操作系统 实验四 内核线程管理实验报告

实验目的：

了解内核线程创建、执行的管理过程

了解内核线程的切换和基本调度过程

实验内容：

当一个程序加载到内存中运行时，首先通过Ucore os 的内存管理子系统分配合适的空间，然后就需要考虑如何分配合适的空间，然后就需要考虑如何分时使用cpu来并发执行多个程序

让每个运行的程序 （这里用进程或线程小时）感到他们拥有自己的cpu

本次实验将会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，内核线程于用户进程的区别：

内核线程只运行在内核态，用户进程会在用户态和内核态交替进行，所有内核线程共用ucore内核内存空间 不需要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。

实验内容：

练习1 分配并初始化一个进程控制块（需要编码）

Alloc\_proc 函数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。

简要说明设计的实现过程：

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，而用户进程会在在用户态和内核态交替运行；所有内核线程直接使用共同 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。

这里主要是从 kern\_init 函数的物理内存管理初始化开始的，按照函数的次序做了一个简单的总结：1、pmm\_init()(1) 初始化物理内存管理器。(2) 初始化空闲页，主要是初始化物理页的 Page 数据结构，以及建立页目录表和页表。(3) 初始化 boot\_cr3 使之指向了 ucore 内核虚拟空间的页目录表首地址，即一级页表的起始物理地址。(4) 初始化第一个页表 boot\_pgdir。(5) 初始化 GDT，即全局描述符表。

1. pic\_init()初始化 8259A 中断控制器 3、idt\_init()初始化 IDT，即中断描述符表
2. vmm\_init() 主要就是实验了一个 do\_pgfault() 函数达到页错误异常处理功能，以及虚拟内存相关的 mm,vma 结构数据的创建/销毁/查找/插入等函数5、proc\_init()这个函数启动了创建内核线程的步骤，完成了 idleproc 内核线程和 initproc 内核线程的创建或复制工作，这是本次实验分析的重点，后面将详细分析。6、ide\_init()完成对用于页换入换出的硬盘(简称 swap 硬盘)的初始化工作7、swap\_init()swap\_init() 函数首先建立完成页面替换过程的主要功能模块，即 swap\_manager，其中包含了页面置换算法的实现操作系统是以进程为中心设计的，所以其首要任务是为进程建立档案，进程档案用于表示、标识或描述进程，即进程控制块。这里需要完成的就是一个进程控制块的初始化。

而这里我们分配的是一个内核线程的 PCB，它通常只是内核中的一小段代码或者函数，没有用户空间。而由于在操作系统启动后，已经对整个核心内存空间进行了管理，通过设置页表建立了核心虚拟空间(即 boot\_cr3 指向的二级页表描述的空间)。所以内核中的所有线程都不需要再建立各自的页表，只需共享这个核心虚拟空间就可以访问整个物理内存了。

首先在 kern/process/proc.h 中定义了 PCB，即进程控制块的结构体 proc\_struct，如下：

struct proc\_struct { //进程控制块

enum proc\_state state; //进程状态

int pid; //进程ID

int runs; //运行时间

uintptr\_t kstack; //内核栈位置

volatile bool need\_resched; //是否需要调度

struct proc\_struct \*parent; //父进程

struct mm\_struct \*mm; //进程的虚拟内存

struct context context; //进程上下文

struct trapframe \*tf; //当前中断帧的指针

uintptr\_t cr3; //当前页表地址

uint32\_t flags; //进程

char name[PROC\_NAME\_LEN + 1];//进程名字

list\_entry\_t list\_link; //进程链表

list\_entry\_t hash\_link; //进程哈希表

};

这里简单介绍下各个参数：

state：进程所处的状态。PROC\_UNINIT // 未初始状态PROC\_SLEEPING // 睡眠（阻塞）状态PROC\_RUNNABLE // 运行与就绪态PROC\_ZOMBIE // 僵死状态

pid：进程 id 号。

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

parent：用户进程的父进程。

mm：即实验三中的描述进程虚拟内存的结构体

context：进程的上下文，用于进程切换。

tf：中断帧的指针，总是指向内核栈的某个位置。中断帧记录了进程在被中断前的状态。

cr3：记录了当前使用的页表的地址

而这里要求我们完成一个 alloc\_proc 函数来负责分配一个新的 struct proc\_struct 结构，根据提示我们需要初始化一些变量，具体的代码如下：

\* 实现思路：

该函数的具体含义为创建一个新的进程控制块，并且对控制块中的所有成员变量进行初始化，根据实验指导书中的要求，除了指定的若干个成员变量之外，其他成员变量均初始化为0，取特殊值的成员变量如下所示：

proc->state = PROC\_UNINIT;

proc->pid = -1;

proc->cr3 = boot\_cr3; // 由于是内核线程，共用一个虚拟内存空间

对于其他成员变量中占用内存空间较大的，可以考虑使用 memset 函数进行初始化。

--------------------------------------------------------------------------------------------

