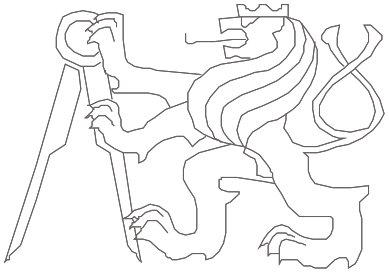
Advanced Computer Architectures

Multiprocessor systems and memory consistency problems



Czech Technical University in Prague, Faculty of Electrical Engineering
Slides authors: Michal Štepanovský, update Pavel Píša

Terminology of the lecture topic

- Memory operation execution rules,
  - Memory coherence – last lecture
    - Rules for access to individual locations in memory
  - Memory consistency – today lecture
    - Rules for mutual order of execution and visibility of memory operations
- Ensuring sequential consistency,
- Weaker memory consistency models
  - Consistency achieved by synchronization, that is by special synchronization instructions.

Memory coherence definition (in common sense)

We say that a multiprocessor memory system is coherent if the results of any execution of a program are such that for each location, it is possible to construct a hypothetical serial order of all operations (reads and writes) to the location that is consistent with the results of the execution and in which:

1) Memory operations to a given memory location for each process are performed in the order in which they were initiated by the process.
2) The values returned by each read operation are the values of the most recent write operation in a given memory location with respect to the serial order.

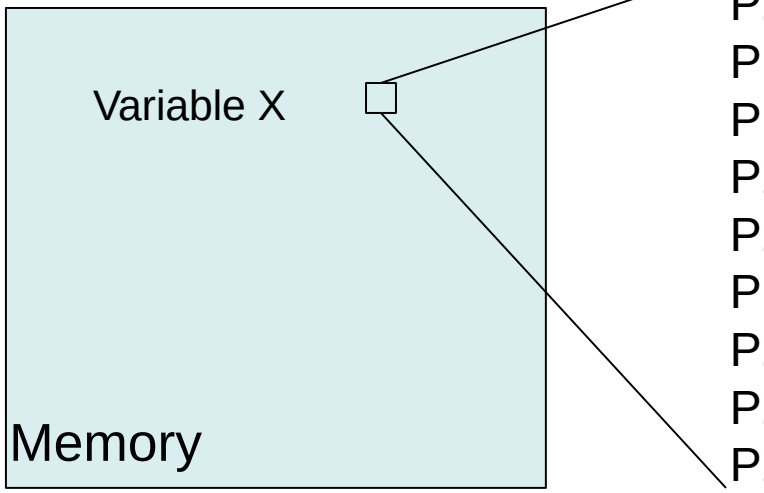
Coherence

Proces P1:

```
X=0;
if(X ==0) {
  y=fun();
  X = 1;
}
```

Proces P2:

```
X=0;
while(X ==0)
  { ; }
X = 2;
```



P2: X=0;
P1: X=0;
P1: read(X)
P2: read(X)
P2: read(X)
P1: X=1;
P2: read(X)
P2: read(X)
P2: X=2;

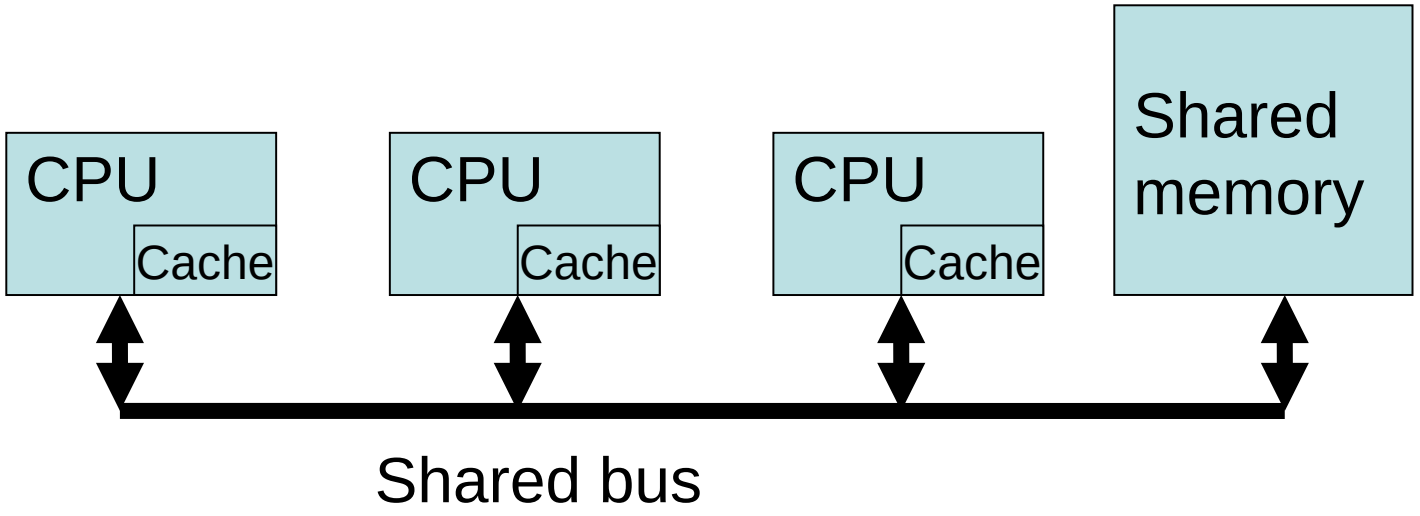
P2: read(X)
P2: X=0;
P1: X=0;
P1: read(X)
P2: read(X)
P1: X=1;
P2: read(X)
P2: read(X)
P2: X=2;

At the time when P2 reads X==1, is it ensured that function fun() called by process P1 is executed with all side effects including global memory?

Consistency

- **Consistency** (when compared to coherence) specifies the order in which the individual processes execute their memory operations and or how is this order viewed by other processes.
- Sequential order of all memory operations to all locations is considered.
- **Coherence** focuses **only** on hypothetical sequential order to **individual memory locations** but guarantees neither order nor visibility of accesses to different locations.
- Consistency defines what is expected behavior of shared memory regarding all reads and writes

Example of program execution on multiprocessor system



Variables initialization seen by both: `x=0, y=0`

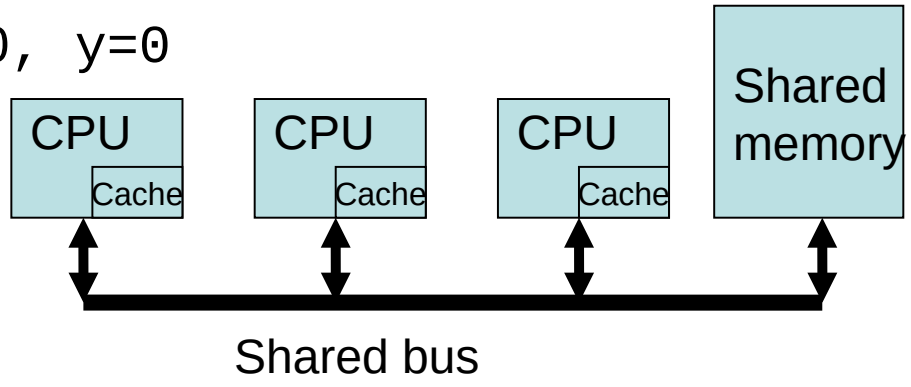
```
P1:          P2:
x = 1;       while(y==0) {;}
y = 1;       print(x);
```

It is expected that *print(x)* writes *1* to output.

Example of program execution on multiprocessor system

Variables initialization seen by both: `x=0, y=0`

```
P1:         P2:
x = 1;      while(y==0){;}
y = 1;      print(x);
```



Possible scenario of execution:

1. Processor P2 does not find y in cache and initiates a request to read from memory. The bus has to be obtained through arbitration first.
2. Processor P2 starts reading of x speculatively – line „print(x)“. It finds x value (0) in its cache. Speculation is conditionalized by variable y==1.
3. Processor P1 acquires the bus and executes write to variable x: „x=1“. The corresponding cacheline is marked as M (MESI protocol) and invalidated in P2.
4. Processor P1 acquires the bus and writes y=1 into memory.
5. Processor P2 acquires the bus and reads y value. This confirms „correctness“/condition of speculation and speculative instructions are completed.
6. Processor P2 outputs 0.

Is coherence enough to ensure expected program behavior?

- Variable *y* indicates that variable *x* has been changed.
- But memory coherence provides no guarantee for mutual execution order of memory operations (read, write) by P1 and P2 and the order in which the writes to *x* and *y* (different variables) are visible to P2.
- Coherence ensures only that the new values of *x* and *y* are finally visible to P2 but provides no guarantee about the order in which these values are obtained.

