Lab6: Copy-on-Write Fork for xv6 实验报告

在xv6中创建进程时，首先会 fork 一个子进程，然后在子进程中使用 exec 执行 shell 中的指令。在这个过程中，fork 需要完整的拷贝所有父进程的地址空间，但在 exec 执行时，又会完全丢弃这个地址空间，创建一个新的，因此会造成很大的浪费。

因此采用copy-on-write (COW：写时复制)机制，很好的解决了这个问题。

它只为子进程创建页表，但页表项指向父进程的物理页内存，然后在父子进程页表中标记所有用户页表项 PTE 均为不可写。当父子进程之一尝试写这些页面时，CPU 将产生缺页异常。内核页面错误处理程序检测到这种情况后，就会给产生异常的进程分配物理内存页，同时复制原始页面进入新页面，并将页表项标记为可写。当页面错误处理程序返回时，用户进程就能够对页面进行写操作了。

给定的物理页可能会被多个进程的页表引用，并且只有在最后一个引用消失时才应该被释放。

首先切换分支：

```
$ git fetch
$ git checkout cow
$ make clean
```

Implement copy-on write (hard)

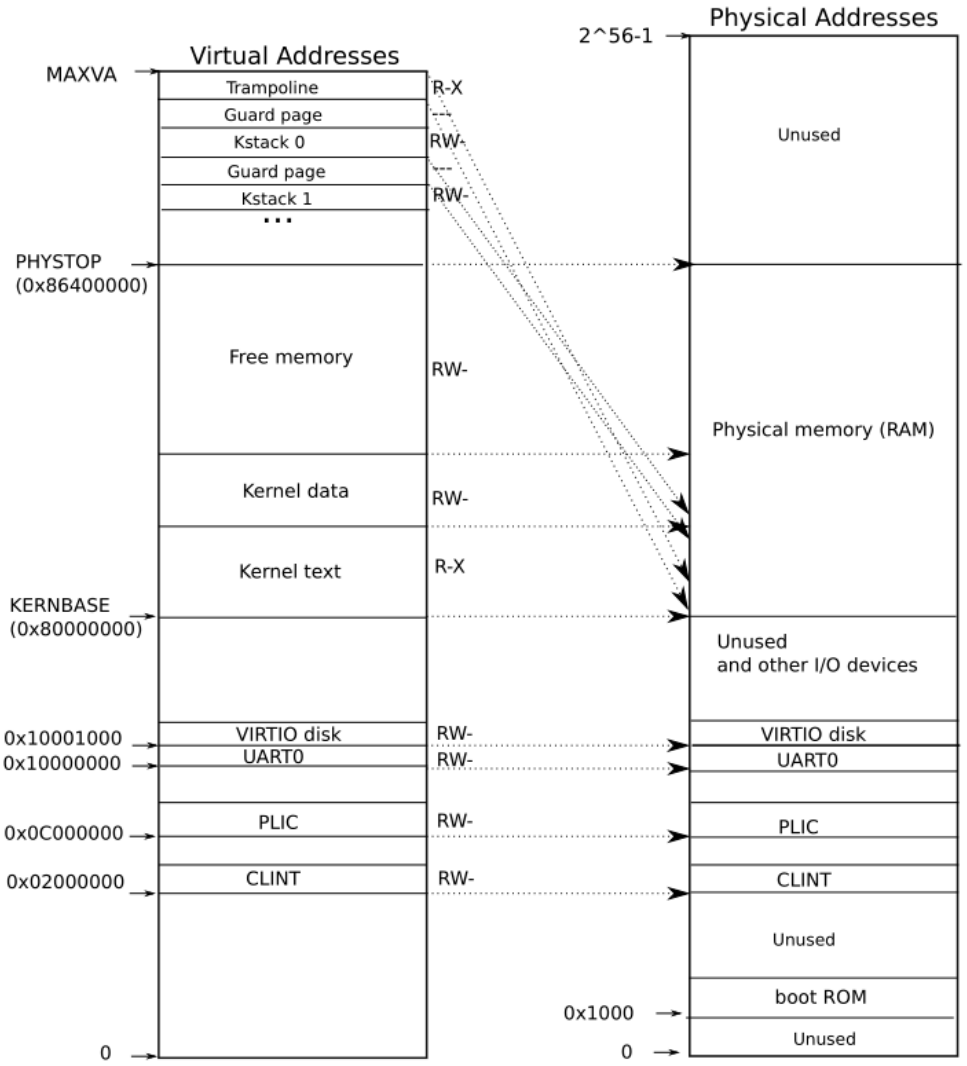
有两个场景需要处理 cow 的写入内存页场景：

- 一个是用户进程写入内存，此时会触发 page fault 中断（15号中断是写入中断，只有这个时候会触发 cow，而13号中断是读页面，不会触发 cow）；

- 另一个是直接在内核状态下写入对应的内存，此时不会触发 usertrap 函数，需要另做处理。

1. 在 kernel/kalloc.c 中定义一个用于计数的数组，对每一个页面统计有多少个进程指向了它。

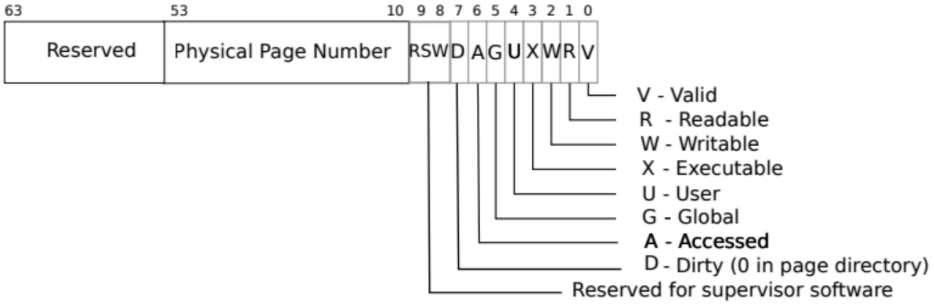
