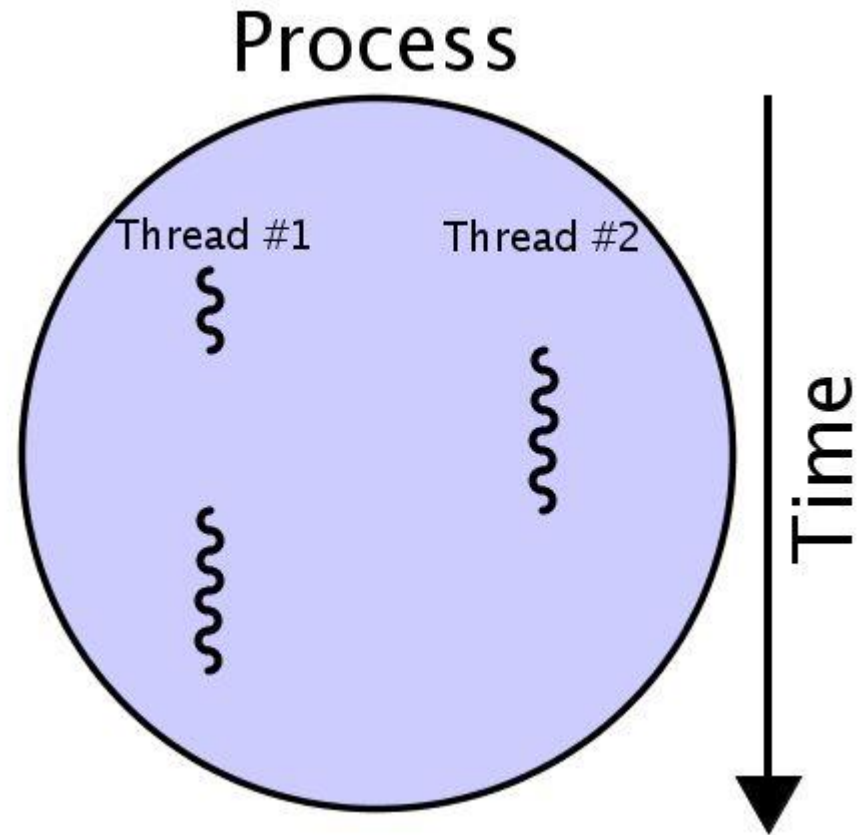


Synchronization



Terminology

- Threads
- Processes (or tasks)
 - Independent
 - Considerable state info
 - Separate address space
- Resources
 - Memory, file handles, sockets, device handles
 - Owned by processes





Threads and processes

- Thread of a same process share the same resources
- Processes can share resources only through explicit methods
- The only resources exclusively owned by a thread are the thread stack, register status and thread-local storage (if any)



Scheduling

- A scheduler is responsible to allocate tasks and threads to the available computing resources
- Various kind of scheduling algorithms
 - Best-effort → minimize makespan
 - Real-Time → meet deadlines
 - ... other metrics to minimize
- Preemption and context changes
- Access to shared resources



Multithreading

- Classic multithreading on single processor systems required Time Division Multiplexing (TDM)
 - Time driven
 - Event driven
- Multiprocessors → different threads and processes can run on different CPUs
- Multithreading is easier (“native”) on multicore platforms
- But scheduling requires more attention



Multithreading issues

- Race conditions
- Starvation, priority inversion, deadlock, livelock
 - Mamihlapinatapai
- Synchronization (mutex, lock)
- Atomic execution (semaphores)
- Communication:
 - shared-memory (requires locking)
 - message-passing (slower but easier)



Different kinds of Parallelisms

- Instruction Level Parallelism (ILP)
- Data Level Parallelism (DLP)
- Thread Level Parallelism (TLP)



Instruction Level Parallelism (ILP)

- Execute multiple instruction per clock cycle
- Each functional unit on a core is an execution resource within a single CPU:
 - Arithmetic Logic Unit (ALU)
 - Floating Point Unit (FPU)
 - bit shifter, multiplier, etc.
- Need to solve data dependencies



Data dependency

- Consider the sequential code:
 1. $e = a + b$
 2. $f = c + d$
 3. $g = e * f$
- Operation 3. depends on the results of operations 1. and 2.
- Cannot execute 3. before 1. and 2. are completed



How to “parallelize” software?

- Parallelism can be extracted from ordinary programs
 - At run-time (by complex specific HW)
 - At compile-time (simplifying CPU design and improving run-time performances)
- Degree of ILP is application dependent

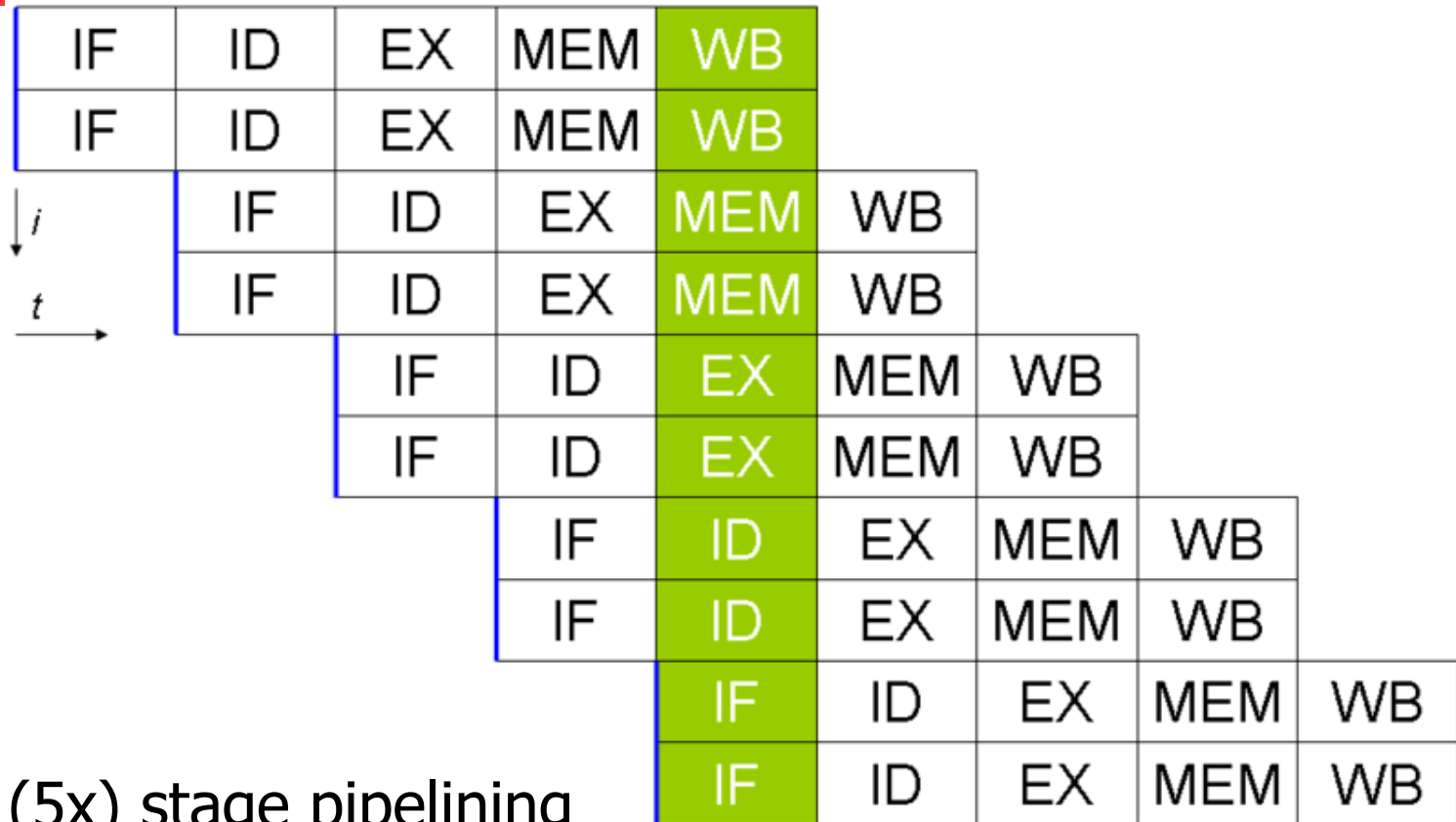


ILP: superscalar architectures

- Data dependency check in HW
- Complex mechanisms
 - power, die-space and time consuming
- Problematic when
 - code difficult to predict
 - Instructions have many interdependencies



Superscalar pipelining



(5x) stage pipelining

(2x) Superscalar execution



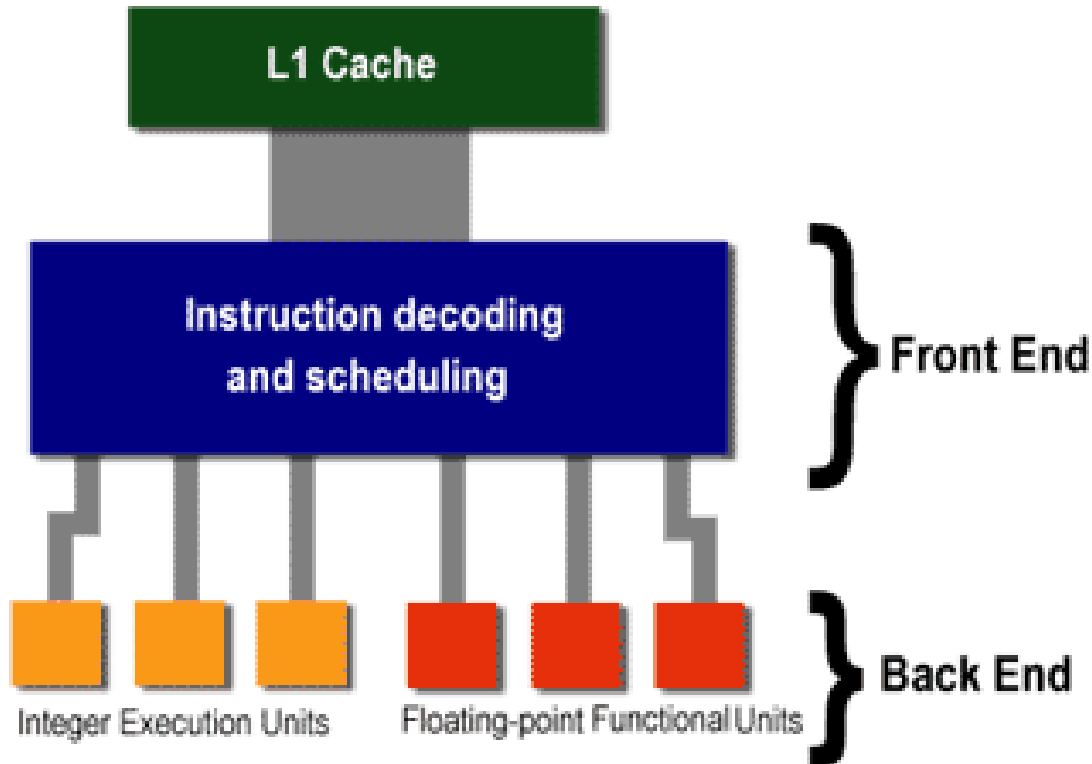
ILP optimizations

- Instruction pipelining
- Superscalar execution
- Out-of-order execution
 - deferred memory accesses
 - combined load and store
- Speculative execution
 - branch prediction,
 - speculative load
 - ...



Processor front and back end

- Intel describes its processors having
 - “in-order front end”
 - “out-of-order execution engine”





ILP: compile-time techniques

- Compiler decides which operations can run in parallel
- Removes the complexity of instruction scheduling from HW to SW
- New instruction sets that explicitly encode multiple independent operations per instruction
 - Very Long Instruction Word (VLIW): one instruction encodes multiple operations (one for each execution unit)
 - Explicitly Parallel Instruction Computing (EPIC): adds features to VLIW (cache prefetching instructions, ...)



Data Level Parallelism (DLP)

- Higher parallelism than superscalar architecture
- SIMD instructions (Single Instruction, Multiple Data)
 - Intel's MMX, SSE, SSE2, SSE3, SSE3, SSSE3, SSE4, AVX
 - AMD's 3DNow!, SSE5
 - ARM's NEON, IBM's AltiVec and SPE, etc.
 - Graphic cards (GPU)
 - Cell Processor's SPU
- Useful when the same operation has to be applied to a large set of data (i.e., multimedia, graphic operations on pixels, etc.)
- Multiple data are read and/or modified at the same same



Thread Level Parallelism (TLP)

- Higher level parallelism than ILP
- Different kinds of TLP
 - Superthreading
 - Hyperthreading or Symultaneous MultiThreading (SMT)
 - Needs superscalar processor
 - Chip-level MultiProcessing (CMP)
 - Needs multicore architecture
 - Combinations of the above solutions



Superthreading

- Temporal Multithreading (fine- or coarse-grained) → when processor idle, execute instruction of another thread
- Makes better use of the computing resources when a thread is blocked
- Requires adequate hardware support to minimize context change overhead

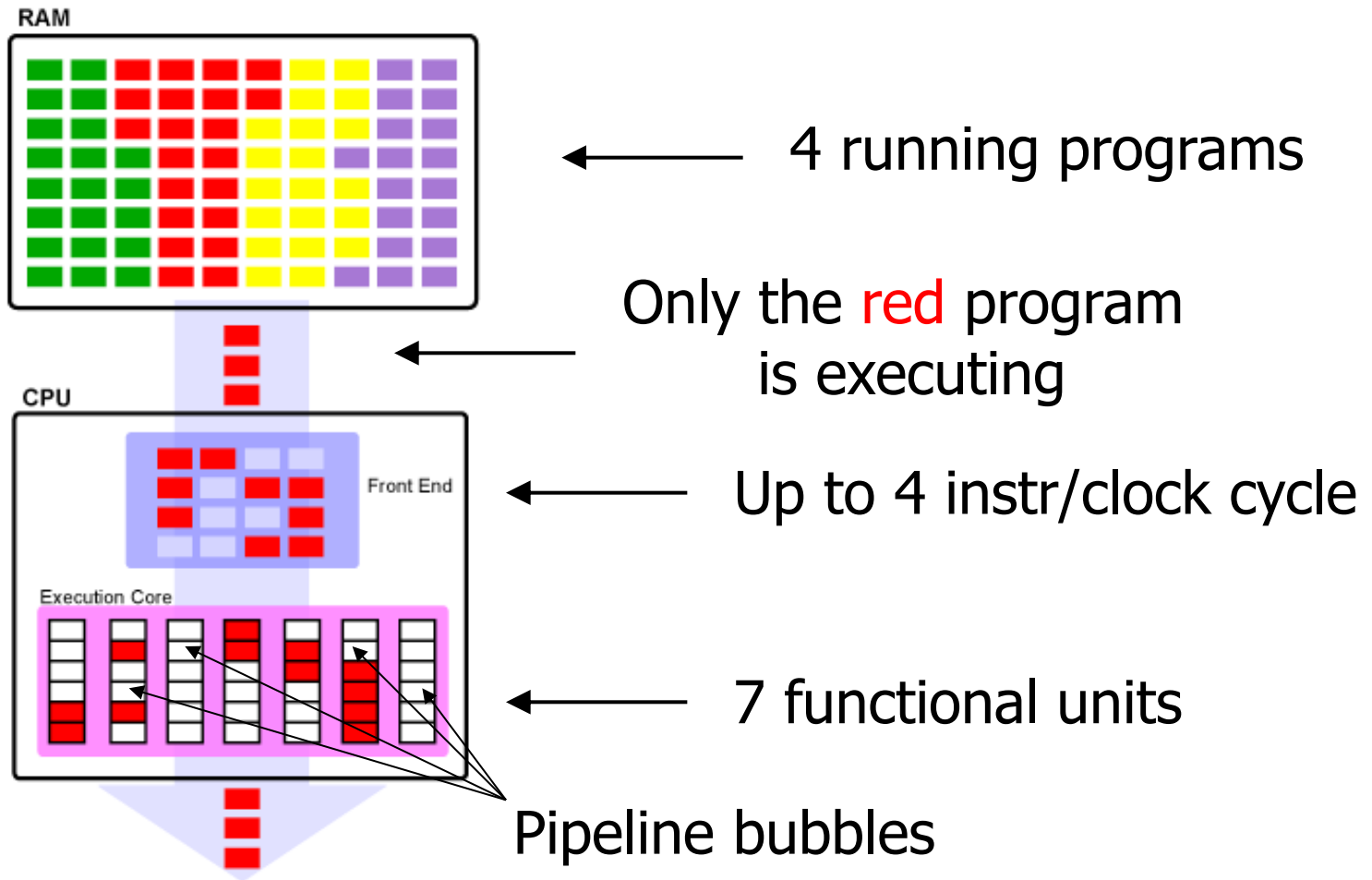


Hyperthreading

- Simultaneous MultiThreading (SMT)
- Introduced in late 90s: Intel's Pentium 4
- Execute instructions from multiple threads simultaneously → needs superscalar support
- Energy inefficient
 - Increases cache thrashing by 42%, whereas dual core results in a 37% decrease