- **That is why P2 can print the old value of *x* (which is 0) even on computer with coherent memory system.**

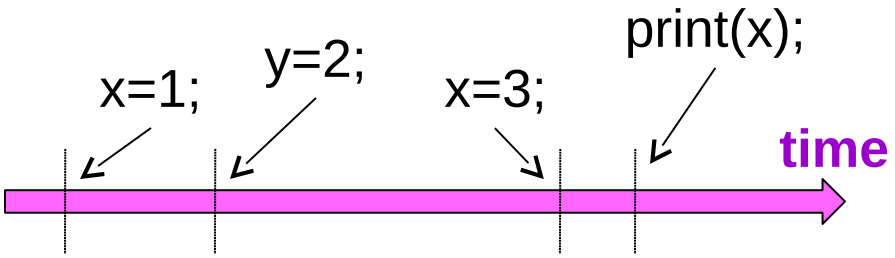
Coherence of cache memories is necessary (but not enough) for ensuring data (memory) consistency in a multiprocessor system.

- Coherence – which value is returned by read
- Consistency – when is the written value returned by read

Strict consistency

- Single-processor system:

```
x = 1;
y = 2;
x = 3;
print(x);
```



(**Each read from address *x* returns the last value written to address *x*.**)

- For multi-processor system:
  - Existence of global precise time in all nodes and immediate modification propagation
  - Non-realistic (absurd) requirement

Sequential consistency

- Definition (Lamport, 1979): “Computer 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.

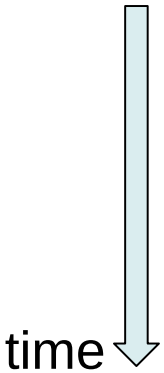
- Sequential consistency is weaker model than strict consistency but it is implementable…
- If the processes are running on different processors, arbitrary interleaving of instructions execution is allowed, but all processes recognize memory changes in exactly the same order (including the writing one). Modifications are not propagated immediately, only their order is guaranteed (the consequence does not precede the cause).

Sequential consistency

Let the variables be initialized with a=0, b=0, c=0.

P1:	***P2:***	***P3:***
`a=1;`	`b=1;`	`c=1;`
`print(b,c);`	`print(a,c);`	`print(a,b);`

It can come as follows:




```
a=1;
b=1;
c=1;
print(b,c);
print(a,c);
print(a,b);
```

Output: 111111

```
a=1;
print(b,c);
b=1;
print(a,c);
c=1;
print(a,b);
```

Output: 001011

etc.

There exist 6! different permutations of instruction interleaving but not all fulfill the sequential consistency requirement

6! / 8 = 90

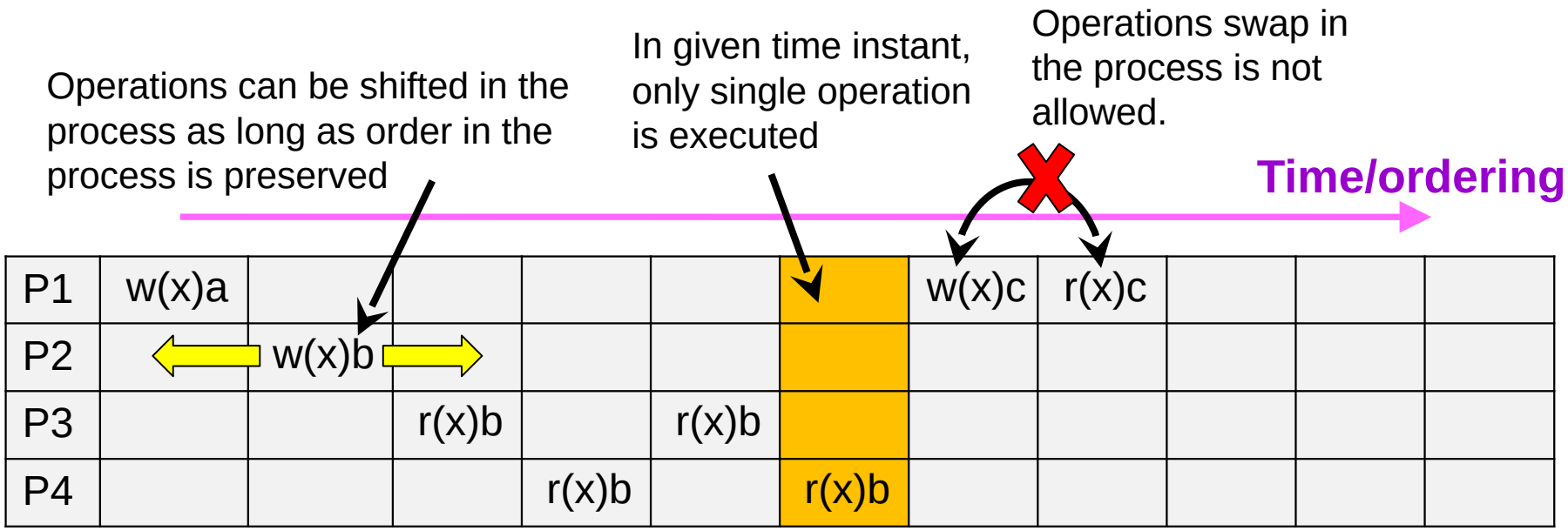
Sequential consistency

Legend:

- Write value „**a**“ to address „**x**“: **w(x)a**
- Read from address „**x**“. Return value is „**a**“: **r(x)a**

Example – consider 4 processors (processes) which are executed in parallel:

- P1: w(x)a, w(x)c, r(x)?
- P2: w(x)b
- P3: r(x)?, r(x)?
- P4: r(x)?, r(x)?



Sufficient conditions to ensure SC

I. Each processor P(i) issues memory operations in program order.

II. Before issuing next memory operation, processor P(i) waits until the last memory operation issued by P(i) completes (i.e., performs w.r.t. all the other processors).

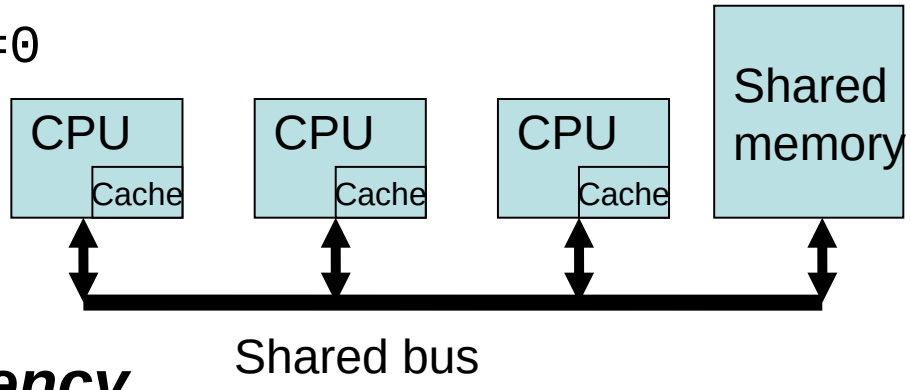
III. When Processor P(i) issues a Read operation, it does not issue another memory operation before the issued read operation is finished, and before the Write operation, whose value will be returned by the Read, is finished (w.r.t. all the other processors) $\rightarrow$ write atomicity.

- Not only the HW is required to keep sequential order, but even the compiler is not allowed to alternate the order of memory operations. But their reordering and elimination is usual/necessary for program optimization on single-processor system.

Analysis of execution of program on SC system

Let the variables be initialized: `x=0, y=0`

```
P1:          P2:
x = 1;       while(y==0){;}
y = 1;       print(x);
```



Assume sequential consistency

Condition violation III.

One of possible scenarios:

1. Processor P2 does not find y in cache and initiates a request to read from memory. The bus has to be obtained through arbitration first.
2. ~~Processor P2 starts reading of x speculatively – line „print(x)“. It finds x value (0) in its cache. Speculation is donditionalized by variable y==1.~~
3. Processor P1 acquires the bus and executes write to variable x: „x=1“. The corresponding cacheline is marked as M (MESI protocol) and invalidated in P2.
4. Processor P1 acquires the bus and writes y=1 into memory. This invalidates y in P2 cache.
5. Processor P2 acquires the bus and reads y value. ~~This confirms „correctness“ of speculation and speculative instructions are completed.~~
6. Processor P2 outputs ~~0~~ 1. It has to read „x“ again/there because read in step number 3 is forbidden or aborted.

