchar* epa;
8582
8583   epa = pa + count;
8584
8585   // Round down to sector boundary.
8586   pa -= offset % SECTSIZE;
8587
8588   // Translate from bytes to sectors; kernel starts at sector 1.
8589   offset = (offset / SECTSIZE) + 1;
8590
8591   // If this is too slow, we could read lots of sectors at a time.
8592   // We'd write more to memory than asked, but it doesn't matter --
8593   // we load in increasing order.
8594   for(; pa < epa; pa += SECTSIZE, offset++)
8595     readsect(pa, offset);
8596 }
8597
8598
8599
```

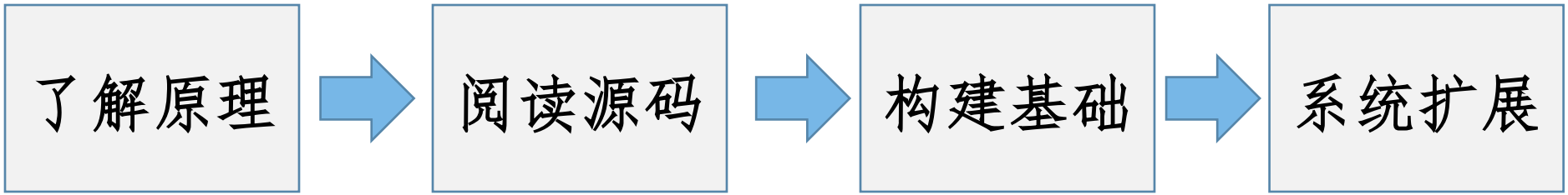
几千行



几千万行

聚沙成塔、集腋成裘

实验路线

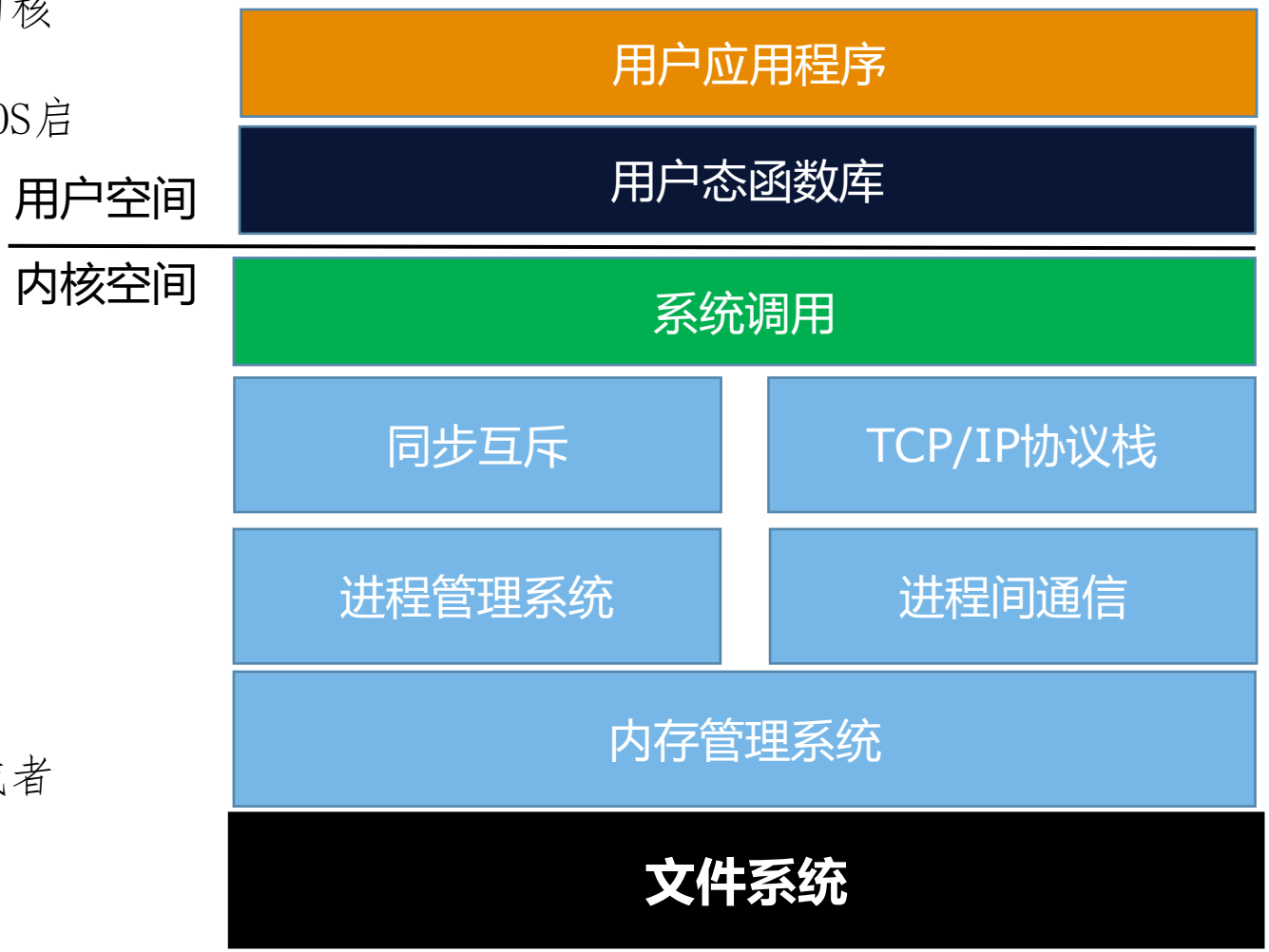


实验安排

本实验课程主要根据操作系统原理课程所讲授的原理知识,进行实验，从0设计与实现面向i386（32位）平台的原型操作系统。涉及的内容包括：***内核、启动、I/O、中断、虚拟存储、CPU调度、多线程并发、文件系统、Shell、多租户***等多方面的内容。共包括13次实验，其中2次实验可选。

实验安排

1. 编译内核/利用已有内核构建OS;
2. 实模式和保护模式下OS启动;
3. OS中断/异常;
4. 物理内存管理;
5. 虚拟内存管理;
6. 内核模式线程管理;
7. 用户模式线程管理;
8. 系统调用;
9. 处理器调度;
10. 同步与互斥;
11. 文件系统;
12. Shell （可选）;
13. 迁移到Risc-V架构/或者ARM架构（可选）;



实验安排

实验-1： 编译内核利用已有内核构建OS

熟悉现有Linux内核的编译过程和启动过程，并在自行编译内核的基础上构建简单应用并启动；利用精简的Busybox工具集构建简单的OS，熟悉现代操作系统的构建过程。此外，熟悉编译环境、相关工具集，并能够实现内核远程调试；

1. 搭建OS内核开发环境包括：代码编辑环境、编译环境、运行环境、调试环境等；
