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

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, 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*.

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. 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.

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:

User programs

API

trap

User level

Kernel level

System call interface

File subsystem

FAT   · · ·

Process control
subsystem

Graphic

Scheduler

Memory
management

Device drivers

Hardware control

Kernel level

Hardware level

Hardware
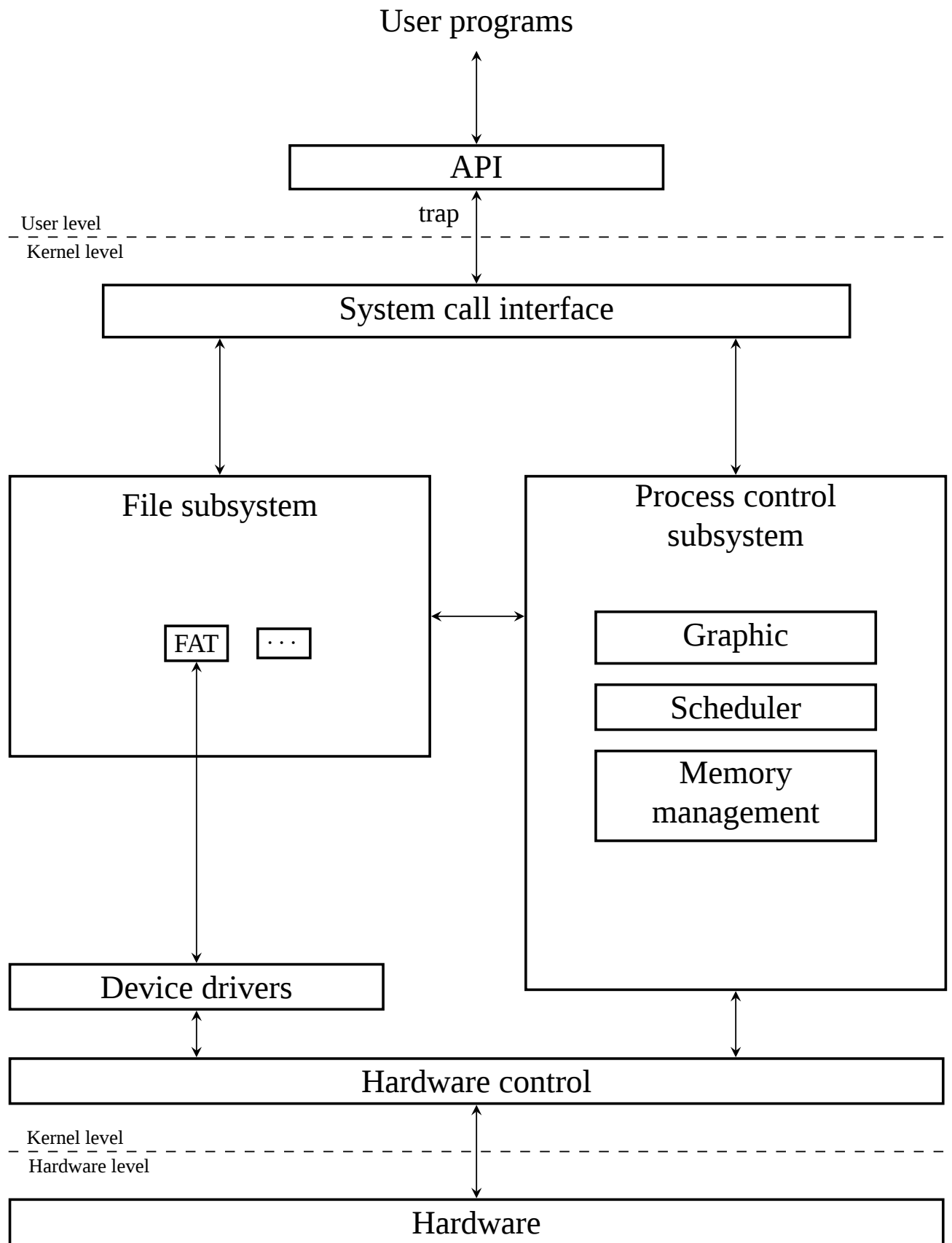
Fig. 2-1   top-level design

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 is the core part of the operating system. Usually the kernel is loaded after the bootloader finishes loading. The code in the kernel is separate from the application and other non-critical operating system code in memory. This is mainly to protect the kernel.

The kernel receives the API calls from user processes, and operates the hardwares through the device drivers. The functions it provides mainly include the following parts

1. Mediate the use of the CPU, random-access memory and input/output devices and other devices for each program.

2. Provides mechanisms for process communication.

3. The kernel itself and other user programs can be protected even when there are malicious programs.

4. Serve user programs through system calls.

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 pag-

ing 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

```
1  struct FREEINFO
2  {
3    unsigned int addr; /* start address of free space */
4    unsigned int size; /* size of free mem in bytes*/
5  };
```

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

```
1  struct MEMMAN
2  {
3    int frees;     /* free memory blocks */
4    int maxfrees;
5    int lostsize; /* the sum of the failed memory size to freed*/
6    int losts;     /* number of failures */
7    struct FREEINFO free[MEMMAN_FREES]; /* free memory block info */
8  };
```

- 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.
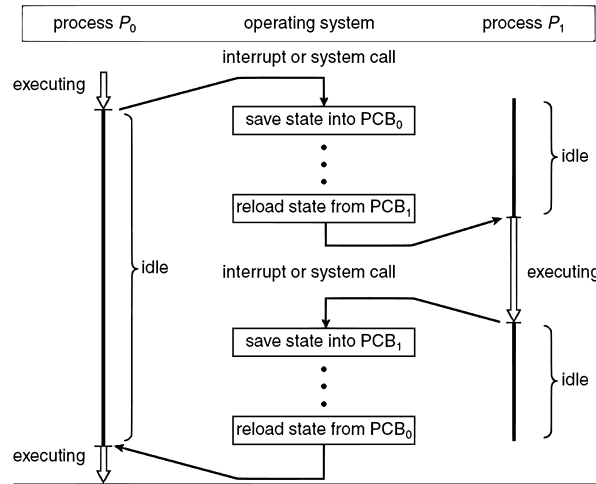
Fig. 2-2   context switch

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.

The operating system is responsible for allocating resources to processes when managing tasks. Manage the communication between them. To meet these goals, the operating system maintains a data structure called PCB(process control block) for each task. The PCB describes the resources and the state of the task, which also allows the operating system to exert control to each task.

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 level. The definition of level, see 3.2.2.

```
1   struct TSS32
2   {
3     int esp0, esp1, esp2; /* stack pointer register */
4     int ss0, ss1, ss2;    /* stack segment register */
5     int cr3;    /* control register */
6     int eip;    /* instruct pointer register */
7     int eflags; /* registers flag */
8     int eax;    /* accumulator register */
9     int ecx;    /* counter register */
10    int edx;    /* data register */
11    int ebx;    /* base register */
12    int esp;    /* stack pointer register */
13    int ebp;    /* base pointer register*/
14    int esi;    /* source index register */
15    int edi;    /* destination index register */
16    int es;     /* extra segment register */
17    int cs;     /* code segment register */
18    int ss;     /* stack segment register */
19    int ds;     /* data segment register */
20    int fs;     /* segment part 2 */
21    int gs;     /* segment part 3 */
22    int ldtr;   /* LDT segment selector */
23    int iomap;  /* I/O map base address */
24  };
```

- holds information about task status segments, which are based on CPU specifications[6]. The *eip* register records where in the memory the next instruction to be executed by the CPU is located.

```
1   struct TASKCTL
2   {
3     int now_lv;    /* current activity level */
4     int lv_change; /* does the level need to be changed next time the task
    ↪  is switched */
5     struct TASKLEVEL level[MAX_TASKLEVELS]; /* all levels*/
6     struct TASK tasks0[MAX_TASKS];         /* all running tasks */
7   };
```

- is used to control all tasks in the system. This structure uses the *now_lv* variable to record the current running *level* in the system. See Fig. 3-8.

```
1  struct TASK
2  {
3    int sel;              /* the number of GDT */
4    int flags;            /* the state of task */
5    int level;            /* the level of task */
6    int priority;         /* the priority of task */
7    struct FIFO32 fifo;   /* a fifo buffer */
8    TSS32 tss;            /* TSS segment for a task */
9    struct CONSOLE* cons; /* the console window address of task */
10   int ds_base;          /* data segment address of APPs */
11   int cons_stack;       /* the stack address of APPs */
12   struct SEGMENT_DESCRIPTOR ldt[2]; /* tow LDT segments of task */
13   struct FILEHANDLE* fhandle; /* file handles for manipulating files */
14   int* fat; /* file allocation table */
15   char* cmdline; /* store the command line context */
16   unsigned char langmode;  /* which font to use */
17   unsigned char langbyte1; /* store the first byte of the full-width
   ↪  character */
18 };
```

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

Task Management Functions

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

- Returns the address of which task level of which task is currently running.

```
struct TASK *task_init(struct MEMMAN *memman);
```

- Initialize all empty tasks. This function will register all 1000 empty tasks to the GDT table. Initialize all levels of task running variables *running* to 0. The pointer *now* of each level indicating that the task is currently running is also set to 0. In fact, this function registers all empty PCBs for future management tasks. At the same time, the task that calls this task needs tobe managed so the address of a PCB is returned. An idle task needs to be set in the last *level*. Ensure that at least one task is running in the system.

