

Lab5: Copy-on-Write Fork for xv6

2351882 王小萌

Tongji University, 2025 Summer

代码仓库链接: <https://github.com/wmxmw/Tongji-University-xv6-labs-2021.git>

1. Implement copy-on-write

1.1 实验目的

1.2 实验步骤

1.3 实验中遇到的问题和解决办法

1.4 实验心得

2 实验检验得分

虚拟内存提供了一种间接级别：内核可以通过将页表项（PTE）标记为无效或只读来拦截内存引用，导致页面错误，并且可以通过修改页表项来改变地址的含义。在计算机系统中，有一种说法是任何系统问题都可以通过增加一个间接层来解决。惰性分配实验提供了一个例子。本实验探讨了另一个例子：写时复制的 fork。

1 Implement copy-on write

1.1 实验目的

实验的主要目的是在 xv6 操作系统中实现写时复制（Copy-on-Write, COW）的 fork 功能。传统的 fork() 系统调用会复制父进程的整个用户空间内存到子进程，而 COW fork() 则通过延迟分配和复制物理内存页面，只在需要时才进行复制，从而提高性能和节省资源。通过这个实验，你将了解如何使用写时复制技术优化进程的 fork 操作。

1.2 实验步骤

1. 修改 kernel/vm.c 文件中的 uvmcopy() 函数，删除实际物理页面复制的语句。改为只复制父进程的页表并将父进程的物理页面映射到子进程，而不是分配新的页面。同时，清除父子进程的 PTE 中的 PTE_W 标志；页面标志 PTE_COW 用于标记写时复制页面。

```

301 uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
302 {
303     pte_t *pte;
304     uint64 pa, i;
305     uint flags;
306     char *mem;
307
308     for(i = 0; i < sz; i += PGSIZE){
309         if((pte = walk(old, i, 0)) == 0)
310             panic("uvmcopy: pte should exist");
311         if((*pte & PTE_V) == 0)
312             panic("uvmcopy: page not present");
313         pa = PTE2PA(*pte);
314         *pte &= ~PTE_W; //change to readonly
315         flags = PTE_FLAGS(*pte);
316
317         if(mappages(new, i, PGSIZE, (uint64)mem, flags)
318             //kfree(mem);
319             goto err;
320         }
321         adjustref(pa,1);//add counters
322     }
323
324     return 0;
325
326 err:
327     uvmunmap(new, 0, i / PGSIZE, 1);
328     return -1;
329 }

```

```

kernel > C kalloc.c
83
84 void adjustref(uint64 pa,int num){
85     if(pa>=PHYSTOP){
86         panic("addref:pa too big\n");
87     }
88     acquire(&kmem.lock);
89     cowcount([PA2INDEX(pa)])+=num;
90     release(&kmem.lock);
91 }

```

2. 在 kernel/riscv.h 中设置新的 PTE 标记位，标记是否为 COW 机制的页面。

```

342 #define PTE_R (1L << 1)
343 #define PTE_W (1L << 2)
344 #define PTE_X (1L << 3)
345 #define PTE_U (1L << 4) // 1 -> user can access
346 #define PTE_RSW (1L << 8) //for the error of COW page
347 // shift a physical address to the right place for a PTE.

```

3. 在 defs.h 中添加函数声明

```

62 // kalloc.c
63 void*      kalloc(void);
64 void      kfree(void *);
65 void      kinit(void);
66 void      adjustref(uint64,int);

```

```

71 int      copyin(pagetable_t, char *, uint64, uint64);
72 int      copyinstr(pagetable_t, char *, uint64, uint64);
73 int      cowalloc(pagetable_t ,uint64);
74

```

4. 在 kernel/trap.c 文件中，修改 usertrap() 函数，添加对页面错误的处理。当页面错误是写 错误时（ r_scause 寄存器值为 15 ），调用 cowfault() 函数进行写时复制处理。

```

61      p->trapframe->epc += 4;
62
63      // an interrupt will change sstatus &c registers,
64      // so don't enable until done with those registers.
65      intr_on();
66
67      syscall();
68  }else if(r_scause()==15){
69      if(cowalloc(p->pagetable,r_stval())<0){
70          p->killed = 1;
71      }
72  }else if((which_dev = devintr()) != 0){
73      // ok

```

5. . 在 kernel/kalloc.c 文件中，为每个页面维护一个引用计数。在页面分配时，将页面的引用计 数初始化为 1。

```

26 #define PA2INDEX(pa)(((uint64)pa)/PGSIZE)
27
28 int cowcount[PHYSTOP/PGSIZE];

```

```

72 kalloc(void)
73 {
74     struct run *r;
75
76     acquire(&kmem.lock);
77     r = kmem.freelist;
78     if(r)
79         kmem.freelist = r->next;
80     release(&kmem.lock);
81
82     if(r){
83         memset((char*)r, 5, PGSIZE); // fill with junk
84         int idx= PA2INDEX(r);
85         if(cowcount[idx] !=0){
86             panic("kalloc : cowcount[idx]!=0\n");
87         }
88         cowcount[idx] = 1;
89     }
90     return (void*)r;
91 }

```

freerange()中设所有物理页引用数=1;

```

36 void
37 freerange(void *pa_start, void *pa_end)
38 {
39     char *p;
40     p = (char*)PGROUNDUP((uint64)pa_start);
41     for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE){
42         cowcount[PA2INDEX(p)] = 1;
43         kfree(p);
44     }
45 }