2. 下载并编译i386（32位）内核，并利用qemu启动内核；
3. 熟悉制作initramfs的方法；
4. 编写简单应用程序随内核启动运行；
5. 编译i386版本的Busybox，随内核启动，构建简单的OS；
6. 开启远程调试功能，进行调试跟踪代码运行；
7. 撰写实验报告；

实验安排

实验-2： 实模式和保护模式下的OS启动

了解操作系统启动的原理，利用汇编语言实模式即（20位地址空间）的启动和保护模式即（32位地址空间）下的启动方法，并能够在此基础上利用汇编或者C程序实现简单的应用；

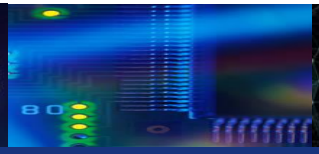
1. 回顾、学习32位汇编语言的基本语法；
2. 编写简单的汇编程序，进行中断、输入输出测试；
3. 实现实模式下OS启动；
4. 在实模式下利用汇编/C/Rust等实现简单的应用；
5. 实现保护模式下OS启动；
6. 在保护模式下利用汇编/C/Rust等实现简单的应用；
7. 比较实模式和保护模式的不同；

实验安排

实验-4： OS中断/异常

启动操作系统的 bootloader，了解操作系统启动前的状态和要做的事，了解运行操作系统的硬件支持，操作系统如何加载到内存中，理解两类中断–“外设中断”，“异常”等。

- 编译运行直接与硬件交互的系统程序
- 启动 bootloader 的过程
- 实现中断处理机制
- 输出字符的方法
- 调试系统程序的方法



实验安排

实验-4：物理内存管理；

理解分页模式，了解操作系统如何管理连续空间的物理内存。

- 理解内存地址的转换和保护
- 实现页表的建立和使用方法
- 实现物理内存的管理方法
- 了解常用的减少碎片的方法

实验安排

实验-5：虚拟内存管理；

了解页表机制和换出（swap）机制，以及中断-“故障中断”、缺页故障处理等，基于页的内存替换算法。

- 理解换页的软硬件协同机制
- 实现虚拟内存的 Page Fault 异常处理
- 实现页替换算法

实验安排

实验-6：内核模式线程；

了解如果利用 CPU 来高效地完成各种工作的设计与实现基础，如何创建相对与用户进程更加简单的内核态线程，如何对内核线程进行动态管理等。

- 建立内核线程的关键信息
- 实现内核线程的管理方法

实验安排

实验-7：用户模式线程；

了解用户态进程创建、执行、切换和结束的动态管理过程，了解在用户态通过系统调用得到内核态的内核服务的过程。

- 建立用户进程的关键信息
- 实现用户进程管理
- 分析进程和内存管理的关系
- 实现系统调用的处理过程

实验安排

实验-8： 系统调用

