Programmation Concurrente

CM2 - The Shared-Memory Model

Florian Brandner & Laurent Pautet
LTCI, Télécom ParisTech, Université Paris-Saclay

Outline

Course Outline

- CM1: Introduction
- **CM2: The *shared-memory* model**
  - Computer architecture
  - Issues with shared memory
  - Concurrent memory accesses
  - Coherence, consistency and memory models
  - Synchronization in C11
- CM3-6: Concurrent programming POSIX/Java (L. Pautet)
  Patterns and Algorithms (L. Pautet)
- CM7: Actor-based programming (me)
- CM8: Transactional memory (me)

Recapture

Forms of Parallelism

- Instruction-centric parallelism
  - Pipelining, Instruction-level parallelism
  - Vector-like data-level parallelism
  - Modern computers combine all those techniques
- Beyond instructions:
  - Thread-level parallelism (aka task-level parallelism)

Dependencies

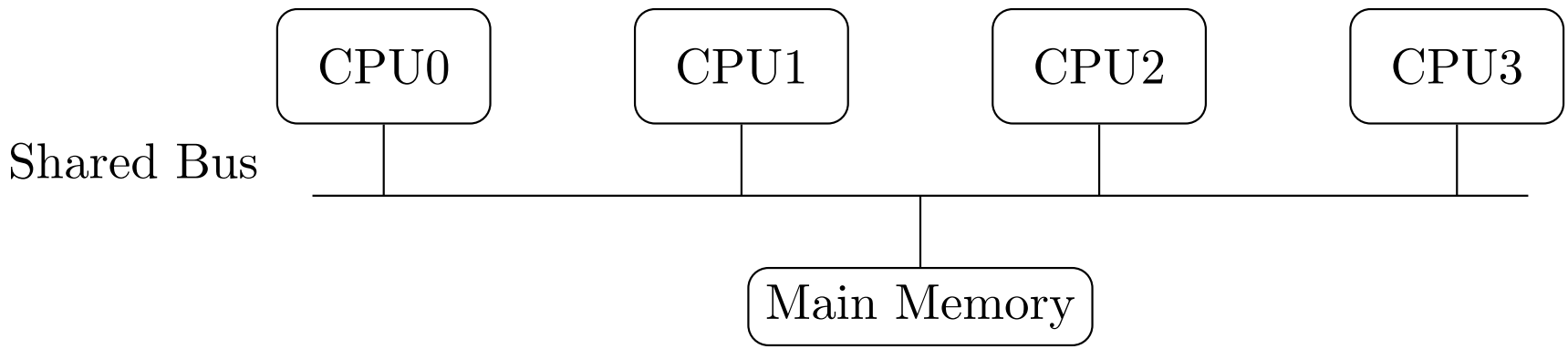
- Amdahl's Law
  - The amount of strictly sequential code limits the speedup
  - Quite drastically
- Dependencies determine amount of sequential code
- Control- and data dependencies
  - True dependence (RAW)
  - Anti dependence (WAR)
  - Output dependence (WAW)
- Loop-carried dependencies
- Iteration space graphs

The Shared-Memory Model

Shared-Memory Computers

A view of the computer architecture:

- Several processors forming a single *computer*
- All processors *logically* access a single main memory
- All data in this main memory is *shared* by all processors



Example: 2 Threads Accessing Shared Memory

```
a = 0
while(a != 5)
{
  printf("w:%d\n", a++);
}
printf("done\n");
```

Thread 1 reads/writes shared global variable `a`

```
a = 0
while(a != 5)
{
  printf("r:%d\n", a);
}
printf("done\n");
```

Thread 2 only reads shared global variable `a` (well almost)

Group Exercise: Parallel Memory Accesses

Look at the two threads from the previous slide:

- Will both threads eventually print `"done"` on the screen?
- Could this be an output you may observe:

```
w: 0
w: 0
w: 1
w: 2
w: 3
w: 4
done                    done
```

Thread 1 Thread 2

Parallelism in the Shared-Memory Model

What does it mean to execute a set of threads T in parallel?