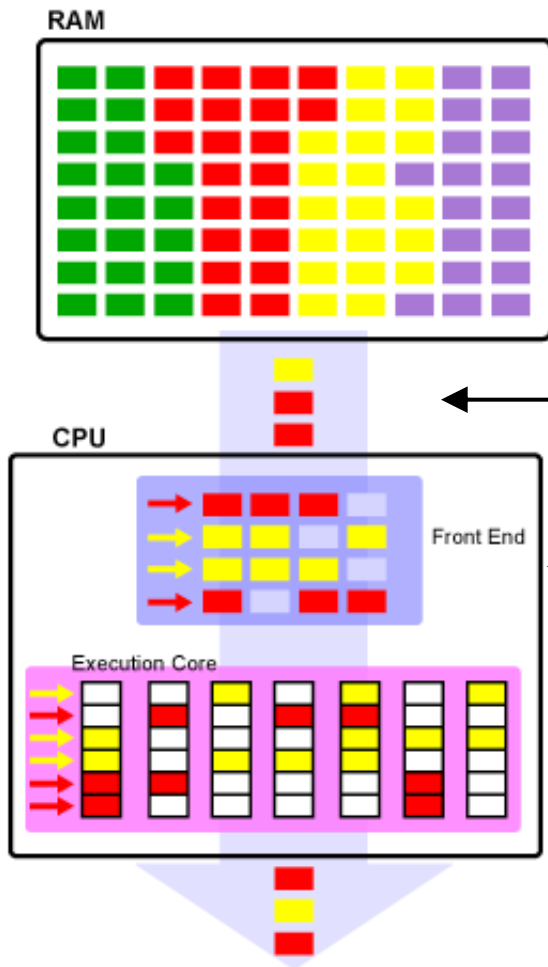


Single-threaded CPU





Super-threading (time-slice multithreading)



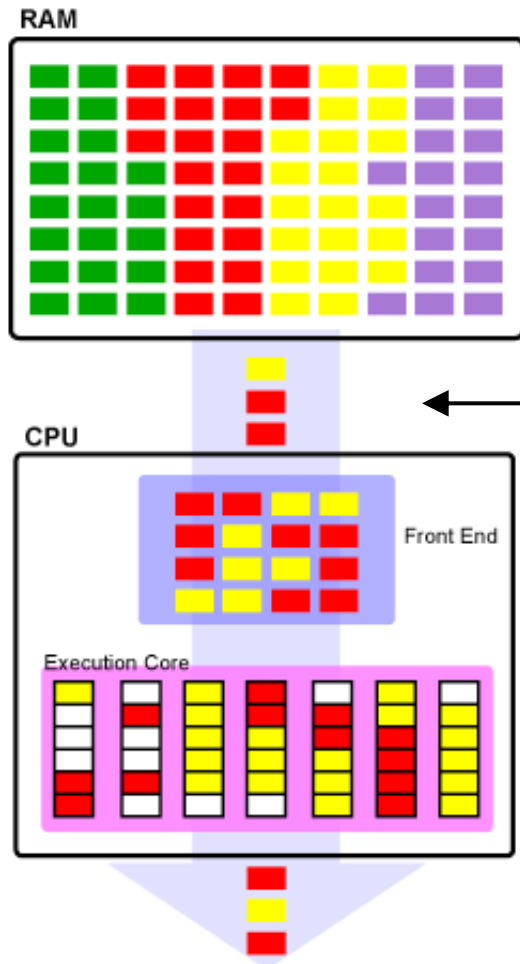
Multithreaded processor:
able to handle more than
one thread at a time

Interleaved execution of
different threads (helps
masking memory latency)

All instructions in a pipeline
stage must come from
the **same** thread



Hyper-threading (Simultaneous MultiThreading)



Interleaved execution of different threads (helps masking memory latency)

Instructions in a pipeline stage may come from **different** threads



Hyper-threading (SMT)

- From OS perspective: many “logical” processors
- Average ILP for a thread = 2.5 instr/cycle
 - Pentium 4 issues at most 3 instr/cycle to the execution core
- Hyperthreaded processor can exploit parallelism beyond a single thread ILP



But..

- SMT can be worse than non-SMT approaches
 - A thread monopolizing an execution unit for many consecutive pipeline stages can stall the available functional units (that could have been used by other threads)
- Cache thrashing problem
 - Different logical processor can execute two threads accessing completely different memory areas



Not a problem for Multicores

- A smart SMT-aware OS running on a multicore would schedule two different tasks on different processors → resource contention is minimized



Summary

- Out-of-order execution problems on multicore platforms
- Data dependencies
- Race conditions
- Memory barriers
- Locking mechanisms
- Spinlocks, semaphores, mutexes, RCU



Introduction: execution ordering

- A processor can execute instructions in any order (or in parallel), provided it maintains **program causality** with respect to itself
- The compiler may reorder instructions in many ways, provided causality maintenance
- Some CPUs are more constrained than others
 - E.g., i386, x86_64, UltraSPARC are more constrained than PowerPC, Alpha, etc.
 - Linux assumes the DEC Alpha execution ordering → the most relaxed one



Out-of-order execution

- Loads are more likely to need to be completed immediately
- Stores are often deferred
- Loads may be done speculatively, leading to discarded results
- Order of memory accesses may be rearranged to better use buses and caches
- Multiple loads and stores can be parallelized



Optimizations

- CPU or Compiler optimizations: overlapping instructions may be replaced
 - E.g., two consecutive loads to the same value/register

1) $A = V;$
2) $A = W;$ \longrightarrow $A = W;$

- A load operation may be performed entirely inside the CPU

1) $*P = A;$
2) $B = *P;$ \longrightarrow 1) $*P = A;$
2) $B = A;$



Cache coherency is not enough

- While the caches are expected to be coherent, there's no guarantee that that coherency will be ordered
- Whilst changes made on one CPU will eventually become visible on all CPUs, there's no guarantee that they will become apparent in the same order on those other CPUs



Causality maintenance

- If an instruction in the stream depends on an earlier instruction, then that earlier instruction must be “*sufficiently complete*” before the later instruction may proceed
- Need to analyse the *dependencies* between operations



Dependencies

- A “dependency” is a situation in which an instruction refers to the data of a preceding instruction
- Data dependencies
 - Read after Write
 - Write after Read
 - Write after Write
- Control dependencies



Read after Write

- True (or flow) dependence:
 - an instruction depends on the result of a previous instruction
- Example:

```
1) A = 3;  
2) B = A + 1;
```

A is modified by the first instruction and used in the second one



It is not possible to use instruction level parallelism



Write after Read

- Antidependence:
 - an instruction requires a value that is updated by a later instruction
- Example:

```
1) B = A + 1;  
2) A = 3;
```

A is modified by the second instruction and used by the first one



It is not possible to use instruction level parallelism



Write after Write

- Output dependence:
 - two instructions modify the same resource
- Example:

```
1) A = 4;  
2) A = 5;
```

A is modified by both
instructions



It is not possible to use
instruction level parallelism



Other dependencies

- Read after Read a.k.a Input dependency
 - two instructions read the same resource

```
1) B = A + 1;  
2) C = A + 2;
```

A is read by both instructions



ILP is possible!

- Control dependency
 - instruction execution depends on a previous instruction → e.g., conditional statements

```
if (x == true)  
    A = 5;
```

“A = 5” executed only if x is true in the previous instruction



Considerations on dependencies

- No need to consider dependencies when a strict sequential execution model is used
- Otherwise, dependence analysis is needed for
 - CPU exploiting instruction level parallelism
 - out-of-order execution
 - parallelizing compiler
- Need to enforce execution-order constraints to limit the ILP of a system
- Usually automatically solved by compiler/HW
- Concerns the behavior of a single thread



More than dependencies...

- When more threads are concurrently executed → additional “dependency” problems
- Instructions from different tasks/threads may need to access the same global resources
- Problems similar to dependency problems, but more difficult to detect/solve
- Explicit programmer intervention is needed
- Need to synchronize the accesses to shared resources



Synchronization mechanisms

- Process synchronization
 - Barrier
 - Lock/semaphore
 - Critical sections
 - Thread join
 - Mutex
 - Non-blocking synchronization
- Data synchronization
 - Keep multiple copies of a set of data coherent with one another
 - E.g., cache coherency, cluster file systems, RAID, etc.



Process synchronization problems

- Race conditions
- Deadlock
- Livelock
- Starvation
- Lack of fairness



Race conditions

- More tasks read/modify the same global variable
- “Race” for which task modifies the global value first
- Output depends on the particular sequence of task operations
- Non-deterministic behavior



Race conditions: example 1

```
global int A = 0

task T1 ()
    A = A + 1
    print A
end task

task T2 ()
    A = A + 1
    print A
end task
```

- Two concurrent tasks T1 and T2 are activated
- Both tasks modify the same global variable A
- Expected behavior:
 - T1 reads 0
 - T1 increments A to 1
 - T1 prints 1
 - T2 reads 1
 - T2 increments A to 2
 - T2 prints 2
- **Expected final output = 2**



Race conditions: example 1

```
global int A = 0

task T1 ()
    A = A + 1
    print A
end task

task T2 ()
    A = A + 1
    print A
end task
```

- Problem if operation “A=A+1” is not performed atomically
- Possible behavior:
 - T1 reads 0
 - T2 reads 0
 - T1 increments A to 1
 - T2 increments A to 1
 - T1 prints 1
 - T2 prints 1
- **Final output = 1 !!!**



Race conditions: example 2

Two global variables

{ A = 1; B = 2 }

CPU1

```
A = 3;  
B = 4;
```

with out-of-
order execution

CPU2

```
x = A;  
y = B;
```

- $n! = 24$ different combinations for memory accesses
- 4 different outputs:
 $(x,y) = (1,2); (1,4); (3,2); (3,4)$



Race conditions: example 3

Five global variables

{ A = 1, B = 2, C = 3, P = &A, Q = &C }

CPU1

```
B = 4;  
P = &B;
```

with out-of-
order execution

CPU2

```
Q = P;  
D = *Q;
```

NB: *Data dependency* → Q is modified in the first instruction and used in the second one



Race conditions: example 3

Five global variables

{ A = 1, B = 2, C = 3, P = &A, Q = &C }

CPU1

```
B = 4;  
P = &B;
```

with out-of-
order execution

CPU2

```
Q = P;  
D = *Q;
```

- Partial order enforcement by CPU2 → only 12 different combinations for memory accesses and three different outputs:
(Q=P,D) = (&A,1); (&B,2); (&B,4)
- D will never receive the value in C



Race conditions on devices

- Some devices present their control interfaces as collections of memory locations
- The order in which control registers are accessed may be crucial
- E.g.: Ethernet card with internal register set accessed through address (A) and data (D) port registers

To read internal register #5:

```
A = 5;  
x = *D;
```



Race conditions on devices

Read register #5:

```
A = 5;  
x = *D;
```

- Out-of-order execution may cause second instruction be executed before the first one
→ malfunction!
- Since there is no explicit data dependency, these problems are difficult to detect



Things that can be assumed

- Dependent memory access on a given CPU are always executed in order

```
Q = P;  
D = *Q;
```

- Overlapping load and stores within a CPU maintain functional dependencies

```
A = *P;  
*P = B;
```

or

```
*P = A;  
B = *P;
```



...and that cannot be assumed

- Order of execution of independent load and stores

```
A = *P;  
B = *Q;  
*R = C;
```

- Order of execution of overlapping load and stores (preserving functional dependencies)

```
1) A = *P;  
2) B = *(P + 4);
```

or

```
1) *P = A;  
2) B = *P;
```

may be: 1 → 2
2 → 1
1 & 2

may be: 1 → 2
1 & 2 (*P=B=A)



Memory barriers (membar)

- Class of instructions to enforce an order on memory operations
- Low-level machine code operating on shared memory
- Ensures that all operations preceding a membar are executed before all operations following the barrier



Membar: example

CPU 1:

```
load  eax, 0
while (eax == 0)
    load  eax, [a]
print  [b]
```

CPU 2:

```
store 33, [b]
store 1, [a]
```

- Two CPUs, each one running a task using global variables a and b
- Expected behavior:
 - CPU1 spins until a becomes $\neq 0$
 - Then prints the b value stored by CPU2
- Expected output: 33



but...

CPU 1:

```
load  eax, 0
while (eax == 0)
    load  eax, [a]
print  [b]
```

CPU 2:

```
store 33, [b]
store 1, [a]
```

- When CPU2's operations may be executed out-of-order:
 - a may be updated before b
 - CPU1 prints a meaningless value
- Not acceptable!



Solution: memory barrier

CPU 1:

```
load  eax, 0
while (eax == 0)
    load  eax, [a]
print  [b]
```

CPU 2:

```
store 33, [b]
membar
store 1, [a]
```

- Use a memory barrier before CPU2's second instruction
- "store 33, [b]" will be always executed before "store 1, [a]"
- Final output: 33



Why/when using membars?

- Needed when out-of-order execution or concurrent programming is used
- Many tricks to improve performances
 - reordering
 - deferral and combination of memory operations;
 - speculative loads and branch prediction
 - various types of caching
- Memory barriers allow overriding such tricks
- Instruct the compiler and the CPU to restrict the order



Varieties of memory barriers

1. Write (or store) memory barriers
2. Data dependency barriers
3. Read (or load) memory barriers
4. General memory barriers
5. Implicit varieties:
 - LOCK operations
 - UNLOCK operations



Write (or store) memory barrier

- All STORE operations before the barrier will appear (to other system components) to happen before all STORE operations after the barrier
- No effect on load operations



Wr_membar: example

```
A = 4;  
P = &A;
```

No data dependency:
later load operations referencing
*P may obtain old values $\neq 4$

- To enforce partial order between both operations \rightarrow insert a Wr_membar:

```
A = 4;  
Wr_membar;  
P = &A;
```

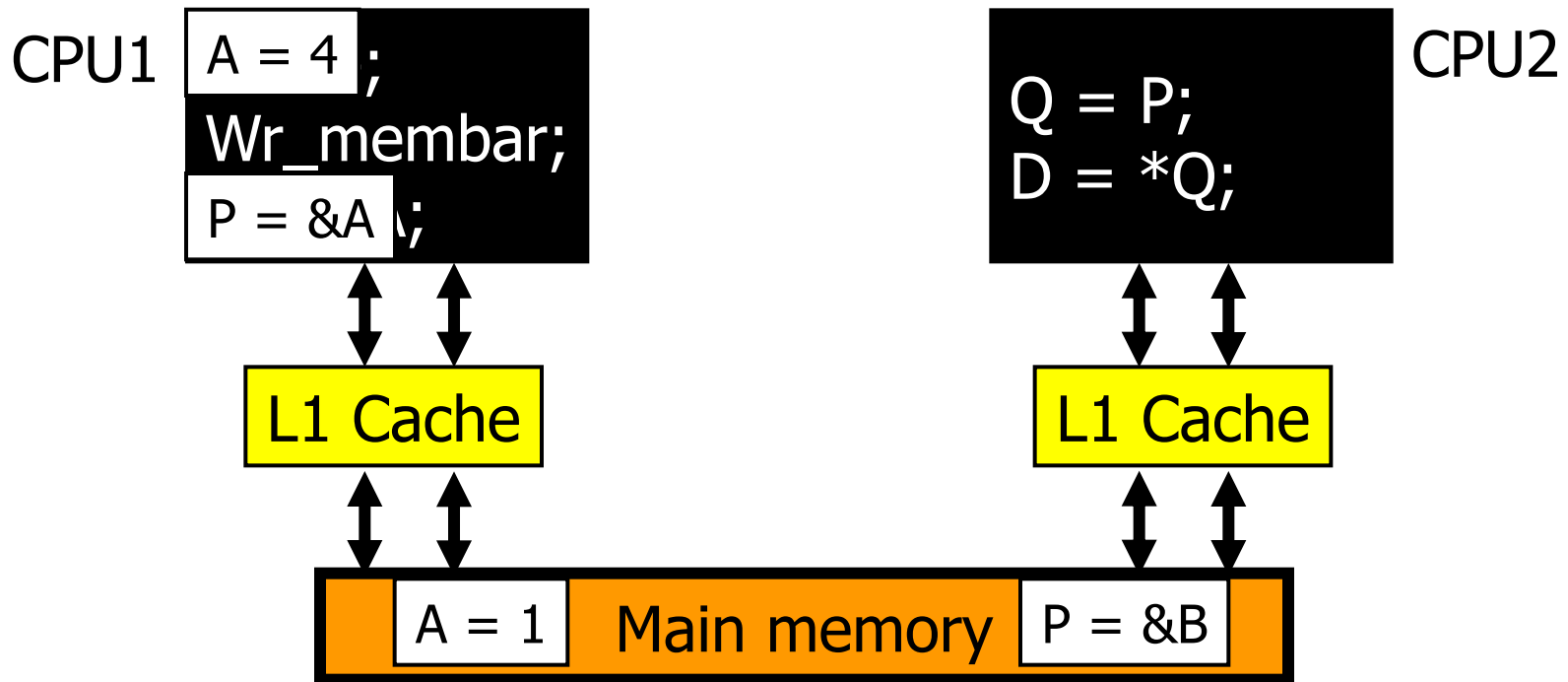
Later loads of *P (by the
same processor) will obtain
the updated value = 4



However..

