

操作系统实验报告 9

蔡日骏 12348003

2014 年 6 月 20 日

1 简介

本次实验为 AssignmentOS 添加文件系统支持，包括一个简单的软盘控制器驱动程序和 FAT12 文件系统驱动程序，以及一个对底层文件操作进行封装虚拟文件系统层。此外，为了支持应用程序的加载运行，内核中添加了多个系统调用的支持。

2 实现

2.1 软盘控制器驱动程序

AssignmentOS 中实现的软盘控制器驱动程序非常简单，只实现了读取功能，且没有足够的错误恢复机制，但是足以用作演示。

简单来说，此驱动程序对软盘控制器进行操作，并通过 DMA¹进行数据访问。在进行 DMA 传输等不需要 CPU 参与操作时，驱动程序会自动挂起当前进程，内核的任务调度器会把控制器交给其他进程。

驱动程序的主要代码如下。

```
/* 初始化软盘控制器 */
bool init_fdc()
{
    int i;
    uint8_t tmp;
    IDT[0x26] = make_int_desc(0x8, (uint32_t)inthandler26);
    enable_irq(6);
    fdc_reset = false;
    outb(DATARATE_SELECT_REGISTER, 0x80); /* 重置软盘控制器 */
    while(!fdc_reset) /* 等待 IRQ6, 但调度器未初始化只能挂起 */
        hlt();
    for(i = 0; i < 4; ++i) {
        fdc_send_byte(SENSE_INTERRUPT);
        fdc_get_byte(&tmp);
    }
}
```

¹ISA DMA

```

        fdc_get_byte(&tmp);
    }
    outb(CONFIGURATION_CONTROL_REGISTER, 0); /* 设置数据传输速率 */
    fdc_send_byte(SPECIFY);                  /* 设置软盘控制器工作参数 */
    fdc_send_byte(0xCF);
    fdc_send_byte(0x6);
    outb(DIGITAL_OUTPUT_REGISTER, 0x1C);    /* 启动马达 */
    return true;
}

/* 移动磁头到指定的磁道 */
bool fdc_seek(uint8_t cylinder)
{
    bool result = true;
    uint8_t tmp;
    cli();
    fdc_send_byte(SEEK);
    fdc_send_byte(0);
    fdc_send_byte(cylinder);
    wait_event(WAIT_IRQ6); /* 等待 IRQ6, 此时调度器已初始化, 可以进程切换 */
    fdc_send_byte(SENSE_INTERRUPT);
    fdc_get_byte(&tmp);
    if(tmp & 0xC0)
        result = false;
    fdc_get_byte(&tmp);
    sti();
    return result;
}

/* 设置 ISA DMA 控制器 */
bool fdc_DMA_init(bool write, void *addr, size_t len)
{
    if((uint32_t)addr > 0xFFFFFFF) /* ISA DMA 只能在物理内存前 16M 上操作 */
        return false;
    if(len > (1 << 16))
        return false;
    --len;
    cli();
    outb(0x0A, 0x6);
    outb(0x0C, 0);
    outb(0x0B, write ? 0x4A : 0x46);

```


2.2 文件系统

AssignmentOS 从设计上支持多种不同的文件系统格式²，为了统一文件访问的操作，内核中实现了一个简易的类似虚拟文件系统的层次对下层的操作进行封装。

下面先介绍 AssignmentOS 对 FAT12 文件系统的实现。

由于 AssignmentOS 在为进程分配进程任务控制块所需的内存空间时会进行页对齐，因此实际上每个进程都会有一段多余的空间。在实现文件系统时，由于这部分多余空间正好能够保证在物理内存的前 16M 内，因此可以被利用起来作为 IO 缓冲区以减少 IO 次数，极大地提升性能。

下面的函数会检查当前进程的 IO 缓冲区中数据是否所需，如果是则直接返回，否则进行读取。

```
static bool floppy_buf_read(int lba)
{
    if(CURRENT_TASK->io_buf_valid && CURRENT_TASK->io_buf_idx == lba)
        return true;
    if(fdc_seek(LBA_C(lba)) &&
        fdc_read(LBA_C(lba), LBA_H(lba), LBA_S(lba), 1, CURRENT_TASK->io_buf)) {
        CURRENT_TASK->io_buf_valid = true;
        CURRENT_TASK->io_buf_idx = lba;
        return true;
    }
    return false;
}
```

