华东师范大学软件工程学院实验报告

姓　　名:	李鹏达	学　　号:	10225101460
实验编号:	Project 2	实验名称:	User Programs in Pintos

1　实验目的

1) 实现参数传递
2) 实现系统调用

2　实验内容与实验步骤

为防止之前的 Project1 可能存在的 bug 对本次实验造成影响，我们从未完成 Project1 的代码开始，进行本次实验的实现。

首先，我们将 `src/utils` 中的 `pintos` 和 `Pintos.pm` 中对应的 `src/threads` 改为 `src/userprog`。

2.1　参数传递

首先，我们需要再线程结构体中添加 `struct semaphore sema` 和 `struct thread* parent`，分别用于实现进程的同步和父进程的记录，以及 `bool success`，用于记录进程是否成功加载。

src/threads/thread.h

```
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. */

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

```
17      struct semaphore sema;
18      struct thread* parent;
19      bool success;
20  #endif
21
22      /* Owned by thread.c. */
23      unsigned magic;                     /* Detects stack overflow. */
24    };
```

然后在线程的初始化函数 thread_init() 中，对这些变量进行初始化。

src/threads/thread.c

```
1  static void
2  init_thread (struct thread *t, const char *name, int priority)
3  {
4    enum intr_level old_level;
5
6    ASSERT (t != NULL);
7    ASSERT (PRI_MIN <= priority && priority <= PRI_MAX);
8    ASSERT (name != NULL);
9
10   memset (t, 0, sizeof *t);
11   t->status = THREAD_BLOCKED;
12   strlcpy (t->name, name, sizeof t->name);
13   t->stack = (uint8_t *) t + PGSIZE;
14   t->priority = priority;
15   t->magic = THREAD_MAGIC;
16
17 #ifdef USERPROG
18   if (t == initial_thread)
19     t->parent = NULL;
20   else
21     t->parent = thread_current ();
22   sema_init (&t->sema, 0);
23   t->success = true;
24 #endif
25
26   old_level = intr_disable ();
27   list_push_back (&all_list, &t->allelem);
28   intr_set_level (old_level);
29 }
```

接下来，我们需要修改 src/userprog/process.c 中的 process_execute() 函数，使得其能够将参数传递给新的进程。

src/userprog/process.c - process_execute()

```
process_execute (const char *file_name)
{
  char *fn_copy;
  char *fn_copy2;
  char *save_ptr;
  char *token;
  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;
  fn_copy2 = palloc_get_page (0);
  if (fn_copy2 == NULL)
  {
    palloc_free_page(fn_copy);
    return TID_ERROR;
  }

  strlcpy (fn_copy, file_name, PGSIZE);
  strlcpy (fn_copy2, file_name, PGSIZE);

  token = strtok_r (fn_copy, " ", &save_ptr);

  /* Create a new thread to execute FILE_NAME. */
  tid = thread_create (token, PRI_DEFAULT, start_process, fn_copy2);
  palloc_free_page(fn_copy);
  if (tid == TID_ERROR)
  {
    palloc_free_page (fn_copy2);
    return tid;
  }
  sema_down(&thread_current ()->sema);
  if (!thread_current ()->success)
    return TID_ERROR;
  return tid;
}
```

在 process_execute() 中，我们首先将 file_name 复制到 fn_copy 和 fn_copy2 中，然后使用 strtok_r() 函数将 fn_copy 中的字符串按照空格分割，得到第一个参数，即可执行的程序名。然后，我们调用 thread_create() 函数创建一个新的线程，将第一个参数作为线程的名字，将 fn_copy2 作为线程的参数，将 start_process 作为线程的执行函数。最后，我们调用 sema_down() 函数，使得父进程等待子进程

执行完毕，然后再返回。

接下来，我们需要修改 `src/userprog/process.c` 中的 `start_process()` 函数，使得其能够将参数压入栈中。

为此，我们定义一个 `push_args()` 函数，用于将参数压入栈中。

其中压入参数的顺序和结构如图所示：

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 `0xbfffffcc`.

图 1: 压入参数的顺序和结构

src/userprog/process.c - push_args()

```
1  static void
2  push_args(void **esp, int argc, char* argv[])
3  {
4    int i;
5    *esp = (void *)((int)*esp & 0xfffffffc);
6    *esp -= 4;
7    *(uint32_t *) *esp = 0;
8    for (i = argc - 1; i >= 0; i--)
9    {
10     *esp -= 4;
11     *(uint32_t *) *esp = (uint32_t)argv[i];
12   }
13   *esp -= 4;
14   *(uint32_t *) *esp = (uint32_t)*esp + 4;
15   *esp -= 4;
16   *(uint32_t *) *esp = argc;
17   *esp -= 4;
18   *(uint32_t *) *esp = 0;
19 }
```

在 start_process() 函数中，我们首先将参数 file_name 复制到 fn_copy 中，然后使用 strtok_r() 函数将 fn_copy 中的字符串按照空格分割，得到第一个参数，即可执行的程序名。然后，调用 load() 函数加载程序，如果加载成功，则将参数压入栈中，然后唤醒父进程，否则，唤醒父进程并退出。

src/userprog/process.c - start_process()

