操作系统 Lab4 内核线程管理 实验报告

黄志鹏 PB16150288

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

- 实现内核线程的创建与管理；

基本练习

练习1:分配并初始化一个进程控制块(需要编码)

alloc_proc的实现较为简单，主要是为一个proc_struct结构体对象分配内存，初始化其中的成员并返回这个对象。注意到alloc_proc中进行的初始化只是为了避免结构体的内容不确定。很多真正的初始化工作（例如进程ID分配什么的）都是要留到练习二里面再去做的。

关于context和tf这两个变量，context指的是进程的“上下文”，实际上其中保存了进程运行过程中的几乎全部寄存器，只有这样才能让进程返回时能够处在和原来一样的状态（eax之所以没有保存是因为不需要，fork时eax本身就是用来做返回值的），而tf是每个进程内核栈中的一个地址，指向该进程的陷入帧。在进程切换返回之后，系统可以根据这个陷入帧回到进程之前的状态。

```
static struct proc_struct *alloc_proc(void) {
    ...
        proc->state = PROC_UNINIT;
        proc->pid = -1;
        proc->runs = 0;
        proc->kstack = NULL;
        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+1);
    ...
}
```

1.1 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?

context是进程执行的上下文，用于在进程切换时保存当前ebx、ecx、edx、esi、edi、esp、ebp、eip八个寄存器，即保存当前进程的执行状态上下文。

tf是中断帧，当进程从用户空间转换到内核空间时，中断帧记录了进程在被中断前的状态。当内核需要切换回用户空间时，需要调整中断帧以恢复让进程继续执行的各寄存器值。

练习2:为新创建的内核线程分配资源需要编码

新创建的内核线程分配资源的过程主要是：

1. 申请一个初始化后的进程控制块
2. 为子内核线程建立栈空间
3. 拷贝或者共享内存空间
4. 在进程控制块中设置中断帧和上下文
5. 为进程分配进程号
6. 将新分配的进程控制块插入哈希表和进程链表；
7. 返回进程的进程号

```
int do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    ...
    if ((proc = alloc_proc()) == NULL)
        goto fork_out;
    if ((ret = setup_kstack(proc)) != 0)
        goto fork_out;
    if ((ret = copy_mm(clone_flags, proc)) != 0)
        goto fork_out;
    copy_thread(proc, stack, tf);
    ret = proc->pid = get_pid();
    hash_proc(proc);
    list_add(&proc_list, &(proc->list_link));
    wakeup_proc(proc);
    ...
}
```

2.1 请说明ucore是否做到给每个新fork的线程一个唯一的id？

是。

线程的PID由get_pid函数产生，该函数最终返回的条件是遍历全部进程，其进程号与将返回的新的进程号不同，从而保证了新的进程的pid唯一。同时通过next_safe这个变量确保进程号在一个合法范围内。

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的执行过程为：

1. 保存中断位并关中断
2. 将current指针指向将要执行的进程
3. 更新TSS中的栈顶指针
4. 加载新的页表
5. 调用switch_to进行上下文切换
6. 当调用proc_run的进程重新执行之后恢复中断位

```
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);  // 为新的线程设置栈，在本实验中，每个线程的栈的大小初始均为2个Page, 即8KB
assert(copy_mm(clone_flags, proc) == 0);  // 对虚拟内存空间进行拷贝，由于在本实验中，内核线程之间共享一个虚拟内存空间，因此实际上该函数不需要进行任何操作
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
```

3.1 在本实验的执行过程中，创建且运行了几个内核线程？

两个:

idleproc，这个线程不断试探是否有可以调度的进程，有则执行；

initproc，本实验中的仅输出一段字符串。

3.2 语句local_intr_save(intr_flag);....local_intr_restore(intr_flag);在这里有何作用?请说明理由.

保存中断位并关中断，在进程重新执行时恢复中断位。关中断是防止进程切换过程被中断打断而不得不转中断服务例程。

实验中涉及的知识点列举

在本次实验中设计到的知识点有：

- 线程控制块的概念以及组成；
- 切换不同线程的方法；

对应到的OS中的知识点有：

- 对内核线程的管理；
- 对内核线程之间的切换；

这两者之间的关系为，前者为后者在OS中的具体实现提供了基础；