下面的函数用于在 FAT 中查找下一个簇的簇号。

```
static inline uint16_t fat12_next_cn(int cn)
{
    int n = cn + (cn >> 1);
    if(cn & 1)
        return *((uint16_t *)&major_fat[n] >> 4);
    else
        return *((uint16_t *)&major_fat[n] & 0x0FFF);
}
```

在遍历文件夹中的文件时，使用 `fat12_ls` 函数。该函数通过一个指向目录所在簇的相对簇号和下一个文件的起始查找偏移量来查找下一个文件，并返回再下一个文件的起始查找偏移量。因此，只要重复调用 `fat12_ls` 函数即可实现目录遍历的功能。

```
int fat12_ls(int rel_cn, int offset, struct FAT_File_t *file)
{
    int sector, offset_for_offset;
    if(rel_cn == 0) { /* 根目录 */
        sector = root_dir_cn;
```

²虽然目前只实现了 FAT12

```

    if(offset)
        --offset;
    else { /* 给根目录模拟一个当前目录 '.', 其偏移量为 0 */
        file->cn = 0;
        file->filename[0] = '.';
        file->filename[1] = ' ';
        file->ext[0] = ' ';
        return 1;
    }
    offset_for_offset = (offset / n_files_per_sector) *
        n_files_per_sector + 2;
}
else {
    sector = rel_cn + rel_cn_offset;
    offset_for_offset = (offset / n_files_per_sector) *
        n_files_per_sector + 1;
}
sector += offset / n_files_per_sector; /* 目录所在绝对扇区号 */
offset %= n_files_per_sector;          /* 文件偏移量 */

floppy_buf_read(sector);

struct FAT_File_t *p = (struct FAT_File_t *)CURRENT_TASK->io_buf;
while(offset < n_files_per_sector) {
    if(p[offset].filename[0] == 0) /* 结束查找 */
        return 0;
    if(p[offset].filename[0] == (char)0xE5 || p[offset].attr == 0x0F) {
        ++offset;
    } else {
        *file = p[offset];
        return offset + offset_for_offset; /* 下一个文件起始查找偏移量 */
    }
}
return 0;
}

```

文件系统上层封装比较简单。首先定义了下面一个结构体用来表示一个文件。

```

struct File_t
{
    char filename[13]; /* 文件名, 不同于 FAT12, 这里是一个 C 字符串 */
    enum FileType type; /* 文件类型 (档案或文件夹) */
}

```

```

uint32_t size;          /* 文件大小 */
uint32_t location;      /* 文件所在位置 */
uint32_t p_location;    /* 文件所在的文件夹位置 */
uint32_t next_item;     /* 下一个文件起始查找偏移量 */
};

```

实现的函数主要有以下几个。代码比较简单，不再列出。

```

/* 打开 dir 指向的文件夹，向 sub_item 写入第一个文件的信息 */
bool open_dir(const struct File_t *dir, struct File_t *sub_item);

/* 打开字符串 path 表示的文件夹 */
bool open_dir_path(const char *path, struct File_t *sub_item);

/* 获取 item 的下一个文件 */
bool get_next_file(struct File_t *item);

/* 把 file 指向的文件读取到 dest */
size_t read_file(const struct File_t *file, void *dest);

/* 把字符串 path 表示的文件读取到 dest */
size_t read_file_path(const char *path, void *dest);

/* 根据路径字符串 path 打开文件 */
bool open_file_path(const char *path, struct File_t *file);

