CS 162 Operating Systems and Systems Programming

Professor: Anthony D. Joseph

Spring 2001

Lecture4: Threads and Dispatching

4.0 Main Point:

Goal: Abstraction that each thread has illusion of its own CPU.

Hardware: shared CPU, interrupts.

How does this work?

4.1 Threads

Imagine we had the following C program to compute and list the digits of Pi into a file and to print out the cs162 class list to another file:

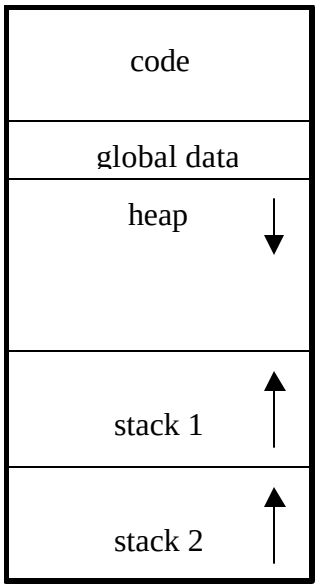
```
main() {
      ComputePi("/tmp/pi.text");
      PrintClassList("/tmp/clist.text");
}
```

This program would unfortunately never print out the class list since ComputePi() will never finish.
With threads we could have the two activities go on in parallel:

```
main() {
      CreateThread(ComputePi, "/tmp/pi.text");
      CreateThread(PrintClassList", "/tmp/clist.text");
}
```

If we stopped this program and examined it with a debugger we would see *two* stacks that could be listed and *two* sets of CPU registers to examine.

Note: we can't simply let each stack "grow" backwards towards the heap anymore! Have to worry about stacks overrunning each other.



Two key concepts:

1. thread control block (per-thread state)
2. dispatching loop

4.1.1 Per-thread state

*Thread Control Bloc*k (TCB)
(sometimes called a "Process Control Block" or PCB)

This is where all information relevant to the thread is kept. There is one per active thread. The contents include:

- execution state: CPU registers, program counter, pointer to stack
- scheduling information: state (see below) , priority, CPU time used
- accounting info
- various pointers (for implementing scheduling queues)
- etc. (add stuff as you find a need)
- In Nachos: "Thread" is a class that includes the TCB

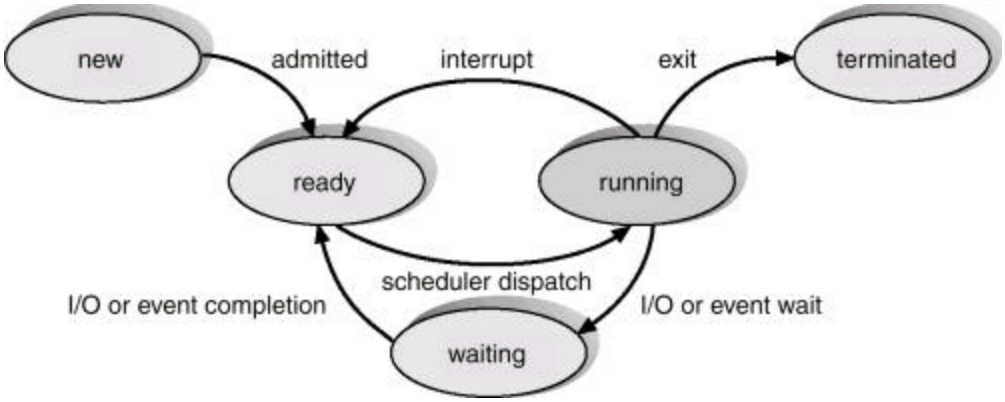
OS keeps an array (or linked list or …) of TCBs in its own protected memory. We describe one approach to this below.

