操作系统原理实验报告

实验七：五状态进程模型

数据科学与计算机学院　17大数据与人工智能

17341015　陈鸿峥

一、 实验目的

- 理解五状态进程模型的原理
- 实现更多的进程状态管理函数，如fork、wait、exit等
- 能够将这些函数封装起来供上层应用程序使用

二、 实验要求

在实验五或更后的原型基础上，进化你的原型操作系统，原型保留原有特征的基础上，设计满足下列要求的新原型操作系统：

1. 实现控制的基本原语do_fork()、do_wait()、do_exit()、blocked()和wakeup()
2. 内核实现三系统调用fork()、wait()和exit()，并在C库中封装相关的系统调用
3. 编写一个C语言程序，实现多进程合作的应用程序。多进程合作的应用程序可以在下面的基础上完成：由父进程生成一个字符串，交给子进程统计其中字母的个数，然后在父进程中输出这一统计结果。参考程序如下：

```
char str[80] = "129djwqhdsajd128dw9i39ie93i8494urjoiew98kdkd";
int LetterNr = 0;

void CountLetter()
{
    int len = strlen(str);
    for(int i = 0; i < len; ++i)
        if(isalpha(str[i]))
            LetterNr++;
}

void main() {
    int pid = fork();
    if (pid == -1){
        printf("Error in fork!\n");
        return;
    }
    if (pid) {
        wait();
```

```
        printf("LetterNr = %d, I'm parent!\n", LetterNr);
    } else {
        CountLetter();
        printf("I'm child!\n");
        exit(0);
    }
    return;
}
```

编译连接你编写的用户程序，产生一个.com文件，放进程原型操作系统映像盘中。

三、 实验环境

具体环境选择原因已在实验一报告中说明。

- Windows 10系统 + Ubuntu 18.04(LTS)子系统
- gcc 7.3.0 + nasm 2.13.02 + GNU ld (Binutils) 2.3.0
- GNU Make 4.1
- Oracle VM VirtualBox 6.0.6
- Bochs 2.6.9
- Sublime Text 3 + Visual Studio Code 1.33.1

虚拟机配置：内存4M，1.44M虚拟软盘引导，1.44M虚拟硬盘。

四、 实验方案

本次实验继续沿用实验六保护模式的操作系统。

本次实验亮点如下：

- 增添了对**ELF文件**的读入、解析、执行
- 实现五状态进程模型，同时确保**不同特权级**的转换能够正常运行，且**不同进程空间、堆栈分配**不会重叠
- 增添`fork`、`wait`、`exit`等系统调用（**内核态**），注意都能在**用户态程序**下调用
- 增添用户C程序的编译支持，增添**链接文件**`link.ld`
- 支持**单用户程序多个fork及多个用户程序的进程管理**

1. ELF文件

可执行与可链接格式(Executable Linkable Format, ELF)是一种x86架构上类Unix操作系统的二进制文件标准格式，其具体内容可见图1[1]。