Sequential consistency and speculation

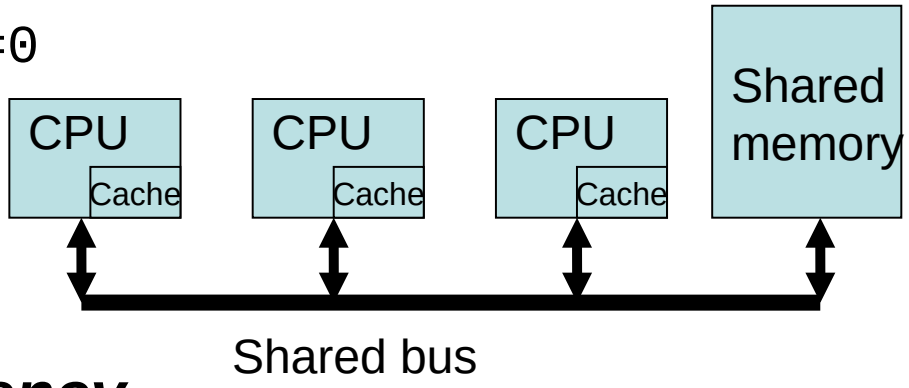
- As shown in the example, forbidding speculation (as well as all read-ahead, reordering of memory operations, etc.) solves the problem.

- Another solution is to isolate the processes as long as no variable sharing emerges – the absence of coherence activities indicates that the processor can reorder memory operations and enable speculation.
- But it is still necessary to keep/propagate the order of memory references regarding cache misses and snooping.
- The solution:
  - Speculative execution is allowed
  - All addresses relating to speculation (or reordering) have to be remembered until the instruction completes
  - If some of these addresses collides with coherence activities, then the whole speculative execution branch is abandoned.

Analysis of execution of program on such system

Let the variables be initialized: `x=0, y=0`

```
P1:          P2:
x = 1;       while(y==0){;}
y = 1;       print(x);
```



Assume sequential consistency

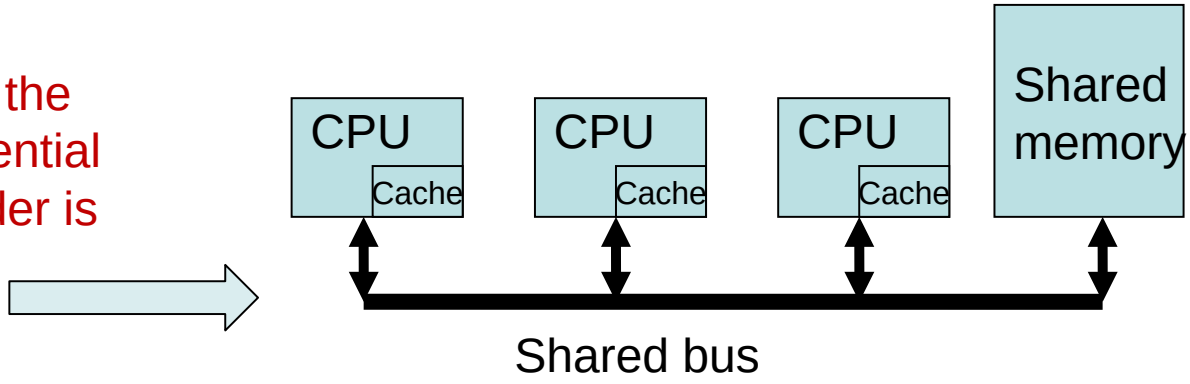
One of possible scenarios:

1. Processor P2 does not find y in cache and initiates a request to read from memory. The bus has to be obtained through arbitration first.
2. Processor P2 starts reading of x speculatively – line „print(x)“. It finds x value (0) in its cache. Speculation is conditionalized by variable y==1.
3. Processor P1 acquires the bus and executes write to variable x: „x=1“. The corresponding cacheline is marked as M (MESI protocol) and invalidated in P2. This collides with address remembered for step 2 speculation. It is abandoned.
4. Processor P1 acquires the bus and writes y=1 into memory.
5. Processor P2 acquires the bus and reads the value of y.
6. Processor P2 acquires the bus, requests the value of x, P1 cache changes state M→S, simultaneously send value of x to memory and P2 which changes state I→S. P2 outputs 1.

Ensuring consistence for SMP system with shared memory

- Definition (Lamport, 1979): “Computer 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.

The shared bus is exactly the place where “some” sequential instruction interleaving/order is ensured ⇒ serialization



If processors (program) fulfill the sequential consistency conditions and the parallel system uses a shared bus, **then the model of sequential consistency is achieved**. Bus arbitration (acquire of time slot) in processor decides the memory operations order – the order can be perturbed for each collision occurrence, but the consistency conditions are preserved.

Consistency – synchronization – example

Problem:
Consider two processors P1 and P2 and a shared variable A.
P1: A = A+1; P2: A = A +2;

As long as addition is atomic, the final value of A is A+3. **However:**

```
P1:     load R1, A      P2:     load R1, A
        addi R1,R1,1            addi R1,R1,2
        store R1,A              store R1,A
```

One of possible execution order results in value A+1:

```
P1:     load R1, A
                        P2:     load R1, A
                                addi R1,R1,2
                                store R1,A
        addi R1,R1,1
        store R1,A
```

This instruction interleaving **fulfills** sequential consistency model, but leads to **„unexpected“ result.**

Consistency – synchronization – example

Problem:
Consider two processors P1 and P2 and a shared variable A.
P1: A = A+1; P2: A = A +2;

Solution:

1. **SW approach.** The code sequence incrementing A needs to be „protected“ against interaction -> **mutual exclusion, critical section**.
   - Mutual exclusion in sequential consistency memory model can be realized with the use of atomic operations Read and Write.
   - **Dekker's algorithm** – the first known correct solution – it guarantees mutual exclusion without the risk of being stuck in a deadlock, and resource allocation.

Peterson's algorithm: initial value of wants_to_enter = { false, false}

```
P1: wants_to_enter[0] = true;
    turn = 1;
    while(wants_to_enter[1] && turn==1)
       ; // busy waiting
    // critical section
    A=A+1;
    // end of critical section
    wants_to_enter[0] = false;
```

```
P2: wants_to_enter[1] = true;
    turn = 0;
    while(wants_to_enter[0] && turn==0)
       ; // busy waiting
    // critical section
    A=A+2;
    // end of critical section
    wants_to_enter[1] = false;
```

Consistency – synchronization – example

Problem:
Consider two processors P1 and P2 and a shared variable A.
P1: A = A+1; P2: A = A +2;

Solution:

2. **SW+HW approach. The code sequence incrementing A needs to be „protected“ again interaction -> mutual exclusion, critical section.**
   SW-only approach is too complicated. We want to implement the code as:

```
while(!acquire(lock)) { waiting algorithm/schedule }
computation with shared data
release(lock)
```

   Because multiple processes may attempt to acquire the lock at the same time, the process of acquiring a lock has to be atomic.
   - Waiting algorithm: ***busy waiting*** or ***blocking waiting***. Busy waiting – continual attempts to acquire the lock – no schedule, deadlock w.r.t. schedule on given processor, blocking waiting – the process enters sleep state, releases the processor (schedule), and is waken up when the lock is released. Combination of both techniques is possible.

Consistency – synchronization

A simple way to realize a lock (spinlock) is to use a shared memory atomic variable, which can signal one of two states - 0 (lock is free) or 1 (lock is acquired by some process). Lock acquisition then means checking that the variable value is 0 and setting it to 1. This operation has to be **atomic** (i.e., no other memory operation to the given location is allowed to occur between the related read and write)!

This requires specific instruction in ISA which:

Reads, modifies and writes (RWM) the value into memory without interference.

***test-and-set* –** all modern processors support such an operation in their ISA, or provide primitives which allow to build such a construct (ll, sc); This operation is a fundamental atomic operation. It writes 1 (set) to memory and returns the previous value of the variable.

