Assignment 1: Userspace Multithreading

September 20, 2021

The purpose of this assignment is getting acquaintained with userspace multithreading and synchronization primitives. This means working with threads, spinlocks, mutexes, and condition variables.

1 Question 0

This introductory question is about using atomic instructions to create a spinlock. The starter code has a set of threads, all incrementing one common counter in memory. The error in the starter code is that the act of incrementing is not atomic, that is, it does not finish in one instruction. Below are the C code and the equivalent x86 assembly code that is actually executed:

```
counter += 1;

movq rax, [counter]
addq rax, 1
movq [counter], rax
```

If multiple threads execute the snippet above concurrently then some increments to the counter are effectively lost. The threads update the counter with a value one more than the one they read. If they read the same value from the memory location, one thread is bound to overwrite the other's update.

Let's see an example of such an error. supppose that two threads both read value 5 from the memory location of the counter. The first thread increments register `rax` to 6, then stores the value back into memory. The other thread *also* increments its own `rax` to 6, storing the value in the memory location. The variable has been updated from 5 to 6 to 6 again, instead of going from 5 to 6 then to 7 as expected.

To solve this problem we create *critical sections* that can only be executed by a single thread at a time. Only one thread can be in such a section at any given time, preventing race conditions like the concurrent update problem above. To other threads, code in a critical section looks like it is run atomically.

In our case we create critical sections using *spinlocks*, simple state machines that transition between their states atomically. A spinlock is a variable that can have a value of 0 (unlocked), or 1 (locked). A thread is able to 'get the lock' if it succeeds in changing the value from 0 to 1. It releases the lock by setting the value by setting the value from 1 to 0. The challenge in creating such a primitive is to atomically execute the following pseudocode:

```
void spinlock_lock(int *lock)
{
    while ( *lock != 0 )
        ;
    *lock = 1;
}

void spinlock_unlock(int *lock)
{
    *lock = 0;
}
```

For this assignment we will use *atomic instructions* to implement spinlocks. We will more specifically use `__atomic_test_and_set` and `__atomic_clear` (see `https://gcc.gnu.org/onlinedocs/gcc/\_005f\_005fatomic-Builtins.html` for details). Use these two functions to implement `spinlock_lock` and `spinlock_unlock`.

2 Question 1

This question introduces threads and the pthread API. Each process can have multiple threads. Threads have their own set of CPU registers and stack segments, but share the code and heap segments with each other. Since each thread runs in a separate CPU, multithreading lets us use more than one cores at a time to speed up computation.

The API used by C to create threads is the POSIX threads (pthreads) API. The main three functions this API provides are:

- `pthread_create`: This call creates a new thread. The thread starts executing from the function given as an argument.
- `pthread_exit`: This call destroys the thread, but not the process. The exiting thread may pass a pointer to a variable in the heap as an exit value, to be read by another thread using `pthread_join` (see below).
- `pthread_join`: Wait for another thread in the same process to be done.

A common pattern with multithreading is to spawn multiple workers using `pthread_create` from an initial thread, then wait for them to be all done by repeatedly calling `pthread_join`. The initial thread communicates to each worker what data it needs to process by passing to each one different arguments; this makes it very easy to parallelize tasks where each thread needs to only work on part of the data at a time.

For this question we parallelize exactly this kind of workload. We are given a 'library' of news articles, each of which is composed of a sequence of words.

The task we need to parallelize is counting the number of times a certain word occurs in all articles. To count the number of occurences we traverse the library one article at a time, and one word at a time. We compare each word against the one we are looking for, and increment a counter if they are identical.

The task is trivially parallelizable because we can process each article separately. That means that we can create an arbitrary number of threads, split the work between them, and gather all the individual counters using `pthread_join`. There is no need for locking since each thread has its own input.

For this exercise fill in the function in the file `map.c`. The solution needs to produce identical results with the single threaded version in `main.c`, and provide considerable speedup. Please note that the number of libraries provided as input should be significantly larger than 1 (e.g., 100), so that parallelizing the task leads to performance gains.

3 Question 2

This question introduces mutexes and condition variables, the two main synchronization primitives of the pthread library. Mutexes function like spinlocks in that they provide mutual exclusion, but are more efficient: A thread waiting on a mutex will be scheduled out of the processor, leaving it free for another thread that can actually do work. A thread waiting on a spinlock on the other hand will continuously try to take the lock by continuously executing the `lock` instruction until it receives it. This leads to wasted CPU cycles if the thread already in the critical section holds it for a long time.

Condition variables are used to deal with race conditions that arise when multiple threads attempt to grab the same mutex. For example, assume that we have a counter that threads either increment or decrement, and which must stay above 0. If a thread wants to decrement the counter but it is at 0, it has to wait until the counter is incremented by another thread before decrementing it. We protect a counter by a mutex both for reads and writes.

If a thread grabs the mutex to decrement the counter and finds it is at 0, it has to wait for another thread to increment it. The derementing thread, however, is holding the mutex and thus preventing any modification to the counter. The decrementing thread must thus leave the lock and wait until an increment happens.

The issue that arises then is how long to wait. A naive solution would be to use `sleep` calls like in the code below:

```
while (true) {
    lock();
    if (condition == true) {
        /* Leave with the lock taken */
        break;
    }

    /* Else try again later */
    unlock();
    sleep(TIMEOUT);
}
```

This solution, however, has two main weaknesses. If the timeout is too large, the thread sleeps even when it could execute, potentially leading to massive performance penalties for the waiting thread. If the timeout is too small, the thread wakes up too often and wastes CPU time by needlessly trying to execute.

The solution is to use *condition variables*, an API for notifying waiting threads to attempt to take the lock and check if the condition they were testing holds. The waiting thread calls `pthread_cond_wait()`, while the thread that notifies uses `pthread_cond_signal()` or `pthread_cond_broadcast` to wake up one or all waiting threads.

In this exercise we use this API to synchronize between threads. Each thread in the code is a producer or a consumer of a shared resource. The threads enter and exit from the resource at regular intervals. At any point in time, the ratio of producers to consumers must be higher than a given value (e.g., for a ratio of two there can only be 4 consumers present in the resource if 2 or more producers are also present).

Use condition variables to ensure that the number of producers and consumers in the resource leads to a valid ratio. While producers enter freely and consumers exit freely, since their arrival and departure respectively can only raise the ratio, producers that want to exit must check whether their movement will drop the ratio below the allowed value. In that case, they need to wait until a producer enters or a consumer leaves, raising the ratio to allow for the exit. The same holds for consumers entering the resource.