LAB4：内核线程管理

- 白欣雨 PB18051183

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。

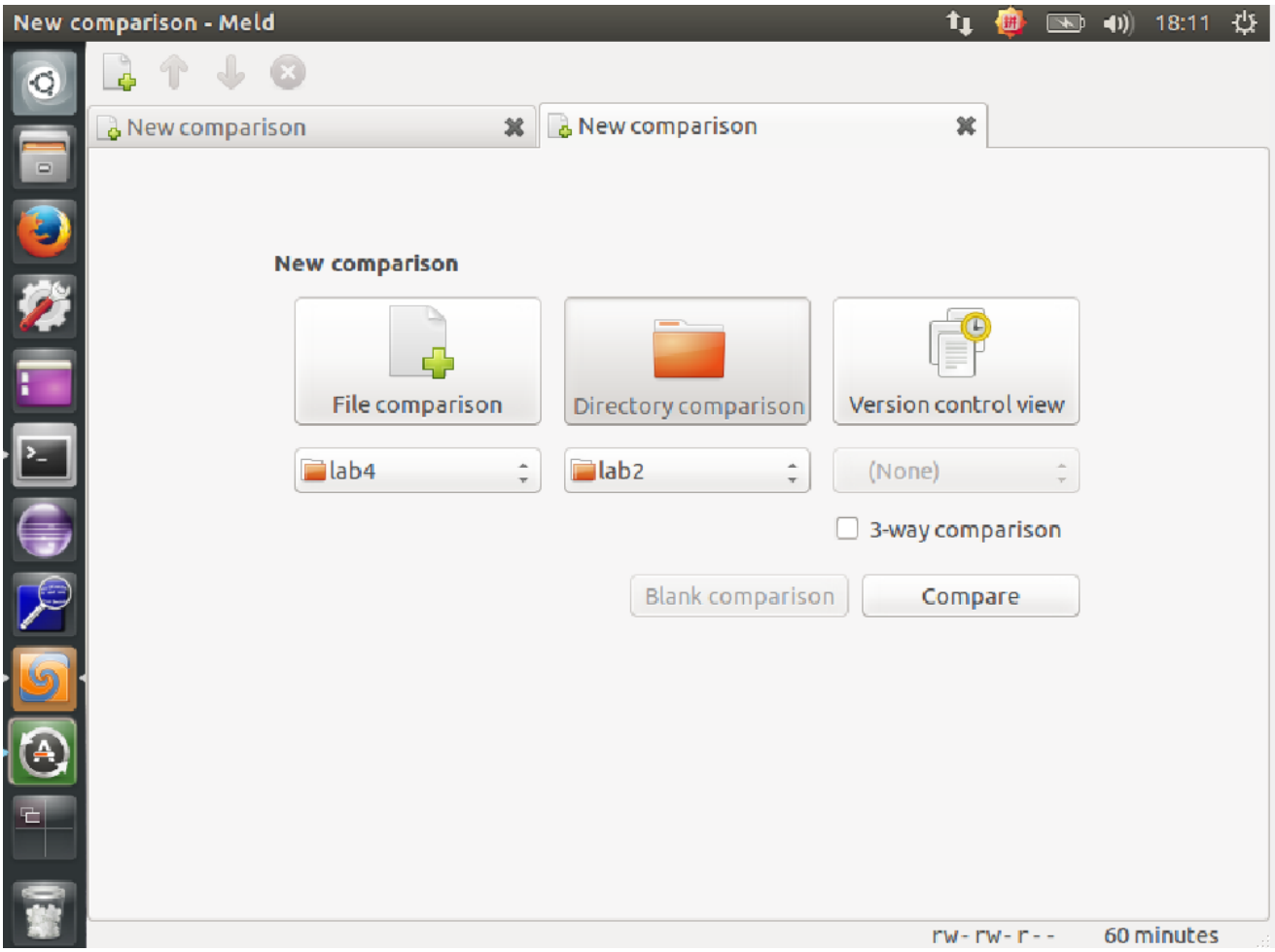
实验内容

前述实验完成了对物理和虚拟内存的管理，这些为本次实验打下了基础。在分配好内存后，就要考虑如何分时利用CPU来“并发”执行多个程序，这为本次实验将进行内核线程的管理。

