# Lab6 Copy-on-Write Fork for xv6

### 实验提要

COW fork() creates just a pagetable for the child, with PTEs for user memory pointing to the parent’s physical pages. COW fork() marks all the user PTEs in both parent and child as not writable. When either process tries to write one of these COW pages, the CPU will force a page fault. The kernel page-fault handler detects this case, allocates a page of physical memory for the faulting process, copies the original page into the new page, and modifies the relevant PTE in the faulting process to refer to the new page, this time with the PTE marked writeable. When the page fault handler returns, the user process will be able to write its copy of the page.

COW fork() makes freeing of the physical pages that implement user memory a little trickier. A given physical page may be referred to by multiple processes’ page tables, and should be freed only when the last reference disappears.

实现 fork 懒复制机制，在进程 fork 后，不立刻复制内存页，而是将虚拟地址指向与父进程相同的物理地址。在父子任意一方尝试对内存页进行修改时，才对内存页进行复制。 物理内存页必须保证在所有引用都消失后才能被释放，这里需要有引用计数机制。

## Implement copy-on write

### 实验目的：

在xv6内核中实现copy-on-write fork。如果修改后的内核同时成功执行`cowtest`和`usertests`程序就完成了。

实验步骤：

1.仿照’PTE\_\*’,在kernel/riscv.h中定义’PTE\_RSW’表示RISC-V中的reserved for software位

#define PTE\_RSW (1L << 8) // reserved for software

2.在kalloc.c中的kmem结构中增加一些成员，并且添加宏INDEX，表示右移12位

struct {

  struct spinlock lock;

  struct run \*freelist;

  struct spinlock ref\_lock;

  uint \*ref\_count;

} kmem;

#define INDEX(pa) (((char\*)pa - (char\*)PGROUNDUP((uint64)end)) >> 12)

3.实现用于抽象计数相关操作的函数并在def.h中添加声明

int get\_kmem\_ref(void \*pa){

  return kmem.ref\_count[INDEX(pa)];

}

void add\_kmem\_ref(void \*pa){

  kmem.ref\_count[INDEX(pa)]++;

}

void acquire\_ref\_lock(){

  acquire(&kmem.ref\_lock);

}

void release\_ref\_lock(){

  release(&kmem.ref\_lock);

}

4.修改kernel/vm.c中的uvmcopy，

`uvmcopy()`负责在`fork`时，完成子进程对父进程用户页表的拷贝，COW的主要机制就是影响这个部分，COW机制不会实际拷贝，而是将子进程虚拟页同样映射在与父进程相同的物理页上。注意的是，需要移除原本的写标志位 `PTE\_W`, 并添加 COW 标志位 `PTE\_COW`；最后需要对这个物理页的引用做+1。

  for(i = 0; i < sz; i += PGSIZE){

    if((pte = walk(old, i, 0)) == 0)

      panic("uvmcopy: pte should exist");

    if((\*pte & PTE\_V) == 0)

      panic("uvmcopy: page not present");

    // 清除 PTE\_W 标志，增加 COW标志

    \*pte = ((\*pte) & (~PTE\_W)) | PTE\_RSW;

    pa = PTE2PA(\*pte);

    flags = PTE\_FLAGS(\*pte);

    // COW不需要copy

    // if((mem = kalloc()) == 0)

    //   goto err;

    // memmove(mem, (char\*)pa, PGSIZE);

    if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){

      // kfree(mem);

      goto err;

    }

    add\_kmem\_ref((void\*)pa);// ref++

  }

  return 0;

5.修改usertrap。当cow页面中出现page fault时就要释放新页面，并把就页面复制到新页面中，设置PTE\_W=true。

  if(r\_scause() == 8){

    // system call

    if(p->killed)

      exit(-1);

    // sepc points to the ecall instruction,

    // but we want to return to the next instruction.

    p->trapframe->epc += 4;

    // an interrupt will change sstatus &c registers,

    // so don't enable until done with those registers.

    intr\_on();

    syscall();

  } else if (r\_scause() == 15) { // write page fault

    uint64 va = PGROUNDDOWN(r\_stval());

    pte\_t \*pte;

    if (va > p->sz || (pte = walk(p->pagetable, va, 0)) == 0){

      p->killed = 1;

      goto end;

    }

    if (((\*pte) & PTE\_RSW) == 0 || ((\*pte) & PTE\_V) == 0 || ((\*pte) & PTE\_U) == 0){

      p->killed = 1;

      goto end;

    }

    uint64 pa = PTE2PA(\*pte);

    acquire\_ref\_lock();

    uint ref = get\_kmem\_ref((void\*)pa);

    if (ref == 1){

      \*pte = ((\*pte) & (~PTE\_RSW)) | PTE\_W;

    }

    else {

      char\* mem = kalloc();

      if (mem == 0){

        p->killed = 1;

        release\_ref\_lock();

        goto end;

      }

      memmove(mem, (char\*)pa, PGSIZE);

      uint flag = (PTE\_FLAGS(\*pte) | PTE\_W) & (~PTE\_RSW);

      if (mappages(p->pagetable, va, PGSIZE, (uint64)mem,flag) != 0){

        kfree(mem);

        p->killed = 1;

        release\_ref\_lock();

        goto end;

      }

      kfree((void\*)pa);

    }

    release\_ref\_lock();

  } else if((which\_dev = devintr()) != 0){

    // ok

  } else {

    printf("usertrap(): unexpected scause %p pid=%d\n", r\_scause(), p->pid);

    printf("            sepc=%p stval=%p\n", r\_sepc(), r\_stval());

    p->killed = 1;

