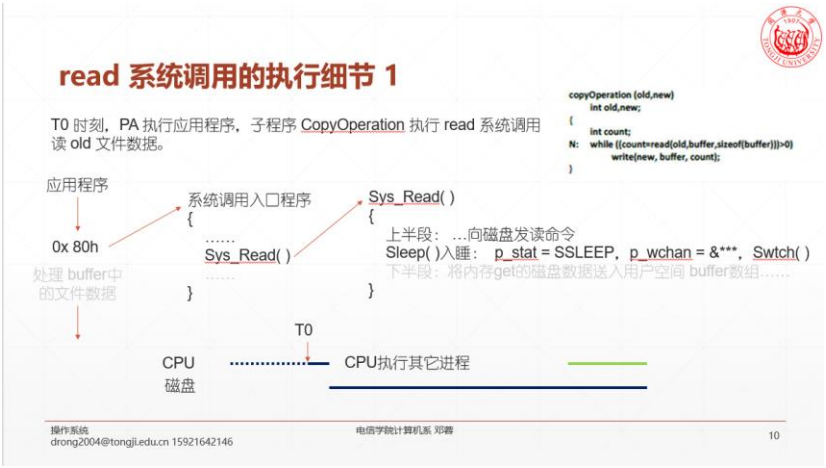
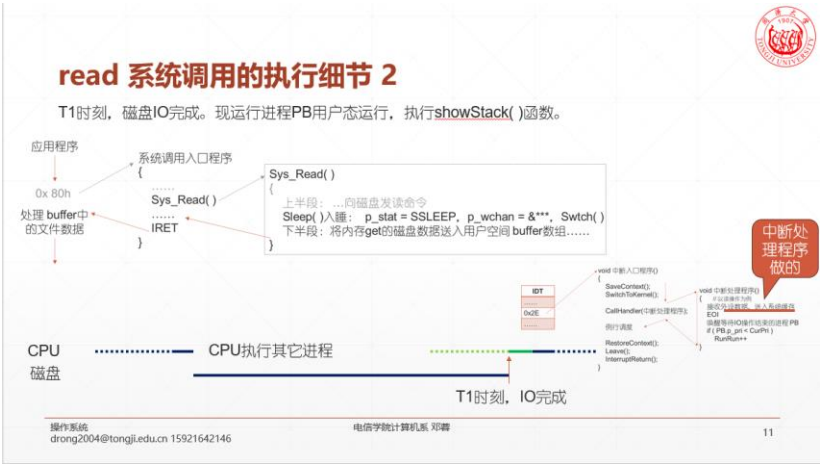
同济大学计算机系操作系统课程作业

进程管理 一　　2023-11-13

学号　　2151769　　姓名　吕博文

一、(1) 注释 PPT10~11，写出 read 系统调用的执行细节（2）画 2 张图，补全随后 write 系统调用的执行细节。不必面面俱到，不清楚的地方红笔标出来，本周四前完成。





1、

（1） read 函数的 UNIX V6++实现如下：

```
/*
读文件系统调用c库封装函数
fd：打开进程打开文件号
ubuf：目的区首地址
nbytes：要求读出的字节数
返回值：读取的实际数目（字节）
*/
int read(int fd, char* buf, int nbytes)
{
    int res;
    __asm__ __volatile__ ("int $0x80":"=a"(res):"a"(3),"b"(fd),"c"(buf),"d"(nbytes));
    if ( res >= 0 )
        return res;
    return -1;
}
```

