Programming Assignment 2 - Pintos

Operating Systems, EDA092 - DIT400

based on Pintos documentation by Ben Pfaff
adapted by Yiannis Nikolakopoulos, Ivan Walulya

Welcome!

During the course, we have discussed and also shared hands on experiences about the different challenges that embrace the design and implementation of modern operating systems. This lab is intended for you to face a challenging dimension at the basis of real operating systems: synchronization. The goal is to see its complexity first-hand and solve synchronization and scheduling related problems that resemble the ones that could appear in practice. In this lab we will use Pintos, an educational operating system supporting kernel threads, loading and running of user programs and a file system. Pintos is among the international well-established platforms for such assignments and is used at various schools, such as Stanford, Virginia Tech and Max Planck Institute, for lab assignments.

This assignment is divided into 2 tasks:

1. The first task will focus on the enhancement of synchronization implementations. One of the classic synchronization methods for a thread is busy-waiting, i.e. spinning in an endless loop until some information from another thread/process stops it. An obvious drawback, especially in uniprocessor machines, is that CPU cycles are wasted without any useful work being done. In this task, you will get deeper in the synchronization implementation and try to provide an alternative implementation of a sleep function.
2. In the second task, you will deal with the synchronization problems that occur when scheduling jobs that send and receive data through a common bus for an external hardware accelerator (e.g. GPU, co-processor).

Short Contents

Table of Contents

1 Introduction

Welcome to Pintos. Pintos is a simple operating system for the 80x86 architecture, built for educational purposes. 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 some of these areas.

For practical reasons, we will run Pintos projects in a system simulator, that is, a program that simulates an 80x86 CPU and its peripheral devices accurately enough that unmodified operating systems and software can run under it. In class we will use the Bochs simulator. Pintos has also been tested with VMware Player and QEMU.

The project is demanding, especially regarding the understanding of the platform, but the knowledge gained is rewarding. EDA092 has been building a reputation of taking a lot of time, and deservedly so.

1.1 READ ME FIRST

At first, don't get intimidated by the size of this document :).

- This chapter explains how to get started working with Pintos, instructions for working in Chalmers STUDAT machines, the source code structure, building, debugging and submission information. You should read the entire chapter before you start work on the assignment.
- However, if you need an overview of what you are going to work on, you can check Chapter 2 [Assignment Description], page 7 and especially section Section 2.1 [Assignment Description Requirements], page 7 for the problems you are called to solve.
- Section 2.2 [Assignment Description Background], page 9 will help you obtain the necessary background for solving the problems in Pintos.
- ⟨undefined⟩ [Reference Guide], page ⟨undefined⟩ serves as a reference for the entire Pintos. Roughly speaking, this appendix replaces the man pages that you would get in a full operating system.

1.2 Getting Started

To get started, you'll have to log into a machine that Pintos can be built on. The EDA092 "officially supported" Pintos development machines are the STUDAT Linux machines. Follow the instructions on how to include the path. 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 A [Installing Pintos], page 21).

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 `bash`, the standard login shell, you can add the following line into your `$HOME/.bashrc` (or create it if it doesn't exist):

```
export PATH=/chalmers/sw/unsup64/phc/b/pkg/bochs-2.6.6/bin:$HOME/
pintos/src/utils:$PATH
```

Remember that files starting with a dot are hidden and may not show up in file managers. Do not forget to reload the configuration using:

```
source $HOME/.bashrc
```

or restart the terminal afterwards to apply the changes.

1.2.1 Source Tree Overview

Now you can fetch the source for Pintos from the Ping Pong page of the course and extract it into your home directory by executing

```
tar xfz pintos-chalmers.tar.gz -C ~/
```

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 in this assignment

'`userprog/`'
: Source code for the user program loader, which you will not need to modify.

'`vm/`'
: An almost empty directory. which you will not need to modify.

'`filesys/`'
: Source code for a basic file system. You will not need to modify this in this assignment.

'`devices/`'
: Source code for I/O device interfacing: keyboard, timer, disk, batch-scheduler etc. You will make modifications in this directory for both tasks in this lab assignment. File `timer.c` for the Task 1 and `batch-scheduler.c` for Task 2.

'`lib/`'
: An implementation of a subset of the standard C library. The code in this directory is compiled into both the Pintos kernel and 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 no 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 general purpose use. You should not need to use this in this assignment.

