本科生实验报告

实验课程：操作系统原理实验

实验名称：CopyOnWrite

专业名称：计算机科学与技术

学生姓名：数据删除

学生学号：数据删除

实验成绩：

报告时间：2025 年 6 月 12 日

1. 实验要求

在我们的 fork 实现当中，我们会复制父进程的所有资源到子进程中。这样的做法虽然简单，但是开销较大。注意到如下特点：

- 父进程中有些资源是只读的，并不会做修改。因此任何时刻，父子进程的只读资源都是相同的。因此父子进程的只读资源可以共享。
- 父进程的有些资源在子进程中可能不会被再次用到，或者是暂时不会被用到。

因此，为了减少资源的开销，我们可以实现 copy-on-write（写时复制）机制。在写时复制机制中，fork 函数不会为子进程复制任何父进程任何物理页，而是父进程共享相同的物理空间，父子共享的部分被标记为只读的。当父/子进程需要修改共享的内容的时候，会产生一个中断。然后，这个中断处理函数会为子进程分配物理空间，然后将需要修改的数据复制到刚刚为子进程分配的物理空间中。中断处理完成后，父/子进程才会修改刚刚需要的数据。

现在，同学们需要在本学期自己的实验基础上实现基于 copy-on-write 机制的 fork 函数。在实现 copy-on-write 后，同学们需要自行提供测例来测试你的 copy-on-write。根据测试方法和输出结果来解释你的 copy-on-write 的正确性。最后将结果截图并说说你是怎么做的。

2. 实验过程

参考 linux 的 cow 实现，基本思路是：

1. fork 复制页表时，将父进程所有 pte 的 R/W 位置 0，即只读，子进程使用相同的 pte。

2. 父进程或子进程试图对页表地址进行写操作时，会产生页面故障，拉起中断处理函数。

3. 中断处理函数申请新的物理页，并修改页表项使访问的虚拟页指向新的物理页。

4. 退出中断回到访问内存的指令重新执行，此时应该不会出现页面故障。

实现过程中存在一系列细节：

原来的只读页怎么处理？如果父进程和子进程都写入而复制了页表，只读页是否会悬垂？

会。所以需要进行引用计数。若只读页计数为 1，则不需要复制。释放进程时将所有引用的页的引用计

数减 1，回收没有被引用的物理页。

页面故障码是多少？中断处理函数的调用约定是什么？

查阅 Intel IA-32 文档得知故障码为 14，硬件调用中断处理时，除了会把基本的上下文压入栈中，还会额外压入一个故障码：

> 异常错误码：有的（特殊格式）。处理器向页故障处理程序提供两个信息项来帮助诊断异常并恢复它：
>
> - 一个栈中的错误码。页故障的错误码不同于其它异常（参看图 5-7）。错误码告诉异常处理程序四件事：
>
> ——P 标志表明异常是由于一个不存在页（0）还是访问权限违例或是使用了保留位（1）。
>
> ——W/R 标志表明引起异常的内存访问是读（0）还是写（1）。
>
> ——U/S 标志表明异常发生时处理器是在用户态（1）执行还是在管理态（0）执行。
>
> ——RSVD 标志表明处理器在页目录的保留位中检测到了 1，此时控制寄存器 CR4 的 PSE 或者 PAE 标志都置为 1。(PSE 标志只在 Pentium 4、Intel Xeon、P6 系列和 Pentium 等处理器中可用，PAE 标志只在 Pentium 4、Intel Xeon、P6 系列等处理器中可用，在早期的 IA-32 处理器中，RSVD 标志位是保留的。)

这个故障码对 COW 的用处不大，但我们需要手动将其出栈。

至于导致页面故障的虚拟地址则会保留在 cr2 寄存器中：

> - CR2 寄存器的内容。处理器把产生异常的 32 位线性地址加载到 CR2 寄存器中。页故障处理程序可以利用这个地址来确定相应的页目录表项和页表项。在页故障处理程序中也可能发生另外一次页故障。处理程序应该在第二次页故障可能发生之前保存 CR2 寄存器的内容[1]。如果页故障是由于页级保护引起的，则当故障发生时，置位页目录表项的访问位。IA-32 处理器如何解释相应页目录表项中的访问位与模型相关，而且在架构上也没有定义。

