211300097-3

姓名：童锦
学号：211300097
邮箱：211300097@smail.nju.edu.cn

实验要求

- 完成库函数
- 时钟中断处理
- 系统调用例程
- 选做：中断嵌套
- 选做：临界区

实验情况

实验进度

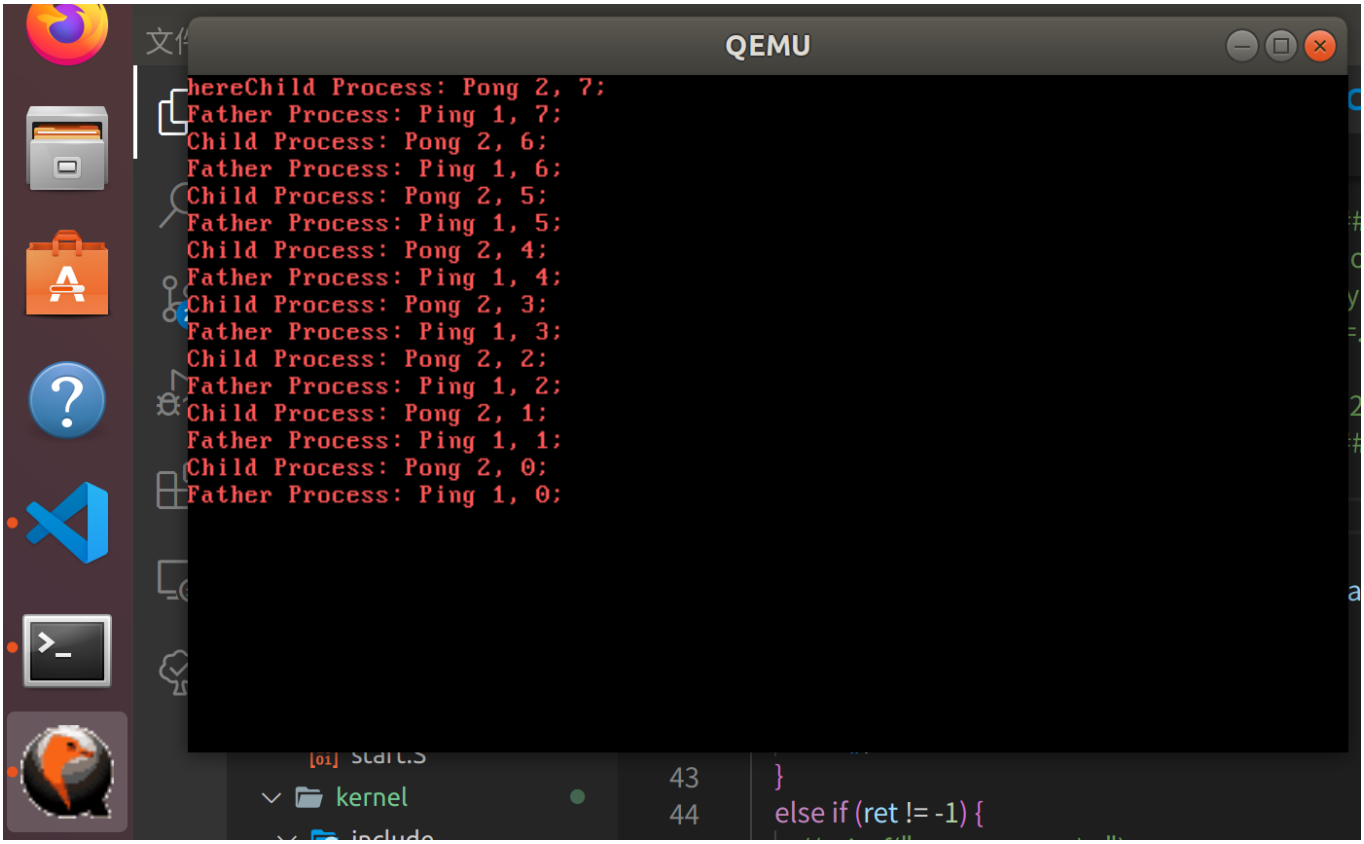
完成了此次实验中的所有任务

代码修改处

此次实验的任务非常集中，所有都是在库函数和系统调用上进行完善：

1. `irqHandle.c`
2. `syscall.c`

结果展示



> **Note**
>
> 接下来的部分为：
> 1.实验心得 – 交代在实验中遇到的困难以及一些思考
> 2.(部分)思考题的回答

实验心得

Task1

> 完成库函数

这一部分比较简单，只需调用封装好的`syscall`，并且根据对应系统调用传递正确的参数即可.

