K22 – Operating Systems: Design Principles and Internals

Fall 2026 @dit

Vaggelis Atlidakis

Lecture 11

References: Similar OS courses @Columbia, @Stanford, @UC San Diego, @Brown, @di (previous years); and textbooks: Operating Systems: Three Easy Pieces, Operating Systems: Principles and Practice, Operating System Concepts, Linux Kernel Development, Understanding the Linux Kernel

Overview

- We'll start from hardware and follow a question-oriented approach
  - ~~Intro [Q: What is an OS?]~~
  - ~~Events [Q: When does the OS run?]~~
  - ~~Runtime [Q: How does a program look like in memory?]~~
  - ~~Processes [Q: What is a process?]~~
  - ~~IPC [Q: How do processes communicate?]~~
  - ~~Threads [Q: What is a thread?]~~
  - Synchronization [Q: What goes wrong w/o synchronization?]
  - Time Management [Q: What is scheduling?]
  - Memory Management [Q: What is virtual memory?]
  - Files [Q: What is a file descriptor?]
  - Storage Management [Q: How do we allocate disk space to files?]

Overview

- We'll start from hardware and follow a question-oriented approach
  - ~~Intro [Q: What is an OS?]~~
  - ~~Events [Q: When does the OS run?]~~
  - ~~Runtime [Q: How does a program look like in memory?]~~
  - ~~Processes [Q: What is a process?]~~
  - ~~IPC [Q: How do processes communicate?]~~
  - ~~Threads [Q: What is a thread?]~~
  - **Synchronization [Q: What is synchronization?]**
  - Time Management [Q: What is scheduling?]
  - Memory Management [Q: What is virtual memory?]
  - Files [Q: What is a file descriptor?]
  - Storage Management [Q: How do we allocate disk space to files?]

* Basic (H/W & S/W)
* Abstractions
* **Primitives**
* Mechanisms

Necessary glossary to talk about *synchronization*

- **Parallel operations:** Operations that are happening at the same time, on different processors
- **Concurrent operations:** Operations that are happening in overlapping time intervals, on the same system, seemingly simultaneously
- **Interleaving of execution:** The order with which concurrent operations are scheduled in for and out of execution

What is *synchronization*?

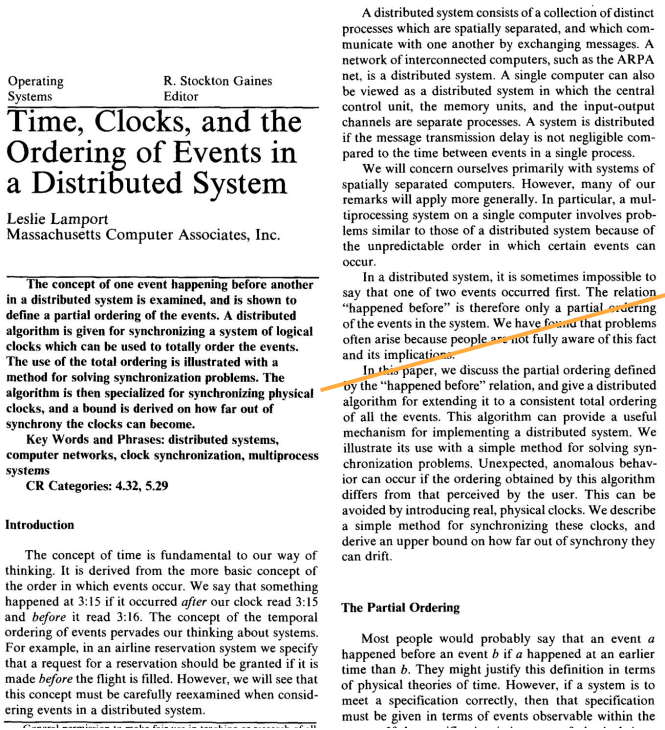
Given two concurrent operations p1, p2 with a "dependency" such that p1 must always "happen before" p2, synchronization mandates that $t1(i) < t2(i)\ \forall\ i \in [1, n]$, where

t1(i) is the time when p1 ends its i-th execution

t2(i) it the time when p2 starts its i-th execution

> We have a problem in the above definition…

"happens before" on distributed systems



Operating Systems — R. Stockton Gaines, Editor

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

Key Words and Phrases: distributed systems, computer networks, clock synchronization, multiprocess systems
CR Categories: 4.32, 5.29

Introduction

The concept of time is fundamental to our way of thinking. It is derived from the more basic concept of the order in which events occur. We say that something happened at 3:15 if it occurred *after* our clock read 3:15 and *before* it read 3:16. The concept of the temporal ordering of events pervades our thinking about systems. For example, in an airline reservation system we specify that a request for a reservation should be granted if it is made *before* the flight is filled. However, we will see that this concept must be carefully reexamined when considering events in a distributed system.

A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. A single computer can also be viewed as a distributed system in which the central control unit, the memory units, and the input-output channels are separate processes. A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process.

We will concern ourselves primarily with systems of spatially separated computers. However, many of our remarks will apply more generally. In particular, a multiprocessing system on a single computer involves problems similar to those of a distributed system because of the unpredictable order in which certain events can occur.

In a distributed system, it is sometimes impossible to say that one of two events occurred first. The relation "happened before" is therefore only a partial ordering of the events in the system. We have found that problems often arise because people are not fully aware of this fact and its implications.

In this paper, we discuss the partial ordering defined by the "happened before" relation, and give a distributed algorithm for extending it to a consistent total ordering of all the events. This algorithm can provide a useful mechanism for implementing a distributed system. We illustrate its use with a simple method for solving synchronization problems. Unexpected, anomalous behavior can occur if the ordering obtained by this algorithm differs from that perceived by the user. This can be avoided by introducing real, physical clocks. We describe a simple method for synchronizing these clocks, and derive an upper bound on how far out of synchrony they can drift.

The Partial Ordering

Most people would probably say that an event *a* happened before an event *b* if *a* happened at an earlier time than *b*. They might justify this definition in terms of physical theories of time. However, if a system is to meet a specification correctly, then that specification must be given in terms of events observable within the

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

> Time, Clocks, and the Ordering of Events in a Distributed System, 1978, by Leslie Lamport

Departing from temporal ordering

> Temporal ordering: Arrangement of events in a sequence according to physical time

- What most human understand when you talk about time!
- Example: An airline reservation request will be granted if (i) it is made before the flight is filled, and (ii) before the flight departs

> Ordering is not temporal on a multiprocessor (distributed) system

- Any conversation is in terms of physical time must be reexamined when considering concurrent events in a distributed system
- Real clocks are not perfectly accurate ⇒ can't keep precise phys. time

Partial ordering of concurrent operations

> Logical clocks allows us to define a partial ordering of concurrent operations on a multiprocessor system

> Synchronization is used to enforce that some partial order of concurrent operations (i.e., some "happens-before" relationship) exists

Given two concurrent operations p1, p2
with a "dependency" such that p1 must always "happen before" p2,
synchronization mandates that $t1(i) < t2(i)\ \forall\ i \in [1, n]$, where
$t1(i)$ is the time when p1 ends its i-th execution
$t2(i)$ is the time when p2 starts its i-th execution

Reasoning about concurrent code

> Concurrent progs have too many execution interleavings

- Too many ways something erroneous could happen
- Need to explore an enormous state space

> Correctness needs a definite and **complete** answer

- Inspected 100% of the state space ⇒ Can make an assessment
- Inspect 99.9% of the state space ⇒ Can't make any assessment

Reasoning about concurrent code

- Correctness needs a definite and **complete** answer
- If we inspect 100% of the state space, we can make an assessment
- If we inspect 99.9% of the state space, the 0.1% makes us unhappy
- Hard to track all the feasible ways by which something erroneous could happen: **too many** execution interleavings

Permutations of the word "MISSISSIPPI"

> We are counting permutations
  - Let's do the exercise

> M-I-S-S-I-S-S-I-P-P-I
  - Length: 11, M: 1, I: 4, S: 4, P: 2
  - Distinct ways to permute a multiset of n elements, where ki is the multiplicity of the ith element?
  - Multinomial coefficient: (k1+k2+...+ kn)! / (k1!*k2!*...*kn!)
  - Permutations of MISSISSIPPI = (11!) / (1!4!4!2!) = **34,650**

How many is "too many"?

> Different schedules for four operations P1, P2, P3, and P4, which run in total 11 time quanta; and where

- P1 runs 1 time
- P2 runs 4 times
- P3 runs 4 times
- P4 runs 2 times

> How many different scheduler plans do we need to inspect, to cover the complete state space of possible interleavings?

- P1-P2-P3-P3-P2-P3-P3-P2-P4-P4-P2 => **34,650**

* Trivial example in terms of no of operations

- **We are not considering myriads of async events**
- **Yet's it's already too difficult**

Reasoning with Sequential Consistency

How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs

LESLIE LAMPORT

Abstract—**Many large sequential computers execute operations in a different order than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a correct execution by each processor does not guarantee the correct execution of the entire program. Additional conditions are given which do guarantee that a computer correctly executes multiprocess programs.**

Index Terms—**Computer design, concurrent computing, hardware correctness, multiprocessing, parallel processing.**

A high-speed processor may execute operations in a different order than is specified by the program. The correctness of the execution is guaranteed if the processor satisfies the following condition: the result of an execution is the same as if the operations had been executed in the order specified by the program. A processor satisfying this condition will be called *sequential*. Consider a computer composed of several such processors accessing a common memory. The customary approach to designing and proving the correctness of multiprocess algorithms [1]–[3] for such a computer assumes that the following condition is satisfied: the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the

Abstract—**Many large sequential computers execute operations in a different order than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a correct execution by each processor does not guarantee the correct execution of the entire program. Additional conditions are given which do guarantee that a computer correctly executes multiprocess programs.**

of each individual processor does not guarantee that the multiprocessor computer is sequentially consistent. In this brief note, we describe a method of interconnecting sequential processors with memory modules that insures the sequential consistency of the resulting multiprocessor.

