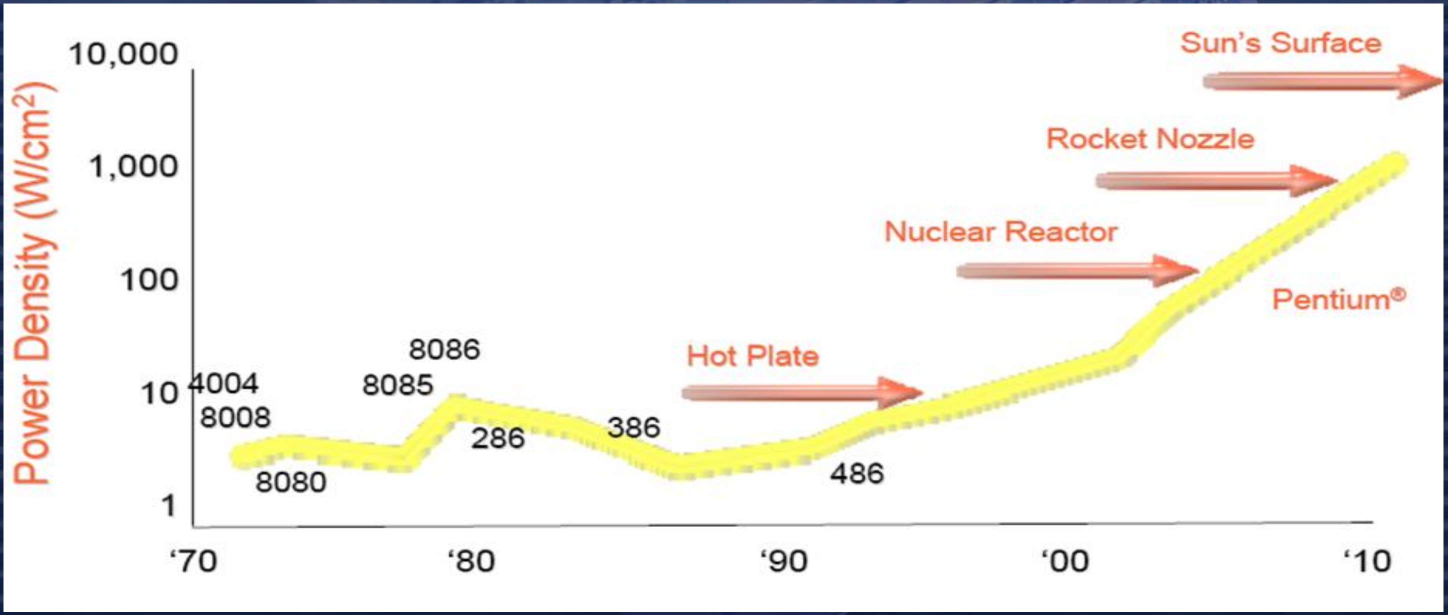
CHALMERS

Parallel Programming

Philippas Tsigas
Chalmers University of Technology
Computer Science and Engineering Department

