# 实验四：内核线程管理

米日阿地·买买提明171491103

## 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;   
proc->pid = -1;   
proc->runs = 0;   
proc->kstack = 0; ？  
proc->need\_resched = 0;  
proc->parent = NULL;  
proc->mm = NULL;   
memset(&(proc->context), 0, sizeof(struct context));   
proc->cr3 = boot\_cr3;  
proc->flags = 0;   
}  
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）

switch\_to是把被切换的进程的各个通用寄存器保存到进程的context结构中，然后加载即将开始运行的进程的context结构中保存的通用寄存器。trapframe代表着中断保存现场。

## 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;  
proc = alloc\_proc();  
if (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;  
}  
if (copy\_mm(clone\_flags, proc) != 0) { // 参考答案添加了错误处理  
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);  
nr\_process++; // 参考答案添加在此处（我本来以为用了get\_pid()就不需要这句了  
list\_add\_before(&proc\_list, &proc->list\_link);  
}  
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。 static int get\_pid(void) { static\_assert(MAX\_PID > MAX\_PROCESS); struct proc\_struct \*proc; list\_entry\_t \*list = &proc\_list, \*le; static int next\_safe = MAX\_PID, last\_pid = MAX\_PID; if (++ last\_pid >= MAX\_PID) { last\_pid = 1; goto inside; } if (last\_pid >= next\_safe) { inside: next\_safe = MAX\_PID; repeat: le = list; while ((le = list\_next(le)) != list) { proc = le2proc(le, list\_link); if (proc->pid == last\_pid) { if (++ last\_pid >= next\_safe) { if (last\_pid >= MAX\_PID) { last\_pid = 1; } next\_safe = MAX\_PID; goto repeat; } } else if (proc->pid > last\_pid && next\_safe > proc->pid) { next\_safe = proc->pid; } } } return last\_pid; }

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关闭中断

2.调用load\_esp0函数，设置next->kstack + KSTACKSIZE

3.设置CR3寄存器的值为要切换到的内核线程的页目录表起始地址

4.由 switch\_to函数完成具体的两个线程的执行现场切换。

### 在本实验的执行过程中，创建且运行了几个内核线程？

两个。

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

关闭中断以及恢复中断。