实验四：内核线程管理

一、实验目的

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

二、实验内容

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。

设计实现：

编码工作集中在proc.c中的alloc\_proc函数中，该函数的具体含义为创建一个新的进程控制块，并且对控制块中的变量进行初始化。对于其他成员变量中占用内存空间较大的，可以考虑使用memset函数进行初始化。

代码如下：

proc->state=PROC\_UNINIT;//初始化具有特殊值的成员变量

proc->pid=-1;

proc->runs=0;//对其他变量清零

proc->kstack=0;

proc->need\_resched=0;  
proc->parent = NULL;  
proc->mm=NULL;  
memset(&(proc->context),0,sizeof(struct context));//使用函数清零占用空间较的成员变量

proc->tf=NULL;  
proc->cr3=boot\_cr3;  
proc->flags=0;  
memset(proc->name,0,PROC\_NAME\_LEN);

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）

1.struct context context的作用：  
首先查看struct context结构体的定义，可以发现在结构体中存储除了eax之外的所有通用寄存器以及eip的数值，这个线程控制块中的context很有可能是用于保存线程运行的上下文信息；

接下来使用find grep命令查找在ucore中对context成员变量进行了设置的代码，总共可以发现两处，分别为Swtich.S和proc.c中的copy\_thread函数中，在其他部分均没有发现对context的引用和定义；那么根据Swtich中代码的语义，可以确定context变量有将内核线程之间进行切换的时候，将原先的线程运行的上下文保存下来这一作用

2.struct trapframe \*tf的作用：  
在代码中寻找对tf变量进行了定义的地方，可以发现在copy\_thread函数中对tf进行了设置，在这个函数中，同时对context变量的esp和eip进行了设置，前者设置为tf变量的地址、后者设置为forkret这个函数的指针，观察forkret函数，发现这个函数最终调用了\_\_trapret进行中断返回，tf变量的作用在于在构造出了新的线程的时候，如果要将控制权交给这个线程，是使用中断返回的方式进行的，因此需要构造出一个伪造的中断返回现场，也就是trapframe，使得可以正确地将控制权转交给新的线程；具体切换到新的线程的做法为，调用switch\_to函数，然后在该函数中进行函数返回，直接跳转到forkret函数，最终进行中断返回函数\_\_trapret，之后便可以根据tf中构造的中断返回地址，切换到新的线程了；

练习2：为新创建的内核线程分配资源（需要编码）

创建一个内核线程需要分配和设置好很多资源。kernel\_thread函数通过调用do\_fork函数完成具体内核线程的创建工作。do\_kernel函数会调用alloc\_proc函数来分配并初始化一个进程控制块，但alloc\_proc只是找到了一小块内存用以记录进程的必要信息，并没有实际分配这些资源。ucore一般通过do\_fork实际创建新的内核线程。do\_fork的作用是，创建当前内核线程的一个副本，它们的执行上下文、代码、数据都一样，但是存储位置不同。在这个过程中，需

要给新内核线程分配资源，并且复制原进程的状态。你需要完成在kern/process/proc.c中的do\_fork函数中的处理过程。它的大致执行步骤包括：调用alloc\_proc，首先获得一块用户信息块。

为进程分配一个内核栈。

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）

复制原进程上下文到新进程

将新进程添加到进程列表

唤醒新进程

返回新进程号

请在实验报告中简要说明你的设计实现过程。

设计实现：

主要实现的代码位于proc.c的do\_fork函数中，该函数的语义为为内核线程创建新的线程控制块，并且对控制块中的每个成员变量进行正确的设置，使得之后可以正确切换到对应的线程中执行

代码如下：

proc=alloc\_proc();//为要创建的新的线程分配线程控制块的空间

if(proc==NULL)

goto fork\_out;//判断是否分配到内存空间

assert(setup\_kstack(proc)==0);//为新的线程设置栈

assert(copy\_kstack(proc0==0）；//对虚拟内存空间进行拷贝

copy\_thread(proc,stack,tf）；//在新创建的内核线程的栈上面设置伪造好的中断帧，使于后文中利用iret命令将控制权转移给新的线程

proc->pid=get\_pid();//为新的线程创建pid

hash\_proc(proc);//将线程放入使用hash组织的链表中，使于加速以后对某个指定的现成的查找

nr\_process++;//将全局线程的数目加1

list\_add(&proc\_list,&proc->list\_link);//将线程加入到所有线程的链表中，使于进行调度

wakeup\_proc(proc);//唤醒该进程，即将该进程的状态设置为可以运行

ret=proc->pid;//返回新的线程的pid

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。

可以。ucore为新的fork的线程分配pid的函数为get\_pid,接下来分析该函数内容：

（1）在该函数中使用到了两个静态的局部变量next\_safe和last\_pid，在每次进入get\_pid函数的时候，这两个变量的数值之间的取值均是合法的pid，如果有严格的next\_safe > last\_pid + 1，那就可以直接取last\_pid + 1作为新的pid；  
（2）如果在进入函数的时候，这两个变量之后没有合法的取值，也就是next\_safe > last\_pid + 1不成立，进入循环，在循环之中首先通过if (proc->pid == last\_pid)这一分支确保了不存在任何进程的pid与last\_pid重合，然后再通过if (proc->pid > last\_pid && next\_safe > proc->pid)这一判断语句保证了不存在任何已经存在的pid满足：last\_pid<pid<next\_safe，这样就确保了最后能够找到这么一个满足条件的区间，获得合法的pid；

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。

首先在本实验框架中，唯一调用到这个函数是在线程调度器的schedule函数中，也就是可以推测proc\_run的语义就是将当前的CPU的控制权交给指定的线程。

void proc\_run(struct proc\_struct \*proc) {  
    if (proc != current) { // 判断需要运行的线程是否已经运行着了  
        bool intr\_flag;  
        struct proc\_struct \*prev = current, \*next = proc;  
        local\_intr\_save(intr\_flag); // 关闭中断  
        {  
            current = proc;   
            load\_esp0(next->kstack + KSTACKSIZE); // 设置TSS  
            lcr3(next->cr3); // 修改当前的cr3寄存器成需要运行线程（进程）的页目录表  
            switch\_to(&(prev->context), &(next->context)); // 切换到新的线程  
        }  
        local\_intr\_restore(intr\_flag);  
    }  
}

代码中可以看到proc\_run中首先进行了TSS以及cr3寄存器的设置，然后调用到了swtich\_to函数来切换线程，根据上文中对switch\_to函数的分析可以知道，在调用该函数之后，首先会恢复要运行的线程的上下文，然后由于恢复的上下文中已经将返回地址修改成了forkret函数的地址(，也就是会跳转到这个函数，最后进一步跳转到了\_\_trapsret函数，调用iret最终将控制权切换到新的线程。

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？

总共创建了两个内核线程，分别为：  
idleproc: 最初的内核线程，在完成新的内核线程的创建以及各种初始化工作之后，进入死循环，用于调度其他线程；  
initproc: 被创建用于打印"Hello World"的线程；

语句

local\_intr\_save(intr\_flag);....local\_intr\_restore(intr\_flag); 在这里有何作用?请说明理由

该语句的作用是关闭中断，使得在这个语句块内的内容不会被中断打断，是一个原子操作；  
这就使得某些关键的代码不会被打断，从而不会一起不必要的错误。

完成代码编写后，编译并运行代码：make qemu

