操作系统-实验七

学号：18340020　姓名：陈贤彪 学院：数据科学与计算机学院

1.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。

2.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。

2、扩展实验6的的内核程序，增加阻塞进程状态，唤醒过程两个进程控制过程。

3、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。

4、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。

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。

6、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

3.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。

```
Char str[80]=”129djwqhdsajd128dw9i39ie93i8494urjoiew98kdkd”;
Int LetterNr=0;
Void main() {
  Int pid;
  int ch;
  (AB)……..;
  pid=fork();
  if (pid==-1) {printf(“error in fork!”);exit(-1);}
```

```
  if (pid) {
     ch=wait();
  printf( “LetterNr=%d” , LetterNr );
  exit(0) ;
}
Else {
  LetterNr =CountLetter(str);
  exit(0);
}
```

4.实验方案

1）实验环境

a)系统： `Linux Ubuntu18.04`

2）实验工具

a) `VM VirtualBox`

虚拟机软件，用于模拟虚拟不同的操作系统，也可以创建多个虚拟软盘

b) `NASM-2.13.02`

汇编语言编译器，可以将写好的 `.asm` 文件编译成二进制文件.bin

c) `gcc (Ubuntu 5.5.0-12ubuntu1) 5.5.0 20171010`

c语言编译器

d)Visual Studio Code

代码编辑器,用于编辑 `asm` 代码

e) `GNU bash version 4.4.20(1)-release (x86_64-pc-linux-gnu)`

系统跟计算机硬件交互时使用的中间介质,用于简便对文件进行转换

f) `GNU ld (GNU Binutils for Ubuntu) 2.30`

链接器，将汇编与c生成的.o文件链接在一起

f) `github`

开源代码托管平台,用于存储管理编写的代码

g) `bochs 2.6.11`

软盘调试工具

3）实验原理

（1）总体磁盘架构

柱面号	磁头号	扇区偏移	占用扇区数	功能
0	0	1	1	引导扇区程序
0	0~1	2	35	操作系统内核
1	0	1	2	`userpro1`
1	0	3	2	`userpro2`
1	0	5	2	`userpro3`
1	0	7	2	`userpro4`
1	0	9	2	用户程序表`list`
1	0	11	22	C语言用户程序
1	1	15	2	系统调用测试程序
1	1	17	2	fork测试程序

相比于之前的代码量，内核代码越来越多，因此我直接将整个柱面的扇区分配给了内核

`userpro1`对应左上角滚动字符，`userpro2`对应又上角滚动字符，`userpro3`对应左下角滚动字符，`userpro4`对应右下角滚动字符.

c语言用户程序，由于编写了库，因此占用比较多的扇区数目

系统调用的测试程序放在最后面

在内核中，我创建了一个结构体存储用户程序的信息

本次实验七新增了fork测试程序

（2）MY操作系统内核的设计

内核程序结构：

新增：在进程调度的函数模块`pcb.asm`，`pcb.h`中加入了关于`fork`，`wait`，`block`的函数

程序名	代码形式	作用
kernel.asm	ASM	内核的入口，调用c中的函数，*新增运行风火轮代码*
kernel_a.asm	ASM	包含一些显示打印，IO借口，扇区加载的函数,*运行 ouch 代码*
stdio.h	C	包含一些对字符串处理的函数
kernel_c.asm	C	内核c代码，内核命令的主要部分（新增关于新指令的代码）
ouch.asm	ASM	封装了关于 ouch 键盘中断的代码
system_a.asm	ASM	包含关于系统调用（汇编）的函数
system_c.c	C	包含关于系统调用（c语言）的函数
pcb.asm	ASM	封装了关于 PCB 保护和恢复的函数**（新增fork，wait,block）**
pcb.h	C	封装了PCB的创建，初始化，进程调度，进程撤销的函数fork测试程序

以下是我已经实现的指令功能（**新增了** runall 7指令:fork的测试程序）

指令名	功能
clear	清楚屏幕
ls	调用 list 程序显示用户程序的信息，需要 按 esc 退出
help	显示帮助的信息
run	可以按照自定义顺序运行1234程序，（新增）run 5为系统调用的测试程序，run 6为c程序测试程序
time	显示当前系统时间
shutdown	关机
runall	同时运行多个用户程序，如 runall 1234 即是同时运行四个用户程序

（3）进程派生fork

本次实验进入了五状态模型，因此我定义了一个枚举类型来表示状态：

```
enum PCB_ZHUANGTAI {NEW, READY, RUNNING, BLOCKED, EXIT};
```