- Other processors can see the update of A after the update of P, due to caching:

{ A = 1, B = 2, C = 3, P = &B, Q = &C }

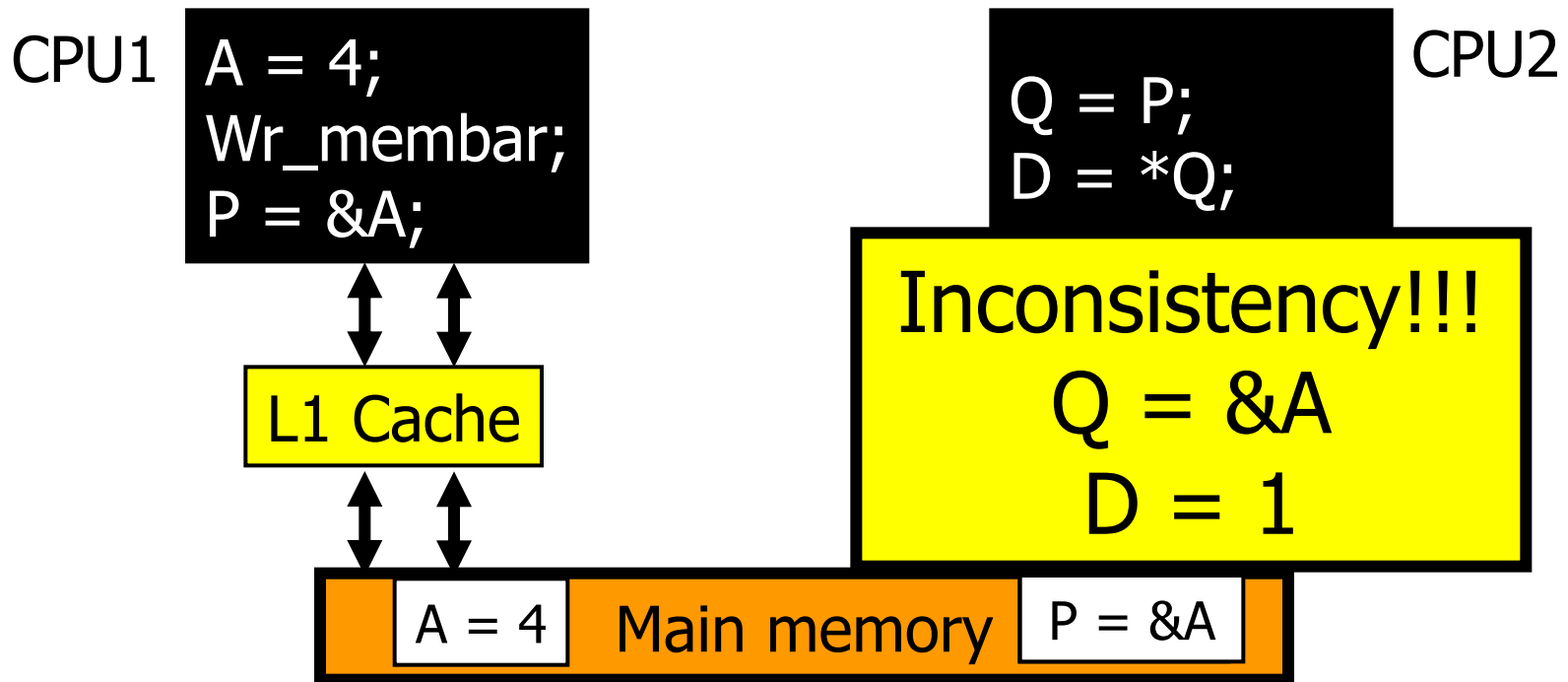




However..

- Other processors can see the update of A after the update of P, due to caching:

{ A = 1, B = 2, C = 3, P = &B, Q = &C }





Solution:

- Delay the read of $*Q$ until its value has been updated in memory
- The process running on CPU2 needs to synchronize with the write barrier on CPU1
- Make sure that the target ($*Q$) of the second load ($D=\text{load } *Q$) is updated, before the address (Q) obtained by the first load ($Q=\text{load } P$) is accessed



Data dependency barrier

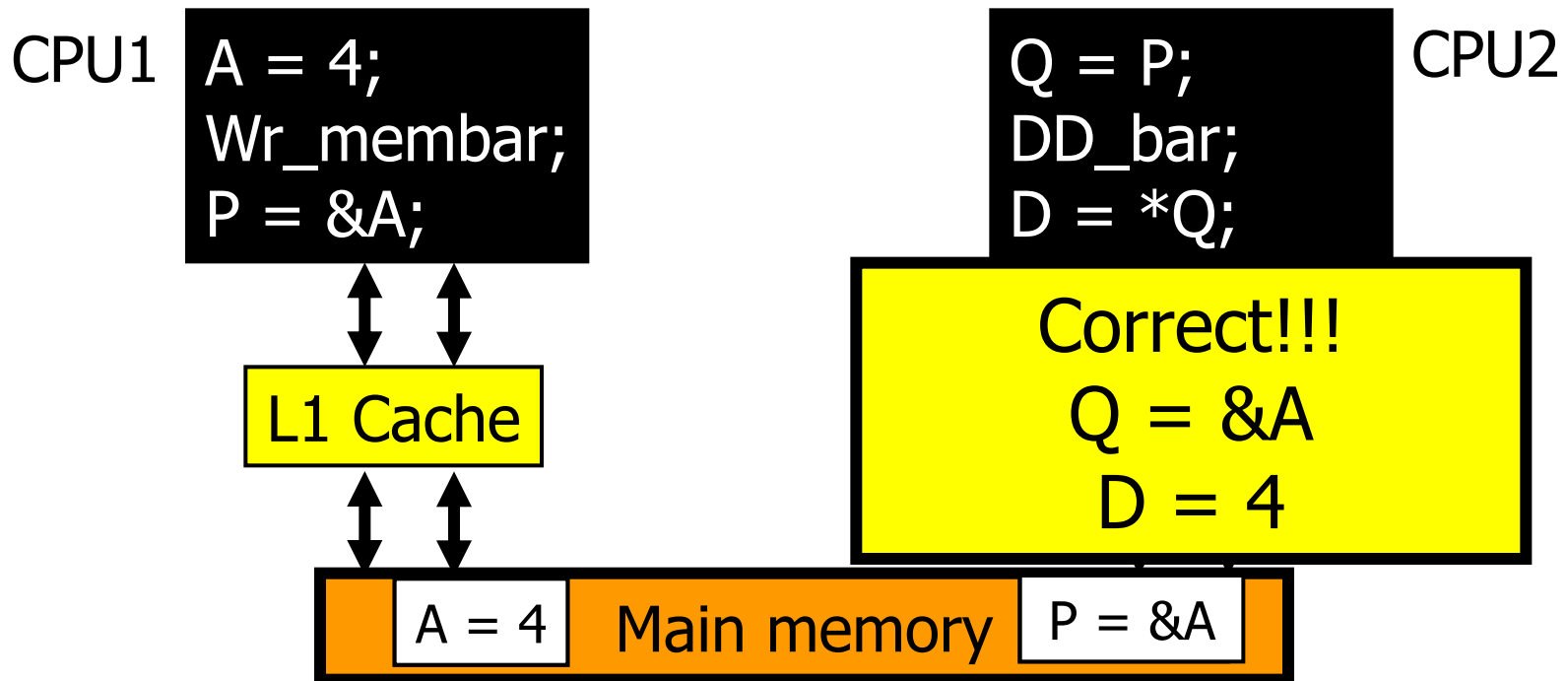
- Enforce a partial ordering on interdependent load operations
- Does not have any effect on
 - stores
 - independent or overlapping loads
- Weaker form of read memory barrier



Example with DD_bar

- CPU2 cache is forced to commit its coherency queue before processing further requests

{ A = 1, B = 2, C = 3, P = &B, Q = &C }





When using DD barriers?

- Two or more consecutive loads with later loads depending on the result of previous ones
- Typical situations:
 - the first load retrieves the address to which the second load will be directed
 - the first load retrieves a number which is then used to calculate the index for an array
- A data dependency barrier would be required to make sure that the target of the second load is updated before the address obtained by the first load is accessed



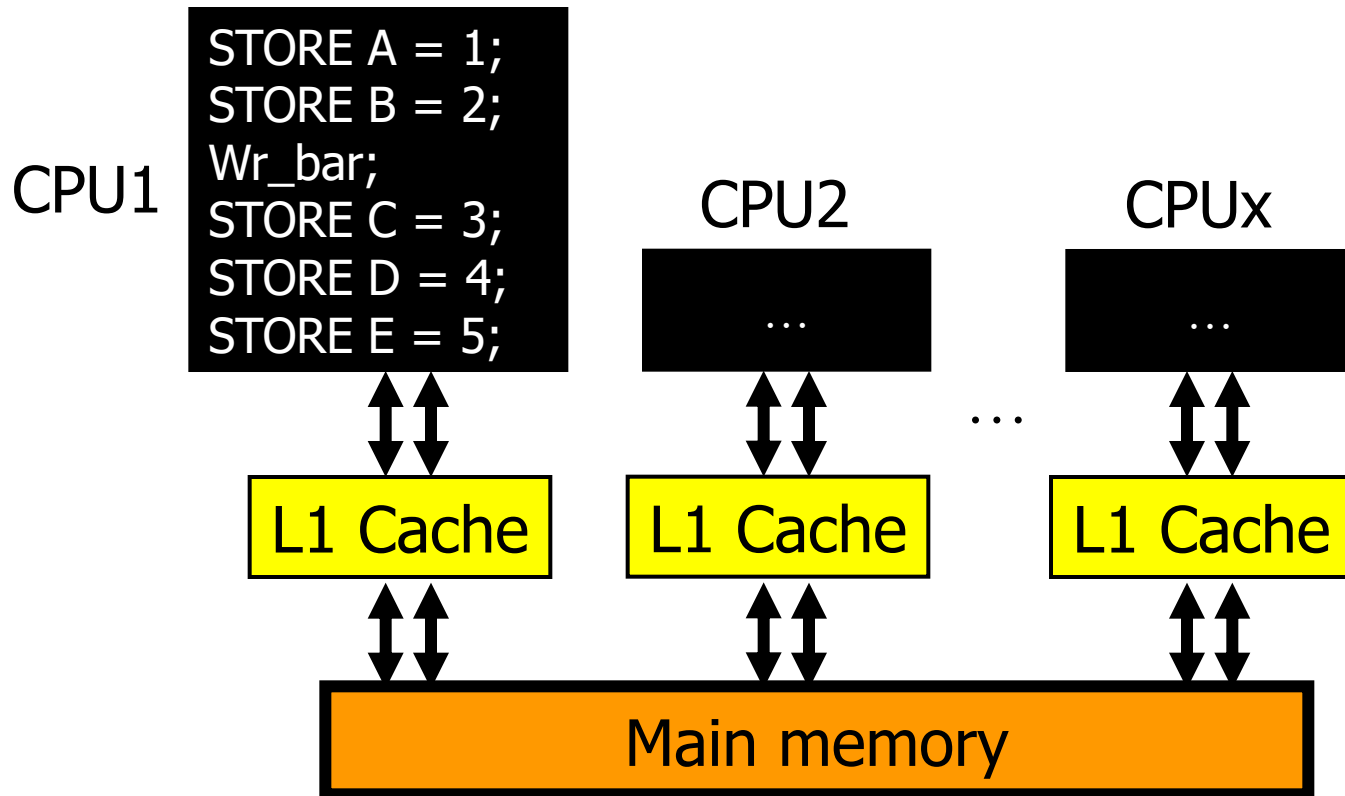
More in detail..

- Definition from Linux kernel manual
 - “A ***data dependency barrier*** issued by the CPU under consideration guarantees that for any load preceding it, if that load touches one of a sequence of stores from another CPU, then by the time the barrier completes, the effects of all the stores prior to that touched by the load will be perceptible to any loads issued after the data dependency barrier”.



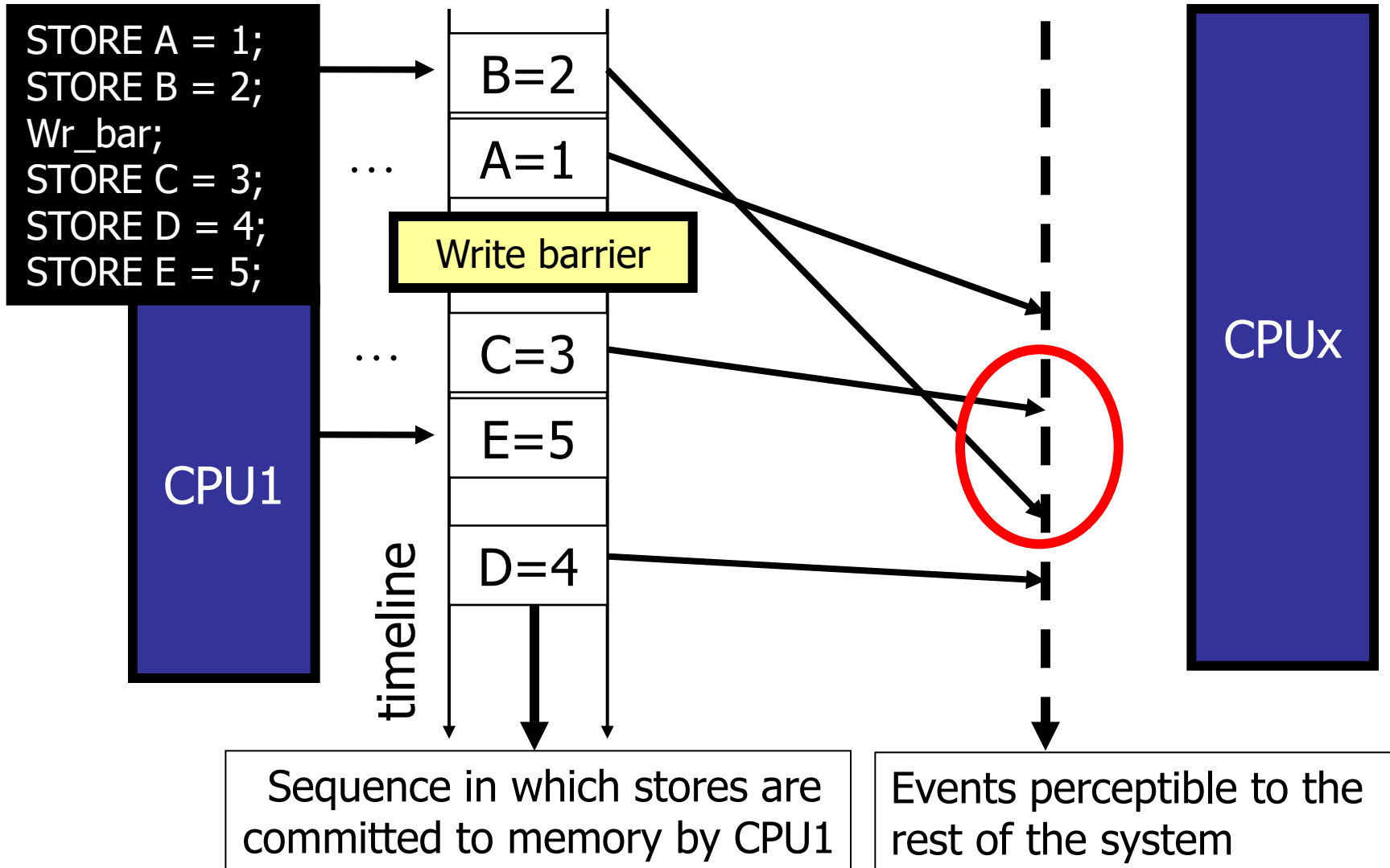
Let's try to understand..

