操作系统原理实验

实验五（Ucore Lab4） - 内核线程管理

实验准备

> 主机环境：macOS X 10.14.4 Mojave
>
> 虚拟机环境：Linux Ubuntu 14.04 LTS
>
> 虚拟机搭载软件：Parallels Desktop.app
>
> 命令行终端：Linux 下 Terminal
>
> 终端shell：zsh
>
> 之前的实验二中，实验用虚拟机硬盘文件.vdi搭建的虚拟机环境存在兼容性不好，卡顿严重，分辨率也不高等问题。故从这次实验开始，在Github上将原实验项目clone下来，自己按照指导书第一章讲的环境配置，安装所需支持，配置实用工具进行实验。
>
> 为满足实验要求，将命令行用户名字段临时指定为姓名。（需要在zsh主题文件中修改显示。因为中文姓名在命令行终端会影响显示效果，故用我的中大NetID作为署名标记：许滨楠 - xubn。）

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

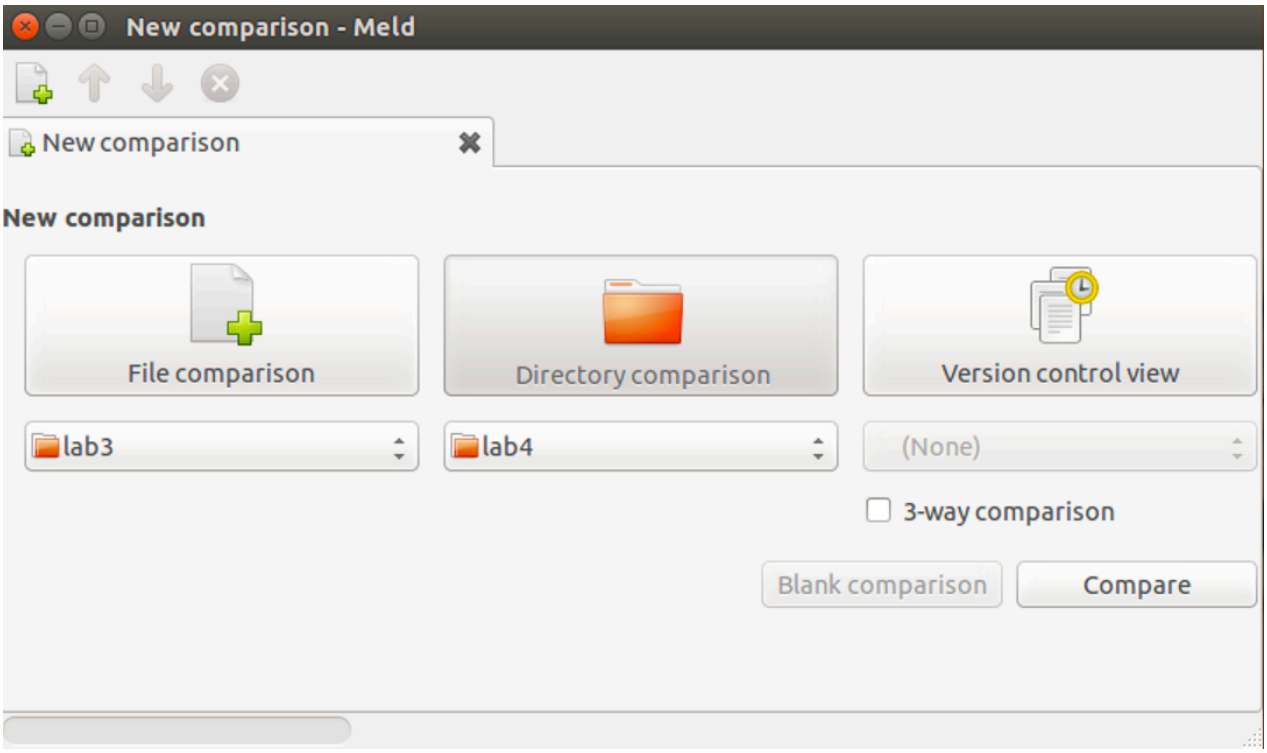
> - 实验将接触的是内核线程的管理。内核线程是一种特殊的进程，内核线程与用户进程的区别有两个：
>   - 内核线程只运行在内核态；用户进程会在用户态和内核态交替运行；
>   - 所有内核线程共用ucore内核内存空间，不需要为每个内核线程维护单独的内存空间；而用户进程需要维护各自的用户内存空间。

