实验六——系统调用

课程名称:	操作系统实践	指导教师:	张民
姓　　名:	王海生	学　　号:	10235101559
实验编号:	实验六	实验名称:	系统调用

代码仓库：https://github.com/Hanson-Wang-chn/ECNU-Operating-System-WHS.git

目录

1 实验目的

1. 完成系统调用，使得 `make check` 通过 `create`、`open`、`close`、`read` 相关测试。

2. 完成实验报告并提交。无需提交代码文件，只需要提交报告的 pdf 即可。在报告中介绍对系统调用的分析过程，展示代码中需要修改的地方，解释修改的含义，和运行结果。

3. 可选扩展任务（不强求）：继续完成其他系统调用，通过 make check 中其他的测试点。相关内容也写在实验报告中。

2 实验内容与设计思想

本次实验的主要任务是实现 Pintos 操作系统中的系统调用功能。系统调用是用户程序与操作系统内核之间的接口，用户程序通过系统调用请求内核提供服务，如文件操作、进程控制等。实验的核心目标是实现一系列系统调用，并通过 Pintos 提供的测试用例验证其正确性。

实验内容主要包括以下几个方面：

1. **系统调用的基本框架**：首先需要理解 Pintos 中系统调用的处理流程，包括如何从用户态切换到内核态，如何传递参数，以及如何返回结果。通过修改 syscall.c 文件，实现系统调用的基本框架。

2. **进程相关的系统调用**：实现 halt、exit、exec、wait 等进程控制系统调用。这些系统调用涉及到进程的创建、终止、等待等操作，需要仔细处理进程间的同步问题，确保父子进程的正确通信。

3. **文件相关的系统调用**：实现 create、remove、open、close、read、write 等文件操作系统调用。这些系统调用涉及到文件的创建、删除、打开、关闭、读写等操作，需要注意文件系统的线程安全性，确保多个进程对文件的并发访问不会导致数据不一致。

4. **内存访问检查**：在系统调用中，用户程序可能会传递非法地址，因此需要实现对用户内存地址的检查，确保用户程序只能访问其合法的内存空间。通过 check_ptr 函数对用户传递的指针进行检查，防止非法访问。

5. **扩展任务**：在完成基本任务后，还可以进一步实现 seek、tell 等文件操作系统调用，并通过 make check 中的更多测试用例，验证系统的完整性和稳定性。

在实验过程中，我逐步理解了系统调用的实现机制，并通过调试和测试，解决了进程同步、文件系统线程安全等问题。最终，**所有的测试用例均通过**。

3 使用环境

3.1 主机系统配置

本次实验的主机系统环境如下表所示：

项目名称	详细信息
操作系统	macOS Sequoia 15.2
系统类型	64 位操作系统，基于 ARM 的处理器
CPU	Apple M1 Pro
GPU	Apple M1 Pro
内存	32GB 统一内存
磁盘	512GB SSD

