

Programmazione concorrente

Laurea Magistrale in Ingegneria Informatica

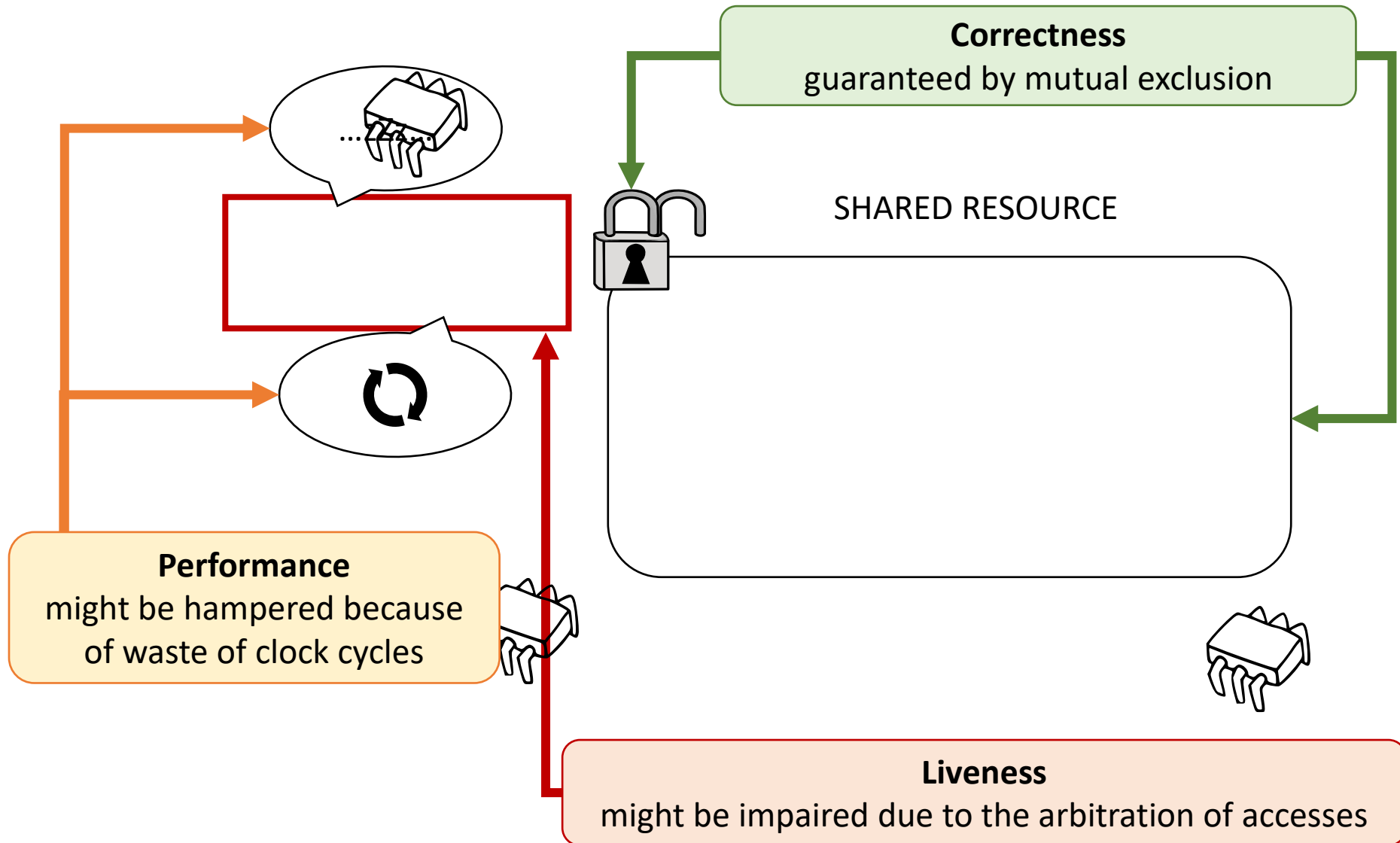
Università Tor Vergata

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

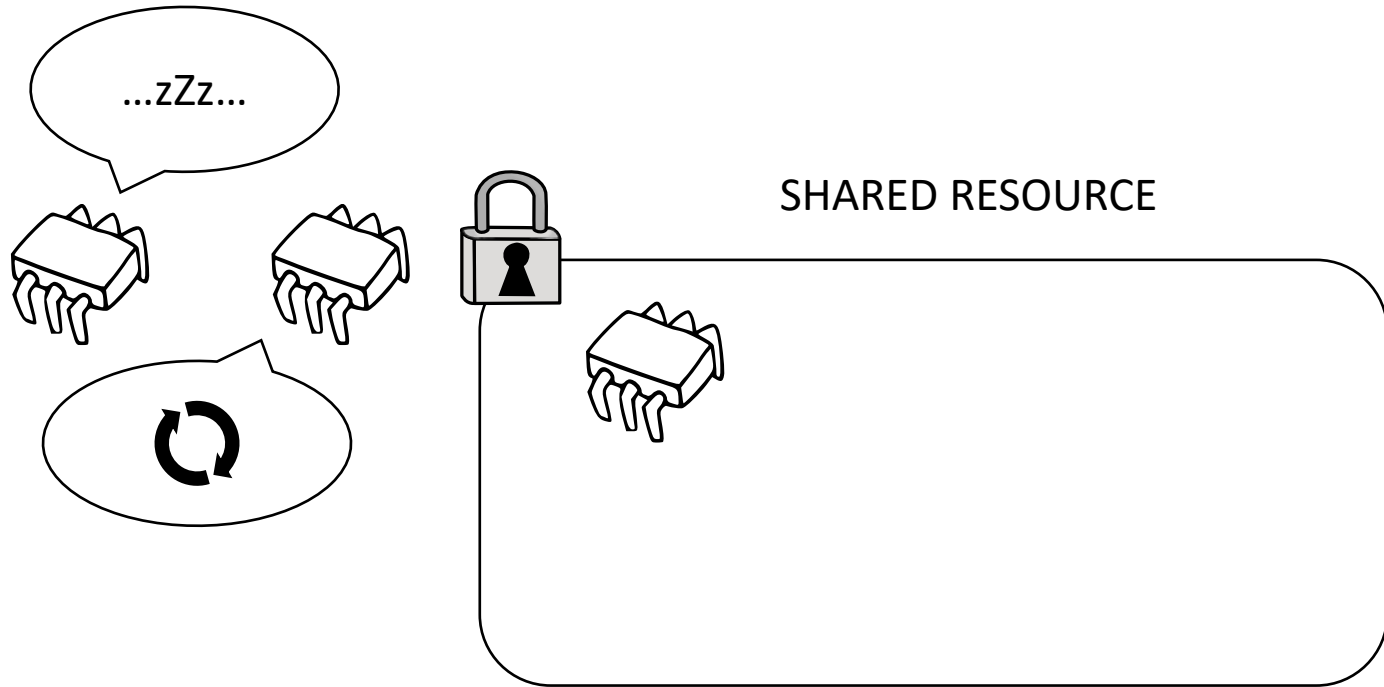
- Threads (aka processes)
- Cores (aka cpus)
- Shared memory
- Arbitrary long asynchronous delays
- Scheduler
 - A system component that decides which/when a thread runs on a given core

