OS-lab3

211275009 陈铭浩
email：**211275009@smail.nju.edu.cn**
实验进度：我完成了所有内容

1. 完成三个库函数(syscall.c)

- 直接在对应函数中填入相应系统调用即可

2. 时钟中断处理（irqHandle.c）

- 完成timerHandle()函数，逻辑为首先遍历pcb，将状态为STATE_BLOCKED的进程的sleepTime减一，如果进程的sleepTime变为0，重新设为STATE_RUNNABLE；而后将当前进程的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，否则继续执行当前进程

```
void timerHandle(struct StackFrame *sf)
{
    //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
    for(int 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;
            }
        }
    }
    //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，否则继续执行当前进程
    //putNum(pcb[current].timeCount);
    //if(pcb[current].timeCount==16){
    //  pcb[current].timeCount=0;
        //return;
    //}
    pcb[current].timeCount++;
    //putNum(pcb[current].timeCount);
    if(pcb[current].timeCount>=MAX_TIME_COUNT){
        pcb[current].timeCount=0;
        pcb[current].state=STATE_RUNNABLE;
        for (int i = (current + 1) % MAX_PCB_NUM; i != current; i = (i+1) % MAX_PCB_NUM) {
            if (pcb[i].state == STATE_RUNNABLE && i != 0) {
                current = i;
                break;
            }
        }
        pcb[current].state = STATE_RUNNING;
        switch_process();
    }else{
        return;
    }
}
```

- 这里由于idle进程的pcb块的timeCount块被初始化为MAX_TIME_COUNT，所以需要小心这里边界条件的处理，必须是>=（而不能是==），否则刚开始时pcb[0]的timeCount会自增为17，大于MAX_TIME_COUT，永远自增下去

3. 实现三个系统调用例程（irqHandle.c）

- 这里我首先将进程切换和轮转调度进程分别封装成了两个独立的函数，需要时直接调用即可

```
void set_new_running_process(void)
{
    //轮转调度
    int if_find = 0;
    for (int i = (current + 1) % MAX_PCB_NUM; i != current; i = (i+1) % MAX_PCB_NUM) {
        if (pcb[i].state == STATE_RUNNABLE && i != 0) {
            if_find = 1;
            current = i;
            break;
        }
    }
    if (!if_find) current = 0;
    pcb[current].state = STATE_RUNNING;

}
```

```
void switch_process(void)
{
    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");
}
```

3.1 syscallFork()

- 首先寻找一个空闲的pcb作为子进程的进程控制块，然后复制相关资源
- 将地址空间的内容全部复制
- 结合kvm.c中initSeg()中可以看出，pcb[i]对应进程的代码段放在gdt的第2*i+1个表项中，其余段放在第2*i+2个表项中，因此此处做相应设置
- 其余仿照initProc()进行初始化即可

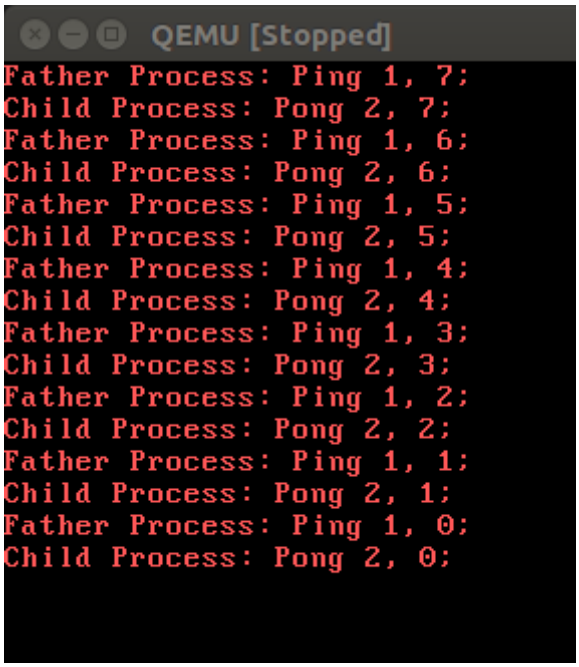
3.2 syscallSleep()

- 首先检查传入参数的合法性（是否<0），不合法直接返回-1
- 确定合法后设置对应状态和sleepTime，然后调用set_new_running_process()找到下一个可以run的进程，最后调用switch_process()即可

3.3 syscallExit()

- 设置状态为DEAD，然后调用set_new_running_process()找到下一个可以run的进程，最后调用switch_process()即可

4. 实验结果


QEMU [Stopped]
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;