**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](mailto:211300024@smail.nju.edu.cn)

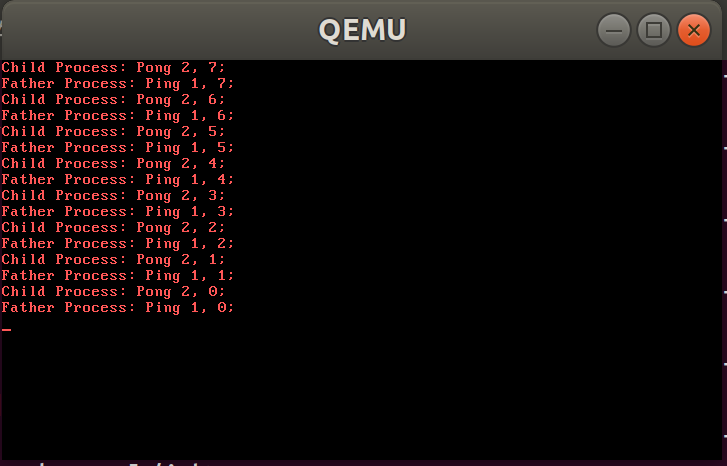
实验时间：2022.4.24

1. **实验进度**

我已经完成了Lab3的所有内容。

1. **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**

1. **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！