- CPUs in the system can be viewed as committing sequences of stores to memory, that other CPU can then perceive





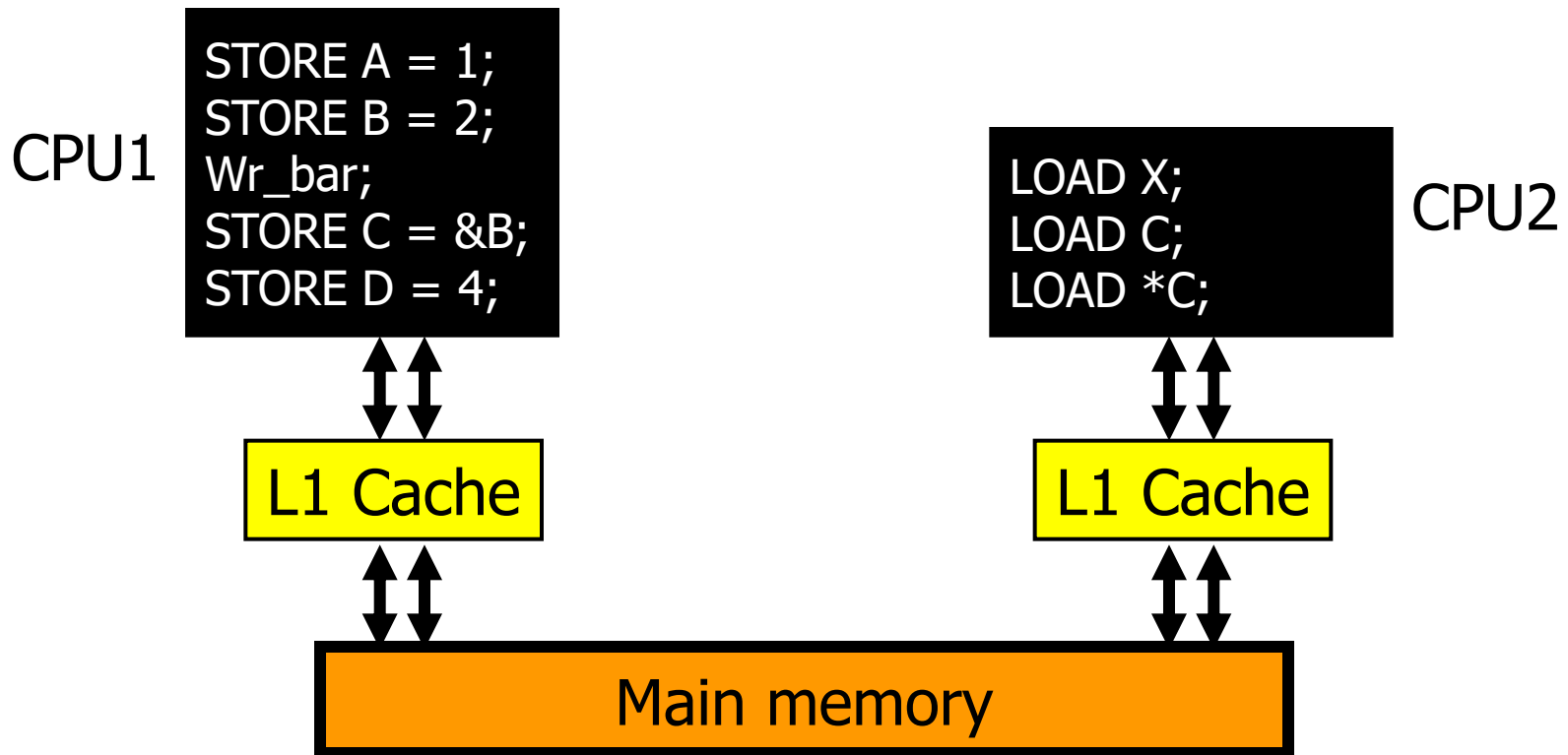
Sequence of memory ops





Example without DD_bar

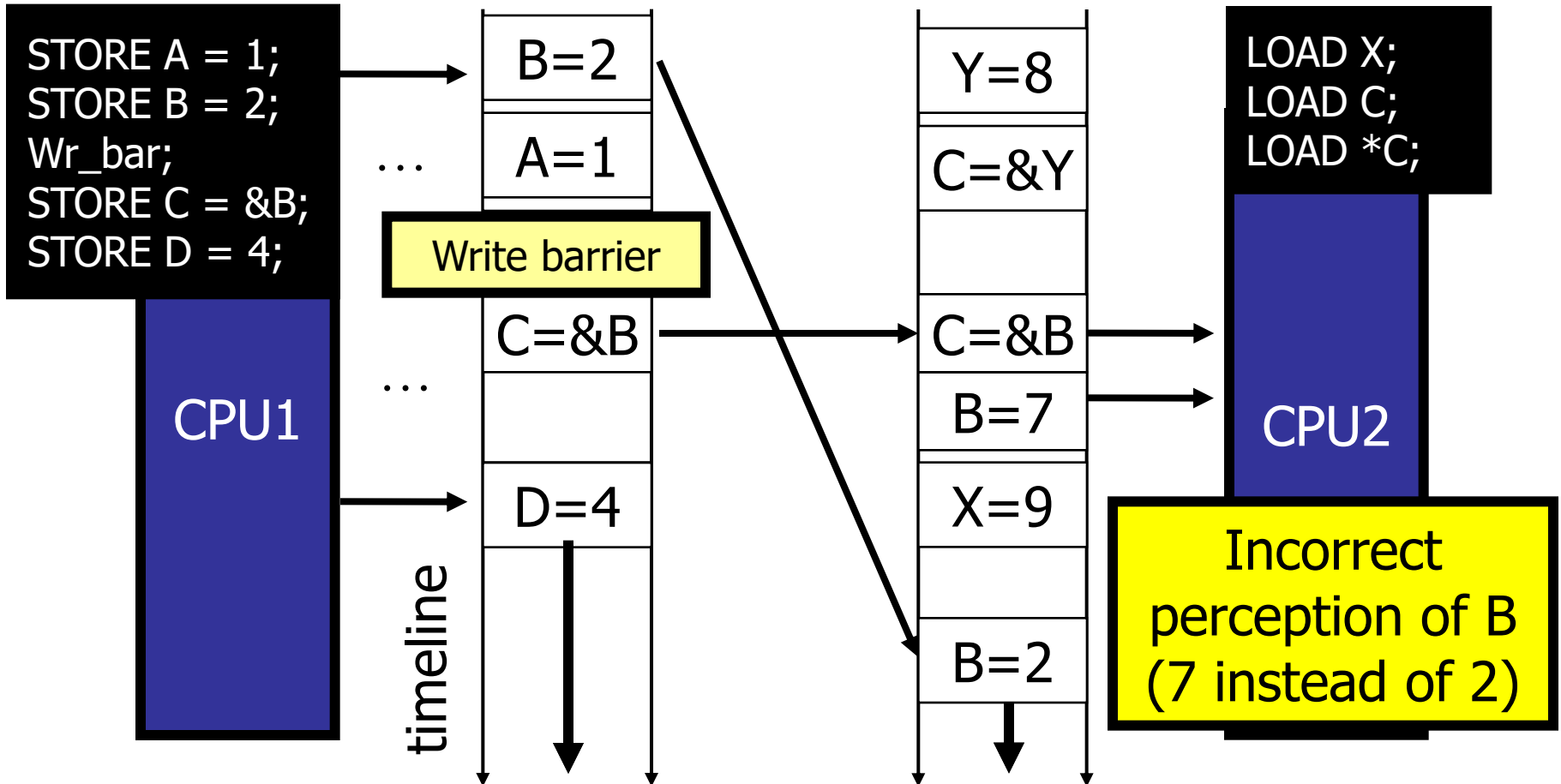
Initially: { B = 7; X = 9; Y = 8; C = &Y }





Sequence of ops without DD_bar

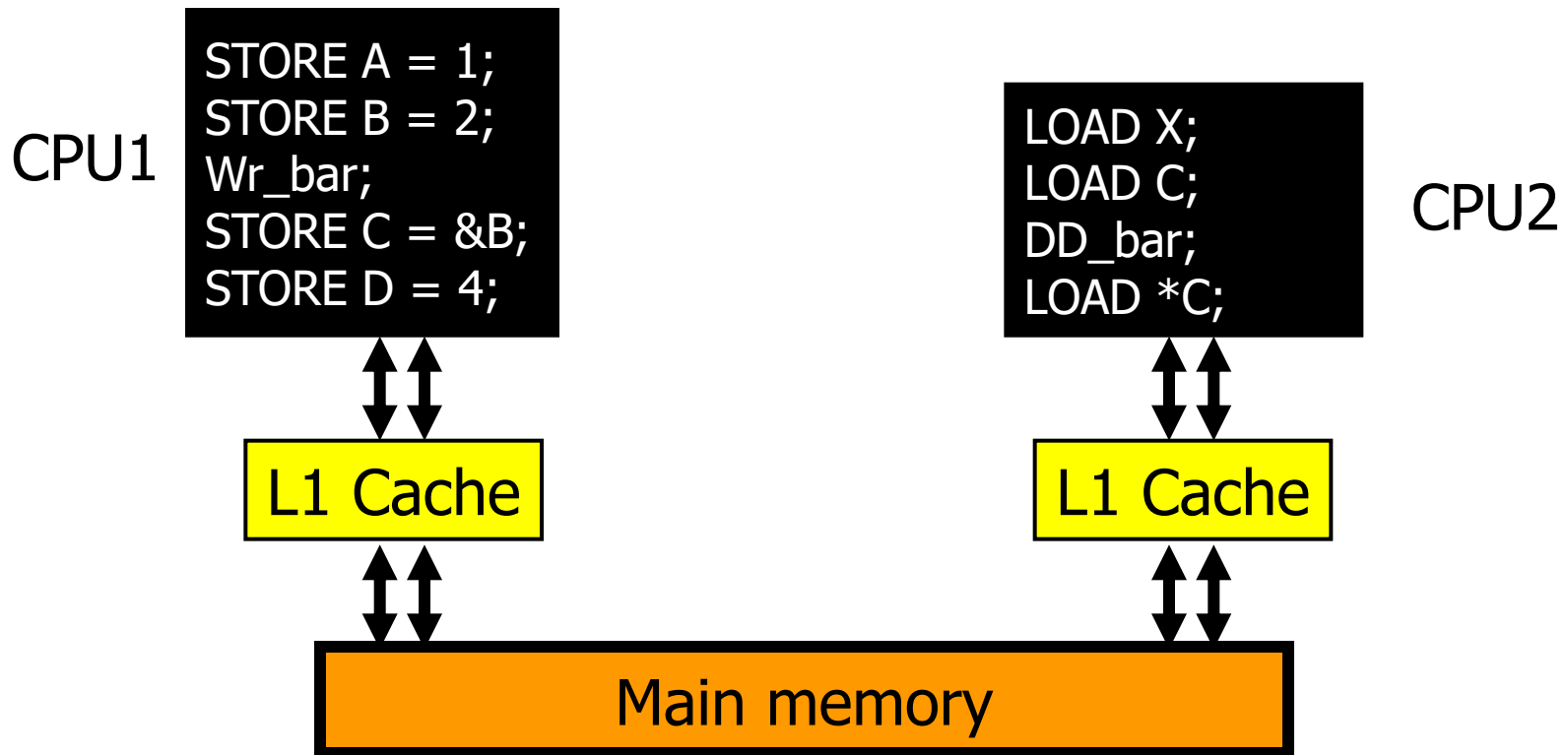
Initially: B = 7; X = 9; Y = 8; C = &Y





Inserting a DD_bar

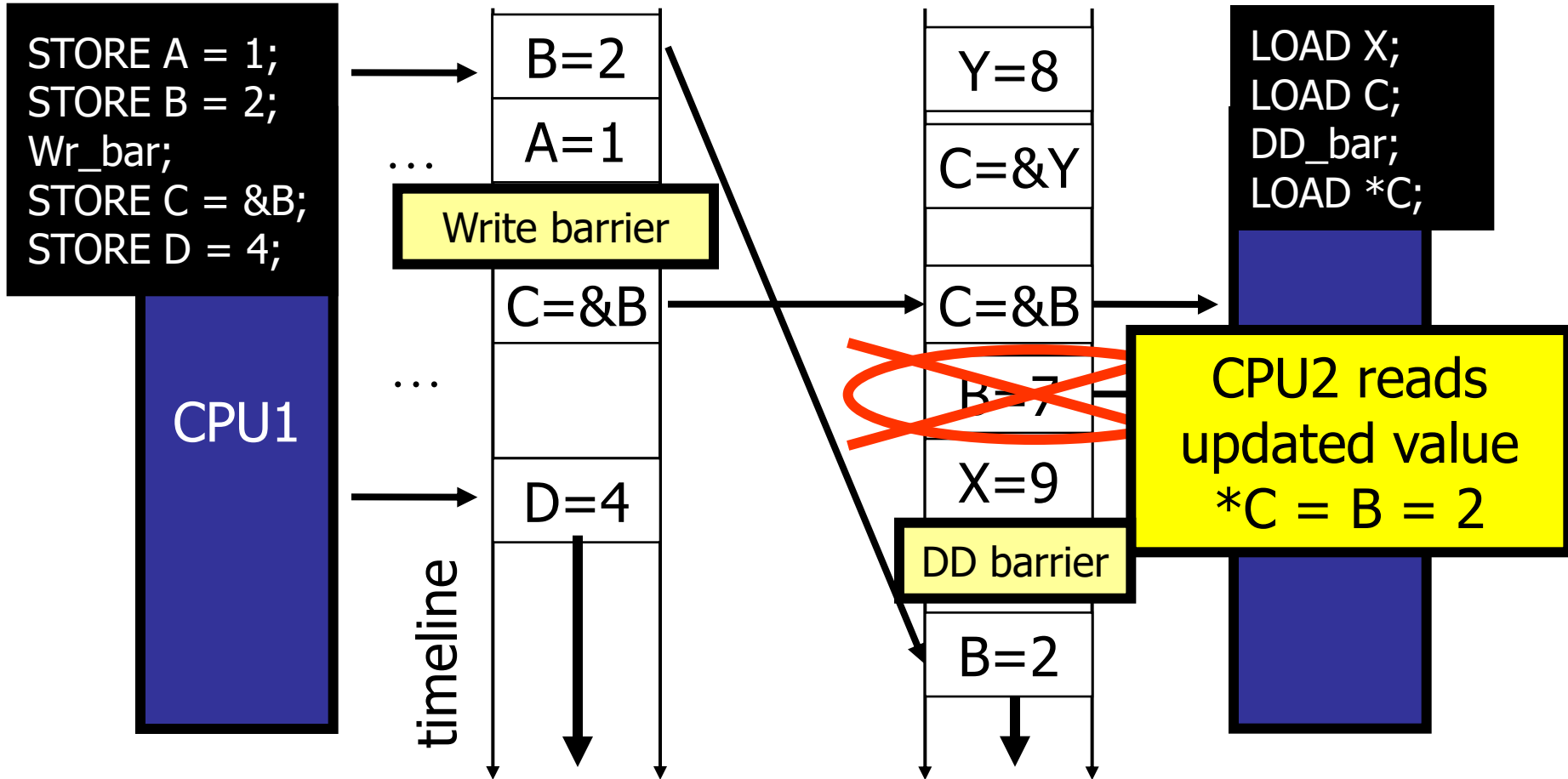
Initially: { B = 7; X = 9; Y = 8; C = &Y }