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

- Get an empty PCB from the *task pool 3.2.2*. Return its address if successful, otherwise 0.

```
void task_run(struct TASK *task, int level, int priority);
```

- Add a task to *task pool*. The *level* and *priority* of tasks through this function can also be set. If the task is asleep, wake it up. Note that this function will not immediately cause the running of task after the execution of this function. Instead wait to be scheduled.

```
void task_switch(void);
```

- Switch to the next task. This function will select one of the highest *level* tasks to run from the *task pool*.

```
void task_sleep(struct TASK *task);
```

- Put a task into sleep. The task that is being promoted to sleep may be the currently running task in the system. If this is the case, it must also schedule the task in the entire *task pool*. Then jump to this new task and start running.

Graphic Management

The patterns on the screen belong to one *sheet*. A *sheet* is a structure that records how a certain height of a certain area of a computer screen should be displayed. Graphics management is also responsible for refreshing the computer screen when the content displayed by a *sheet* changes. Graphic management is also responsible for setting the height of the *sheet*. The *sheet* height setting is a relatively complex part of the graphics management, which will be explained in detail in the implementation section. Usually the lowest *sheet* is the computer desktop, which is the background layer. The highest *sheet* is the mouse layer. In general, for modern operating systems, graphics management is not a matter of the operating system kernel. So this part only shows in detail the more complicated part of the *sheet* height setting. Fig. 2-3 shows the possible state of some of the *sheet* on the computer screen. The figure shows three *sheet* and is labeled 0, 1, 2. Of course there may be more than three *sheet*. *Sheet* 2 is in the highest position, and *sheet* 0 is the lowest.
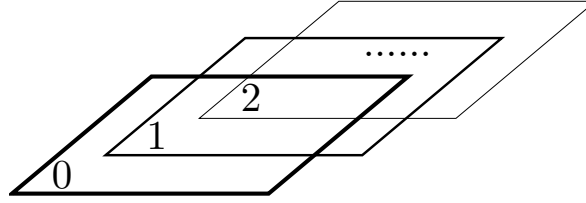
Fig. 2-3   a state of sheets

Graphic Management Data Structures

```
1   struct SHEET
2   {
3     char* buf;   /* address of the graphic content depicted */
4     int bxszie;  /* size of x coordinate of sheet */
5     int bysize;  /* size of y coordinate of sheet */
6     int vx0;     /* x coordinate of sheet */
7     int vy0;     /* y coordinate of sheet */
8     int col_inv; /* number of invisible color */
9     int height;  /* height of sheet */
10    int flags;   /* states of sheet, using or not */
11  };
```

- is used to record layer-related information, including the layer's size and position.

```
1   struct SHTCTL
2   {
3     unsigned char* vram; /* the address of VRAM */
4     unsigned char* map;  /* which layer the pixel on the screen belongs to*/
5     int xsize; /* the x size of screen */
6     int ysize; /* the y size of screen */
7     int top;   /* the height of the top layer */
8     struct SHEET* sheets[MAX_SHEETS]; /* order all layer addresses in order
      ↪  */
9     struct SHEET sheets0[MAX_SHEETS]; /* all layers */
10  };
```

- 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. Mark all sheet structures as 0 to indicate that they are not used.

```
struct SHEET *sheet_alloc(struct SHTCTL *ctl);
```

- Return a *SHEET* structure if success, otherwise 0. Find an unused *sheet* from the sheet structure control structure and making this *sheet* in using.

```
void sheet_setbuf(struct SHEET *sht,unsigned char *buf,int xsize,int
↪   ysize,int col_inv);
```

- Set the properties of the *buf* pf *sheet*.

```
void sheet_updown(struct SHEET *sht, int height);
```

- Set the height of layer *sht*. The height of the *sheet* is more complicated to set, which will be explained in detail in the implementation section.

```
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*. There are many *sheet* on the computer screen. Not all the ranges of all *sheet* will be refreshed when the screen is refreshed. Considering the efficiency of the refresh, this also requires a delicate design. But this is generally not something that the operating system kernel cares about, so it will be omitted.

```
void sheet_free(struct SHEET *sht);
```

- Release a *sheet*. In addition to marking the *sheet* as unused, the first thing to do is to hide the *sheet*.

```
void make_window8(unsigned char *buf, int xsize, int ysize, char *title,
↪   char act);
```

- Make a window in *SHEET buf*. A window is almost indistinguishable from a *sheet*. When we create a window, we simply write different data to the *buf* of the *sheet* to make it appear as a window.

```
void putfonts8_asc_sht(struct SHEET *sht, int x, int y, int c, int b, char
↪   *s, int l);
```

- Paint the background color *b* and write the characters and finish the refreshing. *l* specifies the length of the string *s* to be written.

System Calls

   The application requests the operating system kernel to provide services through system calls. These services include both hardware services such as reading disks and software services such as creating child processes and communicating between processes. But in the *RongOS* operating system, the user process is not directly facing the system call, but the API. System calls provide an essential interface between a API and the operating system kernel.

System Calls Prototype

```
void cons_putchar(struct CONSOLE *cons, int chr, char move);
```

- Put a *chr* character on *cons* console. A *console* is a command line window where users can enter commands to run. Entering a character at the command line involves moving the mouse. This is controlled by *move* variable.

```
void cons_newline(struct CONSOLE *cons);
```

- Newline in *cons* console. The command line needs to wrap when the enter key is pressed.

```
void cons_putstr0(struct CONSOLE *cons, char *s);
```

- Put string *s* in *cons* console, no length was specified. This function will call *void cons_putchar(struct CONSOLE *cons, int chr, char move)* to complete its service.

```
void cons_putstr1(struct CONSOLE *cons, char *s, int l);
```

- Put string *s* in *cons*, *l* is the length of string *s*. In general, the service provided by this function is similar to that of *void cons_putstr0(struct CONSOLE *cons, char *s)*. Only in this function can specify the length of the string by *l*.

```
int cmd_app(struct CONSOLE *cons, int *fat, char *cmdline);
```

- Start an application based on the input from the command line *cmdline*. emphfat is a file allocation table. In this table it records which files are stored on which blocks and how these blocks relate to each other. This function finds the application to start by searching the *fat* table and runs it.

Driver

The device drivers are used to control and manipulate the hardware. 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. In this way, application programmers can develop hardware-independent programs.

When the driver is called, it sends the command to operate the hardware. Once the hardware device returns data, the driver informs the calling program to process it. The entire process is an interruption.

For *RongOS*, the main hardware devices are floppy disks, mice, keyboard, and screen. So the following driver functions are mainly used to manipulate them.

Driver Functions

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

- Initialize the palette. It is necessary to set the color number in the palette register for setting palette mode.

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

- Initialize the PIC. PIC is an abbreviation for programmable interrupt controller. PIC is used to inform the CPU if there is an interrupt.

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

- Initialze the keyboard. The mouse control circuit is included in the keyboard control circuit. Therefore, if the initialization of the keyboard circuit is completed, the initialization of the mouse circuit is completed.

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

- Activate mouse. Makes the mouse itself effective.

FAT File System

File Allocation Table (FAT) is a computer file system architecture and a family of industry-standard file systems utilizing it. The FAT file system is a continuing standard which borrows source code from the original, legacy file system and proves to be simple and robust. It offers useful performance even in lightweight implementations, but cannot deliver the same performance, reliability and scalability as some modern file systems. It is, however, supported for compatibility reasons by nearly all currently developed operating systems for personal computers and many mobile devices and embedded systems, and thus is a well-suited format for data exchange between computers and devices of almost any type and age from 1981 up to the present[1].

The file allocation table(FAT) records which disk blocks the file occupies. According to the FAT table, the contents of the file can be read out. FAT is a chain of connections to each block of a file. Such a chain may be disconnected, so FAT information is stored in two copies on the disk.

FAT Data Structure

---

[1]`https://en.wikipedia.org/wiki/File_Allocation_Table`

```
1  struct FILEINFO
2  {
3    unsigned char name[8];   /* file name */
4    unsigned char ext[3];    /* extend name of file */
5    unsigned char type;      /* file attributes */
6    char reserve[10];        /* reserve byte */
7    unsigned short time;     /* the time for storing file */
8    unsigned short date;     /* the date for storing file */
9    unsigned short  clustno; /* the file from which sector on the disk is
   ↪  stored */
10 };
```

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

FAT Functions

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

- Read file allocation table. The first FAT table is stored at 0x000200. The fat table is compressed. Must be unzipped to *fat* to use.

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

- Read file contents into *buf* memory according to *fat*, *size* is the size of file, *clustno* is the number of the first disk block that the file occupies.

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

- Search for files based on provided *name*. The file name is stored at 0x002600.

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

The application is directly facing the API. The RongOS system has more than 20 APIs. For the API more is introduced.

```
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, *xsiz* is the width. *ysiz* is height, *col_inv* is the color of window, *title* is the window name.

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

- Display string *str* on window, *x* and *y* indicate where the string is displayed, *col* is the color of string, *str* is the address of the string to display.

