**实验七报告**

by 12348006 蔡燕芝

2014年5月19日

**一、实验目的：**

1. 学习多进程控制方法，掌握进程控制的实现方法。

2. 完成调度程序，实现有阻塞状态的进程模型

3. 扩展MyOS，实现多进程模型的父子进程同步。

**二、实验内容：**

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：

(1)实现fork()、wait()和exit()。

(2)内核实现上面三系统调用，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。

**三、实验实现过程：**

依旧采用c和汇编交叉开发的方法

系统调用部分的程序在kliba.asm中

c程序部分在process.h中

**3.1系统调用功能表：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 中断号 | 功能号 | 传入参数 | 返回参数 | 功能 |
| fork() | 33 | Ah =10 | 无 | Ax = Pid | 创建子进程 |
| Wait() | 33 | Ah =11 | 无 | 无 | 进程等待 |
| Exit(int state) | 33 | Ah =12 | Dx = state(退出的状态) | 无 | 进程退出并唤醒父进程(若有) |

**3.2 fork()的具体实现**：

系统调用部分：

int33\_10\_1:

call \_Save

call \_fork

call \_restart

fork()程序的c实现：

/\*

1）在父进程中，fork返回新创建子进程的进程ID;

2）在子进程中，fork返回0;

3）如果出现错误，fork返回一个负值;

\*/

int fork() /\*-1表示分配子程序空间失败， -2表示分配栈空间失败 \*/

{

int stack\_id;

pid = createProcess();

if(pid)

{

/\*申请 分配栈内存空间\*/

stack\_id = requrireStackMemory();

if(stack\_id == -1)

{

/\*分配栈空间失败，释放子程序空间\*/

pcb\_list[pid].used = 0;

pcb\_list[CurrentPCBno].regImg.AX = -2;

}

else

{

pcb\_list[pid] = pcb\_list[CurrentPCBno]; /\*复制父进程的状态信息\*/

pcb\_list[pid].fpid = CurrentPCBno; /\*fpid = CurrentPCBno\*/

pcb\_list[pid].regImg.SS = SS\_baseSeg + stack\_id \* SS\_Len; /\*重新分配栈段\*/

pcb\_list[pid].regImg.SP = pcb\_list[CurrentPCBno].regImg.SP;

memcopy(pcb\_list[pid].regImg.SS,pcb\_list[CurrentPCBno].regImg.SS, SP\_OFF);

pcb\_list[pid].status = READY;

pcb\_list[CurrentPCBno].regImg.AX = pid;

pcb\_list[pid].regImg.AX = 0;

}

}

else

{

/\*表示分配子程序空间失败\*/

pcb\_list[CurrentPCBno].regImg.AX = -1;

}

}

其中memcopy的实现如下，用于复制栈空间

; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; void \_memcopy(int ss1,int ss2, int len);

; ; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 复制内存的数据

public \_memcopy

\_memcopy proc

push ax

push es

push ds

push di

push si

push cx

push bp

mov bp, sp

mov ax,[bp+16] ;ss\_son

mov es,ax

mov di, 0

mov ax, [bp+18] ;ss\_father

mov ds, ax

mov si, 0

mov cx, [bp+20] ;传输长度

cld

rep movsw; ds:si->es:di

pop bp

pop cx

pop si

pop di

pop ds

pop es

pop ax

ret

\_memcopy endp

requireStackMemory()部分实现如下，用于请求栈空间，对于栈空间的分配，我从内存的0xE000h\*16处开始分配，用数组STACK\_LIST[MAX\_NrPCB]记录该段内存的使用情况，将之分配给请求栈空间的子函数。每一个栈内存分配的空间是SS\_Len\* 16（暂时这么分配）

int SS\_baseSeg = 0xE000;

int SS\_Len = 0x100;

int STACK\_LIST[MAX\_NrPCB];/\*SS = SS\_BaseSeg + index \* SS\_LEN; SP = 100-4; 等于0代表该栈区可用\*/

/\*分配栈段，返回可用栈段编号\*/

int requrireStackMemory()

{

int i;

for(i = 0; i < MAX\_NrPCB; i++)

if(STACK\_LIST[i] == 0)

{

STACK\_LIST[i] = 1;

return i;

