Lab5:用户程序

姓名：杨普超　学号：2210751

姓名：刘忠旺　学号：2213306

练习0：填写已有实验

本实验依赖实验2/3/4。请把你做的实验2/3/4的代码填入本实验中代码中
有“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。你需要补充 `load_icode` 的第6步，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。

- 请简要描述这个用户态进程被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。

执行实现

代码如下：

```
1 struct trapframe *tf = current->tf;
2     // Keep sstatus
3     uintptr_t sstatus = tf->status;
4     memset(tf, 0, sizeof(struct trapframe));
5     /* LAB5:EXERCISE1 2210751
6      * should set tf->gpr.sp, tf->epc, tf->status
7      * NOTICE: If we set trapframe correctly, then the user level process can
  return to USER MODE from kernel. So
8      *          tf->gpr.sp should be user stack top (the value of sp)
9      *          tf->epc should be entry point of user program (the value of
  sepc)
```

```
10      *          tf->status should be appropriate for user program (the value
    of sstatus)
11      *          hint: check meaning of SPP, SPIE in SSTATUS, use them by
    SSTATUS_SPP, SSTATUS_SPIE(defined in risv.h)
12      */
13     tf->gpr.sp = USTACKTOP;// 设置用户进程的栈指针为用户栈的顶部.当进程从内核态切换到
    用户态时，栈指针需要指向用户栈的有效地址
14     tf->epc = elf->e_entry; //修改epc,切换为程序入口地址，sret返回地址发生变化
15     // 进程从内核态切换到用户态，需要将中断帧的状态调整为用户态，清除了 SPP 表示的特权级
    信息，以及 SPIE 表示的中断使能信息。
16     tf->status = sstatus & ~(SSTATUS_SPP | SSTATUS_SPIE);// 将 sstatus 寄存器中
    的 SPP和 SPIE位清零
```

该部分实际是构造一个中断返回环境，以便中断处理完成后能够切换到需要执行的程序入口

执行经过

- 用户态进程被ucroe选择占用CPU后，用户态进程调用 `exec` 系统调用，进入正常中断处理流程，控制权最终转移到 `sycall.c` 的 `sycall` ，根据IDT传递给 `sys_exec` 函数，调用 `proc.c` 的 `do_execve` 函数完成进程的加载
- `do_execve` 函数负责回收进程自身所占用的空间，换用 `kernel` 的 `PDT` ，之后调用 `load_icode` 函数，用新的程序覆盖内存空间，形成一个执行新程序的新进程，同时设置好当前系统调用的中断帧，使得中断返回后能够以用户态权限跳转到新的进程的入口处执行
- 中断返回处理， `epc` 指向程序内存入口，同时SPP清零，因此 `trapentry.S` 中的处理部分就会将堆栈切换回用户进程的栈同时完成特权级的切换，跳转到程序入口，开始执行第一条指令

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你的设计实现过程。

- 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（简称COW）的基本概念是指如果有多个使用者对一个资源A（比如内存块）进行读操作，则每个使用者只需获得一个指向同一个资源A的指针，就可以该资源了。若某使用者需要对这个资源A进行写操作，系统会对该资源进行拷贝操作，从而使得该“写操作”使用者获得一个该资源A的“私有”拷贝—资源B，可对资源B进行写操作。该“写操作”使用者对资源B的改变对于其他的使用者而言是不可见的，因为其他使用者看到的还是资源A。

copy_range函数

`copy_range` 函数的主要作用是将一个进程 A 的地址空间的内存内容从地址 start 到 end 复制到另一个进程 B 的地址空间的功能，主要有以下步骤：

1. **检查对齐和有效范围**：确保 `start` 和 `end` 地址都是页对齐的，并且处于用户空间的有效地址范围内。
2. **遍历每一页**：根据 `start` 和 `end` 的范围，按页遍历父进程的地址空间。
3. **查找父进程的页表项**：通过 `get_pte` 函数获取父进程中对应虚拟地址 `start` 的页表项（PTE）。
4. **查找子进程的页表项**：如果父进程的页表项有效（即 PTE 中的有效位 `PTE_V` 被设置），则查找子进程相同虚拟地址的页表项。如果子进程的页表项不存在，则分配一个新的页表项。
5. **复制内存内容**：找到父进程页面后，分配一个新的页面给子进程，并将父进程页面的内容复制到子进程的页面中。
6. **建立映射关系**：使用 `page_insert` 函数将复制后的新页面映射到子进程的虚拟地址空间。