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

- Draw a box on the window. *x0* and *y0* are the starting point of the drawn box. *x1* and *y1* are the end point of the drawn box.

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

- Initialize the structure of the management memory

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

- Allocating memory for the application, *size* is the size of the request. Returns the allocated starting address.

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

- Free up memory used by the application. *addr* is the starting address of the release, *size* is the size of the free.

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

- Draw a pint on *win* window. *x* and *y* indicate the coordinates of the drawing, *col* is the color of the depicted point.

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

- Refresh the *win* window, *x0* and *y0* indicates the starting coordinate of the drawing, *x1* and *y1* indicates the end of the drawing.

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

- Draw a straight line on *win* window, *x0* and *y0* indicates the starting coordinate of the drawing, *x1* and *y1* indicates the end of the drawing.

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

- Close *win* window.

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

- Accept keyboard input, *mode* indicates whether the keyboard is sleeping.

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

- Get timer.

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

- Set the *data* of *timer* for sending.

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

- Set the *time* of *timer*.

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

- Release the *timer*.

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

- Make a buzzer sound.

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

- Open file based on *fname*, return file hanlder.

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

- Close the *fhandle* file.

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

- Locate the *fhandle* file, *offset* indicates the size of the offset, *mode* indicates whereto start positioning. 0 represents the size of the file, 1 from the beginning of the file, 2 from the end of the file.

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

- Return the size of *fhandle* file. The meaning of *mode* is the same as above.

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

- Read the *fhandle* file to *buf*, return read bytes in *eax* register, *maxsize* indicates the maximum number of bytes that can be read at a time.

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

- Return the context of command line in *buf*, *maxsize* indicates how many bytes can be stored.

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

- Return the language mode in *eax*.

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*.

1. *color*: After running the color command on the command line, a window with many colors appears.

2. *sundial*: This is an application used for timing. After running, it will start timing in seconds.

3. *starts2*: This application will imitate the distribution of stars in the sky.

3 Implementation

3.1 Boot Up

3.1.1 Boot Loader

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 ofboot information hit. Interrupt *0x10* is used for showing a character on the screen.

    The number of interrupt functions used in *RongOS* and their register parameters are shown in table 3-1.

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

Table 3-1 *RongOS* interrupt calls

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:

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.

3.1.2   Mode Switch

One of the purposes of developing RongOS is to make it support 32-bit mode. When the CPU is started, it is in real mode, that is, 16-bit mode. Writes out *jmp 0x2c00* at the end of the IPL. This means that after the IPL is loaded, it will jump to the *asmhead* execution. *asmhead* will implement the mode conversion and jump to the operating system's own code to begin execution. This mainly includes the following steps.

Result: Read two sides of one track
1 ENTRANCE: call `readloop()`;
2 **Procedure** `readloop()`
3   clear the times of failed to 0, $si \leftarrow 0$;
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, $si \leftarrow si + 1$;
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, $cl \leftarrow cl + 1$;
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, $dh \leftarrow dh + 1$, 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

1. The CPU cannot interrupt when it performs mode conversion, so first it is necessary to disable the interrupt. Forbidden to interrupt is actually to set the corresponding register to 1. See the code A.1.5 for details.

2. When initially having a computer, the CPU has only 16-bit mode, so the maximum memory is only 1MB. Later, the memory became larger, but in order to be compatible with the previous operating system, the circuit was limited to use only 1 MB of memory before activating the instruction. So the next step is to make the A20 address valid.

The A20GATE signal line can be turned ON by instructing the keyboard control circuit's auxiliary port output 0xdf. This accessory port is connected to many places on the motherboard. Different control functions can be achieved by inputting different values to this port. Since the circuit is to be controlled by the keyboard, the keyboard control circuit must be enabled. See the code for details A.1.6.

Start

$si = msg\_init$

$al = [si]$

$si = si + 1$

$al = 0?$ — no → $int\ 0x10$

yes

$je\ load$

$1 initrigster$

$si = 0$
$2 inirigster$

$int\ 0x13$ ← $reset$

$carry?$ — yes → $si = si + 1$ → $si > 5?$ — yes → $errStop$

no   no

$jnc\ next$
$resizeme$
$cl = cl + 1$

$cl < 18?$ — yes

no

$cl = 1$
$dh = dh + 1$ → $dh < 2?$ — no → $dh = 0$
$ch = ch + 1$ → $ch < CYLS?$

yes   no

$corStop$

yes
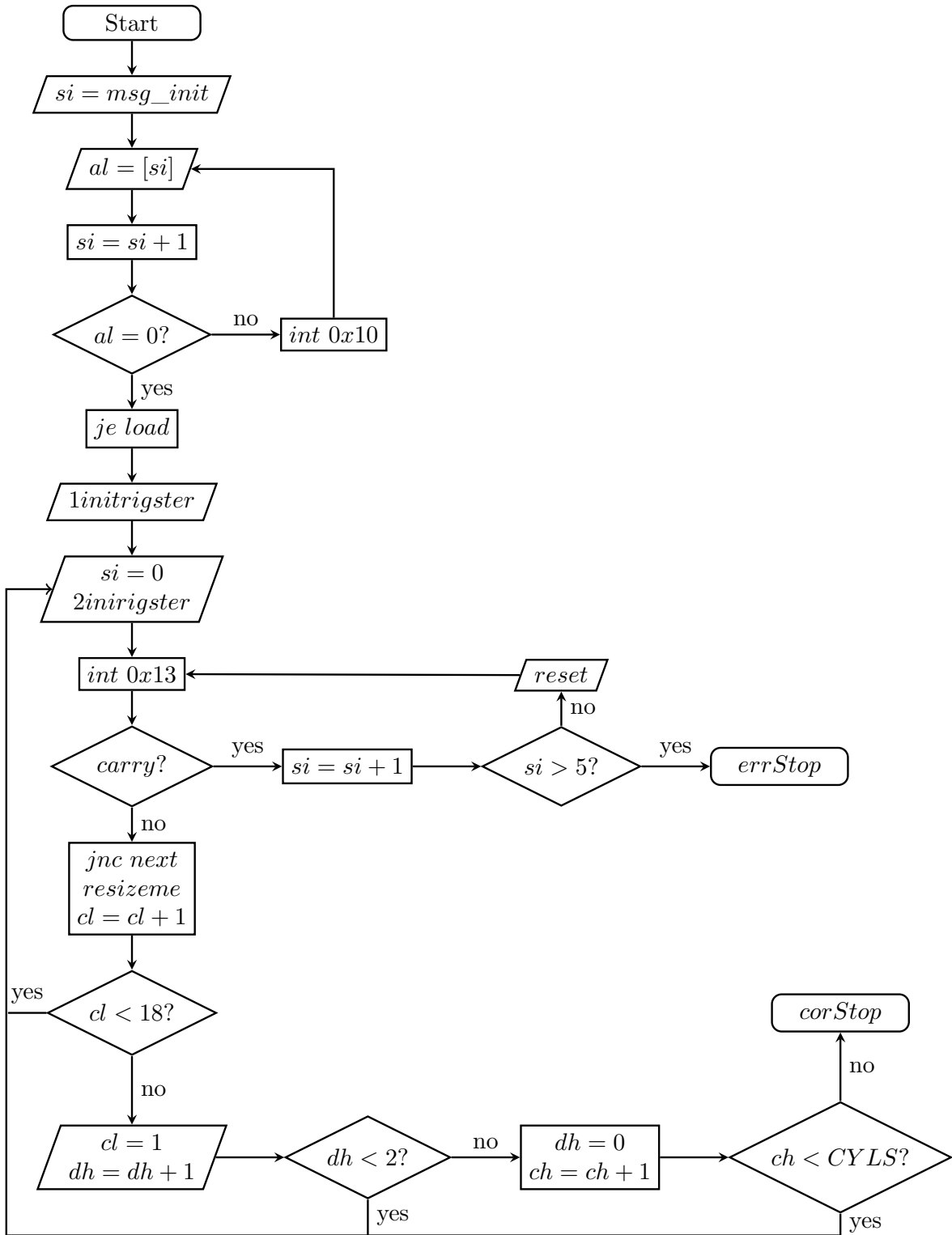
Fig. 3-2   the working flowchart of boot loader

3. Next is the mode switch. This only needs to modify the least significant bit of the *cr0* register to 1. However, RongOS does not support paging, so the highest bit is set to 0, and paging is prohibited. However, the *cr0* register cannot be operated directly, so the value must be moved to the *eax* register. Specific code implementation, see A.1.7. After switching to protected mode, the interpretation of the registers changes, so their values need to be reset.