了解Linux系统调用的基本原理，参考现有系统调用的实现方法，实现简单的系统调用如Write、Read文件等；

1. 了解当前Linux系统调用的实现原理；
2. 在已实现的OS的基础上添加系统调用如内存拷贝、文件读写或者其他自定义调用；
3. 验证系统调用的性能比如执行时间等

实验安排

实验-9：处理器调度；

理解操作系统的调度过程和调度算法。

- 熟悉系统调度器框架，以及内置的 Round-Robin 调度算法
- 基于调度器框架实现一个调度器算法

实验安排

实验-10：同步互斥；

了解进程间如何进行信息交换和共享，并了解同步互斥的具体实现以及对系统性能的影响，研究死锁产生的原因，以及如何避免死锁。

- 熟悉同步互斥的实现机制
- 理解基本的 spinlock、semphpore、condition variable 的实现
- 实现基于各种同步机制的同步问题。

实验安排

实验-11：文件系统；

了解文件系统的具体实现，与进程管理等的关系，了解缓存对操作系统 IO 访问的性能改进，了解虚拟文件系统（VFS）、buffer cache 和 disk driver 之间的关系。

- 掌握基本的文件系统系统调用的实现方法
- 了解一个基于 inode 方式的 SFS 文件系统的设计与实现
- 了解文件系统抽象层-VFS 的设计与实现

实验安排

实验-12：shell；

In this project, you'll build a simple Unix shell. The shell is the heart of the command-line interface, and thus is central to the Unix/C programming environment. Mastering use of the shell is necessary to become proficient in this world; knowing how the shell itself is built is the focus of this project.
There are three specific objectives to this assignment:
•To further familiarize yourself with the Linux programming environment.
•To learn how processes are created, destroyed, and managed.
•To gain exposure to the necessary functionality in shells.

