附录：

参数传递

==========

（一）准备过程

在实验开始之前，我们需要了解一个用户进程从创建到死亡的过程。以下给出函数调用顺序

init.c

main（pintos 主进程）->run_action->run_task->process_execute

process.c

process_execute-> thread_create

thread.c

thread_create ->thread_unblock

process.c

process_execute

init.c

run_task-> process_wait

process.c

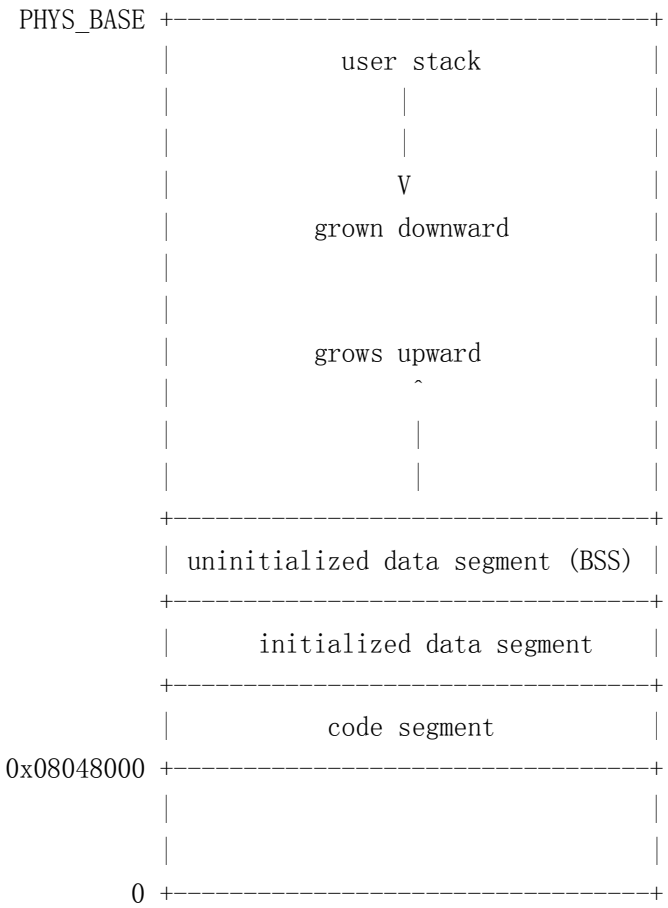
process_wait-> start_process

（二）

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 (*) ()

上图为文档中提供的用户空间的参数存储格式，代码中即按该格式放入栈中。

（三）系统存放数据的位置


PHYS_BASE
user stack
grown downward
grows upward
uninitialized data segment (BSS)
initialized data segment
code segment
0x08048000
0

（四）以下为修改代码

Process.c

/*主要解决了参数传递，内存的安全访问问题*/

添加

```
#include "threads/malloc.h"
#include "userprog/syscall.h"
```

用到了内存分配以及系统调用，故添加两项包含。

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


  char *save;//用于存分离的文件名和参数
  char *fn;//用于存分离的文件名

  struct thread *t;

  tid = TID_ERROR;//默认tid为TID_ERROR


  /* 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);


  fn = malloc (strlen (file_name) + 1);//分配存储的内存
  if (!fn)//异常处理，当分配失败
    goto done;
  memcpy (fn, file_name, strlen (file_name) + 1);//内存copy，将filename中的文件名及参数全部拷贝到fn中
  file_name = strtok_r (fn, " ", &save);//用strtok_r函数将文件名和各个参数分别分开，由于该函数特性，多空格问题也顺带解决