3.2 Docker 配置

在官网下载并安装后，Docker 容器正常运行，如下图所示：

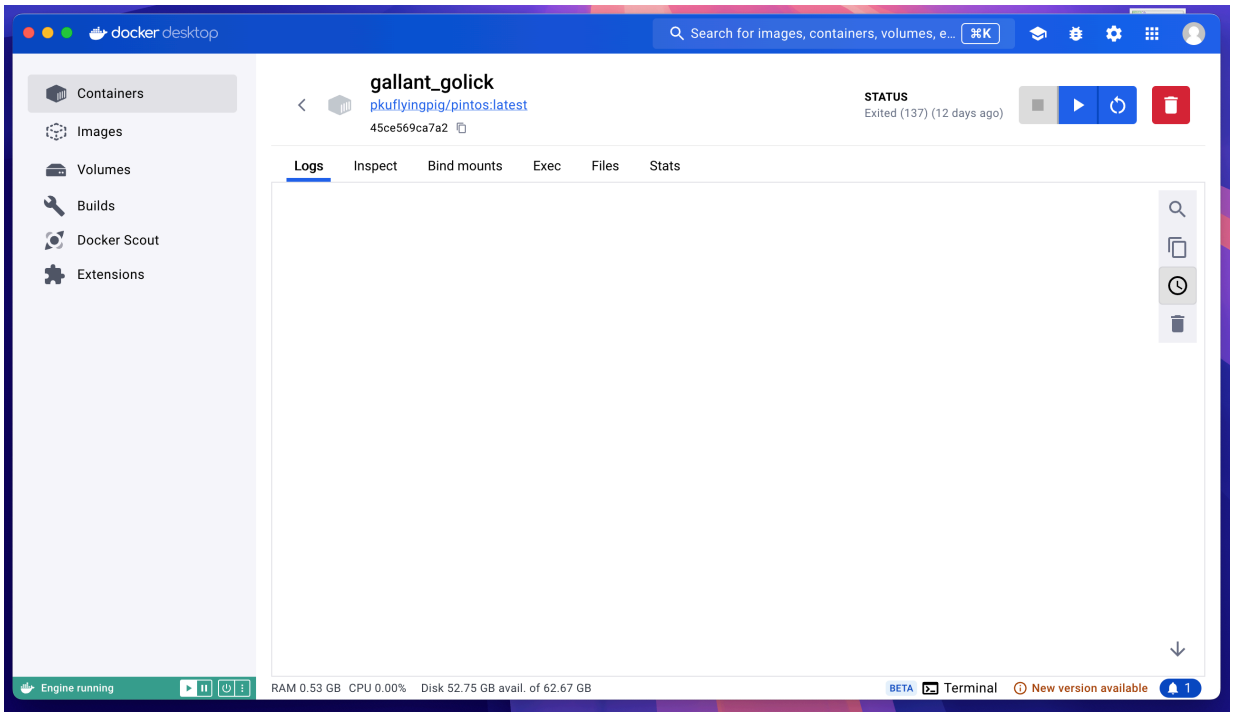

图 1: Docker 容器

接着使用下面的命令实现磁盘挂载，方便文件管理：

启动 Docker 容器并挂载文件

```
1    docker run -it --rm --name pintos --mount type=bind,source=/Users/
       wanghaisheng/Desktop/Coding/Courses/ECNU-Operating-System-WHS/pintos,
       target=/home/PKUOS/pintos pkuflyingpig/pintos bash
```

