

# Lab4 内核线程管理

## 一、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。

## 二、实验内容：

### 1、练习0：填写已有实验

利用meld合并即可，不再赘述。

### 2、练习1：分配并初始化一个进程控制块

#### 实验内容：

虽然本次实验是要我们创建内核线程，但是实验指导书上却一直使用"process"和“进程控制块”这两个词，虽说有点不太合适，但是为了统一，除非特殊说明，否则下文中还是采用“process（进程）”和“PCB（进程控制块）”来进行说明。

练习1的目的是创建一个PCB，为了创建一个PCB，我们首先得了解它的数据结构：

```
struct proc_struct {
    enum proc_state state;           // Process state
    int pid;                         // Process ID
    int runs;                        // the running times of Proces
    uintptr_t kstack;               // Process kernel stack
    volatile bool need_resched;     // bool value: need to be
    rescheduled to release CPU?
    struct proc_struct *parent;      // the parent process
    struct mm_struct *mm;           // Process's memory management
    field
    struct context context;          // Switch here to run process
    struct trapframe *tf;           // Trap frame for current
    interrupt
    uintptr_t cr3;                  // CR3 register: the base addr
    of Page Directroy Table(PDT)
    uint32_t flags;                 // Process flag
    char name[PROC_NAME_LEN + 1];  // Process name
    list_entry_t list_link;         // Process link list
    list_entry_t hash_link;        // Process hash list
};
```

其中几乎每一项实验指导书上都已经详细给出了含义，因此这里不再统一给出所有成员变量的含义，后续若使用到其中某项再做单独说明。

按照练习要求，我们进入kern/process/proc.c中，找到alloc\_proc函数，该函数的作用就是初始化一个PCB（并不是初始化一个进程）。注释告诉了我们需要对哪些变量进行初始化，于是完善代码：

```

static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
        //LAB4:EXERCISE1 YOUR CODE
        /*
         * below fields in proc_struct need to be initialized
         *      enum proc_state state;                // Process state
         *      int pid;                               // Process ID
         *      int runs;                               // the running times of
Proces
         *      uintptr_t kstack;                       // Process kernel stack
         *      volatile bool need_resched;             // bool value: need to
be rescheduled to release CPU?
         *      struct proc_struct *parent;             // the parent process
         *      struct mm_struct *mm;                  // Process's memory
management field
         *      struct context context;                // Switch here to run
process
         *      struct trapframe *tf;                  // Trap frame for
current interrupt
         *      uintptr_t cr3;                         // CR3 register: the
base addr of Page Directroy Table(PDT)
         *      uint32_t flags;                         // Process flag
         *      char name[PROC_NAME_LEN + 1];          // Process name
        */
        proc->state = PROC_UNINIT; // 设置进程为“初始”态
        proc->pid = -1; // 设置进程pid的未初始化值
        proc->runs = 0;                // the running times
of Proces
        proc->kstack = 0;                // Process kernel stack
        proc->need_resched = 0;          // bool value: need to be
rescheduled to release CPU?
        proc->parent = NULL;             // the parent process
        proc->mm = NULL;                 // Process's memory management
field
        memset(&(proc->context), 0, sizeof(struct context));
        // Switch here to run process
        proc->tf = NULL;                 // Trap frame for current
interrupt
        proc->cr3 = boot_cr3; // 使用内核页目录表的基址
        proc->flags = 0;                // Process flag
        memset(proc->name, 0, PROC_NAME_LEN); // Process name
        proc->list_link.prev = proc->list_link.next = NULL;
        // Process link list
        proc->hash_link.prev = proc->hash_link.next = NULL;
        // Process hash list
    }
    return proc;
}

```

依据

```
process state    : meaning          -- reason
PROC_UNINIT     : uninitialized     -- alloc_proc
PROC_SLEEPING   : sleeping          -- try_free_pages, do_wait, do_sleep
PROC_RUNNABLE   : runnable(maybe running) -- proc_init, wakeup_proc,
PROC_ZOMBIE     : almost dead       -- do_exit
```

我们知道刚刚创建PCB时，进程状态为PROC\_UNINIT；pid=-1用指导书上的话说叫“身份证号”还没办好，也就是该进程仅有PCB，还并不拥有内存空间；而由于该内核线程在内核中运行，故采用uCore内核已经建立的页表，即设置proc->cr3为uCore内核页表的起始地址boot\_cr3。除了以上三项，由于没有初始化，显然以下的变量该为0就为0，该为NULL就为NULL。

