



南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

CS 302
OPERATING SYSTEM
SPRING 2021

Project 1: Threads

Design Review Due:	March 21, 2021
Code Due:	April 05, 2021
Final Report Due:	April 05, 2021

DBGGroup@SUSTech
January 20, 2021

Contents

1	Introduction	3
1.1	Pintos introduction	3
1.2	Pintos Getting Started	3
2	Your task	6
2.1	Introduction for Project 1: Threads	6
2.2	Task 1: Efficient Alarm Clock	7
2.3	Task 2: Priority Scheduler	7
2.4	Task 3: Multi-level Feedback Queue Scheduler (MLFQS)	8
2.5	Task 4: Test Pintos with GDB	8
3	Grading Policy	8
3.1	Design Document and Design Review (Due: 3/21)	8
3.1.1	Design Document Guidelines	9
3.1.2	Design Document Additional Questions	11
3.1.3	Design Review	11
3.2	Code (Due 4/05)	11
3.3	Debugging with GDB (Due 4/05)	12
3.4	Final Report (Due 4/05) and Code Quality	12
4	Submission	12
4.1	What to submit	12
4.2	Where to submit and ddl	13
4.3	Remarks	13
5	Reference	13
5.1	Pintos	13
5.1.1	Source Tree	14
5.1.2	Building Pintos	15
5.1.3	Running Pintos	15
5.1.4	Debugging Pintos	16
5.1.5	Debugging Pintos Tests	17
5.1.6	Debugging Page Faults	18
5.1.7	Debugging Kernel Panics	18
5.2	Threads	19
5.2.1	Understanding Threads	19
5.2.2	The Thread Struct	20
5.2.3	Thread Functions	22
5.2.4	Thread Switching	24
5.3	Synchronization	25
5.3.1	Disabling Interrupts	25
5.3.2	Semaphores	26
5.3.3	Locks	27

5.3.4	Monitor	28
5.3.5	Optimization Barriers	29
5.4	Memory Allocation	31
5.4.1	Page Allocator	31
5.4.2	Block Allocator	32
5.5	Linked Lists	33
5.6	Efficient Alarm Clock	35
5.7	Priority Scheduler	35
5.8	Advanced Scheduler	36
5.8.1	Introduction	36
5.8.2	Fixed-point Real Numbers	37
5.8.3	Niceness	37
5.8.4	Calculating Priority	37
5.8.5	Calculating Recent CPU	38
5.8.6	Calculating Load Average	39
5.8.7	Summary	39
5.8.8	Additional Details	40
5.9	Debugging Tips	40
5.9.1	printf	40
5.9.2	ASSERT	41
5.9.3	Function and parameter attributes	41
5.9.4	Backtraces	41
5.9.5	GDB	44
5.9.6	Triple Faults	49
5.9.7	General Tips	50
5.10	FAQ	50
5.10.1	Alarm Clock FAQ	51
5.10.2	Priority Scheduling FAQ	51
5.10.3	Advanced Scheduler FAQ	52
6	Acknowledgment	53

1 Introduction

1.1 Pintos introduction

Welcome to Pintos. Pintos is a simple operating system framework for the 80x86 architecture. 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.

Pintos could, theoretically, run on a regular IBM-compatible PC. Unfortunately, it is impractical to supply every CS 302 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 80x86 CPU and its peripheral devices accurately enough that unmodified operating systems and software can run under it. Simulators also give you the ability to inspect and debug an operating system while it runs.

In class we will use the Bochs simulators. Pintos has also been tested with VMware Player. 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.

1.2 Pintos Getting Started

We have prepared `pintos.tar.gz` and `bochs-2.6.7.tar.gz` for you. What you need to do are:

1. download and install **VMware work station** on your PC.
2. download **ubuntu 14.04 /16.04** and create a virtual machine on **VMware**. (if you want to install a latest version of ubuntu, please make sure that pintos and bochs can work properly)

Here is an instruction about how to install **VMware** and create virtual machine:

[Instruction for Installing VMware](#)

We recommend you to install **VMware Tools** on your virtual machine for easy use. You can take this as a reference:

[Instruction for Installing VMware Tools](#)

After the above operation, you Ubuntu virtual machine are expected to successfully running on your PC.

3. press **Ctrl+Alt+T** to call a terminal, you can use `$ ls`, you have learned in the lab, to see what is in `/home` directory.
4. create a directory named `os` or something you like to put `pintos.tar.gz` and `bochs-2.6.7.tar.gz`.

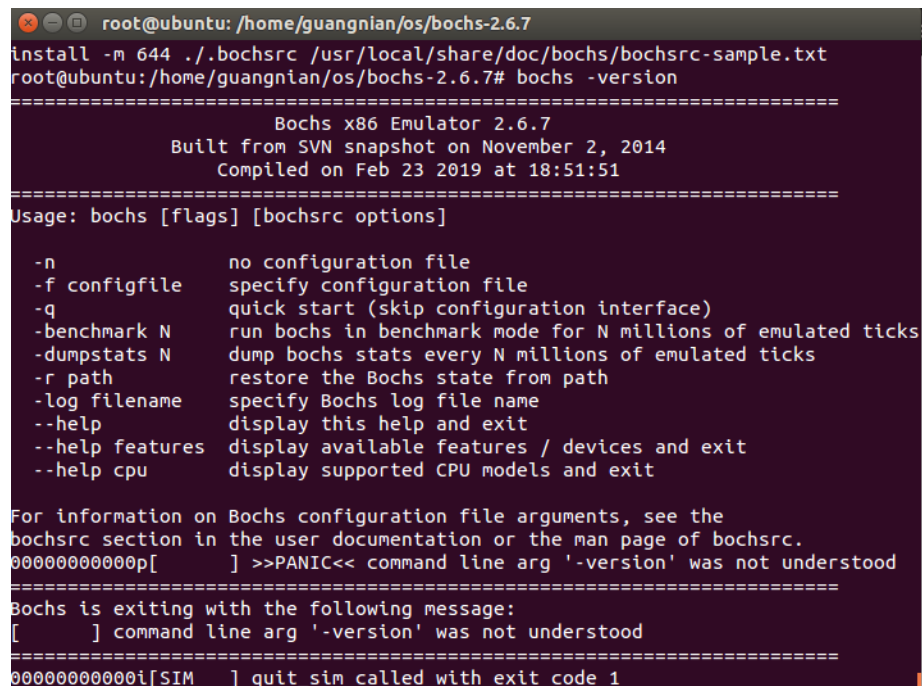
5. step in to file `os` and execute the following instructions to install Prerequisite software:

```
$ sudo apt-get install build-essential
$ sudo apt-get install xorg-dev
$ sudo apt-get install bison
$ sudo apt-get install libgtk2.0-dev
```

