Assignment 4: Synchronization

Due Date: Friday, November 21st

Overview

Have a look at the following demo program. Does this program require explicit synchronization? There's a shared variable, `buf`, accessed both by the thread executing `read_file` and the main thread. However, there are no danger of any concurrent accesses, since our system is running cooperatively on a single kernel thread, and neither thread calls `yield` during its operation.

Thus `buf` is implicitly protected by mutual exclusion. So, what output can we expect from this program? This depends on the implementation of `thread_fork`.

If `thread_fork` causes the child thread to run immediately (as in our implementation), then `read_file` reads random bytes into the buffer, finishes, and `yield`s back to main, which then calls `zero_buf`, which fills the buffer with all zeroes. When the contents of the buffer are printed, it will be all zeroes.

If `thread_fork` enqueues the child thread, but allows the calling thread to continue executing, then `zero_buf` will run first and fill the buffer with all zeroes. `scheduler_end` will then run, which will `yield` to the child thread, which will fill the buffer with random bytes. When the contents of the buffer are printed, it will be all random bytes.

The important thing to note is that the order is deterministic based on the implementation of `thread_fork`.

Now have a look at this very similar program, where the call to `read` is replaced with `read_wrap`.

Assuming that `thread_fork` causes the child thread to run immediately, then `read_wrap` will run first and `yield` after starting the asynchronous read. Then `zero_buf` will run. `zero_buf` doesn't yield, so it should run to completion, and after it completes we should expect that the buffer is filled with all zeroes.

Next, `scheduler_end` will run, which will `yield` back and forth with `read_wrap` until its read is complete. Once the read is complete, the buffer should be filled with all random bytes. So we should expect the program to print out a buffer filled with entirely random bytes.

Go ahead and compile and link `demo2.c` with your scheduler. What do you notice? Here is an section of output from a sample run of `demo2.c`. Contrary to our expectations, the zeroes are overlapped with the random bytes! This is clearly a concurrent access, and the data is corrupted as a result.

What's going on? How did our scheduler achieve true concurrency on a single kernel thread? Well, recall that the `glibc` implementation of `aio_read` works by forking off a kernel thread to complete the read, so our program actually had two kernel threads executing on its behalf, not one -- and they were both accessing the same memory at the same time.

It's worth noting that this behavior depended on my implementation of `read_wrap`, where I supplied the `buf` parameter passed in to `read_wrap` directly to `aio_read`. If instead I had created a buffer local to `read_wrap` and supplied that to `aio_read`, then copied the result back to `buf` after the read completed, this would not have happened. However, this requires performing a potentially expensive copy.

It's better to optimize for the common case, where there are no concurrent accesses, and pass in `buf` directly. However, invalidates our earlier assertion that we could achieve mutual exclusion simply by not calling `yield`. Futhermore, in the next assignment we will add explicit concurrency by introducing a second kernel thread. For both of these reasons our system needs a more explicit mechanism for controlling mutual exclusion: locking.

Design Choices

Synchronization primitives

A fundamental choice to make in this scenario is what kind of synchronization primitives we want to implement. We know we need to achieve mutual exclusion. Condition variables turn out to be quite useful in many situations as well.

You can do quite a bit with semaphores, as we saw in Dijkstra's paper on the 'THE' system, where their team used semaphores to implement both mutex locks and condition variables.

Here we'll design mutex locks and condition variables as separate primitives. In your implementation, however, it's worth nothing that condition variables can be implemented in terms of mutex locks, by initializing the lock to "held".

Blocking vs. Busy-waiting (revisited)

The above discussion implies that we'll want our mutex locks to cause threads to truly block rather than spin, but in light of the similar discussion in last assignment where we decided in favor of spinning instead of blocking, it's worth taking a moment to think about using spinlocks in this situation.

First of all, as we've discussed in class, spin-waiting on a uniprocessor (or a user-level thread system with a single kernel thread) is always wasted work, because there's no chance of progress -- the only thing that can wake up the spinning thread is the lock holder, which is another user level thread, and which by definition is not running whenever the spinning thread is running.

This is in contrast to polling for I/O completion, where the event that will cause our thread to wake up is not the execution of another user-level thread, but instead an external event whose completion time is unknown. Another way to put this is that I/O completion is an asynchronous event and must be polled for (or signalled), while unlocking a mutex is a synchronous event (and indeed is even visible in the code).

Since we know precisely when a mutex will be unlocked (when a thread calls `unlock`), we don't need to poll for completion, and we don't need any fancy data structures to keep track of outstanding requests, other than the waiting queue attached to the mutex variable itself.

Consequences of Blocking

In the absence of true blocking, all previous versions of our scheduler have maintained the invariant that there is always at least one runnable thread. Thus, as long as the current thread enqueues itself on the ready list before dequeuing a thread, the ready list will never run empty. An empty ready list is a very undesirable situation, because control flow has nowhere sane to go!

