Principles of Concurrency and Parallelism

Xiangyu Zhang
xyzhang@cs.purdue.edu
http://www.cs.purdue.edu/homes/xyzhang
https://piazza.com/purdue/fall2014/cs353/home

Purdue University

CS 353 Topics

- Posix threads
- Principles of concurrency
  - Mutual exclusion
  - Correctness criteria
    - Sequential consistency, linearizability
  - Memory models
  - Races, atomicity violations, deadlocks and linearizability violations
    - Detection techniques
- Data structures and algorithms
  - Queques, heaps, trees, and lists
  - Sorting, graph algorithms

CS 353 Topics

- Message passing and Erlang
- Program parallelization
  - cost models, notions of speedup and efficiency, and tradeoffs between concurrency, latency and throughput

Grading

- Quizzes 15%
- Presentation 20%
  - Each student will give a 20 mins presentation about a paper chosen from a pool
- Programming Assignments 65%
  - 6 projects (roughly one project per 2 weeks)

Projects

- Using posix threads to build a parallel version of a prime number identification algorithm
- Deterministic Scheduler
- CHESS and Data race detection
- Using Java atomics to build lock free data structures
- Using Erlang
- Program parallelization

What is Concurrency

- Traditionally, the expression of a task in the form of multiple, *possibly interacting* subtasks, that may *potentially* be executed at the same time.

  - Concurrency is a programming concept. It says nothing about how the subtasks are actually executed.

  - Concurrent tasks may be executed serially or in parallel depending upon the underlying physical resources available.

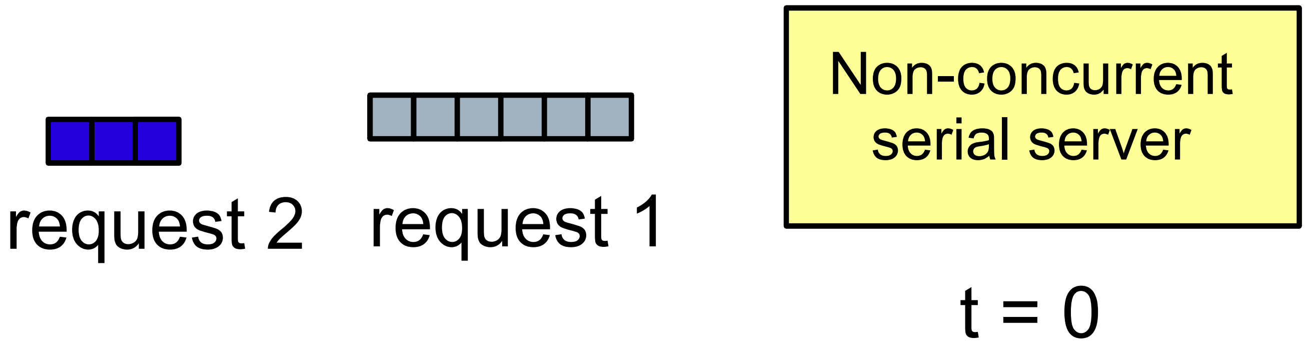
Why Concurrency?

- Concurrency plays a critical role in *sequential* as well as parallel/distributed computing environments.

