EECS 482
Introduction to Operating Systems

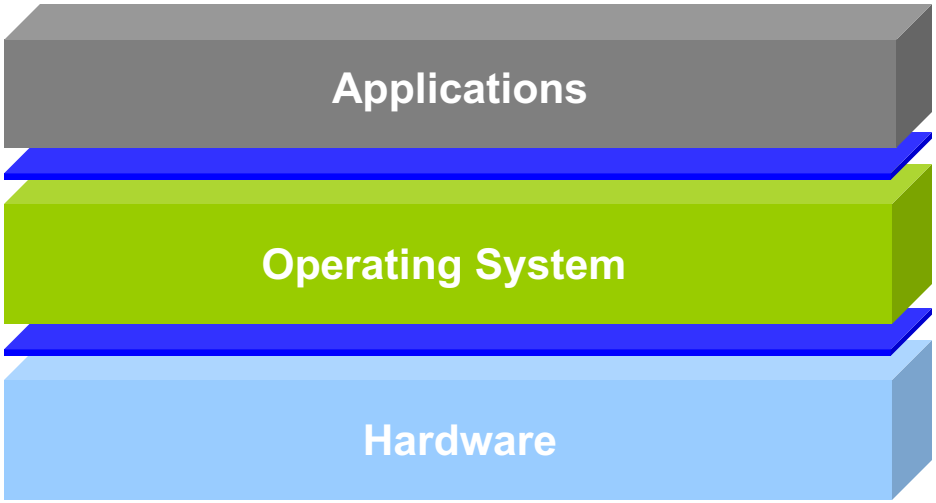
Winter 2018

Baris Kasikci

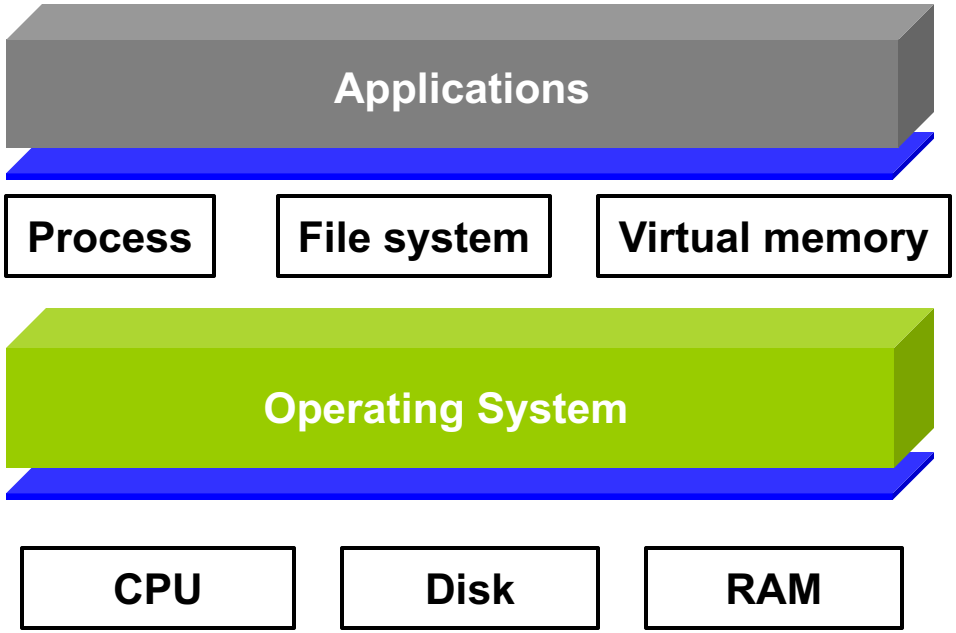
Slides by: Harsha V. Madhyastha

What does an OS do?

