Lab - Copy on Write 目录

1. Implement copy-on-write fork (hard)

1.1 实验目的

- 实现 `fork()` 的 `Copy on Write (COW)` 机制
- 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

1.2 实验步骤

1. 了解 `COW` 机制的原理。xv6中的 `fork()` 系统调用将父进程的所有用户空间内存复制到子进程中。如果父进程较大，则复制可能需要很长时间。而且可能因为不必要的复制造成大量浪费（例如子进程中的 `fork()` 后紧跟 `exec()` 将导致子进程丢弃复制的内存）。因此，可以采取更加高效的策略，即推迟复制的时机，只有子进程实际需要物理内存拷贝时再进行分配和复制物理内存页面。默认情况下，父进程和子进程中的所有 `PTE` 都被标记为不可写。当任一进程试图写入其中一个 `COW` 页时，CPU将强制产生页面错误。内核页面错误处理程序检测到这种情况将为出错进程分配一页物理内存，将原始页复制到新页中，并修改出错进程中的相关 `PTE` 指向新的页面，将PTE标记为可写。这样就完成了复制的延迟，也即写时拷贝。
2. 阅读 `user/cowtest.c` 和 `user/usertests.c`，了解测试和调用时机、逻辑。
3. 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，鉴于第8、9位为RSW（软件保留）位，故使用第8位：

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

4. 修改 `kernel/vm.c` 中的 `uvmcopy()` 函数，将父进程的物理页映射到子进程，并在子进程和父进程的 `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");

        pa = PTE2PA(*pte);

        *pte = (*pte & ~PTE_W) | PTE_COW; // 将所有的pte的写权限都设置为0，
PTE_COW设置为1

        flags = PTE_FLAGS(*pte);

        // 直接映射到子进程
        if (mappages(new, i, PGSIZE, pa, flags) != 0) {
            goto err;
        }

        // 索引计数加1
        if (AddPGRefCount((void *)pa)) {
```

```
            goto err;
        }
    }
    return 0;

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

5. 需要一个结构体来维护物理页的引用数，定义在`kernel/kalloc.c`下：

```
struct refcnt {
    struct spinlock lock;
    int PGCount[PHYSTOP / PGSIZE]; // (最大页数 = 地址空间大小/页大小)
} PGRefCount;
```

6. 其中有函数`AddPGRefCount()`，为页表引用计数+1，详情如下：

```
int AddPGRefCount(void *pa)
{
    if (((uint64)pa % PGSIZE)) {
        return -1;
    }
    if ((char *)pa < end || (uint64)pa >= PHYSTOP) {
        return -1;
    }

    acquire(&PGRefCount.lock);
    PGRefCount.PGCount[(uint64)pa / PGSIZE]++;
    release(&PGRefCount.lock);
    return 0;
}
```

7. 在`kalloc.c`中修改`kinit()`，在`kinit()`中对`PGRefCount`的锁进行初始化：

```
void kinit()
{
    initlock(&kmem.lock, "kmem");
    initlock(&PGRefCount.lock, "PGRefCount"); // 初始化PGRefCount锁
    freerange(end, (void *)PHYSTOP);
}
```

8. 对`kalloc()`函数进行修改，要使在缺省情况下`PGCount`的值为1：

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

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

```
    // pgcount初值赋为1
    if (r) {
        acquire(&PGRefCount.lock);
        PGRefCount.PGCount[(uint64)r / PGSIZE] = 1;
        release(&PGRefCount.lock);
    }

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

9. 对`kfree()`函数进行修改，每次发出命令要释放物理页时，由于子进程并不真正复制了物理页而是映射到父进程的物理页，因此要把引用数-1，只有当引用数为0的时候才需真正释放：

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

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

    acquire(&PGRefCount.lock);
    PGRefCount.PGCount[(uint64)pa / PGSIZE]--;

    if (PGRefCount.PGCount[(uint64)pa / PGSIZE] != 0) {
        release(&PGRefCount.lock);
        return;
    }
    release(&PGRefCount.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);
}
```

10. 由于一个物理页的生命周期从kinit()开始到kfree() 结束，但期间还有调用一个函数freerange()，而在该函数中，需要设所有物理页引用数为1，否则在减小时会发生溢出的错误：

```
void freerange(void *pa_start, void *pa_end)
{
    char *p;
    p = (char *)PGROUNDUP((uint64)pa_start);
    for (; p + PGSIZE <= (char *)pa_end; p += PGSIZE) {
        // 所有页的PGCount值均设为1，使其在后续过沉重可以正常进行释放
        PGRefCount.PGCount[(uint64)p / PGSIZE] = 1;
        kfree(p);
    }
}
```

11. 考虑到要获取引用数，添加函数`GetPGRefCount()`：

```
int GetPGRefCount(void *pa)
{
    return PGRefCount.PGCount[(uint64)pa / PGSIZE];
}
```

