COS 318: Operating Systems

Implementing Threads

Jaswinder Pal Singh
Computer Science Department
Princeton University

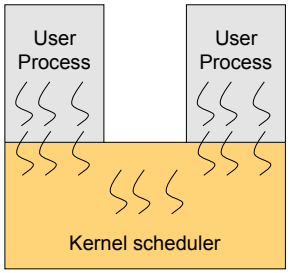
(http://www.cs.princeton.edu/courses/cos318/)

Today's Topics

- Thread implementation
  - Non-preemptive versus preemptive threads
  - Kernel vs. user threads

Revisit Monolithic OS Structure

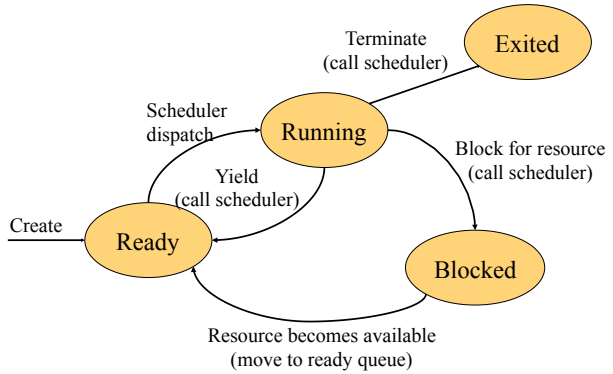
- Kernel has its address space, shared with all processes
- Kernel consists of
  - Boot loader
  - BIOS
  - Key drivers
  - Threads
  - Scheduler
  - …
- Scheduler
  - Use a ready queue to hold all ready threads
  - Schedule in a thread in the same address space (thread context switch)
  - Schedule in a thread with a different address space (process context switch)



Thread context switch

- Scheduler schedules threads on context switch
- Voluntary
  - Thread_yield
  - Thread_join (if child is not done yet)
- Involuntary
  - Interrupt or exception
  - Some other thread is higher priority

Non-Preemptive Scheduling



Non-Preemptive Scheduling (contd.)

- A non-preemptive scheduler invoked by calling
  - block()
  - yield()
- The simplest form

  Scheduler:

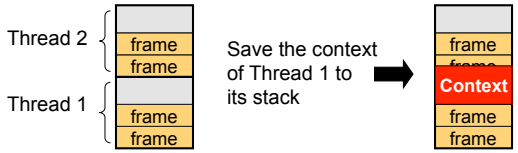
  save current process/thread state
  choose next process/thread to run
  dispatch (load PCB/TCB and jump to it)
- Scheduler can be viewed as just another kernel thread

Thread Context

- Can be classified into two types:
  - Private
  - Shared
- Shared state
  - Contents of memory (global variables, heap)
  - File system
- Private state
  - Program counter
  - Registers
  - Stack

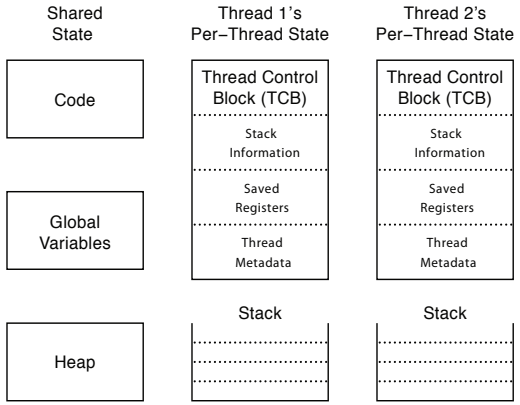
Where and How to Save Thread Context?

- Save the context on the thread's stack
  - Many processors have a special instruction to do it efficiently
  - But, need to deal with the overflow problem



- Check before saving
  - Make sure that the stack has no overflow problem
  - Copy it to the TCB residing in the kernel heap
  - Not so efficient, but no overflow problems

Thread Data Structures



Thread Control Block (TCB)

- Current state
  - Ready: ready to run
  - Running: currently running
  - Blocked: waiting for resources
- Registers
- Status (EFLAGS)
- Program counter (EIP)
- Stack

Voluntary thread context switch

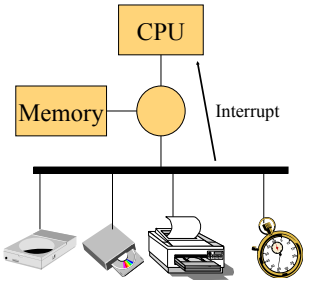
- Save registers on old stack
- Switch to new stack, new thread
- Restore registers from new stack
- Return
- Exactly the same with kernel threads or user threads

