Pintos project2

作者：西安电子科技大学　王永刚　　QQ:357543420

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 需要完成的的任务有四个：

- Task1　Process Termination Messages 进程终止信息
- Task2 Argument Passing　参数传递
- Task3 System Calls　系统调用
- Task4 Denying Writes to Executables 不能写入可执行文件

Task1: Process Termination Messages 进程终止信息

要求:

1.在进程结束时输出退出代码(就是 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”,..);

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;

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 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。

TASK1 OK…

TASK2 Argument Passing　参数传递

要求:

1. 分离从命令行传入的文件名和各个参数。

2. 按照 C 函数调用约定，把参数放入栈中。

实现方法:

1.分离参数的方法:用 string.h 中的 strtok_r()函数，在 string.c 中有详细的说明。

2.在 process_execute()函数中，因为 thread_create()需要一个线程名，此时应该传递给它文件名(不带参数)。可如下处理:

```
char *real_name, *save_ptr;
  real_name = strtok_r (file_name, " ", &save_ptr);
  tid = thread_create (real_name, PRI_DEFAULT, start_process, fn_copy);
```

(3) 在 start_process()函数中，再次分离参数，放入栈中。

由于在 process_execute()对 file_name 作了复制,文件名并未丢失，但是要注意，无论加载用户程序成功还是失败，都得释放 file_name 所占用的一个页的空间(Debug here 3 weeks)。 注意：传给 Load()函数的参数也只能有文件名，所以在 load()函数前要分离出文件名:

```
char *token=NULL, *save_ptr=NULL;
token = strtok_r (file_name, " ", &save_ptr);
success = load (token, &if_.eip, &if_.esp);
```

参数放置的一种方法:

(1) 找到用户栈指针:

在 start_process()函数中有 struct intr_frame if_; 这样一个结构，其中有一个成员 if_.esp,这就是用户栈指针，在 load()函数

中为其赋值，分配了栈空间。

(2) 调用 strtok_r 分离出一个个参数(就是一个个字符串了)，把每个字符串都复制到用户栈中,并把他在栈中的位置记录到一个数组中，以备下一步使用。注意：栈是向下增长，而字符串是向上增长。

```
char *esp=(char *)if_.esp;
char *arg[256];                //assume numbers of argument below 256
int i,n=0;
for (; token != NULL;token = strtok_r (NULL, " ", &save_ptr))
  {
      esp-=strlen(token)+1; //because user stack increase to low addr.
      strlcpy(esp,token,strlen(token)+2);  //copy param to user stack
      arg[n++]=esp;
}
```

(3) 要加入一个双字的对齐，因为是 32 位的，所以就是四字节对齐。

```
      while((int)esp%4make)            //word align
          esp--;                  //注意：栈是向下增长，所以这里是一而不是++;
```

(4)要将第 (2) 步保存下的指针逆序放入栈中。

按照 C 约定，先要放入一个 0，以防没有参数。

```
int *p=esp-4;
*p--=0;
```

然后依次放入参数 n 的地址,参数 n-1 的地址 … 参数 0 的地址。

```
for(i=n-1;i>=0;i--)            //place the arguments' pointers to stack
    *p--=(int *)arg[i];
```

(5)放入 argc,argv

```
*p--=p+1;
*p--=n;
*p--=0;
esp=p+1;
```

(6)让用户栈指针指向新的栈顶

```
if_.esp=esp
```

如下图摆放。 如果命令行是: /bin/ls –l foo bar

The table below shows the state of the stack and the relevant registers right before the beginning of the user program, assuming PHYS_BASE is 0xc0000000:

Address	Name	Data	Type
0xbffffffc	argv[3][...]	'bar\0'	char[4]
0xbffffff8	argv[2][...]	'foo\0'	char[4]
0xbffffff5	argv[1][...]	'-l\0'	char[3]
0xbfffffed	argv[0][...]	'/bin/ls\0'	char[8]
0xbfffffec	word-align	0	uint8_t
0xbfffffe8	argv[4]	0	char *
0xbfffffe4	argv[3]	0xbffffffc	char *
0xbfffffe0	argv[2]	0xbffffff8	char *
0xbfffffdc	argv[1]	0xbffffff5	char *
0xbfffffd8	argv[0]	0xbfffffed	char *
0xbfffffd4	argv	0xbfffffd8	char **
0xbfffffd0	argc	4	int
0xbfffffcc	return address	0	void (*) ()

In this example, the stack pointer would be initialized to 0xbffffffcc.

