ucoreOS操作系统试验——Lab5

问题发现&&改进

- 这次连练习0都不是command+cv就可以解决的了_(:з」∠)_，而且额外补充和添加了许多的内容，而且从老师补充的地方以及我们自己改进的地方可以看出这次实验就是一个更甚过lab4的综合了——终于从核心态出来开始弄用户态了，运用到了许多前面的知识，所以要复习复习，这里记录一下新增加的内容吧~
  - kdebug.c解析用户进程的符号信息表示（可不用理会）主要是stab数据结构以及信息处理
  - memlayout.h：修改：增加了用户虚存地址空间的图形表示和宏定义 （需仔细理解）
    - **在用户虚存空间与KERNBASE中间以及用户虚存空间内部分隔都有一段Invalid Memory，它永远也不会被映射，和Empty Memory不一样，如果非常必须，Empty是会被映射的**
  - pmm.[ch]：修改：添加了用于进程退出（do_exit）的内存资源回收的page_remove_pte、unmap_range、exit_range函数和用于创建子进程（do_fork）中拷贝父进程内存空间的copy_range函数，修改了pgdir_alloc_page函数
    - unmap_range、exit_range都是负责把那些未映射的已经退出进程的内存页free回收放入空闲页中
    - copy_range是在do_fork时调用的，用于复制一整个进程的mm内容到fork出来的进程中(练习二)
    - 对pgdir_alloc_page的修改是加了一段检查mm是否为空以后对应的处理，但是注释掉是怎么回事。。
  - vmm.[ch]：修改：扩展了mm_struct数据结构，增加了一系列函数
    - mm_map/dup_mmap/exit_mmap：设定/取消/复制/删除用户进程的合法内存空间
    - copy_from_user/copy_to_user：用户内存空间内容与内核内存空间内容的相互拷贝的实现
    - user_mem_check：搜索vma链表，检查是否是一个合法的用户空间范围
    - **这里对页表的内容进行扩展，并且能够把部分物理内存映射为用户态虚拟内存**
  - proc.[ch]：修改：扩展了proc_struct数据结构。增加或修改了一系列函数
    - setup_pgdir/put_pgdir：创建并设置/释放页目录表
    - copy_mm：复制用户进程的内存空间和设置相关内存管理（如页表等）信息(因为上一个lab共用同一个内存空间，所以虽然调用了这个函数，但当时它是一个空函数)
    - do_exit：释放进程自身所占内存空间和相关内存管理（如页表等）信息所占空间，唤醒父进程，好让父进程收回自己，并让调度器切换到其他进程
    - do_execve：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
    - load_icode：被do_execve调用，完成加载放在内存中的执行程序到进程空间，这涉及到对页表等的修改，分配用户栈
    - do_yield：让调度器执行一次选择新进程的过程
    - do_wait：父进程等待子进程，并在得到子进程的退出消息后，彻底回收子进程所占的资源（比如子进程的内核栈和进程控制块）
    - do_kill：给一个进程设置PF_EXITING标志（“kill”信息，即要它死掉），这样在trap函数中，将根据此标志，让进程退出
    - KERNEL_EXECVE/**KERNEL_EXECVE**/KERNEL_EXECVE2：被user_main调用，执行一用户进程
  - trap.c：修改：在idt_init函数中，对IDT初始化时，设置好了用于系统调用的中断门（idt[T_SYSCALL]）信息。这主要与syscall的实现相关 就是我们练习0要做的但其实lab1就顺手做了的部分
- 服了，这次的 `make grade` 又愉快的报了错
  - 又是一堆missing什么的，果不其然，上次改过的success和succeed!又出现了kmalloc.c里面的 `check_slab() success` 改成 `check_slab() succeeded!` 就好了_(:з」∠)_或者也许可以悄悄的改一改grade.sh还挺简单(*^▽^*)
  - 还有一个 `init check memory pass.` 的missing后来在issues的fix bug找到了原因，照着把proc.c的check改掉

