HW1: Welcome to Pintos

Andrew Quinn

Due: *Updated* Wednesday April 10n^{th}, 2024 at 11:59 PM.
Learning Objectives:
1. Get comfortable navigating the PintOS source code and using the container toolchain for this course.
2. Setup, build, and execute PintOS.
3. Learn about how computer systems are loaded (i.e., how do we make the whole thing start?).

GETTING STARTED

We have created git repositories for each of you under the path `https://git.ucsc.edu/cse-134/spring24/section01/<yourcruzid>`. If you do not see a git repository at that location, *please let us know immediately!*. You will use this repo for all of the code and design documents throughout this course. Your repository is a private fork of `https://git.ucsc.edu/aquinn1/pintos-reference.git`. We'll be updating this upstream repo to distribute any changes that we have to make to base pintos. If that does not make any sense to you—don't worry, we'll walk you through everything in due time.

SETTING UP PINTOS

The first thing that we need to do is setup the PintOS toolchain. PintOS is designed for 32-bit 80x86 systems. Since you probably do not have a 32-bit x86 PC that you want to dedicate to pintos development, we'll be using simulators (QEMU and BOCHS) instead. To make your life easier, we've put together a Dockerfile for you to build and install our pintos simulators on your local machine. To start, make sure that you have docker installed on your local machine. If you're using a laptop with a graphical display, I suggest using Docker Desktop. To check if you have docker installed correctly, try running the following, it should give you sensible output:

```
docker --version
```

Once you have docker desktop setup, you should build the docker image and name it `cse134`. There are two options.

First, you can build the container directly from the reference git repo. Note that some students on macOS have seen permission issues when doing this:

```
docker build https://git.ucsc.edu/aquinn1/pintos-reference.git -t cse134
```

Second, you can first clone the reference git repo and then build from within that repo:

```
git clone https://git.ucsc.edu/aquinn1/pintos-reference.git
cd pintos-reference
docker build . -t cse134
```

In either case, this command will take a while to complete. It took 493 seconds on my macbook (8-core Apple M1 @ 1.2 GHz, 16GB RAM) and 604 seconds on my desktop (20-core Intel i9–10900T CPU @ 1.90GHz, 32GB RAM).

Once this is built, download your git repository to somewhere on your local machine (not in the docker container). Execute the following, replacing <your cruzid> with, well, your cruzid:

```
git clone git@git.ucsc.edu/cse-134/spring24/section1/<your cruzid>
```

running the PintOS container

If you completed the setup, you are ready to run the container! Use following docker command from a terminal to startup your PintOS container . Note: you should run this command as a single line.

```
docker run -it --rm --name pintos --mount\
  type=bind,source=<path_to_repo>,target=/home/cse134/pintos cse134 bash
```

Let's elaborate on the interesting tid-bits:

1. `docker run -it --rm --name pintos` creates and runs a new container, named `pintos`, that will be removed as soon as the current terminal ends. While you could run the container without `--rm` or `-it`, we find these to be useful settings for development.
2. `--mount type=bind,source=<path\_to\_repo>,target=/home/cse134/pintos` mounts your repository as the directory `/home/cse134/pintos` on the container. Any changes that you make outside the container will be reflected inside of it and vice versa. This is super useful for development, because you can edit the code in your favorite IDE/editor on your local machine.
3. `cse134 bash` says that you want to use the docker image that you created under setup above and run the command `bash`, which is just a standard shell.

You may later find that you want to attach another terminal to the container that you just created. The following executes the command `bash` on the container named `pintos`:

```
docker exec -it pintos bash
```

running PintOS

Great! You've got the container running. Now, let's try actually running PintOS inside of a simulator. Navigate to `/home/cse134/pintos/src/threads` and build the system (specifying `-j` tells `make` to use multiple cores):

```
make -j
```

When you execute the `make` command, PintOS creates a new folder, `/home/cse134/pintos/threads/build`, that contains the new image. Navigate into that folder and use one of the helper scripts, `pintos`, to start the system with a particular workload. If you have build everything correctly, you should see some encouraging output:

```
cd build
pintos -- run alarm-single
```

A BRIEF INTRODUCTION TO PINTOS

This section gives you a brief overview in the PintOS source code tree, the files created when you build pintos, and the various parameters that you can pass to run the system. This section is not *totally* necessary for this homework assignment, but its a good resource. My description is based heavily off of the pintos documentation. I purposefully left some things out (e.g., discussion of design documents), but you might find it worth looking through the original documentation anyway.