- Creates **abstractions** to make hardware easier to use
- Manages **shared** hardware resources



OS Abstractions



Upcoming Schedule

- This lecture starts a class segment that covers processes, threads, and synchronization
  - Perhaps the most important in this class
  - **Basis for Projects 1 and 2**

Managing Concurrency

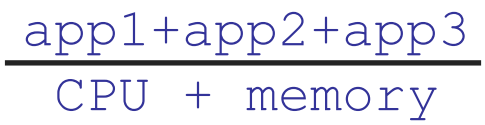
- Recall: Source of OS complexity
  - Multiple users, programs, I/O devices, etc.
  - Originally for efficient use of H/W, but useful even now

- How to manage this complexity?
  - Divide and conquer
  - Modularity and abstraction

The Process

- The process is the OS abstraction for execution
  - Also sometimes called a job or a task

- Recall: For each area of OS, ask
  - What interface does hardware provide?



  - What interface does OS provide?

What is a process

- A process is a **running program**
  - Programs are static entities with potential for execution

- A process is like a play being acted out
  - A program is the script of the play

Process Components

- A process, named using its process ID (PID), contains all the state for a running program

  - Set of threads (active)
    - » a sequence of executing instructions from a program
    - » analogy: acting done by an actor

  Execution

  - An address space (passive)
    - » the memory used by the program as it runs
    - » analogy: all the objects on the stage

  Memory

Process Address Space

- What's in the address space?

