Lab5 用户进程

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。

小组成员：张高2213219 张铭2211289 黄贝杰2210873

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**

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。

改进1：alloc_proc函数

修改alloc_proc函数，设置wait_state和cptr、optr、yptr的值

```
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
    //LAB4:EXERCISE1:2210873 黄贝杰
    /*
     * 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; // 此时未分配该PCB对应的资源，故状态为初始态
        proc->pid = -1; // 与state对应，表示无法运行
        proc->runs = 0; // 分配阶段故运行次数为0
        proc->kstack = 0; // 内核栈暂未分配
        proc->need_resched = 0; // 不调度其他进程、即CPU资源不分配
        proc->parent = NULL; // 当前无父进程
        proc->mm = NULL; // 当前未分配内存
        memset(&(proc->context), 0, sizeof(struct context)); // 上下文置零
        proc->tf = NULL; // 当前无中断帧
        proc->cr3 = boot_cr3; // 内核线程同属于一个内核大进程，共享内核空间，故页表
相同
        proc->flags = 0; // 当前暂无
        memset(&(proc->name), 0, PROC_NAME_LEN); // 当前暂无
        // LAB5新增
        // 初始化进程等待状态、和进程的相关指针，例如父进程、子进程、同胞等等。
        // wait_state是LAB5新增的字段，避免之后由于未定义或未初始化导致管理用户进程时
出现错误。
        proc->wait_state = 0; // 初始化进程为等待状态(wait_state = 0)
        proc->cptr = proc->optr = proc->yptr = NULL; // 新proc相关的proc
    }
    return proc;
}
```

改进2：do_fork函数

更改do_fork函数，确保进程处于等待状态并调用set_links函数， set_links函数用于更新进程链表并建立父子兄弟进程之间的指针关系（cptr，optr，yptr）

```
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {

    //...前面其他代码

    //    1. call alloc_proc to allocate a proc_struct 分配并初始化进程控制块
（alloc_proc函数）
    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }
    proc->parent = current;

    // LAB5新增
    assert(current->wait_state == 0); //确保进程在等待

    //    2. call setup_kstack to allocate a kernel stack for child process
分配并初始化内核栈（setup_stack函数）
    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }
```

```
    //    3. call copy_mm to dup OR share mm according clone_flag 根据
clone_flags决定是复制还是共享内存管理系统（copy_mm函数）
    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_proc;
    }
    //    4. call copy_thread to setup tf & context in proc_struct 设置进程
的中断帧和上下文（copy_thread函数）
    copy_thread(proc, stack, tf);
    //    5. insert proc_struct into hash_list && proc_list 把设置好的进程加入
链表
    int intr_flag;
    local_intr_save(intr_flag);
    {
        proc->pid = get_pid();
        // list_add(&proc_list, &proc->list_link);
        // nr_process++;

        // LAB5新增
        set_links(proc); // 设置进程链接
        hash_proc(proc);

    }
    //...后续其他代码
}
```

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。

  ```
  tf->gpr.sp=USTACKTOP;

  tf->status=sstatus&(~(SSTATUS_SPP|SSTATUS_SPIE));

  tf->epc=elf->e_entry;
  ```

  1. 将 tf 结构中的通用寄存器（gpr）中的栈指针（sp）设置为 USTACKTOP。这表明将用户栈的顶部地址赋给用户进程的栈指针。
  2. 将 tf 结构中的状态寄存器（status）设置为给定的 sstatus，但清除了 SPP（Supervisor PreviousPrivilege）和 SPIE（Supervisor Previous Interrupt Enable）标志。这两个标志通常用于处理从内核返回用户模式时的特权级别和中断使能状态。
  3. 将 tf 结构中的程序计数器（epc）设置为 ELF 文件的入口点地址。这是用户程序的启动地址，将控制权转移到用户程序的执行起点。
  - 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。

调用schedule函数，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过程，将控制权转移到了 syscall.c 中的 syscall 函数，然后根据系统调用号转移给了 sys_exec 函数，在该函数中调用了 do_execve 函数来完成指定应用程序的加载。