随后进程的派生的过程如下：

- 首先在进程表中查找是否有NEW空余的项，如果没有找到，则函数失败
- 若找到空余的表项，则将父进程的整个进程控制块赋值给子进程，特别的是要将父进程的堆栈状态复制给子进程的堆栈，然后吧父进程的id分给子进程

该函数 do_fork 函数定义在 pcb.h 中

函数首先是遍历所有的进程控制块，找到状态为NEW新建的块，若找到则首先将父进程的ax寄存器赋值成子进程的id，然后复制父进程的所有寄存器到子进程的寄存器中，然后将父进程的栈拷贝到子进程的栈中。最后将子进程的ax设置成0，然后还要将子进程的id改成父进程的序号用于exit时恢复父进程的状态。

```
void do_fork()
{
   uint16_t sid = 1;
   for(sid=1;sid<processnum;sid++)
   {
      if(pcb_table[sid].zhuangtai==NEW)break;
   }
   if(sid>=processnum||sid<0)
   {
      getcurrentpcb()->reg.ax=-1;
   }
   else
   {
      getcurrentpcb()->reg.ax=sid;
      initSubPcb(sid);//复制寄存器
      copyStack();//复制堆栈
      pcb_table[sid].reg.ax=0;
      pcb_table[sid].id=current_process;
   }
}
```

关于`initSubPcb`函数，该函数负责将父进程的寄存器复制到子进程的寄存器中，除了ax寄存器（用于返回），还有`ss`寄存器（需要独立的栈位置），然后需要赋值`slength`栈长度，`fseg`起始栈，`tseg`终点栈，用于栈的复制

```
void initsunpcb(uint16_t sunid){
   pcb_table[sunid].id=sunid;
   pcb_table[sunid].zhuangtai=READY;
   pcb_table[sunid].reg.ax=0;
   pcb_table[sunid].reg.cx=getcurrentpcb()->reg.cx;
   pcb_table[sunid].reg.dx=getcurrentpcb()->reg.dx;
   pcb_table[sunid].reg.bx=getcurrentpcb()->reg.bx;
   pcb_table[sunid].reg.sp=getcurrentpcb()->reg.sp;
   pcb_table[sunid].reg.bp=getcurrentpcb()->reg.bp;
   pcb_table[sunid].reg.si=getcurrentpcb()->reg.si;
   pcb_table[sunid].reg.di=getcurrentpcb()->reg.di;
   pcb_table[sunid].reg.ds=getcurrentpcb()->reg.ds;
   pcb_table[sunid].reg.es=getcurrentpcb()->reg.es;
   pcb_table[sunid].reg.fs=getcurrentpcb()->reg.fs;
   pcb_table[sunid].reg.gs=getcurrentpcb()->reg.gs;
   pcb_table[sunid].reg.ss=sunid*0x1000;
   pcb_table[sunid].reg.ip=getcurrentpcb()->reg.ip;
   pcb_table[sunid].reg.cs=getcurrentpcb()->reg.cs;
   pcb_table[sunid].reg.flags=getcurrentpcb()->reg.flags;

   slength=0xFE00-pcb_table[sunid].reg.sp;
   fseg=getcurrentpcb()->reg.ss;
   tseg=pcb_table[sunid].reg.ss;
}
```

随后就是`stackcopy`函数，用于将父进程栈的信息复制到子进程栈的位置，该函数定义在`pcb.asm`

具体的复制方式就是使用`cld  rep movsw`

```
global stackcopy
stackcopy:
   pusha
   push ds
   push es

   mov ax, word[tseg]       ; 子进程 ss
   mov es,ax
   mov di, 0
   mov ax, word[fseg]      ; 父进程 ss
   mov ds, ax
   mov si, 0
   mov cx, word[slength]    ; 栈的大小
   cld
   rep movsw               ; ds:si->es:di

   pop es
   pop ds
   popa
   retf
```

最后将`fork`整个过程定义成一个汇编函数，并写进`22h`向量中，其中`PUSHALLPCB`是将所有寄存器放进栈中，然后调用`pcbsave`函数来将寄存器写进进程控制块，调用完函数后，要使用`RESTARTPCB`寄存器恢复出来。

`WRITE_INT_VECTOR`是之前实验中就定义的写向量的宏定义

```
WRITE_INT_VECTOR 22h, sys_fork
extern do_fork
sys_fork:
   PUSHALLPCB
   call pcbsave            ; 将寄存器的值保存在PCB中
   add sp, 16*2             ; 丢弃参数
   call dword do_fork
   RESTARTPCB               ; 恢复寄存器

   iret                 ; 退出sys_fork
```

