实验 7：多进程模型

姓名：姜洋帆　　学号：17341068
院系：数据科学与计算机学院　专业：17 级计算机科学（大数据）

【实验题目】进程控制与通信

【实验目的】

1. 熟悉五状态进程模型
2. 实现父子进程通信（控制、同步）

【实验要求】

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。

【实验方案】

一、 硬件及虚拟机配置：Lenovo PC ； Oracle Virtual Box

二、 软件工具及作用：

Notepad++ ：写代码

Nasm：编译引导程序的 asm 文件到 bin 文件

Tasm：编译 16 位的内核代码

Tcc：编译 16 位 C 语言，实现 C 语言与 x86 联合编译

Sublime ：查看、编辑二进制文件，将编译完成的程序机器码写入软盘

三、 程序功能

Myos1 作为引导程序，设置好内核的偏移和段地址，加载内核程序，然后跳入内核程序，进行下一步操作。在内核模式下，命令行输入 test，运行指定的用户程序（使用父子进程通信的方式实现字符串计数）。用户程序中，父进程首先进行一些函数的定义以及变量的初始化，这部分代码是父子进程共享的。之后执行 fork，父进程执行 wait 等待子进程执行结束，而子进程执行字符串计数的相关代码，执行结束后将结果保存在父子进程共享的变量中，然后退出，将父进程唤醒（内核 exit 过程中实现，调用 wakeup），父进程将值打印，然后退出。

四、 程序设计

本次实验涉及到编写一个新的用户程序（C、86 汇编交叉编译），同时还需要修改内核的 PCB 数据结构的相关代码、调度部分的代码以及时钟中断，同时增加 fork，wait，exit 三个过程给用户程序调用（封装为系统调用）。

由于涉及到用户程序的多进程协作，主要涉及到两个部分：一个是内核的 fork 等操作的实现以及调度功能的更改、完善，另一个是用户程序的实现以及装载等功能。

同时，将上次多进程模型的实验进行了大规模的修改，首先是调度部分，没有采用原先单一的加减来操作，因为涉及到多个进程执行顺序，结束时间是不确定的，所以编写了一些 for 语句、增加了数据结构来判断每个进程资源、段地址是否可用，来实现更精细的调度功能。其次将一些汇编的代码封装成函数来给内核的 C 代码调用（重启 PCB 等）

有一些汇编实现、涉及到参数传递的函数，没有用栈来传递参数，为了方便调试和编写代码，这里偷懒直接使用了全局变量来实现（函数头没有定义参数）

大致情况如下：

内核部分：

1. 增加了一个数据结构 segement_list、segement_used 和函数 select_seg，数据结构用来保存段地址和该段地址是否被使用，函数 select_seg 用来选择出一个当前空闲的段（返回它在 list 中的下标）。
2. 增加状态变量，改为 5 状态模型：NEW,READY,RUNNING,WAIT,EXIT。这里 EXIT 为 0，即 PCB 状态初始值都为 EXIT。为方便起见，当一个 PCB 的状态为 EXIT 时直接就认为是可用的 PCB（即没有考虑僵尸进程的情况）
3. PCB 增加 ID，FID，SEG 代表自己的 ID，父进程 ID 以及占用的段地址。ID 默认为在 pcb 列表的下标。

4. 编写用户程序装载函数（汇编与 C 交叉调用实现），大致的思路是，首先判断有无空闲的进程块以及段地址，没有则报错并返回，否则就调用汇编写的方法，将进程装载进内存，程序数量+1
5. 重写调度函数 Schedule，因为这次改为 5 状态模型，在调度时需要考虑的状态多了一些：只有当状态为 NEW 或者为 READY 时才会选取这个进程执行。
6. 增加系统调用，将 do_fork 等内核过程封装为 21 号系统调用，供用户程序使用。
7. 将 PCBrestart 改写为函数，供 fork 等函数执行完后调用，从内核跳回用户程序（该函数的功能是将当前进程的上下文（PCB 中的数据）拷贝进寄存器，来使 CPU 执行当前选中的进程）
8. 实现 stackcopy 函数，在 fork 函数执行时，需要调用该函数，将父进程的栈内容拷贝给子进程
9. do_fork,do_wait,do_exit 以及 blocked，wakeup 的实现，do_fork 函数实现需要考虑当前资源占用情况，PCB 拷贝、栈拷贝以及函数返回值这些部分，较为复杂，具体分析见实验报告的后面内容

