西南林业大学

本科毕业（设计）论文

（二〇一八届）

题　　目： RongOS — 简单操作系统的实现

分院系部： 大数据与智能工程学院

专　　业： 计算机科学与技术专业

姓　　名： 蒲启元

导师姓名： 王晓林

导师职称： 讲师

二〇一八年六月

RongOS — 简单操作系统的实现

蒲启元

（西南林业大学 大数据与智能工程学院，云南昆明 650224）

摘　要： 操作系统管理着计算机的硬件和软件资源，它是向上层应用软件提供服务（接口）的核心系统软件，这些服务包括进程管理，内存管理，文件系统，网络通信，安全机制等。操作系统的设计与实现则是软件工业的基础。为此，在国务院提出的《中国制造 2025》中专门强调了操作系统的开发[1]。但长期以来，操作系统核心开发技术都掌握在外国人手中，技术受制，对于我们的软件工业来说很不利。本项目从零开始设计开发个简单的操作系统，包括 boot loader，中断，内存管理，图形接口，多任务，以及在这个系统上的几个小应用等。尽管这个系统很简单，但它为自主开发操作系统做了尝试。

关键词： 操作系统，进程，内存，中断，boot loader

RongOS — A simple OS implementation

Qiyuan Pu

School of Big Data and Intelligence Engineering
Southwest Forestry University
Kunming 650224, Yunnan, China

Abstract: Operating system manages the resources of hardware and software, it lies in the core of the system software and provides services (interfaces) to upper applications. These services include process management, memory management, file system, network communication, security mechanism and more. Operating system development is the foundation and core of software industry. Therefore, *Made in China 2025* emphasizes the development of operating system that put forward by The State Council of China. For a long time, however, the OS kernel development technology is mastered by foreigner, due to technical limitations it is detrimental to our software industry. So this project will design and develop a simple operating system, including boot loader, interrupt, memory management, graphic interface, multitasking, and some little applications based on this system. In spite of the simplicity of this system, it's a small trying for autonomous development operating system.

Key words: operating system, boot loader, process, interrupt, memory management

目　录

插图目录

表格目录

List of Corrections

1 Introduction

1.1 Background

Contemporary software systems are beset by problems that create challenges and opportunities for broad new OS research. There are five areas could improve user experience including dependability, security, system configuration, system extension, and multiprocessor programming.

The products of forty years of OS research are sitting in everyone's desktop computer, cell phone, car, etc., and it is not a pretty picture. Modern software systems are broadly speaking complex, insecure, unpredictable, prone to failure, hard to use, and difficult to maintain. Part of the difficult is that good software is hard to write, but in the past decade, this problem and more specific shortcomings in systems have been greatly exacerbated by increased networking and embedded systems, which placed new demands that existing architectures struggled to meet. These problems will not have simple solutions, but the changes must be pervasive, starting at the bottom of the software stack, in the operating system.

The world needs broad operating system research. Dependability, security, system configuration, system extension, and multi-processor programming illustrate areas were contemporary operating systems have failed to meet the software challenges of the modern computing environment.

Please reference [1].

1.2 Preliminary Works

1.2.1 Development Environment

OS platform: Debian 9, Linux kernel 4.12.0-1-amd64

[1] `https://www.usenix.org/legacy/events/hotos05/final_papers_backup/hunt/hunt_html/`

Editor: GNU Emacs 25.2.2

Run time VM: QEMU emulator 2.8.1

Assembler: Nask.

Compiler: CC1(Based on gcc)

Debugger: TODO: debugger?

TODO!

1.2.2 Tools

Some tools used to develop RongOS, see tools[2]. Note that these tools are exe format based on Windows system. In these tools, nask.exe is used for compiling assembly program and cc1 is a compiler, gas2nask.exe convert gas program to nask program, and obj2bim links all object files to bim format.

1.2.3 Platform Setup

Debian System: there is a small tutorial[3]. This tutorial is a sample one for set up development environment, the main thing is operating system — Debian.

Qemu

QEMU is a generic and open source machine emulator and virtualizer. In this development process, QEMU used to test running result.

Install QEMU, for my x86_64 architecture:

```
$ sudo apt-get install qemu-system-x86_64
```

Wine

Wine (originally an acronym for ”Wine Is Not an Emulator”) is a compatibility layer capable of running Windows applications on several POSIX-compliant operating systems, such as Linux, macOS, and BSD.

