Lab3 实验报告

231880101 孙俊晖

邮箱：2033282932@qq.com

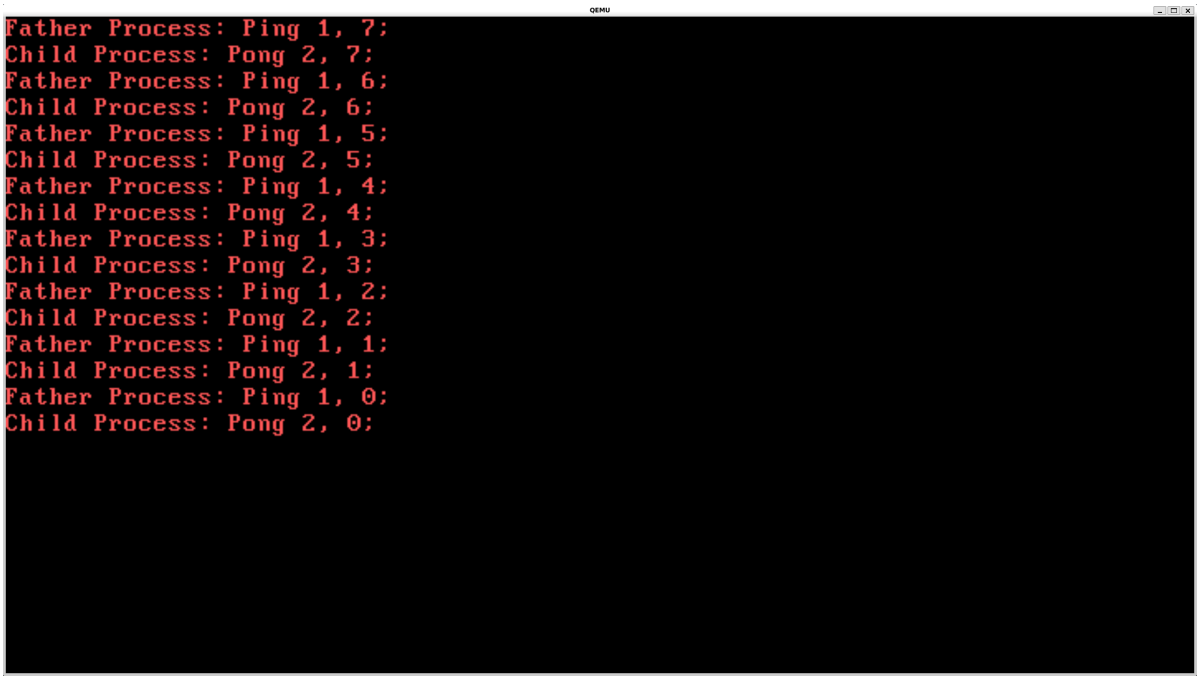
一、实验完成情况

1. 实验进度

我完成了除选做内容3.5(临界区)以外的所有内容。

2. 实验结果

我在启用嵌套中断的情况下顺利通过了用户程序的测试，说明我成功实现了此次实验要求的自制简单操作系统的进程管理功能以及中断嵌套的部分。



3. 实验过程

（1）完成fork, sleep, exit库函数

这个部分异常简单，就是单纯的系统调用，只需要在lab3/lib/syscall.c中调用syscall补全库函数即可。

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


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

int exit() {
```

```
  syscall(SYS_EXIT, 0, 0, 0, 0, 0);
    return 0;
}
```

（2）时钟中断处理

接下来的实验步骤都是在lab3/kernel/kernel/irqHandle.c中完成。

这部分的核心是完成timerHandle函数，用于进程切换。

首先，根据实验文档的指导，先遍历pcb, 将状态为STATE_BLOCKED的进程的sleepTime减一, 如果进程的sleepTime变为0, 说明BLOCKED状态的进程的时间片耗尽，该进程将重新进入RUNNABLE状态。