需要将其压入栈中交给中断处理函数处理。

页面故障可能不仅仅是 COW 引起的，还可能是其他故障，怎么只对 COW 引起的页面故障进行复制操作？

首先可以读 R/W 位识别；其次 pte 的 9~11 位（即 AVL 位）是操作系统定义的标志位，可以使用其中一位标识该页是否是 COW 的只读页。

除此之外还有一系列细节，最重要的一个是：在修改完页表后，务必重新载入页表到 cr3 中（即便 cr3 指向的页目录表地址并没有变化），原因是 CPU 的 TLB 中还可能保留有旧页表的映射，如果不进行刷新可能还会访问原先的地址。

3. 关键代码

构建物理页引用计数：使用静态数组维护，decRefCount 需要考虑到并发，因此为其加锁。

```
inline int page_ref_count[MAX_PHYSICAL_PAGE]; // 物理页引用计数
inline SpinLock page_ref_lock;

void MemoryManager::incRefCount(uint paddr) {
    uint pageIndex = (paddr - 0x100000) / PAGE_SIZE;
    if (pageIndex < MAX_PHYSICAL_PAGE) {
        page_ref_count[pageIndex]++;
    }
}

int MemoryManager::decRefCount(uint paddr) {
    uint pageIndex = (paddr - 0x100000) / PAGE_SIZE;
    if (pageIndex < MAX_PHYSICAL_PAGE && page_ref_count[pageIndex] > 0) {
        page_ref_lock.lock();
        int res = --page_ref_count[pageIndex];
        page_ref_lock.unlock();
        return res;
    }
    return -1;
}

void MemoryManager::setRefCount(uint paddr, int count) {
    uint pageIndex = (paddr - 0x100000) / PAGE_SIZE;
    if (pageIndex < MAX_PHYSICAL_PAGE) {
        page_ref_count[pageIndex] = count;
    }
}

int MemoryManager::getRefCount(uint paddr) {
    uint pageIndex = (paddr - 0x100000) / PAGE_SIZE;
    if (pageIndex < MAX_PHYSICAL_PAGE) {
        return page_ref_count[pageIndex];
    }
    return 0;
```

```
}
```