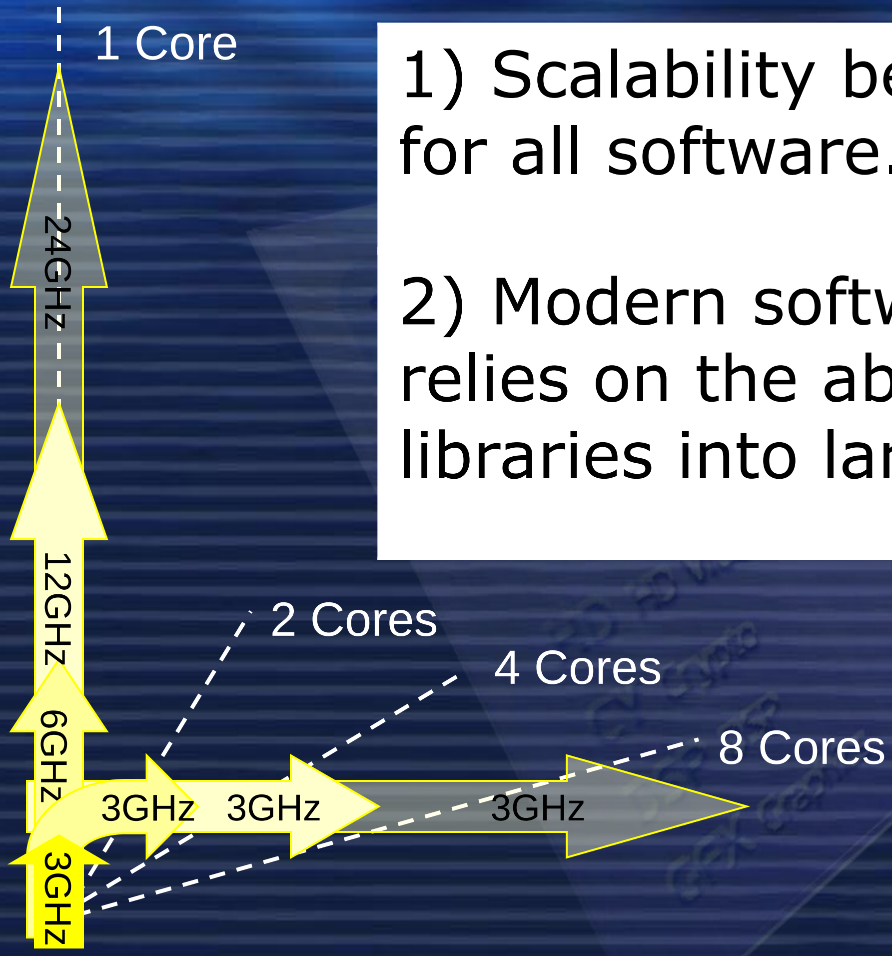
WHY PARALLEL PROGRAMMING IS ESSSENTIAL IN DISTRIBUTED SYSTEMS AND NETWORKING

How did we reach there?



Picture from Pat Gelsinger, Intel Developer Forum, Spring 2004 (Pentium at 90W)

Concurrent Software Becomes Essential



1) Scalability becomes an issue for all software.

2) Modern software development relies on the ability to compose libraries into larger programs.

Our work is to help the programmer to develop efficient parallel programs but also survive the multicore transition.

DISTRIBUTED APPLICATIONS

Data Sharing: Gameplay Simulation as an example

This is the hardest problem…

- 10,000’s of objects
- Each one contains mutable state
- Each one updated 30 times per second
- Each update touches 5-10 other objects

Slide:
Tim Sweeney
CEO Epic Games
POPL 2006

Manual synchronization (shared state concurrency) is
hopelessly intractable here.
Solutions?

Distributed Applications Demand

Quite High Level Data Sharing:

- *Commercial computing* (media and information processing)
- Control Computing (on board flight-control system)

Scene from World of Warcraft

NETWORKING

40 multithreaded packet-processing engines

http://www.cisco.com/assets/cdc_content_elements/embedded-video/routers/popup.html

- On chip, there are 40 32-bit, 1.2-GHz packet-processing engines. Each engine works on a packet from birth to death within the Aggregation Services Router.
- each multithreaded engine handles four threads (each thread handles one packet at a time) so each QuantumFlow Processor chip has the ability to work on 160 packets concurrently

DATA SHARING

Blocking Data Sharing

A typical Counter Impl:

```
class Counter {
  int next = 0;

  synchronized int getNumber () {
    int t;
    t = next;
    next = t + 1;
    return t;
  }
}
```

Thread1: getNumber()

t = 0

result=0

Thread2: getNumber()

Lock acquired

Lock released

result=1

next = 2

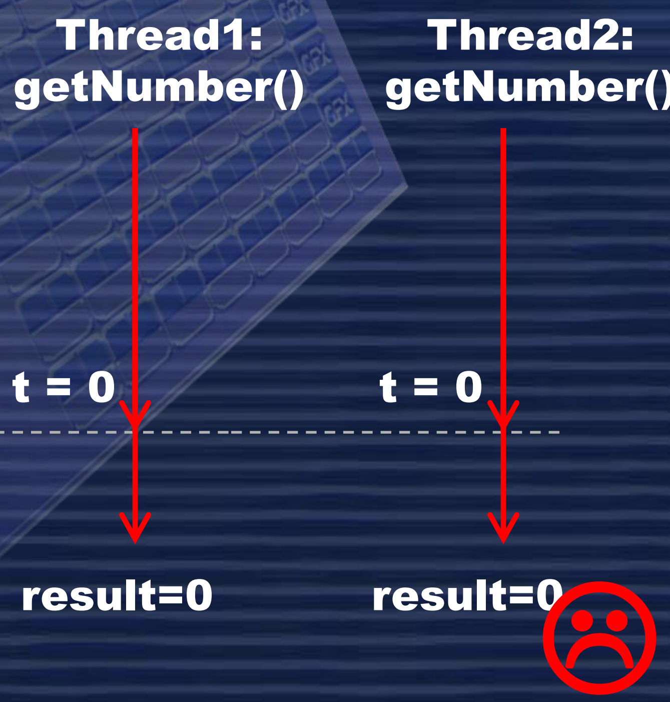
Do we need Synchronization?