```
1  static void
2  start_process (void *file_name_)
3  {
4    char *file_name = file_name_;
5    char *fn_copy;
6    char *save_ptr;
7    char *token;
8    struct intr_frame if_;
9    bool success;
10   int argc = 0;
11   char *argv[50];
12
13   fn_copy = malloc (strlen (file_name) + 1);
14   strlcpy (fn_copy, file_name, strlen (file_name) + 1);
15
16   /* Initialize interrupt frame and load executable. */
17   memset (&if_, 0, sizeof if_);
18   if_.gs = if_.fs = if_.es = if_.ds = if_.ss = SEL_UDSEG;
19   if_.cs = SEL_UCSEG;
20   if_.eflags = FLAG_IF | FLAG_MBS;
21
22   file_name = strtok_r (file_name, " ", &save_ptr);
23   success = load (file_name, &if_.eip, &if_.esp);
24
25   if (success)
26   {
27     for (token = strtok_r (fn_copy, " ", &save_ptr); token != NULL; token =
         strtok_r (NULL, " ", &save_ptr))
28     {
29       if_.esp -= (strlen(token) + 1);
30       memcpy (if_.esp, token, strlen(token) + 1);
31       argv[argc++] = (char *)if_.esp;
32     }
33     push_args (&if_.esp, argc, argv);
34     thread_current ()->parent->success = true;
35     sema_up(&thread_current ()->parent->sema);
36   }
37
38   /* If load failed, quit. */
```

```
39    palloc_free_page (file_name);
40    free (fn_copy);
41    if (!success)
42    {
43      thread_current()->parent->success = false;
44      sema_up(&thread_current ()->parent->sema);
45      thread_exit ();
46    }
47
48    /* Start the user process by simulating a return from an
49      interrupt, implemented by intr_exit (in
50      threads/intr-stubs.S).  Because intr_exit takes all of its
51      arguments on the stack in the form of a `struct intr_frame',
52      we just point the stack pointer (%esp) to our stack frame
53      and jump to it. */
54    asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) : "memory");
55    NOT_REACHED ();
56  }
```

到这里，参数传递的实现就完成了。我们可以尝试运行一个测试用例：

测试用例

```
1  $ pintos -v -k -T 60 --qemu  --filesys-size=2 -p build/tests/userprog/args-none
       -a args-none -- -q  -f run args-none
2  SeaBIOS (version 1.15.0-1)
3  Booting from Hard Disk...
4  PPiiLLoo  hhddaa1
5  1
6  LLooaaddiinngg.....................
7  Kernel command line: -q -f extract run 'args-single onearg'
8  Pintos booting with 3,968 kB RAM...
9  367 pages available in kernel pool.
10 367 pages available in user pool.
11 Calibrating timer...  556,236,800 loops/s.
12 ide0: unexpected interrupt
13 hda: 5,040 sectors (2 MB), model "QM00001", serial "QEMU HARDDISK"
14 hda1: 195 sectors (97 kB), Pintos OS kernel (20)
15 hda2: 4,096 sectors (2 MB), Pintos file system (21)
16 hda3: 117 sectors (58 kB), Pintos scratch (22)
17 ide1: unexpected interrupt
18 filesys: using hda2
19 scratch: using hda3
20 Formatting file system...done.
21 Boot complete.
```

```
Extracting ustar archive from scratch device into file system...
Putting 'args-single' into the file system...
Erasing ustar archive...
Executing 'args-single onearg':
system call!
Execution of 'args-single onearg' complete.
Timer: 71 ticks
Thread: 8 idle ticks, 63 kernel ticks, 0 user ticks
hda2 (filesys): 63 reads, 238 writes
hda3 (scratch): 116 reads, 2 writes
Console: 930 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
```

发现输出了 `system call!`，这是由于我们没有实现相关的系统调用。

2.2 系统调用

首先，我们需要在 `src/userprog/syscall.c` 建议一个指针数组 `syscall_table`，用于存放系统调用的函数指针。

src/userprog/syscall.c

```c
#include <syscall-nr.h>

#define MAX_SYSCALL (SYS_INUMBER)

typedef void (*syscall_func) (struct intr_frame *);

static void syscall_handler (struct intr_frame *);
static syscall_func syscall_table[MAX_SYSCALL + 1];
```

接着，我们定义相关的系统调用函数。

src/userprog/syscall.h

```c
void syscall_halt (struct intr_frame *f);
void syscall_exit (struct intr_frame *f);
void syscall_exec (struct intr_frame *f);
void syscall_wait (struct intr_frame *f);

void syscall_create (struct intr_frame *f);
void syscall_remove (struct intr_frame *f);
void syscall_open (struct intr_frame *f);
void syscall_filesize (struct intr_frame *f);
```

```
10 void syscall_read (struct intr_frame *f);
11 void syscall_write (struct intr_frame *f);
12 void syscall_seek (struct intr_frame *f);
13 void syscall_tell (struct intr_frame *f);
14 void syscall_close (struct intr_frame *f);
```

然后，我们需要在 `syscall_init()` 函数中初始化 `syscall_table`。

src/userprog/syscall.c - syscall_init()

