Winter 2025

EECS 482: Introduction to Operating Systems

Lecture 6: Building Threads

Prof. Ryan Huang

Administration

Project 1 due today

- Soft deadline
  - You may continue to submit to AG after the deadline but the extra points won’t receive the hand-grading multiplier

Project 2 will be posted today

- Implement a thread library

For ungrouped students, we’re in the process of matching

Threads can interact in two ways

Multiple threads share data to cooperate on task

- Coordinate via synchronization operations, e.g., locks, condition variables, semaphores

We've been assuming that each thread has its own processor (of unpredictable speed). But actually…

Multiple threads can share a single processor

What is a thread?

A thread is a sequence of executing instructions

So what's a non-running thread?

- A non-running thread is a paused execution
- Can pausing a thread break a correct concurrent program?

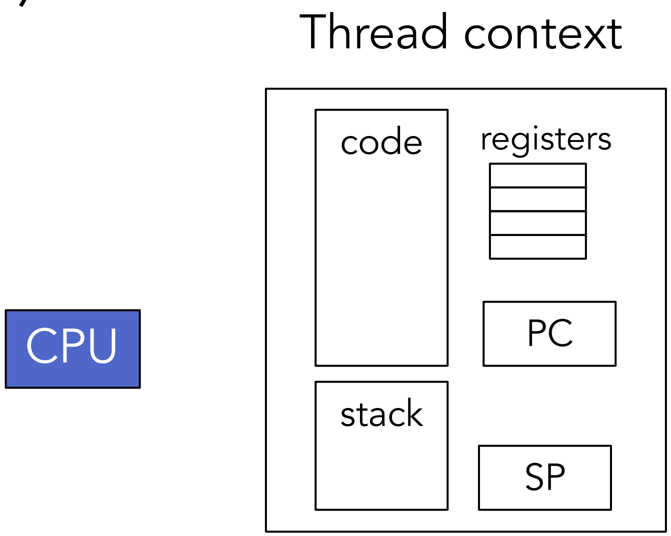
How to pause a thread and resume it later?

- Called "context switch"

