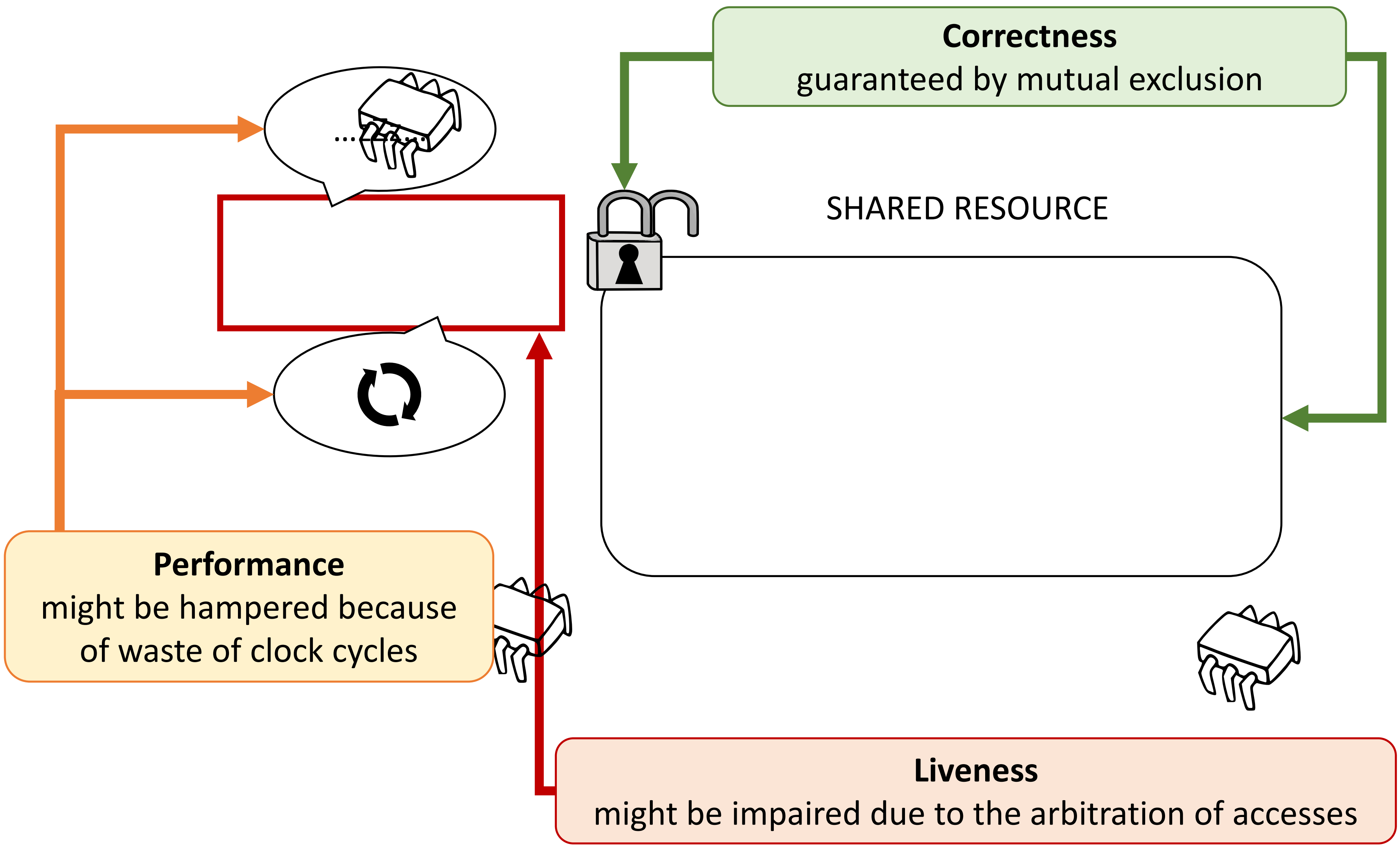
Programmazione concorrente

Laurea Magistrale in Ingegneria Informatica
Università Tor Vergata
Docente: Romolo Marotta

Properties of Concurrent Programs

1. Scalability
2. Correctness
3. Progress

On concurrent programming



What do we want from parallel programs?

- **Safety:** *nothing wrong happens* (Correctness)
  - parallel versions of our programs should be correct as their sequential implementations
- **Liveliness:** *something good happens eventually* (Progress)
  - if a sequential program terminates with a given input, we want that its parallel alternative also completes with the same input
- **Performance**
  - we want to exploit our parallel hardware

A bit of terminology

- Hardware
  - Processor
  - CPU
  - CPU-Core
  - Logical Core
  - Hardware thread
- Software
  - Process
  - Thread
  - Fiber
  - Task
- Programs
  - Sequential
  - Concurrent
  - Parallel
  - Distrubuted
- Memory
  - Shared
  - Distributed

The system model

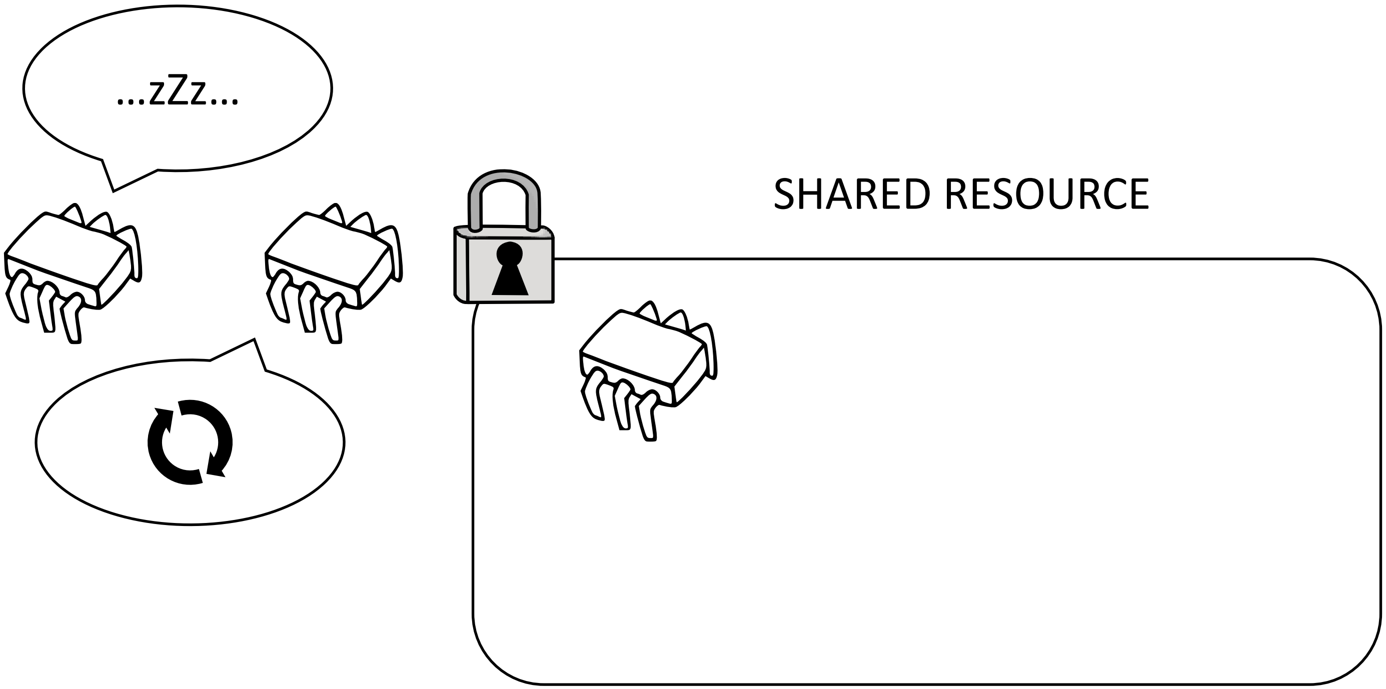
- Threads (aka processes)
- Cores (aka cpus)
- Shared memory
- Arbitrary long asynchronous delays
- Scheduler
  - A system component that decides which/when a thread runs on a given core