修改 copyProcess，不复制整个页表而是先共享父进程页表：

```
bool ProgramManager::copyProcess(PCB* parent, PCB* child) {
    // 复制进程 0 级栈
    auto childpss = (ProcessStartStack*)((int)child + PAGE_SIZE -
sizeof(ProcessStartStack));
    auto parentpss = (ProcessStartStack*)((int)parent + PAGE_SIZE -
sizeof(ProcessStartStack));
    memcpy(childpss, parentpss, sizeof(ProcessStartStack));
    // 设置子进程的返回值为 0
    childpss->eax = 0;

    // 准备执行 asm_switch_thread 的栈的内容
    child->stack = (int*)childpss - 7;
    child->stack[0] = 0;
    child->stack[1] = 0;
    child->stack[2] = 0;
    child->stack[3] = 0;
    child->stack[4] = (int)asm_start_process;
    child->stack[5] = 0;            // asm_start_process 返回地址
    child->stack[6] = (int)childpss; // asm_start_process 参数
    // 设置子进程的 PCB
    child->status = ProgramStatus::READY;
    child->parentPid = parent->pid;
    child->priority = parent->priority;
    child->ticks = parent->ticks;
    child->ticksPassedBy = parent->ticksPassedBy;
    strcpy(child->name, parent->name);

    // 复制用户虚拟地址池
    int bitmapLength = parent->userVirtual.resources.length;
    int bitmapBytes = ceil(bitmapLength, 8);
    memcpy(child->userVirtual.resources.bitmap, parent->userVirtual.resources.bitmap,
bitmapBytes);

    // 从内核中分配一页作为中转页
    // uint buffer = memoryManager.allocatePages(AddressPoolType::KERNEL, 1);
    // if (!buffer) {
    //  child->status = ProgramStatus::DEAD;
    //  return false;
    // }

    // 子进程页目录表物理地址
    uint childPageDirPaddr = memoryManager.vaddr2paddr(child->pageDirectoryAddress);
    // 父进程页目录表物理地址
    uint parentPageDirPaddr =
memoryManager.vaddr2paddr(parent->pageDirectoryAddress);
```

```
    // 子进程页目录表指针(虚拟地址)
    uint* childPageDir = (uint*)child->pageDirectoryAddress;
    // 父进程页目录表指针(虚拟地址)
    uint* parentPageDir = (uint*)parent->pageDirectoryAddress;

    // 子进程页目录表初始化
    memset((void*)child->pageDirectoryAddress, 0, 768 * 4);

    // 复制页目录表
    for (int i = 0; i < 768; ++i) {
        // 无对应页表
        if (!(parentPageDir[i] & 0x1)) {
            continue;
        }

        // 从用户物理地址池中分配一页，作为子进程的页目录项指向的页表
        uint paddr = memoryManager.allocatePhysicalPages(AddressPoolType::USER, 1);
        if (!paddr) {
            child->status = ProgramStatus::DEAD;
            return false;
        }
        // 页目录项
        uint pde = parentPageDir[i];
        // 构造页表的起始虚拟地址
        uint* pageTableVaddr = (uint*)(0xffc00000 + (i << 12));

        asm_update_cr3(childPageDirPaddr); // 进入子进程虚拟地址空间

        childPageDir[i] = (pde & 0x00000fff) | paddr;
        memset(pageTableVaddr, 0, PAGE_SIZE);

        asm_update_cr3(parentPageDirPaddr); // 回到父进程虚拟地址空间
    }

    // 复制页表和物理页
    for (int i = 0; i < 768; ++i) {
        // 无对应页表
        if (!(parentPageDir[i] & 0x1)) {
            continue;
        }

        // 计算页表的虚拟地址
        uint* pageTableVaddr = (uint*)(0xffc00000 + (i << 12));

        // 复制物理页
        for (int j = 0; j < 1024; ++j) {
            // 无对应物理页
            if (!(pageTableVaddr[j] & 0x1)) {
```

```
                continue;
            }

            // 从用户物理地址池中分配一页，作为子进程的页表项指向的物理页
            // uint paddr = memoryManager.allocatePhysicalPages(AddressPoolType::USER,
1);
            // if (!paddr) {
            //  child->status = ProgramStatus::DEAD;
            //  return false;
            // }

            // 构造物理页的起始虚拟地址
            uint pageVaddr = ((i << 22) + (j << 12));
            // 页表项

            uint pte = pageTableVaddr[j];

            uint physicalPageAddr = pte & 0xfffff000; // 获取物理页地址

            memoryManager.incRefCount(physicalPageAddr); // 增加物理页引用计数

            // printf("copy ref address: %x\n", physicalPageAddr);
            // printf("vaddr: %x, paddr: %x\n", pageVaddr, physicalPageAddr);

            // 复制出父进程物理页的内容到中转页
            // memcpy((void*)buffer, pageVaddr, PAGE_SIZE);

            asm_update_cr3(childPageDirPaddr); // 进入子进程虚拟地址空间

            // printf("original pte: %x\n", pageTableVaddr[j]);
            // pageTableVaddr[j] = (pte & 0x00000fff) | paddr;
            pageTableVaddr[j] = (pte & (~2)) | (1 << 9); // 只读并置 COW 位
            // 从中转页中复制到子进程的物理页
            // memcpy(pageVaddr, (void*)buffer, PAGE_SIZE);

            asm_update_cr3(parentPageDirPaddr); // 回到父进程虚拟地址空间

            pageTableVaddr[j] &= (~2); // 对父进程 pte 进行相同的操作
            pageTableVaddr[j] |= (1 << 9);
        }
    }

    // 归还从内核分配的中转页
    // memoryManager.releasePages(AddressPoolType::KERNEL, buffer, 1);
    return true;
}
```

