Lab: Copy-on-Write Fork for xv6

Git链接：[Jeery1/xv6-6.S081 at cow](https://github.com/Jeery1/xv6-6.S081/tree/cow)

**实验目的：**

理解传统 fork() 的局限性

问题背景：传统fork() 会立即复制父进程的全部物理内存给子进程，但实际场景中：子进程可能很快调用 exec() 替换为新程序（复制的内存被浪费）。父子进程可能仅需共享读访问（无需立即复制）。

后果：不必要的内存拷贝导致性能下降，尤其在内存密集型应用中。

2. 实现 COW 机制

核心思想：

共享物理页：fork() 时，子进程与父进程共享所有物理页，仅复制页表。

写时复制：当任一进程尝试修改共享页时，触发页错误，内核再动态分配新物理页并复制数据。

技术要点：

修改 uvmcopy()：将父进程的物理页映射到子进程，并标记为只读（PTE\_W 清零）。

在页错误处理逻辑（usertrap） 中捕获写保护异常（r\_scause() == 15），分配新页并复制数据。

维护引用计数：跟踪每个物理页的被引用次数，当计数归零时释放。

3. 解决 COW 的关键问题

引用计数：

为每个物理页添加计数器，避免过早释放仍被共享的页。

初始化：fork() 时递增父进程所有物理页的计数。

释放：kfree() 仅在计数为 0 时回收内存。

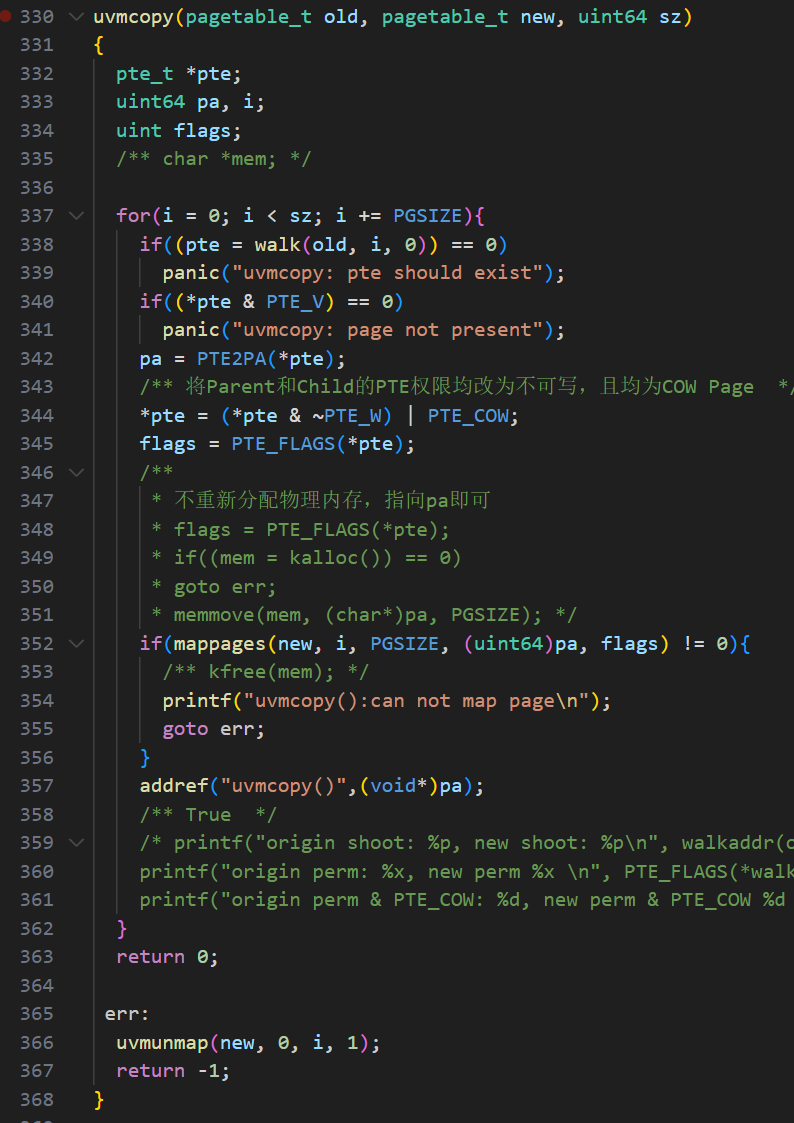
页错误处理：

区分 COW 触发的写保护错误与其他页错误（如非法访问）。

性能优化：

减少原子操作（如引用计数的增减）的开销。

**实验步骤：**

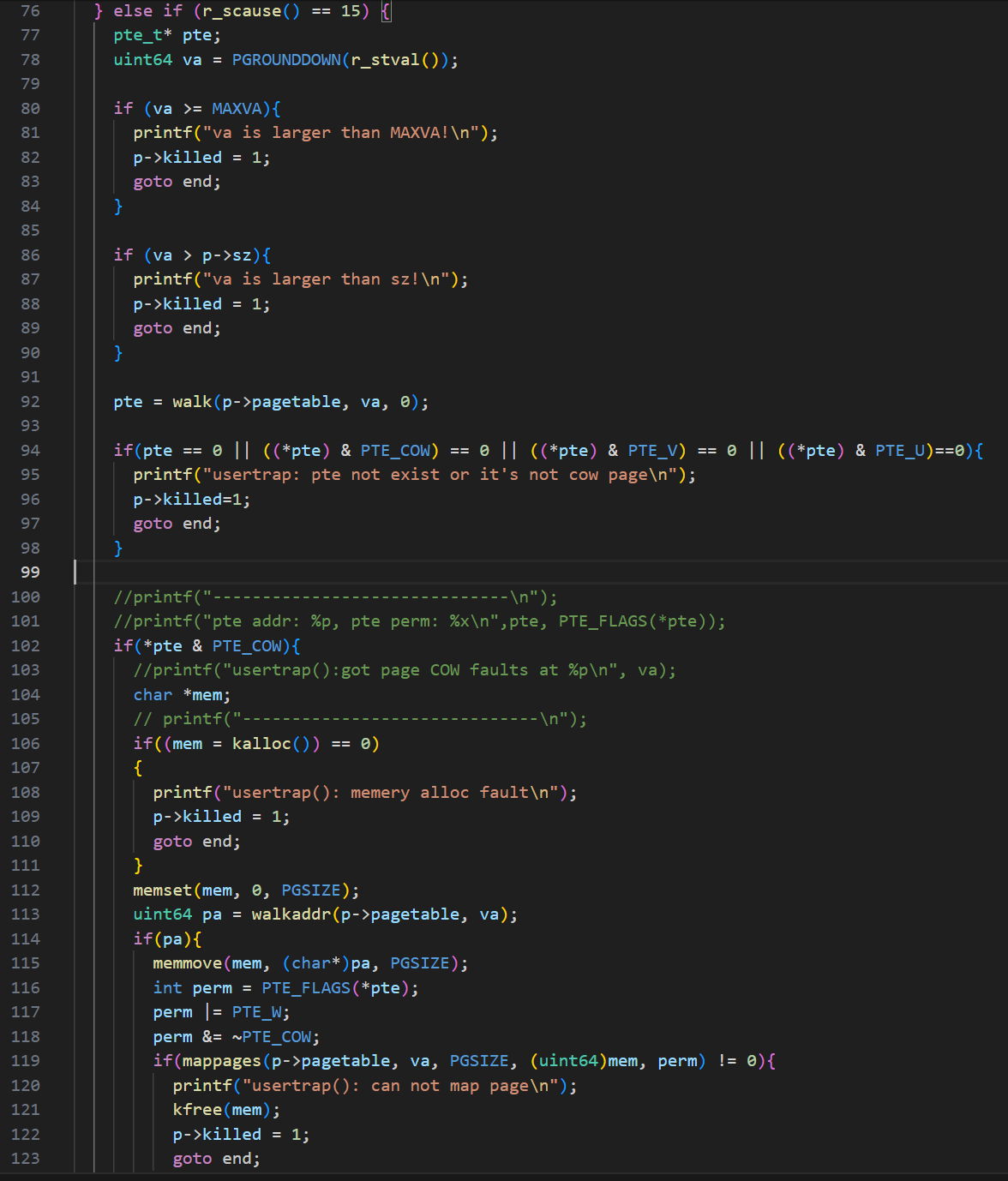
****

我们先更改fork中调用的uvmcopy函数，使其只复制页表而不复制物理页面，并且修改子进程和父进程的pte为不可写，并标记为PTE\_COW。

首先遍历父进程页表，使用 walk(old, i, 0) 查找父进程页表中的 PTE（Page Table Entry）。检查 PTE 是否有效（PTE\_V 必须置位），否则触发 panic。

修改父进程 PTE，清除写权限（PTE\_W），添加自定义 PTE\_COW 标志。

保留其他权限（通过 PTE\_FLAGS(\*pte) 提取）。映射到子进程页表，直接复用父进程的物理页（pa），不分配新内存。调用 mappages(new, i, PGSIZE, pa, flags) 建立子进程映射。