```
1  void
2  syscall_init (void)
3  {
4    intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");
5    syscall_table[SYS_HALT] = syscall_halt;
6    syscall_table[SYS_EXIT] = syscall_exit;
7    syscall_table[SYS_EXEC] = syscall_exec;
8    syscall_table[SYS_WAIT] = syscall_wait;
9    syscall_table[SYS_CREATE] = syscall_create;
10   syscall_table[SYS_REMOVE] = syscall_remove;
11   syscall_table[SYS_OPEN] = syscall_open;
12   syscall_table[SYS_FILESIZE] = syscall_filesize;
13   syscall_table[SYS_READ] = syscall_read;
14   syscall_table[SYS_WRITE] = syscall_write;
15   syscall_table[SYS_SEEK] = syscall_seek;
16   syscall_table[SYS_TELL] = syscall_tell;
17   syscall_table[SYS_CLOSE] = syscall_close;
18 }
```

在进行系统调用时,我们需要访问用户空间的内存,因此,我们需要实现 `get_user()` 函数和 `check_ptr()` 函数，用于检验用户空间的内存是否合法。在内存不合法时，我们定义一个 `exit_error()` 函数，用于退出进程，并返回 `-1`。

src/userprog/syscall.c - get_user()

```
1 static int
2 get_user (const uint8_t *uaddr)
3 {
4   int result;
5   asm ("movl $1f, %0; movzbl %1, %0; 1:"
6        : "=&a" (result) : "m" (*uaddr));
7   return result;
8 }
```

src/userprog/syscall.c - check_ptr()

```
static void *
check_ptr (const void *vaddr)
{
  if (!is_user_vaddr (vaddr))
    exit_error ();
  void *ptr = pagedir_get_page (thread_current ()->pagedir, vaddr);
  if (!ptr)
    exit_error ();
  uint8_t *uaddr = (uint8_t *) vaddr;
  for (; uaddr < (uint8_t *) vaddr + sizeof (int); uaddr++)
    if (get_user (uaddr) == -1)
      exit_error ();
  return ptr;
}
```

src/userprog/syscall.c - exit_error()

```
static void
exit_error (void)
{
  thread_current ()->exit_status = -1;
  thread_exit ();
}
```

在这里，为了存储线程的退出状态，我们需要在线程结构体中添加 `int exit_status`。

src/threads/thread.h

```
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. */

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

```
17      struct semaphore sema;
18      struct thread* parent;
19      bool success;
20      int exit_status;
21  #endif
22
23      /* Owned by thread.c. */
24      unsigned magic;                     /* Detects stack overflow. */
25    };
```

接下来，修改 page_fault() 函数，使得其能够处理用户空间的内存错误。

src/userprog/exception.c - page_fault()

```
1  static void
2  page_fault (struct intr_frame *f)
3  {
4    bool not_present;  /* True: not-present page, false: writing r/o page. */
5    bool write;        /* True: access was write, false: access was read. */
6    bool user;         /* True: access by user, false: access by kernel. */
7    void *fault_addr;  /* Fault address. */
8
9    /* Obtain faulting address, the virtual address that was
10     accessed to cause the fault.  It may point to code or to
11     data.  It is not necessarily the address of the instruction
12     that caused the fault (that's f->eip).
13     See [IA32-v2a] "MOV--Move to/from Control Registers" and
14     [IA32-v3a] 5.15 "Interrupt 14--Page Fault Exception
15     (#PF)". */
16   asm ("movl %%cr2, %0" : "=r" (fault_addr));
17
18   /* Turn interrupts back on (they were only off so that we could
19     be assured of reading CR2 before it changed). */
20   intr_enable ();
21
22   /* Count page faults. */
23   page_fault_cnt++;
24
25   /* Determine cause. */
26   not_present = (f->error_code & PF_P) == 0;
27   write = (f->error_code & PF_W) != 0;
28   user = (f->error_code & PF_U) != 0;
29
30   if (!user)
31   {
```

```
32      f->eip = (void (*) (void)) f->eax;
33      f->eax = UINT32_MAX;
34      return;
35    }
36
37    /* To implement virtual memory, delete the rest of the function
38      body, and replace it with code that brings in the page to
39      which fault_addr refers. */
40    printf ("Page fault at %p: %s error %s page in %s context.\n",
41            fault_addr,
42            not_present ? "not present" : "rights violation",
43            write ? "writing" : "reading",
44            user ? "user" : "kernel");
45    kill (f);
46  }
```

然后，我们需要实现 `syscall_handler()` 函数，用于处理系统调用。

src/userprog/syscall.c - syscall_handler()

```
1  static void
2  syscall_handler (struct intr_frame *f UNUSED)
3  {
4    check_ptr ((int *)f->esp + 1);
5    int syscall_num = *(int *) f->esp;
6    if (syscall_num < 0 || syscall_num > MAX_SYSCALL)
7      exit_error ();
8    syscall_table[syscall_num] (f);
9  }
```

接下来，我们实现相关的系统调用函数。

2.2.1 halt()

`halt()` 函数用于关闭系统。

src/userprog/syscall.c - syscall_halt()

```
1  void
2  syscall_halt (struct intr_frame *f UNUSED)
3  {
4    shutdown_power_off ();
5  }
```