Scalability

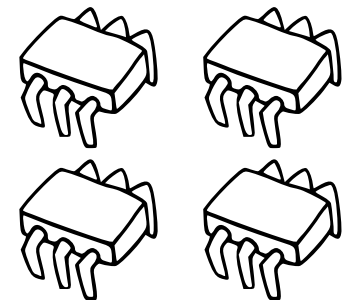
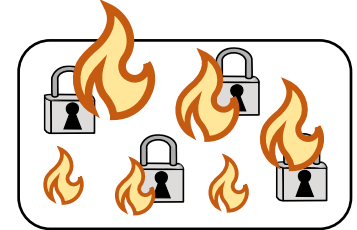
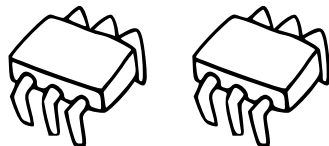
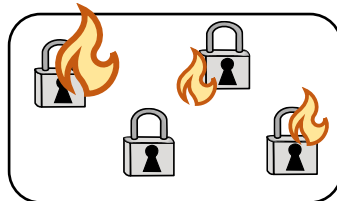
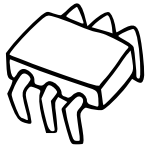
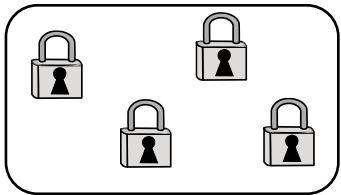
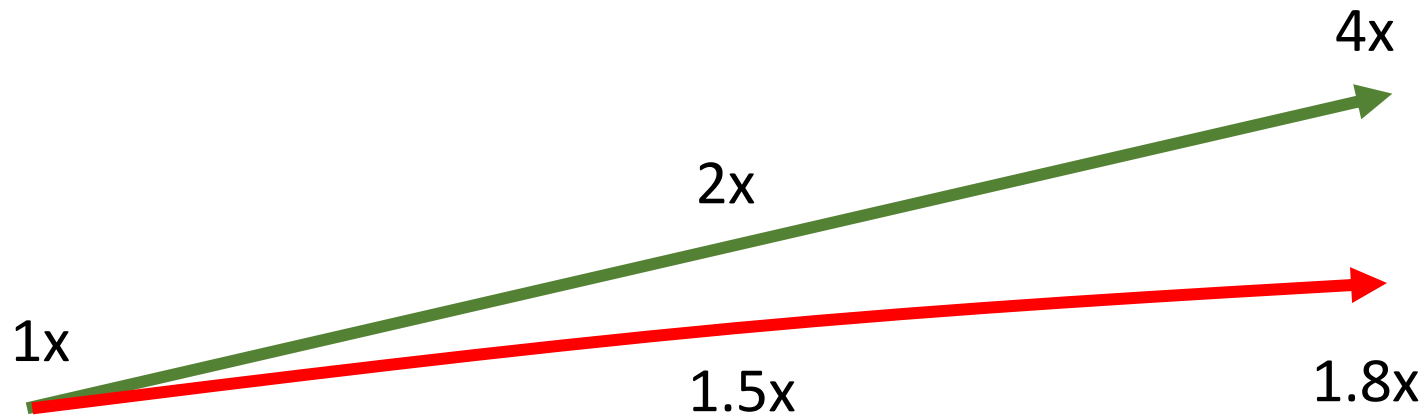
Correctness conditions

Progress conditions

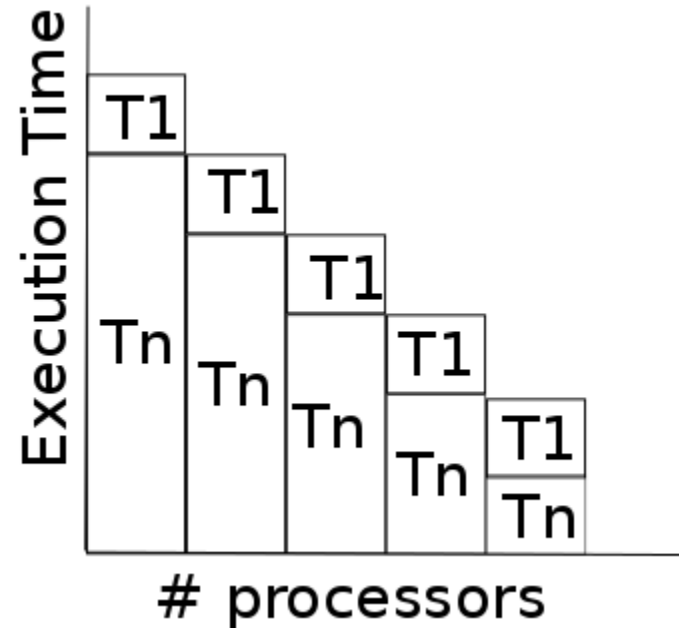
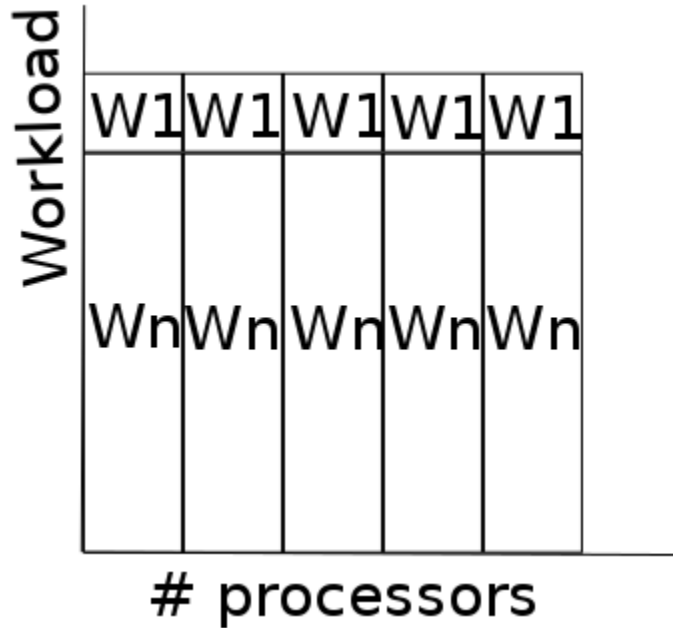
The cost of synchronization