（4）进程阻塞wait

进程`do_wait`函数是将当前进程状态设置成阻塞状态，然后使用`schedulepcb`转到别的进程运行，函数操作比较简单，`do_wait`函数定义在`pcb.h`中

```
void waitfor()
{
   getcurrentpcb()->zhuangtai=BLOCKED;
   schedulepcb();
}
```

随后和fork一样，在`pcb.asm`中定义一个汇编函数，然后写进`23h`向量

```
WRITE_INT_VECTOR 23h, sys_wait
extern waitfor
sys_wait:
   PUSHALLPCB
   call pcbsave
   add sp, 16*2
   call dword waitfor
   RESTARTPCB
   iret
```

（5）进程恢复wakeup

进程恢复`wakeup`函数是直接项进程状态直接设置成准备状态便能成功。`wakeup`函数定义在`pcb.h`中

```
void wakeup(uint16_t id)
{
   pcb_table[id].zhuangtai=READY;
}
```

（6）进程退出exit

进程退出`do_exit`函数的操作则是将当前进程状态设置为退出状态，随后将该进程的父进程（通过id来查找）恢复（使用进程恢复函数），最后调用`schedulepcb`来转到另外的进程运行便可。`do_exit`函数定义在`pcb.h`中

```
void do_exit()
{
   wakeup(getcurrentpcb()->id);
   getcurrentpcb()->zhuangtai=EXIT;
   schedulepcb();
}
```

随后和fork一样，在`pcb.asm`中定义一个汇编向量函数，然后写进`24h`向量

```
WRITE_INT_VECTOR 24h, sys_exit
extern do_exit
sys_exit:
   PUSHALLPCB
   call pcbsave
   add sp,16*2
   call dword do_exit
   RESTARTPCB

   iret
```

（7）派生测试程序的编写

由于`fork,wait,exit`三个函数需要使用系统调用来进行调用，所以我的测试程序是使用汇编语言来写的。该汇编函数为`fork_test.asm`。由于汇编代码比较冗余，所以我使用c来表示该测试程序的思路逻辑。该程序可以在命令行中使用`runall 7`来调用。

测试程序的思路如下：

首先定义一个含有字母和数字的字符串，然后子进程负责计算字符串中的字母数目和数字数目，父进程负责打印两者。若成功进行fork，首先父进程会被阻塞，等子进程计数完退出后，父进程就会被恢复然后打印出来。因此若是程序能成功打印数目则证明代码是正确的。

```
int letter_count = 0;
int number_count=0;
char the_str[] = "129djwqhdsajd128dw9i39ie93i8494urjoiew98kdkd";

void countLetter() {
  char* ptr = the_str;
  while(ptr) {
    if(*ptr >= 'a' && *ptr <= 'z') {
      letter_count += 1;
    }
  }
}
void countnumber() {
  char* ptr = the_str;
  while(ptr) {
    if(*ptr >= '0' && *ptr <= '9') {
      number_count += 1;
    }
  }
}

void cmain()
{
  clearScreen();
  print("This is the `fork_test` user programme.\r\n");

  int pid = fork();

  if(pid < 0) {
    print("error in fork!\r\n");
  }
  else if(pid > 0) { // 父进程
    print("Parent process entered.\r\n");
    wait();

    print("The stri is: ");
    print(the_str);
    print("\r\nLetter number is: ");
    print(itoa(letter_count, 10));
    print("\r\nnumbers of number is: ");
    print(itoa(number_count, 10));
    print("\r\nPlease Press ESC to quit.\r\n");
    exit();
  }
  else { // 子进程
    print("Son process entered.\r\n");
    countLetter(); // 统计字母的个数
```

```
48      countnumber();// 统计数字的个数
49      exit();
50    }
51  }
```

