EECS 482
Introduction to Operating Systems

Winter 2018

Baris Kasikci

Slides by: Harsha V. Madhyastha

Use of CVs in Project 1

- Incorrect use of condition variables:

```
while (cond) {
    cv.signal()
    cv.wait()
}
```

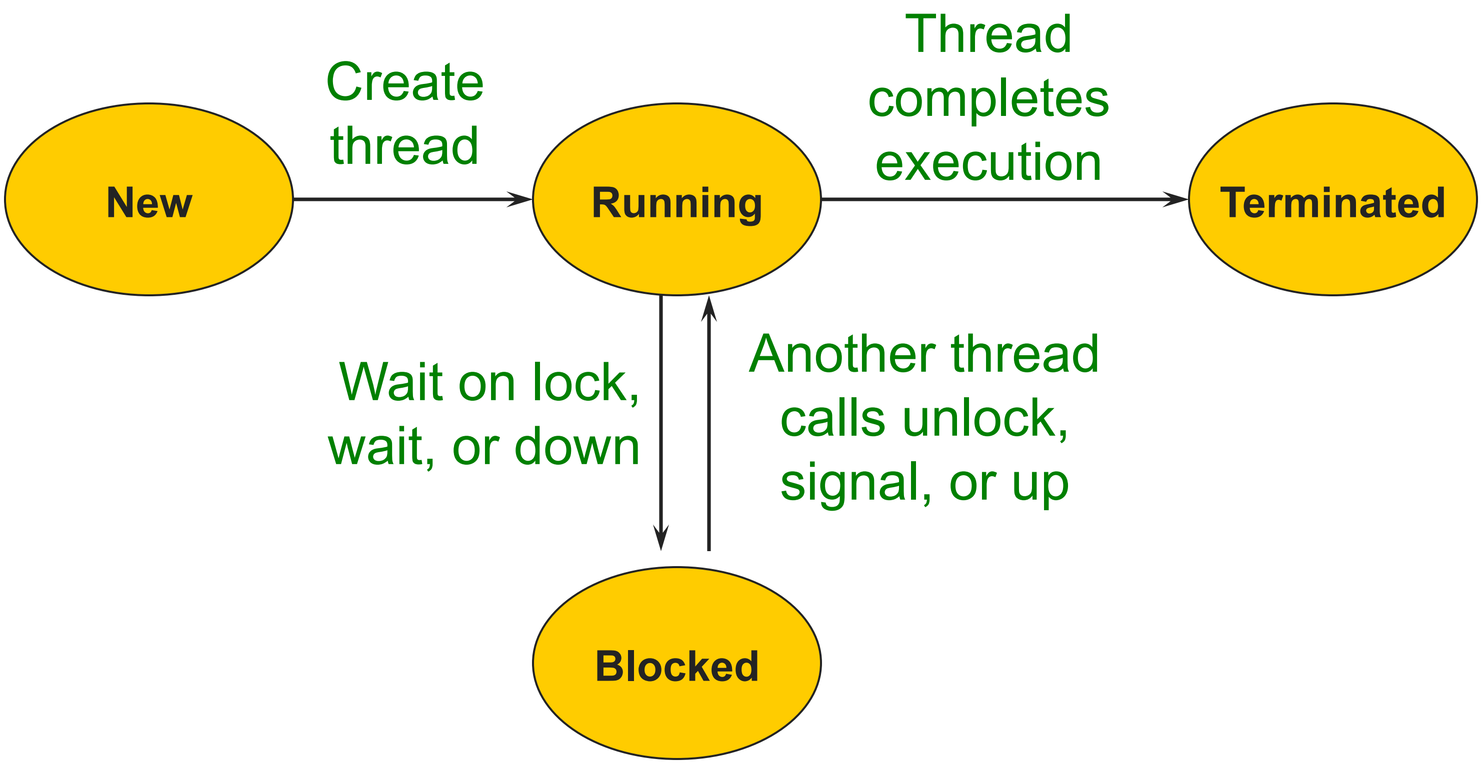
- Thread going to sleep should not be of interest to other threads

Interactions between threads

- Threads must synchronize access to shared data
- High-level synchronization primitives:
  - Locks
  - Condition variables
  - Monitors
  - Semaphores

- Threads share the same CPU
- Then what is a non-running thread?