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

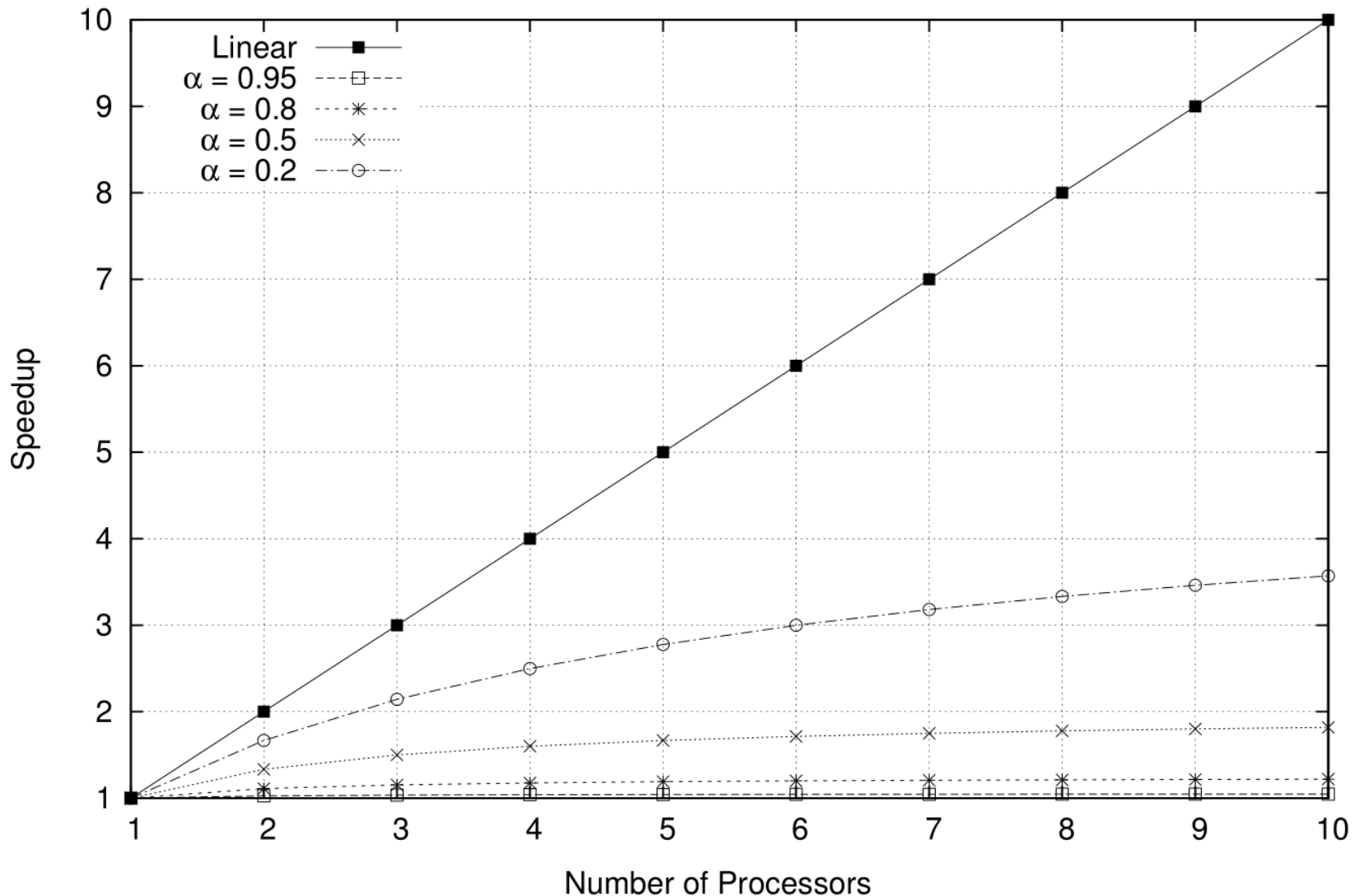
$$S_{Amdahl} = \frac{T_s}{T_p} = \frac{T_s}{\alpha T_s + (1 - \alpha) \frac{T_s}{p}} = \frac{1}{\alpha + \frac{(1 - \alpha)}{p}}$$

- 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)

Parallel Speedup vs. Serial Fraction



How real is this?

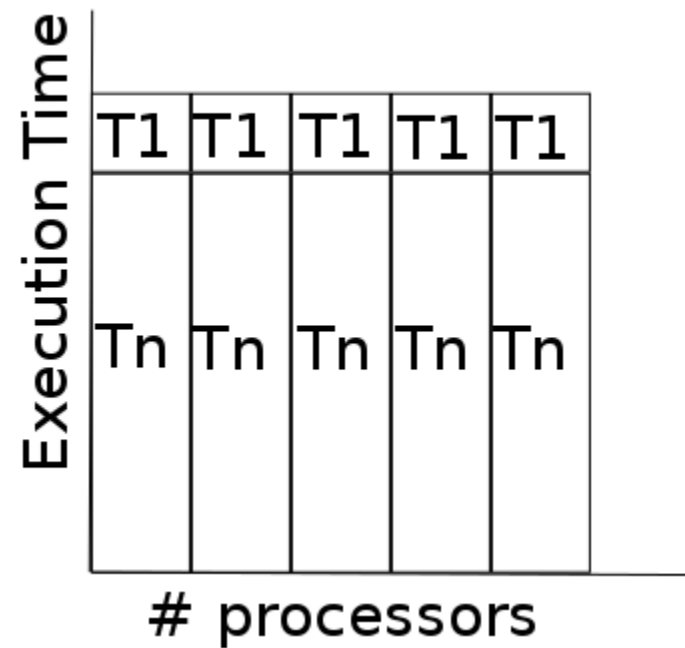
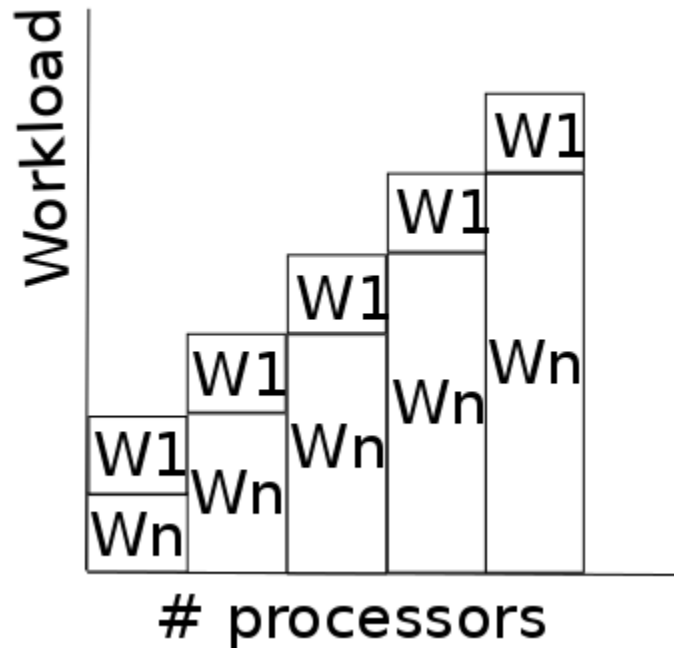
$$\lim_{p \rightarrow \infty} S_{Amdahl} = \lim_{p \rightarrow \infty} \frac{1}{\alpha + \frac{(1 - \alpha)}{p}} = \frac{1}{\alpha}$$