- A thread $\tau \in T$ is comprised of:
  - A set of private registers
  - A set of instructions I^τ (the program)
  - Some instructions are memory accesses $M^\tau \subseteq I^\tau$
  - Others do not touch memory:[1] $O^\tau \subseteq I^\tau, M^\tau \cap O^\tau = \emptyset$
- An execution of a thread τ is:
  - A sequence of instructions $(i_0^\tau, \ldots, i_{n_\tau}^\tau), i_k^\tau \in I^\tau, n_\tau \in \mathbb{N}$
  - That is: an ordering of the instructions
- A parallel execution of all threads in T is:
  - An interleaving of the executions of each thread
  - For example, for $T = \{\alpha, \beta, \gamma\}$:
    $(i_0^\alpha, i_0^\beta, i_0^\gamma, i_1^\alpha, i_1^\beta, i_1^\gamma, \ldots)$

[1] let us ignore the access required to fetch an instruction from memory

Interleavings and Program Semantics

Lets start with some observations:

- Interleavings changing only the order of instructions in the respective O^{τ} sets are equivalent
- Interleavings changing the order of instructions in the respective M^{τ} sets *may* show differing behaviors
- We can simplify our model and only consider memory accesses and their ordering

Example: Interleavings of the Example above

A simplified view of the program:

```
α0: a =     // write a
α1:   = a   // read a
α2:   = a   // read a
α3: a =     // write a
```

Thread 1

```
β0: a =
β1:   = a
β2:   = a
```

Thread 2

An interleaving explaining the output from before:

$$(\alpha_0, \alpha_1, \alpha_2, \alpha_3, \beta_0, \alpha_1, \alpha_2, \alpha_3, \ldots, \beta_1)$$

a

Example: Interleavings of the Example above

A simplified view of the program:

```
α0: a =     // write a
α1:   = a   // read a
α2:   = a   // read a
α3: a =     // write a
```

Thread 1

```
β0: a =
β1:   = a
β2:   = a
```

Thread 2

An interleaving explaining the output from before:

$$(\alpha_0, \alpha_1, \alpha_2, \alpha_3, \beta_0, \alpha_1, \alpha_2, \alpha_3, \ldots, \beta_1)$$

$$\downarrow$$

a 0

Example: Interleavings of the Example above

A simplified view of the program:

```
α0: a =     // write a
α1:   = a   // read a
α2:   = a   // read a
α3: a =     // write a
```

Thread 1

```
β0: a =
β1:   = a
β2:   = a
```

Thread 2

An interleaving explaining the output from before:

$$(\alpha_0, \alpha_1, \alpha_2, \alpha_3, \beta_0, \alpha_1, \alpha_2, \alpha_3, \ldots, \beta_1)$$

$$\downarrow$$

a 1

Example: Interleavings of the Example above

A simplified view of the program:

```
α0: a =     // write a
α1:   = a   // read a
α2:   = a   // read a
α3: a =     // write a
```

Thread 1

```
β0: a =
β1:   = a
β2:   = a
```

Thread 2

An interleaving explaining the output from before:

$$(\alpha_0, \alpha_1, \alpha_2, \alpha_3, \beta_0, \alpha_1, \alpha_2, \alpha_3, \ldots, \beta_1)$$

$$\downarrow$$

a 0

Example: Interleavings of the Example above

A simplified view of the program:

```
α0: a =     // write a
α1:   = a   // read a
α2:   = a   // read a
α3: a =     // write a
```

Thread 1

```
β0: a =
β1:   = a
β2:   = a
```

Thread 2

An interleaving explaining the output from before:

$$(\alpha_0, \alpha_1, \alpha_2, \alpha_3, \beta_0, \alpha_1, \alpha_2, \alpha_3, \ldots, \beta_1)$$

$$\downarrow$$

a 1

Example: Interleavings of the Example above

A simplified view of the program:

```
α0: a =     // write a
α1:   = a   // read a
α2:   = a   // read a
α3: a =     // write a
```

Thread 1

```
β0: a =
β1:   = a
β2:   = a
```

Thread 2

An interleaving explaining the output from before:

$$(\alpha_0, \alpha_1, \alpha_2, \alpha_3, \beta_0, \alpha_1, \alpha_2, \alpha_3, \ldots, \beta_1)$$

$$\downarrow$$

a 2

Example: Interleavings of the Example above

A simplified view of the program:

```
α0: a =     // write a
α1:   = a   // read a
α2:   = a   // read a
α3: a =     // write a
```

Thread 1

```
β0: a =
β1:   = a
β2:   = a
```

Thread 2

An interleaving explaining the output from before:

$$(\alpha_0, \alpha_1, \alpha_2, \alpha_3, \beta_0, \alpha_1, \alpha_2, \alpha_3, \ldots, \beta_1)$$

$$\downarrow$$

a 3

Example: Interleavings of the Example above

A simplified view of the program:

```
α0: a =     // write a
α1:   = a   // read a
α2:   = a   // read a
α3: a =     // write a
```

