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

List of Corrections

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

This chapter describes the design issues in the entire software system, including system startup, the kernel, APIs, and some applications.

2.1 Top Level Design

As shown in Fig. 2-1 shown, Fixme: correct picture, all applications require services from the operating system through a set of function calls. This set of functions provided by the OS kernel is usually called *system calls*. However, the user applications do not invoke these system calls directly. Instead, they call the API library functions. The process in which the API invokes a system call and hands over processing to the kernel is called *trapping*. Fixme!

Within the kernel, there are some important software modules working to keep the system running well. The process control subsystem provides graphics processing, CPU scheduling, and memory management functions for processes. Fixme: do you have VFS? The file subsystem provides a friendly way to the user processes for accessing disk data. For example, to launch an application program stored on the disk, the function in the file system must be invoked to find the specific application. All of these subsystems may interact with the driver or hardware control module so as to operate on the system hardwares. Fixme!

2.2 Detailed Design

This section discusses the system design in details, including the function of each module, data structures used in boot loader, kernel, API, and APPs.

2.2.1 Boot Up

At this stage the boot loader loads the operating system kernel into memory. This is done in the following four steps:

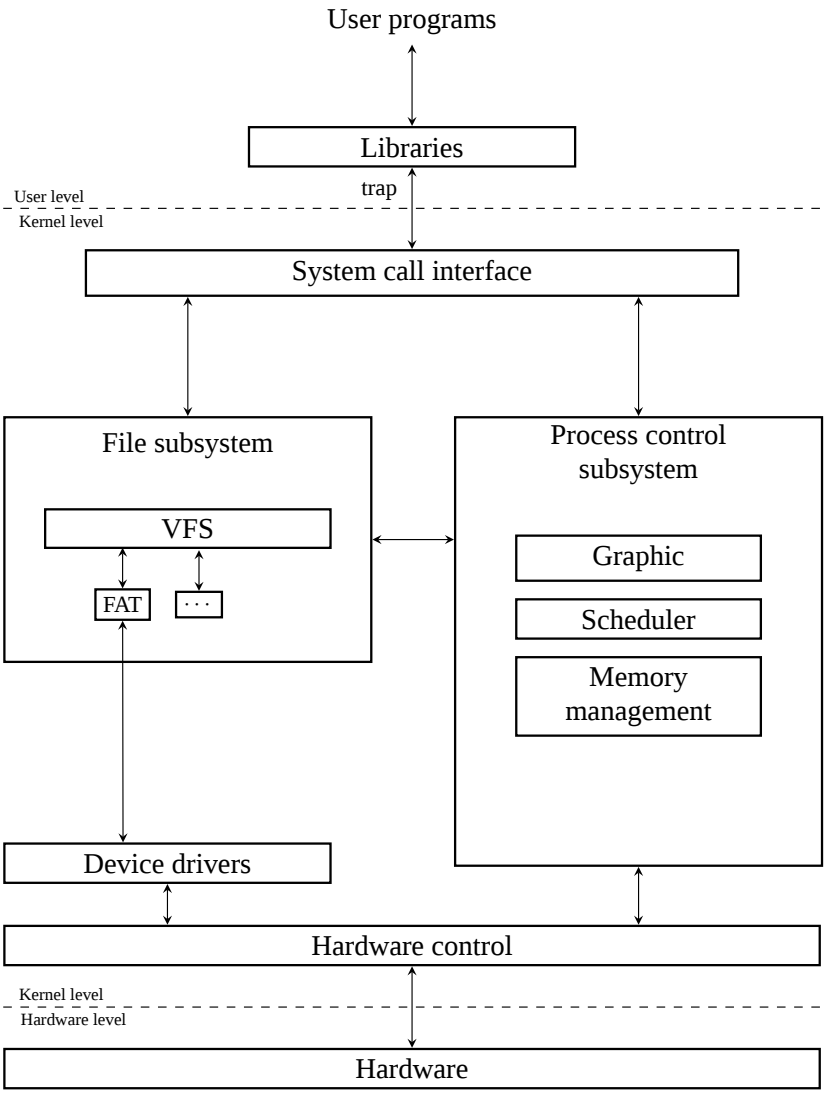

Fig. 2-1 Top-level design

1. Display boot information;
2. Read the second sector;
3. Read two sides of a track;
4. Read the next cylinder until all twenty cylinders have been read.

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 calls from user processes, and operates the hardwares through the device drivers. Fixme: say some more.

Fixme!

Memory Management

Memory management refers to the technology that allocates and uses computer memory resources while the software is running. Its main purpose is how to allocate efficiently and quickly, and release for reusing memory resources when appropriate. This is critical to any advanced computer system where more than a single process might be underway at any time.

Several methods have been invented to improve the effectiveness of memory management. For modern operating systems, memory virtualization technology is used. This technique separates the memory space used by the process from the actual physical memory. Processes that are temporarily not running will be moved to secondary storage through paging or swapping. How virtual memory is designed and how it is implemented will have a wide impact on the overall system performance.

For RongOS, table 2-1 reflects the memory management system. An *entry* is a record

Start Address	Free Size
1	301
348	50
800	1000
3001	...

Table 2-1 memory management table

reflects how much memory space is free from where. Actually, this table is an member

`struct FREEINFO` in `struct MEMMAN`.

Data Structures used in MM

```
struct FREEINFO
{
  unsigned int addr; /* start address of free space */
  unsigned int size; /* size of free mem in bytes*/
};
```

- is used to record the size in bytes of free memory while the system is running.

```
struct MEMMAN
{
  int frees;    /* free memory blocks */
  int maxfrees;
  int lostsize; /* the sum of the failed memory size to freed*/
  int losts;    /* number of failures */
  struct FREEINFO free[MEMMAN_FREES]; /* free memory block info */
};
```

- is used to record the entire memory usage, such as the total remaining free memory space and *entry*.

Memory Management Functions

```
unsigned int memtest(unsigned int start, unsigned int end);
```

- Check the memory capacity and confirm that the memory is intact. `start` is the start address of the memory check and `end` is the end address of the memory check. This function returns the maximum value of available addresses. Memory is available between this value and the `start` value.

```
unsigned int memman_total(struct MEMMAN* man);
```

- Report the sum of all empty space. It will look for each *entry* in the `man` to add up the free space in it.

```
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. `man` records information about all available entries in memory.

```
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);
```

- Basically it is similar to the `memman_alloc` function, but memory is allocated in 4k memory units and the starting address of the allocated memory is returned if successful. `size` is the size of requested space.

