

实验项目 6： 实现五状态进程

姓名：张波 学号：15352400 邮箱：1668417047@qq.com

姓名：张朝强 学号：15352401

姓名：张桂源 学号：15352404

院系：数据科学与计算机学院 专业、年级：15 级移动信息工程 指导教师： 凌应标

【实验 题目】实现二状态进程

【实验目的】

保留原型原有特征的基础上，设计满足下列要求的新原型操作系统：

- 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

硬件：操作系统为 win7 的笔记本电脑

虚拟机配置：无操作系统，10MB 硬盘，4MB 内存，启动时连接软盘

二、 软件工具及作用

Nasm:用于编译汇编程序，生成.bin 文件

VMware Workstation 12 Pro: 用于创建虚拟机，模拟裸机环境

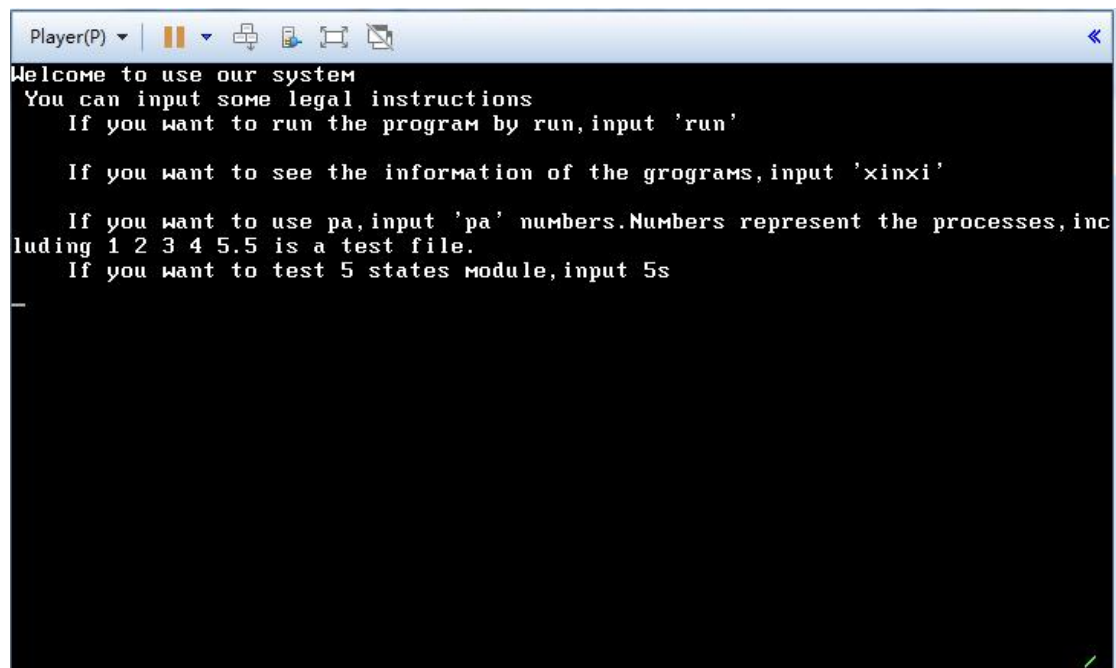
Notepad++: 用于编辑汇编语言文件

Tcc+Tasm+Tlink:用于编译生成 C 和汇编混合的程序

Dosbox:为 Tcc,Tasm,Tlink 提供 16 位运行环境

三、 程序功能

开机进入内核，内核提供 4 种指令输入，例如：



```
Player(P)
Welcome to use our system
You can input some legal instructions
If you want to run the program by run,input 'run'

If you want to see the information of the grograms,input 'xinxi'

If you want to use pa,input 'pa' numbers.Numbers represent the processes,including 1 2 3 4 5.5 is a test file.
If you want to test 5 states module,input 5s

_
```

- 1.输入 run 可以选择运行用户程序。
- 2.输入信息可以显示用户程序的信息。
- 3.在右下角利用时钟中断循环显示 ' | '、' / ' 和 ' \ '。
- 4.在运行用户程序使可以通过键盘中断输出 ' ouch '。

5.通过输入 pa 来进行分时调用用户程序，当前实现二状态转换，可以输入 1，2，3，4 来选择调用用户程序，5 代表调用 1,2,3,4 用户程序。

6.展示五状态模型。(新增)

