# CIS 520 - Fall 2013 - Pintos Project #0 – Warm Up

(Adapted from the Pintos Project Document of Ben Pfaff)

This project will give you have a chance to look around, understand the design structure of Pintos, test your Pintos installation, and add a new test program.

**DUE:** No later than 11:59 pm on Friday, September 6th, 2013

**SUBMIT:** Upload via K-State Online in a single gzipped tar file, proj0.tgz.

- Use the command **`tar czvf proj0.tgz pintos`** to create a compressed tape archive **`proj0.tgz`**; more details below.

## 1. Installation of Pintos

From the CIS department Unix systems (e.g., lab machines in N128), a copy of Pintos can be downloaded from: **/pub/CIS520/pintos.tgz**.

This section explains how to install a Pintos development environment on your own machine. The Pintos development environment is targeted for Unix-like systems. It has been most extensively tested on GNU/Linux, in particular the Debian and Ubuntu distributions, and on Solaris. It is not designed to be installed directly under any form of Windows (but you could use Cygwin or virtualization; e.g., VMware, etc.). The prerequisites for installing a Pintos development environment include the following, on top of standard Unix utilities:

- Required: GCC. Version 4.0 or later is preferred. Version 3.3 or later should work. If the host machine has an 80x86 processor, then GCC should be available as gcc; otherwise, an 80x86 cross-compiler should be available as `i386-elf-gcc`. A sample set of commands for installing GCC 3.3.6 as a cross-compiler are included in "`src/misc/gcc-3.3.6-cross-howto`". The lab machines have Version 4.6.3 installed; e.g., try gcc --version.

- Required: GNU binutils. Pintos uses addr2line, ar, ld, objcopy, and ranlib. If the host machine is not an 80x86, versions targeting 80x86 should be available with an "`i386-elf-`" prefix.

- Required: Perl. Version 5.8.0 or later is preferred. Version 5.6.1 or later should work.

- Required: GNU make, version 3.80 or later.

- Recommended: QEMU version 0.8.0 or later. Bochs version 2.4.5 can also be used. Some other authors reported slowness on old machines, which can be frustrating. Both are installed in the Linux lab in N128. Most students preferred using Bochs last year. Bochs is available in the public directory as bochs-2.4.5.tar.gz.

- Recommended: GDB is helpful in debugging. If the host machine is not an 80x86, a version of GDB targeting 80x86 should be available as "`i386-elf-gdb`".

- Recommended: X11 server. Being able to use an X server makes the virtual machine feel more like a physical machine, but it is not strictly necessary. **Xming** (an X11 server) is installed on the Windows lab machines in N122/N126. It can also be freely downloaded to use on your own machine. The latest stable version is posted in the public CIS520 tools directory as /pub/CIS520/tools/Xming-6-9-0-31-setup.exe. This is a Windows installer.

- Optional: Texinfo, version 4.5 or later. Texinfo is required to build the PDF version of the documentation.
- Optional: TeX. Also required to build the PDF version of the documentation.
- Optional: VMware Player. This is a third platform that can also be used to test Pintos. You can turn Pintos into a virtual appliance and run it on top of Windows 7 using VMware or other virtualization software. We'll discuss more about these virtual machines later in the course.
- Required for remote access: **Putty.exe**: SSH client. If you are working remotely, you can log into the Linux boxes using putty, and work from anywhere. Occasionally, you may want to start up multiple sessions so that you can kill processes that "hang" forever in another session. In this case, you can see all processes that you are responsible for using the command: **ps –u <login name>**; e.g., ps –u neilsen. Then, you can kill processes associated with another session by sending a SIGKILL signal (signal number 9 cannot be ignored); e.g., to kill process with pid 1234, use the command: **kill -9 1234**. A version is posted in the public tools directory, but you can also download a version from the web.
- Frequently used UNIX commands:
    - **ls** – directory listing (what's in the current directory, ~ = home directory)
        - **ls /pub/CIS520**
        - **ls ~/cis520**
    - **pwd** – present working directory (which directory are we in)
    - **cd** – change directory (cd by itself will return us to our home directory**, cd -** to return to the directory that we were just in)
    - **tar** – expand or create a tape archive (just a sequential copy of the files in a folder)
        - **c** – compress
        - **x** – expand
        - **v** – verbose
        - **f** – files
        - **z** – use gzip to compress or expand
    - **ps** – process status – find out which processes are running or are zombies
        - **ps -u <login name>** - list all processes that user "login name" is owner
    - **grep** – gnu regular expression parser
        - **grep -r alarm-multiple *** - recursively find all files containing "alarm-multiple"
    - **which** – determine which version of an executable is being used – order of search is determined by PATH environment variable
        - **which bochs** – which version of bochs are we using,

