Lab10-Mmp

实验目的

- 增强 xv6 操作系统，使其能够支持文件的内存映射，并通过相应的系统调用实现对虚拟内存区域的详细控制。
- 实现 `mmap` 和 `munmap` 系统调用，实现文件映射功能，实现基本的内存映射操作
    - `mmap` 和 `munmap` 系统调用允许 UNIX 程序对其地址空间进行详细控制。通过这些系统调用，可以实现进程间共享内存、将文件映射到进程地址空间，并支持用户级页面错误处理机制，如垃圾收集算法。
- 实现页表懒惰填充机制，维护进程的虚拟内存区域，实现映射和取消映射功能

实验步骤(1-5是添加系统调用函数的基本步骤)

1. 将 `_mmaptest` 添加到 `Makefile` 的 `UPROGS` 中：

```
UPROGS=\
    ...
    $U/_mmaptest\
```

2. 在 `user/user.h` 中添加两个系统调用函数的声明：

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

3. 在 `user/usys.pl` 中添加调用入口：

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

4. 在 `kernel/syscall.h` 中添加系统调用号：

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

5. 在 `kernel/syscall.c` 中添加系统调用

```
extern uint64 sys_mmap(void);
extern uint64 sys_munmap(void);
static uint64 (*syscalls[])(void) = {
      ...
    [SYS_mmap] sys_mmap,
    [SYS_munmap] sys_munmap,
};
```

6. 在 `kernel/proc.h` 中定义一个对应于虚拟内存区域（VMA）的结构，记录地址、长度、权限、文件等信息。在进程结构中添加一个固定大小的 VMA 数组（如 16 个），用于记录映射区域。

```
#define VMASIZE 16
struct VMA {
    int active;
    uint64 addr;
    int length;
    int prot;
    int flags;
    int fd;
    int offset;
    struct file *fp;
};
// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  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

  // wait_lock must be held when using this:
  struct proc *parent;         // Parent process

  // 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];
};
```

7. 在`kernel/sysfile.c`中实现函数`sys_mmap()`，将文件映射到进程地址空间

```
uint64 sys_mmap(void)
{
    int length, prot, flags, fd, offset;
    uint64 addr;
    struct file *fp;
    struct proc *p = myproc();

    // 处理传入参数
    argaddr(0, &addr);        // 获取 addr 参数
    argint(1, &length);       // 获取 length 参数
    argint(2, &prot);         // 获取 prot 参数
    argint(3, &flags);        // 获取 flags 参数
    argfd(4, &fd, &fp);       // 获取文件描述符 fd 和文件指针 fp
    argint(5, &offset);       // 获取 offset 参数

    // 参数检查
    if (!(fp->writable) && (prot & PROT_WRITE) && (flags == MAP_SHARED)) {
        // 如果文件不可写，并且内存保护需要写权限且标志为共享映射，则返回错误
        return -1;
    }

    // 将 length 向上取整到页面大小
    length = PGROUNDUP(length);
    if (p->sz + length > MAXVA) {
        // 如果映射后进程地址空间超出最大虚拟地址，则返回错误
        return -1;
    }

    // 遍历进程的虚拟内存区域（VMA）数组，找到一个未使用的区域
    for (int i = 0; i < VMASIZE; i++) {
        if (p->vma[i].active == 0) {
            // 标记该区域为已使用
            p->vma[i].active = 1;

            // 直接映射到进程当前大小（sz）对应的虚拟地址
            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].fp = fp;
            p->vma[i].offset = offset;

            // 增加文件引用计数，以防止文件被关闭
            filedup(fp);

            // 更新进程的大小
            p->sz += length;

            // 返回映射的虚拟地址
```

```
            return p->vma[i].addr;
        }
    }
    // 如果没有找到可用的 VMA 区域，返回错误
    return -1;
}
```