用户程序部分：

1. 用汇编编写、封装一些最基本的函数，如 printChar，cls，将 21 号中断的 do_fork 中断调用封装为 fork 等函数，共用户程序(C 语言直接调用)。
2. 在 C 语言程序中，实现 printint 函数，打印 16 位以下的整数。实现题目要求的计数程序，用父子进程协作的方式实现：父进程先 fork 出子进程，然后 wait，子进程执行相关的逻辑操作，得出结果后（保存进父子进程共享的一段变量），执行 exit 通知父进程执行结束，然后父进程接着打印出结果，执行 exit 退出，释放资源。

五、主要代码

内核部分

1. PCB 数据结构

增加了 ID,FID,SEG 表示自己的 ID，父进程 ID 以及段地址（在 seg_list 数据结构中下标）

```
typedef struct PCB{
    RegisterImage regImg;
    int Process_Status;
    int ID;
    int FID;
    int SEG;
}PCB;
```

2. 调度函数：

首先将当前进程改为 READY 状态，查询当前所有进程，若 PCB 状态为 NEW 或 READY，则选择它作为下一次运行的程序，若没有程序可以被调度，则跳回内核。（实际上可能所有进程都在 wait 进入死锁，这里就没有考虑这种情况了）

```
void Schedule(){
    pcb_list[CurrentPCBno].Process_Status = READY;

    hflag = 0;
    for (index = 1; index <= Program_Num+1 ; index++) {

        if (((CurrentPCBno + index) % (Program_Num+1)) == 0) continue;

        if (pcb_list[(CurrentPCBno + index) %
(Program_Num+1)].Process_Status == NEW
            || pcb_list[(CurrentPCBno + index) %
(Program_Num+1)].Process_Status == READY) {
            hflag = 1;
            CurrentPCBno = (CurrentPCBno + index) % (Program_Num+1);
            break;
        }
    }
    if (hflag == 0) {
        CurrentPCBno = 0;
        return;
    }

    if( pcb_list[CurrentPCBno].Process_Status != NEW )
        pcb_list[CurrentPCBno].Process_Status = RUNNING;

    return;
}
```

3. 段地址调度：

查询所有段地址，根据 segment_used 数组来确定当前下标对应的段是否已经被占用，若找到空闲的段地址，则返回段地址在 segment_list 中的下标并修改为已经被使用，否则返回-1

```
int select_seg() {
    for (index = 0; index < 7; index++) {
        if(segement_used[index] == 0) {
            segement_used[index] = 1;
            return index;
        }
    }
    return -1;
}
```

