操作系统大作业报告 改进XV6内存管理方式

张昱 *2020012386*

幸若凡 *2020012365*

张凯伦 *2020012387*

王集 *2020012389*

一、选题概述

XV6通过页表机制实现了对于内存空间的控制，将页表的三级索引作为从虚拟地址到物理地址的映射方式。单一页面大小为4096字节。页表中的每一个页表项的后十位用于进行标记，这些标记告诉我们当前页表项的对应页面的状态。XV6使用链表存储当前的空闲页面，在用户请求时逐一进行分配，被释放的页面会重新添加回链表。可以看出XV6的内存管理方式比较简单。在本次大作业中，我们对XV6的内存管理方式进行了功能添加与改进。

二、实验环境

实验环境：Ubuntu 22.04 LTS

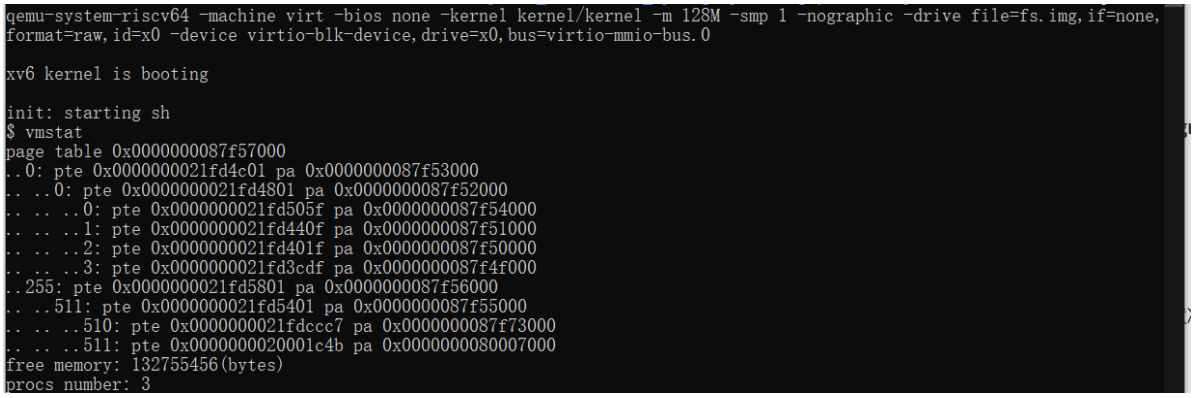
执行：

· `sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu` 安装riscv支持的qemu

· 运行`make qemu`

三、实现功能及方法

为方便调试，我们首先自己实现了一个用户程序“vmstat”（其本质是实现了系统调用“sysinfo”），可以打印XV6目前页表、内存、进程的基本状况。



1. 内存懒惰分配

内存懒惰分配（lazy allocation）功能。我们注意到目前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。

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():

```
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();
  pte_t* pte;
  uint64 virtualA;

  // save user program counter.
  p->trapframe->epc = r_sepc();

  if(r_scause() == 8){
    // system call

    if(p->killed)
      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 sstatus &c registers,
    // so don't enable until done with those registers.
    intr_on();

    syscall();
  }
  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);
    if(r_scause() == 15 && pte && ((*pte & PTE_COW) != 0))
    {
      if(pte == 0)
        panic("cow: pte");
      if(((*pte) & PTE_V) == 0)
        panic("cow: not valid");
      if(((*pte) & PTE_U) == 0)
        panic("cow: not user");
      if(cow(p->pagetable, pte) < 0){
        p->killed = 1;
        exit(-1);
      }
    }
    // lazy allocation
    else{
      if(virtualA >= (uint64)p->sz || virtualA < (uint64)p->trapframe->sp){
        p->killed = 1;
        exit(-1);
      }

      int mmapret = lazy_mapping(p, r_stval());
      if(mmapret < 0) {
        printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p-
>pid);
        printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
        p->killed = 1;
        exit(-1);
      }
      if(mmapret == 5){
        uint64 faultAddr = PGROUNDDOWN(virtualA);
        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);
          }
        }
      }
    }
  }
  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());
    p->killed = 1;
  }

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

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

  usertrapret();
}
```