Sequence of ops with DD_bar

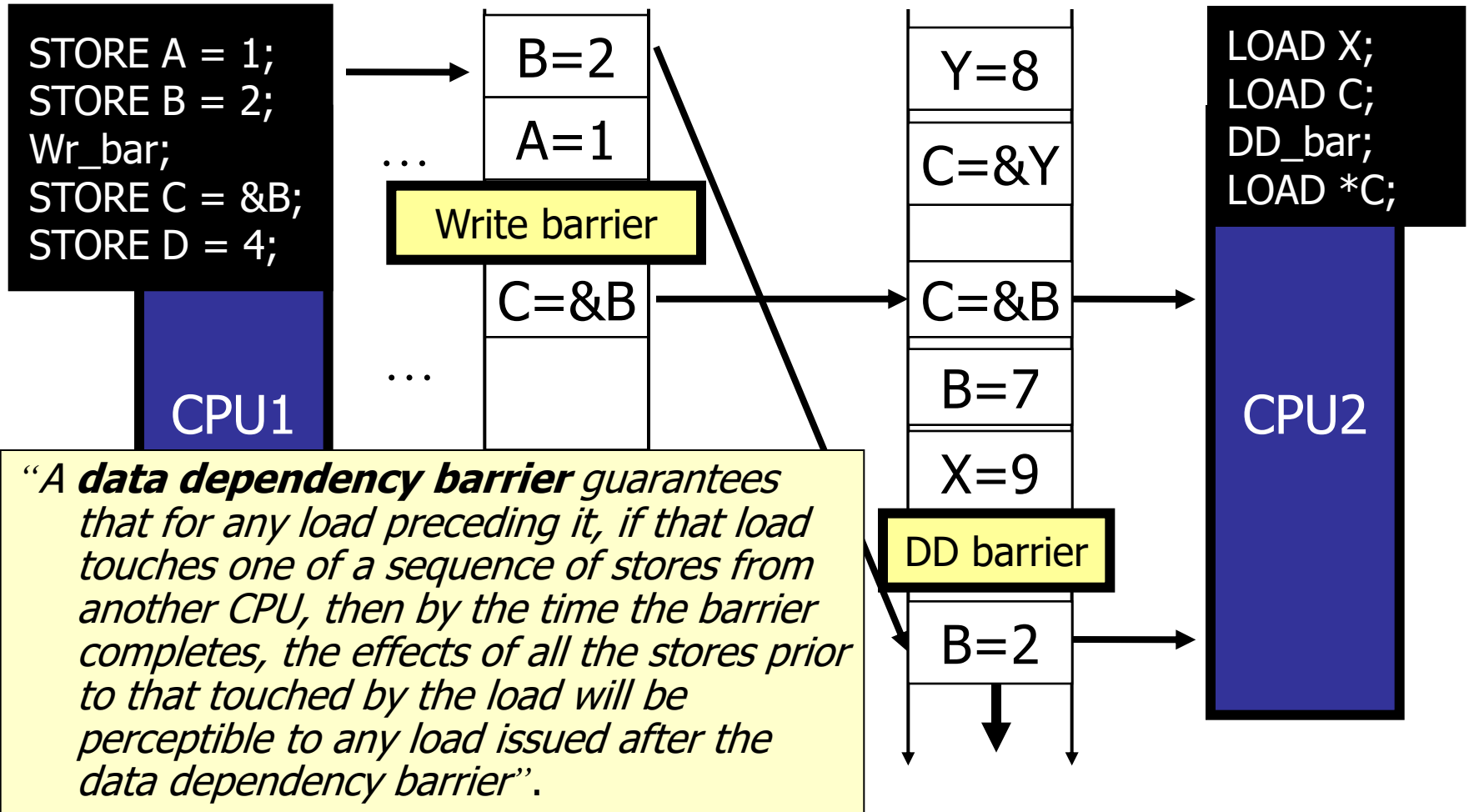
Initially: { B = 7; X = 9; Y = 8; C = &Y }





Sequence of ops with DD_bar

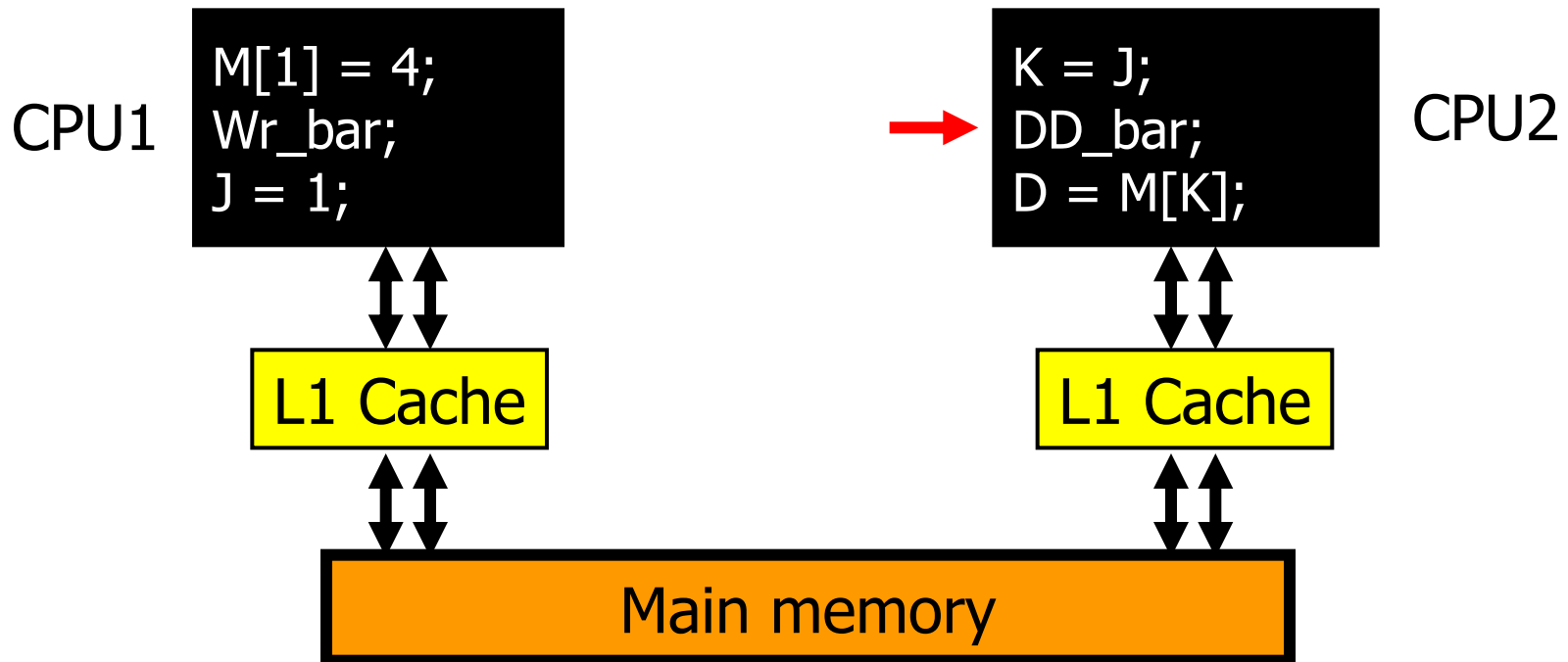
Initially: { B = 7; X = 9; Y = 8; C = &Y }





Example: DD_bar and arrays

Initially: { $M[1] = 2$, $M[3] = 3$, $J = 0$, $K = 3$ }





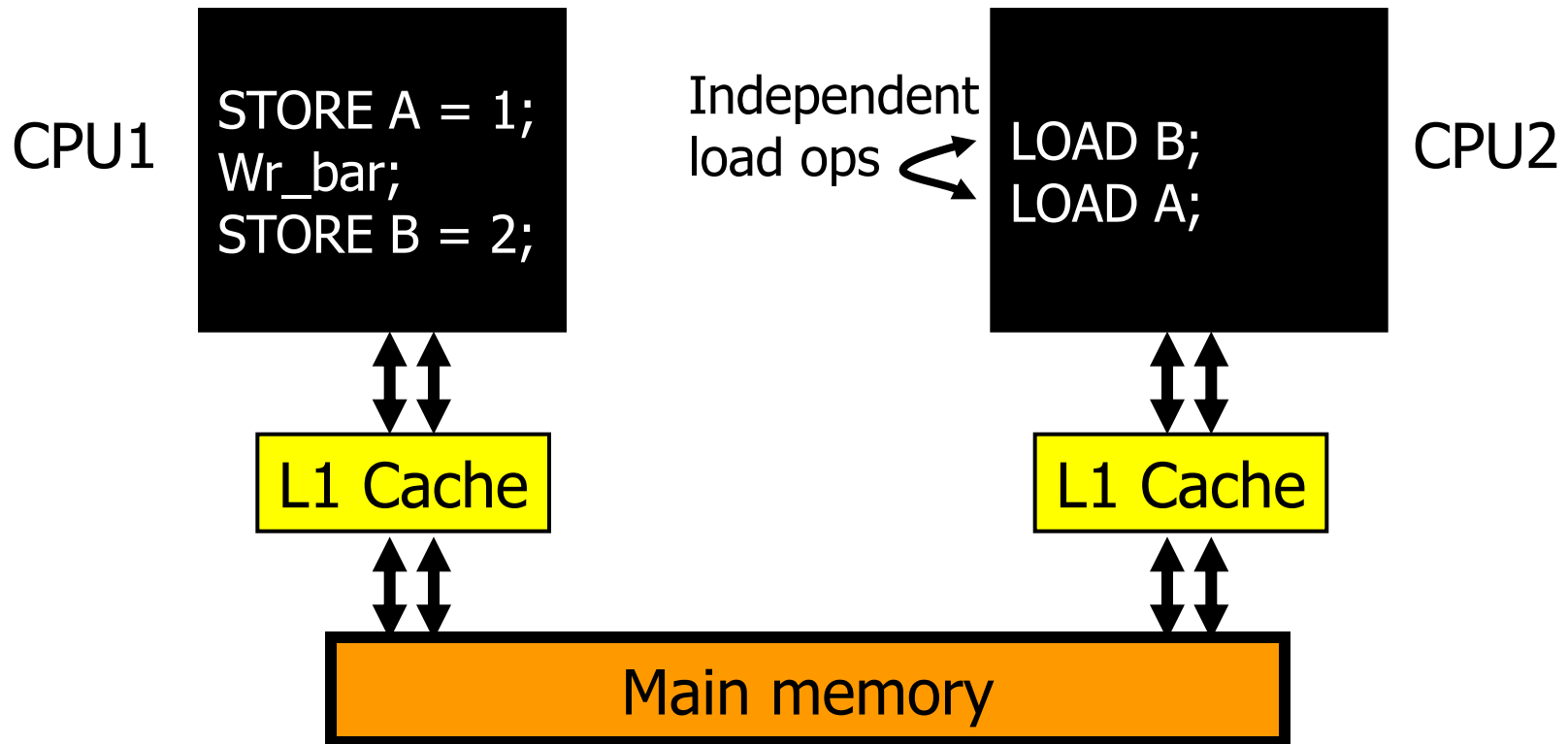
Read (or load) memory barriers

- All LOAD operations before the barrier will appear (to other system components) to happen before all LOAD operations after the barrier
- No effect on store operations
- Stronger than data dependency barrier
 - DD_bar applies only to dependent loads
 - Rd_bar applies to **all** load operations
 - Therefore, a Rd_bar implies a DD_bar



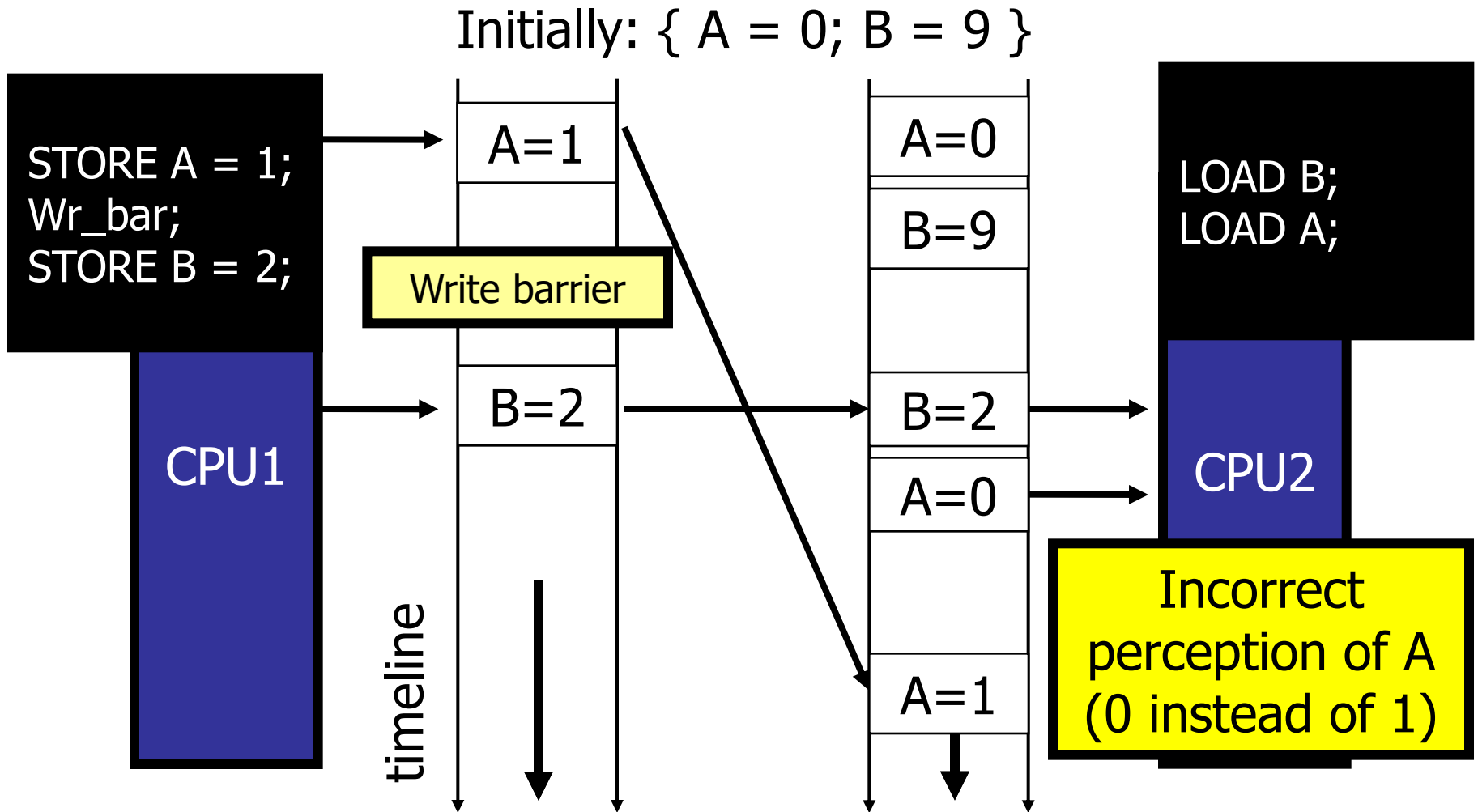
Example (without Rd_bar)

