Pintos Project2　　设计与实现文档

1. 小组成员与具体分工

谢斌：10142510237

System calls、Denying write to in-use executable files
Access safe memory、
设计与实现文档、Github 版本控制

张健宁：10142510262

Argument Passing 、Process Termination Messages
Readme 文档制作

所有代码最后通过 Github 汇总，进行版本控制。
https://github.com/Enverer/myPintosProject/tree/master/src/threads

参考资料：http://www.scs.stanford.edu/10wics140/pintos

2. Argument passing (参数传递)

【需求分析】

3.3.3 Argument Passing

Currently, `process_execute()` does not support passing arguments to new processes. Implement this functionality, by extending `process_execute()` so that instead of simply taking a program file name as its argument, it divides it into words at spaces. The first word is the program name, the second word is the first argument, and so on. That is, `process_execute("grep foo bar")` should run `grep` passing two arguments `foo` and `bar`.

Within a command line, multiple spaces are equivalent to a single space, so that `process_execute("grep foo  bar")` is equivalent to our original example. You can impose a reasonable limit on the length of the command line arguments. For example, you could limit the arguments to those that will fit in a single page (4 kB). (There is an unrelated limit of 128 bytes on command-line arguments that the `pintos` utility can pass to the kernel.)

You can parse argument strings any way you like. If you're lost, look at `strtok_r()`, prototyped in '`lib/string.h`' and implemented with thorough comments in '`lib/string.c`'. You can find more about it by looking at the man page (run `man strtok_r` at the prompt).

See Section 3.5.1 [Program Startup Details], page 34, for information on exactly how you need to set up the stack.

现在的 Pintos 不支持用户程序的正确参数传递。事实上，main()函数初始完毕后并解析完所有的用户命令后，会调用 run_action()函数，又如果检测到用户输入了“run”，则调用 run_task()运行改程序，进一步调用 process_wait()来等待

process_execute()完成任务。其中 process_execute()的参数列表中只定义了一个变量 const char *file_name，就是说它只接收用户程序的文件名为参数，我们需要做的事情是完善它的参数接受列表，使之成为第一个参数为文件名、后面为程序使用参数的形式。另外，原有的代码会分配一个中断页框 if_，这个 if_就是用来把本次程序执行的信息传递给中断处理机制让中断来运行程序的。Pintos 的命令行语句长度最大为 128 字节。

【实现方案】

1. Process excute()

```
/* add by Zhangjianning for argument
passing */
char *save;
char *fn;

struct thread *t;

tid = TID_ERROR;
/* add end */
```

首先在最开始添加两个 char *类型变量和一个 struct 结构体，*fn 指向 file_name 的一个拷贝，是为了实现保存从命令语句中分离出来的真正的可执行文件名，给 fn 分配与命令语句容量相等的内存空间。给 tid 赋一个缺省值 TID_ERROR，便于后面进行判断。

```
/* 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);
```

函数实现的是为传进来的命令语句即 file_name 建立一个备份 fn_copy，因为之后的 thread_create()需要使用文件名，而 start_process()里会调用 load()，load()也会使用文件名，这样会造成冲突。

```
54    /* add by Zhangjianning for argument passing */
55    fn = malloc (strlen (file_name) + 1);
56    if (!fn)
57      goto done;
58    memcpy (fn, file_name, strlen (file_name) + 1);
59    file_name = strtok_r (fn, " ", &save);
60    /* add end */
```

55 行是为 fn 分配与命令语句容量相等的内存空间，如果分配失败，则在 57 行 goto done;。done 块中的代码是在下面一张图中实现的。接下来，58 行，将 file_name 里面的值通过 memcpy 函数拷贝到 fn 中，并且调用 strtok_r()函数，将字符串中第一个空格前的东西（也就是第一个参数）再赋值给 file_name。

```
  /* Create a new thread to execute FILE_NAME. */
  /* modify by Zhangjianning for argument passing */
  tid = thread_create (fn, PRI_DEFAULT, start_process, fn_copy);
  if (tid == TID_ERROR)
    goto done;
  t = get_thread_by_tid (tid);
  sema_down (&t->wait);
  if (t->ret_status == -1)
    tid = TID_ERROR;
  while (t->status == THREAD_BLOCKED)
    thread_unblock (t);
  if (t->ret_status == -1)
    process_wait (t->tid);

done:
  free (fn);
  /* add end */
```