- Generalization of *test-and-set* is exchange-and-swap and compare-and-swap
- example: compare-and-exchange in implemented in x86 ISA by instruction: CMPXCHG with LOCK prefix

Consistency – synchronization – CAS discussion

CMPXCHG or CAS

```
bool CAS(unsigned *ptr, unsigned expected, unsigned new)
{
    unsigned tmp = *ptr;
    if (tmp == expected)
        *ptr = new;
    return tmp == expected; // some implementations returns read tmp
}
```

```
#include <stdint.h> // example on x86 with assembly in C
uint32_t cmpxchg_u32(volatile uint32_t *ptr,
                     uint32_t old_val, uint32_t new_val)
{
   uint32_t ret;
   asm volatile("lock; cmpxchgl %2,%1"
        : "=a" (ret), "+m" (*ptr)
        : "r" (new_val), "0" (old_val)
        : "memory");
    return ret == old_val;
}
```

Load-Reserve (Load-Linked) & Store-Conditional on RISC-V

- lr.[wd].{,aq,rl,aqrl} rd, (rs1) – Load-Reserve (Load-Linked)
  R[rd] = M[R[rs1]], reservation on M[R[rs1]]
- sc.[wd].{,aq,rl,aqrl} rd, rs2, (rs1) – Store-Conditional
  if reserved, M[R[rs1]] = R[rs2], R[rd] = 0; else R[rd] = 1

```
# a0 holds address of memory location
# a1 holds expected value
# a2 holds desired value
# a0 holds return value, 0 if successful, !0 otherwise
cas:
    lr.w t0, (a0)          # Load original value.
    bne t0, a1, fail       # Doesn't match, so fail.
    sc.w t0, a2, (a0)      # Try to update.
    bnez t0, cas           # Retry if store-conditional failed.
    li a0, 1               # Set return to success.
    jr ra                  # Return.
fail:
    li a0, 0               # Set return to failure.
    jr ra                  # Return.
```

Consistency – spinlock synchronization

```
while(!acquire(lock)){ ; }
operations with shared data
release(lock)
```

If ***test-and-set*** is used, then the above code fragment can be implemented as:

```
loop: test-and-set R2, lock  // test lock, fetch old value to R2 and set lock=1
      bnz R2, loop           // if R2 is not 0,  jump to loop, repeat acquire attempt
      load R1, A
      addi R1, R1, 1
      store R1, A
      store #0, lock         // release the lock by writing 0.
```

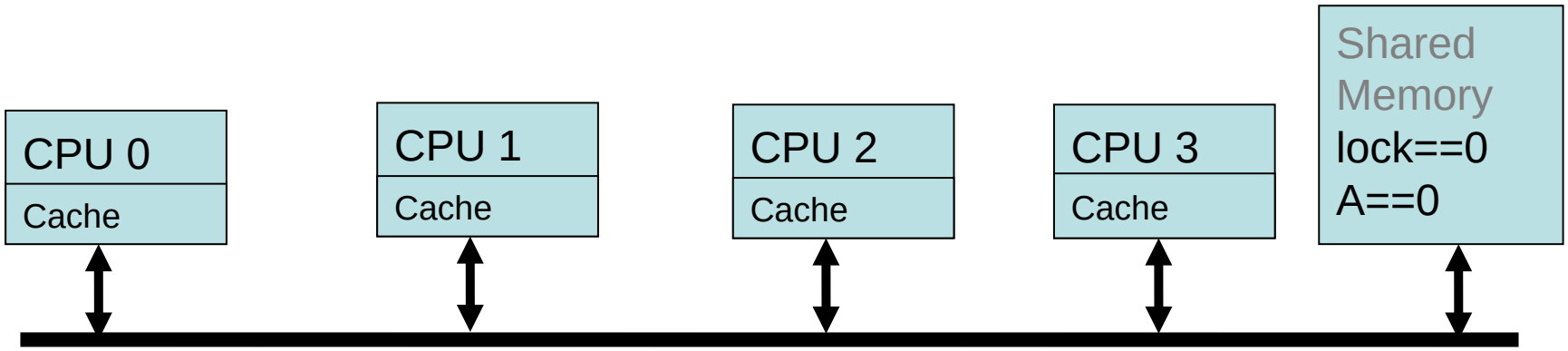
Instruction **test-and-set R2, lock**, executes atomically: **{load R2,lock; store #1,lock}**

Another variation of atomic instructions are operations fetch-and-xx (i.e., fetch-and-increment, fetch-and-add, fetch-and-store, …). If such an operation is used, then the program to increment A can be implemented by a single atomic instruction (or a C++ 11 construct, see later):

P1: fetch-and-inc A; P2: fetch-and-inc A;

MESI protocol and spinlock based on test-and-set instruction

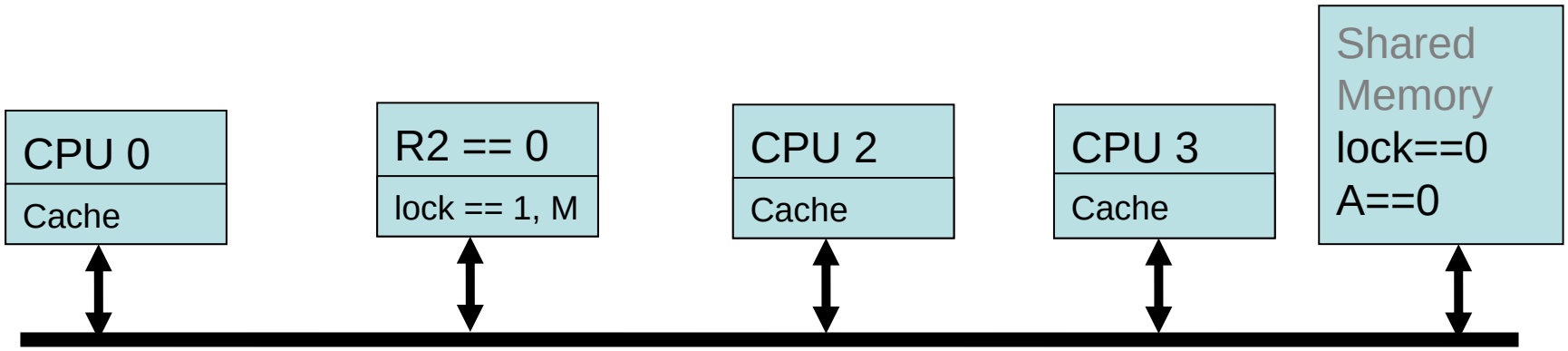
CPU 0	CPU 1	CPU 2	CPU 3
L: {load R2,lock; store #1,lock} bnz R2, L load R1, A addi R1, R1, 1 store R1, A store #0, lock	L: {load R2,lock; store #1,lock} bnz R2, L load R1, A addi R1, R1, 2 store R1, A store #0, lock	L: {load R2,lock; store #1,lock} bnz R2, L load R1, A addi R1, R1, 3 store R1, A store #0, lock	L: {load R2,lock; store #1,lock} bnz R2, L load R1, A addi R1, R1, 4 store R1, A store #0, lock



- All CPUs attempt to execute the test-and-set instruction. That is why all of them request the bus. Only one receives it in a given instant of time.

MESI protocol and spinlock based on test-and-set instruction

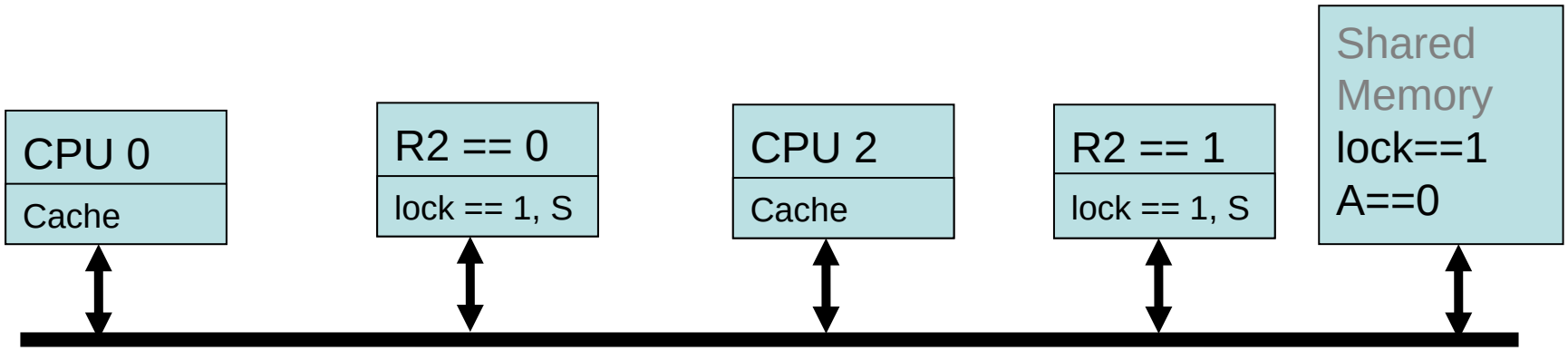
CPU 0	CPU 1	CPU 2	CPU 3
L: {load R2,lock; store #1,lock} bnz R2, L load R1, A addi R1, R1, 1 store R1, A store #0, lock	**L: {load R2,lock; store #1,lock} bnz R2, L load R1, A addi R1, R1, 2 store R1, A store #0, lock**	L: {load R2,lock; store #1,lock} bnz R2, L load R1, A addi R1, R1, 3 store R1, A store #0, lock	L: {load R2,lock; store #1,lock} bnz R2, L load R1, A addi R1, R1, 4 store R1, A store #0, lock



- CPU 1 obtains the bus. It reads the value of „lock“ variable from memory to R2 and writes 1 to memory. The write happens only in its cache. The cache line reaches state M (modified).

