Pintos

Mohamed Thasneem
S2000337 UFCFWK-15-2

Table of Contents

Introduction

Pintos is a simple operating system framework for the 80x86 architecture. It supports kernel threads, loading and running user programs, and a file system, but it implements all of these in a very simple way (Pfaf, 2009). Base code of UWE variation of PintOS does not support argument passing and syscall. This document will go through the assignment tasks that we completed by implementing argument passing and system calls. We were given the base code for the UWE variation of Stanford Pintos. This document is divided into two sections. The first stage included implementing argument passing, while the second stage involved implementing system calls.

Setting up Environment

To get Pintos running correctly, $PATH variables has to set according to local user's folder path. Path for **kernel.bin** (in utils/pintos file) and **loader.bin** (in utils/Pintos.pm file) has to be defined correctly.

File Changes

To implement argument passing and syscall, below mentioned file were modified.

exception.c in userprog folder
process.c in userprog folder
process.h in userprog folder
syscall.c in userprog folder
syscall.h in userprog folder
synch.c in threads folder
thread.c in threads folder
thread.h in threads folder

Argument Passing

By sending arguments in registers and stack, a user application or any other program can call system calls. To pass arguments, we must first ensure that the stack is available and that we may execute the program's main functions.

Base code of the UWE version of Pintos was unable to accept any arguments provided to it. To do this, we must first extract the actual file name from the command line, to do this **get_filename()** has been defined in process.c. And it has been utilized into the **process_execute()** function.

```
tid_t process_execute (const char *args)
{
  char *argmnts_c;
  tid_t tid;
  char file_name[NAME_MAX_SIZE];
  struct process *p;
  struct thread *cur;

  /* Make a copy of FILE_NAME. */
  argmnts_c = palloc_get_page (0);

  if (argmnts_c == NULL)
    return TID_ERROR;
  strlcpy (argmnts_c, args, PGSIZE);
  get_filename (args, file_name);

  /* Create a new thread to execute FILE_NAME. */
  tid = thread_create (file_name, PRI_DEFAULT, start_process, argmnts_c, true);

  if (tid == TID_ERROR)
    palloc_free_page (argmnts_c);

  cur = thread_current ();
  p = process_create (tid);
  if (p == NULL)
    {
      palloc_free_page (argmnts_c);
      return -1;
    }
  list_push_back (&cur->children, &p->elem);
  sema_down (&p->load);
  if (p->load_status == LOAD_FAILED)
    return -1;

  return tid;
}
```

Figure 1 - process_execute() function

The argument is passed to **load** in **start_process().** Below changes are made to **start_process()** and **load()** function.

```
static void start_process (void *args_)
{
  char *args = args_;
  struct intr_frame if_;
  bool success;
  struct thread *t = thread_current ();

  /* 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;

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

  palloc_free_page (args);
  update_parent_load_status (t, success ? LOAD_SUCCESS : LOAD_FAILED);
```

Figure 2 - start_process() function

```
bool load (const char *args, void (**eip) (void), void **esp)
{
  struct thread *t = thread_current ();
  struct Elf32_Ehdr ehdr;
  struct file *file = NULL;
  off_t file_ofs;
  bool success = false;
  int i;
  char file_name[NAME_MAX_SIZE];

  /* Allocate and activate page directory. */
  t->pagedir = pagedir_create ();

  if (t->pagedir == NULL)
    goto done;

  process_activate ();

  get_filename (args, file_name);
```

Figure 3 - load() function

Test Argument Passing

Argument testing was done by using ‘echo’. Below are the step by step commands used to test arguments using ‘echo’.

Go to the directory **~/Pintos /userprog/build** and run below commands in order

pintos-mkdisk filesys.dsk --filesys-size=2
pintos -f -q
pintos -p ../../examples/echo -a echo -- -q
pintos -q run 'echo'

```
Loading.....
Kernel command line: -q run echo
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  104,755,200 loops/s.
hda: 1,008 sectors (504 kB), model "QM00001", serial "QEMU HARDDISK"
hda1: 194 sectors (97 kB), Pintos OS kernel (20)
hdb: 5,040 sectors (2 MB), model "QM00002", serial "QEMU HARDDISK"
hdb1: 4,096 sectors (2 MB), Pintos file system (21)
filesys: using hdb1
Boot complete.
Executing 'echo':
echo
echo: exit(0)
Execution of 'echo' complete.
Timer: 66 ticks
Thread: 1 idle ticks, 64 kernel ticks, 1 user ticks
hdb1 (filesys): 33 reads, 0 writes
Console: 625 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
thasbe@thasbe-virtual-machine:~/Downloads/Pintos/userprog/build$
```

Figure 4 - argument passing

System Calls

To implement syscall functions, first switch case statements are used in **syscall_handler()** function, when the syscall function is called it returns the respective output. And if an invalid syscall function is called, will cause process to exit. Below table represents main syscalls and a short description.

