Lab 5 Copy-on-Write Fork for XV6

目录

[1. Implement coy-on-write fork (hard) 1](#_Toc174491215)

[1.1 实验目的 1](#_Toc174491216)

[1.2 实验步骤 1](#_Toc174491217)

[1.3 实验结果 6](#_Toc174491218)

[1.4 实验中遇到的问题和解决方法 10](#_Toc174491219)

[1.5 实验心得 10](#_Toc174491220)

[2. 测试结果 11](#_Toc174491221)

Implement coy-on-write fork (hard)

* 1. 实验目的
* 理解写时复制（copy-on-write）的概念和原理：通过实现COW fork()系统调用，深入了解写时复制的工作原理，以及如何延迟内存分配和复制来提高性能和节省内存
* 掌握操作系统内核开发的基本技能：通过修改xv6内核来实现COW fork()，获得在实际操作系统内核开发中所需的技能和经验
* 优化操作系统的性能和资源利用率：通过引入写时复制机制，内核可以避免不必要的内存复制操作，从而减少了内存开销和时间消耗
  1. 实验步骤
* 学习COW机制的原理：

传统的 fork( ) 实现：

在传统的 fork( ) 系统调用中，父进程的所有用户空间内存都会被复制到子进程中。这样做的缺点是：如果父进程占用的内存较大，复制操作会非常耗时。许多情况下，子进程在 fork( ) 之后会立刻执行 exec( ) 调用，从而丢弃刚刚复制的内存，导致大量不必要的内存复制和浪费。

COW 机制的优化：为了提高效率，COW 机制推迟了内存复制的时机，只有在子进程实际需要写入内存时才进行物理内存页面的分配和复制。具体做法是：在 fork( ) 时，父进程和子进程共享相同的物理内存页面，而不是直接复制这些页面。共享的页面在页表中被标记为不可写（通过清除 PTE\_W 标志），即父进程和子进程都只能读取这些页面，但不能写入。当父进程或子进程尝试写入某个共享页面时，CPU 会检测到该页面不可写，并产生一个页面错误。

页面错误处理：

当页面错误发生时，内核的页面错误处理程序会执行以下操作：为触发错误的进程分配一个新的物理页面。将原始共享页面的内容复制到新分配的页面中。修改触发错误进程的页表，将相关的页表项指向新的物理页面，并将该页表项标记为可写（设置 PTE\_W 标志）。这样，只有在真正需要写入时才会进行内存复制，实现了内存的延迟复制和高效利用。

* 在  kernel/riscv.h 中设置新的PTE标记位，标记一个页面是否采用了 COW 机制

#define PTE\_COW (1L << 8) // 将RSW(第8位)作为COW标识位

* 修改 uvmcopy() 函数，使其将父进程的物理页面映射到子进程，而不是为子进程分配新页面。对于设置了 PTE\_W 的页面，在父子进程的 PTE 中清除 PTE\_W（可写入标志）

int

uvmcopy(pagetable\_t old, pagetable\_t new, uint64 sz)

{

pte\_t\* pte;

uint64 pa, i;

uint flags;

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");

if (\*pte & PTE\_W) {

\*pte &= ~PTE\_W;

\*pte |= PTE\_COW;

}

pa = PTE2PA(\*pte);

flags = PTE\_FLAGS(\*pte);

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

goto err;

}

kref\_lock();

inc\_refcnt(pa);

kref\_unlock();

}

return 0;

err:

// There is no need to release these pages, because the parent process will

still use them.

uvmunmap(new, 0, i / PGSIZE, 0);

return -1;

} / PGSIZE, 1);

return -1;

}

* 在 kernel/kalloc.c 中新建用于维护物理页的引用数的结构体

static struct {

struct spinlock lock;

int cnt[REF\_CNT\_IDX(PHYSTOP)];

} kref;

* 修改kinit( ),初始化物理页引用计数锁

void

kinit()

{

initlock(&kmem.lock, "kmem");

initlock(&kref.lock, "kref");

init\_ref\_cnt();

