操作系统lab3实验报告

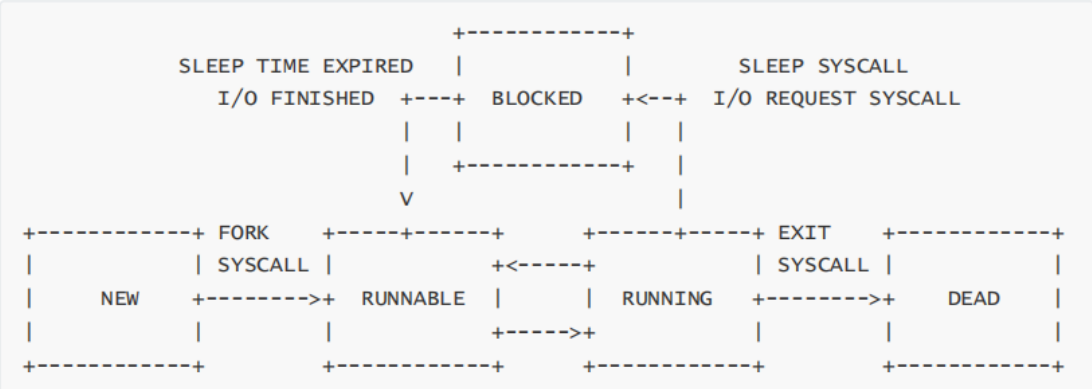
191220156　计算机科学与技术系　张桓　2659428453@qq.com

实验进度

- 实现了**基于时钟中断进行进程切换**完整任务调度的全过程，包括 `fork()` `sleep()` `exit()` 三个函数及其系统服务例程的实现，通过了 `app/main.c` 测试用例。
- 并且实现了**选做中的中断嵌套**，拷贝内存空间时开启时钟中断也可以通过测试用例。

