**练习1:分配并初始化一个进程控制块(需要编码)**

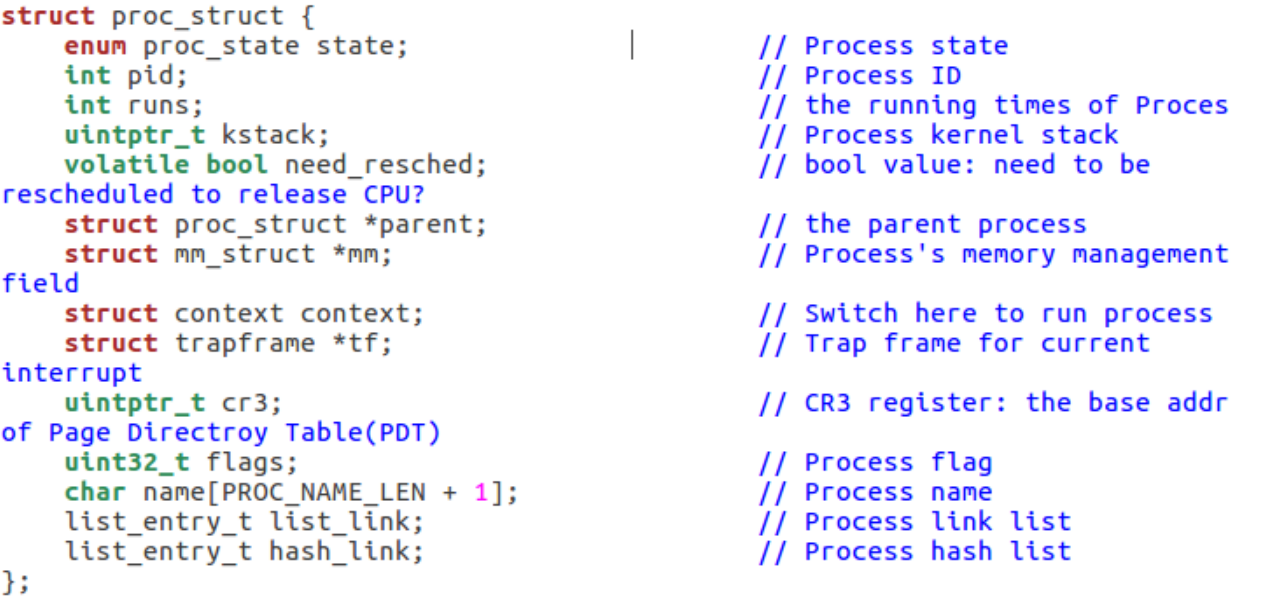
alloc\_proc函数(位于kern/process/proc.c中)负责分配并返回一个新的struct proc\_struct结构,用于存储新建立的内核线程的管理信息。ucore需要对这个结构进行最基本的初始化,你需要完成这个初始化过程。

