实验报告——实验七

姓名：任文頔

学号：14322181

院系：数据科学与计算机专业

专业、年级：14级计算机科学与技术

指导教师：凌应标

【实验题目】

五状态进程模型与进程控制原语

【实验目的】

1、掌握五状态进程模型

2、实现进程的并发和同步

【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：

⑴实现控制的基本原语do_fork()、 do_wait()、do_exit()、blocked()和wakeup()。

⑵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.

⑶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 main() {
  int pid;  char ch;  pid=fork();
if (pid==-1) printf(“error in fork!”);
    if (pid) {  ch=wait(); printf(“LetterNr=”); ntos(LetterNr); }
Else {  CountLetter(str);  exit(0);}
}
```

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。

【实验方案】

1 硬件或虚拟机配置方法

系统环境：Linux Ubuntu 14.04

虚拟机配置方法：在Linux下VMware Player是收费软件，因此选择免费的VirtualBox软件。

配置方法：操作系统选择其他，选择从软盘启动，添加自己的软盘，虚拟机名称为14322181renwendi

2 软件工具与作用

（1）汇编语言编译器NASM：针对Intel x86架构的汇编与反汇编程序

（2）C语言编译器GCC：由 GNU 开发的编程语言编译器。

（3）编辑器Vim：功能强大、高度可定制的文本编辑器。

（4）虚拟机软件Bochs：自己编写的操作系统的测试环境

（5）软盘创建工具:dd 软盘写入工具:Linux 自带挂载命令

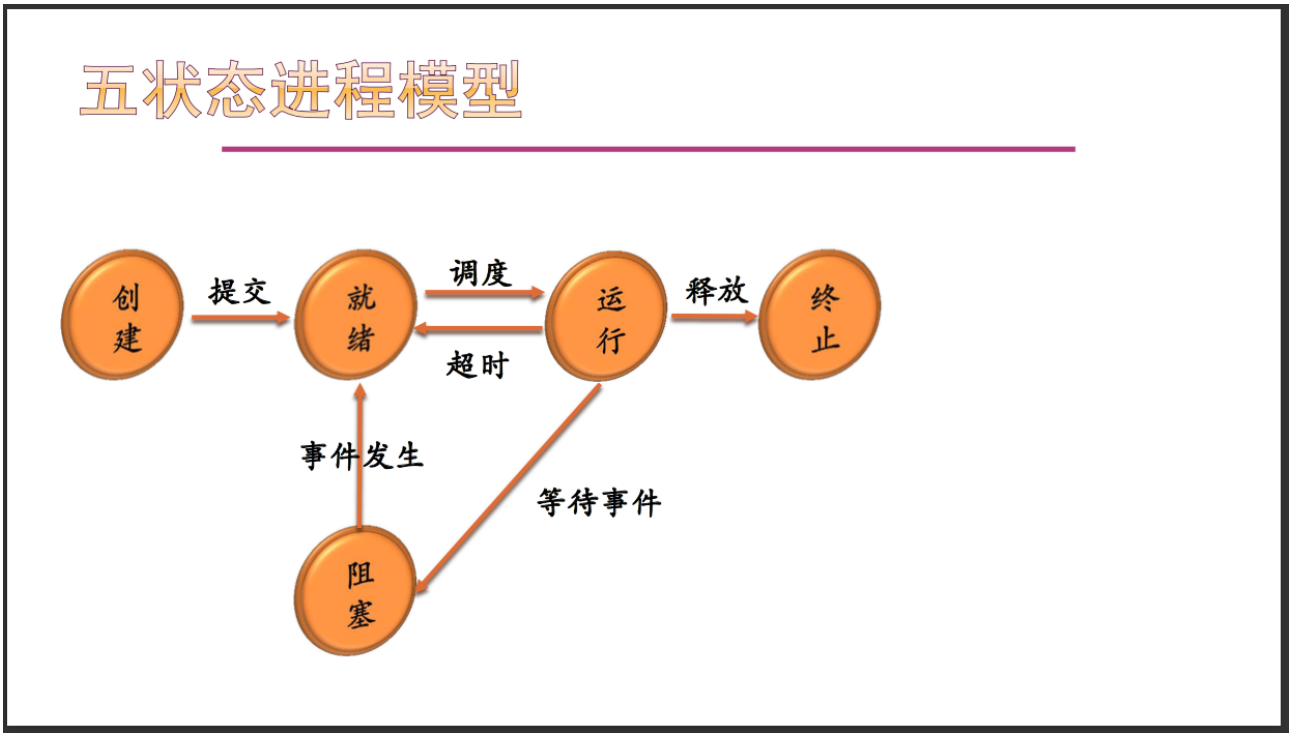
3 方案的思想

在原有实验六的二状态进程模型上，完善进程模型：

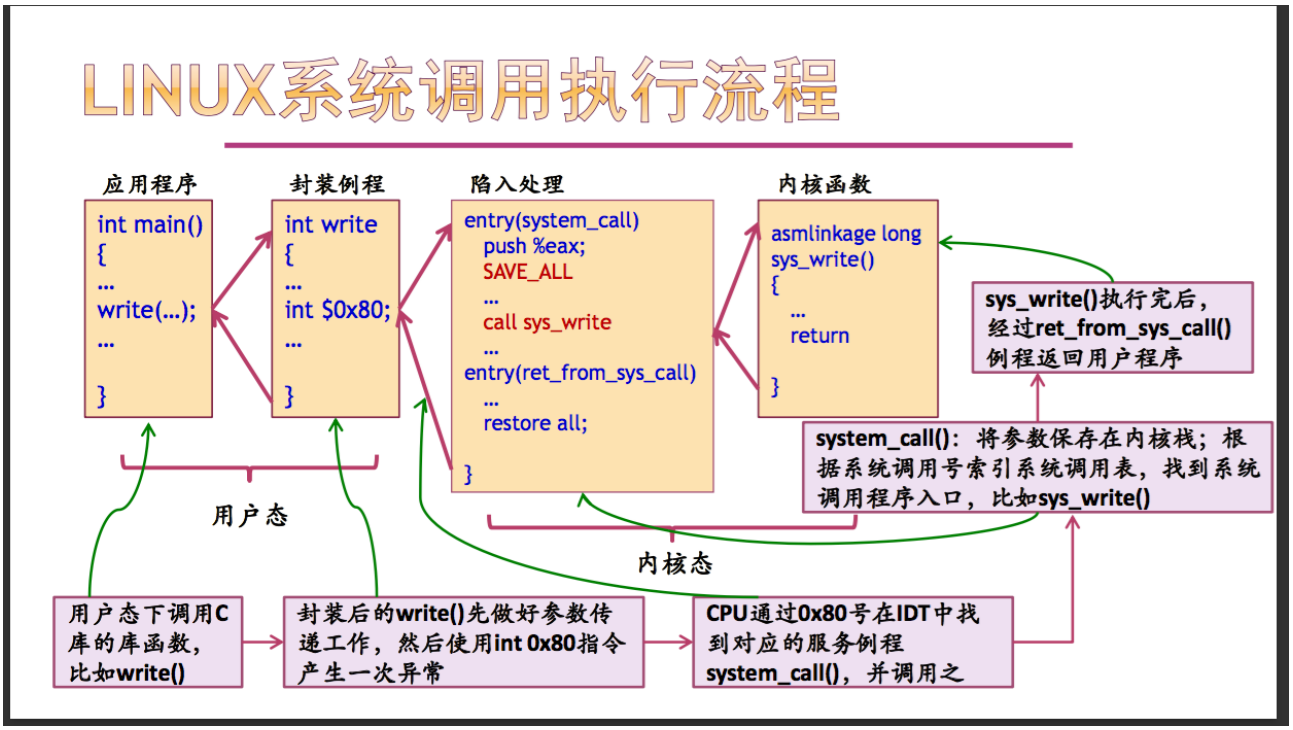
① 扩展PCB结构，增加必要的数据项
② 进程创建do_fork()原语，在c语言中用fork()调用
③ 进程终止do_exit()原语，在c语言中用exit(int exit_value)调用
④ 进程等待子进程结束do_wait()原语，在c语言中用wait(&exit_value)调用
⑤ 进程唤醒wakeup原语（内核过程）
⑥ 进程唤醒blocked原语（内核过程）

4 相关知识原理

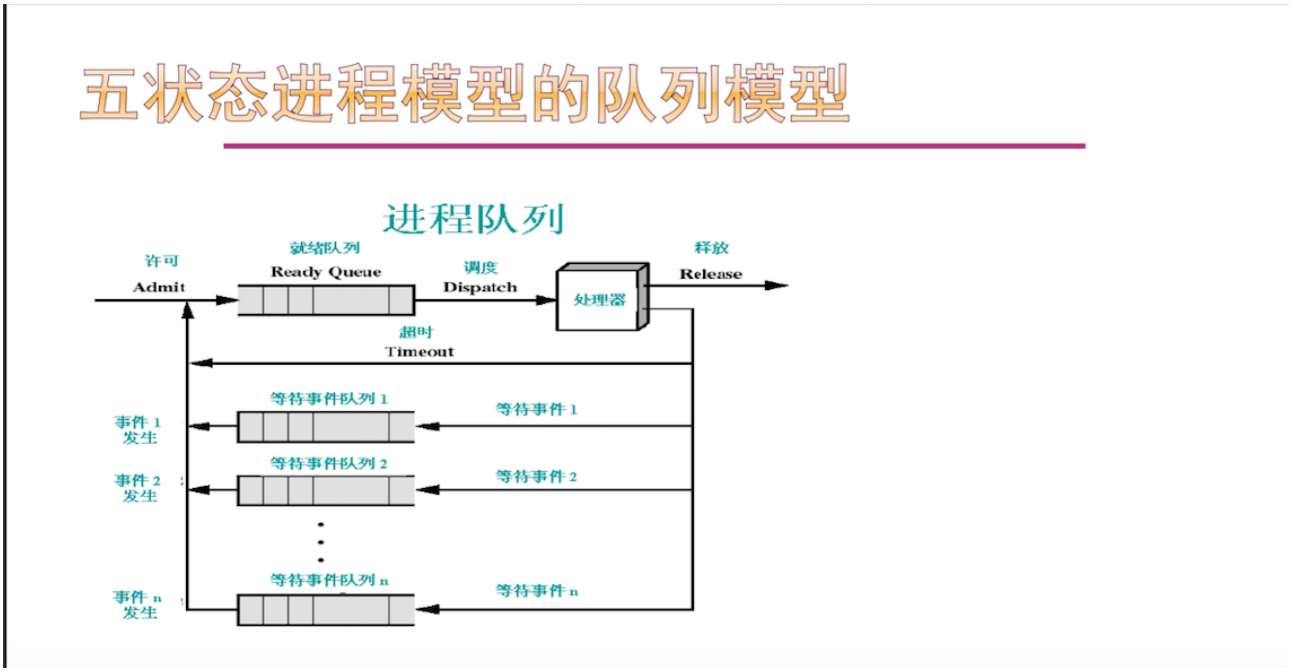
（1）五状态进程模型



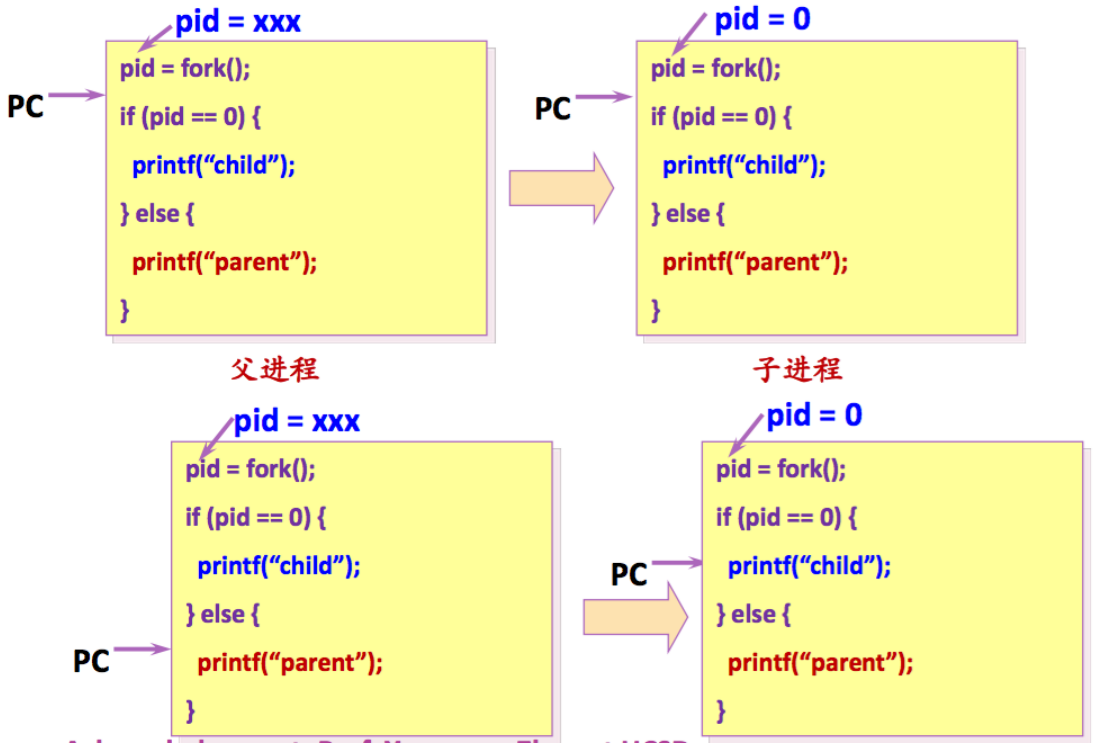
（2）系统调用流程



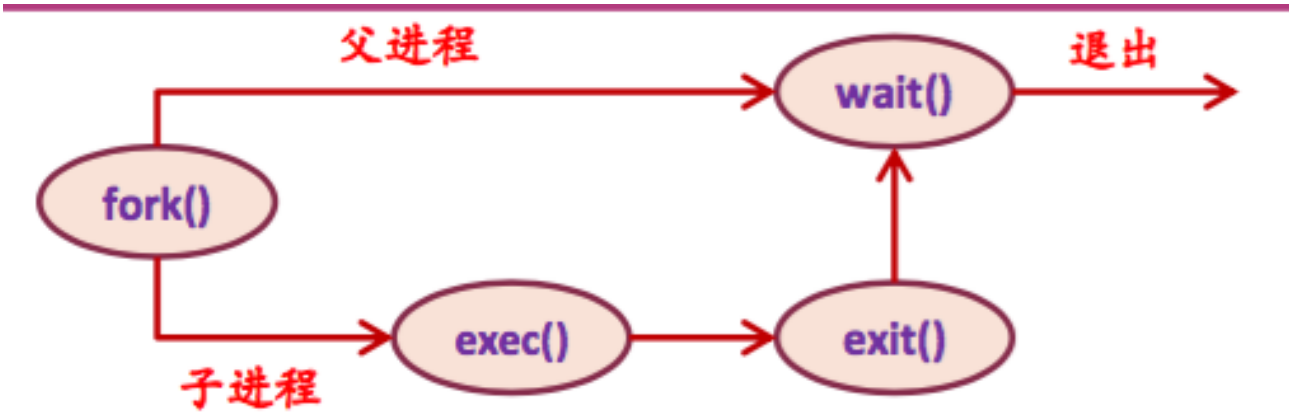
（3）五状态模型的进程队列：



（4）fork（）功能描述



5 程序流程



6 程序关键模块

本次实验的代码组织安排较以往有所不同，增加了一些头文件，并且将内核进行了分块，使得主内核模块os.c的代码量大幅下降，便于查看和修改代码，但是内容并没有发生变化，与实验六相同的部分不再介绍，只展示一下头文件的命名规则和现在的代码模块划分。主要介绍本次实验所需的关键代码。

（1）头文件terminal.h

