实验六 进程控制与通信 实验报告

数据科学与计算机学院　2017 级计算机科学与技术　17341146　王程钥

1　实验题目

进程控制与通信

2　实验目的

了解进程控制与进程通信的原理
实现父子进程间的通信
了解原语的原理与实现方式
实现五状态进程模型

3　实验要求

在实验六的原型基础上，进化你的原型操作系统，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)实现进程控制的基本原语 do_fork(),do_wait(),do_exit(),blocked()和 wakeup()。

(2)在内核实现三个系统调用 fork(),wait()和 exit(),并在 c 库中封装相关的系统调用，使用户程序和内核都能使用。

(3)在原型中保证原有的系统调用服务可用。再编写 1 个或多个用户程序，展示系统原有的调用服务还能工作。

4　实验方案

4.1 实验环境

编程环境：Dosbox+TCC+TASM+Tlink,NASM
16 进制编辑器：Hex Editor
虚拟机：WMware Workstation
虚拟机环境

- 操作系统　MS-DOS
- 内存　1MB
- 硬盘　102MB
- 处理器　1 核心

编译命令

```
nasm -f bin guide.asm -o guide.com
tasm myos.asm myos.obj
tcc -mt -c –omain.obj main.c
tlink /3 /t myos.obj main.obj,test.com,
```

4.2 实现进程控制基本原语

根据实验要求，需要实现 do_fork(),do_wait(),do_exit(),blocked()和 wakeup()

这五个基本原语，实现进程的创建，等待，结束，阻塞，唤醒等功能。为了防止出现互斥的现象，需要以原语的形式实现。

4.3 实现进程控制系统调用

根据实验要求，需要实现 fork(),wait()和 exit()这三个基本的系统调用，使得内核和用户程序都能实现创建子进程，退出，等待等功能。

4.4 实现系统调用测试程序

编写一些简单的测试程序，测试上述实现的系统调用和基本原语能否正常运行。

4.5 程序存储设计与地址存放

以下为存储扇区位置。

功能	程序名	存储扇区
引导程序	guide.com	1
文件夹	file.txt	2
内核	test.com	3-36
用户程序 1	stone1.com	37
用户程序 2	Stone2.com	38
用户程序 3	stone3.com	39
用户程序 4	stone4.com	40
脚本程序 1	shell1.bat	41
脚本程序 2	shell2.bat	42
测试程序 1	tf.com	43-44
测试程序 2	Tf2.com	45-46

以下为内存地址。

功能	地址
引导程序	7c00h
内核	a100h
用户程序	cs=1000h+80h*pid(进程 id) ip=100h 栈地址：cs:0~cs:100

5 实验过程

5.1 相关声明

本次实验以实验六实现的增加了系统调用的原型操作系统为基础，增加了几个原语与系统调用，实现了创建子进程与进程间通信的功能。

为了能在用户程序中实现输出，我将<stdio.h>中的输出函数，即 putchar()函数移到了系统调用中，同时涉及到的几个全局变量的修改与获取也都装载到了系统调用中。

因为增加了不少功能，我对系统调用的代码进行了修改和重构，增加了<syspro.h>和<sysio.h>两个头文件，分别执行系统调用中的进程相关工作和输入输出相关工作。

因为 testfork 程序需要调用 C 库，内存占用较多，超过了一个扇区，因此我对实验三实现的简易版文件系统进行了一点微笑的修改，使得它能够记录更大的用户程序。

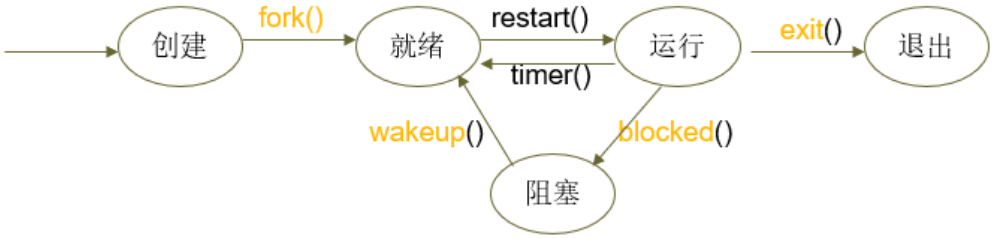
5.2 进程控制块与进程状态的修改

根据本次实验的要求，进程控制块增加了如下几项：

FID	父进程的 id
son_cnt	子进程数量

因为我已经在实验 6 中实现了 4 状态模型，对于新建态可以直接用死亡态，建立完成后改成就绪态即可。所以进程状态还是如下四个

DIE	新建/死亡	0
BLOCK	阻塞	1
READY	就绪	2
RUN	运行	3



以上为状态表和状态图。

5.3 fork()系统调用与控制原语

fork 系统调用要求内核和用户程序均可调用，用于创建子进程使用。我们实现的进程其实本质上是线程，要求父子进程共享数据段和代码段，父子进程独立的部分只有 PCB 和栈空间。fork 系统调用大体上需要按以下步骤实现：