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:

```
int
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);
          }
        }
      }
    }

    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);

    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;
}
```

copyin()同理修改。

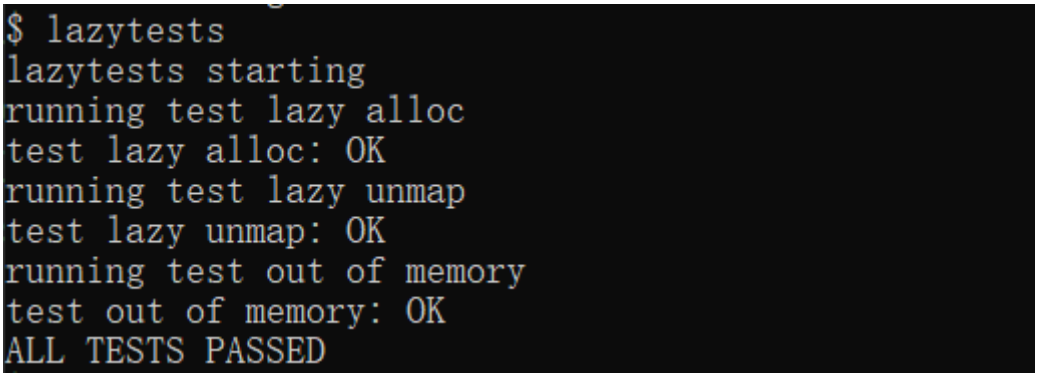
4. 在释放内存时，我们会释放那些本来就没有分配的页面（因为在XV6中本来没有内存懒惰分配机制，其默认释放内存时会认为所有页面都已经正常分配），因而我们还要对内存释放的代码做一点修改，防止panic的产生。
kernel/vm.c:

```
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");

  // printf("free from %p to %p, total %d pages\n",va, va + npages*PGSIZE,
npages);
  for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
    // printf("freeing %p\n", a);
    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;
  }
}
```

通过测试文件user/lazytests.c对于懒惰分配页表功能进行测试，结果如下：



2. 进程写时复制

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**。写时拷贝可以推迟甚至避免冗余的数据拷贝，在进程调用fork时，内核此时并不复制整个进程的地址空间，fork的实际开销就是复制父进程的页表以及给子进程创建一个进程描述符，这种优化可以避免拷贝大量根本不会用到的数据，提升XV6的运行速度。

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():

```
...
if((*pte & PTE_W) == 0){
    if(cow(pagetable, pte) < 0)
    return -1;
}
...
```

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：

```
...
// divided by 4096
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;
}
```

