Concurrency in Modern Hardware

Concurrency

What is concurrency?

```
function foo() { ... }
function bar() { ... }

function main() {
    t1 = startThread(foo)
    t2 = startThread(bar)

    // Wait for t1 and t2 to finish before continuing executing main()
    waitUntilFinished(t1)
    waitUntilFinished(t2)

    // No concurrent execution here anymore
}
```

In this example program, concurrency means that `foo()` and `bar()` are executed *at the same time*.

- How does a CPU actually do this?
- How can concurrency be used to make your programs faster?

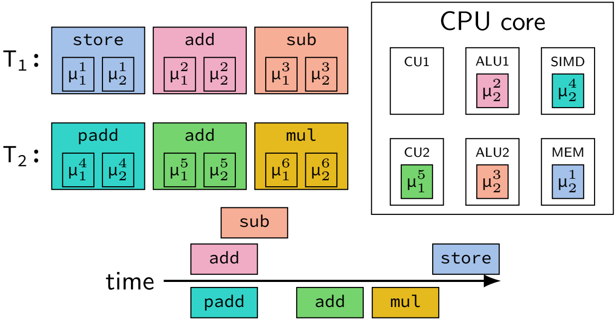
Concurrency in Modern Hardware

- Modern CPUs can execute multiple instruction streams simultaneously:
  - Single CPU cores can execute multiple threads:
    Simultaneous Multi-Threading (SMT), Intel calls it hyper-threading
  - Of course CPUs can also have multiple cores that can run independently
- To get the best performance in C++ systems programming, writing multi-threaded programs is essential
- For this, a basic understanding of how hardware behaves in the context of parallel programming is required
- Actually writing multi-threaded C++ programs will be covered in a future lecture

Most of the low-level implementation details can be found in the Intel Architectures Software Developer's Manual and the ARM Architecture Reference Manual

Simultaneous Multi-Threading (SMT)

- CPUs support *instruction-level parallelism* by using out-of-order execution
- With SMT, CPUs also support *thread-level parallelism*
  - In a single CPU core, multiple threads are executed
  - Many hardware components, like the ALU, the SIMD unit, etc., are shared between the threads
  - Other components are duplicated for each thread, e.g. control unit to fetch and decode instructions, register file

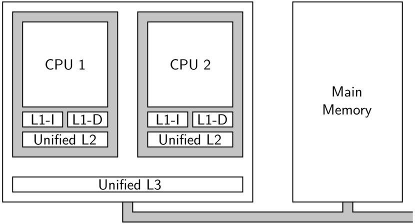


Problems with SMT

When using SMT, multiple instruction streams share parts of the CPU core.

- When one stream alone already utilizes all computation units, SMT does not increase performance
- Same for memory bandwidth
- Some units may only exist once on the core, so SMT can also decrease performance
- When two threads from unrelated processes run on the same core, this can potentially lead to security issues $\rightarrow$ Security issues similar to Spectre and Meltdown are suspected to be enabled by SMT

Cache Coherence

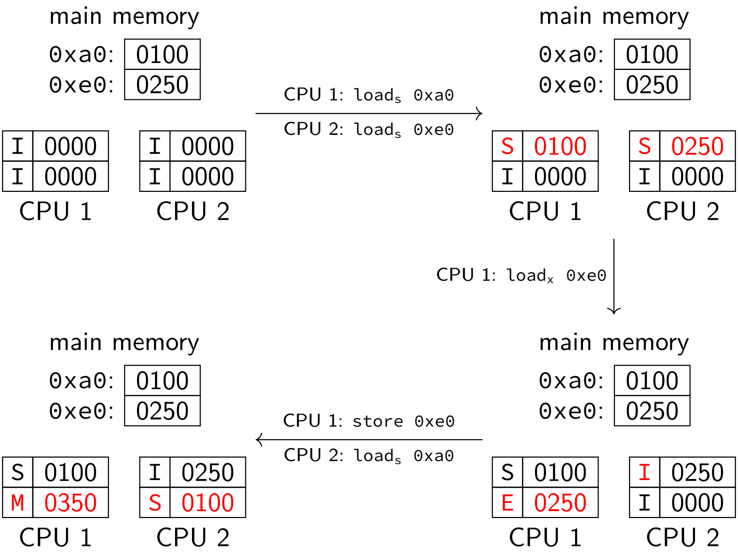


- Different cores can access the same memory at the same time
- Multiple cores potentially share caches
- Caches can be inclusive
- CPU must make sure that caching is consistent even with concurrent accesses → Communication between CPUs with a Cache Coherence Protocol