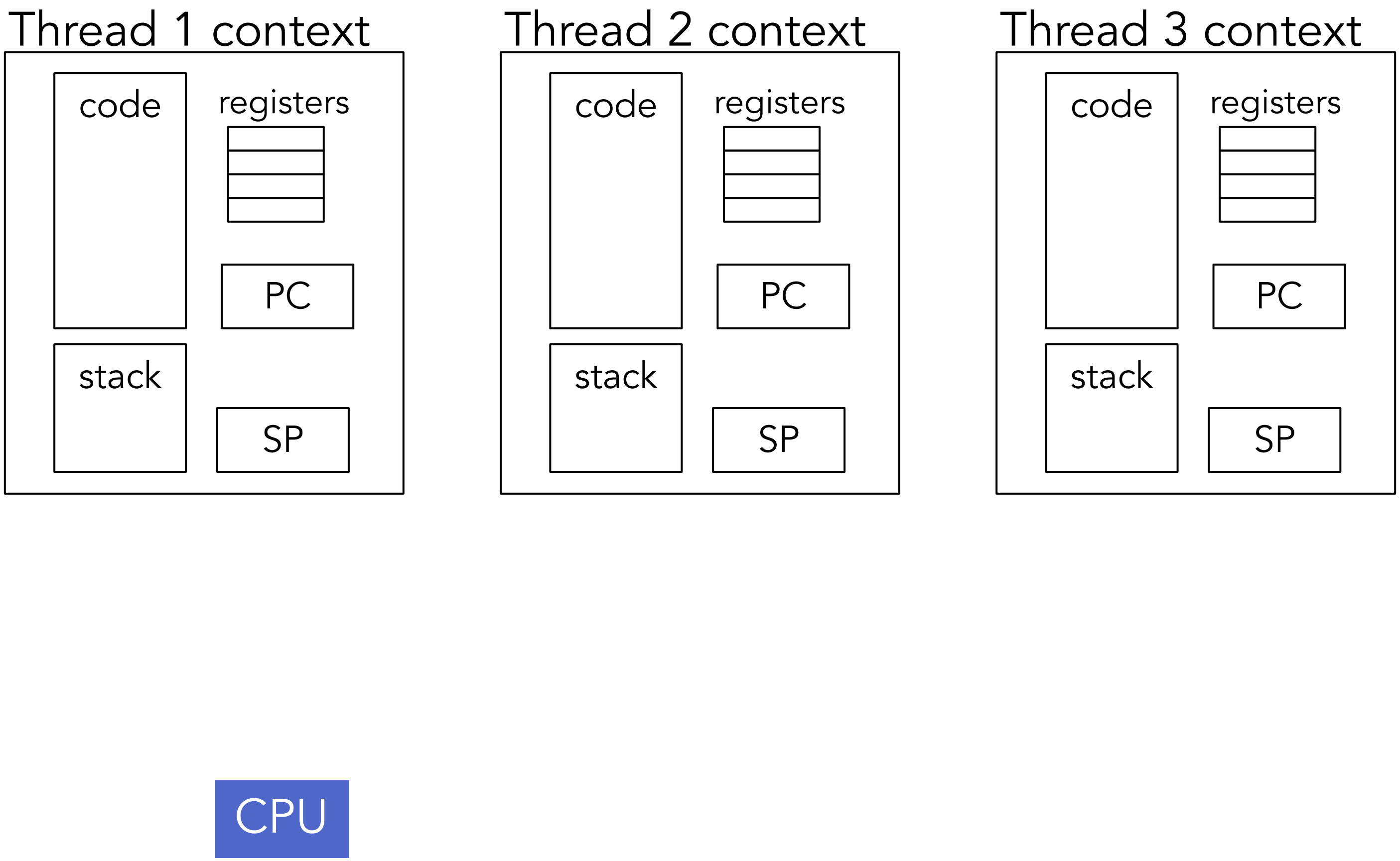
Per-thread state

What state is needed by a thread as it runs?

- Code
  - + program counter (e.g., `eip`)
- Stack
  - + stack pointer (e.g., `esp`)
- Registers (e.g., `eax`, `ebx`, …)



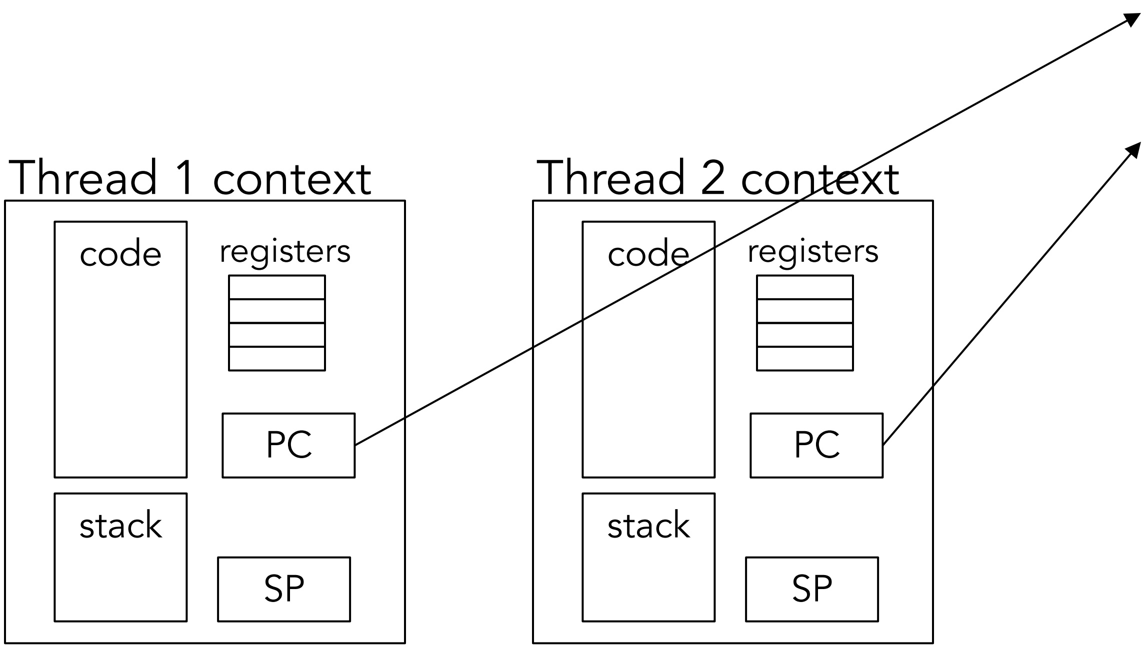
Sharing the CPU



Optimizations

How to avoid copying code back and forth?