8. 在 `kernel/trap.c` 中修改 `usertrap()` 函数以实现了缺页异常处理中的物理页懒分配，确保在需要时为虚拟地址分配物理页面并读取文件内容。文件上面包含头文件 `#include "fcntl.h"`

```
else if (r_scause() == 13 || r_scause() == 15) {
        // supervisor interrupt exception code
        uint64 scause = r_scause();
        // the faulting virtual address
        uint64 addr = r_stval();
        if (addr >= MAXVA) {
          setkilled(p);
          goto out;
        }

        struct VMA* vma = 0;
        for (int i = 0; i < VMASIZE; ++i) {
          if (p->vma[i].active == 0) continue;

          uint64 begin = (uint64)p->vma[i].addr;
          uint64 end = begin + p->vma[i].length;
          if (addr >= begin && addr < end) {
            vma = &p->vma[i];
            break;
          }
        }

        if (vma == 0) {
          printf("usertrap: the faulting virtual address %p is not in the VMA\n",
addr);
          setkilled(p);
          goto out;
        }

        if (scause == 13 && vma->fp->readable == 0) {
          printf("usertrap: the file is unreadable\n");
          setkilled(p);
          goto out;
        }

        if (scause == 15 && vma->fp->writable == 0) {
          printf("usertrap: the file is unwritable\n");
          setkilled(p);
          goto out;
        }

        void* pa = kalloc();
        if (pa == 0) {
          printf("usertrap: unable to allocate memory\n");
          setkilled(p);
          goto out;
        }
        memset(pa, 0, PGSIZE);

        addr = PGROUNDDOWN(addr);
```

```
        int perm = 0;
        // mappages will set PTE_V
        perm |= PTE_U;
        if (vma->prot & PROT_READ)
          perm |= PTE_R;
        if (vma->prot & PROT_WRITE)
          perm |= PTE_W;
        if (vma->prot & PROT_EXEC)
          perm |= PTE_X;
        if (mappages(p->pagetable, addr, PGSIZE, (uint64)pa, perm) != 0) {
          kfree(pa);
          setkilled(p);
          goto out;
        }
        ilock(vma->fp->ip);
        if (readi(vma->fp->ip, 1, addr, addr - (uint64)vma->addr + vma->offset,
PGSIZE) == 0) {
          iunlock(vma->fp->ip);
          uvmunmap(p->pagetable, addr, 1, 1);
          setkilled(p);
          goto out;
        }
        iunlock(vma->fp->ip);
}
```

9. 同时修改 `kernel/vm.c` 中的 `uvmunmap()` 和 `uvmcopy()` 函数的逻辑,正确处理可能未分配物理页的虚拟地址，从而避免错误。

```
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;
        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;
}

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)
            panic("uvmunmap: walk");
        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;
```

```
    }
}
```

10. 在 `kernel/sysfile.c` 中添加函数 `sys_munmap()`,实现了取消虚拟地址的映射关系，并在必要时将修改的内容写回到文件中。

```c
uint64 sys_munmap(void)
{
    int length;
    uint64 addr;

    // 获取参数
    argaddr(0, &addr); // 获取虚拟地址
    argint(1, &length); // 获取解除映射的长度

    struct proc *p = myproc();
    struct VMA *vma = 0;

    // 查找对应的 VMA
    for (int i = 0; i < VMASIZE; i++) {
        if (p->vma[i].active) {
            if (addr == p->vma[i].addr) {
                // 因为 addr 和 length 是页对齐的，所以只要 addr 相等，就一定是同一个
VMA
                vma = &p->vma[i];
                break;
            }
        }
    }

    // 如果没有找到对应的 VMA，返回 0
    if (vma == 0) {
        return 0;
    } else {
        // 更新 VMA 的地址和长度
        vma->addr += length;
        vma->length -= length;

        // 如果是共享映射，需要将内容写回文件
        if (vma->flags & MAP_SHARED)
            filewrite(vma->fp, addr, length);

        // 解除映射
        uvmunmap(p->pagetable, addr, length / PGSIZE, 1);

        // 如果 VMA 的长度为 0，说明已经全部解除映射，需要释放资源
        if (vma->length == 0) {
            fileclose(vma->fp); // 关闭文件
            vma->active = 0; // 标记 VMA 为未使用
        }

        return 0;
    }
}
```

11. 修改`kernel/proc.c`中的`fork()`函数,确保在复制页表时正确处理虚拟内存区域 (VMA) 的复制。

```
// fork时要复制文件内存映射信息
    for (int i = 0; i < VMASIZE; i++) {
        if (p->vma[i].active) {
            memmove(&(np->vma[i]), &(p->vma[i]), sizeof(p->vma[i]));
            filedup(p->vma[i].fp); // refcount++
        }
    }
```