* 1. **完成系统调用函数**

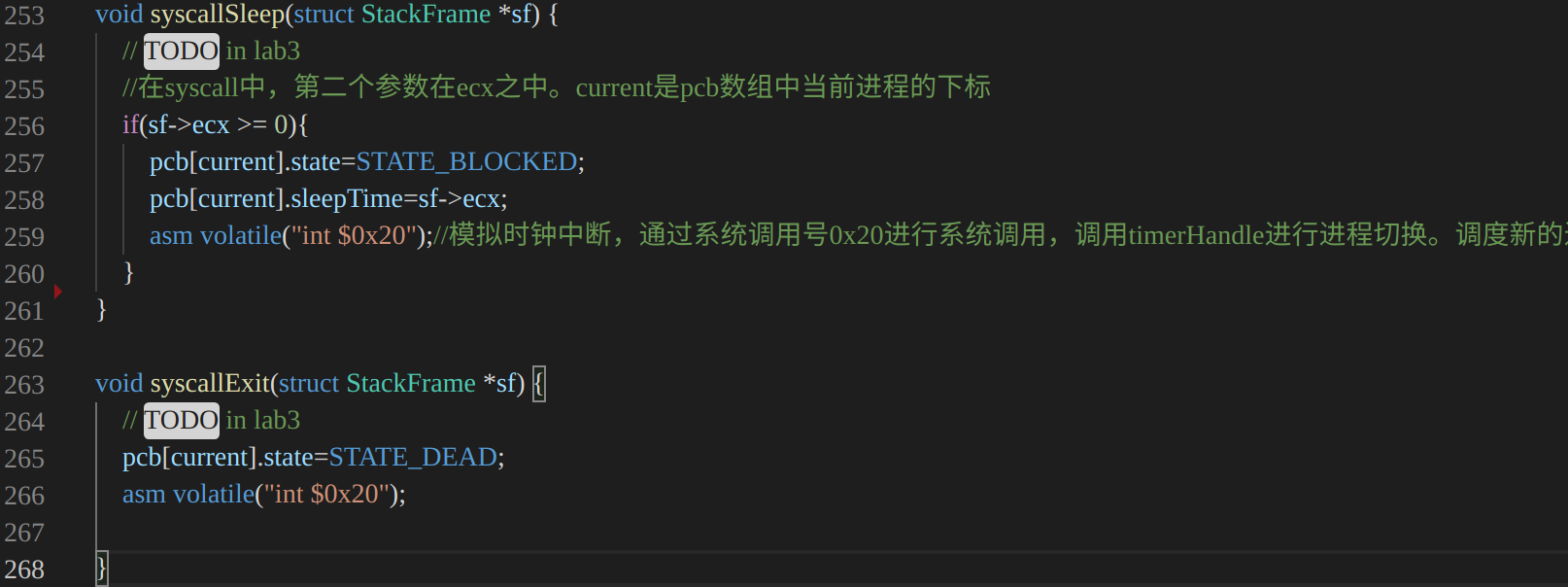
在这一步骤，完成sleep、exit、fork的系统调用服务例程。

**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。

****

**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！！！

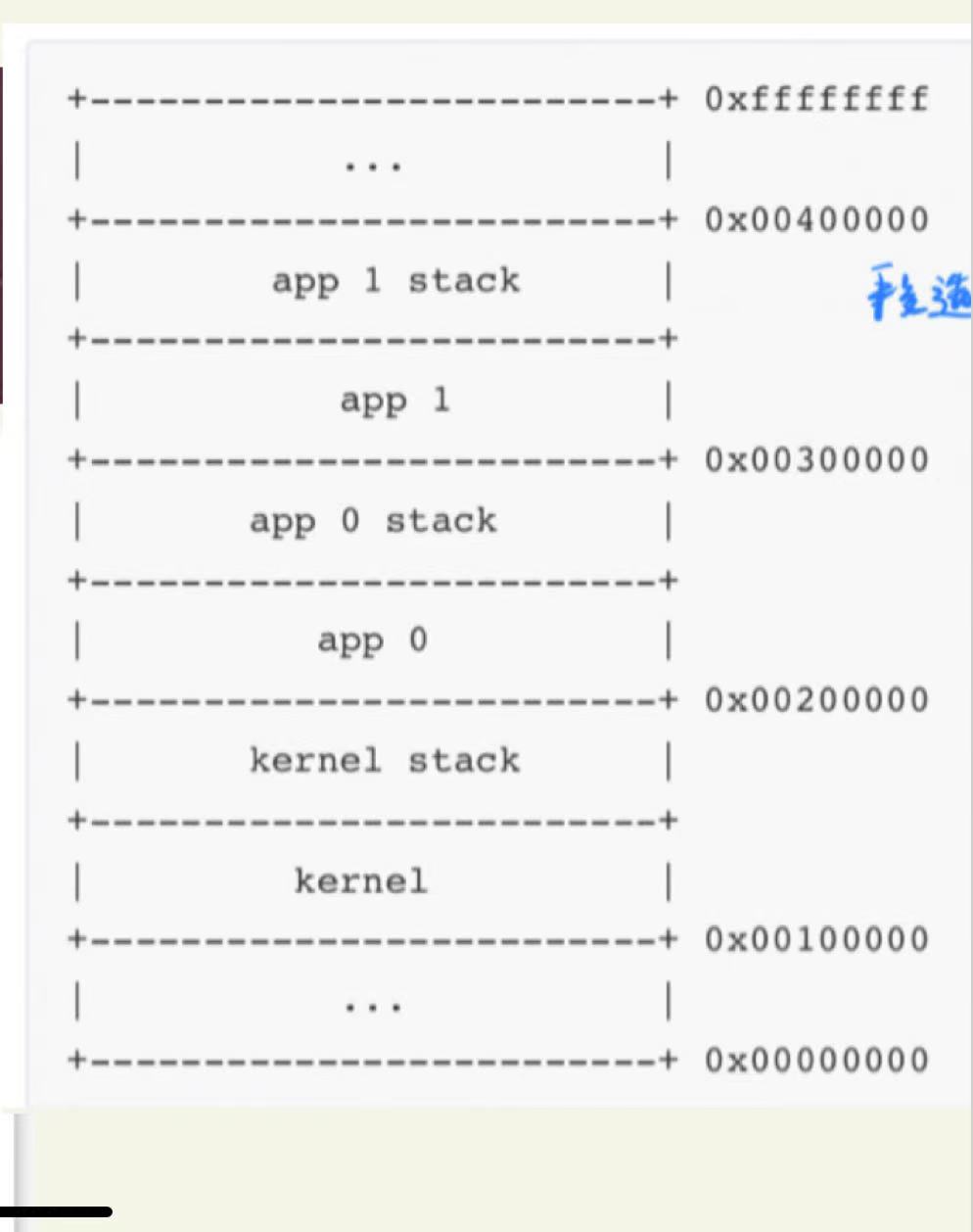
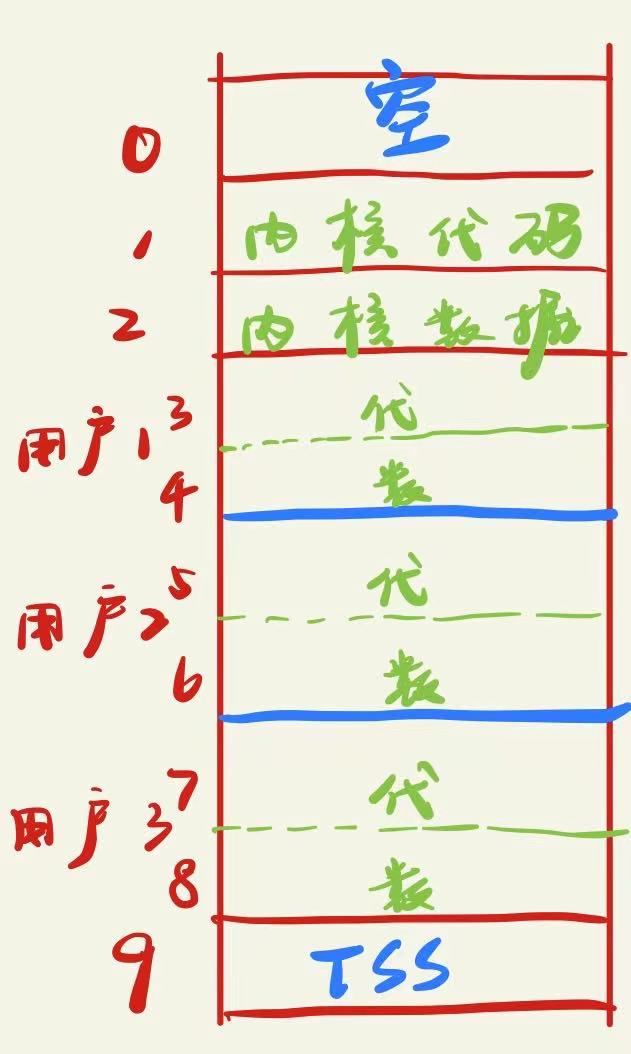
实现fork的主要步骤如下：

