**实验二：内核线程管理**

**一.实验目的**

1.了解内核线程创建/执行的管理过程

2.了解内核线程的切换和基本调度过程

**二.实验内容**

本次实验将首先接触的是内核线程的管理。内核线程是一种特殊的进程，内核线程与用户进程的区别有两个：

内核线程只运行在内核态

用户进程会在在用户态和内核态交替运行

所有内核线程共用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

**三.实验题目**

**练习1：分配并初始化一个进程控制块**

在alloc\_proc函数中，初始化进程控制块。

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;

}

**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？**

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。实际利用context进行上下文切换的函数是kern/process/switch.S:switch\_to

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trapframe，uCore 在内核栈上维护了tf的链，可以参考trap.c::trap函数做进一步的了解

**练习2：为新创建的内核线程分配资源**

/\* 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;

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

proc = alloc\_proc();

if (proc == NULL) {

goto fork\_out;

}

proc -> parent = current;

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

int kstack\_success = setup\_kstack(proc);

if (kstack\_success != 0) {

goto bad\_fork\_cleanup\_proc;

}

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

int copy\_success = copy\_mm(clone\_flags, proc);

if (copy\_success != 0) {

goto bad\_fork\_cleanup\_kstack;

}

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

copy\_thread(proc, stack, tf);

bool intr\_flag;

local\_intr\_save(intr\_flag);

proc -> pid = get\_pid();

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

hash\_proc(proc);

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

nr\_process++;

local\_intr\_restore(intr\_flag);

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

wakeup\_proc(proc);

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

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：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。**

**分析一下proc\_init函数的调用过程：**

初始化proc\_list和hash\_list

调用alloc\_proc函数分配idleproc所需的TCB块，检验是否分配成功

对idleproc进行基本设置

将current变量置为idleproc

调用kernel\_thread函数，用init\_main函数创建一个内核线程

创建所需的trapframe

调用do\_fork函数，创建新进程

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

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

验证创建线程成功

proc\_init函数是由kern\_init函数调用的。在kern\_init完成其余初始化之后，它调用cpu\_idle函数，使得当前的idle\_proc进程让出控制权，交给initproc线程，进行上下文的切换；执行完之后，回到kernel\_thread\_entry，退出。

**在本实验的执行过程中，创建且运行了几个内核线程？**

由以上分析可知，在本实验的执行过程中，一共只创建了两个内核线程