Task2

> 时钟中断处理

其中最主体的函数就是`timerHandle`了，这个函数可以清晰地分为两部分：

1. 遍历`pcb`，自减阻塞进程的`sleepTime`并且将`sleepTime`为0的进程重新设置为`STATE_RUNNABLE`状态．其次，增加当前进程的时间片`timeCount`并且检测是否已经耗尽可执行时间片，如果不是则跳转回用户程序继续执行，如果是就需要考虑进程间的切换；
2. 实现进程间的切换．

对于进程间的切换，pptx里面其实已经提供了有效的代码：

```
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");
```

但起初我对这一段代码的加入是有多处疑惑的，不过经过后续的仔细思考或是实验都解除了疑惑，接下来的部分将一一讨论这些细节：

> 对于第三行代码，为何只设置了`esp0`未设置`ss0`；为何`esp0`设置为`stackTop`(也即之前的`prevStackTop`)．

解答：其实转念一想就很容易明白，因为`ss0`没有必要更换．在本次实验中每个进程的内核栈对应的栈段寄存器都是一样的．第二个问题就稍微复杂一些了，这个问题可以进行分类讨论．我们知道`esp0`作用是在用户态进入内核态时找到内核堆栈的位置，所以如果切换前是内核态(例如在进行打印的时候由于时钟中断切出，此时要切回继续打印)，本就不用在意`esp0`是如何设置的；而对于切换前是用户态的情况，此时设置的`prevStackTop`正是此前有用户态进入内核态时的栈顶(此前将`stackTop`存在了`prevstackTop`之中)．

> 第二行代码的任务是重新设置`stackTop`为`prevStackTop`，可以注意到`irqHandle`函数中最后一行也有相同作用的代码(`pcb[current].stackTop = tmpStackTop`)；除此之外，后面一系列的pop栈内容置寄存器的操作在函数`asmDoIrq`(汇编代码编写)也实现了，为何要在此处多次一举呢?

解答：由于最初觉得这个行为是多此一举的，所以最开始并没有加上这一段代码，但会猛发现其实是无法成功切换进程的，而加上这一段代码后就能神奇地跑通了！所以我开始思考这一段代码的合理性，最后找出了一个个人认为自洽的解释：考虑从IDLE进程切换到`pcb[1]`的过程．在`irqHandle`函数中，在进入时钟中断处理程序之前，我们保存了一个变量`tmpStackTop`，然后在最后将这个值重新设置回`StackTop`．但在从IDLE进程切换到`pcb[1]`后，其实`pcb[1]`对应的内核堆栈中是没有记录到`tmpStackTop`这个变量的，也就无法实现正确地恢复，所以此时需要在时钟中断处理的函数中实现这个设置．那么`tmpStackTop`的存在是否就是毫无意义了呢？答案也是否定的，对于其他的系统调用，其实是需要通过`tmpStackTop`来恢复正确的堆栈位置的．

其余部分就没有太大的理解和实现上的困难了．

Task3

> 系统调用例程

对于`syscallSleep`和`syscallExit`，行为都比较简单：其中前者为设置当前进程的`sleepTime`并将状态修改为`STATE_BLOCKED`；后者的行为是修改状态为`STATE_DEAD`。两者最后都还需要通过`int $0x20`来触发时钟中断切换进程。

这里主要需要讲的是`syscallFork`的实现，这个函数也可以分为两部分来实现：

1. 寻找空闲的`pcb`作为进程控制块，如果没有，直接`eax=-1`，返回。如果有，进入第二阶段的处理；
2. 需要将父进程的地址空间copy到这块空闲的地址空间，然后需要设置子进程的`pcb`，其中某些信息可以继承自父进程，而某些信息需要特殊的设置。