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。

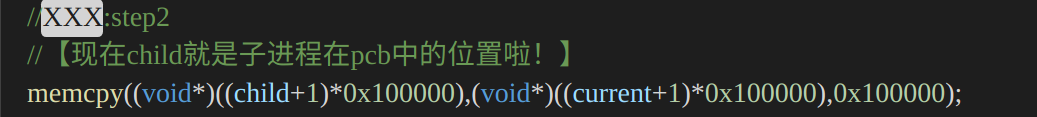
 

在step2中，通过memcpy来完成fork“把父进程的全部地址空间复制到子进程之中”的功能。

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。



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。



* 1. **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。

* + - 1. 【进程调度】

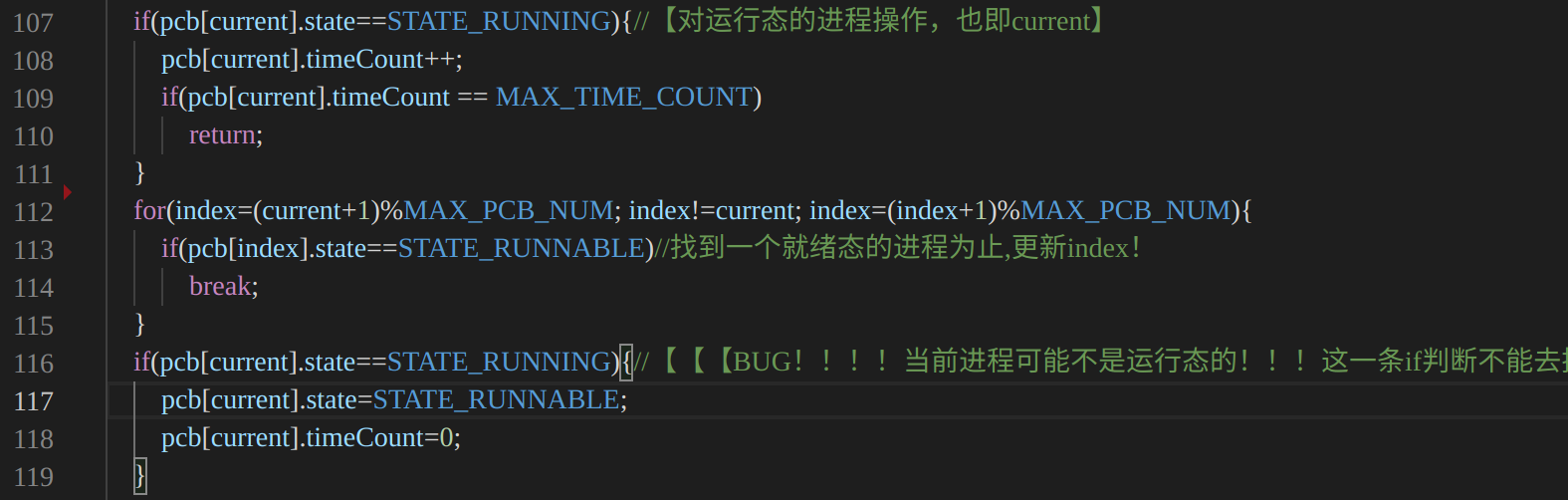
在所有进程中选择一个STATE\_RUNNABLE的进程，把它作为下次要被调度的进程（赋