  /* Create a new thread to execute FILE_NAME. */
  /* 原来的实现
  tid = thread_create (file_name, PRI_DEFAULT, start_process, fn_copy); */


  tid = thread_create (fn, PRI_DEFAULT, start_process, fn_copy);//用分离出来的文件名fn创建一个新的进程
  if (tid == TID_ERROR)//异常处理，当创建失败
    goto done;

  t = get_thread_by_tid (tid);//通过tid找到进程，用以等待子进程载入
  sema_down (&t->wait);/*获得锁,用以解决用户线程与内核线程的优先级相同问题。
                         在process.c中定义一把锁 struct semaphore. （初始化为1）
                        用户线程一旦被创建就会调用sema_down来得到这把锁。(在process_execute ()中，
                        tid = thread_create (fn, PRI_DEFAULT, start_process, fn_copy)之后)
                        之后内核线程再调用sema_down，会被睡眠。直到用户线程释放这把锁
                        用户线程再退出时sema_up()，使得内核线程得以苏醒。
*/
  if (t->ret_status == -1)//异常处理
    tid = TID_ERROR;
  while (t->status == THREAD_BLOCKED)//如果已经block了，则unblock
    thread_unblock (t);
  if (t->ret_status == -1)
    process_wait (t->tid);

done:
  free (fn);//释放申请的内存

  if (tid == TID_ERROR)
    palloc_free_page (fn_copy);
  return tid;
```

以上为 process_execute 函数中的修改，各部分功能已在注释中写明

```
static void
start_process (void *file_name_)
{
  char *file_name = file_name_;
  struct intr_frame if_;
  bool success;


  char *token, *save_ptr;//临时变量，用以存储分离的参数
  void *start;//临时变量，用以记录参数首地址
  int argc, i;//临时变量，用以记录参数数目
  int *argv_off; //用以存储到首地址的偏移量
  size_t file_name_len;
  struct thread *t;


  /* 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;


  t = thread_current ();//获得当前进程
  argc = 0;
  argv_off = malloc (32 * sizeof (int));//为参数存储分配内存
  if (!argv_off)//异常处理，当内存分配失败
    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;
        }                                                    //用循环及strtok_r将各个参数的偏移量记录在对应的argv_off中


  success = load (file_name, &if_.eip, &if_.esp);


//以下为按照文档中给出的用户栈中的参数存储的方式讲数据写入栈中
  if (success)
    {
      t->self = filesys_open (file_name);//打开文件
      file_deny_write (t->self);//文件拒绝写
      if_.esp -= file_name_len + 1;//将指针移到对应位置
      start = if_.esp;//初始地址置为栈顶
      memcpy (if_.esp, file_name, file_name_len + 1);//内存copy，将文件名和参数copy到对应的位置
      if_.esp -= 4 - (file_name_len + 1) % 4; //对齐
      if_.esp -= 4;//继续向下移动
      *(int *)(if_.esp) = 0; //按照格式置为0
      //通过记录的偏移量将各个参数的地址写到对应的位置
      for (i = argc - 1; i >= 0; --i)
        {
          if .esp -= 4;
```

```
          *(void **)(if_.esp) = start + argv_off[i]; //参数地址=首地址+偏移量
        }

      if_.esp -= 4;//继续向下移动
      *(char **)(if_.esp) = (if_.esp + 4);
      if_.esp -= 4;
      *(int *)(if_.esp) = argc;//记录参数数量
      if_.esp -= 4;
      *(int *)(if_.esp) = 0; //按照格式写return address

      sema_up (&t->wait);//释放锁，作用在上一个函数中写明
      intr_disable ();//关中断，以保证下面操作的原子性
      thread_block ();
      intr_enable ();
    }
//以下为异常处理
  else
    {
      free (argv_off);
exit:
      t->ret_status = -1;
      sema_up (&t->wait);
      intr_disable ();
      thread_block ();
      intr_enable ();
      thread_exit ();
    }

  free (argv_off);//释放内存


  /* If load failed, quit. */
  palloc_free_page (file_name);
  /* Old Implementation
  if (!success)
    thread_exit (); */

  /* Start the user process by simulating a return from an
     interrupt, implemented by intr_exit (in
     threads/intr-stubs.S).  Because intr_exit takes all of its
     arguments on the stack in the form of a `struct intr_frame',
     we just point the stack pointer (%esp) to our stack frame
     and jump to it. */
  asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) : "memory");
  NOT_REACHED ();
}
```

以上为 start_process 的修改，配合 process_excute 基本解决参数传递问题。

