实验七：实现五态进程模型及多进程应用

计算机科学与技术 18340146 宋渝杰

实验目的：

1. 理解5状态的进程模型
2. 掌握操作系统内核线程模型设计与实现方法
3. 掌握实现5状态的进程模型方法
4. 实现C库封装多线程服务的相关系统调用

实验要求：

1. 学习内核级线程模型理论，设计总体实现方案
2. 理解类unix的内核线程做法，明确全局数据、代码、局部变量的映像内容哪些共享。
3. 扩展实验6的的内核程序，增加阻塞进程状态和阻塞过程、唤醒过程两个进程控制过程。
4. 修改内核，提供创建线程、撤销线程和等待线程结束，实现你的线程方案。
5. 增加创建线程、撤销线程和等待线程结束等系统调用。修改扩展C库，封装创建线程、撤销线程和等待线程结束等系统调用操作。
6. 设计一个多线程应用的用户程序，展示你的多线程模型的应用效果。
7. 编写实验报告，描述实验工作的过程和必要的细节，如截屏或录屏，以证实实验工作的真实性

实验内容：

1. 修改内核代码，增加阻塞队列，实现5状态进程模型
2. 如果采用线程控制块，就要分离进程控制块的线程相关项目，组成线程控制块，重构进程表数据结构。
3. 修改内核代码，增加阻塞block()、唤醒wakeup()、睡眠sleep()、创建线程do_fork()、撤销线程do_exit()和等待线程结束do_wait()等过程，实现你的线程控制。
4. 修改扩展C库，封装创建线程**fork()**、撤销线程**exit(0)**、睡眠sleep()和等待线程结束**wait()**等系统调用操作。
5. 设计一个多线程应用的用户程序，展示你的多线程模型的应用效果。示范：进程创建2个线程，分别统计全局数组中的字母和数字的出现次数。你也可以另选其他多进程应用。

实验过程：

操作环境：

- 操作系统：win10 + Linux
- 虚拟机：VirtualBox
- C 编译器：gcc
- x86 编译器：nasm
- 链接器：ld

本次实验主要需要实现线程的相关操作：创建线程 fork(), 撤销线程 exit(), 等待线程结束 wait() 以及其它相关的操作，来实现多线程的各种效果。具体的实现步骤如下：

步骤一：实现 fork()

fork() 主要实现多线程的建立，功能通过 c 内核中的 do_fork() 实现。它要进行的操作如下：

1. 在 PCB 表中选取位置（配合多线程应用的用户程序，这里选取两个位置）
2. 将父进程的 PCB 表和堆栈复制给子进程
3. 设置父进程和子进程的 ID

由于**多线程的本质，是父进程创建子进程**，因此实验六的 PCB 表可以直接照用，但是子进程的寄存器初始值的赋值会有不同

- 堆栈段 ss 和寄存器 ax 和父进程不同（分别用于堆栈的复制和进程 ID 的返回）
- 其它寄存器和父进程保持完全相同

因此具体的代码和注释如下：

```
void do_fork() {
    for (int i=0; i<2; i++) { // 选取两个位置
        asm volatile(
            "mov es, ax\n"
            :
            : "a"((i+1)*0x1000)); // 子进程的堆栈位置
        asm volatile(
            "mov ds, ax\n"
            :
            : "a"(qi->ss)); // 父进程的堆栈位置
        asm volatile(
            "mov di, 0\n"
            "mov si, 0\n"
            "cld\n"
            "rep movsw\n"
            :
            : "c"(0xFE00 - qi->sp)); // 堆栈复制：ds:si->es:di
        (qe+i)->sp = qi->sp;
        (qe+i)->bp = qi->bp;
        (qe+i)->si = qi->si;
        (qe+i)->di = qi->di;
        (qe+i)->ss = (i+1)*0x1000; // 子进程 ss 指向刚刚复制的新堆栈
        (qe+i)->es = qi->es;
        (qe+i)->ds = qi->ds;
        (qe+i)->cs = qi->cs;
        (qe+i)->ax = i+1; // 子进程 ID 通过 ax 返回
        (qe+i)->bx = qi->bx;
        (qe+i)->cx = qi->cx;
        (qe+i)->dx = qi->dx;
        (qe+i)->pid = i+1;
        (qe+i)->statu = 1; // 就绪态
        (qe+i)->flags = qi->flags;
        (qe+i)->ip = qi->ip; // ip 放最后
    }
    muti = 1; // 开启时间片轮转
}
```