```
//在irqHandle.c的开头定义全局变量pcb
extern ProcessTable pcb[];
//遍历pcb
for(int i = 0; i < MAX_PCB_NUM; i++)
  {
    if(pcb[i].state == STATE_BLOCKED)
    {
      //将状态为STATE_BLOCKED的进程的sleepTime减一
      if(pcb[i].sleepTime > 0)
      {
        pcb[i].sleepTime--;
      }
      //BLOCKED状态的进程的时间片耗尽
      if(pcb[i].sleepTime == 0)
      {
        pcb[i].state = STATE_RUNNABLE;
      }
    }
}
```

然后，将当前进程的timeCount加一, 如果时间片用完且有其它状态为 STATE_RUNNABLE的进程, 则当前进程进入RUNNABLE状态，将进程切换到其他状态为 STATE_RUNNABLE的进程, 否则继续执行当前进程。其中，实验文档中已经给出了进程切换的代码，可以直接使用。在寻找其他状态为STATE_RUUNALE的进程时，我并没有选择特殊考虑内核IDLE进程(pcb[0])，这对实验结果没有造成影响。

```
pcb[current].timeCount++;
//时间片耗尽
if(pcb[current].timeCount >= MAX_TIME_COUNT)
{
    pcb[current].timeCount = 0;
    pcb[current].state = STATE_RUNNABLE;
    int next = current;
    //寻找其他状态为STATE_RUUNALE的进程
    for(int j = (current + 1) % MAX_PCB_NUM; j != current; j = (j + 1) % MAX_PCB_NUM)
    {
        if(pcb[j].state == STATE_RUNNABLE)
        {
            next = j;
            break;
        }
    }
```

```
    pcb[current].state = STATE_RUNNING;
    //如果找到了其他状态为STATE_RUUNALE的进程，则进行进程切换
    if(next != current)
    {
        current = next;
        uint32_t tmpStackTop = pcb[current].stackTop;
        pcb[current].stackTop = pcb[current].prevStackTop;
        tss.esp0 = (uint32_t)&(pcb[current].stackTop);
        asm volatile("movl %0, %%esp"::"m"(tmpStackTop)); // switch kernel stack
        asm volatile("popl %gs");
        asm volatile("popl %fs");
        asm volatile("popl %es");
        asm volatile("popl %ds");
        asm volatile("popal");
        asm volatile("addl $8, %esp");
        asm volatile("iret");
    }
}
```

这样，就成功实现了timerHandle函数。

与Lab2有所不同，根据实验文档的指导，本次实验的irqHandle函数中增加了保存与恢复堆栈指针的部分。

```
void irqHandle(struct StackFrame *sf) {
    //重新分配段寄存器
    asm volatile("movw %%ax, %%ds"::"a"(KSEL(SEG_KDATA)));
    //保存堆栈指针
    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);
            brea      default:ast(0);
    }
    //恢复堆栈指针
    pcb[current].stackTop = tmpStackTop;
}
```

(3) 实现相关系统调用例程

1. 完成syscallFork函数

首先，寻找一个可用pcb块作为子进程。如果没有空闲pcb，说明fork失败，父进程返回-1(返回值在eax中)。

```
int i;
//寻找空闲的pcb块
for(i = 0; i < MAX_PCB_NUM; i++)
{
    if(pcb[i].state == STATE_DEAD)
    {
        break;
    }
}
//fork失败，父进程返回-1
if(i == MAX_PCB_NUM)
{
    pcb[current].regs.eax = -1;
    return;
}
```

然后进行内存拷贝，将父进程地址空间的内容复制给子进程。为了测试3.4的中断嵌套，需要在这里开启嵌套中断，并手动模拟时钟中断。

```
//打开嵌套中断
enableInterrupt();
//内存拷贝
for (int j = 0; j < 0x100000; j++) {
    *(uint8_t *)(j + (i + 1) * 0x100000) = *(uint8_t *)(j + (current +1)
* 0x100000);
    //模拟时钟中断
    asm volatile("int $0x20");
}
//关闭中断
disableInterrupt();
```

接下来就是pcb的复制。由于prevStackTop和stackTop暂存的是前序进程和当前进程栈基址esp的相对地址，故需要对其进行相对地址的计算转换以得到子进程的prevStackTop和stackTop。子进程的state，timeCount，sleepTime以及pid的初始化可以参考后面initProc()中pcb[0]和pcb[1]的初始化。

