

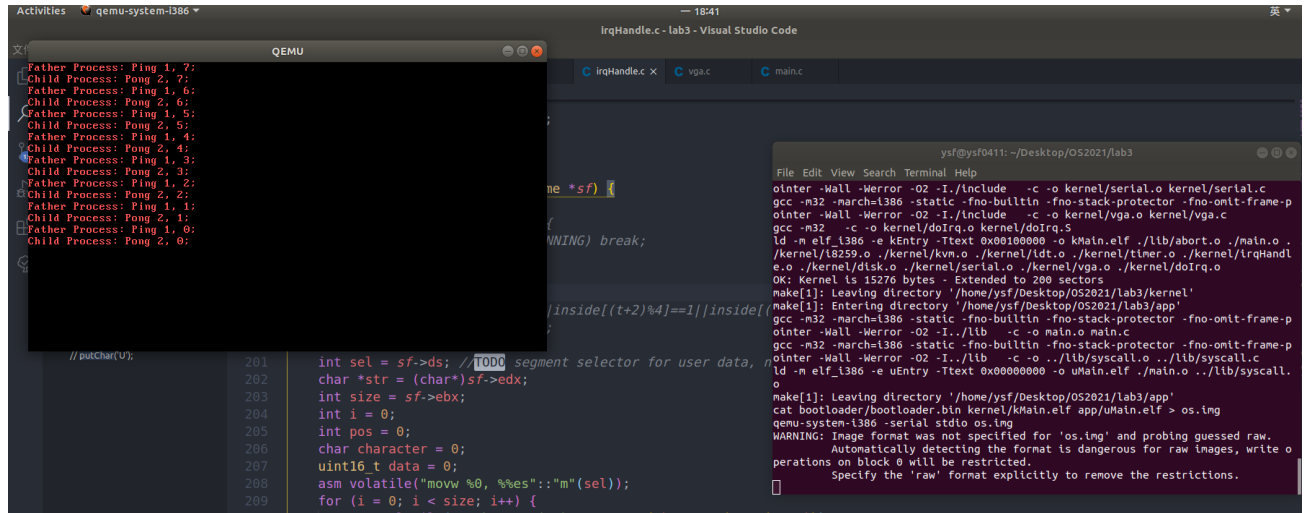
Lab3 实验报告

191300073 杨斯凡 191300073@smail.nju.edu.cn

一、试验进度

我完成了所有试验内容，包括选做部分。

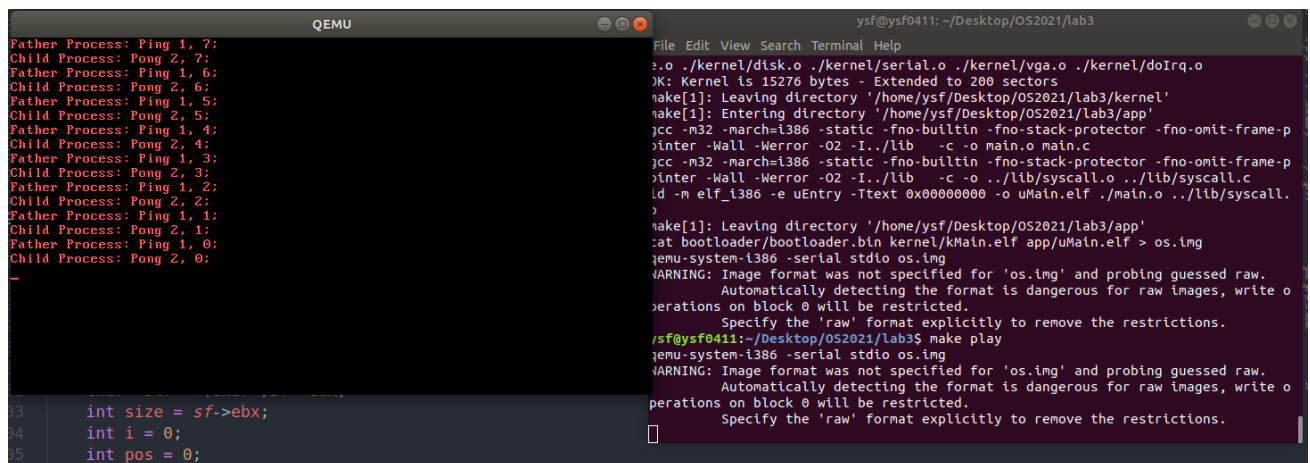
二、实验结果



```
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:

// bufchar(U);
201 int sel = sf->ds; // TODD segment selector for user data, r
202 char *str = (char*)sf->edx;
203 int size = sf->ebx;
204 int i = 0;
205 int pos = 0;
206 char character = 0;
207 uint16_t data = 0;
208 asm volatile("movw %0, %%es":"m"(sel));
209 for (i = 0; i < size; i++) {
```