6. configure bochs on your Ubuntu using the following instructions:

```
$ tar -zxvf bochs-2.6.7.tar.gz
$ cd bochs-2.6.7
$ ./configure --with-nogui --enable-gdb-stub
$ make
$ sudo make install
$ bochs -version
```

then you should see something like this:



```
root@ubuntu: /home/guangnian/os/bochs-2.6.7
install -m 644 ./bochsrc /usr/local/share/doc/bochs/bochsrc-sample.txt
root@ubuntu:/home/guangnian/os/bochs-2.6.7# bochs -version
=====
Bochs x86 Emulator 2.6.7
Built from SVN snapshot on November 2, 2014
Compiled on Feb 23 2019 at 18:51:51
=====
Usage: bochs [flags] [bochsrc options]

-n          no configuration file
-f configfile  specify configuration file
-q          quick start (skip configuration interface)
-benchmark N  run bochs in benchmark mode for N millions of emulated ticks
-dumpstats N  dump bochs stats every N millions of emulated ticks
-r path       restore the Bochs state from path
-log filename specify Bochs log file name
--help       display this help and exit
--help features  display available features / devices and exit
--help cpu    display supported CPU models and exit

For information on Bochs configuration file arguments, see the
bochsrc section in the user documentation or the man page of bochsrc.
00000000000p[      ] >>PANIC<< command line arg '-version' was not understood
=====
Bochs is exiting with the following message:
[      ] command line arg '-version' was not understood
=====
00000000000i[SIM  ] quit sim called with exit code 1
```

Figure 1: Bochs successfully installed

7. configure pintos on your Ubuntu:

```
$ cd ..
$ tar -zxvf pintos.tar.gz
$ cd pintos/src/threads
```

```
$ make
$ cd build
$ ../../utils/pintos run alarm-multiple
```

then you should see information like this:

```
root@ubuntu:/home/guangnian/os/pintos/src/threads/build# ../../utils/pintos run
alarm-multiple
Prototype mismatch: sub main::SIGVTALRM () vs none at ../../utils/pintos line 9
35.
Constant subroutine SIGVTALRM redefined at ../../utils/pintos line 927.
warning: can't find squish-pty, so terminal input will fail
bochs -q
=====
                Bochs x86 Emulator 2.6.7
                Built from SVN snapshot on November 2, 2014
                Compiled on Feb 23 2019 at 18:51:51
=====
00000000000i[      ] reading configuration from bochsrc.txt
00000000000e[      ] bochsrc.txt:8: 'user_shortcut' will be replaced by new 'ke
yboard' option.
00000000000i[      ] installing nogui module as the Bochs GUI
00000000000i[      ] using log file bochsout.txt
PiLo hda1
Loading.....
Kernel command line: run alarm-multiple
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer... 204,600 loops/s.
Boot complete.
Executing 'alarm-multiple':
(alarm-multiple) begin
(alarm-multiple) Creating 5 threads to sleep 7 times each.
```

Figure 2: Pintos successfully installed

8. After running above, we should check whether the program “pintos” is added to the environment. You can enter following command to check:

```
$ pintos
```

If the command line says following, please add pintos to your environment as step 9.

```
~$ pintos
No command 'pintos' found, did you mean:
  Command 'pintod' from package 'pinto' (universe)
  Command 'pinto' from package 'pinto' (universe)
pintos: command not found
```

Figure 3: Pintos not in environment

If the command line says following, pintos is already in your environment.

```

~$ pintos
Use of literal control characters in variable names is deprecated at /home/hadoop/OS
software/pintos/src/utils/pintos line 911.
Prototype mismatch: sub main::SIGVTALRM () vs none at /home/hadoop/OS/software/pinto
src/utils/pintos line 935.
Constant subroutine SIGVTALRM redefined at /home/hadoop/OS/software/pintos/src/utils
pintos line 927.
pintos, a utility for running Pintos in a simulator
Usage: pintos [OPTION...] -- [ARGUMENT...]
where each OPTION is one of the following options
and each ARGUMENT is passed to Pintos kernel verbatim.
Simulator selection:
--bochs          (default) Use Bochs as simulator
--qemu           Use QEMU as simulator
--player        Use VMware Player as simulator
Debugger selection:
--no-debug       (default) No debugger
--monitor        Debug with simulator's monitor
--gdb            Debug with gdb
Display options: (default is both VGA and serial)
-v, --no-vga     No VGA display or keyboard
-s, --no-serial  No serial input or output
-t, --terminal   Display VGA in terminal (Bochs only)
Timing options: (Bochs only)

```

Figure 4: Pintos in environment

9. In order to add pintos to your environment, enter following:

```

$ vi $HOME/.profile
Add your pintos directory to .profile
PATH=".../pintos/src/utils:$PATH"
Save and quit edition
$ source $HOME/.profile
$ pintos

```

10. We have one more step to enable the running of pintos. Since GCC won't automatically link the math library, which is the dependency for pintos running environments, we have to provide pintos with this requirement. Please following the steps below and then you can successfully run tests under `../pintos/src/threads`:

```

$ cd pintos/src/utils
$ vi Makefile
Change ``LDFLAGS = -lm`` to ``LDLIBS = -lm`` and save quit
$ make

```

Until now, Pintos and Bochs are successfully installed.

2 Your task

2.1 Introduction for 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 reference listed in this guide and try to understand the function of Pintos code. To complete this project you will also need to read [5.8 Advanced Scheduler](#). For a better understanding to this project, we also include some common problems encountered by students and a hint of the possible alternations for this project (see [5.10 FAQ](#)). Hope you guys have nice a trip when tackling hardships in project 1.

2.2 Task 1: Efficient Alarm Clock

In Pintos, threads may call this function to put themselves to sleep:

```
/**
 *
 *
 *
 *
 *
 *
 *
 */
void timer_sleep (int64_t ticks);
```

`timer_sleep()` is useful for threads that operate in real-time (e.g. for blinking the cursor once per second). The current implementation of `timer_sleep()` is inefficient, because it calls `thread_yield()` in a loop until enough time has passed. Your task is to re-implement `timer_sleep()` so that it executes efficiently without any "busy waiting".

2.3 Task 2: Priority Scheduler

In Pintos, each thread has a priority value ranging from 0 (PRI MIN) to 63 (PRI MAX). However, the current scheduler does not respect these priority values. You must modify the scheduler so that higher-priority threads always run before lower-priority threads.

You must also modify the 3 Pintos synchronization primitives (lock, semaphore, condition variable), so that these shared resources prefer higher-priority threads over lower-priority threads.

Additionally, you must implement **priority donation** for Pintos locks. When a high-priority thread (A) has to wait to acquire a lock, which is already held by a lower-priority thread (B), we temporarily raise B's priority to A's priority. The purpose of priority donation is to solve the priority inversion problem. For example, a scheduler that did not donate priorities would choose medium-priority threads instead of B. However, we want B to run first, so we can unblock A. Your implementation of priority donation must handle 1) donations

from multiple sources, 2) undoing donations when a lock is released, and 3) nested/recursive donation.

A thread may set its own priority by calling `thread_set_priority(int new_priority)` and get its own priority by calling `thread_get_priority()`.

If a thread no longer has the highest “effective priority” (it called `thread_set_priority()` with a low value or it released a lock), it must immediately yield the CPU to the highest-priority thread.

2.4 Task 3: Multi-level Feedback Queue Scheduler (MLFQS)

In addition to the priority scheduler algorithm, you must implement a multi-level feedback queue scheduler algorithm, which is explained in detail in the reference material. The scheduler will either use the priority scheduling algorithm or the MLFQS algorithm, depending on the value of the variable “`bool thread_mlfqs`” (found in `thread.c`). This variable is toggled with `--mlfqs` command-line option to Pintos.