之后，将 do_fork() 封装为系统调用，中断号为 `int 23h` ，功能号 `ah = 00h` （最后的 `int 23h` 将在后文给出）

```
sys_fork:
    call dword do_fork
    ret
```

最后，封装 fork() 函数，用于用户程序调用：

```
void fork() {
    asm volatile(
        "mov ax, 0\n"   // 功能号
        "int 0x23\n");  // 调用 int 23h
}
```

步骤二：实现 wait()

wait() 主要实现该线程的阻塞，功能通过 c 内核中的 do_wait() 实现。它要进行的操作如下：

1. 将该线程置为阻塞状态
2. 通过 schedule()，切换到下一个线程

这部分较为简单，直接附上代码和注释：

```
void do_wait() {
    qi->statu = 0; // 阻塞状态
    schedule(); // 切换线程
}
```

之后，将 do_wait() 封装为系统调用，中断号为 `int 23h` ，功能号 `ah = 01h` （最后的 `int 23h` 将在后文给出）

```
sys_wait:
    call dword do_wait
    ret
```

最后，封装 wait() 函数，用于用户程序调用：

```
void wait() {
    asm volatile(
        "mov ah, 1\n"   // 功能号
        "int 0x23\n");  // 调用 int 23h
}
```

步骤三：实现 exit()

exit() 主要实现该线程的结束，并根据情况唤醒主线程，功能通过 c 内核中的 do_exit() 实现。它要进行的操作如下：

1. 将该线程置为结束状态（具体的状态值也为 0）
2. 判断所有的子线程是否运行完毕，如果没运行完毕，则通过 schedule()，切换到下一个线程；如果都运行完毕，则先把主线程置为就绪态，再通过 schedule()，切换到主线程

同时，我把唤醒功能 wakeup() 集成到 exit() 上，下面附上代码和注释：

```
void do_exit() {
    qi->statu = 0; // 结束状态
    for (int i=0; i<4; i++)
        if ((qe+i)->statu == 1) { // 还有子线程为就绪态
            schedule(); // 切换到下一个子线程
            return;
        }
    (qe-1)->statu = 1; // 子线程均已结束，恢复主线程（即 wakeup ）
    schedule(); // 切换到主线程
}
```

之后，将 do_exit() 封装为系统调用，中断号为 `int 23h`，功能号 `ah = 02h` （最后的 `int 23h` 将在后文给出）

```
sys_exit:
    call dword do_exit
    ret
```

最后，封装 exit() 函数，用于用户程序调用：

```
void exit() {
    asm volatile(
        "mov ah, 2\n"   // 功能号
        "int 0x23\n");  // 调用 int 23h
}
```

步骤四：整合 int 23h

具体做法和我在实验五中整合系统调用 int 21h 的做法一模一样，故在此不再赘述

```
    extern do_fork,do_wait,do_exit
int23h:
    call save
    push ds
    push si                     ; 用si作为内部临时寄存器
    mov si, cs
    mov ds, si                  ; ds = cs
    mov si, ax
    shr si, 8                   ; si = 功能号
    add si, si                  ; si = 2 * 功能号
    call [sys23_table+si]       ; 系统调用函数
    pop si
    pop ds
    jmp restart
sys23_table:                    ; 存放功能号与系统调用函数映射的表
    dw sys_fork,sys_wait,sys_exit
```

同时别忘了把 int 23h 写入中断向量表：

```
    mov ax, 0000h                ; 内存前64k放置的中断向量表，将段寄存器指向该处
    mov es, ax
    mov ax, 23h                  ; 定义23号中断：多线程
    mov bx, 4
    mul bx
    mov si, ax
    mov ax, int23h
    mov [es:si], ax
    add si, 2
    mov ax, cs
    mov [es:si], ax
```