```
pcb[i].stackTop = pcb[current].stackTop - (uint32_t)&(pcb[current]) +
(uint32_t)&(pcb[i]);
pcb[i].prevStackTop = pcb[current].prevStackTop - (uint32_t)&
(pcb[current]) + (uint32_t)&(pcb[i]);
pcb[i].state = STATE_RUNNABLE;
pcb[i].timeCount = 0;
pcb[i].sleepTime = 0;
pcb[i].pid = i;
```

参考lab3/kernel/kernel/kvm.c中initProc()初始化pcb[0]和pcb[1]的部分，

```
void initProc() {
    int i;
    for (i = 0; i < MAX_PCB_NUM; i++) {
        pcb[i].state = STATE_DEAD;
    }
    // kernel process
    pcb[0].stackTop = (uint32_t)&(pcb[0].stackTop);
    pcb[0].prevStackTop = (uint32_t)&(pcb[0].stackTop);
    pcb[0].state = STATE_RUNNING;
    pcb[0].timeCount = MAX_TIME_COUNT;
    pcb[0].sleepTime = 0;
    pcb[0].pid = 0;
    // user process
    pcb[1].stackTop = (uint32_t)&(pcb[1].regs);
    pcb[1].prevStackTop = (uint32_t)&(pcb[1].stackTop);
    pcb[1].state = STATE_RUNNABLE;
    pcb[1].timeCount = 0;
    pcb[1].sleepTime = 0;
    pcb[1].pid = 1;
    pcb[1].regs.ss = USEL(4);
    pcb[1].regs.esp = 0x100000;
    asm volatile("pushfl");
    asm volatile("popl %0":"=r"(pcb[1].regs.eflags));
    pcb[1].regs.eflags = pcb[1].regs.eflags | 0x200;
    pcb[1].regs.cs = USEL(3);
    pcb[1].regs.eip = loadUMain();
    pcb[1].regs.ds = USEL(4);
    pcb[1].regs.es = USEL(4);
    pcb[1].regs.fs = USEL(4);
    pcb[1].regs.gs = USEL(4);

    current = 0; // kernel idle process
    asm volatile("movl %0, %%esp"::"m"(pcb[0].stackTop)); // switch to
kernel stack for kernel idle process
    enableInterrupt();
    asm volatile("int $0x20"); // trigger irqTimer
    while(1)
        waitForInterrupt();
}
```

可以推断除了寄存器ss,cs,ds,es,fs,gs需要经过计算得到以外，以及eax中是返回值，其余寄存器直接可以直接复制。

```
pcb[i].regs.edi = pcb[current].regs.edi;
pcb[i].regs.esi = pcb[current].regs.esi;
pcb[i].regs.ebp = pcb[current].regs.ebp;
pcb[i].regs.xxx = pcb[current].regs.xxx;
pcb[i].regs.ebx = pcb[current].regs.ebx;
pcb[i].regs.edx = pcb[current].regs.edx;
pcb[i].regs.ecx = pcb[current].regs.ecx;
pcb[i].regs.irq = pcb[current].regs.irq;
pcb[i].regs.error = pcb[current].regs.error;
pcb[i].regs.eip = pcb[current].regs.eip;
pcb[i].regs.eflags = pcb[current].regs.eflags;
pcb[i].regs.esp = pcb[current].regs.esp;
```

在fork成功的情况下，子进程返回0，父进程返回子进程pid。由此可以设置返回值(在eax中)。

```
pcb[current].regs.eax = i;
pcb[i].regs.eax = 0;
```

根据kvm.c中initSeg()中gdt表的处理，

```
void initSeg() { // setup kernel segements
    int i;

    gdt[SEG_KCODE] = SEG(STA_X | STA_R, 0,       0xffffffff, DPL_KERN);
    gdt[SEG_KDATA] = SEG(STA_W,         0,       0xffffffff, DPL_KERN);
    //参考这部分对gdt的处理
    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);
    }

    gdt[SEG_TSS] = SEG16(STS_T32A,      &tss, sizeof(TSS)-1, DPL_KERN);
    gdt[SEG_TSS].s = 0;

    setGdt(gdt, sizeof(gdt)); // gdt is set in bootloader, here reset
gdt in kernel

    /* initialize TSS */
    tss.ss0 = KSEL(SEG_KDATA);
    asm volatile("ltr %%ax":: "a" (KSEL(SEG_TSS)));