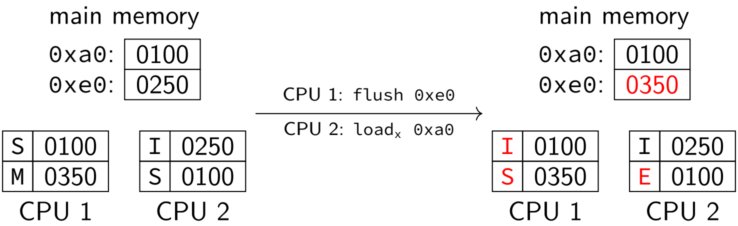
MESI Protocol

- CPUs and caches always read and write at cache line granularity, i.e. 64 byte
- The common MESI cache coherence protocol assigns every cache line one of the four states:
  - **M**odified: Cache line is stored in exactly one cache and was modified in the cache but not yet written back to main memory
  - **E**xclusive: Cache line is stored in exactly one cache to be used exclusively by one CPU
  - **S**hared: Cache line is stored in at least one cache, is currently used by a CPU for read-only access, and was not modified, yet
  - **I**nvalid: Cache line is not loaded or being used exclusively by another cache

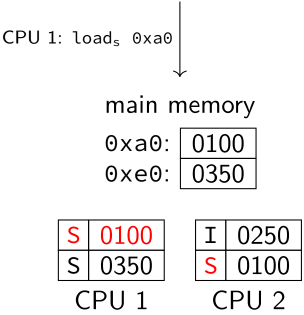
MESI Example (1)



MESI Example (2)



- Memory store instructions don't directly write to main memory
- Can be enforced with explicit flush instructions
- When a cache line is used by multiple CPUs, this can lead to *cache thrashing*
  → bad performance



Memory Accesses and Concurrency

Consider the following example program where `foo()` and `bar()` will be executed concurrently:

```
globalCounter = 0

function foo() {
    repeat 1000 times:
        globalCounter = globalCounter - 1
}

function bar() {
    repeat 1000 times:
        globalCounter = globalCounter + 1
}
```

Machine code for this program could look like this:

```
foo:
  load (globalCounter), %r1
  sub %r1, $1
  store %r1, (globalCounter)
```

```
bar:
  load (globalCounter), %r1
  add %r1, $1
  store %r1, (globalCounter)
```

What is the value of `globalCounter` at the end?

Memory Order

- Out-of-order execution and simultaneous multi-processing leads to unexpected execution of memory load and store instructions
- All executed instructions will complete eventually
- However, effects of memory instructions (i.e. reads and writes) can become visible in a non-deterministic order
- CPU vendors define how reads and writes are allowed to be interleaved $\rightarrow$ *memory order*
- Generally: Dependent instructions within a single thread always work as expected:

  ```
  store $123, A
  load A, %r1
  ```

  If the memory location at `A` is only accessed by this thread, `r1` will always contain 123

Weak and Strong Memory Order

- CPU architectures usually have either *weak memory order* (e.g. ARM) or *strong memory order* (e.g. x86)
- Weak Memory Order:
  - As long as dependencies are respected, memory instructions and their effects can be reordered
  - Different threads will see writes in different orders
- Strong Memory Order:
  - Within a thread, only stores are allowed to be delayed after subsequent loads, everything else is not reordered
  - When two threads execute stores to the same location, all other threads will see the resulting writes in the same order
  - Writes from a set of threads will be seen in the same order by all other threads
- For both:
  - Writes from other threads can be reordered
  - Concurrent memory accesses to the same location can be reordered

Example of Memory Order (1)

In this example, initially the memory at A contains the value 1, the memory at B the value 2.

