# COS 318: Operating Systems

# Non-Preemptive and Preemptive Threads

Prof. Margaret Martonosi
Computer Science Department
Princeton University

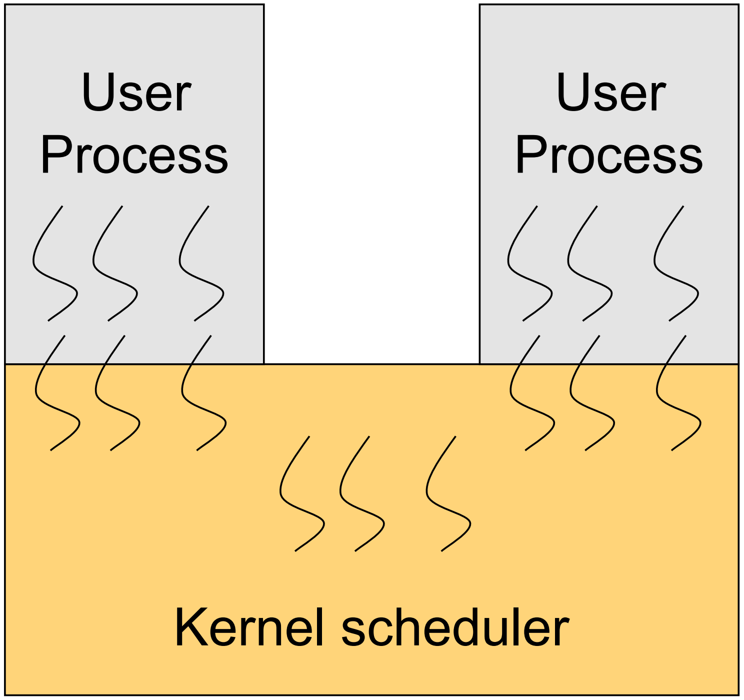
http://www.cs.princeton.edu/courses/archive/fall11/cos318

# Today’s Topics

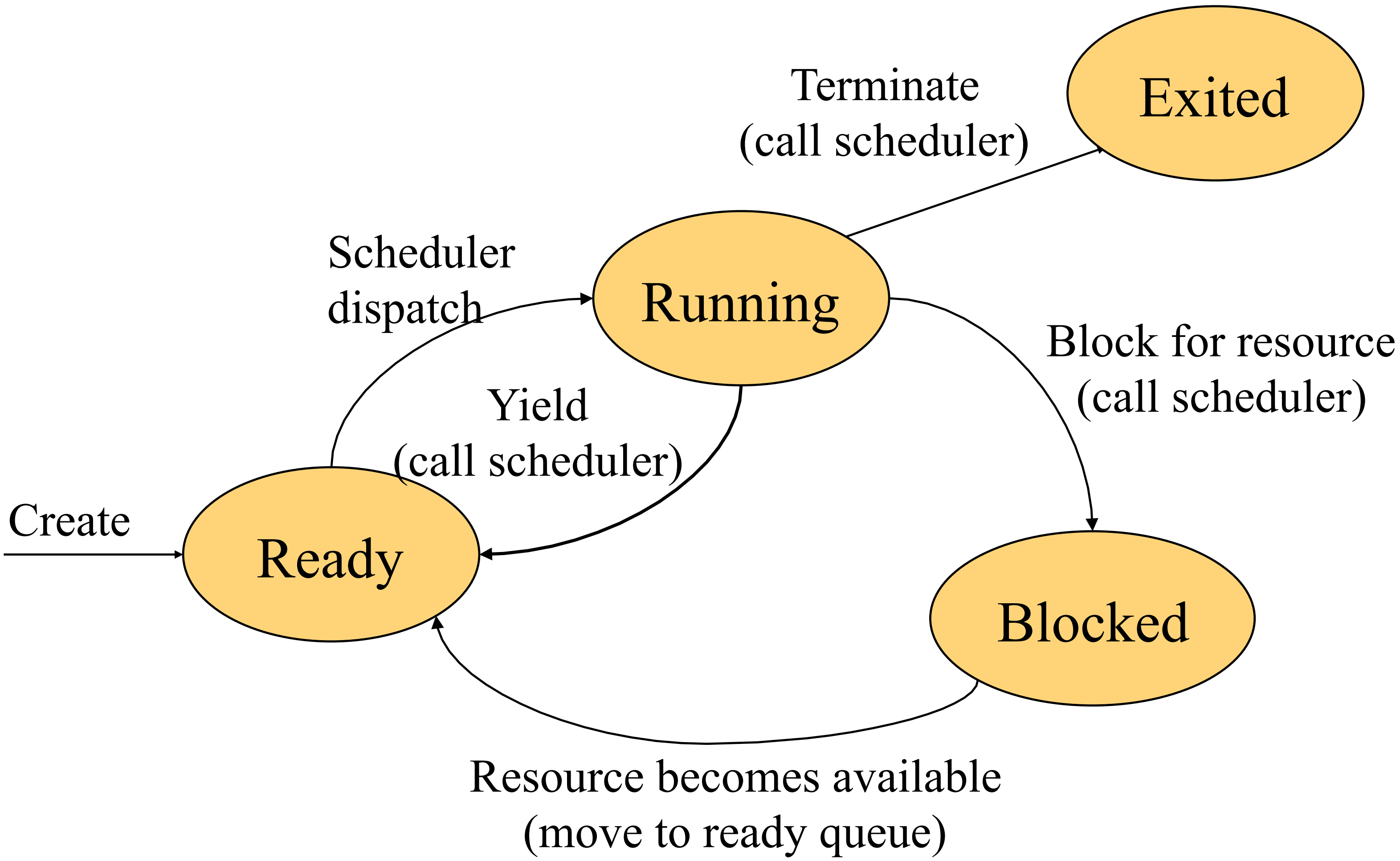
- Non-preemptive threads
- Preemptive threads
- Kernel vs. user threads
- Too much milk problem