```
int memman_free_4k(struct MEMMAN *man, unsigned int addr, unsigned int
↪ size);
```

- Basically it is similar to the `memman_free`, but 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 systems need to be able to support multitasking. Simply saying that multi-tasking is running multiple programs at the same time. But this only gives the user an illusion. For a single CPU, it cannot handle multiple programs at the same time. It merely divides the CPU time into many small pieces for different programs to run. The operating system should be able to do task switching, that is to pass the CPU from one process to another. And at a later time to switch it back, as shown in Fig. 2-2. Therefore, the register values should be saved when the task is switched. The task switching itself also takes 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. Fixme: see wikipedia

Fixme!

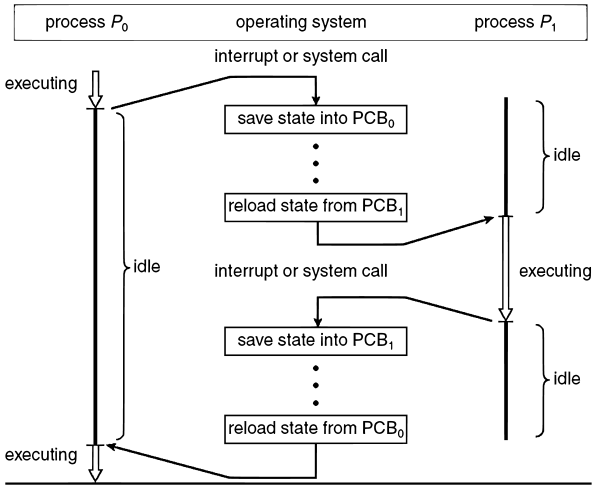

Fig. 2-2 Context switch

Data Structures For Task Management

```
1  struct TASKLEVEL
2  {
3    int running; /* how many tasks are running */
4    int now;     /* which task is currently running */
5    struct TASK* tasks[MAX_TASKS_LV]; /* all tasks in one level */
6  };
```

- is used to record the status of each task in a layer Fixme: What's a layer/level?.

Fixme!

```
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 */
};
```

- holds information about task status segments, which are based on CPU specifications[6]. Fixme: more about these registers Fixme!

```
struct TASKCTL
{
  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 */
};
```

- is used to control all tasks in the system. Fixme: not clear. the memory manage structure Fixme!

```
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 */
};
```

- is used to manage variables for a task. Record the task's sections, permissions, stacks, etc.

Fixme!

Task Management Functions Fixme: more about each function

```
struct TASK *task_now(void);
```

- Returns 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 Fixme: ???. 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. Fixme: bad Fixme! Fixme!

Graphic Management Data Structures

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

- 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 */
};
```

- 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.

Fixme!

Graphic Management Functions Fixme: more about each function

```
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 Calls

The system calls encapsulate each module in the kernel Fixme: ???. These system calls are part of the kernel. In this system, system calls are called by the API instead of the application. Fixme: see wikipedia

Fixme!

Fixme!

Fixme!

System Calls Prototype Fixme: more about each function

```
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 device drivers are used to control and manipulate the hardware. The driver blocked the hardware information Fixme: wikipedia. 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.

Fixme!

Fixme!

Driver Functions Fixme: more about each function

```
void set_palette(int start, int end, unsigned char *rgb);
```

- initialize the palette.

```
void init_pic(void);
```

- initialize the PIC.

```
void init_keyboard(struct FIFO32 *fifo, int data0);
```

- initialze the keyboard.

```
void enable_mouse(struct FIFO32 *fifo, int data0, struct MOUSE_DEC *mdec);
```

- activate mouse.

FAT File System

Fixme!

The file system defines how data is stored and accessed. The file system divides the entire storage space according to a certain form. This facilitates the storage of files while increasing the utilization of storage media. The small blocks thus divided are called sectors. The FAT records which sectors all files are stored on. Fixme: wikipedia

FAT Data Structure

```
struct FILEINFO
{
  unsigned char name[8];   /* file name */
  unsigned char ext[3];    /* extend name of file */
  unsigned char type;      /* file attributes */
  char reserve[10];        /* reserve byte */
  unsigned short time;     /* the time for storing file */
  unsigned short date;     /* the date for storing file */
  unsigned short  clustno; /* the file from which sector on the disk is
   ↪  stored */
};
```

- is used to record file-related information, such as file name, size, etc.

Fixme!

FAT Functions Fixme: more about each function

```
void file_readfat(int *fat, unsigned char *img);
```

- read file allocation table

```
void file_loadfile(int clustno, int size, char *buf, int *fat, char *img);
```

- read file contents into memory

```
struct FILEINFO *file_search(char *name, struct FILEINFO *finfo, int max);
```

- search for files based on provided `name`.

2.2.3 API

In computer programming, APIs are subroutines that are used to develop applications. Usually, it communicates all parts of the software. A good API makes application development easier. By abstracting the underlying implementation and only exposing objects or actions the developer needs, an API simplifies programming.

All APIs

Fixme!

Fixme: more about each function

```
void api_putchar(int c);
```

- output a character `c` on the console window.

```
void api_putstr0(char *s);
```

- ouput a string `s` on the console window.

```
void api_putstr1(char *s, int l);
```

- output a string `s` on console window, `l` is the length of this string.

```
void api_end(void);
```

- end the application's running.

```
int api_openwin(char *buf, int xsiz, int ysiz, int col_inv, char *title);
```

- open a window, return the hanlder of window.

```
void api_putstrwin(int win, int x, int y, int col, int len, char *str);
```

- display string `str` on window.

```
void api_boxfilwin(int win, int x0, int y0, int x1, int y1, int col);
```

- draw a box on the window

```
void api_initmalloc(void);
```

- initialize the structure of the management memory

```
char *api_malloc(int size);
```

- allocating memory for the application.

```
void api_free(char *addr, int size);
```

- free up memory used by the application.

```
void api_point(int win, int x, int y, int col);
```

- draw a pint on `win` window.

```
void api_refreshwin(int win, int x0, int y0, int x1, int y1);
```

- refresh the window.

```
void api_linewin(int win, int x0, int y0, int x1, int y1, int col);
```

- draw a straight line.

```
void api_closewin(int win);
```

- close window.

```
int api_getkey(int mode);
```

- accept keyboard input.

```
int api_alloctimer(void);
```

- get timer.

```
void api_inittimer(int timer, int data);
```

- set the data of timer sending.

```
void api_settimer(int timer, int time);
```

- set the time of timer.

```
void api_freetimer(int timer);
```

- release timer.

```
void api_beep(int tone);
```

- make a buzzer sound

```
int api_fopen(char *fname);
```

- open file, return file hanlder

```
void api_fclose(int fhandle);
```

- close file

```
void api_fseek(int fhandle, int offset, int mode);
```

- locate a file

```
int api_fsize(int fhandle, int mode);
```

- return the size of `fhandle` file.

```
int api_fread(char *buf, int maxsize, int fhandle);
```

- read file, return read bytes.

```
int api_cmdline(char *buf, int maxsize);
```

- return the context of command line.

```
int api_getlang(void);
```

- return the language mode.

