# Lab10 mmap



## 前置知识

## 实验内容

### 任务

- 在本次实验中我们要求实现 `mmap` 和 `munmap` 系统调用，在实现之前，我们首先需要了解一下 `mmap` 系统调用是做什么的。根据 `mmap` 的描述，`mmap` 是用来将文件或设备内容映射到内存的。`mmap` 使用懒加载方法，因为需要读取的文件内容大小很可能要比可使用的物理内存要大，当用户访问页面会造成页错误，此时会产生异常，此时程序跳转到内核态由内核态为错误的页面读入文件并返回用户态继续执行。当文件不再需要的时候需要调用 `munmap` 解除映射，如果存在对应的标志位的话，还需要进行文件写回操作。`mmap` 可以由用户态直接访问文件或者设备的内容而不需要内核态与用户态进行拷贝数据，极大提高了 IO 的性能。

## 实验准备

1. **获取并切换到`mmap`分支：** 打开终端，进入xv6的源代码目录，然后执行以下命令：

```
bash复制代码$ git fetch
$ git checkout mmap
$ make clean
```

2. **理解`mmap`和`munmap`：**

   - `mmap`系统调用的格式如下：

```
c
复制代码
void *mmap(void *addr, size_t len, int prot, int
flags, int fd, off_t offset);
```

     - `addr`为0时，内核决定映射文件的虚拟地址。
     - `len`是要映射的字节数。
     - `prot`表示内存是否应该映射为可读、可写或可执行。
     - `flags`可以是`MAP_SHARED`或`MAP_PRIVATE`。
     - `fd`是要映射的文件的文件描述符。
     - `offset`表示从文件的哪个位置开始映射，可以假设为0。

   - `munmap`系统调用的格式如下：

```
c
复制代码
int munmap(void *addr, size_t len);
```

     - `munmap`应该从指定的地址范围中移除`mmap`映射。

# 实验任务

## 1. 设置系统调用并编译

**任务描述：** 首先为`mmap`和`munmap`创建系统调用编号，并在`user/usys.pl`和`user/user.h`中添加对应的条目。然后在`kernel/sysfile.c`中实现一个空的`sys_mmap`和`sys_munmap`函数。这样你可以编译`mmaptest`测试程序，但最初的`mmap`调用会失败。

**步骤：**

1. 在`UPROGS`中添加`_mmaptest`。
2. 在`kernel/sysfile.c`中实现`sys_mmap`和`sys_munmap`函数，暂时只返回错误。
3. 编译并运行`mmaptest`，测试会在第一次`mmap`调用时失败。

## 2. 实现`mmap`的懒惰分配

**任务描述：** 为提高效率，`mmap`应当懒惰地填充页表，即`mmap`不应立即分配物理内存或读取文件，而是在发生页面错误时才处理。这确保了对大文件的`mmap`操作速度较快，并且可以映射大于物理内存的文件。

**步骤：**

1. 定义一个结构来记录每个进程的`mmap`映射，这个结构对应于“虚拟内存区域”（VMA）。
2. 在`mmap`中找到进程地址空间中未使用的区域来映射文件，并将VMA添加到进程的映射区域表中。
3. 在页面错误处理代码中，处理`mmap`的页面错误，分配物理内存并从文件中读取数据。

## 3. 实现`munmap`

**任务描述：** 实现`munmap`系统调用，查找对应地址范围的VMA并取消映射指定的页。如果该页已被修改且文件映射为`MAP_SHARED`，则将修改写回文件。

**步骤：**

1. 在`munmap`中查找对应的VMA，并使用`uvmunmap`取消映射指定的页。
2. 如果取消映射的页已被修改并且文件映射为`MAP_SHARED`，则将页面写回文件。

### 4. 修改`exit`和`fork`

任务描述：

- 在进程退出时（`exit`），像调用`munmap`一样取消进程的映射区域。
- 修改`fork`，确保子进程与父进程具有相同的映射区域，并在页面错误处理时为子进程分配新的物理页。