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 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
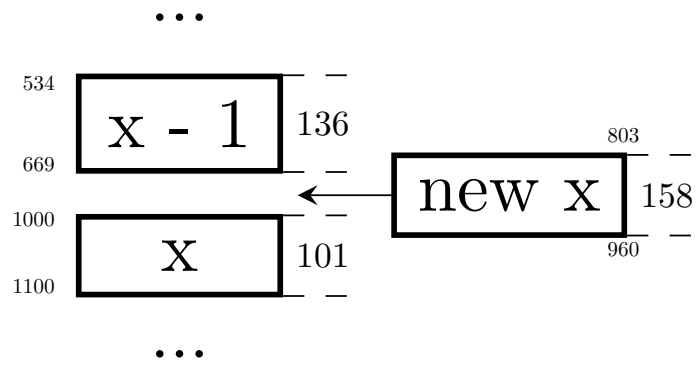
Fig. 3-3   insert an entry no merge
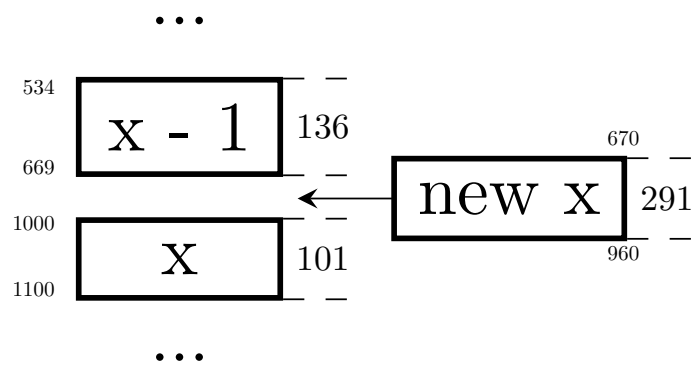
Fig. 3-4   insert an item merged with the previous item

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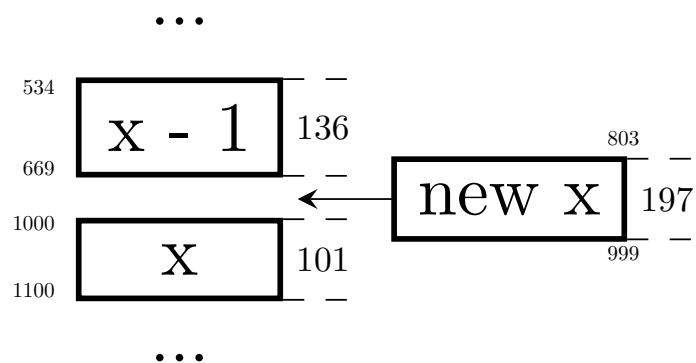
Fig. 3-5   insert an item merged with the latter item

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. Fig. 3-8 clearly shows how all the tasks in the system are managed.

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. Fig.   3-10 is the algorithm of task scheduling and switching.

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.

3.2.3   Graphic Management

In modern operating systems, graphics management is often not a concern of the operating system kernel. However, as a simple multitasking 32-bit operating system with windows it is necessary to demonstrate the implementation of a graphical management subproblem. The issue that was selected for detailed discussion was the more complex *sheet* height settings in the graphics management section.

Sheet Height Setting

There are many *sheet* on the computer screen. Which *sheet* is located at which height requires careful design and detailed consideration. When the *sheet* height is set, the height of other layers must be taken into account. When the setting is complete, the relative height of other *sheet* cannot be changed. First, the algorithm for setting the height of the *sheet* is given in Fig. 3-11.

The height of passing in may not be compliant, for example, pass a negative number and so on. Therefore, it is necessary to check the height value first. If it does not meet the regulations, it needs to be corrected. *Sheet* height settings can be divided into four sections.

1. The original height of the *sheet* is higher than the newly transmitted height. The *sheet* between the newly transmitted height and the original height needs to be moved upward by a height. After the move is complete, place the original height *sheet* on the new height. Fig. 3-12 shows this situation. The following explanations are necessary to understand this diagram. These explanations are also used for the next three cases. 1, each horizontal line represents a *sheet*, the label on the right represents the height of the layer. 2, the old and new heights are mark done the left. 3, the label next to the arrow represents the order of movement. 4, the bottom dashed line represents the hidden *sheet*, while the topmost dashed line represents the highest *sheet*.

2. The original height of the *sheet* is higher than the newly transmitted height. However, the height of the new pass is less than 0. This means that the *sheet* at old height needs to be hidden, the so-called hiding is to set its height to -1. *Sheet* above this height need to be moved down one height. Fig. 3-14 shows this situation.

3. The original height of the *sheet* is less than the newly transmitted height. The *sheet* between the newly transmitted height and the original height needs to be moved down by a height. After the move is complete, place the original height *sheet* on the new height. Fig. 3-13 shows this situation.

4. The original height of the *sheet* is less than the newly transmitted height. However, the height of the new pass is greater than top. This means that the *sheet* at old height needs to be displayed, the so-called displayed is to set its height larger than 0. *Sheet* above this new height need to be moved upward one height. Fig. 3-15 shows this situation.

Specific code implementation, see A.2.10.

3.2.4   Driver

The following parts of the driver are implemented.

Palette

The palette setting needs to write the color number into the palette register. There are two main palette registers, one is register write 0x3c8, and the other is data port 0x3c9. The

palette settings are divided into the following steps.

1. Masking interrupts first.

2. Write the number of the palette to be set to 0x3c8. Finally write 0x3c9 in order of R, G, and B.

3. Allow interrupts.

Specific code implementation, see A.2.11.

PIC

In design, the CPU can only handle one interrupt. The PIC monitors the eight signals at the input pins. As soon as an interrupt signal comes in, the only one is turned on and notified to the CPU.

The primary PIC is directly connected to the CPU, and the slave PIC is connected to the primary PIC. The master PIC is responsible for handling the 0th to 7th interrupt signals, and the slave PIC is responsible for handling the 8th to 15th interrupt signals. So the PIC settings can be divided into the following steps.

1. Disable interrupts.

2. Setting 0 to 7 interrupts is accepted by the main PIC.

3. Setting 8 to 15 interrupts is accepted by the main PIC.

4. Resume interruption.

Specific code implementation, see A.2.12.

Keyboard and Mouse

For the mouse and keyboard to work, instructions must be sent to the appropriate circuits and devices. But the mouse circuit is included in the keyboard circuit. Therefore, only the keyboard circuit needs to be initialized for the circuit. The mouse device needs to be activated additionally. The control circuit needs to be effective first, otherwise the device sending data to the control circuit will have unpredictable consequences.

What command to send, where to send please refer to the specific code implementation A.2.13, these are specified by the hardware.

3.2.5   FAT system

The related implementation of the FAT part focuses on the decompression of fat, the reading and searching of files.

Read Fat

The storage FAT table itself also needs a certain amount of space. So in order to save space, FAT data is compressed. The compression method is as follows.

$$1, \; F0 \; FF \; FF \rightarrow FF0 \; FFF$$

$$2, \; ab \; cd \; ef \rightarrow dab \; efc$$

Equation 2 reflects the general exchange rules, and Equation 1 reflects an exchange example. Three bytes before compression and only two bytes after compression. Specific code implementation, see A.2.14.

Load File

File reading is divided into two cases. First, the block it occupies is full, in this case the entire block is copied, the other is that the file is not full of one block. This requires that the end point of the copy be determined based on the size of the file. When reading a file, the block number also moves downwards. This algorithm is relatively simple, please refer to A. 2.15.

Search File

The search file is based on the file name. In *RongOS* systems, only file names shorter than 8 are supported temporarily. So the file search function first needs to detect the length of the file name. The status of the file also needs to be checked. The deleted file cannot be returned. Specific code implementation, see A.2.16.

3.3   API

There are more than twenty APIs for this system. Their implementation is much the same, they are all stored in the register parameter value, *edx* register to save the function number, and then trigger the 0x40 interrupt by the operating system kernel according to the function number to choose which system call to provide services. If there is a return value, it is stored in the *eax* register. So the implementation section only focuses on the first API: *void api_putchar(int c)*.

The function number of this API is 1, it is stored in the *edx* register. Character encoding is variable *c*, it is stored in the *al* register. Then trigger 0x40 interrupt. Specific code implementation, see A.3.

3.4   APPs

Obviously, application development is not the content of operating system development. But in order to verify that *RongOS* is able to run programs developed in accordance with the *RongOS* API, I decided to introduce an application implementation that is *stars2*. This application is to imitate the stars in the sky, I think the stars are a symbol of dreams, and I intend to end my thesis by showing dream. And developing an operating system is one of my dreams. The paper ends but the dream will not.

This application first initializes the memory management structure. Then apply for memory space. Then open a window. All operations are done by calling the API of *RongOS*. The coordinates of the point are generated by a random function *rand* of C. Then call the *api_point* API to draw a point. Finally wait for the user to enter any key to end the display. The operating result is shown in the Fig. 3-17.

