实验五：用户进程管理

一、实验目的

了解第一个用户进程创建过程

了解系统调用框架的实现机制

了解ucore如何实现系统调用sys\_fork/sys\_exec/sys\_exit/sys\_wait来进行进程管理

二、实验内容

实验4完成了内核线程，但到目前为止，所有的运行都在内核态执行。实验5将创建用户进 程，让用户进程在用户态执行，且在需要ucore支持时，可通过系统调用来让ucore提供服 务。为此需要构造出第一个用户进程，并通过系统调用sys\_fork/sys\_exec/sys\_exit/sys\_wait 来支持运行不同的应用程序，完成对用户进程的执行过程的基本管理。相关原理介绍可看附 录B。

练习

对实验报告的要求：

1.基于markdown格式来完成，以文本方式为主

2.填写各个基本练习中要求完成的报告内容

3.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

4.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）

5.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

**练习0：填写已有实验**

本实验依赖实验1/2/3/4。请把你做的实验1/2/3/4的代码填入本实验中代码中 有“LAB1”/“LAB2”/“LAB3”/“LAB4”的注释相应部分。注意：为了能够正确执行lab5的测试应用 程序，可能需对已完成的实验1/2/3/4的代码进行进一步改进。

**练习1: 加载应用程序并执行（需要编码）**

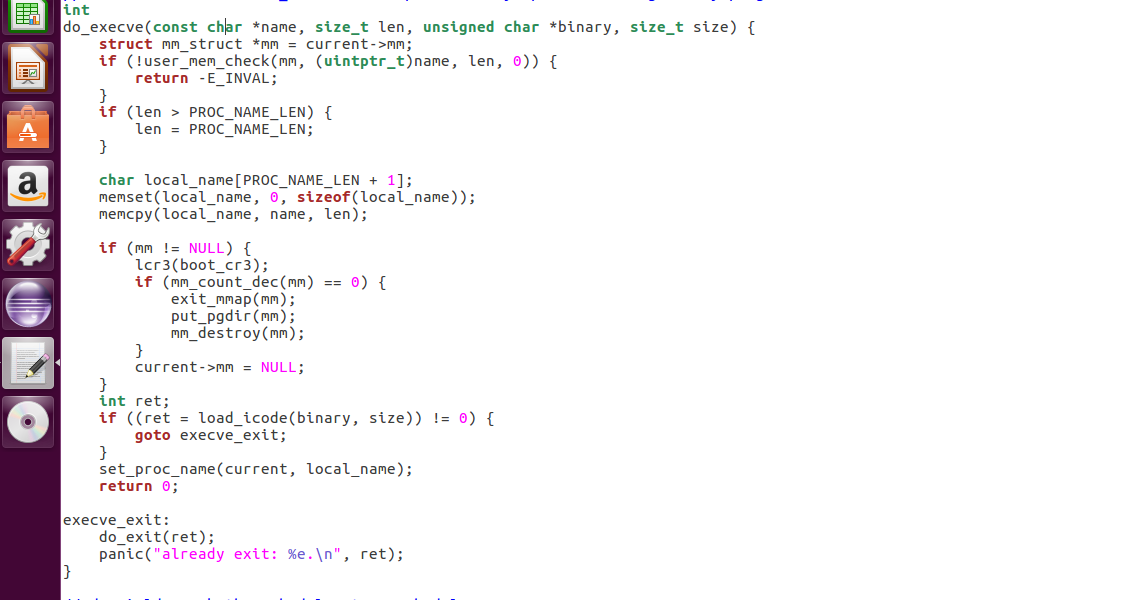
do\_execv函数调用load\_icode（位于kern/process/proc.c中）来加载并解析一个处于内存中 的ELF执行文件格式的应用程序，建立相应的用户内存空间来放置应用程序的代码段、数据段 等，且要设置好proc\_struct结构中的成员变量trapframe中的内容，确保在执行此进程后，能 够从应用程序设定的起始执行地址开始执行。需设置正确的trapframe内容。

请在实验报告中简要说明你的设计实现过程。

请在实验报告中描述当创建一个用户态进程并加载了应用程序后，CPU是如何让这个应用程 序最终在用户态执行起来的。即这个用户态进程被ucore选择占用CPU执行（RUNNING态） 到具体执行应用程序第一条指令的整个经过。

