

# OS lab3实验报告

姓名	学号	邮箱	院系
张洋彬	191220169	<a href="mailto:1016466918@qq.com">1016466918@qq.com</a>	计算机科学与技术系

## OS lab3实验报告

- 一、实验目的
- 二、实验预备知识
- 三、实验过程
  - 3.1 完成库函数
  - 3.2 时间中断处理
  - 3.3 系统调用例程
    - 3.3.1 syscallFork
    - 3.3.2 syscallSleep
    - 3.3.3 syscallExit
  - 3.4 中断嵌套
- 四、实验结果
- 五、感想与体会

## 一、实验目的

- 实现时间中断处理函数 `timerHandle(struct StackFrame*sf)`
- 实现 `fork`、`exit`、`sleep` 库函数即处理例程

## 二、实验预备知识

## 1、进程控制块的结构

```
struct ProcessTable {
    uint32_t stack[MAX_STACK_SIZE];    // 内核堆栈
    struct TrapFrame regs;              // 陷阱帧，保存上下文
    uint32_t stackTop;                  // 保存内核栈顶信息
    uint32_t prevStackTop;              // 中断嵌套时保存待恢复的栈顶信息
    int state;                          // 进程状态：STATE_RUNNABLE、STATE_RUNNING、STATE_BLOCKED、
    STATE_DEAD
    int timeCount;                      // 当前进程占用的时间片
    int sleepTime;                      // 当前进程需要阻塞的时间片
    uint32_t pid;                       // 进程的唯一标识
    char name[32];                      // not used
};
```

其中 TrapFrame 相对实验2也稍微有点不同

```
struct TrapFrame {
    uint32_t gs, fs, es, ds;
    uint32_t edi, esi, ebp, xxx, ebx, edx, ecx, eax;
    uint32_t irq, error;
    uint32_t eip, cs, eflags, esp, ss;
};
```

## 2、各个进程的段基址

+-----+ 0xffffffff
...
+-----+ 0x00400000
app 1 stack
+-----+
app 1
+-----+ 0x00300000
app 0 stack
+-----+
app 0
+-----+ 0x00200000
kernel stack
+-----+
kernel
+-----+ 0x00100000
...
+-----+ 0x00000000

在实验中，采用线性表的方式组织pcb，也就是将pcb以数组形式连续存放，为了简单起见，可以将pcb的pid设为其索引。内核进程会占据0号pcb，剩下的分配给用户进程。同样为了简单，我们默认每个pcb对应进程的内存空间固定，pcb[i] 对应的内存起始地址为  $(i + 1) * 0x100000$ ，大小为  $0x100000$

```
struct ProcessTable pcb[MAX_PCB_NUM];
```

## 三、实验过程

### 3.1 完成库函数

直接系统调用即可，如代码框所示。

```
1  pid_t fork() {
2      return syscall(SYS_FORK, 0, 0, 0, 0, 0);
3  }
4
5
6  int sleep(uint32_t time) {
7      return syscall(SYS_SLEEP, time, 0, 0, 0, 0);
8  }
9
10 int exit() {
11     return syscall(SYS_EXIT, 0, 0, 0, 0, 0);
12 }
```

### 3.2 时间中断处理

1. 遍历pcb，将状态为 `STATE_BLOCKED` 的进程的sleepTime减一，如果进程的sleepTime变为0，重新设为 `STATE_RUNNABLE`
  2. 将当前进程的timeCount加一，如果时间片用完（`timeCount >= MAX_TIME_COUNT`）且有其它状态为 `STATE_RUNNABLE` 的进程，切换;否则切换到内核IDLE进程，等待中断的到来
- 伪代码实现如下：

```
1  for(进程状态为blocked的进程块) do:
2      sleepTime--;
3      if(sleepTime <= 0)
4          把状态改为runnable
5
6  将当前正在运行的进程的timecount++;
7  如果时间片用完就将当前进程的状态置为runnable
8
9  if(pcb[current].state != STATE_RUNNING){
10     int i = (current + 1) % MAX_PCB_NUM;
11     for( ; i != current ; i = (i + 1) % MAX_PCB_NUM){
12         if(pcb[i].state == STATE_RUNNABLE){
13             current = i;
```

```

14     pcb[current].timeCount=0;
15     pcb[current].state=STATE_RUNNING;
16     break;
17 }
18 }
19 uint32_t tmpStackTop = pcb[current].stackTop;
20 pcb[current].stackTop = pcb[current].prevStackTop;
21 tss.esp0 = (uint32_t)&(pcb[current].stackTop);
22 asm volatile("movl %0, %%esp"::"m"(tmpStackTop)); // switch kernel stack
23 asm volatile("popl %gs");
24 asm volatile("popl %fs");
25 asm volatile("popl %es");
26 asm volatile("popl %ds");
27 asm volatile("popal");
28 asm volatile("addl $8, %esp");
29 asm volatile("iret");
30 }

```

最后一步，在irqhandle里增加保存与恢复的内容，实现进程的切换。

## 3.3 系统调用例程

### 3.3.1 syscallFork

- 找到进程状态为 `STATE_DEAD` 的进程，这个位置用于创建子进程
- 把父进程的数据段和代码段复制给子进程
- 复制父进程的pcb给子进程
- 修改子进程中pcb的值
- 父进程返回子进程的pid，子进程返回0

```

1  for (int i = 0; i < 0x100000; i++) {
2      *(uint8_t *) (i + (index + 1) * 0x100000) = *(uint8_t *) (i + (current + 1) *
3      0x100000);
4  }