> How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs, 1977, by Leslie Lamport

Necessary glossary to talk about *synchronization*

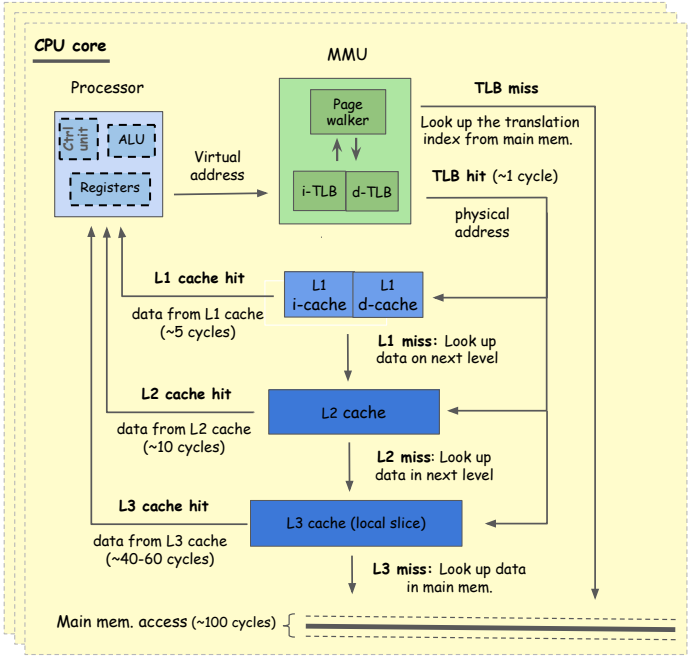
- **Parallel operations:** Operations that are happening at the same time, on different processors
- **Concurrent operations:** Operations that are happening in overlapping time intervals seemingly simultaneously
- **Interleaving of execution:** The order with which concurrent operations are scheduled in for and out of execution
- **Happens-before relationship:** A *partial ordering* of concurrent operations of a program
- **Sequential consistency:** The result of any execution of concurrent operations is the same as if all operations on all processors were executed in some sequential (global) order, and the operations of each individual processor appear in this sequence in the order specified by its program

Necessary glossary to talk about *synchronization*

- **Parallel operations:** Operations that are happening at the same time, on different processors
- **Concurrent operations:** Operations that are happening in overlapping time intervals seemingly simultaneously
- **Interleaving of execution:** The order with which concurrent operations are scheduled in for and out of execution
- **Happens-before relationship:** A *partial ordering* of concurrent operations of a program
- **Sequential consistency:** The result of any execution of concurrent operations is the same as if all operations on all processors were executed in some sequential (global) order, and the operations of each individual processor appear in this sequence in the order specified by its program

Sequential consistency: Requirements

- **Sequential consistency:** Every load from a memory address would get its value from the last store before it to the same address in **global memory**



> Easy to reason / Impractically slow

- The effects of each instruction must be visible on all cores before starting the next instruction

- The first level of "global" memory is the L3 cache with an overhead of at least 40 cycles for access time

- In practice: We relax the memory consistency model to hide store (write) latency and avoid processor stalls

"Problems" due to lack of synchronization

> Race conditions: A timing dependent error involving shared state which occurs when the interleaving of execution of concurrent operations leads to erroneous program behaviour

> **Reasons for race conditions:**

- Data races: Non-atomic, unsynchronized, concurrent operations, at least one of which mutating shared state
- Semantic ordering errors: Code that does not enforce the order programmers intended to for a group of memory accesses
- Weak memory consistency models: The set of allowed behaviours w.r.t. memory operation is not what the programer expected

"Problems" due to lack of synchronization

> Race conditions: A timing dependent error involving shared state which occurs when the interleaving of execution of concurrent operations leads to erroneous program behaviour

> **Reasons for race conditions:**

- Data races: Non-atomic, unsynchronized, concurrent operations, at least one of which mutating shared state
- Semantic ordering errors: Code that does not enforce the order programmers intended to for a group of memory accesses
- Weak memory consistency models: The set of allowed behaviours w.r.t. memory operation is not what the programer expected

Data races

> A program contains a data race iif two or more threads

(1) access the same memory location concurrently

AND (2) at least one of these accesses is a write

AND (3) at least one of the accesses is not atomic

AND (4) neither happens before the other

Such data races may result in undefined program behaviors and may lead to unforeseen errors at runtime

- See the ISO/IEC 9899:2011(C11), Sec.-5.1.2.4/25, on multi-threaded executions and data races

Data races

```c
int total = 0;

void *add(void *arg) {
  for (int i = 0; i < 1e6; ++i)
    ++total;
  return NULL;
}

void main() {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, add, (void *) NULL);
  pthread_create(&t2, NULL, add, (void *) NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("Total-1: %d\n", total);

  total = 0;
  pthread_create(&t1, NULL, add, (void *) NULL);
  pthread_join(t1, NULL);
  pthread_create(&t2, NULL, add, (void *) NULL);
  pthread_join(t2, NULL);
  printf("Total-2: %d\n", total);
}
```

```
➜ obdjump -d ./counter

0000000000001159 <add>:

 1159: push %rbp                  # Save base pointer to stack
 115a: mov  %rsp, %rbp            # Set up new stack frame
 115d: mov  %rdi, -0x18(%rbp)     # *arg = %rdi
 1161: movl $0x0, -0x4(%rbp)      # i = 0
 1168: jmp  117d <add+0x24>       # for-loop start

                                  Data race!
 116a: mov  0x2ebc(%rip), %eax    #  %eax ← total
 1170: add  $0x1, %eax            #  %eax += 1
 1173: mov  %eax, 0x2eb3(%rip)    #  total ← %eax

 1179: addl $0x1, -0x4(%rbp)      # i += 1
 117d: cmpl $0xf423f, -0x4(%rbp)  # loop counter compare
 1184: jle  116a <add+0x11>       # for-loop jump
 1186: mov  $0x0, %eax            # rval = %eax
 118b: pop  rbp                   # Restore stack
 118c: ret                        # Return to caller
```

```
➜ git:(master) ✗ ./counter
Total-1: 1011367
Total-2: 2000000

➜ git:(master) ✗ ./counter
Total-1: 1011367
Total-2: 2000000

➜ git:(master) ✗ ./counter
Total-1: 1028085
Total-2: 2000000

➜ git:(master) ✗ ./counter
Total-1: 1011197
Total-2: 2000000

➜ git:(master) ✗ ./counter
Total-1: 1018502
Total-2: 2000000

➜ git:(master) ✗ ./counter
Total-1: 1013853
Total-2: 2000000
```

Data races

- Two domains of state for each thread

- Processor on thread A
  - Focus on: Processor regs (per-thread, local state) vs. Main memory (global state)
  - %reg <- value at main mem. [global vs. local state: consistent]
  - %reg <- %reg + 1 [global vs. local state: divergent]
  - value at main mem. <- %reg [global vs. local state: consistent]
- Thread A gets preempted [shared value at main mem. +1]
- Processor on thread B
  - Focus on: Processor regs (per-thread, local state) vs. Main memory (global state)
  - %reg <- value at main mem. [global vs. local state: consistent]
  - %reg <- %reg + 1 [global vs. local state: divergent]
  - value at main mem. <- %reg [global vs. local state: consistent]
- Thread B gets preempted [shared value at main mem. +1]

OK Interleaving

Data races

> Another execution interleaving

- Processor on thread A
  - %reg <- value at main mem. [global vs. local state: consistent]
- Processor on thread B
  - %reg <- value at main mem. [global vs. local state: consistent]
  - %reg <- %reg + 1 [global vs. local state: divergent]
  - value at main mem. <- %reg [global vs. local state: consistent]
- Processor back on thread A
  - **OS job**: Load %reg with its value before ctx switch
  - Thead B "thinks" %reg ⇔ shared value at main mem.
  - **But**: shared value has been mutated by someone else

NOT OK Interleaving

Data races

- **Bottom-line**: Concurrent writes on shared state?
  - Each thread must finish its business before it gets preempted
  - Inseparable "instructions" ⇒ **Atomic operations**

- Processor on thread A
  - %reg <- value at main mem. [global vs. local state: consistent]
  - %reg <- %reg + 1 [global vs. local state: divergent]
  - value at main mem. <- %reg [global vs. local state: consistent]

Can these instructions become "inseparable"

- Processor on thread B
  - %reg <- value at main mem. [global vs. local state: consistent]
  - %reg <- %reg + 1 [global vs. local state: divergent]
  - value at main mem. <- %reg [global vs. local state: consistent]

Can these instructions become "inseparable"

"Problems" due to lack of synchronization

> Race conditions: A timing dependent error involving shared state which occurs when the interleaving of execution of concurrent operations leads to erroneous program behaviour

> **Reasons for race conditions:**

- Data races: Non-atomic, unsynchronized concurrent accesses, at least one of which mutating a shared variable
- Semantic ordering errors: Code that does not enforce the order programmers intended to for a group of memory accesses
- Weak memory consistency models: The set of allowed behaviours w.r.t. memory operations is not what the programer expected

"Problems" due to lack of synchronization

> Race conditions: A timing dependent error involving shared state which occurs when the interleaving of execution of concurrent operations leads to erroneous program behaviour

> **Reasons for race conditions:**

- Data races: Non-atomic, unsynchronized concurrent accesses, at least one of which mutating a shared variable
- Semantic ordering errors: Code that does not enforce the order programmers intended to for a group of memory accesses
- Weak memory consistency models: The set of allowed behaviours w.r.t. memory operations is not what the programer expected

Semantic ordering errors: Benign

```c
void* func1(void* arg) {
  printf("1\n");
  return NULL;
}

void* func2(void* arg) {
  printf("2\n");
  return NULL;
}

int main(void) {
  pthread_t t1, t2;

  pthread_create(&t1, NULL, func1, NULL);
  pthread_create(&t2, NULL, func2, NULL);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  return 0;
}
```

```
➜ concurrency git:(master) ✗ ./nosync
1
2

➜ concurrency git:(master) ✗ ./nosync
1
2

➜ concurrency git:(master) ✗ ./nosync
1
2

➜ concurrency git:(master) ✗ ./nosync
1
2

➜ concurrency git:(master) ✗ ./nosync
1
2
```

Semantic ordering errors: Detrimental (See Therac-25)

```c
int mode = 0;              // 1: Low-power; 2: High-power
int filter_engage = 0;     // 0: Filter not engaged; 1: Filter engaged
int input_finalized = 0;   // Has operator finished input?

void *filter_control(void *arg) {
  while (!input_finalized) {
    sched_yield();
  }
  usleep(100);
  if (mode == 2)
    filter_engage = 1;
  else
    filter_engage = 0;
  return NULL;
}

void beam_activate() {
  if (mode == 2 && filter_engage == 0)
    printf("🚨 Treatment with high-power beam and no filter in place\n");
  else
    printf("🟢 Safe setup\n");
}

int main(void) {
  pthread_t filter_control_t;
  pthread_create(&filter_control_t, NULL, filter_control, NULL);
  usleep(100);  // Time window 1:  operator does  initial setup
  mode = 1;
  input_finalized = 1;
  usleep(100); // Time window 2: operator does  final edits
  mode = 2;
  pthread_join(filter_control_t, NULL); // Control logic completed
  beam_activate();
}
```

```
➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.
➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.
➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.
➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.
➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.
➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.
➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.
➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.
➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.
➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.
➜ concurrency git:(master) ✗ ./therac25
🚨 Treatment with high-power beam and no filter in place :-(
```

"Problems" due to lack of synchronization

> Race conditions: A timing dependent error involving shared state which occurs when the interleaving of execution of concurrent operations leads to erroneous program behaviour

> **Reasons for race conditions:**

- Data races: Non-atomic, unsynchronized concurrent accesses, at least one of which mutating a shared variable
- Semantic ordering errors: Code that does not enforce the order programmers intended to for a group of memory accesses
- Weak memory consistency models: The set of allowed behaviours w.r.t. memory operations is not what the programer expected

"Problems" due to lack of synchronization

> Race conditions: A timing dependent error involving shared state which occurs when the interleaving of execution of concurrent operations leads to erroneous program behaviour