# 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



Terminate
(call scheduler)

Exited

Scheduler
dispatch

Running

Block for resource
(call scheduler)

Yield
(call scheduler)

Create

Ready

Blocked

Resource becomes available
(move to ready queue)

# Scheduler

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

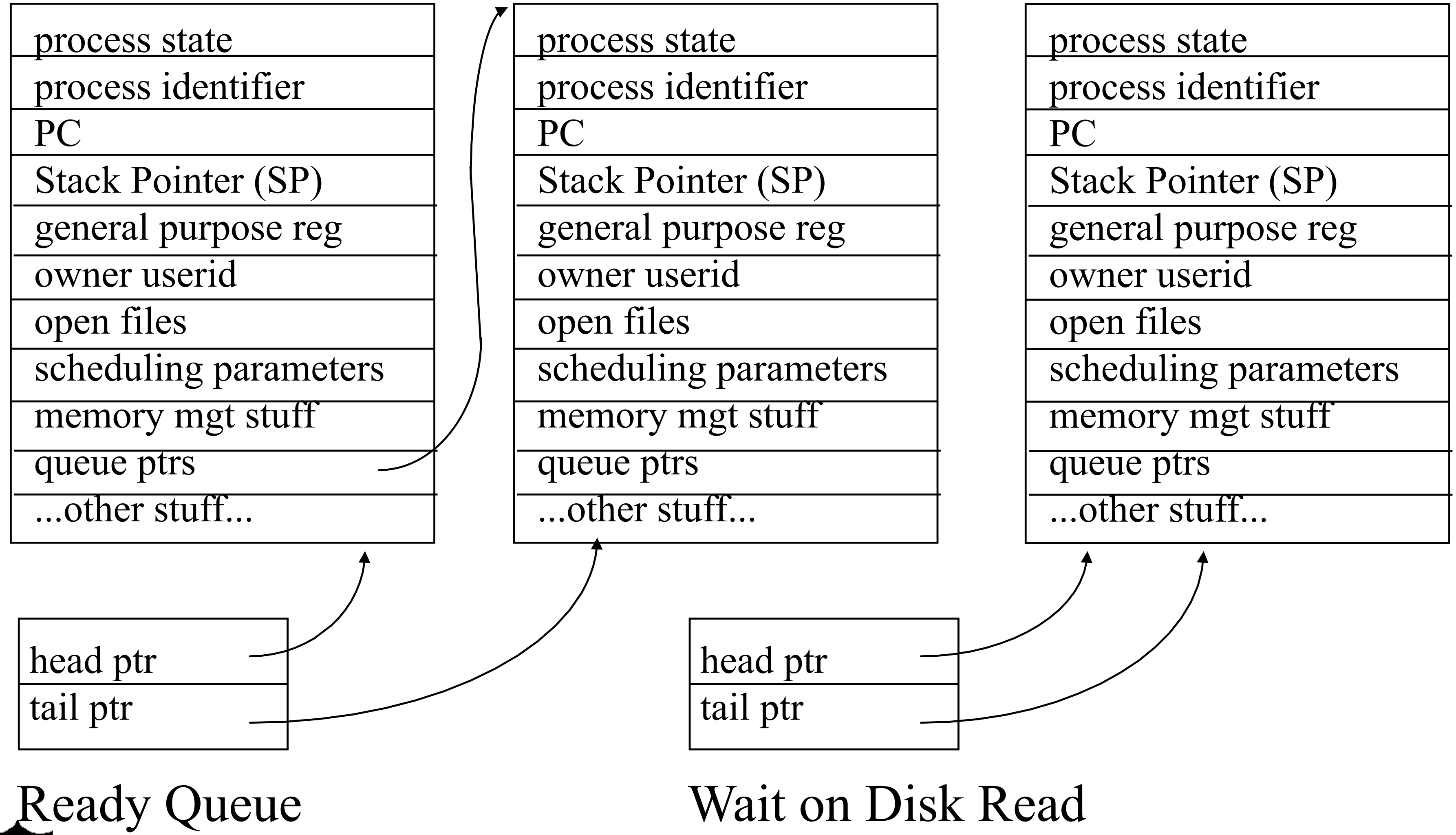
- The simplest form

  Scheduler:

  ```
      save current process/thread state
      choose next process/thread to run
      dispatch (load PCB/TCB and jump to it)
  ```

- Does this work?

# PCBs & Queues



process state
process identifier
PC
Stack Pointer (SP)
general purpose reg
owner userid
open files
scheduling parameters
memory mgt stuff