```
#ifndef _TERMINAL_H
#define _TERMINAL_H
```

```
__asm__(".code16gcc");

extern char gets( char *);
extern char strcmp( char *, const char *);
extern void clear();
extern void time();
extern void date();
extern void man( char *);
extern void run( char *);
extern void syscall_test();
extern void Process();
extern void flag_scroll();
extern void set_pointer_pos();
extern void clear();
extern void init_flag_position();
extern void printToscn( char);
extern void print_message();
extern void print_welcome_msg();

#endif
```

说明：在c文件中#include “terminal.h”即可。

（2）新增进程process6.c

以该进程为散射点，以此说明五状态进程模型的实现过程

```
#include "muti_process.h"
char str[ 80] = "129djwqhdsajd128dw9i39ie93i8494urjoiew98kdkd";
int LetterNr = 0;
void main() {
        __asm__( "sti");
        int pid;
        char ch;
        printf( "\r\n\r\nBefore fork \r\n");
        pid = fork();
        if ( pid == -1) printf( "error in fork!\0");
        if ( pid){
                printf( "\r\nFather process:after fork pid is :");
                printInt( pid);
                ch = wait();
                printf( "\r\n\r\nFather process:LetterNr=");
                ntos( LetterNr);
                exit(0);
        }
        else{
                printf( "\r\nChild process:after fork pid is :");
                printInt( pid);
                CountLetter( str);
                exit( 0);
        }
}
```

在原有实验要求实现的统计字符个数的基础上，增加了三行输出：一个是在fork之前先输出“Before fork”，之后是为了体现出父子进程的合作过程，输出父进程和子进程的pid。

具体的进程间合作过程：fork之后父进程运行了一段时间后被阻塞，然后子进程运行（执行CountLetter函数）直到退出并唤醒父进程，父进程再次执行并把子进程统计的结果打印出来，最后父进程退出。

（3）扩展pcb结构

首先是pcb结构的初始化，增加了部分数据项：

```
#define READY 0
#define RUNNING 1
#define DONE 2
#define BLOCK 3

struct Tss{
	short int SS;
	short int GS;
	short int FS;
	short int ES;
	short int DS;
	short int CS;
	short int DI;
	short int SI;
	short int BP;
	short int SP;
	short int DX;
	short int CX;
	short int BX;
	short int AX;
	short int Stack_END;
	int IP;
	short int Flags;
};

struct pcb{
	struct Tss tss;
	int process_status; // 0 is ready , 1 is runing,2 is done 3 is block
	int process_id;
	int f_pid;
};
struct pcb PCB_queue[ process_num_MAX +1 ];//进程控制模块队列的pcb结构体
void init_pcb( int i, short int current_process_SEG){
	PCB_queue[ i].tss.DS = 0;
	PCB_queue[ i].tss.ES = current_process_SEG;
	PCB_queue[ i].tss.FS = 0;
	PCB_queue[ i].tss.CS = 0;
	PCB_queue[ i].tss.SS = 0;
	PCB_queue[ i].tss.GS = 0;
	PCB_queue[ i].tss.DI = 0;
	PCB_queue[ i].tss.SI = 0;
	PCB_queue[ i].tss.SP = current_process_SEG-4;
	PCB_queue[ i].tss.BP = 0;
	PCB_queue[ i].tss.AX = 0;
	PCB_queue[ i].tss.BX = 0;
```

```
        PCB_queue[ i].tss.CX = 0;
        PCB_queue[ i].tss.DX = 0;
        PCB_queue[ i].tss.IP = current_process_SEG;
        PCB_queue[ i].tss.Flags = 512;
        PCB_queue[ i].tss.Stack_END = current_process_SEG-4;

        PCB_queue[ i].f_pid = -1;
        PCB_queue[ i].process_id = i;
        PCB_queue[ i].process_status = READY;
}
```

（4）do_fork()的实现

参考unix早期的fork()做法，我们实现的进程创建功能中，父子进程共享代码段和全局数据段。
子进程的执行点(CS:IP)从父进程中继承过来，复制而得。
复制父进程的上下文tss之前要保存父进程现在的状态到pcb。

```
short int w_is_r;       //which process is running
short int nw_is_r;      //which next process will run
short int _cs,_flags;
int _ip;
short int _ax,_bx,_cx,_dx,_es,_ds,_sp,_bp,_si,_di,_fs,_gs,_ss;

void do_fork(){
        process_num++;
        PCB_queue[ process_num].f_pid = w_is_r;
        //child -> father
        PCB_queue[ process_num].process_id = process_num;
        //copy_father_Tss

        update_fa();
        //save father regsiters to pcb
        restart_flags();
        __asm__("pop %cx");
        // update fa end
        PCB_queue[ process_num].tss = PCB_queue[ w_is_r].tss;
        //child.tss = father.tss
        PCB_queue[ process_num].tss.SP = _sp + 0x1000;
        PCB_queue[ process_num].tss.AX = 0;
        PCB_queue[ process_num].tss.Stack_END = PCB_queue[ w_is_r].tss.Stack_END
+0x1000;

        sub_stack = (PCB_queue[ process_num].tss.Stack_END-0x200)/16;
        //child_stack_segmentaddress-> find address ss*16+sp
        fa_stack = (PCB_queue[ w_is_r].tss.Stack_END-0x200)/16;
        __asm__("mov $0x104,%cx");
        copy_stack();
        __asm__("pop %cx");

