操作系统实验

Lab3：

进程管理（基于时间中断进行进程切换）

院系：人工智能学院

姓名：石睿

学号：211300024

班级：操作系统-2023 春季学期

邮箱：211300024@smail.nju.edu.cn

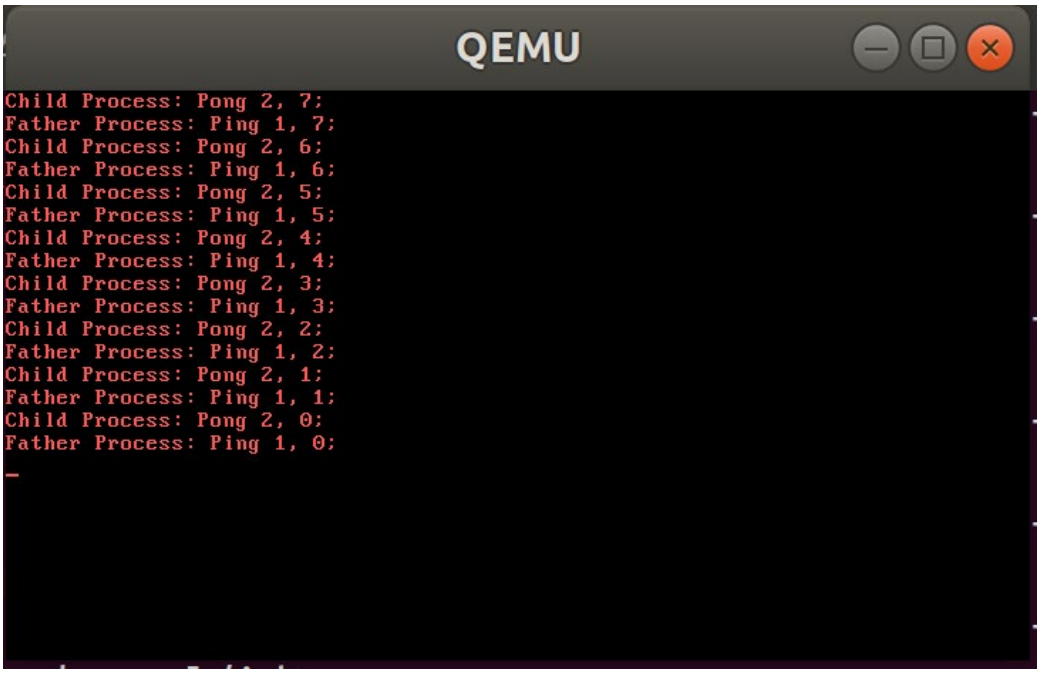
实验时间：2022.4.24

一、 实验进度

我已经完成了 Lab3 的所有内容。

二、 实验结果

结果如下图所示，具体实现过程在第三部分介绍。



补全 fork\sleep\exit，完成系统调用函数，完成时钟中断处理

三、 实验修改的代码位置

① 补全 fork、sleep、exit 函数

首先仿照 fork 填写 sleep 和 exit 即可啦，只需要注意 sleep 系统调用中有一个参数 time，按照 syscall 函数的定义，除了系统调用号（放 eax 中）外，其他的第一个参数放到 ecx 之中。

其次，在 syscallHandle 中补全依系统调用分发不同的系统调用服务例程，依照 lib.h 中定义的系统调用号补全 switch-case 语句块即可啦!