Scalability

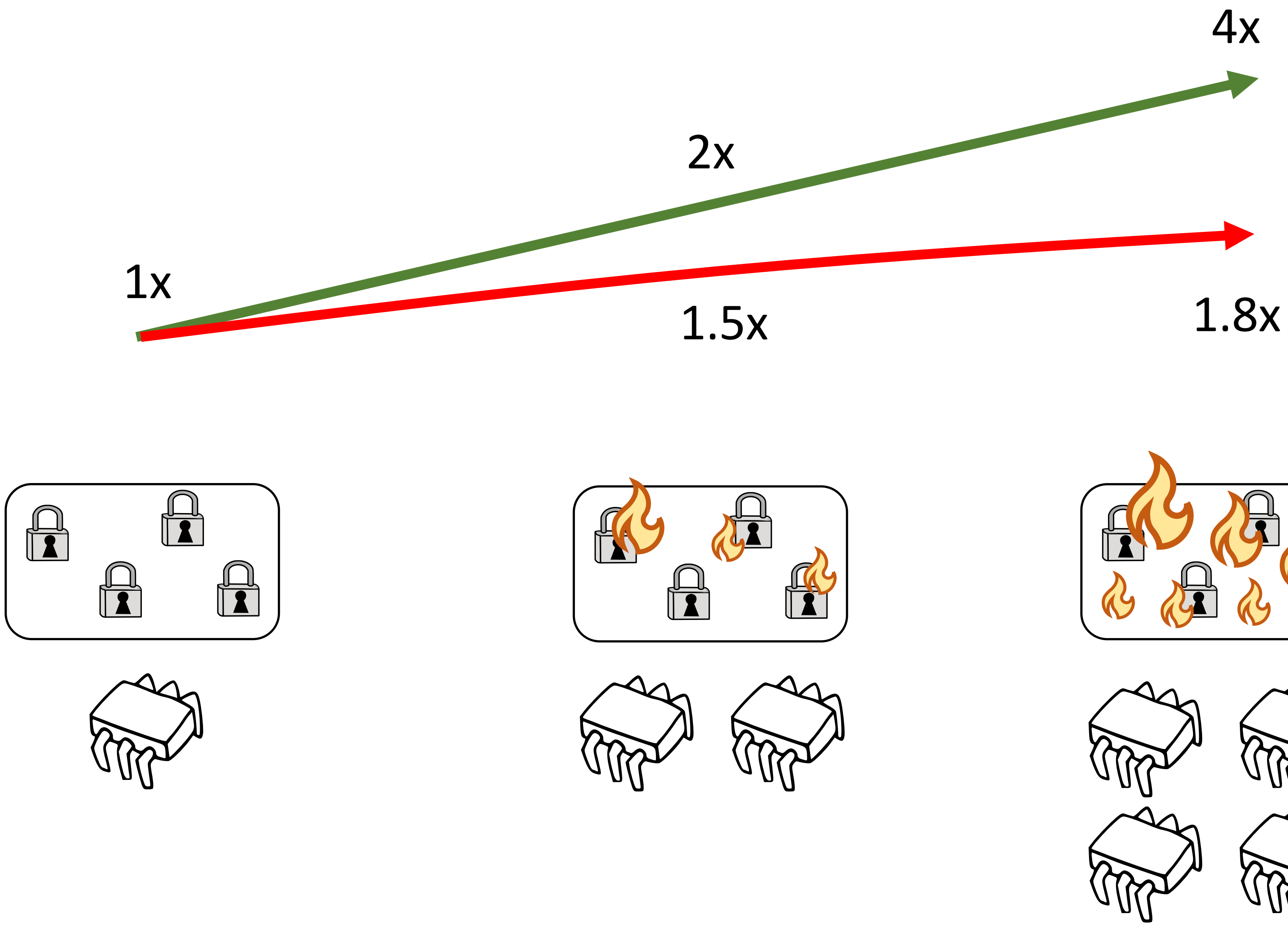
Correctness conditions

Progress conditions

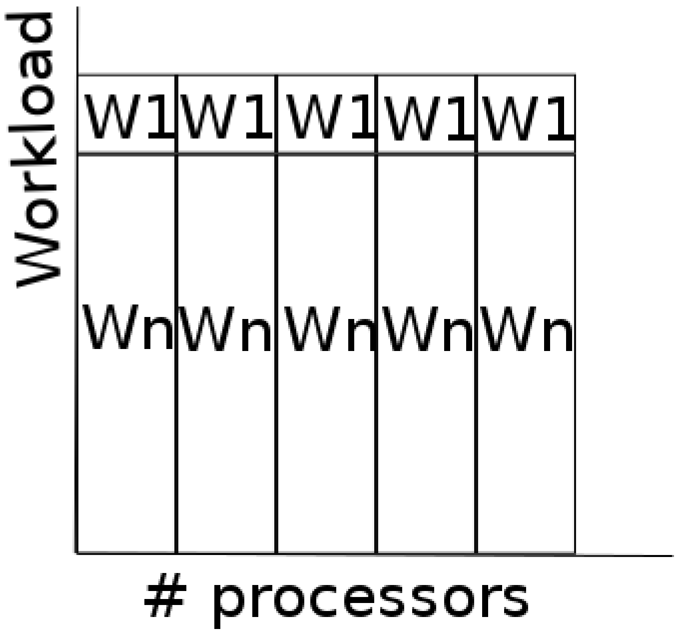
The cost of synchronization

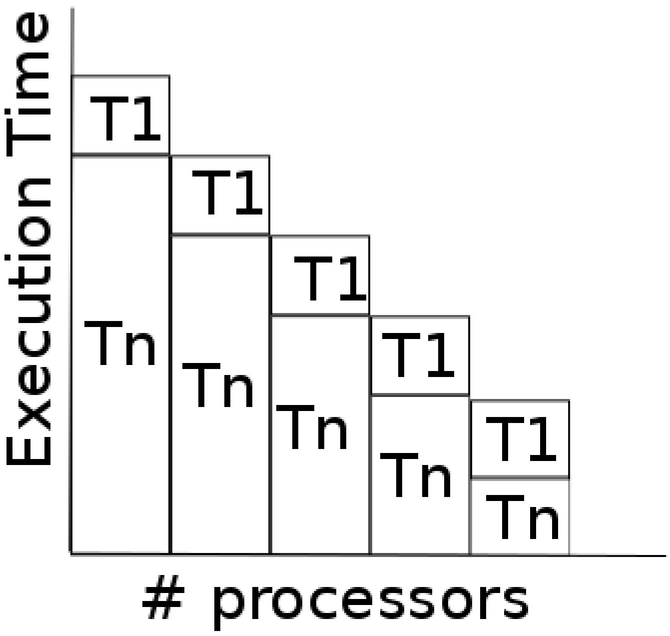


The cost of synchronization



Amdahl Law – Fixed-size Model (1967)





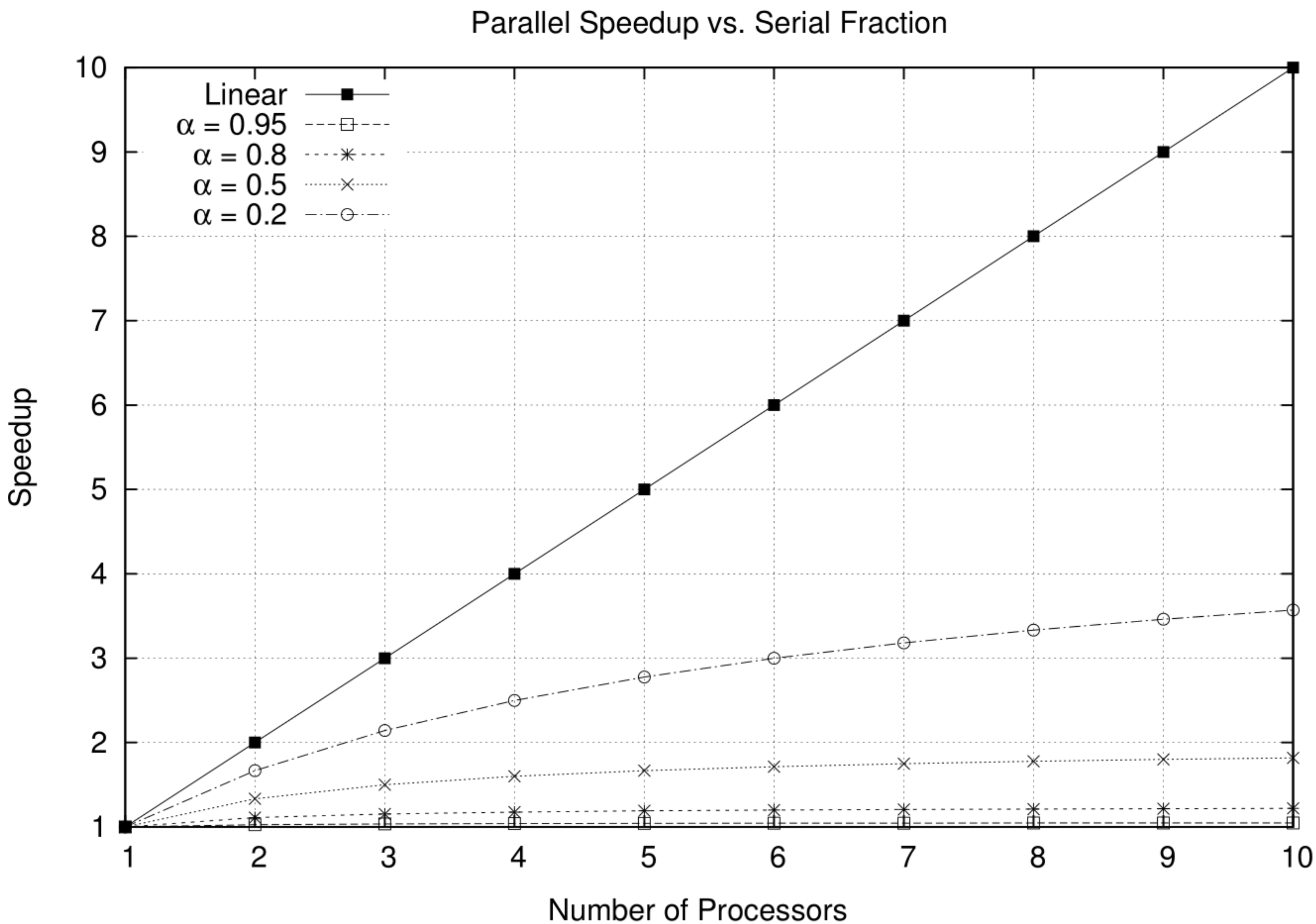
Amdahl Law – Fixed-size Model (1967)