管理引用计数，调用 addref("uvmcopy()", (void\*)pa) 递增物理页的引用计数（需实现引用计数机制）。错误处理方面，若 mappages 失败，回滚已映射的页（uvmunmap(new, 0, i, 1)）。



接着，我们需要处理trap.c中的内容以处理PTE\_COW的写错误。这里写错误的标记为r\_scause() == 15。在if(pa)后面，便是复制页面的操作了，这里要注意va映射到的被复制的页面的权限应该是可写且非PTE\_COW的。



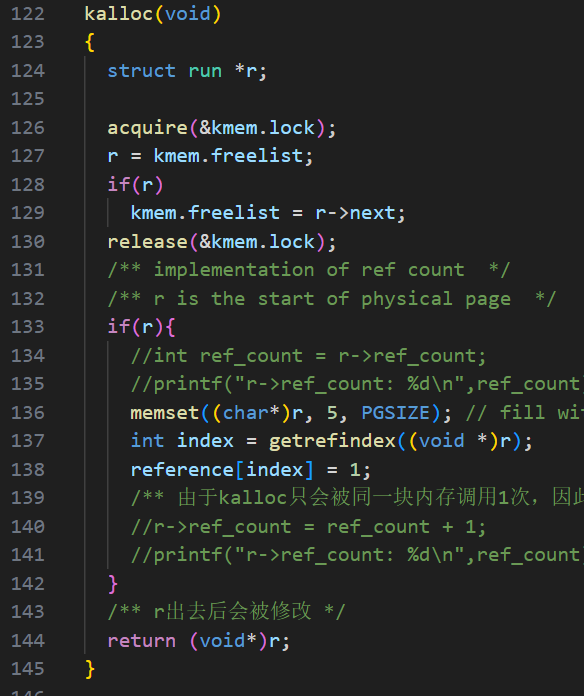
接下来，根据Hint：Next, ensure that each physical page is freed when the last PTE reference to it goes away (but not before!), perhaps by implementing reference counts in kalloc.c，我们要在kalloc.c中实现页面引用。经过计算，kalloc最多可分配32723的物理页面，因此，我直接开辟了一个32723大小的ref数组，用以记录。