How a RongOS application specifically requests operating system services through the API is shown in Fig. 3-18. Specific code implementation of this application, see A.4

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;
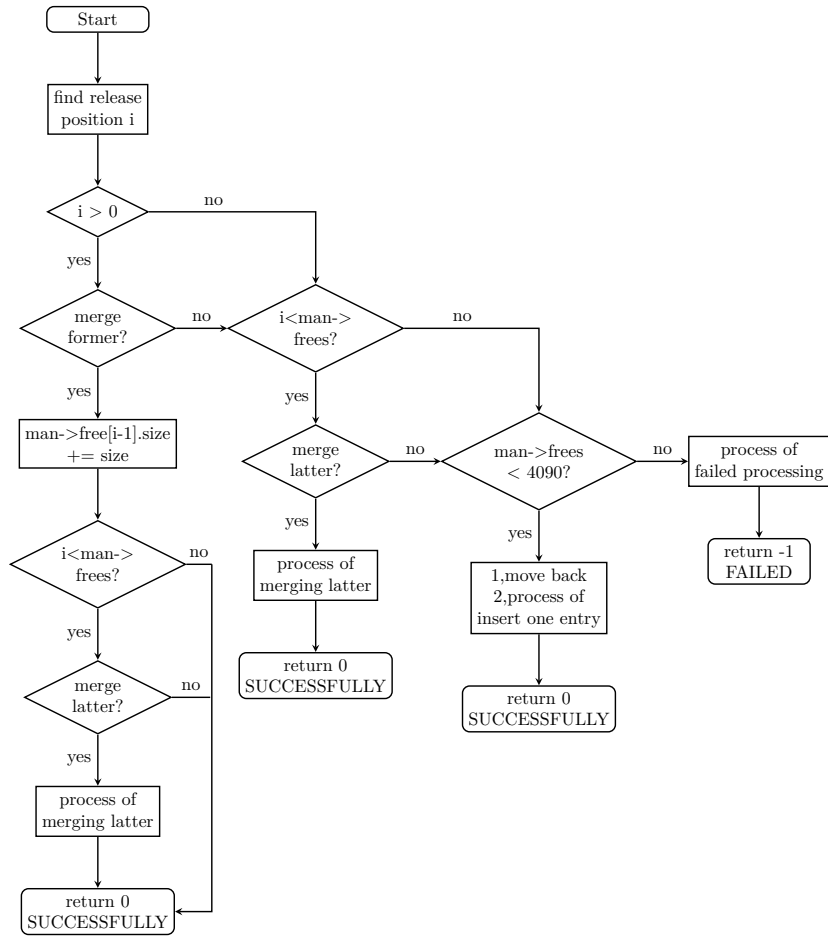
Fig. 3-6   algorithm of release memory
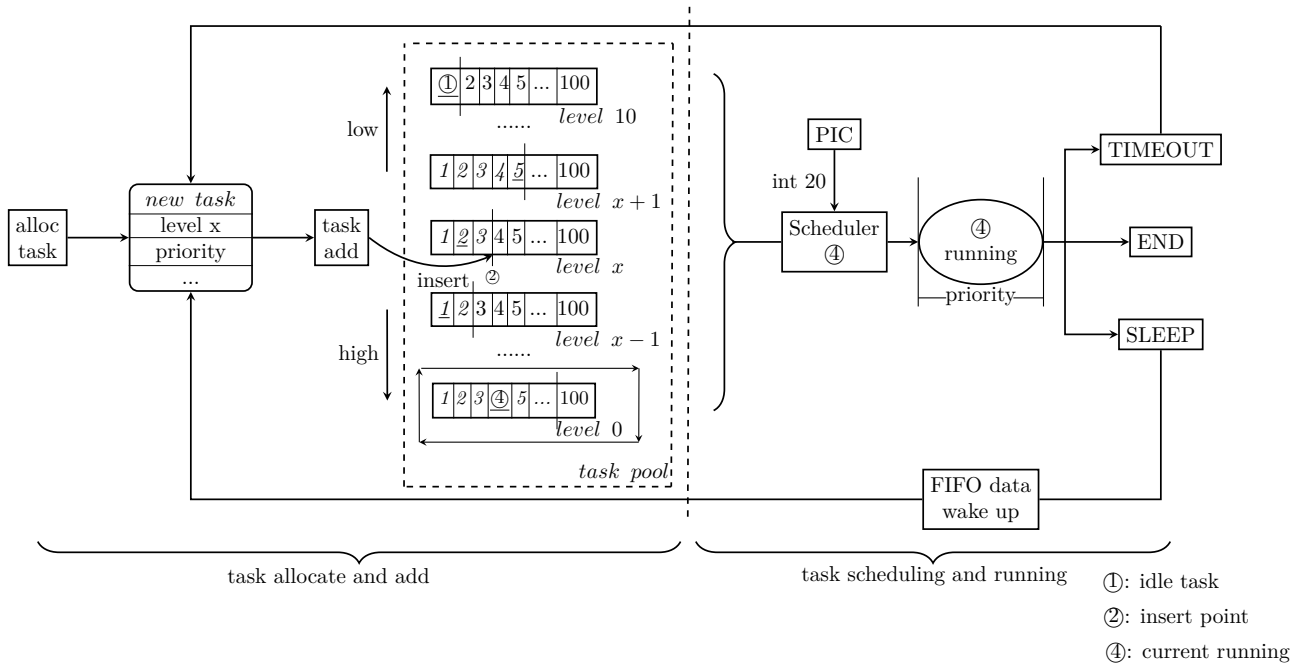
Fig. 3-7   flow chart of relase memory

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, $level \leftarrow task->level$;

5   **else**

6 | *nothing to do here*;

7   **end**

8   **if** *priority > 0* **then**

9 | Change the priority, $task->priority \leftarrow priority$;

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, $task->level \leftarrow level$;

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

Result: Schedule a task to run

1 ENTRANCE: call taskSwitch();

2 **Procedure** taskSwitch()

3   find out what task is running *now_task* in the current system based on *now_lv* and *now*;

4   add the *now* pointer to 1 for the next task scheduling;

5   **if** *if the* now *pointer equals the number of* running *tasks* **then**

6 | zero the *now* pointer;

7   **else**

8 | *nothing to do here*;

9   **end**

10   **if** *lv_change is not 0* **then**

11 | call taskSwitchSub();

12 | get the currently running layer;

13   **else**

14 | *nothing to do here*;

15   **end**

16   get the *new_task* from the currently running level at *now* pointer;

17   set the timer of the task according to the priority of the task;

18   **if** *new_task != now_task* **then**

19 | jump to *new_task* run;

20   **else**

21 | *nothing to do here*;

22   **end**

23 **Procedure** taskSwitchSub()

24   **for** $i \leftarrow 0$ **to** 9 **by** 1 **do**

25   **if** *running task of i level > 0* **then**

26 | break;

27   **else**

28 | *nothing to do here*;

29   **end**

30   **end**

31   modify the current running level *now_lv* to i;

32   modify the *lv_change* to 0, so the next time don't need to call taskSwitchSub();

Fig. 3-10   task scheduling

Result: Set the height of a sheet

1 ENTRANCE: call `sheetMove(`*(struct SHTCTL *ctl, struct SHEET*sht, int height)*`)`;

2 **Procedure** `sheetMove(`*struct SHTCTL *ctl, sturct SHEET *sht, int height*`)`

3   old ← the original height of *sht*;

4   **if** *height > top sheet plus 1* **then**

5   │ height ← top + 1;

6   **else**

7   │ *nothing to do here*;

8   **end**

9   **if** *height < -1* **then**

10   │ this means to hide *sht* sheet, height ← -1;

11   **else**

12   │ *nothing to do here*;

13   **end**

14   **if** *old > height* **then**

15   │ **if** *height >= 0* **then**

16   │ │ **for** *h ← old* **to** *height* **by** 1 **do**

17   │ │ │ increases the height of all sheets between *height* and *old* by one;

18   │ │ │ modify the height of all sheets to *h*;

19   │ │ **end**

20   │ │ place the *sht* sheet at *height*;

21   │ **else**

22   │ │ **if** *highest sheet top > old* **then**

23   │ │ │ **for** *h ← old* **to** *top* **by** 1 **do**

24   │ │ │ │ drop all sheets between *old* and *top* by one height;

25   │ │ │ │ modify the height of all sheets to *h*;

26   │ │ │ **end**

27   │ │ **else**

28   │ │ │ *nothing to do here*;

29   │ │ **end**

30   │ │ the displayed sheet is reduced by one, top ← top - 1;

31   │ **end**

32   │ refersh screen;

33   **else**

34   │ **if** *old < height* **then**

35   │ │ **if** *old >= 0* **then**

36   │ │ │ **for** *h ← old* **to** *height* **by** 1 **do**

37   │ │ │ │ drop all sheets between *old* and *height* by one height;

38   │ │ │ │ modify the height of all sheets to *h*;

39   │ │ │ **end**

40   │ │ │ place the *sht* sheet at *height*;

41   │ │ **else**

42   │ │ │ **for** *h ← top* **to** *height* **by** 1 **do**

43   │ │ │ │ increases the height of all sheets between *top* and *height* by one;

44   │ │ │ │ modify the height of all sheets to *h+1*;

45   │ │ │ **end**

46   │ │ │ place the *sht* sheet at *height*;

47   │ │ │ the displayed sheet is increased by one, top ← top + 1;

48   │ │ **end**

49   │ │ refersh screen;

50   │ **else**

51   │ │ *nothing to do here*;

