扩展练习 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。
>
> 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。

一、思路（练习2中已提到）

Copy on Write 是指在复制一个对象的时候并不是真正的把原先的对象复制到内存的另外一个位置上，而是在新对象的内存映射表中设置一个指针，指向源对象的位置，并把那块内存的Copy-On-Write位设置为1。

那么，在我们的代码中，如何具体实现COW机制呢？分为如下两个核心问题：

- **如何检查是否需要COW机制，并设置共享空间处理呢?**
- **如何保障这个共享空间只允许读呢?**
- **如何在进行写操作的时候，进行私有的资源拷贝呢?**

实际上，对于问题一，我们的代码中，do_fork函数拥有标记位CLONE_VM，即标记该进程的虚拟内存可以在多个进程之间共享。

而在我们的kern_thread函数调用中，生成的内核进程实际上是共享整个内存虚拟空间的，因此也需要对这个标记位置1，即采用如下所示调用：

```
do_fork(clone_flags | CLONE_VM, 0, &tf);
```

而在copy_mm的最一开始，我们也对该标记位进行了判断：

```
if (clone_flags & CLONE_VM) { //可以共享地址空间
    mm = oldmm;  //共享地址空间
    goto good_mm;
}
……
good_mm:
    mm_count_inc(mm);  //共享地址空间的进程数加一
    proc->mm = mm;     //复制空间地址
    proc->cr3 = PADDR(mm->pgdir); //复制页表地址
    return 0;
```

这也就是为什么我们在lab4的内核进程构建中，不需要处理这部分的代码实现，因为实际上内核线程就是共享read和write，共同享有整个内核内存空间。所以可以采用类似的处理即可，我们的代码中也给出了标记位share的实现。

但是，我们的COW机制要求，共享read是可行的，共享write显然是不合适的，因此，还需要进一步考虑问题二和问题三，尝试对其实现：

对于问题二和问题三，实际上，我们可以巧妙地利用trap异常处理来实现：

- 当进行内存访问时，CPU会根据PTE上的**读写位PTE_P、PTE_W**来确定当前内存操作是否允许，如果不允许，则缺页中断，**我们可以在缺页中断处理程序中复制该页内存，并设置该页内存所对应的`PTE_W`为1。**
- 因此，我们可以在copy_range函数中，将父进程中**所有PTE中的PTE_W置为0**，这样便可以将父进程中所有空间都设置为只读。然后使子进程的PTE全部指向父进程中PTE存放的物理地址，这样便可以达到内存共享的目的，又可以保障只读和写时复制，完美完成我们的要求。

综上，我们基于这种设计思想即可完成我们的处理，接下来，我们给出具体的实现和一些处理。

二、代码实现

基于这个思想，我们开始对代码的实现：

首先，我们将vmm.c中的dup_mmap函数中队share变量的设置进行修改，因为dup_mmap函数中会调用copy_range函数，其中，share变量为传入的标记参数，这里修改share为1标志着启动了共享机制，我们可以在copy_range中进行处理。

这样处理的话，我们就会**照常生成新的虚拟内存管理结构，也就达成了只共享物理空间，虚拟空间仍然不一致的目标**

```
int dup_mmap(struct mm_struct *to, struct mm_struct *from) {
    ...
       bool share = 1; // 初始化共享标志为 false
        // 调用 copy_range 函数，将源进程的数据拷贝到目标进程
        if (copy_range(to->pgdir, from->pgdir, vma->vm_start, vma->vm_end, share)
!= 0) {
            return -E_NO_MEM; // 如果拷贝失败，返回错误
        }
   ...
}
```

随后，我们来看看在copy_range函数内部该如何处理。如果share为1，那么我们将子进程的页面映射到父进程的页面即可。

由于两个进程共享一个页面之后，无论任何一个进程修改页面，都会影响另外一个页面，所以需要子进程和父进程对于这个共享页面都保持只读。

```
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);
    assert(page != NULL);
    int ret = 0;
    //判断是否COW
    if (share) {
        // 完成页面分享
        cprintf("Sharing the page 0x%x\n", page2kva(page));
```

```
            page_insert(from, page, start, (perm & (~PTE_W))|PTE_R);
            page_insert(to, page, start, perm & (~PTE_W)|PTE_R);
        } else {
            //如果不分享的话 就正常分配页面
            struct Page *npage=alloc_page();
            assert(npage != NULL);
            ...
        }
    }
```