- The workload is fixed: it studies how the behavior of the same program varies when adding more computing power

$$S_{Amdahl} = \frac{T_s}{T_p} = \frac{T_s}{\alpha T_s + (1-\alpha)\frac{T_s}{p}} = \frac{1}{\alpha + \frac{(1-\alpha)}{p}}$$

- where:
  - $\alpha \in [0,1]$: Serial fraction of the program
  - $p \in N$: Number of processors
  - T_s : Serial execution time
  - T_p : Parallel execution time
- It can be expressed as well vs. the parallel fraction $P = 1 - \alpha$

Amdahl Law – Fixed-size Model (1967)



How real is this?

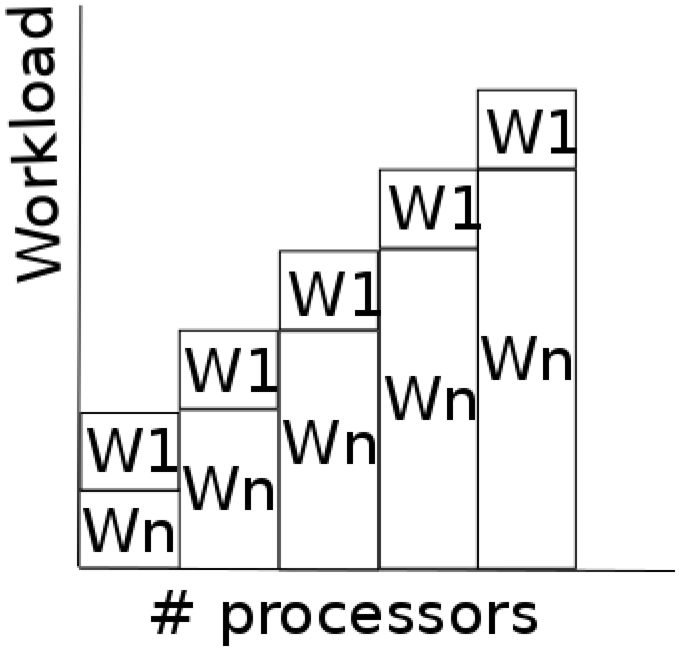
$$\lim_{p\to\infty} S_{Amdahl} = \lim_{p\to\infty} \frac{1}{\alpha + \frac{(1-\alpha)}{p}} = \frac{1}{\alpha}$$

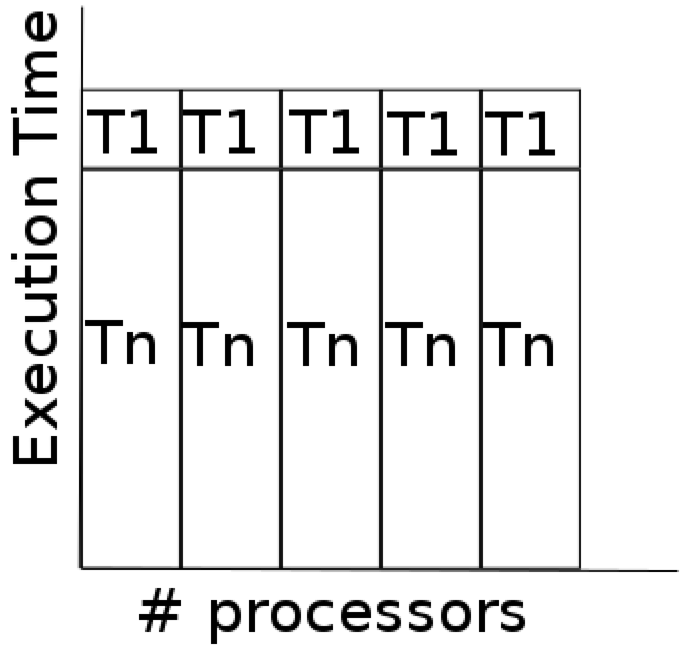
- If the sequential fraction is 20%, we have:

$$\lim_{p\to\infty} S_{Amdahl} = \frac{1}{0.2} = 5$$

- Speedup 5 using infinite processors!

Fixed-time model





Gustafson Law—Fixed-time Model (1989)

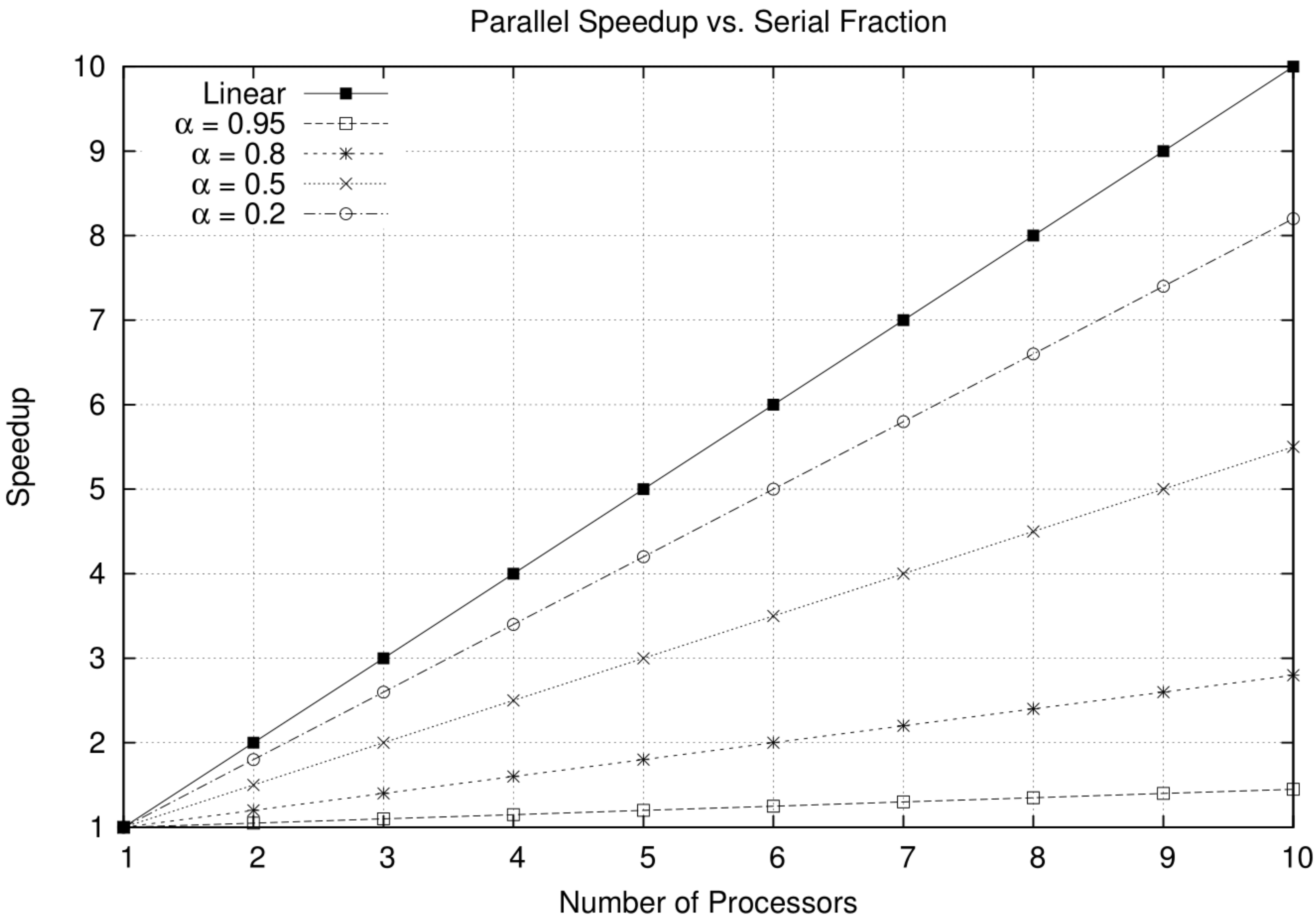
- The execution time is fixed: it studies how the behavior of the *scaled* program varies when adding more computing power

