CMPS111 Winter 2018 : Lab 3

In this lab you will implement user processes and system calls.

As supplied, Pintos is incapable of running user processes and only implements two systems calls. Pintos does, however, have the ability to load ELF binary executable, and has a fully functioning page-based, non-virtual memory management system.

There are three parts to this lab; each depends on the previous one.

- Allow simple user process to run.
- Support argument passing to user processes.
- Implement seven new systems calls.

This lab is worth 15% of your final grade.

Submissions are due NO LATER than 23:59, Monday March 12, 2017 (approximately three weeks)

Setup

SSH in to the CMPS111 teaching server using your CruzID Blue password:

`$ ssh <cruzid>@noggin.soe.ucsc.edu` (use Putty http://www.putty.org/ if on Windows)

Create a suitable place to work: ***(only do this the first time you log in)***

```
$ mkdir -p ~CMPS111/Lab3
$ cd ~CMPS111/Lab3
```

Install the lab environment: ***(only do this the first time you log in)***

```
$ tar xvf /var/classes/CMPS111/Winter18/Lab3.tar.gz
$ ./setenv
$ . ~/.bashrc
```

Build Pintos:

```
$ cd ~/CMPS111/Lab3/pintos/src/userprog
$ make
```

(note this is different to Labs 1 & 2)

Run the first test:

```
$ pintos -v -k -T 60 --qemu --filesys-size=2 \
    -p build/tests/userprog/args-none -a args-none \
    -- -q -f run args-none
```

Also try:

`$ make check` (runs the required functional tests - see below)

`$ make grade` (tells you what grade you will get - see below)

Accessing the teaching server file system from your personal computer

Follow the instructions from Lab 1:

https://classes.soe.ucsc.edu/cmps111/Winter18/SECURE/CMPS111-Lab1.pdf

Background Information

(1) Pintos Calling Conventions

Pinto is a Unix-like operating system and should implement the standard Unix / C calling convention.

To understand how arguments are passed to Unix / C programs, consider the command:

```
/bin/ls -l foo bar
```

Also recall that the prototype for a C program entry point is:

```
int main(int argc, char *argv[])
```

Where `argc` is the number of arguments passed to the program (including the program name) and `argv` is an array containing pointers to each of the arguments stored as null-terminated character arrays.

To execute this program with the supplied arguments, we need to do the following:

- Break this command into words: "`/bin/ls`", "`-l`", "`foo`", and "`bar`".
- Place these words at the top of the stack, in right to left order.
- Push onto the stack the *address* of each string plus a null pointer sentinel, in right-to-left order.
  - These are the elements of `argv`. The null pointer sentinel ensures that `argv[argc]` is a null pointer, as required by the C standard. The order ensures that `argv[0]` is at the lowest virtual address. Word-aligned accesses are faster than unaligned accesses, so for best performance, round the stack pointer down to a multiple of 4 before the first push.
- Push `argv` (the address of `argv[0]`) and `argc`, in that order.
- Finally, push a fake "return address". Although the entry function will never return, its stack frame must have the same structure as any other.

The figure below shows the contents in the stack before executing the user program. We assume here that `PHYS_BASE` is `0xc0000000`.

Address	Name	Data	Type
0xbffffffc	argv[3][...]	"bar\0"	char[4]
0xbffffff8	argv[2][...]	"foo\0"	char[4]
0xbffffff5	argv[1][...]	"-l\0"	char[3]
0xbfffffed	argv[0][...]	"/bin/ls\0"	char[8]
0xbfffffec	word-align	0	uint8_t
0xbfffffe8	argv[4]	0	char *
0xbfffffe4	argv[3]	0xbffffffc	char *
0xbfffffe0	argv[2]	0xbffffff8	char *
0xbfffffdc	argv[1]	0xbffffff5	char *
0xbfffffd8	argv[0]	0xbfffffed	char *
0xbfffffd4	argv	0xbfffffd8	char **
0xbfffffd0	argc	4	int
0xbfffffcc	return address	0	void (*) ()

As shown above, your code should start with the stack at the very top of the user virtual address space, in the page just below virtual address `PHYS_BASE` (defined `in threads/vaddr.h`).

The equivalent output for the “args-none” test is as follows:

```
Address    Name           Data            Type
0xbffffff6 argv[0][..]    'args-none\0'   char[10]
0xbffffff4 word-align     0               char[2]
0xbffffff0 argv[1]        0               char *
0xbfffffec argv[0]        0xbffffff6      char *
0xbfffffe8 argv           0xbfffffec      char **
0xbfffffe4 argc           1               int
0xbfffffe0 fake return    0               void(*)()
```

If your addresses EXACTLY match these, you are well on your way to a passing test.

(2) System Calls

Most user programs require services provided by Pintos; they access those capabilities by making system calls. To support this feature, you will need to extend the existing rudimentary system call implementation in `userprog/syscall.c`.