2.2.2 exit()

exit() 函数用于退出进程。

src/userprog/syscall.c - syscall_exit()

```
1 void
2 syscall_exit (struct intr_frame *f)
3 {
4   check_ptr ((uint32_t *)f->esp + 1);
5   int status = *(int *)((uint32_t *)f->esp + 1);
6   thread_current ()->exit_status = status;
7   thread_exit ();
8 }
```

2.2.3 exec()

exec() 函数用于执行程序。

src/userprog/syscall.c - syscall_exec()

```
1 void
2 syscall_exec (struct intr_frame *f)
3 {
4   check_ptr ((int *)f->esp + 1);
5   char *cmd_line = *(char **)(f->esp + 4);
6   check_ptr (cmd_line);
7   f->eax = process_execute (cmd_line);
8 }
```

2.2.4 wait()

wait() 函数用于等待子进程执行完毕。

src/userprog/syscall.c - syscall_wait()

```
1 void
2 syscall_wait (struct intr_frame *f)
3 {
4   check_ptr ((int *)f->esp + 1);
5   tid_t tid = *(tid_t *)(f->esp + 4);
6   f->eax = process_wait (tid);
7 }
```

然后，我们需要修改 src/userprog/process.c 中的 process_wait() 函数，使得其能够等待子进程执行完毕。

首先，我们需要修改线程结构体，添加 `struct list children`，用于存放子进程的线程结构体，以及定义子线程结构体 `struct child`，用于存放子进程的信息。同时，我们需要在线程结构体中添加 `struct child * child_thread`，用于记录当前线程的子线程结构体。

src/threads/thread.h

```
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. */

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
    struct semaphore sema;
    struct thread* parent;
    bool success;
    int exit_status;
    struct list children;
    struct clild *child_thread;
#endif

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

#ifdef USERPROG
struct child
{
  tid_t tid;
  int exit_status;
  bool running;
  struct semaphore sema;
  struct list_elem elem;
};
#endif
```

接下来，修改 `create_thread()` 函数，和 `init_thread()` 函数，使得其能够初始化这些量。

src/threads/thread.c - create_thread()

```
tid_t
thread_create (const char *name, int priority,
               thread_func *function, void *aux)
{
  struct thread *t;
  struct kernel_thread_frame *kf;
  struct switch_entry_frame *ef;
  struct switch_threads_frame *sf;
  tid_t tid;

  ASSERT (function != NULL);

  /* Allocate thread. */
  t = palloc_get_page (PAL_ZERO);
  if (t == NULL)
    return TID_ERROR;

  /* Initialize thread. */
  init_thread (t, name, priority);
  tid = t->tid = allocate_tid ();
#ifdef USERPROG
  struct child *c = malloc (sizeof (struct child));
  c->tid = tid;
  sema_init (&c->sema, 0);
  list_push_back (&thread_current ()->children, &c->elem);
  c->exit_status = UINT32_MAX;
  c->running = false;
  t->child_thread = c;
#endif

  /* Stack frame for kernel_thread(). */
  kf = alloc_frame (t, sizeof *kf);
  kf->eip = NULL;
  kf->function = function;
  kf->aux = aux;

  /* Stack frame for switch_entry(). */
  ef = alloc_frame (t, sizeof *ef);
  ef->eip = (void (*) (void)) kernel_thread;

  /* Stack frame for switch_threads(). */
```

```
42   sf = alloc_frame (t, sizeof *sf);
43   sf->eip = switch_entry;
44   sf->ebp = 0;
45
46   /* Add to run queue. */
47   thread_unblock (t);
48
49   return tid;
50 }
```

src/threads/thread.c - init_thread()

```
1  static void
2  init_thread (struct thread *t, const char *name, int priority)
3  {
4    enum intr_level old_level;
5
6    ASSERT (t != NULL);
7    ASSERT (PRI_MIN <= priority && priority <= PRI_MAX);
8    ASSERT (name != NULL);
9
10   memset (t, 0, sizeof *t);
11   t->status = THREAD_BLOCKED;
12   strlcpy (t->name, name, sizeof t->name);
13   t->stack = (uint8_t *) t + PGSIZE;
14   t->priority = priority;
15   t->magic = THREAD_MAGIC;
16
17 #ifdef USERPROG
18   if (t == initial_thread)
19     t->parent = NULL;
20   else
21     t->parent = thread_current ();
22   list_init (&t->children);
23   sema_init (&t->sema, 0);
24   t->success = true;
25   t->exit_status = UINT32_MAX;
26 #endif
27
28   old_level = intr_disable ();
29   list_push_back (&all_list, &t->allelem);
30   intr_set_level (old_level);
31 }
```

然后，我们需要修改 process_wait() 函数。

src/userprog/process.c - process_wait()

```
int
process_wait (tid_t child_tid UNUSED)
{
  struct list *l = &thread_current ()->children;
  struct list_elem *child_elem_ptr;
  child_elem_ptr = list_begin (l);
  struct child *child_ptr = NULL;
  while (child_elem_ptr != list_end (l))
  {
    child_ptr = list_entry (child_elem_ptr, struct child, elem);
    if (child_ptr->tid == child_tid)
    {
      if (!child_ptr->running)
      {
        child_ptr->running = true;
        sema_down (&child_ptr->sema);
        break;
      }
      else
        return -1;
    }
    child_elem_ptr = list_next (child_elem_ptr);
  }
  if (child_elem_ptr == list_end (l))
    return -1;
  list_remove (child_elem_ptr);
  return child_ptr->exit_status;
}
```