MESI protocol and spinlock based on test-and-set instruction

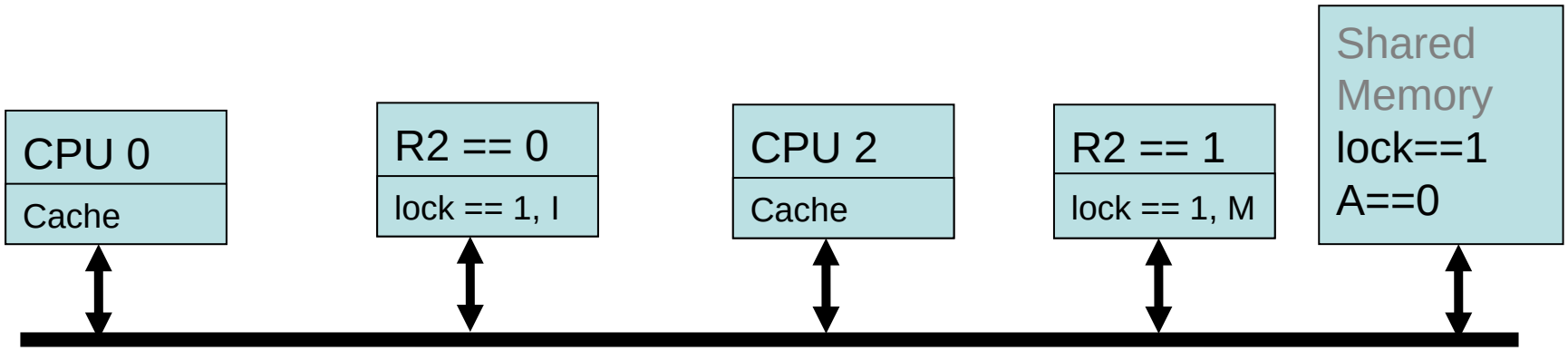
CPU 0	CPU 1	CPU 2	CPU 3
L: {load R2,lock;	L: {load R2,lock;	L: {load R2,lock;	**L: {load R2,lock;** ⇦
store #1,lock}	store #1,lock}	store #1,lock}	**store #1,lock}**
bnz R2, L	**bnz R2, L**	bnz R2, L	bnz R2, L
load R1, A	load R1, A	load R1, A	load R1, A
addi R1, R1, 1	addi R1, R1, 2	addi R1, R1, 3	addi R1, R1, 4
store R1, A	store R1, A	store R1, A	store R1, A
store #0, lock	store #0, lock	store #0, lock	store #0, lock



- CPU 3 obtains the bus. It requests to read „lock“ value to R2. Snooping CPU 1 recognizes MemRead request and propagates the modified data to CPU 3 and memory. The corresponding line changes to S. CPU 3 receives the data and its final cache state is S as well.

MESI protocol and spinlock based on test-and-set instruction

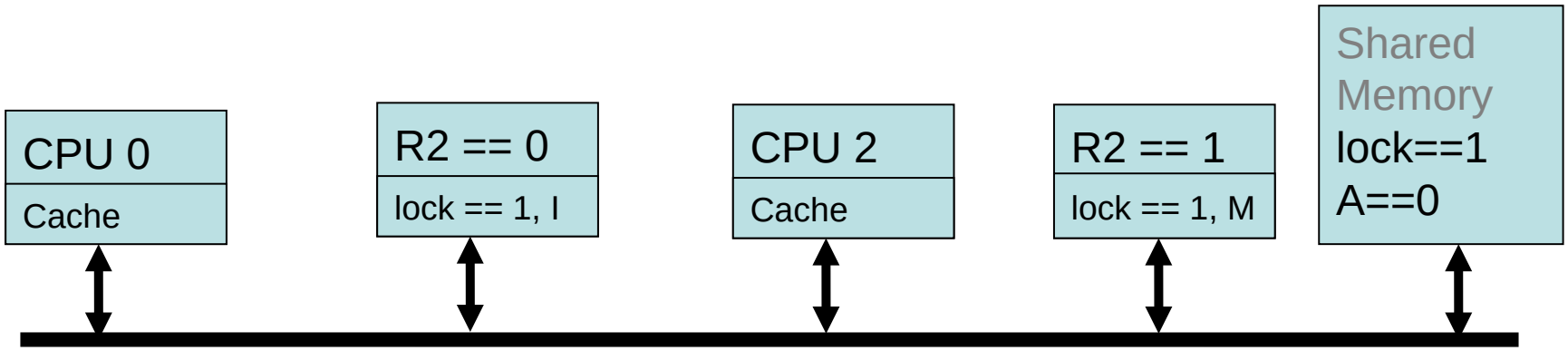
CPU 0	CPU 1	CPU 2	CPU 3
L: {load R2,lock;	L: {load R2,lock;	L: {load R2,lock;	**L: {load R2,lock;**
store #1,lock}	store #1,lock}	store #1,lock}	**store #1,lock}** ⇦
bnz R2, L	bnz R2, L	bnz R2, L	bnz R2, L
load R1, A	load R1, A	load R1, A	load R1, A
addi R1, R1, 1	addi R1, R1, 2	addi R1, R1, 3	addi R1, R1, 4
store R1, A	store R1, A	store R1, A	store R1, A
store #0, lock	store #0, lock	store #0, lock	store #0, lock



- CPU 3 keeps bus control (lock prefix). The next step of the atomic operation test-and-set is to write 1 to “lock” memory location. That is recognized by the snooping CPU 1, which changes its state to I (invalid). CPU 3 reaches state M. CPU3 releases the bus.

MESI protocol and spinlock based on test-and-set instruction

CPU 0	CPU 1	CPU 2	CPU 3
L: {load R2,lock;	L: {load R2,lock;	L: {load R2,lock;	L: {load R2,lock;
store #1,lock}	store #1,lock}	store #1,lock}	store #1,lock}
bnz R2, L	bnz R2, L	bnz R2, L	**bnz R2, L**
load R1, A	load R1, A	load R1, A	load R1, A
addi R1, R1, 1	**addi R1, R1, 2**	addi R1, R1, 3	addi R1, R1, 4
store R1, A	store R1, A	store R1, A	store R1, A
store #0, lock	store #0, lock	store #0, lock	store #0, lock



- CPU 0 obtains the bus. It reads “lock” and writes 1 to it. This results in invalidation of all the other caches and setting M in its own cache.
- CPU 3 tests R2, but the value is 1. It jumps to label L to repeat the attempt to acquire the bus and receive “lock”.

MESI protocol and spinlock based on test-and-set instruction

Observation:

- Each attempt to acquire the lock (successful or unsuccessful) modifies the value in cache line and requires its change to M state.
- Consequence is the invalidation of the corresponding cache line in all the other CPUs attempting to enter the critical section.
- Unsuccessful attempt to acquire the lock leads to the start of another attempt.
- When the number of CPUs increases, the bus load increases quadratically for both reads and writes.
- Remark: spinlock on single CPU without sleep or schedule disable causes deadlock.

Enhancement No 1:

- If an attempt to acquire the lock is unsuccessful, then delay the next attempt – sleep for exponentially increasing or random time.

Enhancement No 2:

- Execution of test-and-set instruction realizes 2 transactions on the bus, the second invalidating all the other caches. It is advantageous to make the attempt to write conditional by checking that the lock is empty – only single MemRead transaction is repeated, which results in state S until there is a chance to acquire the lock released by other CPU. Bus load is decreased and continuous caches trashing is eliminated. But use of **ll** and **sc** is even better.

Atomics implemented by load link + store conditional

Problem:
Consider two processors P1 and P2 and a shared variable A.
P1: A = A+1; P2: A = A +2;

Another alternative is the instruction pair ***load-locked*** (ll) (or **load-link, load-linked, load-and-reserve**) and ***store-conditional*** (sc), found in many modern ISAs.