```
int
process_wait (tid_t child_tid /* Old Implementation UNUSED */)
{
  /* 旧的实现，直接return -1，故用户程序实际上没有能得到执行
  return -1; */


  struct thread *t;
  int ret;

  t = get_thread_by_tid (child_tid);
  if (!t || t->status == THREAD_DYING || t->parent == thread_current ())//判断异常
    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);//状态信息输出
  while (t->status == THREAD_BLOCKED)
    thread_unblock (t);

  return ret;

}

/* Free the current process's resources. */
void
process_exit (void)
{
  struct thread *cur = thread_current ();
  uint32_t *pd;

 //进程结束
  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 ();
```

将进程等待与退出正确实行。其中需要用到的相关函数（如 get_thread_by_tid）在后面会有介绍。

系统调用

==========

（一）关于系统调用的实现

```
#ifndef USERPROG_SYSCALL_H
#define USERPROG_SYSCALL_H

void syscall_init (void);


int sys_exit (int status);

#endif /* userprog/syscall.h */
```

以上为 syscall.h 中的修改，加了一条 sys_exit.

以下为 syscall.c 中修改

```
#include "threads/vaddr.h"
#include "threads/init.h"
#include "userprog/process.h"
#include <list.h>
#include "filesys/file.h"
#include "filesys/filesys.h"
#include "threads/palloc.h"
#include "threads/malloc.h"
#include "devices/input.h"
#include "threads/synch.h"
```

（二）修改 struct thread

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */

    /* My Implementation */
    struct alarm alrm;                  /* alarm object */
    int base_priority;                  /* priority before donate, if nobody donates, then it should be same as prior
    struct list locks;                  /* the list of locks that it holds */
    bool donated;                       /* whether the thread has been donated priority */
    struct lock *blocked;               /* by which lock this thread is blocked */

    int nice;                           /* nice value of a thread */
    int recent_cpu;                     /* recent cpu usage */
    /* == My Implementation */

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */

    /* My Implementation */
    struct semaphore wait;              /* semaphore for process_wait */
    int ret_status;                     /* return status */
    struct list files;                  /* all opened files */
    struct file *self;                  /* the image file on the disk */
    struct thread *parent;              /* parent process */
    /* == My Implementation */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```

（三）引入 fd(文件的 id)
增加关联 fd 和指向文件的指针的数据结构

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

添加fd控制函数，使得程序能够通过fd号得到相应的文件的指针

```
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;
}

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;
}
```

（四）系统调用过程
① 系统调用号（定义在lib/syscall-nr.h下的一组枚举变量）

```
  /* Projects 2 and later. */
  SYS_HALT,                   /* Halt the operating system. */
  SYS_EXIT,                   /* Terminate this process. */
  SYS_EXEC,                   /* Start another process. */
  SYS_WAIT,                   /* Wait for a child process to die. */
  SYS_CREATE,                 /* Create a file. */
  SYS_REMOVE,                 /* Delete a file. */
  SYS_OPEN,                   /* Open a file. */
  SYS_FILESIZE,               /* Obtain a file's size. */
  SYS_READ,                   /* Read from a file. */
  SYS_WRITE,                  /* Write to a file. */
  SYS_SEEK,                   /* Change position in a file. */
  SYS_TELL,                   /* Report current position in a file. */
  SYS_CLOSE,                  /* Close a file. */
```

② 初始化中断（syscall.c/ syscall_init函数）
为每个系统调用申明一个操作函数：

```
static int sys_write (int fd, const void *buffer, unsigned length);//write系统调用
static int sys_halt (void);//终止系统调用
static int sys_create (const char *file, unsigned initial_size);//文件创建
static int sys_open (const char *file);//打开文件
static int sys_close (int fd);//关闭文件
static int sys_read (int fd, void *buffer, unsigned size);//读文件
static int sys_exec (const char *cmd);//运行一个可执行文件
static int sys_wait (pid_t pid);//等待进程pid死亡并且通过exit返回状态
static int sys_filesize (int fd);//获取文件大小
static int sys_tell (int fd);//返回Open 语句打开的文件fd中指定下一个读/写位置，表示为从文件开始的byte数。
static int sys_seek (int fd, unsigned pos);//在 Open 语句打开的文件中指定当前的读/写位置，表示为从文件开始的byte数
static int sys_remove (const char *file);//删除一个叫file的文件
```

初始化30号中断（即系统调用），使其指向syscall_handler函数