①

首先是do\_exceve函数，它调用了load\_icode去加载ELF二进制格式文件到内存并执行：（lab5\_result/kern/process/proc.c 653行起)



do\_execve函数主要做的工作就是先回收自身所占用户空间，然后调用load\_icode，用新的程序覆盖内存空间，形成一个执行新程序的新进程。do\_execv函数调用load\_icode来加载并解析一个处于内存中的ELF执行文件格式的应用程序，建立相应的用户内存空间来放置应用程序的代码段、数据段等，且要设置好proc\_struct结构中的成员变量trapframe中的内容，确保在执行此进程后，能够从应用程序设定的起始执行地址开始执行。需设置正确的trapframe内容。**load\_icode函数的实现和注释：(**lab5\_result/kern/process/proc.c 505行起**)**

**static int**

**load\_icode(unsigned char \*binary, size\_t size) {**

**if (current->mm != NULL) {**

**panic("load\_icode: current->mm must be empty.\n");**

**}//准备部分：当前进程必须为空，这样才能加载到内存。**

**int ret = -E\_NO\_MEM;**

**struct mm\_struct \*mm;//声明一个页表**

**//(1) create a new mm for current process**

**if ((mm = mm\_create()) == NULL) {//申请内存**

**goto bad\_mm;**

**}**

**//(2) create a new PDT, and mm->pgdir= kernel virtual addr of PDT**

**if (setup\_pgdir(mm) != 0) {**

**goto bad\_pgdir\_cleanup\_mm;**

**}**

**//(3) copy TEXT/DATA section, build BSS parts in binary to memory space of process**

**struct Page \*page;**

**//(3.1) get the file header of the bianry program (ELF format)**

**struct elfhdr \*elf = (struct elfhdr \*)binary;**

**//(3.2) get the entry of the program section headers of the bianry program (ELF format)**

**struct proghdr \*ph = (struct proghdr \*)(binary + elf->e\_phoff);**

**//(3.3) This program is valid?**

**if (elf->e\_magic != ELF\_MAGIC) {**

**ret = -E\_INVAL\_ELF;**

**goto bad\_elf\_cleanup\_pgdir;**

**}**

**uint32\_t vm\_flags, perm;**

**struct proghdr \*ph\_end = ph + elf->e\_phnum;**

**for (; ph < ph\_end; ph ++) {**

**//(3.4) find every program section headers**

**if (ph->p\_type != ELF\_PT\_LOAD) {**

**continue ;**

**}**

**if (ph->p\_filesz > ph->p\_memsz) {**

**ret = -E\_INVAL\_ELF;**

**goto bad\_cleanup\_mmap;**

**}**

**if (ph->p\_filesz == 0) {**

**continue ;**

**}**

**//(3.5) call mm\_map fun to setup the new vma ( ph->p\_va, ph->p\_memsz)**

**vm\_flags = 0, perm = PTE\_U;**

**if (ph->p\_flags & ELF\_PF\_X) vm\_flags |= VM\_EXEC;**

**if (ph->p\_flags & ELF\_PF\_W) vm\_flags |= VM\_WRITE;**

**if (ph->p\_flags & ELF\_PF\_R) vm\_flags |= VM\_READ;**

**if (vm\_flags & VM\_WRITE) perm |= PTE\_W;**

**if ((ret = mm\_map(mm, ph->p\_va, ph->p\_memsz, vm\_flags, NULL)) != 0) {**

**goto bad\_cleanup\_mmap;**

**}**

**unsigned char \*from = binary + ph->p\_offset;**

**size\_t off, size;**

**uintptr\_t start = ph->p\_va, end, la = ROUNDDOWN(start, PGSIZE);**

**ret = -E\_NO\_MEM;**

**//(3.6) alloc memory, and copy the contents of every program section (from, from+end) to process's memory (la, la+end)**

