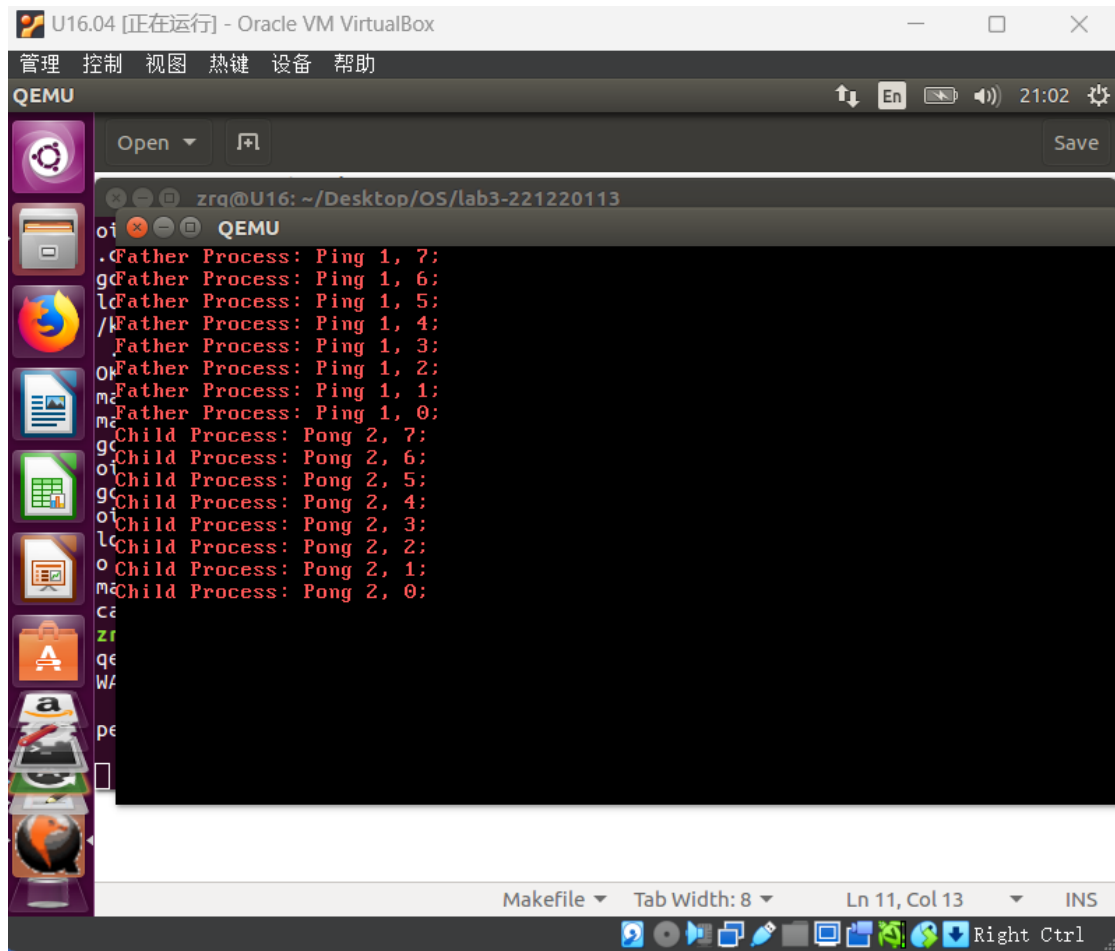


Lab3 实验报告

221220113

首先上实验结果



The screenshot shows a QEMU virtual machine window titled "U16.04 [正在运行] - Oracle VM VirtualBox". The window has a menu bar with "管理", "控制", "视图", "热键", "设备", and "帮助". Below the menu bar is a toolbar with "Open", "Save", and a "QEMU" button. The main area is a terminal window with the title bar "zrq@U16: ~/Desktop/OS/lab3-221220113". The terminal output shows a sequence of ping and pong messages:

```
oi QEMU
Father Process: Ping 1, 7:
gFather Process: Ping 1, 6:
ldFather Process: Ping 1, 5:
/Father Process: Ping 1, 4:
Father Process: Ping 1, 3:
Father Process: Ping 1, 2:
Father Process: Ping 1, 1:
Father Process: Ping 1, 0:
Child Process: Pong 2, 7:
Child Process: Pong 2, 6:
Child Process: Pong 2, 5:
Child Process: Pong 2, 4:
Child Process: Pong 2, 3:
Child Process: Pong 2, 2:
Child Process: Pong 2, 1:
Child Process: Pong 2, 0:
```

The terminal window has a status bar at the bottom showing "Makefile", "Tab Width: 8", "Ln 11, Col 13", and "INS". The bottom of the window shows a taskbar with various icons and a "Right Ctrl" button.

