**Pintos**

## by Ben Pfaff

# Short Contents

##### Introduction [1](#_bookmark0)

##### Project 1: Threads [9](#_bookmark11)

##### Project 2: User Programs [21](#_bookmark24)

##### Project 3: Virtual Memory [37](#_bookmark44)

##### Project 4: File Systems [48](#_bookmark60)

##### Reference Guide [56](#_bookmark68)

##### 4.4BSD Scheduler [88](#_bookmark117)

##### Coding Standards [93](#_bookmark122)

##### Project Documentation [96](#_bookmark126)

##### Debugging Tools [99](#_bookmark127)

##### Development Tools [110](#_bookmark139)

##### Installing Pintos [114](#_bookmark145)

##### Bibliography [117](#_bookmark148)

##### License [119](#_bookmark151)

**Table of Contents**

## Introduction [1](#_bookmark0)

* + - 1. Getting Started [1](#_bookmark0)
         1. Source Tree Overview [1](#_bookmark0)
         2. Building Pintos [2](#_bookmark1)
         3. Running Pintos [3](#_bookmark2)
         4. Debugging versus Testing [4](#_bookmark3)
      2. Grading [5](#_bookmark5)
         1. Testing [5](#_bookmark5)
         2. Design [6](#_bookmark7)

Design Document [6](#_bookmark7)

Source Code [7](#_bookmark9)

* + - 1. Legal and Ethical Issues [8](#_bookmark10)
      2. Acknowledgements [8](#_bookmark10)
      3. Trivia [8](#_bookmark10)

## Project 1: Threads [9](#_bookmark11)

* + - 1. Background [9](#_bookmark11)
         1. Understanding Threads [9](#_bookmark11)
         2. Source Files [10](#_bookmark12)

‘devices’ code [11](#_bookmark13)

‘lib’ files [12](#_bookmark14)

* + - * 1. Synchronization [13](#_bookmark15)
        2. Development Suggestions [14](#_bookmark16)
      1. Requirements [14](#_bookmark16)
         1. Design Document [14](#_bookmark16)
         2. Alarm Clock [14](#_bookmark16)
         3. Priority Scheduling [15](#_bookmark17)
         4. Advanced Scheduler [16](#_bookmark19)
      2. FAQ [16](#_bookmark19)
         1. Alarm Clock FAQ [18](#_bookmark22)
         2. Priority Scheduling FAQ [18](#_bookmark22)
         3. Advanced Scheduler FAQ [19](#_bookmark23)

## Project 2: User Programs [21](#_bookmark24)

* + - 1. Background [21](#_bookmark24)
         1. Source Files [21](#_bookmark24)
         2. Using the File System [22](#_bookmark25)
         3. How User Programs Work [24](#_bookmark27)
         4. Virtual Memory Layout [24](#_bookmark27)

Typical Memory Layout [25](#_bookmark29)

* + - * 1. Accessing User Memory [26](#_bookmark30)
      1. Suggested Order of Implementation [27](#_bookmark31)
      2. Requirements [27](#_bookmark31)
         1. Design Document [27](#_bookmark31)
         2. Process Termination Messages [28](#_bookmark32)
         3. Argument Passing [28](#_bookmark32)
         4. System Calls [28](#_bookmark32)
         5. Denying Writes to Executables [31](#_bookmark35)
      3. FAQ [31](#_bookmark35)
         1. Argument Passing FAQ [33](#_bookmark37)
         2. System Calls FAQ [33](#_bookmark37)
      4. 80*x*86 Calling Convention [34](#_bookmark38)
         1. Program Startup Details [34](#_bookmark38)
         2. System Call Details [36](#_bookmark42)
    1. Project 3: Virtual Memory [37](#_bookmark44)
       1. Background [37](#_bookmark44)
          1. Source Files [37](#_bookmark44)
          2. Memory Terminology [37](#_bookmark44)

Pages [37](#_bookmark44)

Frames [38](#_bookmark45)

Page Tables [38](#_bookmark45)

Swap Slots [38](#_bookmark45)

* + - * 1. Resource Management Overview [39](#_bookmark47)
        2. Managing the Supplemental Page Table [40](#_bookmark48)
        3. Managing the Frame Table [40](#_bookmark48)

Accessed and Dirty Bits [41](#_bookmark50)

* + - * 1. Managing the Swap Table [41](#_bookmark50)
        2. Managing Memory Mapped Files [42](#_bookmark53)
      1. Suggested Order of Implementation [42](#_bookmark53)
      2. Requirements [43](#_bookmark54)
         1. Design Document [43](#_bookmark54)
         2. Paging [43](#_bookmark54)
         3. Stack Growth [44](#_bookmark55)
         4. Memory Mapped Files [45](#_bookmark56)
      3. FAQ [45](#_bookmark56)
    1. Project 4: File Systems [48](#_bookmark60)
       1. Background [48](#_bookmark60)
          1. New Code [48](#_bookmark60)
          2. Testing File System Persistence [49](#_bookmark61)
       2. Suggested Order of Implementation [49](#_bookmark61)
       3. Requirements [49](#_bookmark61)
          1. Design Document [49](#_bookmark61)
          2. Indexed and Extensible Files [50](#_bookmark62)
          3. Subdirectories [50](#_bookmark62)
          4. Buffer Cache [52](#_bookmark64)
          5. Synchronization [52](#_bookmark64)
       4. FAQ [53](#_bookmark66)
          1. Indexed Files FAQ [54](#_bookmark67)
          2. Subdirectories FAQ [54](#_bookmark67)
          3. Buffer Cache FAQ [54](#_bookmark67)

Appendix A Reference Guide [56](#_bookmark68)

* 1. Loading [56](#_bookmark68)
     1. The Loader [56](#_bookmark68)
     2. Kernel Initialization [57](#_bookmark71)
  2. Threads [58](#_bookmark73)
     1. struct thread [58](#_bookmark73)
     2. Thread Functions [61](#_bookmark77)
     3. Thread Switching [62](#_bookmark78)
  3. Synchronization [63](#_bookmark80)
     1. Disabling Interrupts [63](#_bookmark80)
     2. Semaphores [64](#_bookmark83)
     3. Locks [65](#_bookmark85)
     4. Monitors [66](#_bookmark86)
        1. Monitor Example [67](#_bookmark88)
     5. Optimization Barriers [67](#_bookmark88)
  4. Interrupt Handling [69](#_bookmark89)
     1. Interrupt Infrastructure [70](#_bookmark91)
     2. Internal Interrupt Handling [71](#_bookmark92)
     3. External Interrupt Handling [72](#_bookmark94)
  5. Memory Allocation [73](#_bookmark95)
     1. Page Allocator [73](#_bookmark95)
     2. Block Allocator [74](#_bookmark97)
  6. Virtual Addresses [75](#_bookmark99)
  7. Page Table [76](#_bookmark101)
     1. Creation, Destruction, and Activation [76](#_bookmark101)
     2. Inspection and Updates [77](#_bookmark103)
     3. Accessed and Dirty Bits [77](#_bookmark103)
     4. Page Table Details [78](#_bookmark105)
        1. Structure [78](#_bookmark105)
        2. Page Table Entry Format [80](#_bookmark106)
        3. Page Directory Entry Format [81](#_bookmark108)
  8. Hash Table [81](#_bookmark108)
     1. Data Types [82](#_bookmark110)
     2. Basic Functions [83](#_bookmark111)
     3. Search Functions [84](#_bookmark112)
     4. Iteration Functions [85](#_bookmark113)
     5. Hash Table Example [86](#_bookmark114)
     6. Auxiliary Data [87](#_bookmark115)
     7. Synchronization [87](#_bookmark115)

Appendix B 4.4BSD Scheduler [88](#_bookmark117)

* 1. Niceness [88](#_bookmark117)
  2. Calculating Priority [88](#_bookmark117)
  3. Calculating recentcpu [89](#_bookmark119)
  4. Calculating load *avg* [90](#_bookmark120)
  5. Summary [90](#_bookmark120)
  6. Fixed-Point Real Arithmetic [91](#_bookmark121)

Appendix C Coding Standards [93](#_bookmark122)

C.1 Style [93](#_bookmark122)

C.2 C99 [93](#_bookmark122)

C.3 Unsafe String Functions [94](#_bookmark124)

Appendix D Project Documentation [96](#_bookmark126)

* 1. Sample Assignment [96](#_bookmark126)
  2. Sample Design Document [96](#_bookmark126)

Appendix E Debugging Tools [99](#_bookmark127)

* 1. printf() [99](#_bookmark127)
  2. ASSERT [99](#_bookmark127)
  3. Function and Parameter Attributes [99](#_bookmark127)
  4. Backtraces [100](#_bookmark129)
     1. Example [100](#_bookmark129)
  5. GDB [102](#_bookmark131)
     1. Using GDB [102](#_bookmark131)
     2. Example GDB Session [105](#_bookmark133)
     3. Debugging User Programs [107](#_bookmark134)
     4. FAQ [107](#_bookmark134)
  6. Triple Faults [108](#_bookmark136)
  7. Modifying Bochs [108](#_bookmark136)
  8. Tips [109](#_bookmark137)

Appendix F Development Tools [110](#_bookmark139)

* 1. Tags [110](#_bookmark139)
  2. CVS [110](#_bookmark139)
     1. Setting Up CVS [110](#_bookmark139)
     2. Using CVS [111](#_bookmark142)
     3. CVS Locking [112](#_bookmark143)
  3. SourceForge [112](#_bookmark143)
  4. VNC [112](#_bookmark143)

Appendix G Installing Pintos [114](#_bookmark145)

G.1 Building Bochs for Pintos [115](#_bookmark146)

Bibliography [117](#_bookmark148)

Hardware References [117](#_bookmark148)

Software References [117](#_bookmark148)

Operating System Design References [118](#_bookmark149)

License [119](#_bookmark151)

# Introduction

Welcome to Pintos. Pintos is a simple operating system framework for the 80*x*86 archi- tecture. It supports kernel threads, loading and running user programs, and a file system, but it implements all of these in a very simple way. In the Pintos projects, you and your project team will strengthen its support in all three of these areas. You will also add a virtual memory implementation.

Pintos could, theoretically, run on a regular IBM-compatible PC. Unfortunately, it is impractical to supply every CS 140 student a dedicated PC for use with Pintos. Therefore, we will run Pintos projects in a system simulator, that is, a program that simulates an 80*x*86 CPU and its peripheral devices accurately enough that unmodified operating systems and software can run under it. In class we will use the [Bochs](http://bochs.sourceforge.net/) and [QEMU](http://fabrice.bellard.free.fr/qemu/) simulators. Pintos has also been tested with [VMware Player](http://www.vmware.com/).

These projects are hard. CS 140 has a reputation of taking a lot of time, and deservedly so. We will do what we can to reduce the workload, such as providing a lot of support material, but there is plenty of hard work that needs to be done. We welcome your feedback. If you have suggestions on how we can reduce the unnecessary overhead of assignments, cutting them down to the important underlying issues, please let us know.

This chapter explains how to get started working with Pintos. You should read the entire chapter before you start work on any of the projects.

## Getting Started

To get started, you’ll have to log into a machine that Pintos can be built on. The CS140 “officially supported” Pintos development machines are the machines in Sweet Hall managed by Stanford ITSS, as described on the [ITSS webpage](http://www.stanford.edu/services/cluster/environs/sweet/). You may use the Solaris or Linux machines. We will test your code on these machines, and the instructions given here assume this environment. We cannot provide support for installing and working on Pintos on your own machine, but we provide instructions for doing so nonetheless (see [Appendix G](#_bookmark145) [[Installing Pintos], page 114](#_bookmark145)).

Once you’ve logged into one of these machines, either locally or remotely, start out by adding our binaries directory to your PATH environment. Under csh, Stanford’s login shell, you can do so with this command:1

set path = ( /usr/class/cs140/‘uname -m‘/bin $path )

**Notice that both ‘**‘**’ are left single quotes or “backticks,” not apostrophes (‘**’**’).** It is a good idea to add this line to the ‘.cshrc’ file in your home directory. Otherwise, you’ll have to type it every time you log in.

#### Source Tree Overview

Now you can extract the source for Pintos into a directory named ‘pintos/src’, by executing

zcat /usr/class/cs140/pintos/pintos.tar.gz | tar x

Alternatively, fetch <http://www.stanford.edu/class/cs140/pintos/pintos.tar.gz>

and extract it in a similar way.

1 The term ‘‘uname -m‘’ expands to either ‘sun4u’ or ‘i686’ according to the type of computer you’re

logged into.

Let’s take a look at what’s inside. Here’s the directory structure that you should see in ‘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/’

‘devices/’

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.

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/’

‘utils/’ These files may come in handy if you decide to try working with Pintos away from the ITSS machines. Otherwise, you can ignore them.

#### 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. On a SPARC machine, special build tools are used, whose names begin with ‘i386-elf-’, e.g. i386-elf-gcc, i386-elf-ld. These are “cross-compiler” tools. That is, the build is running on a SPARC machine (called the *host*), but the result will run on a simulated 80*x*86 machine (called the *target*). The ‘i386-elf-*program* ’ tools are specially built for this configuration.

Following the build, the following are the interesting files in the ‘build’ directory:

‘Makefile’

‘kernel.o’

A copy of ‘pintos/src/Makefile.build’. It describes how to build the kernel. See [[Adding Source Files], page 17](#_bookmark20), for more information.

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], page 102](#_bookmark132)) or backtrace (see [Section E.4 [Backtraces], page 100](#_bookmark130)) 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.

‘os.dsk’ Disk image for the kernel, which is just ‘loader.bin’ followed by ‘kernel.bin’.

This file is used as a “virtual disk” by the simulator.

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.

#### 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 ‘build’ directory. Then issue the command pintos run alarm-multiple, which passes the arguments run alarm-multiple to the Pin- tos kernel. In these arguments, run instructs the kernel to run a test and alarm-multiple is the test to run.

This command creates a ‘bochsrc.txt’ file, which is needed for running Bochs, and then invoke Bochs. Bochs opens a new window that represents the simulated machine’s display, and a BIOS message briefly flashes. Then Pintos boots and runs the alarm-multiple test program, which outputs a few screenfuls 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 X output: pintos

-v -- run alarm-multiple.)

The text printed by Pintos inside Bochs probably went by too quickly to read. However, you’ve probably noticed by now 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 stdin and stdout. You can log serial output to a file by redirecting at the command line, e.g. pintos run alarm-multiple > logfile.

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. Options can select a simulator to use: the default is Bochs, but ‘--qemu’ selects QEMU. You can run the simulator with a debugger (see [Section E.5 [GDB], page 102](#_bookmark132)). 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.

#### Debugging versus Testing

When you’re debugging code, it’s useful to be able to run a program twice and have it do exactly the same thing. On second and later runs, you can make new observations without having to discard or verify your old observations. This property is called “reproducibility.” One of the simulators that Pintos supports, Bochs, can be set up for reproducibility, and that’s the way that pintos invokes it by default.

Of course, a simulation can only be reproducible from one run to the next if its input is the same each time. For simulating an entire computer, as we do, this means that every part of the computer must be the same. For example, you must use the same command-line argument, the same disks, the same version of Bochs, and you must 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.

While reproducibility is useful for debugging, it is a problem for testing thread synchro- nization, an important part of most of the projects. In particular, when Bochs is set up for reproducibility, timer interrupts will come at perfectly reproducible points, and therefore so will thread switches. That means 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.

So, to make your code easier to test, we’ve added a feature, called “jitter,” to Bochs, that makes timer interrupts come at random intervals, but in a perfectly predictable way.

In particular, if you invoke pintos with the option ‘-j *seed* ’, timer interrupts will come at irregularly spaced intervals. Within a single *seed* value, execution will still be reproducible, but timer behavior will change as *seed* is varied. Thus, for the highest degree of confidence you should test your code with many seed values.

On the other hand, when Bochs runs in reproducible mode, timings are not realistic, meaning that a “one-second” delay may be much shorter or even much longer than one second. You can invoke pintos with a different option, ‘-r’, to set up Bochs for realistic timings, in which a one-second delay should take approximately one second of real time. Simulation in real-time mode is not reproducible, and options ‘-j’ and ‘-r’ are mutually exclusive.

The QEMU simulator is available as an alternative to Bochs (use ‘--qemu’ when invoking pintos). The QEMU simulator is much faster than Bochs, but it only supports real-time simulation and does not have a reproducible mode.

## Grading

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

#### 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], page 4](#_bookmark4)).

All of the tests and related files are in ‘pintos/src/tests’. Before we test your sub- mission, 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.

#### 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.

#### 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. Iden- tify 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 require- ments 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, punctua- tion, spelling, or grammar can also cost points. See [Appendix D [Project Documentation],](#_bookmark126) [page 96](#_bookmark126), for a sample design document for a fictitious project.

#### 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.

## Legal and Ethical Issues

Pintos is distributed under a liberal license that allows free use, modification, and distribu- tion. Students and others who work on Pintos own the code that they write and may use it for any purpose. Pintos comes with NO WARRANTY, not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See [[License], page 119](#_bookmark151), for details of the license and lack of warranty.

In the context of Stanford’s CS 140 course, please respect the spirit and the letter of the honor code by refraining from reading any homework solutions available online or elsewhere. Reading the source code for other operating system kernels, such as Linux or FreeBSD, is allowed, but do not copy code from them literally. Please cite the code that inspired your own in your design documentation.

## Acknowledgements

Pintos and this documentation were written by Ben Pfaff [blp@cs.stanford.edu](mailto:blp@cs.stanford.edu).

The GDB macros supplied with Pintos were written by Godmar Back [gback@cs.vt.edu](mailto:gback@cs.vt.edu), and their documentation is adapted from his work.

The original structure and form of Pintos was inspired by the Nachos instructional operating system from the University of California, Berkeley.

A few of the Pintos source files are derived from code used in the Massachusetts Institute of Technology’s 6.828 advanced operating systems course. These files bear the original MIT license notice.

The Pintos projects and documentation originated with those designed for Nachos by current and former CS 140 teaching assistants at Stanford University, including at least Yu Ping, Greg Hutchins, Kelly Shaw, Paul Twohey, Sameer Qureshi, and John Rector.

Example code for monitors (see [Section A.3.4 [Monitors], page 66](#_bookmark87)) is from classroom slides originally by Dawson Engler and updated by Mendel Rosenblum.

## Trivia

Pintos originated as a replacement for Nachos with a similar design. Since then Pintos has greatly diverged from the Nachos design. Pintos differs from Nachos in two important ways. First, Pintos runs on real or simulated 80*x*86 hardware, but Nachos runs as a process on a host operating system. Second, Pintos is written in C like most real-world operating systems, but Nachos is written in C++.

Why the name “Pintos”? First, like nachos, pinto beans are a common Mexican food. Second, Pintos is small and a “pint” is a small amount. Third, like drivers of the eponymous car, students are likely to have trouble with blow-ups.

# Project 1: Threads

In this assignment, we give you a minimally functional thread system. Your job is to extend the functionality of this system to gain a better understanding of synchronization problems.

You will be working primarily in the ‘threads’ directory for this assignment, with some work in the ‘devices’ directory on the side. Compilation should be done in the ‘threads’ directory.

Before you read the description of this project, you should read all of the following sections: [Chapter 1 [Introduction], page 1](#_bookmark0), [Appendix C [Coding Standards], page 93](#_bookmark122), [Ap-](#_bookmark127) [pendix E [Debugging Tools], page 99](#_bookmark127), and [Appendix F [Development Tools], page 110](#_bookmark139). You should at least skim the material from [Section A.1 [Pintos Loading], page 56](#_bookmark69) through [Sec-](#_bookmark95) [tion A.5 [Memory Allocation], page 73](#_bookmark95), especially [Section A.3 [Synchronization], page 63](#_bookmark81). To complete this project you will also need to read [Appendix B [4.4BSD Scheduler], page 88](#_bookmark117).

## Background

#### Understanding Threads

The first step is to read and understand the code for the initial thread system. Pintos already implements thread creation and thread completion, a simple scheduler to switch between threads, and synchronization primitives (semaphores, locks, condition variables, and optimization barriers).

Some of this code might seem slightly mysterious. If you haven’t already compiled and run the base system, as described in the introduction (see [Chapter 1 [Introduction], page 1](#_bookmark0)), you should do so now. You can read through parts of the source code to see what’s going on. If you like, you can add calls to printf() almost anywhere, then recompile and run to see what happens and in what order. You can also run the kernel in a debugger and set breakpoints at interesting spots, single-step through code and examine data, and so on.

When a thread is created, you are creating a new context to be scheduled. You provide a function to be run in this context as an argument to thread\_create(). The first time the thread is scheduled and runs, it starts from the beginning of that function and executes in that context. When the function returns, the thread terminates. Each thread, there- fore, acts like a mini-program running inside Pintos, with the function passed to thread\_ create() acting like main().

At any given time, exactly one thread runs and the rest, if any, become inactive. The scheduler decides which thread to run next. (If no thread is ready to run at any given time, then the special “idle” thread, implemented in idle(), runs.) Synchronization primitives can force context switches when one thread needs to wait for another thread to do something.

The mechanics of a context switch are in ‘threads/switch.S’, which is 80*x*86 assembly code. (You don’t have to understand it.) It saves the state of the currently running thread and restores the state of the thread we’re switching to.

Using the GDB debugger, slowly trace through a context switch to see what happens (see [Section E.5 [GDB], page 102](#_bookmark132)). You can set a breakpoint on schedule() to start out,

and then single-step from there.1 Be sure to keep track of each thread’s address and state, and what procedures are on the call stack for each thread. You will notice that when one thread calls switch\_threads(), another thread starts running, and the first thing the new thread does is to return from switch\_threads(). You will understand the thread system once you understand why and how the switch\_threads() that gets called is different from the switch\_threads() that returns. See [Section A.2.3 [Thread Switching], page 62](#_bookmark79), for more information.

**Warning**: In Pintos, each thread is assigned a small, fixed-size execution stack just under 4 kB in size. The kernel tries to detect stack overflow, but it cannot do so perfectly. You may cause bizarre problems, such as mysterious kernel panics, if you declare large data structures as non-static local variables, e.g. ‘int buf[1000];’. Alternatives to stack allocation include the page allocator and the block allocator (see [Section A.5 [Memory Allocation], page 73](#_bookmark95)).

#### Source Files

Here is a brief overview of the files in the ‘threads’ directory. You will not need to modify most of this code, but the hope is that presenting this overview will give you a start on what code to look at.

‘loader.S’ ‘loader.h’

The kernel loader. Assembles to 512 bytes of code and data that the PC BIOS loads into memory and which in turn loads the kernel into memory, does basic processor initialization, and jumps to the beginning of the kernel. See [Section A.1.1 [Pintos Loader], page 56](#_bookmark70), for details. You should not need to look at this code or modify it.

‘kernel.lds.S’

The linker script used to link the kernel. Sets the load address of the kernel and arranges for ‘start.S’ to be at the very beginning of the kernel image. See [Section A.1.1 [Pintos Loader], page 56](#_bookmark70), for details. Again, you should not need to look at this code or modify it, but it’s here in case you’re curious.

‘start.S’ Jumps to main(). ‘init.c’

‘init.h’ Kernel initialization, including main(), the kernel’s “main program.” You should look over main() at least to see what gets initialized. You might want to add your own initialization code here. See [Section A.1.2 [Kernel Initialization],](#_bookmark72) [page 57](#_bookmark72), for details.

‘thread.c’ ‘thread.h’

Basic thread support. Much of your work will take place in these files. ‘thread.h’ defines struct thread, which you are likely to modify in all four projects. See [Section A.2.1 [struct thread], page 58](#_bookmark75) and [Section A.2 [Threads],](#_bookmark74) [page 58](#_bookmark74) for more information.

1 GDB might tell you that schedule() doesn’t exist, which is arguably a GDB bug. You can work around this by setting the breakpoint by filename and line number, e.g. break thread.c:*ln* where *ln* is the line number of the first declaration in schedule().

‘switch.S’ ‘switch.h’

‘palloc.c’ ‘palloc.h’

‘malloc.c’ ‘malloc.h’

Assembly language routine for switching threads. Already discussed above. See [Section A.2.2 [Thread Functions], page 61](#_bookmark77), for more information.

Page allocator, which hands out system memory in multiples of 4 kB pages. See [Section A.5.1 [Page Allocator], page 73](#_bookmark96), for more information.

A simple implementation of malloc() and free() for the kernel. See [Sec-](#_bookmark98) [tion A.5.2 [Block Allocator], page 74](#_bookmark98), for more information.

‘interrupt.c’ ‘interrupt.h’

Basic interrupt handling and functions for turning interrupts on and off. See [Section A.4 [Interrupt Handling], page 69](#_bookmark90), for more information.

‘intr-stubs.S’ ‘intr-stubs.h’

Assembly code for low-level interrupt handling. See [Section A.4.1 [Interrupt](#_bookmark91) [Infrastructure], page 70](#_bookmark91), for more information.

‘synch.c’

‘synch.h’ Basic synchronization primitives: semaphores, locks, condition variables, and optimization barriers. You will need to use these for synchronization in all four projects. See [Section A.3 [Synchronization], page 63](#_bookmark81), for more information.

‘io.h’ Functions for I/O port access. This is mostly used by source code in the ‘devices’ directory that you won’t have to touch.

‘vaddr.h’

‘pte.h’ Functions and macros for working with virtual addresses and page table entries. These will be more important to you in project 3. For now, you can ignore them.

‘flags.h’ Macros that define a few bits in the 80*x*86 “flags” register. Probably of no in- terest. See [IA32-v1], section 3.4.3, “EFLAGS Register,” for more information.

* + - 1. **‘**devices**’ code**

The basic threaded kernel also includes these files in the ‘devices’ directory:

‘timer.c’

‘timer.h’ System timer that ticks, by default, 100 times per second. You will modify this code in this project.

‘vga.c’

‘vga.h’ VGA display driver. Responsible for writing text to the screen. You should have no need to look at this code. printf() calls into the VGA display driver for you, so there’s little reason to call this code yourself.

‘serial.c’ ‘serial.h’

Serial port driver. Again, printf() calls this code for you, so you don’t need to do so yourself. It handles serial input by passing it to the input layer (see below).

‘disk.c’

‘disk.h’ Supports reading and writing sectors on up to 4 IDE disks. This won’t actually be used until project 2.

‘kbd.c’

‘kbd.h’ Keyboard driver. Handles keystrokes passing them to the input layer (see be- low).

‘input.c’

‘input.h’ Input layer. Queues input characters passed along by the keyboard or serial drivers.

‘intq.c’

‘intq.h’ Interrupt queue, for managing a circular queue that both kernel threads and interrupt handlers want to access. Used by the keyboard and serial drivers.

* + - 1. **‘**lib**’ files**

Finally, ‘lib’ and ‘lib/kernel’ contain useful library routines. (‘lib/user’ will be used by user programs, starting in project 2, but it is not part of the kernel.) Here’s a few more details:

‘ctype.h’ ‘inttypes.h’ ‘limits.h’ ‘stdarg.h’ ‘stdbool.h’ ‘stddef.h’ ‘stdint.h’ ‘stdio.c’

‘stdio.h’ ‘stdlib.c’ ‘stdlib.h’ ‘string.c’ ‘string.h’

A subset of the standard C library. See [Section C.2 [C99], page 93](#_bookmark123), for infor- mation on a few recently introduced pieces of the C library that you might not have encountered before. See [Section C.3 [Unsafe String Functions], page 94](#_bookmark125), for information on what’s been intentionally left out for safety.

‘debug.c’

‘debug.h’ Functions and macros to aid debugging. See [Appendix E [Debugging Tools],](#_bookmark127) [page 99](#_bookmark127), for more information.

‘random.c’ ‘random.h’

Pseudo-random number generator.

‘round.h’ Macros for rounding. ‘syscall-nr.h’

System call numbers. Not used until project 2.

‘kernel/list.c’ ‘kernel/list.h’

Doubly linked list implementation. Used all over the Pintos code, and you’ll probably want to use it a few places yourself in project 1.

‘kernel/bitmap.c’ ‘kernel/bitmap.h’

Bitmap implementation. You can use this in your code if you like, but you probably won’t have any need for it in project 1.

‘kernel/hash.c’ ‘kernel/hash.h’

Hash table implementation. Likely to come in handy for project 3.

‘kernel/console.c’ ‘kernel/console.h’ ‘kernel/stdio.h’

Implements printf() and a few other functions.

#### Synchronization

Proper synchronization is an important part of the solutions to these problems. Any syn- chronization problem can be easily solved by turning interrupts off: while interrupts are off, there is no concurrency, so there’s no possibility for race conditions. Therefore, it’s tempting to solve all synchronization problems this way, but **don’t**. Instead, use semaphores, locks, and condition variables to solve the bulk of your synchronization problems. Read the tour section on synchronization (see [Section A.3 [Synchronization], page 63](#_bookmark81)) or the comments in ‘threads/synch.c’ if you’re unsure what synchronization primitives may be used in what situations.

In the Pintos projects, the only class of problem best solved by disabling interrupts is coordinating data shared between a kernel thread and an interrupt handler. Because inter- rupt handlers can’t sleep, they can’t acquire locks. This means that data shared between kernel threads and an interrupt handler must be protected within a kernel thread by turning off interrupts.

This project only requires accessing a little bit of thread state from interrupt handlers. For the alarm clock, the timer interrupt needs to wake up sleeping threads. In the advanced scheduler, the timer interrupt needs to access a few global and per-thread variables. When you access these variables from kernel threads, you will need to disable interrupts to prevent the timer interrupt from interfering.

When you do turn off interrupts, take care to do so for the least amount of code possible, or you can end up losing important things such as timer ticks or input events. Turning off

interrupts also increases the interrupt handling latency, which can make a machine feel sluggish if taken too far.

The synchronization primitives themselves in ‘synch.c’ are implemented by disabling interrupts. You may need to increase the amount of code that runs with interrupts disabled here, but you should still try to keep it to a minimum.

Disabling interrupts can be useful for debugging, if you want to make sure that a section of code is not interrupted. You should remove debugging code before turning in your project. (Don’t just comment it out, because that can make the code difficult to read.)

There should be no busy waiting in your submission. A tight loop that calls thread\_ yield() is one form of busy waiting.

#### Development Suggestions

In the past, many groups divided the assignment into pieces, then each group member worked on his or her piece until just before the deadline, at which time the group reconvened to combine their code and submit. **This is a bad idea. We do not recommend this approach.** Groups that do this often find that two changes conflict with each other, requiring lots of last-minute debugging. Some groups who have done this have turned in code that did not even compile or boot, much less pass any tests.

Instead, we recommend integrating your team’s changes early and often, using a source code control system such as CVS (see [Section F.2 [CVS], page 110](#_bookmark141)) or a group collaboration site such as SourceForge (see [Section F.3 [SourceForge], page 112](#_bookmark144)). This is less likely to produce surprises, because everyone can see everyone else’s code as it is written, instead of just when it is finished. These systems also make it possible to review changes and, when a change introduces a bug, drop back to working versions of code.

You should expect to run into bugs that you simply don’t understand while working on this and subsequent projects. When you do, reread the appendix on debugging tools, which is filled with useful debugging tips that should help you to get back up to speed (see [Appendix E [Debugging Tools], page 99](#_bookmark127)). Be sure to read the section on backtraces (see [Section E.4 [Backtraces], page 100](#_bookmark130)), which will help you to get the most out of every kernel panic or assertion failure.

## Requirements

#### Design Document

Before you turn in your project, you must copy the project 1 design document template into your source tree under the name ‘pintos/src/threads/DESIGNDOC’ and fill it in. We recommend that you read the design document template before you start working on the project. See [Appendix D [Project Documentation], page 96](#_bookmark126), for a sample design document that goes along with a fictitious project.

#### Alarm Clock

Reimplement timer\_sleep(), defined in ‘devices/timer.c’. Although a working imple- mentation is provided, it “busy waits,” that is, it spins in a loop checking the current time and calling thread\_yield() until enough time has gone by. Reimplement it to avoid busy waiting.

void timer\_sleep (*int64 t ticks* )

[Function]

Suspends execution of the calling thread until time has advanced by at least *x* timer ticks. Unless the system is otherwise idle, the thread need not wake up after exactly *x* ticks. Just put it on the ready queue after they have waited for the right amount of time.

timer\_sleep() is useful for threads that operate in real-time, e.g. for blinking the cursor once per second.

