



COS 318: Operating Systems

Non-Preemptive and Preemptive Threads

Kai Li

Computer Science Department

Princeton University

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



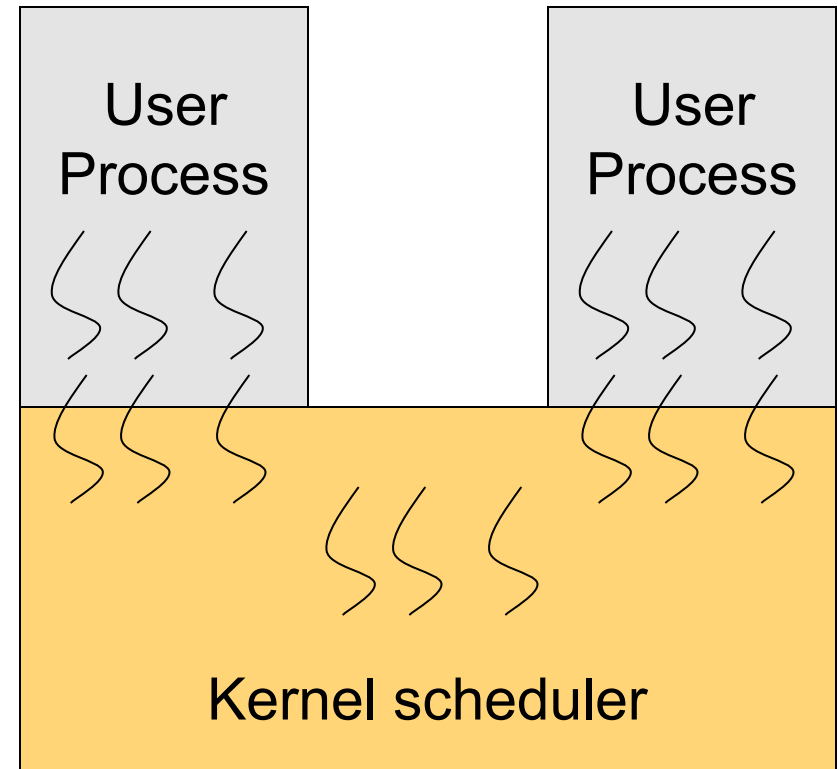
Today's Topics

- ◆ Non-preemptive threads
- ◆ Preemptive threads
- ◆ Kernel vs. user threads
- ◆ Too much milk problem

