**硬件一班**

**王倩倩**

**171491121**

**实验四：内核线程管理**

**一．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

**二．实验内容**

实验2/3完成了物理和虚拟内存管理，这给创建内核线程（内核线程是一种特殊的进程）打下 了提供内存管理的基础。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时使用CPU来“并发”执行多个程序，让每个运行 的程序（这里用线程或进程表示）“感到”它们各自拥有“自己”的CPU。

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

**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。

【提示】在alloc\_proc函数的实现中，需要初始化的proc\_struct结构中的成员变量至少包 括：state/pid/runs/kstack/need\_resched/parent/mm/context/tf/cr3/flags/name

请在实验报告中简要说明你的设计实现过程。请回答如下问题：

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）

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) 初始化了GDT，即全局描述符表。

2、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

4、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

**pid：进程id号。**

**runs：进程运行的时间，既然任务是初始化，那么runs必须是零。**

**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，目前实验未到这一步，暂时不管。**

**parent：用户进程的父进程，这是一个指针变量，记录它的父进程是谁。在所有进程中，只有一个进程没有父进程，就是内核创建的第一个内核线程idleproc。**

**mm：注释说它负责管理进程的虚拟memory，其实就是内存管理的信息，包括内存映射列表、页表指针等。mm成员变量在lab3中用于虚存管理。但在实际OS中，内核线程常驻内存**

**context：Switch here to run process，推测应该是进程的上下文，用于进程切换。使用Switch.S汇编文件中的定义。**

**tf：中断帧的指针，指导书上说，它是中断帧的指针，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。，它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：翻译注释的意思是：记录了当前使用的页表的地址**

**在ucore\_os\_lab-master/labcores/lab4/kern/process/proc.h**

**struct proc\_struct {**

**enum proc\_state state; // Process state**

**int pid; // Process ID**

**int runs; // the running times of Proces**

**uintptr\_t kstack; // Process kernel stack**

**volatile bool need\_resched; // bool value: need to be rescheduled to release CPU?**

**struct proc\_struct \*parent; // the parent process**

**struct mm\_struct \*mm; // Process's memory management field**

**struct context context; // Switch here to run process**

**struct trapframe \*tf; // Trap frame for current interrupt**

**uintptr\_t cr3; // CR3 register: the base addr of Page Directroy Table(PDT)**

**uint32\_t flags; // Process flag**

**char name[PROC\_NAME\_LEN + 1]; // Process name**

**list\_entry\_t list\_link; // Process link list**

**list\_entry\_t hash\_link; // Process hash list**

**};**

它主要定义了一个结构体proc\_struct，它也说，我们需要初始化这样一个结构体的一个对象并返回它。

**PROC\_UNINIT = 0, // uninitialized//未初始状态**

**PROC\_SLEEPING, // sleeping//睡眠（阻塞）状态**

**PROC\_RUNNABLE, // runnable(maybe running)//运行与就绪态**

**PROC\_ZOMBIE, // almost dead, and wait parent proc to reclaim his resource//僵尸状态**

**初始化代码如下：**

**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->tf = NULL;**

**proc->cr3 = boot\_cr3;**

**proc->flags = 0;**

**memset(proc->name, 0, PROC\_NAME\_LEN);**

**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

请在实验报告中简要说明你的设计实现过程。请回答如下问题：

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

**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;**

**第一步：调用alloc\_proc()函数申请内存块，如果失败，直接返回处理，相关的解释是，alloc\_proc（）函数在练习一中实现过，如果分配进程PCB失败，也就是说，进程一开始就是NULL，那么就会被if（proc！=NULL）判定为否，那么就不会分配初始化资源，连初始化资源都没有了，那么就会返回NULL，因此第一步这么处理，代码实现如下：**

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

**goto fork\_out;**

**}**

**第二步：将父进程的节点设置为当前进程**

**proc->parent = current;**

**第三步：调用setup\_stack()函数为进程分配一个内核栈**

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

**goto bad\_fork\_cleanup\_proc;**

**}**

**第四步：调用copy\_mm（）函数复制父进程的内存信息到子进程**

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

goto bad\_fork\_cleanup\_kstack;

**}**

**第五步：调用copy\_thread()函数复制父进程的断帧和上下文信息**

**copy\_thread(proc, stack, tf);**

**第六步：将新进程添加到进程的hash列表中**

**bool intr\_flag;**

**local\_intr\_save(intr\_flag);**

**{**

**proc->pid = get\_pid();**

**hash\_proc(proc);**

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

**nr\_process ++;**

**}**

**local\_intr\_restore(intr\_flag);**

**第七步：唤醒子进程**

**wakeup\_proc(proc);**

**第八步：返回子进程的pid**

**ret = proc->pid;**

**第九步：返回子进程的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）

请在实验报告中简要说明你对proc\_run函数的分析。并回答如下问题：

1.在本实验的执行过程中，创建且运行了几个内核线程？

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

完成代码编写后，编译并运行代码：make qemu

如果可以得到如 附录A所示的显示内容（仅供参考，不是标准答案输出），则基本正确。

**proc\_run函数的基本思路是：**

**1、让 current指向 next内核线程initproc；**

**2、设置任务状态ts中特权态0下的栈顶指针esp0 为 next 内核线程 initproc 的内核栈的栈顶，即 next->kstack + KSTACKSIZE ；**

**3、设置 CR3 寄存器的值为 next 内核线程 initproc 的页目录表起始地址 next->cr3，这实际上是完成进程间的页表切换；**

**4、由 switch\_to函数完成具体的两个线程的执行现场切换，即切换各个寄存器，当 switch\_to 函数执行完“ret”指令后，就切换到initproc执行了。**

**void**

**proc\_run(struct proc\_struct \*proc) {**

**if (proc != current) {**

**bool intr\_flag;**

**struct proc\_struct \*prev = current, \*next = proc;**

**local\_intr\_save(intr\_flag);**

**{**

**current = proc;**

**load\_esp0(next->kstack + KSTACKSIZE);**

**lcr3(next->cr3);**

**switch\_to(&(prev->context), &(next->context));**

**}**

**local\_intr\_restore(intr\_flag);**

**}**

**}**

**Switch函数：**

**.text**

**.globl switch\_to**

**switch\_to: # switch\_to(from, to)**

**# save from's registers**

**movl 4(%esp), %eax # eax points to from**

**popl 0(%eax) # save eip !popl**

**movl %esp, 4(%eax)**

**movl %ebx, 8(%eax)**

**movl %ecx, 12(%eax)**

**movl %edx, 16(%eax)**

**movl %esi, 20(%eax)**

**movl %edi, 24(%eax)**

**movl %ebp, 28(%eax)**

**# restore to's registers**

**movl 4(%esp), %eax # not 8(%esp): popped return address already**

**# eax now points to to**

**movl 28(%eax), %ebp**

**movl 24(%eax), %edi**

**movl 20(%eax), %esi**

**movl 16(%eax), %edx**

**movl 12(%eax), %ecx**

**movl 8(%eax), %ebx**

**movl 4(%eax), %esp**

**pushl 0(%eax) # push eip**

**ret**