同濟大學
TONGJI UNIVERSITY

MIT 6.S081 实验报告

Lab 5

2151767 薛树建

日期：2023 年 7 月 20 日

目录

Lab5: Copy-on-Write Fork for xv6

1. Implement copy-on write(hard)

1) 实验目的

普通的 fork 系统调用时需要为子进程创建和父进程一样大的物理空间。如果父进程内存较小还可接受，但是如果父进程内存过大，同时子进程可以也只是用来执行 exec 调用，那么这时候为子进程分配和父进程一样大的物理内存就有些浪费了，同时进行拷贝也会增加时间开销。

本次实验是就是为 xv6 添加 cow fork 功能，cow fork 就是首先只为子进程创建进程，而不为其分配物理内存，但需要使用到的时候再去具体的分配。

大致思路便是在使用 fork 创建子进程时，只是给子进程创建一个页表，将子进程的虚拟页映射到父进程的物理页，并且将子进程的 PTE 与父进程的 PTE 设置为不可写(清空 PTE_W 位)。当任一进程试图写入其中一个页时，CPU 将强制产生页面错误。内核页面错误处理程序检测到这种情况将为出错进程分配一页物理内存，将原始页复制到新页中，并修改出错进程中的相关 PTE，将 PTE 标记为可写。当页面错误处理程序返回时，用户进程将能够写入其页面副本。

2) 实验步骤

a) 首先我们需要添加一个 PTE_COW 位，用来标记此映射是一个写时拷贝映射

```
#define PTE_U (1L << 4) // 1 -> user can access
#define PTE_COW (1L<<8) //记录此pte项是否为写时拷贝
```

当发生 page fault 时可以用来判断是否为写时拷贝，此时将对其进行内存分配。

b) 我们需要去修改 fork 函数中对子进程的物理内存的分配部分，主要时 uvmcopy 函数。Incr 函数为增加该物理地址的引用计数。

```
pa = PTE2PA(*pte);
*pte=*pte&~PTE_W;//清除写权限
*pte=*pte|PTE_COW;//设置写时拷贝权限
flags = PTE_FLAGS(*pte);
/*
if((mem = kalloc()) == 0)
  goto err;
memmove(mem, (char*)pa, PGSIZE);
*/
//将pa作为物理地址传入
if(mappages(new, i, PGSIZE, (uint64)pa, flags) != 0){

  goto err;
}
incr((void*)pa);
```

c) 之后我们需要再 usertrap 函数中去处理有页面错误引发的中断

```
}else if(r_scause()==13||r_scause()==15){//说明pagefault的中断码是13或15
  uint64 va=r_stval();//发生页面错误的虚拟地址
  if(iscowfault(p->pagetable,va)){//发生错误时，分配新的页
    if(cowalloc(p->pagetable,va)<0){
      printf("usertrap() : cowalloc failed");
      p->killed=1;
    }
  }
  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;
}
```

我们再发生 page fault 时读取发生错误的虚拟地址，检查它是否属于写时拷贝，并为之分配新的内存。

d) Iscowfault 与 cowalloc 的实现如下：

```
int iscowfault(pagetable_t pagetable,uint64 va){
  va=PGROUNDDOWN(va);
  if(va>=MAXVA)
    return 0;
  pte_t *pte=walk(pagetable,va,0);//返回虚拟地址对应
  if(pte == 0)
    return 0;
  if((*pte & PTE_V) == 0)
    return 0;
  if((*pte & PTE_U) == 0)
    return 0;
  if(*pte & PTE_COW){//该pte的cow部分为1，说明是写时
    return 1;
  }
  return 0;
}
```

主要是检查 PTE_COW 位置是否为 1.

```
int cowalloc(pagetable_t pagetable,uint64 va){
  va=PGROUNDDOWN(va);
  pte_t* pte=walk(pagetable,va,0);
  uint64 pa = PTE2PA(*pte);//获取其物理地址，以便于方便解除映射
  int flag=PTE_FLAGS(*pte);//获取pte项的flag部分
  char* mem =kalloc();
  if(mem==0){
    return -1;
  }

  memmove(mem,(char*)pa,PGSIZE);//将内容复制到mem上
  uvmunmap(pagetable,va,1,1);//解除映射

  flag=flag&~(PTE_COW);
  flag=flag|(PTE_W);
  if(mappages(pagetable,va,PGSIZE,(uint64)mem,flag)<0){
    kfree(mem);//映射失败就释放mem
    return -1;
  }
  return 0;

}
```