至此，多线程的功能已基本实现完毕

步骤五：编写多线程测试程序

多线程测试程序的主要过程如下：主线程调用 fork() 生成两个子线程；主线程调用 wait() 等待子线程结束；两个子线程分别统计字符串中字母和数字的数量，之后调用 exit() 结束线程并唤醒主线程；主线程唤醒后输出子线程统计信息，最后调用 int 20h 返回内核

```
int letterNum = 0;
int figureNum = 0;
void fork() {
    asm volatile(
        "mov ax, 0\n"
        "int 0x23\n");
}
void wait() {
    asm volatile(
        "mov ah, 1\n"
        "int 0x23\n");
}
void exit() {
    asm volatile(
        "mov ah, 2\n"
        "int 0x23\n");
}
void cmain() {
    ...
    fork(); // 生成两个子线程
    int pid;
    asm volatile(
        "add ax,0\n"
        : "=a"(pid) // pid = ax
        :);
    if (pid == 0) { // 主线程
        ...
        wait(); // 等待子线程结束
        ... // 输出子线程统计信息
    }
    else if (pid == 1) { // 子线程 1
        ... // 统计字母数量
        exit(); // 结束线程
    }
```

```
    else if (pid == 2) { // 子线程 2
        ... // 统计数字数量
        exit(); // 结束线程
    }
    return;
}
```

和实验六的用户程序一样，需要汇编程序辅助进入用户程序：

```
    bits 16
    extern cmain
start:
    mov ax,cs
    mov ds,ax
    mov es,ax
    mov ss,ax
    call dword cmain ; 跳转到测试程序
end:
    mov ah, 0
    int 16h
    int 20h     ; 返回内核
```

Linux 下使用下列编译命令行整合成用户程序 7.com，参数和之前的编译命令行一样，这里不再赘述

```
    gcc -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding -fno-PIE -
masm=intel -c 7.c -o 7c.o
    gcc -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding -fno-PIE -
masm=intel -c io.c -o io.o
    nasm -felf 7.asm -o 7asm.o
    ld -m elf_i386 -N --oformat binary -Ttext 0xb500 7asm.o 7c.o io.o -o 7.com
```

额外步骤：os 优化

由于加入了新的功能，此处需要对内核要有相应的必要调整

内核汇编部分：除了加入 int 23h 以及相应的子函数外，基本没有任何调整

内核 c 部分：os 的优化主要集中在内核 c 部分，包括新建指令打开多线程用户程序、更新 help 和 ls 指令

新建 fork 指令：当用户输入指令 fork 时，打开多线程的测试程序

```
    const char fork[] = "fork";
    if (cstr(str, fork) == 0) {
        judge = 1;
        LoadnEx(10); // 加载多线程测试程序
    }
```

更新 help 和 ls 指令：加入新的多线程测试程序的信息

```
    static char helpstr[] =
        "You can input these instructions.\n"
        "\n"
        "help         -- Print instructions\n"
```

```
        "exe          -- Open an exe\n"
        "exes         -- Open many exes\n"
        "runs         -- run all basic exes at the same time\n"
        "fork         -- test thread exe\n" // 加入这行
        "ls           -- Show four exes' information\n"
        "cls          -- Clear the screen\n"
        "exit         -- Exit OS\n";
    static char exestr[] =
        "exe1         -- LeftUp     512 bytes\n"
        "exe2         -- RightUp    512 bytes\n"
        "exe3         -- LeftDown   512 bytes\n"
        "exe4         -- RightDown  512 bytes\n"
        "exe5         -- RightDown  401 bytes\n"
        "exe6         -- RightDown  1724 bytes\n"
        "exe7         -- RightDown  2392 bytes\n"; // 加入这行
```

