练习1：分配并初始化一个进程控制块（需要编码）

所有内核线程直接使用共同 ucore 内核内存空间，不需为每个内核线程维护单独的内存空间，而用户进程需要维护各自的用户内存空间。

这里主要是从 kern\_init 函数的物理内存管理初始化开始的，按照函数的次序做了一个简单的总结：

1、pmm\_init()

(1) 初始化物理内存管理器。

(2) 初始化空闲页，主要是初始化物理页的 Page 数据结构，以及建立页目录表和页表。

(3) 初始化 boot\_cr3 使之指向了 ucore 内核虚拟空间的页目录表首地址，即一级页表的起始物理地址。

(4) 初始化第一个页表 boot\_pgdir。

(5) 初始化 GDT，即全局描述符表。

2、pic\_init()

初始化 8259A 中断控制器

3、idt\_init()

初始化 IDT，即中断描述符表

4、vmm\_init()

主要就是实验了一个 do\_pgfault() 函数达到页错误异常处理功能，以及虚拟内存相关的 mm,vma 结构数据的创建/销毁/查找/插入等函数

5、proc\_init()

这个函数启动了创建内核线程的步骤，完成了 idleproc 内核线程和 initproc 内核线程的创建或复制工作。

6、ide\_init()

完成对用于页换入换出的硬盘(简称 swap 硬盘)的初始化工作

7、swap\_init()

swap\_init() 函数首先建立完成页面替换过程的主要功能模块，即 swap\_manager，其中包含了页面置换算法的实现

进程控制块的初始化。

而这里我们分配的是一个内核线程的 PCB，它通常只是内核中的一小段代码或者函数，没有用户空间。而由于在操作系统启动后，已经对整个核心内存空间进行了管理，通过设置页表建立了核心虚拟空间(即 boot\_cr3 指向的二级页表描述的空间)。所以内核中的所有线程都不需要再建立各自的页表，只需共享这个核心虚拟空间就可以访问整个物理内存了。

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

};

alloc\_proc 函数来负责分配一个新的 struct proc\_struct 结构，根据提示我们需要初始化一些变量，具体的代码如下：

\* 实现思路：

该函数的具体含义为创建一个新的进程控制块，并且对控制块中的所有成员变量进行初始化，根据实验指导书中的要求，除了指定的若干个成员变量之外，其他成员变量均初始化为0，取特殊值的成员变量如下所示：

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

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。

/\*code\*/

static struct proc\_struct \*alloc\_proc(void) {

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

if (proc != NULL) {

proc->state = PROC\_UNINIT; //设置进程为未初始化状态

proc->pid = -1; //未初始化的的进程id为-1

proc->runs = 0; //初始化时间片

proc->kstack = 0; //内存栈的地址

proc->need\_resched = 0; //是否需要调度设为不需要

proc->parent = NULL; //父节点设为空

proc->mm = NULL; //虚拟内存设为空

memset(&(proc->context), 0, sizeof(struct context));//上下文的初始化

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; //标志位

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

}

return proc;

}

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）

在 context 中保存着各种寄存器的内容，主要保存了前一个进程的现场（各个寄存器的状态），是进程切换的上下文内容，这是为了保存进程上下文，用于进程切换，为进程调度做准备。

在 ucore 中，所有的进程在内核中也是相对独立的。使用 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 函数。

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（跟lab1中切换特权级类似的技巧），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。

trapframe 保存着用于特权级转换的栈 esp 寄存器，当进程发生特权级转换的时候，中断帧记录了进入中断时任务的上下文。当退出中断时恢复环境。

tf 是一个中断帧的指针，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 trapframe，ucore 在内核栈上维护了 tf 的链。

内核态和用户态的转换首先是留下 SS 和 ESP 的位置，然后调用中断，改中断栈里面的内容， 然后退出中断的时候跳到内核态中，最后将 ebp 赋给 esp 修复 esp 的位置。

练习2：为新创建的内核线程分配资源（需要编码）

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 2 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。

实现过程如下：

\* 实现思路：

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。

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

/\* do\_fork - parent process for a new child process

\* @clone\_flags: used to guide how to clone the child process

\* @stack: the parent's user stack pointer. if stack==0, It means to fork a kernel thread.

\* @tf: the trapframe info, which will be copied to child process's proc->tf

\*/

int do\_fork(uint32\_t clone\_flags, uintptr\_t stack, struct trapframe \*tf) {

int ret = -E\_NO\_FREE\_PROC; //尝试为进程分配内存

struct proc\_struct \*proc; //定义新进程

if (nr\_process >= MAX\_PROCESS) { //分配进程数大于 4096，返回

goto fork\_out; //返回

}

ret = -E\_NO\_MEM; //因内存不足而分配失败

//LAB4:EXERCISE2 YOUR CODE

/\*

\* Some Useful MACROs, Functions and DEFINEs, you can use them in below implementation.