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

}

* 修改 kalloc 函数，将引用计数初始化为1

void\*

kalloc(void)

{

struct run\* r;

acquire(&kmem.lock);

r = kmem.freelist;

if (r)

kmem.freelist = r->next;

release(&kmem.lock);

if (r) {

set\_refcnt((uint64)r, 1);

memset((char\*)r, 5, PGSIZE); // fill with junk

}

return (void\*)r;

}

* 修改 kfree 函数，每次进程从页表中删除页面时，减少引用计数。只有当引用计数为零时，kfree() 才将页面真正放回空闲列表。

void

kfree(void\* pa)

{

struct run\* r;

if (((uint64)pa % PGSIZE) != 0 || (char\*)pa < end || (uint64)pa >= PHYSTOP)

panic("kfree");

acquire(&kref.lock);

if (dec\_refcnt((uint64)pa) > 0) {

release(&kref.lock);

return;

}

release(&kref.lock);

// Fill with junk to catch dangling refs.

memset(pa, 1, PGSIZE);

r = (struct run\*)pa;

acquire(&kmem.lock);

r->next = kmem.freelist;

kmem.freelist = r;

release(&kmem.lock);

}

* 修改 usertrap 函数以识别页面错误。当在最初可写的 COW 页面上发生写页面错误时（错误码为15），分配一个新页面。

void

usertrap(void)

{

int which\_dev = 0;

if ((r\_sstatus() & SSTATUS\_SPP) != 0)

panic("usertrap: not from user mode");

// send interrupts and exceptions to kerneltrap(),

// since we're now in the kernel.

w\_stvec((uint64)kernelvec);

struct proc\* p = myproc();

// save user program counter.

p->trapframe->epc = r\_sepc();

if (r\_scause() == 8) { // Environment call from U-mode

// system call

if (killed(p))

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 sepc, scause, and sstatus,

// so enable only now that we're done with those registers.

intr\_on();

syscall();

}

else if (r\_scause() == 15) { // Store/AMO Page Fault

if (killed(p))

exit(-1);

if (cowtrap(r\_stval()) < 0)

setkilled(p);

}

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());

setkilled(p);

}

if (killed(p))

exit(-1);

// give up the CPU if this is a timer interrupt. if(which\_dev == 2)

yield();

usertrapret();

}

* 修改 copyout( ) 函数，使其在遇到 COW 页面时使用与页面错误相同的机制。

int

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

{

uint64 n, va0, pa0;

while (len > 0) {

va0 = PGROUNDDOWN(dstva);

pa0 = walkaddr(pagetable, va0);

if (pa0 == 0)

return -1;

n = PGSIZE - (dstva - va0);

if (n > len)

n = len;

pte\_t\* pte = walk(pagetable, va0, 0);

if (pte == 0)

return -1;

if (\*pte & PTE\_COW) {

if (cowtrap(va0) < 0)

return -1;

pa0 = walkaddr(pagetable, va0);

}

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

len -= n;

src += n;

dstva = va0 + PGSIZE;

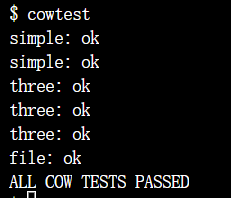
}

return 0;