queue ptrs
...other stuff...

process state
process identifier
PC
Stack Pointer (SP)
general purpose reg
owner userid
open files
scheduling parameters
memory mgt stuff
queue ptrs
...other stuff...

process state
process identifier
PC
Stack Pointer (SP)
general purpose reg
owner userid
open files
scheduling parameters
memory mgt stuff
queue ptrs
...other stuff...

head ptr
tail ptr

Ready Queue

head ptr
tail ptr

Wait on Disk Read

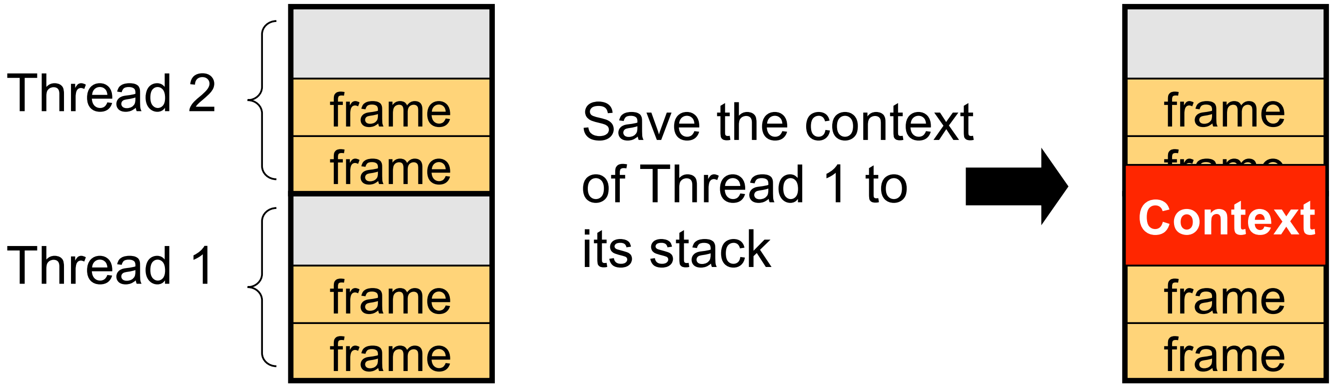
# More on Scheduler

- 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
  - 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
  - No overflow problems



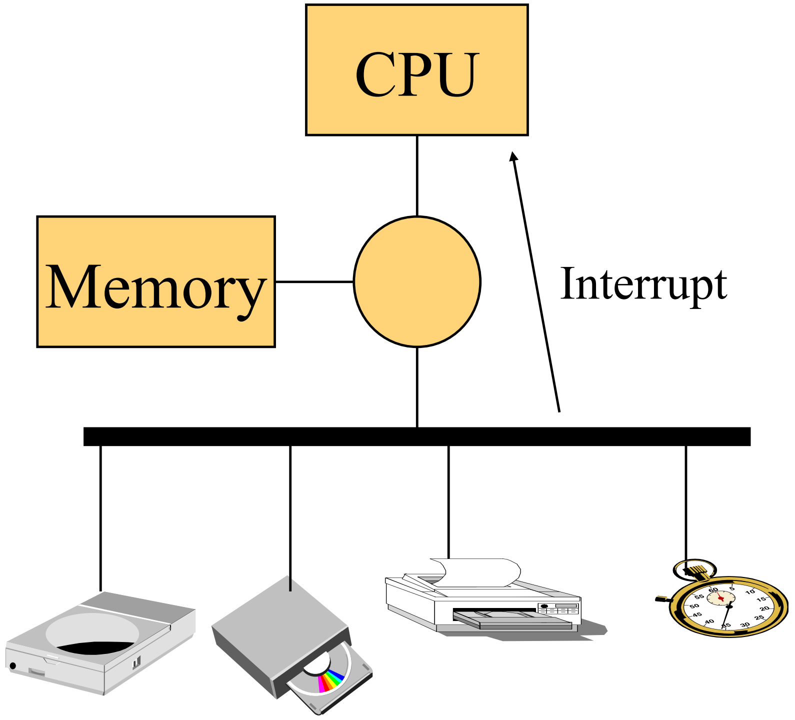
Save the context of Thread 1 to its stack