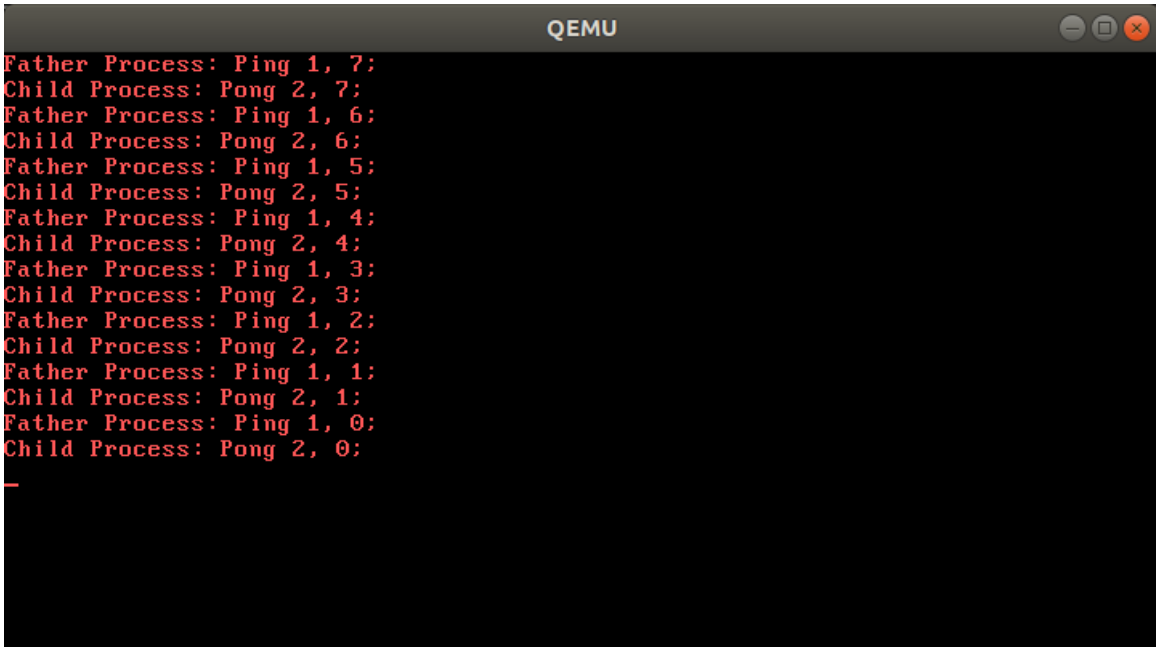
背景知识

- 测试用例中只涉及到**两个**进程的切换，整理一下出现**时间中断进程切换**的过程：
  - 进程 `P1` 在用户态执行， `8253` 可编程计时器**产生时间中断**
  - 依据 `TSS` 中记录的进程 `P1` 的 `SS0:ESP0` ，从 `P1` 的用户态堆栈切换至 `P1` 的内核堆栈，并将 `P1` 的现场信息压入内核堆栈中，**跳转执行时间中断处理程序**
  - 进程 `P1` 的处理**时间片耗尽**，切换至就绪状态的进程 `P2` ，并从当前 `P1` 的内核堆栈切换至 `P2` 的内核堆栈
  - 从进程 `P2` 的内核堆栈中弹出 `P2` 的现场信息，切换至 `P2` 的用户态堆栈，从时间中断处理程序返回执行 `P2`
- `fork` 系统调用用于创建子进程，内核需要为子进程分配一块独立的内存，将父进程的地址空间、用户态堆栈完全拷贝至子进程的内存中，并为子进程分配独立的进程控制块，完成对子进程的进程控制块的设置。
- 进程为操作系统资源分配的单位，每个进程都有独立的地址空间（代码段、数据段），独立的堆栈，独立的进程控制块，以下为一个广义的进程生命周期中的状态转换图：



实验结果

- `make play` 后屏幕输出如下：



- 开启中断嵌套后输出结果同样如上图所示

实验内容

完成库函数

> 为了完成 `app/main.c` 的测试用例，需要实现 `fork()` `sleep()` `exit()` 三个**库函数**包括其内部的**系统服务例程**。

- 这一部分的实现和 `lab2` 一样， `lab2` 中在 `syscall.c` 文件中实现 `printf()` 库函数，而我们知道 `printf()` 函数必须通过内核的 `syscall_write()` 才能实现，因此在库函数中会有诸如下面的的系统调用。

```
1   syscall(SYS_WRITE, STD_OUT, (uint32_t)buffer, (uint32_t)size, 0, 0);
```

同样 `fork()` `sleep()` `exit()` 函数的实现也必须通过内核完成。

- `fork()` 函数，通过调用 `syscall_fork()` 来实现，返回 `pid_t` 值：

```
1   pid_t fork() {
2       return syscall(SYS_FORK, 0, 0, 0, 0, 0);
3   }
```

- `sleep(uint32_t time)` 函数，通过 `syscall_sleep()` 来实现。带有参数 `time` 作为进程阻塞时间，因此在进行系统调用时也需要将 `time` 作为系统调用的参数。

```
int sleep(uint32_t time) {
    return syscall(SYS_SLEEP, time, 0, 0, 0, 0);
}
```

- `exit()`函数，通过调用`syscall_exit()`来实现。

```
int exit() {
    return syscall(SYS_EXIT, 0, 0, 0, 0, 0);
}
```

时间中断处理函数timeHandler()

> 本次实验中进程切换发生在**三个地方**：
>
> - 某个进程被`sleep()`阻塞后调度其他进程
> - 某个进程被`exit()`杀死后调度其他进程
> - 某个进程陷入时钟中断`timeHandler()`后**可能**需要调度其他进程

每个进程的时间片为`MAX_TIME_COUNT`，每经过一个时钟就会产生一个时钟中断，就会调用时钟中断处理函数`timeHandler()`，它实际上需要实现两个重要功能：

