COS 318: Operating Systems

Implementing Threads

Jaswinder Pal Singh
Computer Science Department
Princeton University

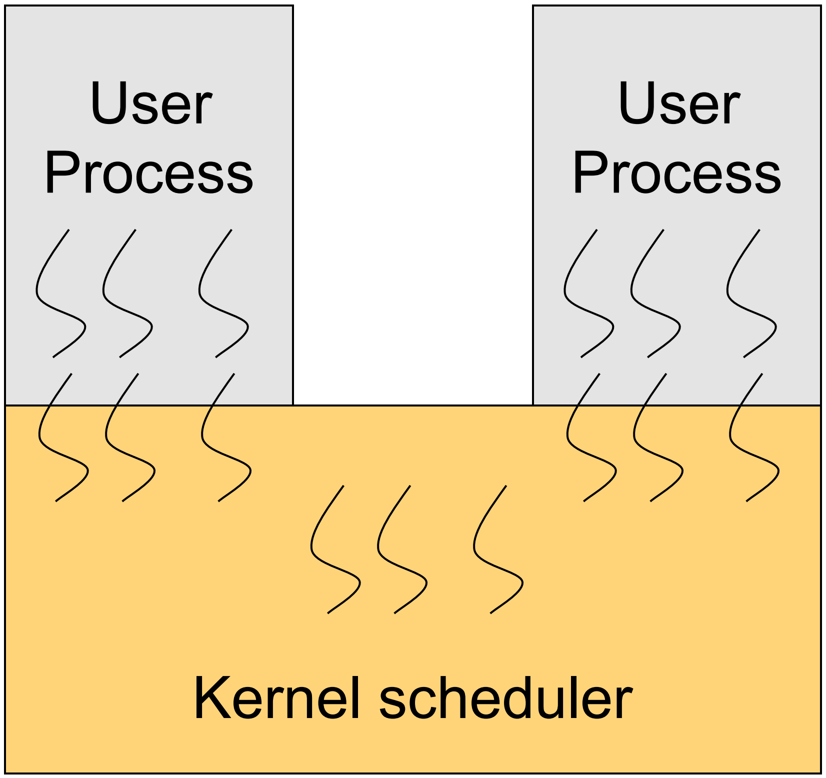
(http://www.cs.princeton.edu/courses/cos318/)