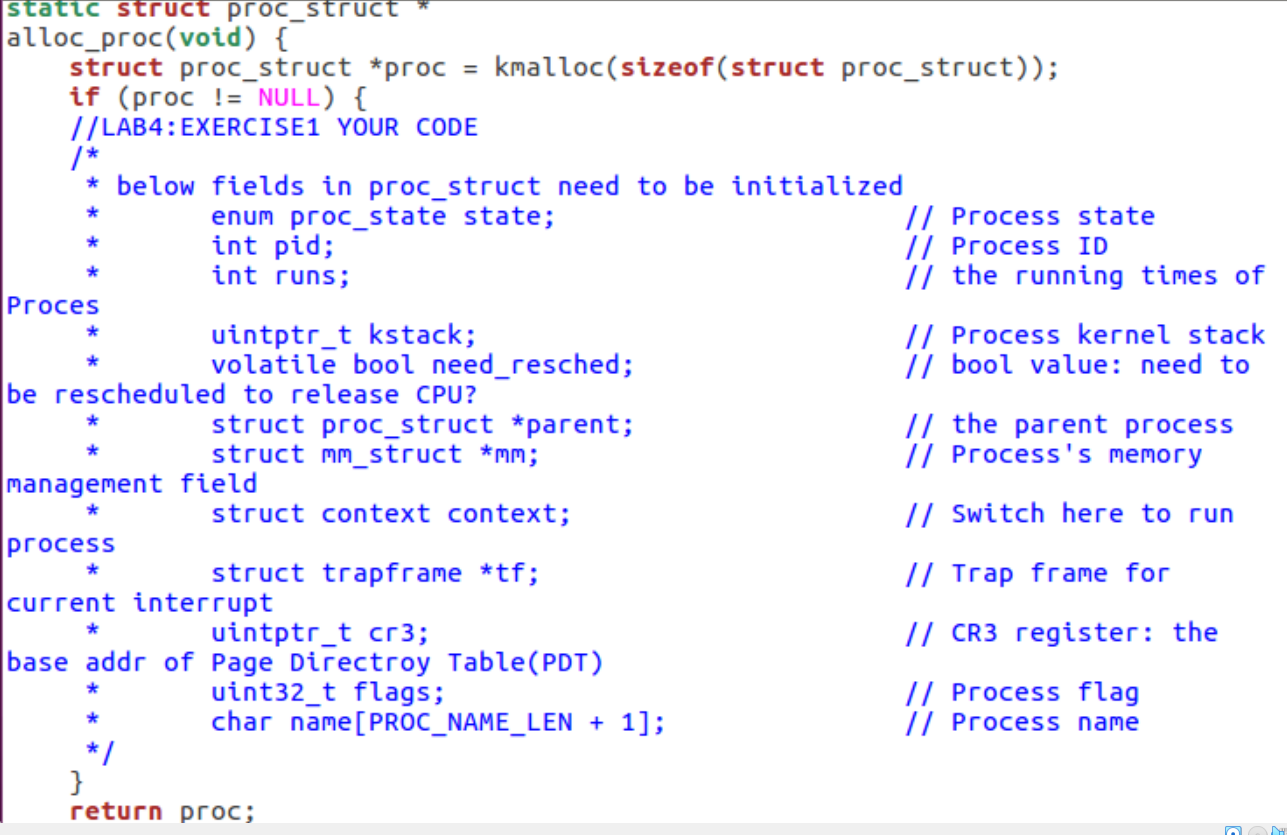
【提示】在alloc\_proc函数的实现中,需要初始化的proc\_struct结构中的成员变量至少括:state/pid/runs/kstack/need\_resched/parent/mm/context/tf/cr3/flags/name。

请在实验报告中简要说明你的设计实现过程。请回答如下问题:

请说明proc\_struct中 struct context context 和 struct trapframe \*tf成员变量含义和在本实验中的作用是啥?(提示通过看代码和编程调试可以判断出来)

首先找到kern/process/proc.c，需要填写的alloc\_proc函数中有这么一段注释，它主要定义了一个结构体proc\_struct，它也说，我们需要初始化这样一个结构体的一个对象并返回它。

看这个结构体里面定义的PCB参数，经过分析我们可以得到以下含义：  
state：进程所处的状态，这个在proc.h有定义，具体如下：  
PROC\_UNINIT //未初始状态  
PROC\_SLEEPING  //睡眠（阻塞）状态  
PROC\_RUNNABLE //运行与就绪态  
PROC\_ZOMBIE //僵尸状态  
**mm：注释说它负责管理进程的虚拟memory，内存管理的信息，包括内存映射列表、页表指针等。   
state：进程所处的状态。   
parent：用户进程的父进程（创建它的进程），这是一个指针变量，记录它的父进程是谁。在所有进程中，只有一个进程没有父进程，就是内核创建的第一个内核线程idleproc。   
kstack：记录了分配给该进程/线程的内核桟的位置。   
need\_resched：是否需要调度   
context：进程的上下文，用于进程切换   
tf：中断帧的指针   
cr3: cr3 保存页表的物理地址  
pid：进程id号。  
runs：进程运行的时间**。



name[PROC\_NAME\_LEN + 1]，这是内核线程（进程）的名称。  
  
 主要的初始化部分如下：  
proc->state = PROC\_UNINIT;  //设置进程为“初始”态    
            proc->pid = -1; //设置进程pid的未初始化值  ，其pid为-1  
            proc->runs = 0;//初始化时间片， 刚刚初始化的进程，运行时间一定为零  
            proc->kstack = 0;//内核栈的地址，为该进程分配的地址为0，因为默认地址都是从0开始的。  
            proc->need\_resched = 0;//是否需要调度 ，刚刚分配出来的进程，都还没有进入CPU，不需要分配  
   