'`misc/`'
'`utils/`'
: These files may come in handy if you decide to try working with Pintos on your own machine. Otherwise, you can ignore them.

1.2.2 Building Pintos

As the next step, build the source code supplied for the first project. First, `cd` into the '`threads`' directory. Then, issue the '`make`' command. This will create a '`build`' directory under '`threads`', populate it with a '`Makefile`' and a few subdirectories, and then build the kernel inside. The entire build should take less than 30 seconds.

Following the build, the following are the interesting files in the '`build`' directory:

'`Makefile`'
: A copy of '`pintos/src/Makefile.build`'. It describes how to build the kernel. See [Adding Source Files], page 15, for more information.

'`kernel.o`'
: Object file for the entire kernel. This is the result of linking object files compiled from each individual kernel source file into a single object file. It contains debug information, so you can run GDB (see ⟨undefined⟩ [GDB], page ⟨undefined⟩) or `backtrace` (see ⟨undefined⟩ [Backtraces], page ⟨undefined⟩) on it.

'`kernel.bin`'
: Memory image of the kernel, that is, the exact bytes loaded into memory to run the Pintos kernel. This is just '`kernel.o`' with debug information stripped out, which saves a lot of space, which in turn keeps the kernel from bumping up against a 512 kB size limit imposed by the kernel loader's design.

'`loader.bin`'
: Memory image for the kernel loader, a small chunk of code written in assembly language that reads the kernel from disk into memory and starts it up. It is exactly 512 bytes long, a size fixed by the PC BIOS.

Subdirectories of '`build`' contain object files ('`.o`') and dependency files ('`.d`'), both produced by the compiler. The dependency files tell `make` which source files need to be recompiled when other source or header files are changed.

1.2.3 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 Pintos kernel. In these arguments, `run` instructs the kernel to run a test and `alarm-multiple` is the test to run.

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, 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. You can run the simulator with a debugger (see ⟨undefined⟩ [GDB], page ⟨undefined⟩).

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`.

1.2.4 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 synchronization, 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. No number of runs can guarantee that your synchronisation is perfect, but the more you do, the more confident you can be that your code doesnt have major flaws.

1.3 Legal and Ethical Issues

Pintos is distributed under a liberal license that allows free use, modification, and distribution. 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 26, for details of the license and lack of warranty.

In the context of Chalmers EDA092 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.

1.4 Acknowledgements

The Pintos core and this documentation were originally written by Ben Pfaff blp@cs.stanford.edu.

Additional features were contributed by Anthony Romano chz@vt.edu.

The GDB macros supplied with Pintos were written by Godmar Back 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 ([Christopher]).

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 ⟨undefined⟩ [Monitors], page ⟨undefined⟩) is from classroom slides originally by Dawson Engler and updated by Mendel Rosenblum.

The current version has been edited and adapted to the requirements of the EDA092/DIT400 Operating Systems course of Chalmers University of Technology by Yiannis Nikolakopoulos and Ivan Walulya, in collaboration with Bhavisya Goel, Vincenzo Gulisano and Marina Papatriantafilou.

1.5 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 80x86 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.

2 Assignment Description

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`' and the '`devices`' directories for this assignment. Compilation should be done in the '`threads`' directory. Before you start working on this project, and for its description to make sense, you should read the following sections: Chapter 1 [Introduction], page 2 and ⟨undefined⟩ [Debugging Tools], page ⟨undefined⟩. You should at least skim the material from ⟨undefined⟩ [Pintos Loading], page ⟨undefined⟩ through ⟨undefined⟩ [Memory Allocation], page ⟨undefined⟩, especially ⟨undefined⟩ [Synchronization], page ⟨undefined⟩.

2.1 Requirements

2.1.1 Task 1: Alarm Clock

One of the classic synchronization methods is busy-waiting, i.e. spinning in an endless loop until some information from another thread/process stops you. An obvious drawback, especially in uniprocessor machines, is that CPU cycles are wasted without any useful work being done. In this task, you will get deeper in the synchronization implementation and try to provide an alternative implementation of a sleep function.

Reimplement `timer_sleep()`, defined in '`devices/timer.c`'. Although a working implementation 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 1.2.4 [Debugging versus Testing], page 5).