Initially: { A = 0; B = 9 }





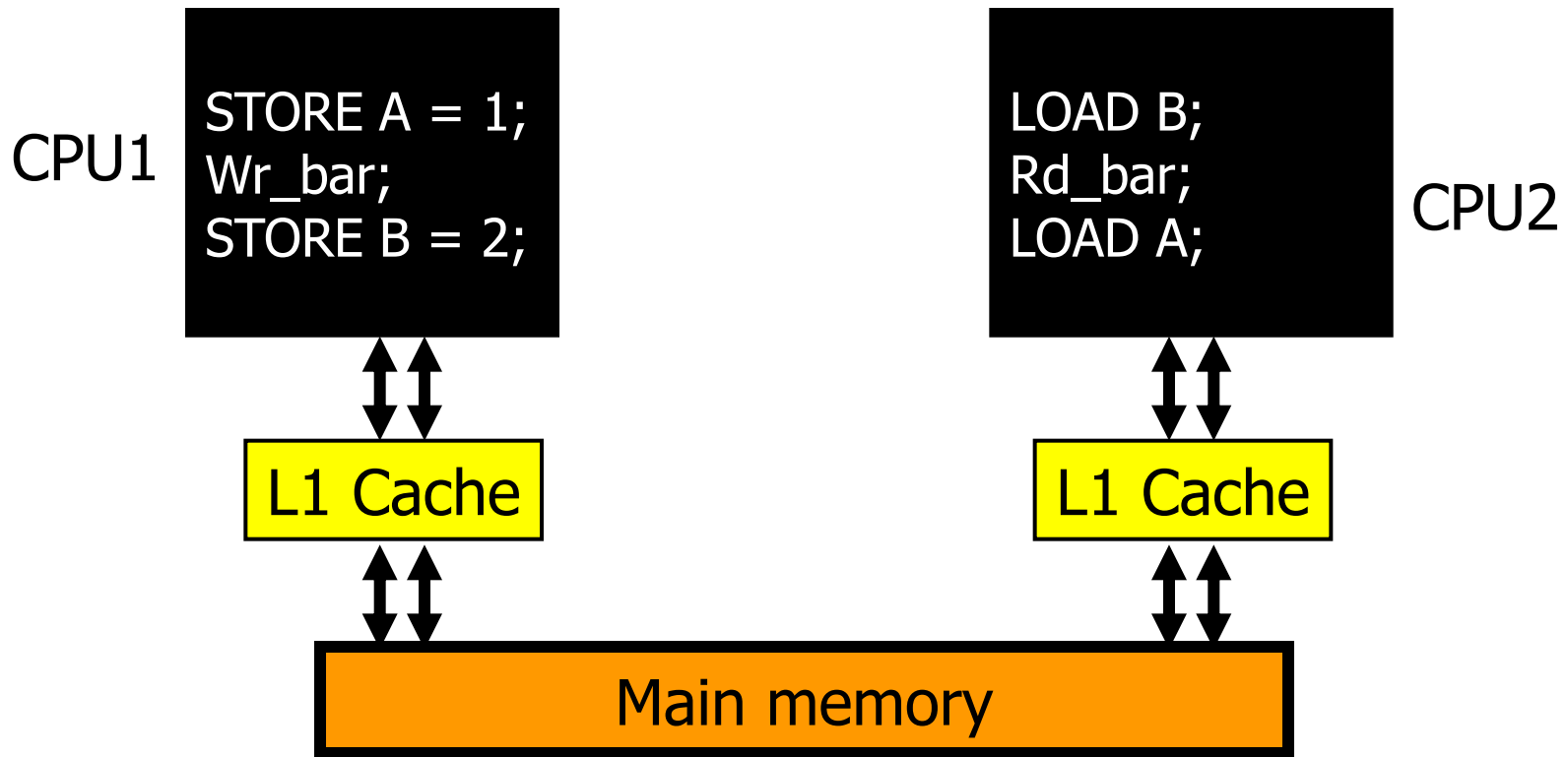
Sequence of ops without Rd_bar





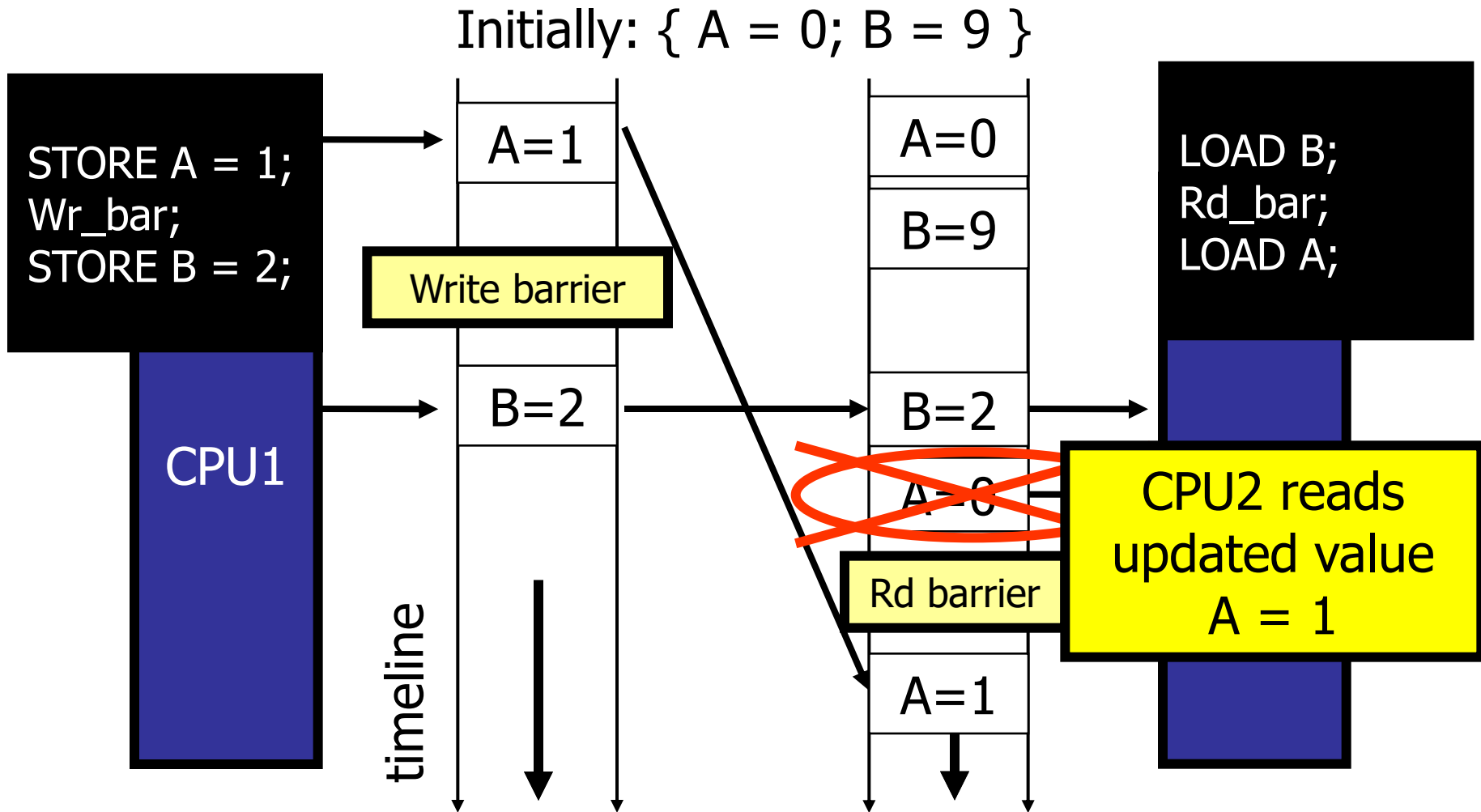
Inserting a Rd_bar

Initially: { A = 0; B = 9 }





Sequence of ops with Rd_bar





Rd_bar vs DD_bar


- A *read barrier* has to be used instead of a *data dependency* barrier when
 - there is no data dependency between the operations involved. E.g.,

```
LOAD B;  
LOAD A;
```

or

- there is a control dependency between the operations involved. E.g.,

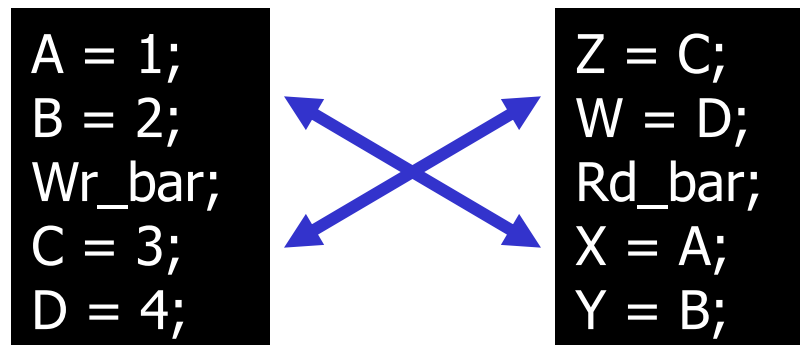
```
Q = &A;  
if (C)  
    Q = &B;  
<BARRIER>  
X = *Q
```

Here we need a Rd_bar! 
A DD_bar is not sufficient since there is a control dependency between “Q=&B” and “X=*Q”



Pairing memory barriers

- A *write barrier* needs a *data dependency barrier* or a *read barrier*, to work properly
- The partial ordering enforced by a `Wr_bar` on a CPU can be perceived by other CPUs only if they use a paired `DD_bar` (or `Rd_bar`)
- Typically the stores before the `Wr_bar` match the loads after the `Rd_bar` (or `DD_bar`) and viceversa





General memory barriers

- All LOAD and STORE operations before the barrier will appear (to other system components) to happen before all LOAD and STORE operations after the barrier
- Combine the functionalities of Wr_bar and Rd_bar
 - Therefore, a Full_bar implies both a Wr_bar and a Rd_bar



Linux kernel barriers

- Compiler barriers
 - `barrier();`
- CPU memory barriers
 - Mandatory barriers: `mb(); wmb(); rmb(); read_barrier_depends();`
 - SMP conditional barriers: `smp_mb(); smp_wmb(); smp_rmb(); smp_read_barrier_depends();`
- MMIO write barriers
 - `mmiowb();`



Implicit kind of barriers

- LOCK and UNLOCK operations
- They are unidirectional barriers that are permeable to read and write accesses only in one way
- Used to delimit Critical Sections of code to which a process need exclusive access



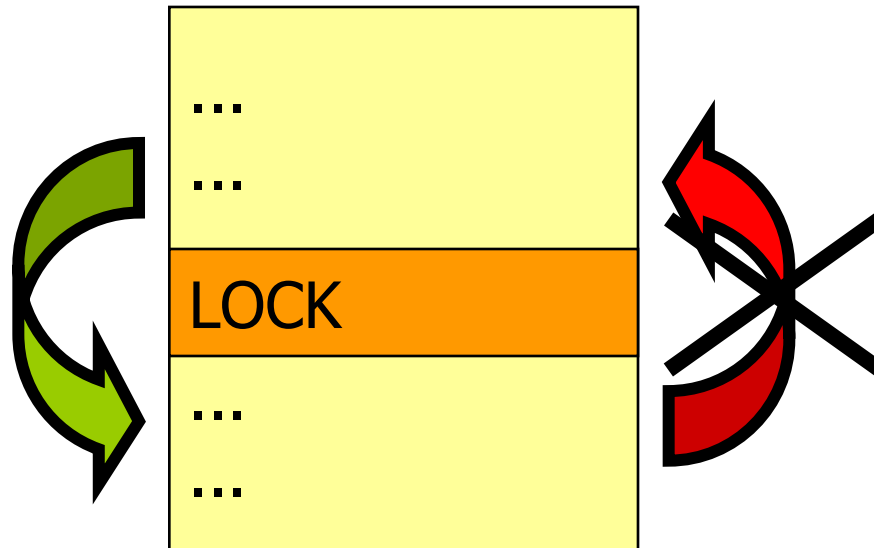
Critical Section (CS)

- A shared resource (data structure or device) that must be exclusively accessed by one thread
- Synchronization mechanism required at Critical Section boundaries
- On uni-processors can be implemented avoiding context switches (e.g., disabling interrupts and preemptions)
- On multi-processors this is no more valid



Lock operations

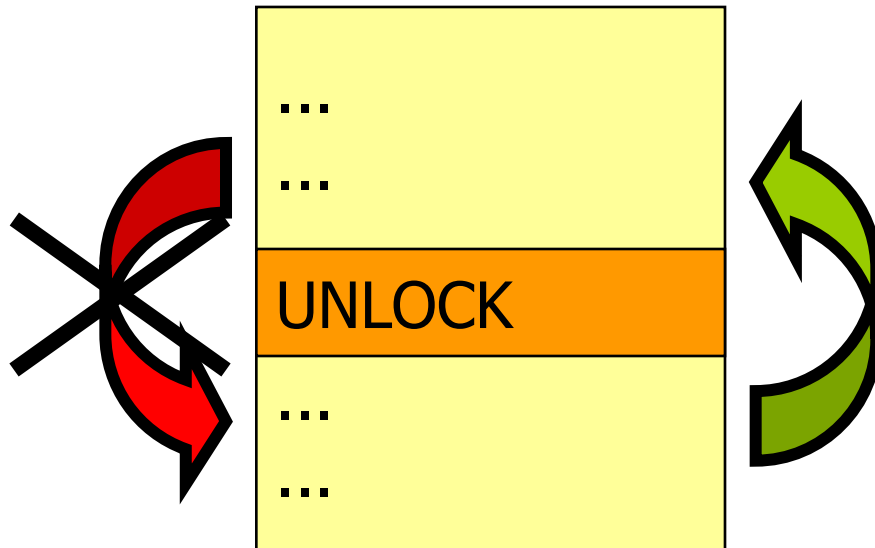
- All LOAD and STORE operations after the *lock* will appear (to other system components) to happen after the *lock*
- Memory operations occurring before the *lock* may appear to happen after it completes





Unlock operations

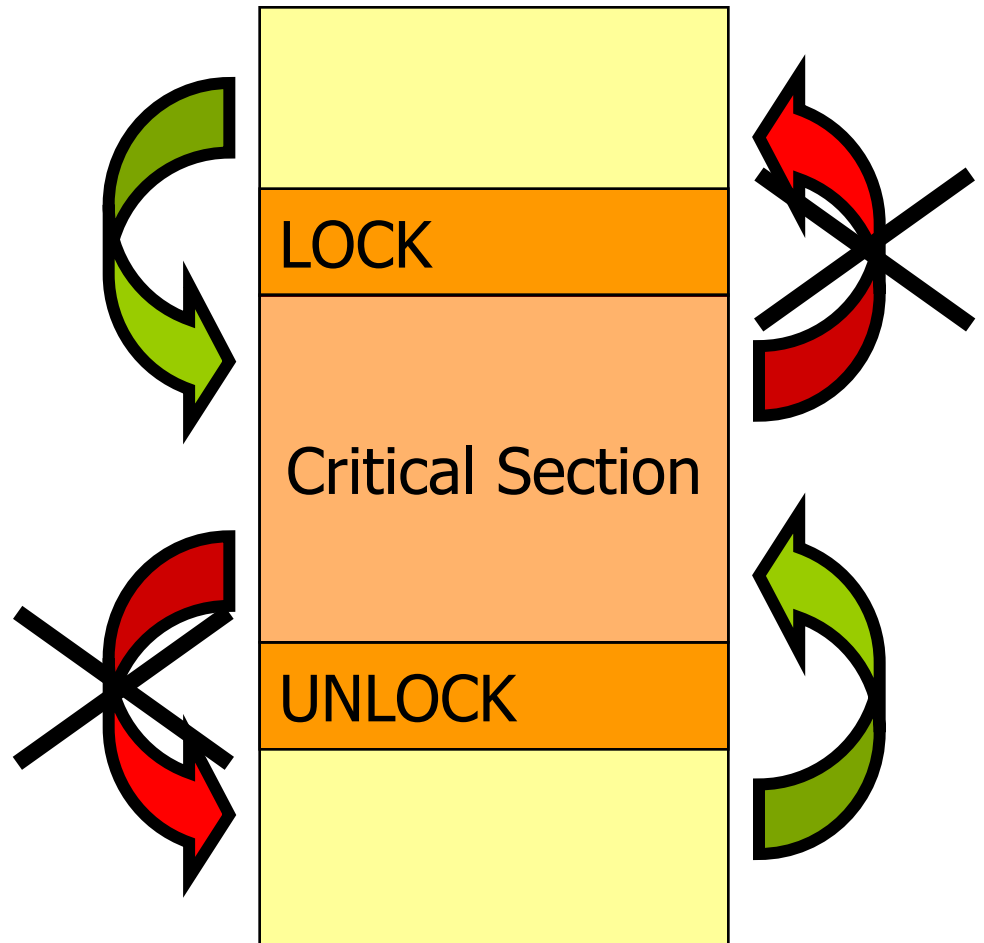
- All LOAD and STORE operations before the *unlock* will appear (to other system components) to happen after the *unlock*
- Memory operations occurring before the *unlock* may appear to happen after it completes





Critical section implementation

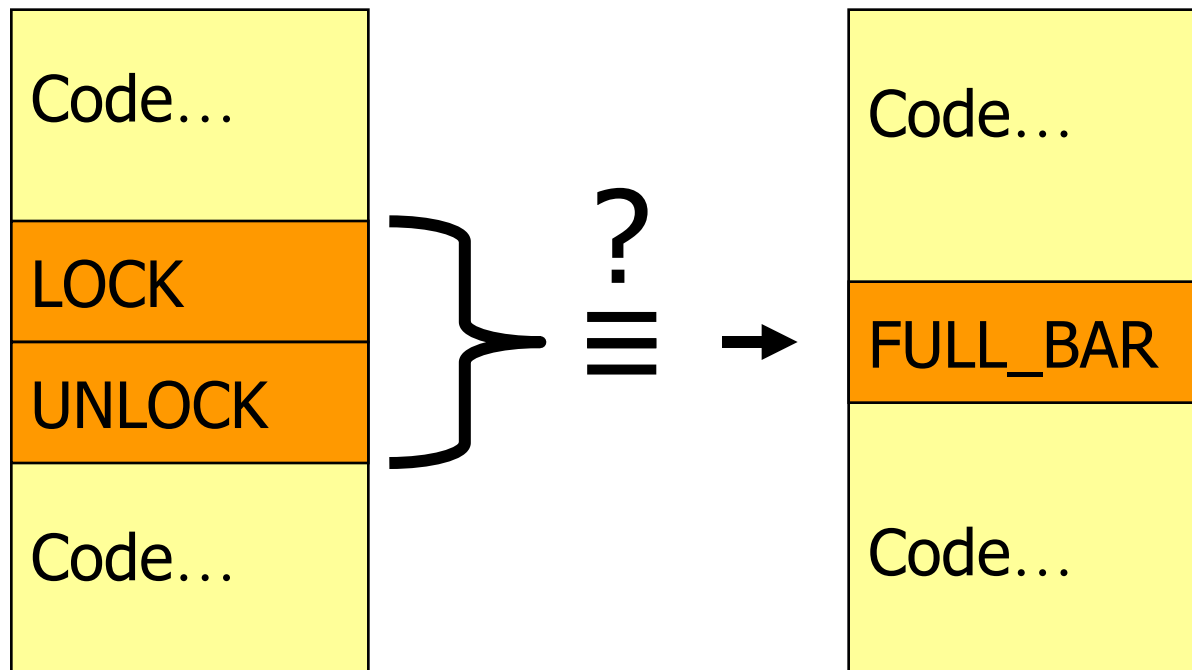
- Lock and Unlock operations are almost always paired
- They delimit Critical Sections
- When lock/unlock are used, no need for explicit memory barriers





