Based on slides by Harsha V. Madhyastha

EECS 482 Introduction to Operating Systems

Spring/Summer 2020

Lecture 7: Thread implementation

Nicole Hamilton
https://web.eecs.umich.edu/~nham/
nham@umich.edu

Project 2

Make sure to start right away on project 2

Differences compared to project 1:

- Much harder!
- 15% of grade
- Test cases will be graded
- Hand grading to check for good coding practices, efficiency, no duplication of code, etc.

Monitors vs. Semaphores

Monitors:

+ Custom user-defined conditions

  Stateless, signals only wake up threads that are waiting

- Developer must use a cv in combination with a lock
- Must always check the condition on wakeup

Semaphores:

+ Access to the value is thread-safe

  State maintained as a positive integer value

- Only condition is *"(value == 0)"*
- Must map the user-defined condition to 0 or more than 0

How to implement custom waiting condition with semaphores?

Implementing condition variables with semaphores

```
queue waiters = {};   // list of semaphores inserted by waiting threads


void wait( mutex *m )
   {
   // create a new semaphore
   semaphore s = 0;

   // add new semaphore to
   // waiting list
   waiters.insert( &s );

   m->up( );
   // go to sleep
   s.down( );
   m->down( );
   }
```

Not atomic

```
void signal( )
   {
   // nothing to do if no waiters
   if ( waiters.empty( ) )
      return;

   // wake up one of the waiters
   semaphore s = waiters.front( );
   s.up( );

   // remove waiter from queue
   waiters.pop( );
   }
```

Exercise to try …

Given implementations of mutex and condition variable, how to implement a semaphore?

Interactions between threads

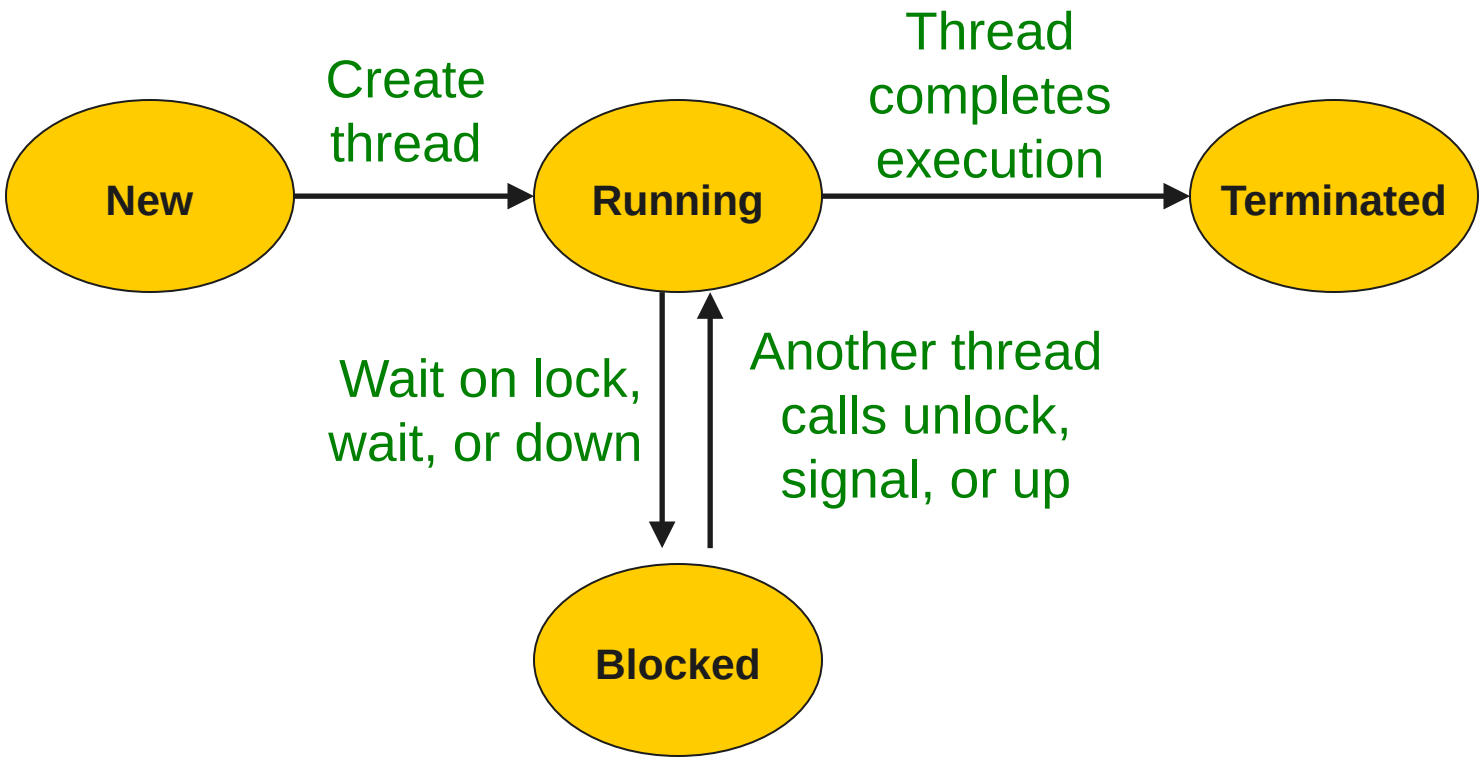
Threads must synchronize access to shared data

