GDB / Pintos mini-tutorial

Written by Kevin Christopher

Pintos runs under the Bochs x86 emulator by default; one significant benefit of Bochs is the ability to run the simulated kernel under a full debugger (like gdb). This brief tutorial will demonstrate the basics of setting up Pintos to run under a debugger. The commands here assume that you have added the appropriate /usr/class/cs140/???/bin directory to your path (using `uname -m` in place of ??? selects this path automatically).

Text in bold is input that you should type.
Text in italics is output you should see on the screen.

Debuggers

You may use whichever debugger you wish; this is not a class about debuggers, though you may find a debugger helpful while completing the projects.

Pintos compiles with gcc, which limits the choice of debuggers to those that understand gcc's object format. gdb is the most popular choice, and you probably already know how to use it. I personally prefer ddd, which is a graphical frontend to gdb (and has a gdb session inside a window within ddd's interface); I find it easier to view source code within ddd.

Starting Pintos

Change to the directory from which you intend to run Pintos. For this tutorial, I will be using pintos/src/threads/build (the build directory for the threads project), but you may use any other build directory (userprog, vm, or filesys) just as well.

```
cd pintos/src/threads/build
```

From here, you will need two terminal windows, one for Pintos and one for the debugger. I usually start an xterm with the command “xterm &”, which opens up a second window (assuming your x server is set up correctly). If you don't have an x server running (ie. you are SSHing to Sweet Hall and do not have a local x server or VNC running), you will need to start a second SSH session to the same machine.

In one window, start Pintos. Here is a command line, this one starts Bochs running in a separate window running the “alarm-multiple” test. (If port 1234 does not work for you, pintos has command-line options to change the port ... run “pintos --help” for more information). (If you don't have an x server running, you will need to put the “-v” switch before the “--gdb” switch to prevent Bochs from starting a separate terminal window). Note that both dashes are double-dashes, and that there is a space before run.

```
pintos --gdb -- run alarm-multiple
```

You should get a few lines of text about Bochs starting (Bochs is the default environment for Pintos), then a line:

```
Waiting for gdb connection on localhost:1234
```

Starting GDB

Now, switch to the other window to start gdb. If you are running on a Linux machine (vine, raptor, or firebird in Sweet Hall), then the command "gdb" goes to the right executable; if you are on a Sparc machine (elaine or saga), you will need to use "i386-elf-gdb" throughout this example. Run:

```
gdb kernel.o
```

kernel.o is the object file for the kernel. Once you are in gdb (or your favorite debugger), establish the remote connection:

```
target remote localhost:1234
```

You should see this output:

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

Debugging the kernel

The current program counter, 0x0000fff0, is the default BIOS entry point – all x86 computers start execution here. Stepping through the BIOS is uninteresting, so set a kernel breakpoint further along and run to it... let's set one at timer_interrupt(). (Another good place might be main() in init.c, where the important kernel routines start)

```
break timer_interrupt
```

Bochs has paused the emulator waiting for the connection, so now we must start the emulator running.

```
continue
```

```
Breakpoint 1 at 0xc0103ea0: file ../../devices/timer.c, line
132.
```

The emulator runs for a while, then returns control to you at the breakpoint. From here on out, you are in a (mostly) standard GDB session. Note that if the kernel crashes (triple-faults), you won't regain control in GDB. You will have to restart both Bochs and GDB if the connection fails for any reason, like a triple-fault.

Useful debugging knowledge

Pintos loads itself starting at memory address 0xc0000000. Any kernel code or global data structures will be in a region of memory starting at that address. malloc'ed memory will also be after that address. While you are inside the kernel, you should never see an address below 0xc0000000. Lower addresses are for user programs, and there are more details about the interaction between kernel and user memory that we will discuss later in lecture.

The Pintos memory allocator sets uninitialized memory to 0xcccccccc to help you identify uninitialized and freed memory. When you see 0xcccccccc, it means you either didn't initialize something or you forgot to free it.

The Bochs emulator really does pause the entire system when the debugger has control. This means any input (e.g. keystrokes) to the emulator will be ignored. For this assignment you won't be doing any inputs, but you will in the future. Note that real operating systems aren't this generous; pausing the kernel inside a debugger drops hardware interrupts and is dangerous on an actual system. All modern operating systems do have a serial-console debugging interface (that's what you are using now); most are not as feature-rich as the Pintos/Bochs combination.