(1) save 当前进程，关中断，并跳入内核。

(2) 为子进程找到一个进程 id，若找不到则返回-1。

(3) 将父进程的 PCB 和栈空间复制给子进程。

(4) 为父进程和子进程分别设置相应的返值，并修改 PCB 相关变量。

(5) restore 进程，开中断，返回父进程。

我依次实现了以上的 5 个步骤，并将该系统调用封装在系统调用的 07h 功能里。相关代码如下。

```
int fork()
{
  asm mov ah,07h
  asm int 21h
}
```

系统调用程序

```
fork_int:
  cli
  call near ptr Save
  mov ax,0a00h
  mov ds,ax
  mov es,ax
  mov ss,ax
```

```
  mov sp,0h
  call near ptr ___fork
  call near ptr Restore
```

Fork 系统调用的汇编部分。

我将实验六中实现的 Save 和 Resotre 进行了一些微小的修改。Call 的时候会将返回地址压入栈中，所以在 Save 的时候需要将其从栈中取出并保存。在 Restore 的时候，运行结束后直接离开中断，因此可以不用管返值，只需要在结束前加上 sti 指令开中断即可。以下为简单代码。

```
pop word ptr [call_ret]   ; stack: *\flags\cs\ip
//取返回地址
mov ax,word ptr [call_ret]
jmp ax
//返回
```

接下来是 fork 原语中 C 语言部分的代码。

```
__fork()
{
  int id,SS;
  PCB *Now=&Pro_List[Cur_Pro],*New;
  Save_Pro(Now);
  id=Find_Empty();
  if(id==-1)Now->registers.AX=-1;
  else
  {
    New=&Pro_List[id];
    SS=DataSegment+id*BASE;
    PCB_Copy(New,Now);
    STACK_Copy(SS,Now->registers.SS);
    New->FID=Cur_Pro;
    New->son_cnt=0;
    strcpy(New->name,"child");
    New->registers.AX=0;
    New->registers.SS=SS;
    Now->son_cnt++;
    Now->registers.AX=id;
    Make_Alive(id);
  }
  Restore_Pro(Now);
}
```

代码的实现基本与上面 5 个步骤的描述相同，还是比较简单易懂的，不再做过多描述。下面贴几个相关函数的实现代码。

```
int Find_Empty()
{
  int i;
```

```
  for(i=0;i<Pro_Size;i++)
    if(Pro_List[i].state==DIE)return i;
  return -1;
}
```

寻找空闲进程控制块。

```
void STACK_Copy(int Now_SS,int Pre_SS)
{
  int i;
  asm push ds
  for(i=0;i<0x100;i+=0x02)
  {
    asm mov ax,Pre_SS
    asm mov ds,ax
    asm mov bx,i
    asm mov dx,[bx]
    asm mov ax,Now_SS
    asm mov ds,ax
    asm mov bx,i
    asm mov [bx],dx
  }
  asm pop ds
}
```

复制栈。

```
void PCB_Copy(PCB *A,PCB *B)
{
  A->registers.AX=B->registers.AX;
  A->registers.BX=B->registers.BX;
  A->registers.CX=B->registers.CX;
  A->registers.DX=B->registers.DX;
  A->registers.SI=B->registers.SI;
  A->registers.DI=B->registers.DI;
  A->registers.BP=B->registers.BP;
  A->registers.FLAGS=B->registers.FLAGS;
  A->registers.CS=B->registers.CS;
  A->registers.DS=B->registers.DS;
  A->registers.ES=B->registers.ES;
  A->registers.SS=B->registers.SS;
  A->registers.IP=B->registers.IP;
  A->registers.SP=B->registers.SP;
}
```

复制进程控制块。

5.4 wait()系统调用与控制原语

wait 系统调用要求实现进程等待的功能。父进程想要等待所有子进程结束后再处理子

进程的后事，就需要 wait 系统调用来实现同步。

wait 系统调用大体上需要按以下步骤实现：

(1) save 当前进程，关中断，并跳入内核。

(2) 将当前进程阻塞。

(3) 选择一个即将执行的新的进程。

(4) restore 新进程，开中断，执行新进程。

我依次实现了以上的 4 个步骤，并将该系统调用封装在系统调用的 08h 功能里。相关代码如下。

```
int wait()
{
  asm mov ah,08h
  asm int 21h
}
```

