Lecture 3: Threads and Dispatching

3.0 Main Point:

Each thread has illusion of its own CPU, yet on a uniprocessor, all threads share the same physical CPU. How does this work?

Two key concepts:

1. thread control block
2. dispatching loop

3.1 Per-thread state

Thread control block (in Nachos, Thread class
- one per thread
  - execution state: registers, program counter, pointer to stack
  - scheduling information
- etc. (add stuff as you find a need)

3.2 Dispatching Loop (scheduler.cc)

```
LOOP
  Run thread

  Save state (into thread control block)

  Choose new thread to run

  Load its state  (into TCB) and loop
```

3.2.1 Running a thread:

How do I run a thread? Load its state (registers, PC, stack pointer) into the CPU, and do a jump.

How does dispatcher get control back? Two ways:

Internal events (Sleeping Beauty -- go to sleep and hope Prince Charming will wake you)

1. Thread blocks on I/O (examples: for disk I/O, or in emacs to wait for you to type at keyboard)

2. Thread blocks waiting for some other thread to do something

3. Yield -- give up CPU to someone else waiting

What if thread never did any I/O, never waited, and didn't yield control? Dispatcher has to gain control back somehow.

External events

1. Interrupts -- type character, disk request finishes wakes up dispatcher, so it can choose another thread to run

2. Timer -- like an alarm clock.

3.2.2 Choosing a thread to run

Dispatcher keeps a list of ready threads -- how does it choose among them?

Zero ready threads -- dispatcher just loops

One ready thread -- easy.

More than one ready thread:

1. LIFO (last in, first out): put ready threads on front of the list, and dispatcher takes threads from front. Results in starvation.

2. FIFO (first in, first out): put ready threads on back of list, pull them off from the front (this is what Nachos does)

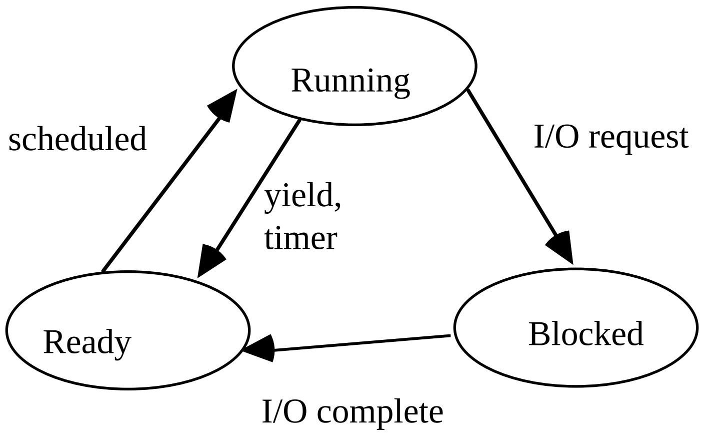
3. Priority queue -- give some threads better shot at CPU

Priority field in thread control block. Keep ready list sorted by priority.

3.2.3 Thread states

Each thread can be in one of three states:

1. Running -- has the CPU
2. Blocked -- waiting for I/O or synchronization with another thread
3. Ready to run -- on the ready list, waiting for the CPU



Thread State Diagram

3.2.4 Saving/restoring state (often called "context switch"):

What do you need to save/restore when the dispatcher switches to a new thread?

Anything next thread may trash: PC, registers, change execution stack

Want to treat each thread in isolation.

To take an example from the Nachos code, what if two threads loop, each calling "Yield"? Yield calls Switch to switch to the next thread, but once you start running the next thread, you are on a different execution stack. Thus, Switch is called in one thread's context, but returns in the other's!

Thread T switching to Thread S

There is a real implementation of Switch in Nachos in switch.s; of course, it's magical!

What if you make a mistake in implementing switch? For instance, suppose you forget to save and restore register 4? Get intermittent failures depending on exactly when context switch occurred, and whether new thread was using r4. Potentially, system will give wrong result, without any warning (if program didn't notice that r4 got trashed).

Can you devise an exhaustive test to guarantee that switch works? No!

3.2.5 Interrupts

Interrupts are a special kind of hardware-invoked context switch:

I/O finishes or timer expires. Hardware causes CPU to stop what it's doing, start running interrupt handler.

Handler saves state of interrupted thread

Handler runs

Handler restores state of interrupted thread (if time-slice, restore state of **new** thread)

Return to normal execution in restored state

3.3 Thread creation

Thread "fork" -- create a new thread

Thread fork implementation:

Allocate a new thread control block and execution call stack

Initialize the thread control block and stack, with initial register values and the address of the first instruction to run

Tell dispatcher that it can run the thread (put thread on ready list).

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.

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.

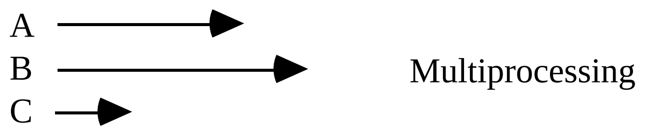
This is a normal procedure call:

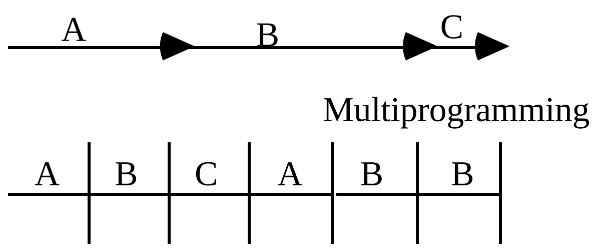
```
A() { B(); }
B() { }
```

The procedure A can also be implemented as:

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

3.4 Multiprocessing vs. Multiprogramming





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?