Lecture 23: Threaded Parallelism

Group B Scribe Notes

serviceserver-06.c

- this is a program that serves information about services
- it aims to handle multiple connections in parallel without running the system out of memory
- it does this by using threads instead of processes but running one thread per connection
- it can only make at most 100 threads
- the problem is that it uses polling and that is bad utilization

Mutual Exclusion

At most, one thread's program counter or instruction pointer can be in some region at a time. This region is the *critical region*.

Locking and unlocking give us this important mutual exclusion property.

incr.c

- in this program, we create four threads, and each thread adds 10 million to a shared counter (stored on the stack)
- the four threads run in parallel and then print the value of n
- this operation accesses memory, and operations that access memory are rarely ever atomic (indivisible)

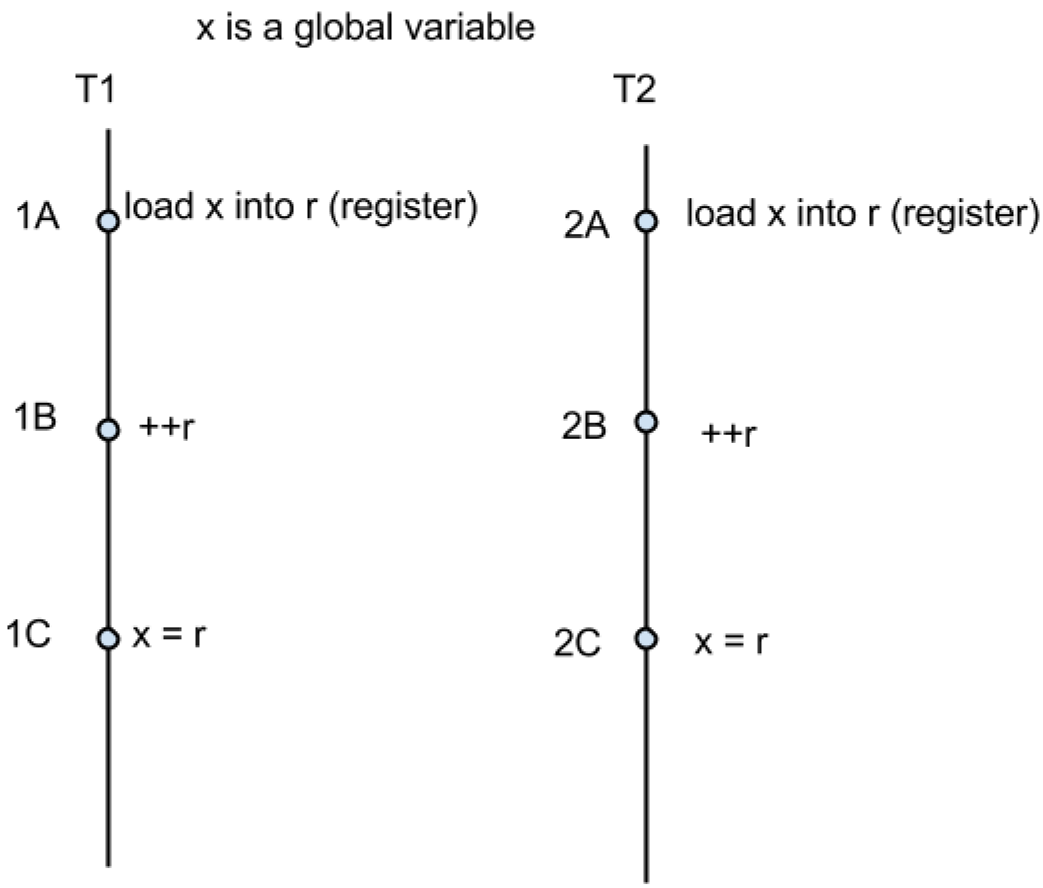
Consider:

++mem;

In assembly, this might be
incl(mem)

But the processor actually

1. loads mem into a register (e.g. %eax)
2. increments the register
3. writes that register value back to mem