补全主要补全该函数的第5步和第6步，补全的程序代码如下：

```
if (*ptep & PTE_V) {
            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
            struct Page *npage = alloc_page();
            assert(page != NULL);
            assert(npage != NULL);
            int ret = 0;
            /* LAB5:EXERCISE2 2210751
             * replicate content of page to npage, build the map of phy addr of
             * nage with the linear addr start
             *
             * Some Useful MACROs and DEFINEs, you can use them in below
             * implementation.
             * MACROs or Functions:
             *    page2kva(struct Page *page): return the kernel vritual addr
of
             * memory which page managed (SEE pmm.h)
             *    page_insert: build the map of phy addr of an Page with the
             * linear addr la
             *    memcpy: typical memory copy function
             *
             * (1) find src_kvaddr: the kernel virtual address of page
```

```
27              * (2) find dst_kvaddr: the kernel virtual address of npage
28              * (3) memory copy from src_kvaddr to dst_kvaddr, size is PGSIZE
29              * (4) build the map of phy addr of  nage with the linear addr
    start
30              */
31             //(1) find src_kvaddr: the kernel virtual address of page
32             void* src_kvaddr = page2kva(page); // 源页的内核虚拟地址
33             //(2) find dst_kvaddr: the kernel virtual address of npage
34             void* dst_kvaddr = page2kva(npage); // 目标页的内核虚拟地址
35             //(3) memory copy from src_kvaddr to dst_kvaddr, size is PGSIZE
36             memcpy(dst_kvaddr, src_kvaddr, PGSIZE); // 复制页面内容
37             //(4) build the map of phy addr of  nage with the linear addr start
38             // 将目标进程 B 中的页表项和页结构体建立映射关系
39             ret = page_insert(to, npage, start, perm);
40             // 断言映射建立成功
41             assert(ret == 0);
42         }
```

COW设计

根据题干的讲述，我在此给出简要的一个设计

- `copy_range` 时，不对内存进行复制
- 在内存页出现访问异常的之后再对共享的内存页进行复制，再在新的内存页上进行修改操作。
- 具体实现：
  - 在 `copy_range` 部分的内存复制时，不对内存进行复制，而是将两个进程的内存页映射到同一个物理页，在各自的虚拟页上标记该页为不可写，同时设置一个额外的标记位为共享位 `share` ，表示该页和某些虚拟页共享了一个物理页，当发生修改异常时，进行对应的处理；
  - 在 `do_pgfault` 部分对是否是由于写共享页引起的异常增加一个判断，是的话再申请一个物理页来将共享页复制一份，交给出错的进程进行处理，将其原本映射关系改成新的物理页，设置该虚拟页为非共享、可写。

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）

请在实验报告中简要说明你对 fork/exec/wait/exit函数的分析。并回答如下问题：

- 请分析fork/exec/wait/exit的执行流程。重点关注哪些操作是在用户态完成，哪些是在内核态完成？内核态与用户态程序是如何交错执行的？内核态执行结果是如何返回给用户程序的？
- 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）

函数作用：

- fork：创建一个新的进程。用户进程通过调用 `fork` 启动系统调用，内核会为新进程分配资源，复制父进程的地址空间等。
- exec：用新程序替换当前进程的地址空间。用户进程调用 `exec` 后，内核会加载新程序，并跳转到新程序的入口。
- wait：等待子进程的结束。用户进程调用 `wait` 后，内核会挂起当前进程，直到子进程结束并返回状态。
- exit：进程退出。用户进程调用 `exit` 时，内核清理进程资源，并可能将进程标记为僵尸，等待父进程回收。