         proc->parent = NULL;//父节点为空 ，父进程，虚拟memory，都是不存在的   
            proc->mm = NULL; //内存管理初始化    
            memset(&(proc->context), 0, sizeof(struct context));//进程上下文初始化    
            proc->tf = NULL; //中断帧指针置为空，总是能够指向中断前的trapframe    
            proc->cr3 = boot\_cr3;//设置内核页目录表的基址    
            proc->flags = 0; //标志位初始化    
            memset(proc->name, 0, PROC\_NAME\_LEN); //进程名初始化

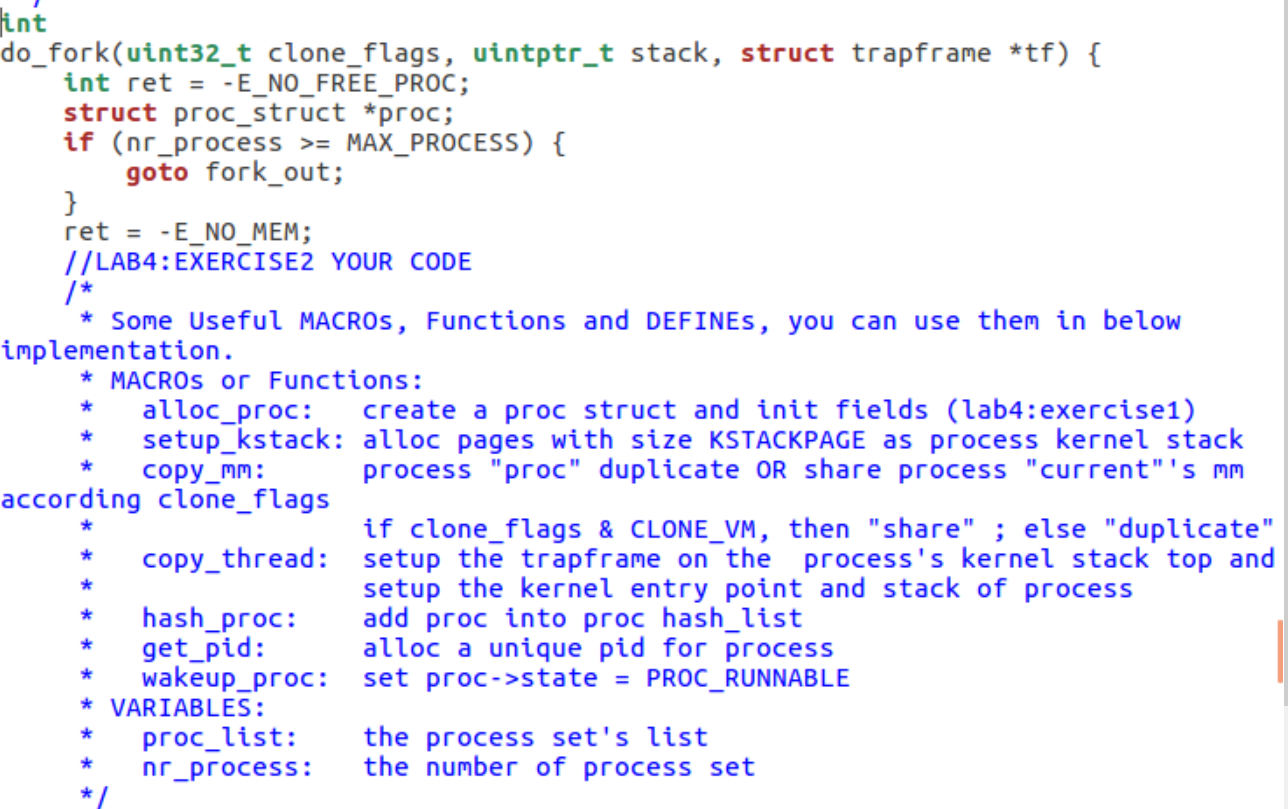
问题：context和\*tf的作用分析  
①context：进程的上下文，用于进程切换。起到的作用就是保存了现场。在 ucore中，所有的进程在内核中也是相对独立的，因此context 保存寄存器的目的就在于在内核态中能够进行上下文之间的切换。实际利用context进行上下文切换的函数是在kern/process/switch.S中定义switch\_to。  
  
