Lab 0 - Pintos: Introduction and Installation on Linux.

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. In the Pintos projects, you and your project team will strengthen its support by adding functionality such as synchronization of threads and sleep.

Pintos could, theoretically, run on a regular IBM-compatible PC. Unfortunately, it is impractical to supply every student a dedicated PC for use with Pintos. Therefore, we will run Pintos projects in a system simulator, that is, a program that simulates an 80*x*86 CPU and its peripheral devices accurately enough that unmodified operating systems and software can run under it.

In class we will use the Bochs simulator. Pintos has also been tested with VMware Player.

These projects are hard and have a reputation of taking a lot of time, and deservedly so. We do what we can to reduce the workload, such as providing a lot of support material, but there is plenty of hard work that needs to be done.

Lab 0 explains how to get started working with Pintos. You should read entirely before you start work on any of the projects.

Getting started.

1. Log into a machine that Pintos can be built on. (ED4220 Lab or for remote access STUDAT Linux machines i.e. remote11.chalmers.se, remote12.chalmers.se).
   ```
   ssh -X <cid>@remote11.chalmers.se
   ```
2. Download and unpack pintos.tar.gz into your home directory
   ```
   tar xfz pintos.tar.gz -C ~/
   ```
3. Adding our binaries directory to your PATH environment.
   ```
   export PATH="/chalmers/sw/unsup64/phc/b/pkg/bochs-2.6.6/bin:$HOME/pintos/src/utils:$PATH"
   ```
   Under bash, the standard login shell, you can add the following line into your `$HOME/.bashrc` (or create it if it does not exist).

   Do not forget to reload the configuration using:
   ```
   source $HOME/.bashrc
   ```
4. Make sure the binaries in pintos/src/utils/ are executable:
   ```
   chmod +x pintos/src/utils/pintos*
   chmod +x pintos/src/utils/backtrace
   ```
5. The deadline for the **Test on Pintos Overview** is 17/9/2017 - 23:59.

Source Tree Overview.

Let's take a look at what's inside. Here's the directory structure that you should see in `pintos/src`:

`shell/`
A directory containing code skeleton for the Lab 1. You will implement a command shell in this directory.

`threads/`
Source code for the base kernel, which you will modify starting in Lab 2 for both Tasks 1 and 2.

`userprog/`
Source code for the user program loader, which you will not need to modify.

`vm/`
An almost empty directory, which you will not modify for this course.

`filesys/`
Source code for a basic file system. You will use this file system starting with project 4, but you will not modify it until project 6.

`devices/`
Source code for I/O device interfacing: keyboard, timer, disk, etc. You will modify the timer implementation in Lab 2.File `timer.c` for the **Task 1** and `batch-scheduler.c` for **Task 2**. O therwise you should have no need to change this code.

`lib/`
An implementation of a subset of the standard C library. The code in this directory is compiled into both the Pintos kernel and, starting from project 4, user programs that run under it. In both kernel code and user programs, headers in this directory can be included using the `#include <...>` not ation. You should have little need to modify this code.

`lib/kernel/`
Parts of the C library that are included only in the Pintos kernel. This also includes implementations of some data types that you are free to use in your kernel code: bitmaps, doubly linked lists, and hash tables. In the kernel, headers in this directory can be included using the `#include <...>` n otation.

`lib/user/`

Parts of the C library that are included only in Pintos user programs. In user programs, headers in this directory can be included using the `#include <...>` notation.

`tests/`
Tests for each project. You can modify this code if it helps you test your submission, but we will replace it with the originals before we run the tests.

`examples/`
Example user programs for general purpose use. We look at the `shell.c` example in the first part of Lab 1

`misc/`

`utils/`
These files may come in handy if you decide to try working with Pintos on your own machine. Otherwise, you can ignore them.

Compilation.

To compile Pintos for the first time, change your current directory to the `threads*` directory (`cd pintos/src/threads`) and issue the command:

```
make
```

Continue to the `build` directory and start Pintos to verify that it runs:

```
cd build
pintos -- run alarm-multiple
```
or
```
pintos -- -q run alarm-multiple
```

`run` instructs the kernel to run a test and `alarm-multiple` is the test to run. "-q" terminates the simulator on completion of the execution.

If logged in with -X or on one of the lab computers, Bochs opens a new window that represents the simulated machine's display, and a BIOS message briefly flashes. Then Pintos boots and runs the `alarm-multiple` test program, which outputs a few screenfuls of text. When it's done, you can close Bochs by clicking on the "Power" button in the window's top right corner, or rerun the whole process by clicking on the "Reset" button just to its left. The other buttons are not very useful for our purposes.

(If no window appeared at all, then you're probably logged in remotely and X forwarding is not set up correctly. In this case, you can fix your X setup, or you can use the `-v` option to disable X output: `pintos -v -- run alarm-multiple`.)