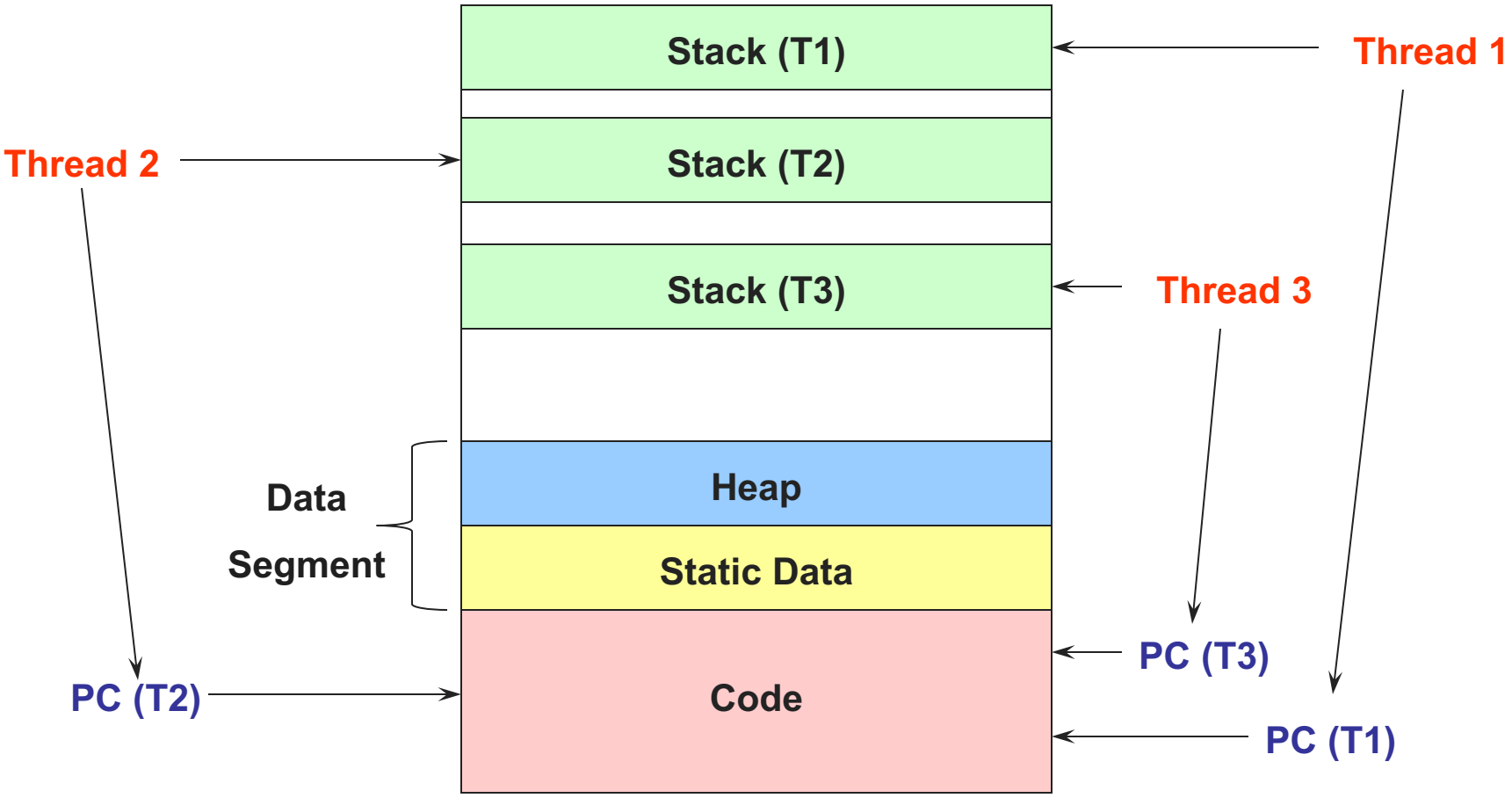
Cross-thread state

  - » The code and input data for the executing program
  - » The memory allocated by the executing program
  - » Open files, network connections, etc.

Per-thread state

  - » An execution stack encapsulating the state of procedure calls
  - » The program counter (PC) indicating the next instruction
  - » A set of general-purpose registers with current values

Process Address Space



Review of Stack Frames

```
A(int tmp) {
    B(tmp);
}

B(int val) {
    C(val, val + 2);
    A(val - 1);
}

C(int foo, int bar) {
    int v = bar - foo;
}
```

A(tmp = 1)

Review of Stack Frames

```
A(int tmp) {
    B(tmp);
}

B(int val) {
    C(val, val + 2);
    A(val - 1);
}

C(int foo, int bar) {
    int v = bar - foo;
}
```

A(tmp = 1)

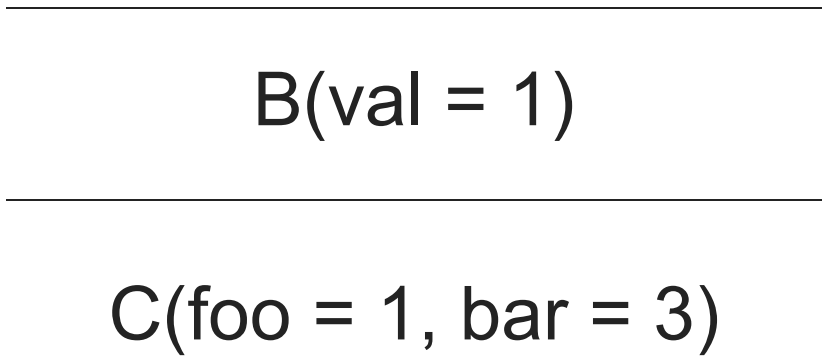
B(val = 1)

Review of Stack Frames

```
A(int tmp) {
    B(tmp);
}

B(int val) {
    C(val, val + 2);
    A(val - 1);
}

C(int foo, int bar) {
    int v = bar - foo;
}
```



Review of Stack Frames

```
A(int tmp) {
    B(tmp);
}

B(int val) {
    C(val, val + 2);
    A(val - 1);
}

C(int foo, int bar) {
    int v = bar - foo;
}
```

A(tmp = 1)