系统调用程序

```
wait_int:
  cli
  call near ptr Save
  mov ax,0a00h
  mov ds,ax
  mov es,ax
  mov ss,ax
  mov sp,0h
  call near ptr ___wait
  call near ptr Restore
```

wait 系统调用的汇编部分。

```
__wait()
{
  Save_Pro(&Pro_List[Cur_Pro]);
  Make_Pause(Cur_Pro);
  Schedule();
  Restore_Pro(&Pro_List[Cur_Pro]);
}
```

wait 系统调用的 C 语言部分。

5.5 exit()系统调用与控制原语

exit 原语用于实现进程结束信息传递的功能。父进程在系统调用 wait 中被阻塞，在子进程终止时，调用 exit，向父进程报告这一事件，传递一个返回值，并根据父亲中子进程阻塞的数量决定是否接触父亲的阻塞。

exit 系统调用大体上需要按以下步骤实现：

(1) 跳入内核，并开中断。

(2) 杀死当前进程，使其不会再被时钟中断调用。

(3) 向父进程传递信息。

(4) 杀死所有直接父亲或间接父亲为该进程的进程。

(5) 让死亡进程自生自灭。

我依次实现了以上的 5 个步骤，并将该系统调用封装在系统调用的 09h 功能里。相关代码如下。

```
void exit(char c)
{
  asm mov ah,09h
  asm mov cl,c
  asm int 21h
}
```

系统调用程序。

```
int_21h_main:
  push ds
  push es
  push di
  push si
  push bp

  push ax
  mov ax,0a00h
  mov ds,ax
  mov es,ax
  pop ax

  mov [_Ax],ax
  mov [_Bx],bx
  mov [_Cx],cx
  mov [_Dx],dx
  call near ptr _int21h_main
  mov ax,_Ax
  mov bx,_Bx
  mov cx,_Cx
  mov dx,_Dx

  pop bp
  pop si
  pop di
  pop es
  pop ds
  mov al,20h      ; AL = EOI
  out 20h,al      ; 发送 EOI 到主 8529A
  out 0A0h,al     ; 发送 EOI 到从 8529A
```

```
  iret
```

汇编部分放在了系统调用总入口中，以上为系统调用总入口代码。

```
__exit(char RET)
{
  asm cli
  Kill_Pro(Cur_Pro,RET);
  asm sti
  asm jmp $
}
```

exit 系统调用的 C 语言部分。

```
void Kill_Pro(int id,char ret)
{
  int i,x;PCB *Now,*Fa;
  Make_Die(id);
  //杀死当前进程
  Now=&Pro_List[id];
  if(Now->FID)
  {
    Fa=&Pro_List[Now->FID];
    Fa->son_cnt--;
    if(Fa->son_cnt==0)
    {
      Fa->registers.AX=ret;
      Fa->state=READY;
    }
  }
  //返值传递
  for(i=0;i<Pro_Size;i++)
  {
    if(!Pro_List[i].state)continue;
    for(x=i;x&&x!=id;x=Pro_List[x].FID);
    if(x==id)Make_Die(i);
  }
  //杀死以当前进程为直接或间接父亲的进程
}
```

Kill_Pro 函数，杀死一个进程并将返值传递给父进程。相关代码解析见注释即可。

5.6 进程状态控制原语

我实现了几个进程状态修改原语，分别设置进程状态为就绪，阻塞，死亡等状态。相关代码如下。

```
void Make_Alive(int id){Pro_List[id].state=READY;}
void Make_Die(int id){Pro_List[id].state=DIE;}
void Make_Pause(int id){Pro_List[id].state=BLOCK;}
```

5.7 系统调用测试程序

我设计了两个系统调用测试程序。

第一个程序参考了老师的系统调用测试程序，建立父子进程，统计字符串中字母个数。相关代码如下：

```
#include"system.h"
#define BUFLEN 10
char str[80]="129djwqhdsajd128dw9i39ie93i8494urjoiew98kdkd";
int LetterNr=0;
/*IO 部分 略*/

int CountLetter(char *s)
{
  int i,ans=0;
  for(i=0;s[i]!='\000';i++)
  {
    if('a'<=s[i]&&s[i]<='z')ans++;
    if('A'<=s[i]&&s[i]<='Z')ans++;
  }
  return ans;
}

test_main()
{
  int pid=fork(),ch;
  if(pid==-1)puts("error in fork!");
  else if(pid)
  {
    puts("I'm father!!");
    printf("Child_Id= ");putint(pid);puts("");
    ch=wait();
    puts("I'm father again!!");
    printf("Ret_Val= ");putint(ch);puts("");
    printf("Father_LetterNr= ");putint(LetterNr);puts("");
  }
  else
  {
    puts("I'm child!!");
    LetterNr=CountLetter(str);
    printf("Child_LetterNr= ");putint(LetterNr);puts("");
  }
```

```
  exit(0);
}
```

