

指导书

实验任务

基于完成的 COW 实验，完成 SOW，即在 fork 时把页面复制到 swap 区域，在写页面时把 swap 区域中的该页面换回到内存。

提示

使用数组模拟在磁盘中的 swap 区域

实验过程

Cow 部分

首先在 riscv.h 中添加 pte 标志位 PTE_C 来标志通过 cow 的 pte
然后在 kalloc.c 里设置一个数组用来存储每个物理页面的引用计数，因为有并

行操作，所以加一个自旋锁

```
struct {  
    struct spinlock lock;  
    uint cnt[(PHYSTOP - KERNBASE) / PGSIZE];  
} refcnt;
```

再为这个数组设置一些函数方便其他文件内对其操作

再修改 fork，fork 调用了 uvmcopy，所以直接修改 uvmcopy。

修改*pte，去掉写权限并添加 COW 标志

```
*pte &= ~PTE_W;  
*pte |= PTE_C;再修改 mappages，把 remap 删除。然后继续在 uvmcopy 中调用函数  
refinc(pa, 1);  
void  
refinc(uint64 pa, int n){  
    acquire(&refcnt.lock);  
    refcnt.cnt[searchr(pa)] += n;  
    release(&refcnt.lock);  
}
```

把该页面数组计数+1

然后修改 kfree 函数

```
if(refcnt.cnt[index] > 1){
refcnt.cnt[index] -= 1;
release(&refcnt.lock);
return;
}
```

在页面计数大于 1 的时候只把计数-1，不进行 free，而在页面计数等于 1 的时候进行 free，

释放掉页面

修改 kalloc 函数

```
if(r)
refinc((uint64)r, 1);
```

在分配页面时把计数+1

修改 trap 中的 usertrap 函数，并且同步修改 copyout 函数

```
else if(r_scause() == 15){
uint64 va = r_stval();
if (va >= MAXVA) p->killed = 1;
if(cow(va) == -1) p->killed = 1;
}
```

当写页异常时，进行 cow 操作，具体 cow 函数如下

```
int
cow(uint64 va){
va = PGROUNDDOWN(va);
pagetable_t p = myproc()->pagetable;pte_t* pte = walk(p, va, 0);
uint64 pa = PTE2PA(*pte);
uint flags = PTE_FLAGS(*pte);
if(!(flags & PTE_C)){
return -1;
}
acqr();
uint ref = refget(pa);
if(ref > 1){
char* mem = kalloc_nolock();
if (!mem) {
relr();
return -1;
}
memmove(mem, (char*)pa, PGSIZE);
if(mappages(p, va, PGSIZE, (uint64)mem, (flags & (~PTE_C)) | PTE_W) !=
0){
kfree(mem);
relr();
return -1;
}
```

```

}
refset(pa, ref - 1);
}else{
*pte = ((*pte) & (~PTE_C)) | PTE_W;
}
relr();
return 0;
}

```

这个函数首先判断了这个页面是不是 `cow` 页面，如果不是直接返回错误，如果是，继续执行。

如果引用计数大于 1，那么此时需要复制页面确保写操作能够正常执行。首先通过 `kalloc` 开辟内存空间，然后把页面内容复制到内存中，再把新页面映射到这块内存，并更新 `pte` 标志后减少原始页面引用次数。如果引用计数等于 1，直接更新 `pte` 标志，把这个页面改为可写并且取消 `cow`。

Sow 部分

首先创建 `swap.c` 和 `swap.h`。

在 `swap.h` 中定义如下内容

```

#define SWAP_SIZE (64*1024*1024)
#define SWAP_PAGES (SWAP_SIZE/4096)

```

```

extern char swap_area[SWAP_SIZE];
extern short swap_bool[SWAP_PAGES];

```

```

struct swap_text{
    uint64 pa1;
    int index;
};

```

在 `swap.c` 中初始化

```

char swap_area[SWAP_SIZE];
short swap_bool[SWAP_PAGES];
struct swap_text swapt[SWAP_PAGES];

```

```

void swap_init() {
    for(int i = 0; i < SWAP_PAGES; i++) {
        swap_bool[i] = 0;
        swapt[i].pa1 = 0;
    }
}

```

```

        swapt[i].index = -1;
    }
}

```

并把 init 函数加入到 main.c 中启动

接下来创建 swap 函数

```

int swap_out(uint64 pa, uint64 index) {
    if(index >= SWAP_PAGES) {
        return -1; // 没有足够的 swap 空间
    }

    memmove(&swap_area[index * PGSIZE], (char*)pa, PGSIZE);

    swapt[index].pa1 = pa;
    swapt[index].index = index;
    swap_bool[index] = 1;

    return 0;
}

```

在 vm.c 的 uvmcopy 中做出修改

```

if(swap_out(pa, swap_index) != 0) {
    goto err;
}

while(swap_index < SWAP_PAGES && swap_bool[swap_index]) {
    swap_index++;
}

if(swap_index > SWAP_PAGES) goto err;

```

在 cow 函数中作出修改

```

for(i = 0; i < SWAP_PAGES; i++)
{
    if (pa == swapt[i].pa1)
    {
        index = swapt[i].index;
    }
}

```

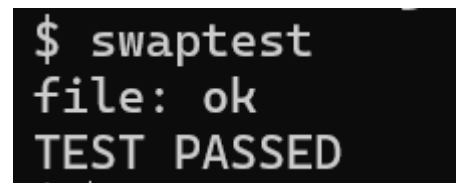
```
        break;
    }
}

memmove(mem, (char*)&swap_area[index * PGSIZE], PGSIZE);
```

实验测试

make qemu

swaptest



```
$ swaptest
file: ok
TEST PASSED
```

文件中标有//answer 的部分为答案