2.5 Task 4: Test Pintos with GDB

GDB is good tool when testing and debugging designed software in UNIX/ LINUX systems. Employing GDB to diagnose the shortness of new designed functions is a worthy try. For this task, we will release a standard version of Pintos with some code components altered. There are one or several bugs within this release version, resulting the improper running of Pintos. You are required to identify the locations and causes with GDB rendering help. Several basic operations and cases are included in the reference materials (see [5.9.5 GDB](#)). The detailed requirement of submission is in [3.3 Debugging with GDB](#)

3 Grading Policy

Your project grade will be made up of 4 components, with 100% in total:

- ▶ **20%** — Design Document and Design Review
- ▶ **50%** — Code
- ▶ **10%** — GDB Test Report
- ▶ **20%** — Final Report and Code Quality

3.1 Design Document and Design Review (Due: 3/21)

Before you start writing any code for your project, you should create an implementation plan for each feature and convince yourself that your design is correct. For this project, you **must submit a design document** and **can voluntarily attend a design review** with your project TA.

3.1.1 Design Document Guidelines

We suggest you write your design document in `.md` to make the design document easy to follow. For first three tasks of this project, you must explain the following 4 aspects of your proposed design.

1. **Data structures and functions** – Write down any struct definitions, global (or static) variables, typedefs, or enumerations that you will be adding or modifying (if it already exists). These definitions should be written with the **C programming language**, not with pseudocode. Include a brief explanation the purpose of each modification. Your explanations should be as concise as possible. Leave the full explanation to the following sections.
2. **Algorithms** – This is where you tell us how your code will work. Your description should be at a level below the high level description of requirements given in the assignment. We have read the project spec too, so it is unnecessary to repeat or rephrase what is stated here. On the other hand, your description should be at a level above the code itself. Don't give a line-by-line run-down of what code you plan to write. Instead, you should try to convince us that your design satisfies all the requirements, **including any uncommon edge cases**.

The length of this section depends on the complexity of the task and the complexity of your design. Simple explanations are preferred, but if your explanation is vague or does not provide enough details, you will be penalized. Here are some tips:

- For complex tasks, like the priority scheduler, we recommend that you split the task into parts. Describe your algorithm for each part in a separate section. Start with the simplest component and build up your design, one piece at a time. For example, your algorithms section for the Priority Scheduler could have sections for:
 - Choosing the next thread to run
 - Acquiring a Lock
 - Releasing a Lock
 - Computing the effective priority
 - Priority scheduling for semaphores and locks
 - Priority scheduling for condition variables
 - Changing thread's priority
- Use 'backticks' around variable names and function names. Use **bold**, *italics*, and other Markdown styles to improve the readability of your design document.
- Lists can make your explanation more readable. If your paragraphs seem to lack coherency, consider using a list.
- A good length for this section could be 1 paragraph for a simple task (Alarm Clock) or 2 screen pages for a complex task (Priority Scheduler). Make sure your explanation covers all of the required features.

- We fully expect you to read a lot of Pintos code to prepare for the design document. You won't be able to write a good description of your algorithms if you don't know any specifics about Pintos.
3. **Synchronization** – Describe your strategy for preventing race conditions and convince us that it works in all cases. Here are some tips for writing this section:
- This section should be structured as a **list of all potential concurrent accesses to shared resources**. For each case, you should prove that your synchronization design ensures correct behavior.
 - An operating system kernel is a complex, multi-threaded program, in which synchronizing multiple threads can be difficult. The best synchronization strategies are simple and easily verifiable, which leaves little room for mistakes. If your synchronization strategy is difficult to explain, consider how you could simplify it.
 - You should also aim to make your synchronization as efficient as possible, in terms of time and memory.
 - Synchronization issues revolve around shared data. A good strategy for reasoning about synchronization is to identify which pieces of data are accessed by multiple independent actors (whether they are threads or interrupt handlers). Then, prove that the shared data always remains consistent.
 - Lists are a common cause of synchronization issues. Lists in Pintos are not thread-safe.
 - Do not forget to consider memory deallocation as a synchronization issue. If you want to use pointers to `struct thread`, then you need to prove those threads can't exit and be deallocated while you're using them.
 - If you create new functions, you should consider whether the function could be called in 2 threads at the same time. If your function access any global or static variables, you need to show that there are no synchronization issues.
 - Interrupt handlers cannot acquire locks. If you need to access a synchronized variable from an interrupt handler, consider disabling interrupts.
 - Locks do not prevent a thread from being preempted. Threads can be interrupted during a critical section. Locks only guarantee that the critical section is only entered by one thread at a time.
4. **Rationale** – Tell us why your design is better than the alternatives that you considered, or point out any shortcomings it may have. You should think about whether your design is easy to conceptualize, how much coding it will require, the time/space complexity of your algorithms, and how easy/difficult it would be to extend your design to accommodate additional features.

3.1.2 Design Document Additional Questions

You must also answer these additional questions in your design document:

- (This question uses the MLFQS scheduler.) Suppose threads A, B, and C have nice values 0, 1, and 2. Each has a `recent_cpu` value of 0. Fill in the table below showing the scheduling decision and the `recent_cpu` and `priority` values for each thread after each given number of timer ticks. We can use $R(A)$ and $P(A)$ to denote the `recent_cpu` and `priority` values of thread A, for brevity.

timer ticks	$R(A)$	$R(B)$	$R(C)$	$R(A)$	$R(B)$	$R(C)$	thread to run
0							
4							
8							
12							
16							
20							
24							
28							
32							
36							

- Did any ambiguities in the scheduler specification make values in the table (in the previous question) uncertain? If so, what rule did you use to resolve them?

3.1.3 Design Review

You can **voluntarily** schedule a 20-25 minute design review with your project TA. During the design review, your TA will ask you questions about your design for the project. You should be prepared to defend your design and answer any questions your TA may have about your design document. The design review is also a good opportunity to get to present your design and help you reach full marks for the design review.

3.2 Code (Due 4/05)

The code section of your grade will be determined by your score in our final test. Pintos comes with a test suite that you can run locally on your VM and we run the same tests in the final test. Your marks will be a weighted one, after considering your rank in your class.

3.3 Debugging with GDB (Due 4/05)

We will release a standard version of Pintos, with several alternations in code. These alternations result the failure execution of Pintos. You are required to find the causes and the failure locations with help of GDB. Your score for this part will be decided by your debugging logic. It is necessary to illustrate your inference process clearly in your submitted final report.

3.4 Final Report (Due 4/05) and Code Quality

After you complete the code for your project, you will submit a final report. Please include the following in your final report:

- the changes you made since your initial design document and why you made them
- a reflection on the project – Was you pushed by ddl? What went well, and what could be improved?

You will also be graded on the quality of your code. This will be based on many factors:

- Did you use consistent code style? Your code should blend in with the existing Pintos code. Check your use of indentation, your spacing, and your naming conventions.
- Is your code simple and easy to understand?
- If you have very complex sections of code in your solution, did you add enough comments to explain them?
- Did you copy-paste code instead of creating reusable functions?