# Today’s Topics

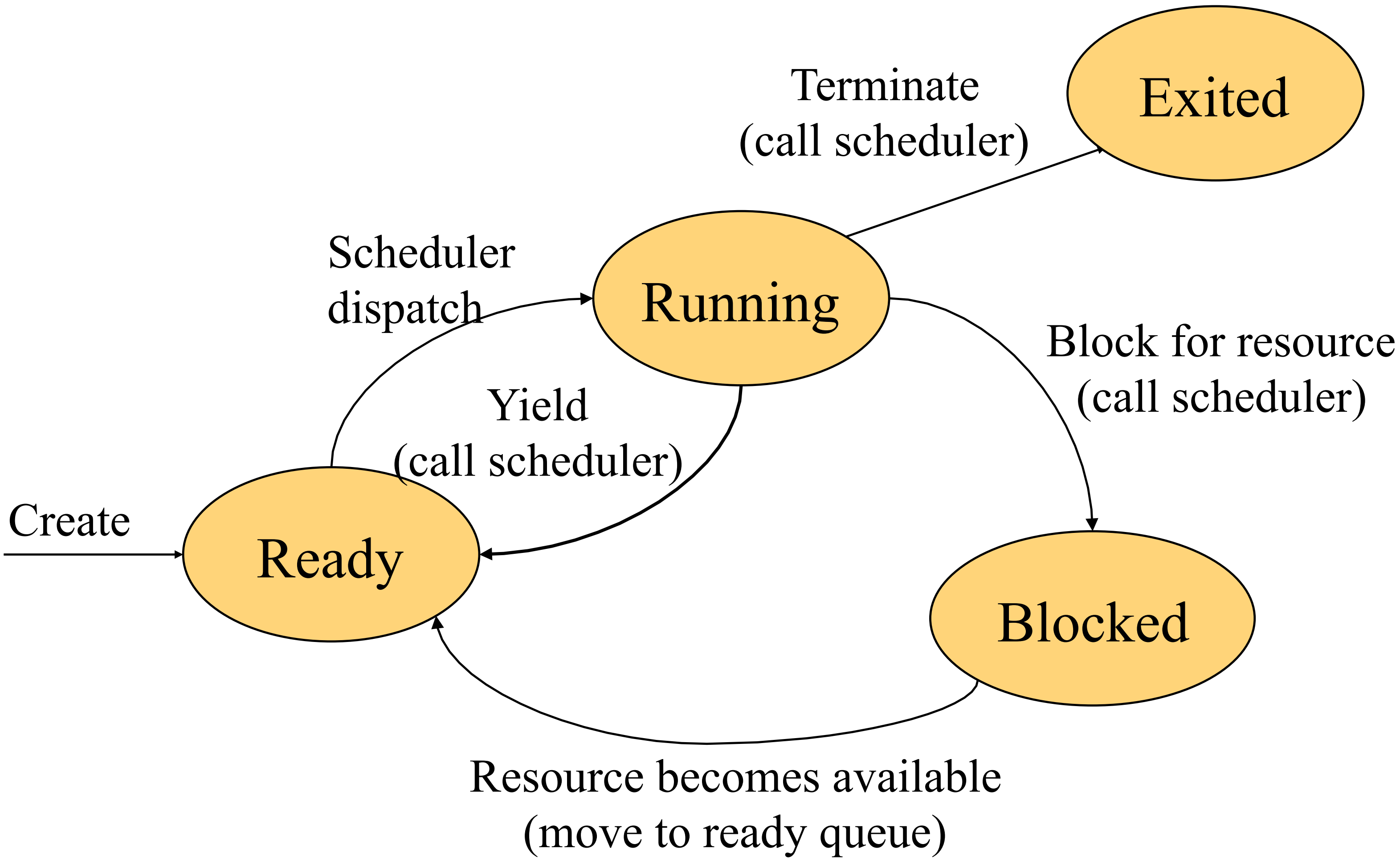
- Non-preemptive threads
- Preemptive threads
- Kernel vs. user threads
- Too much milk problem

# 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



Exited

Terminate
(call scheduler)

Scheduler
dispatch

Running

Block for resource
(call scheduler)

Yield
(call scheduler)