Hint For a thread not to 'busy wait' after calling the `timer_sleep()` function, the thread has to block, thus changing its state from running to blocked. Please be

reminded that the processor does not store any information about the state of each thread, a blocked thread should store the information about how long it is blocked.

For each clock tick, a timer interrupt is triggered and the `timer_interrupt()` interrupt handler is executed. We exploit this interrupt handler to activate sleeping threads and also update thread statistics. The `thread_foreach()` function should be used to iterate over all blocked threads. Check if a blocked thread is ready to wakeup and call `thread_unblock()` to activate the thread or update its sleep timer.

2.1.2 Task 2: Batch scheduling

In this task you are called to solve a simple batch scheduling problem, and more specifically to handle the synchronization issues that arise when scheduling different batches of jobs. The assumption is that our system is extended with an external processing accelerator (e.g. a GPU or a co-processor) with X Processing Units (PUs). Tasks `task_t` are handled by one thread each, and contain the appropriate data/results from/to the accelerator. However, the communication bus with the accelerator is half duplex (i.e. one direction can be used at a time) and has limited bandwidth as only 3 slots can be used by tasks at a time.

```
typedef struct {
    int direction;
    int priority;
} task_t


OneTask(task_t task) {
  getSlot(task);
  transferData(task);
  leaveSlot(task);
}
```

In the code above, direction is either 0 or 1; it gives the direction in which the task's data are copied (from/to the accelerator respectively). The parameter priority indicates if this is a high priority task (when it is set to the value 1), in which case it should have priority over other tasks. When such a task needs to send data, it should be allowed access as soon as possible.

The main part of this assignment is to:

- Implement the procedures `getSlot` and `leaveSlot` using only basic synchronization primitives: semaphores, locks and condition variables.
- You must also implement the `transferData` procedure, but this should just print out a debug message upon entrance, sleep the thread for a random amount of time, and print another debug message upon exit.
- `getSlot` must not return (i.e., it blocks the thread) until it is safe for the thread to send the data through the bus in the given direction.
- `leaveSlot` is called to indicate that the caller has finished sending data; `leaveSlot` should take steps to let additional tasks send data (i.e., unblock them).

This is a lightly used accelerator, so you do not need to guarantee fairness or freedom from starvation, other than what has been indicated for high priority tasks.

Furthermore you have to:

- implement the function `batchScheduler` that takes four parameters which represent the number of tasks of each type and direction that use the bus (these are parameters to the functions invoked by the tests we provide). For each task, the main thread must spawn a new thread that executes the `OneTask` procedure, which you must implement.

Your solution will be graded through code inspection to verify the correctness of the synchronization algorithms (in addition to basic functionality checks of the running code). To accomplish this task, implement the function prototypes provided in '`devices/batch-scheduler.c`' enforcing the required constraints.

2.1.3 Design Document

Before you turn in your assignment, you must copy the assignment design document template ('`project.tmpl`') into your source tree under the name '`pintos/DESIGNDOC`' and fill it in. We recommend that you read the design document template before you start working on the assignment.

2.2 Background

2.2.1 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 2), 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, therefore, 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 ⟨undefined⟩ [GDB], page ⟨undefined⟩). 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 ⟨undefined⟩ [Thread Switching], page ⟨undefined⟩, 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 ⟨undefined⟩ [Memory Allocation], page ⟨undefined⟩).

2.2.2 Source Files

Despite Pintos being a tiny operating system, the code volume can be quite discouraging at first sight. Dont panic: the Alarm Clock exercise for Task 1 will help you understand Pintos by working on a small fragment of the code. 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.

2.2.2.1 'threads' code

Here is a brief overview of the files in the '`threads`' directory.

'`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 finds the kernel on disk, loads it into memory, and jumps to `start()` in '`start.S`'. See ⟨undefined⟩ [Pintos Loader], page ⟨undefined⟩, for details. You should not need to look at this code or modify it.

'`start.S`'
: Does basic setup needed for memory protection and 32-bit operation on 80x86 CPUs. Unlike the loader, this code is actually part of the kernel. See ⟨undefined⟩ [Low-Level Kernel Initialization], page ⟨undefined⟩, for details.

'`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 near the beginning of the kernel image. See ⟨undefined⟩ [Pintos Loader], page ⟨undefined⟩, for details. Again, you should not need to look at this code or modify it, but it's here in case you're curious.

