# 实验四 内核线程管理实验

## 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

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。

## 实验步骤



代码：

proc->state = PROC\_UNINIT;

proc->pid = -1;

proc->runs = 0;

proc->kstack = 0;

proc->need\_resched = 0;

proc->parent = NULL;

proc->mm = NULL;

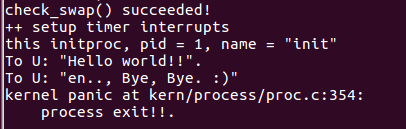
memset(&(proc->context), 0, sizeof(struct context));

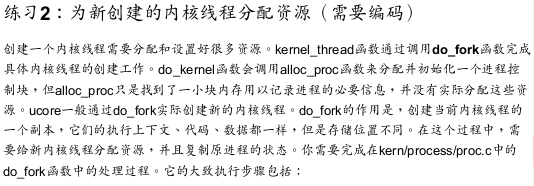
proc->tf = NULL;

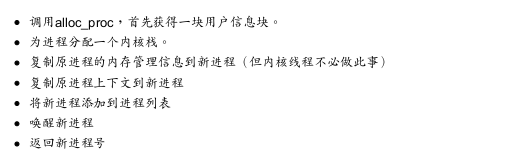
proc->cr3 = boot\_cr3;

proc->flags = 0;

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







编码：

if ((proc = alloc\_proc()) == NULL) {

         goto fork\_out;

  }

proc->parent = current;

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

goto bad\_fork\_cleanup\_proc;

}

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

Goto bad\_fork\_cleanup\_kstack;

}

copy\_thread(proc,stack,tf);

   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);  
wakeup\_ proc(proc);

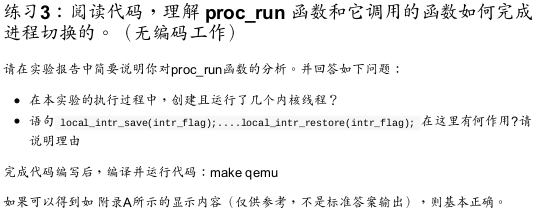
ret = proc->pid;



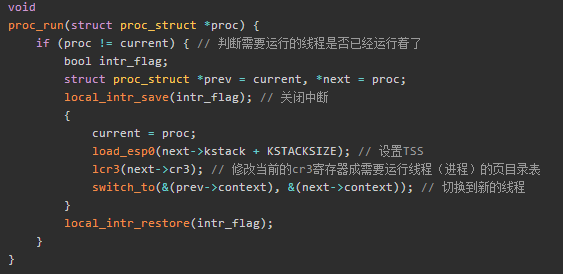
### 答：

可以。

ucore中为新的fork的线程分配pid的函数为get\_pid，在该函数中使用的两个静态局部变量next\_safe和last\_pid，在每次进入get\_pid函数的时候，这两个变量的数值取值均是没有被使用过的，这样，如果有next\_safe > last\_pid + 1，那就可以直接取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，那么进入循环，在循环之中首先通过if (proc->pid == last\_pid)这一分支保证了不存在任何进程的pid和last\_pid重合，然后再通过if (proc->pid > last\_pid && next\_safe > proc->pid)判断保证了不存在任何已经存在的pid满足：last\_pid<pid<next\_safe，这就保证了最后能够找到一个满足条件的区间，获得合法的pid。



分析代码：



问题回答：

1. 总共创建了两个内核线程，分别为：

* idleproc: 最初的内核线程，在完成新的内核线程的创建以及各种初始化工作之后，进入死循环，用于调度其他线程；
* initproc: 被创建用于打印"Hello World"的线程；

1. 语句 local\_intr\_save(intr\_flag);....local\_intr\_restore(intr\_flag);

的作用

* 这个语句的左右是关闭中断，使这个语句块内的内容不会被中断打断，是一个原子操作；
* 这使得一些关键的代码不会被打断，从而引起不必要的错误；
* 如在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；

编译make qemu