```

在 kfree 函数中对内存引用计数减 1，如果引用计数为 0，当且仅当物理页没有被引用时才真正释放：

```

51 void
52 kfree(void *pa)
53 {
54     struct run *r;
55
56     if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYMEM)
57         panic("kfree");
58     acquire(&kmem.lock);
59     int remain = --cowcount[PA2INDEX(pa)];
60     //r->next = kmem.freelist;
61     //kmem.freelist = r;
62     release(&kmem.lock);
63     if(remain>0){
64         //仅当最后一个ref被删除时才真正释放物理页
65         return;
66     }
67
68     // Fill with junk to catch dangling refs.
69     memset(pa, 1, PGSIZE);
70
71     r = (struct run*)pa;
72 }
73
74

```

- 在 kernel/vm.c 修改 copyout() 函数，使其在遇到 COW 页面时使用与页面故障相同的方式进行处理，并定义 cow_alloc() 函数分配新的物理页：

```

348 copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
349 {
350     uint64 n, va0, pa0;
351
352     while(len > 0){
353         va0 = PGROUNDDOWN(dstva);
354         if(va0>=MAXVA){
355             printf("copyout:va exceeds MAXVA\n");
356             return -1;
357         }
358         pte_t *pte = walk(pagetable, va0, 0);
359         if(pte==0 || (*pte&PTE_U)==0 || (*pte&PTE_V)==0){
360             printf("copyout:invalid pte\n");
361             return -1;
362         }
363         if((*pte&PTE_W)==0){
364             if(cowalloc(pagetable, va0)<0){
365                 return -1;
366             }
367         }
368         pa0 = walkaddr(pagetable, va0);
369         if(pa0 == 0)
370             return -1;

```

```

451 int cowalloc(pagetable_t pagetable, uint64 va){
452     if(va >= MAXVA){
453         printf("cowalloc : exceeds MAXVA\n");
454         return -1;
455     }
456
457     pte_t *pte = walk(pagetable, va, 0);
458     if(pte == 0){
459         printf("cowalloc : pte not exists\n");
460     }
461     if((*pte & PTE_U) == 0 || (*pte & PTE_V) == 0){
462         printf("cowalloc : pte permission error\n");
463     }
464     uint64 pa_new = (uint64)kalloc();
465     if(pa_new == 0){
466         printf("cowalloc : kalloc fails\n");
467         return -1;
468     }
469     uint64 pa_old = PTE2PA(*pte);
470     memmove((void *)pa_new, (const void *)pa_old, PGSIZE);
471     kfree((void *)pa_old);
472     *pte = PA2PTE(pa_new) | PTE_FLAGS(*pte) | PTE_W;
473     return 0;
474 }

```

7. make qemu 指令运行 xv6：在命令行中分别输入 cowtest, usertests, 结果如下：

```

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

```

```

test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED

```

1.3 实验中遇到的问题和解决办法

问题：在写时复制过程中，需要正确判断页面是否需要复制。如果没有正确判断，可能会导致页面被错误地共享。

解决办法：在页面错误处理函数（cowfault()）中进行了如下判断：首先，判断触发

页面错误的虚拟地址是否有效，并检查页面是否被标记为写时复制页面（使用 PTE_COW 标志）；如果页面不是写时复制页面，表示出现了异常情况，终止进程（通过设置 killed 标志并调用 exit() 函数）；如果页面是写时复制页面，并且引用计数大于等于 2，表示页面正在被多个进程共享，此时需要为新的进程分配一个新的物理页面，并将原页面的内容复制到新页面。

1.4 实验心得


通过这个实验，深入了解了写时复制技术在操作系统中的实现原理及其优势。我们 通过修改 xv6 操作系统，成功地实现了一个简单但有效的 COW fork 功能。虽然在实现过程中遇到了一些挑战，但最终的实现证明了写时复制在提高系统性能和节省资源方面的显著优势。通过进一步的优化和改进，可以使这一技术在实际应用中发挥更大的作用。

2 实验检验得分

1. 在实验目录下创建 time.txt，填写完成实验时间数。

| | | | |
|---|----------------|------|------|
|  time.txt | 2025/7/10 0:01 | 文本文档 | 1 KB |
|---|----------------|------|------|

2. 创建 answers-cow.txt 文件,将程序运行结果填入。

| | | | |
|---|----------------|------|------|
|  answers-cow.txt | 2025/7/13 5:19 | 文本文档 | 0 KB |
|---|----------------|------|------|

3. 在终端中执行 make grade,得到 lab5 总分：

```
$ make qemu-gdb
(3.9s)
== Test simple ==
  simple: OK
== Test three ==
  three: OK
== Test file ==
  file: OK
== Test usertests ==
$ make qemu-gdb

(61.2s)
== 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
vale@Puppyyoo:~/xv6-labs-2021$
vale@Puppyyoo:~/xv6-labs-2021$
```

4. Lab5 代码提交

保存并提交到本地分支:

```
vale@Puppyyoo:~/xv6-labs-2021$ git add .
vale@Puppyyoo:~/xv6-labs-2021$ git commit -m "cow finished"
[cow 15b42b8] cow finished
7 files changed, 122 insertions(+), 12 deletions(-)
create mode 100644 answers-cow.txt
create mode 100644 time.txt
vale@Puppyyoo:~/xv6-labs-2021$
```