```

2.3 新增系统调用

用户程序不能直接访问系统的各种服务，因此，为了方便在用户程序开发，把一些基本的系统服务进行系统调用封装。主要有read、write、clock、sleep、tty_print、getpid、getppid。这些系统调用实现比较简单，不再介绍。下面主要介绍用于运行应用程序的execve 系统调用。

execve 系统调用用于加载一个程序到内存并把当前进程切换成该程序。因此，需要运行一个程序时，可以进行fork，并在子进程中执行execve 系统调用。POSIX 中的execve 系统调用支持传递命令行参数和环境变量，但 AssignmentOS 目前并不支持命令行参数和环境变量，因此 execve 系统调用只实现了最基本的程序运行功能。

```

int sys_execve(int _filename, int _argv, int _envp)
{
    const char *filename = (const char *)_filename;
    struct File_t program;
    if(!open_file_path(filename, &program))    /* 打开程序文件 */
        return -1;
}

```

```

size_t program_size = program.size,
      n_pages = ((program_size - 1) >> 12) + 1;
size_t i;
/* 申请程序需要的内存空间，并映射到相应的虚拟地址 */
for(i = 0; i < n_pages; ++i)
    map_page(CURRENT_TASK->page_dir,
              (USERSPACE_START >> 12) + i,
              alloc_phy_page(1024, max_page_num),
              false, false, true);
/* 读取程序文件到内存 */
if(!read_file(&program, (void *)USERSPACE_START))
    return -1;
CURRENT_TASK->sigint_handler = sigint_default;
/* 跳转到程序入口点 */
__asm__ volatile (
    "jmp *%0;"
    :: "r" (USERSPACE_START):
);
return 0;
}

```

2.4 应用程序加载运行

应用程序的加载运行大致为fork 然后execve。以下为 SHELL 中运行程序的run_program 函数的片段。

```

char program_filename[70];
int wd_len = strlen(CURRENT_TASK->working_dir);
/* 在程序名字前加上当前工作目录 */
strncpy(program_filename, CURRENT_TASK->working_dir, 70);
strncpy(program_filename + wd_len, program_name_buf, 70 - wd_len);

struct File_t file;
if(open_file_path(program_filename, &file)) {
    if(file.type != ARCHIVE) {
        sh_write_with_color(file.filename, 0x0C);
        sh_write_with_color(" is not a file.\n", 0x0C);
        return;
    }
}
if((pid = fork())) { /* 父进程 (即 SHELL 程序) */
    if(!background) { /* 根据需要进行 wait */
        FOREGROUND_TASK = task_list[pid];
    }
}

```

```

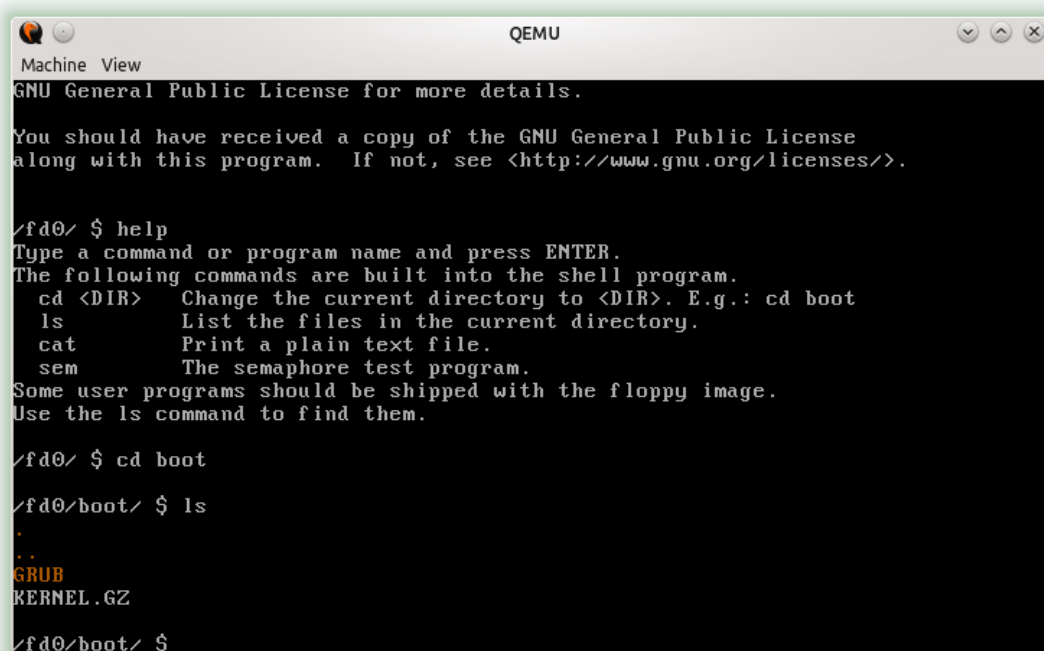
        wait(pid);
    } else {
        FOREGROUND_TASK = NULL;
    }
} else {
    /* 子进程 */
    if(execve(program_filename) != 0) /* 运行程序 */
        _exit(0);
}
return;
}