之后修改了 tid 的获取方法，这里调用了 thread_create()函数，它的功能及相应的修改将在之后给出。如果获取不到相应的 tid，跳转到 done 块。接着通过写出的 get_thread_by_tid()辅助函数获取到当前线程，并调用 sema_down()使其等待其子线程加载。如果这个线程执行不成功，即它的 ret_status 被置为了-1，那么把 tid 设为 TID_ERROR。如果它处于阻塞状态就将它解阻塞，促使其尽快运行。最后调用 process_wait()来等待这个线程执行完成。最后无论是被跳转到这里还是直接到这里，调用 free()函数，释放其所占有的内存。最后，如果 tid 还是 TID_ERROR，那么将释放 fn_copy 所占有的内存，以保持数据一致性，最后返回 tid。

2.thread create()

```
/* add by Zhangjianning for argument passing */
  ASSERT (priority >= PRI_MIN && priority <= PRI_MAX);
#ifdef USERPROG
  if (list_size (&all_list) >= 35) /* Maximum capacity of threads */
    return TID_ERROR;
#endif
```

```
  /* add by Zhangjianning for argument passing */
  #ifdef USERPROG
    sema_init (&t->wait, 0);
    t->ret_status = RET_STATUS_DEFAULT;
    list_init (&t->files);
    t->parent = NULL;
  #endif
    /* add end */
```

这里首先断言了线程的优先级时在有效范围之内。如果现有的线程数量>=35则创建线程失败，因为得不到正确的 TID。同时对于信号量进行初始化，并且初始化了默认的返回状态，初始化了文件列表及将父线程置为 null。

3.process wait()

```
struct thread *t;
int ret;

t = get_thread_by_tid (child_tid);
if (!t || t->status == THREAD_DYING || t->parent == thread_current ())
  return -1;
if (t->ret_status != RET_STATUS_DEFAULT)
  return t->ret_status;

t->parent = thread_current ();
intr_disable ();
thread_block ();
intr_enable ();
ret = t->ret_status;
printf ("%s: exit(%d)\n", t->name, t->ret_status);
while (t->status == THREAD_BLOCKED)
  thread_unblock (t);

return ret;
```

将传递进来的 child_tid 值通过 get_thread_by_tid()辅助函数查询出所对应的线程。如果出现了图中这些情况，则需要立刻返回-1；并且如果 t->ret_status 被重置，需要返回该值，接着，将 t 的父亲线程设置为当前线程，并且将当前线程阻塞。之后就是 Message 输出的任务。

4.start process（）

```
/* add by Zhangjianning for argument passing */
char *token, *save_ptr;
void *start;
int argc, i;
int *argv_off; /* Maximum of 2 arguments */
size_t file_name_len;
struct thread *t;
/* add end */
```

此函数是 Project 2 中参数传递的关键，实现了参数分离。*token 和 *save_ptr 是用来在接下来的 str_tok_r 中使用，start 用来记录开始压栈的地方，argc 保存参数的个数，argv_off 记录每个参数距离压栈起点的长度，file_name_len 是整个文件名加参数的字符长度。

```
/* add by Zhangjianning for argument passing */
t = thread_current ();
argc = 0;
argv_off = malloc (32 * sizeof (int));
if (!argv_off)
  goto exit;
file_name_len = strlen (file_name);
argv_off[0] = 0;
for (
      token = strtok_r (file_name, " ", &save_ptr);
      token != NULL;
      token = strtok_r (NULL, " ", &save_ptr)
      )
       {
         while (*(save_ptr) == ' ')
           ++save_ptr;
         argv_off[++argc] = save_ptr - file_name;
       }
/* add end */
```