[1]图源自https://www.cirosantilli.com/elf-hello-world/

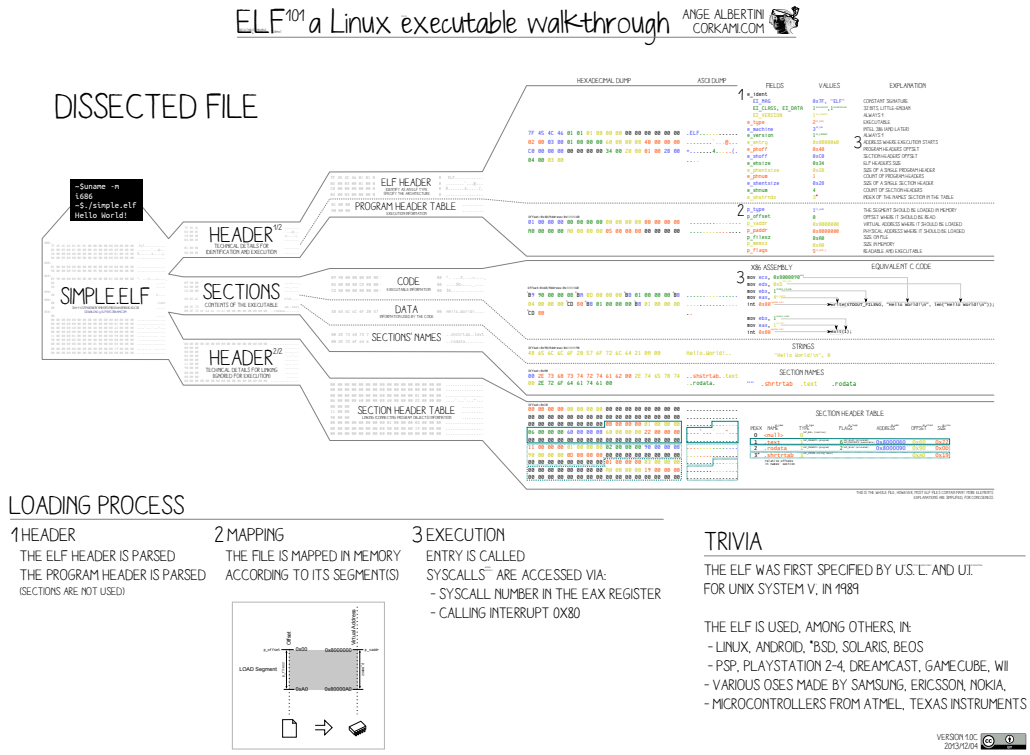

图 1: ELF格式说明

通过以下几条指令可以查看ELF文件的内容

```
readelf -h fork_test.out ## show elf header
readelf -S fork_test.out ## show section headers
readelf -l fork_test.out ## show program headers
objdump -d fork_test.out ## show assembly
```

ELF文件由四个部分组成，分别是ELF头、程序头表、节和节头表四个部分组成。其基本信息包含在`elf.h`中，借鉴了清华UCore的格式定义，如下所示。

```
/* file header */
typedef struct elfhdr {
    uint32_t e_magic;   // must equal ELF_MAGIC
    uint8_t e_elf[12];
    uint16_t e_type;    // 1=relocatable, 2=executable, 3=shared object, 4=core
        ↪ image
    uint16_t e_machine; // 3=x86, 4=68K, etc.
    uint32_t e_version; // file version, always 1
    uint32_t e_entry;   // entry point if executable
    uint32_t e_phoff;   // file position of program header or 0
    uint32_t e_shoff;   // file position of section header or 0
    uint32_t e_flags;   // architecture-specific flags, usually 0
```

```
    uint16_t e_ehsize;  // size of this elf header
    uint16_t e_phentsize; // size of an entry in program header
    uint16_t e_phnum;   // number of entries in program header or 0
    uint16_t e_shentsize; // size of an entry in section header
    uint16_t e_shnum;   // number of entries in section header or 0
    uint16_t e_shstrndx; // section number that contains section name strings
} elfhdr;

/* program section header */
typedef struct prghdr {
    uint32_t p_type; // loadable code or data, dynamic linking info, etc.
    uint32_t p_offset; // file offset of segment
    uint32_t p_va;    // virtual address to map segment
    uint32_t p_pa;    // physical address
    uint32_t p_filesz; // size of segment in file
    uint32_t p_memsz; // size of segment in memory (bigger if contains bss)
    uint32_t p_flags; // read/write/execute bits
    uint32_t p_align; // required alignment, invariably hardware page size
} prghdr;
```

其实我们需要关心的只有其中几项，包括

- e_magic：是否为合法的ELF文件
- e_entry：可执行文件的入口，这里我通过链接文件link.ld指定入口点为main函数
- p_offset：程序段在ELF文件中的偏移量
- p_memsz：程序段的大小

故可以得到下面的ELF文件加载、执行逻辑（见user.h文件），这一部分全部是**原创内容**。一定要注意，ELF文件头是占空间的，故要先对文件头进行解析，然后再将后面实际程序的执行代码加载到内存中。

```
oid exec_elf(int num) {
    disable();

    uintptr_t addr_exec = ADDR_USER_START+(num-1)*PROC_SIZE;
    uintptr_t addr = (uintptr_t)bin_img;

    read_sectors(addr,(num-1)*2,30);

    // parse elf header
    elfhdr eh;
    memcpy((void*)&eh,(void*)addr,sizeof(elfhdr));

    if (eh.e_magic != ELF_MAGIC){
```

```
        put_error("Bad elf file!");
        for(;;){}
    }

    // parse program header
    prghdr ph;
    for (int i = 0, offset = eh.e_phoff;
            i < eh.e_phnum; i++, offset += eh.e_phentsize){
        memcpy((void*)&ph,(void*)(addr+offset),eh.e_phentsize);
        if (i == 0)
            memcpy((void*)addr_exec,(void*)(addr+ph.p_offset),ph.p_memsz);
    }

    process* pp = proc_create(USER_CS,USER_DS,eh.e_entry);
    schedule_proc();
}
```