PINTOS SOURCE TREE

The source code for PintOS is located under the `src` directory. You'll be making all of your changes to the system by modifying or creating files under the directory. Here's brief description of each of the subdirectories:

- "`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 4.
- "`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 project 2, 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/`" and "`utils/`": These files may come in handy if you decide to try working with PintOS on your own machine. Otherwise, you can ignore them.

PintOS build output

Your implementation for each of the PintOS projects will take place in a separate directory: project 1 will be in "`threads/`", project 2 in "`userprog/`", project 3 in "`vm/`", and project 4 in "`filesys/`". PintOS provides a makefile for each of these directories, so that building your implementation for a project involves navigating to the correct directory and executing `make`. Executing `make` in any of these folders creates a subdirectories, `build` that includes the following:

- "`Makefile`" A copy of "`pintos/src/Makefile.build`" describing how to build the kernel.
- "`kernel.o`" An 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 (see section E.5 GDB) or backtrace (see section E.4 Backtraces) on it.
- "`kernel.bin`" Memory image of the kernel, that is, the exact bytes loaded into memory to run the PintOS kernel. This is just "`kernel.o`" with debug information stripped out, which saves a lot of space, which in turn keeps the kernel from bumping up against a 512 KiB size limit imposed by the kernel loader's design.
- "`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.

Additionally, `make` creates subdirectories in `build` with object files ("`.o`") and dependency files ("`.d`"). The dependency files tell make which source files need to be recompiled when other source or header files are changed.

running PintOS

PintOS provides a program, `pintos`, that makes it easy to run PintOS in a simulator; we used this command earlier to test that you installed and built the system correctly. The command for using the script is

```
pintos option... -- argument...
```

where `option...` refers to options that configure the simulator and `argument...` refers to arguments that you pass to the kernel. You will usually pass arguments as `run <test>` to run the program `test` after PintOS loads. Options can select a simulator to use: the default

is QEMU, but you can also run with the Bochs simulator by specifying "–bochs". You can run the simulator with a debugger (see below). You can set the amount of memory to give the VM. The Pintos kernel has a whole host of other commands and options which you can see by using `-h`, as in: `pintos -h`.

Debugging PintOS

DEBUGGING NONDETERMINISM. Contending with nondeterminism is very challenging, as I am sure you recall from CSE 130. The especially challenging bugs are so called *heisenbugs*—nondeterministic bugs whose behavior goes away when you add *print* statements to your code. In other words: bugs whose behavior changes due to observation.

To make your life easier, PintOS has extended the Bochs simulator with a reproducibility feature. To use this feature, use the Bochs simulator through the `pintos` program, specify the same command-line argument, the same disks, and *do 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. Bochs will ensure that timer interrupts come at perfectly reproducible points, and therefore so will thread switches.

USING GDB. In addition to using Bochs for debugging, PintOS also provides tooling that you can use to run `gdb` on it. You will probably find this feature useful for this homework assignment. See this link for a detailed rundown of its use (there are a number of extra commands that PintOS provides). As a short tutorial, you can execute the following:

```
pintos --gdb -- run mytest
```

Then, create a second terminal window:

```
docker exec -it pintos bash
```

Navigate to the current build directory (e.g., `pintos/src/threads/build`) and execute the following:

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

Finally, once the `gdb` starts, execute the following to attach to your running PintOS instanced:

```
debugpintos
```

Testing PintOS

While reproducibility helps with debugging, it is detrimental for testing. The issue is 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. How can you figure out if your PintOS changes actually handle nondeterminstic inputs?

To make this possible, PintOS adds a feature, called jitter, to Bochs. Jitter makes timer interrupts predictably random: they arrive at random intervals, but at deterministic intervals

based upon a `seed`. To use jitter, invoke `pintos` with the option `-j seed`. For the highest degree of confidence you should test your code with *all possible* seeds, or, at least a bunch of different ones.