步骤：

1. 在`exit`中取消进程的所有映射区域。
2. 修改`fork`，确保子进程继承父进程的映射区域。

## 代码编写

- 接下来，我们来研究一下实现，首先我们需要一个结构体用来保存 mmap 的映射关系，也就是文档中的映射关系，用于在产生异常的时候映射与解除映射，我们添加了 `virtual_memory_area` 这个结构体:

```
// 记录 mmap 信息的结构体
struct virtual_memory_area {
    // 映射虚拟内存起始地址
    uint64 start_addr;
    // 映射虚拟内存结束地址
    uint64 end_addr;
    // 映射长度
    uint64 length;
    // 特权
    int prot;
    // 标志位
    int flags;
    // 文件描述符
    // int fd;
    struct file* file;
    // 文件偏移量
    uint64 offset;
```

```
};
```

- 每个进程都需要有这样一个结构体用于记录信息，因此我们也需要在 `proc` 中添加这个结构体，由于其属于进程的私有域，所以不需要加锁访问。之后我们就将要去实现 `mmap` 系统调用：

```
uint64 sys_mmap(void) {
  uint64 addr, length, offset;
  int prot, flags, fd;
  argaddr(0, &addr);
  argaddr(1, &length);
  argint(2, &prot);
  argint(3, &flags);
  argint(4, &fd);
  argaddr(5, &offset);
  uint64 ret;
  if((ret = (uint64)mmap((void*)addr, length, prot, flags, fd, offset)) < 0){
    printf("[Kernel] fail to mmap.\n");
    return -1;
  }
  return ret;
}

void* mmap(void* addr, uint64 length, int prot, int flags, int fd, uint64 offset) {
  // 此时应当从该进程中发现一块未被使用的虚拟内存空间
  uint64 start_addr = find_unallocated_area(length);
  if(start_addr == 0){
    printf("[Kernel] mmap: can't find unallocated area");
    return (void*)-1;
  }
  // 构造 mm_area
  struct virtual_memory_area m;
  m.start_addr = start_addr;
  m.end_addr = start_addr + length;
  m.length = length;
  m.file = myproc()->ofile[fd];
  m.flags = flags;
  m.prot = prot;
  m.offset = offset;
```

```
  // 检查权限位是否满足映射要求
  if((m.prot & PROT_WRITE) && (m.flags == MAP_SHARED) &&
(!m.file->writable)){
    printf("[Kernel] mmap: prot is invalid.\n");
    return (void*)-1;
  }
  // 增加文件的引用
  struct file* f = myproc()->ofile[fd];
  filedup(f);
  // 将 mm_area 放入结构体中
  if(push_mm_area(m) == -1){
    printf("[Kernel] mmap: fail to push memory area.\n");
    return (void*)-1;
  }
  return (void*)start_addr;
}
```

- 在实现中我们首先从虚拟内存域中找到一块可用的内存，然后不实际分配内存而只是构造 `virtual_memory_area` 结构体并将其存到进程的 `mm_area` 中去，然后增加文件引用并返回。如果用户程序访问这段内存，我们的操作系统将会触发异常并调用 `map_file` 来进行处理：

```
int map_file(uint64 addr) {
    struct proc* p = myproc();
    struct virtual_memory_area* mm_area = find_area(addr);
    if(mm_area == 0){
      printf("[Kernel] map_file: fail to find mm_area.\n");
      return -1;
    }
    // 从文件中读一页的地址并映射到页中
    char* page = kalloc();
    // 将页初始化成0
    memset(page, 0, PGSIZE);
    if(page == 0){
      printf("[Kernel] map_file: fail to alloc kernel page.\n");
      return -1;
    }
    int flags = PTE_U;
    if(mm_area->prot & PROT_READ){
```

```
      flags |= PTE_R;
    }
    if(mm_area->prot & PROT_WRITE){
      flags |= PTE_W;
    }
    if(mappages(p->pagetable, addr, PGSIZE, (uint64)page, flags) != 0) {
      printf("[Kenrel] map_file: map page fail");
      return -1;
    }

    struct file* f = mm_area->file;
    if(f->type == FD_INODE){
      ilock(f->ip);
      // printf("[Kernel] map_file: start_addr: %p, addr: %p\n", mm_area->start_addr, addr);
      uint64 offset = addr - mm_area->start_addr;
      if(readi(f->ip, 1, addr, offset, PGSIZE) == -1){
        printf("[Kernel] map_file: fail to read file.\n");
        iunlock(f->ip);
        return -1;
      }
      // mm_area->offset += PGSIZE;
      iunlock(f->ip);
      return 0;
    }
    return -1;
}
```

