CS342: Operating Systems Lab
Department of Computer Science and Engineering,
Indian Institute of Technology, Guwahati
North Guwahati, Assam 781 039 Exercise UP-01

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

Rating: Hard

Last update: 20 Feb 2017

This exercise is the first of the four exercises prepared to complete PintOS project User Programs. PintDoc lists this project as Project 2 and provides instructions and expectations in Chapter 3: User Programs.

Some general organizational arrangements and suggestions for the Exercise:

1. Use `cscope`, `ctags`, `gdb` and careful 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 search for the path that is necessary for setting up stack begins at function `run_action()` and ends at function `setup_stack()`. You may like to read guidance for *Exercise UP-04* to get a better view of some specialized programming patterns that operating systems use to create the user programs as separate entities from the kernel codes.

2. The primary directory for project User Programs is `pintos/src/userprog`. To move to this directory, we need to run `make clean` in directory `pintos/src/threads`. In addition, make the following changes to the primary script file.

   Change `Perl` script pintos (which was previously placed in directory `~/bin`) 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 `Pintos` installation instructions previously)

   Further, please change the last line in file `pintos/src/userprog/Make.vars` to:
   `SIMULATOR = --bochs`

3. This is also a good place to caution students about the limited amount of space available for the Kernel stack(s). Do not write a recursive functions in your kernel code to avoid stack overflow problems.

4. In all modern operating systems, the kernel and the user virtual address spaces are separate and all memory accesses across these boundaries is carefully monitored through the features built into the processor hardware and kernel code.

Statement of the Exercise UP-01

A curt description of the tasks for the exercise is to write code to meet the specifications set in PintDoc Section 3.3.3 *Argument Passing* on page 29. We will explain the specification in considerable detail here as this exercise is quite challenging.

First, please read document [Executing `main()` function on Linux] on the Internet.

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 a PintOS user program is organized. 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 reproduced here (The code assumes that a c-pointer and `int` are same size objects.):

```
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]);
}
```

One important aim of this exercise can be stated as a question: Where can we call this function to print the contents of the activation record that function `main()` will receive in the stack?

A search for the answer to the question will provide us a good overview of the kernel code that the exercise must expand on way to completing project User Programs.

User Interface to Create User Programs

There are two ways to start a user program in PintOS. One is to write a command line argument and pass it to PintOS during the kernel load time. The kernel command-line directive `run` (see Section 3.1.2 *Using the File System*) takes a user command as its argument. The other way to start a user program is through a system call from an already running user program; we will return to this method in a later exercise (UP-04).

A user command may have arguments in addition to the name of a file containing code to run as a user program. This makes some commands multi-word commands. A quote pair (“ ”) is used to group such commands as a single argument for PintOS kernel load-time command directive `run`. PintOS kernel load-time directives are similar to the built-in and separately-coded commands in Unix/Linux shells. See near the bottom of page 24 [PintDoc] to learn how a command is specified as a single argument to directive `run`. For this exercise, load-time commands are the only way we use to run user programs.

What resources does a user program need 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 to execute program instructions and receive run-time on the processor.
- Each process needs memory pages to store program instructions.
- Each process also needs initialized and uninitialized data segments in memory (also called static data area). And,
- Each process needs a user stack to perform function calls (see Section 3.1.4.1 on page 26) by passing arguments and return values.

A thread is ready to run as a user program/process when the above-listed resources are available in the process. A thread however can run without some of these resources (components) as a kernel thread.

All ready-to-run threads are placed in `ready_list` and are scheduled (given time to run on the computer processor) by PintOS scheduler. From the time the `run` directive receives a command to run to the time the kernel has made available all resources needed by the process, the process is in a state of *being prepared*.

Three Contexts and Three Phases

Students reading the kernel code for this exercise will find that it is organized quite differently from their previous understanding of program organizations. The reading of the code for this exercise is easier if we view the code as grouped into three discrete execution contexts.

First context is the context of the thread that initiates the creation of the user program as an entity (process) to be run on PintOS operating system. We call this context as the context of the parent thread. The kernel code that this parent thread runs to support the creation of the user program/process makes a unit. This demarcation will help in reading, understanding and modifying the kernel code as it addresses some specific requirements. A clear focus on this set of requirements would avoid distraction from the issues that concern the other parts of the kernel code that run in other contexts.

The other context is the context of the thread created to execute user program code. This thread is called a child thread. However, the context of a child thread is divided in two distinct phases or contexts. We discuss the final phase first.

It is obvious that the third or the final context is the context in which the child thread is able to run the user program. In this context the child thread is primarily controlled by the instructions in the user program and the thread's access to the kernel code is significantly limited and carefully regulated. This, however, is the target context of the exercise – the very purpose of this exercise.

The context of the execution between the first and the final context is the initial execution context of the child thread. In this context, PintOS scheduler schedules the child thread and the thread runs the kernel code. Once again it will aid understanding if all functions accessed by the child thread in this context are identified. This is so because the changes to the kernel code in these functions has a clear scope. This containment will help you decide the changes you wish to implement for completing exercise UP-01.

Summary: It is important to understand the existence of three separate contexts of code execution as a kernel builds a user program. Three execution contexts of interest are (i) Context of the parent thread running kernel code, (ii) Context of the child thread running kernel code, and (iii) Context of the child thread running a user program. The kernel codes run in parent and child contexts is well separated in PintOS. The students should clearly identify the functions used in each of these two contexts.

