西 南 林 业 大 学

本 科 毕 业 （设 计） 论 文

（二〇一八届）

题　　目： RongOS — 一个简单操作系统的设计与实现

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 — 一个简单操作系统的设计与实现

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一个简单的操作系统，包括 boot loader，中断，内存管理，图形接口，多任务等功能模块，以及能运行在这个系统之上的几个小应用程序。尽管这个系统很简单，但它是自主开发操作系统的一次尝试。

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 hardware and software resources in a running computer system. It is the core of any modern software system and provides services (interfaces) to upper layer applications. The services it provides 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 long time, however, the OS kernel development technology is dominated by foreigners. This technical limitation is detrimental to the development of our software industry. In this project, we presents a simple operating system which includes a boot loader, interrupt services, memory management functions, a graphic interface, and multi-process management functions. Also, some trivial user-level applications are provided for system testing purpose. This simple toy OS is an experimental trial for developing an operating system from scratch.

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

Contents

List of Figures

List of Tables

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[2].

1.2 Preliminary Works

1.2.1 Development Environment

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

Editor: GNU Emacs 25.2.2

Run time VM: QEMU emulator 2.8.1

Assembler: Nask

Compiler: CC1(Based on gcc)

Debugger: GNU gdb 7.12

Version Control: git 2.15

1.2.2 Tools

Some tools were used to develop RongOS, See *tools*[1]. Note that these tools are Windows executable. Please install wine if you want to run these tools on Linux. In these tools, the most important ones are:

nask.exe: the assembler, a modified version of NASM[3]

cc1: the C compiler

1.2.3 Platform Setup

The development platform (mainly the Debian system) was set up by following the *Debian Installation tutorial*[2]. The main steps include:

1. Installing the base Debian system;
2. Installing necessary software tools, such as emacs, web browser, qemu, wine, etc.;
3. Cloning configuration files by following the tutorial mentioned above;
4. Some more fine tweaks to satisfy my personal needs.

Qemu

QEMU is a generic and open source machine emulator and virtualizer[4]. In this project, QEMU was used as the test bed.

Installing QEMU for my x86_64 architecture can be easily done by executing the following command:

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

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

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[5].

Because the tools I used in this project are in Windows executable format, so on Debian system, Wine is needed to be installed:

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

Debian i386 support

On 64-bit systems you need to enable multi-arch support for running 32-bit Windows applications (many modern apps are still 32-bit, also for large parts of the Windows subsystem itself). Our development tools were 32-bit Windows applications, so we needed to have i386 support for our 64-bit Linux system.

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

2 Leading Knowledge

2.1 Layers

2.2 Memory Management

2.2.1 Overview

2.2.2 Round Down/Up and Page Size

2.3 Mouse

2.4 The Leap — Road to the 32 Bit Mode

2.5 Data Structure

2.6 Programmable Interrupt Controller

2.7 C Language Basic

2.8 Segments and Descriptors

The so-called segmentation is to divide a total of 4 GB of memory into many blocks in its own way. The start address of each block is treated as 0.

In this way, in order to represent a segment, the following information is required:

- The size of the segment
- Where is the starting address of the segment
- Segment management properties

All this information is represented by 8 bytes(64 bits). But the register used to specify the segment is only 16 bits. Therefore, the segment selector is stored in the segment register, and the segment management information(the above three information) is referenced by the segment selector. Although the segment register has 16 bits, only high 13 bits are available due to the CPU design. Therefore, the segment selector is in the range of 0 to 8191. In total, there are 8192 segments, and a total of $8192 \times 8 = 65536$(64KB) bytes are required to store the management information of these segments. This 64-byte message is called GDT. Obviously, the CPU does not have such a large storage capacity. So store this information somewhere in memory. A special register in the CPU is GDTR(global descriptor table register). This register is used to reference the GDT address in memory and record how many valid segments are set.

2.9 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[6].

This simple RongOS is based on x86 architecture, the following instructions are commonly used in programming 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.

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.

lgdt: load content from specified memory to initialize GDT (global descriptor table) register.

lidt: load content from specified memory to initialize IDT (interrupt descriptor table) register.

2.10 x86 Registers

In computer architecture, a processor register is a quick 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[7].

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, the following registers exist, see 3.4.1 and 3.4.2[8]:

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

gs: no name

Among 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 is used, the number of bits of source number should be the same with destination operand.

16-bit registers: `ax`, `cx`, `dx`, `bx`, `sp`, `bp`, `si`, `di`, `es`, `cs`, `ss`, `ds`, and `fs`.