The argument to timer\_sleep() is expressed in timer ticks, not in milliseconds or any another unit. There are TIMER\_FREQ timer ticks per second, where TIMER\_FREQ is a macro defined in devices/timer.h. The default value is 100. We don’t recommend changing this value, because any change is likely to cause many of the tests to fail.

Separate functions timer\_msleep(), timer\_usleep(), and timer\_nsleep() do exist for sleeping a specific number of milliseconds, microseconds, or nanoseconds, respectively, but these will call timer\_sleep() automatically when necessary. You do not need to modify them.

If your delays seem too short or too long, reread the explanation of the ‘-r’ option to

pintos (see [Section 1.1.4 [Debugging versus Testing], page 4](#_bookmark4)).

The alarm clock implementation is not needed for later projects, although it could be useful for project 4.

#### Priority Scheduling

Implement priority scheduling in Pintos. When a thread is added to the ready list that has a higher priority than the currently running thread, the current thread should immediately yield the processor to the new thread. Similarly, when threads are waiting for a lock, semaphore, or condition variable, the highest priority waiting thread should be awakened first. A thread may raise or lower its own priority at any time, but lowering its priority such that it no longer has the highest priority must cause it to immediately yield the CPU.

Thread priorities range from PRI\_MIN (0) to PRI\_MAX (63). Lower numbers correspond to lower priorities, so that priority 0 is the lowest priority and priority 63 is the highest. The initial thread priority is passed as an argument to thread\_create(). If there’s no reason to choose another priority, use PRI\_DEFAULT (31). The PRI\_ macros are defined in ‘threads/thread.h’, and you should not change their values.

One issue with priority scheduling is “priority inversion”. Consider high, medium, and low priority threads *H*, *M*, and *L*, respectively. If *H* needs to wait for *L* (for instance, for a lock held by *L*), and *M* is on the ready list, then *H* will never get the CPU because the low priority thread will not get any CPU time. A partial fix for this problem is for *H* to “donate” its priority to *L* while *L* is holding the lock, then recall the donation once *L* releases (and thus *H* acquires) the lock.

Implement priority donation. You will need to account for all different situations in which priority donation is required. Be sure to handle multiple donations, in which multiple priorities are donated to a single thread. You must also handle nested donation: if *H* is waiting on a lock that *M* holds and *M* is waiting on a lock that *L* holds, then both *M* and *L* should be boosted to *H* ’s priority. If necessary, you may impose a reasonable limit on depth of nested priority donation, such as 8 levels.

You must implement priority donation for locks. You need not implement priority do- nation for semaphores or condition variables (but you are welcome to do so). You do need to implement priority scheduling in all cases.

Finally, implement the following functions that allow a thread to examine and modify its own priority. Skeletons for these functions are provided in ‘threads/thread.c’.

void thread\_set\_priority (*int new\_priority* )

[Function]

Sets the current thread’s priority to *new priority*. If the current thread no longer has the highest priority, yields.

int thread\_get\_priority (*void*) [Function]

Returns the current thread’s priority. In the presence of priority donation, returns the higher (donated) priority.

You need not provide any interface to allow a thread to directly modify other threads’ priorities.

The priority scheduler is not used in any later project.

#### Advanced Scheduler

Implement a multilevel feedback queue scheduler similar to the 4.4BSD scheduler to reduce the average response time for running jobs on your system. See [Appendix B [4.4BSD](#_bookmark117) [Scheduler], page 88](#_bookmark117), for detailed requirements.

Like the priority scheduler, the advanced scheduler chooses the thread to run based on priorities. However, the advanced scheduler does not do priority donation. Thus, we recom- mend that you have the priority scheduler working, except possibly for priority donation, before you start work on the advanced scheduler.

You must write your code to allow us to choose a scheduling algorithm policy at Pintos startup time. By default, the priority scheduler must be active, but we must be able to choose the 4.4BSD scheduler with the ‘-mlfqs’ kernel option. Passing this option sets thread\_mlfqs, declared in ‘threads/thread.h’, to true when the options are parsed by parse\_options(), which happens early in main().

When the 4.4BSD scheduler is enabled, threads no longer directly control their own priorities. The *priority* argument to thread\_create() should be ignored, as well as any calls to thread\_set\_priority(), and thread\_get\_priority() should return the thread’s current priority as set by the scheduler.

The advanced scheduler is not used in any later project.

## FAQ

How much code will I need to write?

Here’s a summary of our reference solution, produced by the diffstat program. The final row gives total lines inserted and deleted; a changed line counts as both an insertion and a deletion.

The reference solution represents just one possible solution. Many other so- lutions are also possible and many of those differ greatly from the reference solution. Some excellent solutions may not modify all the files modified by the

reference solution, and some may modify files not modified by the reference solution.

devices/timer.c | 42 +++++- threads/fixed-point.h | 120 ++++++++++++++++++ threads/synch.c | 88 ++++++++++++-

threads/thread.c | 196 ++++++++++++++++++++++++++---- threads/thread.h | 23 +++

5 files changed, 440 insertions(+), 29 deletions(-)

‘fixed-point.h’ is a new file added by the reference solution.

How do I update the ‘Makefile’s when I add a new source file?

To add a ‘.c’ file, edit the top-level ‘Makefile.build’. Add the new file to variable ‘*dir* \_SRC’, where *dir* is the directory where you added the file. For this project, that means you should add it to threads\_SRC or devices\_SRC. Then run make. If your new file doesn’t get compiled, run make clean and then try again.

When you modify the top-level ‘Makefile.build’ and re-run make, the modified version should be automatically copied to ‘threads/build/Makefile’. The converse is not true, so any changes will be lost the next time you run make clean from the ‘threads’ directory. Unless your changes are truly temporary, you should prefer to edit ‘Makefile.build’.

A new ‘.h’ file does not require editing the ‘Makefile’s.

**What does** warning: no previous prototype for ‘*func* ’ **mean?**

It means that you defined a non-static function without preceding it by a prototype. Because non-static functions are intended for use by other ‘.c’ files, for safety they should be prototyped in a header file included before their definition. To fix the problem, add a prototype in a header file that you include, or, if the function isn’t actually used by other ‘.c’ files, make it static.

What is the interval between timer interrupts?

Timer interrupts occur TIMER\_FREQ times per second. You can adjust this value by editing ‘devices/timer.h’. The default is 100 Hz.

We don’t recommend changing this value, because any changes are likely to cause many of the tests to fail.

How long is a time slice?

There are TIME\_SLICE ticks per time slice. This macro is declared in ‘threads/thread.c’. The default is 4 ticks.

We don’t recommend changing this value, because any changes are likely to cause many of the tests to fail.

How do I run the tests?

See [Section 1.2.1 [Testing], page 5](#_bookmark6).

Why do I get a test failure in pass()?

You are probably looking at a backtrace that looks something like this: 0xc0108810: debug\_panic (lib/kernel/debug.c:32) 0xc010a99f: pass (tests/threads/tests.c:93)

0xc010bdd3: test\_mlfqs\_load\_1 (...threads/mlfqs-load-1.c:33) 0xc010a8cf: run\_test (tests/threads/tests.c:51)

0xc0100452: run\_task (threads/init.c:283) 0xc0100536: run\_actions (threads/init.c:333) 0xc01000bb: main (threads/init.c:137)

This is just confusing output from the backtrace program. It does not actu- ally mean that pass() called debug\_panic(). In fact, fail() called debug\_ panic() (via the PANIC() macro). GCC knows that debug\_panic() does not return, because it is declared NO\_RETURN (see [Section E.3 [Function and Pa-](#_bookmark128) [rameter Attributes], page 99](#_bookmark128)), so it doesn’t include any code in pass() to take control when debug\_panic() returns. This means that the return address on the stack looks like it is at the beginning of the function that happens to follow fail() in memory, which in this case happens to be pass().

See [Section E.4 [Backtraces], page 100](#_bookmark130), for more information.

How do interrupts get re-enabled in the new thread following schedule()?

Every path into schedule() disables interrupts. They eventually get re-enabled by the next thread to be scheduled. Consider the possibilities: the new thread is running in switch\_thread() (but see below), which is called by schedule(), which is called by one of a few possible functions:

* thread\_exit(), but we’ll never switch back into such a thread, so it’s uninteresting.
* thread\_yield(), which immediately restores the interrupt level upon re- turn from schedule().
* thread\_block(), which is called from multiple places:
  + sema\_down(), which restores the interrupt level before returning.
  + idle(), which enables interrupts with an explicit assembly STI in- struction.
  + wait() in ‘devices/intq.c’, whose callers are responsible for re- enabling interrupts.

There is a special case when a newly created thread runs for the first time. Such a thread calls intr\_enable() as the first action in kernel\_thread(), which is at the bottom of the call stack for every kernel thread but the first.

* + 1. **Alarm Clock FAQ**

Do I need to account for timer values overflowing?

Don’t worry about the possibility of timer values overflowing. Timer values are expressed as signed 64-bit numbers, which at 100 ticks per second should be good for almost 2,924,712,087 years. By then, we expect Pintos to have been phased out of the CS 140 curriculum.

#### Priority Scheduling FAQ

Doesn’t priority scheduling lead to starvation?

Yes, strict priority scheduling can lead to starvation because a thread will not run if any higher-priority thread is runnable. The advanced scheduler introduces a mechanism for dynamically changing thread priorities.

Strict priority scheduling is valuable in real-time systems because it offers the programmer more control over which jobs get processing time. High priorities are generally reserved for time-critical tasks. It’s not “fair,” but it addresses other concerns not applicable to a general-purpose operating system.

What thread should run after a lock has been released?

When a lock is released, the highest priority thread waiting for that lock should be unblocked and put on the list of ready threads. The scheduler should then run the highest priority thread on the ready list.

If the highest-priority thread yields, does it continue running?

Yes. If there is a single highest-priority thread, it continues running until it blocks or finishes, even if it calls thread\_yield(). If multiple threads have the same highest priority, thread\_yield() should switch among them in “round robin” order.

What happens to the priority of a donating thread?

Priority donation only changes the priority of the donee thread. The donor thread’s priority is unchanged. Priority donation is not additive: if thread *A* (with priority 5) donates to thread *B* (with priority 3), then *B*’s new priority is 5, not 8.

Can a thread’s priority change while it is on the ready queue?

Yes. Consider this case: low-priority thread *L* holds a lock that high-priority thread *H* wants, so *H* donates its priority to *L*. *L* releases the lock and thus loses the CPU and is moved to the ready queue. Now *L*’s old priority is restored while it is in the ready queue.

Can a thread’s priority change while it is blocked?

Yes. While a thread that has acquired lock *L* is blocked for any reason, its priority can increase by priority donation if a higher-priority thread attempts to acquire *L*. This case is checked by the priority-donate-sema test.

Can a thread added to the ready list preempt the processor?

Yes. If a thread added to the ready list has higher priority than the running thread, the correct behavior is to immediately yield the processor. It is not acceptable to wait for the next timer interrupt. The highest priority thread should run as soon as it is runnable, preempting whatever thread is currently running.

**How does** thread\_set\_priority() **affect a thread receiving donations?**

It sets the thread’s base priority. The thread’s effective priority becomes the higher of the newly set priority or the highest donated priority. When the donations are released, the thread’s priority becomes the one set through the function call. This behavior is checked by the priority-donate-lower test.

#### Advanced Scheduler FAQ

How does priority donation interact with the advanced scheduler?

It doesn’t have to. We won’t test priority donation and the advanced scheduler at the same time.

Can I use one queue instead of 64 queues?

Yes. In general, your implementation may differ from the description, as long as its behavior is the same.

Some scheduler tests fail and I don’t understand why. Help!

If your implementation mysteriously fails some of the advanced scheduler tests, try the following:

* + - * Read the source files for the tests that you’re failing, to make sure that you understand what’s going on. Each one has a comment at the top that explains its purpose and expected results.
      * Double-check your fixed-point arithmetic routines and your use of them in the scheduler routines.
      * Consider how much work your implementation does in the timer interrupt. If the timer interrupt handler takes too long, then it will take away most of a timer tick from the thread that the timer interrupt preempted. When it returns control to that thread, it therefore won’t get to do much work before the next timer interrupt arrives. That thread will therefore get blamed for a lot more CPU time than it actually got a chance to use. This raises the interrupted thread’s recent CPU count, thereby lowering its priority. It can cause scheduling decisions to change. It also raises the load average.

# Project 2: User Programs

Now that you’ve worked with Pintos and are becoming familiar with its infrastructure and thread package, it’s time to start working on the parts of the system that allow running user programs. The base code already supports loading and running user programs, but no I/O or interactivity is possible. In this project, you will enable programs to interact with the OS via system calls.

You will be working out of the ‘userprog’ directory for this assignment, but you will also be interacting with almost every other part of Pintos. We will describe the relevant parts below.

You can build project 2 on top of your project 1 submission or you can start fresh. No code from project 1 is required for this assignment. The “alarm clock” functionality may be useful in projects 3 and 4, but it is not strictly required.

You might find it useful to go back and reread how to run the tests (see [Section 1.2.1](#_bookmark6) [[Testing], page 5](#_bookmark6)).

## Background

Up to now, all of the code you have run under Pintos has been part of the operating system kernel. This means, for example, that all the test code from the last assignment ran as part of the kernel, with full access to privileged parts of the system. Once we start running user programs on top of the operating system, this is no longer true. This project deals with the consequences.

We allow more than one process to run at a time. Each process has one thread (multi- threaded processes are not supported). User programs are written under the illusion that they have the entire machine. This means that when you load and run multiple processes at a time, you must manage memory, scheduling, and other state correctly to maintain this illusion.

In the previous project, we compiled our test code directly into your kernel, so we had to require certain specific function interfaces within the kernel. From now on, we will test your operating system by running user programs. This gives you much greater freedom. You must make sure that the user program interface meets the specifications described here, but given that constraint you are free to restructure or rewrite kernel code however you wish.

#### Source Files

The easiest way to get an overview of the programming you will be doing is to simply go over each part you’ll be working with. In ‘userprog’, you’ll find a small number of files, but here is where the bulk of your work will be:

‘process.c’ ‘process.h’

Loads ELF binaries and starts processes.

‘pagedir.c’ ‘pagedir.h’

A simple manager for 80*x*86 hardware page tables. Although you probably won’t want to modify this code for this project, you may want to call some of its functions. See [Section 4.1.2.3 [Page Tables], page 38](#_bookmark46), for more information.

‘syscall.c’ ‘syscall.h’

Whenever a user process wants to access some kernel functionality, it invokes a system call. This is a skeleton system call handler. Currently, it just prints a message and terminates the user process. In part 2 of this project you will add code to do everything else needed by system calls.

‘exception.c’ ‘exception.h’

When a user process performs a privileged or prohibited operation, it traps into the kernel as an “exception” or “fault.”1 These files handle exceptions. Currently all exceptions simply print a message and terminate the process. Some, but not all, solutions to project 2 require modifying page\_fault() in this file.

‘gdt.c’

‘gdt.h’ The 80*x*86 is a segmented architecture. The Global Descriptor Table (GDT) is a table that describes the segments in use. These files set up the GDT. You should not need to modify these files for any of the projects. You can read the code if you’re interested in how the GDT works.

‘tss.c’

‘tss.h’ The Task-State Segment (TSS) is used for 80*x*86 architectural task switching. Pintos uses the TSS only for switching stacks when a user process enters an interrupt handler, as does Linux. You should not need to modify these files for any of the projects. You can read the code if you’re interested in how the TSS works.

#### Using the File System

You will need to interface to the file system code for this project, because user programs are loaded from the file system and many of the system calls you must implement deal with the file system. However, the focus of this project is not the file system, so we have provided a simple but complete file system in the ‘filesys’ directory. You will want to look over the ‘filesys.h’ and ‘file.h’ interfaces to understand how to use the file system, and especially its many limitations.

There is no need to modify the file system code for this project, and so we recommend that you do not. Working on the file system is likely to distract you from this project’s focus.

Proper use of the file system routines now will make life much easier for project 4, when you improve the file system implementation. Until then, you will have to tolerate the following limitations:

* + - * No internal synchronization. Concurrent accesses will interfere with one another. You should use synchronization to ensure that only one process at a time is executing file system code.

1 We will treat these terms as synonyms. There is no standard distinction between them, although Intel

processor manuals make a minor distinction between them on 80*x*86.

* + - * File size is fixed at creation time. The root directory is represented as a file, so the number of files that may be created is also limited.
      * File data is allocated as a single extent, that is, data in a single file must occupy a contiguous range of sectors on disk. External fragmentation can therefore become a serious problem as a file system is used over time.
      * No subdirectories.
      * File names are limited to 14 characters.
      * A system crash mid-operation may corrupt the disk in a way that cannot be repaired automatically. There is no file system repair tool anyway.

One important feature is included:

* + - * Unix-like semantics for filesys\_remove() are implemented. That is, if a file is open when it is removed, its blocks are not deallocated and it may still be accessed by any threads that have it open, until the last one closes it. See [[Removing an Open File],](#_bookmark39) [page 34](#_bookmark39), for more information.

You need to be able to create simulated disks. The pintos-mkdisk program provides this functionality. From the ‘userprog/build’ directory, execute pintos-mkdisk fs.dsk 2. This command creates a 2 MB simulated disk named ‘fs.dsk’. Then format the disk by passing ‘-f -q’ on the kernel’s command line: pintos -f -q. The ‘-f’ option causes the disk to be formatted, and ‘-q’ causes Pintos to exit as soon as the format is done.

You’ll need a way to copy files in and out of the simulated file system. The pintos ‘-p’ (“put”) and ‘-g’ (“get”) options do this. To copy ‘*file* ’ into the Pintos file system, use the command ‘pintos -p *file* -- -q’. (The ‘--’ is needed because ‘-p’ is for the pintos script, not for the simulated kernel.) To copy it to the Pintos file system under the name ‘*newname* ’, add ‘-a *newname* ’: ‘pintos -p *file* -a *newname* -- -q’. The commands for copying files out of a VM are similar, but substitute ‘-g’ for ‘-p’.

Incidentally, these commands work by passing special commands put and get on the kernel’s command line and copying to and from a special simulated “scratch” disk. If you’re very curious, you can look at the pintos script as well as ‘filesys/fsutil.c’ to learn the implementation details.

Here’s a summary of how to create and format a disk, copy the echo program into the new disk, and then run echo, passing argument x. (Argument passing won’t work until you implemented it.) It assumes that you’ve already built the examples in ‘examples’ and that the current directory is ‘userprog/build’:

pintos-mkdisk fs.dsk 2 pintos -f -q

pintos -p ../../examples/echo -a echo -- -q pintos -q run ’echo x’

The three final steps can actually be combined into a single command:

pintos-mkdisk fs.dsk 2

pintos -p ../../examples/echo -a echo -- -f -q run ’echo x’

If you don’t want to keep the file system disk around for later use or inspection, you can even combine all four steps into a single command. The --fs-disk=*n* option creates a temporary disk approximately *n* megabytes in size just for the duration of the pintos run. The Pintos automatic test suite makes extensive use of this syntax:

pintos --fs-disk=2 -p ../../examples/echo -a echo -- -f -q run ’echo x’

You can delete a file from the Pintos file system using the rm *file* kernel action, e.g. pintos -q rm *file* . Also, ls lists the files in the file system and cat *file* prints a file’s contents to the display.

#### How User Programs Work

Pintos can run normal C programs, as long as they fit into memory and use only the system calls you implement. Notably, malloc() cannot be implemented because none of the system calls required for this project allow for memory allocation. Pintos also can’t run programs that use floating point operations, since the kernel doesn’t save and restore the processor’s floating-point unit when switching threads.

The ‘src/examples’ directory contains a few sample user programs. The ‘Makefile’ in this directory compiles the provided examples, and you can edit it compile your own programs as well. Some of the example programs will only work once projects 3 or 4 have been implemented.

Pintos can load *ELF* executables with the loader provided for you in ‘userprog/process.c’. ELF is a file format used by Linux, Solaris, and many other operating systems for object files, shared libraries, and executables. You can actually use any compiler and linker that output 80*x*86 ELF executables to produce programs for Pintos. (We’ve provided compilers and linkers that should do just fine.)

You should realize immediately that, until you copy a test program to the simulated disk, Pintos will be unable to do useful work. You won’t be able to do interesting things until you copy a variety of programs to the disk. You might want to create a clean reference disk and copy that over whenever you trash your ‘fs.dsk’ beyond a useful state, which may happen occasionally while debugging.

#### Virtual Memory Layout

Virtual memory in Pintos is divided into two regions: user virtual memory and kernel virtual memory. User virtual memory ranges from virtual address 0 up to PHYS\_BASE, which is defined in ‘threads/vaddr.h’ and defaults to 0xc0000000 (3 GB). Kernel virtual memory occupies the rest of the virtual address space, from PHYS\_BASE up to 4 GB.

User virtual memory is per-process. When the kernel switches from one process to another, it also switches user virtual address spaces by changing the processor’s page di- rectory base register (see pagedir\_activate() in ‘userprog/pagedir.c’). struct thread contains a pointer to a process’s page table.

Kernel virtual memory is global. It is always mapped the same way, regardless of what user process or kernel thread is running. In Pintos, kernel virtual memory is mapped one- to-one to physical memory, starting at PHYS\_BASE. That is, virtual address PHYS\_BASE accesses physical address 0, virtual address PHYS\_BASE + 0x1234 access physical address 0x1234, and so on up to the size of the machine’s physical memory.

A user program can only access its own user virtual memory. An attempt to access kernel virtual memory causes a page fault, handled by page\_fault() in ‘userprog/exception.c’, and the process will be terminated. Kernel threads can access both kernel virtual memory and, if a user process is running, the user virtual memory of the running process. However,

even in the kernel, an attempt to access memory at an unmapped user virtual address will cause a page fault.

#### 3.1.4.1 Typical Memory Layout

Conceptually, each process is free to lay out its own user virtual memory however it chooses. In practice, user virtual memory is laid out like this:

PHYS\_BASE +----------------------------------+

| user stack |

| | |

| | |

| V |

| grows downward |

| |

| |

| |

| |

| grows upward |

| ^ |

| | |

| | |

+----------------------------------+

| uninitialized data segment (BSS) |

+----------------------------------+

| initialized data segment |

+----------------------------------+

| code segment | 0x08048000 +----------------------------------+

| |

| |

| |

| |

| |

0 +----------------------------------+

In this project, the user stack is fixed in size, but in project 3 it will be allowed to grow. Traditionally, the size of the uninitialized data segment can be adjusted with a system call, but you will not have to implement this.

The code segment in Pintos starts at user virtual address 0x08084000, approximately 128 MB from the bottom of the address space. This value is specified in [SysV-i386] and has no deep significance.

The linker sets the layout of a user program in memory, as directed by a “linker script” that tells it the names and locations of the various program segments. You can learn more about linker scripts by reading the “Scripts” chapter in the linker manual, accessible via ‘info ld’.

To view the layout of a particular executable, run objdump (80*x*86) or i386-elf-objdump (SPARC) with the ‘-p’ option.

#### 3.1.5 Accessing User Memory

As part of a system call, the kernel must often access memory through pointers provided by a user program. The kernel must be very careful about doing so, because the user can pass a null pointer, a pointer to unmapped virtual memory, or a pointer to kernel virtual address space (above PHYS\_BASE). All of these types of invalid pointers must be rejected without harm to the kernel or other running processes, by terminating the offending process and freeing its resources.

There are at least two reasonable ways to do this correctly. The first method is to verify the validity of a user-provided pointer, then dereference it. If you choose this route, you’ll want to look at the functions in ‘userprog/pagedir.c’ and in ‘threads/vaddr.h’. This is the simplest way to handle user memory access.

The second method is to check only that a user pointer points below PHYS\_BASE, then dereference it. An invalid user pointer will cause a “page fault” that you can handle by modifying the code for page\_fault() in ‘userprog/exception.cc’. This technique is normally faster because it takes advantage of the processor’s MMU, so it tends to be used in real kernels (including Linux).

In either case, you need to make sure not to “leak” resources. For example, suppose that your system call has acquired a lock or allocated memory with malloc(). If you encounter an invalid user pointer afterward, you must still be sure to release the lock or free the page of memory. If you choose to verify user pointers before dereferencing them, this should be straightforward. It’s more difficult to handle if an invalid pointer causes a page fault, because there’s no way to return an error code from a memory access. Therefore, for those who want to try the latter technique, we’ll provide a little bit of helpful code:

/\* Reads a byte at user virtual address UADDR. UADDR must be below PHYS\_BASE.

Returns the byte value if successful, -1 if a segfault occurred. \*/

static int

get\_user (const uint8\_t \*uaddr)

{

int result;

asm ("movl $1f, 0; movzbl 1, 0; 1:"

: "=&a" (result) : "m" (\*uaddr)); return result;

}

/\* Writes BYTE to user address UDST. UDST must be below PHYS\_BASE.

Returns true if successful, false if a segfault occurred. \*/ static bool

put\_user (uint8\_t \*udst, uint8\_t byte)

{

int error\_code;

asm ("movl $1f, 0; movb b2, 1; 1:"

: "=&a" (error\_code), "=m" (\*udst) : "r" (byte)); return error\_code != -1;

}

Each of these functions assumes that the user address has already been verified to be

below PHYS\_BASE. They also assume that you’ve modified page\_fault() so that a page fault in the kernel merely sets eax to 0xffffffff and copies its former value into eip.

## Suggested Order of Implementation

We suggest first implementing the following, which can happen in parallel:

* + - Argument passing (see [Section 3.3.3 [Argument Passing], page 28](#_bookmark33)). Every user program will page fault immediately until argument passing is implemented.

For now, you may simply wish to change

\*esp = PHYS\_BASE;

to

\*esp = PHYS\_BASE - 12;

in setup\_stack(). That will work for any test program that doesn’t examine its arguments, although its name will be printed as (null).

Until you implement argument passing, you should only run programs without passing command-line arguments. Attempting to pass arguments to a program will include those arguments in the name of the program, which will probably fail.

* + - User memory access (see [Section 3.1.5 [Accessing User Memory], page 26](#_bookmark30)). All system calls need to read user memory. Few system calls need to write to user memory.
    - System call infrastructure (see [Section 3.3.4 [System Calls], page 28](#_bookmark34)). Implement enough code to read the system call number from the user stack and dispatch to a handler based on it.
    - The exit system call. Every user program that finishes in the normal way calls exit. Even a program that returns from main() calls exit indirectly (see \_start() in ‘lib/user/entry.c’).
    - The write system call for writing to fd 1, the system console. All of our test programs write to the console (the user process version of printf() is implemented this way), so they will all malfunction until write is available.
    - For now, change process\_wait() to an infinite loop (one that waits forever). The pro- vided implementation returns immediately, so Pintos will power off before any processes actually get to run. You will eventually need to provide a correct implementation.

After the above are implemented, user processes should work minimally. At the very least, they can write to the console and exit correctly. You can then refine your implemen- tation so that some of the tests start to pass.

## Requirements

#### Design Document

Before you turn in your project, you must copy the project 2 design document template into your source tree under the name ‘pintos/src/userprog/DESIGNDOC’ and fill it in. We recommend that you read the design document template before you start working on the project. See [Appendix D [Project Documentation], page 96](#_bookmark126), for a sample design document that goes along with a fictitious project.

#### Process Termination Messages

Whenever a user process terminates, because it called exit or for any other reason, print the process’s name and exit code, formatted as if printed by printf s: d)\n",

...);. The name printed should be the full name passed to process\_execute(), omitting command-line arguments. Do not print these messages when a kernel thread that is not a user process terminates, or when the halt system call is invoked. The message is optional when a process fails to load.

Aside from this, don’t print any other messages that Pintos as provided doesn’t already print. You may find extra messages useful during debugging, but they will confuse the grading scripts and thus lower your score.

#### Argument Passing

Currently, process\_execute() does not support passing arguments to new processes. Im- plement this functionality, by extending process\_execute() so that instead of simply tak- ing a program file name as its argument, it divides it into words at spaces. The first word is the program name, the second word is the first argument, and so on. That is, process\_ execute("grep foo bar") should run grep passing two arguments foo and bar.

Within a command line, multiple spaces are equivalent to a single space, so that process\_execute("grep foo bar") is equivalent to our original example. You can impose a reasonable limit on the length of the command line arguments. For example, you could limit the arguments to those that will fit in a single page (4 kB). (There is an unrelated limit of 128 bytes on command-line arguments that the pintos utility can pass to the kernel.)

You can parse argument strings any way you like. If you’re lost, look at strtok\_r(), pro- totyped in ‘lib/string.h’ and implemented with thorough comments in ‘lib/string.c’. You can find more about it by looking at the man page (run man strtok\_r at the prompt).

See [Section 3.5.1 [Program Startup Details], page 34](#_bookmark41), for information on exactly how you need to set up the stack.

#### System Calls

Implement the system call handler in ‘userprog/syscall.c’. The skeleton implementation we provide “handles” system calls by terminating the process. It will need to retrieve the system call number, then any system call arguments, and carry out appropriate actions.

Implement the following system calls. The prototypes listed are those seen by a user program that includes ‘lib/user/syscall.h’. (This header, and all others in ‘lib/user’, are for use by user programs only.) System call numbers for each system call are defined in ‘lib/syscall-nr.h’:

void halt (*void*) [System Call]

Terminates Pintos by calling power\_off() (declared in ‘threads/init.h’). This should be seldom used, because you lose some information about possible deadlock situations, etc.

void exit (*int status* )

[System Call]

Terminates the current user program, returning *status* to the kernel. If the process’s parent waits for it (see below), this is the status that will be returned. Conventionally, a *status* of 0 indicates success and nonzero values indicate errors.

pid\_t exec (*const char \*cmd\_line* )

[System Call]

Runs the executable whose name is given in *cmd line*, passing any given arguments, and returns the new process’s program id (pid). Must return pid -1, which otherwise should not be a valid pid, if the program cannot load or run for any reason.

int wait (*pid t pid* )

[System Call]

Waits for process *pid* to die and returns the status it passed to exit. Returns -1 if *pid* was terminated by the kernel (e.g. killed due to an exception). If *pid* is does not refer to a child of the calling thread, or if wait has already been successfully called for the given *pid*, returns -1 immediately, without waiting.

You must ensure that Pintos does not terminate until the initial process exits. The supplied Pintos code tries to do this by calling process\_wait() (in ‘userprog/process.c’) from main() (in ‘threads/init.c’). We suggest that you implement process\_wait() according to the comment at the top of the function and then implement the wait system call in terms of process\_wait().

All of a process’s resources, including its struct thread, must be freed whether its parent ever waits for it or not, and regardless of whether the child exits before or after its parent.

Children are not inherited: if *A* has child *B* and *B* has child *C*, then wait(C) always returns immediately when called from *A*, even if *B* is dead.

Consider all the ways a wait can occur: nested waits (*A* waits for *B*, then *B* waits for

*C*), multiple waits (*A* waits for *B*, then *A* waits for *C*), and so on.

Implementing this system call requires considerably more work than any of the rest.

bool create (*const char \*file, unsigned initial\_size* )

[System Call]

Creates a new file called *file* initially *initial size* bytes in size. Returns true if suc- cessful, false otherwise.

bool remove (*const char \*file* )

Deletes the file called *file*. Returns true if successful, false otherwise.

int open (*const char \*file* )

[System Call]

[System Call]

Opens the file called *file*. Returns a nonnegative integer handle called a “file descrip- tor” (fd), or -1 if the file could not be opened.

File descriptors numbered 0 and 1 are reserved for the console: fd 0 (STDIN\_FILENO) is standard input, fd 1 (STDOUT\_FILENO) is standard output. The open system call will never return either of these file descriptors, which are valid as system call arguments only as explicitly described below.

Each process has an independent set of file descriptors. File descriptors are not inherited by child processes.

When a single file is opened more than once, whether by a single process or different processes, each open returns a new file descriptor. Different file descriptors for a single

file are closed independently in separate calls to close and they do not share a file position.

int filesize (*int fd* )

Returns the size, in bytes, of the file open as *fd*.

int read (*int fd, void \*buffer, unsigned size* )

[System Call]

[System Call]

Reads *size* bytes from the file open as *fd* into *buffer*. Returns the number of bytes actually read (0 at end of file), or -1 if the file could not be read (due to a condition other than end of file). Fd 0 reads from the keyboard using input\_getc().

int write (*int fd, const void \*buffer, unsigned size* )

[System Call]

Writes *size* bytes from *buffer* to the open file *fd*. Returns the number of bytes actually written, or -1 if the file could not be written.