```
intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");
```

初始化系统调用列表，使其指向相应的函数指针

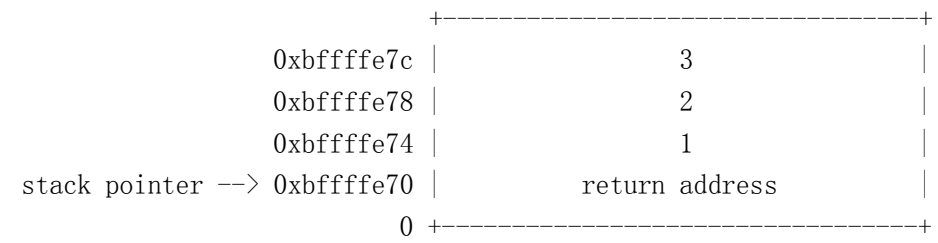
```
syscall_vec[SYS_EXIT] = (handler)sys_exit;
syscall_vec[SYS_HALT] = (handler)sys_halt;
syscall_vec[SYS_CREATE] = (handler)sys_create;
syscall_vec[SYS_OPEN] = (handler)sys_open;
syscall_vec[SYS_CLOSE] = (handler)sys_close;
syscall_vec[SYS_READ] = (handler)sys_read;
syscall_vec[SYS_WRITE] = (handler)sys_write;
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;
```

初始化系统打开文件列表和文件锁

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

③当pintos进行系统调用的时候，它先会调用静态函数syscall_handler (struct intr_frame *f)。Intr_frame是指向用户程序寄存器。这里的寄存器是指第一部分参数传递所压栈的数据。

例如：



address里存系统调用号，1 2 3（只是一个例子，可以是一个参数，也可以是多个参数）分别是三个参数指针的指针。

1、传数据

传递系统调用号：

```
int *p = f->esp;
```

传递三个参数：

```
int arg1 = *(p+1);
int arg2 = *(p+2);
int arg3 = *(p+3);
```

2、判断是否合法

```
if (!is_user_vaddr())
    sys_exit(-1);
```

具体实现如下（syscall.c/ syscall_handler函数）

```
handler h;
```

```
static void
syscall_handler (struct intr_frame *f /* Old Implementation UNUSED */)
{
  /* 旧的实现，直接输出一句信息
  printf ("system call!\n");
  thread_exit (); */

  handler h;
  int *p;
  int ret;

  p = f->esp;

  if (!is_user_vaddr (p))
    goto terminate;

  if (*p < SYS_HALT || *p > SYS_INUMBER)//异常处理
    goto terminate;

  h = syscall_vec[*p];//获得向量中的系统调用号

  if (!(is_user_vaddr (p + 1) && is_user_vaddr (p + 2) && is_user_vaddr (p + 3)))//判断是否需要继续执行
    goto terminate;

  ret = h (*(p + 1), *(p + 2), *(p + 3));//处理对应的系统调用

  f->eax = ret;

  return;

terminate:
  sys_exit (-1);

}
```

（五）用到了虚地址空间，list 结构，文件系统及内存分配等相关内容，故添加相关包含。

system call 的具体步骤：

将要进行的操作以及参数存入栈中

通过软件中断切换到操作系统的 kernel 模式（int$0x30）

在操作系统的 kenel 模式下读取并验证参数，通过读取 struct intr_frame 中的栈 esp 获取

执行请求的操作

保存进程的结果,保存到 eax 栈中

从终端中返回进程

（六）系统调用的实现

写操作（sys_write (int fd, const void *buffer, unsigned length)）

1、在写的时候，我们需要给文件加锁，防止在读的过程中被改动

2、先判断是否是标准写入流，如果是标准写入的话，直接调用putbuf()写到控制台，如果是标准写入流，则调用sys_exit(-1)，如果不是标准读或者标准写，则说明是从文件写入。判断指向buffer指针是否正确（是否有效且在用户空间），如果正确，则根据fd找到文件，然后调用file_write(f, buffer, size)函数写入buffer到文件，反之则调用sys_exit(-1)退出。

3、注意在退出之前或者写文件完成之后要释放锁

4、具体代码实现如下：