- 首先将ELF文件读入到bin_img中，这里预设程序大小为30个扇区
- 然后直接拷贝ELF头到结构体eh中，由于顺序已经确定，故不需要对结构体元素一一赋值
- 通过ELF_MAGIC判断ELF头是否合法，合法则再执行下面步骤
- 拷贝程序头到结构体ph中，然后将[p_offset,p_offset+p_memsz]实际程序的执行代码全部拷贝到进程空间中
- 创建进程入口地址为e_entry，然后开始进程调度执行

2. 五状态进程模型

五状态进程模型我依照图2进行命名和定义。

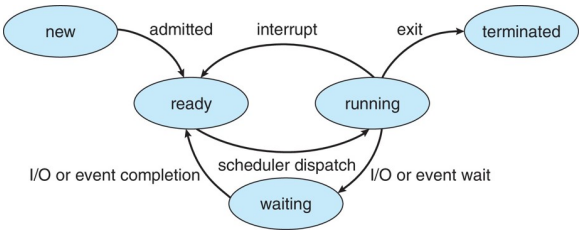

图 2: 五状态进程模型

(i) fork

这里一个关键问题在于保护模式下用户程序都处在**用户态**执行，而创建管理进程这些函数都属于**内核态**函数，如何ring 3的程序调用ring 0的例程这是需要考虑的。而我的解决方案是通过**中断调用**的方式，需要满足以下两点。

- 系统中断0x80设置代码段为内核代码段，但特权级DPL为3
- 内核代码段与用户代码段重叠，在GDT中定义

用户C程序会包含api.h的头文件，其中封装了用户态的fork函数，如下所示，在C中内嵌汇编代码实现。

通过eax寄存器传递功能号，然后调用系统中断0x80，进入内核态的do_fork进行执行，然后返回用户态，再次利用eax寄存器进行返回值传递。

```
int fork() {
    int pid;
    asm volatile (
        "mov eax, 10\n\t"
        "int 0x80\n\t"
        :"=a"(pid):);
    return pid;
}
```

当调用了0x80中断后，会跳转到sys_interrupt_handler。先关中断，然后立即对现场进行保护save_proc_entry（这里逻辑与实验六切换进程第一步保存进程状态的逻辑相同，不再赘述），方便后续的fork操作。注意这里保存的是上述int 0x80的下一条指令mov [pid], eax的eip值，这对父子进程的返回值有着至关重要的作用[2]。

由于C语言采用eax存储返回值，故这里注意不要破坏现场，执行完中断处理程序后直接返回即可。

```
sys_interrupt_handler:
    cli
    jmp save_proc_entry
save_proc_entry_ret:
    push eax           ; Functional number
    call sys_interrupt_handler_main
    pop ebx            ; Remember to pop out
    ; eax store the return value
    sti
    iretd
```

fork的核心代码如下，除了要拷贝PCB外，核心关键点是还要**拷贝整个进程栈**。否则父进程继续执行将会将原来堆栈的内容弹出，使得子进程无法再继续使用。

至于父子进程返回值不同的问题，前面脚注已经提及。父进程直接返回子进程ID，而子进程通过修改regImg.eax的值，使得进程切换后的eax就是0。

[2]保存现场不一定要在中断调用这里完成，我见到有些操作系统实现是在内核态的中断处理程序中直接读取eip创建新的进程的，然后通过当前进程是否是父进程判断返回什么值。这种情况下，fork出来的进程将处在内核态中，返回用户态程序还要切换堆栈非常麻烦。而采用我原创的这种方法则可以在fork出来后，依然在ring 3顺序执行用户态程序，且确保了父子进程的返回值不同。

```
int do_fork() {

    disable();
    process* child;

    // find empty entry
    if ((child = proc_alloc()) == 0) {
        enable();
        return -1; // fail to create child process
    }

    // copy PCB, which has been saved by interrupt
    copy_PCB(child,curr_proc); // stack must be changed!

    // copy stack
    memcpy((void*)(child->regImg.user_esp),
        (void*)(curr_proc->regImg.user_esp),
        (curr_proc->regImg.ebp - curr_proc->regImg.user_esp)*2); // each entry 32-
            ↪ bit

    // set state
    child->regImg.eax = 0; // set child process return value
    child->parent = curr_proc;
    child->status = PROC_READY;
    reset_time(child);

    return child->pid;
}
```