> **Reasons for race conditions:**

- Data races: Non-atomic, unsynchronized concurrent accesses, at least one of which mutating a shared variable
- Semantic ordering errors: Code that does not enforce the order programmers intended to for a group of memory accesses
- Weak memory consistency models: The set of allowed behaviours w.r.t. memory operations is not what the programer expected

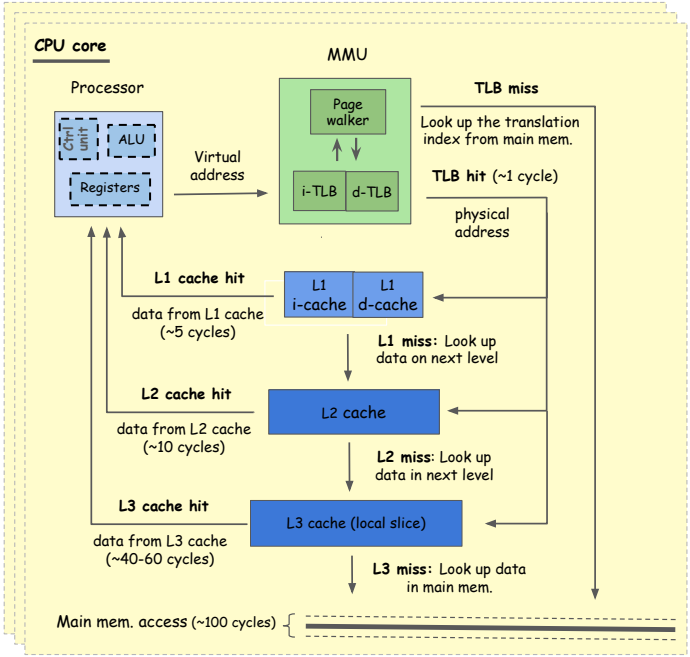
Sequential consistency: Reminder

Pragmatic explanation

- Processors issue loads and stores in their local memory respecting their local program order
- Every load from a memory address gets its value from the last store before it, on the same memory address, in global memory

Relaxing sequential consistency

- **Sequential consistency:** Every load from a memory address gets its value from the last store before it to the same address in **global memory**



> Easy to reason / Impractically slow

- The effects of each instruction must be visible on all cores before starting the next instruction

- The first level of "global" memory is the L3 cache with an overhead of at least 40 cycles for access time

- **In practice: We relax the memory consistency model to hide write latency and avoid processor stalls**

Relaxing sequential consistency

- **x86 Total Store Order (TSO)**
  - "Stores" are ordered between cores
  - "Store-Load" pairs can be reordered between cores

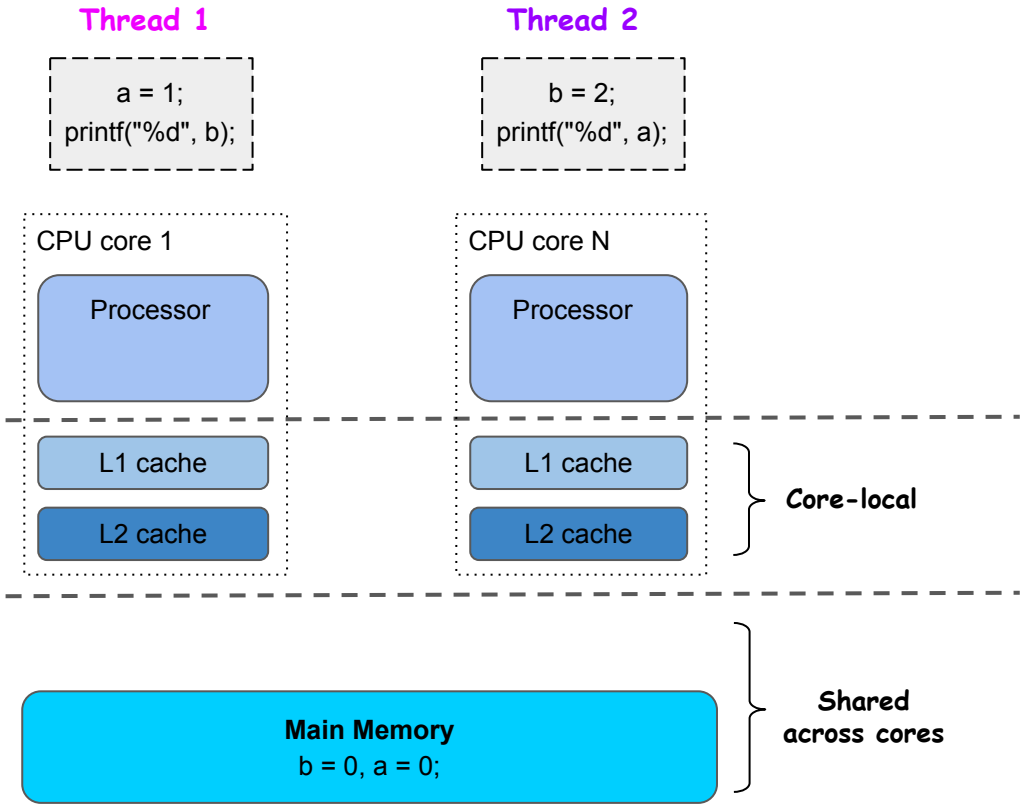
Relaxing sequential consistency

- **x86 Total Store Order (TSO)**
  - "Stores" are ordered between cores
  - "Store-Load" pairs can be reordered between cores
- **Why?** Store latency
  - Before committing data on a core-local cache line (L1 or L2 cache), a core must wait for the cache lines to be invalidated on all other cores
  - The hardware does this via cache coherency protocols
  - Latency of store instructions even on core-local mem. references

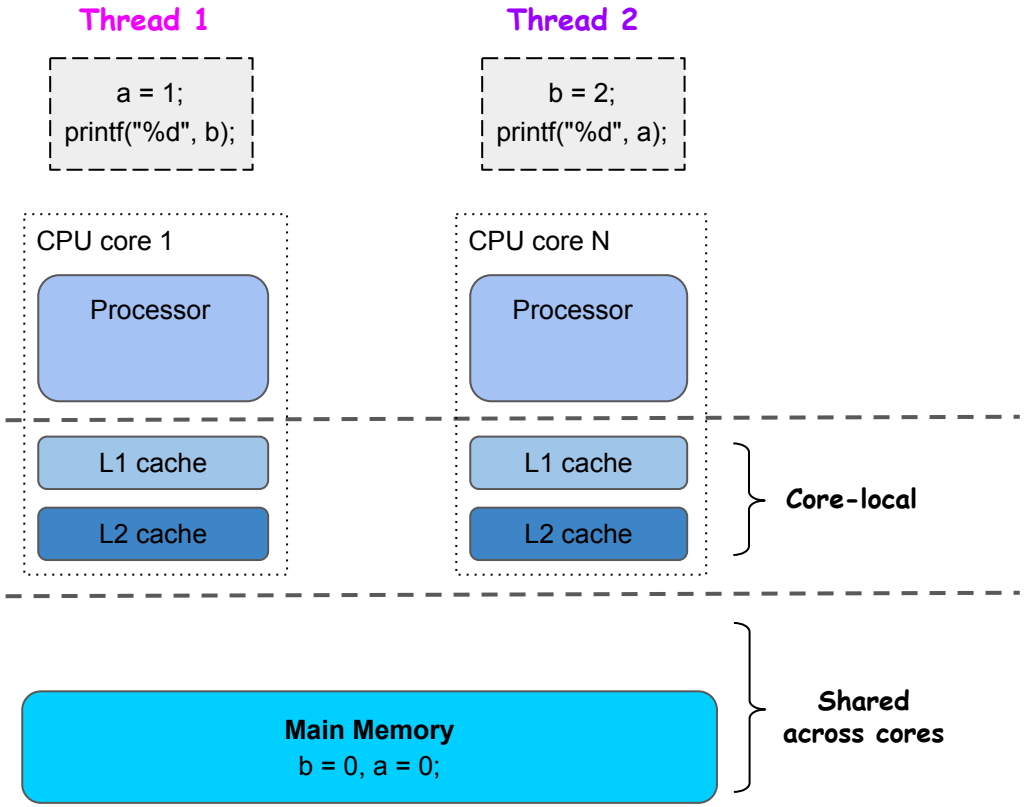
Relaxing sequential consistency

- **x86 Total Store Order (TSO)**
  - "Stores" are ordered between cores
  - "Store-Load" pairs can be reordered between cores
- **Why?** Store latency
  - Before committing data on a core-local cache line (L1 or L2 cache), a core must wait for the cache lines to be invalidated on all other cores
  - The hardware does this via cache coherency protocols
  - Latency of store instructions even on core-local mem. references
- Solution
  - Use of processor-internal write buffers to hide store latency
  - Memory model violates sequential consistency
  - Reorderings of operations ⇒ potential for unexpected program behaviour

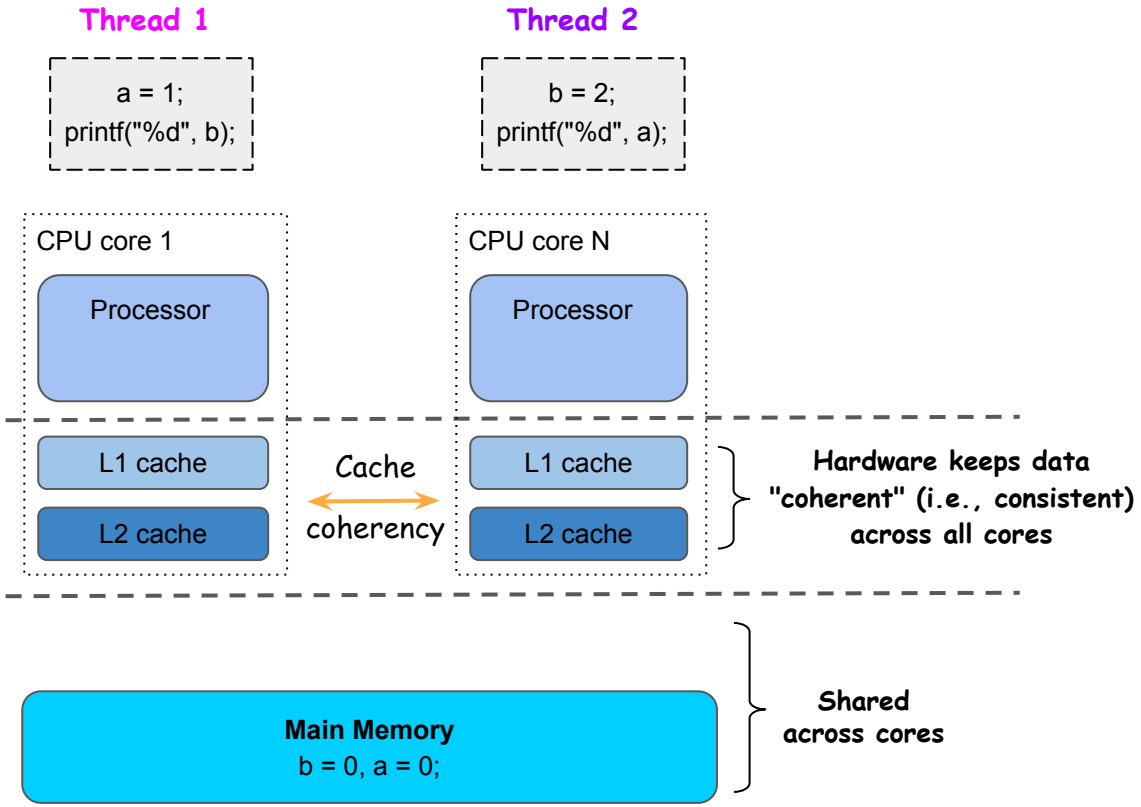
Can this program print "00"?