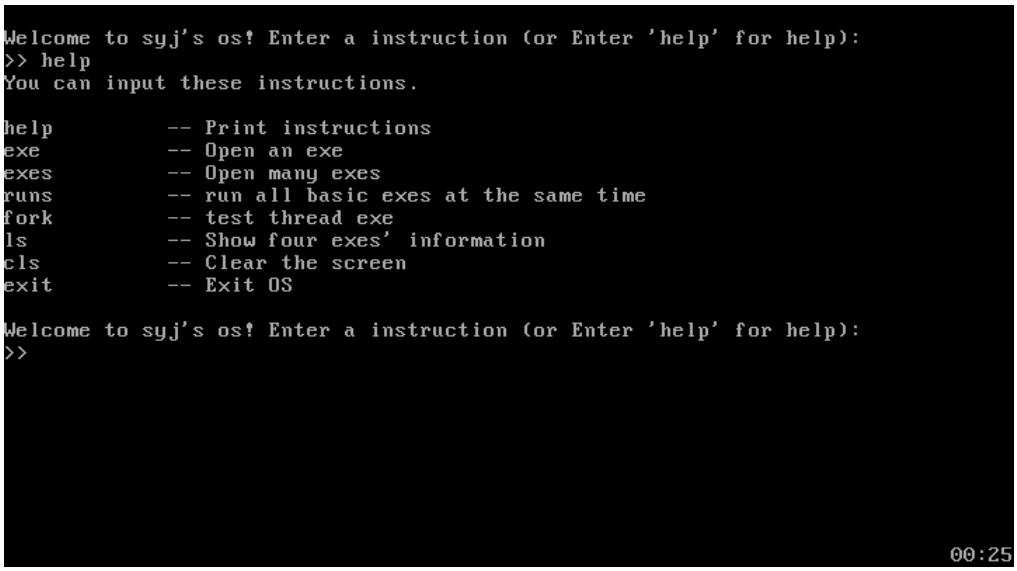
Linux 下运行以下编译命令行，生成 7-1.img 文件，参数和之前的实验一样，这里不再赘述

```
    // 内核
    gcc -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding -fno-PIE -
masm=intel -c 7-1c.c -o 7-1c.o
    nasm -felf 7-1asm.asm -o 7-1asm.o
    ld -m elf_i386 -N --oformat binary -Ttext 0x7e00 7-1asm.o 7-1c.o -o 7-1.bin
    nasm 7-1os.asm -o 7-1os.bin

    // 用户程序
    nasm 1.asm -o 1.com
    nasm 2.asm -o 2.com
    nasm 3.asm -o 3.com
    nasm 4.asm -o 4.com
    nasm 5.asm -o 5.com
    gcc -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding -fno-PIE -
masm=intel -c io.c -o io.o
    gcc -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding -fno-PIE -
masm=intel -c 6.c -o 6c.o
    nasm -felf 6.asm -o 6asm.o
    ld -m elf_i386 -N --oformat binary -Ttext 0xa100 6asm.o 6c.o io.o -o 6.com
    gcc -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding -fno-PIE -
masm=intel -c 7.c -o 7c.o
    nasm -felf 7.asm -o 7asm.o
    ld -m elf_i386 -N --oformat binary -Ttext 0xb500 7asm.o 7c.o io.o -o 7.com

    // 生成 7-1.img 文件
    rm -f 7-1.img
    /sbin/mkfs.msdos -C 7-1.img 1440
    dd if=1.com of=7-1.img seek=18 conv=notrunc
    dd if=2.com of=7-1.img seek=19 conv=notrunc
    dd if=3.com of=7-1.img seek=20 conv=notrunc
    dd if=4.com of=7-1.img seek=21 conv=notrunc
    dd if=5.com of=7-1.img seek=22 conv=notrunc
    dd if=6.com of=7-1.img seek=23 conv=notrunc
    dd if=7.com of=7-1.img seek=27 conv=notrunc
    dd if=7-1.bin of=7-1.img seek=1 conv=notrunc
    dd if=7-1os.bin of=7-1.img conv=notrunc
```