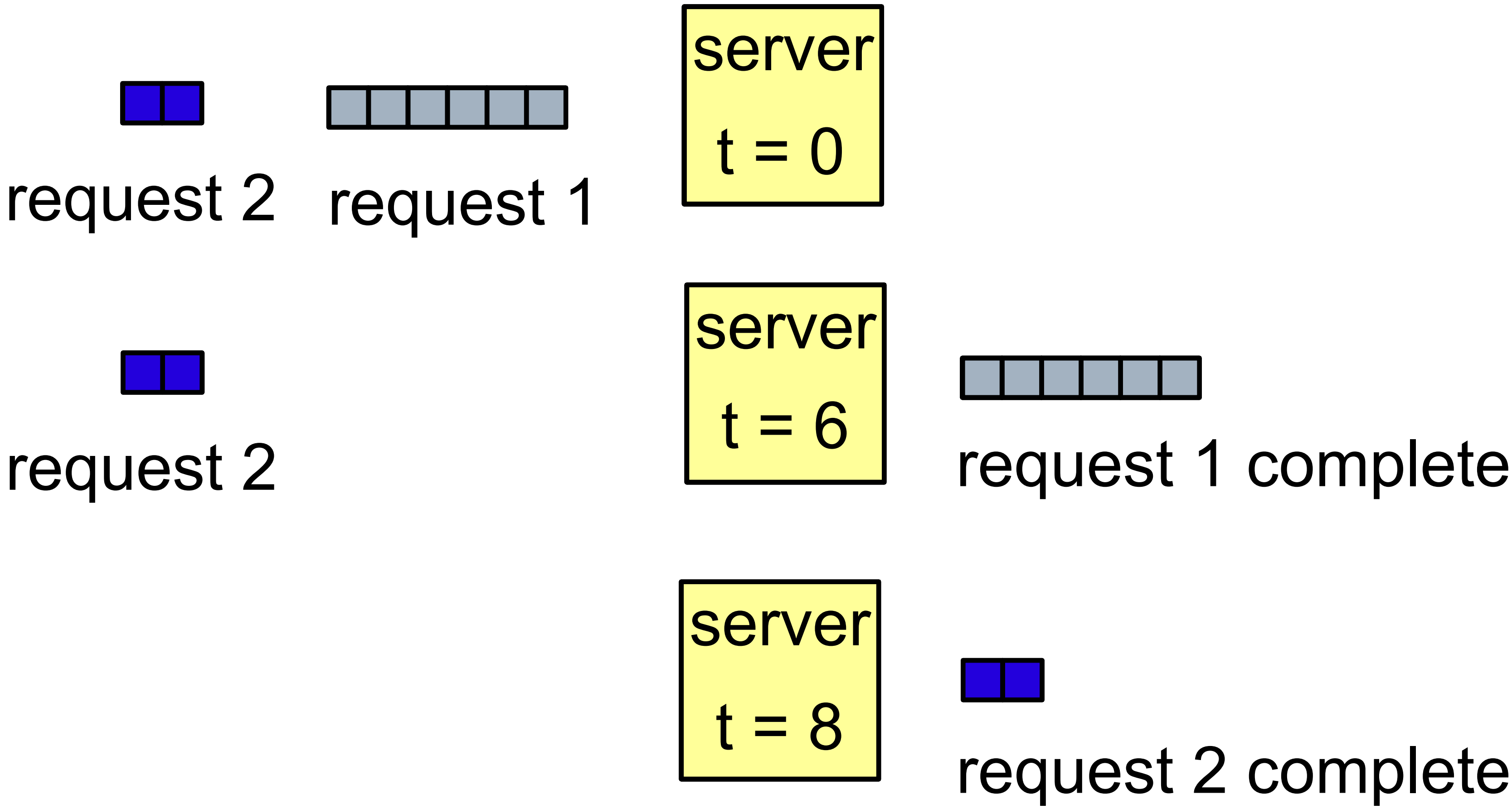
- It provides a way to *think and reason* about computations, rather than necessarily a way of improving overall performance.

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)

