# Introduction to Consensus Problem and Protocols

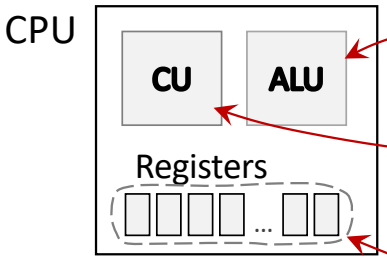
Artem Burmyakov

May 15, 2021

Talk Outline

- Recap of concurrent systems and their problems
- Race Condition – a canonical problem of concurrent systems
- Blocking and Non-Blocking concurrent algorithms
- Consensus Problem – a theoretical toy for studying concurrent systems
- Consensus Protocol – the solution of a consensus problem
- Consensus Numbers as the measure of synchronization ”strength” for data structures and CPU instructions

# Key Components of a CPU

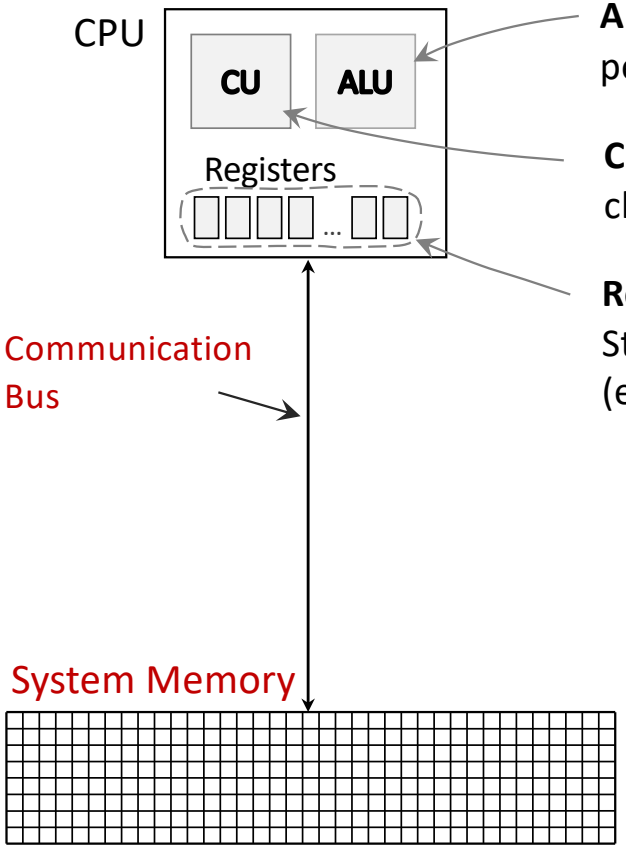



**ALU (Arithmetic-Logic Unit):**
performs arithmetic and logic operations

**CU (Control Unit):**
chooses the next instruction to execute, etc.

**Registers:**
Store data required by an executed instruction
(e.g. input arguments)

# Communication Bus and System Memory

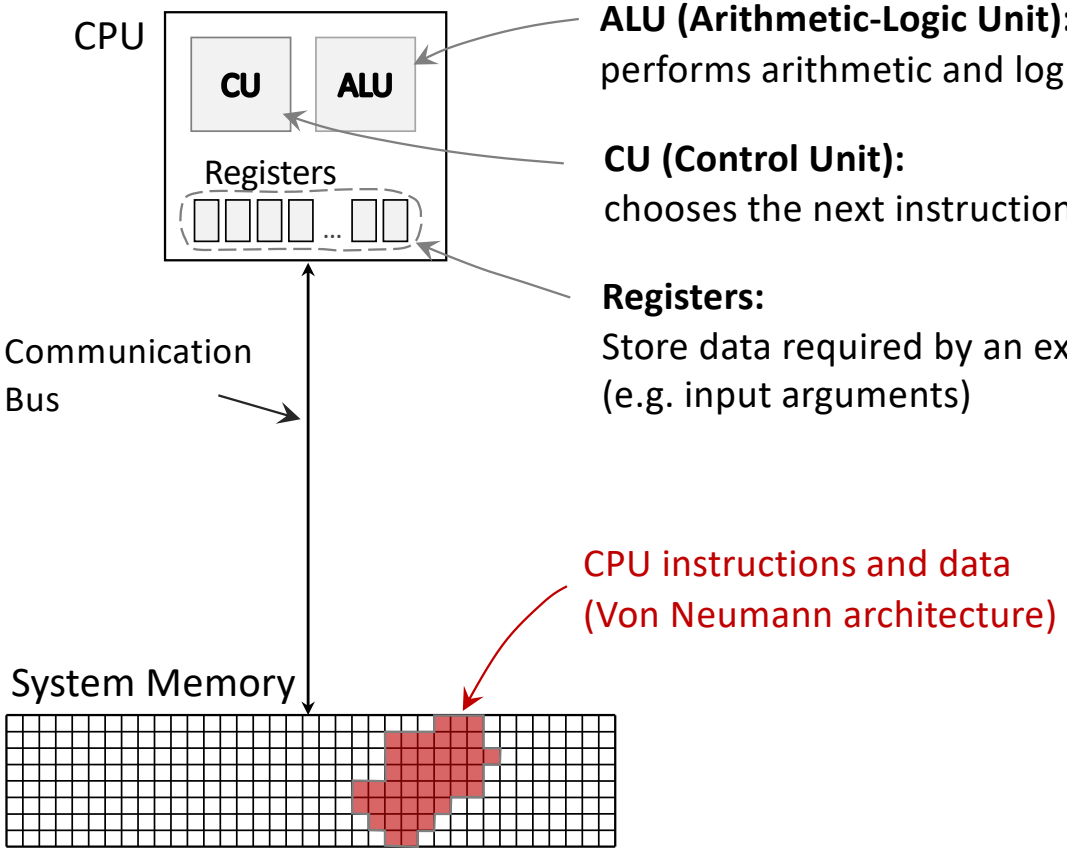



**ALU (Arithmetic-Logic Unit):**
performs arithmetic and logic operations

**CU (Control Unit):**
chooses the next instruction to execute, etc.

**Registers:**
Store data required by an executed instruction
(e.g. input arguments)

# Von Neumann Architecture



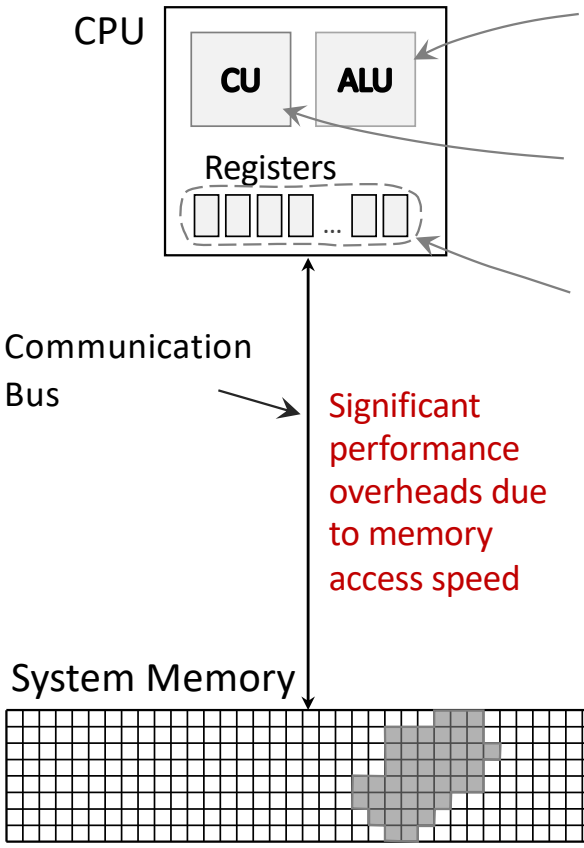

**ALU (Arithmetic-Logic Unit):**
performs arithmetic and logic operations

**CU (Control Unit):**
chooses the next instruction to execute, etc.

**Registers:**
Store data required by an executed instruction
(e.g. input arguments)

CPU instructions and data
(Von Neumann architecture)

# Performance Overheads Related to Memory Access

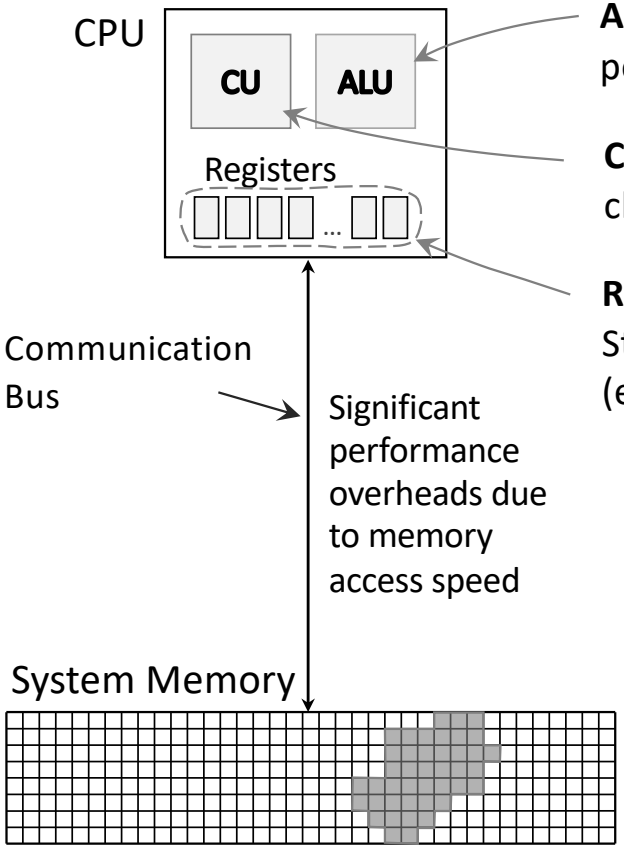



**ALU (Arithmetic-Logic Unit):**
performs arithmetic and logic operations

**CU (Control Unit):**
chooses the next instruction to execute, etc.

**Registers:**
Store data required by an executed instruction (e.g. input arguments)

# Performance Overheads Related to Memory Access

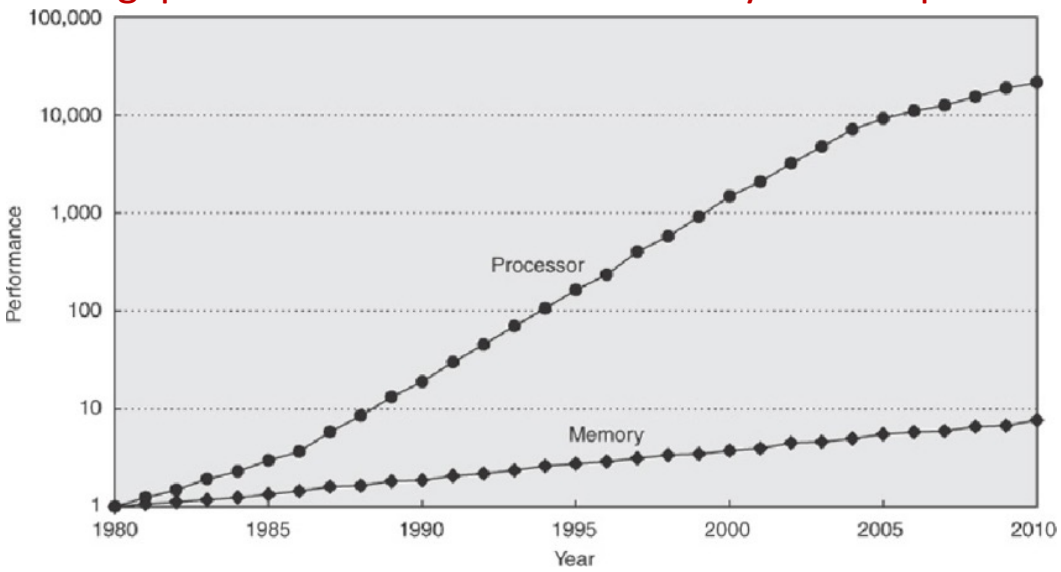



**ALU (Arithmetic-Logic Unit):**
performs arithmetic and logic operations

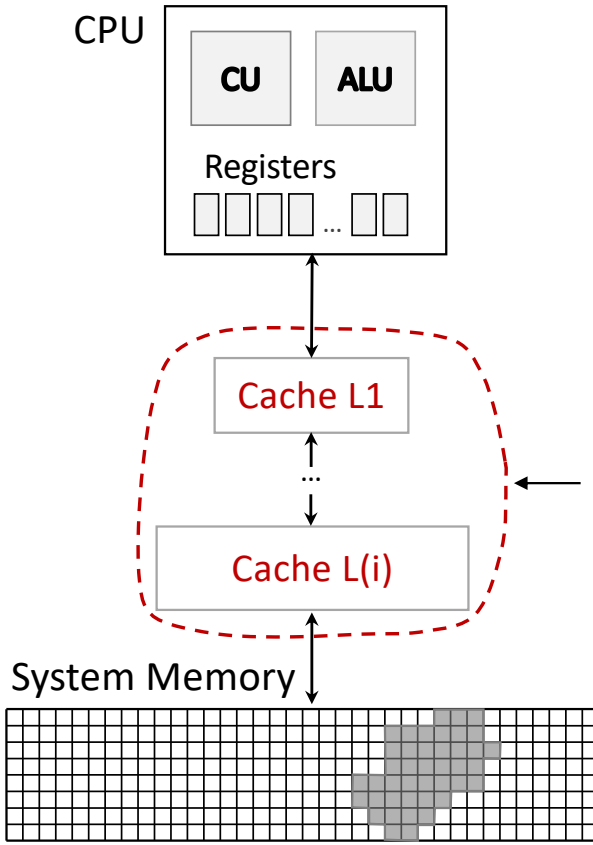
**CU (Control Unit):**
chooses the next instruction to execute, etc.

**Registers:**
Store data required by an executed instruction
(e.g. input arguments)

A “memory wall problem”
- the gap between the CPU and memory access speed

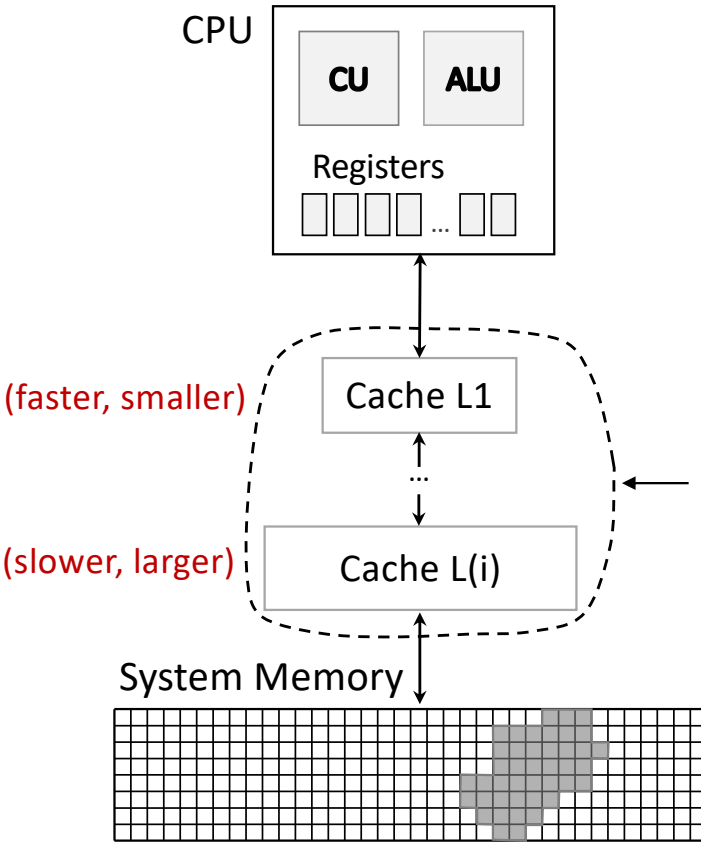


# Cache Hierarchy



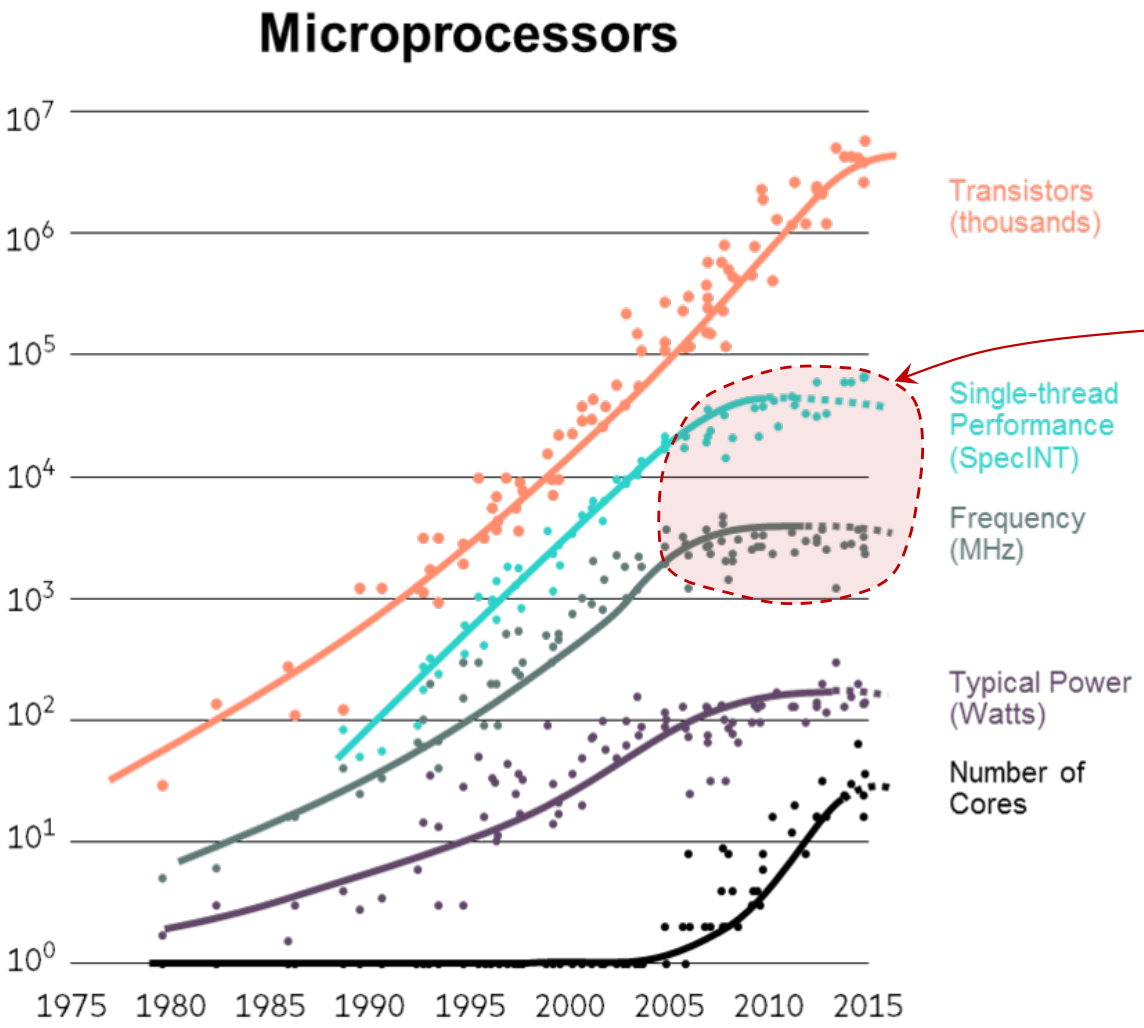

Cache hierarchy aims at minimizing the gap between a CPU speed and a system memory access speed

# Cache Hierarchy




Cache hierarchy aims at minimizing the gap between a CPU speed and a system memory access speed

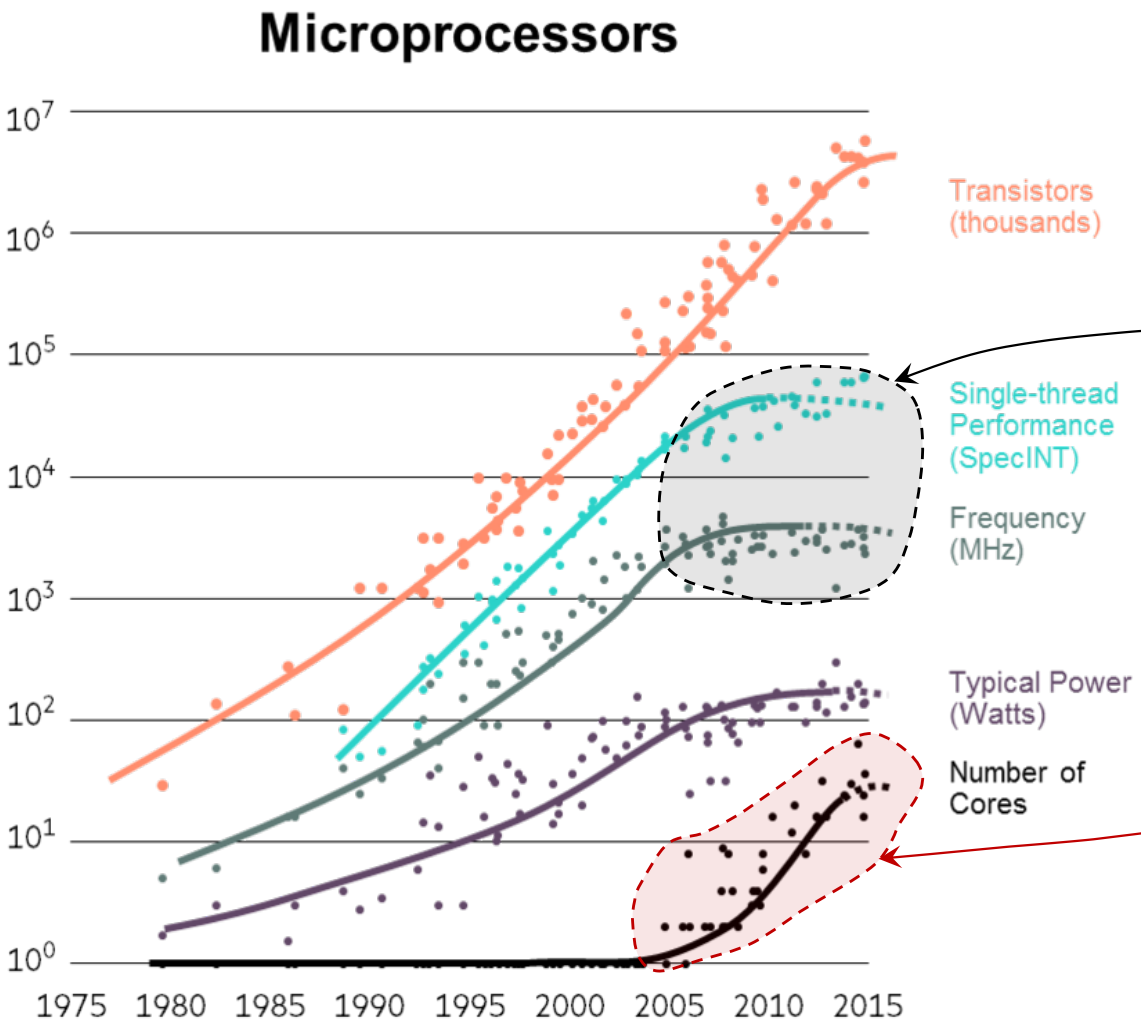
# Motivation for Multiprocessor Systems: Moore's Law Stagnation




CPU speed and single-thread performance seem not to increase since ~2008

Image is taken from https://www.quora.com/Why-is-Moores-law-no-longer-valid

# Motivation for Multiprocessor Systems: Moore's Law Stagnation

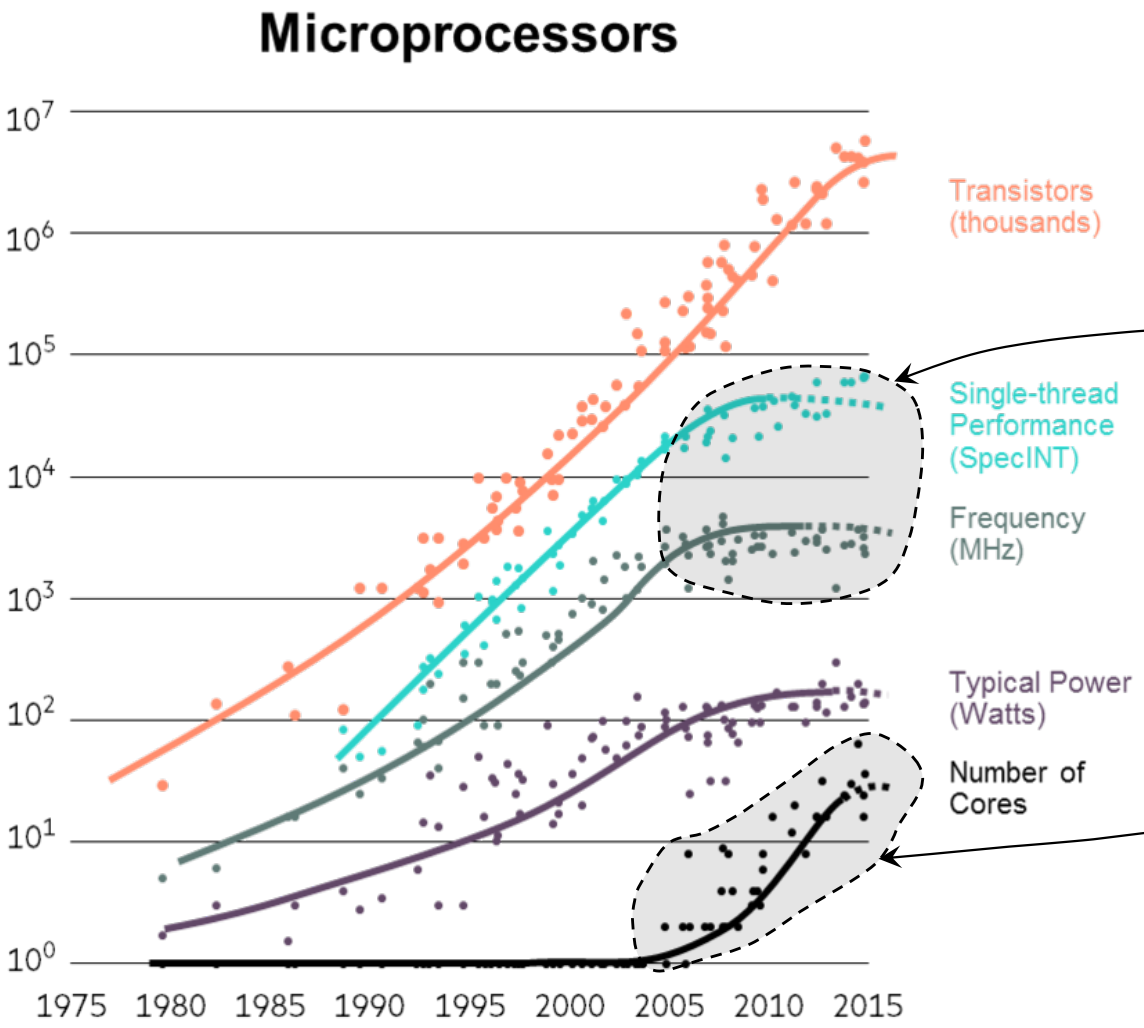



CPU speed and single-thread performance seem not to increase since ~2008

The number of cores increases instead

Image is taken from https://www.quora.com/Why-is-Moores-law-no-longer-valid

# Motivation for Multiprocessor Systems: Moore's Law Stagnation



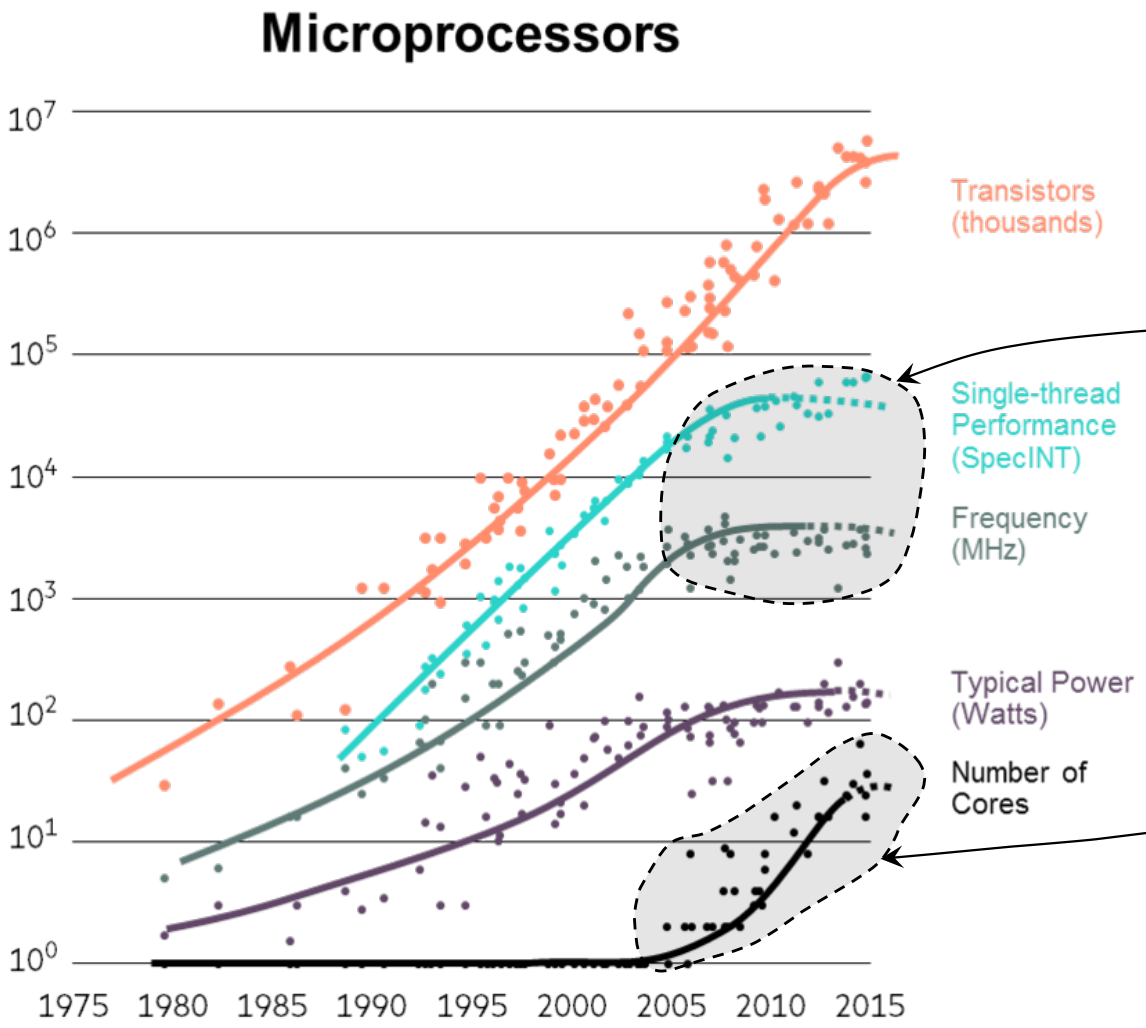

CPU speed and single-thread performance seem not to increase since ~2008

The number of cores increases instead

Nowadays, the increase of performance is achieved through the number of CPUs (or CPU cores), rather than a further increase of CPU speed

Image is taken from https://www.quora.com/Why-is-Moores-law-no-longer-valid

# Motivation for Multiprocessor Systems: Moore's Law Stagnation




CPU speed and single-thread performance seem not to increase since ~2008

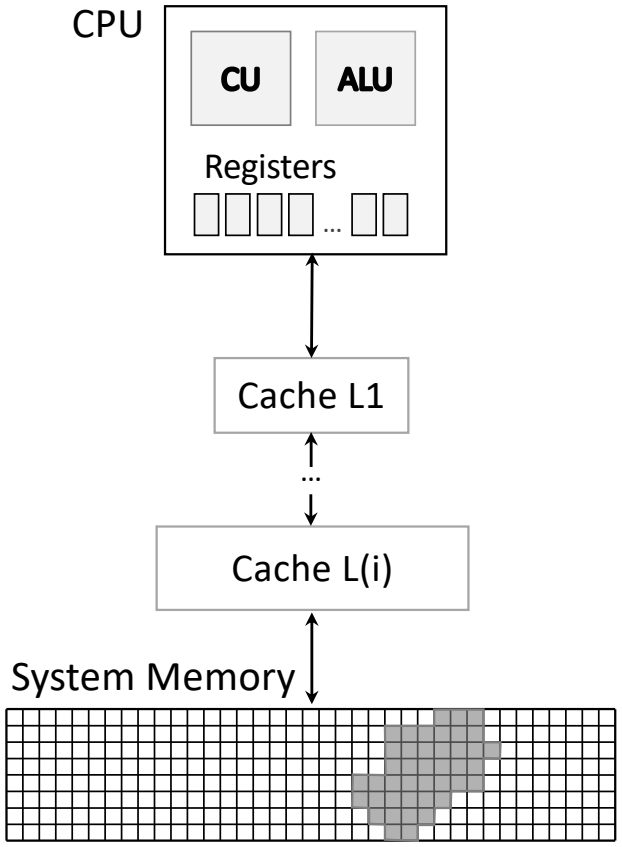
The number of cores increases instead

Nowadays, the increase of performance is achieved through the number of CPUs (or CPU cores), rather than a further increase of CPU speed

