Project 2: User Program

Instructor: Prof. Seokin Hong (seokin@knu.ac.kr)
TA : Seungmin Shin (steve6238@naver.com)

Assigned: April 30, 2019
Due: 11:59 pm May 17, 2019

1. Description

In this project, your job is to extend the PintOS to run user programs. We will allow more than one process to run at a time on PintOS, which means that PintOS should be able to load and run multiple processes at a time. In the project 1, we compiled the test code directly into the kernel. So, we need to extend the PintOS to provide some specific user program interfaces (system call interface). For more information, please refer to the official Pintos documentation on the following website.
https://web.stanford.edu/class/cs140/projects/pintos/pintos_1.html

2. What to do

a. Part 1: Argument Passing and Setup Stack

Support the feature of parsing the user program's filename and setup the stack correctly. You will be mainly working in src/userprog/process.c.

b. Part 2: System Calls

Implement the required functions in src/usrprog/syscall.h and src/usrprog/syscall.c

3. Recommended order of implementation for this project

a. **change process_wait() to an infinite loop**

b. **Argument passing and setup stack**

c. **System call infrastructure**

Implement code to read the system call number from the user stack and invoke a system call handler based on it.

d. **exit() system call**

e. **write() system call**

f. **other system calls**

Implement the system call handler in 'useprog/syscall.c'. There is a skeleton implementation for each system call in the file. You need to implement each system call handler in a way that it retrieves the system call number, any system call arguments, and performs appropriate action. System call numbers for each system call are defined in ‘lib/syscall-nr.h’

List of system call you need to implement.

halt(), exit(), exec(), wait(), create(), remove(), open(), filesize(), read(), write(), seek(), tell(), close()

You can find more information about the system call in the section 3.3.4 of the Stanford Pintos document(https://web.stanford.edu/~ouster/cgi-bin/cs140-winter13/pintos/pintos.pdf).

g. **Denying writes to executables**
Add code to deny writes to files in use as executables. You can use file_deny_write() to prevent writes to an open file.

4. **Background** (Contents are mainly from Stanford Pintos documents)

a. Using the file systems

For this project, you will need to interface to the file system because the user programs are loaded from the file system. You can come to know how to use the file system by looking over the ‘filesys.h’ and ‘file.h’ interfaces. The current implementation of the file system is very simple and has several limitations. But, it is enough for this project.
For this project, you need a simulated disk with a file system partition. So, you need to create a simulated disk with a file system, format the with the file system, copy the program into the new disk, and run the program with the following command lines.

```
$ cd userprog/build
$ pintos-mkbuild filesys.dsk --filesys-size=2          //create a simulated disk
$ pintos -f -q                                        //format
$ pintos -p ../../examples/echo -a echo -- -q          //copy “echo” program into the new disk
$ pintos -q run ‘echo x’                               //run the “echo” program
```

b. How User Program Work

Pintos can run normal C programs if they fit into memory and use only the system calls you implement in this project.
The ‘src/examples’ directory contains some user programs that can run on Pintos. You can compile the example programs with a Makefile provided in the directory.
Pintos can load ELF executables with the loader provided in ‘userprog/process.c’. ELF is a file format used by Linux, Solaris, and many other operating systems for object files, shared libraries, and executables.

c. Virtual Memory Layout

Virtual memory in Pintos is divided into two regions: user virtual memory and kernel virtual memory. User virtual memory ranges from virtual address 0 up to PHYS_BASE that is defined in ‘threads/vaddr.h’. Kernel virtual memory ranges from PHYS_BASE to 4GB. struct
User virtual memory is per-process. So, on context switch, Pintos switches user virtual address space by changing the processor’s page directory base register. On the other hand, kernel virtual memory is global, which means that it is mapped one-to-one to physical memory regardless of what user process is running. In Pintos, virtual address PHYS_BASE accesses physical address 0, virtual address PHYS_BASE+0x1234 accesses physical address 0x1234.

User process can only access its own virtual memory. Kernel thread can access both kernel virtual memory and user virtual memory.
The following figures show the typical user virtual memory layout.

```
   PHYS_BASE +----------------------------------+
             |            user stack            |
             |                 |                |
             |                 |                |
             |                 V                |
             |          grows downward          |
             |                                  |
             |                                  |
             |                                  |
             |                                  |
             |           grows upward           |
             |                 ^                |
             |                 |                |
             |                 |                |
             +----------------------------------+
             | uninitialized data segment (BSS) |
             +----------------------------------+
             |     initialized data segment     |
             +----------------------------------+
             |           code segment           |
  0x08048000 +----------------------------------+
             |                                  |
             |                                  |
             |                                  |
             |                                  |
             |                                  |
           0 +----------------------------------+
```

In this project, the user stack and the uninitialized data segment are fixed in size. The code segment in Pintos starts at user virtual address 0x08048000 (approximately 128MB from the bottom of the address space).

5. What you need to submit

a. Design document (see section 7 below)

You need to submit a design document. Design document is one of the important parts for grading. Thus, please write the design document well. It will help use better understand your solution of the project and test your understanding of Pintos. You can find a design document template in section 7.

b. PintOS source code.

Caution! : Whenever a user process terminates, it will print a termination message. Your final submitted code should not print any other messages (e.g., debugging message) except the termination message.

6. How to evaluate your code

```
$ cd src/userprog
$ make
$ cd build
$ make SIMULATOR=--boch check
```

Expected results

```
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
0 of 76 tests failed
```

7. Design document template

```
---- GROUP ----

Fill in the names and email addresses of your group members.

FirstName LastName <email@domain.example>
FirstName LastName <email@domain.example>
FirstName LastName email@domain.example


---- PRELIMINARIES ----

Describe briefly which parts of the assignment were implemented by each member
of your team and specify the contribution between your member, say 3:3:4, or
1:3:6.

FirstName LastName: contribution
FirstName LastName: contribution
FirstName LastName: contribution
```

```
                         ARGUMENT PASSING
                         ================

---- DATA STRUCTURES ----

A1: Copy here the declaration of each new or changed `struct' or `struct'
member, global or static variable, `typedef', or enumeration.  Identify the
purpose of each in 25 words or less.

---- ALGORITHMS ----

A2: Briefly describe how you implemented argument parsing.  How do you arrange
for the elements of argv[] to be in the right order? How do you avoid
overflowing the stack page?


                           SYSTEM CALLS
                           ============

---- DATA STRUCTURES ----

B1: Copy here the declaration of each new or changed `struct' or `struct'
member, global or static variable, `typedef', or enumeration.  Identify the
purpose of each in 25 words or less.

B2: Describe how file descriptors are associated with open files. Are file
descriptors unique within the entire OS or just within a single process?

---- ALGORITHMS ----

B3: Describe your code for reading and writing user data from the kernel.

B4: Briefly describe your implementation of each system call.
```