通过测试函数cowtest.c对于写时拷贝功能进行测试。

user/cowtest.c

```
//
// tests for copy-on-write fork() assignment.
//

#include "kernel/types.h"
#include "kernel/memlayout.h"
#include "user/user.h"

// allocate more than half of physical memory,
// then fork. this will fail in the default
// kernel, which does not support copy-on-write.
void
simpletest()
{
  uint64 phys_size = PHYSTOP - KERNBASE;
```

```
  int sz = (phys_size / 3) * 2;

  printf("simple: ");

  char *p = sbrk(sz);
  if(p == (char*)0xffffffffffffffffL){
    printf("sbrk(%d) failed\n", sz);
    exit(-1);
  }

  for(char *q = p; q < p + sz; q += 4096){
    *(int*)q = getpid();
  }

  int pid = fork();
  if(pid < 0){
    printf("fork() failed\n");
    exit(-1);
  }

  if(pid == 0)
    exit(0);

  wait(0);

  if(sbrk(-sz) == (char*)0xffffffffffffffffL){
    printf("sbrk(-%d) failed\n", sz);
    exit(-1);
  }

  printf("ok\n");
}

// three processes all write COW memory.
// this causes more than half of physical memory
// to be allocated, so it also checks whether
// copied pages are freed.
void
threetest()
{
  uint64 phys_size = PHYSTOP - KERNBASE;
  int sz = phys_size / 4;
  int pid1, pid2;

  printf("three: ");

  char *p = sbrk(sz);
  if(p == (char*)0xffffffffffffffffL){
    printf("sbrk(%d) failed\n", sz);
    exit(-1);
  }

  pid1 = fork();
  if(pid1 < 0){
    printf("fork failed\n");
    exit(-1);
  }
  if(pid1 == 0){
```

```
    pid2 = fork();
    if(pid2 < 0){
      printf("fork failed");
      exit(-1);
    }
    if(pid2 == 0){
      for(char *q = p; q < p + (sz/5)*4; q += 4096){
        *(int*)q = getpid();
      }
      for(char *q = p; q < p + (sz/5)*4; q += 4096){
        if(*(int*)q != getpid()){
          printf("wrong content\n");
          exit(-1);
        }
      }
      exit(-1);
    }
    for(char *q = p; q < p + (sz/2); q += 4096){
      *(int*)q = 9999;
    }
    exit(0);
  }

  for(char *q = p; q < p + sz; q += 4096){
    *(int*)q = getpid();
  }

  wait(0);

  sleep(1);

  for(char *q = p; q < p + sz; q += 4096){
    if(*(int*)q != getpid()){
      printf("wrong content\n");
      exit(-1);
    }
  }

  if(sbrk(-sz) == (char*)0xffffffffffffffffL){
    printf("sbrk(-%d) failed\n", sz);
    exit(-1);
  }

  printf("ok\n");
}

char junk1[4096];
int fds[2];
char junk2[4096];
char buf[4096];
char junk3[4096];

// test whether copyout() simulates COW faults.
void
filetest()
{
  printf("file: ");
```

```
  buf[0] = 99;

  for(int i = 0; i < 4; i++){
    if(pipe(fds) != 0){
      printf("pipe() failed\n");
      exit(-1);
    }
    int pid = fork();
    if(pid < 0){
      printf("fork failed\n");
      exit(-1);
    }
    if(pid == 0){
      sleep(1);
      if(read(fds[0], buf, sizeof(i)) != sizeof(i)){
        printf("error: read failed\n");
        exit(1);
      }
      sleep(1);
      int j = *(int*)buf;
      if(j != i){
        printf("error: read the wrong value\n");
        exit(1);
      }
      exit(0);
    }
    if(write(fds[1], &i, sizeof(i)) != sizeof(i)){
      printf("error: write failed\n");
      exit(-1);
    }
  }

  int xstatus = 0;
  for(int i = 0; i < 4; i++) {
    wait(&xstatus);
    if(xstatus != 0) {
      exit(1);
    }
  }

  if(buf[0] != 99){
    printf("error: child overwrote parent\n");
    exit(1);
  }

  printf("ok\n");
}

int
main(int argc, char *argv[])
{
  simpletest();

  // check that the first simpletest() freed the physical memory.
  simpletest();

  threetest();
  threetest();
```

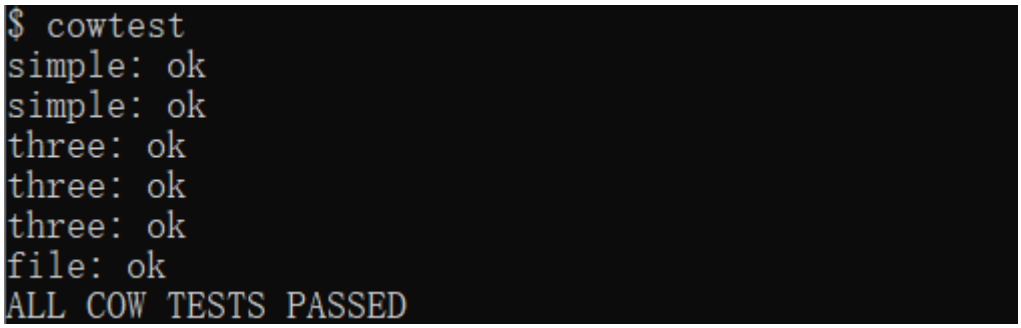
```
  threetest();

  filetest();

  printf("ALL COW TESTS PASSED\n");

  exit(0);
}
```

运行结果如下：



3. 更改进程用户栈的可用大小

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，这使得XV6操作系统可以支持更大栈的进程的运行。

XV6本来在exec分配栈页面时分配两个页面，将第二个页面用做用户栈。我们现在为其分配多个页面，第一个用作保护页面，其余用作栈。

kernel/exec.c

```
uint64 pageNum = 2;
int
exec(char *path, char **argv)
{
  ...
  sz = PGROUNDUP(sz);
  uint64 sz1;
  if((sz1 = uvmalloc(pagetable, sz, sz + (1 + pageNum) * PGSIZE)) == 0)
    goto bad;
  sz = sz1;
  uvmclear(pagetable, sz - (1 + pageNum) * PGSIZE);
  sp = sz;
  stackbase = sp - pageNum * PGSIZE;

  for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
      goto bad;
    sp -= strlen(argv[argc]) + 1;
    sp -= sp % 16; // riscv sp must be 16-byte aligned
    if(sp < stackbase)
      goto bad;
    if(copyout(pagetable, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
      goto bad;
    ustack[argc] = sp;
  }
  ustack[argc] = 0;
  ...
```

```
}
```

