OS_lab5_report

21302010042
侯斌洋

运行结果

1. cowtest

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

2. usertests -q

```
$ usertests -q
usertests starting
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test rwsbrk: OK
...
test sbrklast: OK
test sbrk8000: OK
test badarg: OK
ALL TESTS PASSED
```

3. make grade

```
== Test running cowtest ==
$ make qemu-gdb
(9.5s)
== Test   simple ==
simple: OK
== Test   three ==
three: OK
== Test   file ==
file: OK
== Test usertests ==
$ make qemu-gdb
(99.8s)
== 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
```

实验思路

由于要执行写时复制，故在刚fork时是不分配页的，而是多个页表项指向同一个页并标记该页为COW页。当对一个COW页进行读操作时，无需分配新的页，但当对其进行写的时候，就要对目前写的进程分配新的页，将当前进程的页表项指向该新的页。因此在kalloc.c，即管理页的分配的文件中就要加上一个新的结构mem_ref_num用于记录一个页的引用页表项数。当进行kalloc时设置该引用数为1，在kfree时判断引用数，若大于1则直接去掉该引用即可，不实际释放页，引用数等于1时再实际释放该页。

fork中进程的复制实际是通过uvmcopy进行的，故在此过程中可直接将子进程的pte指向父进程中的页，暂时不分配新的页，并设置两个进程中的页均为只读的且带有COW标记来实现写时复制在fork中的机制。

由于在uvmcopy中对实际可写的页设置为只读的，故会引发由于COW机制导致的缺页错误，接下来就要识别这种类型的缺页错误并进行处理。

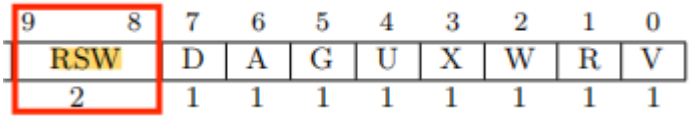
该COW缺页错误发生在usertrap中和copyout中。在usertrap中，可通过scause获取中断类型检测到写时缺页错误:

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*

之后可通过stval获得产生该写时缺页错误的地址，然后再通过该地址找到pte并通过pte检测是否为COW缺页错误并进行处理。若确实为COW错误，若该页引用数大于1，应为该进程分配新的页，并将该进程的页表项指向该新的页。若该页引用数等于1，说明是之前设置了COW标志的原因，将pte中的标志位改为正常即可。在copyout中，通过直接检测目的地址dstva的pte来判断是否为COW缺页错误，若确实为COW缺页错误，与上面的处理过程相同，然后重新获取对应的物理地址即可。

实验过程及代码

1. 查询 ricsv-privileged 手册，得到 PTE_RSW 在第 8 和 9 位，在 riscv.h 中添加如下内容标记 cow 页。



9 8	7	6	5	4	3	2	1	0
RSW	D	A	G	U	X	W	R	V
2	1	1	1	1	1	1	1	1

```
//ricsv.h
#define PTE_COW (1L << 8)
```

2. 在kalloc.c中增加以下结构记录一个页的引用页表项数，并增加了一个锁防止竞争。

```
//kalloc.c
struct
{
    struct spinlock lock;
    int num[(PHYSTOP - KERNBASE) / PGSIZE];
} mem_ref_num;
```

增加了以下函数方便获取和设置num

```
//kalloc.c
int get_mem_ref(uint64 pa)
{
    return mem_ref_num.num[(pa - KERNBASE) / PGSIZE];
}
void set_mem_ref(uint64 pa, int n)
{
    mem_ref_num.num[(pa - KERNBASE) / PGSIZE] = n;
}
```

3. 修改 uvmcopy为如下代码直接将子进程的pte指向父进程中的页，暂时不分配新的页，并设置两个进程中的页均为只读的且带有COW标记

```c
//vm.c
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");

        // set pte_cow and remove pte_w
        *pte = (*pte) | PTE_COW;
        *pte = (*pte) & (~PTE_W);

        pa = PTE2PA(*pte);
        flags = PTE_FLAGS(*pte);
        if (mappages(new, i, PGSIZE, pa, flags) != 0)
        {
            goto err;
        }

        // set mem_ref_num
        acquire(&mem_ref_num.lock);
        set_mem_ref(pa, get_mem_ref(pa) + 1);
        release(&mem_ref_num.lock);
    }
    return 0;

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

4. 在usertrap中添加一个判断条件用于处理写时缺页错误，并识别出cowpage并处理：