而do_execve 函数：

1. **参数检查**：首先，函数检查传入的程序名称指针 `name` 是否指向有效的用户空间内存。`user_mem_check` 函数用于验证内存地址和长度是否有效。
2. **限制程序名称长度**：如果程序名称的长度 `len` 超过了 `PROC_NAME_LEN`（进程名称的最大长度），则将其截断为最大长度。
3. **复制程序名称**：将程序名称从用户空间复制到内核空间的 `local_name` 数组中，以便后续使用。
4. **回收旧内存空间**：
   - 如果当前进程 `current` 有关联的内存管理结构 `mm`，则需要回收其内存空间。
   - 首先，将页表基址寄存器 `CR3` 设置为启动时的页表 `boot_cr3`，以确保在回收内存时使用内核的页表。
   - 然后，减少 `mm` 的引用计数。如果引用计数降至0，说明没有其他进程共享这个内存管理结构，可以将其销毁。
   - 调用 `exit_mmap` 函数来卸载进程的页表映射。
   - 调用 `put_pgdir` 函数释放页目录页。
   - 调用 `mm_destroy` 函数销毁内存管理结构。
   - 最后，将 `current->mm` 设置为 `NULL`，表示当前进程没有关联的内存管理结构。
5. **加载新程序**：调用 `load_icode` 函数加载新的程序到当前进程的地址空间。这个函数会创建新的页表，并将程序的代码段、数据段和BSS段加载到内存中。
6. **设置进程名称**：如果 `load_icode` 函数成功，将新程序的名称设置为 `local_name`。
7. **错误处理**：如果 `load_icode` 函数返回错误，`do_execve` 函数会调用 `do_exit` 函数退出当前进程，并打印错误信息。
8. **返回**：如果一切顺利，`do_execve` 函数返回0，表示成功执行了新程序。

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_range`实现思路

`copy_range`函数负责将父进程的内存直接复制给子进程，其调用过程如下： **do_fork()-->copy_mm()-->dup_mmap()-->copy_range()** `do_fork`函数用于创建一个进程，并放入CPU中调度，该函数中调用了`copy_mm`函数 `copy_mm`函数进行内存空间的复制，该函数中调用了`dup_mmap`函数 `dup_mmap`函数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中去

`copy_range`的实现思路如下： - **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** - **为子进程当前分配这一物理页映射上对应的在子进程虚拟地址空间里的一个虚拟页**

copy_range函数代码实现

```
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
        // pte is NULL, just alloc a PT
        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:EXERCISE2：2211289 张铭
             * 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
             * (2) find dst_kvaddr: the kernel virtual address of npage
             * (3) memory copy from src_kvaddr to dst_kvaddr, size is
PGSIZE
             * (4) build the map of phy addr of  nage with the linear addr
start
             */
```

```
            //(1)获取父进程的内存页对应的内核虚拟地址，page是父进程的内存页，
page2kva函数将页结构转换为对应的内核虚拟地址
            void * kva_src = page2kva(page);
            //(2)获取子进程对应的内核虚拟页地址，npage是新分配给子进程的内存页
            void * kva_dst = page2kva(npage);
            //(3)将父进程的内存页内容复制到子进程的内存页
            memcpy(kva_dst, kva_src, PGSIZE);
            //(4)将子进程的内存页插入到子进程的页表中。to是子进程的内存管理结构，
npage是子进程的内存页，start是虚拟地址，perm是页的权限。
            ret = page_insert(to, npage, start, perm);

            assert(ret == 0);
        }
        start += PGSIZE;
    } while (start != 0 && start < end);
    return 0;
}
```

Copy on Write机制设计思路

设计核心

- **当多个进程需要读取同一个资源时，它们初始时共享同一份拷贝**
- **当其中一个进程尝试修改这个共享资源时，系统才会创建一个新的副本**

设计思路

- **设置共享内存标记** 引入一个标记位R/W Bit,来标识某块内存是否可以共享 在do_fork时不进行内存复制，只将对应内存页的页目录项中的R/W Bit设为只读
- **page fault** 一旦发生写操作，就会引发page fault 在中断处理程序中恢复内存页的R/W状态 进行程序代码段、数据段的复制 返回继续执行

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？

  答：

  1.fork：

  执行完毕后，如果创建新进程成功，则出现两个进程，一个是子进程，一个是父进程。在子进程中，fork函数返回0，在父进程中，fork返回新创建子进程的进程ID。我们可以通过fork返回的值来判断当前进程是子进程还是父进程.