**end = ph->p\_va + ph->p\_filesz;**

**//(3.6.1) copy TEXT/DATA section of bianry program**

**while (start < end) {**

**if ((page = pgdir\_alloc\_page(mm->pgdir, la, perm)) == NULL) {**

**goto bad\_cleanup\_mmap;**

**}**

**off = start - la, size = PGSIZE - off, la += PGSIZE;**

**if (end < la) {**

**size -= la - end;**

**}**

**memcpy(page2kva(page) + off, from, size);**

**start += size, from += size;**

**}**

**//(3.6.2) build BSS section of binary program**

**end = ph->p\_va + ph->p\_memsz;**

**if (start < la) {**

**/\* ph->p\_memsz == ph->p\_filesz \*/**

**if (start == end) {**

**continue ;**

**}**

**off = start + PGSIZE - la, size = PGSIZE - off;**

**if (end < la) {**

**size -= la - end;**

**}**

**memset(page2kva(page) + off, 0, size);**

**start += size;**

**assert((end < la && start == end) || (end >= la && start == la));**

**}**

**while (start < end) {**

**if ((page = pgdir\_alloc\_page(mm->pgdir, la, perm)) == NULL) {**

**goto bad\_cleanup\_mmap;**

**}**

**off = start - la, size = PGSIZE - off, la += PGSIZE;**

**if (end < la) {**

**size -= la - end;**

**}**

**memset(page2kva(page) + off, 0, size);**

**start += size;**

**}**

**}**

**//(4) build user stack memory**

**//除了数据段、代码段、进程还需要用户堆栈空间。这里是构造用户堆栈。**

**vm\_flags = VM\_READ | VM\_WRITE | VM\_STACK;**

**if ((ret = mm\_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm\_flags, NULL)) != 0) {**

**goto bad\_cleanup\_mmap;**

**}**

**assert(pgdir\_alloc\_page(mm->pgdir, USTACKTOP-PGSIZE , PTE\_USER) != NULL);**

**assert(pgdir\_alloc\_page(mm->pgdir, USTACKTOP-2\*PGSIZE , PTE\_USER) != NULL);**

**assert(pgdir\_alloc\_page(mm->pgdir, USTACKTOP-3\*PGSIZE , PTE\_USER) != NULL);**

**assert(pgdir\_alloc\_page(mm->pgdir, USTACKTOP-4\*PGSIZE , PTE\_USER) != NULL);**

**//(5) set current process's mm, sr3, and set CR3 reg = physical addr of Page Directory**

**mm\_count\_inc(mm);**

**current->mm = mm;**

**current->cr3 = PADDR(mm->pgdir);**