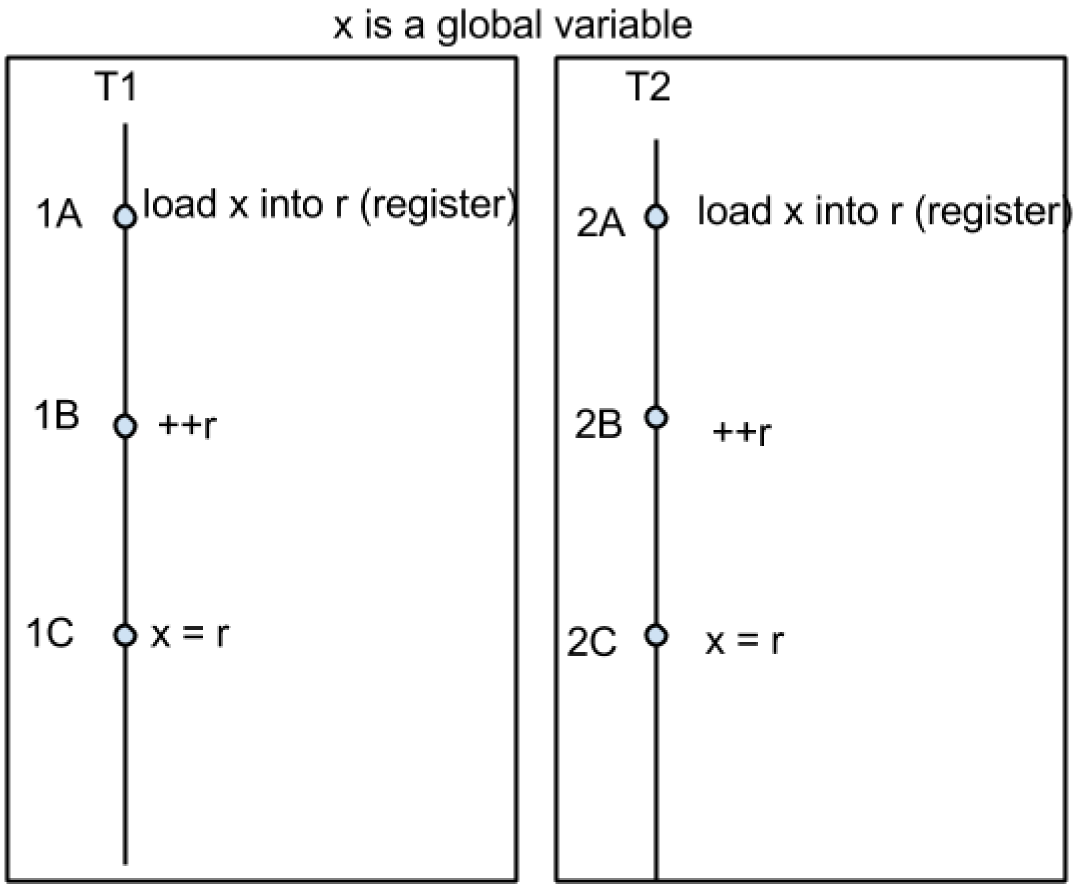
Let's say we had the global variable x, and two threads (T1 and T2). Let's say we set x = 0, and we increment twice. Then we would want x == 2 after the two threads run their instructions. This would require the instructions to run in either of the following orders:

1A, 1B, 1C, 2A, 2B, 2C
2A, 2B, 2C, 1A, 1B, 1C

But the above instructions can happen in any order.

Let's say the instructions ran as 1A, 1B, 2A, 2B, 2C, 1C, then we would have:

1A) $r_1 = 0$
1B) $r_1 = 1$
2A) $r_2 = 0$
2B) $r_2 = 1$
2C) x = 1
1C) x = 1



If these boxes were a critical region, then the only orders we would observe are:

1A, 1B, 1C, 2A, 2B, 2C
2A, 2B, 2C, 1A, 1B, 1C

This is because we would not allow another core to enter into this region until the current one had exited the region.

Progress graphs also allow us to reason about exactly two threads. (The book has good examples on this.)

We can atomically run these instructions with a lock.

A lock is a synchronization object. It has two methods: acquire (lock), and release (unlock). It locks and unlocks a region so that other threads can access it. Only one thread can have a lock on at a region at a time.

Let's say we have a lock z.

```
lock(z)
	if z is not locked
		lock z
		return
	else
		try again

unlock(z)
	mark z as unlocked
```

If we make it so that a locked state is represented by 0 and locked by 1:

```
lock(z)
	while(z == 1); // do nothing
	z = 1;
unlock(z)
	z = 0
```

This has synchronization issues. We need to be able to write a lock that reads and writes in one atomic step. We need to keep the code in critical regions (represented by bold)

z = 0
lock:
    while(**++z** > 1)
        --z;
unlock:
    z = 0;

This does not work because a second thread can unlock z (z = 0) before the lock decrements, causing z == -1.

To fix this:

z = 0
lock:
    while(**++z** > 1)
        --z;
unlock:
    --z;

To make this work with one box (critical region), we use compxchg (*cmpxchg runs atomically*)

```
cmpxchg(int* m, int expected, int desired) {
    int actual = *m;
    if(actual == expected)
        *m = desired;
    return actual;
}
```

Our lock now looks like:

```
lock:
    while(cmpxchg(&z, 0, 1) != 0);

unlock:
    z = 0;   // this works now because the check and increment are atomic and the lock
             // can only be 0 or 1
```

How would we write an add function that uses a lock?

```
void lockadd(int* m, int a) {
    int x = *m;
    while(cmpxchg(m, x, a + x) != x) {
        x = *m;
    }
}
```