并且替换一下grade.sh就好了。。大概是有一个变量的重复调用然后assert以后不对了直接退出所以没来得及输出上面那句话

- do_exit()里面用 `current->mm != NULL` 来区分用户和内核进程，确实很妙啊，毕竟内核进程都在内存空间，并没有新的mm

练习0

> 复制粘贴永不出错，而且这次的练习0他们再也不能打patch了！

但是这一次练习0说为了保证lab5不出错，要对先前的代码进行进一步改进，这简直大海捞针？？所以现在连练习0都不可以无脑完成了是吗？幸好项目组成以及后续对实验执行的流程讲解(help_comment)中都告诉了我们需要了解哪些重要的改动以及需要做哪些改动。

主要是以下几个方面：

- 在初始化IDT的时候，设置系统调用对应的中断描述符，使其能够在用户态下被调用，并且设置为trap类型

```
//设置给用户态用的中断门 让用户态能够进行系统调用trap.c  idit_init
//但其实lab1已经顺手做了
SETGATE(idt[T_SWITCH_TOK], 1, KERNEL_CS, __vectors[T_SWITCH_TOK], 3);
```

- 在时钟中断的处理部分，每过TICK_NUM个中断，就将当前的进程设置为可以被重新调度的，这样使得当前的线程可以被换出，从而实现多个线程的并发执行

```
case IRQ_OFFSET + IRQ_TIMER:
  if (++ticks % TICK_NUM == 0) {// 时间片用完 设置进程为需要被调度
        assert(current != NULL);
        current->need_resched = 1;}
  break;   //trap.c   原先写过print_ticks处
```

- proc_alloc中，额外对进程控制块中新增加的wait_state, cptr, yptr, optr成员变量进行初始化

```
proc->wait_state = 0;
proc->cptr = proc->yptr = proc-> optr = NULL;
//父进程            弟弟进程       哥哥进程       proc.c
```

- do_fork中，使用set_links函数来完成将fork得到的线程添加到线程链表中的过程

```
// 为新线程分配PCB
if ((proc = alloc_proc()) == NULL)
    go 正在加载... t; // 判断是否分配到内存空间
proc->parent = current;
assert(current->wait_state == 0);//New 确保当前进程正在等待
assert(setup_kstack(proc) == 0);
// 设置内核栈
assert(copy_mm(clone_flags, proc) == 0);
// 对虚拟内存空间进行拷贝，lab4内核线程之间共享一个虚拟内存空间，所以并不需要进行任何操作
copy_thread(proc, stack, tf);
// 在内核栈上面设置伪造好的tf，便于利用iret命令将控制权转移给新的线程
proc->pid = get_pid(); // 创建pid
hash_proc(proc); // 将线程放入使用hash表，加速查找
// nr_process ++; // 全局线程数加1  New
set_links(proc);//New      将原来的简单计数改成设置进程的相关链接
//list_add(&proc_list, &proc->list_link); //New  将线程加入链表
wakeup_proc(proc); // 唤醒线程，即将该线程的状态设置为可以运行
ret = proc->pid; // 返回新线程的pid
```

练习一

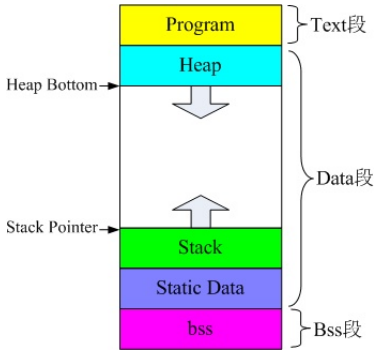
> 加载应用程序并执行（需要编码）
>
> do_execv函数调用load_icode（位于kern/process/proc.c中）来加载并解析一个处于内存中的ELF执行文件格式的应用程序，建立相应的用户内存空间来放置应用程序的代码段、数据段等，且要设置好proc_struct结构中的成员变量trapframe中的内容，确保在执行此进程后，能够从应用程序设定的起始执行地址开始执行。需设置正确的trapframe内容。

按照题目要求以及help_comment，只需要对trapeframe进行设置就可以完成，但是首先先理解一下load_icode做了些什么

load_icode函数的功能在于为执行新的程序初始化好内存空间，在调用该函数之前，do_execv中已经退出了当前进程的内存空间，改使用了内核的内存空间，这样就可以对原先用户态的内存空间进行操作