}

return -1;/\*分配失败\*/

}

**3.3 wait()的具体实现:**

系统调用部分：

int33\_11\_1:

call \_Save

call \_wait

call \_Schedule

call \_restart

c部分

/\*wait阻塞自己\*/

void wait()

{

myprintf("Prog ");

int2Chars(tmpMessage,CurrentPCBno);

myprintf(tmpMessage);

myprintf(" wait\r\n");

pcb\_list[CurrentPCBno].status = BLOCKED;

}

**3.4 exit(int state)的实现：**

系统调用部分：

int33\_12\_1:

mov ax, cs

mov es, ax

mov ds, ax

push dx ;当前栈顶:\psw\cs\ip\dx\call\_save ->exit()中会重新开始下一进程

call \_exit

call \_Schedule

call \_restart

c部分

/\*退出\*/

void exit(int ch)

{

myprintf("Prog ");

int2Chars(tmpMessage,CurrentPCBno);

myprintf(tmpMessage);

myprintf(" exit\r\n");

fpid = pcb\_list[CurrentPCBno].fpid;

pcb\_list[CurrentPCBno].status = EXIT;

pcb\_list[CurrentPCBno].used = 0;

if( fpid != -1)

{

pcb\_list[fpid].status = READY;

pcb\_list[fpid].regImg.AX = ch; /\*记录退出信息\*/

}

processNum--;

if(processNum == 0)

{

isUserMode = 0;

}

}

**3.5 将wait，fork，exit封装成c库，放在userlib.asm中：**

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

;\* int fork(); \*

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 33号中断号功能

public \_fork

\_fork proc

mov ah, 10

int 33

ret

\_fork endp

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

;\* int wait(); \*

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 33号中断11号功能

public \_wait

\_wait proc

mov ah, 11

int 33

ret

\_wait endp

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

;\* void exit(int n); \*

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 33号中断12号功能

public \_exit

\_exit proc

push bp

mov bp, sp

mov dx, [bp+4]

mov ah, 12

int 33

pop bp

ret

\_exit endp

**四、多进程合作的应用程序的实现**

**a.asm + a.c 🡪 a.com**

实现的功能是：创建子进程，子进程替父进程数字符串的字符个数，有错误判断

**a.asm部分的实现：**

extrn \_cmain:near

;主要功能在cmain实现

.8086

\_TEXT segment byte public 'CODE'

DGROUP group \_TEXT,\_DATA,\_BSS

assume cs:\_TEXT

org 100h

start:

mov ax, cs

mov es, ax

mov ds, ax

mov ss, ax

mov sp, 100h-4

call \_cmain

include userlib.asm

\_TEXT ends

;\*\*\*\*\*\*\*\*\*\*\*\*DATA segment\*\*\*\*\*\*\*\*\*\*\*\*\*

\_DATA segment word public 'DATA'

\_DATA ends

;\*\*\*\*\*\*\*\*\*\*\*\*\*BSS segment\*\*\*\*\*\*\*\*\*\*\*\*\*

\_BSS segment word public 'BSS'

\_BSS ends

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*end of file\*\*\*\*\*\*\*\*\*\*\*

end start

**a.c 部分的实现**：

extern void myprintf(char \*mess);

extern void exit(int a);

extern void myInt2Chars(int num, char \*pChars);

extern int fork();

extern void wait();

char str[100] = "hello i love os!! But can you show me no mistakes? T^T";

int strSize;

char tempMess[100];

int countStr(char \*mess)

{

int size = 0;

while(mess[size])

{

size++;

}

return size;

}

void cmain()

{

int pid;

myprintf("test fork!!");

/\*

1）在父进程中，fork返回新创建子进程的进程ID;

2）在子进程中，fork返回0;

3）如果出现错误，fork返回一个负值;

\*/

pid = fork();

if(pid == 0)

{

myprintf("hello!! I am the child.\r\n");

strSize = countStr(str);

/\*进程退出\*/

exit(0);

}

else if(pid == -1)

{

myprintf("create child failed！\r\n");

exit(-1);

}

else

{

myprintf("hello!! I am the father of prog ");

myInt2Chars(pid,tempMess);

myprintf(tempMess);

myprintf(".\r\n");

myprintf("ready to wait.\r\n");

/\*进程等待\*/

wait();

myprintf("wait end!\r\n");

myprintf("Result: the size of string is ");

myInt2Chars(strSize, tempMess);

myprintf(tempMess);

myprintf(".\r\n");

exit(0);