you will implement a *command line interpreter (CLI)* or, as it is more commonly known, a *shell*. The shell should operate in this basic way: when you type in a command (in response to its prompt), the shell creates a child process that executes the command you entered and then prompts for more user input when it has finished. The basic function includes:
Interactive loop; exit(); getline();ls;cd;path;redirection（重定向）;program error（故障处理）;self-defined functions（支持自定义函数）;

实验安排

实验-13：迁移到Risc-V架构或者ARM架构；

可以参考uCore/XV6的ARM和Risc-V实现，将自研操作系统迁移到ARM或者Risc-V架构，进行验证，并进行性能评测以及与Intel i386平台对比。了解不同架构的实现区别。

1. 熟悉Risc-V或者ARM架构；
2. 迁移已实现的OS到上述两种架构之一；
3. 使用Qemu模拟Risc-V或者ARM，运行操作系统；
4. 对比不同架构OS的区别；
5. 总结不同架构的优势；

实验环境

推荐以下环境和工具：

1. 操作系统Ubuntu(64位)；
2. Windows主机虚拟化软件Virtualbox；
3. 代码编辑环境：Vscode+nasm+C/C++插件；
4. 汇编编译器：nasm （32位汇编）；
5. C/C++编译器：gcc/g++ （64位）；
6. 装载工具ld（64位）；
7. ELF分析工具:readelf；
8. 反汇编：objdump；
9. 调试工具：gdb；
10. 写磁盘文件工具：dd
11. Make工具：cmake；
12. 运行工具：qemu（32位）；

实验考核

1. 不限语言，C/C++/Rust都可以；
2. 不限平台，Windows/Linux都可以；
3. 不限CPU，ARM/Intel/Risc-V都可以；
4. 实现原型操作系统32位以上；
5. 实现实验所要求的的功能；
6. 提供实验报告、源码、运行截屏；
7. 根据完成实验的数量和质量打分，参考已有源码但是没有任何修改，报告详细完整可以得到基本分数；
8. 可以将现有的参考实现如Xv6，uCore等改造成其他架构如将uCore运行在ARM64上等；

实验参考

1. 清华大学: 操作系统实验指导，https://github.com/kiukotsu/ucore；https://objectkuan.gitbooks.io/ucore-docs/content/；
2. XV6中文文档：https://th0ar.gitbooks.io/xv6-chinese/content/;
3. https://github.com/ranxian/xv6-Chinese;
4. OSTEP实验参考：https://github.com/remzi-arpacidusseau/ostep-projects;
5. 汇编语言编程：http://c.biancheng.net/asm/10/；
6. Linux Kernel: https://www.kernel.org/;
7. MikeOS: http://mikeos.sourceforge.net/; http://mikeos.sourceforge.net/write-your-own-os.html;
8. **rust-raspberrypi-OS-tutorials, https://github.com/rust-embedded/rust-raspberrypi-OS-tutorials;**
9. **Writing an OS in Rust: https://os.phil-opp.com/;**
10. 使用Rust写OS: https://12101111.github.io/write-os-in-rust/;
11. OS-tutorial: https://github.com/cfenollosa/os-tutorial;
12. Xv6 code: https://github.com/mit-pdos/xv6-public;
13. Redox OS: https://github.com/redox-os;
14. 中山大学张钧宇：NeXon-tutorial-private-main,

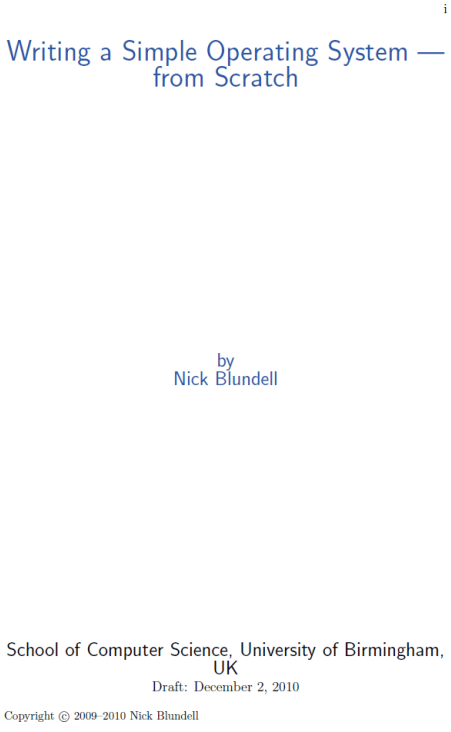
实验参考

参考文档

The little book about OS development

Erik Helin, Adam Renberg