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

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

- 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

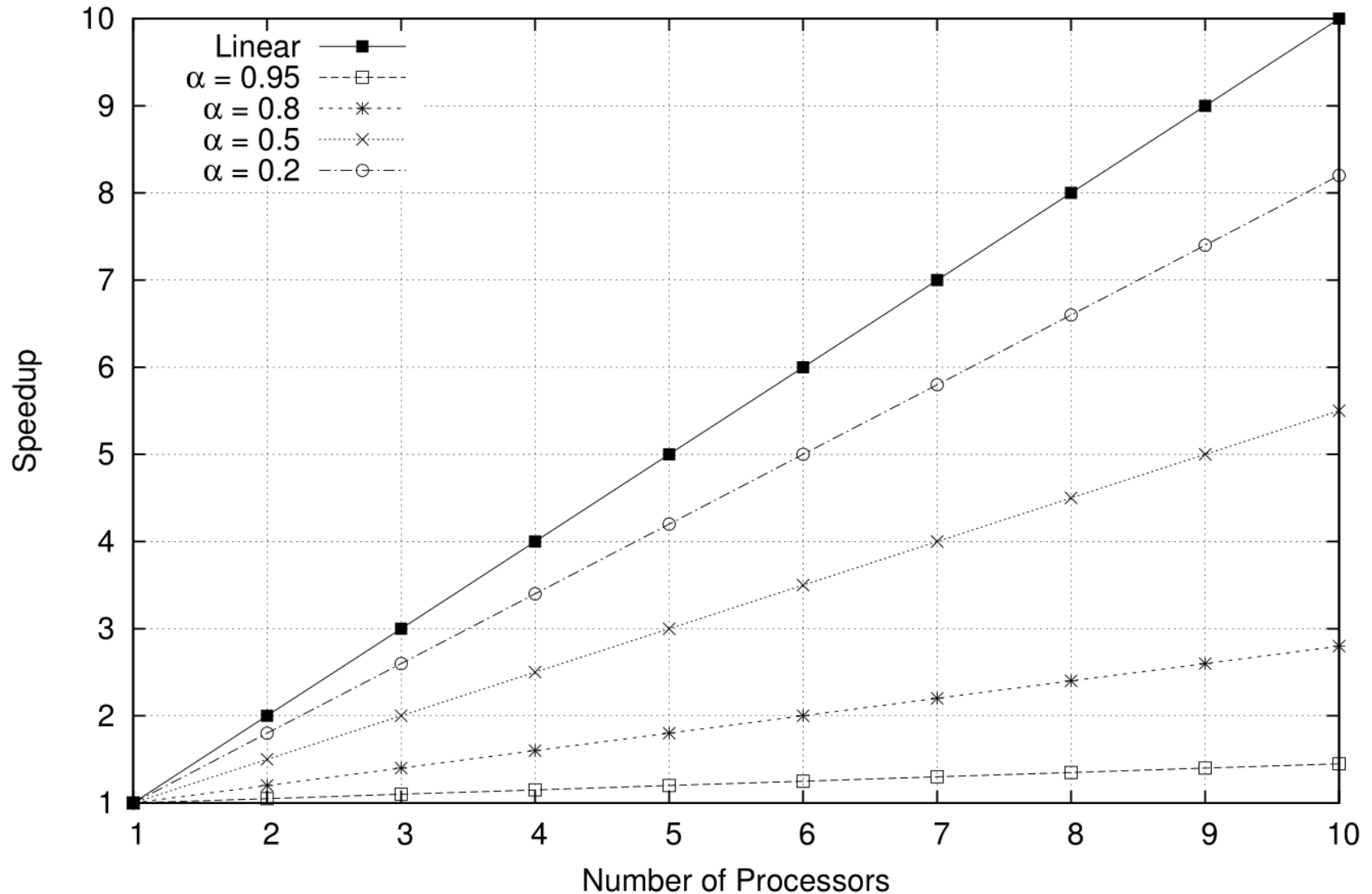
$$W' = \alpha W + (1 - \alpha)pW$$

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

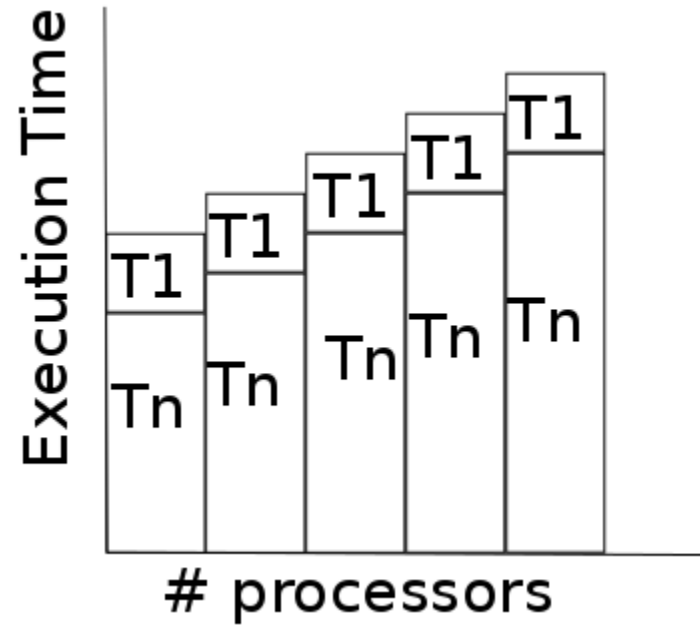
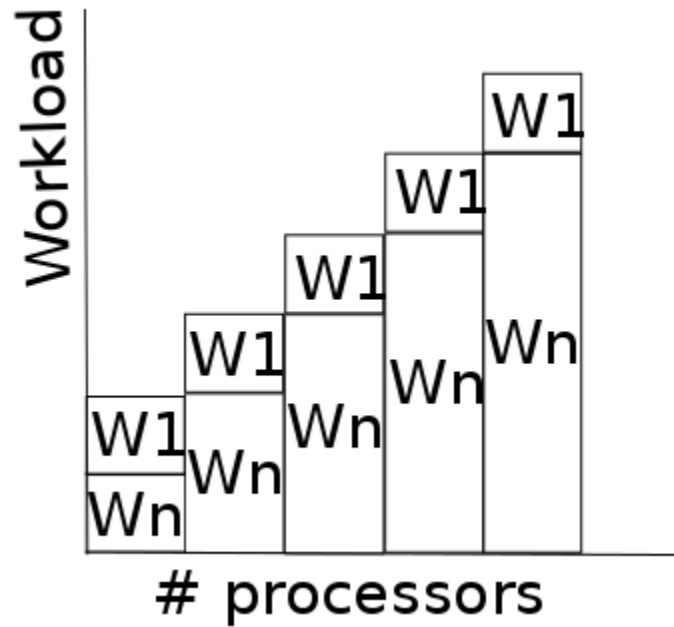
- where:
 - $\alpha \in [0,1]$: Serial fraction of the program
 - $p \in N$: Number of processors
 - W : Original workload
 - W' : Scaled workload

Speed-up according to Gustafson

Parallel Speedup vs. Serial Fraction



Memory-bounded model



Sun Ni Law—Memory-bounded Model (1993)

- The workload is scaled, bounded by memory

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

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

- 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

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

- **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

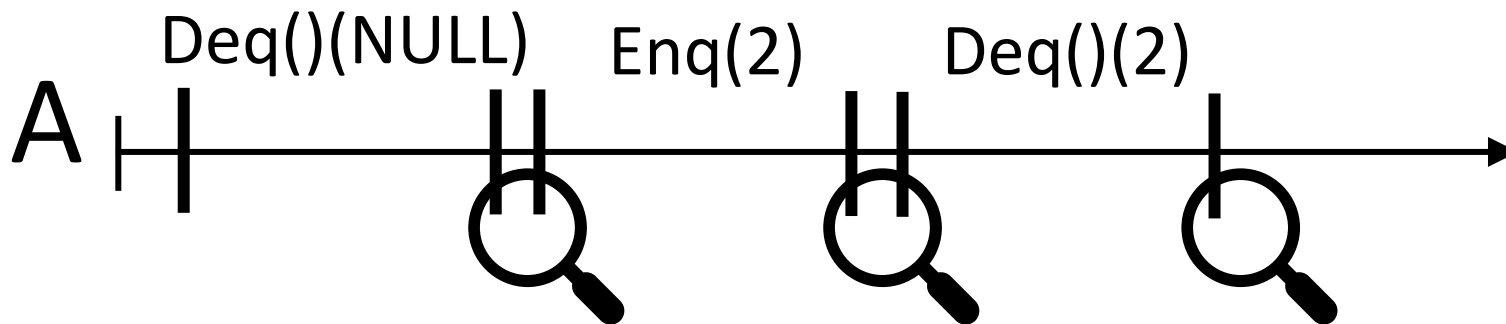
- What does it mean for a program to be correct?
- **Simplification:** We mask any program/algorithm behind the concept of ABSTRACT DATA TYPE
- An Abstract Data Type (ADT) defines:
 - A state
 - The domain of its values
 - Operations/methods
 - Constraints to apply operations/methods
- An ADT specification 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: -