BSS段：

BSS（Block Started by Symbol segment）通常是指用来存放程序中未初始化的全局变量的一块内存区域，一般在初始化时bss 段部分将会清零，是由系统初始化的，不占用可执行文件的空间



load_icode执行流程：

- 给要执行的用户进程创建一个新的内存管理结构mm，原先该进程的mm已经在do_execv中被释放掉了

- 创建用户内存空间的新的页目录表PDT
- 将磁盘上的ELF文件的TEXT/DATA/BSS段正确地加载到用户空间中
    - 从磁盘中读取elf文件的header
    - 根据elfheader中的信息，获取到磁盘上的program header
    - 对于每一个program header:
        - 为TEXT/DATA段在用户内存空间上的保存分配物理内存页，同时建立物理页和虚拟页的映射关系
        - 从磁盘上读取TEXT/DATA段，并且复制到用户内存空间上去；
        - 根据program header得知是否需要创建BBS段，如果是，则分配相应的内存空间，并且全部初始化成0，并且建立物理页和虚拟页的映射关系
- 将用户栈的虚拟空间设置为合法，并且为栈顶部分先分配4个物理页，建立好映射关系
- 切换到用户地址空间
- 设置好用户栈上的信息，即需要传递给执行程序的参数
- 设置好中断帧，保证应用程序可以从设定的开始地址执行

lab5我们只做第六部分，如下

```
//(6) setup trapframe for user environment
struct trapframe *tf = current->tf;
memset(tf, 0, sizeof(struct trapframe));    //清空进程原先的中断帧
/* 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
 *          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
 */
tf->tf_cs = USER_CS;     //中断帧的代码段和数据段分别设置为用户态的段选择子
tf->tf_ds = tf->tf_es = tf->tf_ss = USER_DS;
tf->tf_esp = USTACKTOP;       //esp指向先前创建的用户栈栈顶;
tf->tf_eip = elf->e_entry;     //eip指向ELF可执行文件加载到内存之后的入口处
tf->tf_eflags = 0x00000002 | FL_IF; // eflags初始化为中断，保证可以响应中断
ret = 0;
```

回答问题

> 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。

- 在经过调度器占用了CPU的资源之后，用户态进程调用了exec系统调用，从而转入到了系统调用的处理例程
- 在经过了正常的中断处理例程之后，最终控制权转移到了syscall.c的syscall,根据系统调用号转移给sys_exec，在该函数中调用了练习一的do_execv完成指定应用程序的加载
- 在do_execve中进行了若干设置，退出当前进程的页表，换用内核PDT后，使用load_icode函数完成整个用户线程内存空间的初始化，包括堆栈的设置以及将ELF可执行文件的加载，之后通过current->tf指针修改当前系统调用的trapframe，使得最终中断返回的时候能够切换到用户态，并且同时可以正确地将控制权转移到应用程序的入口处
- 进行正常的中断返回的流程，由于中断处理例程的栈上面的eip已经被修改成了应用程序的入口处，而cs上的CPL(当前特权级——低两位)是用户态，因此iret进行中断返回的时候会将堆栈切换到用户的栈，并且完成特权级的切换，并且跳转到要求的应用程序的入口处

- 之后执行应用程序的第一条指令

练习二

> 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。

这个问题是父进程调用fork产生子进程的一个步骤，那么首先就分析一下父进程是如何生成子进程的

- 父进程调用fork的系统调用，进入正常的中断处理机制并最终交由syscall进行处理
- syscall中根据系统调用交由sys_fork处理
- sys_fork调用do_fork，实现了创建子进程、将父进程的内存空间复制给子进程
- do_fork中，调用copy_mm进行内存空间的复制，copy_mm调用了dup_mmap，dup_mmap遍历了父进程的所有合法虚拟内存空间，并且将这些空间的内容复制到子进程的内存空间，最后具体进行内存复制的函数就是需要完善的copy_range
- copy_range函数中，对需要复制的内存空间以页为单位从父进程的内存空间复制到子进程的内存空间中

copy_range执行流程如下

- 遍历父进程指定的某一段内存空间中的每一个虚拟页
- 如果虚拟页存在，为子进程对应的同一个地址（因为PDT不同所以不是同一个mm）申请分配一个物理页并将前者中的所有内容复制到后者中
- 为子进程的这个物理页和对应的虚拟地址（la）建立映射关系

- 练习二需要完成第三步的 复制和映射：
  - 找到父进程指定的某一物理页对应的内核虚拟地址
  - 找到需要拷贝过去的子进程的对应物理页对应的内核虚拟地址
  - 将前者的内容拷贝到后者中去；
  - 为子进程当前分配的这一物理页映射上对应在子进程虚拟地址空间里的虚拟页