   由于用户进程映射到的范围为 KERNBASE 到 PHYSTOP（如下图），一个页表的大小（PGSIZE）是 4096，因此数组的长度可以定义为：`(PHYSTOP - KERNBASE) / PGSIZE`




```
17  uint page_ref[(PHYSTOP - KERNBASE) / PGSIZE];
```

2. 在 kernel/riscv.h 中定义 COW 标记位和计算物理内存页下标的宏函数

RISC-V 的 PTE 有 10 个标志位（如下图），其中第 8、9 位是为用户保留的，因此我们选择第 8 位作为 PTE_COW 的标志位，表示该 PTE 是否需要 copy-on-write。




```
#define PTE_V (1L << 0) // valid
#define PTE_R (1L << 1)
#define PTE_W (1L << 2)
#define PTE_X (1L << 3)
#define PTE_U (1L << 4) // 1 -> user can access
#define PTE_COW (1L << 8)  // copy on write
#define PAGE_INDEX(p) (((uint64)(p)-KERNBASE)/PGSIZE) // 物理地址p对应的物理页号
```

3. 在kernel/kalloc.c中修改kalloc函数，初始化引用计数page_ref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) {
    memset((char*)r, 5, PGSIZE); // fill with junk
    page_ref[PAGE_INDEX(r)] = 1;
  }
  return (void*)r;
}
```

修改kfree函数，释放物理页的一个引用，引用计数减 1；如果计数变为 0，则回收物理页。

