Principles of Concurrency

Suresh Jagannathan
suresh@cs.purdue.edu

http://www.cs.purdue.edu/homes/suresh/353-Spring2022

Course Overview

- Foundations

  Threads, Mutual Exclusion Algorithms,
  Concurrent Objects and Data Structures

- Models

  Linearizability, Relaxed Memory

- Languages and Concurrency Abstractions

  Rust, STM Haskell, Go, Erlang, Concurrent ML

- Testing and Verification

  Model Checking, Refinement, Rely/Guarantee, TLA

Grading and Evaluation

One semester-long project

- ★ initial proposal (due February 10th), approx. 2 pages
- ★ final proposal (due March 4th), approx. 5 pages
- ★ final report and demonstration (due April 28th), approx. 10 pages

Can work individually or in pairs

Can be on any topic covered in the syllabus

Examples:

- evaluate a tool
- implement or propose an algorithm/data structure
- devise an application
- explore compiler transformations
- examine language abstractions

Introduction

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.

Introduction

What is Concurrency?

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.

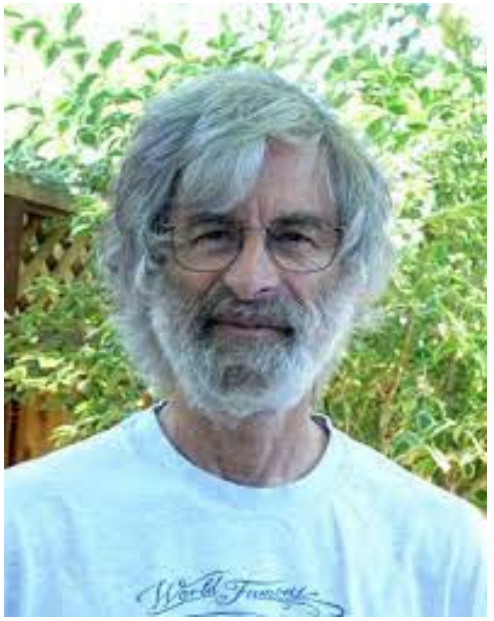
An Old Problem

Over de sequentialiteit van procesbeschrijvingen (EWD-35)

About the Sequentiality of Process Descriptions (1965)