对于用户态空间(主要是代码段，数据段等)，直接copy父进程的地址空间即可。

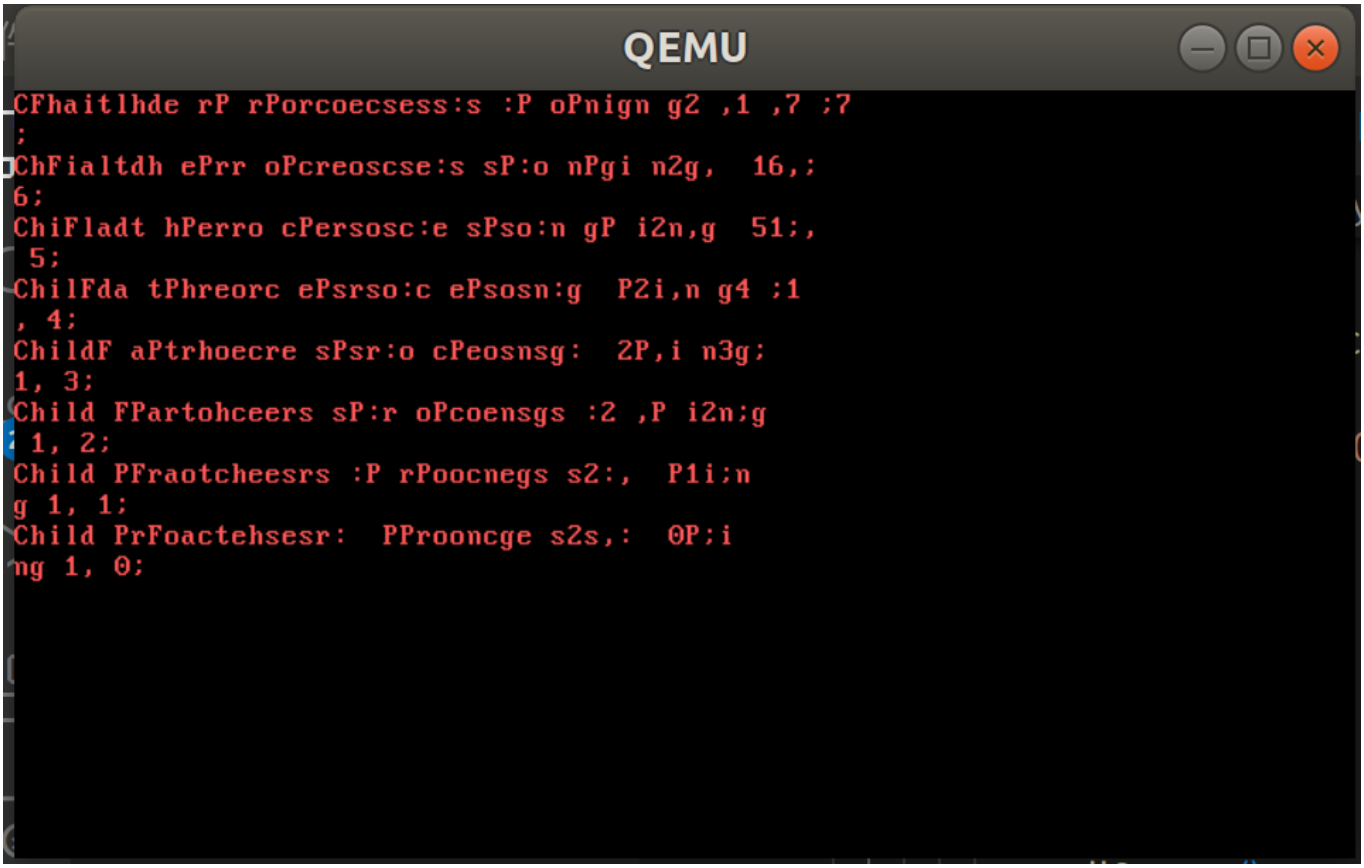
对于`pcb`的设置，就比较有讲究了：

1. 寄存器部分。最初使用`pcb[usable_pcb].regs = pcb[current].regs`的方法来设置，发现并不能通过测试，后来发现这行代码并没有成功将父进程的寄存器信息copy到子进程中，所有的寄存器仍是0。遂只能一行行地进行寄存器的复制。其中有大多数寄存器是可以直接复制的，不过也有些需要额外注意：
    1. `eax`：对应系统调用返回值，对于子进程应该设置为`0`，父进程则设置为子进程的`pid`；
    2. 段寄存器：这一块都需要重新设置，否则在进入用户态是会误访问到父进程的地址空间，造成难以想象的后果。这里的设置只要按`gdt`中的设置分配对应的索引即可：`cs`为`USEL(usable_pcb*2+1)`，其余的为`USEL(usable_pcb*2+2)`；
    3. `esp`：最初其实有个疑问，`esp`设置为和父进程一致能正确访问栈位置吗，最后结合寻址方式发现确实是如此，由于已经在之前正确设置子进程栈段寄存器，相对的便宜保持与父进程一致即可。
2. 其他辅助信息。其中最有讲究的是`prevStackTop`和`prevStackTop`两个变量的设置，最初没考虑清楚的时候以为与`esp`的设置一个意思，直接从父进程copy了过来。后来意识到内核态与用户态的区别和所有`pcb`都存储在内核，解决了这个问题，最初的内核堆栈只需仿照`initProc`中初始化`pcb[1]`的做法即可。

Task4

> 选做：中断嵌套

