仓库链接： https://github.com/zhaoyihui233/mit-6.S081-lab-2021/tree/main/lab5%20Copy-onWrite%20fork%20for%20xv6

# 1 Implement copy-on write

## 1.1实验内容

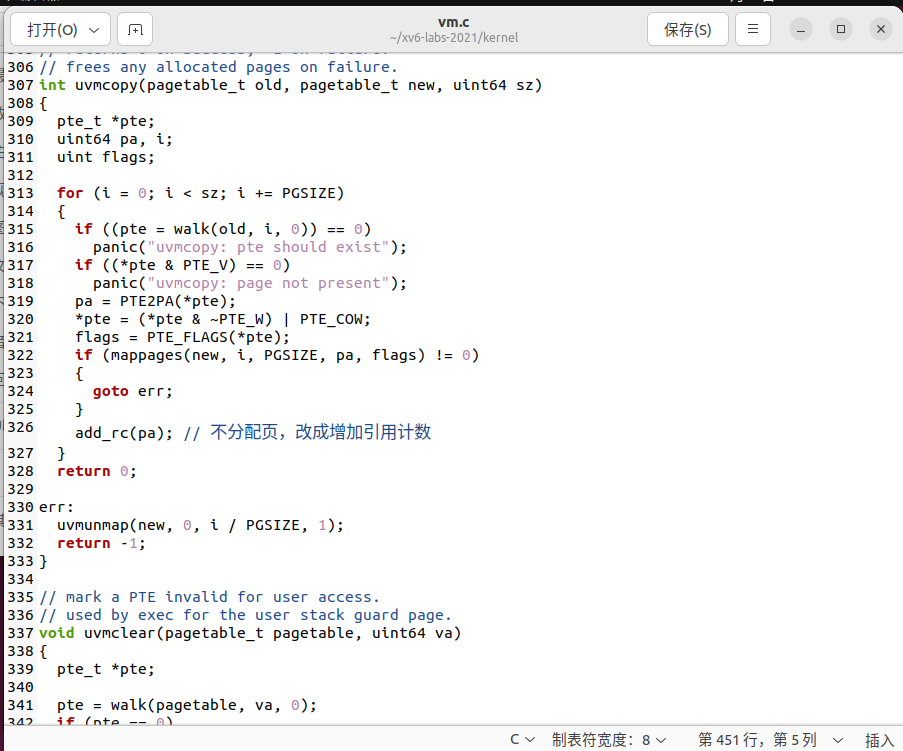
xv6中的fork()系统调用将父进程的所有用户空间内存复制到子进程中。如果父进程较大，则复制可能需要很长时间。更糟糕的是，这项工作经常造成大量浪费；例如，子进程中的fork()后跟exec()将导致子进程丢弃复制的内存，而其中的大部分可能都从未使用过。另一方面，如果父子进程都使用一个页面，并且其中一个或两个对该页面有写操作，则确实需要复制。

在本次实验中，我们需要实现在fork系统调用上实现写时复制机制（COW）。COW的fork()只为子进程创建一个页表，用户内存的PTE指向父进程的物理页。COW fork()将父进程和子进程中的所有用户PTE标记为不可写。当任一进程试图写入其中一个COW页时，CPU将强制产生页面错误。内核页面错误处理程序检测到这种情况将为出错进程分配一页物理内存，将原始页复制到新页中，并修改出错进程中的相关PTE指向新的页面，将PTE标记为可写。当页面错误处理程序返回时，用户进程将能够写入其页面副本。

引入COW机制会使释放用户内存的物理页面变得更加复杂。因为引入该机制后一个物理页可能会被多个进程的页表引用，所以只有在最后一个引用消失时该物理页才应该被释放。用来检验我们所编写的代码的cowtest.c程序，其中就有让一个占用了超过一半内存空间的进程执行fork系统调用的测试，如果是原fork，这个操作是不会成功的，但如果采用了COW，则能成功。

