**Project2-User Programs**

**一、任务需求分析**

这个项目将使pintos可以加载并执行用户程序，并且为用户程序提供系统调用。

Project2 需要完成的任务有四个：

Task1 Process Termination Messages

Task2 Argument Passing

Task3 System Calls

Task4 Denying Writes to Executables

由于处于时间较为紧张的复习周，所以我们决定完成第一个、第三个和第四个Task，任务分工大致明确，但是每项任务都是大家进行讨论研究的实验方案，最终的实现是由某位同学来完成的。

**二、设计方案**

**Task1 Process Termination Messages（进程终止信息）**

pintos在启动时通过调用run\_task（）

--> process\_wait(process\_execute(task))来执行用户程序

在用户执行过程中，pintos系统线程一直处在process\_wait状态来等待用户线程执行完毕，从而完成pintos运行，但对于用户线程是正常完成还是异常中断并不知情

我们需要实现在用户线程完毕时在屏幕上输出是否正常完成的相关信息,即用户程序退出时返回退出代码，正确退出为0，遇到错误被终止则返回非零值

要求：1.在进程结束时输出退出代码（main函数的返回值，或者异常退出代码）

注意：用户进程结束时输入退出代码，核心线程返回时不输入。

输出格式被规定如下： 

printf (“%s: exit(%d)\n”,..);

**Task3 System Calls（系统调用）**

要求：

（1）实现以下系统调用： 

pfn[SYS\_WRITE]=IWrite; //printf和写文件需要。  pfn[SYS\_CREATE]=ICreate; //创建文件  pfn[SYS\_CLOSE]=IClose; //关闭文件  pfn[SYS\_FILESIZE]=IFileSize; //返回文件大小   pfn[SYS\_WAIT]=IWait; //等待子进程结束

pfn[SYS\_SEEK]=ISeek; //移动文件指针  pfn[SYS\_TELL]=ITell; //返回文件指针位置  pfn[SYS\_HALT]=IHalt; //关机 要想完成以上系统调用，还参考文件有：src/lib/user/syscall.c 了解每个系统调用的形式。 src/lib/syscall-nr.h 了解每个系统调用号。

**Task4 Denying Writes to Executables**

当一个可执行文件正在被执行时，不允许其被修改

使用file\_deny\_write()来避免对正打开文件的写入

关闭一个文件的时候激活写的权限

在可执行文件执行过程中，文件一直保持打开状态

使用锁或者信号量来保证操作的互斥

在start\_process函数中加入

    t->FileSelf=filesys\_open(token);

file\_deny\_write(t->FileSelf);

其中FileSelf变量是要在struct thread结构中添加的。

进程退出时就解除： 在process\_exit()中加入

      if(cur->FileSelf!=NULL)  //撤销对自己人deny\_write       {

          file\_allow\_write(cur->FileSelf);           file\_close (cur->FileSelf);       }

注意：所有系统调用的返回值都放到用户的eax寄存器中。 取出参数时要对用户栈指针作详细的检查，是否越界，越界则直接调用Exit\_Status(-1)中止用户进程。

**三、具体实现**

**Task1 Process Termination Messages（进程终止信息）**

实现方法：

1. 既然要打印返回值，就得用一个变量保存返回值，于是在struct thread 结构中加入一个变量回保存返回值： int ret\_status;



在init\_thread()函数中初始化为0（这里可以不用初始化）。

2. 在线程退出里要保存其返回值到ret\_status中，这个将在系统调用里的exit函数中保存，这里先不考虑。 在什么地方加入printf()呢？ 每个线程结束后，都要调用thread\_exit()函数，如果是加载了用户进程，在thread\_exit()函数中还会调用process\_exit()函数， 在process\_exit()函数中，如果是用户进程，那么其页表一定不为NULL,而核心进程页表一定为NULL,即只有用户进程退出时if(pd!=NULL){ }就会成立，所以在大括号中加入:

printf (“%s: exit(%d)\n”,cur->name,cur->ret);

其中cur=thread\_current();即当前线程的struct thread 指针。

**Task3 System Calls（系统调用）**

实现方法：

（1） 搭建框架

用一个数组保存各函数名，数组下标就是系统调用号。

在syscall\_init()函数中初始化数组pfn[]为NULL

在syscall\_handler()函数中依据系统调用号调用相函数。

typedef void (\*CALL\_PROC)(struct intr\_frame\*);

CALL\_PROC pfn[MAXCALL];

void syscall\_init (void) {

intr\_register\_int (0x30, 3, INTR\_ON, syscall\_handler, "syscall");

int i;

for(i=0;i<MAXCALL;i++) pfn[i]=NULL; }

static void syscall\_handler (struct intr\_frame \*f /\*UNUSED\*/) {

if(!is\_user\_vaddr(f->esp))

ExitStatus(-1);

int No=\*((int \*)(f->esp));

if(No>=MAXCALL||MAXCALL<0)