Thread 1
```
store $3, A
```

Thread 2
```
store $4, B
```

Thread 3
```
load A, %r1
load B, %r2
```

Thread 4
```
load B, %r3
load A, %r4
```

Weak memory order:

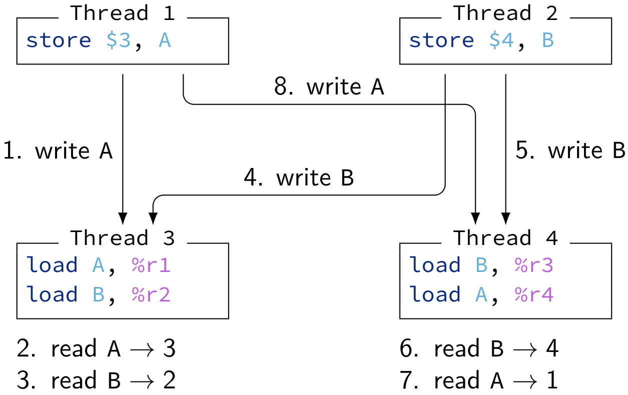
- Threads do not have dependent instructions
- Memory instructions can be reordered arbitrarily
- r1 $= 3$, r2 $= 2$, r3 $= 4$, r4 $= 1$ is allowed

Strong memory order:

- Threads 3 and 4 must see writes from threads 1 and 2 in the same order
- Example from weak memory order is not allowed
- r1 $= 3$, r2 $= 2$, r3 $= 4$, r4 $= 3$ is allowed

Example of Memory Order (2)

Visualization of the example for weak memory order:



- Thread 3 sees write to A (1.) before write to B. (4.)
- Thread 4 sees write to B (5.) before write to A. (8.)
- In strong memory order 5. is not allowed to happen before 8.

Memory Barriers

- Multi-core CPUs have special *memory barrier* (also called *memory fence*) instructions that can enforce stricter memory orders requirements
- This is especially useful for architectures with weak memory order
- x86 has the following barrier instructions:
  - `lfence`: Earlier loads cannot be reordered beyond this instruction, later loads and stores cannot be reordered before this instruction
  - `sfence`: Earlier stores cannot be reordered beyond this instruction, later stores cannot be reordered before this instruction
  - `mfence`: No loads or stores can be reordered beyond or before this instruction
- ARM has the *data memory barrier* instruction that supports different modes:
  - `dmb ishst`: All writes visible in or caused by this thread before this instruction will be visible to all threads before any writes from stores after this instruction
  - `dmb ish`: All writes visible in or caused by this thread and dependent reads before this instruction will be visible to all threads before any reads and writes after this instruction
- To additionally control out-of-order execution, ARM has the *data synchronization barrier* instructions: `dsb ishst`, `dsb ish`

Atomic Operations

- Memory order is only concerned about memory loads and stores
- Concurrent stores to the same memory location do not have any memory order constraints → order is possibly non-deterministic
- To allow deterministic concurrent modifications, most architectures support *atomic operations*
- An atomic operation is usually a sequence of: load data, modify data, store data
- Also called Read-Modify-Write (RMW)
- CPU ensures that all RMW operations are executed *atomically*, i.e. no other concurrent loads and stores are allowed in-between
- Usually only supported for individual arithmetic and bit-wise instructions

Atomic add on x86

```
lock addl $1, (%rdi)
```

Atomic add on ARM

```
ldrex   r1, [r0]
add     r1, r1, #1
strex   r2, r1, [r0]
```

Compare-And-Swap Operations (1)

- On x86, RMW instructions potentially lock the memory bus
- To avoid performance issues, only very few RMW instructions exist
- To facilitate more complex atomic operations, the *Compare-And-Swap* (CAS) atomic operation can be used
- ARM does not support locking the memory bus, so all RMW operations are implemented with CAS
- A CAS instruction has three parameters: The memory location `m`, the expected value `e`, and the desired value `d`
- The CAS operation conceptually works as follows:

```
tmp = load(m)
if (tmp == e) {
    store(m, d)
    success = true
} else {
    success = false
}
```

- Note: The CAS operation can fail, e.g. due to concurrent modifications!

Compare-And-Swap Operations (2)

Because CAS operations can fail, they are usually used in a loop with the following steps:

1. Load value from memory location into local register
2. Do computation with the local register assuming that no other thread will modify the memory location
3. Generate new desired value for the memory location
4. Do a CAS operation on the memory location with the value in the local register as expected value
5. Start the loop from the beginning if the CAS operation fails

Note that steps 2 and 3 can contain any number of instructions and are not limited to RMW instructions!