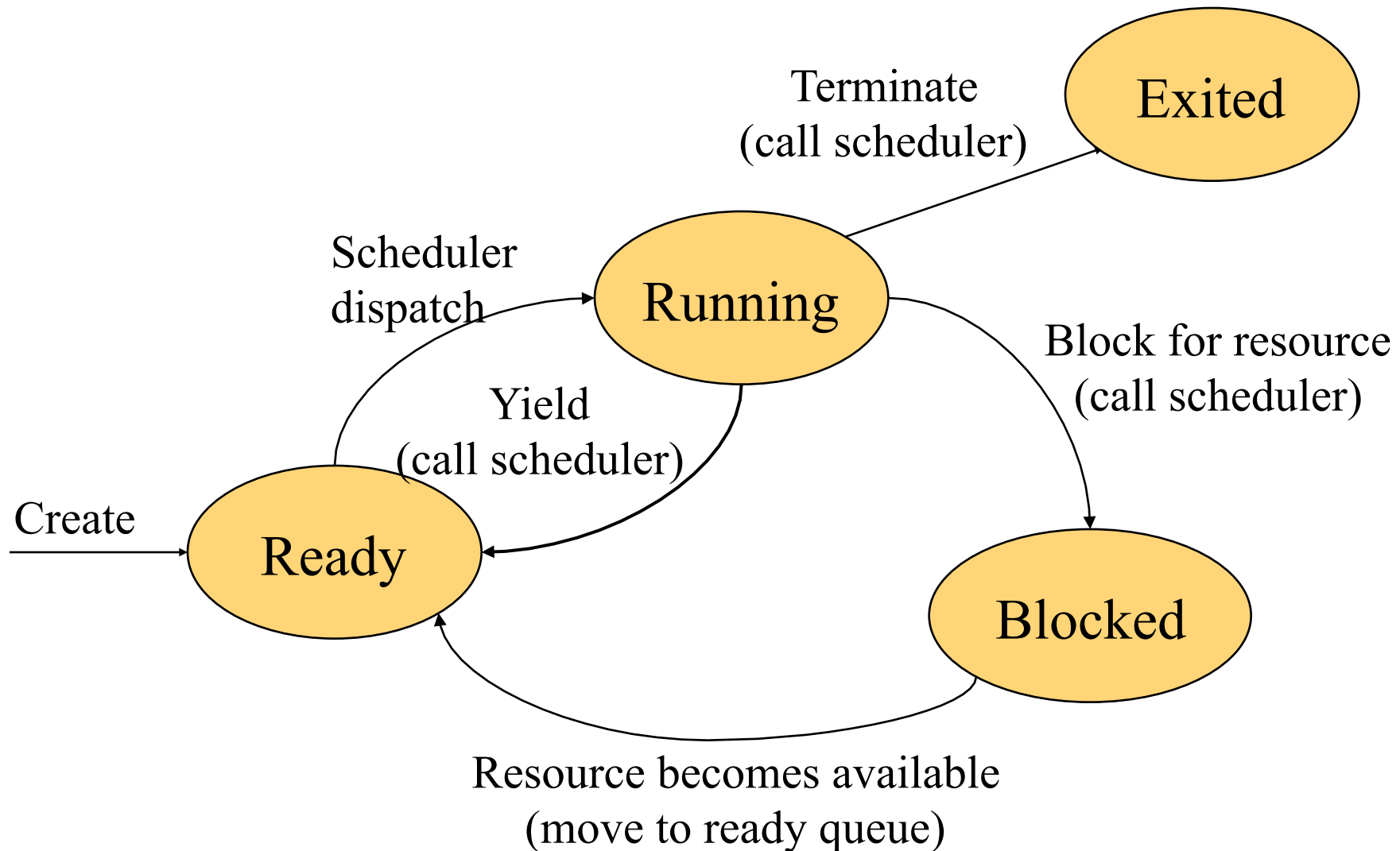


Revisit Monolithic OS Structure

- ◆ 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 the same address space (thread context switch)
 - Schedule in a new 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)

- ◆ Does this work?



More on Scheduler

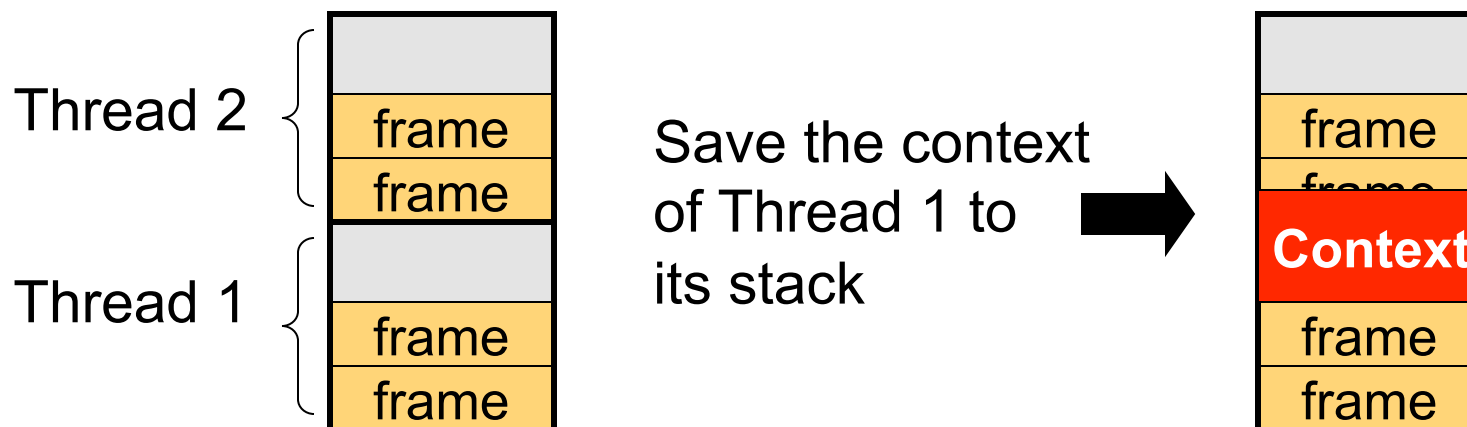
- ◆ Should the scheduler use a special stack?