(ii) 进程管理

wait逻辑很简单，就是切换进程状态，然后立即执行调度。

```
void do_wait() {
    disable();
    curr_proc->status = PROC_WAITING;
    schedule_proc();
    enable();
}
```

同理exit如下，需要唤醒父进程。

```
void do_exit() {
    disable();
    curr_proc->status = PROC_TERMINATED;
    wakeup(curr_proc->parent->pid);
```

```
    enable();
}
```

wakeup则找到父进程，然后将其状态更改回就绪，调用进程调度。

```
void wakeup(uint8_t pid) {
    disable();
    for (process* pp = &proc_list[0]; pp < &proc_list[MAX_PROCESS]; ++pp){
        if (pp->pid == pid && pp->status == PROC_WAITING){
            pp->status = PROC_READY;
            schedule_proc();
            enable();
            return;
        }
    }
    enable();
}
```

调度模块相比起实验六做了一些改变，如下所示。

```
process* proc_pick()
{
    int curr_index = -1;
    // find current running process
    for (int i = 0; i < MAX_PROCESS; ++i){
        if (proc_list[i].status == PROC_RUNNING){
            curr_index = i;
            break;
        }
    }
    // round-robin
    for (int i = 0; i < MAX_PROCESS; ++i){
        int index = (i + 1 + curr_index) % MAX_PROCESS;
        if (proc_list[index].status == PROC_READY)
            return &proc_list[index];
    }
    if (curr_index != -1) // no choice but the process still running
        return &proc_list[curr_index];
    else
        return NULL;
}

void schedule_proc()
{
    process* pp = proc_pick();
    if (pp == NULL)
```

```
        return;

    proc_switch(pp);
}
```

(iii) 系统调用

目前关于进程的操作函数有以下这些（内核态），大部分都已在实验六中说明。

- `proc_init`：初始化
- `proc_alloc`：分配新进程，状态由`NEW`变更为`READY`
- `proc_create`：根据地址创建新进程，结合`alloc`一起使用
- `proc_pick`：挑选`READY`进程
- `save_proc`：保存进程状态
- `restart_proc`：恢复进程状态
- `proc_switch`：进程切换
- `schedule_proc`：进程调度
- `do_fork`：进程分支
- `do_wait/blocked`：进程等待/阻塞
- `do_exit`：(子)进程退出
- `wakeup`：进程唤醒
- `sys_get_pid`：返回进程ID
- `kill`：结束进程