Compare-And-Swap Operations (3)

A typical loop using CAS looks like this:

```
success = false
while (not success) { (Step 5)
    expected = load(A) (Step 1)
    desired = non_trivial_operation(expected) (Steps 2, 3)
    success = CAS(A, expected, desired) (Step 4)
}
```

- With this approach, arbitrarily complex atomic operations on a memory location can be performed
- However, the likelihood for failure increases the more time is spent on the non-trivial operation
- Also, the non-trivial operation is potentially executed much more often than necessary

Parallel Programming

Parallel Programming

Multi-threaded programs usually contain many shared resources

- Data structures
- Operating system handles (e.g. file descriptors)
- Individual memory locations
- ...

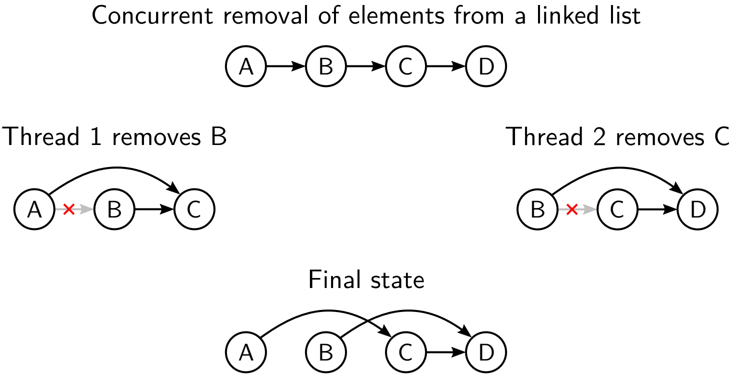
Concurrent access to shared resources needs to be controlled

- Uncontrolled access leads to race conditions
- Race conditions usually end in inconsistent program state
- Other outcomes such as silent data corruption are also possible

Synchronization can be achieved in different ways

- Operating system support, e.g. through mutexes
- Hardware support, especially through atomic operations

Mutual Exclusion (1)



Observations

- C is not actually removed
- Threads might also deallocate node memory after removal

Mutual Exclusion (2)

Protect shared resources by only allowing accesses within critical sections

- Only one thread at a time can enter a critical section
- Ensures that the program state is always consistent if used correctly
- Non-deterministic (but consistent) program behavior is still possible

There are various possibilities for implementing mutual exclusion

- Atomic test-and-set operations
  - usually requires spinning which can be dangerous
- Operating system support
  - E.g. mutexes in Linux

Locks

Implement mutual exclusion by acquiring locks on mutex objects

- Only one thread at a time can acquire a lock on a mutex
- Trying to acquire a lock on an already locked mutex will block the thread until the mutex becomes available again
- Blocked threads can be suspended by the kernel to free compute resources

Multiple mutex objects can be used to represent separate critical sections

- Only one thread at a time may enter the same critical section, but threads may simultaneously enter distinct critical sections
- Allows for more fine-grained synchronization
- Requires careful implementation to avoid deadlocks

Shared Locks

Strict mutual exclusion is not always necessary

- Commonly concurrent read-only accesses to the same shared resource do not interfere with each other
- Using strict mutual exclusion introduces an unnecessary bottleneck as readers would block each other
- We only need to make sure that write accesses can not happen concurrently with other write or read accesses

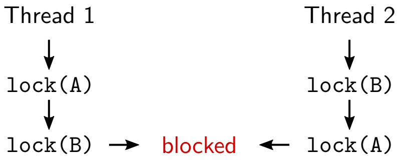
Shared locks provide a solution

- Threads can acquire either an exclusive or a shared lock on a mutex
- Multiple threads can simultaneously acquire a shared lock on a mutex if it is not locked exclusively
- One thread at a time can acquire an exclusive lock on a mutex if it is not locked in any other way (exclusive or shared)

Problems with Mutual Exclusion (1)

Deadlocks

- Multiple threads each wait for the other threads to release a lock



Avoiding deadlocks

- If possible, threads should never acquire multiple locks
- If not avoidable, locks must always be acquired in a globally consistent order

