

操作系统原理实验

实验六

多进程和多线程模型

姓名： 吴侃

学号： 14348134

班别： 2014 级计算机系一班

日期： 2016.04.26 – 2016.04.28

目录

零、特色先览.....	3
一、实验目的.....	3
二、实验要求.....	3
三、实验环境与工具.....	4
(一) 实验环境.....	4
(二) 实验工具.....	4
四、实验方案.....	5
(1) 多进程的实现.....	5
(2) 优先级实现.....	5
(3) fork()实现.....	5
(4) 多线程实现.....	6
(5) 进程通信(使用端口通信方式).....	6
(6) 跳转表.....	6
五、 实验操作.....	7
多线程测试.....	7
进程通信(系统调用).....	9
系统调用测试.....	10
六、小结.....	11

零、特色先览

我在第三个实验的时候已经实现了多进程模型(在内存允许下, 可以创建无限多个进程, 这里限制为 16 个), 在第五个实验中实现了 fork 函数, 进程通信. 在之前的实验中, 实现了初始态, 就绪态, 执行态, 等待态, 终止态(这里还没有准确的初始态实现, 因为新建进程时, 在创建 PCB 项同时已经分配好资源), 并且能够唤醒, 挂起, 杀死单一进程. 这里重点介绍多线程模型的实现.

本次实验的亮点包括:

1. 多线程模型的实现: thread_create, thread_join
2. 进程优先级的实现
3. 正确的 fork()函数(更正返回值了)
4. 进程通信(在上一个实验中已实现, 这里再提一下)
5. 中断分支使用跳转表

一、实验目的

实验多进程与多线程模型

二、实验要求

1. 实现多进程模型
2. 实现多线程模型
3. 测试系统调用还可用

三、实验环境与工具

(一) 实验环境

物理机操作系统: Arch Linux 4.5.1-1

调试使用虚拟机: qemu-system-i386, bochs

虚拟机软件: VMware Workstation 12 Pro

虚拟机配置: CPU: i7-4702MQ @ 2.20GHz, 使用单核单线程

内存: 4 MB

硬盘: 32 MB

(二) 实验工具

编辑器: Vim 7.4

汇编工具: NASM 2.11.08

C++编译器: g++ 5.3.0

链接工具: GNU ld 2.26.0.20160302

构建工具: GNU Make 4.1

调试工具: Bochs x86 Emulator 2.6.8

虚拟机: qemu-system-i386

VMWare Workstation 12 Pro

合并文件: dd

四、实验方案

(1) 多进程的实现

多进程实现的主要原理为，建立一个 PCB 表，表中记录着每个进程的状态值和被切换时保存的寄存器值。

当发生时钟中断时，将当前进程的状态与寄存器保存到 PCB 表，进行进程调度，将新进程的状态和寄存器从 PCB 表中读出，然后执行 `iret` 切换到新的进程。

(2) 优先级实现

在 PCB 中添加 `priority` 这一项，当进行进程调度时，通过计数器计数，当计数器的值大于优先级值时则切换下一个进程。

(3) `fork()` 实现

这里引用实验五的描述: (蓝色字为更正)

`fork()` 的原理是将父进程所占的内存段复制到新的空间，并复制父进程的 PCB 表。需要修改的信息为 PCB 表中，段寄存器的值。这个过程中，我觉得最困难的是 IP 寄存器的值的处理。我的处理方式是：在 `fork()` 函数刚开始执行时，调用 8 号中断(时钟中断)，保存当前父进程的寄存器值 (使对应 PCB 为最新)，此时 PCB 中的 IP 值指向调用 8 号中断的下一条指令。接下来重新取最新的 PCB 值，进入一条判断语句。经过判断 PCB 中的值，可以区分出父进程和子进程，若是父进程，创建新进程，再返回子进程 `pid`；若是子进程，返回 0。

(4) 多线程实现

我采用类似 `fork()` 的实现来实现多线程，即将线程信息也放在 PCB 表中。与 `fork()` 类似，不同点是只拷贝进程的栈空间，更改线程的 SS 值。而 CS, DS 则不变，即采用原进程的 CS, DS。

这样可以实现全局变量共享。

还需要修改进程调度程序，对于线程进行特别的处理：

1. 线程状态为终止态时，不像进程一样马上回收资源
- 2 当进程结束时，回收其创建的线程的资源。

多线程的创建函数: `thread_create`

等待结束函数: `thread_join` （判断该线程状态是否为终止态）

(5) 进程通信(使用端口通信方式)

引用实现五的描述：

我的实现方式为由一个内存段管理端口到一个变量的段地址和偏移地址，并且有对应端口属性值（如信号量，数据大小）。这里使用了 3 号端口记录内存段分配值，5 号端口为 IO 端口。当从 5 号端口写入字符串时，内核将会检测到该信息，并将字符串输出，并将内容清空。

(6) 跳转表

使用一个连续的内存段存取要跳转的位置，使用时计算跳转地址存储位置，得到跳转地址，跳转。需要注意的是，汇编中若把跳转表放在代码区域，会被当作代码运行。时间复杂度 $O(1)$ 。