  > 这里是一页一页进行复制的，范围从start→end

实现如下

```
// 找到父进程需要复制的物理页在内核地址空间中的虚拟地址，因为这个函数执行的时候用的是内核地址空间
uintptr_t src_kvaddr = page2kva(page);
uintptr_t dst_kvaddr = page2kva(npage);//找到子进程需要被填充的物理页的内核虚拟地址
memcpy(dst_kvaddr, src_kvaddr, PGSIZE);
page_insert(to, npage, start, perm);//建立子进程的物理页与虚拟页的映射关系
```

问题回答

> 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。

COW机制主要是①进程执行fork系统调用进行复制的时候，父进程并不是简单地将整个内存中的内容复制给子进程，而是暂时共享相同的物理内存页；②当其中一个进程需要对内存进行修改的时候，再额外创建一个自己私有的物理内存页，将共享的内容复制过去，并在自己的内存页中进行修改

所以对实验框架的修改应当主要有两个部分，①进行fork操作的时候不直接复制内存，而是将状态设置为共享②出现了内存页访问异常的时，将共享的内存页复制一份，在新的内存页进行修改

- do_fork：在进行内存复制的部分，如copy_range函数内部，不进行内存的复制，而是**将子进程和父进程的虚拟页映射上同一个物理页面**，分别在这两个进程的虚拟页对应的PTE部分将该页置为不可写，同时将这个页设置为一个共享页面，如此设置后如果应用程序试图写某一个共享页就会出现页访问异常，从而可以让操作系统进行page_fault的处理
- do_pgfault：
  - 在ISR部分，新增对当前异常是否由于尝试写了某一个共享页面引起的判断，是则额外申请分配一个物理页面，并将当前共享页的内容复制过去，建立出错的线性地址与新创建的物理页面的映射关系，将PTE置为非共享的
  - 查询原先共享的物理页面是否仍由多个其他进程共享使用的，不是则将对应虚地址的PTE进行修改，恢复写标记

> COW有一个缺点：如果在fork()之后，父子进程都还需要继续进行写操作，那么会产生大量的分页错误(页异常中断page-fault)，这样性能反而不会变好，就是得不偿失了

练习三

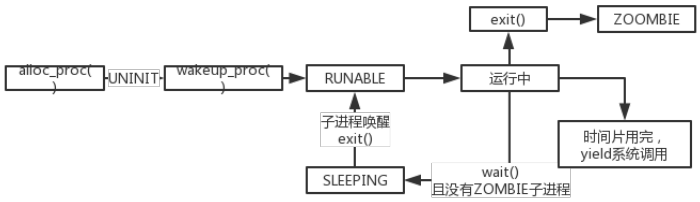
> 阅读分析源代码，理解进程执行 fork/exec/wait/exit 的实现，以及系统调用的实现

- fork：
  - 检查当前总进程数目是否到达限制，如果到达限制，那么返回 `E_NO_FREE_PROC`
  - 调用 `alloc_proc` 申请一个初始化后的进程控制块PCB
  - 调用 `setup_kstack` 为进程建立栈空间
  - 调用 `copy_mm` 拷贝或者共享内存空间
  - 调用 `copy_thread` 建立trapframe以及上下文
  - 调用 `get_pid()` 为进程分配一个PID
  - 将PCB加入哈希表和链表
  - 返回进程的PID
- exec：
  - 检查进程名称的地址和长度是否合法，如果合法，那么将名称暂时保存在函数栈中
  - 原先的内存内容不再需要，将进程的内存全部释放
  - 调用 `load_icode` 将代码加载进内存，如果加载错误，那么调用 `panic` 报错
  - 调用 `set_proc_name` 设置进程名称
- wait：
  - 检查用于保存返回码的 `code_store` 指针地址在合法的范围内
  - 根据PID找到需要等待的子进程PCB：
    - 如果没有需要等待的子进程，那么返回 `E_BAD_PROC`
    - 如果子进程正在可执行状态中，那么将当前进程休眠，在被唤醒后再次尝试
    - 如果子进程处于僵尸状态，那么回收PCB
- exit：
  - 释放进程的虚拟内存空间
  - 设置当前进程状态为 `PROC_ZOMBIE` 并同时设置返回码
  - 如果父进程等待当前进程，那么将父进程唤醒
  -
  - 将当前进程的所有子进程变为init的子进程
  - 主动调用调度函数进行调度
- 系统调用
  - 在idt_init里初始化中断描述符表并设置一个特定中断号的中断门用于syscall
  - 应用程序指令将需要使用的系统调用编号放入eax寄存器(并且返回结果也是eax)，系统调用可以有5个参数分别放于edx、ebx、ecx、edi、esi中
  - 使用 `INT 0x80` 指令进入内核态
  - 操作系统根据中断号0x80得知是系统调用时，根据系统调用号和参数执行相应的操作

