实验报告

Lab COW

1. 实验背景与原理

xv6中，系统调用`fork()`**将父进程的在用户空间的所有内容复制到子进程中**。这也就是说，父进程和子进程**有着独立的地址空间，它们将相同的数据放在不同的地方**。

但这种做法造成了很大的浪费。第一，如果父进程很大，那么创建子进程就很花费时间；第二，子进程经常会调用`exec()`以执行新的任务，这会抛弃掉所有复制的内容，而这些内容却很可能根本没用过。

通过完成copy-on-write(COW) fork()，可以将分配和复制物理页推迟到这些复制确实需要时。它为子进程创建的页表包含PTE指向父进程的物理页，并将父进程和子进程中这些PTE都标记为只读(PTE_W那一位设为0)。当任一进程要写这些COW页时，CPU会产生一个缺页故障，内核中的缺页handler将会为产生故障的进程分配新的页，并将原本的页复制到新页上，这次将对应的PTE标记为可写。当缺页故障handler返回时，用户进程就能写页了。

在COW fork()下，一个物理页可能被映射到多个进程的页表上，并且仅当最后指向该物理页的引用消失时释放掉该物理页。

2. 实验过程

首先可以执行`cowtest`：

可以看到直接fail了。在`cowtest.c`中可以看到，因为它会分配超过半数的物理内存，从而fork时没有足够的内存用来保存复制的内容。

2.1 修改`uvmcopy()`

我们首先需要修改`kernel/vm.c uvmcopy()`函数，这个函数是用于将父进程的页表和页表对应的内容复制到子进程的。

```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;
  char *mem;

  for(i = 0; i < sz; i += PGSIZE){
    // 获取每个虚拟地址对应的PTE
    if((pte = walk(old, i, 0)) == 0)
      panic("uvmcopy: pte should exist");
    if((*pte & PTE_V) == 0)
      panic("uvmcopy: page not present");
    // 转换为物理地址，并取出标志位
    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte);
    // 是否还能分配内存
    if((mem = kalloc()) == 0)
      goto err;
    // 将pa对应的内容复制到分配的地址mem
    memmove(mem, (char*)pa, PGSIZE);
    // mappages函数在新页表中为每个虚拟地址建立到分配的地址mem的映射，并设置控制位也相同
    if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
      kfree(mem);
      goto err;
    }
  }
  return 0;

 err:
  // 出错时，释放掉新页表中已经建立的PTE
  uvmunmap(new, 0, i / PGSIZE, 1);
```

```
  return -1;
}
```

按照提示，我们需要修改`uvmcopy()`函数，将**父进程的物理页映射到子进程页表上**,而不是为其分配新物理页。并且需要将子进程和父进程页表的`PTE_W`位都设为0。

首先，照旧用`walk()`函数在`old`页表中找寻对应的PTE，并获取对应的物理地址和标志位。之后，我们直接用`mappages()`函数在`new`页表上建立一系列映射，并将这些虚拟地址映射到之前获取的物理地址。

```
...
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);
    flags = PTE_FLAGS(*pte);
    // my edit
    /*if((mem = kalloc()) == 0)
      goto err;
    memmove(mem, (char*)pa, PGSIZE);
    if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
      kfree(mem);
      goto err;
    }*/
    if(mappages(new,i,PGSIZE,pa,flags) != 0)
    {
      //kfree(mem);
      goto err;
    }
    // edit ends
  }
...
```

最后，将子进程和父进程的`PTE_W`位置0。对于子进程，在`mappages()`前修改flags即可；对于父进程，通过`walk()`获取PTE后修改控制位，注意这个修改最好在`mappages`正确完成后再执行，保持与子进程页表一致。

```
  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);
    flags = PTE_FLAGS(*pte);
    // my edit
    /*if((mem = kalloc()) == 0)
      goto err;
    memmove(mem, (char*)pa, PGSIZE);
    if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
      kfree(mem);
      goto err;
    }*/
    flags = flags & ~PTE_W;

    if(mappages(new,i,PGSIZE,pa,flags) != 0)
    {
        //kfree(mem);
        goto err;
    }
    *pte = *pte & ~PTE_W;
    // edit ends
  }
```