- Instruction **ll** returns the value stored in memory, **sc** stores a new value to the address only if the value on linked address has not been modified by other thread/CPU – atomic operation is successful – implementation can be based on *load address register (LAR)* and added *lock flag (LF)*.

```
loop:   ll R1, A        // read A into R1, address A into LAR. LF=1;
        addi R1, R1, 1
        sc R1, A        // if(LF==1) store R1 into A;        R1=LF;
        bz R1, loop
```

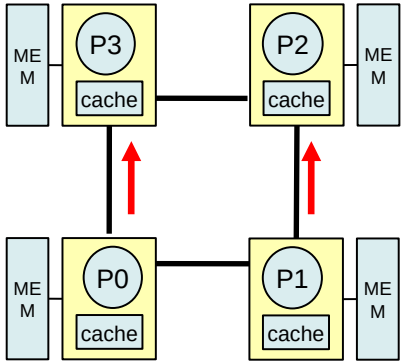
- IBM PowerPC, DEC Alpha, MIPS, ARM, RISC-V, IA-64

Atomics implemented by load link + store conditional

- ll and sc implementation requires at least minimal support in HW: link address register **LAR** for address monitoring and link flag **LF**.
- **ll** instruction: sets LF and LAR value – location or cache line is **reserved**/remembered for monitoring.
- **sc** instruction: if LF==1, then store data into memory. Return LF value.
- **Important: sc** instruction does not generate any transaction for unsuccessful state = does not invalidate cache lines.
- **When content is changed** or **exception/interrupt** occurs**:** clear LF
- **Possible cache controller ll+sc support:**
  - compare RWITM transactions addresses with address stored in LAR. Clear LF in case of addresses match.
  - Do not allow linked cache line replacement as a result of cache lines reuse (cache replacement policy – e.g. LRU) when LF==1. Replacement would clear LF and result in situation when sc can never succeed. That would result in infinite repeat of code between ll-sc $\rightarrow$ **active blocking – livelock**. SW-side solution is to forbid the use of any memory referencing instructions = no read, no write between ll and sc instructions and use memory barriers for out-of-order execution to prevent instructions in and around the ll+sc block to get out of or into the ll+sc region.

Discussion

- Compare **test-and-set** and instruction pair **ll-sc** methods. Which variant loads the bus less?
- Is the memory coherent model enough to ensure sequential consistency model for lock?
- Shared bus is not used today for cores/processors interconnection. It is possible that more requests are in flight simultaneously…
  - What happens if 2 processors do RWITM simultaneously?
  - What happens if requests and responses are delivered to different processors in different order?



Solution:

Serialization (or synchronization) of requests (required for coherence and consistency) – same as on the bus … But there is no shared bus …
Instead of serialization: Home Node (see previous lecture), but only for single address/memory block

Discussion

- In a sequential program exists this fragment of code:

```
Instr.1:    load  R1, A   // read of value A from memory to R1
Instr.2:    load  R2, B
Instr.3:    store R3, C   // value of R3 into C
Instr.4:    load  R4, D
…
Instr.N:    store R5, A
```

Question No 1

- Is there problem to finish (execute) instruction No 2 before No 1?

Question No 2

- Is there problem to finish (execute) instruction N before No 1?

Question No 3

- Is there problem to finish (execute) instruction No 4 before No 3?

We already know...

- **Load / Store instructions** are responsible for data transfers from and to memory and processor general purpose registers
- Processor is equipped with only limited number of registers
- Compiler generates so called ***spill code***, which swaps used variables data into memory temporarily to make registers available for processed variables – load/store instructions are used for this task

- Data dependencies – RAW, WAR, WAW – between load/store instructions referencing the same address
- **Total ordering** – keeping the program order of all load/store instructions. Is it necessary?

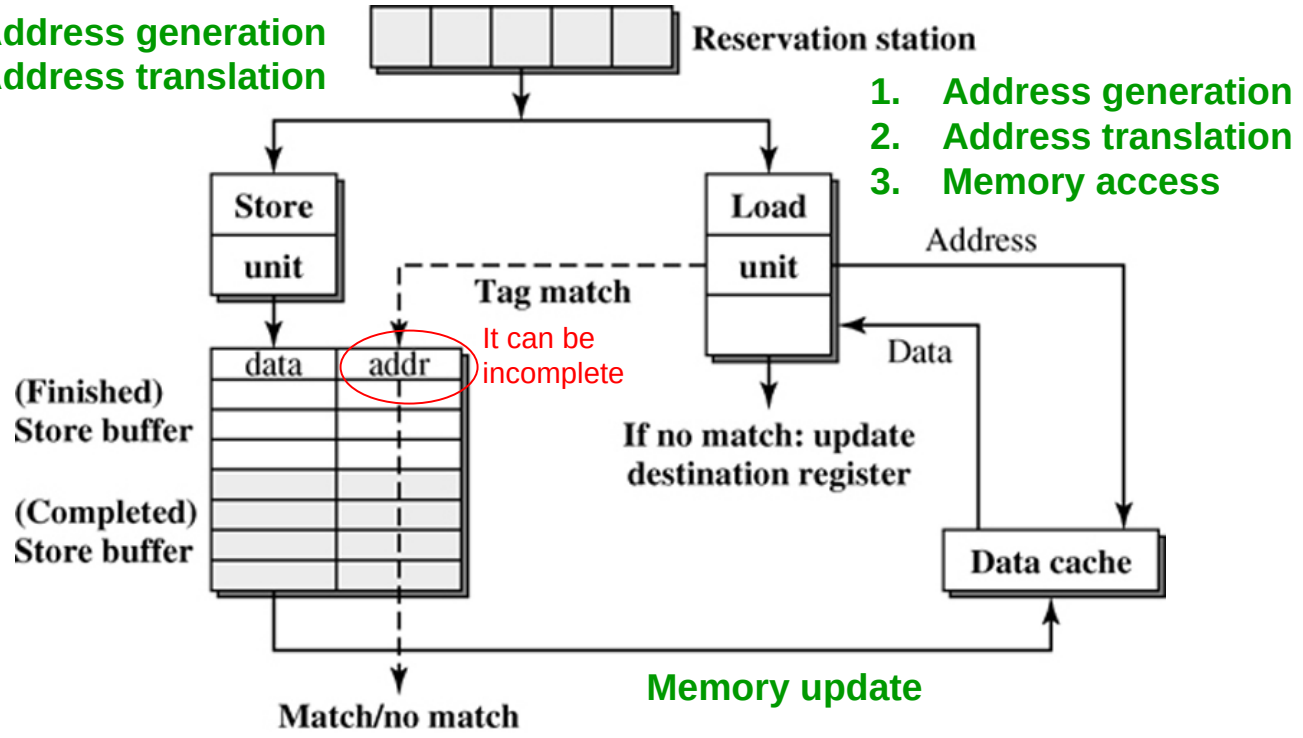
Model of sequential consistency

- **The requirement of sequential consistency** results in some restrictions on out-of-order execution of load/store instructions
- What happens if an exception occurs?
- Memory state must be based **on the sequential order** of load/store instructions
- This results in requirement that memory operations must be executed in program order, or precisely, that memory must be updated as if the instructions were executed in program order
- If **store** instructions are executed **in program order**, the fulfillment of WAW and WAR dependencies is guaranteed. RAW dependencies are the only ones to care about …
- Load instructions – out-of-order

Load forwarding and Load bypassing

For now, expect issuing of load/store instructions from the reservation station in order