在这个函数中，我们首先遍历当前线程的子线程，找到对应的子线程，然后判断子线程是否已经执行完毕，如果已经执行完毕，则返回子线程的退出状态，否则，返回 -1。

最后，还需要修改 thread_exit() 函数，使得其能够唤醒父进程。

2.2.5 write()

write() 函数用于向文件中写入数据。

为了实现文件操作，我们需要在线程结构体中添加 struct list files，用于存放文件的结构体和 int max_fd，用于存储最大的文件标识符，以及定义文件结构体 struct struct_file 来存储文件信息。

src/threads/thread.h

```
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. */

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
    struct semaphore sema;
    struct thread* parent;
    bool success;
    int exit_status;
    struct list children;
    struct clild *child_thread;
    struct list files;
    int max_fd;
#endif

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

#ifdef USERPROG
struct thread_file
{
  struct file *file;
  int fd;
  struct list_elem elem;
};
#endif
```

然后，我们定义获取文件的锁和释放文件的锁的操作。

src/threads/thread.c

```
#ifdef USERPROG
static struct lock lock_f;
```

```
3  static void acquire_f (void);
4  static void release_f (void);
5
6  static void
7  acquire_f (void)
8  {
9    lock_acquire (&lock_f);
10 }
11
12 static void
13 release_f (void)
14 {
15   lock_release (&lock_f);
16 }
17 #endif
```

然后，在 thread_init() 和 init_thread() 函数中初始化这些量。

src/threads/thread.c - thread_init()

```
1  void
2  thread_init (void)
3  {
4    ASSERT (intr_get_level () == INTR_OFF);
5
6    lock_init (&tid_lock);
7    list_init (&ready_list);
8    list_init (&all_list);
9
10 #ifdef USERPROG
11   lock_init (&lock_f);
12 #endif
13
14   /* Set up a thread structure for the running thread. */
15   initial_thread = running_thread ();
16   init_thread (initial_thread, "main", PRI_DEFAULT);
17   initial_thread->status = THREAD_RUNNING;
18   initial_thread->tid = allocate_tid ();
19 }
```

src/threads/thread.c - init_thread()

```
1  static void
2  init_thread (struct thread *t, const char *name, int priority)
3  {
```

```
4   enum intr_level old_level;
5
6   ASSERT (t != NULL);
7   ASSERT (PRI_MIN <= priority && priority <= PRI_MAX);
8   ASSERT (name != NULL);
9
10  memset (t, 0, sizeof *t);
11  t->status = THREAD_BLOCKED;
12  strlcpy (t->name, name, sizeof t->name);
13  t->stack = (uint8_t *) t + PGSIZE;
14  t->priority = priority;
15  t->magic = THREAD_MAGIC;
16
17 #ifdef USERPROG
18  if (t == initial_thread)
19    t->parent = NULL;
20  else
21    t->parent = thread_current ();
22  list_init (&t->children);
23  list_init (&t->files);
24  sema_init (&t->sema, 0);
25  t->success = true;
26  t->exit_status = UINT32_MAX;
27  t->max_fd = 2;
28 #endif
29
30  old_level = intr_disable ();
31  list_push_back (&all_list, &t->allelem);
32  intr_set_level (old_level);
33 }
```

然后，在 thread_exit() 函数中，将所有文件释放。

src/threads/thread.c - thread_exit()

```
1  void
2  thread_exit (void)
3  {
4    ASSERT (!intr_context ());
5
6  #ifdef USERPROG
7    process_exit ();
8  #endif
9
10   /* Remove thread from all threads list, set our status to dying,
```

```
11      and schedule another process.  That process will destroy us
12      when it calls thread_schedule_tail(). */
13    intr_disable ();
14
15  #ifdef USERPROG
16    thread_current ()->child_thread->exit_status = thread_current ()->exit_status;
17    sema_up (&thread_current ()->child_thread->sema);
18
19    struct list_elem *e;
20    struct list *files = &thread_current()->files;
21    while(!list_empty (files))
22    {
23      e = list_pop_front (files);
24      struct thread_file *f = list_entry (e, struct thread_file, elem);
25      acquire_f ();
26      file_close (f->file);
27      release_f ();
28      list_remove (e);
29      free (f);
30    }
31  #endif
32
33    list_remove (&thread_current()->allelem);
34    thread_current ()->status = THREAD_DYING;
35    schedule ();
36    NOT_REACHED ();
37  }
```

然后，我们还需要定义一个 `find_file()` 函数，用来通过文件标识符找到文件。

src/userprog/syscall.c - find_file()

```
1  static struct thread_file *
2  find_file (int fd)
3  {
4    struct list_elem *e;
5    struct thread_file * thread_file_temp = NULL;
6    struct list *files = &thread_current ()->files;
7    for (e = list_begin (files); e != list_end (files); e = list_next (e))
8    {
9      thread_file_temp = list_entry (e, struct thread_file, elem);
10     if (fd == thread_file_temp->fd)
11       return thread_file_temp;
12   }
13   return false;
```

```
14 }
```