4. 封装 PCBRestart 函数

该函数功能是将当前 PCB 中的数据保存进寄存器中，使下一时刻 CPU 执行当前 PCB 保存的进程，在每次调用 fork 后，需要调用该函数，跳回用户程序，让父进程继续执行

```
public _PCBrestart
_PCBrestart proc
;Pre:
    mov ax, cs
    mov ds, ax
    mov es, ax

    call near ptr _Current_Process
    mov bp, ax          ; 汇编 n 调用 C 代码时，C 函数的返回值保存在 AX 里

    mov ss,word ptr ds:[bp+0]
    mov sp,word ptr ds:[bp+16]

    cmp word ptr ds:[bp+32],3 ; bp+32 对应 PCB 结构体里面的 status，3 表示 NEW
    jnz No_First_Time
Restart:
    call near ptr _special
    ; 改寄存器上下文，运行当前 PCB 记录的进程
    ;bp  指向结构体，用来获取寄存器的值
    push word ptr ds:[bp+30]    ; FLAGS
    push word ptr ds:[bp+28]    ; CS
    push word ptr ds:[bp+26]    ; IP
                              ; 调用中断时会先将 FLAGS，CS，IP 入栈
    push word ptr ds:[bp+2]     ; 执行 iret 后会把 SP,SP+2,SP+4 分别出栈到
    push word ptr ds:[bp+4]     ; IP,CS,FLAGS 寄存器
    push word ptr ds:[bp+6]
    push word ptr ds:[bp+8]
    push word ptr ds:[bp+10]
    push word ptr ds:[bp+12]
```

```
    push word ptr ds:[bp+14]
    push word ptr ds:[bp+18]
    push word ptr ds:[bp+20]
    push word ptr ds:[bp+22]
    push word ptr ds:[bp+24]

    pop ax
    pop cx
    pop dx
    pop bx
    pop bp
    pop si
    pop di
    pop ds
    pop es
    .386
    pop fs
    pop gs
    .8086

    push ax
    mov al,20h
    out 20h,al
    out 0A0h,al
    pop ax
    iret

No_First_Time:
    add sp,16
    jmp Restart
endp _PCBrestart
```

5. Stackcopy 函数

被 do_fork 函数调用，将父进程的栈数据复制给子进程。这里父子进程的段地址没有用参数来传递，直接用全局变量来实现了。通过 cld，rep movsw 两条指令，以及 es，ds，cx 寄存器的数值来实现内存的复制，其中_new_ss 为子进程的段地址，_f_ss 为父进程的段地址，重复复制 100h 次（假设栈大小为 100h）

代码如下:

```
public _stackcopy
_stackcopy proc
    push ax
    push bx
    push cx ;作为 rep 的计数器
    push ds
    push si
    push es
    push di

    mov ax, word ptr[_new_ss] ; new_ss
    mov es,ax
```

```
    mov di,0
    mov bx, word ptr[_f_ss] ; f_ss
    mov ds,bx
    mov si,0
    mov cx,0100h

    cld
    rep movsw

    pop di
    pop es
    pop si
    pop ds
    pop cx
    pop bx
    pop ax
    ret
_stackcopy endp
```