$$W' = \alpha W + (1-\alpha)pW$$

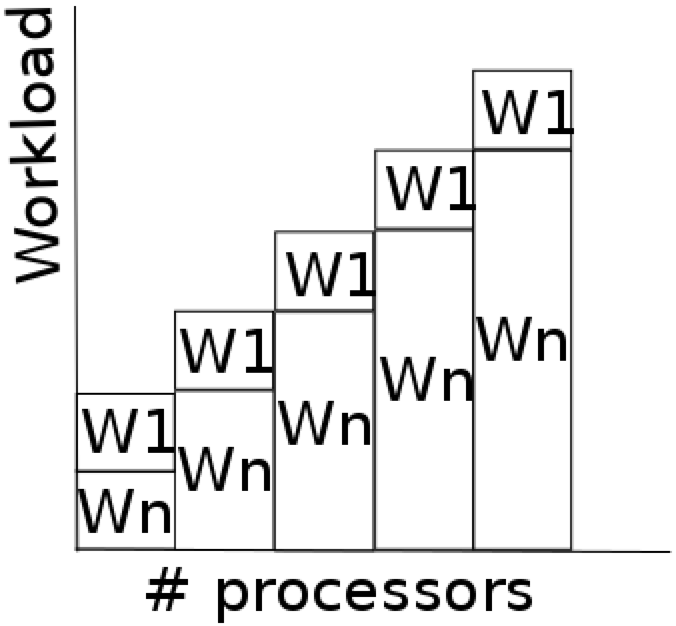
$$S_{Gustafson} = \frac{W'}{W} = \alpha + (1-\alpha)p$$

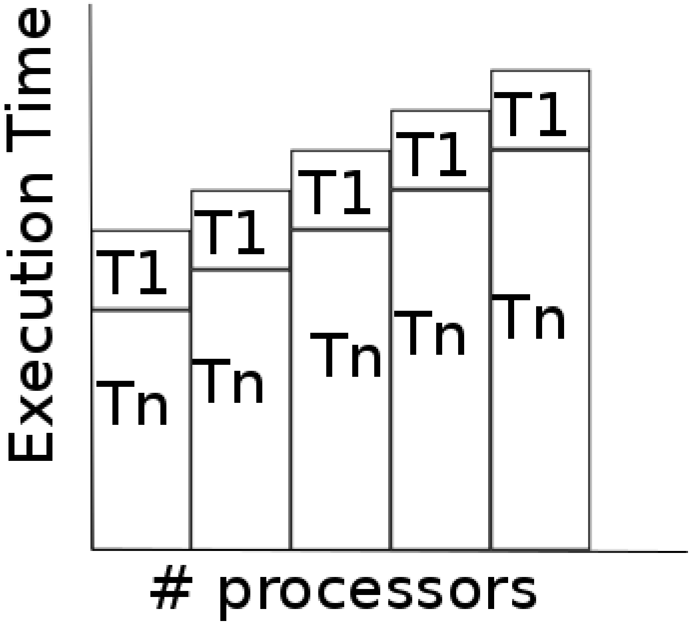
- where:
  - $\alpha \in [0,1]$: Serial fraction of the program
  - $p \in N$: Number of processors
  - W : Original workload
  - W' : Scaled workload

Speed-up according to Gustafson



Memory-bounded model





Sun Ni Law—Memory-bounded Model (1993)

- The workload is scaled, bounded by memory

$$S_{Sun-Ni} = \frac{sequential\ time\ for\ W^*}{parallel\ time\ for\ W^*}$$

$$S_{Sun-Ni} = \frac{\alpha W + (1-\alpha)G(p)W}{\alpha W + (1-\alpha)G(p)\frac{W}{p}} = \frac{\alpha + (1-\alpha)G(p)}{\alpha + (1-\alpha)\frac{G(p)}{p}}$$

- where:
  - $G(p)$ describes the workload increase as the memory capacity increases
  - $W^* = \alpha W + (1-\alpha)G(p)W$

Speed-up according to Sun Ni

$$S_{Sun-Ni} = \frac{\alpha + (1-\alpha)G(p)}{\alpha + (1-\alpha)\frac{G(p)}{p}}$$

- If $G(p) = 1$

$$S_{Amdahl} = \frac{1}{\alpha + \frac{(1-\alpha)}{p}}$$

- If $G(p) = p$

$$S_{Gustafson} = \alpha + (1-\alpha)p$$

- In general, $G(p) > p$ gives a higher scale-up

Superlinear speedup

- Can we have a Speed-up > p ? Yes!
  - Workload increases more than computing power (G(p) > p)
  - Cache effect: larger accumulated cache size. More or even all of the working set can fit into caches and the memory access time reduces dramatically
  - RAM effect: enables the dataset to move from disk into RAM drastically reducing the time required, e.g., to search it.

Scalability

- Efficiency

$$E = \frac{speedup}{\#processors}$$

- **Strong Scalability**: If the efficiency is kept fixed while increasing the number of processes and maintain fixed the problem size

- **Weak Scalability**: If the efficiency is kept fixed while increasing at the same rate the problem size and the number of processes

Scalability

Correctness conditions

Progress conditions

Correctness in a sequential world

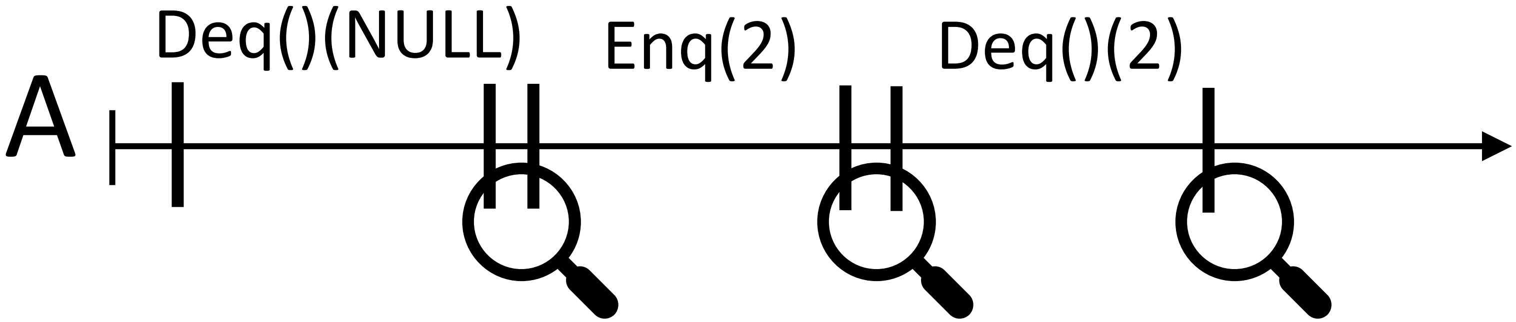
- What does it mean for a program to be correct?
- **Simplification:** We mask any program/algorithm behind the concept of ABSTRACT DATA TYPE
- An Abstract Data Type (ADT) defines:
  - A state
  - The domain of its values
  - Operations/methods
  - Constraints to apply operations/methods
- An ADT specification do not care about implementations
- Typically, operations and their constraints are defined via pre-conditions and post-conditions

Example: FIFO Queue ADT

- init():
  - pre: state = NULL
  - post: state = []
- enqueue(x):
  - pre: state != NULL
  - post: state = state,x
- dequeue()(x):
  - pre: state = x,seq
  - post: state = seq
- dequeue()(NULL):
  - pre: state = []
  - post: -

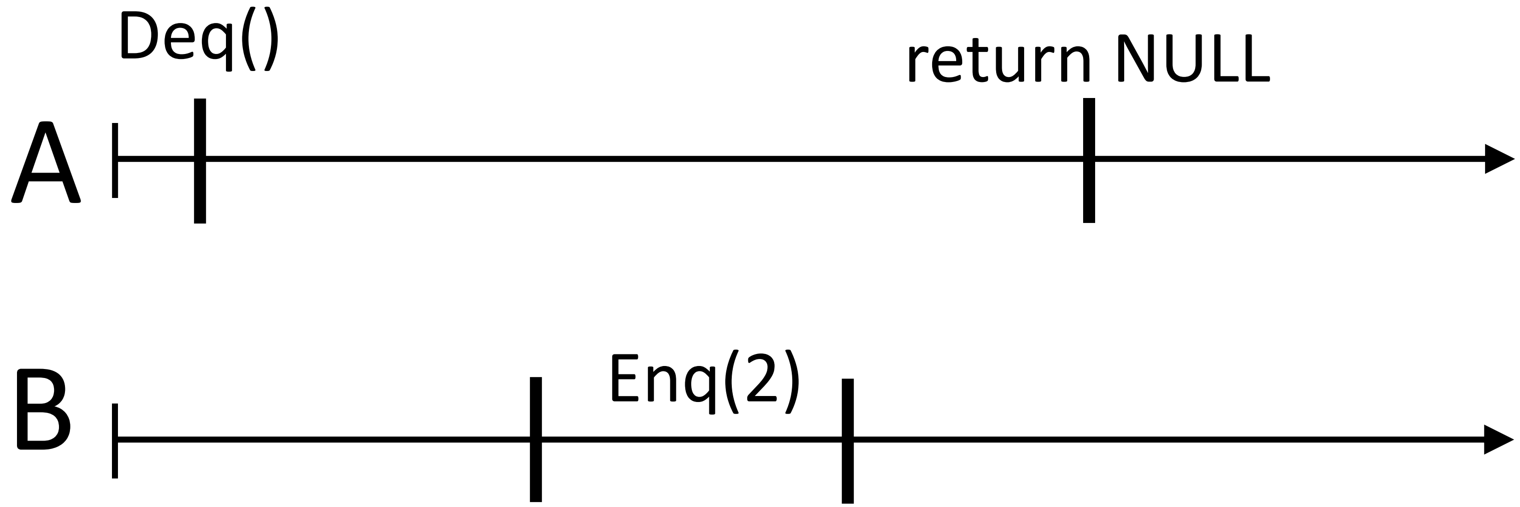
Correctness in a sequential world

- When considering only sequential:
  - Methods do not overlap each other
  - The effect/result of a method can be checked by inspecting the state before/after their ending
- We totally ignore the fact that methods take time!
- We totally ignore the state during method invocations!
- Proving that a sequential implementation is correct:
  - Ensure that for all possible (sequential) executions both pre and post conditions always hold
- Focus on the correctness of an individual execution



Correctness in a concurrent world?