```
51  void
52  kfree(void *pa)
53  {
54    struct run *r;
55
56    if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
57      panic("kfree");
58
59    acquire(&ref_lock);
60    if(page_ref[PAGE_INDEX(pa)] > 1) {
61      page_ref[PAGE_INDEX(pa)]--;
62      release(&ref_lock);
63      return;
64    }
65    page_ref[PAGE_INDEX(pa)] = 0;
66    release(&ref_lock);
67
68    // Fill with junk to catch dangling refs.
69    memset(pa, 1, PGSIZE);
70
71    r = (struct run*)pa;
72
73    acquire(&kmem.lock);
74    r->next = kmem.freelist;
75    kmem.freelist = r;
76    release(&kmem.lock);
77  }
```

设置用于page_ref数组的锁，防止内存泄漏：

```
18  struct spinlock ref_lock;
```

在kinit()函数中初始化自旋锁：

```
29  void
30  kinit()
31  {
32    initlock(&kmem.lock, "kmem");
33    initlock(&ref_lock, "ref");
34    freerange(end, (void*)PHYSTOP);
35  }
```

4. 在 kernel/vm.c 中修改 uvmcopy 函数

   将开辟内存并复制内容的代码删除，改为直接向原空间建立映射（修改父进程的权限后，子进程直接进行映射，即可获得和父进程相同的权限），并把 PTE_W 置零、 PTE_COW 置一。上述操作结束后，将该物理页的引用次数加一

```
311  int
312  uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
313  {
314    pte_t *pte;
315    uint64 pa, i;
316    uint flags;
317    //char *mem;
318
319    for(i = 0; i < sz; i += PGSIZE){
320      if((pte = walk(old, i, 0)) == 0)
321        panic("uvmcopy: pte should exist");
322      if((*pte & PTE_V) == 0)
323        panic("uvmcopy: page not present");
324      pa = PTE2PA(*pte);
325      *pte = (*pte & ~PTE_W) | PTE_COW;
326      flags = PTE_FLAGS(*pte);
327      //if((mem = kalloc()) == 0)
328      //  goto err;
329      //memmove(mem, (char*)pa, PGSIZE);
330      if(mappages(new, i, PGSIZE, pa, flags) != 0){
331        //kfree(mem);
332        goto err;
333      }
334      //acquire(&ref_lock);
335      page_ref[PAGE_INDEX(pa)]++;
336      //release(&ref_lock);
337    }
338    return 0;
339
340   err:
341    uvmunmap(new, 0, i / PGSIZE, 1);
342    return -1;
343  }
```

在代码前面添加extern声明，引入外部page_ref变量和自旋锁：

```
18  extern uint page_ref[]; // kalloc.c
19
20  extern struct spinlock ref_lock;
```

5. 与lab4相同，在 *kernel/trap.c* 中的usertrap函数中，找到中断处理的逻辑进行更改

   当写页面发生异常时，scause 寄存器的值会被置为 15，stval 寄存器会存储导致异常的地址

```
68    } else if((which_dev = devintr()) != 0){
69      // ok
70    } else if(r_scause() == 15 ){  // 如果是写时复制导致缺页异常
71      uint64 va = r_stval();
72      if(va >= p->sz)
73        p->killed = 1;
74      else if(alloc_cow(p->pagetable, va) != 0) // 给该页表项分配物理内存
75        p->killed = 1;  // 如果内存不足，则杀死进程
76    } else {
77      printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
78      printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
79      p->killed = 1;
80    }
```

6. 在kernel/ kalloc.c 中实现alloc_cow 函数，进行对copy-on-write的处理： 先解析虚拟地址，如果发现某页表项的的 PTE_COW 被置为1，则为该页分配独立内存并复制原数据，同时把 PTE_W置1、PTE_COW 置0。这样，当我们返回用户空间时，用户进程就能正常执行了。