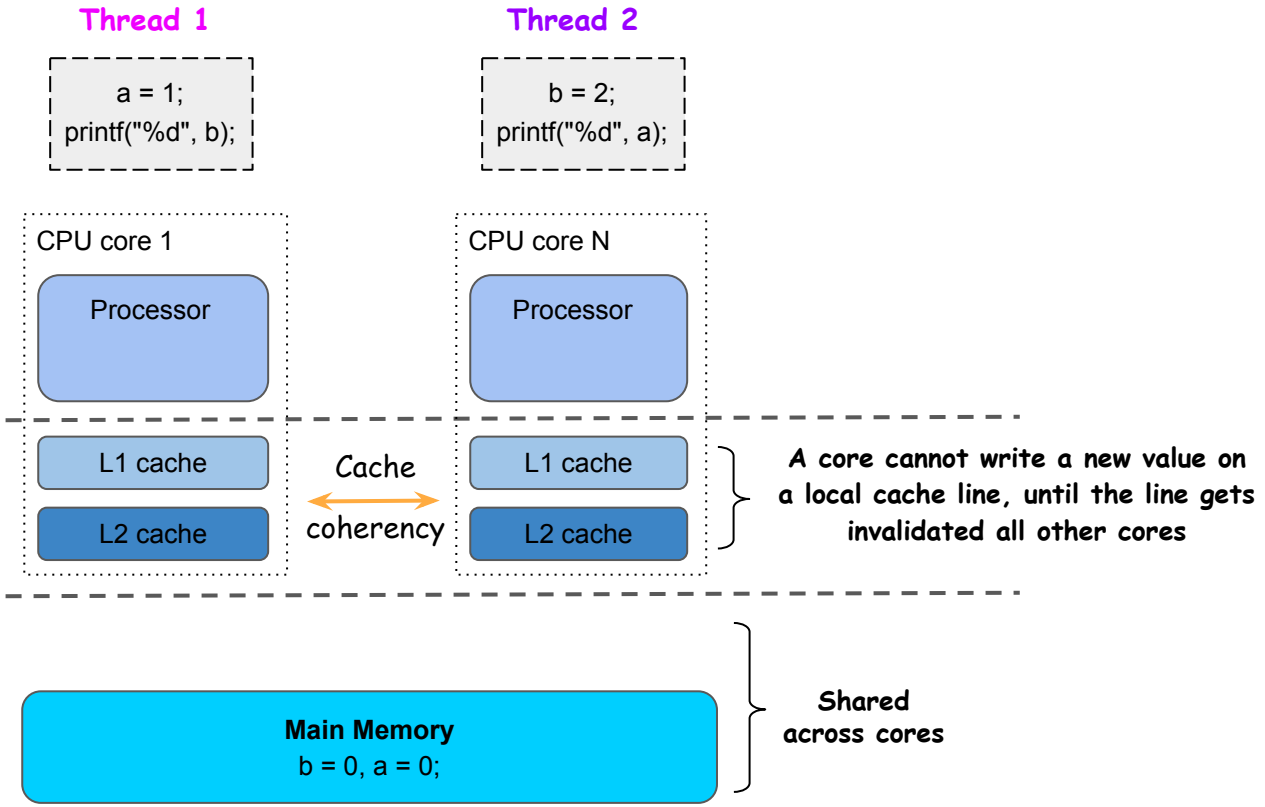
Can this program print "00"? We do **not** know…



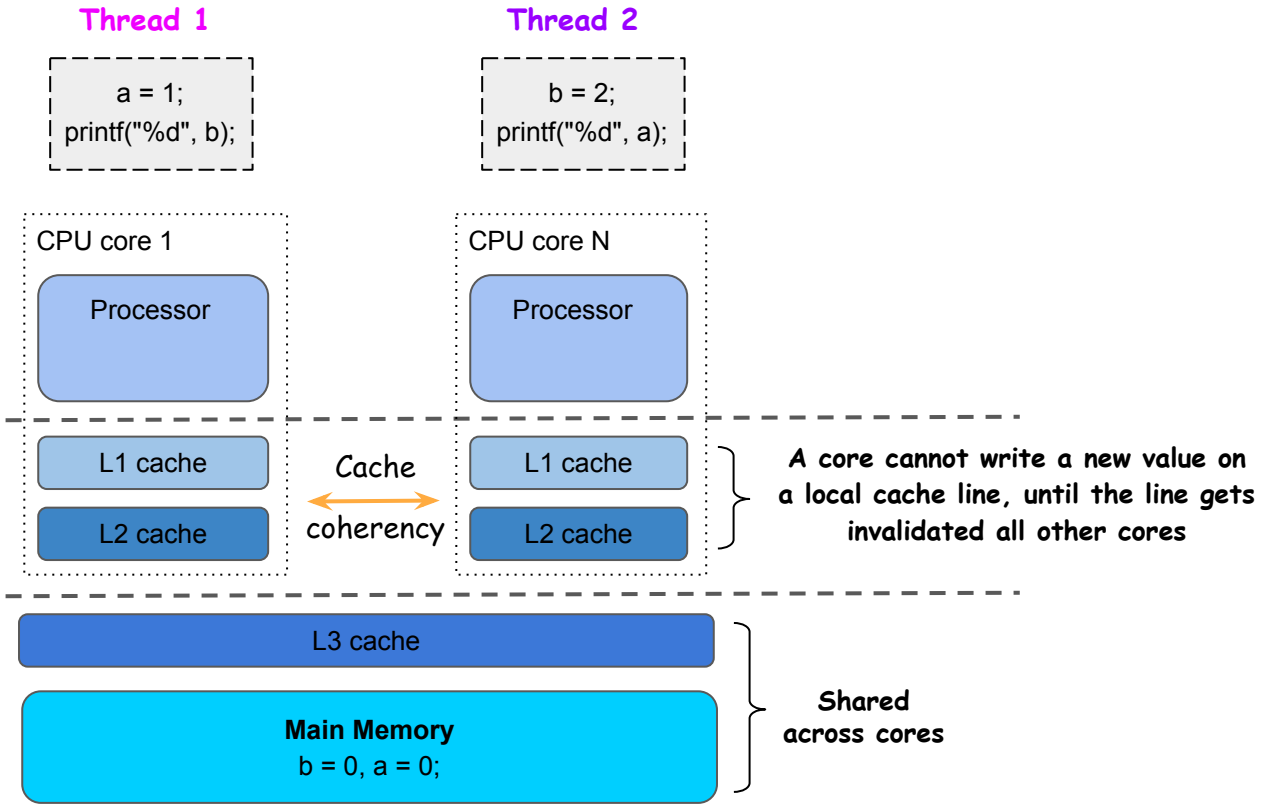
x86 Total Store Order (TSO)



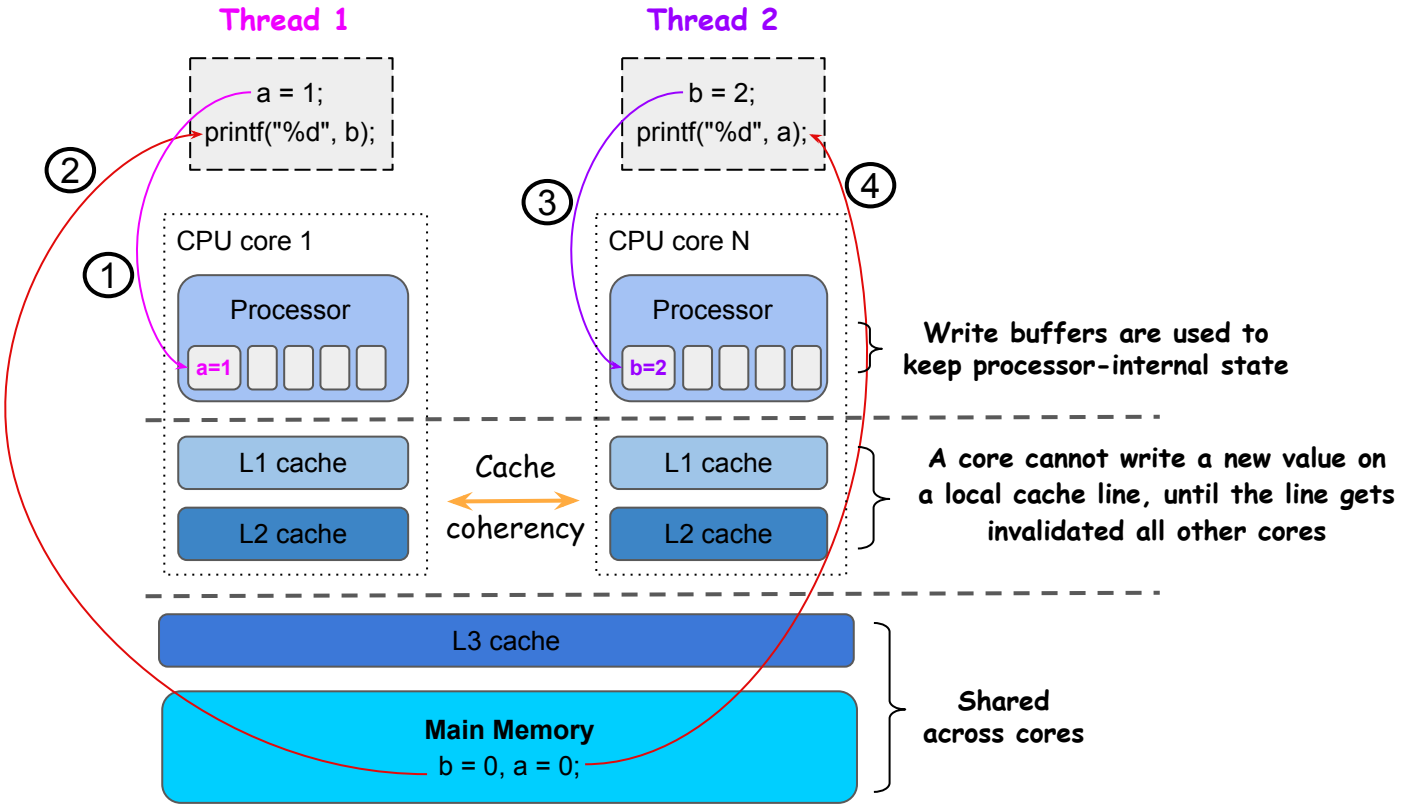
x86 Total Store Order (TSO)