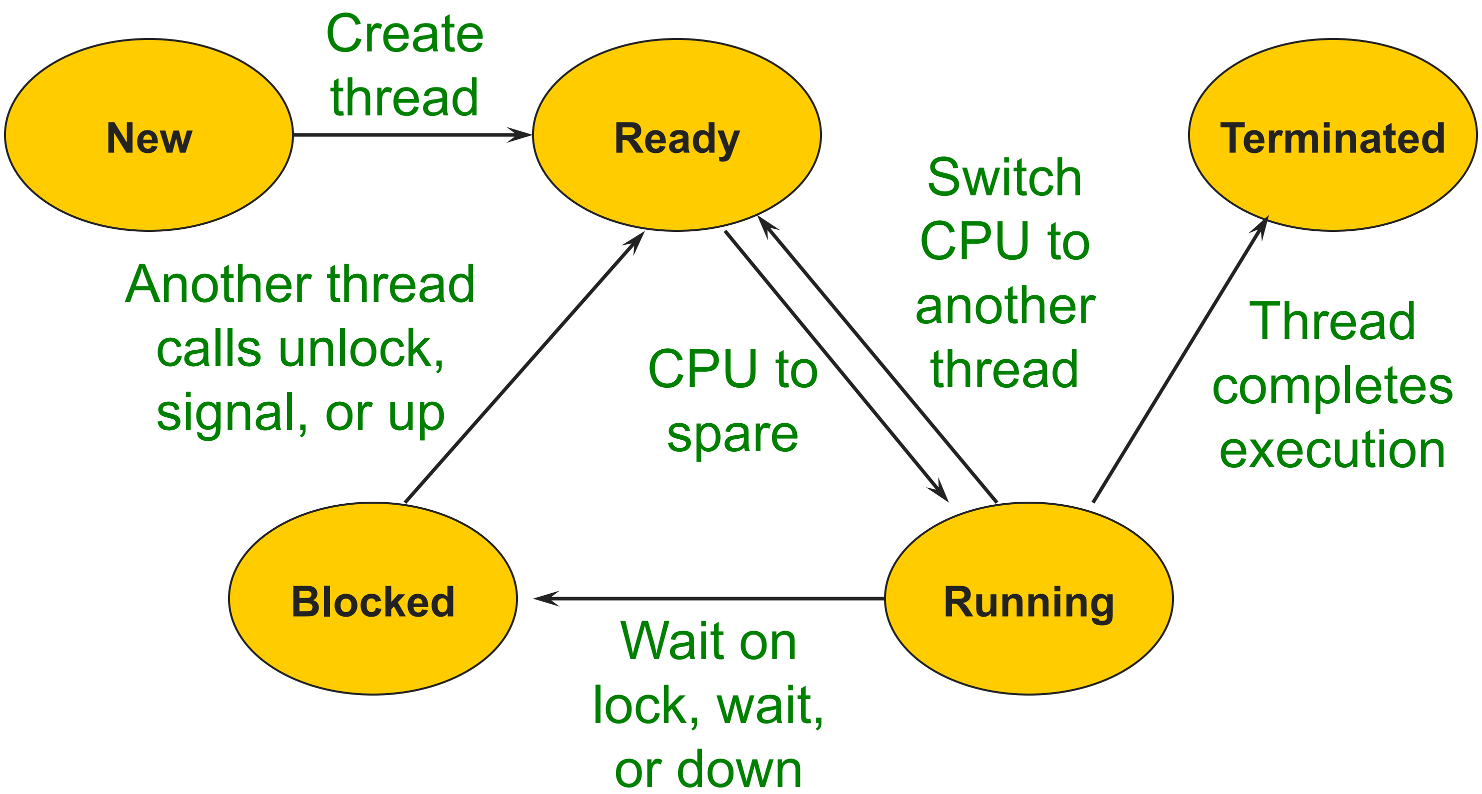
States of a Thread



What if there are more threads than CPUs?