        PCB_queue[ process_num].process_status = READY;
        //child process into READY queue
        restart_ax_pid();
        //child process return pid is self pid
        __asm__("pop %bx");
```

```
        __asm__("pop %bx");
        __asm__("pop %bx");
        __asm__("pop %bx");
        __asm__("jmp *%bx");
}
```

copy_stack模块,使用串操作复制父进程栈的内容到子进程栈

```
global copy_stack
extern child_stack,fa_stack
copy_stack:
        push ax
        push es
        push ds
        push di
        push si
        push cx
        mov ax,[ child_stack]
        mov es,ax
        mov edi,4

        mov byte [es:di],0x12   ;3df0

        mov ax,[ fa_stack]
        mov ds,ax
        mov esi,4
        cld
        rep movsw                       ;ds:si  ->  es:di
        pop cx
        pop si
        pop di
        pop ds
        pop es
        pop ax
ret
```

update_fa函数，更新父进程

```
void update_fa(){
        saveall_reg();                  //*** do not inclue sp
        __asm__("pop %cx");
        saveToqueue();                          //code order don't change

        __asm__("mov %sp,%dx");   //save fa ip flags  cs sp
        __asm__("add $22,%sp");
        __asm__("pop %ax");
        __asm__("pop %bx");
        __asm__("pop %cx");

        saveall_reg_seg();              //include sp
        __asm__("pop %cx");
        __asm__("mov %dx,%sp");
        PCB_queue[ w_is_r].tss.SP = _sp;
        PCB_queue[ w_is_r].tss.IP = _ip;
        PCB_queue[ w_is_r].tss.CS = _cs;
        PCB_queue[ w_is_r].tss.Flags = _flags;
}
```

（5）do_wait()的实现

改变进程状态即可。

父进程如果想等待子进程结束后再处理子进程的后事，需要一个系统调用实现同步。我们模仿UNIX的做法，设置wait()实现这一功能。

相应地，内核的进程应该增加一种阻塞状态。当进程调用wait()系统调用时，内核将当前进程阻塞，并调用进程调度过程挑选另一个就绪进程接权。

```
void do_wait(){
        PCB_queue[ w_is_r].process_status = BLOCK;
        __asm__("int $0x1c");
        __asm__("pop %ax");
        __asm__("pop %ax");
        __asm__("pop %ax");
        __asm__("jmp *%ax");
}
```

（6）do_exit()的实现

改变进程状态即可。

父进程如果想等待子进程结束后再处理子进程的后事，需要一个系统调用wait()，进程被阻塞。而子进程终止时，调用exit()，向父进程报告这一事件，可以传递一个字节的信息给父进程，并解除父进程的阻塞，并调用进程调度过程挑选另一个就绪进程接权。

```
void do_exit(){
        PCB_queue[ w_is_r].process_status = DONE;
        PCB_queue[ PCB_queue[ w_is_r].f_pid].process_status = READY;
        __asm__("int $0x1c");
        __asm__("pop %ax");
        __asm__("pop %ax");
        __asm__("pop %ax");
        __asm__("jmp *%ax");
}
```

（7）进程调度模块的修改

调整了对不同进程状态的筛选，禁止将CPU交权给阻塞状态的进程。

```
void schedule(){
        saveall_reg();                  //***do not inclue sp
        __asm__("pop %cx");
        __asm__("pop %eax");

        if( PCB_queue[ w_is_r].process_status == RUNNING){
                PCB_queue[ w_is_r].process_status = READY;
        }
        while(1){
                if( w_is_r == 0)
                        nw_is_r = start_process_num;
                else
                        nw_is_r = w_is_r + 1;

                if( nw_is_r > process_num){
                        nw_is_r = start_process_num;
                }
                if( PCB_queue[ nw_is_r].process_status == READY) break;
```

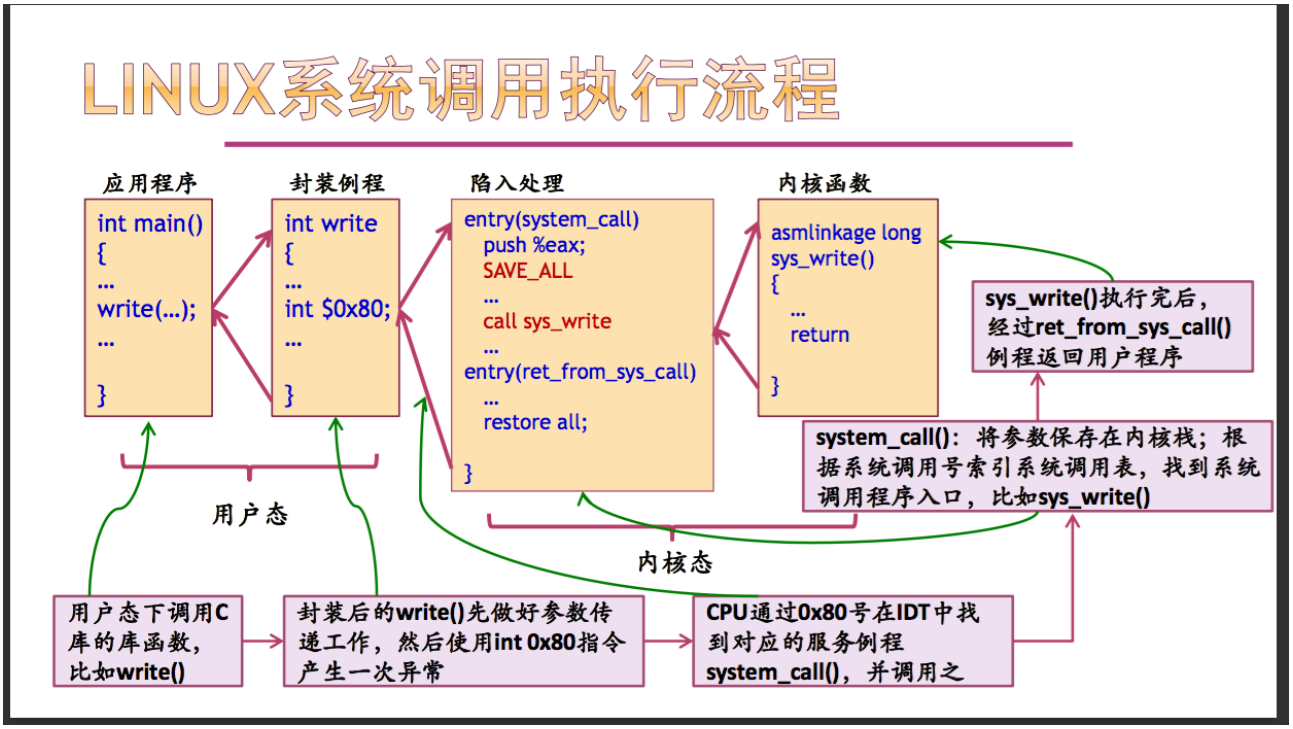
```
            w_is_r = nw_is_r;
        }

