同济大学操作系统课程设计——Lab10: mmap

2151422武芷朵 Tongji University, 2024 Summer

Lab10: mmap：内存映射实验

项目地址：wzd232604/TJOS-xv6-2024-labs: 同济大学操作系统课程设计-xv6实验 (github.com)

综述

- 熟悉系统调用的实现过程和原理。
- 掌握在操作系统中如何管理虚拟内存和内存映射。
- 实现 mmap 系统调用，将文件映射到指定的虚拟地址。
- 实现 munmap 系统调用，取消内存映射。

切换到 `mmap` 分支：

`git fetch`

`git checkout mmap`

`make clean`

```
wzd@ubuntu:~/Desktop/xv6-labs-2021$ git fetch
wzd@ubuntu:~/Desktop/xv6-labs-2021$ git checkout mmap-f
error: pathspec 'mmap-f' did not match any file(s) known to git
wzd@ubuntu:~/Desktop/xv6-labs-2021$ git checkout mmap -f
Branch 'mmap' set up to track remote branch 'mmap' from 'origin'.
Switched to a new branch 'mmap'
wzd@ubuntu:~/Desktop/xv6-labs-2021$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*/*.o */*.d */*.asm */*.sym \
user/initcode user/initcode.out kernel/kernel fs.img \
mkfs/mkfs .gdbinit \
        user/usys.S \
user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill user/_ln us
er/_ls user/_mkdir user/_rm user/_sh user/_stressfs user/_usertests user/_grind
user/_wc user/_zombie  \
ph barrier
wzd@ubuntu:~/Desktop/xv6-labs-2021$
```

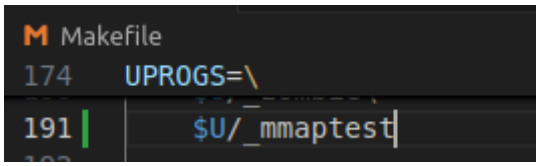
1. mmap (hard)