6. do_fork 函数：

实现方式为，首先寻找是否有空闲的 PCB 和段地址，若没有则报错并返回。

然后将子进程 PCB 指针指向找到的新的空闲的 PCB，ID 改为 pcb 对应的下标，FID 改为当前进程的 ID，修改状态为 READY，同时将寄存器 AX 赋值为，这样在调用 fork 后，子进程就会得到一个返回值 0。而父进程 AX 则赋值为子进程 ID。之后进行 PCB 的拷贝，其中子进程需要将 SS 寄存器赋值为通过调度得到的段地址，AX 为 0，其它寄存器的值直接拷贝父进程即可，最后执行栈拷贝、增加当前程序数量，并执行 PCBrestart，返回用户程序，执行后续代码。

```
int do_fork() {
    PCB* curr_pro = Current_Process();
    PCB* new_PCB;
    int pcb_num;

    curr_pro->Process_Status = READY;

    if (Program_Num == 7) {
        printf("Program_Num == 7\n\r");
        curr_pro->regImg.AX = -1;
        return -1;
    }
    hflag = 0;
    /* find a free PCB */
    for(index = 1; index <=7; index++) {
        if(pcb_list[index].Process_Status == EXIT) {
            new_PCB = &pcb_list[index];
            hflag = 1;
            break;
        }
```

```
    }

    pcb_num = index;
    /* 没有空闲的进程块 */
    if (!hflag) {
        printf("No free PCB\n\r");
        curr_pro->regImg.AX = -1;
        return -1;
    }

    new_PCB->ID = index;

    new_PCB->Process_Status = READY;
    new_PCB->FID = curr_pro->ID;

    curr_pro->regImg.AX = pcb_num;

    new_PCB->SEG = select_seg();

    if(new_PCB->SEG == -1) {
        printf("No free segment\n\r");
        curr_pro->regImg.AX = -1;
        return -1;
    }

    curr_seg = segement_list[new_PCB->SEG];

    {
    new_PCB->regImg.GS = 0xb800;
    new_PCB->regImg.SS = curr_seg;
    new_PCB->regImg.ES = curr_pro->regImg.ES;
    new_PCB->regImg.DS = curr_pro->regImg.DS;
    new_PCB->regImg.CS = curr_pro->regImg.CS;
    new_PCB->regImg.FS = curr_pro->regImg.FS;
    new_PCB->regImg.IP = curr_pro->regImg.IP;
    new_PCB->regImg.SP = curr_pro->regImg.SP;
    new_PCB->regImg.AX = 0;
    new_PCB->regImg.BX = curr_pro->regImg.BX;
    new_PCB->regImg.CX = curr_pro->regImg.CX;
    new_PCB->regImg.DX = curr_pro->regImg.DX;
    new_PCB->regImg.DI = curr_pro->regImg.DI;
    new_PCB->regImg.SI = curr_pro->regImg.SI;
    new_PCB->regImg.BP = curr_pro->regImg.BP;
    new_PCB->regImg.FLAGS = curr_pro->regImg.FLAGS;
    }
    new_ss = new_PCB->regImg.SS;
    f_ss = curr_pro->regImg.SS;
    /* copy stack */
    stackcopy();

    Program_Num++;

    /* 直接改变寄存器继续 fork 后的代码，
    相当于 PCB 执行后，不执行这之后的语句了，
    父进程 AX 已经改变，所以相当于已经返回了 */
    PCBrestart();
}
```

7. blocked 函数

被 wait 函数调用，将当前进程 PCB 状态改为 wait，执行一次调度函数，然后执行 PCBrestart 运行其它没有被阻塞的进程。

```
void blocked() {
    PCB* curr_pro = Current_Process();
    curr_pro->Process_Status = WAIT;
    Schedule();
    PCBrestart();
}
```

8. do_wait 函数

直接调用 blocked，打印相关信息以供测试和调试。

```
void do_wait() {

    printf("wating...\n\r");

    blocked();

}
```

9. do_exit

将当前进程PCB状态改为EXIT，然后释放相应的段地址（将segment_used相关项改为0），进程数量减1，当前进程编号改为父进程ID，最后执行 wakeup 函数，将修改父进程的状态为 READY

```
void do_exit() {
    PCB* curr_pro = Current_Process();
    printf("exiting...\n\r");
    curr_pro->Process_Status = EXIT;
    segement_used[curr_pro->SEG] = 0;

    CurrentPCBno = curr_pro->FID;
    Program_Num--;
    wakeup();
}
```

10. wakeup

```
void wakeup() {
    PCB* curr_pro = Current_Process();
    curr_pro->Process_Status = READY;
    Schedule();
    PCBrestart();
}
```

11. runtest 函数

在操作系统与用户交互的界面，输入 test 命令，将执行该函数，主要的功能是分配一个空闲的 PCB 和段地址，然后将本次实验测试的用户程序装载进相应的段地址，修改相应的 PCB。其中 load_pro()函数是用汇编实现的一个简单的装载的函数，使用 13H 中断实现，这里就不列出代码了。

```
void run_test() {
    int pcb_num = 0;
    /* find a free PCB */
    for(index = 1; index <=7; index++) {
        if(pcb_list[index].Process_Status == EXIT) {
            hflag = 1;
            break;
        }
    }

    pcb_num = index;
    printf("index\n\r");
    printChar('0' + index);

    /* 没有空闲的进程块 */
    if (!hflag) {
        printf("run test<> No free PCB\n\r");
        return;
    }

    curr_seg = select_seg();
    printf("\n\rseg\n\r");
    printChar('0' + curr_seg);
    if(curr_seg == -1) {
        printf("run test<> No free segment\n\r");
```

```
        return;
    }

    curr_seg = segement_list[curr_seg];
    load_pro();

    init(&pcb_list[pcb_num], curr_seg, 0x100);
    pcb_list[pcb_num].ID = pcb_num;

    Program_Num++;
}
```