② 完成系统调用函数

在这一步骤，完成 sleep、exit、fork 的系统调用服务例程。

2.1 sleep

Sleep 的功能是让进程进入阻塞态，并赋予这个进程阻塞的时间 sleepTime。最终调用

timeHandle 进行时钟中断，并且在本次 lab 中，timeHandle 还担任着进程调度的功能！

2.2 exit

Exit 让当前进程结束，即修改当前进程的状态。最终也是调用 timeHandle 系统调用进行时钟中断，进行进程切换。

```
253  void syscallSleep(struct StackFrame *sf) {
254      // TODO in lab3
255      //在syscall中，第二个参数在ecx之中。current是pcb数组中当前进程的下标
256      if(sf->ecx >= 0){
257          pcb[current].state=STATE_BLOCKED;
258          pcb[current].sleepTime=sf->ecx;
259          asm volatile("int $0x20");//模拟时钟中断，通过系统调用号0x20进行系统调用，调用timerHandle进行进程切换。调度新的
260      }
261  }
262
263  void syscallExit(struct StackFrame *sf) {
264      // TODO in lab3
265      pcb[current].state=STATE_DEAD;
266      asm volatile("int $0x20");
267
268  }
```

Sleep 和 exit 的系统调用服务例程实现

2.3 fork

Fork 的实现就略显繁琐，fork 的**主要功能**：把父进程的全部地址空间复制到子进程之中，为子进程创建自己的 pcb，并对其中部分信息进行修改（如地址空间等信息）。

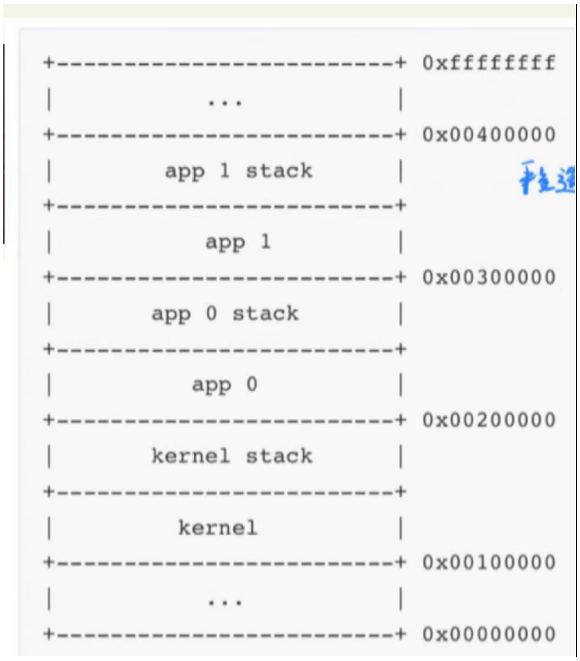
在此过程中**注意**：1. 复制父进程的 pcb 时不能完全复制，如 ss、ds、cs 不能完全 copy！！！；2. eflags 要通过复制而非汇编！！！；3. 要考虑 fork 失败的情况！！！；4. eax 要作为 fork 的返回值对父、子进程单独设置！！！

实现 fork 的主要步骤如下：

step1:从 pcb 数组中找到一个空的位置，给子进程分配【要考虑失败的情况！！！】

step2：把父进程的全部地址空间复制到子进程中去！我写了一个 memcpy 的函数对复制过程进行了封装，在复制父进程的全部地址空间 & 后续整体复制 pcb 的时候可以使用。

step3：设置子进程的 pcb，部分可以拷贝父进程。部分不可以拷贝。在参考了 kvm.c 中的实现，也即使用 USEL 定义的宏，对段寄存器进行初始化。



在 step2 中，通过 memcpy 来完成 fork“把父进程的全部地址空间复制到子进程之中”的功能。在实现的过程中，也要注意 lab3 中采用分段内存管理中的内存分布方式，也如上面左图所示。当前新创建的子进程在 pcb 数组中下标为 child，父进程为 current。所以他们所拥有的的物理地址空间即为(child+1)*0x100000 和(current+1)*0x100000 啦。实现如下。

```
//XXX:step2
//【现在child就是子进程在pcb中的位置啦！】
memcpy((void*)((child+1)*0x100000),(void*)((current+1)*0x100000),0x100000);
```

在 step3 中，需要通过当前子进程在 GDT 中的位置下标来对各个段寄存器进行初始化。其中本 lab 中 GDT 的分布情况如上面右图所示。所以对于新创建在 pcb 中下标为 child 的子进程而言，它的代码段&数据段在 GDT 中的位置下标为：

