

## Lab 2: Alarm Clock in Pintos

Operating Systems Course  
Chalmers and Gothenburg University

August 17, 2020

# 1 Introduction to Pintos

In this lab (as well as lab3), your job is to extend the functionality of Pintos. Pintos is a simple operating system for the  $80 \times 86$  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 will strengthen the support of Pintos in some of these areas. For practical reasons, we will run Pintos projects in the Bochs system simulator, that is, a program that simulates an  $80 \times 86$  CPU and its peripheral devices accurately enough that unmodified operating systems and software can run under it.

This chapter explains how to get started working with Pintos, instructions for working in Chalmers STUDAT machines, the source code structure, building, debugging, submission information, and lab2 description. You should read the entire chapter before you start work on the assignment.

## 1.1 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 (i.e. `remote11.chalmers.se`, `remote12.chalmers.se`). 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.

Once you've logged into one of these machines, either locally or remotely, fetch the source for Pintos from the Canvas page of the course and extract it into your home directory. Then, follow the instructions on how to add the 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.

Moreover, make sure the binaries in `pintos/src/utils/` are executable:

```
chmod +x pintos/src/utils/pintos*
```

```
chmod +x pintos/src/utils/backtrace
```

## 1.2 Source Tree Overview

Let's take a look at what's inside Pintos. 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 lab.

`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.

`filesystem/`

Source code for a basic file system, which you will not need to modify.

`devices/`

Source code for I/O device interfacing: keyboard, timer, disk, batch-scheduler etc. You will make modifications in this directory for this lab and also lab3. File `timer.c` for the lab2 and `batch-scheduler.c` for lab3.

`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.

## 1.3 Building Pintos

As the next step, build the source code. 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.

#### **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 or **backtrace** 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 512kB 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.

## 1.4 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...*, where 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.

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`.

## 1.5 Debugging versus Testing

When you're debugging the 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 doesn't have major flaws.

## 1.6 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.

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 report.

## 1.7 Acknowledgements

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

Additional features were contributed by Anthony Romano [chz@vt.edu](mailto:chz@vt.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 [? ].

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.

The current version has been edited and adapted to the requirements of the EDA092/DIT400 Operating Systems course of Chalmers University of Technology by Hannaneh Najdataei and Dimitris Palyvos in collaboration with Vincenzo Gulisano and Marina Papatriantafilou.

## 1.8 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  $80 \times 86$  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.

## 2 Assignment Description

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 lab, you will get deeper in the synchronization implementation and try to provide an alternative implementation of a sleep function.

Your assignment in this lab is to re-implement `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. Re-implement this function to avoid busy waiting.

**`void timer_sleep(int64 t ticks)`** 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 1.5).

**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, the blocked thread itself should store the information about how long it is blocked.

For each clock tick, a timer interrupt is triggered and `timer_interrupt()`, the 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.

### 3 How to test - What to submit

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

#### 3.1 Testing

This lab 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 `PINTSOPTS='@dots'`, e.g.:

```
make check PINTSOPTS='-j 1'
```

to select a jitter value of 1.

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 report and the source code that you submit. To pass the lab, you need to implement all the requested specifications and verify your code with the self-test examples found below. You also need to write a report where you describe the design and behavior of your solution. Finally, you need to upload both the report and your code to Canvas. The following instructions describe the submission process in detail:

1. **Writing the report** For your report, begin by describing the implementation of your solution. More specifically, briefly analyze how you implemented each of the following:
  - **Data Structures** Highlight for us the actual changes to data structures. Also add 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. 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 report 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 description 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 report to explain how your code works to implement the requirements.
  - **Synchronization** An operating system kernel is a complex, multi-threaded program, in which synchronizing multiple threads can be difficult. That is why we want you to explain explicitly 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 would like 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).
2. **Preparing the 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 report 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.
- After you have verified that your code works correctly on remote11/12, run the `prepare-submission` script found in the lab folder. The script will check that your code compiles correctly and it will create an archive with only the necessary files for grading.
3. **Final submission** For the final submission, prepare an archive containing the archive of your code (prepared as per the instructions above) and the report file and upload it to canvas.

## 4 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.

## Bibliography

- [1] W. A. Christopher, S. J. Procter, and T. E. Anderson. The nachos instructional operating system. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, page 4, USA, 1993. USENIX Association.