        PCB_queue[ nw_is_r].process_status = RUNNING;

        saveToqueue();                          //code order don't change

        //----------------set ip cs flag--------
        __asm__("pop %ax");
        __asm__("pop %bx");
        __asm__("pop %cx");

        saveall_reg_seg();              //include sp
        __asm__("pop %cx");

        if( _di == 0x1234){
            isProcessRun = 0;                       //shut down process
            nw_is_r = 0;
            process_num --;
            backto_os();
        }else{
            isProcessRun = 1;
        }


        PCB_queue[ w_is_r].tss.SP = _sp;
        PCB_queue[ w_is_r].tss.IP = _ip;
        PCB_queue[ w_is_r].tss.CS = _cs;
        PCB_queue[ w_is_r].tss.Flags = _flags;

        //-----------------end-----------------
        _ip = PCB_queue[ nw_is_r].tss.IP;
        _cs = PCB_queue[ nw_is_r].tss.CS;
        _flags = PCB_queue[ nw_is_r].tss.Flags;
        _sp = PCB_queue[ nw_is_r].tss.SP;

        restart_reg_seg();              //include sp
        __asm__("pop %cx");

        queueTodata();                  // ax bx cx...

        w_is_r = nw_is_r;                       //change now running process
        restart_reg();
        __asm__(" pop %di");            //don't use di in any process is dangerous

        __asm__(" jmp schedule_end");
        while(1);
}
```

（8）系统调用

在内核中新增加一个文件os_syscall.asm来实现系统调用

1号、2号、3号分别对应wait, fork, exit

原理如图：



系统调用号示例 (INCLUDE/ASM-I386/UNISTD.H)

```
#define __NR_exit          1
#define __NR_fork          2
#define __NR_read          3
#define __NR_write         4
#define __NR_open          5
#define __NR_close         6
#define __NR_waitpid       7
#define __NR_creat         8
#define __NR_link          9
#define __NR_unlink        10
#define __NR_execve        11
#define __NR_chdir         12
#define __NR_time          13
...
```

```
[bits 16]
extern main             ;forbid run this file any time
jmp main
```

```
global syscall_init,run_syscall
extern syscall_num
extern do_fork, do_wait, do_exit

setting_up_syscall:
        mov bx,0
        mov es,bx
        mov al,ah
        mov ah,0
        shl al,2
        mov bx,0xfe00
        add bx,ax
        mov [es:bx],ecx
ret

syscall_init:

;----#1 syscall(wait)
        mov ah,1
        mov ecx,0
        mov cx,do_wait
        call setting_up_syscall

;----#2 syscall(fork)
        mov ah,2
        mov ecx,0
        mov cx,do_fork
        call setting_up_syscall

;----#3 syscall(exit)
        mov ah,3
        mov ecx,0
        mov cx,do_exit
        call setting_up_syscall

ret

run_syscall:
        cli
        mov ax,0
        mov es,ax
        mov al,[ syscall_num]
        shl ax,2
        mov bx,0xfe00
        add bx,ax
        call [ es:bx]
        sti
ret
```

在中断初始化中增加80号异常：

```
        ;#5 int 80
        mov ax,0x80
        mov [ interrupt_num], ax
        mov ax, process_int80
        mov [ interrupt_vector_offset],ax
```

```
        call insert_interrupt_vector
```

process6.c中进行调用：

```
char wait(){
        __asm__("cli");
        __asm__("mov $1,%ah");
        __asm__("int $0x80");
        __asm__("sti");
        return pid;
}

char fork(){
        __asm__("cli");
        __asm__("mov $2,%ah");
        __asm__("int $0x80");

        pid = return_ax_Tpid();
        __asm__("pop %cx");
        __asm__("sti");
        return pid;
}

void exit( char x){
        __asm__("cli");
        __asm__("mov $3,%ah");
        __asm__("int $0x80");
        __asm__("sti");
        while(1);
}
```

（9）统计字符串个数

