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. The process control subsystem includes graphics processing, scheduler, and memory management. This system will interact with the file system. For example, to launch an application, the function in the file system must be used to search for related applications.Of course, the file system part will provide a file search service. All of these subsystems may interact with the driver or hardware control part. Various functions in the kernel are packaged into system call application programs. However, applications do not use these system calls directly, but use the API. The process in which the API requests a system call and hands over processing to the kernel is called trapping.

2.2 Detailed Design

This section will introduce the detailed design of the entire operating system. Including the function of each module, data structure in boot loader, kernel, API, APPs.

2.2.1 Boot Up

At this stage the boot loader will load the operating system into memory. It is divided into the following four steps: 1, display boot information. 2, read the second sector. 3, read two sides of a track. 4, the next cylinder. Until all twenty cylinders have been read.

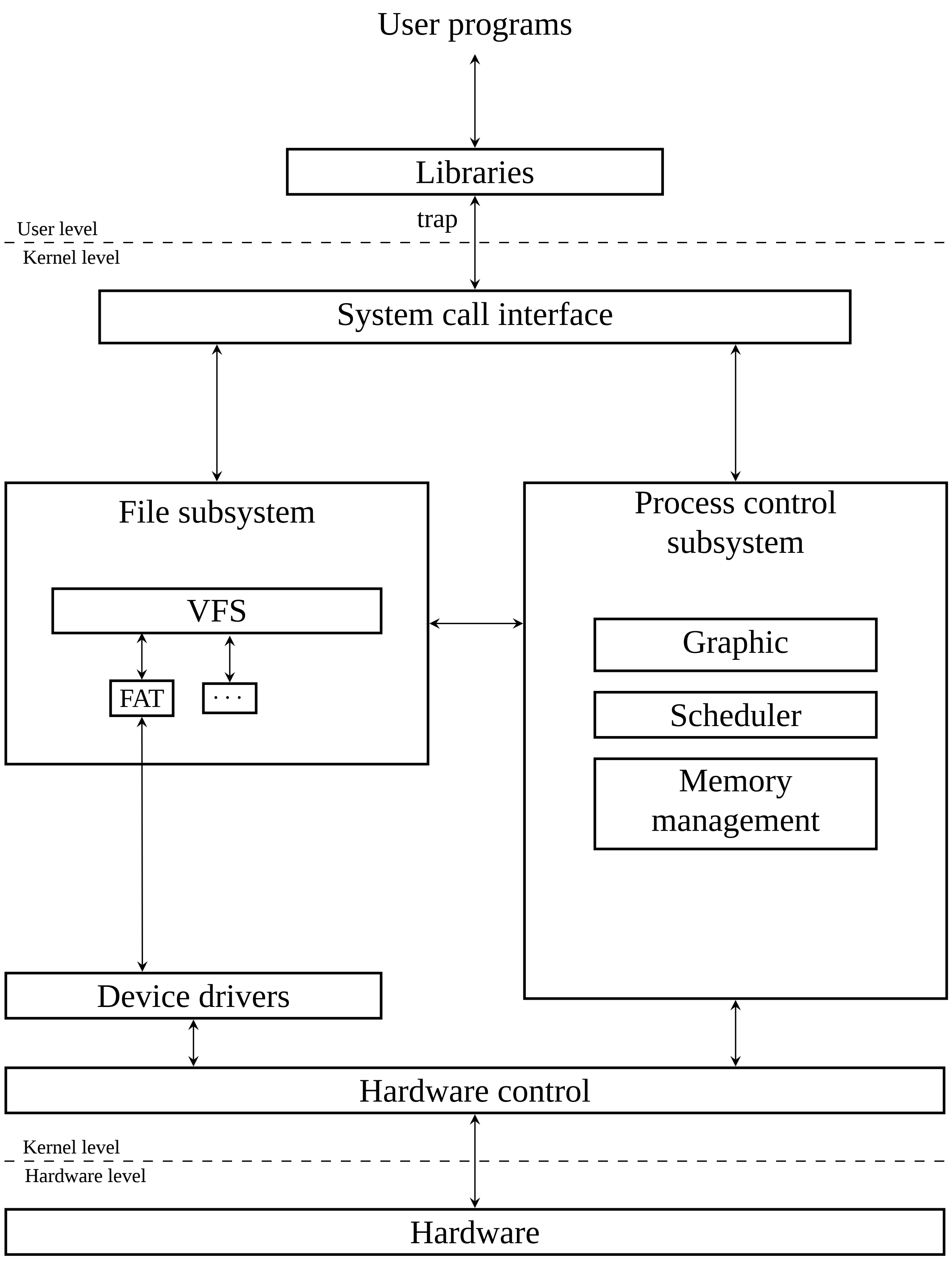

Fig. 2-1 Top-level design

At this stage, it is also necessary to complete the 32-bit protection mode switch and jump to the entry point of the operating system.

2.2.2 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.

Memory Management

The process requires memory space at run time. When it is finished running, it will return space to the operating system. So the operating system should be able to manage memory. Memory utilization should be improved when managing memory.

Memory Management Data Structure

```c
struct FREEINFO
{
  unsigned int addr; /* the starting address of free space
   ↪  */
  unsigned int size; /* how many size is free */
};
```

Code 2-1 `struct FREEINFO`

FREEINFO `struct FREEINFO` (Code 2-1) structure is used to record the size in bytes of free memory while the system is running.

```
struct MEMMAN
{
  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 */
};
```

Code 2-2 `struct MEMMAN`

MEMMAN `struct MEMMAN` (code 2-2) structure is used to store the entire memory usage, such as the total remaining memory space and entries.

Memory Management Functions

`void memman_init(struct MEMMAN* man);` The memory space is initialized and the `man` variable is used to record the memory space usage and free information.

`unsigned int memman_total(struct MEMMAN* man);` Report the sum of all empty space. Return the size of free memory.

`unsigned int memman_alloc(struct MEMMAN *man, unsigned int size);` Allocate memory to the application, where `size` is the space requested by the application. Success returns the available starting address, otherwise 0.