- ◆ Should the scheduler simply be a 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 by I/O and Timer Interrupts

◆ Why

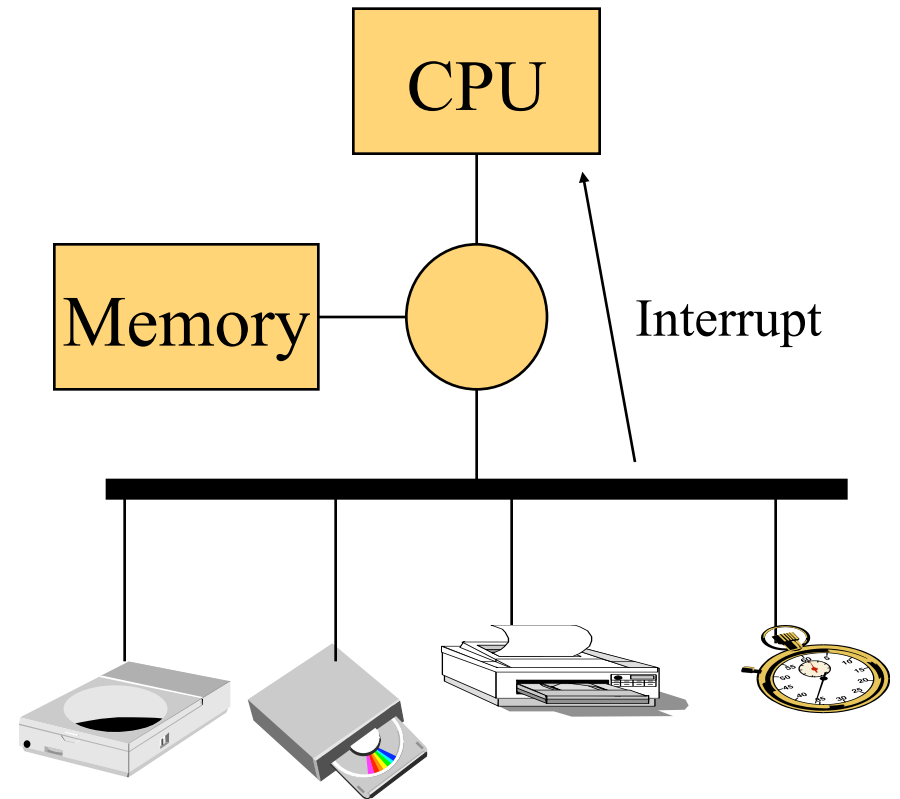
- Timer interrupt to help CPU management
- Asynchronous I/O to overlap with computation