  顺序：sys_fork->do_fork

  do_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。

- 用户态程序调用sys_fork()系统调用，通过syscall进入内核态。
- 内核态处理sys_fork()系统调用，调用do_fork()函数创建子进程，完成后返回到用户态。

2.exit:

回收当前进程所占的大部分内存资源，并通知父进程完成最后的回收工作。

顺序：sys_exit->exit

do_exit():

1: 检查特殊进程

- **关键点**：确保 `idleproc`（空闲进程）和 `initproc`（初始化进程）不会执行退出操作。这两个进程是系统运行的关键，它们的退出会导致系统不稳定。

2: 处理内存管理结构

- 关键点：如果当前进程有关联的内存管理结构

  ```
  mm
  ```

  则需要释放与该进程相关的所有内存资源。

  - **释放内存映射**：`exit_mmap(mm)` 释放进程的内存映射。
  - **释放页目录**：`put_pgdir(mm)` 释放页目录。
  - **销毁内存管理结构**：`mm_destroy(mm)` 销毁 `mm` 结构本身。
  - **设置为空**：`current->mm = NULL` 将当前进程的 `mm` 指针设置为 NULL，表示没有内存管理结构与之关联。

3: 设置进程状态

- **关键点**：将当前进程的状态设置为 `PROC_ZOMBIE`（僵尸状态），表示进程已经结束但尚未被其父进程回收。

4: 保存和恢复退出代码

- **关键点**：保存传入的 `error_code` 作为进程的退出代码，以便父进程可以检索。

5: 处理中断和父进程

- 关键点：使用

  ```
  local_intr_save
  ```

  和

  ```
  local_intr_restore
  ```

  来保护临界区，防止中断干扰进程状态的变更。

  - **唤醒父进程**：如果父进程处于等待状态（`WT_CHILD`），则通过 `wakeup_proc(proc)` 唤醒父进程，以便它可以回收僵尸进程。

6: 重新分配子进程

- 关键点：将当前进程的所有子进程重新分配给

  ```
  initproc（初始化进程），确保这些子进程不会被孤儿化。
  ```

  - **更新子进程链表**：遍历当前进程的子进程链表，更新它们的父进程指针，并将其添加到 `initproc` 的子进程链表中。

7: 调度其他进程

- **关键点**：调用 `schedule()` 函数，将控制权交给调度器，以便选择另一个进程运行。

8: 确保不返回

- **关键点**：由于 `do_exit` 函数不应该返回，因此在函数末尾调用 `panic` 函数，确保如果代码执行到这里，系统会报错并停止执行。

进入以及退出内核态的情况与fork同理

3.execve:

完成用户进程的创建工作。首先为加载新的执行码做好用户态内存空间清空准备。接下来的一步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。

顺序：sys_exec->do_execve

do_execve():

1、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，且

进一步判断mm的引用计数减1后是否为0，如果为0，则表明没有进程再需要此进程所占用的内存空间，为此将

根据mm中的记录，释放进程所占用户空间内存和进程页表本身所占空间。最后把当前进程的mm内存管理指针为

空。

2、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load_icode从而使

之准备好执行。

4.wait:

等待任意子进程的结束通知。

顺序：sys_wait->do_wait

do_wait():

1. 根据 PID 查找子进程

- 关键点

  ：函数首先检查传入的

  ```
  pid
  ```

  参数。

  - 如果 `pid` 不为0，表示父进程只关心特定 ID 的子进程。函数会调用 `find_proc(pid)` 来查找这个特定的子进程。
  - 如果 `pid` 为0，表示父进程希望等待任意一个处于僵尸状态的子进程。

2. 等待子进程变为僵尸状态

- 关键点

  ：如果找到的子进程状态不是

  ```
  PROC_ZOMBIE
  ```

  ，说明子进程尚未退出。

  - **设置睡眠状态**：当前进程（父进程）会将自己的状态设置为 `PROC_SLEEPING`，并将等待原因设置为 `WT_CHILD`。
  - **调度其他进程**：调用 `schedule()` 函数，使当前进程让出 CPU，调度器会选择其他进程执行。

- **重复检查**：如果父进程被唤醒（可能是因为其他子进程退出变为僵尸状态），它会重复检查是否有子进程变为僵尸状态。

3. 回收僵尸子进程资源

- 关键点

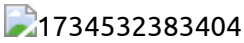
  ：一旦找到处于

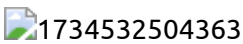
  ```
  PROC_ZOMBIE
  ```

  状态的子进程，父进程将执行以下操作来回收资源：

  - **检查特殊进程**：确保不是在等待 `idleproc` 或 `initproc`，这两个进程不应该被回收。
  - **存储退出代码**：如果 `code_store` 不为空，将子进程的退出代码存储到提供的地址。
  - **保护临界区**：使用 `local_intr_save` 和 `local_intr_restore` 来保护临界区，防止中断干扰进程状态的变更。
  - **删除进程控制块**：从 `proc_list` 和 `hash_list` 中删除子进程的控制块。
  - **释放内核堆栈**：调用 `put_kstack(proc)` 释放子进程的内核堆栈。
  - **释放进程控制块**：调用 `kfree(proc)` 释放子进程的进程控制块。