Problems with Mutual Exclusion (2)

Starvation

- High contention on a mutex may lead to some threads making no progress
- Can partially be alleviated by using less restrictive locking schemes

High latency

- Some threads are blocked for a long time if a mutex is highly contended
- Can lead to noticeably reduced system performance
- Performance can possibly even drop below single-threaded performance

Priority inversion

- A high-priority thread may be blocked by a low-priority thread
- Due to the priority differential, the low-priority thread may not be allowed sufficient compute resources to quickly release the lock

Hardware-Assisted Synchronization

Using mutexes is usually relatively expensive

- Each mutex requires some state (16 to 40 bytes)
- Acquiring locks potentially requires system calls which can take thousands of cycles or more

For this reason, mutexes are best suited for coarse-grained locking

- E.g. locking an entire data structure instead of parts of it
- Sufficient if only very few threads contend for locks on the mutex
- Sufficient if the critical section protected by the mutex is much more expensive than a (potential) system call to acquire a lock

The performance of mutexes quickly degrades under high contention

- In particular, the latency of lock acquisition increases dramatically
- This even occurs when we only acquire shared locks on a mutex
- We can exploit hardware support for more efficient synchronization

Optimistic Locking (1)

Often, read-only accesses to a resource are more common than write accesses

- Thus we should optimize for the common case of read-only access
- In particular, parallel read-only access by many threads should be efficient
- Shared locks are not well-suited for this (see previous slide)

Optimistic locking can provide efficient reader-writer synchronization

- Associate a *version* with the shared resource
- Writers still have to acquire an exclusive lock of some sort
  - This ensures that only one writer at a time has access to the resource
  - At the end of its critical section, a writer atomically increases the version
- Readers only have to read the version
  - At the begin of its critical section, a reader atomically reads the current version
  - At the end of its critical section, a reader validates that the version did not change
  - Otherwise, a concurrent write occurred and the critical section is restarted

Optimistic Locking (2)

Example (pseudocode)

```
writer(optLock) {
    lockExclusive(optLock.mutex) // begin critical section

    // modify the shared resource

    storeAtomic(optLock.version, optLock.version + 1)

    unlockExclusive(optLock.mutex) // end critical section
}

reader(optLock) {
    while(true) {
        current = loadAtomic(optLock.version); // begin critical section

        // read the shared resource

        if (current == loadAtomic(optLock.version)) // validate
            return; // end critical section
    }
}
```

Optimistic Locking (3)

Why is optimistic locking efficient?

- Readers only have to execute two atomic load instructions
- This is much cheaper than acquiring a shared lock
- But requires that modifications are rare, otherwise readers have to restart frequently

A careful implementation of readers is required

- The shared resource may be modified while a reader is accessing it
- We cannot assume that we read from a consistent state
- Additional intermediate validation may be required for more complex read operations

Beyond Mutual Exclusion

In many cases, strict mutual exclusion is not required in the first place

- E.g. parallel insertion into a linked list
- We do not care about the order of insertions
- We only need to guarantee that all insertions are reflected in the final state

This can be implemented efficiently by using atomic operations (pseudocode)

```
threadSafePush(linkedList, element) {
    while (true) {
        head = loadAtomic(linkedList.head)
        element.next = head
        if (CAS(linkedList.head, head, element))
            break;
    }
}
```

Non-Blocking Algorithms

Algorithms or data structures that do not rely on locks are called *non-blocking*

- E.g. the `threadSafePush` function on the previous slide
- Synchronization between threads is usually achieved using atomic operations
- Enables more efficient implementations of many common algorithms and data structures

Such algorithms can provide different levels of progress guarantee

- Wait-freedom: There is an upper bound on the number of steps it takes to complete each operation
  - Hard to achieve in practice
- Lock-freedom: At least one thread makes progress if the program is run for sufficient time
  - Often informally (and technically incorrectly) used as a synonym for non-blocking

A-B-A Problem (1)

Non-blocking data structures need to be implemented carefully

- We do not have the luxury of critical sections anymore
- Threads can execute different operations on a data structure in parallel (e.g. insert and remove)
- The individual atomic operations comprising these compound operations can be interleaved arbitrarily
- This can lead to hard-to-debug anomalies, such as lost updates or the A-B-A problem

