CS342: OS Lab
Department of CSE,
IIT, Guwahati, Assam 781 039 Exercise 03

OS Lessons: Process, Kernel data-structures, User and Kernel Stacks
Rating: Very hard

In this exercise, we shift our attention to Project 2: User Programs as described in Chapter 3 of PintDoc. This is also one of the harder exercises as we need to find answers for many questions.

A few changes that you need to make to run pintos code for this new project:

The primary directory for User Programs project is `pintos/src/userprog`. To move to this directory, we need to run `make clean` in directory `pintos/src/threads`. And, also make the following changes in the script files:

The `perl` script `pintos` (which was placed previously in directory `~/bin`) be changed as follows:

> Line no.24: Replace "`$HOME/pintos/src/threads/build/os.dsk`" by "`$HOME/pintos/src/userprog/build/os.dsk`"
> (This file was mentioned in the installation instructions for Pintos)
>
> Also, please change the last line in `Make.vars` file in directory `pintos/src/userprog/userprog` to:
> `SIMULATOR = --bochs`

This is also a good place to caution you about the small size of space available for the Kernel stack(s). In fact, the page hosting the kernel stack for a thread also hosts all data-structures required for the thread management by the kernel. You are also reminded that the kernel and user virtual address spaces are disjoint and there are significant but essential controls built into the software to monitor access across the boundary between these two regions.

Task for the Exercise:

A curt description of the task for the exercise is to write code to meet the specifications set in PintDoc Section 3.3.3 *Argument Passing* on page 29. Let us be more friendly and supportive. It would be helpful if you also read the document [Executing `main()` function on Linux].

The first dot-point listed in Section 3.2 (page 28) suggests that we must write code in function `setup_stack()`. Section 3.1.4.1 on page 26 in PintDoc describes how the initial stack for function `main()` of the user program is organised. You must also carefully read the example in Section 3.5.1 *Program Startup Details* on pages 36-37.

We have already seen a function to print the contents of this initial stack in a previous exercise. The function is again listed here:

```
void test_stack(int *t)
{        int i;
         int argc = t[1];
         char ** argv;

         argv = (char **) t[2];
         printf("ARGC:%d  ARGV:%x\n", argc, (unsigned int)argv);
         for (i = 0; i < argc; i++)
                 printf("Argv[%d] = %x pointing at %s\n",
                        i, (unsigned int)argv[i], argv[i]);
}
```

Now the question we must ask and answer is when can we call this function to print the contents of the activation record that function `main()` will receive?

There are two ways to start a user program in PintOS. One is to write a command line and pass it to PintOS during the kernel load time. The kernel command directive `run` (see Section 3.1.2 *Using the File System*) takes a user command as its argument. The other way is through an already running user program; we will come back to this method in a later exercise.

A user command may have its arguments. This may make some commands multi-word commands. Quotes (‟ ‟) are used to group such commands as a single argument for directive `run`. PintOS kernel load time directives are similar to built-in commands in Unix/Linux shells/ See near the bottom of page 24 [PintDoc] to understand how a command is specified as a single argument to directive `run`. For this exercise, this is the only way we use to start a user program.

What resources a program needs to run? A program running on a computer is called a process. A process is made of many tangible and virtual components:

- A process needs a thread trace (= a linear sequence of memory addresses – some executed and others to be executed – listing every instruction run on the computer processor by the user program) to execute instructions,
- Memory pages to store program instructions,
- Initialized and uninitialized data segments in memory (also called static data area), and
- A user stack (see Section 3.1.4.1 on page 26).

A user program is ready to run when these resources are available to the process. As we know from a previous exercise, all ready to run processes are placed in `ready_list` to be scheduled by the PintOS scheduler. From the time the `run` directive notes a command to be run to the time it can assemble all resources needed by the process, the process is in the state preceding THREAD_READY. After PintOS kernel has assembled a process with all the needed resources, the newly created process is inserted into `ready_list`. This occurs near label `done` in function `load()` in file `pintos/src/userprog/process.c`. This obviously is the place to ensure that we have set the program stack correctly. Consider if you wish to call function `test_stack()` near label `done`.

Let us now consider the activity history of a thread as the control arrives at label `done`. The directive `run` noted the user command to load and run in function `run_actions()` of file `pintos/src/threads/init.c`. We need to get a good understanding of the path from `run_action()` to `done` as the user command arguments must flow along this path to finally land in the run-time user stack to be available to function `main()` of the user program.

