# CS 162 Stretch Activity: `Pintos Fun`

## Contents

**Everything in this document is optional.** That said, we **highly recommend** that you try out at least A Shell for Pintos. You've spent a significant amount of time implementing Pintos this term, and hopefully this stretch exercise will give you a chance to have some fun with what you've built.

Before attempting *anything* in this document, you need to have, at minimum, implemented Project 1. To do anything other than Simple Shell, you'll need to have implemented Project 1 and Project 3. If you reach the part of this exercise where you start playing/creating music, it's recommended (but not required) that you have implemented Task 1 (Alarm Clock) of Project 2.

If you choose to work on this exercise after the term ends, you should make a copy of your GitHub repository, because we will delete all group and student repositories on GitHub after the term ends. **Simply making a fork of the repository is not enough** because when a GitHub repository is deleted, forks of that repository also disappear. Either keep your local `git` repository safe as the only copy of your code. Or, create a new ***private*** repository on GitHub, add it as a new remote, and push your local code to that new repository. Remember to also keep a copy of the `group0` repository, as that contains the code you'll need to merge in. Forking `group0` is OK because we will not delete that repository (though we will change the code in it).

## 1 Getting Started

Before beginning this exercise, you'll need to update the `Pintos.pm` and `pintos-set-cmdline` scripts to the latest version. If you're reading this after Fall 2019, then the changes have probably already been made in the class VM distributed at the beginning of the semester.

Run the following commands (make sure to type the long `wget` commands each as a single line):

```
$ cd ~/.bin
$ rm -f Pintos.pm pintos-set-cmdline
$ wget https://raw.githubusercontent.com/Berkeley-CS162/vagrant/master/modules/
cs162/files/shell/bin/Pintos.pm
$ wget https://raw.githubusercontent.com/Berkeley-CS162/vagrant/master/modules/
cs162/files/shell/bin/pintos-set-cmdline
$ chmod +x pintos-set-cmdline
```

Now you're ready to get started on the exercise below.

## 2 A Shell for Pintos

In Homework 2, you implemented a shell for Unix-based systems. Similarly, one can implement a shell for Pintos. This will allow you, the user, to launch processes in Pintos interactively.

Before starting, pull the latest code by running:

```
$ git fetch staff
$ git merge staff/fun-simple-shell
```

This will download the scripts that we will use the exercises below.

### 2.1 Simple Shell

See the `shell.c` program in `src/examples`. It is a simple shell capable of spawning new processes and waiting for them to complete. To run this simple shell, you only need to have implemented Project 1. You do not need to have implemented Project 3, but the shell will be nicer if you have.

First, we will run this shell as a process in Pintos, with Pintos running in QEMU. To do so, navigate to the `src/filesys` directory and run:

```
$ make
$ cd build
$ ../../../utils/bootable-cs162.sh --simple
```

The result is a bootable disk image, `cs162proj.dsk`, which contains both the kernel image and a file system partition. The script builds the Pintos user applications in `src/examples` and copies them into the file system in `cs162proj.dsk`.

Once you have the `cs162proj.dsk` file, you can start the Pintos shell by running:

```
$ pintos -k --qemu --disk cs162proj.dsk -- -q run "shell"
```

Alternatively, if you would like it to run on the command line only, then run:

```
$ pintos -v -k --qemu --disk cs162proj.dsk -- -q run "shell"
```

This boots Pintos off of the bootable disk image `cs162proj.dsk` and runs the `shell` program as the first process. When you type a command, the `shell` program issues an `exec` system call directly, with no path resolution. so the executable name is always interpreted as a relative path. When you are all done, use the `exit` command, which will tell the `shell` to exit. When the first process exits, Pintos shuts down, so this will cause Pintos to shut down. Alternatively, you can run the `halt` program, which will issue a `halt` system call to make Pintos shut down.