`int memman_free(struct MEMMAN *man, unsigned int addr, unsigned int size);` Releases memory, where `addr` is the starting address variable and `size` is the size of the release variable. Returns 0 if successful, otherwise -1.

`unsigned int memman_alloc_4k(struct MEMMAN *man, unsigned int size);` Memory is allocated in 4k memory units and the starting address of the allocated memory is returned. `size` is the size of requested space.

`int memman_free_4k(struct MEMMAN *man, unsigned int addr, unsigned int size);` Memory space is freed in units of 4k, `addr` is the starting point variable, and `size` is the release size.

Task Management(Scheduler)

The general operating system needs to be able to support multitasking. Simply saying that multi-tasking is running multiple programs at the same time. But this only caused the user an illusion. For a single CPU, it does not implement a real sense of running programs at the same time. It merely divides the running time into many small pieces for different programs to run. The operating system should be able to return to the original program when switching tasks. Therefore, the register value should be saved when the task is switched. The task switching itself will also take time. The operating system should try to shorten this time. Doing this on the one hand provides the user with a good experience and does not leave the user with a delayed impression. On the other hand, reducing this time can improve CPU utilization. Because this time can not be used for program operation.

Task Management Data Structure

```c
struct TASKLEVEL
{
  int running;                        /* how many tasks are
   ↪  running */
  int now;                            /* which task is
   ↪  currently running */
  struct TASK* tasks[MAX_TASKS_LV]; /* all tasks in one
   ↪  level */
};
```

Code 2-3 `struct TASKLEVEL`

TASKLEVEL `struct TASKLEVEL` (Code 2-10) is used to record the status of each task in a layer.

```c
struct TSS32
{
  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 */
};
```

Code 2-4 `struct TSS32`

TSS32 `struct TSS32` (Code 2-4) structure holds information about task status segments, which are based on CPU specifications(See 6.2.1[6]).

```c
struct
{
  int now_lv;                               /* current
   ↪  activity level */
  int lv_change;                            /* does the
   ↪  hierarchy need to be changed next time the task is
   ↪  switched
  *                                         /
  struct TASKLEVEL level[MAX_TASKLEVELS]; /*all levels*/
  struct TASK tasks0[MAX_TASKS];            /* all running
   ↪  program */

};
```

Code 2-5 `struct TASKCTL`

TASKCTL `struct` (Code 2-5) is used to control all tasks in the system.

