K22 - Operating Systems: Design Principles and Internals

Fall 2025 @dit

Vaggelis Atlidakis

Lecture 08

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 processor, <u>seemingly simultaneously</u>
- **Interleaving of execution:** The order with which concurrent operations are scheduled in for and out of execution
- **Happens-before relationship?**

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…

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?**

Departing from temporal ordering

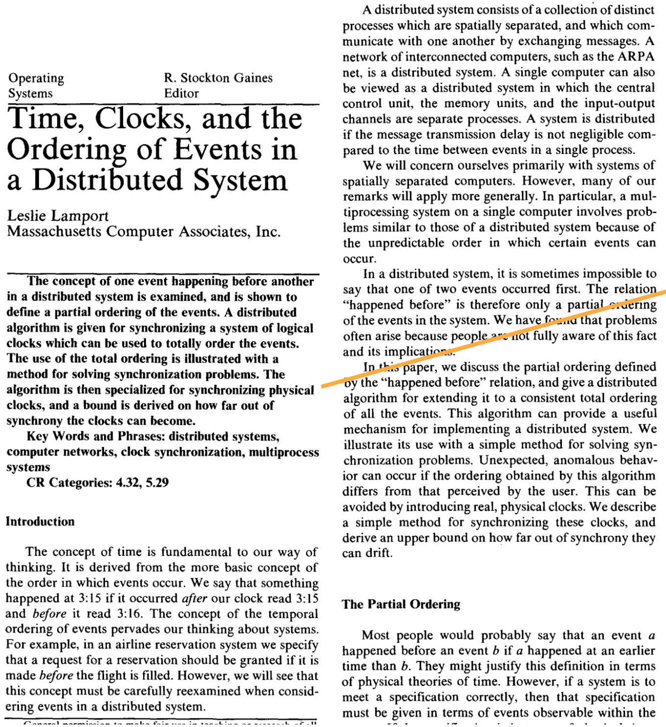
> Temporal ordering: Arrangement of events in a sequence according to physical time

- 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

"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.

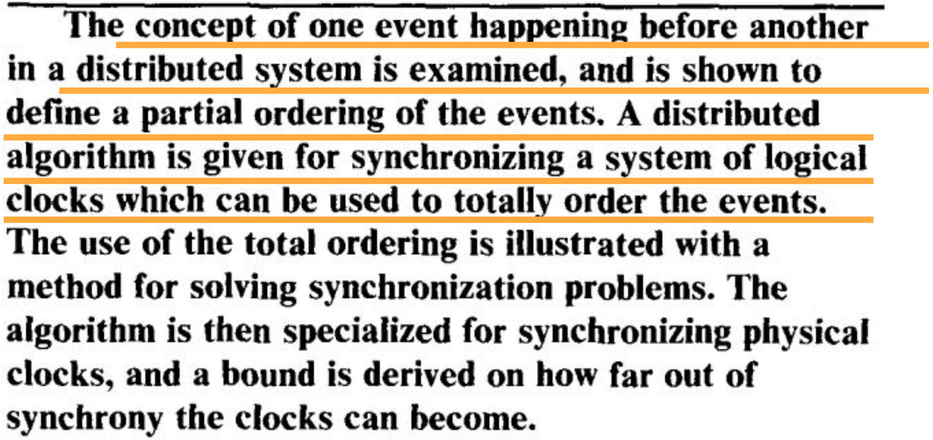
We will concern ourselves primarily with systems of spatially separated computers. However, many of our remarks will apply more generally. In particular, a multiprocessing system on a single computer involves problems similar to those of a distributed system because of the unpredictable order in which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

