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

This section will introduce the purpose and current status of the operating system research. The setup of the development environment will also be presented here.

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 Design

In this section will introduce the design of the entire system including the kernel, API, and applications.

2.1 Top Level Design

All applications use the functions provided by the operating system kernel through API calls. This facilitates the application's ability to call the operating system. The overall system architecture is as 2-1 shown:

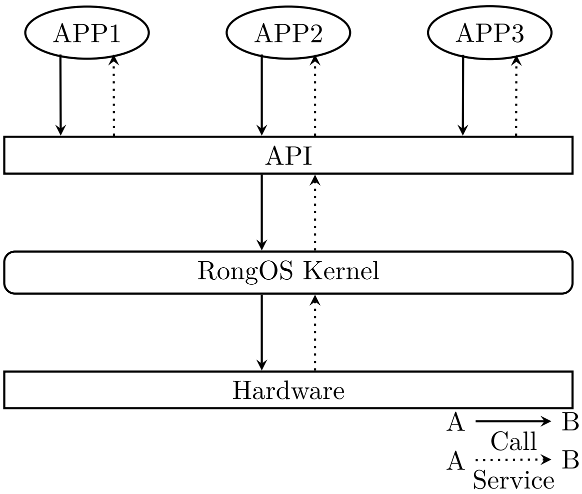

Fig. 2-1 Top-level design

2.2 Detailed Design

2.2.1 Kernel

The kernel receives the API call in the upward direction and the kernel requests the hardware service through the driver in the downward direction.

1. Module Relationship

Fig. 2-2 shows how the various modules in the kernel are related. `bootpack` completes startup-related settings such as keyboard, PIC, GDT/IDT and mouse settings. `ipl` loads the entire operating system into memory. `asmhead` completes the switch to 32-bit mode and calls the C function. `naskfunc` is used to provide functions that the C language cannot do and thus requires assembly. `PIC`, `keyboard` and `mouse` is used to complete hardware-related initialization. `console` is used to accept command line arguments and run various commands related to the application. `graph` is used to depict the mouse, graphics etc. `window` for making windows. `sheet` is used to control layers, such as layer height settings etc. `memory` for managing memory. `task` is used to manage multiple tasks, such as task switching, scheduling. `timer` for managing time slices. `fifo` is used to manage FIFO buffers that are used to accept various data. `dsctbl` for GDT/IDT setting. `file` is used to manage file-related operations such as reading, loading, and searching for files.

2. Data Structure in Kernel

The following describes the data structure used in the RongOS operating system.

1. NAME

    `BOOTINFO`

    SYNOPSIS

    The structure `BOOTINFO` stores startup-related information, such as how many cylinders were read, the status of the keyboard indicator, the mode of the screen, the size of the screen, and the memory address of the graphics card.