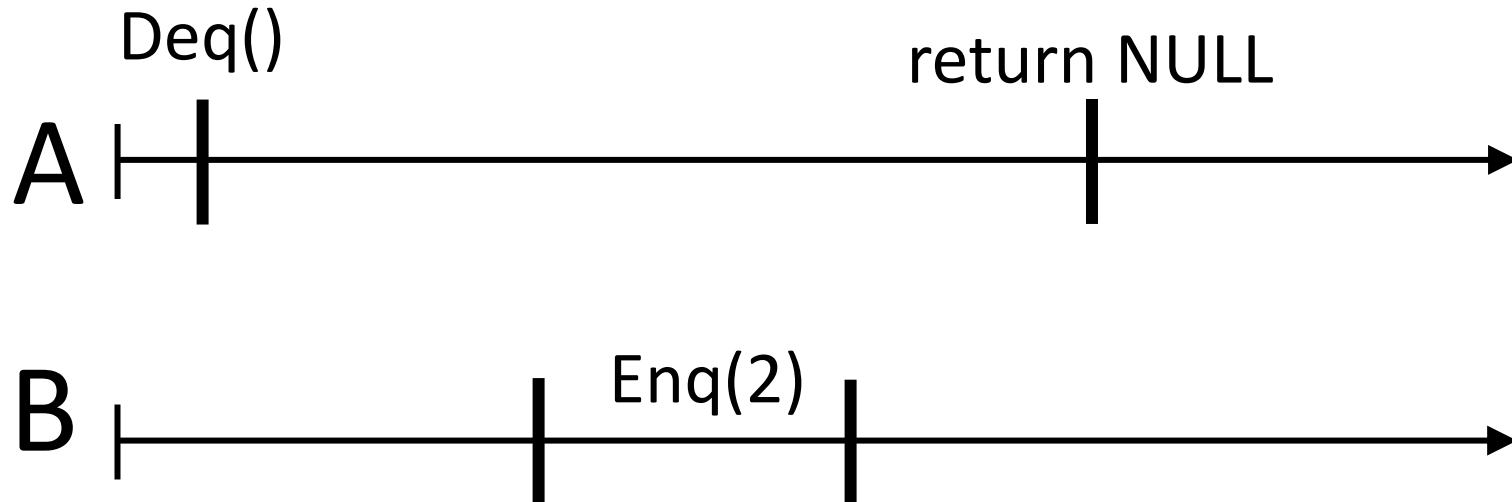
Correctness in a sequential world

- 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?

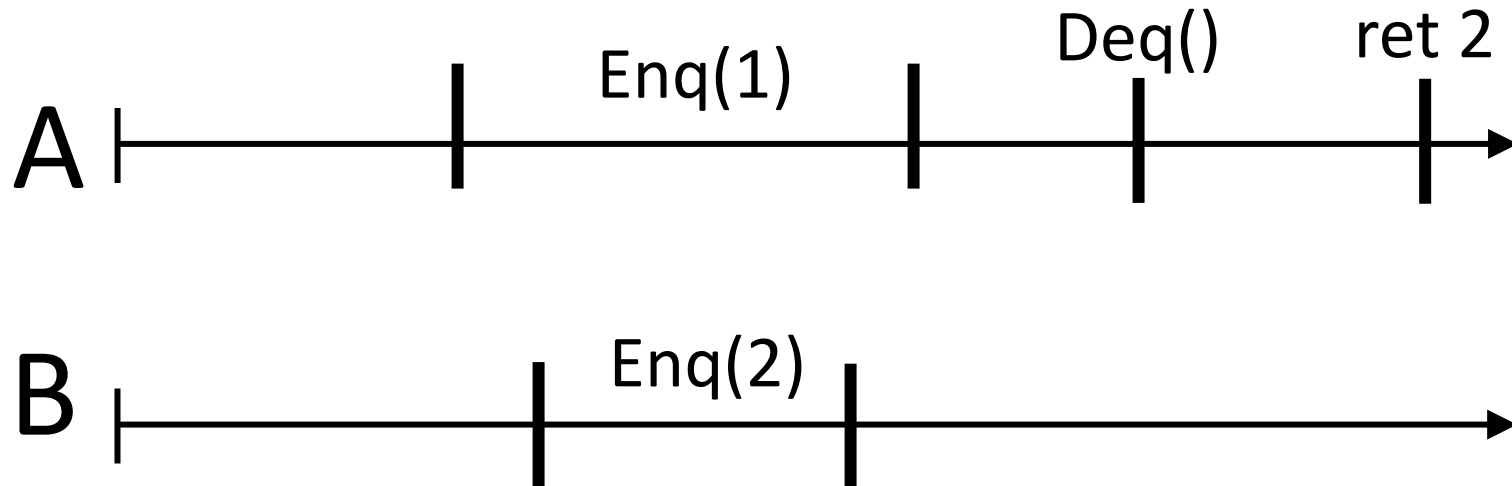
- Threads invoke methods
- Threads can experience arbitrary large delays



- 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

```
1. Semaphore p, c = 0;  
2. Buffer b;  
3.  
4. while(1) {  
5. wait(c);  
6. <Write on b>  
7. signal(p);  
8. }
```



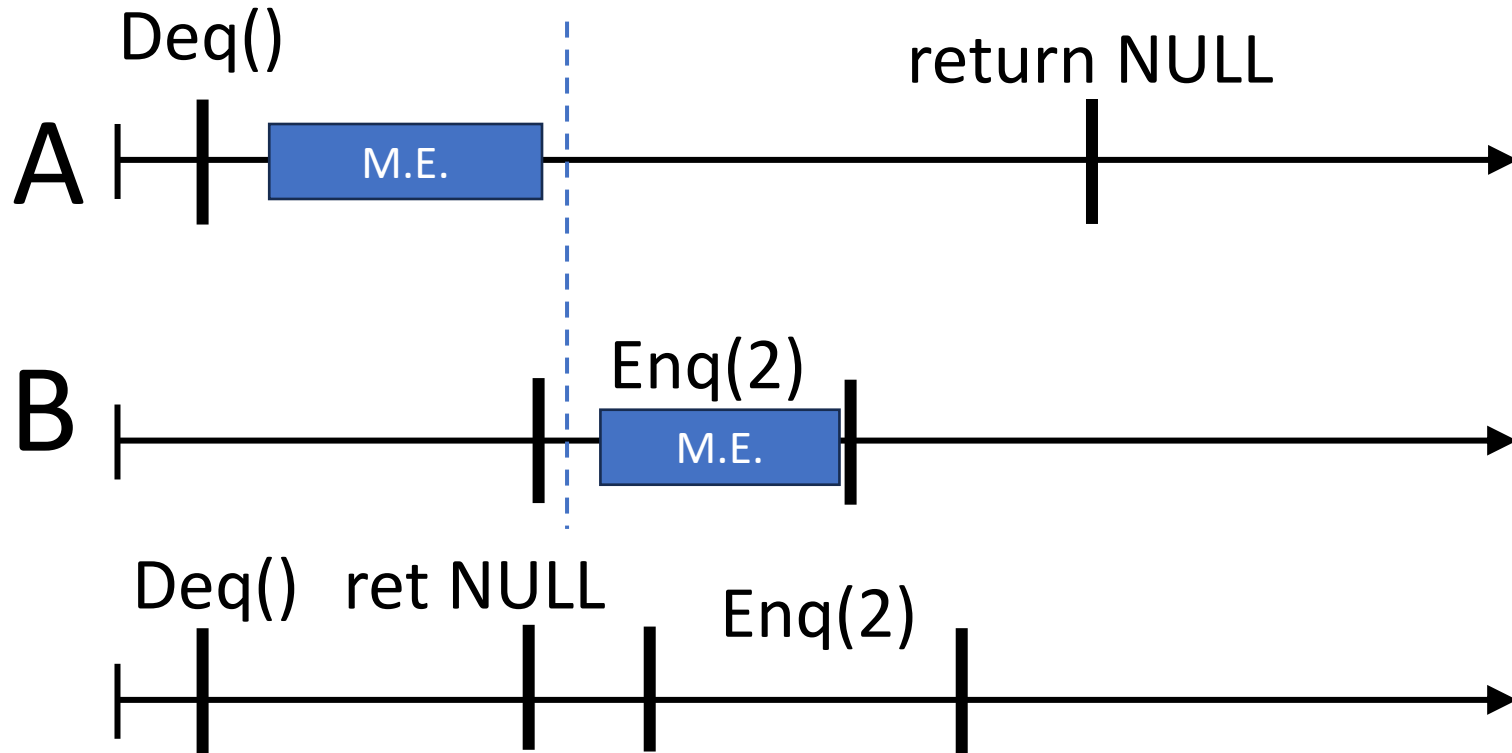
CONSUMER

```
1. Semaphore p, c = 0;  
2. Buffer b;  
3.  
4. while(1) {  
5. wait(p);  
6. <Read from b>  
7. signal(c);  
8. }
```