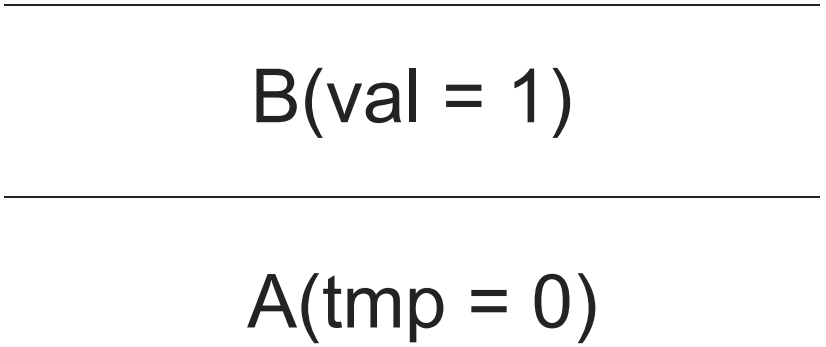
B(val = 1)

Review of Stack Frames

```
A(int tmp) {
    B(tmp);
}

B(int val) {
    C(val, val + 2);
    A(val - 1);
}

C(int foo, int bar) {
    int v = bar - foo;
}
```



Multiple Threads

- Can have several threads in a single address space
  - Sometimes they interact
  - Sometimes they work independently
  - Analogy: multiple roles being acted out
- What does a thread need to execute?
  - Stack (and SP)
  - PC
  - Code specific to the thread
- What does a thread need to communicate with other threads?
  - Data segment

Upcoming Topics

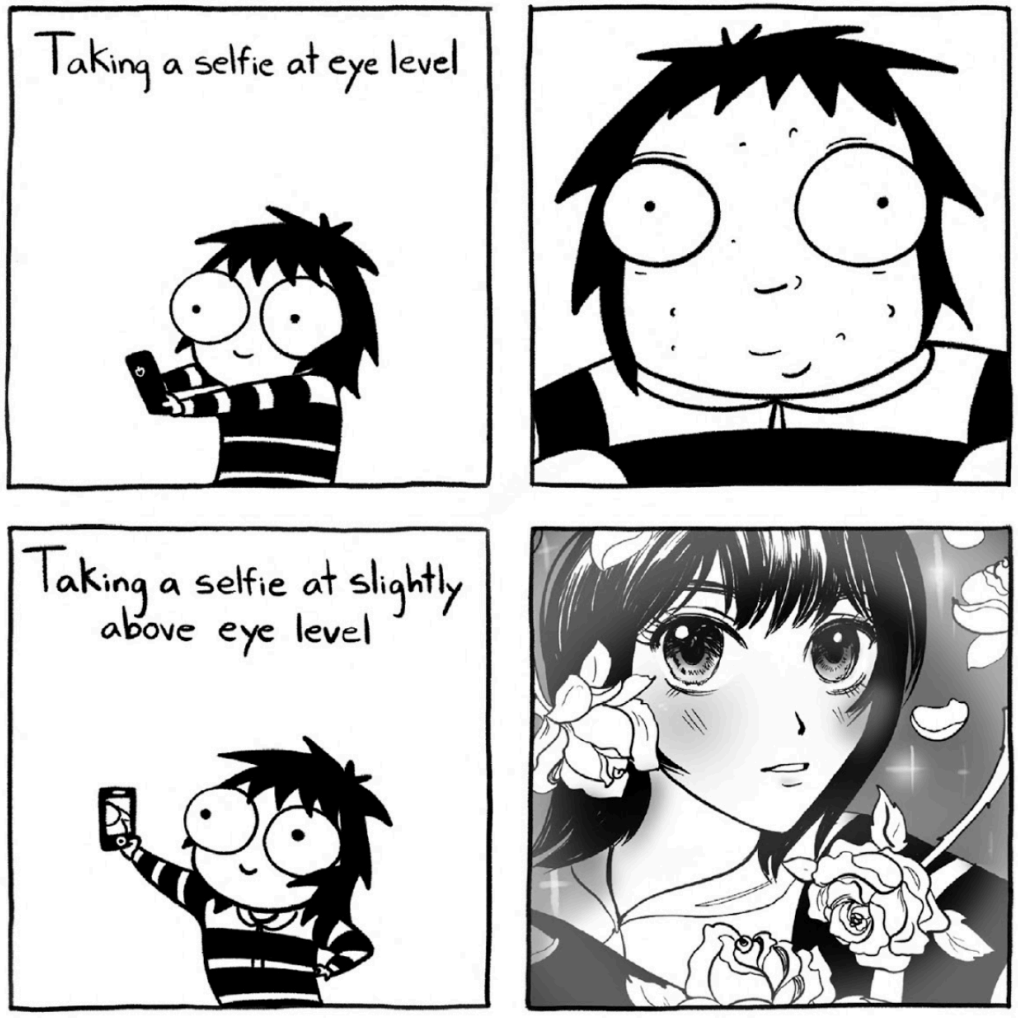
- Threads: unit of concurrency
  - How multiple threads can cooperate to accomplish a single task? (Project 1)
  - How multiple threads can share limited number of CPUs? (Project 2)

