Lecture 22

Topic: Mutual Exclusion

Context: Service Server-06.c

At the end of the last lecture, we were trying to run the server out of memory by requesting multiple processes until the server could not spin off any more. We attempted to solve this by creating threads instead of spinning off processes and then limited the number of threads to 100 by waiting to start new connections when an old one dies. This creates a new problem: We use a polling mechanism which has bad utilization. Solution: Sleep using a condition variable

Mutual Exclusion definition: At most one thread's program counter/instruction pointer can be in some region at a time. This region is called the critical region.

incr.c (in l23 directory): Each thread adds ten million to a shared counter. This counter is stored on the stack.

- Creates four threads, stores thread ID's, waits for each of the threads to exit, print the value of **n** (a variable on the stack)
- digression: pthread_join behaves similarly to waitpid, without the various options/arguments that can be passed to waitpid

Result of the program:

- Four threads adding 10 million each to a number using ++. Result is not consistently 40 million.
- The reason why the result is not consistently 40 million is because the program is incrementing memory. Accesses to memory are not usually guaranteed to be atomic, thus while the increment might seem like 1 command, the CPU likely does the increment in three steps.
    - 1) loads memory value into register, 2) increments register, 3) writes register back to memory.
- Each thread is doing these three operations (potentially running on different cores, so the registers are different). There are no dependencies that force these operations across threads to take place in any particular order.

Let's say we have two cores, T1 and T2 (where T1a loads x into r, T1b increments r, and T1c loads the new value back into r, and T2a, b, and c operate similarly). Then the only two orderings that will produce the expected value are T1a, T1b, T1c, followed by T2a, T2b, T2c, and vice-versa. Any interleaving of the steps leads to an undesired result.

Progress graphs (diagram of an x-y graph):

- Help us reason about 2 threads
- First thread's instruction pointer is on one axis; second thread's instruction pointer is on the other axis
- 6 steps: 1. while(i< 10^7), 2. r = x, 3. ++r, 4. x = r, 5. i ++, 6. }. (r represents a register).

- Individual threads' program counters move around on the progress graph. Each axis is used to mark the step that its respective thread is in (each axis labeled with values 1-6)
- Critical region on this graph corresponds to the region that should never be entered. This is the region where both threads are in steps 2, 3, or 4.
- How can we ensure that critical regions are never entered?

Use a lock!

- What should the lock look like? Should probably be an if statement that checks some memory
- Lock has two methods: acquire and release
- Lock should have the following semantics:
- L = 0;
- Lock(L);
    - while (L == 1)
        - do nothing
    - L = 1;
    - return;

  Unlock(L)
    - L = 0;
- These two functions seem reasonable enough.

Can you write a lock that works only in software? Lamport-Bakery algorithm allows lock/unlock with only memory reads and writes. Uses assumption that memory reads and writes happen in an order.

- This isn't actually true because of how caches work. Reads and writes require exclusive access of the cache line (this is mutual exclusion). The cache line will effectively bounce between cores, but obtaining exclusive access requires coordination between processors, which is very slow. Writing values to variables doesn't require exclusive access. The value of a variable is instead placed in a **store buffer**, and exclusive access is only maintained when the store buffer is flushed.
    - Example: we have two threads, with int x = y = 0.
    - Thread 1: Sets x = 1. Reads y into register r1.
    - Thread 2: Sets y = 1. Reads x into register r2.
    - If we have ordered sequential reads and writes, what values can we get? We can't get 0 into both registers. Because if they happen in an order, either writing x or y happens first. In an ideal world, we would never have both registers be 0.
    - On an x86 machine, we can get both registers to be 0 because things don't happen in an order. Assignments to x and y can be stored in a store buffer and those changes are not written into memory. The reads from memory do not see the changes.

Memory models! (What the machine is allowed to do with loads and stores, the bleeding edge of concurrent code). You feel like a badass.

- The store buffer is flushed during locks or when it is full.

Intel has created a lock. What memory operations need to be atomic to make a lock work?

- Check/Set. need to be able to read and write in one atomic step. This is the core of the functionality we need for a lock to work. Form of exclusive access: access to a cache line.
- Processors are unable to read/write unless they possess exclusive access: so as long as we include the prefix “**lock**”, the reads/writes will run atomically (check assembly code to verify)
- The assembly shows that we need support from the architecture to run correct, concurrent code
    - lock addl ensures that the instructions happen in one atomic instruction
- Using this atomic instruction, increment program (incr.c from l23 repository) works every time. This implementation takes about 10 times more time than the incorrect code (the incorrect code is not locking cache lines)

So how do we build lock/unlock out of the lock primitive we have?

- “_sync_fetch_and_add” is an atomic version of incrementing a variable given to us by gcc
- l = 0
- lock;
    - while (locked_add(++l) > 1)
        - (locked_add(--l)
- unlock;
    - locked_add(--l)
- (note that ++l, --l, and --l are all implemented atomically here)
- Unlock as l =0 is an incorrect mutual exclusion lock because it might cause lock to be set to -1 while one thread tries to lock and another tries to unlock

Can we make a lock that requires only one atomic process in its implementation?

- Atomic instruction that only increments on 0.
- **The compare and exchange instruction** (**cmpxchg**):
- C pseudocode:
- int cmpxchg(int* m, int expected, int desired) {
-   int actual = *m;
-   if (actual == expected)
-     *m = desired;
-   return actual;
- }
- Implementing a lock using cmpxchg:
- l = 0;
- lock;
- while (cmpxchg(&l,0,1) != 0)
    - /* do nothing */

- unlock;
    - l = 0;
- In this implementation, lock only ever has the value 0 or 1. Check/set happen atomically.Cmpxchg is the most general basis for fast, comparable locks in parallel programming.
- How do we build a locked addition function using the cmpxchg interface?
- void lockadd(int * m, int a){
    - int x = * m;
    - while(cmpxchg(m,x,a+x) != x)
        - x = *m;
- }

Storytime:

- Debian SSH disaster: Predictable random number generator. Debian is a vendor of linux. Vendors make changes to linux. OpenSSL, one of the foundational pieces of security code on everyone's computer. Debian removed some uninitialized memory from OpenSSL (valgrind warning on unitialized memory). Unitialized memory had been requested from kernel to contain random data. As a result, if you created an SSH key on a Debian machine, there were only about 10,000 possible SSH keys.

Happy Thanksgiving!