Today's Topics

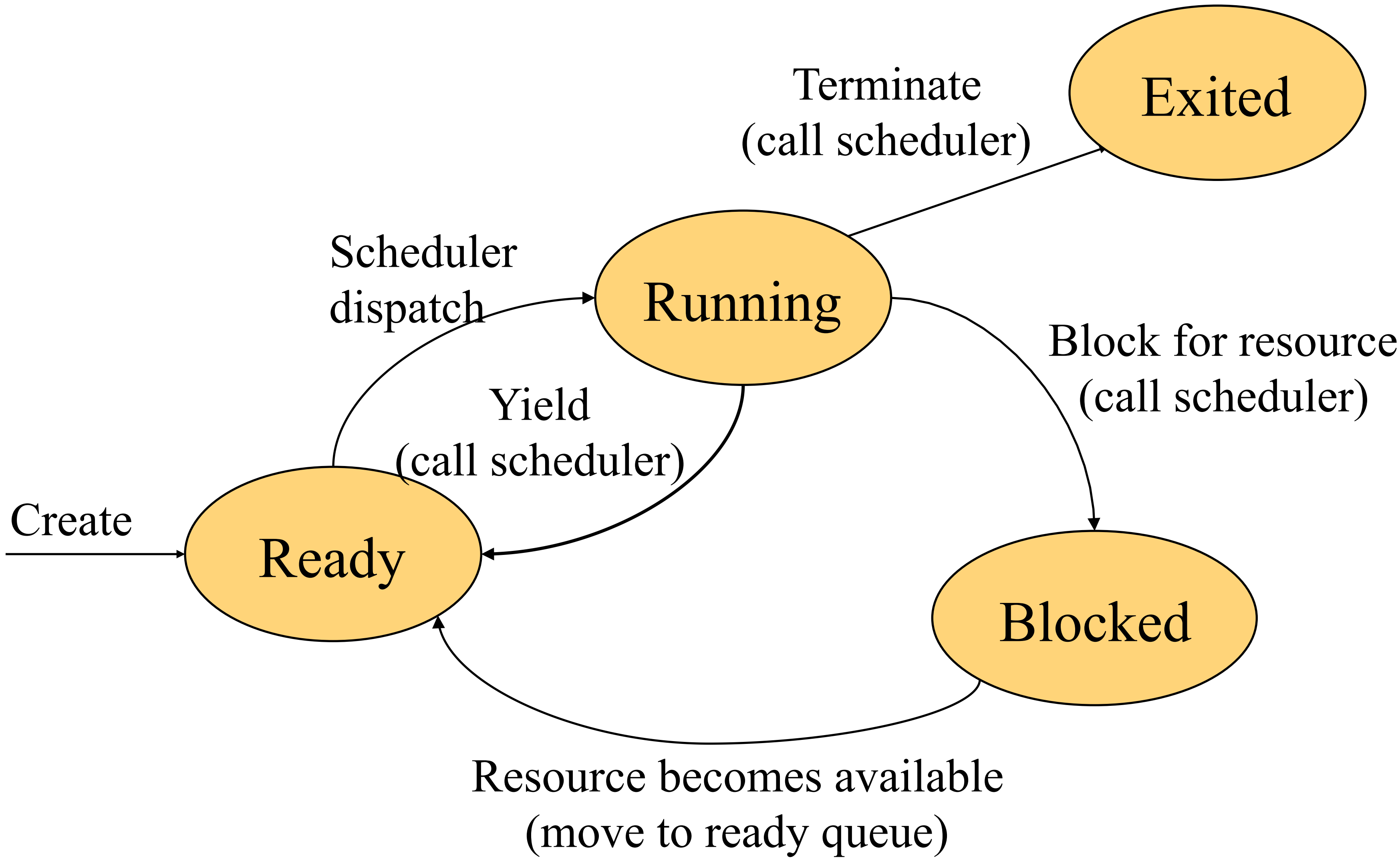
- Thread implementation
  - Non-preemptive versus preemptive threads
  - Kernel vs. user threads
- Cookies: always too many or too few

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)



Non-Preemptive Scheduling



Scheduler

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

- The simplest form

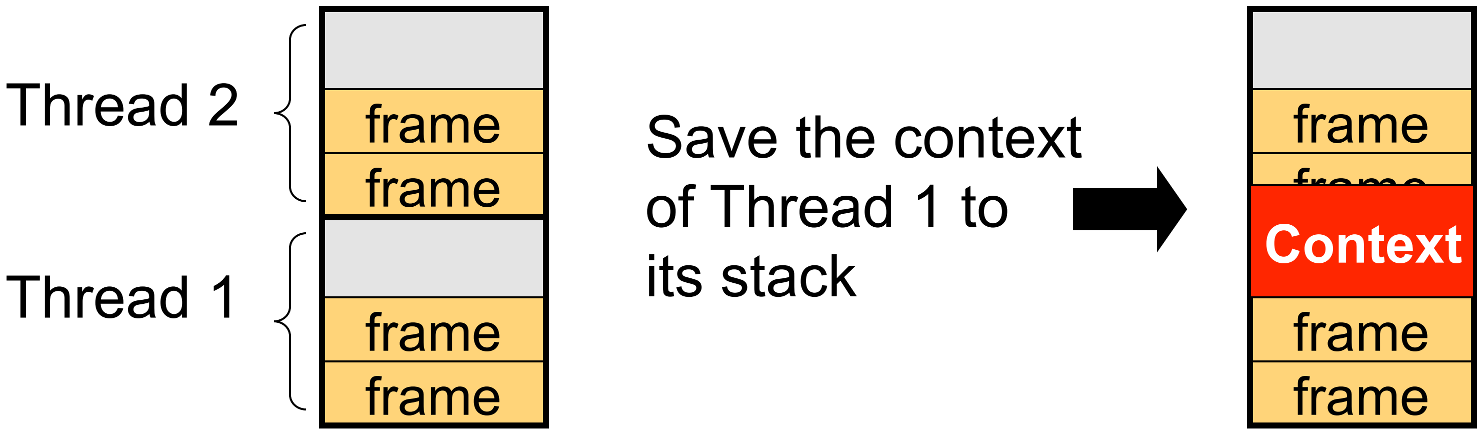
  Scheduler:

  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

Where and How to Save Thread Context?