如此便完成了练习1的任务。

## 回答问题：

- 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. 首先是struct context context，我们可以看到结构体context的定义如下：

```
struct context {
    uint32_t eip;
    uint32_t esp;
    uint32_t ebx;
    uint32_t ecx;
    uint32_t edx;
    uint32_t esi;
    uint32_t edi;
    uint32_t ebp;
};
```

实际上就是定义了一些整型变量，通过变量名不难看出这其实就是寄存器的值，结合context的命名显然知道这就是用于进程切换时的上下文保存。

2. 然后是struct trapframe \*tf:

```
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));

```

\*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。这是实验指导书上的解释，我觉得可以这么理解：本实验中我们创建了一个进程initproc，而创建进程的时候cpu的控制权在操作系统，那么当进程创建完毕后我们需要将cpu的控制权转交给新创建的进程，这时候就可以利用中断返回，而返回的地址就是tf所指的内核栈的某个位置，然后恢复出新进程的运行环境。

## 2、练习2：为新创建的内核线程分配资源

### 实验内容：

前面也已经提到过，练习1仅仅是创建了新进程的PCB，但这个新进程本身实际还没有进入内存，而这也就是练习2的任务。

在练习2里，大致流程是proc\_init函数先对idleproc进行创建和初始化，然后调用kernel\_thread，进行一系列操作之后来到do\_fork，完成对initproc的创建及初始化。而我们需要完善的就是kern/process/proc.c中的do\_fork函数。

do\_fork函数主要做了以下6件事情：

1. 分配并初始化进程控制块（alloc\_proc函数）；
2. 分配并初始化内核栈（setup\_stack函数）；
3. 根据clone\_flag标志复制或共享进程内存管理结构（copy\_mm函数）（与本次实验无关）；
4. 设置进程在内核（将来也包括用户态）正常运行和调度所需的中断帧和执行上下文（copy\_thread函数）；
5. 把设置好的进程控制块放入hash\_list和proc\_list两个全局进程链表中；
6. 自此，进程已经准备好执行了，把进程状态设置为“就绪”态；
7. 设置返回码为子进程的id号。

以上几乎也就是注释里面的内容。

```

int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    int ret = -E_NO_FREE_PROC; // E_NO_FREE_PROC是错误码，代表一种出错类型
    struct proc_struct *proc; //定义PCB指针
    if (nr_process >= MAX_PROCESS) { //如果创建的进程数过多，则创建失败
        goto fork_out;
    }
    ret = -E_NO_MEM;
    //LAB4:EXERCISE2 YOUR CODE
    /*
     * Some Useful MACROS, Functions and DEFINES, you can use them in below
     implementation.
     * MACROS or Functions:
     *   alloc_proc:   create a proc struct and init fields (lab4:exercise1)
     *   setup_kstack: alloc pages with size KSTACKPAGE as process kernel stack

```

```

    *   copy_mm:      process "proc" duplicate OR share process "current"'s mm
according clone_flags
    *               if clone_flags & CLONE_VM, then "share" ; else
"duplicate"
    *   copy_thread:  setup the trapframe on the  process's kernel stack top
and
    *               setup the kernel entry point and stack of process
    *   hash_proc:    add proc into proc hash_list
    *   get_pid:      alloc a unique pid for process
    *   wakeup_proc:  set proc->state = PROC_RUNNABLE
    * VARIABLES:
    *   proc_list:    the process set's list
    *   nr_process:   the number of process set
    */

//   1. call alloc_proc to allocate a proc_struct
//   2. call setup_kstack to allocate a kernel stack for child process
//   3. call copy_mm to dup OR share mm according clone_flag
//   4. call copy_thread to setup tf & context in proc_struct
//   5. insert proc_struct into hash_list && proc_list
//   6. call wakeup_proc to make the new child process RUNNABLE
//   7. set ret vaule using child proc's pid
if((proc = alloc_proc()) == NULL) //如果创建PCB失败, 则goto fork_out
    goto fork_out;
proc->parent = current; //创建子进程的父进程为current即当前正在running的进程
if(setup_kstack(proc) != 0) //设立内核栈成功应返回0, 则若不为0说明失败, goto
bad_fork_cleanup_proc, 清理掉之前创建的PCB
    goto bad_fork_cleanup_proc;
if(copy_mm(clone_flags, proc) != 0) //与上同样的道理
    goto bad_fork_cleanup_kstack;
copy_thread(proc, stack, tf); //调用copy_thread函数来设置中断帧指针tf以及context中
部分寄存器的值 (EIP,ESP)
bool intr_flag;
local_intr_save(intr_flag); //关中断
{
    proc->pid = get_pid(); //获取子进程pid
    hash_proc(proc); //加入hash表
    list_add(&proc_list, &(proc->list_link)); //加入进程链表
    nr_process ++; //进程数+1
}
local_intr_restore(intr_flag); //开中断
wakeup_proc(proc); //准备就绪, 唤醒子进程
ret = proc->pid; //返回子进程的pid
fork_out:
    return ret;

bad_fork_cleanup_kstack:
    put_kstack(proc);
bad_fork_cleanup_proc:
    kfree(proc);
    goto fork_out;
}