完成后如下图所示：

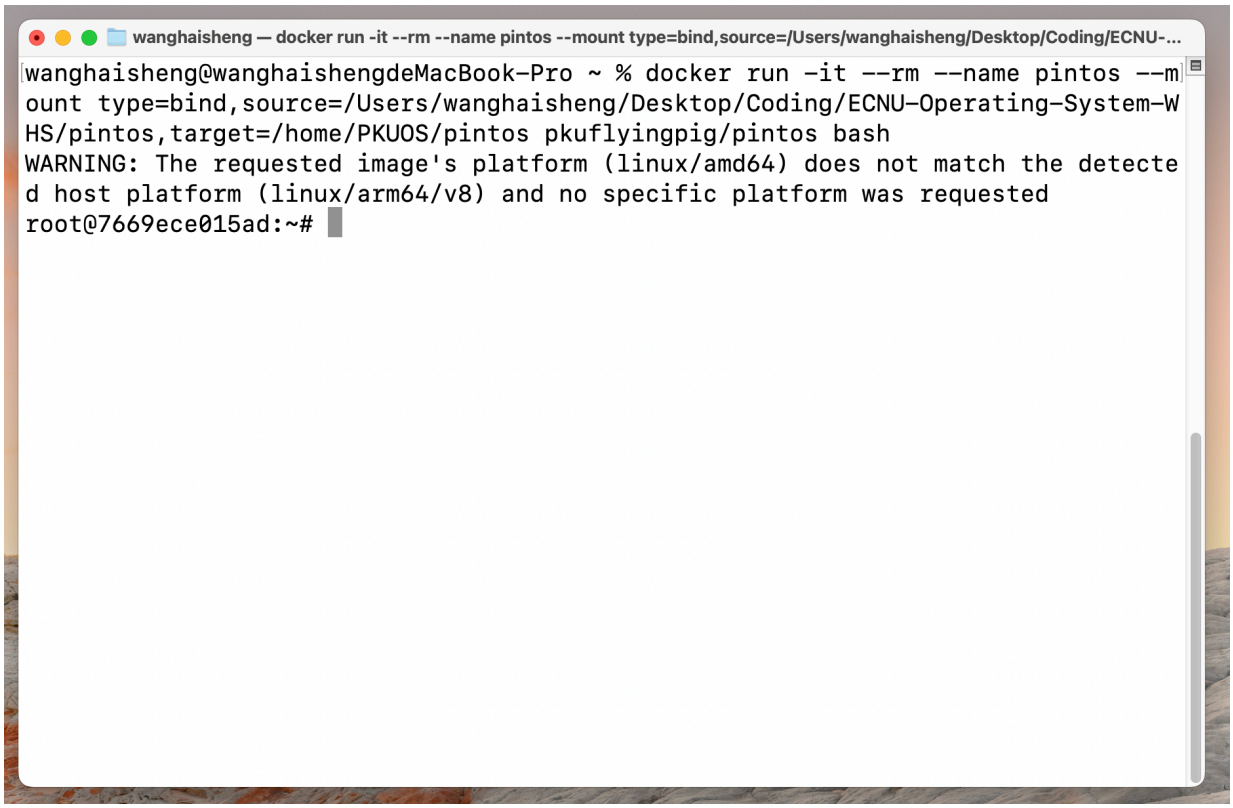

图 2: 完成环境配置

4 Accessing User Memory

在用户使用系统调用时，可能会出现访问的地址超出用户可以访问的地址范围的情况，因此需要检查用户地址是否合法。对于整数类型，需要检查四个字节；对于字符串，则需要逐个字节进行检查。

根据文档说明，阅读 `userprog/pagedir.c` 和 `threads/vaddr.h` 文件可知，对于一个进程，用户可以访问的内存地址应在用户可访问的合理地址范围内，并且该地址必须已经映射为物理地址，即在页表中能找到对应的页。只有在满足这两个条件后，才能确认该地址是合法的。

```
1    void *check_ptr(void *ptr, int byte)
2    {
3            int pd = thread_current()->pagedir;
4            for (int i = 0; i < byte; i++)
5            if (!is_user_vaddr(ptr + i) || pagedir_get_page(pd, ptr + i) == NULL
                 )
6            error_exit();
7            return ptr;
8    }
```

在之前的系统调用中加入指针检查后，make_check 能够通过所有以 sc 开头的测试点。

```
1   void syscall_exit(struct intr_frame *f)
2   {
3           int exit_code = *(int *)check_ptr(f->esp + 4, 4);
4           thread_current()->exit_code = exit_code;
5           thread_exit();
6   }
7   void syscall_wait(struct intr_frame *f)
8   {
9           int pid = *(int *)check_ptr(f->esp + 4, 4);
10          f->eax = process_wait(pid);
11  }
12  void syscall_write(struct intr_frame *f)
13  {
14          int fd = *(int *)check_ptr(f->esp + 4, 4);
15          char *buf = *(char **)check_ptr(f->esp + 8, 4);
16          check_str(buf);
17          int size = *(int *)check_ptr(f->esp + 12, 4);
18          if (fd == 0)
19          error_exit();
20          if (fd == 1)
21          {
22                  putbuf(buf, size);
23                  f->eax = size;
24          }
25  }
```

```
1   pass tests/userprog/sc-bad-sp
2   pass tests/userprog/sc-bad-arg
3   pass tests/userprog/sc-boundary
4   pass tests/userprog/sc-boundary-2
5   pass tests/userprog/sc-boundary-3
```

5 Process System Call

5.1 halt & exit

根据文档，halt 直接调用了 shutdown_power_off 函数。需要注意的是，每个 exit_info 的生命周期都比对应的进程生命周期长，因此与 exit 相关的错误码可以仅保存在每个进程的 exit_info 中，从而避免冗余存储。请确保将所有 thread_current()->exit_code 替换为 thread_current()->linked_exit->exit_code，同时在 process_exit 中不再需要传递错误码。

```
1   void syscall_halt(struct intr_frame *f)
2   {
3           shutdown_power_off();
4   }
5
6   void syscall_exit(struct intr_frame *f)
7   {
8           int exit_code = *(int *)check_ptr(f->esp + 4, 4);
9           thread_current()->linked_exit->exit_code = exit_code;
10          thread_exit();
11  }
```

5.2 wait & exec

wait 函数在之前的实验中已经实现。根据 exec 函数的定义 pid_t exec (const char *cmd_line)，该函数接收一个命令行字符串作为参数，并返回子进程的 id。如果子进程创建失败，则返回 -1。父进程需要等待子进程创建完成。

```
1   void syscall_exec(struct intr_frame *f)
2   {
3           char *cmd = *(char **)check_ptr(f->esp + 4, 4);
4           check_str(cmd);
5           f->eax = process_execute(cmd);
6   }
```