Correctness in a concurrent world?

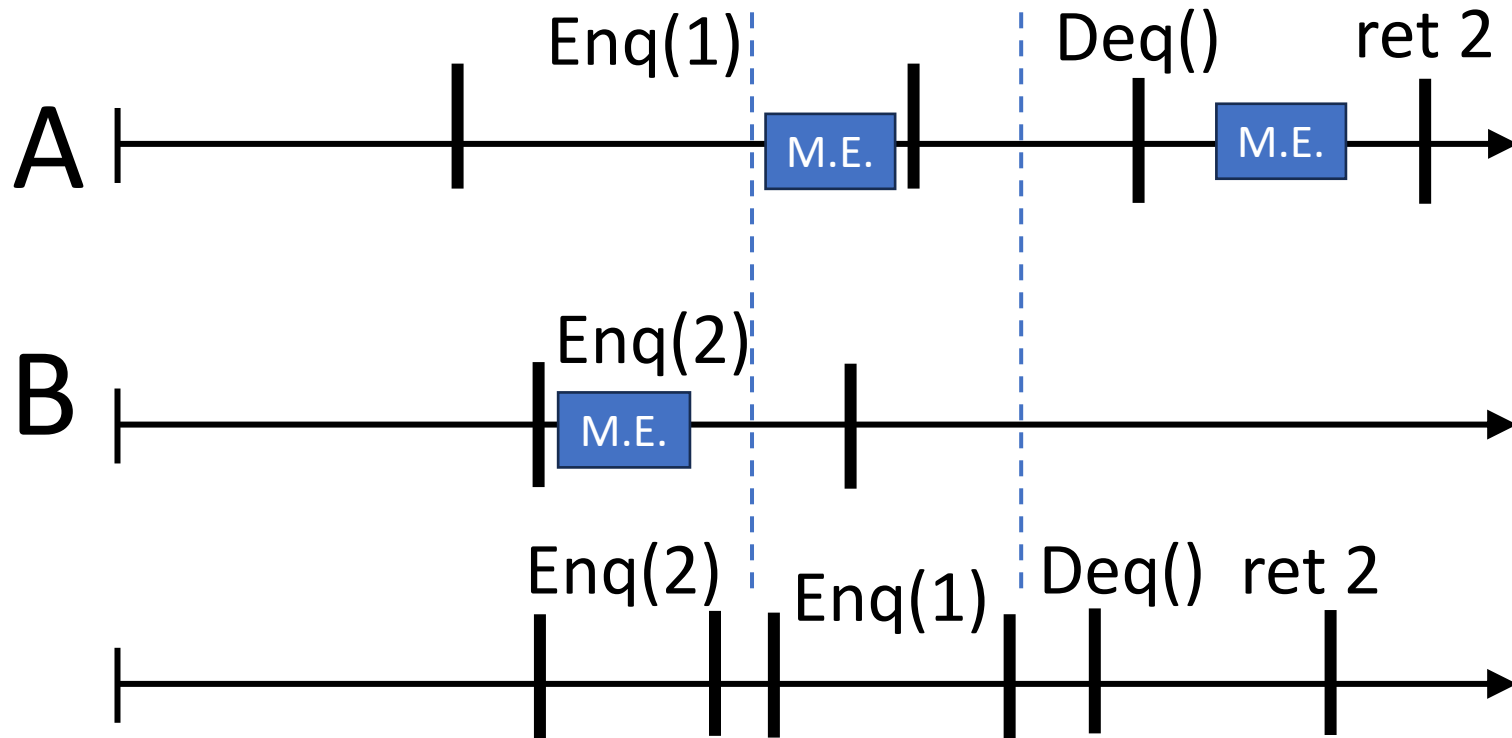
- Threads invoke methods
- Threads can experience arbitrary large delays



- 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

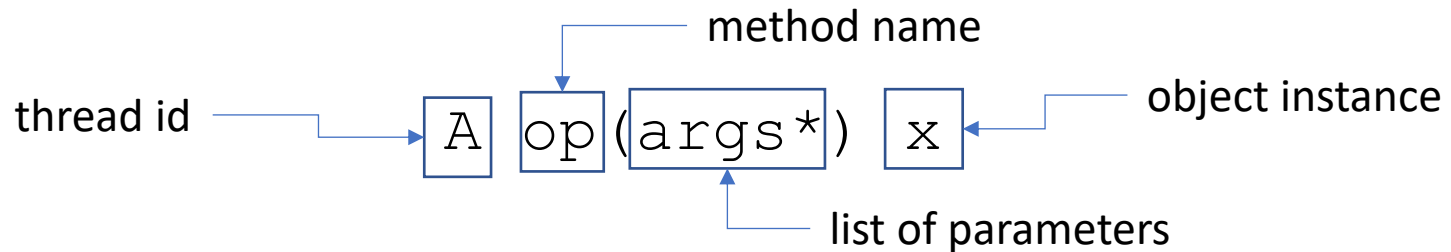
- An **execution** is correct if it is equivalent to a correct **sequential execution**

A simplified model of a concurrent system

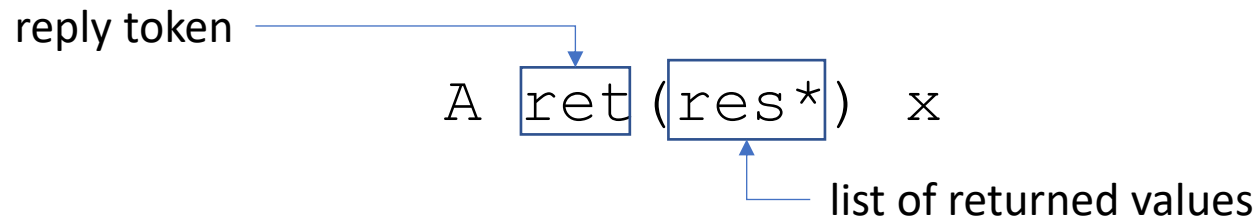
- A concurrent system is a collection of sequential threads/processes that communicate through shared data structures called objects.
- An object has a unique name and a set of primitive operations.