- 遍历进程`PCB`表，**维护每个进程的时间片和状态**，将状态为`STATE_BLOCKED`的进程的`sleepTime`减一，如果进程的`sleepTime`变为0，则将其状态设置为`STATE_RUNNABLE`。
- 将当前进程`pcb[current]`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**就切换进程，**否则继续**执行当前进程。切换进程时，将当前进程的时间片设置为0，状态设置为`STATE_RUNNABLE`，再去调度其他进程。

经过上面的分析也可以看出`timeHandler()`实际上的*主要工作*还在于**维护每个进程的时间信息**，进程切换**只发生在某个进程的时间片耗尽时**，实现代码如下：

```
void timerHandle(struct StackFrame *sf)
{
    int i = 0;
    for(i = 0; i < MAX_PCB_NUM; i++)
    {
        if(pcb[i].state == STATE_BLOCKED)
        {
            --pcb[i].sleepTime;
            if(pcb[i].sleepTime == 0)
                pcb[i].state = STATE_RUNNABLE;
        }
    }
```

```
13      ++pcb[current].timeCount;
14      if(pcb[current].timeCount >= MAX_TIME_COUNT)//current process's time have
    exhausted
15      {
16          pcb[current].timeCount = 0;
17          pcb[current].state = STATE_RUNNABLE;
18          schedule();
19      }
20      return ;
21  }
```

调度函数schedule()

> 上面代码的第 `18` 行 `schedule()` 是调度函数，经过上面分析可知调度发生的位置只可能有三个地方，因此完全可以**把调度函数抽象出来**，这样划分更清晰。

那么调度如何实现？**调度实际上就是调度算法 + 进程切换**：

- 本次实验选择的**调度算法**采用**轮转调度(Round Robin)**的策略，即依次调度进程1, 进程2, ..., 进程n, 进程1...。这样只需要**遍历** `pcb` **表**，当进程i的状态为 `STATE_RUNNABLE` 时**就切换到该进程。**
- 进程切换时，比如进程 `P1` 被时钟中断打断，陷入内核将 `P1` 的**现场信息**压入 `P1` 的内核栈中，若 `P1` 的时间片耗尽则**需要切换**到就绪态的进程 `P2` 。这时首先需要从 `P1` 的内核栈切换到 `P2` 的内核栈，然后从 `P2` 的内核栈中**弹出** `P2` **的现场信息**，再切换到 `P2` 的用户态堆栈即可。

  按照手册中给出的**示例进程切换代码**即可：

```
1   tmpStackTop = pcb[current].stackTop;
2   pcb[current].stackTop = pcb[current].prevStackTop;
3   tss.esp0 = (uint32_t)&(pcb[current].stackTop);
4   asm volatile("movl %0, %%esp"::"m"(tmpStackTop)); // switch kernel stack
5   asm volatile("popl %gs");
6   asm volatile("popl %fs");
7   asm volatile("popl %es");
8   asm volatile("popl %ds");
9   asm volatile("popal");
10  asm volatile("addl $8, %esp");
11  asm volatile("iret");
```

上述代码就实现了切换过程：

- 由于每个用户进程的**内核堆栈不同**，每次进程切换时需要将 `tss` 的 `esp0` 设置对应用户进程的内核堆栈位置，即 `tss.esp0 = (uint32_t)&(pcb[current].stackTop);` 这样**内核栈已经切换**成新进程的了。
- 接下来就是**弹出新进程的现场信息**，上述代码 `5-8` 行依次弹出 `GS FS ES DS` **段寄存器**，第9行 `popal` 指令弹出**所有通用寄存器**，这样就切换成了新进程的**用户态堆栈**了。