```

3 演示

3.1 目录切换与列出文件

在 SHELL 中可以通过 `cd` 文件夹名命令进行工作目录切换。运行 `ls` 命令可以列出当前工作目录中的文件，其中文件夹将会以橙色显示。这两个命令内建到 SHELL 中。演示如图。



```

Machine View
QEMU
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.

/fd0/ $ help
Type a command or program name and press ENTER.
The following commands are built into the shell program.
  cd <DIR>  Change the current directory to <DIR>. E.g.: cd boot
  ls       List the files in the current directory.
  cat      Print a plain text file.
  sem      The semaphore test program.
Some user programs should be shipped with the floppy image.
Use the ls command to find them.

/fd0/ $ cd boot

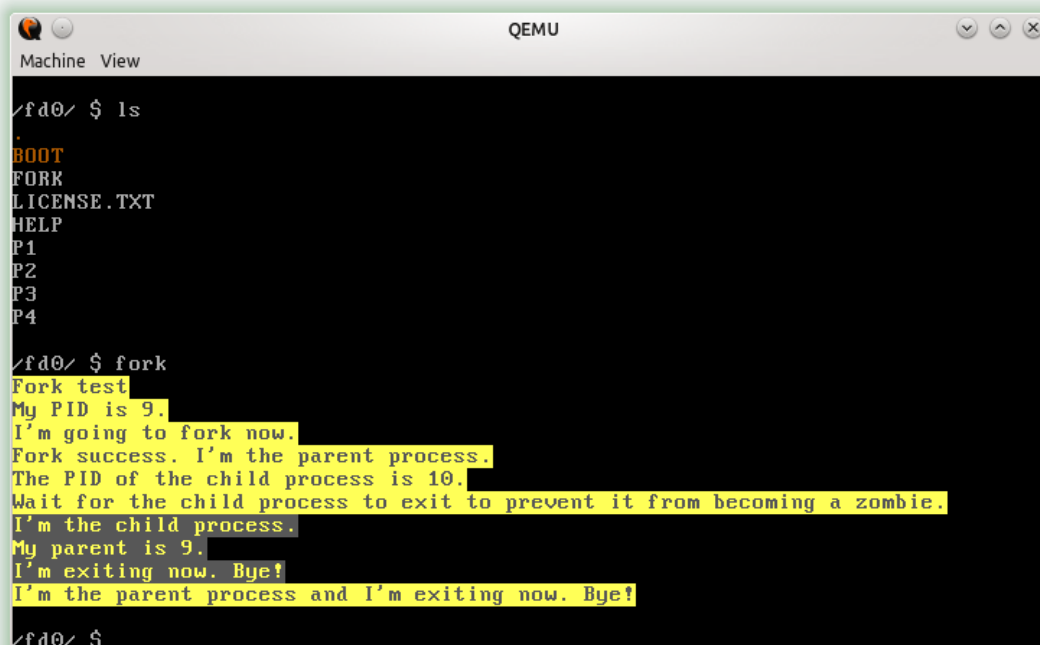
/fd0/boot/ $ ls
.
..
GRUB
KERNEL.GZ

/fd0/boot/ $

```

在软盘根目录中包含了几个 COM 格式的用户程序。可以在根目录中通过 `ls` 命令显示出来。

由于前一个实验中的信号量演示程序 `sem` 需要共享内存，但 AssignmentOS 尚未实现 `shmX` 系列系统调用，因此该演示程序仍然只能内建到内核中，通过 `sem` 命令运行。