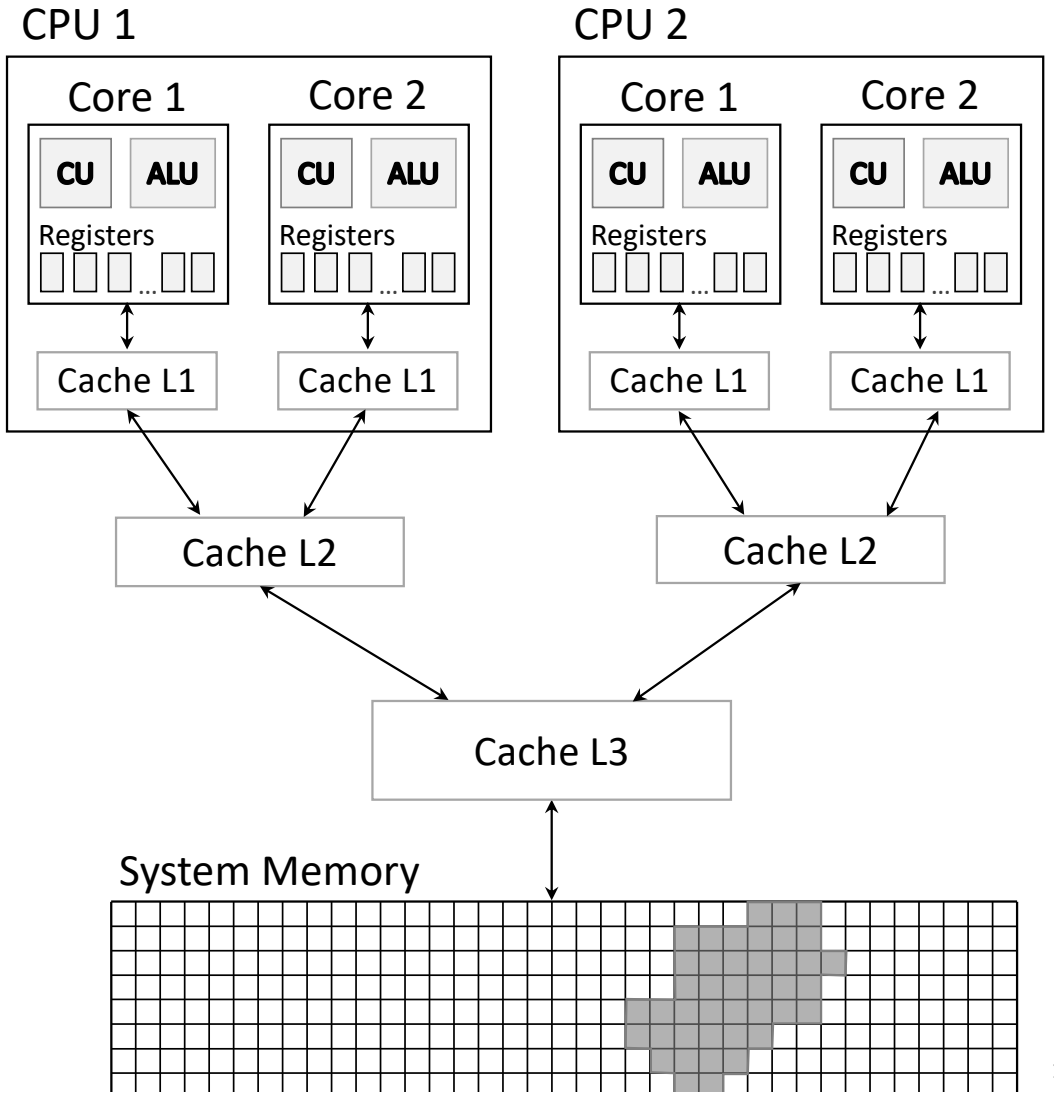
Concurrent programming gets important

Image is taken from https://www.quora.com/Why-is-Moores-law-no-longer-valid

# Single Core vs. Multicore Platform




Multicore (Multiprocessor) Platform



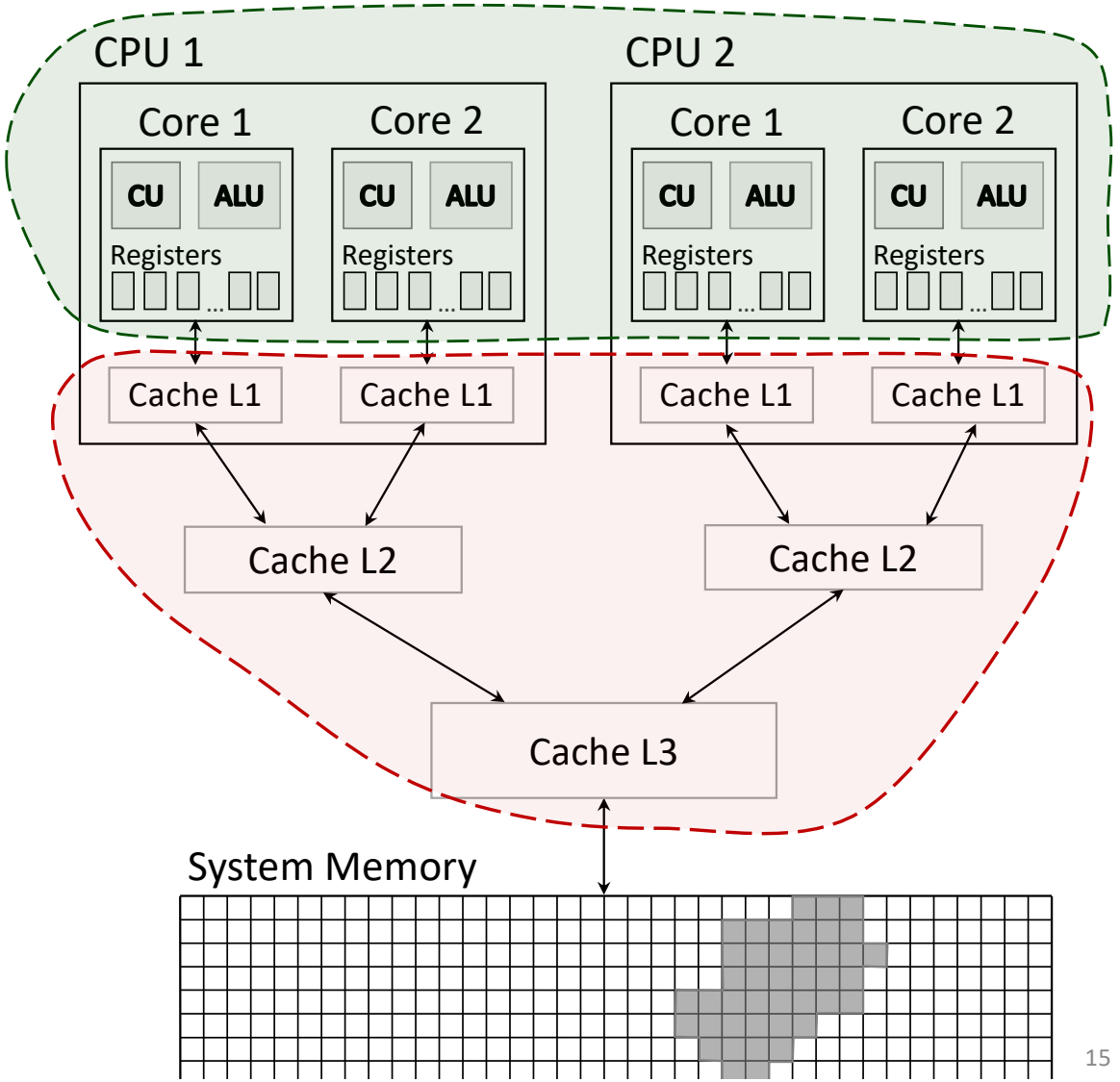
# Multicore Platform: Advantages and Disadvantages

**Higher execution throughput**:

supports multiple threads of execution

**Higher performance overheads**:

- Expensive context switches and migrations between CPUs;
- Preemption overheads (depend on OS, significant)



# Multicore Platform: Advantages and Disadvantages

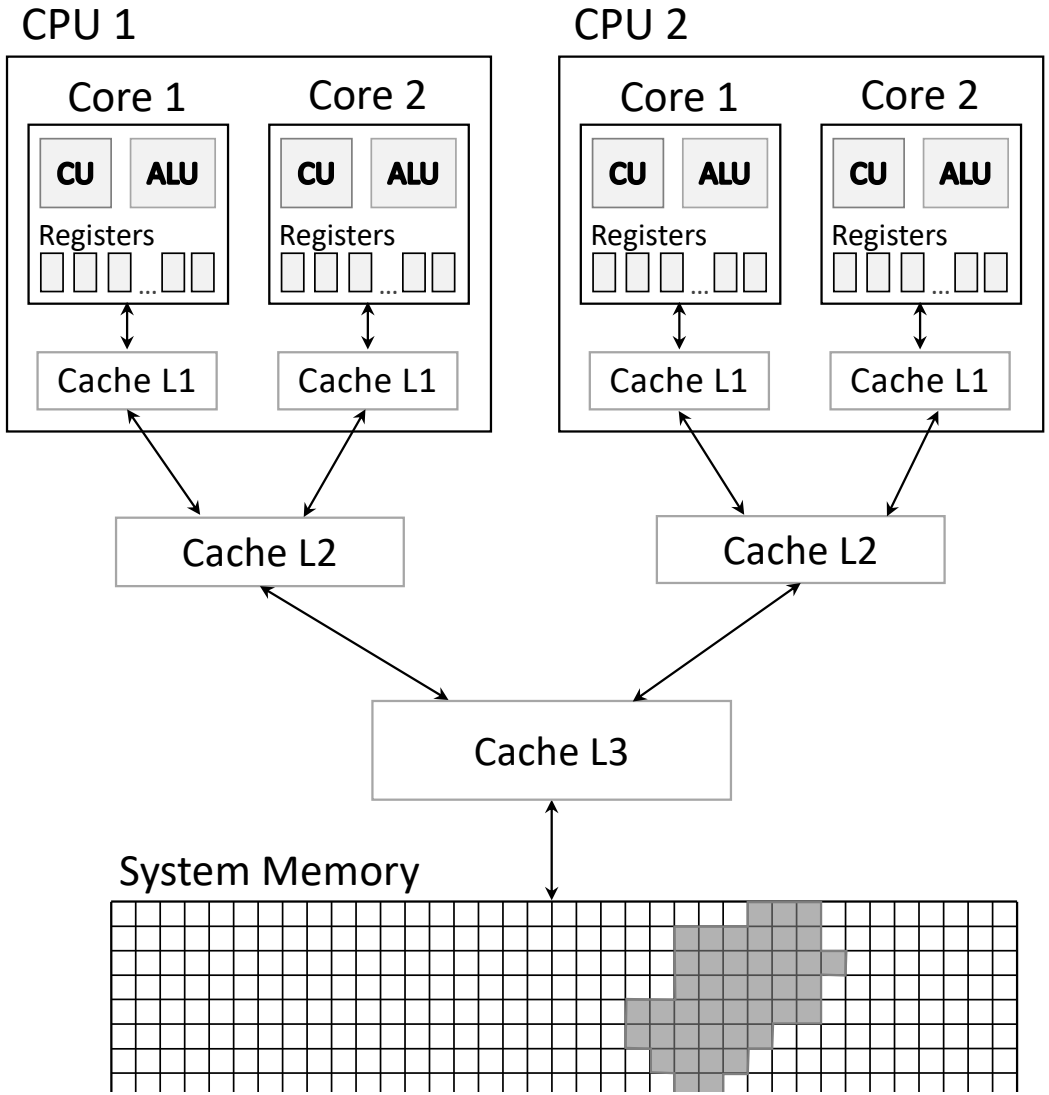
**Higher execution throughput:**

supports multiple threads of execution

**Higher performance overheads**:

- Expensive context switches and migrations between CPUs;
- Preemption overheads (depend on OS, significant)

There are multiple other problems as well



# Execution of a Sample Single-Threaded Program

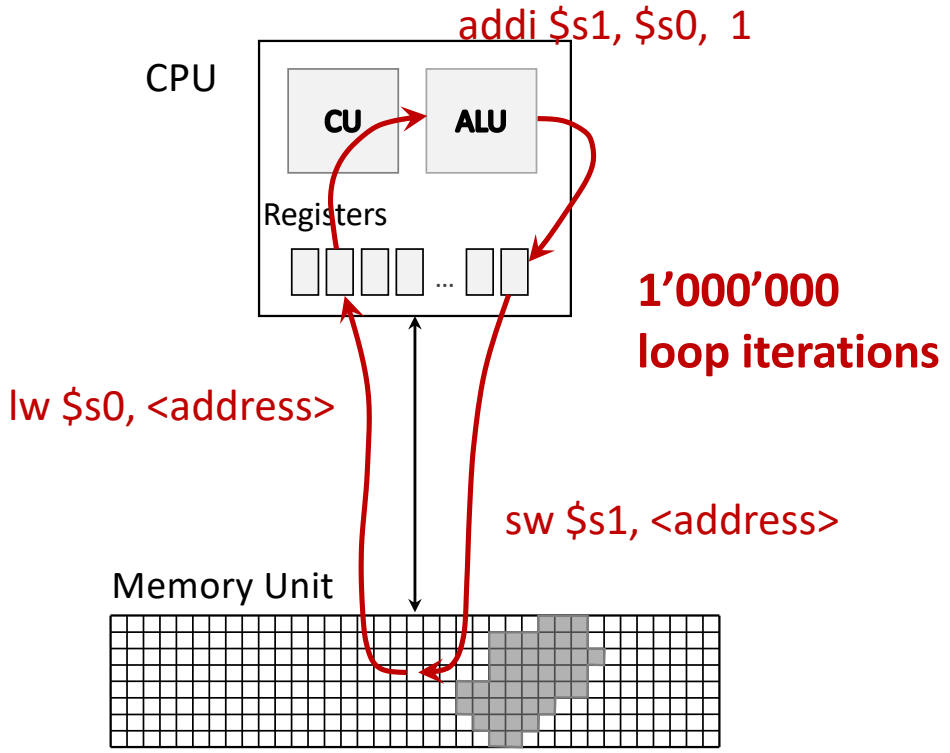
```
unsigned single_thread()
{
  unsigned a = 0;
  for (unsigned i = 0; i < 1’000’000; i++)
       a++;

  return a;
}
```

Function always returns a = 1’000’000

# Execution of a Sample Single-Threaded Program

```
unsigned single_thread()
{
  unsigned a = 0;
  for (unsigned i = 0; i < 1'000'000; i++)
       a++;

  return a;
}
```

Function always returns a = 1'000'000




Assumption: No cache considered (for simplicity)

# Execution of a Sample Two-Threaded Program

```
void thread1(unsigned *a) {
  for (unsigned i = 0; i < 500’000; i++)
       (*a)++;
  return a;
}

void thread2(unsigned *a) {
  for (unsigned i = 0; i < 500’000; i++)
       (*a)++;
  return a;
}

int main() {
  unsigned a = 0;
  thread first(thread1, &a);
  thread second(thread2, &a);
  first.join();
  second.join();
  return 1;
}
```

What do we expect to happen?

# Execution of a Sample Two-Threaded Program

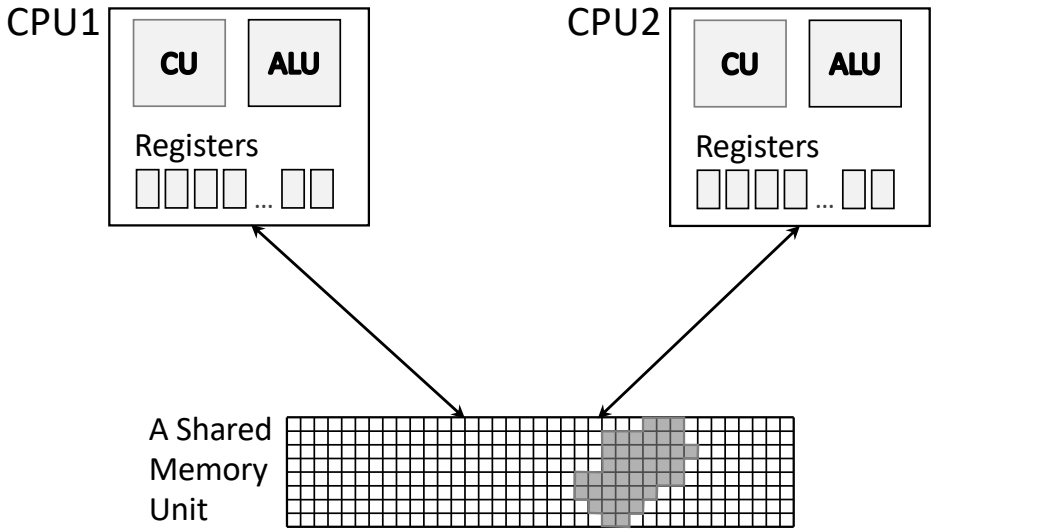
```
void thread1(unsigned *a) {
  for (unsigned i = 0; i < 500’000; i++)
       (*a)++;
  return a;
}

void thread2(unsigned *a) {
  for (unsigned i = 0; i < 500’000; i++)
       (*a)++;
  return a;
}

int main() {
  unsigned a = 0;
  thread first(thread1, &a);
  thread second(thread2, &a);
  first.join();
  second.join();
  return 1;
}
```

What do we expect to happen?

- “a” to be equal to 1’000’000 ?
- ~2 times faster computation time ?

# Race Condition – a Canonical Problem of Concurrent Systems

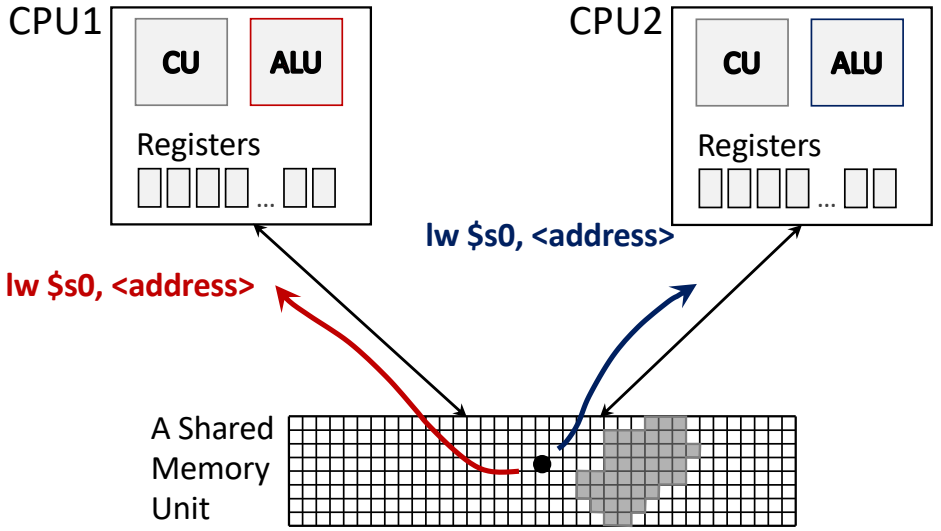
```
void thread1(unsigned *a) {
  for (unsigned i = 0; i < 500’000; i++)
       (*a)++;
  return a;
}

void thread2(unsigned *a) {
  for (unsigned i = 0; i < 500’000; i++)
       (*a)++;
  return a;
}

int main() {
  unsigned a = 0;
  thread first(thread1, &a);
  thread second(thread2, &a);
  first.join();
  second.join();
  return 1;
}
```




Assumption: No cache for simplicity

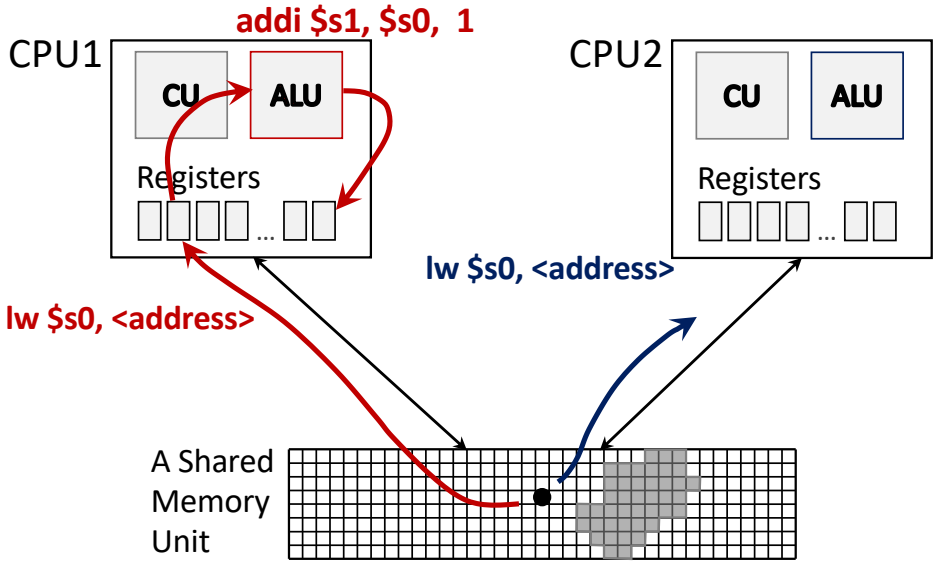
# Race Condition – a Canonical Problem of Concurrent Systems

```
void thread1(unsigned *a) {
  for (unsigned i = 0; i < 500’000; i++)
       (*a)++;
  return a;
}

void thread2(unsigned *a) {
  for (unsigned i = 0; i < 500’000; i++)
       (*a)++;
  return a;
}
```



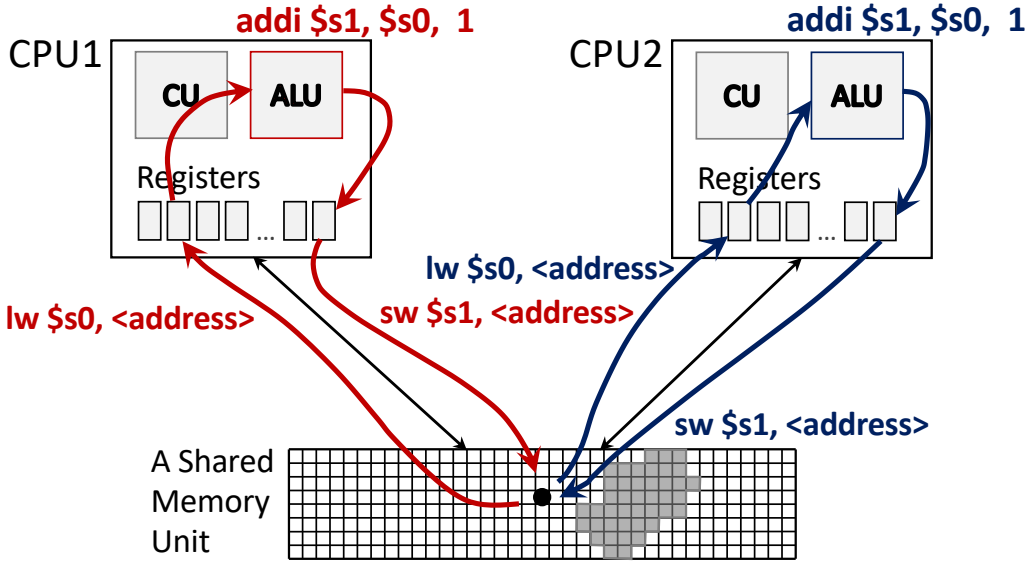

# Race Condition – a Canonical Problem of Concurrent Systems

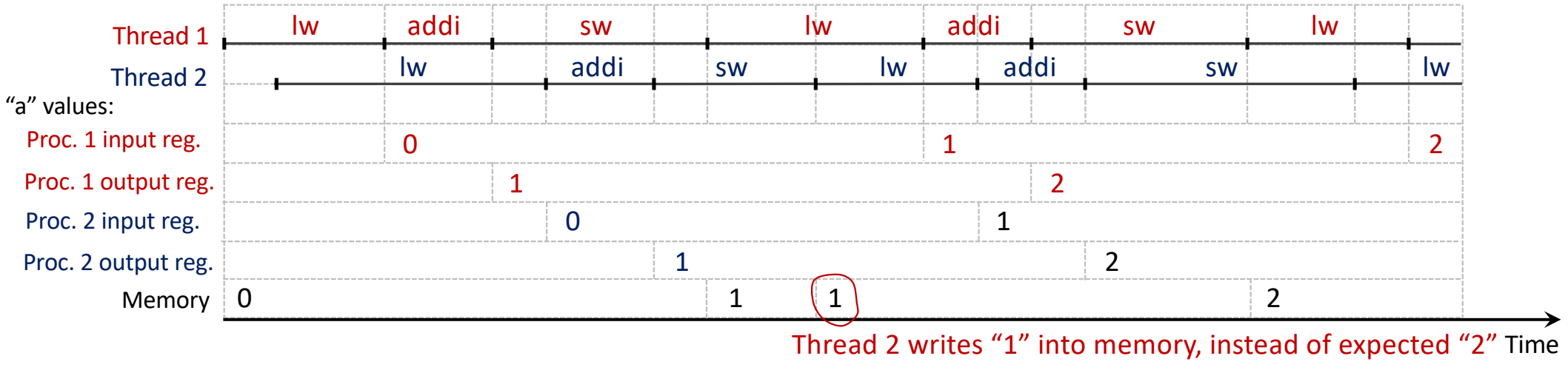
```
void thread1(unsigned *a) {
  for (unsigned i = 0; i < 500’000; i++)
       (*a)++;
  return a;
}

void thread2(unsigned *a) {
  for (unsigned i = 0; i < 500’000; i++)
       (*a)++;
  return a;
}
```




# Race Condition – a Canonical Problem of Concurrent Systems

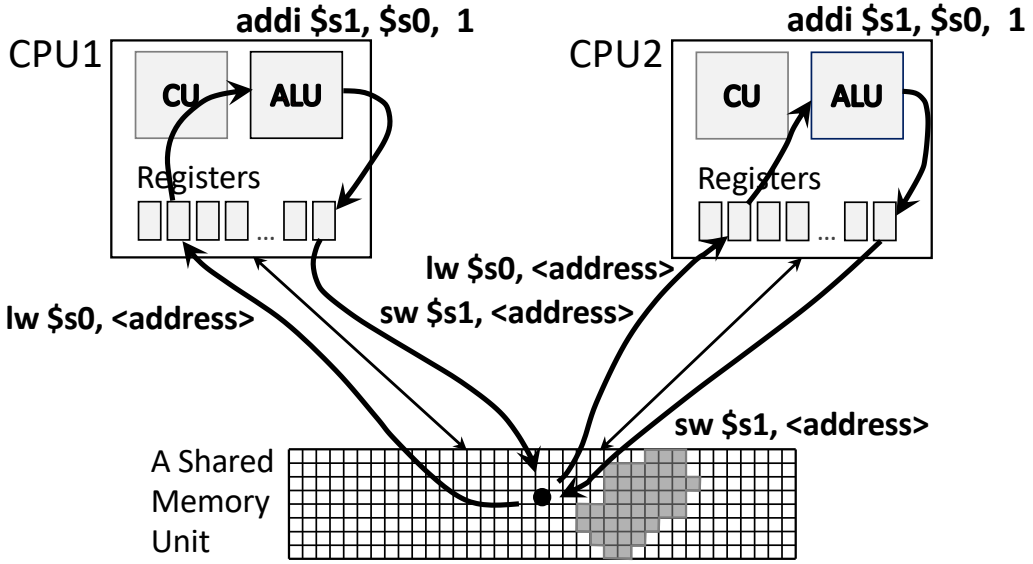
```
void thread1(unsigned *a) {
  for (unsigned i = 0; i < 500'000; i++)
       (*a)++;
  return a;
}

void thread2(unsigned *a) {
  for (unsigned i = 0; i < 500'000; i++)
       (*a)++;
  return a;
}
```



# Race Condition – a Canonical Problem of Concurrent Systems

```
void thread1(unsigned *a) {
  for (unsigned i = 0; i < 500’000; i++)
       (*a)++;
  return a;
}

void thread2(unsigned *a) {
  for (unsigned i = 0; i < 500’000; i++)
       (*a)++;
  return a;
}

int main() {
  unsigned a = 0;
  thread first(thread1, &a);
  thread second(thread2, &a);
  first.join();
  second.join();
  return 1;
}
```

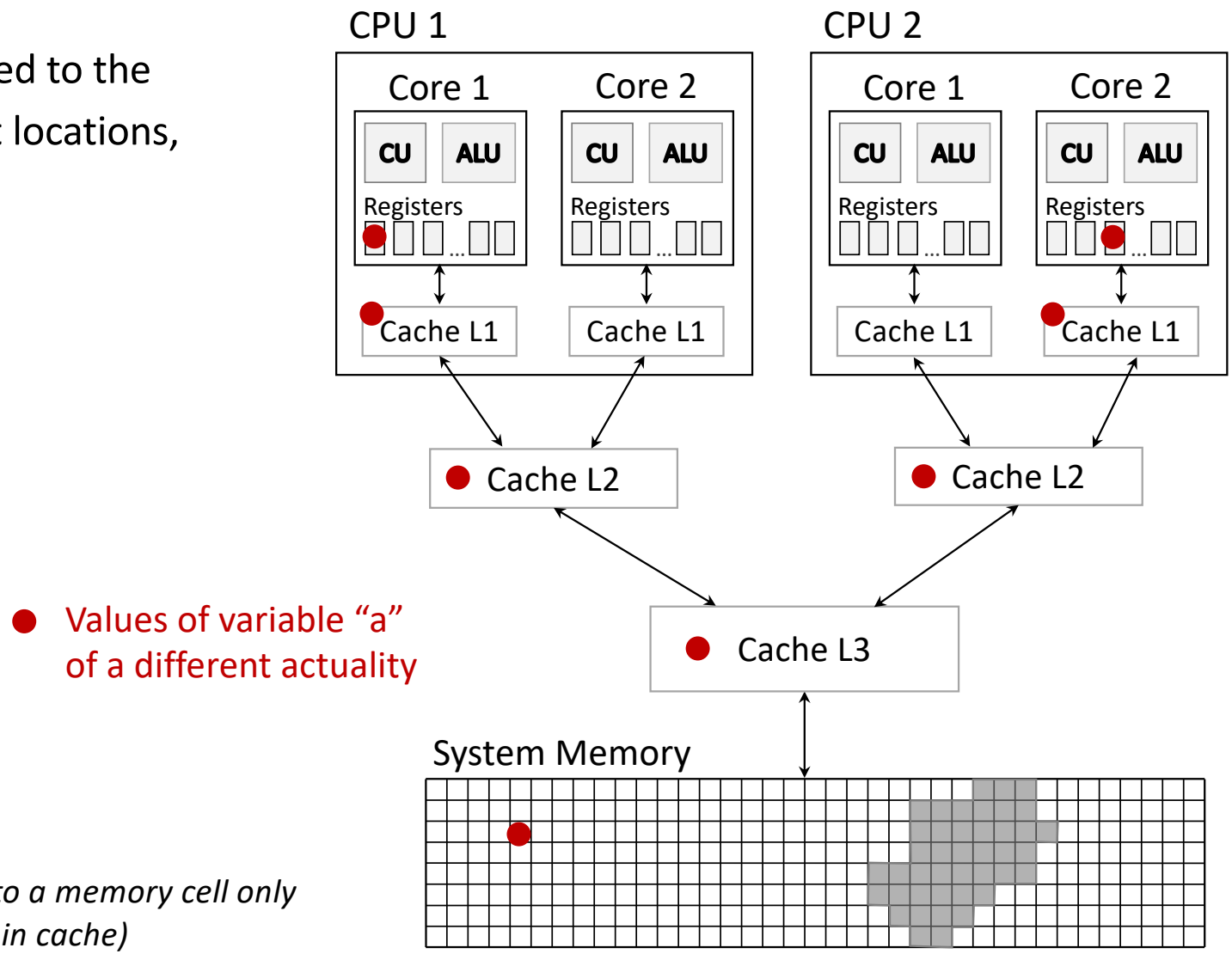



The resulted “a” will be less than 1’000’000

## Value “Actuality” for a Program Variable

There are several values associated to the same variable, stored at different locations, with a different “actuality”:

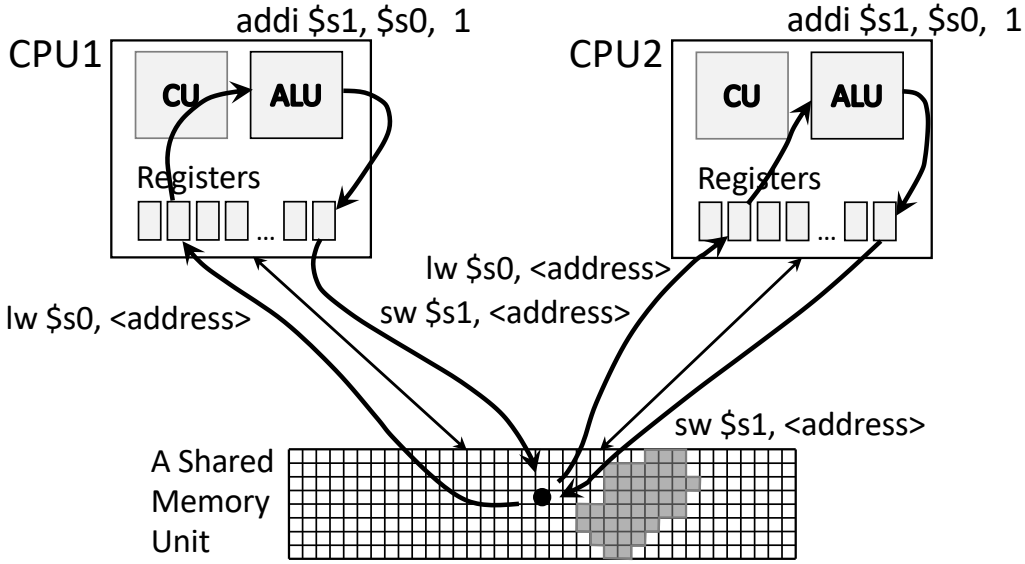
- Processor register;
- Different caches;
- Main memory




*Note: the pointer is dedicated to point to a memory cell only (not to a processor register, or location in cache)*

# The Notion of Critical Sections

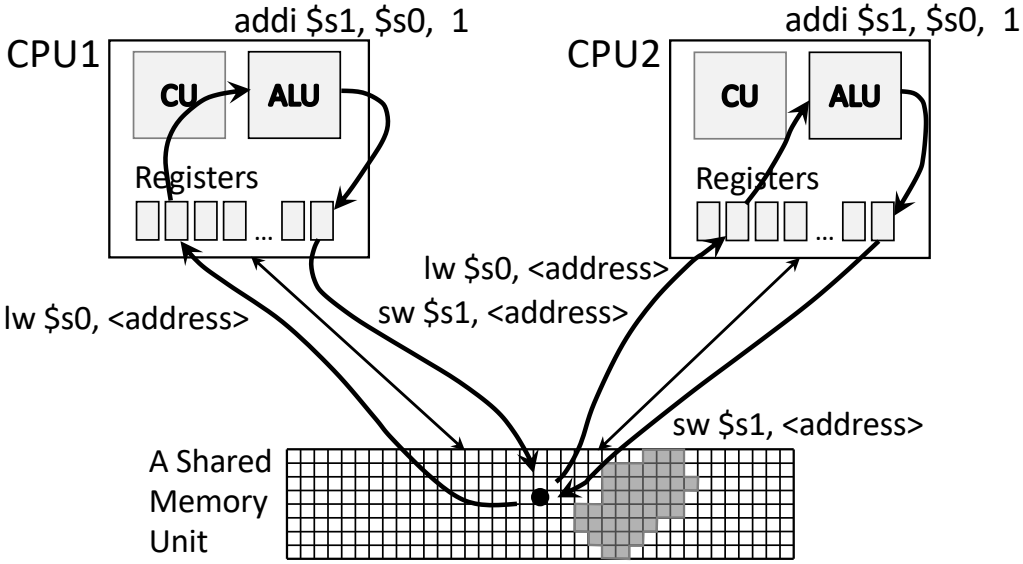
```
void thread1(unsigned *a) {
  for (unsigned i = 0; i < 100’000; i++)
        (*a)++;
  return a;
}

void thread2(unsigned *a) {
  for (unsigned i = 0; i < 100’000; i++)
        (*a)++;
  return a;
}

int main() {
  unsigned a = 0;
  thread first(thread1, &a);
  thread second(thread2, &a);
  first.join();
  second.join();
  return 1;
}
```




**To put critical sections** – blocks of code, providing mutual exclusion for access to a shared variable

# Race Condition: Explicit Synchronization of Critical Sections

```
void thread1(unsigned *a) {
  for (unsigned i = 0; i < 100’000; i++)
       (*a)++;
  return a;
}

void thread2(unsigned *a) {
  for (unsigned i = 0; i < 100’000; i++)
       (*a)++;
  return a;
}
```




Thread 1

Thread 2

“a” values:

Proc. 1 load reg.

Proc. 1 store reg.

Proc. 2 load reg.

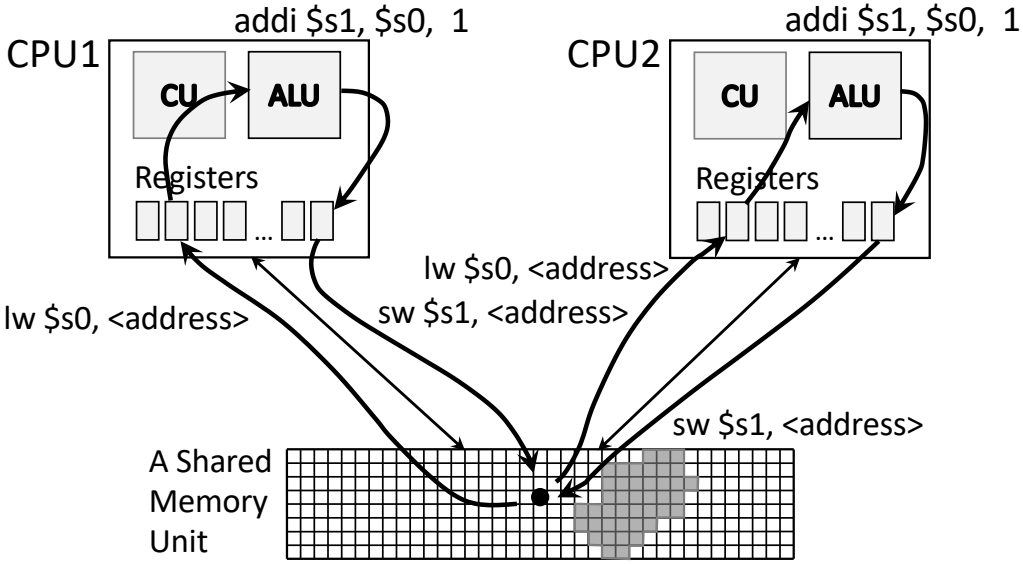
Proc. 2 store reg.