大体逻辑便是先获取 va 地址页对齐，然后获取其物理地址用来拷贝内存，同时解除先前的映射。修改标志位并与新的物理内存做映射。

e) 还需要定义一个各个物理内存的引用数组。如下：

```
int refcount[PHYSTOP/PGSIZE];
```

获取对应内存的引用计数与增加其引用计数的接口（注意要加锁）。

```
int getrefcount(void *pa){//获取计数
  acquire(&kmem.lock);
  int pn = (uint64)pa/PGSIZE;
  int tmp = refcount[pn];
  release(&kmem.lock);
  return tmp;
}

void incr(void *pa){//增加引用
  acquire(&kmem.lock);
  int pn = (uint64)pa/PGSIZE;
  if((uint64)pa>=PHYSTOP||refcount[pn]<1)
    panic("incr");

  refcount[pn]++;
  release(&kmem.lock);
}
//用于在uvmcopy函数映射时调用。
//释放页时会根据其引用的数值决定是否释放页
```

f) 在 kalloc 函数与 kfree 函数添加对应操作

```
acquire(&kmem.lock);
int pn = (uint64)pa / PGSIZE;
if(refcount[pn]<1)
  panic("kfree");//若引用计数小于1，则执行panic
refcount[pn]--;
int tmp = refcount[pn];
release(&kmem.lock);
if(tmp>0)
  return;//只有当引用计数为1时，此时temp应该为0，才执行释放页的操作。
struct run *r;
```

kfree 函数减少对应的引用计数，当引用计数为 0 时还正常执行释放物理页的操作。

```
  acquire(&kmem.lock);
  r = kmem.freelist;
  if(r){
    kmem.freelist = r->next;
    //当分配一个页面时，需要将引用计数设置为1
    int pn=(uint64)r/PGSIZE;
    if(refcount[pn]!=0){//若分配时，该页面引用计数不为0，执行panic
      panic("kalloc");
    }
    refcount[pn]=1;
  }
  release(&kmem.lock);
```

kalloc 首先是先从空闲列表去出一块空闲内存，将其引用计数置为 1，接着在去返回该内存的地址。

freerange 函数也要稍作修改。

```
for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE){
  refcount[(uint64)p / PGSIZE] = 1;//初始化阶段refcount
  //此时调用kfree会执行panic
  kfree(p);
```

g) 根据提示，在 copyout 中也进行对应的修改。

```
if(iscowfault(pagetable,va0)){//如果该pte项PTE_COW值为1，则应分配新的页
  if(cowalloc(pagetable,va0)<0){
    printf("copyout:alloc failed");
    return -1;
  }
}
```

h) 进行测试，结果如下：

```
$ cowtest
simple: ok
simple: ok
three: ok
three: ok
three: ok
file: ok
ALL COW TESTS PASSED
```

```
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
```

3) 实验中遇到的问题和解决方法

本实验主要是实现一个写时复制的一个功能，大体思路倒也清晰，就是创建子进程时先不要分配物理内存，只是去增添映射，当后面需要对内存写入，在单独对该进程的该写入地址分配物理页面。

实际上实现还是比较复杂的，因为各个内核函数之间关系较为紧密，在增添功能的时候往往不能考虑完全而报错。

在我实验过程 test 中出现了以下问题：

```
$ usertests
usertests starting
test MAXVAplus: panic: walk
```

在网上参考别人的做法之后发现需要在 iscowfault 中添加以下判断

```
int iscowfault(pagetable_t pagetable,uint64 va){
  va=PGROUNDDOWN(va);
  if(va>=MAXVA)
    return 0;
  pte_t *pte=walk(pagetable,va,0);//返回虚拟地址对应页表上的pte部分
  if(pte == 0)
```

4) 实验心得

通过完成本次实验，我对于操作系统的写时拷贝机制有了更加深入的理解，该机制在创建进程时，不分配内存，只有当需要对某物理内存写入时才对其分配一块的新的内存，并进行拷贝，在释放物理内存也是仅仅减少去引用计数，只有当内存块的引用计数为 0 时才释放（和 c++中的智能指针有些相似），从而提高系统运行效率与内存利用率。

不仅如此，此次实验也提高了我对操作系统的理解与编码能力。

总结

通过完成本次实验，我对于写时拷贝这一机制有了一定的了解。同时也对操作系统的内存管理理解也变得更加深入。

本实验总体评分如下：

```
tjxsj@LAPTOP-ONAKFDL3:~/xv6-labs-2021$ ./grade-lab-cow
make: 'kernel/kernel' is up to date.
== Test running cowtest == (5.4s)
== Test   simple ==
  simple: OK
== Test   three ==
  three: OK
== Test   file ==
  file: OK
== Test usertests == (115.5s)
== Test   usertests: copyin ==
  usertests: copyin: OK
== Test   usertests: copyout ==
  usertests: copyout: OK
== Test   usertests: all tests ==
  usertests: all tests: OK
== Test time ==
time: OK
Score: 110/110
```