Here are some of the commands you can run:

- `cat` - prints out the contents of a file (try it on `file1.txt` and `file2.txt`)
- `cmp` - compares two files (try it on `file1.txt` and `file2.txt`)
- `cp` - copies a file
- `echo` - prints out its command line arguments to the console
- `hex-dump` - prints out a file in hexadecimal
- `insult` - runs the Stanford insult generator
- `ls` - prints the names of all files in the current working directory (requires Project 3)
- `mkdir` - creates a new directory (requires Project 3)
- `pwd` - prints out the current working directory (requires Project 3)
- `rm` - unlinks a file, or a directory if it is empty
- `halt` - issues a `halt` system call, which shuts down the system
- `shell` - spawns a new shell
- `cd` (shell builtin) - changes the current working directory (requires Project 3)
- `exit` (shell builtin) - causes the shell program to exit, which will cause the system to shut down if the shell was the first process

There are also the commands

Notice that whenever a program finishes running, you will see the `exit(0)` message (or some other return value). This is the functionality you implemented in Project 1. Then the shell prints an additional message once its `wait` system call returns to indicate that the program has finished.

If you make any changes to the file system they will persist in the `cs162proj.dsk` file—if you reboot Pintos, you will see the updated file system.

You can find the source code for all of the example programs in `src/examples`. Not all of them are included in the simple shell, since the default root directory (if you're using your Project 1 code) has a cap on how many entries it can contain. Some utilities, like `mcat` and `mcp`, issue system calls that you didn't implement in your projects this term, so unfortunately they will not work.

#### 2.1.1 Troubleshooting

**When I type at the shell prompt ("`--`"), nothing shows up, or garbled characters show up.**

You probably did not correctly implement the `read` system call in Project 1. If a program attempts to `read` from file descriptor `0` (`STDIN_FILENO`), then your implementation of `read` should read from console using `input_getc()`. This functionality was not tested by the Project 1 test suite. You should make sure to handle this case correctly, and then try again.

**It worked the first time I booted from `cs162proj.dsk`, but it fails when I boot again from the same file.**

When you're done using the shell, make sure to shut down the Pintos operating system cleanly by `exit`ing the shell or running the `halt` program, which will cause Pintos to shut down normally. If you just close QEMU (e.g., with Ctrl-C), then the Pintos shutdown procedure will not run, and your buffer cache (from Project 3) will not be flushed. This may leave the on-disk file system in an inconsistent state, meaning that your file system code may not be able to properly interpret the file system the next time it boots. Real file systems have tools, like `fsck`, to repair the file system in this case, but in Project 3 you did not implement such a tool for your file system design.

### 2.2 Advanced Shell

In the simple shell, you may have noticed that the shell does not do path resolution, and that there is no program you can use to create and write to files. This is partially because the interface to file creation in Project 1 is awkward to use: you must specify the length of a file at the time it is created. This is acceptable for copying file into the Pintos file system, but not for creating files within Pintos itself.

Fortunately, you fixed these shortcomings in Project 3. We have provided utilities that take advantage of the functionality you implemented in Project 3. You can obtain by running the following commands:

```
$ git fetch staff
$ git merge staff/fun-shell
```

This will enhance the `shell.c` program with rudimentary path resolution; if you type a command but the executable is not found in the current directory, it also checks in the directory `/bin/`. This is like path resolution with the `PATH` variable hardcoded to `/bin/` (note that Pintos does not support environment variables). Furthermore, the `cd` command, which allows you to change the current working directory by issuing a `chdir` system call, should now work.

We have also provided three new programs:

- `fcreate` - Creates a new file
- `fappend` - Appends a line to a file
- `rmrec` - Deletes a directory recursively (like `rm -r` on Linux)

When running the advanced shell, you can also omit the `--simple` flag to copy the executables to the directory `/bin/` (Project 3 required):

```
$ make
$ cd build
$ ../../../utils/bootable-cs162.sh
```

This will create a file system consisting of a single directory `/bin/` containing the executables in `src/examples`. Because of the rudimentary path resolution in the advanced shell, you can invoke the executables by name even if your current working directory is not `/bin/`. If you do this, you'll need to specify `/bin/shell` as the executable to run as the first process:

```
$ pintos -k --qemu --disk cs162proj.dsk -- -q run "/bin/shell"
```

### 2.3 Question to Think About

If you wanted to implement the `>` operator in the Pintos shell, what changes would you have to make to the Pintos kernel? What about the `|` operator?

## 3 Running Pintos in VMWare

Later in this exercise, you will write code to use the computer's speaker device from Pintos. Unfortunately, our QEMU setup in the VirtualBox VM does not virtualize the speaker hardware well enough to properly play the music. In contrast, VMWare virtualizes the speaker hardware well enough to emulate the sounds that would be played by real speaker hardware.