52   │ **end**

53   **end**

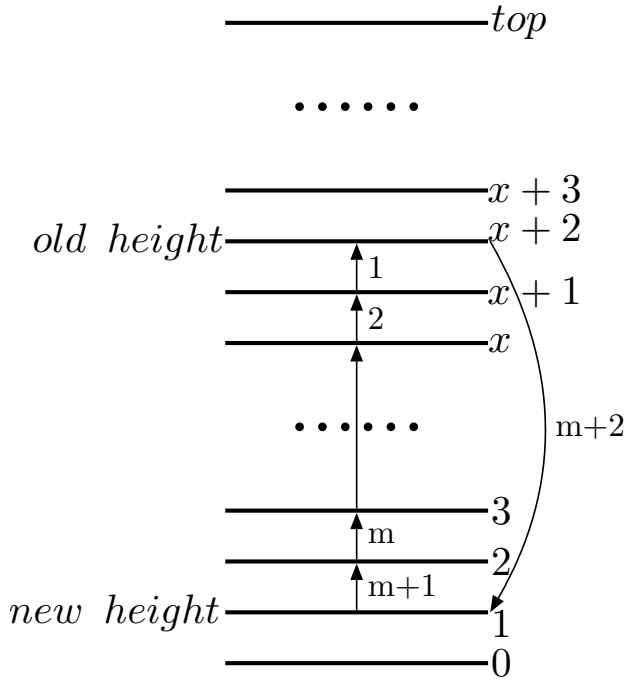Fig. 3-11   algorithm of sheet height setting

