

Lab5 Copy-on-Write Fork for xv6

Lab5 Copy-on-Write Fork for xv6

实验内容

Implement copy-on-write fork(hard)

任务

思路

改写 `kernel/vm.c` 中的 `uvmcopy()`

代码原理说明

编写 COW handler

增加物理页计数器(`kalloc.c`)

修改 `kernel/vm.c` 中的 `copyout`

在 `kernel/riscv.h` 中添加宏定义

添加头文件

添加函数 `checkcowpage`

测试成功

实验内容

Implement copy-on-write fork(hard)

任务

- 在这个实验中，你的任务是为 xv6 实现 Copy-on-Write (COW) 的 `fork()` 系统调用。COW `fork()` 的目标是延迟物理内存页的分配和复制，直到这些页面真正需要时才执行，从而优化内存使用。
- xv6 操作系统中原来对于 `fork()` 的实现是将父进程的用户空间全部复制到子进程的用户空间。但如果父进程地址空间太大，那这个复制过程将非常耗时。另外，现实中经常出现 `fork() + exec()` 的调用组合，这种情况下 `fork()` 中进行的复制操作完全是浪费。基于此，我们可以利用页表实现写时复制机制。

思路

- COW `fork()` 的实现思路是：在 `fork()` 时不立即复制物理内存页，而是让子进程的页表指向父进程的物理页，并将这些页标记为只读。当父进程或子进程尝试写这些页时，会触发页面错误，内核会在页面错误处理程序中分配一个新页，复制原来的页到新页，并更新页表，使其指向新页并允许写入。

改写 `kernel/vm.c` 中的 `uvmcopy()`

- 在 xv6 的 `fork` 函数中，会调用 `uvmcopy` 函数给子进程分配页面，并将父进程的地址空间里的内容拷贝给子进程。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。

```
1 // Just declare the variables from kernel/kalloc.c
2 extern int useReference[PHYSTOP/PGSIZE];
3 extern struct spinlock ref_count_lock;
4
5
6 // Given a parent process's page table, copy
7 // its memory into a child's page table.
8 // Copies both the page table and the
9 // physical memory.
10 // returns 0 on success, -1 on failure.
11 // frees any allocated pages on failure.
12 int
13 uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
14 {
15     pte_t *pte;
16     uint64 pa, i;
17     uint flags;
18     // char *mem;
19
20     for(i = 0; i < sz; i += PGSIZE){
21         if((pte = walk(old, i, 0)) == 0)
22             panic("uvmcopy: pte should exist");
23         if((*pte & PTE_V) == 0)
24             panic("uvmcopy: page not present");
25         // PAY ATTENTION!!!
26         // 只有父进程内存页是可写的，才会将子进程和父进程都设置为COW和只读的；否则，都是只读的，但是不标记为COW，因为本来就是只读的，不会进行写入
```

```

27 // 如果不这样做，父进程内存只读的时候，标记为COW，那么经过缺页中断，程序
    就可以写入数据，于原本的不符合
28 if (*pte & PTE_W) {
29     // set PTE_W to 0
30     *pte &= ~PTE_W;
31     // set PTE_RSW to 1
32     // set COW page
33     *pte |= PTE_RSW;
34 }
35 pa = PTE2PA(*pte);
36
37 // increment the ref count
38 acquire(&ref_count_lock);
39 useReference[pa/PGSIZE] += 1;
40 release(&ref_count_lock);
41
42 flags = PTE_FLAGS(*pte);
43 // if((mem = kalloc()) == 0)
44 //     goto err;
45 // memmove(mem, (char*)pa, PGSIZE);
46 if(mappages(new, i, PGSIZE, (uint64)pa, flags) != 0){
47     // kfree(mem);
48     goto err;
49 }
50 }
51 return 0;
52
53 err:
54     uvmunmap(new, 0, i / PGSIZE, 1);
55     return -1;
56 }

```

代码原理说明

1. 页表项复制与COW机制：

- 这个函数的主要任务是将父进程的页表项复制到子进程的页表中，但与传统的 `fork()` 不同，COW机制下并不会立即为子进程分配新的物理内存页。相反，父进程和子进程共享相同的物理页面。
- 为了实现这一点，父进程的页表项会被修改，将页表项中的写标志位 (`PTE_W`) 清除，从而将页面设置为只读，同时设置COW标志 (`PTE_COW`)。

2. 引用计数:

- 在 `uvmcopy()` 函数中, 每个被共享的物理页面的引用计数会被增加。引用计数是用于跟踪某个物理页面被多少个进程共享。当引用计数减少到0时, 该物理页面可以被安全地释放。

3. 页面错误处理:

- 当父进程或子进程尝试写入一个被标记为COW的页面时, 由于页面被设置为只读, 会触发页面错误。此时, 内核会捕获该错误, 为进程分配一个新的物理页面, 将原页面内容复制到新页面中, 然后更新页表项, 使其指向新页面, 并设置写标志位。这就是COW机制的核心: 只有在实际写操作发生时, 才会进行页面的复制, 从而节省内存。

4. 错误处理:

- 如果在为子进程映射页面的过程中发生错误, 例如 `mappages()` 失败, 那么 `uvmcopy()` 会解除子进程中已完成的映射, 并返回错误代码 `-1`。

编写 COW handler

- 此时父子进程对所有的 COW 页都没有写权限, 如果某个进程试图对某个页进行写, 就会触发 `page fault(scause = 15)`, 因此需要在 `trap.c/usertrap` 中处理这个异常。

```
1 //
2 // handle an interrupt, exception, or system call from user
  space.
3 // called from trampoline.S
4 //
5 void
6 usertrap(void)
7 {
8     int which_dev = 0;
9
10    if((r_sstatus() & SSTATUS_SPP) != 0)
11        panic("usertrap: not from user mode");
12
13    // send interrupts and exceptions to kerneltrap(),
14    // since we're now in the kernel.
15    w_stvec((uint64)kernelvec);
16
17    struct proc *p = myproc();
18
```

```

19 // save user program counter.
20 p->trapframe->epc = r_sepc();
21
22 if(r_scause() == 8){
23     // system call
24
25     if(killed(p))
26         exit(-1);
27
28     // sepc points to the ecall instruction,
29     // but we want to return to the next instruction.
30     p->trapframe->epc += 4;
31
32     // an interrupt will change sepc, scause, and sstatus,
33     // so enable only now that we're done with those registers.
34     intr_on();
35
36     syscall();
37 }
38 else if (r_scause() == 15) {
39     // Store/AMO page fault(write page fault) and Load page
    fault
40     // see volume II: RISC-V Privileged Architectures v20211203
    Page 71
41
42     // the faulting virtual address
43     // see volume II: RISC-V Privileged Architectures v20211203
    Page 41
44     // the download url is https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf
45     uint64 va = r_stval();
46     if (va >= p->sz)
47         p->killed = 1;
48     int ret = cowhandler(p->pagetable, va);
49     if (ret != 0)
50         p->killed = 1;
51 } else if((which_dev = devintr()) != 0){
52     // ok
53 } else {
54     printf("usertrap(): unexpected scause %p pid=%d\n",
    r_scause(), p->pid);

```

```

55     printf("          sepc=%p stval=%p\n", r_sepc(),
r_stval());
56     setkilled(p);
57 }
58
59 if(killed(p))
60     exit(-1);
61
62 // give up the CPU if this is a timer interrupt.
63 if(which_dev == 2)
64     yield();
65
66 usertrapret();
67 }

```

- 我们会检查 `scause` 寄存器的值是否是 15，如果是的话就调用 `cowhandler` 函数。

```

1  int
2  cowhandler(pagetable_t pagetable, uint64 va)
3  {
4      char *mem;
5      if (va >= MAXVA)
6          return -1;
7      pte_t *pte = walk(pagetable, va, 0);
8      if (pte == 0)
9          return -1;
10     // check the PTE
11     if ((*pte & PTE_RSW) == 0 || (*pte & PTE_U) == 0 || (*pte &
PTE_V) == 0) {
12         return -1;
13     }
14     if ((mem = kalloc()) == 0) {
15         return -1;
16     }
17     // old physical address
18     uint64 pa = PTE2PA(*pte);
19     // copy old data to new mem
20     memmove((char*)mem, (char*)pa, PGSIZE);
21     // PAY ATTENTION
22     // decrease the reference count of old memory page, because
a new page has been allocated

```

```

23     kfree((void*)pa);
24     uint flags = PTE_FLAGS(*pte);
25     // set PTE_W to 1, change the address pointed to by PTE to
    new memory page(mem)
26     *pte = (PA2PTE(mem) | flags | PTE_W);
27     // set PTE_RSW to 0
28     *pte &= ~PTE_RSW;
29     return 0;
30 }
31