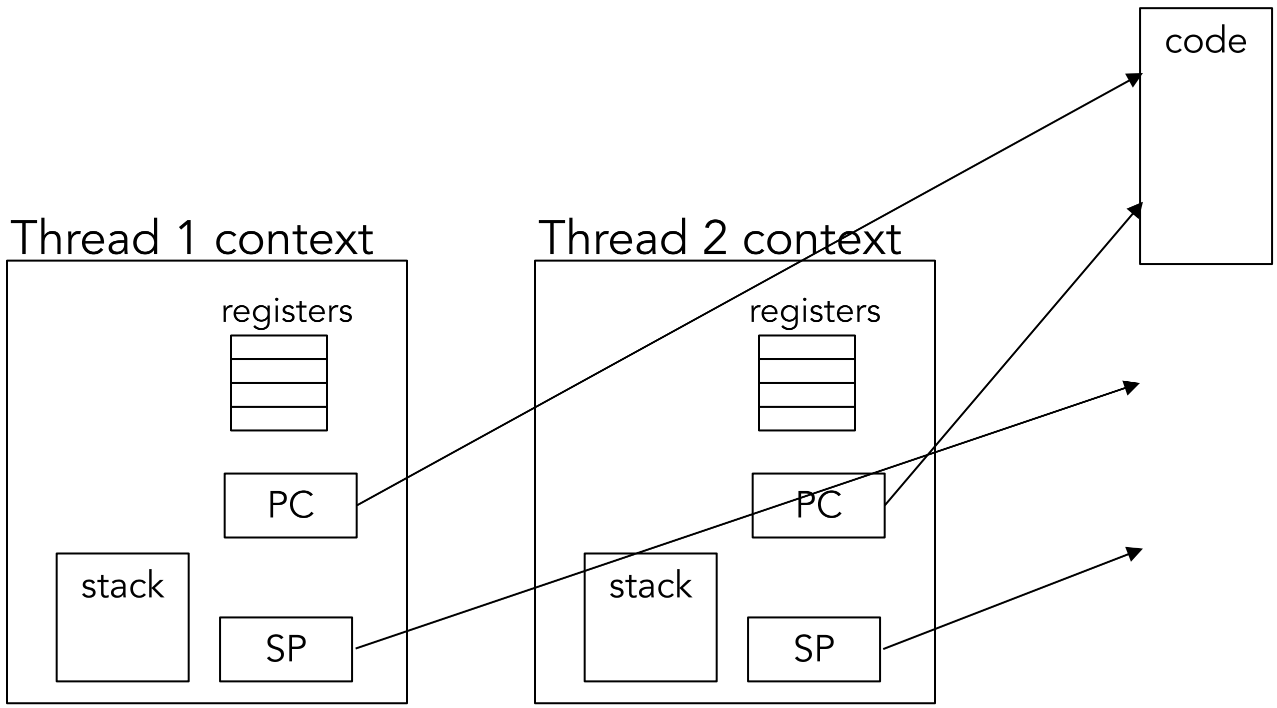
- Store separately from thread context (PC is still part of context)
- Threads use same code



Optimizations

How to avoid copying the stack?

- Store separately from thread context (SP is still part of context)



Thread control block (TCB)

OS data structure to store thread info

- In memory
- Record execution context (SP, PC, registers)
  - when the thread is not running (paused)
- Why is TCB needed?
  - What if we do not record the execution context?

Allocated when a new thread is created

- Typically a thread stack is allocated together
  - e.g., in one 4KB page
- Destroyed when a thread is finished