}

}

**编译连接成a.c：**



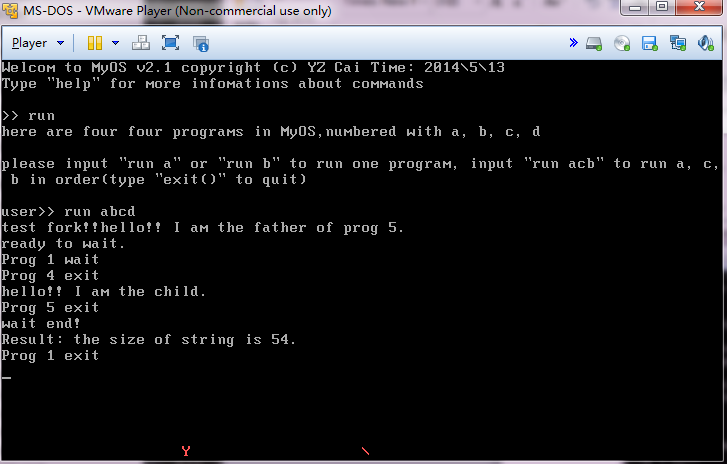
**五、效果展示**：

**5.1程序功能：**

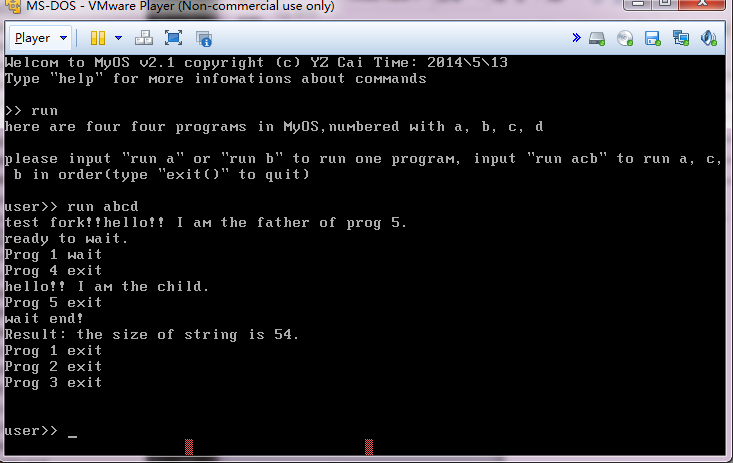
|  |  |
| --- | --- |
| 程序编号 | 功能 |
| a | 测试多进程合作，子进程替父进程数字符串的字符个数 |
| b | 在(24,20)处递增显示字符6000次后退出 |
| c | 在(24,40)处递增显示字符6000次后退出 |
| d | 直接退出 |

**5.2效果：**

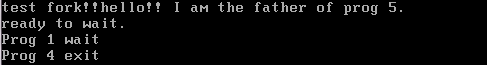
运行四个程序，执行的某个中间过程：



最后结果如下：



可以看到进程1开始执行后阻塞自己，开始等待



接着，轮转到进程4时，程序退出：



紧跟着进程a创建的子进程5执行完毕，退出：



子进程5退出激活进程1，进程1显示结果并退出：



等到进程2,和3完成6000次显示后，进程调度结束，返回到内核态：



至此，实验7大功告成！！

**六、实验体会和收获：**

这次实验，本来在实验六的基础上改改应该是不难实现的，但是一开始在fork函数实现过程中没有重新给子程序分配栈空间，并复制，导致了一些问题的发生，不过所幸后来发现了。而上一次的实验六实现是有一些bug的，导致了因用户程序的不同而导致调度过程时而正确时而错误，后来做了如下的修改：

restart中：

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

mov sp,word ptr ds:[si+2\*8]

save中：

pcb\_list[CurrentPCBno].regImg.SP = sp + 24; /\*恢复栈顶\*/

restart中：

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

mov sp,word ptr ds:[si+2\*8]

add sp, 2\*12;恢复进入时间中断前栈顶

即将恢复sp操作弄在了save中，结果就不会出现异常。这种现象让我匪夷所思。不过现在重新测试了多遍调度过程，子程序都能正常执行了。

这一次实验，清楚的了解了fork过程和进程间的合作问题，较深的理解了内存分配和进程调度。