Note that the tools are exe format, so on Debian system, Wine need to be install:

[2] https://github.com/Puqiyuan/RongOS/tree/master/Tools
[3] http://cs2.swfc.edu.cn/~wx672/lecture_notes/linux/install.html

```
$ sudo apt-get update
$ sudo apt-get install wine
```

Debian i386 support

On 64-bit systems you should enable a 32-bit architecture for multiarch. This is needed for running 32-bit Windows applications (many modern apps are still 32-bit), but also for large parts of the Windows subsystem itself, our development tools are 32-bit Windows applications, so add this architecture:

```
$ sudo dpkg --add-architecture i386
$ sudo apt-get update
```

2 Leading Knowledge

2.1 Instruction Set

An instruction set architecture (ISA) is an abstract model of a computer. It is also referred to as architecture or computer architecture. An ISA defines everything a machine language programmer needs to know in order to program a computer.

An ISA may be classified in a number of different ways. A common classification is by architectural complexity. A complex instruction set computer (CISC) has many specialized instructions, some of which may only be rarely used in practical programs. A reduced instruction set computer (RISC) simplifies the processor by efficiently implementing only the instructions that are frequently used in programs, while the less common operations are implemented as subroutines, having their resulting additional processor execution time offset by infrequent use.

On traditional architectures, an instruction includes an opcode that specifies the operation to perform, such as add contents of memory to register—and zero or more operand specifiers, which may specify registers, memory locations, or literal data. Please reference [1].

The simple RongOS is based on x86 architecture, the following instructions are common for RongOS:

db: the abbreviation of define byte, write a byte, also 8 bits to file.

resb: the abbreviation of reserve byte, reserved bytes and filling 0x00 in these reserved space.

dw: the abbreviation of define word, write two bytes, also 16 bits to file.

dd: the abbreviation of define double-word, write four bytes, also 32 bits to file.

org: load the program to specified address.

jmp: jump to another instruction.

mov: assign the right value to left variable.

[1] https://en.wikipedia.org/wiki/Instruction_set_architecture

jc: the abbreviation of jump if carry, it means if carry flag is 1, jump.

jnc: jump if not carry.

jae: jump if above or equal.

jbe: jump if below or equal.

jb: jump if below.

equ: equ is the abbreviation of equal.

ret: end of function, return.

in: get signal from device.

out: send signal to device.

cli: clear interrupt flag, set it to 0.

sti: set interrupt flag, set it to 1.

pushfd: push flags double-word

popfd: pop flags double-word

2.2 x86 Registers

In computer architecture, a processor register is a quickly accessible location available to a computer's central processing unit (CPU). Registers usually consist of a small amount of fast storage, although some registers have specific hardware functions, and may be read-only or write-only. Almost all computers, whether load/store architecture or not, load data from a larger memory into registers where it is used for arithmetic operations and is manipulated or tested by machine instructions. Manipulated data is then often stored back to main memory, either by the same instruction or by a subsequent one. Modern processors use either static or dynamic RAM as main memory, with the latter usually accessed via one or more cache levels.

Processor registers are normally at the top of the memory hierarchy, and provide the fastest way to access data. The term normally refers only to the group of registers that are directly encoded as part of an instruction, as defined by the instruction set. Registers are normally measured by the number of bits they can hold, for example, an "8-bit register" or a "32-bit register". For x86 architecture, these registers existing:

ax: accumulator

bx: base

cx: counter

dx: data

bl: base low

al: accumulator low

cl: counter low

dl: data low

bh: base high

ah: accumulator high

ch: counter high

dh: data high

sp: stack pointer

bp: base pointer

si: source index

di: destination index

es: extra segment

cs: code segment

ss: stack segment

ds: data segment

fs: no name

In these registers, bx, bp, si and di can be used to specify the address of memory. But ax, cx, dx and sp can not. When mov instruction using, the number of bit of operation number should be same. 16 bits registers: ax, cx, dx, bx, sp, bp, si, di, es, cs, ss, ds and fs. 8 bits registers: al, cl, dl, bl, ah, ch, dh and bh. Actually, all these 8 bits registers are part of corresponding 16 bits registers, low 8 bits or high 8 bits.

