

中期报告——改进XV6内存管理方式

张昱 2020012386

幸若凡 2020012365

张凯伦 2020012387

王集 2020012389

一、题目调研

本组成员对XV6操作系统现有的内存管理方式进行了较为细致的学习。

我们了解到：

- XV6采用页式存储管理，单一页面大小为4096字节，从虚拟地址到物理地址的映射通过三级树形结构PTE完成，详细的映射方式见kernel/vm.c/walk()，详细的PTE构建方式见kernel/vm.c/mappages()。每一进程拥有自己的页表，内核也有自己的页表。
- 每一PTE的后10位用于进行标记，这些标记告诉我们当前PTE对应页面的状态（包括是否有效PTE_V、是否可读PTE_R/写PTE_W/执行PTE_X，是否对用户态可见PTE_U等等）。
- XV6的内存分配方式较为简易，使用一链表（freelist）存储当前空闲的页面，并在用户请求（kalloc）时逐一进行分配。使用后释放（kfree）的页面会重新添加回链表。
- 通过在栈下方设置guard page（将其PTE_V设为0），可以检测栈的溢出。
- 内核的页表多采用直接映射，除去trampoline page与进程的内核栈。Trampoline page在内核与所有进程中的虚拟地址相同，从而使进程可以与内核通信（trap）。
- 当出现异常（如读取PTE_V为0的页面），内核将触发panic（但实际上我们可以对此进行处理，这非常重要，我们大量的扩展功能都将基于此实现）。

我们学习的内容还有很多，难以在此详尽列出。

二、预期功能目标

1. 增加内存懒惰分配（lazy allocation）功能。目前XV6会在进程申请内存时立刻为其分配相应数量的物理页面，但实际上进程可能并不会使用这么多物理内存。通过内存懒惰分配，我们可以在进程真正使用内存时再为其分配物理内存，从而节省物理内存与时间。

已实现。

2. 增加进程写时复制（copy-on-write fork）功能。目前XV6的fork()会立刻将当前进程的所有内容复制一份，但实际上我们调用fork()多是在exec()中，exec()中执行完fork()又会立刻将复制的全部内容释放，从而造成浪费。通过进程写时复制，我们可以在fork()后需要写入时再真正拷贝副本，从而在exec()等不需要写入的场景中节省大量时间。

已实现。

3. 更改进程用户栈的可用大小。目前XV6进程的用户栈仅有一个页面，且不可扩张。我们希望能够实现更大的用户栈，从而使系统支持需要更多栈内存的进程的运行。

预计在2周内实现。

4. 改进空闲内存的管理方式。目前XV6“粗糙”地将所有空闲页面存在一个链表中。我们希望尝试使用更高效的数据结构完成对空闲页面的管理。

预计在3周内实现。

5. 增加内存映射文件（mmap）功能。该功能可以将文件加载到内存中，通过内存相关指令来直接操作文件，从而加速文件相关操作。

预计在5周内实现。

6. 增加请求调页功能。该功能可以实现不在运行某进程时将其所有页面载入物理内存，而是只载入部分的页面，并在需要时载入相应的页面，从而节省内存。

预计在6周内实现。

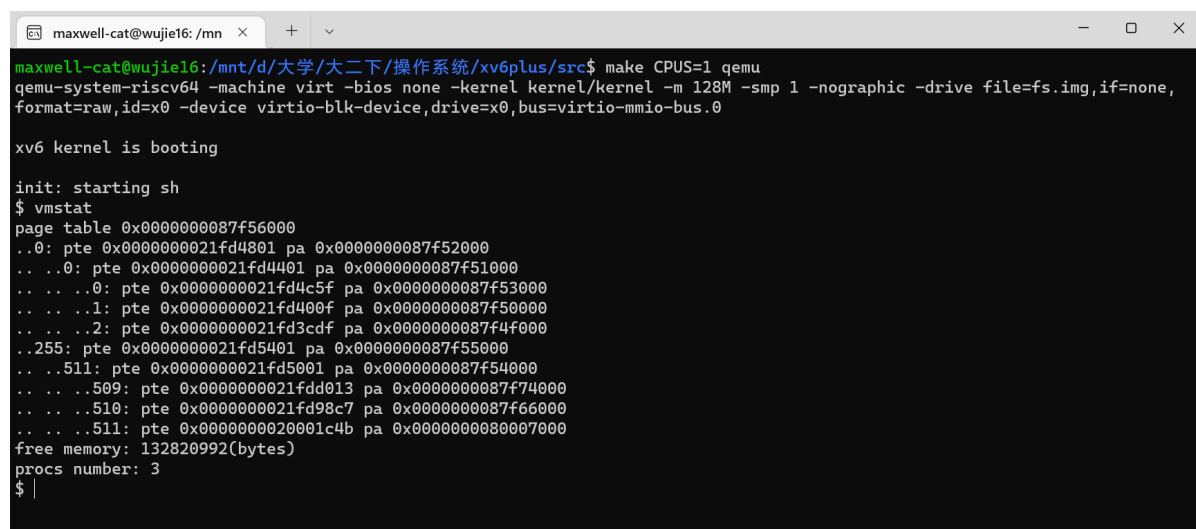
7. 增加页面置换功能。该功能可以实现物理页面不足时，将“最不重要”的旧有页面用新的所需页面置换，从而使进程能够正常运行。