Pseudo code for thread_switch

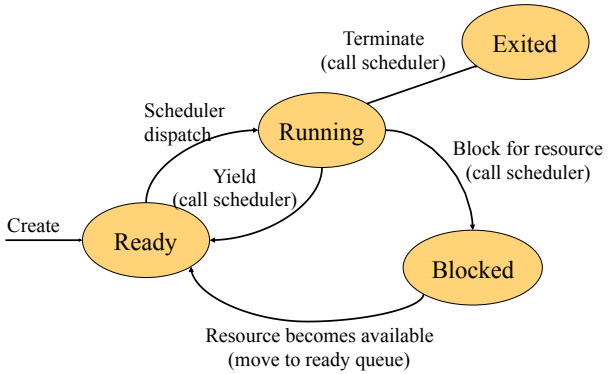
```
// We enter as oldThread, but we return as newThread.
// Returns with newThread's registers and stack.

void thread_switch(oldThreadTCB, newThreadTCB) {
    pushad;                 // Push general register values onto the old stack.
    oldThreadTCB->sp = %esp; // Save the old thread's stack pointer.
    %esp = newThreadTCB->sp; // Switch to the new stack.
    popad;                  // Pop register values from the new stack.
    return;
}
```

Preemption

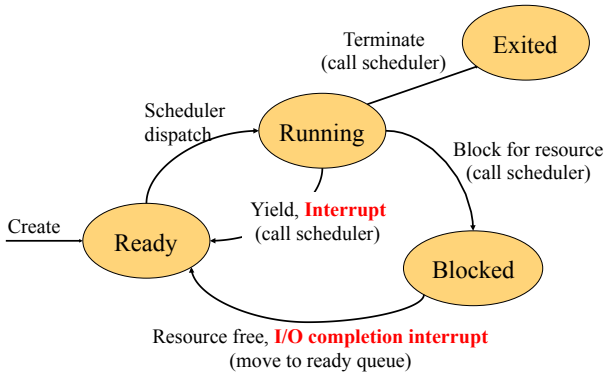
- Why?
  - Timer interrupt for CPU management
  - Asynchronous I/O completion
- When is CPU interrupted?
  - Between instructions
  - Within an instruction, except atomic ones
- Manipulate interrupts
  - Disable (mask) interrupts
  - Enable interrupts
  - Non-Maskable Interrupts



State Transition for Non-Preemptive Scheduling



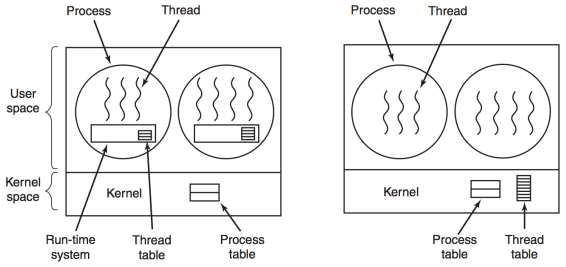
State Transition for Preemptive Scheduling



Interrupt Handling for Preemptive Scheduling

- Timer interrupt handler:
  - Save the current process / thread to its PCB / TCB
  - Call scheduler
- I/O interrupt handler:
  - Save the current process / thread to its PCB / TCB
  - Do the I/O job
  - Call scheduler
- Issues
  - Disable/enable interrupts
  - Make sure that it works on multiprocessors

User Threads vs. Kernel Threads



- Kernel knows only about processes, not threads
- Context switch at user-level without OS (Java threads)
- Preemptive scheduling?
- What about I/O events?

- A user thread
  - Makes a system call (e.g. I/O)
  - Gets interrupted
- Context switch in the kernel

Summary of User vs. Kernel Threads

- User-level threads
  - User-level thread package implements thread context switches
  - OS doesn't know the process has multiple threads
  - Timer interrupt (signal facility) can introduce preemption
  - When a user-level thread is blocked on an I/O event, the whole process is blocked
    - Precisely the case for which threads are often useful …
  - Allows user-level code to build custom schedulers
- Kernel-threads
  - Kernel-level threads are scheduled by a kernel scheduler
  - A context switch of kernel-threads is more expensive than user threads due to crossing protection boundaries
- Hybrid
  - It is possible to have a hybrid scheduler, but it is complex

Interactions between User and Kernel Threads

- Each thread has its own user stack. What about kernel stack? Two possibilities:
  - Each user thread has its own kernel stack
  - All threads of a process share the same kernel stack

	Private kernel stack	Shared kernel stack
Memory usage	More	Less
System services	Concurrent access	Serial access
Multiprocessor	Yes	Not within a process
Complexity	More	Less

Summary

- Non-preemptive threads issues
  - Scheduler
  - Where to save contexts
- Preemptive threads
  - Interrupts can happen any where!
- Kernel vs. user threads
  - Main difference is which scheduler to use