2.3 Interrupt Call

BIOS interrupt calls perform hardware control or I/O functions requested by a program, return system information to the program, or do both. A key element of the purpose of BIOS calls is abstraction-the BIOS calls perform generally defined functions, and the specific details of how those functions are executed on the particular hardware of the system are encapsulated in the BIOS and hidden from the program. The following interrupt calls are common for RongOS:

2.4 Memory Map

In the boot process, a memory map is passed on from the firmware in order to instruct an operating system kernel about memory layout. It contains the information regarding the size of total memory, any reserved regions and may also provide other details specific to the architecture. For load RongOS to memory, the memory lay out should be clarify:

Interrupt Number	Register Parameter	Return Value	Function
0x10	ah=0x0e(write character in tty mode) al=character code bh=0, bl=colorcolor	null	video services
0x13	ah=0x02(read sectors) ah=0x03(write sectors) ah=0x04(verify sectors) ah=0x0c(seek to specified track) al=number of sectors processing ch=cylinder & 0xff cl=sector number dh=header number dl=driver number es:bx=buffer address	FLACS.CF=0 no error, ah = 0 FLAGS.CF=1 error, ah=error number	disk services

表 2-1 RongOS interrupt calls

Range(in hexadecimal)	Range(in decimal)	Size(in bytes)	Usage
0000 — 03ff	0000 — 1023	1024	interrupt vector table
0400 — 04ff	1024 — 1279	256	BIOS data area
0500 — 051f	1280 — 1311	32	Reserved
0520 — 7bff	1312 — 31743	30432	conventional memory
7c00 — 7dff	31744 — 32255	512	master boot record
7e00 — 9ffff	32256 — 655359	623104	conventional memory
a0000 — affff	655360 — 720895	64K	VGA graphics RAM
b0000 — b7fff	720896 — 753663	32K	monochrome text mode
b8000 — bffff	753664 — 786431	32K	color text mode
c0000 — c7fff	786432 — 819199	32K	VGA video ROM
c8000 — cbfff	819200 — 835583	16K	IDE hard drive
cc000 — cffff	835584 — 851967	16K	optional adaper

2.5 Floppy Disk

There are many ways to boot an operating system, from hard disk, USB, floppy disk,etc. The structure of floppy disk is simple and for this simple operating system, it’s enough.

Fig. 2-1 shows the inside of a floppy disk:

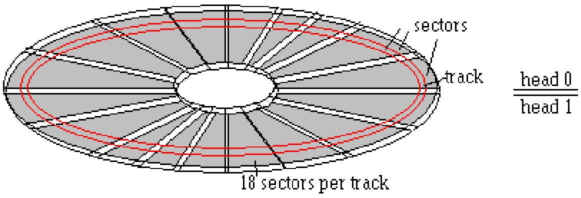

图 2-1　Floppy Disk Structure

A floppy disk, also called a floppy, diskette, or just disk, is a type of disk storage composed of a disk of thin and flexible magnetic storage medium, sealed in a rectangular plastic enclosure lined with fabric that removes dust particles. Floppy disks are read and written by a floppy disk drive (FDD).

For 3.5 inch HD floppy, There are 80 cylinders from the outermost to the core on each side, numbering 0, 1, ..., 79. The head can assign be 0 or 1, representing two sides of floppy. When specify head number and cylinder number, forming a ring, named track in jargon. The track is large so we divide it to 18 small parts, named sector. A sector can store 512 byte. So the capacity of a floppy is:

$$18 \times 80 \times 2 \times 512 = 1474560 Byte = 1440 KB$$

3 Design

3.1 Top Level Design

3.1.1 32-bit Mode and Import C Codes

4 Implementation

4.1 Boot Loader

4.1.1 Flowchart of Boot Loader

Fig. 4-1 shows how the boot loader works.

TODO: need a better chart

图 4-1 Flowchart of Boot Loader

The boot loader is implemented in Intel assembly. It works as following:

1. **Display boot information:** Firstly, the boot sector display some boot information, when $al = 0$, the null character of boot information hit. Interrupt $0x10$ is used for show a character. Appendix A.1.1 is the code to perform this function.
2. **Read the second sector:** Then jump to load C0-H0-S2, ax register saved the address where beginning puts the sectors from floppy. And preparing parameters for interrupt $0x13$ in registers. The $0x13$ interrupt used for read sector from floppy to memory. Appendix A.1.2 is the code to perform this function.
3. **Read two sides of a track:** If there is a carry, representing some thing wrong when read floppy, so reset the registers and try again read floppy, until five times trying. Register si is a counter. If no carry, jump to next segmentation, as one sector read to memory already, the address space should increase 512 byte. Then sector number(cl register) added 1 and compare it to 18, if it's smaller than 18, jump to $readloop$, read the next sector. If the value of cl register bigger or equal to than 18, meaning that one track 18 sector in this side of floppy read already, then reversed the head, add 1 to dh register. If the value of dh register after adding larger than or equal to 2, it's saying the original head is 1, one track of two sides read already. Otherwise the value of dh register smaller than 2, read this side indicating by dh register, jump to $readloop$ segmentation. Appendix A.1.3 is the code to perform this function.