```

- cowhandler 做的事情也很简单，它首先会检查一系列权限位，然后分配一个新的物理页，并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。

增加物理页计数器(kalloc.c)

- 由于现在可能有多个进程拥有同一个物理页，如果某个进程退出时 free 掉了这个物理页，那么其他进程就会出错。所以我们得设置一个全局数组，记录每个物理页被几个进程所拥有。同时注意这个数组可能会被多个进程同时访问，因此需要用一个锁来保护。

```

1 // the reference count of physical memory page
2 int useReference[PHYSTOP/PGSIZE];
3 struct spinlock ref_count_lock;

```

- 每个物理页所对应的计数器将在下面几个函数内被修改：
- 首先在 kalloc 分配物理页函数中将对应计数器置为 1

```

1 // Allocate one 4096-byte page of physical memory.
2 // Returns a pointer that the kernel can use.
3 // Returns 0 if the memory cannot be allocated.
4 void *
5 kalloc(void)
6 {
7     struct run *r;
8
9     acquire(&kmem.lock);
10    r = kmem.freelist;
11    if(r) {

```

```

12     kmem.freelist = r->next;
13     acquire(&ref_count_lock);
14     // initialization the ref count to 1
15     useReference[(uint64)r / PGSIZE] = 1;
16     release(&ref_count_lock);
17 }
18 release(&kmem.lock);
19
20 if(r)
21     memset((char*)r, 5, PGSIZE); // fill with junk
22 return (void*)r;
23 }

```

- 进程在 fork 时会调用 uvmcopy 函数，我们要在其中将 COW 页对应的计数器加 1。
- 另外在某个进程想 free 掉某个物理页时，我们要将其计数器减 1。

```

1 // Free the page of physical memory pointed at by pa,
2 // which normally should have been returned by a
3 // call to kalloc(). (The exception is when
4 // initializing the allocator; see kinit above.)
5 void
6 kfree(void *pa)
7 {
8     struct run *r;
9     int temp;
10
11     if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa
12     >= PHYSTOP)
13         panic("kfree");
14
15     acquire(&ref_count_lock);
16     // decrease the reference count, if use reference is not zero,
17     // then return
18     useReference[(uint64)pa/PGSIZE] -= 1;
19     temp = useReference[(uint64)pa/PGSIZE];
20     release(&ref_count_lock);
21     if (temp > 0)
22         return;

```



```

22 // Fill with junk to catch dangling refs.
23 memset(pa, 1, PGSIZE);
24
25 r = (struct run*)pa;
26
27 acquire(&kmem.lock);
28 r->next = kmem.freelist;
29 kmem.freelist = r;
30 release(&kmem.lock);
31 }
32

```

修改 kernel/vm.c 中的 copyout

- 最后，如果内核调用 copyout 函数试图修改一个进程的 COW 页，也需要进行 cowhandler 类似的操作来处理。

```

1 // Copy from kernel to user.
2 // Copy len bytes from src to virtual address dstva in a given
  page table.
3 // Return 0 on success, -1 on error.
4 int
5 copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64
  len)
6 {
7     uint64 n, va0, pa0;
8
9     while(len > 0){
10         va0 = PGROUNDDOWN(dstva);
11         pa0 = walkaddr(pagetable, va0);
12         if(pa0 == 0)
13             return -1;
14
15         struct proc *p = myproc();
16         pte_t *pte = walk(pagetable, va0, 0);
17         if (*pte == 0)
18             p->killed = 1;
19         // check
20         if (checkcowpage(va0, pte, p))
21             {
22                 char *mem;

```

```

23     if ((mem = kalloc()) == 0) {
24         // kill the process
25         p->killed = 1;
26     }else {
27         memmove(mem, (char*)pa0, PGSIZE);
28         // PAY ATTENTION!!!
29         // This statement must be above the next statement
30         uint flags = PTE_FLAGS(*pte);
31         // decrease the reference count of old memory that va0
point
32         // and set pte to 0
33         uvmunmap(pagetable, va0, 1, 1);
34         // change the physical memory address and set PTE_W to 1
35         *pte = (PA2PTE(mem) | flags | PTE_W);
36         // set PTE_RSW to 0
37         *pte &= ~PTE_RSW;
38         // update pa0 to new physical memory address
39         pa0 = (uint64)mem;
40     }
41 }
42
43 n = PGSIZE - (dstva - va0);
44 if(n > len)
45     n = len;
46 memmove((void *)(pa0 + (dstva - va0)), src, n);
47
48 len -= n;
49 src += n;
50 dstva = va0 + PGSIZE;
51 }
52 return 0;
53 }