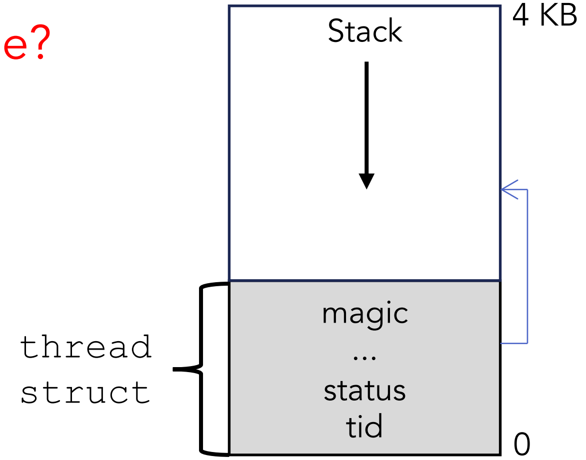
TCB example

An example TCB struct

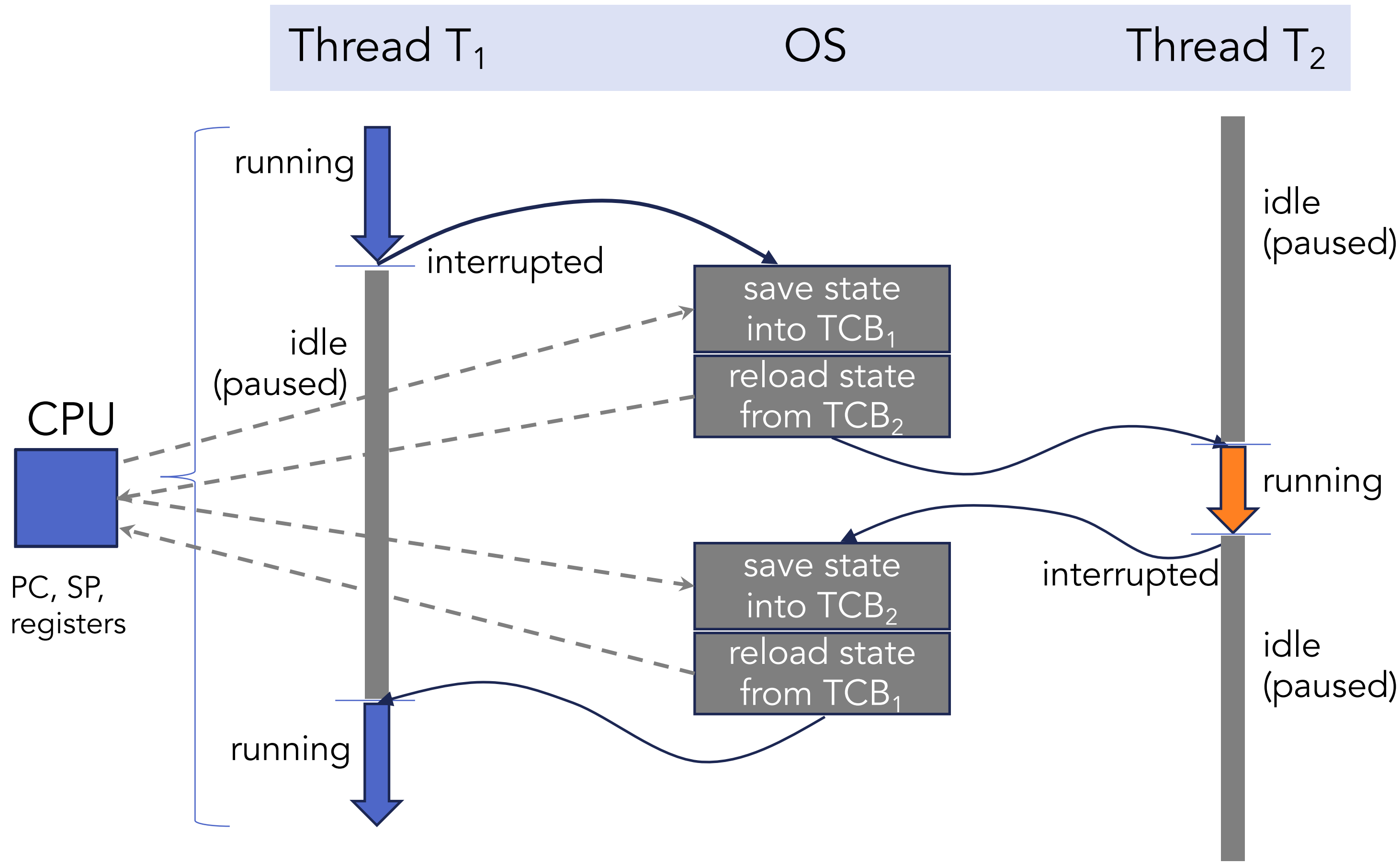
```
struct thread {
    tid_t tid;                    /* Thread identifier. */
    enum thread_status status;    /* Thread state. */
    char name[16];                /* Name (for debugging purposes). */
    uint8_t *sp;                  /* Saved stack pointer. */
    int priority;                 /* Priority. */
    struct list_elem allelem;     /* List element for all threads list. */
    struct list_elem elem;        /* List element. */
    unsigned magic;               /* Detects stack overflow. */
};
```

Where are the PC and registers stored in this example?

