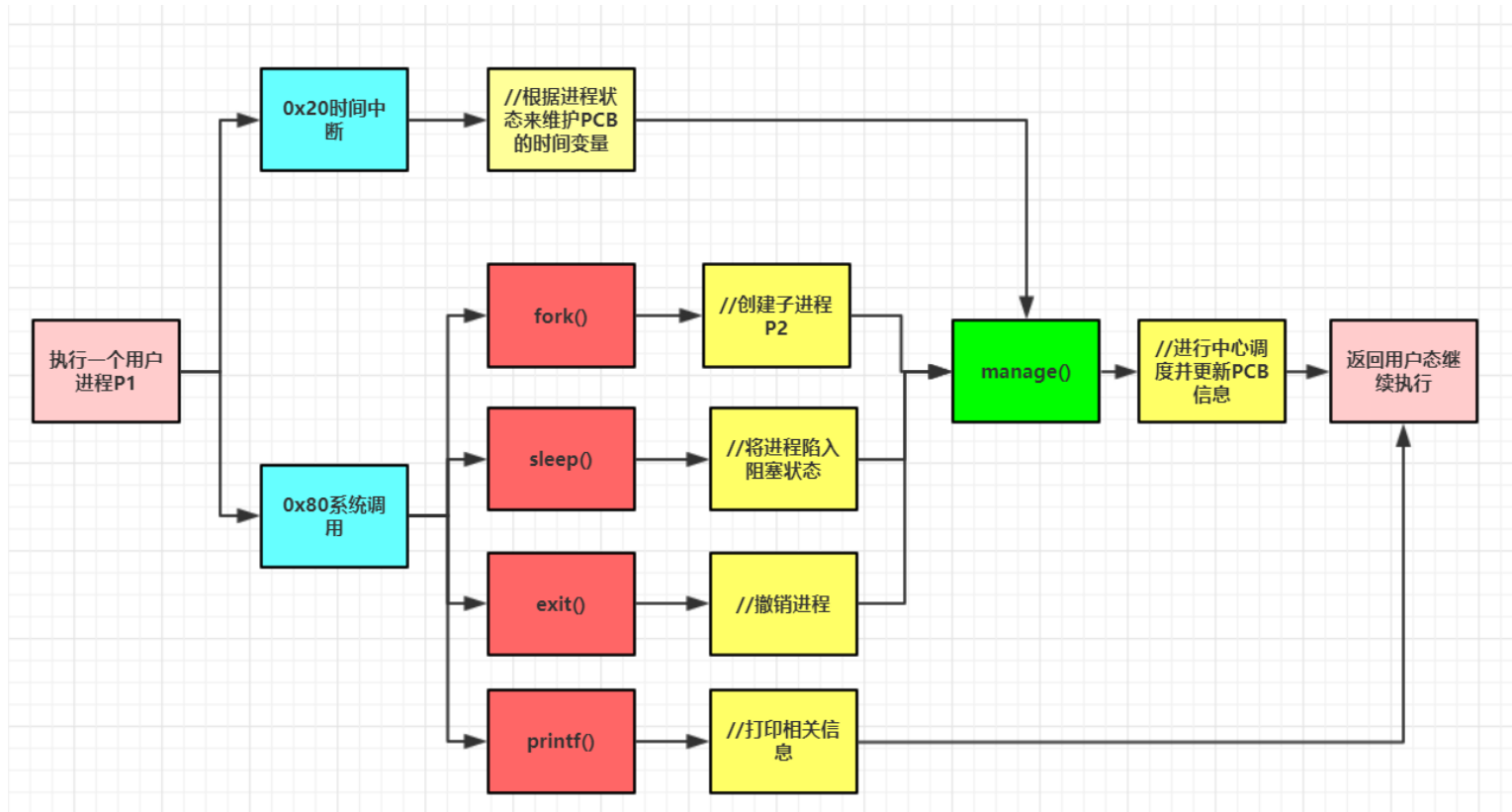


# lab3 实验报告

161220124 王明

## 1.1 代码流程与实验结果

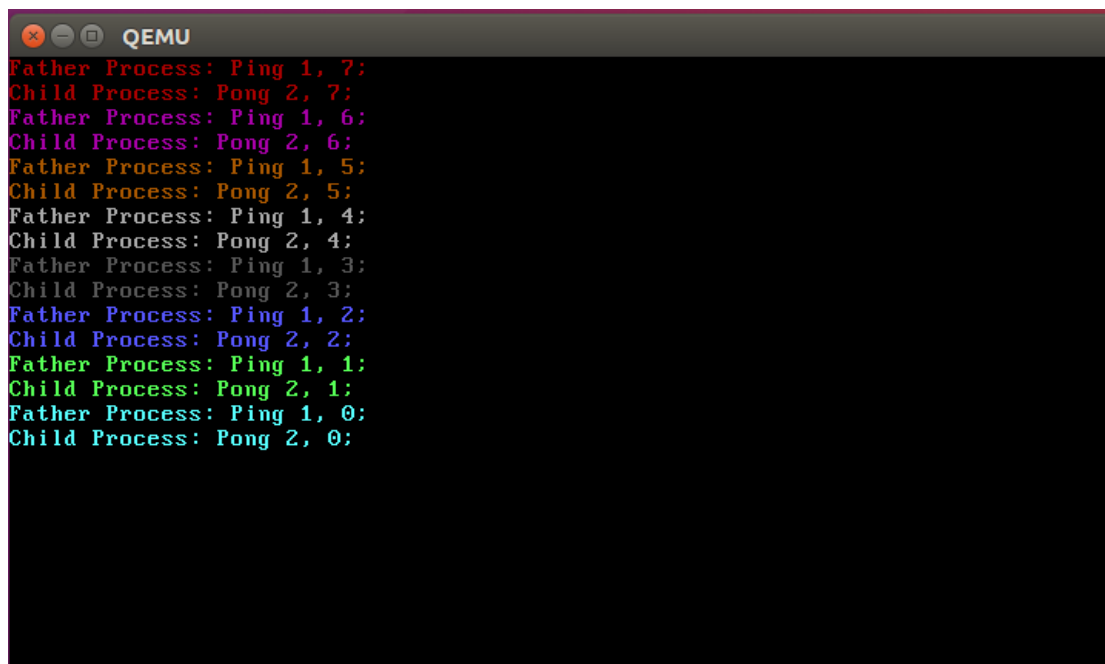
代码流程图解：



运行结果如下：

```
QEMU
Father Process: Ping 1, 7;
Child Process: Pong 2, 7;
Father Process: Ping 1, 6;
Child Process: Pong 2, 6;
Father Process: Ping 1, 5;
Child Process: Pong 2, 5;
Father Process: Ping 1, 4;
Child Process: Pong 2, 4;
Father Process: Ping 1, 3;
Child Process: Pong 2, 3;
Father Process: Ping 1, 2;
Child Process: Pong 2, 2;
Father Process: Ping 1, 1;
Child Process: Pong 2, 1;
Father Process: Ping 1, 0;
Child Process: Pong 2, 0;
```

再来个骚一点的结果：



```
QEMU
Father Process: Ping 1, 7:
Child Process: Pong 2, 7:
Father Process: Ping 1, 6:
Child Process: Pong 2, 6:
Father Process: Ping 1, 5:
Child Process: Pong 2, 5:
Father Process: Ping 1, 4:
Child Process: Pong 2, 4:
Father Process: Ping 1, 3:
Child Process: Pong 2, 3:
Father Process: Ping 1, 2:
Child Process: Pong 2, 2:
Father Process: Ping 1, 1:
