# 操作系统Lab4

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。

（1）此题的代码工作在proc.c中的alloc\_proc函数中，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;

代码如下

struct proc\_struct \*proc = kmalloc(sizeof(struct proc\_struct)); // 为线程控制块分配空间

if (proc != NULL) {

proc->state = PROC\_UNINIT; // 初始化具有特殊值的成员变量

proc->cr3 = boot\_cr3;

proc->pid = -1;

proc->runs = 0; // 对其他成员变量清零处理

proc->kstack = 0;

proc->need\_resched = 0;

proc->parent = NULL;

proc->mm = NULL;

memset(&proc->context, 0, sizeof(struct context)); // 使用memset函数清零占用空间较大的成员变量

proc->tf = NULL;

proc->flags = 0;

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

}

（2）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？

**struct context context含义和作用**

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。

**struct trapframe \*tf成员变量含义和在本实验中的作用**

同样在代码中寻找，关于对tf变量进行定义的地方，可发现在copy\_thread函数中对tf进行了设置，但是在这个函数中，同时又对context变量的esp和eip进行了设置，前一个设置为tf变量的地址、后一个设置为forkret这个函数的指针，接着观察forkret函数，我们发现这个函数也调用了\_\_trapret进行中断返回，这样便可以清晰的得出tf变量的作用。tf变量的作用在于在构造新的线程的时候，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；

练习2：为新创建的内核线程分配资源（需要编码）

**（1）设计过程及代码**

在这题中，主要需要实现的代码位于proc.c的do\_fork函数中，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);

assert(copy\_mm(clone\_flags, proc) == 0);

copy\_thread(proc, stack, tf);

proc->pid = get\_pid(); // 为新的线程创建pid

hash\_proc(proc);

nr\_process ++; // 将全局线程的数目加1

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

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

（2）请说明ucore是否做到给每个fork的线程一个唯一的id？请说明你的分析和理由

可以。ucore中为新的fork的线程分配pid的函数为get\_pid。

在该函数中使用到了两个静态的局部变量next\_safe和last\_pid，根据推测，在每次进入get\_pid函数的时候，这两个变量的数值之间的取值均是合法的pid，如果有严格的next\_safe > last\_pid + 1，那么久可以直接取last\_pid + 1作为新的pid；

如果在进入函数的时候，这两个变量之后没有合法的取值，也就是说next\_safe > last\_pid + 1不成立，那么进入循环，在循环之中首先通过if (proc->pid == last\_pid)这一分支确保了不存在任何进程的pid与last\_pid重合，然后再通过if (proc->pid > last\_pid && next\_safe > proc->pid)这一判断语句保证了不存在任何已经存在的pid满足：last\_pid<pid<next\_safe，这样就确保了最后能够找到这么一个满足条件的区间，获得合法的pid；

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）

(1)代码及分析

void

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

if (proc != current) { // 判断需要运行的线程是否已经运行着了

bool intr\_flag;

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

local\_intr\_save(intr\_flag); // 关闭中断

{

current = proc;

load\_esp0(next->kstack + KSTACKSIZE); // 设置TSS

lcr3(next->cr3); // 修改当前的cr3寄存器成需要运行线程（进程）的页目录表

switch\_to(&(prev->context), &(next->context)); // 切换到新的线程

}

local\_intr\_restore(intr\_flag);

}

在 proc\_run中首先进行了TSS以及cr3寄存器的设置，然后调用到了swtich\_to函数来切换线程，根据上文中对switch\_to函数的分析可以知道，在调用该函数之后，首先会恢复要运行的线程的上下文，然后由于恢复的上下文中已经将返回地址（copy\_thread函数中完成）修改成了forkret函数的地址(如果这个线程是第一运行的话，否则就是切换到这个线程被切换出来的地址)，也就是会跳转到这个函数，最后进一步跳转到了\_\_trapsret函数，调用iret最终将控制权切换到新的线程；

（2）在本实验的执行过程中，创建且运行了几个内核线程？

两个内核线程，分别为：

idleproc: 最初的内核线程

initproc: 被创建用于打印"Hello World"的线程；