系统调用 Sys_Read 函数如下：

```
/*   3 = read     count = 2    */
int SystemCall::Sys_Read()
{
    FileManager& fileMgr = Kernel::Instance().GetFileManager();
    fileMgr.Read();

    return 0;    /* GCC likes it ! */
}
```

```
void FileManager::Read()
{
    /* 直接调用Rdwr()函数即可 */
    this->Rdwr(File::FREAD);
}
```

```
void FileManager::Rdwr( enum File::FileFlags mode )
{
    File* pFile;
    User& u = Kernel::Instance().GetUser();

    /* 根据Read()/Write()的系统调用参数fd获取打开文件控制块结构 */
    pFile = u.u_ofiles.GetF(u.u_arg[0]);    /* fd */
    if ( NULL == pFile )
    {
        /* 不存在该打开文件，GetF已经设置过出错码，所以这里不需要再设置了 */
        /*  u.u_error = User::EBADF;    */
        return;
    }


    /* 读写的模式不正确 */
    if ( (pFile->f_flag & mode) == 0 )
    {
        u.u_error = User::EACCES;
        return;
    }

    u.u_IOParam.m_Base = (unsigned char *)u.u_arg[1];    /* 目标缓冲区首址 */
    u.u_IOParam.m_Count = u.u_arg[2];        /* 要求读/写的字节数 */
    u.u_segflg = 0;      /* User Space I/O，读入的内容要送数据段或用户栈段 */
```

```
/* 管道读写 */
if(pFile->f_flag & File::FPIPE)
{
    if ( File::FREAD == mode )
    {
        this->ReadP(pFile);
    }
    else
    {
        this->WriteP(pFile);
    }
}
else
/* 普通文件读写 ，或读写特殊文件。对文件实施互斥访问，互斥的粒度：每次系统调用。
为此Inode类需要增加两个方法：NFlock()、NFrele()。
这不是V6的设计。read、write系统调用对内存i节点上锁是为了给实施IO的进程提供一致的文件视图。*/
{
    pFile->f_inode->NFlock();
    /* 设置文件起始读位置 */
    u.u_IOParam.m_Offset = pFile->f_offset;
    if ( File::FREAD == mode )
    {
        pFile->f_inode->ReadI();
    }
    else
    {
        pFile->f_inode->WriteI();
    }
```

```
    /* 根据读写字数，移动文件读写偏移指针 */
    pFile->f_offset += (u.u_arg[2] - u.u_IOParam.m_Count);
    pFile->f_inode->NFrele();
}

/* 返回实际读写的字节数，修改存放系统调用返回值的核心栈单元 */
u.u_ar0[User::EAX] = u.u_arg[2] - u.u_IOParam.m_Count;
```

（2） 系统调用 Read 的具体实现细节：

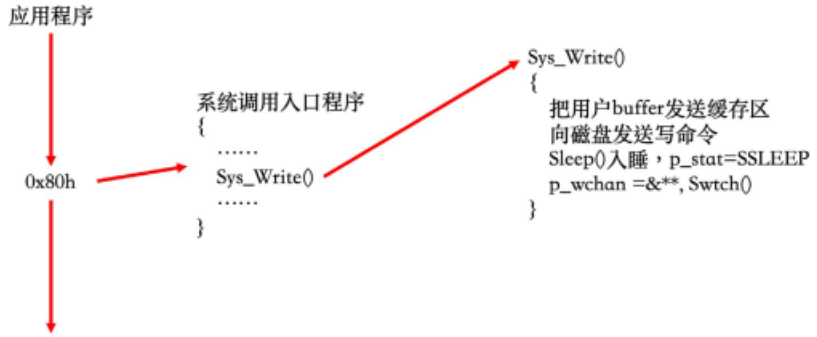
在具体的 read（）函数实现中，首先，根据系统调用参数 fd 获取打开文件控制块结构，并判断文件打开是否成功，读写模式是否正确，之后，首先记录目标缓冲区首地址，要求读入的字节数，以及读入内容送达地址，之后根据是否为管道读写分类实现，如果是管道读写直接调用函数读入即可，如果是普通文件读写或是特殊文件，则对普通文件或特殊文件实施互斥访问，通过调用文件对应的 Inode 的 NFlock 和 NFrele 方法实现。设置文件起始读位置为文件控制块的偏移量（f_offset）。如果是读操作，调用 Inode 的 ReadI 方法；如果是写操作，调用 Inode 的 WriteI 方法。

pFile->f_offset += (u.u_arg[2] - u.u_IOParam.m_Count);: 根据读写字节数，移动文件读写偏移指针。

pFile->f_inode->NFrele();: 释放对 Inode 的锁定，表示文件读写操作完成。

u.u_ar0[User::EAX] = u.u_arg[2] - u.u_IOParam.m_Count;: 返回实际读写的字节数，修改存放系统调用返回值的核心栈单元（EAX 寄存器）。

2、应用程序使用 int 0x80 进行系统调用，在系统调用入口调用 Sys_write()执行磁盘命令，随后该进程睡觉，切换进程



写任务完成之后，当前进程中断，进入对应的入口程序，在中断处理程序时唤醒进程并回送 EOI 信号，进行例行调度。

二、修改 Kernel.cpp 中的 GetUser( )函数，用 ESP 寄存器计算得到现运行进程 user 结构的起始地址（调通系统和我说一声，加分）。评价系统性能。

```
166 User& Kernel::GetUser()
167 {
168     return *(User*)USER_ADDRESS;
169 }
```

User 结构与核心栈同时位于一个 4K 的字节块中，User 结构的首地址位于该块的顶部，而此时 EBP 一定指向核心栈，所以 EBP 和 User 结构首地址一定共享高 20 位，而 User 结构首地址低 12 位一定是 0，所以只需要将 EBP 的低 12 位清零即可。

```
User& Kernel::GetUser()
{
    //return *(User*)USER_ADDRESS;
    unsigned long user_addr;
    __asm__ __volatile__(" movl %%esp, %0" : "=r"(user_addr))
    user_addr = user_addr & 0xfffff000;
    return *(User*)user_addr;
}
```