8-bit registers `al`, `cl`, `dl`, `bl`, `ah`, `ch`, `dh`, and `bh`.

Actually, as shown in Fig. 2-1, all these 8-bit registers are parts of corresponding 16-bit registers.

	8 bits	8 bits
AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

Fig. 2-1 x86 registers

2.11 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[9]. The interrupt calls are commonly used in RongOS are listed in Table 2-1.

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

Table 2-1 RongOS interrupt calls

2.12 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[1]. For loading RongOS to memory, the memory layout should be clarified as in Table 2-2.

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 adapter

Table 2-2 RongOS Memory Layout

2.13 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. 2-2 shows the inside of a floppy disk:

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

[1] http://hypervsir.blogspot.com/2014/09/approach-to-retrieving-bios-memory-map.html

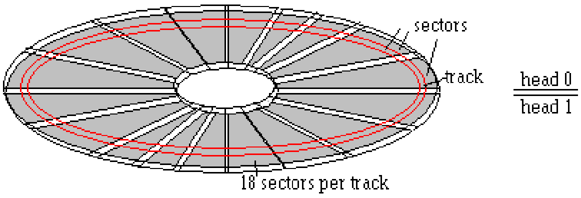

Fig. 2-2 Floppy disk structure

enclosure lined with fabric that removes dust particles. Floppy disks are read and written by a floppy disk drive (FDD)[10].

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:

$$18 \times 80 \times 2 \times 512 = 1474560\,Byte = 1440\,KiB$$

3 Design

3.1 Top Level Design

All applications use the functions provided by the operating system kernel through API calls. Doing so protects the operating system. As picture 3-1 shown:

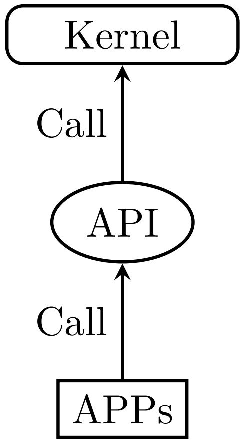

Fig. 3-1 Top-level design

3.2 Kernel

3.2.1 Module Relationship

Fig. 3-2 shows how the various modules in the kernel are related.

Fig. 3-3, 3-4 and 3-5 show the usage instructions of various arrow in 3-2.

3.2.2 Datastructure in Kernel

The tables 3-1 describes the data structure used by RangOS in tabular form.

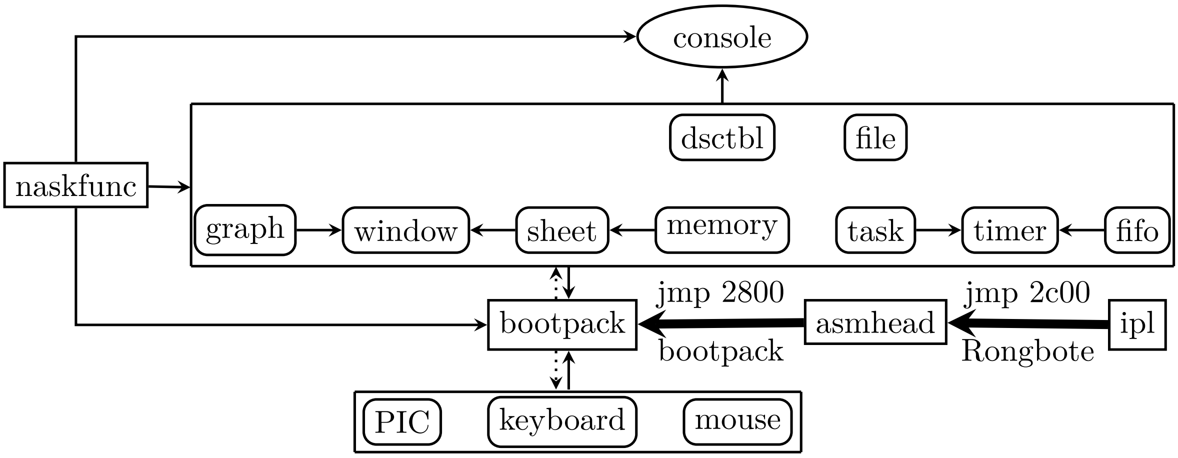

Fig. 3-2 modules in kernel

Fig. 3-3 A calls the initialization function in B to initialize the structure in B.