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。

****

**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）

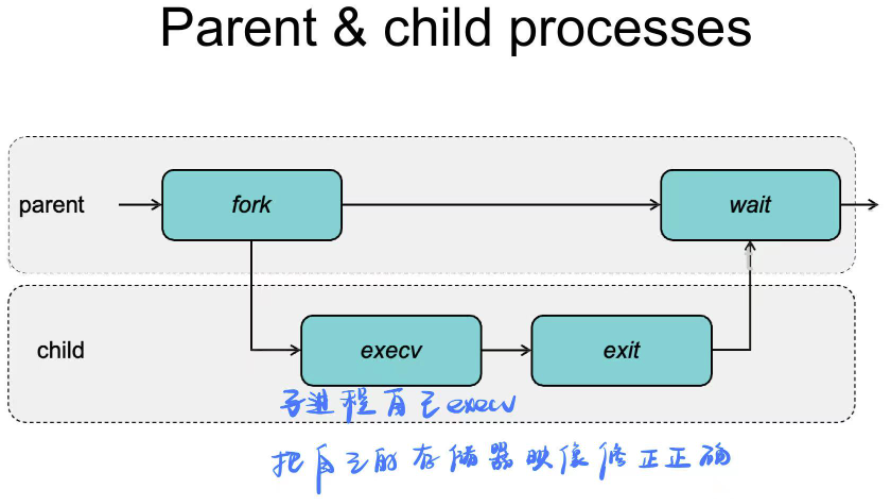
EX2：请在实验报告中说明对fork\exec\wait\exit函数的分析，并分析fork\exec\wait\exit在实现中是如何影响进程的执行状态的？

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。

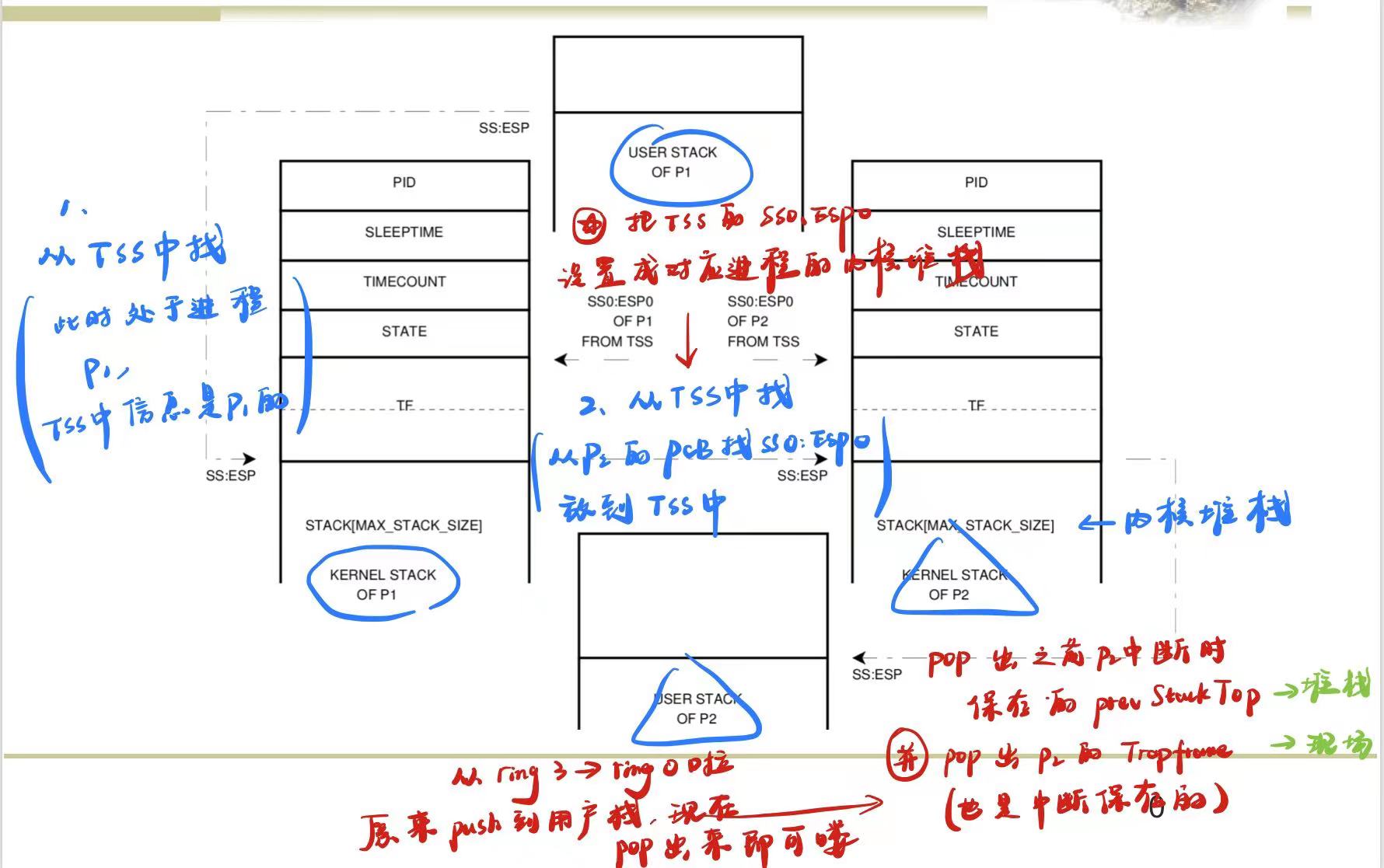
Wait->让当前进程变为BLOCKED态。Exit->让当前进程变为DEAD态。