- 回答问题

- 请分析fork/exec/wait/exit在实现中是如何影响进程的执行状态的?
- 请给出ucore中一个用户态进程的执行状态生命周期图（包执行状态，执行状态之间的变换关系，以及产生变换的事件或函数调用）。（字符方式画即可）

1. 如何影响执行状态
    - fork 创建一个子进程，将子进程状态从UNINIT变为RUNABLE，但不改变父进程状态
    - exec 不影响当前进程的执行状态，但是会修改当前进程中执行的程序
    - wait 取决于是否存在可以释放资源（ZOMBIE）的子进程，如果存在则不会发生状态的改变，如果不存在则会将当前进程置为SLEEPING态，等待执行了exit的子进程将其唤醒
    - exit 将当前进程的状态修改为ZOMBIE态，并将父进程唤醒（修改为RUNNABLE），最后主动让出CPU使用权
2. 一个用户态进程的执行状态生命周期图



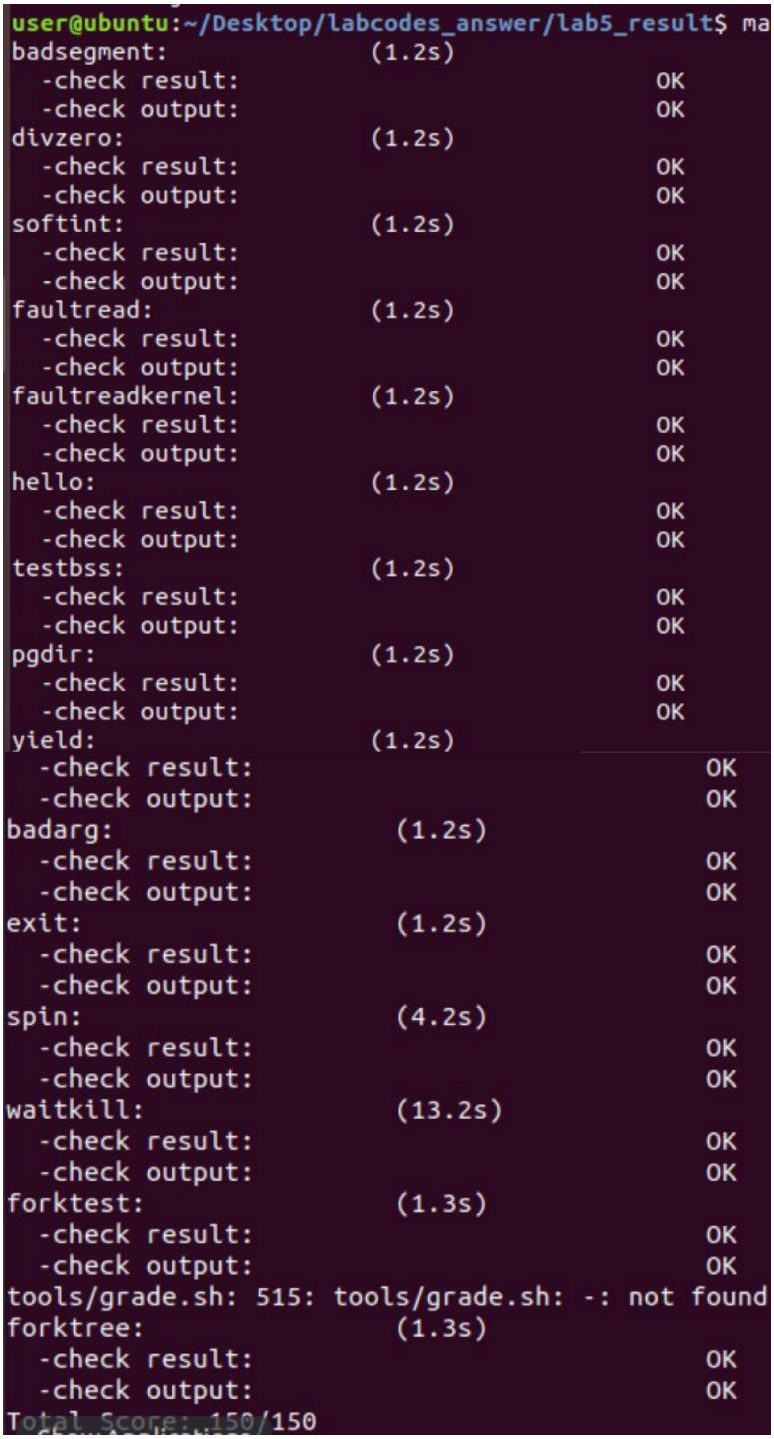
这个是一个自己理解的画的丑陋版本，画完才想起来一开始看的proc.c里好像有。。。

```
process state changing:

  alloc_proc                                 RUNNING
      +                                  +--<----<--+
      +                                  + proc_run +
      V                                  +-->---->--+
PROC_UNINIT -- proc_init/wakeup_proc --> PROC_RUNNABLE -- try_free_pages/do_wait/do_sleep --> PROC_SLEEPING --
                                           A      +                                                           +
                                           |      +--- do_exit --> PROC_ZOMBIE                                +
                                           +                                                                  +
                                           ----------------------------wakeup_proc--------------------------------
```