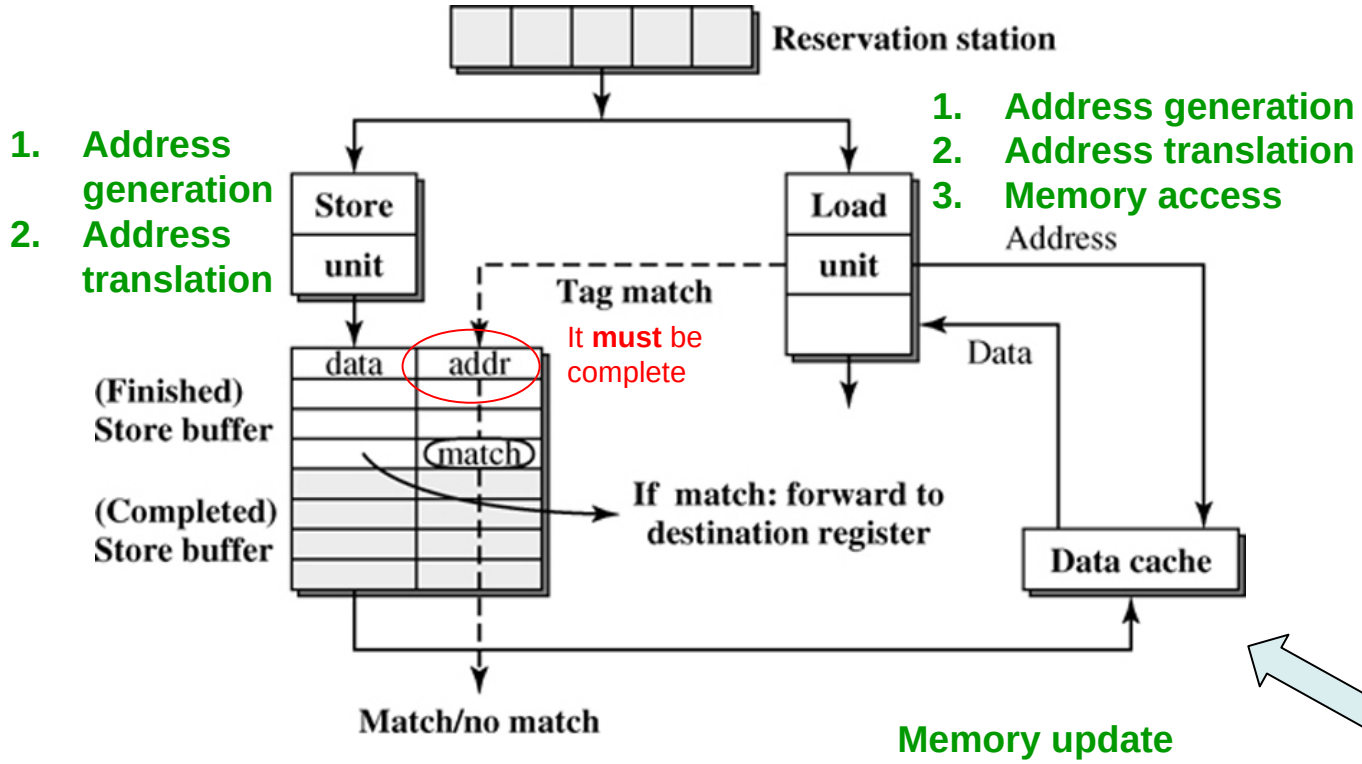
- ***Load bypassing*** allows to execute load before store if they are memory independent. In other cases (if dependency exists): Load forwarding.



Load forwarding and Load bypassing

For now, expect issuing of load/store instructions from the reservation station in order

- ***Load bypassing*** allows to execute load before store if they are memory independent. In other cases (if dependency exists): Load forwarding.



Store: dispatched, issued, finished, completed, retired

Load – if match: discard read data and take one available in Store buffer

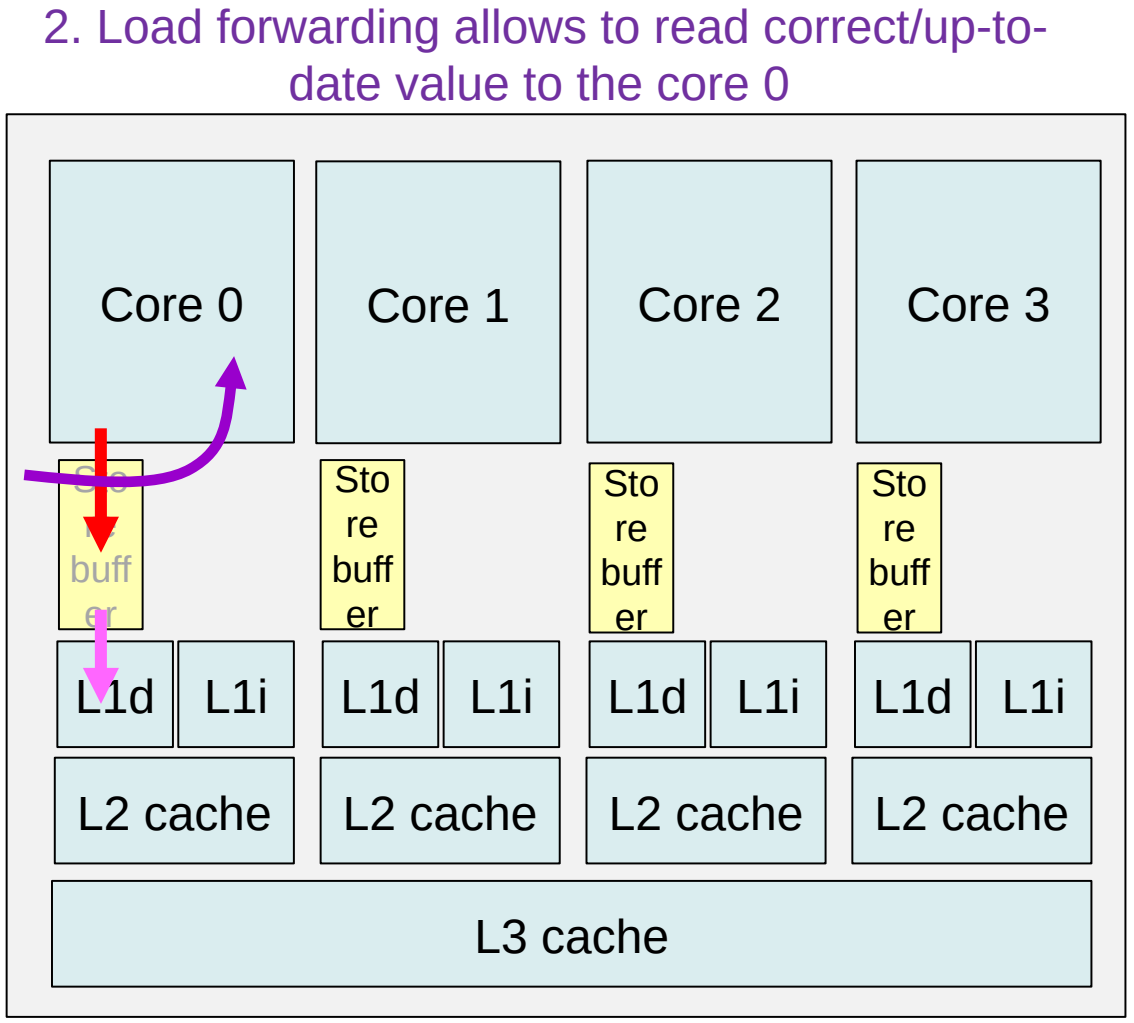
This solution (complete address) allows both: load bypassing and load forwarding

Store buffer

- Store buffer use enables **significant speedup** of sequential program execution… However:

1. Memory write is recorded in Store buffer

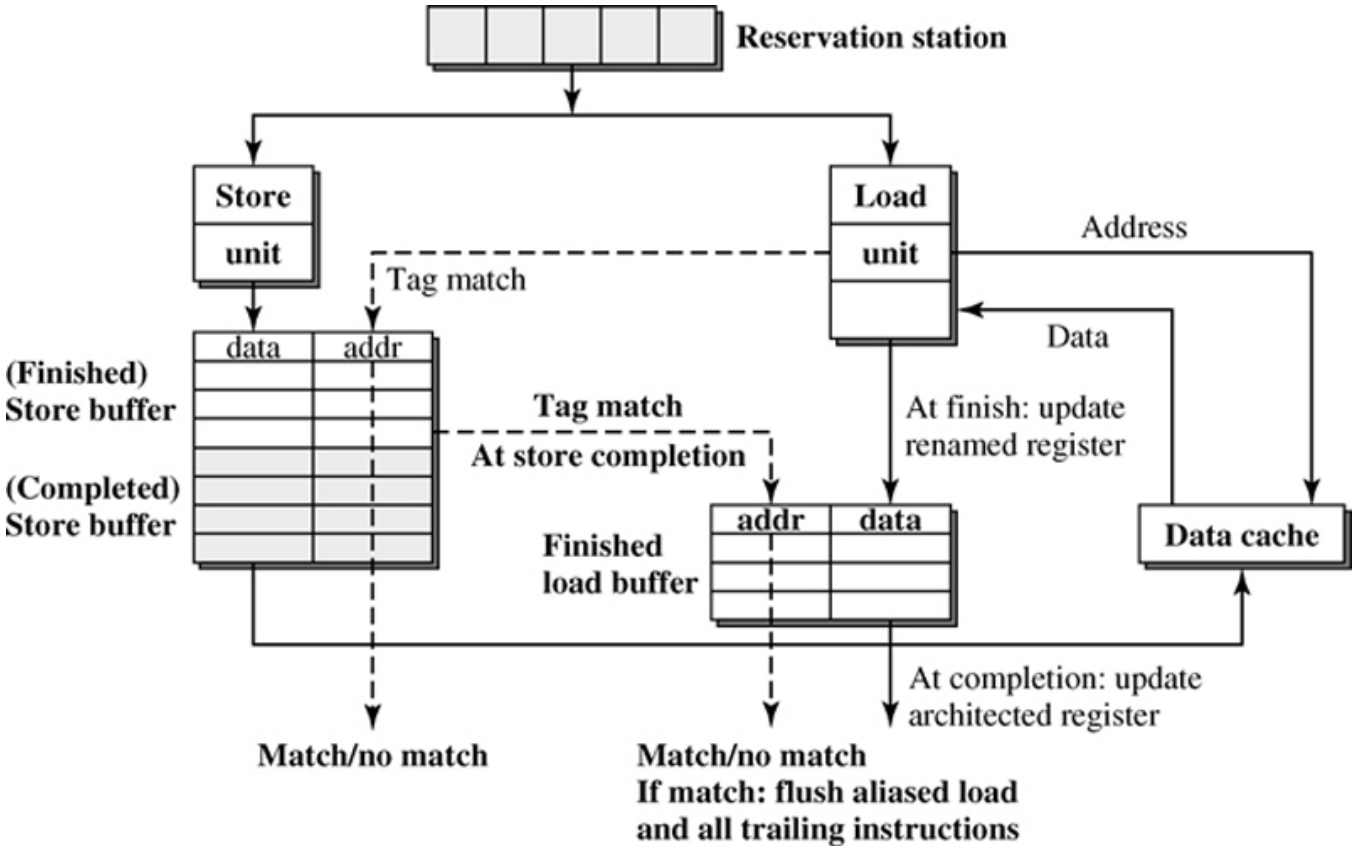
3. Propagation from Store buffer into cache which triggers coherence mechanism takes some time (same to regain consistency). Consistency is violated for that time.