What can go wrong here?

```
class Counter {
  int next = 0;

  int getNumber () {
    int t;
    t = next;
    next = t + 1;
    return t;
  }
}
```

next = 0

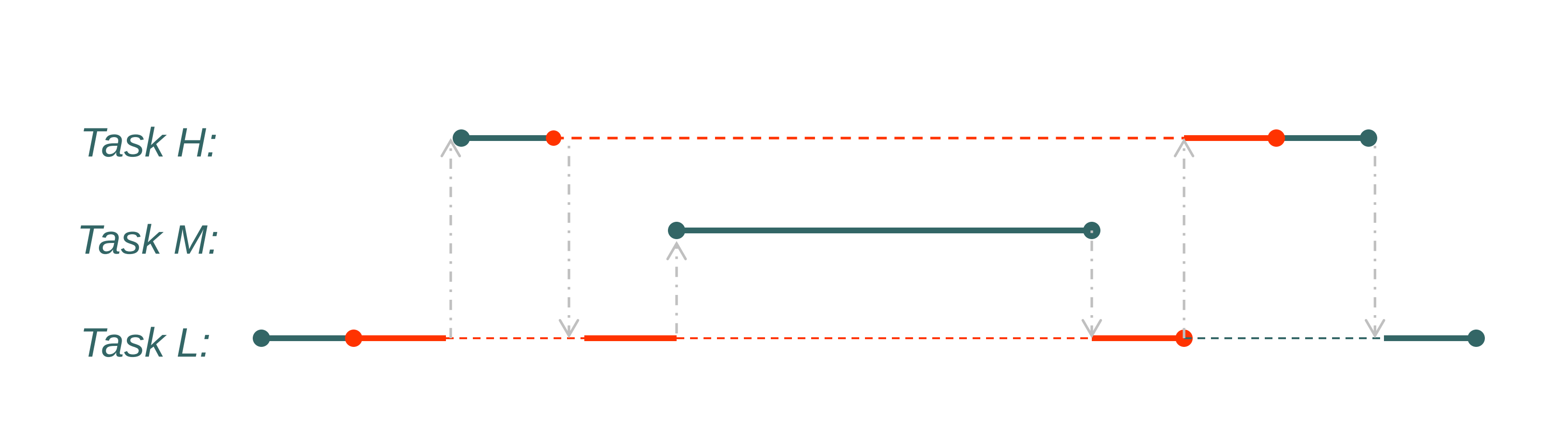


Blocking Synchronization = Sequential Behavior

BS ->Priority Inversion

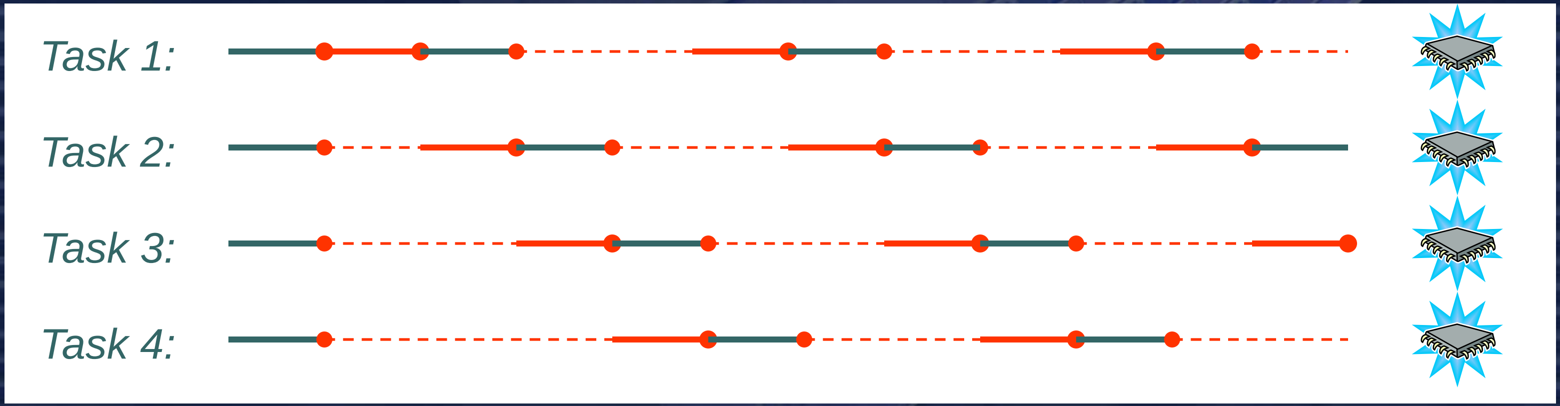
- A high priority task is delayed due to a low priority task holding a shared resource. The low priority task is delayed due to a medium priority task executing.

- Solutions: Priority inheritance protocols
  - Works ok for single processors, but for multiple processors …



Critical Sections + Multiprocessors

- **Reduced Parallelism**. Several tasks with overlapping critical sections will cause waiting processors to go idle.



The BIGEST Problem with Locks?

Blocking Locks are not composable

All code that accesses a piece of shared state must know and obey the locking convention, regardless of who wrote the code or where it resides.