4 Submission

4.1 What to submit

Each student is asked to submit 1 pdf and 1 zip file respecting the following requirements:

The first pdf file should:

- A pdf file named as **{studentnumber}-proj1DesignDoc.pdf** for your design document.
Example: 12345678-proj1DesignDoc.pdf

The zip file should:

- Create a folder named as **{studentnumber}-project1**.
Example: 12345678-project1
 - A folder named **{studentnumber}-pintos** for your pintos source code.
Example: 12345678-pintos

- A pdf file named as **{studentnumber}-report.pdf** for your final report.
Example: 12345678-report.pdf

- Zip the folder and name your zip: The zip should be named as **{studentnumber}-project1.zip**.
Example: 12345678-project1.zip

4.2 Where to submit and ddl

Upload your pdf and zip file via Blackboard.

The pdf file should be submitted no later than **23:00 (Beijing time) 21 March, 2021**.

The zip file should be submitted no later than **23:00 (Beijing time) 05 April, 2021**.

4.3 Remarks

You should read carefully all requirements of this project.

Prohibition

You will get 0 as score for this project if you break any of the prohibitions:

- You should not Plagiarism.
- You should give clear citation when you use others ideas.
- You should follow the naming policy.
- You should give a clear report for others to read.
- Your code should be well documented for others to understand.

Contact

For any question regarding this project, please email to **11749127@mail.sustech.edu.cn** (**Long XIANG**). The subject of the email should respect the format: **[OS] Project 1 (LastName/FirstName-StudentNumber)**. Example: **[OS] Project 1(Long/Xiang-123)**.

5 Reference

5.1 Pintos

To compile Pintos and run the Project 1 tests:

```
$ cd ~/pintos/src/threads
$ make
$ make check
```

The last command should run the Pintos test suite. These are the same tests for our final test. The skeleton code already passes some of these tests. By the end of the project, your code is expected to pass all of the tests.

5.1.1 Source Tree

threads/

The base Pintos kernel. Most of the modifications you will make for Project 1 will be in this directory.

userprog/

The user program loader. You will modify this for Project 2.

vm/

We will not use this directory.

filesystem/

The Pintos file system. You will use this file system in Project 2 and modify it in Project 3.

devices/

Source code for I/O device interfacing: keyboard, timer, disk, etc. You will modify the timer implementation in Project 1.

lib/

An implementation of a subset of the C standard library. The code in this directory is compiled into both the Pintos kernel and user programs that run inside it. You can include header files from this directory using the `#include <...>` notation. You should not have to modify this code.

lib/kernel/

Library functions that are only included in the Pintos kernel (not the user programs). It contains implementations of some data types that you should 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/

Library functions that are included only in Pintos user programs (not the kernel). In user programs, headers in this directory can be included using the `#include <...>` notation.

tests/

Tests for each project. You can add extra tests, but do not modify the given tests.

examples/

Example user programs that you can use in Project 2.

`misc/`, `utils/`

These files help you run Pintos. You should not need to interact with them directly.

`Makefile.build`

Describes how to build the kernel. Modify this file if you would like to add source code.

5.1.2 Building Pintos

This section describes the interesting files inside the `threads/build` directory.

`threads/build/Makefile`

A copy of `Makefile.build`. Don't change this file, because your changes will be overwritten if you make `clean` and re-compile. Make your changes to `Makefile.build` instead. For more information, see [Adding Source Files](#).

`threads/build/kernel.o`

Object file for the entire kernel. This is the result of linking object file compiled from each individual kernel source file into a single object file. It contains debug information, so you can run GDB or back- trace on it.

`threads/build/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.

`threads/build/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 file(`.o`) and dependency file(`.d`), both produced by the compiler. The dependency file tell make which source file need to be recompiled when other source or header file are changed.

5.1.3 Running Pintos

We've supplied a program for running Pintos in a simulator, called `pintos`, which should already be in your `PATH`. (If not, add `$HOME/.bin` to your `PATH`.)

The Pintos kernel takes a list of arguments, which tell the kernel what actions to perform. These actions are specified in the file `threads/init.c` on line 309 and look something like this.

```
static const struct action actions[] =  
{
```



```

    {"run", 2, run_task},
#ifdef FILESYS
    {"ls", 1, fsutil_ls},
    {"cat", 2, fsutil_cat},
    {"rm", 2, fsutil_rm},
    {"extract", 1, fsutil_extract},
    {"append", 2, fsutil_append},
#endif
    {NULL, 0, NULL},
};

```

The preprocessor macro `FILESYS` was not defined when we built Pintos earlier (it will be enabled in Project 2 and Project 3). So, we can only take one action: `run`. This `run` action will run a test specified by the next command-line argument. The number next to each action's name tells Pintos how many arguments there are (including the name of the action itself). For example, the `run` action's arguments are "`run <test_name>`". The tests that you can run are the files names in `tests/threads` (without the file extensions).

Let's try it out! First "`cd`" into the `threads` directory. Then run "`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 named `alarm-multiple`.

5.1.4 Debugging Pintos

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. One of the most important options is `--gdb` which will allow you to use `gdb` to debug the code you've written. For example, to run the debugger for the `alarm-multiple` test we would perform the following steps:

- `cd` into `~/pintos/src/threads`
- Run `pintos` with the debugger option "`pintos --gdb -- run alarm-multiple`". At this point, Pintos should say that it is "Waiting for gdb connection on port 1234".
- Open a new terminal and SSH into the VM
- `cd` into `~/pintos/src/threads/build`
- Open `cgdb` by running "`pintos-gdb kernel.o`". The `pintos-gdb` script loads `cgdb` with many useful GDB macros that will help you debug Pintos.
- In GDB, attach to Pintos by running "`target remote localhost:1234`". On your VM, you should be able to use "`debugpintos`" or "`deb`" as a shortcut to performing this step.
- Set a breakpoint at an interesting point in the program. Try "`break thread_init`".

- “continue” or “c” to start Pintos.

When you are working with the Pintos test suite, you should not start Pintos by constructing the command-line arguments manually. Many of the tests in the test suite require specific arguments to be passed to the simulator or to Pintos itself. You should read the next section for information about how to debug Pintos tests.

5.1.5 Debugging Pintos Tests

To debug Pintos test cases, you should first run the “make check” command, then **copy the command line arguments for the test that you are debugging**, and then add `--gdb` to it. Then, run `pintos-gdb` like you did in the previous section.

Try it out yourself. SSH into your VM and `cd` to `~/pintos/src/threads`. Then, run:

```
$ make clean
$ make
$ make check
```

Pay attention to the output of “make check”. You should see lines like these in the output:

```
pintos -v -k -T 60 --bochs -- -q run alarm-multiple < /dev/null 2> tests/threads...
perl -I../..
pass tests/threads/alarm-multiple
```

Here is an explanation:

- The first line runs the `pintos` script to start the Pintos kernel in a simulator, like we did before. The command uses the `-v -k -T 60 --bochs` options for the simulator, and the `-q run alarm-multiple` arguments for the Pintos kernel. Then, we use the `<`, `2>`, and `>` symbols to redirect standard input, standard error, and standard output to files.
- The second line uses the Perl programming language to run `../..tests/threads/alarm-multiple.ck` to verify the output of Pintos kernel.
- Using the Perl script from line 2, the build system can tell if this test passed or failed. If the test passed, we will see “**pass**”, followed by the test name. Otherwise, we will see “**fail**”, followed by the test name, and then more details about the test failure.

