实验二

练习1：分配并初始化一个进程控制块

实验编码：

proc->state = PROC\_UNINIT;

proc->pid = -1;

proc->cr3 = boot\_cr3;

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));

proc->tf = NULL;

proc->flags = 0;

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

}

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？

1. struct context context的作用：

首先查看struct context结构体的定义，可以发现在结构体中存储这除了eax之外的所有通用寄存器以及eip的数值，这就提示这个线程控制块中的context很有可能是保存的线程运行的上下文信息；



接下来使用find grep命令查找在ucore中对context成员变量进行了设置的代码，总共可以发现两处，分别为Swtich.S和proc.c中的copy\_thread函数中，在其他部分均没有发现对context的引用和定义（除了初始化）；那么根据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。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，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。

2. struct trapframe \*tf的作用：

在代码中寻找对tf变量进行了定义的地方，可以发现在copy\_thread函数中对tf进行了设置，在这个函数中，同时对context变量的esp和eip进行了设置，前者设置为tf变量的地址、后者设置为forkret这个函数的指针，接下来观察forkret函数，发现这个函数最终调用了\_\_trapret进行中断返回，tf变量的作用就变得清晰起来：tf变量的作用在于在构造出了新的线程的时候，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，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

设计实现过程：

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。

实验编码：

proc = alloc\_proc(); // 为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); // 为新的线程设置栈，

assert(copy\_mm(clone\_flags, proc) == 0); // 对虚拟内存空间进行拷贝

copy\_thread(proc, stack, tf); // 在新创建的内核线程的栈上面设置伪造好的中端帧，便于后文中利用iret命令将控制权转移给新的线程

proc->pid = get\_pid(); // 为新的线程创建pid

hash\_proc(proc); // 将线程放入使用hash组织的链表中，便于加速以后对某个指定的线程的查找

nr\_process ++; // 将全局线程的数目加1

list\_add(&proc\_list, &proc->list\_link); // 将线程加入到所有线程的链表中，便于进行调度

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

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。

可以。ucore中为新的fork的线程分配pid的函数为get\_pid，接下来分析该函数的内容:

在该函数中使用到了两个静态的局部变量next\_safe和last\_pid，根据命名推测，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，这样的话，如果有严格的next\_safe > last\_pid + 1，那么就可以直接取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，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。

之所以在该函数中使用这种方法，维护一个合法的pid的区间，是为了优化时间效率，如果简单的话，每次需要枚举所有的pid，并且遍历所有的线程，这就使得时间代价过大，并且不同的调用get\_pid函数的时候不能利用到先前调用这个函数的中间结果。

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。

分析：

首先注意到在本实验框架中，唯一调用到这个函数是在线程调度器的schedule函数中，也就是可以推测proc\_run的语义就是将当前的CPU的控制权交给指定的线程。

接下来结合代码分析函数的内部构成：

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

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);

}

}

可以看到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，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。

在本实验的执行过程中，创建且运行了几个内核线程？

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。

语句local\_intr\_save(intr\_flag);....local\_intr\_restore(intr\_flag); 在这里有何作用?请说明理由

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，是一个原子操作。

使得某些关键的代码不会被打断，从而不会引起不必要的错误。

比如说在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。

运行结果：