```
//trap.c
//......
else if ((r_scause() == 15) && iscowpage(r_stval()))
{
    // cow page fault
    if ((r_stval() < PGSIZE) || (cowcopy(r_stval()) == -1))
    {
        p->killed = 1;
    }
}
else if ((which_dev = devintr()) != 0)
//......
```

下面为识别cowpage的函数和处理cow错误的函数：

```c
//trap.c
int iscowpage(uint64 va)
{
    struct proc *p = myproc();
    va = PGROUNDDOWN((uint64)va);
    if (va >= MAXVA)
        return 0;
    pte_t *pte = walk(p->pagetable, va, 0);
    if (pte == 0)
        return 0;
    if ((va < p->sz) && (*pte & PTE_COW) && (*pte & PTE_V))
        return 1;
    else
        return 0;
}

int cowcopy(uint64 va)
{
    // find pte
    va = PGROUNDDOWN(va);
    pagetable_t pagetable = myproc()->pagetable;

    // avoid panic: walk in  usertests
    if (va >= MAXVA)
        return 0; // do nothing

    pte_t *pte = walk(pagetable, va, 0);
    uint64 pa = PTE2PA(*pte);

    // check cow fault
    if (pte == 0 || !((*pte) & PTE_V))
        return 0; // do nothing
    if (!((*pte) & PTE_COW))
        return 0; // do nothing

    // check mem_ref_num
    acquire(&mem_ref_num.lock);
    if (get_mem_ref(pa) > 1) // num > 1, alloc
    {
        // alloc new page and copy
        char *mem = kalloc();
        if (mem == 0)
            return -1; // no free
```

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

            // set flag and map
            *pte = (*pte) & (~PTE_V);
            uint64 flag = PTE_FLAGS(*pte);
            flag = flag | PTE_W;
            flag = flag & (~PTE_COW);
            if (mappages(pagetable, va, PGSIZE, (uint64)mem, flag) != 0)
            {
                kfree(mem);
                release(&mem_ref_num.lock);
                return -1;
            }
            set_mem_ref((uint64)pa, get_mem_ref((uint64)pa) - 1);
        }
        else // num == 1, unset cow and set w
        {
            *pte = (*pte) & (~PTE_COW);
            *pte = (*pte) | PTE_W;
        }
        release(&mem_ref_num.lock);
        return 0;
    }
```

5. 在copyout中识别出cowpage并处理：

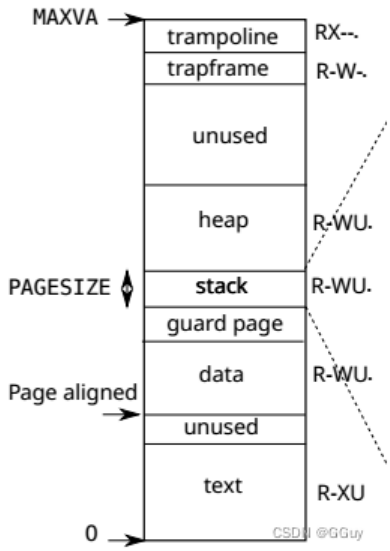
```c
//vm.c
int copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
    uint64 n, va0, pa0;

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

        //handle cow page fault
        if(iscowpage(va0)){
            cowcopy(va0);
            pa0 = walkaddr(pagetable, va0);
        }

        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;
}
```

遇到的问题

usertests中的 textwrite 不通过，找不到原因。最后后去网上搜索找到了答案。以下为其解释：

- **usertests中增加了一个难缠的测试函数textwrite，这个也是去年lab没有的，困扰了我很久，然后发现其实这个函数是新增了一个对cow的bug的检查**

  子进程copy完父进程的页表后，会将每一页的pte的pte_W置0，pte_C置1，我们就可以通过判断pte_W和pte_C判断该页是不是cow页



  那么随之也会出现一个bug,每个用户进程的低地址段都会用于储存代码(即text区域)，根据book描述，这一段本来也没有pte_W标志。那么如果我们对其进行COW操作就会引发一系列的错误

  所以在usertrap中对这一bug直接拦截

```
1 if(r_stval() < PGSIZE)
2     p->killed = 1;
```

因此在usertrap中以下判断条件左侧会有一个 r_stval() < PGSIZE 的先决条件。

```
if ((r_stval() < PGSIZE) || (cowcopy(r_stval()) == -1))
{
    p->killed = 1;
}
```

在加上该条件后，textwrite可以正常通过。