## 1.1 Getting Started

Now you can extract the source for Pintos into a directory named "`~/cis520/pintos/src`", by fetching the Pintos package (gzipped tape archive) and executing the following commands:

```
cd
mkdir cis520
cd cis520
cp /pub/CIS520/pintos.tgz .
tar xvzf pintos.tgz
```

Let's take a look at what's inside. Here's the directory structure under `"pintos/src"`:

`"threads/"`
Source code for the base kernel, which you will modify starting in project 1.

`"userprog/"`
Source code for the user program loader, which you will modify starting with project 2.

`"vm/"`
An almost empty directory. You will implement virtual memory here in project 3.

`"filesys/"`
Source code for a basic file system. You will use this file system starting with project 2, but you will not modify it until project 3.

`"devices/"`
Source code for I/O device interfacing: keyboard, timer, disk, etc. You will modify the timer implementation in project 1. Otherwise 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 project2, user programs that run under it. In both kernel code and user programs, headers in this directory can be included using the `#include` < ... > notation. 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 < ... > notation.

`"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 use starting with project 2.

`"misc/"`
`"utils/"`
These files may come in handy if you decide to try working with Pintos away from the lab machines. Otherwise, you can ignore them.

## 1.2 Building Pintos

For the next step, build the source code supplied for the first project. First, cd into the "`threads`" directory. Then, issue the "`make`" command. This will create a "`build`" directory under "`threads`", populate it with a "`Makefile`" and a few subdirectories, and then build the kernel inside. The entire build should take less than 30 seconds.

Following the build, the following are some of the files that are created in the "`build`" directory:

"`Makefile`" A copy of "`pintos/src/Makefile.build`". It describes how to build the kernel.

"`kernel.o`"
Object file for the entire kernel. This is the result of linking object files compiled from each individual kernel source file into a single object file. It contains debug information, so you can run GDB or backtrace on it.

"`kernel.bin`"
Memory image of the kernel. These are the exact bytes loaded into memory to run the Pintos kernel. To simplify loading, it is always padded out with zero bytes up to an exact multiple of 4 KB in size.

"`loader.bin`"
Memory image for the kernel loader, a small chunk of code written in assembly language that reads the kernel from disk into memory and starts it up. It is exactly 512 bytes long, a size fixed by the PC BIOS.

Subdirectories of "`build`" contain object files("`.o`") and dependency files ("`.d`"), both produced by the compiler. The dependency files tell make which source files need to be recompiled when other source or header files are changed.

Next, we need an x86 processor emulator that can be used to test our new operating system. Bochs or QEMU can be used for this task as outlined in the next section. Bochs versions 2.4.5 and 2.4.6 are posted in the public tools directory. Either is fine, but the instructions are for Bochs 2.4.5, you can just replace 2.4.5 with 2.4.6 to install the newer version, but it has not been extensively tested.

## 1.3 Building Bochs for Pintos

A copy of Bochs can be downloaded online, or from the public directory:

```
  /pub/CIS520/tools/bochs-2.4.5.tar.gz
```

Just jump back to your cis520 sub-directory and copy the gzipped tar archive to it and extract the files in the archive (of course the first command assumes that cis520 is in your home directory):

```
  cd ~/cis520
  cp /pub/CIS520/tools/bochs-2.4.5.tar.gz .
  tar xvzf bochs-2.4.5.tar.gz
```

Upstream Bochs doesn't support timer interrupts that are delivered at random intervals. However, a patch is available to update Bochs. Thus, Bochs should be installed manually for use with Pintos, instead of using the packaged version of Bochs included with an operating system distribution.

This version of Pintos is designed for use with Bochs 2.2.6, however, we're going to use it with Bochs 2.4.5. A number of patches for the 2.2.6 version of Bochs are included in `"src/misc"`. You can ignore these, but an updated version of a jitter patch for version 2.4.5 is included in /pub/CIS520/ (it should be applied).

`"bochs-2.4.5-jitter.patch" -` adds the "jitter" feature, in which timer interrupts are delivered at random intervals (see section 1.1.4 Debugging versus Testing).