Getrefindex函数计算物理地址 pa 在 reference 数组中的索引

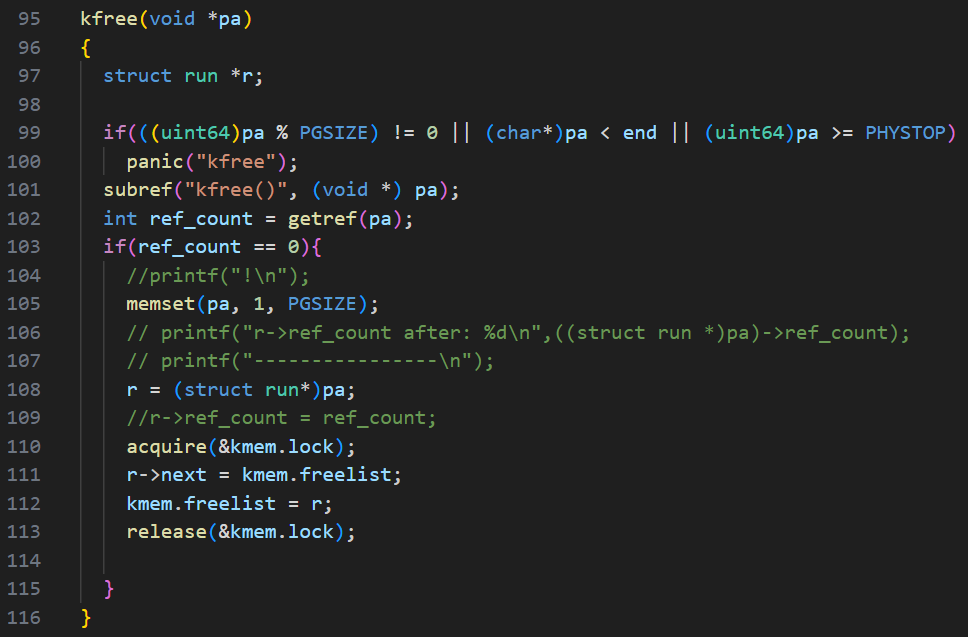
Getref函数返回指定物理页的引用计数

Addref函数递增物理页的引用计数

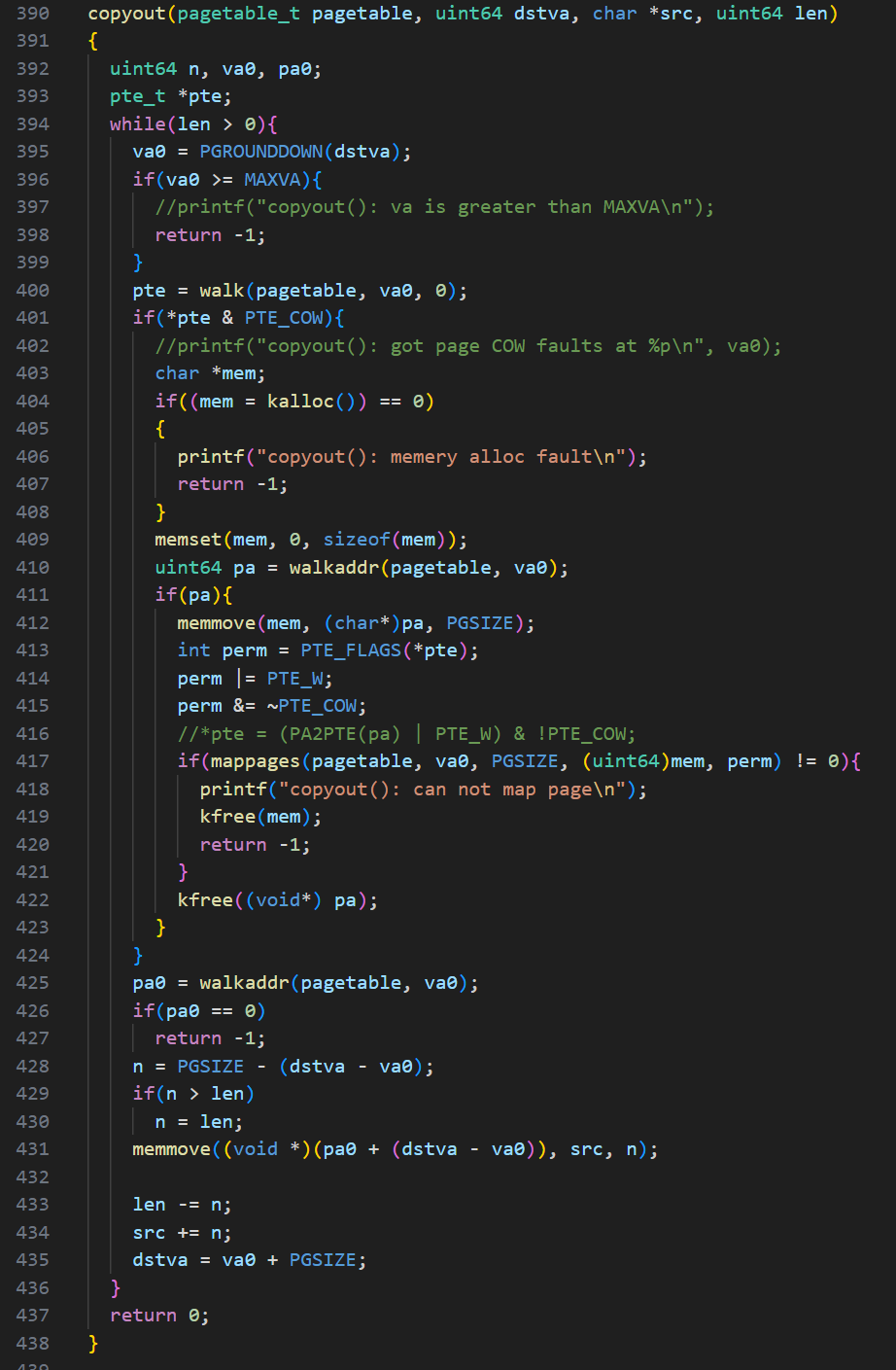
Subref函数递减物理页的引用计数，若计数归零可触发释放



在调用kalloc后，物理页应该被引用1次。Reference置1

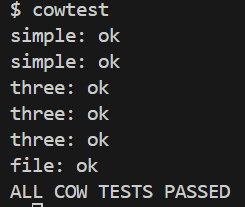


修改kfree，调用 subref("kfree()", pa) 递减该物理页的引用计数。通过 getref(pa) 获取当前引用计数 ref\_count。仅当 ref\_count == 0 时，才实际释放内存到空闲链表 kmem.freelist。



最后，根据Hint：Finally, modify copyout() to use the same scheme as page faults when it encounters a COW page，我们去修改copyout。为什么要修改copyout呢，是因为copyout是将Kernel Page复制给User Page，而用户进程要求页面是可读写的，因此要做出修改。

**实验现象：**



通过测试

**实验中遇到的问题和解决办法：**

问题描述：