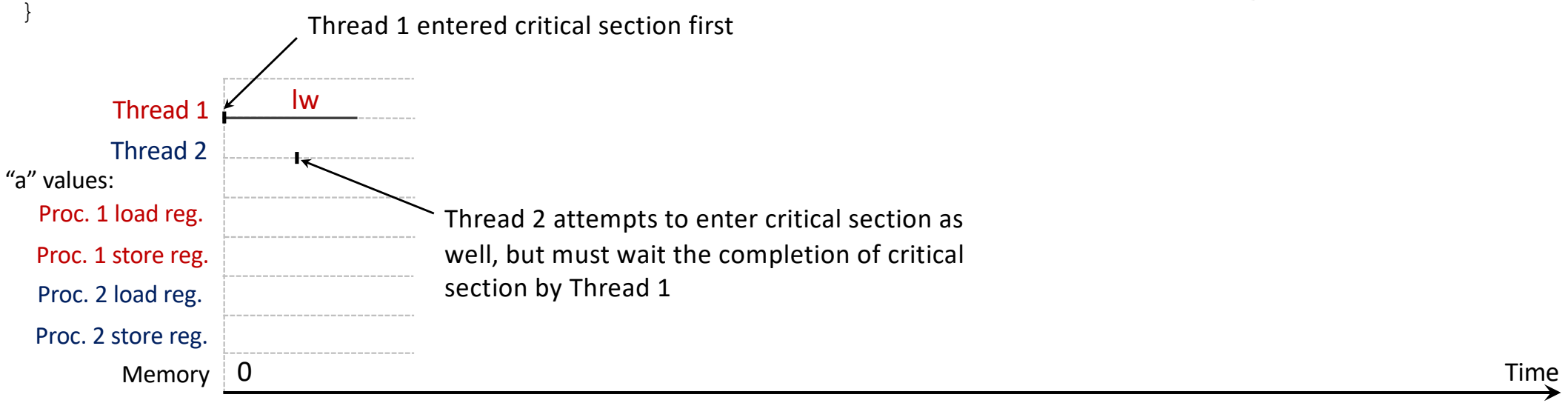
Memory 0

Time

# Race Condition: Explicit Synchronization of Critical Sections

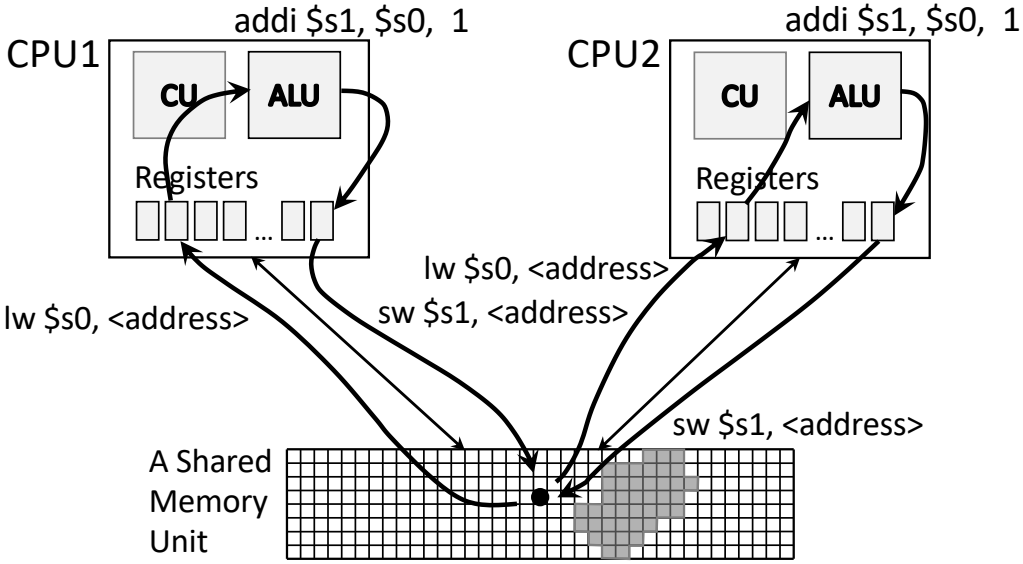
```
void thread1(unsigned *a) {
  for (unsigned i = 0; i < 100’000; i++)
       (*a)++;
  return a;
}

void thread2(unsigned *a) {
  for (unsigned i = 0; i < 100’000; i++)
       (*a)++;
  return a;
}
```

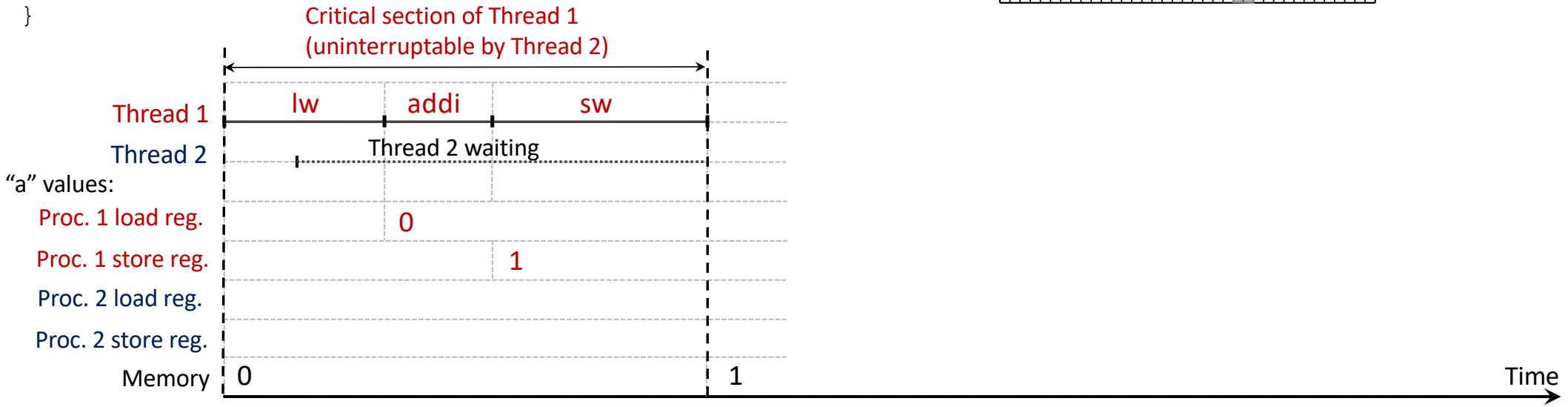


# Race Condition: Explicit Synchronization of Critical Sections

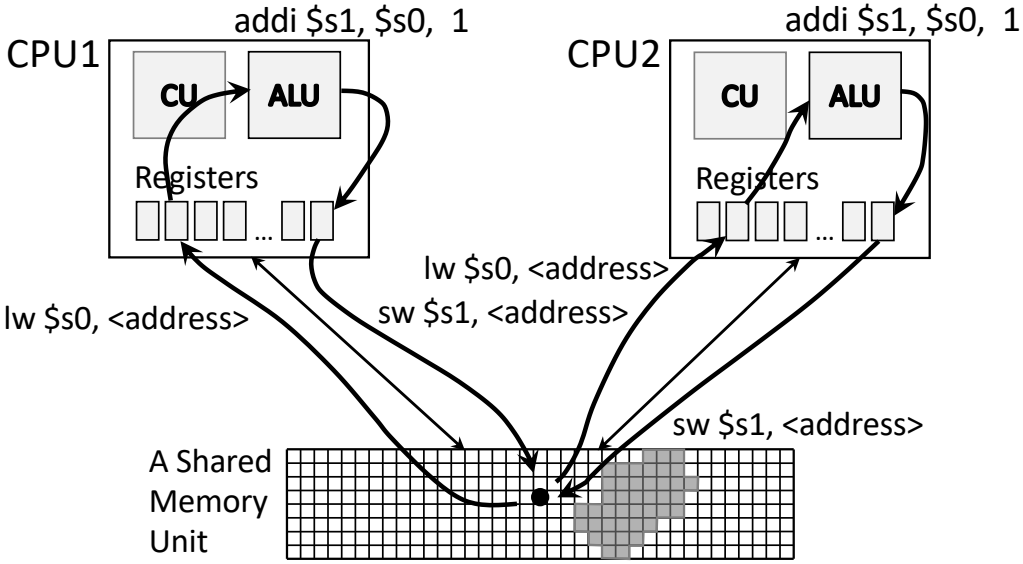
```
void thread1(unsigned *a) {
  for (unsigned i = 0; i < 100’000; i++)
       (*a)++;
  return a;
}

void thread2(unsigned *a) {
  for (unsigned i = 0; i < 100’000; i++)
       (*a)++;
  return a;
}
```

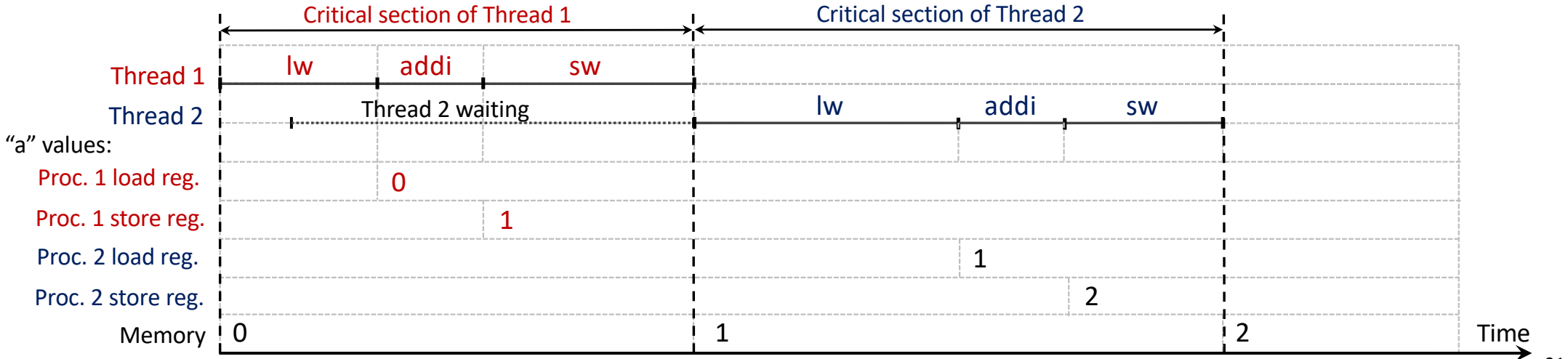


# Race Condition: Explicit Synchronization of Critical Sections

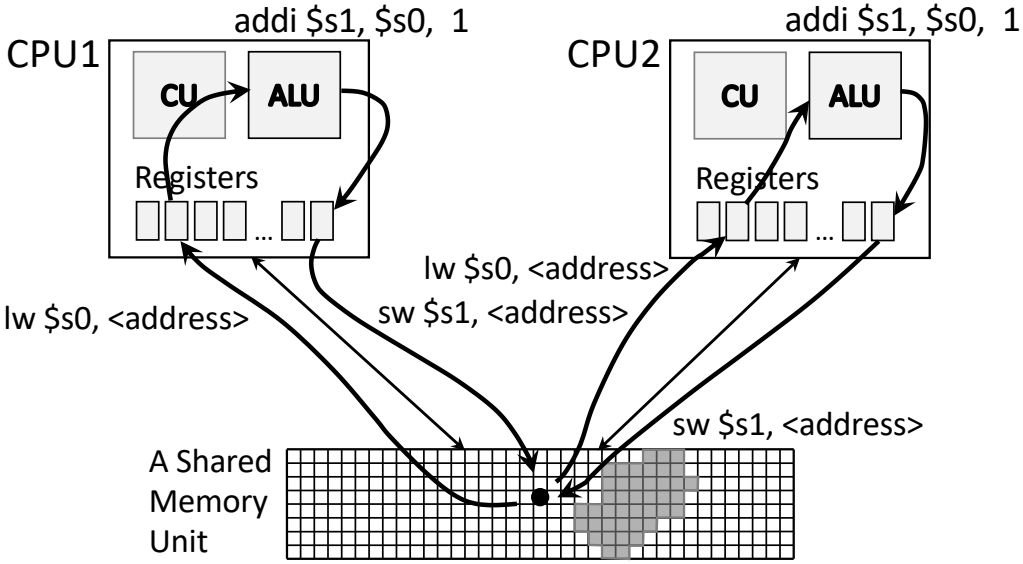
```
void thread1(unsigned *a) {
  for (unsigned i = 0; i < 100’000; i++)
       (*a)++;
  return a;
}

void thread2(unsigned *a) {
  for (unsigned i = 0; i < 100’000; i++)
       (*a)++;
  return a;
}
```

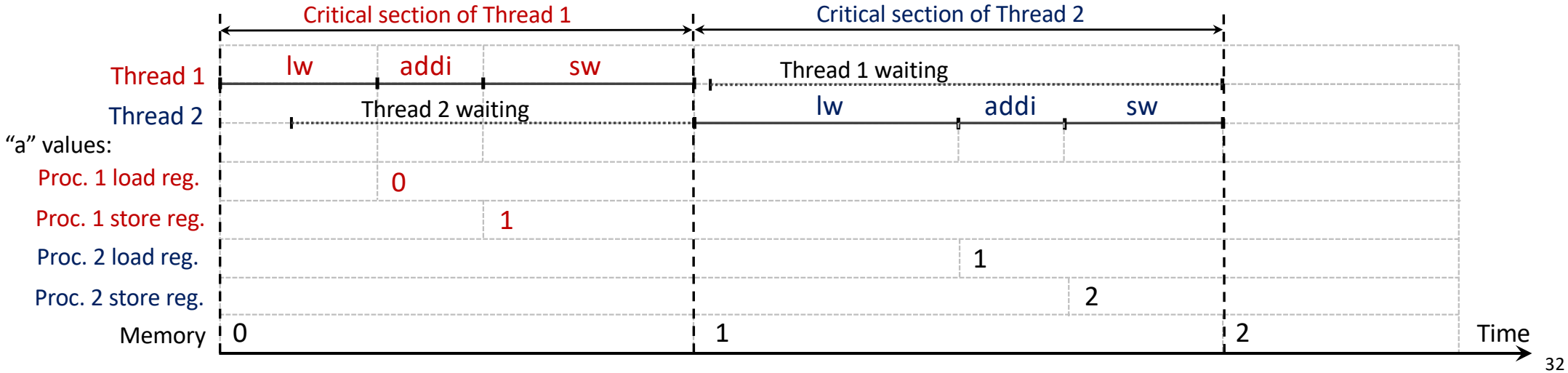


# Race Condition: Explicit Synchronization of Critical Sections

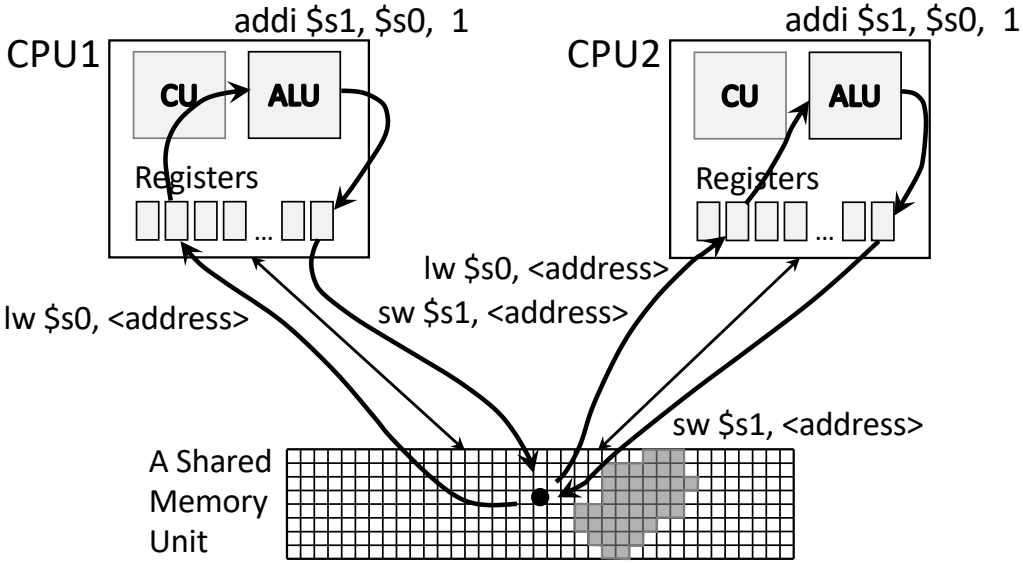
```
void thread1(unsigned *a) {
  for (unsigned i = 0; i < 100’000; i++)
       (*a)++;
  return a;
}

void thread2(unsigned *a) {
  for (unsigned i = 0; i < 100’000; i++)
       (*a)++;
  return a;
}
```

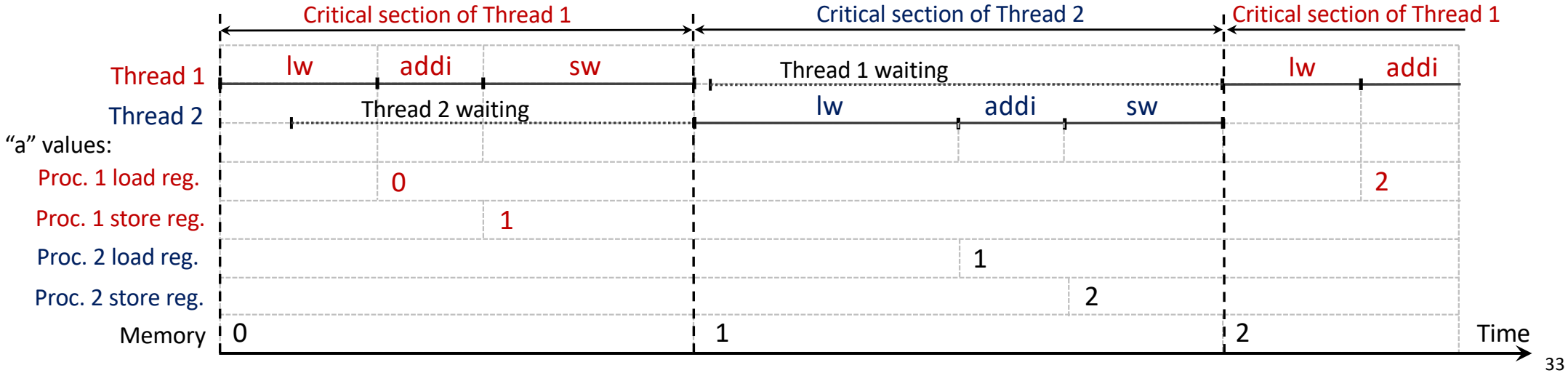


# Race Condition: Explicit Synchronization of Critical Sections

```
void thread1(unsigned *a) {
  for (unsigned i = 0; i < 100’000; i++)
       (*a)++;
  return a;
}

void thread2(unsigned *a) {
  for (unsigned i = 0; i < 100’000; i++)
       (*a)++;
  return a;
}
```



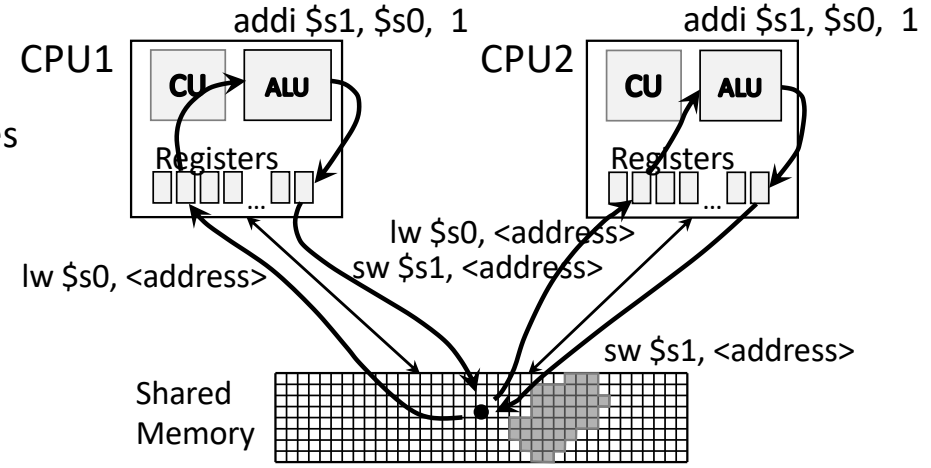



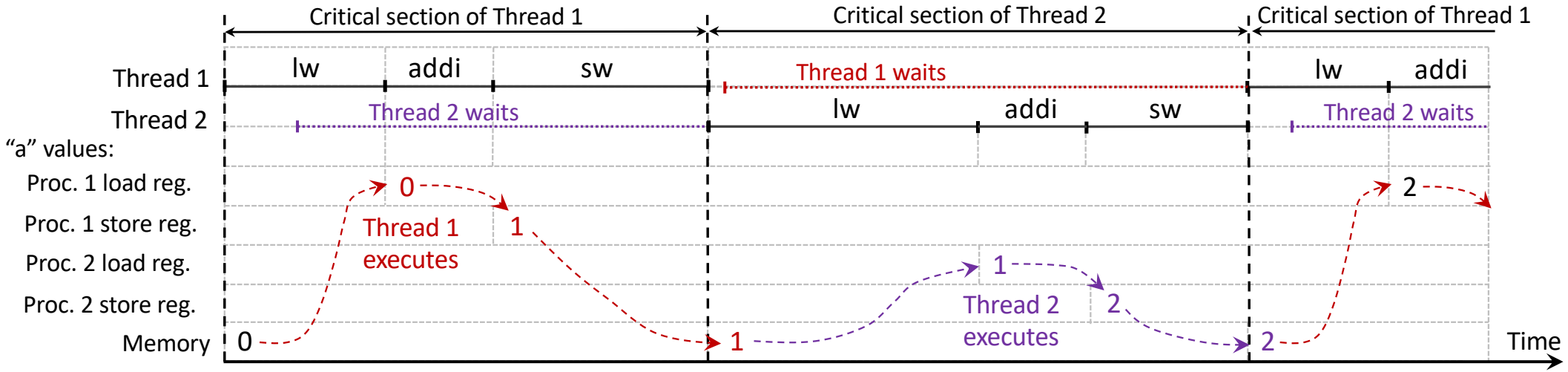
# Blocking vs. Non-Blocking Implementation

The same problem can be solved by blocking or non-blocking approaches

For example, to resolve race condition:

1) **Blocking**: critical sections to provide mutually exclusive access to a shared variable;



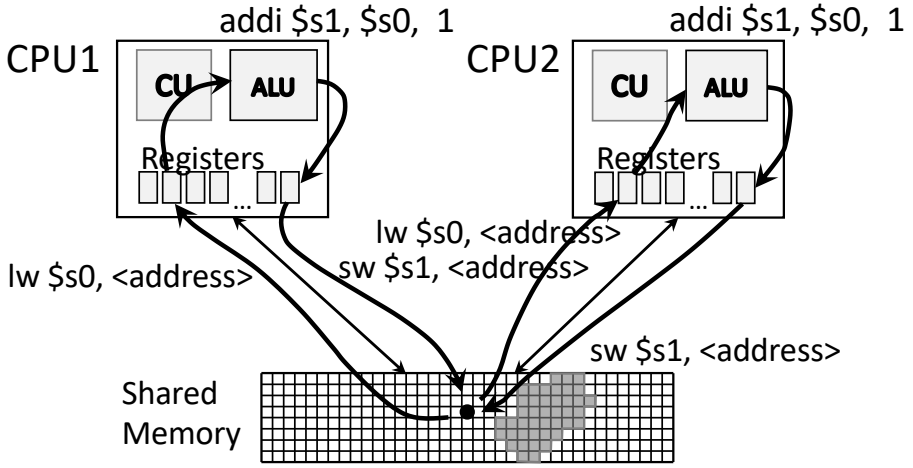



# Blocking vs. Non-Blocking Implementation

The same program can be implemented as blocking or non-blocking

For example, these are possible ways to resolve race condition:

1) **Blocking**: critical sections to provide mutually exclusive access to a shared variable;
2) **Non-blocking:** for example, by using CAS-like (“compare-and-swap”) operations; no critical sections and mutual exclusion used

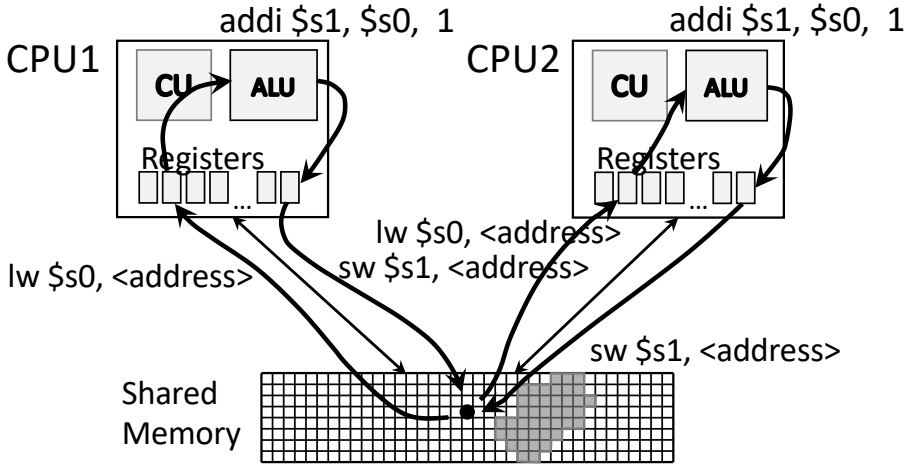


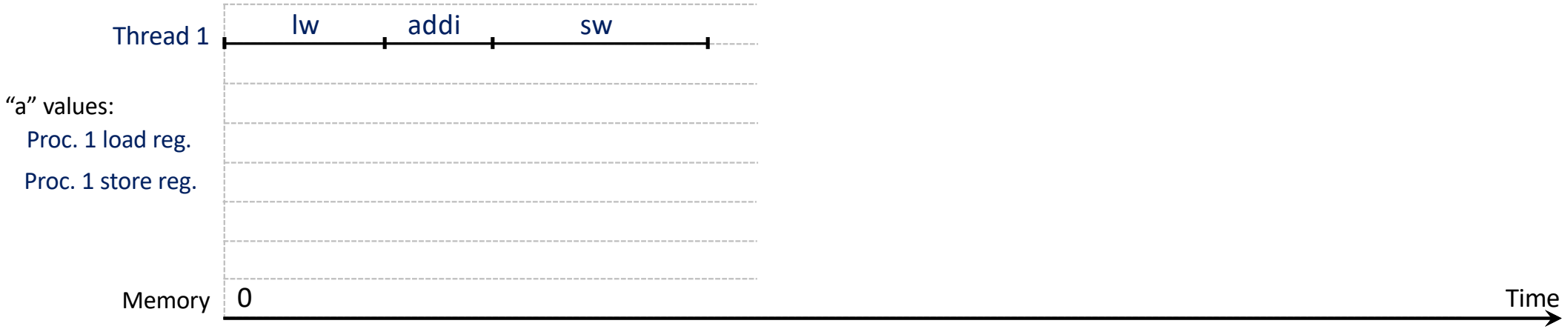
# Blocking vs. Non-Blocking Implementation

The same program can be implemented as blocking or non-blocking

For example, these are possible ways to resolve race condition:

1) **Blocking**: critical sections to provide mutually exclusive access to a shared variable;
2) **Non-blocking:** for example, by using CAS-like ("compare-and-swap") operations; no critical sections and mutual exclusion used



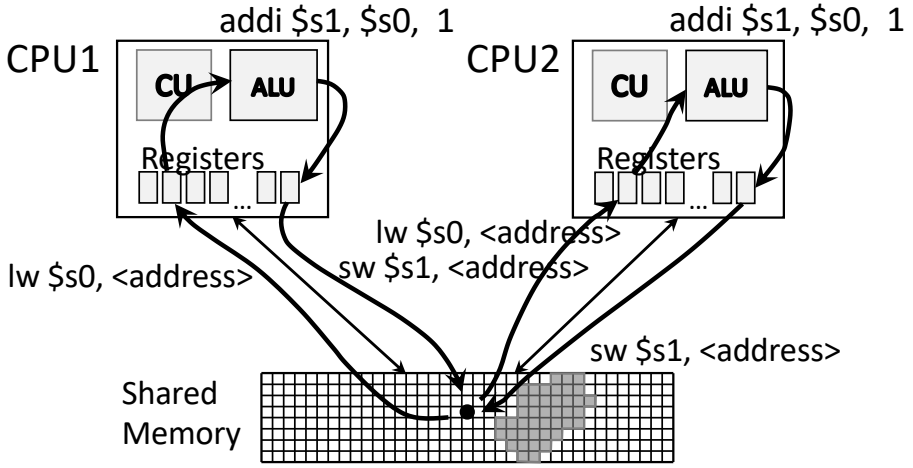



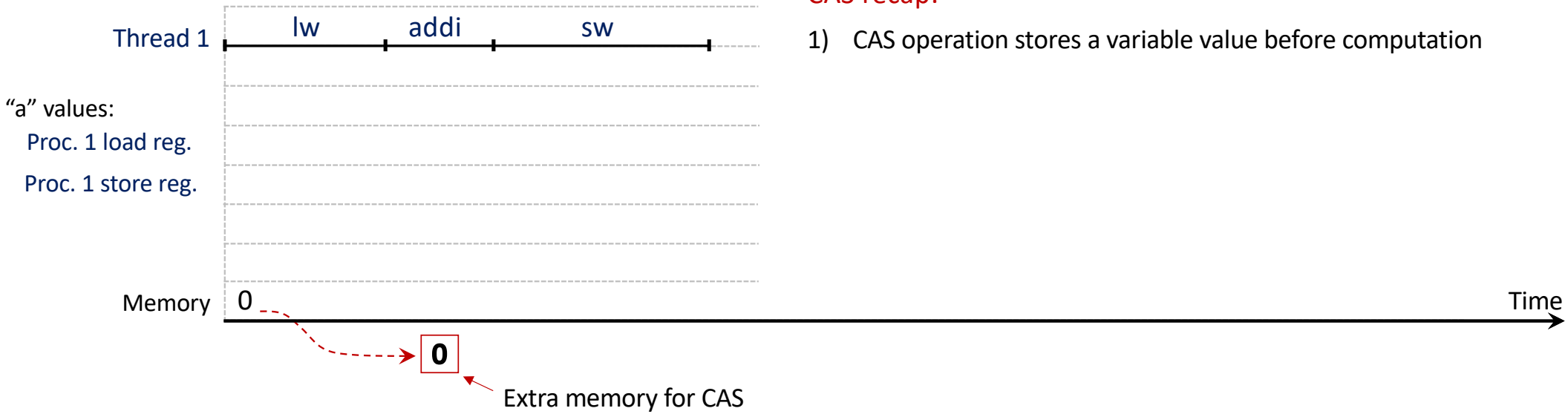
# Blocking vs. Non-Blocking Implementation

The same program can be implemented as blocking or non-blocking

For example, these are possible ways to resolve race condition:

1) **Blocking**: critical sections to provide mutually exclusive access to a shared variable;
2) **Non-blocking:** for example, by using CAS-like ("compare-and-swap") operations; no critical sections and mutual exclusion used




CAS recap:

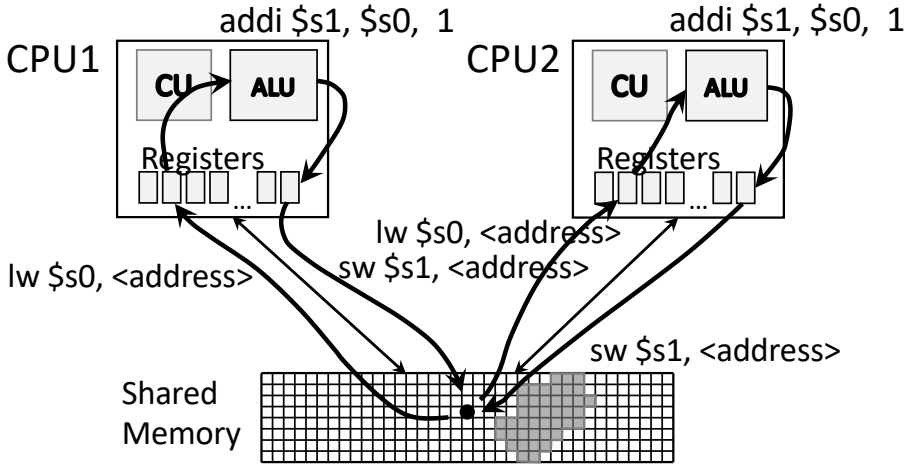
1) CAS operation stores a variable value before computation

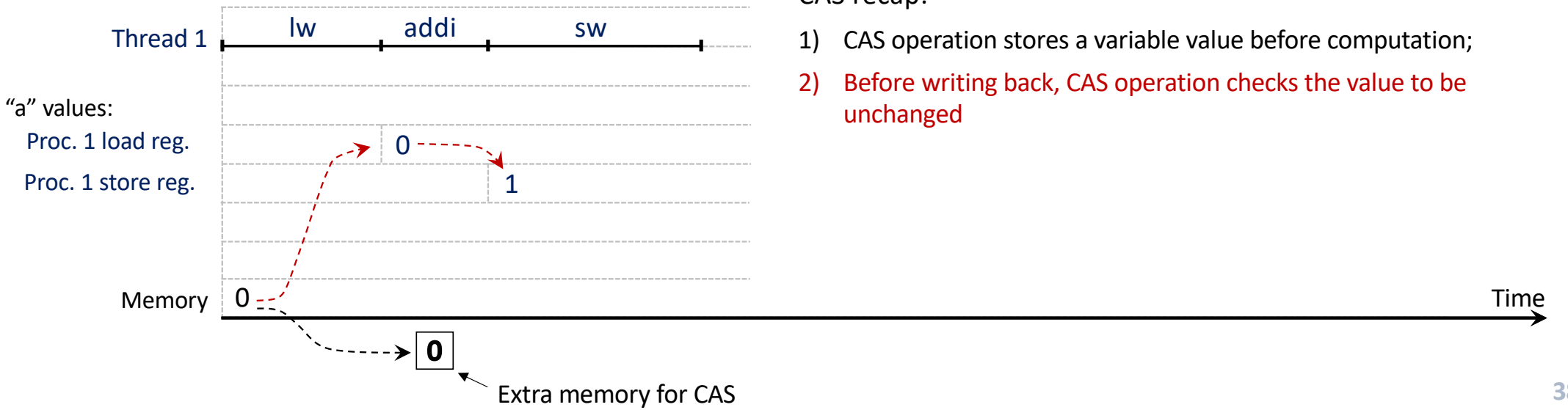
# Blocking vs. Non-Blocking Implementation

The same program can be implemented as blocking or non-blocking

For example, these are possible ways to resolve race condition:

1) **Blocking**: critical sections to provide mutually exclusive access to a shared variable;
2) **Non-blocking:** for example, by using CAS-like ("compare-and-swap") operations; no critical sections and mutual exclusion used




CAS recap:

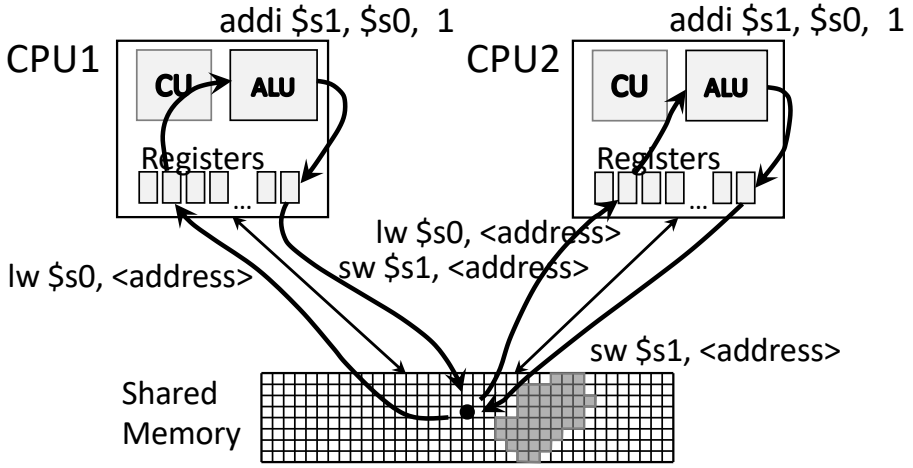
1) CAS operation stores a variable value before computation;
2) Before writing back, CAS operation checks the value to be unchanged