【代码段】 1（空）+ 2*child 【数据段】1（空）+2*child+1=2+2*child

(原因为段寄存器的 16 位，第三位要给 DPL，高 13 位才是在 GDT 中的下标，所以要通过拼接来产生段寄存器的信息呢！（在宏 USEL 中实现啦！）)。实现如下。

```
241     pcb[child].regs.cs = USEL(1+2*child);//代码
242     pcb[child].regs.ds = USEL(2+2*child);//数据
243     pcb[child].regs.es = USEL(2+2*child);
244     pcb[child].regs.fs = USEL(2+2*child);
245     pcb[child].regs.gs = USEL(2+2*child);
246     pcb[child].regs.ss = USEL(2+2*child);
247     pcb[child].regs.eflags = pcb[current].regs.eflags | 0x200;
248     pcb[child].stackTop = (uint32_t)&(pcb[child].regs);//本进程的上下文信息全在regs之中了
249     pcb[child].prevStackTop = (uint32_t)&(pcb[child].stackTop);//中断嵌套时保存待恢复的栈顶信息（用不到呢！）
250
```

③ 完成时钟中断处理

在这一步骤，完成时钟中断处理函数 timeHandle。它将完成两个任务：

1. 【时间更新】

   1.1 对 STATE_BLOCKED 的进程的 sleepTime 进行--，并在减到 0 的时候转换状态。

   1.2 对 STATE_RUNNING 的进程（如果 pcb[current]是 STATE_RUNNING 的话）的 timeCount 进行++，并在加到 MAX_TIME_COUNT 的时候进行转换状态，并进行进程调度。

2. 【进程调度】

   在所有进程中选择一个 STATE_RUNNABLE 的进程，把它作为下次要被调度的进程（赋值给 current），并通过老师给出的代码进行进程切换【本质上是把构建出来的上下文 pop 出来，恢复要被切换进程的状态，ppt 中有写，这里就不再分析啦！】

本次调度策略**第一种做法**选择 RR 算法，维护一个循环数组，有 rear 和 front 两个下标指针，tag 指示上一次操作是入队（1）还是出队（0）。在 rear==front && tag 的时候队列满。Rear==front&&!tag 的时候队列空。

但是我选择的策略是创建了一个新数组 int ra_list[MAX_PCB_NUM]={1,0,0,0}，这个数组无法进行全局变量的初始化，或在函数切换的过程 ra_list 全被覆盖成 0，之前存储的信息无法被保存！！！【本代码将在实验报告最后给出，也请助教哥哥帮我看看╥﹏╥】。我跟舍友讨论了一下可能和**用户程序装载位置有问题** 或 **硬件切换产生时间中断到保存上下文的过程中对数据段寄存器进行了更改，导致访问数组位置发生变化**。最终这种做法无法产生任何输出，也即一直在 pcb[0]中无法被调度到 pcb[1]之中。

故采用了**第二种做法**，不使用新创建的数组，直接在 pcb 数组中进行选择。只要从当前 current 进程往后遍历找到一个 STATE_RUNNABLE 的进程，就选择它进行调度。这种调度也是可以的，因为本次任务中没有对交互式进程或实时进程，只要最后所有进程都完成了自己的既定任务，也就在正确性上没有错误！。具体实现如下图所示。

```
107     if(pcb[current].state==STATE_RUNNING){//【对运行态的进程操作，也即current】
108         pcb[current].timeCount++;
109         if(pcb[current].timeCount == MAX_TIME_COUNT)
110             return;
111     }
112     for(index=(current+1)%MAX_PCB_NUM; index!=current; index=(index+1)%MAX_PCB_NUM){
113         if(pcb[index].state==STATE_RUNNABLE)//找到一个就绪态的进程为止,更新index!
114             break;
115     }
116     if(pcb[current].state==STATE_RUNNING){//【【【BUG！！！！当前进程可能不是运行态的！！！这一条if判断不能去
117         pcb[current].state=STATE_RUNNABLE;
118         pcb[current].timeCount=0;
119     }
```