Writing past end-of-file would normally extend the file, but file growth is not imple- mented by the basic file system. The expected behavior is to write as many bytes as possible up to end-of-file and return the actual number written, or -1 if no bytes could be written at all.

Fd 1 writes to the console. Your code to write to the console should write all of *buffer* in one call to putbuf(), at least as long as *size* is not bigger than a few hundred bytes. (It is reasonable to break up larger buffers.) Otherwise, lines of text output by different processes may end up interleaved on the console, confusing both human readers and our grading scripts.

void seek (*int fd, unsigned position* )

[System Call]

Changes the next byte to be read or written in open file *fd* to *position*, expressed in bytes from the beginning of the file. (Thus, a *position* of 0 is the file’s start.)

A seek past the current end of a file is not an error. A later read obtains 0 bytes, indicating end of file. A later write extends the file, filling any unwritten gap with zeros. (However, in Pintos files have a fixed length until project 4 is complete, so writes past end of file will return an error.) These semantics are implemented in the file system and do not require any special effort in system call implementation.

unsigned tell (*int fd* )

[System Call]

Returns the position of the next byte to be read or written in open file *fd*, expressed in bytes from the beginning of the file.

void close (*int fd* )

[System Call]

Closes file descriptor *fd*. Exiting or terminating a process implicitly closes all its open file descriptors, as if by calling this function for each one.

The file defines other syscalls. Ignore them for now. You will implement some of them in project 3 and the rest in project 4, so be sure to design your system with extensibility in mind.

To implement syscalls, you need to provide ways to read and write data in user virtual address space. You need this ability before you can even obtain the system call number, because the system call number is on the user’s stack in the user’s virtual address space. This can be a bit tricky: what if the user provides an invalid pointer, a pointer into kernel memory,

or a block partially in one of those regions? You should handle these cases by terminating the user process. We recommend writing and testing this code before implementing any other system call functionality. See [Section 3.1.5 [Accessing User Memory], page 26](#_bookmark30), for more information.

You must synchronize system calls so that any number of user processes can make them at once. In particular, it is not safe to call into the file system code provided in the ‘filesys’ directory from multiple threads at once. Your system call implementation must treat the file system code as a critical section. Don’t forget that process\_execute() also accesses files. For now, we recommend against modifying code in the ‘filesys’ directory.

We have provided you a user-level function for each system call in ‘lib/user/syscall.c’. These provide a way for user processes to invoke each system call from a C program. Each uses a little inline assembly code to invoke the system call and (if appropriate) returns the system call’s return value.

When you’re done with this part, and forevermore, Pintos should be bulletproof. Nothing that a user program can do should ever cause the OS to crash, panic, fail an assertion, or otherwise malfunction. It is important to emphasize this point: our tests will try to break your system calls in many, many ways. You need to think of all the corner cases and handle them. The sole way a user program should be able to cause the OS to halt is by invoking the halt system call.

If a system call is passed an invalid argument, acceptable options include returning an error value (for those calls that return a value), returning an undefined value, or terminating the process.

See [Section 3.5.2 [System Call Details], page 36](#_bookmark43), for details on how system calls work.

#### Denying Writes to Executables

Add code to deny writes to files in use as executables. Many OSes do this because of the unpredictable results if a process tried to run code that was in the midst of being changed on disk. This is especially important once virtual memory is implemented in project 3, but it can’t hurt even now.

You can use file\_deny\_write() to prevent writes to an open file. Calling file\_allow\_ write() on the file will re-enable them (unless the file is denied writes by another opener). Closing a file will also re-enable writes. Thus, to deny writes to a process’s executable, you must keep it open as long as the process is still running.

## FAQ

How much code will I need to write?

Here’s a summary of our reference solution, produced by the diffstat program. The final row gives total lines inserted and deleted; a changed line counts as both an insertion and a deletion.

The reference solution represents just one possible solution. Many other so- lutions are also possible and many of those differ greatly from the reference solution. Some excellent solutions may not modify all the files modified by the reference solution, and some may modify files not modified by the reference solution.

threads/thread.c | 13 threads/thread.h | 26 + userprog/exception.c | 8

userprog/process.c | 247 ++++++++++++++-- userprog/syscall.c | 468 ++++++++++++++++++++++++++++++- userprog/syscall.h | 1

6 files changed, 725 insertions(+), 38 deletions(-)

The kernel always panics when I run pintos -p *file* -- -q.

Did you format the disk (with ‘pintos -f’)?

Is your file name too long? The file system limits file names to 14 characters. A command like ‘pintos -p ../../examples/echo -- -q’ will exceed the limit. Use ‘pintos -p ../../examples/echo -a echo -- -q’ to put the file under the name ‘echo’ instead.

Is the file system full?

Does the file system already contain 16 files? The base Pintos file system has a 16-file limit.

The file system may be so fragmented that there’s not enough contiguous space for your file.

When I run pintos -p ../file --, ‘file’ isn’t copied.

Files are written under the name you refer to them, by default, so in this case the file copied in would be named ‘../file’. You probably want to run pintos

-p ../file -a file -- instead.

All my user programs die with page faults.

This will happen if you haven’t implemented argument passing (or haven’t done so correctly). The basic C library for user programs tries to read *argc* and *argv* off the stack. If the stack isn’t properly set up, this causes a page fault.

All my user programs die with system call!

You’ll have to implement system calls before you see anything else. Every reasonable program tries to make at least one system call (exit()) and most programs make more than that. Notably, printf() invokes the write system call. The default system call handler just prints ‘system call!’ and terminates the program. Until then, you can use hex\_dump() to convince yourself that argument passing is implemented correctly (see [Section 3.5.1 [Program Startup](#_bookmark41) [Details], page 34](#_bookmark41)).

How can I can disassemble user programs?

The objdump (80*x*86) or i386-elf-objdump (SPARC) utility can disassem- ble entire user programs or object files. Invoke it as objdump -d *file* . You can use GDB’s disassemble command to disassemble individual functions (see [Section E.5 [GDB], page 102](#_bookmark132)).

Why do many C include files not work in Pintos programs? Can I use lib*foo* in my Pintos programs?

The C library we provide is very limited. It does not include many of the features that are expected of a real operating system’s C library. The C library

must be built specifically for the operating system (and architecture), since it must make system calls for I/O and memory allocation. (Not all functions do, of course, but usually the library is compiled as a unit.)

The chances are good that the library you want uses parts of the C library that Pintos doesn’t implement. It will probably take at least some porting effort to make it work under Pintos. Notably, the Pintos user program C library does not have a malloc() implementation.

How do I compile new user programs?

Modify ‘src/examples/Makefile’, then run make.

Can I run user programs under a debugger?

Yes, with some limitations. See [Section E.5.3 [Debugging User Programs],](#_bookmark135) [page 107](#_bookmark135).

What’s the difference between tid\_t and pid\_t?

A tid\_t identifies a kernel thread, which may have a user process running in it (if created with process\_execute()) or not (if created with thread\_ create()). It is a data type used only in the kernel.

A pid\_t identifies a user process. It is used by user processes and the kernel in the exec and wait system calls.

You can choose whatever suitable types you like for tid\_t and pid\_t. By default, they’re both int. You can make them a one-to-one mapping, so that the same values in both identify the same process, or you can use a more complex mapping. It’s up to you.

#### Argument Passing FAQ

Isn’t the top of stack in kernel virtual memory?

The top of stack is at PHYS\_BASE, typically 0xc0000000, which is also where kernel virtual memory starts. But before the processor pushes data on the stack, it decrements the stack pointer. Thus, the first (4-byte) value pushed on the stack will be at address 0xbffffffc.

Is PHYS\_BASE fixed?

No. You should be able to support PHYS\_BASE values that are any multiple of

0x10000000 from 0x80000000 to 0xf0000000, simply via recompilation.

#### System Calls FAQ

Can I just cast a struct file \* to get a file descriptor? Can I just cast a struct thread \* to a pid\_t?

You will have to make these design decisions yourself. Most operating systems do distinguish between file descriptors (or pids) and the addresses of their kernel data structures. You might want to give some thought as to why they do so before committing yourself.

Can I set a maximum number of open files per process?

It is better not to set an arbitrary limit. You may impose a limit of 128 open files per process, if necessary.

What happens when an open file is removed?

You should implement the standard Unix semantics for files. That is, when a file is removed any process which has a file descriptor for that file may continue to use that descriptor. This means that they can read and write from the file. The file will not have a name, and no other processes will be able to open it, but it will continue to exist until all file descriptors referring to the file are closed or the machine shuts down.

How can I run user programs that need more than 4 kB stack space?

You may modify the stack setup code to allocate more than one page of stack space for each process. In the next project, you will implement a better solution.

## 80*x*86 Calling Convention

This section summarizes important points of the convention used for normal function calls on 32-bit 80*x*86 implementations of Unix. Some details are omitted for brevity. If you do want all the details, refer to [SysV-i386].

The calling convention works like this:

* + 1. The caller pushes each of the function’s arguments on the stack one by one, normally using the PUSH assembly language instruction. Arguments are pushed in right-to-left order.

The stack grows downward: each push decrements the stack pointer, then stores into the location it now points to, like the C expression ‘\*--sp = *value* ’.

* + 1. The caller pushes the address of its next instruction (the *return address*) on the stack and jumps to the first instruction of the callee. A single 80*x*86 instruction, CALL, does both.
    2. The callee executes. When it takes control, the stack pointer points to the return address, the first argument is just above it, the second argument is just above the first argument, and so on.
    3. If the callee has a return value, it stores it into register EAX.
    4. The callee returns by popping the return address from the stack and jumping to the location it specifies, using the 80*x*86 RET instruction.
    5. The caller pops the arguments off the stack.

Consider a function f() that takes three int arguments. This diagram shows a sample stack frame as seen by the callee at the beginning of step 3 above, supposing that f() is invoked as f(1, 2, 3). The initial stack address is arbitrary:

+----------------+

0xbffffe7c | 3 |

0xbffffe78 | 2 |

0xbffffe74 | 1 | stack pointer --> 0xbffffe70 | return address |

+----------------+

#### Program Startup Details

The Pintos C library for user programs designates \_start(), in ‘lib/user/entry.c’, as the entry point for user programs. This function is a wrapper around main() that calls exit() if main() returns:

void

\_start (int argc, char \*argv[])

{

exit (main (argc, argv));

}

The kernel must put the arguments for the initial function on the stack before it allows the user program to begin executing. The arguments are passed in the same way as the normal calling convention (see [Section 3.5 [80x86 Calling Convention], page 34](#_bookmark40)).

Consider how to handle arguments for the following example command: ‘/bin/ls -l foo bar’. First, break the command into words: ‘/bin/ls’, ‘-l’, ‘foo’, ‘bar’. Place the words at the top of the stack. Order doesn’t matter, because they will be referenced through pointers.

Then, push the address of each string plus a null pointer sentinel, on the stack, in right- to-left order. These are the elements of argv. The order ensure that argv[0] is at the lowest virtual address. Word-aligned accesses are faster than unaligned accesses, so for best performance round the stack pointer down to a multiple of 4 before the first push.

Then, push argv (the address of argv[0]) and argc, in that order. Finally, push a fake “return address”: although the entry function will never return, its stack frame must have the same structure as any other.

The table below show the state of the stack and the relevant registers right before the beginning of the user program, assuming PHYS\_BASE is 0xc0000000:

|  |  |  |  |
| --- | --- | --- | --- |
| Address | Name | Data | Type |
| 0xbffffffc | argv[3][...] | ‘bar\0’ | char[4] |
| 0xbffffff8 | argv[2][...] | ‘foo\0’ | char[4] |
| 0xbffffff5 | argv[1][...] | ‘-l\0’ | char[3] |
| 0xbfffffed | argv[0][...] | ‘/bin/ls\0’ | char[8] |
| 0xbfffffec | word-align | 0 | uint8\_t |
| 0xbfffffe8 | argv[4] | 0 | char \* |
| 0xbfffffe4 | argv[3] | 0xbffffffc | char \* |
| 0xbfffffe0 | argv[2] | 0xbffffff8 | char \* |
| 0xbfffffdc | argv[1] | 0xbffffff5 | char \* |
| 0xbfffffd8 | argv[0] | 0xbfffffed | char \* |
| 0xbfffffd4 | argv | 0xbfffffd8 | char \*\* |
| 0xbfffffd0 | argc | 4 | int |
| 0xbfffffcc | return address | 0 | void (\*) () |

In this example, the stack pointer would be initialized to 0xbfffffcc.

As shown above, your code should start the stack at the very top of the user virtual address space, in the page just below virtual address PHYS\_BASE (defined in ‘threads/vaddr.h’).

You may find the non-standard hex\_dump() function, declared in ‘<stdio.h>’, useful for debugging your argument passing code. Here’s what it would show in the above example:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| bfffffc0 |  |  |  |  |  |  |  |  |  |  |  | 00 | 00 | 00 | 00 | | ....| |
| bfffffd0 | 04 | 00 | 00 | 00 | d8 | ff | ff | bf-ed | ff | ff | bf | f5 | ff | ff | bf | |................| |
| bfffffe0 | f8 | ff | ff | bf | fc | ff | ff | bf-00 | 00 | 00 | 00 | 00 | 2f | 62 | 69 | |............./bi| |
| bffffff0 | 6e | 2f | 6c | 73 | 00 | 2d | 6c | 00-66 | 6f | 6f | 00 | 62 | 61 | 72 | 00 | |n/ls.-l.foo.bar.| |

#### System Call Details

The first project already dealt with one way that the operating system can regain control from a user program: interrupts from timers and I/O devices. These are “external” inter- rupts, because they are caused by entities outside the CPU (see [Section A.4.3 [External](#_bookmark94) [Interrupt Handling], page 72](#_bookmark94)).

The operating system also deals with software exceptions, which are events that occur in program code (see [Section A.4.2 [Internal Interrupt Handling], page 71](#_bookmark93)). These can be errors such as a page fault or division by zero. Exceptions are also the means by which a user program can request services (“system calls”) from the operating system.

In the 80*x*86 architecture, the ‘int’ instruction is the most commonly used means for invoking system calls. This instruction is handled in the same way as other software ex- ceptions. In Pintos, user programs invoke ‘int $0x30’ to make a system call. The system call number and any additional arguments are expected to be pushed on the stack in the normal fashion before invoking the interrupt (see [Section 3.5 [80x86 Calling Convention],](#_bookmark40) [page 34](#_bookmark40)).

Thus, when the system call handler syscall\_handler() gets control, the system call number is in the 32-bit word at the caller’s stack pointer, the first argument is in the 32- bit word at the next higher address, and so on. The caller’s stack pointer is accessible to syscall\_handler() as the ‘esp’ member of the struct intr\_frame passed to it. (struct intr\_frame is on the kernel stack.)

The 80*x*86 convention for function return values is to place them in the EAX register. System calls that return a value can do so by modifying the ‘eax’ member of struct intr\_ frame.

You should try to avoid writing large amounts of repetitive code for implementing system calls. Each system call argument, whether an integer or a pointer, takes up 4 bytes on the stack. You should be able to take advantage of this to avoid writing much near-identical code for retrieving each system call’s arguments from the stack.

# Project 3: Virtual Memory

By now you should have some familiarity with the inner workings of Pintos. Your OS can properly handle multiple threads of execution with proper synchronization, and can load multiple user programs at once. However, the number and size of programs that can run is limited by the machine’s main memory size. In this assignment, you will remove that limitation.

You will build this assignment on top of the last one. Test programs from project 2 should also work with project 3. You should take care to fix any bugs in your project 2 submission before you start work on project 3, because those bugs will most likely cause the same problems in project 3.

You will continue to handle Pintos disks and file systems the same way you did in the previous assignment (see [Section 3.1.2 [Using the File System], page 22](#_bookmark26)).

## Background

#### Source Files

You will work in the ‘vm’ directory for this project. The ‘vm’ directory contains only ‘Makefile’s. The only change from ‘userprog’ is that this new ‘Makefile’ turns on the setting ‘-DVM’. All code you write will be in new files or in files introduced in earlier projects.

You will probably be encountering just a few files for the first time:

‘devices/disk.h’ ‘devices/disk.c’

Provides access to the physical disk, abstracting away the rather awful IDE interface. You will use this interface to access the swap disk.

#### Memory Terminology

Careful definitions are needed to keep discussion of virtual memory from being confus- ing. Thus, we begin by presenting some terminology for memory and storage. Some of these terms should be familiar from project 2 (see [Section 3.1.4 [Virtual Memory Layout],](#_bookmark28) [page 24](#_bookmark28)), but much of it is new.

#### Pages

A *page*, sometimes called a *virtual page*, is a continuous region of virtual memory 4,096 bytes (the *page size*) in length. A page must be *page-aligned*, that is, start on a virtual address evenly divisible by the page size. Thus, a 32-bit virtual address can be divided into a 20-bit *page number* and a 12-bit *page offset* (or just *offset*), like this:

31 12 11 0

+-------------------+-----------+

| Page Number | Offset |

+-------------------+-----------+

Virtual Address

Each process has an independent set of *user (virtual) pages*, which are those pages below virtual address PHYS\_BASE, typically 0xc0000000 (3 GB). The set of *kernel (virtual) pages*,

on the other hand, is global, remaining the same regardless of what thread or process is active. The kernel may access both user and kernel pages, but a user process may access only its own user pages. See [Section 3.1.4 [Virtual Memory Layout], page 24](#_bookmark28), for more information.

Pintos provides several useful functions for working with virtual addresses. See [Sec-](#_bookmark100) [tion A.6 [Virtual Addresses], page 75](#_bookmark100), for details.

#### Frames

A *frame*, sometimes called a *physical frame* or a *page frame*, is a continuous region of physical memory. Like pages, frames must be page-size and page-aligned. Thus, a 32-bit physical address can be divided into a 20-bit *frame number* and a 12-bit *frame offset* (or just *offset*), like this:

31 12 11 0

+-------------------+-----------+

| Frame Number | Offset |

+-------------------+-----------+

Physical Address

The 80*x*86 doesn’t provide any way to directly access memory at a physical address. Pintos works around this by mapping kernel virtual memory directly to physical memory: the first page of kernel virtual memory is mapped to the first frame of physical memory, the second page to the second frame, and so on. Thus, frames can be accessed through kernel virtual memory.

Pintos provides functions for translating between physical addresses and kernel virtual addresses. See [Section A.6 [Virtual Addresses], page 75](#_bookmark100), for details.

#### Page Tables

In Pintos, a *page table* is a data structure that the CPU uses to translate a virtual address to a physical address, that is, from a page to a frame. The page table format is dictated by the 80*x*86 architecture. Pintos provides page table management code in ‘pagedir.c’ (see [Section A.7 [Page Table], page 76](#_bookmark102)).

The diagram below illustrates the relationship between pages and frames. The virtual address, on the left, consists of a page number and an offset. The page table translates the page number into a frame number, which is combined with the unmodified offset to obtain the physical address, on the right.

+----------+

.--------------->|Page Table|-----------.

/ +----------+ |

0 | 12 11 0 0 V 12 11 0

+---------+----+ +---------+----+

|Page Nr | Ofs| |Frame Nr | Ofs|

+---------+----+ +---------+----+

Virt Addr | Phys Addr ^

\ /

#### Swap Slots

A *swap slot* is a continuous, page-size region of disk space on the swap disk. Although hardware limitations dictating the placement of slots are looser than for pages and frames, swap slots should be page-aligned because there is no downside in doing so.

#### Resource Management Overview

You will need to design the following data structures:

Supplemental page table

Enables page fault handling by supplementing the page table. See [Section 4.1.4](#_bookmark48) [[Managing the Supplemental Page Table], page 40](#_bookmark48).

Frame table

Swap table

Allows efficient implementation of eviction policy. See [Section 4.1.5 [Managing](#_bookmark49) [the Frame Table], page 40](#_bookmark49).

Tracks usage of swap slots. See [Section 4.1.6 [Managing the Swap Table],](#_bookmark52) [page 41](#_bookmark52).

Table of file mappings

Processes may map files into their virtual memory space. You need a table to track which files are mapped into which pages.

You do not necessarily need to implement four completely distinct data structures: it may be convenient to wholly or partially merge related resources into a unified data structure.

For each data structure, you need to determine what information each element should contain. You also need to decide on the data structure’s scope, either local (per-process) or global (applying to the whole system), and how many instances are required within its scope.

To simplify your design, you may store these data structures in non-pageable memory.

That means that you can be sure that pointers among them will remain valid.

Possible choices of data structures include arrays, lists, bitmaps, and hash tables. An array is often the simplest approach, but a sparsely populated array wastes memory. Lists are also simple, but traversing a long list to find a particular position wastes time. Both arrays and lists can be resized, but lists more efficiently support insertion and deletion in the middle.

Pintos includes a bitmap data structure in ‘lib/kernel/bitmap.c’ and ‘lib/kernel/bitmap.h’. A bitmap is an array of bits, each of which can be true or false. Bitmaps are typically used to track usage in a set of (identical) resources: if resource *n* is in use, then bit *n* of the bitmap is true. Pintos bitmaps are fixed in size, although you could extend their implementation to support resizing.

Pintos also includes a hash table data structure (see [Section A.8 [Hash Table], page 81](#_bookmark109)). Pintos hash tables efficiently support insertions and deletions over a wide range of table sizes.

Although more complex data structures may yield performance or other benefits, they may also needlessly complicate your implementation. Thus, we do not recommend imple- menting any advanced data structure (e.g. a balanced binary tree) as part of your design.

#### Managing the Supplemental Page Table

The *supplemental page table* supplements the page table with additional data about each page. It is needed because of the limitations imposed by the page table’s format. Such a data structure is often called a “page table” also; we add the word “supplemental” to reduce confusion.

The supplemental page table is used for at least two purposes. Most importantly, on a page fault, the kernel looks up the virtual page that faulted in the supplemental page table to find out what data should be there. Second, the kernel consults the supplemental page table when a process terminates, to decide what resources to free.

You may organize the supplemental page table as you wish. There are at least two basic approaches to its organization: in terms of segments or in terms of pages. Optionally, you may use the page table itself as an index to track the members of the supplemental page table. You will have to modify the Pintos page table implementation in ‘pagedir.c’ to do so. We recommend this approach for advanced students only. See [Section A.7.4.2 [Page](#_bookmark107) [Table Entry Format], page 80](#_bookmark107), for more information.

The most important user of the supplemental page table is the page fault handler. In project 2, a page fault always indicated a bug in the kernel or a user program. In project 3, this is no longer true. Now, a page fault might only indicate that the page must be brought in from a file or swap. You will have to implement a more sophisticated page fault handler to handle these cases. Your page fault handler, which you should implement by modifying page\_fault() in ‘threads/exception.c’, needs to do roughly the following:

* + - 1. Locate the page that faulted in the supplemental page table. If the memory reference is valid, use the supplemental page table entry to locate the data that goes in the page, which might be in the file system, or in a swap slot, or it might simply be an all-zero page. If you implement sharing, the page’s data might even already be in a page frame, but not in the page table.

If the page is unmapped, that is, if there’s no data there, or if the page lies within kernel virtual memory, or if the access is an attempt to write to a read-only page, then the access is invalid. Any invalid access terminates the process and thereby frees all of its resources.

* + - 1. Obtain a frame to store the page. See [Section 4.1.5 [Managing the Frame Table],](#_bookmark49) [page 40](#_bookmark49), for details.

If you implement sharing, the data you need may already be in a frame, in which case you must be able to locate that frame.

* + - 1. Fetch the data into the frame, by reading it from the file system or swap, zeroing it, etc.

If you implement sharing, the page you need may already be in a frame, in which case no action is necessary in this step.

* + - 1. Point the page table entry for the faulting virtual address to the physical page. You can use the functions in ‘userprog/pagedir.c’.

#### Managing the Frame Table

The *frame table* contains one entry for each frame that contains a user page. Each entry in the frame table contains a pointer to the page, if any, that currently occupies it, and other

data of your choice. The frame table allows Pintos to efficiently implement an eviction policy, by choosing a page to evict when no frames are free.

The frames used for user pages should be obtained from the “user pool,” by calling palloc\_get\_page(PAL\_USER). You must use PAL\_USER to avoid allocating from the “kernel pool,” which could cause some test cases to fail unexpectedly (see [[Why PAL USER?],](#_bookmark59) [page 46](#_bookmark59)). If you modify ‘palloc.c’ as part of your frame table implementation, be sure to retain the distinction between the two pools.

The most important operation on the frame table is obtaining an unused frame. This is easy when a frame is free. When none is free, a frame must be made free by evicting some page from its frame.

If no frame can be evicted without allocating a swap slot, but swap is full, panic the kernel. Real OSes apply a wide range of policies to recover from or prevent such situations, but these policies are beyond the scope of this project.

The process of eviction comprises roughly the following steps:

* + - 1. Choose a frame to evict, using your page replacement algorithm. The “accessed” and “dirty” bits in the page table, described below, will come in handy.
      2. Remove references to the frame from any page table that refers to it.

Unless you have implemented sharing, only a single page should refer to a frame at any given time.

* + - 1. If necessary, write the page to the file system or to swap. The evicted frame may then be used to store a different page.

#### 4.1.5.1 Accessed and Dirty Bits

80*x*86 hardware provides some assistance for implementing page replacement algorithms, through a pair of bits in the page table entry (PTE) for each page. On any read or write to a page, the CPU sets the *accessed bit* to 1 in the page’s PTE, and on any write, the CPU sets the *dirty bit* to 1. The CPU never resets these bits to 0, but the OS may do so.

You need to be aware of *aliases*, that is, two (or more) pages that refer to the same frame. When an aliased frame is accessed, the accessed and dirty bits are updated in only one page table entry (the one for the page used for access). The accessed and dirty bits for the other aliases are not updated.

In Pintos, every user virtual page is aliased to its kernel virtual page. You must manage these aliases somehow. For example, your code could check and update the accessed and dirty bits for both addresses. Alternatively, the kernel could avoid the problem by only accessing user data through the user virtual address.

Other aliases should only arise if you implement sharing for extra credit (see [[VM Extra](#_bookmark58) [Credit], page 46](#_bookmark58)), or if there is a bug in your code.

See [Section A.7.3 [Page Table Accessed and Dirty Bits], page 77](#_bookmark104), for details of the functions to work with accessed and dirty bits.

#### Managing the Swap Table

The swap table tracks in-use and free swap slots. It should allow picking an unused swap slot for evicting a page from its frame to the swap disk. It should allow freeing a swap slot when its page is read back or the process whose page was swapped is terminated.

You may use the disk on interface hd1:1 as the swap disk, using the disk interface prototyped in devices/disk.h. From the ‘vm/build’ directory, use the command pintos- mkdisk swap.dsk *n* to create an *n* MB swap disk named ‘swap.dsk’. Afterward, ‘swap.dsk’ will automatically be attached as hd1:1 when you run pintos. Alternatively, you can tell pintos to use a temporary *n*-MB swap disk for a single run with ‘--swap-disk=*n* ’.

Swap slots should be allocated lazily, that is, only when they are actually required by eviction. Reading data pages from the executable and writing them to swap immediately at process startup is not lazy. Swap slots should not be reserved to store particular pages.

Free a swap slot when its contents are read back into a frame.

#### Managing Memory Mapped Files

The file system is most commonly accessed with read and write system calls. A secondary interface is to “map” the file into virtual pages, using the mmap system call. The program can then use memory instructions directly on the file data.

Suppose file ‘foo’ is 0x1000 bytes (4 kB, or one page) long. If ‘foo’ is mapped into mem- ory starting at address 0x5000, then any memory accesses to locations 0x5000. . . 0x5fff will access the corresponding bytes of ‘foo’.

Here’s a program that uses mmap to print a file to the console. It opens the file specified on the command line, maps it at virtual address 0x10000000, writes the mapped data to the console (fd 1), and unmaps the file.

#include <stdio.h>

#include <syscall.h>

int main (int argc UNUSED, char \*argv[])

{

void \*data = (void \*) 0x10000000; /\* Address at which to map. \*/

int fd = open (argv[1]); /\* Open file. \*/ mapid\_t map = mmap (fd, data); /\* Map file. \*/

write (1, data, filesize (fd)); /\* Write file to console. \*/ munmap (map); /\* Unmap file (optional). \*/ return 0;

}

A similar program with full error handling is included as ‘mcat.c’ in the ‘examples’ directory, which also contains ‘mcp.c’ as a second example of mmap.

Your submission must be able to track what memory is used by memory mapped files. This is necessary to properly handle page faults in the mapped regions and to ensure that mapped files do not overlap any other segments within the process.

## Suggested Order of Implementation

We suggest the following initial order of implementation:

* + 1. Frame table (see [Section 4.1.5 [Managing the Frame Table], page 40](#_bookmark49)). Change ‘process.c’ to use your frame table allocator.

Do not implement swapping yet. If you run out of frames, fail the allocator or panic the kernel.

After this step, your kernel should still pass all the project 2 test cases.

* + 1. Supplemental page table and page fault handler (see [Section 4.1.4 [Managing the Sup-](#_bookmark48) [plemental Page Table], page 40](#_bookmark48)). Change ‘process.c’ to record the necessary infor- mation in the supplemental page table when loading an executable and setting up its stack. Implement loading of code and data segments in the page fault handler. For now, consider only valid accesses.

After this step, your kernel should pass all of the project 2 functionality test cases, but only some of the robustness tests.

From here, you can implement stack growth, mapped files, and page reclamation on process exit in parallel.

The next step is to implement eviction (see [Section 4.1.5 [Managing the Frame Table],](#_bookmark49) [page 40](#_bookmark49)). Initially you could choose the page to evict randomly. At this point, you need to consider how to manage accessed and dirty bits and aliasing of user and kernel pages. Synchronization is also a concern: how do you deal with it if process A faults on a page whose frame process B is in the process of evicting? Finally, implement a eviction strategy such as the clock algorithm.

## Requirements

This assignment is an open-ended design problem. We are going to say as little as possible about how to do things. Instead we will focus on what functionality we require your OS to support. We will expect you to come up with a design that makes sense. You will have the freedom to choose how to handle page faults, how to organize the swap disk, how to implement paging, etc.

#### Design Document

Before you turn in your project, you must copy the project 3 design document template into your source tree under the name ‘pintos/src/vm/DESIGNDOC’ and fill it in. We recommend that you read the design document template before you start working on the project. See [Appendix D [Project Documentation], page 96](#_bookmark126), for a sample design document that goes along with a fictitious project.

#### Paging

Implement paging for segments loaded from executables. All of these pages should be loaded lazily, that is, only as the kernel intercepts page faults for them. Upon eviction, pages modified since load (e.g. as indicated by the “dirty bit”) should be written to swap. Unmodified pages, including read-only pages, should never be written to swap because they can always be read back from the executable.

Implement a global page replacement algorithm that approximates LRU. Your algorithm should perform at least as well as the “second chance” or “clock” algorithm.

Your design should allow for parallelism. If one page fault requires I/O, in the meantime processes that do not fault should continue executing and other page faults that do not require I/O should be able to complete. This will require some synchronization effort.

You’ll need to modify the core of the program loader, which is the loop in load\_ segment() in ‘userprog/process.c’. Each time around the loop, page\_read\_bytes re- ceives the number of bytes to read from the executable file and page\_zero\_bytes receives

the number of bytes to initialize to zero following the bytes read. The two always sum to

PGSIZE (4,096). The handling of a page depends on these variables’ values:

* + - * If page\_read\_bytes equals PGSIZE, the page should be demand paged from disk on its first access.
      * If page\_zero\_bytes equals PGSIZE, the page does not need to be read from disk at all because it is all zeroes. You should handle such pages by creating a new page consisting of all zeroes at the first page fault.
      * Otherwise, neither page\_read\_bytes nor page\_zero\_bytes equals PGSIZE. In this case, an initial part of the page is to be read from disk and the remainder zeroed.

#### Stack Growth

Implement stack growth. In project 2, the stack was a single page at the top of the user virtual address space, and programs were limited to that much stack. Now, if the stack grows past its current size, allocate additional pages as necessary.

Allocate additional pages only if they “appear” to be stack accesses. Devise a heuristic that attempts to distinguish stack accesses from other accesses.

User programs are buggy if they write to the stack below the stack pointer, because typical real OSes may interrupt a process at any time to deliver a “signal,” which pushes data on the stack.1 However, the 80*x*86 PUSH instruction checks access permissions before it adjusts the stack pointer, so it may cause a page fault 4 bytes below the stack pointer. (Otherwise, PUSH would not be restartable in a straightforward fashion.) Similarly, the PUSHA instruction pushes 32 bytes at once, so it can fault 32 bytes below the stack pointer.