Fig. 3-12   height setting one
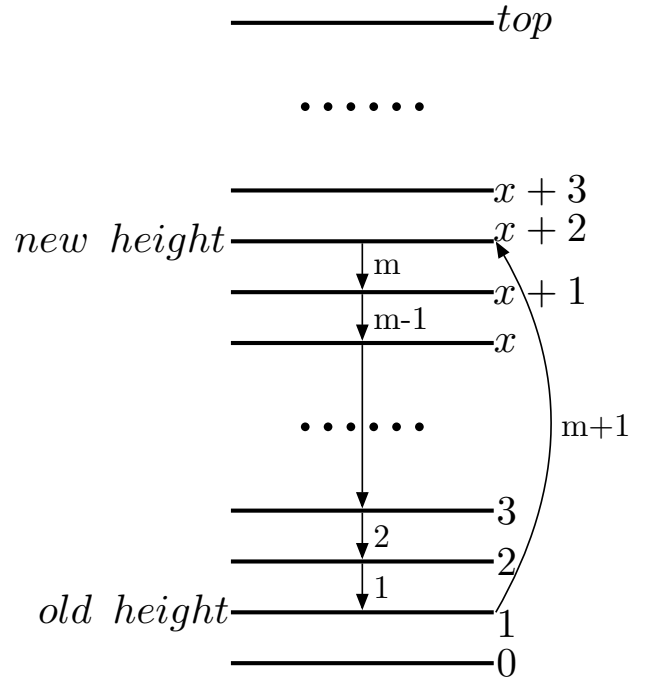

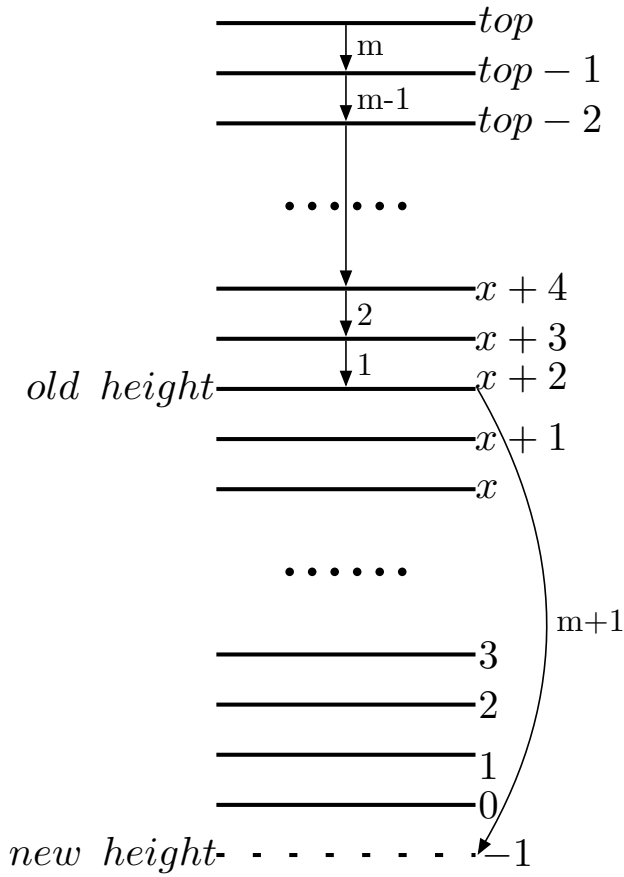Fig. 3-13   height setting three

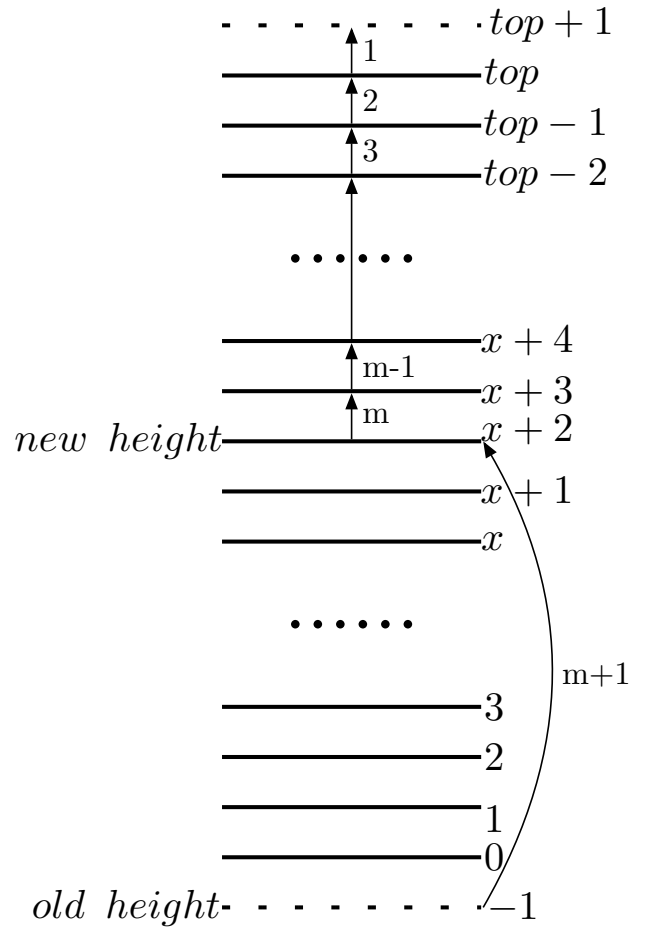Fig. 3-14   height setting two

Fig. 3-15   height setting four

Fig. 3-16   all four setting height conditions
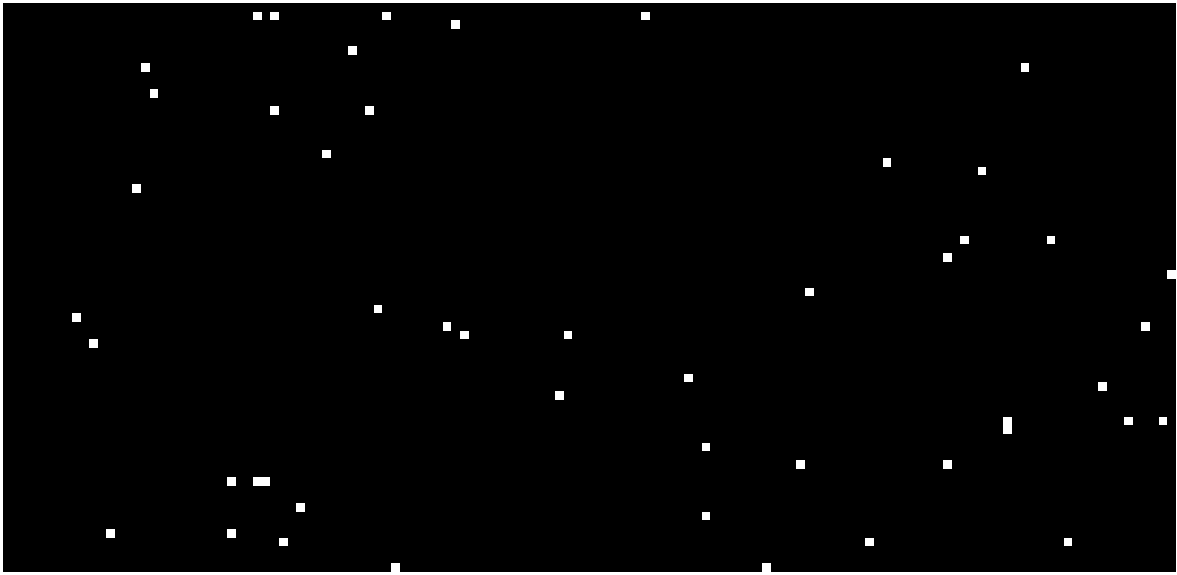
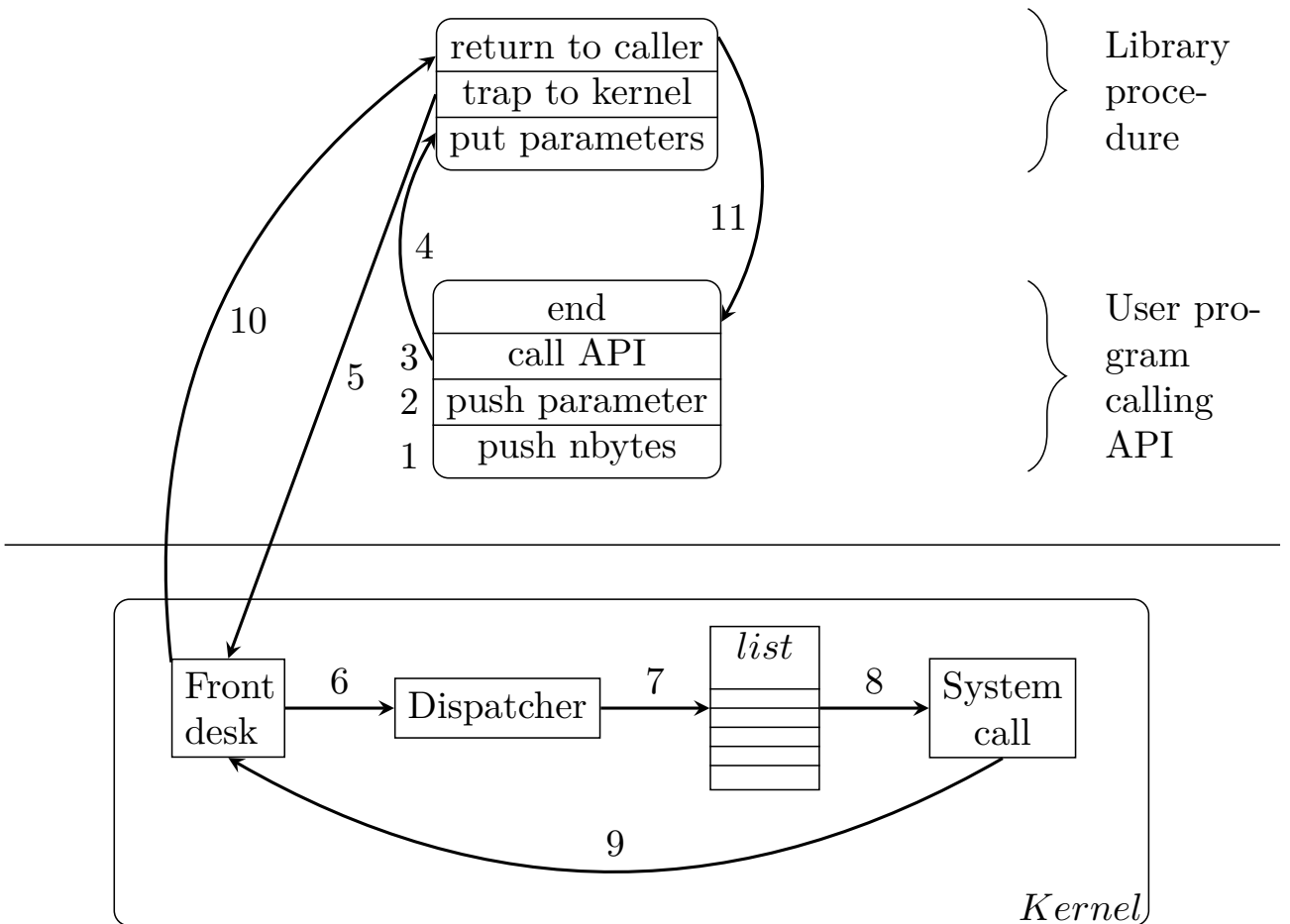Fig. 3-17   results of stars2

Fig. 3-18   the process of an application requesting a kernel service

4 Conclusions

This project intends to develop a simple operating system from scratch. The so-called start from scratch means that directly facing the hardware device does not depend on any existing three-party software. The main work to be done is to complete the loading of the operating system, memory management, and task management.

The purpose of this project is to develop a simple 32-bit multitasking operating system with windows. All these goals have been achieved in this development.

The results of this operating system development test mainly rely on the QEMU simulator. Decide on how to modify the code based on whether the results given by the QEMU simulator are as expected.

Developing an operating system is not an easy task. Many of its modules are interrelated. The relationship between these modules is complicated. The development of operating systems should adopt a modular approach. A complete and robust module is necessary to develop a module that depends on it. Process management should be the most complex part of developing an operating system. Despite this, the development of this part has a particularly large impact on the performance of the entire system. The illustration of these abstract concepts is helpful for an in-depth understanding of the various modules and their relationships.

For future research, consider more services that the operating system should have provided. For example, provide network services. Of course, based on the concept of starting from scratch, you can also have your own ideas on how to implement the TCP/IP protocol. The file system is not perfect. It can even be said that there is basically no file system in this system. The reading and writing of hard disks and memory cards have not been able to achieve support. Implementing virtual memory is also a major issue. Almost all modern operating systems support virtual memory. In short, despite a lot of work, this simple operating system can only be more of a demonstration program.

I see no ending, yet high and low I'll search with my will unbending.

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 Up

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.1.5   Disable interrupts

```
75  ; make sure that PIC does not accept any interrupts
76  ; with AT compatible machine specifications, if you initialize PIC.
77  ; Hang up occasionally if you do not do this before CLI.
78  ; Initialize PIC later.
79          mov al,0xff
80          out 0x21,al ; io_out(PIC0_IMR, 0xff), all interrupts of the
            ↪   primary PIC are prohibited.
81          nop ; if the out command is executed continuously, some models
            ↪   will not work normally.
82       out 0xa1,al ; io_out(PIC1_IMR, 0xff), all interrupts of the slave PIC
         ↪   are prohibited.
83          cli ; in addition, interrupt disabled even at CPU level.
```

A.1.6   Enable A20

```
85  ; set A20GATE so that CPU can access 1 MB or more of memory.
86  ; Send 0xdf to the slave port of the keyboard control circuit. This port
    ↪   is connected to the motherboard.
```

```
87   ; so enable A20GATE.
88           call waitkbdout ; wait_KBC_sendready(), send commands to the
             ↪   keyboard and wait for the keyboard to be ready.
89           mov al,0xd1 ;
90           out 0x64,al ; io_out8(PORT_KEYCMD, KEYCMD_WRITE_OUTPORT)
91           call waitkbdout ; wait_KBC_sendready()
92           mov al,0xdf ; enable A20
93           out 0x60,al
94           call waitkbdout
```

A.1.7   Mode switch

```
98
99           LGDT        [GDTR0] ; set interim GDT
100          mov eax,cr0
101          and eax,0x7fffffff ; set bit 31 to 0(because paging is prohibited)
102          or eax,0x00000001 ; set bit 0 to 1(for protect mode transition)
103          mov cr0,eax ; Until now, the interpretation of the segment
             ↪   register is no longer 16 times, but the use of GDT.
104          jmp pipelineflush ; after switch mode, jmp must.
105  pipelineflush:
106          mov ax,1*8 ; readable / writable segment 32 bits. The value of all
             ↪   segment registers changed from 0000 to 0008 in addition to cs,
             ↪   also gdt + 1.
107          mov ds,ax
108          mov es,ax
109          mov fs,ax
110          mov gs,ax
111          mov ss,ax
```

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  }
```

A.2.10   Graphic height setting

```
250  void sheet_updown(struct SHEET *sht, int height)// insert the sheet at the
     ↪  location of height
251  {
252    struct SHTCTL *ctl = sht -> ctl;
253    int h, old = sht -> height; // remember the height before setting
254
255    // if the height is too low or too high, fix it.
256    if (height > ctl -> top + 1)
257      {
258        height = ctl -> top + 1;
259      }
260    if (height < -1)
261      {
262        height = -1; // -1, hidden the sheet
263      }
264    sht -> height = height; // set height
265
266    //the following are mainly sort of sheets[]
267    if (old > height) // it gets lower than before
268      {
269        if (height >= 0) //still showing, pull up
270          {
271            for (h = old; h > height; h--)
272              {
273                ctl -> sheets[h] = ctl -> sheets[h - 1];
274                ctl -> sheets[h] -> height = h;
275              }
276            ctl -> sheets[height] = sht;
277            sheet_refreshmap(ctl, sht -> vx0, sht -> vy0, sht -> vx0 + sht
               ↪  -> bxsize, sht -> vy0 + sht -> bysize, height + 1);
```

```
278            sheet_refreshsub(ctl, sht -> vx0, sht -> vy0, sht -> vx0 + sht
               ↪ -> bxsize, sht -> vy0 + sht -> bysize, height + 1, old);
279          }
280      else // hide
281        {
282          if (ctl -> top > old)// pull down
283            {
284              for (h = old; h < ctl -> top; h++)
285                {
286                  ctl -> sheets[h] = ctl -> sheets[h + 1];
287                  ctl -> sheets[h] -> height = h;
288                }
289            }
290          ctl -> top--; //since the number of sheets decrease by one, the
               ↪ height at the top decreases
291          sheet_refreshmap(ctl, sht->vx0, sht->vy0, sht->vx0 +
               ↪ sht->bxsize, sht->vy0 + sht->bysize, 0);
292          sheet_refreshsub(ctl, sht -> vx0, sht -> vy0, sht -> vx0 + sht
               ↪ -> bxsize, sht -> vy0 + sht -> bysize, 0, old - 1);