    /* reassign segment register */
    asm volatile("movw %%ax,%%ds":: "a" (KSEL(SEG_KDATA)));
    asm volatile("movw %%ax,%%ss":: "a" (KSEL(SEG_KDATA)));

    lLdt(0);

}
```

以及initProc()中对pcb[1]ss,cs,ds,es,fs,gs的初始化，

```
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);
```

可知pcb[i]中ss,cs,ds,es,fs,gs应该这样初始化(cs与其他几个寄存器的初始化不同):

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

这样，就成功实现了syscallFork函数。

2. 完成syscallSleep函数

首先，将当前进程的状态设置为STATE_BLOCKED。然后，与timeHandle函数的实现类似，寻找其他状态为STATE_RUUNALE的进程(找不到就切换到内核进程pcb[0])，并切换进程。

```
void syscallSleep(struct StackFrame *sf) {
    //将当前进程的状态设置为STATE_BLOCKED
    pcb[current].state = STATE_BLOCKED;
    int next = current;
    for(int i = (current + 1) % MAX_PCB_NUM; i != current; i = (i + 1) % MAX_PCB_NUM)
    {
        if(pcb[i].state == STATE_RUNNABLE)
        {
            next = i;
            break;
        }
    }
    if(next == current)
    {
        current = 0;
    }
    else
    {
        current = next;
    }
    pcb[current].state = STATE_RUNNING;
    uint32_t tmpStackTop = pcb[current].stackTop;
    pcb[current].stackTop = pcb[current].prevStackTop;
    tss.esp0 = (uint32_t)&(pcb[current].stackTop);
    asm volatile("movl %0, %%esp"::"m"(tmpStackTop)); // switch kernel stack
    asm volatile("popl %gs");
    asm volatile("popl %fs");
    asm volatile("popl %es");
    asm volatile("popl %ds");
    asm volatile("popal");
```

```
    asm volatile("addl $8, %esp");
    asm volatile("iret");
}
```

3. 完成syscallExit函数

与syscallSleep函数的实现基本相同，唯一的不同就是一开始将当前进程的状态设置为STATE_DEAD。

```
void syscallExit(struct StackFrame *sf) {
    //将当前进程的状态设置为STATE_DEAD
    pcb[current].state = STATE_DEAD;
    int next = current;
    for(int i = (current + 1) % MAX_PCB_NUM; i != current; i = (i + 1) % MAX_PCB_NUM)
    {
        if(pcb[i].state == STATE_RUNNABLE)
        {
            next = i;
            break;
        }
    }
    if(next == current)
    {
        current = 0;
    }
    else
    {
        current = next;
    }
    pcb[current].state = STATE_RUNNING;
    uint32_t tmpStackTop = pcb[current].stackTop;
    pcb[current].stackTop = pcb[current].prevStackTop;
    tss.esp0 = (uint32_t)&(pcb[current].stackTop);
    asm volatile("movl %0, %%esp"::"m"(tmpStackTop)); // switch kernel stack
    asm volatile("popl %gs");
    asm volatile("popl %fs");
    asm volatile("popl %es");
    asm volatile("popl %ds");
    asm volatile("popal");
    asm volatile("addl $8, %esp");
    asm volatile("iret");
}
```

在完成syscallFork,syscallSleep,syscallExit后，补充syscallHandle函数，根据系统调用号调用对应函数(本次实验不涉及syscallExec)：

```
#define SYS_WRITE 0
#define SYS_FORK 1
#define SYS_EXEC 2
#define SYS_SLEEP 3
#define SYS_EXIT 4
void syscallHandle(struct StackFrame *sf) {
    switch(sf->eax) { // syscall number
        case 0:
```