2.2.4 APPs

Since this design is only about the operating system itself, only three representative applications are introduced. The first is the application of the display color: `color`. The second is the application for timing: `sundial`. The third application is to imitate the application of stars in the sky: `stars2`.

3 Implementation

3.1 Boot Up

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

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. Fixme! Interrupt `0x10` is used for showing a character on the screen. Fixme: Not clear. You have to explain what is al/ah for. For example, as shown in table... al is for..., ah is for... int 0x10 is for...
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 is bigger or equal to 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 performing this function.

There is a pseudo code about this process:

```
   Result: Read two sides of one track
 1 ENTRANCE: call readloop();
 2 Procedure readloop()
 3 |  clear the times of failed to 0, si ← 0;
 4 |  call retry();
 5 Procedure retry()
 6 |  register parameter preparing;
 7 |  read a sector;
 8 |  if no carry then
 9 |  |  call next();
10 |  else
11 |  |  add 1 to si, si ← si + 1;
12 |  |  compare si with 5;
13 |  |  if si >= 5 then
14 |  |  |  goto error, FINISHED;
15 |  |  else
16 |  |  |  reset registers and call retry() to read again;
17 |  |  end
18 |  end
19 Procedure next()
20 |  memory address moved back 0x200;
21 |  add 1 to cl, preparing for reading the next sector, cl ← cl + 1;
22 |  if cl <= 18 then
23 |  |  call readloop() to read this sector;
24 |  else
25 |  |  cl > 18, it means that one side of this track is read already;
26 |  |  add 1 to dh, dh ← dh + 1, reverse the head pointer;
27 |  |  if dh < 2 then
28 |  |  |  it means the 1 side has not read yet, call readloop();
29 |  |  else
30 |  |  |  both sides have finished reading, FINSHED;
31 |  |  end
32 |  end
```

Fig. 3-1 algorithm of 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. 3-2.

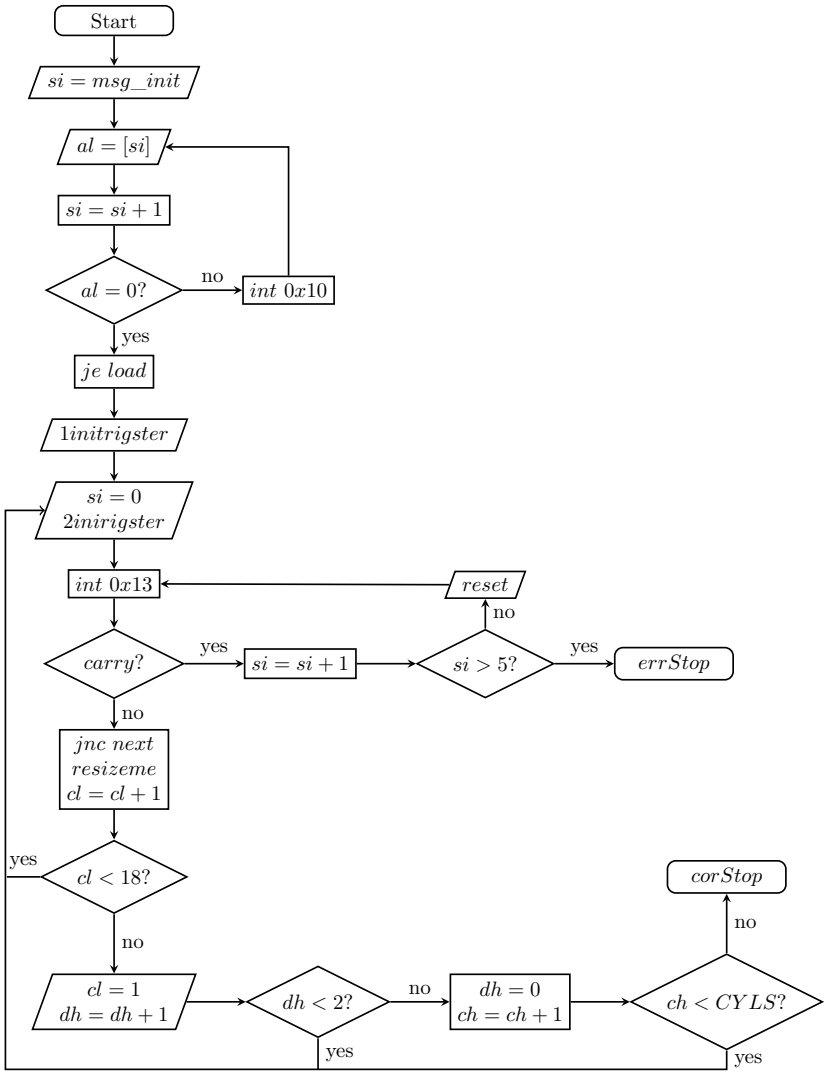

Fig. 3-2 the working flowchart of boot loader

3.2 Kernel

3.2.1 Memory Management

Memory Check

For memory management, memory must be checked to see how large the memory is. After i486, the CPU turned on the cache function. In order to avoid checking the cache, the CPU's cache function must be turned off. So first check the version of the CPU. Then check the memory capacity. When checking the memory capacity, first write a value to the same address and immediately read out whether the observation is equal to the previously written value. The entire inspection process can be summarized as follows.

1. Check the version of the CPU.
2. If the version of the CPU is greater than 386 then caching is disabled, otherwise nothing will be done.
3. Check memory
4. If the CPU version is greater than the 386 restore cache feature. Otherwise do nothing.

Step 2 can be divided into the following steps

1. Save the original value of a location in memory.
2. Write a constant to the above memory address.
3. Read the above memory address and invert it.
4. The inverted value of the constant prepared in advance is compared with the value obtained in the third step.
5. Equal to continue to check, otherwise stop.