```
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))//异常处理，防止用户进程访问内核空间
    {
      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;
}
```

进程终止（sys_exit (int status)）

1、 得到当前用户线程的指针

2、 讲该用户线程的对应的文件打开列表清空，并关闭对应的文件

3、 调用 thread_exit()函数，并返回-1 表示结束进程

4、 在 thread.c 里面我们添加了 process_exit()函数，并移除所有子线程并关闭文件

```
int
sys_exit (int status)
{

  struct thread *t;
  struct list_elem *l;

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

  t->ret_status = status;
  thread_exit ();//结束进程
  return -1;
}

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);
}
```

创建文件（sys_create (const char *file, unsigned initial_size)）

1、 获得需要创建的文件的文件名称

2、 如果文件名为空，则返回-1 退出，如果存在，则调用 filesys.c 下面的 filesys_create()函数

3、 具体代码实现如上：

打开文件（sys_open (const char *file)）

1、 定义返回值为打开文件的fd，如果打开失败，则返回-1

2、 判断传进来的文件名，如果为空或者它的地址不在用户空间，返回-1

3、 调用filesys_open(file)函数，如果打开失败（原因是文件名对应的文件不存在），则返回-1

4、 分配空间个fd对应的struct fde，如果内存空间不够，则调用file_close(f)关闭文件，返回-1

5、 初始化fde，并将其压入系统打开文件列表和进程打开文件列表相对应的栈中，并返回对应的fd号

6、 具体代码实现如下：

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

  ret = -1; //初始化为-1
  if (!file) //异常处理，不存在文件
    return -1;
  if (!is_user_vaddr (file))//确认在用户程序空间中
    sys_exit (-1);
  f = filesys_open (file);
  if (!f) //文件名字不符合文件系统的规则
    goto done;

  fde = (struct fd_elem *)malloc (sizeof (struct fd_elem));
  if (!fde) //内存不足
    {
      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;
}
```

关闭文件（sys_close(int fd)）

1、 根据 fd 找到系统中对应的打开文件

2、 判断文件是否存在，如果不存在，则不需要关闭，返回 0，如果存在，则调用 file_close(f) 将其关闭，并讲对应的 fd 从系统打开文件列表和进程打开文件列表中删除

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

  f = find_fd_elem_by_fd_in_process (fd);

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

done:
  return 0;
}
```

读操作（static int sys_read (int fd, void *buffer, unsigned size)）

1、 在读的时候，我们需要给文件加锁，防止在读的过程中被改动

2、 先判断是否是标准读入流，如果是标准读入的话，直接调用input_getc()从控制台读入，如果是标准写入流，则调用sys_exit(-1)，如果不是标准读或者标准写，则说明是从文件读入。判断指向buffer指针是否正确（是否有效且在用户空间），如果正确，则根据fd找到文件，然后调用file_read(f, buffer, size)函数读取到buffer，反之则调用sys_exit(-1)退出。

3、 注意在退出之前或者读文件完成之后要释放锁

4、具体代码实现如下

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

  ret = -1;
  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)) //异常处理
    {
      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;
}
```

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

  if (!cmd || !is_user_vaddr (cmd)) //异常处理
    return -1;
  lock_acquire (&file_lock);
  ret = process_execute (cmd);
  lock_release (&file_lock);
  return ret;
}

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

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;
}

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;
}
```

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

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

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

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

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

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;
}

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

  return filesys_remove (file);
}
```

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

  return filesys_remove (file);
}

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;
}
```

主要是调用已有的一些调用来实现，其中保证了用户进程不会访问内核空间中的东西。达到了各类异常处理的目的。

为了实现这些东西，对 thread 的结构做了一些修改和完善

以下为 threads 中的修改