As shown above, your code should start the stack at the very top of the user virtual address space, in the page just below virtual address PHYS_BASE (defined in

完整代码见附录!

TASK 3 system call 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_EXIT]=IExit; //退出时 return 后调用
- pfn[SYS_CREATE]=ICreate; //创建文件
- pfn[SYS_OPEN]=IOpen; //打开文件
- pfn[SYS_CLOSE]=IClose; //关闭文件
- pfn[SYS_READ]=IRead; // 读文件
- pfn[SYS_FILESIZE]=IFileSize; //返回文件大小

- pfn[SYS_EXEC]=IExec;　　//加载用户程序
- pfn[SYS_WAIT]=IWait;　//等待子进程结束
- pfn[SYS_SEEK]=ISeek;　　//移动文件指针
- pfn[SYS_REMOVE]=IRemove;　　//删除文件
- 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，还要明白系统调用的机制，见后边。

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。

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。完整代码见附录

①SYS_WRITE----------void IWrite(struct intr_frame *f)

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。

②SYS_EXIT---------- void IExit(struct intr_frame *f);

用户程序正常退出会调用这个系统调用。

取出返回值，保存到进程控制块的 ret 变量中。

调用 thread_exit()函数退出进程

用户程序非正常退出(如越界访问等原因)需要另加一个函数来实现。

```
void ExitStatus(int status)      //非正常退出时使用
{
    struct thread *cur=thread_current();
    cur->ret=status;           //保存返回值。
    thread_exit();
}
```

③SYS_CREATE-创建文件 void ICreate(struct intr_frame *f)

取出仅有的一个参数—文件名。

调用 filesys_create()函数。

保存返回值。

④SYS_OPEN---打开文件 void IOpen(struct intr_frame *f)

取出文件名。

调用 filesys_open()函数打开文件。

这里需要为每个进程维护一个打开文件表。

打开文件后要为这个文件分配一个句柄号。

在 struct thread 结构中加入:

```
int FileNum;        //打开文件数   限制进程打开文件数
struct list file_list;  //打开文件列表
```

```
 int maxfd;              //句柄分配使用
```

每打开一个文件就让 maxfd 加 1，关闭文件可以不减小。

关联文件句柄与文件指针的结构：（被链入 file_list）

```
struct file_node
{
    int fd;
    struct list_elem elem;
    struct file *f;
};
```

有了以上准备，每打开一个文件都要新创建一个 file_node 结构，分配句柄，并把 file_node 加入 file_list；最后返回文件句柄就 OK.

⑤SYS_CLOSE—关闭文件 void IClose(struct intr_frame *f)

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。 Debug here 3 weeks

⑥SYS_READ—读文件 IRead()

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。

⑦SYS_FILESIZE — 获取文件大小　IFileSize()

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

⑧SYS_EXEC --- 加载用户程序　IExec()

用户程序通过 SYS_EXEC 这个系统调用创建子进程。

在 IExec()函数中,

分配一个页，复制一份用户提供的用户名。否则在后来分离参数时,加入’\0’时出现核心线程写入用户内存空间的页错误。

还要注意线程同步问题。

在 IExec()中调用 process_execute()函数创建子进程，但是从process_execute()得到了用户进程 pid 后，用户程序并没用加载。所以要等待用户进程被调度后—调用了 start_process()函数才能知道。 Start_process()函数真正加载用户程序，可能会因为找不到程序文件或内存不足等原因导致加载失败。 所以父进程调用process_execute()后不能立即返回，要在一个信号量上等待sema_down(sema)，直到 start_process()函数中加载用户程序成功后再 semp_up(sema)激活父进程，激活父进程后应该立即挂起自己—sema_down(sema),这里父进程获取子进程状态信息后,再出父进程sema_up()激活子进程。 如果父进程创建了一个优先级比自己高的子进程,如果不这样坐,start_process()函数每一次执行 sema_up(sema)