1.1 实验目的

添加 `mmap` 与 `mumap` 两个系统调用，实现对进程地址空间的详细控制。前者将一个文件内存映射到进程的地址空间，后者取消已有地址空间的映射。

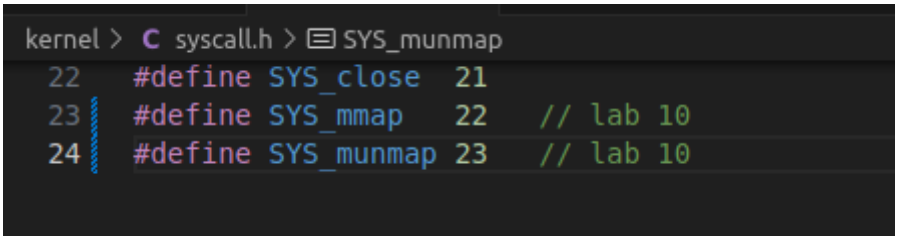
1.2 实验步骤

1. 在 `Makefile` 中添加 `$U/_mmaptest`

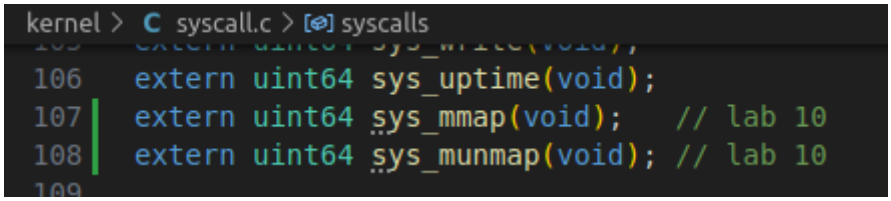


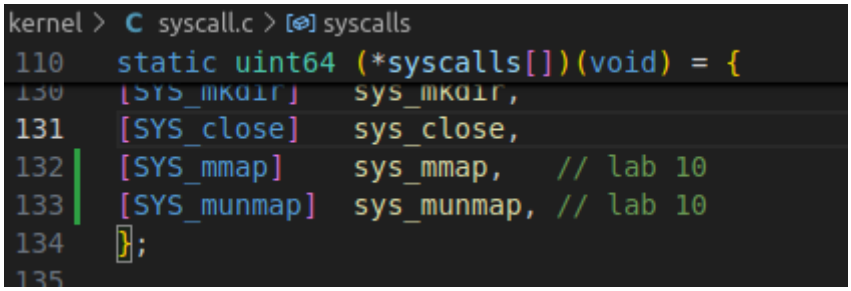
2. 添加有关 `mmap` 和 `munmap` 系统调用的定义声明：

`kernel/syscall.h`：

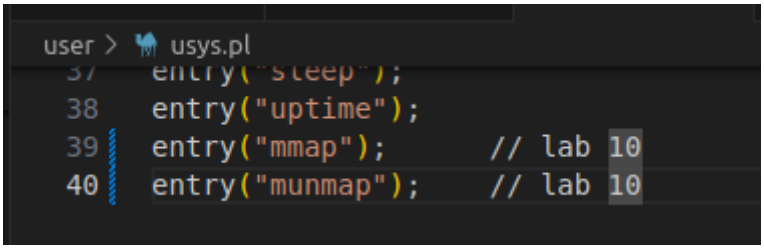


`kernel/syscall.c`：

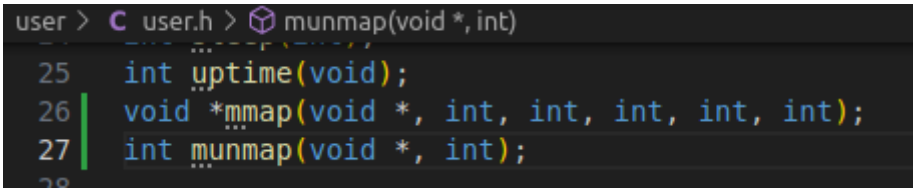




`user/usys.pl`：



`user/user.h`：



`kernel/defs.h`：

```
kernel > C defs.h > ...
183   void            virtio_disk_intr(void);
184
185   // sysfile.c
186   uint64          munmap(uint64, int);
187
```

3. 在 `kernel/proc.h` 中定义 `struct vma` 结构体。同时在 `struct proc` 结构体中定义相关字段。

   根据实验指导的提示，对于每个进程都使用一个 VMA 的数组来记录映射的内存。此处定义了 MAXVMA 表示 VMA 数组的大小，并在 struct proc 结构体中定义了 vma 数组，又因为 VMA 是进程的私有字段，因此对于 xv6 的进程单用户线程的系统，访问 VMA 无需加锁。

```
kernel > C proc.h > proc > state
85  //使用 mmap 系统调用文件映射的虚拟内存的区域的位置、大小、权限等。
86  //包括映射的起始地址、映射内存长度(大小)、权限、mmap 标志位、文件偏移以及指向的文件结构体指针。
87  #define MAXVMA 16
88  struct vma {
89    int mapped;
90    uint64 addr;// mmap address
91    int len;// mmap memory length
92    int prot;// permission
93    int flags;// the mmap flags
94    int offset;// the file offset
95    struct file *f; // pointer to the mapped file
96  };
97
98  // Per-process state
99  struct proc {
100   struct spinlock lock;
101   struct vma vma_table[MAXVMA];// Table of mapped regions
102   // p->lock must be held when using these:
```

4. 在 `kernel/sysfile.c` 中实现系统调用 `sys_mmap()`，找到进程地址空间中未使用的区域，用于映射文件，并将 VMA 添加到进程的映射区域表中。VMA 应包含指向要映射文件的 `struct file` 的指针。确保在 `mmap` 中增加文件的引用计数，以防止文件在关闭时被删除：

```
kernel > C sysfile.c > munmap(uint64, int)
488 uint64
489 sys_mmap(void)
490 {
491   // 获取参数和进程
492   struct proc *p = myproc();
493   uint64 addr;
494   int len;
495   int prot;
496   int flags;
497   int offset;
498   struct file *f;
499   if(argaddr(0, &addr) < 0 || argfd(4, 0, &f) < 0){
500     return -1;
501   }
502   if(argint(1, &len) < 0 || argint(2, &prot) < 0 || argint(3, &flags) < 0 || argint(5, &offset) < 0){
503     return -1;
504   }
505   //check the mmaptest.c to solve 确保权限
506   if(!f->writable && (prot & PROT_WRITE) && flags == MAP_SHARED) return -1;
507   // 找到一个空的 VMA 并初始化
508   for(int i = 0; i < MAXVMA; i++){
509     if(p->vma_table[i].mapped == 0){
510       p->vma_table[i].mapped = 1;
511       //if addr = 0, vma_table[i].addr is the top of heap (i.e. bottom of trapframe)
512       p->vma_table[i].addr = addr + p->sz;
513       p->vma_table[i].len = PGROUNDUP(len);
514       p->vma_table[i].prot = prot;
515       p->vma_table[i].flags = flags;
516       p->vma_table[i].offset = offset;
517       p->vma_table[i].f = filedup(f);
518       p->sz += PGROUNDUP(len);
519       return p->vma_table[i].addr;
520     }
521   }
522   return -1;
523 }
524
```