```

### 3.3.2 syscallSleep

将当前的进程的sleepTime设置为传入的参数，将当前进程的状态设置为STATE\_BLOCKED，然后模拟时钟中断。

```

1  void syscallSleep(struct StackFrame *sf) {
2      pcb[current].sleepTime = sf->ecx;
3      pcb[current].state = STATE_BLOCKED;
4      asm volatile("int $0x20");
5  }

```

### 3.3.3 syscallExit

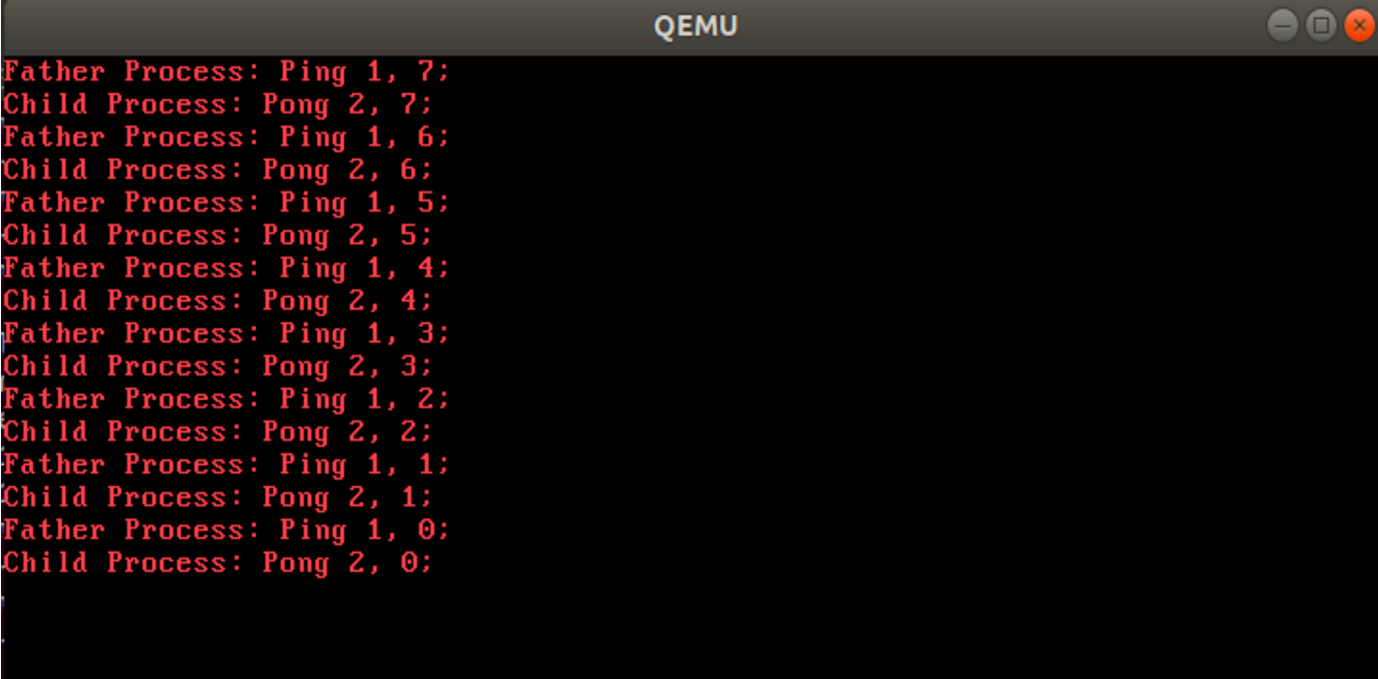
将当前进程的状态设置为STATE\_DEAD，然后模拟时钟中断进行进程切换

```
1 void syscallExit(struct StackFrame *sf) {
2     pcb[current].state = STATE_DEAD;
3     asm volatile("int $0x20");
4 }
```

### 3.4 中断嵌套

```
1     enableInterrupt();
2     for (int i = 0; i < 0x100000; i++) {
3         *(uint8_t *) (i + (index + 1) * 0x100000) = *(uint8_t *) (i + (current + 1) *
0x100000);
4         if(i%0x10000==0)
5             asm volatile("int $0x20");
6     }
7     disableInterrupt();
```

按照以上方式打开中断嵌套，运行结果如下

A screenshot of a QEMU terminal window. The window title is "QEMU". The terminal output shows a sequence of messages from two processes: "Father Process" and "Child Process". The Father Process sends "Ping" messages with values from 1 to 7, then 6, 5, 4, 3, 2, 1, and 0. The Child Process responds with "Pong" messages with the same values from 2 to 7, then 6, 5, 4, 3, 2, 1, and 0. The messages are interleaved, showing that the child process responds to each ping before the father process sends the next one.

```
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;
```

## 四、实验结果

测试结果如下图所示：

```
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;
```

## 五、感想与体会

本次实验难度和上一次相比差不多，但是写着特别舒服，因为实验的index介绍得很清楚，通过这次实验，更加清楚的理解了进程的切换，父子进程的创建等等问题，没有特别繁琐的过程，在这其中找到了实验的乐趣所在