The system calls of interest in this lab are:

- **create :** Creates a new file. Return true if successful and false otherwise.
- **open** : Open a file and return the corresponding file descriptor (i.e. an integer handle). Note that file descriptor 0 is reserved for standard input and file descriptor 1 is reserved for standard output.
- **read** : Read a specified number of bytes from an existing, open file into a buffer in the user program, returning the number of bytes actually read, or -1 if read failed.
- **write**: Write a specified number of bytes to an open file from a buffer in the user program. Return the number of bytes actually written or -1 if an error occurs.
- **close** : Close an open file.
- **exec** : Starts the execution of a user program and returns the ID of the newly created child process if successful. The parent process should not return from the exec system call until it knows whether the child process has successfully loaded its executable code.
- **wait :** Waits for a child process to complete and retrieves the child's exit value.

Note that the above descriptions are a guide only, your system calls must do whatever the tests demand they do!

Using the GIT revision control system with Pintos

The CMPS111 teaching server has GIT installed and we *highly* recommend you use it to track the changes you make, though this is not required.

CMPS111 is not, however, a GIT training course. But Google is your friend, so find some beginner tutorials and study them if you have never used GIT. But also remember you have friends in the shape of your classmates, the TAs, and your instructor. If you want to know how to setup and use GIT, just ask someone.

Using the GNU Debugger (GDB) with Pintos

In Lab 1 and Lab 2, many of you will have "debugged" you code using `printf` statements and assertions.

The code you will write in this Lab is significantly more complex and an appropriately more sophisticated mechanism for examining your code as it runs is required.

Run a test in debug node (note the new argument):

```
$ pintos -v -k -T 60 --qemu --filesys-size=2 \
    -p build/tests/userprog/args-none -a args-none \
    --gdb -- -q -f run args-none
```

In a separate terminal, run GDB for Pintos:

```
$ pintos-gdb build/kernel.o
```

At the prompt, connect to the running test:

```
(gdb) target remote localhost:1234
```

GDB will respond:

```
Remote debugging using localhost:1234
0x0000fff0 in ?? ()
```

From there on in, set breakpoints and step into or over functions as required.

To get the test to run to completion, enter "C" at the `(gdb)` prompt.

If you don't know how to use GDB, remember, Google is your friend. A quick search for **"gdb tutorial C"** will find you an abundance of information.

What to submit

In a command prompt:

```
$ cd ~/CMPS111/Lab3/pintos/src/userprog
$ make submit
```

This creates a gzipped tar archive named `CMPS111-Lab3.tar.gz` in your home directory.

UPLOAD THIS FILE TO THE APPROPRIATE CANVAS ASSIGNMENT.

In addition to submitting modified and new source files, you are required to write a short report (no more than two pages) on your work.

This report should contain at least:

- A defense of the rationale behind your design
- Details of tests your submission fails and what investigations you undertook to try and find out why

If you keep a simple journal as you work your way through this lab, writing the report will be easy - it's essentially a tidied up version of your journal.

SUBMIT YOU REPORT TO CANVAS IN THE SAME ASSIGNMENT AS YOUR CODE ARCHIVE.

Note that the report *WILL NOT BE READ* unless plagiarism is detected in your submission.

Requirements

User Processes:

- Give Pintos the ability to execute user processes mapped one-to-one with kernel threads.
- Pass the following test:
    - args-none

Arguments to User Processes:

- Allow Pintos user processes to accept command line arguments.
- Pass the following tests:
    - args-single
    - args-multiple
    - args-many
    - args-dbl-space

System Calls:

- Implement the `create`, `open`, `read`, `write`, `close`, `exec`, and `wait` function calls.
- Pass the following tests:
    - create-empty
    - create-long
    - create-normal
    - create-exists
    - open-missing
    - open-normal
    - open-twice
    - read-normal
    - read-zero
    - write-normal
    - write-zero
    - close-normal
    - exec-once
    - exec-multiple
    - wait-simple
    - wait-twice

Grading Scheme

The following aspects will be assessed:

1. (80%) **Does it work?**

    a. User Processes (40%)
    b. Argument Passing (20%)
    c. System Calls (40%)

2. (-100%) **Did you give credit where credit is due?**

    a. You submission is found to contain code segments copied from on-line resources and you failed to give clear and unambiguous credit to the original author(s) in your source code (-100%)
    b. You submission is determined to be a copy of another past or current CMPS111 student's submission (-100%)
    c. Your submission is found to contain code segments copied from on-line resources that you did give a clear an unambiguous credit to in your source code, but the copied code constitutes too significant a percentage of your submission:

        - < 25% copied code No deduction
        - 25% to 50% copied code (-50%)
        - 50% to 75% copied code (-75%)
        - > 75% (-100%)