让当前线程赋值给 t，将参数个数设置为 0。由于需求中提到，pintos 的命令行语句长度最大为 128 字节，也就是说每个参数距离压栈起点的最大长度不会超过 128 字节，因此为 argv_off 分配 128 字节的空间，如果分配空间失败，则跳转，跳转的目的地将在下面的代码中给出。调用 strlen()将传入的文件名字符串的长度赋给 file_name_len，并将 argv_off 指向的第一个数值为零，因为为了方便对应，我们从 argv_off 代表的数组的下标 1 开始存放参数，这样就不要每次都考虑那个 1 的问题了。for 循环进行参数分离：采用 strtok_r()方法以空格为分隔符分割 file_name。但是因为用户有可能不小心在两个参数间输入多个空格，还要在每次 for 循环体里加入一个 while 循环将多余的空格过滤。每执行一次分割就将 argc 加 1，并在 argv_off 数组的相应元素里存入当前参数与压栈起点的距离。

```
/* add by Zhangjianning for argument passing */
/* Setting up stack */
if (success)
  {
    t->self = filesys_open (file_name);
    file_deny_write (t->self);
    if_.esp -= file_name_len + 1;
    start = if_.esp;
    memcpy (if_.esp, file_name, file_name_len + 1);
    if_.esp -= 4 - (file_name_len + 1) % 4; /* alignment */
    if_.esp -= 4;
    *(int *)(if_.esp) = 0; /* argv[argc] == 0 */
    /* Now pushing argv[x], and this is where the fun begins */
    for (i = argc - 1; i >= 0; --i)
      {
        if_.esp -= 4;
        *(void **)(if_.esp) = start + argv_off[i]; /* argv[x] */
      }

    if_.esp -= 4;
    *(char **)(if_.esp) = (if_.esp + 4); /* argv */
    if_.esp -= 4;
    *(int *)(if_.esp) = argc;
    if_.esp -= 4;
    *(int *)(if_.esp) = 0; /* Fake return address */

    sema_up (&t->wait);
    intr_disable ();
    thread_block ();
    intr_enable ();
```

将参数压入栈中。打开可执行文件并将其设置为该线程的映像文件，禁止其他文件向该文件进行写操作。

```
    else
      {
        free (argv_off);
exit:
        t->ret_status = -1;
        sema_up (&t->wait);
        intr_disable ();
        thread_block ();
        intr_enable ();
        thread_exit ();
      }

    free (argv_off);
    /* add end */

    /* modify by Zhangjianning for argument passing */
    palloc_free_page (file_name);
    /* modify end */
```

直到*(int*)(if_.esp)=0，因为 Stanford Guide 中给出的 80x86 Calling Convention 方法。将 if_.esp 倒退文件名长度个单位后赋值给 start，接着使用 memcpy 把所有参数都复制到栈里。对齐并将 argv[argc]压栈后，在 for 循环中我们将每个参数的地址压栈。接下来将 argv，即 argv[0]的地址以及 argc 和一个假返回地址 0 压栈。这之后参数压栈的任务就全部完成了，只需释放分配给辅助变量 argv_off 与 file_name 的内存并进行一些后续处理即可。

5.process exit（）

```
  /* add by Zhangjianning for argument passing */
  while (cur->parent && cur->parent->status == THREAD_BLOCKED)
    thread_unblock (cur->parent);
  file_close (cur->self);
  cur->self = NULL;
  intr_disable ();
  thread_block ();
  intr_enable ();
  /* add end */
```

由于需要退出当前线程，需要将该线程及该线程的父亲及祖父等等等等都解阻塞，并且调用 file_close()将自己的映射文件关闭。禁止外部中断后，将自己阻塞并且恢复原有外部中断状
态。

3. Access safe memory（内存安全访问）

【需求分析】

3.1.5 Accessing User Memory

As part of a system call, the kernel must often access memory through pointers provided by a user program. The kernel must be very careful about doing so, because the user can pass a null pointer, a pointer to unmapped virtual memory, or a pointer to kernel virtual address space (above PHYS_BASE). All of these types of invalid pointers must be rejected without harm to the kernel or other running processes, by terminating the offending process and freeing its resources.

【实现方案】

```
/* added by Xiebin for access safe memory */
t = thread_current ();
if (not_present || (is_kernel_vaddr (fault_addr) && user) || lookup_page( t->pagedir,fault_addr, false))
  sys_exit (-1);
/* add end */
```