You should get a window with output similar to the following:

```
bochs -q
========================================================================
                    Bochs x86 Emulator 2.6.6.svn
             Built from SVN snapshot after release 2.6.6
                 Compiled on Oct  2 2014 at 19:49:41
========================================================================
00000000000i[      ] reading configuration from bochsrc.txt
00000000000e[      ] bochsrc.txt:8: 'user_shortcut' will be replaced by
new 'keyboard' option.
00000000000i[      ] installing nogui module as the Bochs GUI
00000000000i[      ] using log file bochsout.txt
PiLo hda1
Loading.......
Kernel command line: run alarm-multiple
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer...  204,600 loops/s.
Boot complete.
Executing 'alarm-multiple':
(alarm-multiple) begin
(alarm-multiple) Creating 5 threads to sleep 7 times each.
(alarm-multiple) Thread 0 sleeps 10 ticks each time,
(alarm-multiple) thread 1 sleeps 20 ticks each time, and so on.
(alarm-multiple) If successful, product of iteration count and
(alarm-multiple) sleep duration will appear in nondescending order.
(alarm-multiple) thread 0: duration=10, iteration=1, product=10
(alarm-multiple) thread 0: duration=10, iteration=2, product=20
(alarm-multiple) thread 1: duration=20, iteration=1, product=20
(alarm-multiple) thread 0: duration=10, iteration=3, product=30
(alarm-multiple) thread 2: duration=30, iteration=1, product=30
(alarm-multiple) thread 0: duration=10, iteration=4, product=40
(alarm-multiple) thread 1: duration=20, iteration=2, product=40
(alarm-multiple) thread 3: duration=40, iteration=1, product=40
(alarm-multiple) thread 0: duration=10, iteration=5, product=50
(alarm-multiple) thread 4: duration=50, iteration=1, product=50
(alarm-multiple) thread 0: duration=10, iteration=6, product=60
(alarm-multiple) thread 1: duration=20, iteration=3, product=60
(alarm-multiple) thread 2: duration=30, iteration=2, product=60
(alarm-multiple) thread 0: duration=10, iteration=7, product=70
(alarm-multiple) thread 3: duration=40, iteration=2, product=80
```

```
(alarm-multiple) thread 1: duration=20, iteration=4, product=80
(alarm-multiple) thread 2: duration=30, iteration=3, product=90
(alarm-multiple) thread 4: duration=50, iteration=2, product=100
(alarm-multiple) thread 1: duration=20, iteration=5, product=100
(alarm-multiple) thread 3: duration=40, iteration=3, product=120
(alarm-multiple) thread 1: duration=20, iteration=6, product=120
(alarm-multiple) thread 2: duration=30, iteration=4, product=120
(alarm-multiple) thread 1: duration=20, iteration=7, product=140
(alarm-multiple) thread 2: duration=30, iteration=5, product=150
(alarm-multiple) thread 4: duration=50, iteration=3, product=150
(alarm-multiple) thread 3: duration=40, iteration=4, product=160
(alarm-multiple) thread 2: duration=30, iteration=6, product=180
(alarm-multiple) thread 3: duration=40, iteration=5, product=200
(alarm-multiple) thread 4: duration=50, iteration=4, product=200
(alarm-multiple) thread 2: duration=30, iteration=7, product=210
(alarm-multiple) thread 3: duration=40, iteration=6, product=240
(alarm-multiple) thread 4: duration=50, iteration=5, product=250
(alarm-multiple) thread 3: duration=40, iteration=7, product=280
(alarm-multiple) thread 4: duration=50, iteration=6, product=300
(alarm-multiple) thread 4: duration=50, iteration=7, product=350
(alarm-multiple) end
Execution of 'alarm-multiple' complete.
```

The `pintos` program offers several options for configuring the simulator or the virtual hardware. If you specify any options, they must precede the commands passed to the Pintos kernel and be separated from them by `--`, so that the whole command looks like `pintos` *`option`*`... --` *`argument`*`....` Invoke `pintos` without any arguments to see a list of available options.

The Pintos kernel has commands and options other than `run`. These are not very interesting for now, but you can see a list of them using `-h`, e.g. `pintos -h`.

Debugging using bactrace.

The backtrace tool is very convenient to find out where the pintos kernel panics. When a kernel panic happens pintos will print out the addresses call stack of the functions that were running when the panic happend. However, from a humans perspecitve the adresses will not be of much help. The `backtrace` command can help translate addresses to human understandable form. For example if we run the following command (without having a disk file):

```
pintos -- ls
```