② tf：中断帧的指针，总是指向内核栈的某个位置：当进程从用户空间跳到内核空间时，中断帧记录了进程在被中断前的状态。当内核需要跳回用户空间时，需要调整中断帧以恢复让进程继续执行的各寄存器值。除此之外，ucore内核允许嵌套中断。因此为了保证嵌套中断发生时tf 总是能够指向当前的tf，ucore 在内核栈上维护了 tf 的链。

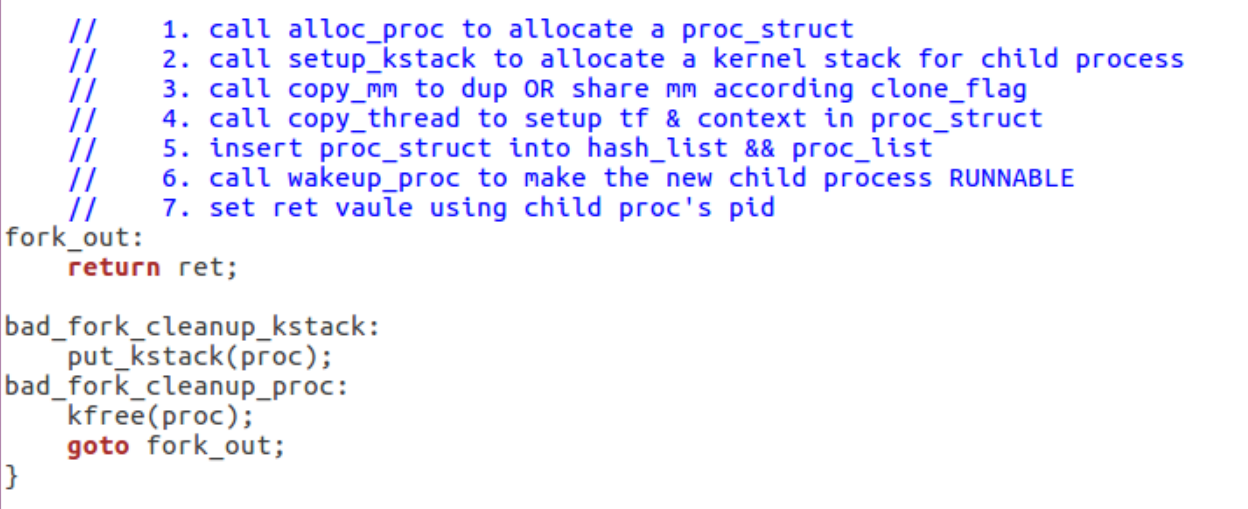
**练习2:为新创建的内核线程分配资源(需要编码)**

完成在kern/process/proc.c中的do\_fork函数中的处理过程。  
 执行步骤  
①调用alloc\_proc，首先获得一块用户信息块。   
②为进程分配一个内核栈。   
③复制原进程的内存管理信息到新进程（但内核线程不必做此事）   
④复制原进程上下文到新进程   
⑤将新进程添加到进程列表   
⑥唤醒新进程   
⑦返回新进程号（设置子进程号为返回值）

请在实验报告中简要说明你的设计实现过程。请回答如下问题:

请说明ucore是否做到给每个新fork的线程一个唯一的id?请说明你的分析和理由





//第一步：调用alloc\_proc()函数申请内存块，  
    if ((proc = alloc\_proc()) == NULL) {  
        goto fork\_out;  
}  
//第二步：将子进程的父节点设置为当前进程，  
proc->parent = current;  
//第三步：调用setup\_stack()函数为进程分配一个内核栈  
    if (setup\_kstack(proc) != 0) {  
        goto bad\_fork\_cleanup\_proc;  
}   
//第四步：调用copy\_mm()函数复制父进程的内存信息到子进程  
    if (copy\_mm(clone\_flags, proc) != 0) {  
        goto bad\_fork\_cleanup\_kstack;  
}  
//第五步：调用copy\_thread()函数复制父进程的中断帧和上下文信息  
copy\_thread(proc, stack, tf);  
//第六步：将新进程添加到进程的（hash）列表中  
    bool intr\_flag;  
local\_intr\_save(intr\_flag);  
    {  
        proc->pid = get\_pid();  
        hash\_proc(proc); //建立映射  
        nr\_process ++;  //进程数加1  
        list\_add(&proc\_list, &(proc->list\_link));//将进程加入到进程的链表中  
    }  
local\_intr\_restore(intr\_flag);  
//步骤七：唤醒子进程  
wakeup\_proc(proc);  
//步骤八：返回子进程的pid  
ret = proc->pid;  
//下面的部分已经给出，不需要自己实现  
fork\_out:  
    return ret；  
bad\_fork\_cleanup\_kstack:  
    put\_kstack(proc);  
bad\_fork\_cleanup\_proc:  
    kfree(proc);  
    goto fork\_out;  
}