To apply the patch jump to the Bochs directory, and issue a patch command:

```
cd bochs-2.4.5
patch -p1 < /pub/CIS520/tools/bochs-2.4.5-jitter.patch
```

Two different Bochs binaries should be installed. One, named simply `bochs`, should have the GDB stub enabled, by passing "`--enable-gdb-stub`" to the Bochs configure script. The other, named `bochs-dbg`, should have the internal debugger enabled, by passing `"--enable-debugger"` to configure. (The pintos script selects a binary based on the options passed to it.) In each case, the `X`, terminal, and `"no GUI"` interfaces should be configured, by passing the arguments `"--with-x --with-x11 --with-term --with-nogui"` to configure. Thus, the commands are (note that the first two lines should be entered as a single line ./configure …):

```
cd ~/cis520
pwd
mkdir -p usr/local
cd bochs-2.4.5
./configure --prefix=/<home directory>/cis520/usr/local  --enable-gdb-stub
--with-x --with-x11 --with-term --with-nogui
make
make install

./configure --enable-debugger --with-x --with-x11 --with-term --with-nogui
make
cp bochs ~/cis520/usr/local/bin/bochs-dbg
```

Note that **<home directory>** should be replaced with the full path to your home directory; e.g., /home/n/neilsen/ for me. Also, note that the second version of bochs is renamed to **bochs-dbg** in the installation bin folder.
Finally, add /<home directory>/cis520/usr/local/bin and /<home directory>/cis520/pintos/src/utils to the beginning of the PATH environment variable (see below), so that the new version of bochs can be found. Hint: use the command: **which bochs** to ensure that the correct version is found.

### 1.4 Install Pintos

Once these prerequisites are available, follow these instructions to install Pintos:

1. Install Bochs, version 2.4.5 (completed in the previous section)

2. Install scripts from `"src/utils"`. Include `"backtrace"`, `"pintos"`, `"pintos-gdb"`, `"pintos-mkdisk"` in the default `PATH`. You can check your `PATH` by using command **`"echo $PATH"`**. It depends on the shell you're using, you can simply add the src/utils directory to your `PATH` using C-shell (edit .cshrc) or Bash-shell (edit .bashrc) in your home directory and restart your shell; e.g., cd ~/ and edit .bashrc.

   For example, in bash, just add the following line to .bashrc:

   ```
   export PATH=/<home directory>/cis520/pintos/src/utils:/<home directory>/cis520/usr/local/bin:$PATH
   ```

   Also, add the environment variable BXSHARE to your .bashrc:

   ```
   export BXSHARE=/<home directory>/cis520/usr/local/share/bochs
   ```

   where /<home directory>/ is replaced with the full path to your login folder. Remember, you can just type the command: **`pwd`** – to get the present working directory to see which **full path** prefix you should use.

3. Copy `"pintos/src/misc/gdb-macros"` to your ~/cis520/usr/local/bin folder. Then, use a text editor to edit the copy of `"pintos-gdb"`, in pintos/src/utils, changing the definition of GDBMACROS to point to where you installed `"gdb-macros"`. Test the installation by running `pintos-gdb` without any arguments. If it does not complain about missing `"gdb-macros"`, it is installed correctly. At the debugger prompt, (gdb), just type the command **quit** to exit.

4. Compile the remaining Pintos utilities by typing **make** in `"src/utils"`. In general, you need to install `"squish-pty"` somewhere in `PATH`. But, since `"src/utils"` is already in your path, you're good to go. To support VMware Player, you also need to install `"squish-unix"`. If your Perl is older than version 5.8.0, also need to install `"setitimer-helper"`; otherwise, it is unneeded.

5. Pintos should now be ready for use.

### 1.5 Running Pintos

We've supplied a program for conveniently running Pintos in a simulator, called `pintos`. In the simplest case, you can invoke `pintos` as `pintos argument...`. Each argument is passed to the Pintos kernel for it to act on.