In order to debug this test, you should copy the command on the first line. Remove the input/output redirection (everything after the “`< /dev/null`”), because we want to see the output on the terminal when we’re debugging. Finally, add `--gdb` to the simulator options. (The `--gdb` must be before the double dashes, `--`, because everything after the double dashes is passed to the kernel, not the simulator.)

Your final command should look like:

```
$ pintos --gdb -v -k -T 60 --bochs -- -q run alarm-multiple
```

Run this command. Then, open a new terminal and cd to `~/code/group/pintos/src/threads/build` and run “`pintos-gdb kernel.o`” and type `debugpintos`” like you did in the previous section.

- You do not need to quit GDB after each run of the Pintos kernel. Just start pintos again and type `debugpintos` into GDB to attach to a new instance of Pintos. (If you re-compile your code, you must quit GDB and start it again, so it can load the new symbols from `kernel.o`.)
- Take some time to learn all the GDB shortcuts and how to use the CGDB interface. You may also be interested in looking at the Pintos GDB macros found in `~/bin/gdb-macros`

5.1.6 Debugging Page Faults

If you encounter a page fault during a test, you should use the method in the previous section to debug Pintos with GDB. If you use `pintos-gdb` instead of plain `gdb` or `cgdb`, you should get a backtrace of the page fault when it occurs:

```
pintos-debug: a page fault occurred in kernel mode
#0 test_alarm_negative () at ../../tests/threads/alarm-negative.c:14
#1 0xc000ef4c in ?? ()
#2 0xc0020165 in start () at ../../threads/start.S:180
```

If you want to inspect the original environment where the page fault occurred, you can use this trick:

```
(gdb) debugpintos
(gdb) continue
```

Now, wait until the kernel encounters the page fault. Then run these commands:

```
(gdb) set $eip = ((void**) $esp)[1]
(gdb) up
(gdb) down
```

You should now be able to inspect the local variables and the stack trace when the page fault occurred.

5.1.7 Debugging Kernel Panics

The Pintos source code contains a lot of “`ASSERT (condition)`” statements. When the condition of an assert statement evaluates to false, the kernel will panic and print out some debugging information. Usually, you will get a line of output that looks like this:

```
Call stack: 0xc0106eff 0xc01102fb 0xc010dc22 0xc010cf67 0xc0102319
0xc010325a 0x804812c 0x8048a96 0x8048ac8.
```

This is a list of instruction addresses, each of which corresponds to a frame on the kernel stack when the panic occurred. You can decode this information into a helpful stack trace by using the `backtrace` utility that is included in your VM.

```
$ cd
$ backtrace kernel.o 0xc0106eff 0xc01102fb 0xc010dc22 0xc010cf67 0xc0102319 ...
```

If you run your tests using “`make check`”, the testing framework will run `backtrace` automatically when it detects a kernel panic.

To debug a kernel panic with GDB, you can usually just set a breakpoint at the innermost line of code inside the stack trace. However, if your kernel panic occurs inside a function that is called many times, you may need to type `continue` a bunch of times before you reach the point in the test where the kernel panic occurs.

One trick you can use to improve this technique is to transform the code itself. For example, if you have an assert statement that looks like:

```
ASSERT (is_thread (next));
```

You can transform it into this:

```
if (!is_thread(next)) {
    barrier(); // Set a breakpoint HERE!
} ASSERT (is_thread (next));
```

Then, set a breakpoint at the line containing `barrier()`. You can use any line of code instead of `barrier()`, but you must ensure that the compiler cannot reorder or eliminate that line of code. For example, if you created a dummy variable “`int hello = 1;`” instead of using `barrier()`, the compiler could decide that line of code wasn’t needed and omit instructions for it! If you get a compile error while using `barrier()`, make sure you’ve included the `synch.h` header file. You can also use GDB’s conditional breakpoints, but if the assertion makes use of C macros, GDB might not understand what you mean.

5.2 Threads

5.2.1 Understanding Threads

The first step is to read and understand the code for the 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. 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, step through code and examine data, and so on.

When a thread is created, the creator specifies a function for the thread to run, as one of the arguments to `thread_create()`. The first time the thread is scheduled and runs, it starts executing from the beginning of that function. 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 become inactive. The scheduler decides which thread to run next. (If no thread is ready to run, then the special “idle” thread runs.)

The mechanics of a context switch are in `threads/switch.S`, which is x86 assembly code. It saves the state of the currently running thread and restores the state of the next thread onto the CPU.

Using GDB, try tracing through a context switch to see what happens. You can set a breakpoint on `schedule()` to start out, and then single-step from there (use “`step`” instead of “`next`”). Be sure to keep track of each thread's address and state, and what procedures are on the call stack for each thread (try “`backtrac`”). 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.

5.2.2 The Thread Struct

Each thread struct represents either a kernel thread or a user process. In each of the 3 projects, you will have to add your own members to the thread struct. You may also need to change or delete the definitions of existing members.

Every thread struct occupies the beginning of its own 4KiB 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 following **Thread Structure**.

This layout 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 the Memory Allocation section for more details.

- **Member of struct thread:** `tid_t tid`

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.

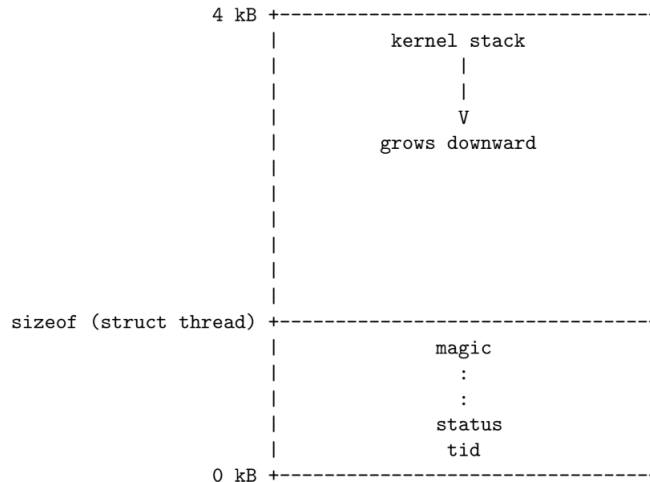


Fig - Thread Structure

- **Member of struct thread:** `enum thread_status status`

The thread's state, one of the following:

- Thread State: `THREAD_RUNNING`
The thread is running. Exactly one thread is running at a given time. `thread_current()` returns the running thread.
- Thread State: `THREAD_READY`
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 State: `THREAD_BLOCKED`
The thread is waiting for something, e.g. a lock to become available, an interrupt 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 conveniently done indirectly, using one of the Pintos synchronization primitives that block and unblock threads automatically.
- Thread State: `THREAD_DYING` The thread has exited and will be destroyed by the scheduler after switching to the next thread.

- **Member of struct thread:** `char name[16]`

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

- **Member of struct thread:** `uint8_t *stack`

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.

- **Member of struct thread:** `int priority`
A thread priority, ranging 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. Pintos currently ignores these priorities, but you will implement priority scheduling in this project.
- **Member of struct thread:** `struct list_elem allelem`
This “`list element`” is used to link the thread into the list of all threads. Each thread is inserted into this list when it is created and removed when it exits. The `thread_foreach()` function should be used to iterate over all threads.
- **Member of struct thread:** `struct list_elem elem`
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.
- **Member of struct thread:** `uint32_t *pagedir`
(Only used in Project 2 and later.) The page table for the process, if this is a user process.
- **Member of struct thread:** `unsigned magic`
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.