完成到这里的话，我们已经实现了读的共享，但是并没有对写做处理，因此需要对由于写了只能读的页面导致的页错误进行处理：当程序尝试修改只读的内存页面的时候，将触发Page Fault中断。

这时候我们可以检测出是超出权限访问导致的中断，说明进程访问了共享的页面且要进行修改，因此内核此时需要重新为进程分配页面、拷贝页面内容、建立映射关系。

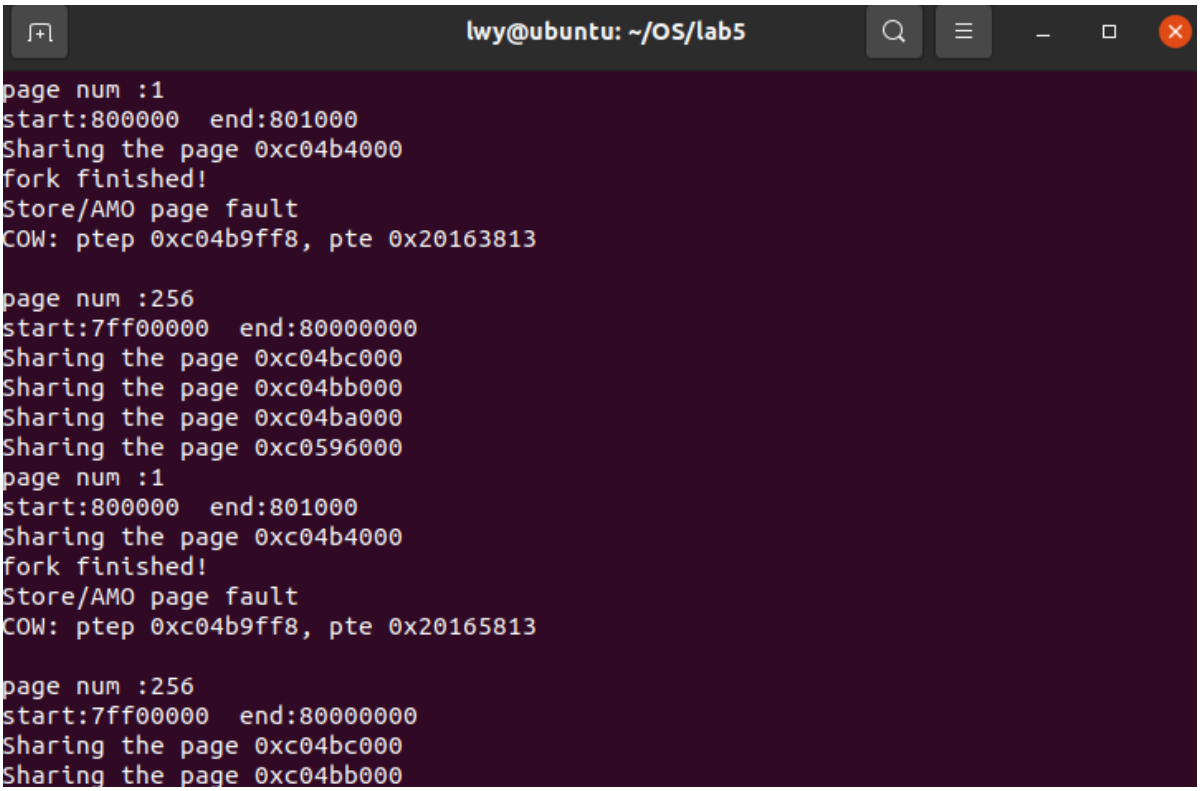
我们在先前几次实验中的do_pgfault函数中给出这个处理：

```
 // 如果当前页有效
if (*ptep & PTE_V) {
    //如果是由于COW原因产生
    if(error_code & 0b111 == 0b111){
        // 写时复制：复制一块内存给当前进程
        cprintf("COW: ptep 0x%x, pte 0x%x\n\n",ptep, *ptep);
        // 原先所使用的只读物理页
        page = pte2page(*ptep);
        // 如果该物理页面被多个进程引用
        if(page_ref(page) > 1)
        {
            // 释放当前PTE的引用并分配一个新物理页
            struct Page* newPage = pgdir_alloc_page(mm->pgdir, addr, perm);
            void * kva_src = page2kva(page);
            void * kva_dst = page2kva(newPage);
            // 拷贝数据
            memcpy(kva_dst, kva_src, PGSIZE);
        }
        // 如果该物理页面只被当前进程所引用,即page_ref等1
        else
            // 则可以直接执行page_insert，保留当前物理页并重设其权限。
            page_insert(mm->pgdir, page, addr, perm);
    }
}
```