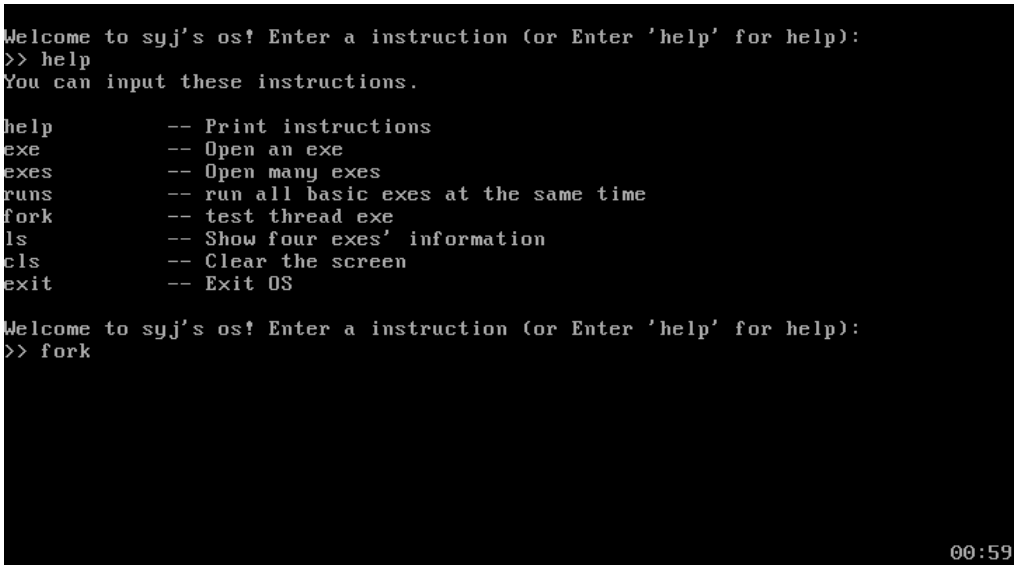
测试多线程：

对最终文件 7-1.img 的测试效果如下图：

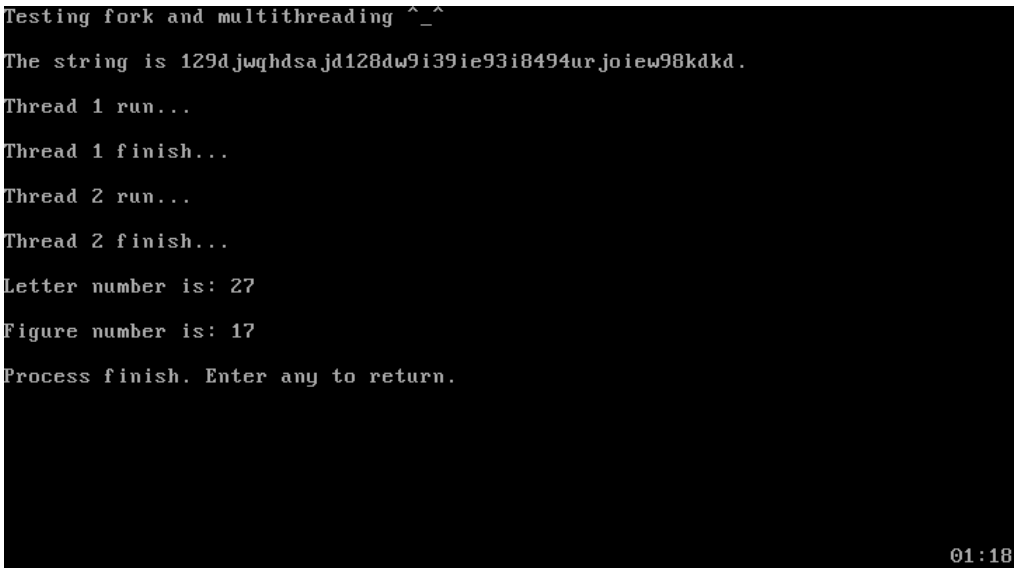
进入内核后，输入指令 help，可以看出 help 指令多了对 fork 指令的提示信息：



输入指令 fork，再回车，内核打开多线程测试程序：



多线程测试结果如下：主线程先输出前两句提示信息，子线程 1、2 开始运行并结束，最后主线程输出字母和数字的统计信息



实验心得：

这次实验难度比较简单，主要是 fork(), wait(), exit() 三个函数的实现。而多线程并行状态也和实验六的多进程并行类似，只是会有稍微的不同。

而对于其它技术层面的心得，我也在此一并讲述：

- **代码纠错：**这次程序在所有的步骤中均未参考老师代码，因此不存在老师代码纠错部分
- **实验过程中出现的出错问题及解决方式总结**：在上述三个函数的编写中，有走过几次弯路：一个是没有做栈复制和 ss 的变换，会导致多个线程同时处理同一个栈，用户程序就会崩掉；第二个是在 wait() 函数里，我原本是用一个 `while (qi->statu == 0) {}` 代替 `schedule()`，然后等待时钟中断切换到子线程，后来发现时钟中断不响应，是 wait() 调用 int 23h 中断，而这个中断不结束的话，时钟中断就会被屏蔽；
- **多进程和多线程的区别：**通过个人实验去理解，我觉得：多进程是多个用户程序并行运行，多线程是多个进程同时执行一个用户程序，但每个进程有每个独立的进程号，执行一个用户程序不同部分的代码；多进程之间的数据完全独立，同名的变量也独立开来，多线程每个线程内部的变量独立，外部的变量共享，子线程可以通过修改外部变量来实现对主线程的值传递。
- **修改实验六的一个 bug：**实验六的 io 库其中一个函数 pint()，在屏幕上输出一个数字，当输出的数字是 0 时，这个函数不会输出任何信息，在这次实验七已修改该 bug。