# 实验项目：内核线程管理

姓名：张伟焜　学号：17343155　邮箱：zhangwk8@mail2.sysu.edu.cn
院系：数据科学与计算机学院　专业：17 级软件工程　指导教师：张永东

**【实验题目】**

内核线程管理

**【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

**【实验要求】**

根据指导，完成练习 0~3。

**【实验方案】**

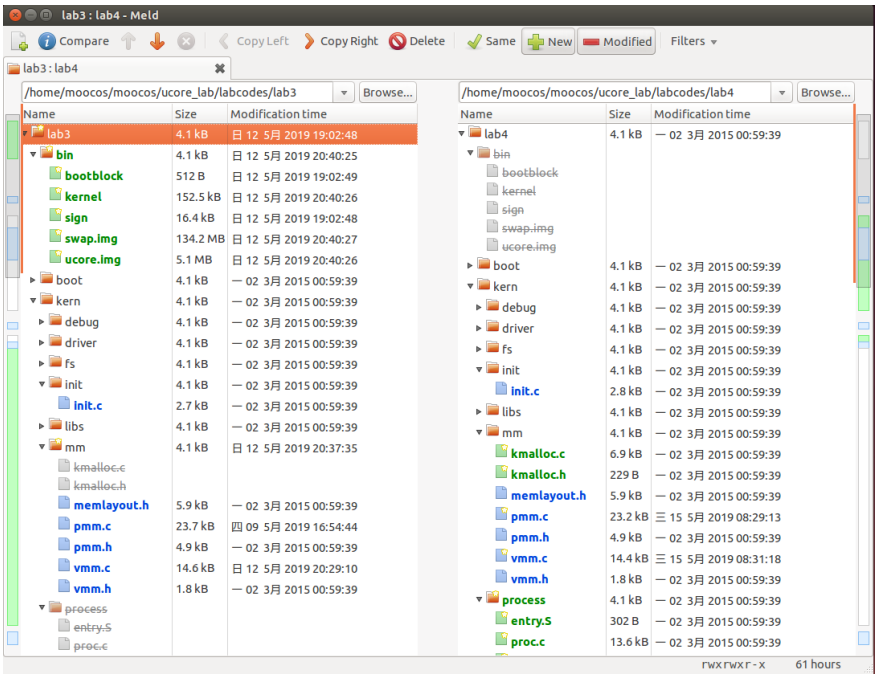
实验环境：老师提供的虚拟机（Virtual box），无特殊硬件要求

实验思路：根据实验指导，先了解理论知识，再进行实验

**【实验过程】**

**练习 0：填写已有实验。**

使用 meld 软件将 ucore 启动实验的代码导入。




注意要点击标星文件进行对比，将上次实验完成的函数复制过来，不要将整个文件进行覆盖。之前实验修改的内容主要在 kdebug.c trap.c pmm.c default_pmm.c vmm.c swap_fifo.c 。

**练习 1：分配并初始化一个进程控制块。**

*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。*

*【提示】在 alloc_proc 函数的实现中，需要初始化的 proc_struct 结构中的成员变量至少包括：state/pid/runs/kstack/need_resched/parent/mm/context/tf/cr3/flags/name。*

*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 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）*

**分析：**

该函数的具体功能为创建一个新的进程控制块，并且对控制块中的所有成员变量进行初始化，除了指定的若干个成员变量之外，其他成员变量均初始化为 0。

实验给出的注释如下：

```
89    //LAB4:EXERCISE1 YOUR CODE
90    /*
91     * below fields in proc_struct need to be initialized
92     *       enum proc_state state;                      // Process state
93     *       int pid;                                    // Process ID
94     *       int runs;                                   // the running times of Proces
95     *       uintptr_t kstack;                           // Process kernel stack
96     *       volatile bool need_resched;                 // bool value: need to be rescheduled to release
   CPU?
97     *       struct proc_struct *parent;                 // the parent process
98     *       struct mm_struct *mm;                       // Process's memory management field
99     *       struct context context;                     // Switch here to run process
100    *       struct trapframe *tf;                       // Trap frame for current interrupt
101    *       uintptr_t cr3;                              // CR3 register: the base addr of Page Directroy
   Table(PDT)
102    *       uint32_t flags;                             // Process flag
103    *       char name[PROC_NAME_LEN + 1];               // Process name
104    */
```