Use `cscope`, `ctags`, `gdb` and actual reading of the code to understand how the command made of a command-file name and its arguments is passed (or could be passed) from a calling function in the PintOS kernel code to the called function. Your path begins at function `run_action()` and ends at function `setup_stack()`. You may like to read *Exercise 06* document to get a better view of the mysterious tricks used to let the user program load itself into the memory.

Once you have understood the route to carry the required information from the command to function `setup_stack()` in file `process.c`, you are ready to write code in function `setup_stack()` to build the stack that will be made available to the user program when the user program begins execution. The new user program begins at the start of its function `main()`.

However, testing this user program is not possible at this stage! The reason for this limitation is lack of system calls to write messages on the computer console! Only kernel code can print messages; user programs cannot write on console yet.

We overcome this limitation by calling function `test_stack()` in the kernel code (and not in the user program code).

How we tested our implementation:

The test sequence below is a reconstruction of the original activities that we used when we first completed this exercise. (The PintOS code being used to re-enact the exercise for this section is the completed code for the project. This completed code performance many more tasks.) Caution: You may experience significant differences in your output.

In the script below, the text typed by the user has been shown in bold. Some output from the standard Pintos utilities has been deleted as it provides no useful insight. The output that is of minor interest is shown in smaller fonts to fit the page width neatly.

These commands were used to compile programs in directory `~/pintos/src/examples`

```
[vmm@progsrv ~]$ cd pintos/src/examples/
[vmm@progsrv examples]$ make
```

[Output deleted]

These commands were used to setup PintOS to load and run a user program.

```
[vmm@progsrv examples]$ cd ../userprog/
[vmm@progsrv userprog]$ make
cd build && make all
make[1]: Entering directory
`/home/CS342/2016/FAC/vmm/pintos/src/userprog/build'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory
`/home/CS342/2016/FAC/vmm/pintos/src/userprog/build'
[vmm@progsrv userprog]$ cd build/
[vmm@progsrv build]$ pintos-mkdisk fs.dsk 2
[vmm@progsrv build]$ pintos -q –f
```

[Output deleted]

Finally, we test our implementation of function `stack_setup()` :

```
[vmm@progsrv build]$ pintos -p ../../examples/echo -a echo -- -q
[vmm@progsrv build]$ pintos -q run "echo My stack_setup() works"
```

```
Writing command line to /tmp/EHg1akwWBy.dsk...
squish-pty bochs -q
========================================================================
                        Bochs x86 Emulator 2.5.1
              Built from SVN snapshot on January 6, 2012
                  Compiled on Oct 10 2012 at 11:12:02
========================================================================
00000000000i[     ] reading configuration from bochsrc.txt
00000000000i[     ] installing nogui module as the Bochs GUI
00000000000i[     ] using log file bochsout.txt
Kernel command line: -q run 'echo My stack_setup() works'
Pintos booting with 4,096 kB RAM...
370 pages available in kernel pool.
369 pages available in user pool.
Calibrating timer...  204,600 loops/s.
hd0:0: detected 1,008 sector (504 kB) disk, model "Generic 1234", serial
"BXHD00011"
hd0:1: detected 4,032 sector (1 MB) disk, model "Generic 1234", serial
"BXHD00012"
Boot complete.
Executing 'echo My stack_setup() works':
ARGC:4  ARGV:bfffffc4
Argv[0] = bffffff0 pointing at echo
Argv[1] = bfffffed pointing at My
Argv[2] = bfffffdf pointing at stack_setup()
Argv[3] = bfffffd9 pointing at works
```

The last few lines above are of primary interest to verify the completion of the exercise. The remaining output below is from PintOS code that has not yet been included in your project. You may notice significant variation in your output (but the variation is not relevant to your exercise.)

```
echo My stack_setup() works
echo: exit(0)
Execution of 'echo My stack_setup() works' complete.
Timer: 183 ticks
```

```
Thread: 0 idle ticks, 133 kernel ticks, 53 user ticks
hd0:0: 0 reads, 0 writes
hd0:1: 28 reads, 0 writes
Console: 819 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
=========================================================================
Bochs is exiting with the following message:
[UNMP ] Shutdown port: shutdown requested
=========================================================================
[vmm@progsrv build]$
```

Contributing Authors:

Vishv Malhotra, Gautam Barua, Rashmi Dutta Baruah