（8）程序的编译与整合

由于程序的编译以及整合是一个大量重复工作，因此我使用bash脚本来快速进行编译与整合，本次实验加上的文件为`fork_test.asm`。

`combine.sh`

```
1  #!/bin/bash
2  rm -rf temp
3  mkdir temp
4  rm *.img
5
6  nasm booter.asm -o ./temp/booter.bin
7
8  cd usrprog
9  nasm topleft.asm -o ../temp/topleft.com
10 nasm topright.asm -o ../temp/topright.bin
11 nasm bottomleft.asm -o ../temp/bottomleft.bin
12 nasm bottomright.asm -o ../temp/bottomright.bin
13 nasm list.asm -o ../temp/list.bin
14 nasm sys_test.asm -o ../temp/sys_test.bin
15 nasm fork_test.asm -o ../temp/fork_test.bin
16
17 cd c_test
18 nasm -f elf32 main.asm -o ../../temp/main_a.o
19 gcc -c -m16 -march=i386 -masm=intel -nostdlib -ffreestanding -mpreferred-stack-boundary=2 -lgcc
   -shared main.c -fno-pic  -o ../../temp/main_c.o
20 ld -m elf_i386 -N -Ttext 0xB900 --oformat binary ../../temp/main_a.o  ../../temp/main_c.o -o
   ../../temp/main.bin
21 cd ..
22
23 cd ..
24
25 cd lib
26 nasm -f elf32  system_a.asm -o ../temp/system_a.o
27 gcc -c -m16 -march=i386 -masm=intel -nostdlib -ffreestanding -mpreferred-stack-boundary=2 -lgcc
   -shared system_c.c -fno-pic  -o ../temp/system_c.o
28 cd ..
29 cd kernel
30 nasm -f elf32 kernel.asm -o ../temp/kernel.o
31 nasm -f elf32 kernel_a.asm -o ../temp/kernel_a.o
32 nasm -f elf32 ouch.asm -o ../temp/ouch.o
33 nasm -f elf32 pcb.asm -o ../temp/pcb.o
34 gcc -c -m16 -march=i386 -masm=intel -nostdlib -ffreestanding -mpreferred-stack-boundary=2 -lgcc
   -shared kernel_c.c -fno-pic  -o ../temp/kernel_c.o
35 ld -m elf_i386 -N -Ttext 0x7e00 --oformat binary ../temp/kernel.o ../temp/kernel_a.o
   ../temp/kernel_c.o  ../temp/ouch.o  ../temp/system_a.o ../temp/system_c.o  ../temp/pcb.o  -o
   ../temp/kernel.bin
36 cd ..
37 rm ./temp/*.o
```

```
38
39 dd if=./temp/booter.bin of=myosv7.img bs=512 count=1 2>/dev/null
40 dd if=./temp/kernel.bin of=myosv7.img bs=512 seek=1 count=35 2>/dev/null
41 dd if=./temp/topleft.com of=myosv7.img bs=512 seek=36 count=2 2>/dev/null
42 dd if=./temp/topright.bin of=myosv7.img bs=512 seek=38 count=2 2>/dev/null
43 dd if=./temp/bottomleft.bin of=myosv7.img bs=512 seek=40 count=2 2>/dev/null
44 dd if=./temp/bottomright.bin of=myosv7.img bs=512 seek=42 count=2 2>/dev/null
45 dd if=./temp/list.bin of=myosv7.img bs=512 seek=44 count=2 2>/dev/null
46 dd if=./temp/main.bin of=myosv7.img bs=512 seek=46 count=22 2>/dev/null
47 dd if=./temp/sys_test.bin of=myosv7.img bs=512 seek=68 count=2 2>/dev/null
48 dd if=./temp/fork_test.bin of=myosv7.img bs=512 seek=70 count=2 2>/dev/null
49 echo "[+] Done."
50
51
```

该脚本需要严格对应磁盘的放置，譬如`dd`时的扇区号，以及`ld`中`-Ttext 0x7E00`需要严格对照内存放置情况，不然会导致错误。

5.实验过程

1）踩坑过程

- 子进程栈的复制问题

一开始我编写fork代码的时候，本以为直接将父进程全部的进程控制块复制给子进程就可以了，但是我尝试后发现这样直接操作之后会导致两个进程的栈会乱成一套。随后通过查找资料才发现，需要给子进程重新建立一个栈的位置，然后通过深拷贝到对应位置