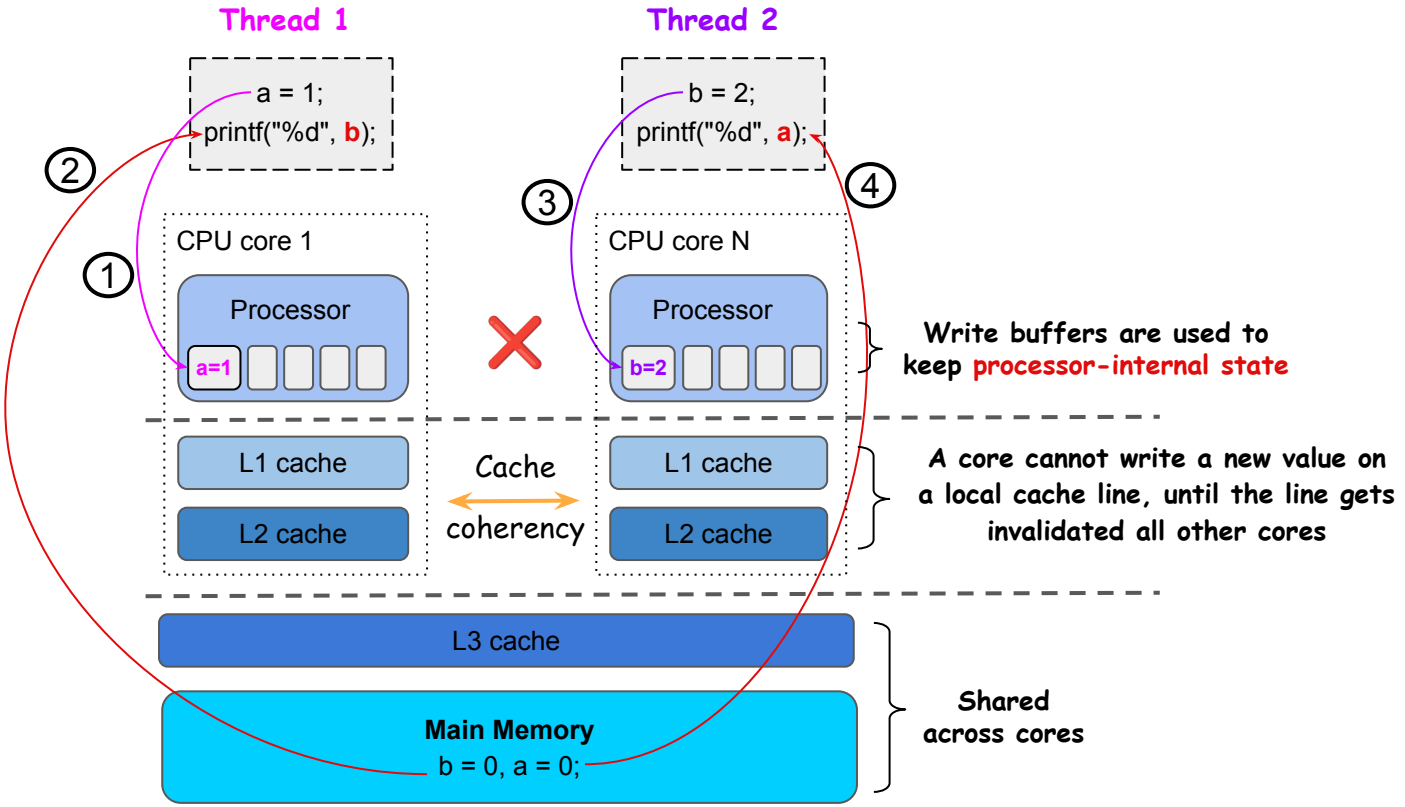
x86 Total Store Order (TSO)



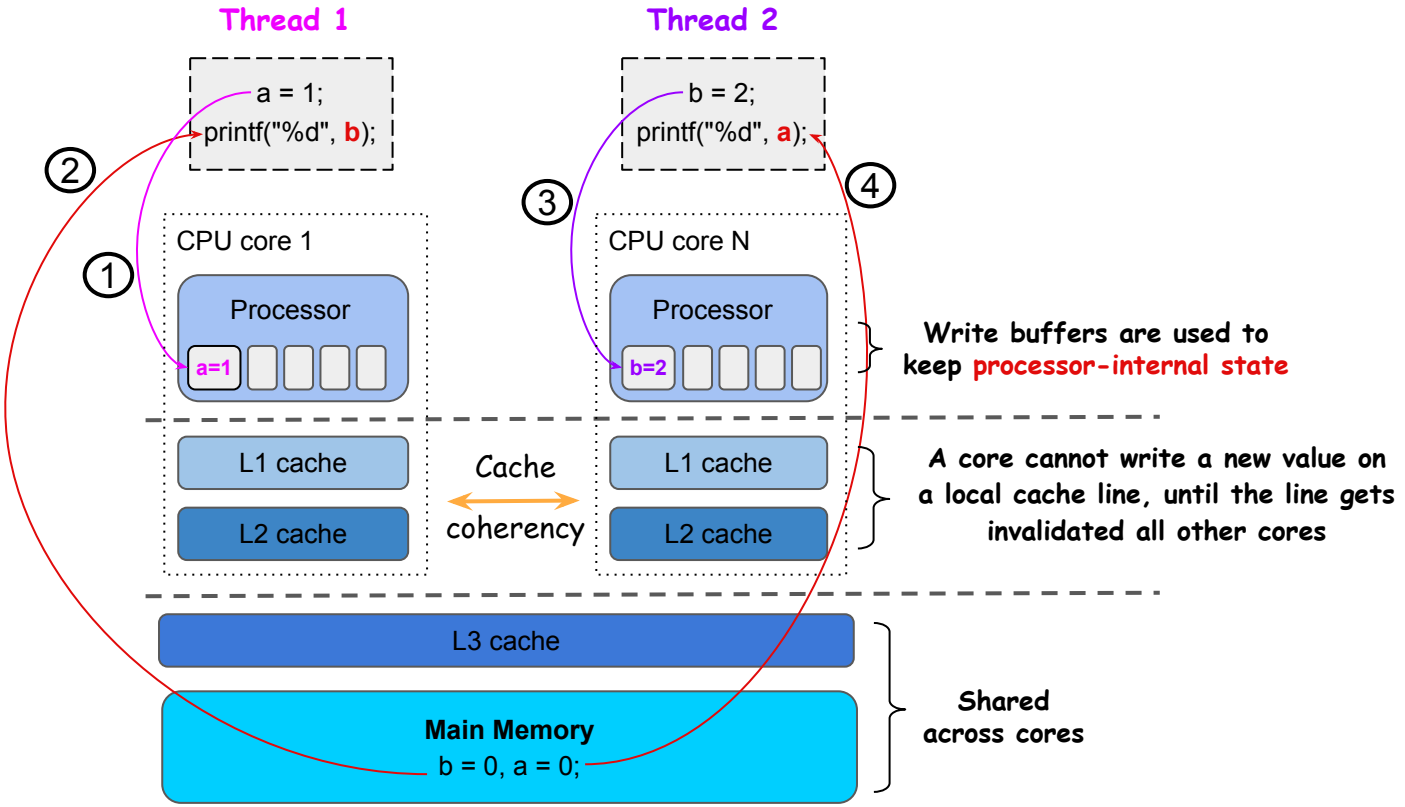
x86 Total Store Order (TSO)



x86 Total Store Order (TSO)



x86 Total Store Order (TSO)



Memory Barriers

Memory Barriers

- Architecture-specific instructions, which explicitly prevent compiler- and processor-dependent reorderings of memory ops.

Memory Barriers

- Architecture-specific instructions, which explicitly prevent compiler- and processor-dependent reorderings of memory ops.
- x86 memory barrier instructions (Linux kernel macros)

Memory Barriers

> Architecture-specific instructions, which explicitly prevent compiler- and processor-dependent reorderings of memory ops.

> x86 memory barrier instructions (Linux kernel macros)

- mfencle: Prevents reordering of loads and stores; ensures that all memory reads and writes before the mfence, complete before any that follow.

Memory Barriers

> Architecture-specific instructions, which explicitly prevent compiler- and processor-dependent reorderings of memory ops.

> x86 memory barrier instructions (Linux kernel macros)

- mfencle: Prevents reordering of loads and stores; ensures that all memory reads and writes before the mfence, complete before any that follow.
- lfence: Prevents reordering of loads before loads; ensures all prior loads complete before any subsequent load.

Memory Barriers

> Architecture-specific instructions, which explicitly prevent compiler- and processor-dependent reorderings of memory ops.

> x86 memory barrier instructions (Linux kernel macros)

- mfencle: Prevents reordering of loads and stores; ensures that all memory reads and writes before the mfence, complete before any that follow.
- lfence: Prevents reordering of loads before loads; ensures all prior loads complete before any subsequent load.
- sfence: Prevents reordering of stores before stores; ensures all prior stores complete before any subsequent store

Recap

> Race conditions: A timing dependent error involving shared state which occurs when the interleaving of execution of concurrent operations leads to erroneous program behaviour

Recap

> Race conditions: A timing dependent error involving shared state which occurs when the interleaving of execution of concurrent operations leads to erroneous program behaviour

> **Reasons for race conditions:**

- Data races: Non-atomic, unsynchronized concurrent accesses, at least one of which mutating a shared variable
- Semantic ordering errors: Code that does not enforce the order programmers intended to for a group of memory accesses
- Weak memory consistency models: The set of allowed behaviours w.r.t. memory operations is not what the programer expected

How to prevent race conditions

- **Solutions**
  - re:: Data races:

How to prevent race conditions

- **Solutions**
  - re:: Data races: Disable preemption (s/w); Atomicity (h/w);

How to prevent race conditions

> **Solutions**

- re:: Data races: Disable preemption (s/w); Atomicity (h/w);

>> Atomicity hits data races on their head **by definition**

How to prevent race conditions

> **Solutions**

- re:: Data races: Disable preemption (s/w); Atomicity (h/w);

>> Atomicity hits data races on their head **by definition**

- Processor on thread A
  - %reg <- value at main mem. [global vs. local state: consistent]
  - %reg <- %reg + 1 [global vs. local state: divergent]
  - value at main mem. <- %reg [global vs. local state: consistent]

Have h/w execute these "inseparable" instructions in one go?

- Processor on thread B
  - %reg <- value at main mem. [global vs. local state: consistent]
  - %reg <- %reg + 1 [global vs. local state: divergent]
  - value at main mem. <- %reg [global vs. local state: consistent]

Have h/w execute these "inseparable" instructions in one go?

How to prevent race conditions

> **Solutions**

- re:: Data races: Disable preemption (s/w); Atomicity (h/w);

>> Atomicity hits data races on their head **by definition**

>> "Relaxing" synchronization: "**a** happens before" ⇒ "**some** happens before"

- Processor on thread A
  - %reg <- value at main mem. [global vs. local state: consistent]
  - %reg <- %reg + 1 [global vs. local state: divergent]
  - value at main mem. <- %reg [global vs. local state: consistent]

Have h/w execute these "inseparable" instructions in one go?