12. 要注意退出进程时要将映射也一并清空，因此需要修改`kernel/exit()`函数,此外，主要在`proc.c`文件上面包含头文件`#include "fcntl.h"`

```
// exit时清空进程的文件内存映射信息
    for (int i = 0; i < VMASIZE; i++) {
        if (p->vma[i].active) {
            if (p->vma[i].flags & MAP_SHARED)
                // 写回磁盘文件
                filewrite(p->vma[i].fp, p->vma[i].addr, p->vma[i].length);
            fileclose(p->vma[i].fp);

            // 取消虚拟内存映射
            uvmunmap(p->pagetable, p->vma[i].addr, p->vma[i].length / PGSIZE, 1);

            // 复位
            p->vma[i].active = 0;
        }
    }
```

13. 测试

运行make qemu后执行mmaptest

```
test mmap two files: OK
mmap_test: ALL OK
fork_test starting
map start
map start
map start
map start
map start
fork_test OK
mmaptest: all tests succeeded
```

运行`make grade`:

```
== Test running mmaptest ==
$ make qemu-gdb
(5.0s)
== 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 (87.0s)
    (Old xv6.out.usertests failure log removed)
== Test time ==
time: OK
Score: 140/140
```

实验中遇到的问题和解决办法

在 `proc.c` 和 `trap.c` 文件中使用了 `MAP_SHARED` 等常量，而这些常量已经定义在 `kernel/fcntl.h` 中。然而，如果在相应文件中没有包含 `fcntl.h`，则会导致编译错误或常量未定义。因此要注意在 `proc.c` 和 `trap.c` 文件的顶部添加 `#include "kernel/fcntl.h"` 以确保这些常量可以被正确引用。

在 `usertrap` 函数中处理页面错误时，访问文件系统需要加锁以确保文件操作的原子性和一致性。忘记加锁会导致数据竞争和文件系统不一致等问题。

```
      ilock(vma->fp->ip);
         if (readi(vma->fp->ip, 1, addr, addr - (uint64)vma->addr + vma->offset,
PGSIZE) == 0) {
           iunlock(vma->fp->ip);
           uvmunmap(p->pagetable, addr, 1, 1);
           setkilled(p);
           goto out;
```

实验心得

Lab10 作为最后一个综合性实验，贯穿了前面实验中的许多内容，帮助我们更全面地理解和掌握操作系统的核心概念和实现方法。Lab10 的核心是实现 `mmap` 和 `munmap` 系统调用，以支持将文件映射到进程的地址空间。这一功能不仅用于进程间的内存共享，还可以优化文件访问性能。

Lab10 涉及了多个前面实验中的重要内容，包括系统调用的实现、页表管理、陷阱处理、进程管理和文件系统操作。

- 在Lab2中，我们学习了如何添加新的系统调用。Lab10通过实现 `mmap` 和 `munmap` 系统调用，进一步巩固了对系统调用机制的理解。
- Lab3 中的页表管理为 Lab10 中的页面映射和管理奠定了基础。Lab10需要在 `usertrap` 函数中处理页面错误，并进行页表更新。
- Lab4 中的陷阱处理与 Lab10 中的页面错误处理直接相关。Lab10则扩展了 `usertrap` 函数，以处理 `mmap` 映射区域的页面错误。
- Lab5 中的写时复制技术在 Lab10 中也有应用。在实现 `mmap` 时，需要确保对共享映射区域的写操作正确处理，即在写时进行物理页的复制。
- 虽然Lab10的重点不在多线程，但多线程的基本概念和同步机制在处理并发文件访问时仍然重要。在Lab10中，我们需要确保对文件的并发访问是安全的。
- Lab8 中的锁机制在Lab10中再次被使用。在处理 `mmap` 和 `munmap` 时，需要确保对文件和内存的访问是原子性的，避免数据竞争和不一致。
- Lab9 中的文件系统操作直接应用于Lab10。在实现 `mmap` 时，我们需要处理文件的读取和写入。

Lab10 作为一个综合性实验，充分利用了前面实验中的知识和技能，帮助我更全面地理解操作系统的核心概念和实现方法。通过这个实验，我不仅掌握了 `mmap` 和 `munmap` 的实现，还深化了对

系统调用、页表管理、陷阱处理、进程管理和文件系统操作的理解。更重要的是，这个实验培养了我分析和解决复杂问题的能力，为进一步的学习和研究奠定了坚实的基础。