结合分析与注释，得到代码：

```
// alloc_proc - alloc a proc_struct and init all fields of proc_struct
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
        //初始化为特殊值
        proc->state = PROC_UNINIT; //进程状态初始化
        proc->cr3 = boot_cr3;      //初始化页目录为内核页目录表的基址
        proc->pid = -1;            //进程pid初始化为-1
        //初始化为0
        proc->runs = 0;            //初始化时间片
        proc->kstack = 0;          //初始化内核栈地址
        proc->need_resched = 0;    //初始化不需要调度
        proc->parent = NULL;       //初始化父进程为空
        proc->mm = NULL;           //初始化虚拟内存为空
        memset(&(proc->context), 0, sizeof(struct context));  //初始化上下文
        proc->tf = NULL;           //初始化中断帧指针为空
        proc->flags = 0;           //初始化标志位为0
        memset(proc->name, 0, PROC_NAME_LEN);//初始化进程名为0
    }
```

```
    return proc;
}
```

**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）**

proc_struct 结构体的定义如下：

```
42 struct proc_struct {
43     enum proc_state state;                      // Process state
44     int pid;                                    // Process ID
45     int runs;                                   // the running times of Proces
46     uintptr_t kstack;                           // Process kernel stack
47     volatile bool need_resched;                 // bool value: need to be rescheduled to release CPU?
48     struct proc_struct *parent;                 // the parent process
49     struct mm_struct *mm;                       // Process's memory management field
50     struct context context;                     // Switch here to run process
51     struct trapframe *tf;                       // Trap frame for current interrupt
52     uintptr_t cr3;                              // CR3 register: the base addr of Page Directroy Table(PDT)
53     uint32_t flags;                             // Process flag
54     char name[PROC_NAME_LEN + 1];               // Process name
55     list_entry_t list_link;                     // Process link list
56     list_entry_t hash_link;                     // Process hash list
57 };
```

context 结构体定义如下

```
18 // Saved registers for kernel context switches.
19 // Don't need to save all the %fs etc. segment registers,
20 // because they are constant across kernel contexts.
21 // Save all the regular registers so we don't need to care
22 // which are caller save, but not the return register %eax.
23 // (Not saving %eax just simplifies the switching code.)
24 // The layout of context must match code in switch.S.
25 struct context {
26     uint32_t eip;
27     uint32_t esp;
28     uint32_t ebx;
29     uint32_t ecx;
30     uint32_t edx;
31     uint32_t esi;
32     uint32_t edi;
33     uint32_t ebp;
34 };
```

查看结构体代码，可以发现结构体中存储了除 eax 之外的所有通用寄存器以及 eip 的值，并结合注释“Saved registers for kernel context switches”,表明这个线程控制块中的 context 是保存的线程运行的上下文信息。

tf：结合 proc_struct 结构体注释“Trap frame for current interrupt.”,可以得出 tf 为中断帧的指针。tf 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。由于 uCore 内核允许嵌套中断，为了保证嵌套中断发生时 tf 总是能够指向当前的 trapframe，uCore 在内核栈上维护了 tf 链。

**练习 2：为新创建的内核线程分配资源。**

*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。*

*do_fork 的大致执行步骤包括：*

*1) 调用 alloc_proc，首先获得一块用户信息块。*
*2) 为进程分配一个内核栈。*
*3) 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）*
*4) 复制原进程上下文到新进程*
*5) 将新进程添加到进程列表*
*6) 唤醒新进程*
*7) 返回新进程号*

**分析：**

do_fork 用于创建新的内核线程。它涉及到许多虚函数的调用，如 alloc_proc, setup_kstack 等（具体涉及到的函数可以看下面的注释提示），它创建内核线程的一个副本给新的内核线程分配资源，并且复制原进程的状态，使得之后可以正确切换到对应的线程中执行。

实验给出的注释如下:

```
//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
 *   wakup_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 wakup_proc to make the new child process RUNNABLE
//    7. set ret vaule using child proc's pid
```