◆ Interrupts

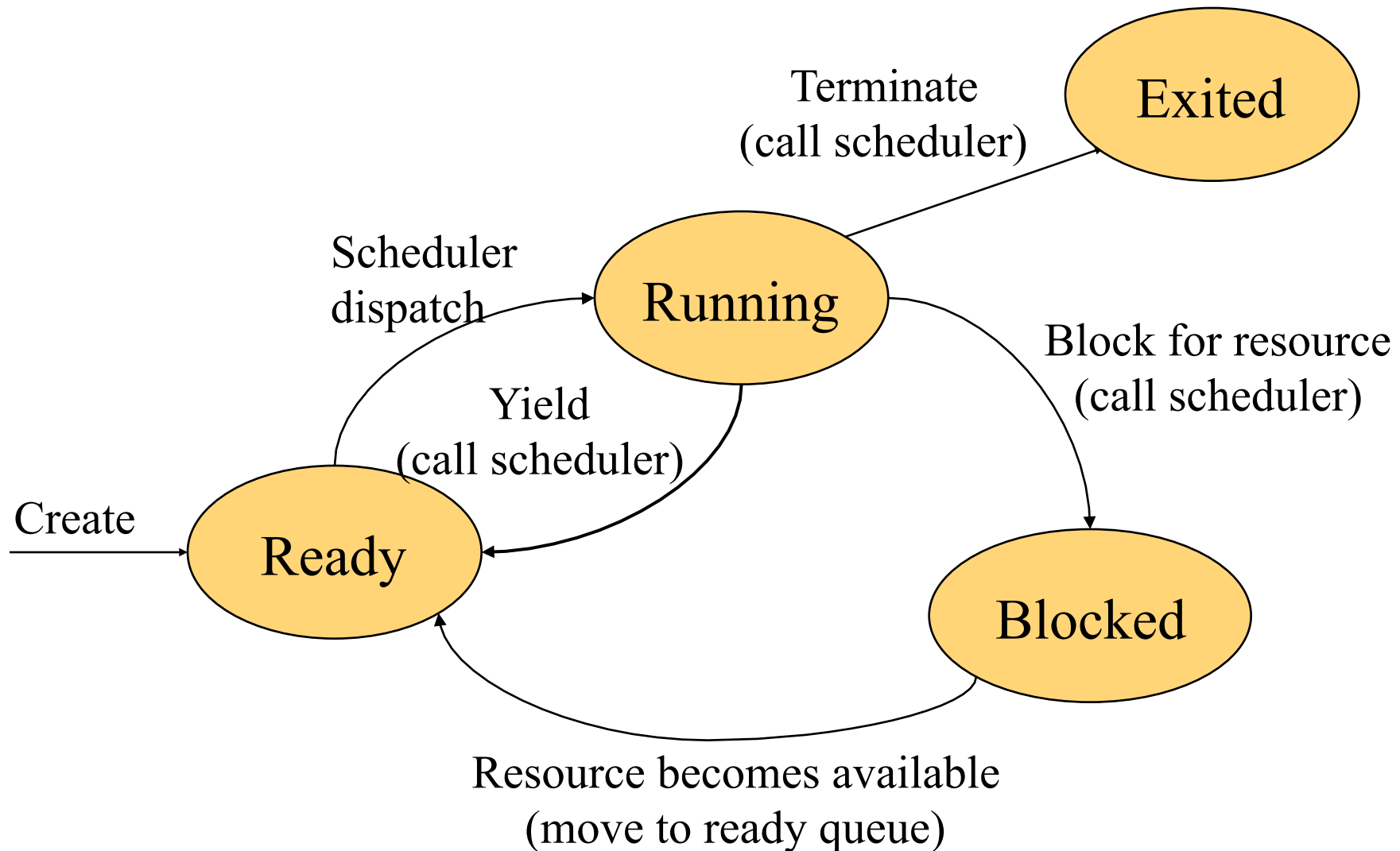
- Between instructions
- Within an instruction except atomic ones