### 3.1 Obtaining VMWare

Unlike VirtualBox, VMWare's virtualization products are not free. Fortunately, the University of California, Berkeley has already paid VMWare for a license to their software. As a UC Berkeley student, you can obtain VMWare's software for free at software.berkeley.edu[1]. You should download and install VMWare on your **host machine** (i.e., NOT within the class VM). If you are running Windows or Linux, you should install VMWare Workstation; if you are on Mac OS, you should install VMWare Fusion.

### 3.2 Prepare Pintos to run in VMWare

Before Pintos will boot properly in VMWare, **you must comment out or delete the `serial_putc(c);` statement in `src/lib/kernel/console.c`**. The setup for running tests on the project code involves a virtual serial line from the virtual machine running Pintos. All console output is written to this serial line, which allows the testing framework to verify that your implementation is behaving properly, without having to parse the screen image. Because nothing is connected to the other end of the serial line in the VMWare setup, it will just cause the console to block after trying to write too many characters; therefore, you must disable it by commenting out or deleting the above line. **Once you complete this step, all tests will fail, and you will no longer see output on the command line, except for the bootloader's output. The only output will be on the virtual screen, which you can see by running `pintos` without the `-v` flag (see the example in Advanced Shell).**

### 3.3 Prepare a VMWare Disk Image

Follow the instructions in Advanced Shell to invoke the `bootable-cs162.sh` script and obtain the `cs162proj.dsk` file. This is a bootable disk image for your Pintos implementation, including a file system initialized as described in Advanced Shell. You could copy the data in this file to a disk device, and then boot from it. In order to create a VMWare VM that boots from it, you'll need to first convert it into the virtual disk format that VMWare knows how to use.

[1] https://software.berkeley.edu

Copy the `cs162proj.dsk` file out of the class VM. Then run the following command:

```
$ vboxmanage convertfromraw cs162proj.dsk cs162proj.vmdk --format VMDK
```

If you're on Windows, you should instead run (in one line)

```
> "C:\Program Files\Oracle\VirtualBox\VBoxManage.exe" cs162proj.dsk cs162proj.vmdk
   --format VMDK
```

This should produce the file `cs162proj.vmdk`, which is a bootable virtual disk image you can use with VMWare. The `vboxmanage` program ships with VirtualBox. Because you installed VirtualBox to run the class VM, this should be installed for you locally.

## 3.4 Create and Run a Pintos VM in VMWare

The instructions for creating the workstation are different depending on whether you are using VMWare Workstation (Windows/Linux) or VMWare Fusion (Mac OS).

### 3.4.1 Instructions for VMWare Workstation

Open VMWare Workstation. Click `File > New Virtual Machine...` to open the New Virtual Machine Wizard. Specify the configuration of the Pintos VM as described in the following steps:

1. For the type of configuration, choose "Custom (advanced)."
2. For **Hardware Compatibility**, choose the latest available version (15.x at the time of writing).
3. For **Guest Operating System Installation**, select "I will install the operating system later."
4. For **Guest Operating System**, select "Other," and for the version, choose "Other."
5. For **Virtual Machine Name**, choose any name you desire.
6. For **Processors**, set the number of processors to 1 and the number of cores per processor to 1.
7. For **Memory**, set the amount of available memory to 64 MB.
8. For **Network Connection**, choose "Do not use a network connection."
9. For **I/O Controller Types**, choose "BusLogic (Recommended)" for the SCSI controller, which is the default at the time of writing.
10. For **Virtual Disk Type**, choose "IDE."
11. For **Disk**, choose "Use an existing virtual disk."
12. For **Existing Disk File**, choose the `cs162proj.vmdk` file you created in Prepare a VMWare Disk Image.
13. Finally, click "Finish."

### 3.4.2 Instructions for VMWare Fusion

Open VMWare Fusion. On the "Select the Installation Method" menu choose "Create a custom virtual machine."