1. 用户访问系统内存:用户程序访问内核内存时,系统会调用 page_fault 函数，根据 page_fault 函数中确定产生原因的代码段功能，我们可以通过加一个 if(is_kernel_vaddr(fault_addr)&&user)判断解析出是否是由用户程序造成的错误,如果是,按照确定的错误处理机制,调用 sysy_exit(-1)终止进程，返回失败信息-1.
2. 用户访问尚未被映射虚存地址空间：只要判断出它是由用户进程产生的 not_present 错误即可按照错误处理机制解决
3. 用户访问其他用户的虚存地址空间：通过在 page_fault 函数里加判断是难以解决的。注意到用户调用 system call 时，实际上运行的是内核的代码,所以当由于 system call 参数不合法产生 page fault 时,Determine Cause 代码段认为它是内核造成的错误，从而引发 kernel panic，系统崩溃。所以，我们在 system call 的调用机制里要加上错误检查，在检查出错误时直接终止该用户进程。具体的做法是利用 pagedir.c 中的 bool lookup_page(uint32_t *pd, const void *vaddr, bool create)这个函数,第一个参数指的是页表索引,第二个参数代表需要访问的虚存地址,函数的功能是判断该虚存地址是否在该页表索引指向的页中，第三个参数是一个执行选项，当 create 为 true 时，激活该虚存地址所在页， 否则错误处理只做判断。我们利用这个函数，以及 struct thread 的成员变量 pagedir，只利用其判断功能可以得知当前线程是否访问合法的地址，这样能处理一个用户进程访问其它用户进程内存的错误。

4. System calls (系统调用)

【需求分析】

3.3.4 System Calls

Implement the system call handler in ‘userprog/syscall.c’. The skeleton implementation we provide “handles” system calls by terminating the process. It will need to retrieve the system call number, then any system call arguments, and carry out appropriate actions.

Implement the following system calls. The prototypes listed are those seen by a user program that includes ‘lib/user/syscall.h’. (This header, and all others in ‘lib/user’, are for use by user programs only.) System call numbers for each system call are defined in ‘lib/syscall-nr.h’:

系统调用是由系统提供的一组完成底层操作的函数集合，由用户程序通过中断调用，系统根据中断向量表和中断服务号确定函数调用，调用相应的函数完成相应的服务。在 syscall.c 中只提供了 handles 这样的功能，然而查看了该函数后，发现它只是输出了一句”system call!”来告诉开发人员现在进行到了系统调用中，随即退出了当前线程，根据 Guide 里面接下来的描述，我们需要在 syscall.c 中添加 13 个要完成的系统调用，分别为 halt, exit, exec, wait, create, remove, open, filesize, read,write, seek, tell 及 close。其中的前四个为 syscall 部分，后面九个都与 filesys 有关。

【实现方案】

```
/* added by Xiebin for system call */
int sys_exit (int status);
/* add end */
```

由于要保证各个系统调用的唯一性，在实现的时候仅把 sys_exit(int)放在 syscall.h 中，其余的都放在 syscall.c 中。

```
typedef int pid_t;

static int sys_write (int fd, const void *buffer, unsigned length);
static int sys_halt (void);
static int sys_create (const char *file, unsigned initial_size);
static int sys_open (const char *file);
static int sys_close (int fd);
static int sys_read (int fd, void *buffer, unsigned size);
static int sys_exec (const char *cmd);
static int sys_wait (pid_t pid);
static int sys_filesize (int fd);
static int sys_tell (int fd);
static int sys_seek (int fd, unsigned pos);
static int sys_remove (const char *file);
```

```
static struct file *find_file_by_fd (int fd);
static struct fd_elem *find_fd_elem_by_fd (int fd);
static int alloc_fid (void);
static struct fd_elem *find_fd_elem_by_fd_in_process (int fd);

typedef int (*handler) (uint32_t, uint32_t, uint32_t);
static handler syscall_vec[128];
static struct lock file_lock;

struct fd_elem
  {
    int fd;
    struct file *file;
    struct list_elem elem;
    struct list_elem thread_elem;
  };

static struct list file_list;
```

在 syscall.c 里定义这些系统调用函数以及实现系统调用所需要的一些辅助函数。Pintos 文档的 Requirements-SystemCalls 小节里有给出各个系统调用函数的原型。将系统调用的 handler 定义为 int 型的指针，由于所有系统调用函数中最长使用 3 个参数，因此给它的参数个数统一规定为 3。并定