- Processor on thread B
  - %reg <- value at main mem. [global vs. local state: consistent]
  - %reg <- %reg + 1 [global vs. local state: divergent]
  - value at main mem. <- %reg [global vs. local state: consistent]

Have h/w execute these "inseparable" instructions in one go?

How to prevent race conditions

> **Solutions**

- re:: Data races: Disable preemption (s/w); Atomicity (h/w);

>> Atomicity hits data races on their head **by definition**

>> "Relaxing" synchronization: "**a** happens before" ⇒ "**some** happens before"

- re:: Semantic ordering errors:

How to prevent race conditions

> **Solutions**

- re:: Data races: Disable preemption (s/w); Atomicity (h/w);

>> Atomicity hits data races on their head **by definition**

>> "Relaxing" synchronization: "**a** happens before" ⇒ "**some** happens before"

- re:: Semantic ordering errors: h/w guardrails;

>> Implement control mechanism directly in the hardware

How to prevent race conditions

> **Solutions**

- re:: Data races: Disable preemption (s/w); Atomicity (h/w);

>> Atomicity hits data races on their head **by definition**

>> "Relaxing" synchronization: "**a** happens before" ⇒ "**some** happens before"

- re:: Semantic ordering errors: h/w guardrails; Verification;

>> Implement control mechanism directly in the hardware

>> Formally verify code to prove that a set of behaviors **cannot** happen

How to prevent race conditions

> **Solutions**

- re:: Data races: Disable preemption (s/w); Atomicity (h/w);

>> Atomicity hits data races on their head **by definition**

>> "Relaxing" synchronization: "**a** happens before" ⇒ "**some** happens before"

- re:: Semantic ordering errors: h/w guardrails; Verification;

>> Implement control mechanism directly in the hardware

>> Formally verify code to prove that a set of behaviors **cannot** happen

- re:: Weak memory consistency models:

How to prevent race conditions

> **Solutions**

- re:: Data races: Disable preemption (s/w); Atomicity (h/w);

>> Atomicity hits data races on their head **by definition**

>> "Relaxing" synchronization: "**a** happens before" ⇒ "**some** happens before"

- re:: Semantic ordering errors: h/w guardrails; Verification;

>> Implement control mechanism directly in the hardware

>> Formally verify code to prove that a set of behaviors **cannot** happen

- re:: Weak memory consistency models: Memory barriers

>> Explicitly prevent reorderings of memory ops.

How to prevent race conditions

> **Solutions**

- re:: Data races: Disable preemption (s/w); Atomicity (h/w);

>> Atomicity hits data races on their head **by definition**

>> "Relaxing" synchronization: "**a** happens before" ⇒ "**some** happens before"

- re:: Semantic ordering errors: h/w guardrails; Verification;

>> Implement control mechanism directly in the hardware

>> Formally verify code to prove that a set of behaviors **cannot** happen

- re:: Weak memory consistency models: Memory barriers

>> Explicitly prevent reorderings of memory ops

> Looks like we are getting somewhere, but…

- What operations can "inseparable" instructions express?

Step back on the big picture

- Where we are?

Step back on the big picture

- **Where we are?** Solved most of the **evils** we've seen so far

Step back on the big picture

- **Where we are?** Solved most of the **evils** we've seen so far
  - **Memory barriers** ⇒ Prevent reorderings

Step back on the big picture

- **Where we are?** Solved most of the **evils** we've seen so far
  - **Memory barriers** ⇒ Prevent reorderings
  - **Atomicity** => Some "happens-before" => Enough for data races

Step back on the big picture

- **Where we are?** Solved most of the **evils** we've seen so far
  - **Memory barriers** ⇒ Prevent reorderings
  - **Atomicity** => Some "happens-before" => Enough for data races
    - Atomic instructions only express too primitive operations

Step back on the big picture

- **Where we are?** Solved most of the **evils** we've seen so far
  - **Memory barriers** ⇒ Prevent reorderings
  - **Atomicity** => Some "happens-before" => Enough for data races
    - Atomic instructions only express too primitive operations
    - Need atomicity of composite operations

Step back on the big picture

- **Where we are?** Solved most of the **evils** we've seen so far
  - **Memory barriers** ⇒ Prevent reorderings
  - **Atomicity** => Some "happens-before" => Enough for data races
    - Atomic instructions only express too primitive operations
    - Need atomicity of composite operations
    - **Ideally:** atomicity of arbitrary many instructions

Step back on the big picture

> **Where we are?** Solved most of the **evils** we've seen so far

>> **Memory barriers** ⇒ Prevent reorderings

>> **Atomicity** => Some "happens-before" => Enough for data races

- Atomic instructions only express too primitive operations
- Need atomicity of composite operations
- **Ideally:** atomicity of arbitrary many instructions

> Arbitrary many instructions ⇒ Critical Section

Step back on the big picture

> **Where we are?** Solved most of the **evils** we've seen so far

>> **Memory barriers** ⇒ Prevent reorderings

>> **Atomicity** => Some "happens-before" => Enough for data races

- Atomic instructions only express too primitive operations
- Need atomicity of composite operations
- **Ideally:** atomicity of arbitrary many instructions

> Arbitrary many instructions ⇒ Critical Section

> Atomicity of arbitrary instructions ⇒ Mutual exclusion

The Critical Section (CS) problem

- *Critical section* (tangible set)
  - A sequence of instructions operating on shared state that ought to be executed by one thread at a time
- *Mutual exclusion* (conceptual property)
  - If I do, you wait; If you do, I wait…
  - **More generally:** At most one does at a time, all others wait

The Critical Section (CS) problem

- *Critical section* (tangible set)
  - A sequence of instructions operating on shared state that ought to be executed by one thread at a time
- *Mutual exclusion* (conceptual property)
  - If I do, you wait; If you do, I wait…
  - **More generally:** At most one does at a time, all others wait
- *The critical section problem*
  - How to allow only one thread at a time in a cs?

The Critical Section (CS) problem

- *Critical section* (tangible set)
  - A sequence of instructions operating on shared state that ought to be executed by one thread at a time
- *Mutual exclusion* (conceptual property)
  - If I do, you wait; If you do, I wait...
  - **More generally:** At most one does at a time, all others wait
- *The critical section problem*
  - How to allow only one thread at a time in a cs?
  - Rephrase: How to turn a sequence of instructions into an "atomic block"?

The Critical Section (CS) problem

- ***Critical section*** (tangible set)
  - A sequence of instructions operating on shared state that ought to be executed by one thread at a time
- ***Mutual exclusion*** *(conceptual property)*
  - If I do, you wait; If you do, I wait...
  - **More generally:** At most one does at a time, all others wait
- ***The critical section problem***
  - How to allow only one thread at a time in a cs?
  - Rephrase: How to turn a sequence of instructions into an "atomic block"?
  - Rephrase: How to restrict the interleavings of thread executions re: entering the cs? (some happens-before relationship exists)

Designing a solution to the CS problem

- Correctness properties (non-negotiable)

Designing a solution to the CS problem

- Correctness properties (non-negotiable)
  - Mutual exclusion: At most one thread inside the CS at any time

Designing a solution to the CS problem

- Correctness properties (non-negotiable)
  - Mutual exclusion: At most one thread inside the CS at any time
  - Progress: If multiple threads attempt to enter the CS, one must be allowed to proceed (a.k.a. liveness)

Designing a solution to the CS problem

- Correctness properties (non-negotiable)
  - Mutual exclusion: At most one thread inside the CS at any time
  - Progress: If multiple threads attempt to enter the CS, one must be allowed to proceed (a.k.a. liveness)
  - Starvation-free: If a thread is waiting to enter the CS, it must eventually enter (a.k.a. bounded-wait)

Designing a solution to the CS problem

- Correctness properties (non-negotiable)
  - Mutual exclusion: At most one thread inside the CS at any time
  - Progress: If multiple threads attempt to enter the CS, one must be allowed to proceed (a.k.a. liveness)
  - Starvation-free: If a thread is waiting to enter the CS, it must eventually enter (a.k.a. bounded-wait)
- Efficiency and Fairness properties (good to have)

Designing a solution to the CS problem

- Correctness properties (non-negotiable)
  - Mutual exclusion: At most one thread inside the CS at any time
  - Progress: If multiple threads attempt to enter the CS, one must be allowed to proceed (a.k.a. liveness)
  - Starvation-free: If a thread is waiting to enter the CS, it must eventually enter (a.k.a. bounded-wait)
- Efficiency and Fairness properties (good to have)
  - Resource-efficiency: Don't waste processor cycles while waiting (busy-waiting) to enter the CS; voluntarily yield the processor

Designing a solution to the CS problem

- Correctness properties (non-negotiable)
  - Mutual exclusion: At most one thread inside the CS at any time
  - Progress: If multiple threads attempt to enter the CS, one must be allowed to proceed (a.k.a. liveness)
  - Starvation-free: If a thread is waiting to enter the CS, it must eventually enter (a.k.a. bounded-wait)
- Efficiency and Fairness properties (good to have)
  - Resource-efficiency: Don't waste processor cycles while waiting (busy-waiting) to enter the CS; voluntarily yield the processor
  - Fairness: All threads must wait approximately the same amount of time outside of the CS

Synchronized access to CS with locking

- We have reached our destination!

Synchronized access to CS with locking

- We have reached our destination!
  - Synchronized access to critical sections

Synchronized access to CS with locking

- We have reached our destination!
  - Synchronized access to critical sections
  - Coordinate among threads to restrict the possible interleavings

Synchronized access to CS with locking

- We have reached our destination!
  - Synchronized access to critical sections
  - Coordinate among threads to restrict the possible interleavings
  - How? **Lock/Locking**

Synchronized access to CS with locking

- We have reached our destination!
  - Synchronized access to critical sections
  - Coordinate among threads to restrict the possible interleavings
  - How? **Lock/Locking**
  - Divide code in two parts

Synchronized access to CS with locking

- We have reached our destination!
  - Synchronized access to critical sections
  - Coordinate among threads to restrict the possible interleavings
  - How? **Lock/Locking**
  - Divide code in two parts
    - **CS**: Executed with serialized access on some partial order
    - **Code outside** CS: Executed concurrently without any concerns