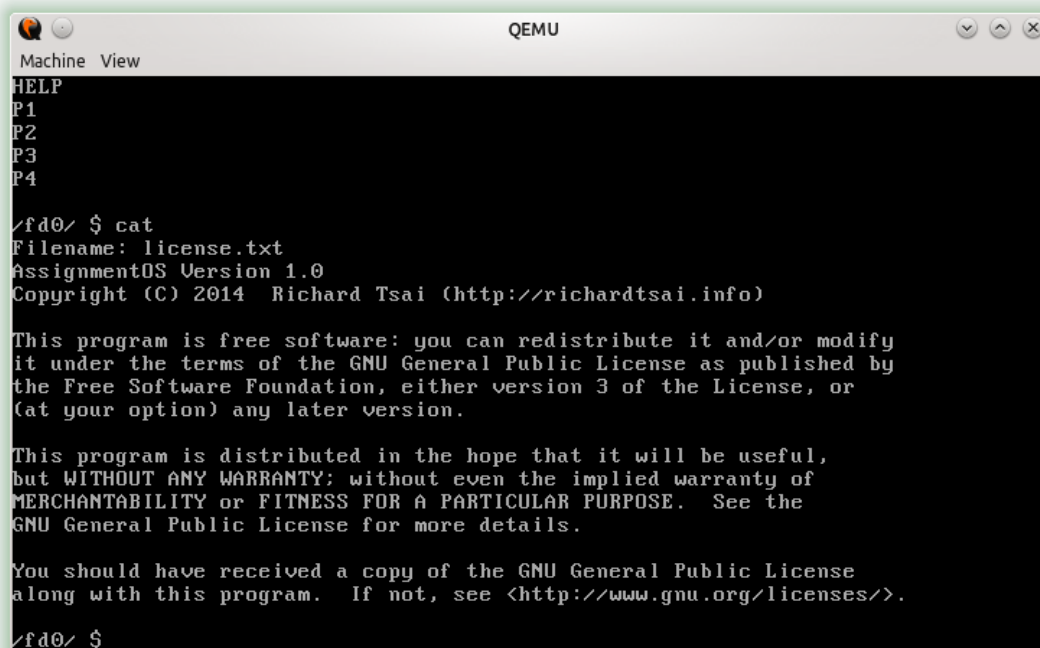


A screenshot of a QEMU terminal window. The window title is "QEMU" and it has standard window controls. The terminal shows a prompt "/fd0/ \$" followed by the command "ls". The output lists files: ".", "BOOT", "FORK", "LICENSE.TXT", "HELP", "P1", "P2", "P3", and "P4". Then, the command "fork" is entered. The output shows a "Fork test" where the parent process (PID 9) prints "My PID is 9.", "I'm going to fork now.", "Fork success. I'm the parent process.", "The PID of the child process is 10.", and "Wait for the child process to exit to prevent it from becoming a zombie." The child process (PID 10) prints "I'm the child process.", "My parent is 9.", "I'm exiting now. Bye!", and "I'm the parent process and I'm exiting now. Bye!". The terminal ends with the prompt "/fd0/ \$".

```
/fd0/ $ ls
.
BOOT
FORK
LICENSE.TXT
HELP
P1
P2
P3
P4

/fd0/ $ fork
Fork test
My PID is 9.
I'm going to fork now.
Fork success. I'm the parent process.
The PID of the child process is 10.
Wait for the child process to exit to prevent it from becoming a zombie.
I'm the child process.
My parent is 9.
I'm exiting now. Bye!
I'm the parent process and I'm exiting now. Bye!

/fd0/ $
```



A screenshot of a QEMU terminal window. The window title is "QEMU" and it has standard window controls. The terminal shows a prompt "/fd0/ \$" followed by the command "cat". The output displays the contents of "license.txt", which includes the GNU General Public License version 1.0 text. The terminal ends with the prompt "/fd0/ \$".

```
/fd0/ $ cat
Filename: license.txt
AssignmentOS Version 1.0
Copyright (C) 2014 Richard Tsai (http://richardtsai.info)

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.

/fd0/ $
```