在 fork() 中将父进程的页表项标记为只读（PTE\_W=0）后，父进程后续写操作触发缺页异常，但内核误判为 COW 页，导致无限递归缺页。

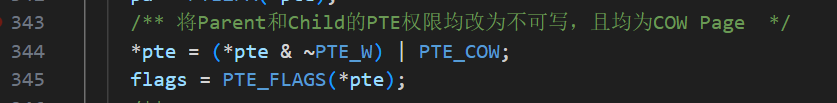
调试过程：

打印缺页地址和页表项，发现父进程的正常写操作也被当作 COW 处理。

未在页表项中记录 COW 页的特殊标记，无法区分真正的 COW 页和普通只读页。

解决办法：

利用 PTE 的保留位（如 PTE\_RSW）作为 COW 标记：

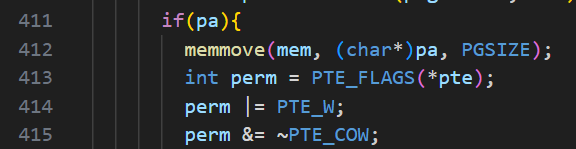


问题描述：

read()/write() 等系统调用访问用户 COW 页时，因内核无法处理用户态缺页而崩溃。

解决办法：

在 copyout() 中临时恢复写权限：



**实验心得：**

通过本次xv6写时复制（COW）fork实验，我对操作系统内存管理机制有了更深刻的理解。在实现过程中，最令我印象深刻的是通过页表权限的动态切换和引用计数机制，实现了内存资源的智能共享与保护。调试过程中遇到的递归缺页问题让我意识到内核开发必须考虑所有可能的执行路径，而引用计数的实现则教会我如何精确管理共享资源。这次实验不仅让我掌握了COW的核心原理，更让我体会到操作系统设计中"零拷贝优化"的精妙之处——在保证正确性的前提下，通过延迟实际拷贝操作来提升性能。这些经验加深了我对虚拟内存、进程隔离等概念的理解，也为后续学习更高级的内存管理技术打下了坚实基础。