练习

练习0

> 填写已有实验。

利用 meld 工具，将之前完成的 lab1/2/3 代码 merge 到 lab4 的代码中。因为之前已经将 lab1/2 的代码整合到 lab3 ，所以这次直接将 lab3 和 lab4 的代码在 meld 中进行目录比较，合并已经完成的代码。

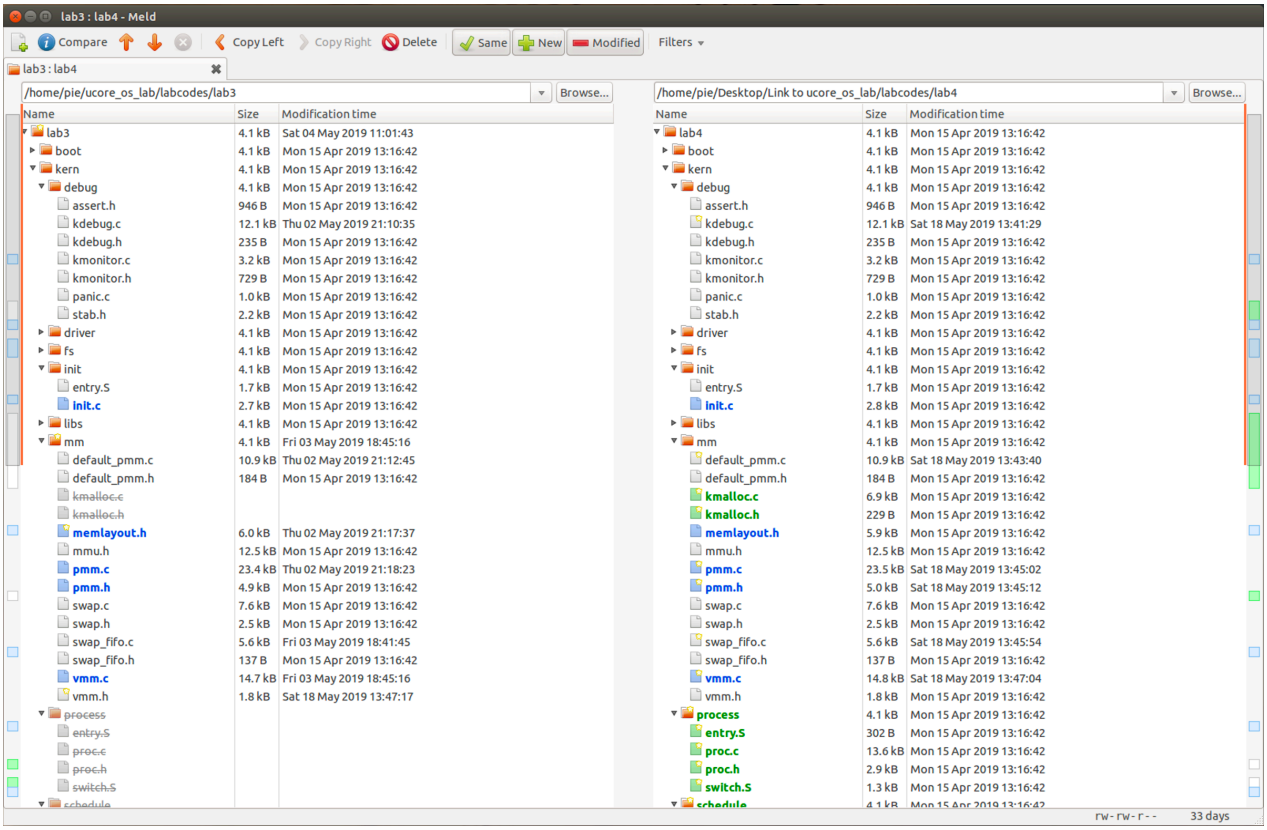


主要在之前编写过代码的文件中做比较，补充填入已完成的代码。具体需要填写的文件和函数如下：

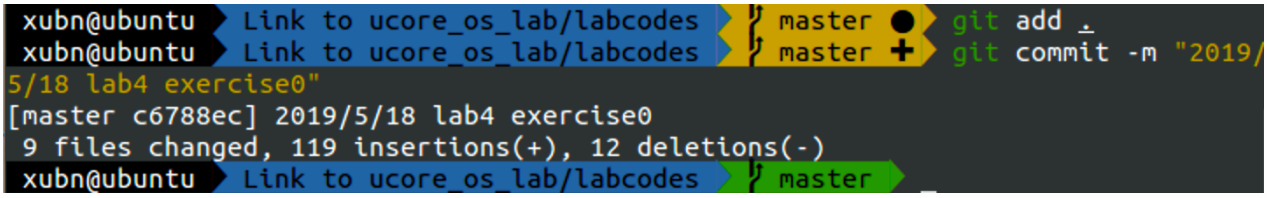
- Lab 1
  - kdebug.c : print_stackframe
  - trap.c : idt_init
  - trap.c : trap_dispatch
- Lab 2
  - default_pmm.c : default_init
  - default_pmm.c : default_init_memmap
  - default_pmm.c : default_alloc_pages

- default_pmm.c : default_free_pages
- pmm.c : get_pte
- pmm.c : page_remove_pte

- Lab 3
  - vmm.c : do_pgfault
  - swap_fifo.c : __fifo_map_swappable
  - swap_fifo.c : __fifo_swap_out_victim



先前操作的文件基本比对合并完毕，练习前的准备完成，将当前的代码进行commit保存到版本管理仓库以备不时之需。接着进行后续练习。



练习1

> 分配并初始化一个进程控制块。

理解与设计

本练习的重点是 **/kern/process/proc.c** 中的 **alloc_proc** 函数。该函数的功能主要为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。练习中要实现的功能是对这个结构进行基本的初始化，初始化的成员变量至少包括：**state / pid / runs / kstack / need_resched / parent / mm / context / tf / cr3 / flags / name** 。

先看到对应目录中的函数原型：

```