States of a Thread

Why no transition from Ready to Blocked?



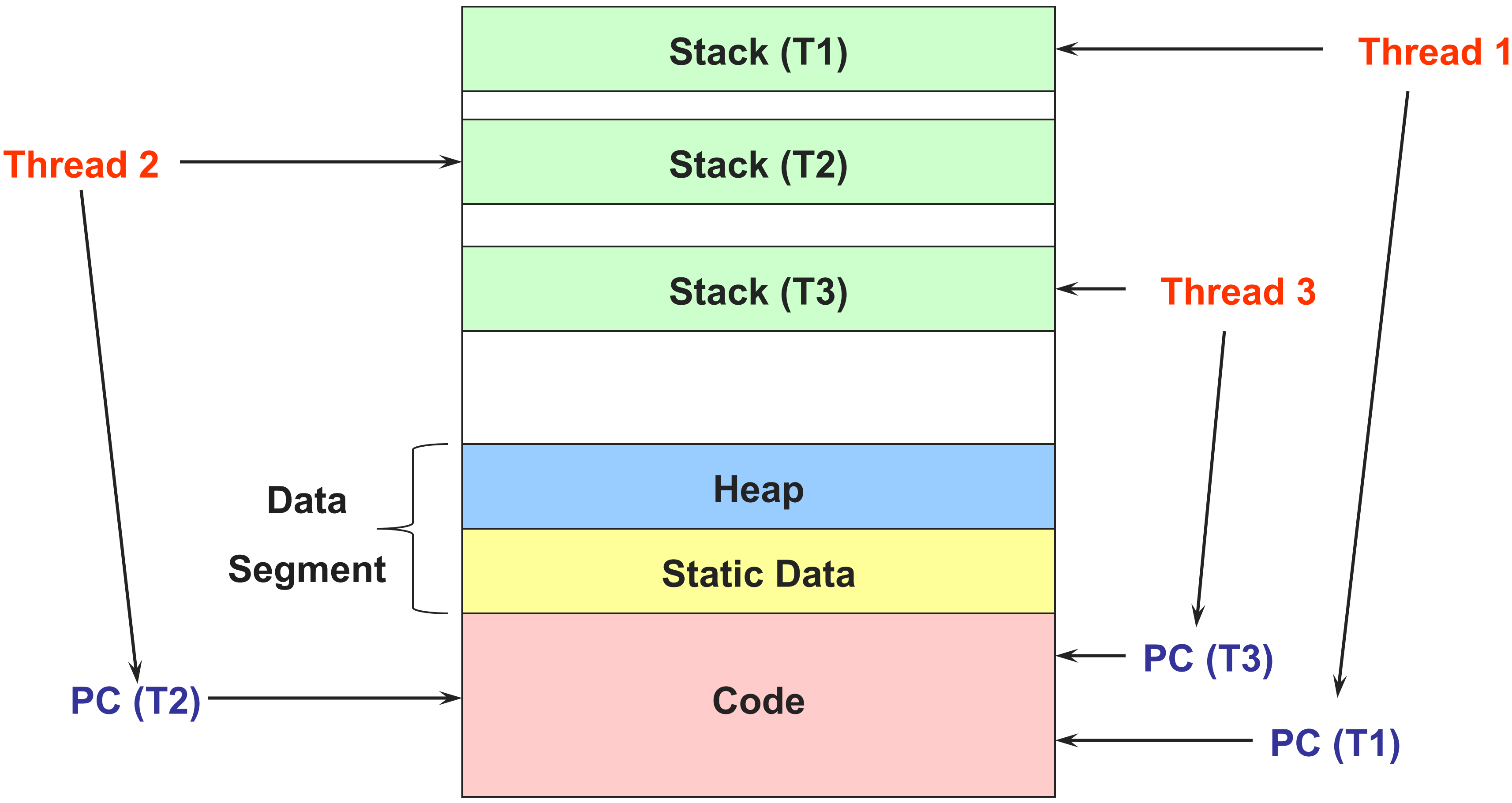
Ready threads

- What to do with thread while it's not running?
  - Must save its private state somewhere