义一个 handler 型数组 syscall_vec 存放系统调用函数。file_lock 是用来作互斥锁的。接着定义结构体 fd_elem，因为之前也提到过许多系统调用都是要操作文件的，这个结构体能将线程与它要操作的文件以及文件 fd 号关联起来。我们还要维护一个列表 file_list 来报错当前所有已打开的文件。

```
syscall_vec[SYS_EXIT] = (handler)sys_exit;
syscall_vec[SYS_HALT] = (handler)sys_halt;
syscall_vec[SYS_CREATE] = (handler)sys_create;
syscall_vec[SYS_OPEN] = (handler)sys_open;
syscall_vec[SYS_CLOSE] = (handler)sys_close;
syscall_vec[SYS_READ] = (handler)sys_read;
syscall_vec[SYS_WRITE] = (handler)sys_write;
syscall_vec[SYS_EXEC] = (handler)sys_exec;
syscall_vec[SYS_WAIT] = (handler)sys_wait;
syscall_vec[SYS_FILESIZE] = (handler)sys_filesize;
syscall_vec[SYS_SEEK] = (handler)sys_seek;
syscall_vec[SYS_TELL] = (handler)sys_tell;
syscall_vec[SYS_REMOVE] = (handler)sys_remove;

list_init (&file_list);
lock_init (&file_lock);
```

根据定义顺序，给出了调用函数的 syscall_vec 向量表中的位置。并对文件列表和文件锁表进行初始化。

```
/* modified by Xiebin for system call */
handler h;
int *p;
int ret;

p = f->esp;

if (!is_user_vaddr (p))
  goto terminate;

if (*p < SYS_HALT || *p > SYS_INUMBER)
  goto terminate;

h = syscall_vec[*p];

if (!(is_user_vaddr (p + 1) && is_user_vaddr (p + 2) && is_user_vaddr (p + 3)))
  goto terminate;
```

```
ret = h (*(p + 1), *(p + 2), *(p + 3));

f->eax = ret;

return;

terminate:
  sys_exit (-1);
  /*modify end */
```

P=f->esp;获取这次要调用的系统编号所存放的地址。判断如果不能访问地址或者地址值不对的话，则跳转到最后面，以-1 退出系统调用。否则，从 syscall_vec[]中取出需要系统调用的赋给 h。之后的 if 语句是为了判断接下来的三个参数必须是在用户虚存内的，如果是，则调用相应的系统调用，又因为：

The 80x86 convention for function return values is to place them in the EAX register.

所以最后把返回值还给 ret。

- sys_exit(int status)

```
int
sys_exit (int status)
{
  /* Close all the files */
  struct thread *t;
  struct list_elem *l;

  t = thread_current ();
  while (!list_empty (&t->files))
    {
      l = list_begin (&t->files);
      sys_close (list_entry (l, struct fd_elem, thread_elem)->fd);
    }

  t->ret_status = status;
  thread_exit ();
  return -1;
}
```

退出的是当前线程，因此先获取指向当前线程的指针 t,并通过一个 while 循环关闭所有它打开的文件。关闭文件的操作由另外一个系统调用 sys_close()完成，然后把 t 的 ret_status 置为参数 status，并使用 thread_exit()来真正退出当前线程。

- sys_write(int fd,cosnt void *buffer,unsigned size)

```
static int
sys_write (int fd, const void *buffer, unsigned length)
{
  struct file * f;
  int ret;

  ret = -1;
  lock_acquire (&file_lock);
  if (fd == STDOUT_FILENO) /* stdout */
    putbuf (buffer, length);
  else if (fd == STDIN_FILENO) /* stdin */
    goto done;
  else if (!is_user_vaddr (buffer) || !is_user_vaddr (buffer + length))
    {
      lock_release (&file_lock);
      sys_exit (-1);
    }
  else
    {
      f = find_file_by_fd (fd);
      if (!f)
        goto done;
```

```
        ret = file_write (f, buffer, length);
      }

done:
  lock_release (&file_lock);
  return ret;
}
```

首先检查 fd，如果是 1(标准输出)我们向控制台输出信息，如果是 0(标

准输入)转入错误处理部分。接着判断缓冲区是否超出用户虚存地址，若合法则调用 find_file_by_fd(fd)找到要写入的文件 f，然后使用 file_write()来向 f 写入。最后返回实际写入的字符数 ret。

```
static struct file *
find_file_by_fd (int fd)
{
  struct fd_elem *ret;

  ret = find_fd_elem_by_fd (fd);
  if (!ret)
    return NULL;
  return ret->file;
}
```