12. 增加系统调用

在 syscall.asm 文件中增加中断，将 fork，wait，exit 封装为 21H 号系统调用，供用户程序使用。由于三者实现方式相同（仅在调用具体的函数一行不同，其它相同），这里就以 fork 为例说明：在调用 pcb.h 中的 do_fork 之前，需要额外执行一次_Save_Process，由于涉及到传参，需要先压栈。这样可以时父进程的 PCB 进行一次更新，获取当前的寄存器上下文，然后执行 fork 函数，创建子进程。

```
forking:
    .386
    push ss

    .8086
    push ax
    push bx
    push cx
    push dx
    push sp
    push bp
    push si
    push di

    .386
    push ds
    push es
    push fs
    push gs
    .8086

    mov ax,cs
```

```
    mov ds, ax
    mov es, ax

    call near ptr _Save_Process
    call near ptr _do_fork
    iret
```

用户程序部分

1. 封装相关操作

在 userlib.asm 文件中，将 do_fork 等系统调用封装为 fork 等函数，供用户程序使用，同时用汇编写了一些基本的函数，用来测试输出。

```
;*************** ******************
;*  int _fork()                    *
;*************** ******************
public _fork
_fork proc
    mov al,6
    int 21h
    ret
_fork endp

;*************** ******************
;*  void _wait()                   *
;*************** ******************
;
public _wait
_wait proc
    mov al,7
    int 21h
    ret
_wait endp

;*************** ******************
;*  void _exit()                   *
;*************** ******************
;
public _exit
_exit proc
    push bp
    mov bp,sp
    push bx

    mov al,8
    mov bx,[bp+4]
    int 21h
```

```
    pop bx
    pop bp

    ret
_exit endp
```

2. 测试 fork

在 user.c 文件中，按照实验要求，编写测试代码，调用 fork，wait，exit 实现父子进程通信、同步，子进程进行字符串字母的计数，然后将结果保存在一个共享的变量中，退出，父进程在子进程推出后从阻塞态变为就绪态，然后打印出相应的结果。（额外实现了一个打印整数的函数 printint，方便测试）

```
extern int fork();
extern void wait();
extern void exit();
extern void printChar();
extern void cls();

char str[80]="129djwqhdsajd128dw9i39ie93i8494urjoiew98kdkd";
int count = 0;
int i = 0;

void printf(char *str) {
    while(*str!='\0') {
        printChar(*str);
        str++;
    }
}

void printint(int n) {
    int i = 0;
    int tmp = 0;
    int len = 0;
    char ans[100];
    char out[100];
    if (n == 0) {
        printf("0");
        return;
    }
```

