Project2

11510366 王梓芃

课程：cs302

实验人员：王梓芃 学号：11510366

时间：5.1~6.17

实验环境：pintos OS
linux 14.04

实验步骤：

Task1: Argument Passing

- 问题描述：
  The \process_execute(char *file_name)” function is used to create new user-level processes in Pin-
  tos. Currently, it does not support command-line arguments. You must implement argument passing,
  so that calling \process_execute(“ls -ahl”)” will provide the 2 arguments, [“ls”, “-ahl”], to the user program using argc and argv.
  All of our Pintos test programs start by printing out their own name (e.g. argv[0]). Since argument
  passing has not yet been implemented, all of these programs will crash when they access argv[0]. Until
  you implement argument passing, none of the user programs will work.
- 方法描述：
  我认为task1主要分两步：

    1. 分离参数
    2. 按照规则，把参数放入栈中。

- 实现方法：
  分离使用的是strtok_r();

```
  char *save_ptr;
  strlcpy (cmd_name, fn_copy, PGSIZE);
 //这里使用strtok函数分隔名字和参数提取参数
  cmd_name = strtok_r(cmd_name, " ", &save_ptr);
```

第二步,按照program的性质排序。
struct thread *t = thread_get(tid);
t->next_fd = 2;
t->prog_name = cmd_name;
list_init(&t->desc_table);
list_init(&t->children);

```
  int status = ipc_pipe_read("exec", tid);
  if (status != -1){
    struct process_pid *p = malloc(sizeof(struct process_pid));
    p->pid = status;
    //这里添加子进程
    list_push_back(&thread_current()->children, &p->elem);
  }
  return status;
```

cmd_name = strtok_r(cmd_name, ” “, &save_ptr);

Task2: Process Control Syscalls

- 问题描述:
  Pintos currently only supports one syscall: exit. You will add support for the following new syscalls: `halt`, `exec`, `wait`, and `practice`.
- 实现方法:
  所有的系统调用都放在数组里面。

```
  static int (*syscall_handlers[20]) (struct intr_frame *);
```

```
void
syscall_init (void)
{
  intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");
  syscall_handlers[SYS_EXIT] = &syscall_exit_wrapper;
  syscall_handlers[SYS_EXEC] = &syscall_exec_wrapper;
  syscall_handlers[SYS_HALT] = &syscall_halt_wrapper;
```

```
  syscall_handlers[SYS_WAIT] = &syscall_wait_wrapper;
  syscall_handlers[SYS_PRACTISE] = &syscall_practise;
}
```

halt:
关机，调用shutdown（）进一步调用shutdown_power_off（）。

```
void
shutdown (void)
{
    shutdown_power_off ();
}
```

exec:
类似Linux的fork系统调用，调用之前写好的process_execute（）来创建子进程。然后立即在子进程中使用Linux的execve系统调用。等待系统调用将等待特定的子进程退出。

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

  fn_copy = palloc_get_page (0);
  if (fn_copy == NULL)
    return TID_ERROR;
    //复制filename到fncopy
  strlcpy (fn_copy, file_name, PGSIZE);


  char *cmd_name = malloc (strlen(fn_copy)+1);
  if (cmd_name == NULL)
    return TID_ERROR;
  extract_command_name(fn_copy, cmd_name);

  tid = thread_create (file_name, PRI_DEFAULT, start_process, fn_copy);
  if (tid == TID_ERROR){
    palloc_free_page (fn_copy);
    free(cmd_name);
    return -1;
  }


  struct thread *t = thread_get(tid);
  t->next_fd = 2;
  t->prog_name = cmd_name;
  list_init(&t->desc_table);
```

```
  list_init(&t->children);

  int status = ipc_pipe_read("exec", tid);
  if (status != -1){

    struct process_pid *p = malloc(sizeof(struct process_pid));
    p->pid = status;
    list_push_back(&thread_current()->children, &p->elem);
  }
  return status;
}
```

wait:
等待线程死亡并返回其退出的状态。 如果它被内核终止，那么就返回-1。如果TID无效或者它不是调用进程的子进程，或者process_wait（）已经完成已成功的被给定的TID调用，那么就立即返回-1不执行等待。

```
int process_wait (pid_t child_tid)
{
  if(!process_is_parent_of(child_tid))
    return -1;
  remove_child(child_tid); //删除确保不会等待两次
  return ipc_pipe_read("wait", child_tid);
}
```

practice：
就是写一个简单的系统调用，没什么好说的。