    MEMBER

    char cyls

    number of cylinders to read

    char leds

    keyboard state at boot

    char vmode

    bits of color of graphics card

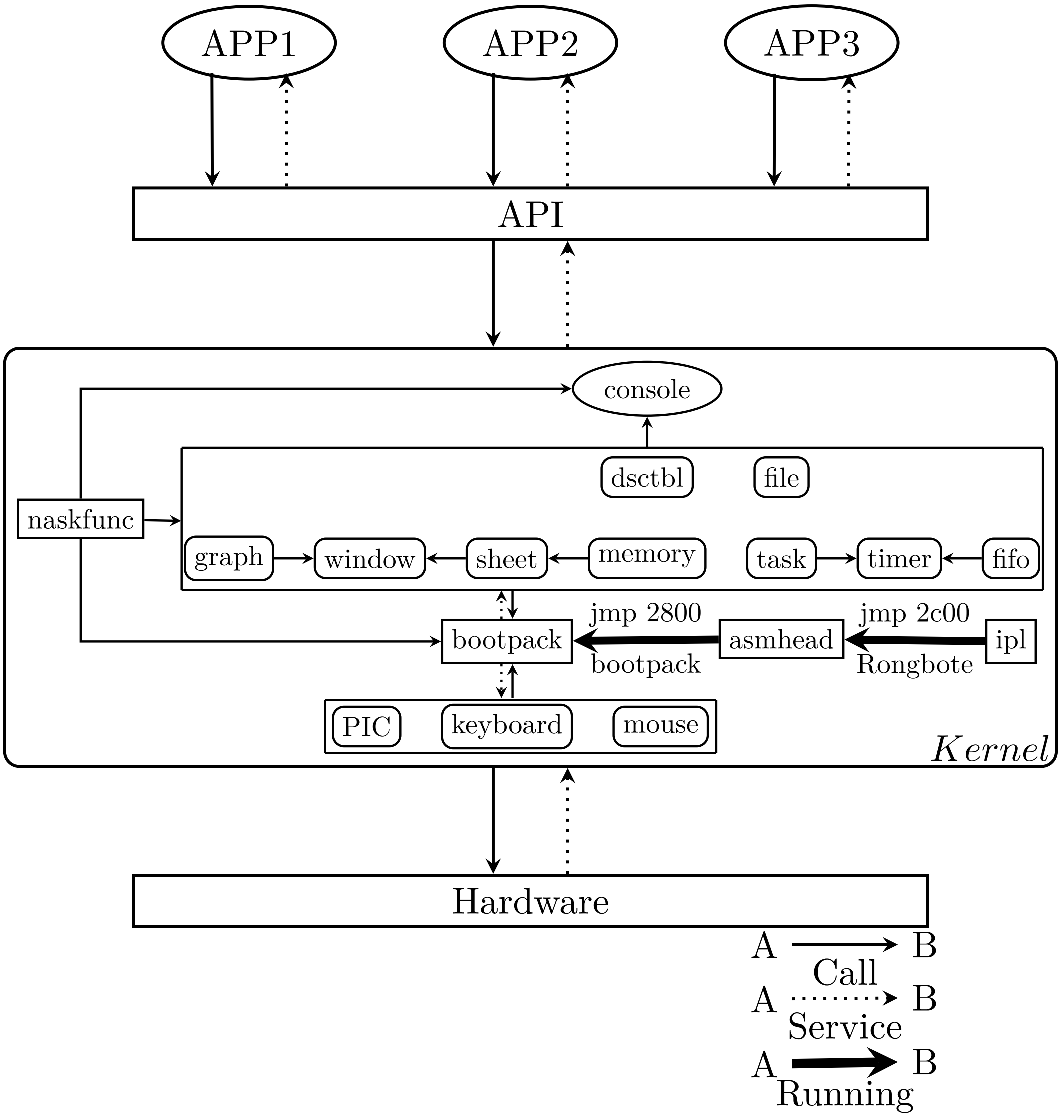

Fig. 2-2 modules in kernel

char reserve

reserved bytes

short scrnx

screen resolution of x

short scrny

screen resolution of y

char* vram

the starting address of the image buffer

2. NAME

`FIFO32`

SYNOPSIS

`FIFO32` is used to describe a FIFO structure. This structure is used to receive various kinds of information. `FIFO32` is used to describe a FIFO structure. This structure is used to receive various kinds of information. It specifies where to read and write the FIFO structure and the size of the buffer, available size.

MEMBER

int* buf

the address of FIFO32 buffer

int p

the writing address

int q

the reading address

int size

the size of FIFO32 buffer

int free

how many space free

int flags

the states of FIFO32 buffer

struct TASK* task

point to a task

3. NAME

    `SEGMENT_DESCRIPTOR`

    SYNOPSIS

    `SEGMENT_DESCRIPTOR` structure is used to store GDT related information, which is based on CPU specifications(3.5.1 and 3.4.5[6]). GDT is stored at 270000 in memory.

    MEMBER

    short limit_low

    the low part of segment size

    short base_low

    the low part of base address

    char base_mid

    the middle part of base address

    char access_right

    read and write permissions etc

    char limit_high

    the high part of segment size

    char base_high

    the high part of base address

4. NAME

    `GATE_DESCRIPTOR`

    SYNOPSIS

    `GATE_DESCRIPTOR` structure is used to store IDT related information, which is based on CPU specifications(3.5.1 and 3.4.5[6]). IDT is at 26f800 memory.

    MEMBER

    short offset_low

    the low part of offset

short selector

which interrupt to choose

char dw_count

how many interrupts are registered

char access_right

access permission

short offset_high

high part of offset

5. NAME

   `MOUSE_DEC`

   SYNOPSIS

   `MOUSE_DEC` structure is used to store information about the mouse, such as the location of the mouse, whether the mouse is pressed or not.

   MEMBER

   unsigned char buf[3]

   store the data from mouse

   unsigned char phase

   the stage of receiving mouse data

   int x

   the x point of mouse

   int y

   the y point of mouse

   int btn

   whether the mouse is pressed

6. NAME

   `FREEINFO`

   SYNOPSIS

   `FREEINFO` structure stores how many bytes are free from where in memory.

MEMBER

unsigned int addr

the starting address of free space

unsigned int size

how many size is free

7. NAME

`MEMMAN`

SYNOPSIS

`MEMMAN` structure is used to store the entire memory usage, such as the total remaining memory space and entries.