Thread 1

```
β0: a =
β1:   = a
β2:   = a
```

Thread 2

An interleaving explaining the output from before:

$$(\alpha_0, \alpha_1, \alpha_2, \alpha_3, \beta_0, \alpha_1, \alpha_2, \alpha_3, \ldots, \beta_1)$$

$$\downarrow$$

a 4

Example: Interleavings of the Example above

A simplified view of the program:

```
α0: a =     // write a
α1:   = a   // read a
α2:   = a   // read a
α3: a =     // write a
```

Thread 1

```
β0: a =
β1:   = a
β2:   = a
```

Thread 2

An interleaving explaining the output from before:

$$(\alpha_0, \alpha_1, \alpha_2, \alpha_3, \beta_0, \alpha_1, \alpha_2, \alpha_3, \ldots, \beta_1)$$

$$\downarrow$$

a 5

Example: Interleavings of the Example above

A simplified view of the program:

```
α0: a =     // write a
α1:   = a   // read a
α2:   = a   // read a
α3: a =     // write a
```

Thread 1

```
β0: a =
β1:   = a
β2:   = a
```

Thread 2

An interleaving explaining the output from before:

$$(\alpha_0, \alpha_1, \alpha_2, \alpha_3, \beta_0, \alpha_1, \alpha_2, \alpha_3, \ldots, \beta_1)$$

$$\uparrow$$

a 5

Group Exercise: Constructing an Interleaving

Can you construct an interleaving where Thread 2 prints `"5"`?

```
α0: a =     // write a
α1:   = a   // read a
α2:   = a   // read a
α3: a =     // write a
```

Thread 1

```
β0: a =
β1:   = a
β2:   = a
```

Thread 2

Group Exercise: Constructing an Interleaving

Can you construct an interleaving where Thread 2 prints `"5"`?

```
α0: a =     // write a
α1:   = a   // read a
α2:   = a   // read a
α3: a =     // write a
```

Thread 1

```
β0: a =
β1:   = a
β2:   = a
```

Thread 2

An interleaving resulting in Thread 2 printing `"5"`:

$$(\ldots, \alpha_1, \alpha_2, \beta_1, \alpha_3, \beta_2)$$

$$\downarrow$$

a 0

Group Exercise: Constructing an Interleaving

Can you construct an interleaving where Thread 2 prints `"5"`?

```
α0: a =     // write a
α1:   = a   // read a
α2:   = a   // read a
α3: a =     // write a
```

Thread 1

```
β0: a =
β1:   = a
β2:   = a
```

Thread 2

An interleaving resulting in Thread 2 printing `"5"`:

$$(\ldots, \alpha_1, \alpha_2, \beta_1, \alpha_3, \beta_2)$$

$$\downarrow$$

a 1

Group Exercise: Constructing an Interleaving

Can you construct an interleaving where Thread 2 prints `"5"`?

```
α0: a =     // write a
α1:   = a   // read a
α2:   = a   // read a
α3: a =     // write a
```

Thread 1

```
β0: a =
β1:   = a
β2:   = a
```

Thread 2

An interleaving resulting in Thread 2 printing `"5"`:

$$(\ldots, \alpha_1, \alpha_2, \beta_1, \alpha_3, \beta_2)$$

$$\downarrow$$

a 2

Group Exercise: Constructing an Interleaving

Can you construct an interleaving where Thread 2 prints `"5"`?

```
α0: a =     // write a
α1:   = a   // read a
α2:   = a   // read a
α3: a =     // write a
```

Thread 1

```
β0: a =
β1:   = a
β2:   = a
```

Thread 2

An interleaving resulting in Thread 2 printing `"5"`:

$$(\ldots, \alpha_1, \alpha_2, \beta_1, \alpha_3, \beta_2)$$

$$\downarrow$$

a 3

Group Exercise: Constructing an Interleaving

Can you construct an interleaving where Thread 2 prints `"5"`?

```
α0: a =     // write a
α1:   = a   // read a
α2:   = a   // read a
α3: a =     // write a
```

Thread 1

```
β0: a =
β1:   = a
β2:   = a
```

Thread 2

An interleaving resulting in Thread 2 printing `"5"`:

$$(\ldots, \alpha_1, \alpha_2, \beta_1, \alpha_3, \beta_2)$$

$$\downarrow$$

a 4

Group Exercise: Constructing an Interleaving

Can you construct an interleaving where Thread 2 prints `"5"`?

```
α0: a =     // write a
α1:   = a   // read a
α2:   = a   // read a
α3: a =     // write a
```