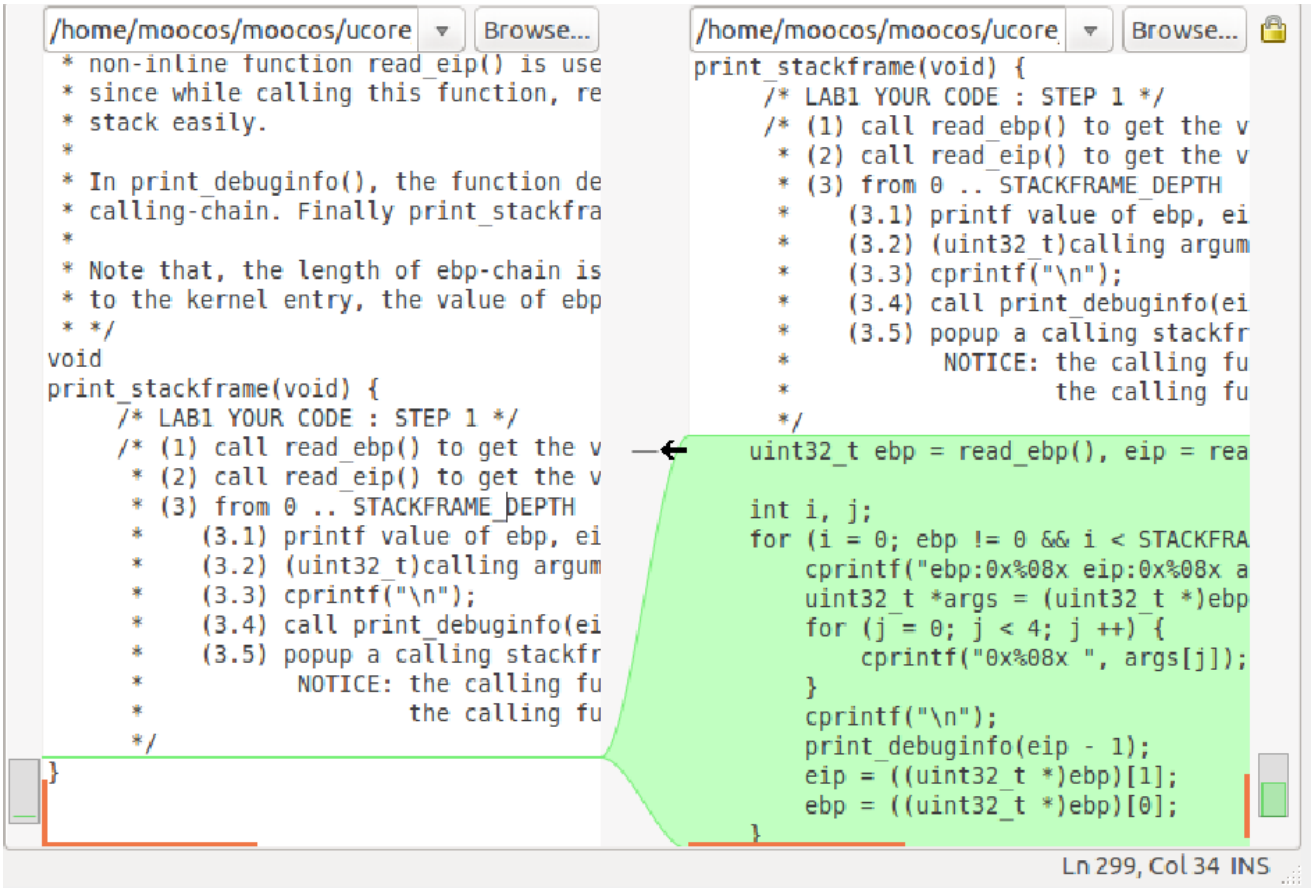
实验过程

练习0：填写已有实验

此练习需要填写Lab1、Lab2、Lab3的代码，使用meld图形化方式进行合并，选择两个目录进行比较：



根据比较结果，我们找到需要填写的部分，要修改的文件有：trap.c 、kdebug.c 、default_pmm.c 、pmm.c 、swap_fifo.c 、vmm.c，具体比较填写可以根据meld的对比添加与替换，下以一个举例说明：



按照此步骤填写完成即可。

练习1：分配并初始化一个进程控制块

实验准备

在正式实验之前，首先根据实验指导书了解一下相关内容。练习1是让我们完成初始化分配一个内核线程的PCB。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，而用户进程会在在用户态和内核态交替运行。
- 所有内核线程直接使用共同ucore内核内存空间，不需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。

操作系统是以进程为中心设计的，所以其首要任务是为进程建立档案，进程档案用于表示、标识或描述进程，即进程控制块。而这里我们分配的是一个内核线程的PCB，它通常只是内核中的一小段代码或者函数，没有用户空间，通过设置页表建立核心虚拟空间。内核中的所有线程都不需要再建立各自的页表，只需共享这个核心虚拟空间就可以访问整个物理内存了。

接下来我们查看proc.h头文件中关于PCB的定义（结构体），结构体中各参数的含义参见注释：

```
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
};
```

对于比较重要的、需要我们初始化的参数进行一个简单的说明：

- state：进程所处的状态，主要有以下类型：

  PROC_UNINIT = 0 // uninitialized
  PROC_SLEEPING // sleeping
  PROC_RUNNABLE // runnable(maybe running)
  PROC_ZOMBIE // almost dead, and wait parent proc to reclaim his resource
- pid：进程ID。
- kstack：分配给该进程/线程的内核栈位置。
- need_resched：是否需要调度。
- parent：进程的父进程。
- mm：描述进程虚拟内存的结构体。
- context：上下文，用于切换。
- tf：中断帧的指针，指向内核栈，中断帧记录了进程在被中断前的状态。
- cr3：当前使用的页表地址。

函数实现

根据实验指导书，我们需要实验一个 alloc_proc 函数，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管理信息。这个函数位于 kern/process/proc.c 中。 根据注释的提示：

```
/*
     * 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
     */
```

在alloc_proc函数的实现中，需要初始化的成员变量已经在注释中说明，直接根据实验准备中的分析，按照含义依次进行初始化即可，具体实现过程见注释：

```
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
```

```
                                        //新PCB空间已申请
    if (proc != NULL) {                 //分配成功
        proc->state = PROC_UNINIT;  //设置进程为未初始化状态
        proc->pid = -1;             //设置未初始化的的进程id为-1
        proc->runs = 0;             //初始化运行时间
        proc->kstack = 0;           //初始化内核栈地址
        proc->need_resched = 0;     //将是否需要调度设为不需要
        proc->parent = NULL;        //父节点设为空
        proc->mm = NULL;           //虚拟内存设为空
        memset(&(proc->context), 0, sizeof(struct context));
                                //上下文的初始化
        proc->tf = NULL;          //中断帧指针置为空
        proc->cr3 = boot_cr3;     //页目录设为内核页目录表的基址
        proc->flags = 0;         //标志位置零
        memset(proc->name, 0, PROC_NAME_LEN);    //进程名初始化
    }
    return proc;
}
```

此部分比较简单，与参考答案代码实现过程相似，均为直接赋值即可。

- 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?

  struct context context

  查看其在头文件中的定义：

  ```
  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;
  };
  ```

  可以看到在结构体中存储很多通用寄存器以及eip 的数值，可以推测context 很有可能是保存的线程运行的上下文信息。

  通过查找，context成员变量在Swtich.s 和proc.c 中的 copy_thread函数中被引用。因此根据Swtich 中代码的语义，可以确定context变量的意义就在于内核线程之间进行切换的时候，将原先的线程运行的上下文保存下来这一作用。

  总结如下：

  含义：存储进程上下文

  作用：用于进程切换

- **struct trapframe *tf**

  通过查找声明的头文件，找到trapframe的定义：

  ```
  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));
```