// 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) {
    //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
     */
    }
    return proc;
}
```

函数中对我们要进行初始化的量进行了比较详细的注释说明，基本上各个变量的含义和作用如下：

- enum proc_state state：用于标识所记录进程的状态，其枚举类型在 **proc.h** 中列出：

```
enum proc_state {
    PROC_UNINIT = 0,  // uninitialized
    PROC_SLEEPING,    // sleeping
    PROC_RUNNABLE,    // runnable(maybe running)
    PROC_ZOMBIE,      // almost dead, and wait parent proc to
                      // reclaim his resource
};
```

可知在函数中实现初始化时，应该将其赋值为 0 PROC_UNINIT 表示未初始化状态；

- int pid：标识进程 id，初始应该赋值为 -1 表示尚未分配；
- int runs：记录已经运行的次数，初始化的时候尚未运行过，赋值为0；
- uintptr_t kstack：记录内核堆栈(kernel stack)的起始地址，初始状态下堆栈尚未分配，应该赋值为 0；
- volatile bool need_resched：标识当前进程是否需要调度，初始为不需要(0)；
- struct proc_struct *parent：记录当前进程的父进程，初始化为 NULL；
- struct mm_struct *mm：记录维护当前的内存空间，初始化为 NULL；
- struct context context：与后续内容相关，但此处影响不大，默认同前初始化为 0；
- struct trapframe *tf：记录当前的中断帧，初始化为 NULL；
- uintptr_t cr3：记录当前进程页表的基地址，初始状态下未分配，赋值为 kernel 的页表基地址；
- uint32_t flags：标识当前进程的属性状态的标志位，初始置 0；
- char name[PROC_NAME_LEN + 1]：保存当前进程的名称，初始清零。