由于 load() 函数中，我们也对文件进行了操作，因此，我们需要在 load() 函数中添加获取文件的锁和释放文件的锁的操作。

src/userprog/process.c - load()

```
1  bool
2  load (const char *file_name, void (**eip) (void), void **esp)
3  {
4    struct thread *t = thread_current ();
5    struct Elf32_Ehdr ehdr;
6    struct file *file = NULL;
7    off_t file_ofs;
8    bool success = false;
9    int i;
10
11   /* Allocate and activate page directory. */
12   t->pagedir = pagedir_create ();
13   if (t->pagedir == NULL)
14     goto done;
15   process_activate ();
16
17   /* Open executable file. */
18   acquire_f ();
19   file = filesys_open (file_name);
20   if (file == NULL)
21     {
22       printf ("load: %s: open failed\n", file_name);
23       goto done;
24     }
25
26   struct thread_file *thread_file_temp = malloc(sizeof(struct thread_file));
27   thread_file_temp->file = file;
28   list_push_back (&thread_current()->files, &thread_file_temp->elem);
29   file_deny_write (file);
30
31   /* Read and verify executable header. */
32   if (file_read (file, &ehdr, sizeof ehdr) != sizeof ehdr
33       || memcmp (ehdr.e_ident, "\177ELF\1\1\1", 7)
34       || ehdr.e_type != 2
35       || ehdr.e_machine != 3
36       || ehdr.e_version != 1
37       || ehdr.e_phentsize != sizeof (struct Elf32_Phdr)
38       || ehdr.e_phnum > 1024)
```

```
39      {
40        printf ("load: %s: error loading executable\n", file_name);
41        goto done;
42      }
43
44    /* Read program headers. */
45    file_ofs = ehdr.e_phoff;
46    for (i = 0; i < ehdr.e_phnum; i++)
47      {
48        struct Elf32_Phdr phdr;
49
50        if (file_ofs < 0 || file_ofs > file_length (file))
51          goto done;
52        file_seek (file, file_ofs);
53
54        if (file_read (file, &phdr, sizeof phdr) != sizeof phdr)
55          goto done;
56        file_ofs += sizeof phdr;
57        switch (phdr.p_type)
58          {
59          case PT_NULL:
60          case PT_NOTE:
61          case PT_PHDR:
62          case PT_STACK:
63          default:
64            /* Ignore this segment. */
65            break;
66          case PT_DYNAMIC:
67          case PT_INTERP:
68          case PT_SHLIB:
69            goto done;
70          case PT_LOAD:
71            if (validate_segment (&phdr, file))
72              {
73                bool writable = (phdr.p_flags & PF_W) != 0;
74                uint32_t file_page = phdr.p_offset & ~PGMASK;
75                uint32_t mem_page = phdr.p_vaddr & ~PGMASK;
76                uint32_t page_offset = phdr.p_vaddr & PGMASK;
77                uint32_t read_bytes, zero_bytes;
78                if (phdr.p_filesz > 0)
79                  {
80                    /* Normal segment.
81                      Read initial part from disk and zero the rest. */
82                    read_bytes = page_offset + phdr.p_filesz;
```

```
83                    zero_bytes = (ROUND_UP (page_offset + phdr.p_memsz, PGSIZE)
84                                  - read_bytes);
85                }
86              else
87                {
88                  /* Entirely zero.
89                    Don't read anything from disk. */
90                  read_bytes = 0;
91                  zero_bytes = ROUND_UP (page_offset + phdr.p_memsz, PGSIZE);
92                }
93              if (!load_segment (file, file_page, (void *) mem_page,
94                                 read_bytes, zero_bytes, writable))
95                goto done;
96            }
97          else
98            goto done;
99          break;
100       }
101   }
102
103 /* Set up stack. */
104 if (!setup_stack (esp))
105   goto done;
106
107 /* Start address. */
108 *eip = (void (*) (void)) ehdr.e_entry;
109
110 success = true;
111
112 done:
113 /* We arrive here whether the load is successful or not. */
114 release_f ();
115 return success;
116 }
```

接下来，我们修可以编写 `syscall_write()` 函数，使得其能够向文件中写入数据。

src/userprog/syscall.c - syscall_write()

```
1  void
2  syscall_write (struct intr_frame *f)
3  {
4    check_ptr ((uint32_t *)f->esp + 7);
5    check_ptr ((void *)*((uint32_t *)f->esp + 6));
6    int fd = *((uint32_t *) f->esp + 1);
```

```
7   const char *buffer = (const char *)*((uint32_t *)f->esp + 2);
8   off_t size = *((uint32_t *)f->esp + 3);
9   if (fd == 1)
10  {
11    putbuf (buffer,size);
12    f->eax = size;
13  }
14  else
15  {
16    struct thread_file * thread_file_temp = find_file (fd);
17    if (thread_file_temp)
18    {
19      acquire_f ();
20      f->eax = file_write (thread_file_temp->file, buffer, size);
21      release_f ();
22    }
23    else
24      f->eax = 0;
25  }
26 }
```

