Lab10: mmap 实验报告

mmap是一种内存映射文件的方法，即将一个文件或者其它对象映射到进程的地址空间，实现文件磁盘地址和进程虚拟地址空间中一段虚拟地址的一一对映关系。

实现这样的映射关系后，进程就可以采用指针的方式读写操作这一段内存，而操作系统会自动回写脏页面到对应的文件磁盘上，即完成了对文件的操作而不必再调用read、write等系统调用函数。相应的，内核空间对这段区域的修改也直接反映用户空间，从而可以实现不同进程间的文件共享。

在本实验中，需要向 xv6 添加 `mmap` 和 `munmap` 系统调用。

- `mmap` 函数是将文件地址映射到虚拟内存中，返回映射后的地址，同时记录该进程映射到的文件信息。
- `munmap` 函数就是取消进程地址空间中，文件地址某一部分的映射。

---

1. 首先切换分支：

```
$ git fetch
$ git checkout mmap
$ make clean
```

mmap(hard)

1. 添加系统调用：
   - 在 user/user.h 中添加系统调用声明：

```
void* mmap(void *addr, int length, int prot, int flags, int fd, uint offset);
int munmap(void *addr, int length);
```

   - 在 user/usys.pl 中添加 entry：

```
entry("mmap");
entry("munmap");
```

- 在 Makefile 中添加编译声明：

```
    $U/_zombie\
    $U/_mmaptest\
```

- 在 kernel/syscall.h 中添加系统调用码：

```
#define SYS_mmap   22
#define SYS_munmap 23
```

- 在 kernel/syscall.c 中添加系统调用函数引用：

```
extern uint64 sys_mmap(void);
extern uint64 sys_munmap(void);

static uint64 (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_mmap]    sys_mmap,
[SYS_munmap]  sys_munmap,
};
```

2. 在kernel/proc.h中构建一个VMA结构体数组，长度为16

```
#define VMASIZE 16
struct vma {
  int used;
  uint64 addr;
  int length;
  int prot;
  int flags;
  int fd;
  int offset;
  struct file *file;
};

enum procstate { UNUSED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  struct proc *parent;         // Parent process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  struct vma vma[VMASIZE];     // vma of file system
};
```

3. 在 kernel/sysfile.c 中添加sys_mmap函数的声明和实现：

   首先接收传来的参数，判断参数的合法性，然后遍历 VMA 数组，找到还没有使用的 vma，将参数信息添加进去。这里映射的虚拟地址，可以直接填写堆的最高地址，然后让堆继续生长。

```
uint64
sys_mmap(void)
{
  uint64 addr;
  int length, prot, flags, fd, offset;
  struct file *file;
  struct proc *p = myproc();
  if(argaddr(0, &addr) || argint(1, &length) || argint(2, &prot) ||
    argint(3, &flags) || argfd(4, &fd, &file) || argint(5, &offset)) {
    return -1;
  }
  if(!file->writable && (prot & PROT_WRITE) && flags == MAP_SHARED)
    return -1;
  length = PGROUNDUP(length);
  if(p->sz > MAXVA - length)
    return -1;
  for(int i = 0; i < VMASIZE; i++) {
    if(p->vma[i].used == 0) {
      p->vma[i].used = 1;
      p->vma[i].addr = p->sz;
      p->vma[i].length = length;
      p->vma[i].prot = prot;
      p->vma[i].flags = flags;
      p->vma[i].fd = fd;
      p->vma[i].file = file;
      p->vma[i].offset = offset;
      filedup(file);
      p->sz += length;
      return p->vma[i].addr;
    }
  }
  return -1;
}
```