实现与代码

清楚了以上参量的含义和初始化方式之后，着手实现该函数，通过 **kmalloc** 函数分配获取一个新的 **proc_struct** 之后将其中参量初始化赋值，补充完整后整个函数的具体代码如下：

```
// 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) {
        proc->state = PROC_UNINIT;  // state to be uninitialized
        proc->pid = -1;             // id to be -1 before
        proc->runs = 0;             // have run for 0 time
        proc->kstack = 0;           // doesn't have stack allocated yet
        proc->need_resched = 0;     // doesn't need to be rescheduled
        proc->parent = NULL;        // doesn't have parent by default
        proc->mm = NULL;            // doesn't have memory allocated yet
        memset(&(proc->context), 0, sizeof(struct context));
        proc->tf = NULL;            // doesn't have trap frame by default
        proc->cr3 = boot_cr3;       // assign to the kernel PDT by
default
        proc->flags = 0;            // flags set to be 0
        memset(proc->name, 0, PROC_NAME_LEN);   // name set to void
    }
    return proc;
}
```

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。（通过看代码和编程调试可以判断出来）

在实验的指导书P115中对成员变量的解读提及：

context：进程的上下文，用于进程切换（参见 **switch.S**）。在 ucore 中，所有的进程在内核中也是相对独立的（例如：独立的内核堆栈以及上下文等）。使用 **context** 保存寄存器的目的就在于在内核状态下能够进行上下文之间的切换。实际利用 **context** 进行上下文切换的函数是在 **kern/process/switch.S** 中定义的 **switch_to**。

tf：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转到内核空间时，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。除此之外，ucore 内核允许嵌套中断。因此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** 的链，可以参考 **trap.c** : **trap** 函数做进一步了解。

再结合指导书的解释和其中提及的参照代码，大致可以做如下解读和概括：

struct context context : 用于保存进程的上下文。在进程切换过程中， **context** 相当于上一个进程的容器，保存上一个进程相关的寄存器情况。作用是使内核态中的进程在调度切换的时候能完成其间上下文的保存、载入。

struct trapframe *tf : 是中断帧的指针。它指向内核栈中的某个位置，当进程从用户空间跳转到内核空间时，**tf** 记录该进程在被中断之前的具体状态（寄存器信息），当进程跳回用户空间，通过获取该指针中的信息和寄存器值，可以让进程正常恢复。即中断发生时的保存现场和完成后的恢复现场。

练习2

> 为新创建的内核线程分配资源。

理解与设计

这个练习主要的函数 **do_fork** 会在内核线程创建的时候，由 **kernel_thread** 函数进行调用。其中待完成部分的注释如下：

```
//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
 *   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
```

结合指导课件和指导书的提醒，**do_fork** 在进行对内核线程的资源分配时大体的步骤如下：

- 调用之前编写的 **alloc_proc** 函数，获得一块可用的已经初始化后的用户信息块
- 调用 **setup_kstack** 函数，为进程分配一个内核堆栈；
- 调用 **copy_mm** 函数，将原进程的内存管理信息复制到新进程；
- 调用 **copy_thread** 函数，将原进程上下文复制到新进程；
- 将新建进程以前述提及的 **proc_struct** 形式添加到 **hash_list && proc_list** 中；
- 调用 **wakeup_proc** 函数，将新的子进程唤醒；
- 将新建进程的进程号 **pid** 作为函数的返回值。