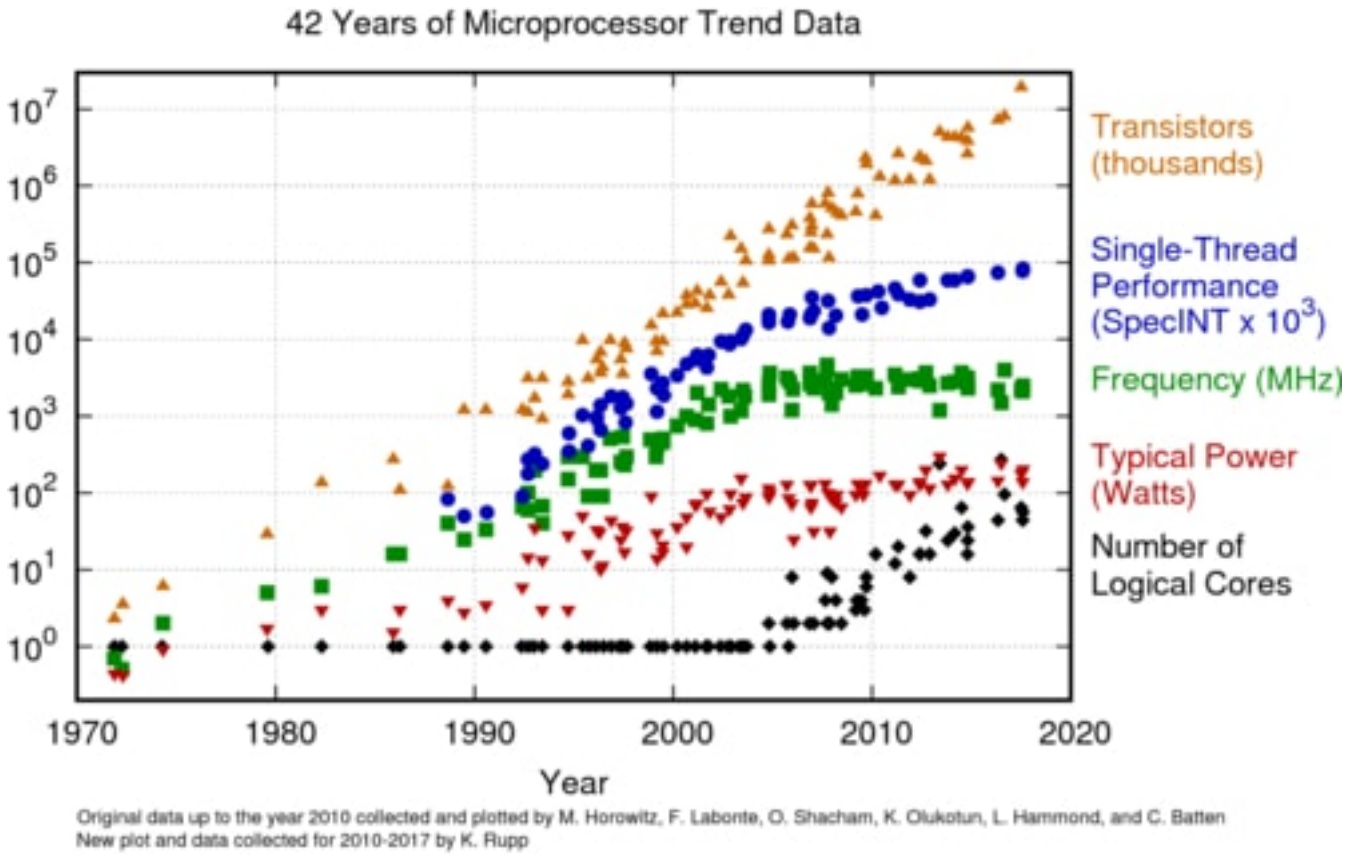
Simula 67: coroutines

Concurrent Pascal (1975)



A New Solution of Dijkstra's Concurrent Programming Problem (1974)

With New Challenges



Moore's law and rise of multicore machines

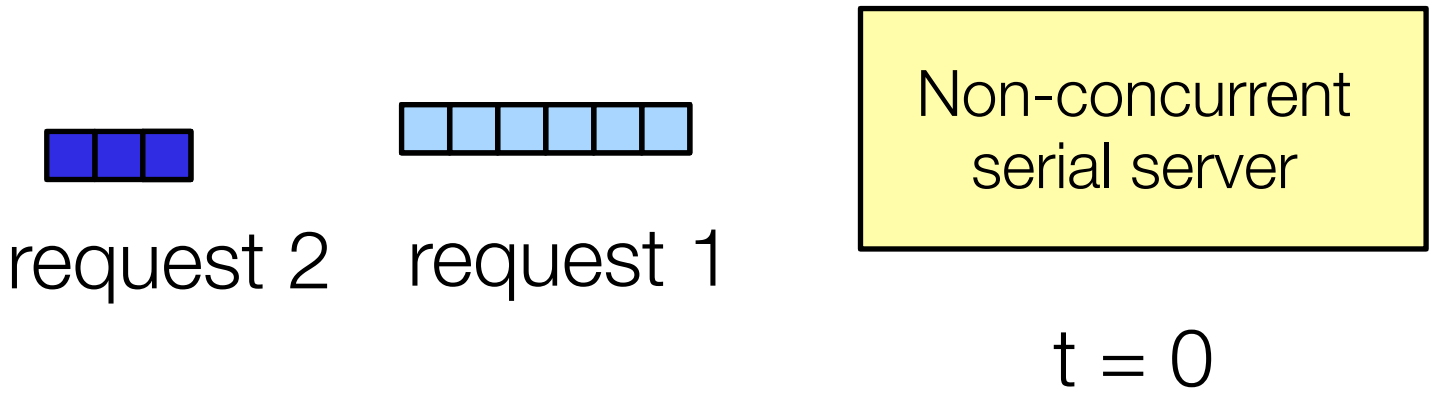
This is a software problem ..