页面故障处理函数：

```
asm_page_fault_handler:
    cli
    pushad
    push ds
    push es
    push fs
    push gs

    mov eax, [esp + 12 * 4] ; 获取错误码
    mov ebx, cr2            ; 获取导致页错误的地址

    push ebx
    push eax

    call c_page_fault_handler

    add esp, 2 * 4          ; 清理参数
    pop gs
    pop fs
    pop es
    pop ds
    popad

    add esp, 4              ; 清理错误码
    sti
    iret
```

```
extern "C" void c_page_fault_handler(uint errorCode, uint addr) {
    printf("Page fault at address: 0x%x, error code: 0x%x\n", addr, errorCode);
    addr = addr & 0xfffff000;
    uint& pte = *toPTE(addr);
    // printf("Page table entry: 0x%x\n", pte);
    if (!((pte & (1 << 9)) && !(pte & 2))) {
        printf("not readonly or cow\n");
        asm_halt();
    }
    uint paddr_original = memoryManager.vaddr2paddr(addr);
    if (memoryManager.getRefCount(paddr_original) == 1) {
        printf("only one reference, no need to copy\n");
        pte |= 2;
        return;
    }
    memoryManager.decRefCount(paddr_original);
    // printf("Original physical address: 0x%x\n", paddr_original);

    // 分配一个新的物理页，因为物理页必须挂在页表上才能访问并复制数据，所以同时分配一个中转虚拟页
    uint addr_new = memoryManager.allocatePages(AddressPoolType::USER, 1);
```

```
    if (!addr_new) {
        printf("Failed to allocate new physical page, halting.\n");
        asm_halt();
    }
    memcpy((void*)addr_new, (void*)addr, PAGE_SIZE);

    uint paddr_new = memoryManager.vaddr2paddr(addr_new);
    // 将原虚拟页重新映射到新物理页
    memoryManager.connectPhysicalVirtualPage(addr, paddr_new);
    // 释放中转虚拟页
    memoryManager.releaseVirtualPages(AddressPoolType::USER, addr_new, 1);
    // 激活页表刷新 TLB
    programManager.activateProgramPage(programManager.running);
    printf("page fault handled\n");
}
```

测试用例一：使用 lab8 的 fork 测例，检查 COW 是否对用户完全透明：

```
void first_process() {
    int pid = fork();
    if (pid == -1) {
        printf("can not fork\n");
    }
    else {
        if (pid) {
            printf("I am father, fork return: %d\n", pid);
        }
        else {
            printf("I am child, fork return: %d, my pid: %d\n", pid,
programManager.running->pid);
        }
    }
}
```

