Lab - Memory Map 目录

1. mmap (hard)

1.1 实验目的

- 实现 `mmap` 和 `munmap` 系统调用

1.2 实验步骤

1. `mmap` 和 `munmap` 系统调用允许程序对其地址空间进行详细控制。它们可用于在进程之间共享内存、将文件映射到进程地址空间，并作为用户级页面错误方案的一部分。
2. 将 `mmaptest` 添加到 `Makefile` 中：

```
UPROGS=\
    $U/_cat\
    $U/_echo\
    $U/_forktest\
    $U/_grep\
    $U/_init\
    $U/_kill\
    $U/_ln\
    $U/_ls\
    $U/_mkdir\
    $U/_rm\
    $U/_sh\
    $U/_stressfs\
    $U/_usertests\
    $U/_grind\
    $U/_wc\
    $U/_zombie\
    $U/_mmaptest\
```

3. 在 `user/user.h` 中添加函数定义：

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

4. 在user/usys.pl 中添加调用入口：

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

5. 在 `kernel/syscall.h` 中添加系统调用号：

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

6. 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,
};
...
```

7. 对于`mmap()`函数，其原型如下：

```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

其中：

- `addr`始终为零，内核决定映射文件的虚拟地址，`mmap()`将会返回该地址，如果失败则返回`0xffffffffffffffff`。
- `length`是要映射的字节数。
- `prot`指示内存是否应映射为可读、可写或者可执行的。
- `flags`是映射标志，要么为`MAP_SHARED`（映射内存的修改应写回文件），要么为`MAP_PRIVATE`（映射内存的修改不应写回文件）。
- `fd`为映射文件的文件描述符。
- `offset`是地址偏移量，可以假定为零。

对于munmap()函数，原型如下：

```
int munmap(void *addr, size_t length);
```

将会删除指定地址范围内的`mmap`映射。如果进程修改了内存并将其映射为`MAP_SHARED`，则应首先将修改写入文件。同时，`munmap`调用可能只覆盖`mmap`区域的一部分。

8. 根据上述参数，定义`VMA`结构体，在一个进程当中有大小为`VMASIZE`大小的`VMA`数组。

```
#define VMASIZE 16

struct VMA {
    int active;
    uint64 addr;
    int length;
    int prot;
    int flags;
    int fd;
    int offset;
    struct file *fp;
};

struct proc {
    ...
    struct VMA vma[VMASIZE];
};
```

9. 在`kernel/sysfile.c`中添加函数`sys_mmap()`。

对于`sys_map()`函数，应当先解析调用的参数，进行参数的相应检查，并把传入的地址映射到对应地址。然而真正的分配工作并不在这个函数当中，而是采取了`Lazy Allocation`，只有当实际需要写入（读取）物理页时，才会触发一个中断并申请到真正的物理页。因此该函数主要是找到空闲的`VMA`并写入相应参数。

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

    struct proc *p = myproc();
    // 传入参数处理
    argaddr(0, &addr);
    argint(1, &length);
    argint(2, &prot);
    argint(3, &flags);
    argfd(4, &fd, &fp);
    argint(5, &offset);

    // 参数检查
    if (!(fp->writable) && (prot & PROT_WRITE) && (flags == MAP_SHARED)) {
        // 权限冲突，无法写入
        return -1;
    }

    length = PGROUNDUP(length);
    if (p->sz + length > MAXVA) {
        return -1;
    }

    for (int i = 0; i < VMASIZE; i++) {
        if (p->vma[i].active == 0) {
            // 标记为占用
            p->vma[i].active = 1;

            // 直接映射到p->sz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;

            // 文件引用计数增加
            filedup(fp);

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

            // 返回值应当是映射的虚拟地址
            return p->vma[i].addr;
        }
    }
    return -1;
}
```

10. 接下来在`kernel/trap.c`中修改`usertrap()`函数以实现物理页的`Lazy Allocation`，具体实现逻辑见注释：

```
void usertrap(void)
{
    int which_dev = 0;

    ...

    else if (r_scause() == 13 || r_scause() == 15) {
        // 缺页异常
        uint64 va = r_stval(); // 获取缺页地址
        if (va > MAXVA || va >= p->sz) {
            // 越界
            p->killed = 1;
        }
        else {
            struct VMA *pvma = 0;
            for (int i = 0; i < VMASIZE; i++) {
                // 找到对应的VMA
                if (p->vma[i].active == 0)
                    continue;
                if (va >= p->vma[i].addr && va < p->vma[i].addr + p-
>vma[i].length) {
                    pvma = &p->vma[i];
                }
            }

            if (pvma) {
                // 若找到了对应的VMA，则要把对应的虚拟地址分配到物理地址，同时把文件内容读到
物理地址
                va = PGROUNDDOWN(va);
                uint64 pa = (uint64)kalloc();
                if (pa == 0) {
                    // 分配失败
                    p->killed = 1;
                }
                else {
                    memset((void *)pa, 0, PGSIZE);
                    ilock(pvma->fp->ip);                                    // 加锁
                    readi(pvma->fp->ip, 0, pa, va - pvma->addr, PGSIZE); // 读取
文件内容
                    iunlock(pvma->fp->ip);                                  // 解锁