There is strong *dependency* among the three contexts. Thus, the three contexts run in three phases one after another. This may cause some confusion to the naïve developers that the students doing this exercise obviously are. If a student focuses exclusively on the phases, they may not understand the association with the active thread running the code. Therefore, the synchronization issues may become difficult to comprehend and debug. Students are advised to keep the context of execution clearly in their view while developing the enhancements needed in this exercise.

A big advantage of building understanding of the kernel code around contexts rather than phases is partitioning of the code. Phases do not partition the kernel code in obvious partitions.

Where Does the Chicken Cross the Road?

The three-context view of the kernel code provides a clear separation for writing code to complete this exercise. However, we have not yet provided guidance to the students to determine the groupings of the functions in PintOS code with each context. Perhaps, smarter students have already figured this out.

User program codes are outside the kernel code. Therefore, none of the code in PintOS kernel is part of the third (or User Program) context. Function `main()` of the user program is called when all needed resources for user program process have been assembled. The creation of the child process is complete and the child thread is ready for invoking function `main()` of the user program near label `done` in function `load()` in file `pintos/src/userprog/process.c`.

This obviously is the place to confirm that we have set the program stack correctly. Carefully determine where exactly you wish to call function `test_stack()` around label `done`.

The location in PintOS code that separates the parent thread's context from the child thread's context is obviously at the location where the child thread is added to `ready_list` and unblocked to run independently. Again, it is not too difficult to locate this in function `thread_create()` in file `threads/thread.c`. To provide further hints one notes that function `thread_create()` is called from function `process_execute()` in file `userprog/process.c`. An argument in the call listed `start_process()` as the function defining the starting point for the newly created child thread. This function is located in file `userprog/process.c`.

You would also notice that because the parent and the child threads are different threads, a tradition call to `start_process()` is not used in function `thread_create()`. If the threads running two functions were the same, then we would have expected a traditional call to function `start_process()` in function `process_execute()`.

It is unnecessary to say that all activities in function `start_process()` and functions called from it occur in the context of the child thread. Thus, the child thread is responsible for loading of the user program and for setting up the initial stack for function `main()`. The thread is also responsible for invoking function `main()` and thus starting the new process. The thread must receive its command – not just the file name – as a parameter to function `start_process()`.

Before the Chicken Came on the Road

The responsibilities of the parent thread include creation of the child thread and passing a single command to the newly created child thread to run the intended user program. A command string is made of the name of a program file containing an executable code and the arguments for the program.

The directive `run` is first noticed as a user command to load and run in kernel function `run_actions()` of file `pintos/src/threads/init.c`. We need to understand the path from `run_action()` to `thread_create()` in file `threads/thread.c`.

Since all these actions occur in the context of a single parent thread, the traditional practices of program reading will be sufficient to trace the activities.

Appendix provides a sequence of activities that you may find helpful in exploring the kernel code.

How we tested our implementation?

Once you have completed the exercise, we still have a small hurdle. Testing of the user program is not possible yet! The reason for this limitation is non-availability of the system calls to write messages on the computer console! Only kernel code can print messages; user programs cannot write on the console yet.

We partially overcome this limitation by calling function `test_stack()` in the kernel code (and not in the user program code). Caution: You may notice some 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.

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

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

The following 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]$
```

Appendix

This appendix describes the story that traces the activities preceding the start of the execution of function `main()` in a user program on *PintOS*.

1. `Make` files provided in the project compiles and links the `PintOS` kernel code and places the ready-to-load image of the kernel in the boot disk: `OS.dsk`.
2. You run programs on this kernel by using command pattern: `pintos arguments`. File `pintos` is a *Perl script* that interprets the command. String `arguments` is copied into the simulated disk `OS.dsk`. These arguments will be read by the kernel when it starts running.
3. Script `pintos` starts running AI-32 simulator *Bochs*. *PintOS* kernel is loaded into this simulated computer.
4. Like any Unix program, *PintOS* kernel also starts its execution from function `main()`. PintOS function `main()` is in file `threads/init.c`.
5. The call that is of interest to understand the creation of a user thread is call to function `run_actions(argv)` in file `threads/init.c`. Parameter `argv` carries the command line arguments we wrote to the script `pintos` in step 2.
6. This function in turn calls function `run_task()` in file `threads/init.c`. Here `argv` is split into individual tasks. A task is either a run of a user programs or an in-built action.
7. From here the call (`run` request) goes to function `process_execute(task)` in file `userprog/process.c`. User program `task` will run as a child thread; the child thread will separate from the parent thread. (More on this in step 9 below)
8. The final function called by the parent thread is function `process_wait()` in file `userprog/process.c`. This function just terminates in the code provided in the initial implementation of PintOS.
9. However, before calling function `process_wait()` the parent thread calls function `process_execute(task)` in file `userprog/process.c`. Which calls function `thread_create(`with 4 arguments`)` in file `threads/thread.c`. This call creates a child thread and obliges the child to load the task code.
10. A child thread is created and allowed to run by calling `thread_unblock()`. This unblock is the formal point at which child and parent threads become separate entities and receive full access to processor time through PintOS scheduler.
11. The child thread begins its life at the start of function `start_process()` in file `userprog/process.c`. This function will load the user program and setup the stack for the call to function `main()` in user program.
12. Child thread morphs into a user thread (or process) as it invokes and starts executing function `main()` of the user program.

Contributing Authors:

Vishv Malhotra, Gautam Barua, Rashmi Dutta Baruah