7.命令行输入，输入错误可以删除，对空格不敏感。

四、 程序设计

Fork（）设计

分析：fork（）用于通过系统调用派生子进程。

首先修改 37h 号中断的中断向量表，即用 37h 号中断实现 fork。

```
int37h proc
    cli
    push es
    push ax
    ;es 置零
    xor ax,ax
    mov es,ax
    ;填充中断向量
    mov word ptr es:[0dch],offset do_fork
    mov ax,cs
    mov word ptr es:[0deh],ax
    pop ax
    pop es
    sti
    ret
int37h endp
```

用于实现在 c 程序中可以通过 fork 来进行中断调用

```
public _fork
_fork proc
    int 37h
    ret
_fork endp
```

中断调用时首先保存现场，然后进行子进程的派生，派生完后恢复现场。

```
do_fork proc
    cli
    push ss
    push ax
    push bx
    push cx
    push dx
    push sp
    push bp
    push si
    push di
    push ds
```

push es

.386

push fs

push gs

.8086

mov ax,cs

mov ds, ax

mov es, ax

call near ptr _Save_Process

call near ptr _c_fork

call near ptr _Current_Process

mov bp, ax

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

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

add sp,16

push word ptr ds:[bp+30]

push word ptr ds:[bp+28]

push word ptr ds:[bp+26]

push word ptr ds:[bp+2]

push word ptr ds:[bp+4]

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
    sti
    iret
do_fork endp
子进程的派生过程
void c_fork()
{
    printstring(" c_fork work\n");
    newpid=-1;           //首先要寻找看是否还有空余的 pcb 表，如果没有则将
                        pcb_list[CurrentPCBno].regImg.AX 修改为-1，用于改变 pid 的
                        值。

    for(i=0;i<6;++i)
    {
        if(pcb_list[i].used==FREE)
        {
            newpid=i;
            break;
        }
    }
    printstring(" find newpid ");
    printchar(newpid+'0');
    if(newpid==-1) pcb_list[CurrentPCBno].regImg.AX=-1;
    Else                //如果找到空余表，就将程序加
1, 并将父进程的信息复制给子进程，同时修改一些参数。
    {
        Program_Num++;
        pcb_list[CurrentPCBno].Cpid=newpid; //父进程的孩子 pid 是子进程的 pid
        pcb_list[newpid].Process_Status=READY; //子进程的状态为就绪态
        pcb_list[newpid].used=USING;          //子进程使用的 pcb 修改为在用
        pcb_list[newpid].Pid=newpid;          //子进程的 pid 为 newpid
        pcb_list[newpid].Fpid=CurrentPCBno; //子进程的指向父进程的参数为父进程 pid
        pcb_list[newpid].Cpid=-1;            //子进程的指向子进程的参数为-1
        printstring("\n copy start\n");
        copy(newpid);
        printstring(" copy done\n");
        pcb_list[CurrentPCBno].regImg.AX=CurrentPCBno; //用于 fork 的返回值
        pcb_list[newpid].regImg.AX=newpid;

    }
}
copy()
{
    pcb_list[newpid].regImg.AX = pcb_list[CurrentPCBno].regImg.AX;

```

```
pcb_list[newpid].regImg.BX = pcb_list[CurrentPCBno].regImg.BX;
pcb_list[newpid].regImg.CX = pcb_list[CurrentPCBno].regImg.CX;
pcb_list[newpid].regImg.DX = pcb_list[CurrentPCBno].regImg.DX ;
```

```
pcb_list[newpid].regImg.DS = pcb_list[CurrentPCBno].regImg.DS;
pcb_list[newpid].regImg.ES = pcb_list[CurrentPCBno].regImg.ES;
pcb_list[newpid].regImg.FS = pcb_list[CurrentPCBno].regImg.FS;
pcb_list[newpid].regImg.GS = pcb_list[CurrentPCBno].regImg.GS;
```

```
pcb_list[newpid].regImg.SS=(newpid+1)*0x1000;    //子进程的栈和父进程不同，找到 pcb 表对应的栈段
```

```
copystack(pcb_list[CurrentPCBno].regImg.SS,pcb_list[newpid].regImg.SS);// 将父进程栈中内容复制给子进程
```