测试用例二：在用例一的基础上，额外再申请一个页，子进程会修改该页内的数据，检查父进程数据是

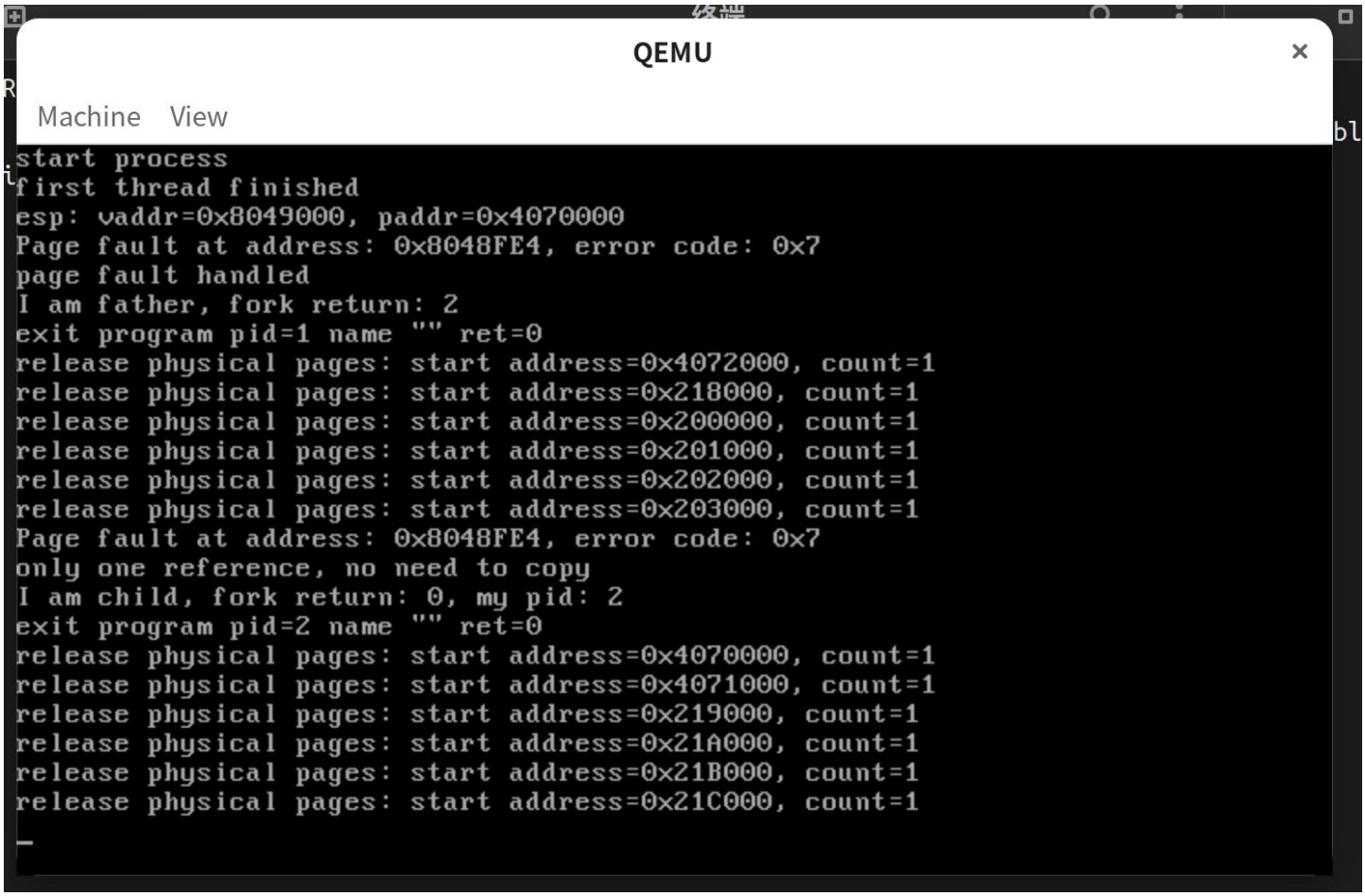
否被修改、COW 前后页的物理地址是否有变化：

```
void second_process() {
    int* p = (int*)memoryManager.allocatePages(AddressPoolType::USER, 1);
    if (!p) {
        printf("Failed to allocate memory\n");
        return;
    }
    *p = 114514;
    int pid = fork();
    if (pid == -1) {
        printf("can not fork\n");
    }
    else {
```

```
        if (pid) {
            printf("I am father, fork return: %d\n", pid);
            wait(nullptr);
            printf("child: p = %d, paddr=0x%x\n", *p,
memoryManager.vaddr2paddr((uint)p));
        }
        else {
            printf("I am child, fork return: %d, my pid: %d\n", pid,
programManager.running->pid);
            printf("child: p = %d, paddr=0x%x\n", *p,
memoryManager.vaddr2paddr((uint)p));
            *p = 1919810;
            printf("child: p = %d, paddr=0x%x\n", *p,
memoryManager.vaddr2paddr((uint)p));
            exit(0);
        }
    }
}
```