结合分析与注释，得到代码:

```
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;

    if ((proc = alloc_proc()) == NULL) { //若内存分配失败
        goto fork_out;
    }
    proc->parent = current; //设置父进程
    if (setup_kstack(proc) != 0) {//为新进程分配栈
        goto bad_fork_cleanup_proc;
    }
```

```
    if (copy_mm(clone_flags, proc) != 0) { //对虚拟内存空间进行拷贝
        goto bad_fork_cleanup_kstack;
    }
    copy_thread(proc, stack, tf); //复制中断帧和上下文信息
    bool intr_flag;
    local_intr_save(intr_flag);  //intr_flag设为1，屏蔽中断
    {
        proc->pid = get_pid(); //获取当前进程pid
        hash_proc(proc);  //建立hash映射
        list_add(&proc_list, &(proc->list_link));//将新进程加入进程链表
        nr_process++;  //进程数加1
    }
    local_intr_restore(intr_flag); //恢复中断
    wakeup_proc(proc); //唤醒新进程
    ret = proc->pid; //返回当前进程的pid
fork_out:
    return ret;

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

**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。**

可以做到给每个新 fork 的线程一个唯一的 id。

程序中使用 get_pid()来为新线程分配 pid，该函数代码如下:

```
// 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;
}
```

我们可以看出有两个静态的局部变量 next_safe 和 last_pid，这两个变量的数值之间的取值均是合法的 pid。

如果有 next_safe > last_pid + 1，那么直接取 last_pid + 1 作为新的 pid。

如果 next_safe > last_pid + 1 不成立，则进入循环，if (proc->pid == last_pid)代码块确保了不存在任何进程的 pid 与 last_pid 相同； if (proc->pid > last_pid && next_safe > proc->pid)保证了不存在任何已经存在的 pid 满足：last_pid<pid<next_safe，保证最后能够找到一个满足条件的区间，获得合法的 pid。

若 last_pid 超出 MAX_PID，会置为 1。

这样在就唯一地分配了一个 PID。

**练习 3：分析代码: proc_run 函数。**

**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。**

**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。**

对 proc_run 的代码增加注释：

```
// proc_run - make process "proc" running on cpu
// NOTE: before call switch_to, should load  base addr of "proc"'s new PDT
void
proc_run(struct proc_struct *proc) {
    if (proc != current) { //proc是否已经正在运行
        bool intr_flag;
        struct proc_struct *prev = current, *next = proc;
        local_intr_save(intr_flag);//关中断
        {
```

