OS Project 2 Report

User Program

Yue Leng 11510213

1. Introduction

This project is to finish three tasks in the user program realization part of pintos. To finish these task, firstly, I read the related resources about pintos and try my best to understand the code. And then, modify the code according to the task requirements.

This project aims to implement the necessary features for user programs (the test programs) to request kernel functionality. We need to finish three tests: argument passing, process control syscalls, file operation syscalls.

2. Methodology

2.1. Task1 - Argument Passing

2.1.1. Main Ideas

Currently, pintos does not support command-line arguments. We must implement argument passing. To understand the process of argument passing, we need three steps:

a. Parsing the command line
b. Put the argument into stack
c. Check the safe memory access

2.1.2. Implementation

a. Add a function string partition method, which minus the stack pointer 4 bytes once and copy 4 bytes from memory ptr points at to esp points at (the stack). Modify the function execute_process to cut the long cmd and give the process a real name.

```
char *token, *save_ptr;
file_name = strtok_r (file_name, " ", &save_ptr);
success = load (file_name, &if_.eip, &if_.esp);
free (file_name);
```

b. Modify the function setup_stack to count the number of args, parse the args and push the stack to store args and address one by one. Use function strtok_r to cut the command. Notice: after pushing the args, there's a word-align needed to be paid attention to.

```
//参数地址入栈
int zero=0;
if_.esp-=4;
memcpy(if_.esp,&zero, sizeof(int));

for(int i=argc-1;i>=0;i--){
  if_.esp-=4;
  memcpy(if_.esp,&argv[i],sizeof(int));
}
int argv_start=(int)if_.esp;
if_.esp-=4;
memcpy(if_.esp,&argv_start,sizeof(int));
if_.esp-=4;
memcpy(if_.esp,&argc,sizeof(int));
if_.esp-=4;
memcpy(if_.esp,&zero,sizeof(int));
```

2.2. Task2 - Process Control Syscalls

2.2.1. Main Ideas

To finish this task, we must understand how a system call is happened, and how the operation system change from the user space to the kernel space. Therefore, I summary the main steps of a system call happened in the following.

a. Pintos uses int 0x30 for system calls
b. Pintos has code for dispatching syscalls from user programs. i.e. user processes will push parameters onto the stack and execute int 0x30
c. In the kernel, Pintos will handle int 0x30 by calling syscall_handler() in userprog/syscall.c

In task2, I will add support for the following new syscalls: halt, exec, wait, and practice. Each of these syscalls has a corresponding function inside the user-level library, lib/user/syscall.c.The kernel's syscall handlers are located in userprog/syscall.c.

a. Halt: shutdown the system.
b. Exec: start a new program with process_execute().The Pintos exec syscall is similar to calling Linux's fork syscall and then Linux's execve syscall in the child process immediately afterward.)
c. Wait: wait for a specific child process to exit. This system call is the most complicated one in this project.

2.2.2. Implementation

a. For process execute function, call sema_down for the current running thread to make it synchronized for the execution. If the process is executed successfully, return the child process's tid. If the process isn't executed successfully, then return TID_ERROR.

b. SYS_HALT: Just add function shutdown_power_off().

c. For process wait function, firstly, if the arg is TID_ERROR, return -1 as an error message. For every process, it will always setup a semaphore for his parent process to wait. The parent process will wait for the child process's termination and then get the semaphore. Because the system will free all information of a dead process. We then use the struct heritage to keep these information.

d. SYS_EXIT: Firstly, check if the argument it passed (the exit code) is valid. If it's valid, pass it to current thread's exit_code and call thread_exit(), which is modified also and will be described later.

e. SYS_EXEC: Firstly, check if the argument it passed (the file name pointer) is valid and check if the content that the pointer points at is valid, too. If it's valid, strtok_r the file name and pass it to the function process_execute().

f. SYS_WAIT: Firstly, check if the argument it passed (the pid) is valid. Then call the wait function.

```
switch (type){
  case SYS_HALT:
    shutdown_power_off();

  case SYS_EXIT:
    verity_address((void *)p);
    thread_current()->exit_status = *p;
    thread_exit ();

  case SYS_EXEC:
    verity_address((void *)p);
    verity_address((void*)*p);
    f->eax = process_execute((char*)*p);
    break;

  case SYS_WAIT:
    verity_address((void *)p);
    f->eax = process_wait(*p);
    break;
```

2.3. Task3 – File Operation Syscalls

2.3.1. Main Ideas

In addition to the process control syscalls, you will also need to implement these file operation syscalls: create, remove, open, filesize, read, write, seek, tell, and close. The are some things to notice:

a. While a user process is running, you must ensure that nobody can modify its executable on disk. The tests ensure that you deny writes to current-running program files. The functions file_deny_write() and file_allow_write() can assist the function.
b. We must make sure that your file operation syscalls do not call multiple filesystem functions concurrently, The Pintos filesystem is not thread-safe. Every file operation (like create, open, etc.) needs to call function. lock_acquire and lock_release to keep synchronization for file operations.

2.3.2. Implementation

a. Use a global lock to ensure the file syscalls is thread safe. When every syscall is called, it must acquire lock and after then, release the lock.
b. Add a new struct file_obj to keep more information of a single file, whose variable has struct file *ptr to keep the address of a file, int fd to keep the file descriptor of a file, struct list_elem elem to be pushed into a thread's files list.
c. Add new member variables to struct thread: int fd_others to keep the file descriptor larger than STDIN_FILENO and STDOUT_FILENO, list files to keep a thread's opened files, file *self to keep its own file info.
d. For read system call, Firstly, if the file descriptor is STDIN_FILENO, the function will store input to the buffer and then put them to the console. If

the file descriptor is larger than or equal to 2, then we traverse current thread's file list and call file_read() to read several size of contents.

```
case SYS_OPEN:
  verity_address((void *)p);
  verity_address((void*)*p);
  struct thread * t=thread_current();
  acquire_file_lock();
  struct file * thefile =filesys_open((const char *)*p);
  release_file_lock();
  if(thefile){
    struct opened_file *of = malloc(sizeof(struct opened_file));
    of->fd = t->next_fd++;
    of->file = thefile;
    list_push_back(&t->files, &of->file_elem);
    f->eax = of->fd;
  } else
    f->eax = -1;
  break;
```

e. For READ sysyem call. Firstly, if the file descriptor is STDOUT_FILENO, the function will put content in buffer to the console. If the file descriptor is larger than or equal to 2, then we traverse current thread's file list and call file_write() to write several size of contents to the file.

f. And don not forget to close the file, and then release the resoueces.

```
case SYS_CLOSE:
  verity_address((void *)p);
  struct opened_file * openf=search_file(*p);
  if (openf){
    acquire_file_lock();
    file_close(openf->file);
    release_file_lock();
    list_remove(&openf->file_elem);
    free(openf);
  }
  break;
```

3. Experiment Verification

All pass

```
yue@ubuntu:~/os/project2/userprog$ make check
cd build && make check
make[1]: Entering directory '/home/yue/os/project2/userprog/build'
pass tests/userprog/args-none
pass tests/userprog/args-single
pass tests/userprog/args-multiple
pass tests/userprog/args-many
pass tests/userprog/args-dbl-space
pass tests/userprog/sc-bad-sp
pass tests/userprog/sc-bad-arg
pass tests/userprog/sc-boundary
pass tests/userprog/sc-boundary-2
pass tests/userprog/halt
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-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.
make[1]: Leaving directory '/home/yue/os/project2/userprog/build'
```