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

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?**

Reasoning about 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

How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs

LESLIE LAMPORT

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

Reasoning about 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*

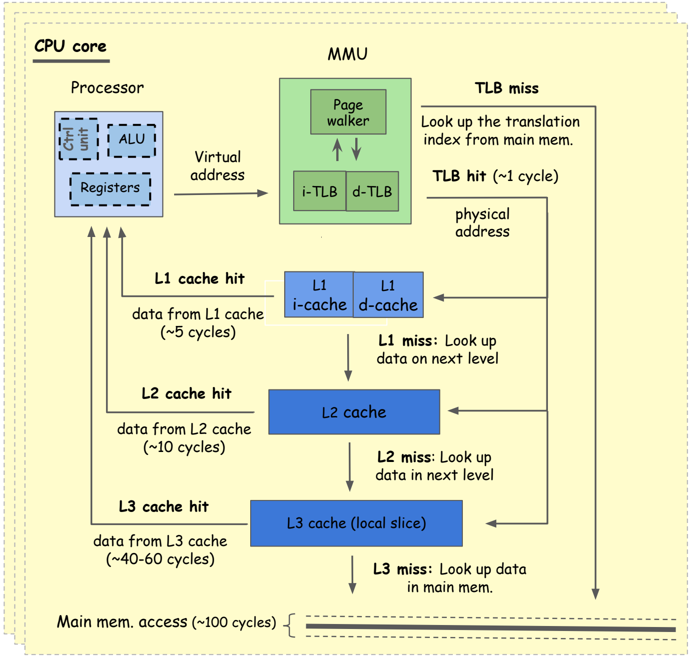
- **Parallel operations:** Operations that are happening at the same time, on different processors
- **Concurrent operations:** Operations that are happening in overlapping time intervals seemingly simultaneously
- **Interleaving of execution:** The order with which concurrent operations are scheduled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

What is so hard about correct concurrent code?

> Concurrent progs have too many execution interlevings

- 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

How many is "too many"?

- 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? **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**

A few dedicates of state space exploration

> We' ve been searching for decades ways to reduce the size of the state space of concurrent programs and test them

- Partial-Order Methods for the Verification of Concurrent Systems, in 1995, by Patrice Godefroid.
- Model checking to find serious file system errors, in 2006, by Junfeng Yang et al.
- RESTler: Stateful REST API Fuzzing, in 2019, by Atlidakis et al.

"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

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


  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

```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

Data races

> Two domains of state for each thread

- Processor regs (per-thread, local state) vs. Main memory (global state)

Data races

> Two domains of state for each thread

- Processor regs (per-thread, local state) vs. Main memory (global state)
- Processor on thread *A*

Data races

> Two domains of state for each thread

- Processor regs (per-thread, local state) vs. Main memory (global state)
- Processor on thread *A*
  - %reg <- value at main mem. [global vs. local state: consistent]

Data races

- Two domains of state for each thread
  - Processor regs (per-thread, local state) vs. Main memory (global state)
  - Processor on thread A
    - %reg <- value at main mem. [global vs. local state: consistent]
    - %reg <- %reg + 1 [global vs. local state: divergent]

Data races

> Two domains of state for each thread

- Processor regs (per-thread, local state) vs. Main memory (global state)
- 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]

Data races

> Two domains of state for each thread

- Processor regs (per-thread, local state) vs. Main memory (global state)
- 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]
- Thread A gets preemtped [shared value at main mem. +1]

Data races

> Two domains of state for each thread

- Processor regs (per-thread, local state) vs. Main memory (global state)
- 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]
- Thread A gets preemtped [shared value at main mem. +1]
- Processor on thread B

Data races

> Two domains of state for each thread

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

Data races

> Two domains of state for each thread

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

Data races

> Two domains of state for each thread

- Processor regs (per-thread, local state) vs. Main memory (global state)
- 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]
- Thread A gets preemtped [shared value at main mem. +1]
- 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]

Data races

> Two domains of state for each thread

- Processor regs (per-thread, local state) vs. Main memory (global state)
- 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]
- Thread A gets preemtped [shared value at main mem. +1]
- 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]
- Thread B gets preemtped [shared value at main mem. +1]

Data races

> Two domains of state for each thread

- Processor regs (per-thread, local state) vs. Main memory (global state)
- 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]
- Thread A gets preemtped [shared value at main mem. +1]
- 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]
- Thread B gets preemtped [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]

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]

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 on thread A
    - **OS job**: Load %reg with its value before ctx switch

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 on thread A
  - **OS job:** Load %reg with its value before ctx switch
  - Thead B "thinks" %reg ⇔ shared value at main mem.

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 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

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 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

> Yet, another execution interleaving

- Processor on thread B
  - %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 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

Data races

> **Bottom-line**: Concurrent writes on shared state?

- Each thread must finish its business before it gets preempted

- 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
  - %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]

Data races

> **Bottom-line**: Concurrent writes on shared state?

- Each thread must finish its business before it gets preempted
- Inseperable "instructions"

- 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]

These instructions are "inseperable"

- 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]

These instructions are "inseperable"

Data races

> **Bottom-line**: Concurrent writes on shared state?

- Each thread must finish its business before it gets preempted
- Inseperable "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]

These instructions are "inseperable"

- 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]

These instructions are "inseperable"