You will need to be able to obtain the current value of the user program’s stack pointer. Within a system call or a page fault generated by a user program, you can retrieve it from esp member of the struct intr\_frame passed to syscall\_handler() or page\_fault(), respectively. If you verify user pointers before accessing them (see [Section 3.1.5 [Accessing](#_bookmark30) [User Memory], page 26](#_bookmark30)), these are the only cases you need to handle. On the other hand, if you depend on page faults to detect invalid memory access, you will need to handle another case, where a page fault occurs in the kernel. Since the processor only saves the stack pointer when an exception causes a switch from user to kernel mode, reading esp out of the struct intr\_frame passed to page\_fault() would yield an undefined value, not the user stack pointer. You will need to arrange another way, such as saving esp into struct thread on the initial transition from user to kernel mode.

You should impose some absolute limit on stack size, as do most OSes. Some OSes make the limit user-adjustable, e.g. with the ulimit command on many Unix systems. On many GNU/Linux systems, the default limit is 8 MB.

The first stack page need not be allocated lazily. You can allocate and initialize it with the command line arguments at load time, with no need to wait for it to be faulted in.

All stack pages should be candidates for eviction. An evicted stack page should be written to swap.

1 This rule is common but not universal. One modern exception is the [*x*86-64 System V ABI](http://www.x86-64.org/documentation/abi.pdf), which

designates 128 bytes below the stack pointer as a “red zone” that may not be modified by signal or interrupt handlers.

#### Memory Mapped Files

Implement memory mapped files, including the following system calls.

mapid\_t mmap (*int fd, void \*addr* )

[System Call]

Maps the file open as *fd* into the process’s virtual address space. The entire file is mapped into consecutive virtual pages starting at *addr*.

Your VM system must lazily load pages in mmap regions and use the mmap’d file itself as backing store for the mapping. That is, evicting a page mapped by mmap writes it back to the file it was mapped from.

If the file’s length is not a multiple of PGSIZE, then some bytes in the final mapped page “stick out” beyond the end of the file. Set these bytes to zero when the page is faulted in from disk, and discard them when the page is written back to disk.

If successful, this function returns a “mapping ID” that uniquely identifies the map- ping within the process. On failure, it must return -1, which otherwise should not be a valid mapping id, and the process’s mappings must be unchanged.

A call to mmap may fail if the file open as *fd* has a length of zero bytes. It must fail if *addr* is not page-aligned or if the range of pages mapped overlaps any existing set of mapped pages, including the stack or pages mapped at executable load time. It must also fail if *addr* is 0, because some Pintos code assumes virtual page 0 is not mapped. Finally, file descriptors 0 and 1, representing console input and output, are not mappable.

void munmap (*mapid t mapping* )

[System Call]

Unmaps the mapping designated by *mapping*, which must be a mapping ID returned by a previous call to mmap by the same process that has not yet been unmapped.

All mappings are implicitly unmapped when a process exits, whether via exit or by any other means. When a mapping is unmapped, whether implicitly or explicitly, all pages written to by the process are written back to the file, and pages not written must not be. The pages are then removed from the process’s list of virtual pages.

Closing or removing a file does not unmap any of its mappings. Once created, a mapping is valid until munmap is called or the process exits, following the Unix convention. See [[Removing an Open File], page 34](#_bookmark39), for more information.

If two or more processes map the same file, there is no requirement that they see con- sistent data. Unix handles this by making the two mappings share the same physical page, but the mmap system call also has an argument allowing the client to specify whether the page is shared or private (i.e. copy-on-write).

## FAQ

How much code will I need to write?

Here’s a summary of our reference solution, produced by the diffstat program. The final row gives total lines inserted and deleted; a changed line counts as both an insertion and a deletion.

This summary is relative to the Pintos base code, but the reference solution for project 3 starts from the reference solution to project 2. See [Section 3.4](#_bookmark36) [[Project 2 FAQ], page 31](#_bookmark36), for the summary of project 2.

The reference solution represents just one possible solution. Many other so- lutions are also possible and many of those differ greatly from the reference solution. Some excellent solutions may not modify all the files modified by the reference solution, and some may modify files not modified by the reference solution.

|  |  |  |  |
| --- | --- | --- | --- |
| Makefile.build | | | 4 |  |
| devices/timer.c | | | 42 | ++ |
| threads/init.c | | | 5 |  |
| threads/interrupt.c | | | 2 |  |
| threads/thread.c | | | 31 | + |
| threads/thread.h | | | 37 | +- |
| userprog/exception.c | | | 12 |  |
| userprog/pagedir.c | | | 10 |  |
| userprog/process.c | | | 319 | +++++++++++++----- |
| userprog/syscall.c | | | 545 | ++++++++++++++++++++++++++++++- |
| userprog/syscall.h | | | 1 |  |
| vm/frame.c | | | 162 | +++++++++ |
| vm/frame.h | | | 23 | + |
| vm/page.c | | | 297 | ++++++++++++++++ |
| vm/page.h | | | 50 | ++ |
| vm/swap.c | | | 85 | ++++ |
| vm/swap.h | | | 11 |  |

17 files changed, 1532 insertions(+), 104 deletions(-)

Do we need a working Project 2 to implement Project 3?

Yes.

What extra credit is available?

You may implement sharing: when multiple processes are created that use the same executable file, share read-only pages among those processes instead of creating separate copies of read-only segments for each process. If you carefully designed your data structures, sharing of read-only pages should not make this part significantly harder.

Does the virtual memory system need to support data segment growth?

No. The size of the data segment is determined by the linker. We still have no dynamic allocation in Pintos (although it is possible to “fake” it at the user level by using memory-mapped files). Supporting data segment growth should add little additional complexity to a well-designed system.

Why should I use PAL\_USER for allocating page frames?

Passing PAL\_USER to palloc\_get\_page() causes it to allocate memory from the user pool, instead of the main kernel pool. Running out of pages in the user pool just causes user programs to page, but running out of pages in the kernel pool will cause many failures because so many kernel functions need to obtain memory. You can layer some other allocator on top of palloc\_get\_page() if you like, but it should be the underlying mechanism.

Also, you can use the ‘-ul’ option to pintos to limit the size of the user pool, which makes it easy to test your VM implementation with various user memory sizes.

# Project 4: File Systems

In the previous two assignments, you made extensive use of a file system without actually worrying about how it was implemented underneath. For this last assignment, you will im- prove the implementation of the file system. You will be working primarily in the ‘filesys’ directory.

You may build project 4 on top of project 2 or project 3. In either case, all of the functionality needed for project 2 must work in your filesys submission. If you build on project 3, then all of the project 3 functionality must work also, and you will need to edit ‘filesys/Make.vars’ to enable VM functionality. You can receive up to 5% extra credit if you do enable VM.

## Background

#### New Code

Here are some files that are probably new to you. These are in the ‘filesys’ directory except where indicated:

‘fsutil.c’

Simple utilities for the file system that are accessible from the kernel command line.

‘filesys.h’ ‘filesys.c’

Top-level interface to the file system. See [Section 3.1.2 [Using the File System],](#_bookmark26) [page 22](#_bookmark26), for an introduction.

‘directory.h’ ‘directory.c’

Translates file names to inodes. The directory data structure is stored as a file.

‘inode.h’

‘inode.c’ Manages the data structure representing the layout of a file’s data on disk.

‘file.h’

‘file.c’ Translates file reads and writes to disk sector reads and writes.

‘lib/kernel/bitmap.h’ ‘lib/kernel/bitmap.c’

A bitmap data structure along with routines for reading and writing the bitmap to disk files.

Our file system has a Unix-like interface, so you may also wish to read the Unix man pages for creat, open, close, read, write, lseek, and unlink. Our file system has calls that are similar, but not identical, to these. The file system translates these calls into disk operations.

All the basic functionality is there in the code above, so that the file system is usable from the start, as you’ve seen in the previous two projects. However, it has severe limitations which you will remove.

While most of your work will be in ‘filesys’, you should be prepared for interactions with all previous parts.

#### Testing File System Persistence

By now, you should be familiar with the basic process of running the Pintos tests. See [Section 1.2.1 [Testing], page 5](#_bookmark6), for review, if necessary.

Until now, each test invoked Pintos just once. However, an important purpose of a file system is to ensure that data remains accessible from one boot to another. Thus, the tests that are part of the file system project invoke Pintos a second time. The second run combines all the files and directories in the file system into a single file, then copies that file out of the Pintos file system into the host (Unix) file system.

The grading scripts check the file system’s correctness based on the contents of the file copied out in the second run. This means that your project will not pass any of the extended file system tests until the file system is implemented well enough to support tar, the Pintos user program that produces the file that is copied out. The tar program is fairly demanding (it requires both extensible file and subdirectory support), so this will take some work. Until then, you can ignore errors from make check regarding the extracted file system.

Incidentally, as you may have surmised, the file format used for copying out the file system contents is the standard Unix “tar” format. You can use the Unix tar program to examine them. The tar file for test *t* is named ‘*t*.tar’.

## Suggested Order of Implementation

We suggest implementing the parts of this project in the following order to make your job easier:

* + 1. Buffer cache (see [Section 5.3.4 [Buffer Cache], page 52](#_bookmark65)). Implement the buffer cache and integrate it into the existing file system. At this point all the tests from project 2 (and project 3, if you’re building on it) should still pass.
    2. Extensible files (see [Section 5.3.2 [Indexed and Extensible Files], page 50](#_bookmark62)). After this step, your project should pass the file growth tests.
    3. Subdirectories (see [Section 5.3.3 [Subdirectories], page 50](#_bookmark63)). Afterward, your project should pass the directory tests.
    4. Remaining miscellaneous items.

You can implement extensible files and subdirectories in parallel if you temporarily make the number of entries in new directories fixed.

You should think about synchronization throughout.

## Requirements

#### Design Document

Before you turn in your project, you must copy the project 4 design document template into your source tree under the name ‘pintos/src/filesys/DESIGNDOC’ and fill it in. We recommend that you read the design document template before you start working on the project. See [Appendix D [Project Documentation], page 96](#_bookmark126), for a sample design document that goes along with a fictitious project.

#### Indexed and Extensible Files

The basic file system allocates files as a single extent, making it vulnerable to external fragmentation, that is, it is possible that an *n*-block file cannot be allocated even though *n* blocks are free. Eliminate this problem by modifying the on-disk inode structure. In practice, this probably means using an index structure with direct, indirect, and doubly indirect blocks. You are welcome to choose a different scheme as long as you explain the rationale for it in your design documentation, and as long as it does not suffer from external fragmentation (as does the extent-based file system we provide).

You can assume that the disk will not be larger than 8 MB. You must support files as large as the disk (minus metadata). Each inode is stored in one disk sector, limiting the number of block pointers that it can contain. Supporting 8 MB files will require you to implement doubly-indirect blocks.

An extent-based file can only grow if it is followed by empty space, but indexed inodes make file growth possible whenever free space is available. Implement file growth. In the basic file system, the file size is specified when the file is created. In most modern file systems, a file is initially created with size 0 and is then expanded every time a write is made off the end of the file. Your file system must allow this.

There should be no predetermined limit on the size of a file, except that a file cannot exceed the size of the disk (minus metadata). This also applies to the root directory file, which should now be allowed to expand beyond its initial limit of 16 files.

User programs are allowed to seek beyond the current end-of-file (EOF). The seek itself does not extend the file. Writing at a position past EOF extends the file to the position being written, and any gap between the previous EOF and the start of the write must be filled with zeros. A read starting from a position past EOF returns no bytes.

Writing far beyond EOF can cause many blocks to be entirely zero. Some file systems allocate and write real data blocks for these implicitly zeroed blocks. Other file systems do not allocate these blocks at all until they are explicitly written. The latter file systems are said to support “sparse files.” You may adopt either allocation strategy in your file system.

#### Subdirectories

Implement a hierarchical name space. In the basic file system, all files live in a single directory. Modify this to allow directory entries to point to files or to other directories.

Make sure that directories can expand beyond their original size just as any other file can.

The basic file system has a 14-character limit on file names. You may retain this limit for individual file name components, or may extend it, at your option. You must allow full path names to be much longer than 14 characters.

Maintain a separate current directory for each process. At startup, set the root as the initial process’s current directory. When one process starts another with the exec system call, the child process inherits its parent’s current directory. After that, the two processes’ current directories are independent, so that either changing its own current directory has no effect on the other. (This is why, under Unix, the cd command is a shell built-in, not an external program.)

Update the existing system calls so that, anywhere a file name is provided by the caller, an absolute or relative path name may used. The directory separator character is forward

slash (‘/’). You must also support special file names ‘.’ and ‘..’, which have the same meanings as they do in Unix.

Update the open system call so that it can also open directories. Of the existing system calls, only close needs to accept a file descriptor for a directory.

Update the remove system call so that it can delete empty directories (other than the root) in addition to regular files. Directories may only be deleted if they do not contain any files or subdirectories (other than ‘.’ and ‘..’). You may decide whether to allow deletion of a directory that is open by a process or in use as a process’s current working directory. If it is allowed, then attempts to open files (including ‘.’ and ‘..’) or create new files in a deleted directory must be disallowed.

Implement the following new system calls:

bool chdir (*const char \*dir* )

[System Call]

Changes the current working directory of the process to *dir*, which may be relative or absolute. Returns true if successful, false on failure.

bool mkdir (*const char \*dir* )

[System Call]

Creates the directory named *dir*, which may be relative or absolute. Returns true if successful, false on failure. Fails if *dir* already exists or if any directory name in *dir*, besides the last, does not already exist. That is, mkdir("/a/b/c") succeeds only if ‘/a/b’ already exists and ‘/a/b/c’ does not.

bool readdir (*int fd, char \*name* )

[System Call]

Reads a directory entry from file descriptor *fd*, which must represent a directory. If successful, stores the null-terminated file name in *name*, which must have room for READDIR\_MAX\_LEN + 1 bytes, and returns true. If no entries are left in the directory, returns false.

‘.’ and ‘..’ should not be returned by readdir.

If the directory changes while it is open, then it is acceptable for some entries not to be read at all or to be read multiple times. Otherwise, each directory entry should be read once, in any order.

READDIR\_MAX\_LEN is defined in ‘lib/user/syscall.h’. If your file system supports longer file names than the basic file system, you should increase this value from the default of 14.

bool isdir (*int fd* )

[System Call]

Returns true if *fd* represents a directory, false if it represents an ordinary file.

int inumber (*int fd* )

[System Call]

Returns the *inode number* of the inode associated with *fd*, which may represent an ordinary file or a directory.

An inode number persistently identifies a file or directory. It is unique during the file’s existence. In Pintos, the sector number of the inode is suitable for use as an inode number.

We have provided ls and mkdir user programs, which are straightforward once the above syscalls are implemented. We have also provided pwd, which is not so straightforward. The shell program implements cd internally.

The pintos ‘put’ and ‘get’ commands should now accept full path names, assuming that the directories used in the paths have already been created. This should not require any significant extra effort on your part.

#### Buffer Cache

Modify the file system to keep a cache of file blocks. When a request is made to read or write a block, check to see if it is in the cache, and if so, use the cached data without going to disk. Otherwise, fetch the block from disk into cache, evicting an older entry if necessary. You are limited to a cache no greater than 64 sectors in size.

You must implement a cache replacement algorithm that is at least as good as the “clock” algorithm. We encourage you to account for the generally greater value of meta- data compared to data. Experiment to see what combination of accessed, dirty, and other information results in the best performance, as measured by the number of disk accesses.

You can keep a cached copy of the free map permanently in memory if you like. It doesn’t have to count against the cache size.

The provided inode code uses a “bounce buffer” allocated with malloc() to translate the disk’s sector-by-sector interface into the system call interface’s byte-by-byte interface. You should get rid of these bounce buffers. Instead, copy data into and out of sectors in the buffer cache directly.

Your cache should be *write-behind*, that is, keep dirty blocks in the cache, instead of immediately writing modified data to disk. Write dirty blocks to disk whenever they are evicted. Because write-behind makes your file system more fragile in the face of crashes, in addition you should periodically write all dirty, cached blocks back to disk. The cache should also be written back to disk in filesys\_done(), so that halting Pintos flushes the cache.

If you have timer\_sleep() from the first project working, write-behind is an excellent application. Otherwise, you may implement a less general facility, but make sure that it does not exhibit busy-waiting.

You should also implement *read-ahead*, that is, automatically fetch the next block of a file into the cache when one block of a file is read, in case that block is about to be read. Read-ahead is only really useful when done asynchronously. That means, if a process requests disk block 1 from the file, it should block until disk block 1 is read in, but once that read is complete, control should return to the process immediately. The read-ahead request for disk block 2 should be handled asynchronously, in the background.

**We recommend integrating the cache into your design early.** In the past, many groups have tried to tack the cache onto a design late in the design process. This is very difficult. These groups have often turned in projects that failed most or all of the tests.

#### Synchronization

The provided file system requires external synchronization, that is, callers must ensure that only one thread can be running in the file system code at once. Your submission must adopt a finer-grained synchronization strategy that does not require external synchronization. To the extent possible, operations on independent entities should be independent, so that they do not need to wait on each other.

Operations on different cache blocks must be independent. In particular, when I/O is required on a particular block, operations on other blocks that do not require I/O should proceed without having to wait for the I/O to complete.

Multiple processes must be able to access a single file at once. Multiple reads of a single file must be able to complete without waiting for one another. When writing to a file does not extend the file, multiple processes should also be able to write a single file at once. A read of a file by one process when the file is being written by another process is allowed to show that none, all, or part of the write has completed. (However, after the write system call returns to its caller, all subsequent readers must see the change.) Similarly, when two processes simultaneously write to the same part of a file, their data may be interleaved.

On the other hand, extending a file and writing data into the new section must be atomic. Suppose processes A and B both have a given file open and both are positioned at end-of-file. If A reads and B writes the file at the same time, A may read all, part, or none of what B writes. However, A may not read data other than what B writes, e.g. if B’s data is all nonzero bytes, A is not allowed to see any zeros.

Operations on different directories should take place concurrently. Operations on the same directory may wait for one another.

Keep in mind that only data shared by multiple threads needs to be synchronized. In the base file system, struct file and struct dir are accessed only by a single thread.

## FAQ

How much code will I need to write?

Here’s a summary of our reference solution, produced by the diffstat program. The final row gives total lines inserted and deleted; a changed line counts as both an insertion and a deletion.

This summary is relative to the Pintos base code, but the reference solution for project 4 is based on the reference solution to project 3. Thus, the reference solution runs with virtual memory enabled. See [Section 4.4 [Project 3 FAQ],](#_bookmark57) [page 45](#_bookmark57), for the summary of project 3.

The reference solution represents just one possible solution. Many other so- lutions are also possible and many of those differ greatly from the reference solution. Some excellent solutions may not modify all the files modified by the reference solution, and some may modify files not modified by the reference solution.

|  |  |  |  |
| --- | --- | --- | --- |
| Makefile.build | | | 5 |  |
| devices/timer.c | | | 42 | ++ |
| filesys/Make.vars | | | 6 |  |
| filesys/cache.c | | | 473 | +++++++++++++++++++++++++ |
| filesys/cache.h | | | 23 | + |
| filesys/directory.c | | | 99 | ++++- |
| filesys/directory.h | | | 3 |  |
| filesys/file.c | | | 4 |  |
| filesys/filesys.c | | | 194 | +++++++++- |
| filesys/filesys.h | | | 5 |  |
| filesys/free-map.c | | | 45 | +- |

|  |  |  |  |
| --- | --- | --- | --- |
| filesys/free-map.h | | | 4 |  |
| filesys/fsutil.c | | | 8 |
| filesys/inode.c | | | 444 | ++++++++++++++++++----- |
| filesys/inode.h | | | 11 |  |
| threads/init.c | | | 5 |  |
| threads/interrupt.c | | | 2 |  |
| threads/thread.c | | | 32 | + |
| threads/thread.h | | | 38 | +- |
| userprog/exception.c | | | 12 |  |
| userprog/pagedir.c | | | 10 |  |
| userprog/process.c | | | 332 | +++++++++++++---- |
| userprog/syscall.c | | | 582 | ++++++++++++++++++++++++++++++- |
| userprog/syscall.h | | | 1 |  |
| vm/frame.c | | | 161 | ++++++++ |
| vm/frame.h | | | 23 | + |
| vm/page.c | | | 297 | +++++++++++++++ |
| vm/page.h | | | 50 | ++ |
| vm/swap.c | | | 85 | ++++ |
| vm/swap.h | | | 11 |  |

30 files changed, 2721 insertions(+), 286 deletions(-)

**Can** DISK\_SECTOR\_SIZE **change?**

No, DISK\_SECTOR\_SIZE is fixed at 512. This is a fixed property of IDE disk hardware.

#### Indexed Files FAQ

What is the largest file size that we are supposed to support?

The disk we create will be 8 MB or smaller. However, individual files will have to be smaller than the disk to accommodate the metadata. You’ll need to consider this when deciding your inode organization.

#### Subdirectories FAQ

How should a file name like ‘a//b’ be interpreted?

Multiple consecutive slashes are equivalent to a single slash, so this file name is the same as ‘a/b’.

How about a file name like ‘/../x’?

The root directory is its own parent, so it is equivalent to ‘/x’.

How should a file name that ends in ‘/’ be treated?

Most Unix systems allow a slash at the end of the name for a directory, and reject other names that end in slashes. We will allow this behavior, as well as simply rejecting a name that ends in a slash.

#### Buffer Cache FAQ

**Can we keep a** struct inode\_disk **inside** struct inode**?**

The goal of the 64-block limit is to bound the amount of cached file system data. If you keep a block of disk data—whether file data or metadata—anywhere in

kernel memory then you have to count it against the 64-block limit. The same rule applies to anything that’s “similar” to a block of disk data, such as a struct inode\_disk without the length or sector\_cnt members.

That means you’ll have to change the way the inode implementation accesses its corresponding on-disk inode right now, since it currently just embeds a struct inode\_disk in struct inode and reads the corresponding sector from disk when it’s created. Keeping extra copies of inodes would subvert the 64-block limitation that we place on your cache.

You can store a pointer to inode data in struct inode, but it you do so you should carefully make sure that this does not limit your OS to 64 simultaneously open files. You can also store other information to help you find the inode when you need it. Similarly, you may store some metadata along each of your 64 cache entries.

You can keep a cached copy of the free map permanently in memory if you like. It doesn’t have to count against the cache size.

byte\_to\_sector() in ‘filesys/inode.c’ uses the struct inode\_disk directly, without first reading that sector from wherever it was in the storage hierarchy. This will no longer work. You will need to change inode\_byte\_to\_sector() to obtain the struct inode\_disk from the cache before using it.

# Appendix A Reference Guide

This chapter is a reference for the Pintos code. It covers the entire code base, but you’ll only be using Pintos one part at a time, so you may find that you want to read each part as you work on the project where it becomes important.

(Actually, the reference guide is currently incomplete.)

We recommend using “tags” to follow along with references to function and variable names (see [Section F.1 [Tags], page 110](#_bookmark140)).

## Loading

This section covers the Pintos loader and basic kernel initialization.

#### The Loader

The first part of Pintos that runs is the loader, in ‘threads/loader.S’. The PC BIOS loads the loader into memory. The loader, in turn, is responsible for initializing the CPU, loading the rest of Pintos into memory, and then jumping to its start. It’s not important to understand exactly what the loader does, but if you’re interested, read on. You should probably read along with the loader’s source. You should also understand the basics of the 80*x*86 architecture as described by chapter 3, “Basic Execution Environment,” of [IA32-v1].

Because the PC BIOS loads the loader, the loader has to play by the BIOS’s rules. In particular, the BIOS only loads 512 bytes (one disk sector) into memory. This is a severe restriction and it means that, practically speaking, the loader has to be written in assembly language.

The Pintos loader first initializes the CPU. The first important part of this is to enable 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 20th power) will fail. Pintos wants to access more memory than this, so we have to enable it.

Next, the loader asks the BIOS for the PC’s memory size. Again for historical reasons, the function that we call in the BIOS to do this can only detect up to 64 MB of RAM, so that’s the practical limit that Pintos can support. The memory size is stashed away in a location in the loader that the kernel can read after it boots.

Third, the loader creates a basic page table. This page table maps the 64 MB at the base of virtual memory (starting at virtual address 0) directly to the identical physical addresses. It also maps the same physical memory starting at virtual address LOADER\_ PHYS\_BASE, which defaults to 0xc0000000 (3 GB). The Pintos kernel only wants the latter mapping, but there’s a chicken-and-egg problem if we don’t include the former: our current virtual address is roughly 0x7c00, the location where the BIOS loaded us, and we can’t jump to 0xc0007c00 until we turn on the page table, but if we turn on the page table without jumping there, then we’ve just pulled the rug out from under ourselves.

After the page table is initialized, we load the CPU’s control registers to turn on protected mode and paging, and then we set up the segment registers. We aren’t yet equipped to handle interrupts in protected mode, so we disable interrupts.

Finally it’s time to load the kernel from disk. We use a simple but inflexible method to do this: we program the IDE disk controller directly. We assume that the kernel is stored

starting from the second sector of the first IDE disk (the first sector normally contains the boot loader). We also assume that the BIOS has already set up the IDE controller for us. We read KERNEL\_LOAD\_PAGES pages of data (4 kB per page) from the disk directly into virtual memory, starting LOADER\_KERN\_BASE bytes past LOADER\_PHYS\_BASE, which by default means that we load the kernel starting 1 MB into physical memory.

Then we jump to the start of the compiled kernel image. Using the “linker script” in ‘threads/kernel.lds.S’, the kernel has arranged to begin with the assembly module ‘threads/start.S’. This assembly module just calls main(), which never returns.

There’s one more trick: the Pintos kernel command line is in stored the boot loader. The pintos program actually modifies a copy of the boot loader on disk each time it runs the kernel, putting in whatever command line arguments the user supplies to the kernel, and then the kernel at boot time reads those arguments out of the boot loader in memory. This is not an elegant solution, but it is simple and effective.

#### Kernel Initialization

The kernel proper starts with the main() function. The main() function is written in C, as will be most of the code we encounter in Pintos from here on out.

When main() starts, the system is in a pretty raw state. We’re in 32-bit protected mode with paging enabled, but hardly anything else is ready. Thus, the main() function consists primarily of calls into other Pintos modules’ initialization functions. These are usually named *module* \_init(), where *module* is the module’s name, ‘*module*.c’ is the module’s source code, and ‘*module*.h’ is the module’s header.

First we initialize kernel RAM in ram\_init(). The first step is to clear out the kernel’s so-called “BSS” segment. The BSS is a segment that should be initialized to all zeros. In most C implementations, whenever you declare a variable outside a function without providing an initializer, that variable goes into the BSS. Because it’s all zeros, the BSS isn’t stored in the image that the loader brought into memory. We just use memset() to zero it out. The other task of ram\_init() is to read out the machine’s memory size from where the loader stored it and put it into the ram\_pages variable for later use.

Next, main() calls read\_command\_line() to break the kernel command line into argu- ments, then parse\_options() to read any options at the beginning of the command line. (Actions specified on the command line execute later.)

thread\_init() initializes the thread system. We will defer full discussion to our dis- cussion of Pintos threads below. It is called so early in initialization because a valid thread structure is a prerequisite for acquiring a lock, and lock acquisition in turn is important to other Pintos subsystems. Then we initialize the console and print a startup message to the console.

The next block of functions we call initialize the kernel’s memory system. palloc\_ init() sets up the kernel page allocator, which doles out memory one or more pages at a time (see [Section A.5.1 [Page Allocator], page 73](#_bookmark96)). malloc\_init() sets up the allocator that handles allocations of arbitrary-size blocks of memory (see [Section A.5.2 [Block Allocator],](#_bookmark98) [page 74](#_bookmark98)). paging\_init() sets up a page table for the kernel (see [Section A.7 [Page Table],](#_bookmark102) [page 76](#_bookmark102)).

In projects 2 and later, main() also calls tss\_init() and gdt\_init().

The next set of calls initializes the interrupt system. intr\_init() sets up the CPU’s *interrupt descriptor table* (IDT) to ready it for interrupt handling (see [Section A.4.1 [In-](#_bookmark91) [terrupt Infrastructure], page 70](#_bookmark91)), then timer\_init() and kbd\_init() prepare for handling timer interrupts and keyboard interrupts, respectively. input\_init() sets up to merge se- rial and keyboard input into one stream. In projects 2 and later, we also prepare to handle interrupts caused by user programs using exception\_init() and syscall\_init().

Now that interrupts are set up, we can start the scheduler with thread\_start(), which creates the idle thread and enables interrupts. With interrupts enabled, interrupt-driven serial port I/O becomes possible, so we use serial\_init\_queue() to switch to that mode. Finally, timer\_calibrate() calibrates the timer for accurate short delays.

If the file system is compiled in, as it will starting in project 2, we initialize the disks with disk\_init(), then the file system with filesys\_init().

Boot is complete, so we print a message.

Function run\_actions() now parses and executes actions specified on the kernel com- mand line, such as run to run a test (in project 1) or a user program (in later projects).

Finally, if ‘-q’ was specified on the kernel command line, we call power\_off() to ter- minate the machine simulator. Otherwise, main() calls thread\_exit(), which allows any other running threads to continue running.

## Threads

* + 1. struct thread

The main Pintos data structure for threads is struct thread, declared in ‘threads/thread.h’.

struct thread [Structure]

Represents a thread or a user process. In the projects, you will have to add your own members to struct thread. You may also change or delete the definitions of existing members.

Every struct thread occupies the beginning of its own page of memory. The rest of the page is used for the thread’s stack, which grows downward from the end of the page. It looks like this:

4 kB +---------------------------------+

sizeof (struct thread)

|  |  |  |
| --- | --- | --- |
| | | kernel stack | | |
| | | | | | |
| | | | | | |
| | | V | | |
| | | grows downward | | |
| | |  | | |
| | |  | | |
| | |  | | |
| | |  | | |
| | |  | | |
| | |  | | |
| | |  | | |
| | |  | | |
| +---------------------------------+ | | |
| | | magic | | |
| | | : | | |
| | | : | | |
| | | status | | |
| | | tid | | |

0 kB +---------------------------------+

This has two consequences. First, struct thread must not be allowed to grow too big. If it does, then there will not be enough room for the kernel stack. The base struct thread is only a few bytes in size. It probably should stay well under 1 kB.

Second, kernel stacks must not be allowed to grow too large. If a stack overflows, it will corrupt the thread state. Thus, kernel functions should not allocate large structures or arrays as non-static local variables. Use dynamic allocation with malloc() or palloc\_get\_page() instead (see [Section A.5 [Memory Allocation], page 73](#_bookmark95)).

tid\_t tid [Member of struct thread]

The thread’s thread identifier or *tid*. Every thread must have a tid that is unique over the entire lifetime of the kernel. By default, tid\_t is a typedef for int and each new thread receives the numerically next higher tid, starting from 1 for the initial process. You can change the type and the numbering scheme if you like.

##### enum thread\_status status

The thread’s state, one of the following:

[Member of struct thread]

THREAD\_RUNNING [Thread State]

The thread is running. Exactly one thread is running at a given time. thread\_ current() returns the running thread.

THREAD\_READY [Thread State]

The thread is ready to run, but it’s not running right now. The thread could be selected to run the next time the scheduler is invoked. Ready threads are kept in a doubly linked list called ready\_list.

THREAD\_BLOCKED [Thread State]

The thread is waiting for something, e.g. a lock to become available, an inter- rupt to be invoked. The thread won’t be scheduled again until it transitions to the THREAD\_READY state with a call to thread\_unblock(). This is most conve- niently done indirectly, using one of the Pintos synchronization primitives that block and unblock threads automatically (see [Section A.3 [Synchronization],](#_bookmark81) [page 63](#_bookmark81)).

There is no *a priori* way to tell what a blocked thread is waiting for, but a backtrace can help (see [Section E.4 [Backtraces], page 100](#_bookmark130)).

THREAD\_DYING [Thread State]

The thread will be destroyed by the scheduler after switching to the next thread.

char name[16] [Member of struct thread]

The thread’s name as a string, or at least the first few characters of it.

uint8\_t \* stack [Member of struct thread]

Every thread has its own stack to keep track of its state. When the thread is running, the CPU’s stack pointer register tracks the top of the stack and this member is unused. But when the CPU switches to another thread, this member saves the thread’s stack pointer. No other members are needed to save the thread’s registers, because the other registers that must be saved are saved on the stack.

