COS 318: Operating Systems

Implementing Threads

Jaswinder Pal Singh and a Fabulous Course Staff
Computer Science Department
Princeton University

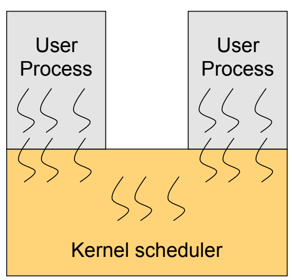
(http://www.cs.princeton.edu/courses/cos318/)

Today' s Topics

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

Revisit Monolithic OS Structure

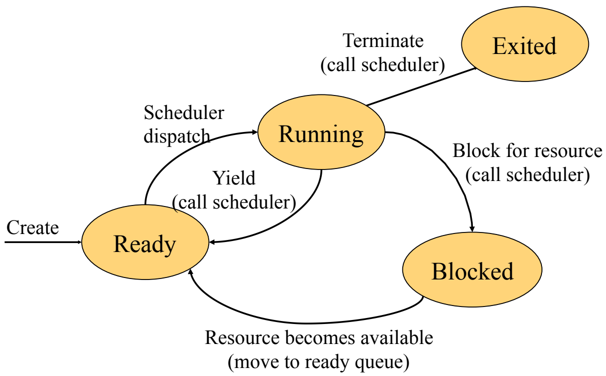
- Kernel consists of
  - Boot loader
  - BIOS
  - Key drivers
  - Threads
  - Scheduler
  - …
- Scheduler
  - Use a ready queue to hold all ready threads
  - Schedule in a thread in the same address space (thread context switch)
  - Schedule in a thread with a different address space (process context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



Thread context switch

- Scheduler schedules threads on context switch
- Voluntary
  - Thread yields or blocks, e.g. for a resource like disk, a synchronization variable etc
  - Thread_join (wait for a target process, e.g. child, to terminate)
- Involuntary
  - Interrupt or exception
  - Some other thread of higher priority needs to run

Non-Preemptive Scheduling



Non-Preemptive Scheduling (contd.)

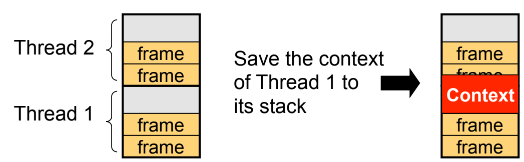
- A non-preemptive scheduler invoked by thread calling a yield, block, join or similar
- Simplest form of scheduler: When invoked:
  - **save current process/thread state**
  - **choose next process/thread to run**
  - **dispatch (load PCB/TCB and jump to it)**
- Scheduler can be viewed as just another 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 Control Block (TCB)

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

10

Voluntary thread context switch

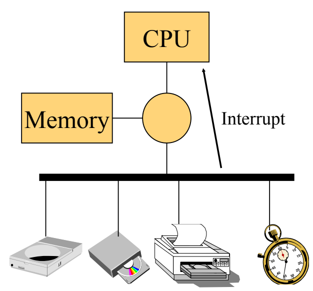
- Save registers on old stack
- Switch to new stack, new thread
- Restore registers from new stack
- Return
- Exactly the same with kernel threads or user threads

```
// 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;
}
```