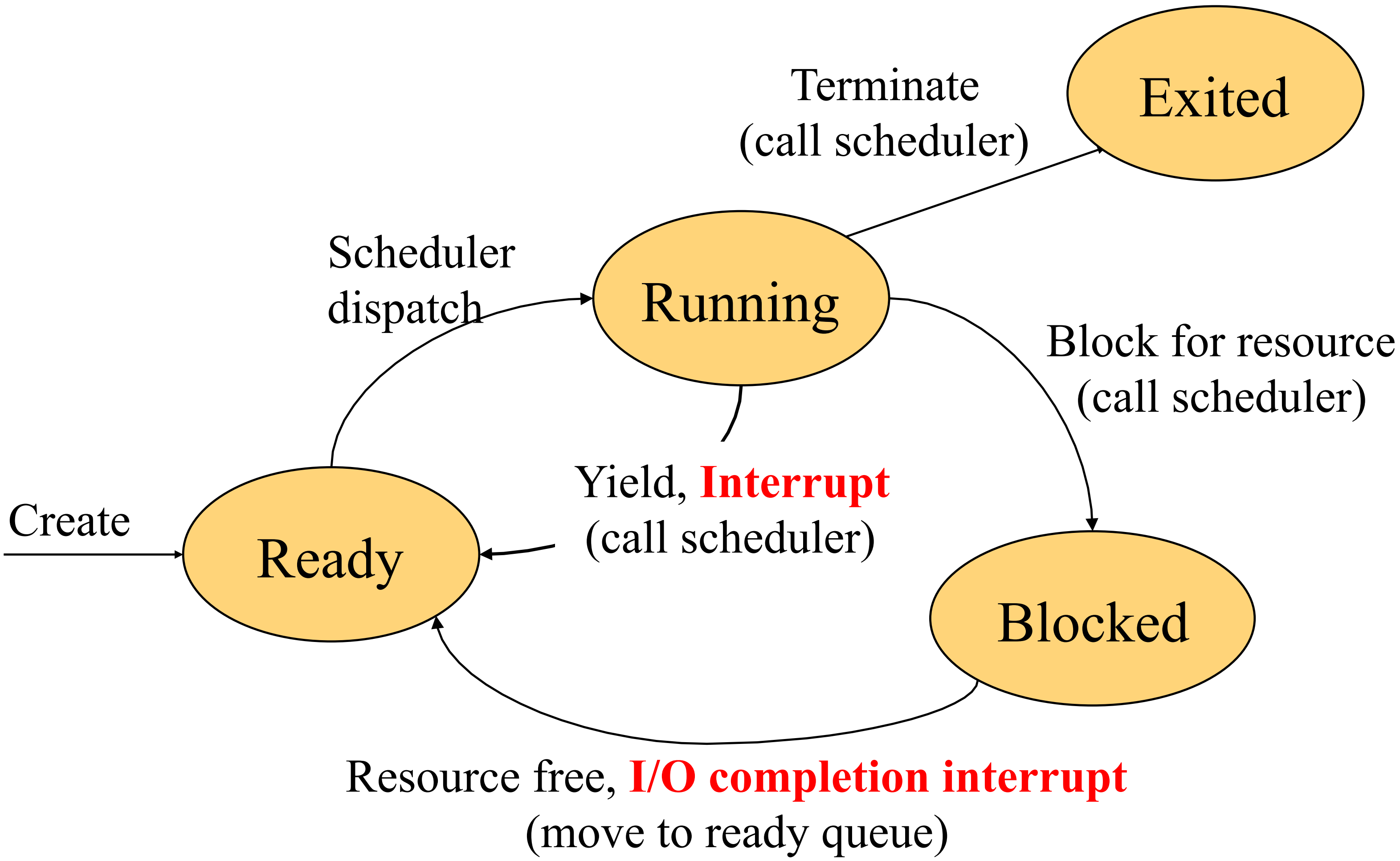
Create

Ready

Blocked

Resource becomes available
(move to ready queue)

# State Transition for Preemptive Scheduling



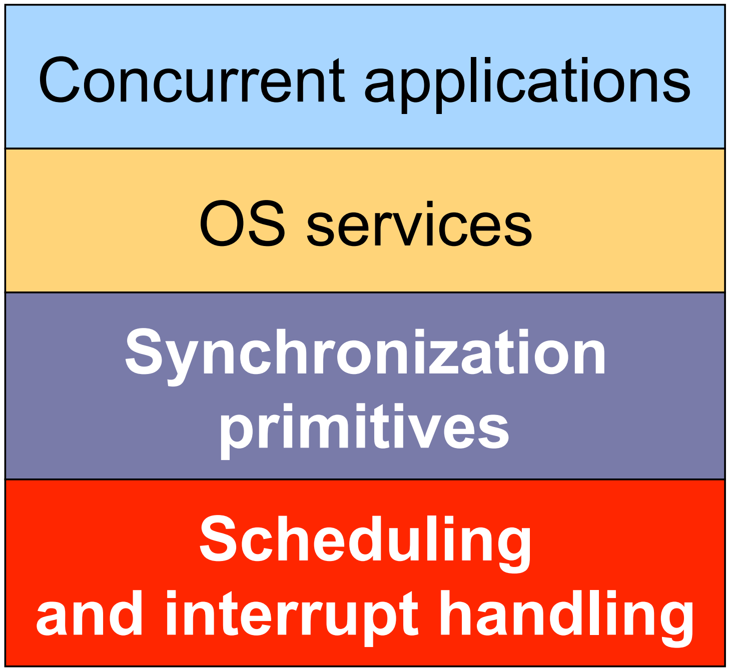
# Interrupt Handling for Preemptive Scheduling

- 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

- When to disable/enable interrupts?