```
static bool thread_sort_less (const struct list_elem *lhs, const struct list_elem *rhs, void *aux UNUSED);
static bool thread_insert_less_head (const struct list_elem *lhs, const struct list_elem *rhs, void *aux UNUSED);
static bool thread_insert_less_tail (const struct list_elem *lhs, const struct list_elem *rhs, void *aux UNUSED);

static void thread_calculate_priority_other (struct thread *curr);
static void thread_calculate_recent_cpu_other (struct thread *curr);

static int load_avg;
```

添加了以上的函数和性质

```c
/* 告诉 list_sort 如何排列 ready list
 */
static bool
thread_sort_less (const struct list_elem *lhs, const struct list_elem *rhs, void *aux UNUSED)
{
  struct thread *a, *b;

  ASSERT (lhs != NULL && rhs != NULL);

  a = list_entry (lhs, struct thread, elem);
  b = list_entry (rhs, struct thread, elem);

  return (a->priority > b->priority);
}


void
sort_thread_list (struct list *l)
{
  if (list_empty (l))
    return;

  list_sort (l, thread_sort_less, NULL);
}
/* 告诉 list_sort 如何排列 ready list
 */
static bool
thread_insert_less_head (const struct list_elem *lhs, const struct list_elem *rhs, void *aux UNUSED)
{
  struct thread *a, *b;

  ASSERT (lhs != NULL && rhs != NULL);

  a = list_entry (lhs, struct thread, elem);
  b = list_entry (rhs, struct thread, elem);

  return (a->priority >= b->priority);
}


static bool
thread_insert_less_tail (const struct list_elem *lhs, const struct list_elem *rhs, void *aux UNUSED)
{
  return thread_sort_less (lhs, rhs, NULL);
}

struct thread *
get_thread_by_tid (tid_t tid)
{
  struct list_elem *f;
  struct thread *ret;

  ret = NULL;
  for (f = list_begin (&all_list); f != list_end (&all_list); f = list_next (f))
    {
      ret = list_entry (f, struct thread, allelem);
      ASSERT (is_thread (ret));
      if (ret->tid == tid)
        return ret;
```

```c
void sort_thread_list (struct list *l);
void thread_set_priority_other (struct thread *curr, int new_priority, bool forced);
void thread_yield_head (struct thread *curr);

void thread_calculate_load_avg (void);
void thread_calculate_recent_cpu (void);
void thread_calculate_priority (void);
void thread_calculate_recent_cpu_for_all (void);
void thread_calculate_priority_for_all (void);
struct thread *get_thread_by_tid (tid_t);
```

添加了上述功能函数（在 process.c 等处使用）

结果

```
pass tests/userprog/open-missing
pass tests/userprog/open-boundary
pass tests/userprog/open-empty
pass tests/userprog/open-null
pass tests/userprog/open-bad-ptr
pass tests/userprog/open-twice
pass tests/userprog/close-normal
pass tests/userprog/close-twice
pass tests/userprog/close-stdin
pass tests/userprog/close-stdout
pass tests/userprog/close-bad-fd
pass tests/userprog/read-normal
pass tests/userprog/read-bad-ptr
pass tests/userprog/read-boundary
pass tests/userprog/read-zero
pass tests/userprog/read-stdout
pass tests/userprog/read-bad-fd
pass tests/userprog/write-normal
pass tests/userprog/write-bad-ptr
pass tests/userprog/write-boundary
pass tests/userprog/write-zero
pass tests/userprog/write-stdin
pass tests/userprog/write-bad-fd
pass tests/userprog/exec-once
pass tests/userprog/exec-arg
pass tests/userprog/exec-multiple
pass tests/userprog/exec-missing
pass tests/userprog/exec-bad-ptr
pass tests/userprog/wait-simple
pass tests/userprog/wait-twice
pass tests/userprog/wait-killed
pass tests/userprog/wait-bad-pid
pass tests/userprog/multi-recurse
pass tests/userprog/multi-child-fd
pass tests/userprog/rox-simple
pass tests/userprog/rox-child
pass tests/userprog/rox-multichild
pass tests/userprog/bad-read
pass tests/userprog/bad-write
pass tests/userprog/bad-read2
pass tests/userprog/bad-write2
pass tests/userprog/bad-jump
pass tests/userprog/bad-jump2
pass tests/userprog/no-vm/multi-oom
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
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.
```