2015-01-19 | Commit: fe83e27dab3c39930354d2dea83f6d4ee2928212



i

Writing a Simple Operating System — from Scratch

by
Nick Blundell

School of Computer Science, University of Birmingham, UK

Draft: December 2, 2010

实验-1

实验-1： 编译内核利用已有内核构建OS

熟悉现有Linux内核的编译过程和启动过程，并在自行编译内核的基础上构建简单应用并启动；利用精简的Busybox工具集构建简单的OS，熟悉现代操作系统的构建过程。此外，熟悉编译环境、相关工具集，并能够实现内核远程调试；

1. 搭建OS内核开发环境包括：代码编辑环境、编译环境、运行环境、调试环境等；
2. 下载并编译i386（32位）内核，并利用qemu启动内核；
3. 熟悉制作initramfs的方法；
4. 编写简单应用程序随内核启动运行；
5. 编译i386版本的Busybox，随内核启动，构建简单的OS；
6. 开启远程调试功能，进行调试跟踪代码运行；
7. 撰写实验报告；

实验-1

安装步骤

1. 在Windows环境中下载安装Virtualbox，用于启动虚拟机；如果本身是Ubuntu环境则不需要这个步骤；
2. 安装Ubuntu 1804桌面版，并配置清华安装源；
3. 安装Vscode以及汇编、C/C++插件；
4. 安装nasm；
5. 安装qemu-system-i386；
6. 安装安装cmake、gdb工具；
7. 安装objdump，readelf；
8. 保存虚拟机镜像；

实验-1

❖ 编译Linux内核

1. 在https://www.kernel.org/下载内核5.10;
2. 将内核编译成i386 32位版本;
   - 利用i386配置文件配置内核，该配置在arch/x86/configs下面;
   - make i386_defconfig
   - make menuconfig (安装libncurses5-dev)
   - 开启debug; (compile the kernel with debug)
   - 保存退出;
   - Make -j4（时间较长）;
   - 找到linux压缩镜像：bzImage，一般在arch/x86/boot下面

实验-1

- Qemu启动内核并开启远程调试

1. qemu -kernel bzImage -s -S -append "console=ttyS0" -nographic

2. Gdb装载vmlinux进行远程调试;
   break start_kernel;

实验-1

制作Initramfs

Linux系统启动时使用initramfs (initram file system), initramfs可以在启动早期提供一个用户态环境，借助它可以完成一些内核在启动阶段不易完成的工作。当然initramfs是可选的，Linux中的内核编译选项默认开启initrd。在下面的示例情况中你可能要考虑用initramfs。

- 加载模块，比如第三方driver
- 定制化启动过程 (比如打印welcome message等)
- 制作一个非常小的rescue shell
- 任何kernel不能做的，但在用户态可以做的 (比如执行某些命令)

一个initramfs至少要包含一个文件，文件名为/init。内核将这个文件执行起来的进程作为main init进程(pid 1)。当内核挂载initramfs后，文件系统的根分区还没有被mount，这意味着你不能访问文件系统中的任何文件。如果你需要一个shell，必须把shell打包到initramfs中，如果你需要一个简单的工具，比如ls，你也必须把它和它依赖的库或者模块打包到initramfs中。总之，initramfas是一个完全独立运行的体系。

另外initramfs打包的时候，要求打包成压缩的cpio档案。cpio档案可以嵌入到内核image中，也可以作为一个独立的文件在启动的过程中被GRUB load。

实验-1

制作Initramfs

1. 编写简单的C程序如HelloWorld;
2. 编译成32位可执行文件 （apt-get install libc6-dev-i386）;
3. cpio打包成initramfs, echo helloworld | cpio -o --format=newc > hwinitramfs;
4. 启动内核，并加载initramfs:

```
# qemu -kernel bzImage -initrd hwinitramfs -append "console=ttyS0 rdinit=helloworld" -nographic
```

实验-1

- 编译并启动Busybox

1. 在https://github.com/meefik/busybox，下载Busybox;
2. 配置Busybox，静态编译成32位可执行文件;
3. Qemu启动内核并加载Busybox;

谢谢