Interprocess Synchronization = Data Sharing

- **Synchronization** is **required** for concurrency
- Mutual exclusion (Semaphores, mutexes, spin-locks, disabling interrupts: Protects critical sections)
  - Locks limits concurrency
  - Busy waiting – repeated checks to see if lock has been released or not
  - Convoying – processes stack up before locks
  - **Blocking Locks are not composable**
  - All code that accesses a piece of shared state must know and obey the locking convention, regardless of who wrote the code or where it resides.
- A **better** approach is to use data structures that are …

A Lock-free Implementation

- In this case a non-blocking design is easy:

```
class Counter {
  int next = 0;

  int getNumber () {
    int t;
    do {
      t = next;
    } while (CAS (&next, t, t + 1) != t);
    return t;
  }
}
```

Atomic compare and swap

New value

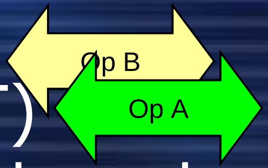
Expected value

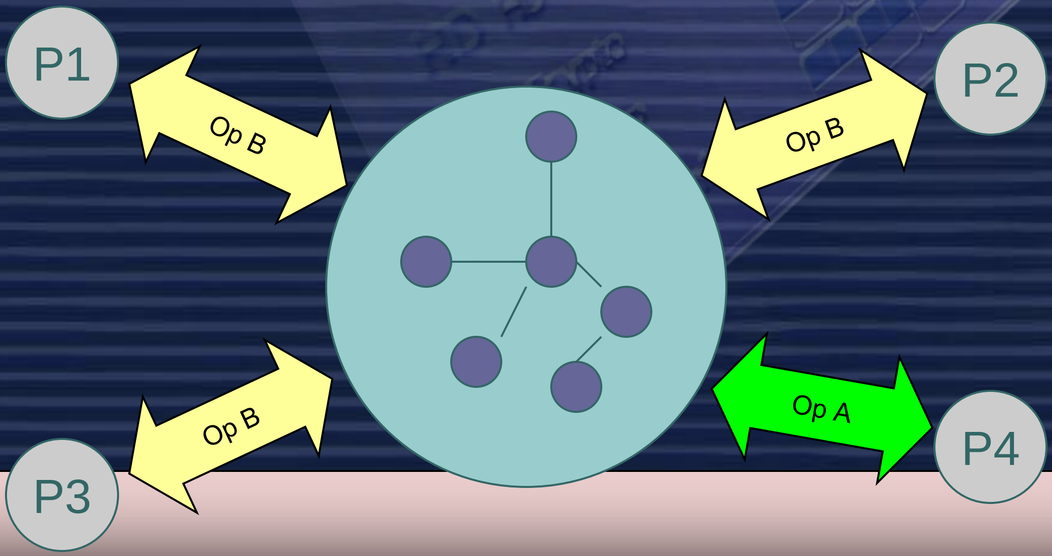
Location

LET US START FROM THE BEGINING

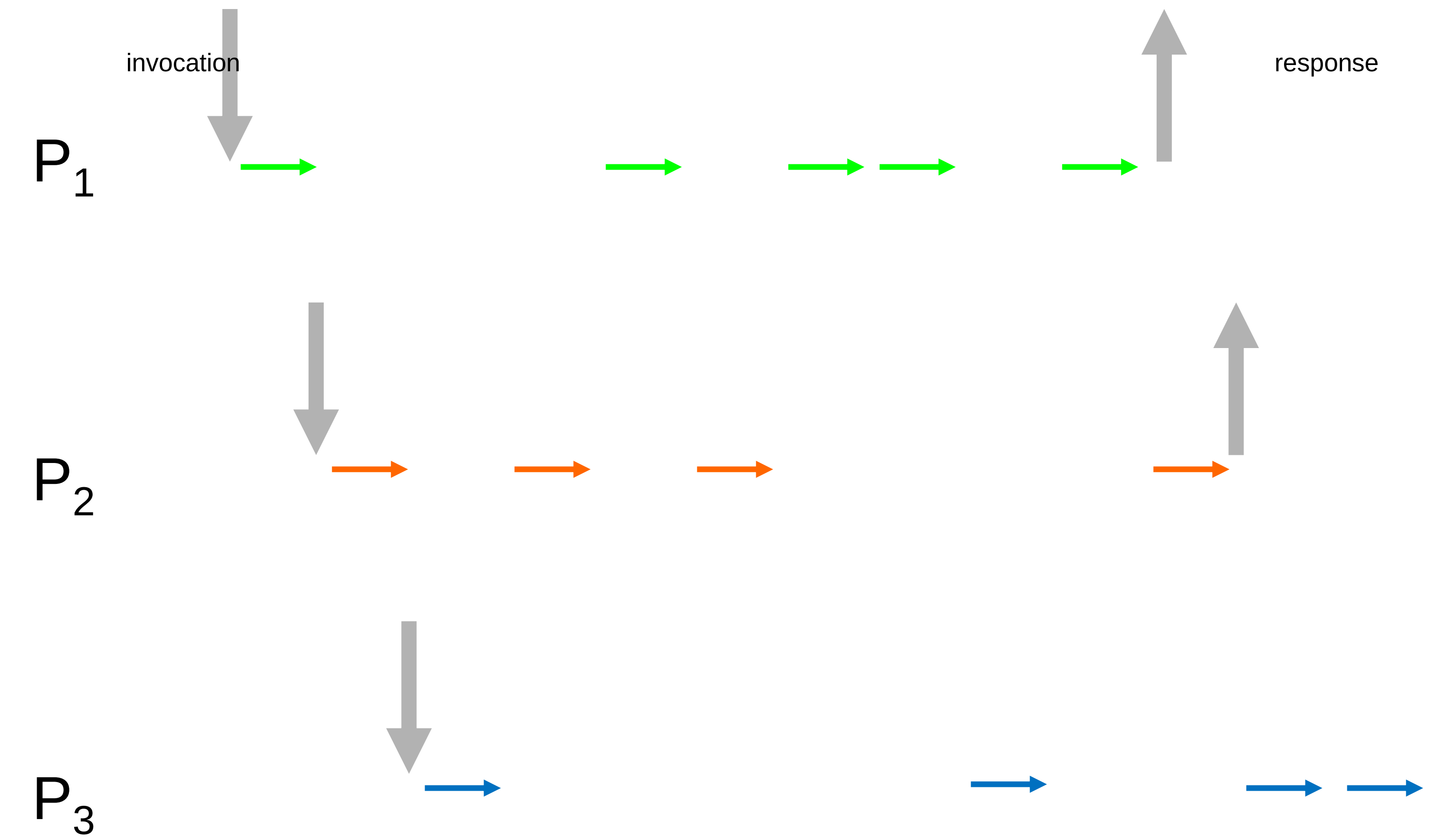
Shared Memory Data-structures

- Object in shared memory
  - Supports some set of operations (ADT)
  - Concurrent access by many processes/threads
  - Useful to e.g.
    - Exchange data between threads
    - Coordinate thread activities