## 1.2代码位置与截图

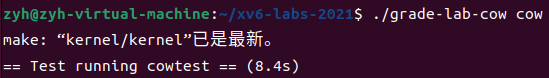




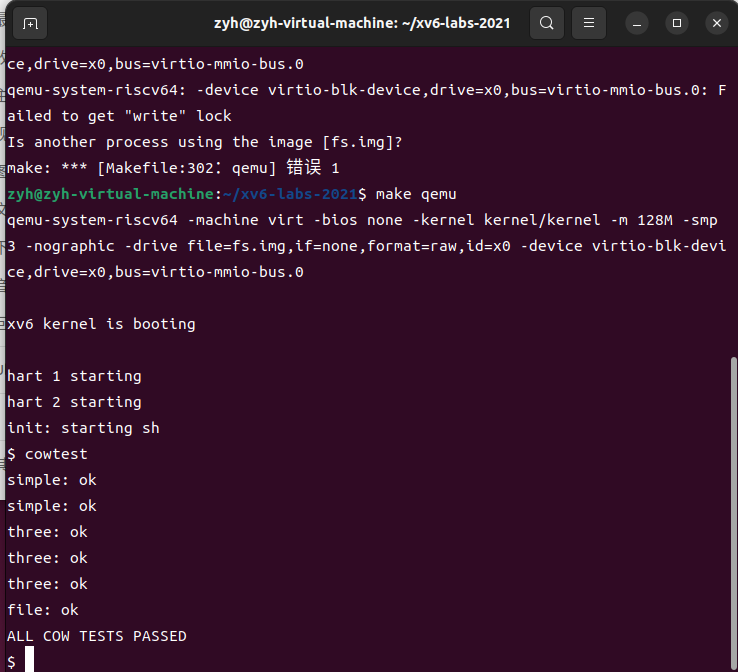


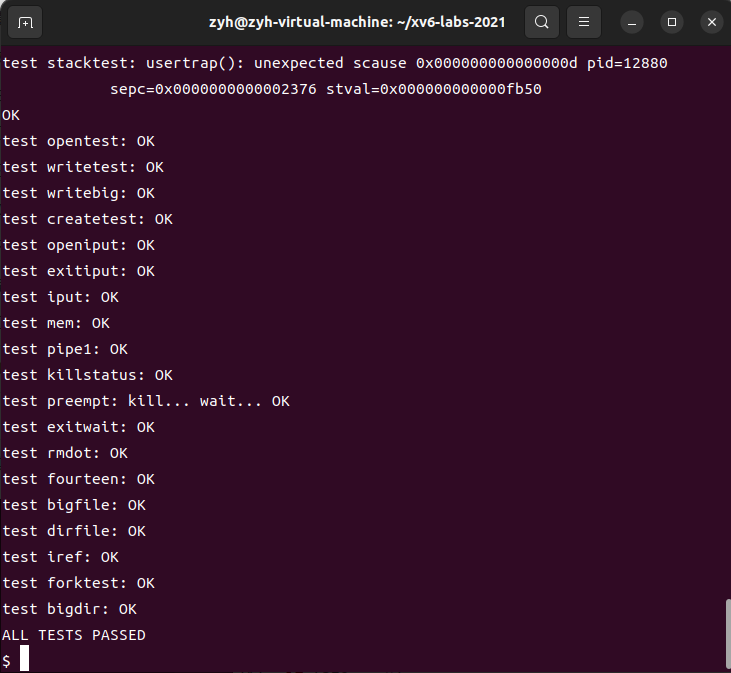
## 1.3实验结果

./grade-lab-cow cow

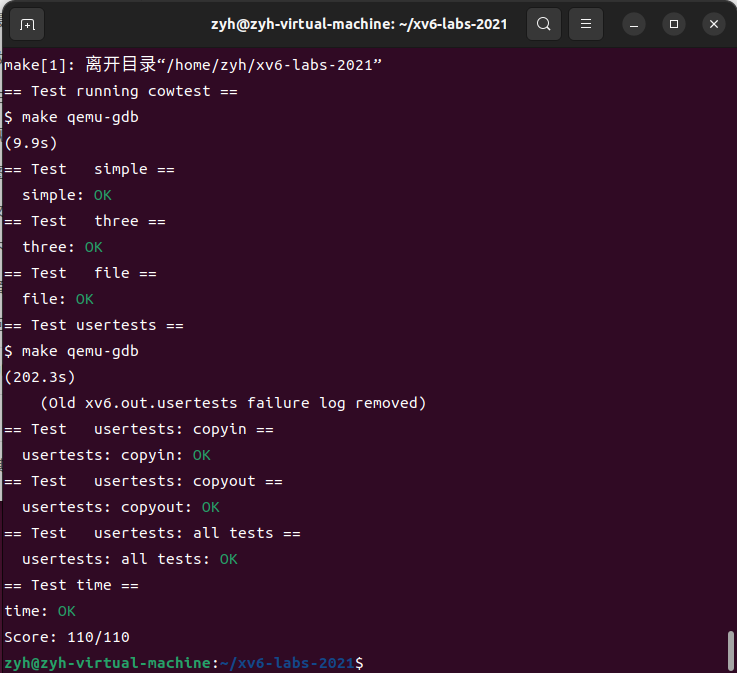


make grade





make grade



## 1.4代码实现

### 1.4.1实现思路

首先，我们应该要修改创建子进程后代码，因为原fork系统调用就是在创建子进程后，把父进程的各种内容都直接复制给子进程。根据提示，我们可以将子进程的物理页映射到父进程，并取消物理页的写权限。这样如果父子进程只是对这些内容进行读，就不需要另外开辟空间复制这些数据。这时，对于一个物理地址，会有两个虚拟地址映射于其上。

为实现子进程的物理页映射到父进程，并取消进程对其的写权限，我们要修改vmcopy函数，它是一个被fork调用的函数，功能是把父进程的页表复制给子进程。该函数位于kernel/vm.c里。主要做出三个方面的改动：去除分配新页表的代码；修改虚拟地址与物理地址的映射关系；取消页表写权限。代码mappages(new, i, PGSIZE, (uint64)mem, flags)是在给新的虚拟地址(new)与新的物理页地址(mem)直接建立映射关系，因为我们要让父子进程的虚拟地址映射到同一个物理地址上，因此上面代码中的mem修改为pa，即旧的物理页地址。

PTE\_COW是我们自定义的标志位，定义在第8位（因为8、9位都是保留位），用于表示一个页表是否被用于实现COW，这样我们也好做页表的释放逻辑。

接下来，就是应对父子进程写的情况。因为上面我们取消了父子进程对于相应物理页的写权限，所以父子进程在尝试写时会触发Page Fault。于是我们还要修改处理Page Fault的相关代码，我们要使系统能识别出引发Page Fault的是否是COW机制，如果是，还要能将要写的原内容进行拷贝，具体来说是分配一个新页表，该进程对页表有写权限，然后原内容要复制到其上，且进程原来的虚拟地址映射到这个新页表上。