However, the issue with running Bochs in reproducible mode is that timings are not realistic. So, a "one-second" delay may be much shorter or longer than one second. To get realistic timings, you can invoke pintos with "realistic bochs" by specifying the options `--bochs -r` [1]. Alternatively, QEMU, the default simulator used by the `pintos` script, supports realistic timings and is much faster than Bochs.

BACKGROUND ON BOOTSTRAPPING

Much of this homework is about how systems load their operating systems into memory on startup, called *bootloading*. The discussion here is focused on how this works on the 80x86 architecture, but the approach has changed surprisingly little on more modern hardware. There are two helper programs that are responsible for making this happen: the BIOS (Basic Input/Output System) and bootloader. Hardware ensures that the BIOS has control of the machine when it is powered on. The BIOS performs basic initialization, e.g., checking memory available and activating video card, and then tries to find a bootloader on peripheral devices such as floppy disks, hard disks, CD-ROMs, etc. To do this, the BIOS looks at the first *sector* of attached devices, where a sector is a small (typically 512 bytes) chunk of a device, to see if it contains a *boot sector*. If it finds a boot sector, the BIOS loads the sector into a specific physical address (on the 80x86 this happens to be at location `0x7c00` to `0x7dfff`) and then passes control of the machine to the bootloader. The bootloader is then responsible for loading the operating system.

The helpers used for bootstrapping are required to use very few system resources. They must take up little space (e.g., bootloaders are typically limited to hundreds of bytes) and cannot use standard libraries like `printf` or `read`. Instead, these helpers must interact with hardware through their complex hardware programming routines using interrupts. To limit code bloat, most of these helpers are written in assembly.

To illustrate these challenges, we will discuss the physical memory layout of typical machine. Early 16-bit systems, such as Intel's 8088 processor, were only capable of addressing 1MB of physical memory. Thus, the physical address space started at `0x00000000` and ended at `0x000FFFFF`. These systems reserved the 384KB area from `0x000A0000` to `0x000FFFFF` for special hardware (e.g., the video display buffers) and the BIOS ROM, which occupied the 64KB region from `0x000F0000` to `0x000FFFFF`.

To ensure backwards compatibility, architectures retained these reservations even after address spaces expanded beyond 1MB. Thus, the 80x86, and most modern systems, have a "hole" in their physical memory from `0x000A0000` to `0x00100000` reserved for the BIOS and 16-bit devices. This divides the physical address space of a system into "low" or "conventional" memory (the first 640KB) and "extended" memory (everything else). Additionally, the BIOS often reserves memory at the top of the 32-bit address space 32-bit devices. We show this layout in fig. 1.

[1] Note that Bochs cannot simultaneously support realistic timings and reproducibility

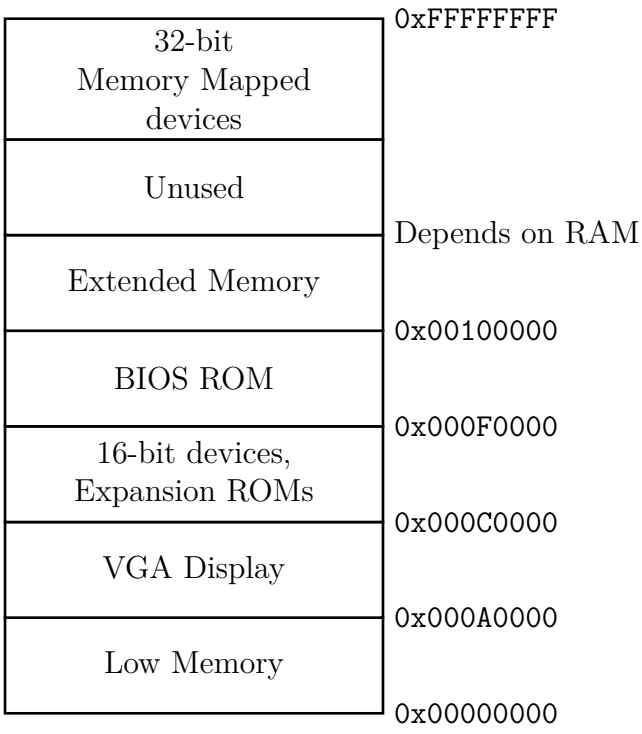

Figure 1: The memory layout of an 80x86 machine.