Once we introduce true blocking, we also introduce the possibility that all threads might become blocked, which could result in the ready list becoming empty. Consider the case where there is one runnable thread, and it blocks -- it would not enqueue itself on the ready list, thus there would be nothing to dequeue (and nothing to switch to). In our implementation, this would cause our program to crash.

You might notice, however, that this situation is a fatal error with a precise cause: since our system does not feature asynchronous software interrupts (signals), if a thread blocks and there are no other runnable threads, then there are no events that can come to that thread's "rescue" and cause it to unblock. This is an unrecoverable deadlock.

If our system did use signals (for instance to deal with I/O completion events instead of polling), then we would need to introduce an "idle thread" whose job it was to *never* block: it would simply sit and yield forever until the ready list became empty *and* there were no pending asynchronous interrupts. At this point, the idle thread could signal the main thread that the system was idle, so it could continue with any code on the other side of `scheduler_end`. If the main thread was blocked, the idle thread would report a deadlock and/or exit.

Implementation

1. Modify your `yield` routine to prevent a thread whose status is `BLOCKED` from enqueuing itself on the ready list.

2. Design a mutex lock that provides mutual exclusion between the lock and unlock primitives by blocking any thread that attempts to lock a mutex that is currently held.

   If you want some ideas, look over these figures that describe a feasible design for a blocking mutex lock.

3. Code a data structure, `struct mutex` that represents your lock, and three functions:

   - `void mutex_init(struct mutex *)`
   - `void mutex_lock(struct mutex *)`
   - `void mutex_unlock(struct mutex *)`

   `mutex_init` should initialize all fields of `struct mutex`. `mutex_lock` should attempt to acquire the lock, and block the calling thread if the lock is already held. `mutex_unlock` should at least wake up a thread waiting for the lock. It may do other things as well depending on the design you choose.

4. Design a condition variable that has MESA semantics. You may design it in terms of your mutex lock, or on its own.

5. Code a data structure `struct condition`, that represents your condition variable, and four functions:

   - `void condition_init(struct condition *)`
   - `void condition_wait(struct condition *)`
   - `void condition_signal(struct condition *)`
   - `void condition_broadcast(struct condition *)`

   `condition_init` should initialize all fields of `struct condition`. `condition_wait` should cause a thread to wait on the condition, and `condition_signal` should wake up a waiting thread (but not suspend the current thread, as per MESA semantics). `condition_broadcast` should signal all waiting threads.

Note: a reasonable place to place these routines is in `scheduler.c`, but feel free to make a new file, `lock.c` if you so choose.

Testing

1. Design a test to verify the semantics of your mutex lock, namely that a thread holding the lock has exclusive access to the critical section protected by the lock, and that all blocked threads eventually wake up and have a chance to run in the critical section.

   Since we still only have one kernel thread, note that your test will have to "force" a race condition by yielding in the critical section. For instance, if our test was having multiple threads "simultaneously" increment a shared counter, we would have to write code like:

   ```
   mutex_lock(&m);
   int temp = shared_counter;
   yield();
   shared_counter = temp + 1;
   mutex_unlock(&m);
   ```

   Your test should involve several threads (at least five) accessing the same shared variable, several times each.

2. Design a test that uses your condition variables. You are free to attempt one of the "classic" synchronization problems (dining philosophers, producer/consumer, sleeping barber, etc), or design your own. Cite any code that you pull from other sources.

Discussion

Our mutex locks works because the lock variables themselves are implicitly protected by mutual exclusion -- the `lock` and `unlock` operations are "atomic" because they do not yield and should not be affected by asynchronous reading operations.

Your next assignment will be to add a second kernel thread to your user-level thread scheduler, to allow two user-level threads to run in parallel. How will this affect the operation of your blocking mutex lock? What additional mechanisms might we need in order to ensure that our programs are safe?

What To Hand In

If you are working with a partner, please prepare just one submission for the both of you. Make sure your (and your partner's) names are included in each file you submit.

You should submit:

1. All code, including tests. This should include:

   - `switch.s`
   - `queue.h`

- `queue.c`
- `scheduler.h`
- `scheduler.c`
- `async.c`
- `main.c`
- `lock.c` (you may put your lock functions in `scheduler.c` if you choose)

2. A brief written report, including:

    1. A description of what you did and how you tested it.
    2. Your response to the question in "Discussion", above.

Please submit your code files *as-is*; do not copy them into a Word document or PDF.
Plain text is also preferred for your write-up.
You may wrap your files in an archive such as a .tar file.

Email your submission to the TA at kstew2 at cs.pdx.edu on or before the due date. The subject line should be "CS533 Assignment 4".

Need Help?

If you have any questions or concerns, or want clarification, feel free to contact the TA by coming to office hours or sending an email.

You may also send an email to the class mailing list, but please pose discussion questions only; **do not** email any code to the class mailing list! (It is okay to send code directly to the TA, however).