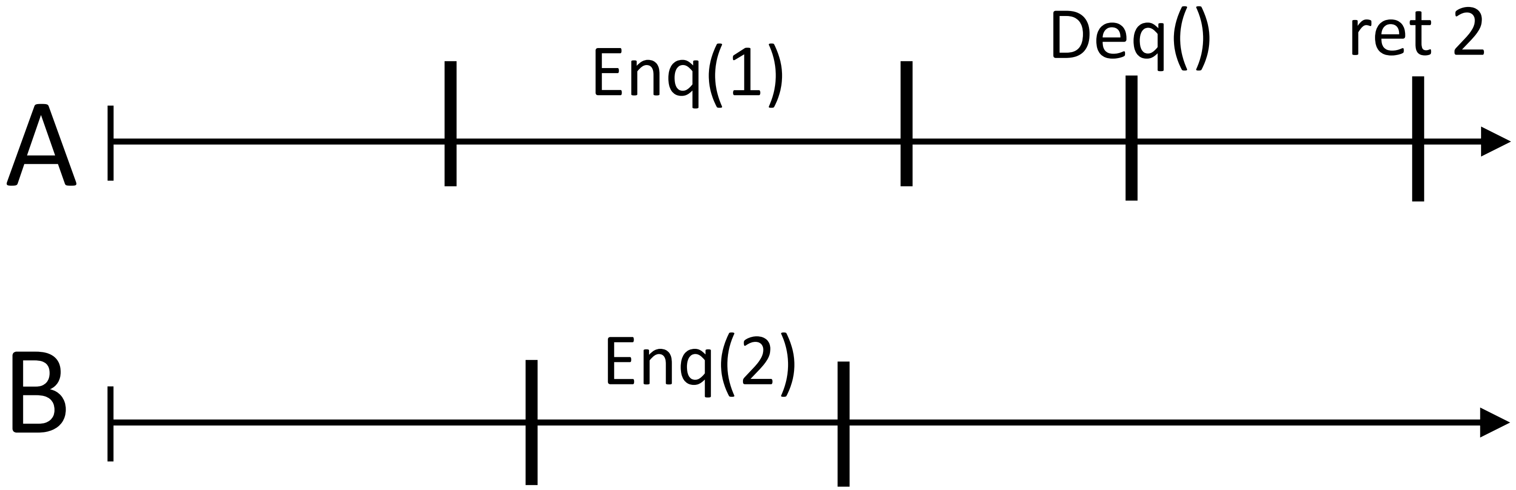
- Threads invoke methods
- Threads can experience arbitrary large delays



- Is it correct?

Correctness in a concurrent world?

- Threads invoke methods
- Threads can experience arbitrary large delays



- Is it correct?

Correctness in a concurrent world?

- Threads invoke methods
- Threads can experience arbitrary large delays
- Methods are partially ordered intervals
- Methods could never be executed in isolation!
- We should describe any possible interleaving!
- What does it mean for a concurrent program to be correct?
  - What’s exactly a concurrent FIFO queue?
  - FIFO implies a strict temporal ordering
  - Concurrency implies an ambiguous temporal ordering

Classical approach to concurrent programming

Based on blocking primitives

- Semaphores
- Locks acquiring
- Simple??

PRODUCER

```
Semaphore p, c = 0;
Buffer b;

while(1) {
wait(c);
<Write on b>
signal(p);
}
```

☹

CONSUMER

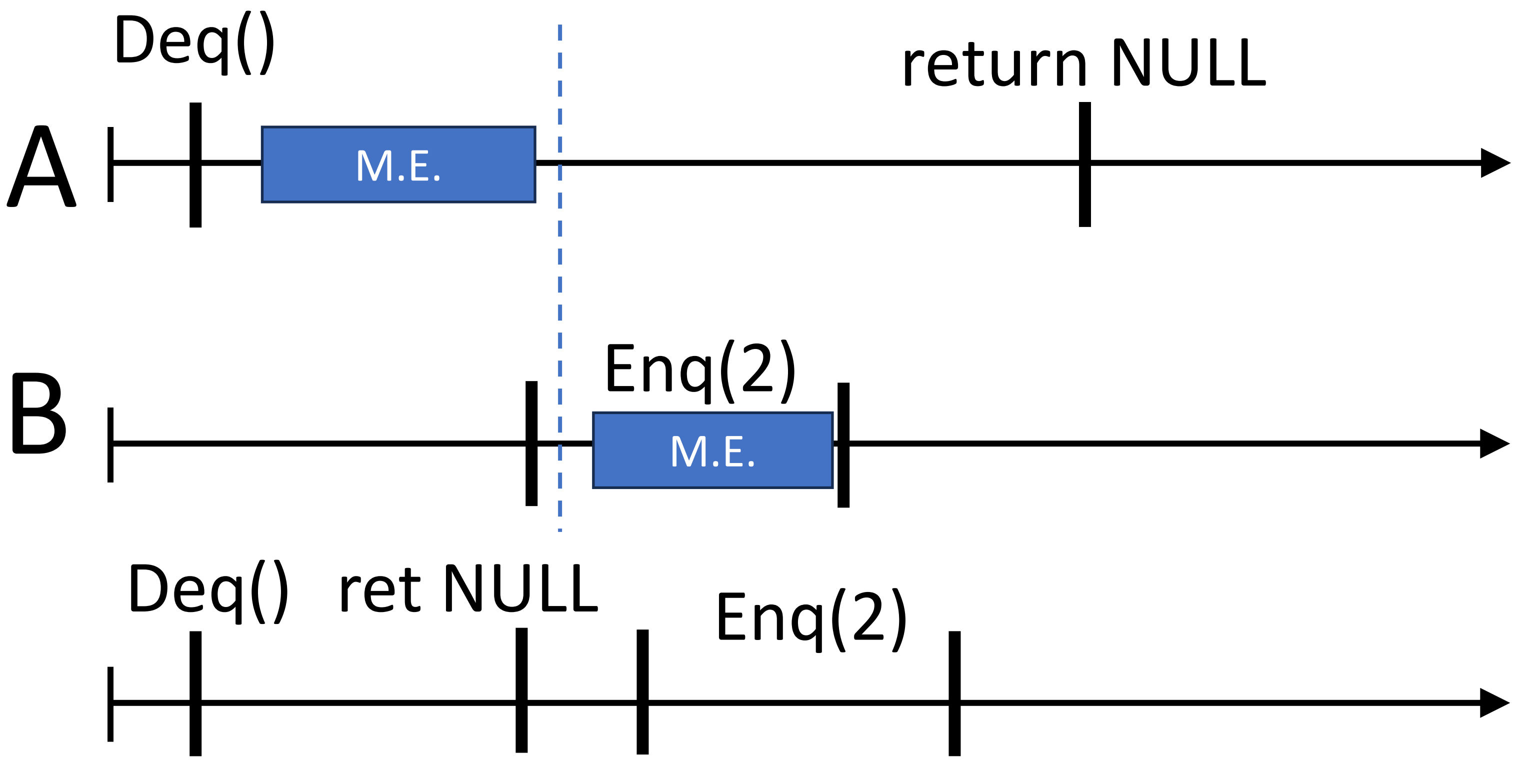
```
Semaphore p, c = 0;
Buffer b;

while(1) {
wait(p);
<Read from b>
signal(c);
}
```

☹

Correctness in a concurrent world?