In general, the C compiler has optimized optimization, so it seems that the above comparison is definitely equivalent. So the above check subfunction should be done in assembly language. Specific implementation, please see appendix A.2.1.

Memory Allocation

First explain some of the variables used to implement the memory allocation function. `size` refers to the space requested by the application but `SIZE` is the free space of one *entry*. `free` is the number of memory available memory.

1. Traversing all the free entries in memory to find an *entry* gives it more free space than the application requested. If found then 2, otherwise 6.
2. Add `size` to start address of *entry* found in 1. Subtract `size` from free space `SIZE` in *entry* found in 1. Jump to 3.
3. Determine if the remaining free space `SIZE` in the entry is 0. If so, go to 4. Otherwise 5.
4. Subtract the number of memory available entries `free` by 1.
5. Returns the starting address of available memory.
6. Returns 0 for memory allocation failure.

Specific implementation see code A.2.2.

Memory Release

Memory release is a more complex part of memory management. In order to keep the memory free entries in an orderly manner and merge adjacent entries, the memory release code becomes somewhat complicated. It is mainly divided into the following categories.

1. Neither merge with the previous *entry* nor with the latter *entry*. For this case, this entry needs to be inserted into the free memory entry. As the figure 3-3 shows. As shown in

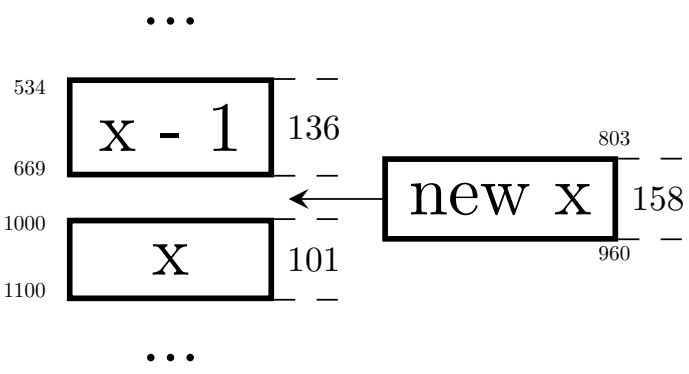

Fig. 3-3 insert an entry no merge

the figure, the starting address of `new x` 803 is not adjacent to the previous termination address 669, and its termination address 960 is also not adjacent to the starting address 1000 of the next entry. The so-called neighboring means that the difference between the two addresses is 1. Same as below.

2. Can be merged with the previous entry. As the figure 3-4 shows. As shown in the

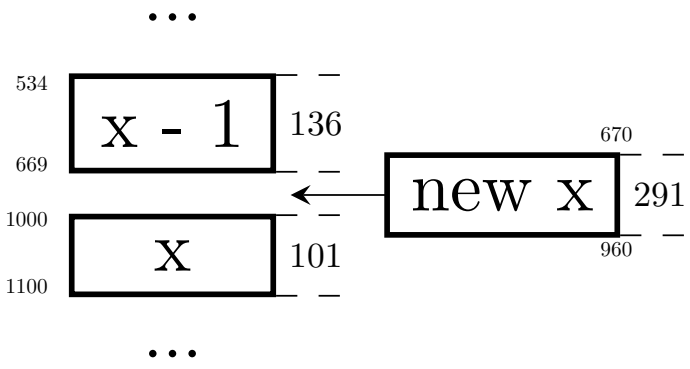

Fig. 3-4 Insert an item merged with the previous item

figure, the starting address of `new x` 670 is adjacent to the previous termination address 669, but its termination address 960 is not adjacent to the starting address 1000 of the next entry.

3. Can be merged with the latter entry. As the figure 3-5 shows. The termination address

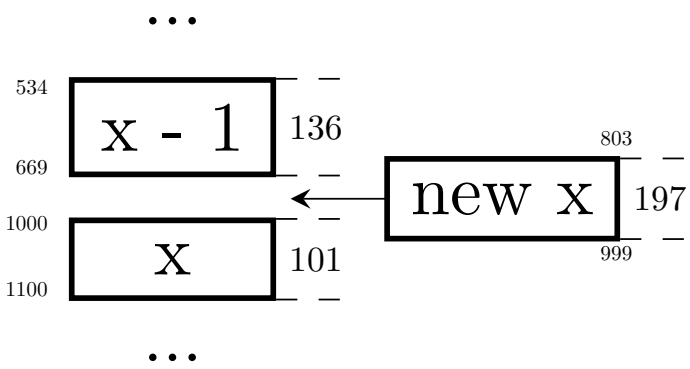

Fig. 3-5 Insert an item merged with the latter item

999 of `new x` is adjacent to the starting address 1000 of the next entry, but its starting address of `new x` 803 is not adjacent to the previous termination address 669.

The memory release process can be described as follows.

1. Find the insertion position *i*. The so-called insertion position is an `entry` number that

satisfies certain conditions. This condition is that the `size` of this `entry` is larger than the size requested by the application.

2. If $i > 0$, merge if the newly inserted *entry* can be merged with the previous `entry`, the same as for the latter *entry*. As long as it merges with the former, the function returns 0 and the release is successful. Otherwise jump to 3.
3. The program proceeds here so that the newly inserted `entry` must not be merged with the previous `entry`. Now check if the newly inserted `entry` can be merged with the latter `entries`. If so, merges it and return 0. Otherwise jump to 4.
4. The program proceeds here to represent that the newly inserted `entry` cannot be merged with any of the entries. Now check the number of used entries with 4090. If the former is less than the latter, move all items after the i'th item back one position to make room for new items. Then insert this new `entry`. Otherwise jmup to 5.
5. Failed to release, return -1.

The above release process can be explained by the algorithm 3-6. Of course, the flow chart 3-7 will be more clear. For specific code implementation, see A.2.3

3.2.2 Process Management

Process management is a very complex but very important part of operating system development. This part of the development has a very big impact on the performance of the entire system. For the RongOS operating system, all tasks are managed hierarchically. From *level 0* to *level 10*, there are a total of ten levels in the system. All these ten levels and the space they prepare for the task are called *task pool*. For each task, it also has a *priority* attribute. It actually specifies how long each process can run, also known as a *time slice*.

As Fig. 3-8 shows, each level can hold 100 tasks. The lower the level number, the higher the level's privilege. For example, *x-level* permissions are higher than *x+1*. The scheduler will select the task to run from a high-privilege level. Only lower-level tasks are scheduled when the task at the higher privilege level is completed. While tasks within the same layer are run in turn, their running time may be different, that is, the *time slice* is also called *priority*. This can be illustrated by the peripheral loop arrows in *level 0* in Fig. 3-8. The other *levels*

are also similar except that the peripheral loop arrows are ignored.