4. 在kernel/trap.c中修改usertrap函数，实现trap中断处理：

   与lab5相似，由于是懒加载，在读取或写入相应的虚拟地址时，会存在地址未映射的情况。这时需要将物理地址上的数据读到虚拟地址中，然后重新进行读取或写入操作

```
72    } else if((which_dev = devintr()) != 0){
73      // ok
74    } else if(r_scause() == 13 || r_scause() == 15) {
75      uint64 va = r_stval();
76      if(va >= p->sz || va > MAXVA || PGROUNDUP(va) == PGROUNDDOWN(p->trapframe->sp)) p->killed = 1;
77    else {
78        struct vma *vma = 0;
79        for (int i = 0; i < VMASIZE; i++) {
80          if (p->vma[i].used == 1 && va >= p->vma[i].addr && va < p->vma[i].addr + p->vma[i].length) {
81            vma = &p->vma[i];
82        //printf("trap mmap: %d, addr: %d, used:%d\n", i, p->vma[i].addr, p->vma[i].used);
83            break;
84          }
85        }
86        if(vma) {
87          va = PGROUNDDOWN(va);
88          uint64 offset = va - vma->addr;
89          uint64 mem = (uint64)kalloc();
90          if(mem == 0) {
91            p->killed = 1;
92          } else {
93            memset((void*)mem, 0, PGSIZE);
94        ilock(vma->file->ip);
95            readi(vma->file->ip, 0, mem, offset, PGSIZE);
96            iunlock(vma->file->ip);
97            int flag = PTE_U;
98            if(vma->prot & PROT_READ) flag |= PTE_R;
99            if(vma->prot & PROT_WRITE) flag |= PTE_W;
100           if(vma->prot & PROT_EXEC) flag |= PTE_X;
101           if(mappages(p->pagetable, va, PGSIZE, mem, flag) != 0) {
102             kfree((void*)mem);
103             p->killed = 1;
104           }
105         }
106       }
107     }
108   } else {
109     printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
```

5. 在kernel/vm.c文件中修改uvmunmap和uvmcopy函数，由于一些地址并没有进行映射，因此在 walk 的时候，遇到这些地址直接跳过即可：

```
162  void
163  uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
164  {
165    uint64 a;
166    pte_t *pte;
167
168    if((va % PGSIZE) != 0)
169      panic("uvmunmap: not aligned");
170
171    for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
172      if((pte = walk(pagetable, a, 0)) == 0)
173        panic("uvmunmap: walk");
174      if((*pte & PTE_V) == 0)
175        continue;
176        // panic("uvmunmap: not mapped");
177      if(PTE_FLAGS(*pte) == PTE_V)
178        panic("uvmunmap: not a leaf");
179      if(do_free){
180        uint64 pa = PTE2PA(*pte);
181        kfree((void*)pa);
182      }
183      *pte = 0;
184    }
185  }
```

```c
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
  pte_t *pte;
  uint64 pa, i;
  uint flags;
  char *mem;

  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walk(old, i, 0)) == 0)
      panic("uvmcopy: pte should exist");
    if((*pte & PTE_V) == 0)
      continue;
      // panic("uvmcopy: page not present");
    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
      goto err;
    memmove(mem, (char*)pa, PGSIZE);
    if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
      kfree(mem);
      goto err;
    }
  }
  return 0;

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

6. 在 kernel/sysfile.c 中添加sys_munmap函数的声明和实现：

```c
uint64
sys_munmap(void)
{
  uint64 addr;
  int length;
  struct proc *p = myproc();
  struct vma *vma = 0;
  if(argaddr(0, &addr) || argint(1, &length))
    return -1;
  addr = PGROUNDDOWN(addr);
  length = PGROUNDUP(length);
  for(int i = 0; i < VMASIZE; i++) {
    if (addr >= p->vma[i].addr || addr < p->vma[i].addr + p->vma[i].length) {
      vma = &p->vma[i];
      break;
    }
  }
  if(vma == 0) return 0;
  if(vma->addr == addr) {
    vma->addr += length;
    vma->length -= length;
    if(vma->flags & MAP_SHARED)
      filewrite(vma->file, addr, length);
    uvmunmap(p->pagetable, addr, length/PGSIZE, 1);
    if(vma->length == 0) {
      fileclose(vma->file);
      vma->used = 0;
    }
  }
  return 0;
}
```

7. 在kernel/proc.c中修改fork和exit函数，在进程创建和退出时，复制和清空相应的文件映射：

```
300   safestrcpy(np->name, p->name, sizeof(p->name));
301
302   pid = np->pid;
303
304   np->state = RUNNABLE;
305
306   for(int i = 0; i < VMASIZE; i++) {
307     if(p->vma[i].used){
308       memmove(&(np->vma[i]), &(p->vma[i]), sizeof(p->vma[i]));
309       filedup(p->vma[i].file);
310     }
311   }
312
313   release(&np->lock);
314
315   return pid;
316 }
```