第二个程序是一个两层 fork 的程序，意在测试双层 fork 的结果。相关代码如下。

```
#include"system.h"
#define BUFLEN 10
/*IO 部分 略*/

test_main()
{
  int id1,id2;id1=fork();id2=fork();
  if(id1&&id2)
  {
    puts("I'm father!!");
    printf("Son_Id= ");putint(id1);puts("");
    printf("Son_Id= ");putint(id2);puts("");
    wait();
  }
  else if(id1||id2)
  {
    puts("I'm son!!");
    printf("Son_Id= ");
    if(id1)putint(id1);
    else putint(id2);
    puts("");wait();
  }
  else
  {
    puts("I'm grandson!!");
  }
  exit(0);
}
```

5.8 遇到的问题

在修改完代码，并写完三个系统调用并装机运行后，我发现了不少问题。

首先我将系统调用进行了重构，增加了不少功能。我尝试使用用户程序执行输出功能，结果却发现不能再正确的位置输出。由于之前的系统调用都没有用到内核的全局变量，因此在进入系统调用的时候我一直没有将数据段置为内核的数据段。这样其实是错误的，也就没有办法访问内核的数据。在所有系统调用入口都加入了修改 ds 寄存器的步骤后，我解决了这个问题。

我一开始测试 fork，结果用户程序居然会连续不停地 fork 很多次，后来经过调试才发现栈段的复制和寄存器的复制有 bug。

在测试的时候，我发现有时候在 testfork 程序执行 exit 之后会死锁。我思考了并测试了很久，后来发现是在 exit 原语执行前忘了关中断。我在 exit 原语前后加上了关中断和开中断，就解决了这个问题。

在最后测试的时候，我还发现了一个很大的问题。在多个程序运行的时候，再运行 testfork 程序时会出现代码的丢失，即有些字符串显示不出来。但是如果只运行 tastfork 程序，不运行其他程序就没有这个问题。我尝试将代码段扩大，从 0x400b 扩大到 0x800b，结果就解决了这个问题。看来数据段大小还是不能乱开的。

5.9 系统调用功能总结

总结一下目前实习显得系统调用的功能。

功能号	作用
00h	带颜色带格式处理(滚屏，删除)输出
01h	不带颜色不带格式处理输出
02h	改变光标位置
03h	获取光标位置
04h	清屏
05h	屏幕上滚
06h	屏幕下滚
07h	Fork 原语
08h	Wait 原语
09h	Exit 原语
10h	随机位置显示 ouch
11h	显示一首诗(登鹤雀楼)
12h	显示一首歌(后来-刘若英)
13h	修改 POS 变量的值
14h	修改 have_run 变量的值
4ch	退出操作系统

6 操作系统使用指南

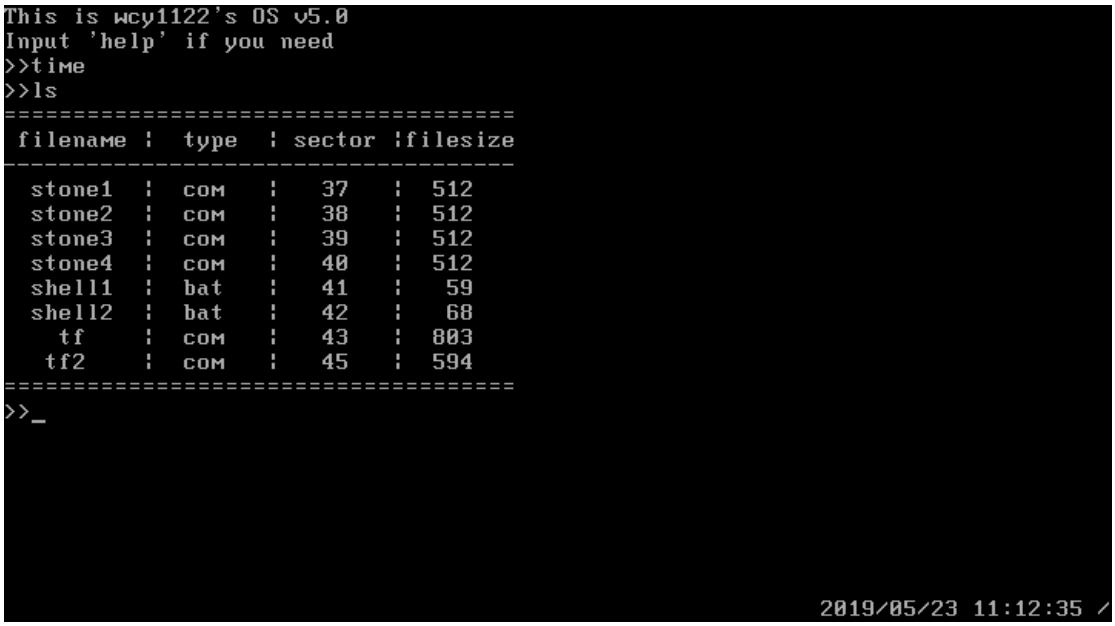
6.1 命令解释

ls	显示用户程序表
show	显示进程表
cls	界面清屏
time	显示/隐藏系统时间
system	测试系统调用
res	重启操作系统
exit	离开操作系统
run <name1>,<name2>	开始执行/恢复执行多个用户程序
stop <name1>,<name2>	结束执行多个用户程序