In Fig. 3-8, a task is added to its corresponding level. There are many tasks in each level waiting for scheduling or a running task. Tasks that are waiting for scheduling are represented in italics but the task being run is underlined in each level. A place after the last task waiting to be scheduled is the *insertion point* of a new task at a certain level.

The PIC chip will generate 100 clock interrupts per second. Each time a clock interrupt occurs, it is serviced by the interrupt service routine 0x20. This subroutine will call the scheduler to schedule a task to run based on the above method of selecting the task to run. When a run is completed in one round, there are three destinations for a process.

1. The task did not complete, but the *time slice* was used up. It will be added to the task pool again and wait for it to be selected again to begin another run.
2. The task completes the run and then enters the end processing subroutine.
3. The *time slice* of the process runs out, but it goes to *sleep*. The so-called *sleep* state means that the process is paused, waiting for something to happen and waking it up again, adding it to the *task pool* and starting a round of running. Something such as a keyboard input program waits for user input.

As shown in Fig. 3-8, it can be seen as a snapshot of all the tasks in the system. In this transient state, a task is being added to the $(3 + 1 = 4)$ point of *x-level* of *task pool*. What is running in the system is task ④ of *level 0*.

Task Allocate and Add

After the new process arrives, it must apply for a task structure to record its information. This process is simpler simply by iterating through the *tasks0* structure array variable to find an unused task structure and assign it to the new task. *tasks0* is used to record information for all tasks. At the same time put the address of this structure into the *tasks* structure array. This structure *tasks* is used to record the addresses of all tasks. Note that *tasks* and *tasks0* are functionally different arrays of structures. *tasks0* records information for all 1000 tasks in the system, while *tasks* record their addresses.

The next step is to add to the *task pool* and wait to be scheduled. When adding, specify the *level* and *priority*. The specified *level* and *priority* may not be the same as the original

task, or they may be the same. Different situations need to be handled differently. This can be done by the following steps.

1. If the *level* passed in is less than 0, it means that the task is added without changing the *level* of the task, and the *level* of the original task is used to cover the incoming level. The check here is mainly for processing when the task is awakened. When the task is awakened it should not change its *level*. Continue to step 2.
2. If the passed in *priority* is greater than 0, the *priority* of the original task is overwritten with the passed in priority. Continue to step 3.
3. If this task is waiting to be scheduled and the incoming *level* is different from its original *level*. This means that this task must be removed from the original *level*. Continue to step 4.
4. If the task is not in the task pool, it is added to the *task pool* according to its *level*. These steps may change the level of the task, set the level change flag *lv_change* to 1 so that the next time the scheduler schedules the task, it must find the task in the entire *task pool* to run again.

Specific code implementation, see A.2.4 and A.2.5. Fig. 3-9 is an algorithm that adds tasks to the task pool to run.

Task Scheduling and Running

The next step is for the scheduler to start scheduling task to running. The work of the scheduler requires the support of clock interrupts. Every time a clock interrupt occurs, the scheduler is run once. The *now_lv* variable is used to record which level is currently running. The *now* variable is a pointer to a running task of a certain level. The specific workflow of the scheduler is described in the following steps.

1. Find the currently running level *tl* by *now_lv*. Move the *now* pointer of this layer by one position. Continue to step 2.
2. If the number of *running* tasks is equal to the *now* pointer. Move back *now* pointer to 0. Continue to step 3
3. Check if the *lv_change* flag is 1. If it is 1, check the entire *task pool* to find the new *now_lv* and set the *lv_change* flag to 0. Continue to step 4.

4. Get the currently running task from *now_lv* with the *now* pointer, set the time slice, and jump to the task to start running.

Specific code implementation, see A.2.6 and A.2.7.

Task Sleep

The task may go to *sleep* after it finishes running. A task going to sleep is actually removing it from the *task pool*. Sleeping the task requires considering whether this task is a running task. If this is the case, after the task is removed, the task switch will be performed. Then jump to the newly switched task. These steps can be summarized as follows

1. Call *task_now* to return to the currently running task. Go to the next step.
2. Call *task_remove* to remove this task from *task pool*. Go to the next step.
3. The task that is going to sleep are compared with the tasks that is running. If they are equal, switch to the new task to start running.

Specific code implementation, see A.2.8.

Task_now and Task_remove Functions

There are also two small functions that facilitate task management and are introduced here. One is the *task_now* function and the other is the *task_remove* function. *task_now*. The *task_now* function is used to return the address of the task that is currently running on the system. This function is relatively simple. The first is to find the current running level based on the *now_lv*, then the level of the now pointer to find the task that the layer is running, and finally returns its address.

The *task_remove* function is somewhat complicated. It may involve moving all tasks for a layer. It may also cause various pointers to appear strange. All of these must be considered carefully, otherwise the system may experience unpredictable and strange behavior. Its running process can be summarized as follows.

1. First find the corresponding level of the task *tl*. Go to the next step.
2. Find out where this task is. Go to the next step.
3. Reduce the number of processes running by 1. Go to the next step.

4. If the position of this task is before the *now* pointer, the *now* pointer is decremented by one. Go to the next step.
5. If the *now* pointer is greater than the number of *running* processes, it is necessary to correct the *now* pointer to 0. Go to the next step.
6. Change the *flags* of the task to 1 to indicate that the task is dormant. Go to the next step.
7. Move the tasks between i and *running* one step forward.

Specific code implementation, see A.2.9.