Question

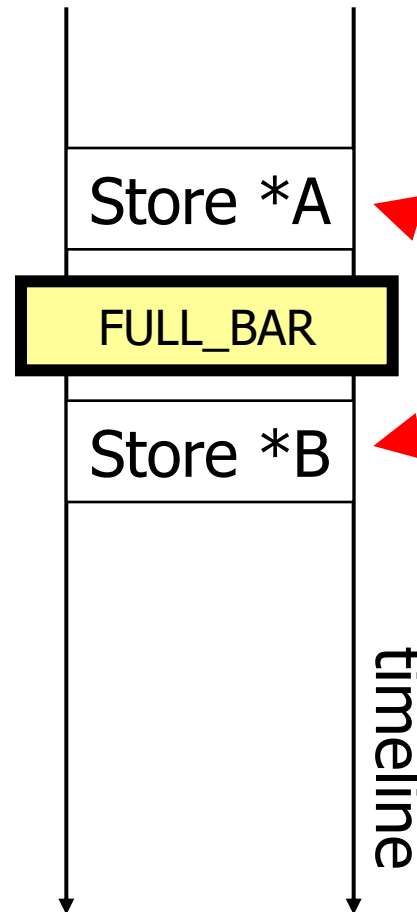
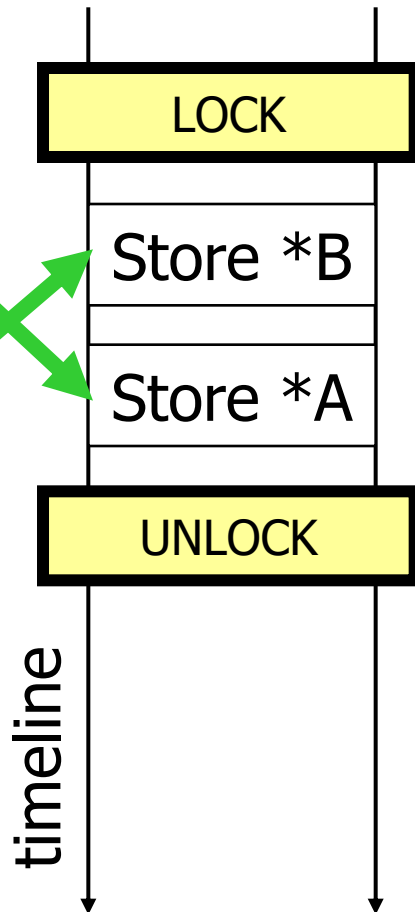
- Is a LOCK followed by an UNLOCK equivalent to a full memory barrier?





Lock/Unlock vs Full_bar

```
*A = a;  
LOCK;  
UNLOCK;  
*B = b;
```

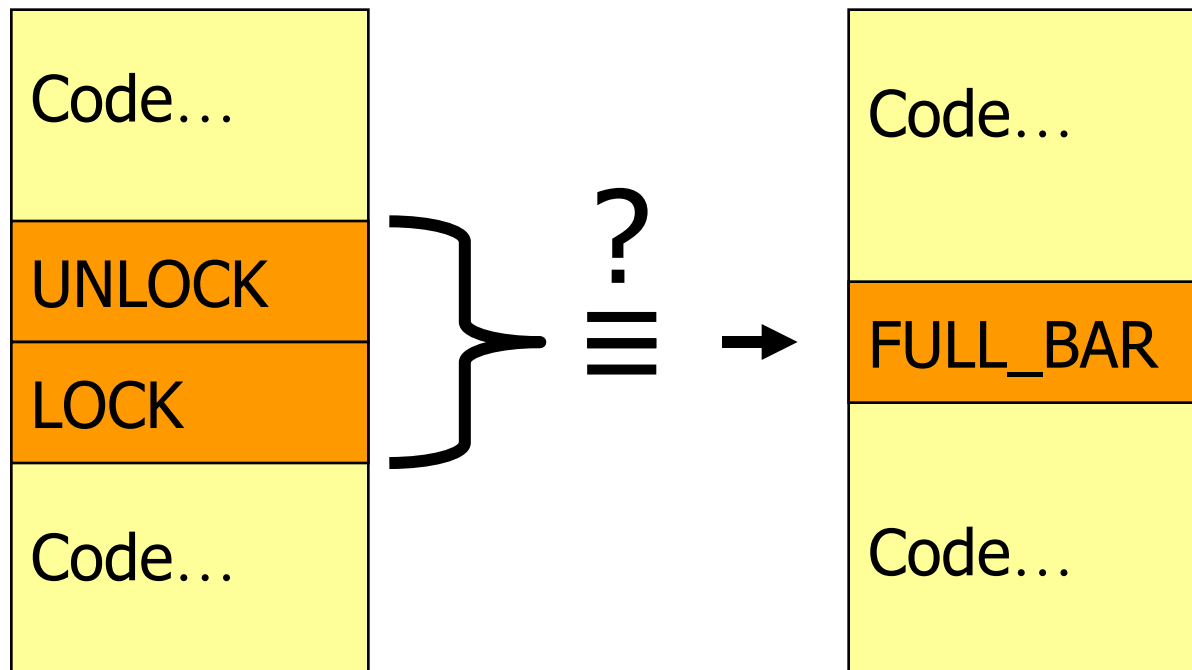


```
*A = a;  
FULL_BAR;  
*B = b;
```



Question

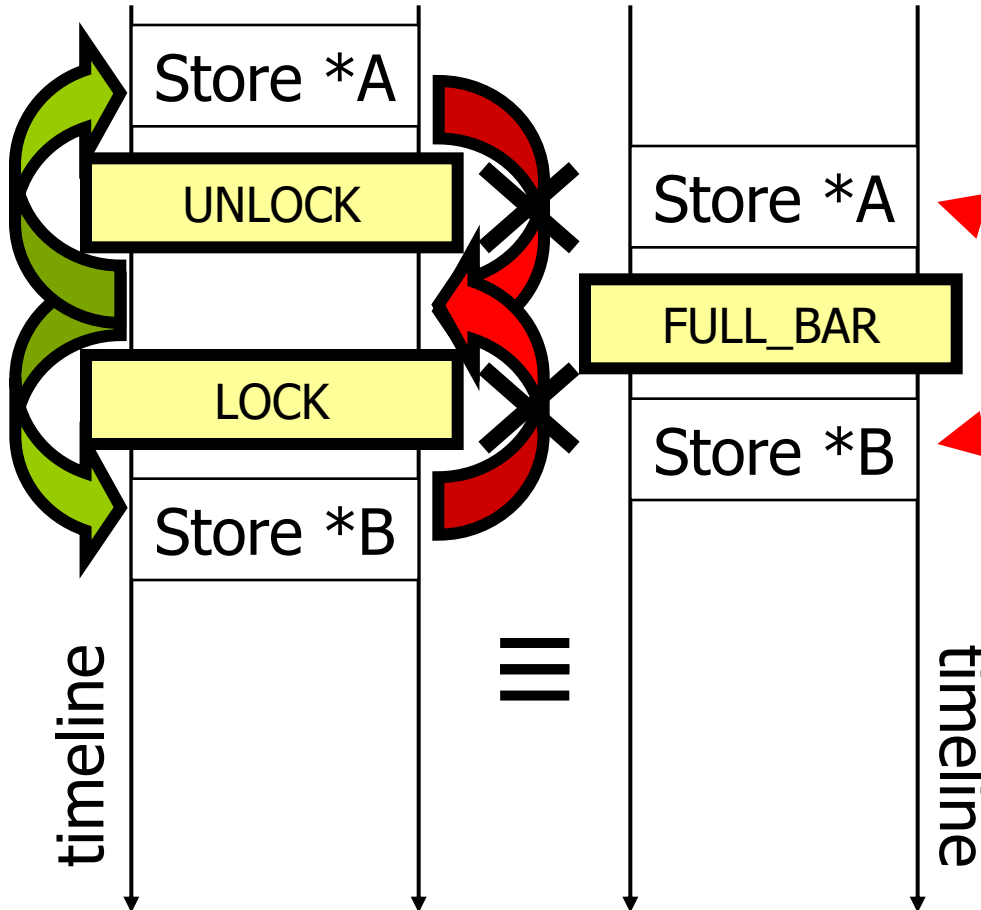
- Is an UNLOCK followed by a LOCK equivalent to a full memory barrier?





Unlock/Lock vs Full_bar

```
*A = a;  
UNLOCK;  
LOCK;  
*B = b;
```



```
*A = a;  
FULL_BAR;  
*B = b;
```



Example 1

CPU1

```
*A = a;  
LOCK M;  
*B = b;  
*C = c;  
UNLOCK M;  
*D = d;
```

CPU2

```
*E = e;  
LOCK Q;  
*F = f;  
*G = g;  
UNLOCK Q;  
*H = h;
```

Different spin locks M and Q

Other CPUs might see, for example:

- *E, LOCK M, LOCK Q, *G, *C, *F, *A, *B, UNLOCK Q, *D, *H, UNLOCK M

But they will never see:

- *B, *C or *D before LOCK M
- *A, *B or *C after UNLOCK M
- *F, *G or *H before LOCK Q
- *E, *F or *G after UNLOCK Q



Example 2

CPU1

CPU2

[1] *A = a;
LOCK M;
*B = b;
*C = c;
[1] UNLOCK M;
*D = d;

*E = e;
LOCK M;
*F = f;
*G = g;
UNLOCK M;
*H = h;

Same spin lock M

Other CPUs might see, for example:

- *E, LOCK M[1], *C, *B, *A, UNLOCK M[1], LOCK M[2], *H, *F, *G, UNLOCK M[2], *D

But, assuming CPU1 gets the lock first, they will never see:

- *B, *C, *D, *F, *G or *H before LOCK M[1]
- *A, *B or *C after UNLOCK M[1]
- *F, *G or *H before LOCK M[2]
- *A, *B, *C, *E, *F or *G after UNLOCK M[2]



Advisory vs mandatory locks

- Advisory lock
 - Each thread cooperates by acquiring the lock before accessing the protected resource
- Mandatory lock
 - Attempting unauthorized access to a locked resource forces an exception



Lock implementation

- For single processors the exclusive access to a shared resource can be implemented disabling interrupts
- For multiprocessors this is not enough!
- Hardware support required for efficient implementation
- Atomic instruction(s) needed: test if lock is free and acquire the lock in a single atomic operation



Read-Modify-Write (RMW)

- RMW instructions atomically do both the following operations:
 - read a memory location
and
 - write a new value into it simultaneously (either a completely new value or some function of the previous value)
- Example: Test-and-set(); Compare-and-swap(); Fetch-and-add(); Dec_and_test(); etc.

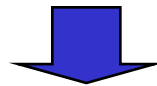


Why atomicity is required?

- Consider two processes executing the following piece of code to acquire the same lock:

```
if (lock == 0)
    lock = process_PID;
```

- If both tasks test the “lock” value at the same time, they will both detect that the lock is free → they will both acquire the lock!



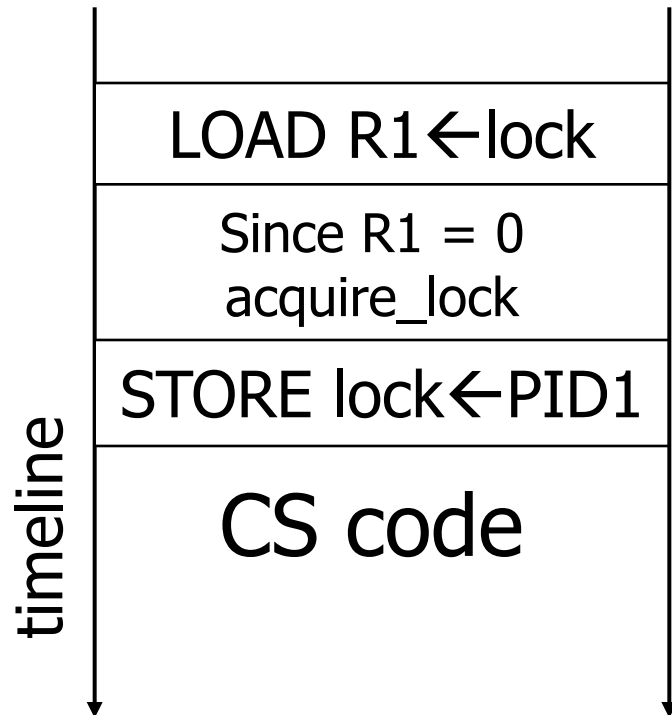
The access to the CS is not exclusive



Without atomic locking

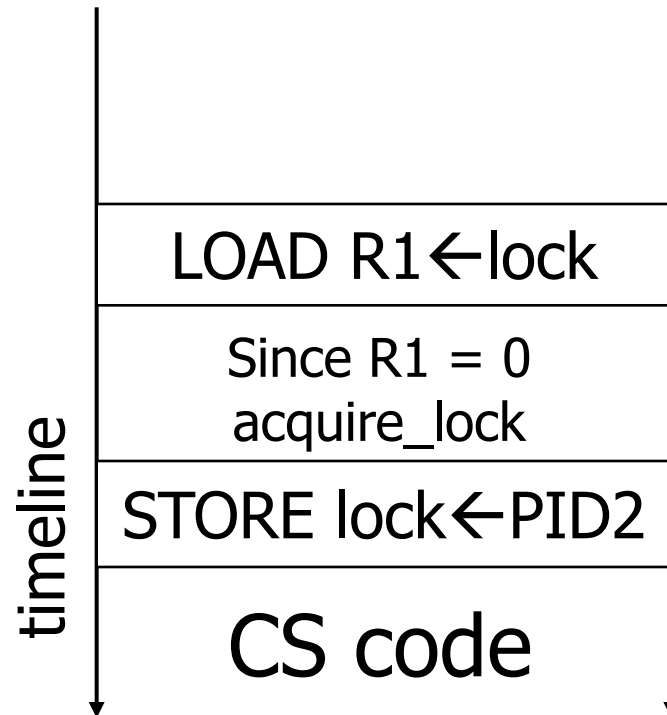
Thread 1

```
if (lock == 0)
    lock = process_PID;
```



Thread 2

```
if (lock == 0)
    lock = process_PID;
```





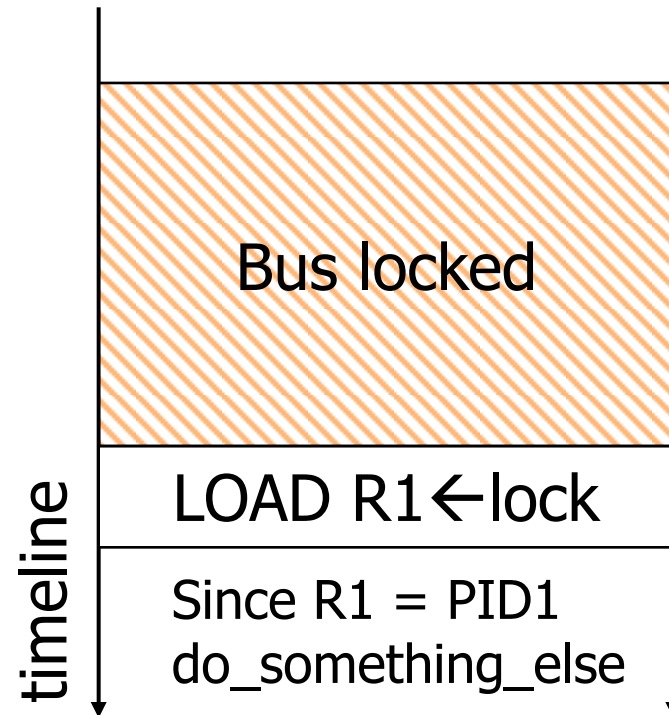
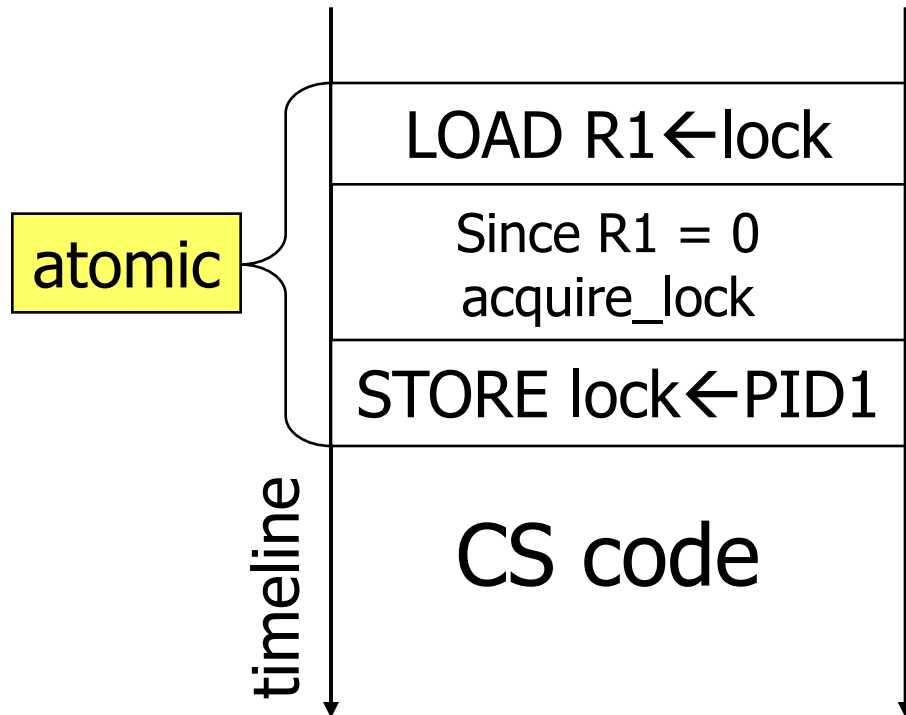
With atomic locking