目前的系统调用如表1所示，均可直接在用户态调用。

表 1: 系统调用功能表

功能号	功能
0	输出OS Logo
1	睡眠100ms
10	fork
11	wait
12	exit
13	get_pid
100	返回内核Shell

五、 实验结果

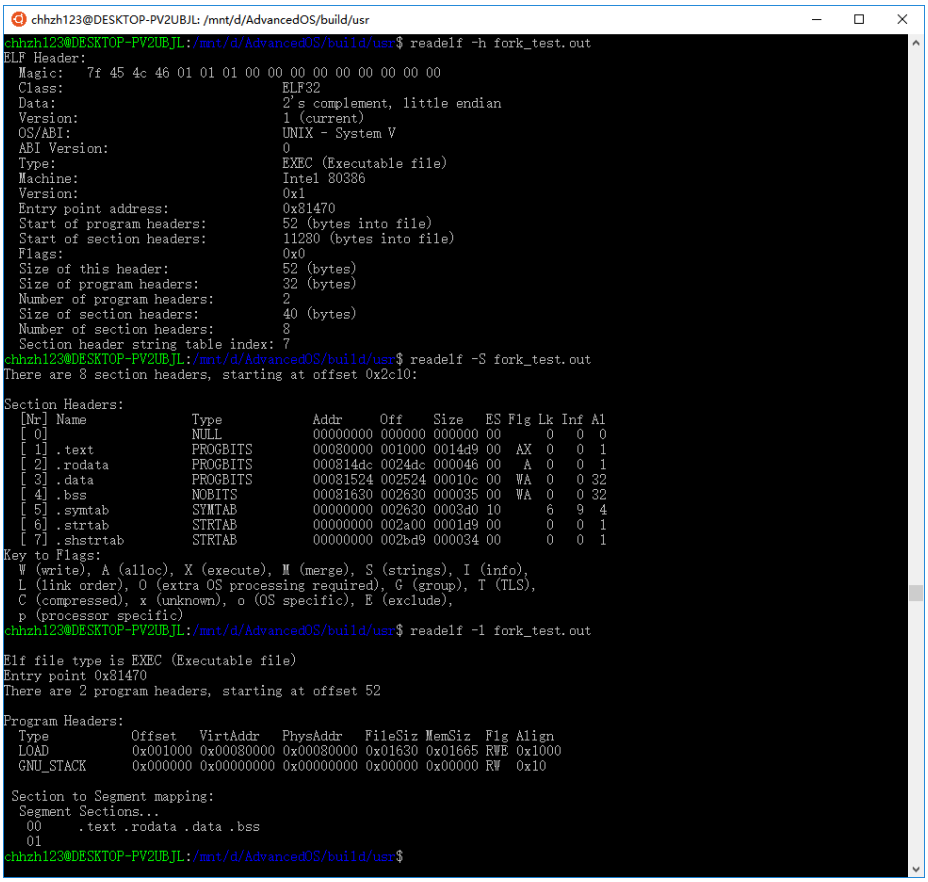

图 3: 利用readelf查看ELF文件内容

参考程序已在老师程序的基础上进行了修改，见首页，分别输出父子进程，如图4所示，exec 7调用执行fork_test.c，结果为27。

同时，我也自己再写了一个测试程序fork2.c，如下，用于测试**多个子进程**环境下是否还能正常运行。

```
#include "stdio.h"
#include "api.h"

void main()
{
    int a = fork();
    int b = fork();
    int c = fork();
    if (a == 0 || b == 0 || c == 0)
        printf("This is hello from process %d! (child)\n", get_pid());
    else
```

```
        printf("This is hello from process %d! (parent)\n", get_pid());
    return;
}
```

同样结果如图4所示，由于fork2.c没有wait指令等待子进程完成，故会返回多次Shell。抛出Shell命令，可以看出，结果产生8个进程是正确的。

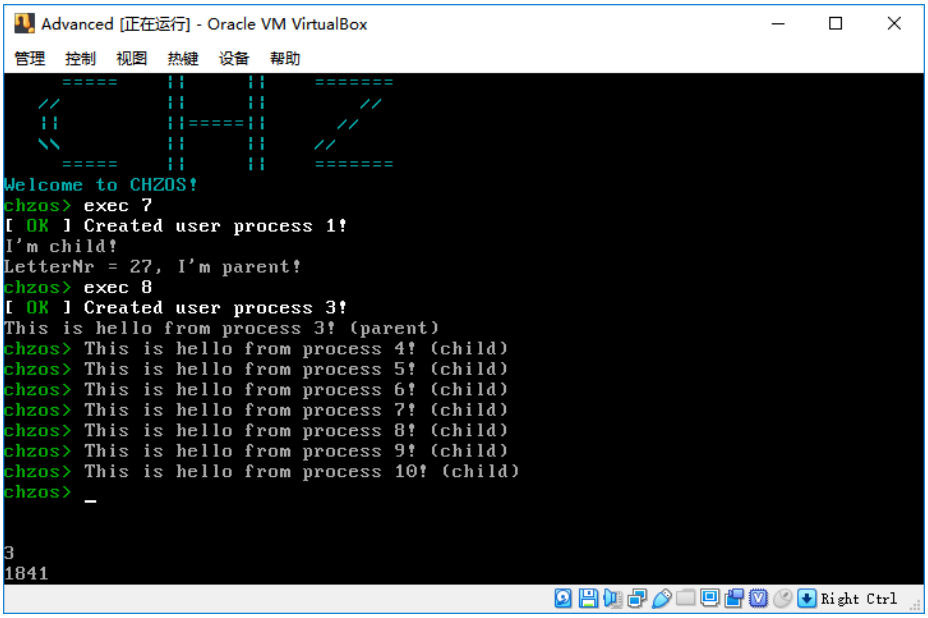

图 4: fork演示多进程合作

注意：

- 这里的用户程序是用gcc编译为**ELF文件**存储入虚拟硬盘后，在OS内核执行过程中**动态载入、解析、执行**的
- 执行完用户程序（用户态）后将**返回内核Shell**（内核态）
- 这里的exec 7和exec 8都是在交互界面输入后才执行，不是预先安排好的。可以看出本套进程管理系统不仅对单个程序内部的多进程管理做得很好，而且对多个程序的进程管理也可以胜任。