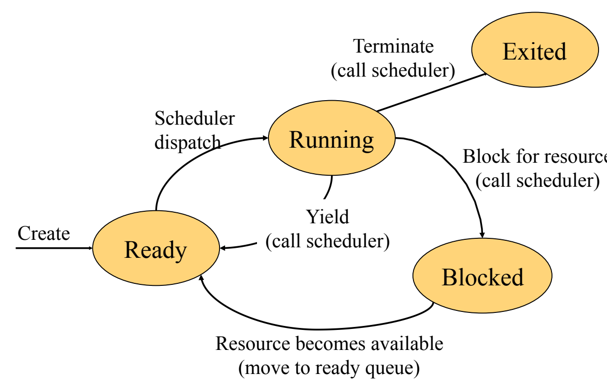
Preemption

- 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



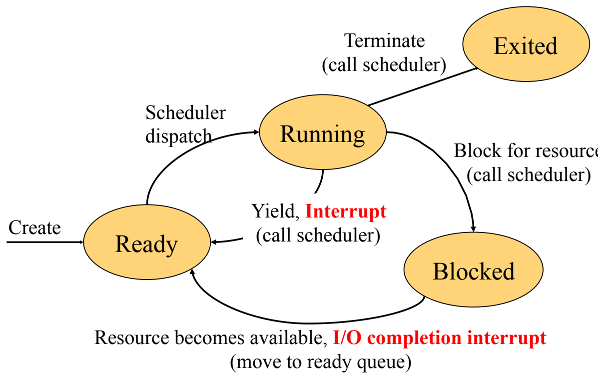
12

Recall: Non-Preemptive Scheduling



13

State Transitions for Preemptive Scheduling



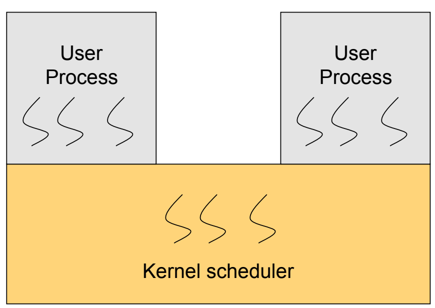
14

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

15

User- and Kernel-level Threads

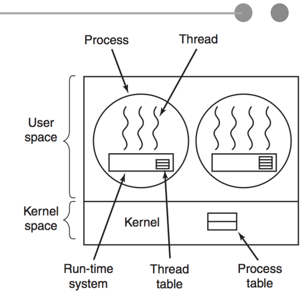


- Threads at user level (in user space, user mode) and at kernel level
- User level threads map to kernel level threads, which are all the operating system really knows about

17

User-level Threads

- Managed by user-level runtime software, run in user mode
- Kernel knows only about user processes, not user threads
- Invoking thread API leads to user-level function call
- Context switch at user-level
- Preemption?



- Fast (could be as fast as function call)
- Can have custom user-level schedulers
- Lower kernel complexity
- Can implement on kernels that are single-threaded

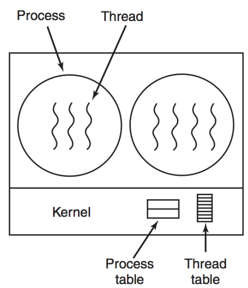
- Extreme case: kernel is single-threaded

18

Kernel Threads

- Managed by OS, run in kernel mode
- Invoking thread API causes system call
- Context switch invokes OS
- PCB per process and TCB per thread in kernel

- Kernel has knowledge of threads so can optimize better
  - E.g. give more CPU time to processes with more threads
- When one thread in a process blocks, others can still run
  - Good for cases where threads block frequently



- Extreme case: one kernel thread per user thread, so no need to for user thread mgmt

19

Disadvantages of User and Kernel Threads

- User threads
  - 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 …
  - Kernel may not be able to schedule processes optimally
    - May schedule process with idle threads
    - May not give more CPU to processes with many threads
    - May need OS modifications or other mechanisms to solve
  - Multiprocessor or multi-core systems need at least one kernel thread per processor/core, so hard to do only user-level
- Kernel threads
  - Thread context switches and thread operations more expensive (cross OS boundary)
  - More complexity in kernel

20

Implementation Models for User-level Threads

- User threads are mapped to kernel threads
  - Can think of them as a kernel thread per “virtual processor”
  - (hence need at least one kernel-level thread per core)
- Simplest case, discussed so far, is many to 1
  - Only one user-level thread runs at a time, since only one kernel thread
  - Case of kernel level threads so far can be viewed as 1:1
- Other models exist
  - m user threads mapped to n kernel threads
  - certain user level threads bound to a subset of kernel threads
  - Dynamically change-able no. of kernel threads for user process (but needs more communication mechanisms up/down), etc.

21

Summary of User vs. Kernel Threads

- 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
  - Pure user-level threads hard to make work on multiprocessor: need multiple kernel-level threads (virtual processors)
- Hybrid models exist, but are complicated
  - E.g. using kernel level threads can be 1-1 for user-to-kernel level thread mapping, using user-level threads is many-to-1, and there are many-to-many alternatives as well

22

Interactions between User and Kernel Threads

- Every thread has its own user stack. What about kernel stack? Two possibilities:
  - Every 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