                    // 根据flag参数设置PTE
                    int PTE_flags = PTE_U;
                    if (pvma->prot & PROT_READ)
                        PTE_flags |= PTE_R;
                    if (pvma->prot & PROT_WRITE)
                        PTE_flags |= PTE_W;
                    if (pvma->prot & PROT_EXEC)
                        PTE_flags |= PTE_X;

                    // 映射
                    printf("map start\n");
```

```
                    if (mappages(p->pagetable, va, PGSIZE, pa, PTE_flags) != 0)
{
                        // 映射失败
                        kfree((void *)pa);
                        p->killed = 1;
                    }
                }
            }
        }
    }

    ...

}
```

11. 要注意的是，应该同时修改`kernel/vm.c`中的`uvmunmap()`和`uvmcopy()`函数的逻辑（因为映射到的地址可能并没有真正地被分配物理页，所以在进行`(*pte & PTE_V) == 0`的判断时会返回`True`）：

```
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;
    }
}

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;
}
```

12. 在`kernel/sysfile.c`中添加函数`sys_munmap()`，取消虚拟地址的映射关系：

```
uint64 sys_munmap(void)
{
    int length;
    uint64 addr;
    argaddr(0, &addr);
    argint(1, &length);

    struct proc *p = myproc();
    struct VMA *vma = 0;
    for (int i = 0; i < VMASIZE; i++) {
        // 找到对应的VMA
        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;
            }
        }
    }

    if (vma == 0) {
        return 0;
    }
    else {
        vma->addr += length;
        vma->length -= length;
        if (vma->flags & MAP_SHARED)
            // 如果是共享映射，需要把文件内容写回
            filewrite(vma->fp, addr, length);

        uvmunmap(p->pagetable, addr, length / PGSIZE, 1);
        if (vma->length == 0) {
            // 如果VMA的长度为0，说明已经全部解除映射，需要释放资源
            fileclose(vma->fp);
            vma->active = 0;
        }
        return 0;
    }
}
```

13. 修改`kernel/proc.c`中的`fork()`函数，在进程复制时也要复制映射关系：

```
int fork(void)
{
    ...

    acquire(&np->lock);

    // 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++
        }
    }

    np->state = RUNNABLE;
    release(&np->lock);

    return pid;
}
```

14. 修改exit()函数，在进程退出时清空映射：

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

    ...

    // 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;
        }
    }

    ...
}
```

15. 运行`mmaptest`测试，结果如下：

```
$ mmaptest
mmap_test starting
test mmap f
map start
map start
test mmap f: OK
test mmap private
map start
map start
test mmap private: OK
test mmap read-only
test mmap read-only: OK
test mmap read/write
map start
map start
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
map start
map start
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
```

16. 运行`grade`测试，结果如下：

```
== Test running mmaptest ==
$ make qemu-gdb
(3.9s)
== 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
```

1.3 实验中遇到的问题和解决方法

- **问题**：运行测试时，出现如下错误信息：

  ```
  panic: mappages: remap
  ```

  解决方法：定位问题，发现是在映射时发生的错误，对于`remap`错误，可能是由于占用了已映射的地址，经过检查发现是在设置`PTE`位时传入`flags`导致的错误，改为传入`prot`即可。

- **问题**：把物理内容写入文件时发生错误

  解决方法：文件的使用也需要锁的获取与释放以避免冲突，使用

  ```
  ilock(pvma->fp->ip);                                    // 加锁
  readi(pvma->fp->ip, 0, pa, va - pvma->addr, PGSIZE); // 读取文件内容
  iunlock(pvma->fp->ip);                                  // 解锁
  ```

  完成读取过程。

1.4 实验心得

通过完成本实验，我深入了解了 mmap 和 munmap 系统调用的实现原理，以及如何在 xv6 操作系统中进行内存映射，也对文件相关的机制有了更深入的理解。这个实验不仅要将文件映射到进程的地址空间中，实现虚拟内存和物理内存之间的映射，也要使用惰性分配机制，只有当发生中断需要实际物理页面时再去进行实际分配，采取类似于`COW`实验的思想，能够大大提高资源的利用效率，避免不必要的内存消耗，还能使`mmap()`对大文件的映射更加高效。在实现的过程中还要考虑其他系统调用的影响，需要在`fork`系统调用中做一些额外的工作，也需要增加例如`struct file`的引用计数，确保子进程可以正确访问共享的内存。总之这个实验很大程度上锻炼了我的能力。