实验五

Copy-on-Write Fork

目的：在程序之中，当父进程拥有非常大的内存使用空间的时候，调用fork会耗费相当长的时间，并且会造成很大的资源浪费。但大多数情况之下，父进程和子进程都不会更改进程的页，这样的话，他们就可以分享相同的物理内存。更特别地，当子进程调用exec()的时候，整个页都会被替换掉。

因此，更改fork使得只有父子至少一方需要写入时才复制原有的数据。值得注意的是，fork的页表仍然是新页表，但是页表映射到物理地址还是一样的。

**具体操作：**

关键问题：如果事实上usertrap解决不了所有的CoW问题呢？如果进程对页表进行释放、添加等操作会怎样。比如：sbrk调用growproc，再会调用uvmdealloc，再会调用unmap，unmap在CoW模式下还是会操作对应的内存。那么即使是没有写入，也会使得父子进程有所出入。因此内存可以共享，但是页表不能共享。那么CoW位就不能在根页表上了，就必须在每一个页表项。接下来，我们来叙述对应的设计。

设置不可写：fork调用后，将PTE设置不可以写入，当对相关PTE进行写入的时候，造成page fault。

更改page fault处理：分配一页物理内存，并且将原来的内存拷贝进去，并且修改相关的PTE以指向新的page，这个时候还要设置他们都可以进行写入。

释放物理页：注意，共享物理页的时候，只有当父子进程都释放了该空间才能够真正的释放。那如何知道是否真正释放了呢？采用页表存储？不现实，其是否真正释放是根据物理内存来决定的，那么就要对每个物理页设立一个标志位，让它能指示物理页是否被共享。

步骤：

1. 修改uvmcopy：原程序为复制页表以及对应数据，现在仅仅将子进程复制同一页表，同时删去写权限，并标志共享。



但是proc\_pagetable是一定有的，在init时需要调用，不只是fork需要，这一部分不需要关心，不在我们的CoW考虑范围之内。

加一个新的uvmcopycow函数：从根节点copy新的page table。步骤：如果原来的不是共享内存，直接copy，如果是，还要注意去掉共享标志，并且还原write位。

1. 修改usertrap：当发现page fault是因为COW（可以用unused bit以及PTE\_U看出），就分配新的page，并且拷贝老的page，设置两方都可以写。

http://www.five-embeddev.com/riscv-isa-manual/latest/supervisor.html#sec:scause

该网站说明了RISC-V的trap类型，其中scause==15的时候表示Store/AMO page fault，也就是Store异常，CoW的异常就是因为写无效才导致的异常，因此trap中判定为15，并且发现pagetable是处于CoW模式就可以判定对应page fault是CoW造成的。

uvmcopycow，之后若不为共享内存，还原write位。

加一个新的uvm\_deshare函数：减少分享位一次，表示有一个进程复制了新的页表，如果当前只剩最后一个页表，那么还原write位。

页表项使用两个附加位：CoW位和PTE\_WHIDE位，分别表示该页表项物理地址已被共享，该页表项原来的PTE\_W位。当发生CoW page fault的时候，不重新为整个页表分配内存，而是采用CoW，仅仅对产生page fault的页表项分配空间。

1. 修改 freepage的相关操作：使得释放page的时候只有当父子进程都释放了该空间才能够真正的释放。

修改proc.c中proc\_freepagetable函数，只有共享位为0才执行原本的操作，否则只是减少对应的共享位。

修改kalloc相关操作：额外添加对应物理页的共享标志位，每一页分配一个标志位，表示当前物理页被共享的次数。每次fork和trap的时候分别加一和减一对应的块。注意，因为需要额外增加对应的标志位，那么就要额外的使用对应的空间，需要在kalloc初始化留出一部分空间，并清零。

1. 修改copyout：操作和COW page faults一样。

Copyout是复制一部分数据到对应页表的虚拟地址所在的物理地址区域，本质上就是一次写操作，因此也要copynewvm，其他和步骤二类似。

技巧：

Riscv.h 的定义可能有帮助。