**lcr3(PADDR(mm->pgdir));**

**//(6) setup trapframe for user environment**

**struct trapframe \*tf = current->tf;**

**memset(tf, 0, sizeof(struct trapframe));**

**/\* LAB5:EXERCISE1 YOUR CODE**

**\* should set tf\_cs,tf\_ds,tf\_es,tf\_ss,tf\_esp,tf\_eip,tf\_eflags**

**\* NOTICE: If we set trapframe correctly, then the user level process can return to USER MODE from kernel. So**

**\* tf\_cs should be USER\_CS segment (see memlayout.h)**

**\* tf\_ds=tf\_es=tf\_ss should be USER\_DS segment**

**\* tf\_esp should be the top addr of user stack (USTACKTOP)**

**\* tf\_eip should be the entry point of this binary program (elf->e\_entry)**

**\* tf\_eflags should be set to enable computer to produce Interrupt**

**\*/**

*/***/完成一个优先级的转变，从内核态切换到用户态（特权级从0到3）实现部分**

**tf->tf\_cs = USER\_CS;**

**tf->tf\_ds = tf->tf\_es = tf->tf\_ss = USER\_DS;**

**tf->tf\_esp = USTACKTOP;**

**tf->tf\_eip = elf->e\_entry;**

**tf->tf\_eflags = FL\_IF;**

**\*tf是一个是中断帧的指针，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。其定义在（kern/trap/trap.h，60——82行）。**

**1、将tf\_cs设置为用户态，这个定义在（kern/mm/memlayout.h，第21行），有一个宏定义已经定义了用户态和内核态。**

**2、tf\_ds=tf\_es=tf\_ss也需要设置为用户态：定义在（kern/mm/memlayout.h，第26行）**

**3、需要将esp设置为用户栈的栈顶，直接使用之前建立用户栈时的参数USTACKTOP就可以。**

**4、eip是程序的入口，elf类的e\_entry函数直接声明了，直接使用。**

**5、FL\_IF打开中断。**

**ret = 0;**

**out:**

**return ret;**

**bad\_cleanup\_mmap:**

**exit\_mmap(mm);**

**bad\_elf\_cleanup\_pgdir:**

**put\_pgdir(mm);**

**bad\_pgdir\_cleanup\_mm:**

**mm\_destroy(mm);**

**bad\_mm:**

**goto out;**

**}**

**练习2: 父进程复制自己的内存空间给子进程（需要编码）**

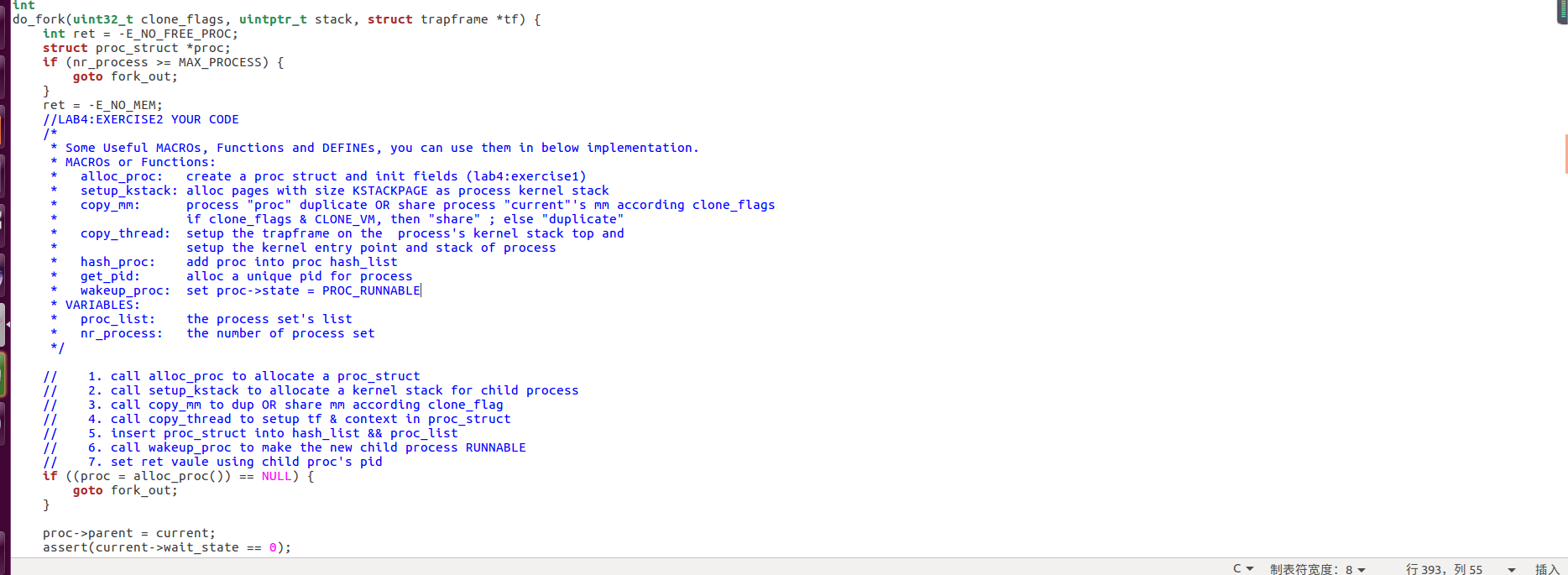
创建子进程的函数do\_fork在执行中将拷贝当前进程（即父进程）的用户内存地址空间中的合 法内容到新进程中（子进程），完成内存资源的复制。具体是通过copy\_range函数（位于 kern/mm/pmm.c中）实现的，请补充copy\_range的实现，确保能够正确执行。

请在实验报告中简要说明如何设计实现”Copy on Write机制“，给出概要设计，鼓励给出详细 设计。

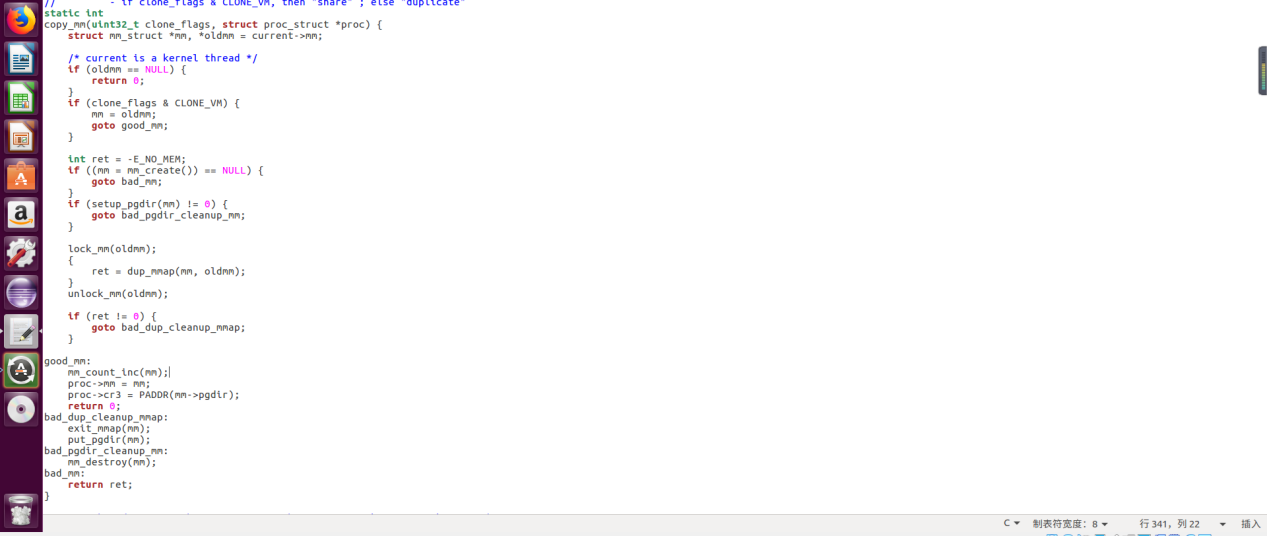
Copy on Write：创建子进程的时候能够共享父进程的内存空间，节省内存占用。

copy\_range函数的调用过程：do\_fork()---->copy\_mm()---->dup\_mmap()---->copy\_range()