实现与代码

有了上面的知识基础和接口说明，代码就不难实现了，基本是函数的调用和一些值的设置，完成代码如下：

```
1  int
2  do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
3      int ret = -E_NO_FREE_PROC;
4      struct proc_struct *proc;
5      if (nr_process >= MAX_PROCESS) {
6          goto fork_out;
7      }
8      ret = -E_NO_MEM;
9      //    1. call alloc_proc to allocate a proc_struct
10     if (NULL == (proc = alloc_proc())) {
11         goto fork_out;
12     }
13     proc->parent = current;
14     //    2. call setup_kstack to allocate a kernel stack for child
   process
15     if (setup_kstack(proc) != 0) {
```

```
        goto bad_fork_cleanup_proc;
    }
    //    3. call copy_mm to dup OR share mm according clone_flag
    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }
    //    4. call copy_thread to setup tf & context in proc_struct
    copy_thread(proc, stack, tf);
    //    5. insert proc_struct into hash_list && proc_list
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        proc->pid = get_pid();
        hash_proc(proc);
        list_add(&proc_list, &(proc->list_link));
        ++nr_process;
    }
    local_intr_restore(intr_flag);
    //    6. call wakeup_proc to make the new child process RUNNABLE
    wakeup_proc(proc);
    //    7. set ret vaule using child proc's pid
    ret = proc->pid;
fork_out:
    return ret;

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

问题解答

> 请说明 ucore 是否做到给每个新 fork 的进程一个唯一的 id？ 说明分析和理由。

ucore做到了给每个新 fork 的线程一个唯一的 id。阅读之前 **do_fork** 函数中调用的 **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;
}
```

可知该函数通过查看当前的全部进程，避免 **pid** 的重复使用，在分配层面上保证了每个进程有唯一的 **pid**。

还有一个问题是如果多个进程同时被创建，会不会被分配到相同的 **pid** 。ucore 的实现中目前还是单核状态，参照示例答案代码可知，在之前 **get_pid** 函数调用并将进程块插入链表的时候代码实现了关中断的操作，在单核的 ucore 中，关中断就可以保证若有多个进程同时申请获取进程号，则他们的获取操作是互斥的。

这样，就保证了每个新 fork 的进程都会获得唯一的 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。

代码理解

首先定位到 **/kern/process/proc.c** 中的 **proc_run** 函数：

```
// 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) {
        bool intr_flag;
```

```
        struct proc_struct *prev = current, *next = proc;
        local_intr_save(intr_flag);
        {
            current = proc;
            load_esp0(next->kstack + KSTACKSIZE);
            lcr3(next->cr3);
            switch_to(&(prev->context), &(next->context));
        }
        local_intr_restore(intr_flag);
    }
}
```

从代码上看，该函数的执行过程如下：

- 函数调用表示要从当前进程切换到参数传递 **proc** 所表示进程，**proc** 不为 **current** 当前进程时进行切换；
- 关闭中断，保证操作的原子性，防止切换过程中被打断带来无法控制的效果；
- 设置当前 PCB **current** 为目的进程的 PCB **proc** ，设置任务状态段 task state **ts** 的栈顶指针 esp0 为 next 的内核栈栈顶；
- 设置进程页表基址指针 cr3 指向将要切换的进程的页目录表起始地址，完成进程间的页表切换；
- 通过汇编 switch_to 函数完成两个进程的 CPU 现场切换，切换各个寄存器的值，这里再分析一下汇编函数源代码及其运行过程和切换方式：

```
.text
.globl switch_to
switch_to:                      # switch_to(from, to)

    # save from's registers
    movl 4(%esp), %eax          # eax points to from
    popl 0(%eax)                # save eip !popl
    movl %esp, 4(%eax)          # save esp::context of from
    movl %ebx, 8(%eax)          # save ebx::context of from
    movl %ecx, 12(%eax)         # save ecx::context of from
    movl %edx, 16(%eax)         # save edx::context of from
    movl %esi, 20(%eax)         # save esi::context of from
    movl %edi, 24(%eax)         # save edi::context of from
    movl %ebp, 28(%eax)         # save ebp::context of from

    # restore to's registers
    movl 4(%esp), %eax          # not 8(%esp): popped return address already
                                # eax now points to to
    movl 28(%eax), %ebp         # restore ebp::context of to
    movl 24(%eax), %edi         # restore edi::context of to
    movl 20(%eax), %esi         # restore esi::context of to
    movl 16(%eax), %edx         # restore edx::context of to
    movl 12(%eax), %ecx         # restore ecx::context of to
    movl 8(%eax), %ebx          # restore ebx::context of to
    movl 4(%eax), %esp          # restore esp::context of to