4. 实验结果

测例一：

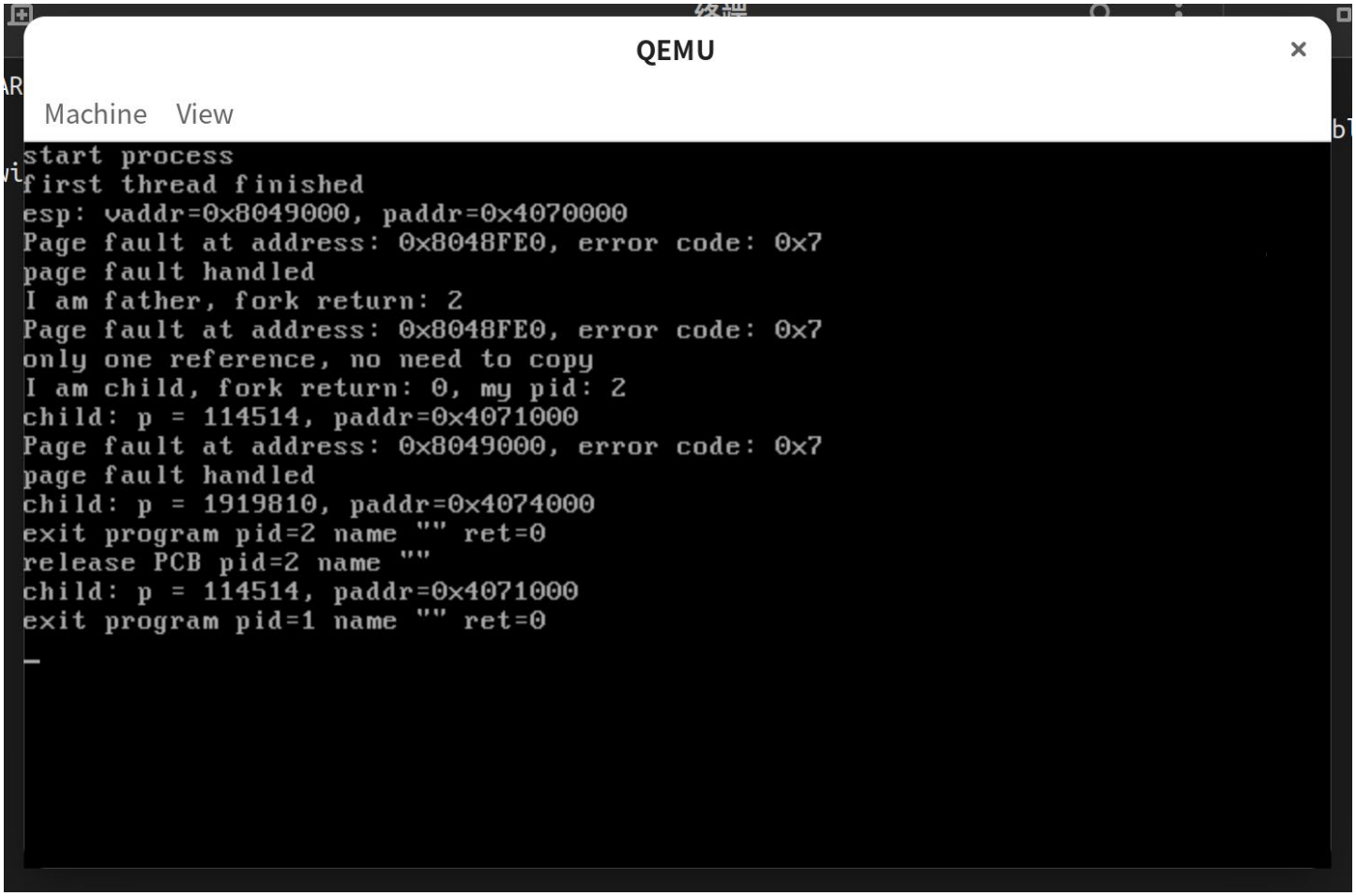


error code 为 0x7，说明错误原因是访问权限违例、写模式、用户状态，符合预期。

触发了两次页面故障，一次成功复制，另外一次因为页面引用数为 1，因此可以直接复用该页，直接将 R/W 位重新置 1 即可。

测试输出物理页释放过程，同一个物理页只被释放一次，结果正确。

测例二：



子进程对页面的修改不影响父进程；输出父进程和子进程的页的物理地址均相等，子进程写入后，页的物理地址发生变化，符合预期。

5. 总结

本次实验完成了基于 COW 机制的 fork 函数实现。通过修改页表，将父进程的页表项设置为只读，子进程共享相同的页表。当父进程或子进程尝试写入只读页面时，会触发页面故障，此时分配新的物理页面，并将数据复制到新页面，从而实现写时复制。实现过程经历了比较多的挑战，但同时也学到了很多新的知识，为学习操作系统提供了宝贵的经验。