High-level synchronization primitives:

- Locks
- Condition variables
- Monitors
- Semaphores

Threads share the same CPU

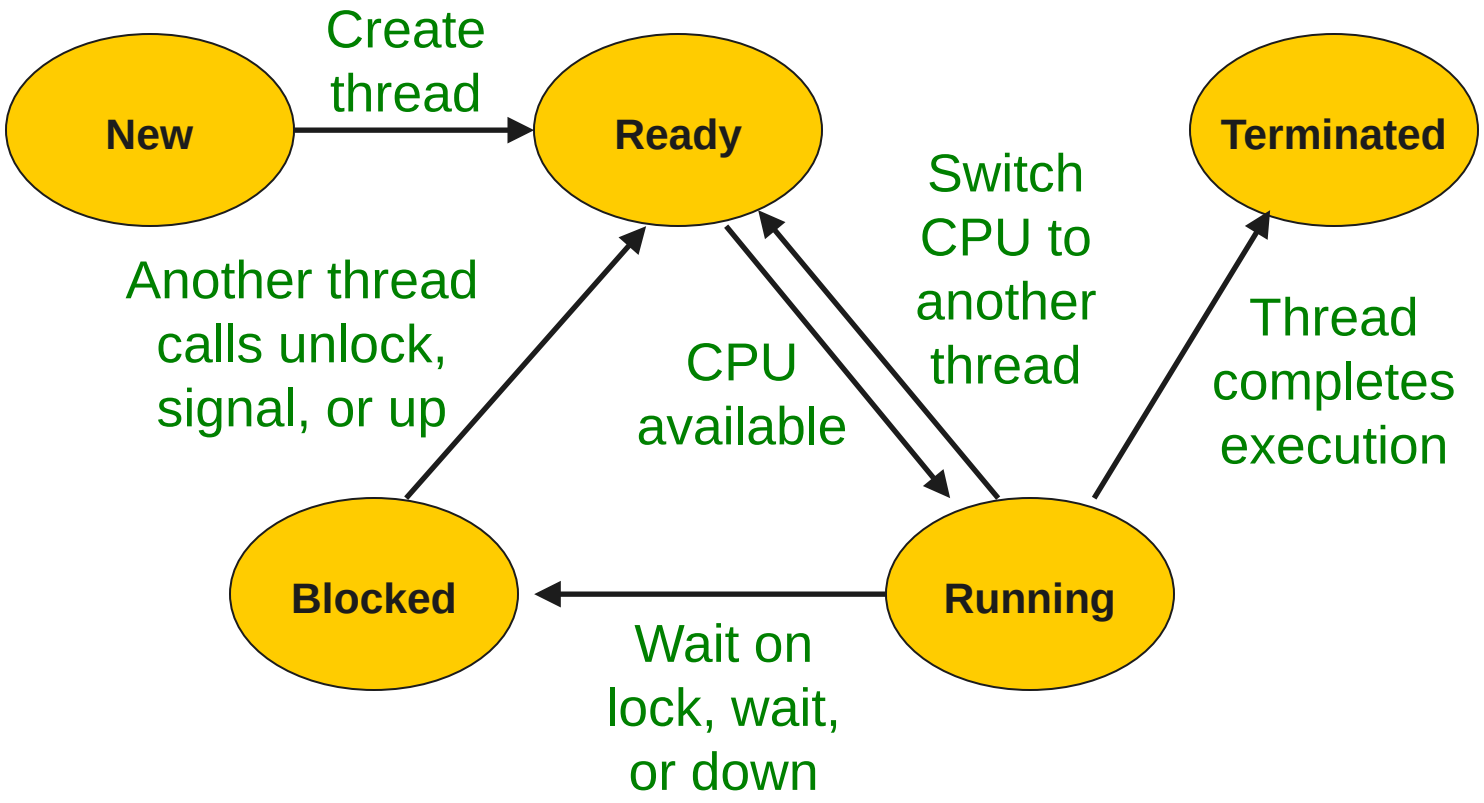
States of a Thread



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?