- 第11行`iret`指令从时间**中断处理程序返回**，开始执行新进程。

这样**调度算法 + 进程切换**两部分都实现了，整体的调度函数`schedule()`的实现如下：

```
1  void schedule()
2  {
3      int j = (current + 1) % MAX_PCB_NUM;
4      //find another runnable process
5      for(; j != current; j = (j + 1) % MAX_PCB_NUM)
6          if(pcb[j].state == STATE_RUNNABLE)
7              break;
8      //have runnalbe process, then change process
9      current = j;
10     pcb[current].state = STATE_RUNNING;
11     if(current == MAX_PCB_NUM) //have no other runnable, change into IDLE
12         while(1) { waitForInterrupt(); }
13
14     uint32_t tmpStackTop = pcb[current].stackTop;
15     pcb[current].stackTop = pcb[current].prevStackTop;
16     tss.esp0 = (uint32_t)&(pcb[current].stackTop);
17     asm volatile("movl %0, %%esp"::"m"(tmpStackTop)); // switch kernel stack
18     asm volatile("popl %gs");
19     asm volatile("popl %fs");
20     asm volatile("popl %es");
21     asm volatile("popl %ds");
22     asm volatile("popal");
23     asm volatile("addl $8, %esp");
24     asm volatile("iret");
25     return ;
26 }
```

- 注意到当找不到可以调度的就绪态进程时，第`12`行`while(1) { waitForInterrupt(); }`会调度`IDLE`程序将`CPU`挂起，等待中断到来。实际上这种情况在本实验是**不可能出现的**。

系统调用例程

syscallFork()函数

> `fork()`函数创建一个子进程，将父进程的资源复制给子进程。如果`fork()`返回-1则创建失败，成功则父进程返回子进程`PID`，子进程返回0。
>
> - 主要分成三个任务：
>   - 找到一个空闲的`PCB`用于分配给子进程
>   - 拷贝父进程的代码段和数据段拷贝到子进程的内存
>   - 设置子进程的`PCB`块

> - 为父进程和子进程设置对应的返回值

- 寻找空闲`PCB`直接遍历`pcb`表，找一个状态为`STATE_DEAD`的进程即可。
- 拷贝父进程的代码段和数据段比较简单，由于进程i即`pcb[i]`对应的内存起始地址为`(i + 1)* 0x100000`，大小为`0x100000`，因此**直接for遍历**父进程的内存部分全部赋值给子进程的内存即可。
- 设置子进程的`PCB`块时，我**参考`kernel/kvm.c`中`initProc()`函数中的`user process`部分**。我的做法是先**完全**将父进程的`PCB`块拷贝给子进程，然后再**修改子进程PCB块的部分内容**：
  - 显然子进程`sleepTime`和`timeCount`都应该是0。
  - 子进程的`PID`为`i`，状态为`STATE_RUNNABLE`
  - 内核栈顶信息**参考示例**：`pcb[i].stackTop = (uint32_t)&(pcb[i].regs);` `pcb[i].prevStackTop = (uint32_t)&(pcb[i].stackTop);`
  - 设置**子进程的段寄存器**，参考**示例中的进程`PID`为`1`**，它的段寄存器设置如下：

```
pcb[1].regs.ss = USEL(4);
pcb[1].regs.cs = USEL(3);
pcb[1].regs.ds = USEL(4);
pcb[1].regs.es = USEL(4);
pcb[1].regs.fs = USEL(4);
pcb[1].regs.gs = USEL(4);
```

这一部分我开始比较疑惑，不知道`USEL()`中的数字怎么填写。后来在`kernel/kvm.c`中的`initSeg()`函数中设置`GDT`表，发现**每个用户进程`pcb[i]`的段描述符**是通过以下代码设置的：

```
for (i = 1; i < MAX_PCB_NUM; i++) {
        gdt[1+i*2] = SEG(STA_X | STA_R, (i+1)*0x100000,0x00100000,
DPL_USER);
        gdt[2+i*2] = SEG(STA_W,         (i+1)*0x100000,0x00100000,
DPL_USER);
}
```

结合示例给出的`pcb[1]`的段寄存器设置，可以知道用户进程`pcb[i]`的段寄存器设置应该为：

```
//set segment regs
pcb[i].regs.ss = USEL(2+2*i);
pcb[i].regs.cs = USEL(1+2*i);
pcb[i].regs.ds = USEL(2+2*i);
pcb[i].regs.es = USEL(2+2*i);
pcb[i].regs.fs = USEL(2+2*i);
pcb[i].regs.gs = USEL(2+2*i);
```