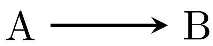

Fig. 3-4 A provides services to B

Fig. 3-5 Program running from A to B

struct BOOTINFO		
Name	Type	Meaning
cyls	char	number of cylinders to read
leds	char	keyboard state at boot
vmode	char	bits of color of graphics card
reserve	char	reserved bytes
scrnx	short	screen resolution of x
scrny	short	screen resolution of y
vram	char*	the starting address of the image buffer

Table 3-1 Structure of BOOTINFO

struct FIFO32		
Name	Type	Meaning
buf	int*	the address of FIFO32 buffer
p	int	the writing address
q	int	the reading address
size	int	the size of FIFO32 buffer
free	int	how many space free
flags	int	the states of FIFO32 buffer
task	struct TASK*	point to a task

Table 3-2 Structure of FIFO32

struct SEGMENT_DESCRIPTOR		
Name	Type	Meaning
limit_low	short	the low part of segment size
base_low	short	the low part of base address
base_mid	char	the middle part of base address
access_right	char	read and write permissions etc
limit_high	char	the high part of segment size
base_high	char	the high part of base address

Table 3-3 Structure of SEGMENT DESCRIPTOR(See 3.4.5[11])

struct GATE_DESCRIPTOR		
Name	Type	Meaning
offset_low	short	the low part of offset
selector	short	which interrupt to choose
dw_count	char	how many interrupts are registered
access_right	char	access permission
offset_high	short	high part of offset

Table 3-4 Structure of GATE DESCRIPTOR

struct MOUSE_DEC		
Name	Type	Meaning
buf[3]	unsigned char	Store the data from mouse
phase	unsigned char	the stage of receiving mouse data
x	int	the x point of mouse
y	int	the y point of mouse
btn	int	whether the mouse is pressed

Table 3-5 Structure of MOUSE DEC

struct FREEINFO		
Name	Type	Meaning
addr	unsigned int	the starting address of free space
size	unsigned int	how many size is free

Table 3-6 Structure of FREEINFO

struct MEMMAN		
Name	Type	Meaning
frees	int	how many memory blocks are free
free[MEMMAN_FREES]	struct FREEINFO	record all free memory block information

Table 3-7 Structure of MEMMAN

struct SHEET		
Name	Type	Meaning
buf	char*	the address of the graphic content depicted
bxszie	int	the size of x coordinate of sheet
bysize	int	the size of y coordinate of sheet
vx0	int	the x coordinate of sheet
vy0	int	the y coordinate of sheet
col_inv	int	the number of invisible color
height	int	the height of sheet
flags	int	the states of sheet, using or not

Table 3-8 Structure of SHEET

struct SHTCTL		
Name	Type	Meaning
vram	unsigned char*	the address of VRAM
map	unsigned char*	which layer the pixel on the screen belongs to
xsize	int	the x size of screen
ysize	int	the y size of screen
top	int	the height of the top layer
sheets[MAX_SHEETS]	struct SHEET*	order all layer addresses in order
sheets0[MAX_SHEETS]	struct SHEET	all layers

Table 3-9 Structure of SHTCTL

p

struct TIMER		
Name	Type	Meaning
next	struct TIMER*	the next timer that is about to timeout
timeout	unsigned int	how long is the timeout
flags	char	the states of timer
flgas2	char	whether to allow automatic cancellation
fifo	struct FIFO32*	store data(from mouse, keyboard etc)
data	int	accept data

Table 3-10 Structure of TIMER

struct TIMER		
Name	Type	Meaning
count	unsigned int	count variable
next	unsigned int	the next timeout timer
t0	sturct TIMER*	the shortest timeout timer
timers0	struct TIMER	all timers

Table 3-11 Structure of TIMERCTL

p

3.3 API

3.4 APPs

struct TSS32		
Name	Meaning	Type
backlink	previous task link	int
esp0 esp1 esp2	stack pointer register	
ss0 ss1 ss2	stack segment register	
cr3	control register	
eip	instruct pointer register	
eflags	registers flag	
eax	accumulator register	
ecx	counter register	
edx	data register	
ebx	base register	
esp	stack pointer register	
ebp	base pointer register	
esi	source index register	
esi	destination index register	
edi	destination index register	
es	extra segment register	
cs	code segment register	
ss	stack segment register	
ds	data segment register	
fs	segment part 2	
gs	segment part 3	
ldtr	LDT segment selector	
iomap	I/O map base address	

Table 3-12 Structure of TSS32(See 6.2.1[11])

4 Implementation

4.1 Kernel

4.1.1 Boot Loader(ipl.asm)

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

1. **Display boot information:** Firstly, the code in boot sector (See Appendix A.1.1) outputs some boot information. When `al=0`, the null character of boot information hit. Interrupt `0x10` is used for showing a character.
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. (See Appendix A.1.2).
3. **Read two sides of a track:**

   If there is a carry indicating some thing went wrong while reading the floppy disk, reset the registers and try reading it again. The read process aborts after five unsuccessful read.