**do\_fork函数**调用的copy\_mm是创建一个进程，并放入CPU中调度，而本次我们主要关注的是父子进程之间如何拷贝内存。



**copy\_mm函数：**



其中， 有一个互斥锁，用于避免多个进程同时访问内存，在这里进行了下一层调用：

lock\_mm(oldmm);//互斥锁

{

ret = dup\_mmap(mm, oldmm);

}

unlock\_mm(oldmm);

**dup\_mmap函数：（kern/mm/vmm.c，191行）**

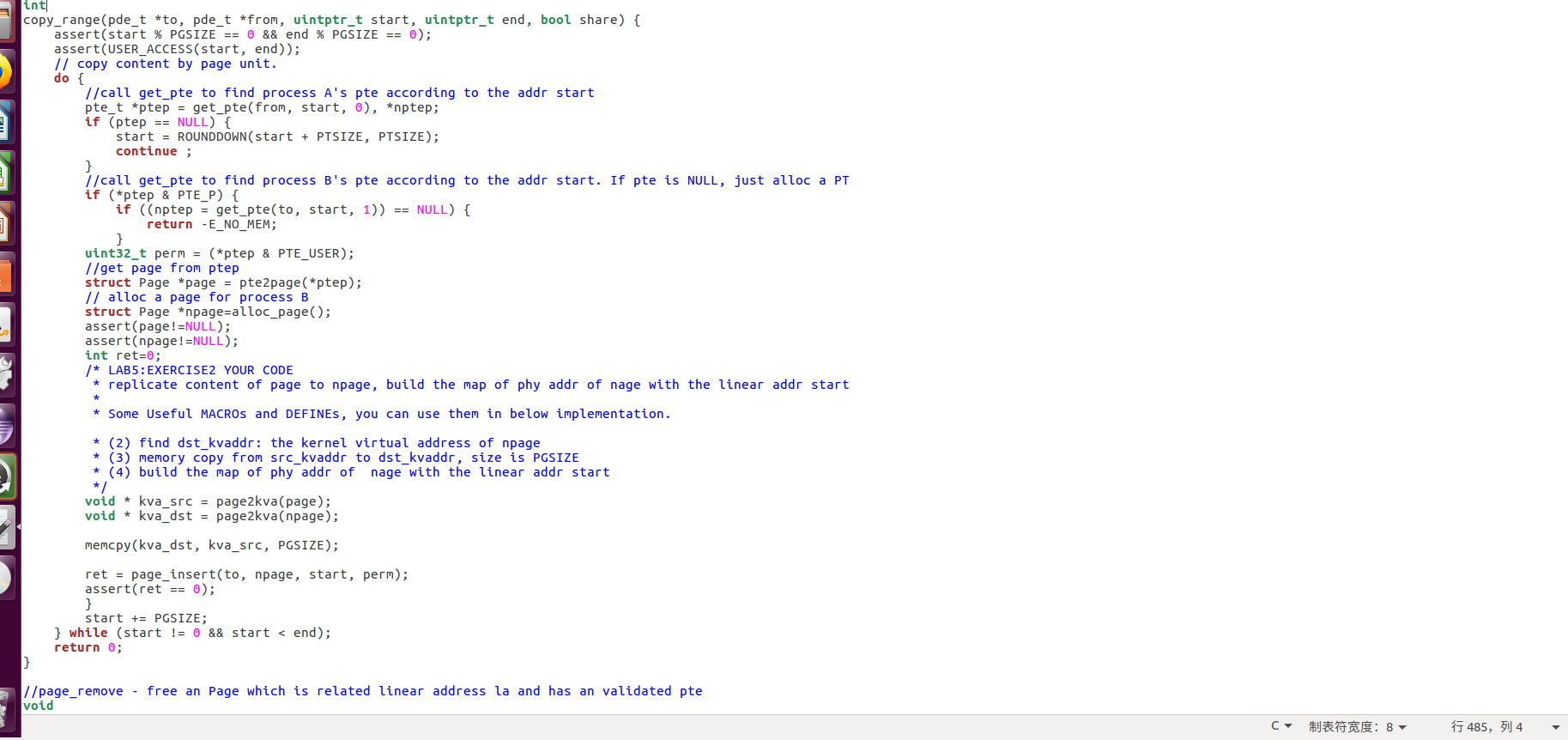
传入的参数 ，是两个mm

在上一个函数copy\_mm中，传入的两个内存叫做mm和oldmm，其中，第一个mm只是调用了mm\_create()声明，但没有初始化，更没有分配内容；第二个oldmm是current进程的内存空间，由此可见，前一个mm是待复制的内存，而复制的源内容在oldmm（父进程）内容中。