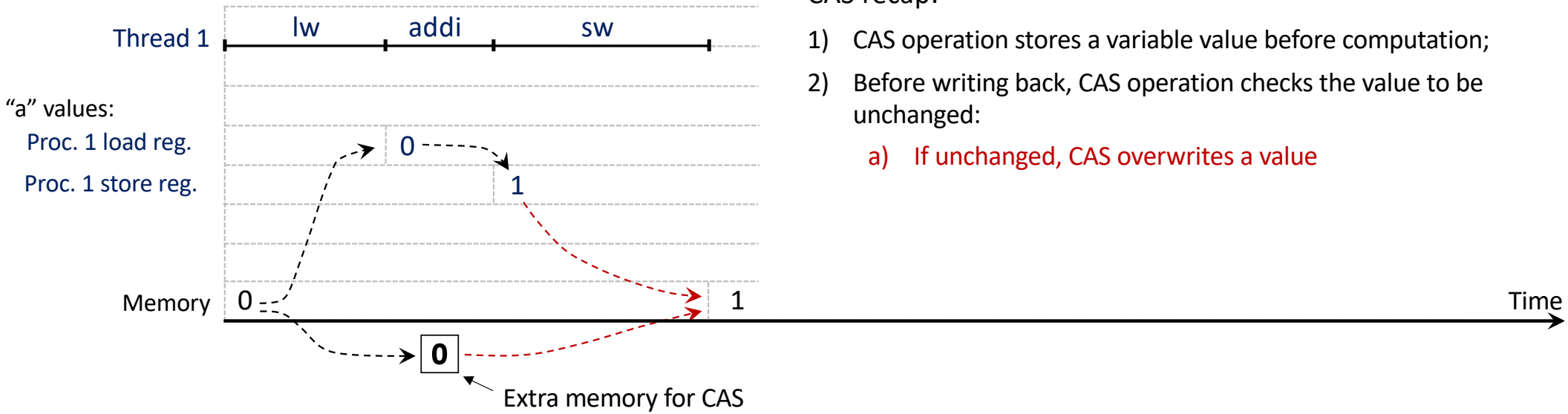
# Blocking vs. Non-Blocking Implementation

The same program can be implemented as blocking or non-blocking

For example, these are possible ways to resolve race condition:

1) **Blocking**: critical sections to provide mutually exclusive access to a shared variable;
2) **Non-blocking:** for example, by using CAS-like ("compare-and-swap") operations; no critical sections and mutual exclusion used




CAS recap:

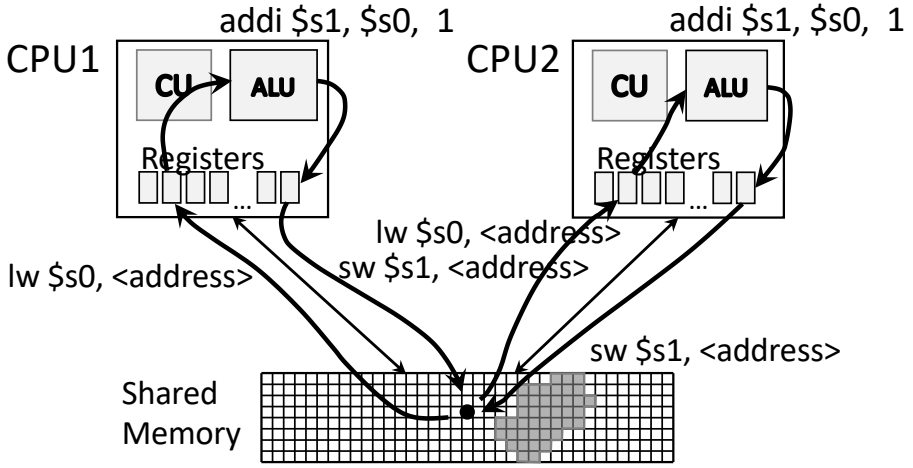
1) CAS operation stores a variable value before computation;
2) Before writing back, CAS operation checks the value to be unchanged:
   a) If unchanged, CAS overwrites a value

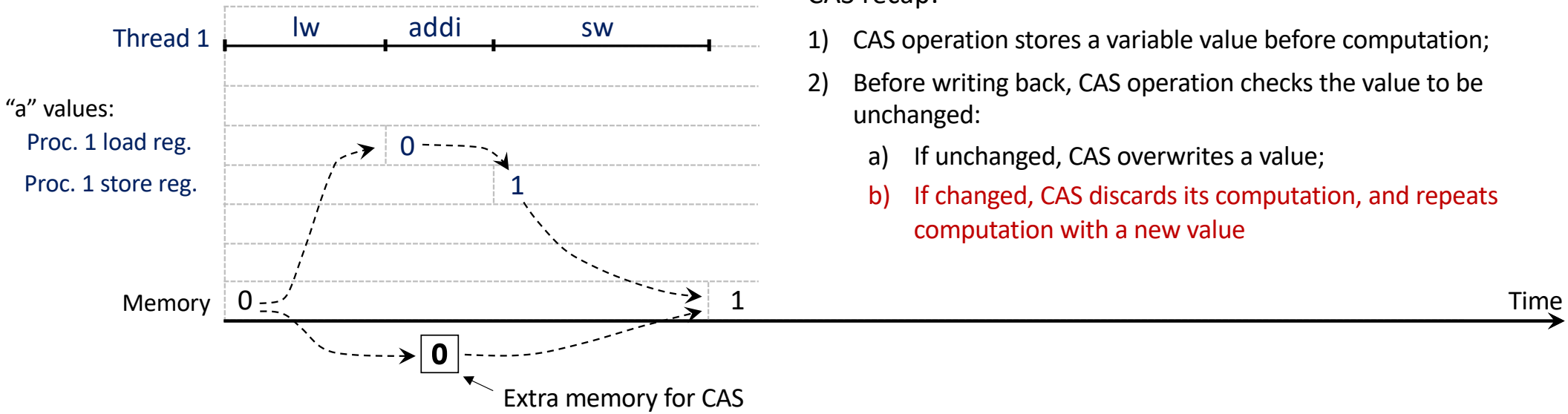
# Blocking vs. Non-Blocking Implementation

The same program can be implemented as blocking or non-blocking

For example, these are possible ways to resolve race condition:

1) **Blocking**: critical sections to provide mutually exclusive access to a shared variable;
2) **Non-blocking:** for example, by using CAS-like ("compare-and-swap") operations; no critical sections and mutual exclusion used




CAS recap:

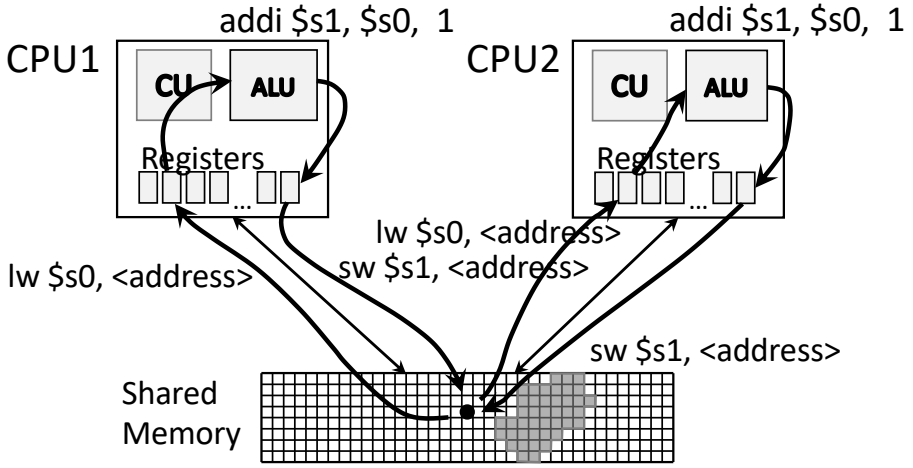
1) CAS operation stores a variable value before computation;
2) Before writing back, CAS operation checks the value to be unchanged:
   a) If unchanged, CAS overwrites a value;
   b) If changed, CAS discards its computation, and repeats computation with a new value

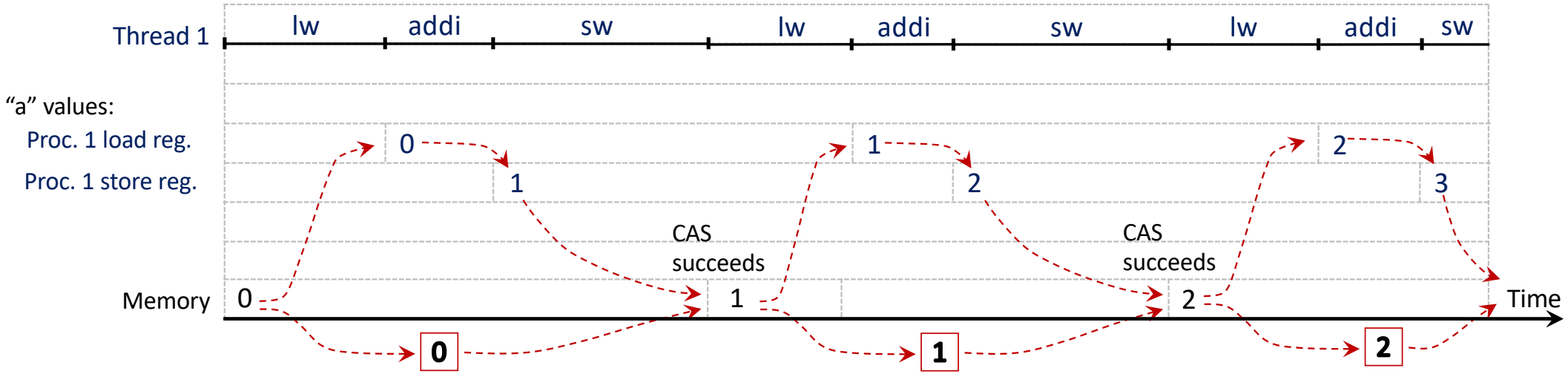
# Blocking vs. Non-Blocking Implementation

The same program can be implemented as blocking or non-blocking

For example, these are possible ways to resolve race condition:

1) **Blocking**: critical sections to provide mutually exclusive access to a shared variable;
2) **Non-blocking:** for example, by using CAS-like ("compare-and-swap") operations; no critical sections and mutual exclusion used



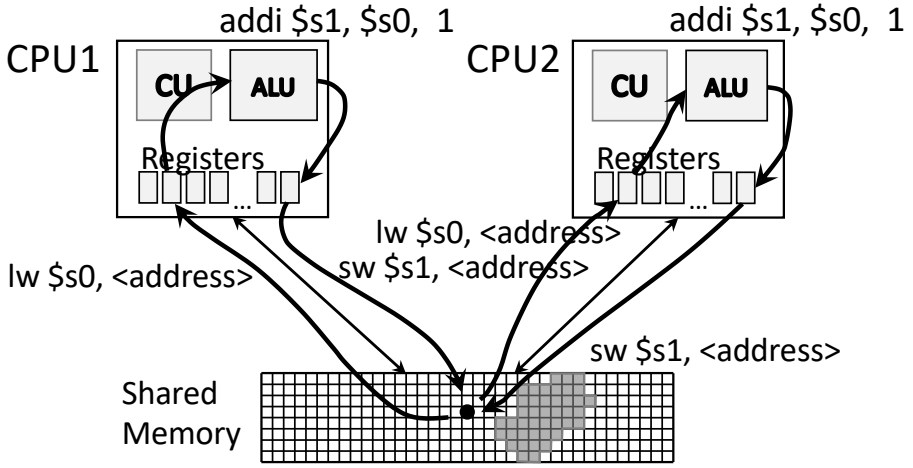



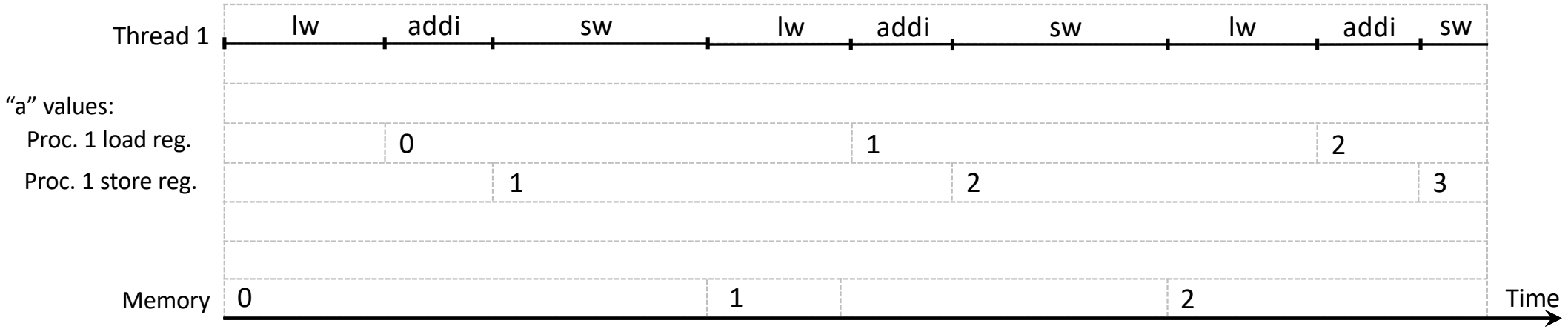
# Blocking vs. Non-Blocking Implementation

The same program can be implemented as blocking or non-blocking

For example, these are possible ways to resolve race condition:

1) **Blocking**: critical sections to provide mutually exclusive access to a shared variable;
2) **Non-blocking:** for example, by using CAS-like ("compare-and-swap") operations; no critical sections and mutual exclusion used



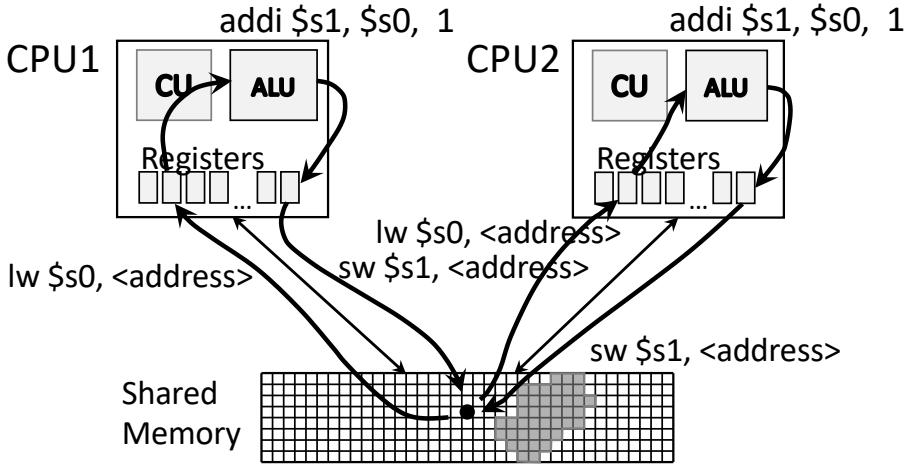



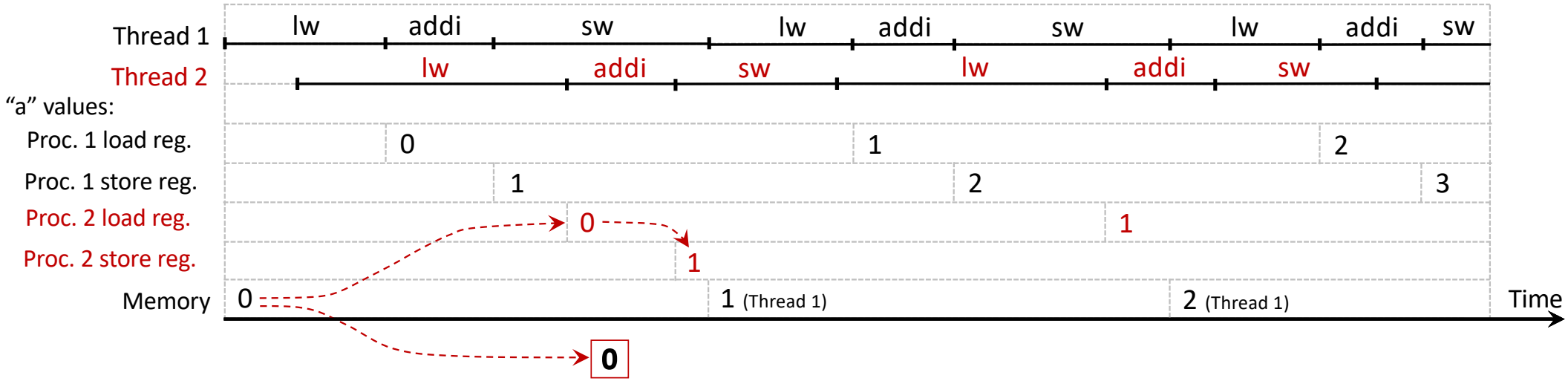
# Blocking vs. Non-Blocking Implementation

The same program can be implemented as blocking or non-blocking

For example, these are possible ways to resolve race condition:

1) **Blocking**: critical sections to provide mutually exclusive access to a shared variable;
2) **Non-blocking:** for example, by using CAS-like ("compare-and-swap") operations; no critical sections and mutual exclusion used



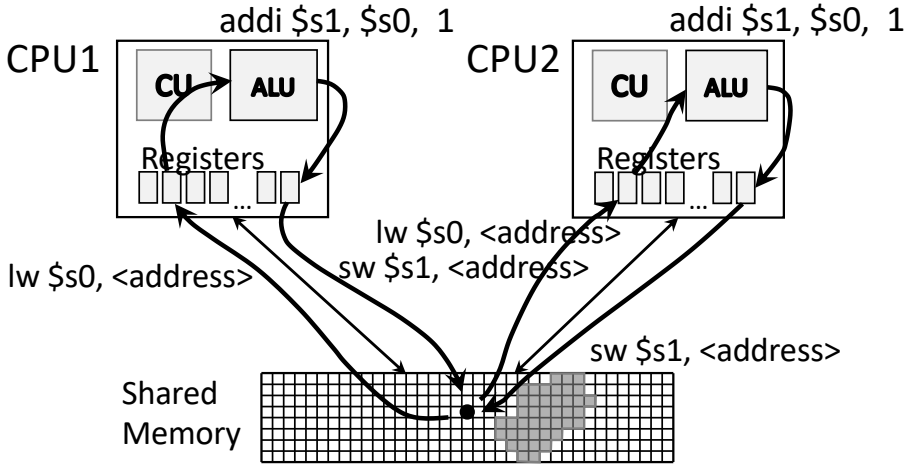



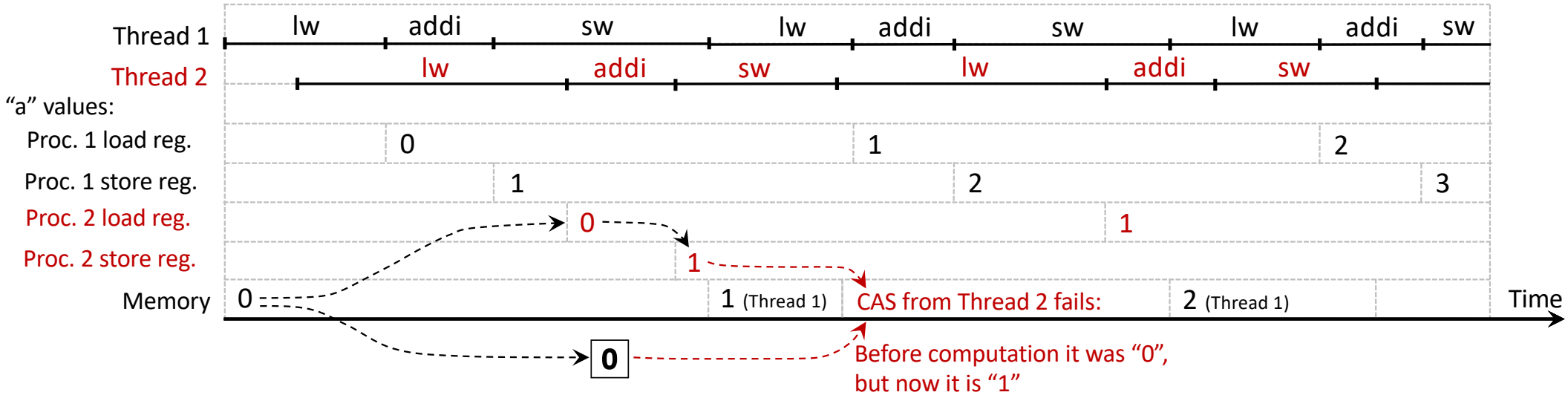
# Blocking vs. Non-Blocking Implementation

The same program can be implemented as blocking or non-blocking

For example, these are possible ways to resolve race condition:

1) **Blocking**: critical sections to provide mutually exclusive access to a shared variable;
2) **Non-blocking:** for example, by using CAS-like ("compare-and-swap") operations; no critical sections and mutual exclusion used



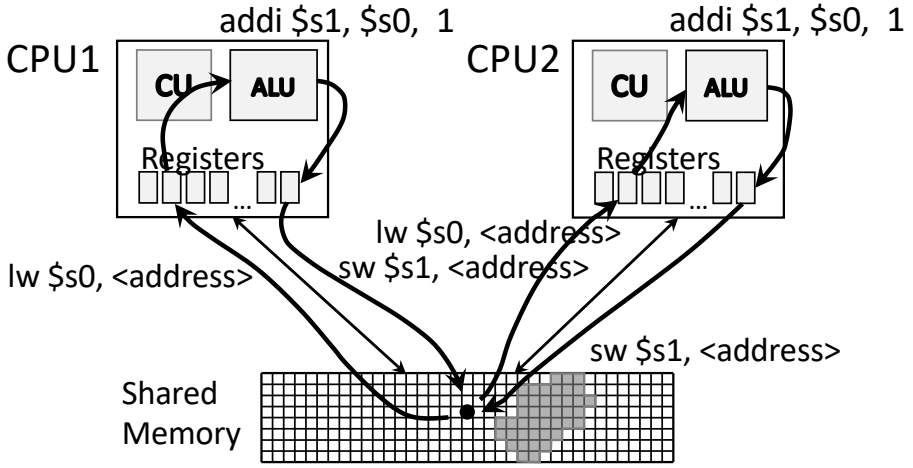



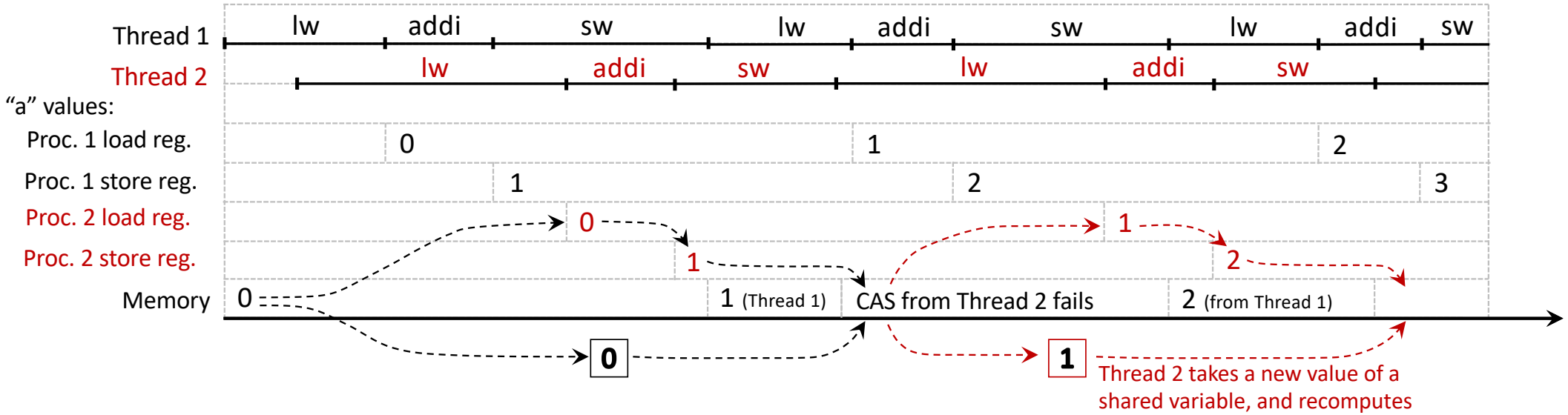
# Blocking vs. Non-Blocking Implementation

The same program can be implemented as blocking or non-blocking

For example, these are possible ways to resolve race condition:

1) **Blocking**: critical sections to provide mutually exclusive access to a shared variable;
2) **Non-blocking:** for example, by using CAS-like ("compare-and-swap") operations; no critical sections and mutual exclusion used



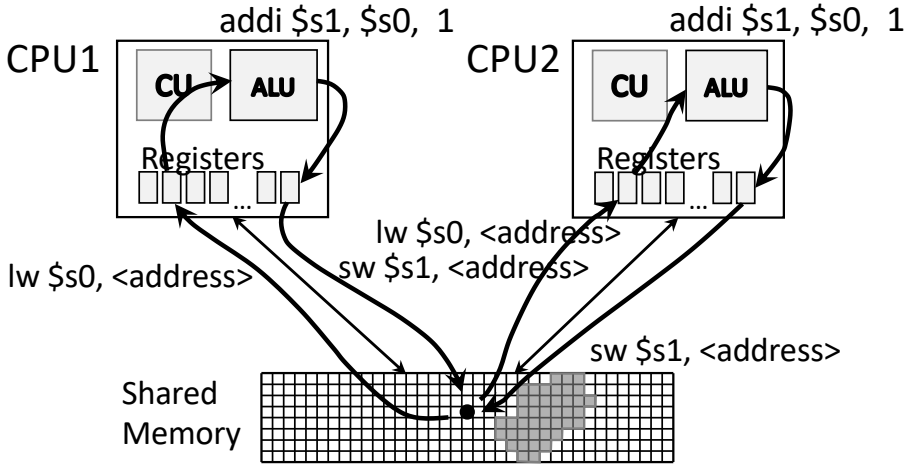



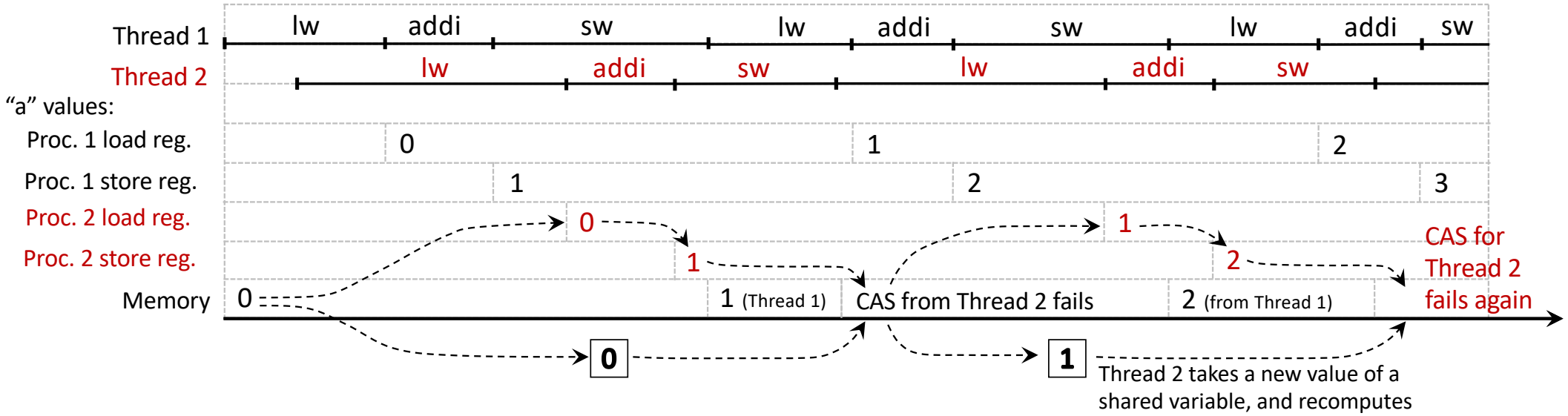
# Blocking vs. Non-Blocking Implementation

The same program can be implemented as blocking or non-blocking

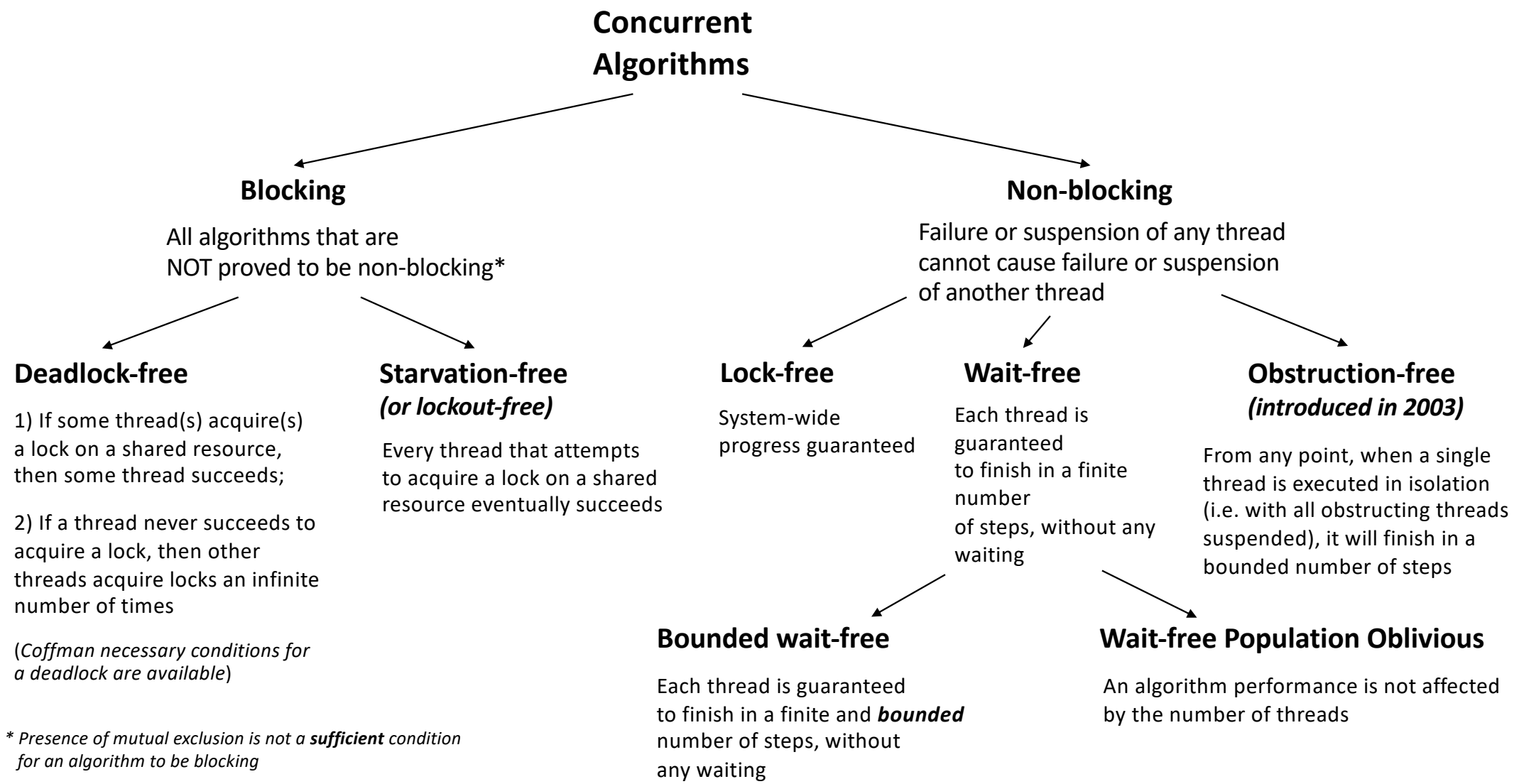
For example, these are possible ways to resolve race condition:

1) **Blocking**: critical sections to provide mutually exclusive access to a shared variable;
2) **Non-blocking:** for example, by using CAS-like ("compare-and-swap") operations; no critical sections and mutual exclusion used



Classification of Concurrent Algorithms




** Presence of mutual exclusion is not a **sufficient** condition for an algorithm to be blocking*

*Note: Livelock problem can occur for both, blocking and non-blocking algorithms*

## Threads Synchronization by Using a FIFO queue

Given:

- 2 asynchronous threads, and
- a FIFO queue, denoted by Q, that provides the following methods:

  **enqueue**( ), **dequeue**( ), **is_empty**()

## Threads Synchronization by Using a FIFO queue

Given:

- 2 asynchronous threads, and
- a FIFO queue, denoted by Q, that provides the following methods:

  **enqueue**( ), **dequeue**( ), **is_empty**()

**A sample consensus protocol:**

| Thread 1 | |
|---|---|
| 1 | bool proposed = 0; |
| 2 | bool decision = proposed; |
| 3 | if (Q.**is_empty**()) Q.**enqueue**(proposed); |
| 4 | else decision = Q.**dequeue**(); |
| 5 | return decision; |

| Thread 2 | |
|---|---|
| 1 | bool proposed = 1; |
| 2 | bool decision = proposed; |
| 3 | if (Q.**is_empty**()) Q.**enqueue**(proposed); |
| 4 | else decision = Q.**dequeue**(); |
| 5 | return decision; |

*Assumption: for simplicity of explanation, code in line 3 is assumed to execute atomically*

## Threads Synchronization by Using a FIFO queue

Given:

- 2 asynchronous threads, and
- a FIFO queue, denoted by Q, that provides the following methods:

  enqueue( ), dequeue( ), is_empty()

**A sample consensus protocol:**

| Thread 1 | |
|---|---|
| 1 | bool **proposed = 0**; |
| 2 | bool decision = proposed; |
| 3 | if (Q.is_empty()) Q.enqueue(proposed); |
| 4 | else decision = Q.dequeue(); |
| 5 | return **decision**; |

| Thread 2 | |
|---|---|
| 1 | bool **proposed = 1**; |
| 2 | bool decision = proposed; |
| 3 | if (Q.is_empty()) Q.enqueue(proposed); |
| 4 | else decision = Q.dequeue(); |
| 5 | return **decision**; |

*Assumption: for simplicity of explanation, code in line 3 is assumed to execute atomically*

## One Feasible Relative Execution Scenario

| Linearization Points | Thread 1 | Thread 2 |
|---|---|---|
| 1 | bool proposed = 0; | |
| 2 | bool decision = proposed; | |
| 3 | if (Q.**is_empty**())  Q.**enqueue**(proposed); | |
| 4 | else decision = Q.**dequeue**(); | |
| 5 | return decision; | |
| 6 | | bool proposed = 1; |
| 7 | | bool decision = proposed; |
| 8 | | if (Q.**is_empty**())  Q.**enqueue**(proposed); |
| 9 | | else decision = Q.**dequeue**(); |
| 10 | | return decision; |

Observation:

Both threads always return the same “decision” value, despite being asynchronous and having different “proposed” values!

## Another Feasible Relative Execution Scenario

| Linearization Points | Thread 1 | Thread 2 |
|---|---|---|
| 1 | bool proposed = 0; | |
| 2 | bool decision = proposed; | |
| **3** | | bool proposed = 1; |
| **4** | | bool decision = proposed; |
| **5** | | if (Q.**is_empty**()) Q.**enqueue**(proposed); |
| **6** | if (Q.**is_empty**()) Q.**enqueue**(proposed); | |
| **7** | else decision = Q.**dequeue**(); | |
| **8** | return decision; | |
| **9** | | else decision = Q.**dequeue**(); |
| **10** | | return decision; |

Observation:

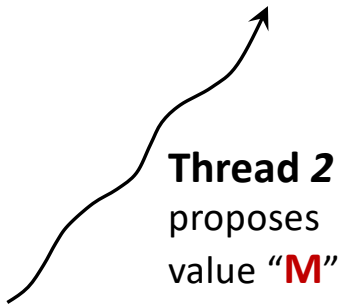
Both threads always return the same “decision” value, despite being asynchronous and having different “proposed” values!