预计在6周内实现。

三、已实现功能目标

目前已实现的功能有“内存懒惰分配”与“进程写时复制”。

为方便调试，我们首先自己实现了一个用户程序“vmstat”，可以打印XV6目前页表、内存、进程的基本状况。



```
maxwell-cat@wujie16: /mnt/d/大学/大二下/操作系统/xv6plus/src$ make CPUS=1 qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 1 -nographic -drive file=fs.img,if=none,
format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

init: starting sh
$ vmstat
page table 0x0000000087f56000
..0: pte 0x0000000021fd4801 pa 0x0000000087f52000
.. ..0: pte 0x0000000021fd4401 pa 0x0000000087f51000
.. ..0: pte 0x0000000021fd4c5f pa 0x0000000087f53000
.. ..1: pte 0x0000000021fd400f pa 0x0000000087f50000
.. ..2: pte 0x0000000021fd3cdf pa 0x0000000087f4f000
..255: pte 0x0000000021fd5401 pa 0x0000000087f55000
.. ..511: pte 0x0000000021fd5001 pa 0x0000000087f54000
.. ..509: pte 0x0000000021fdd013 pa 0x0000000087f74000
.. ..510: pte 0x0000000021fd98c7 pa 0x0000000087f66000
.. ..511: pte 0x0000000020001c4b pa 0x0000000080007000
free memory: 132820992(bytes)
procs number: 3
$ |
```

以下是“内存懒惰分配”的实现步骤：

1. 在进程调用sbrk()申请更改内存大小时，若是要缩小内存，则正常执行；若是要增大内存，则不真正分配页面，而是仅告知进程其内存大小增加了。

kernel/sysproc.c:

```
uint64
sys_sbrk(void)
{
    int addr;
    int n;

    if(argint(0, &n) < 0)
        return -1;
    addr = myproc()->sz;
    if(n < 0 && (growproc(n) < 0)) return -1;
    if(n > 0) myproc()->sz += n;
    return addr;
}
```

2. 当进程访问其本应拥有但实际未映射的虚拟内存时，将触发trap，我们对其进行处理，当指令合法时分配该页面。

kernel/trap.c/usertrap():

```
else if(r_scause() == 15 || r_scause() == 13){
    virtualA = r_stval();
    if(virtualA >= MAXVA) {
        p->killed = 1;
        exit(-1);
    }

    pte = walk(p->pagetable, virtualA, 0);
    uint64 faultAddr = PGROUNDOWN(virtualA);
    if(faultAddr >= (uint64)p->sz || faultAddr < (uint64)p->trapframe->sp){
        p->killed = 1;
        exit(-1);
    }
    char* mem;
    mem = kalloc();
    if(mem == 0){
        p->killed = 1;
        exit(-1);
    }
    else{
        memset(mem, 0, PGSIZE);
        if(mappages(p->pagetable, faultAddr, PGSIZE, (uint64)mem, PTE_W|PTE_R|PTE_U) != 0){
            kfree(mem);
            p->killed = 1;
            exit(-1);
        }
    }
}
```

3. 在拷贝内存时，我们可能会拷贝本应分配但由于使用内存懒惰分配机制而实际未分配的页面，因此在拷贝时我们应检测出这种情况，并为其分配页面。

kernel/vm.c:

```

copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
    uint64 n, va0, pa0;
    struct proc* p = myproc();

    while(len > 0){
        va0 = PGROUNDDOWN(dstva);
        pa0 = walkaddr(pagetable, va0);
        if(pa0 == 0){
            if(va0 >= (uint64)p->sz || va0 < p->trapframe->sp){
                return -1;
            }
            else{
                pa0 = (uint64)kalloc();
                if(pa0 == 0){
                    acquire(&p->lock);
                    p->killed = 1;
                    release(&p->lock);
                }
                else{
                    memset((void*) pa0, 0, PGSIZE);
                    if(mappages(pagetable, va0, PGSIZE, (uint64)pa0, PTE_W|PTE_R|PTE_U) != 0){
                        kfree((void*) pa0);
                        acquire(&p->lock);
                        p->killed = 1;
                        release(&p->lock);
                    }
                }
            }
        }
        len -= PGSIZE;
    }
}