五、实验操作

多线程测试:

(多线程矩阵乘法, 结果的每一项对应一个线程)

```
MiraiOS 0.3 #462
You can input 'help' to get more info
wkc n >
wkc n > mat
      1 4
      2 5
      3 6
      X
      8 7 6
      5 4 3
The result is:
      28 23 18
      41 34 27
      54 45 36

wkc n > _
```

检验结果: (使用 octave 验证)

```
>> a = [1 4;2 5;3 6]
a =

     1     4
     2     5
     3     6

>> b = [8 7 6;5 4 3]
b =

     8     7     6
     5     4     3

>> a * b
ans =

    28    23    18
    41    34    27
    54    45    36

>> |
```

结果正确!

进程优先级测试:

```
wkcn > top
There are 8 Progresses :-)
```

PID	Name	PR	Size	SEG	CS	IP	Parent	State
0	Mirai-Shell	0	0	0x0000	0xEA60	0x25B2	0	Running
1	WRCN1.COM	0	512	0x4137	0x4137	0x019B	0	Suspend
2	WRCN2.COM	0	512	0x4167	0x4167	0x017D	0	Suspend
3	WRCN3.COM	0	512	0x4197	0x4197	0x01C2	0	Suspend
4	WRCN4.COM	0	512	0x41C7	0x41C7	0x01B0	0	Suspend
5	BOX.COM	0	327	0x41F7	0x41F7	0x023C	0	Suspend
6	BOX.COM	0	327	0x421C	0x421C	0x023C	0	Suspend
7	BOX.COM	0	327	0x4241	0x4241	0x023C	0	Suspend

```
wkcn > pr 5 3
wkcn > pr 6 10
wkcn > top
There are 8 Progresses :-)
```

PID	Name	PR	Size	SEG	CS	IP	Parent	State
0	Mirai-Shell	0	0	0x0000	0xEA60	0x25B2	0	Running
1	WRCN1.COM	0	512	0x4137	0x4137	0x019B	0	Suspend
2	WRCN2.COM	0	512	0x4167	0x4167	0x017D	0	Suspend
3	WRCN3.COM	0	512	0x4197	0x4197	0x01C2	0	Suspend
4	WRCN4.COM	0	512	0x41C7	0x41C7	0x01B0	0	Suspend
5	BOX.COM	3	327	0x41F7	0x41F7	0x023C	0	Suspend
6	BOX.COM	10	327	0x421C	0x421C	0x023C	0	Suspend
7	BOX.COM	0	327	0x4241	0x4241	0x023C	0	Suspend

```
wkcn > _
```

可以看见新的 top: (增加了优先级, 存储栈段, 父进程的信息)

可以看见 pid 为 5,6 的进程被修改了优先级, 使用 pr 进程 id 优先级值

输入 wake 5 6 7 , 或者输入 r

可以看到三个进程的运行速度不一样, 运行系统观看效果更佳:-)



进程通信: (系统调用)

porttest 代码:

```

1 #include "include/port.h"
2
3 char GoodJob[32] = "GoodJob!";
4 int main(){
5     WritePort(5, GoodJob, 32);
6     SetPortMsgV(5, 1);
7     return 0;
8 }

```

使用 0 号端口传输字符串给 Shell, Shell 检测出后输出该字符串

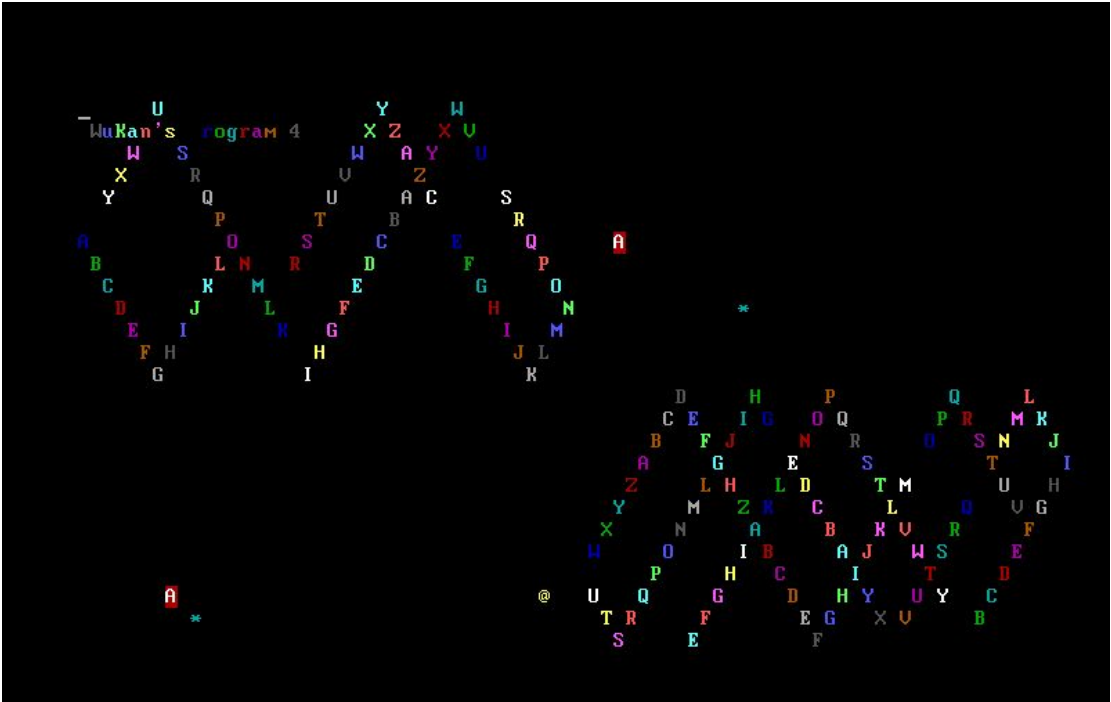
```

7 DOX.COM 8 327
wken > porttest
wken > GoodJob!
wken > _