- 串操作指令

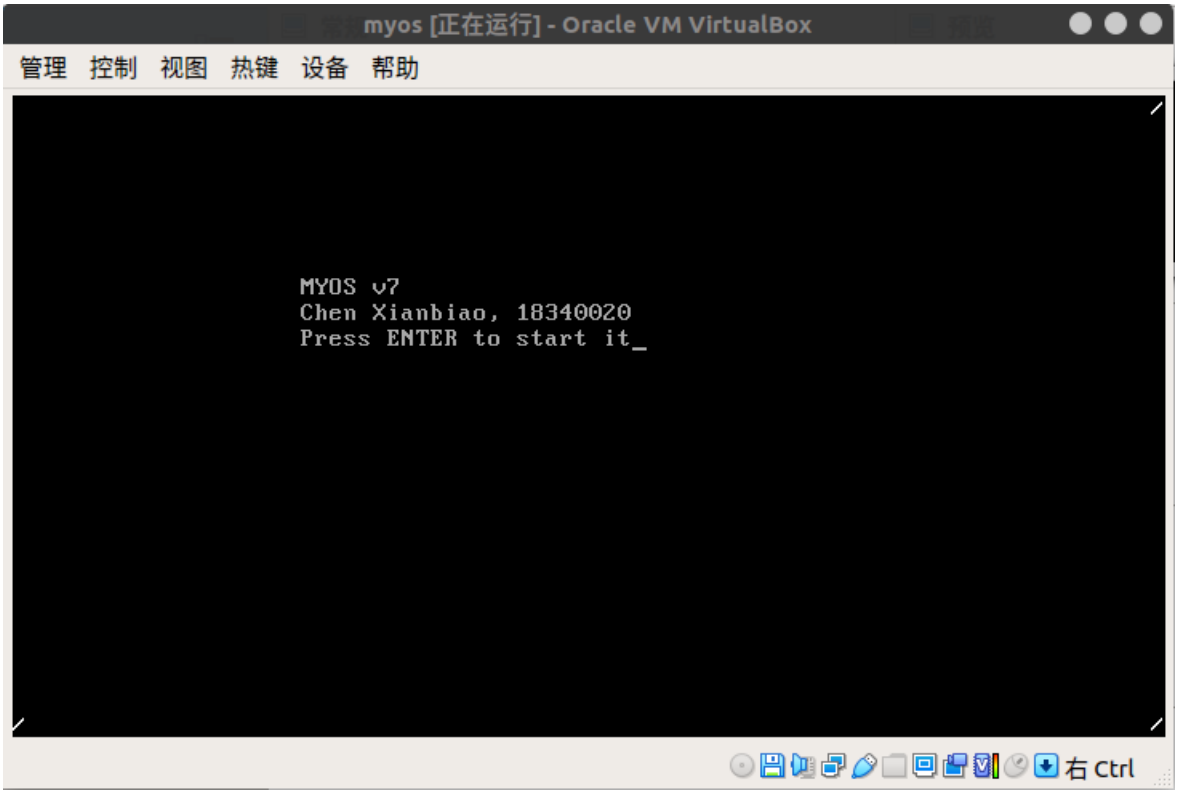
16 位汇编中有`movsw`指令，以字（2 个字节）为单位进行串操作，从 `[ds:si]` 移动到`[es:di]`，每次移动后，`si`和`di`将会递增或递减 2 个单位。指针移动的方向由方向标志位 `DF`决定，该标志位可以用`cld`和`std`设置。通过串操作就能够快速进行深拷贝。