3.3 API

3.4 APPs

```
Result: free memory
1  ENTRANCE: call memFree(struct MEMMAN *man, unsigned int
     addr, unsigned int size);
2  Procedure memFree(struct MEMMAN *man, unsigned int addr,
     unsigned int size)
3      Finding the starting address of the first entry i is greater than the
         released address addr;
4      if i > 0 then
5          if i'th entry can be merged with the previous one then
6              man->free[i - 1].size ← man->free[i - 1].size + size;
7              if i entry is less than the number of free entries then
8                  if i'th can be merged with the latter then
9                      man->free[i - 1].size ← man->free[i-1].size +
                         man->free[i].size;
10                     man->frees ← man->frees - 1;
11                     move all free entries after i by one;
12                 else
13                     nothing to do here;
14                 end
15             else
16                 nothing to do here;
17             end
18         else
19             Successfully released, return 0, FINISHED;
20         end
21     else
22         nothing to do here;
23     end
24     if i < the number of free entries then
25         if i'th entry can be merged with the latter then
26             man->free[i].addr ← addr;
27             man->free[i].size ← man->free[i].size + size;
28             Successfully released, return 0, FINISHED;
29         else
30             nothing to do here;
31         end
32     else
33         nothing to do here;
34     end
35     if the number of free items used < the maximum number of free
         items(4090) then
36         move back all free entries after i'th entry by one;
37         man-> frees ← man->frees + 1;
38         man -> free[i].addr ← addr;
39         man -> free[i].size ← size;
40         Successfully released, return 0, FINISHED;
41     else
42         nothing to do here;
43     end
44     the number of release failures plus one;
45     increase the size to release the failed space;
46     Release failed, return -1, FINISHED;
```

Fig. 3-6 algorithm of release memory

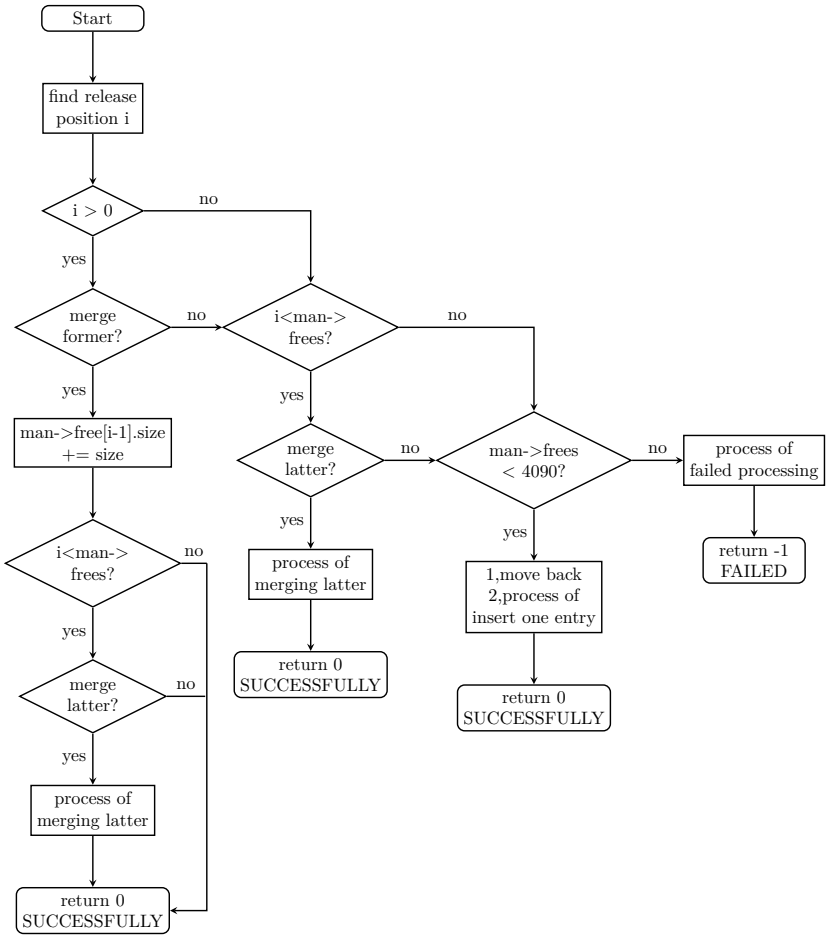

Fig. 3-7 flow chart of relase memory

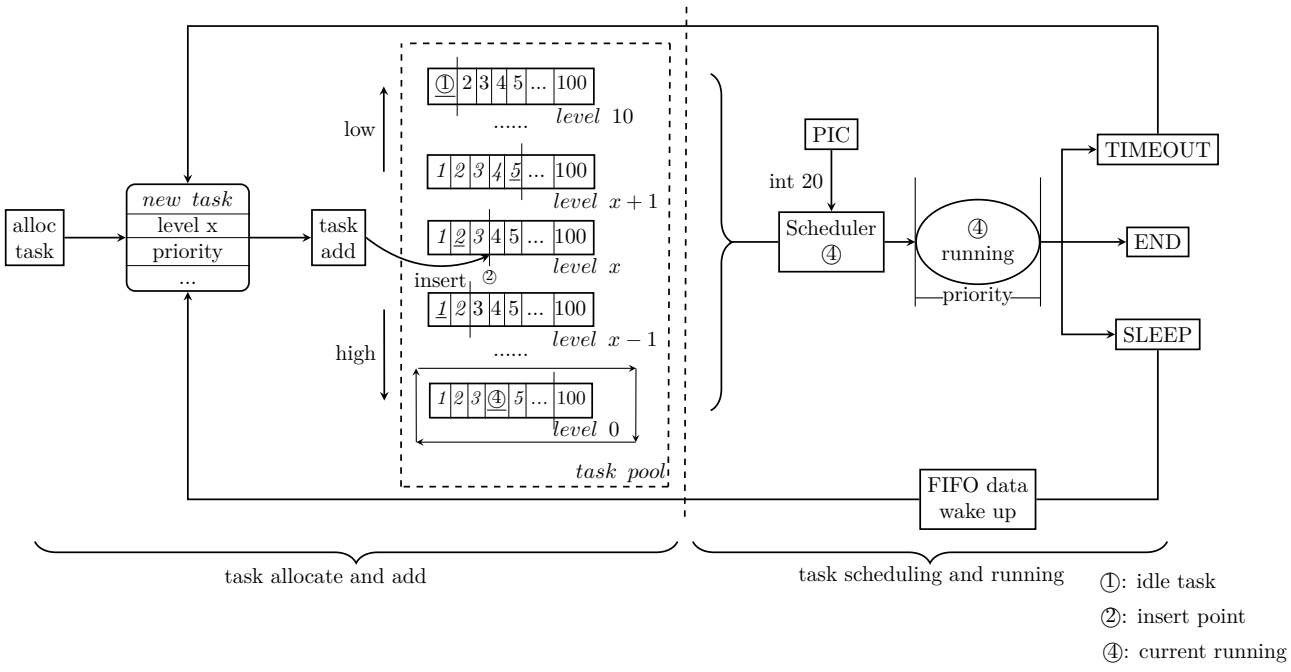

Fig. 3-8 process of task management

```
   Result: schedule a task to run
 1 ENTRANCE: call taskRun(sturct TASK *task, int level, int priority);
 2 Procedure taskRun(struct TASK *task, int level, int priority)
 3     if level < 0 then
 4         don't change level, level ← task− > level;
 5     else
 6         nothing to do here;
 7     end
 8     if priority > 0 then
 9         Change the priority, task− > priority ← priority;
10     else
11         nothing to do here;
12     end
13     if the task is running and not at the level you want to set then
14         Remove the task from the original level;
15     else
16         nothing to do here;
17     end
18     if task is not running, but sleeping then
19         set the new level, task− > level ← level;
20         waking up from sleep, call taskAdd(task);
21     else
22         nothing to do here;
23     end
24     change the lv_change flag to 1, therefore, tasks must be switched
         when the task is scheduled next time;
25 Procedure taskAdd(struct TASK *task)
26     look for the corresponding level of the task;
27     place the task in the location indicated by running;
28     runinng++; Mark the task as running;
```

