Lab 6: Copy-on-Write Fork for xv6

1950787 杨鑫

实验目的：

- 了解 copy-on-write 的原理和机制
- 通过修改操作系统内部文件实现 copy-on-write
- 体会 copy-on-write 的必要性和优点

实验步骤：

首先切换分支至 cow branch，以获取本次实验的内容

同时清理文件，以获得纯净的初始文件系统

```
git checkout cow
make clean
```

a. Implement copy-on write

实验目的：

写时复制 (COW) fork() 的目标是推迟为子进程分配和复制物理内存页面，直到实际需要副本（如果有的话）。
COW fork() 只为子级创建一个页表，用户内存的 PTE 指向父级的物理页面。 COW fork() 将 parent 和 child 中的所有用户 PTE 标记为不可写。当任一进程尝试写入这些 COW 页之一时，CPU 将强制发生页错误。内核页面错误处理程序检测到这种情况，为出错进程分配物理内存页面，将原始页面复制到新页面中，并修改出错进程中的相关 PTE 以引用新页面，这次使用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它的页面副本。

COW fork() 使实现用户内存的物理页面的释放变得有点棘手。一个给定的物理页可以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。

你的任务是在 xv6 内核中实现 copy-on-write fork。如果你修改的内核成功地执行了cowtest 和 usertests 程序，你就完成了。

实验分析：

本实验需要实现写时复制功能。和上一个惰性分配类似，都是基于一种需要时再分配的思想，因为在 fork() 时，一开始需要完全复制父进程的页表信息，可能会造成很大的不必要的开销，于是可以考虑只是将子进程的 PTE 映射到父进程页表上而不是完全复制其物理内容，这样将可以大大减少新建子进程的开销。但是，如果子进程要对其页表进行写入时，会造成父子进程的页表内容不一致的情况，这时候不能共享同一份内存了，必须要为子进程申请一份新的页表空间保存相应信息。

于是，我们可以在子进程创建时先使用简单的映射使子进程不真的分配内存，同时将 PTE 的 W 设为不可写状态，将其 COW （代表是否是读后写的块）设为是。然后在子进程需要写某块时，会触发页错误并由内核分配物理页面然后再修改W和COW标志。同时，删除某个页面时，需要注意是否正在被共享，如果是，则只是删除该进程的映射并减少引用数，如果只被一个进程占有则直接删除。这里需要为物理块新增一个引用的计数来判断是否应当删除，而且名次增加/减少计数时需要使用P/V操作，这是为了防止某些特殊情况的发生，这一点会在下面详细说明。

实验核心代码：

trap.c

```
int handleCOWfault(pagetable_t pagetable, uint64 va)
{
  if(va >= MAXVA)
    return -1;
  pte_t *pte = walk(pagetable, va, 0);
  if(pte == 0 || (*pte & (PTE_V)) == 0 || (*pte & PTE_U) == 0)
    return -1;
  if(*pte & PTE_W) return 0;
  if((*pte & PTE_COW) == 0)
    return -1;

  uint64 pa = PTE2PA(*pte);
  void *mem = kalloc();
  if(mem == 0)
    return -1;
  *pte = (PTE_FLAGS(*pte) & ~PTE_COW) | PTE_W | PA2PTE(mem);
  memmove((char*)mem, (char*)pa, PGSIZE);
  kfree((void*)pa);
  return 0;
}


//
// handle an interrupt, exception, or system call from user space.
// called from trampoline.S
//
void
usertrap(void)
{
  int which_dev = 0;

  if((r_sstatus() & SSTATUS_SPP) != 0)
    panic("usertrap: not from user mode");

  // send interrupts and exceptions to kerneltrap(),
  // since we're now in the kernel.
  w_stvec((uint64)kernelvec);

  struct proc *p = myproc();

  // save user program counter.
  p->trapframe->epc = r_sepc();

  if(r_scause() == 8){
```

```
    // system call

    if(p->killed)
      exit(-1);

    // sepc points to the ecall instruction,
    // but we want to return to the next instruction.
    p->trapframe->epc += 4;

    // an interrupt will change sstatus &c registers,
    // so don't enable until done with those registers.
    intr_on();

    syscall();
  } else if(r_scause() == 13 || r_scause() == 15) {
    uint64 va = r_stval();
      if (handleCOWfault(p->pagetable, va) == -1)
      p->killed = 1;
  } else if((which_dev = devintr()) != 0){
    // ok
  } 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;
  }

  if(p->killed)
    exit(-1);

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
    yield();

  usertrapret();
}
```

vm.c

```
/ Given a parent process's page table, copy
// its memory into a child's page table.
// Copies both the page table and the
// physical memory.
// returns 0 on success, -1 on failure.
// frees any allocated pages on failure.
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");
    pa = PTE2PA(*pte);
```

```
    *pte = *pte & ~PTE_W;
    *pte = *pte | PTE_COW;
    flags = PTE_FLAGS(*pte);

    if(mappages(new, i, PGSIZE, pa, flags) != 0){
      goto err;
    }
    V(PA2PID(pa));
  }
  return 0;

 err:
  uvmunmap(new, 0, i / PGSIZE, 1);
  return -1;
}


// Copy from kernel to user.
// Copy len bytes from src to virtual address dstva in a given page table.
// Return 0 on success, -1 on error.
int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
  uint64 n, va0, pa0;

  while(len > 0){
    va0 = PGROUNDDOWN(dstva);
    if (handleCOWfault(pagetable, dstva) == -1)
      return -1;
    pa0 = walkaddr(pagetable, va0);
    if(pa0 == 0)
      return -1;

    n = PGSIZE - (dstva - va0);
    if(n > len)
      n = len;
    memmove((void *)(pa0 + (dstva - va0)), src, n);

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

kalloc.c