5.2.3 Thread Functions

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

- **Function:** `void thread_init (void)`
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 indirectly, including `lock_acquire()` for locking a lock, so `thread_init()` is called early in Pintos initialization.

- **Function:** `void thread_start (void)`
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()`.
- **Function:** `void thread_tick (void)`
Called by the timer interrupt at each timer tick. It keeps track of thread statistics and triggers the scheduler when a time slice expires.
- **Function:** `void thread_print_stats (void)`
Called during Pintos shutdown to print thread statistics.
- **Function:** `tid_t thread_create (const char *name, int priority, thread_func *func, void *aux)`
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 thread struct and stack and initializes its members, then it sets up a set of fake stack frames for it. The thread is initialized in the blocked state, then unblocked just before returning, which allows the new thread to be scheduled.
 - **Type:** `void thread_func (void *aux)`
This is the type of the function passed to `thread_create()`, whose `aux` argument is passed along as the function's argument.
- **Function:** `void thread_block (void)`
Transitions the running thread from the running state to the blocked state. 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.
- **Function:** `void thread_unblock (struct thread *thread)`
Transitions thread, which must be in the blocked state, to the ready state, allowing it to resume running. 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.
- **Function:** `struct thread *thread_current (void)`
Returns the running thread.
- **Function:** `tid_t thread_tid (void)`
Returns the running thread's thread id. Equivalent to `thread_current ()->tid`.
- **Function:** `const char *thread_name (void)`
Returns the running thread's name. Equivalent to `thread_current ()->name`.
- **Function:** `void thread_exit (void) NO_RETURN`
Causes the current thread to exit. Never returns, hence `NO_RETURN`.

- **Function:** `void thread_yield (void)`
 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.
- **Function:** `void thread_foreach (thread_action_func *action, void *aux)`
 Iterates over all threads `t` and invokes `action(t, aux)` on each. `action` must refer to a function that matches the signature given by `thread_action_func()`:
 - **Type:** `void thread_action_func (struct thread *thread, void *aux)`
 Performs some action on a thread, given `aux`.
- **Function:** `int thread_get_priority (void)`
Function: `void thread_set_priority (int new_priority)`
 Stub to set and get thread priority.
- **Function:** `int thread_get_nice (void)`
Function: `void thread_set_nice (int new_nice)`
Function: `int thread_get_recent_cpu (void)`
Function: `int thread_get_load_avg (void)`
 Stubs for the advanced MLFQS scheduler.

5.2.4 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 `thread_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

rst 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_threads()`.
- The next fake stack frame is for `switch_entry()`, an assembly language routine in `threads/switch.S` that adjusts the stack pointer, calls `thread_schedule_tail()` (this special case is why `thread_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.

5.3 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.

5.3.1 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.

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.

Some external interrupts cannot be postponed, even by disabling interrupts. These interrupts, 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`.

- **Type:** `enum intr_level` One of `INTR_OFF` or `INTR_ON`, denoting that interrupts are disabled or enabled, respectively.
- **Function:** `enum intr_level intr_get_level (void)`
Returns the current interrupt state.
- **Function:** `enum intr_level intr_set_level (enum intr_level level)`
Turns interrupts on or off according to level. Returns the previous interrupt state.
- **Function:** `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.

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.

5.3.2 Semaphores

A **semaphore** is a nonnegative integer together with two operators that manipulate it atomically, 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 semaphore. In such a case a lock, described below, may be more appropriate.

Semaphores can also be initialized to 0 or values larger than 1.

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

- **Type:** `struct semaphore`
Represents a semaphore.
- **Function:** `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)`
Executes the “down” or “P” operation on `sema`, waiting for its value to become positive and then decrementing it by one.
- **Function:** `bool sema_try_down (struct semaphore *sema)`
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.
- **Function:** `void sema_up (struct semaphore *sema)`
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.

Semaphores are internally built out of disabling interrupt 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`.

5.3.3 Locks

A **lock** is like a semaphore with an initial value of 1. A lock’s equivalent of “up” is called “release”, and the “down” operation is called “acquire”.

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`.

- **Type:** `struct lock`
Represents a lock.

- **Function:** `void lock_init (struct lock *lock)`
Initializes lock as a new lock. The lock is not initially owned by any thread.
- **Function:** `void lock_acquire (struct lock *lock)`
Acquires lock for the current thread, first waiting for any current owner to release it if necessary.
- **Function:** `bool lock_try_acquire (struct lock *lock)`
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.
- **Function:** `void lock_release (struct lock *lock)`
Releases lock, which the current thread must own.
- **Function:** `bool lock_held_by_current_thread (const struct lock *lock)`
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.

5.3.4 Monitor

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.

- **Type:** `struct condition`
Represents a condition variable.
- **Function:** `void cond_init (struct condition *cond)`
Initializes `cond` as a new condition variable.
- **Function:** `void cond_wait (struct condition *cond, struct lock *lock)`
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.

- **Function:** `void cond_signal (struct condition *cond, struct lock *lock)`
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.
- **Function:** `void cond_broadcast (struct condition *cond, struct lock *lock)`
Wakes up all threads, if any, waiting on `cond` (protected by monitor lock `lock`). `lock` must be held before calling this function.

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

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

5.3.5 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 unmodification 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 an 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.

5.4 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.

5.4.1 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 above 1 MiB, but the division can be changed with the `-ul` kernel command line option. 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 distinction will only become important starting with Project 2. 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.

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.

Page allocator types and functions are described below.

- **Function:** `void * palloc_get_page (enum palloc_flags flags)`

Function: `void * palloc_get_multiple (enum palloc_flags flags, size_t page_cnt)`

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:

- **Page Allocator Flag:** `PAL_ASSERT`

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.

- **Page Allocator Flag:** `PAL_ZERO`
Zero all the bytes in the allocated pages before returning them. If not set, the contents of newly allocated pages are unpredictable.
- **Page Allocator Flag:** `PAL_USER`
Obtain the pages from the user pool. If not set, pages are allocated from the kernel pool.
- **Function:** `void palloc_free_page (void *page)`
Function: `void palloc_free_multiple (void *pages, size_t page_cnt)`
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`.

5.4.2 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 KiB 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 KiB. 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 satis

ed 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 KiB each.

When a block is freed, all of its bytes are cleared to 0xcc, as a debugging aid.

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.

- **Function:** `void * malloc (size_t size)`
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.

- **Function:** `void * calloc (size_t a, size_t b)`
Obtains and 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.
- **Function:** `void * realloc (void *block, size_t new_size)`
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 equivalent to `free`.
- **Function:** `void free (void *block)`
Frees `block`, which must have been previously returned by `malloc`, `calloc`, or `realloc` (and not yet freed).