1. For **Chooose Operating System**, select `Other > Other`.
2. For **Choose Firmware Type**, select "Legacy BIOS."

3. For **Choose a Virtual Disk**, select "Use an existing virtual disk." Choose the `cs162proj.vmdk` file you created in Prepare a VMWare Disk Image. When choosing that file, choose the option "Share this virtual disk with the virtual machine that created it."
4. For **Finish**, you can just click "Finish." If you like, you can click "Customize Settings" to customize the memory and network connection as specified in the VMWare Workstation instructions, but the default settings work fine.

#### 3.4.3 Running the VM

Now, start the VM, and you should see the same shell setup you saw in QEMU in Advanced Shell. The configuration above is one that I have verified works, but other configurations will work too. Feel free to try changing the configuration if you're feeling adventurous.

### 3.5 Updating the VMWare VM

In future parts of this exercise, you may need to update your Pintos implementation and run the new Pintos in VMWare. You don't have to set up a new VMWare VM from scratch. Instead, just power off the VMWare VM, obtain the new `cs162proj.dsk` file, convert it into a new `cs162proj.vmdk` file, and replace the old `cs162proj.vmdk` file with the new one. When you power on the VMWare VM again, it will read the new `cs162proj.vmdk` file and boot the new Pintos kernel with the new file system stored in the virtual disk.

### 3.6 Troubleshooting

**When I try booting Pintos in VMWare, it gets stuck in the boot process or shortly thereafter.**
You probably did not correctly disable the serial functionality by deleting or commenting out the line of code specified in Prepare Pintos to run in VMWare.

If you're running into an issue, and you're sure you have tried the fixes above, you should try downloading and using the staff `cs162proj.dsk` file or `cs162proj.vmdk` file from the course website. This will help you determine whether the issue is caused by your Pintos implementation, or by how you configured the VM in VMWare.

### 3.7 Question to Think About

Could you instead run Pintos on your host machine in a Docker container, without a hardware virtualization layer of any kind?

## 4 Playing Music from a Pintos Application

In this part of the exercise, we will provide user programs with the ability to control the speaker. This will allow us to write and run pintos applications that play songs.

### 4.1 The Speaker Device

The PC Speaker[2] has two positions. By moving the speaker between the positions, it is possible to generate sound. By moving the speaker between those two positions at a certain frequency, one can generate sound at that frequency. The processor can use the `out` instruction to move the speaker from one position to the other.[3]

[2]https://en.wikipedia.org/wiki/PC_speaker

[3]Recall that there are two ways for the processor to interact with I/O peripherals: port-mapped I/O (via I/O instructions, like `in` and `out`) and memory-mapped I/O.

Producing a pure tone would require a sinusoidal wave form. On the surface, this seems incompatible with the speaker having only two positions. However, the speaker takes a small amount of time to switch between positions. It is possible to switch the speaker between the two provided positions very rapidly, without allowing it to settle in either one, effectively holding the speaker at a position *between* the two provided ones. This allows the CPU to drive the speaker directly to approximate arbitrary waveforms, but it requires very precise timing on the CPU. In practice this tends to take a lot of CPU time away from other useful work, especially so in the early days when CPUs were slower than they are now.

To avoid this, it is common to use a *sound card*. The sound card is a dedicated chip, separate from the CPU, that drives the speaker. The CPU can program the sound card and then do other useful work while sound plays in the background—the sound card, not the CPU, drives the speaker. Sound cards also typically provide hardware support for digital signal processing to generate and mix different audio signals to feed to the speaker.

The Pintos speaker driver[4] uses the Programmable Interrupt Timer (PIT) in a way reminiscent of a sound card. Channel 0 of the PIT is used to generate interrupts for scheduling (see `timer_init` in timer driver[5]). Channel 2 of the PIT is configured by the speaker driver to output a square wave to the speaker, where the position of the square wave (high or low) corresponds to the position of the speaker. Controlling the frequency of the square wave adjusts the rate at which the speaker oscillates between its two positions, and consequently allows one to control the frequency of the tone that is played. The square wave is produced by the PIT, without any involvement from the CPU once the PIT and speaker are configured. Thus, the CPU can perform other useful work while the tone is being played; only *changing* the tone requires the CPU's involvement.