/\*code\*/

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\_stuct中 struct context context和struct trapframe \*tf 成员变量含义和在本实验中的作用是什么？

首先不妨查看 struct context 结构体的定义，可以发现在结构体中存储这除了 eax 之外的所有通用寄存器以及 eip 的数值，这就提示我们这个线程控制块中的 context 很有可能是保存的线程运行的上下文信息；

接下来使用 find grep 命令查找在 ucore 中对 context 成员变量进行了设置的代码，总共可以发现两处，分别为 Swtich.S 和 proc.c 中的 copy\_thread 函数中，在其他部分均没有发现对 context 的引用和定义（除了初始化）；那么根据 Swtich 中代码的语义，可以确定 context 变量的意义就在于内核线程之间进行切换的时候，将原先的线程运行的上下文保存下来这一作用。

那么为什么没有对 eax 进行保存呢？注意到在进行切换的时候调用了 switch\_to 这一个函数，也就是说这个函数的里面才是线程之间切换的切换点，而在这个函数里面，由于 eax 是一个 caller-save 寄存器，并且在函数里 eax 的数值一直都可以在栈上找到对应，因此没有比较对其进行保存。

在 context 中保存着各种寄存器的内容，主要保存了前一个进程的现场（各个寄存器的状态），是进程切换的上下文内容，这是为了保存进程上下文，用于进程切换，为进程调度做准备。

在 ucore 中，所有的进程在内核中也是相对独立的。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在 kern/process/switch.S 中定义 switch\_to 函数。

--------------------------------------------------------------------------------------------

/\*code\*/

struct context {

uint32\_t eip;

uint32\_t esp;

uint32\_t ebx;

uint32\_t ecx;

uint32\_t edx;

uint32\_t esi;

uint32\_t edi;

uint32\_t ebp;

};

--------------------------------------------------------------------------------------------

接下来同样在代码中寻找对 tf 变量进行了定义的地方，最后可以发现在 copy\_thread 函数中对 tf 进行了设置，但是值得注意的是，在这个函数中，同时对 context 变量的 esp 和 eip 进行了设置。前者设置为 tf 变量的地址，后者设置为 forkret 这个函数的指针。接下来观察 forkret 函数，发现这个函数最终调用了 \_\_trapret 进行中断返回，这样的话，tf 变量的作用就变得清晰起来了。

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（跟lab1中切换特权级类似的技巧），因此需要构造出一个伪造的中断返回现场，也就是 trapframe，使得可以正确地将控制权转交给新的线程；具体切换到新的线程的做法为：

\* 调用switch\_to函数。

\* 然后在该函数中进行函数返回，直接跳转到 forkret 函数。

\* 最终进行中断返回函数 \_\_trapret，之后便可以根据 tf 中构造的中断返回地址，切换到新的线程了。

trapframe 保存着用于特权级转换的栈 esp 寄存器，当进程发生特权级转换的时候，中断帧记录了进入中断时任务的上下文。当退出中断时恢复环境。

tf 是一个中断帧的指针，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。

--------------------------------------------------------------------------------------------

/\*code\*/

struct trapframe {

struct pushregs {

uint32\_t reg\_edi;

uint32\_t reg\_esi;

uint32\_t reg\_ebp;

uint32\_t reg\_oesp; /\* Useless \*/

uint32\_t reg\_ebx;

uint32\_t reg\_edx;

uint32\_t reg\_ecx;

uint32\_t reg\_eax;

};

uint16\_t tf\_gs;

uint16\_t tf\_padding0;

uint16\_t tf\_fs;

uint16\_t tf\_padding1;

uint16\_t tf\_es;

uint16\_t tf\_padding2;

uint16\_t tf\_ds;

uint16\_t tf\_padding3;

uint32\_t tf\_trapno;

/\* below here defined by x86 hardware \*/

uint32\_t tf\_err;

uintptr\_t tf\_eip;

uint16\_t tf\_cs;

uint16\_t tf\_padding4;

uint32\_t tf\_eflags;

/\* below here only when crossing rings, such as from user to kernel \*/

uintptr\_t tf\_esp;

uint16\_t tf\_ss;

uint16\_t tf\_padding5;

} \_\_attribute\_\_((packed));

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，并未实际分配这些资源。Ucore一般通过do\_fork实际创建新的内核进程。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，但是存储位置不一样。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。

