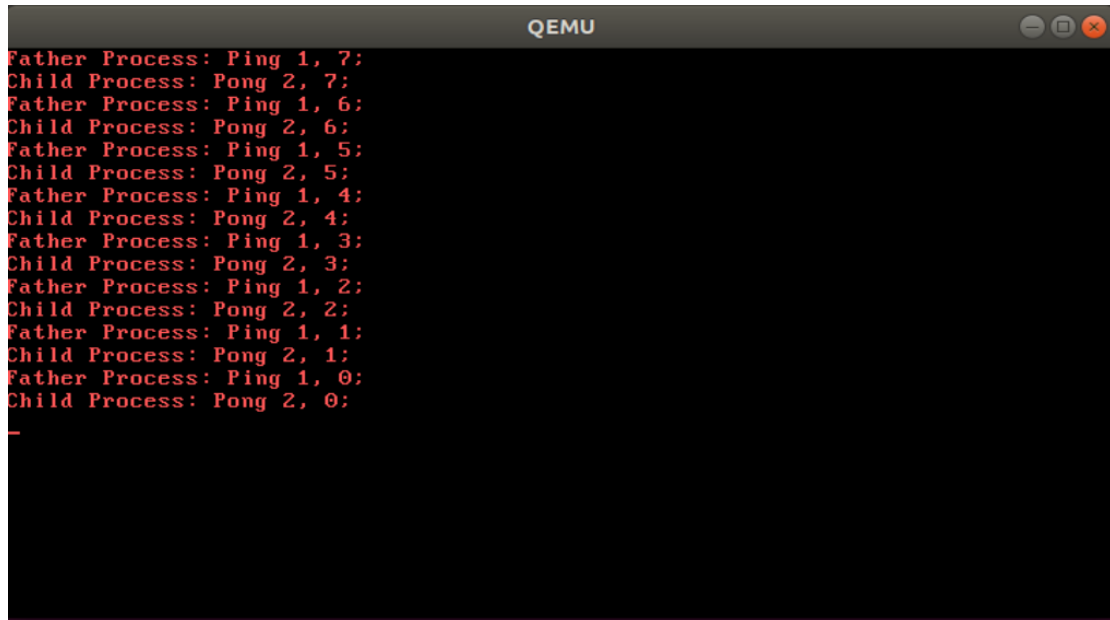


姓名：张涵之 学号：191220154 邮箱：1683762615@qq.com

实验进度：我完成了 3.1-3.3 的必做内容，没有涉及选做内容

实验结果：下图为官方测试代码运行效果



```
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;
```

实验修改的代码位置：

3.1. 完成库函数 lab3/lib/syscall.c

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

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

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

仿照实验 2 调用 syscall 完善库函数

3.2. 时钟中断处理 lab3/kernel/kernel/irqHandle.c

定义时钟中断处理函数 timerHandle，其功能为：

1. 遍历 pcb 中的进程，将状态为 STATE_BLOCKED 的进程的 sleepTime 减一，如果某个进程的 sleepTime 变为 0，其状态重新设为 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;
    }
}
```

2. 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

```

pcb[current].timeCount++;
if (pcb[current].timeCount > MAX_TIME_COUNT) {
    pcb[current].state = STATE_RUNNABLE;
    pcb[current].timeCount = 0;
    for (int i = (current + 1) % MAX_PCB_NUM; i != current;
         i = (i + 1) % MAX_PCB_NUM) {
        if (pcb[i].state == STATE_RUNNABLE) {
            current = i;
            break;
        }
    }
    pcb[current].state = STATE_RUNNING;
    pcb[current].timeCount = 0;
}
}

```

使用手册中提供的参考代码进行进程切换

理解 irqHandle 对比 lab2 增加了部分保存与恢复的内容，其具体功能为在执行系统调用前后分别保存和恢复当前进程的 esp 栈指针位置，并实现内存的分配与回收

3.3. 系统调用例程

1. syscallFork 的实现 lab3/kernel/kernel/irqHandle.c

寻找一个空闲的 pcb 做为子进程的进程控制块

```

int newPcbIndex = -1;
for (int i = 0; i < MAX_PCB_NUM; i++) {
    if (pcb[i].state == STATE_DEAD) {
        newPcbIndex = i;
        break;
    }
}

```

将父进程的资源复制给子进程，代码段和数据段完全拷贝，且需在中断状态下进行复制 pcb 时，先将父进程的内容直接复制，再仿照 initProc 中对 pcb[1] 的初始化进行修改：其中，栈顶指针需根据父进程（已复制）进行设置，运行状态、timeCount、sleepTime 和 pid 与父进程无关，ss、cs 等段寄存器的数值需手动计算

Fork 成功，子进程返回值设为 0，父进程返回值设为子进程的 pid