```

```
    pushl 0(%eax)                # push eip

    ret
```

前面的 movl 相关操作都是对各寄存器现场的保存和切换，但 **popl 0(%eax)** 一句中将保存在寄存器 eax 中的 esp ，即原进程的指令地址弹出，后面 **pushl 0(%eax)** 又将下一个进程要执行的指令地址推入堆栈顶，在执行完 **ret** 语句之后，再从栈顶 **esp** 中指针指向的地址执行，其实就是 **context** 中保存的指令地址 **eip**了。这样就完成了从原进程到新进程的执行切换。

问题解答

> 在本实验的执行过程中，创建且运行了几个内核线程?

首先看到 **proc_init** 函数的执行过程：

```
void
proc_init(void) {
    int i;

    list_init(&proc_list);
    for (i = 0; i < HASH_LIST_SIZE; i ++) {
        list_init(hash_list + i);
    }

    if ((idleproc = alloc_proc()) == NULL) {
        panic("cannot alloc idleproc.\n");
    }

    idleproc->pid = 0;
    idleproc->state = PROC_RUNNABLE;
    idleproc->kstack = (uintptr_t)bootstack;
    idleproc->need_resched = 1;
    set_proc_name(idleproc, "idle");
    nr_process ++;

    current = idleproc;

    int pid = kernel_thread(init_main, "Hello world!!", 0);
    if (pid <= 0) {
        panic("create init_main failed.\n");
    }

    initproc = find_proc(pid);
    set_proc_name(initproc, "init");

    assert(idleproc != NULL && idleproc->pid == 0);
    assert(initproc != NULL && initproc->pid == 1);
}
```

- 初始化 **proc_list**, **hash_list**链表之后，调用 **alloc_proc** 函数分配 PCB 块，进行检验；
- 对分配好的 **idleproc** 进行基本的设置，包括之前提到分配时初始化的函数中设置的许多内容；
- 将记录当前运行进程的变量 **current** 的值置为该进程；
- 调用 **kernel_thread** 函数，用 **init_main** 接口创建一个内核线程；

```
int
kernel_thread(int (*fn)(void *), void *arg, uint32_t clone_flags) {
    struct trapframe tf;
    memset(&tf, 0, sizeof(struct trapframe));
    tf.tf_cs = KERNEL_CS;
    tf.tf_ds = tf.tf_es = tf.tf_ss = KERNEL_DS;
    tf.tf_regs.reg_ebx = (uint32_t)fn;
    tf.tf_regs.reg_edx = (uint32_t)arg;
    tf.tf_eip = (uint32_t)kernel_thread_entry;
    return do_fork(clone_flags | CLONE_VM, 0, &tf);
}
```

  - 创建 **trapframe** 并初始化；
  - 调用 **do_fork** 函数创建新的进程；
    - 调用 **alloc_proc** ，取得新块、为新进程分配内核栈、相关信息和必要数据的复制、添加新进程到链表、唤醒、返回新进程号
- 检验进程均创建成功。

