第三次实验报告

练习一：分配并初始化一个进程控制块

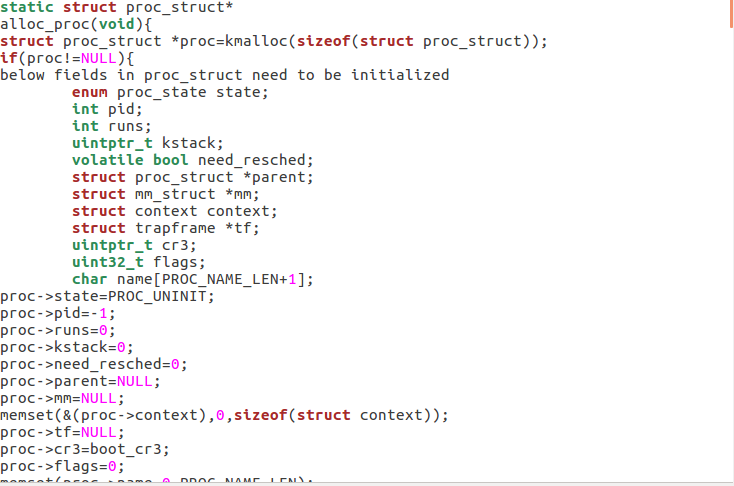
alloc\_proc函数（位于kern/process/proc.c中）负责分配并返回一个新的struct proc\_struct结构，用于存储新建立的内核线程

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。

首先在 kern/process/proc.h 中定义了 PCB，即进程控制块的结构体 proc\_struct，如下： struct proc\_struct { //进程控制块 enum proc\_state state; //进程状态 int pid; //进程 ID int runs; //运行时间

uintptr\_t kstack; //内核栈位置 volatile bool need\_resched; //是否需要调度 struct proc\_struct \*parent; //父进程 struct mm\_struct \*mm; //进程的虚拟内存 struct context context; //进程上下文 struct trapframe \*tf; //当前中断帧的指针 uintptr\_t cr3; //当前页表地址 uint32\_t flags; //进程 char name[PROC\_NAME\_LEN + 1];//进程名字 list\_entry\_t list\_link; //进程链表 list\_entry\_t hash\_link; //进程哈希表 }; 这里简单介绍下各个参数： state：进程所处的状态。 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 // 僵死状态 pid：进程 id 号。 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。 mm：即实验三中的描述进程虚拟内存的结构体 context：进程的上下文，用于进程切换。 tf：中断帧的指针，总是指向内核栈的某个位置。中断帧记录了进程在被中断前 的状态。 cr3：记录了当前使用的页表的地址

实验代码：



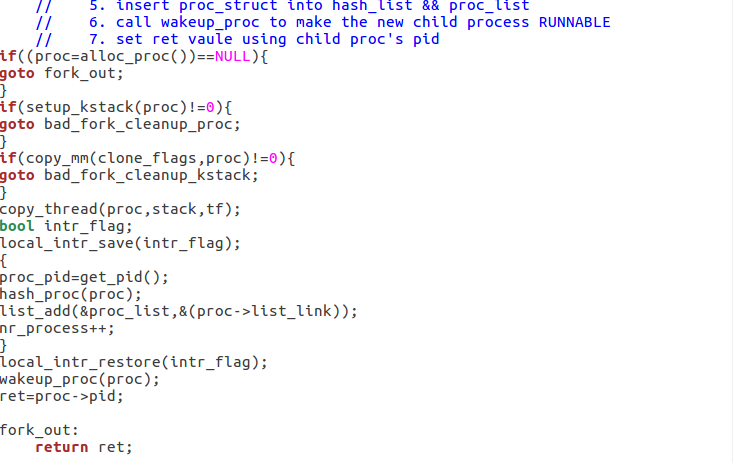
请说明proc\_struct中 struct context context 和 struct trapframe \*tf 成员变量含义和在本实验中的作用是？