为此，我们要修改usertrap函数（位于kernel/vm.c文件里），修改其中的应对Page Fault的代码。应该添加对于scause为13或15时，这表示当前是Page Fault中断。相应的处理代码就写在这其中。我们需要创建新的页表，并把要执行写操作的原页表内容复制给新页表，映射到原页表的虚拟地址需要映射到新页表。该新页表可写。

最后，我们还要解决物理页的释放问题。对于父子进程都没有尝试去写的物理页，父子进程的虚拟地址都是映射到该物理页上的，只有父子进程都释放后，该物理页才能真正释放。所以我们还要修改进程释放的相关代码，修改其中的释放物理页部分的代码。根据提示，我们可以使用“引用计数”的方法来应对这个问题。我创建了一个特别大的数组（static int reference\_count[(PHYSTOP - KERNBASE) / PGSIZE + 1];），数组长度(PHYSTOP - KERNBASE) / PGSIZE +1，其中PHYSTOP表示内存的上限地址，KERNBASE表示内核空间的基地址，PGSIZE表示页面的大小，所以计算得到页表的总数加一。每一个元素对应一个物理页表，其数字表示该页表被引用的次数。所谓引用就是映射到该物理地址的虚拟地址的个数。只有一个页表的引用次数为1，当执行释放操作时才能真正释放该页表的空间。

当引入“引用计数”的概念后，kfree和kalloc函数都需要做修改（都在kernel/kalloc.c文件里）。kalloc用于申请页表空间，我们需要给这些页表对应的引用记为1。kfree用于释放页表空间，我们稍加修改，如果页表的引用次数大于1，则不释放空间，引用次数加一；如果页表的引用次数为1，才真正释放物理空间。

回看COW机制，它有效地节省了时间空间，尤其是大大优化了sh的性能。因为我们在sh执行指令时，sh进程的操作就是先后执行fork、exec系统调用，但执行exec后子进程会抛弃掉父进程复制过来的数据，也就是说，可能花费了一些时间与空间才复制过来的数据其实绝大部分是无用的。而COW就避免了这个问题，所以说COW优化了sh的执行过程。