练习一

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。

struct proc\_struct {

enum proc\_state state;

int pid;

int runs;

uintptr\_t kstack;

volatile bool need\_resched;

struct proc\_struct \*parent;

struct mm\_struct \*mm;

struct context context;

struct trapframe \*tf;

uintptr\_t cr3;

Directroy Table(PDT)

uint32\_t flags;

char name[PROC\_NAME\_LEN + 1];

list\_entry\_t list\_link;

list\_entry\_t hash\_link;

};

static struct proc\_struct \*alloc\_proc(void) {

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

if (proc != NULL) {

proc->state = PROC\_UNINIT; //设置进程为未初始化状态

proc->pid = -1; //未初始化的的进程id为-1

proc->runs = 0; //初始化时间片

proc->kstack = 0; //内存栈的地址

proc->need\_resched = 0; //是否需要调度设为不需要

proc->parent = NULL; //父节点设为空

proc->mm = NULL; //虚拟内存设为空

memset(&(proc->context), 0, sizeof(struct context));//上下文的初始化

proc->tf = NULL; //中断帧指针置为空

proc->cr3 = boot\_cr3; //页目录设为内核页目录表的基址

proc->flags = 0; //标志位

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

}

return proc;

}\*\*\*参考\*\*\*

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)

struct context context：进程的上下文，用于进程切换。

struct trapframe \*tf：中断帧的指针，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。

练习二

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;

//1：调用alloc\_proc()函数申请内存块，如果失败，直接返回处理

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

goto fork\_out;

}

//2.将子进程的父节点设置为当前进程

proc->parent = current;

//3.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;

}

//4.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;

}

//5.调用copy\_thread()函数复制父进程的中断帧和上下文信息

copy\_thread(proc, stack, tf);

//6.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); //建立映射

nr\_process ++; //进程数加1

list\_add(&proc\_list, &(proc->list\_link));//将进程加入到进程的链表中

}

local\_intr\_restore(intr\_flag);

// 7.一切就绪，唤醒子进程

wakeup\_proc(proc);

// 8.返回子进程的pid

ret = proc->pid;

fork\_out:

return ret;

bad\_fork\_cleanup\_kstack:

put\_kstack(proc);

bad\_fork\_cleanup\_proc:

kfree(proc);

goto fork\_out;

}

可以做到，在使用 fork 或 clone 系统调用时产生的进程均会由内核分配一个新的唯一的PID值。具体来说，就是在分配PID时，设置一个保护锁，暂时不允许中断，这样在就唯一地分配了一个PID。

练习三

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:

在本实验的执行过程中,创建且运行了几个内核线程?

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

说明理由

1.两个线程

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；

2. 该语句的左右是关闭中断，使得在这个语句块内的内容不会被中断打断，这就使得某些关键的代码不会被打断，从而不会一起不必要的错误；