**copy\_range函数：（kern/mm/pmm.c）**

在上一个函数中（dup\_mmap），只是完成了新进程中的段创建，但是段中还没有具体内容，需要在copy\_range中具体复制父进程对应段中的具体内容。这个函数传入的参数都是段指针，告诉系统应该复制内存中需要复制内容的起止地址。



**练习3**: 阅读分析源代码，理解进程执行 fork/exec/wait/exit 的实 现，以及系统调用的实现（不需要编码）

Fork：

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);

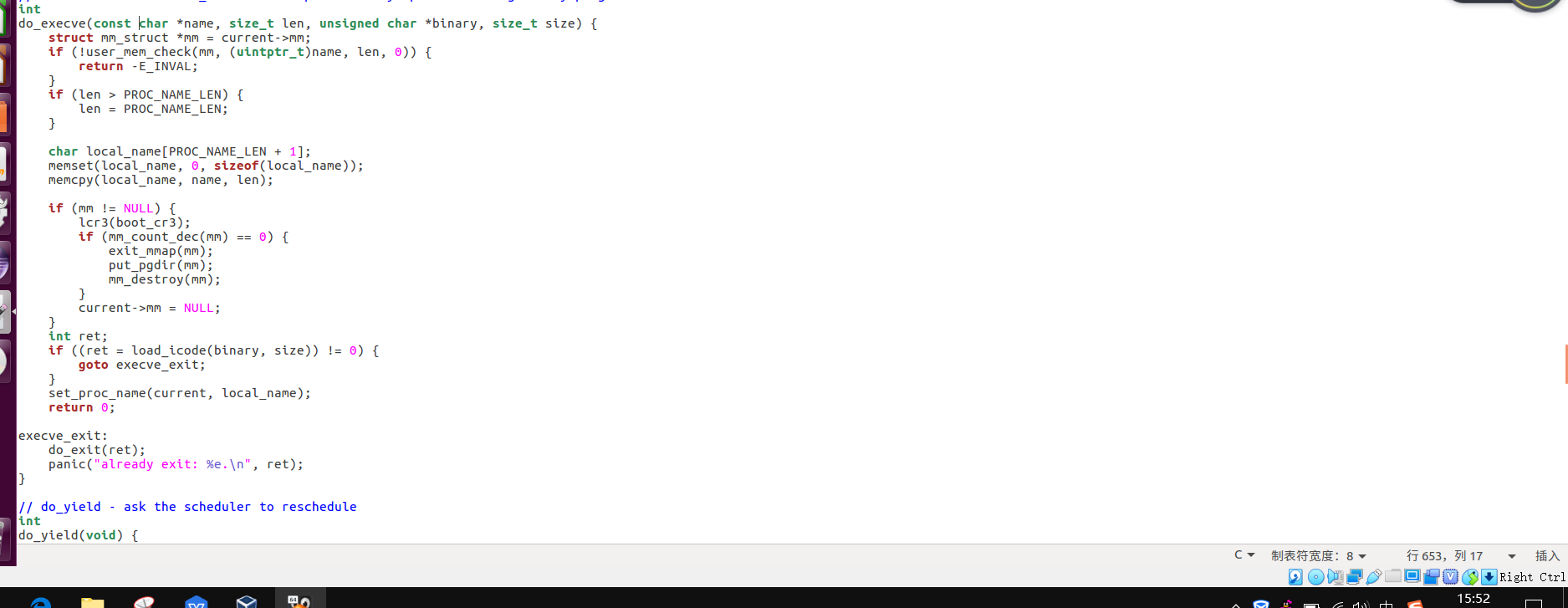
3、根据 clone\_flag标志复制或共享进程内存管理结构(copy\_mm 函数);