[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()`.

‘`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 ⟨undefined⟩ [High-Level Kernel Initialization], page ⟨undefined⟩, 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 ⟨undefined⟩ [struct thread], page ⟨undefined⟩ and ⟨undefined⟩ [Threads], page ⟨undefined⟩ for more information.

‘`switch.S`’
‘`switch.h`’
Assembly language routine for switching threads. Already discussed above. See ⟨undefined⟩ [Thread Functions], page ⟨undefined⟩, for more information.

‘`palloc.c`’
‘`palloc.h`’
Page allocator, which hands out system memory in multiples of 4 kB pages. See ⟨undefined⟩ [Page Allocator], page ⟨undefined⟩, for more information.

‘`malloc.c`’
‘`malloc.h`’
A simple implementation of `malloc()` and `free()` for the kernel. See ⟨undefined⟩ [Block Allocator], page ⟨undefined⟩, for more information.

‘`interrupt.c`’
‘`interrupt.h`’
Basic interrupt handling and functions for turning interrupts on and off. See ⟨undefined⟩ [Interrupt Handling], page ⟨undefined⟩, for more information.

‘`intr-stubs.S`’
‘`intr-stubs.h`’
Assembly code for low-level interrupt handling. See ⟨undefined⟩ [Interrupt Infrastructure], page ⟨undefined⟩, 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 Task 2 of the Lab Assignment. See ⟨undefined⟩ [Synchronization], page ⟨undefined⟩, 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.

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

2.2.2.2 '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.

'batch-scheduler.c'
Contains code skeleton to be used for implementing Task 2 of this Lab Assignment.

'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).

'block.c'
'block.h' An abstraction layer for *block devices*, that is, random-access, disk-like devices that are organized as arrays of fixed-size blocks. Out of the box, Pintos supports two types of block devices: IDE disks and partitions.

'ide.c'
'ide.h' Supports reading and writing sectors on up to 4 IDE disks.

'partition.c'
'partition.h'
Understands the structure of partitions on disks, allowing a single disk to be carved up into multiple regions (partitions) for independent use.

'kbd.c'
'kbd.h' Keyboard driver. Handles keystrokes passing them to the input layer (see below).

'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.

'rtc.c'
'rtc.h' Real-time clock driver, to enable the kernel to determine the current date and time. By default, this is only used by 'thread/init.c' to choose an initial seed for the random number generator.

‘`speaker.c`’
‘`speaker.h`’
: Driver that can produce tones on the PC speaker.

‘`pit.c`’
‘`pit.h`’
: Code to configure the 8254 Programmable Interrupt Timer. This code is used by both ‘`devices/timer.c`’ and ‘`devices/speaker.c`’ because each device uses one of the PIT’s output channel.

2.2.2.3 ‘`lib`’ files

Finally, ‘`lib`’ and ‘`lib/kernel`’ contain useful library routines. (‘`lib/user`’ can be used by user programs but it is not part of the kernel, thus not useful for you in this project.) 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 ⟨undefined⟩ [C99], page ⟨undefined⟩, for information on a few recently introduced pieces of the C library that you might not have encountered before. See ⟨undefined⟩ [Unsafe String Functions], page ⟨undefined⟩, for information on what’s been intentionally left out for safety.

‘`debug.c`’
‘`debug.h`’
: Functions and macros to aid debugging. See ⟨undefined⟩ [Debugging Tools], page ⟨undefined⟩, for more information.

‘`random.c`’
‘`random.h`’
: Pseudo-random number generator. The actual sequence of random values will not vary from one Pintos run to another, unless you do one of three things: specify a new random seed value on the ‘`-rs`’ kernel command-line option on each run, or use a simulator other than Bochs, or specify the ‘`-r`’ option to `pintos`.

‘`round.h`’
: Macros for rounding.

‘`syscall-nr.h`’
: System call numbers.

‘`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.

‘`kernel/console.c`’
‘`kernel/console.h`’
‘`kernel/stdio.h`’
: Implements `printf()` and a few other functions.

2.2.3 Synchronization

Proper synchronization is an important part of the solutions to these problems. Any synchronization 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 ⟨undefined⟩ [Synchronization], page ⟨undefined⟩) 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 interrupt 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.

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 assignment. (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.