Often problems can be avoided by making sure that only the same operation (e.g. insert) is executed in parallel

- E.g. insert elements in parallel in a first step, and remove them in parallel in a second step

A-B-A Problem (2)

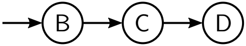
Consider the following simple linked-list based stack (pseudocode)

```
threadSafePush(stack, element) {
    while (true) {
        head = loadAtomic(stack.head)
        element.next = head
        if (CAS(stack.head, head, element))
            break;
    }
}

threadSafePop(stack) {
    while (true) {
        head = loadAtomic(stack.head)
        next = head.next
        if (CAS(stack.head, head, next))
            return head
    }
}
```

A-B-A Problem (3)

Consider the following initial state of the stack, on which two threads perform some operations in parallel



Thread 1

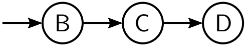
```
x = threadSafePop(stack)
```

Thread 2

```
y = threadSafePop(stack)
z = threadSafePop(stack)
threadSafePush(stack, y)
```

A-B-A Problem (4)

Our implementation would allow the execution to be interleaved as follows

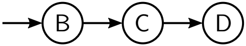


Thread 1

Thread 2

A-B-A Problem (5)

Our implementation would allow the execution to be interleaved as follows



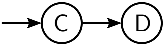
Thread 1

```
head = loadAtomic(stack.head)
// head == B
next = head.next
// next == C
```

Thread 2

A-B-A Problem (6)

Our implementation would allow the execution to be interleaved as follows



Thread 1

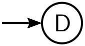
```
head = loadAtomic(stack.head)
// head == B
next = head.next
// next == C
```

Thread 2

```
y = threadSafePop(stack)
// y == B
```

A-B-A Problem (7)

Our implementation would allow the execution to be interleaved as follows



Thread 1

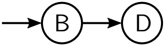
```
head = loadAtomic(stack.head)
// head == B
next = head.next
// next == C
```

Thread 2

```
y = threadSafePop(stack)
// y == B
z = threadSafePop(stack)
// z == C
```

A-B-A Problem (8)

Our implementation would allow the execution to be interleaved as follows



Thread 1

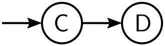
```
head = loadAtomic(stack.head)
// head == B
next = head.next
// next == C
```

Thread 2

```
y = threadSafePop(stack)
// y == B
z = threadSafePop(stack)
// z == C
threadSafePush(stack, y)
```

A-B-A Problem (9)

Our implementation would allow the execution to be interleaved as follows



Thread 1

```
head = loadAtomic(stack.head)
// head == B
next = head.next
// next == C





CAS(stack.head, head, next)
// inconsistent state!
```

Thread 2

```




y = threadSafePop(stack)
// y == B
z = threadSafePop(stack)
// z == C
threadSafePush(stack, y)
```

The Dangers of Spinning (1)

It is possible to implement a “better” mutex that requires less space and uses no system calls by using atomic operations:

- The mutex is represented in a single atomic integer
- It has the value 0 when it is unlocked, 1 when it is locked
- To lock the mutex, the value is changed atomically to 1 only if it was 0 by using a CAS
- The CAS is repeated as long as another thread holds the mutex

```
function lock(mutexAddress) {
    while (CAS(mutexAddress, 0, 1) not sucessful) {
        <noop>
    }
}

function unlock(mutexAddress) {
    atomicStore(mutexAddress, 0)
}
```

The Dangers of Spinning (2)

Using this CAS loop as a mutex, also called *spin lock*, has several disadvantages:

- It has no fairness, i.e. does not guarantee that a thread will acquire the lock eventually → *starvation*
- The CAS loop consumes CPU cycles (waste of energy and resources)
- Can easily lead to *priority inversion*
  - The scheduler of the operating system thinks that the spinning thread requires a lot of CPU time
  - The spinning thread actually does no useful work at all
  - In the worst-case, the scheduler takes CPU time away from the thread that holds the lock to give it to the spinning thread

  → Spinning thread needs to spin even longer which makes the situation worse

Possible solutions:

- Spin for a limited number of times (e.g. several hundred thousand iterations)
- If the lock could not be acquired, fall back to a "real" mutex
- This is actually already how mutexes are usually implemented