Thread 1

```
β0: a =
β1:   = a
β2:   = a
```

Thread 2

An interleaving resulting in Thread 2 printing `"5"`:

$$(\ldots, \alpha_1, \alpha_2, \beta_1, \alpha_3, \beta_2)$$

$\uparrow$

a 4

Group Exercise: Constructing an Interleaving

Can you construct an interleaving where Thread 2 prints `"5"`?

```
α0: a =     // write a
α1:   = a   // read a
α2:   = a   // read a
α3: a =     // write a
```

Thread 1

```
β0: a =
β1:   = a
β2:   = a
```

Thread 2

An interleaving resulting in Thread 2 printing `"5"`:

$$(\ldots, \alpha_1, \alpha_2, \beta_1, \alpha_3, \beta_2)$$

$$\downarrow$$

a 5

Group Exercise: Constructing an Interleaving

Can you construct an interleaving where Thread 2 prints `"5"`?

```
α0: a =     // write a
α1:   = a   // read a
α2:   = a   // read a
α3: a =     // write a
```

Thread 1

```
β0: a =
β1:   = a
β2:   = a
```

Thread 2

An interleaving resulting in Thread 2 printing `"5"`:

$$(\ldots, \alpha_1, \alpha_2, \beta_1, \alpha_3, \beta_2)$$

a 5

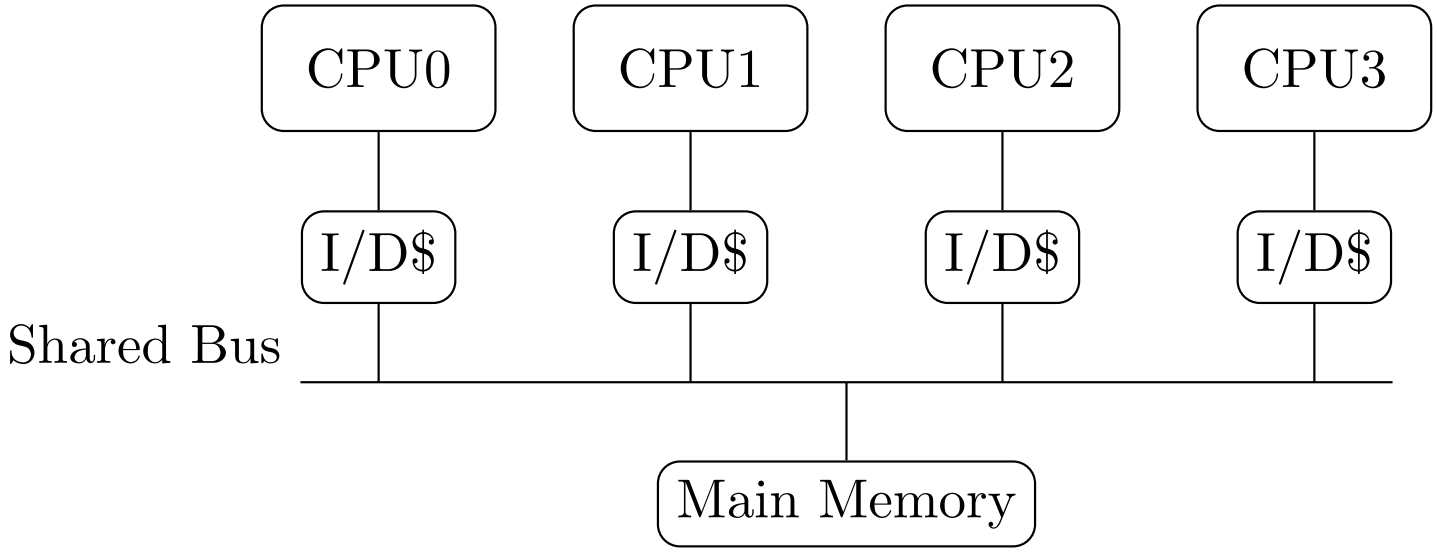
Interleavings and Program Semantics

- Not all interleavings give rise to an actual execution
  - Semantics of programs
  - Explicit synchronization between threads
  - Implicit synchronization between threads
  - External events (I/O devices, . . .)
- Not all interleavings give the same result
  - Indeterministic programs!!
  - This is often caused by bugs, called *race conditions*

Coherence, Consistency & Memory Models

Shared Memory and Caching

Lets have a look at a more realistic computer architecture:



Introduce small and fast cache memories[2] close to each CPU. Hold *copies* of data items from the main memory to speed-up accesses to the cached data.