4.2 The Lifecycle of a Thread

Threads all go through the lifecycle shown below:

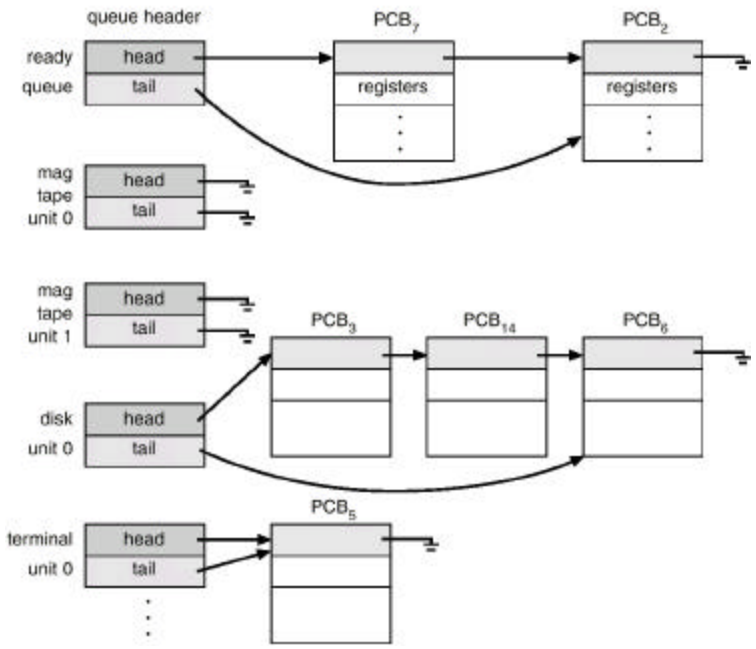
1) The are created and admitted to the system
2) They remain in the system, sharing resources with all other threads in the system. Of course, only one thread can be actually "running" on the CPU at any time (assuming a single-CPU system).
3) They leave the system.

We will first describe how threads are managed when they are in the system. We will then discuss how thread creation and termination are handle



Silberschatz and Galvin ©1999

An "active" thread (i.e., one that is in the *ready, waiting,* or *running* state) is represented in the system by its TCB. The TCBs are organized into queues based on their state. For threads waiting for an event (I/O, other thread completion, etc.) they can be queues for each such event. An example of such an arrangement is shown below:



Silberschatz and Galvin © 1999

4.3 Dispatching Loop (scheduler.cc)

The main loop of the scheduler is responsible for choosing a thread to run and ensuring that thread switching is done correctly. *Conceptually*, it looks like this:

```
Loop {
      Run thread

      Choose new thread to run

      Save state of old thread from CPU into its TCB

      Load state of new thread from its TCB into CPU

}
```

4.3.1 Running a thread:

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

How does dispatcher get control back? Two ways:

- Internal events: thread hands control back voluntarily
- External events: thread gets *preempted*

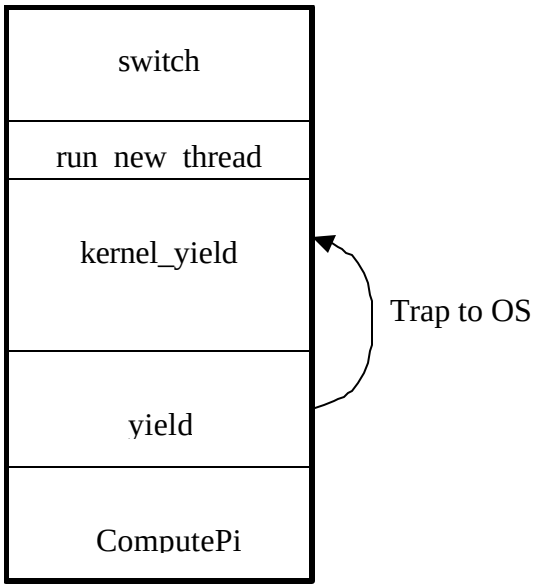
Internal events:

- Thread blocks on I/O (e.g., disk I/O, or waiting on keyboard input)
- Thread blocks waiting on a "signal" from another thread
- Yield (give CPU to someone else who's ready to run)
-

Yield thread switch example:

```
ComputePi() {
          while (TRUE) {
               ComputeNextDigit();
               yield();
          }
}
```

Stack for yielding thread:




```
run_new_thread() {
          newThread = PickNewThread();
          switch(curThread, newThread);
          ThreadHouseKeeping(); /*covered next lecture */
}
```