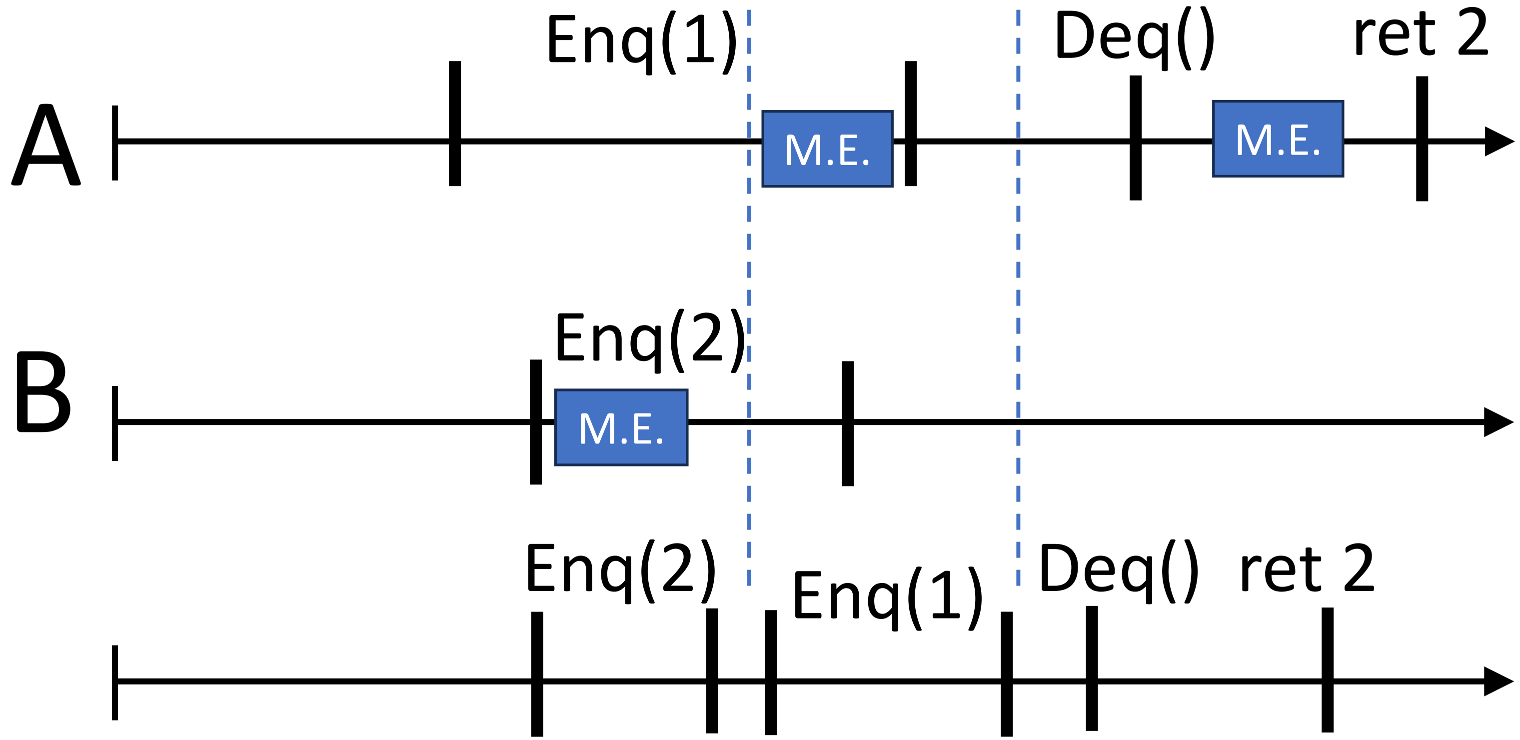
- Threads invoke methods
- Threads can experience arbitrary large delays



- Is it correct? Yes!

Correctness in a concurrent world?

- Threads invoke methods
- Threads can experience arbitrary large delays



- Is it correct? Yes!

Correctness

- Intuitively, if we rely on locks, changes happen in a non-interleaved fashion, resembling a sequential execution
- We can say a concurrent execution is correct only because we can associate it with a sequential one, which we know the functioning of
- An execution is correct if it is equivalent to a correct sequential execution

Correctness

- An **execution** is correct if it is equivalent to a correct **sequential execution**

A simplified model of a concurrent system

- A concurrent system is a collection of sequential threads/processes that communicate through shared data structures called objects.
- An object has a unique name and a set of primitive operations.

A simplified model of a concurrent execution

- A *history* is a sequence of <u>invocations</u> and <u>replies</u> generated on an <u>object</u> by a set of threads
- Invocation:

method name

thread id $\boxed{\texttt{A}}$ $\boxed{\texttt{op}}$($\boxed{\texttt{args*}}$) $\boxed{\texttt{x}}$ object instance

list of parameters

- Reply:

reply token

$\texttt{A}$ $\boxed{\texttt{ret}}$($\boxed{\texttt{res*}}$) $\texttt{x}$

list of returned values

A simplified model of a concurrent execution

- A *sequential history* is a history where all the invocations have an immediate response
- A *concurrent history* is a history that is not sequential

Sequential

```
H': A op() x
    A ret() x
    B op() x
    B ret() x
    A op() y
    A ret() y
```

Concurrent

```
H:  A op() x
    B op() x
    A ret() x
    A op() y
    B ret() x
    A ret() y
```

Correctness

- An **execution** is correct if it is equivalent to a correct **sequential execution**

⇒ A **history** is correct if it is **equivalent** to a correct **sequential history**

A simplified model of a concurrent execution

- A *process subhistory H|P* of a history *H* is the subsequence of all events in *H* whose process names are *P*

```
H:  A op()  x
    B op()  x
    A ret() x
    A op()  y
    B ret() x
    A ret() y
```

```
H|A: A op()  x
     A ret() x
     A op()  y
     A ret() y
```

- Process subhistories are always sequential

Equivalence between histories

- Two histories *H* and *H'* are equivalent if for every process *P*, *H|P=H'|P*

```
H:  A op()  x        H': B op()  x
    B op()  x            B ret() x
    A ret() x            A op()  x
    A op()  y            A ret() x
    B ret() x            A op()  y
    A ret() y            A ret() y
```

```
 H|A:
H'|A: A op()  x
      A ret() x
      A op()  y
      A ret() y

 H|B:
H'|B: B op()  x
      B ret() x
```

Correctness conditions

- A **concurrent execution** is correct if it is equivalent to a correct **sequential execution**

⇒ A **history** is correct if it is **equivalent** to a ~~correct~~ **sequential history** which satisfies a given correctness condition

- A correctness condition specifies the set of histories to be considered as reference

⇒In order to implement correctly a concurrent object wrt a correctness condition, we must guarantee that every possible history on our implementation satisfies the correctness condition

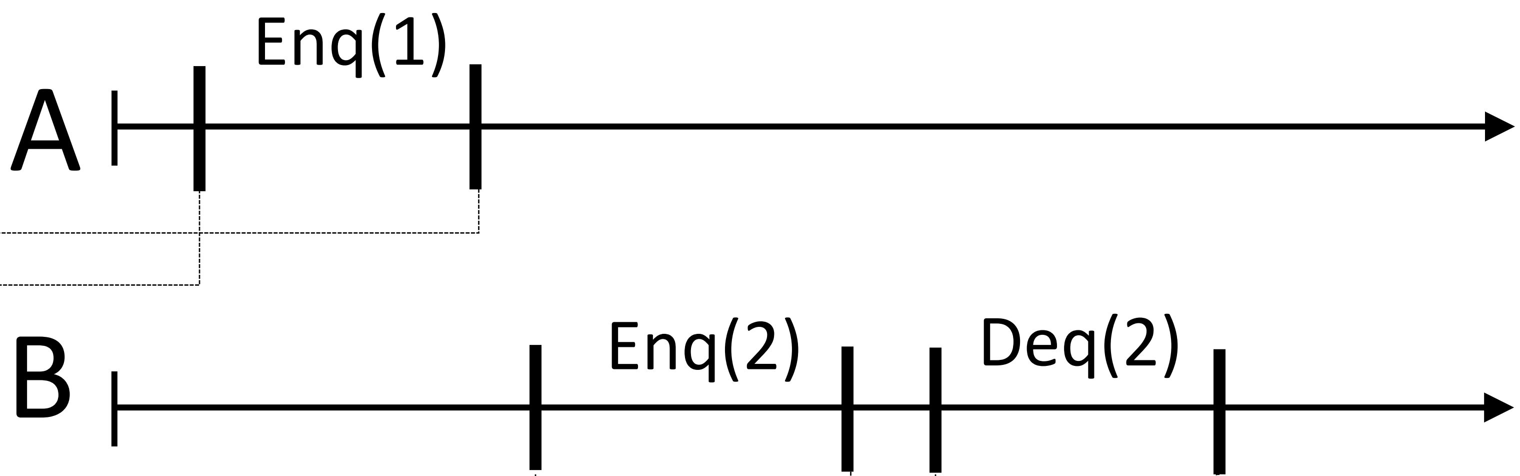
Sequential Consistency [Lamport 1970]

- A history H is sequentially consistent if

1. it is equivalent to a sequential history S
2. S is legal according to the sequential definition of the object

$\Rightarrow$ An object implementation is sequentially consistent if every history associated with its usage is sequentially consistent

Sequential Consistency [Lamport 1970]




```
A Enq(1)   x
A ret()    x
B Enq(2)   x
B ret()    x
B Deq(2)   x
B ret()    x
```