# 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


Concurrent applications

OS services

**Synchronization primitives**

**Scheduling and interrupt handling**

# Preemption

- Scheduling policies may be *preemptive* or *non-preemptive*.
    - *Preemptive*: scheduler may unilaterally force a task to relinquish the processor before the task blocks, yields, or completes.
- *timeslicing* prevents jobs from monopolizing the CPU
    - Scheduler chooses a job and runs it for a *quantum* of CPU time.
    - A job executing longer than its quantum is forced to yield by scheduler code running from the clock interrupt handler.
- use preemption to honor priorities
    - Preempt a job if a higher priority job enters the *ready* state.

# Priority

- Some goals can be met by incorporating a notion of *priority* into a “base” scheduling discipline.
    - Each job in the ready pool has an associated priority value; the scheduler favors jobs with higher priority values.
- *External priority* values:
    - imposed on the system from outside
    - reflect external preferences for particular users or tasks
        - “All jobs are equal, but some jobs are more equal than others.”
    - *Example*: Unix **nice** system call to lower priority of a task.
    - *Example*: Urgent tasks in a real-time process control system.
- *Internal priorities*
    - scheduler dynamically calculates and uses for queuing discipline. System adjusts priority values internally as as an *implementation technique* within the scheduler.

# Internal Priority

- Drop priority of tasks consuming more than their share
- Boost tasks that already hold resources that are in demand
- Boost tasks that have starved in the recent past
- Adaptive to observed behavior: typically a continuous, dynamic, readjustment in response to observed conditions and events
  - May be visible and controllable to other parts of the system
  - Priority reassigned if I/O bound (large unused portion of quantum) or if CPU bound (nothing left)

# Keeping Your Priorities Straight

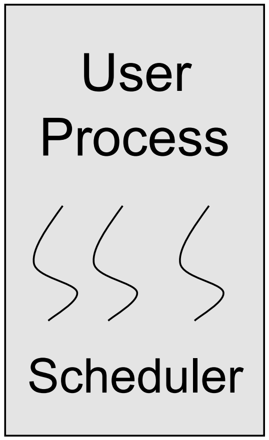
◆ Priorities must be handled carefully when there are dependencies among tasks with different priorities.

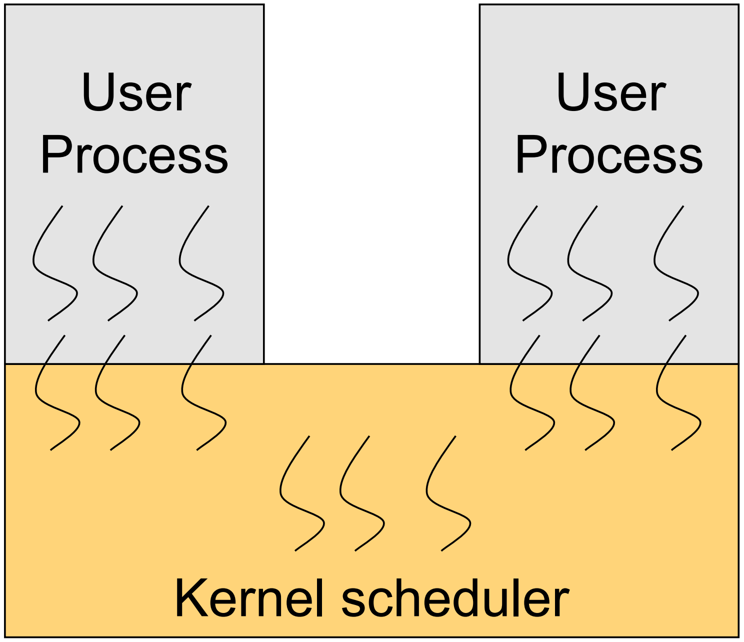
- A task with priority ***P*** should never impede the progress of a task with priority ***Q*** > ***P***.
  - This is called *priority inversion*, and it is to be avoided.
- The basic solution is some form of *priority inheritance*.
  - When a task with priority ***Q*** waits on some resource, the holder (with priority ***P)*** temporarily inherits priority ***Q*** if ***Q*** > ***P***.
  - Inheritance may also be needed when tasks coordinate with IPC.
- Inheritance is useful to meet deadlines and preserve low-jitter execution, as well as to honor priorities.