```
            current = proc;
            load_esp0(next->kstack + KSTACKSIZE);
            lcr3(next->cr3);//修改cr3为需要运行线程（进程）的页目录表
            switch_to(&(prev->context), &(next->context));//切换到新线程
        }
        local_intr_restore(intr_flag);//开中断
    }
}
```

proc_run 函数将当前 CPU 的控制权交给指定的线程。

首先屏蔽中断，接着修改 esp0 和页表项，然后调用 swtich_to 函数切换线程，switch_to 函数主要完成的是进程的上下文切换，先保存当前寄存器的值，然后再将下一进程的上下文信息保存到对于寄存器中。最后开中断。

在本实验的执行过程中，创建且运行了两个内核线程。

（1）idleproc：第一个内核线程，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，之后立即调度执行其他线程；

（2）initproc：被创建用于打印"Hello World"的线程；

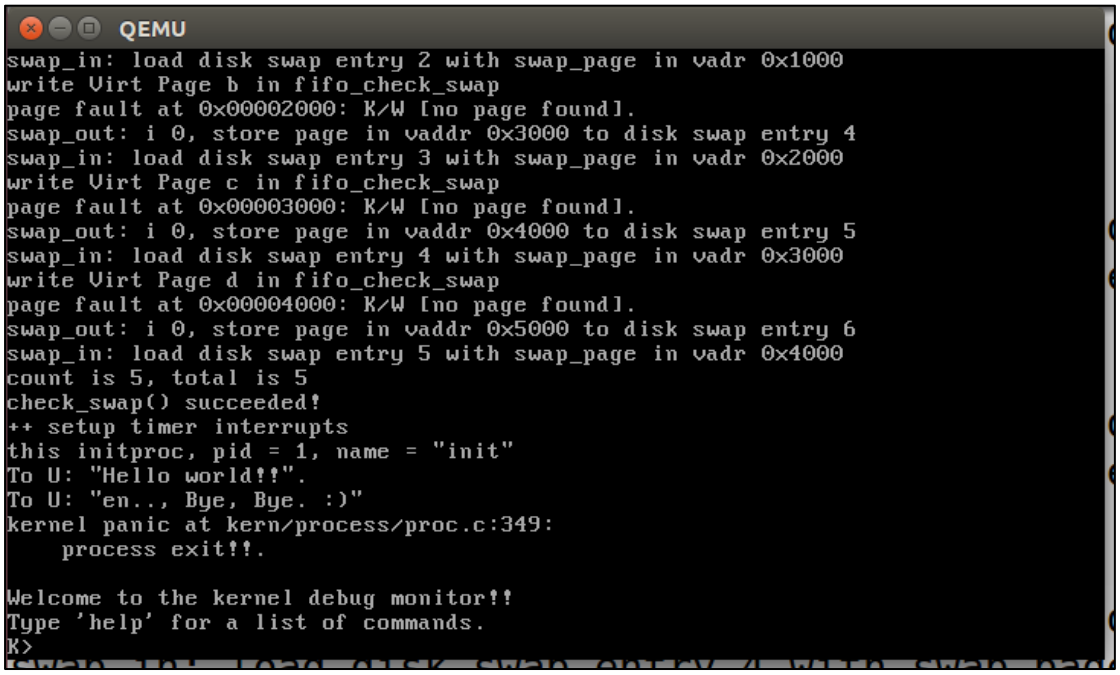
local_intr_save(intr_flag);　关中断，防止

local_intr_restore(intr_flag);　开中断

两者使得之间的语句不会被再次中断，是一个原子操作，能够避免进程切换时其他进程再进行调度。

**运行结果：**

**make qemu:**



make grade:

```
zhangweikun$>make grade
Check VMM:                (3.0s)
  -check pmm:                                  OK
  -check page table:                           OK
  -check vmm:                                  OK
  -check swap page fault:                      OK
  -check ticks:                                OK
  -check initproc:                             OK
Total Score: 90/90
zhangweikun$>
```

**【实验总结】**

**完成实验后，请分析 ucore_lab 中提供的参考答案，并请在实验报告中说明你的实现与参考答案的区别**

结合注释打代码，注释中每一步都很详细，按照注释打下来几乎和答案一模一样，有些语句的顺序不一样，但最终运行效果是一样的。

最后在写实验报告的时候给代码加上了注释。

**列出你认为本实验中重要的知识点，以及与对应的 OS 原理中的知识点，并简要说明你对二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知识点）**

1.实验涉及到线程控制块的概念以及组成。在操作系统中学习过内核线程经常被称之为内核守护进程。内核线程是被调度的实体，它被加入到某种数据结构中，调度程序根据实际情况进行线程的调度。内核线程与用户态线程的作用类似，通常用于执行某些周期性的计算任务，或者在后台执行需要大量计算的任务。

2.实验设计到切换不同线程的方法。操作系统中学习过内核线程间的切换。两者是相通的。

**列出你认为 OS 原理中很重要，但在实验中没有对应上的知识点**

1.内核线程与用户线程的多对多模型。

2.线程池的概念

**心得体会**

通过本次实验，我对内核线程管理有了更深入的认识，本次实验中也涉及到了一些错误处理操作，比如考虑开关中断来处理中断打断当前的操作可能会引起的错误。这需要我们更深入、周全地考虑问题。实验涉及的编程任务不多，但是每个分析问题都很不错，帮助我更深入地了解了知识的应用。此外，回答本次实验的一个问题，可能会牵扯到许多不同的函数，对我们阅读代码的能力有了一定的要求，这是我在之后要努力的方向，要学会结合注释更加熟练地阅读代码，理清不同函数之间的逻辑关系。

**【参考文献】**

*《操作系统实验指导(清华大学)陈渝、向勇编著》*