在Task2中其实已经阐述过这部分的细节了，根据上面的分析我们已经知道讲义提供的代码已经能够成功地完成中断嵌套的任务(不过由于当前还未着手解决临界区的问题，所以输出结果有些诡异...)：



Task5

> 选做：临界区

在Task4我们发现打印的时候存在Race condition的问题，稍加分析就可以发现问题其实出在内核区中的共享变量`displayRow`和`displayCol`上，两个不同的进程都可以在对方未完成一次完整的输出时进入临界区进行自己的输出．所以应该对临界区的访问进行一些限制．

由于处于内核状态中的一次输出任务往往是比较高效的，所以在此处选择~~粗暴地~~在进出临界区时关/开时钟中断，避免Race Condition.

思考题

> Exercise1：linux下进程的创建及运行有两个命令fork和exec，请说明他们的区别？Ps：鼓励自己实现相关代码，并附上关键部分代码的区别。

解答：`fork`命令能够创建一个新的进程，当执行`fork`命令时会搜寻空闲的进程，然后复制一份与父进程几乎一模一样的子进程，但并不能指定运行一个新的程序；而`exec`命令并不能够申请一块新的进程，但是当一个进程执行`exec`命令时，会将目标程序的代码段，数据段等load进地址空间中并覆盖，并修改原先的`pcb`并将pc指向起始位置(例如操作系统提供的`start`函数)开始执行新的程序．
`fork`函数在实验中已经实现，现在主要讲一讲`exec`函数的实现思路：首先，需要指定需要运行的程序(拼接上环境变量找到目标程序的路径)，如果由于各种原因无法执行(如无法找到程序位置等)则返回错误代码；否则根据elf

文件将程序load进当前的地址空间，并且可以仿照`initProc`程序对`pcb`进行初始化，在一切准备工作完成之后，就可以开始执行.