Blast from the past

```c
int total = 0;

void *add(void *arg) {
  for (int i = 0; i < 1e6; ++i) {
    pthread_mutex_lock(&l);
    ++total;
    pthread_mutex_unlock(&l);
  }
  return NULL;
}

void main() {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, add, (void *) NULL);
  pthread_create(&t2, NULL, add, (void *) NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("Total-1: %d\n", total);

  total = 0;
  pthread_create(&t1, NULL, add, (void *) NULL);
  pthread_join(t1, NULL);
  pthread_create(&t2, NULL, add, (void *) NULL);
  pthread_join(t2, NULL);
  printf("Total-2: %d\n", total);
}
```

```
➜ obdjump -d ./counter

0000000000001159 <add>:
  1159: push %rbp                  # Save base pointer to stack
  115a: mov  %rsp, %rbp            # Set up new stack frame
  115d: mov  %rdi, -0x18(%rbp)     # *arg = %rdi
  1161: movl $0x0, -0x4(%rbp)      # i = 0
  1168: jmp  117d <add+0x24>       # for-loop start
  ...
  118e: lea  0x2eeb(%rip),%rax
  1195: mov  %rax,%rdi
  1198: call 1070 <pthread_mutex_lock@plt>   } lock the lock
  ...
  116a: mov  0x2ebc(%rip), %eax    #  %eax ← total
  1170: add  $0x1, %eax            #  %eax += 1
  1173: mov  %eax, 0x2eb3(%rip)    #  total ← %eax   } Critical section
  ...
  11ac: lea  0x2ecd(%rip),%rax
  11b3: mov  %rax,%rdi
  11b6: call 1040 <pthread_mutex_unlock@plt>   } Unlock the lock
  ...
  1179: addl $0x1, -0x4(%rbp)      # i += 1
  117d: cmpl $0xf423f, -0x4(%rbp)  # loop counter compare
  1184: jle  116a <add+0x11>       # for-loop jump
  1186: mov  $0x0, %eax            # rval = %eax
  118b: pop  rbp                   # Restore stack
  118c: ret                        # Return to caller
```

Blast from the past

> Yet, another execution interleaving

- Processor on thread B
  - %reg <- value at main mem. [global vs. local state: consistent]
- Processor on thread A
  - %reg <- value at main mem. [global vs. local state: consistent]
  - %reg <- %reg + 1 [global vs. local state: divergent]
  - value at main mem. <- %reg [global vs. local state: consistent]
- Processor on thread B
  - **OS job:** Load %reg with its value before ctx switch
  - Thead B "thinks" %reg ⇔ shared value at main mem.
  - **But:** shared value has been mutated by someone else

Cannot happen

Implementing locks

Implementing locks

> ***Lock:*** A token-like synchronization primitive used to coordinate concurrent thread accesses on CS

Implementing locks

> ***Lock:*** A token-like synchronization primitive used to coordinate concurrent thread accesses on CS

- While a thread cannot acquire the token, it waits until it acquires the token

Implementing locks

> ***Lock:*** A token-like synchronization primitive used to coordinate concurrent thread accesses on CS

- While a thread cannot acquire the token, it waits until it acquires the token
- When a thread acquires the token, it holds it and no other thread can acquire it

Implementing locks

> ***Lock:*** A token-like synchronization primitive used to coordinate concurrent thread accesses on CS

- While a thread cannot acquire the token, it waits until it acquires the token
- When a thread acquires the token, it holds it and no other thread can acquire it
- When a thread releases the token, any thread can acquire the token

Implementing locks

> ***Lock:*** A token-like synchronization primitive used to coordinate concurrent thread accesses on CS

- While a thread cannot acquire the **lock**, it waits until it acquires the **lock**
- When a thread acquires the **lock**, it holds it and no other thread can acquire it
- When a thread releases the **lock**, any thread can acquire the **lock**

Implementing locks

> ***Lock:*** A token-like synchronization primitive used to coordinate concurrent thread accesses on CS

- While a thread cannot acquire the **lock**, it waits outside of the CS until it acquires the **lock**
- When a thread acquires the **lock**, it execute instructions inside the cs, while all other threads wait outside the CS
- When a thread finishes executing instructions inside the CS, it releases the **lock** and any thread can acquire it

Draft API for locks

- Initialize n **shared** lock variable
  >> Prototype: void **init_lock**(lock_type *lock_ptr);

- Acquire (verb: lock) the (noun: lock) variable
  >> Prototype: void **lock**(lock_type *lock_ptr);

- Release (verb: unlock) the (noun: lock) variable
  >> Prototype: void **unlock**(lock_type *lock_ptr);

Uniprocessor lock (CONFIG_SMP=off, CONFIG_PREEMPT=off)

Uniprocessor lock (CONFIG_SMP=off, CONFIG_PREEMPT=off)

```
>> typedef uint8_t raw_spinlock_t;
>> void init(raw_spinlock_t *lock) {
        *lock = 0; // 0 = unlocked, 1 = locked
   }
```

Uniprocessor lock (CONFIG_SMP=off, CONFIG_PREEMPT=off)

```
>> typedef uint8_t raw_spinlock_t;
>> void init(raw_spinlock_t *lock) {
        *lock = 0; // 0 = unlocked, 1 = locked
   }
>> void lock(raw_spinlock_t *lock) {
        return;
    }
```

Uniprocessor lock (CONFIG_SMP=off, CONFIG_PREEMPT=off)

```
>> typedef uint8_t raw_spinlock_t;
>> void init(raw_spinlock_t *lock) {
        *lock = 0; // 0 = unlocked, 1 = locked
   }
>> void lock(raw_spinlock_t *lock) {
        return;
    }
>> void unlock(raw_spinlock_t *lock) {
        return;
    }
```

Uniprocessor lock (CONFIG_SMP=off, CONFIG_PREEMPT=on)

```
>> typedef uint8_t raw_spinlock_t;
>> void init(raw_spinlock_t *lock) {
        *lock = 0; // 0 = unlocked, 1 = locked
   }
```

Uniprocessor lock (CONFIG_SMP=off, CONFIG_PREEMPT=on)

```
>> typedef uint8_t raw_spinlock_t;
>> void init(raw_spinlock_t *lock) {
        *lock = 0; // 0 = unlocked, 1 = locked
   }
>> void lock(raw_spinlock_t *lock) {
        preempt_disable();
    }
```

Uniprocessor lock (CONFIG_SMP=off, CONFIG_PREEMPT=on)

```
>> typedef uint8_t raw_spinlock_t;
>> void init(raw_spinlock_t *lock) {
        *lock = 0; // 0 = unlocked, 1 = locked
   }
>> void lock(raw_spinlock_t *lock) {
        preempt_disable();
    }
>> void unlock(raw_spinlock_t *lock) {
        preempt_enable();
    }
```

Another blast from the past

- What operations can "inseparable" instructions express?

  - %reg <- [mem.]
  - %reg <- %reg + 1
  - [mem.] <- %reg

  } Atomic operation

Another blast from the past

- Consider another atomic operation

Another blast from the past

- Consider another atomic operation

  xchgb reg, [mem] (composite operation / "inseparable" instructions)

  - temp <- %reg1
  - %reg1 <- [mem.]
  - [mem.] <- temp

  } Atomic swap (exchange): [mem] <-> %reg

Another blast from the past

- Consider another atomic operation

  xchgb reg, [mem] (composite operation / "inseparable" instructions)

  - temp <- %reg1
  - %reg1 <- [mem.]
  - [mem.] <- temp

  } Atomic swap (exchange): [mem] <-> %reg

- Why the hassle?

```
uint8_t test_and_set(uint8_t *flag) {



}
```

Another blast from the past

- Consider another atomic operation

  xchgb reg, [mem] (composite operation / "inseparable" instructions)

  - temp <- %reg1
  - %reg1 <- [mem.]
  - [mem.] <- temp

  } Atomic swap (exchange): [mem] <-> %reg

- Why the hassle?

```
uint8_t test_and_set(uint8_t *flag) {
    %reg <- 1              // "set" val of %reg to 1


}
```

Another blast from the past

- Consider another atomic operation

  xchgb reg, [mem] (composite operation / "inseparable" instructions)

  - temp <- %reg1
  - %reg1 <- [mem.]
  - [mem.] <- temp

  } Atomic swap (exchange): [mem] <-> %reg

- Why the hassle?

```
uint8_t test_and_set(uint8_t *flag) {
    %reg <- 1              // "set" val of %reg to 1
    xchgb %reg, *flag      // val of reg <->  val of flag

}
```

Another blast from the past

- Consider another atomic operation

  xchgb reg, [mem] (composite operation / "inseparable" instructions)

  - temp <- %reg1
  - %reg1 <- [mem.]
  - [mem.] <- temp

  } Atomic swap (exchange): [mem] <-> %reg

- Why the hassle?

```
uint8_t test_and_set(uint8_t *flag) {
    %reg <- 1              // "set" val of %reg to 1
    xchgb %reg, *flag      // val of reg <->  val of flag
    return %reg            // %reg has the old val of flag
 }
```

Another blast from the past

- Consider another atomic operation

  xchgb reg, [mem] (composite operation / "inseparable" instructions)

  - temp <- %reg1
  - %reg1 <- [mem.]
  - [mem.] <- temp

  } Atomic swap (exchange): [mem] <-> %reg

- Why the hassle?

```
uint8_t test_and_set(uint8_t *flag) {
    %reg <- 1              // "set" val of %reg to 1
    xchgb %reg, *flag      // val of reg <->  val of flag
    return %reg            // %reg has the old val of flag
 }
```

  >> **Return 1?** flag was 1

Another blast from the past

- Consider another atomic operation

  xchgb reg, [mem] (composite operation / "inseparable" instructions)

  - temp <- %reg1
  - %reg1 <- [mem.]
  - [mem.] <- temp

  } Atomic swap (exchange): [mem] <-> %reg

- Why the hassle?

```
uint8_t test_and_set(uint8_t *flag) {
    %reg <- 1              // "set" val of %reg to 1
    xchgb %reg, *flag      // val of reg <->  val of flag
    return %reg            // %reg has the old val of flag
 }
```

  >> **Return 1?** flag was 1 (**lock was locked**)

Another blast from the past

- Consider another atomic operation

  xchgb reg, [mem] (composite operation / "inseparable" instructions)

  - temp <- %reg1
  - %reg1 <- [mem.]
  - [mem.] <- temp

  } Atomic swap (exchange): [mem] <-> %reg

- Why the hassle?

```
uint8_t test_and_set(uint8_t *flag) {
    %reg <- 1              // "set" val of %reg to 1
    xchgb %reg, *flag      // val of reg <->  val of flag
    return %reg            // %reg has the old val of flag
 }
```

  >> **Return 1?** flag was 1 (**lock was locked**)

  >> **Return 0?** flag was 0, and is now 1

Another blast from the past

- Consider another atomic operation

  xchgb reg, [mem] (composite operation / "inseparable" instructions)

  - temp <- %reg1
  - %reg1 <- [mem.]
  - [mem.] <- temp

  } Atomic swap (exchange): [mem] <-> %reg

- Why the hassle?

```
uint8_t test_and_set(uint8_t *flag) {
    %reg <- 1              // "set" val of %reg to 1
    xchgb %reg, *flag      // val of reg <->  val of flag
    return %reg            // %reg has the old val of flag
 }
```

  >> **Return 1?** flag was 1 (**lock was locked**)

  >> **Return 0?** flag was 0, and is now 1 (**lock was unlocked and just got locked**)

Draft SMP spinlock implementation

```
>> void init(lock_type *lock) {
        *lock = 0; // 0 = unlocked, 1 = locked
   }
>> void lock(lock_type *lock) {
        preempt_disable();          // Don't lose processor, if getting the lock
        while (test_and_set(lock)) ; // try while lock not free
    }
>> void unlock(lock_type *lock) {
        __asm__ volatile("sfence" ::: "memory"); // mem. barrier
        // all write mem. ops from the cs must completed before this point
        *lock = 0;
        preempt_enable();
    }
```