```

代码中这部分内容会在练习3中详细阐述:

```

local_intr_save(intr_flag);
{
    .....
}
local_intr_restore(intr_flag);

```

## 回答问题:

- 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。  
是的。

这里查看get\_pid函数:

```

// get_pid - alloc a unique pid for process
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;
        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) {
                if (++last_pid >= next_safe) {
                    if (last_pid >= MAX_PID) {
                        last_pid = 1;
                    }
                    next_safe = MAX_PID;
                    goto repeat;
                }
            }
            else if (proc->pid > last_pid && next_safe > proc->pid) {
                next_safe = proc->pid;
            }
        }
    }
    return last_pid;
}

```

首先，一开头注释就说了alloc a unique pid for process；其次，通过分析代码我们知道最终返回的pid是last\_pid这个变量，整个函数可以认为是在寻找一个last\_pid，使得既不存在已有进程的pid: proc->pid==last\_pid，也不存在某个进程的proc->pid在last\_pid和next\_safe之间，这样的话只要选取[last\_pid, next\_safe)这个区间内的值作为pid返回，就可以保证线程id的唯一性了。此外，由于我们是找到了一个合法的id区间，可以想象，当下次创建一个新进程的时候可以直接沿用后面的id值（前提是此时next\_safe > last\_pid+1）。

### 3、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。

#### 实验内容：

首先还是先看一下proc\_run函数：

```
void
proc_run(struct proc_struct *proc) {
    if (proc != current) { //current指向当前正在running的进程的PCB，若proc==current当然就不需要调度了，所以proc!=current才运行以下语句
        bool intr_flag;
        struct proc_struct *prev = current, *next = proc; //定义了两个PCB的指针，perv指向当前进程，next指向准备切换的目标进程
        local_intr_save(intr_flag);
        {
            current = proc; //将切换的目标进程设置为当前正运行的进程
            load_esp0(next->kstack + KSTACKSIZE); //设置任务状态段tss中的特权级0下的esp0指针为next内核线程的内核栈的栈顶
            lcr3(next->cr3); //修改当前的cr3寄存器（页目录表基址），完成进程间的页表切换
            switch_to(&(prev->context), &(next->context)); //调用switch_to进行上下文的保存与切换，切换到新的进程
        }
        local_intr_restore(intr_flag);
    }
}
```

以上是proc\_run函数的内容，下面再来看看其内部的load\_esp0，lcr3以及switch\_to是怎么回事：

```
load_esp0(uintptr_t esp0) {
    ts.ts_esp0 = esp0;
}
```

可以看到，load\_esp0将esp0的值赋给了ts.ts\_esp0，其中esp0为传入参数，而我们在调用的时候传入了next->kstack + KSTACKSIZE，即next指向的内核线程的内核栈的栈顶，于是也就是将ts.ts\_esp0的值设置为该栈顶的地址了。

此外，ts为taskstate类型的结构体变量，而我们可以在mmu.h中找到taskstate的定义：

```
/* task state segment format (as described by the Pentium architecture book) */
struct taskstate {
    uint32_t ts_link;           // old ts selector
    uintptr_t ts_esp0;          // stack pointers and segment selectors（这正是被我们赋值的esp0堆栈指针）
    uint16_t ts_ss0;            // after an increase in privilege level
    uint16_t ts_padding1;
    uintptr_t ts_esp1;
    uint16_t ts_ss1;
    uint16_t ts_padding2;
    uintptr_t ts_esp2;
    uint16_t ts_ss2;
    uint16_t ts_padding3;
```

```

uintptr_t ts_cr3;           // page directory base
uintptr_t ts_eip;           // saved state from last task switch
uint32_t ts_eflags;
uint32_t ts_eax;           // more saved state (registers)
uint32_t ts_ecx;
uint32_t ts_edx;
uint32_t ts_ebx;
uintptr_t ts_esp;
uintptr_t ts_ebp;
uint32_t ts_esi;
uint32_t ts_edi;
uint16_t ts_es;           // even more saved state (segment selectors)
uint16_t ts_padding4;
uint16_t ts_cs;
uint16_t ts_padding5;
uint16_t ts_ss;
uint16_t ts_padding6;
uint16_t ts_ds;
uint16_t ts_padding7;
uint16_t ts_fs;
uint16_t ts_padding8;
uint16_t ts_gs;
uint16_t ts_padding9;
uint16_t ts_ldt;
uint16_t ts_padding10;
uint16_t ts_t;           // trap on task switch
uint16_t ts_iomb;         // i/o map base address
} __attribute__((packed));