- 为父进程和子进程设置返回值时，返回值应该放在各自进程的`EAX`寄存器中，即`pcb[i].regs.eax`。根据`fork()`函数的功能，父进程`EAX`设置为子进程`PID`，子进程`EAX`设置为0。

综上可以得到`syscallFork()`函数的整体代码如下：

```
void syscallFork(struct StackFrame *sf)
{
    int i = 0;
    for(i = 0; i < MAX_PCB_NUM; i++)//find a empty PCB
        if(pcb[i].state == STATE_DEAD)
            break;
    if(i == MAX_PCB_NUM)//no PCB is empty
    {
        pcb[current].regs.eax = -1;//unsuccessful
        return;
    }
    else//copy dad's userspace all
    {
        int child_begin = (i + 1)*0x100000;
        int dad_begin = (current + 1)*0x100000;
        enableInterrupt();
        for(int offset = 0; offset < 0x100000; offset++)
        {
            *(uint8_t *)(child_begin + offset) = *(uint8_t *)(dad_begin + offset);
            if(offset % 1000 == 0)
                asm volatile("int $0x20"); //XXX Testing irqTimer during syscall
        }
        disableInterrupt();
        //copy dad's PCB to kid
        for (int j = 0; j < sizeof(ProcessTable); ++j)
            *((uint8_t *)(&pcb[i]) + j) = *((uint8_t *)(&pcb[current]) + j);
        //modify kid's pid, state, time, stack
        pcb[i].stackTop = (uint32_t)&(pcb[i].regs);
        pcb[i].prevStackTop = (uint32_t)&(pcb[i].stackTop);
        pcb[i].state = STATE_RUNNABLE;
        pcb[i].timeCount = 0;
        pcb[i].sleepTime = 0;
        pcb[i].pid = i;
        //return value
        pcb[i].regs.eax = 0;//for kid return 0
        pcb[current].regs.eax = i;//for dad return kid's pid
        //set segment regs
        pcb[i].regs.ss = USEL(2+2*i);
        pcb[i].regs.cs = USEL(1+2*i);
        pcb[i].regs.ds = USEL(2+2*i);
        pcb[i].regs.es = USEL(2+2*i);
        pcb[i].regs.fs = USEL(2+2*i);
        pcb[i].regs.gs = USEL(2+2*i);
    }
```

```
45      return ;
46  }
```

---

syscallSleep()函数

> 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` 。根据之前的分析， `sleep()` 函数属于调度发生的三个地方，因此设置完 `PCB` 后还需要调用**调度函数** `schedule()` 。

- 由于 `sleep()` 函数的库函数设置中的系统调用为： `syscall(SYS_SLEEP, time, 0, 0, 0, 0);` 。查看 `syscall()` 函数原型：

```
1   int32_t syscall(int num, uint32_t a1,uint32_t a2,
2           uint32_t a3, uint32_t a4, uint32_t a5)
3   {
4       ...
5           asm volatile("movl %0, %%ecx"::"m"(a1));
6       ...
7   }
```

  可知参数 `time` 实际上被放在了 `ECX` 中，因此设置当前进程的 `sleepTime` ： `pcb[current].sleepTime = sf->ecx;` 。

这样整体代码如下：

```
1   void syscallSleep(struct StackFrame *sf)
2   {
3       pcb[current].sleepTime = sf->ecx; //look in Function: syscall()
4       pcb[current].state = STATE_BLOCKED;
5       schedule();
6   }
```

`shchedule()` 函数为调度函数，定义在这里。

---

syscallExit()函数

> 将当前进程的状态设置为 `STATE_DEAD` ，同样 `exit()` 函数属于调度发生的三个地方，还需要调用调度函数 `schedule()` 。

```
void syscallExit(struct StackFrame *sf)
{
    pcb[current].state = STATE_DEAD;
    schedule();
}
```

