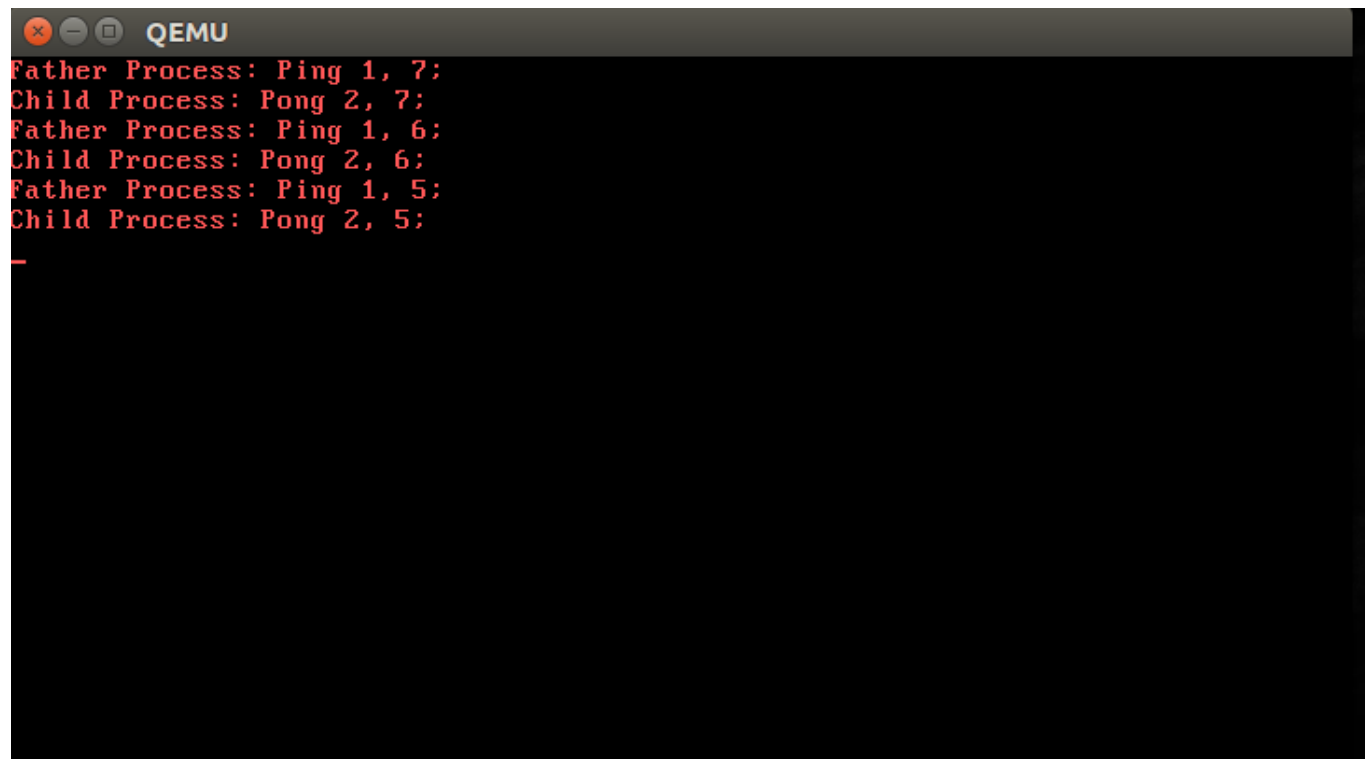


OSlab3 实验报告

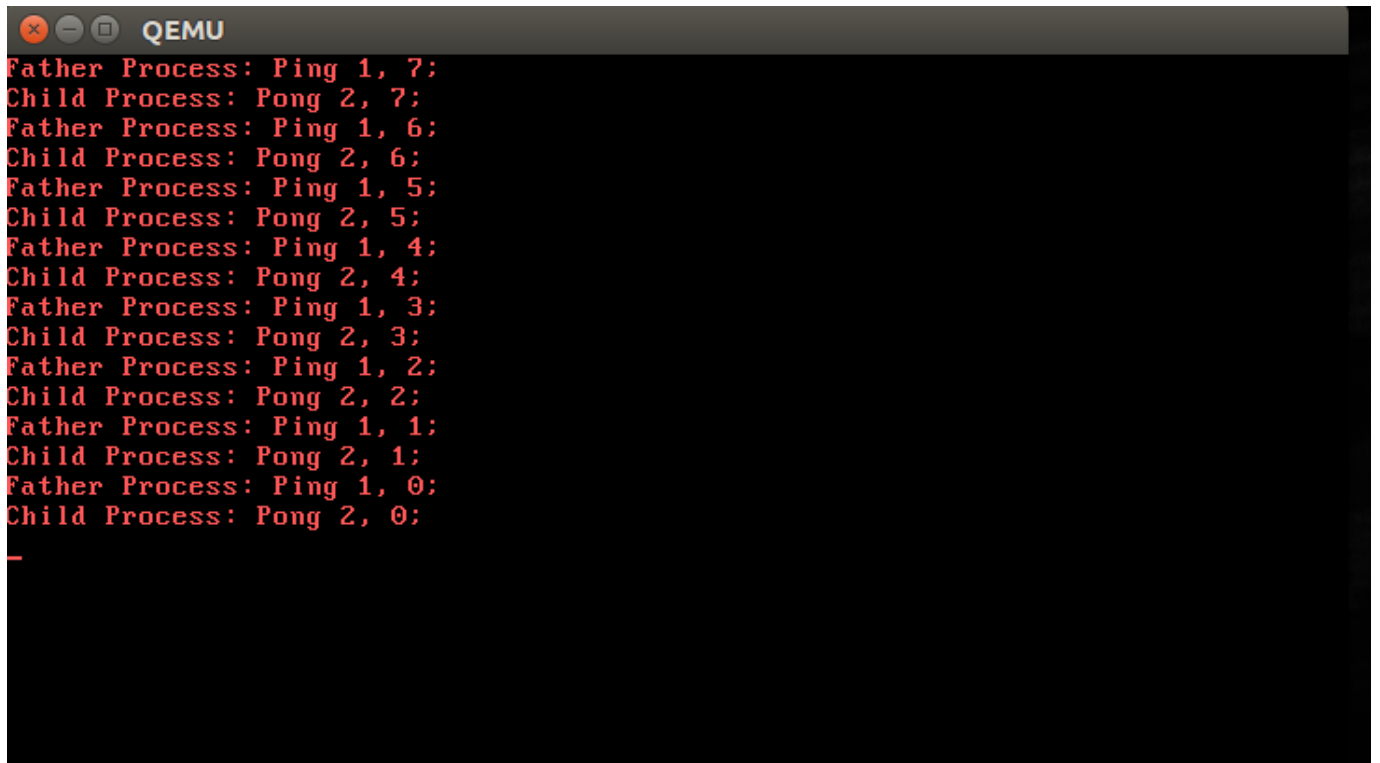
实验基本信息

- 姓名：王旭
- 学号：221220034
- 邮箱：2069625874@qq.com
- 实验进度：已完成本次实验全部内容。

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;
```

A screenshot of a QEMU terminal window. The window title is "QEMU". The terminal output shows a sequence of messages from a "Father Process" and a "Child Process". The Father Process prints "Ping 1, i;" where i ranges from 7 down to 0. The Child Process prints "Pong 2, i;" where i ranges from 7 down to 0. The messages are interleaved, showing that the child process prints its pong message before the father process prints its next ping message, demonstrating a race condition or synchronization issue.

```
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;
```

按照顺序打印父进程、子进程，打出ping pong；

实验内容

sysycall.c文件：

此处增加三个函数的编写

```
pid_t fork()
{
    // TODO:call syscall
    return syscall(SYS_FORK, 0, 0, 0, 0, 0);
}

int sleep(uint32_t time)
{
    // TODO:call syscall
    if (time > 0)
    {
        return syscall(SYS_SLEEP, time, 0, 0, 0, 0);
    }
    else
    {
        return -1;
    }
}

int exit()
{
    // TODO:call syscall
    return syscall(SYS_EXIT, 0, 0, 0, 0, 0);
}
```

其中fork()和exit()是直接调用syscall();

sleep()函数则需要先判断传入参数是否合法。

irqHandle.c文件:

1. 补全 `void syscallHandle(struct StackFrame *sf)` 函数调用, 即增加case 的情况;
2. 编写 `void syscallFork(struct StackFrame *sf)` 函数:

```

void syscallFork(struct StackFrame *sf)
{
    //遍历寻找一个空的PCB
    int new = -1;
    for (int i = 0; i < MAX_PCB_NUM; i++) {
        if(pcb[i].state == STATE_DEAD){
            new = i;
            break;
        }
    }
    //如果没有空的PCB
    if (new == -1) {
        pcb[current].regs.eax = -1;
        return;
    }
    else{
        enableInterrupt(); // 开中断
        int i = 0;
        // 复制内存
        for (i = 0; i < 0x100000; i++) {
            *((uint8_t *)((new + 1) * 0x100000 + i)) = *((uint8_t *) (i + 0x100000 *
(current + 1)));
        }
        disableInterrupt(); // 关中断

        // 复制PCB
        for ( i = 0; i < sizeof(ProcessTable); i++)
        {
            *((uint8_t *)(&pcb[new]) + i) = *((uint8_t *)(&pcb[current]) + i);
        }

        // 改变pcb的信息
        pcb[new].stackTop = (uint32_t)&(pcb[new].regs);
        pcb[new].prevStackTop = (uint32_t)&(pcb[new].stackTop);
        pcb[new].state = STATE_RUNNABLE;
        pcb[new].timeCount = 0;
        pcb[new].sleepTime = 0;
        pcb[new].pid = new;

        // 改变regs的信息
        pcb[new].regs.ss = USEL(2 * new + 2);
        pcb[new].regs.ds = USEL(2 * new + 2);
        pcb[new].regs.es = USEL(2 * new + 2);
        pcb[new].regs.fs = USEL(2 * new + 2);
        pcb[new].regs.gs = USEL(2 * new + 2);
        pcb[new].regs.cs = USEL(2 * new + 1);

        // return value
        pcb[current].regs.eax = new;
        pcb[new].regs.eax = 0;
    }
}

```

```

        return;

    }

}

```

主体逻辑是先判断是否有空的pcb块，若有，则将该块分给fork产生的进程，先将父进程的内存拷贝至子进程，然后进行pcb的全部拷贝，最后修改子进程中pcb的一些信息，使其成为独立、新产生的pcb控制块，最后分别为父进程的eax和子进程的eax赋返回值；如果没有空的pcb，则父进程返回-1。

3. void timerHandle(struct StackFrame *sf) :

我采取了timerhandler 和 sleep、exit分开执行调度程序，这样子可以避免在 void timerHandle(struct StackFrame *sf) 函数中进行多次的分辨究竟调用该函数的是什么操作。

void timerHandle(struct StackFrame *sf) 函数按照正常逻辑编写，先进行遍历pcb看看有没有处于 STATE_BLOCKED 状态的pcb，若有，则sleepTime减一，然后看当前进程是否时间片用完，若是，则考虑更换进程。

本次实验中更换进程使用的是相同的逻辑：

```

int i;
for (i = (current + 1) % MAX_PCB_NUM; i != current; i = (i + 1) % MAX_PCB_NUM)
    if (pcb[i].state == STATE_RUNNABLE )
        break; //寻找可以切换的进程
current = i;
pcb[current].state = STATE_RUNNING;
//进行进程的切换
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");

```

不过在时间中断函数中会增加一些判定条件，比如是否要轮换，以及如果没有进程可以轮换怎么办。

4. void syscallSleep(struct StackFrame *sf) :

将current进程状态改为 `STATE_BLOCKED` , 然后对其sleepime赋值, 在进行切换进程。

5. `void syscallExit(struct StackFrame *sf) :`

进程销毁, pcb块状态改为`STATE_DEAD`, 进行进程切换。

实验心得与想法

1. 手册上建议sleep()、exit()可以通过调用时钟中断处理来实现进程切换, 但我认为这样会让时间中断处理部分冗余, 它需要判断究竟是sleep、exit还是时钟中断引起它的, 这样会比较麻烦, 所以我觉得应该分开实现这三个函数, 其中进程切换可以独立出来。
2. 这次实验中让我困惑的是fork()函数中, 子进程的pcb块哪些是要更新的, 哪些是可以复制父进程pcb的, 仿照kvim.c文件中的写法去悟出来, 我认为框架代码中或者手册中可以将pcb中的量说的再明白些, 手册上的有些抽象。
3. 这次感谢我的室友落同学, 我的代码出了bug, 我调试了许久, 后来他向我提了一些可能的方向、问题, 最终得以解决。