Wait	int **sys_wait** (pid_t pid)
	This function is used to wait for pid verification
Remove	bool **sys_remove** (const char *file)
	When this function is called, it removes the file of provided name.
Read	int **sys_read** (int fd, void *buffer, unsigned length)
	file_read function is used within this function to read the bytes to buffer from the given file, and returns the bytes as an integer.
Write	int **sys_write** (int fd, const void *buffer, unsigned int length)
	file_write function is used within this function to write the bytes from buffer from the given file, and returns the number of written bytes.
Close	void **sys_close** (int fd)
	file_close function is used within this function to close the file. Memory is deallocated and filemap structure is removed from threads after closing.
Tell	unsigned **sys_tell** (int fd)
	A file descriptor is passed as an argument to retrieve file structure from threads file list.
Open	int **sys_open** (const char *file)
	A file descriptor is added to filemap, and opens the file using filesys_open function.
Create	bool **sys_create** (const char *name, unsigned int initial_size)
	When this function is called, it creates a new file of provided name and size.
Halt	void **sys_halt** (void)
	calls shutdown_power_off (); function and shutdowns
Exit	void **sys_exit** (int status)
	This function is used to terminate current user program

Testing

To test the syscalls, navigate to **~/Pintos /userprog/build** directory and run the command **make check.**

Before executing the command ensure that **make** command has been executed in below mentioned folders.

~/Pintos /threads/
~/Pintos /examples/
~/Pintos /userprog/
~/Pintos /utils/
~/Pintos /vm/
~/Pintos /filesys/

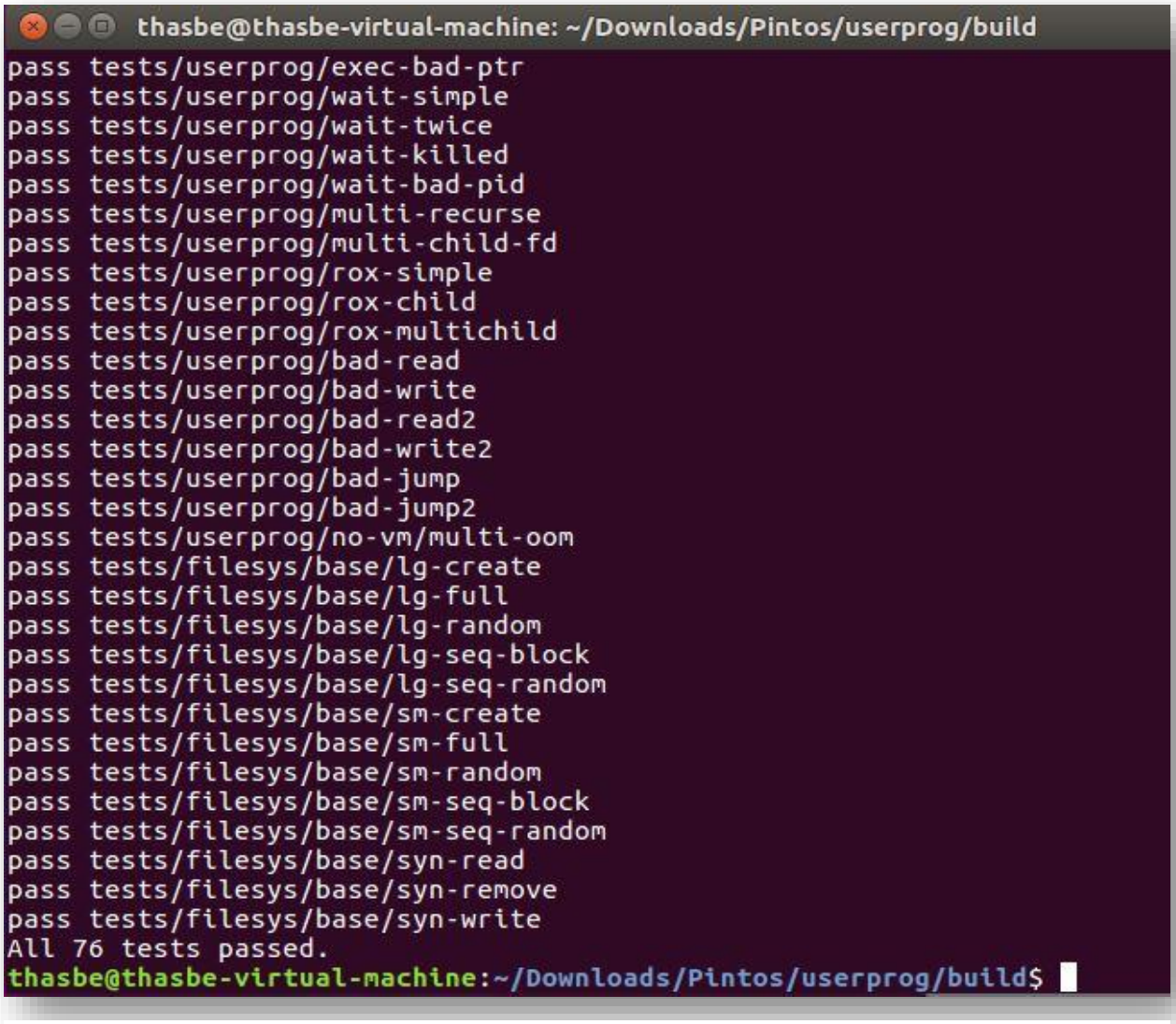

Figure 5 - syscall test results

References

Pfaf, B. (2009) *Pintos Projects: Introduction - Stanford University*. December 29, 2009 [online]. web.stanford.edu. Available from: https://web.stanford.edu/class/cs140/projects/pintos/pintos_1.html#SEC1 [Accessed 25 August 2022].