# Today’s Topics

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

# 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
  - 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 |

# Today’s Topics

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

# “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<br>Oh No! |

# “Too Much Milk” : A different interleaving

| | Student A | Student B |
|---|---|---|
| 15:00 | Look at fridge: out of milk | |
| 15:05 | Leave for Wawa | |
| 15:10 | Arrive at Wawa | |
| 15:15 | Buy milk | |
| 15:20 | Arrive home; put milk away | |
| 15:25 | | Look at fridge: plenty of milk |
| | | Yay! |

# “Too Much Milk”: A Third Interleaving

- 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 | Look at fridge: out of milk | Arrive at Wawa |
| 15:15 | Leave for Wawa | Buy milk |
| 15:20 | Arrive at Wawa | Arrive home; put milk away |
| 15:25 | Buy milk | |
| | Arrive home; put milk away<br>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?

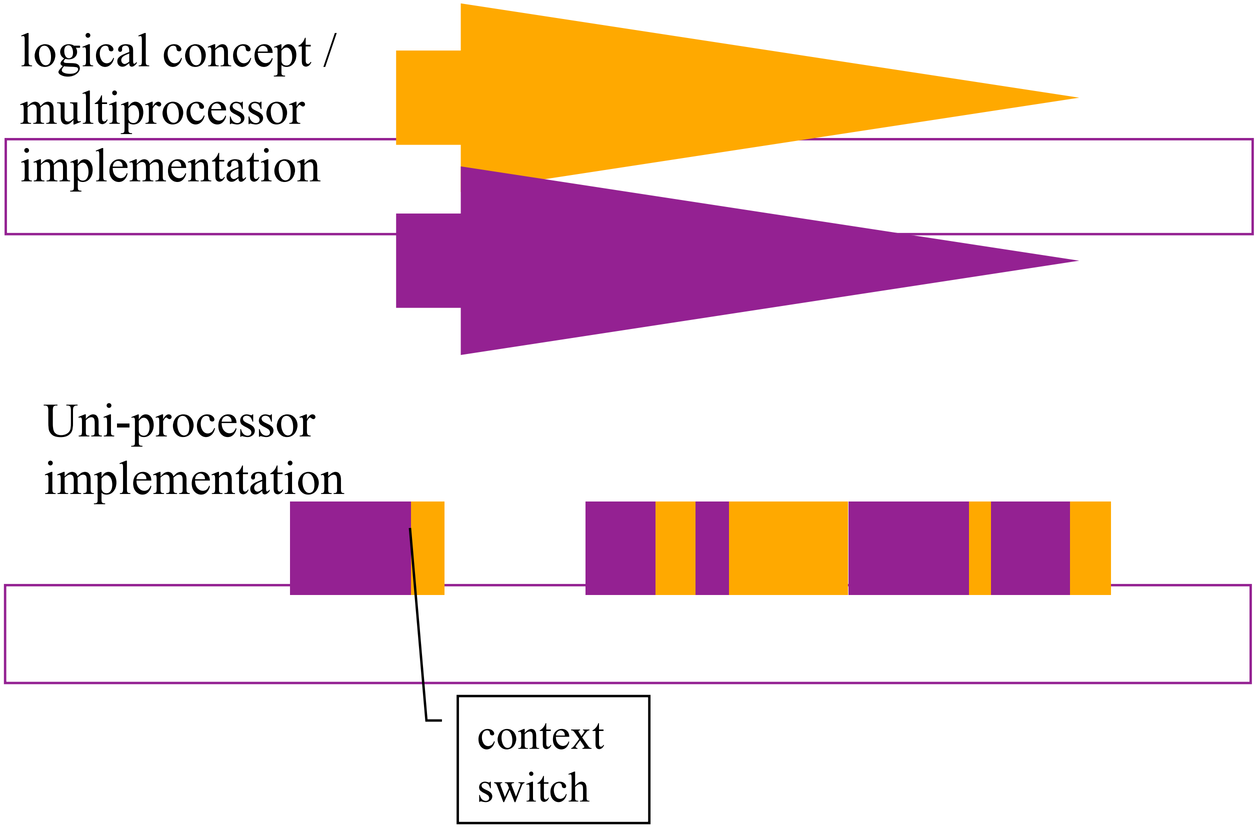
# Remarks

- 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 makes these scenarios hard to reason about is arbitrary interleaving.
- What we want is:

```
Acquire(lock);
if (noMilk)
  buy milk;
Release(lock);
```