fork函数分析：

执行流程：

我们在user/libs/ulib.c中找到了fork()函数，该函数调用了sys_fork()函数作为返回结果；

我们在user/libs/syscall.c中找到了sys_fork()函数，该函数调用了syscall()函数作为返回结果；

我们在user/libs/syscall.c中找到了syscall()函数，该函数将ecall指令执行后的结果作为返回结果；

执行ecall后，产生trap, 进入内核态进行异常处理；

我们在kern/trap/trap.c中找到了exception_handler()函数，该函数调用了syscall()函数；

我们在kern/syscall/syscall.c中找到了syscall()函数，该函数通过syscalls[]调用了sys_fork()函数；

我们在kern/syscall/syscall.c中找到了sys_fork()函数，该函数通过调用了do_fork()函数作为返回结果；

我们在kern/process/proc.c中找到了do_fork()函数，该函数完成对新进程的进程控制块的初始化、内容的设置以及进程列表的更新，并将pid作为返回值。

分析：

流程中的前四行都是在用户态进行的，后面的都是在内核态进行的。

用户态通过ecall,产生trap，进入内核态。

内核态结束do_fork()函数后，返回子进程的pid，最后通过sret 指令返回用户态, 传递返回值。

exec函数分析：

执行流程：

我们在kern/process/proc.c中找到了user_main()函数，该函数调用了KERNEL_EXECVE()函数；

我们在kern/process/proc.c中找到了KERNEL_EXECVE()函数，该函数调用了kernel_execve()函数；

我们在kern/process/proc.c中找到了kernel_execve()函数，该函数执行ebreak指令；

执行ebreak后，产生trap, 进入内核态进行异常处理；

我们在kern/trap/trap.c中找到了exception_handler()函数，该函数调用了syscall()函数；

我们在kern/syscall/syscall.c中找到了syscall()函数，该函数通过syscalls[]调用了sys_exec()函数；

我们在kern/syscall/syscall.c中找到了sys_exec()函数，该函数通过调用了do_execve()函数作为返回结果；

我们在kern/process/proc.c中找到了do_execve()函数，该函数调用了load_icode()函数和set_proc_name()函数；do_execve函数负责回收进程自身所占用的空间，之后调用load_icode函数，用新的程序覆盖内存空间，形成一个执行新程序的新进程，同时设置好中断帧，使得中断返回后能够跳转到新的进程的入口处执行。

我们在kern/process/proc.c中找到了load_icode()函数，该函数加载新的可执行文件，创建新的内存管理结构 ，设置新的页表，解析 ELF 文件，加载程序段到内存中，建立新的内存映射，设置用户栈。

我们在kern/process/proc.c中找到了set_proc_name()函数，该函数设置进程名称。

分析

流程中的前四行都是在用户态进行的，后面的都是在内核态进行的。

用户态通过ecall,产生trap，进入内核态。

内核态结束do_execve()函数后，通过设置进程的中断帧 `tf` ，确保返回用户态时，从新程序的入口地址开始执行。

wait函数分析：

执行流程：

我们在user/libs/ulib.c中找到了wait()函数，该函数调用了sys_wait()函数作为返回结果；

我们在user/libs/syscall.c中找到了sys_wait()函数，该函数调用了syscall()函数作为返回结果；

我们在user/libs/syscall.c中找到了syscall()函数，该函数将ecall指令执行后的结果作为返回结果；

执行ecall后，产生trap, 进入内核态进行异常处理；

我们在kern/trap/trap.c中找到了exception_handler()函数，该函数调用了syscall()函数；

我们在kern/syscall/syscall.c中找到了syscall()函数，该函数通过syscalls[]调用了sys_wait()函数；

我们在kern/syscall/syscall.c中找到了sys_wait()函数，该函数通过调用了do_wait()函数作为返回结果；

我们在kern/process/proc.c中找到了do_wait()函数，该函数搜索指定进程是否存在指定的ZOMBIE态子进程（或任意的ZOMBIE态子进程），根据查找结果，找到则直接将其占用的所有剩余资源，如内核栈、进程控制块等全部释放；未找到则将进程状态设置为SLEEPING并设置等待状态为等待子进程、调用调度器切换到别的可执行进程，直至对应的子进程陷入ZOMBIE态唤醒这个父进程。