问题：

ucore可以做到给每个新fork的线程一个唯一的id。它的做法是在分配PID时设置一个保护锁，暂时不允许中断，这样就可以唯一分配PID了。

**练习3:阅读代码,理解 proc\_run 函数和它调用的函数如何完成**

**进程切换的。(无编码工作)**

请在实验报告中简要说明你对proc\_run函数的分析。并回答如下问题:

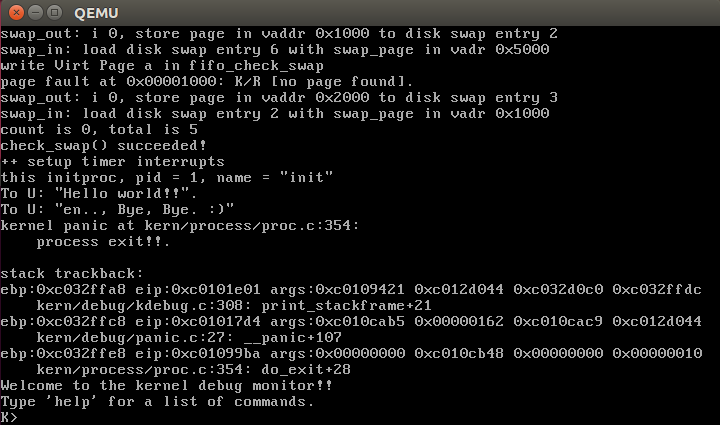
在本实验的执行过程中,创建且运行了几个内核线程?

语句 local\_intr\_save(intr\_flag);....local\_intr\_restore(intr\_flag); 在这里有何作用?请

说明理由

完成代码编写后,编译并运行代码:make qemu

运行make qemu命令，得到的结果如下：



void proc\_run(struct proc\_struct \*proc) {  
    if (proc != current) {  
        bool intr\_flag;  
        struct proc\_struct \*prev = current, \*next = proc;  
        local\_intr\_save(intr\_flag);  
        {  
            current = proc;  
            load\_esp0(next->kstack + KSTACKSIZE);  
            lcr3(next->cr3);  
            switch\_to(&(prev->context), &(next->context));  
        }  
        local\_intr\_restore(intr\_flag);  
    }  
}  
这个函数所做的工作主要就是调配调度，具体逻辑大致如下：  
1、设置当前内核线程 current->need\_resched 为 0（即练习一中的PCB “是否需要调度”）;  
2、遍历进程hash队列，在proc\_list 队列中查找下一个处于就绪态的线程或进程next;（比如，这里有一句：next state=runnable）  
3、找到这样的进程后，就调用 proc\_run函数，保存当前进程current的执行现场(进程上下文)，恢复新进程的执行现场，完成进程切换。  
最后，通过proc\_run函数，就可以跑当前被调度选出的进程，从runable状态正式开始运行。  
  
通过proc\_run和进一步的switch\_to函数完成两个执行现场的切换，具体流程如下：   
①让current指向next内核线程initproc；   
②设置任务状态段ts中特权态0下的栈顶指针esp0为next内核线程initproc的内核栈的栈顶，即next->kstack + KSTACKSIZE ；   
③设置CR3寄存器的值为next内核线程initproc的页目录表起始地址next->cr3，这实际上是完成进程间的页表切换；   
由switch\_to函数完成具体的两个线程的执行现场切换，即切换各个寄存器，当switch\_to函数执行完“ret”指令后，就切换到initproc执行了。

实验总结：   
了解内核线程创建/执行的管理过程  
了解内核线程的切换和基本调度过程