# CS240: Programming in C

## Lecture 17: Threads

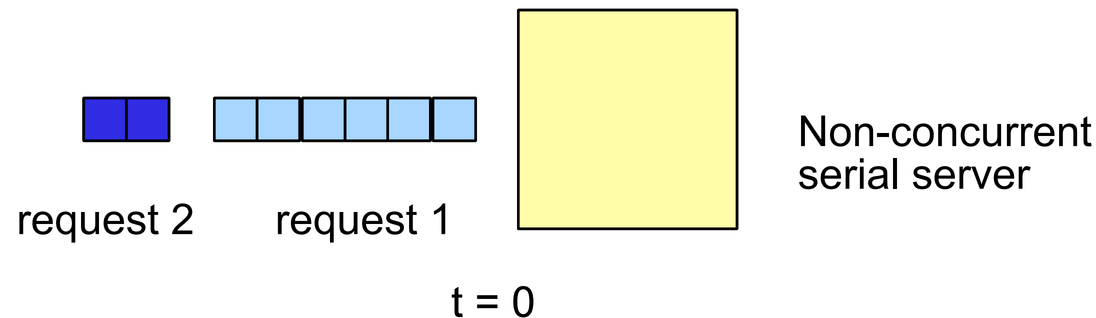
# Concurrency and Parallelism

- Concurrency is concerned with the management of logically simultaneous activities
  - Best-fit job scheduling
  - event handling (GUI)
  - web server

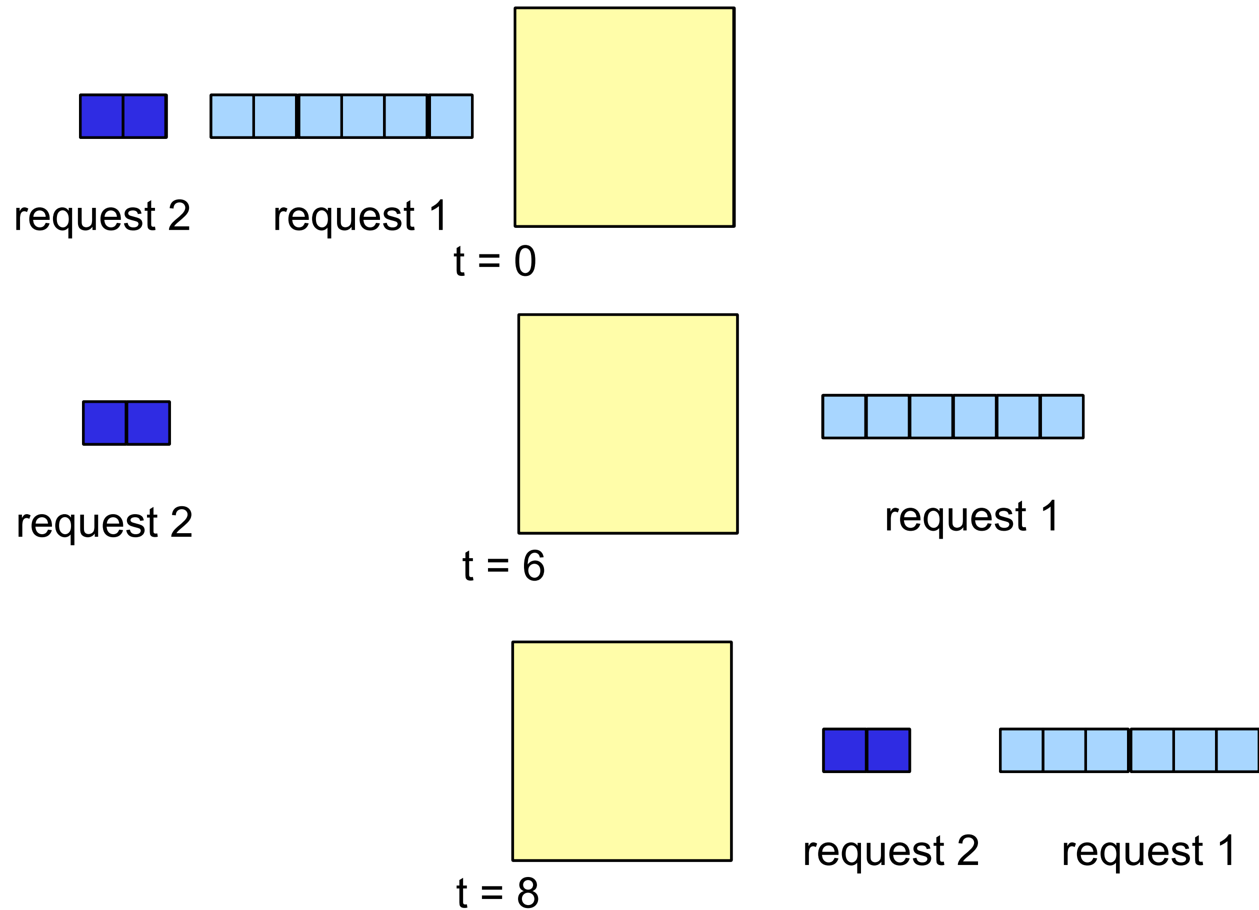
- Parallelism is concerned with performance of concurrent activities
  - weather forecasting
  - simulations