Sequential Consistency [Lamport 1970]

H|A:

- H’ is legal and sequential
- H is equivalent to H’
- H is correct w.r.t sequential consistency

H:

```
A Enq(1)   x
A ret()    x
B Enq(2)   x
B ret()    x
B Deq(2)   x
B ret()    x
```

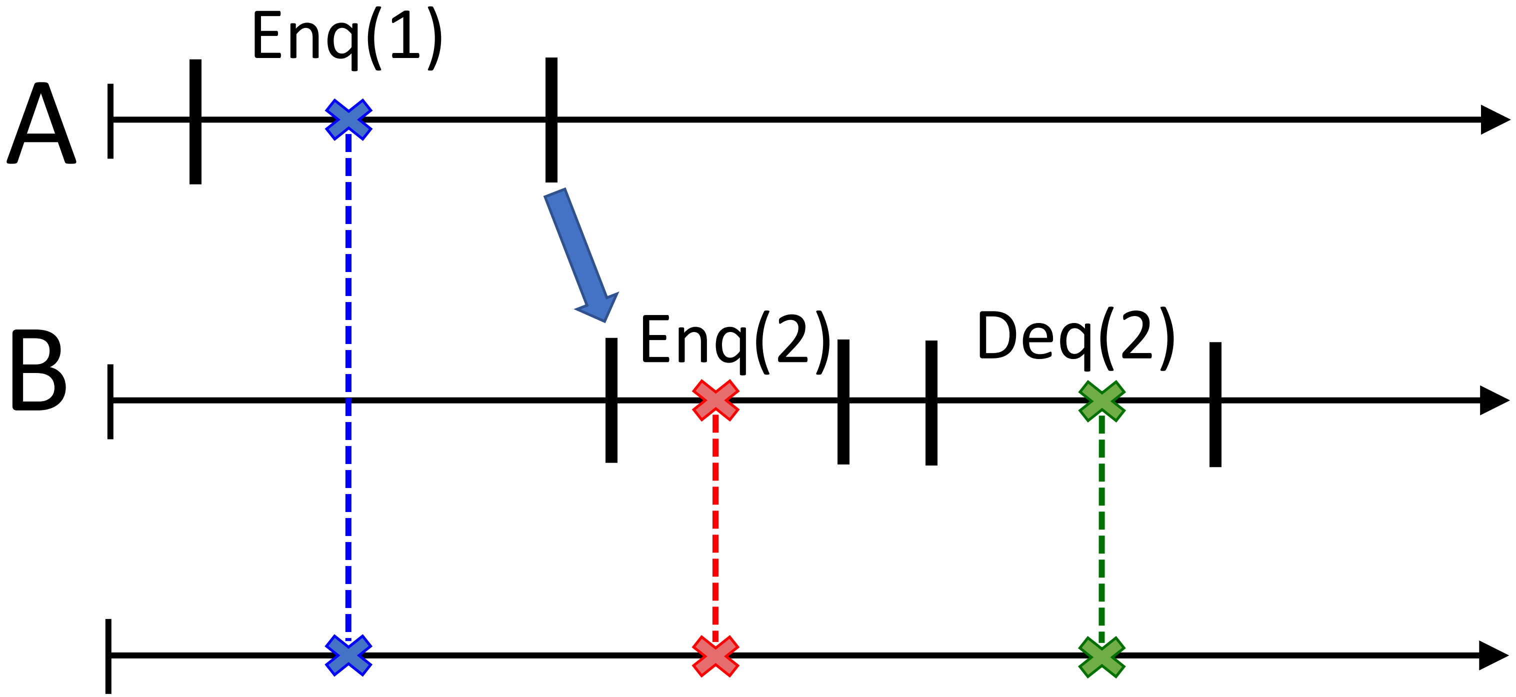
H’:

```
B Enq(2)   x
B ret()    x
A Enq(1)   x
A ret()    x
B Deq(2)   x
B ret()    x
```

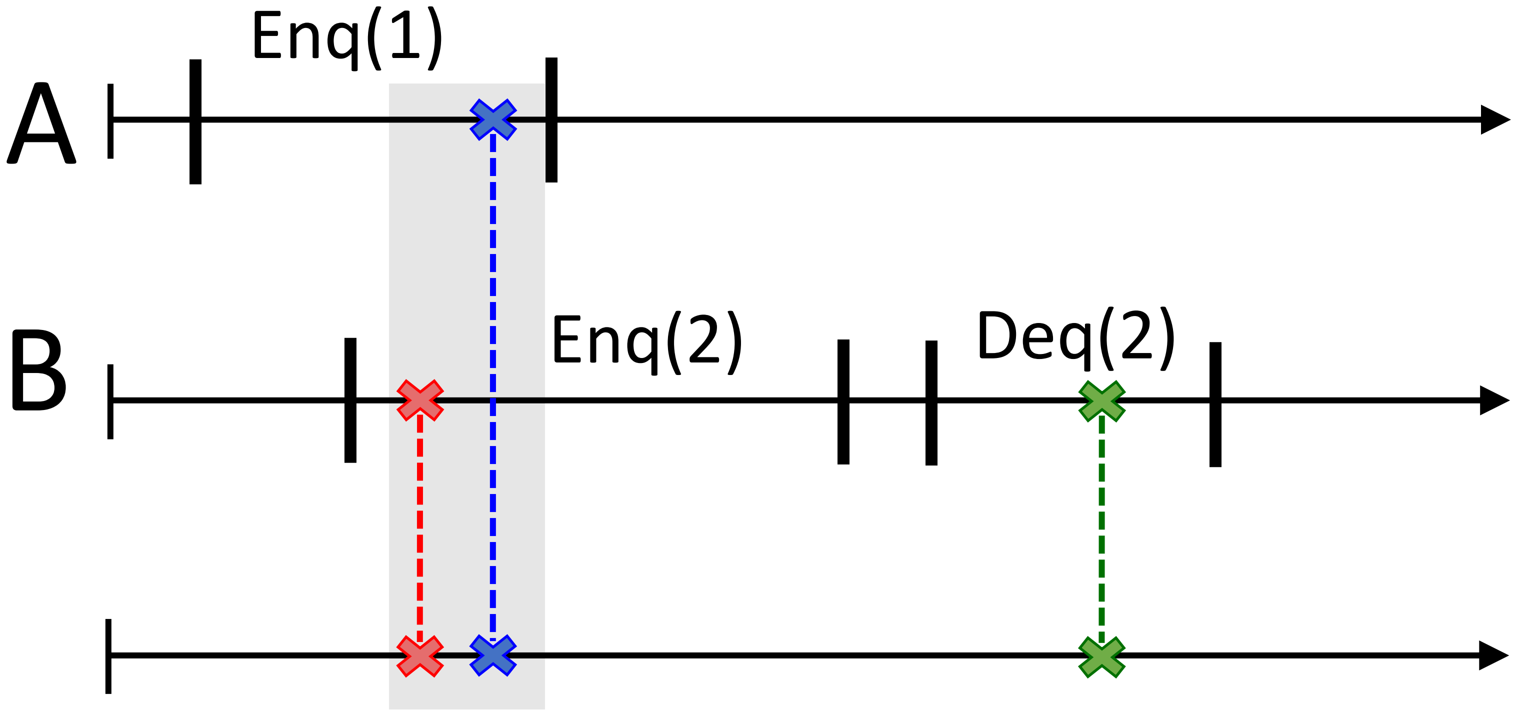
Linearizability [Herlihy 1990]

- A concurrent execution is linearizable if:
  - Each procedure appears to be executed in an indivisible point (*linearization point*) between its invocation and completion
  - The order among those points is correct according to the sequential definition of objects

Linearizability [Herlihy 1990]



Linearizability [Herlihy 1990]



Linearizability [Herlihy 1990]

- A history H is linearizable if:

1. it is equivalent to sequential history S
2. S is correct according to the sequential definition of objects
3. If a response precedes an invocation in the original history, then it must precede it in the sequential one as well

$\Rightarrow$ An object implementation is linearizable if every history associated with its usage can be linearized

