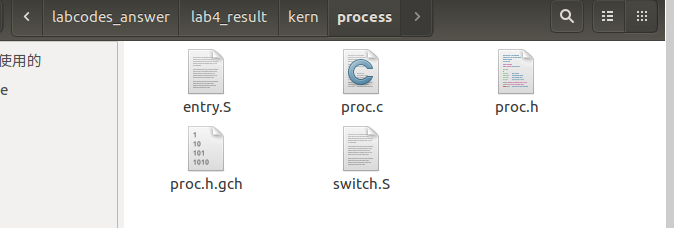
练习**1**：分配并初始化一个进程控制块（需要编码）

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\_structstate/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。



**在kern/process/proc.c中找到proc.c文件 将它根据lab1中的一些头文件 编写出Makefile文件 然后运行（make qemu）**



**初始化参数：**

**static struct proc\_struct \***

**alloc\_proc(void) {**

**struct proc\_struct \*proc = kmalloc(sizeof(struct proc\_struct));**

**if (proc != NULL) {**

**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**

**proc->runs = 0;**

**proc->kstack = 0;**

**proc->need\_resched = 0;**

**proc->parent = NULL;**

**proc->mm = NULL;**

**memset(&(proc->context), 0, sizeof(struct context));**

**proc->tf = NULL;**

**proc->cr3 = boot\_cr3; // 使用内核页目录表的基址**

**proc->flags = 0;**

**memset(proc->name, 0, PROC\_NAME\_LEN + 1);**

**}**

**return proc;**

**}**

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）

**struct context和struct trapframe中有很多寄存器是一样的，前者用于进程上下文切换，后者用于中断上下文切换。注意这两者的含义是不一样的，在本实验中一个进程开始执行需要系统进行初始化，此时tf被用来保存中断帧，而进程执行时是通过context来完成切换的**

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。

**分配并初始化进程控制块（alloc\_proc）**

**分配并初始化内核栈（setup\_stack）**

**根据clone\_flag标志复制或共享内存管理结构（copy\_mm）**

**设置进程在内核正常运行和调度所需要的中断帧和上下文（copy\_thread）**

**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**

**将进程状态设置为“就绪”态**

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;

if ((proc = alloc\_proc()) == NULL) {

goto fork\_out;

}

proc->parent = current;

if (setup\_kstack(proc) != 0) {

goto bad\_fork\_cleanup\_proc;

}

if (copy\_mm(clone\_flags, proc) != 0) {

goto bad\_fork\_cleanup\_kstack;

}

copy\_thread(proc, stack, tf);

bool intr\_flag;

local\_intr\_save(intr\_flag);

{

proc->pid = get\_pid();

hash\_proc(proc);

list\_add(&proc\_list, &(proc->list\_link));

nr\_process ++;

}

local\_intr\_restore(intr\_flag);

wakeup\_proc(proc);

ret = proc->pid;

fork\_out:

return ret;

bad\_fork\_cleanup\_kstack:

put\_kstack(proc);

bad\_fork\_cleanup\_proc:

kfree(proc);

goto fork\_out;

}

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。

**可以。ucore中为新的fork的线程分配pid的函数为get\_pid，接下来不妨分析该函数的内容:**

**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（也就是说没有被使用过），这样的话，如果有严格的next\_safe > last\_pid + 1，那么久可以直接取last\_pid + 1作为新的pid（需要last\_pid没有超出MAX\_PID从而变成1）；**

**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：

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);

}

}

**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

**该语句的左右是关闭中断，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；**

**比如说在proc\_run函数中，将current指向了要切换到的线程，但是此时还没有真正将控制权转移过去，如果在这个时候出现中断打断这些操作，就会出现current中保存的并不是正在运行的线程的中断控制块，从而出现错误；**

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。