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。

**do\_execve：（kern/process/proc 652 ）：**

首先为加载新的执行码做好用户态内存空间清空准备。如果mm不为NULL，则设置页表为内核空间页表。

接下来是加载应用程序执行码到当前进程的新创建的用户态虚拟空间中。之后就是调用。

**do\_wait函数：（kern/process/proc.c， 698）：**





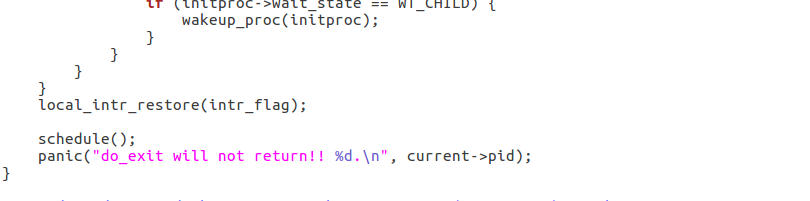
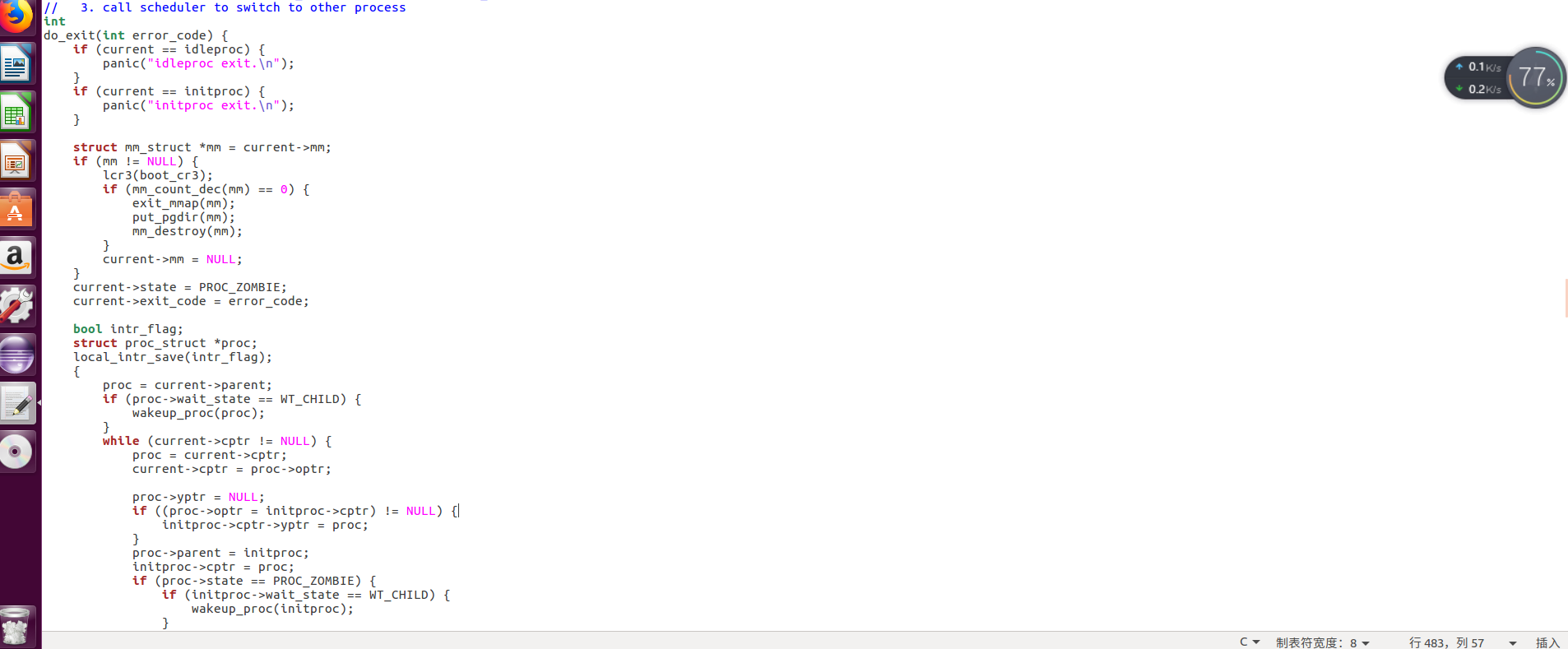
当执行wait功能的时候，会调用系统调用SYS\_wait，而该系统调用的功能则主要由do\_wait函数实现，主要工作就是父进程如何完成对子进程的最后回收工作，具体的功能实现如下：

1、 如果 pid!=0，表示只找一个进程 id 号为 pid 的退出状态的子进程，否则找任意一个处于退出状态的子进程;

2、 如果此子进程的执行状态不为PROC\_ZOMBIE，表明此子进程还没有退出，则当前进程设置执行状态为PROC\_SLEEPING（睡眠），睡眠原因为WT\_CHILD(即等待子进程退出)，调用schedule()函数选择新的进程执行，自己睡眠等待，如果被唤醒，则重复跳回步骤 1 处执行;