When an interrupt occurs, whether in the kernel or a user program, an struct intr\_ frame is pushed onto the stack. When the interrupt occurs in a user program, the struct intr\_frame is always at the very top of the page. See [Section A.4 [Interrupt](#_bookmark90) [Handling], page 69](#_bookmark90), for more information.

int priority [Member of struct thread]

A thread priority, ranging from PRI\_MIN (0) to PRI\_MAX (63). Lower numbers corre- spond to lower priorities, so that priority 0 is the lowest priority and priority 63 is the highest. Pintos as provided ignores thread priorities, but you will implement priority scheduling in project 1 (see [Section 2.2.3 [Priority Scheduling], page 15](#_bookmark18)).

struct list\_elem elem [Member of struct thread]

A “list element” used to put the thread into doubly linked lists, either ready\_list (the list of threads ready to run) or a list of threads waiting on a semaphore in sema\_ down(). It can do double duty because a thread waiting on a semaphore is not ready, and vice versa.

uint32\_t \* pagedir [Member of struct thread]

Only present in project 2 and later. See [Section 4.1.2.3 [Page Tables], page 38](#_bookmark46).

unsigned magic [Member of struct thread]

Always set to THREAD\_MAGIC, which is just an arbitrary number defined in ‘threads/thread.c’, and used to detect stack overflow. thread\_current() checks that the magic member of the running thread’s struct thread is set to THREAD\_MAGIC. Stack overflow tends to change this value, triggering the assertion. For greatest benefit, as you add members to struct thread, leave magic at the end.

* + 1. **Thread Functions**

‘threads/thread.c’ implements several public functions for thread support. Let’s take a look at the most useful:

void thread\_init (*void*) [Function]

Called by main() to initialize the thread system. Its main purpose is to create a struct thread for Pintos’s initial thread. This is possible because the Pintos loader puts the initial thread’s stack at the top of a page, in the same position as any other Pintos thread.

Before thread\_init() runs, thread\_current() will fail because the running thread’s magic value is incorrect. Lots of functions call thread\_current() directly or indi- rectly, including lock\_acquire() for locking a lock, so thread\_init() is called early in Pintos initialization.

void thread\_start (*void*) [Function]

Called by main() to start the scheduler. Creates the idle thread, that is, the thread that is scheduled when no other thread is ready. Then enables interrupts, which as a side effect enables the scheduler because the scheduler runs on return from the timer interrupt, using intr\_yield\_on\_return() (see [Section A.4.3 [External](#_bookmark94) [Interrupt Handling], page 72](#_bookmark94)).

void thread\_tick (*void*) [Function]

Called by the timer interrupt at each timer tick. It keeps track of thread statistics and triggers the scheduler when a time slice expires.

##### void thread\_print\_stats (*void*)

Called during Pintos shutdown to print thread statistics.

tid\_t thread\_create (*const char \*name, int priority, thread func*

*\*func, void \*aux* )

[Function]

[Function]

Creates and starts a new thread named *name* with the given *priority*, returning the new thread’s tid. The thread executes *func*, passing *aux* as the function’s single argument.

thread\_create() allocates a page for the thread’s struct thread and stack and initializes its members, then it sets up a set of fake stack frames for it (see [Section A.2.3](#_bookmark79) [[Thread Switching], page 62](#_bookmark79)). The thread is initialized in the blocked state, then unblocked just before returning, which allows the new thread to be scheduled (see [[Thread States], page 59](#_bookmark76)).

##### void thread\_func (void \**aux* )

[Type]

This is the type of the function passed to thread\_create(), whose *aux* argu- ment is passed along as the function’s argument.

void thread\_block (*void*) [Function]

Transitions the running thread from the running state to the blocked state (see [[Thread States], page 59](#_bookmark76)). The thread will not run again until thread\_unblock() is called on it, so you’d better have some way arranged for that to happen. Because thread\_block() is so low-level, you should prefer to use one of the synchronization primitives instead (see [Section A.3 [Synchronization], page 63](#_bookmark81)).

void thread\_unblock (*struct thread \*thread* )

[Function]

Transitions *thread*, which must be in the blocked state, to the ready state, allowing it to resume running (see [[Thread States], page 59](#_bookmark76)). This is called when the event that the thread is waiting for occurs, e.g. when the lock that the thread is waiting on becomes available.

##### struct thread \* thread\_current (*void*)

Returns the running thread.

tid\_t thread\_tid (*void*)

[Function]

[Function]

Returns the running thread’s thread id. Equivalent to thread\_current ()->tid.

const char \* thread\_name (*void*) [Function]

Returns the running thread’s name. Equivalent to thread\_current ()->name.

void thread\_exit (*void*) NO\_RETURN [Function]

Causes the current thread to exit. Never returns, hence NO\_RETURN (see [Section E.3](#_bookmark128) [[Function and Parameter Attributes], page 99](#_bookmark128)).

void thread\_yield (*void*) [Function]

Yields the CPU to the scheduler, which picks a new thread to run. The new thread might be the current thread, so you can’t depend on this function to keep this thread from running for any particular length of time.

##### int thread\_get\_priority (*void*)

void thread\_set\_priority (*int new\_priority* )

[Function] [Function]

Stub to set and get thread priority. See [Section 2.2.3 [Priority Scheduling], page 15](#_bookmark18).

int thread\_get\_nice (*void*)

void thread\_set\_nice (*int new\_nice* ) int thread\_get\_recent\_cpu (*void*) int thread\_get\_load\_avg (*void*)

[Function] [Function] [Function] [Function]

Stubs for the advanced scheduler. See [Appendix B [4.4BSD Scheduler], page 88](#_bookmark117).

#### Thread Switching

schedule() is responsible for switching threads. It is internal to ‘threads/thread.c’ and called only by the three public thread functions that need to switch threads: thread\_block(), thread\_exit(), and thread\_yield(). Before any of these functions call schedule(), they disable interrupts (or ensure that they are already disabled) and then change the running thread’s state to something other than running.

schedule() is short but tricky. It records the current thread in local variable *cur*, determines the next thread to run as local variable *next* (by calling next\_thread\_to\_ run()), and then calls switch\_threads() to do the actual thread switch. The thread we switched to was also running inside switch\_threads(), as are all the threads not currently running, so the new thread now returns out of switch\_threads(), returning the previously running thread.

switch\_threads() is an assembly language routine in ‘threads/switch.S’. It saves registers on the stack, saves the CPU’s current stack pointer in the current struct thread’s

stack member, restores the new thread’s stack into the CPU’s stack pointer, restores registers from the stack, and returns.

The rest of the scheduler is implemented in schedule\_tail(). It marks the new thread as running. If the thread we just switched from is in the dying state, then it also frees the page that contained the dying thread’s struct thread and stack. These couldn’t be freed prior to the thread switch because the switch needed to use it.

Running a thread for the first time is a special case. When thread\_create() creates a new thread, it goes through a fair amount of trouble to get it started properly. In particular, the new thread hasn’t started running yet, so there’s no way for it to be running inside switch\_threads() as the scheduler expects. To solve the problem, thread\_create() creates some fake stack frames in the new thread’s stack:

* + - * The topmost fake stack frame is for switch\_threads(), represented by struct switch\_threads\_frame. The important part of this frame is its eip member, the return address. We point eip to switch\_entry(), indicating it to be the function that called switch\_entry().
      * The next fake stack frame is for switch\_entry(), an assembly language routine in ‘threads/switch.S’ that adjusts the stack pointer,1 calls schedule\_tail() (this special case is why schedule\_tail() is separate from schedule()), and returns. We fill in its stack frame so that it returns into kernel\_thread(), a function in ‘threads/thread.c’.
      * The final stack frame is for kernel\_thread(), which enables interrupts and calls the thread’s function (the function passed to thread\_create()). If the thread’s function returns, it calls thread\_exit() to terminate the thread.

## Synchronization

If sharing of resources between threads is not handled in a careful, controlled fashion, the result is usually a big mess. This is especially the case in operating system kernels, where faulty sharing can crash the entire machine. Pintos provides several synchronization primitives to help out.

#### Disabling Interrupts

The crudest way to do synchronization is to disable interrupts, that is, to temporarily prevent the CPU from responding to interrupts. If interrupts are off, no other thread will preempt the running thread, because thread preemption is driven by the timer interrupt. If interrupts are on, as they normally are, then the running thread may be preempted by another at any time, whether between two C statements or even within the execution of one.

Incidentally, this means that Pintos is a “preemptible kernel,” that is, kernel threads can be preempted at any time. Traditional Unix systems are “nonpreemptible,” that is, kernel threads can only be preempted at points where they explicitly call into the scheduler. (User programs can be preempted at any time in both models.) As you might imagine, preemptible kernels require more explicit synchronization.

1 This is because switch\_threads() takes arguments on the stack and the 80*x*86 SVR4 calling convention requires the caller, not the called function, to remove them when the call is complete. See [SysV-i386] chapter 3 for details.

You should have little need to set the interrupt state directly. Most of the time you should use the other synchronization primitives described in the following sections. The main reason to disable interrupts is to synchronize kernel threads with external interrupt handlers, which cannot sleep and thus cannot use most other forms of synchronization (see [Section A.4.3 [External Interrupt Handling], page 72](#_bookmark94)).

Some external interrupts cannot be postponed, even by disabling interrupts. These inter- rupts, called *non-maskable interrupts* (NMIs), are supposed to be used only in emergencies,

e.g. when the computer is on fire. Pintos does not handle non-maskable interrupts.

Types and functions for disabling and enabling interrupts are in ‘threads/interrupt.h’.

enum intr\_level [Type]

One of INTR\_OFF or INTR\_ON, denoting that interrupts are disabled or enabled, re- spectively.

##### enum intr\_level intr\_get\_level (*void*)

Returns the current interrupt state.

enum intr\_level intr\_set\_level (*enum intr level level* )

[Function]

[Function]

Turns interrupts on or off according to *level*. Returns the previous interrupt state.

##### enum intr\_level intr\_enable (*void*)

Turns interrupts on. Returns the previous interrupt state.

[Function]

##### enum intr\_level intr\_disable (*void*)

Turns interrupts off. Returns the previous interrupt state.

#### Semaphores

[Function]

A *semaphore* is a nonnegative integer together with two operators that manipulate it atom- ically, which are:

* + - * “Down” or “P”: wait for the value to become positive, then decrement it.
      * “Up” or “V”: increment the value (and wake up one waiting thread, if any).

A semaphore initialized to 0 may be used to wait for an event that will happen exactly once. For example, suppose thread *A* starts another thread *B* and wants to wait for *B* to signal that some activity is complete. *A* can create a semaphore initialized to 0, pass it to *B* as it starts it, and then “down” the semaphore. When *B* finishes its activity, it “ups” the semaphore. This works regardless of whether *A* “downs” the semaphore or *B* “ups” it first.

A semaphore initialized to 1 is typically used for controlling access to a resource. Before a block of code starts using the resource, it “downs” the semaphore, then after it is done with the resource it “ups” the resource. In such a case a lock, described below, may be more appropriate.

Semaphores can also be initialized to values larger than 1. These are rarely used.

Semaphores were invented by Edsger Dijkstra and first used in the THE operating system ([Dijkstra]).

Pintos’ semaphore type and operations are declared in ‘threads/synch.h’.

##### struct semaphore

Represents a semaphore.

[Type]

void sema\_init (*struct semaphore \*sema, unsigned value* )

Initializes *sema* as a new semaphore with the given initial *value*.

[Function]

void sema\_down (*struct semaphore \*sema* )

[Function]

Executes the “down” or “P” operation on *sema*, waiting for its value to become positive and then decrementing it by one.

bool sema\_try\_down (*struct semaphore \*sema* )

[Function]

Tries to execute the “down” or “P” operation on *sema*, without waiting. Returns true if *sema* was successfully decremented, or false if it was already zero and thus could not be decremented without waiting. Calling this function in a tight loop wastes CPU time, so use sema\_down() or find a different approach instead.

void sema\_up (*struct semaphore \*sema* )

[Function]

Executes the “up” or “V” operation on *sema*, incrementing its value. If any threads are waiting on *sema*, wakes one of them up.

Unlike most synchronization primitives, sema\_up() may be called inside an external interrupt handler (see [Section A.4.3 [External Interrupt Handling], page 72](#_bookmark94)).

Semaphores are internally built out of disabling interrupt (see [Section A.3.1 [Disabling](#_bookmark82) [Interrupts], page 63](#_bookmark82)) and thread blocking and unblocking (thread\_block() and thread\_ unblock()). Each semaphore maintains a list of waiting threads, using the linked list implementation in ‘lib/kernel/list.c’.

#### Locks

A *lock* is like a semaphore with an initial value of 1 (see [Section A.3.2 [Semaphores], page 64](#_bookmark84)). A lock’s equivalent of “up” is called “acquire”, and the “down” operation is called “release”.

Compared to a semaphore, a lock has one added restriction: only the thread that acquires a lock, called the lock’s “owner”, is allowed to release it. If this restriction is a problem, it’s a good sign that a semaphore should be used, instead of a lock.

Locks in Pintos are not “recursive,” that is, it is an error for the thread currently holding a lock to try to acquire that lock.

Lock types and functions are declared in ‘threads/synch.h’.

##### struct lock

Represents a lock.

[Type]

void lock\_init (*struct lock \*lock* )

[Function]

Initializes *lock* as a new lock. The lock is not initially owned by any thread.

void lock\_acquire (*struct lock \*lock* )

[Function]

Acquires *lock* for the current thread, first waiting for any current owner to release it if necessary.

bool lock\_try\_acquire (*struct lock \*lock* )

[Function]

Tries to acquire *lock* for use by the current thread, without waiting. Returns true if successful, false if the lock is already owned. Calling this function in a tight loop is a bad idea because it wastes CPU time, so use lock\_acquire() instead.

void lock\_release (*struct lock \*lock* )

Releases *lock*, which the current thread must own.

bool lock\_held\_by\_current\_thread (*const struct lock \*lock* )

[Function]

[Function]

Returns true if the running thread owns *lock*, false otherwise. There is no function to test whether an arbitrary thread owns a lock, because the answer could change before the caller could act on it.

#### Monitors

A *monitor* is a higher-level form of synchronization than a semaphore or a lock. A monitor consists of data being synchronized, plus a lock, called the *monitor lock*, and one or more *condition variables*. Before it accesses the protected data, a thread first acquires the monitor lock. It is then said to be “in the monitor”. While in the monitor, the thread has control over all the protected data, which it may freely examine or modify. When access to the protected data is complete, it releases the monitor lock.

Condition variables allow code in the monitor to wait for a condition to become true. Each condition variable is associated with an abstract condition, e.g. “some data has arrived for processing” or “over 10 seconds has passed since the user’s last keystroke”. When code in the monitor needs to wait for a condition to become true, it “waits” on the associated condition variable, which releases the lock and waits for the condition to be signaled. If, on the other hand, it has caused one of these conditions to become true, it “signals” the condition to wake up one waiter, or “broadcasts” the condition to wake all of them.

The theoretical framework for monitors was laid out by C. A. R. Hoare ([Hoare]). Their practical usage was later elaborated in a paper on the Mesa operating system ([Lampson]).

Condition variable types and functions are declared in ‘threads/synch.h’.

##### struct condition

Represents a condition variable.

void cond\_init (*struct condition \*cond* )

Initializes *cond* as a new condition variable.

void cond\_wait (*struct condition \*cond, struct lock \*lock* )

[Type] [Function]

[Function]

Atomically releases *lock* (the monitor lock) and waits for *cond* to be signaled by some other piece of code. After *cond* is signaled, reacquires *lock* before returning. *lock* must be held before calling this function.

Sending a signal and waking up from a wait are not an atomic operation. Thus, typically cond\_wait()’s caller must recheck the condition after the wait completes and, if necessary, wait again. See the next section for an example.

void cond\_signal (*struct condition \*cond, struct lock \*lock* )

[Function]

If any threads are waiting on *cond* (protected by monitor lock *lock*), then this function wakes up one of them. If no threads are waiting, returns without performing any action. *lock* must be held before calling this function.

void cond\_broadcast (*struct condition \*cond, struct lock \*lock* )

[Function]

Wakes up all threads, if any, waiting on *cond* (protected by monitor lock *lock*). *lock*

must be held before calling this function.

#### A.3.4.1 Monitor Example

The classical example of a monitor is handling a buffer into which one or more “producer” threads write characters and out of which one or more “consumer” threads read characters. To implement this we need, besides the monitor lock, two condition variables which we will call *not full* and *not empty* :

char buf[BUF\_SIZE]; /\* Buffer. \*/

size\_t n = 0; /\* 0 <= n <= *BUF SIZE*: # of characters in buffer. \*/ size\_t head = 0; /\* *buf* index of next char to write (mod *BUF SIZE*). \*/ size\_t tail = 0; /\* *buf* index of next char to read (mod *BUF SIZE*). \*/ struct lock lock; /\* Monitor lock. \*/

struct condition not\_empty; /\* Signaled when the buffer is not empty. \*/ struct condition not\_full; /\* Signaled when the buffer is not full. \*/

...initialize the locks and condition variables... void put (char ch) {

lock\_acquire (&lock);

while (n == BUF\_SIZE) /\* Can’t add to *buf* as long as it’s full. \*/ cond\_wait (&not\_full, &lock);

buf[head++  BUF\_SIZE] = ch; /\* Add *ch* to *buf*. \*/ n++;

cond\_signal (&not\_empty, &lock); /\* *buf* can’t be empty anymore. \*/ lock\_release (&lock);

}

char get (void) { char ch;

lock\_acquire (&lock);

while (n == 0) /\* Can’t read *buf* as long as it’s empty. \*/ cond\_wait (&not\_empty, &lock);

ch = buf[tail++  BUF\_SIZE]; /\* Get *ch* from *buf*. \*/ n--;

cond\_signal (&not\_full, &lock); /\* *buf* can’t be full anymore. \*/ lock\_release (&lock);

}

Note that BUF\_SIZE must divide evenly into SIZE\_MAX + 1 for the above code to be completely correct. Otherwise, it will fail the first time head wraps around to 0. In practice, BUF\_SIZE would ordinarily be a power of 2.

#### A.3.5 Optimization Barriers

An *optimization barrier* is a special statement that prevents the compiler from making assumptions about the state of memory across the barrier. The compiler will not reorder

reads or writes of variables across the barrier or assume that a variable’s value is unmod- ified across the barrier, except for local variables whose address is never taken. In Pintos, ‘threads/synch.h’ defines the barrier() macro as an optimization barrier.

One reason to use an optimization barrier is when data can change asynchronously, without the compiler’s knowledge, e.g. by another thread or an interrupt handler. The too\_many\_loops() function in ‘devices/timer.c’ is an example. This function starts out by busy-waiting in a loop until a timer tick occurs:

/\* Wait for a timer tick. \*/ int64\_t start = ticks;

while (ticks == start) barrier ();

Without an optimization barrier in the loop, the compiler could conclude that the loop would never terminate, because start and ticks start out equal and the loop itself never changes them. It could then “optimize” the function into an infinite loop, which would definitely be undesirable.

Optimization barriers can be used to avoid other compiler optimizations. The busy\_ wait() function, also in ‘devices/timer.c’, is an example. It contains this loop:

while (loops-- > 0) barrier ();

The goal of this loop is to busy-wait by counting loops down from its original value to

0. Without the barrier, the compiler could delete the loop entirely, because it produces no useful output and has no side effects. The barrier forces the compiler to pretend that the loop body has an important effect.

Finally, optimization barriers can be used to force the ordering of memory reads or writes. For example, suppose we add a “feature” that, whenever a timer interrupt occurs, the character in global variable timer\_put\_char is printed on the console, but only if global Boolean variable timer\_do\_put is true. The best way to set up ‘x’ to be printed is then to use an optimization barrier, like this:

timer\_put\_char = ’x’; barrier (); timer\_do\_put = true;

Without the barrier, the code is buggy because the compiler is free to reorder operations when it doesn’t see a reason to keep them in the same order. In this case, the compiler doesn’t know that the order of assignments is important, so its optimizer is permitted to exchange their order. There’s no telling whether it will actually do this, and it is possible that passing the compiler different optimization flags or using a different version of the compiler will produce different behavior.

Another solution is to disable interrupts around the assignments. This does not prevent reordering, but it prevents the interrupt handler from intervening between the assignments. It also has the extra runtime cost of disabling and re-enabling interrupts:

enum intr\_level old\_level = intr\_disable (); timer\_put\_char = ’x’;

timer\_do\_put = true; intr\_set\_level (old\_level);

A second solution is to mark the declarations of timer\_put\_char and timer\_do\_put as ‘volatile’. This keyword tells the compiler that the variables are externally observable and restricts its latitude for optimization. However, the semantics of ‘volatile’ are not well- defined, so it is not a good general solution. The base Pintos code does not use ‘volatile’ at all.

The following is *not* a solution, because locks neither prevent interrupts nor prevent the compiler from reordering the code within the region where the lock is held:

lock\_acquire (&timer\_lock); /\* INCORRECT CODE \*/ timer\_put\_char = ’x’;

timer\_do\_put = true; lock\_release (&timer\_lock);

The compiler treats invocation of any function defined externally, that is, in another source file, as a limited form of optimization barrier. Specifically, the compiler assumes that any externally defined function may access any statically or dynamically allocated data and any local variable whose address is taken. This often means that explicit barriers can be omitted. It is one reason that Pintos contains few explicit barriers.

A function defined in the same source file, or in a header included by the source file, cannot be relied upon as a optimization barrier. This applies even to invocation of a function before its definition, because the compiler may read and parse the entire source file before performing optimization.

## Interrupt Handling

An *interrupt* notifies the CPU of some event. Much of the work of an operating system relates to interrupts in one way or another. For our purposes, we classify interrupts into two broad categories:

* + - *Internal interrupts*, that is, interrupts caused directly by CPU instructions. System calls, attempts at invalid memory access (*page faults*), and attempts to divide by zero are some activities that cause internal interrupts. Because they are caused by CPU in- structions, internal interrupts are *synchronous* or synchronized with CPU instructions. intr\_disable() does not disable internal interrupts.
    - *External interrupts*, that is, interrupts originating outside the CPU. These interrupts come from hardware devices such as the system timer, keyboard, serial ports, and disks. External interrupts are *asynchronous*, meaning that their delivery is not syn- chronized with instruction execution. Handling of external interrupts can be postponed with intr\_disable() and related functions (see [Section A.3.1 [Disabling Interrupts],](#_bookmark82) [page 63](#_bookmark82)).

The CPU treats both classes of interrupts largely the same way, so Pintos has com- mon infrastructure to handle both classes. The following section describes this common infrastructure. The sections after that give the specifics of external and internal interrupts.

If you haven’t already read chapter 3, “Basic Execution Environment,” in [IA32-v1], it is recommended that you do so now. You might also want to skim chapter 5, “Interrupt and Exception Handling,” in [IA32-v3a].

#### Interrupt Infrastructure

When an interrupt occurs, the CPU saves its most essential state on a stack and jumps to an interrupt handler routine. The 80*x*86 architecture supports 256 interrupts, numbered 0 through 255, each with an independent handler defined in an array called the *interrupt descriptor table* or IDT.

In Pintos, intr\_init() in ‘threads/interrupt.c’ sets up the IDT so that each entry points to a unique entry point in ‘threads/intr-stubs.S’ named intr*NN* \_stub(), where *NN* is the interrupt number in hexadecimal. Because the CPU doesn’t give us any other way to find out the interrupt number, this entry point pushes the interrupt number on the stack. Then it jumps to intr\_entry(), which pushes all the registers that the processor didn’t already push for us, and then calls intr\_handler(), which brings us back into C in ‘threads/interrupt.c’.

The main job of intr\_handler() is to call the function registered for handling the particular interrupt. (If no function is registered, it dumps some information to the console and panics.) It also does some extra processing for external interrupts (see [Section A.4.3](#_bookmark94) [[External Interrupt Handling], page 72](#_bookmark94)).

When intr\_handler() returns, the assembly code in ‘threads/intr-stubs.S’ restores all the CPU registers saved earlier and directs the CPU to return from the interrupt.

The following types and functions are common to all interrupts.

##### void intr\_handler\_func (struct intr\_frame \**frame* )

[Type]

This is how an interrupt handler function must be declared. Its *frame* argument (see below) allows it to determine the cause of the interrupt and the state of the thread that was interrupted.

struct intr\_frame [Type]

The stack frame of an interrupt handler, as saved by the CPU, the interrupt stubs, and intr\_entry(). Its most interesting members are described below.

##### uint32\_t edi uint32\_t esi uint32\_t ebp uint32\_t esp\_dummy uint32\_t ebx uint32\_t edx uint32\_t ecx uint32\_t eax uint16\_t es uint16\_t ds

[Member of struct intr\_frame] [Member of struct intr\_frame] [Member of struct intr\_frame] [Member of struct intr\_frame] [Member of struct intr\_frame] [Member of struct intr\_frame] [Member of struct intr\_frame] [Member of struct intr\_frame] [Member of struct intr\_frame] [Member of struct intr\_frame]

Register values in the interrupted thread, pushed by intr\_entry(). The esp\_dummy

value isn’t actually used (refer to the description of PUSHA in [IA32-v2b] for details).

uint32\_t vec\_no [Member of struct intr\_frame]

The interrupt vector number, ranging from 0 to 255.

uint32\_t error\_code [Member of struct intr\_frame]

The “error code” pushed on the stack by the CPU for some internal interrupts.

void (\*eip) (*void*) [Member of struct intr\_frame]

The address of the next instruction to be executed by the interrupted thread.

##### void \* esp

The interrupted thread’s stack pointer.

[Member of struct intr\_frame]

const char \* intr\_name (*uint8 t vec* )

[Function]

Returns the name of the interrupt numbered *vec*, or "unknown" if the interrupt has no registered name.

#### Internal Interrupt Handling

Internal interrupts are caused directly by CPU instructions executed by the running kernel thread or user process (from project 2 onward). An internal interrupt is therefore said to arise in a “process context.”

In an internal interrupt’s handler, it can make sense to examine the struct intr\_frame passed to the interrupt handler, or even to modify it. When the interrupt returns, modifi- cations in struct intr\_frame become changes to the calling thread or process’s state. For example, the Pintos system call handler returns a value to the user program by modifying the saved EAX register (see [Section 3.5.2 [System Call Details], page 36](#_bookmark43)).

There are no special restrictions on what an internal interrupt handler can or can’t do. Generally they should run with interrupts enabled, just like other code, and so they can be preempted by other kernel threads. Thus, they do need to synchronize with other threads on shared data and other resources (see [Section A.3 [Synchronization], page 63](#_bookmark81)).

Internal interrupt handlers can be invoked recursively. For example, the system call handler might cause a page fault while attempting to read user memory. Deep recursion would risk overflowing the limited kernel stack (see [Section A.2.1 [struct thread], page 58](#_bookmark75)), but should be unnecessary.

void intr\_register\_int (*uint8 t vec, int dpl, enum intr level level, intr handler func \*handler, const char \*name* )

[Function]

Registers *handler* to be called when internal interrupt numbered *vec* is triggered. Names the interrupt *name* for debugging purposes.

If *level* is INTR\_ON, external interrupts will be processed normally during the interrupt handler’s execution, which is normally desirable. Specifying INTR\_OFF will cause the CPU to disable external interrupts when it invokes the interrupt handler. The effect is slightly different from calling intr\_disable() inside the handler, because that leaves a window of one or more CPU instructions in which external interrupts are still enabled. This is important for the page fault handler; refer to the comments in ‘userprog/exception.c’ for details.

*dpl* determines how the interrupt can be invoked. If *dpl* is 0, then the interrupt can be invoked only by kernel threads. Otherwise *dpl* should be 3, which allows user processes to invoke the interrupt with an explicit INT instruction. The value of *dpl* doesn’t affect user processes’ ability to invoke the interrupt indirectly, e.g. an invalid memory reference will cause a page fault regardless of *dpl*.

#### External Interrupt Handling

External interrupts are caused by events outside the CPU. They are asynchronous, so they can be invoked at any time that interrupts have not been disabled. We say that an external interrupt runs in an “interrupt context.”

In an external interrupt, the struct intr\_frame passed to the handler is not very meaningful. It describes the state of the thread or process that was interrupted, but there is no way to predict which one that is. It is possible, although rarely useful, to examine it, but modifying it is a recipe for disaster.

Only one external interrupt may be processed at a time. Neither internal nor external interrupt may nest within an external interrupt handler. Thus, an external interrupt’s handler must run with interrupts disabled (see [Section A.3.1 [Disabling Interrupts], page 63](#_bookmark82)).

An external interrupt handler must not sleep or yield, which rules out calling lock\_ acquire(), thread\_yield(), and many other functions. Sleeping in interrupt context would effectively put the interrupted thread to sleep, too, until the interrupt handler was again scheduled and returned. This would be unfair to the unlucky thread, and it would deadlock if the handler were waiting for the sleeping thread to, e.g., release a lock.

An external interrupt handler effectively monopolizes the machine and delays all other activities. Therefore, external interrupt handlers should complete as quickly as they can. Anything that require much CPU time should instead run in a kernel thread, possibly one that the interrupt triggers using a synchronization primitive.

External interrupts are controlled by a pair of devices outside the CPU called *pro- grammable interrupt controllers*, *PICs* for short. When intr\_init() sets up the CPU’s IDT, it also initializes the PICs for interrupt handling. The PICs also must be “acknowl- edged” at the end of processing for each external interrupt. intr\_handler() takes care of that by calling pic\_end\_of\_interrupt(), which properly signals the PICs.

The following functions relate to external interrupts.

void intr\_register\_ext (*uint8 t vec, intr handler func \*handler, const char \*name* )

[Function]

Registers *handler* to be called when external interrupt numbered *vec* is triggered. Names the interrupt *name* for debugging purposes. The handler will run with inter- rupts disabled.

bool intr\_context (*void*) [Function]

Returns true if we are running in an interrupt context, otherwise false. Mainly used in functions that might sleep or that otherwise should not be called from interrupt context, in this form:

ASSERT (!intr\_context ());

void intr\_yield\_on\_return (*void*) [Function]

When called in an interrupt context, causes thread\_yield() to be called just before the interrupt returns. Used in the timer interrupt handler when a thread’s time slice expires, to cause a new thread to be scheduled.

## Memory Allocation

Pintos contains two memory allocators, one that allocates memory in units of a page, and one that can allocate blocks of any size.

#### Page Allocator

The page allocator declared in ‘threads/palloc.h’ allocates memory in units of a page. It is most often used to allocate memory one page at a time, but it can also allocate multiple contiguous pages at once.

The page allocator divides the memory it allocates into two pools, called the kernel and user pools. By default, each pool gets half of system memory, but this can be changed with the ‘-ul’ kernel command line option (see [[Why PAL USER?], page 46](#_bookmark59)). An allocation request draws from one pool or the other. If one pool becomes empty, the other may still have free pages. The user pool should be used for allocating memory for user processes and the kernel pool for all other allocations. This will only become important starting with project 3. Until then, all allocations should be made from the kernel pool.

Each pool’s usage is tracked with a bitmap, one bit per page in the pool. A request to allocate *n* pages scans the bitmap for *n* consecutive bits set to false, indicating that those pages are free, and then sets those bits to true to mark them as used. This is a “first fit” allocation strategy (see [[Wilson], page 118](#_bookmark150)).

The page allocator is subject to fragmentation. That is, it may not be possible to allocate *n* contiguous pages even though *n* or more pages are free, because the free pages are separated by used pages. In fact, in pathological cases it may be impossible to allocate 2 contiguous pages even though half of the pool’s pages are free. Single-page requests can’t fail due to fragmentation, so requests for multiple contiguous pages should be limited as much as possible.

Pages may not be allocated from interrupt context, but they may be freed.

When a page is freed, all of its bytes are cleared to 0xcc, as a debugging aid (see [Section E.8 [Debugging Tips], page 109](#_bookmark138)).

Page allocator types and functions are described below.

void \* palloc\_get\_page (*enum palloc flags flags* )

void \* palloc\_get\_multiple (*enum palloc flags flags, size t*

###### *page\_cnt* )

[Function] [Function]

Obtains and returns one page, or *page cnt* contiguous pages, respectively. Returns a null pointer if the pages cannot be allocated.

The *flags* argument may be any combination of the following flags:

PAL\_ASSERT [Page Allocator Flag]

If the pages cannot be allocated, panic the kernel. This is only appropriate during kernel initialization. User processes should never be permitted to panic the kernel.