   Register `si` is a counter. If no carry (success), jump to next segment, as one sector has been read into memory already. The address should increase 512 byte. Then sector number (`cl` register) is added by 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.

There is a pseudo code about this process:

```
Result: Read two sides of one track
ENTRANCE: call readloop();
Procedure readloop()
    clear the times of failed to 0, si ← 0;
    call retry();
Procedure retry()
    register parameter preparing;
    read a sector;
    if no carry then
        call next();
    else
        add 1 to si, si ← si + 1;
        compare si with 5;
        if si >= 5 then
            goto error, FINISHED;
        else
            reset registers and call retry() to read again;
        end
    end
Procedure next()
    memory address moved back 0x200;
    add 1 to cl, preparing for reading the next sector, cl ← cl + 1;
    if cl <= 18 then
        call readloop() to read this sector;
    else
        cl > 18, it means that one side of this track is read already;
        add 1 to dh, dh ← dh + 1, reverse the head pointer;
        if dh < 2 then
            it means the 1 side has not read yet, call readloop();
        else
            both sides have finished reading, FINSHED;
        end
    end
```

Algorithm 1: read two sides of one track

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.

The above four steps can be intuitively reflected in the Fig. **??**.

4.2 API

4.3 APPs

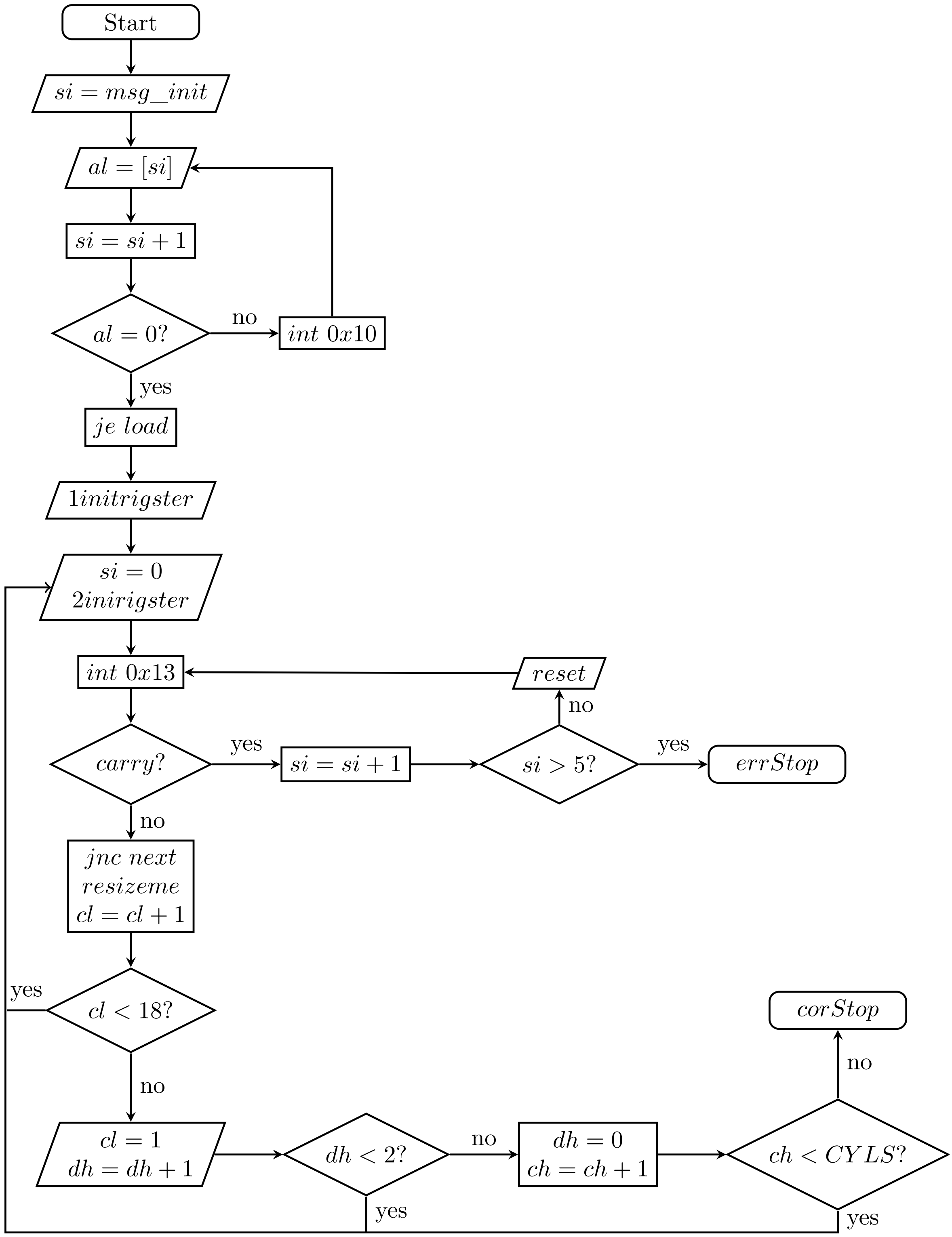

Fig. 4-1 the working flowchart of boot loader

5 Conclusions

What goes in your "Conclusions" chapter? The purpose of this chapter is to provide a summary of the whole thesis or report. In this context, it is similar to the Abstract, except that the Abstract puts roughly equal weight on all thesis/ report chapters, whereas the Conclusions chapter focuses primarily on the findings, conclusions and/or recommendations of the project.

There are a couple of rules –one rigid, one common sense, for this chapter:

- All material presented in this chapter must have appeared already in the report; no new material can be introduced in this chapter. (rigid rule of technical writing)
- Usually, you would not present any new figures or tables in this chapter. (rule of thumb)

Generally, for most technical reports and Masters theses, the Conclusions chapter would be 3 to 5 pages long (double spaced). It would generally be longer in a large PhD thesis. Typically you would have a paragraph or two for each chapter or major subsection. Aim to include the following (typical) content.

1. Re–introduce the project and the need for the work –though more briefly than in the intro;
2. Re–iterate the purpose and specific objectives of your project.