{

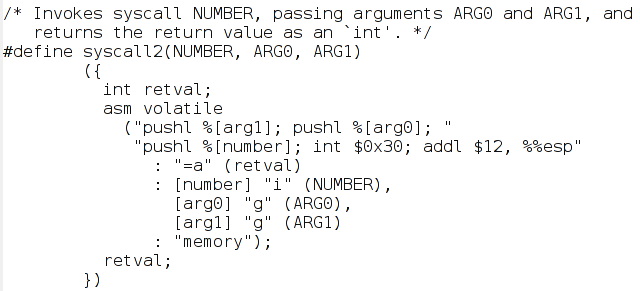
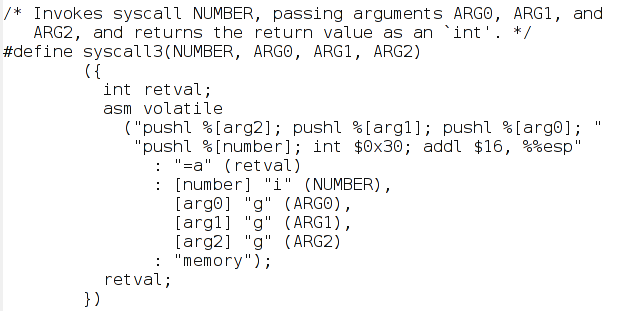
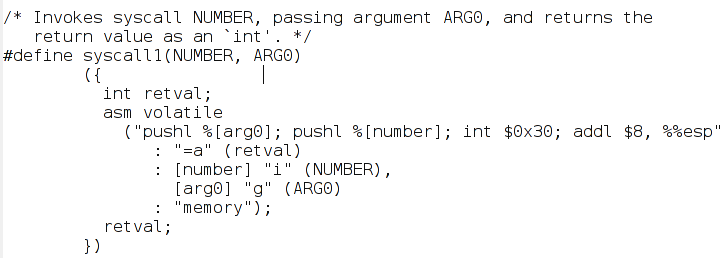
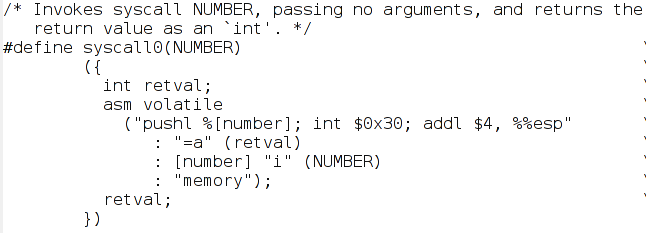
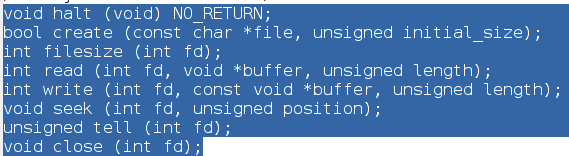
printf("We don't have this System Call!\n"); ExitStatus(-1); }

if(pfn[No]==NULL) {

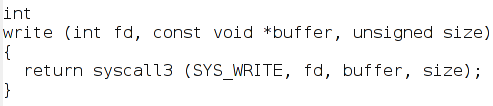
printf("this System Call %d not Implement!\n",No); ExitStatus(-1); }

pfn[No](f); }

(2)每一个系统调用的实现。

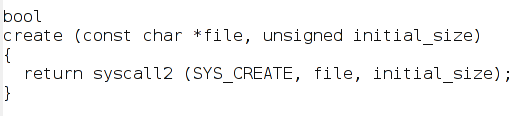


pfn[SYS\_WRITE]=IWrite; //printf和写文件需要。 



printf 函数会调用这个系统调用向屏幕输出，所以不实现这个系统调用，用户程序将无法输出任何字符。 写文件也要用这个系统调用。所以要使用pintos自带的一个简单的文件系统。 首先从用户栈中取出三个参数---fd,buffer,size 如果 fd 是文件句柄，先要从进程打开文件表中找到该句柄对应的文件指针再调用pintos提供的file\_write()函数向文件写入数据。 打开文件表将在打开文件时建立，到SYS\_OPEN系统调用实现时再讲其具体实现。 如果fd是标准输出stdout句柄则调用putbuf 函数向终端输出。

pfn[SYS\_CREATE]=ICreate; //创建文件

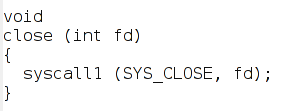


取出仅有的一个参数—文件名。

调用filesys\_create()函数。

保存返回值。

pfn[SYS\_CLOSE]=IClose; //关闭文件



一种是关闭一个文件。

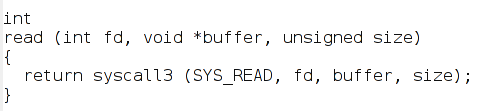
一种是进程退出时关闭所有文件。

从用户栈中获取要关闭文件的句柄。

在用户打开文件列表中找到对应文件，以得到文件指针。调用file\_close()函数关闭文件，释放struct file\_node。

关闭所有文件自然是每一个都要关闭，释放了。

pfn[SYS\_READ]=IRead; // 读文件



从用户栈中获得fd buffer size三个参数

如果fd是标准输入设备，则调用input\_getc()