```
pcb_list[newpid].regImg.IP = pcb_list[CurrentPCBno].regImg.IP ;
pcb_list[newpid].regImg.CS = pcb_list[CurrentPCBno].regImg.CS;
pcb_list[newpid].regImg.FLAGS = pcb_list[CurrentPCBno].regImg.FLAGS;
```

```
pcb_list[newpid].regImg.DI = pcb_list[CurrentPCBno].regImg.DI;
pcb_list[newpid].regImg.SI = pcb_list[CurrentPCBno].regImg.SI;
pcb_list[newpid].regImg.SP = pcb_list[CurrentPCBno].regImg.SP;
pcb_list[newpid].regImg.BP = pcb_list[CurrentPCBno].regImg.BP;
```

```
}
```

栈的复制，将两个栈段地址作为参数传入函数，利用寄存器 **ax** 为中介不断将父进程栈中内容复制到子进程，直到通过 **sp** 判断栈已经复制完全。

```
public _copystack
```

```
_copystack proc
```

```
    push ax
```

```
    push bx
```

```
    push cx
```

```
    push bp
```

```
    mov bp,sp
```

```
    mov bx,[bp+10]
```

```
    mov cx,[bp+12]
```

```
    add bp,14
```

```
continuecopy:
```

```
    mov ss,bx
```

```
    mov ax,ss:[bp]
```

```
    mov ss,cx
```

```
    mov ss:[bp],ax
```

```
    add bp,2
```

```
    cmp bp,100h
```

```

jnz continuecopy
mov ss,bx
pop bp
pop cx
pop bx
pop ax

```

```
ret
```

```
_copystack endp
```

进程轮转，将进行态的当前进程修改为就绪态，然后去找到下一个就绪态的进程，将找到的进程作为当前进程，并将其状态修改为进行态。

```
void Schedule(){
```

```

    printstring("schedule start\n");
    if(pcb_list[CurrentPCBno].Process_Status==RUNNING)
        pcb_list[CurrentPCBno].Process_Status = READY;
    printchar(CurrentPCBno+'0');
    CurrentPCBno ++;
    if( CurrentPCBno > Program_Num )
        CurrentPCBno = 0;

    while(pcb_list[CurrentPCBno].Process_Status != READY)
    {
        CurrentPCBno ++;
        if( CurrentPCBno > Program_Num )
            CurrentPCBno = 0;
    }
    printstring("schedule done\n");
    printchar(CurrentPCBno+'0');
    printchar(Program_Num+'0');
    pcb_list[CurrentPCBno].Process_Status = RUNNING;

    return;
}

```

Wait () 设计

分析：在 c 中使用该函数通过系统调用实现让一个进程等待另一个进程。

修改 39h 中断向量表。int39h proc

```

cli
push es
push ax
;es 置零
xor ax,ax
mov es,ax
;填充中断向量

```

```

        mov word ptr es:[0e4h],offset do_wait
        mov ax,cs
        mov word ptr es:[0e6h],ax
        pop ax
        pop es
        sti
        ret
int39h endp

```

将 wait 通过调用 38h 中断实现。

```

public _wait
_wait proc
    int 39h
    ret
_wait endp

```

实现进程等待。首先要保护好现场，然后通过 c_wait 修改即将等待的进程的状态为阻塞态，然后立即进行轮转，即找到下一个在就绪态的进程实现进程切换，切换后利用新进程的 pcb 表恢复新进程上次停止的现场。

```

do_wait proc
    cli
    push ss
    push ax
    push bx
    push cx
    push dx
    push sp
    push bp
    push si
    push di
    push ds
    push es
    .386
    push fs
    push gs
    .8086

    mov ax,cs
    mov ds, ax
    mov es, ax

    call near ptr _Save_Process
    call near ptr _c_wait
    call near ptr _Schedule
    call near ptr _Current_Process
    mov bp, ax

```

```

mov ss,word ptr ds:[bp+0]
mov sp,word ptr ds:[bp+16]
add sp,16
push word ptr ds:[bp+30]
push word ptr ds:[bp+28]
push word ptr ds:[bp+26]

```

```

push word ptr ds:[bp+2]
push word ptr ds:[bp+4]
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
sti
iret

```

do_wait endp

将即将等待的程序状态设置为阻塞态

```

void c_wait() {
    pcb_list[CurrentPCBno].Process_Status=BLOCK;
}

```

Exit（）设计

分析：用于实现一个进程的结束，同时修改等待他结束的进程的状态，以及可以传递一个字符的信息给等待他的进程。

修改 38h 中断的中断向量表。

int38h proc