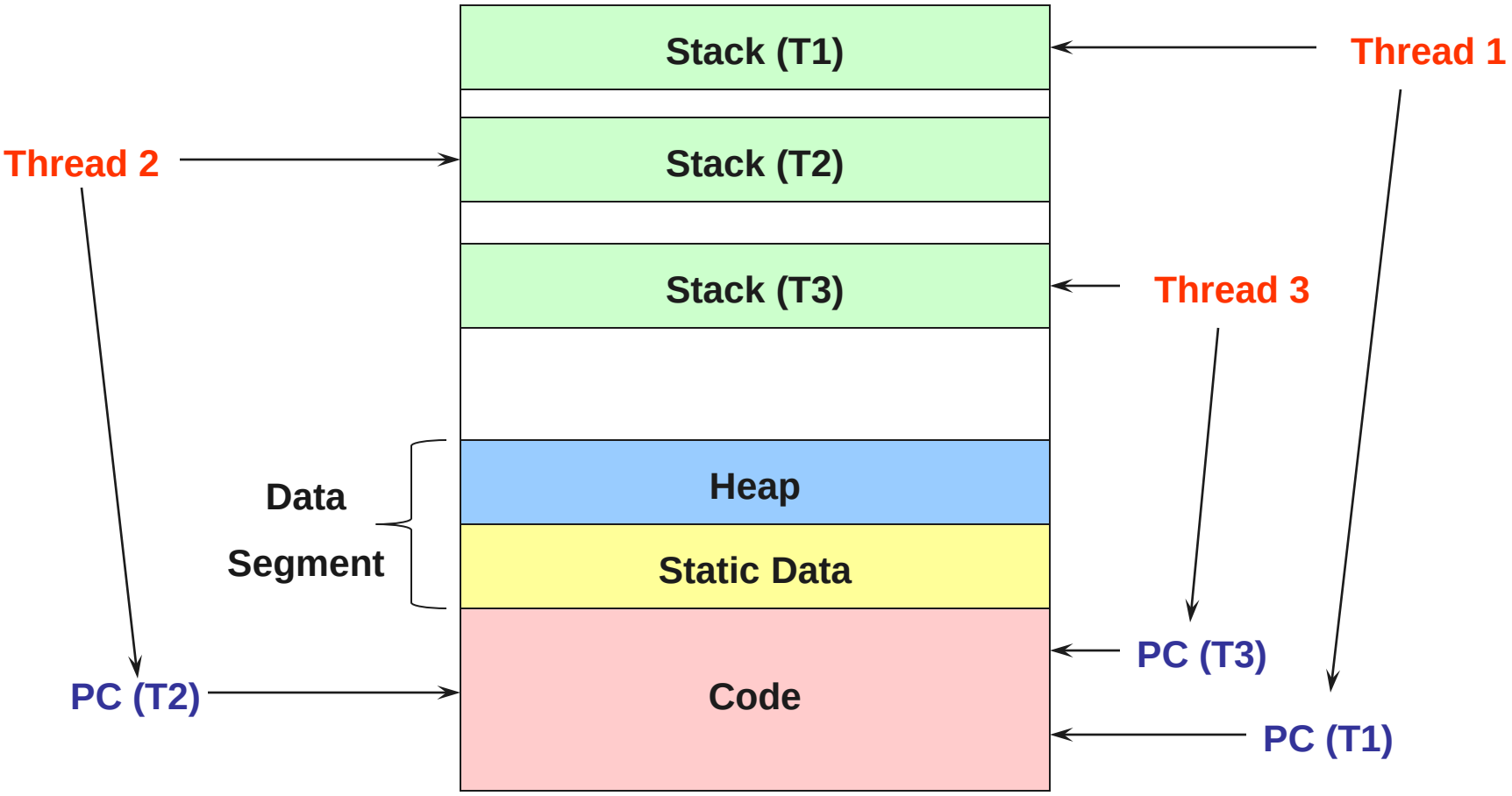
- Essentially, a "paused" execution
- Must save its private state somewhere

Thread control block (TCB)

- Per-thread OS data structure for thread info
- Store thread "context" when not running

What context 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
- Also need to store general-purpose registers

Keep track of ready threads (e.g., queue of TCBs)

Now, any thread can be

- Running on the CPU
- Ready with TCB on ready queue
- Blocked with TCB in waiting queue of lock, CV, etc.