`shchedule()` 函数定义在这里。

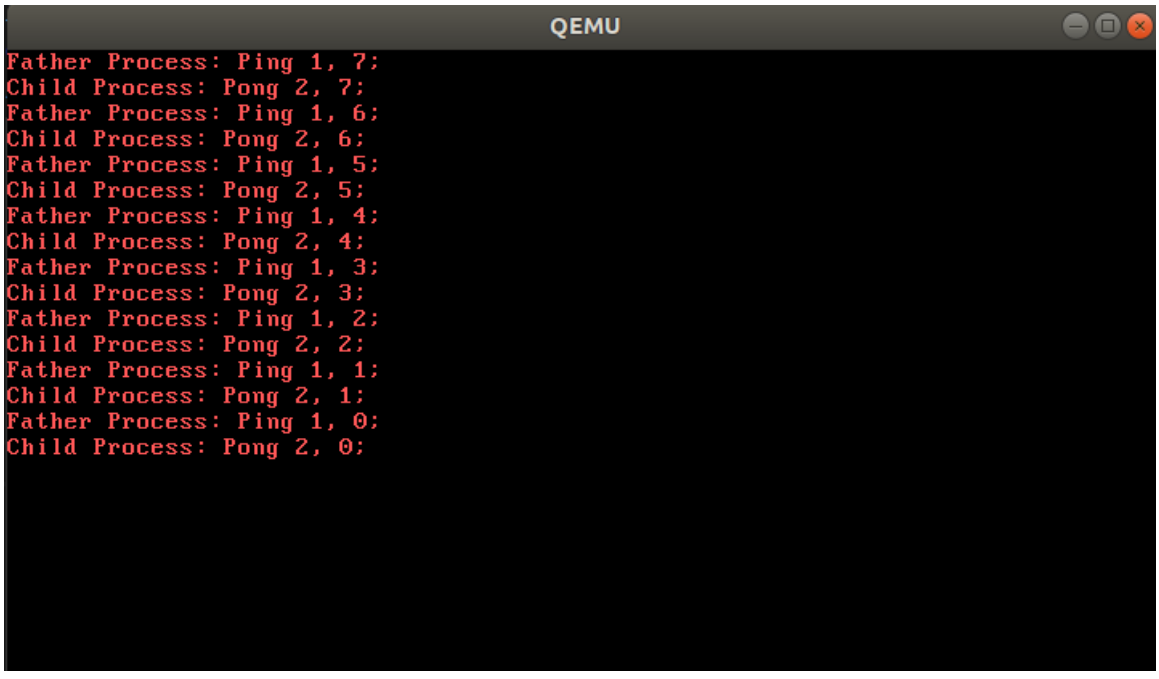
选做：中断嵌套

中断嵌套发生在 `syscallFork()` 函数中，该函数的定义在这里，**子进程拷贝父进程用户空间内容**的代码如下：

```
enableInterrupt();
for(int offset = 0; offset < 0x100000; offset++)
{
        *(uint8_t *)(child_begin + offset) = *(uint8_t *)(dad_begin + offset);
        if(offset % 1000 == 0)
            asm volatile("int $0x20"); //XXX Testing irqTimer during syscall
}
disableInterrupt();
```

我们用 `enableInterrupt()` 函数使得在拷贝内存空间过程中开启了嵌套中断，并且用 `int $0x20` 模拟时钟中断。为了避免每拷贝一个单元就陷入嵌套时钟中断，导致**拷贝过程过慢**，我设置每拷贝 `1000` 个单元陷入一次中断，即 `offset % 1000 == 0` 时才产生一个时钟中断。

这样一来仍然可以通过测试用例：



但是区别在于`make play`之后需要等待较长的一段时间后才会有显示，这个原因也很容易知道，因为测试用例代码如下：

```
int uEntry(void)
{
    int ret = fork();
    int i = 8;
    if (ret == 0)
    {
        sleep(128);
        exit();
    }
    else if (ret != -1)
    {
        sleep(128);
        exit();
    }
    while(1);
    return 0;
}
```