它调用的是另一个辅助函数 find_fd_elem_by_fd，即通过 fd 号找到 fd_elem。在 find_fd_elem_by_fd 中遍历整个 file_list.

```
static struct fd_elem *
find_fd_elem_by_fd (int fd)
{
  struct fd_elem *ret;
  struct list_elem *l;

  for (l = list_begin (&file_list); l != list_end (&file_list); l = list_next (l))
    {
      ret = list_entry (l, struct fd_elem, elem);
      if (ret->fd == fd)
        return ret;
    }

  return NULL;
}
```

- sys_halt(void)

```
static int
sys_halt (void)
{
  power_off ();
}

static int
sys_create (const char *file, unsigned initial_size)
{
  if (!file)
    return sys_exit (-1);
  return filesys_create (file, initial_size);
}
```

调用 power_off()来进行关机处理。

- sys_create(const char *file,unsigned initial_size)

```
static int
sys_create (const char *file, unsigned initial_size)
{
  if (!file)
    return sys_exit (-1);
  return filesys_create (file, initial_size);
}
```

调用 filesys_create()来创建文件。

- sys_open(const char *file)

```
static int
sys_open (const char *file)
{
  struct file *f;
  struct fd_elem *fde;
  int ret;

  ret = -1; /* Initialize to -1 */
  if (!file) /* file == NULL */
    return -1;
  if (!is_user_vaddr (file))
    sys_exit (-1);
  f = filesys_open (file);
  if (!f) /* Bad file name */
    goto done;

  fde = (struct fd_elem *)malloc (sizeof (struct fd_elem));
  if (!fde) /* Not enough memory */
    {
      file_close (f);
      goto done;
    }
```

```
  fde->file = f;
  fde->fd = alloc_fid ();
  list_push_back (&file_list, &fde->elem);
  list_push_back (&thread_current ()->files, &fde->thread_elem);
  ret = fde->fd;
done:
  return ret;
}
```

为要打开的文件创建一个 fd 号。首先判断这个文件的文件名有效且文件位于用户虚存内。然后调用 filesys_open(file)来打开它，如果打开成功就为该文件建立一个 fd_elem 名为 fde。将 fde 的 file 属性设为 filesys_open(file)返回的 file 结构体 f,并调用 alloc_fid()分配一个 fd 号给这个打开的文件。在 file_list 里插入 fde 的 elem，为了与当前线程对应把 fde 的 thread_elem 插入当前线程的 files 列表里。最后，返回被打开文件的 fd 号。

```
static int
alloc_fid (void)
{
  static int fid = 2;
  return fid++;
}
```

用来分配 fd 号，因为 0 与 1 已经留作标准输入与输出了,所以从 2 开始分配，每分配一个 fd 就加 1，也能保证不会重复。

- sys_close(int fd)

```
static int
sys_close(int fd)
{
  struct fd_elem *f;
  int ret;

  f = find_fd_elem_by_fd_in_process (fd);

  if (!f) /* Bad fd */
    goto done;
  file_close (f->file);
  list_remove (&f->elem);
  list_remove (&f->thread_elem);
  free (f);

done:
  return 0;
}
```

关闭文件。调用 find_fd_elem_by_fd_in_process(fd)得到 fd 对应的 fd_elem 并赋给 f，然后调用 file_close(f->file)关闭 f 对应的文件。接着将 f 的 elem 和 thread_elem 分别从 file_list 与当前线程的 file 这两个列表中移除，最后释放 f。

```
static struct fd_elem *
find_fd_elem_by_fd_in_process (int fd)
{
  struct fd_elem *ret;
  struct list_elem *l;
  struct thread *t;

  t = thread_current ();

  for (l = list_begin (&t->files); l != list_end (&t->files); l = list_next (l))
    {
      ret = list_entry (l, struct fd_elem, thread_elem);
      if (ret->fd == fd)
        return ret;
    }

  return NULL;
}
```

在当前线程打开的文件列表里找文件。因为要读取或写入等操作的文件对象在 file_list 里面，所以 sys_write()与 sys_read()等都是使用 find_file_by_fd 来查找文件。因为关闭文件的时候只会关闭当前线程打开的文件，所以 sys_close 则使用 find_fd_elem_by_fd_in_process。