# Why Concurrency?

- In a serial environment, consider the following simple example of a server, serving requests from clients (e.g., a web server and web clients)

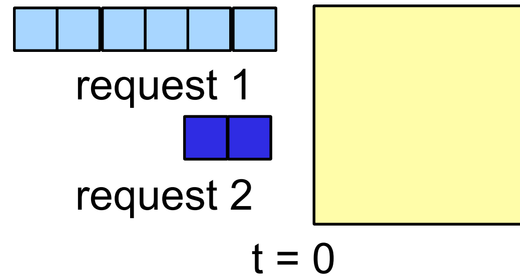


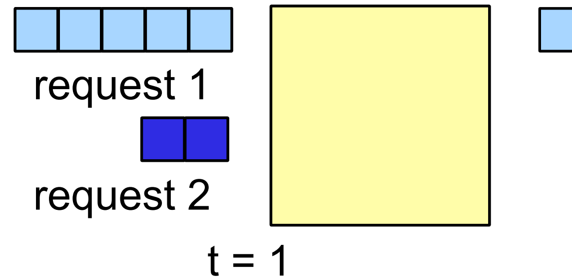
# Let us process requests serially

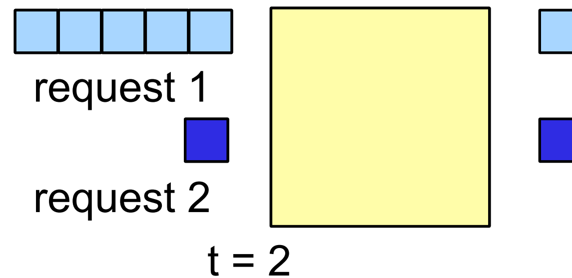



Total completion time = 8 units, Average service time = (6 + 8)/2 = 7 units

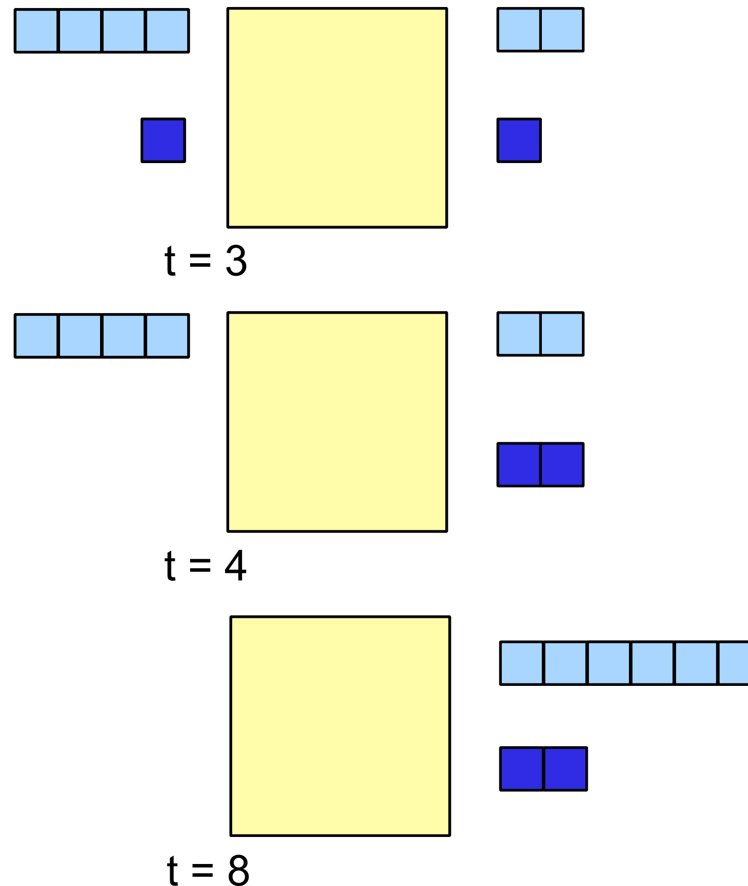
# Try a concurrent server now!