[2] I/D$ . . . Instruction and/or data cache.

Write Policy

Defines how data written to *memory* is handled:

Write-through: Data is written into the cache and to main memory at once.

Write-back: Data is written into cache only. Memory is updated when data is evicted from cache.

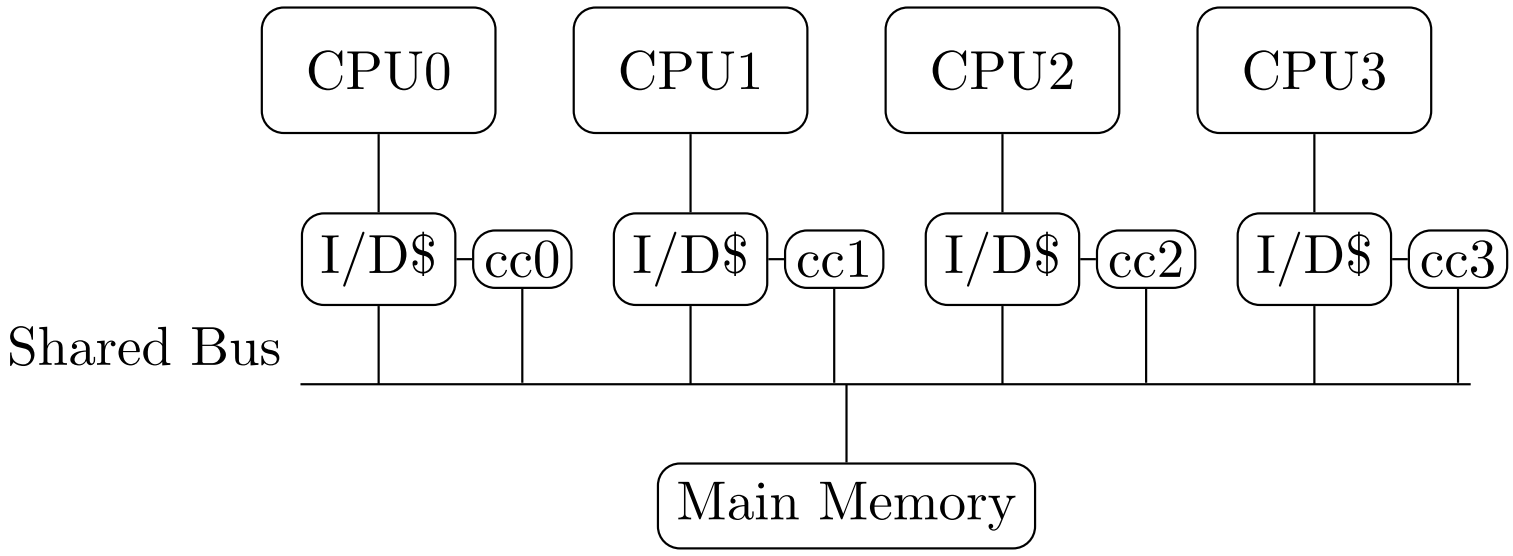
Most modern processors use write-back!

Incoherent Caches

Parallel writes to the same address in parallel:

- Write-through:
  - Each CPU updates its own cache with its own value
  - Each CPU sends a write-request to main memory
  - These requests are executed in some order
  - **Result:** *n* different values in each cache. Only one cache is coherent with main memory.
- Write-back:
  - Each CPU updates its own cache with its own value
  - Who knows what happens then . . .
  - **Result:** *n* different values in each cache. In the worst case, not a single cache is coherent with main memory.

Coherent Caches



- Caches exchange information about their contents.
- This protocol allows to keep data coherent.
- Does not scale to large numbers of cores.

Reordering Memory Accesses

Most modern processors execute instructions in parallel:

- Independent memory accesses can be reordered
  - Store buffer (buffer short sequences of writes without delay)
  - The compiler might reorder accesses
- This is ok with regard to a single thread . . .
- . . . but may cause troubles in multi-threaded programs:

```
α1: x = 1                    β1: y = 1
α2: if (y == 0)              β2: if (x == 0)
        printf("zero");              printf("zero");
```

Both threads might print `"zero"`, e.g., when the processor happens to reorder the accesses to `x` and `y` ($\alpha_2, \beta_2, \alpha_1, \beta_1$).

Implications for Parallel Programming

Parallel programming models have to define:

- What does it mean to access a (shared) global variable?
- What is the hardware supposed to do?
- What is the compiler allowed to do?
- What can one expect to happen given some program?
- How can one express what is supposed to happen?

