**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，完成这个初始化过程。

关键数据结构 struct proc\_struct

**struct proc\_struct {**

**enum proc\_state state; // Process state**

**int pid; // Process ID**

**int runs; // the running times of Proces**

**uintptr\_t kstack; // Process kernel stack**

**volatile bool need\_resched; // need to be rescheduled to release CPU?**

**struct proc\_struct \*parent; // the parent process**

**struct mm\_struct \*mm; // Process's memory management field**

**struct context context; // Switch here to run process**

**struct trapframe \*tf; // Trap frame for current interrupt**

**uintptr\_t cr3; // the base addr of Page Directroy Table(PDT)**

**uint32\_t flags; // Process flag**

**char name[PROC\_NAME\_LEN + 1]; // Process name**

**list\_entry\_t list\_link; // Process link list**

**list\_entry\_t hash\_link; // Process hash list**

**};**

mm：内存管理的信息，包括内存映射列表、页表指针等。

state：进程所处的状态。

parent：用户进程的父进程（创建它的进程）。

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

context：进程的上下文，用于进程切换

tf：中断帧的指针

cr3: cr3 保存页表的物理地址

代码填写

根据题目中的提示填写代码

**static struct proc\_struct \***

**alloc\_proc(void) {**

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

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

**//LAB4:EXERCISE1 YOUR CODE**

**/\***

**\* below fields in proc\_struct need to be initialized**

**\* enum proc\_state state; // Process state**

**\* int pid; // Process ID**

**\* int runs; // the running times of Proces**

**\* uintptr\_t kstack; // Process kernel stack**

**\* volatile bool need\_resched; // bool value: need to be rescheduled to release CPU?**

**\* struct proc\_struct \*parent; // the parent process**

**\* struct mm\_struct \*mm; // Process's memory management field**

**\* struct context context; // Switch here to run process**

**\* struct trapframe \*tf; // Trap frame for current interrupt**

**\* uintptr\_t cr3; // CR3 register: the base addr of Page Directroy Table(PDT)**

**\* uint32\_t flags; // Process flag**

**\* char name[PROC\_NAME\_LEN + 1]; // Process name**

**\*/**

**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;//初始化时间片**

**proc->kstack = 0;//内核栈的地址**

**proc->need\_resched = 0;//是否需要调度**

**proc->parent = NULL;//父节点为空**

**proc->mm = NULL; //内存管理初始化**

**memset(&(proc->context), 0, sizeof(struct context));//进程上下文初始化**

**proc->tf = NULL; //中断帧指针置为空，总是能够指向中断前的trapframe**

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

**proc->flags = 0; //标志位初始化**

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

**}**

**return proc;**

**}**

**context和\*tf的作用分析**

**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2 为新创建的内核线程分配资源

代码填写

**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;**

**//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**

**\* wakup\_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 wakup\_proc to make the new child process RUNNABLE**

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

**//第一步：申请内存块，如果失败，直接返回处理**

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

**goto fork\_out;**

**}**

**//将子进程的父节点设置为当前进程**

**proc->parent = current;**

**//第二步：为进程分配一个内核栈**

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

**goto bad\_fork\_cleanup\_proc;**

**}**

**//第三步：复制父进程的内存信息到子进程**

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

**goto bad\_fork\_cleanup\_kstack;**

**}**

**//第四步：复制父进程相关寄存器信息（上下文）**

**copy\_thread(proc, stack, tf);**

**//第五步：将新进程添加到进程列表（此过程需要加保护锁）**

**bool intr\_flag;**

**local\_intr\_save(intr\_flag);**

**{**

**proc->pid = get\_pid();**

**//建立散列映射方便查找**

**hash\_proc(proc);**

**//将进程链节点加入进程列表**

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

**//进程数+1**

**nr\_process ++;**

**}**

**local\_intr\_restore(intr\_flag);**

**//第六步：一切准备就绪，唤醒子进程**

**wakeup\_proc(proc);**

**//第七步：别忘了设置返回的子进程号**

**ret = proc->pid;**

**fork\_out:**

**return ret;**

**bad\_fork\_cleanup\_kstack:**

**put\_kstack(proc);**

**bad\_fork\_cleanup\_proc:**

**kfree(proc);**

**goto fork\_out;**

**}**

**练习3 理解proc\_run和它调用的函数如何完成进程切换的**

**proc\_run代码分析**

**void proc\_run(struct proc\_struct \*proc) {**

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

**bool intr\_flag;**

**struct proc\_struct \*prev = current, \*next = proc;**

**local\_intr\_save(intr\_flag);**

**{**

**current = proc;**

**load\_esp0(next->kstack + KSTACKSIZE);**

**lcr3(next->cr3);**

**switch\_to(&(prev->context), &(next->context));**

**}**

**local\_intr\_restore(intr\_flag);**

**}**

}