四、思考题的看法

EX1：Linux 下进程创建和运行有两个命令 fork 和 exec，他们的区别是什么？

A1：**fork()**创建了一个和调用进程相同的副本 。fork 的工作有以下几步：

1. 在内核创建并初始化一个新的 PCB（给子进程）

2. 创建一个新的地址空间

3. 从父进程的地址空间中把所有的信息都复制到子进程的地址空间中。复制父进程的 data\heap\stack 段，但共享 code 段。tip：从逻辑上，子进程和父进程的内存地址空间相互独立

4. 继承父进程的**执行状态上下文（父进程 PCB 中的部分信息）**，如父进程打开的文件的优先级 tip：尽管地址空间独立，但两者共享其他很多资源：如打开文件的文件描述符、根目录、当前工作目录.....

5. 指示调度器指向新的进程

execve()对当前执行的进程进行替换：修改内存映象，把子进程加载到内存，替换掉用户空间程序并执行新的程序。execve 的工作有以下几步：

1. 把新程序加载到内存中的地址空间【原来在物理内存中的数据区和代码区的信息被新进程替换】

2. 从内存中把参数拷贝到当前内存中的地址空间中

3. 初始化硬件上下文,并开始执行子进程（Linux：从__start 的位置开始）

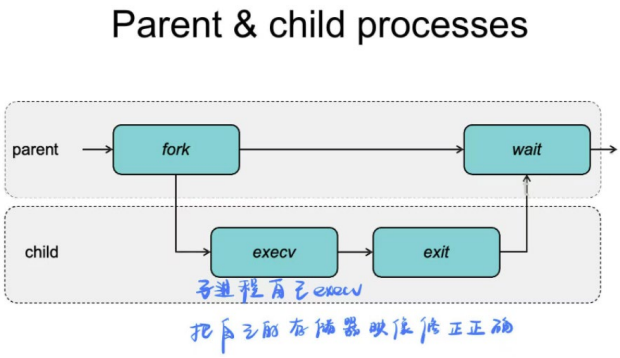
EX2：请在实验报告中说明对 fork\exec\wait\exit 函数的分析，并分析 fork\exec\wait\exit 在实现中是如何影响进程的执行状态的？

A2：已在之前的实验报告中进行说明啦.

Fork->让新的子进程变为 RUNNABLE。Exec->让子进程被调度，变为 RUNNING 态。

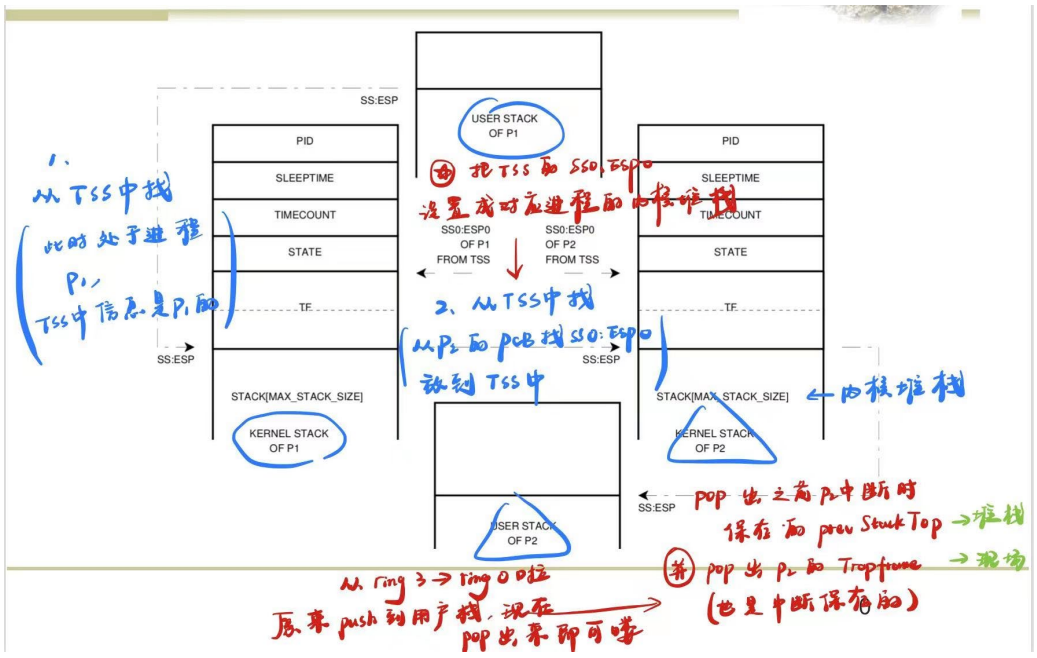
Wait->让当前进程变为 BLOCKED 态。Exit->让当前进程变为 DEAD 态。

