CS 343 Operating Systems, Fall 2022
Producer-Consumer Lab: Concurrency Control

Contents

1 Introduction

The purpose of this lab is for you to engage with the challenges of concurrency control in the context of an important problem in every concurrent system: the producer-consumer problem. The framework of the lab, while user-level, attempts to emulate the environment of a modern kernel, for example Linux.

You may work in a group of up to three people in this lab. Clarifications and revisions will be posted to the course discussion group.

2 Setup

You can work on this lab on any modern Linux system, although we will test your work on the class server (Moore). We will describe the details of how to access the lab repo via Github Classroom on Campuswire. Use this information to clone the repo. At this point you should will have a subdirectory named something like `pclab`. If this is on a shared machine, you probably want to mark the directory as private (`chmod 700 pclab`).

Within the clone repository, there will be two implementation directories each of which contains a copy of the same starter code:

- `atomics.[ch]`: A small (and incomplete) set of primitives for concurrency control that are built on top of hardware mechanisms.
- `ring.[ch]`: A ring buffer implementation that has no concurrency control and thus will not work correctly but will do so very fast.
- `harness.c`: A test harness that evaluates your implementation for correctness and performance.
- `Makefile`: Makefile for the project.
- `README.md`: More information.
- `config.h`: Configuration information including DEBUG printing.

Please be sure to read the `README` file.

To compile the lab, with an implementation just run `make`. This will build the program `harness` (the test harness). `harness` has numerous options, which you can see by running it, but here is a simple invocation:

```
$ ./harness 2 4 16 1024
```

This will create an environment in which there are 2 producer threads feeding 4 consumer threads using a 16 element queue, and then it will operate it for 1024 uses (the producers will push 1024 elements onto the queue, and the consumers will pull 1024 elements from it). After everything is done, `harness` will check for correctness and also tell you the throughput.

Note that, out of the box, there is no synchronization at all and thus the code has numerous race conditions. As a consequence, harness will indicate failure, unless you are very lucky. Harness may even segfault due to its race conditions.

The `harness.c` and `ring.[ch]` makes use of the macro `DEBUG` for debugging output. It is important to note that when you do performance testing, this macro needs to be disabled so that no debug output occurs. You can disable debug printing by setting `#DEFINE DEBUG_OUTPUT 0` in `config.h`. It may seem like printing things out is a fast operation, but, in fact, it's glacial and can severely reduce the throughput you see here. Make sure that you do not add `printf` statements, which won't be disabled, or else your performance could be severely impacted. Only use `DEBUG` statements.

3 Ring buffers

A ring buffer is a fixed size queue that connects one or more producers with one or more consumers. In this lab, the elements in the queue are void pointers (`void*`), meaning that anything can be pushed into the queue by reference. You can consult `ring.h` to see the specific details of the interface required of a ring buffer for this lab, but here are the core operations:

- Push: This pushes one element into the queue, waiting until it is possible to do so.
- Try Push: This pushes one element into the queue, if possible. If not possible, because the queue is full, it returns immediately.
- Pull: This pulls one element from the queue, waiting until it is possible to do so.
- Try Pull: This pulls one element from the queue, if possible. If not possible, because the queue is empty, it returns immediately.

As you might guess, producers use Push and Try Push, while consumers use Pull and Try Pull. Note that the default implementations do not check if there is room in the queue before pushing or items in the queue before pulling. You'll do that when implementing waiting.

4 Task 0: Run the code, including in gdb

Get it, build it, run it. Make sure you have a sense for how it works and what is going on.

Run it again in `gdb`. Learn about `gdb`'s support for threads and signal handlers. Note that `info threads` will show you the threads in the program, while `thread 3` will switch to thread 3 of the program. Breakpoints and watchpoints apply in all threads and signal handlers.

Note that all the `DEBUG` statements print to `STDERR`. Which means that if you are attempting to capture the output of running `harness`, it won't work properly. To save the output to a file, you'll also need to redirect `STDERR` to `STDOUT`.

For example, in the default tcsh shell you can do:

```
$ ./harness 2 4 16 1024 >& OUTPUT.txt
```

Or in a bash shell (or fish from CS211) you can do:

```
$ ./harness 2 4 16 1024 &> OUTPUT.txt
```

Yes, it is indeed quite frustrating that they are arbitrarily different.

5 Task 1: Implement a mutex

Start off in the `mutex_implementation/` directory.

Your overall job in this lab is to make the ring buffer implementation perform correctly by introducing synchronization as needed. At the same time, your implementation should strive to minimize performance impact. That is, you want to achieve the highest possible throughput, while being correct.

To begin with you need to implement a mutex as we described in class. There are multiple mutex implementations that would do, but we suggest you start with a spinlock. It is both straightforward to implement and performs quite well in multicore scenarios. Take a look in `atomics.h` to see some of the tools you can work with.