```

cli
push es
push ax
;es 置零
xor ax,ax
mov es,ax
;填充中断向量
mov word ptr es:[0e0h],offset do_exit
mov ax,cs
mov word ptr es:[0e2h],ax
pop ax
pop es
sti
ret
int38h endp

```

实现在 c 中可以通过 `exit` 调用 38h 号中断。

```

public _exit
_exit proc
    int 38h
    ret
_exit endp

```

保护现场，结束进程，进程转换，恢复进程

```

do_exit proc
    cli
    push ss
    push ax
    push bx
    push cx
    push dx
    push sp
    push bp
    push si
    push di
    push ds
    push es
    .386
    push fs
    push gs
    .8086

    mov ax,cs
    mov ds, ax
    mov es, ax

```

```

call near ptr _Save_Process
call near ptr _c_exit
call near ptr _Schedule
call near ptr _Current_Process
mov bp, ax
mov ss, word ptr ds:[bp+0]
mov sp, word ptr ds:[bp+16]
add sp, 16
push word ptr ds:[bp+30]
push word ptr ds:[bp+28]
push word ptr ds:[bp+26]

push word ptr ds:[bp+2]
push word ptr ds:[bp+4]
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
sti
iret
do_exit endp

```

实现结束进程时的参数修改

```

void c_exit(int ch) {
    pcb_list[CurrentPCBno].Process_Status=END;    //结束进程船台修改为结束
    pcb_list[CurrentPCBno].used=FREE;             //结束进程所使用的 pcb 表转态修改
}

```

为 free

```
pcb_list[pcb_list[CurrentPCBno].Fpid].Process_Status=READY;    //等待他结束的进程
转态修改为就绪态
    pcb_list[pcb_list[CurrentPCBno].Fpid].reglmg.AX=ch;    //如果需要可以传递一个字符
给等待他的进程
}
```

用户程序设计

父进程想要计算一个字符串长度，通过派生子进程来计算长度，结束后通过父进程将计算结果输出。

```
char str[80]="ZhangBo";
int LetterNr=0;
int pid;
display5s() {
    x=0;
    y=0;
    printstring(" 5 states show:\n");
    fork();
    pid=pcb_list[CurrentPCBno].reglmg.AX;
    printstring(" fork done\n");
    if (pid!=-1)
    {
        printstring(" error in fork!\n");
        exit(-1);
    }
    if (pid==0)
    {
        printstring(" This is father process.\n");
        printstring(" I want to count the number of chars of \"ZhangBo\".\n");
        printstring(" waiting...\n");
        wait();
        printstring(" Father process continues.\n");
        printstring(" LetterNr=");
        printchar(LetterNr+'0');
        exit(0) ;
    }
    else if(pid==1)
    {
        printstring(" This is child process.\n");
        printstring(" counting...\n");
        CountLetter(str);
        printstring(" Child process ends.\n");
        exit(0);
    }
    check();
}
```

}

五. 效果展示:



```
MS-DOS (6) - VMware Workstation 12 Player (仅用于非商业用途)
Player(P)
5 states show:
c_fork work
find newpid 1
copy start
copy done
fork done
This is father process.
I want to count the number of chars of "ZhangBo".
waiting...
fork done
This is child process.
counting...
Child process ends.
Father process continues.
LetterNr=7_
```

六. 实验总结

问题:

- 1.在 c 语言中一个函数的内部定义一个变量会出现无法预知的问题。

```
Void test{
    Int pid=1;
}
```

Pid 不是 1.

然而

```
Int pid
Void test{
    pid=1;
}
```

Pid 是 1.

- 2.本来打算用一个 com 文件作为用户程序的,但是一旦有两个 com 文件后,无论怎么修改引导程序第二个 com 文件都无法运行,只能放弃,改为在内核里写用户程序。

体会:

主要是要搞清楚各个寄存器的变化过程,以及栈中的变化,一步步跟踪,就可以将问题看清楚,然后解决掉。最后感觉对栈的变化理解还是不够,本来想简化几个中断的代码,将保护现场和恢复独立出来通过 call 来调用,但是在恢复的时候 sp 始终调不对,感觉是对于 call 时栈的变化还没理解清楚还是什么原因,最后只能无奈的将他们展开写,来避免掉 call 对栈的影响。