运行结果

```
++ setup timer interrupts
kernel_execve: pid = 2, name = "exit".
I am the parent. Forking the child...
I am parent, fork a child pid 3
I am the parent, waiting now..
I am the child.
waitpid 3 ok.
exit pass.
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:454:
    initproc exit.
```



Challenge

按照上述想法

- 首先在vmm.c中将dup_mmap中的share变量的值改为1，表示启用共享

  `bool share=1;`

- 然后在 `copy_range` 里面对共享进行处理，不进行练习二中完成的复制和建立映射部分，而是将标记为共享的页面设置为不可写，因为不是同一个mm所以两个虚拟地址都要进行设置

```
int
copy_range(pde_t *to, pde_t *from, uintptr_t start, uintptr_t end, bool share)
    assert(start % PGSIZE == 0 && end % PGSIZE == 0);
    assert(USER_ACCESS(start, end));
    // copy content by page unit.
    do {
        //call get_pte to find process A's pte according to the addr start
        pte_t *ptep = get_pte(from, start, 0), *nptep;
        if (ptep == NULL) {
            start = ROUNDDOWN(start + PTSIZE, PTSIZE);
            continue ;
        }
        //call get_pte to find process B's pte according to the addr start. If
        if (*ptep & PTE_P) {
            if ((nptep = get_pte(to, start, 1)) == NULL) {
                return -E_NO_MEM;
            }
        uint32_t perm = (*ptep & PTE_USER);
        //get page from ptep
        struct Page *page = pte2page(*ptep);
        // alloc a page for process B
        assert(page!=NULL);
        int ret=0;
        if (share) {   //NEW
                page_insert(from, page, start, perm & (~PTE_W));  //修改为不可写
                ret = page_insert(to, page, start, perm & (~PTE_W));
            }
        else {
                // alloc a page for process B
                struct Page *npage=alloc_page();
                assert(npage!=NULL);
        /* LAB5:EXERCISE2 YOUR CODE
```

- 然后是vmm.c里面对do_pgfault()进行修改，也就是上一个lab的不报错情况(W/R=1, P=1): write, present

  这一状态说明进程访问到了共享页面，内核需要对这一pagefault处重新分配页面、拷贝页面内容

```
//check the error_code
switch (error_code & 3) {
default:
    struct Page *page = pte2page(*ptep);
    struct Page *npage = pgdir_alloc_page(mm->pgdir, addr, perm);//分配一个新物理页
    uintptr_t src_kvaddr = page2kva(page);//父进程虚拟地址
    uintptr_t dst_kvaddr = page2kva(npage);//子进程虚拟地址
    memcpy(dst_kvaddr, src_kvaddr, PGSIZE); //拷贝
    break;
```