```

接下来是lcr3:

```

static inline void
lcr3(uintptr_t cr3) {
    asm volatile ("mov %0, %%cr3" :: "r" (cr3) : "memory");
}

```

这是一行嵌入式汇编代码，其含义是将下一个进程的页目录表起始地址cr3存放在某个寄存器中，然后用mov指令将其写入CR3寄存器中（其中参数cr3正是我们传入的next->cr3）。经过这条语句的处理，CR3就指向新进程next的页目录表，完成了页表的切换。

最后是switch\_to:

```

.globl switch_to
switch_to: # switch_to(from, to)
# save from's registers
movl 4(%esp), %eax # eax points to from
popl 0(%eax) # esp--> return address, so save return addr in FROM's context
movl %esp, 4(%eax)
.....
movl %ebp, 28(%eax)
# restore to's registers
movl 4(%esp), %eax # not 8(%esp): popped return address already
# eax now points to to
movl 28(%eax), %ebp
.....
movl 4(%eax), %esp

```



```
pushl 0(%eax) # push TO's context's eip, so return addr = TO's eip
ret # after ret, eip= TO's eip
```

其实也没什么太多需要说明的，switch\_to也就是字面意思，先save from's registers，然后restore to's registers，通过对寄存器值的修改来完成上下文切换。

## 回答问题：

1. 在本实验的执行过程中，创建且运行了几个内核线程？

一共两个线程：

- idel\_proc: kern\_init函数中调用了proc\_init函数，而在proc\_init里面，我们创建了idel\_proc这个进程并设置其need\_resched为1，当proc\_init函数退出后，在kern\_init的最后会调用cpu\_idel使得idel\_proc开始工作，而它的工作就是查询current->need\_resched值，若为1（现在确实为1），那么就调用schedule()来进行进程的调度；
- init\_proc: 接上文，调用schedule()完成进程调度后，此时执行的进程就变成了init\_proc，而该进程的作用就是输出字符串："Hello world!!"。