分析：

流程中的前四行都是在用户态进行的，后面的都是在内核态进行的。

用户态通过ecall,产生trap，进入内核态。

内核态调用 sys_wait, 传递子进程的 PID 和状态存储地址。

exit函数分析：

执行流程：

我们在user/libs/ulib.c中找到了exit()函数，该函数调用了sys_exit()函数作为返回结果；

我们在user/libs/syscall.c中找到了sys_exit()函数，该函数调用了syscall()函数作为返回结果；

我们在user/libs/syscall.c中找到了syscall()函数，该函数将ecall指令执行后的结果作为返回结果；

执行ecall后，产生trap, 进入内核态进行异常处理；

我们在kern/trap/trap.c中找到了exception_handler()函数，该函数调用了syscall()函数；

我们在kern/syscall/syscall.c中找到了syscall()函数，该函数通过syscalls[]调用了sys_exit()函数；

我们在kern/syscall/syscall.c中找到了sys_exit()函数，该函数通过调用了do_exit()函数作为返回结果；

我们在kern/process/proc.c中找到了do_exit()函数，该函数释放当前进程的大部分资源，更改该进程状态为ZOMBIE，并在父进程进入wait等待状态时唤醒，调用调度器换出其他进程，等待父进程进一步完成对剩余资源的回收。

