**实验六 Copy-on-Write Fork for xv6**

**6.1 实验目的**

1. 了解COW是什么

2. 了解COW有什么优势以及他是怎么对操作系统运行进行优化的

3. 了解COW的局限性

**6.2 实验内容**

6.2.1 Implement copy-on write（hard）

您的任务是在xv6内核中实现写时拷贝fork。如果修改后的内核成功地执行了cowtest和usertests程序，那么就完成了。

6.2.2 submit

提交实验。

**6.3 实验过程**

6.3.1 implement copy-on write

（1）引入COW机制后，创建子进程不是创建父进程的拷贝，而是只创建指向父进程物理页面的页表，所以我们改写uvmcopy()函数，不再给子进程分配页面，而是将父进程的物理页映射进子进程的页表，并将两个进程的PTE\_W都清零。

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");

    pa = PTE2PA(\*pte);

    reference\_count[pa >> 12] += 1; // reference count ++;

    \*pte &= ~PTE\_W;   // both child and parent can not write into this page

    \*pte |= PTE\_COW;  // flag the page as copy on write

    flags = PTE\_FLAGS(\*pte);

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

      goto err;

    }

  }

  return 0;

 err:

  uvmunmap(new, 0, i / PGSIZE, 1);

  return -1;

}

（2）当某个进程试图写某个页会触发page fault(scause=15)，这是因为此时父子进程对所有的COW页都没有写权限，因此需要在trap.c中处理异常。

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

  {

    if (cowhandler(p->pagetable, r\_stval()) < 0)

      p->killed = 1;

  }

其中cowhandler函数代码如下，其功能是检查权限位并分配新的物理页，将它映射到产生缺页异常的进程的页表中，同时设置写权限位。

int cowhandler(pagetable\_t pagetable, uint64 va)

{

  if (va >= MAXVA)

    return -1;

  pte\_t \*pte;

  pte = walk(pagetable, va, 0);

  if (pte == 0) return -1;

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

    return -1;

  // allocate a new page

  uint64 pa = PTE2PA(\*pte); // original physical address

  uint64 ka = (uint64) kalloc(); // newly allocated physical address

  if (ka == 0){

    return -1;

  }

  memmove((char\*)ka, (char\*)pa, PGSIZE); // copy the old page to the new page

  kfree((void\*)pa);

  uint flags = PTE\_FLAGS(\*pte);

  \*pte = PA2PTE(ka) | flags | PTE\_W;

  \*pte &= ~PTE\_COW;

  return 0;

}

（3）设置全局数组记录每个物理页被几个进程所拥有，这个数组可能会被多个进程同时访问，因此用一个锁进行保护。

int reference\_count[PHYSTOP >> 12];

struct spinlock ref\_cnt\_lock;

（4）对分配物理页函数kalloc进行修改

void \*

kalloc(void)

{

  struct run \*r;

  acquire(&kmem.lock);

  r = kmem.freelist;

  if(r) {

    kmem.freelist = r->next;

    acquire(&ref\_cnt\_lock);

    reference\_count[(uint64)r>>12] = 1; // first allocate, reference = 1

    release(&ref\_cnt\_lock);

  }

  release(&kmem.lock);

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

  return (void\*)r;

}

（5）在进程fork时调用uvmcopy函数使COW页对应的计数器加一，free时将计数器减一。

void

kfree(void \*pa)

{

  struct run \*r;

  int tmp, pn;

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

    panic("kfree");

  acquire(&ref\_cnt\_lock);

  pn = (uint64) pa >> 12;

  if (reference\_count[pn] < 1)

    panic("kfree ref");

  reference\_count[pn] -= 1;

  tmp = reference\_count[pn];

  release(&ref\_cnt\_lock);

  if (tmp > 0) return;

  // 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);

}

（6）如果内核调用copyout函数修改一个进程的COW页也需要进行cowhandler类似的操作。.

int

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

{

  uint64 n, va0, pa0;

  pte\_t \*pte;

  while(len > 0){

    va0 = PGROUNDDOWN(dstva);

    pa0 = walkaddr(pagetable, va0);

    if (pa0 == 0) {

      return -1;

    }

  pte = walk(pagetable, va0, 0);

    if (\*pte & PTE\_COW)

    {

      // allocate a new page

      uint64 ka = (uint64) kalloc(); // newly allocated physical address

      if (ka == 0){

        struct proc \*p = myproc();

        p->killed = 1; // there's no free memory

      } else {

        memmove((char\*)ka, (char\*)pa0, PGSIZE); // copy the old page to the new page

        uint flags = PTE\_FLAGS(\*pte);

        uvmunmap(pagetable, va0, 1, 1);

        \*pte = PA2PTE(ka) | flags | PTE\_W;

        \*pte &= ~PTE\_COW;

        pa0 = ka;

      }

    }

    n = PGSIZE - (dstva - va0);

    if(n > len)

      n = len;

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

    len -= n;

    src += n;

    dstva = va0 + PGSIZE;

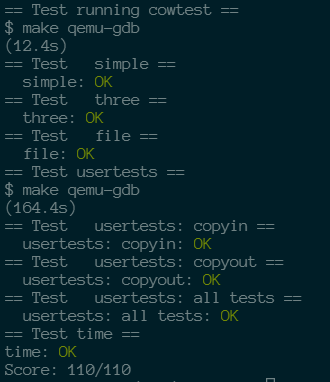
  }

  return 0;

}

**6.4 实验结果**

通过make grade对所有实验内容进行测试，结果如下图所示。



**4.5 实验小结**

本次实验介绍了COW机制，即写时拷贝技术，这项技术会在各段内容要发生变化的时候才会将父进程的内容复制给子进程。这减少了因为无意义的复制而导致的效率下降，让操作系统更加高效。

本次实验也让我更进一步地了解到一块内存的分配、使用、销毁、回收的一系列过程，即整个生命周期是如何变化的，让我对内存分配策略有了更进一步的认识。