- Address spaces: unit of state partitioning
  - How multiple address spaces can share a single physical memory efficiently, flexibly, and safely? (Project 3)

Announcements

- First discussion section on Friday
  - Bring your laptop
- Sign up for GitHub and Piazza
  - Must do by discussion section on Friday
- Started putting together a project group?

Selfies

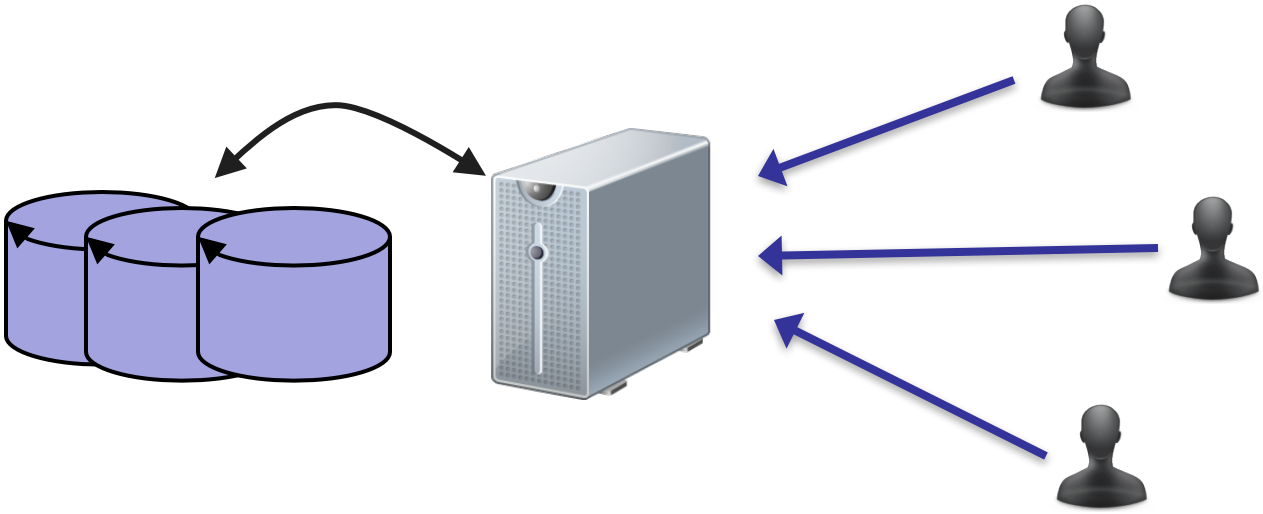


Big Mushy Happy Lump, Sarah Andersen

Send them to barisk@umich.edu

Why do we need threads?