而在copy_thread函数中，对tf变量进行了设置：

```
static void
copy_thread(struct proc_struct *proc, uintptr_t esp, struct trapframe *tf) {
  proc->tf = (struct trapframe *)(proc->kstack + KSTACKSIZE) - 1;
    *(proc->tf) = *tf;
    proc->tf->tf_regs.reg_eax = 0;
    proc->tf->tf_esp = esp;
    proc->tf->tf_eflags |= FL_IF;

    proc->context.eip = (uintptr_t)forkret;
    proc->context.esp = (uintptr_t)(proc->tf);
}
```

可以发现tf变量被设置为esp。结合前述内容可以推测，tf变量的作用在于在构造出了新的线程的时候，使用中断返回的方式将控制权交给这个线程，因此需要构造出一个中断返回现场，也就是trapframe，使得可以正确地将控制权转交给新的线程。

可以总结如下：

含义：中断帧指针。

作用：总是指向内核栈的某个位置。中断帧记录了进程在被中断前的状态。

至此，练习1的全部实验内容结束。

练习2：为新创建的内核线程分配资源

实验准备

创建一个内核线程需要分配和设置很多资源，kernel_thread函数通过调用do_fork函数完成具体内核线程创建工作。alloc_proc实质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。do_fork才是真正完成了资源分配的工作，但它也只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。根据实验指导书，需要完成在 kern/process/proc.c中的 do_fork 函数中的处理过程。

根据注释提示：

```
/*
    * 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
```

注释给我们提供了一些有用的宏、函数和定义，可以在函数实现中使用，还给我们提供了它的执行步骤，大致如下：

1. 调用alloc_proc，获得一块初始化的进程控制块。
2. 调用 setup_kstack，为进程分配并初始化一个内核栈。
3. 调用copy_mm，根据clone_flag，复制原进程的内存管理信息到新进程。
4. 调用copy_thread，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. 通过wakup_proc，将新进程设置为“就绪态”。
7. 设置返回新进程的PID。

函数实现

根据实验准备的分析，可以得到补全后的代码，实现过程见注释：

```
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    int ret = -E_NO_FREE_PROC;  //尝试分配内存
    struct proc_struct *proc;   //新内存
    if (nr_process >= MAX_PROCESS) {
        goto fork_out;          //返回
    }
    ret = -E_NO_MEM;            //内存不足，分配失败

    //1: 调用 alloc_proc 函数申请内存块，如果失败，直接返回处理
    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;              //返回
    }

    //2.将子进程的父节点设置为当前进程
    proc->parent = current;

    //3.调用 setup_stack 函数为进程分配一个内核栈
    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_proc;
```

```
    }

    //4.调用 copy_mm 函数将父进程的内存信息复制到子进程
    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }

    //5.调用 copy_thread 函数复制父进程的中断帧和上下文信息
    copy_thread(proc, stack, tf);

    //6.将新进程添加到进程的hash列表中
    bool intr_flag = 0;          //中断标记，初值为false
    local_intr_save(intr_flag);{ //屏蔽中断，保护
    proc->pid = get_pid();       //获取PID
    hash_proc(proc);             //建立hash映射
    list_add(&proc_list, &(proc->list_link)); //将新进程加入进程列表
    nr_process ++;}              //进程数增加1个
    local_intr_restore(intr_flag);

    //7.唤醒子进程
    wakeup_proc(proc);

    //8.返回子进程的PID
    ret = proc->pid;

fork_out:       //分配进程数超过最大值
    return ret;

bad_fork_cleanup_kstack:    //处理失败
    put_kstack(proc);
bad_fork_cleanup_proc:
    kfree(proc);
    goto fork_out;
}
```

此部分需要根据注释的提示，并且自己查找相关宏定义的含义，才能较好的理解代码的含义，当然在了解代码的含义以及作用之后，实现起来会相对容易。将自己答案与参考答案对比时发现，答案考虑了许多分配失败的处理，自己的代码考虑得还是不够周全，包括本函数中使用的屏蔽中断，代码的鲁棒性需要进一步加强；当然也发现答案对很多参数进行了默认赋值，我认为虽然后续的函数会处理，但是还是在声明变量时直接赋初值更保险（尤其是在写代码的过程与调试过程）。

- 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。

  在分配ID时调用了get_pid()函数，找到此函数的定义：

```
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;
}
```

每次都从进程控制块链表中找到合适的ID。程序首先判断是否还有多余的PID可以使用。线程的PID由get_pid函数产生，该函数中包含了两个变量last_pid以及next_safe ， last_pid变量保存上次分配的PID，这样就可以next_safe和last_pid一起表示一段可以使用的PID取值范围，这两个变量的数值之间的取值均是可以使用的 pid，为保证这么做的正确性，需要有严格的next_safe > last_pid + 1，那么久可以直接取last_pid + 1作为新的PID。此函数还处理了如果两个变量之间没有合法取值的情况。因此，在通常情况下，都可以分配给每个新fork的线程一个唯一的ID。

至此，练习2的全部内容结束。

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

首先我们找到proc_run函数，初步分析见注释：

```
void
proc_run(struct proc_struct *proc) {
    if (proc != current) {          //判断是否在运行
        bool intr_flag;             //中断标记
        struct proc_struct *prev = current, *next = proc;
        local_intr_save(intr_flag);  //屏蔽中断
        {
            current = proc;         //切换到当前进程
            load_esp0(next->kstack + KSTACKSIZE);
            lcr3(next->cr3);//修改当前的cr3寄存器成需要运行线程的页目录表
            //调用switch_to函数进行上下文的保存并切换到当前进程
            switch_to(&(prev->context), &(next->context));

        }
        local_intr_restore(intr_flag);
    }
}
```

其中，switch_to函数内容与作用比较不清晰，我们做一个简要分析：

```
switch_to: # switch_to(from, to)
    # 保存前一进程的执行现场
    movl 4(%esp), %eax
    popl 0(%eax)
    # 保存寄存器中的内容到 context 对应区域中
    movl %esp, 4(%eax)
    movl %ebx, 8(%eax)
    movl %ecx, 12(%eax)
    movl %edx, 16(%eax)
    movl %esi, 20(%eax)
    movl %edi, 24(%eax)
    movl %ebp, 28(%eax)

    # 恢复一个进程的执行现场
    movl 4(%esp), %eax
    movl 28(%eax), %ebp
    movl 24(%eax), %edi
    movl 20(%eax), %esi
    movl 16(%eax), %edx
    movl 12(%eax), %ecx
    movl 8(%eax), %ebx
    movl 4(%eax), %esp
    pushl 0(%eax)               # push eip
    ret
```

