CS 343 Operating Systems, Winter 2020
Producer-Consumer Lab: Concurrency Control

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(s). We will describe the details of how to access the lab repo via github classroom in lecture and on the discussion group. Use this information to clone the repo. At this point you should will have a subdirectory named `pclab`. If this is on a shared machine, you probably want to mark the directory as private (`chmod 700 pclab`).

Looking in your `pclab` directory, you'll see at least the following files:

- `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 do so very fast.
- `harness.c`: A test harness that evaluates your implementation for correctness and performance.
- `Makefile`: Makefile for the project.
- `README`: More information.
- `config.h`: Configuration information you can ignore.

Please be sure to read the `README` file.

To compile the lab, 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. 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.

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, 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.

4 Task 0: Run the code, including in gdb

Get it, make sure `DEBUG` is turned on, build it, run it. 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.

5 Task 1: Build synchronization primitives

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 synchronization 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 build or select to use a synchronization primitive from among the various ones described in class. We suggest you build a spinlock. Take a look in `atomics.h` to see some of the tools you can work with. You are welcome to use any other primitives. You can also build synchronization primitives on top of `pthread` library synchronization primitives (e.g. `pthread_mutex_t`, `pthread_cond_t`, etc.) or Linux-level primitives (e.g. `futex`).

6 Task 2: Apply your synchronization primitives

Use your synchronization primitives within `ring.[ch]` to make the four operations described earlier correct under all conditions involving threads.

7 Task 3: Consider interrupts

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` us apart. The interrupt handlers will themselves also produce and consume items using the Try Push and Try Pull interfaces.

8 Task 4: Enhance performance for simultaneous threads

A common use case for producer-consumer is to allow threads running on separate CPUs to communicate efficiently. In this scenario, we know the producer and consumer threads are running simultaneously. You can create this scenario using a command like this:

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

In this version of our running sample command, the 2 producer threads will run on CPUs 0 and 1, while the 4 consumer threads will run on CPUs 2 through 5.

How can you revisit synchronization to make such a scenario have higher throughput?

9 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).

We will test your code on the class server(s) using similar commands to those given above, but with different parameters. This will constitute correctness.

The breakdown in score will be as follows:

- 20% Task 1—Functional and sensible implementation of a synchronization primitive.
- 30% Task 2—Sensible implementation of ring buffer concurrency using your primitive that passes concurrency tests that only involve threads.
- 30% Task 3—Sensible implementation of ring buffer concurrency using your primitive that passes concurrency tests that use both threads and interrupts.
- 20% Task 4—Good faith effort to try to specialize for the simultaneous threads scenario.

Reasonable performance is expected, but correctness is essential. We will provide a place for students to report their performance numbers, for those students who would like to have a friendly competition.

10 Extra credit

We will allow up to 20% extra credit in this lab. If you would like to do extra credit, please complete the main part of the lab first, then reach out to the instructor and TAs with a plan. Some possible extra credit concepts are the following:

- Implement a second form of synchronization and compare. For example, you might also implement a ticket lock or MCS lock.
- Specialize your synchronization for single-producer, single-consumer, single-producer, multiple-consumer, and/or multiple-producer/single-consumer. Can you make these special cases of the producer-consumer problem faster?
- Implement a lock-free scheme for the ring buffer. A lock-free data structure does not use synchronization primitives. Instead, it directly uses atomic primitives to manipulate the data structure in such a way that no races exist. Even more interestingly, there exist wait-free data structures which can guarantee that no computation is ever blocked.
- Build a synchronized, linked-list-based producer-consumer queue.