They are pushed onto the stack



Sharing the CPU



Two perspectives when threads are sharing a processor

Thread view:

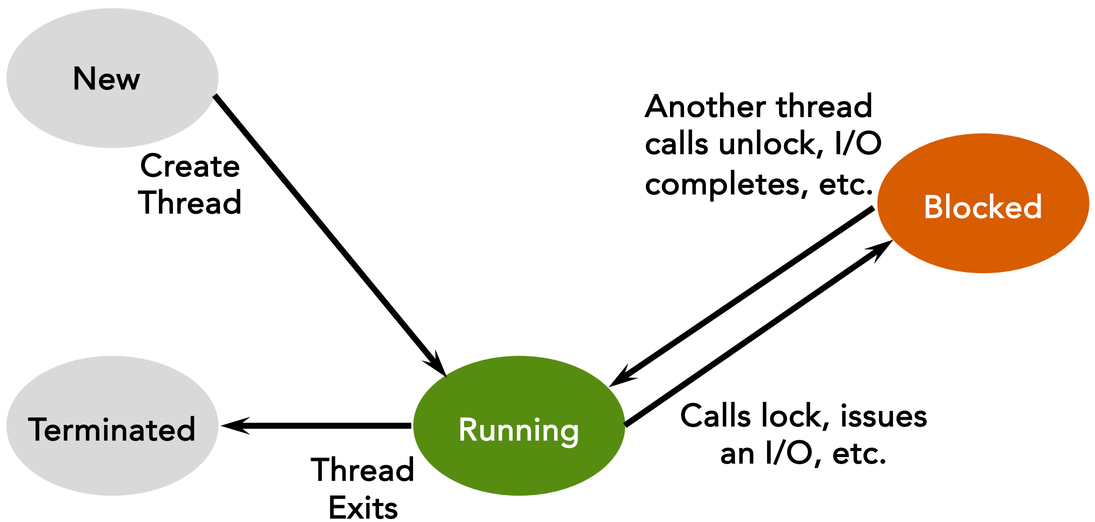
- Running → Paused → Running

CPU view:

- Thread 1 → Thread 2 → Thread 1

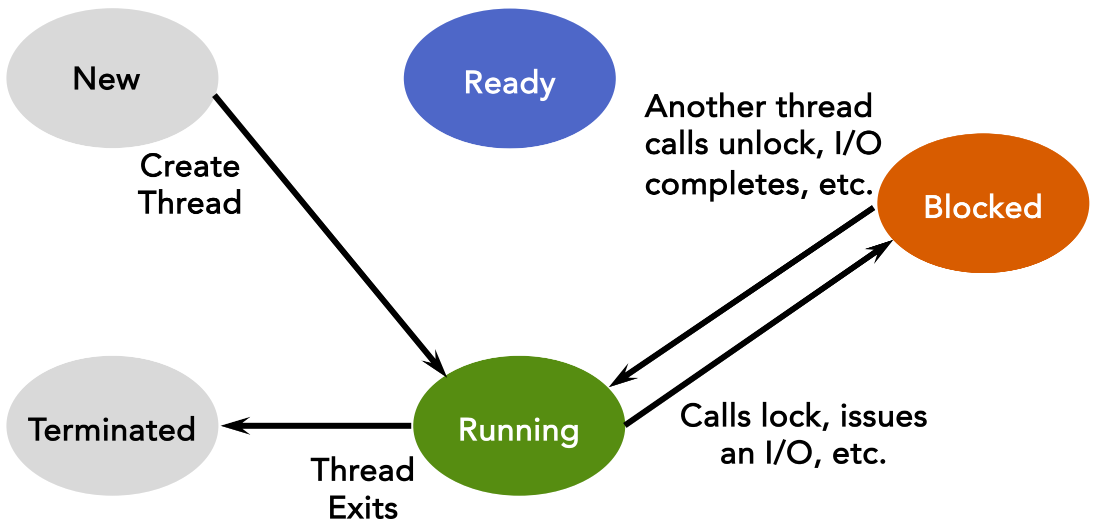
First, let's take the thread view

States of a thread



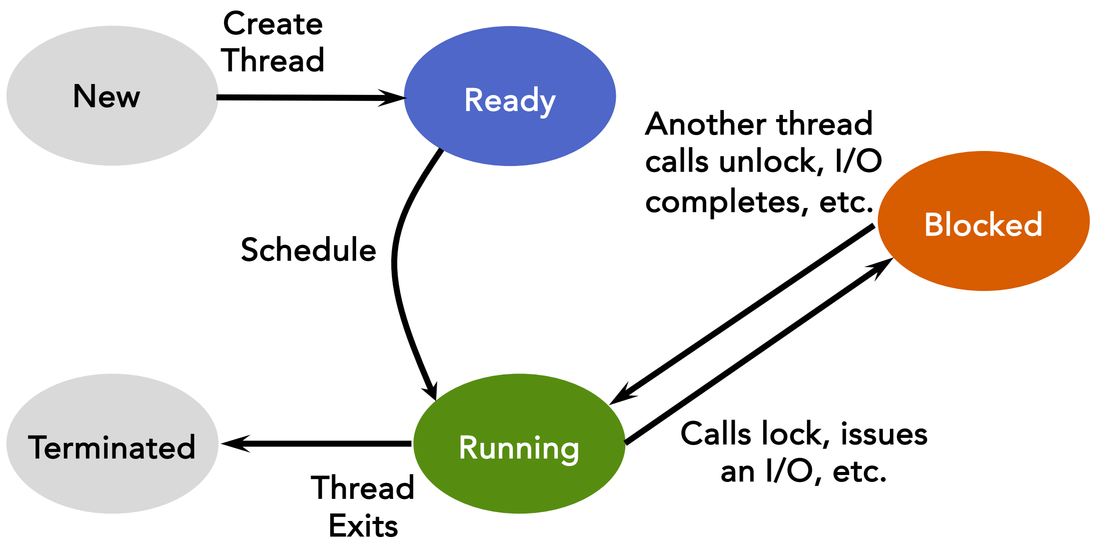
What if there are more threads than CPUs?

States of a thread



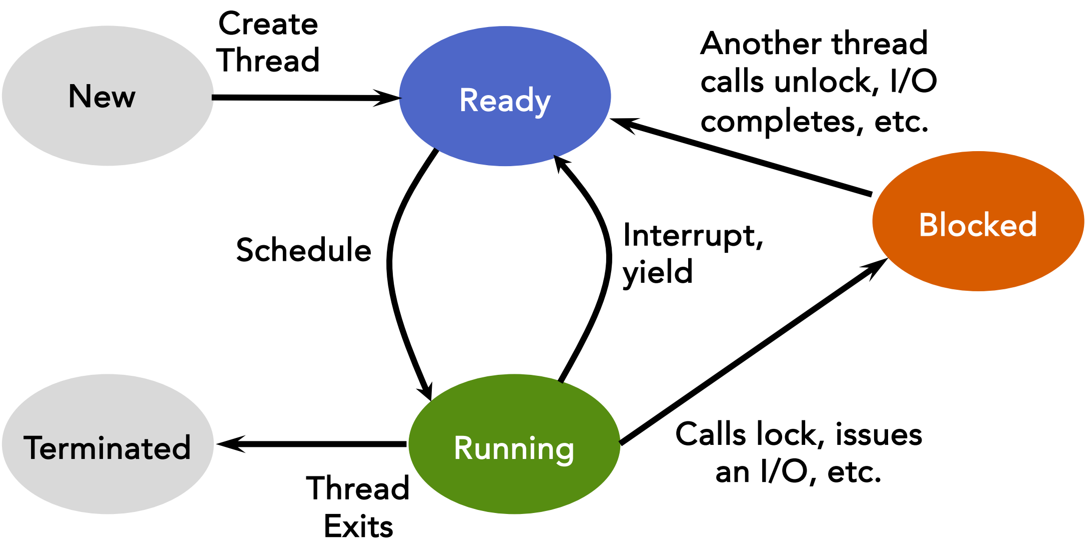
What if there are more threads than CPUs?

