Operating Systems Design (CS 423)

Elsa L Gunter
2112 SC, UIUC
http://www.cs.illinois.edu/class/cs423/

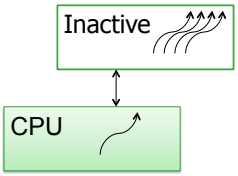
Based on slides by Roy Campbell, Sam King, and Andrew S Tanenbaum

Thread implementation on uni-proc.

- So far, we've been assuming that we have enough physical process to run each thread on its own processor
  - But threads are useful also for running when you have more threads than CPUs (web server example)
  - How to give the illusion of infinite physical processors on a finite set of processors?

Ready threads

- What to do with thread while it's not running
  - Must save private state somewhere
  - What constitutes private data for a thread?



Ready threads

- What to do with thread while it's not running
  - Must save private state somewhere
  - What constitutes private data for a thread?
  - Registers including program counter
  - Stack contents and stack pointer
  - Code
  - All needs to be save when thread swapped out and restored when thread swapped back in

Thread context

- This information is called the thread "context" and is stored in a "thread control block" when the thread isn't running
  - To save space, share code among all threads
  - To save space, don't copy stack to the thread control block
  - Multiple stacks in same address space, copy stack pointer in thread control block

Thread context

- Keep thread control blocks for threads that aren't running on a queue of **ready** threads
- Thread state can now be:
  - running
  - ready
  - or blocked

Switching threads

- Steps to switch to another thread
  - Thread returns control to the OS
  - Choose new thread to run
    - scheduling
  - Save state of current thread
  - Load context of the next thread
  - Run thread

Returning control to the OS

- Come up with a list of ways for a thread to switch to the OS

Returning control to OS

- How does thread return control back to the OS (so system can save state of current thread and run new one)?

Returning control to OS

- How does thread return control back to the OS (so system can save state of current thread and run new one)?
- System call
- Yield
- When it exits
- Call to `lock` or `wait`

Returning control to OS

- Is it enough to depend on internal events?

Returning control to OS

- Is it enough to depend on internal events?
- If we must wait for thread to give up control, may not be able to be fair to others
- Need interupts

Choosing the next thread to run

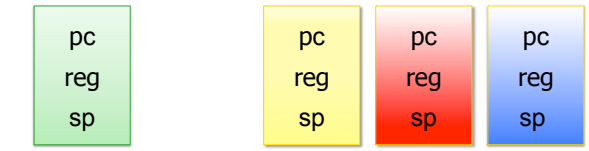
- If no ready thread, just loop idly
  - Loop switches to a thread when one is ready
- If 1 ready thread, run it
- If more than 1 ready, choose one
  - FIFO
  - Priority queue (more on this later)

Context switching

- What is a thread?
- What do you do with a thread when it is not running?

Context switching

- What is a thread?
- What do you do with a thread when it is not running?



Context switching

- What is a thread?
- What do you do with a thread when it is not running?

Saving state of current thread

- How to save state of the current thread?
- Save registers, PC, stack pointer
  - Very tricky assembly-language code
  - Why won't the following code work?

```
100 save PC (I.e. value 100)
101 switch to next thread
```

  - In MP3, we'll use Unix's `swapcontext()`

Loading context of new thread

- How to load the context of the next thread to run and run it?
  - Registers?
  - Stack?
  - Resume execution?
- Who is running these steps?
- How does the thread that just gave up control run again?

swapcontext

- Unix support for switching threads
- Like Linux context switching function
  - Hides many of the details
  - Moves context running on CPU to mem add or context at mem add to CPU or combination
  - What does `swapcontext` store?
  - We will talk about how to use `swapcontext` next time

swapcontext

- Unix support for switching threads
- Like Linux context switching function
  - Hides many of the details
  - Moves context running on CPU to mem add or context at mem add to CPU or combination
  - What does `swapcontext` store?
    Stores registers (including PC), stack pointer
  - We will talk about how to use `swapcontext` next time

Example of thread switching

- Thread 1

```
print "start thread 1";
yield();
print "end thread 1";
```

- Thread 2

```
print "start thread 2";
yield();
print "end thread 2";
```

- Yield

```
print "start yield (thread %d)";
switch to next thread (swapcontext);
print "end yield (current thread %d);
```


Thread Output

- Thread 1

```
start thread 1
start yield (thread 1)


end yield (thread 1)
end thread 1
```

- Thread 2

```
start thread 2
start yield (thread 2)


end yield (thread 2)
end thread 2
```


Thread switching in Linux

- PCB == TCB conceptually
- Thread switching is the same as Process switching except that the address space stays the same
- To make switching work in Linux, any thread that switches **must** do so through same one switching function

Thread switching in Linux

- When executing in kernel, executing on behalf of a thread
- Kernel stack key to this abstraction on x86
  - Contains local state (stack) and process struct
  - E.g., current pointer (recall it from MP1)
- Other architectures use different techniques

```
switch_to(task_struct *prev_p,
          task_struct *next_p)
```