# 2023/10/8~2023/10/9

## XV6第四章

仍然是xv6第四章的内容，和lazy allocation类似，是处理pagefault中的一种trick。Cow是一个非常通用的特性，不仅仅局限于操作系统内核的实现上，在数据库等用户应用程序场景也有许多cow的实现。

## Lab6 Cow

Cow的实现不涉及非常多的代码，或者说，cow的代码都相对集中，其核心为一个处理pagefault的函数cowalloc

int cowalloc(pagetable\_t pagetable,uint64 va){

  pte\_t\* pte;

  uint64 pa;

  uint64 flags;

  char\* mem;

  if(va>=MAXVA){

    printf("va out of bound\n");

    return -1;

  }

  if((pte=walk(pagetable,va,0))==0){

    printf("failed to find a correspond valid user pte\n");

    return -1;

  }

  if((\*pte&PTE\_C)==0){

    return 0;

  }

  pa = PTE2PA(\*pte);

  flags = (PTE\_FLAGS(\*pte)|PTE\_W)&(~PTE\_C);

  if((mem = kalloc()) == 0){

    printf("failed to alloc a new page\n");

    return -1;

  }

  memmove(mem, (char\*)pa, PGSIZE);

  uvmunmap(pagetable,PGROUNDDOWN(va),1,1);

  if(mappages(pagetable, PGROUNDDOWN(va), PGSIZE, (uint64)mem, flags) != 0){

    printf("failed to map the new page\n");

    kfree(mem);

    return -1;

  }

  return 0;

}

实现的功能为为传入的va分配一页新的内存，并且对旧物理内存的引用值减一。前置的判断主要用于检测va是否合法，以及判断这是一个应当被cow处理的蓄意的pagefault还是错误的va导致的pagefault。

需要注意的是uvmunmap的处理时机，应当在对旧物理地址的利用结束之后再进行unmap（即拷贝旧数据到新分配的页）。这个bug非常隐蔽，它会导致进程在某一地址死循环地调用cowalloc（因为当新分配的页发生错误的时候，会再次触发pagefault）；而这个bug的表现为进程执行后卡死。起初我认为是进程并发使引用计数不正确而导致的内存错误……最后上了gdb调试才发现进程在不停地调用cowalloc。

对于物理地址的引用计数，需要修改kfree（对引用计数进行判断），uvmcopy（修改引用计数），kalloc（引用计数初值）,kinit（初始化引用计数数组）。

当然，还需要一个用于引用计数的数组：

其中宏的意义很字面，PHYSPG为物理页总数，PA2INDEX用于将pa转化为refcounter数组的下标。

struct {

    struct spinlock lock;

    int refcounter[PHYSPG];

} ref\_count;

int refcount(void\* pa){

  return ref\_count.refcounter[PA2INDEX(pa)];

}

void refinit(void\* pa){

  ref\_count.refcounter[PA2INDEX(pa)] = 1;

}

void refinc(void\* pa){

  ref\_count.refcounter[PA2INDEX(pa)]++;

}

void refdec(void\* pa){

  ref\_count.refcounter[PA2INDEX(pa)]--;

}

kinit（）：

void

kinit()

{

  initlock(&kmem.lock, "kmem");

  for(uint64 p=KERNBASE;p<PHYSTOP;p+=PGSIZE){

    refinit((void\*)p);

  }

  freerange(end, (void\*)PHYSTOP);

}

kfree（）：在对pa进行操作前，对pa的引用计数进行操作和判断：

  refdec(pa);

  if(refcount(pa)>0)

    return;

kalloc（）：在成功分配物理页后，初始化其计数：

    refinit((char\*)r);

uvmcopy（）：在copy中，不再对新页表中的va分配物理页，而是建立其与旧页表对应物理页的映射。同时对对应物理页引用计数加一。

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");

    \*pte=(\*pte|PTE\_C)&(~PTE\_W);

    pa = PTE2PA(\*pte);

    flags = PTE\_FLAGS(\*pte);

    //if((mem = kalloc()) == 0)

    //  goto err;

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

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

      //kfree(mem);

      goto err;

    }

    refinc((void\*)pa);

  }

  return 0;

 err:

  uvmunmap(new, 0, i / PGSIZE, 1);

  return -1;

}

当然，对pagefault的响应应当是在usertrap的处理中进行。

else if(r\_scause()==15){

    if(cowalloc(p->pagetable,r\_stval())!=0)

      p->killed=1;

  }

根据提示，还需要修改copyout函数，因为copyout也会修改va对应的物理页。在每次copyout发生时，对对应的va调用cowalloc。因为cowalloc对于非cow va会直接返回，所以不必再加判断，统一处理即可。

while(len > 0){

    va0 = PGROUNDDOWN(dstva);

    if(cowalloc(pagetable,va0)!=0){

      return -1;

    }

    pa0 = walkaddr(pagetable, va0);

    if(pa0 == 0)

      return -1;

令人疑惑的是，我在de uvmunmap位置不合适导致的bug时，上网查阅了别人的代码，发现都在refcount中引入了锁……似乎是合理的，但是不加锁也能通过usertests和cowtest……我还曾一度以为是因为进程竞争导致了这个bug。

后续更正：不加锁会随机地在usertests中失败，这似乎很符合CPU竞争而导致的错误特征。锁的内容详见Lab8。

### 总结：

我认为我的的编码方式比网络上能找到的的大部分代码更加优越。对于refcount的处理大部分最终都在kalloc.c内，而不是四散在vm.c中；cowalloc尽可能地封装了对于cow的所有功能，而不必将一些本应交给cow处理模块的代码暴露给外部函数，并且cow对于refcount为1的特例并没有额外处理（虽然这会导致一点点额外开销）。我非常注重函数功能的完整性和封装性。

我将对其中的巧妙逻辑稍作介绍：对于所有以cow方式共享了某一内存的进程并不存在父子关系，他们是等同的。每当触发cow pagefault的时候，无论父子，都应当是复制新的物理页并建立新的映射（而不是子进程修改时子进程复制，父进程修改时仍然子进程复制）。对于refcount为1的页也同样如此，他会一致地“让出”原有页并建立新的页。

但是注意！在uvmcopy中，调用uvmunmap时dofree参数始终为1。也就是说，每次进程转移映射的时候，都会有一次释放(“让出”)旧物理页的“尝试”。一般情况下，这次尝试会最终在kfree中转变为对refcount减一；而在refcount为1时，这次尝试将会释放旧物理页。于是便“一般化”地处理了“特例”。

整个实验过程中，除开必须的refcount较为分散的代码(也几乎仅仅分散于kalloc.c中)，cow本身的实现仅仅需要自定义一个cowalloc函数，修改uvmcopy的逻辑和在copyout和trap中调用cowalloc,非常简洁。、