CS 162 Operating Systems and Systems Programming

Professor: Anthony D. Joseph

Spring 2001

Lecture 5: Independent vs. cooperating threads

5.0 Main points

Thread Creation

Why do we need to handle cooperating threads?

Atomic operations

5.1 Thread creation

Thread "fork" – create a new thread, three arguments:

- Pointer to application routine to execute (fcnPtr)
- Pointer to arguments records (fcnArgPtr)
- Size of stack to allocate

Thread fork implementation:

- Sanity check arguments.
- Enter kernel mode and allocate a stack.
- Allocate a new TCB and initialize its register fields. In particular, the stack pointer is made to point at the stack, the PC return address is made to point at an OS (assembler) routine ThreadRoot, and two of the registers are initialized to fcnPtr and fcnArgPtr
- Put the newly allocated TCB on the ready list (Runnable). This will cause it to eventually be dispatched by run_new_thread, and start running the routine ThreadRoot.
- ThreadRoot:
  - Do start-up housekeeping (e.g., record start time).
  - Return to user mode.

    - Call fcnPtr(fcnArgPtr).
    - Do thread finish-up:call ThreadFinish.
- ThreadFinish:
    - Put any threads waiting on the termination of this thread on the ready list.
    - Can't deallocate thread yet, since we're still running on its stack. Record thread as "waitingToBeDestroyed". Call run_new_thread to run anther thread. ThreadHouseKeeping will examine waithingToBeDestroyed and deallocate the finished thread's TCB and stack.

```
run_new_thread() {
      newThread = PickNewThread();
      switch(curThread, newThread);
      ThreadHouseKeeping(); /* discussed later */
}
```

Thread fork is not the same thing as UNIX "fork". UNIX fork creates a new **process**, so it has to create a new address space, in addition to a new thread.

For now, don't worry about how switching between different processes' address spaces is done.

Thread fork is very much like an asynchronous procedure call – it means, go do this work, where the calling thread does not wait for the callee to complete. What if the calling thread needs to wait?

Thread **Join** – wait for a forked thread to finish.

Thus, a traditional procedure call is logically equivalent to doing a fork then immediately doing a join.

This is a normal procedure call:

```
A() { B(); }
B() { }
```

The procedure A can also be implemented as:

```
A'() {
      Thread t = new Thread;
      t->Fork(B);
      t->Join();
}
```

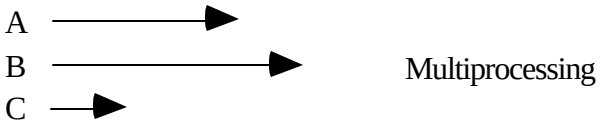
5.2 Multiprocessing vs. Multiprogramming

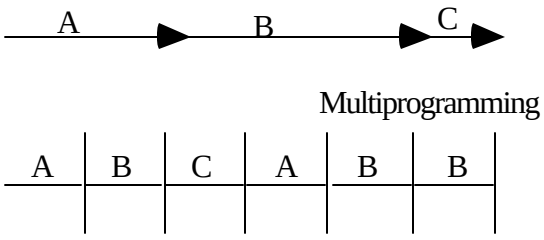
Multiprocessing = multiple CPU

Multiprogramming = multiple jobs or processes

Definition of "run concurrently" – scheduler is free to run threads in any order (e.g., FIFO, random, etc.)

For example:





Dispatcher can choose to run each thread to completion, or time-slice in big chunks, or time slice so that each thread executes only one instruction at a time (simulating a multiprocessor, where each CPU operates in lockstep).

If the dispatcher can do any of the above, programs must work under all cases, for all interleavings.

So how can you know if your concurrent program works? Whether **all** interleavings will work?

5.3 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!

5.4 Why allow cooperating threads?

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

1. **Share resources/information**
   - one computer, many users
   - one bank balance, many ATMs
   - embedded systems (ex: robot control)

2. **Speedup**
   - overlap I/O and computation
     – UNIX file system does read ahead
   - multiprocessors – chop up program into smaller pieces

3. **Modularity**
   - chop large problem up into simpler pieces.
     For example, to compile: gcc – cpp | cc1 | cc2 | as | ld
     This makes the system easier to extend; you can replace the assembler without changing the loader.

5.5 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 = 1`	`y = 2`
`x = y + 1`	`y = y * 2`

What are the possible values for x after the above?
What are the possible 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!

5.6 Atomic operations

What we want is some way of allowing a thread to perform a task without having other threads interfere with the task.

Atomic operation: an operation that always runs to completion, or not at all. It is *indivisible*: it can't be stopped in the middle, and its state can't be modified by someone else during the operation.

On most machines, memory reference and assignment (i.e., 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.