三、测试程序及结果

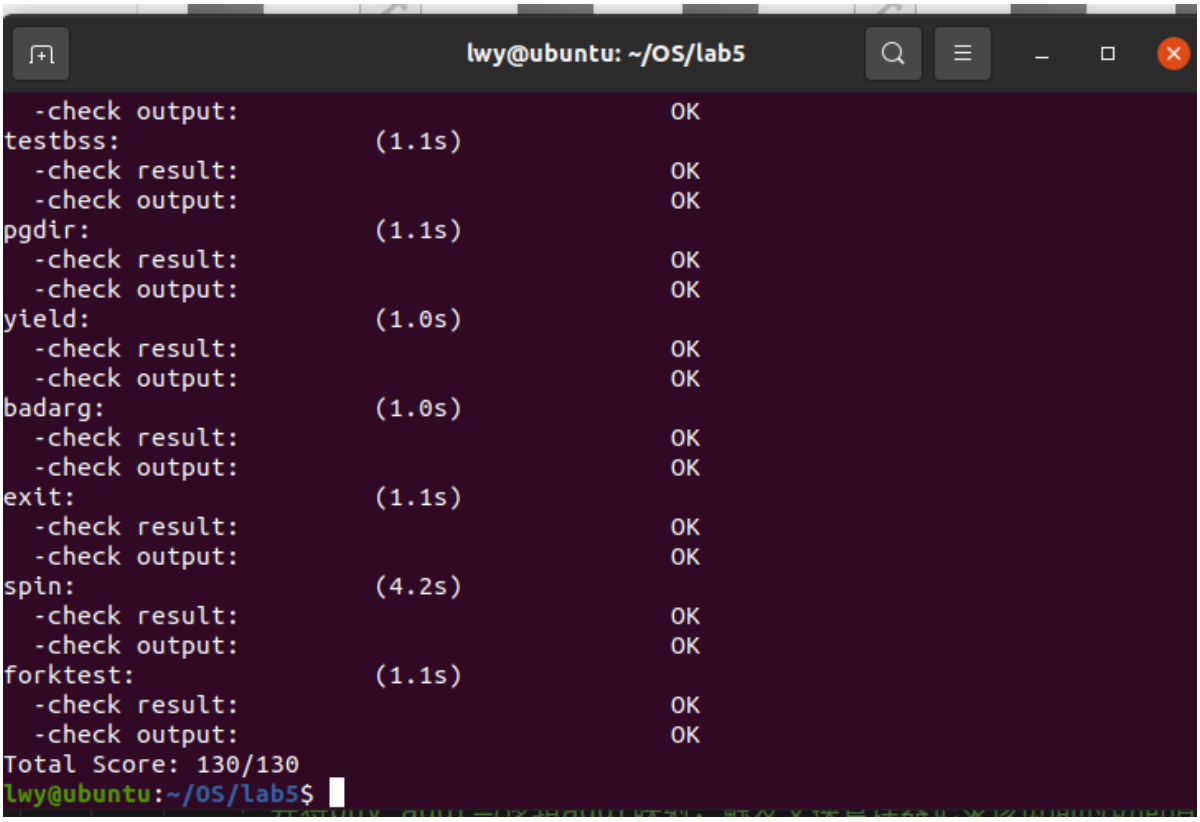
本次实验中，实际上我们无需设计专用的测试用例，结合我们grade.sh中提供的测试脚本，和特定的用户程序forktest.cpp即可实现测试：