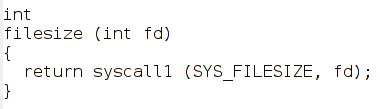
如果fd是文件句柄

由fd 从进程打开文件表中得到文件指针

调用file\_read()函数从文件中读数据。

pfn[SYS\_FILESIZE]=IFileSize; //返回文件大小





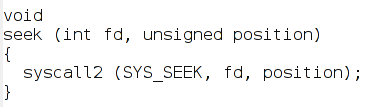
从用户栈中获得fd

由fd 从进程打开文件表中得到文件指针

调用file\_len gth得到文件大小

pfn[SYS\_SEEK]=ISeek; //移动文件指针



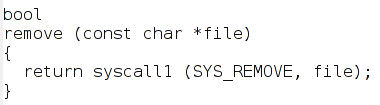


从用户栈中取出文件句fd柄要移动的距离，

把fd转为文件指针，

调用file\_seek()函数移动文件指针即可。

pfn[SYS\_REMOVE]=IRemove; //删除文件 

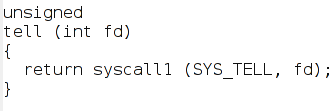


从用户栈中取出要删除文件的文件名。

调用filesys\_remove()删除文件。

pfn[SYS\_TELL]=ITell; //返回文件指针位置

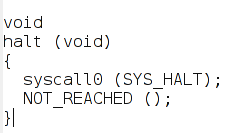




从用户栈中取出要删除文件的文件名。

调用filesys\_remove()删除文件。

pfn[SYS\_HALT]=IHalt; //关机



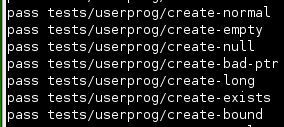
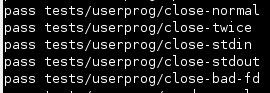
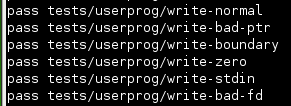
调用shutdown\_power\_off()函数关机

用户程序导致页错误时，会进入page\_fault()函数，在exception.c中。

在page\_fault()中加入

if(not\_present||(is\_kernel\_vaddr(fault\_addr)&&user)) ExitStatus(-1); 来处理页错误。

执行截图：



**Task4 Denying Writes to Executables**

用户程序加载过程：

(1)核心线程通过调用process\_execute(char \*file\_name); 函数来创建用户进程。 File\_name为要加载的文件名。这个函数中还调用了thread\_create()函数为用户进程创建了一个线程。File\_name 和一个叫start\_process的函数 被传递给了thread\_create(), thread\_create()函数创建线程后就把线程放入ready\_list()等待被调度;

(2) 得到CPU后就开始start\_process(void \*file\_name)函数。

这个函数做了以下几件事儿：

1根据file\_name把用户程序从硬盘调入内存，还为其分配了虚拟内存。注意这里要完成task2，不然文件名不正确，就没法打开文件。 2给用户分配了栈空间3GB开始。向低字节增长。 在这里要实现参数分离，并且把参数按规定放入用户栈中。

(3)通过内嵌汇编 asm volatile (………….)调用了用户程序中的main()函数。

(4）main()函数从栈中取出传给他的参数，执行完毕后会调用系统调用exit(),Exit()函数又调用thread\_exit()函数，thread\_exit()函数又调用process\_exit()函数，最后在thread\_exit（）函数中把即将退出的函数的进程控制块struct thread从all\_list中remove掉，调用了进程调度器schedule()函数，调用下一下进程执行。

系统调用过程：

在用户程序使用一个系统调用，如printf();在必然会触发一个30号中断，正如src/lib/user/syscall.c文件中所述。可见参数个数不同，系统调用不同。 这个30号中断调用之前，把系统调用号、用户参数（0到3个不等）压入栈中。然后开始执行中断程序，中断程序又调系统调用过程：

在用户程序使用一个系统调用，如printf();在必然会触发一个30号中断，正如src/lib/user/syscall.c文件中所述。可见参数个数不同，系统调用不同。 这个30号中断调用之前，把系统调用号、用户参数（0到3个不等）压入栈中。然后开始执行中断程序，中断程序又调用了syscall\_handler(struct intr\_frame \*f)函数，其中f是一个指向了用户程序当前运行信息的的指针，其中就有用户栈指针esp,所以在我们添加的系统调用中，就可以根据这个指针取出系统调用号和各个参数。 系统调用结束后，要把返回值入如f->eax中. 注意： 用户栈中的各个参数并不连续存放： 

三个参数

write(fd,buffer,size);

int fd=\*(esp+2);

char \*buffer=(char \*)\*(esp+6);

unsigned size=\*(esp+3); 

两个参数

create(pFileName,size);

bool ret=filesys\_create((const char \*)\*((unsigned int \*)f->esp+4),\*((unsigned int \*)f->esp+5)); 

一个参数 exit(-1); cur->ret=\*((int \*)f->esp+1);