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

Code 2-6 `struct TASK`

TASK `struct TASK` (Code 2-6) is used to manage variables for a task. Record the task's sections, permissions, stacks, etc.

Task Management Functions

`struct TASK *task_now(void);` eturns which level of which task is currently running.

`struct TASK *task_init(struct MEMMAN *memman);` initialize a task.

`struct TASK *task_alloc(void);` initialize the structure of a task.

`void task_run(struct TASK *task, int level, int priority);` add a task to list.

`void task_switch(void);` switch to the next task.

`void task_sleep(struct TASK *task);` put a task to sleep.

Graphic Management

The patterns on the screen belong to one layer. The movement of the window is achieved through layers. The height of the layer affects the layout of the screen. Layers to consider how to cover when moving, which one is on and below.

Graphic Management Data Structure

```c
struct SHEET
{
  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 */
};
```

Code 2-7 `struct SHEET`

SHEET `struct SHEET` (Code 2-7) is used to record layer-related information, including the layer’s size and position.

```
struct SHTCTL
{
  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 */
};
```

Code 2-8 `struct SHTCTL`

SHTCTL `struct SHTCTL` (Code 2-8) 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.

Graphic Management Functions

`struct SHTCTL *shtctl_init(struct MEMMAN *memman,unsigned char *vram,int xsize,int ysize)`
initialize a sheet control structure, `vram` is the address of video RAM. `xsize` and `ysize` is the size of sheet.

`struct SHEET *sheet_alloc(struct SHTCTL *ctl);` return a SHEET structure if success, otherwise 0.

`void sheet_setbuf(struct SHEET *sht,unsigned char *buf,int xsize,int ysize,int col_inv);`
set the properties of the layer `buf`.

`void sheet_updown(struct SHEET *sht, int height);` set the height of layer `sht`.

`void sheet_refresh(struct SHEET *sht, int bx0, int by0, int bx1, int by1);` refreshes the screen range specified by `bx0`, `by0`, `bx1` and `by1`.

`void sheet_free(struct SHEET *sht);` release layer.

`void make_window8(unsigned char *buf, int xsize, int ysize, char *title, char act);`
make a window in SHEET `buf`.

`void putfonts8_asc_sht(struct SHEET *sht, int x, int y, int c, int b, char *s, int l);`
paint the background color and write the characters and finish the refresh.

System Call

The system call encapsulates each module in the kernel. These system calls are part of the kernel. In this system, system calls are called by the API instead of the application.

System Calls Prototype

`void cons_putchar(struct CONSOLE *cons, int chr, char move);` put a `chr` character on `cons` console.

`void cons_newline(struct CONSOLE *cons);` newline in `cons` console.

`void cons_putstr0(struct CONSOLE *cons, char *s);` put string `s` in `cons` console, no length.

`void cons_putstr1(struct CONSOLE *cons, char *s, int l);` put string `s` in `cons`, `l` is the length of string `s`.

`int cmd_app(struct CONSOLE *cons, int *fat, char *cmdline);` Start an application based on the input from the command line `cmdline`;

Driver

The driver is used to control and manipulate the hardware. The driver blocked the hardware information. It is the dividing line between hardware and software. A driver provides a software interface to hardware devices, enabling operating systems and other computer programs to access hardware functions without needing to know precise details of the hardware being used.

Data Structure

```
struct
{

};
```

Code 2-9 `struct`

`struct` (Code 2-10)

Functions

Data Structure

```
struct
{

};
```

Code 2-10 `struct`

`struct` (Code 2-10)

