Lab5:Copy-on-Write Fork for xv6

2251920
朱明灿

环境搭建

新建lab5文件夹，重新git clone源码
git checkout cow切换分支

实验目的

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。如果父节点很大，复制可能需要很长时间。更糟糕的是，这些工作通常都被浪费了;例如，fork()后跟子进程中的exec()将导致子进程放弃复制的内存，可能根本不会使用大部分内存。另一方面，如果父节点和子节点都使用一个页面，并且其中一个或两个都写入该页面，则确实需要副本。
copy-on-write (COW) fork()的目标是推迟为子进程分配和复制物理内存页面，直到实际需要这些副本(如果需要的话)。

实验内容

1. 新建cow.c并定义cows数组记录每个物理页的引用数

```
struct {
  uint8 ref_cnt;
  struct spinlock lock;
} cows[(PHYSTOP - KERNBASE) >> 12];
```

2. cow.c中定义increfcnt()和decrefcnt()函数用于增加或减少某个页面的引用数
3. 添加相关声明并修改makefile
4. uvmcopy函数负责fork时将父进程用户页表拷贝进子进程，修改该函数，不再使用kalloc()分配页面而是将子进程的虚拟页映射在父进程相同的物理页上，并添加PTE_COW标志位
5. 编写walkcowaddr函数，用于COW页面的获取物理地址，核心是

```
// 拷贝页表内容
memmove(mem, (void*)pa, PGSIZE);
// 更新标志位
flags = (PTE_FLAGS(*pte) & (~PTE_COW)) | PTE_W;
// 取消原映射
uvmunmap(pagetable, PGROUNDDOWN(va), 1, 1);
// 更新新映射
if (mappages(pagetable, PGROUNDDOWN(va), PGSIZE, (uint64)mem, flags) != 0) {
    kfree(mem);
    return 0;
}
```

6. 在usertrap函数中，需要增加一个walkcowaddr的判断

```
if (walkcowaddr(p->pagetable, r_stval()) == 0) {
      goto bad;
}
```

7. 在coupyout函数中，将walkaddr修改为walkcowaddr
8. 考虑其他函数对引用数的影响，在kalloc函数中，分配了物理页则需要调用increfcnt增加引用数；在kfree函数中，释放了物理页，需要调用decrefcnt减少引用数，并确保引用数为0时才进行释放操作；在freerange函数中，会调用kfree将空闲内存添加到freelist中，为了确保不会对刚初始化的页面调用kfree导致溢出，在调用kfree之前先调用increfcnt，这样刚好使空闲页面的引用数为0执行释放

问题的发现与解决

1. kalloc报错panic，显然是由于无空闲页面可供分配，因为freerange函数调用kfree时引用数为0，再减1导致溢出为255，无法识别所有的空闲页，通过在freerange调用kfree之前调用increfcnt即可
2. remap报错panic，表明是重映射了，在walkcowaddr中需要调用uvmunmap进行解除映射再重新映射

实验心得

通过本次Lab5的实验，我深入了解了Copy-on-Write Fork在xv6操作系统中的实现原理和作用。在实验中，我们通过新增cow.c文件和相应的函数，实现了延迟为子进程分配和复制物理内存页面的功能，直到实际需要这些副本。这样可以避免不必要的内存复制和提高系统性能。
在实验过程中，我学习了如何管理物理页面的引用计数，通过增加和减少引用数来控制页面的复制和释放。同时，我对虚拟内存的管理和页面映射有了更深入的理解，在实现COW时需要谨慎处理页面映射

关系，避免出现错误。
在解决问题的过程中，我遇到了kalloc和remap等函数报错的情况，通过分析代码逻辑和调试排查，我成功解决了这些问题并加深了对代码执行流程的理解。同时，通过调整内存分配和释放的顺序，我成功避免了一些潜在的内存管理问题，保证了系统的稳定性和正确性。