```

copyin()同理修改。

4. 在释放内存时，我们会释放那些本来就没有分配的页面（因为在XV6中本来没有内存懒惰分配机制，其默认释放内存时会认为所有页面都已经正常分配），因而我们还要对内存释放的代码做一点修改，防止panic的产生。

kernel/vm.c:

```

void
uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
{
    uint64 a;
    pte_t *pte;

    if((va % PGSIZE) != 0)
        panic("uvmunmap: not aligned");

    for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
        if((pte = walk(pagetable, a, 0)) == 0)
            continue;
        if((*pte & PTE_V) == 0)
            continue;
        if(PTE_FLAGS(*pte) == PTE_V)
            panic("uvmunmap: not a leaf");
        if(do_free){
            uint64 pa = PTE2PA(*pte);
            kfree((void*)pa);
        }
        *pte = 0;
    }
}

```

以下是“进程写时复制”的实现步骤：

1. 在fork()时，不直接拷贝内存，而是标记那些需要写时复制的页面。同时，对每一个页表，记录其被引用的次数。

kernel/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){
        // LAZY ALLOCATION
        if((pte = walk(old, i, 0)) == 0)
            continue;
        if((*pte & PTE_V) == 0)
            continue;
        pa = PTE2PA(*pte);

        (*pte) = (*pte) & (~PTE_W);
        (*pte) = (*pte) | PTE_COW; // COW

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

        acquire(&rcLock);
        ref_cnt[(uint64)pa >> 12]++;
        release(&rcLock);
    }
    return 0;

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

2. 在需要写入页面时，若被写入页面标记了PTE_COW，则对其执行cow()，cow()会将页面复制出一份可写的副本，并对原页面执行kfree()。同时，我们修改了kfree()，每次减少引用计数，当引用计数为0时真正执行原kfree()。

kernel/vm.c/copyout():

```
pte_t* pte = walk(pagetable, va0, 0);
if((pte == 0) || ((*pte & PTE_V) == 0) || ((*pte & PTE_U) == 0)) return -1;
if((*pte & PTE_W) == 0){
    if(cow(pagetable, pte) < 0)
        return -1;
}
pa0 = PTE2PA(*pte);
```

kernel/vm.c:

```

int
cow(pagetable_t pagetable, pte_t* pte){
    char* mem;
    uint64 pa = PTE2PA(*pte);
    if((mem = kalloc()) == 0){
        return -1;
    }

    memmove(mem, (char*)pa, PGSIZE);
    *pte = *pte & (~PTE_COW);
    *pte = *pte | PTE_W;
    uint64 flags = PTE_FLAGS(*pte);
    *pte = PA2PTE(mem) | flags;

    kfree((void*)pa);
    return 0;
}

```

kernel/kalloc:

```

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

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

    // COW: update reference
    acquire(&rclock);
    uint8* ref = &ref_cnt[((uint64)pa) >> 12];
    *ref = (*ref > 0) ? (*ref - 1) : (*ref);
    release(&rclock);

    // COW: No free if still have reference
    if((*ref) > 0) return;

    // 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);
}

```

3. 在kernel/kalloc中多处进行修改以支持页表的引用计数。

如：

```
uint8 ref_cnt[PHYSTOP >> 12];
void *
kalloc(void)
{
    struct run *r;

    acquire(&kmem.lock);
    r = kmem.freelist;
    if(r){
        kmem.freelist = r->next;
        acquire(&rcLock);
        ref_cnt[(uint64)r >> 12] = 1;
        release(&rcLock);
    }
    release(&kmem.lock);

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

四、目前遇到的问题

我们对于很多功能都有一些想尝试的想法，但并不清楚这些想法最终是否能实现。由于缺少“标准答案”，我们往往担心在某项本无法实现的功能上投入过多精力导致时间的浪费。另外，对汇编代码的不熟悉一定程度上拖慢了我们的学习进度。

五、人员分工

题目调研由 全体组员 共同完成。

目前已实现的两个功能由 王集 完成。

该报告由 张昱 完成。

预期未来功能的实现将主要由 张昱 和 王集 完成，功能的测试与效果展示将主要由 幸若凡 和 张凯伦 完成。