PAL\_ZERO [Page Allocator Flag]

Zero all the bytes in the allocated pages before returning them. If not set, the contents of newly allocated pages are unpredictable.

PAL\_USER [Page Allocator Flag]

Obtain the pages from the user pool. If not set, pages are allocated from the kernel pool.

void palloc\_free\_page (*void \*page* )

void palloc\_free\_multiple (*void \*pages, size t page\_cnt* )

[Function] [Function]

Frees one page, or *page cnt* contiguous pages, respectively, starting at *pages*. All of the pages must have been obtained using palloc\_get\_page() or palloc\_get\_ multiple().

#### Block Allocator

The block allocator, declared in ‘threads/malloc.h’, can allocate blocks of any size. It is layered on top of the page allocator described in the previous section. Blocks returned by the block allocator are obtained from the kernel pool.

The block allocator uses two different strategies for allocating memory. The first strategy applies to blocks that are 1 kB or smaller (one-fourth of the page size). These allocations are rounded up to the nearest power of 2, or 16 bytes, whichever is larger. Then they are grouped into a page used only for allocations of that size.

The second strategy applies to blocks larger than 1 kB. These allocations (plus a small amount of overhead) are rounded up to the nearest page in size, and then the block allocator requests that number of contiguous pages from the page allocator.

In either case, the difference between the allocation requested size and the actual block size is wasted. A real operating system would carefully tune its allocator to minimize this waste, but this is unimportant in an instructional system like Pintos.

As long as a page can be obtained from the page allocator, small allocations always succeed. Most small allocations do not require a new page from the page allocator at all, because they are satisfied using part of a page already allocated. However, large allocations always require calling into the page allocator, and any allocation that needs more than one contiguous page can fail due to fragmentation, as already discussed in the previous section. Thus, you should minimize the number of large allocations in your code, especially those over approximately 4 kB each.

When a block is freed, all of its bytes are cleared to 0xcc, as a debugging aid (see [Section E.8 [Debugging Tips], page 109](#_bookmark138)).

The block allocator may not be called from interrupt context.

The block allocator functions are described below. Their interfaces are the same as the standard C library functions of the same names.

void \* malloc (*size t size* )

[Function]

Obtains and returns a new block, from the kernel pool, at least *size* bytes long. Returns a null pointer if *size* is zero or if memory is not available.

void \* calloc (*size t a, size t b* )

[Function]

Obtains a returns a new block, from the kernel pool, at least *a* \* *b* bytes long. The block’s contents will be cleared to zeros. Returns a null pointer if *a* or *b* is zero or if insufficient memory is available.

void \* realloc (*void \*block, size t new\_size* )

[Function]

Attempts to resize *block* to *new size* bytes, possibly moving it in the process. If successful, returns the new block, in which case the old block must no longer be accessed. On failure, returns a null pointer, and the old block remains valid.

A call with *block* null is equivalent to malloc(). A call with *new size* zero is equiv- alent to free().

void free (*void \*block* )

[Function]

Frees *block*, which must have been previously returned by malloc(), calloc(), or

realloc() (and not yet freed).

## Virtual Addresses

A 32-bit virtual address can be divided into a 20-bit *page number* and a 12-bit *page offset*

(or just *offset*), like this:

31 12 11 0

+-------------------+-----------+

| Page Number | Offset |

+-------------------+-----------+

Virtual Address

Header ‘threads/vaddr.h’ defines these functions and macros for working with virtual addresses:

##### PGSHIFT PGBITS

[Macro] [Macro]

The bit index (0) and number of bits (12) of the offset part of a virtual address, respectively.

PGMASK [Macro]

A bit mask with the bits in the page offset set to 1, the rest set to 0 (0xfff).

##### PGSIZE

The page size in bytes (4,096).

unsigned pg\_ofs (*const void \*va* )

Extracts and returns the page offset in virtual address *va*.

uintptr\_t pg\_no (*const void \*va* )

Extracts and returns the page number in virtual address *va*.

void \* pg\_round\_down (*const void \*va* )

[Macro] [Function] [Function]

[Function]

Returns the start of the virtual page that *va* points within, that is, *va* with the page offset set to 0.

void \* pg\_round\_up (*const void \*va* )

Returns *va* rounded up to the nearest page boundary.

[Function]

Virtual memory in Pintos is divided into two regions: user virtual memory and ker- nel virtual memory (see [Section 3.1.4 [Virtual Memory Layout], page 24](#_bookmark28)). The boundary between them is PHYS\_BASE:

PHYS\_BASE [Macro]

Base address of kernel virtual memory. It defaults to 0xc0000000 (3 GB), but it may be changed to any multiple of 0x10000000 from 0x80000000 to 0xf0000000.

User virtual memory ranges from virtual address 0 up to PHYS\_BASE. Kernel virtual memory occupies the rest of the virtual address space, from PHYS\_BASE up to 4 GB.

bool is\_user\_vaddr (*const void \*va* )

bool is\_kernel\_vaddr (*const void \*va* )

[Function] [Function]

Returns true if *va* is a user or kernel virtual address, respectively, false otherwise.

The 80*x*86 doesn’t provide any way to directly access memory given a physical address. This ability is often necessary in an operating system kernel, so Pintos works around it by mapping kernel virtual memory one-to-one to physical memory. That is, virtual address PHYS\_BASE accesses physical address 0, virtual address PHYS\_BASE + 0x1234 accesses phys- ical address 0x1234, and so on up to the size of the machine’s physical memory. Thus, adding PHYS\_BASE to a physical address obtains a kernel virtual address that accesses that address; conversely, subtracting PHYS\_BASE from a kernel virtual address obtains the cor- responding physical address. Header ‘threads/vaddr.h’ provides a pair of functions to do these translations:

void \* ptov (*uintptr t pa* )

[Function]

Returns the kernel virtual address corresponding to physical address *pa*, which should be between 0 and the number of bytes of physical memory.

uintptr\_t vtop (*void \*va* )

[Function]

Returns the physical address corresponding to *va*, which must be a kernel virtual address.

## Page Table

The code in ‘pagedir.c’ is an abstract interface to the 80*x*86 hardware page table, also called a “page directory” by Intel processor documentation. The page table interface uses a uint32\_t \* to represent a page table because this is convenient for accessing their internal structure.

The sections below describe the page table interface and internals.

#### Creation, Destruction, and Activation

These functions create, destroy, and activate page tables. The base Pintos code already calls these functions where necessary, so it should not be necessary to call them yourself.

uint32\_t \* pagedir\_create (*void*) [Function]

Creates and returns a new page table. The new page table contains Pintos’s normal kernel virtual page mappings, but no user virtual mappings.

Returns a null pointer if memory cannot be obtained.

void pagedir\_destroy (*uint32 t \*pd* )

[Function]

Frees all of the resources held by *pd*, including the page table itself and the frames that it maps.

void pagedir\_activate (*uint32 t \*pd* )

[Function]

Activates *pd*. The active page table is the one used by the CPU to translate memory references.

#### Inspection and Updates

These functions examine or update the mappings from pages to frames encapsulated by a page table. They work on both active and inactive page tables (that is, those for running and suspended processes), flushing the TLB as necessary.

bool pagedir\_set\_page (*uint32 t \*pd, void \*upage, void \*kpage, bool*

###### *writable* )

[Function]

Adds to *pd* a mapping from user page *upage* to the frame identified by kernel virtual address *kpage*. If *writable* is true, the page is mapped read/write; otherwise, it is mapped read-only.

User page *upage* must not already be mapped in *pd*.

Kernel page *kpage* should be a kernel virtual address obtained from the user pool with palloc\_get\_page(PAL\_USER) (see [[Why PAL USER?], page 46](#_bookmark59)).

Returns true if successful, false on failure. Failure will occur if additional memory required for the page table cannot be obtained.

void \* pagedir\_get\_page (*uint32 t \*pd, const void \*uaddr* )

[Function]

Looks up the frame mapped to *uaddr* in *pd*. Returns the kernel virtual address for that frame, if *uaddr* is mapped, or a null pointer if it is not.

void pagedir\_clear\_page (*uint32 t \*pd, void \*page* )

Marks *page* “not present” in *pd*. Later accesses to the page will fault.

[Function]

Other bits in the page table for *page* are preserved, permitting the accessed and dirty bits (see the next section) to be checked.

This function has no effect if *page* is not mapped.

#### Accessed and Dirty Bits

80*x*86 hardware provides some assistance for implementing page replacement algorithms, through a pair of bits in the page table entry (PTE) for each page. On any read or write to a page, the CPU sets the *accessed bit* to 1 in the page’s PTE, and on any write, the CPU sets the *dirty bit* to 1. The CPU never resets these bits to 0, but the OS may do so.

Proper interpretation of these bits requires understanding of *aliases*, that is, two (or more) pages that refer to the same frame. When an aliased frame is accessed, the accessed and dirty bits are updated in only one page table entry (the one for the page used for access). The accessed and dirty bits for the other aliases are not updated.

See [Section 4.1.5.1 [Accessed and Dirty Bits], page 41](#_bookmark51), on applying these bits in imple- menting page replacement algorithms.

bool pagedir\_is\_dirty (*uint32 t \*pd, const void \*page* )

bool pagedir\_is\_accessed (*uint32 t \*pd, const void \*page* )

[Function] [Function]

Returns true if page directory *pd* contains a page table entry for *page* that is marked dirty (or accessed). Otherwise, returns false.

void pagedir\_set\_dirty (*uint32 t \*pd, const void \*page, bool value* )

void pagedir\_set\_accessed (*uint32 t \*pd, const void \*page, bool*

###### *value* )

[Function] [Function]

If page directory *pd* has a page table entry for *page*, then its dirty (or accessed) bit is set to *value*.

#### Page Table Details

The functions provided with Pintos are sufficient to implement the projects. However, you may still find it worthwhile to understand the hardware page table format, so we’ll go into a little detail in this section.

#### Structure

The top-level paging data structure is a page called the “page directory” (PD) arranged as an array of 1,024 32-bit page directory entries (PDEs), each of which represents 4 MB of virtual memory. Each PDE may point to the physical address of another page called a “page table” (PT) arranged, similarly, as an array of 1,024 32-bit page table entries (PTEs), each of which translates a single 4 kB virtual page to a physical page.

Translation of a virtual address into a physical address follows the three-step process illustrated in the diagram below:2

* + - * 1. The most-significant 10 bits of the virtual address (bits 22. . . 31) index the page direc- tory. If the PDE is marked “present,” the physical address of a page table is read from the PDE thus obtained. If the PDE is marked “not present” then a page fault occurs.
        2. The next 10 bits of the virtual address (bits 12. . . 21) index the page table. If the PTE is marked “present,” the physical address of a data page is read from the PTE thus obtained. If the PTE is marked “not present” then a page fault occurs.
        3. The least-significant 12 bits of the virtual address (bits 0. . . 11) are added to the data page’s physical base address, yielding the final physical address.

2 Actually, virtual to physical translation on the 80*x*86 architecture occurs via an intermediate “linear

address,” but Pintos (and most modern 80*x*86 OSes) set up the CPU so that linear and virtual addresses are one and the same. Thus, you can effectively ignore this CPU feature.

31 22 21 12 11 0

+----------------------+----------------------+----------------------+

| Page Directory Index | Page Table Index | Page Offset |

+----------------------+----------------------+----------------------+

| | |

/ / /

/ / /

/ Page Directory / Page Table / Data Page

/ . . / . . / . .

|1,023| | |1,023| | | | |

|1,022| | |1,022| | | | |

|1,021| | |1,021| | \ \| |

|1,020| | |1,020| | /| |

| | | | | | | |

| | | \ \| |\_ | |

. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| | | . | | | /| . | | \ | | |
| \ \| . | |\_ | | . | | | | | |
| /| . | | \ | | . | | | | | |
| | . | | | | | . | | | | | |
| | | | | | | | | | | | |

. |

. |

. |

|

| | | | | | | | 4| | | 4| | | | |

3| | | 3| | | | |

2| | | 2| | | | |

1| | | 1| | | | |

0| | \ \0| | \ \| |

/ /

Pintos provides some macros and functions that are useful for working with raw page tables:

##### PTSHIFT PTBITS

[Macro] [Macro]

The starting bit index (12) and number of bits (10), respectively, in a page table index.

PTMASK [Macro]

A bit mask with the bits in the page table index set to 1 and the rest set to 0 (0x3ff000).

PTSPAN [Macro]

The number of bytes of virtual address space that a single page table page covers (4,194,304 bytes, or 4 MB).

##### PDSHIFT PDBITS

[Macro] [Macro]

The starting bit index (22) and number of bits (10), respectively, in a page directory index.

PDMASK [Macro]

A bit mask with the bits in the page directory index set to 1 and other bits set to 0 (0xffc00000).

uintptr\_t pd\_no (*const void \*va* )

uintptr\_t pt\_no (*const void \*va* )

[Function] [Function]

Returns the page directory index or page table index, respectively, for virtual address

*va*. These functions are defined in ‘threads/pte.h’.

unsigned pg\_ofs (*const void \*va* )

[Function]

Returns the page offset for virtual address *va*. This function is defined in ‘threads/vaddr.h’.

#### Page Table Entry Format

You do not need to understand the PTE format to do the Pintos projects, unless you wish to incorporate the page table into your supplemental page table (see [Section 4.1.4 [Managing](#_bookmark48) [the Supplemental Page Table], page 40](#_bookmark48)).

The actual format of a page table entry is summarized below. For complete information, refer to section 3.7, “Page Translation Using 32-Bit Physical Addressing,” in [IA32-v3a].

31 12 11 9 6 5 2 1 0

+---------------------------------------+----+----+-+-+---+-+-+-+

| Physical Address | AVL| |D|A| |U|W|P|

+---------------------------------------+----+----+-+-+---+-+-+-+

Some more information on each bit is given below. The names are ‘threads/pte.h’ macros that represent the bits’ values:

PTE\_P [Macro]

Bit 0, the “present” bit. When this bit is 1, the other bits are interpreted as described below. When this bit is 0, any attempt to access the page will page fault. The remaining bits are then not used by the CPU and may be used by the OS for any purpose.

PTE\_W [Macro]

Bit 1, the “read/write” bit. When it is 1, the page is writable. When it is 0, write attempts will page fault.

PTE\_U [Macro]

Bit 2, the “user/supervisor” bit. When it is 1, user processes may access the page. When it is 0, only the kernel may access the page (user accesses will page fault).

Pintos clears this bit in PTEs for kernel virtual memory, to prevent user processes from accessing them.

PTE\_A [Macro]

Bit 5, the “accessed” bit. See [Section A.7.3 [Page Table Accessed and Dirty Bits],](#_bookmark104) [page 77](#_bookmark104).

PTE\_D [Macro]

Bit 6, the “dirty” bit. See [Section A.7.3 [Page Table Accessed and Dirty Bits], page 77](#_bookmark104).

PTE\_AVL [Macro]

Bits 9. . . 11, available for operating system use. Pintos, as provided, does not use them and sets them to 0.

PTE\_ADDR [Macro]

Bits 12. . . 31, the top 20 bits of the physical address of a frame. The low 12 bits of the frame’s address are always 0.

Other bits are either reserved or uninteresting in a Pintos context and should be set to 0. Header ‘threads/pte.h’ defines three functions for working with page table entries:

uint32\_t pte\_create\_kernel (*uint32 t \*page, bool writable* )

[Function]

Returns a page table entry that points to *page*, which should be a kernel virtual address. The PTE’s present bit will be set. It will be marked for kernel-only access. If *writable* is true, the PTE will also be marked read/write; otherwise, it will be read-only.

uint32\_t pte\_create\_user (*uint32 t \*page, bool writable* )

[Function]

Returns a page table entry that points to *page*, which should be the kernel virtual address of a frame in the user pool (see [[Why PAL USER?], page 46](#_bookmark59)). The PTE’s present bit will be set and it will be marked to allow user-mode access. If *writable* is true, the PTE will also be marked read/write; otherwise, it will be read-only.

void \* pte\_get\_page (*uint32 t pte* )

[Function]

Returns the kernel virtual address for the frame that *pte* points to. The *pte* may be present or not-present; if it is not-present then the pointer returned is only meaningful if the address bits in the PTE actually represent a physical address.

#### Page Directory Entry Format

Page directory entries have the same format as PTEs, except that the physical address points to a page table page instead of a frame. Header ‘threads/pte.h’ defines two functions for working with page directory entries:

uint32\_t pde\_create (*uint32 t \*pt* )

[Function]

Returns a page directory that points to *page*, which should be the kernel virtual address of a page table page. The PDE’s present bit will be set, it will be marked to allow user-mode access, and it will be marked read/write.

uint32\_t \* pde\_get\_pt (*uint32 t pde* )

[Function]

Returns the kernel virtual address for the page table page that *pde*, which must be marked present, points to.

## Hash Table

Pintos provides a hash table data structure in ‘lib/kernel/hash.c’. To use it you will need to include its header file, ‘lib/kernel/hash.h’, with #include <hash.h>. No code provided with Pintos uses the hash table, which means that you are free to use it as is, modify its implementation for your own purposes, or ignore it, as you wish.

Most implementations of the virtual memory project use a hash table to translate pages to frames. You may find other uses for hash tables as well.

#### Data Types

A hash table is represented by struct hash.

##### struct hash

[Type]

Represents an entire hash table. The actual members of struct hash are “opaque.” That is, code that uses a hash table should not access struct hash members directly, nor should it need to. Instead, use hash table functions and macros.

The hash table operates on elements of type struct hash\_elem.

struct hash\_elem [Type]

Embed a struct hash\_elem member in the structure you want to include in a hash table. Like struct hash, struct hash\_elem is opaque. All functions for operating on hash table elements actually take and return pointers to struct hash\_elem, not pointers to your hash table’s real element type.

You will often need to obtain a struct hash\_elem given a real element of the hash table, and vice versa. Given a real element of the hash table, you may use the ‘&’ operator to obtain a pointer to its struct hash\_elem. Use the hash\_entry() macro to go the other direction.

*type* \* hash\_entry (*struct hash elem \*elem, type, member* )

[Macro]

Returns a pointer to the structure that *elem*, a pointer to a struct hash\_elem, is embedded within. You must provide *type*, the name of the structure that *elem* is inside, and *member*, the name of the member in *type* that *elem* points to.

For example, suppose h is a struct hash\_elem \* variable that points to a struct thread member (of type struct hash\_elem) named h\_elem. Then, hash\_entry (h, struct thread, h\_elem) yields the address of the struct thread that h points within.

See [Section A.8.5 [Hash Table Example], page 86](#_bookmark114), for an example.

Each hash table element must contain a key, that is, data that identifies and distinguishes elements, which must be unique among elements in the hash table. (Elements may also contain non-key data that need not be unique.) While an element is in a hash table, its key data must not be changed. Instead, if need be, remove the element from the hash table, modify its key, then reinsert the element.

For each hash table, you must write two functions that act on keys: a hash function and a comparison function. These functions must match the following prototypes:

##### unsigned hash\_hash\_func (const struct hash\_elem \**element*, void \**aux* )

[Type]

Returns a hash of *element*’s data, as a value anywhere in the range of unsigned int. The hash of an element should be a pseudo-random function of the element’s key. It must not depend on non-key data in the element or on any non-constant data other than the key. Pintos provides the following functions as a suitable basis for hash functions.

unsigned hash\_bytes (*const void \*buf, size t \*size* )

[Function]

Returns a hash of the *size* bytes starting at *buf*. The implementation is the general-purpose [Fowler-Noll-Vo hash](http://en.wikipedia.org/wiki/Fowler_Noll_Vo_hash) for 32-bit words.

unsigned hash\_string (*const char \*s* )

Returns a hash of null-terminated string *s*.

unsigned hash\_int (*int i* )

Returns a hash of integer *i*.

[Function] [Function]

If your key is a single piece of data of an appropriate type, it is sensible for your hash function to directly return the output of one of these functions. For multiple pieces of data, you may wish to combine the output of more than one call to them using, e.g., the ‘^’ (exclusive or) operator. Finally, you may entirely ignore these functions and write your own hash function from scratch, but remember that your goal is to build an operating system kernel, not to design a hash function.

See [Section A.8.6 [Hash Auxiliary Data], page 87](#_bookmark116), for an explanation of *aux*.

##### bool hash\_less\_func (const struct hash\_elem \**a*, const struct hash\_elem \**b*, void \**aux* )

[Type]

Compares the keys stored in elements *a* and *b*. Returns true if *a* is less than *b*, false if *a* is greater than or equal to *b*.

If two elements compare equal, then they must hash to equal values.

See [Section A.8.6 [Hash Auxiliary Data], page 87](#_bookmark116), for an explanation of *aux*.

See [Section A.8.5 [Hash Table Example], page 86](#_bookmark114), for hash and comparison function examples.

A few functions accept a pointer to a third kind of function as an argument:

##### void hash\_action\_func (struct hash\_elem \**element*, void \**aux* )

Performs some kind of action, chosen by the caller, on *element*.

See [Section A.8.6 [Hash Auxiliary Data], page 87](#_bookmark116), for an explanation of *aux*.

#### Basic Functions

These functions create, destroy, and inspect hash tables.

[Type]

bool hash\_init (*struct hash \*hash, hash hash func \*hash\_func, hash less func \*less\_func, void \*aux* )

[Function]

Initializes *hash* as a hash table with *hash func* as hash function, *less func* as compar- ison function, and *aux* as auxiliary data. Returns true if successful, false on failure. hash\_init() calls malloc() and fails if memory cannot be allocated.

See [Section A.8.6 [Hash Auxiliary Data], page 87](#_bookmark116), for an explanation of *aux*, which is most often a null pointer.

void hash\_clear (*struct hash \*hash, hash action func \*action* )

[Function]

Removes all the elements from *hash*, which must have been previously initialized with

hash\_init().

If *action* is non-null, then it is called once for each element in the hash table, which gives the caller an opportunity to deallocate any memory or other resources used by the element. For example, if the hash table elements are dynamically allocated using malloc(), then *action* could free() the element. This is safe because hash\_ clear() will not access the memory in a given hash element after calling *action* on

it. However, *action* must not call any function that may modify the hash table, such as hash\_insert() or hash\_delete().

void hash\_destroy (*struct hash \*hash, hash action func \*action* )

[Function]

If *action* is non-null, calls it for each element in the hash, with the same semantics as a call to hash\_clear(). Then, frees the memory held by *hash*. Afterward, *hash* must not be passed to any hash table function, absent an intervening call to hash\_init().

size\_t hash\_size (*struct hash \*hash* )

Returns the number of elements currently stored in *hash*.

bool hash\_empty (*struct hash \*hash* )

[Function]

[Function]

Returns true if *hash* currently contains no elements, false if *hash* contains at least one element.

#### Search Functions

Each of these functions searches a hash table for an element that compares equal to one provided. Based on the success of the search, they perform some action, such as inserting a new element into the hash table, or simply return the result of the search.

struct hash\_elem \* hash\_insert (*struct hash \*hash, struct hash elem \*element* )

[Function]

Searches *hash* for an element equal to *element*. If none is found, inserts *element* into *hash* and returns a null pointer. If the table already contains an element equal to *element*, it is returned without modifying *hash*.

struct hash\_elem \* hash\_replace (*struct hash \*hash, struct hash elem \*element* )

[Function]

Inserts *element* into *hash*. Any element equal to *element* already in *hash* is removed. Returns the element removed, or a null pointer if *hash* did not contain an element equal to *element*.

The caller is responsible for deallocating any resources associated with the returned element, as appropriate. For example, if the hash table elements are dynamically allocated using malloc(), then the caller must free() the element after it is no longer needed.

The element passed to the following functions is only used for hashing and comparison purposes. It is never actually inserted into the hash table. Thus, only key data in the element needs to be initialized, and other data in the element will not be used. It often makes sense to declare an instance of the element type as a local variable, initialize the key data, and then pass the address of its struct hash\_elem to hash\_find() or hash\_ delete(). See [Section A.8.5 [Hash Table Example], page 86](#_bookmark114), for an example. (Large structures should not be allocated as local variables. See [Section A.2.1 [struct thread],](#_bookmark75) [page 58](#_bookmark75), for more information.)

struct hash\_elem \* hash\_find (*struct hash \*hash, struct hash elem*

###### *\*element* )

[Function]

Searches *hash* for an element equal to *element*. Returns the element found, if any, or a null pointer otherwise.

struct hash\_elem \* hash\_delete (*struct hash \*hash, struct hash elem \*element* )

[Function]

Searches *hash* for an element equal to *element*. If one is found, it is removed from

*hash* and returned. Otherwise, a null pointer is returned and *hash* is unchanged.

The caller is responsible for deallocating any resources associated with the returned element, as appropriate. For example, if the hash table elements are dynamically allocated using malloc(), then the caller must free() the element after it is no longer needed.

#### Iteration Functions

These functions allow iterating through the elements in a hash table. Two interfaces are supplied. The first requires writing and supplying a *hash action func* to act on each element (see [Section A.8.1 [Hash Data Types], page 82](#_bookmark110)).

void hash\_apply (*struct hash \*hash, hash action func \*action* )

[Function]

Calls *action* once for each element in *hash*, in arbitrary order. *action* must not call any function that may modify the hash table, such as hash\_insert() or hash\_delete().  *action* must not modify key data in elements, although it may modify any other data.

The second interface is based on an “iterator” data type. Idiomatically, iterators are used as follows:

struct hash\_iterator i;

hash\_first (&i, h); while (hash\_next (&i))

{

struct foo \*f = hash\_entry (hash\_cur (&i), struct foo, elem);

. . . do something with *f* . . .

}

struct hash\_iterator [Type]

Represents a position within a hash table. Calling any function that may modify a hash table, such as hash\_insert() or hash\_delete(), invalidates all iterators within that hash table.

Like struct hash and struct hash\_elem, struct hash\_elem is opaque.

void hash\_first (*struct hash iterator \*iterator, struct hash \*hash* )

Initializes *iterator* to just before the first element in *hash*.

struct hash\_elem \* hash\_next (*struct hash iterator \*iterator* )

[Function]

[Function]

Advances *iterator* to the next element in *hash*, and returns that element. Returns a null pointer if no elements remain. After hash\_next() returns null for *iterator*, calling it again yields undefined behavior.

struct hash\_elem \* hash\_cur (*struct hash iterator \*iterator* )

[Function]

Returns the value most recently returned by hash\_next() for *iterator*. Yields un- defined behavior after hash\_first() has been called on *iterator* but before hash\_ next() has been called for the first time.

#### Hash Table Example

Suppose you have a structure, called struct page, that you want to put into a hash table. First, define struct page to include a struct hash\_elem member:

struct page

{

struct hash\_elem hash\_elem; /\* Hash table element. \*/ void \*addr; /\* Virtual address. \*/

/\* . . . other members. . . \*/

};

We write a hash function and a comparison function using *addr* as the key. A pointer can be hashed based on its bytes, and the ‘<’ operator works fine for comparing pointers:

/\* Returns a hash value for page *p*. \*/ unsigned

page\_hash (const struct hash\_elem \*p\_, void \*aux UNUSED)

{

const struct page \*p = hash\_entry (p\_, struct page, hash\_elem); return hash\_bytes (&p->addr, sizeof p->addr);

}

/\* Returns true if page *a* precedes page *b*. \*/ bool

page\_less (const struct hash\_elem \*a\_, const struct hash\_elem \*b\_, void \*aux UNUSED)

{

const struct page \*a = hash\_entry (a\_, struct page, hash\_elem); const struct page \*b = hash\_entry (b\_, struct page, hash\_elem);

return a->addr < b->addr;

}

(The use of UNUSED in these functions’ prototypes suppresses a warning that *aux* is un- used. See [Section E.3 [Function and Parameter Attributes], page 99](#_bookmark128), for information about UNUSED. See [Section A.8.6 [Hash Auxiliary Data], page 87](#_bookmark116), for an explanation of *aux*.)

Then, we can create a hash table like this:

struct hash pages;

hash\_init (&pages, page\_hash, page\_less, NULL);

Now we can manipulate the hash table we’ve created. If *p* is a pointer to a struct page, we can insert it into the hash table with:

hash\_insert (&pages, &p->hash\_elem);

If there’s a chance that *pages* might already contain a page with the same *addr*, then we should check hash\_insert()’s return value.

To search for an element in the hash table, use hash\_find(). This takes a little setup, because hash\_find() takes an element to compare against. Here’s a function that will find and return a page based on a virtual address, assuming that *pages* is defined at file scope:

/\* Returns the page containing the given virtual *address*, or a null pointer if no such page exists. \*/

struct page \*

page\_lookup (const void \*address)

{

struct page p; struct hash\_elem \*e;

p.addr = address;

e = hash\_find (&pages, &p.hash\_elem);

return e != NULL ? hash\_entry (e, struct page, hash\_elem) : NULL;

}

struct page is allocated as a local variable here on the assumption that it is fairly small. Large structures should not be allocated as local variables. See [Section A.2.1 [struct thread],](#_bookmark75) [page 58](#_bookmark75), for more information.

A similar function could delete a page by address using hash\_delete().

#### Auxiliary Data

In simple cases like the example above, there’s no need for the *aux* parameters. In these cases, just pass a null pointer to hash\_init() for *aux* and ignore the values passed to the hash function and comparison functions. (You’ll get a compiler warning if you don’t use the *aux* parameter, but you can turn that off with the UNUSED macro, as shown in the example, or you can just ignore it.)

*aux* is useful when you have some property of the data in the hash table is both constant and needed for hashing or comparison, but not stored in the data items themselves. For example, if the items in a hash table are fixed-length strings, but the items themselves don’t indicate what that fixed length is, you could pass the length as an *aux* parameter.

#### Synchronization

The hash table does not do any internal synchronization. It is the caller’s responsibility to synchronize calls to hash table functions. In general, any number of functions that examine but do not modify the hash table, such as hash\_find() or hash\_next(), may execute simultaneously. However, these function cannot safely execute at the same time as any function that may modify a given hash table, such as hash\_insert() or hash\_delete(), nor may more than one function that can modify a given hash table execute safely at once.

It is also the caller’s responsibility to synchronize access to data in hash table elements. How to synchronize access to this data depends on how it is designed and organized, as with any other data structure.

# Appendix B 4.4BSD Scheduler

The goal of a general-purpose scheduler is to balance threads’ different scheduling needs. Threads that perform a lot of I/O require a fast response time to keep input and output devices busy, but need little CPU time. On the other hand, compute-bound threads need to receive a lot of CPU time to finish their work, but have no requirement for fast response time. Other threads lie somewhere in between, with periods of I/O punctuated by periods of computation, and thus have requirements that vary over time. A well-designed scheduler can often accommodate threads with all these requirements simultaneously.

For project 1, you must implement the scheduler described in this appendix. Our sched- uler resembles the one described in [McKusick], which is one example of a *multilevel feedback queue* scheduler. This type of scheduler maintains several queues of ready-to-run threads, where each queue holds threads with a different priority. At any given time, the scheduler chooses a thread from the highest-priority non-empty queue. If the highest-priority queue contains multiple threads, then they run in “round robin” order.

Multiple facets of the scheduler require data to be updated after a certain number of timer ticks. In every case, these updates should occur before any ordinary kernel thread has a chance to run, so that there is no chance that a kernel thread could see a newly increased timer\_ticks() value but old scheduler data values.

The 4.4BSD scheduler does not include priority donation.

## Niceness

Thread priority is dynamically determined by the scheduler using a formula given below. However, each thread also has an integer *nice* value that determines how “nice” the thread should be to other threads. A *nice* of zero does not affect thread priority. A positive *nice*, to the maximum of 20, decreases the priority of a thread and causes it to give up some CPU time it would otherwise receive. On the other hand, a negative *nice*, to the minimum of -20, tends to take away CPU time from other threads.

The initial thread starts with a *nice* value of zero. Other threads start with a *nice* value inherited from their parent thread. You must implement the functions described below, which are for use by test programs. We have provided skeleton definitions for them in ‘threads/thread.c’.

int thread\_get\_nice (*void*)

Returns the current thread’s *nice* value.

void thread\_set\_nice (*int new\_nice* )

[Function]

[Function]

Sets the current thread’s *nice* value to *new nice* and recalculates the thread’s priority based on the new value (see [Section B.2 [Calculating Priority], page 88](#_bookmark118)). If the running thread no longer has the highest priority, yields.