下图可以清晰说明，这四个函数在父进程创建子进程中的使用情况。



EX3：描述当创建一个用户态进程（fork）并加载之后（memcpy+bootloader），即从被 OS 选择占用 CPU 执行（RUNNING 态），到具体执行用户程序第一条指令的整个过程。

A3：



当 8259A 硬件产生了一个时间中断，且这个时间中断的时候要进行进程切换，也即进程（上图中的 P1）被 OS 放弃而选择将新进程（P2）时。

Step1：先根据 TSS 中的 ss0 & esp0 找到 P1 的内核栈。此时涉及到特权级的转换，压入用户

栈的 ss 和 sep、eflags、cs、eip。【进内核】

Step2：当前的时间中断会到中断描述符表 IDT 中找到对应的处理程序的入口地址（doirq.s 中的 offset）。因为此时为时间中断，故会去到 doirq.S 中 irqSyscall 压入错误号、中断调用号。并跳转到 amDoIrq 中。

Step3: 在 amDoIrq 中保存段寄存器 ds、es、fs、gs、esp 等。当 push 了 esp 时，除了 StackFrame 作为调用 irqHandle 的参数之外，又多加了一个 esp 作为了参数，也就形成了上一个用户态调用栈的栈顶。

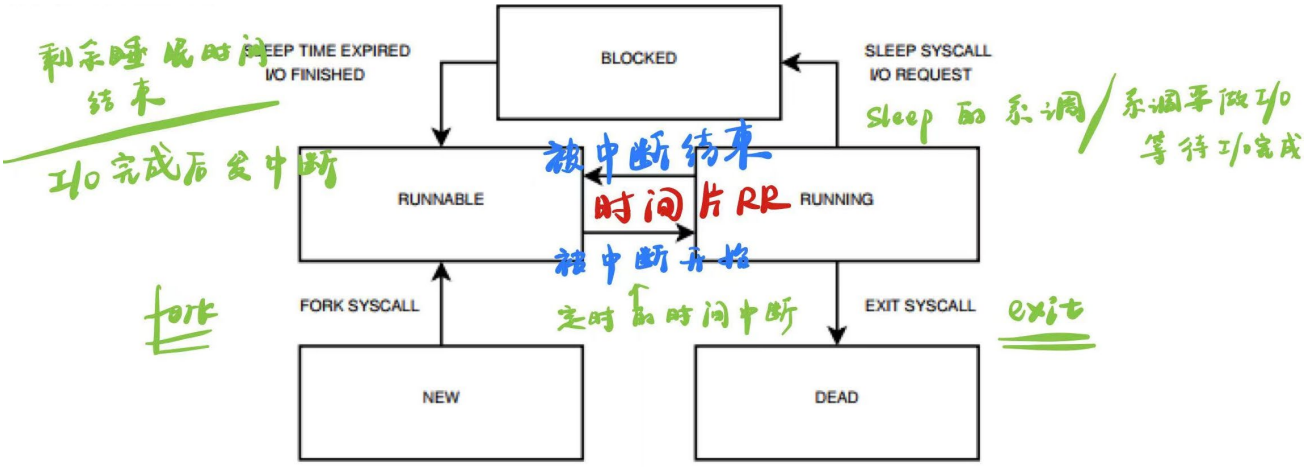
Step4：调用 irqHandle，参数为*sf，也就是*esp。在上面几步中，通过在汇编代码中对栈帧进行压入，隐式地实现了进程 P1 的上下文结构体 StackFrame。【保存 P1 的上下文】