```
void inline CountLetter( char *str){
        char i;
        for (i = 0;i<80;i++){
                if( str[ i] <='z' && str[ i] >='a'){
                        LetterNr++;
                }
        }
}
```

【实验过程】

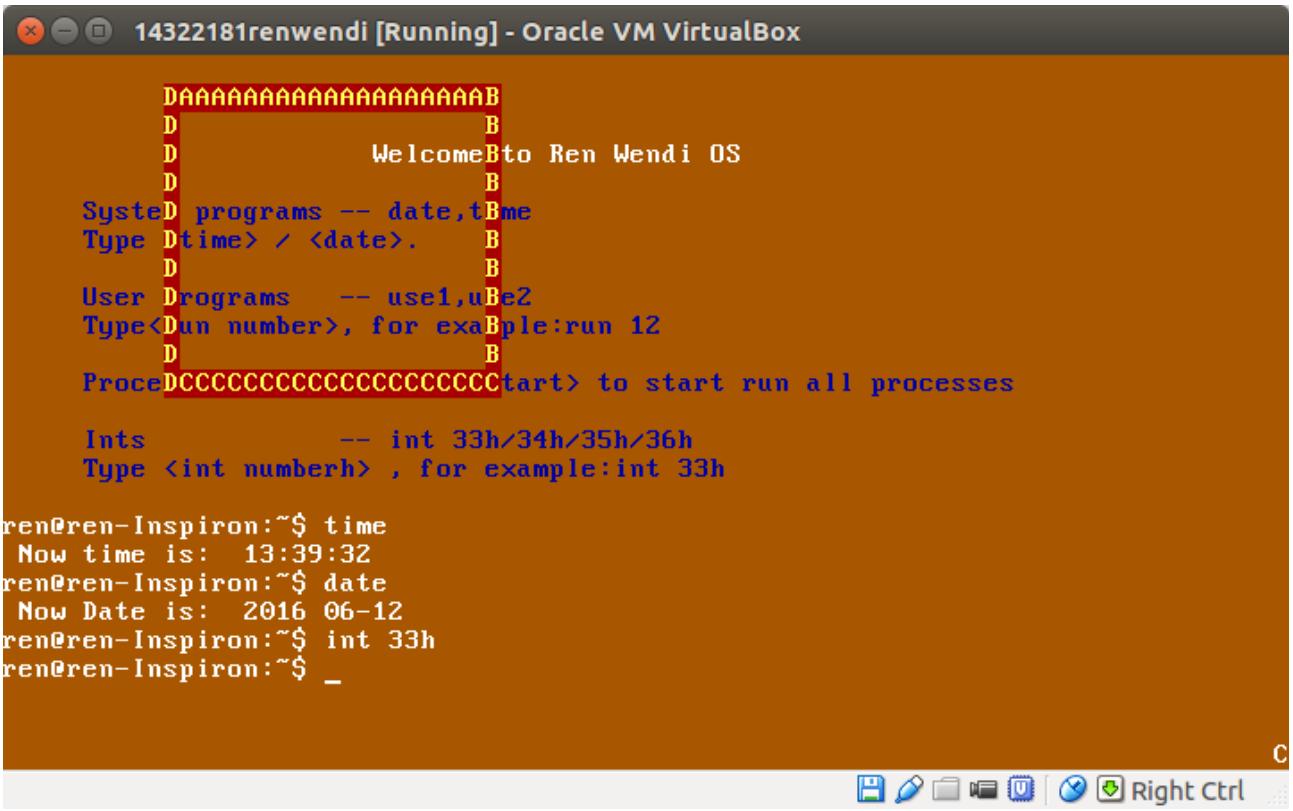
1 输出说明

（1）旧功能展示