可以发现， switch_to函数就是完成进程的上下文切换，先保存当前寄存器的值然后再将下一个进程的上下文信息保存到寄存器中。其中，最后一条指令：pushl 0(%eax)把 context 中保存的下一个进程要执行的指令地址放到了堆栈顶，接下来执行最后一条指令ret时,会把栈顶的内容赋值给 EIP 寄存器，这样就切换到下一个进程执行，从而完成了进程的切换。

综上，可以总结proc_run函数的执行过程大致为：

1. 如果当前进程没有运行，则开始执行proc_run的主体部分。
2. 屏蔽中断后，将当前进程设置为proc。
3. 设置任务状态段 ts 中特权态 0 下的栈顶指针 esp0 为 next 内核线程 proc 的内核栈的栈顶。
4. 设置 cr3 寄存器的值为 next 内核线程proc 的页目录表起始地址 next->cr3，进行进程间的页表切换；
5. 由 switch_to函数完成两个线程的执行现场切换，切换到我们需要的进程执行。

- 在本实验的执行过程中，创建且运行了几个内核线程？

  总共创建了两个线程idle_proc以及init_proc。

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

  考虑这样一种情况：proc_run函数将current指向了要切换到的线程，但是此时还没有真正将控制权转移过去，如果在这个时候出现中断，就会出现current中保存的并不是正在运行的线程的中断控制块导致不匹配。因此，在进行进程切换的时候，需要避免出现中断干扰这个过程，以免造成可能出现的错误。该语句的左右是关闭中断，使得在这个语句块内的内容不会被中断打断。

实验结果与验证

使用make grade命令查看完成情况：

```
[~/moocos/ucore_lab/labcodes/lab4]
moocos-> make grade
Check VMM:                (4.5s)
  -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
```

然后执行make qemu，结果如下：

```
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:355:
    process exit!!.

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

完整代码：

```
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc010002a (phys)
  etext  0xc010b220 (phys)
  edata  0xc0128a90 (phys)
  end    0xc012bc18 (phys)
Kernel executable memory footprint: 175KB
ebp:0xc0127f38 eip:0xc0101e6a args:0x00010094 0x00000000 0xc0127f68 0xc01000d3
    kern/debug/kdebug.c:308: print_stackframe+21
ebp:0xc0127f48 eip:0xc0102159 args:0x00000000 0x00000000 0x00000000 0xc0127fb8
    kern/debug/kmonitor.c:129: mon_backtrace+10
ebp:0xc0127f68 eip:0xc01000d3 args:0x00000000 0xc0127f90 0xffff0000 0xc0127f94
    kern/init/init.c:57: grade_backtrace2+33
ebp:0xc0127f88 eip:0xc01000fc args:0x00000000 0xffff0000 0xc0127fb4 0x0000002a
    kern/init/init.c:62: grade_backtrace1+38
ebp:0xc0127fa8 eip:0xc010011a args:0x00000000 0xc010002a 0xffff0000 0x0000001d
```

```
    kern/init/init.c:67: grade_backtrace0+23
ebp:0xc0127fc8 eip:0xc010013f args:0xc010b23c 0xc010b220 0x00003188 0x00000000
    kern/init/init.c:72: grade_backtrace+34
ebp:0xc0127ff8 eip:0xc010007f args:0x00000000 0x00000000 0x0000ffff 0x40cf9a00
    kern/init/init.c:32: kern_init+84
memory management: default_pmm_manager
e820map:
  memory: 0009fc00, [00000000, 0009fbff], type = 1.
  memory: 00000400, [0009fc00, 0009ffff], type = 2.
  memory: 00010000, [000f0000, 000fffff], type = 2.
  memory: 07efe000, [00100000, 07ffdfff], type = 1.
  memory: 00002000, [07ffe000, 07ffffff], type = 2.
  memory: 00040000, [fffc0000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
-------------------- BEGIN --------------------
PDE(0e0) c0000000-f8000000 38000000 urw
  |-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
  |-- PTE(000e0) faf00000-fafe0000 000e0000 urw
  |-- PTE(00001) fafeb000-fafec000 00001000 -rw
--------------------- END ---------------------
use SLOB allocator
check_slab() success
kmalloc_init() succeeded!
check_vma_struct() succeeded!
page fault at 0x00000100: K/W [no page found].
check_pgfault() succeeded!
check_vmm() succeeded.
ide 0:      10000(sectors), 'QEMU HARDDISK'.
ide 1:     262144(sectors), 'QEMU HARDDISK'.
SWAP: manager = fifo swap manager
BEGIN check_swap: count 31950, total 31950
setup Page Table for vaddr 0X1000, so alloc a page
setup Page Table vaddr 0~4MB OVER!
set up init env for check_swap begin!
page fault at 0x00001000: K/W [no page found].
page fault at 0x00002000: K/W [no page found].
page fault at 0x00003000: K/W [no page found].
page fault at 0x00004000: K/W [no page found].
set up init env for check_swap over!
write Virt Page c in fifo_check_swap
write Virt Page a in fifo_check_swap
write Virt Page d in fifo_check_swap
write Virt Page b in fifo_check_swap
write Virt Page e in fifo_check_swap
page fault at 0x00005000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
write Virt Page b in fifo_check_swap
write Virt Page a in fifo_check_swap
page fault at 0x00001000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x2000 to disk swap entry 3
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:354:
    process exit!!.