(未加入临界区)



```
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:

13 int size = sf->ebx;
14 int i = 0;
15 int pos = 0;
```

(加入临界区后)

三、修改代码位置

首先我阅读了整个框架代码，和 lab2 的差别不是很大，具体多了几个函数，因此两者的具体过程是大同小异的。在阅读了整个框架后，我依照了教程上的指导对代码进行了填写。

首先需要在 syscall.c 中加入这几个函数的系统调用，使得调用的时候可以进行系统调用。

然后进行时钟中断的处理，首先是找到所有处于阻塞态的进程，对他们的等待时间减少 1，如果 sleeptime 为 0，则表示等待的事件已经完成，切换为就绪态（RUNNABLE）。然后判断当前进程是否为运行态，如果为运行态，则将其运行时间加一，如果运行时间未到限制，则不需要调度，直接返回即可。如果时间达到限制，则说明需要进行调度，在所有进程中进行查找，如果有就绪态的进程，则将其挂起执行，如果没有就绪态的进程，则执行内核进程。

然后进行进程的切换即可。并且在 irqHandle 中增加保存与恢复的内容。

对于 syscallFork, 首先要找到一个空闲的进程块, 如果没有, 那么就返回-1, 表示 fork 失败, 如果找到了, 那么就将当前进程的内容拷贝到该空闲进程块中, 首先拷贝内存, 然后拷贝栈, 栈顶, 通用寄存器等一系列寄存器, 进程名, 然后把进程状态设为执行, 运行时间设为 0。对于 syscallSleep, 把进程的 sleepTime 设置参数, 将进程的状态设置为阻塞态, 然后使用“int \$0x20”模拟时钟中断即可。syscallExit, 将当前进程的状态设置为阻塞态, 然后模拟时钟中断进行进程切换即可。

对于临界区, 我是用了 Peterson 算法进行保护, 使得在临界区的时候不能进行进程切换, 我是用了两个全局标量, turn 表示当前进程号, inside 数组表示每个进程的状态, 当 while 条件成立的时候, 必须等到切换到当前进程才能继续进行。

```
void syscallPrint(struct StackFrame *sf) {
    int t;
    for(t=0;t<MAX_PCB_NUM;t++){
        if(pcb[t].state==STATE_RUNNING) break;
    }
    inside[t]=1;
    turn=t;
    while((inside[(t+1)%4]==1||inside[(t+2)%4]==1||inside[(t+3)%4]==1)&&turn==t){
        asm volatile("int $0x20");
    }
    int sel = sf->ds; //TODO segment selector for user data, need further modification
    char *str = (char*)sf->edx;

    }
    }
    }
    updateCursor(displayRow, displayCol);
    asm volatile("int $0x20"); //XXX Testing irqTimer during syscall
    //asm volatile("int $0x20:::"memory"); //XXX Testing irqTimer during syscall
}

updateCursor(displayRow, displayCol);
inside[t]=0;
//TODO take care of return value
return;
}
```

最后将当前进程的状态设为 0 即可结束。

四、心得体会

在完成 Fork 这一功能的时候, 知道要去复制每个段寄存器的值, 但是却不知道该如何去实现, 通过阅读初始化代码之后, 发现了 USEL 这个宏定义, 查看 USEL 的宏定义之后发现传入的参数是段选择子的索引, 在初始化代码部分, USEL 的参数是 3 和 4, 我认为这是因为 GDT 表中第 0 个表项保留不使用, 第 1, 2 个表项用于 proc[0], 所以 proc[1] 是从 3, 4 个表项开始; 并且除了代码段寄存器之外, 其他段寄存器的在同一个表项中, 这应该这是由于其他段是重叠的。因此我得出了段寄存器的赋值 USEL 的参数应该是 2i+2 这一方式。