- 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）





扩展练习1 Challenge

实现 Copy on Write （COW）机制

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）。

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。

设计思路

1. 原代码没有实现 COW 机制，在父进程创建子进程时，调用 `fork` 函数，直接复制父进程的内存映射到子进程
2. 函数调用的总体流程为：`do_fork`-->`copy_mm`-->`dup_mmap`-->`copy_range`
3. 但为了提高效率，尝试在上述函数框架中引入 COW 机制。父子进程初始时共享同一块内存区域，只有在某一方修改内存时才会真正复制一份新的内存。
4. 需要引入`do_pgfault`函数添加页面写保护处理，并在此时复制页面，实现写时复制的功能。

实验源码:

1. do_fork函数：

- 创建新进程结构
- 对非idleproc启用 COW 机制
- 设置share_mem标志和CLONE_VM标志来实现内存共享

```
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    ...
    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }
    proc->parent = current;
    assert(current->wait_state == 0); //确保进程在等待

    // idleproc 在创建 initproc 时不需要 COW 机制，保证 COW 机制用于普通进程
    bool need_cow = (current != idleproc && current->mm != NULL);

    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }

    // 如果需要利用 COW 机制，则设置相关参数
    if (need_cow) {
        proc->share_mem = 1;  // 新增：标记此进程共享内存
        clone_flags |= CLONE_VM;  // 新增：确保共享虚拟内存
    }

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

2. copy_mm函数

- 为子进程创建内存管理结构
- 复制父进程的页目录
- 调用dup_mmap处理内存映射

```
static int
copy_mm(uint32_t clone_flags, struct proc_struct *proc, bool need_cow) {
    struct mm_struct *mm, *oldmm = current->mm;

    if (oldmm == NULL) {
        return 0;
    }

    // 如果当前父进程是 idleproc，则不适用 COW 机制，单独处理
    if (!need_cow) {
```

```
        mm = oldmm;
        goto good_mm;
    }

    int ret = -E_NO_MEM;
    if ((mm = mm_create()) == NULL) {
        goto bad_mm;
    }

    if (setup_pgdir(mm) != 0) {
        goto bad_pgdir_cleanup_mm;
    }

    lock_mm(oldmm);
    {
        // 普通进程的创建，总是执行dup_mmap，处理父子进程页面共享
        ret = dup_mmap(mm, oldmm);
    }
    unlock_mm(oldmm);

    if (ret != 0) {
        goto bad_dup_cleanup_mmap;
    }

good_mm:
    mm_count_inc(mm);
    proc->mm = mm;
    proc->cr3 = PADDR(mm->pgdir);
    return 0;
bad_dup_cleanup_mmap:
    exit_mmap(mm);
    put_pgdir(mm);
bad_pgdir_cleanup_mm:
    mm_destroy(mm);
bad_mm:
    return ret;
}
```

3. dup_mmap函数

- 复制虚拟内存区域 (VMA) 结构
- 设置 share 标志为 1 实现 COW 机制
- 调用copy_range实现页面共享或者复制

```
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;
    while ((le = list_prev(le)) != list) {
        struct vma_struct *vma, *nvma;
        vma = le2vma(le, list_link);
```

```
        nvma = vma_create(vma->vm_start, vma->vm_end, vma->vm_flags);
        if (nvma == NULL) {
            return -E_NO_MEM;
        }

        insert_vma_struct(to, nvma);

        bool share = 1;  // 将 share 参数设置为1，确保父子进程页面共享

        if (copy_range(to->pgdir, from->pgdir, vma->vm_start, vma->vm_end,
share) != 0) {
            return -E_NO_MEM;
        }
    }
    return 0;
}
```

4. copy_range函数

- 遍历需要复制的地址空间，查找父进程的页表项
- 判断是否共享或复制内存
- 如果 share 为 1，则采用 COW 机制，子进程和父进程共享该内存页，直到其中一个进程写入该页时才会复制一份
- 如果 share 为 0，则直接复制父进程的内存页内容到子进程的内存