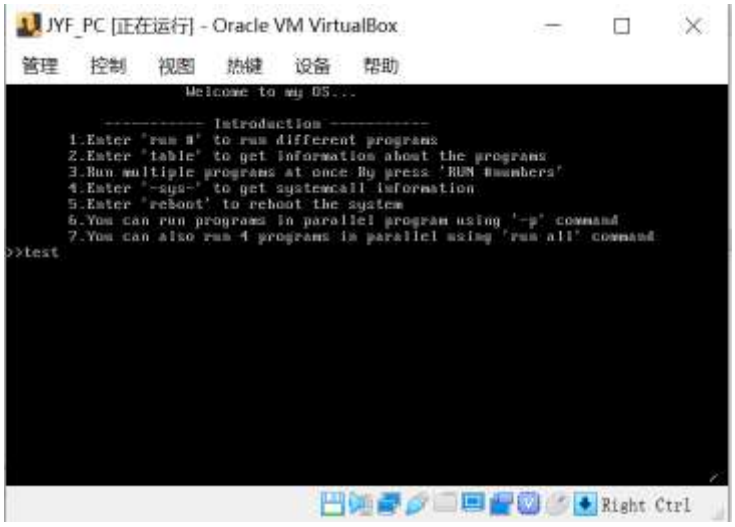
```
    while(n) {
        tmp = n % 10;
        n /= 10;
        ans[len++] = '0' + tmp;
    }
    for(i=0 ;i<len; i++) {
        out[i] = ans[len -1 - i];
    }
    out[i] = '\0';
    printf(out);
}
void main()
{
    int pid = 0;

    cls();

    pid = fork();
    if(pid > 0) {
        printf("print from father:\n\r");
        wait();
        printf("father: AFTER WAITING \n\r");
        printf("The letter number is: \n\r");
        printint(count);
        printf("\n\r");
        exit();
    }
    else if(pid == 0) {
        printf("\n\r");
        printf("print from son:\n\r");
        i = 0;
        while(str[i]!='\0') {
            if('a'<=str[i] && str[i]<='z')
                count++;
            i++;
        }
        exit();
    }
    else {
        printf("fork error\n\r");
    }
}
```

【实验过程】

在控制台输入 test 命令，执行本次实验的测试：



结果如下：

（在用户程序以及 fork 等函数中增加了输出，方便追踪函数执行过程，验证程序执行顺序的正确性）

根据输出的字符进行分析：

用户程序首先执行父进程，在 fork 后输出 print from father，然后执行 wait，被阻塞。

之后进入子进程，执行完计数操作，执行 exit 退出，唤醒父进程

父进程在子进程执行了 exit 后，状态改为就绪态，被内核调度执行，输出子进程的执行计数结果 27，然后退出，释放资源。

之后按任意键返回内核。

【实验总结】

本次实验由于没有找到老师给的参考原型，写起来花了不少时间，也踩了不少坑…首先由于需要编写一个用户进程，之前都是直接用汇编实现的展示功能的程序，而这次是汇编与 C 交叉调用的程序，这里稍微花时间复习了以下之前引导程序装载内核时的实现方式。

由于用户程序需要使用内核实现的 fork 等操作，还需在 user.asm 文件中将系统调用中断封装为函数。

另外一开始没有注意到，直接执行 do_fork 会导致子进程 PCB 不能保存到父进程最新的寄存器上下文，虽然只是个很显然的问题，但是自己 debug 的方向搞错了，浪费了不少时间。解决方案是，在调用 do_fork 之前，将当前的寄存器上下文进行一次保存。

还有一个问题就是.386 和.8086 寄存器格式的问题，ss,ds,es,fs,gs 作为参数压栈时，需要加上.386，这里之前一直都没有去注意，这次实验才关注到这个点，也算是一个学习的过程吧。

因为这次实验修改的地方特别多，涉及到内核的 pcb，增加了各种数据结构以及调度，内核的一些过程也做了相应的添加和修改，PCB 状态，调度程序的执行结果，fork 函数的执行结果都很容易出问题。同时还需要考虑内核到用户程序之间的切换等等，甚至需要在恰当的位置手动执行调度和保存寄存器的操作，所以调错变得特别麻烦，经常调整了少量代码后，整个操作系统崩溃（出栈顺序搞错等等）。而且不同模块间的关联度非常高，比如一个 SAVEPCB，各种调度程序出现问题，就会导致这个实验用户程序用到的几乎所有（内核实现）函数出问题。

调错的过程也是一个学习的过程，在这中间发现自己之前对多进程的一些理解似乎有点问题。。。同时加深刻的了解了 5 状态进程模型，以及进程间通信的具体细节，对 fork，exit 等操作有了底层方面的认识。另外还意识到如果之前肯多花一些时间学习调试工具的使用，写这个实验应该会事半功倍。。。