```
static int syscall_practise(int i){
    return i+1;
    }
```

Task 3: File Operation Syscalls

- 问题描述：
  implement these file operation syscalls: `create`, `remove`, `open`, `filesize`, `read`, `write`, `seek`, `tell`, and `close`.
- 实现方法：
  文件系统结构如下：

```
struct file
  {
    struct inode *inode;        //内节点
    off_t pos;
```

```
    bool deny_write;
  };
```

```
    struct inode
     {
       struct list_elem elem;
       block_sector_t sector;              // 磁盘扇区位置
       int open_cnt;                       //磁道
       bool removed;
       int deny_write_cnt;                 // 0: 可写,  >0: 不可写
       struct inode_disk data;
     };
```

虽然可能不太合适，我还是把文件操作调用的函数也放到Task2里面提到的系统调用的数组里了。

```
void
syscall_init (void)
{
  intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");
  syscall_handlers[SYS_EXIT] = &syscall_exit_wrapper;
  syscall_handlers[SYS_WRITE] = &syscall_write_wrapper;
  syscall_handlers[SYS_EXEC] = &syscall_exec_wrapper;
  syscall_handlers[SYS_HALT] = &syscall_halt_wrapper;
  syscall_handlers[SYS_WAIT] = &syscall_wait_wrapper;
  syscall_handlers[SYS_CREATE] = &syscall_create_wrapper;
  syscall_handlers[SYS_REMOVE] = &syscall_remove_wrapper;
  syscall_handlers[SYS_OPEN] = &syscall_open_wrapper;
  syscall_handlers[SYS_CLOSE] = &syscall_close_wrapper;
  syscall_handlers[SYS_READ] = &syscall_read_wrapper;
  syscall_handlers[SYS_FILESIZE] = &syscall_filesize_wrapper;
  syscall_handlers[SYS_SEEK] = &syscall_seek_wrapper;
  syscall_handlers[SYS_TELL] = &syscall_tell_wrapper;
  syscall_handlers[SYS_PRACTISE] = &syscall_practise;
}
```

create：
创建，需要一个参数定义大小，只需要考虑是否重复定义相同文件和大小是否足够。返回一个boolean。

```
bool
filesys_create (const char *name, off_t initial_size)
{
  block_sector_t inode_sector = 0;
  struct dir *dir = dir_open_root ();
  bool success = (dir != NULL
                  && free_map_allocate (1, &inode_sector)
                  && inode_create (inode_sector, initial_size)
                  && dir_add (dir, name, inode_sector));
```

```
  if (!success && inode_sector != 0)
    free_map_release (inode_sector, 1);
  dir_close (dir);

  return success;
}
```

remove：
删除制定的文件。先指定文件名，和目录，搜索如果有删除。

```
bool
dir_remove (struct dir *dir, const char *name)
{
  struct dir_entry e;
  struct inode *inode = NULL;
  bool success = false;
  off_t ofs;

  ASSERT (dir != NULL);
  ASSERT (name != NULL);


  if (!lookup (dir, name, &e, &ofs))
    goto done;

  inode = inode_open (e.inode_sector);
  if (inode == NULL)
    goto done;

  e.in_use = false;
  if (inode_write_at (dir->inode, &e, sizeof e, ofs) != sizeof e)
    goto done;

  inode_remove (inode);
  success = true;

 done:
  inode_close (inode);
  return success;
}
```

open：
用给定的文件名打开文件。成功返回文件。否则，如果没有名为NAME的文件存在，则失败。

```
struct file *
filesys_open (const char *name)
{
  struct dir *dir = dir_open_root ();
  struct inode *inode = NULL;
```

```
  if (dir != NULL)
    dir_lookup (dir, name, &inode);
  dir_close (dir);

  return file_open (inode);
}
```

filesize：

```
int process_filesize (int fd)
{
  if (get_fd_entry(fd) != NULL){
    struct fd_entry *fd_entry = get_fd_entry(fd);
    return file_length(fd_entry->file);
  }
  return -1;
}
```

最终调用到了inode里面的length，按照byte返回大小。

```
off_t
inode_length (const struct inode *inode)
{
  return inode->data.length;
}
```

read，write：
读是通过读指定file, buffer和size，将file的size个字节读到buffer，返回实际读取的大小。（读到0就可能小于size）
写也是相似的操作。