2.2.4 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 SVN or GIT. 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 Lab assignment. 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 ⟨undefined⟩ [Debugging Tools], page ⟨undefined⟩). Be sure to read the section on backtraces (see ⟨undefined⟩ [Backtraces], page ⟨undefined⟩), which will help you to get the most out of every kernel panic or assertion failure.

2.3 FAQ

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 3.1 [Testing], page 18.

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 actually 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 ⟨undefined⟩ [Function and Parameter Attributes], page ⟨undefined⟩), so it doesn't include any code in `fail()` 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 ⟨undefined⟩ [Backtraces], page ⟨undefined⟩, 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 return 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 instruction.
  - `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.

2.3.1 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 EDA092 curriculum.

3 How to test - What to submit

We will grade your assignments based on both test results and design quality. The grade will be Pass or Fail.

3.1 Testing

Each task 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.

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 1.2.4 [Debugging versus Testing], page 5).

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

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

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

3.2 Submission

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. Submit a `tar.gz` of your pintos directory in including all your pintos source code and your design document in the related Ping Pong page.

3.2.1 Design Document

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

Data Structures

The instructions for this section are always the same:

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

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

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

Algorithms

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

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

Synchronization

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

Rationale

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

An incomplete, evasive, or non-responsive design document or one that strays from the template without good reason may be penalized. Incorrect capitalization, punctuation, spelling, or grammar can also cost points.

3.2.2 Source Code

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

between the description and the actual code will be penalized, as will be any bugs we find by spot checks.

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.

Appendix A Installing Pintos

This chapter explains how to install a Pintos development environment on your own machine. 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. 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. 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. Version 5.8.0 or later is preferred. Version 5.6.1 or later should work.
- Required: GNU make, version 3.80 or later.
- Recommended: QEMU, version 0.11.0 or later. If QEMU is not available, Bochs can be used, but its slowness is frustrating.
- Recommended: GDB. GDB is helpful in debugging (see ⟨undefined⟩ [GDB], page ⟨undefined⟩). 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. 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, version 4.5 or later. Texinfo is required to build the PDF version of the documentation.
- Optional: TEX. Also required to build the PDF version of the documentation.
- Optional: VMware Player. 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, version 2.2.6, as described below (see Section A.1 [Building Bochs for Pintos], page 22).
2. Install scripts from '`src/utils`'. Copy '`backtrace`', '`pintos`', '`pintos-gdb`', '`pintos-mkdisk`', '`pintos-set-cmdline`', and '`Pintos.pm`' into the default `PATH`.
3. 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.
4. 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.

5. 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.
6. Optional: Build the documentation, by running `make dist` in the top-level '`doc`' directory. This creates a '`WWW`' subdirectory within '`doc`' that contains HTML and PDF versions of the documentation, plus the design document templates and various hardware 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.

A.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 1.2.4 [Debugging versus Testing], page 5).

'`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 ⟨undefined⟩ [Triple Faults], page ⟨undefined⟩).

'`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 80x86 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. 80x86 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. 80x86 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 Programming Guide. Operating system support, including segmentation, paging, tasks, interrupt 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 Receiver/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.

[MC146818A]. Motorola MC146818A Real Time Clock Plus Ram (RTC). Datasheet for PC real-time clock 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). 80x86-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 applications interface with the OS under Unix.

[SysV-i386]. System V Application Binary Interface: Intel386 Architecture Processor Supplement: Fourth Edition. 80x86-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.

[SUSv3]. The Open Group, Single UNIX Specification V3, 2001.

[Partitions]. A. E. Brouwer, Minimal partition table specification, 1999.

[IntrList]. R. Brown, Ralf Brown's Interrupt List, 2000.

Operating System Design References

[Christopher]. W. A. Christopher, S. J. Procter, T. E. Anderson, *The Nachos instructional operating system*. Proceedings of the USENIX Winter 1993 Conference. http://portal.acm.org/citation.cfm?id=1267307.

[Dijkstra]. E. W. Dijkstra, *The structure of the "THE" multiprogramming system*. Communications 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*. Communications 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 © 2004, 2005, 2006 Board of Trustees, Leland Stanford Jr. University. 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 PARTICULAR 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 CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION 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 © 1992-1996 The Regents of the University of California. All rights reserved.
>
> Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without written agreement is hereby granted, provided that the above copyright notice and the following two paragraphs 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.