   1. 从用户传递的参数中提取信息，`len` 和 `offset` 参数使用 `int` 类型。

2. 进行参数检查，`flags` 参数只能为 `MAP_SHARED` 或 `MAP_PRIVATE`，检查权限和偏移量的合法性。
3. 从当前进程的 VMA 数组中分配一个 VMA 结构。
4. 记录本次 `mmap` 的参数到分配的 VMA 结构中，在 `kernel/memlayout.h` 中定义 `MMAPMINADDR` 宏定义, 用于表示 `mmap` 可以用于映射的最低地址。为了确定映射地址，找到已映射内存的最高地址，向上对齐以确定新映射地址。若映射的地址空间会覆盖 `TRAPFRAME` 则会报错。
5. 在 VMA 结构中记录映射的地址和长度，使用 Lazy allocation，实际内存页面在陷阱处理页面错误时分配。
6. 记录其他参数到 VMA 结构中，对于文件指针，使用 `filedup()` 增加引用计数。
7. 返回分配的地址，失败则返回 `-1`。

5. 修改 `kernel/trap.c` 中的 `usertrap()` 函数，处理页错误（Page Fault）情况。

由于在 sys_mmap() 中对文件映射的内存采用的是 Lazy allocation，在访问未加载界面时，会产生一个缺页中断。因此需要对访问文件映射内存产生的 page fault 进行处理。当在 `mmap` 映射的区域发生页面错误时，分配一个物理内存页，从相关文件中读取 4096 字节到该页面，并将其映射到用户地址空间。

kernel > C trap.c > usertrap(void)

```
40  usertrap(void)
70      syscall();
71    } else if(r_scause() == 13 || r_scause() == 15){// 缺页中断
72      uint64 va = r_stval();
73      struct vma *pvma;
74      int i = 0;
75      //p->sz point to the top of heap(i.e. bottom of trapframe)
76      //p->trapframe->sp point to the top of stack
77      if((va >= p->sz) || (va < PGROUNDDOWN(p->trapframe->sp))){
78        p->killed = 1;
79      } else{
80        va = PGROUNDDOWN(va);
81        for(; i < MAXVMA; i++){
82          pvma = &p->vma_table[i];
83          if(pvma->mapped && (va >= pvma->addr) && (va < (pvma->addr + pvma->len))){
84            char *mem;
85            mem = kalloc();
86            if(mem == 0){
87              p->killed = 1;
88              break;
89            }
90            memset(mem, 0, PGSIZE);
91            //PTE_R (1L << 1), so prot also needs to move left one bit
92            if(mappages(p->pagetable, va, PGSIZE, (uint64)mem, (pvma->prot << 1) | PTE_U) != 0){
93              kfree(mem);
94              p->killed = 1;
95              break;
96            }
97            break;
98          }
99        }
100     }
101     if(p->killed != 1 && i <= MAXVMA){
102       ilock(pvma->f->ip);
103       readi(pvma->f->ip, 1, va, va - pvma->addr, PGSIZE);
104       iunlock(pvma->f->ip);
105     }
106     }else if((which_dev = devintr()) != 0){
```