接下来对实验中做的修改进行解释

3.1 完善 syscall

和 lab2 完全一样。

首先在用户接口处填好调用号，然后在 syscallHandle 处调用对应的处理例程即可。

为 fork、sleep、exit 分别分配 1, 2, 3 号。

函数的实现在下面说明。

3.2 时钟中断处理 timerHandle

在这里，对 IDLE 做特殊对待：

除非没有其他进程可选，否则不会选择 IDLE；若当前进程是 IDLE，则无视 sleepTime，总是尝试进行轮转。

根据实验报告，整理出：

pcb[0]是内核 IDLE，1~MAX_PCB_NUM 是用户进程。所以特殊对待 pcb[0]即可。

timerHandle 的基本流程如下：

为所有 STATE_BLOCK 进程减少 sleep 时间，并进行相应的向 STATE_RUNNABLE 转换的处理；

累计当前进程时间片，若需要更换进程或者当前进程是 IDLE：

顺序轮转寻找下一个 STATE_RUNNABLE 进程 next（除非没有其他 STATE_RUNNABLE 进程，否则不会选择 IDLE）

将 current 进程转变为 STATE_RUNNABLE 并将时间片归零

将 current 设为 next

激活 current 进程，即转变为 STATE_RUNNING

另外，转换进程代码的含义解释如下：

```
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");
```

由于第二行，这段代码的指向比较难懂，应该结合另一处 prevStackTop 出现的代码进行理解：

```
void irqHandle(struct TrapFrame *tf) { // pointer tf = esp
    asm volatile("movw %%ax, %%ds"::"a"(KSEL(SEG_KDATA)));
```

```
+   uint32_t tmpStackTop = pcb[current].stackTop;
+   pcb[current].prevStackTop = pcb[current].stackTop;
+   pcb[current].stackTop = (uint32_t)tf;
```

```
+   pcb[current].stackTop = tmpStackTop;
}
```

可以看到，在进行中断处理的时候，将中断的 stackFrame 作为新的 stackTop，而旧的 stackTop 存储于 prevStackTop。

那么 prevStackTop 的作用很显然就是用于中断时，存储用户堆栈信息：因为 stackTop 被占用去存储中断现场（以备中断嵌套）了。

那么可以分两种情况：

1. 要启动的进程未进行系统调用，那么此时 stackTop==prevStackTop，转换进程代码的前两行就可以忽略了，就是在进行：改写 tss，弹出用户栈信息，跳转执行用户程序。
2. 要启动的进程之前被中断嵌套了，那么就激活中断现场，并将用户堆栈信息

弹回 stackTop

3.3.1 syscallFork

首先要找一个 STATE_DEAD 的 pcb 分配给新进程

然后将父进程的一切 (length = 0x100000) 复制给子进程

将父进程 pcb 的一切 (length = sizeof(processTable)) 也复制给子进程 pcb

下面是要区别的地方 (都可以去 init_proc 找到):

```
//开始调整 pcb
pcb[i].stackTop = (uint32_t)&(pcb[i].regs);
pcb[i].prevStackTop = (uint32_t)&(pcb[i].stackTop);
pcb[i].state = STATE_RUNNABLE;
pcb[i].timeCount = 0;
pcb[i].sleepTime = 0;
pcb[i].pid = i;

//esp 不变
//eflags 不变
//eip 不变

pcb[i].regs.cs = USEL(2 * i + 1);
pcb[i].regs.ss = USEL(2 * i + 2);
pcb[i].regs.ds = USEL(2 * i + 2);
pcb[i].regs.es = USEL(2 * i + 2);
pcb[i].regs.fs = USEL(2 * i + 2);
pcb[i].regs.gs = USEL(2 * i + 2);

pcb[i].regs.eax = 0;    //子进程返回 0
pcb[current].regs.eax = i; //主进程返回 i
```

3.3.2 syscallSleep

找到参数: ecx, 根据 ecx 设置 sleep 时间, 转变为 STATE_BLOCKED 后, 复用 timerHandle 即可。

3.3.3 syscallExit

转变为 STATE_DEAD 后, 复用 timerHandle 即可。