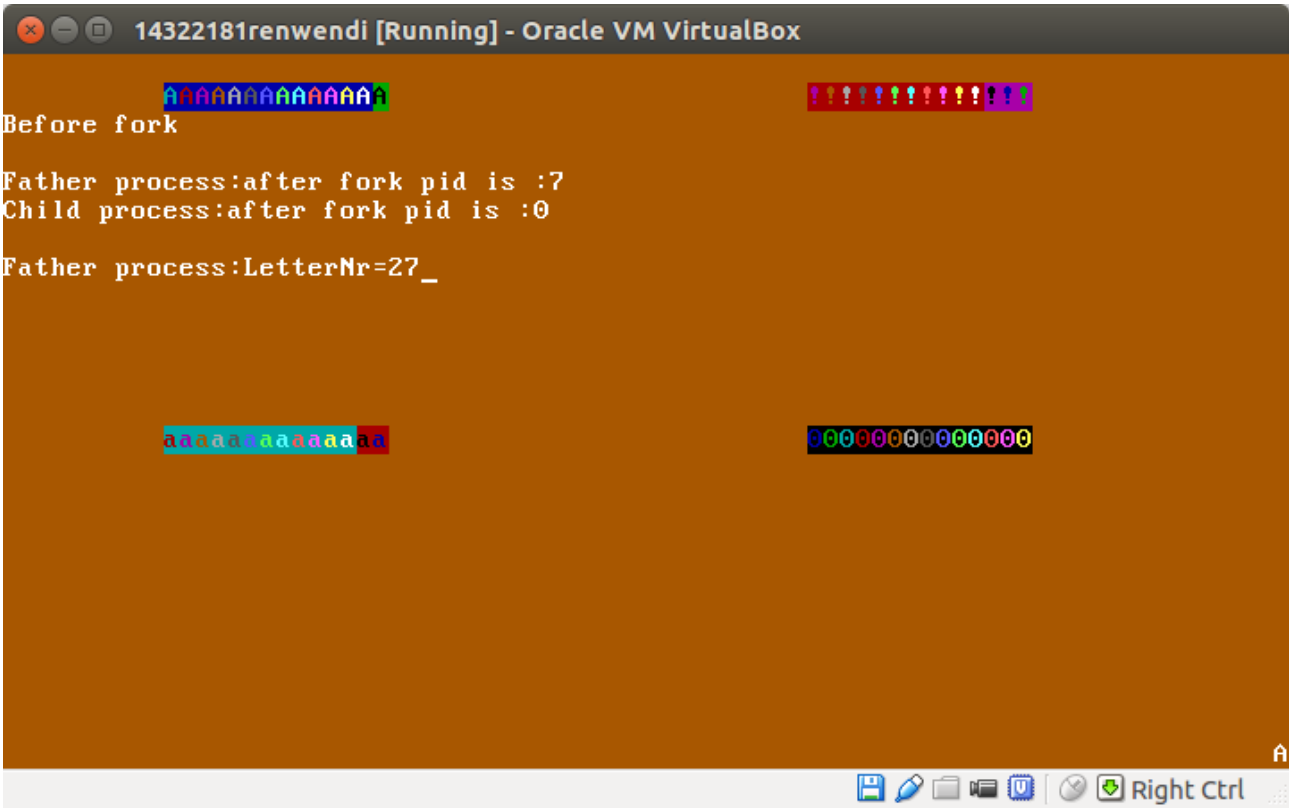
包括时间、日期、用户程序1和2

ouch键盘中断

右下角时钟中断

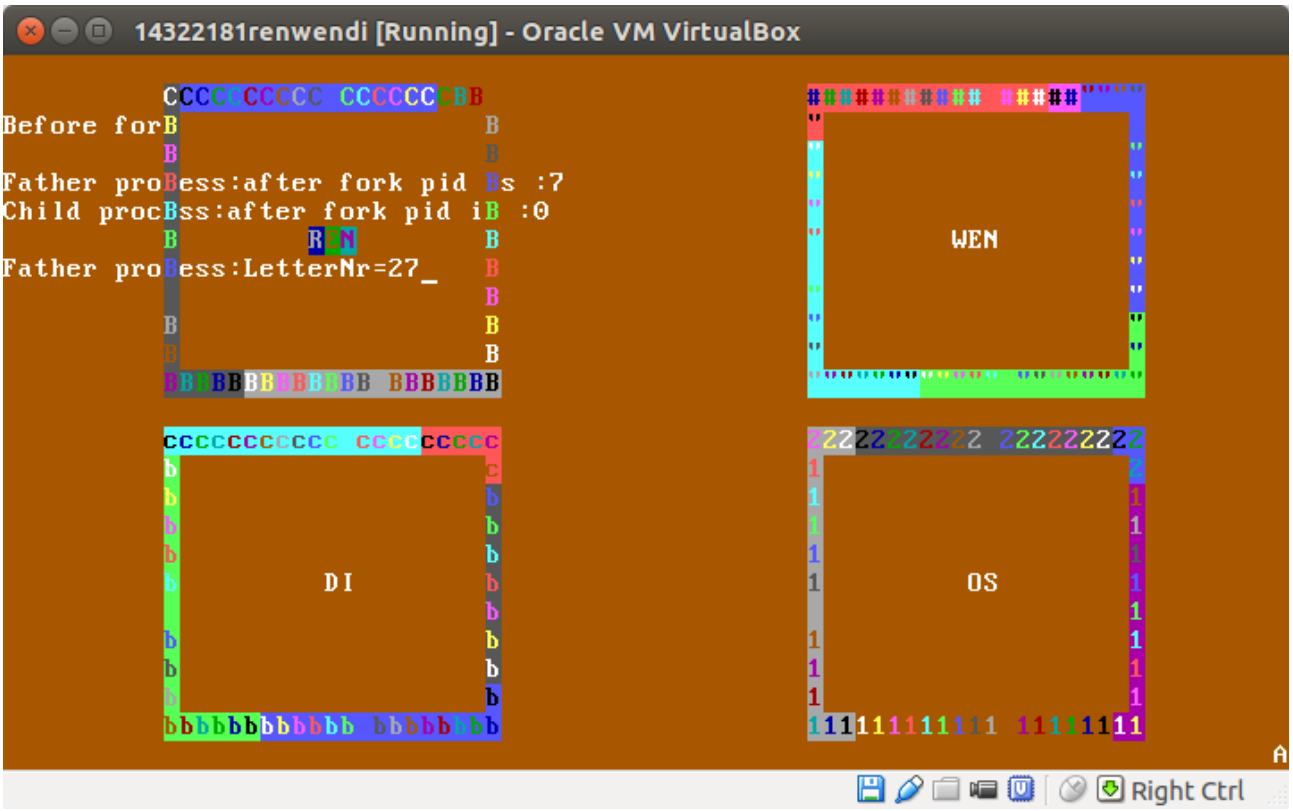


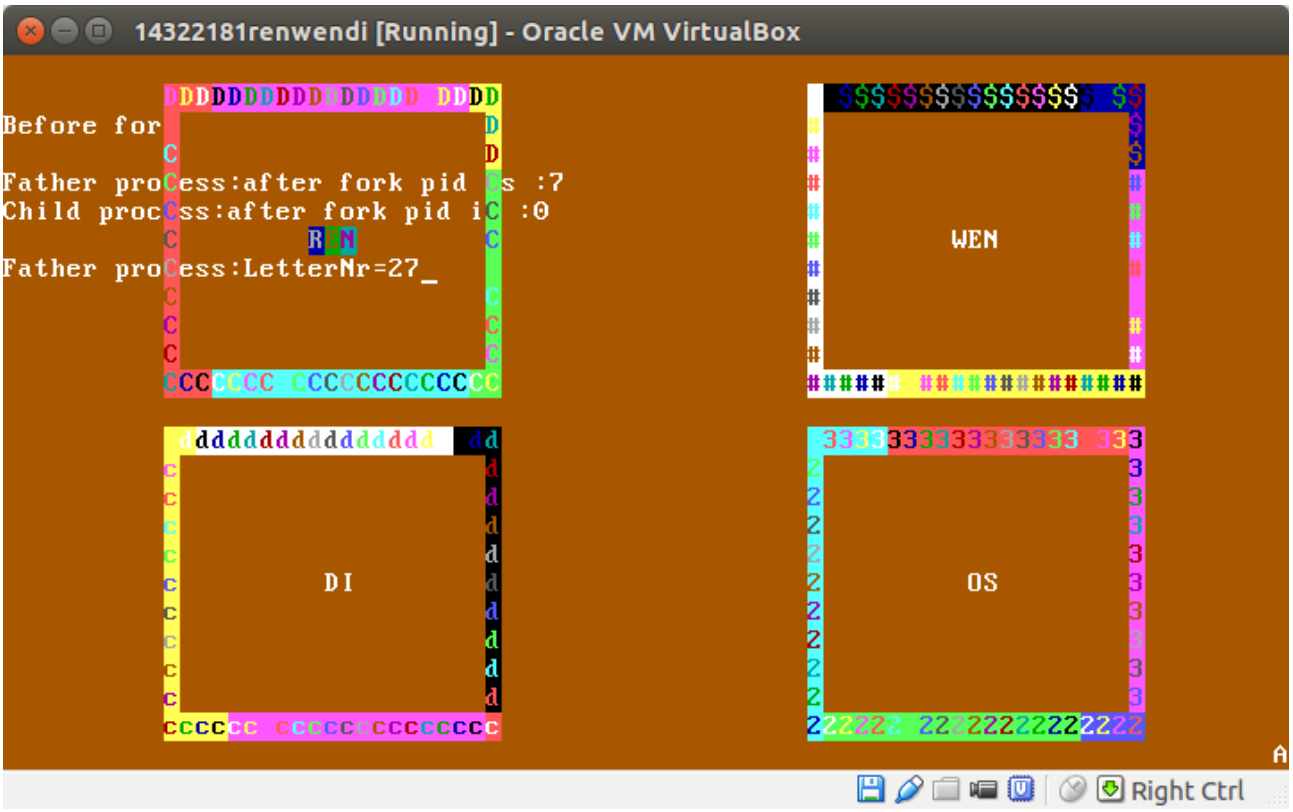
（2）输入start运行六个进程



可以看到第六个进程的运行顺序

父进程——子进程——父进程，最后统计字符个数为27





4 遇到的问题及解决情况

（1）编译链接中遇到的问题

这一类问题主要是因为新增加的一些函数没有修改makefile文件所导致，较为容易解决。