Child Process: Pong 2, 1:
Father Process: Ping 1, 0:
Child Process: Pong 2, 0:
```

## 1.2.实验流程

### PCB 队列的设计：

事实上这个实验只需要三个进程，一个父进程、一个子进程、一个空闲进程。但是为了模拟得更加贴近实际情形，要设计不同的队列：运行队列、空闲队列、就绪队列、阻塞队列、死亡队列。我的设计是直接用一个 PCB 数组表示进程，然后根据运行队列和空闲队列可能有一个，因此不需要寻找，其他三种队列采用记录队列队首进程下标的方式来维护队列。再设一个全局变量 now 来指向当前进程。

```
23 extern int s_runnable;
24 extern int s_block;
25 extern int s_dead;

44 extern struct ProcessTable pcb[MAX_PCB_NUM];
45 extern struct ProcessTable *now;
```

用这三个变量来指示每个队列的队首进程的下标，若队列为空，则令它的值为-1。对于初始化 pcb 表，将状态全改成 dead。然后 pcb[0]留给 IDLE 进程，pcb[1]留给初始的用户进程。将相应的信息赋给 PCB 中的参数就完成了初始化。

### 启动时钟源：

这个直接采用实验给的代码，设置 100HZ 的时钟定时产生时钟中断，代码如下：

```
1 #include "x86.h"
2
3 #define TIMER_PORT 0x40
4 #define FREQ_8253 1193182
5 #define HZ 100
6
7 void initTimer() {
8     int counter = FREQ_8253 / HZ;
9     outByte(TIMER_PORT + 3, 0x34);
10    outByte(TIMER_PORT + 0, counter % 256);
11    outByte(TIMER_PORT + 0, counter / 256);
12 }
```