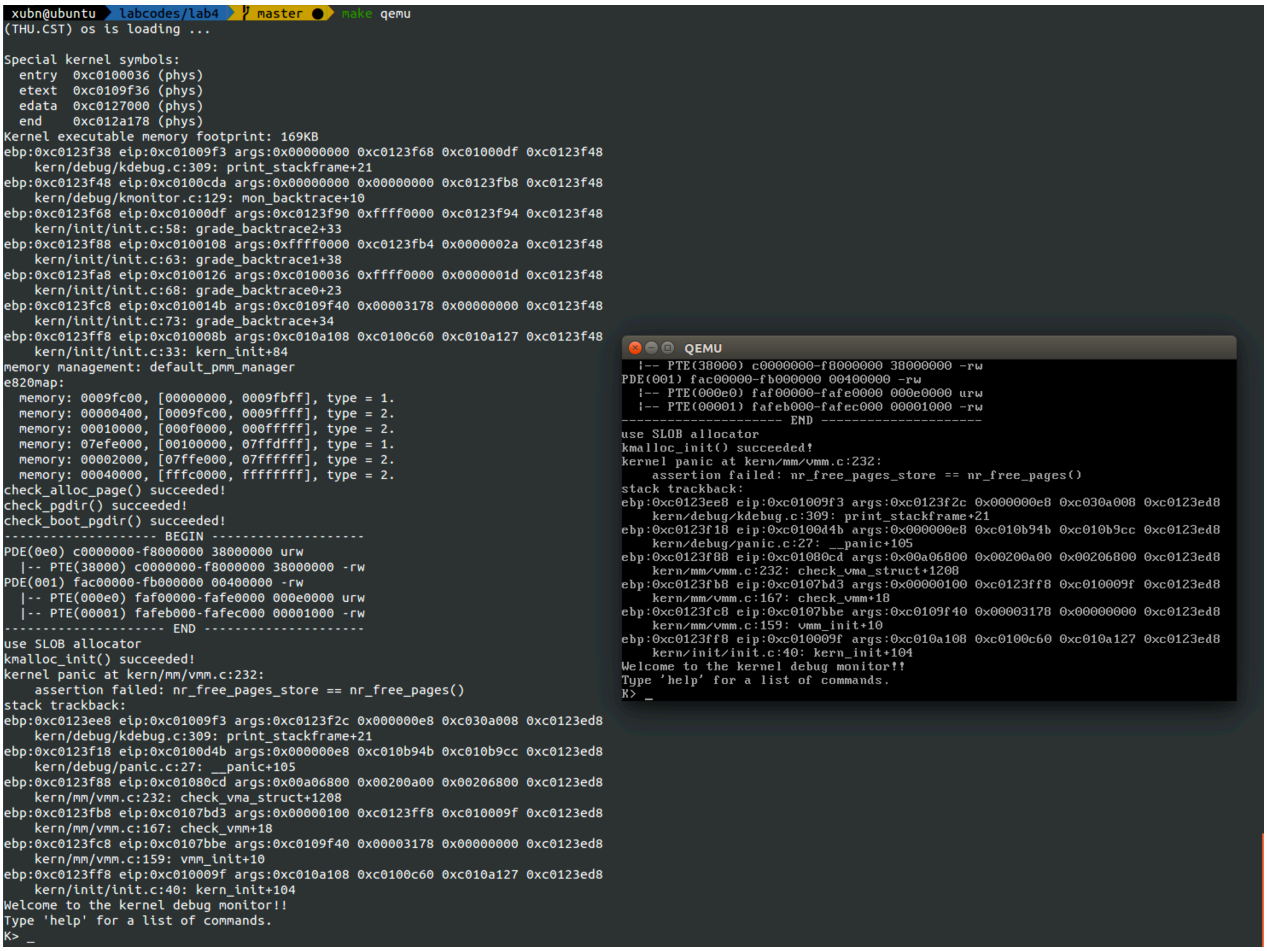
由以上大概流程可知，本实验在执行过程中一共只创建了两个内核线程，分别为 **idleproc** 和 **initproc** 。

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

这个语句段起到关中断 - 处理中断 - 重启中断功能的作用。如前面探究进程/线程所分配到的 id 是否一致的问题中所说的，关中断的操作，在单核的 ucore 系统中起到的是类似于互斥的作用，当一个进程陷入中断处理，关闭中断以保证其他中断不会被执行，其中的临界区、量不会被重复访问修改，保证了相关处理和操作的原子性，避免发生冲突和难以预测和控制的后果。