# We reduced mean service time!




Total completion time = 8 units, Average service time = (4 + 8)/2 = 6 units

# Why Concurrency?

- The lesson from the example is quite simple:
  - Not knowing anything about execution times, we can reduce average service time for requests by processing them concurrently!
- But what if I knew the service time for each request?
  - Would “shortest job first” not minimize average service time anyway?
  - Aha! But what about the poor guy standing at the back never getting any service (starvation/ fairness)?

# Why Concurrency?

- Notions of service time, starvation, and fairness motivate the use of concurrency in virtually all aspects of computing:
  - Operating systems are multitasking
  - Web/database services handle multiple concurrent requests
  - Browsers are concurrent
  - Virtually all user interfaces are concurrent

# Why Concurrency?

- In a parallel context, the motivations for concurrency are more obvious:
  - Concurrency + parallel execution = performance

# What is Parallelism?

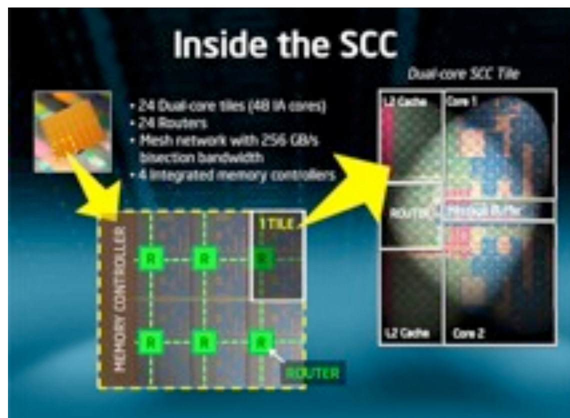
- Traditionally, the execution of concurrent tasks on platforms capable of executing more than one task at a time is referred to as “parallelism”
- Parallelism integrates elements of execution -- and associated overheads
- For this reason, we typically examine the correctness of concurrent programs and performance of parallel programs.

# Why Parallelism?

- We can broadly view the resources of a computer to include the processor, the data-path, the memory subsystem, the disk, and the network.
- Contrary to popular belief, each of these resources represents a major bottleneck.
- Parallelism alleviates all of these bottlenecks.