- sys_read(int fd,void *buffer,unsigned size)

```
static int
sys_read (int fd, void *buffer, unsigned size)
{
  struct file * f;
  unsigned i;
  int ret;

  ret = -1; /* Initialize to zero */
  lock_acquire (&file_lock);
  if (fd == STDIN_FILENO) /* stdin */
    {
      for (i = 0; i != size; ++i)
        *(uint8_t *)(buffer + i) = input_getc ();
      ret = size;
      goto done;
    }
  else if (fd == STDOUT_FILENO) /* stdout */
      goto done;
  else if (!is_user_vaddr (buffer) || !is_user_vaddr (buffer + size)) /* bad ptr */
    {
      lock_release (&file_lock);
      sys_exit (-1);
    }
```

```
  else
    {
      f = find_file_by_fd (fd);
      if (!f)
        goto done;
      ret = file_read (f, buffer, size);
    }

done:
  lock_release (&file_lock);
  return ret;
}
```

读取文件内容。对 fd 以及 buffer 的检验与 sys_write 一样，只不过在这里 fd 为 0(标准输入)时从键盘读入，为 1(标准输出)时出错。如果 fd 不

为 0 也不为 1，再通过 find_file_by_fd(fd)找出对应文件然后调用 file_read()来读取。最后返回实际读取的字符数 ret。

- sys_exec(const char *cmd_line)

```
static int
sys_exec (const char *cmd)
{
  int ret;

  if (!cmd || !is_user_vaddr (cmd)) /* bad ptr */
    return -1;
  lock_acquire (&file_lock);
  ret = process_execute (cmd);
  lock_release (&file_lock);
  return ret;
}
```

执行用户输入的命令。它的参数是命令行传来的用户输入 cmd。直接调用 process_execute(cmd)，判断如果命令为空或者非法访问内存时需要返回。lock_acquire 和 lock_release()对文件进行互斥操作，需要获取和释放互斥锁.

- sys_wait (pid_t pid)

```
static int
sys_wait (pid_t pid)
{
  return process_wait (pid);
}
```

要求等待参数 pid 所指向的进程消亡。直接调用 process_wait(pid).

- sys_filesize(int fd)

```
static int
sys_filesize (int fd)
{
  struct file *f;

  f = find_file_by_fd (fd);
  if (!f)
    return -1;
  return file_length (f);
}
```

返回 fd 对应文件的大小，单位是字节，通过调用 file_length()实现。

- sys_tell(int fd)

```
static int
sys_tell (int fd)
{
  struct file *f;

  f = find_file_by_fd (fd);
  if (!f)
    return -1;
  return file_tell (f);
}
```

返回 fd 对应的文件中下一个被操作的位置，通过调用 file_tell()实现。

- sys_seek(int fd,unsigned position)

```
static int
sys_seek (int fd, unsigned pos)
{
  struct file *f;

  f = find_file_by_fd (fd);
  if (!f)
    return -1;
  file_seek (f, pos);
  return 0; /* Not used */
}
```

把 fd 对应的文件下一个要操作的位置设为传入的参数 pos，通过调用 file_seek()实现。

- sys_remove(cosnt char *file)

```
static int
sys_remove (const char *file)
{
  if (!file)
    return false;
  if (!is_user_vaddr (file))
    sys_exit (-1);

  return filesys_remove (file);
}
```

删除文件，通过调用 filesys_remove()实现。

5. Process termination messages(进程退出信息)

【需求分析】

3.3.2 Process Termination Messages

Whenever a user process terminates, because it called `exit` or for any other reason, print the process's name and exit code, formatted as if printed by `printf ("%s: exit(%d)\n", ...);`. The name printed should be the full name passed to `process_execute()`, omitting command-line arguments. Do not print these messages when a kernel thread that is not a user process terminates, or when the `halt` system call is invoked. The message is optional when a process fails to load.

Aside from this, don't print any other messages that Pintos as provided doesn't already print. You may find extra messages useful during debugging, but they will confuse the grading scripts and thus lower your score.

在运行用户程序结束时，我们需要得到进程退出信息，以反映给用户进程的运行情况，这些进程退出信息有时是用户需要计算机来运算的结果，有时是用户文件操作是否成功的标志，当用户进程因为不当操作被终止时，我们需要通知用户来修改自己的程序，可见，进程退出信息是十分必要的。

【实现方案】

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);

6. Denying write to in-use executable files（禁止写入可执行文件）

【需求分析】

3.3.5 Denying Writes to Executables

Add code to deny writes to files in use as executables. Many OSes do this because of the unpredictable results if a process tried to run code that was in the midst of being changed on disk. This is especially important once virtual memory is implemented in project 3, but it can't hurt even now.