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

与参考结果一致。

Challenge：实现支配任意大小的内存分配算法

根据实验要求，我们先来了解一下Slub与其他分配算法的区别与联系。

首先内核里小内存分配一共有三种，Slab/Slub/Slob，Slub分配器是Slab分配器的进化版，而Slob是一种精简的小内存分配算法，主要用于嵌入式系统。发明Slub分配器的主要目的就是减少Slab缓冲区的个数，让更多的空闲内存得到使用。在LAB2中实现了伙伴系统算法，与Slub不同的是：伙伴算法以页为单位管理内存。但在大多数情况下，程序需要的并不是一整页，而是几个、几十个字节的小内存。于是需要Slub系统。Slub系统运行在伙伴系统之上，为内核提供小内存管理的功能。

因此我们可以将实验2中的算法进行改进，以实现Slub算法，在实现之前，先了解一下Slub算法需要定义的一些特殊函数（接口）：

Slub接口:

```
struct kmem_cache *kmem_cache_create(const char *name,
size_t size,
size_t align,
unsigned long flags,
void (*ctor)(void *));
void kmem_cache_destroy(struct kmem_cache *);
void *kmem_cache_alloc(struct kmem_cache *cachep, int flags);
void kmem_cache_free(struct kmem_cache *cachep, void *objp);
```

- kmem_cache_create是创建kmem_cache数据结构。kmem_cache用于管理大小不等的object，包括大小、页数等等。通常其定义如下：

```
struct kmem_cache {
struct kmem_cache_cpu __percpu *cpu_slab;
/* Used for retriving partial slabs etc */
slab_flags_t flags;
unsigned long min_partial;
int size; /* The size of an object including meta data */
int object_size; /* The size of an object without meta data */
```

```
int offset; /* Free pointer offset. */
#ifdef CONFIG_SLUB_CPU_PARTIAL
int cpu_partial; /* Number of per cpu partial objects to keep around */
#endif
struct kmem_cache_order_objects oo;
/* Allocation and freeing of slabs */
struct kmem_cache_order_objects max;
struct kmem_cache_order_objects min;
gfp_t allocflags; /* gfp flags to use on each alloc */
int refcount; /* Refcount for slab cache destroy */
void (*ctor)(void *);
int inuse; /* Offset to metadata */
int align; /* Alignment */
int reserved; /* Reserved bytes at the end of slabs */
const char *name; /* Name (only for display!) */
struct list_head list; /* List of slab caches */
struct kmem_cache_node *node[MAX_NUMNODES];
};
```

- kmem_cache_destroy是销毁创建的kmem_cache。
- kmem_cache_alloc是从cachep参数指定的kmem_cache管理的内存缓存池中分配一个对象。
- kmem_cache_free是释放对象。

由于我在LAB2中实现的是伙伴算法，因此查阅相关资料，得知还需要定义一些基本函数：

- kmem_int：初始化kmem_cache仓库。定义为：

```
void kmem_int() {
    int i;
    //Init cache for kmem_cache
    cache_cache.objsize = sizeof ( struct kmem_cache_t) ;
    cache_cache.num = PGSIZE / (sizeof (int16_t ) + sizeof ( structkmem_cache_t ) );
    cache_cache.ctor = NULL;cache_cache.dtor = NULL;
    memcpy(cache_cache.name, cache_cache_name,CACHE_NAMELEN);
    list_init(&(cache_cache.slabs_full));
    list_init(&(cache_cache.slabs _partial));
    list_init(&(cache_cache.slabs_free));
    list_init(&(cache_chain));
    list_add (&(cache_chain),&(cache_cache.cache_link));
    //Init sized cache
    for ( i = 0,size = 16; i < SIZED_CACHE_NUM; i++, size *= 2)
        sized_caches[i] = kmem_cache_create(sized_cache_name,size, NULL,NULL );
    check _kmem( );
}
```

然后就可以进行算法的改进：

- 初始化Slub：

```
void
pmm_init(void) {
    //We need to alloc/free the physical memory (granularity is 4KB or other size).
    //So a framework of physical memory manager (struct pmm_manager)is defined in pmm.h
    //First we should init a physical memory manager(pmm) based on the framework.
    //Then pmm can alloc/free the physical memory.
    //Now the first_fit/best_fit/worst_fit/buddy_system pmm are available.
    init_pmm_manager();
```

```
    // detect physical memory space, reserve already used memory,
    // then use pmm->init_memmap to create free page list
    page_init();

    //use pmm->check to verify the correctness of the alloc/free function in a pmm
    check_alloc_page();

    // create boot_pgdir, an initial page directory(Page Directory Table, PDT)
    boot_pgdir = boot_alloc_page();
    memset(boot_pgdir, 0, PGSIZE);
    boot_cr3 = PADDR(boot_pgdir);

    check_pgdir();

    static_assert(KERNBASE % PTSIZE == 0 && KERNTOP % PTSIZE == 0);

    // recursively insert boot_pgdir in itself
    // to form a virtual page table at virtual address VPT
    boot_pgdir[PDX(VPT)] = PADDR(boot_pgdir) | PTE_P | PTE_W;

    // map all physical memory to linear memory with base linear addr KERNBASE
    //linear_addr KERNBASE~KERNBASE+KMEMSIZE = phy_addr 0~KMEMSIZE
    //But shouldn't use this map until enable_paging() & gdt_init() finished.
    boot_map_segment(boot_pgdir, KERNBASE, KMEMSIZE, 0, PTE_W);

    //temporary map:
    //virtual_addr 3G~3G+4M = linear_addr 0~4M = linear_addr 3G~3G+4M = phy_addr 0~4M
    boot_pgdir[0] = boot_pgdir[PDX(KERNBASE)];

    enable_paging();

    //reload gdt(third time,the last time) to map all physical memory
    //virtual_addr 0~4G=liear_addr 0~4G
    //then set kernel stack(ss:esp) in TSS, setup TSS in gdt, load TSS
    gdt_init();

    //disable the map of virtual_addr 0~4M
    boot_pgdir[0] = 0;

    //now the basic virtual memory map(see memalyout.h) is established.
    //check the correctness of the basic virtual memory map.
    check_boot_pgdir();

    print_pgdir();
    kmem_int();
}
```