$\Rightarrow$ Memory models address these problems

Sequential Consistency

A strong memory model defined by Lamport:

- A single processor is **sequential** when “the result of an execution is the same as if the operations had been executed in the order specified by the program.”
- A multi-processor system is **sequentially consistent** if “the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.”[3]
- Recall the interleavings we discussed before!

[3] Leslie Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs", IEEE Trans. Comput. C-28,9 (Sept. 1979), 690-691.

Weaker Memory Models

- Total Store Order (TSO)
  - A *strong* memory model used by x86, SPARC, ...
  - Relaxes ordering between certain memory writes with respect to following loads
  - Memory writes (stores) are fully ordered though
- More relaxed models used by Power and ARM

The Memory Model of C11

Guarantees sequential consistency:

- But only for race-free programs
- Operations are classified as:
  (1) synchronization, (2) atomic load/store, (3) load/store
- Inter-thread communicate via synchronization and atomics
- Well-defined partial ordering of operations per thread
- Executions as interleavings of the threads
- **Race:** existence of two interleavings in which (1) two memory operations accessing the same address are reordered and (2) at least on of them is a write and (3) one of them is a non-atomic load or store
- Undefined semantics of racy programs

Synchronization in C11

Threads in C

Use `pthread` library to create and manage threads:

```
#include <stdio.h>
#include <pthread.h>

void *myThread(void *parameter) {
  puts("Hello␣Thread!\n");
  return NULL;
}

int main() {
  pthread_t th;                  // create and start thread
  pthread_create(&th, NULL, myThread, NULL);

  pthread_join(th, NULL);        // wait for thread to terminate
  puts("Hello␣Main!\n");
  return 0;
}
```

Threads Explained

- Threads are normal C objects (`pthread_t`)
- Creating a new thread also starts its execution, e.g.:
  `pthread_create(&th, NULL, myThread, NULL)`
- Invoking the `pthread_join()` method on a thread causes the current thread to wait for its termination:
  `pthread_join(th, NULL)`
- More thread-related functions are available
  (we won't discuss them here)

Atomics

```c
#include <stdatomic.h>
#include <stdio.h>
#include <pthread.h>

atomic_int a = 0;

void *myThread(void *parameter) {
  int x = atomic_fetch_add(&a, 5); // atomically increment by 5
  printf("%d\n", x);               // print exactly once 0 then 5
  return NULL;
}

int main() {
  pthread_t th1, th2;              // create and start threads
  pthread_create(&th1, NULL, myThread, NULL);
  pthread_create(&th2, NULL, myThread, NULL);

  pthread_join(th1, NULL);         // wait for threads to terminate
  pthread_join(th2, NULL);

  // always prints 10!
  printf("%d\n", atomic_load_explicit(&a, memory_order_seq_cst));
  return 0;
}
```

Atomics Explained

Control the way shared data is accessed:

- All accesses are atomic operations
  `atomic_fetch_add(&a, 5)`
- All accesses are data-race free
  `atomic_fetch_add(&a, 5)` and
  `atomic_load_explicit(&a)`
- *Memory order* can be controlled (`..._explicit`):

```
enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst  // this is the default
};
```

Summary

In a shared-memory setting:

- Semantics of thread-level parallelism
  - Interleavings of all operations of the parallel threads
  - Only the order of memory operations is relevant
  - Interleavings are not necessarily deterministic (Race)
- Coherence and consistency
  - Coherent caches exchange information on cache contents
  - Reordering of memory accesses (store buffers)
  - Memory models define how threads agree on the value of shared data
- Memory models
  - Sequential consistency means that for every execution an interleaving exists explaining that execution
  - Processors implement weaker memory models (e.g., TSO)

Summary (2)

In a shared-memory setting:

- C11 formally defines a memory/thread model
  - Sequential consistency for race free programs
  - Undefined semantics for racy programs
- C11 provides features for synchronization
  - Atomic memory operations (using type wrappers)

Further Reading

- A Primer on Memory Consistency and Cache Coherence
  Daniel J. Sorin, Mark D. Hill, David A. Wood (2011)
- Foundations of the C++ Concurrency Memory Model
  Hans-J. Boehm, Sarita V. Adve (PLDI 2008)
- Lots of work by Peter Sewell
  http://www.cl.cam.ac.uk/ pes20/weakmemory/
- C Library documentation:
  http://en.cppreference.com/w/c/atomic