操作系统——Lab4

学号：2013921
姓名：周延霖
专业：信息安全

> 由于没有什么抱怨的，所以再次回答一下指导书上提出的问题

练习1：分配并初始化一个进程控制块（需要编码）

问题回答

> 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指进程上下文，这部分空间用于保存创建进程时父进程的部分寄存器值：eip, esp, ebx, ecx, edx, esi, edi, ebp；其他寄存器在切换进程时值不变，故不需要保存。在本实验中，当idle进程被CPU切换为init进程时，将idle进程的上下文保存在`idleproc->context`中，如果uCore调度器选择了idleproc执行，就要根据idleproc->context恢复现场，继续执行。

tf是中断帧的指针，总是指向内核栈的某个位置：当进程从用户空间跳到内核空间时，中断帧记录了进程在被中断前的状态。当内核需要跳回用户空间时，需要调整中断帧以恢复让进程继续执行的各寄存器值。

其结构体如下：

```
struct trapframe {
    struct pushregs tf_regs;
    uint16_t tf_gs;
    uint16_t tf_padding0;
    uint16_t tf_fs;
    uint16_t tf_padding1;
    uint16_t tf_es;
    uint16_t tf_padding2;
    uint16_t tf_ds;
    uint16_t tf_padding3;
    uint32_t tf_trapno;
    /* below here defined by x86 hardware */
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding4;
    uint32_t tf_eflags;
    /* below here only when crossing rings, such as from user to kernel */
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding5;
} __attribute__((packed));
```

练习2：为新创建的内核线程分配资源（需要编程）

问题回答

> 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

对于这个问题，分配ID的代码是get_pid()，其内容如下

```
static int
get_pid(void) {
    static_assert(MAX_PID > MAX_PROCESS);
    struct proc_struct *proc;
    list_entry_t *list = &proc_list, *le;
    static int next_safe = MAX_PID, last_pid = MAX_PID;
    if (++ last_pid >= MAX_PID) {
        last_pid = 1;//当第一次调用时lastpid=1
        goto inside;
    }
    if (last_pid >= next_safe) {
    inside:
        next_safe = MAX_PID;
    repeat:
        le = list;
        while ((le = list_next(le)) != list)
        {
            proc = le2proc(le, list_link);
            if (proc->pid == last_pid)
            {//如果当前进程pid等于lastpid，lastpid++
                if (++ last_pid >= next_safe)
                {//如果lastpid大于nextsafe
                    if (last_pid >= MAX_PID)
                    {//如果lastpid大于MAX_PID,lastpid=1
                        last_pid = 1;
                    }
                    next_safe = MAX_PID;//把next_safe设置为MAXPID，跳回repeat
                    goto repeat;
                }
            }
            else if (proc->pid > last_pid && next_safe > proc->pid)
            {//如果proc->pid在lastpid到safe之间，就把safe设置为proc->pid
                next_safe = proc->pid;
            }
        }
    }
    return last_pid;
}
```

这段代码会不断的在链表中遍历，直到找到一个合适的last_pid才会返回，这个last_pid满足两个条件(1)不大于MAX_PID(2)未被分配过，因此ucore为每个新fork的线程分配了一个唯一的id。

能做到给每个新fork的线程一个唯一的id，在该函数中使用到了两个静态的局部变量next_safe和last_pid，在每次进入get_pid函数的时候，这两个变量都设置为MAX_PID，然后last_pid被置为1，进入repeat中， 此时的遍历范围是[1，MAX_PID]，然后当last_pid没有和已有的pid冲突时，缩小next_safe，使last_pid~MAX_PID间都是没被使用的pid值，当冲突时，last_pid+1，并检查[last_pid,MAX_PID]是否合法，如果合法则继续扫描，无需从头开始，如果不合法，则表示小于last_pid的pid已经被用完，这要从头开始扫描[last_pid,MAX_PID]。扫描结束时，last_pid为没有被占用过的pid，可以作新的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）

> 在本实验的执行过程中，创建且运行了几个内核线程?

有两个内核线程:

1） 创建第0个内核线程idleproc。在 init.c::kern_init 函数调用了 proc.c::proc_init 函数。 proc_init 函数启动了创建内核线程的步骤。首先当前的执行上下文（从 kern_init 启动至今）就可以看成是 uCore 内核（也可看做是内核进程）中的一个内核线程的上下文。为此，uCore 通过给当前执行的上下文分配一个进程控制块以及对它进行相应初始化，将其打造成第 0 个内核线程 – idleproc。

2） 创建第 1 个内核线程 initproc。第 0 个内核线程主要工作是完成内核中各个子系统的初始化，然后就通过执行 cpu_idle 函数开始过退休生活了。所以 uCore 接下来还需创建其他进程来完成各种工作，但 idleproc 内核子线程自己不想做，于是就通过调用 kernel_thread 函数创建了一个内核线程 init_main。在Lab4中，这个子内核线程的工作就是输出一些字符串，然后就返回了（参看 init_main 函数）。但在后续的实验中，init_main 的工作就是创建特定的其他内核线程或用户进程。

```
// init_main - the second kernel thread used to create user_main kernel
threads
static int
init_main(void *arg) {
    cprintf("this initproc, pid = %d, name = \"%s\"\n", current->pid,
get_proc_name(current));
    cprintf("To U: \"%s\".\n", (const char *)arg);
    cprintf("To U: \"en.., Bye, Bye. :)\"\n");
    return 0;
}
```

> 语句 `local_intr_save(intr_flag); …local_intr_restore(intr_flag);` 在这里有何作用?请说明理由。

关闭中断和打开中断。

这两个函数实现的意义就是 **避免在进程切换过程中处理中断** 。因为有些过程是互斥的，只允许一个线程进入，因此需要关闭中断来处理临界区；如果此时在切换过程中又一次中断的话，那么该进程保存的值就很可能出bug并且丢失难寻回了。