后，父进程还是不会被调度，而子进程可以已经运行完毕，这样父进程就得不到子进程的状态了。

在 struct_thread 结构中加入 semaphore SemaWaitSuccess;

可以在父进程的的这个信号量上等，也可是子进程的 SemaWaitSuccess 上等。

如果子进程创建成功则返回 pid，失败返回-1。

⑨SYS_WAIT—等待函数　IWait()

主线程创建子进程后，出于他与子进程优先级一样，所以，二者交替执行，这样主线程就有可能先结束，这导致了一开始的 test 失败。

起初可以通过创建子进程时提高子进程优先级或者在 process_wait()中加入 while(true)这样的死循环来解决。后期要通过信号量同步。

这个系统调用的需求：父进程创建子进程后可能要等子进程结束。

Process_wait()要返回子进程的返回值。

情况有如下：

父进程调用 process_wait()时子进程还未结束，此进父进程将被挂起，等子进程结束后再唤醒父进程，父进程再取得返回值。

父进程调用 process_wait()时子进程已经结束,这就要求子进程结束后应该把返回值保存到父进程的进程控控制块中。

于是在 struct thread 要加入一个链表，struct list sons_ret;

结构:

```
struct ret_data
{
      int pid;
      int ret;
      struct list_elem elem;
};
```

这样就能保存子进程的返回值了。

在 struct thread 结构中加入 bool bWait; 表示进程本身有没有被父进程等待。

在 struct thread 结构中加入 bool SaveData; 如果子进程已经把返回值保存到父进程里了就设 SaveData 为 true； SaveData 应该被初始化为 false;

在 struct thread 结构中加入 struct thread *father;表示父线程。每创建一个子线程，都要在子线程中记录父线程。

信号量同步方法:

在 struct thread 结构中加入 semaphore SemaWait;

这里选择在父进程的 SemaWait 上等。这个等待会把父进程的 struct thread 进程控制块插入到 SemaWait 的 list 中去。要想同时等待多个进程则不可能把父进程插入到多个子进程中去。当然，这里的测试只能等一个子进程，所以在父进程和子进程上等都可以。

父进程执行 process_wait(child_pid)后，可以由 child_pid 得到子进程 struct thread 指针 t。通过遍历 all_list 比较 pid 实现.

如果在 all_list 没有发现子进程的进程控制块或者发现 t->SaveData==true||t->status==THREAD_DYING;表示子进程已经结束，直接从自己的 sons_ret 链表中找到子进程的返回值返回就 OK. 如果子进程还在运行，则执行 sema_down(t->father->SemaWait)把自己挂起，子进程执行完毕后，发现在 bWait==true,自己被等待了，再释放父进程 sema_up(SemaWait)； 如果 bWait==fale,则不用唤醒父进程。

父进程被唤醒后，再从 sons_ret 链表中得到子进程的返回值。每个子进程只能等一次，第二次等同一个子进程只能返回-1.

一个进程结束时，在 process_exit()函数中，要释放自己打开的所有文件，保存返回值到父进程，输出退出信息，如果有父进程在等他就唤醒父进程，释放子进程链表。

⑩SYS_SEEK ---移动文件指针　ISeek()

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。

⑪SYS_REMOVE 删除文件　IRemove

从用户栈中取出要删除文件的文件名。

调用 filesys_remove()删除文件。

⑫SYS_TELL 返回文件指针当前位置 ITell()

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_tell()函数得到指针位置。

⑬SYS_HALT 关机 IHALT

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。

Task4 Denying Writes to Executables 不能写入可执行文件

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。

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);
```

附录:

Task2 参数传递代码(红色)

Task3 系统调用 (蓝色)

Task4 deny write (绿色)

```
tid_t
process_execute (const char *file_name)
{
  char *fn_copy;
  tid_t tid;

  /* Make a copy of FILE_NAME.
     Otherwise there's a race between the caller and load(). */
  fn_copy = palloc_get_page (0);
  if (fn_copy == NULL)
    return TID_ERROR;
  strlcpy (fn_copy, file_name, PGSIZE);
  char *real_name, *save_ptr;
  real_name = strtok_r (file_name, " ", &save_ptr);
  /* Create a new thread to execute FILE_NAME. */
  tid = thread_create (real_name, PRI_DEFAULT, start_process, fn_copy);
  if (tid == TID_ERROR)
    palloc_free_page (fn_copy);
  return tid;
}