```
// Free the page of physical memory pointed at by v,
// which normally should have been returned by a
// call to kalloc().  (The exception is when
// initializing the allocator; see kinit above.)
void
kfree(void *pa)
{
  struct run *r;

  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
```

```
    panic("kfree");

  int refcount = P(PA2PID(pa));
  if(refcount < 0) {
    panic("kfree");
  }
  if(refcount > 0) return;

  // Fill with junk to catch dangling refs.
  memset(pa, 1, PGSIZE);

  r = (struct run*)pa;

  acquire(&kmem.lock);
  r->next = kmem.freelist;
  kmem.freelist = r;
  release(&kmem.lock);
}


// Allocate one 4096-byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
void *
kalloc(void)
{
  struct run *r;

  acquire(&kmem.lock);
  r = kmem.freelist;
  if(r) {
    kmem.freelist = r->next;
    pagerefcount.refNum[PA2PID(r)] = 1;
  }
  release(&kmem.lock);

  if(r)
    memset((char*)r, 5, PGSIZE); // fill with junk
  return (void*)r;
}
```

测试：

```
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
$
```

```
$ usertests
usertests starting
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test rwsbrk: OK
test truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
test pgbug: OK
```

```
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
$
```

可以看到，可以成功通过 cowtest 和 usertests，测试成功。

实验评分：

按照实验要求对本次实验的所有小实验进行评分，结果如图：

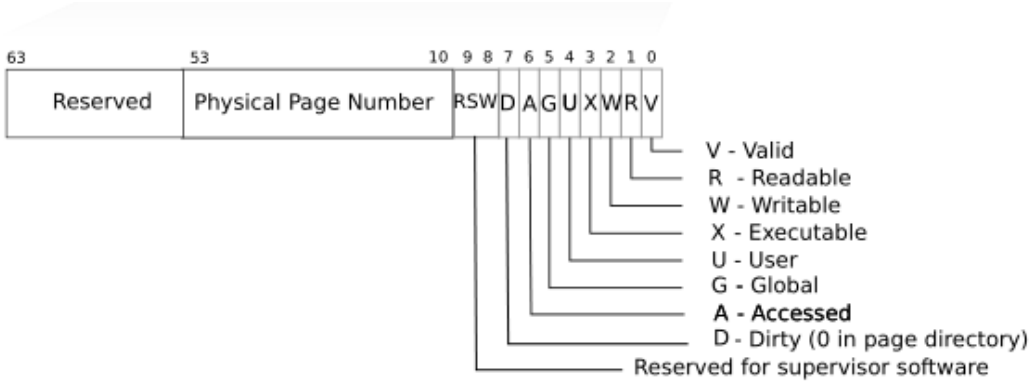
```
== Test running cowtest ==
$ make qemu-gdb
(11.5s)
== Test   simple ==
  simple: OK
== Test   three ==
  three: OK
== Test   file ==
  file: OK
== Test usertests ==
$ make qemu-gdb
(184.9s)
    (Old xv6.out.usertests failure log removed)
== Test   usertests: copyin ==
  usertests: copyin: OK
== Test   usertests: copyout ==
  usertests: copyout: OK
== Test   usertests: all tests ==
  usertests: all tests: OK
== Test time ==
time: OK
Score: 110/110
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$
```

通过了所有测试

问题以及解决办法：

a. COW 标志

由于在发生页错误时，需要判断是不是由于COW引起的，所以需要为所有COW的块增设一个标志。这里需要查看PTE的结构，如下图所示。可以发现大多数已经被使用，不能被取代，只有RSW作为保留位可以使用，于是这里取第8位（0为起始位）作为COW标志，用于区分是否是COW的块。



b. 报错：FAILED -- lost some free pages

在实验过程中，我期间总是遇到这个报错，cowtest可以正常通过，但是在usertests里就会出现这个报错。我查阅了一些资料和查看了代码后，发现该错误是 usertests 检测到运行前后的空闲页数量减少，也就是检测到发生了内存泄漏。内存泄漏的主要原因是因为进程调度的存在。因为xv6是一个分时操作系统，他并不是安装顺序执行进程，而是通过调度算法分时的执行一系列进程，所以要注意进程同步，互斥可能带来的异常。因为一开始使用的是简单的引用+/-1，所以没法排除这种机制带来的隐患，可能

会导致内存泄漏等情况发生。所以需要使用P/V机制来避免这种错误，这也让我对于进程同步/互斥的理论知识得到了回顾。

实验心得：

1. 本次实验与上次实验大体思想类似，都是一种惰性分配的思想，从而尽可能的减少不必要的开销。
2. 这次实验主要解决了 fork() 时的惰性分配的问题，同时对于PTE的结构和页表机制也有了新的体会。
3. 对于页表的分配，释放等操作有了更多的了解。
4. 此次实验还踩了进程同步的“坑”，经过查阅资料后通过P/V机制来解决了问题，对于进程的同步互斥机制有了新的体会，并尝试去解决问题。