4. 内存映射文件

内存映射文件(mmap)是由一个文件到一块内存的映射，使进程虚拟地址空间的某个区域与磁盘上某个文件的部分或全部内容的建立映射。建立映射后，通过该区域可以直接对被映射的磁盘文件进行访问，而不必执行文件I/O操作也无需对文件内容进行缓冲处理。就好像整个被映射的文件都加载到了内存一样，因此内存文件映射非常适合于用来管理大文件。内存映射文件对程序的提速，只在处理大文件或非常频繁的文件读写操作时效果才明显。通过内存映射，相当于将磁盘上的文件所在空间建立成一块虚拟内存，程序访问时可按内存的方式进行，省去了普通io方式的一些环节，其实真正要读写操作时，会进行换页，将这些个“虚拟内存”读到物理内存中，从而加速文件相关操作。

1. 定义VM_Area结构体，加入proc结构体，并且将vmas初始化为0。

   kernel/proc.h:

   ```
   struct VMA{
     uint64 used;
     uint64 address;
     uint64 size;
     uint64 endaddr;
     uint64 permissions;
     struct file* the_file_being_mapped;
     uint64 flags;
     uint64 offset;
   };

   // 将VM_Area加入proc结构体中
   struct proc {
     ...
     struct VMA vmas[16];     // 虚拟内存区域
   }
   ```

2. 参考懒惰分配方法，将`p->size`作为分配的虚拟地址基，但不实际分配物理页面。通过这种方式来响应页错误。

   kernel/mmap.c

   ```
   void *
   mmap(void *addr, uint length, int prot, int flags, struct file* file, uint
   offset)
   {
       // find free space
       // p->size shall be a good place
       struct proc* p = myproc();
       if(!file->writable && (prot & PROT_WRITE) && (flags & MAP_SHARED))
           return (void *) MAP_FAILED;

       // add this address to VMA
       for(int i = 0; i < 16; i++){
           if(p->vmas[i].used == 1) continue;
           p->vmas[i].used = 1;
           p->vmas[i].address = PGROUNDUP(p->sz);
           p->vmas[i].size = length;
           p->vmas[i].permissions = prot;
           p->vmas[i].the_file_being_mapped = filedup(file);
           p->vmas[i].endaddr = PGROUNDUP(p->sz + length);
   ```

```
        p->vmas[i].flags = flags;
        p->sz += PGROUNDUP(length);
        return (void *)p->vmas[i].address;
    }
    return (void *) MAP_FAILED;
}
int lazy_mapping(struct proc* p, uint64 addr){
    for (int i = 0; i < 16; i++)
    {
        struct VMA* v = &(p->vmas[i]);
        if (v->used == 1 && addr >= v->address && addr < v->endaddr)
        {
            uint64 start = PGROUNDDOWN(addr);
            uint64 offset = start - v->address + v->offset;

            char* mem = kalloc();
            if(!mem) return -1;
            memset(mem, 0, PGSIZE);

            if(mappages(p->pagetable, start, PGSIZE,
            (uint64)mem, (v->permissions << 1)|PTE_U) != 0){
                kfree(mem);
                return -1;
            }

            ilock(v->the_file_being_mapped->ip);
            readi(v->the_file_being_mapped->ip, 1, start, offset, PGSIZE);
            iunlock(v->the_file_being_mapped->ip);
            return 0;
        }

    }
    //not found
    return 5;
}
```

kernel/sysfile.c

```
uint64
sys_mmap(void){
  uint64 addr;
  if(argaddr(0, &addr) < 0){
    panic("mmap: no address");
  }

  int length;
  if(argint(1, &length) < 0){
    panic("mmap: no length");
  }

  int prot;
  if(argint(2, &prot) < 0){
    panic("mmap: no prot");
  }

  int flags;
  if(argint(3, &flags) < 0){
```

```
    panic("mmap: no flags");
  }

  int fd;
  struct file* mappedfile;
  if (argfd(4, &fd, &mappedfile) < 0)
  {
    panic("mmap: no fd");
  }

  int offset;
  if (argint(5, &offset) < 0){
    panic("mmap: no offset");
  }

  return (uint64)mmap((void *)addr, length, prot, flags, mappedfile,
offset);
}
```