六、 实验总结

一开始想着实验六已经将保护模式操作系统的整个框架搭建起来，继续做实验七应该很快。但事实证明我又错了。保护模式每一个实验都非常耗时，核心问题都在于**用户态和内核态**之间如何进行切换。

为了更好地管理用户程序，以及具有一定的普适性，这一次将ELF文件支持添加到我的操作系统中。虽然ELF的格式较为简单，但要理解透彻并具体实施还是要考虑很多细节的。比如

最开始老是没想明白如何将核心用户程序段加载到对应位置，因为以往都有汇编作为入口，跳转地址就直接是汇编程序的第一行。但现在程序不是一个扁平的二进制文件，编译出来有一大堆函数，入口点就不再是第一条指令了。而ELF头正是给出了文件的这些信息，故需要先进行解析，然后再将后面的程序正文读到内存的指定位置，然后通过ELF头给出的入口地址直接跳转。其实从这个角度看，ELF反而让用户程序好管理了很多。

`fork`函数的编写也耗费了我很多脑细胞，因为没有采用常规思路，完全依照`fork`函数要达成的功能自己想怎么实现。关键在于怎么让`fork`的父子进程返回不同的值，这一点深入了解了C函数调用的汇编原理就好做很多，核心就是通过`eax`寄存器进行传递。所以针对父进程可以直接返回子进程ID；而针对子进程，则在父进程中对子进程的`eax`进行设置，切换进程时就会自动恢复。

现在我的操作系统最大的问题还在于用户程序的管理，每次添加一个用户程序，都要改一遍`Makefile`文件，然后改一大批头文件（只要包含用户程序的都要进行修改），还要自己算内存地址，这些都是相当不智能的地方。但之后继续往下做了分页和文件系统应该就更好管理。

七、 参考资料

1. OS Development Series, http://www.brokenthorn.com/Resources/OSDevIndex.html
2. Roll your own toy UNIX-clone OS, http://www.jamesmolloy.co.uk/tutorial_html/
3. The little book about OS development, http://littleosbook.github.io/
4. Writing a Simple Operating System from Scratch, http://www.cs.bham.ac.uk/~exr/lectures/opsys/10_11/lectures/os-dev.pdf
5. Intel® 64 and IA-32 Architectures Software Developer's Manual
6. UCore OS Lab, https://github.com/chyyuu/ucore_os_lab
7. CMU CS 15-410, Operating System Design and Implementation, https://www.cs.cmu.edu/~410/
8. 李忠，王晓波，余洁，《x86汇编语言-从实模式到保护模式》，电子工业出版社，2013
9. ELF Hello World Tutorial, https://www.cirosantilli.com/elf-hello-world/

附录 A. 程序清单

1. 内核核心代码

序号	文件	描述
1	`bootloader.asm`	主引导程序
2	`kernel_entry.asm`	内核汇编入口程序
3	`kernel.c`	内核C入口程序
4	`Makefile`	自动编译指令文件
5	`bootflpy.img`	引导程序/内核软盘
6	`mydisk.hdd`	虚拟硬盘
7	`bochsrc.bxrc`	Bochs配置文件

2. 内核头文件

序号	文件	描述
1	disk_load.inc	BIOS读取磁盘
2	show.inc	常用汇编字符显示
3	gdt.inc	汇编全局描述符表
4	gdt.h	C全局描述符表
5	idt.h	中断描述符表
6	hal.h	硬件抽象层
6.1	pic.h	可编程中断控制器
6.2	pit.h	可编程区间计时器
6.3	keyboard.h	键盘处理
6.4	tss.h	任务状态段
6.5	ide.h	硬盘读取
7	io.h	I/O编程
8	exception.h	异常处理
9	syscall.h	系统调用
10	task.h	多进程设施
11	user.h	用户程序处理
12	terminal.h	Shell
13	scancode.h	扫描码
14	stdio.h	标准输入输出
15	string.h	字符串处理
16	elf.h	ELF文件处理
17	api.h	进程管理API

3. 用户程序

用户程序都放置在usr文件夹中。

序号	文件	描述
1-4	prgX.asm	飞翔字符用户程序
5	box.asm	画框用户程序
6	sys_test.asm	系统中断测试
7	fork_test.c	进程分支测试
8	fork2.c	进程多分支测试