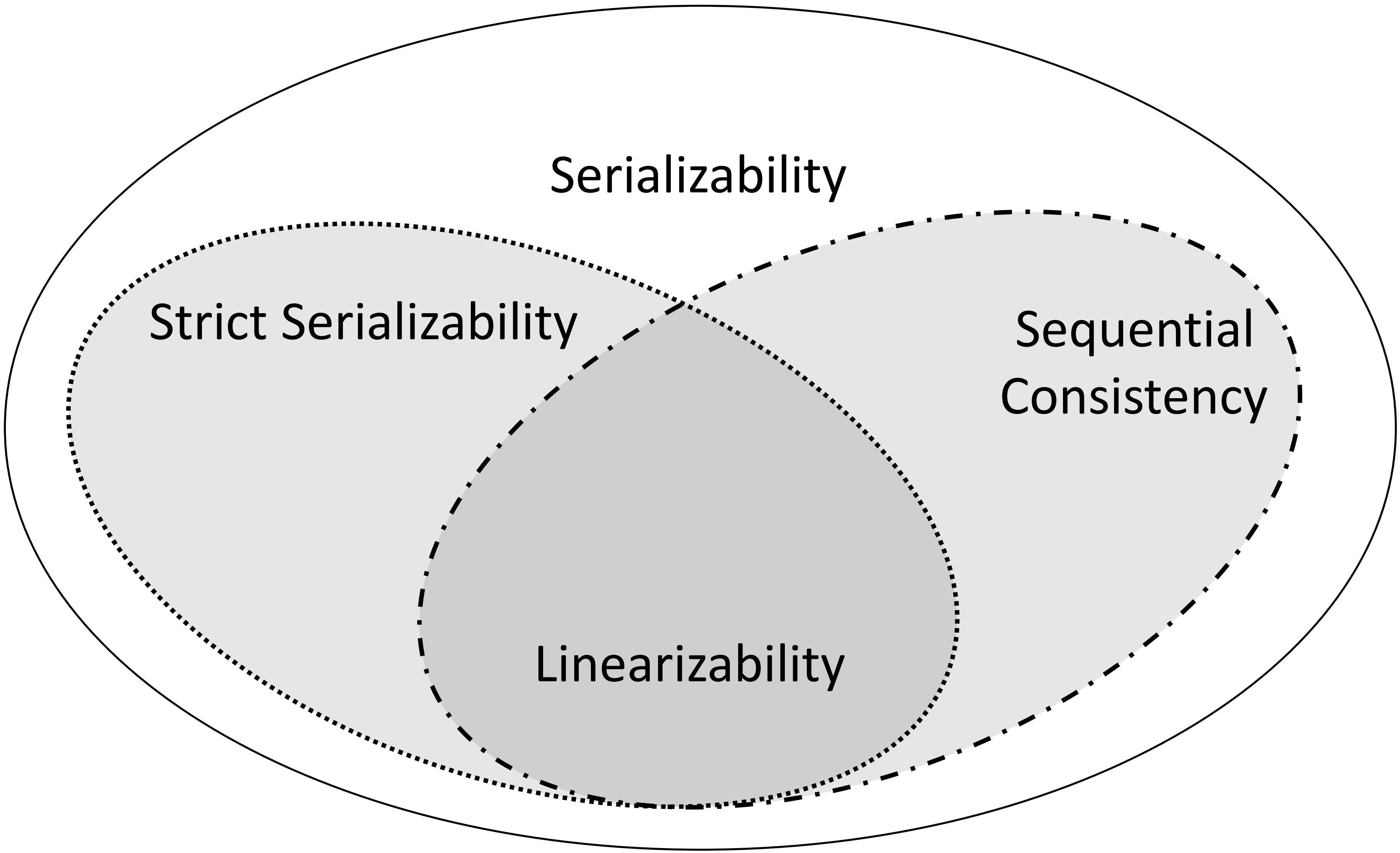
Linearizability [Herlihy 1990]

- Linearizability requires:
  - Sequential Consistency
  - Real-time order
- Linearizability $\Rightarrow$ Sequential Consistency
- The composition of linearizable histories is still linearizable
- Linearizability is a *local* property (closed under composition)

Quick look on transaction correctness conditions

- We can see a transaction as a set of procedures on different object that has to appear as atomic
- Serializability requires that transactions appear to execute sequentially, i.e., without interleaving.
  - A sort of sequential consistency for multi-object atomic procedures
- Strict-Serializability requires the transactions' order in the sequential history is compatible with their precedence order
  - A sort of linearizability for multi-object atomic procedures

A bird's eye view on correctness conditions



Correctness conditions (incomplete) taxonomy

	Sequential Consistency	**Linearizability**	**Serializability**	**Strict Serializability**
Equivalent to a sequential order	✓	✓	✓	✓

Correctness conditions (incomplete) taxonomy

	Sequential Consistency	Linearizability	Serializability	Strict Serializability
Equivalent to a sequential order	✓	✓	✓	✓
Respects program order in each thread	✓	✓	✓	✓
Consistent with real-time ordering	✗	✓	✗	✓
Access multiple objects atomically	✗	✗	✓	✓
Locality	✗	✓	✗	✗

Scalability

Correctness conditions

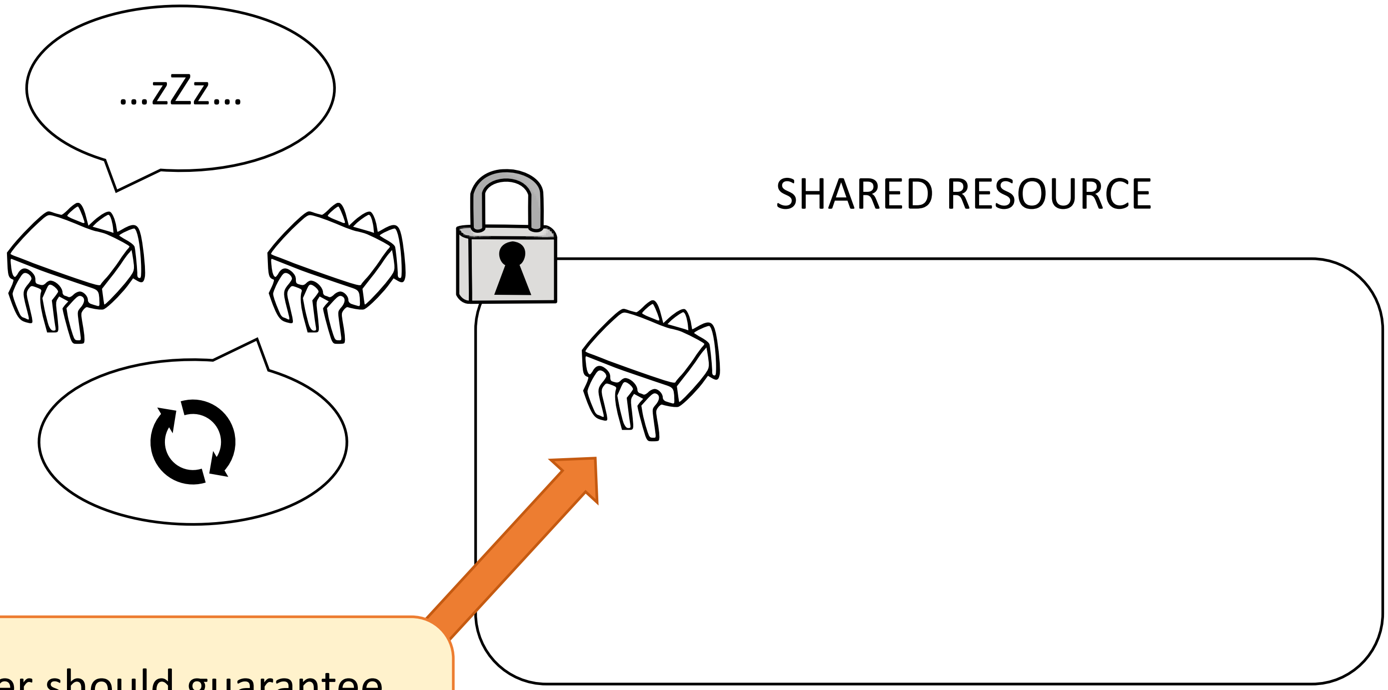
Progress conditions

Progress conditions

- **Deadlock-freedom**:
  - *Some thread acquires a lock eventually*

- **Starvation-freedom**:
  - *Every thread acquires a lock eventually*

Blocking synchronization



The scheduler should guarantee that the thread holding the lock completes its critical section

Scheduler's role

Progress conditions on multiprocessors

- Are not only about guarantees provided by a method implementation
- Are also about the scheduling support needed to provide progress

Requirement for lock-based applications

- Fair histories

Every thread takes an infinite number of concrete steps

Progress conditions

- **Deadlock-freedom**:
  - ~~*Some thread acquires a lock eventually*~~
  - *Some method call completes in every fair execution*
- **Starvation-freedom**:
  - ~~*Every thread acquires a lock eventually*~~
  - *Every method call completes in every fair execution*
- **Lock-freedom**:
  - *Some method call completes in every execution*
- **Wait-freedom**:
  - *Every method call completes in every execution*
- **Obstruction-freedom**:
  - *Every method call, which executes in isolation, completes*

Progress taxonomy

	Non-blocking		Blocking
For everyone	Wait freedom	Obstruction freedom	Starvation freedom
For someone	Lock freedom		Deadlock freedom

Progress taxonomy

	Non-blocking		Blocking
For everyone	-	Thread executes in isolation	Fairness
For someone	-		Fairness

Progress taxonomy

	Independent	Dependent	
	Non-blocking		Blocking
For everyone	Wait freedom	Obstruction freedom	Starvation freedom
For someone	Lock freedom		Deadlock freedom

- The Einsteinium of progress conditions: it does not exist in nature and (maybe) has no “commercial” value
- Clash freedom is a strictly weaker property than obstruction freedom