Lab5-Copy-on-Write-Fork-for-XV6

Implement copy-on-write fork

实验目的

- 实现写时复制（Copy-on-Write）机制

  在 xv6 内核中实现写时复制（COW）fork机制。COW 允许父进程和子进程在初始 fork 时共享相同的物理内存页面，只有在其中一个进程尝试写入时才会创建页面的副本。这种机制可以大幅度减少 fork 操作中的内存开销。

- 掌握RISC-V汇编中函数参数传递、函数调用和内存访问的方法

实验步骤

1. 学习COW机制的原理：

   1. **传统的 fork() 实现**：

      - 在传统的 fork() 系统调用中，父进程的所有用户空间内存都会被复制到子进程中。这样做的缺点是：如果父进程占用的内存较大，复制操作会非常耗时。许多情况下，子进程在 fork() 之后会立刻执行 exec() 调用，从而丢弃刚刚复制的内存，导致大量不必要的内存复制和浪费。

   2. **COW 机制的优化**：

      - 为了提高效率，COW 机制推迟了内存复制的时机，只有在子进程实际需要写入内存时才进行物理内存页面的分配和复制。
      - 具体做法是：在 fork() 时，父进程和子进程共享相同的物理内存页面，而不是直接复制这些页面。共享的页面在页表中被标记为不可写（通过清除 PTE_W 标志），即父进程和子进程都只能读取这些页面，但不能写入。当父进程或子进程尝试写入某个共享页面时，CPU 会检测到该页面不可写，并产生一个页面错误。

   3. **页面错误处理**：

      - 当页面错误发生时，内核的页面错误处理程序会执行以下操作：为触发错误的进程分配一个新的物理页面。将原始共享页面的内容复制到新分配的页面中。修改触发错误进程的页表，将相关的页表项指向新的物理页面，并将该页表项标记为可写（设置 PTE_W 标志）。
      - 这样，只有在真正需要写入时才会进行内存复制，实现了内存的延迟复制和高效利用。

2. 在 `kernel/riscv.h` 中设置新的PTE标记位，标记一个页面是否采用了 `COW` 机制

```
#define PTE_COW (1L << 8) // 将RSW(第8位)作为COW标识位
```

3. 修改 uvmcopy() 函数，使其将父进程的物理页面映射到子进程，而不是为子进程分配新页面。对于设置了 PTE_W 的页面，在父子进程的 PTE 中清除 PTE_W（可写入标志）。

```
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");
    if (*pte & PTE_W) {
      *pte &= ~PTE_W;
      *pte |= PTE_COW;
    }
    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte);
    if(mappages(new, i, PGSIZE, pa, flags) != 0){
      goto err;
    }
    kref_lock();
    inc_refcnt(pa);
    kref_unlock();
  }
  return 0;

 err:
  // There is no need to release these pages, because the parent process will
still use them.
  uvmunmap(new, 0, i / PGSIZE, 0);
  return -1;
} / PGSIZE, 1);
    return -1;
}
```

4. 在 `kernel/kalloc.c` 中新建用于维护物理页的引用数的结构体

```
static struct {
  struct spinlock lock;
  int cnt[REF_CNT_IDX(PHYSTOP)];
} kref;
```

5. 修改kinit(),初始化物理页引用计数锁

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

6. 修改`kalloc`函数，将引用计数初始化为1

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

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

  if (r) {
    set_refcnt((uint64)r, 1);
    memset((char*)r, 5, PGSIZE); // fill with junk
  }
  return (void*)r;
}
```

7. 修改`kfree`函数，每次进程从页表中删除页面时，减少引用计数。只有当引用计数为零时，`kfree()`才将页面真正放回空闲列表。

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

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

  acquire(&kref.lock);
  if (dec_refcnt((uint64)pa) > 0) {
    release(&kref.lock);
    return;
  }
  release(&kref.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);
}
```