由于中断嵌套只发生在`fork()`函数中，而之后的`sleep()`和`exit()`都不受影响，因此只有在程序开始`fork()`子进程时由于不断处理中断嵌套，**时间消耗会变长**，`fork()`完之后恢复正常。

- srds，在做完必做之后尝试中断嵌套居然直接就过了，这是什么原理呢？后来才发现手册中有一部分内容：

  > 说到这里得提一下`irqHandle`对比lab2也增加了部分保存与恢复的内容，同学们自行理解

  ```
  void irqHandle(struct TrapFrame *tf) { // pointer tf = esp
     asm volatile("movw %%ax, %%ds"::"a"(KSEL(SEG_KDATA)));

  +   uint32_t tmpStackTop = pcb[current].stackTop;
  +   pcb[current].prevStackTop = pcb[current].stackTop;
  +   pcb[current].stackTop = (uint32_t)tf;
  ```

这个对于`irqHandle()`的修改原来我是直接`copy`的，没有在意什么目的，经过我的理解，这应该就是实现中断嵌套的关键，代码如下：

```
void irqHandle(struct StackFrame *sf) { // pointer sf = esp
    /* Reassign segment register */
    asm volatile("movw %%ax, %%ds"::"a"(KSEL(SEG_KDATA)));
    /*TODO Save esp to stackTop */

    +uint32_t tmpStackTop = pcb[current].stackTop;
    +pcb[current].prevStackTop = pcb[current].stackTop;
    +pcb[current].stackTop = (uint32_t)sf;

    switch(sf->irq) {
        case -1:
```

```
            break;
        case 0xd:
            GProtectFaultHandle(sf);
            break;
        case 0x20:
            timerHandle(sf);
            break;
        case 0x80:
            syscallHandle(sf);
            break;
        default:assert(0);
    }
    /*TODO Recover stackTop */
    +pcb[current].stackTop = tmpStackTop;
}
```

比如在`syscallFork()`中发生嵌套中断`int $0x20`后，会调用中断处理程序`irqHandle()`，添加的那部分内容：

```
+uint32_t tmpStackTop = pcb[current].stackTop;
+pcb[current].prevStackTop = pcb[current].stackTop;
+pcb[current].stackTop = (uint32_t)sf;
```

首先将原来进程的栈顶信息`pcb[current].stackTop`保存到`tmpStackTop`中，再将它保存到中断嵌套时保存**待恢复的栈顶信息**`prevStackTop`中。这样一来**被嵌套中断的进程**的栈顶信息就存好了，可以将**当前栈顶**切换成**中断进程**了。

当中断处理完后，用`pcb[current].stackTop = tmpStackTop;`恢复原来**被嵌套中断的进程**的栈顶信息，这样就回到了原进程，从而正确实现了中断嵌套。

> 注意:中断嵌套的相关代码在源码中对应位置被我**注释掉了**，如果需要检查**去掉注释**即可。

实验心得

这次实现**相比上一个实验**是比较简单的。一方面由于上个实验中对库函数和系统服务例程的关系，以及中断的实现都有了清楚的了解，所以做本实验时不像上次那样一头雾水，这次很清楚自己的工作就是实现三个函数的库函数及对应系统服务例程。

当然本次实验名曰：进程切换，可实际上实现时考虑切换的地方并不太多，堆栈切换及进程切换的代码教程中都给出了示例，主要在于理解了不同用户进程切换的内部工作，调度算法也采用最简单的轮询法。学到的东西更多在于`fork()  sleep()  exit()`三个函数本身的功能，实现细节等等。

- 参考文章：

  fork()函数的使用与底层原理

  fork的原理及实现

最后感谢助教gg/助教jj的耐心批改☺☺

如有疑问请联系我：2659428453@qq.com