MEMBER

int frees

how many memory blocks are free

int maxfrees

the maximum of frees

int lostsize

release the sum of the failed memory size

int losts

the number of failures

struct FREEINFO free[MEMMAN_FREES]

record all free memory block information

8. NAME

`SHEET`

SYNOPSIS

`SHEET` structure is used to record the position, usage, and color of a layer.

MEMBER

char* buf

the address of the graphic content depicted int bxszie

the size of x coordinate of sheet

int bysize

the size of y coordinate of sheet

int vx0

the x coordinate of sheet

int vy0

the y coordinate of sheet

int col_inv

the number of invisible color

int height

the height of sheet

int flags

the states of sheet, using or not

9. NAME

`SHTCTL`

SYNOPSIS

`SHTCTL` structure is used to manage the structure of multiple layer information, including how many layers there are in total, the size and height of each layer.

MEMBER

unsigned char* vram

the address of VRAM

unsigned char* map

which layer the pixel on the screen belongs to

int xsize

the x size of screen

int ysize

the y size of screen

int top

the height of the top layer

struct SHEET* sheets[MAX_SHEETS]

order all layer addresses in order

struct SHEET sheets0[MAX_SHEETS]

all layers

10. NAME

`TIMER`

SYNOPSIS

`TIMER` structure is used to manage the time slice of the CPU. The timer interrupts the CPU at regular intervals. This structure records the length of the timer, usage status and other information.

MEMBER

struct TIMER* next

the next timer that is about to timeout

unsigned int timeout

how long is the timeout

char flags

the states of timer

char flgas2

whether to allow automatic cancellation

struct FIFO32* fifo

store data(from mouse, keyboard etc)

int data

accept data

11. NAME

`TIMERCTL`

SYNOPSIS

`TIMERCTL` structure is used to manage all timers in the system. Including how many timers are in total, current use, and the next timer to be used.

MEMBER

unsigned int count

count variable

unsigned int next

the next timeout timer

sturct TIMER* t0

the shortest timeout timer

struct TIMER timers0

all timers

12. NAME

`TSS32`

SYNOPSIS

`TSS32` structure holds information about task status segments, which are based on CPU specifications(See 6.2.1[6]).

MEMBER

int esp0, esp1, esp2

stack pointer register

int ss0, ss1, ss2

stack segment register

int cr3

control register

int eip

instruct pointer register

int eflags

registers flag

int eax

accumulator register

int ecx

counter register

int edx

data register

int ebx

base register

int esp

stack pointer register

int ebp

base pointer register

int esi

source index register

int edi

destination index register

int es

extra segment register

int cs

code segment register

int ss

stack segment register

int ds

data segment register

int fs

segment part 2

int gs

segment part 3

int ldtr

LDT segment selector

int iomap

I/O map base address

13. NAME

SYNOPSIS

MEMBER

14. NAME

SYNOPSIS

MEMBER

The tables 2-1 — 2-18 describes the data structure used by RangOS in tabular form.

As shown in 2-1, the structure `BOOTINFO` stores startup-related information, such as how many cylinders were read, the status of the keyboard indicator, the mode of the screen, the size of the screen, and the memory address of the graphics card.

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 2-1 Structure of BOOTINFO

As shown in 2-2, `FIFO32` is used to describe a FIFO structure. This structure is used to receive various kinds of information. `FIFO32` is used to describe a FIFO structure. This structure is used to receive various kinds of information. It specifies where to read and write the FIFO structure and the size of the buffer, available size.

As shown in 2-3, the `SEGMENT_DESCRIPTOR` structure is used to store GDT related information, which is based on CPU specifications(3.5.1 and 3.4.5[6]). GDT is stored at 270000 in memory.

As shown in 2-4, the `GATE_DESCRIPTOR` structure is used to store IDT related infor-

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 2-2 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 2-3 Structure of SEGMENT DESCRIPTOR(See 3.4.5[6])

mation, which is based on CPU specifications(3.5.1 and 3.4.5[6]). IDT is at 26f800 memory.

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 2-4 Structure of GATE DESCRIPTOR

As shown in 2-5, the `MOUSE_DEC` structure is used to store information about the mouse, such as the location of the mouse, whether the mouse is pressed or not.

As shown in 2-6, the `FREEINFO` structure stores how many bytes are free from where in memory.

As shown in 2-7, the `MEMMAN` structure is used to store the entire memory usage, such as the total remaining memory space and entries.