## Consensus Problem: An Informal Introduction

- We have *n* asynchronous threads;

## Consensus Problem: An Informal Introduction

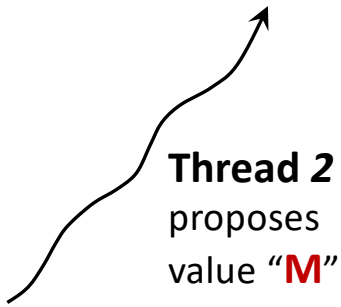
- We have *n* asynchronous threads;
- Every thread, at some point of its execution, proposes a value (e.g. a local clock time), for example, by placing it into a buffer shared among all threads;



## Consensus Problem: An Informal Introduction

- We have *n* asynchronous threads;
- Every thread, at some point of its execution, proposes a value (e.g. a local clock time), for example, by placing it into a buffer shared among all threads;

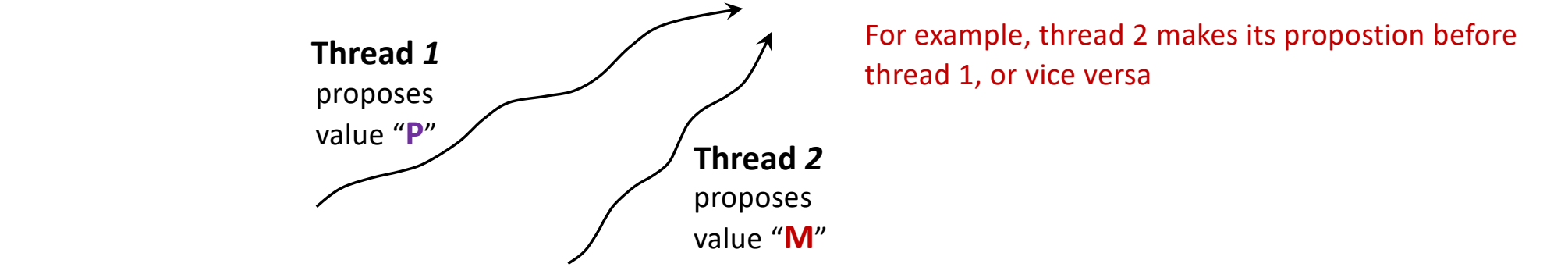
Some data structure(s) and synchronization primitive(s) used



## Consensus Problem: An Informal Introduction

- We have *n* asynchronous threads;
- Every thread, at some point of its execution, proposes a value (e.g. a local clock time), for example, by placing it into a buffer shared among all threads;
- Propositions by threads are done at a completely random (unpredictable) order;

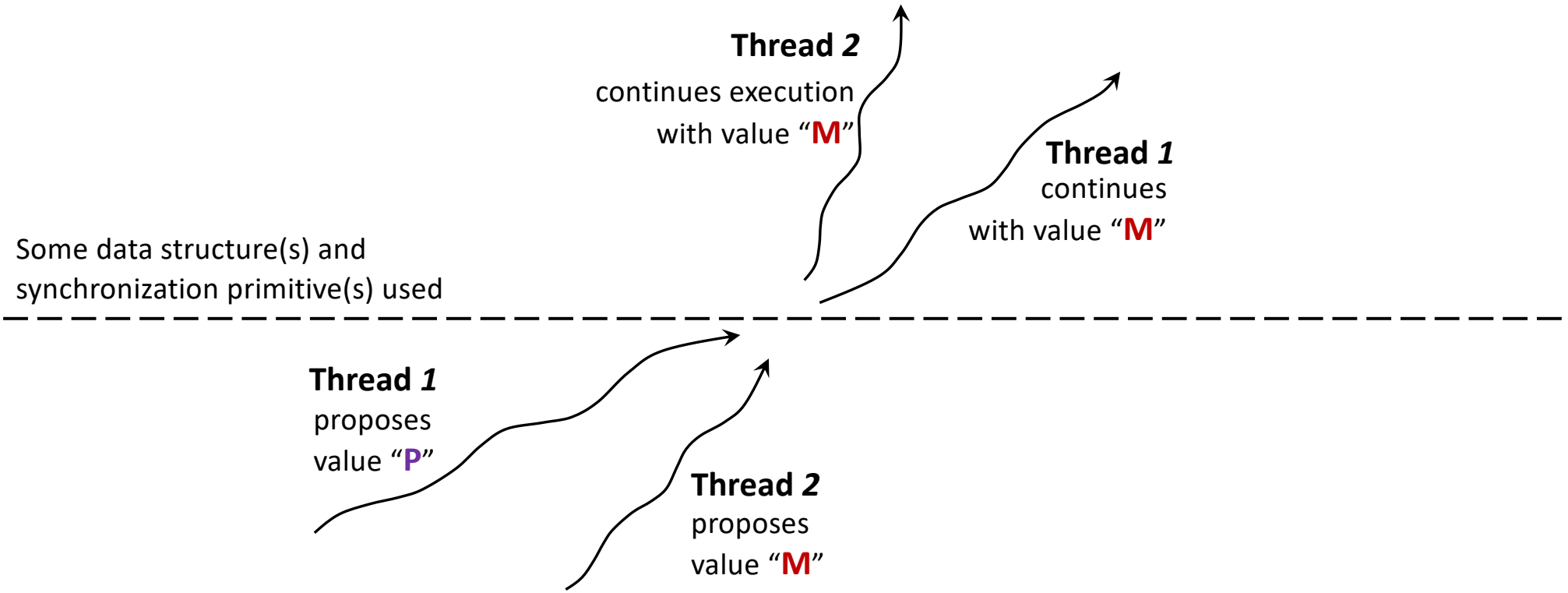
Some data structure(s) and synchronization primitive(s) used




For example, thread 2 makes its propostion before thread 1, or vice versa

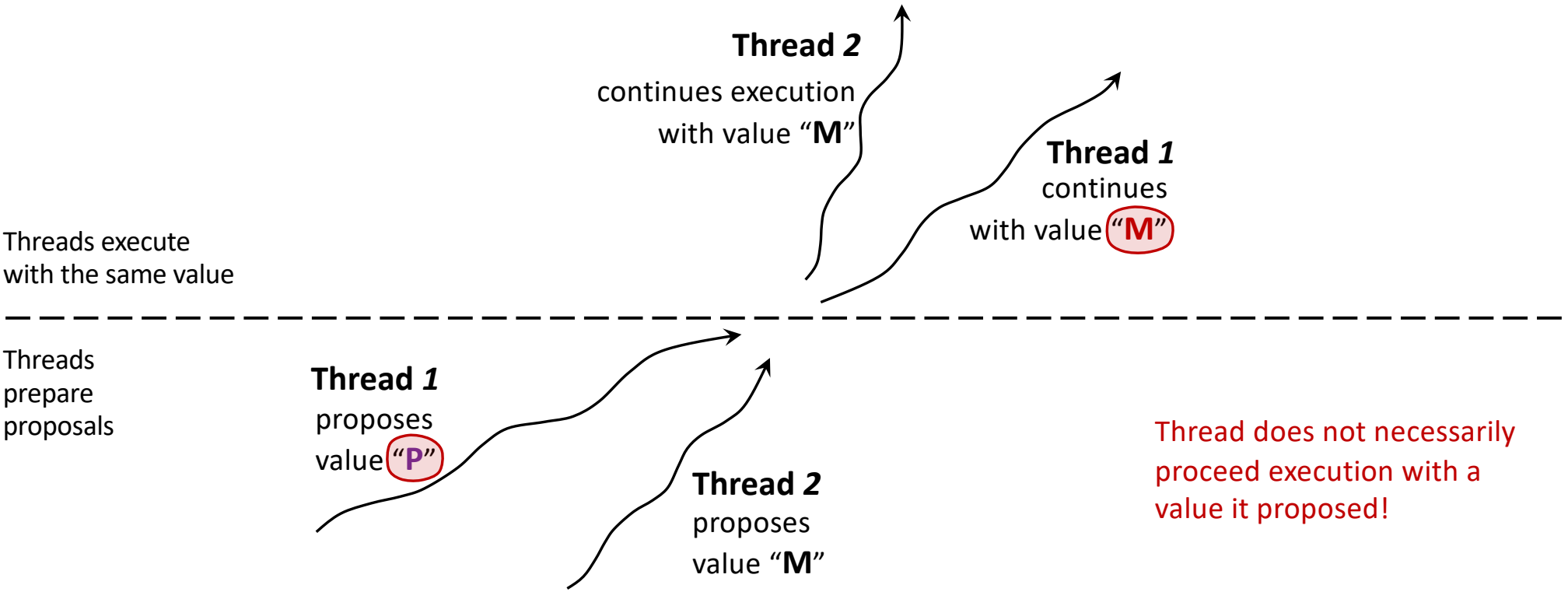
## Consensus Problem: An Informal Introduction

- We have *n* asynchronous threads;
- Every thread, at some point of its execution, proposes a value (e.g. a local clock time), for example, by placing it into a buffer shared among all threads;
- Propositions are done at a completely random (unpredictable) order;
- After that point, every thread continues execution with a value, that is the same for all other threads (both, which have already made propositions, or are going to do so)



## Consensus Problem: An Informal Introduction

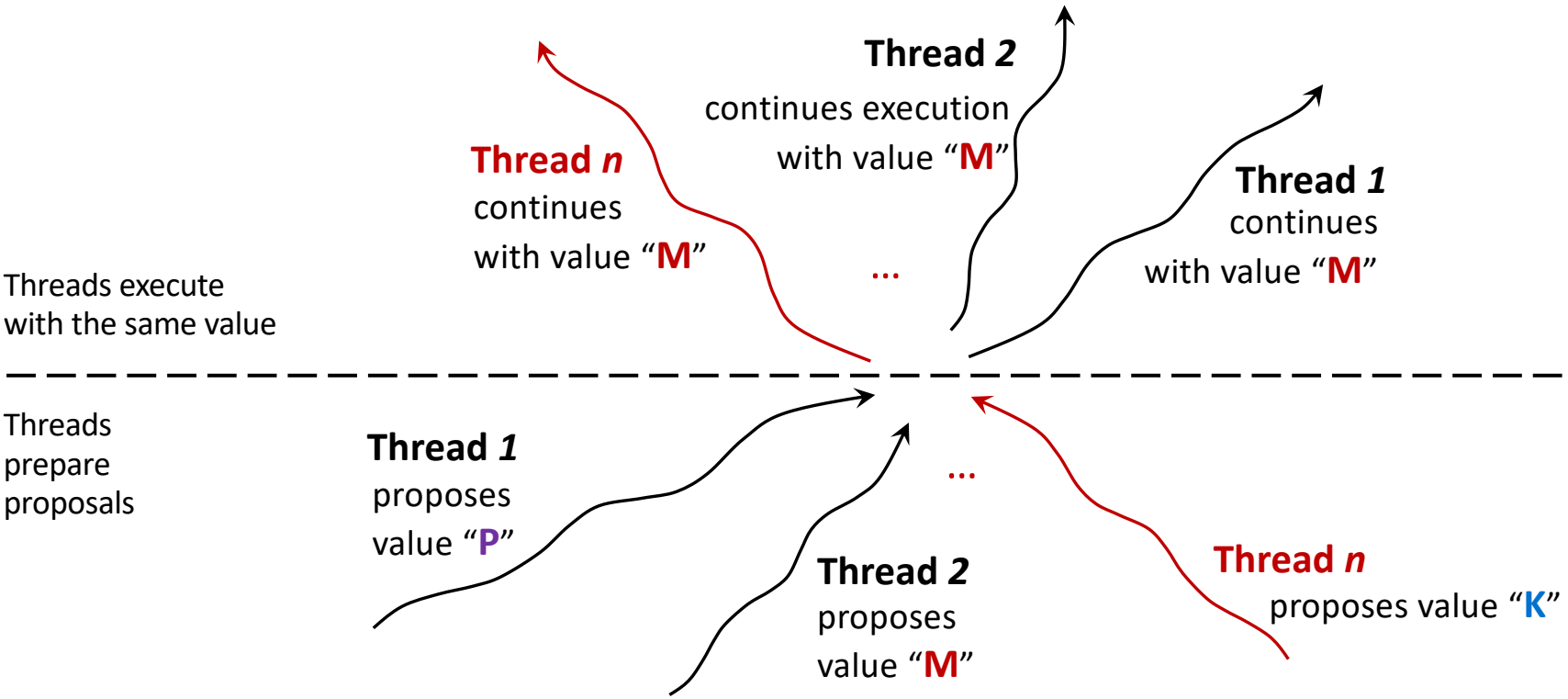
- We have *n* asynchronous threads;
- Every thread, at some point of its execution, proposes a value (e.g. a local clock time), for example, by placing it into some shared buffer;
- Propositions are done at a completely random (unpredictable) order;
- After that point, every thread continues execution with a value, that is the same for all other threads (both, which have already made propositions, or are going to do so)




Thread does not necessarily proceed execution with a value it proposed!

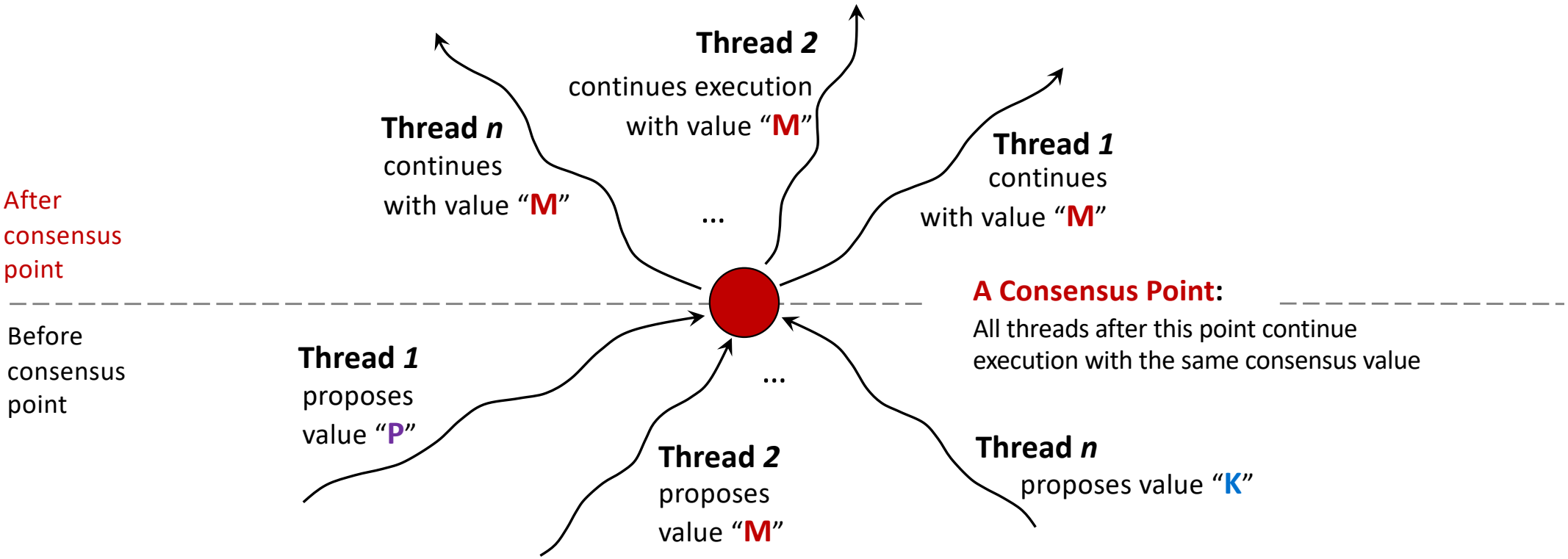
## Consensus Problem: An Informal Introduction

- We have *n* asynchronous threads;
- Every thread, at some point of its execution, proposes a value (e.g. a local clock time), for example, by placing it into some shared buffer;
- Propositions are done at a completely random (unpredictable) order;
- After that point, every thread continues execution with a value, that is the same for all other threads (both, which have already made propositions, or are going to do so)



# Consensus Problem: An Informal Introduction

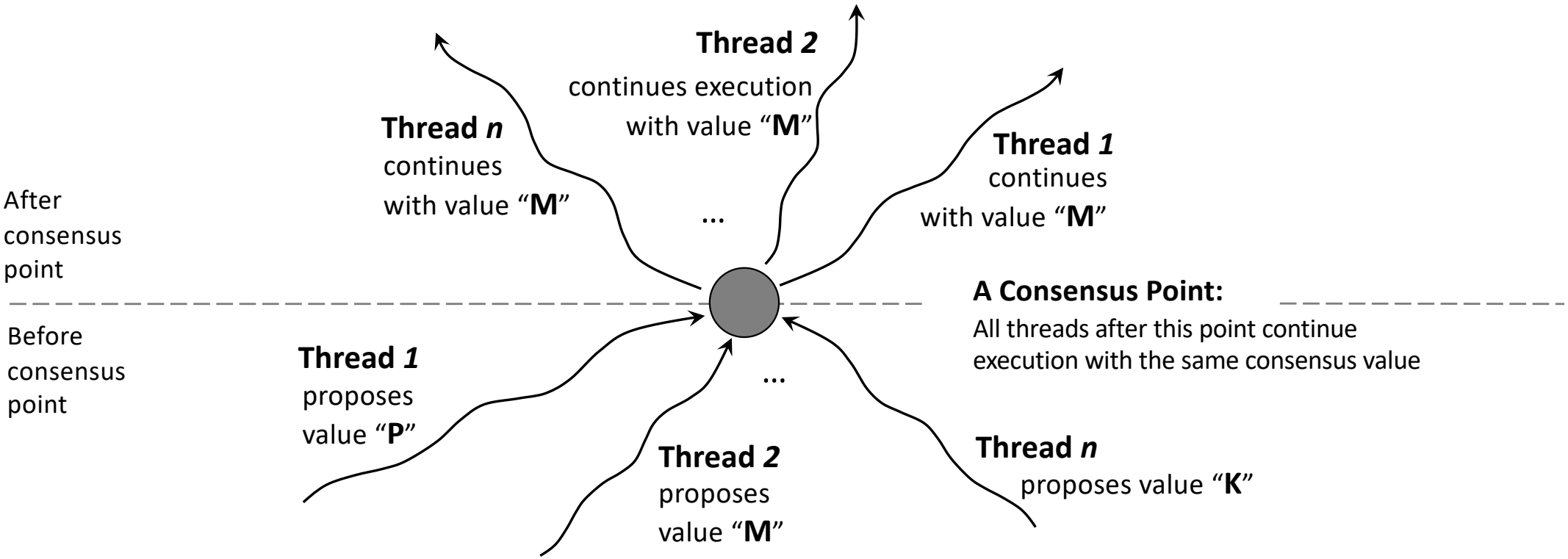
- We have *n* asynchronous threads;
- Every thread, at some point of its execution, proposes a value (e.g. a local clock time), for example, by placing it into some shared buffer;
- Propositions are done at a completely random (unpredictable) order;
- After that point, every thread continues execution with a value, that is the same for all other threads (both, which have already made propositions, or are going to do so)



# Synchronous and Asynchronous Consensus Problems

Two major types of consensus problems:

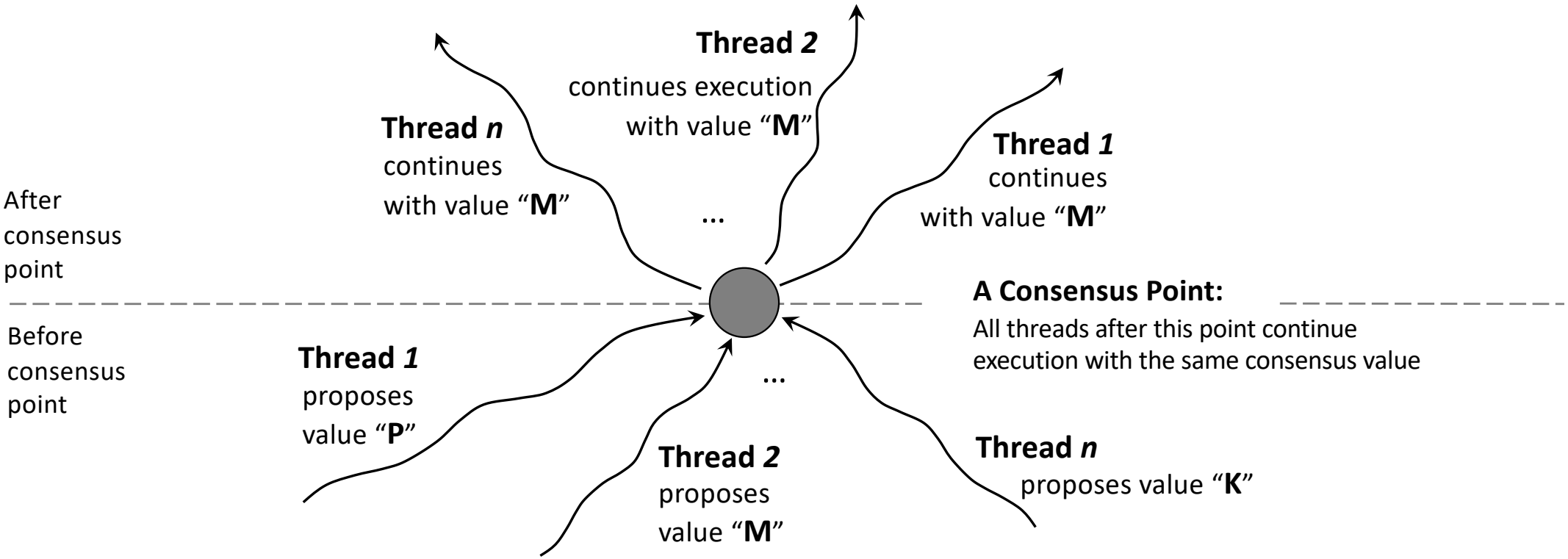
- **Asynchronous**: threads do not wait for each other (a wait-freedom requirement);
- **Synchronous**: When reaching a consensus point, threads wait for each-other, before making a consensus decision (threads blocking at a consensus point takes place)



# Synchronous and Asynchronous Consensus Problems

Two major types of consensus problems:

- **Asynchronous**: threads do not wait for each other (a wait-freedom requirement);
- **Synchronous**: When reaching a consensus point, threads wait for each-other, before making a consensus decision (threads blocking at a consensus point takes place)

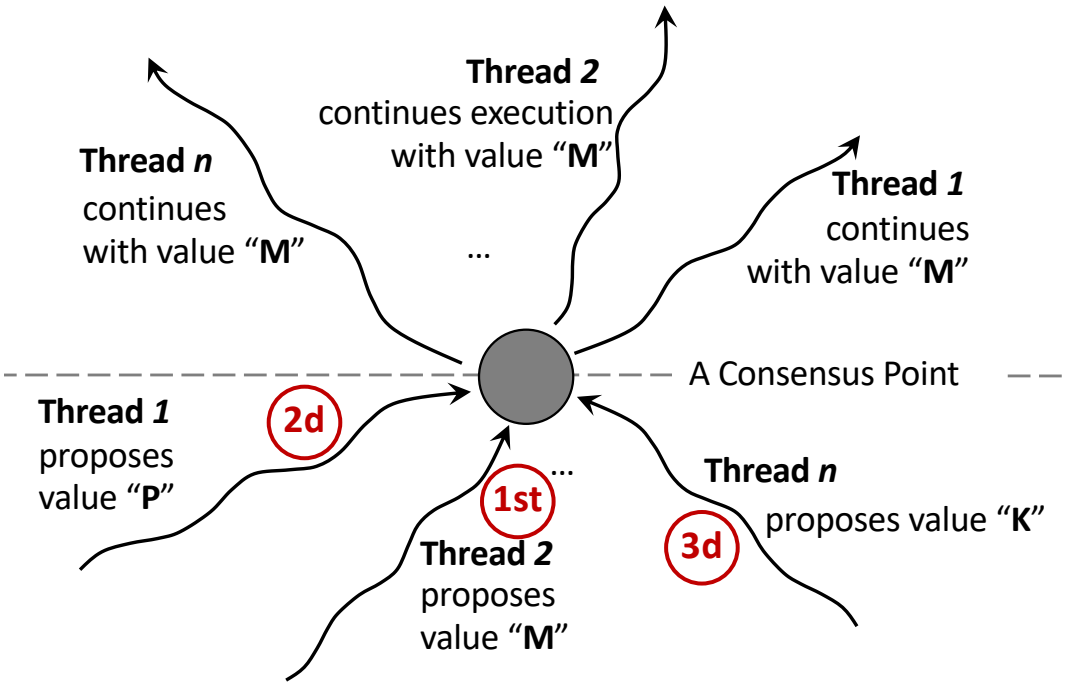



Unless stated otherwise, we assume an asynchronous consensus problem (threads are not synchronized)

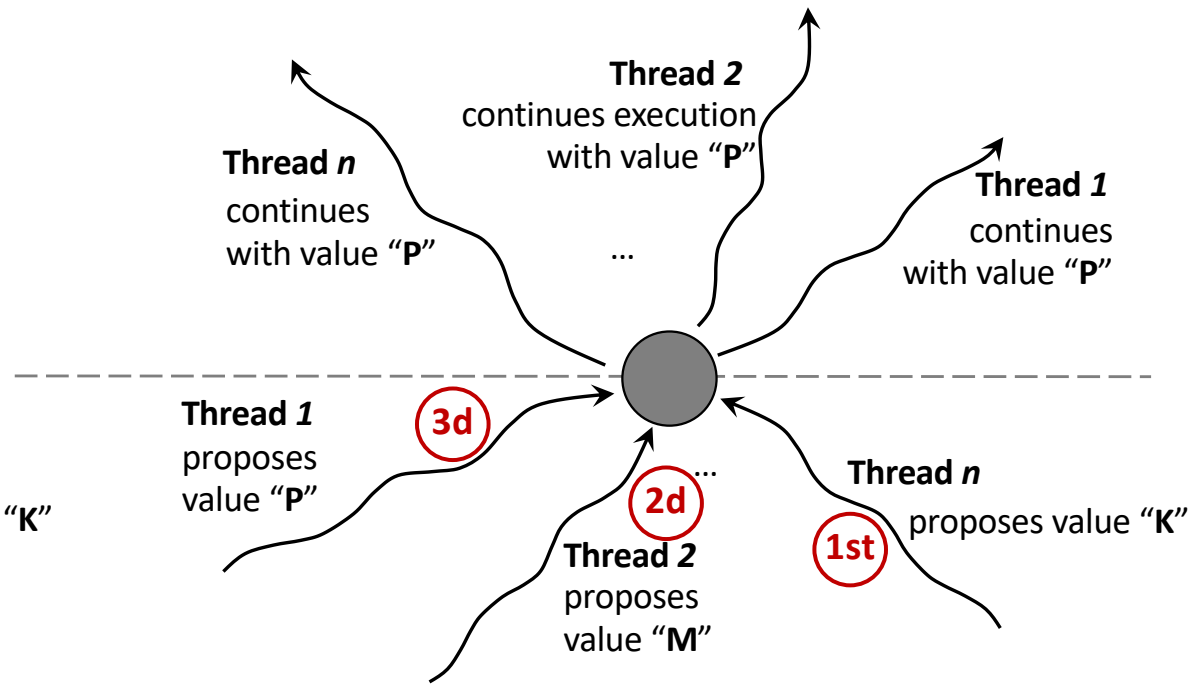
# An Asynchronous Consensus Problem: An Informal Introduction

The key difficulty is that threads make propositions at an unpredictable order

One feasible scenario:



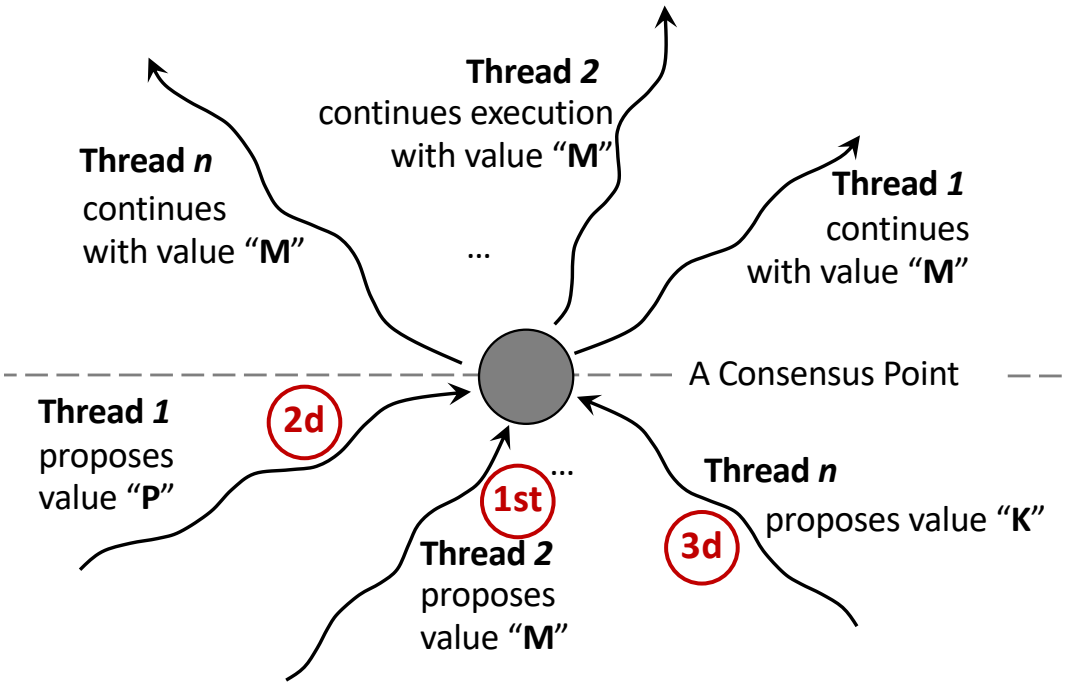

Another feasible scenario:



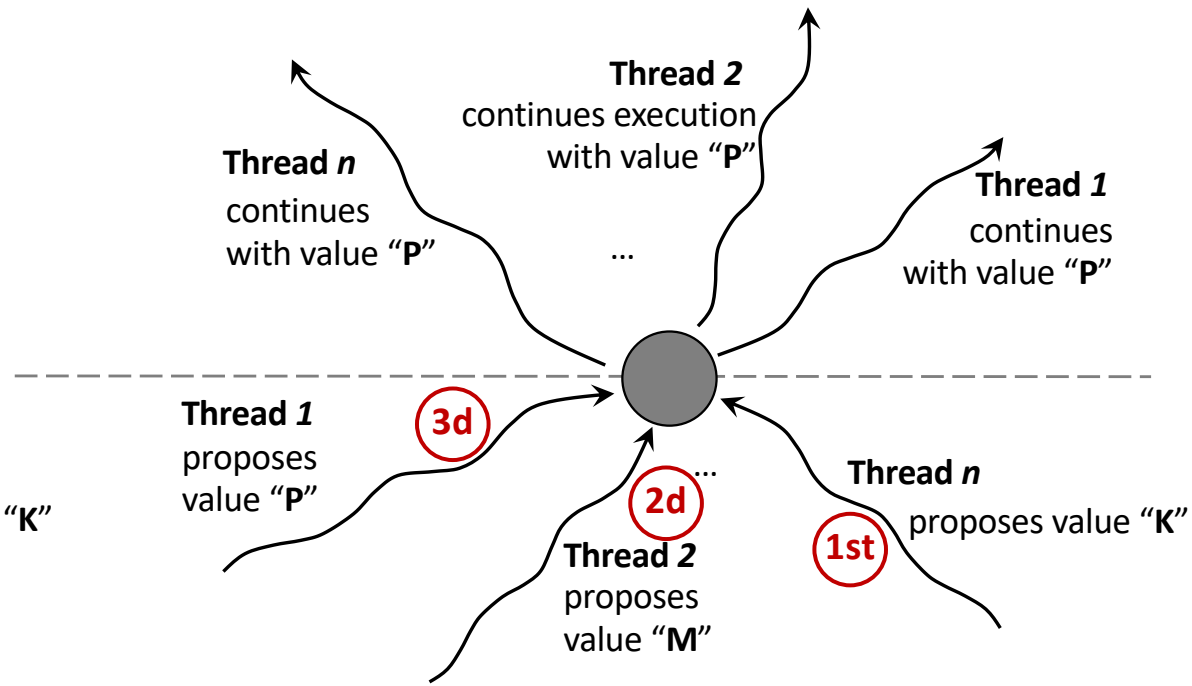
# An Asynchronous Consensus Problem: An Informal Introduction

The key difficulty is that threads make propositions at an unpredictable order

One feasible scenario:




Another feasible scenario:

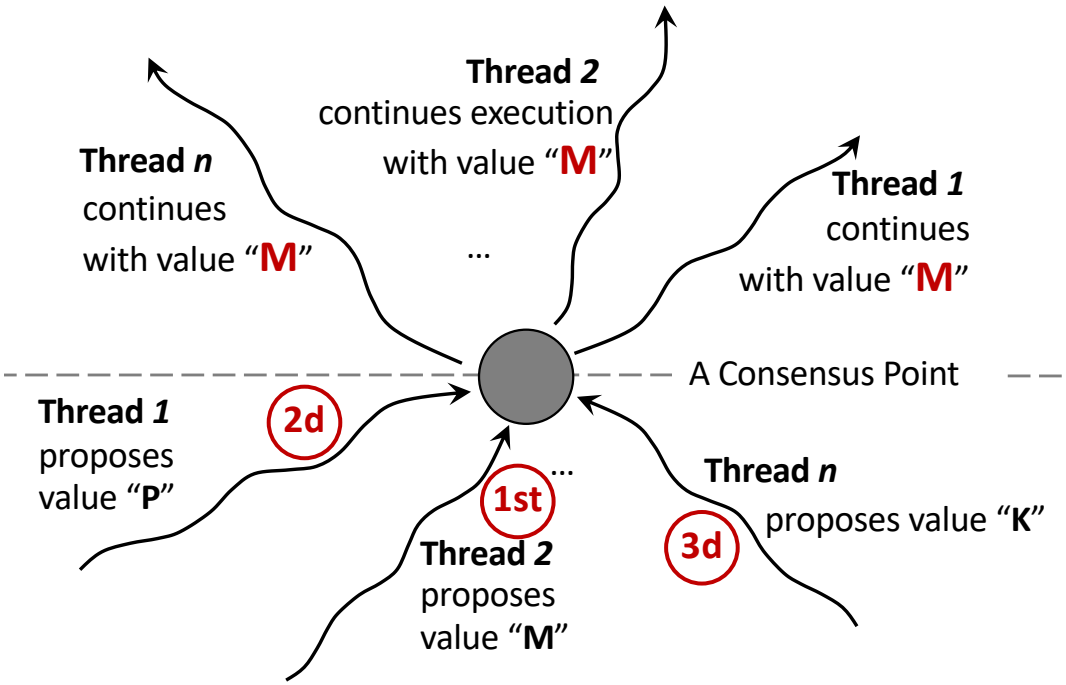



The order of threads proposition typically affects a consensus value

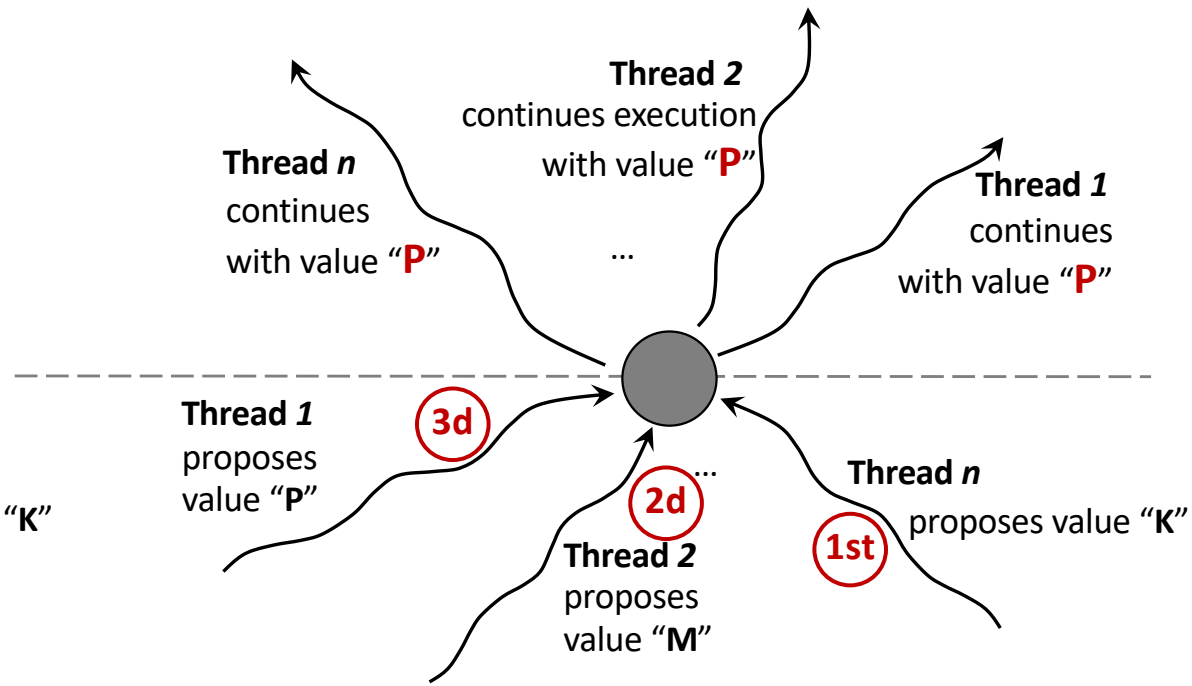
# An Asynchronous Consensus Problem: An Informal Introduction

The key difficulty is that threads make propositions at an unpredictable order

One feasible scenario:




Another feasible scenario:

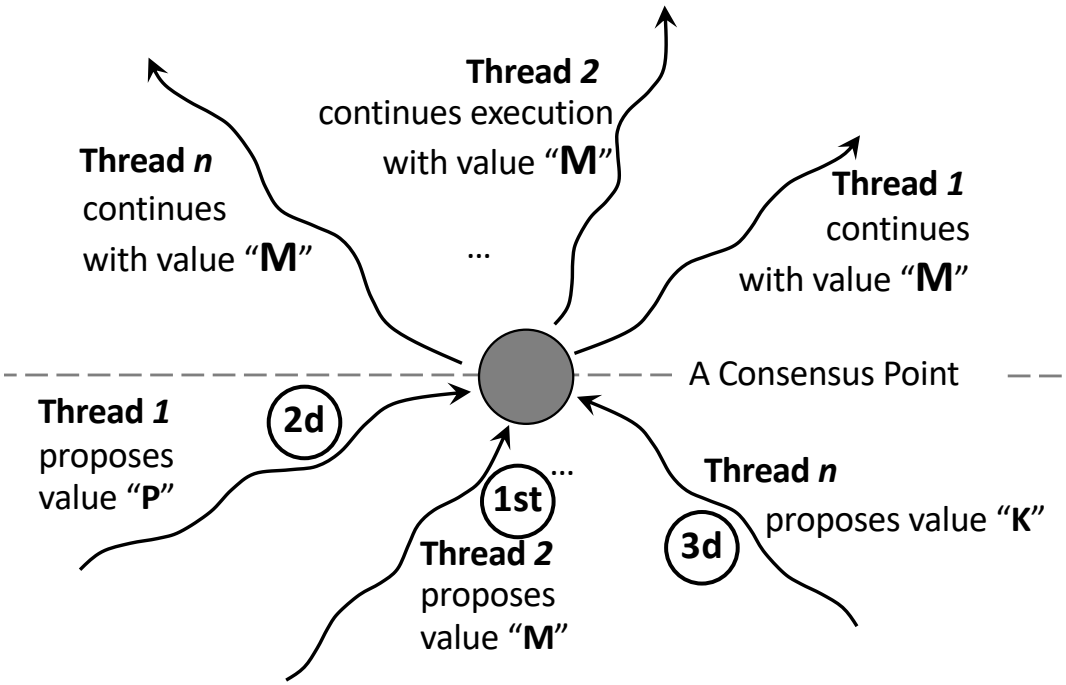



The order of threads proposition typically affects a consensus value

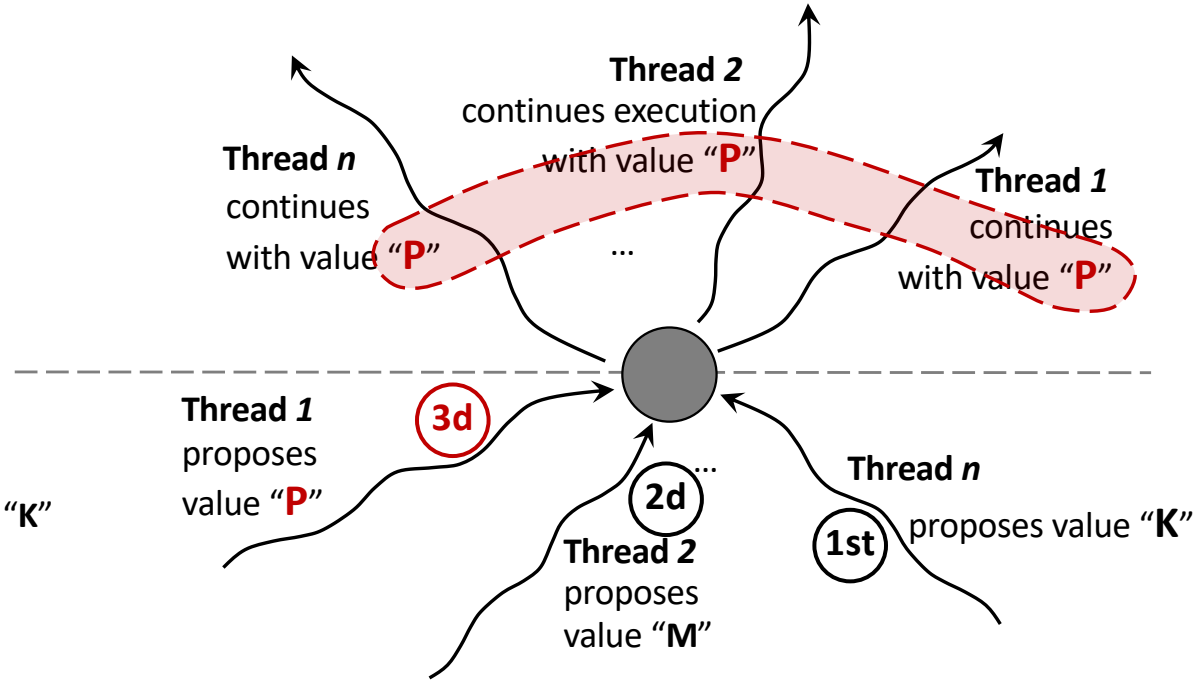
# An Asynchronous Consensus Problem: An Informal Introduction

The key difficulty is that threads make propositions at an unpredictable order

One feasible scenario:




Another feasible scenario:



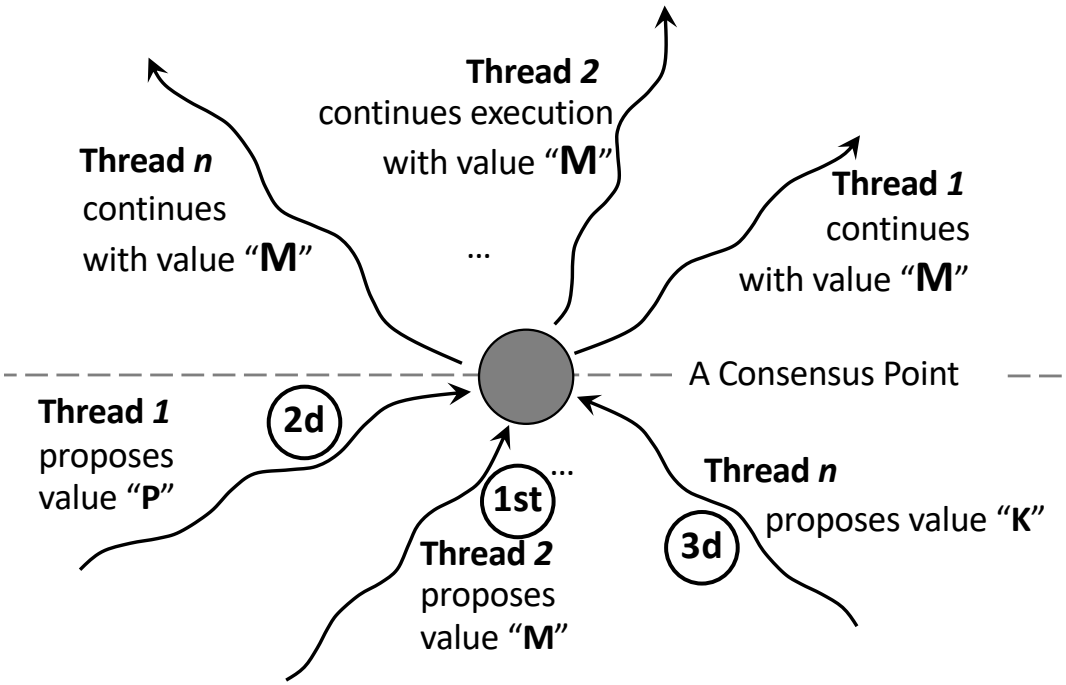

The order of threads proposition typically affects a consensus value;

A consensus value is not necessarily the one proposed **first**

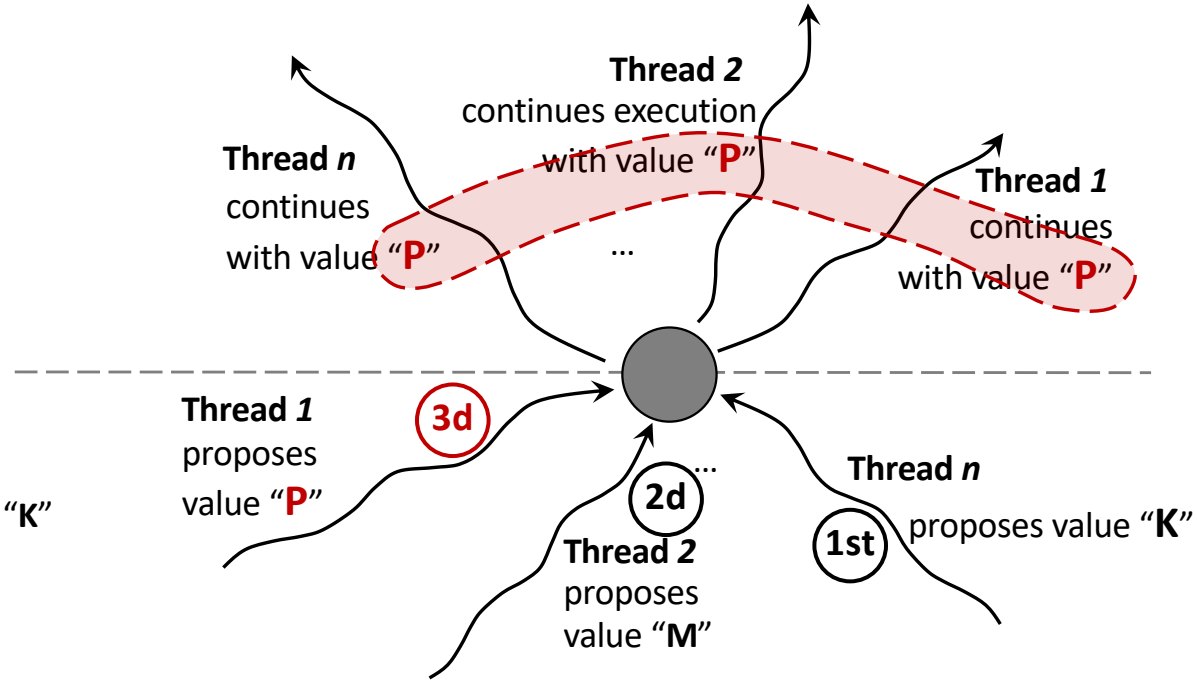
# An Asynchronous Consensus Problem: An Informal Introduction

The key difficulty is that threads make propositions at an unpredictable order

One feasible scenario:




Another feasible scenario:



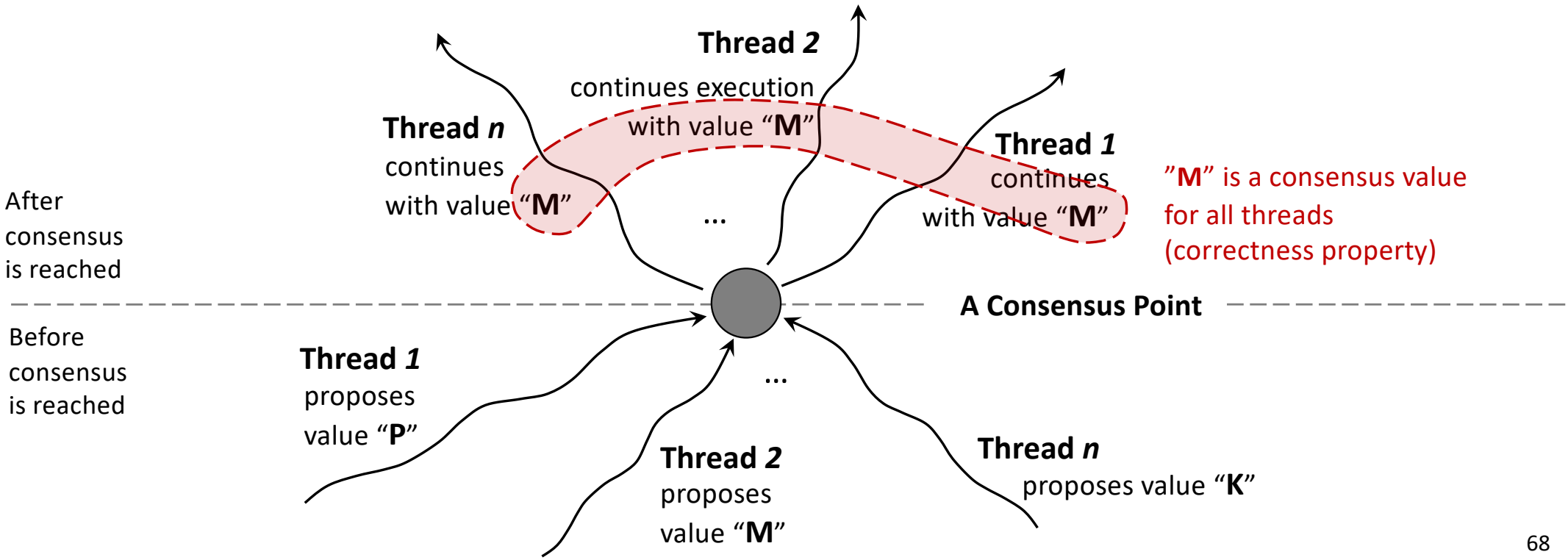

The order of threads proposition typically affects a consensus value;

A consensus value is not necessarily the one proposed first;

A consensus value must instead satisfy 3 major requirements: value consistency, correctness, and completeness

# Consensus Value Properties: Correctness, Consistency, and Completeness

**Correctness**:
After passing a consensus point, every thread proceeds with the same consensus value



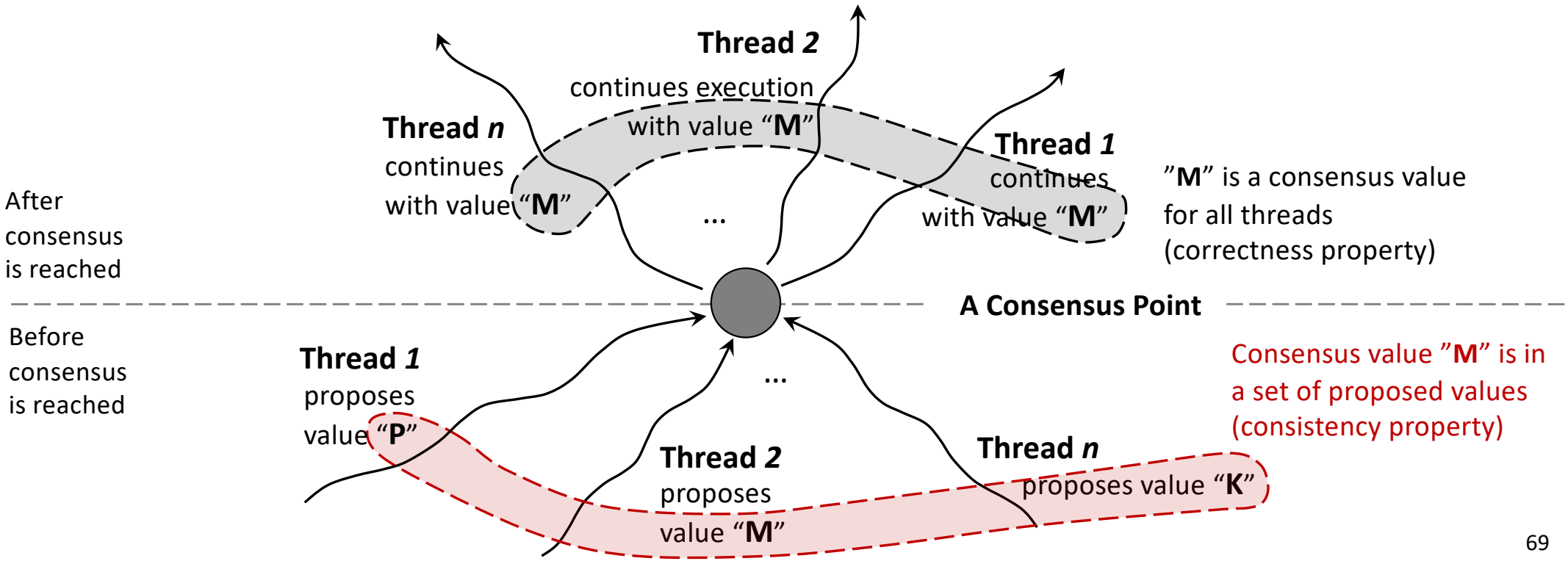
# Consensus Value Properties: Correctness, Consistency, and Completeness

**Correctness**:

After passing a consensus point, every thread proceeds with the same consensus value

**Consistency**:

A consensus value is the one proposed by some thread



# Consensus Value Properties: Correctness, Consistency, and Completeness

**Correctness**:

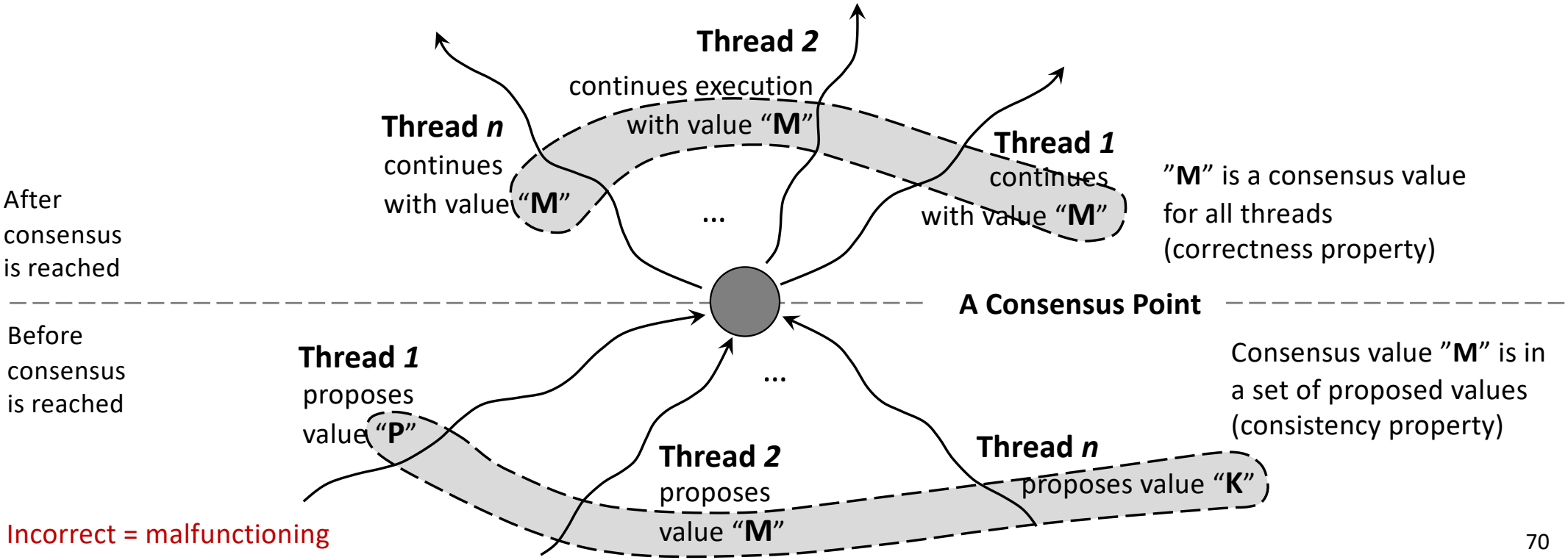
After passing a consensus point, every thread proceeds with the same consensus value

**Consistency**:

A consensus value is the one proposed by some thread

**Completeness**:

Every correct* thread will accept decision at some point




* Incorrect = malfunctioning

# Introduction to Consensus Problem

**The consensus problem** – an abstract theoretical problem, which has enormous consequences to:

- Concurrent algorithms and hardware architecture
- Distributed computing
- Multi-agent systems
- Many other areas

# Introduction to Consensus Problem

**The consensus problem** – an abstract theoretical problem, which has enormous consequences to:

- Concurrent algorithms and hardware architecture
- Distributed computing
- Multi-agent systems
- Many other areas

**Key idea**:

to make all processes (threads) agree on some single value, to proceed with computations further

# Introduction to Consensus Problem

**The consensus problem** – an abstract theoretical problem, which has enormous consequences to:

- Concurrent algorithms and hardware architecture
- Distributed computing
- Multi-agent systems
- Many other areas

**Key idea**:

to make all processes (threads) agree on some single value, to proceed with computations further

**Our key assumption of asynchronous threads:**

Unless stated otherwise, we assume an asynchronous consensus problem, with asynchronous threads, meaning that

- Threads start execution at completely arbitrary (asynchronous) times;
- Threads propose values at unpredictable order

## Application of Consensus Protocols

The fundamental problem of concurrent and distributed systems

| Application Examples | Description |
| --- | --- |
| Clock synchronization in a distributed system | The need to synchronize independent clocks (e.g. due to a clock drift) |

## Application of Consensus Protocols

The fundamental problem of concurrent and distributed systems

| Application Examples | Description |
| --- | --- |
| Clock synchronization in a distributed system | The need to synchronize independent clocks (e.g. due to a clock drift)<br>A widely used Berkeley algorithm for clock synchronization:<br>1) A time server periodically fetches time from all clients;<br>2) Averages the result, and<br>3) Reposts back to all clients;<br>(An example of a synchronous consensus problem) |

## Application of Consensus Protocols

The fundamental problem of concurrent and distributed systems

| Application Examples | Description |
|---|---|
| Clock synchronization in a distributed system | The need to synchronize independent clocks (e.g. due to a clock drift)<br>A widely used Berkeley algorithm for clock synchronization:<br>1) A time server periodically fetches time from all clients;<br>2) Averages the result, and<br>3) Reposts back to all clients;<br>(An example of a synchronous consensus problem) |
| Blockchain Systems | Relies on a fault-tolerant Consensus Problem:<br>- All threads (peers) must agree on the order of transactions;<br>- Some threads (peers) can fail or behave maliciously |

## Application of Consensus Protocols

The fundamental problem of concurrent and distributed systems

| Application Examples | Description |
|---|---|
| Clock synchronization in a distributed system | The need to synchronize independent clocks (e.g. due to a clock drift)<br>A widely used Berkeley algorithm for clock synchronization:<br>1) A time server periodically fetches time from all clients;<br>2) Averages the result, and<br>3) Reposts back to all clients;<br>(An example of a synchronous consensus problem) |
| Blockchain Systems | Relies on a fault-tolerant Consensus Problem:<br>- All threads (peers) must agree on the order of transactions;<br>- Some threads (peers) can fail or behave maliciously |
| Academic studies of non-blocking algorithms | The Consensus Protocol is a toy algorithm, to study the properties of wait-free non-blocking algorithms |

## Application of Consensus Protocols

The fundamental problem of concurrent and distributed systems

| Application Examples | Description |
|---|---|
| Clock synchronization in a distributed system | The need to synchronize independent clocks (e.g. due to a clock drift)<br>A widely used Berkeley algorithm for clock synchronization:<br>1) A time server periodically fetches time from all clients;<br>2) Averages the result, and<br>3) Reposts back to all clients;<br>(An example of a synchronous consensus problem) |
| Blockchain Systems | Relies on a fault-tolerant Consensus Problem:<br>- All threads (peers) must agree on the order of transactions;<br>- Some threads (peers) can fail or behave maliciously |
| Academic studies of non-blocking algorithms | The Consensus Protocol is a toy algorithm, to study the properties of wait-free non-blocking algorithms |

Many other application areas as well

## Application of Consensus Protocols

The fundamental problem of concurrent and distributed systems

| Application Examples | Description |
|---|---|
| Clock synchronization in a distributed system | The need to synchronize independent clocks (e.g. due to a clock drift)<br>A widely used Berkeley algorithm for clock synchronization:<br>1) A time server periodically fetches time from all clients;<br>2) Averages the result, and<br>3) Reposts back to all clients;<br>(An example of a synchronous consensus problem) |
| Blockchain Systems | Relies on a fault-tolerant Consensus Problem:<br>- All threads (peers) must agree on the order of transactions;<br>- Some threads (peers) can fail or behave maliciously |
| Academic studies of non-blocking algorithms | The Consensus Protocol is a toy algorithm, to study the properties of wait-free non-blocking algorithms |

Many other application areas as well

## Consensus Protocol for 2 (Asynchronous) Threads by Using a FIFO queue

Given:

- 2 asynchronous threads, and
- a FIFO queue, denoted by Q, that provides the following methods:

  enqueue( ), dequeue( ), is_empty()

**A sample consensus protocol:**

| Thread 1 | |
|---|---|
| 1 | bool **proposed = 0**; |
| 2 | bool decision = proposed; |
| 3 | if (Q.is_empty()) Q.enqueue(proposed); |
| 4 | else decision = Q.dequeue(); |
| 5 | return **decision**; |

| Thread 2 | |
|---|---|
| 1 | bool **proposed = 1**; |
| 2 | bool decision = proposed; |
| 3 | if (Q.is_empty()) Q.enqueue(proposed); |
| 4 | else decision = Q.dequeue(); |
| 5 | return **decision**; |

*Assumption: for simplicity of explanation, code in line 3 is assumed to execute atomically*

Observation:

Both threads always return the same “decision” value, despite being asynchronous and having different “proposed” values!

**Q.: How to prove this observation?**

## Consensus Protocol for 2 (Asynchronous) Threads by Using a FIFO queue

Given:

- 2 asynchronous threads, and
- a FIFO queue, denoted by Q, that provides the following methods:

  enqueue( ), dequeue( ), is_empty()

**A sample consensus protocol:**

| Thread 1 | |
|---|---|
| 1 | bool **proposed = 0**; |
| 2 | bool decision = proposed; |
| 3 | if (Q.is_empty()) Q.enqueue(proposed); |
| 4 | else decision = Q.dequeue(); |
| 5 | return **decision**; |

| Thread 2 | |
|---|---|
| 1 | bool **proposed = 1**; |
| 2 | bool decision = proposed; |
| 3 | if (Q.is_empty()) Q.enqueue(proposed); |
| 4 | else decision = Q.dequeue(); |
| 5 | return **decision**; |

*Assumption: for simplicity of explanation, code in line 3 is assumed to execute atomically*

Observation:

Both threads always return the same “decision” value, despite being asynchronous and having different “proposed” values!

**Q.: How to prove this observation?** **A.: To enumerate all feasible relative execution scenarios for threads!**

## Enumeration of Feasible Execution Orders for Threads by Using an Execution Tree

Move 1: if (Q.is_empty()) Q.enqueue(proposed);

Move 2: else decision = Q.dequeue();

## Enumeration of Feasible Execution Orders for Threads by Using an Execution Tree

Move 1: if (Q.is_empty()) Q.enqueue(proposed);

Move 2: else decision = Q.dequeue();

Initial protocol state

*Q is empty*

## Enumeration of Feasible Execution Orders for Threads by Using an Execution Tree

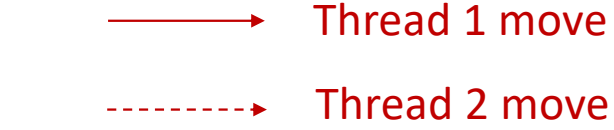
Move 1: if (Q.is_empty()) Q.enqueue(proposed);

Move 2: else decision = Q.dequeue();

## Enumeration of Feasible Execution Orders for Threads by Using an Execution Tree

Move 1: if (Q.is_empty()) Q.enqueue(proposed);

Move 2: else decision = Q.dequeue();




Suppose that:

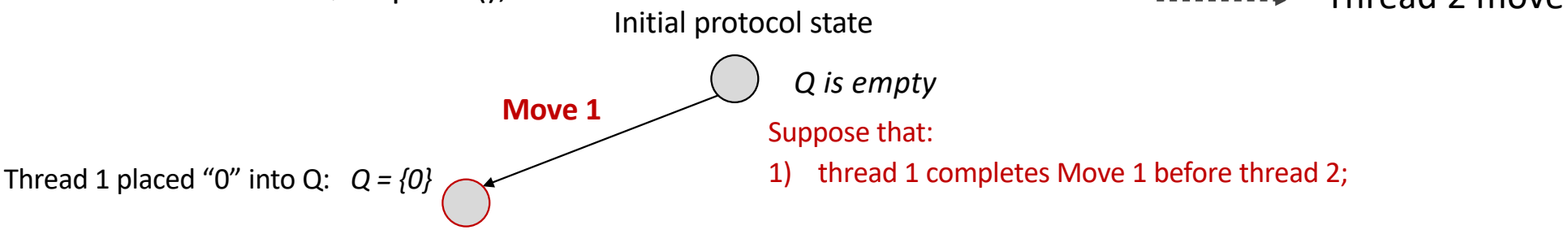
1) thread 1 completes Move 1 before thread 2;

| Lin. Points | Thread 1 | Thread 2 |
|---|---|---|
| 1 | bool proposed = 0; | |
| 2 | bool decision = proposed; | |
| 3 | if (Q.**is_empty**()) Q.**enqueue**(proposed); | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

## Enumeration of Feasible Execution Orders for Threads by Using an Execution Tree

Move 1: if (Q.is_empty()) Q.enqueue(proposed);

Move 2: else decision = Q.dequeue();




| Lin. Points | Thread 1 | Thread 2 |
|---|---|---|
| 1 | bool proposed = 0; | |
| 2 | bool decision = proposed; | |
| 3 | if (Q.**is_empty**()) Q.**enqueue**(proposed); | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

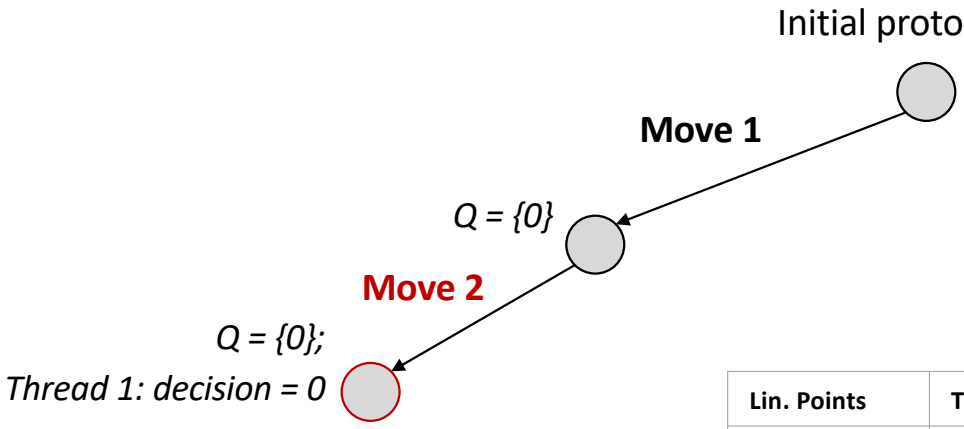
# Enumeration of Feasible Execution Orders for Threads by Using an Execution Tree

Move 1: if (Q.is_empty()) Q.enqueue(proposed);

Move 2: else decision = Q.dequeue();

Thread 1 move

Thread 2 move




Suppose that:

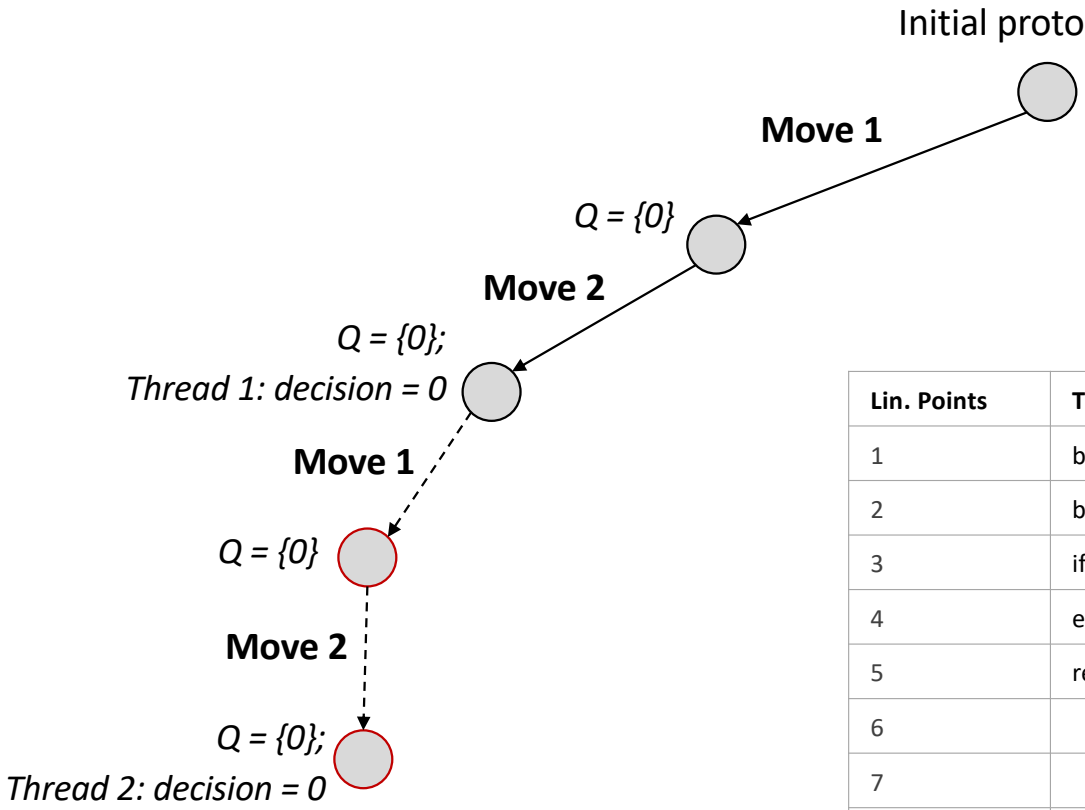
1) thread 1 completes Move 1 before thread 2;
2) thread 1 completes Move 2 before thread 2 completes Move 1;

| Lin. Points | Thread 1 | Thread 2 |
|---|---|---|
| 1 | bool proposed = 0; | |
| 2 | bool decision = proposed; | |
| 3 | if (Q.**is_empty**()) Q.**enqueue**(proposed); | |
| 4 | else decision = Q.**dequeue**(); | |
| 5 | return decision; | |
| | | |
| | | |
| | | |
| | | |
| | | |

# Enumeration of Feasible Execution Orders for Threads by Using an Execution Tree

Move 1: if (Q.is_empty()) Q.enqueue(proposed);

Move 2: else decision = Q.dequeue();




Suppose that:

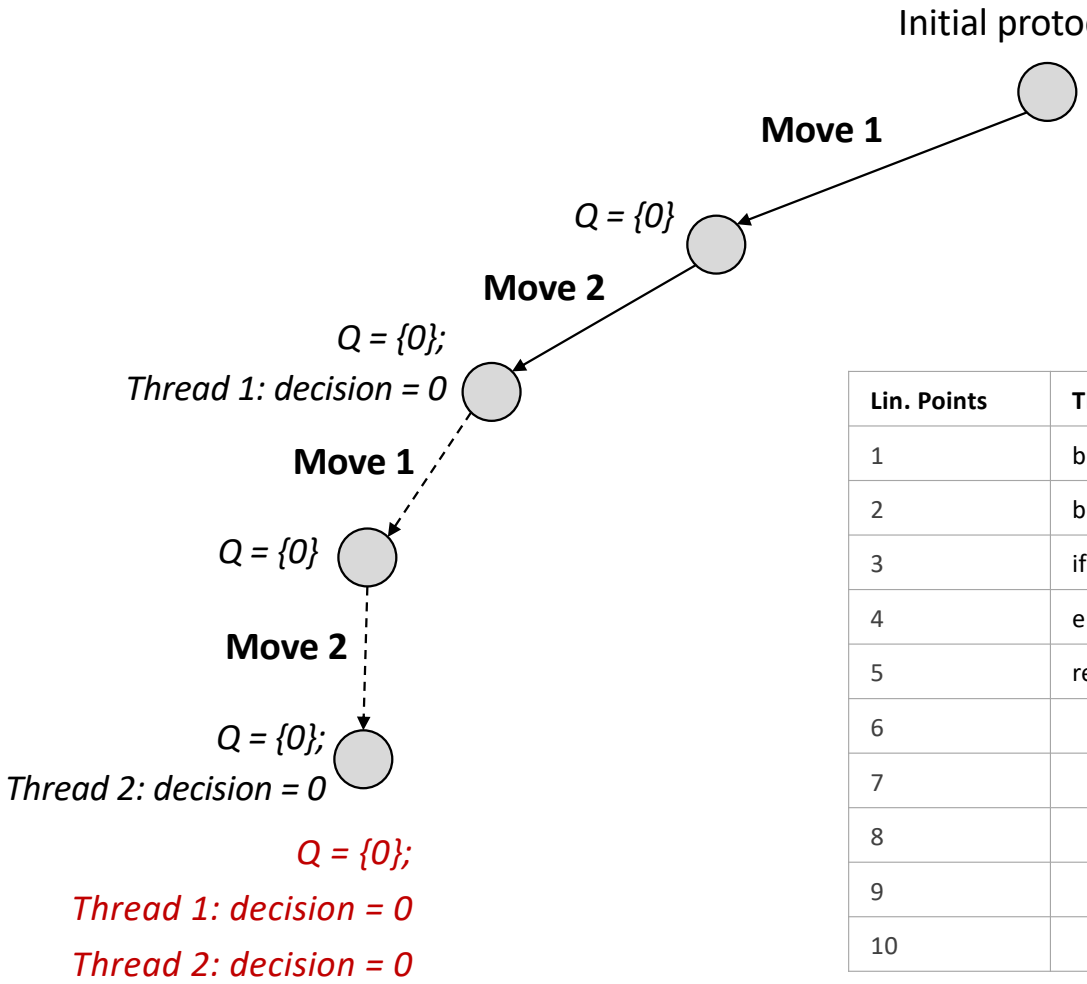
1) thread 1 completes Move 1 before thread 2;
2) thread 1 completes Move 2 before thread 2 completes Move 1;
3) thread 2 finally completes Move 1, and then Move 2

| Lin. Points | Thread 1 | Thread 2 |
|---|---|---|
| 1 | bool proposed = 0; | |
| 2 | bool decision = proposed; | |
| 3 | if (Q.**is_empty**()) Q.**enqueue**(proposed); | |
| 4 | else decision = Q.**dequeue**(); | |
| 5 | return decision; | |
| 6 | | bool proposed = 1; |
| 7 | | bool decision = proposed; |
| 8 | | if (Q.**is_empty**()) Q.**enqueue**(proposed); |
| 9 | | else decision = Q.**dequeue**(); |
| 10 | | return decision; |

