操作系统实验二

——UCORE lab 4

班级：硬件一班 姓名：于华宇 学号：171491120

## 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

当一个程序加载到内存中运行时,首先通过ucoreOS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:

1. 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
2. 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
3. 练习一：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。

步骤：a、找到alloc\_proc函数，代码如下：

**// alloc\_proc - alloc a proc\_struct and init all fields of proc\_struct**

**static struct proc\_struct \***

**alloc\_proc(void) {**

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

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

**//LAB4:EXERCISE1 YOUR CODE**

**/\***

**\* below fields in proc\_struct need to be initialized**

**\* 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**

**\*/**

**}**

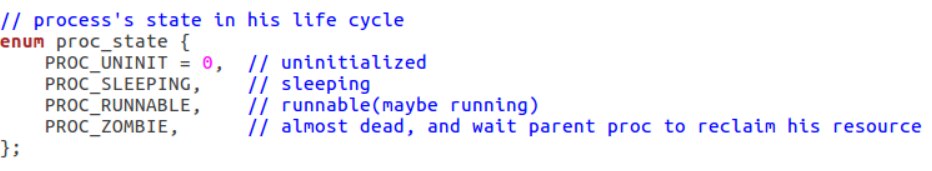
**return proc;**

**}**

函数中已经注明了需要初始化的部分，即一个struct proc\_struct类型的变量。结构体的具体信息可以查阅proc.h文件。

b、初始化如下：

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

****

关于proc\_state，在proc.h中定义了进程生命周期的四种状态，由于初始化进程控制块，还没有进程进入。所以状态为uninitialized

**答案：proc->state=PROC\_UNINIT;**

**2、int pid; // Process ID**

由于进程未初始化，pid=-1

**答案：proc->pid=-1;**

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

进程没有运行，所以运行时间为0

**答案：proc->runs=0;**

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

kstack记录了分配给该进程或线程的内核栈的位置。还没有执行进程，默认分配地址为0

**答案：proc->kstack=0;**

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

记录是否需要调度，此时还不需要

**答案：proc->need\_resched=0;**

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

记录用户进程的父进程，在所有进程中，只有内核创建的第一个内核线程idleproc没有父进程。初始化过程时，没有父进程。

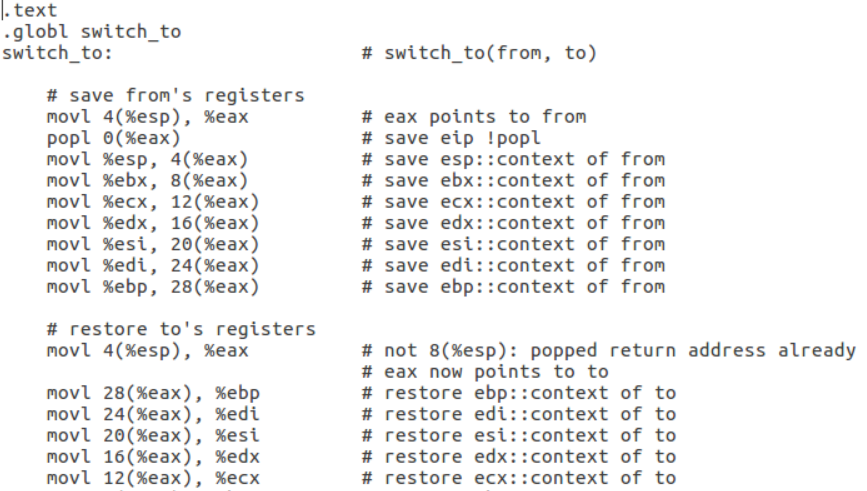
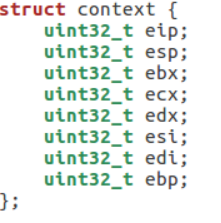
**答案：proc->parent=NULL;**

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

内存管理信息。初始化时为空。

**答案：proc->mm=NULL;**

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

进程的上下文,用于进程切换(参见switch.S)。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。

资料显示void\*memset（void \*s，int ch，size\_t n）将s中当前位置后面的n个字节用ch替换并返回s

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

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

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

**答案：proc->tf=NULL;**

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

cr3 保存页表的物理地址,目的就是进程切换的时候方便直接使用 lcr3实现页表切换,避免每次都根据 mm来计算 cr3。mm数据结构是用来实现用户空间的虚存管理的,但是内核线程没有用户空间,它执行的只是内核中的一小段代码(通常是一小段函数),所以它没有mm结构,也就是NULL。当某个进程是一个普通用户态进程的时候,PCB中的cr3就是mm中页表(pgdir)的物理地址;而当它是内核线程的时候,cr3 等于boot\_cr3。而boot\_cr3指向了uCore启动时建立好的内核虚拟空间的页目录表首地址。

**答案：proc->cr3=boot\_cr3;**

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

**答案：proc->flags=0；**

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

进程名称。

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

（二）练习二：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:

（1）调用alloc\_proc,首先获得一块用户信息块。

（2）为进程分配一个内核栈。

（3）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)

（4）复制原进程上下文到新进程

（5）将新进程添加到进程列表

（6）唤醒新进程

（7）返回新进程号

步骤：a、do\_fork函数代码如下：

**/\* 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;**

**//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**

**fork\_out:**

**return ret;**

**}**

b、添加代码如下（参考labcodes\_answer）

**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;**

**}**

**//调用alloc\_proc()创建新进程控制块 call alloc\_proc to allocate a proc\_struct**

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

**goto fork\_out;//若一开始创建进程控制块就失败了，那么创建新进程失败**

**proc->parent=current; //子进程进程控制块创建成功，将其中的父进程参数值改为为当前进程**

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

**//调用setup\_kstack(struct proc\_struct \* proc) 给子进程创建一个内核栈**

**//查看setup\_kstack函数，发现分配成功返回0**

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

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

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

**//根据clone\_flag调用copy\_mm()复制或共享内存**

**//copy\_mm()正常返回0**

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

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

**}**

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

**//调用copy\_thread（）给进程控制块中的tf中断指针和上下文赋值**

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

**// insert proc\_struct into hash\_list && proc\_list**

**//把新进程插入到进程的哈希表中和进程控制块双向链表中**

**bool intr\_flag;**

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

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

**hash\_proc(proc);//加入哈希表**

**nr\_process++;//进程数加一**

**list\_add(&proc\_list,&(proc->list\_link));//加入进程控制块双向链表**

**}**

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

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

**//调用wakeup\_proc函数使新的子进程进入就绪态**

**wakeup\_proc(proc);**

**// set ret vaule using child proc's pid**

**//将ret的值设为子进程的pid**

**ret=proc->pid;**

**ret = -E\_NO\_MEM;**

**fork\_out:**

**return ret;**

**}**

(三) 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)

proc\_run函数代码如下：

**// proc\_run - make process "proc" running on cpu**

**// NOTE: before call switch\_to, should load base addr of "proc"'s new PDT**

**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);**

**}**

**}**

如果当前运行的进程不是proc，则说明创建了新的进程。首先要用链表记录前一个进程为current，后一个进程为新创建的proc进程。之后进行进程切换。将当前进程变为proc，将proc的kstack放入栈中，进行页表切换，上下文也从原来进程的上下文切换到新进程的上下文。但一切都保存在链表中，不会丢失原来进程的栈，页，上下文。（switch to中前一部分保存上下文切换现场，便于进程下一次执行，后一部分读取之前的进程现场，继续之前被调度下去的进程）