```
100 int
101 alloc_cow(pagetable_t pagetable, uint64 va) {
102   va = PGROUNDDOWN(va);
103   if(va >= MAXVA) return -1;
104
105   pte_t *pte = walk(pagetable, va, 0); // 通过页表获取虚拟地址对应的页表项
106   if(pte == 0) return -1;
107
108   uint64 pa = PTE2PA(*pte);
109   if(pa == 0) return -1;
110
111   uint64 flags = PTE_FLAGS(*pte);
112
113   if(flags & PTE_COW) {
114     uint64 mem = (uint64)kalloc();
115     if (mem == 0) return -1;
116     memmove((char*)mem, (char*)pa, PGSIZE);
117     uvmunmap(pagetable, va, 1, 1);// 取消之前的映射
118     flags = (flags | PTE_W) & ~PTE_COW;
119   if (mappages(pagetable, va, PGSIZE, mem, flags) != 0) {
120       kfree((void*)mem);
121       return -1;
122     }
123   }
124   return 0;
125 }
```

添加引用：

```
11  #include "proc.h"
```

在kernel/defs.h中声明：

```
// kalloc.c
void*           kalloc(void);
void            kfree(void *);
void            kinit(void);
int             alloc_cow(pagetable_t, uint64);
```

7. 在kernel/vm.c中修改copyout函数，因为在呼叫copyout时，我们处于内核态，并不会触发usertrap，所以我们需要手动添加同样的监测代码，检测接收的页是否是共享COW页，若是，要额外进行复制操作

```
int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
  uint64 n, va0, pa0;
  pte_t *pte;

  while(len > 0){
    va0 = PGROUNDDOWN(dstva);
    if(alloc_cow(pagetable, va0) != 0)
      return -1;
    pa0 = walkaddr(pagetable, va0);
    if(pa0 == 0)
      return -1;
    n = PGSIZE - (dstva - va0);
    if(n > len)
      n = len;
    pte = walk(pagetable, va0, 0);
    if(pte == 0)
      return -1;
    memmove((void *)(pa0 + (dstva - va0)), src, n);

    len -= n;
    src += n;
    dstva = va0 + PGSIZE;
  }
  return 0;
}
```

8. 在kernel/defs.h中添加walk声明：

```
// vm.c
void            kvminit(void);
void            kvminithart(void);
pte_t *         walk(pagetable_t , uint64 , int);
uint64          kvmpa(uint64);
```

9. 执行测试命令：`cowtest`

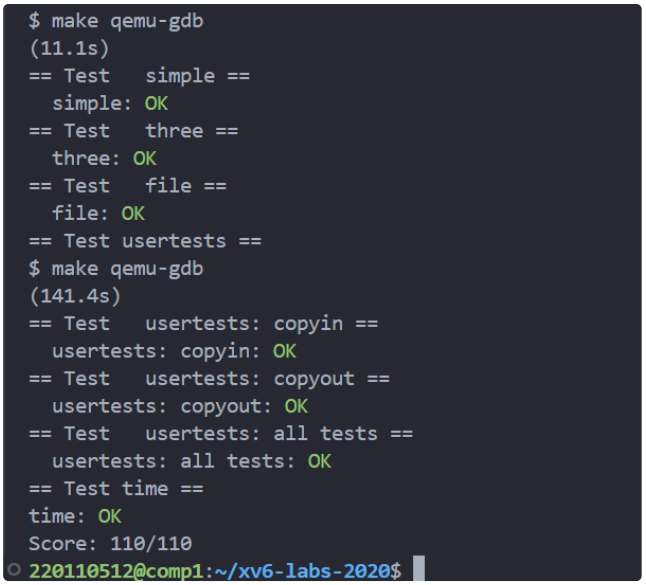
```
220110512@comp1:~/xv6-labs-2020$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=f
s.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ cowtest
simple: ok
simple: ok
three: ok
three: ok
three: ok
file: ok
ALL COW TESTS PASSED
$
```

执行测试命令：`usertests`

```
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
$
```

结果截图

执行命令`make grade`


$ make qemu-gdb
(11.1s)
== Test   simple ==
  simple: OK
== Test   three ==
  three: OK
== Test   file ==
  file: OK
== Test usertests ==
$ make qemu-gdb
(141.4s)
== 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
220110512@comp1:~/xv6-labs-2020$