States of a thread



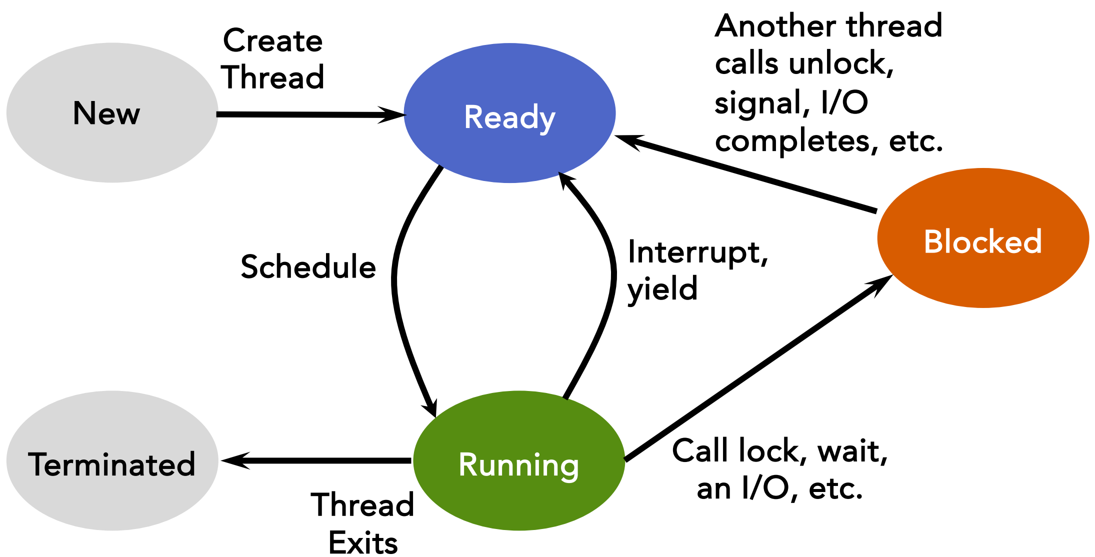
What if there are more threads than CPUs?

States of a thread



What if there are more threads than CPUs?

States of a thread



Why no transition from Ready to Blocked?

Why no transition from Blocked to Running?

State queues

How does the OS keep track of threads?

