Part Three: Multiple Threads

Implementation

My implementation uses pthreads mutex to avoid data races and ensure safe access to memory. First, I initialize 64 bits of memory to contain the unsigned integer 1. Next, I use pthread_create to create W threads that execute the function worker_process. worker_process runs a loop that retrieves the value stored in memory, increments that value by 1, and stores the incremented value. To ensure memory is being accessed by only one thread at a time, I use pthread_mutex_lock and pthread_mutex_unlock for each iteration of the loop.

Mutex Implementation Discussion

Mutex has the advantage that only a single thread is ever running the critical piece of code at a time, so data races will be avoided with certainty. Implementing mutex is also extremely easy and convenient – the exact lines of code that must be synchronized can be isolated with pthread_mutex_lock.

Some possible disadvantages include potentially unnecessary overhead and lack of prioritization. If multiple threads are attempting to access the same locked code, most of them will waste many cycles simply waiting for the lock instead of executing some other functionality to maximize time. The programmer also has no control over which threads access the critical code first; if some threads are higher priority than others and should access the critical code first, they cannot because whichever thread happens to access first will lock all other threads out.

Difficulties

Having only used fork() before to create threads, pthread was a new discovery – I was unfamiliar with its idiosyncrasies and syntax. At first, I was attempting to use a combination of pthread_mutex and fork() (as recommended by Chat GPT :() which did not work and confused me for a while.

Document at what values of W you receive incorrect results. What is the exact reason the incorrect results occur? Be specific.

I received incorrect results at W = 2 consistently. The incorrect results are occurring because the first thread and the second thread are attempting to read and write to the same memory simultaneously. Either the first thread reads the number at the same time as the second, resulting in both threads updating the number by 1, or the first thread reads the number followed by the second thread reading and writing to the memory before being overwritten by the first thread, or any number of these combinations.

How could you avoid this problem from within kernel space? In other words, how could your memory driver module ensure that user programs get the correct result no matter how many threads are simultaneously accessing it?

The mutex solution is applicable in kernel space as well – if the locks were applied to the correct pieces of code inside the kernel solution, the correct synchronization would be achieved.

What other kinds of unsafe behavior does your module allow to happen? For example, what happens if you close the file descriptor before all worker threads have completed? What happens if a worker thread tries to access a byte of memory outside of the region's bounds? How could you avoid these problems?

If the file descriptor is closed prematurely, the user space threads are still reading/writing from/to the now closed memory. This could lead to the user space accessing memory that has been deallocated or allocated to a new operation by the kernel, causing undefined behavior. A worker thread accessing memory outside the region's bound could overwrite important memory belonging to another operation or cause the kernel to crash. These issues could be avoided with very careful kernel code programming – ensuring every function has the proper protections for everything the user could do.

AI

The Clarity platform wrote about 95% of the code – I had to modify the program to replace everything related to fork with pthread (modifying a code example from claude.ai).