- 初始化仓库：

```
//在vmm.c中修改
#include <vmm.h>
#include <sync.h>
#include <string.h>
#include <assert.h>
#include <stdio.h>
#include <error.h>
#include <pmm.h>
#include <x86.h>
#include <swap.h>
#include <kmalloc.h>
```

```
struct kmem_cache_t *vma_cache = NULL;
struct kmem_cache_t *mm_cache = NULL;
static void check_vmm(void);
static void check_vma_struct(void);
static void check_pgfault(void);

// mm_create -  alloc a mm_struct & initialize it.
struct mm_struct *
mm_create(void) {
    struct mm_struct *mm = kmem_cache_alloc(mm_cache);

    if (mm != NULL) {
        list_init(&(mm->mmap_list));
        mm->mmap_cache = NULL;
        mm->pgdir = NULL;
        mm->map_count = 0;

        if (swap_init_ok) swap_init_mm(mm);
        else mm->sm_priv = NULL;
    }
    return mm;
}

// vma_create - alloc a vma_struct & initialize it. (addr range: vm_start~vm_end)
struct vma_struct *
vma_create(uintptr_t vm_start, uintptr_t vm_end, uint32_t vm_flags) {
    struct vma_struct *vma = kmem_cache_alloc(vma_cache);

    if (vma != NULL) {
        vma->vm_start = vm_start;
        vma->vm_end = vm_end;
        vma->vm_flags = vm_flags;
    }
    return vma;
}


// find_vma - find a vma  (vma->vm_start <= addr <= vma_vm_end)
struct vma_struct *
find_vma(struct mm_struct *mm, uintptr_t addr) {
    struct vma_struct *vma = NULL;
    if (mm != NULL) {
        vma = mm->mmap_cache;
        if (!(vma != NULL && vma->vm_start <= addr && vma->vm_end > addr)) {
                bool found = 0;
                list_entry_t *list = &(mm->mmap_list), *le = list;
                while ((le = list_next(le)) != list) {
                    vma = le2vma(le, list_link);
                    if (vma->vm_start<=addr && addr < vma->vm_end) {
                        found = 1;
                        break;
                    }
                }
                if (!found) {
                    vma = NULL;
                }
        }
        if (vma != NULL) {
            mm->mmap_cache = vma;
        }
```

```
    }
    return vma;
}


// check_vma_overlap - check if vma1 overlaps vma2 ?
static inline void
check_vma_overlap(struct vma_struct *prev, struct vma_struct *next) {
    assert(prev->vm_start < prev->vm_end);
    assert(prev->vm_end <= next->vm_start);
    assert(next->vm_start < next->vm_end);
}


// insert_vma_struct -insert vma in mm's list link
void
insert_vma_struct(struct mm_struct *mm, struct vma_struct *vma) {
    assert(vma->vm_start < vma->vm_end);
    list_entry_t *list = &(mm->mmap_list);
    list_entry_t *le_prev = list, *le_next;

        list_entry_t *le = list;
        while ((le = list_next(le)) != list) {
            struct vma_struct *mmap_prev = le2vma(le, list_link);
            if (mmap_prev->vm_start > vma->vm_start) {
                break;
            }
            le_prev = le;
        }

    le_next = list_next(le_prev);

    /* check overlap */
    if (le_prev != list) {
        check_vma_overlap(le2vma(le_prev, list_link), vma);
    }
    if (le_next != list) {
        check_vma_overlap(vma, le2vma(le_next, list_link));
    }

    vma->vm_mm = mm;
    list_add_after(le_prev, &(vma->list_link));

    mm->map_count ++;
}

// mm_destroy - free mm and mm internal fields
void
mm_destroy(struct mm_struct *mm) {

    list_entry_t *list = &(mm->mmap_list), *le;
    while ((le = list_next(list)) != list) {
        list_del(le);
        kmem_cache_free(mm_cache, le2vma(le, list_link));  //kfree vma
    }
    kmem_cache_free(mm_cache, mm); //kfree mm
    mm=NULL;
}

// vmm_init - initialize virtual memory management
//          - now just call check_vmm to check correctness of vmm
```

```
void
vmm_init(void) {
    mm_cache = kmem_cache_create("mm", sizeof(struct mm_struct), NULL, NULL);
    vma_cache = kmem_cache_create("vma", sizeof(struct vma_struct), NULL, NULL);
    check_vmm();
}

// check_vmm - check correctness of vmm
static void
check_vmm(void) {
    size_t nr_free_pages_store = nr_free_pages();

    check_vma_struct();
    check_pgfault();

 //   assert(nr_free_pages_store == nr_free_pages());

    cprintf("check_vmm() succeeded.\n");
}

static void
check_vma_struct(void) {
    size_t nr_free_pages_store = nr_free_pages();

    struct mm_struct *mm = mm_create();
    assert(mm != NULL);

    int step1 = 10, step2 = step1 * 10;

    int i;
    for (i = step1; i >= 1; i --) {
        struct vma_struct *vma = vma_create(i * 5, i * 5 + 2, 0);
        assert(vma != NULL);
        insert_vma_struct(mm, vma);
    }

    for (i = step1 + 1; i <= step2; i ++) {
        struct vma_struct *vma = vma_create(i * 5, i * 5 + 2, 0);
        assert(vma != NULL);
        insert_vma_struct(mm, vma);
    }

    list_entry_t *le = list_next(&(mm->mmap_list));

    for (i = 1; i <= step2; i ++) {
        assert(le != &(mm->mmap_list));
        struct vma_struct *mmap = le2vma(le, list_link);
        assert(mmap->vm_start == i * 5 && mmap->vm_end == i * 5 + 2);
        le = list_next(le);
    }

    for (i = 5; i <= 5 * step2; i +=5) {
        struct vma_struct *vma1 = find_vma(mm, i);
        assert(vma1 != NULL);
        struct vma_struct *vma2 = find_vma(mm, i+1);