Thread 1

```
if (test_and_set(lock,0,PID1))  
  <CS code>
```

Thread 2

```
if (test_and_set(lock,0,PID2))  
  <CS code>
```





Atomic instructions in Linux

- Read-Modify-Write instructions:
 - `xchg(); cmpxchg(); atomic_cmpxchg();`
 - `atomic_[inc|dec|add|sub]_return();`
 - `atomic_[inc|dec|sub]_and_test();`
 - `atomic_add_[negative|unless]();`
 - `test_and_[set|clear|change]_bit();`
- Other atomic instructions:
 - `atomic_set(); [set|clear|change]_bit();`
 - `atomic_[inc|dec|add|sub]();`



Atomic operations in Linux

- Are executed without being interrupted by other operations
- Atomic operations that modify some state in memory and return information about the state (old or new) imply an SMP-conditional general memory barrier (`smp_mb()`) on each side of the actual operation
 - E.g., `cmpxchg()`; `atomic_dec_and_test()`; `test_and_set_bit()`; ...



Otherwise...

- Round-robin
 - Strict alternation: each thread can lock the resource at its turn
- Dekker's algorithm
 - Limited to two processes and busy waiting
- Peterson's algorithm
 - Originally formulated for two processes, can be generalized to more than two



Dekker's algorithm

- Uses two flags (f0 and f1) to indicate “intention to enter”, and a turn variable

Initially, f0 = false; f1 = false; turn = 0 (or 1)

Thread1

```
f0 = true
while f1 {
    if turn ≠ 0 {
        f0 = false
        while turn ≠ 0 { }
        f0 = true
    }
}
// ... Critical Section ...
turn = 1
f0 = false
```

Thread2

```
f1 = true
while f0 {
    if turn ≠ 0 {
        f1 = false
        while turn ≠ 1 { }
        f1 = true
    }
}
// ... Critical Section ...
turn = 0
f1 = false
```



Peterson's algorithm

- Uses two flags (f0 and f1) to indicate “intention to enter”, and a turn variable

Initially, f0 = false; f1 = false; turn = 0 (or 1)

Thread1

```
f0 = true
turn = 1
while (f1 && turn==1) {}

// ... Critical Section ...

f0 = false
```

Thread2

```
f1 = true
turn = 0
while (f0 && turn==0) {}

// ... Critical Section ...

f1 = false
```




Considerations

- Dekker's and Peterson's algorithm would need memory barriers when used on processors with instruction reordering
- More efficient solutions are provided using atomic operations



Key design aspects for locks

- Overhead
 - extra resources used just for the implementation of the lock → memory space, time for lock initialization/destruction and acquire/release
- Contention
 - number of threads that can concurrently request the lock → a lock shared by many processes implies a larger blocking probability
- Granularity
 - size of the protected region → coarse vs fine granularity



Lock granularity

- Coarse granularity
 - less overhead, but more lock contention
- Fine granularity
 - larger number of “faster” locks
 - larger system overhead
 - higher risk of deadlocks
- Example: locking a whole table, a row, or a single entry



Risks related to locking

- Blocking time → starvation, lack of fairness
- Locking tasks stalls/blocks/dies/loops
- Error-prone due to crossed dependencies
difficult to detect for larger program sizes
 - Deadlocks, livelocks
 - Priority inversion
- Bugs difficult to reproduce



Linux kernel locking constructs

- Spin locks
- R/W spin locks
- Mutexes
- Semaphores
- R/W semaphores
- RCU

All require atomic instruction support



Spinlock

- A thread waits (“spins”) until the lock is available
- Wasteful if locks are held for a long time
 - On a single processor, many time quanta may be wasted due to spinning on a resource locked by a preempted thread
- Efficient for short locks → avoid process re-scheduling



Spinlock: example

```
lock:
    dd    0

spin_lock:
    mov   eax, 1

loop:
    xchg  eax, [lock]
    test  eax, eax
    jnz   loop
    ret
```

- The lock variable is 1 when locked, 0 when unlocked
- “xchg eax, [lock]” atomically swap the EAX register with the lock variable
- “test eax, eax” sets the zero flag if $EAX = 0$
- If zero flag is set, the lock has been acquired
- Otherwise spin



Spinlock: unlock

```
lock:
    dd    0

spin_unlock:
    mov   eax, 0
    xchg  eax, [lock]
    ret
```

- To release the lock, reset the lock variable to zero



Spinlock and interrupts

- When an interrupt handler can access the lock as well, the following situation might occur

Process on CPU1

```
spin_lock(&lock);  
...
```

← Interrupt comes in

```
...  
spin_lock(&lock);
```

DEADLOCK!!!



Waits for the release of the lock which will never happen



Solution: disable interrupt

- Disable interrupts

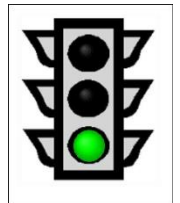
Process on CPU1

```
spin_lock(&lock);  
cli();  
...  
sti();  
spin_unlock(&lock)
```

Interrupt arrival

handling
deferred

```
spin_lock(&lock);  
...  
spin_unlock(&lock);
```





Spinlock and interrupts

- Only local interrupts need to be disabled → not necessary to disable interrupts on other CPUs

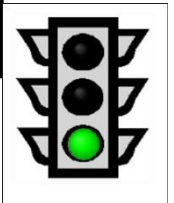
Process on **CPU1**

```
spin_lock(&lock);  
...  
<continue exec>  
spin_unlock(&lock)
```

Interrupt arrival
on **CPU2**



```
spin_lock(&lock);  
...  
spin_unlock(&lock);
```



- In Linux
 - xxx_lock_irqsave(...)
 - xxx_unlock_irqrestore(...)



Reader-Writer spinlocks

- Allow multiple readers to be in the same critical section at once
- Do not allow more than one single writer at a time
- Usually, there is no need to read a shared resource with an exclusive access to it...
- ... as long as no other process modifies it!



R/W spin locks in Linux: read

- Readers acquire/release the lock with
 - `read_lock(&xxx_lock, flags);`
 - `read_unlock(&xxx_lock, flags);`
- If no process is holding a write lock on the same resource, the read lock can be acquired
- The only result will be an increment/decrement in the counter of the processes currently holding the lock



R/W spin locks in Linux: write

- Writer acquires/releases the lock with
 - `write_lock(&xxx_lock, flags);`
 - `write_unlock(&xxx_lock, flags);`
- It can acquire the lock only when there is no process holding a (read or write) lock → check the `lock_counter`
- When a process is holding a write lock, no other process can acquire the lock



Considerations

- R/W spin locks are faster than normal spin locks, allowing more readers to temporarily access the resource
- If we know that interrupts will only need read locks, possible to use
 - *read_lock(&lock)* for read accesses
 - and
 - *write_lock_irqsave(&lock, flags)* for write accesses



Busy-wait

- Busy-wait is an anti-pattern associated to “spinning” before entering a critical section
- Waste of CPU cycles
- May delay subsequent requests, reducing the spinning frequency
- Better to **block** the process on events like lock acquisitions, timers, I/O availability, or signals → the blocked thread is put in sleeping state and other threads are executed



Locking strategies

- Instead of “spinning”, a more efficient method is using semaphores with blocking
- Thread must acquire a semaphore before entering a CS
- Block the execution of the thread requesting the lock until it is allowed to acquire the semaphore
- Other threads can execute other code/critical sections



Semaphores

- A protected variable to restrict the access to shared resources
- Implemented by a counter for a set of available resource → counting semaphore
- Binary semaphores are called mutexes
- Prevents race conditions but not deadlocks



Non-blocking synchronization

- Overcomes the disadvantages of using locks
- Lock-free
 - a thread cannot lock up: every step it takes brings progress to the system
- Wait-free
 - a thread can complete any operation in a finite number of steps, regardless of other threads
- All wait-free algorithms are lock-free (but not viceversa)
- No semaphores nor mutexes



Lock-free algorithms

- Still need atomic instructions like Test-and-set, Compare-and-swap, etc.
- Example:

```
CAS(addr, old, new)
{
  atomic
    if (*addr == old)
      then {*addr = new ;
           return true}
    else return false
  endatomic
}
```



Example: bank account

- Each thread represents a teller trying to make a deposit onto the same account



- Need to synchronize simultaneous deposits to the account



Write conflict

Thread 1

```
deposit(money)
{
    A = read_account();
    A = A + money;
    write_account(A);
}
```

Thread 2

```
deposit(money)
{
    A = read_account();
    A = A + money;
    write_account(A);
}
```

- If both threads simultaneously read the account → a transaction is lost!



Locking solution

- Each teller has to acquire a lock before doing a deposit:

```
deposit(money)
{
    lock(account);
    A = read_account();
    A = A + money;
    write_account(A);
    unlock(account);
}
```

- Implies locking overhead



Lock-free solution

- Use Compare_And_Swap(address,old,new):

```
deposit(money)
{
    do{
        A = read_account();
        B = A + money;
    }
    while (! CAS(account, A, B) )
}
```

- Lock-free but not wait-free: other tellers may keep writing new values → failing teller tries again indefinitely



Reducing contention

- To reduce traffic on the bus due to repeated reads, failing threads may wait some time before trying another update
 - constant delay
 - incremental delay
 - random delay



ABA problem

- Arises when a thread reads a location a second time to detect if anything has changed from the first read
- But another thread could have modified and restored the value at that location, e.g.
 - Thread 1 reads A from a shared memory location
 - Thread 2 modifies the same location from A to B and back to A
 - Thread 1 reads again the value A and assumes nothing changes → ERROR!!!



ABA problem with lock-free

- Push and Pop function of a stack list

```
Obj* Pop()
{
    while(1) {
        Obj* ret = top;
        if (!ret) return NULL;
        Obj* next = ret->next;
        if (CAS(&top, ret, next))
            return ret;
    }
}
```

```
void Push(Obj* obj)
{
    while(1) {
        Obj* next = top;
        obj->next = next;
        if (CAS(&top, next, obj))
            return;
    }
}
```



ABA problem with CAS

- Suppose initially the stack contains

top → A → B → C

- Suppose Thread1 is preempted during the Pop() by Thread2

```
Thread1
{
    Pop();
}
```

```
Thread2
{
    Pop(); → top → B → C
    Pop(); → top → C
    Push(A); → top → A → C
}
```



ABA problem with CAS

- Thread1's Pop operation:

```
Obj* Pop()
```

```
{
```

```
  while(1) {
```

```
    Obj* ret = top;
```

```
    if (!ret) return NULL;
```

```
    Obj* next = ret->next;
```

```
    //--- PREEMPTED BY THREAD2---
```

```
    if (CAS(&top, ret, next))  
        return ret;
```

```
  }
```

```
}
```

ret → A

next → B

top → A → C

CAS succeeds but leaves
the stack in wrong state:

top → ???!

(B was popped
by Thread2)



Transactional memory

- Lock-free approach that is able to deal with the ABA problem
- Allows a group of load and store instructions to execute in an atomic way
 - Load-Link/Store-Conditional
 - Software Transactional Memory (STM)



Load-Link/Store-Conditional

- LL works like a normal load from a memory location
- A subsequent SC to the same memory location will store a new value *only if no updates have occurred to that location since the load-link*, otherwise it fails
- SC fails even if the value read by LL has since been updated and then restored (ABA problem) → LL/SC is stronger than read/CAS



Software Transactional Memory (STM)

- Optimistic behavior: threads complete modifications to shared memory regardless of other threads
- They record every read and write in a log
- Each reader verifies that other threads have not concurrently made changes to memory that it accessed in the past
 - *commit* permanent changes if validation is successful
 - otherwise *abort*, undoing all its prior changes



STM: considerations

- The conflict control is placed on the reader instead of the writer
- Increased concurrency: no need to wait for access to a resource
- Increased overhead in case of failing
- Good performance in practice → conflicts arise rarely in practice



Read-Copy Update (RCU)

- Split updates into “*removal*” and “*reclamation*” phases
- The removal phase
 - removes references to data items within a data structure (possibly by replacing them with references to new versions of these data items)
 - can run concurrently with readers which will see either the old or the new version of the data structure rather than a partially updated reference



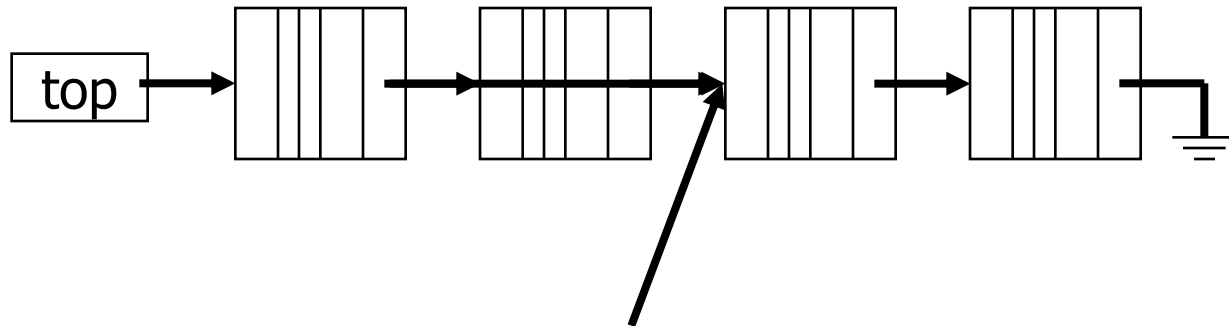
RCU

- The reclamation phase
 - frees the data items removed from the data structure during the removal phase
 - must not start until readers no longer hold references to those data items
 - reclaiming can be done either by blocking or by registering a callback
- No need to consider readers starting after the removal phase → they are unable to gain a reference to the removed data items



RCU typical sequence

1. Remove pointers to a data structure → later readers cannot gain a reference to it
2. Wait for all previous readers to complete their RCU read-side critical sections
3. Reclaim the data structure (e.g., using *kfree* in the Linux kernel)





Advantages of RCU

- Wait-free reads
- RCU readers use much lighter-weight synchronization → low overhead
- Reclamation phase may be done by entirely different thread → e.g., Linux directory entry cache



Where does the name come from?

- Read-Copy Update
- A thread wishing to update a linked structure in place does the following
 - creates a new structure, **copying** the data from the old structure into the new one
 - modifies the new, copied, structure
 - **updates** the pointer to refer to the new structure
 - sleeps until there are no **readers** left
- Therefore, an RCU protected structure is *Read* concurrently with a thread *Copying* in order to do an *Update*



Linux RCU implementation

- RCU API
 - `rcu_read_lock()`
 - `rcu_read_unlock()`
 - `synchronize_rcu()` / `call_rcu()`
 - `rcu_assign_pointer()`
 - `rcu_dereference()`



Conclusion

- Sharing data on multicore platforms requires attention
- Possible to choose among different constructs (spinlocks, semaphores, R/W locks, RCU, ...)
- Using proper mechanism it is possible to exploit the power of multicore
- Start thinking parallel!