3. 在userstrap中处理相应的页面错误。

   kernel/trap.c

```
int mmapret = lazy_mapping(p, r_stval());
      if(mmapret < 0) {
        printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p-
>pid);
        printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
        p->killed = 1;
        exit(-1);
      }
```

4. 对于munmap：找到地址范围的VMA并取消映射指定页面，如果该函数删除了先前mmap所有页面，它应该减少相应引用计数。如果未映射的页面已被修改，并且文件已映射到MAP_SHARED，将页面写回该文件。

   kernel/mmap.c

```
int
filewrite_offset(struct file *f, uint64 addr, int n, int offset) {
    int r, ret = 0;
    if(f->writable == 0)
        return -1;
    if(f->type != FD_INODE) {
        panic("filewrite: only FINODE implemented!");
    }

    int max = ((MAXOPBLOCKS-1-1-2) / 2) * BSIZE;
    int i = 0;
    while(i < n) {
        int n1 = n - i;
        if(n1 > max)
            n1 = max;

        begin_op();
        ilock(f->ip);
        if ((r = writei(f->ip, 1, addr + i, offset, n1)) > 0)
```

```
            offset += r;
        iunlock(f->ip);
        end_op();

        if(r != n1) {
            break;
        }
        i += r;
    }
    ret = (i == n ? n : -1);
    return ret;
}
int
munmap(void *addr, uint length){
    uint64 addri = (uint64) addr;
    struct proc* p = myproc();
    for (int i = 0; i < 16; i++)
    {
        struct VMA* v = &(p->vmas[i]);
        if(v->used == 1 && addri < v->endaddr && addri >= v->address){
            int toclose = 0;
            int offset = v->offset;
            addri = PGROUNDDOWN(addri);
            length = PGROUNDUP(length);

            if (addri == v->address)
            {
                if (length == v->size)
                {
                    v->size = 0;
                    toclose = 1;
                }
                else{
                    v->address += length;
                    v->size -= length;
                    v->offset += length;
                }
            }
            else
            {
                v->size -= length;
            }
            if (v->flags & MAP_SHARED)
            {
                filewrite_offset(v->the_file_being_mapped, (uint64)addr,
length, offset);
            }
            if(walkaddr(p->pagetable, addri) != 0)
                uvmunmap(p->pagetable, addri, length/PGSIZE, 1);
            if(toclose)
                fileclose(v->the_file_being_mapped);
        }
    }
    return 0;
}
```

5. 在exit中将进程已映射区域取消映射。修改fork以确保子对象与父对象有相同的映射区域。在应该增加VMA的struct file的引用计数。在子进程的页面错误处理程序中，可以分配新的物理页面。

   kernel/proc.c

```
void
exit(int status)
{
  struct proc *p = myproc();

  if(p == initproc)
    {panic("init exiting");}

  for(int i = 0; i < 16; i++) {
      struct VMA *v = &(p->vmas[i]);
      if(v->size != 0){
        munmap((void *)v->address, v->size);
  }
  }
  // Close all open files.
  for(int fd = 0; fd < NOFILE; fd++){
    if(p->ofile[fd]){
      struct file *f = p->ofile[fd];
      fileclose(f);
      p->ofile[fd] = 0;
    }
  }
...

int
fork(void)
{
   ...
   safestrcpy(np->name, p->name, sizeof(p->name));

   for(int i = 0; i < 16; i++) {
   if(p->vmas[i].size) {
       memmove(&(np->vmas[i]), &(p->vmas[i]), sizeof(struct VMA));
       filedup(p->vmas[i].the_file_being_mapped);
   } else {
       np->vmas[i].used = 0;
       }
   }
  ...
}
```

