#### **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包括：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）

初始化：

proc->state = PROC\_UNINIT;

proc->pid = -1;

proc->cr3 = boot\_cr3; // 由于是内核线程，共用一个虚拟内存空间

struct proc\_struct \*proc = kmalloc(sizeof(struct proc\_struct)); // 为线程控制块分配空间

if (proc != NULL) {

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

proc->cr3 = boot\_cr3;

proc->pid = -1;

proc->runs = 0; // 对其他成员变量清零处理

proc->kstack = 0;

proc->need\_resched = 0;

proc->parent = NULL;

proc->mm = NULL;

memset(&proc->context, 0, sizeof(struct context)); // 使用memset函数清零占用空间较大的成员变量，如数组，结构体等

proc->tf = NULL;

proc->flags = 0;

memset(proc->name, 0, PROC\_NAME\_LEN);

}

初始化内涵包括lisk\_link、hash\_link以外的所有内容

#### **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。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。

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，在本实验中，每个线程的栈的大小初始均为2个Page, 即8KB assert(copy\_mm(clone\_flags, proc) == 0); // 对虚拟内存空间进行拷贝，由于在本实验中，内核线程之间共享一个虚拟内存空间，因此实际上该函数不需要进行任何操作

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端帧，便于后文中利用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，便于加速以后对某个指定的线程的查找

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，便于进行调度

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

ret = proc->pid; // 返回新线程的pid

可以做到给每个新fork的线程一个唯一的id，ucore中为新的fork的线程分配pid的函数为get\_pid，在该函数中使用到了两个静态的局部变量next\_safe和last\_pid，根据命名推测，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，这样的话，如果有严格的next\_safe > last\_pid + 1，那么就可以直接取last\_pid + 1作为新的pid。

如果在进入函数的时候，这两个变量之后没有合法的取值，也就是说next\_safe > last\_pid + 1不成立，那么进入循环，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。

之所以在该函数中使用了如此曲折的方法，维护一个合法的pid的区间，是为了优化时间效率，如果简单的暴力的话，每次需要枚举所有的pid，并且遍历所有的线程，这就使得时间代价过大，并且不同的调用get\_pid函数的时候不能利用到先前调用这个函数的中间结果。

#### **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

如果可以得到如 附录A所示的显示内容（仅供参考，不是标准答案输出），则基本正确。

代码：

void

Void run(struct proc\_struct \*proc) {

if (proc != current) { // 判断需要运行的线程是否已经运行

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); } }

本实验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；