\* MACROs or Functions:

\* alloc\_proc: create a proc struct and init fields (lab4:exercise1)

\* setup\_kstack: alloc pages with size KSTACKPAGE as process kernel stack

\* copy\_mm: process "proc" duplicate OR share process "current"'s mm according clone\_flags

\* if clone\_flags & CLONE\_VM, then "share" ; else "duplicate"

\* copy\_thread: setup the trapframe on the process's kernel stack top and

\* setup the kernel entry point and stack of process

\* hash\_proc: add proc into proc hash\_list

\* get\_pid: alloc a unique pid for process

\* wakeup\_proc: set proc->state = PROC\_RUNNABLE

\* VARIABLES:

\* proc\_list: the process set's list

\* nr\_process: the number of process set

\*/

// 1. call alloc\_proc to allocate a proc\_struct

// 2. call setup\_kstack to allocate a kernel stack for child process

// 3. call copy\_mm to dup OR share mm according clone\_flag

// 4. call copy\_thread to setup tf & context in proc\_struct

// 5. insert proc\_struct into hash\_list && proc\_list

// 6. call wakeup\_proc to make the new child process RUNNABLE

// 7. set ret vaule using child proc's pid

if ((proc = alloc\_proc()) == NULL) { //调用 alloc\_proc() 函数申请内存块，如果失败，直接返回处理

goto fork\_out;//返回

}

proc->parent = current; //将子进程的父节点设置为当前进程

if (setup\_kstack(proc) != 0) { //调用 setup\_stack() 函数为进程分配一个内核栈

goto bad\_fork\_cleanup\_proc; //返回

}

if (copy\_mm(clone\_flags, proc) != 0) { //调用 copy\_mm() 函数复制父进程的内存信息到子进程

goto bad\_fork\_cleanup\_kstack; //返回

}

copy\_thread(proc, stack, tf); //调用 copy\_thread() 函数复制父进程的中断帧和上下文信息

//将新进程添加到进程的 hash 列表中

bool intr\_flag;

local\_intr\_save(intr\_flag); //屏蔽中断，intr\_flag 置为 1

{

proc->pid = get\_pid(); //获取当前进程 PID

hash\_proc(proc); //建立 hash 映射

list\_add(&proc\_list, &(proc->list\_link)); //将进程加入到进程的链表中

nr\_process ++; //进程数加 1

}

local\_intr\_restore(intr\_flag); //恢复中断

wakeup\_proc(proc); //一切就绪，唤醒子进程

ret = proc->pid; //返回子进程的 pid

fork\_out: //已分配进程数大于 4096

return ret;

bad\_fork\_cleanup\_kstack: //分配内核栈失败

put\_kstack(proc);

bad\_fork\_cleanup\_proc:

kfree(proc);

goto fork\_out;

}

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 [https://camo.githubusercontent.com/a0bd3566194ca6278916be90f4da5bcb2998f39c/687474703a2f2f6c617465782e636f6465636f67732e636f6d2f6769662e6c617465783f2532386c6173742532302535435f2532307069642532432532306e6578742532302535435f25323073616665253239](https://camo.githubusercontent.com/a0bd3566194ca6278916be90f4da5bcb2998f39c/687474703a2f2f6c617465782e636f6465636f67732e636f6d2f6769662e6c617465783f2532386c6173742532302535435f2532307069642532432532306e6578742532302535435f25323073616665253239)， 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。（无编码工作）

在 kern\_init() 最后，它通过 cpu\_idle() 唤醒了 0 号 idle 进程，在分析 proc\_run 函数之前，我们先分析调度函数 schedule() 。

可以看到 ucore 实现的是 FIFO 调度算法：

1、调度开始时，先屏蔽中断，设置当前内核线程 current->need\_resched 为 0。

2、在进程链表中，查找第一个可以被调度的程序，即在 proc\_list 队列中查找下一个处于就绪态的线程或进程 next。

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，运行新进程，允许中断，完成进程切换。

即 schedule 函数通过查找 proc\_list 进程队列，在这里只能找到一个处于就绪态的 initproc 内核线程。于是通过 proc\_run 和进一步的 switch\_to 函数完成两个执行现场的切换。

再分析 switch\_to 函数

\* 实现思路：

switch\_to 函数主要完成的是进程的上下文切换，先保存当前寄存器的值，然后再将下一进程的上下文信息保存到对于寄存器中。

1. 首先，保存前一个进程的执行现场，即 movl 4(%esp), %eax 和 popl 0(%eax) 两行代码。

2. 然后接下来的七条指令如下：

movl %esp, 4(%eax)

movl %ebx, 8(%eax)

movl %ecx, 12(%eax)

movl %edx, 16(%eax)

movl %esi, 20(%eax)

movl %edi, 24(%eax)

movl %ebp, 28(%eax)

这些指令完成了保存前一个进程的其他 7 个寄存器到 context 中的相应域中。至此前一个进程的执行现场保存完毕。

3. 再往后是恢复向一个进程的执行现场，这其实就是上述保存过程的逆执行过程，即从 context 的高地址的域 ebp 开始，逐一把相关域的值赋值给对应的寄存器。

4. 最后的 pushl 0(%eax) 其实是把 context 中保存的下一个进程要执行的指令地址 context.eip 放到了堆栈顶，这样接下来执行最后一条指令 “ret” 时,会把栈顶的内容赋值给 EIP 寄存器，这样就切换到下一个进程执行了，即当前进程已经是下一个进程了，从而完成了进程的切换。

最后分析一下 proc\_run 函数

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。

proc\_run 的执行过程为：

保存 IF 位并且禁止中断；

将 current 指针指向将要执行的进程；

更新 TSS 中的栈顶指针；

加载新的页表；

调用 switch\_to 进行上下文切换；

当执行 proc\_run 的进程恢复执行之后，需要恢复 IF 位。

以下是对 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：

idle\_proc，为第 0 个内核线程，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，用于调度其他进程或线程；

init\_proc，被创建用于打印 "Hello World" 的线程。本次实验的内核线程，只用来打印字符串。

语句

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

在进行进程切换的时候，需要避免出现中断干扰这个过程，所以需要在上下文切换期间清除 IF 位屏蔽中断，并且在进程恢复执行后恢复 IF 位。

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。