```

多进程测试:

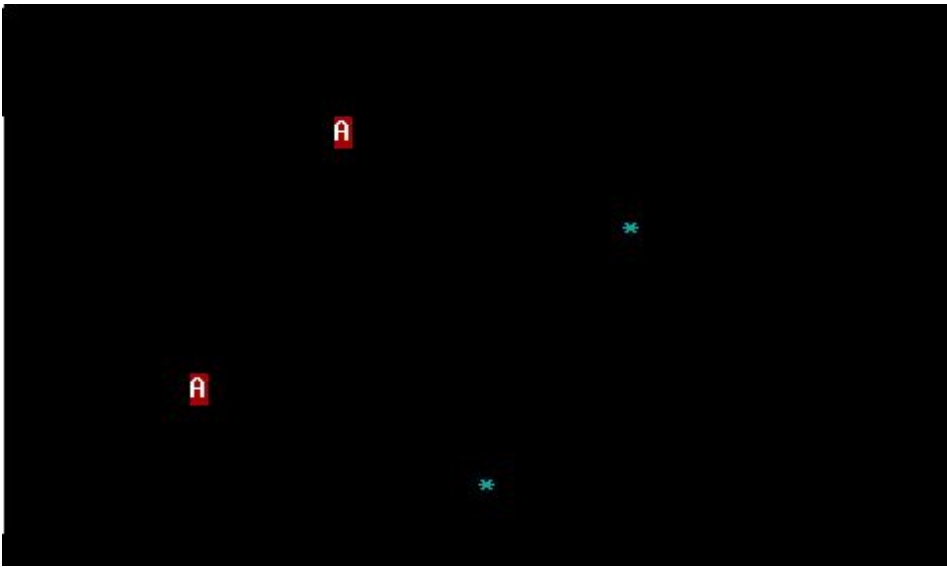
四个不同的程序的进程:



同一个程序运行了两个进程

这个程序只有一个红 A 和绿*

由于为两个进程, 可以看到共四个字符.



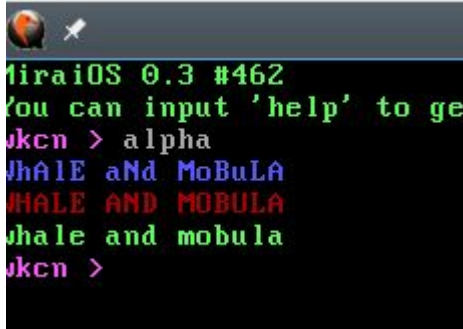
系统调用测试:

由于我的系统有很多系统调用功能, 这里不能一一列出:

比如在 `fork()`, `thread_create()` 函数中使用了系统调用

porttest 测试了 0 号端口屏幕 IO 通信(int 22h)

alpha 程序实现大小写转换(int 24h)和 `fork()`



```
miraiOS 0.3 #462
You can input 'help' to ge
jken > alpha
jHALE aNd MoBuLA
jHALE AND MOBULA
jHale and mobula
jken >
```

输入 int 33h, int 34h, int 35h, int 36h, 用中断执行用户程序

六、小结

这次实现的内容为多进程实现，我在第三个实现中已经实现分时系统，并且在内存允许下支持无限个进程，同一个程序可以有多个进程。

实验五中，我实现了 `fork()`，进程通信(使用端口通信机制)。

因此，这次实验，我的系统的扩展之处在于更正了上一个实现版本 1 中 `fork()` 的返回值，我也实现了多线程模型，进程优先级。

有趣的是，我在实现多线程模型时，在想进程与线程的区别，最终意识到线程的创建与 `fork()` 的实现很相似，只是新建了一个用户栈段，与原进程共用数据段与代码段。后来我查找了相关资料，发现早期的 linux 多线程实现，也是把线程看作一个进程实现的。

这次添加多线程和进程优先级花了两三天的时间，感觉还比较顺利。

我想这都是之前积累的结果，我觉得很有收获，希望做得更好.