Functions

2.2.3 API

2.2.4 APPs

3 Implementation

3.1 Boot Up

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:
4. **The next cylinder:** So the next step is moving a cylinder, add 1 to register `ch`. Oth-

```
Result: Read two sides of one track
ENTRANCE: call readloop();
Procedure readloop()
    clear the times of failed to 0, $si \leftarrow 0$;
    call retry();
Procedure retry()
    register parameter preparing;
    read a sector;
    if no carry then
        call next();
    else
        add 1 to si, $si \leftarrow si + 1$;
        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, $cl \leftarrow cl + 1$;
    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, $dh \leftarrow dh + 1$, 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

erwise 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 Kernel

3.3 API

3.4 APPs

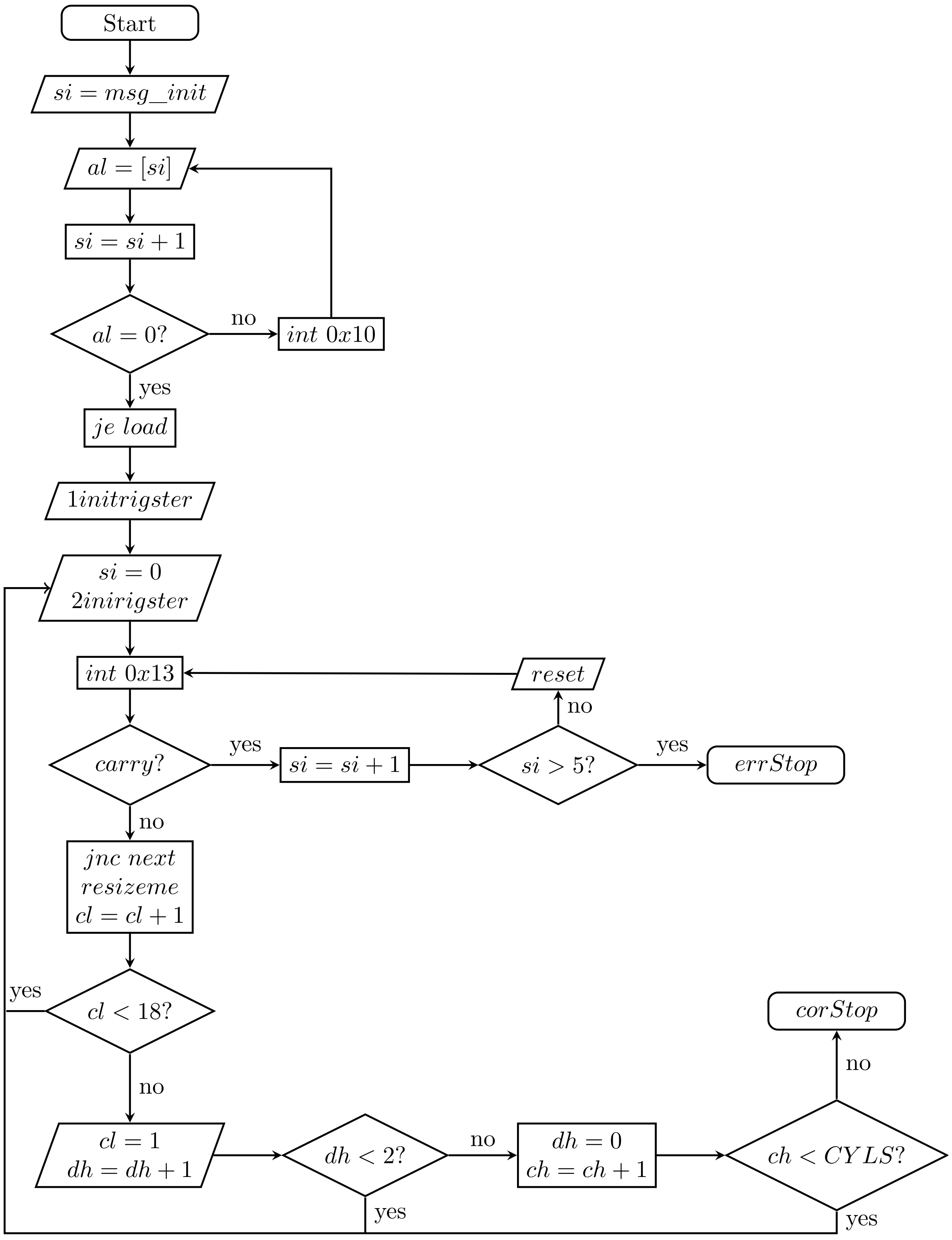

Fig. 3-1 the working flowchart of boot loader

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
```