4.3.2 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

switch routine (shown here in C syntax, but would be implemented in assembler):

```
switch(int tCur, int tNew) {
          /* Save registers of running thread to TCB */
          TCB[tCur].regs.r7 = CPU.r7;
          …
          TCB[tCur].regs.r0 = CPU.r0;
          TCB[tCur].regs.sp = CPU.sp;
          TCB[tCur].regs.retpc = CPU.retpc; /* return addr */

          /* Load state of new thread into CPU */
          CPU.r7 = TCB[tNew].regs.r7;
          …
          CPU.r0 = TCB[tNew].regs.r0;
          CPU.sp = TCB[Nnew].regs.sp;
          /* Henceforth running on new thread's stack */
          CPU.retpc = TCB[tNew].regs.retpc;

          return;

          /* Return from switch returns to CPU.retpc */
}
```

Note, the "retpc" is when the return from switch should jump to. In practice this is really implemented as a "jump" to that location.
There is a real implementation of Switch in Nachos in switch.s; of course, it's magical! (but you should be able to follow it)

What if you make a mistake in implementing switch?

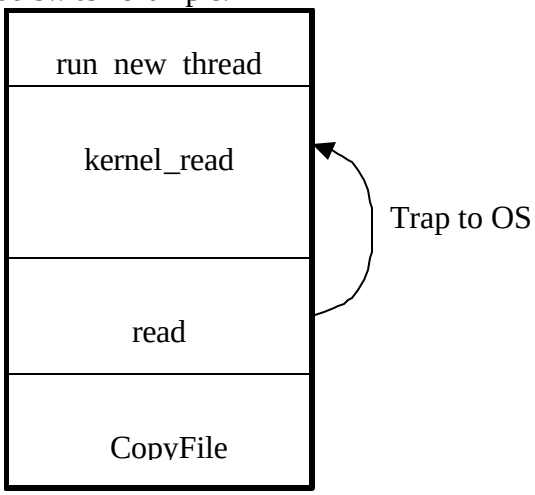
For instance, suppose you forget to save and restore register 4?

Get intermittent failures depending on exactly when context switch occurred, and whether new thread was using register 4. Potentially, system will give wrong result, without any warning (if program didn't notice that register 4 got trashed).

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

4.3.3 Note: We assume that all threads go through run_new_thread() in order to be removed from and reinstated upon the CPU.

Blocked on I/O thread switch example:



Blocked on thread signal example is similar - join.

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

External events:

- Interrupts – type character, disk I/O request finishes wakes up dispatcher, so it can choose another thread to run
- Timer – like an alarm clock that goes off every n milliseconds

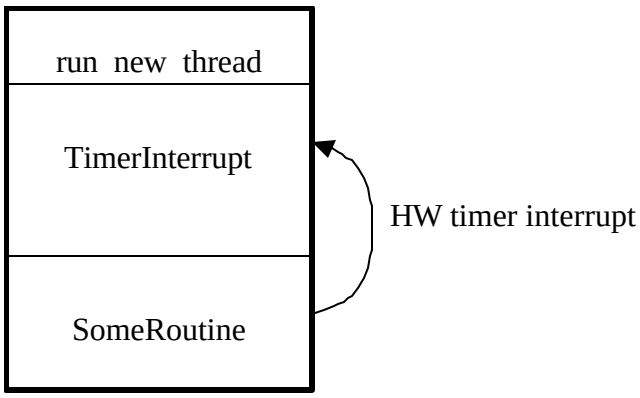
Interrupts are a special kind of hardware-invoked context switch. No separate step to choose what to run next; always run the interrupt handler immediately.

Timer interrupt routine:

```
TimerInterrupt() {
```

```
            DoPeriodicHouseKeeping();
            run_new_thread();
}
```

Stack of preempted thread:



I/O interrupt: same as timer interrupt except that DoHouseKeeping() is replaced by ServiceIO().

4.3.4 Choosing a thread to run

Dispatcher keeps a list of ready (*runnable*) threads – how does it choose among them?

- Zero ready threads – dispatcher just loops.

  Alternative is for OS to create an "idle thread" that simply loops or does nothing. Then there's always at least one runnable thread.
- Exactly 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. Pick one at random: Starvation can occur.
3. FIFO (first in, first out): put ready threads on back of list, pull them off from the front (this is what Nachos does). Fair.
4. Priority queue – keep ready list sorted by priority field of TCB. This allows important threads to always get the CPU whenever they need it.