- Example: Web server
  - Receives multiple simultaneous requests
  - Reads web pages from disk to satisfy each request



Option 1: Handle one request at a time

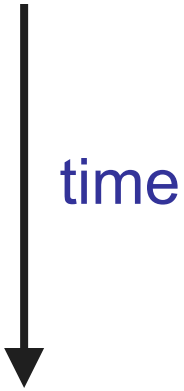
- Example execution schedule (single CPU):

  Request 1 arrives
  Server receives request 1
  Server starts disk I/O 1a
  Request 2 arrives
  Server waits for I/O 1a to finish

- Easy to program, but slow
  - Why slow?
  - Can't overlap disk requests with computation, or with network receives

Option 2: Event-driven web server (asynchronous I/O)

- Issue I/Os, but don't wait for them to complete (single CPU)



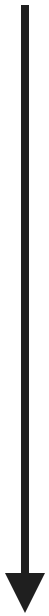
Request 1 arrives
Server receives request 1
Server starts disk I/O 1a to satisfy request 1
Request 2 arrives
Server receives request 2
Server starts disk I/O 2a to satisfy request 2
Request 3 arrives
Disk I/O 1a finishes

Web server must remember
What requests are being served, and what stage they're in
What disk I/Os are outstanding (and which requests they belong to)

Lots of extra state!

Multi-threaded web server

- One thread per request (single CPU)
  - Thread issues disk (or n/w) I/O, then waits for it to finish
  - Though thread is blocked on I/O, other threads can run
  - Where is the state of each request stored?



Thread 1	Thread 2	Thread 3
Request 1 arrives		
Receive request 1		
Start disk I/O 1a		
	Request 2 arrives	
	Receive request 2	
	Start disk I/O 2a	
		Request 3 arrives
		Receive request 3
Disk I/O 1a finishes		
Continue handling request 1		

Benefits of Threads

- Thread manager takes care of CPU sharing
  - Other threads can progress when one thread issues blocking I/Os
  - Private state for each thread

- Applications get a simpler programming model
  - The illusion of a dedicated CPU per thread

When are threads useful?

- Multiple things happening at once
- Usually some slow resource

- Examples:
  - Network server
  - Controlling a physical system (e.g., airplane controller)
  - Window system

Ideal Scenario

- Split computation into threads
- Threads run independently of each other
  - Divide and conquer works best if divided parts are independent

Is independence of threads practically possible?

Dependence between threads

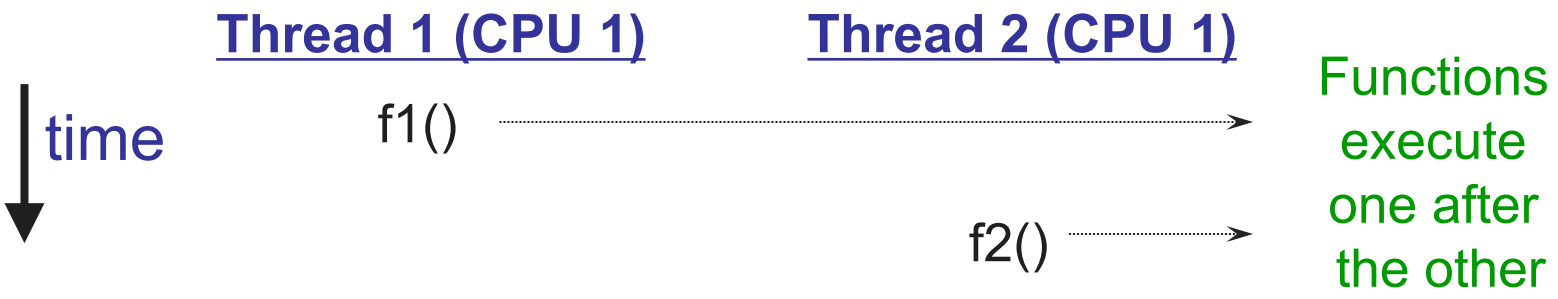
- Example 1: Microsoft Word
  - One thread formats document
  - Another thread spell checks document
- Example 2: Desktop computer
  - One thread plays World of Warcraft
  - Another thread compiles EECS 482 project
- Two types of sharing: app resource or H/W