分析：

流程中的前四行都是在用户态进行的，后面的都是在内核态进行的。

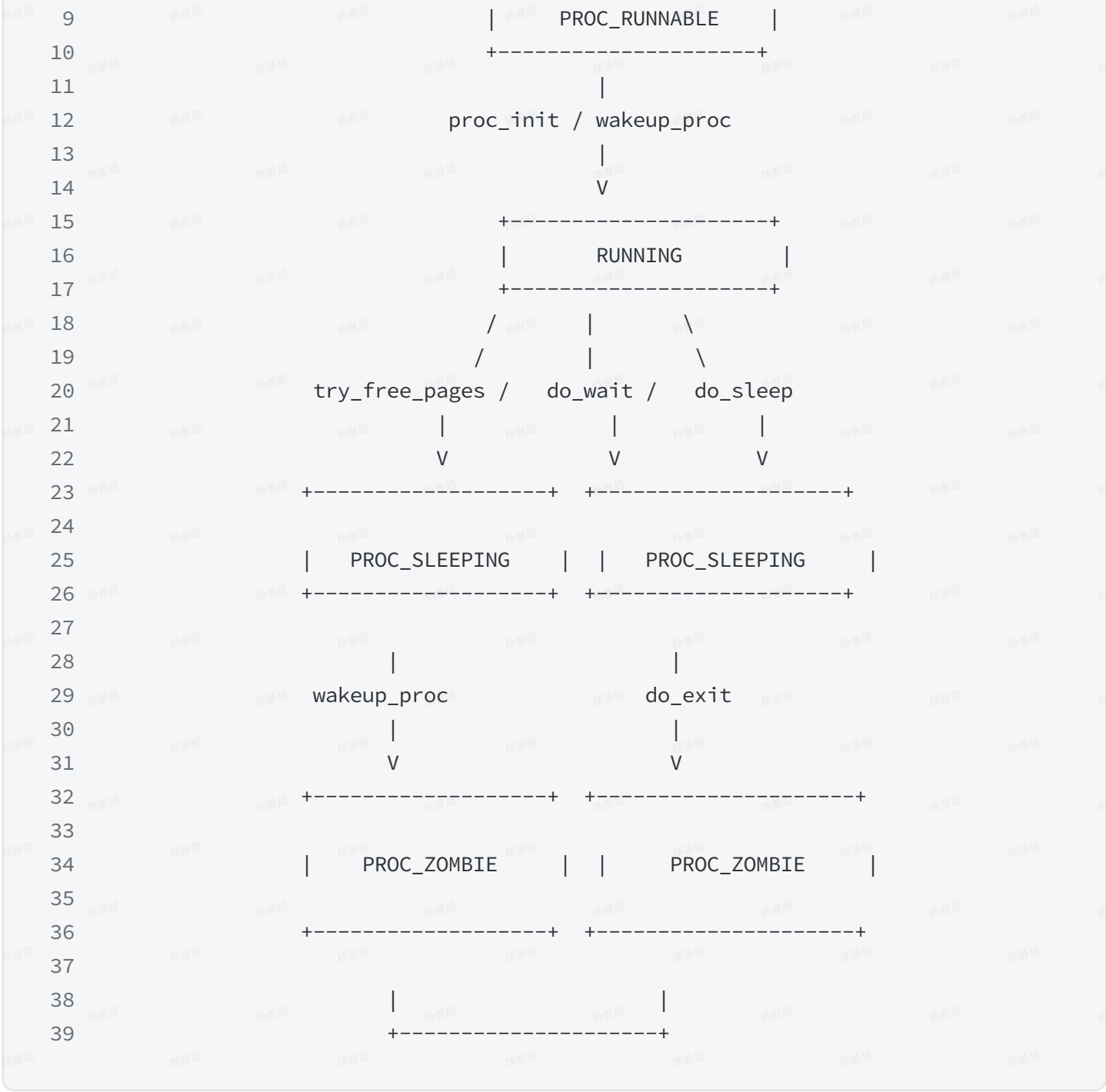
用户态通过ecall,产生trap，进入内核态。

内核态不再返回用户态，没有返回。

状态图：

以下是基于 `do_wait` 和 `do_exit` 函数的用户态进程执行状态生命周期图，包含了各个状态以及状态之间的转移关系，并标明了具体的函数调用：

```
                              +---------------------+
                              |      PROC_UNINIT    |
                              +---------------------+
                                        |
                               alloc_proc
                                        |
                                        V
                              +---------------------+
```



扩展练习 Challenge

1. 实现 Copy on Write （COW）机制
2. 给出实现源码,测试用例和设计报告（包括在cow情况下的各种状态转换（类似有限状态自动机）的说明）。
3. 这个扩展练习涉及到本实验和上一个实验“虚拟内存管理”。在ucore操作系统中，当一个用户父进程创建自己的子进程时，父进程会把其申请的用户空间设置为只读，子进程可共享父进程占用的用户内存空间中的页面（这就是一个共享的资源）。当其中任何一个进程修改此用户内存空间中的某页面时，ucore会通过page fault异常获知该操作，并完成拷贝内存页面，使得两个进程都有各自的内存页面。这样一个进程所做的修改不会被另外一个进程可见了。请在ucore中实现这样的COW机制。

4. 由于COW实现比较复杂，容易引入bug，请参考 https://dirtycow.ninja/ 看看能否在ucore的COW实现中模拟这个错误和解决方案。需要有解释。
5. 这是一个big challenge.
6. 说明该用户程序是何时被预先加载到内存中的？与我们常用操作系统的加载有何区别，原因是什么？

设计报告：

COW原理：

简而言之，COW就是读文件就直接读原件；当写文件时，再创建一个新的副本。

具体的细节有：写文件是要加锁的，同一时间只能有一个人在写文件。写完文件并不意味着原来的文件会立刻更新。读原来文件的用户读的依旧是原来的文件，但是新的用户会使用新的文件。

状态转化：

- 全都只读状态：多进程共享一个页面，该页面对所有进程都是只读。页面的状态标记为共享，进程可以读取该页面，但不能修改。
- 有写进程状态（只会存在一个）：当一个进程试图修改共享页面时，操作系统会复制页面并将其标记为独立状态。此时，修改操作只会影响当前进程，其他进程仍然可以共享原页面。
- 新旧交替状态：旧页面的访问者不在访问后，旧页面的引用数相应减少，直到为零后，直接删除旧页面。

核心代码：

- **创建子进程时的共享内存页面设置**：当我们需要共享资源的时候，由原来的父进程创建一个子进程，二者共享同一份资源。注意此时父进程应当设为只读模式，避免冲突。

```
int
dup_mmap(struct mm_struct *to, struct mm_struct *from) {
    assert(to != NULL && from != NULL);
    list_entry_t *list = &(from->mmap_list), *le = list;
    //遍历list
    while ((le = list_prev(le)) != list) {
        struct vma_struct *vma, *nvma;
        vma = le2vma(le, list_link);
        //创建新的vma
        nvma = vma_create(vma->vm_start, vma->vm_end, vma->vm_flags);
        if (nvma == NULL) {
            return -E_NO_MEM;
        }

        //插入vma
```

```
16          insert_vma_struct(to, nvma);
17
18          // 实现COW：设置共享标志，启用页面共享
19          bool share = 1;
20          //调用copy_range函数复制虚拟内存区域中的具体内容
21          if (copy_range(to->pgdir, from->pgdir, vma->vm_start, vma->vm_end,
    share) != 0) {
22              return -E_NO_MEM;
23          }
24      }
25      return 0;
26  }
```