- 在此函数中我们检查权限位并实际分配物理内存并进行映射，随后将文件内容读入到内存中来。当用户态不再需要文件内容的时候就会调用 `munmap` 进行取消映射:

```
uint64 sys_munmap(void){
  uint64 addr, length;
  argaddr(0, &addr);
  argaddr(1, &length);
  uint64 ret;
  if((ret = munmap((void*)addr, length)) < 0){
    return -1;
```

```
  }
  return ret;
}

int munmap(void* addr, uint64 length){
  // 找到地址对应的区域
  struct virtual_memory_area* mm_area = find_area((uint64)addr);
  // 根据地址进行切割，暂时进行简单地考虑
  uint64 start_addr = PGROUNDDOWN((uint64)addr);
  uint64 end_addr = PGROUNDUP((uint64)addr + length);
  if(end_addr == mm_area->end_addr && start_addr == mm_area->start_addr){
    struct file* f = mm_area->file;
    if(mm_area->flags == MAP_SHARED && mm_area->prot & PROT_WRITE){
      // 将内存区域写回文件
      printf("[Kernel] start_addr: %p, length: 0x%x\n", mm_area->start_addr, mm_area->length);
      if(filewrite(f, mm_area->start_addr, mm_area->length) < 0){
        printf("[Kernel] munmap: fail to write back file.\n");
        return -1;
      }
    }
    // 对虚拟内存解除映射并释放
    for(int i = 0; i < mm_area->length / PGSIZE; i++){
      uint64 addr = mm_area->start_addr + i * PGSIZE;
      uint64 pa = walkaddr(myproc()->pagetable, addr);
      if(pa != 0){
        uvmunmap(myproc()->pagetable, addr, PGSIZE, 1);
      }
    }
    // 减去文件引用
    fileclose(f);

    // 将内存区域从表中删除
    if(rm_area(mm_area) < 0){
      printf("[Kernel] munmap: fail to remove memory area from table.\n");
      return -1;
    }
    return 0;
```

```
  }else if(end_addr <= mm_area->end_addr && start_addr >= mm_area->start_addr){
    // 此时表示该区域只是一个子区域
    struct file* f = mm_area->file;
    if(mm_area->flags == MAP_SHARED && mm_area->prot & PROT_WRITE){
      // 写回文件
      // 获取偏移量
      uint offset = start_addr - mm_area->start_addr;
      uint len = end_addr - start_addr;
      if(f->type == FD_INODE){
        begin_op(f->ip->dev);
        ilock(f->ip);
        if(writei(f->ip, 1, start_addr, offset, len) < 0){
          printf("[Kernel] munmap: fail to write back file.\n");
          iunlock(f->ip);
          end_op(f->ip->dev);
          return -1;
        }
        iunlock(f->ip);
        end_op(f->ip->dev);
      }
    }
    // 对虚拟内存解除映射并释放
    uint64 len = end_addr - start_addr;
    for(int i = 0; i < len / PGSIZE; i++){
      uint64 addr = start_addr + i * PGSIZE;
      uint64 pa = walkaddr(myproc()->pagetable, addr);
      if(pa != 0){
        uvmunmap(myproc()->pagetable, addr, PGSIZE, 1);
      }
    }
    // 修改 mm_area 结构体
    if(start_addr == mm_area->start_addr) {
      mm_area->offset = end_addr - mm_area->start_addr;
      mm_area->start_addr = end_addr;
      mm_area->length = mm_area->end_addr - mm_area->start_addr;
    }else if(end_addr == mm_area->end_addr){
      mm_area->end_addr = start_addr;
      mm_area->length = mm_area->end_addr - mm_area->start_addr;
    }else{
      // 此时需要进行分块
```

```
85        panic("[Kernel] munmap: no implement!\n");
86      }
87      return 0;
88    }else if(end_addr > mm_area->end_addr){
89      panic("[Kernel] munmap: out of current range.\n");
90    }else{
91      panic("[Kernel] munmap: unresolved solution.\n");
92    }
93  }
```