为实现同步，需要再定义一个 sema_exec，同时在 init_thread 中进行初始化。然而，子进程无法通知父进程是否创建成功，因此使用一个 exec_success 字段，让子进程修改以指示创建是否成功。

```
1   struct thread
2   {
3           /* Owned by thread.c. */
4           tid_t tid;    /** Thread identifier. */
5           enum thread_status status;    /** Thread state. */
6           char name[16];    /** Name (for debugging purposes). */
7           uint8_t *stack;    /**< Saved stack pointer. */
8           int priority;    /**< Priority. */
9           struct list_elem allelem;    /**< List element for all threads list.
                */
10          struct list_elem waitelem;
11          int64_t wake_tick;
12          /* Shared between thread.c and synch.c. */
13          struct list_elem elem;    /**< List element. */
14          #ifdef USERPROG
```

```
15              /* Owned by userprog/process.c. */
16              uint32_t *pagedir;    /**< Page directory. */
17              struct list child_list;
18              struct exit_info *linked_exit;
19              struct semaphore sema_wait;
20              struct semaphore sema_exec;
21              bool exec_success;
22              #endif
23              /* Owned by thread.c. */
24              unsigned magic;    /**< Detects stack overflow. */
25      };
26
27      static void
28      init_thread (struct thread *t, const char *name, int priority)
29      {
30              enum intr_level old_level;
31              ASSERT (t != NULL);
32              ASSERT (PRI_MIN <= priority && priority <= PRI_MAX);
33              ASSERT (name != NULL);
34              memset (t, 0, sizeof *t);
35              t->status = THREAD_BLOCKED;
36              strlcpy (t->name, name, sizeof t->name);
37              t->stack = (uint8_t *) t + PGSIZE;
38              t->priority = priority;
39              t->magic = THREAD_MAGIC;
40              t->wake_tick = -1;
41              list_init(&t->child_list);
42              sema_init(&t->sema_wait, 0);
43              sema_init(&t->sema_exec, 0);
44              old_level = intr_disable ();
45              list_insert_ordered(&all_list, &t->allelem, thread_more_priority,
                    NULL);
46              intr_set_level (old_level);
47      }
```

在父进程的 process_execute 中，调用 thread_create 后，需要进行 P 操作以等待子进程完成创建。随后，父进程获取子进程的状态并返回。在子进程的 start_process 中，管理好 file_name 和 cmd 的生命周期后，填写父进程的 exec_success 字段，接着进行 V 操作并退出。

```
1       tid_t
2       process_execute (const char *file_name)
3       {
4               char *fn_copy, *name_copy;
5               tid_t tid;
```

```
6
7               fn_copy = palloc_get_page(0);
8               name_copy = palloc_get_page(0);
9               if (fn_copy == NULL || name_copy == NULL)
10              return TID_ERROR;
11
12              strlcpy (fn_copy, file_name, PGSIZE);
13              strlcpy (name_copy, file_name, PGSIZE);
14
15              char *save_ptr;
16              name_copy = strtok_r(name_copy, "␣", &save_ptr);
17
18              tid = thread_create (name_copy, PRI_DEFAULT, start_process, fn_copy)
                    ;
19
20              palloc_free_page(name_copy);
21
22              if (tid == TID_ERROR)
23              palloc_free_page (fn_copy);
24
25              struct thread* cur = thread_current();
26              sema_down(&cur->sema_exec);
27              if (!cur->exec_success) return TID_ERROR;
28
29              return tid;
30      }
31
32      static void
33      start_process (void *file_name_)
34      {
35              char *file_name = file_name_;
36              struct intr_frame if_;
37              bool success;
38
39              /* Initialize interrupt frame and load executable. */
40              memset (&if_, 0, sizeof if_);
41              if_.gs = if_.fs = if_.es = if_.ds = if_.ss = SEL_UDSEG;
42              if_.cs = SEL_UCSEG;
43              if_.eflags = FLAG_IF || FLAG_MBS;
44
45              struct thread *cur = thread_current();
46
47              char *cmd = palloc_get_page(0);
48              if (cmd != NULL)
```

```
49              {
50                      strlcpy(cmd, file_name_, PGSIZE);
51
52                      char *save_ptr;
53                      file_name = strtok_r(file_name, "␣", &save_ptr);
54                      success = load (file_name, &if_.eip, &if_.esp);
55
56                      if (success)
57                      {
58                              push_argument(&if_.esp, cmd);
59                              palloc_free_page(cmd);
60                              palloc_free_page(file_name);
61
62                              cur->linked_exit->parent->exec_success = true;
63                              sema_up(&cur->linked_exit->parent->sema_exec);
64                              asm volatile ("movl␣%0,␣%%esp;␣jmp␣intr_exit" : : "g
                                    " (&if_) : "memory");
65                              NOT_REACHED ();
66                      }
67                      palloc_free_page(cmd);
68              }
69
70              palloc_free_page(file_name);
71              cur->linked_exit->parent->exec_success = false;
72              sema_up(&cur->linked_exit->parent->sema_exec);
73              error_exit();
74      }
```