/* A thread function that loads a user process and starts it
```

```
     running. */
  static void
```

start_process (void *file_name_)

```
  {
    char *file_name = file_name_;
    struct intr_frame if_;
    bool success;
    char *token=NULL, *save_ptr=NULL;
    token = strtok_r (file_name, " ", &save_ptr);  // get real file name, use it
in load()
    /* Initialize interrupt frame and load executable. */
    memset (&if_, 0, sizeof if_);
    if_.gs = if_.fs = if_.es = if_.ds = if_.ss = SEL_UDSEG;
    if_.cs = SEL_UCSEG;
    if_.eflags = FLAG_IF | FLAG_MBS;

    success = load (token, &if_.eip, &if_.esp);
    struct thread *t=thread_current();
      if (!success)
      {
      palloc_free_page (file_name);

        t->tid=-1;
        sema_up(&t->SemaWaitSuccess);
        ExitStatus(-1);
      }

    sema_up(&t->SemaWaitSuccess);
       t->FileSelf=filesys_open(token);
      file_deny_write(t->FileSelf);
    char *esp=(char *)if_.esp;
    char *arg[256];               //assume numbers of argument below 256
    int i,n=0;
    for (; token != NULL;token = strtok_r (NULL, " ", &save_ptr))   //copy the
argument to user stack
      {
          esp-=strlen(token)+1;                              //because user stack
increase to low addr.
          strlcpy(esp,token,strlen(token)+2);          //copy param to user stack
          arg[n++]=esp;

      }
```

```
  while((int)esp%4make)            //word align
    esp--;

  int *p=esp-4;
  *p--=0;                        //first 0
  for(i=n-1;i>=0;i--)             //place the arguments' pointers to stack
    *p--=(int *)arg[i];
  *p--=p+1;
  *p--=n;
  *p--=0;
  esp=p+1;
  if_.esp=esp;                     //set new stack top
  palloc_free_page (file_name);
  asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) : "memory");
  NOT_REACHED ();
}
```

Syscall.c 中所有代码:

```
#include "userprog/syscall.h"
#include "threads/vaddr.h"
#include <stdio.h>
#include <syscall-nr.h>
#include "threads/interrupt.h"
#include "threads/thread.h"
#include "filesys/filesys.h"
#include "filesys/file.h"
#include "devices/input.h"
#include  "process.h"
#include <string.h>
#include "devices/shutdown.h"
#define MAXCALL 21
#define MaxFiles 200
#define stdin 1
static void syscall_handler (struct intr_frame *);
typedef void (*CALL_PROC)(struct intr_frame*);
CALL_PROC pfn[MAXCALL];
void IWrite(struct intr_frame*);
void IExit(struct intr_frame *f);
void ExitStatus(int status);
void ICreate(struct intr_frame *f);
void IOpen(struct intr_frame *f);
void IClose(struct intr_frame *f);
void IRead(struct intr_frame *f);
void IFileSize(struct intr_frame *f);
```

```
void IExec(struct intr_frame *f);
void IWait(struct intr_frame *f);
void ISeek(struct intr_frame *f);
void IRemove(struct intr_frame *f);
void ITell(struct intr_frame *f);
void IHalt(struct intr_frame *f);
struct file_node *GetFile(struct thread *t,int fd);
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;

  pfn[SYS_WRITE]=IWrite;
  pfn[SYS_EXIT]=IExit;
  pfn[SYS_CREATE]=ICreate;
  pfn[SYS_OPEN]=IOpen;
  pfn[SYS_CLOSE]=IClose;
  pfn[SYS_READ]=IRead;
  pfn[SYS_FILESIZE]=IFileSize;
  pfn[SYS_EXEC]=IExec;
  pfn[SYS_WAIT]=IWait;
  pfn[SYS_SEEK]=ISeek;
  pfn[SYS_REMOVE]=IRemove;
  pfn[SYS_TELL]=ITell;
  pfn[SYS_HALT]=IHalt;
}

