UCB CS 162 - Operating Systems and Systems Programming "Lecture 8 - Synchronization 3- Atomic Instructions, Monitors, Readers/Writers"

Lecture 8 - Synchronization 3: Atomic Instructions, Monitors, Readers/Writers

The previous lock implementation we saw was impractical because it involved a system call every time we wanted to acquire or release a lock. A system call is at least 25x more expensive than a normal function call, so this approach limited our performance severely.

What we need is just to be able to read a value (of the lock), check if it is 0 or 1, and then write to the value atomically. If we removed the need to disable and re-enable interrupts for this operation, we would be able to implement locks at the user level.

Therefore, all architectures provide atomic instructions for this purpose. These instructions can read, modify, and write data atomically, such that they cannot be interleaved. Some of these instructions are -

```
// Return the value stored at address, and store a 1 there
test&set(&address) {
    result = M[address];
    M[address] = 1;
    return result;
}

// Swap "value" with value stored at address
swap(&address, value) {
    temp = M[address];
    M[address] = value;
    value = temp;
}

// If value at address == reg1, store reg2 there and return success, else failure
```

```
compare&swap(&address, reg1, reg2) {
    if (reg1 == M[address]) {
        M[address] = reg2;
        return success;
    }
    else return failure;
}
```

Implementing Locks With test&set

The lecture gives a few ways in which we can implement locks using `test&set()`, but I won't write about them here because they are very intuitive to come up with but result in busy waiting, so we don't use those implementations anyway.

Monitors

Monitors are a concurrent programming paradigm that are, in a way, cleaner and more intuitive than semaphores. A monitor is a lock and zero or more *condition variables* associated with it. A condition variable is just a variable that represents some constraint on the system.

We have the following functions to use with monitors - 1. `Wait(&lock)` - Automatically release the lock and go to sleep 2. `Signal()` - Wake up one waiter, if present 3. `Broadcast()` - Wake up all waiters, if present

For example, let us look at a database with multiple readers and writers. The constraints we have are that the readers can access the database only when there are no active writers or waiting writers, and the writers can only access the database when there are no active readers.

We maintain the following state variables - 1. `int activeReaders = 0` 2. `int waitingReaders = 0` 3. `int activeWriters = 0` 4. `int waitingWriters = 0` 5. `Condition okayToRead = NULL` 6. `Condition okayToWrite = NULL`

Then the code for a reader will look as follows -

```
Reader() {
    // First check self into system
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead,&lock);// Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
```

```
    release(&lock);
    // Perform actual read-only access
    AccessDatabase(ReadOnly);
    // Now, check out of system
    acquire(&lock);
    AR--; // No longer active
    if (AR == 0 && WW > 0) // No other active readers
        cond_signal(&okToWrite);// Wake up one writer
    release(&lock);
}
```