You must build your mutex using the atomic instructions provided. For this part of the lab you may not use the `pthread` library synchronization primitives.

6 Task 2: Apply the mutex for synchronization

Use your mutex within `ring.[ch]` to make the four operations described earlier correct under all conditions involving threads. Note that there are two issues here that need to be solved. First, there could be a data race between any two threads that are running. Consider what might happen if two threads attempt to modify the ring buffer concurrently. Second, there is no guarantee about what order producers or consumers might run in. Consider what might happen if all the producers run for many iterations before any consumer runs.

7 Task 3: Consider interrupts for the mutex implementation

Within a kernel, concurrency due to hardware interrupts is unavoidable, and must be dealt with. In some cases, user-level code faces a similar situation. The user-level analog to an interrupt is a signal. The combination of signals and threads at user-level exhibits most of the same special concerns that the combination of interrupts and threads within a kernel does. The `harness.c` code emulates the kernel environment of kernel threads and interrupts using preemptable user threads and signals.

A key issue with interrupts and the producer-consumer problem occurs when an interrupt handler can be a producer or consumer. Consider producers. A producer *thread* can wait to acquire a lock on the queue, and wait for the queue to drain enough to make room for new data. Depending on the synchronization primitive, the way in which it waits may be more or less efficient, but it *can* wait indefinitely. The thread scheduler can assure that other threads can make progress. For example, it can switch to the thread that currently holds the lock, or a consumer thread that will drain the queue.

In contrast, an interrupt handler *cannot* wait indefinitely. On x64 machines, for example, interrupts are disabled on entry to the interrupt handler. Even if the programmer reenables them, the interrupt controller will only allow in interrupts of higher priority than the one currently active. The interrupt handler is also not a thread, and so is not schedulable. In other words, for the duration of the interrupt handler, nothing else will happen on the CPU on which the interrupt is running.

Note also that there is an entirely new opportunity for deadlock when interrupt handlers are considered. If, for example, a thread is holding a simple lock, and then is interrupted by a handler that then needs to acquire the same lock, the handler will wait forever trying to acquire it.

Your next task is to enhance your solution for synchronizing the ring buffer assuming that producers and consumers can run within interrupt handlers. You can create this scenario using a command like this:

```
$ ./harness -i pc -t 100000 2 4 16 1024
```

As before, this indicates 2 producer threads, 4 consumer threads, a 16 element ring, and 1024 operations. In addition, both the producer and consumer threads will see interrupts (`-i pc`), and these will occur at random points in time with an average of 100000 µs apart. The interrupt handlers will themselves also produce and consume items using the Try Push and Try Pull interfaces.

Be warned that debugging from within an interrupt context is not always easy. Notably, `DEBUG()` does not function correctly in a signal handler. The way `printf()` works properly across threads is by using an internal mutex, which means that when used in an interrupt/signal handler it could cause a deadlock!

8 Task 4: Semaphore implementation

For the last part of the lab you should finally work in the `semaphore_implementation/` directory.

Now that you've implemented the basic solution with a mutex, your task is to implement a second solution using the semaphore primitive. This solution should follow the general producer-consumer solution pattern. Three total semaphores should exist: one semaphore used as a mutex, one semaphore used for producers, and one semaphore used for consumers. Your solution should be performant and should also handle interrupts as with your mutex solution.

Unlike the prior tasks, you will not have to implement your own semaphore. Instead you should use the POSIX semaphore implementation provided in `<semaphore.h>`. It includes the following functions:

- `sem_init()` which initializes a semaphore with an initial value.
- `sem_post()` which increments the value and wakes another thread.
- `sem_wait()` which decrements the value and possibly blocks the current thread.
- `sem_trywait()` which checks to see if a decrement would block the thread, and if so returns an error instead.

More details for each of these functions can be found by following the link on their names or generally at `https://man7.org/linux/man-pages/man7/sem_overview.7.html`. The "unnamed" memory-based semaphores are what we will be using.

Two important notes about the semaphore library. First, when initializing the semaphores you should ensure that they are shared between all threads of the process (`pshared` should be zero). Second, be aware that `sem_wait()` can return without succeeding! These spurious wakeups include the occurrence of a signal, which will happen in this application if interrupts are enabled. Always be sure to check the return value of `sem_wait()`, which will return zero on success and non-zero on failure. In the case of a failure, the internal value will not be modified and your code should simply call `sem_wait()` again.

You should, hopefully, find that this solution can be far more efficient than a mutex alone in some cases. Unlike the prior solution, threads in this solution will only contend for the mutex if they are actually able to perform the desired action. So if there are many threads, this solution will perform better. Remember that the overhead of using a semaphore is non-zero, so for simple workloads with few threads, the mutex implementation will still win out.

9 Testing your code

For testing the performance of your implementations, we will be using the following tests as well as a few "secret" ones. We will compare your performance to our relatively naive staff solution and will provide a large range of acceptable values. Remember that correctness is more important than running fast.