3、 如果此子进程的执行状态为 PROC\_ZOMBIE，表明此子进程处于退出状态，需要当前进程(即子进程的父进程)完成对子进程的最终回收工作，即首先把子进程控制块从两个进程队列proc\_list和hash\_list中删除，并释放子进程的内核堆栈和进程控制块。自此，子进程才彻底地结束了它的执行过程，它所占用的所有资源均已释放。

**do\_exit函数：（kern/process/proc.c，448）：**

**实验六: 调度器**

实验目的

1.理解操作系统的调度管理机制

2.熟悉 ucore 的系统调度器框架，以及缺省的Round-Robin 调度算法

3.基于调度器框架实现一个(Stride Scheduling)调度算法来替换缺省的调度算法

**练习0：填写已有实验**

本实验依赖实验1/2/3/4/5。请把你做的实验2/3/4/5的代码填入本实验中代码中 有“LAB1”/“LAB2”/“LAB3”/“LAB4”“LAB5”的注释相应部分。并确保编译通过。注意：为了能够 正确执行lab6的测试应用程序，可能需对已完成的实验1/2/3/4/5的代码进行进一步改进。

**练习1: 使用 Round Robin 调度算法（不需要编码）**

完成练习0后，建议大家比较一下（可用kdiff3等文件比较软件）个人完成的lab5和练习0完成 后的刚修改的lab6之间的区别，分析了解lab6采用RR调度算法后的执行过程。执行make grade，大部分测试用例应该通过。但执行priority.c应该过不去。

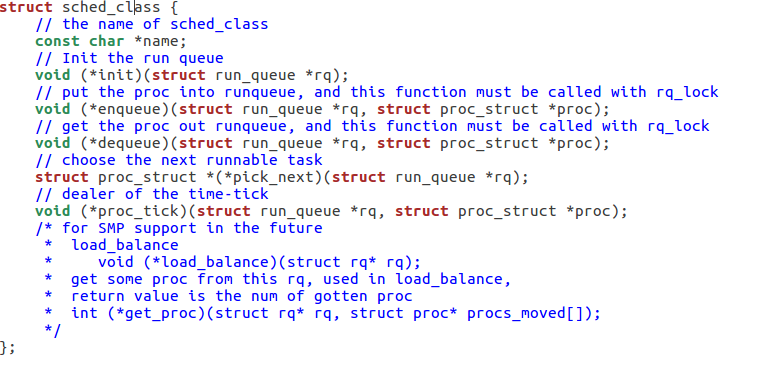
请在实验报告中完成：

请理解并分析sched\_calss中各个函数指针的用法，并接合Round Robin 调度算法描 ucore的调度执行过程 请在实验报告中简要说明如何设计实现”多级反馈队列调度算法“，给出概要设计，鼓励给 出详细设计

Round Robin调度算法（简称RR，轮转调度）的调度思想是让所有 runnable 态的进程分时轮流使用 CPU 时间。调度器维护当前 runnable进程的有序运行队列。当前进程的时间片用完之后,调度器将当前进程放置到运行队列的尾部，再从其头部取出进程进行调度。

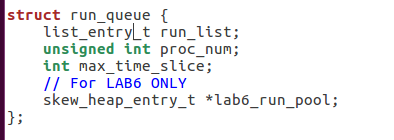
首先，本实验中的调度都是基于调度类的成员函数实现的，相关定义在

（lab6\_result/kern/schedule/sched.h，17），定义了几个成员函数。



Init：是初始化环节，初始化rq的进程队列，并将其进程数量置零。

其中，struct run\_queue的定义如下：（kern/schedule/sched.h，39行）;



enqueue：是一个进程入队的操作：进程队列是一个双向链表，一个进程加入队列的时候，会将其加入到队列的第一位，并给它初始数量的时间片；并更新队列的进程数量。

dequeue：从就绪队列中取出这个进程，并将其调用list\_del\_init删除。同时，进程数量减一。

pick\_next：通过list\_next函数的调用，会从队尾选择一个进程，代表当前应该去执行的那个进程。如果选不出来有处在就绪状态的进程，那么返回NULL，并将执行权交给内核线程idle，idle的功能是不断调用schedule，直到整个系统出现下一个可以执行的进程。

第五部分：proc\_tick：产生时钟中断的时候，会触发tick函数的调用，对应于上图中调度点的第六种情况。

在schedule初始化的时候，需要填写一个初始化信息，那么这里就填上我们所实现的类函数，那么系统就可以按照这个方式去执行了。

调度初始化的函数sched\_init被定义在（kern/schedule/sched.c，45行）：

如下所示，将sched\_class设置为刚刚定义的类名，就可以完成初始化绑定。