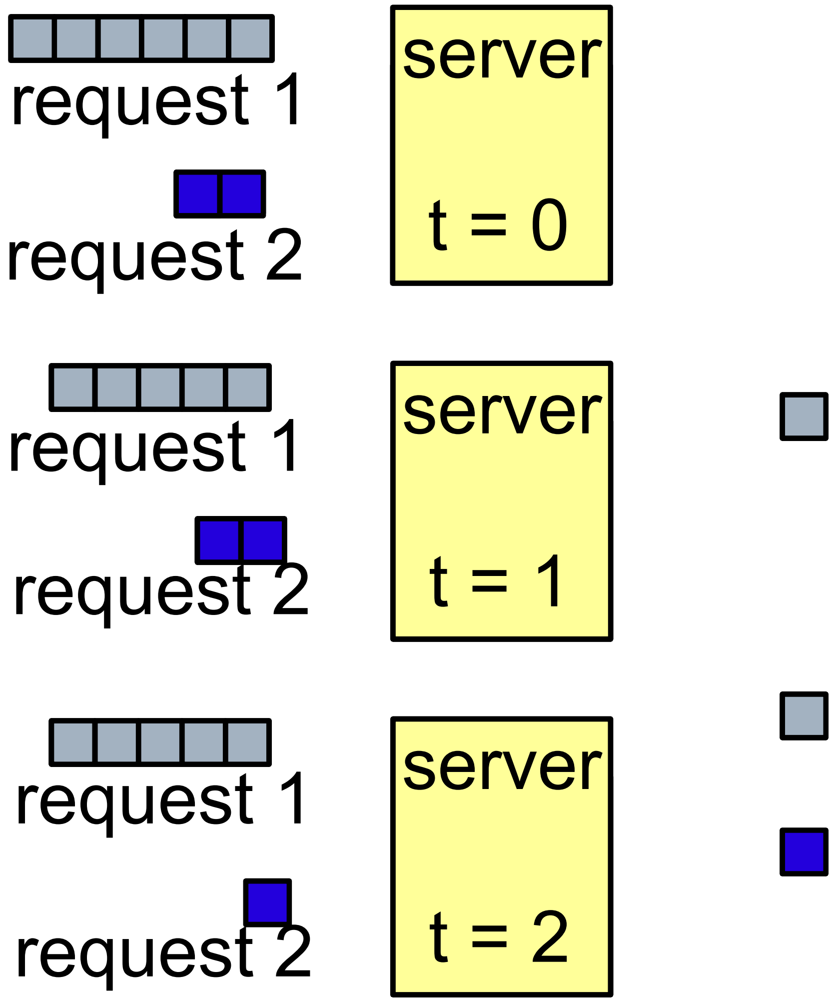


Serial Processing

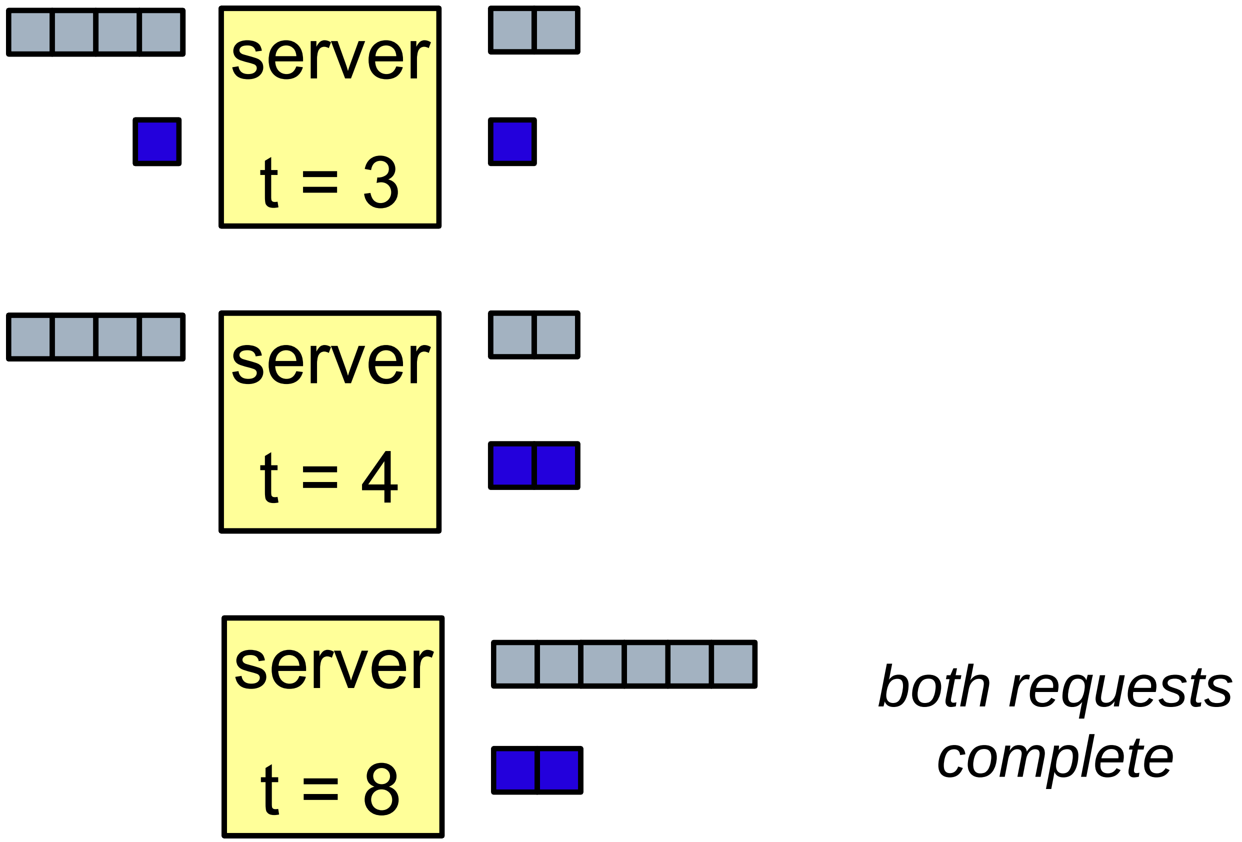


Total completion time = 8 units,
Average service time = (6 + 8)/2 = 7 units

Concurrent Processing



Mean Service Time Reduction



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

Why Parallelism

- Parallelizing disks
  - I/O (disks) represent major bottlenecks in terms of their bandwidth and latency
  - Parallelism enables us to extract data from multiple disks at the same time, effectively scaling the throughput of the I/O subsystem
    - An excellent example is the large server farms (several thousand computers) that ISPs maintain for serving content (html, movies, music, mail).

Why Parallelism

- Most programs are memory bound – i.e., they operate at a small fraction of peak CPU performance (10 – 20%)
- They are, for the most part, waiting for data to come from the memory.
- Parallelism provides multiple pathways to memory – effectively scaling memory throughput as well!

Why Parallelism

- The processor itself is the most obvious bottleneck.
- Moore's law states that the component count on a die doubles every 18 months.
- Contrary to popular belief, Moore's law says nothing about processor speed.
- What does one do with all of the available “components” on the die?

Parallelism in Processors

- The primary motivation for multicore processors, contrary to belief is not speed, it is power.
- Power consumption scales quadratically in supply voltage.
- Reduce voltage, simplify cores, and have more of them – this is the philosophy of multicore processors

Azul
864 cores
16 x 54 cores

Full cache coherence
But, slower processors
(roughly 1/3 speed of Core2 duo)