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。

首先说结果，local\_intr\_save(intr\_flag)是关中断，local\_intr\_restore(intr\_flag)是开中断，其定义如下：

```
static inline bool
__intr_save(void) {
    if (read_eflags() & FL_IF) {
        intr_disable();
        return 1;
    }
    return 0;
}

static inline void
__intr_restore(bool flag) {
    if (flag) {
        intr_enable();
    }
}

#define local_intr_save(x)      do { x = __intr_save(); } while (0)
#define local_intr_restore(x)  __intr_restore(x);
```

```
/* intr_enable - enable irq interrupt */
void
intr_enable(void) {
    sti();
}

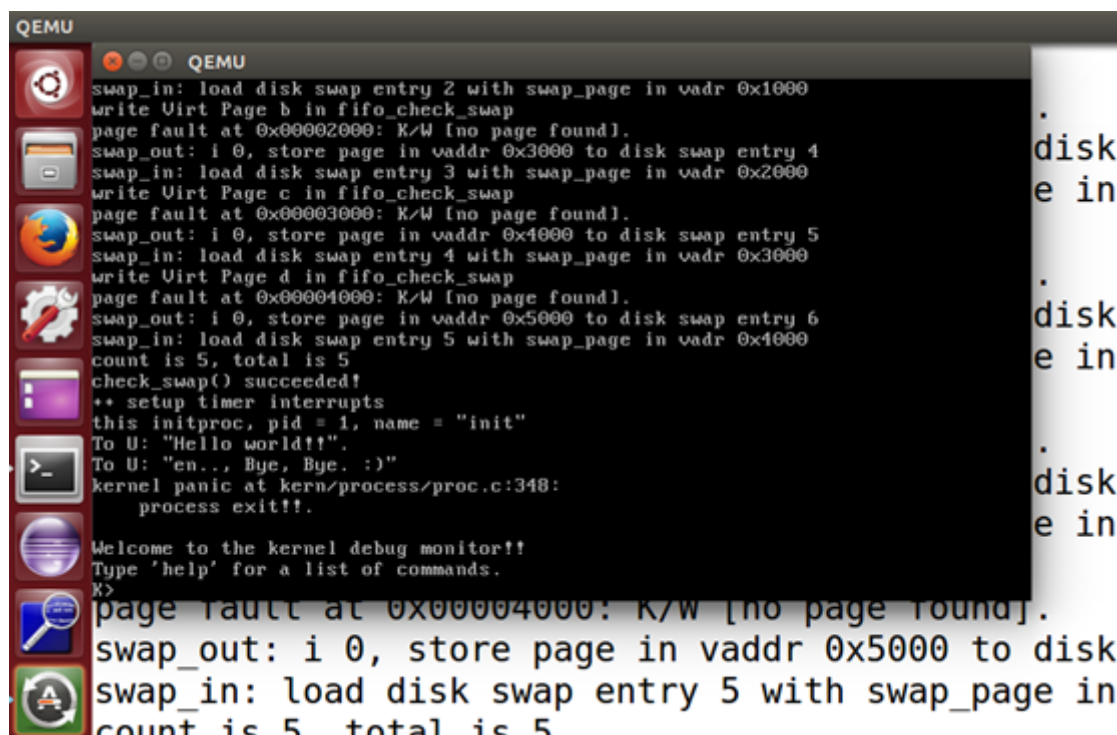
/* intr_disable - disable irq interrupt */
void
intr_disable(void) {
    cli();
}
```

注意：汇编指令cli可以清除IF标志（IF=0），而汇编指令sti可以设置IF标志（IF=1），如此完成了关中断到开中断的操作。

至于为什么要这么做，那是因为我们保证进程切换是一个原子操作，即在进行进程切换的时候，我们不希望这个过程被打断，因为一旦被打断，会出现严重的后果，比如proc\_run中 `current = proc` 这句代码，如果在执行过程中被中断，会导致进程切换后current并没有指向当前进程的PCB。

### 三、实验结果：

make qemu编译运行：



```
QEMU
swap_in: load disk swap entry 2 with swap_page in vadr 0x1000
write Virt Page b in fifo_check_swap
page fault at 0x00002000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x3000 to disk swap entry 4
swap_in: load disk swap entry 3 with swap_page in vadr 0x2000
write Virt Page c in fifo_check_swap
page fault at 0x00003000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x4000 to disk swap entry 5
swap_in: load disk swap entry 4 with swap_page in vadr 0x3000
write Virt Page d in fifo_check_swap
page fault at 0x00004000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x5000 to disk swap entry 6
swap_in: load disk swap entry 5 with swap_page in vadr 0x4000
count is 5, total is 5
check_swap() succeeded!
** setup timer interrupts
this initproc, pid = 1, name = "init"
To U: "Hello world!!".
To U: "en.., Bye, Bye. :)"
kernel panic at kern/process/proc.c:348:
process exit!!
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
page fault at 0x00004000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x5000 to disk
swap_in: load disk swap entry 5 with swap_page in
count is 5 total is 5
```

### 四、实验总结：

做第二次实验时明显要比做第一次实验要得心应手不少，做起来速度也快得多。个人认为一方面是因为理论课正好讲到这部分内容，另一方面就是实验4中涉及到的一些基本的数据结构及操作，以及内存管理的相关概念在之前的实验中已经有过接触，所以总的来说本次实验做起来还是不难的。

不过，我最初写完代码运行的时候，系统报错，然后直接去看了答案，发现跟我唯一不同的地方就是这样一段话：

```
local_intr_save(intr_flag);
{
    .....
}
local_intr_restore(intr_flag);
```

然而当时我也不知道这段代码是在做什么，随后查了资料才明白这是对中断的操作。想来因为创建一个进程为一个原语，不能被中断，自然是要关中断的，不过确实不是很理解练习2为什么是在那个时间点关中断，为什么不一开始就关中断，直到创建结束再恢复中断。

最后，通过实验，一方面确实让我了解了PCB中都有一些什么东西，并且起到什么作用；另一方面，也是明白了创建一个进程，从创建PCB到分配内存，再到最后run的整个过程是怎么回事。

