### 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。

编码如下：

// alloc\_proc - alloc a proc\_struct and init all fields of proc\_struct

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;

}

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

do\_fork是创建线程的主要函数，kernel\_thread通过调用do\_fork函数来完成内核线程的创建工作，练习2中要求完成的do\_fork函数主要完成了6项工作

* 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

编码如下：

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; // 分配失败，直接返回

}

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

goto bad\_fork\_cleanup\_kstack; // 堆栈初始化失败，释放已占用的空间并返回

}

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

goto bad\_fork\_cleanup\_proc; // 复制或共享内存管理结构失败，释放已占用的空间并返回

}

copy\_thread(proc, stack, tf); // 复制中断帧和上下文

proc->pid = get\_pid(); // 分配pid

hash\_proc(proc); // 将新进程加入哈希表

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

nr\_process ++;

wakeup\_proc(proc); // 唤醒进程

ret = proc->pid; // 返回进程pid

fork\_out:

return ret;

bad\_fork\_cleanup\_kstack:

put\_kstack(proc);

bad\_fork\_cleanup\_proc:

kfree(proc);

goto fork\_out;

}

### 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)

1、kern\_init调用了proc\_init函数，后者启动了创建内核线程的步骤。idleproc = alloc\_proc()通过kmalloc函数获得了proc\_struct作为第0个进程的控制块，并初始化。proc\_init函数对idleproc内核线程进行进一步初始化。调用pid = kernel\_thread(init\_main, "Hello world!!", 0)创建一个内核线程init\_main

2、kernel\_thread函数创建了内核线程的临时中断帧，并调用do\_fork函数来进行进一步产生新的内核线程

3、do\_fork完成的工作在练习2中已充分说明，这里详细说明最重要的copy\_thread函数。随后到do\_fork完成所有工作，返回到kernel\_thread再返回到proc\_init再返回到kern\_init

4、此时在kern\_init后续有cpu\_idle和schedule进行进程调度

5、schedule将返回之前创建的进程，并调用proc\_run进行运行

6、proc\_run完成一些准备工作后，调用switch\_to最终完成切换

注意，新进程就是前面创建的init\_main，参考3可以知道当时proc->context.eip = (uintptr\_t)forkret，当switch\_to返回时，把栈顶的内容赋值给EIP寄存器，此时跳转到了forkret进行执行

7、forkret调用forkrets完成准备并最终进入init\_main。注意参考3可以知道proc->context.esp = (uintptr\_t)(proc->tf)，而在6中switch\_to最后压入了proc->context.eip，故在forkrets中[esp+4]即指向context.esp，这里就是中断帧proc->tf，参考栈的内容、struct trapframe及\_\_trapret就会理解跳转情况。当iret返回时就会进入此时的esp所指向的proc->tf.tf\_eip，在2中这个值被初始化为kernel\_thread\_entry。由此进入init\_main，当返回时调用do\_exit完成所有过程。