Concurrency vs. Parallelism

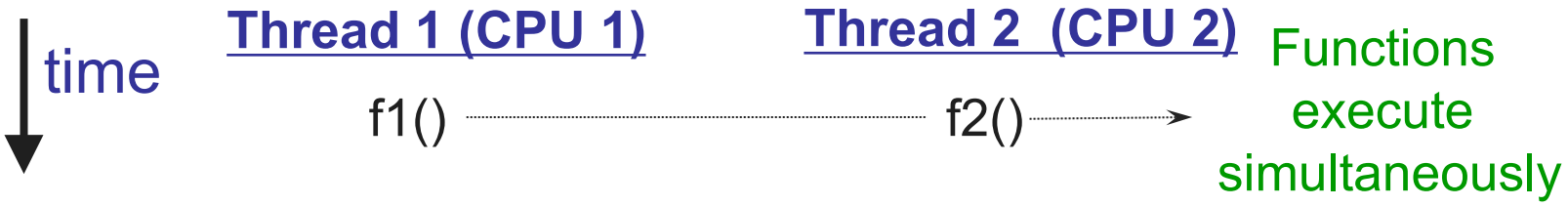
- Concurrency
  - A way to reason about your program (code) as a collection of communicating threads
  - Execution is not guarantees to be simultaneous
  - Could run on a single CPU
- Parallelism
  - A model of program execution
  - Execution happens simultaneously

Concurrency vs. Parallelism

- Concurrency with single CPU



- Concurrency with 2 CPUs -> Parallelism is possible



Cooperating threads

- How can multiple threads cooperate on a single task?
  - Example: Ticketmaster's webserver
  - Assume each thread has a dedicated processor (multiple CPUs)
- The main problem:
  - Ordering of events across threads is non-deterministic
  - Speed of each processor is unpredictable

- Consequences:
  - Many possible global ordering of events
  - Some may produce incorrect results

Non-deterministic ordering → Non-deterministic results

- Printing example

Thread 1	Thread 2
Print ABC	Print 123

  - Possible outputs?
    - » 20 outputs: ABC123, AB1C23, AB12C3, AB123C, A1BC23, A12BC3, A123BC, 1ABC23, 1A2BC3, …
  - Impossible outputs?
    - » ABC321

- Ordering within thread is sequential, but many ways to merge per-thread order into a global order
- What's being shared between these threads?

Non-deterministic ordering → Non-deterministic results

- Arithmetic example (y is initially 10)

Thread A	Thread B
x = y + 1	y = y * 2

  - What's being shared between these threads?
  - Possible results?
    - » If A runs first: x = 11 and y = 20
    - » If B runs first: x = 21 and y = 20

- Another example (x is initially 0)

Thread A	Thread B
x = 1	x = -1

  - Possible results?
    - » x = 1 or -1
  - Impossible results?
    - » x = 0

Thread A	Thread B
x = 0	x = 0
x++	x--

Atomic operations

- Atomic operations
  - Indivisible, i.e., happens in its entirety or not at all
  - No events from other threads can occur in between

- Print example:
  - What if each print statement were atomic?
  - What if printing a single character were not atomic?

- Most computers
  - Memory load and store are atomic (*e.g.,* mov on x86)
  - Many other instructions are not atomic
    - » Example: double-precision floating point
  - Need a low-level atomic operation (test-and-set) to build a high-level atomic operation (lock)

Example

Thread A

```
i=0
while (i<10) {
    i++
}
print "A finished"
```

Thread B

```
i=0
while (i> -10) {
    i--
}
print "B finished"
```

- Which thread will finish first?
- Is the winner (one that reaches the end of the loop first) guaranteed to print first?
- Is it guaranteed that someone will win?
- Say both threads run at *exactly* the same speed, and start close together
  - Is it guaranteed that both threads will loop forever?

- Non-deterministic interleaving makes debugging challenging
  - Heisenbug: a bug that occurs non-deterministically