Two Perspectives to Execution

Thread view:

Running → (Paused) → Resume

CPU view:

Thread 1 → Thread 2 → Thread 1

Context switch

1. Current thread returns control to OS
2. OS chooses new thread to run
3. OS saves current thread state: CPU to TCB
4. OS loads context of next thread: TCB to CPU
5. 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( ) or exit( ).

Are these enough?

Also need external events:

- Interrupts (e.g., timer, I/O, hardware exceptions like null pointer deref).
- Hardware prioritizes interrupts and transfers control to an OS by picking up a pointer to an interrupt handler function from an interrupt vector table.

Interrupts

Hardware events (implemented by CPU).

Stop current execution (e.g., thread function).

Start running OS interrupt handler.

In P2, only timer and inter-processor interrupts (IPI), but they may happen while in either user or OS code.

OS registers handlers in interrupt vector table.

Example: timer interrupt

- OS may set timer to go off every 10 ms.
- Guarantees that it will get control back in <= 10 ms.

Context switch

1. Current thread returns control to OS
2. OS chooses new thread to run
3. OS saves current thread state: CPU to TCB
4. OS loads context of next thread: TCB to CPU
5. OS runs next thread

Choosing next thread to run

1 ready thread: just run it.

What if only thread that exists calls yield?

>1 ready thread: need to make a decision.

CPU's scheduling policy.

Lots of options: FIFO, SJF, priority, round robin, etc.

What should CPU do if no ready threads?

Modern CPUs suspend their execution and resume on an interrupt.

interrupt_enable_suspend( ) in Project 2.

Context switch

1. Current thread returns control to OS
2. OS chooses new thread to run
3. OS saves current thread state: CPU to TCB
4. OS loads context of next thread: TCB to CPU
5. OS runs next thread

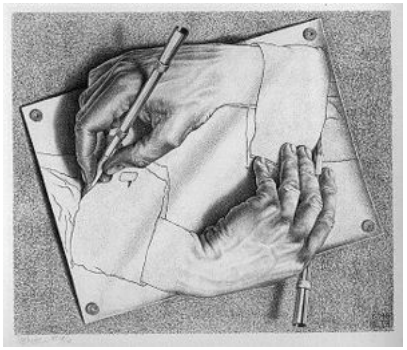
Saving state of current thread

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()*

Context switch

1. Current thread returns control to OS
2. OS chooses new thread to run
3. OS saves current thread state: CPU to TCB
4. OS loads context of next thread: TCB to CPU
5. OS runs next thread

Load context and run

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

```
start thread 1
start yield: thread 1


end yield: thread 1
end thread 1
```

Thread 2 output

```
start thread 2
start yield: thread 2


end yield: thread 2
end thread 2
```

Creating a new thread

Create a running thread? Seems challenging.

Instead, create a paused thread.

Key idea: pretend it was running, put it on the ready queue, then just wait for it to be scheduled!

Implication:

Construct TCB as if it were paused at thread start.

Recipe for creating a thread

1. Allocate and initialize TCB.

   Set PC to start of thread function.

   Set general-purpose registers to func parameters.

2. Allocate and initialize stack.

   What goes on stack?

   Set TCB stack pointer to stack top.