# Modern Architectures

***AMD***
***32 dual cores***

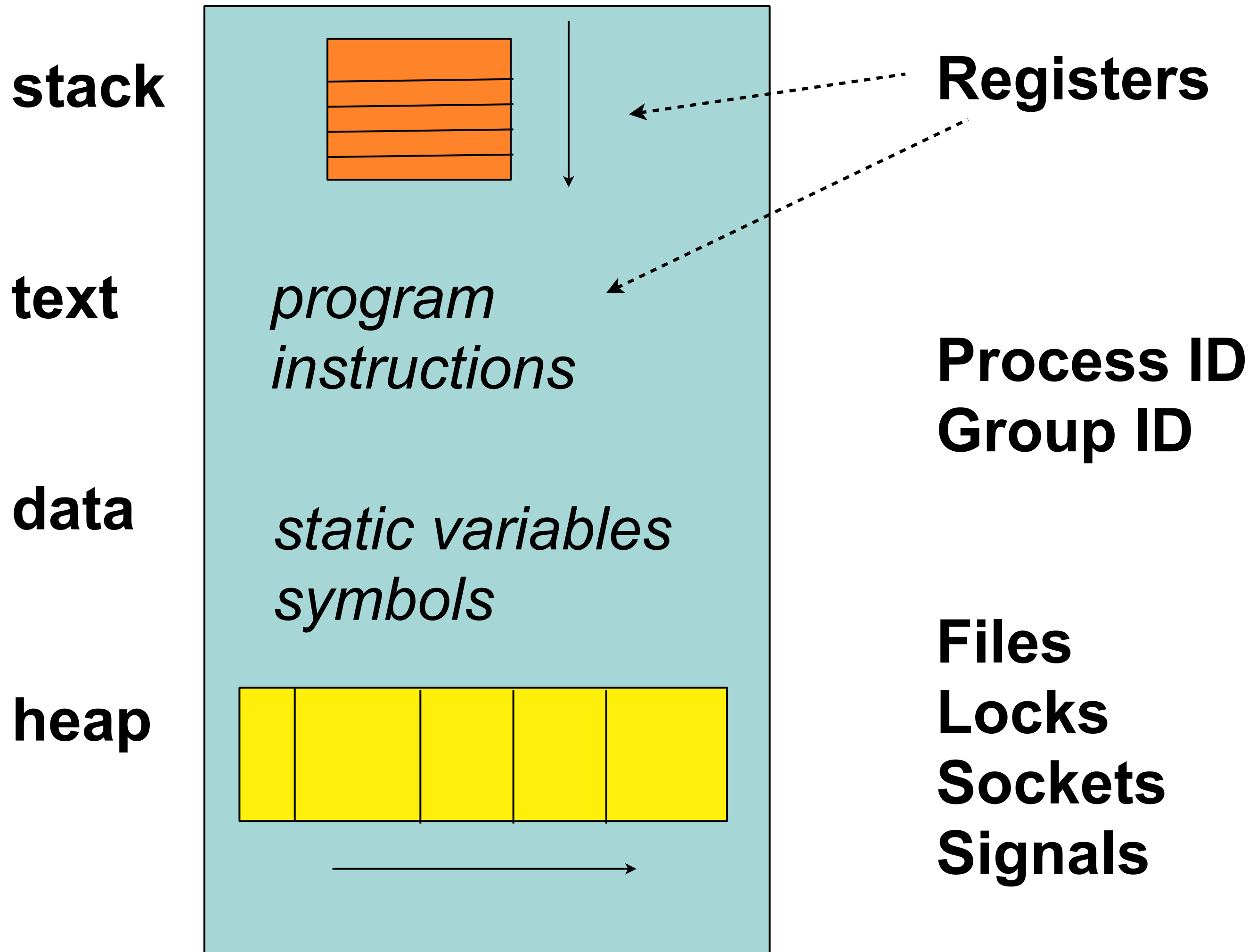



***SCC***
***24 dual cores***

***Azul***
***864 cores***
***16 x 54 cores***

*How should we program these kinds of machines?*

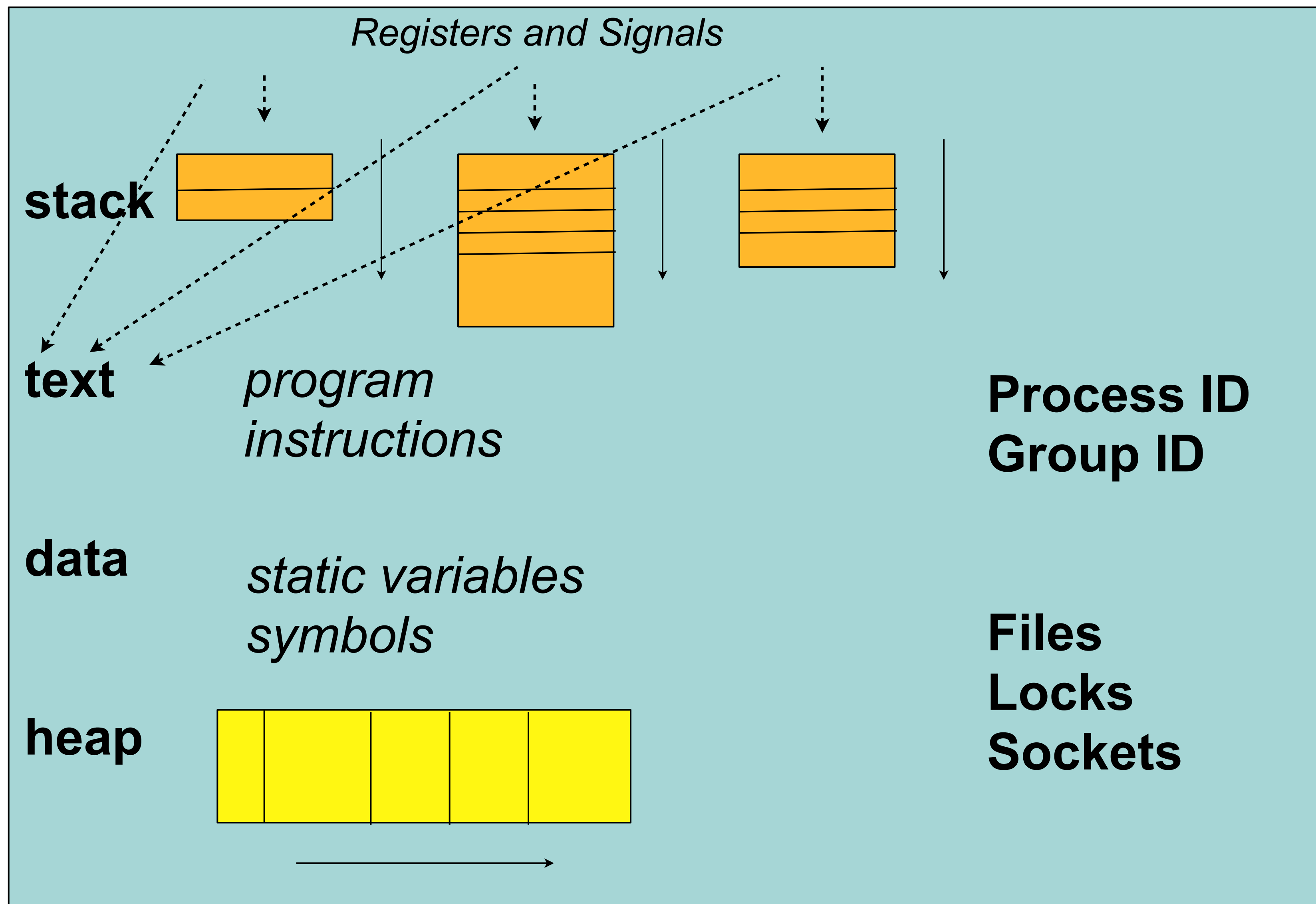
# A Process



# Threads and Processes

- Thread: an independent (concurrent) unit of execution that shares many resources with other threads
- Process: an independent (concurrent) unit of execution that is isolated from all other processes and shares no resources
- Resources:
  - Instructions
  - Registers
  - Stack
  - Heap
  - File descriptors
  - Shared libraries
  - Program instructions

# Threads within a Process



# Threads

- Exists within a process
  - But, independent control flow
  - share common process resources (like the heap and file descriptors)
    - changes made by one thread visible to others
    - pointers have meaning across threads
    - two threads can concurrently read and write to the same memory location
- Maintain their own stack pointer
- Registers
- Pending and blocked signals
- Can be scheduled by the operating system

# Desired structure

Programs can be decomposed into discrete (mostly) independent tasks

The points where they overlap should be easily discerned and amenable for protection

Three basic structures

*master-worker*
*result-oriented*
*pipeline-oriented*

# Architectural abstraction

- Shared memory
  - Every thread can observe actions of other threads on non-thread-local data (e.g., heap)
  - Data visible to multiple threads must be protected (*synchronized*) to ensure the absence of *data races*
    - A data race consists of two concurrent accesses to the same shared data by two separate threads, at least one of which is a write
- Thread safety
  - Suppose a program creates *n* threads, each of which calls the same procedure found in some library
  - Suppose the library modifies some global (shared) data structure
  - Concurrent modifications to this structure may lead to data corruption

# Example

```
THREAD 1
a = data;
a++;
data += a;
```

```
THREAD 2
b = data;
b++;
data += b;
```

**Assuming data = 0 initially, can data be 1 after the program completes?**