# Enumeration of Feasible Execution Orders for Threads by Using an Execution Tree

Move 1: if (Q.is_empty()) Q.enqueue(proposed);

Move 2: else decision = Q.dequeue();




Suppose that:

1) thread 1 completes Move 1 before thread 2;
2) thread 1 completes Move 2 before thread 2 completes Move 1;
3) thread 2 finally completes Move 1, and then Move 2

| Lin. Points | Thread 1 | Thread 2 |
|---|---|---|
| 1 | bool proposed = 0; | |
| 2 | bool decision = proposed; | |
| 3 | if (Q.**is_empty**()) Q.**enqueue**(proposed); | |
| 4 | else decision = Q.**dequeue**(); | |
| 5 | return decision; | |
| 6 | | bool proposed = 1; |
| 7 | | bool decision = proposed; |
| 8 | | if (Q.**is_empty**()) Q.**enqueue**(proposed); |
| 9 | | else decision = Q.**dequeue**(); |
| 10 | | return decision; |

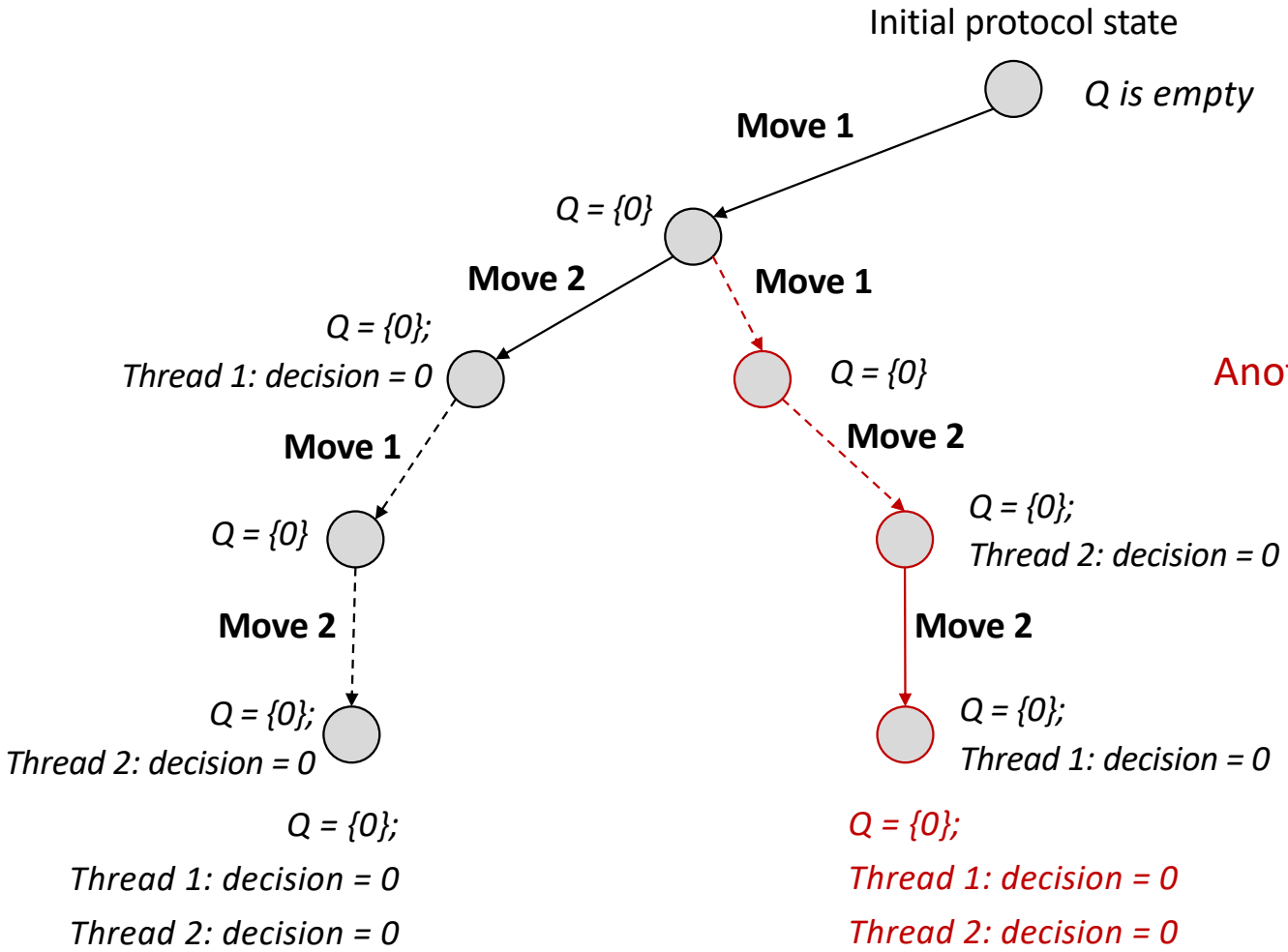
## Enumeration of Feasible Execution Orders for Threads by Using an Execution Tree

Move 1: if (Q.is_empty()) Q.enqueue(proposed);

Move 2: else decision = Q.dequeue();

Thread 1 move

Thread 2 move

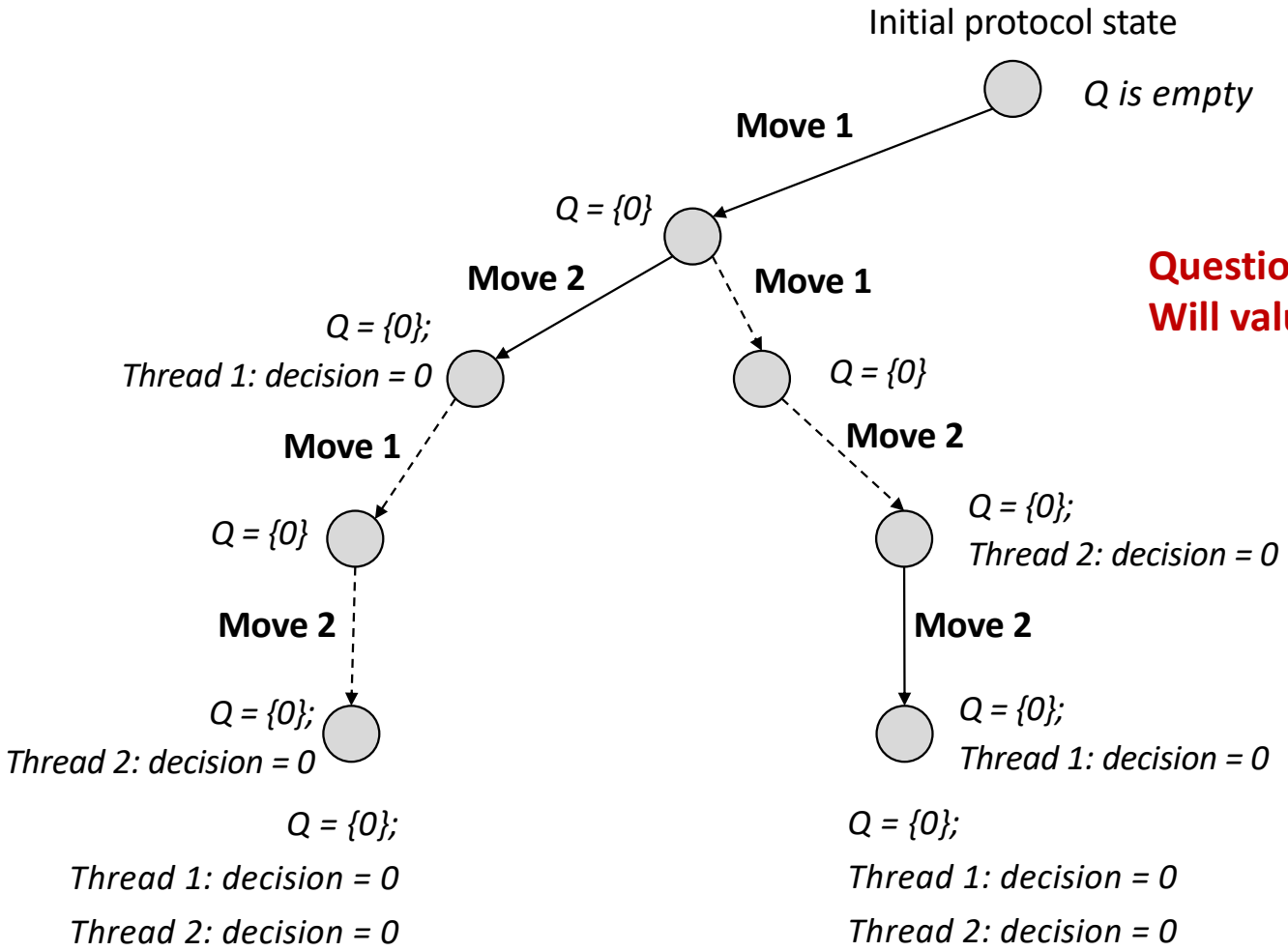



Another feasible scenario

## Enumeration of Feasible Execution Orders for Threads by Using an Execution Tree

Move 1: if (Q.is_empty()) Q.enqueue(proposed);

Move 2: else decision = Q.dequeue();

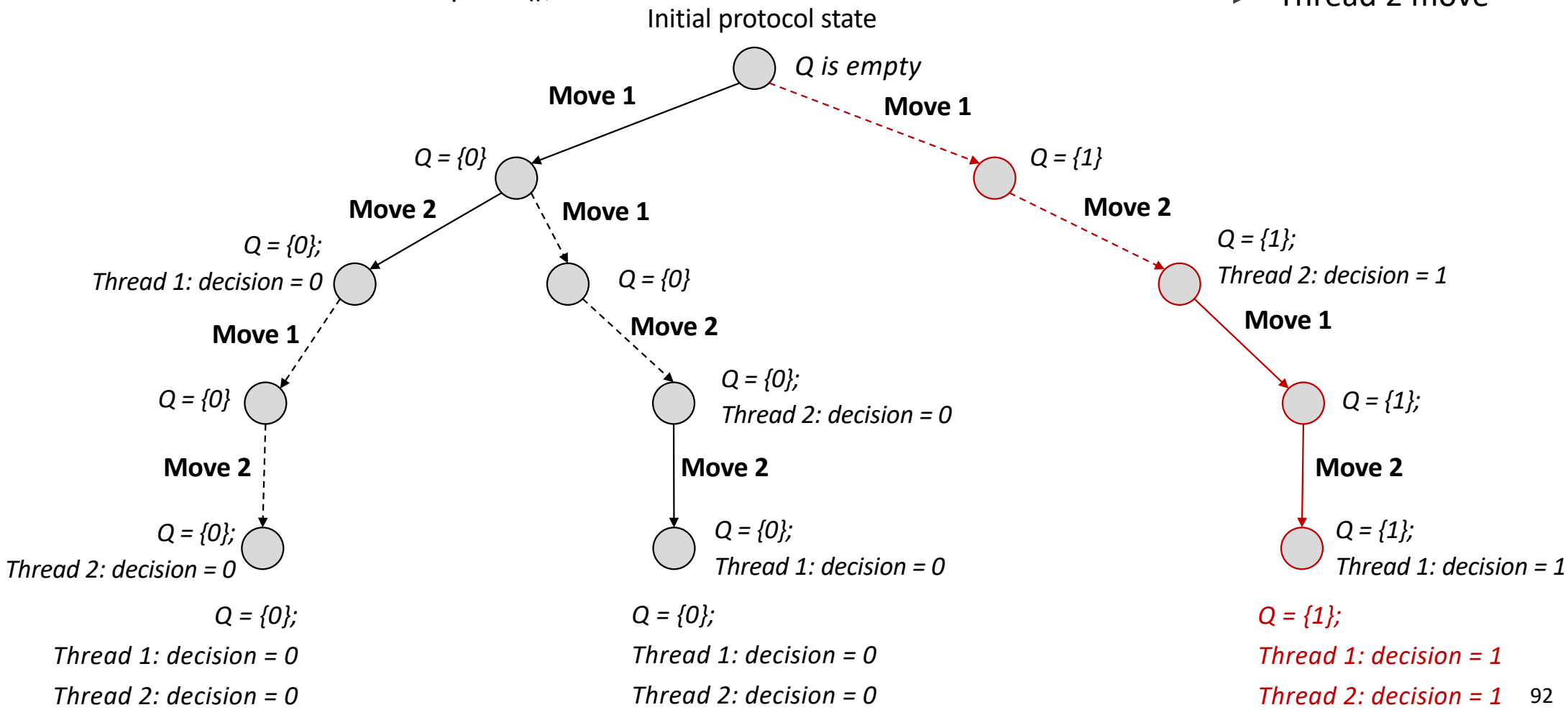



**Question:**
**Will value of "decision" varible always equal to"0"?**

# Enumeration of Feasible Execution Orders for Threads by Using an Execution Tree

Move 1: if (Q.is_empty()) Q.enqueue(proposed);

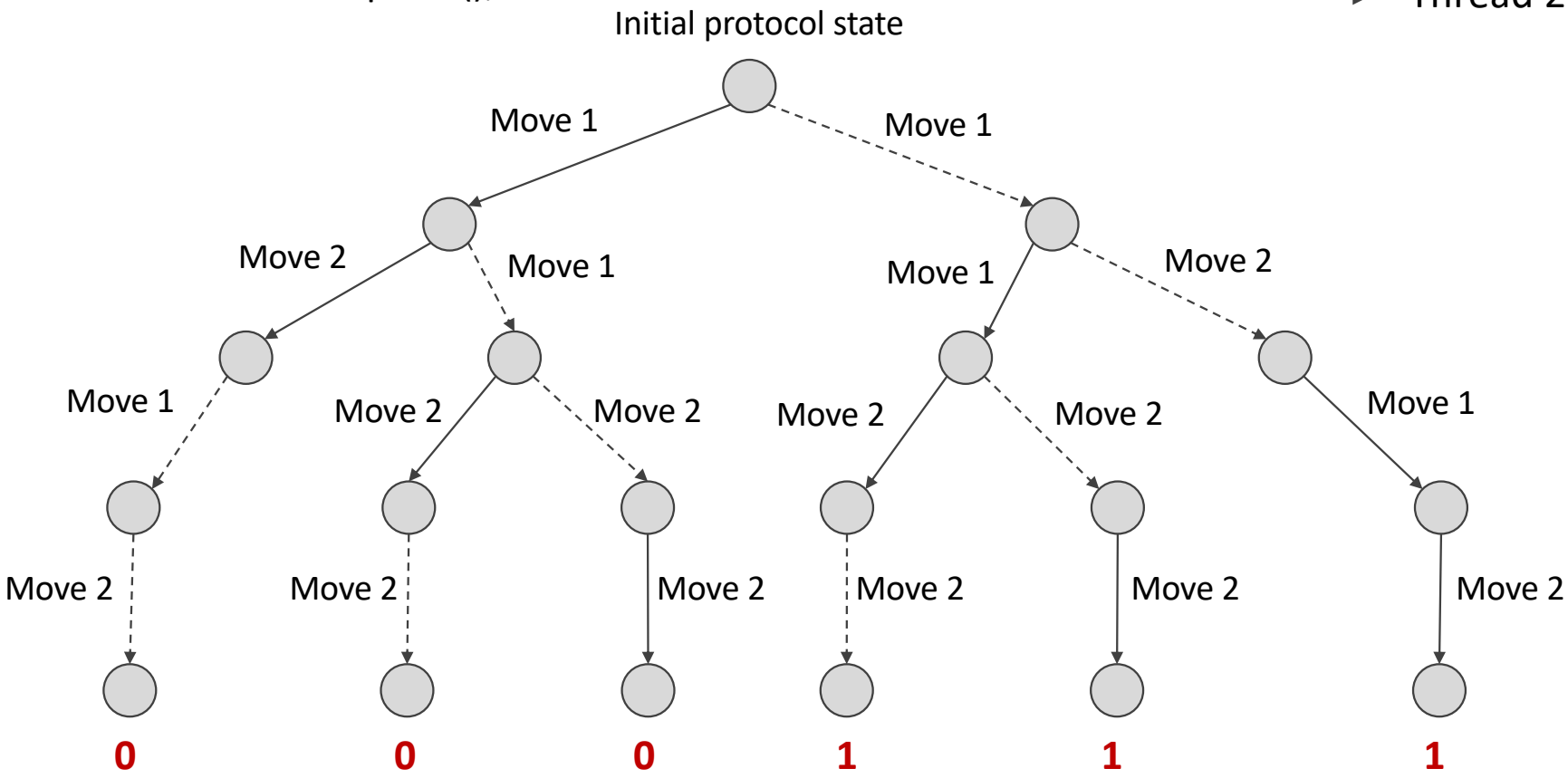
Move 2: else decision = Q.dequeue();



## An Execution Tree for Consensus Protocol

Move 1: if (Q.is_empty()) Q.enqueue(proposed);

Move 2: else decision = Q.dequeue();

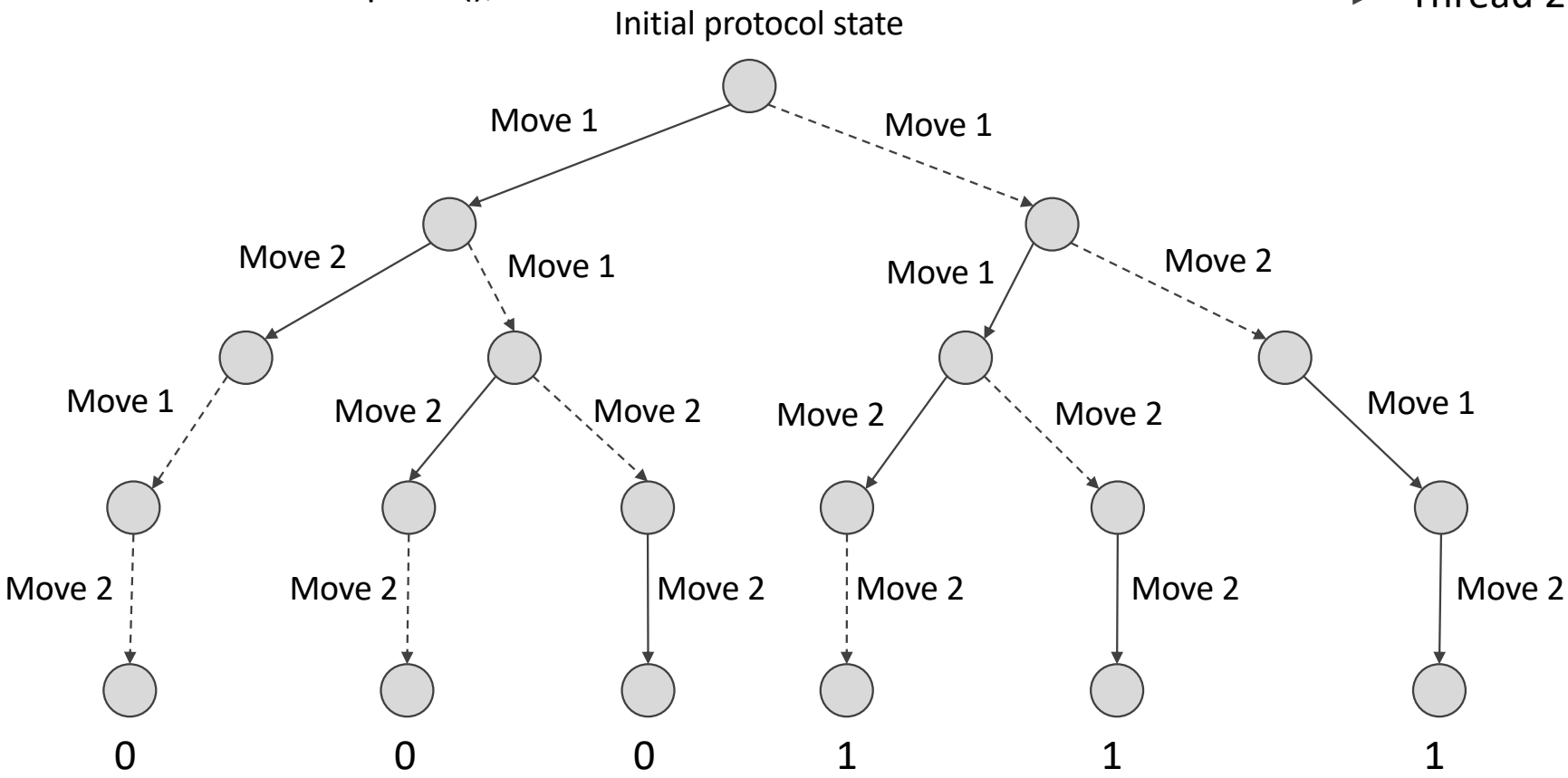



Consensus (decision) values

## An Execution Tree for Consensus Protocol

Move 1: if (Q.is_empty()) Q.enqueue(proposed);

Move 2: else decision = Q.dequeue();

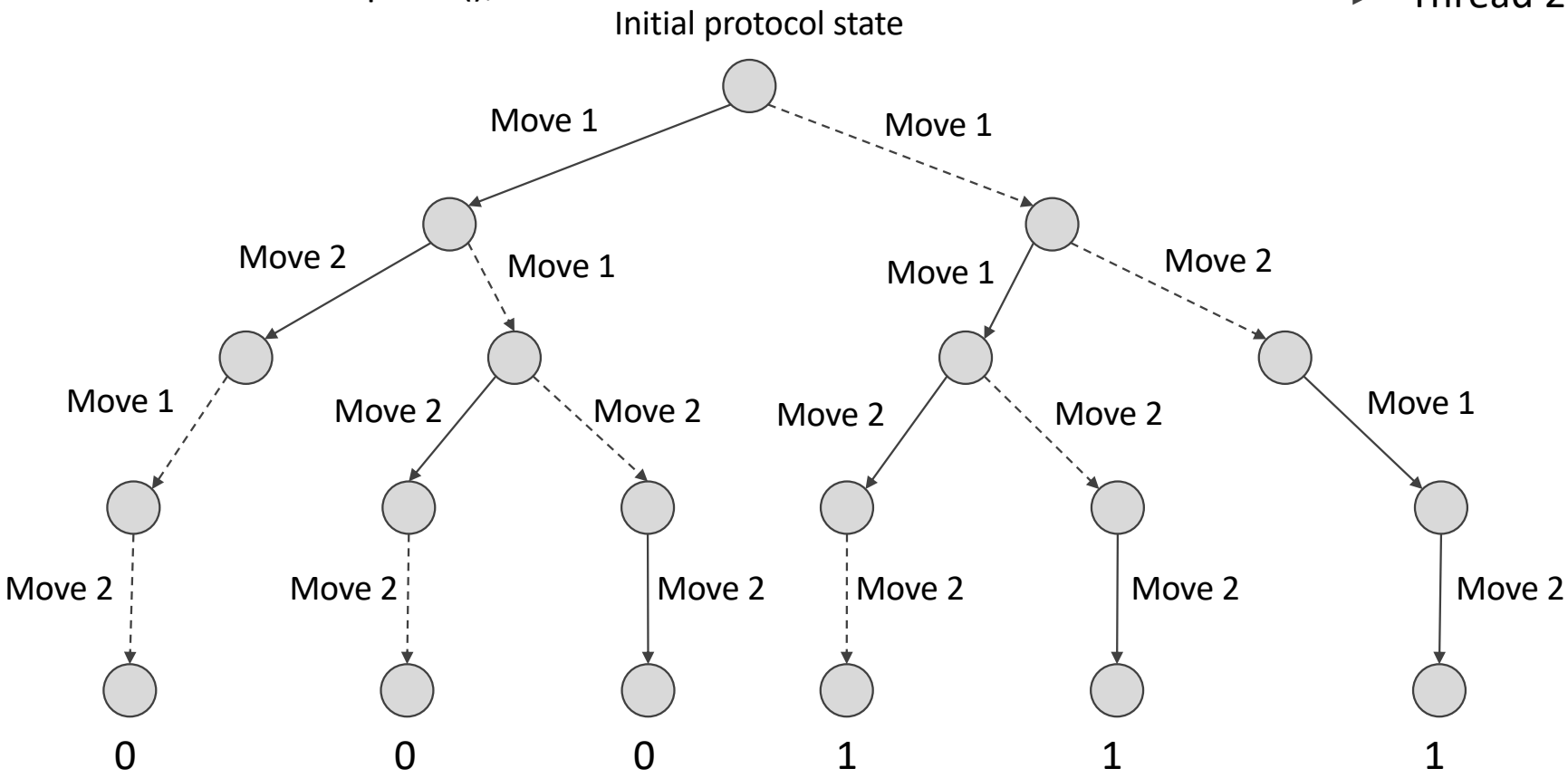



This execution tree enumerates all feasible execution scenarios for 2 asynchronous threads

## An Execution Tree for Consensus Protocol

Move 1: if (Q.is_empty()) Q.enqueue(proposed);

Move 2: else decision = Q.dequeue();




This execution tree enumerates all feasible execution scenarios for 2 asynchronous threads;

*Note: if we relax the assumption of Move 1 to be atomic, a tree will be significantly larger*

# Recap: A Consensus Protocol for 2 (Asynchronous) Threads by Using a FIFO queue

Given:
- 2 asynchronous threads, and
- a FIFO queue, denoted by Q, that provides the following methods:

  **enqueue**( ), **dequeue**( ), **is_empty**()

**A sample consensus protocol:**

| Thread 1 | |
|---|---|
| 1 | bool proposed = 0; |
| 2 | bool decision = proposed; |
| 3 | if (Q.**is_empty**()) Q.**enqueue**(proposed); |
| 4 | else decision = Q.**dequeue**(); |
| 5 | return decision; |

| Thread 2 | |
|---|---|
| 1 | bool proposed = 1; |
| 2 | bool decision = proposed; |
| 3 | if (Q.**is_empty**()) Q.**enqueue**(proposed); |
| 4 | else decision = Q.**dequeue**(); |
| 5 | return decision; |

*Assumption: for simplicity of explanation, code in line 3 is assumed to execute atomically*

The protocol above synchronizes 2 threads

# Recap: A Consensus Protocol for 2 (Asynchronous) Threads by Using a FIFO queue

Given:
- 2 asynchronous threads, and
- a FIFO queue, denoted by Q, that provides the following methods:

  **enqueue**( ), **dequeue**( ), **is_empty**()

**A sample consensus protocol:**

| Thread 1 | |
|---|---|
| 1 | bool proposed = 0; |
| 2 | bool decision = proposed; |
| 3 | if (Q.**is_empty**()) Q.**enqueue**(proposed); |
| 4 | else decision = Q.**dequeue**(); |
| 5 | return decision; |

| Thread 2 | |
|---|---|
| 1 | bool proposed = 1; |
| 2 | bool decision = proposed; |
| 3 | if (Q.**is_empty**()) Q.**enqueue**(proposed); |
| 4 | else decision = Q.**dequeue**(); |
| 5 | return decision; |

*Assumption: for simplicity of explanation, code in line 3 is assumed to execute atomically*

The protocol above synchronizes 2 threads

How to synchronize 3 threads by using the same queue?

# Recap: A Consensus Protocol for 2 (Asynchronous) Threads by Using a FIFO queue

Given:
- 2 asynchronous threads, and
- a FIFO queue, denoted by Q, that provides the following methods:

  **enqueue**( ), **dequeue**( ), **is_empty**()

**A sample consensus protocol:**

| Thread 1 | |
|---|---|
| 1 | bool proposed = 0; |
| 2 | bool decision = proposed; |
| 3 | if (Q.**is_empty**()) Q.**enqueue**(proposed); |
| 4 | else decision = Q.**dequeue**(); |
| 5 | return decision; |

| Thread 2 | |
|---|---|
| 1 | bool proposed = 1; |
| 2 | bool decision = proposed; |
| 3 | if (Q.**is_empty**()) Q.**enqueue**(proposed); |
| 4 | else decision = Q.**dequeue**(); |
| 5 | return decision; |

*Assumption: for simplicity of explanation, code in line 3 is assumed to execute atomically*

The protocol above synchronizes 2 threads

How to synchronize 3 threads by using the same queue?

**It is impossible to synchronize more than 2 threads by using such a FIFO queue!**

## Another Example: An Extended FIFO Queue

An extended FIFO queue *Q* provides 2 methods:

**enqueue**( ) – to enqueue an element into *Q*

**peek**() – to read the first (top) element without modifying *Q*

Can we solve the consensus problem for 2 threads?

And for 3 threads?

## An Extended FIFO Queue: A Consensus Object for Multiple Threads

An extended FIFO queue *Q* provides 2 methods:

**enqueue**( ) – to enqueue an element into *Q*

**peek**() – to read the first (top) element without modifying *Q*

**Consensus protocol for N threads** (again, threads are asynchronous):

| Thread 1 | |
|---|---|
| 1 | int proposed = 1; |
| 2 | int decision; |
| 3 | Q.**enqueue**(proposed); |
| 4 | decision = Q.**peek**(); |
| 5 | return decision; |

...

| Thread N | |
|---|---|
| 1 | int proposed = N; |
| 2 | int decision; |
| 3 | Q.**enqueue**(proposed); |
| 4 | decision = Q.**peek**(); |
| 5 | return decision; |

An extended queue allows to synchronize an infinite number of threads

(unlike a standard FIFO queue, which applies to 2 threads at most)

# Consensus Numbers for Sample Synchronization Objects

| **Object Type** | **Consensus Number** |
|---|---|
| Queues of types: FIFO, double ended, or priority queue | 2 |
| An extended queue (supports peek()) | Infinite |
| A register, that supports CAS operation | Infinite |
| Set | 2 |
| Atomic register | 1 |

Consensus Number – the number of threads, for which consensus problem is solved

# Consensus Numbers for Sample Synchronization Objects

| **Object Type** | **Consensus Number** |
|---|---|
| Queues of types: FIFO, double ended, or priority queue | 2 |
| An extended queue (supports peek()) | Infinite |
| A register, that supports CAS operation | Infinite |
| Set | 2 |
| Atomic register | 1 |

Consensus Number – the number of threads, for which consensus problem is solved

Consensus Theory allows to prove consensus numbers of certain objects

# A Binary Consensus Problem

A particular case of a consensus problem, with the following restrictions:

- Only two threads
- Threads propose either 0 or 1 for a consensus value
- 0 and 1 are both reachable consensus values

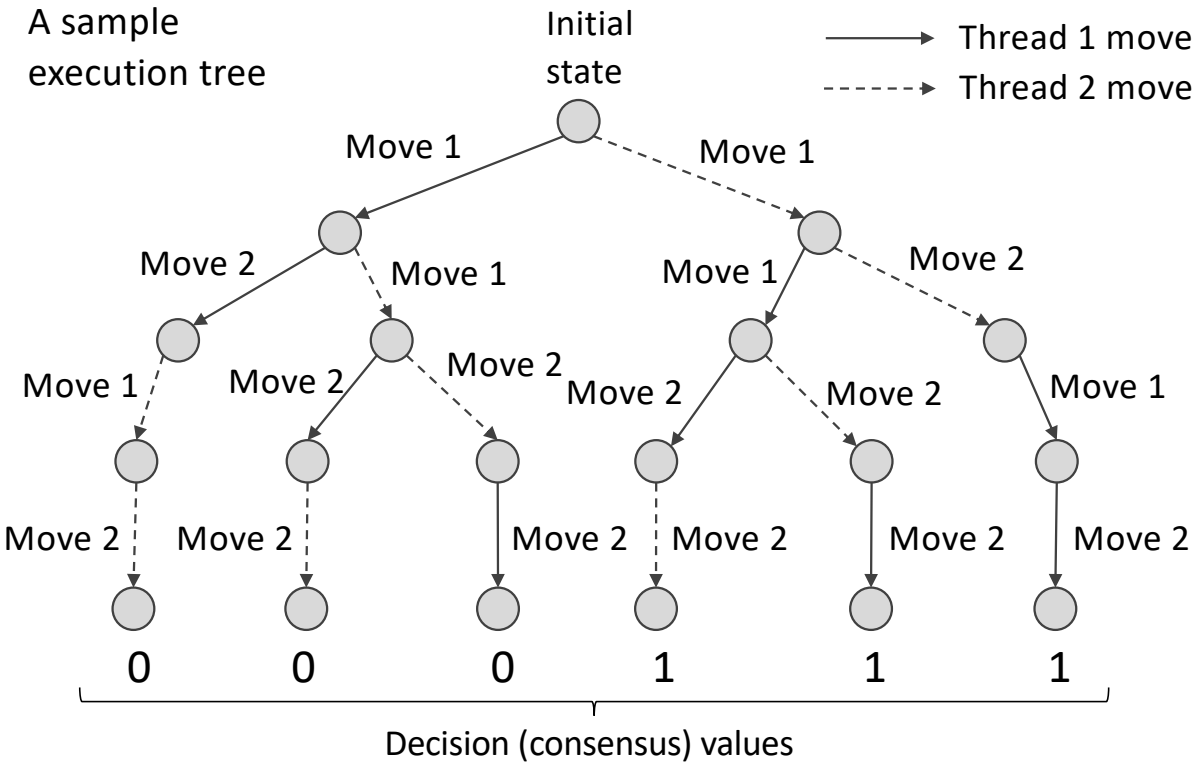
A binary consensus problem is our toy example, to demonstrate a few math techniques for:

- proving several properties of a binary consensus protocols, and
- analysing consensus numbers for some concurrent objects

# Recap: An Execution Tree for a Consensus Protocol

A protocol state characterises:

- the state of every concurrent thread (e.g. the index of an executing instruction, and local variable values);
- the state of (a) concurrent object(s) used for threads synchronisation (e.g. a concurrent queue);