2.2 缺页故障处理

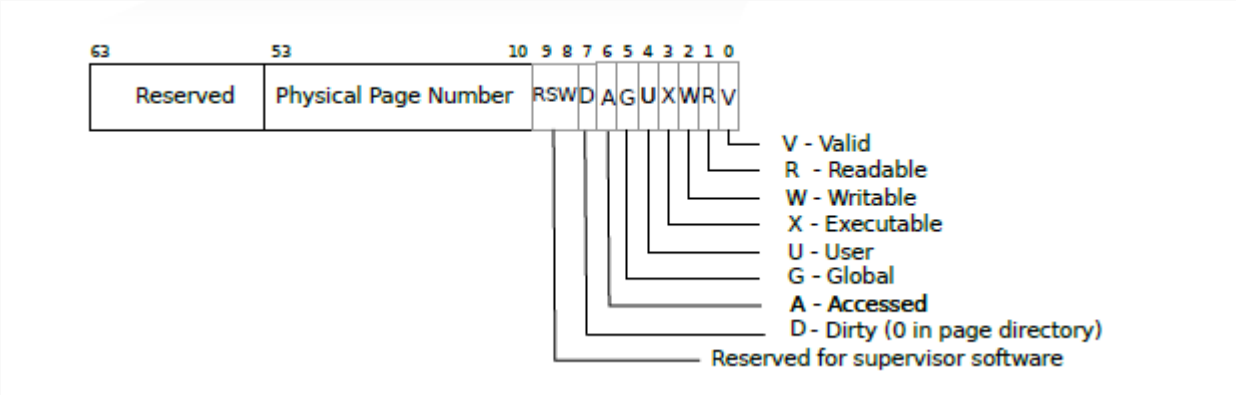
接下来，需要修改`usertrap()`函数以处理缺页故障。当在这些COW页上发生缺页故障时，分配新创建的物理页，将故障页的内容复制到新页上，并将新页的`PTE_W`位设置为1。原本就不可写的页应当保持可读并在父进程和子进程间共享，试图写这些页的进程将会被杀死。

2.2.1 区分COW页和普通页

首先要解决的问题是，**我们如何区分那些COW页和普通页**？如果按照2.1中的修改，那么COW页和普通页并没有任何区别，我们也无法区分在写这些不可写的页时，哪些应该分配新页，哪些应该直接拒绝。

按照提示，我们应该在每个PTE的标志位上做一些手脚，用于告诉我们哪些是COW页。xv6中有2位

保留的标志位称为RSW位，可以自由使用。我们将第9位记为COW位，用于标志该PTE对应的页是否为COW页。



在`kernel/riscv.h`中添加`PTE_COW`的定义：

```
// my edit
#define PTE_COW (1L << 8)
// edit ends
```

相应地，对2.1中`uvmcopy()`的代码做相应的修改。注意，**只有那些原本可写的页才被标记为COW页。**

```
    ...
    if(*pte & PTE_W)
      flags = (flags & ~PTE_W)| PTE_COW;
    if(mappages(new,i,PGSIZE,pa,flags) != 0)
    {
        //kfree(mem);
        goto err;
    }

    if(*pte & PTE_W)
      *pte = (*pte & ~PTE_W)| PTE_COW;
    ...
```

2.2.2 修改`usertrap()`函数

首先，缺页故障是一种**异常(Exception)**。根据xv6-book 4.2的描述:

> If the trap is a system call,usertrap calls syscall to handle it; if a device interrupt, devintr;
> **otherwise it's an exception,and the kernel kills the faulting process.**

查看`usertrap()`函数，与之对应的是这一段代码:

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

  ...
} 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());
  setkilled(p);
}
...
```

当`devintr()`的返回值为0时，对应的就是异常(Exception)，此时内核直接杀死该进程。

我们需要进一步处理，首先，可以看到上面系统调用对应的情况是`r_scause()==8`，即scause寄存器的值为8。我们只需要找到缺页(准确来说，是**store page faults**)对应的scause寄存器的值，就能进一步处理了。

根据riscv-privileged手册4.1.8 scause和表4.2，可知写页缺失对应的scause值应为15。

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2–4	*Reserved*
1	5	Supervisor timer interrupt
1	6–8	*Reserved*
1	9	Supervisor external interrupt
1	10–15	*Reserved*
1	≥16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10–11	*Reserved*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	≥64	*Reserved*

在确认了故障后，我们进一步需要检查PTE。根据riscv-privileged手册4.1.9 stval：

> If stval is written with a nonzero value when a breakpoint, address-misaligned, access-fault, or **page-fault exception occurs on an instruction fetch, load, or store, then stval will contain the faulting virtual address.**

stval寄存器保存发生故障时的故障(虚拟)地址。因此我们只需要读stval寄存器的内容，然后查进程页表找到对应的PTE检查标志位即可。这里最好直接将虚拟地址按页向下对齐，便于后续处理。

如果检查发现该页是COW页，按照`uvmcopy()`原本的方式进行创建页和复制页的工作，并用`mappages()`函数创建映射。注意按照提示，当没有足够内存分配，即`kalloc()==0`时，要杀死进程。这里定义了两个函数`isCOW()`和`cowcopy()`，分别用于检查va对应的页是否是COW页和执行cowcopy。因为后面的`copyout()`的修改和`usertrap()`的修改类似，所以写成独立的函数。注意这些自定义函数都应该在`kernel/defs.h`中声明。

```c
// my edit
int isCOW(uint64 va)
{
  pte_t* pte;
  struct proc* p = myproc();
  // 是否超过最大地址?
  if(va>=MAXVA)
    return 0;
  // 是否存在对应PTE?
  if((pte = walk(p->pagetable,va,0))==0)
    return 0;
  // 是否有效?
  if((*pte & PTE_V) == 0)
    return 0;
  // 是否是COW页?
  if((*pte & PTE_COW) == 0)
    return 0;

  return 1;
}

int cowcopy(uint64 va)
{
  struct proc* p = myproc();
  pte_t* pte = walk(p->pagetable,va,0);
  uint64 pa = PTE2PA(*pte);
  uint flags = PTE_FLAGS(*pte);

  char* mem;
  if((mem = kalloc()) == 0)
  {
    // 申请失败
    setkilled(p);
    return 0;
  }
  memmove(mem, (char*)pa, PGSIZE);
  flags = (flags | PTE_W) & ~PTE_COW;
  // 注意删掉旧映射，否则会触发panic:remapping
  uvmunmap(p->pagetable,va,1,0);
  if(mappages(p->pagetable, va, PGSIZE, (uint64)mem, flags) != 0)
  {
```

```
    kfree(mem);
    uvmunmap(p->pagetable, va, 1, 0);
    return 0;
  }
  return 1;
}
// edit ends
```

2.3 物理页引用

在Copy-on-Write下，要求仅当最后一个指向该物理页的引用消失后才释放该物理页。按照提示，可以在`kalloc()`每个页时设置**页引用计数(Page Reference Count)** 为1，并在每次`fork()`时将引用计数加1，每次一个进程撤销对该页的引用时将引用计数减1。`kfree()`只能在引用计数为0时才能将页加到freelist上。

按照提示，应该维护一个固定大小的整数数组，用于保存每个页的引用计数。可以通过将页的物理地址除以4096(因为PGSIZE=4096)获取下标，并且每个元素对应到`kinit()`中的freelist上的每个页的最高地址。

首先要决定这个数组要保存在哪里。显然不应该保存在进程结构体`proc`中，因为每个页会被多个进程(的页表)引用。考虑在`kernel/kalloc.c`中声明一个int数组`nref`作为引用计数数组，这样该数组能全局保持，大小可以设置为`(PHYSTOP-KERNBASE)/PGSIZE`，即32768。并定义获取下标的方式:(给定地址-KERNBASE)/PGSIZE。

```
#define PA2IDX(pa) (((uint64)pa - KERNBASE)/ PGSIZE)
int nref[(PHYSTOP-KERNBASE)/PGSIZE];
```

另外，就像kmem结构体一样，还需要一个**锁**用于避免并发问题，这是因为`nref`数组在所有进程间共享，涉及多个线程同时修改，如果没有锁很可能出现问题。考虑把`nref`和申请的锁`lock`放在同一个结构体中，记为`ref`。