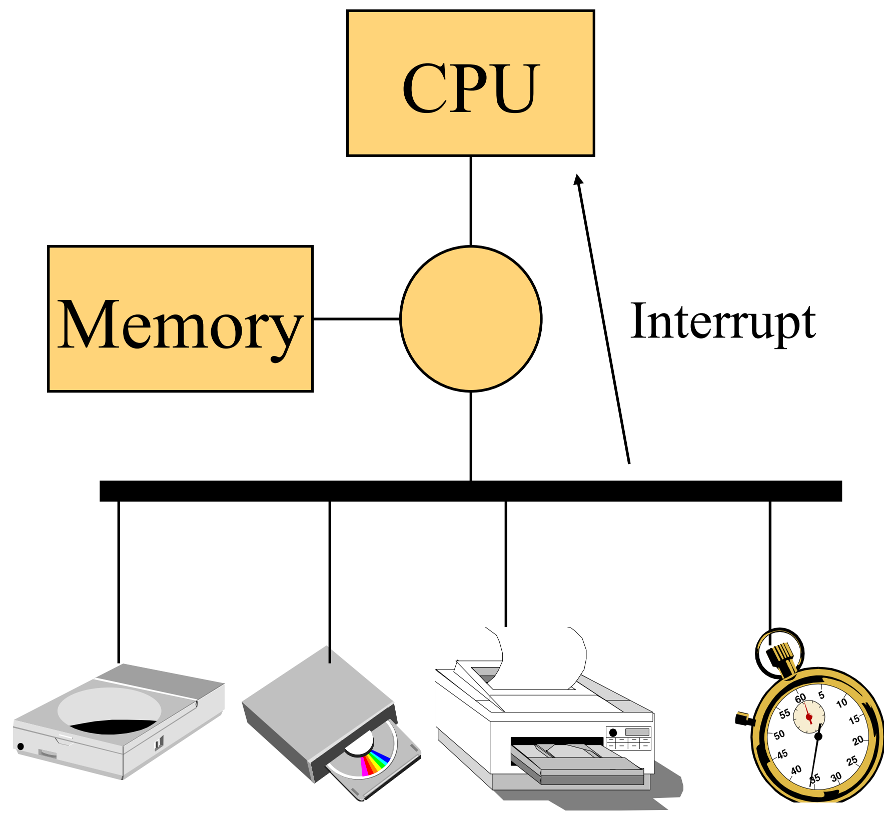
- Save the context on the thread's stack
  - Many processors have a special instruction to do it efficiently
  - But, need to deal with the overflow problem