## Calculating Priority

Our scheduler has 64 priorities and thus 64 ready queues, numbered 0 (PRI\_MIN) through 63 (PRI\_MAX). Lower numbers correspond to lower priorities, so that priority 0 is the lowest priority and priority 63 is the highest. Thread priority is calculated initially at thread

initialization. It is also recalculated once every fourth clock tick, for every thread. In either case, it is determined by the formula

*priority* = PRI\_MAX - (*recent\_cpu* / 4) - (*nice* \* 2),

where *recent cpu* is an estimate of the CPU time the thread has used recently (see be- low) and *nice* is the thread’s *nice* value. The coefficients 1*/*4 and 2 on *recent cpu* and *nice*, respectively, have been found to work well in practice but lack deeper meaning. The calculated *priority* is always adjusted to lie in the valid range PRI\_MIN to PRI\_MAX.

This formula gives a thread that has received CPU time recently lower priority for being reassigned the CPU the next time the scheduler runs. This is key to preventing starvation: a thread that has not received any CPU time recently will have a *recent cpu* of 0, which barring a high *nice* value should ensure that it receives CPU time soon.

* 1. **Calculating *recent cpu***

We wish *recent cpu* to measure how much CPU time each process has received “recently.” Furthermore, as a refinement, more recent CPU time should be weighted more heavily than less recent CPU time. One approach would use an array of *n* elements to track the CPU time received in each of the last *n* seconds. However, this approach requires O(*n*) space per thread and O(*n*) time per calculation of a new weighted average.

Instead, we use a *exponentially weighted moving average*, which takes this general form:

*x*(0) = *f* (0)*,*

*x*(*t*) = *ax*(*t −* 1) + (1 *− a*)*f* (*t*)*, a* = *k/*(*k* + 1)*,*

where *x*(*t*) is the moving average at integer time *t ≥* 0, *f* (*t*) is the function being averaged, and *k >* 0 controls the rate of decay. We can iterate the formula over a few steps as follows: *x*(1) = *f* (1),

*x*(2) = *af* (1) + *f* (2),

.

.

.

*x*(5) = *a*4*f* (1) + *a*3*f* (2) + *a*2*f* (3) + *af* (4) + *f* (5).

The value of *f* (*t*) has a weight of 1 at time *t*, a weight of *a* at time *t* + 1, *a*2 at time *t* + 2, and so on. We can also relate *x*(*t*) to *k*: *f* (*t*) has a weight of approximately 1*/e* at time *t* + *k*, approximately 1*/e*2 at time *t* + 2*k*, and so on. From the opposite direction, *f* (*t*) decays to weight *w* at time *t* + log*a w*.

The initial value of *recent cpu* is 0 in the first thread created, or the parent’s value in other new threads. Each time a timer interrupt occurs, *recent cpu* is incremented by 1 for the running thread only, unless the idle thread is running. In addition, once per second the value of *recent cpu* is recalculated for every thread (whether running, ready, or blocked), using this formula:

*recent\_cpu* = (2\**load\_avg* )/(2\**load\_avg* + 1) \* *recent\_cpu* + *nice* ,

where *load avg* is a moving average of the number of threads ready to run (see below). If *load avg* is 1, indicating that a single thread, on average, is competing for the CPU, then the current value of *recent cpu* decays to a weight of .1 in log2*/*3 *.*1 *≈* 6 seconds; if *load avg* is 2, then decay to a weight of .1 takes log3*/*4 *.*1 *≈* 8 seconds. The effect is that *recent cpu* estimates the amount of CPU time the thread has received “recently,” with the rate of decay inversely proportional to the number of threads competing for the CPU.

Assumptions made by some of the tests require that updates to *recent cpu* be made exactly when the system tick counter reaches a multiple of a second, that is, when timer\_ ticks ()  TIMER\_FREQ == 0, and not at any other time.

The value of *recent cpu* can be negative for a thread with a negative *nice* value. Do not clamp negative *recent cpu* to 0.

You may need to think about the order of calculations in this formula. We recommend computing the coefficient of *recent cpu* first, then multiplying. Some students have reported that multiplying *load avg* by *recent cpu* directly can cause overflow.

You must implement thread\_get\_recent\_cpu(), for which there is a skeleton in ‘threads/thread.c’.

int thread\_get\_recent\_cpu (*void*) [Function]

Returns 100 times the current thread’s *recent cpu* value, rounded to the nearest integer.

* 1. **Calculating *load avg***

Finally, *load avg*, often known as the system load average, estimates the average number of threads ready to run over the past minute. Like *recent cpu*, it is an exponentially weighted moving average. Unlike *priority* and *recent cpu*, *load avg* is system-wide, not thread- specific. At system boot, it is initialized to 0. Once per second thereafter, it is updated according to the following formula:

*load\_avg* = (59/60)\**load\_avg* + (1/60)\**ready\_threads* ,

where *ready threads* is the number of threads that are either running or ready to run at time of update (not including the idle thread).

Because of assumptions made by some of the tests, *load avg* must be updated exactly when the system tick counter reaches a multiple of a second, that is, when timer\_ticks ()  TIMER\_FREQ == 0, and not at any other time.

You must implement thread\_get\_load\_avg(), for which there is a skeleton in ‘threads/thread.c’.

int thread\_get\_load\_avg (*void*) [Function]

Returns 100 times the current system load average, rounded to the nearest integer.

## Summary

The following formulas summarize the calculations required to implement the scheduler. They are not a complete description of scheduler requirements.

Every thread has a *nice* value between -20 and 20 directly under its control. Each thread also has a priority, between 0 (PRI\_MIN) through 63 (PRI\_MAX), which is recalculated using the following formula every fourth tick:

*priority* = PRI\_MAX - (*recent\_cpu* / 4) - (*nice* \* 2).

*recent cpu* measures the amount of CPU time a thread has received “recently.” On each timer tick, the running thread’s *recent cpu* is incremented by 1. Once per second, every thread’s *recent cpu* is updated this way:

*recent\_cpu* = (2\**load\_avg* )/(2\**load\_avg* + 1) \* *recent\_cpu* + *nice* .

*load avg* estimates the average number of threads ready to run over the past minute. It is initialized to 0 at boot and recalculated once per second as follows:

*load\_avg* = (59/60)\**load\_avg* + (1/60)\**ready\_threads* .

where *ready threads* is the number of threads that are either running or ready to run at time of update (not including the idle thread).

## Fixed-Point Real Arithmetic

In the formulas above, *priority*, *nice*, and *ready threads* are integers, but *recent cpu* and *load avg* are real numbers. Unfortunately, Pintos does not support floating-point arithmetic in the kernel, because it would complicate and slow the kernel. Real kernels often have the same limitation, for the same reason. This means that calculations on real quantities must be simulated using integers. This is not difficult, but many students do not know how to do it. This section explains the basics.

The fundamental idea is to treat the rightmost bits of an integer as representing a fraction. For example, we can designate the lowest 14 bits of a signed 32-bit integer as fractional bits, so that an integer *x* represents the real number *x/*214. This is called a 17.14 fixed-point number representation, because there are 17 bits before the decimal point, 14 bits after it, and one sign bit.1 A number in 17.14 format represents, at maximum, a value of (231 *−* 1)*/*214 *≈* 131,071.999.

Suppose that we are using a *p.q* fixed-point format, and let *f* = 2*q* . By the definition above, we can convert an integer or real number into *p.q* format by multiplying with *f* . For example, in 17.14 format the fraction 59/60 used in the calculation of *load avg*, above, is (59*/*60)214 = 16,111 (rounded to nearest). To convert a fixed-point value back to an integer, divide by *f* . (The normal ‘/’ operator in C rounds toward zero, that is, it rounds positive numbers down and negative numbers up. To round to nearest, add *f/*2 to a positive number, or subtract it from a negative number, before dividing.)

Many operations on fixed-point numbers are straightforward. Let x and y be fixed-point numbers, and let n be an integer. Then the sum of x and y is x + y and their difference is x

- y. The sum of x and n is x + n \* f; difference, x - n \* f; product, x \* n; quotient, x / n.

Multiplying two fixed-point values has two complications. First, the decimal point of the result is *q* bits too far to the left. Consider that (59*/*60)(59*/*60) should be slightly less than 1, but 16*,* 111 *×* 16*,* 111 = 259,564,321 is much greater than 214 = 16,384. Shifting *q* bits right, we get 259*,* 564*,* 321*/*214 = 15,842, or about 0.97, the correct answer. Second, the multiplication can overflow even though the answer is representable. For example, 64 in 17.14 format is 64 *×* 214 = 1,048,576 and its square 642 = 4,096 is well within the 17.14

range, but 1*,* 048*,* 5762 = 240, greater than the maximum signed 32-bit integer value 231 *−* 1. An easy solution is to do the multiplication as a 64-bit operation. The product of x and y is then ((int64\_t) x) \* y / f.

Dividing two fixed-point values has opposite issues. The decimal point will be too far to the right, which we fix by shifting the dividend *q* bits to the left before the division. The left shift discards the top *q* bits of the dividend, which we can again fix by doing the division in 64 bits. Thus, the quotient when x is divided by y is ((int64\_t) x) \* f / y.

1 Because we are working in binary, the “decimal” point might more correctly be called the “binary” point,

but the meaning should be clear.

This section has consistently used multiplication or division by *f* , instead of *q*-bit shifts, for two reasons. First, multiplication and division do not have the surprising operator precedence of the C shift operators. Second, multiplication and division are well-defined on negative operands, but the C shift operators are not. Take care with these issues in your implementation.

The following table summarizes how fixed-point arithmetic operations can be imple- mented in C. In the table, x and y are fixed-point numbers, n is an integer, fixed-point numbers are in signed *p.q* format where *p* + *q* = 31, and f is 1 << q:

Convert n to fixed point: n \* f

Convert x to integer (rounding toward zero): x / f

Convert x to integer (rounding to nearest): (x + f / 2) / f if x >= 0,

(x - f / 2) / f if x <= 0.

Add x and y: x + y

Subtract y from x: x - y

Add x and n: x + n \* f

Subtract n from x: x - n \* f

Multiply x by y: ((int64\_t) x) \* y / f

Multiply x by n: x \* n

Divide x by y: ((int64\_t) x) \* f / y

Divide x by n: x / n

# Appendix C Coding Standards

All of you should have taken a class like CS 107, so we expect you to be familiar with some set of coding standards such as [CS 107 Coding Standards](http://www.stanford.edu/class/cs140/projects/misc/CodingStandards.pdf). Even if you’ve taken 107, we recommend reviewing that document. We expect code at the “Peer-Review Quality” level described there.

Our standards for coding are most important for grading. We want to stress that aside from the fact that we are explicitly basing part of your grade on these things, good coding practices will improve the quality of your code. This makes it easier for your partners to interact with it, and ultimately, will improve your chances of having a good working program. That said once, the rest of this document will discuss only the ways in which our coding standards will affect our grading.

## Style

Style, for the purposes of our grading, refers to how readable your code is. At minimum, this means that your code is well formatted, your variable names are descriptive and your functions are decomposed and well commented. Any other factors which make it hard (or easy) for us to read or use your code will be reflected in your style grade.

The existing Pintos code is written in the GNU style and largely follows the [GNU Coding](http://www.gnu.org/prep/standards_toc.html) [Standards](http://www.gnu.org/prep/standards_toc.html). We encourage you to follow the applicable parts of them too, especially chapter 5, “Making the Best Use of C.” Using a different style won’t cause actual problems, but it’s ugly to see gratuitous differences in style from one function to another. If your code is too ugly, it will cost you points.

Please limit C source file lines to at most 79 characters long.

Pintos comments sometimes refer to external standards or specifications by writing a name inside square brackets, like this: [IA32-v3a]. These names refer to the reference names used in this documentation (see [[Bibliography], page 117](#_bookmark148)).

If you remove existing Pintos code, please delete it from your source file entirely. Don’t just put it into a comment or a conditional compilation directive, because that makes the resulting code hard to read.

We’re only going to do a compile in the directory for the project being submitted. You don’t need to make sure that the previous projects also compile.

Project code should be written so that all of the subproblems for the project function together, that is, without the need to rebuild with different macros defined, etc. If you do extra credit work that changes normal Pintos behavior so as to interfere with grading, then you must implement it so that it only acts that way when given a special command-line option of the form ‘-*name* ’, where *name* is a name of your choice. You can add such an option by modifying parse\_options() in ‘threads/init.c’.

The introduction describes additional coding style requirements (see [Section 1.2.2 [De-](#_bookmark8) [sign], page 6](#_bookmark8)).

## C99

The Pintos source code uses a few features of the “C99” standard library that were not in the original 1989 standard for C. Many programmers are unaware of these feature, so we will describe them. The new features used in Pintos are mostly in new headers:

‘<stdbool.h>’

Defines macros bool, a 1-bit type that takes on only the values 0 and 1, true, which expands to 1, and false, which expands to 0.

‘<stdint.h>’

On systems that support them, this header defines types int*n* \_t and uint*n* \_t for *n* = 8, 16, 32, 64, and possibly other values. These are 2’s complement signed and unsigned types, respectively, with the given number of bits.

On systems where it is possible, this header also defines types intptr\_t and

uintptr\_t, which are integer types big enough to hold a pointer.

On all systems, this header defines types intmax\_t and uintmax\_t, which are the system’s signed and unsigned integer types with the widest ranges.

For every signed integer type *type* \_t defined here, as well as for ptrdiff\_t defined in ‘<stddef.h>’, this header also defines macros *TYPE* \_MAX and *TYPE* \_ MIN that give the type’s range. Similarly, for every unsigned integer type *type* \_t defined here, as well as for size\_t defined in ‘<stddef.h>’, this header defines a *TYPE* \_MAX macro giving its maximum value.

‘<inttypes.h>’

‘<stdint.h>’ provides no straightforward way to format the types it defines with printf() and related functions. This header provides macros to help with that. For every int*n* \_t defined by ‘<stdint.h>’, it provides macros PRId*n* and PRIi*n* for formatting values of that type with d" and i". Similarly, for every uint*n* \_t, it provides PRIo*n* , PRIu*n* , PRIu*x* , and PRIu*X* .

You use these something like this, taking advantage of the fact that the C compiler concatenates adjacent string literals:

#include <inttypes.h>

...

int32\_t value = ...;

printf 08"PRId32"\n", value);

The ’ is not supplied by the PRI macros. As shown above, you supply it yourself and follow it by any flags, field width, etc.

‘<stdio.h>’

The printf() function has some new type modifiers for printing standard types: ‘j’ For intmax\_t (e.g. ‘jd’) or uintmax\_t (e.g. ‘ju’).

‘z’ For size\_t (e.g. zu’).

‘t’ For ptrdiff\_t (e.g. td’).

Pintos printf() also implements a nonstandard ‘’’ flag that groups large num- bers with commas to make them easier to read.

## Unsafe String Functions

A few of the string functions declared in the standard ‘<string.h>’ and ‘<stdio.h>’ headers are notoriously unsafe. The worst offenders are intentionally not included in the Pintos C library:

strcpy() When used carelessly this function can overflow the buffer reserved for its out- put string. Use strlcpy() instead. Refer to comments in its source code in lib/string.c for documentation.

strncpy()

This function can leave its destination buffer without a null string terminator. It also has performance problems. Again, use strlcpy().

strcat() Same issue as strcpy(). Use strlcat() instead. Again, refer to comments in its source code in lib/string.c for documentation.

strncat()

The meaning of its buffer size argument is surprising. Again, use strlcat().

strtok() Uses global data, so it is unsafe in threaded programs such as kernels. Use strtok\_r() instead, and see its source code in lib/string.c for documenta- tion and an example.

sprintf()

vsprintf()

Same issue as strcpy(). Use snprintf() instead. Refer to comments in

lib/stdio.h for documentation.

Same issue as strcpy(). Use vsnprintf() instead.

If you try to use any of these functions, the error message will give you a hint by referring to an identifier like dont\_use\_sprintf\_use\_snprintf.

# Appendix D Project Documentation

This chapter presents a sample assignment and a filled-in design document for one possible implementation. Its purpose is to give you an idea of what we expect to see in your own design documents.

## Sample Assignment

Implement thread\_join().

void thread\_join (*tid t tid* )

[Function]

Blocks the current thread until thread *tid* exits. If *A* is the running thread and *B* is the argument, then we say that “*A* joins *B*.”

Incidentally, the argument is a thread id, instead of a thread pointer, because a thread pointer is not unique over time. That is, when a thread dies, its memory may be, whether immediately or much later, reused for another thread. If thread *A* over time had two children *B* and *C* that were stored at the same address, then thread\_join(*B* ) and thread\_join(*C* ) would be ambiguous.

A thread may only join its immediate children. Calling thread\_join() on a thread that is not the caller’s child should cause the caller to return immediately. Children are not “inherited,” that is, if *A* has child *B* and *B* has child *C*, then *A* always returns immediately should it try to join *C*, even if *B* is dead.

A thread need not ever be joined. Your solution should properly free all of a thread’s resources, including its struct thread, whether it is ever joined or not, and regardless of whether the child exits before or after its parent. That is, a thread should be freed exactly once in all cases.

Joining a given thread is idempotent. That is, joining a thread multiple times is equivalent to joining it once, because it has already exited at the time of the later joins. Thus, joins on a given thread after the first should return immediately.

You must handle all the ways a join can occur: nested joins (*A* joins *B*, then *B* joins

*C*), multiple joins (*A* joins *B*, then *A* joins *C*), and so on.

## Sample Design Document

+-----------------+

| CS 140 |

| SAMPLE PROJECT |

| DESIGN DOCUMENT |

+-----------------+

---- GROUP ----

Ben Pfaff [<blp@stanford.edu>](mailto:blp@stanford.edu)

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for

>> the TAs, or extra credit, please give them here. (This is a sample design document.)

>> Please cite any offline or online sources you consulted while

>> preparing your submission, other than the Pintos documentation,

>> course text, and lecture notes. None.

JOIN

====

---- DATA STRUCTURES ----

>> 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.

A "latch" is a new synchronization primitive. Acquires block until the first release. Afterward, all ongoing and future acquires pass immediately.

/\* Latch. \*/ struct latch

{

bool released; /\* Released yet? \*/ struct lock monitor\_lock; /\* Monitor lock. \*/

struct condition rel\_cond; /\* Signaled when released. \*/

};

Added to struct thread:

/\* Members for implementing thread\_join(). \*/

struct latch ready\_to\_die; /\* Release when thread about to die. \*/ struct semaphore can\_die; /\* Up when thread allowed to die. \*/ struct list children; /\* List of child threads. \*/

list\_elem children\_elem; /\* Element of ‘children’ list. \*/

---- ALGORITHMS ----

>> Briefly describe your implementation of thread\_join() and how it

>> interacts with thread termination.

thread\_join() finds the joined child on the thread’s list of children and waits for the child to exit by acquiring the child’s

ready\_to\_die latch. When thread\_exit() is called, the thread releases its ready\_to\_die latch, allowing the parent to continue.

---- SYNCHRONIZATION ----

>> Consider parent thread P with child thread C. How do you ensure

>> proper synchronization and avoid race conditions when P calls wait(C)

>> before C exits? After C exits? How do you ensure that all resources

>> are freed in each case? How about when P terminates without waiting,

>> before C exits? After C exits? Are there any special cases?

C waits in thread\_exit() for P to die before it finishes its own exit, using the can\_die semaphore "down"ed by C and "up"ed by P as it exits. Regardless of whether whether C has terminated, there is no race on wait(C), because C waits for P’s permission before it frees itself.

Regardless of whether P waits for C, P still "up"s C’s can\_die semaphore when P dies, so C will always be freed. (However, freeing C’s resources is delayed until P’s death.)

The initial thread is a special case because it has no parent to wait for it or to "up" its can\_die semaphore. Therefore, its can\_die semaphore is initialized to 1.

---- RATIONALE ----

>> Critique your design, pointing out advantages and disadvantages in

>> your design choices.

This design has the advantage of simplicity. Encapsulating most of the synchronization logic into a new "latch" structure abstracts what little complexity there is into a separate layer, making the design easier to reason about. Also, all the new data members are in ‘struct thread’, with no need for any extra dynamic allocation, etc., that would require extra management code.

On the other hand, this design is wasteful in that a child thread cannot free itself before its parent has terminated. A parent thread that creates a large number of short-lived child threads could unnecessarily exhaust kernel memory. This is probably acceptable for implementing kernel threads, but it may be a bad idea for use with user processes because of the larger number of resources that user processes tend to own.

# Appendix E Debugging Tools

Many tools lie at your disposal for debugging Pintos. This appendix introduces you to a few of them.

### printf()

Don’t underestimate the value of printf(). The way printf() is implemented in Pintos, you can call it from practically anywhere in the kernel, whether it’s in a kernel thread or an interrupt handler, almost regardless of what locks are held.

printf() is useful for more than just examining data. It can also help figure out when and where something goes wrong, even when the kernel crashes or panics without a useful error message. The strategy is to sprinkle calls to printf() with different strings (e.g. "<1>", "<2>", . . . ) throughout the pieces of code you suspect are failing. If you don’t even see <1> printed, then something bad happened before that point, if you see <1> but not <2>, then something bad happened between those two points, and so on. Based on what you learn, you can then insert more printf() calls in the new, smaller region of code you suspect. Eventually you can narrow the problem down to a single statement. See [Section E.6 [Triple Faults], page 108](#_bookmark136), for a related technique.

### ASSERT

Assertions are useful because they can catch problems early, before they’d otherwise be noticed. Ideally, each function should begin with a set of assertions that check its arguments for validity. (Initializers for functions’ local variables are evaluated before assertions are checked, so be careful not to assume that an argument is valid in an initializer.) You can also sprinkle assertions throughout the body of functions in places where you suspect things are likely to go wrong. They are especially useful for checking loop invariants.

Pintos provides the ASSERT macro, defined in ‘<debug.h>’, for checking assertions.

ASSERT (*expression*) [Macro]

Tests the value of *expression*. If it evaluates to zero (false), the kernel panics. The panic message includes the expression that failed, its file and line number, and a backtrace, which should help you to find the problem. See [Section E.4 [Backtraces],](#_bookmark130) [page 100](#_bookmark130), for more information.

* 1. **Function and Parameter Attributes**

These macros defined in ‘<debug.h>’ tell the compiler special attributes of a function or function parameter. Their expansions are GCC-specific.

UNUSED [Macro]

Appended to a function parameter to tell the compiler that the parameter might not be used within the function. It suppresses the warning that would otherwise appear.

NO\_RETURN [Macro]

Appended to a function prototype to tell the compiler that the function never returns. It allows the compiler to fine-tune its warnings and its code generation.

NO\_INLINE [Macro]

Appended to a function prototype to tell the compiler to never emit the function in-line. Occasionally useful to improve the quality of backtraces (see below).

PRINTF\_FORMAT (*format, first* )

[Macro]

Appended to a function prototype to tell the compiler that the function takes a printf()-like format string as the argument numbered *format* (starting from 1) and that the corresponding value arguments start at the argument numbered *first*. This lets the compiler tell you if you pass the wrong argument types.

## Backtraces

When the kernel panics, it prints a “backtrace,” that is, a summary of how your program got where it is, as a list of addresses inside the functions that were running at the time of the panic. You can also insert a call to debug\_backtrace(), prototyped in ‘<debug.h>’, to print a backtrace at any point in your code.

The addresses in a backtrace are listed as raw hexadecimal numbers, which are difficult to interpret. We provide a tool called backtrace to translate these into function names and source file line numbers. Give it the name of your ‘kernel.o’ as the first argument and the hexadecimal numbers composing the backtrace (including the ‘0x’ prefixes) as the remaining arguments. It outputs the function name and source file line numbers that correspond to each address.

If the translated form of a backtrace is garbled, or doesn’t make sense (e.g. function A is listed above function B, but B doesn’t call A), then it’s a good sign that you’re corrupting a kernel thread’s stack, because the backtrace is extracted from the stack. Alternatively, it could be that the ‘kernel.o’ you passed to backtrace is not the same kernel that produced the backtrace.

Sometimes backtraces can be confusing without any corruption. Compiler optimizations can cause surprising behavior. When a function has called another function as its final action (a *tail call*), the calling function may not appear in a backtrace at all. Similarly, when function A calls another function B that never returns, the compiler may optimize such that an unrelated function C appears in the backtrace instead of A. Function C is simply the function that happens to be in memory just after A. In the threads project, this is commonly seen in backtraces for test failures; see [[pass() Fails], page 17](#_bookmark21), for more information.

#### E.4.1 Example

Here’s an example. Suppose that Pintos printed out this following call stack, which is taken from an actual Pintos submission for the file system project:

Call stack: 0xc0106eff 0xc01102fb 0xc010dc22 0xc010cf67 0xc0102319 0xc010325a 0x804812c 0x8048a96 0x8048ac8.

You would then invoke the backtrace utility like shown below, cutting and pasting the backtrace information into the command line. This assumes that ‘kernel.o’ is in the current directory. You would of course enter all of the following on a single shell command line, even though that would overflow our margins here:

backtrace kernel.o 0xc0106eff 0xc01102fb 0xc010dc22 0xc010cf67

0xc0102319 0xc010325a 0x804812c 0x8048a96 0x8048ac8

The backtrace output would then look something like this: 0xc0106eff: debug\_panic (lib/debug.c:86) 0xc01102fb: file\_seek (filesys/file.c:405)

0xc010dc22: seek (userprog/syscall.c:744)

0xc010cf67: syscall\_handler (userprog/syscall.c:444) 0xc0102319: intr\_handler (threads/interrupt.c:334) 0xc010325a: intr\_entry (threads/intr-stubs.S:38) 0x0804812c: (unknown)

0x08048a96: (unknown) 0x08048ac8: (unknown)

(You will probably not see exactly the same addresses if you run the command above on your own kernel binary, because the source code you compiled and the compiler you used are probably different.)

The first line in the backtrace refers to debug\_panic(), the function that implements kernel panics. Because backtraces commonly result from kernel panics, debug\_panic() will often be the first function shown in a backtrace.

The second line shows file\_seek() as the function that panicked, in this case as the result of an assertion failure. In the source code tree used for this example, line 405 of ‘filesys/file.c’ is the assertion

ASSERT (file\_ofs >= 0);

(This line was also cited in the assertion failure message.) Thus, file\_seek() panicked because it passed a negative file offset argument.

The third line indicates that seek() called file\_seek(), presumably without validating the offset argument. In this submission, seek() implements the seek system call.

The fourth line shows that syscall\_handler(), the system call handler, invoked seek(). The fifth and sixth lines are the interrupt handler entry path.

The remaining lines are for addresses below PHYS\_BASE. This means that they refer to addresses in the user program, not in the kernel. If you know what user program was running when the kernel panicked, you can re-run backtrace on the user program, like so: (typing the command on a single line, of course):

backtrace tests/filesys/extended/grow-too-big 0xc0106eff 0xc01102fb 0xc010dc22 0xc010cf67 0xc0102319 0xc010325a 0x804812c 0x8048a96

0x8048ac8

The results look like this: 0xc0106eff: (unknown) 0xc01102fb: (unknown)

0xc010dc22: (unknown)

0xc010cf67: (unknown) 0xc0102319: (unknown) 0xc010325a: (unknown)

0x0804812c: test\_main (...xtended/grow-too-big.c:20) 0x08048a96: main (tests/main.c:10)

0x08048ac8: \_start (lib/user/entry.c:9)

You can even specify both the kernel and the user program names on the command line, like so:

backtrace kernel.o tests/filesys/extended/grow-too-big 0xc0106eff 0xc01102fb 0xc010dc22 0xc010cf67 0xc0102319 0xc010325a 0x804812c 0x8048a96 0x8048ac8

The result is a combined backtrace:

In kernel.o:

0xc0106eff: debug\_panic (lib/debug.c:86) 0xc01102fb: file\_seek (filesys/file.c:405) 0xc010dc22: seek (userprog/syscall.c:744)

0xc010cf67: syscall\_handler (userprog/syscall.c:444) 0xc0102319: intr\_handler (threads/interrupt.c:334) 0xc010325a: intr\_entry (threads/intr-stubs.S:38)

In tests/filesys/extended/grow-too-big:

0x0804812c: test\_main (...xtended/grow-too-big.c:20) 0x08048a96: main (tests/main.c:10)

0x08048ac8: \_start (lib/user/entry.c:9)

Here’s an extra tip for anyone who read this far: backtrace is smart enough to strip the Call stack: header and ‘.’ trailer from the command line if you include them. This can save you a little bit of trouble in cutting and pasting. Thus, the following command prints the same output as the first one we used:

backtrace kernel.o Call stack: 0xc0106eff 0xc01102fb 0xc010dc22 0xc010cf67 0xc0102319 0xc010325a 0x804812c 0x8048a96 0x8048ac8.

## GDB

You can run Pintos under the supervision of the GDB debugger. First, start Pintos with the ‘--gdb’ option, e.g. pintos --gdb -- run mytest. Second, open a second terminal on the same machine and use pintos-gdb to invoke GDB on ‘kernel.o’:1

pintos-gdb kernel.o

and issue the following GDB command:

target remote localhost:1234

Now GDB is connected to the simulator over a local network connection. You can now issue any normal GDB commands. If you issue the ‘c’ command, the simulated BIOS will take control, load Pintos, and then Pintos will run in the usual way. You can pause the process at any point with *(*Ctrl+C*)*.

#### Using GDB

You can read the GDB manual by typing info gdb at a terminal command prompt. Here’s a few commonly useful GDB commands:

##### c

Continues execution until *(*Ctrl+C*)* or the next breakpoint.

[GDB Command]

1 pintos-gdb is a wrapper around gdb (80*x*86) or i386-elf-gdb (SPARC) that loads the Pintos macros

at startup.

break *function* break *file:line* break *\*address*

[GDB Command] [GDB Command] [GDB Command]

Sets a breakpoint at *function*, at *line* within *file*, or *address*. (Use a ‘0x’ prefix to specify an address in hex.)

Use break main to make GDB stop when Pintos starts running.

p *expression* [GDB Command]

Evaluates the given *expression* and prints its value. If the expression contains a function call, that function will actually be executed.

l *\*address* [GDB Command]

Lists a few lines of code around *address*. (Use a ‘0x’ prefix to specify an address in hex.)

bt [GDB Command]

Prints a stack backtrace similar to that output by the backtrace program described above.

p/a *address* [GDB Command]

Prints the name of the function or variable that occupies *address*. (Use a ‘0x’ prefix to specify an address in hex.)

diassemble *function*

Disassembles *function*.

[GDB Command]

We also provide a set of macros specialized for debugging Pintos, written by Godmar Back [gback@cs.vt.edu](mailto:gback@cs.vt.edu). You can type help user-defined for basic help with the macros. Here is an overview of their functionality, based on Godmar’s documentation:

debugpintos [GDB Macro]

Attach debugger to a waiting pintos process on the same machine. Shorthand for

target remote localhost:1234.

dumplist *list type element* [GDB Macro]

Prints the elements of *list*, which should be a struct list that contains elements of the given *type* (without the word struct) in which *element* is the struct list\_elem member that links the elements.

Example: dumplist all\_list thread all\_elem prints all elements of struct thread that are linked in struct list all\_list using the struct list\_elem all\_elem which is part of struct thread. (This assumes that you have added all\_list and all\_elem yourself.)

btthread *thread* [GDB Macro]

Shows the backtrace of *thread*, which is a pointer to the struct thread of the thread whose backtrace it should show. For the current thread, this is identical to the bt (backtrace) command. It also works for any thread suspended in schedule(), provided you know where its kernel stack page is located.

btthreadlist *list element* [GDB Macro]

Shows the backtraces of all threads in *list*, the struct list in which the threads are kept. Specify *element* as the struct list\_elem field used inside struct thread to link the threads together.

Example: btthreadlist all\_list all\_elem shows the backtraces of all threads con- tained in struct list all\_list, linked together by all\_elem. This command is useful to determine where your threads are stuck when a deadlock occurs. Please see the example scenario below. (This assumes that you have added all\_list and all\_elem yourself.)

btpagefault [GDB Macro]

Print a backtrace of the current thread after a page fault exception. Normally, when a page fault exception occurs, GDB will stop with a message that might say:

Program received signal 0, Signal 0. 0xc0102320 in intr0e\_stub ()

In that case, the bt command might not give a useful backtrace. Use btpagefault

instead.

You may also use btpagefault for page faults that occur in a user process. In this case, you may also wish to load the user program’s symbol table (see [Section E.5.3](#_bookmark135) [[Debugging User Programs], page 107](#_bookmark135)).

hook-stop [GDB Macro]

GDB invokes this macro every time the simulation stops, which Bochs will do for every processor exception, among other reasons. If the simulation stops due to a page fault, hook-stop will print a message that says and explains further whether the page fault occurred in the kernel or in user code.