Fig. 3-9 algorithm to add tasks to task pool and prepare to run

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/

参考文献

[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

```
55 init:
56         mov al, [si]
57         add si, 1 ; increment by 1.
58         cmp al, 0
59         je load ; if al == 0, jmp to load, the msg_init info displayed.
60 ; the lastest character is null character, coding in 0.
61
62         mov ah, 0x0e ; write a character in TTY mode.
63         mov bx, 15   ; specify the color of the character.
64         int 0x10 ; call BIOS function, video card is number 10.
65         jmp init
```

A.1.2 Read the second sector

```
87 load:
88         mov ax, 0
89         mov ax, 0x0820 ; load C0-H0-S2 to memory begin with 0x0820.
90         mov es, ax
91         mov ch, 0 ; cylinder 0.
92         mov dh, 0 ; head 0.
93         mov cl, 2 ; sector 2.
94
95
96 readloop:
97         mov si, 0 ; si register is a counter, try read a sector
98 ; five times.
99
100
101 retry:
102        mov ah, 0x02 ; parameter 0x02 to ah, read disk.
103        mov al, 1 ; parameter 1 to al, read disk.
104        mov bx, 0
105        mov dl, 0x00 ; the number of driver number.
106        int 0x13 ; after prepared parameters, call 0x13 interrupted.
```

A.1.3 Read two sides of a track

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

```
112
113         ; reset the status of floppy and read again.
114         mov ah, 0x00
115         mov dl, 0x00
116         int 0x13
117         jmp retry
118
119
120 next:
121         mov ax, es
122         ; we can not directly add to es register.
123         add ax, 0x0020 ; add 0x0020 to ax
124         mov es, ax ; the memory increase 0x0020 * 16 = 512 byte.
125         ; size of a sector.
126         add cl, 1 ; sector number add 1.
127         cmp cl, 18 ; one track have 18 sector.
128         jbe readloop ; jump if below or equal 18, read the next sector.
129         mov cl, 1 ; cl number reset to 1, ready to read the other side.
130         add dh, 1 ; the other side of floppy.
131         cmp dh, 2 ; only two sides of floppy.
132         jb readloop ; if dh < 2, read 18 sectors of the other sides
```

A.1.4 The next cylinder

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

A.2 Kernel Modules

A.2.1 Memory check process

```
7  unsigned int memtest(unsigned int start, unsigned int end)
8  {
9          char flg486 = 0;
10         unsigned int eflg, cr0, i;
11
12         //check whether it is 386 or 486 or later.
13         eflg = io_load_eflags();
14         eflg |= EFLAGS_AC_BIT;
15         io_store_eflags(eflg);
16         eflg = io_load_eflags();
17
18         if ((eflg & EFLAGS_AC_BIT) != 0)
19                 flg486 = 1;
20
21         eflg &= ~EFLAGS_AC_BIT;
22         io_store_eflags(eflg);
```

```
23
24          if (flg486 != 0)
25                  {
26                          cr0 = load_cr0();
27                          cr0 |= CR0_CACHE_DISABLE; // caching prohibited
28                          store_cr0(cr0);
29                  }
30
31          i = memtest_sub(start, end);
32
33          if (flg486 != 0)
34                  {
35                          cr0 = load_cr0();
36                          cr0 &= ~CR0_CACHE_DISABLE; // cache permission
37
38                          store_cr0(cr0);
39                  }
40
41          return i;
42  }
```

```
220 _memtest_sub: ; unsigned int memtest_sub(unsigned int start, unsigned int
    ↪  end)
221                 push edi
222                 push esi
223                 push ebx ; store the value of three registers
224                 mov esi, 0xaa55aa55 ; pat0 = 0xaa55aa55
225                 mov edi, 0x55aa55aa ; pat1 = 0x55aa55aa
226                 mov eax, [esp + 12 + 4] ; i = start
227
228 mts_loop:
229                 mov ebx, eax
230                 add ebx, 0xffc ; p = i + 0xffc
231                 mov edx, [ebx] ; old = *p;
232                 mov [ebx], esi ; *p = pat0
233                 xor dword [ebx], 0xffffffff ; *p ^= 0xffffffff
234                 cmp edi, [ebx] ; if (*p != pat1) goto fin
235                 jne mts_fin
236                 xor dword [ebx], 0xffffffff ; *p ^= 0xffffffff
237                 cmp esi, [ebx] ; if (*p != pat0) goto fin;
238
239                 jne mts_fin
240                 mov [ebx], edx ; *p = old;
241                 add eax, 0x1000 ; i += 0x1000;
242                 cmp eax, [esp + 12 + 8] ; if (i <= end) goto mts_loop;
243                 jbe mts_loop
244                 pop ebx
245                 pop esi
246                 pop edi
247                 ret
248
```

```
249 mts_fin:
250                 mov [ebx], edx ; *p = old;
251                 pop ebx
252                 pop esi
253                 pop edi
254                 ret
```

A.2.2 Memory allocation process

```
64 unsigned int memman_alloc(struct MEMMAN *man, unsigned int size)
65 {
66   unsigned int i, a;
67   for (i = 0; i < man -> frees; i++)
68     {
69       if (man -> free[i].size >= size) // discovering an free space of
         ↪  sufficient size
70         {
71           a = man -> free[i].addr;
72           man -> free[i].addr += size;
73           man -> free[i].size -= size;
74           if (man -> free[i].size == 0)// if the free size of the i'th
             ↪  item is zero
75             {
76               man -> frees--;
77               for (; i < man -> frees; i++)
78                 man -> free[i] = man -> free[i + 1]; // move
79             }
80           return a;
81         }
82     }
83
84   return 0;
85 }
```