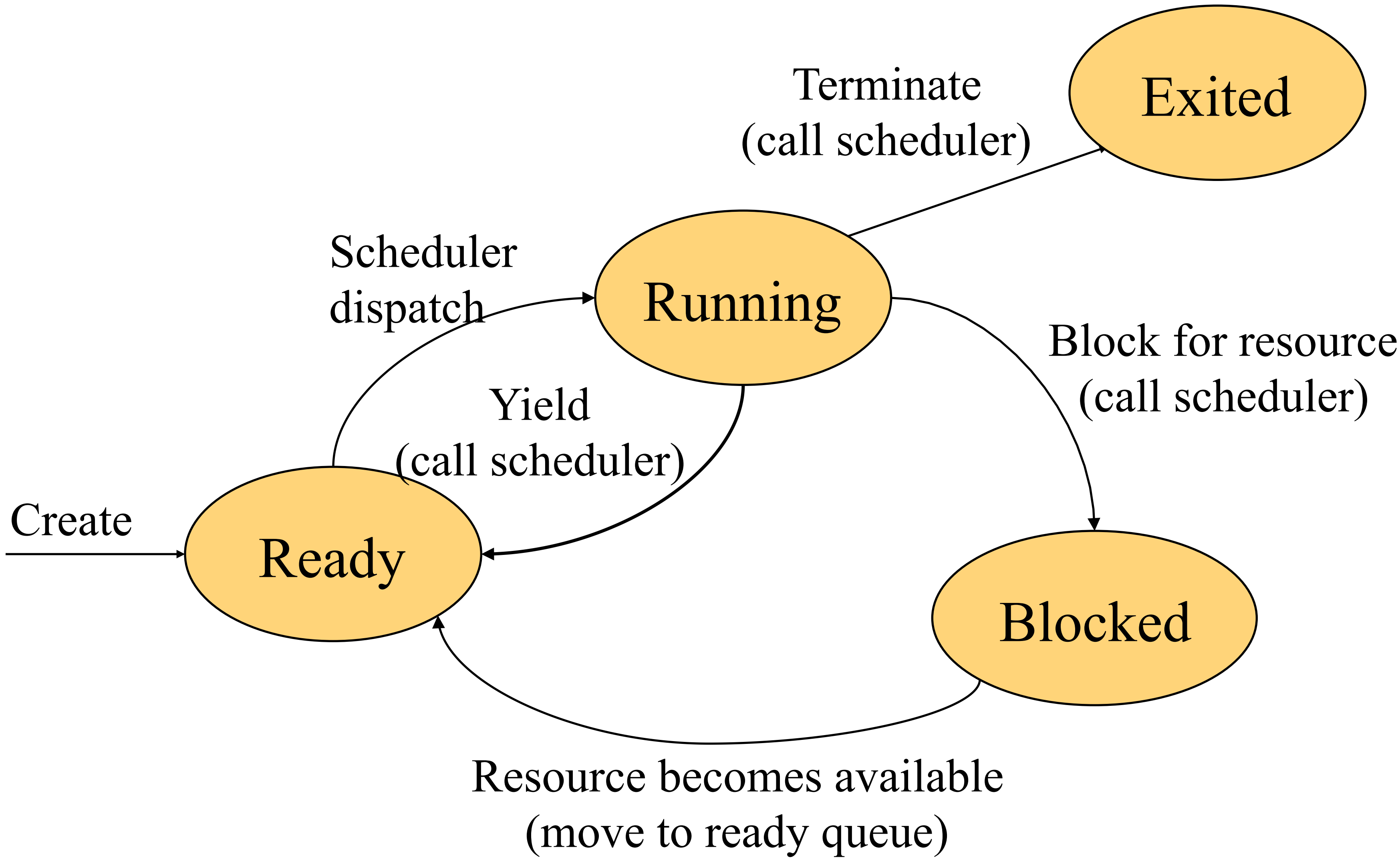
- Check before saving
  - Make sure that the stack has no overflow problem
  - Copy it to the TCB residing in the kernel heap
  - Not so efficient, but no overflow problems