If the exception occurred from user code, hook-stop will say:

pintos-debug: a page fault exception occurred in user mode

pintos-debug: hit ’c’ to continue, or ’s’ to step to intr\_handler

In Project 2, a page fault in a user process leads to the termination of the process. You should expect those page faults to occur in the robustness tests where we test that your kernel properly terminates processes that try to access invalid addresses. To debug those, set a break point in page\_fault() in ‘exception.c’, which you will need to modify accordingly.

In Project 3, a page fault in a user process no longer automatically leads to the termination of a process. Instead, it may require reading in data for the page the process was trying to access, either because it was swapped out or because this is the first time it’s accessed. In either case, you will reach page\_fault() and need to take the appropriate action there.

If the page fault did not occur in user mode while executing a user process, then it occurred in kernel mode while executing kernel code. In this case, hook-stop will print this message:

pintos-debug: a page fault occurred in kernel mode

followed by the output of the btpagefault command.

Before Project 3, a page fault exception in kernel code is always a bug in your kernel, because your kernel should never crash. Starting with Project 3, the situation will change if you use get\_user() and put\_user() strategy to verify user memory accesses (see [Section 3.1.5 [Accessing User Memory], page 26](#_bookmark30)).

If you don’t want GDB to stop for page faults, then issue the command handle SIGSEGV nostop. GDB will still print a message for every page fault, but it will not come back to a command prompt.

#### Example GDB Session

This section narrates a sample GDB session, provided by Godmar Back. This example illustrates how one might debug a Project 1 solution in which occasionally a thread that calls timer\_sleep() is not woken up. With this bug, tests such as mlfqs\_load\_1 get stuck.

This session was captured with a slightly older version of Bochs and the GDB macros for Pintos, so it looks slightly different than it would now. Program output is shown in normal type, user input in **strong** type.

First, I start Pintos:

$ **pintos -v –gdb – -q -mlfqs run mlfqs-load-1**

Writing command line to /tmp/gDAlqTB5Uf.dsk... bochs -q

========================================================================

Bochs x86 Emulator 2.2.5

Build from CVS snapshot on December 30, 2005

========================================================================

00000000000i[ ] reading configuration from bochsrc.txt 00000000000i[ ] Enabled gdbstub

00000000000i[ ] installing nogui module as the Bochs GUI 00000000000i[ ] using log file bochsout.txt

Waiting for gdb connection on localhost:1234

Then, I open a second window on the same machine and start GDB:

$ **pintos-gdb kernel.o**

GNU gdb Red Hat Linux (6.3.0.0-1.84rh) Copyright 2004 Free Software Foundation, Inc.

GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions. Type "show copying" to see the conditions.

There is absolutely no warranty for GDB. Type "show warranty" for details. This GDB was configured as "i386-redhat-linux-gnu"...

Using host libthread\_db library "/lib/libthread\_db.so.1".

Then, I tell GDB to attach to the waiting Pintos emulator:

(gdb) **debugpintos**

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

Reply contains invalid hex digit 78

Now I tell Pintos to run by executing c (short for continue) twice:

(gdb) **c**

Continuing.

Reply contains invalid hex digit 78 (gdb) **c**

Continuing.

Now Pintos will continue and output:

Pintos booting with 4,096 kB RAM...

Kernel command line: -q -mlfqs run mlfqs-load-1

374 pages available in kernel pool.

373 pages available in user pool. Calibrating timer... 102,400 loops/s. Boot complete.

Executing ’mlfqs-load-1’: (mlfqs-load-1) begin

(mlfqs-load-1) spinning for up to 45 seconds, please wait... (mlfqs-load-1) load average rose to 0.5 after 42 seconds (mlfqs-load-1) sleeping for another 10 seconds, please wait...

. . . until it gets stuck because of the bug I had introduced. I hit *(*Ctrl+C*)* in the debugger window:

Program received signal 0, Signal 0.

0xc010168c in next\_thread\_to\_run () at ../../threads/thread.c:649 649 while (i <= PRI\_MAX && list\_empty (&ready\_list[i]))

(gdb)

The thread that was running when I interrupted Pintos was the idle thread. If I run

backtrace, it shows this backtrace:

(gdb) **bt**

#0 0xc010168c in next\_thread\_to\_run () at ../../threads/thread.c:649

#1 0xc0101778 in schedule () at ../../threads/thread.c:714

#2 0xc0100f8f in thread\_block () at ../../threads/thread.c:324

#3 0xc0101419 in idle (aux=0x0) at ../../threads/thread.c:551

#4 0xc010145a in kernel\_thread (function=0xc01013ff , aux=0x0) at ../../threads/thread.c:575

#5 0x00000000 in ?? ()

Not terribly useful. What I really like to know is what’s up with the other thread (or threads). Since I keep all threads in a linked list called all\_list, linked together by a struct list\_elem member named all\_elem, I can use the btthreadlist macro from the macro library I wrote. btthreadlist iterates through the list of threads and prints the backtrace for each thread:

(gdb) **btthreadlist all list all elem**

pintos-debug: dumping backtrace of thread ’main’ @0xc002f000

#0 0xc0101820 in schedule () at ../../threads/thread.c:722

#1 0xc0100f8f in thread\_block () at ../../threads/thread.c:324

#2 0xc0104755 in timer\_sleep (ticks=1000) at ../../devices/timer.c:141

#3 0xc010bf7c in test\_mlfqs\_load\_1 () at ../../tests/threads/mlfqs-load-1.c:49

#4 0xc010aabb in run\_test (name=0xc0007d8c "mlfqs-load-1") at ../../tests/threads/tests.c:50

#5 0xc0100647 in run\_task (argv=0xc0110d28) at ../../threads/init.c:281

#6 0xc0100721 in run\_actions (argv=0xc0110d28) at ../../threads/init.c:331

#7 0xc01000c7 in main () at ../../threads/init.c:140

pintos-debug: dumping backtrace of thread ’idle’ @0xc0116000

#0 0xc010168c in next\_thread\_to\_run () at ../../threads/thread.c:649

#1 0xc0101778 in schedule () at ../../threads/thread.c:714

#2 0xc0100f8f in thread\_block () at ../../threads/thread.c:324

#3 0xc0101419 in idle (aux=0x0) at ../../threads/thread.c:551

#4 0xc010145a in kernel\_thread (function=0xc01013ff , aux=0x0) at ../../threads/thread.c:575

#5 0x00000000 in ?? ()

In this case, there are only two threads, the idle thread and the main thread. The kernel stack pages (to which the struct thread points) are at 0xc0116000 and 0xc002f000, respectively. The main thread is stuck in timer\_sleep(), called from test\_mlfqs\_load\_1. Knowing where threads are stuck can be tremendously useful, for instance when diag-

nosing deadlocks or unexplained hangs.

#### Debugging User Programs

You can also use GDB to debug a user program running under Pintos. Start by issuing this GDB command to load the program’s symbol table:

add-symbol-file *program*

where *program* is the name of the program’s executable (in the host file system, not in the Pintos file system). After this, you should be able to debug the user program the same way you would the kernel, by placing breakpoints, inspecting data, etc. Your actions apply to every user program running in Pintos, not just to the one you want to debug, so be careful in interpreting the results. Also, a name that appears in both the kernel and the user program will actually refer to the kernel name. (The latter problem can be avoided by giving the user executable name on the GDB command line, instead of ‘kernel.o’, and then using add-symbol-file to load ‘kernel.o’.)

#### FAQ

GDB can’t connect to Bochs.

If the target remote command fails, then make sure that both GDB and pintos are running on the same machine by running hostname in each ter- minal. If the names printed differ, then you need to open a new terminal for GDB on the machine running pintos.

GDB doesn’t recognize any of the macros.

If you start GDB with pintos-gdb, it should load the Pintos macros automat- ically. If you start GDB some other way, then you must issue the command source *pintosdir* /src/misc/gdb-macros, where *pintosdir* is the root of your Pintos directory, before you can use them.

Can I debug Pintos with DDD?

Yes, you can. DDD invokes GDB as a subprocess, so you’ll need to tell it to invokes pintos-gdb instead:

ddd --gdb --debugger pintos-gdb

Can I use GDB inside Emacs?

Yes, you can. Emacs has special support for running GDB as a subprocess. Type *M-x gdb* and enter your pintos-gdb command at the prompt. The Emacs manual has information on how to use its debugging features in a section titled “Debuggers.”

GDB is doing something weird.

If you notice strange behavior while using GDB, there are three possibilities: a bug in your modified Pintos, a bug in Bochs’s interface to GDB or in GDB itself, or a bug in the original Pintos code. The first and second are quite likely, and you should seriously consider both. We hope that the third is less likely, but it is also possible.

## Triple Faults

When a CPU exception handler, such as a page fault handler, cannot be invoked because it is missing or defective, the CPU will try to invoke the “double fault” handler. If the double fault handler is itself missing or defective, that’s called a “triple fault.” A triple fault causes an immediate CPU reset.

Thus, if you get yourself into a situation where the machine reboots in a loop, that’s probably a “triple fault.” In a triple fault situation, you might not be able to use printf() for debugging, because the reboots might be happening even before everything needed for printf() is initialized.

There are at least two ways to debug triple faults. First, you can run Pintos in Bochs under GDB (see [Section E.5 [GDB], page 102](#_bookmark132)). If Bochs has been built properly for Pintos, a triple fault under GDB will cause it to print the message “Triple fault: stopping for gdb” on the console and break into the debugger. (If Bochs is not running under GDB, a triple fault will still cause it to reboot.) You can then inspect where Pintos stopped, which is where the triple fault occurred.

Another option is what I call “debugging by infinite loop.” Pick a place in the Pintos code, insert the infinite loop for (;;); there, and recompile and run. There are two likely possibilities:

* + - The machine hangs without rebooting. If this happens, you know that the infinite loop is running. That means that whatever caused the reboot must be *after* the place you inserted the infinite loop. Now move the infinite loop later in the code sequence.
    - The machine reboots in a loop. If this happens, you know that the machine didn’t make it to the infinite loop. Thus, whatever caused the reboot must be *before* the place you inserted the infinite loop. Now move the infinite loop earlier in the code sequence.

If you move around the infinite loop in a “binary search” fashion, you can use this technique to pin down the exact spot that everything goes wrong. It should only take a few minutes at most.

## Modifying Bochs

An advanced debugging technique is to modify and recompile the simulator. This proves useful when the simulated hardware has more information than it makes available to the OS. For example, page faults have a long list of potential causes, but the hardware does not report to the OS exactly which one is the particular cause. Furthermore, a bug in the kernel’s handling of page faults can easily lead to recursive faults, but a “triple fault” will cause the CPU to reset itself, which is hardly conducive to debugging.

In a case like this, you might appreciate being able to make Bochs print out more debug information, such as the exact type of fault that occurred. It’s not very hard. You start by retrieving the source code for Bochs 2.2.6 from [http://bochs.sourceforge.net](http://bochs.sourceforge.net/) and extracting it into a directory. Then read ‘pintos/src/misc/bochs-2.2.6.README’ and apply the patches needed. Then run ‘./configure’, supplying the options you want (some suggestions are in the patch file). Finally, run make. This will compile Bochs and eventually produce a new binary ‘bochs’. To use your ‘bochs’ binary with pintos, put it in your PATH, and make sure that it is earlier than ‘/usr/class/cs140/‘uname -m‘/bochs’.

Of course, to get any good out of this you’ll have to actually modify Bochs. Instructions for doing this are firmly out of the scope of this document. However, if you want to debug page faults as suggested above, a good place to start adding printf()s is BX\_CPU\_ C::dtranslate\_linear() in ‘cpu/paging.cc’.

## Tips

The page allocator in ‘threads/palloc.c’ and the block allocator in ‘threads/malloc.c’ clear all the bytes in memory to 0xcc at time of free. Thus, if you see an attempt to dereference a pointer like 0xcccccccc, or some other reference to 0xcc, there’s a good chance you’re trying to reuse a page that’s already been freed. Also, byte 0xcc is the CPU opcode for “invoke interrupt 3,” so if you see an error like Interrupt 0x03 (#BP Breakpoint Exception), then Pintos tried to execute code in a freed page or block.

An assertion failure on the expression sec\_no < d->capacity indicates that Pintos tried to access a file through an inode that has been closed and freed. Freeing an inode clears its starting sector number to 0xcccccccc, which is not a valid sector number for disks smaller than about 1.6 TB.

# Appendix F Development Tools

Here are some tools that you might find useful while developing code.

## Tags

Tags are an index to the functions and global variables declared in a program. Many editors, including Emacs and vi, can use them. The ‘Makefile’ in ‘pintos/src’ produces Emacs-style tags with the command make TAGS or vi-style tags with make tags.

In Emacs, use *M-.* to follow a tag in the current window, *C-x 4 .* in a new window, or *C-x 5 .* in a new frame. If your cursor is on a symbol name for any of those commands, it becomes the default target. If a tag name has multiple definitions, *M-0 M-.* jumps to the next one. To jump back to where you were before you followed the last tag, use *M-\**.

## CVS

CVS is a version-control system. That is, you can use it to keep track of multiple versions of files. The idea is that you do some work on your code and test it, then check it into the version-control system. If you decide that the work you’ve done since your last check-in is no good, you can easily revert to the last checked-in version. Furthermore, you can retrieve any old version of your code as of some given day and time. The version control logs tell you who made changes and when.

CVS is not the best version control system out there, but it’s free, it’s fairly easy to use, and it’s already available on the Leland machines you’re using for the projects.

For more information, visit the [CVS home page](https://www.cvshome.org/).

#### Setting Up CVS

To set up CVS for use with Pintos on the Leland machines, start by choosing one group member as the keeper of the CVS repository. Everyone in the group will be able to use the CVS repository, but the keeper will actually create the repository, keep its files in his or her home directory, and maintain permissions for its contents.

The keeper has to perform several steps to set up the repository. First, create a new AFS group for the repository by executing ‘pts creategroup *keeper* :pintos-cvs’, where *keeper* is the keeper’s Leland username. Then, add each group member to the new group by repeatedly using the command ‘pts adduser -user *username* -group *keeper* :pintos-cvs’, where *username* is the name of a group member. After the group is created and its members added, ‘pts membership *keeper* :pintos-cvs’ should report that each group member is a member of the ‘*keeper* :pintos-cvs’ group.

The keeper now creates the repository directory and gives the group members access to it. We will assume that the repository will be in a directory called ‘cvs’ in the keeper’s home di- rectory. First create this directory with ‘mkdir $HOME/cvs’, then give group members access to it with ‘fs setacl -dir $HOME/cvs -acl *keeper* :pintos-cvs write’. Group members also need to be able to look up the ‘cvs’ directory in the keeper’s home directory, which can be enabled via ‘fs setacl -dir $HOME -acl *keeper* :pintos-cvs l’ (that’s letter “ell,” not digit “one.”).1

1 This command will allow group members to list the files in your home directory, but not read or write them. It should not create a security risk unless the names of files in your home directory are secret.

Now initialize the repository. To initialize the repository, execute ‘cvs -d $HOME/cvs init’.

Finally, import the Pintos sources into the newly initialized repository. If you have an existing set of Pintos sources you want to add to the repository, ‘cd’ to its ‘pintos’ directory now. Otherwise, to import the base Pintos source tree, ‘cd’ to ‘/usr/class/cs140/pintos/pintos’ (note the doubled ‘pintos’). After changing the current directory, execute this command:

cvs -d $HOME/cvs import -m "Imported sources" pintos foobar start

Here is a summary of the commands you have now executed:

pts creategroup *keeper* :pintos-cvs

pts adduser -user *username* -group *keeper* :pintos-cvs mkdir $HOME/cvs

fs setacl -dir $HOME/cvs -acl *keeper* :pintos-cvs write fs setacl -dir $HOME -acl *keeper* :pintos-cvs l

cvs -d $HOME/cvs init

cd /usr/class/cs140/pintos/pintos

cvs -d $HOME/cvs import -m "Imported sources" pintos foobar start

The repository is now ready for use by any group member, as described below. Keep in mind that the repository should only be accessed using CVS commands—it is not generally useful to examine them by hand, and you should definitely not modify them yourself.

#### Using CVS

To use CVS, start by check out a working copy of the contents of the CVS repository into a directory named ‘*dir* ’. To do so, execute ‘cvs -d ~*keeper* /cvs checkout -d *dir* pintos’, where *keeper* is the CVS keeper’s Leland username.

(If this fails due to some kind of permission problem, then run aklog and try again. If it still doesn’t work, log out and back in. If that still doesn’t fix the problem, the CVS repository may not be initialized properly.)

At this point, you can modify any of the files in the working copy. You can see the changes you’ve made with ‘cvs diff -u’. If you want to commit these changes back to the repository, making them visible to the other group members, you can use the CVS commit command. Within the ‘pintos’ directory, execute ‘cvs commit’. This will figure out the files that have been changed and fire up a text editor for you to describe the changes. By default, this editor is ‘vi’, but you can select a different editor by setting the CVSEDITOR environment variable, e.g. with ‘setenv CVSEDITOR emacs’ (add this line to your ‘.cvsrc’ to make it permanent).

Suppose another group member has committed changes. You can see the changes com- mitted to the repository since the time you checked it out (or updated from it) with ‘cvs diff -u -r BASE -r HEAD’. You can merge those change into your working copy using ‘cvs update’. If any of your local changes conflict with the committed changes, the CVS com- mand output should tell you. In that case, edit the files that contain conflicts, looking for ‘<<<’ and ‘>>>’ that denote the conflicts, and fix the problem.

You can view the history of *file* in your working directory, including the log messages, with ‘cvs log *file* ’.

You can give a particular set of file versions a name called a *tag*. First ‘cd’ to the root of the working copy, then execute ‘cvs tag *name* ’. It’s best to have no local changes in the working copy when you do this, because the tag will not include uncommitted changes. To recover the tagged repository later, use the ‘checkout’ command in the form ‘cvs -d

~*keeper* /cvs checkout -r *tag* -d *dir* pintos’, where *keeper* is the username of the CVS keeper and *dir* is the directory to put the tagged repository into.

If you add a new file to the source tree, you’ll need to add it to the repository with ‘cvs add *file* ’. This command does not have lasting effect until the file is committed later with ‘cvs commit’.

To remove a file from the source tree, first remove it from the file system with rm, then tell CVS with ‘cvs remove *file* ’. Again, only ‘cvs commit’ will make the change permanent.

To discard your local changes for a given file, without committing them, use ‘cvs update

-C *file* ’.

To check out a version of your repository as of a particular date, use the command ‘cvs -d ~*keeper* /cvs checkout -D ’*date* ’ -d *dir* pintos’, where *keeper* is the username of the CVS keeper and *dir* is the directory to put the tagged repository into.. A typical format for *date* is ‘YYYY-MM-DD HH:MM’, but CVS accepts several formats, even something like ‘1 hour ago’.

For more information, visit the [CVS home page](https://www.cvshome.org/).

#### CVS Locking

You might occasionally see a message like this while using CVS:

waiting for blp’s lock in /afs/ir/users/b/l/blp/cvs

This normally means that more than one user is accessing the repository at the same time. CVS should automatically retry after 30 seconds, at which time the operation should normally be able to continue.

If you encounter a long wait for a lock, of more than a minute or so, it may indicate that a CVS command did not complete properly and failed to remove its locks. If you think that this is the case, ask the user in question about it. If it appears that an operation did go awry, then you (or the named user) can delete files whose names start with ‘#cvs.rfl’, ‘#cvs.wfl’, or ‘#cvs.lock’ in the directory mentioned in the message. Doing so should allow your operation to proceed. Do not delete or modify other files.

## SourceForge

SourceForge is a web-based system for facilitating software development. It provides you with a version-control system (typically CVS, as described above) and other tools for track- ing your software. You can use it to store files, track bugs, and post notes about development progress. You can set up your own project in SourceForge at [sourceforge.net](http://sourceforge.net/).

## VNC

VNC stands for Virtual Network Computing. It is, in essence, a remote display system which allows you to view a computing “desktop” environment not only on the machine where it is running, but from anywhere on the Internet and from a wide variety of machine

architectures. It is already installed on the Leland machines. For more information, look at the [VNC Home Page](http://www.realvnc.com/).

# Appendix G Installing Pintos

This chapter explains how to install a Pintos development environment on your own ma- chine. If you are using a Pintos development environment that has been set up by someone else, you do not need to read this chapter or follow these instructions.

The Pintos development environment is targeted at Unix-like systems. It has been most extensively tested on GNU/Linux, in particular the Debian and Ubuntu distributions, and Solaris. It is not designed to install under any form of Windows.

Prerequisites for installing a Pintos development environment include the following, on top of standard Unix utilities:

* + - Required: [GCC](http://gcc.gnu.org/). Version 4.0 or later is preferred. Version 3.3 or later should work. If the host machine has an 80*x*86 processor, then GCC should be available as gcc; otherwise, an 80*x*86 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’.
    - Required: [GNU binutils](http://www.gnu.org/software/binutils/). Pintos uses addr2line, ar, ld, objcopy, and ranlib. If the host machine is not an 80*x*86, versions targeting 80*x*86 should be available with an ‘i386-elf-’ prefix.
    - Required: [Perl](http://www.perl.org/). Version 5.8.0 or later is preferred. Version 5.6.1 or later should work.
    - Required: [GNU make](http://www.gnu.org/software/make/), version 3.80 or later.
    - Recommended: [QEMU](http://fabrice.bellard.free.fr/qemu/), version 0.8.0 or later. If QEMU is not available, Bochs can be used, but its slowness is frustrating.
    - Recommended: [GDB](http://www.gnu.org/software/gdb/). GDB is helpful in debugging (see [Section E.5 [GDB], page 102](#_bookmark132)). If the host machine is not an 80*x*86, a version of GDB targeting 80*x*86 should be available as ‘i386-elf-gdb’.
    - Recommended: [X](http://www.x.org/). Being able to use an X server makes the virtual machine feel more like a physical machine, but it is not strictly necessary.
    - Optional: [Texinfo](http://www.gnu.org/software/texinfo/), version 4.5 or later. Texinfo is required to build the PDF version of the documentation.
    - Optional: [TEX](http://www.tug.org/). Also required to build the PDF version of the documentation.
    - Optional: [VMware Player](http://www.vmware.com/). This is a third platform that can also be used to test Pintos. Once these prerequisites are available, follow these instructions to install Pintos:

1. Install [Bochs](http://bochs.sourceforge.net/), version 2.2.6, as described below (see [Section G.1 [Building Bochs for](#_bookmark147)

[Pintos], page 115](#_bookmark147)).

1. Install scripts from ‘src/utils’. Copy ‘backtrace’, ‘pintos’, ‘pintos-gdb’, ‘pintos-mkdisk’ into the default PATH.
2. Install ‘src/misc/gdb-macros’ in a public location. Then use a text editor to edit the installed copy of ‘pintos-gdb’, 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.
3. Compile the remaining Pintos utilities by typing make in ‘src/utils’. Install ‘squish-pty’ somewhere in PATH. To support VMware Player, install ‘squish-unix’. If your Perl is older than version 5.8.0, also install ‘setitimer-helper’; otherwise, it is unneeded.
4. Pintos should now be ready for use. If you have the Pintos reference solutions, which are provided only to faculty and their teaching assistants, then you may test your installation by running make check in the top-level ‘tests’ directory. The tests take between 20 minutes and 1 hour to run, depending on the speed of your hardware.
5. Optional: Build the documentation, by running make dist in the top-level ‘doc’ di- rectory. This creates a ‘WWW’ subdirectory within ‘doc’ that contains HTML and PDF versions of the documentation, plus the design document templates and various hard- ware specifications referenced by the documentation. Building the PDF version of the manual requires Texinfo and TEX (see above). You may install ‘WWW’ wherever you find most useful.

The ‘doc’ directory is not included in the ‘.tar.gz’ distributed for Pintos. It is in the

Pintos CVS tree available via :pserver:anonymous@footstool.stanford.edu:/var/lib/cvs, in the pintos module. The CVS tree is *not* the authoritative source for Stanford

course materials, which should be obtained from the course website.

## G.1 Building Bochs for Pintos

Upstream Bochs has bugs and warts that should be fixed when used with Pintos. Thus, Bochs should be installed manually for use with Pintos, instead of using the packaged version of Bochs included with an operating system distribution.

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 ‘--with-x --with-x11 --with-term --with-nogui’ to configure.

This version of Pintos is designed for use with Bochs 2.2.6. A number of patches for this version of Bochs are included in ‘src/misc’:

‘bochs-2.2.6-big-endian.patch’

Makes the GDB stubs work on big-endian systems such as Solaris/Sparc, by doing proper byteswapping. It should be harmless elsewhere.

‘bochs-2.2.6-jitter.patch’

Adds the “jitter” feature, in which timer interrupts are delivered at random intervals (see [Section 1.1.4 [Debugging versus Testing], page 4](#_bookmark4)).

‘bochs-2.2.6-triple-fault.patch’

Causes Bochs to break to GDB when a triple fault occurs and the GDB stub is active (see [Section E.6 [Triple Faults], page 108](#_bookmark136)).

‘bochs-2.2.6-ms-extensions.patch’

Needed for Bochs to compile with GCC on some hosts. Probably harmless elsewhere.

‘bochs-2.2.6-solaris-tty.patch’

Needed for Bochs to compile in terminal support on Solaris hosts. Probably harmless elsewhere.

‘bochs-2.2.6-page-fault-segv.patch’

Makes the GDB stub report a SIGSEGV to the debugger when a page-fault exception occurs, instead of “signal 0.” The former can be ignored with handle SIGSEGV nostop but the latter cannot.

‘bochs-2.2.6-paranoia.patch’

Fixes compile error with modern versions of GCC.

‘bochs-2.2.6-solaris-link.patch’

Needed on Solaris hosts. Do not apply it elsewhere.

To apply all the patches, cd into the Bochs directory, then type:

patch -p1 < $PINTOSDIR/src/misc/bochs-2.2.6-big-endian.patch patch -p1 < $PINTOSDIR/src/misc/bochs-2.2.6-jitter.patch

patch -p1 < $PINTOSDIR/src/misc/bochs-2.2.6-triple-fault.patch patch -p1 < $PINTOSDIR/src/misc/bochs-2.2.6-ms-extensions.patch patch -p1 < $PINTOSDIR/src/misc/bochs-2.2.6-solaris-tty.patch patch -p1 < $PINTOSDIR/src/misc/bochs-2.2.6-page-fault-segv.patch patch -p1 < $PINTOSDIR/src/misc/bochs-2.2.6-paranoia.patch

patch -p1 < $PINTOSDIR/src/misc/bochs-2.2.6-solaris-link.patch

You will have to supply the proper $PINTOSDIR, of course. You can use patch’s ‘--dry-run’ option if you want to test whether the patches would apply cleanly before trying to apply them.

Sample commands to build and install Bochs for Pintos are supplied in ‘src/misc/bochs-2.2.6-build.sh’.

# Bibliography

## Hardware References

[IA32-v1]. IA-32 Intel Architecture Software Developer’s Manual Volume 1: Basic Architecture. Basic 80*x*86 architecture and programming environment. Available via developer.intel.com. Section numbers in this document refer to revision 18.

[IA32-v2a]. IA-32 Intel Architecture Software Developer’s Manual Volume 2A: Instruction Set Reference A-M. 80*x*86 instructions whose names begin with A through M. Available via developer.intel.com. Section numbers in this document refer to revision 18.

[IA32-v2b]. IA-32 Intel Architecture Software Developer’s Manual Volume 2B: Instruction Set Reference N-Z. 80*x*86 instructions whose names begin with N through Z. Available via developer.intel.com. Section numbers in this document refer to revision 18.

[IA32-v3a]. IA-32 Intel Architecture Software Developer’s Manual Volume 3A: System Pro- gramming Guide. Operating system support, including segmentation, paging, tasks, inter- rupt and exception handling. Available via developer.intel.com. Section numbers in this document refer to revision 18.

[FreeVGA]. FreeVGA Project. Documents the VGA video hardware used in PCs. [kbd]. Keyboard scancodes. Documents PC keyboard interface.

[ATA-3]. AT Attachment-3 Interface (ATA-3) Working Draft. Draft of an old version of the ATA aka IDE interface for the disks used in most desktop PCs.

[PC16550D]. National Semiconductor PC16550D Universal Asynchronous Re- ceiver/Transmitter with FIFOs. Datasheet for a chip used for PC serial ports.

[8254]. Intel 8254 Programmable Interval Timer. Datasheet for PC timer chip.

[8259A]. Intel 8259A Programmable Interrupt Controller (8259A/8259A-2). Datasheet for PC interrupt controller chip.

## Software References

[ELF1]. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2 Book I: Executable and Linking Format. The ubiquitous format for executables in modern Unix systems.

[ELF2]. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2 Book II: Processor Specific (Intel Architecture). 80*x*86-specific parts of ELF.

[ELF3]. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2 Book III: Operating System Specific (UNIX System V Release 4). Unix-specific parts of ELF.

[SysV-ABI]. System V Application Binary Interface: Edition 4.1. Specifies how applica- tions interface with the OS under Unix.

[SysV-i386]. System V Application Binary Interface: Intel386 Architecture Processor Sup- plement: Fourth Edition. 80*x*86-specific parts of the Unix interface.

[SysV-ABI-update]. System V Application Binary Interface—DRAFT—24 April 2001. A draft of a revised version of [SysV-ABI] which was never completed.

## Operating System Design References

[Dijkstra]. E. W. Dijkstra, *The structure of the “THE” multiprogramming system*. Com- munications of the ACM 11(5):341–346, 1968. <http://doi.acm.org/10.1145/363095.363143>.

[Hoare]. C. A. R. Hoare, *Monitors: An Operating System Structuring Concept*. Commu- nications of the ACM, 17(10):549–557, 1974. <http://www.acm.org/classics/feb96/>.

[Lampson]. B. W. Lampson, D. D. Redell, *Experience with processes and monitors in Mesa*. Communications of the ACM, 23(2):105–117, 1980. <http://doi.acm.org/10.1145/358818.358824>.

[McKusick]. M. K. McKusick, K. Bostic, M. J. Karels, J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, 1996.

[Wilson]. P. R. Wilson, M. S. Johnstone, M. Neely, D. Boles, *Dynamic Storage Allocation: A Survey and Critical Review*. International Workshop on Memory Management, 1995. <http://www.cs.utexas.edu/users/oops/papers.html#allocsrv>.

# License

Pintos, including its documentation, is subject to the following license:

Copyright *§*c 2004, 2005, 2006 Board of Trustees, Leland Stanford Jr. Univer-

sity. All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTIC- ULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CON- TRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CON- NECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

A few individual files in Pintos were originally derived from other projects, but they have been extensively modified for use in Pintos. The original code falls under the original license, and modifications for Pintos are additionally covered by the Pintos license above.

In particular, code derived from Nachos is subject to the following license:

Copyright *§*c reserved.

1992-1996 The Regents of the University of California. All rights

Permission to use, copy, modify, and distribute this software and its documen- tation for any purpose, without fee, and without written agreement is hereby granted, provided that the above copyright notice and the following two para- graphs appear in all copies of this software.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN “AS IS” BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Code derived from MIT’s 6.828 course code is subject to the following license:

Copyright *§*c

1997 Massachusetts Institute of Technology

This software is being provided by the copyright holders under the following license. By obtaining, using and/or copying this software, you agree that you have read, understood, and will comply with the following terms and conditions:

Permission to use, copy, modify, distribute, and sell this software and its doc- umentation for any purpose and without fee or royalty is hereby granted, pro- vided that the full text of this NOTICE appears on ALL copies of the software and documentation or portions thereof, including modifications, that you make.

THIS SOFTWARE IS PROVIDED “AS IS,” AND COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IM- PLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE OR DOCUMENTATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS. COPYRIGHT HOLDERS WILL BEAR NO LIABILITY FOR ANY USE OF THIS SOFTWARE OR DOCUMENTATION.

The name and trademarks of copyright holders may NOT be used in advertising or publicity pertaining to the software without specific, written prior permis- sion. Title to copyright in this software and any associated documentation will at all times remain with copyright holders. See the file AUTHORS which should have accompanied this software for a list of all copyright holders.

This file may be derived from previously copyrighted software. This copyright applies only to those changes made by the copyright holders listed in the AU- THORS file. The rest of this file is covered by the copyright notices, if any, listed below.