Step5：在 timeHandle 的实现中对进程 P2 的上下文不断 pop，恢复 P2 的栈指针，依次弹出 StackFrame 中的信息。因为 P2 的 StackFrame 中包含它的上一次运行指令的位置 eip，在 pop 的时候也被弹出。故 P2 可以继续运行它的指令啦！【出内核，恢复 P2 的上下文】

EX4：给出用户态进程的执行状态生命周期图

A4：

以下即为五个状态之间的转换，以及引起转换的原因。



附录：【有问题的代码】循环数组实现调度

（我和小伙伴们没有找到逻辑上错误，还请助教哥哥帮忙看一下啦(┬┬ ^ ┬┬)）

（运行结果：只有光标，没有输出。应该还在 pcb[0]中没有调度到 pcb[1]）

Front-下次出队的位置。Rear-下次入队的位置。Tag-上次是入队，tag=1.否则 tag=0

```
static int front=0;
static int rear=1;
static int tag=0;
static int ra_list[MAX_PCB_NUM]={1,0,0,0};//状态为STATE_RUNNABLE进程的下标，用循环数组来模拟链表的
void timerHandle(struct StackFrame *sf) {
    // TODO in lab3
        for(int i=0;i<MAX_PCB_NUM;i++){//【对阻塞态的进程操作】
                if(pcb[i].state==STATE_BLOCKED && pcb[i].sleepTime>1)
                        pcb[i].sleepTime--;
                else if (pcb[i].state==STATE_BLOCKED && pcb[i].sleepTime==1){
                        pcb[i].state=STATE_RUNNABLE;//[阻塞态->就绪态，装入ra_list中！！！]
                        pcb[i].sleepTime=0;
                        if(!(rear==front && tag)){//ra_list队列中没有满
                                tag=1;
                                ra_list[rear]=i;
                                rear=(rear+1)%MAX_PCB_NUM;
                        }
                }
        }
        if(pcb[current].state==STATE_RUNNING && pcb[current].timeCount < MAX_TIME_COUNT )//【对
                pcb[current].timeCount++;
        else {
                //XXX：对老进程进行放入ra_list的操作,只有运行态的才会放入！
                if(pcb[current].state==STATE_RUNNING){
                        pcb[current].state=STATE_RUNNABLE;
                        pcb[current].timeCount=0;//[运行态->就绪态，装入ra_list中！！！！]
                        if(!(rear==front && tag)){//ra_list队列中没有满
                                                tag=1;
                                                ra_list[rear]=current;
                                                rear=(rear+1)%MAX_PCB_NUM;
                                        }
                }
                //XXX：判断有没有就绪态。对current进行更新
                if(!(rear==front&&!tag) ){//当前可运行的队列不是空的！
                        current=ra_list[front];//[就绪态->运行态，从ra_list中拿出来！！！]
                        front=(front+1)%MAX_PCB_NUM;
                        tag=0;
                }
                else{//当前可运行的队列是空的，要来换乘pcb[0]啦！
                        current=0;//换到pcb[0]中呢！
                }
                //XXX：对新进程current进行切换啦！！！
                        pcb[current].state=STATE_RUNNING;
                        pcb[current].timeCount=0;//仿照kvm.c对第一个用户进程初始化的操作呢！
                        pcb[current].sleepTime=0;
                        uint32_t tmpStackTop=pcb[current].stackTop;
                        tss.esp0=(uint32_t)&(pcb[current].stackTop);
                        asm volatile("movl %0,%%esp"::"m"(tmpStackTop));
                        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");
        }
}
```