Lab 3: Batch Scheduling in Pintos

Operating Systems Course
Chalmers and Gothenburg University

October 4, 2021

1 Assignment Description

Similar to lab2, in this lab you are called to solve a simple problem in Pintos, and more specifically to handle the synchronization issues that arise when scheduling different batches of jobs. The assumption is that our system is extended with an external processing accelerator (e.g. a GPU or a co-processor) with X Processing Units (PUs). Tasks `task_t` are handled by one thread each, and contain the appropriate data/results from/to the accelerator. However, the communication bus with the accelerator is half duplex (i.e. one direction can be used at a time) and has limited bandwidth as only 3 slots can be used by tasks at a time.

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

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

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

The main part of this assignment is to:

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

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

To accomplish this lab, implement the function prototypes provided in `'devices/batch-scheduler.c'` enforcing the required constraints.

2 Background

2.1 Understanding Threads

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

Some of this code might seem slightly mysterious. If you haven't already compiled and run the base system, as described in the lab2, you should do so now. You can read through parts of the source code to see what's going on. If you like, you can add calls to `printf()` almost anywhere, then recompile and run to see what happens and in what order. You can also run the kernel in a debugger and set breakpoints at interesting spots, single-step through code and examine data, and so on.

When a thread is created, you are creating a new context to be scheduled. You provide a function to be run in this context as an argument to `thread_create`. The first time the thread is scheduled and runs, it starts from the beginning of that function and executes in that context. When the function returns,

the thread terminates. Each thread, therefore, acts like a mini-program running inside Pintos, with the function passed to `thread_create` acting like `main`.

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

The mechanics of a context switch are in `threads/switch.S`, which is $80x86$ assembly code. (You don't have to understand it.) It saves the state of the currently running thread and restores the state of the thread we're switching to.

Using the GDB debugger, slowly trace through a context switch to see what happens. You can set a break point on `schedule` to start out, and then single-step from there. [1] Be sure to keep track of each thread's address and state, and what procedures are on the call stack for each thread. You will notice that when one thread calls `switch_threads`, another thread starts running, and the first thing the new thread does is to return from `switch_threads`. You will understand the thread system once you understand why and how the `switch_threads` that gets called is different from the `switch_threads` that returns.

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

2.2 Source Files

Despite Pintos being a tiny operating system, the code volume can be quite discouraging at first sight. Do not panic: lab2 has already helped you understand Pintos by working on a small fragment of the code. You will not need to modify most of this code, but the hope is that presenting this overview will give you a start on what code to look at.

2.2.1 *'threads'* code

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

`'loader.S'`
`'loader.h'`
The kernel loader. Assembles to 512 bytes of code and data that the PC BIOS loads into memory and which in turn finds the kernel on disk, loads it into memory, and jumps to `start` in `start.S`. You should not need to look at this code or modify it.

`'start.S'`
Does basic setup needed for memory protection and 32-bit operation on $80x86$ CPUs. Unlike the loader, this code is actually part of the kernel.

`'kernel.lds.S'`
The linker script used to link the kernel. Sets the load address of the kernel and arranges for `start.S` to be near the beginning of the kernel image. Again, you should not need to look at this code or modify it, but it's here in case you're curious.

`'init.c'`
`'init.h'`
Kernel initialization, including `main`, the kernel's "main program". You should look over `main` at least to see what gets initialized.

`'thread.c'`
`'thread.h'`
Basic thread support. `thread.h` defines `thread`. You have already modified these files for lab2.

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

`'switch.S'`
`'switch.h'`
Assembly language routine for switching threads. Already discussed above.

`'palloc.c'`
`'palloc.h'`
Page allocator, which hands out system memory in multiples of 4 kB pages.

`'malloc.c'`
`'malloc.h'`
A simple implementation of `malloc` and `free` for the kernel.

`'interrupt.c'`
`'interrupt.h'`
Basic interrupt handling and functions for turning interrupts on and off.

`'intr-stubs.S'`
`'intr-stubs.h'`
Assembly code for low-level interrupt handling.