        assert(vma2 != NULL);
        struct vma_struct *vma3 = find_vma(mm, i+2);
        assert(vma3 == NULL);
        struct vma_struct *vma4 = find_vma(mm, i+3);
        assert(vma4 == NULL);
        struct vma_struct *vma5 = find_vma(mm, i+4);
```

```
        assert(vma5 == NULL);

        assert(vma1->vm_start == i  && vma1->vm_end == i  + 2);
        assert(vma2->vm_start == i  && vma2->vm_end == i  + 2);
    }

    for (i =4; i>=0; i--) {
        struct vma_struct *vma_below_5= find_vma(mm,i);
        if (vma_below_5 != NULL ) {
           cprintf("vma_below_5: i %x, start %x, end %x\n",i, vma_below_5->vm_start,
vma_below_5->vm_end);
        }
        assert(vma_below_5 == NULL);
    }

    mm_destroy(mm);

//    assert(nr_free_pages_store == nr_free_pages());

    cprintf("check_vma_struct() succeeded!\n");
}

struct mm_struct *check_mm_struct;

// check_pgfault - check correctness of pgfault handler
static void
check_pgfault(void) {
    size_t nr_free_pages_store = nr_free_pages();

    check_mm_struct = mm_create();
    assert(check_mm_struct != NULL);

    struct mm_struct *mm = check_mm_struct;
    pde_t *pgdir = mm->pgdir = boot_pgdir;
    assert(pgdir[0] == 0);

    struct vma_struct *vma = vma_create(0, PTSIZE, VM_WRITE);
    assert(vma != NULL);

    insert_vma_struct(mm, vma);

    uintptr_t addr = 0x100;
    assert(find_vma(mm, addr) == vma);

    int i, sum = 0;
    for (i = 0; i < 100; i ++) {
        *(char *)(addr + i) = i;
        sum += i;
    }
    for (i = 0; i < 100; i ++) {
        sum -= *(char *)(addr + i);
    }
    assert(sum == 0);

    page_remove(pgdir, ROUNDDOWN(addr, PGSIZE));
    free_page(pa2page(pgdir[0]));
    pgdir[0] = 0;

    mm->pgdir = NULL;
    mm_destroy(mm);
    check_mm_struct = NULL;
```

```
    assert(nr_free_pages_store == nr_free_pages());

    cprintf("check_pgfault() succeeded!\n");
}
//page fault number
volatile unsigned int pgfault_num=0;

int
do_pgfault(struct mm_struct *mm, uint32_t error_code, uintptr_t addr) {
    int ret = -E_INVAL;
    //try to find a vma which include addr
    struct vma_struct *vma = find_vma(mm, addr);

    pgfault_num++;
    //If the addr is in the range of a mm's vma?
    if (vma == NULL || vma->vm_start > addr) {
        cprintf("not valid addr %x, and  can not find it in vma\n", addr);
        goto failed;
    }
    //check the error_code
    switch (error_code & 3) {
    default:
            /* error code flag : default is 3 ( W/R=1, P=1): write, present */
    case 2: /* error code flag : (W/R=1, P=0): write, not present */
        if (!(vma->vm_flags & VM_WRITE)) {
            cprintf("do_pgfault failed: error code flag = write AND not present, but the
addr's vma cannot write\n");
            goto failed;
        }
        break;
    case 1: /* error code flag : (W/R=0, P=1): read, present */
        cprintf("do_pgfault failed: error code flag = read AND present\n");
        goto failed;
    case 0: /* error code flag : (W/R=0, P=0): read, not present */
        if (!(vma->vm_flags & (VM_READ | VM_EXEC))) {
            cprintf("do_pgfault failed: error code flag = read AND not present, but the
addr's vma cannot read or exec\n");
            goto failed;
        }
    }

    uint32_t perm = PTE_U;
    if (vma->vm_flags & VM_WRITE) {
        perm |= PTE_W;
    }
    addr = ROUNDDOWN(addr, PGSIZE);

    ret = -E_NO_MEM;

    pte_t *ptep=NULL;
#if 0
    /*LAB3 EXERCISE 1: YOUR CODE*/
    ptep = ???              //(1) try to find a pte, if pte's PT(Page Table) isn't existed,
then create a PT.
    if (*ptep == 0) {
                            //(2) if the phy addr isn't exist, then alloc a page & map the
phy addr with logical addr

    }
    else {
```

```
        if(swap_init_ok) {
            struct Page *page=NULL;
        }
        else {
            cprintf("no swap_init_ok but ptep is %x, failed\n",*ptep);
            goto failed;
        }
   }
#endif
    // try to find a pte, if pte's PT(Page Table) isn't existed, then create a PT.
    // (notice the 3th parameter '1')
    if ((ptep = get_pte(mm->pgdir, addr, 1)) == NULL) {
        cprintf("get_pte in do_pgfault failed\n");
        goto failed;
    }

    if (*ptep == 0) { // if the phy addr isn't exist, then alloc a page & map the phy addr
with logical addr
        if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) {
            cprintf("pgdir_alloc_page in do_pgfault failed\n");
            goto failed;
        }
    }
    else { // if this pte is a swap entry, then load data from disk to a page with phy addr
           // and call page_insert to map the phy addr with logical addr
        if(swap_init_ok) {
            struct Page *page=NULL;
            if ((ret = swap_in(mm, addr, &page)) != 0) {
                cprintf("swap_in in do_pgfault failed\n");
                goto failed;
            }
            page_insert(mm->pgdir, page, addr, perm);
            swap_map_swappable(mm, addr, page, 1);
        }
        else {
            cprintf("no swap_init_ok but ptep is %x, failed\n",*ptep);
            goto failed;
        }
   }
   ret = 0;
failed:
    return ret;
}
```