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);
}
```

void IWrite(struct intr_frame *f) //三个参数

```
{

    int *esp=(int *)f->esp;
      if(!is_user_vaddr(esp+7))
        ExitStatus(-1);
    int fd=*(esp+2);               //文件句柄
    char *buffer=(char *)*(esp+6); //要输出人缓冲
    unsigned size=*(esp+3);        //输出内容大小。

    if(fd==STDOUT_FILENO)         //标准输出设备
    {
        putbuf (buffer, size);
        f->eax=0;
    }
    else                          //文件
    {
        struct thread *cur=thread_current();
        struct file_node *fn=GetFile(cur,fd); //获取文件指针
        if(fn==NULL)
        {
            f->eax=0;
            return;
        }

        f->eax=file_write(fn->f,buffer,size);//写文件

    }

}
```

void IExit(struct intr_frame *f) //一个参数 正常退出时使用

```
{
    if(!is_user_vaddr(((int *)f->esp)+2))
      ExitStatus(-1);
    struct thread *cur=thread_current();
    cur->ret=*((int *)f->esp+1);
    f->eax=0;
    thread_exit();

}
```

void ExitStatus(int status) //非正常退出时使用

```
{
    struct thread *cur=thread_current();
    cur->ret=status;
    thread_exit();
}
```

void ICreate(struct intr_frame *f) //两个参数

```
{

    if(!is_user_vaddr(((int *)f->esp)+6))
      ExitStatus(-1);
    if((const char *)*((unsigned int *)f->esp+4)==NULL)
        {
            f->eax=-1;
            ExitStatus(-1);
        }
    bool ret=filesys_create((const char *)*((unsigned int *)f->esp+4),*((unsigned int *)f->esp+5));
    f->eax=ret;
}
```

void IOpen(struct intr_frame *f)

```
{
```

```
    if(!is_user_vaddr(((int *)f->esp)+2))
      ExitStatus(-1);
    struct thread *cur=thread_current();
    const char *FileName=(char *)*((int *)f->esp+1);
    if(FileName==NULL)
    {
        f->eax=-1;
        ExitStatus(-1);
    }
    struct file_node *fn=(struct file_node *)malloc(sizeof(struct file_node));

    fn->f=filesys_open(FileName);
    if(fn->f==NULL|| cur->FileNum>=MaxFiles)//
        fn->fd=-1;
    else
        fn->fd=++cur->maxfd;

    f->eax=fn->fd;
    if(fn->fd==-1)
        free(fn);

    else
    {
        cur->FileNum++;
        list_push_back(&cur->file_list,&fn->elem);
    }
}
```

void IClose(struct intr_frame *f)

```
{
    if(!is_user_vaddr(((int *)f->esp)+2))
      ExitStatus(-1);
    struct thread *cur=thread_current();
    int fd=*((int *)f->esp+1);
    f->eax=CloseFile(cur,fd,false);

}
```

int CloseFile(struct thread *t,int fd,int bAll)

```
{
    struct list_elem *e,*p;
```

```
    if(bA11)
    {
        while(!list_empty(&t->file_list))
        {
            struct file_node *fn = list_entry (list_pop_front(&t->file_list),
struct file_node, elem);
            file_close(fn->f);
             free(fn);
        }
        t->FileNum=0;
        return 0;
    }

   for (e = list_begin (&t->file_list); e != list_end (&t->file_list);)
   {
        struct file_node *fn = list_entry (e, struct file_node, elem);
            if(fn->fd==fd)
            {
                list_remove(e);
                if(fd==t->maxfd)
                    t->maxfd--;
                t->FileNum--;
                file_close(fn->f);
               free(fn);

                return 0;
            }
   }

 }
```

void IRead(struct intr_frame *f)

```
 {
    int *esp=(int *)f->esp;
    if(!is_user_vaddr(esp+7))
        ExitStatus(-1);
    int fd=*(esp+2);
    char *buffer=(char *)*(esp+6);
    unsigned size=*(esp+3);

    if(buffer==NULL||!is_user_vaddr(buffer+size))
    {
```

```
        f->eax=-1;
        ExitStatus(-1);
    }

    struct thread *cur=thread_current();
    struct file_node *fn=NULL;
    unsigned int i;
    if(fd==STDIN_FILENO)                //从标准输入设备读
    {
        for(i=0;i<size;i++)
            buffer[i]=input_getc();