- 在 `munmap` 中我们判断是否写回页面并取消映射，减少文件引用。注意这里需要根据系统调用的地址和长度来判断不同的取消映射方式，一共有三种情况：
    - 要取消映射区间的一个端点或者两个端点与内存区域重合
    - 要取消映射区间的两个端点都在内存区域范围内
    - 要取消映射区间的一个端点或者两个端点在内存区域外

在最后我们需要在 `exit` 的时候对所有内存取消映射和对文件减少引用；在 `fork` 的时候子进程需要从父进程这里拿到 `mm_area` 并对文件增加引用:

```
1  void
2  exit(int status)
3  {
4    struct proc *p = myproc();
5
6    if(p == initproc)
7      panic("init exiting");
8
9    // Close all open files.
10   for(int fd = 0; fd < NOFILE; fd++){
11     if(p->ofile[fd]){
12       struct file *f = p->ofile[fd];
13       fileclose(f);
14       p->ofile[fd] = 0;
15     }
16   }
17
18   // 释放所有 mmap 区域的内存
19   for(int i = 0; i < MM_SIZE; i++){
```

```
    if(p->mm_area[i].start_addr != 0){
      munmap((void*)p->mm_area[i].start_addr, p->mm_area[i].length);
    }
  }

  begin_op(ROOTDEV);
  iput(p->cwd);
  end_op(ROOTDEV);
  p->cwd = 0;

  // we might re-parent a child to init. we can't be precise about
  // waking up init, since we can't acquire its lock once we've
  // acquired any other proc lock. so wake up init whether that's
  // necessary or not. init may miss this wakeup, but that seems
  // harmless.
  acquire(&initproc->lock);
  wakeup1(initproc);
  release(&initproc->lock);

  // grab a copy of p->parent, to ensure that we unlock the same
  // parent we locked. in case our parent gives us away to init while
  // we're waiting for the parent lock. we may then race with an
  // exiting parent, but the result will be a harmless spurious wakeup
  // to a dead or wrong process; proc structs are never re-allocated
  // as anything else.
  acquire(&p->lock);
  struct proc *original_parent = p->parent;
  release(&p->lock);

  // we need the parent's lock in order to wake it up from wait().
  // the parent-then-child rule says we have to lock it first.
  acquire(&original_parent->lock);
```

```
52
53    acquire(&p->lock);
54
55    // Give any children to init.
56    reparent(p);
57
58    // Parent might be sleeping in wait().
59    wakeup1(original_parent);
60
61    p->xstate = status;
62    p->state = ZOMBIE;
63
64    release(&original_parent->lock);
65
66
67    // Jump into the scheduler, never to return.
68    sched();
69    panic("zombie exit");
70  }
71
72  int
73  fork(void)
74  {
75    int i, pid;
76    struct proc *np;
77    struct proc *p = myproc();
78
79    // Allocate process.
80    if((np = allocproc()) == 0){
81      return -1;
82    }
83
84    // Copy user memory from parent to child.
85    if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
86      freeproc(np);
87      release(&np->lock);
88      return -1;
89    }
90    np->sz = p->sz;
91
92    np->parent = p;
93
```

```
  // copy saved user registers.
  *(np->tf) = *(p->tf);

  // Cause fork to return 0 in the child.
  np->tf->a0 = 0;

  // increment reference counts on open file descriptors.
  for(i = 0; i < NOFILE; i++)
    if(p->ofile[i])
      np->ofile[i] = filedup(p->ofile[i]);
  np->cwd = idup(p->cwd);

  // 复制 mmap 结构体
  for(int i = 0; i < MM_SIZE; i++){
    if(p->mm_area[i].start_addr != 0){
      np->mm_area[i] = p->mm_area[i];
      // 增加文件引用
      filedup(p->mm_area[i].file);
    }
  }


  safestrcpy(np->name, p->name, sizeof(p->name));

  pid = np->pid;

  np->state = RUNNABLE;

  release(&np->lock);

  return pid;
}
```

实验得分