Because the signal produced by the PIT is a square wave, it is not a pure tone. This means that, while the primary frequency ("fundamental") will be the one configured on the PIT, there will be additional frequencies ("harmonics") in the output. If you have taken EE 120, then you know that these additional frequencies are odd-numbered multiples of the primary frequency, represented as Dirac deltas in the Fourier transform, with magnitudes diminishing according to the sinc function[6]. For example, a 200 Hz square wave will have additional harmonics at 600 Hz, 1000 Hz, etc.

## 4.2 The `tone` System Call

**For this part of the exercise, it is recommended, but not required, that you have included your Alarm Clock implementation from Project 2.**

The speaker in driver allows one to drive the speaker using the PIT by calling three functions: `speaker_on`, `speaker_off`, and `speaker_beep`. These functions can only be called in the kernel, however. To allow a user program to use the speaker, you need to implement a system call. In principle, we could just wrap each of these functions in a system call and expose them to the user program directly. For this exercise, you should just implement a single system call, which we will call `tone`:

**System Call: void tone (int frequency, unsigned milliseconds)** Plays a tone at the specified frequency for the specified number of milliseconds. The calling process is blocked while the tone is played. If the specified frequency is 0, then no tone is played and the process just waits for the specified duration.

You should implement a `tone` function in `lib/user/syscall.h`, like you did with `sbrk` in Homework 5, so that user programs can issue `tone` system calls. Then, implement the `tone` system call in your Pintos kernel. To make this easier for you, we've provided the following example handler for the `tone` system call:

[4]https://github.com/Berkeley-CS162/group0/blob/master/pintos/src/devices/speaker.c
[5]https://github.com/Berkeley-CS162/group0/blob/master/pintos/src/devices/timer.c
[6]https://dsp.stackexchange.com/questions/34844/why-fourier-series-and-transform-of-a-square-wave-are-different

```
static void syscall_tone(int frequency, unsigned milliseconds) {
    if (frequency != 0) {
        speaker_on(frequency);
    }
    timer_msleep((int64_t) milliseconds);
    speaker_off();
}
```

Incidentally, no other way currently exists in Pintos for a user program to sleep for a certain duration. If you have included your Alarm Clock implementation from Project 2, this allows a process to sleep without busy-waiting. You could also implement a `sleep` system call specifically for this purpose.[7]

## 4.3 The `music.h` Library

Once you have implemented the `tone` system call, you can write Pintos applications that issue `tone` system calls to play music! We've written some examples that you can try out. Run the following commands:

```
$ git fetch staff
$ git merge staff/fun-music
```

You should be able to build the following new programs in `src/examples`: `yankee`, `railroad`, `railroad162`, and `auld`. Repeat the instructions in Advanced Shell to create a fresh disk image containing your updated Pintos kernel, which implements the `tone` system call, and a fresh file system including these new programs in `/bin/`. Then boot a VMWare VM with this disk image, following the instructions in Running Pintos in VMWare. If you run one of the above programs, you should hear a song and see the words printed out. The process will exit once the song is played to completion.

Read the source files for these songs (`yankee.c`, `railroad.c`, `railroad162.c`, and `auld.c`) to see how they work. They make use of a primitive music library that I implemented in `music.h`. Feel free to read these files, understand how they work, and use `music.h` to write songs of your own.

## 4.4 Sound Cards

For a further stretch, you can configure VMWare to provide you a virtual sound card, and then write a driver for it in Pintos. This will allow you to use the sound card's digital signal processing capabilities to produce more complex music. As a start, most sound cards will (1) have multiple *channels*, allowing you to play multiple tones at once, each with a different waveform, and (2) support playback of sampled audio. The sound card will take care of properly mixing the different signals and driving the speaker accordingly according to the synthesized result. I would recommend starting with a relatively simple sound card, like the Sound Blaster 16[8]. I've only managed to detect a virtualized Sound Blaster 16 chip in VirtualBox, but this link[9] says it can be done in VMWare too.

The first two minutes of this video[10] provide an understandable introduction to sound cards, specifically their multi-channel capabilities and FM synthesis. The rest of the video isn't relevant to this exercise, but may be still be interesting to you.