**Critical section**

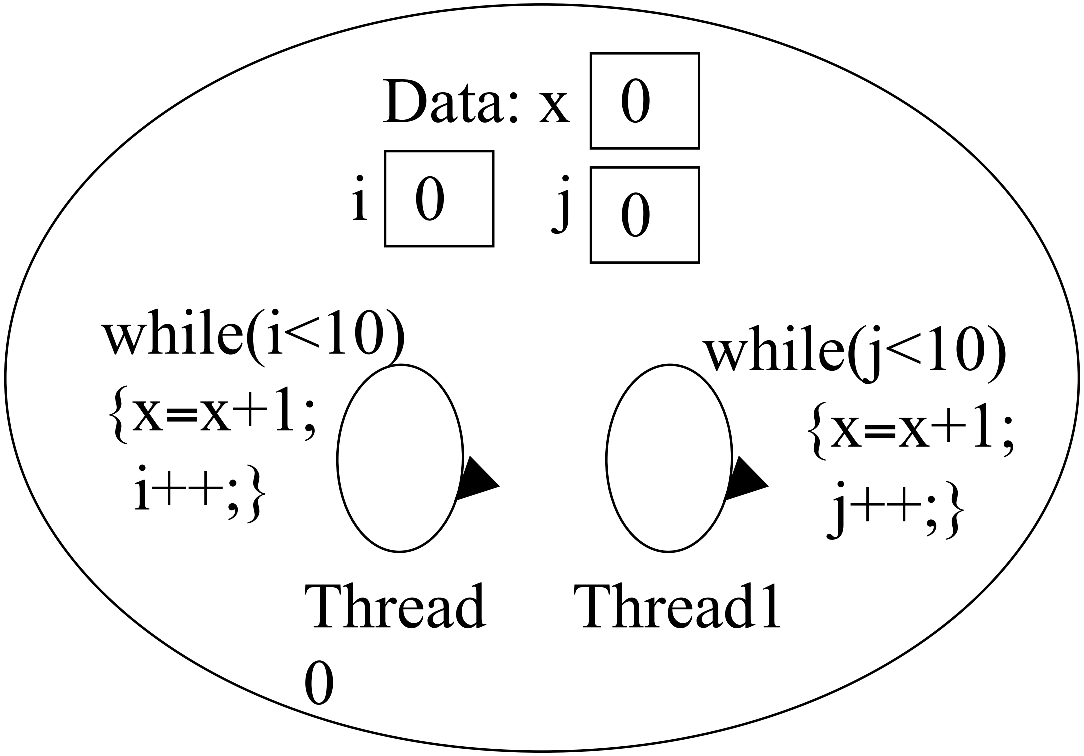
# Interleaved Schedules


logical concept / multiprocessor implementation

Uni-processor implementation

context switch

# The Trouble with Concurrency in Threads...



What is the value of x when both threads leave this while loop?

# Range of Answers

| Process 0 | Process1 |
|---|---|
| LD x // x currently 0 | |
| | LD x // x currently 0 |
| | Add 1 |
| | ST x // x now 1 |
| | Do 8 more full loops // x = 9 |
| Add 1 | |
| ST x // x now 1, stored over 9 | |
| | LD x // x now 1 |
| Do 9 more full loops // leaving x at 10 | |
| | Add 1 |
| | ST x // x = 2 stored over 10 |

# Nondeterminism

- What unit of work can be performed without interruption? ***Indivisible*** or ***atomic*** operations.
- ***Interleavings*** - possible execution sequences of operations drawn from all threads.
- ***Race condition*** - final results depend on ordering and may not be “correct”.

while (i<10) {x=x+1; i++;}

load value of x into reg

yield( )

add 1 to reg

yield ( )

store reg value at x

yield ( )

# Reasoning about Interleavings

- On a uniprocessor, the possible execution sequences depend on when context switches can occur
  - Voluntary context switch - the process or thread explicitly yields the CPU (blocking on a system call it makes, invoking a Yield operation).
  - Interrupts or exceptions occurring - an asynchronous handler activated that disrupts the execution flow.
  - Preemptive scheduling - a timer interrupt may cause an involuntary context switch at any point in the code.
- On multiprocessors, the ordering of operations on shared memory locations is the important factor.

# 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

# Pitfalls:
# Mars Pathfinder Example

- In July 1997, Pathfinder's computer reset itself several times during data collection and transmission from Mars.
  - One of its processes failed to complete by a deadline, triggering the reset.
- Priority Inversion Problem.
  - Low priority process was inside a critical section to write a shared data structure, but was preempted to let higher priority processes run.
  - The higher priority process was blocked waiting, and failed to complete in time.
  - Meanwhile a bunch of medium priority processes ran, until finally the deadline ran out. They kept low priority process inside critical section from running again to release.
  - Priority inheritance had not been enabled on critical sections.
    - Low-pri "becomes" if holding an important resource!