2.2.6 create()

create() 函数用于创建文件。

src/userprog/syscall.c - syscall_create()

```
1  void
2  syscall_create (struct intr_frame *f)
3  {
4    check_ptr((uint32_t *)f->esp + 5);
5    check_ptr((void *)*((uint32_t *)f->esp + 4));
6    const char *file = (const char *)*((uint32_t *)f->esp + 1);
7    int initial_size = *((uint32_t *)f->esp + 2);
8    acquire_f ();
9    f->eax = filesys_create (file, initial_size);
10   release_f ();
11 }
```

2.2.7 remove()

remove() 函数用于删除文件。

src/userprog/syscall.c - syscall_remove()

```
1 void
2 syscall_remove (struct intr_frame *f)
3 {
4   check_ptr ((uint32_t *)f->esp + 1);
5   check_ptr ((void *)*((uint32_t *)f->esp + 1));
6   acquire_f ();
7   f->eax = filesys_remove ((const char *)*((uint32_t *)f->esp + 1));
8   release_f ();
9 }
```

2.2.8 open()

open() 函数用于打开文件。

src/userprog/syscall.c - syscall_open()

```
1  void
2  void
3  syscall_open (struct intr_frame *f)
4  {
5    check_ptr ((uint32_t *)f->esp + 1);
6    check_ptr ((void *)*((uint32_t *)f->esp + 1));
7    acquire_f ();
8    struct file * file_opened = filesys_open((const char *)*((uint32_t *)f->esp +
         1));
9    release_f ();
10   struct thread * t = thread_current();
11   if (file_opened)
12   {
13     struct thread_file *thread_file_temp = malloc(sizeof(struct thread_file));
14     thread_file_temp->fd = t->max_fd++;
15     thread_file_temp->file = file_opened;
16     list_push_back (&t->files, &thread_file_temp->elem);
17     f->eax = thread_file_temp->fd;
18   }
19   else
20     f->eax = -1;
21 }
```

2.2.9 filesize()

filesize() 函数用于获取文件的大小。

src/userprog/syscall.c - syscall_filesize()

```
1  void
2  syscall_filesize (struct intr_frame *f)
3  {
4    check_ptr ((uint32_t *)f->esp + 1);
5    struct thread_file * thread_file_temp = find_file (*((uint32_t *)f->esp + 1));
6    if (thread_file_temp)
7    {
8      acquire_f ();
9      f->eax = file_length (thread_file_temp->file);
10     release_f ();
11   }
12   else
13     f->eax = -1;
14 }
```

2.2.10 read()

read() 函数用于从文件中读取数据。

src/userprog/syscall.c - syscall_read()

```
1  void
2  syscall_read (struct intr_frame *f)
3  {
4    int fd = *((uint32_t *)f->esp + 1);
5    uint8_t * buffer = (uint8_t*)*((uint32_t *)f->esp + 2);
6    off_t size = *((uint32_t *)f->esp + 3);
7    if (!is_valid_p (buffer, 1) || !is_valid_p (buffer + size, 1)){
8      exit_error ();
9    }
10   if (fd == 0)
11   {
12     for (int i = 0; i < size; i++)
13       buffer[i] = input_getc();
14     f->eax = size;
15   }
16   else
17   {
18     struct thread_file * thread_file_temp = find_file (fd);
19     if (thread_file_temp)
20     {
21       acquire_f ();
22       f->eax = file_read (thread_file_temp->file, buffer, size);
```

```
23        release_f ();
24      }
25      else
26        f->eax = -1;
27    }
28  }
```

2.2.11 seek()

seek() 函数用于设置文件的偏移量。

src/userprog/syscall.c - syscall_seek()

```
1  void
2  syscall_seek (struct intr_frame *f)
3  {
4    check_ptr ((uint32_t *)f->esp + 5);
5    struct thread_file *file_temp = find_file (*((uint32_t *)f->esp + 1));
6    if (file_temp)
7    {
8      acquire_f ();
9      file_seek (file_temp->file, *((uint32_t *)f->esp + 2));
10     release_f ();
11   }
12 }
```

2.2.12 tell()

tell() 函数用于获取文件的偏移量。

src/userprog/syscall.c - syscall_tell()

```
1  void
2  syscall_tell (struct intr_frame *f)
3  {
4    check_ptr ((uint32_t *)f->esp + 1);
5    struct thread_file *thread_file_temp = find_file (*((uint32_t *)f->esp + 1));
6    if (thread_file_temp)
7    {
8      acquire_f ();
9      f->eax = file_tell (thread_file_temp->file);
10     release_f ();
11   }
12   else
13     f->eax = -1;
```

```
14 }
```

2.2.13 close()