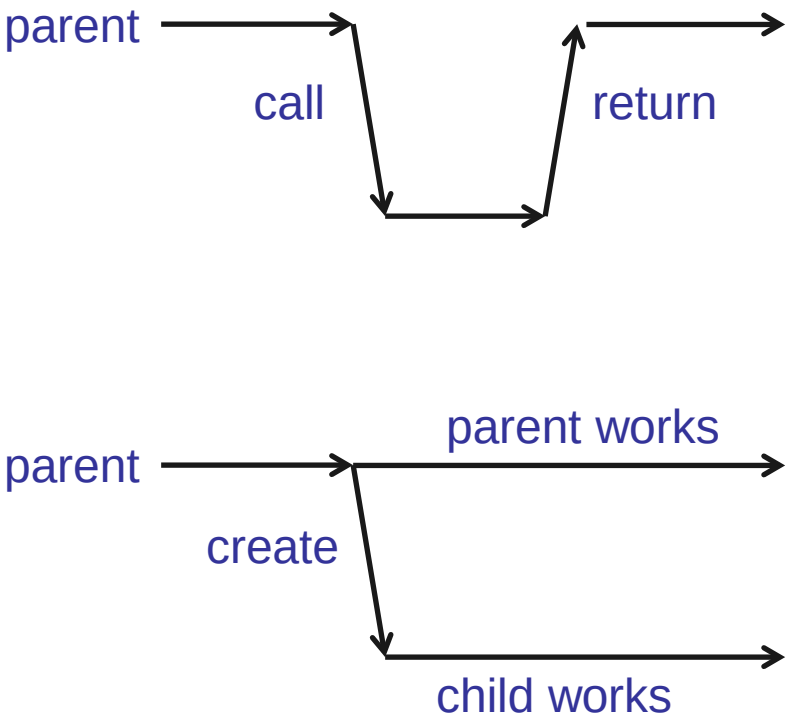
   getcontext( ) and *makecontext( )* in Project 2.

3. Add TCB to ready queue.

We've talked here about creating threads.
Coming up later: creating a new process with fork( ).

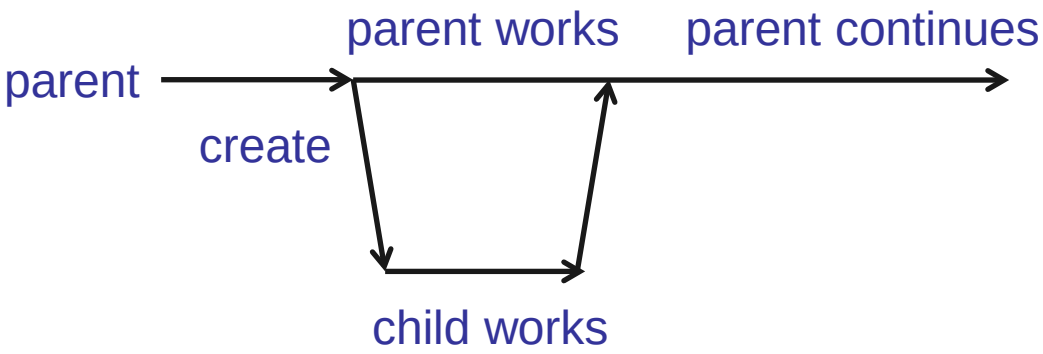
How to use new thread

Creating a thread is like an asynchronous procedure call



Synchronizing with child

What if parent wants to work for a while, then wait for child to finish?



Synchronizing with child

```
parent( )
   create child thread
   print "parent works"
   …
   // Print this after the
   // child is done.
   print "parent continues"
   …

child( )
   …
   print "child is done"
```

Desired output

```
parent works
child is done
parent continues
```

OR

```
child is done
parent works
parent continues
```

Can we guarantee the desired output with this code?

Synchronizing with child

```
create child thread
   print "parent works"
   …
   // Give up the processor so
   // the child will run.
   yield( )
   print "parent continues"
   …

child( )
   …
   print "child is done"
```

Desired output

```
parent works
child is done
parent continues
```

OR

```
child is done
parent works
parent continues
```

Does this work?

Synchronizing with child

```
create child thread
   print "parent works"
   …
   // Use a join( ).
   childThread.join( )
   print "parent continues"
   …

child( )
   …
   print "child is done"
```

Desired output

```
parent works
child is done
parent continues
```

OR

```
child is done
parent works
parent continues
```

How to make do without join?

Synchronizing with monitors

```
parent( )
      childDone = 0
      create child thread
      print "parent works"
      lock( )
      while ( !childDone )
          wait( )
      unlock( )
      print "parent continues"

child( )
      print "child is done"
      lock( )
      childDone = 1
      signal( )
      unlock( )
```

Desired output

```
parent works
child is done
parent continues
```

OR

```
child is done
parent works
parent continues
```

Project 2 update

You can now do a substantial part of project 2

- Thread create
- Context switch
- Thread join

Next topic: Implementing synchronization

- Need to protect OS data structures (ready queue)
- Need to block without (much) busy waiting