Preemption

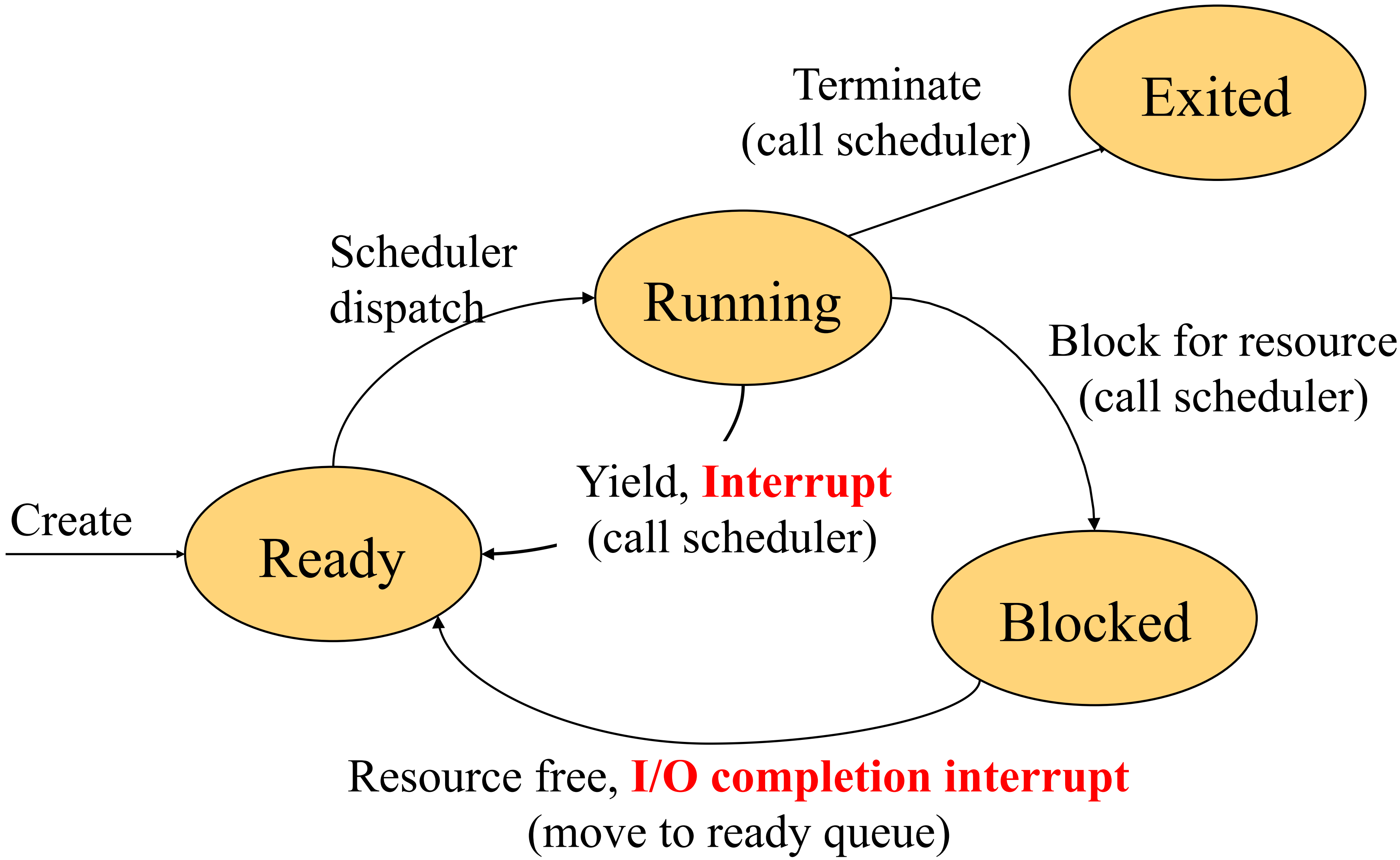
- Why?
  - Timer interrupt for CPU management
  - Asynchronous I/O completion
- When is CPU interrupted?
  - Between instructions
  - Within an instruction, except atomic ones
- Manipulate interrupts
  - Disable (mask) interrupts
  - Enable interrupts
  - Non-Maskable Interrupts



State Transition for Non-Preemptive Scheduling



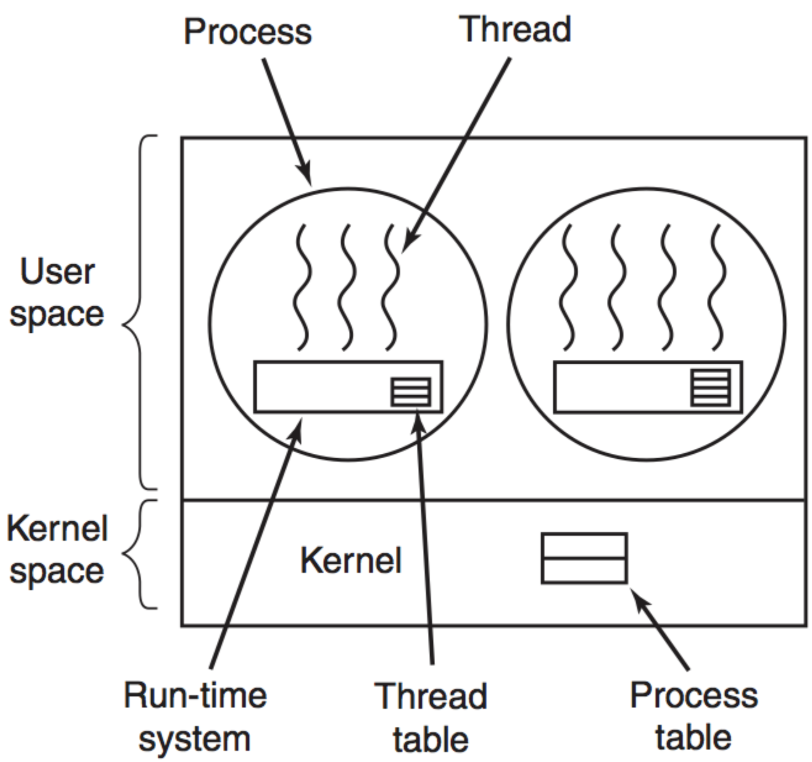
State Transition for Preemptive Scheduling