4. **The next cylinder:** So the next step is moving a cylinder, add 1 to register ch. Otherwise the value of dh register smaller than 2, read this side indicating by dh register, jump to $readloop$ segmentation. After ch register add 1, if it's smaller than 10, jump to $readloop$, otherwise end loading floppy to memory process, for we only load ten cylinders of floppy. Appendix A.1.4 is the code to perform this function.

4.1.2 Running Result

Fig. 4-2 shows the running results of boot loader. From this picture we see that the boot loader loaded 10 cylinders from floppy successfully.

TODO: need a better pic.

图 4-2 Running Result of Boot Loader

4.2 32-bit Mode and Import C Codes

4.3 Screen Display and Text

4.4 Control Mouse

4.5 Memory Management

4.6 Making Window

4.7 Timer

4.8 Multitasking

4.9 Command Line Window

4.10 API

4.11 OS Protection

4.12 Graphics Processing

4.13 Window Operation

4.14 Application Protection

4.15 File Operation

4.16 Some Applications

5 Prospects And Shortages

参考文献

[1] 国务院, 中国制造 2025, chinese, **2015-05**.

指导教师简介

王晓林，男，49 岁，硕士，讲师，毕业于英国格林尼治大学，分布式系统专业，现任西南林业大学计信学院教师，执教 Linux、操作系统、网络技术等方面的课程，有丰富的 Linux 教学和系统管理经验。

致　谢

首先我想感谢我的老师，王晓林。大学期间，他给了我很多指导，包括专业方面和上大学的意义等。很多时候，他对学生的要求看起来都是不近情理的，但正是通过这个“痛苦”的过程，我锻炼了坚强的意志，和战胜困难的信心。谢谢你，王老师。我最想感谢的是我的女友，她容忍我在完成这个设计时的很多个夜晚不陪她，给我支持，鼓励我，不抱怨。所以我愿意把这个简单操作系统命名为 RongOS, 蓉便是她名字的最后一个字。谢谢你，我最亲爱的。

附录 A　Main Program Code

A.1　Boot loader

A.1.1　Display boot information

```
init:
        mov al, [si]
        add si, 1 ; increment by 1.
        cmp al, 0
        je load ; if al == 0, jmp to load, the msg_init info displayed.
; the lastest character is null character, coding in 0.

        mov ah, 0x0e ; write a character in TTY mode.
        mov bx, 15   ; specify the color of the character.
        int 0x10 ; call BIOS function, video card is number 10.
        jmp init
```

A.1.2　Read the second sector

```
load:
        mov ax, 0
        mov ax, 0x0820 ; load C0-H0-S2 to memory begin with 0x0820.
        mov es, ax
        mov ch, 0 ; cylinder 0.
        mov dh, 0 ; head 0.
        mov cl, 2 ; sector 2.

```

```

readloop:
        mov si, 0 ; si register is a counter, try read a sector
; five times.


retry:
        mov ah, 0x02 ; parameter 0x02 to ah, read disk.
        mov al, 1 ; parameter 1 to al, read disk.
        mov bx, 0
        mov dl, 0x00 ; the number of driver number.
        int 0x13 ; after prepared parameters, call 0x13 interrupted.
```

A.1.3 Read two sides of a track

```
        jnc next ; if no carry read next sector.
        add si, 1 ; tring again read sector, counter add 1.
        cmp si, 5 ; until five times
        jae error ; if tring times large than five, failed.

        ; reset the status of floppy and read again.
        mov ah, 0x00
        mov dl, 0x00
        int 0x13
        jmp retry


next:
        mov ax, es
        ; we can not directly add to es register.
        add ax, 0x0020 ; add 0x0020 to ax
```

```
        mov es, ax ; the memory increase 0x0020 * 16 = 512 byte.
        ; size of a sector.
        add cl, 1 ; sector number add 1.
        cmp cl, 18 ; one track have 18 sector.
        jbe readloop ; jump if below or equal 18, read the next sector.
        mov cl, 1 ; cl number reset to 1, ready to read the other side.
        add dh, 1 ; the other side of floppy.
        cmp dh, 2 ; only two sides of floppy.
        jb readloop ; if dh < 2, read 18 sectors of the other sides
```

A.1.4 The next cylinder

```
mov dh, 0 ; after finished read the other side, reset head to 0.
add ch, 1 ; two sides of a cylinder readed, add 1 to ch.
cmp ch, CYLS ; read 10 cylinders.
jb readloop
```