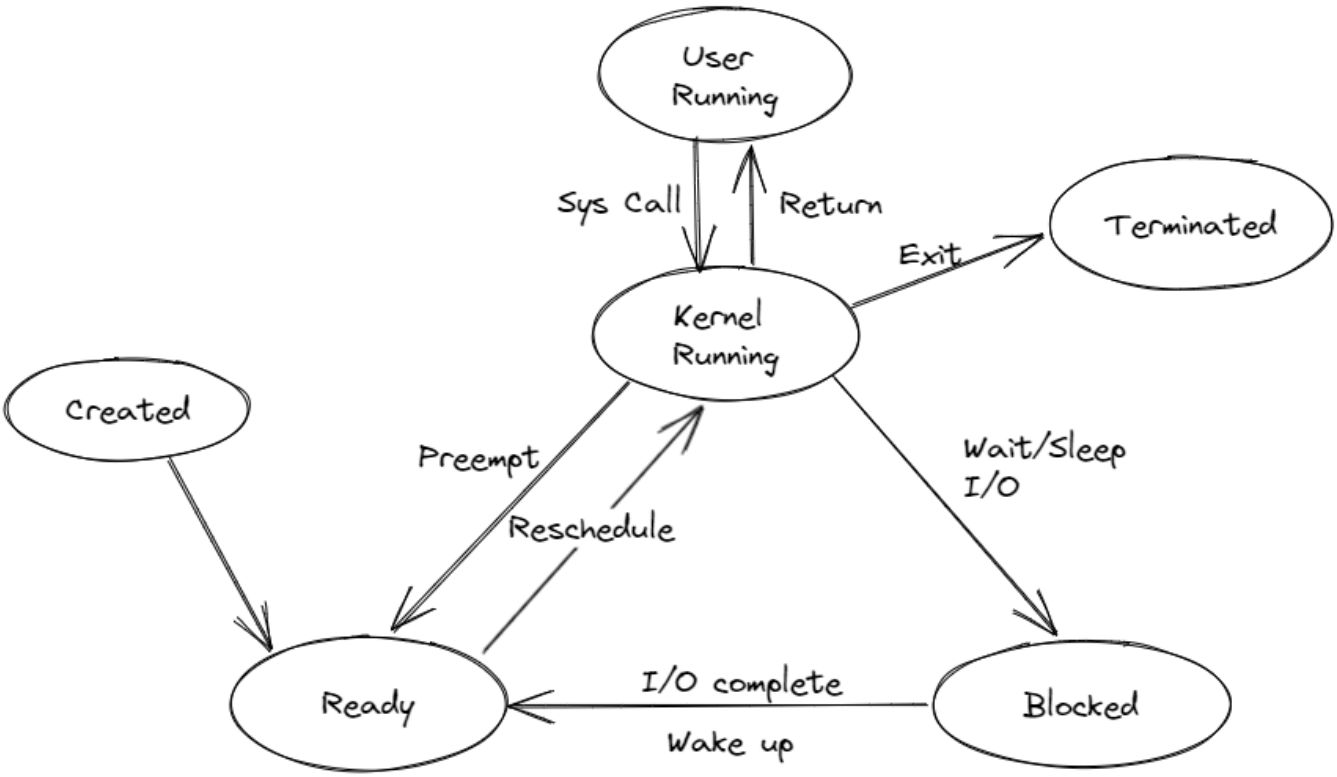
> Exercise2：请在实验报告中简要说明你对 fork/exec/wait/exit函数的分析，并分析fork/exec/wait/exit在实现中是如何影响进程的执行状态的?

解答：
`fork`：申请一块新的地址空间与进程块(创建了一个新的进程)，将父进程几乎原封不动地复制到刚创建的进程中.
影响：进程被创建，指定为Ready状态，加入Ready List;
`exec`：根据elf文件将指定程序装入当前的地址空间并对当前进程的pcb等进程环境进行重新设置. 影响：相当于对原先的进程进行了更新，更新后的进程被指定为Ready状态，加入Ready List;
`wait`：挂起，等待子进程的结束. 影响：在子进程结束之前为Blocked状态，无法运行;
`exit`：退出当前的执行流程，该进程被释放. 如果有父进程通过`wait`等待此释放进程，则提醒该父进程可以继续运行. 影响：进程状态变为Terminated

> Exercise3：请在实验报告中描述当创建一个用户态进程并加载了应用程序后，CPU是如何让这个应用程序最终在用户态执行起来的。即这个用户态进程被OS选择占用CPU执行（RUNNING态）到具体执行应用程序第一条指令的整个经过。

解答：以此次实验的流程为例进行解释. 首先需要将`esp`指向之前准备好的存放寄存器值的区域；此外还需要设置好tss中的`esp0`与`ss0`，以便之后再次从用户态进入内核态时正确跳转；之后根据顺序依次将栈上的值pop置对应的寄存器中，所以执行完之后`pc`将会指向应用程序的第一条指令，此外各个段寄存器也会指向该进程对应的用户段，最后，修改`esp`指向用户栈的栈顶. 至此就完成了执行用户程序的所有环境的准备.

> Exercise4: 请给出一个用户态进程的执行状态生命周期图（包执行状态，执行状态之间的变换关系，以及产生变换的事件或函数调用）



总结

由于在lab2中已经踩过了虚拟机配置的坑，以及经历了从RISC思维向CISC思维转变，再加上在实验中始终贯彻"程序是个状态机"的理念，这次的实验过程相较于上次轻松了许多；除此之外，由于在讲义中已经提供了大量的提示甚至是可以直接copy下来的代码，为代码编写减轻了很多的负担，所以本次实验的主要精力主要用在理解各种概念/机制上和后期的总结复盘/编写实验报告上．再次感谢助教精心准备的讲义，精准地解答了很多实验过程中的困惑．