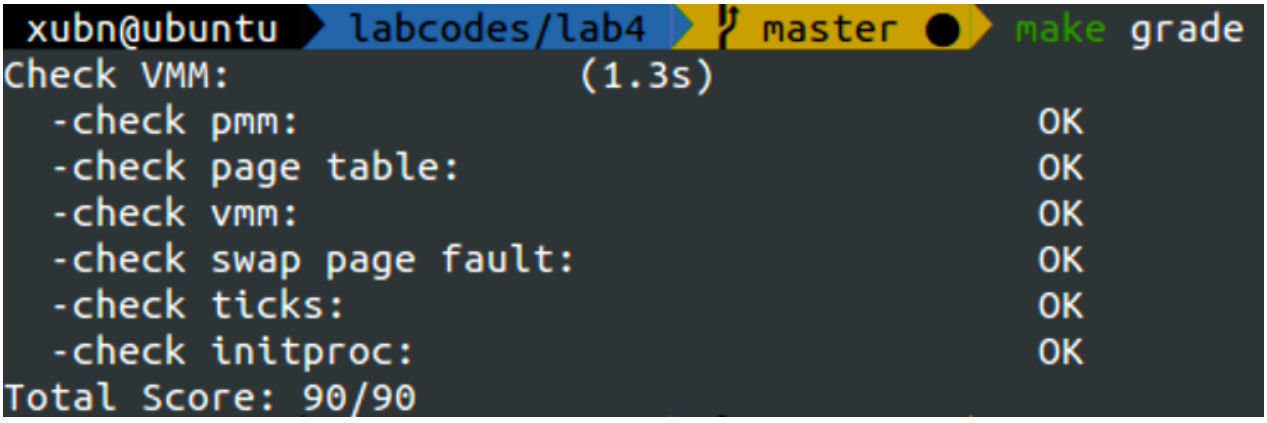
实验结果

完成实验中各部分代码的编写后，命令行 cd 进入实验目录 lab4 ，随后执行 **`make qemu`** 命令，查看结果：



可以看到显示结果基本如实验要求中，内容基本一致。

执行 **`make grade`** 命令，查看利用 shell 脚本进行检查的检查结果：



基本满足实验要求。

实验总结

分析与区别

- 练习1参考了注释操作，与答案一致；
- 练习2基本与答案一致，原先未注意到函数末尾 goto 标签的错误处理，对照答案之后已经进行修改；

并且其中的将进程插入链表操作原先实现没有完成关中断的互斥操作，参照答案之后添加了 intr_flag 使用。

重要知识点

- 进程创建中对 PCB 的初始化操作；
- 进程切换过程中现场保存、上下文切换的过程。

补充知识点

- 进程切换具体流程，可能因为过程太过复杂，战线不宜过长而未在实验中实践，但可以从代码中了解；
- 进程上下文保存的寄存器具体信息的含义，以及通过对 esp eip 寄存器实现进程切换的细节是查找了相关资料和分析之后才理解的。

体会与反思

ucore 的第四次实验，是关于内核线程管理的实验。本次实验依然是之前一两次实验的细节化的实现，通过让我们自己编写比较关键的中间函数的过程，让我们对内核线程管理的执行过程，相关参量和函数的操作有一个大体的直观感受。虽然练习中要求我们所做的工作只是一些基本的函数理解和调用，但是这两处地方代码的理解是比较关键的，这让我们对基本所有关键的功能函数都有大体的浏览和理解，需要自己编写的代码中更不乏有关中断这样的比较"意外"的处理，没有做到后面的思考题或者看答案的话很可能会忽略这一点，同时这也让我联想到了前一段时间理论课中正好接触到了的同步互斥问题和信号量的运用，的确在单核的 ucore 中，这样的操作也就是互斥了，将内核操作和处理当作临界操作，对中断标识的控制当作信号量的控制，就很好理解。

即使这样的实验安排已经算是比较深入底层的实现，但还是会遇到很多比如看内嵌汇编代码的时候一时间理解不了，调用最基本的核心的函数的时候非常生疏无从下手等的问题，所以这样的练习和实现应该只能算是"中下层"的吧，已经离抽象层次的概念比较远，但同时又还不够底层，这一层的理解还是需要多花点功夫的。希望在后续的实验中自己的理解可以慢慢加深，对 ucore 的内核和原理能有更好的理解和把握。