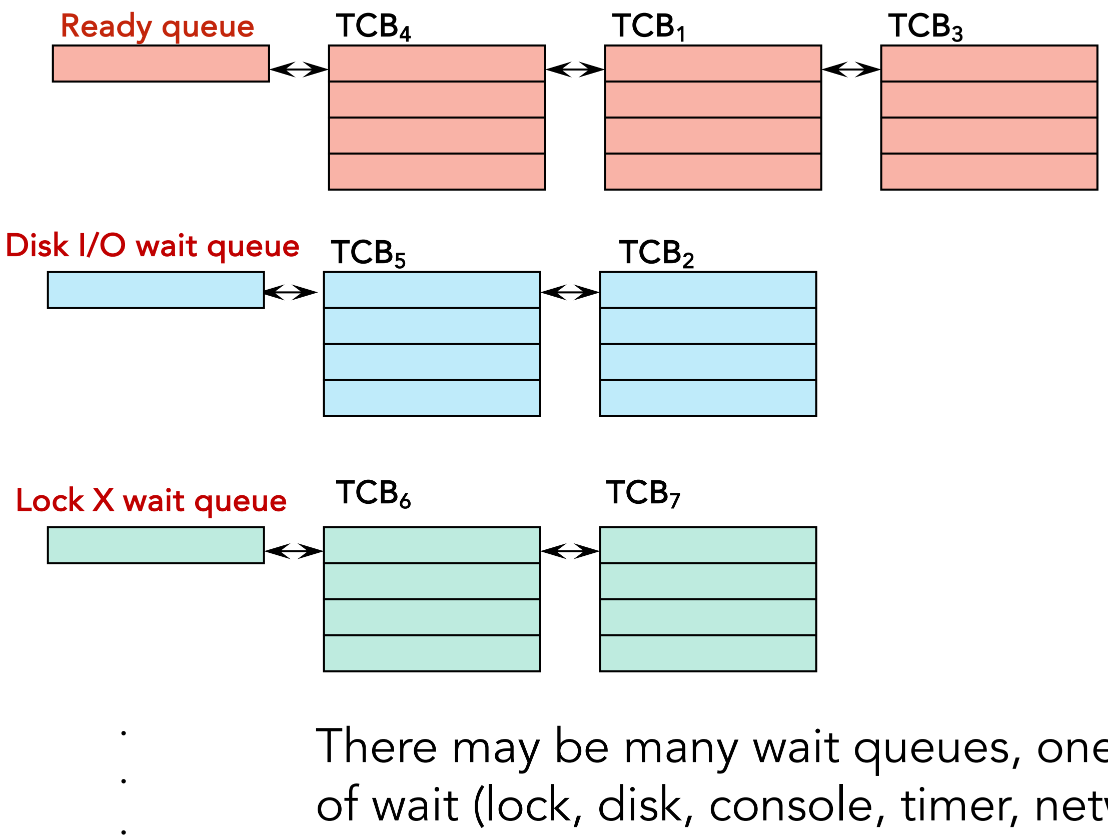
Simple approach: all threads list

- How to find out threads in the ready state?
  - Iterate through the list
- Problem: slow!

Improvement: partition list based on states

- OS maintains a collection of queues
  - Typically one queue for each state: ready, waiting, etc.
- A TCB is queued on a state queue according to its current state
  - Queue element is typically a pointer to a TCB
- As a thread changes state, its TCB is moved from one queue into another

State queues



There may be many wait queues, one for each type of wait (lock, disk, console, timer, network, etc.)

What about a running queue?

How many threads can be in the running state simultaneously?

Two perspectives when threads are sharing a processor

Thread view:

- Running → Paused → Running

CPU view:

- Thread 1 → Thread 2 → Thread 1

Next, let's take the CPU view

Switching threads

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 to OS?

Internal events

- Initiated by the current thread itself

Examples

- Thread calls lock(), wait(), down(), etc.
- Thread requests OS to do some work (e.g., I/O)
- Thread voluntarily gives up CPU with yield()

Thread yielding

Threads voluntarily give up the CPU with `yield`

Ping Thread

```
while (true) {
  printf("ping\n");
  yield();
}
```

Pong Thread

```
while (true) {
  printf("pong\n");
  yield();
}
```

What is the output of running these two threads?

What does it mean for yield to return?

Cooperative thread scheduling

How does thread return control to OS?

Internal events

Are internal events sufficient?

External events

- Initiated by something outside the current thread
- Interrupts: a hardware event that transfers control from thread to OS interrupt handler

Interrupts

An interrupt is an event raised by some hardware component (e.g., an I/O device)

- *Asynchronous:* originating outside the thread
  - not caused by the current executing instruction
- Indicates that some device needs services
- All interrupts are assigned a number, typically [0, 255]

Processor checks for pending interrupt

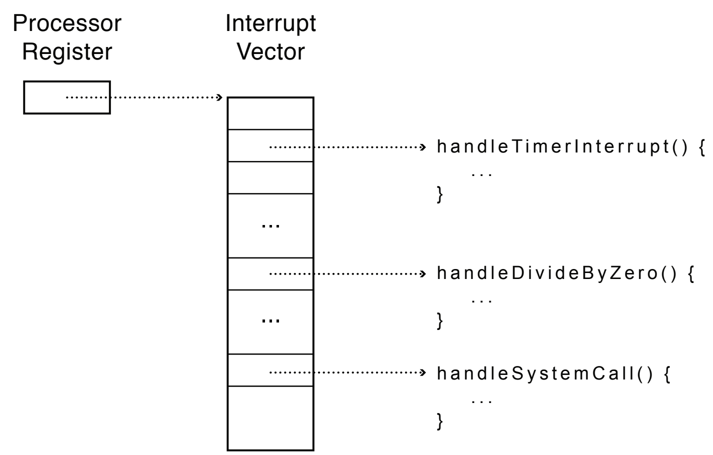
When a pending interrupt is detected

1. CPU stops executing current code
2. CPU calls an OS interrupt handler

Interrupt Vector Table (IVT)

A data structure to associate interrupts with handlers

- Each entry is called an interrupt vector
  - specifies the address of the interrupt handler
- Programmed by the OS, *i.e.*, software interface for interrupts



CPU invokes the corresponding interrupt handler in the *IVT* based on the pending interrupt number

Example: timer interrupt

Timer is critical for an OS

- Fallback mechanism for OS to reclaim control
- OS could set timer to go off every 10 ms
- When timer expires, it generates an interrupt
  - Guarantees that OS will get control back in <= 10 ms

Timer interrupt handler forces the current thread to "call" yield

- Achieves preemptive scheduling
- Causes an involuntary context switch