6. 在网上找到对于mmaptest.c对于mmap功能的映射功能和子进程能否共享mmap文件中的地址进行测试，结果如下：

```
$ mmaptest
mmap_test starting
test mmap f
test mmap f: OK
test mmap private
test mmap private: OK
test mmap read-only
test mmap read-only: OK
test mmap read/write
test mmap read/write: OK
test mmap dirty
test mmap dirty: OK
test not-mapped unmap
test not-mapped unmap: OK
test mmap two files
test mmap two files: OK
mmap_test: ALL OK
fork_test starting
fork_test OK
mmaptest: all tests succeeded
```

四、过程中遇到的困难及解决方案

1. **实验框架较新**

   XV6的实验框架比较复杂，在一开始接触之后并不知道该从何下手。而且在网络上能够查到的大部分资料都是基于x86实现的，关于riscv的资料比较少，由于两种框架的内存映射方式不同，所以这给我们改进内存管理方式时造成了很大的困难。对于一些没有实现思路的改进方案，我们通过查阅资料将其代码的思路看懂，并且尝试在riscv的XV6下进行复现。

2. **Debug**困难

   xv6是多核心操作系统，而且处于硬件-软件交互的边界，因此很难debug。用gdb设置断点，有很多情况下根本没用。有一次，操作系统的内存分配出了问题。我们在程序出错的位置之前设置断点，想要单步跟踪过去，程序却在一个根本不可能出错的位置panic。这完全出乎我们的意料之外。无论怎么设置断点，都找不到问题出在哪里。后来又回归printf法，寻找多次，才终于发现问题所在。

3. **知识储备不足**

   因为操作系统处于硬件-软件交互的边界，很多内容涉及到计算机内部的运行机制，对risc-v的寄存器和指令集需要有一定程度的了解。例如我们需要知道r_scause等寄存器中存储的内容是什么。而我们此前没有学过这些知识。通过详细阅读risc-v的手册，以及xv6手册，我们才逐步学习到了这些东西，克服了困难。

4. **牵连项过多**

   内存管理是操作系统中最复杂的一块。它上接进程，下连文件，因此与各方面都联系在一起。进程的创建，执行，退出；系统调用的参数传递都离不开内存。文件读写，I/O，也都离不开内存。正是因为这个原因，内存出问题，一连串的组件都会出问题，而他们出问题不一定报错说是内存管理出了问题。稳健又高效的内存管理策略正是程序在计算机系统上良好运转的坚实基础。我们在写内存管理的代码时也不得不慎之又慎。

5. **页表，虚拟、物理地址，指针等易混淆**

   页表是虚拟地址到物理地址的映射关系，c语言中又用指针对地址进行引用和修改。这导致代码变得很绕，需要弄清楚改了什么东西，这里的地址是虚拟地址还是物理地址，非常麻烦。往往很简单的几行代码要反复阅读才能理解。

五、人员分工

题目的前期调研与项目相关知识的学习由全体组员共同完成。

项目的整体框架由全体组员共同商讨完成。

由部分组员（个人）完成的任务列举如下：

张昱：中期报告、用户栈的修改、期末答辩

幸若凡：内存映射文件的实现、末期报告

张凯伦：内存映射文件的实现

王集：内存懒惰分配的实现、进程写时复制的实现

六、一些题外话（by 张昱）

不得不承认，项目的最终进度与我（中期报告）的预期有较大差距。一方面是由于期末这段时间组员们的时间的确比较紧张，另一方面也是由于我没有提前做好规划、没有及时督促组员们。对于中期报告提到而最终没有实现或是实现不完全的功能，我想在这里做一些解释。

关于用户栈，我们只实现了最简单的修改。实际上，写一个系统调用实时控制pageNum变量也并非难事。但我们由于并没有真实模拟大量不同用户进程的使用场景，也就没有什么动机去做进一步的改进。

关于使用链表存储空闲页面，我们起初希望能改用更高级的数据结构，但在更充分的思考后，我们发现，对于页式存储，实际物理页面的位置并不影响性能，耗费精力去维护更高级的数据结构反而会浪费时间。我看到有一个小组为每一个CPU设计了单独的freelist，我觉得这挺有创意，但细想来，这就像为固态硬盘分区一样，只是看起来整齐，实际上性能并没有什么提升，反而多了一些限制（比如限制了每个CPU的可用物理页面）。

关于请求调页与页面置换功能，出于对XV6文件管理的不了解（不清楚需要替换的内容如何存、存在什么位置），我们在实现了他们的前置功能内存映射文件后，并没能在预期的时间内实现这两个功能。