执行步骤：

\* 实现思路：

该函数的语义为为内核线程创建新的线程控制块，并且对控制块中的每个成员变量进行正确的设置，使得之后可以正确切换到对应的线程中执行。

proc = alloc\_proc(); // 为要创建的新的线程分配线程控制块的空间

if (proc == NULL) goto fork\_out; // 判断是否分配到内存空间

assert(setup\_kstack(proc) == 0); // 为新的线程设置栈，在本实验中，每个线程的栈的大小初始均为 2 个 Page，即 8KB

assert(copy\_mm(clone\_flags, proc) == 0); // 对虚拟内存空间进行拷贝，由于在本实验中，内核线程之间共享一个虚拟内存空间，因此实际上该函数不需要进行任何操作

copy\_thread(proc, stack, tf); // 在新创建的内核线程的栈上面设置伪造好的中端帧，便于后文中利用 iret 命令将控制权转移给新的线程

proc->pid = get\_pid(); // 为新的线程创建 pid

hash\_proc(proc); // 将线程放入使用 hash 组织的链表中，便于加速以后对某个指定的线程的查找

nr\_process ++; // 将全局线程的数目加 1

list\_add(&proc\_list, &proc->list\_link); // 将线程加入到所有线程的链表中，便于进行调度

wakeup\_proc(proc); // 唤醒该线程，即将该线程的状态设置为可以运行

ret = proc->pid; // 返回新线程的pid

--------------------------------------------------------------------------------------------

/\*code\*/

/\* 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; //尝试为进程分配内存

struct proc\_struct \*proc; //定义新进程

if (nr\_process >= MAX\_PROCESS) { //分配进程数大于 4096，返回

goto fork\_out; //返回

}

ret = -E\_NO\_MEM; //因内存不足而分配失败

//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

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

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

if ((proc = alloc\_proc()) == NULL) { //调用 alloc\_proc() 函数申请内存块，如果失败，直接返回处理

goto fork\_out;//返回

}

proc->parent = current; //将子进程的父节点设置为当前进程

if (setup\_kstack(proc) != 0) { //调用 setup\_stack() 函数为进程分配一个内核栈

goto bad\_fork\_cleanup\_proc; //返回

}

if (copy\_mm(clone\_flags, proc) != 0) { //调用 copy\_mm() 函数复制父进程的内存信息到子进程

goto bad\_fork\_cleanup\_kstack; //返回

}

copy\_thread(proc, stack, tf); //调用 copy\_thread() 函数复制父进程的中断帧和上下文信息

//将新进程添加到进程的 hash 列表中

bool intr\_flag;

local\_intr\_save(intr\_flag); //屏蔽中断，intr\_flag 置为 1

{

proc->pid = get\_pid(); //获取当前进程 PID

hash\_proc(proc); //建立 hash 映射

list\_add(&proc\_list, &(proc->list\_link)); //将进程加入到进程的链表中

nr\_process ++; //进程数加 1

}

local\_intr\_restore(intr\_flag); //恢复中断

wakeup\_proc(proc); //一切就绪，唤醒子进程

ret = proc->pid; //返回子进程的 pid

fork\_out: //已分配进程数大于 4096

return ret;

bad\_fork\_cleanup\_kstack: //分配内核栈失败

put\_kstack(proc);

bad\_fork\_cleanup\_proc:

kfree(proc);

goto fork\_out;

}

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。复制远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，返回新进程号。

Ucore是否做到给每个fork的线程一个唯一的id：

可以。保证每个 fork 的线程给的 ID 唯一，调用的 get\_pid() 函数，每次都从进程控制块链表中找到合适的 ID。线程的 PID 由 get\_pid 函数产生，该函数中包含了两个静态变量 last\_pid 以及 next\_safe。last\_pid 变量保存上一次分配的 PID，而 next\_safe 和 last\_pid 一起表示一段可以使用的 PID 取值范围 ，同时要求 PID 的取值范围为 ，last\_pid 和 next\_safe 被初始化为 MAX\_PID。每次调用 get\_pid 时，除了确定一个可以分配的 PID 外，还需要确定 next\_safe 来实现均摊以此优化时间复杂度，PID 的确定过程中会检查所有进程的 PID，来确保 PID 是唯一的。

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）

对proc\_run函数的分析：

在本实验的执行过程中，创建并运行了几个内核线程：

语句…local\_intr\_save(intr\_flag);....local\_intr\_restore(intr\_flag);…在这有何作用：

说明理由

在进行进程切换的时候，需要避免出现中断干扰这个过程，所以需要在上下切换期间清除 IF 位屏蔽中断，并且在进程恢复执行后恢复 IF 位。该语句的左右是关闭中断，使得在这个语句块内的内容不会被中断打断，是一个原子操作；这就使得某些关键的代码不会被打断，从而不会一起不必要的错误；比如说在 proc\_run 函数中，将 current 指向了要切换到的线程，但是此时还没有真正将控制权转移过去，如果在这个时候出现中断打断这些操作，就会出现 current 中保存的并不是正在运行的线程的中断控制块，从而出现错误；

.