[7]On Linux, the equivalent system call is `nanosleep`.
[8]https://wiki.osdev.org/Sound_Blaster_16
[9]https://communities.vmware.com/docs/DOC-40539
[10]https://youtu.be/q_3d1x2VPxk?t=19

### 4.5 Questions to Think About

What problems may arise with the `tone` system call if two user applications want to use the speaker concurrently? There various ways in which OS can allow applications to share the speaker, and you should think about the trade-offs of each approach. With modern sound cards, one can simply "mix" the signals (i.e., play both tones at the same time) to avoid this problem. In the Pintos setup, where this is not possible, a simple solution is to just lock around the `tone` system call handler, but this could cause multiple songs to become interleaved between notes in undesirable ways. Another possibility is to allow a process to "acquire" the speaker as a resource while it is using it, blocking other processes that want to acquire it meanwhile until it becomes available. Using this approach for other I/O devices could cause applications to deadlock if they aren't careful, however.

Why do we need to write a new system call to allow user programs to use the speaker? Why can't the user application call the `speaker_on`, `speaker_off`, and `speaker_beep` functions directly?

## 5 Booting Pintos Natively

For the contest, it's fine to run your Pintos implementation in VMWare. This section describes how to run your Pintos implementation natively on real hardware. **There are inherent risks to doing this, as any bugs in your Pintos implementation could result in loss of data or could even damage your computer. We suggest waiting until after the semester to try this out.**

In Summer 2020, I posted a video[11] showing what it looks like to run Pintos natively. Feel free to watch the video if you want to see what it looks like to run Pintos natively, without the hassle of setting it up yourself. Depending on your computer, you may need to explicitly select the device to boot from, if you boot Pintos from a USB drive. See Booting Pintos off of a USB Drive for details.

### 5.1 Creating a Bootable Disk

The `cs162proj.dsk` file is a raw disk image. By copying this image to a disk drive, we can obtain a drive that can boot Pintos. You can use the `dd` utility on Linux to do this:

```
$ dd if=cs162proj.dsk of=/dev/sda bs=1M
```

**If you do this, then the device represented by `/dev/sda` will now boot your Pintos implementation, and you will lose all existing data on that device.** You'll probably also need to use `sudo` to have write permissions to the block device. Therefore, we do not recommending using a disk drive that contains any valuable data on it for booting Pintos.

If you'd like to keep your data, you can try partitioning your hard disk drive using a tool like `gparted`. We intentionally do not provide detailed instructions for this, because `gparted` can be tricky to use. **You should make sure you *really* understand what you're doing before trying to partition your computer's hard drive.**

### 5.2 Booting Pintos off of a USB Drive

An alternative, that allows you to keep your data, is to instead create a USB drive that can boot Pintos, and then boot Pintos off of that USB drive. To create such a USB drive, replace `/dev/sda` in the command above with the device corresponding to the USB drive that you want to use.

Simply copying Pintos to a USB drive won't work, however. Pintos won't be able to find the file system, because there is no USB support in Pintos. Fortunately, I managed to find some code online that implements a USB driver for Pintos, which I could modify to work with the Pintos we used in the class

[11]https://youtu.be/6pZZpCWGXPM

projects. **The downside is that these drivers only work with USB 1.x, so you will need a computer with a UHCI controller to be able to boot Pintos off of USB. Typically, only older computers, like those made in the 2000s or early 2010s, have a UHCI controller.**