Draft SMP spinlock implementation

```
>> void init(lock_type *lock) {
        *lock = 0; // 0 = unlocked, 1 = locked
   }
>> void lock(lock_type *lock) {
        preempt_disable();          // Don't lose processor, if getting the lock
        while (test_and_set(lock)) ; // try while lock not free
    }
>> void unlock(lock_type *lock) {
        __asm__ volatile("sfence" ::: "memory"); // mem. barrier
        // all write mem. ops from the cs must completed before this point
        *lock = 0;
        preempt_enable();
    }
```

Designing a "fair" spinlock

- Correctness properties (non-negotiable)
  - Mutual exclusion: At most one thread inside the CS at any time
  - Progress: If multiple threads attempt to enter the CS, one must be allowed to proceed (a.k.a. liveness)
  - Starvation-free: If a thread is waiting to enter the CS, it must eventually enter (a.k.a. bounded-wait)
- Efficiency and Fairness properties (good to have)
  - Resource-efficiency: Don't waste processor cycles while waiting (busy-waiting) to enter the CS; voluntarily yield the processor
  - Fairness: All threads must wait approximately the same amount of time outside of the CS

Designing a "fair" spinlock

- Correctness properties (non-negotiable)
  - Mutual exclusion: At most one thread inside the CS at any time
  - Progress: If multiple threads attempt to enter the CS, one must be allowed to proceed (a.k.a. liveness)
  - Starvation-free: If a thread is waiting to enter the CS, it must eventually enter (a.k.a. bounded-wait)
- Efficiency and Fairness properties (good to have)
  - Resource-efficiency: Don't waste processor cycles while waiting (busy-waiting) to enter the CS; voluntarily yield the processor
  - Fairness: All threads must wait approximately the same amount of time outside of the CS ⇒ **Lamport's Bakery algorithm**

Designing a "fair" spinlock

- Correctness properties (non-negotiable)
  - Mutual exclusion: At most one thread inside the CS at any time
  - Progress: If multiple threads attempt to enter the CS, one must be allowed to proceed (a.k.a. liveness)
  - Starvation-free: If a thread is waiting to enter the CS, it must eventually enter (a.k.a. bounded-wait)
- Efficiency and Fairness properties (good to have)
  - Resource-efficiency: Don't waste processor cycles while waiting (busy-waiting) to enter the CS; voluntarily yield the processor
  - Fairness: All threads must wait approximately the same amount of time outside of the CS ⇒ Lamport's Bakery algorithm ⇒ Ticket spinlock

Types of locks

Types of locks

> Depending on what the thread trying to acquire an unavailable lock does, we have two categories of locks

Types of locks

> Depending on what the thread trying to acquire an unavailable lock does, we have two categories of locks

>> Spinlocks: Spin continuously while trying to acquire the lock

Types of locks

> Depending on what the thread trying to acquire an unavailable lock does, we have two categories of locks

>> Spinlocks: Spin continuously while trying to acquire the lock

>> Sleeping locks (a.k.a. mutex): Self-preempt and sleep if can't acquire the lock

Types of locks

> Depending on what the thread trying to acquire an unavailable lock does, we have two categories of locks

>> Spinlocks: Spin continuously while trying to acquire the lock

- Do use when: Cost of 2*ctx switch > cost of instruction of cs

>> Sleeping locks (a.k.a. mutex): Self-preempt and sleep if can't acquire the lock

Types of locks

> Depending on what the thread trying to acquire an unavailable lock does, we have two categories of locks

>> Spinlocks: Spin continuously while trying to acquire the lock

- **Do** use when: Cost of 2*ctx switch > cost of instruction of cs
- **Do NOT use when**: In unicore

>> Sleeping locks (a.k.a. mutex): Self-preempt and sleep if can't acquire the lock

Types of locks

> Depending on what the thread trying to acquire an unavailable lock does, we have two categories of locks

>> Spinlocks: Spin continuously while trying to acquire the lock

- Do use when: Cost of 2*ctx switch > cost of instruction of cs
- Do NOT use when: In unicore — why?

>> Sleeping locks (a.k.a. mutex): Self-preempt and sleep if can't acquire the lock

Types of locks

> Depending on what the thread trying to acquire an unavailable lock does, we have two categories of locks

>> Spinlocks: Spin continuously while trying to acquire the lock

- **Do** use when: Cost of 2*ctx switch > cost of instruction of cs
- **Do NOT use when**: In unicore — why?
- **Do NOT use when**: CS contains operations that may sleep

>> Sleeping locks (a.k.a. mutex): Self-preempt and sleep if can't acquire the lock

Types of locks

> Depending on what the thread trying to acquire an unavailable lock does, we have two categories of locks

>> Spinlocks: Spin continuously while trying to acquire the lock

- **Do** use when: Cost of 2*ctx switch > cost of instruction of cs
- **Do NOT use when**: In unicore — why?
- **Do NOT use when**: CS contains operations that may sleep — why?

>> Sleeping locks (a.k.a. mutex): Self-preempt and sleep if can't acquire the lock

Types of locks

> Depending on what the thread trying to acquire an unavailable lock does, we have two categories of locks

>> Spinlocks: Spin continuously while trying to acquire the lock

- **Do** use when: Cost of 2*ctx switch > cost of instruction of cs
- **Do NOT use when**: In unicore — why?
- **Do NOT use when**: CS contains operations that may sleep — why? "copy_*_user"?

>> Sleeping locks (a.k.a. mutex): Self-preempt and sleep if can't acquire the lock

Implementing a sleeping lock

```
typedef struct mutex_t {
  // the "lock" variable
  // initially "unlocked"
  int lock = 0;
  // keeps track of waiters
  queue_t queue = NULL;
} mt;
```

Implementing a sleeping lock

```
typedef struct mutex_t {
  // the "lock" variable
  // initially "unlocked"
  int lock = 0;
  // keeps track of waiters
  queue_t queue = NULL;
} mt;
```

```
void mutex_lock(mutex_t *mt) {
  while (test_and_set(&mt->lock)) {

    // lock is being held
    // register self on waiters
    enqueue(&mt->queue, self);
    // yield the processor
    add_task_to_waitq(self);
  }
}
```

Implementing a sleeping lock

```
typedef struct mutex_t {
  // the "lock" variable
  // initially "unlocked"
  int lock = 0;
  // keeps track of waiters
  queue_t queue = NULL;
} mt;
```

```
void mutex_lock(mutex_t *mt) {
  while (test_and_set(&mt->lock)) {

    // lock is being held
    // register self on waiters
    enqueue(&mt->queue, self);
    // yield the processor
    add_task_to_waitq(self);
  }
}
```

```
void mutex_unlock(mutex_t *mt) {
  // release the lock
  mt->lock = 0
  // let a waiters know
  if (!queue_empty(mt->queue)) {
    wake_up(dequeue(mt->queue))
  }
}
```

Implementing a sleeping lock: Lost wakeup

```
typedef struct mutex_t {
  // the "lock" variable
  // initially "unlocked"
  int lock = 0;
  // keeps track of waiters
  queue_t queue = NULL;
} mt;
```

```
void mutex_lock(mutex_t *mt) {
  while (test_and_set(&mt->lock)) {

    // lock is being held
    // register self on waiters
    enqueue(&mt->queue, self);
    // yield the processor
    add_task_to_waitq(self);
  }
}
```

```
void mutex_unlock(mutex_t *mt) {
  // release the lock
  mt->lock = 0
  // let a waiters know
  if (!queue_empty(mt->queue)) {
    wake_up(dequeue(mt->queue))
  }
}
```

Implementing a sleeping lock

```
typedef struct mutex_t {
  // the "lock" variable
  // initially "unlocked"
  int lock = 0;
  // lock is a shared variable
  raw_spinlock_t guard = 0;
  // keeps track of waiters
  queue_t queue = NULL;
} mt;
```

```
void mutex_lock(mutex_t *mt) {
  spinlock_lock(&mt->guard)
  // lock is being held
  while (mt->lock != 0) {
    // register self on waiters
    enqueue(&mt->queue, self);
    spinlock_unlock(&mt->guard);
    // yield the processor
    add_task_to_waitq(self);
    spinlock_lock(&mt->guard);
  }
  // acquire the lock
  mt->lock = 1;
  spinlock_unlock(&mt->guard);
}
```

```
void mutex_unlock(mutex_t *mt) {
  spinlock_lock(&mt->guard);
  // release the lock
  mt->lock = 0
  // let a waiters know
  if (!queue_empty(mt->queue)) {
    wake_up(dequeue(mt->queue))
  }
  spinlock_unlock(&mt->guard)
}
```

Types of locks

> Depending on what the thread trying to acquire an unavailable lock does, we have two categories of locks

>> Spinlocks: Spin continuously while trying to acquire the lock

- **Do** use when: Cost of 2*ctx switch > cost of instruction of cs
- **Do NOT use when**: In unicore — why?
- **Do NOT use when**: CS contains operations that may sleep — why? "copy_*_user"?

>> Sleeping locks (a.k.a. mutex): Self-preempt and sleep if can't acquire the lock

Types of locks

> Depending on what the thread trying to acquire an unavailable lock does, we have two categories of locks

>> Spinlocks: Spin continuously while trying to acquire the lock

- **Do** use when: Cost of 2*ctx switch > cost of instruction of cs
- **Do NOT use when**: In unicore — why?
- **Do NOT use when**: CS contains operations that may sleep — why? "copy_*_user"?

>> Sleeping locks (a.k.a. mutex): Self-preempt and sleep if can't acquire the lock

- **Do** use when: Cost of 2*ctx switch << cost of instruction of cs

Types of locks

> Depending on what the thread trying to acquire an unavailable lock does, we have two categories of locks

>> Spinlocks: Spin continuously while trying to acquire the lock

- **Do** use when: Cost of 2*ctx switch > cost of instruction of cs
- **Do NOT use when**: In unicore — why?
- **Do NOT use when**: CS contains operations that may sleep — why? "copy_*_user"?

>> Sleeping locks (a.k.a. mutex): Self-preempt and sleep if can't acquire the lock

- **Do** use when: Cost of 2*ctx switch << cost of instruction of cs
- **Do** use when: Relatively larger CS

Types of locks

> Depending on what the thread trying to acquire an unavailable lock does, we have two categories of locks

>> Spinlocks: Spin continuously while trying to acquire the lock

- **Do** use when: Cost of 2*ctx switch > cost of instruction of cs
- **Do NOT use when**: In unicore — why?
- **Do NOT use when**: CS contains operations that may sleep — why? "copy_*_user"?

>> Sleeping locks (a.k.a. mutex): Self-preempt and sleep if can't acquire the lock

- **Do** use when: Cost of 2*ctx switch << cost of instruction of cs
- **Do** use when: Relatively larger CS
- **Do NOT use when**: In interrupt handlers — why?