至此可以通过 exec 和 wait 相关的测试点。

```
1       pass tests/userprog/exec-once
2       pass tests/userprog/exec-arg
3       pass tests/userprog/exec-bound
4       pass tests/userprog/exec-bound-2
5       pass tests/userprog/exec-bound-3
6       pass tests/userprog/exec-multiple
7       pass tests/userprog/exec-missing
8       pass tests/userprog/exec-bad-ptr
9       pass tests/userprog/wait-simple
10      pass tests/userprog/wait-twice
11      pass tests/userprog/wait-killed
12      pass tests/userprog/wait-bad-pid
```

6 File System Call

根据文档描述，当前的文件系统是非线程安全的。因此，需要引入一个全局的文件锁来确保文件系统的互斥访问。具体实现如下：在 `thread.h` 中定义 `struct lock filesys_lock`，并在 `thread_init` 函数中通过 `lock_init(&filesys_lock)` 初始化该全局锁。在完善系统调用后，即可完成与 `create` 相关的测试点。

6.1 create & remove

```
1   void syscall_create(struct intr_frame *f)
2   {
3           char *file_name = *(char **)check_ptr(f->esp + 4, 4);
4           check_str(file_name);
5           int file_size = *(int *)check_ptr(f->esp + 8, 4);
6
7           lock_acquire(&filesys_lock);
8           f->eax = filesys_create(file_name, file_size);
9           lock_release(&filesys_lock);
10  }
11
12  void syscall_remove(struct intr_frame *f)
13  {
14          char *file_name = *(char **)check_ptr(f->esp + 4, 4);
15          check_str(file_name);
16
17          lock_acquire(&filesys_lock);
18          f->eax = filesys_remove(file_name);
19          lock_release(&filesys_lock);
20  }
```

```
1   pass tests/userprog/create-normal
2   pass tests/userprog/create-empty
3   pass tests/userprog/create-null
4   pass tests/userprog/create-bad-ptr
5   pass tests/userprog/create-long
6   pass tests/userprog/create-exists
7   pass tests/userprog/create-bound
```

6.2 open & close

根据文档，每个进程需要维护其打开的文件信息。为此，需要在 `thread` 结构中新增一个文件列表，并维护文件的相关信息，如文件描述符和文件指针。在 `init_thread` 函数中初始化该链表。同时，需要记录分配文件的下标，并将其初始化为 2，以避开标准输入输出 `buffer`。

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

struct list child_list;
struct exit_info *linked_exit;
struct semaphore sema_wait;
struct semaphore sema_exec;
bool exec_success;

struct list file_list;
uint32_t file_index;

#endif

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

为了对指定文件进行读写操作，需要在 file_list 中找到相应的文件，实现辅助函数来遍历链表。

```
struct file_info *foreach_file(int fd)
{
        struct thread* cur = thread_current();
        struct list * l = &cur->file_list;
        struct list_elem *e;

        for (e = list_begin(l); e != list_end(l); e = list_next(e)) {
                struct file_info* tmp = list_entry(e, struct file_info, elem
                    );
                if (tmp->fd == fd) return tmp;
        }
        return NULL;
}
```