```

在 `kernel/riscv.h` 中添加宏定义

```

1 #define PTE_RSW (1L << 8) // RSW

```

添加头文件

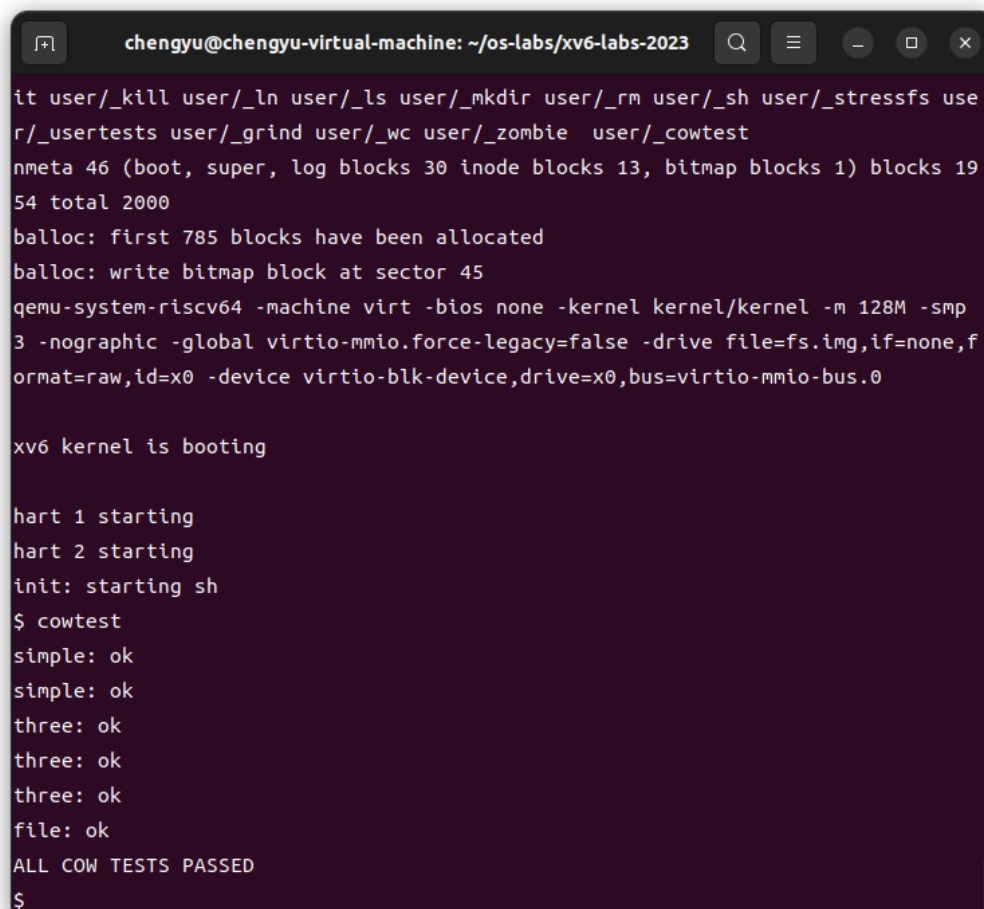
- 在 `kernel/vm.c` 中添加 `#include "proc.h"` 以及 `#include "spinlock.h"`

添加函数 `checkcowpage`

- 在 `kernel/vm.c` 中添加:

```
1 int checkcowpage(uint64 va, pte_t *pte, struct proc* p) {
2     return (va < p->sz) // va should blow the size of process
   memory (bytes)
3     && (*pte & PTE_V)
4     && (*pte & PTE_RSW); // pte is COW page
5 }
```

测试成功



```
chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023
it user/_kill user/_ln user/_ls user/_mkdir user/_rm user/_sh user/_stressfs use
r/_usertests user/_grind user/_wc user/_zombie user/_cowtest
nmeta 46 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 1) blocks 19
54 total 2000
balloc: first 785 blocks have been allocated
balloc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
3 -nographic -global virtio-mmio.force-legacy=false -drive file=fs.img,if=none,f
ormat=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

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