CS 343 Operating Systems, Fall 2024
Producer-Consumer Lab: Concurrency Control

Contents

1 Introduction **2**

2 Setup **2**

3 Ring buffers **3**

4 Task 0: Run the code, including in gdb **3**

5 Synchronization primitive concepts **4**

6 Task 1: Implement a spinlock **5**

7 Task 2: Apply your spinlock for synchronization **6**

8 Task 3: Consider interrupts for your spinlock implementation **6**

9 Task 4: Repeat tasks 2+3 with a mutex **7**

10 Task 5: Repeat tasks 2+3 with semaphores **7**

11 Testing your code **8**

12 Grading **10**

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 Piazza. 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 three 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. Please read this for more details!
- `config.h`: Configuration information including DEBUG printing.

You should only make modifications to `ring.[ch]` and `atomics.[ch]`, as we will use the default `harness.c`, `config.h`, and `Makefile` when grading.

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.[1] 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.

[1] You may think that printing something out is a quick thing to do, but while it is quick to you, as a human, it actually takes an eternity compared to the timescales at which we will work in this lab. Printing something out requires an algorithm to convert to a human-readable representation, buffering that, and then calling the kernel to actually do output.

5 Synchronization primitive concepts

You will implement concurrency control with three different approaches: spinlocks, mutexes, and semaphores. For the mutex and semaphore approaches, you will use library implementations. The trick will be to use them correctly. For the spinlock approach, you will build your own (very simple) spinlock implementation and then use it correctly. We will present the three approaches in the order given above because, for this problem, a simple spinlock is very easy to implement and use, your mutex solution will look like a variant of your spinlock solution, and your semaphore solution will be more involved.

A spinlock is an extremely simple concept: it is either locked or not, and when multiple threads try to lock it at once, only one wins; the other threads wait, spinning (repeatedly trying to lock the spinlock until they succeed). A spinlock makes direct use of hardware atomic instructions that operate over shared memory. *A critical point is that the kernel and its scheduler are not involved in the implementation of the spinlock.* This is both an advantage and disadvantage. The advantage is that if a thread doesn't typically have to wait for long to lock the spinlock, then we avoid an expensive interaction with the kernel/scheduler—the lock operation involves only a few cycles. The disadvantage is that if a thread has to wait for a long time, it does so in a very inefficient manner (spinning). This is particularly problematic if the threads are on the same hardware CPU: Suppose thread A successfully locks the spinlock, then the scheduler context-switches to thread B, which tries to lock the spinlock. Thread B will now spin uselessly until the scheduler happens to context-switch back to A and A unlocks the spinlock. Later, the scheduler will happen to context-switch back to B, which will then be able to lock the spinlock.[2] Recall that the scheduler is unaware of the spinlock *by design*.

Spinlocks are extremely widely used in kernels, database engines, and parallel language implementations that target multicore computers (pretty much all computers outside of small embedded systems). They are sometimes combined in a kernel with the ability to temporarily disable interrupts on the current hardware CPU. Disabling interrupts stops preemption on the current hardware CPU, while locking the spinlock stops code on other CPUs from doing so (until we unlock it).

More complex/abstract synchronization primitives are often built on top of spinlocks within the kernel. It is hard to beat using hardware primitives at the lowest level of abstraction. Spinlocks build on top of the hardware's cache coherence and consistency models. There are *numerous* variants of spinlocks (indeed, a whole research cottage industry!) beyond the basic spinlock you'll build that optimize for all sorts of conditions, including efficiency, fairness, etc. You might consider trying alternative spinlock implementations on your own.

A mutex, which is described in detail in your reading, looks very similar to a spinlock (it has the same concept of locking and unlocking), but it also interacts with the kernel/scheduler. The basic idea is that when a mutex cannot be immediately locked, the kernel is invoked. Let's say thread A currently has the mutex locked, but thread B is attempting to lock it. The scheduler puts B to sleep waiting on the mutex, and switches to some other thread.[3] When A eventually runs and unlocks the mutex, this invokes the kernel. The kernel then "wakes up" B and tells it it is has successfully locked the mutex.

In contrast to spinlocks, the disadvantage with mutexes is that locks and unlocks now involve a trip

[2] By the way, if, in this example, B was an interrupt handler, then B would spin *forever* because interrupt handlers are not preemptable by threads; this is a form of deadlock. If this were to happen in a kernel, the *watchdog timer* would likely fire, leading to a kernel panic.

[3] Putting a thread to sleep just means that it is no longer in the set of threads the scheduler will consider for the CPU. Waiting on the mutex means the thread is put into the set of threads waiting on a unlock of the mutex. Typically, these "sets" are queues, with the queue of threads waiting on the CPU called the "run queue", while the queue of threads waiting on some event is called the "wait queue [for that event]".

through the kernel/scheduler, which makes the "fast" case (an "uncontended lock" or a situation in which the lock is held for only a tiny period of time) much slower. It also considerably increases the time taken from when one thread unlocks ("releases the lock") and another thread locks ("acquires the lock"). The huge advantage of mutexes is that waiting to lock is now very efficient—instead of spinning, the thread attempting to lock the mutex is not running at all, leaving room for the thread that currently has the mutex locked to run (and unlock it!), or for an unrelated thread to run and make better use of the CPU than just spinning.[4] Usually, we think about efficiency in terms of getting useful work done, but it's important to note that this is tied to energy efficiency as well. Spinning is among the most energy-consuming things you can make the hardware do. If the code on your phone, say, does a lot of spinning, the result would be a very hot phone which is just quickly eating through battery charge to warm your pocket.