As shown in 2-8, the `SHEET` structure is used to record the position, usage, and color of

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 2-5 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 2-6 Structure of FREEINFO

struct MEMMAN		
Name	Type	Meaning
frees	int	how many memory blocks are free
maxfrees	int	the maximum of frees
lostsize	int	release the sum of the failed memory sizea
losts	int	the number of failures
free[MEMMAN_FREES]	struct FREEINFO	record all free memory block information

Table 2-7 Structure of MEMMAN

a layer.

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 2-8 Structure of SHEET

As shown in 2-9, the `SHTCTL` structure is used to manage the structure of multiple layer information, including how many layers there are in total, the size and height of each layer.

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 2-9 Structure of SHTCTL

As shown in 2-10, the `TIMER` structure is used to manage the time slice of the CPU. The timer interrupts the CPU at regular intervals. This structure records the length of the timer, usage status and other information.

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 2-10 Structure of TIMER

As shown in 2-11, the `TIMERCTL` structure is used to manage all timers in the system. Including how many timers are in total, current use, and the next timer to be used.

struct TIMERCTL		
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 2-11 Structure of TIMERCTL

As shown in 2-12, the `TSS32` structure holds information about task status segments, which are based on CPU specifications(See 6.2.1[6]).

<table>
<tr><th colspan="3">struct TSS32</th></tr>
<tr><th>Name</th><th>Meaning</th><th>Type</th></tr>
<tr><td>backlink</td><td>previous task link</td><td rowspan="24">int</td></tr>
<tr><td>esp0
esp1
esp2</td><td>stack pointer register</td></tr>
<tr><td>ss0
ss1
ss2</td><td>stack segment register</td></tr>
<tr><td>cr3</td><td>control register</td></tr>
<tr><td>eip</td><td>instruct pointer register</td></tr>
<tr><td>eflags</td><td>registers flag</td></tr>
<tr><td>eax</td><td>accumulator register</td></tr>
<tr><td>ecx</td><td>counter register</td></tr>
<tr><td>edx</td><td>data register</td></tr>
<tr><td>ebx</td><td>base register</td></tr>
<tr><td>esp</td><td>stack pointer register</td></tr>
<tr><td>ebp</td><td>base pointer register</td></tr>
<tr><td>esi</td><td>source index register</td></tr>
<tr><td>esi</td><td>destination index register</td></tr>
<tr><td>edi</td><td>destination index register</td></tr>
<tr><td>es</td><td>extra segment register</td></tr>
<tr><td>cs</td><td>code segment register</td></tr>
<tr><td>ss</td><td>stack segment register</td></tr>
<tr><td>ds</td><td>data segment register</td></tr>
<tr><td>fs</td><td>segment part 2</td></tr>
<tr><td>gs</td><td>segment part 3</td></tr>
<tr><td>ldtr</td><td>LDT segment selector</td></tr>
<tr><td>iomap</td><td>I/O map base address</td></tr>
</table>

Table 2-12 Structure of TSS32(See 6.2.1[6])

struct FILEHANDLE		
Name	Type	Meaning
buf	char*	store the handler of file
size	int	the size of file
pos	int	where to read the file

Table 2-13 Structure of FILEHANDLE

struct TASK		
Name	Type	Meaning
sel	int	the number of GDT
flags	int	the state of task
level	int	the level of task
priority	int	the priority of task
fifo	struct FIFO32	a fifo buffer
tss	TSS32	TSS segment for a task
cons	struct CONSOLE*	the console window address of task
ds_base	int	data segment address of APPs
cons_stack	int	the stack address of APPs
ldt[2]	struct SEGMENT_DESCRIPTOR	tow LDT segments of task
fhandle	struct FILEHANDLE*	file handles for manipulating files
fat	int*	file allocation table
cmdline	char*	store the command line context
langmode	unsigned char	which font to use
langbyte1	unsigned char	store the first byte of the full-width character

Table 2-14 Structure of TASK

struct TASKLEVEL		
Name	Type	Meaning
running	int	how many tasks are running
now	int	which task is currently running
tasks[MAX_TASKS_LV]	struct TASK*	all tasks in one level

Table 2-15 Structure of TASKLEVEL

struct TASKCTL		
Name	Type	Meaning
now_lv	int	current activity level
lv_change	int	does the hierarchy need to be changed next time the task is switched

Table 2-16 Structure of TASKCTL

struct CONSOLE		
Name	Type	Meaning
sht	struct SHEET*	Which layer is used on the command line
cur_x	int	the x position of console
cur_y	int	the y position of console
cur_c	int	the color of console
timer	struct TIMER*	timer to control cursor blinking

Table 2-17 Structure of CONSOLE

struct FILEINFO		
Name	Type	Meaning
name[8]	unsigned char	file name
ext[3]	unsigned char	extend name of file
type	unsigned char	file attributes
char	reserve[10]	reserve byte
time	unsigned short	the time for storing file
date	unsigned short	the date for storing file
clustno	unsigned short	the file from which sector on the disk is stored

Table 2-18 Structure of FILEINFO

2.2.2 API

2.2.3 APPs

3 Implementation

3.1 Kernel

3.1.1 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. 3-1.