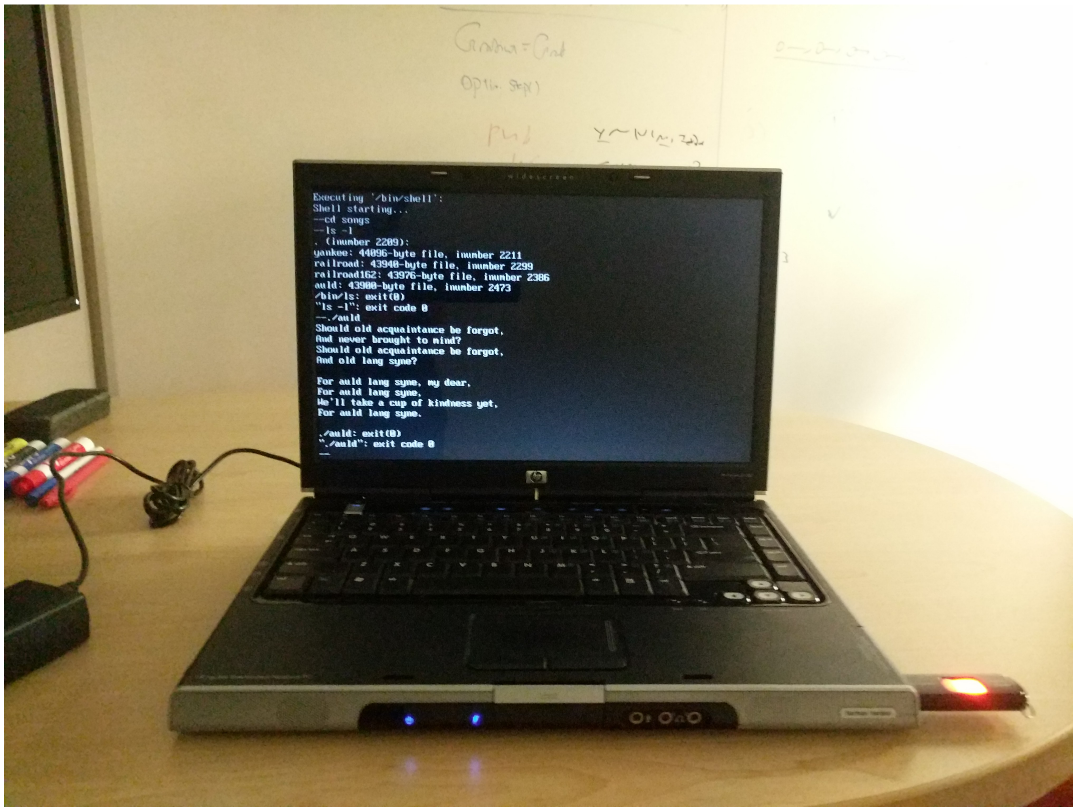
You can find this code on the `fun-drivers` branch. Merge it into your Pintos implementation with the following commands:

```
$ git fetch staff
$ git merge staff/fun-drivers
```

You should run the above commands to add PCI and USB storage support to your Pintos implementation, and then create a new `cs162proj.dsk`. Then copy the disk image to the USB storage device using the above `dd` command.

To boot Pintos off of the USB drive, you may need to select the USB drive in the BIOS. The procedure for this is different depending on your computer, but it is often done by pressing `F2`, `F10`, or `F12` keys when your computer is initially powered on. This is partially demonstrated in the live Pintos demo that I did in lecture on December 3, 2019[12].

Once you successfully boot Pintos off of a USB drive, you should be able to interact with the shell, just as you did in the VM. It should look something like this:



```
Executing '/bin/shell':
Shell starting...
--cd songs
--ls -l
. (inumber 2209):
yankee: 44096-byte file, inumber 2211
railroad: 43940-byte file, inumber 2299
railroad162: 43976-byte file, inumber 2386
auld: 43900-byte file, inumber 2473
/bin/ls: exit(0)
"ls -l": exit code 0
--./auld
Should old acquaintance be forgot,
And never brought to mind?
Should old acquaintance be forgot,
And old lang syne?

For auld lang syne, my dear,
For auld lang syne,
We'll take a cup of kindness yet,
For auld lang syne.

./auld: exit(0)
"./auld": exit code 0
--
```

## 5.3 Troubleshooting

**When I try booting Pintos, it gets stuck in the middle of a print statement.**

[12]https://youtu.be/OnnvzB2zRmw?t=2420

You probably did not correctly disable the serial functionality by deleting or commenting out the line of code specified in Prepare Pintos to run in VMWare.

**When I try booting Pintos, it gets stuck while setting up a particular hardware device.**

A simple fix is to comment out some code in the Pintos initialization routine to prevent Pintos from interacting with that hardware device. For example, if it gets stuck when scanning the PCI bus, you can try commenting out this line[13]. The important thing is that Pintos finds and initializes your USB drive containing the Pintos file system.

If you're running into an issue, and you're sure you have tried the fixes above, you should try downloading and using the staff `cs162proj.dsk` file from the course website. This will help you determine whether the issue is caused by your Pintos implementation, or by how you configured the VM in VMWare.

[13] https://github.com/Berkeley-CS162/group0/blob/fun-drivers/src/devices/pci.c#L559