```

if (newPcbIndex != -1) {
    enableInterrupt();
    for (int j = 0; j < 0x100000; j++)
        *(uint8_t*)(j + (newPcbIndex + 1) * 0x100000) =
            *(uint8_t*)(j + (current + 1) * 0x100000);
    disableInterrupt();
    for (int j = 0; j < sizeof(ProcessTable); j++)
        *((uint8_t*)&pcb[newPcbIndex] + j) =
            *((uint8_t*)&pcb[current] + j);
    pcb[newPcbIndex].stackTop = (uint32_t)&(pcb[newPcbIndex].regs);
    pcb[newPcbIndex].prevStackTop = (uint32_t)&(pcb[newPcbIndex].stackTop);
    pcb[newPcbIndex].state = STATE_RUNNABLE;
    pcb[newPcbIndex].timeCount = 0;
    pcb[newPcbIndex].sleepTime = 0;
    pcb[newPcbIndex].pid = newPcbIndex;
    pcb[newPcbIndex].regs.ss = USEL(2 + 2 * newPcbIndex);
    pcb[newPcbIndex].regs.cs = USEL(1 + 2 * newPcbIndex);
    pcb[newPcbIndex].regs.ds = USEL(2 + 2 * newPcbIndex);
    pcb[newPcbIndex].regs.es = USEL(2 + 2 * newPcbIndex);
    pcb[newPcbIndex].regs.fs = USEL(2 + 2 * newPcbIndex);
    pcb[newPcbIndex].regs.gs = USEL(2 + 2 * newPcbIndex);
    pcb[newPcbIndex].regs.eax = 0;
    pcb[current].regs.eax = newPcbIndex;
}

```

如果没有空闲 pcb，则 fork 失败，父进程返回值设为 -1

```

else
    pcb[current].regs.eax = -1;

```

2. syscallSleep 的实现 lab3/kernel/kernel/irqHandle.c

将当前的进程的 sleepTime 设置为传入的参数, 状态设置为 STATE_BLOCKED, 然后利用 asm volatile("int \$0x20"); 模拟时钟中断, 利用 timerHandle 进行进程切换

```
pcb[current].state = STATE_BLOCKED;
pcb[current].sleepTime = sf->ecx;
pcb[current].timeCount = MAX_TIME_COUNT;
asm volatile("int $0x20");
```

对传入的参数要进行合法性判断, 在这里体现为时间必须是正数

3. syscallExit 的实现 lab3/kernel/kernel/irqHandle.c

将当前进程的状态设置为 STATE_DEAD, 然后模拟时钟中断进行进程切换

```
pcb[current].state = STATE_DEAD;
asm volatile("int $0x20");
```

思考和总结:

因为框架代码和汇编指令比较陌生, 而且文件夹较多, 几乎每次开始写实验的时候都觉得无从下手。大概没有别的捷径, 只能耐心阅读和理解框架代码, 结合手册进行理解。

```
if (ret == 0) {
    data = 2;
    while(i != 0) {
        i--;
        //printf("Child Process: Pong %d, %d;\n", data, i);
        printf("Child Process %d: Sleep\n", i);
        sleep(128);
        //printf("Child Process %d: Wake\n", i);
    }
    //printf("Child Process %d: Exit\n", i);
    exit();
}
else if (ret != -1) {
    data = 1;
    while(i != 0) {
        i--;
        //printf("Father Process: Ping %d, %d;\n", data, i);
        printf("Father Process %d: Sleep\n", i);
        sleep(128);
        //printf("Father Process %d: Wake\n", i);
    }
    //printf("Father Process %d: Exit\n", i);
    exit();
}
```

```
Father Process: Ping 1, 2;
Father Process 2: Sleep
Child Process 3: Wake
Child Process: Pong 2, 2;
Child Process 2: Sleep
Father Process 2: Wake
Father Process: Ping 1, 1;
Father Process 1: Sleep
Child Process 2: Wake
Child Process: Pong 2, 1;
Child Process 1: Sleep
Father Process 1: Wake
Father Process: Ping 1, 0;
Father Process 0: Sleep
Child Process 1: Wake
Child Process: Pong 2, 0;
Child Process 0: Sleep
Father Process 0: Wake
Father Process 0: Exit
Child Process 0: Wake
Child Process 0: Exit
```

利用调试输出语句, 结合用户程序代码可以验证调用顺序符合逻辑。