A simplified model of a concurrent execution

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



- Reply:



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
B op() x
A ret() x
A op() y
B ret() x
A ret() y

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

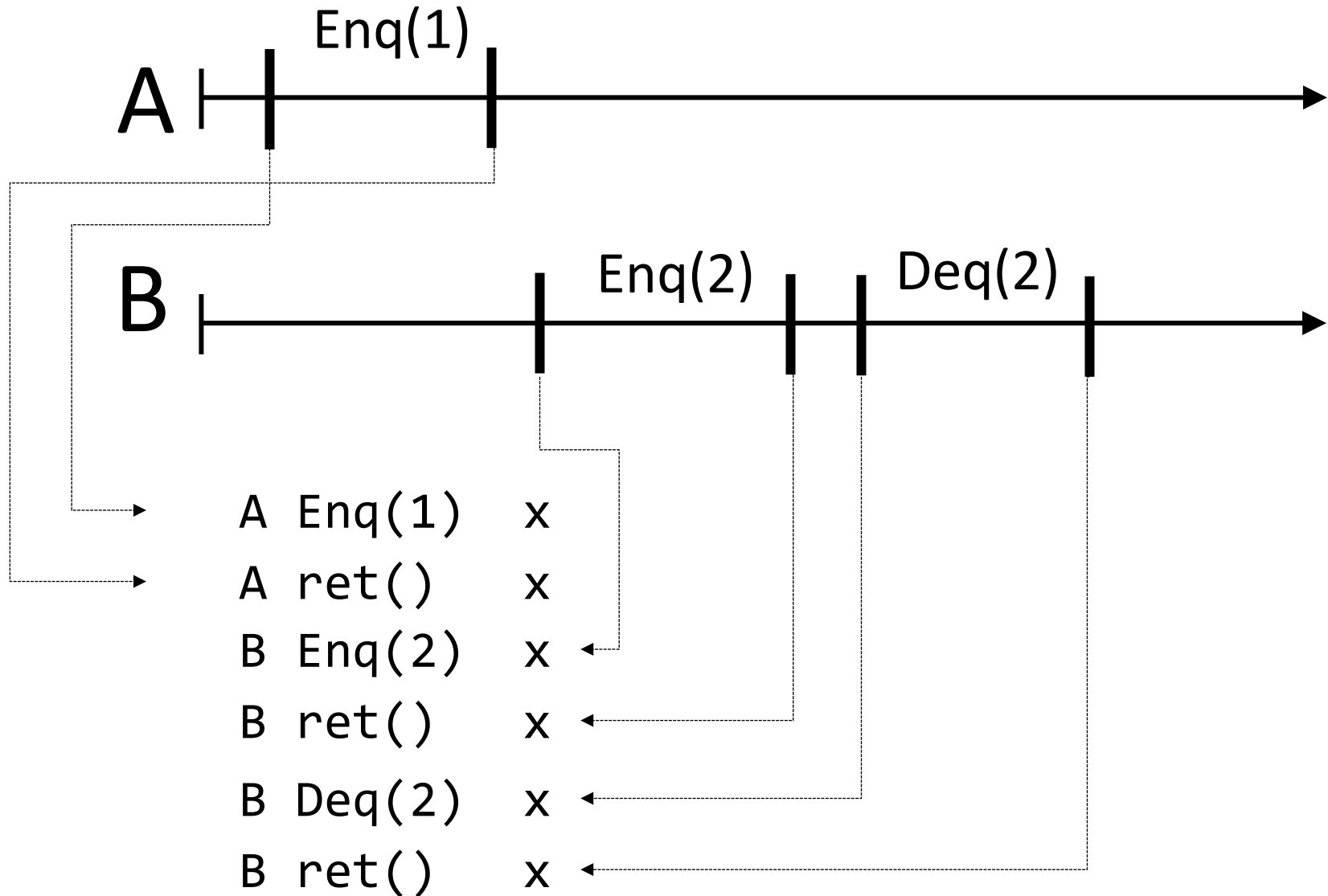
- A correctness condition specifies the set of histories to be considered as reference

⇒ 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
- ⇒ An object implementation is sequentially consistent if every history associated with its usage is sequentially consistent

Sequential Consistency [Lamport 1970]



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

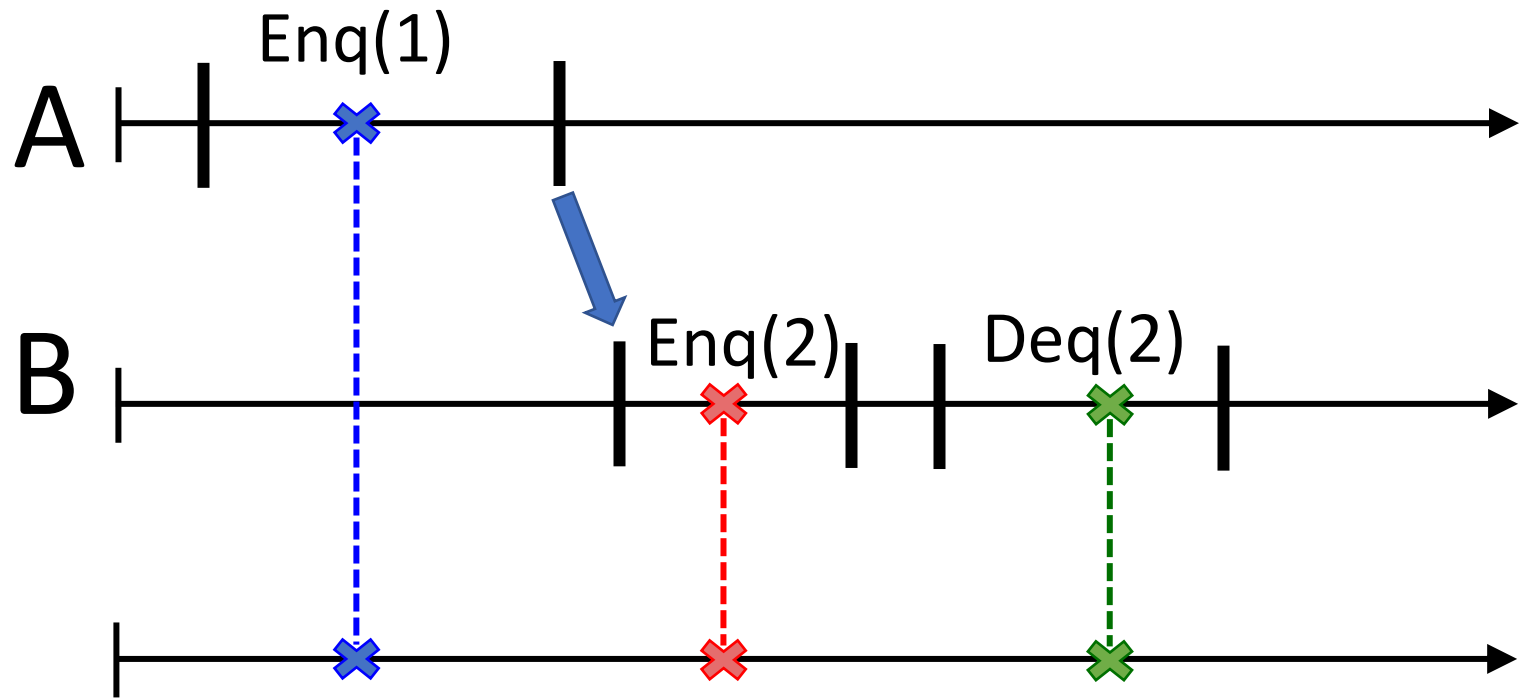
H' :