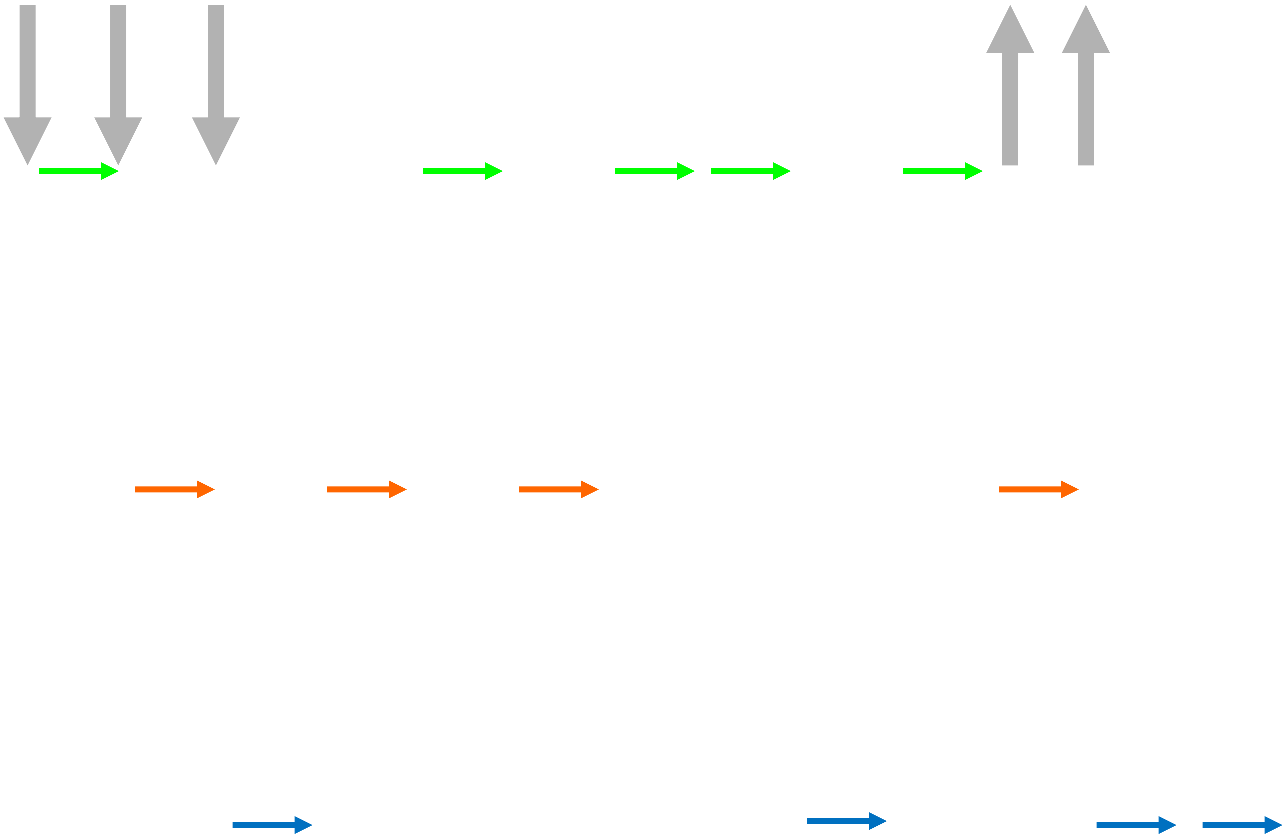




Executing Operations



Interleaving Operations

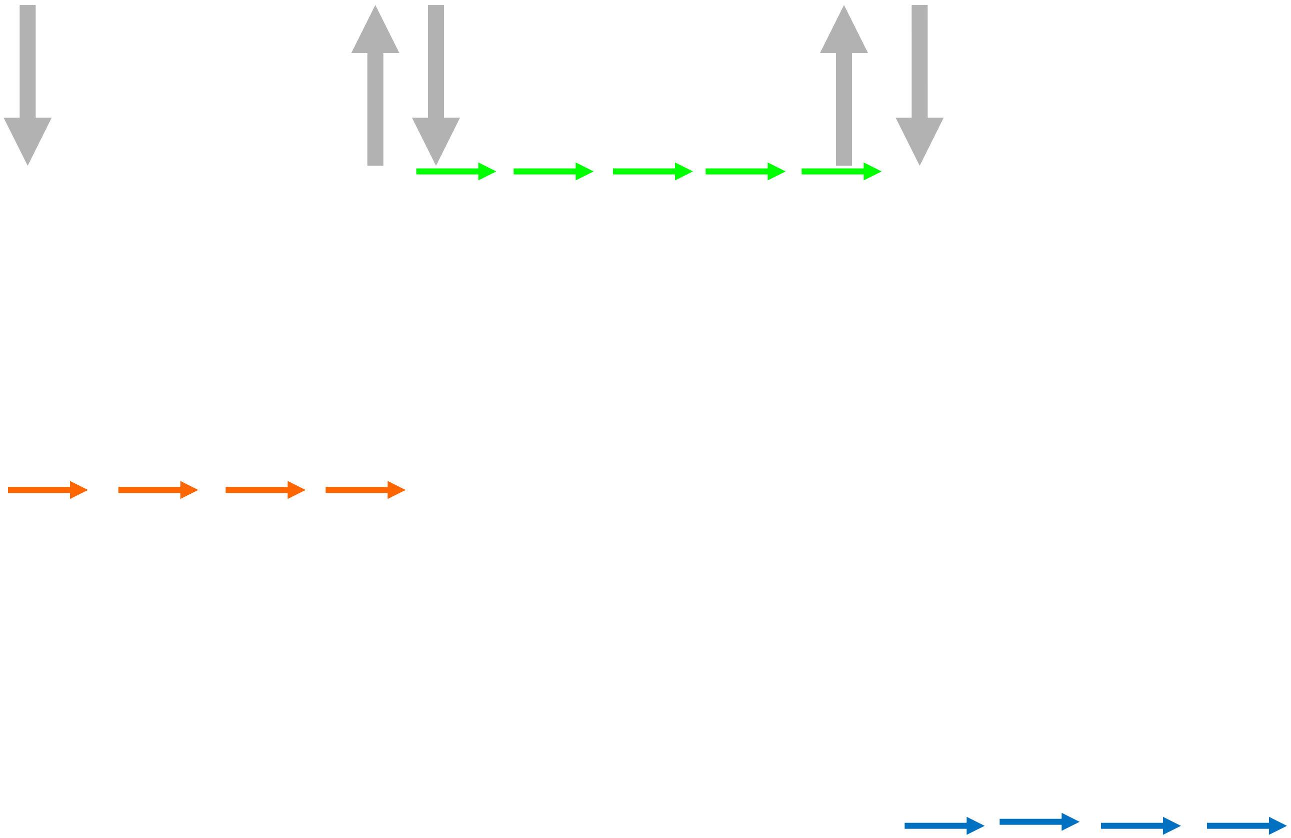


Concurrent execution

Interleaving Operations

(External) behavior

Interleaving Operations, or Not



Sequential execution

Interleaving Operations, or Not

- Sequential behavior: invocations & response alternate and match (on process & object)
- Sequential specification: All the legal sequential behaviors, satisfying the semantics of the ADT
  - E.g., for a (LIFO) stack: pop returns the last item pushed

Correctness: Sequential consistency

[Lamport, 1979]

- For every concurrent execution there is a sequential execution that
  - Contains the same operations
  - Is legal (obeys the sequential specification)
  - Preserves the order of operations by the same process

Sequential Consistency: Examples

Concurrent (LIFO) stack

push(4)

push(7)

pop():4

Last In

First Out

✓

push(4)

push(7)

pop():4

✓

Sequential Consistency: Examples

Concurrent (LIFO) stack

push(4)

push(7)