```
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));
    do {
        pte_t *ptep = get_pte(from, start, 0), *nptep;
        if (ptep == NULL) {
            start = ROUNDDOWN(start + PTSIZE, PTSIZE);
            continue;
        }
        if (*ptep & PTE_V) {
            if ((nptep = get_pte(to, start, 1)) == NULL) {
                return -E_NO_MEM;
            }
            struct Page *page = pte2page(*ptep);
            assert(page != NULL);
            int ret = 0;

            if (share) {
                // 当 share 参数为 1 时，使用 COW 机制
                uint32_t perm = ((*ptep & PTE_USER) & ~PTE_W) | PTE_V; //
更新父进程的用户访问权限，强制页面变为只读
                // 将该页面和权限信息插入到父和子进程进程的页表中
                ret = page_insert(from, page, start, perm) &
page_insert(to, page, start, perm);
            } else {
```

```
                // 当 share 参数为 0 时，复制父进程的内存页到子进程
                uint32_t perm = (*ptep & PTE_USER);
                struct Page *npage = alloc_page();
                assert(npage != NULL);
                void * kva_src = page2kva(page);
                void * kva_dst = page2kva(npage);
                memcpy(kva_dst, kva_src, PGSIZE);
                ret = page_insert(to, npage, start, perm);
                assert(ret == 0);
            }
        }
        start += PGSIZE;
    } while (start != 0 && start < end);
    return 0;
}
```

5. do_pgfault函数

- 在进程发生页错误时进行必要的处理
- 页面分配，写时复制
- 页错误的原因是写入一个只读页面，并且该页表项没有 PTE_W 标志，则表示发生了写时复制的异常
- 只有当前进程引用该页面，则可以启用 PTE_W，允许写入
- 有多个进程共享该页面，则需要复制该页面，创建一个新的物理页面，并将内容从父页面复制到新的页面

```
int
do_pgfault(struct mm_struct *mm, uint_t error_code, uintptr_t addr) {
    ...
    // 添加页面写保护处理
    if (*ptep & PTE_V) {
        // 检查是否是 写时复制（COW）产生的异常
        if ((error_code & 2) && !(*ptep & PTE_W)) {  // 对只读页面尝试进行写入
            struct Page *page = pte2page(*ptep);

            // 页面引用数只有 1，直接实现写使能
            if (page_ref(page) == 1) {
                *ptep |= PTE_W;
                ret = 0;
                goto failed;
            }

            // 否则，需要复制页面
            struct Page *npage = pgdir_alloc_page(mm->pgdir, addr, perm);
            if (npage == NULL) {
                ret = -E_NO_MEM;
                goto failed;
            }
            void *kva_src = page2kva(page);
            void *kva_dst = page2kva(npage);
            memcpy(kva_dst, kva_src, PGSIZE);
            ret = page_insert(mm->pgdir, npage, addr, perm | PTE_W); // 将
```

```
新复制的页面和修改后的权限（加入写权限）更新到页表中
            goto failed;
        }
    }
    ...
failed:
    return ret;
}
```

在 COW 情况下的各种状态转换：

1. 状态定义：

- 共享状态（Shared）：页面由多个进程共享。父进程和子进程在创建时会共享相同的物理页面，且这些页面是只读的（PTE_W 被清除）。
- 私有状态（Private）：页面已经被某个进程修改，因此该页面被复制，成为该进程的私有页面。页面表项更新，指向新的物理页面，且该页面设置为可写（PTE_W）。
- 未映射状态（Unmapped）：页面尚未被映射或已被回收。这通常发生在某个页面被撤销映射或进程结束时。

2. 状态转换：

- 从共享状态（Shared）到私有状态（Private）
  - 触发条件：进程尝试写入共享页面，触发 COW 页面缺页异常。
  - 发生行为：操作系统为进程分配新的页面，并复制原始页面的内容。
- 从共享状态（Shared）到未映射状态（Unmapped）
  - 触发条件：进程不再使用该页面，页面被回收，引用计数为 0。
  - 发生行为：页面从页表中移除，物理页面被回收。
- 从未映射状态（Unmapped）到共享状态（Shared）或私有状态（Private）
  - 触发条件：进程访问未映射的页面，发生页面故障。
  - 发生行为：操作系统为该页面分配新的物理页面，并更新进程的页表。
- 从私有状态（Private）到私有状态（Private）
  - 触发条件：进程修改已经拥有的私有页面。
  - 发生行为：进程继续修改该页面，无需新的复制操作。

扩展练习2 Challenge

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？

何时被预先加载到内存

在ucore项目编译的时候载入内存的，在宏定义KERNEL_EXECVE我们可以发现用户态的程序载入其实是通过特定的编译输出文件， 此次实验更改了Makefile，并且通过ld指令将用户态程序（user文件夹下的代码）编译链接到项目中。

与常用操作系统的加载的区别

常用操作系统中，应用程序会在需要运行时才会被加载到内存中