A semaphore provides a more abstract synchronization primitive that is often very useful when doing concurrency control that involves counting or signalling. There is much more info in your book on semaphores, which we will not cover here. The implementation of semaphores you will use in this lab involves interaction with the kernel/scheduler, similar to mutexes, and thus has similar advantages/disadvantages from a performance perspective. The important idea with the semaphore abstraction might be a cleaner match to a producer-consumer concurrency control problem. Consider that a producer pushes to the ring, it can signal to consumers that a new item is available, while when a consumer pulls from the queue, it can signal to producers that a new place to put an item is available.

The line between synchronization primitives that do not involve the kernel/scheduler (e.g., spinlocks) and those that do (e.g, mutex, semaphore, etc) is a bit blurry in a modern kernel. The mutex and semaphore implementations you will use are built on a kernel feature called *futexes* that make it possible to build synchronization primitives that can avoid the kernel/scheduler when possible.

Modern software also makes use of lock-free/wait-free data structures. Here, the basic idea is to use hardware atomic instructions to directly implement the concurrent data structure (the ring in our case) so that no separate locking of any kind is needed. That is, we make concurrency control a part of the data structure design instead of adding it afterwards. An important example in most kernels are "read-copy-update" mechanisms, which basically use the atomic instructions to version-control the data structure. You don't need to worry about these for this lab.

6 Task 1: Implement a spinlock

Start off in the `spinlock_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 simple 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 spinlock using the atomic instructions provided. For this part of the lab you may not use the `pthread` or other library synchronization primitives.

[4] If all threads are currently waiting on events, then the scheduler runs the "idle thread", which uses a special hardware instruction ("halt") to efficiently wait for an interrupt (interrupts are the source of those events, ultimately)

7 Task 2: Apply your spinlock 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.

8 Task 3: Consider interrupts for your spinlock 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 or spinlock, which means that when used in an interrupt/signal handler it could cause a deadlock!

Your successful interrupt rate is unlikely to reach 100%. However, a good solution will have at least *some* successes given enough operations. We will be grading you on correctness, not performance.

9 Task 4: Repeat tasks 2+3 with a mutex

For this part of the lab, work in the `mutex_implementation/` directory.

In this task, you will use the `pthread` library implementation of mutexes instead of your implementation of spinlocks to achieve correct concurrency control for the ring. Your code must be performant and must handle interrupts.

`pthread` is huge, but don't let it daunt you. It's huge because `pthread` (POSIX Thread) is a standard for writing multithreaded programs that is implemented on pretty much all platforms (certainly Linux, Windows, MacOS, ...). As a standard, it's comprehensive, and is also designed to be implementable in numerous ways even on unusual platforms.

The `pthread` mutex is provided in `<pthread.h>`. It includes the following functions:

- `pthread_mutex_init()` initializes a mutex as unlocked.
- `pthread_mutex_unlock()` unlocks a mutex and wakes up a waiting thread (if any).
- `pthread_mutex_lock()` attempts to lock the mutex and puts the calling thread to sleep if this cannot be done now.
- `pthread_mutex_trylock()` attempts to lock the mutex and returns an error if this cannot be done now.

Two important notes about `pthread` mutexes. First, when you initialize a mutex, you supply a mutex attribute set. To start just use the default one (pass in NULL). Second, all of the mutex functions return an int. In standard Unix fashion, a zero return value means success, while a nonzero value means some sort of failure has occurred. It is important that you check the return values. This is also how `pthread_mutex_trylock()` indicates the the mutex could not be locked.

10 Task 5: Repeat tasks 2+3 with semaphores

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

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

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).[5] 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 spinlock or mutex alone in some cases. Unlike the prior solutions, 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 spinlock or mutex implementations will still win out.

11 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 usually take 0.1–20 seconds for most of the implementations depending on the load on the class server. Be careful reading these commands, as the last parameter varies between ten thousand and a million depending on the test case. Some implementations may take considerably longer than the expected 20 seconds for some of these tests. If you're finding that one is taking an extremely long time, consider the number of producers/consumers and the implementation to determine whether this is expected or not.

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, such that there are several of each on every CPU.

```
$ ./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
```

[5]The reason behind names, pshared, etc is to allow semaphores to work across processes, not just across threads within one process. This is a feature that we are not using in this lab.

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 seconds 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.)

12 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.**

You will also need to submit your files to Gradescope. To do so, in the root of the repository run:

```
server> ./submit
```

That will upload the `ring.[ch]` and `atomics.[ch]` files from each implementation to Gradescope along with your `STATUS` file. After you submit, you'll need to mark your group members on your submission. Unfortunately, you will need to do this each time you submit.

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 `atomics.[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 spinlock synchronization primitive.
- 25% Task 2—Spinlock implementation of ring buffer concurrency that passes concurrency tests that only involve threads.
- 20% Task 3—Spinlock implementation of ring buffer concurrency that passes concurrency tests that use both threads and interrupts.
- 10% Task 4—Mutex implementation of ring buffer concurrency that passes concurrency tests that use both threads and interrupts.
- 30% Task 5—Semaphore implementation of ring buffer concurrency that passes concurrency tests that use both threads and interrupts.

Reasonable performance is expected, but correctness is essential.