When testing, be sure to disable debugging in `config.h`. Otherwise the print statements will slow down your code dramatically.

These commands should take 0.1–20 seconds depending on your implementation and the load on Moore. Be careful reading these commands, as the last parameter varies between ten thousand and a million depending on the test case.

Separate cores This test places each producer and each consumer on their own cores to consider communication between threads. In this version the 2 producer threads will run on CPUs 0 and 1, while the 4 consumer threads will run on CPUs 2 through 5.

```
$ ./harness -p 2 -c 4 2 4 16 1024
```

Many to many Creates many producers and many consumers.

```
$ ./harness 100 100 16 100000
```

One to one Creates only one producer and one consumer. Also limits the ring buffer to a single slot.

```
$ ./harness 1 1 1 1000000
```

One to many (Black Friday) Creates one producer but many consumers. The producer is placed on one core and all consumers share one other core.

```
$ ./harness -p 1 -c 1 1 200 10 100000
```

Interrupter Causes frequent interrupts during the producer/consumer exchange.

```
$ ./harness -i pc -t 1 2 4 16 10000
```

Halfsies Uses all cores on Moore: half for producers and half for consumers.

```
$ ./harness -p 24 -c 24 24 24 1024 1000000
```

Important note on sharing the server(s) with your fellow students

As you scale up to more threads, interrupts, and CPUs, it is possible for `harness` to consume massive amounts of CPU time, across many or even all CPUs, on the server, slowing everyone down. Even worse, some implementations of some synchronization primitives can cause the *hardware itself* to run very slowly as it works to maintain correctness for the underlying atomic instructions. Some of the testcases/scenarios we may give you to try out can do this, especially with problematic implementation.

This can look like a "runaway process". A runaway process is one that is consuming lots of resources continuously, and never seems to finish. If you have a runaway process, it may affect other students (and yourself) until it's been stopped.

Here are some things you can do to prevent runaway processes:

- Periodically run the `top` command. If you have a `harness` process high on the list and it's there for a long time, you should use the `kill <pid>` command to kill it. If it doesn't want to stop, you can use `kill -9 <pid>`. "`-9`" is like "force quit" in MacOS or Windows. The kernel just nukes the process instead of asking it to stop.
- You might have left `harness` processes running in the background without realizing it. To kill such processes, you can run `killall -9 harness`. This will try to kill every process which is running an executable named `harness`. You may get some error messages, but that's OK—it's just telling you it can't kill processes you don't own (those of other students).
- You can restrict the amount of time that your `harness` process is allowed to run. To do this, use the following.

  ```
  server> timeout 10 ./harness ..
  ```

  This will stop (kill) your `harness` command after 10 seconds.
- You can restrict the total amount of CPU time that your `harness` process is allowed to run. To do this, use the following. This requires you to be using the bash shell or similar (not tcsh):

  ```
  server> ulimit -t 10 # set limit to 10 second of CPU time
  server> ./harness ..
  ```

  After 10 seconds of CPU time are consumed by `harness`, it will be killed. `ulimit` applies to all child processes of the process in which it is run (your shell), so **be careful not to run something important, like your editor, in the same shell as you've run** `ulimit` If you do, it will *also* be killed after it accumulates 10 seconds of CPU time.
- You can run your program at a lower priority. To do this:

  ```
  server> nice -n +20 ./harness ...
  ```

  (Negative niceness is limited to privileged users.)

10 Grading

Your group should regularly push commits to Github. You also should create a file named `STATUS` in which you regularly document (and push) what is going on, todos, what is working, etc. Your commits are visible to us, but not to anyone else outside of your group. The commits that we see up to deadline will constitute your hand-in of the code. The `STATUS` file should, at that point, clearly document that state of your lab (what works, what doesn't, etc). **Make sure the `STATUS` file includes the names and NetIDs of everyone working on the project.**

We will test your code on the class server (moore) using similar commands to those given above, but with different parameters. This will constitute correctness. In each implementation, we will replace `harness.c`, `config.h`, and `Makefile` with their default initial versions as included in your starter code. Be sure to only make modifications to `ring.[ch]` and `atomic.[ch]`. We will also disable debug printing when testing your code. Make sure that you do not add `printf()` statements, which won't be disabled, or else your performance could be severely impacted.

The breakdown in score will be as follows:

- 15% Task 1—Functional and sensible implementation of a mutex synchronization primitive.
- 30% Task 2—Mutex implementation of ring buffer concurrency that passes concurrency tests that only involve threads.
- 25% Task 3—Mutex implementation of ring buffer concurrency that passes concurrency tests that use both threads and interrupts.
- 30% Task 4—Semaphore implementation of ring buffer concurrency that passes concurrency tests that use both threads and interrupts.

Reasonable performance is expected, but correctness is essential.