5.5 Linked Lists

Pintos contains a linked list data structure in `lib/kernel/list.h`, used for many different purposes. This linked list implementation differs from most other linked list implementations you may have encountered, because **it does not use any dynamic memory allocation**.

```
/* List element.
struct list_elem
{
    struct list_elem *prev; /* Previous list element.
    struct list_elem *next; /* Next list element.
};

/* List.
struct list
{
    struct list_elem head; /* List head.
    struct list_elem tail; /* List tail.
};
```

In a Pintos linked list, each list element contains a “`struct list_elem`”, which contains the pointers to the next and previous element. Since list elements themselves have enough space to hold the `prev` and `next` pointers, we don't need to allocate any extra space to support our linked list. Here is an example of a linked list element which can hold an integer:

```
struct int_list_elem
{
    int value;
    struct list_elem elem;
```

```
};
```

Next, you must create a “`struct list`” to represent the whole list. Initialize it with `list_init()`.

```
/* Declare and initialize a list */
struct list my_list;
list_init (&my_list);
```

Now, you can declare a list element and add it to the end of the list. Notice that the second argument of `list_push_back()` is the address of a “`struct list_elem`”, not the “`struct int_list_elem`” itself.

```
/* Declare a list element.
struct int_list_elem three = 3, NULL, NULL;
/* Add it to the list */
list_push_back (&my_list, &three.elem);
```

We can use the `list_entry()` macro to convert a generic “`struct list_elem`” into our custom “`struct int_list_elem`” type. Then, we can grab the “value” attribute and print it out:

```
/* Fetch elements from the list */
struct list_elem *first_list_element = list_begin (&my_list);
struct int_list_elem *first_integer = list_entry (first_list_element,
                                                  struct int_list_elem,
                                                  elem);
printf("The first element is: %d\n", first_integer->value);
```

By storing the `prev` and `next` pointers inside the structs themselves, we can avoid creating new “linked list element” containers. However, this also means that a `list_elem` can only be part of one list a time. Additionally, our list should be homogeneous (it should only contain one type of element).

The `list_entry()` macro works by computing the offset of the `elem` field inside of “`struct int_list_elem`”. In our example, this offset is 4 bytes. To convert a pointer to a generic “`struct list_elem`” to a pointer to our custom “`struct int_list_elem`”, the `list_entry()` just needs to subtract 4 bytes! (It also casts the pointer, in order to satisfy the C type system.)

Linked lists have 2 sentinel elements: the `head` and `tail` elements of the “`struct list`”. These sentinel elements can be distinguished by their `NULL` pointer values. Make sure to distinguish between functions that return the first actual element of a list and functions that return the sentinel `head` element of the list.

There are also functions that sort a link list (using quicksort) and functions that insert an element into a sorted list. These functions require you to provide a list element comparison

function (see `lib/kernel/list.h` for more details).

5.6 Efficient Alarm Clock

Here are some more details about the Efficient Alarm Clock task.

1. If `timer_sleep()` is called with a zero or negative argument, then you should just return immediately.
2. When you run Pintos, the clock does **not** run in realtime by default. So, if a thread goes to sleep for 5 “seconds” (e.g. ticks = $5 \times \text{TIMER_FREQ}$), it will actually be much shorter than 5 seconds in terms of wall clock time. You can use the `--realtime` flag for Pintos to override this.
3. 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.
4. The code that runs in interrupt handlers (i.e. `timer_interrupt()`) should be as fast as possible. It’s usually wise to do some pre-computation outside of the interrupt handler, in order to make the interrupt handler as fast as possible. Additionally, you may not acquire locks while executing `timer_interrupt()`.
5. Pay close attention to the Pintos linked-list implementation. Each linked list requires a dedicated `list_elem` member inside its elements. Every element of a linked list should be the same type. If you create new linked lists, make sure that they are initialized. Finally, make sure that there are no race conditions for any of your linked lists (the list manipulation functions are **NOT** thread-safe).

5.7 Priority Scheduler

Here are some more details about the Priority Scheduler task.

1. A thread’s initial priority is an argument of `thread_create()`. You should use `PRI_DEFAULT (31)`, unless there is a reason to use a different value.
2. Your implementation must handle nested donation: Consider a high-priority thread H, a medium- priority thread M, and a low-priority thread L. If H must wait on M and M must wait on L, then we should donate H’s priority to L.
3. A thread can only donate to 1 thread at a time, because once it calls `lock_acquire()`, the donor thread is blocked.
4. If there are multiple waiters on a lock when you call `lock_release()`, then all of those priority donations must apply to the thread that receives the lock next.

5. You do not need to handle priority values outside of the allowed range, `PRI_MIN (0)` to `PRI_MAX (63)`.
6. You only need to implement priority donation for locks. Do not implement them for other synchronization variables (it doesn't make any sense to do it for semaphores or monitors anyway). However, you need to implement priority scheduling for locks, semaphores, and condition variables. Priority scheduling is when you unblock the highest priority thread when a resource is released or a monitor is signaled.
7. Don't forget to implement `thread_get_priority()`, which is the function that returns the current thread's priority. This function should take donations into account. You should return the **effective priority** of the thread.
8. A thread cannot change another thread's priority, except via donations. The `thread_set_priority()` function only acts on the current thread.
9. If a thread no longer has the highest effective priority (e.g. because it released a lock or it called `thread_set_priority()` with a lower value), it must immediately yield the CPU. If a lock is released, but the current thread still has the highest effective priority, it should not yield the CPU.

5.8 Advanced Scheduler

Here are some more details about the Advanced Scheduler task.

5.8.1 Introduction

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 this task, you must implement the scheduler according to the specification in this section. However, the exact method of implementation is up to you. As long as the behavior of your scheduler matches the specification here, it is acceptable.

Your advanced scheduler is 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. The scheduler always 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 parts of this 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.

When the advanced scheduler is enabled, you should **NOT** do priority donation.

5.8.2 Fixed-point Real Numbers

Many of the calculations in the following section assume that you're using real numbers, not integers. However, Pintos does not support floating point number operations. We have provided the `fixed-point.h` library inside `pintos/src/threads/fixed-point.h`, which will allow you to use fixed point numbers to represent real numbers. You should use `fixed_point_t` and the functions defined in `fixed-point.h` to represent any value that needs to be a real number. If you use integers, your values will not be correct.

5.8.3 Niceness

Each thread 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 the test framework. We have provided skeleton definitions for them in “`threads/thread.c`”.

- **Function:** `int thread_get_nice (void)`
Returns the current thread's `nice` value.
- **Function:** `void thread_set_nice (int new_nice)`
Sets the current thread's `nice` value to `new_nice` and recalculates the thread's priority based on the new value. If the running thread no longer has the highest priority, it should yield the CPU.

5.8.4 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

$$\text{priority} = \text{PRI_MAX} - (\text{recent_cpu}/4) - (\text{nice} \times 2)$$

In this formula, `recent_cpu` is an estimate of the CPU time the thread has used recently (see the next section on `recent_cpu`) and `nice` is the thread's `nice` value. The result should be rounded down to the nearest integer (truncated). 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 is designed so that threads that have recently been scheduled on the CPU will have a lower priority the next time the scheduler picks a thread to run. 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 very high nice value, should ensure that it receives CPU time soon.

5.8.5 Calculating Recent CPU

We wish `recent_cpu` to measure how much CPU time each process has received “recently.” 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 an exponentially weighted moving average, which takes this general form:

$$\begin{aligned}x(0) &= f(0) \\x(t) &= a \times x(t-1) + f(t) \\a &= k/(k+1)\end{aligned}$$

In this formula, $x(t)$ is the moving average at integer time $t \geq 0$, $f(t)$ is the function being averaged, and k controls the rate of decay. We can iterate the formula over a few steps as follows:

$$\begin{aligned}x(1) &= f(1) \\x(2) &= a \times f(1) + f(2) \\x(3) &= a^2 \times f(1) + a \times f(2) + f(3) \\x(4) &= a^3 \times f(1) + a^2 \times f(2) + a \times f(3) + f(4)\end{aligned}$$

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 $t+2 \times k$, and so on. From the opposite direction, $f(t)$ decays to weight w at time $t + \ln(w)/\ln(a)$.