- 同时，我们在 `pmm.c` 中的 `copy_range` 函数中，进行相应的补充share为1时的情况

```
1           //下面实现COW
2           int ret = 0;
3
4           //页面共享时,源页面映射到目标进程，设置为只读
5           if(share)
6           {
7               cprintf("Sharing the page 0x%x\n", page2kva(page));
8               page_insert(from, page, start, perm & (~PTE_W));
9               ret = page_insert(to, page, start, perm & (~PTE_W));
10          }
```

- **写时异常、新旧交换**：当子进程对数据进行修改时，会产生一个COW异常，此时我们创建一个新页，把原页的信息复制过来，增加写权限，将新页指向子进程，这样子进程便能够进行修改。这部分的实现我们在处理缺页异常的函数 `do_pgfault` 进行

```
1           //下面为实现COW机制，所进行修改，原有部分在下方else处
2           struct Page *page=NULL;
3
4           //COW机制
5           if (*ptep & PTE_V)
6           {
7               cprintf("\n\nCOW: ptep 0x%x, pte 0x%x\n",ptep, *ptep);//输出调试信
    息，页表项的地址和页表项的值
8               page = pte2page(*ptep);//获取对应物理页面
9
10              //检查该物理页面被引用的次数，如果被多个进程引用，需进行复制
11              if (page_ref(page) > 1)
12              {
```

```
13                //为当前进程分配一个物理页
14                struct Page *newPage = pgdir_alloc_page(mm->pgdir, addr, perm);
15
16                //获取源页和新页的地址
17                void *kva_src = page2kva(page);
18                void *kva_dst = page2kva(newPage);
19
20                //进行复制
21                memcpy(kva_dst, kva_src, PGSIZE); // 复制整个页面的内容
22            }
23            //引用次数为1,可以直接修改
24            else
25            {
26                //直接将该页面映射到当前进程的地址空间，并重新设置PTE权限为写权限
27                page_insert(mm->pgdir, page, addr, perm);
28            }
```

说明该用户程序是何时被预先加载到内存中的？与我们常用操作系统的加载有何区别，原因是什么？

在 uCore 操作系统中，用户程序是通过 `execve` 系统调用加载到内存中的。具体过程如下：

1. 加载时机

用户程序会在执行 `execve` 系统调用时被加载到内存中。`execve` 是一个由用户态发起的系统调用，通常由用户程序通过内联汇编或系统调用接口触发。以下是相关过程：

- 用户进程通过 `execve` 请求内核执行新的程序。
- 内核处理该请求时，会首先释放当前进程的资源（如虚拟内存空间、页目录等），然后将新的用户程序加载到当前进程的虚拟内存中。

在内核代码中，`kernel_execve` 会触发 `syscall`，最终调用 `do_execve` 函数来完成新的程序的加载。程序二进制文件（`binary`）和相关的信息（如程序的路径名 `name`）会作为参数传递给内核。内核通过 `load_icode` 等函数，将用户程序的二进制代码加载到当前进程的内存中。

2. 与常用操作系统加载的区别

常见的操作系统（如 Linux、Windows 等）在加载用户程序时，通常使用更为复杂的加载机制，包括：

- **内核加载程序**：操作系统内核会负责在用户请求时将程序从磁盘加载到内存。这通常涉及到读取 ELF 或 PE 格式的可执行文件，解析程序的头部信息，分配合适的内存空间，将程序的代码段、数据段等加载到进程的虚拟内存空间。

- **程序分配内存**：操作系统会分配合适的虚拟内存空间给用户程序，支持动态链接库（DLL）或共享库（shared libraries）的加载，并且可能使用分页机制来按需加载程序的不同部分（如 Linux 的 `demand paging`）。
- **原因**：uCore 是一个简化的教学操作系统，还未实现文件系统功能和磁盘管理，所以只能采用这种方式模拟用户程序的加载。