```
off_t
file_read (struct file *file, void *buffer, off_t size)
{
  off_t bytes_read = inode_read_at (file->inode, buffer, size, file->po
s);
  file->pos += bytes_read;
  return bytes_read;
}

off_t
file_write (struct file *file, const void *buffer, off_t size)
{
  off_t bytes_written = inode_write_at (file->inode, buffer, size, file->
pos);
  file->pos += bytes_written;
  return bytes_written;
```

```
}
```

seek，tell：
从用户栈中取出文件句fd柄要移动的距离，把fd转为文件指针，调用file_seek()函数移动文件指针即可。
tell类似的，调用file_tell（）得到指针位置。

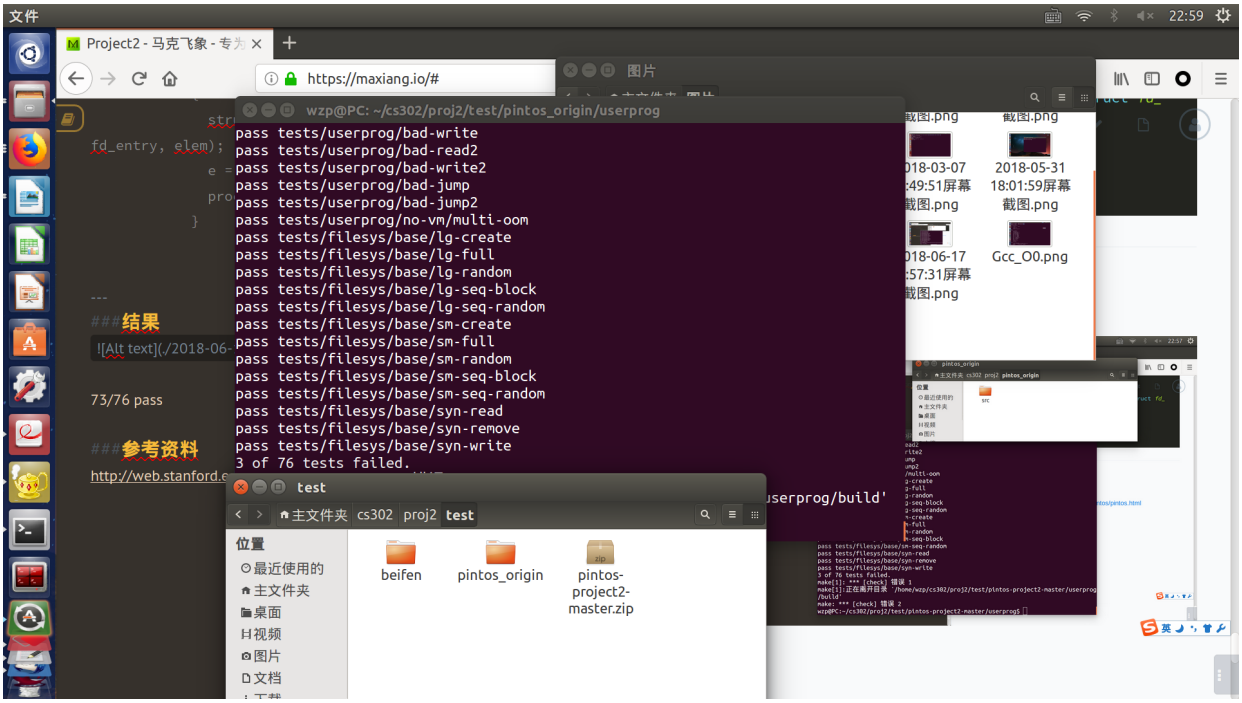
```
void process_seek (int fd, unsigned position){
  if (get_fd_entry(fd) != NULL){
    struct fd_entry *fd_entry = get_fd_entry(fd);
    file_seek(fd_entry->file, position);
  }
}

int process_tell (int fd)
{
  if (get_fd_entry(fd) != NULL){
    struct fd_entry *fd_entry = get_fd_entry(fd);
    return file_tell(fd_entry->file);
  }
  return -1;
}
```

close`.
调用process_close_all()关闭所有打开的文件

```
void process_close_all(void)
{
  struct list *fd_table = &thread_current()->desc_table;
  struct list_elem *e = list_begin (fd_table);
  while (e != list_end (fd_table))
    {
      struct fd_entry *tmp = list_entry (e, struct fd_entry, elem);
      e = list_next (e);
      process_close(tmp->fd);
    }
```

结果

73/76 pass



参考资料

http://web.stanford.edu/class/cs140/projects/pintos/pintos.html