Try it out. First, cd into the newly created `"pintos/src/threads/"` directory. Then, issue the command **make** to execute the commands in the Makefile which creates a subdirectory build. Then, cd into the newly created `"pintos/src/threads/build/"` directory. Issue the command **`pintos –v -- run alarm-multiple`**, which passes the arguments **`run alarm-multiple`** to the Pintos kernel. In these arguments, **`run`** instructs the kernel to run a test and **`alarm-multiple`** is the test to run. Note: You may need to type <ctrl-c> to interrupt the processing by sending a SIGINT signal to bochs. If you want to see the emulated machine, use the command **pintos run alarm-multiple**. In this case, if you are running the command remotely, say from N126, you will need to start up an X server; e.g., Xming, before issuing the command, otherwise you will get an error – unable to connect to X server. Finally, bochs doesn't work well with Unity 3d, if it causes the system to freeze, you can kill the offending bochs process using <ctrl>-alt F1. Then, select Unity 2d when logging into a lab machine in N128. Alternatively, you can run under Unity 3d using the command-line version of pintos; e.g., using **`pintos –v -- run ..`**

The pintos command creates a `"bochsrc.txt"` file, which is needed for running bochs, and then invokes bochs. Bochs opens a new window that represents the simulated machine's display (if the -v -- is not used), and a BIOS message briefly flashes. Then, Pintos boots and runs the **alarm-multiple** test program, which outputs a few screens full 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, and you just got a terminal full of corrupt-looking text, 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 X11 output:

```
      pintos -v -- run alarm-multiple
```

Again, you may need to type <ctrl – c> to terminate Bochs. Also, note that the text printed by Pintos inside Bochs probably went by too quickly to read. However, you've probably noticed that the same text was displayed in the terminal you used to run pintos. This is because Pintos sends all output both to the VGA display and to the first serial port, and by default the serial port is connected to Bochs's **`stdout`**. You can log serial output to a file by redirecting at the command line, e.g.

```
      pintos –v -- run alarm-multiple > log-multiple.txt
```

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. As we have seen, options can select a simulator to use: the default is Bochs, but `"--qemu"` selects QEMU. You can run the simulator with a debugger. You can set the amount of memory to give the VM. Finally, you can select how you want VM output to be displayed: use `"-v"` to turn off the VGA display, `"-t"` to use your terminal window as the VGA display instead of opening a new window (Bochs only), or`"-s"` to suppress serial input from `stdin` and output to `stdout`.
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`.

## 2. Writing a new test program

For this project, you need to write a small test program to verify your installation is successful. The program only needs to generate more alarms (100 instead of 7). In order to know how to add a new program into Pintos, you may refer to some existing programs in `pintos/src/tests/threads`. As before, jump back to the `pintos/src/threads` directory and build the system again after making changes to the test suite to add a new test called "**alarm-mega**". You need to run the new test and generate a second log file showing that the test works; e.g., **`pintos run alarm-mega > log-mega.txt.`** In this case, the number of alarms generated should be 100 instead of 7. Hint: you can use the command **grep –r alarm-multiple *** in the **pintos/src/tests** folder to see what changes are needed. Even though you are building the code from the pintos/src/threads folder, files in several other directories are compiled as well.

## Grading

This assignment will count for 50 points. The points break down as follows:

- 20 points: Include a logfile that results from the Pintos execution with alarm-multiple, in the src/threads folder, called **log-multiple.txt**.
- 30 points: For the new test program, include the following in the src/threads folder of your modified project submission (see above to create proj0.tgz to be uploaded):
    - Output sent to a logfile, called **log-mega.txt**, of your test program output.
    - A brief README.txt file explaining changes required to incorporate the above test. Include a description of the files changed or added to the src/threads folder.

**What to Submit:** Upload via K-State Online in a single gzipped tar file, **proj0.tgz**.

- Jump to the top-level folder, cd ~/cis520.

- Use the command **`tar czvf proj0.tgz pintos`** to create a compressed tape archive **`proj0.tgz`**; this file is a sequential compilation of all files in the pintos subfolder. In particular, “c = compress”, “z = using gzip”, “v = verbose – tell me all about it”, “f = files”. If you want to check the tgz file, you could create a temporary folder, say tmp; e.g., **mkdir tmp**, copy proj0.tgz to the tmp folder (**cp proj0.tgz tmp**), jump to the tmp folder (**cd tmp**), and expand the files in the gzipped tar file (**tar xvzf proj0.tgz**). This will create a subfolder called pintos under tmp. Next, check to see if the newly created folder is the same as the original; e.g., **diff –r pintos ../pintos**; that is, “-r = recursively” look for differences “diff” between the newly created files in “pintos” with the original files in “../pintos”. There should be no differences. Just for fun, you can change something in the tmp folder, and check to see that diff catches the change.