Program: Ticket Lock

CIS 310: Operating Systems

Brian C. Ladd

Fall 2020

Learning Outcomes

- Using `SpinLock`
- Thinking about critical regions, locks, and threads.
- Fairness in concurrency.

Introduction

Mutual Exclusion

An atomic instruction such as `test_and_set` is a low-level **mechanism** to permit safe concurrency. It is *atomic* because the hardware is designed to make it atomic. Atomicity is costly in silicon and clock-cycles so the instruction is as basic as possible; `test_and_set` is the smallest brick in building mutual exclusion. A `SpinLock` enforces mutual exclusion on a single critical region (as when changing a single variable) or multiple, related critical regions (as inside *each* method of a `LinkedList` data structure). The `SpinLock` is **correct** in that at most one thread can hold the lock and, as long as the thread holding it eventually releases it, some thread will be able to enter the critical region. `SpinLock` is not, however, **fair**: mutual exclusion is fair if, when thread A is waiting on the lock while thread B holds it, A will get the lock before B can **reclaim** it after releasing it. One fair mutual exclusion primitive is the `TicketLock`.

`TicketLock`

A `TicketLock` uses two counters, protected as necessary with `SpinLock`, to provide mutual exclusion and fairness.

Motivation: think of a deli counter

1. Initially: ticket (order number) is 0.
   (a) A thread "takes" a ticket when it wants to enter the critical region.

   (b) When a ticket is taken, the ticket number is incremented.

2. Initially: turn ("Now Serving" number) is 0.
   (a) Thread with ticket == turn can enter its critical region.
   (b) On exit, turn is incremented.

Ticket and turn are incremented modulo some N.

N is a parameter to constructing a ticket lock.

N must be no less than the number of threads that might contend for the critical region.

(Why? Many a final exam has had a question like that.)

Testing requires the use of an increment function that attempts to maximize the chances of race conditions turning out badly. This means that commandline testing is more likely to produce anomalous results until the `TicketLock` is correct.

Overview

Multiple threads that share related critical regions must each refer to a single `TicketLock`.

The ticket lock is shared: the counters are therefore critical regions.

1. Mutual exclusion should be encapsulated within the `TicketLock`.
2. Lock as little code as possible.
3. This is where you can use `SpinLock`.
4. Use one lock for both counters.

Getting Started

`src/` in the assignment repo, `pTicketLock` on Gitea, contains starting code.

`tools/ticketLockTest.cpp` contains

- `main` processes command-line parameters (iterations and threads), creates, runs, and joins all threads.
- `thread_worker` creates a randomness generator for the thread and then loops, pausing and incrementing the global `counter` variable.
- `random_pause` a function that sleeps for a random amount of time on a given range.

`os_dependent` folder contains

- `xchg.h` header-only implementation of the `xchg` function looked at in class.
- `SpinLock.h` header-only implementation of `SpinLock`; contains its own, private implementation of `xchg`.

`locks/TicketLock.*` These are the .h and .cpp for your code. The `TicketLock` class has three functions defined and a small embedded class.

- `TicketLock` constructor. Takes the size of the ticket cycle.
- `lock` obtains the lock. Will not return until the thread that called it has the lock (or has the ticket equal to the turn).
- `unlock` advances the turn, thereby invalidating its own (already used) ticket.
- `evil_increment` an increment function for `unsigned int` variables. Students **must** use this function to increment values in `TicketLock`: do not use ++ or – within this class.

1. `Ticket` is the type returned by the `lock`. It has two fields:
   (a) `ticket` the ticket number issued to the thread when it called the lock function. We know that it is also the value of the turn counter when the thread's call to the lock function returns.
   (b) `initial_turn` the number in the turn counter when the function was first entered. This indicates how long the thread waited inside the lock function before incrementing the counter.

Use any concurrency code in the repository (or described in the book) to implement the ticket lock.

If using a different primitive: include header/implementation files here as necessary.

Documentation

Note that these requirements, repeated or not, apply to *all* programming assignments in CIS 310.

Do not forget the `README.org` or `README.txt` file

The `README` document goes in the root directory of the project (where the `Makefile` lives)

It is in plain text or Org mode formatting

It must contain (at least) the following:

Identification Block Much as described in the next section, the `README` must identify the programmer (with e-mail address) and the problem being solved. No ID block is the same as no `README`.

Problem Restatement Restate the problem being solved to make the project self-contained. Restating the problem is also good practice to check that you understand what you are supposed to do.

Testing Criteria You know by now that "it must be right, it compiles" is a silly statement. So, how do you know that you are done? You must document exactly how you tested your program with

Test Input Files or descriptions of what to give as input

Test Execution Commandlines and answers to prompts to execute your program with each set of test data.

Expected Output How to find the output and what the output is supposed to be. This should refer back to the input data and the assignment to establish that the expected output matches the problem being solved.

Compiling and Exectuitg Instructions Give clear *commandline specifications* for compiling and running your program. What folder should the user be in to run the commands? What tool(s) does the process require? What do the commandline arguments *mean*?

The `README` must accompany every program you turn in.

Do not forget ID blocks in each C++ file and `README`

Example header block for a Java/C++ file

Taken from Departmental Coding Standards

```
/**
 * Gargoyle draws a random ASCII art monster on standard output.
 *
 * Gargoyle has all static methods (and no constructor) including
 * main. It is run with a single integer on the command-line that
 * is used to randomize the monster that is generated.
 *
 * @author Jimmy A. Student
 * @email studeja199@potsdam.edu
 * @course CIS 203 Computer Science II
 * @assignment p004
 * @due 04/25/2018
```

```
*/
```

Function comments must document intent.

Why is this computation broken out into a function?

What does it do?

1. This is in the language of the *caller*.
   (a) A function is the **interface** between two levels of *abstraction*.
      i. The header documentation is written for the *higher level* of abstraction.
      ii. The code (and its included documentation) is for the lower level of abstraction.

What ore the *parameters*?

1. Document expected range of values, checks done on parameters, etc.

What errors/exceptions can happen?

1. Document both what exceptions and what they mean (to the *caller*).

What *preconditions* must pertain for this function to perform correctly?

What *postconditions* will this function put in place when run?