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

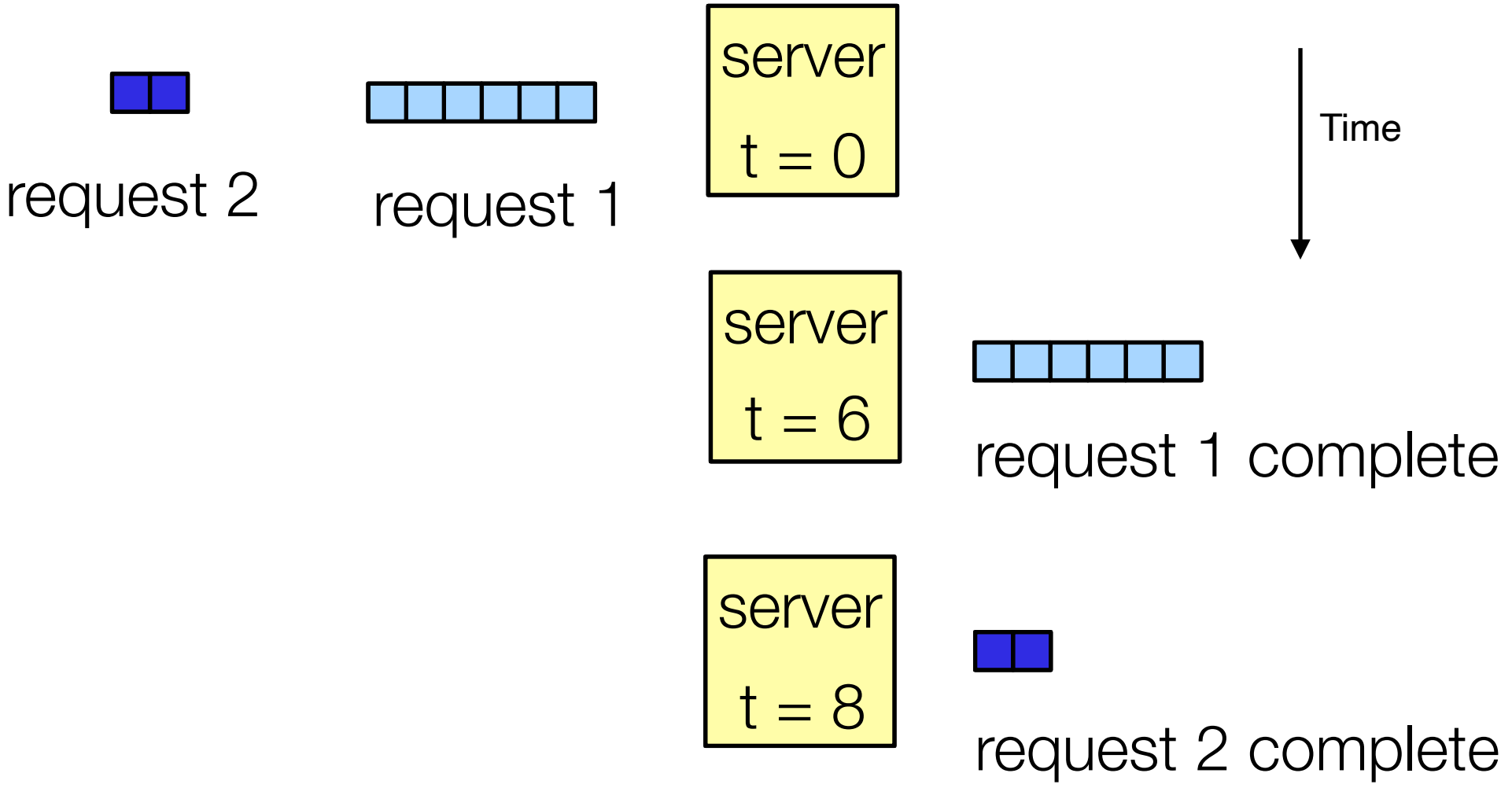
It provides a way to *think and reason* about computations, rather than necessarily a way of improving overall performance.

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)