8. 修改 `usertrap` 函数以识别页面错误。当在最初可写的 COW 页面上发生写页面错误时（错误码为15），分配一个新页面。

```
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) {  // Environment call from U-mode
    // system call

    if(killed(p))
      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 sepc, scause, and sstatus,
    // so enable only now that we're done with those registers.
    intr_on();

    syscall();
  } else if(r_scause() == 15) {  // Store/AMO Page Fault
    if (killed(p))
      exit(-1);

    if (cowtrap(r_stval()) < 0)
      setkilled(p);
  } 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);
  }

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

  // give up the CPU if this is a timer interrupt.
```

```
  if(which_dev == 2)
    yield();

  usertrapret();
}
```

9. 修改 copyout() 函数，使其在遇到 COW 页面时使用与页面错误相同的机制。

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

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

    pte_t* pte = walk(pagetable, va0, 0);
    if (pte == 0)
      return -1;
    if (*pte & PTE_COW) {
      if (cowtrap(va0) < 0)
        return -1;
      pa0 = walkaddr(pagetable, va0);
    }

    memmove((void *)(pa0 + (dstva - va0)), src, n);

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

10. 测试

运行`make qemu`后执行`cowtest`

```
$ cowtest
simple: ok
simple: ok
three: ok
three: ok
three: ok
file: ok
ALL COW TESTS PASSED
```

执行 `usertests -q`

```
usertrap(): unexpected scause 0x000000000000000d pid=6527
            sepc=0x00000000000021f2 stval=0x00000000801dc130
OK
test MAXVAplus: OK
test sbrkfail: usertrap(): unexpected scause 0x000000000000000d pid=6566
            sepc=0x0000000000004994 stval=0x0000000000013000
OK
test sbrkarg: OK
test validatetest: OK
test bsstest: OK
test bigargtest: OK
test argptest: OK
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
```

运行 `make grade`

```
== Test running cowtest ==
$ make qemu-gdb
(12.9s)
== Test   simple ==
  simple: OK
== Test   three ==
  three: OK
== Test   file ==
  file: OK
== Test usertests ==
$ make qemu-gdb
(66.0s)
== 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
```

实验中遇到的问题和解决办法

usertests中增加了一个很难通过的测试函数textwrite，`textwrite` 测试的主要目的是验证写时复制（COW）机制是否能够正确处理以下几种情况：

1. **文本段（只读段）的写入尝试**：文本段通常是只读的，尝试写入应该导致页面错误并终止进程。
2. **COW 页面的处理**：当一个进程尝试写入共享的 COW 页面时，应该触发页面错误，内核分配新的物理页面，并将旧页面的数据复制到新页面中。

解决办法是`cowtrap` 函数首先检查页面是否是 COW 页面。

如果页面不是 COW 页面，即文本段页面，它将返回错误。在 `usertrap` 函数中，当 `cowtrap` 返回错误时，进程会被标记为 `killed`，从而终止进程。这确保了尝试写入只读的文本段时，进程

会被正确终止。

当页面是 COW 页面时，`cowtrap` 函数会分配一个新的物理页面，将旧页面的数据复制到新页面，并更新页表项以指向新页面，同时设置适当的标志位。这确保了在 COW 页面的写入尝试时，内核能够正确地进行页面复制和更新，从而支持写时复制机制。

实验心得

在这次实验中，最让我感到挑战的是如何正确实现写时复制（COW）机制，特别是处理文本段的写入操作。在调试 `textwrite` 测试时，我遇到了许多困难，这个测试新增了对 COW 机制的特定检查，直接暴露了我们实现中的一些潜在问题。起初，我按照书中的描述实现了基本的 COW 机制，但在运行 `textwrite` 测试时总是失败。经过多次尝试和阅读相关资料，我意识到问题的关键在于文本段的处理。文本段默认是只读的，并没有 `PTE_W` 标志，因此在进行 COW 操作时，如果不仔细区分这些段，就会引发错误。

最终，通过增加一个专门的函数 `cowtrap` 来处理 COW 页面错误。这个函数通过检查页表项的标志位，确保只对 COW 页面进行处理，而不影响文本段。这个改进使得 `textwrite` 测试能够正确识别和处理文本段的写入尝试，进而保证进程在不合法写入时被正确终止。

通过这个过程，我深刻体会到在系统级编程中，细节决定成败。一个小小的标志位检查错误，就可能导致整个机制的失败。