实验四：内核线程管理

**练习0：填写已有实验**

**本实验依赖实验1/2/3。请把你做的实验1/2/3的代码填入本实验中代码中 有“LAB1”,“LAB2”,“LAB3”的注释相应部分。**

Lab1：

kdebug.c:print\_stackframe

trap.c:idt\_init

trap.c:trap\_dispatch

Lab2：

default\_pmm.c:default\_init

default\_pmm.c:default\_init\_memmap

default\_pmm.c:default\_alloc\_pages

default\_pmm.c:default\_free\_pages

pmm.c:get\_pte

pmm.c:page\_remove\_pte

Lab3：

vmm.c:do\_pgfault

swap\_fifo.c:\_\_fifo\_map\_swappable

swap\_fifo.c:\_\_fifo\_swap\_out\_victim

**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。**

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）**

proc\_struct中的context：进程的上下文，用于进程切换。在 uCore中，所有的进程在内核中也是相对独立的（例如独立的内核堆栈以及上下文等）。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。具体切换过程定义在switch.S中。

proc\_struct中的tf：当前中断帧的指针。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。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，因此需要构造出一个伪造的中断返回现场，使得可以正确地将控制权转交给新的线程。

**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。**

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;

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

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

goto fork\_out;

proc->parent=current;

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

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

goto bad\_fork\_cleanup\_proc;

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

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

goto bad\_fork\_cleanup\_proc;

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

copy\_thread(proc,stack,tf);

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

proc->pid = get\_pid(); //创建一个id

// 将线程放入使用hash组织的链表以及所有线程的链表中

hash\_proc(proc);

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

nr\_process ++; // 将全局线程的数目加1

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

wakeup\_proc(proc);

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

ret = proc->pid; // 返回新线程的pid

fork\_out:

return ret;

bad\_fork\_cleanup\_kstack:

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。**

**是。get\_id将为每个调用fock的线程返回不同的id。**

**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．设置当前内核线程current->need\_resched为0；

2．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，完成进程切换

**在本实验的执行过程中，创建且运行了几个内核线程？**

通过kernel\_thread函数、proc\_init函数以及具体的实现结果可知，本次实验共建立了两个内核线程。

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

该语句应该是关闭中断以及恢复中断。