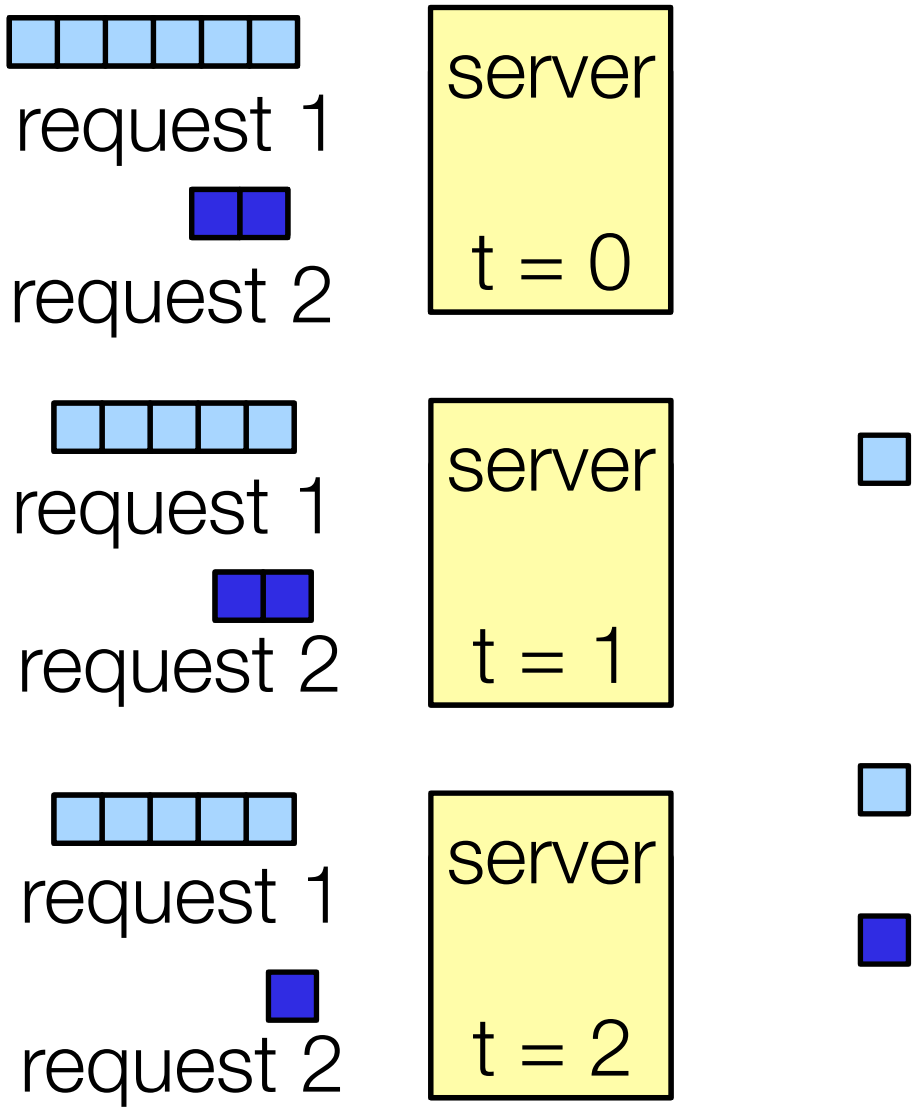


Serial Processing

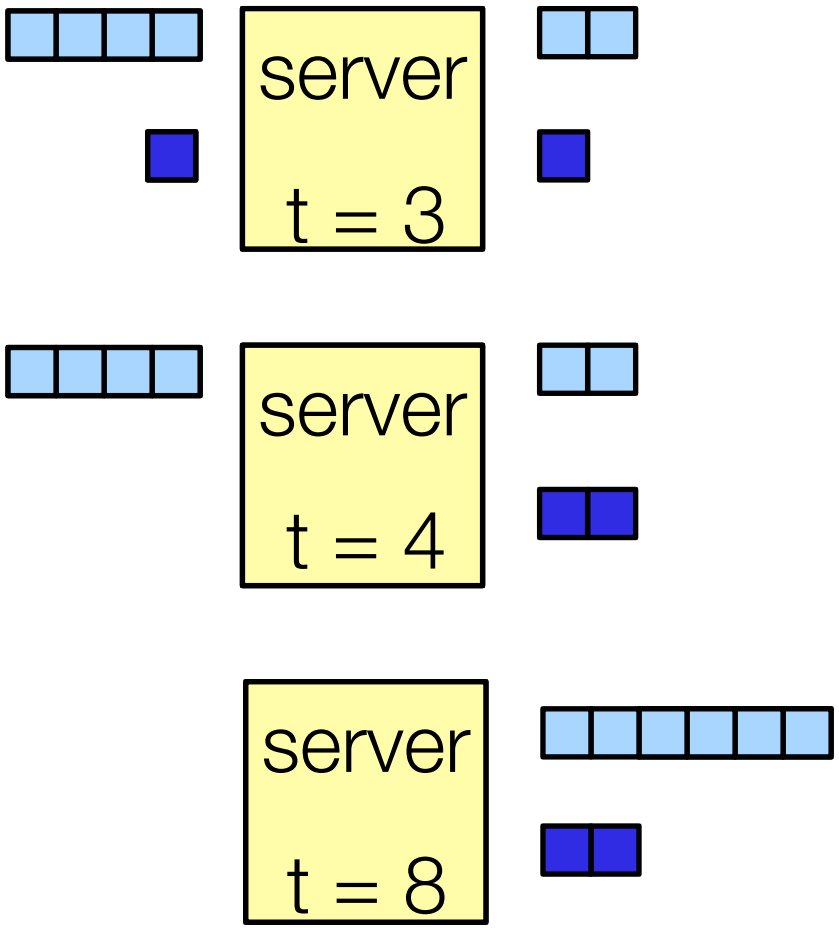


Total completion time = 8 units,
Average service time = (6 + 8)/2 = 7 units

Concurrent Processing



Mean Service Time Reduction



both requests complete

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
  - Parallelism increasingly important in a multicore era
- 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.

Our Focus

We'll concentrate on *concurrency* rather than parallelism in this course

- emphasis on programmability and correctness, rather than performance
  - how do we express the notion of a concurrent activity?
    - what is the "right" model for thinking about concurrency?
    - how are these models informed by hardware design and compilers?
  - how do we safely allow concurrent activities to interact with one another?
  - how do we identify, repair, or prevent errors due to unwanted or unexpected interaction?

What makes thinking about concurrency hard?

- How do we *order* concurrently executed events?
- How do we *coordinate* concurrently executing actions?
- How do we *communicate* effects from one concurrently executing action to another?
- When is it safe to make the effects of one action *visible* to another?

Tradeoffs

- Enforce strong ordering properties:
  - reduced set of allowed behaviors
- Enforce strong visibility properties:
  - high coordination overhead
- Enforce precise communication effects:
  - non-local control-flow