Pintos will panic and output similar to the following will show up:

```
bochs -q
========================================================================
                        Bochs x86 Emulator 2.6.6.svn
```

```
                Built from SVN snapshot after release 2.6.6
                    Compiled on Oct  2 2014 at 19:49:41
========================================================================
00000000000i[      ] reading configuration from bochsrc.txt
00000000000e[      ] bochsrc.txt:8: 'user_shortcut' will be replaced by
new 'keyboard' option.
00000000000i[      ] installing nogui module as the Bochs GUI
00000000000i[      ] using log file bochsout.txt
PiLo hda1
Loading.......
Kernel command line: ls
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer...  204,600 loops/s.
Boot complete.
Kernel PANIC at ../../threads/init.c:330 in run_actions(): unknown action
`ls' (use -h for help)
Call stack: 0xc0028309 0xc00206da.
The `backtrace' program can make call stacks useful.
Read "Backtraces" in the "Debugging Tools" chapter
of the Pintos documentation for more information.
```

If we run backtrace with the call stack address:

```
backtrace 0xc00206da 0xc00206da
```

A hint where the crash might have happen will show.

```
0xc0028309: debug_panic (.../../lib/kernel/debug.c:38)

0xc00206da: run_actions (.../../../threads/init.c:330)
```

Be aware that the line number might be a bit inaccurate.

Debugging with GDB.

You can run Pintos under the supervision of the GDB debugger.

1. Change to the `pintos/threads/build` directory.
2. Start Pintos with the `--gdb` option e.g. `pintos --gdb -- run mytest`
   Something similar will show up:

   ```
   00000000000i[      ] using log file bochsout.txt
   Waiting for gdb connection on port 1234
   ```

   * pintos will halt
3. open a second terminal on the same machine and use `pintos-gdb` to invoke GDB on `ker`

nel.o

```
pintos-gdb kernel.o
```

4. Issue the following GDB command:

```
target remote localhost:1234
```

5. Now GDB is connected to the simulator over a local network connection. You can now issue any normal GDB commands. If you issue the `c` command, the simulated BIOS will take control, load Pintos, and then Pintos will run in the usual way. You can pause the process at any point with `Ctrl+C`.
6. Go back to the first terminal and see that pintos now will continue executing!

Debugging versus Testing

When you're debugging code, it's useful to be able to run a program twice and have it do exactly the same thing. On second and later runs, you can make new observations without having to discard or verify your old observations. This property is called "reproducibility." One of the simulators that Pintos supports, Bochs, can be set up for reproducibility, and that's the way that `pintos` invokes it by default.

Of course, a simulation can only be reproducible from one run to the next if its input is the same each time. For simulating an entire computer, as we do, this means that every part of the computer must be the same. For example, you must use the same command-line argument, the same disks, the same version of Bochs, and you must not hit any keys on the keyboard (because you could not be sure to hit them at exactly the same point each time) during the runs.

While reproducibility is useful for debugging, it is a problem for testing thread synchronization, an important part of most of the projects. In particular, when Bochs is set up for reproducibility, timer interrupts will come at perfectly reproducible points, and therefore so will thread switches. That means that running the same test several times doesn't give you any greater confidence in your code's correctness than does running it only once.

So, to make your code easier to test, a feature, called "jitter," was added to Bochs, that makes timer interrupts come at random intervals, but in a perfectly predictable way. In particular, if you invoke `pintos` with the option `-j` *`seed`*, timer interrupts will come at irregularly spaced intervals. Within a single *seed* value, execution will still be reproducible, but timer behavior will change as *se ed* is varied. Thus, for the highest degree of confidence you should test your code with many seed values.

On the other hand, when Bochs runs in reproducible mode, timings are not realistic, meaning that a "one-second" delay may be much shorter or even much longer than one second. You can invoke `pi ntos` with a different option, `-r`, to set up Bochs for realistic timings, in which a one-second delay should take approximately one second of real time. Simulation in real-time mode is not reproducible, and options `-j` and `-r` are mutually exclusive.

TODO: User Programs.

Look at the main function in the shell program under pintos/src/examples/shell.c. This is a simple impementation of a command shell, in Lab 1 you will have a chance to implement a much better version, and thus it is useful to understand the limitations of the current version.

Acknowledgements.

The Pintos core and this documentation were originally written by Ben Pfa blp@cs.stanford.edu.

Additional features were contributed by Anthony Romano chz@vt.edu.

The GDB macros supplied with Pintos were written by Godmar Back gback@cs.vt.edu, and their documentation is adapted from his work.

The original structure and form of Pintos was inspired by the Nachos instructional operating system from the University of California, Berkeley ([Christopher]).

The Pintos projects and documentation originated with those designed for Nachos by current and former CS 140 teaching assistants at Stanford University, including at least Yu Ping, Greg Hutchins, Kelly Shaw, Paul Twohey, Sameer Qureshi, and John Rector.

Example code for monitors (see Section A.3.4 [Monitors]) is from classroom slides originally by Dawson Engler and updated by Mendel Rosenblum.

The current version has been edited and adapted to the requirements of the EDA093/DIT401 Operating Systems course of Chalmers University of Technology by Yiannis Nikolakopoulos and Ivan Walulya, in collaboration with Bhavisya Goel, Vincenzo Gulisano and Marina Papatrianta lou.