You can use `file_deny_write()` to prevent writes to an open file. Calling `file_allow_write()` on the file will re-enable them (unless the file is denied writes by another opener). Closing a file will also re-enable writes. Thus, to deny writes to a process's executable, you must keep it open as long as the process is still running.

由于 Pintos 没有实现在运行某程序时，它所打开的文件不被修改的功能，很可能因为被其他程序修改而造成意想不到的后果。在 file.c 的 struct file 中已经为 file 定义了 bool deny_write 的属性以判断是否不允许被写，同时提供了 file_deny_write()和 file_allow_write()的接口。我们需要在适当地方调用这两个函数以控制文件允许/不允许被写。

【实现方案】

```
132     if (success)
133       {
134         t->self = filesys_open (file_name);
135         file_deny_write (t->self);
136         if_.esp -= file_name_len + 1;
137         start = if_.esp;
```

134 行，打开可执行文件并将其设置为该线程的映像文件，135 行，禁止其他文件向该文件进行写操作，file_deny_write(t->self)拒绝其他线程对 file_name 的写入操作。

7. 实验结果

结果部分截图

```
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
All 76 tests passed.
```

8. 实验总结

总的来说，project2 的难度相较于 project1 要更大一些。不仅仅是任务触及到了系统更底层的地方，需要我们去阅读和理解更多复杂的内容；更麻烦的是和 project2 相关的资料和解析在网上非常少，更要求我们去思考和理解整个实验的过程。整个实验中，参数传递和系统调用是实验的难点，也是实验的核心要求。这两个任务花费了我们许多的时间和精力，当然，安全访问内存也给我们造成了一定的麻烦。例如，在传递参数的时候，我们需要掌握好其传递的时机。由于 pintos 在初始化时并不会为用户地址空间建立实存与虚存的映射，所以如果在这个时候进行参数的传递则会导致 page_fault。操作系统为用户分配地址空间的操作是在 load()之中，在调用了 load()之后，通过一个中断返回进入 elf 文件入口，这个时候才可以进行参数的传递了。因为这样才有栈的存在。同时还要注意的是，参数传递工作必须在用户 main 函数被执行之前完成，所以也不能放在太晚进行。关于参数传递，还有一点值得注意的就是对齐内存 alignment，使字符部分的长度为 4 的整数倍，这样在访问时的速度会加快。而对于系统调用来说，有一个很重要的点就是调用号。通常，当用户程序调用系统调用之后，会引发 30 号中断，从而通过中断向量指向并运行 syscall_handler()函数，在 syscall_handler()函数中，根据系统调用号来调用不同的系统调用。所以，在 syscall_handler()函数中，得到调用号就十分重要，而调用号恰恰藏在了 syscall_handler()函数参数的中断帧之中，从而得到调用号，再以此来确定要调用的函数的指针，完成调用。同时系统调用中用到了一些第四个 Task File System 中的一些内容 file 文件中的一些函数，运用得好对于系统调用函数的实现会带来方便。还有安全访问内存在防止用户访问其余用户虚拟内存的问题时，通过在 page_fault 函数里加判断是难以解决的。很久才想到在 system call 的调用机制里要加上错误检查，在检查出错误时直接终止该用户进程。并要利用 threads/pagedir.c 中的 bool lookup_page(uint32_t *pd, const void *vaddr, bool create)这个函数，第一个参数指的是页表索引，第二个参数代表需要访问的虚存地址,函数的功能是判断该虚存地址是否在该页表索引指向的页中，第三个参数是一个执行选项，当 create 为 true 时，激活该虚存地址所在页，否则错误处理只做判断。我们利用这个函数，以及 struct thread 的成员变量 pagedir，只利用其判断功能可以得知当前线程是否访问合法的地址，终于能处理一个用户进程访问其它用户进程内存的错误。

总而言之，整个 project2 实验对我们都是一个巨大的挑战，为她奋斗数个日炎，打扰多位学长学姐，不过当所有问题都迎刃解决的时候，还是非常开心的，也让我们对于操作系统基层的原理有了更深刻的认识。总之，整个学期的操作系统课程还是让我们受益匪浅，学到了许多的新知识。当然，也很高兴认识高老师和肖冰助教！