    }
    else                                //从文件读
    {
        fn=GetFile(cur,fd);             //获取文件指针
        if(fn==NULL)
        {
            f->eax=-1;
            return;
        }
        f->eax=file_read(fn->f,buffer,size);
    }
}
```

struct file_node *GetFile(struct thread *t,int fd) //依据文件句柄从进程打开文件表中找到文件指针

```
{
    struct list_elem *e;

    for (e = list_begin (&t->file_list); e != list_end (&t->file_list);e=list_next (e))
    {
       struct file_node *fn = list_entry (e, struct file_node, elem);
       if(fn->fd==fd)
            return fn;
    }
    return NULL;
}
```

void IFileSize(struct intr_frame *f)

```
{
    if(!is_user_vaddr(((int *)f->esp)+2))
      ExitStatus(-1);
    struct thread *cur=thread_current();
    int fd=*((int *)f->esp+1);
    struct file_node *fn=GetFile(cur,fd);
    if(fn==NULL)
    {
        f->eax=-1;
        return;
    }
    f->eax=file_length (fn->f);
}
```

void IExec(struct intr_frame *f)

```
{
     if(!is_user_vaddr(((int *)f->esp)+2))
      ExitStatus(-1);
    const char *file=(char*)*((int *)f->esp+1);
    tid_t tid=-1;
    if(file==NULL)
    {
        f->eax=-1;
        return;
    }
    char *newfile=(char *)malloc(sizeof(char)*(strlen(file)+1));

    memcpy(newfile,file,strlen(file)+1);
    tid=process_execute (newfile);
    struct thread *t=GetThreadFromTid(tid);
    sema_down(&t->SemaWaitSuccess);
    f->eax=t->tid;
    t->father->sons++;
    free(newfile);
    sema_up(&t->SemaWaitSuccess);
}
```

void IWait(struct intr_frame *f)

```
{
    if(!is_user_vaddr(((int *)f->esp)+2))
      ExitStatus(-1);
    tid_t tid=*((int *)f->esp+1);
    if(tid!=-1)
    {
        f->eax=process_wait(tid);

    }
    else
        f->eax=-1;
}
```

void ISeek(struct intr_frame *f)

```
{
    if(!is_user_vaddr(((int *)f->esp)+6))
      ExitStatus(-1);

    int fd=*((int *)f->esp+4);
    unsigned int pos=*((unsigned int *)f->esp+5);
    struct file_node *f1=GetFile(thread_current(),fd);
    file_seek (f1->f,pos);
}
```

void IRemove(struct intr_frame *f)

```
{
    if(!is_user_vaddr(((int *)f->esp)+2))
      ExitStatus(-1);
    char *f1=(char *)*((int *)f->esp+1);
    f->eax=filesys_remove (f1);
}
```

void ITell(struct intr_frame *f)

```
{
    if(!is_user_vaddr(((int *)f->esp)+2))
      ExitStatus(-1);
    int fd=*((int *)f->esp+1);
     struct file_node *f1=GetFile(thread_current(),fd);
     if(f1==NULL||f1->f==NULL)
     {
         f->eax=-1;
         return;
     }
    f->eax=file_tell (f1->f);
}
```

void IHalt(struct intr_frame *f)

```
{
    shutdown_power_off();
    f->eax=0;

}

int
```

process_wait (tid_t child_tid)

```
{
   struct thread *t=GetThreadFromTid(child_tid);
    if(t==NULL||t->status==THREAD_DYING || t->SaveData)  //子进程已经把返回值保存到 Sons_ret 链表中了
   {
       int ret=-1;
       ret=GetRetFromSonsList(thread_current(),child_tid);  //从 sons_ret 中取回子进程的返回值

       return ret;
   }
    t->bWait=true;
   sema_down(&t->father->SemaWait);                    //在这个信号量上等。
   int ret=-1;
    ret=GetRetFromSonsList(thread_current(),child_tid);
```