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:

$$recent_cpu = (2 \times load_avg)/(2 \times load_avg + 1) \times recent_cpu + nice$$

In this formula, `load_avg` is a moving average of the number of threads ready to run (see the next section). 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 0.1 in $\ln(0.1)/\ln(\frac{2}{3}) = approx.6$ seconds; if `load_avg` is 2, then decay to a weight of 0.1 in $\ln(0.1)/\ln(\frac{3}{4}) = approx.8$ 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 these recalculations of `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`”.

- **Function:** `int thread_get_recent_cpu(void)`

Returns 100 times the current thread’s `recent_cpu` value, rounded to the nearest integer.

5.8.6 Calculating Load Average

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:

$$\text{load_avg} = (59/60) \times \text{load_avg} + (1/60) \times \text{ready_threads}$$

In this formula, `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`”.

- **Function:** `int thread_get_load_avg(void)`

Returns 100 times the current system load average, rounded to the nearest integer.

5.8.7 Summary

The following formulas summarize the calculations required to implement the scheduler.

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:

$$\text{priority} = \text{PRI_MAX} - (\text{recent_cpu}/4) - (\text{nice} \times 2)$$

`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:

$$\text{recent_cpu} = (2 \times \text{load_avg}) / (2 \times \text{load_avg} + 1) \times \text{recent_cpu} + \text{nice}$$

`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:

$$\text{load_avg} = (59/60) \times \text{load_avg} + (1/60) \times \text{ready_threads}$$

`ready_threads` is the number of threads that are either running or ready to run at time of update (not including the idle thread).

5.8.8 Additional Details

1. When the advanced scheduler is enabled, you should **NOT** do priority donation.
2. When the advanced 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.
3. Because many of these formulas involve fractions, you should use **fixed-point real arithmetic** for your calculations. Use the `fixed_point_t` type and the library functions inside `pintos/src/threads/fixed-point.h` to do your advanced scheduler calculations.

5.9 Debugging Tips

Many tools lie at your disposal for debugging Pintos. This section introduces you to a few of them.

5.9.1 `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 Triple Faults, for a related technique.

5.9.2 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) 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 Backtraces, for more information.

5.9.3 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 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 Appended to a function prototype to tell the compiler that the function never returns. It allows the compiler to
ne-tune its warnings and its code generation.

NO INLINE 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) 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.

5.9.4 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. `debug_backtrace_all`, also declared in `<debug.h>`, prints backtraces of all threads.

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.

Here's an example. Suppose that Pintos printed out this following call stack:

```
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:
0xc010cf67:
0xc0102319:
0xc010325a:
0x0804812c:
0x08048a96:
0x08048ac8:
```

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 `fileys/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/fileys/extended/grow-too-big 0xc0106eff 0xc01102fb 0xc010dc22
0xc010cf67 0xc0102319 0xc010325a 0x804812c 0x8048a96 0x8048ac8
```

The results look like this:

```
0xc0106eff:
0xc01102fb:
0xc010dc22:
0xc010cf67:
0xc0102319:
0xc010325a:
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/fileys/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 (fileys/file.c:405)|
```

```
0xc010dc22:
0xc010cf67:
0xc0102319:
0xc010325a:
```

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 0x0804812c 0x08048a96 0x08048ac8.
```

5.9.5 GDB

Please change the path `GDBMACROS=/usr/class/cs140/pintos/pintos/src/misc/gdb-macros` within `.../pintos/src/utlis/pintos-gdb` to your `gdb-macros`, like `GDBMACROS=.../pintos/src/misc/gdb-macros` and then perform following debugging steps. When connecting `gdb` to current running `pintos`, please enter the corresponding `build` directory, in order to find `kernel.o`

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`

```
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:

Continues execution until **ctrl+c** or the next breakpoint.

break function

break file:line

break *address 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 Evaluates the given **expression** and prints its value. If the expression contains a function call, that function will actually be executed.

l *address Lists a few lines of code around **address**. (use a **0x** prefix to specify an address in hex.)

bt Prints a stack backtrace similar to that output by the **backtrace** program described above.

p/a address Prints the name of the function or variable that occupies address. (use a **0x** prefix to specify an address in hex.)

disassemble function disassembles **function**.

We also provide a set of macros specialized for debugging Pintos, written by Godmar Back (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 Attach debugger to a waiting Pintos process on the same machine. Shorthand for **target remote localhost:1234**.

dumplist &list type element 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 allelem** prints all elements of **struct thread** that are linked in **struct list all_list** using the **struct list_elem allelem** which is part of **struct thread**.

btthread thread 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 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 allelem** shows the backtraces of all threads contained

in `struct list all_list`, linked together by `allelem`. This command is useful to determine where your threads are stuck when a deadlock occurs. Please see the example scenario below.

btthreadall short-hand for `btthreadlist all_list allelem`.

btpagefault 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 wish to also load the user program's symbol table using the `loadusersymbols` macro, as described above.

hook-stop 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.

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.

Sample GDB section 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 `m1fqslload_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.

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
```

```
=====
```

```
0000000000i[ ] reading configuration from Bochssrc.txt
```

```
0000000000i[ ] enabled gdbstub
```

```
0000000000i[ ] installing nogui module as the Bochs gui
```

```
0000000000i[ ] 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
```

```
0x0000ffff in ?? ()
```

```
reply contains invalid hex digit 78
```

Now I tell Pintos to run by executing c (short for `continue`):

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...
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 `allelem`, 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 allelem
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 diagnosing deadlocks or unexplained hangs.

loadusersymbols You can also use `gdb` to debug a user program running under Pintos. To do that, use the `loadusersymbols` macro to load the program’s symbol table:

```
loadusersymbol program
```

Where `program` is the name of the program’s executable (in the host file system, not in the Pintos file system). For example, you may issue:

```
(gdb) loadusersymbols tests/userprog/exec-multiple
add symbol table from file "tests/userprog/exec-multiple" at
.text_addr = 0x80480a0
```

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: `GDB` does not know which process is currently active (because that is an abstraction the Pintos kernel creates). 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 `loadusersymbols` to load `kernel.o`.) `loadusersymbols` is implemented via `gdb`’s `add-symbol-file` command.

5.9.6 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. 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 in

nite 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.

5.9.7 General 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.

5.10 FAQ

- **How much code will I need to write?**

Here's a summary of a reference solution. 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 solutions 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/synch.c | 88 ++++++++
threads/thread.c | 196 ++++++++-----
threads/thread.h | 23 +++
4 files changed, 310 insertions(+), 29 deletions(-)
```

5.10.1 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 302 curriculum.

5.10.2 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 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.

5.10.3 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.

6 Acknowledgment

This project is originally designed by Stanford for their operating System course. Our project materials are based on the released guide for UC Berkeley CS 162 Operating System. We check the project materials from both schools and combine useful information together to form this one. You can find the original version on internet.