### 对于时间中断的处理：

如果当前进程不为空，那么将当前进程的时间片减 1，并将所有阻塞状态的进程的睡眠时间减 1，代码如下：

```
100 void TimeHandle(struct TrapFrame *tf)
101 {
102     if(now != NULL)
103         now->timeCount--;
104     int record = s_block;
105     while(record != -1)
106     {
107         if(pcb[record].state == STATE_BLOCK)
108         {
109             pcb[record].sleepTime--;
110         }
111         record = (record + 1) % MAX_PCB_NUM;
112         if(record == s_block)
113             break;
114     }
115     manage();
116 }
```

### 实现 fork()：

首先通过寻找当前处于 dead 的进程，然后将它的状态设置为 runnable，采用直接复制的方法将父进程的 PCB 和用户和内核栈信息全部复制到子进程相应的空间，返回值是进程的 pid，代码如下：

```
103 int fork()
104 {
105     int index = index_dead();
106     struct ProcessTable *pt = &pcb[index];
107     pt->state = STATE_RUNNABLE;
108     pt->pid = index;
109     succeed_father_regs(pt);
110     uint32_t father = 0x200000 + (now->pid - 1) * PROCESS_SPACE_SIZE;
111     uint32_t child = 0x200000 + (pt->pid - 1) * PROCESS_SPACE_SIZE;
112
113     int i;
114     for(i = 0; i < USER_SPACE_SIZE; i++)
115     {
116         *(char *)(child + i) = *(char *)(father + i);
117     }
118     update_s();
119     return index;
120 }
```

我这里每一个进程都分配了 0x200000 大小的空间。

### 实现 sleep()：

这个就是把参数 time 赋给当前进程的睡眠时间并把状态设置为阻塞状态，代码如下：

```
182 void sleep(uint32_t time)
183 {
184     now->sleepTime = time;
185     now->state = STATE_BLOCK;
186     update_s();
187 }
```

### 实现 exit() :

将当前进程的状态设为 dead 并且令 now 等于 NULL 来撤销进程，代码如下：

```
189 void exit()
190 {
191     now->state = STATE_DEAD;
192     now = NULL;
193     update_s();
194 }
```

### 实现中心调度 manage() :

这个函数要做的是根据当前进程经上面函数修改后的 PCB 信息来决定是继续执行当前进程还是调度另一个进程来执行（进程切换）。

```
if(s_runnable == -1)
{
    if(now->state == STATE_IDLE)
    {
        now->timeCount = 1;
        update_s();
        return;
    }
    now = &pcb[0];
    now->state = STATE_IDLE;
    now->timeCount = 1;
    tss.esp0 = (uint32_t)&(now->state);
    update_s();
    return;
}
now = &pcb[s_runnable];
now->timeCount = RUN_TIME;
now->state = STATE_RUNNING;
tss.esp0 = (uint32_t)&(now->state);
update_s();
if(now->state != STATE_IDLE)
{
    uint32_t offset = (now->pid - 1) * PROCESS_SPACE_SIZE;
    gdt[SEG_UCODE] = SEG(STA_X | STA_R, offset, 0xffffffff, DPL_USER);
    gdt[SEG_UDATA] = SEG(STA_W, offset, 0xffffffff, DPL_USER);
    setGdt(gdt, sizeof(gdt));
}
```

如果没有就绪的进程，就运行 pcb[0]这个 IDLE 进程。否则就运行就绪队列队首的进程，调整 TSS 和段的相关信息，将 TSS 设置到需要运行进程的 PCB 的 TF 的首地址，以便从内核态返回用户态时弹出需要运行进程的信息。

### 进程切换：

主要是基于 TSS 的功能来实现。细节就是比较返回的 esp 是否是原来进程的 esp，如果不是则需要进程切换，那么必须更新 esp 而完成进程的切换，代码如下：

```

193 .global asmDoIrq
194 asmDoIrq:
195     cli
196     pushal // push process state into kernel stack
197     pushl %esp
198     movw $16, %ax
199     movw %ax, %ds
200     call irqHandle
201     add $36, %esp
202     cmpl %eax, %esp
203     je FLAG
204     pushl %eax
205     popl %esp
206 FLAG:
207     add $4, %esp
208     add $4, %esp
209     popl %gs
210     popl %fs
211     popl %es
212     popl %ds
213     popal
214     sti
215     iret

```

### 1.3. 实验感想

这次实验真的花了大量的时间在理解代码框架和程序执行流程上，而且由于上次对于 TSS 的作用没有理解透彻，导致了这次做得很艰难。还是要多学多学多学多学多学多学多学多学多学多学。