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。

练习二：为新创建的内核线程分配资源

alloc\_proc 实质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，而练习 二完成的 do\_fork 才是真正完成了资源分配的工作，当 然，do\_fork 也只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。 根据提示及阅读源码可知，它完成的工作主要如下： 1、分配并初始化进程控制块（ alloc\_proc 函数）; 2、分配并初始化内核栈，为内核进程（线程）建立栈空间（ setup\_stack 函数） ; 3、根据 clone\_flag 标志复制或共享进程内存管理结构（ copy\_mm 函数）; 4、设置进程在内核（将来也包括用户态）正常运行和调度所需的中断帧和执行 上下文 （ copy\_thread 函数）; 5、为进程分配一个 PID（ get\_pid() 函数）; 6、把设置好的进程控制块放入 hash\_list 和 proc\_list 两个全局进程链表中; 7、自此，进程已经准备好执行了，把进程状态设置为“就绪”态; 8、设置返回码为子进程的 PID 号。



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。保证每个 fork 的线程给的 ID 唯一，调用的 get\_pid() 函数，每次都 从进程控制块链表中找到合适的 ID。线程的 PID 由 get\_pid 函数产生，该函 数中包含了两个静态变量 last\_pid 以及 next\_safe。last\_pid 变量保存上一

次分配的 PID，而 next\_safe 和 last\_pid 一起表示一段可以使用的 PID 取值

范围 ， 同 时 要 求 PID 的 取 值 范 围

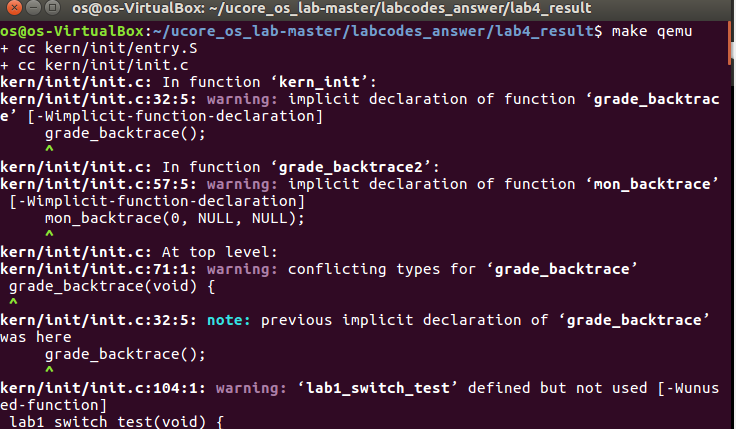
为 ，last\_pid 和 next\_safe 被初始化为 MAX\_PID。每次调

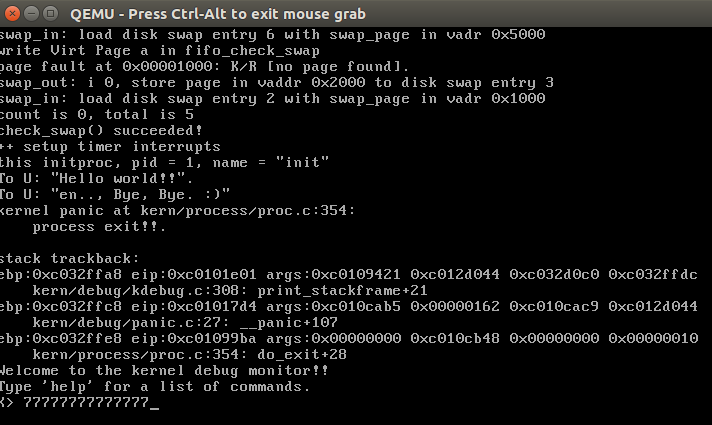
用 get\_pid 时，除了确定一个可以分配的 PID 外，还需要确定 next\_safe 来 实现均摊以此优化时间复杂度，PID 的确定过程中会检查所有进程的 PID，来确 保 PID 是唯一的。

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。

对 proc\_run 函数的具体分析过程： \* 实现思路： 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； 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。

\* 当前进程/线程 切换到 proc 这个进程/线程 \* 注意到在本实验框架中，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； \* 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（ copy\_thread 函数中完成）修改成了 forkret 函数的地址 (如果这个线程是第一运行的话，否则就是切换到这个线程被切换出来的地址)， 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；

在指定文件下输入 make qemu 得到以下结果



在本实验的执行过程中，创建且运行了几个内核线程？

本实验创建且运行了两个内核线程，分别是idle和init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

避免在进程切换过程中处理中断。