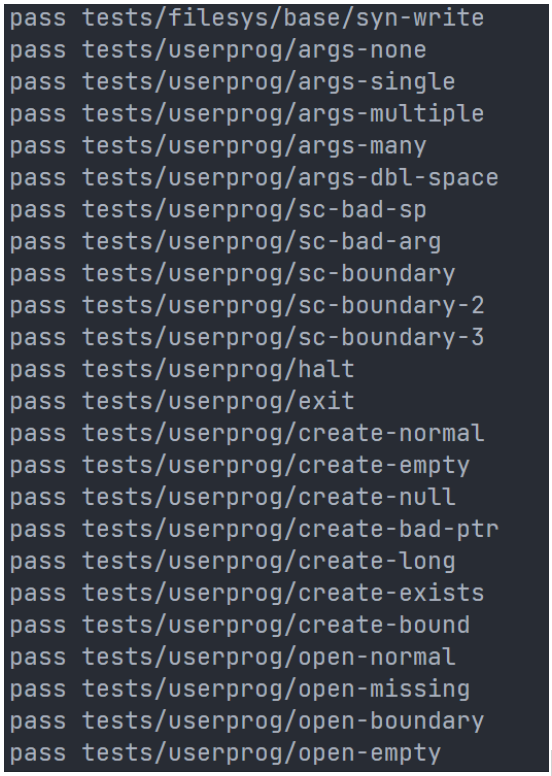
close() 函数用于关闭文件。

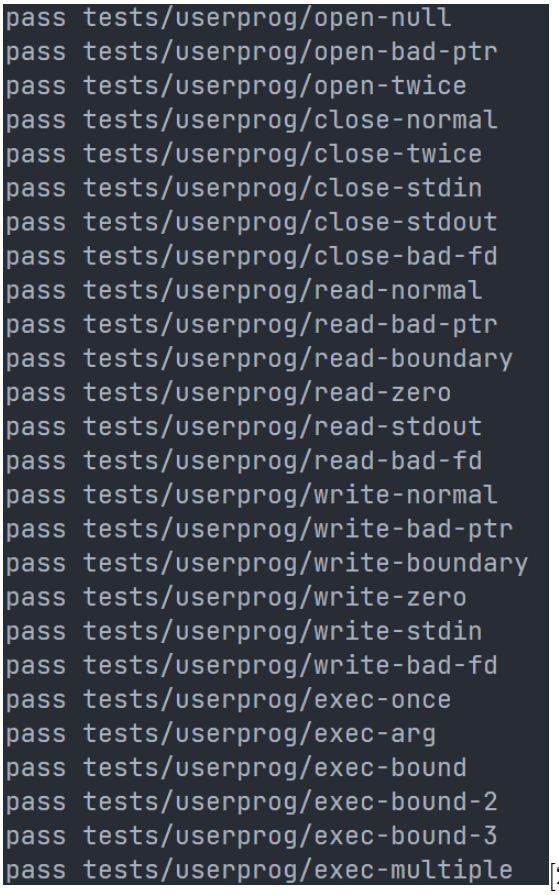
src/userprog/syscall.c - syscall_close()

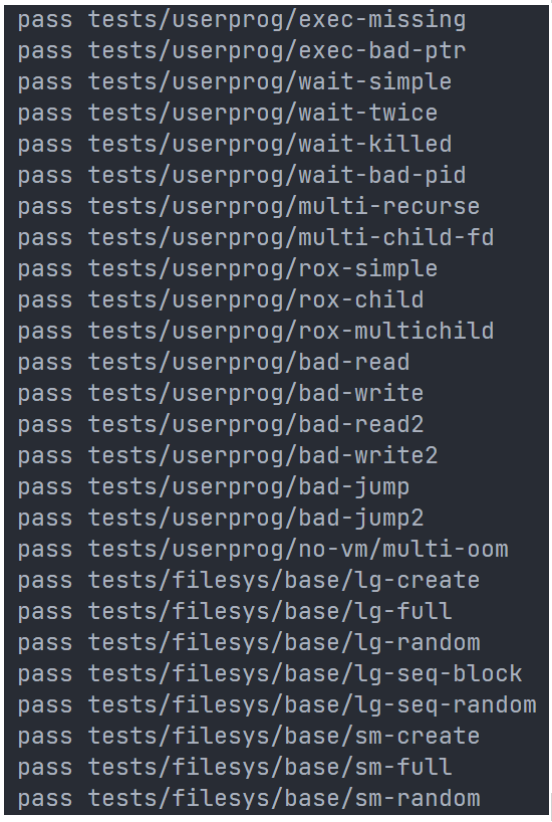
```
1  void
2  syscall_close (struct intr_frame *f)
3  {
4    check_ptr ((uint32_t *)f->esp + 1);
5    struct thread_file * opened_file = find_file (*((uint32_t *)f->esp + 1));
6    if (opened_file)
7    {
8      acquire_f ();
9      file_close (opened_file->file);
10     release_f ();
11     list_remove (&opened_file->elem);
12     free (opened_file);
13   }
14 }
```

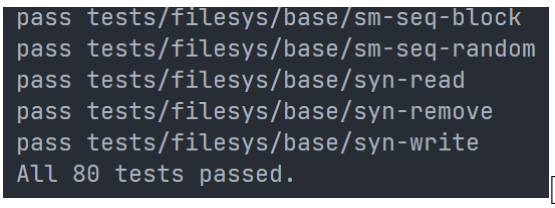
3 实验过程与分析

实验的测试截图如下：



[1]


[2]


[3]


[4]

图 2: 实验的测试截图

具体实验分析可以参考上文。

4 实验结果总结

在本次实验中，我完成了参数传递和 13 个系统调用的实现，对操作系统进程管理、内存管理、文件系统管理有了更深入的了解。

5 附录（源代码）

见 `source-code-pintos-anon.tar.gz` 或 `source-code-pintos-anon.zip`。
也可以访问仓库 `https://github.com/llipengda/pintos-anon/tree/pdli-project2`。