```
  return ret;
}
```

int GetRetFromSonsList(struct thread *t,tid_t tid)

```
{
    struct list_elem *e;
    int ret=-1;
    for(e=list_begin(&t->sons_ret);e!=list_end(&t->sons_ret);e=list_next(e))
    {
        struct ret_data *rd=list_entry(e,struct ret_data,elem);
        if(rd->tid==tid)
        {
            ret=rd->ret;
            rd->ret=-1;        //每个子进程只能被等一次,第二此等返回-1
            break;
        }
    }
    return ret;
}
/* Free the current process's resources. */
void
```

process_exit (void)

```
{
  struct thread *cur = thread_current ();

  uint32_t *pd;

  /* Destroy the current process's page directory and switch back
     to the kernel-only page directory. */
  pd = cur->pagedir;

  if (pd != NULL)
    {
      /* Correct ordering here is crucial.  We must set
         cur->pagedir to NULL before switching page directories,
         so that a timer interrupt can't switch back to the
         process page directory.  We must activate the base page
         directory before destroying the process's page
```

```
           directory, or our active page directory will be one
           that's been freed (and cleared). */

        CloseFile(cur,-1,true);  //关闭打开的文件

        if(cur->FileSelf!=NULL)  //撤销对自己人 deny_write
        {
            file_allow_write(cur->FileSelf);
            file_close (cur->FileSelf);
        }
        printf("%s: exit(%d)\n",cur->name,cur->ret); //输出推出消息
        record_ret(cur->father,cur->tid,cur->ret); //保存返回值到父进程
        cur->SaveData=true;
         if(cur->father!=NULL&&cur->bWait)  //如果有父进程在等就唤醒他

        {
            while(!list_empty(&cur->father->SemaWait.waiters))
              sema_up(&cur->father->SemaWait);

        }
        while(!list_empty(&cur->sons_ret))    //释放孩子返回值链表
        {
            struct ret_data *rd=list_entry(list_pop_front(&cur->sons_ret),struct
ret_data,elem);
            free(rd);
        }
        cur->pagedir = NULL;
        pagedir_activate (NULL);
        pagedir_destroy (pd);
      }



  }
```

void record_ret(struct thread *t,int tid,int ret)

```
  {
      struct ret_data *rd=(struct ret_data *)malloc(sizeof(struct ret_data));
      rd->ret=ret;
      rd->tid=tid;
      list_push_back(&t->sons_ret,&rd->elem);
  }
```

```
static void
```

page_fault (struct intr_frame *f)

```
{
  bool not_present;  /* True: not-present page, false: writing r/o page. */
  bool write;        /* True: access was write, false: access was read. */
  bool user;         /* True: access by user, false: access by kernel. */
  void *fault_addr;  /* Fault address. */

  /* Obtain faulting address, the virtual address that was
     accessed to cause the fault.  It may point to code or to
     data.  It is not necessarily the address of the instruction
     that caused the fault (that's f->eip).
     See [IA32-v2a] "MOV--Move to/from Control Registers" and
     [IA32-v3a] 5.15 "Interrupt 14--Page Fault Exception
     (#PF)". */
  asm ("movl %%cr2, %0" : "=r" (fault_addr));

  /* Turn interrupts back on (they were only off so that we could
     be assured of reading CR2 before it changed). */
  intr_enable ();

  /* Count page faults. */
  page_fault_cnt++;

  /* Determine cause. */
  not_present = (f->error_code & PF_P) == 0;
  write = (f->error_code & PF_W) != 0;
  user = (f->error_code & PF_U) != 0;
  if(not_present||(is_kernel_vaddr(fault_addr)&&user))
      ExitStatus(-1);
   //if(!not_present&&is_user_vaddr(fault_addr)&&!user)
    //  return;
  /* To implement virtual memory, delete the rest of the function
     body, and replace it with code that brings in the page to
     which fault_addr refers. */
  printf ("Page fault at %p: %s error %s page in %s context.\n",
          fault_addr,
          not_present ? "not present" : "rights violation",
          write ? "writing" : "reading",
          user ? "user" : "kernel");
  kill (f);
```

```
}
```