1. 根据 `r_scause()` 值，检查是否发生页错误，可能的值为 13 和 15。
2. 在当前进程的 VMA 数组中查找对应的 VMA 结构，比较虚拟地址 `va` 和 `pvma->addr + pvma->len`。
3. 如果找到匹配的 VMA，继续处理，否则忽略该页错误。
4. 对于存储错误（Store Page Fault），进行特殊处理，设置脏页标志位 `PTE_D`。
5. 执行惰性分配，使用 `kalloc()` 分配物理页，并使用 `memset()` 清空物理页。
6. 使用 `readi()` 从文件中读取数据到物理页，大小为 `PGSIZE`，对文件 inode 进行加锁。
7. 根据 VMA 记录的权限参数将 PTE 权限标志位设置为对应的读、写、执行权限。
8. 使用 `mappages()` 将物理页映射到用户进程的虚拟地址。

6. 在 `kernel/sysfile.c` 中实现系统调用 `munmap()`，找到要取消映射的地址范围的 VMA，并取消映射指定的页面。如果 `munmap` 移除了前一个 `mmap` 的所有页面，则应减少相应的 `struct file` 的引用计数。如果一个页面被修改且文件是 `MAP_SHARED` 映射，则将修改的页面写回文件：

kernel > C sysfile.c > munmap(uint64, int)

```
524
525  uint64
526  munmap(uint64 addr, int len){// 获取参数和进程
527    struct proc *p = myproc();
528    struct vma *pvma;
529    int i = 0;
530    // 找到目标 VMA
531    for(; i < MAXVMA; i++){
532      pvma = &p->vma_table[i];
533      if(pvma->mapped == 1 && addr >= pvma->addr && ((addr + len) < (pvma->addr + pvma->len))){
534        break;
535      }
536    }
537    // not found
538    if(i > MAXVMA){
539      return -1;
540    }
541    int end = addr + len;
542    int _addr = addr;
543    //readonly file can be MAP_SHARED, so need another condition f->writable
544    // 如果页面已经映射，写回修改（如果需要）并解映射
545    if((pvma->flags == MAP_SHARED) && pvma->f->writable){
546    //steal from filewritei()
547      while(addr < end){
548        int size = min(end-addr, PGSIZE);
549        begin_op();
550        ilock(pvma->f->ip);
551        if(writei(pvma->f->ip, 1, addr, addr - pvma->addr, size) != size){
552          return -1;
553        }
554        iunlock(pvma->f->ip);
555        end_op();
556        uvmunmap(p->pagetable, addr, 1, 1);
557        addr += PGSIZE;
558      }
559    }
560    // 维护文件引用和 VMA 有效性
561    if(_addr == pvma->addr){
562      //head
563      pvma->addr += len;
564      pvma->len -= len;
565    }else if(_addr + len == pvma->addr + pvma->len){
566      //tail
567      pvma->len -= len;
568    }
569
570    if(pvma->len == 0 && pvma->mapped == 1){
571      fileundup(pvma->f);
572      pvma->mapped = 0;
573    }
574    return 0;
575  }
576
```

kernel > C defs.h > fileread(file *, uint64, int)

```
30   void            fileclose(struct file*);
31   struct file*    filedup(struct file*);
32   struct file*    fileundup(struct file*);
```

```
kernel > C file.c > ...
58  struct file*
59  fileundup(struct file *f)
60  {
61    acquire(&ftable.lock);
62    if(f->ref < 1)
63      panic("fileundup");
64    f->ref--;
65    release(&ftable.lock);
66    return f;
67  }
```

1. 提取参数 `addr` 和 `length`。
2. 检查参数，确保 `length` 非负，`addr` 是 `PGSIZE` 的整数倍。
3. 根据 `addr` 和 `length` 找到对应的 VMA 结构体，未找到则返回失败。
4. 若 `length` 为 0，直接返回成功。
5. 判断 VMA 的标志位，若有 `MAP_SHARED`，则需要将修改的页面写回文件。
6. 使用脏页标志位 `PTE_D` 判断哪些页面需要写回，设置为脏页后需要写入文件。
7. 对于写回文件的大小，通常为 `PGSIZE`，但要考虑 `len` 可能不为 `PGSIZE` 整倍数。
8. 使用类似于 `filewrite()` 的方法，分批次将修改写回文件，受日志块大小影响。
9. 写回文件后，使用 `uvmunmap()` 取消用户页表中的映射，采用向上取整的取消映射方法。
10. 更新 VMA 结构体，若取消的是整个文件映射内存区域，清空该 VMA。