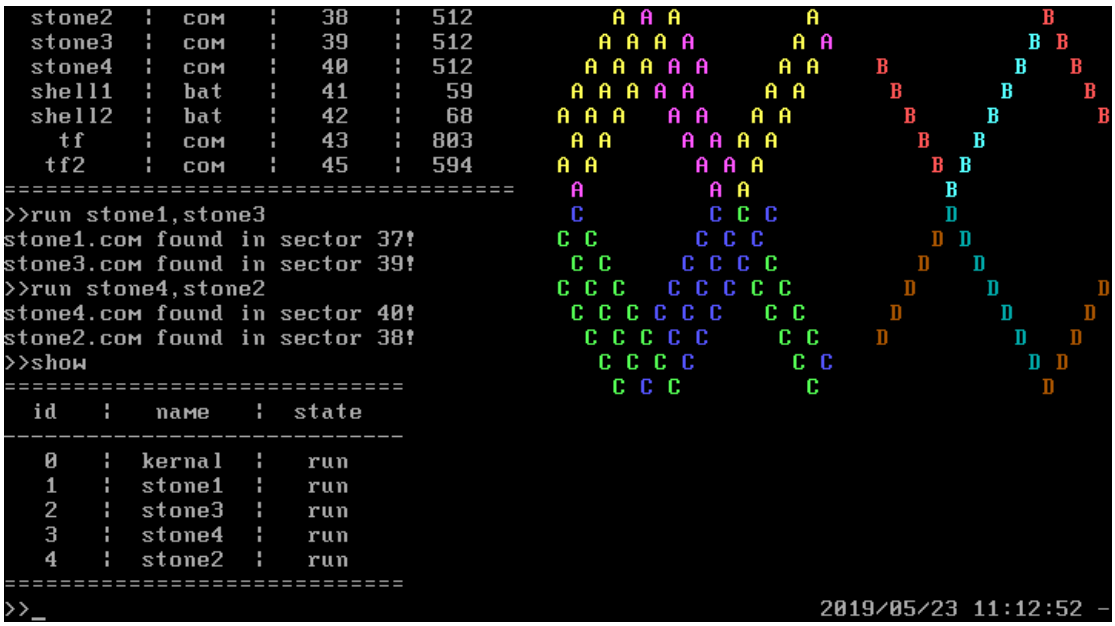
pause <name1>,<name2> shell <name1>,<name2>	暂停执行多个用户程序 执行脚本程序
上下左右与退格键都有相关功能，具体功能与 windows 中的 cmd 或 linux 中的 terminal 类似（如删除，移动光标，使用历史命令）。你可以放心地使用这些快捷键。	