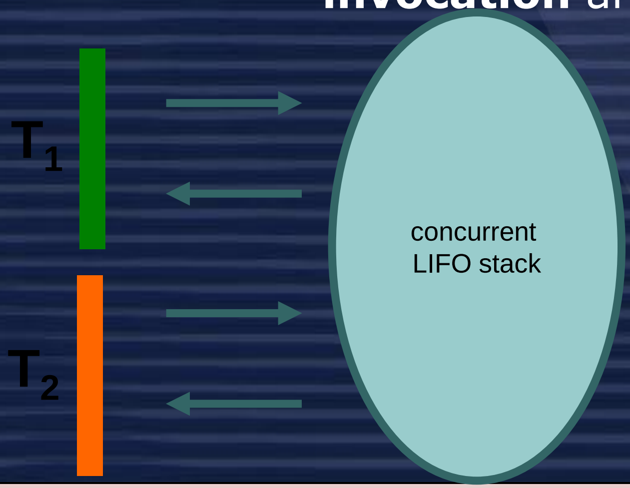
pop():7

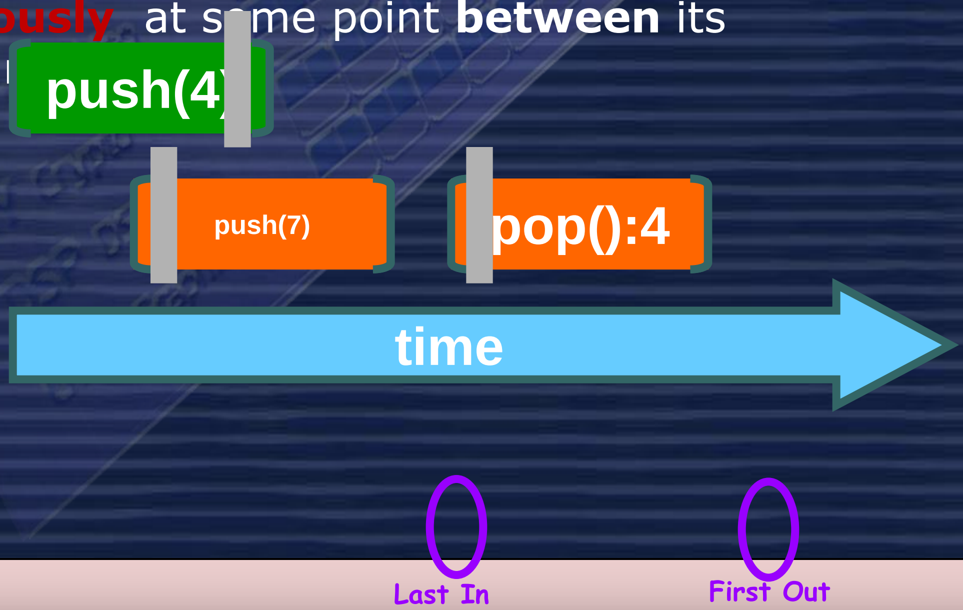
Last In

First Out

Safety: Linearizability

- **Linearizable data structure**
  - **Sequential specification** defines legal sequential executions
  - Concurrent operations allowed to be **interleaved**
  - Operations **appear to execute atomically**
    - External observer gets the **illusion** that each operation **takes effect instantaneously** at some point **between** its **invocation** and its r





Safety II

An accessible node is never freed.

Liveness

Non-blocking implementations

- **Wait-free implementation of a DS** [Lamport, 1977]
  - Every operation finishes in a finite number of its own steps.
- **Lock-free (≠ FREE of LOCKS) implementation of a DS** [Lamport, 1977]
  - At least one operation (from a set of concurrent operation) finishes in a finite number of steps (the data structure as a system always make progress)