- 父子进程返回值问题

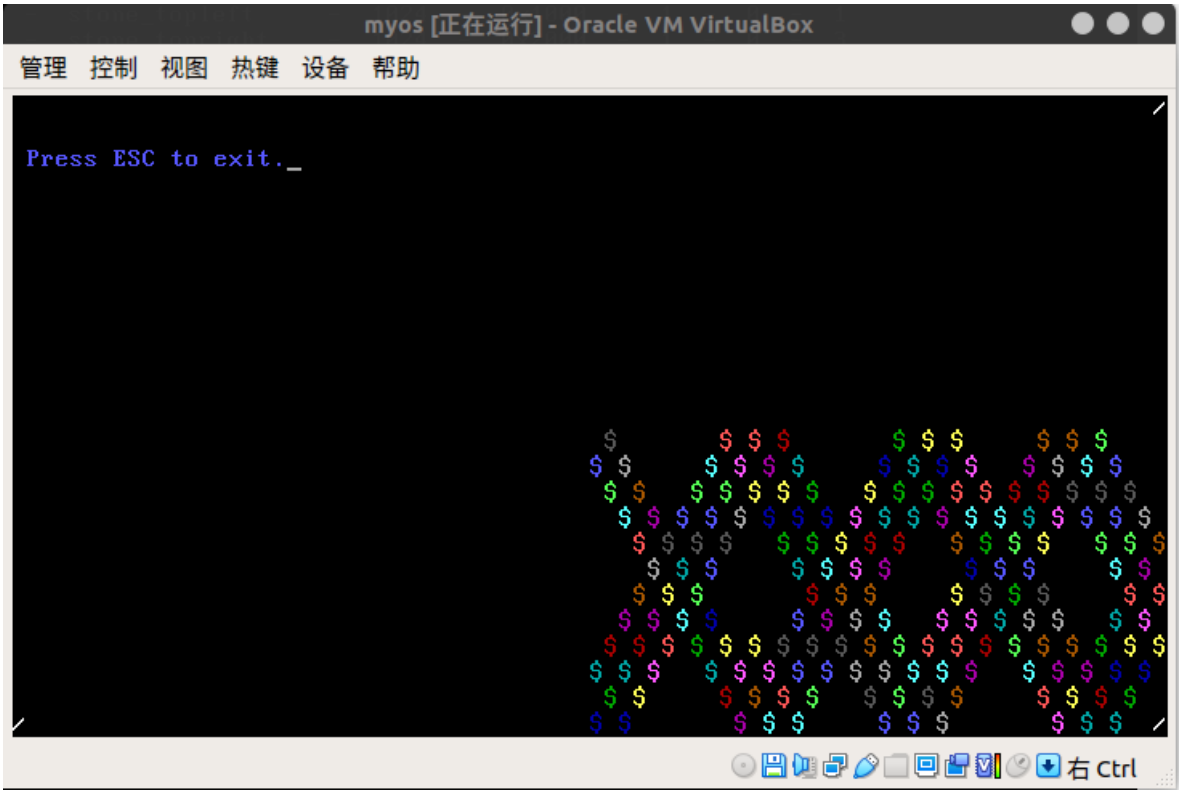
fork是通过系统调用来实现的，而返回值是ax，因此如果我需要把某个特定的值返回到用户程序，则只需要将该值在系统调用期间赋值到对应进程的控制块的ax中，这样就能够成功返回值。用户程序的父子进程的区分就在这里。