```
            syscallWrite(sf);
            break; // for SYS_WRITE
            /*TODO Add Fork,Sleep... */
        case 1:
            syscallFork(sf);
            break;
        case 2:
            //syscallExec(sf);
            break;
        case 3:
            syscallSleep(sf);
            break;
        case 4:
            syscallExit(sf);
            break;
        default:break;
    }
}
```

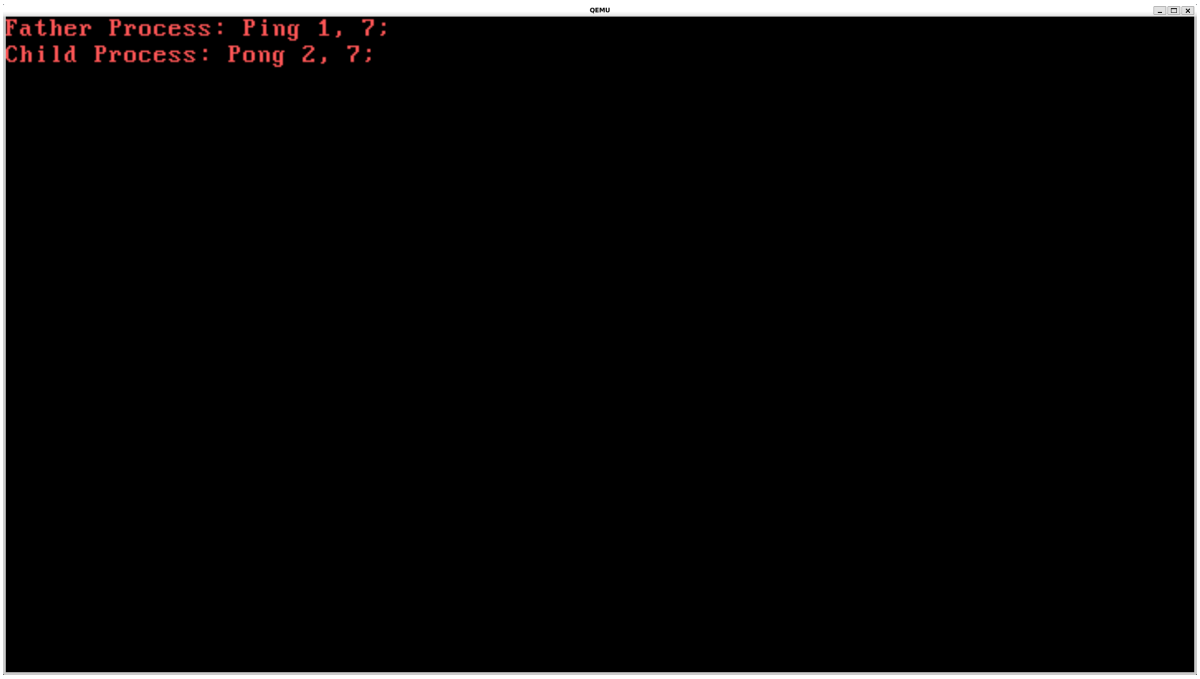
至此，就完成了本次实验的所有内容(除3.5以外)。

二、实验中遇到的问题

我在timerHandle函数中遍历pcb时，一开始没有判断状态为STATE_BLOCKED的进程的sleepTime是否大于0，就直接减一了。

```
for(int i = 0; i < MAX_PCB_NUM; i++)
  {
    if(pcb[i].state == STATE_BLOCKED)
    {
      pcb[i].sleepTime--; //应该是if(pcb[i].sleepTime>0)pcb[i].sleepTime--;
      //BLOCKED状态的进程的时间片耗尽
      if(pcb[i].sleepTime == 0)
      {
        pcb[i].state = STATE_RUNNABLE;
      }
    }
}
```

然后导致用户程序的测试与期望输出有很大差异：



这让我意识到写代码时还应该更细致一点，差之毫厘就足以谬以千里。

三、实验总结

在本次实验中，我主要完成了自制简单操作系统的进程管理功能, 通过手动实现一个简单的任务调度，我对时间中断、进程切换以及轮转调度策略都有了更深的理解，虽然对一些底层的东西还是一知半解的状态，但我仍收获满满。