Liveness II

- every garbage node is eventually collected

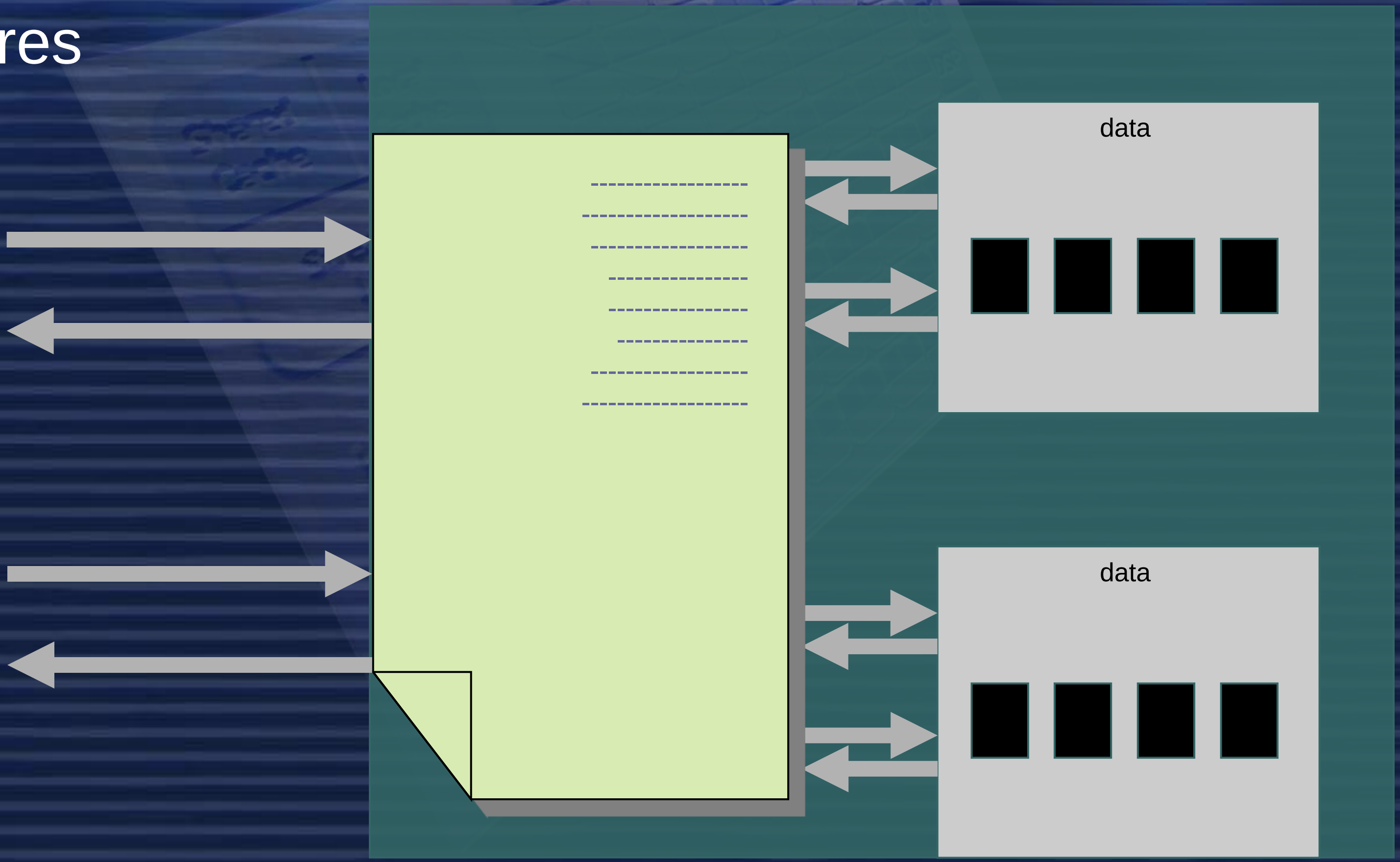
Abstract Data Types (ADT)

- Cover most concurrent applications
  - At least encapsulate their data needs
  - An object-oriented programming point of view
- Abstract representation of data & set of methods (operations) for accessing it
  - Signature
  - Specification

Implementing High-Level ADT

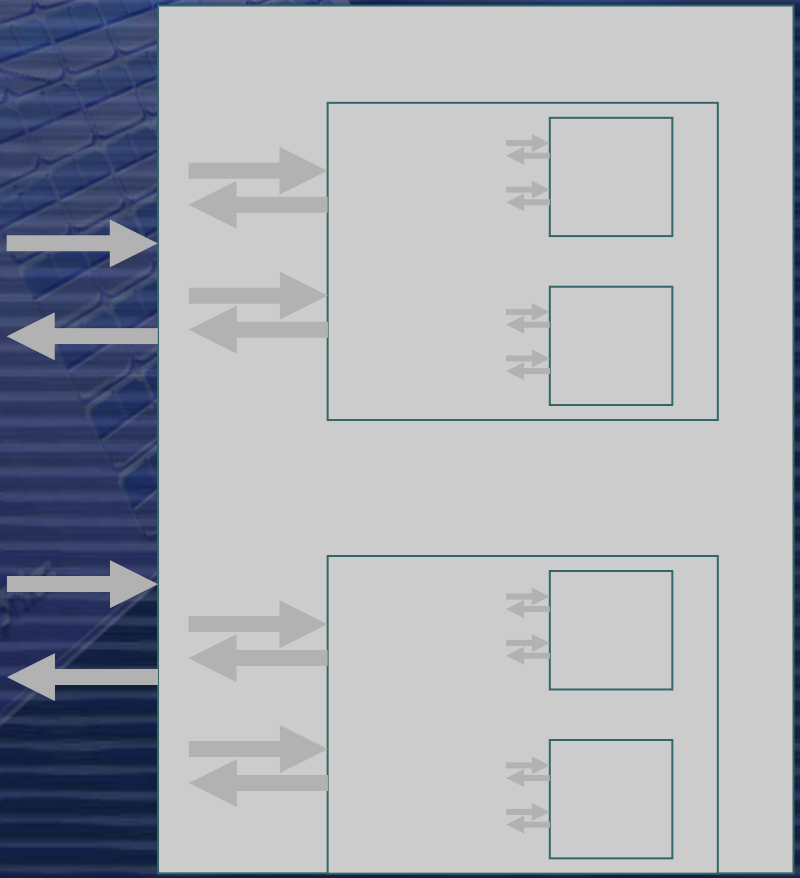
Using lower-level ADTs

& procedures



Lower-Level Operations

- High-level operations translate into primitives on base objects that are available on H/W
  - Obvious: read, write
  - Common: compare&swap (CAS), LL/SC, FAA



Our Research: Non-Blocking Synchronization for Accessing Shared Data

We are trying to design efficient non-blocking implementations of building blocks that are used in concurrent software design for data sharing.