# 操作系统实验二

**学号：171491219**

**姓名：崔凯慧**

### 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

### 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。

### 三、实验题目及过程

**（一）分配并初始化一个进程控制块**

**1.proc.c的相关注释**

**enum proc\_state state**：表示进程状态，在此函数中应赋值为PROC\_UNINIT，表示该进程的初始化尚未完成。

**int pid：**初始赋值为-1，表示尚未分配（pid在do\_fork函数中通过调用get\_pid进行分配）

**int runs：**已运行次数，此处赋值为0

**uintptr\_t kstack：**内核堆栈起始地址，此时堆栈尚未分配，因此置为0；实际在do\_fork函数中通过调用setup\_kstack进行分配

**volatile bool need\_resched：**当前进程是否需要调度；初始化为不需要（0）

**struct proc\_struct \*parent：**当前进程的父进程，初始化为NULL；在do\_fork中初始化为调用do\_fork的当前进程

**struct mm\_struct \*mm：**内存管理，初始化为NULL；在do\_fork中通过调用copy\_mm进行初始化

**struct context context：**在do\_fork中通过调用copy\_thread函数进行初始化

**struct trapframe \*tf：**当前的中断帧，初始化为NULL；在do\_fork中通过调用copy\_thread函数进行初始化

**uintptr\_t cr3：**当前进程的页表基地址；直接初始化为kernel的页表基地址boot\_cr3

**uint32\_t flags：**当前进程属性，因为是初始化，所以置为0

**char name[PROC\_NAME\_LEN + 1]**：进程的名称

2.代码：

**static struct proc\_struct \***

**alloc\_proc(void) {**

**//首先调用alloc\_proc函数来通过kmalloc函数获得proc\_struct结构的一块内存块，作为第0个进程控制块。并把proc进行初步初始化（即把proc\_struct中的各个成员变量清零）。但有些成员变量设置了特殊的值**

**struct proc\_struct \*proc = kmalloc(sizeof(struct proc\_struct));**

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

**proc->state = PROC\_UNINIT; // 正在创建和初始化状态中**

**proc->pid = -1; // 未初始化的进程id为-1**

**proc->runs = 0; // 还没有运行过**

**proc->kstack = 0; // 初始化内核堆栈**

**proc->need\_resched = 0; // 初始化为不需要调度**

**proc->parent = NULL;**

**proc->mm = NULL;**

**memset(&(proc->context), 0, sizeof(struct context));**

**proc->cr3 = boot\_cr3; // 使用内核页目录表的基址**

**proc->flags = 0; // 标志位置为0**

**memset(proc->name, 0, PROC\_NAME\_LEN); // 将进程名清零**

**}**

**return proc;**

**}**

1. 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什么？

答：context是进程的上下文，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在*kern/process/switch.S*中定义switch\_to。

tf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。

1. 为新创建的内核线程分配资源
2. 分析：

do\_fork是创建线程的主要函数。kernel\_thread函数通过调用do\_fork函数最终完成了内核线程的创建工作。do\_fork函数主要做了以下6件事情：

①分配并初始化进程控制块（alloc\_proc函数）；

②分配并初始化内核栈（setup\_stack函数）；

③根据clone\_flag标志复制或共享进程内存管理结构（copy\_mm函数）；

④设置进程在内核（将来也包括用户态）正常运行和调度所需的中断帧和执行上下文（copy\_thread函数）；

⑤把设置好的进程控制块放入hash\_list和proc\_list两个全局进程链表中；

⑥自此，进程已经准备好执行了，把进程状态设置为“就绪”态；

⑦设置返回码为子进程的id号。

这里需要注意的是，如果上述前3步执行没有成功，则需要做对应的出错处理，把相关已经占有的内存释放掉。

2.代码

**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) { // 添加了错误处理**

**goto fork\_out;**

**}**

**proc->parent = current;**

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

**if (setup\_kstack(proc) != 0) { // 添加了错误处理**

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

**}**

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

**if (copy\_mm(clone\_flags, proc) != 0) { // 添加了错误处理**

**goto bad\_fork\_cleanup\_kstack;**

**}**

**// 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**

**// 关中断的原因是，进程号要求唯一性，此操作需要为原子操作，防止被打断而重复添加**

**bool intr\_flag;**

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

**{**

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

**hash\_proc(proc);**

**nr\_process++;**

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

**}**

**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;**

**}**

* **2.**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。
* 答：找到 get\_pid 函数，可以看到 get\_pid 函数有做规避重复的措施，因此只要 get\_pid 互斥（例如关闭中断），就可以保证 pid 唯一。

1. 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。

答：对进程切换的控制是通过sched.c:schedule函数完成的。一旦当前进程的need\_resched变量被置为1，就调用schedule函数选择下一个要运行的进程，调用proc\_run函数开始运行。它的主要过程为：将当前进程设为传入的进程；修改 esp 指针的值；修改页表项；使用 switch\_to 进行上下文切换。