3.2 API

3.3 APPs

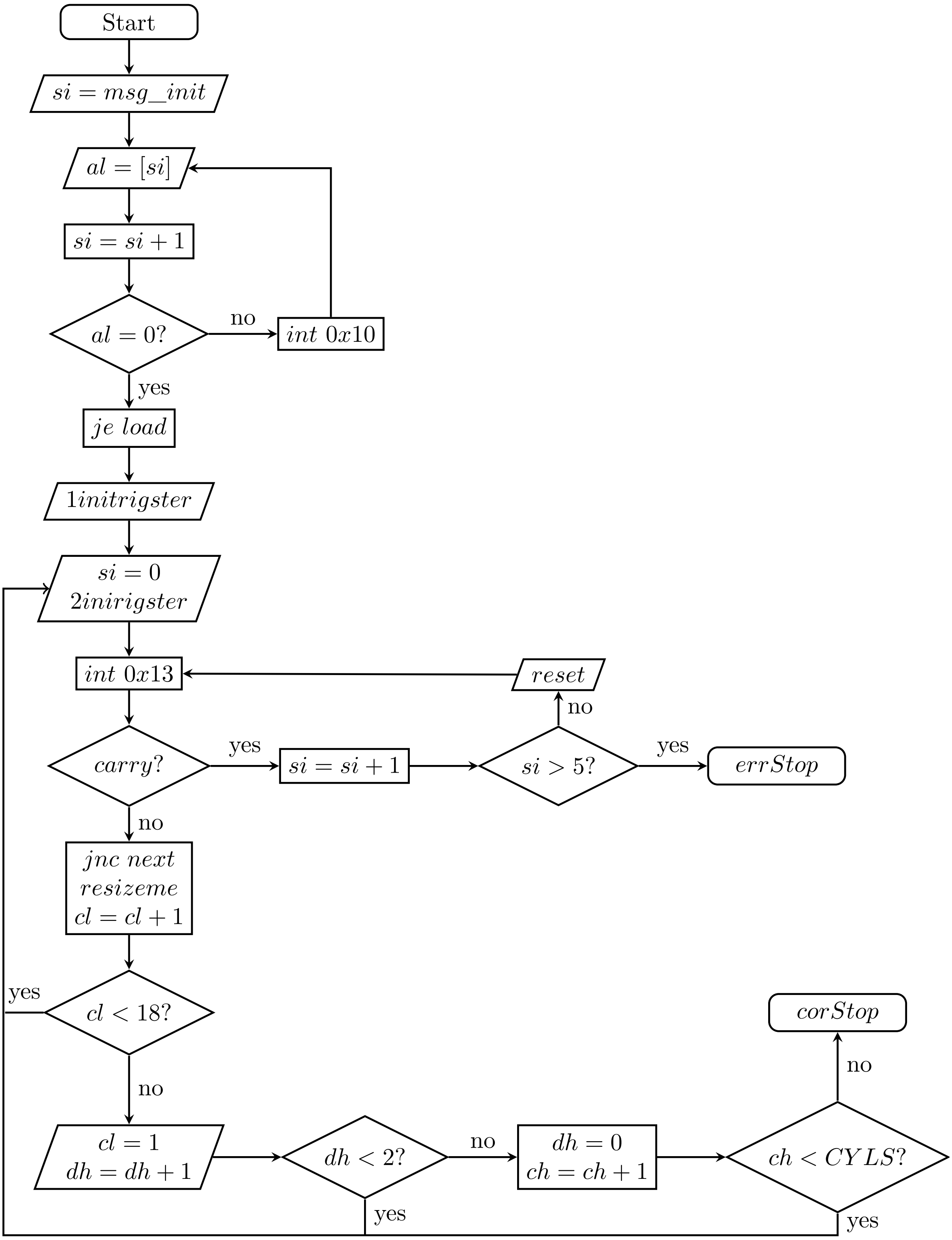

Fig. 3-1 the working flowchart of boot loader

4 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-summary/

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] 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
```