此时可以实现系统调用。打开文件时，将文件初始化并插入链表；关闭文件时，将其从链表中删除并释放内存。最后在 process_exit 时释放所有打开的文件。

```
while (!list_empty(&cur->file_list)) {
        e = list_pop_front(&cur->file_list);
```

```
3               struct file_info* tmp = list_entry(e, struct file_info, elem);
4               lock_acquire(&filesys_lock);
5               file_close(tmp->f);
6               lock_release(&filesys_lock);
7               free(tmp);
8       }
9
10      void syscall_open(struct intr_frame *f) {
11              char *file_name = *(char **)check_ptr(f->esp + 4, 4);
12              check_str(file_name);
13
14              lock_acquire(&filesys_lock);
15              struct file* open_file = filesys_open(file_name);
16              lock_release(&filesys_lock);
17
18              if (open_file == NULL) {
19                      f->eax = -1;
20                      return;
21              }
22
23              struct thread *cur = thread_current();
24              struct file_info *tmp = malloc(sizeof(struct file_info));
25              tmp->f = open_file;
26              tmp->fd = cur->file_index++;
27              list_push_back(&cur->file_list, &tmp->elem);
28              f->eax = tmp->fd;
29      }
30
31      void syscall_close(struct intr_frame *f) {
32              int fd = *(int *)check_ptr(f->esp + 4, 4);
33
34              struct file_info* tmp = foreach_file(fd);
35              if (tmp != NULL) {
36                      lock_acquire(&filesys_lock);
37                      file_close(tmp->f);
38                      list_remove(&tmp->elem);
39                      free(tmp);
40                      lock_release(&filesys_lock);
41              }
42      }
```

此时可以通过 open 和 close 相关的测试点。

```
1       pass tests/userprog/open-null
2       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
```

6.3 read & write & filesize

由于 read 和 write 的测试需要调用 filesize 系统调用，因此在此处也需要实现 filesize。直接根据文档调用文件系统的 API 即可，但不要忘记加锁。此外，还需要注意将 src/tests/vm/sample.txt 和 src/tests/userprog/sample.txt 文件的行尾序列改为 LF。完成这些后，即可通过 read 和 write 相关的测试点。

```
void syscall_read(struct intr_frame *f)
{
        int fd = *(int *)check_ptr(f->esp + 4, 4);
        char *buf = *(char **)check_ptr(f->esp + 8, 4);
        check_str(buf);
        int size = *(int *)check_ptr(f->esp + 12, 4);

        if (fd == 1) error_exit();
        if (fd == 0)
        {
                for (int i = 0; i < size; i++)
                *buf = input_getc(), buf++;
                f->eax = size;
        }
        else
        {
                struct file_info* tmp = foreach_file(fd);
                if (tmp == NULL) f->eax = -1;
                else
                {
                        lock_acquire(&filesys_lock);
                        f->eax = file_read(tmp->f, buf, size);
                        lock_release(&filesys_lock);
                }
        }
}

void syscall_write(struct intr_frame *f)
{
```

```
30              int fd = *(int *)check_ptr(f->esp + 4, 4);
31              char *buf = *(char **)check_ptr(f->esp + 8, 4);
32              check_str(buf);
33              int size = *(int *)check_ptr(f->esp + 12, 4);
34
35              if (fd == 0) error_exit();
36              if (fd == 1)
37              {
38                      putbuf(buf, size);
39                      f->eax = size;
40              }
41              else
42              {
43                      struct file_info* tmp = foreach_file(fd);
44                      if (tmp == NULL) f->eax = -1;
45                      else
46                      {
47                              lock_acquire(&filesys_lock);
48                              f->eax = file_write(tmp->f, buf, size);
49                              lock_release(&filesys_lock);
50                      }
51              }
52      }
53
54      void syscall_filesize(struct intr_frame *f)
55      {
56              int fd = *(int *)check_ptr(f->esp + 4, 4);
57              struct file_info* tmp = foreach_file(fd);
58              if (tmp->f == NULL) f->eax = -1;
59              else
60              {
61                      lock_acquire(&filesys_lock);
62                      f->eax = file_length(tmp->f);
63                      lock_release(&filesys_lock);
64              }
65      }
```

```
1       pass tests/userprog/read-normal
2       pass tests/userprog/read-bad-ptr
3       pass tests/userprog/read-boundary
4       pass tests/userprog/read-zero
5       pass tests/userprog/read-stdout
6       pass tests/userprog/read-bad-fd
7       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
```

6.4 seek & tell

同样根据文档实现，直接调用文件系统的 api。

```
void syscall_seek(struct intr_frame * f)
{
        int fd = *(int *)check_ptr(f->esp + 4, 4);
        int pos = *(int *)check_ptr(f->esp + 8, 4);
        struct file_info* tmp = foreach_file(fd);
        if (tmp != NULL)
        {
                lock_acquire(&filesys_lock);
                file_seek(tmp->f, pos);
                lock_release(&filesys_lock);
        }
}

void syscall_tell(struct intr_frame *f)
{
        int fd = *(int *)check_ptr(f->esp + 4, 4);
        struct file_info* tmp = foreach_file(fd);
        if (tmp != NULL)
        {
                lock_acquire(&filesys_lock);
                f->eax = file_tell(tmp->f);
                lock_release(&filesys_lock);
        }
        else f->eax = -1;
}
```