```
355 // Close all open files.
356 for(int fd = 0; fd < NOFILE; fd++){
357   if(p->ofile[fd]){
358     struct file *f = p->ofile[fd];
359     fileclose(f);
360     p->ofile[fd] = 0;
361   }
362 }
363
364 for(int i = 0; i < VMASIZE; i++) {
365   if(p->vma[i].used) {
366     if(p->vma[i].flags & MAP_SHARED)
367       filewrite(p->vma[i].file, p->vma[i].addr, p->vma[i].length);
368     fileclose(p->vma[i].file);
369     uvmunmap(p->pagetable, p->vma[i].addr, p->vma[i].length/PGSIZE, 1);
370     p->vma[i].used = 0;
371   }
372 }
373
374 begin_op();
375 iput(p->cwd);
376 end_op();
377 p->cwd = 0;
378
379 // we might re-parent a child to init. we can't be precise about
380 // waking up init, since we can't acquire its lock once we've
381 // acquired any other proc lock. so wake up init whether that's
382 // necessary or not. init may miss this wakeup, but that seems
383 // harmless.
384 acquire(&initproc->lock);
385 wakeup1(initproc);
386 release(&initproc->lock);
```

8. 执行测试命令：`mmapest`

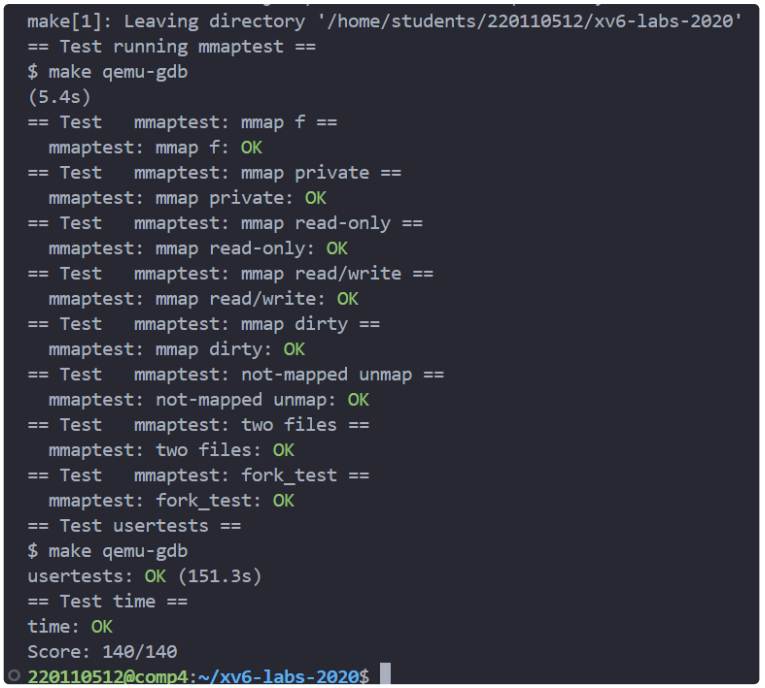
```
220110512@comp4:~/xv6-labs-2020$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0
-device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ 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
$
```

执行测试命令：`usertests`

```
test sbrkbasic: OK
test sbrkmuch: OK
test kernmem: OK
test sbrkfail: OK
test sbrkarg: OK
test validatetest: OK
test stacktest: OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
$
```

结果截图

执行命令`make grade`


make[1]: Leaving directory '/home/students/220110512/xv6-labs-2020'
== Test running mmaptest ==
$ make qemu-gdb
(5.4s)
== Test   mmaptest: mmap f ==
  mmaptest: mmap f: OK
== Test   mmaptest: mmap private ==
  mmaptest: mmap private: OK
== Test   mmaptest: mmap read-only ==
  mmaptest: mmap read-only: OK
== Test   mmaptest: mmap read/write ==
  mmaptest: mmap read/write: OK
== Test   mmaptest: mmap dirty ==
  mmaptest: mmap dirty: OK
== Test   mmaptest: not-mapped unmap ==
  mmaptest: not-mapped unmap: OK
== Test   mmaptest: two files ==
  mmaptest: two files: OK
== Test   mmaptest: fork_test ==
  mmaptest: fork_test: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (151.3s)
== Test time ==
time: OK
Score: 140/140
220110512@comp4:~/xv6-labs-2020$