`'synch.c'`
`'synch.h'`
Basic synchronization primitives: semaphores, locks, condition variables, and optimization barriers. You will need to use these for synchronization in lab3.

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

`'vaddr.h'`
`'pte.h'`
Functions and macros for working with virtual addresses and page table entries.

`'flags.h'`
Macros that define a few bits in the 80x86 "flags" register. Probably of no interest

2.2.2 *'devices'* code

The basic threaded kernel also includes these files in the *'devices'* directory:

`'timer.c'`
`'timer.h'`
System timer that ticks, by default, 100 times per second. You have already modified this code for lab2.

`'batch-scheduler.c'`
Contains code skeleton to be used for implementing this assignment.

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

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

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

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

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

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

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

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

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

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

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

2.2.3 *‘lib’* files

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

‘ctype.h’
‘inttypes.h’
‘limits.h’
‘stdarg.h’
‘stdbool.h’
‘stddef.h’
‘stdint.h’
‘stdio.c’
‘stdio.h’
‘stdlib.c’
‘stdlib.h’
‘string.c’
‘string.h’
A subset of the standard C library.

‘debug.c’
‘debug.h’
Functions and macros to aid debugging.

‘random.c’
‘random.h’
Pseudo-random number generator. The actual sequence of random values will not vary from one

Pintos run to another, unless you do one of three things: specify a new random seed value on the `-rs` kernel command-line option on each run, or use a simulator other than Bochs, or specify the `-r` option to `pintos`.

`'round.h'`
Macros for rounding.

`'syscall-nr.h'`
System call numbers.

`'kernel/list.c'`
`'kernel/list.h'`
Doubly linked list implementation. Used all over the Pintos code, and you probably want to use it a few places yourself.

`'kernel/bitmap.c'`
`'kernel/bitmap.h'`
Bitmap implementation. You can use this in your code if you like, but you probably won't have any need for it.

`'kernel/hash.c'`
`'kernel/hash.h'`
Hash table implementation.

`'kernel/console.c'`
`'kernel/console.h'`
`'kernel/stdio.h'`
Implements `printf` and a few other functions.

2.3 Synchronization

Proper synchronization is an important part of the solutions to these problems. Any synchronization problem can be easily solved by turning interrupts off: while interrupts are off, there is no concurrency, so there's no possibility for race conditions. Therefore, it's tempting to solve all synchronization problems this way, but `don't`. Instead, use semaphores, locks, and condition variables to solve the bulk of your synchronization problems.

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

When you do turn off interrupts, take care to do so for the least amount of code possible, or you can end up losing important things such as timer ticks or input events. Turning off interrupts also increases the interrupt handling latency, which can make a machine feel sluggish if taken too far.

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

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

There should be no busy waiting in your submission. A tight loop that calls `thread_yield` is one form of busy waiting.

3 Development Suggestions

In the past, many groups divided the assignment into pieces, then each group member worked on his or her piece until just before the deadline, at which time the group reconvened to combine their code and submit. `This is a bad idea. We do not recommend this approach.` Groups that do this often

find that two changes conflict with each other, requiring lots of last-minute debugging. Some groups who have done this have turned in code that did not even compile or boot, much less pass any tests.

Instead, we recommend integrating your team's changes early and often, using a source code control system such as SVN or GIT. This is less likely to produce surprises, because everyone can see everyone else's code as it is written, instead of just when it is finished. These systems also make it possible to review changes and, when a change introduces a bug, drop back to working versions of code.

4 Testing

This lab has one test, to check the termination of the execution. It is important you get $'pass'$ from the test, however, this does not mean passing the lab. Your solution will be graded through code inspection to verify the correctness of the synchronization algorithms (in addition to basic functionality checks of the running code).