自此，除了 rox 相关的测试点，其他全部通过。

```
FAIL tests/userprog/rox-simple
FAIL tests/userprog/rox-child
FAIL tests/userprog/rox-multichild
```

7 Denying Writes to Executables

在程序运行时，拒绝其他程序对其进行写入操作。在 thread 中记录当前运行的文件 exec_file，注意初始化为 NULL。按照文档要求，在加载程序文件前后获取和释放锁。读文件成功后，调用 file_deny_write 并更新 exec_file。在 thread_exit 的结尾，判断如果 exec_file 不为空，则调用 file_allow_write 并关闭文件。

```
1     // start process
2     lock_acquire(&filesys_lock);
3     success = load (file_name, &if_.eip, &if_.esp);
4     if (success)
5     {
6             struct file *f = filesys_open(file_name);
7             cur->exec_file = f;
8             file_deny_write(f);
9     }
10    lock_release(&filesys_lock);
11
12    // thread_exit
13    if (cur->exec_file != NULL)
14    {
15            lock_acquire(&filesys_lock);
16            file_allow_write(cur->exec_file);
17            file_close(cur->exec_file);
18            lock_release(&filesys_lock);
19    }
```

8 实验结果：make check 全部通过

至此，make check 中的测试点全部通过。

```
1     pass tests/userprog/args-none
2     pass tests/userprog/args-single
3     pass tests/userprog/args-multiple
4     pass tests/userprog/args-many
5     pass tests/userprog/args-dbl-space
6     pass tests/userprog/sc-bad-sp
7     pass tests/userprog/sc-bad-arg
8     pass tests/userprog/sc-boundary
9     pass tests/userprog/sc-boundary-2
10    pass tests/userprog/sc-boundary-3
11    pass tests/userprog/halt
12    pass tests/userprog/exit
```

```
pass tests/userprog/create-normal
pass tests/userprog/create-empty
pass tests/userprog/create-null
pass tests/userprog/create-bad-ptr
pass tests/userprog/create-long
pass tests/userprog/create-exists
pass tests/userprog/create-bound
pass tests/userprog/open-normal
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-bound
pass tests/userprog/exec-bound-2
pass tests/userprog/exec-bound-3
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
```

```
57      pass tests/userprog/multi-child-fd
58      pass tests/userprog/rox-simple
59      pass tests/userprog/rox-child
60      pass tests/userprog/rox-multichild
61      pass tests/userprog/bad-read
62      pass tests/userprog/bad-write
63      pass tests/userprog/bad-read2
64      pass tests/userprog/bad-write2
65      pass tests/userprog/bad-jump
66      pass tests/userprog/bad-jump2
67      pass tests/userprog/no-vm/multi-oom
68      pass tests/filesys/base/lg-create
69      pass tests/filesys/base/lg-full
70      pass tests/filesys/base/lg-random
71      pass tests/filesys/base/lg-seq-block
72      pass tests/filesys/base/lg-seq-random
73      pass tests/filesys/base/sm-create
74      pass tests/filesys/base/sm-full
75      pass tests/filesys/base/sm-random
76      pass tests/filesys/base/sm-seq-block
77      pass tests/filesys/base/sm-seq-random
78      pass tests/filesys/base/syn-read
79      pass tests/filesys/base/syn-remove
80      pass tests/filesys/base/syn-write
81      All 80 tests passed.
82      make[1]: Leaving directory /Users/wanghaisheng/Desktop/Coding/Courses/ECNU-
            Operating-System-WHS/pintos/src/userprog/build
```

9 实验总结

通过本次实验，我深入理解了操作系统中的系统调用机制，并成功实现了 Pintos 中的一系列系统调用功能，通过了 make check 中的全部测试点。在实验过程中，我不仅掌握了如何从用户态切换到内核态、传递参数和返回结果的基本流程，还学会了如何处理进程同步和文件系统的线程安全问题。特别是在实现文件操作系统调用时，我遇到了多个并发访问的挑战，通过引入全局锁和合理的内存管理，最终确保了系统的稳定性和正确性。