```
// my edit
#define PA2IDX(pa) (((uint64)pa - KERNBASE)/ PGSIZE)
struct{
  struct spinlock lock;
  int nref[(PHYSTOP-KERNBASE)/PGSIZE];
}ref;
// edit ends
```

首先在`kinit()`中初始化锁。

```
void
kinit()
{
  initlock(&kmem.lock, "kmem");
  initlock(&ref.lock,"ref");
  freerange(end, (void*)PHYSTOP);
}
```

当调用`kalloc()`时，此时申请了一页新的内存，令该页对应的引用计数初始化为1。注意用锁保护对`nref`数组操作的部分。

```
void *
kalloc(void)
{
  struct run *r;

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

  acquire(&ref.lock);
  if(r)
    ref.nref[PA2IDX(r)]=1;
  release(&ref.lock);

  if(r)
    memset((char*)r, 5, PGSIZE); // fill with junk

  return (void*)r;
}
```

当调用`kfree()`时，此时进程试图释放某个物理页，我们首先检查该页的引用计数是否为1，如果为1，则申请释放的该页只有最后一个PTE引用他了，所以直接释放该页，并将引用计数重置为0;如果大于1，此时还有多个PTE引用该页，将该页的引用计数减1并直接返回。

```
void
kfree(void *pa)
{
  struct run *r;

  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
    panic("kfree");

  acquire(&ref.lock);
  if(ref.nref[PA2IDX(pa)] > 1)
  {
    ref.nref[PA2IDX(pa)]--;
    release(&ref.lock);
    return;
  }

  ref.nref[PA2IDX(pa)]=0;
  release(&ref.lock);
  // 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);
}
```

此外，提供一个接口函数用于增加引用计数，在其他函数中使用。

```
void addNref(uint64 pa)
{
  acquire(&ref.lock);
  ++ref.nref[PA2IDX(pa)];
  release(&ref.lock);
}
```

接下来修改之前的函数。对于`uvmcopy()`，这个函数是在`fork()`时调用的，用于将父进程的内容复制到子进程，所以在`uvmcopy()`中让引用计数加1。

```
...
    if(*pte & PTE_W)
      *pte = (*pte & ~PTE_W)| PTE_COW;
    addNref(pa);
...
```

对于`cowcopy()`，此时由于申请了新页并让要写的虚拟地址指向新页的物理地址，所以原本的旧业的引用计数应该减1。这里我们调用`kfree()`函数，当引用计数大于1时减1，等于1时就释放掉该页。

```
  ...
  if(mappages(p->pagetable, va, PGSIZE, (uint64)mem, flags) != 0)
  {
    kfree(mem);
    uvmunmap(p->pagetable, va, 1, 0);
    return 0;
  }
  kfree((void*)pa);
  ...
```

2.4 `copyout()`函数修改

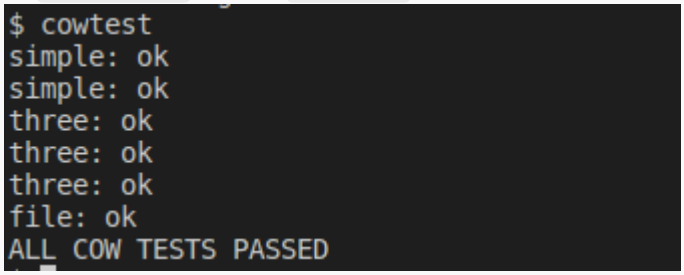
`copyout()`函数的功能是将指定物理地址的内容复制到虚拟地址va指定的物理地址，这也同样涉及到写页的问题。

如果va对应的页是COW页，那么按`usertrap()`中同样的方式处理；否则照常处理。注意通过`walkaddr()`获取物理地址这一步应该在完成`cowcopy()`之后，保证物理地址是最新的。

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

  while(len > 0)
  {
    va0 = PGROUNDDOWN(dstva);
    if(isCOW(va0))
      cowcopy(va0);

    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;
}
```