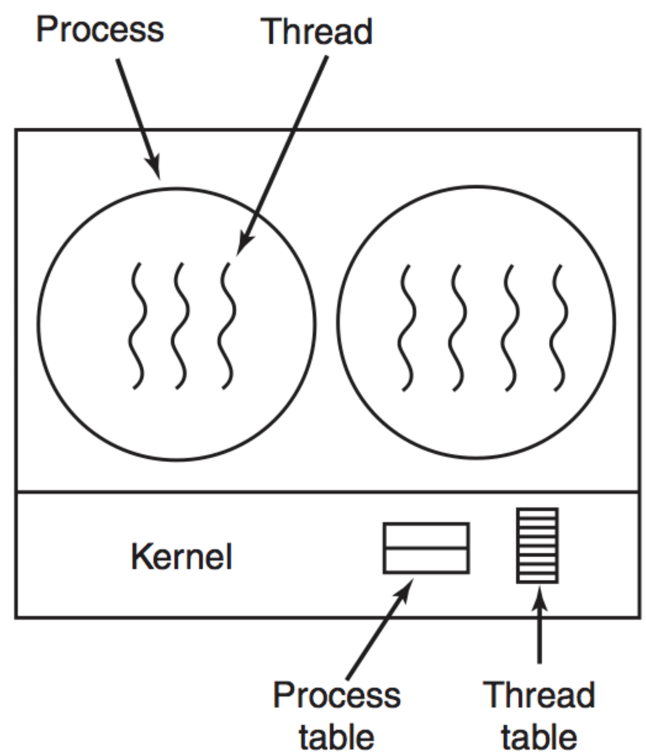


Interrupt Handling for Preemptive Scheduling

- 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

"Too Many Cookies" Problem

- Want cookies, but don't want to buy too many cookies
- Any person can be distracted at any point

	RoomMate A	RoomMate B
15:00	Look in cabinet: out of cookies	
15:05	Leave for Wawa	
15:10	Arrive at Wawa	Look at fridge: out of cookies
15:15	Buy a bag of cookies	Leave for Wawa
15:20	Arrive home; put cookies away	Arrive at Wawa
15:25		Buy a bag of cookies
		Arrive home; put cookies away Oh No! Too many cookies.

Using A Note?

Thread A

```
if (noCookies) {
   if (noNote) {
      leave note;
      buy cookies;
      remove note;
   }
}
```

Thread B

```
if (noCookies) {
   if (noNote) {
      leave note;
      buy cookies;
      remove note;
   }
}
```

- Any issue with this approach?

Another Possible Solution: Leave Note First

Thread A

```
leave noteA
if (noNoteB) {
 if (noCookies) {
     buy cookies
 }
}
remove noteA
```

Thread B

```
leave noteB
if (noNoteA) {
 if (noCookies)
{
     buy cookies
 }
}
remove noteB
```

Didn't buy cookies

Didn't buy cookies

- Does this method work?

Possible Solution: One of them Spin-waits?

- Problem was that threads checked once and moved on
  - So have one of them spin-wait on the note

Thread A

```
leave noteA
while (noteB)
  do nothing;
if (noCookies)
  buy cookies;
remove noteA
```

Thread B

```
leave noteB
if (noNoteA) {
  if (noCookies) {
    buy cookies
  }
}
remove noteB
```

- Would this fix the problem?

Remarks

- The last solution works, but
  - Life is too complicated
  - A's code is different from B's
  - Busy waiting is a waste
- What we want is:

```
Acquire(lock);
if (noCookies)
  buy cookies;
Release(lock);
```

} **Critical section**

What Is A Good Solution

- Only one process/thread inside a critical section
- No assumption about CPU speeds
- A process/thread inside a critical section should not be blocked by any process outside the critical section
- No one waits forever

- Works for multiprocessors
- Same code for all processes/threads

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
- Too many cookies problem
  - What we want is mutual exclusion