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。

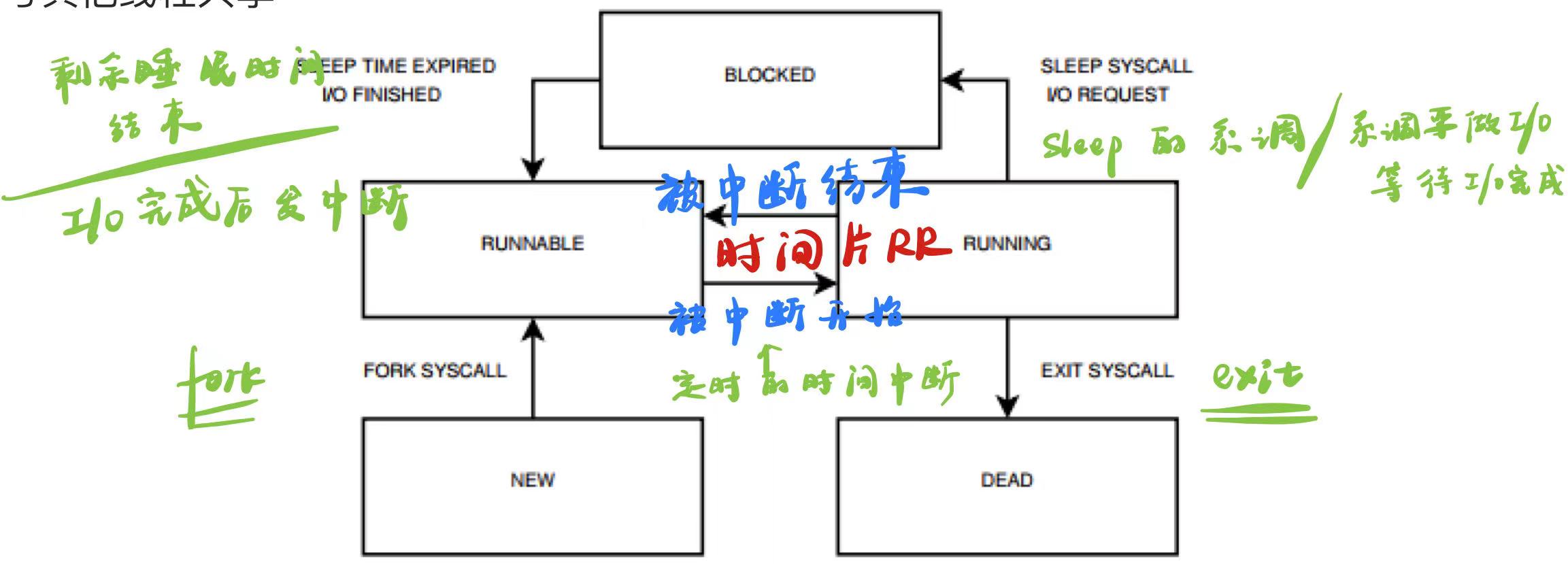
Step4：调用irqHandle，参数为\*sf，也就是\*esp。在上面几步中，通过在汇编代码中对栈帧进行压入，隐式地实现了进程P1的上下文结构体StackFrame。【保存P1的上下文】

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**