3. Re–cap the approach taken –similar to the road map in the intro; however, in this case, you are re–capping the data, methodology and results as you go.
4. Summarize the major findings and recommendations of your work.
5. Make recommendations for future research.

[0] https://thesistips.wordpress.com/2012/03/25/how-to-write-your-introduction-abstract-and-summa

Bibliography

[1] 国务院, 中国制造 2025, **2015-05**.

[2] G. C. Hunt, J. R. Larus, D. Tarditi, T. Wobber in HotOS, **2005**.

[3] 川合秀实, *30 天自制操作系统*, 1st ed., 人民邮电出版社, **2012-08**.

[4] W. contributors, QEMU — Wikipedia, The Free Encyclopedia, [Online; accessed 12-January-2018], **2017**.

[5] W. contributors, Wine (software) — Wikipedia, The Free Encyclopedia, [Online; accessed 12-January-2018], **2017**.

[6] W. contributors, Instruction set architecture — Wikipedia, The Free Encyclopedia, [Online; accessed 12-January-2018], **2017**.

[7] W. contributors, Processor register — Wikipedia, The Free Encyclopedia, [Online; accessed 12-January-2018], **2018**.

[8] Intel, *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, 1st ed., **2018-03**.

[9] W. contributors, BIOS interrupt call — Wikipedia, The Free Encyclopedia, [Online; accessed 12-January-2018], **2017**.

[10] W. contributors, Floppy disk — Wikipedia, The Free Encyclopedia, [Online; accessed 12-January-2018], **2018**.

[11] Intel, *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, 1st ed., **2006-10**.

Supervisor

Xiaolin WANG (Mr.), 49 years old, got his MSc degree at University of Greenwich in UK. Currently he's been working as a lecturer at the School of Big Data and Intelligence Engineering, Southwest Forestry University in China, teaching Linux, Operating Systems, and Computer Networking.

Acknowledgments

I would like to thank my supervisor Mr. WANG Xiaolin for his continuous support of my four years undergraduate study. I am extremly thankful to him for sharing expertise, and sincere and valuable guidance and encouragement extended to me.

What I most want to thank is my girlfriend. She tolerated me when I finished this graduation project many nights did not accompany her, gave me support, encouraged me, and did not complain. So I would like to name this simple operating system as RongOS. Rong is the last word of her name. Thank you, my dearest.

My special thanks to a great company - Google, I think I need to thank you in this very formal place in my graduation thesis. Every time you gave me a lot of help, the knowledge and other abilities I learned from you will have a profound impact on my future life. I am grateful for every search, because I know you will give me the results I want. Without you, this paper cannot be completed. Thank you.

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
```