4.1 Submission

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

1. **Writing the report** For your report, begin by describing the implementation of your solution. More specifically, briefly analyze how you implemented each of the following:

   - **Data Structures** Highlight for us the actual changes to data structures. Also add a very brief description of the purpose of each new or changed data structure. The limit of 25 words or less is a guideline intended to save your time and avoid duplication with later areas.
   - **Algorithms** This is where you tell us how your code works. We might not be able to easily figure it out from the code, because many creative solutions exist for most OS problems. Help us out a little. Your report should be at a level below the high level description of requirements given in the assignment. We have read the assignment too, so it is unnecessary to repeat or rephrase what is stated there. On the other hand, your description should be at a level above the low level of the code itself. Don't give a line-by-line run-down of what your code does. Instead, use your report to explain how your code works to implement the requirements.
   - **Synchronization** An operating system kernel is a complex, multi-threaded program, in which synchronizing multiple threads can be difficult. That is why we want you to explain explicitly how you chose to synchronize this particular type of activity.
   - **Rationale** Whereas the other sections primarily ask "what" and "how", the rationale section concentrates on "why". This is where we would like you to justify some design decisions, by explaining why the choices you made are better than alternatives. You may be able to state these in terms of time and space complexity, which can be made as rough or informal arguments (formal language or proofs are unnecessary).

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

   Pintos is written in a consistent style. Make your additions and modifications in existing Pintos source files blend in, not stick out. In new source files, adopt the existing Pintos style by preference, but make your code self-consistent at the very least. There should not be a patchwork of different styles that makes it obvious that three different people wrote the code. Use horizontal and vertical white space to make code readable. Add a brief comment on every structure, structure member, global or static variable, typedef, enumeration, and function definition. Update existing comments as you modify code. Don't comment out or use the preprocessor to ignore blocks of code (instead,

remove it entirely). Use assertions to document key invariants. Decompose code into functions for clarity. Code that is difficult to understand because it violates these or other "common sense" software engineering practices will be penalized.

After you have verified that your code works correctly on the StuDAT machines, run the `prepare-submission` script found in the lab folder. The script will check that your code compiles correctly and it will create an archive with only the necessary files for grading.

3. **Final submission** For the final submission, prepare an archive containing the archive of your code (prepared as per the instructions above) and the report file and upload it to canvas.

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

5 FAQ

- **How do I update the `'Makefile'`s when I add a new source file?**

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

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

  A new `'.h'` file does not require editing the `'Makefile'`s.

- **What does `warning: no previous prototype for 'func'` mean?**

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

- **What is the interval between timer interrupts?**

  Timer interrupts occur `TIMER_FREQ` times per second. You can adjust this value by editing `'devices/timer.h'`. The default is 100 Hz.

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

- **How long is a time slice?**

  There are `TIME_SLICE` ticks per time slice. This macro is declared in `'threads/thread.c'`. The default is 4 ticks.

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

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

  You are probably looking at a backtrace that looks something like this:

$$0xc0108810 : debug\_panic(lib/kernel/debug.c : 32)$$

$$0xc010a99f : pass(tests/threads/tests.c : 93)$$

$$0xc010bdd3 : test\_mlfqs\_load\_1(...threads/mlfqs - load - 1.c : 33)$$

$$0xc010a8cf : run\_test(tests/threads/tests.c : 51)$$

$$0xc0100452 : run\_task(threads/init.c : 283)$$

$0xc0100536 : run\_actions(threads/init.c : 333)$

$0xc01000bb : main(threads/init.c : 137)$

This is just confusing output from the `backtrace` program. It does not actually mean that `pass()` called `debug_panic()`. In fact, `fail()` called `debug_panic()` (via the `PANIC` macro). GCC knows that `debug_panic()` does not return, because it is declared `NO_RETURN`, so it doesn't include any code in `fail()` to take control when `debug_panic()` returns. This means that the return address on the stack looks like it is at the beginning of the function that happens to follow `fail()` in memory, which in this case happens to be `pass()`.

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

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

  - `thread_exit()`, but we'll never switch back into such a thread, so it's uninteresting.
  - `thread_yield()`, which immediately restores the interrupt level upon return from `schedule()`.
  - `thread_block()`, which is called from multiple places:
    - `sema_down()`, which restores the interrupt level before returning.
    - `idle()`, which enables interrupts with an explicit assembly STI instruction.
    - `wait()` in `'devices/intq.c'`, whose callers are responsible for re-enabling interrupts.

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