2）实验结果展示

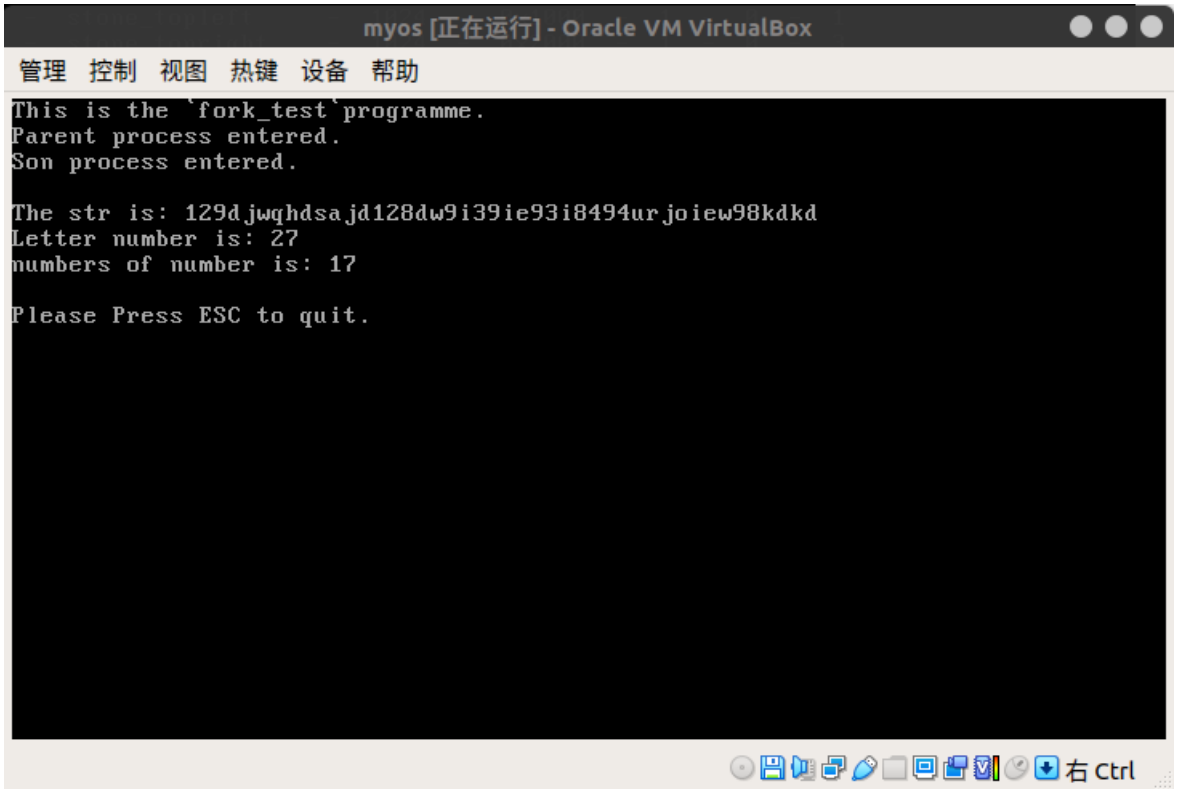
- 启动虚拟机，进入开始画面，可以看到风火轮仍然存在



- 运行 `run 24` 测试之前的用户程序是否可以

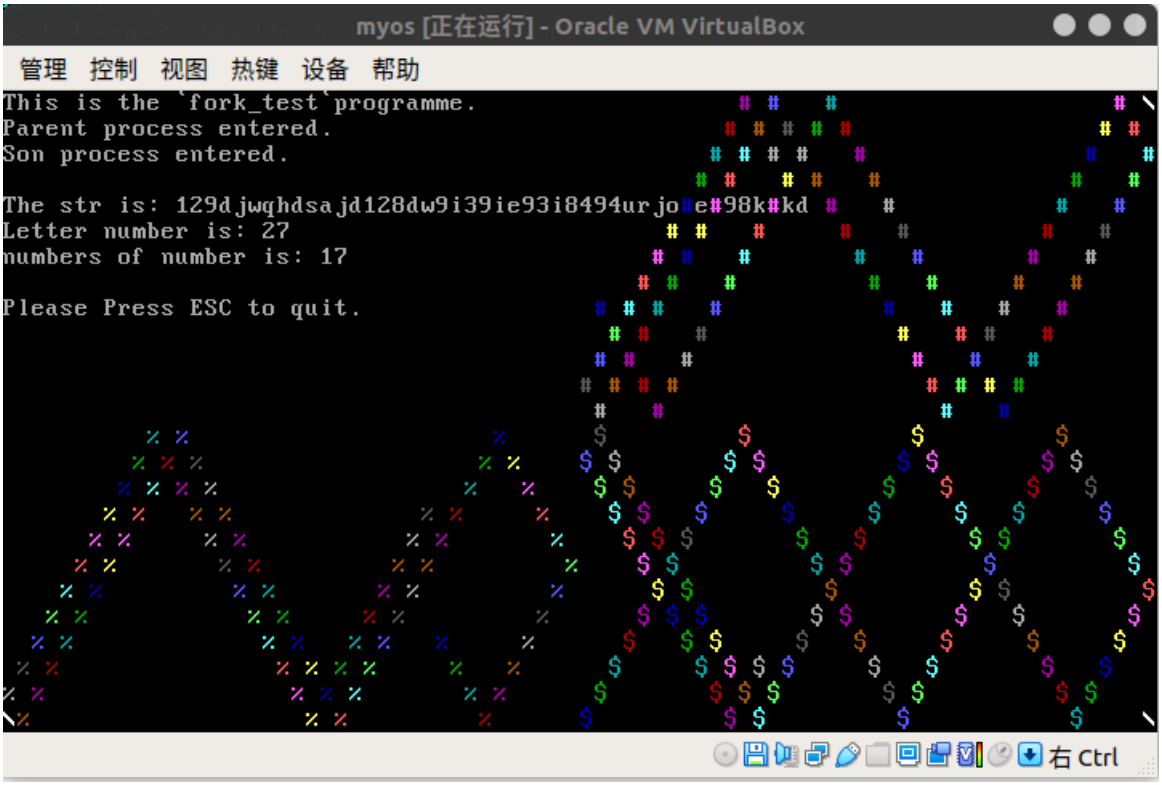


- 本次`fork_test`程序是用户程序7，因此我调用`runall 7`就可以运行



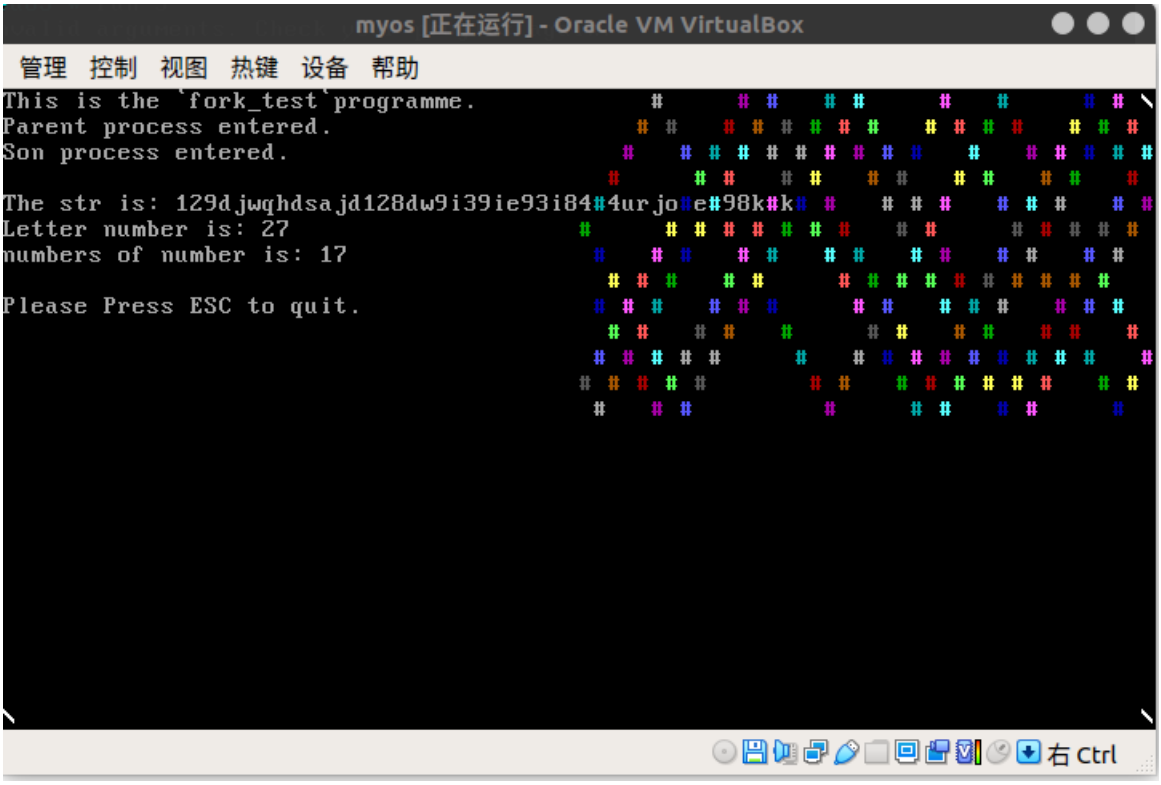
可以看到输出结果，首先程序只有父进程，打印了几行字符串，随后fork出子进程后。父进程进行wait阻塞状态，子进程负责统计字母和数字的数量，统计完后退出唤醒父进程，之后父进程就打印输出字母数量为27个，数字数量为17个，与事实是相符的。

- 测试多个程序同时运行，`runall 2347`



可以看到用户程序234正在运行，并且 fork_test 测试程序也正确地运行

runall 27



- 输入 shutdown 关机

5.实验体会

本次实验七实现了五状态进程模型，进程状态有新建态，就绪态，运行态，阻塞态和退出态，五个状态的转换比起实验六的二状态进程模型要复制一点。