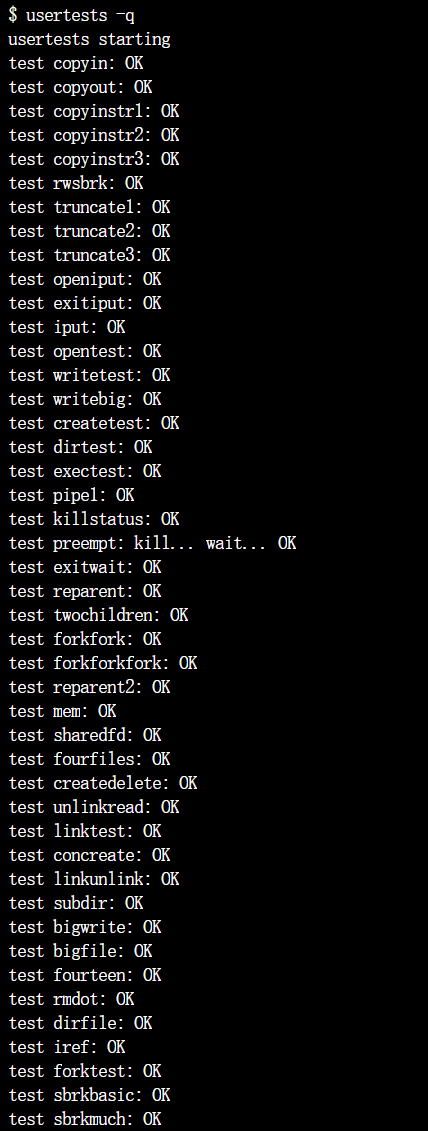
}

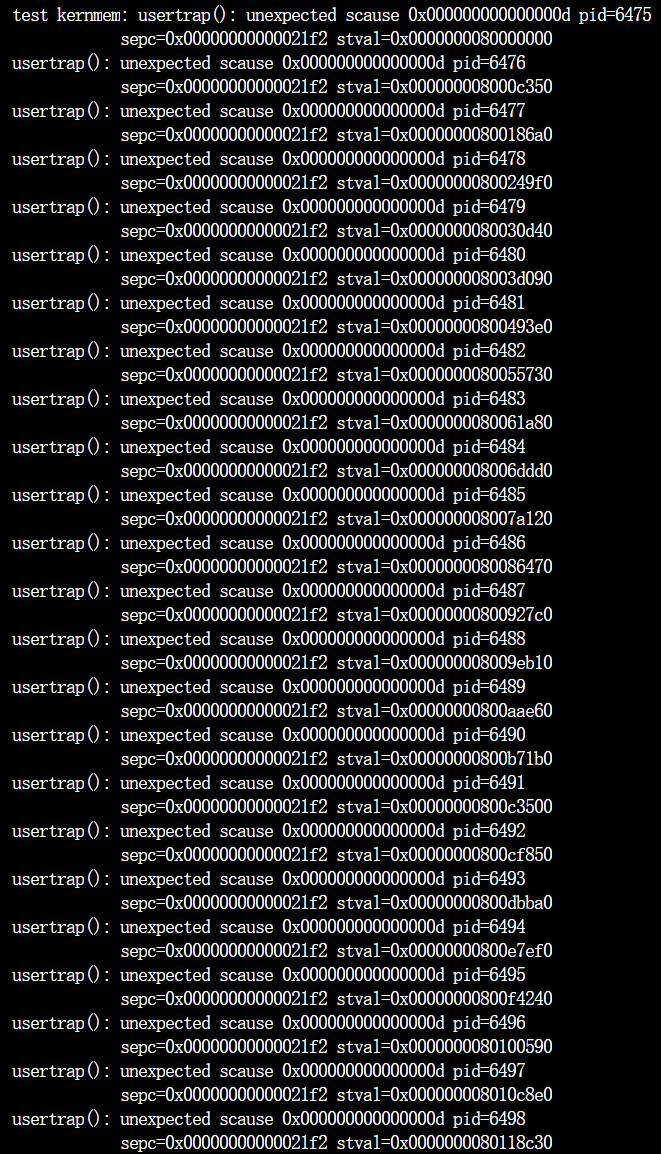
* 1. 实验结果

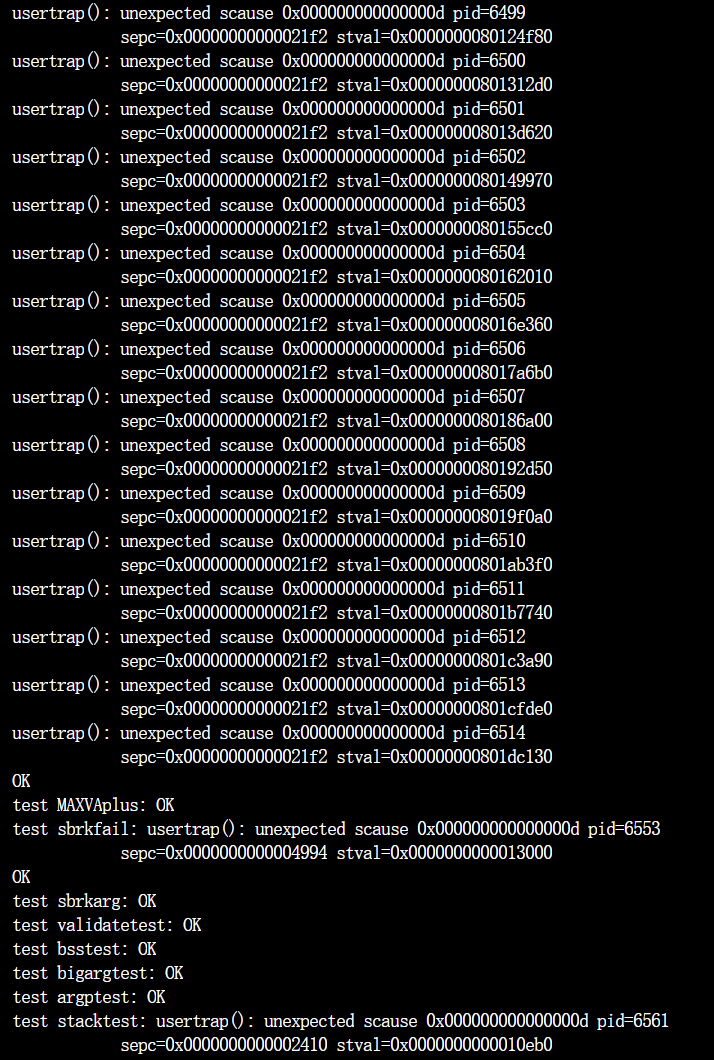
运行 make qemu 后执行 cowtest

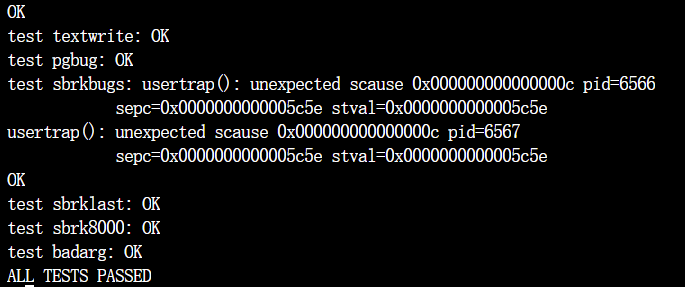


执行 usertests -q









* 1. 实验中遇到的问题和解决方法
* 问题：

在运行usertests时出现如下报错信息：

$ usertests

usertests starting

test copyin : OK

test copyout : panic: walk

* 解决方法：

定位 panic: walk 的位置，发现其判断条件是 if (va >= MAXVA) ，也即意味着发生了虚拟地址越界的错误，再定位添加的函数 isCOWPG() ，发现并没有对越界进行处理，反而是作为正常值返回，因此导致错误。添加条件判断即可。

* 1. 实验心得

通过此次实验，我深入理解了写时复制（Copy-On-Write, COW）的概念和工作原理。COW 是一种延迟内存分配和复制的技术，能够显著提升操作系统的性能并节省内存资源。通过在 xv6 操作系统中实现 COW 机制的 fork() 系统调用，我学会了如何在内核开发中进行模块的修改和功能的扩展，尤其是如何处理内存管理和进程控制。

理解了如何通过共享内存页面来避免不必要的内存复制，以及在实际写操作发生时，如何动态分配和复制页面。通过修改 uvmcopy、kalloc、kfree 等函数，学习了如何在现有内核结构中引入新的功能，特别是在保持数据一致性和资源有效利用方面。在处理页面错误时，如何为进程分配新的物理页面，并更新页表，使之指向正确的物理地址。

测试结果