我们首先make qemu，来看看forktest.cpp的执行是否正确输出了COW的提示语句：

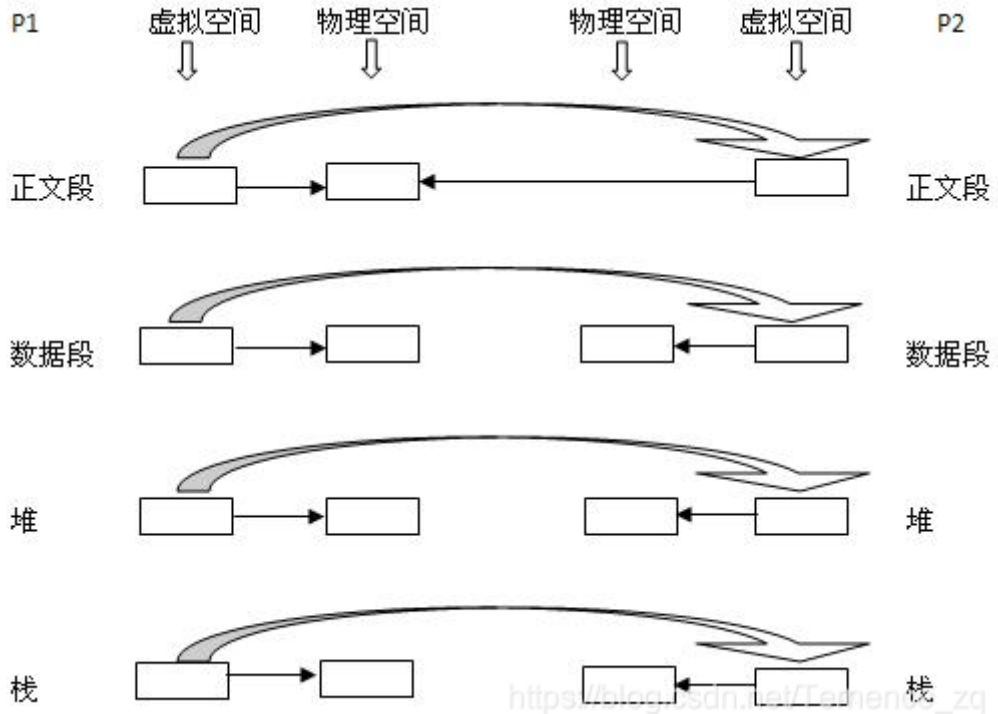


在最开始的时候，该程序执行父线程，发现父线程输出了共享物理页的信息，表明启用了COW机制，而随后发生了写入页异常，也就是store/AMD page fault，这里，发生了写时复制，也就是整个COW机制都有所体现。

此外，仅仅有提示信息还是不够的，还需要确保能够正常运行，我们执行make grade，发现能够正常运行所有程序，效果和非COW机制相同，表明我们的COW机制正确实现了。



四、COW状态转移图



由于课业压力较大，这次实验的状态转移就不仔细手动绘制流程图了，我给出如上的参考图例，并结合语言描述形容一下：

COW和fork函数是分不开的，我们结合fork函数的过程来讲讲：

1. 首先，父进程执行fork函数，复制出子线程，子线程的物理内存空间与父线程完全一致，但虚拟内存空间不相同，父进程和子进程的物理空间都是只读的。
2. 当子进程被写入的时候，触发写时复制，将原先的物理内存空间复制一份，赋给子进程。
3. 当进程结束后，如果页面只有它一个引用，需要释放写时复制的空间，否则只需要释放引用。