THE BOOTLOADER

Bootloaders for the 80x86 run in real-addressing or "real" mode, in which the system uses segment registers to compute memory address according to the formula: address= 16 * segment + offset. There are different segments and segment registers for different memory regions. For example, instructions reside in the code segment, whose start is specified by the register CS. So, when the BIOS passes control to the bootloader at address `0x7c000`, it does so by setting CS to 0 and executing a `jmp` instruction to `0x7c00`. The physical address calculation is then computed as $16 * 0 + \texttt{0x7c00} = \texttt{0x7c00}$. Other segments include a stack segment (register SS) and two data segments (registers DS and ES). Each segments are 64KiB in size; since bootloaders often have to load kernels that are larger than 64KiB, they must utilize the segment registers carefully.

The pintos bootloader makes a number of simplifications: it only supports small kernels, always loads the kernel starting at address `0x20000`, only supports booting off of hard-drives, etc.

After loading the kernel, the bootloader transfers control to it. In PintOS, the entry point is start(), in file `src/threads/start.S`. The entry point is not at a predictable location in the kernel image, but the kernel's ELF header contains a pointer to it. The loader extracts the pointer and jumps to the location it points to.

THE KERNEL

The kernel's entry point takes a number of steps to finalize the operating system's startup. It first obtains the machine's memory size from the BIOS. PintOS's BIOS uses a simple function that detects at most 64 MB of RAM, so that is the practical limitation of PintOS. Then, the kernel startup code sets the A20 line, that is, the CPU's address line numbered 20. For historical reasons, PCs boot with this address line fixed at 0, which means that attempts to access memory beyond the first 1 MB (2 raised to the 20^{th} power) will fail. Next, the kernel sets up basic page tables and turns on protected mode—taking the system out of real mode and into the addressing expected on more modern machines. (details omitted for now). Finally, PintOS calls into the C code of the Pintos kernel.

COMMAND LINE ARGUMENTS

Pintos manages command line arguments in an inelegant, but effect manner. The `pintos` program that you use to start the system inserts the command-line arguments into a copy of the boot loader each time it starts the kernel. Then, the kernel reads the arguments from the boot loader's memory when after it starts executing.

YOUR HOMEWORK QUESTIONS

Finally, we've reached the homework questions! Don't worry, it won't usually take this long. For this assignment, you'll practice the turn-in method that we will use for projects throughout the quarter. Namely, put the answers to the homework questions in a file named `docs/hw1.md` in your repository. Make sure to include the question and/or question number for each of your answers. When you are finished, commit your answers to your repository and make sure to push your the changes. Then, navigate to the canvas assignment and enter the 40-character hash as your submission.

You will find that `gdb` is required to answer most of the homework questions. You will also likely find it useful to read the bootloader's code from `src/threads/loader.S` (and its disassembly at `src/threads/build/loader.asm`). The questions are as follows:

1. What is the physical address of the first instruction that is executed by the BIOS? *hint: it is the address of $eip when you start PintOS.*
2. How does the bootloader read disk sectors? In particular, what BIOS interrupt does it use?
3. How does the bootloader decide if it successfully finds the PintOS kernel?
4. What happens if the bootloader is unsuccessful in finding the PintOS kernel?
5. How exactly does the bootloader transfer control to PintOS?
6. Tracing the behavior of the `palloc_get_page()` function from PintOS. Set a breakpoint on this function and identify the following values *the first time* that it is called:

   (a) What is the call stack when the function is called?
   (b) What is the return value of the function, in hexadecimal format, on its first invocation?

7. Tracing the behavior of the `palloc_get_page()` function from PintOS. Set a breakpoint on this function and identify the following values *the third time* that it is called:

   (a) What is the call stack when the function invocation?
   (b) What is the return value of the function, in hexadecimal format, on its first invocation?