2.5 实验结果

`make qemu`后执行`cowtest`，结果如图:



执行`usertests -q`，结果如图:

```
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6574
            sepc=0x0000000000002410 stval=0x0000000000010eb0
OK
test textwrite: OK
test pgbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=6579
            sepc=0x0000000000005c5e stval=0x0000000000005c5e
usertrap(): unexpected scause 0x000000000000000c pid=6580
            sepc=0x0000000000005c5e stval=0x0000000000005c5e
OK
test sbrklast: OK
test sbrk8000: OK
test badarg: OK
ALL TESTS PASSED
```

退出qemu后执行`make grade`，结果如图：

```
== Test   file ==
  file: OK
== Test usertests ==
$ make qemu-gdb
(122.2s)
== 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
```

3. 实验问题

3.1 并发问题

最开始完全没考虑并发方面的问题，如果没有锁的话，直接执行`cowtest`和`usertests -q`的话其实也能通过。只是从原理上考虑，加锁会更好点。

3.2 usertests问题

在测试的时候碰到了两个问题：

1. 在测试`kernmem`的时候总是会跳出一堆`unexpected scause`，比如:

```
test kernmem: usertrap(): unexpected scause 0x000000000000000d pid=6488
              sepc=0x00000000000021f2 stval=0x0000000080000000
usertrap(): unexpected scause 0x000000000000000d pid=6489
              sepc=0x00000000000021f2 stval=0x000000008000c350
usertrap(): unexpected scause 0x000000000000000d pid=6490
              sepc=0x00000000000021f2 stval=0x00000000800186a0
usertrap(): unexpected scause 0x000000000000000d pid=6491
              sepc=0x00000000000021f2 stval=0x0000000800249f0
usertrap(): unexpected scause 0x000000000000000d pid=6492
              sepc=0x00000000000021f2 stval=0x0000000080030d40
```

我花了很多时间去修改，但结果没有任何区别。后面我看了下`usertests`的源码，`kernmem()`函数是这样写的：

```
// can we read the kernel's memory?
void
kernmem(char *s)
{
  char *a;
  int pid;

  for(a = (char*)(KERNBASE); a < (char*) (KERNBASE+2000000); a += 50000){
    ...
  }
    ...
}
```

然后又想到，scause=0x0d的话对应的不正是"load page fault"，即该页不能读吗，内核的页本来就不可读，所以这个就是正常的结果。

2. 在测试`MAXVAplus`的时候，总是会直接卡死，无法继续测试。我最开始以为是不是出现了死锁，所以调试看了一下，continue后总是停在这行：

```
void
acquire(struct spinlock *lk)
{
  ...
  __sync_synchronize();
  ...
}
```

这让我更加觉得是死锁的问题了，于是我把锁删了重新调试一遍，发现还是卡死在原来的地方。这次我考虑直接调试`usertests.c`中的`MAXVAplus()`函数，在这个函数处打上断点，`make qemu`后输入`usertests MAXVAplus`调试该函数，结果卡死在此处：

```
  int xstatus;
  // here
  wait(&xstatus);
  if(xstatus != -1)  // did kernel kill child?
    exit(1);
```

函数持续卡在`wait()`上，说明子进程没有被杀死。回头看了下`usertrap()`函数，可能是这里没有加`setkilled()`函数:

```
else if(r_scause()==15)
{
  // store page fault

  // make va page-aligned
  uint64 va = PGROUNDDOWN(r_stval());
  if(isCOW(va))
    cowcopy(va);
}
```

加了一句`else setkilled(p)`后就成功通过整个测试了，主要是加了一句`else if(r_scause()==15)`后，原本触发写页错误的进程是直接被杀死的，现在如果不另外处理就相当于对这种错误(不是Copy-on-Write的话)没有处理，`MAXVAplus`的测试就是针对写不合法地址的。

4. 实验感想

这次实验主要就实现一个Copy-on-Write机制，原理上其实并不很难理解，但实现起来第一是比较繁琐，很多细节要考虑到，比如标志位的情况，比如地址是否合法，又比如失败情况要怎么处理；第二是调试起来不太方便，很多时候完全不知道哪里出错了，只能一个一个去试。