A.2.3 Memory release process

```
88 int memman_free(struct MEMMAN *man, unsigned int addr, unsigned int size)
   ↪  // release
89 {
90   int i, j;
91   /*
92     considering easiness of summarization, it is better for free[] to be
   ↪  arranged in order of addr
93     So first, decide where to put
94   */
95   for (i = 0; i < man -> frees; i++)
96     {
97       if (man -> free[i].addr > addr)
98         break;
99     }
```

```
100     // free[i - 1].addr < addr < free[i].addr
101     if (i > 0) // if there is an free entry near the released entry
102       {
103         if (man -> free[i - 1].addr + man -> free[i - 1].size == addr)
104           {
105             man -> free[i - 1].size += size;
106             if (i < man -> frees)
107               {
108                 if (addr + size == man -> free[i].addr)
109                   {
110                     man -> free[i - 1].size += man -> free[i].size;
111                     //delete man->free[i]
112                     man -> frees--;
113                     for (; i < man -> frees; i++)
114                       {
115                         man -> free[i] = man -> free[i + 1]; // moving all
                              ↪  after i'th item.
116                       }
117                   }
118               }
119             return 0; // successful completion
120           }
121
122       }
123
124     if (i < man -> frees)
125       {
126         if (addr + size == man -> free[i].addr)
127           {
128             man -> free[i].addr = addr;
129             man -> free[i].size += size;
130             return 0; // successful completion
131           }
132       }
133
134     if (man -> frees < MEMMAN_FREES)
135       {
136         for (j = man -> frees; j > i; j--)
137           {
138             man -> free[j] = man -> free[j - 1];
139           }
140
141         man -> frees++;
142
143         if (man -> maxfrees < man -> frees)
144           {
145             man -> maxfrees = man -> frees; // update maximum value
146           }
147
148         man -> free[i].addr = addr;
```

```
149         man -> free[i].size = size;
150         return 0; // successful completion
151       }
152
153     // can not move backwards.
154     man -> losts++;
155     man -> lostsize += size;
156
157     return -1;// failed end.
158   }
```

A.2.4 Task allocate

```
115   struct TASK* task_alloc(void)
116   {
117     int i;
118     struct TASK *task;
119     for (i = 0; i < MAX_TASKS; i++)
120       {
121         if (taskctl -> tasks0[i].flags == 0)
122           {
123             task = &taskctl -> tasks0[i];
124             task -> flags = 1; // mark in use
125             task -> tss.eflags = 0x00000202;
126             task -> tss.eax = 0; // leave it 0 for now
127             task -> tss.ecx = 0;
128             task -> tss.edx = 0;
129             task -> tss.ebx = 0;
130             task -> tss.ebp = 0;
131             task -> tss.esi = 0;
132             task -> tss.edi = 0;
133             task -> tss.es = 0;
134             task -> tss.ds = 0;
135             task -> tss.fs = 0;
136             task -> tss.gs = 0;
137             task -> tss.iomap = 0x40000000;
138             task -> tss.ss0 = 0;
139             return task;
140           }
141       }
142     return 0; // all are in use
143   }
```

A.2.5 Task add

```
14  void task_add(struct TASK *task)
15  {
16    struct TASKLEVEL *tl = &taskctl -> level[task -> level];
17    tl -> tasks[tl -> running] = task;
18    tl -> running++;
```

```
19    task -> flags = 2; // in running
20    return;
21  }
```

A.2.6 Task scheduling

```
186 void task_switch(void)
187 {
188   struct TASKLEVEL *tl = &taskctl -> level[taskctl -> now_lv];
189   struct TASK *new_task, *now_task = tl -> tasks[tl -> now];
190   tl -> now++; // addition for the next task switch in the same level
191   if (tl -> now == tl -> running)
192     tl -> now = 0;
193
194   if (taskctl -> lv_change != 0)// need to change level
195     {
196       task_switchsub();// find the highest level tasks
197       tl = &taskctl -> level[taskctl -> now_lv];
198     }
199   new_task = tl -> tasks[tl -> now];
200   timer_settime(task_timer, new_task -> priority);
201   if (new_task != now_task)
202     farjmp(0, new_task -> sel);
203
204   return;
205 }
```

```
49  void task_switchsub(void)
50  {
51    int i;
52
53    // find the highest level
54    for (i = 0; i < MAX_TASKLEVELS; i++)
55      {
56        if (taskctl -> level[i].running > 0)
57          break; // found
58      }
59
60    taskctl -> now_lv = i;
61    taskctl -> lv_change = 0;
62    return;
63  }
```

A.2.7 Task running

```
146 void task_run(struct TASK *task, int level, int priority)
147 {
148   if (level < 0)
149     level = task -> level; // don't change level
150
```

```
151   if (priority > 0)
152     task -> priority = priority;
153
154   if (task -> flags == 2 && task -> level != level)// change in active
      ↪  level
155     task_remove(task);// flag will be 1
156
157   if (task -> flags != 2) // waking up from sleep
158     {
159       task -> level = level;
160       task_add(task);
161     }
162
163   taskctl -> lv_change = 1; // change the lv_change, so next time must
      ↪  check level
164   return;
165 }
```

A.2.8 Task sleep

```
167 void task_sleep(struct TASK *task)
168 {
169   struct TASK *now_task;
170   if (task -> flags == 2)
171     {
172       // if it is running
173       now_task = task_now();
174       task_remove(task); // when this function done, flags will be 1
175       if (task == now_task)
176         {
177           // sleeping myself, task switch needing
178           task_switchsub();
179           now_task = task_now();
180           farjmp(0, now_task -> sel);
181         }
182     }
183   return;
184 }
```

A.2.9 Task remove

```
23 void task_remove(struct TASK *task)
24 {
25   int i;
26   struct TASKLEVEL *tl = &taskctl -> level[task -> level];
27
28   for (i = 0; i <tl -> running; i++)// find where the task is
29     {
30       if (tl -> tasks[i] == task)
31         break; // here
```

```
32      }
33
34    tl -> running--;
35    if (i < tl -> now)
36      tl -> now--;
37
38    if (tl -> now >= tl -> running)
39      tl -> now = 0; // if now is a strange value, fix it
40
41    task -> flags = 1;// sleeping
42
43    for (; i < tl -> running; i++)
44      tl -> tasks[i] = tl -> tasks[i + 1];
45
46    return;
47  }
```