**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】**

1. 硬件及虚拟机配置：Lenovo PC ；Oracle Virtual Box
2. 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

1. 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。

1. 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;

1. **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;

}

1. **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;

}

1. **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

1. **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

1. **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();

}

1. **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();

}

1. **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();

}

1. **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();

}

1. **wakeup**

void wakeup() {

PCB\* curr\_pro = Current\_Process();

curr\_pro->Process\_Status = READY;

Schedule();

PCBrestart();

}

1. **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++;

}

1. **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

1. **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 {

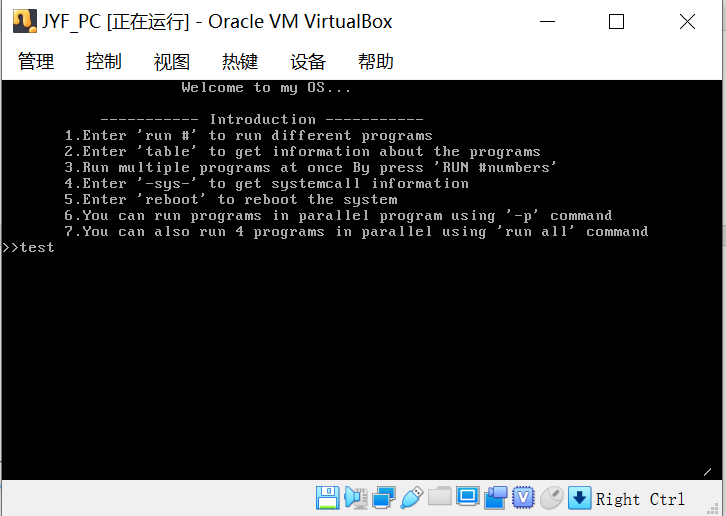
printf("fork error\n\r");

}

}

**【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。。。