  }

  if(p->killed)

    exit(-1);

  // give up the CPU if this is a timer interrupt.

  if(which\_dev == 2)

    yield();

  usertrapret();

}

1. 修改mappages，防止该函数在PTE\_V非法时出现panic

for(;;){

    if((pte = walk(pagetable, a, 1)) == 0)

      return -1;

    // if(\*pte & PTE\_V)

    //   panic("remap");

    \*pte = PA2PTE(pa) | perm | PTE\_V;

    if(a == last)

      break;

    a += PGSIZE;

    pa += PGSIZE;

1. 修改kinit、freerange和kfree，初始化计数并计数为0后才释放

void

kinit()

{

  initlock(&kmem.lock, "kmem");

  freerange(end, (void\*)PHYSTOP);

  uint64 physical\_pages = ((PHYSTOP - (uint64)end) >> 12) + 1;

  physical\_pages = ((physical\_pages \* sizeof(uint)) >> 12) + 1;

  kmem.ref\_count = (uint\*) end;

  uint64 offset = physical\_pages << 12;

  freerange(end + offset, (void\*)PHYSTOP);

}

  for(; p + PGSIZE <= (char\*)pa\_end; p += PGSIZE){

    kmem.ref\_count[INDEX((void\*)p)] = 1;

    kfree(p);

  }

  // 用于检查是否计数为0

  acquire(&kmem.lock);

  if (--kmem.ref\_count[INDEX(pa)]){

    release(&kmem.lock);

    return;

  }

  release(&kmem.lock);

1. 修改kalloc，将kmem.ref\_count[INDEX((void\*)r)]设为1

  acquire(&kmem.lock);

  r = kmem.freelist;

  if(r){

    kmem.freelist = r->next;

    kmem.ref\_count[INDEX((void\*)r)] = 1;

  }

  release(&kmem.lock);

1. 仿照usertraps，修改copyout

int

copyout(pagetable\_t pagetable, uint64 dstva, char \*src, uint64 len)

{

  uint64 n, va0, pa0;

  pte\_t\* pte;

  while(len > 0){

    va0 = PGROUNDDOWN(dstva);

    if(va0 >= MAXVA)

      return -1;

    if((pte = walk(pagetable, va0, 0)) == 0)

      return -1;

    if (((\*pte & PTE\_V) == 0) || ((\*pte & PTE\_U)) == 0)

      return -1;

    pa0 = PTE2PA(\*pte);

    if (((\*pte & PTE\_W) == 0) && (\*pte & PTE\_RSW)){

      acquire\_ref\_lock();

      if (get\_kmem\_ref((void\*)pa0) == 1) {

          \*pte = (\*pte | PTE\_W) & (~PTE\_RSW);

      }

      else {

        char\* mem = kalloc();

        if (mem == 0){

          release\_ref\_lock();

          return -1;

        }

        memmove(mem, (char\*)pa0, PGSIZE);

        uint new\_flags = (PTE\_FLAGS(\*pte) | PTE\_RSW) & (~PTE\_W);

        if (mappages(pagetable, va0, PGSIZE, (uint64)mem, new\_flags) != 0){

          kfree(mem);

          release\_ref\_lock();

          return -1;

        }

        kfree((void\*)pa0);

      }

      release\_ref\_lock();

    }

    pa0 = walkaddr(pagetable, va0);

    if(pa0 == 0)

      return -1;

    n = PGSIZE - (dstva - va0);

    if(n > len)

      n = len;

    memmove((void \*)(pa0 + (dstva - va0)), src, n);

    len -= n;

    src += n;

    dstva = va0 + PGSIZE;

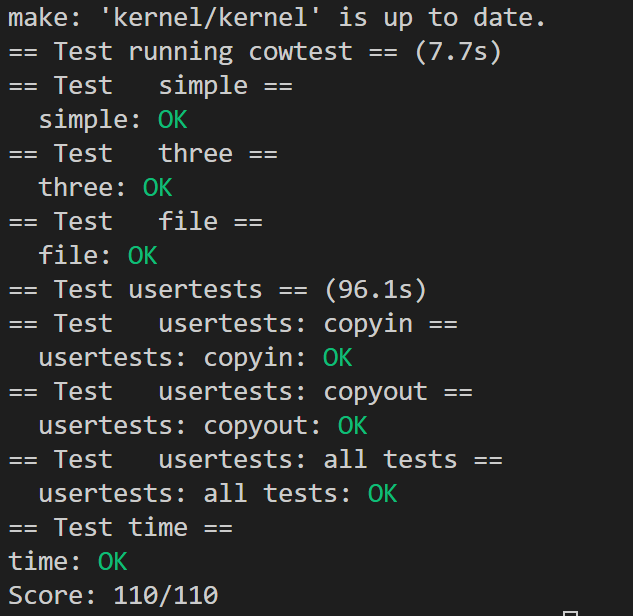
  }

  return 0;

}

退出xv6，在终端输入./grade-lab-cow

得到实验结果如下：



符合预期。

### 实验中遇到的问题及解决办法

1. 当进行cowtest时出现了panic:remap，表示虚拟页发生重复映射

解决方法：’walkcowaddr()’调用’uvmunmap’先移除原先的映射

1. 这次实验使用了位运算，起初弄不清具体逻辑，最后在舍友帮助下理解好了位运算等运算符的具体作用

### 实验小结

这次实验实现了类似lazy的策略，copy on write fork，这是一种非常常见的操作系统优化方法，压缩数据大小等优化思想都能在这个实验中得以体现，这对于今后的算法优化等方面都是非常重要的。