◆ Manipulate interrupts

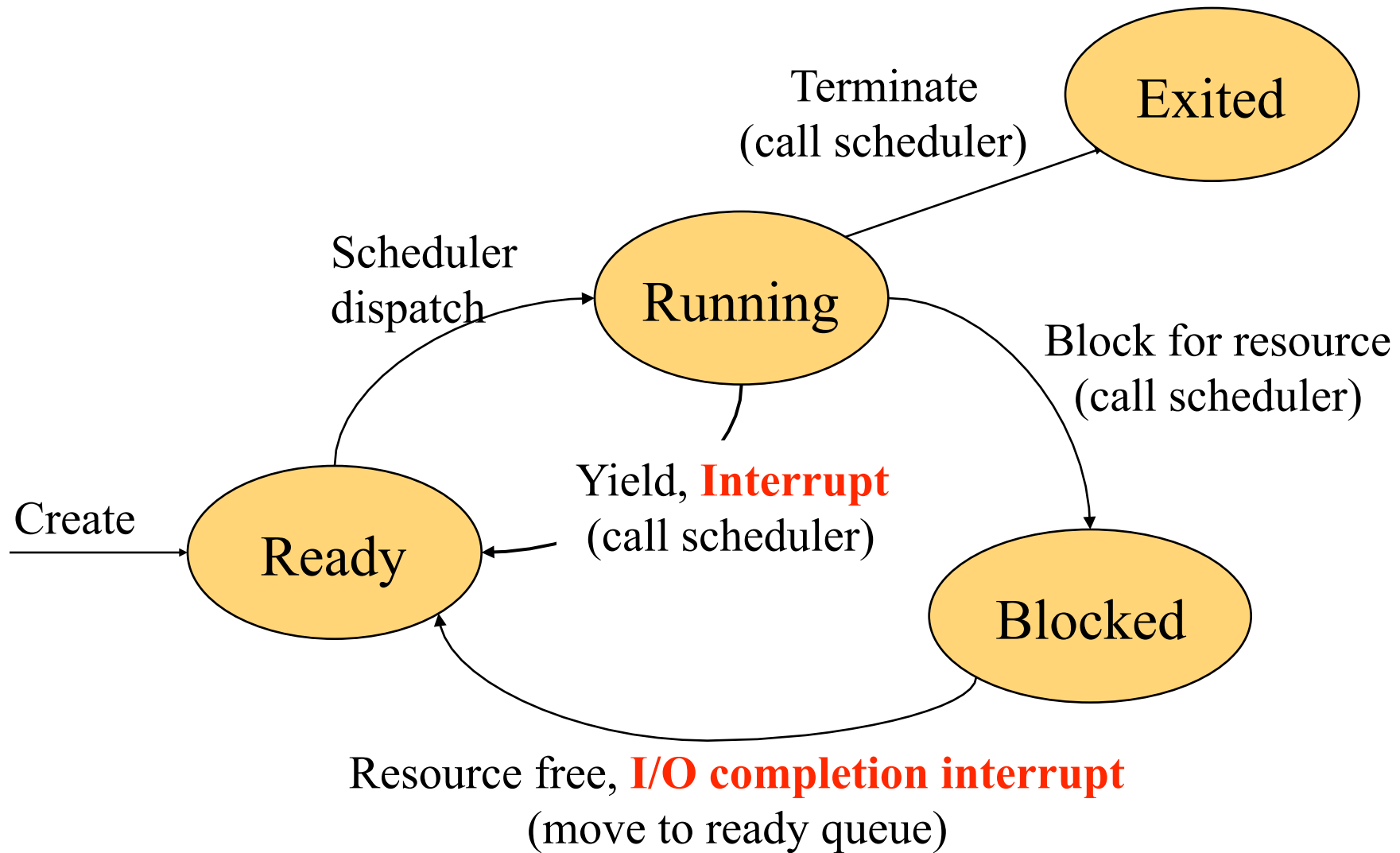
- Disable (mask) interrupts
- Enable interrupts
- Non-Masking Interrupts