```
ren@ren-Inspiron:~/Documents/OS/pro7$ make
nasm -f elf kernel/os.asm -o os.o
gcc -m32 -mpreferred-stack-boundary=2 -ffreestanding -c kernel/os.c -o osc.o
nasm -f elf kernel/oslib.asm -o oslib.o
gcc -m32 -mpreferred-stack-boundary=2 -ffreestanding -c kernel/osclib.c -o oscli
b.o
gcc -m32 -mpreferred-stack-boundary=2 -ffreestanding -c kernel/process.c -o proc
ess.o
ld -m elf_i386 -Ttext 0x7e00 --oformat binary os.o osc.o oslib.o osclib.o proces
s.o -o os.bin -e main
osc.o: In function `main':
os.c:(.text+0x2e2): undefined reference to `syscall_init'
make: *** [os.bin] Error 1
ren@ren-Inspiron:~/Documents/OS/pro7$ _
```

还有一个完全因为惯性，忘记新的进程是c语言的，直接用了汇编编译。

```
54 process6.com:process6.asm
55     nasm -f bin $^ -o $@
```

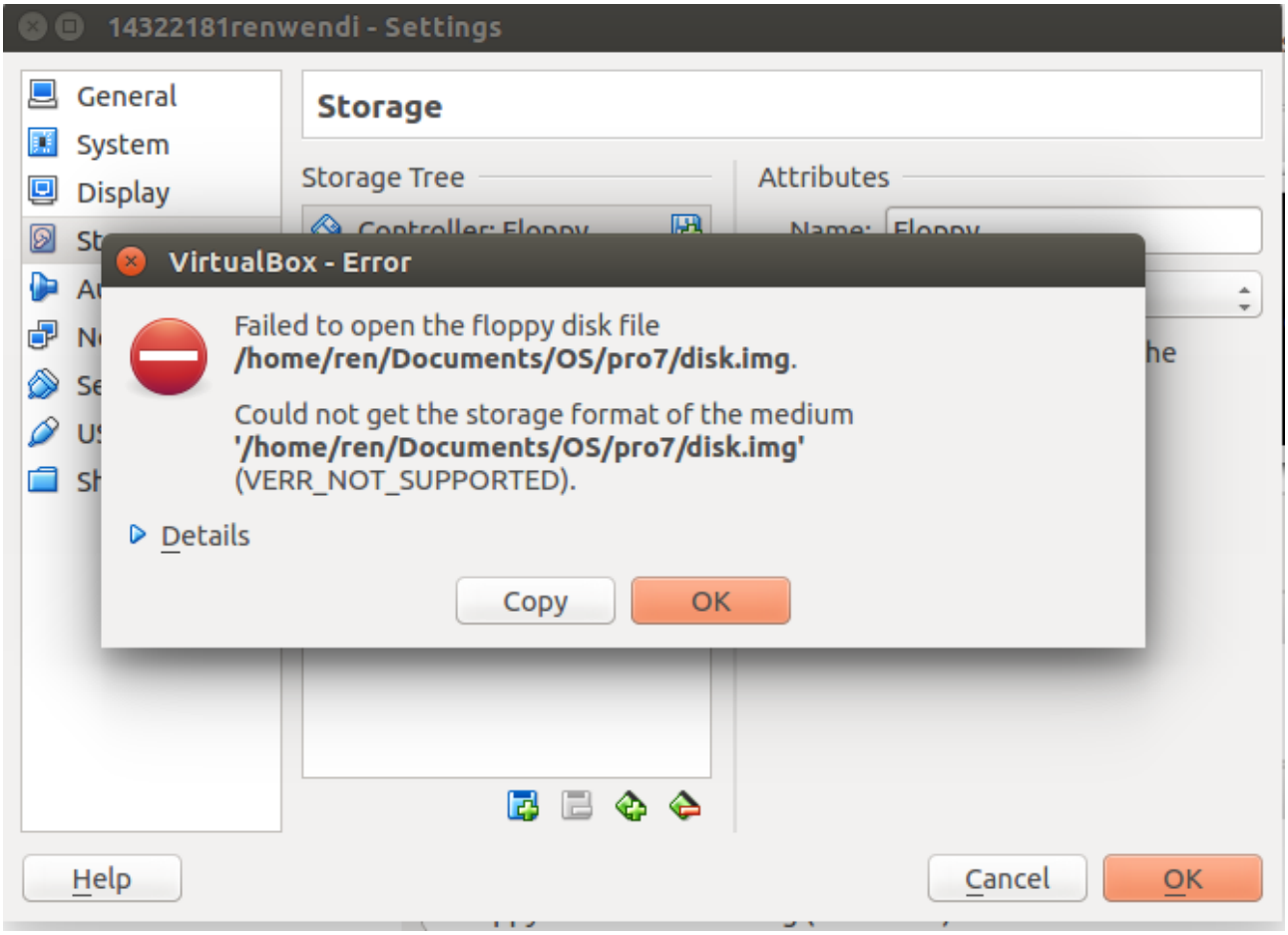
（2）接着上一个问题，将process6用gcc编译之后发现有一个严重的问题在于，该进程要调用内核中相关的函数，因此就需要用到内核的一些依赖。起初写的是将所有的内核函数都添加到依赖，也就是

process6.com:process6.elf os.o osc.o oslib.o osclib.o

    ld $(LD_flags) -Ttext 0x3000 --oformat binary $^ -o $@ -e main

但是这样会导致用户进程代码量过大，所以在改成这样之后产生了非常奇怪的结果，make之后在终端编译的语句还是正常的，但是生产的软盘会在虚拟机报错。

截图如下：



然后我就修改了内核的文件模块，既然是在修改process6.c的编译链接语句之后出现了问题，所以就认为是这里的错误，可能是链接需要的依赖文件过多。

所以就将内核中用户进程需要调用的函数封装在了osclib_share.c和oslib_share.asm中，使得编译语句变为

process6.com:process6.elf osclib_share.o oslib_share.o

在这之后要注意一些问题，我之前没有删除内核中一样的函数，因此会报错，重复定义。还有就是在内核的编译链接语句中也要加上osclib_share.o oslib_share.o。

osclib_share.c实现的函数：

```
void puts(char *key)
void putch( char ch)
char * itoa( short int value)
char gets( char *key)
void print_str( const char *p , unsigned short int l)
char strlen( char*p)
```

oslib_share.asm实现的函数：

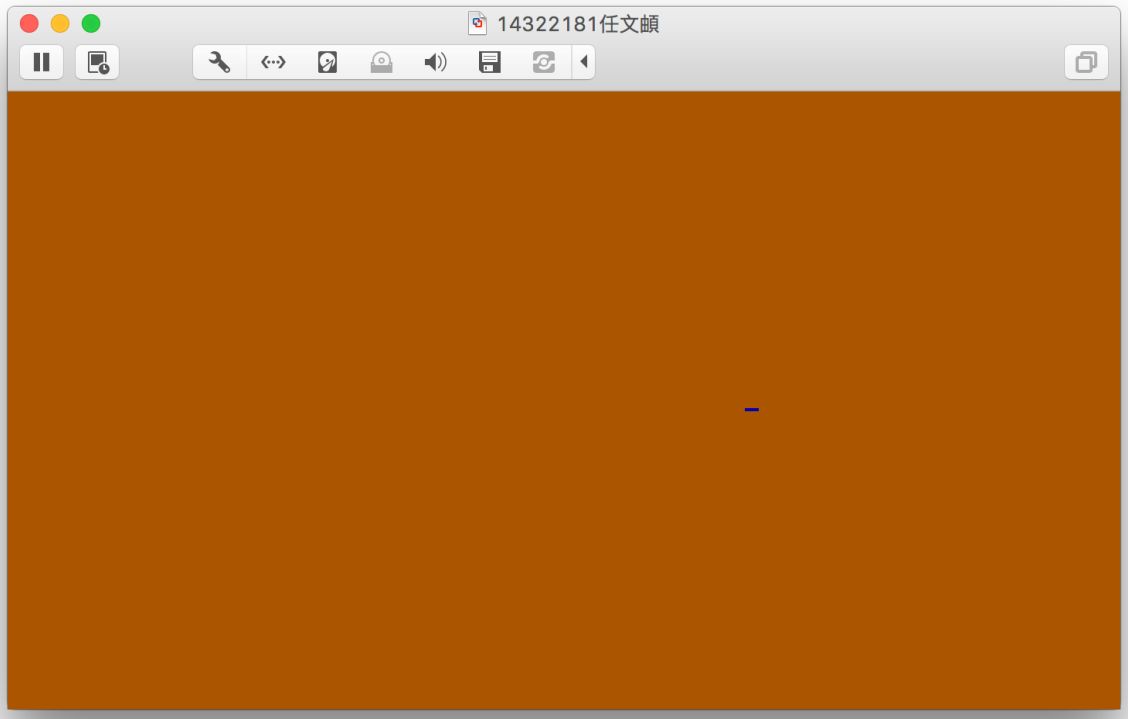
```
global screen_init
global input_char
global printToscn
```

global get_pointer_pos
具体的代码与之前实验的相同，不再说明。

【实验总结】

有了上一个实验的进程模型之后，本次实验的进行就顺利一些了。本次学习理论知识我是看了同学推荐的网上的公开课，对系统调用、五状态进程模型、进程队列等有了更进一步的认识。

本次实验遇到最大的问题就是虚拟机打不开软盘，我在网上搜索了相应的解答，但是没有找到完全符合的，也就按照自己的理解试着修改了代码，与此同时，重新安装了一遍virtual box，因为我在其他电脑上用虚拟机（VMware）打开软盘是可以成功的，只是显示有问题，如图：



此外，也是在做完实验之后才看到老师对上一个实验的评价，来不及进行修改和进一步的完善了，但是按照之前的方法还是能实现本次实验需要完成的功能。

调度算法和进程控制模块的存储有待优化。还有一个与实验八相关的问题，如果时间片很小，父进程在执行到 wait() 之前就被调度，接下来子进程完成了整个程序的执行，然后exit()，然后父进程重新开始执行,那么此时父进程执行了wait()，变成阻塞态，接下来一直无法运行，也没有子进程来更改父进程的状态，父进程就会无限等待下去。所以，这个问题应该要靠使用信号量的PV操作来解决。不确定还能不能完成实验八，如果时间来得及的话还是希望能够实现。

【参考文献】

于渊 《Orange's——一个操作系统的实现》

陈向群 MOOC 《操作系统原理》