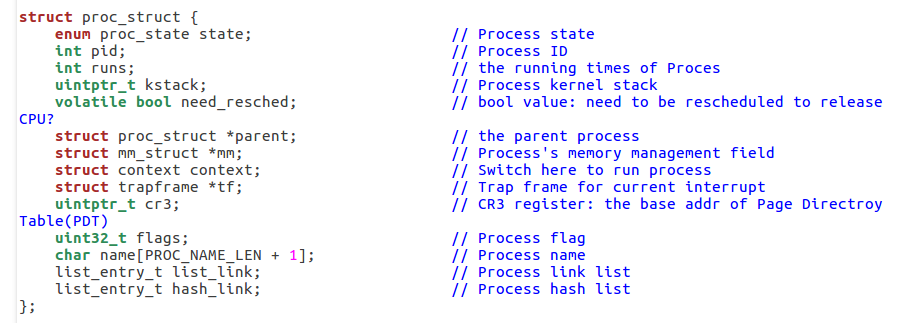
**练习1:分配并初始化一个进程控制块(需要编码)**

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  
  
pid：进程id号。  
runs：进程运行的时间。  
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。  
parent：用户进程的父进程，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。  
mm：注释说它负责管理进程的虚拟memory，其实就是内存管理的信息，包括内存映射列表、页表指针等。  
context：推测应该是进程的上下文，用于进程切换。  
tf：中断帧的指针。  
cr3：记录了当前使用的页表的地址。  
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;  //给进程设置为未初始化状态  
  
proc->pid = -1; //未初始化的进程，其pid为-1  
  
proc->runs = 0; //刚刚初始化的进程，运行时间一定为零   
  
proc->kstack = 0; //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;  //刚刚分配出来的进程，都还没有进入CPU，不需要分配  
proc->parent = NULL;  //父进程，虚拟memory，都是不存在的  
  
proc->mm = NULL;  
  
memset(&(proc->context), 0, sizeof(struct context));  //初始化上下文用  
proc->tf = NULL;  //中断帧指针置为空  
  
proc->cr3 = boot\_cr3;  //页目录设为内核页目录表的基址  
  
proc->flags = 0;  
  
memset(proc->name, 0, PROC\_NAME\_LEN);

**练习2:为新创建的内核线程分配资源(需要编码)**

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。

完整的do\_fork函数如下：  
int do\_fork(uint32\_t clone\_flags, uintptr\_t stack, struct trapframe \*tf) {  
    int ret = -E\_NO\_FREE\_PROC;  
    struct proc\_struct \*proc;  
    if (nr\_process >= MAX\_PROCESS) {  
        goto fork\_out;  
    }  
ret = -E\_NO\_MEM;  
//上面的部分已经给出，不需要自己实现  
//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;  
}

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:

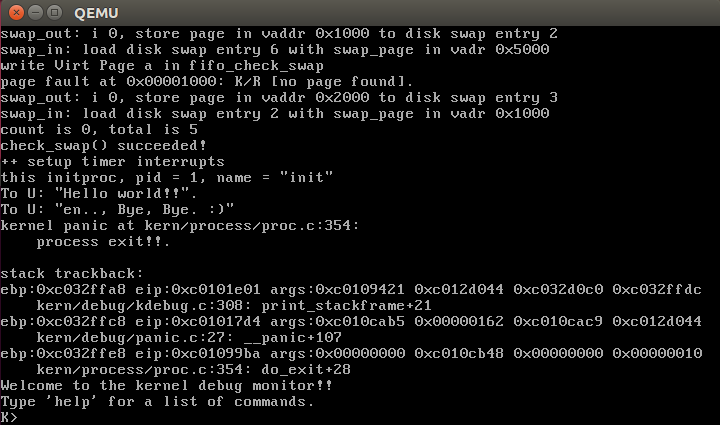
在本实验的执行过程中,创建且运行了几个内核线程?

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：



对proc\_run函数及其调用函数的分析：  
首先，在初始化内核的kern\_init（kern/init/init.c）函数中，调用了cpu\_idle（process/proc.c）；  
在init.c::kern\_init函数调用了proc.c::proc\_init函数。proc\_init函数启动了创建内核线程的步骤。首先当前的执行上下文（从kern\_init 启动至今）就可以看成是uCore内核（也可看做是内核进程）中的一个内核线程的上下文。为此uCore通过给当前执行的上下文分配一个进程控制块以及对它进行相应初始化，将其打造成第0个内核线程 -- idle。  
首先，idleproc = alloc\_proc())，为第一个进程进行分配PCB初始化，那么就有：  
proc->state = PROC\_UNINIT;  
proc->pid = -1;  
proc->cr3 = boot\_cr3;  
//等等这些之前写过的初始化语句，分配给了第一个进程，但是这样不行，第一个进程需要运行，那么此时proc\_init就会对它进行初始化：  
  
    idleproc->pid = 0;  
    idleproc->state = PROC\_RUNNABLE;  
    idleproc->kstack = (uintptr\_t)bootstack;  
    idleproc->need\_resched = 1;  
    set\_proc\_name(idleproc, "idle");  
    nr\_process ++;  
首先设置第一个进程id等于0，当前正在运行，分配了栈空间，需要调度，名字叫做idle。  
第0个内核线程主要工作是完成内核中各个子系统的初始化，所以uCore接下来还需创建其他进程来完成各种工作，但idleproc内核子线程自己不想做，于是就通过调用kernel\_thread函数创建了一个内核线程init\_main。  
然后，在proc\_init()函数完成了 idleproc 和 initproc 内核线程的初始化。  
所以在kern\_init() 最后，它通过 cpu\_idle()唤醒了0号idle 进程,cpu\_idle（proc.c最后）查找到一个需要调度的线程，开始调度它，调用schedule，接着分析调度函数 schedule() 。  
  
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。  
  
最后，proc\_run函数的基本思路是：  
1、让 current指向 next内核线程initproc；  
2、设置任务状态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；  
4、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。

本实验运行过程中，创建并运行了2个内核进程：

Idleproc:第一个内核进程，完成内核中各个子系统的初始化，之后立即调度，执行其他过程。

Initproc:用于完成实验的功能而调度的内核进程。

语句local\_intr\_save(intr\_flag);....local\_intr\_restore(intr\_flag);的作用：屏蔽中断和打开中断，以免进程切换时其他进程再进行调度。