Lecture 4: Independent vs. cooperating threads

4.0 Main points

Why do we need to handle cooperating threads?
Atomic operations

4.1 Definitions

Independent threads:

No state shared with other threads
Deterministic -- input state determines result
Reproducible
Scheduling order doesn't matter

Cooperating threads:

Shared state
Non-deterministic
Non-reproducible

Non-reproducibility and non-determinism means that bugs can be intermittent. This makes debugging really hard!

4.2 Why allow cooperating threads?

People cooperate; and computers model people's behavior, so computers at some level have to cooperate!

1. **Share resources/information**
   a. one computer, many users
   b. one bank balance, many tellers
   c. embedded systems (ex: robot control)

2. **Speedup**
   a. overlap I/O and computation
      UNIX file system does read ahead

b. multiprocessors -- chop up program into smaller pieces

3. **Modularity**

chop large problem up into simpler pieces

For example, to do typesetting: ref | grn | tbl | eqn | troff

This makes the system easier to extend; you can write eqn without changing troff

4.3 Some simple concurrent programs

Most of the time, threads are working on separate data, so scheduling order doesn't matter:

```
Thread A                    Thread B
x = 1                       y = 2
```

What about:

```
        initially, y = 12

x = y + 1                   y = y * 2
```

What are the possible values for x after the above? What are the posssible values of x below?

```
x = 1                       x = 2
```

Can't say anything useful about a concurrent program without knowing what are the underlying indivisible operations!

4.4 Atomic operations

Atomic operation: operation always runs to completion, or not at all. Indivisible, can't be stopped in the middle.

On most machines, memory reference and assignment (load and store) of **words**, are atomic.

Many instructions are **not** atomic. For example, on most 32-bit architectures, double precision floating point store is not atomic; it involves two separate memory operations.

4.5 A Larger Concurrent Program Example

Two threads, A and B, compete with each other; one tries to increment a shared counter, the other tries to decrement the counter.

For this example, assume that memory load and memory store are atomic, but incrementing and decrementing are **not** atomic.

```
   Thread A
i = 0
while (i < 10)
  i = i + 1;
print A wins
```

```
   Thread B
i = 0
while (i > -10)
  i = i - 1;
print B wins
```

Questions:

1. Who wins? Could be either.

2. Is it guaranteed that someone wins? Why not?

3. What if both threads have their own CPU, running in parallel at exactly the same speed. Is it guaranteed that it goes on forever?

In fact, if they start at the same time, with A 1/2 an instruction ahead, **B** will win quickly.

4. Could this happen on a uniprocessor?

Yes! Unlikely, but if you depend on it **not** happening, it will happen, and your system will break and it will be very difficult to figure out why.