7. 修改 `kernel/proc.c` 中的 `exit()` 函数和 `fork()` 函数，复制、删除当前进程的 VMA 字段。

```
kernel > C proc.c > fork(void)
273  fork(void)
302    np->cwd = idup(p->cwd);
303
304    for(int i = 0; i < MAXVMA; i++){
305      if(p->vma_table[i].mapped){
306        memmove(&np->vma_table[i], &p->vma_table[i], sizeof(struct vma));
307        filedup(np->vma_table[i].f);
308      }
309    }
310
311    safestrcpy(np->name, p->name, sizeof(p->name));
```

在进程退出时，需要像 `munmap()` 一样对文件映射部分内存进行取消映射。因此添加的代码与 `munmap()` 中部分基本系统，区别在于需要遍历 VMA 数组对所有文件映射内存进行整个部分的映射取消。

```
kernel > C proc.c > fork(void)
347  exit(int status)
363    struct vma *pvma;
364    for(int i = 0; i < MAXVMA; i++){
365      pvma = &p->vma_table[i];
366      if(pvma->mapped){
367        //uvmunmap(p->pagetable, pvma->addr, pvma->len / PGSIZE, 0);
368        //memset(pvma, 0, sizeof(struct vma));
369        if(munmap(pvma->addr, pvma->len) < 0){
370          panic("exit munmap");
371        }
372      }
373    }
```

修改 `fork`，以确保子进程与父进程具有相同的映射区域。不要忘记增加 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，而不是与父进程共享页面。

8. 更改`kernel/vm.c`修改`uvmcopy()`函数中

```
kernel > C vm.c > uvmcopy(pagetable_t, pagetable_t new, uint64)
302   uvmcopy(pagetable_t old, pagetable_t new, uint64 sz
311       panic("uvmcopy: pte should exist");
312     if((*pte & PTE_V) == 0)
313       continue;
314       // panic("uvmcopy: page not present");
315     pa = PTE2PA(*pte);
```

9. 利用make qemu指令运行xv6：

10. 在命令行中输入`mmaptest`：

```
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
$
```

1.3 实验中遇到的问题和解决办法

1. 问题：在测试过程中出现：

```
fork_test starting
panic: uvmcopy: page not present
QEMU: Terminated
```

- 解决办法：更改`kernel/vm.c`修改`uvmcopy()`函数，`*pte & PTE_V) == 0`时继续执行而不报错。

2. 问题： 在映射地址的确定过程中，需要考虑如何正确处理延迟申请的情况。

- 解决办法：选择从 trapframe 的底部向下生长，同时修改对 uvmunmap 的实现，使得只取消已经映射的页，以解决这个问题。

1.4 实验心得

- 通过本次实验，我深入了解了操作系统中内存映射的原理与实现。学习如何处理系统调用，如何管理虚拟内存，以及如何在内核中维护进程的内存映射信息。
- 在整个实验过程中，我加深了对操作系统内存管理的理解，掌握了操作系统内核中的数据结构和函数调用，提升了我的编程和调试能力。通过不断的实践和探索，我对操作系统的各个组成部分有了

更深入的认识，为我今后在系统编程和操作系统领域的学习和研究打下了坚实的基础。

- 实验中涉及到的 Lazy Allocation（延迟分配）机制是通过页表来实现的。在进行文件映射时，并不会立即分配物理内存，而是在页面首次访问时才触发物理内存的分配和内容填充。此间，页表记录了虚拟地址与物理地址之间的映射关系，Lazy Allocation 的实现需要在虚拟地址首次访问时，操作系统根据页表的映射关系将物理内存进行分配。
- mmap 系统调用的核心是将文件内容映射到虚拟内存中。这涉及到文件系统和虚拟内存的结合。操作系统需要维护文件的相关信息，如打开的文件描述符，然后通过文件描述符来获取文件内容，并将其映射到虚拟内存的合适位置。这种结合使得用户程序可以直接访问文件内容，而无需显式的读写操作。

2. 实验检验得分

1. 在实验目录下创建`time.txt`，填写完成实验时间数
2. 在终端中执行`make grade`

```
(8.4s)
== 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 (232.7s)
== Test time ==
time: OK
Score: 140/140
```