在本次实验中遇到的最大的困难就是fork的过程。父进程派生出一个子进程，父子进程是需要同时使用同一段的代码，但是特别的就是两者需要有独立的堆栈。并且fork之后父子进程需要有两个不同的返回值，实现这个返回值就是将父子进程的进程控制块中的ax修改，这样就能在回到用户进程的时候返回的值是不一样的。

困扰了我最久的一个点就是堆栈的复制问题，本来我使用的方式是逐个复制，但是我发现这样操作过于繁琐，通过查资料才发现有成片统一复制的汇编方式。4

让我越来越深刻的一个知识点就是系统调用的使用。由于我们使用fork是在用户程序上的，但是用户程序是无法访问我们在内核中的代码。fork、wait、exit 过程实现在内核中，用户程序是无法直接使用函数调用的方式来使用，所以要将他们写进中断向量里，然后用户程序就能进行中断系统调用。

本次实验，逻辑思路是比较清晰的，但是实现起来也还是好不容易。在进程切换、创建进程、派生退出进程这类过程中，需要频繁地保护寄存器、恢复寄存器，还要精确地控制程序的执行流，稍微不注意就容易导致错误，使得原有用户程序失效。

6.参考资料

- 进程的五状态模型：https://blog.csdn.net/weixin_30537451/article/details/99768742
- 汇编问题rep movsb 和 cld 是什么意思：https://blog.csdn.net/qq_41076797/article/details/89879242