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
 - ... (What to do here?)
 - Call scheduler
- ◆ Other 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



Dealing with Preemptive Scheduling

◆ Problem

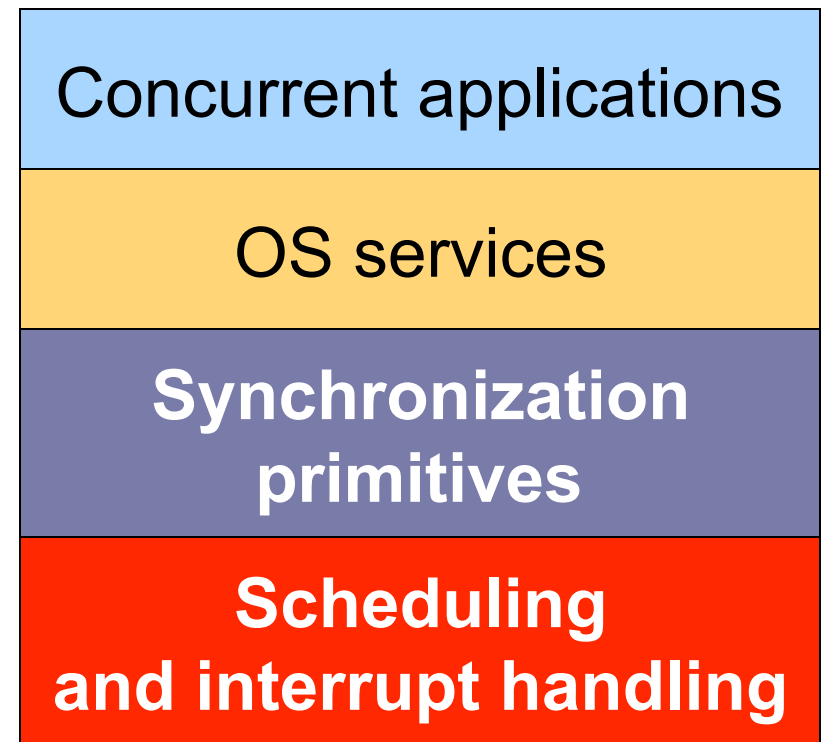
- Interrupts can happen anywhere

◆ An obvious approach

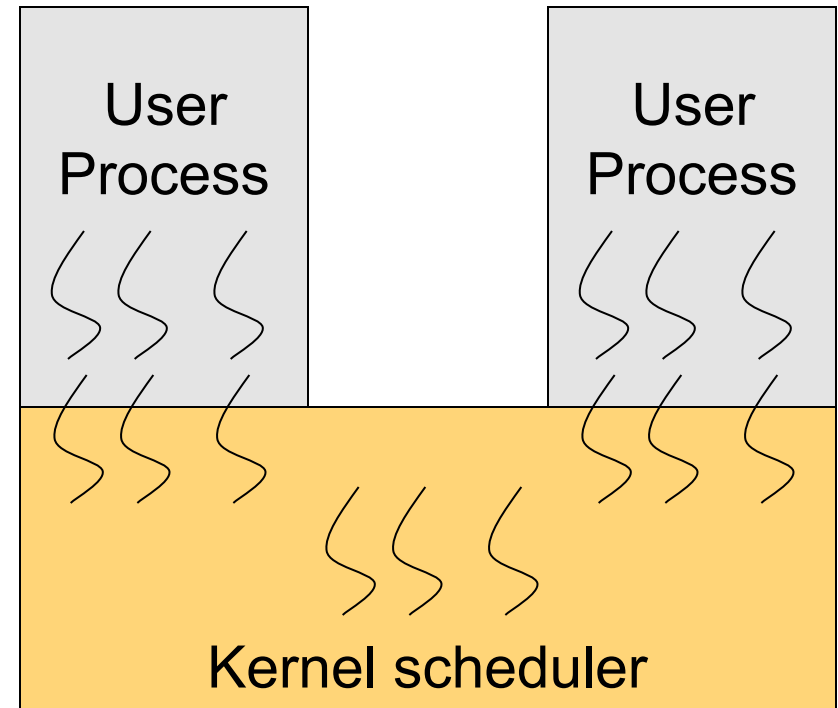
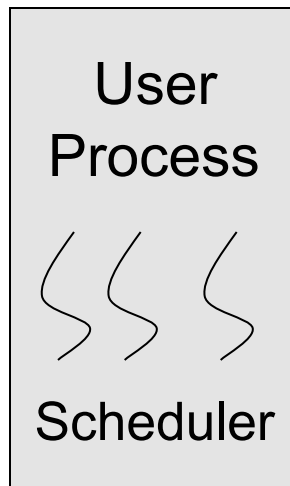
- Worry about interrupts and preemptions all the time

◆ What we want

- Worry less all the time
- Low-level behavior encapsulated in “primitives”
- Synchronization primitives worry about preemption
- OS and applications use synchronization primitives



User Threads vs. Kernel Threads



- ◆ Context switch at user-level without a system call (Java threads)
- ◆ Is it possible to do 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 using codes like co-routines
- Timer interrupt (signal facility) can introduce preemption
- When a user-level thread is blocked on an I/O event, the whole process is blocked

◆ 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

◆ Two approaches

- 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 Much Milk” Problem

- ◆ Do not want to buy too much milk
- ◆ Any person can be distracted at any point

	Student A	Student B
15:00	Look at fridge: out of milk	
15:05	Leave for Wawa	
15:10	Arrive at Wawa	Look at fridge: out of milk
15:15	Buy milk	Leave for Wawa
15:20	Arrive home; put milk away	Arrive at Wawa
15:25		Buy milk
		Arrive home; put milk away Oh No!



Using A Note?

Thread A

```
if ( noMilk ) {  
    if (noNote) {  
        leave note;  
        buy milk;  
        remove note;  
    }  
}
```

Thread B

```
if ( noMilk ) {  
    if (noNote) {  
        leave note;  
        buy milk;  
        remove note;  
    }  
}
```



- ◆ Any issue with this approach?

Another Possible Solution?

Thread A

```
leave noteA
if (noNoteB) {
  if (noMilk)
    buy milk
}
remove noteA
```



Didn't buy milk

Thread B

```
leave noteB
if (noNoteA)
  if (noMilk)
    buy milk
}
remove noteB
```



Didn't buy milk

- ◆ Does this method work?

Yet Another Possible Solution?

Thread A

```
leave noteA
while (noteB)
    do nothing;
if (noMilk)
    buy milk;
remove noteA
```

Thread B

```
leave noteB
if (noNoteA) {
    if (noMilk) {
        buy milk
    }
}
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
- ◆ Peterson's solution is also complex
- ◆ What we want is:

```
Acquire(lock) ;  
if (noMilk)  
    buy milk ;  
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 much milk problem
 - What we want is mutual exclusion