6.2 实验结果

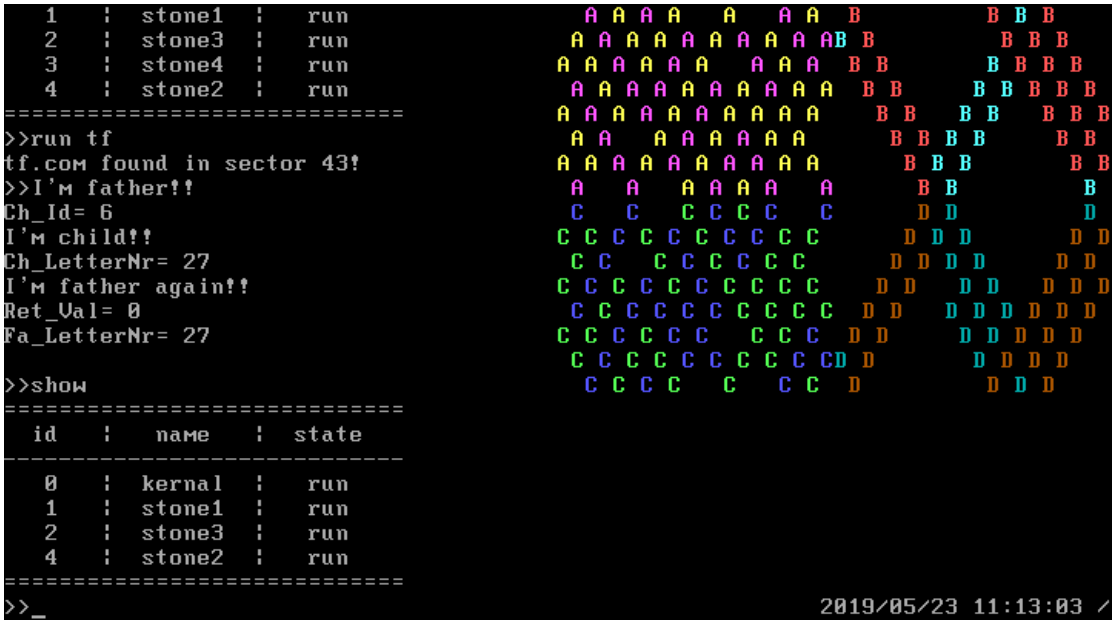
进入操作系统



显示系统时间与文件表



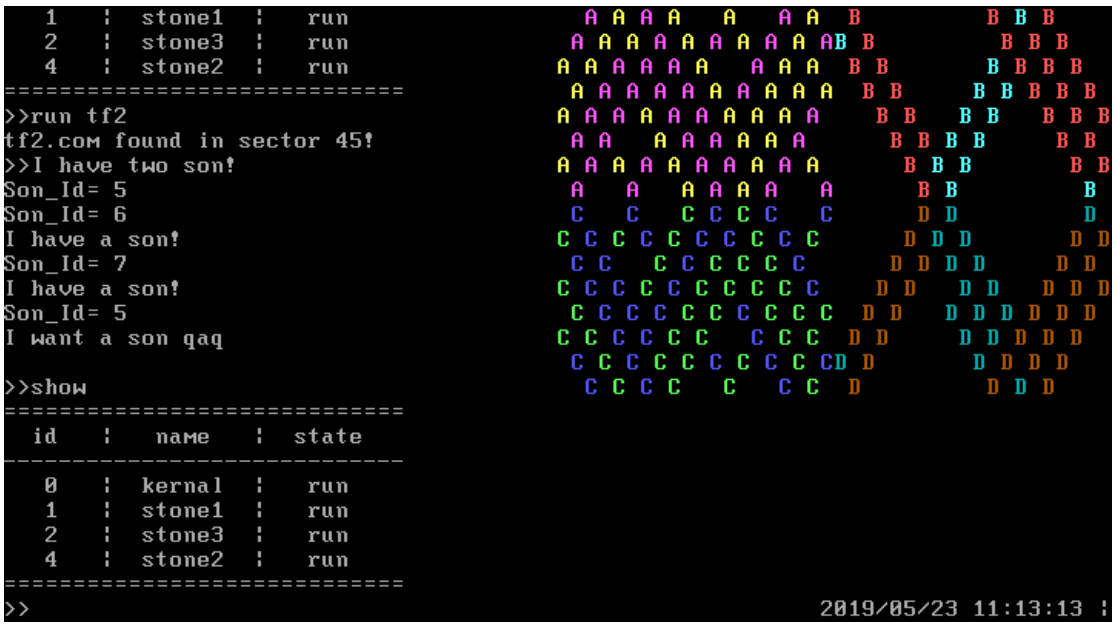
运行四个弹球用户程序并显示进程表。



运行 tf 测试程序，输出如图，成功运行。

显示进程表，stone4 已经停止运行，因此没有在进程表中出现。

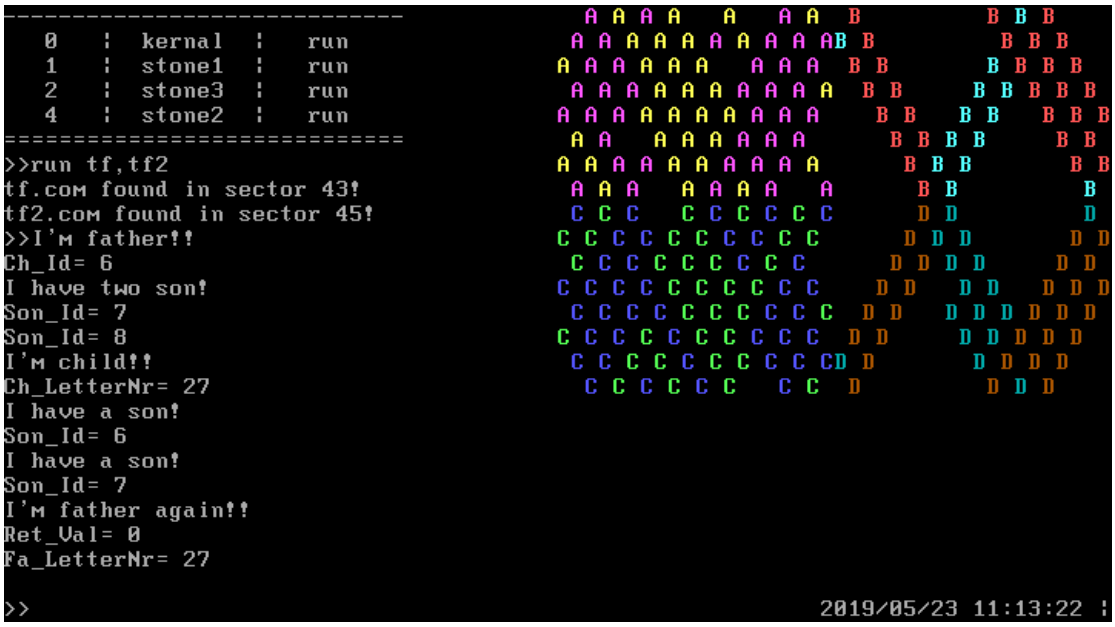
而 tf 程序运行结束，也没有在进程表中出现。



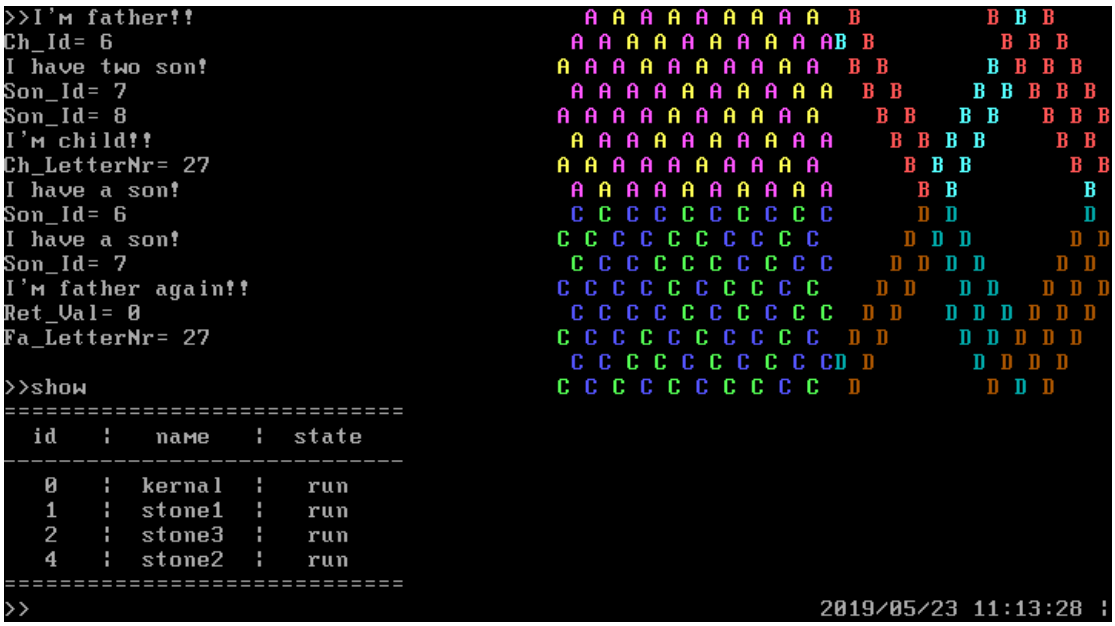
运行 tf2，并输出进程表。

如图,运行正确。第三个子进程因为继承了第一个子进程的栈段所以有 5 这个 son_id。

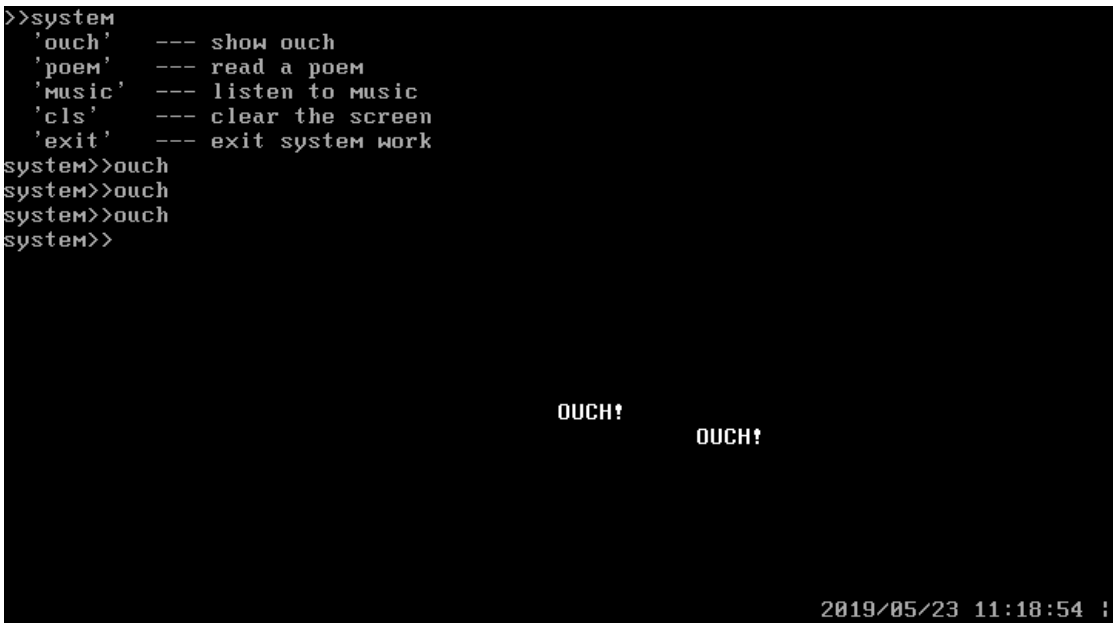
运行结束后不出现在进程表中。

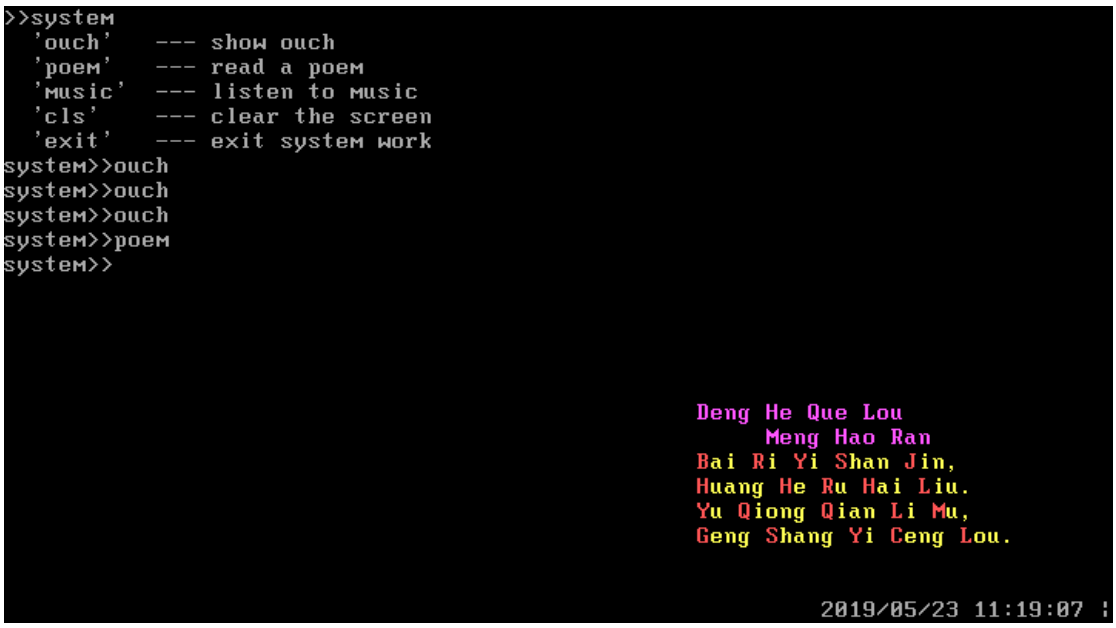