- 给进程分配内存算法的修改：

```
//在proc.c中修改
#include <proc.h>
#include <kmalloc.h>
#include <string.h>
#include <sync.h>
#include <pmm.h>
#include <error.h>
#include <sched.h>
#include <elf.h>
#include <vmm.h>
#include <trap.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <assert.h>

// the process set's list
list_entry_t proc_list;

#define HASH_SHIFT          10
#define HASH_LIST_SIZE      (1 << HASH_SHIFT)
#define pid_hashfn(x)       (hash32(x, HASH_SHIFT))

// has list for process set based on pid
static list_entry_t hash_list[HASH_LIST_SIZE];

// idle proc
struct proc_struct *idleproc = NULL;
// init proc
struct proc_struct *initproc = NULL;
// current proc
struct proc_struct *current = NULL;

static int nr_process = 0;

struct kmem_cache_t *proc_cache = NULL; //声明仓库指针
void kernel_thread_entry(void);
void forkrets(struct trapframe *tf);
void switch_to(struct context *from, struct context *to);

// alloc_proc - alloc a proc_struct and init all fields of proc_struct
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmem_cache_alloc(proc_cache);
    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;
        proc->pid = -1;
        proc->runs = 0;
        proc->kstack = 0;
        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);
```

```
    }
    return proc;
}

// set_proc_name - set the name of proc
char *
set_proc_name(struct proc_struct *proc, const char *name) {
    memset(proc->name, 0, sizeof(proc->name));
    return memcpy(proc->name, name, PROC_NAME_LEN);
}

// get_proc_name - get the name of proc
char *
get_proc_name(struct proc_struct *proc) {
    static char name[PROC_NAME_LEN + 1];
    memset(name, 0, sizeof(name));
    return memcpy(name, proc->name, PROC_NAME_LEN);
}

// 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;
}

// 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);
    }
}

// forkret -- the first kernel entry point of a new thread/process
// NOTE: the addr of forkret is setted in copy_thread function
//       after switch_to, the current proc will execute here.
static void
forkret(void) {
    forkrets(current->tf);
}

// hash_proc - add proc into proc hash_list
static void
hash_proc(struct proc_struct *proc) {
    list_add(hash_list + pid_hashfn(proc->pid), &(proc->hash_link));
}

// find_proc - find proc frome proc hash_list according to pid
struct proc_struct *
find_proc(int pid) {
    if (0 < pid && pid < MAX_PID) {
        list_entry_t *list = hash_list + pid_hashfn(pid), *le = list;
        while ((le = list_next(le)) != list) {
            struct proc_struct *proc = le2proc(le, hash_link);
            if (proc->pid == pid) {
                return proc;
            }
        }
    }
    return NULL;
}

// kernel_thread - create a kernel thread using "fn" function
// NOTE: the contents of temp trapframe tf will be copied to
//       proc->tf in do_fork-->copy_thread function
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);
}

// setup_kstack - alloc pages with size KSTACKPAGE as process kernel stack
static int
setup_kstack(struct proc_struct *proc) {
    struct Page *page = alloc_pages(KSTACKPAGE);
    if (page != NULL) {
        proc->kstack = (uintptr_t)page2kva(page);
```

```
        return 0;
    }
    return -E_NO_MEM;
}

// put_kstack - free the memory space of process kernel stack
static void
put_kstack(struct proc_struct *proc) {
    free_pages(kva2page((void *)(proc->kstack)), KSTACKPAGE);
}

// copy_mm - process "proc" duplicate OR share process "current"'s mm according clone_flags
//         - if clone_flags & CLONE_VM, then "share" ; else "duplicate"
static int
copy_mm(uint32_t clone_flags, struct proc_struct *proc) {
    assert(current->mm == NULL);
    /* do nothing in this project */
    return 0;
}

// copy_thread - setup the trapframe on the  process's kernel stack top and
//             - setup the kernel entry point and stack of process
static void
copy_thread(struct proc_struct *proc, uintptr_t esp, struct trapframe *tf) {
    proc->tf = (struct trapframe *)(proc->kstack + KSTACKSIZE) - 1;
    *(proc->tf) = *tf;
    proc->tf->tf_regs.reg_eax = 0;
    proc->tf->tf_esp = esp;
    proc->tf->tf_eflags |= FL_IF;

    proc->context.eip = (uintptr_t)forkret;
    proc->context.esp = (uintptr_t)(proc->tf);
}

/* do_fork -     parent process for a new child process
 * @clone_flags: used to guide how to clone the child process
 * @stack:       the parent's user stack pointer. if stack==0, It means to fork a kernel
thread.
 * @tf:          the trapframe info, which will be copied to child process's proc->tf
 */
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;
    proc = alloc_proc();
    if (proc == NULL) {
        goto fork_out;
    }

    proc->parent = current;

    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_proc;
    }
    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }
```

```
    copy_thread(proc, stack, tf);

    bool intr_flag = 0;
    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);

    wakeup_proc(proc);

    ret = proc->pid;
fork_out:
    return ret;

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

int
do_exit(int error_code) {
    panic("process exit!!.\n");
}

// 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;
}

// proc_init - set up the first kernel thread idleproc "idle" by itself and
//           - create the second kernel thread init_main
void
proc_init(void) {
    int i;
    proc_cache = kmem_cache_create("proc", sizeof(struct proc_struct), NULL, NULL);
    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);
}

// cpu_idle - at the end of kern_init, the first kernel thread idleproc will do below works
void
cpu_idle(void) {
    while (1) {
        if (current->need_resched) {
            schedule();
        }
    }
}
```

由于我在LAB2中实现的是伙伴系统，在实现Slub过程中参考了大量的资料（见参考资料），最后根据参考代码，自己补充完成了Slub的简要实现。

感想与体会

相较于LAB2的实验，由于有了前一次的实验的经验，这一次的实验做起来比较的顺利。本次实验加深了我对解内核线程创建以及执行的管理过程，对进程的创建、初始化PCB、分配内存等过程有了更深一步的理解；进一步的，此次实验还完成了对内核线程的切换和基本调度过程，了解了在进行上下文切换时操作系统需要做的具体操作。在实验过程中也遇到了很多bug，通过耐心的查找资料、查看各种宏定义一点点改正，与答案进行对比，发现自己的代码部分的鲁棒性需要进一步加强，让自己写的代码更完美。

参考资料

1. https://www.cnblogs.com/chaozhu/p/9668871.html
2. https://github.com/AngelKitty/review_the_national_post-graduate_entrance_examination/blob/master/books_and_notes/
3. https://www.cnblogs.com/tolimit/p/4654109.html

知识点小结

本实验中相关的知识点

- 进程状态模型
- 内核线程创建与初始化
- 进程切换相关知识（上下文等等）

本实验未涉及

- 处于挂起状态的进程处理