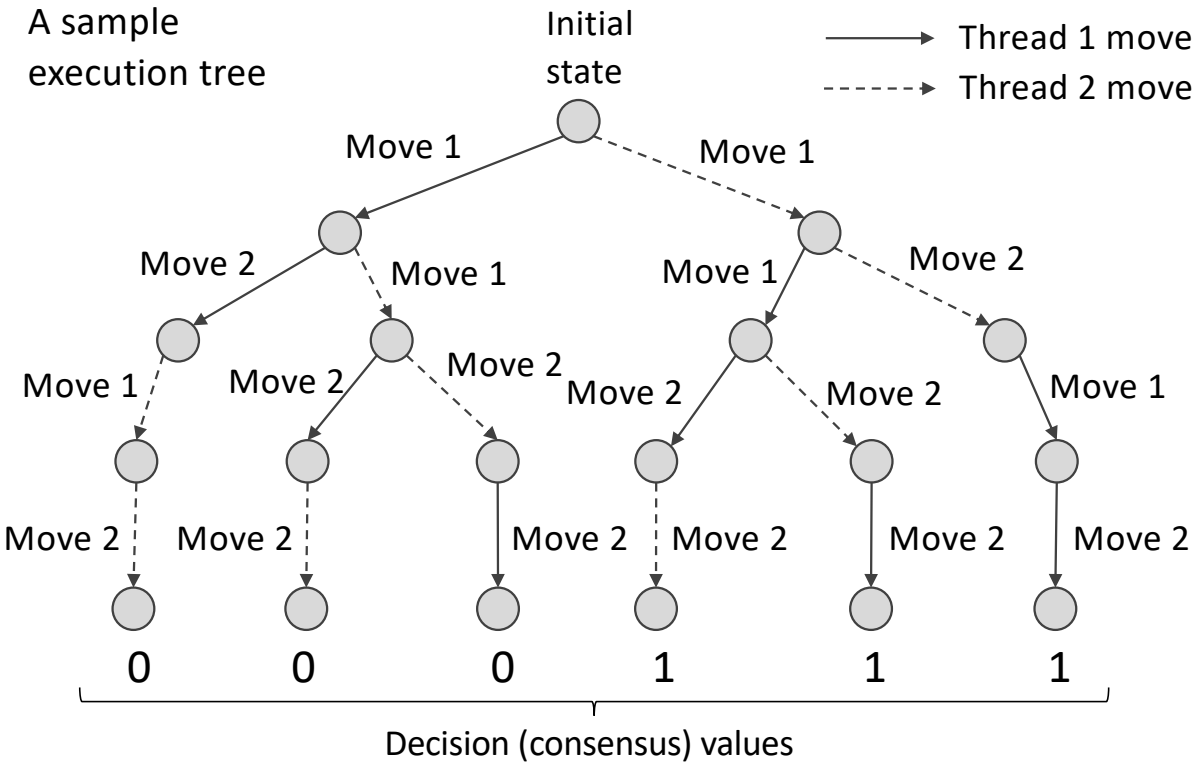


# Recap: An Execution Tree for a Consensus Protocol

A protocol state characterises:

- the state of every concurrent thread (e.g. the index of an executing instruction, and local variable values);
- the state of (a) concurrent object(s) used for threads synchronisation (e.g. a concurrent queue);

An execution tree of a consensus protocol – a set of all feasible protocol states and transitions between them

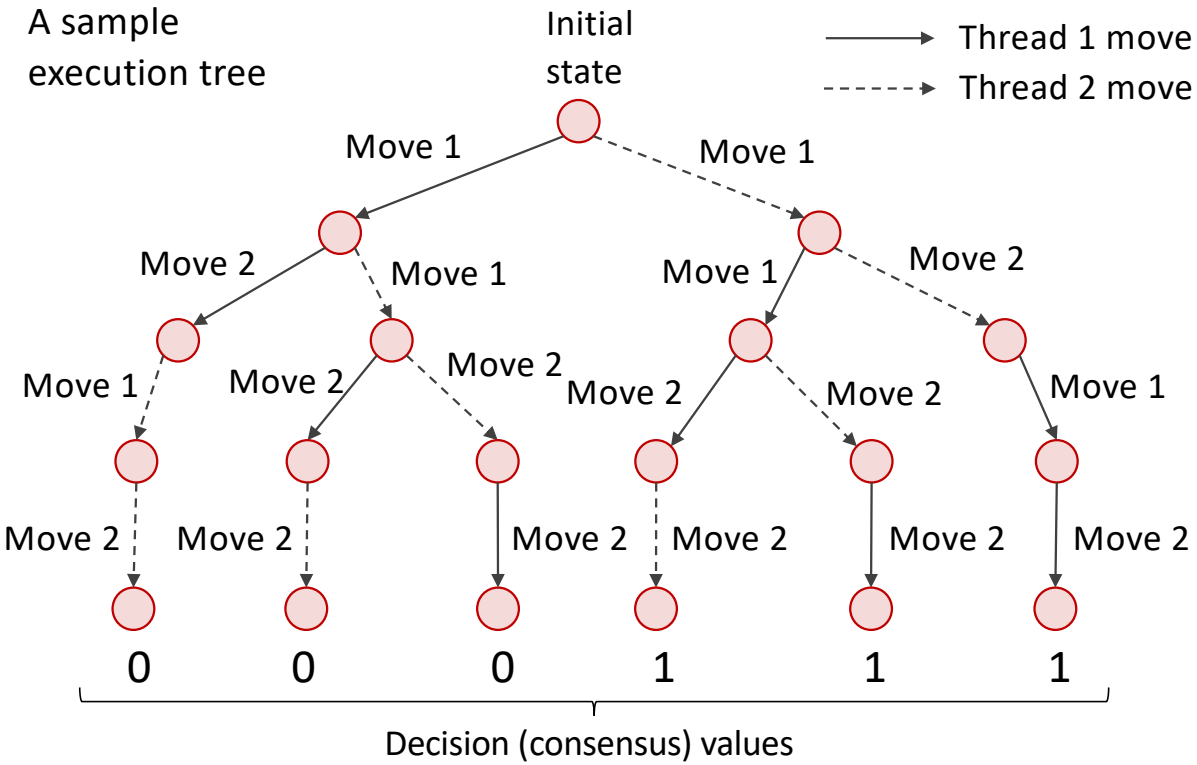


# An Execution Tree for a Consensus Protocol

A protocol state characterises:

- the state of every concurrent thread (e.g. the index of an executing instruction, and local variable values);
- the state of (a) concurrent object(s) used for threads synchronisation (e.g. a concurrent queue);

An execution tree of a consensus protocol – a set of all feasible protocol states and transitions between them




**An execution tree:**

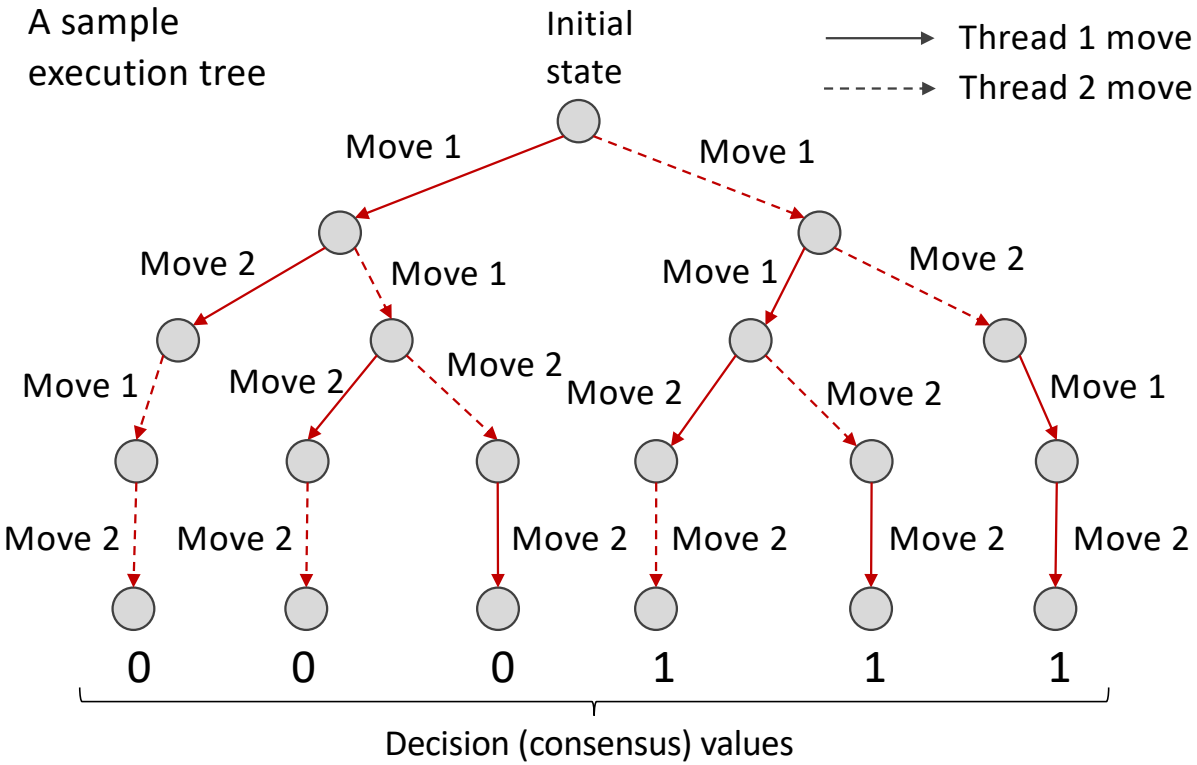
- Node – a protocol state

# An Execution Tree for a Consensus Protocol

A protocol state characterises:

- the state of every concurrent thread (e.g. the index of an executing instruction, and local variable values);
- the state of (a) concurrent object(s) used for threads synchronisation (e.g. a concurrent queue);

An execution tree of a consensus protocol – a set of all feasible protocol states and transitions between them




**An execution tree:**

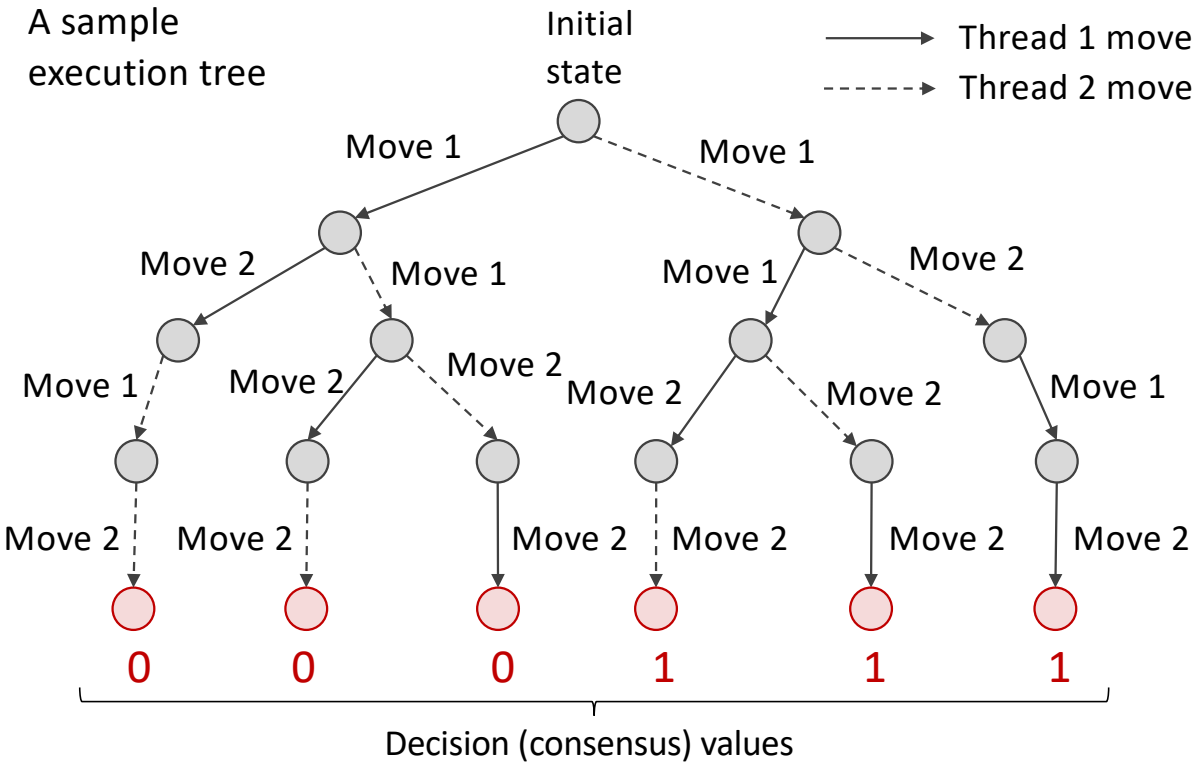
- Node – a protocol state
- Edge – move by one of the threads, that leads to a new state

# An Execution Tree for a Consensus Protocol

A protocol state characterises:

- the state of every concurrent thread (e.g. the index of an executing instruction, and local variable values);
- the state of (a) concurrent object(s) used for threads synchronisation (e.g. a concurrent queue);

An execution tree of a consensus protocol – a set of all feasible protocol states and transitions between them

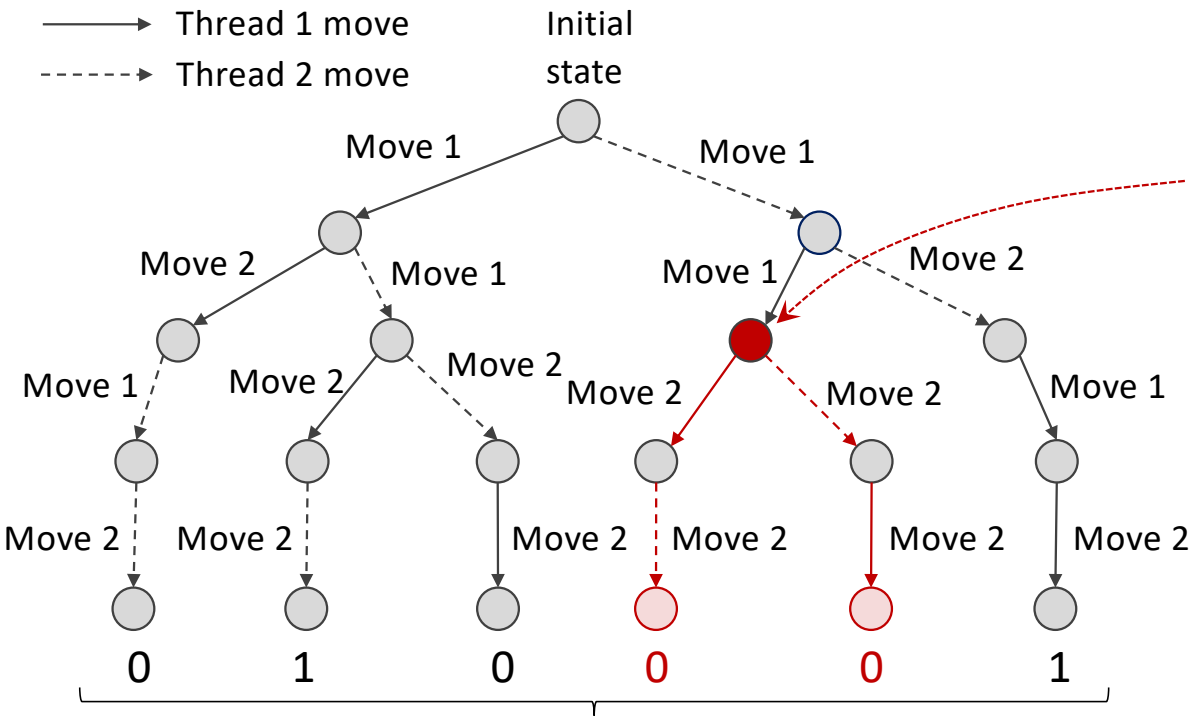



**An execution tree:**

- Node – a protocol state
- Edge – move by one of the threads, that leads to a new state
- Final state is marked with a decision value

# The Notion of a Protocol State Valency

- An univalent state: all final states, following from this state, have the same decision (consensus) value



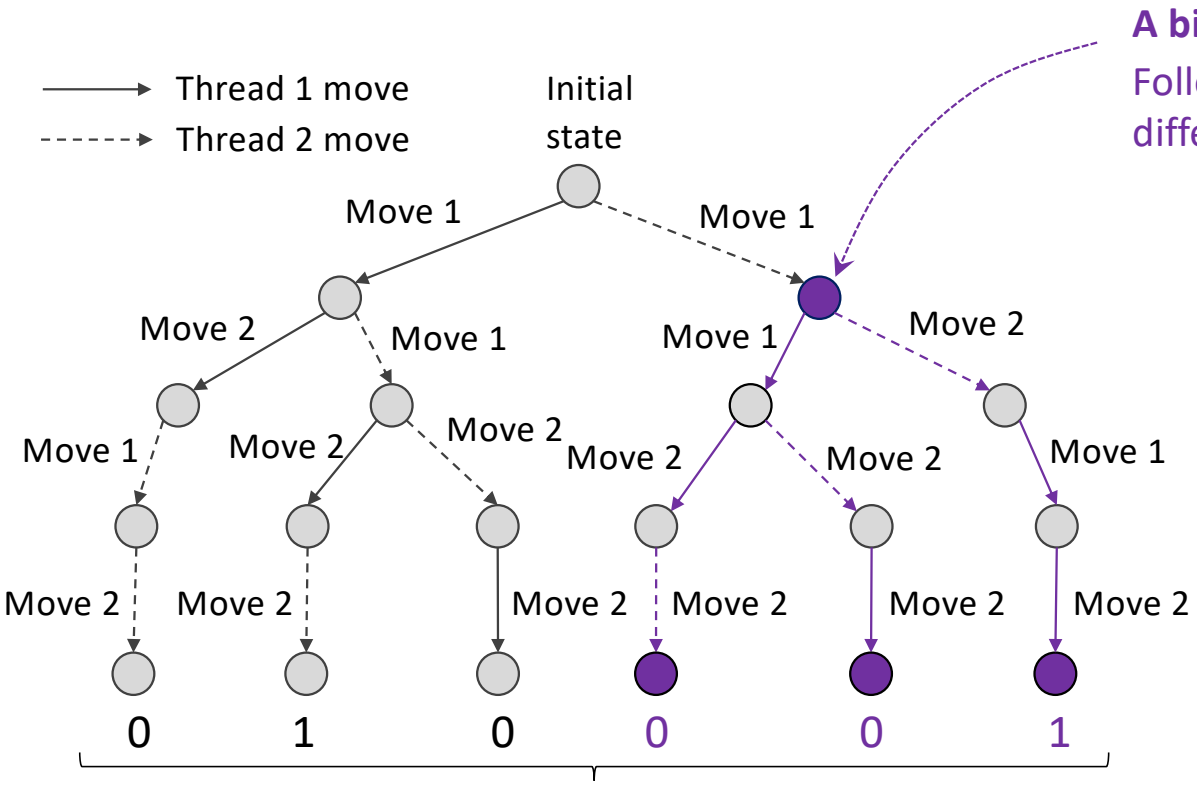

**An univalent state:**

All following final states have the same consensus value

(We shuffled decision values compared to a previous slide)

# The Notion of a Protocol State Valency

- An univalent state: all final states, following from this state, have the same decision (consensus) value
- A bivalent state: the decision value for a consensus problem is not yet fixed

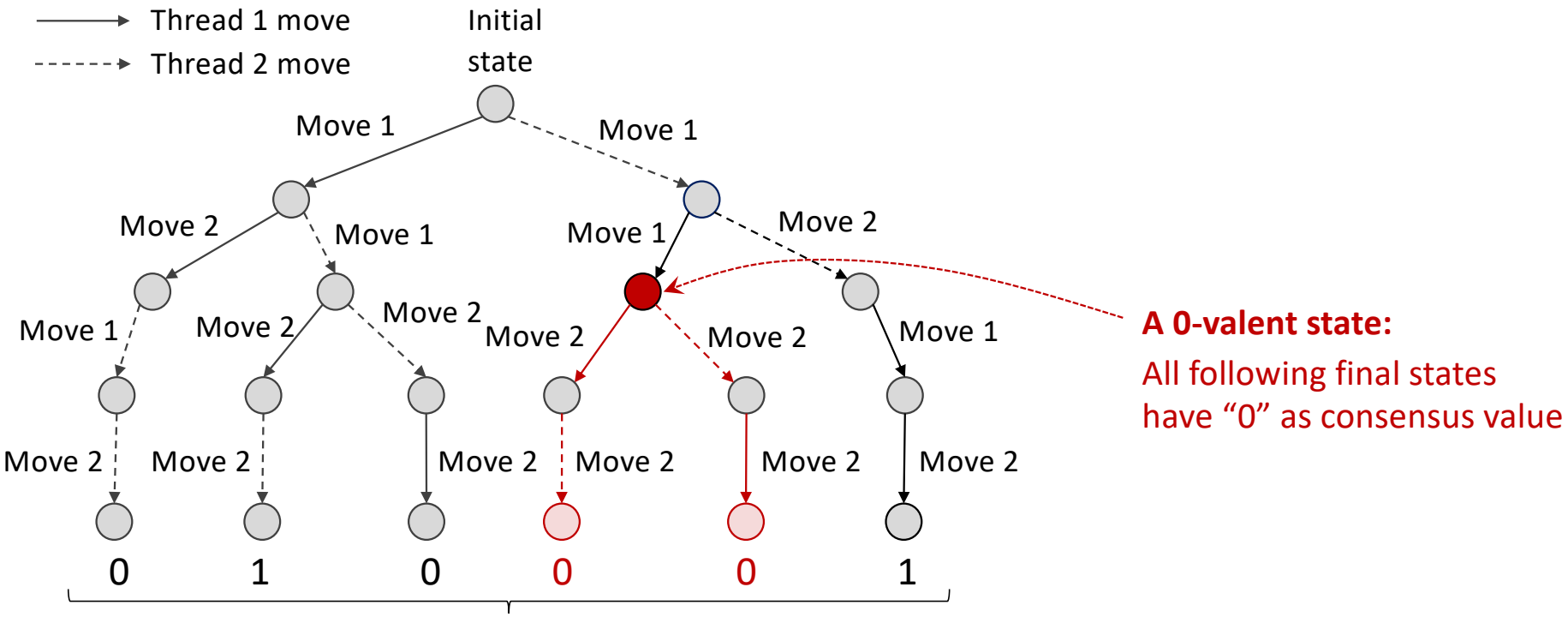



**A bivalent state:**
Following final states have different consensus values

(We shuffled decision values compared to a previous slide)

# The Notion of a Protocol State Valency

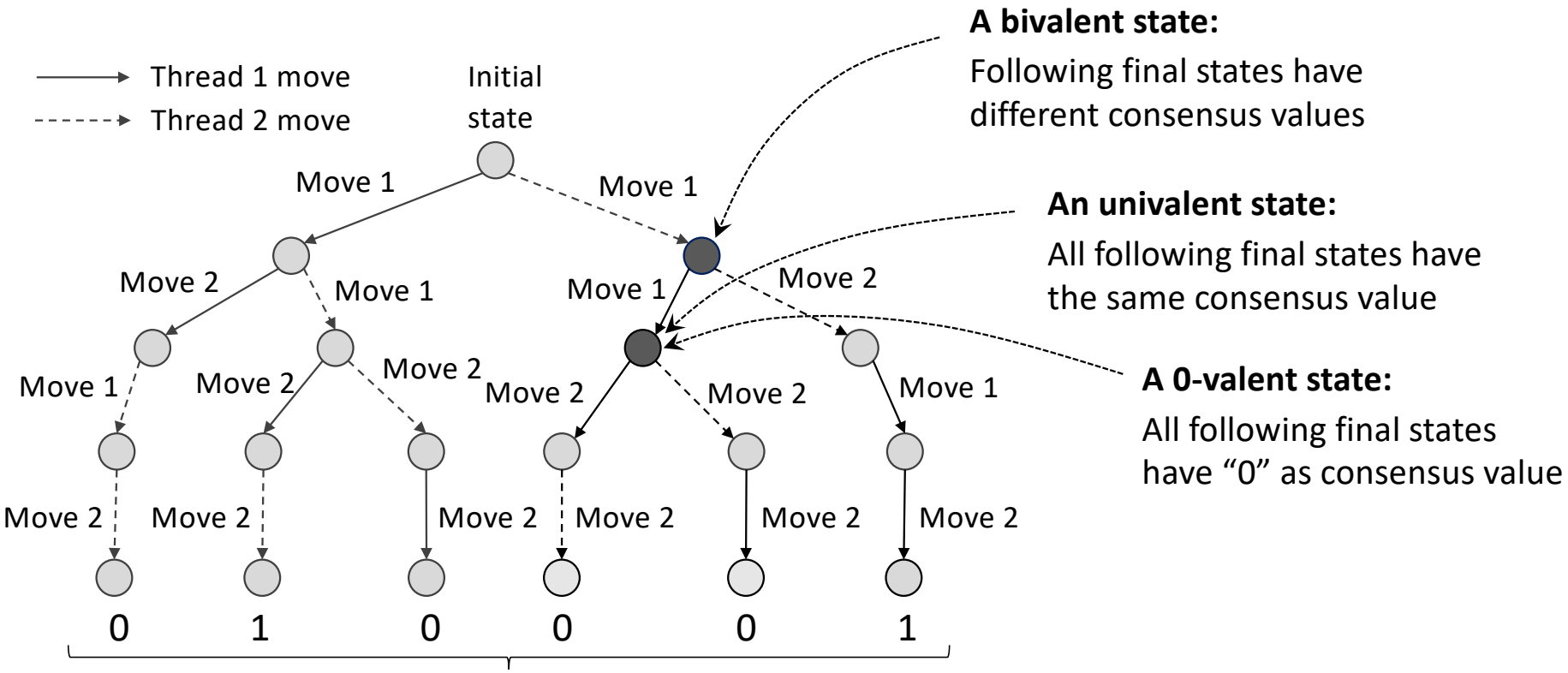
- An univalent state: all final states, following from this state, have the same decision (consensus) value
- A bivalent state: the decision value for a consensus problem is not yet fixed
- An X-valent state: an univalent state with decision value X




(We shuffled decision values compared to a previous slide)

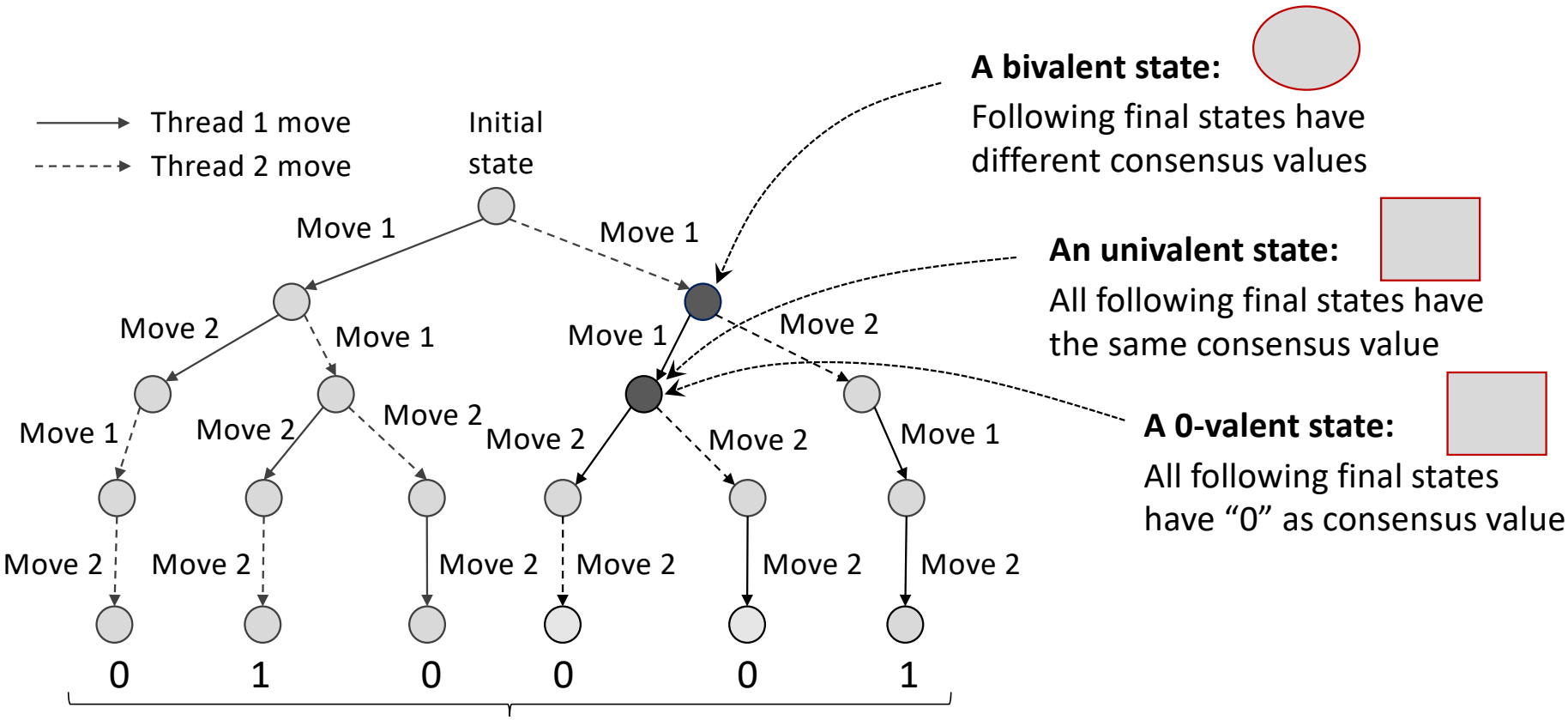
# The Notion of a Protocol State Valency

- An univalent state: all final states, following from this state, have the same decision (consensus) value
- Bivalent state: the decision value for a consensus problem is not yet fixed
- X-valent state: an univalent state with decision value X



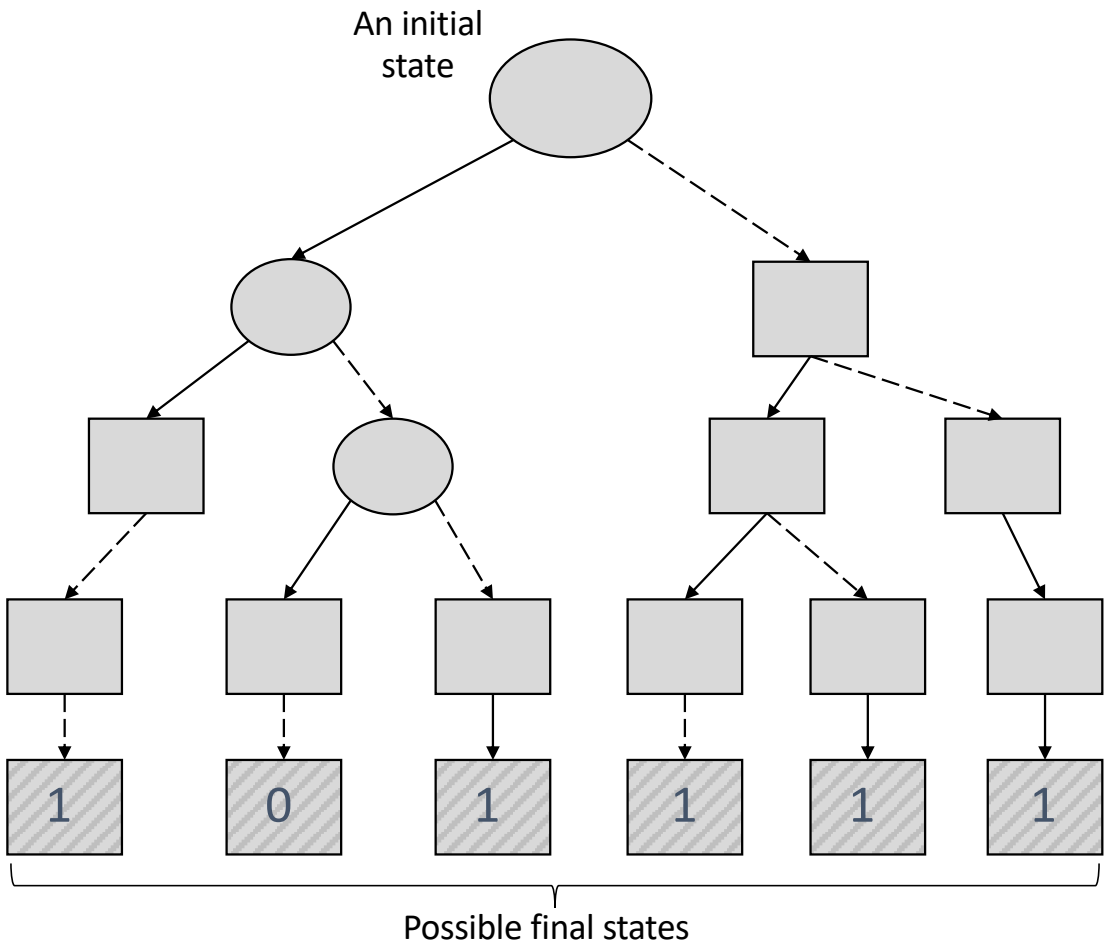

(We shuffled decision values compared to a previous slide)

# Graphical Notation for Univalent and Bivalent States




(We shuffled decision values compared to a previous slide)

# A Sample Execution Tree for a 2-thread Binary Consensus Problem




Types of protocol states:

- Bivalent: the decision value is not yet fixed
- Univalent: the decision value is fixed; x-valent: the decision value is x
- X - Final; x – the decision value

Edges (thread moves):

- Thread A moves
- Thread B moves

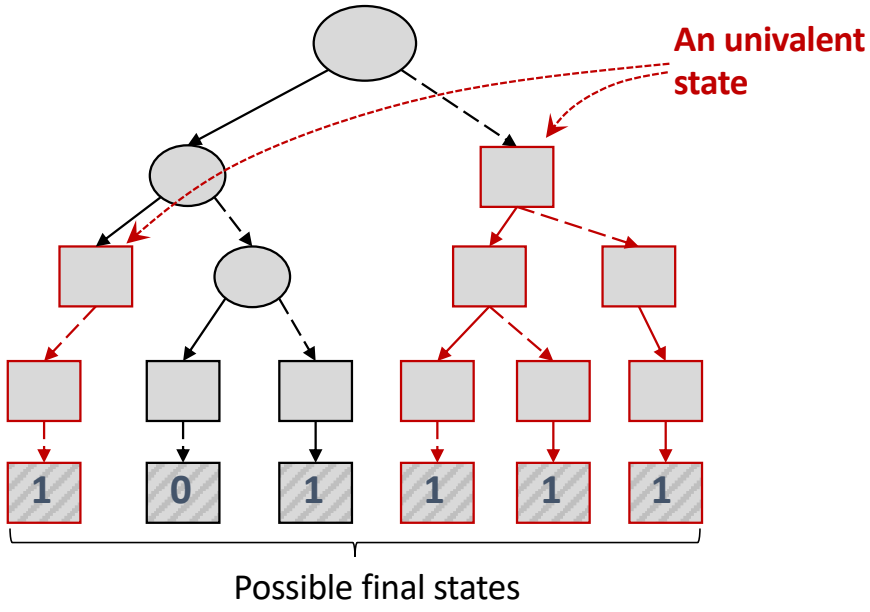
Assumption for this sample tree: Each thread moves only twice

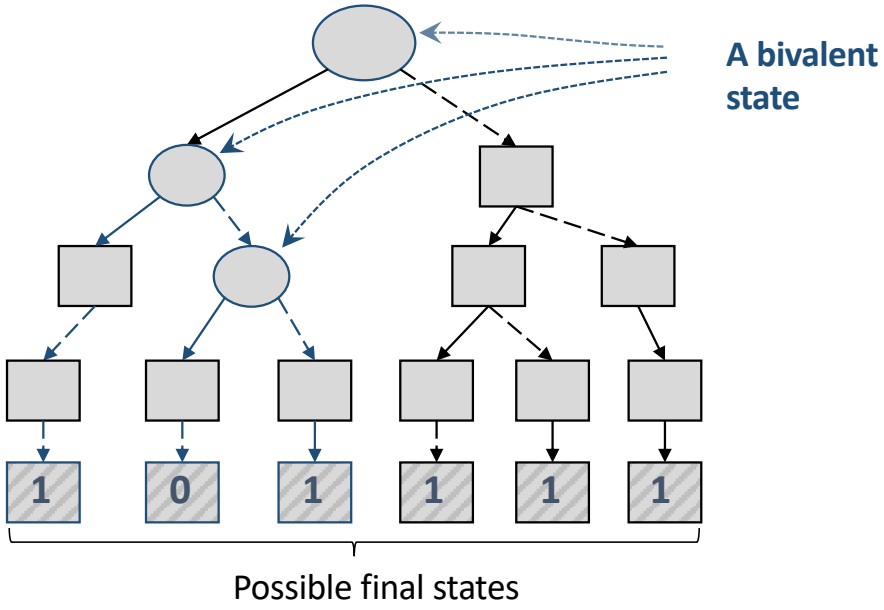
# State Valency: Univalent and Bivalent States

The notion of a state valency (bivalent or univalent) is used to prove various theorems

Properties:

- All final states, that follow from any **univalent** state, correspond to the same decision value
- Final states, that follow from any **bivalent** state, correspond to different decision values



# Consensus Numbers for Some Concurrent Objects

| **Object Type** | **Consensus Number** |
|---|---|
| Read-write register (safe, regular, or atomic type) | 1 |
| Memory snapshot primitive | 1 |
| FIFO, double ended, or priority queue | 2 |
| Set | 2 |
| An atomic (n, n(n+1)/2) – register assignment | At least n |
| An atomic multiple assignment into m registers | 2m - 2 |
| A read-modify-write (RMW) register | At least 2 |
| A register supporting CAS operation | Infinite |
| An extended queue | Infinite |