同时运行 tf 和 tf2，两个程序交替运行。如图，运行正确。


>>I'm father!!
Ch_Id= 6
I have two son!
Son_Id= 7
Son_Id= 8
I'm child!!
Ch_LetterNr= 27
I have a son!
Son_Id= 6
I have a son!
Son_Id= 7
I'm father again!!
Ret_Val= 0
Fa_LetterNr= 27
>>show
id | name | state
0 | kernal | run
1 | stone1 | run
2 | stone3 | run
4 | stone2 | run
>>
2019/05/23 11:13:28

显示进程表，结果正确。


>>system
'ouch' --- show ouch
'poem' --- read a poem
'music' --- listen to music
'cls' --- clear the screen
'exit' --- exit system work
system>>ouch
system>>ouch
system>>ouch
system>>
OUCH!
OUCH!
2019/05/23 11:18:54



运行原系统调用，仍能正常运行。

7 实验总结

本次实验要求实现 fork，wait，exit 三个原语，并编写用户程序测试相关功能。我总体上还是按照实验要求完成了相关功能，并对相关功能进行了测试。总的来说效果还是不错的。

不过本次实验其实真的花费了我很长的时间，每天都在 debug，都在寻找各种错误。就如 5.8 节中看到的，在本次实验中我遇到了各种各样的问题，有的问题甚至查了一两天也都没有看出来。通过本次实验我感受到了操作系统调试的困难和艰辛。打码一时爽，debug 火葬场，打码不规范，debug 两行泪。

同时，由于实验 5 做得不够完善，没有吧 IO 也放进系统调用，因此为了完成本次实验的要求，又增加了很多新的工作。