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

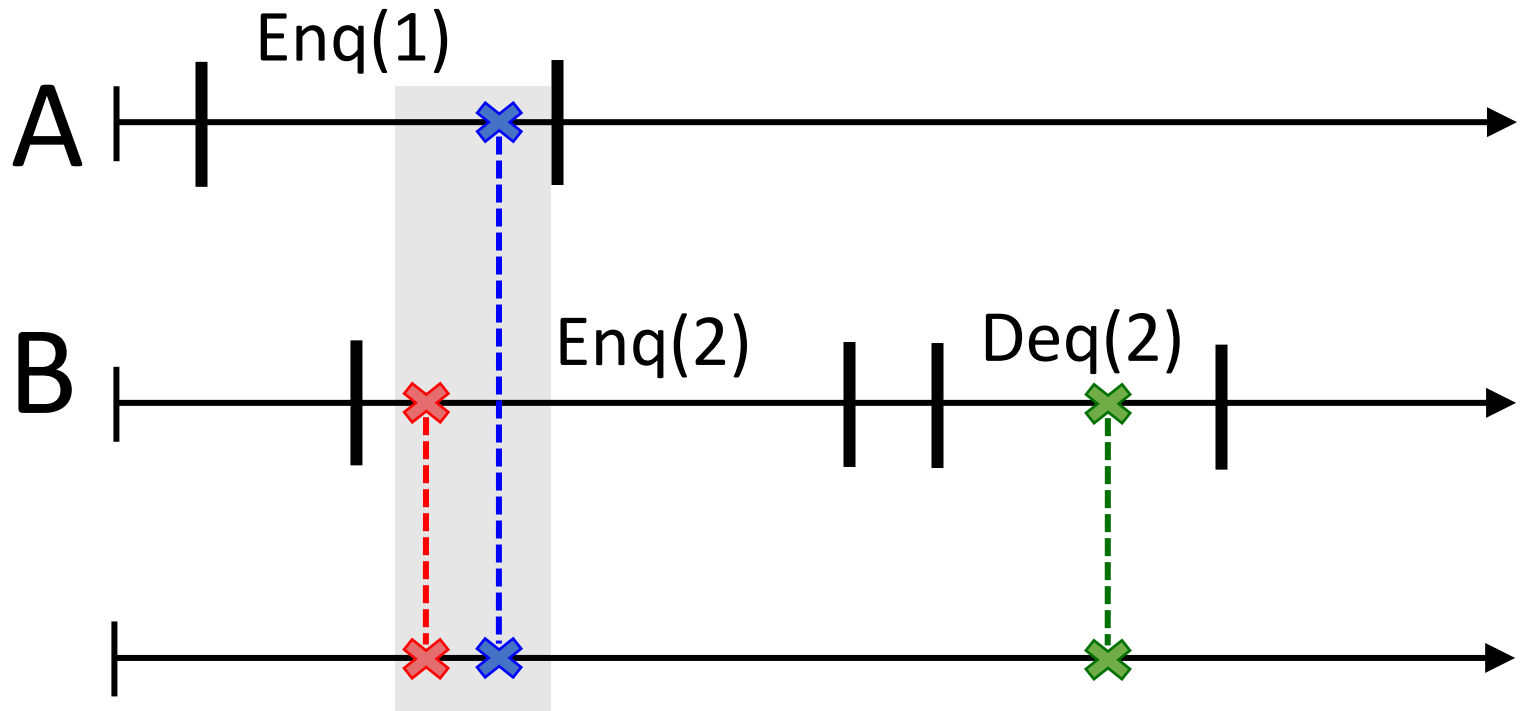
Linearizability [Herlihy 1990]

- 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
- ⇒ An object implementation is linearizable if every history associated with its usage can be linearized

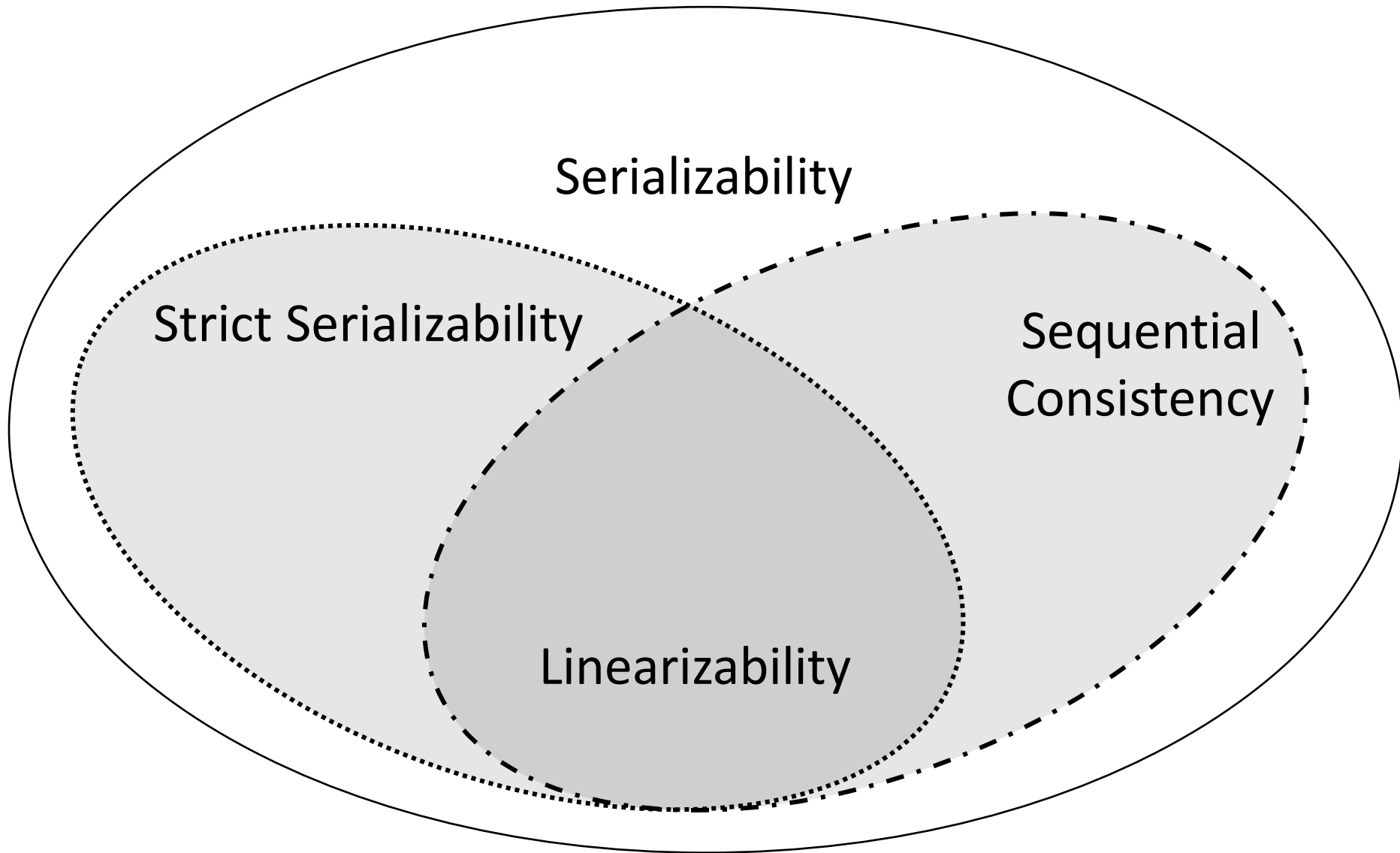
Linearizability [Herlihy 1990]

- 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	✗	✓	✗	✗