294        }
295    }
296  else if (old < height) // it gets higher than before
297    {
298      if (old >= 0) // pull down
299        {
300          for (h = old; h < height; h++)
301            {
302              ctl -> sheets[h] = ctl -> sheets[h + 1];
303              ctl -> sheets[h] -> height = h;
304            }
305          ctl -> sheets[height] = sht;
306        }
307      else // from non-display state to display state
308        {
309          for (h = ctl -> top; h >= height; h--)// lift up
310            {
311              ctl -> sheets[h + 1] = ctl -> sheets[h];
312              ctl -> sheets[h + 1] -> height = h + 1;
313            }
314          ctl -> sheets[height] = sht;
315          ctl -> top++; // since the number of  being displayed increases
               ↪ by one, the height at the top increases
316        }
317      sheet_refreshmap(ctl, sht->vx0, sht->vy0, sht->vx0 + sht->bxsize,
         ↪ sht->vy0 + sht->bysize, height);
318      sheet_refreshsub(ctl, sht -> vx0, sht -> vy0, sht -> vx0 + sht ->
         ↪ bxsize, sht -> vy0 + sht -> bysize, height, height);
319    }
320
```

```
321     return;
322   }
```

A.2.11   Palette setting

```
3   void init_palette(void)
4   {
5         static unsigned char table_rgb[16 * 3] =
6                  {
7                          0x00, 0x00, 0x00,
8                          0xff, 0x00, 0x00,
9                          0x00, 0xff, 0x00,
10                         0xff, 0xff, 0x00,
11                         0x00, 0x00, 0xff,
12                         0xff, 0x00, 0xff,
13                         0x00, 0xff, 0xff,
14                         0xff, 0xff, 0xff,
15                         0xc6, 0xc6, 0xc6,
16                         0x84, 0x00, 0x00,
17                         0x00, 0x84, 0x00,
18                         0x84, 0x84, 0x00,
19                         0x00, 0x00, 0x84,
20                         0x84, 0x00, 0x84,
21                         0x00, 0x84, 0x84,
22                         0x84, 0x84, 0x84
23                 };
24
25         unsigned char table2[216 * 3];
26     int r, g, b;
27     set_palette(0, 15, table_rgb);
28     for (b = 0; b < 6; b++)
29       {
30         for (g = 0; g < 6; g++)
31           {
32             for (r = 0; r < 6; r++)
33               {
34                 table2[(r + g * 6 + b * 36) * 3 + 0] = r * 51;
35                 table2[(r + g * 6 + b * 36) * 3 + 1] = g * 51;
36                 table2[(r + g * 6 + b * 36) * 3 + 2] = b * 51;
37               }
38           }
39       }
40     set_palette(16, 231, table2);
41
42         return;
43   }
44
45
46   void set_palette(int start, int end, unsigned char *rgb)
47   {
```

```
48          int i, eflags;
49
50          eflags = io_load_eflags();
51          io_cli();
52          io_out8(0x03c8, start);
53
54          for (i = start; i <= end; i++)
55                  {
56                          io_out8(0x03c9, rgb[0] / 4);
57                          io_out8(0x03c9, rgb[1] / 4);
58                          io_out8(0x03c9, rgb[2] / 4);
59                          rgb += 3;
60                  }
61
62          io_store_eflags(eflags);
63          return;
64  }
```

A.2.12   PIC

```
6   void init_pic(void)
7   {
8     io_out8(PIC0_IMR, 0xff); // all interrupts are forbidden
9     io_out8(PIC1_IMR, 0xff);
10
11    io_out8(PIC0_ICW1, 0x11); // edge trigger mode
12    io_out8(PIC0_ICW2, 0x20); // IRQ0 - 7 received by int20 - 27
13    io_out8(PIC0_ICW3, 1 << 2); // connect PIC1 via IRQ2
14    io_out8(PIC0_ICW4, 0x01); // no buffer mode
15
16    io_out8(PIC1_ICW1, 0x11); // edge trigger mode
17    io_out8(PIC1_ICW2, 0x28); // IRQ8 - 15 received by int28 - 2f
18    io_out8(PIC1_ICW3, 2); // connect PIC1 via IRQ2
19    io_out8(PIC1_ICW4, 0x01); // no buffer mode
20
21    io_out8(PIC0_IMR, 0xfb); //11111011, all forbidden expect PIC1
22    io_out8(PIC1_IMR, 0xff); // all interrupt forbidden
23
24    return;
25  }
```

A.2.13   Keyboard and mouse

```
21  void wait_KBC_sendready(void)
22  {
23    // wait for the keyboard controller to be able to transmit data.
24    for (;;)
25      {
26        if ((io_in8(PORT_KEYSTA) & KEYSTA_SEND_NOTREADY) == 0)
27          break;
```

```
28         }
29      return;
30   }
31
32
33   void init_keyboard(struct FIFO32 *fifo, int data0)
34   {
35      // memory of FIFO buffer to be written is stored
36      keyfifo = fifo;
37      keydata0 = data0;
38
39      // initialization of keyboard controller
40      wait_KBC_sendready();
41      io_out8(PORT_KEYCMD, KEYCMD_WRITE_MODE);
42      wait_KBC_sendready();
43      io_out8(PORT_KEYDAT, KBC_MODE);
44      return;
45   }

21   void enable_mouse(struct FIFO32 *fifo, int data0, struct MOUSE_DEC *mdec)
22   {
23      // memory of FIFO buffer to be written is stored
24      mousefifo = fifo;
25      mousedata0 = data0;
26
27      // mouse effective
28      wait_KBC_sendready();
29      io_out8(PORT_KEYCMD, KEYCMD_SENDTO_MOUSE);
30      wait_KBC_sendready();
31      io_out8(PORT_KEYDAT, MOUSECMD_ENABLE);
32
33      // if it works, ACK(oxfa) will be sent.
34      mdec -> phase = 0; // waiting for mouse 0xfa.
35      return;
36   }
```

A.2.14   Read fat

```
5    void file_readfat(int *fat, unsigned char *img)
6    // decompress FAT in disk image
7    {
8       int i, j = 0;
9       for (i = 0; i < 2880; i += 2)
10        {
11          fat[i + 0] = (img[j + 0] | img[j + 1] << 8) & 0xfff;
12          fat[i + 1] = (img[j + 1] >> 4 | img[j + 2] << 4) & 0xfff;
13          j += 3;
14        }
15      return;
16   }
```

A.2.15   Load file

```
18  void file_loadfile(int clustno, int size, char *buf, int *fat, char *img)
19  {
20    int i;
21    for (;;)
22      {
23        if (size <= 512)
24          {
25            for (i = 0; i < size; i++)
26              {
27                buf[i] = img[clustno * 512 + i];
28              }
29            break;
30          }
31        for (i = 0; i < 512; i++)
32          {
33            buf[i] = img[clustno * 512 + i];
34          }
35        size -= 512;
36        buf += 512;
37        clustno = fat[clustno];
38      }
39    return;
40  }
```

A.2.16   Search file

```
42  struct FILEINFO *file_search(char *name, struct FILEINFO *finfo, int max)
43  {
44    int i, j;
45    char s[12];
46    for (j = 0; j < 11; j++)
47      {
48        s[j] = ' ';
49      }
50    j = 0;
51    for (i = 0; name[i] != 0; i++)
52      {
53        if (j >= 11)
54          {
55            return 0; // not find it
56          }
57        if (name[i] == '.' && j <= 8)
58          {
59            j = 8;
60          }
61        else
62          {
63            s[j] = name[i];
```

```
64        if ('a' <= s[j] && s[j] <= 'z')
65          {
66            // make lowercase letters capitalize
67            s[j] -= 0x20;
68          }
69        j++;
70      }
71    }
72  for (i = 0; i < max; )
73    {
74      if (finfo[i].name[0] == 0x00)
75        {
76          break;
77        }
78      if ((finfo[i].type & 0x18) == 0)
79        {
80          for (j = 0; j < 11; j++)
81            {
82              if (finfo[i].name[j] != s[j])
83                {
84                  goto next;
85                }
86            }
87          return finfo + i; // file found
88        }
89    next:
90      i++;
91    }
92  return 0; // could not find it.
93  }
```

A.3   API

```
10  _api_putchar:                        ; void api_putchar(int c)
11        mov edx, 1
12        mov al, [esp + 4]              ; c
13        int 0x40
14        ret
```

A.4   APPs

```
1  #include "../../header/apilib.h"
2
3  int rand(void);
4
5  void HariMain(void)
6  {
7    char *buf;
```

```
 8     int win, i, x, y;
 9     api_initmalloc();
10     buf = api_malloc(150 * 100);
11     win = api_openwin(buf, 150, 100, -1, "stars2");
12     api_boxfilwin(win + 1, 6, 26, 143, 93, 0);
13     for (i = 0; i < 50; i++)
14       {
15         x = (rand() % 137) + 6;
16         y = (rand() % 67) + 26;
17         api_point(win + 1, x, y, 7);
18       }
19     api_refreshwin(win, 6, 26, 144, 94);
20     for (;;)
21       {
22         if (api_getkey(1) == 0x0a)
23           {
24             break;
25           }
26       }
27     api_end();
28   }
```