1.1.2 Building Pintos

As 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.

Watch the commands executed during the build. On the Linux machines, the ordinary system tools are used.

Following the build, the following are the interesting files in the "build" directory:

"Makefile"
: A copy of "pintos/src/Makefile.build". It describes how to build the kernel. See Adding Source Files, for more information.

"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 (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 kB 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.

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.

1.2 Grading

We will grade your assignments based on test results and design quality, each of which comprises 50% of your grade.

1.2.1 Testing

Your test result grade will be based on our tests. Each project has several tests, each of which has a name beginning with "tests". To completely test your submission, invoke make check from the project "build" directory. This will build and run each test and print a "pass" or "fail" message for each one. When a test fails, make check also prints some details of the reason for failure. After running all the tests, make check also prints a summary of the test results.

For project 1, the tests will probably run faster in Bochs. For the rest of the projects, they will run much faster in QEMU. make check will select the faster simulator by default, but you can override its choice by specifying "`SIMULATOR=--bochs`" or "`SIMULATOR=--qemu`" on the make command line.

You can also run individual tests one at a time. A given test *t* writes its output to "*t*.output", then a script scores the output as "pass" or "fail" and writes the verdict to "*t*.result". To run and grade a single test, make the ".result" file explicitly from the "build" directory, e.g. make tests/threads/alarm-multiple.result. If make says that the test result is up-to-date, but you want to re-run it anyway, either run make clean or delete the ".output" file by hand.

By default, each test provides feedback only at completion, not during its run. If you prefer, you can observe the progress of each test by specifying "`VERBOSE=1`" on the make command line, as in make check VERBOSE=1. You can also provide arbitrary options to the pintos run by the tests with "`PINTOSOPTS='...'`", e.g. make check PINTOSOPTS='-j 1' to select a jitter value of 1 (see section 1.1.4 Debugging versus Testing).

All of the tests and related files are in "pintos/src/tests". Before we test your submission, we will replace the contents of that directory by a pristine, unmodified copy, to ensure that the correct tests are used. Thus, you can modify some of the tests if that helps in debugging, but we will run the originals.

All software has bugs, so some of our tests may be flawed. If you think a test failure is a bug in the test, not a bug in your code, please point it out. We will look at it and fix it if necessary.

Please don't try to take advantage of our generosity in giving out our test suite. Your code has to work properly in the general case, not just for the test cases we supply. For example, it would be unacceptable to explicitly base the kernel's behavior on the name of the running test case. Such attempts to side-step the test cases will receive no credit. If you think your solution may be in a gray area here, please ask us about it.

1.2.2 Design

We will judge your design based on the design document and the source code that you submit. We will read your entire design document and much of your source code.

Don't forget that design quality, including the design document, is 50% of your project grade. It is better to spend one or two hours writing a good design document than it is to spend that time getting the last 5% of the points for tests and then trying to rush through writing the design document in the last 15 minutes.

1.2.2.1 Design Document

We provide a design document template for each project. For each significant part of a project, the template asks questions in four areas:

Data Structures

The instructions for this section are always the same:

> Copy here the declaration of each new or changed struct or struct member, global or static variable, typedef, or enumeration. Identify the purpose of each in 25 words or less.

The first part is mechanical. Just copy new or modified declarations into the design document, to highlight for us the actual changes to data structures. Each declaration should include the comment that should accompany it in the source code (see below).

We also ask for a very brief description of the purpose of each new or changed data structure. The limit of 25 words or less is a guideline intended to save your time and avoid duplication with later areas.

Algorithms

This is where you tell us how your code works, through questions that probe your understanding of your code. We might not be able to easily figure it out from the code, because many creative solutions exist for most OS problems. Help us out a little.

Your answers should be at a level below the high level description of requirements given in the assignment. We have read the assignment too, so it is unnecessary to repeat or rephrase what is stated there. On the other hand, your answers should be at a level above the low level of the code itself. Don't give a line-by-line run-down of what your code does. Instead, use your answers to explain how your code works to implement the requirements.

Synchronization

An operating system kernel is a complex, multithreaded program, in which synchronizing multiple threads can be difficult. This section asks about how you chose to synchronize this particular type of activity.

Rationale

Whereas the other sections primarily ask "what" and "how," the rationale section concentrates on "why." This is where we ask you to justify some design decisions, by explaining why the choices you made are better than alternatives. You may be able to state these in terms of time and space complexity, which can be made as rough or informal arguments (formal language or proofs are unnecessary).

An incomplete, evasive, or non-responsive design document or one that strays from the template without good reason may be penalized. Incorrect capitalization, punctuation, spelling, or grammar can also cost points. See section D. Project Documentation, for a sample design document for a fictitious project.

1.2.2.2 Source Code

Your design will also be judged by looking at your source code. We will typically look at the differences between the original Pintos source tree and your submission, based on the output of a command like diff -urpb pintos.orig pintos.submitted. We will try to match up your description of the design with the code submitted. Important discrepancies between the description and the actual code will be penalized, as will be any bugs we find by spot checks.

The most important aspects of source code design are those that specifically relate to the operating system issues at stake in the project. For example, the organization of an inode is an important part of file system design, so in the file system project a poorly designed inode would lose points. Other issues are much less important. For example, multiple Pintos design problems call for a "priority queue," that is, a dynamic collection from which the minimum (or maximum) item can quickly be extracted. Fast priority queues can be implemented many ways, but we do not expect you to build a fancy data structure even if it might improve performance. Instead, you are welcome to use a linked list (and Pintos even provides one with convenient functions for sorting and finding minimums and maximums).

Pintos is written in a consistent style. Make your additions and modifications in existing Pintos source files blend in, not stick out. In new source files, adopt the existing Pintos style by preference, but make your code self-consistent at the very least. There should not be a patchwork of different styles that makes it obvious that three different people wrote the code. Use horizontal and vertical white space to make code readable. Add a brief comment on every structure, structure member, global or static variable, typedef, enumeration, and function definition. Update existing comments as you modify code. Don't comment out or use the preprocessor to ignore blocks of code (instead, remove it entirely). Use assertions to document key invariants. Decompose code into functions for clarity. Code that is difficult to understand because it violates these or other "common sense" software engineering practices will be penalized.

In the end, remember your audience. Code is written primarily to be read by humans. It has to be acceptable to the compiler too, but the compiler doesn't care about how it looks or how well it is written.