12. 接下来需要在`kernel/trap.c`下的`usertrap()`中添加中断的处理，对于`COW`机制，一旦发生了这样的缺页中断，应当为其分配物理页并复制：

```
void usertrap(void)
{
    ...
    if (r_scause() == 8) {
        // 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();
    }
    // 异常代码15：Fault Fetch（缺页错误）
    // 异常代码13：Misaligned Fetch（对齐错误）
    else if (r_scause() == 13 || r_scause() == 15) {
        uint64 va = r_stval();
        if (va >= p->sz || isCOWPG(p->pagetable, va) != 1 || allocCOWPG(p-
>pagetable, va) == 0)
            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());
        setkilled(p);
    }

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

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

    usertrapret();
}
```

其中使用到了两个函数，第一个函数为`isCOWPG()`，用于判断是否为`COW`页表：

```
int isCOWPG(pagetable_t pg, uint64 va)
{
    if (va > MAXVA)
        return -1;
    pte_t *pte = walk(pg, va, 0);
    if (pte == 0)
        return 0;
    if ((*pte & PTE_V) == 0)
        return 0;
    if ((*pte & PTE_COW))
        return 1;
    return 0;
}
```

第二个函数为`allocCOWPG()`，用于分配物理页并复制内容：

```
void *allocCOWPG(pagetable_t pg, uint64 va)
{
    va = PGROUNDDOWN(va);
    if (va % PGSIZE != 0 || va > MAXVA)
        return 0;

    uint64 pa = walkaddr(pg, va);
    if (pa == 0)
        return 0;

    pte_t *pte = walk(pg, va, 0);
    if (pte == 0)
        return 0;

    int count = GetPGRefCount((void *)pa);
    if (count == 1) {
        // 只有一个进程在使用当前页，设置为可写并将COW标志位复位
        *pte = (*pte & ~PTE_COW) | PTE_W;
        return (void *)pa;
    }
    else {
        // 有多个进程在使用当前页，需要分配新的物理页
        char *mem = kalloc();
        if (mem == 0)
            return 0;

        memmove(mem, (char *)pa, PGSIZE);

        // 清除有效位PTE_V，否则在mappagges中会出错
        *pte = (*pte) & ~PTE_V;

        if (mappages(pg, va, PGSIZE, (uint64)mem, (PTE_FLAGS(*pte) &
~PTE_COW) | PTE_W) != 0) {
            kfree(mem);
            *pte = (*pte) | PTE_V;
            return 0;
```

```
        }

        kfree((void *)PGROUNDDOWN(pa));

        return (void *)mem;
    }
}
```

13. 还需在kernel/vm.c中修改copyout()函数，使在传递时能正确传出物理页：

```
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);

        // 若是COW页，则要分配新的物理页再复制
        if (isCOWPG(pagetable, va0) == 1) {
            pa0 = (uint64)allocCOWPG(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;
}
```

14. 运行`cowtest`，结果如下：

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

15. 运行`usertests`，部分结果如下：

```
$ usertests
usertests starting
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
```

16. 运行`grade`程序，最终结果如下：

```
== Test running cowtest ==
$ make qemu-gdb
(7.8s)
== Test   simple ==
  simple: OK
== Test   three ==
  three: OK
== Test   file ==
  file: OK
== Test usertests ==
$ make qemu-gdb
(53.2s)
== Test   usertests: copyin ==
  usertests: copyin: OK
== Test   usertests: copyout ==
  usertests: copyout: OK
== Test   usertests: all tests ==
  usertests: all tests: OK
```

1.3 实验中遇到的问题和解决方法

- **问题**：在运行usertests时出现如下报错信息：

  ```
  $ usertests
  usertests starting
  test copyin: OK
  test copyout: panic: walk
  ```

  解决方法：定位`panic: walk`的位置，发现其判断条件是`if (va >= MAXVA)`，也即意味着发生了虚拟地址越界的错误，再定位添加的函数`isCOWPG()`，发现并没有对越界进行处理，反而是作为正常值返回，因此导致错误。添加条件判断即可。

- **问题**：在运行usertests时出现如下报错信息：

  ```
  test stacktest: panic: allocCOWPG: walkaddr
  ```

  解决方法：这是自定义的`panic`语句，并没有返回一个值，因此导致崩溃。

- **问题**：难以预估的一些错误

解决方法：定位后发现错误发生在`kfree()`中，一开始的引用数为0，但在`kfree()`中执行减1命令后会发生溢出，从而变为`INTMAX-1`，无法释放物理页。

1.4 实验心得

本次实验中我实现了`copy-on-write (COW) fork()`功能，延迟了分配和复制物理内存页面的过程，从而提高了`fork()`的效率。在实验过程中，我学会了如何在xv6内核中实现`COW`技术，深刻理解了`COW`技术的机制名表了如何处理页面错误中断和正确管理物理页面引用计数。我认识到采用该类延迟的方法是一种有效提高性能的方法，核心思想为有需要时再分配。这让我对操作系统的理解更加深刻。