- Thread "context" stored in a "thread control block" (TCB) when thread isn't running

- What should be stored in TCB?

Process Address Space



Thread context

- To save space in TCB
  - Share code among all threads and store only PC
  - Use multiple stacks and copy only SP to TCB
- Keep track of ready threads (e.g., on a queue)
- Any thread can be in one of three states
  - Running on the CPU
  - TCB is in ready queue
  - Blocked: TCB is in waiting queue of synchronization primitive

Project 2 is out

- Implement a thread library
  - Create threads
  - Switch between threads
  - Manage interactions (locks and CVs)
  - Schedule threads on CPUs
- Due Feb 17th
  - Start early!
- Everyone should now be in a group

Two Perspectives to Execution

- Thread view:
  - Running → Paused → Resume
- CPU view:
  - Thread 1 → Thread 2 → Thread 1

Steps in Switching threads

- Current thread returns (yields) control to OS
- OS chooses next thread to run
- OS saves state of current thread from CPU to its thread control block
- OS loads context of next thread from its thread control block
- OS runs next thread

How does thread return control back to OS?

Returning control to OS

- Three types of internal events:
  - Thread calls wait(), lock(), etc.
  - Thread requests OS to do some work (e.g., I/O)
  - Thread voluntarily gives up CPU with yield()
- Are these enough?
- Also need external events:
  - Interrupts are hardware events that transfer control from CPU to OS's interrupt handler

Steps in Switching threads

- Current thread returns control to OS
- OS chooses next thread to run
- OS saves state of current thread from CPU to its thread control block
- OS loads context of next thread from its thread control block
- OS runs next thread

How does the OS choose the next thread to run?

Choosing next thread to run

- 1 ready thread
  - What if thread calls yield?
- >1 ready thread
  - FIFO
  - Priority
- What should CPU do if no ready threads?
  - Modern CPUs suspend their execution and resume on an interrupt
  - *interrupt_enable_suspend()* in Project 2

Steps in Switching threads

- Current thread returns control to OS
- OS chooses new thread to run
- OS saves state of current thread from CPU to its thread control block
- OS loads context of next thread from its thread control block
- OS runs next thread

How do you save the state of the current thread?

Saving state of current thread

- Save registers, PC, stack pointer
- Tricky to get right!
  - Why won't the following code work?

```
100    save PC
101    switch to next thread
```

- Involves tricky assembly-language code
- In Project 2, we'll use Linux's *swapcontext()*

Steps in Switching threads

- Current thread returns control to OS
- OS chooses new thread to run
- OS saves state of current thread from CPU to its thread control block
- OS loads context of next thread from its thread control block
- OS runs next thread

How do you load the TCB of the next thread and run?

Loading context of next thread and running it

- How to load registers?
- How to load stack?
- How to resume execution?
- Who is carrying out these steps?
- How does thread that gave up control run again?

Example of thread switching

```
Thread 1
      print “start thread 1”
      yield()
      print “end thread 1”
Thread 2
      print “start thread 2”
      yield()
      print “end thread 2”
yield()
      print “start yield: thread %d”
      switch to next thread (swapcontext)
      print “end yield: thread %d”
```

Thread 1 output	Thread 2 output
start thread 1	
start yield: thread 1	
	start thread 2
	start yield: thread 2
end yield: thread 1	
end thread 1	
	end yield: thread 2
	end thread 2