Load forwarding and Load bypassing

- If it is allowed to **issue** instructions from reservation station **out-of-ordert**, then it is possible that a load instruction can be already executed but a preceding conflicting store (RAW hazard) is not in the store buffer yet (it can be executed, in reservation station or even in dispatch buffer). Information about conflicting store address is not known and RAW hazard cannot be detected.
- Solution?
- Assume that there is no dependency and check for dependency later … => **speculative execution**
- Speculative load execution is supported by ***Finished load buffer*** (Finish load queue)

Speculative execution of load instructions



- Load instruction is stored in **Finished load buffer** between execution finishing and completion.
- Each time when store reaches completion, alias checking with FLB entries is performed. No conflict → store is finished ; Conflict→abandon load instr. speculation

Speculative execution

- Why to enable speculation of load instructions?
- It is useful to perform load as early as possible – other computation depends on it usually
- In addition, earlier load execution can initiate *cache miss* in advance
- It can mask *cache miss penalty* (main memory access time)

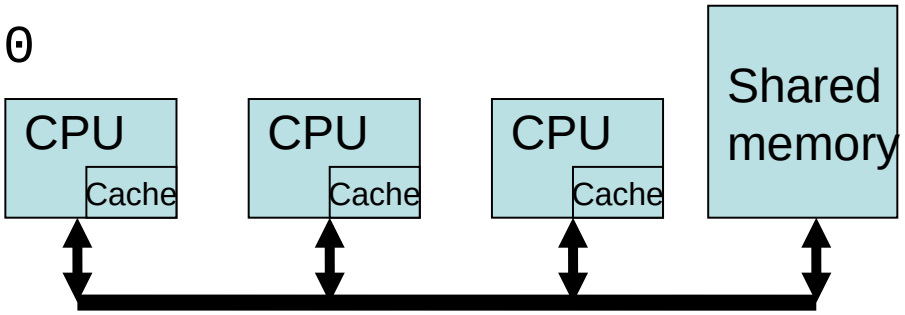
- However: In case of incorrect speculation – abandoning of speculated instructions (sequence starting by load) costs time and resources which could be better utilized...
- That is why to add: ***Dependence prediction***

  Dependency between store and load is quite predictable for typical programs
- ***Memory dependence predictor*** then decides if the speculative load and following instruction should be started

Execute example program on this system

Let the variables be initialized: `x=0, y=0`

```
P1:          P2:
x = 1;       while(y==0){;}
y = 1;       print(x);
```



Assume sequential consistency

But possible cache miss cannot be propagated out …

One of possible scenarios:

1. Processor P2 does not find y in cache and initiates a request to read from memory. The bus has to be obtained through arbitration first.
2. Processor P2 starts reading of x speculatively – line „print(x)“. It finds x value (0) in its cache. Speculation is conventionalized by variable y==1.
3. Processor P1 acquires the bus and executes write to variable x: „x=1“. The corresponding cacheline is marked as M (MESI protocol) and invalidated in P2. This collides with address remembered for step 2 speculation. It is abandoned.
4. Processor P1 acquires the bus and writes y=1 into memory.
5. Processor P2 acquires the bus and reads the value of y.
6. Processor P2 acquires the bus, requests the value of x, P1 cache changes state M→S, simultaneously sending the value of x to memory and to P2 which changes state I→S. P2 outputs 1.

Sequential consistency – summary

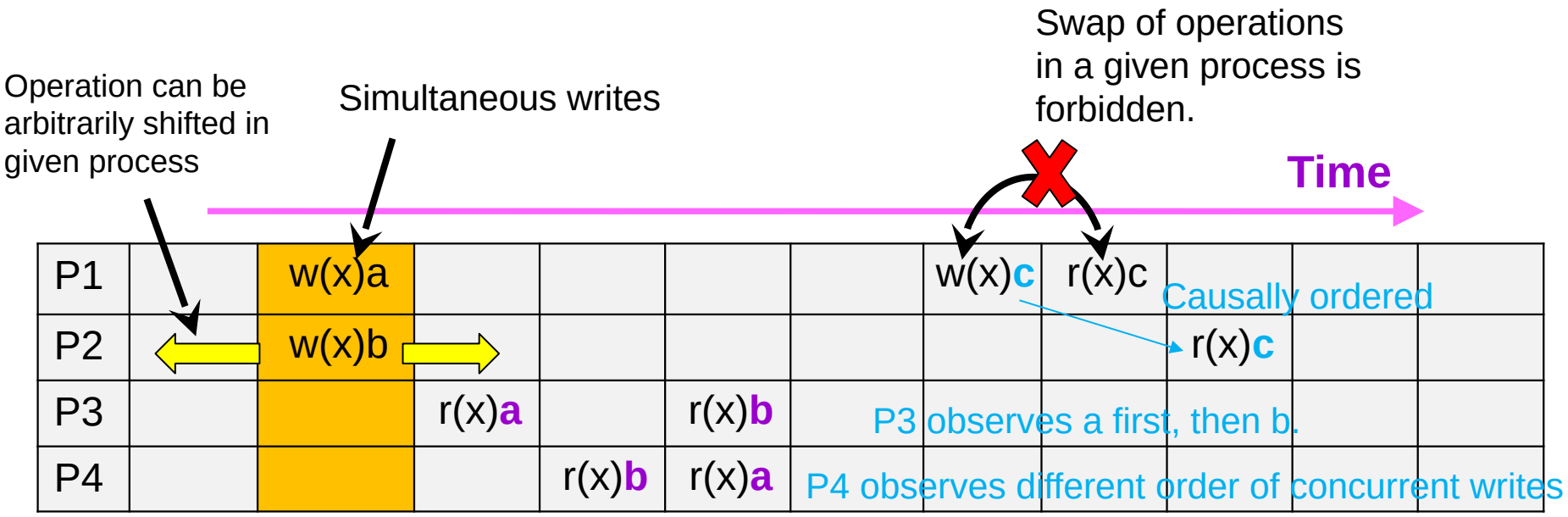
- Significant efforts have been made to accelerate the execution of applications on single-core processors – out-of-order, speculation, store buffer before cache, ...
- These techniques are often not compatible with the sequential consistency model
- So what will we give up?
- Answer: The sequential consistency model

- But how can we ensure that the programmer does not get unexpected results?
- Answer: We will offer another consistency model – it will provide a sequentially consistent view only at certain times
- For this we need additional instructions … => HW and ISA support

Another consistence models

Causal consistency *(Hutto, Ahamad, 1990)*

- Writes that are potentially causally bound must be seen by all processes in the same order. Concurrent writes may be seen in different order.
- Distinguishing events that are potentially dependent and which are not
  - Reading on a given P is causally ordered before writing (even to another address) – the written value may depend on the read value
  - Reading is causally ordered after an earlier write to the same address if the read has received data written by that write
  - Writes at the same address by a given P are causally ordered as they were performed
- **Weaker than sequential consistency**



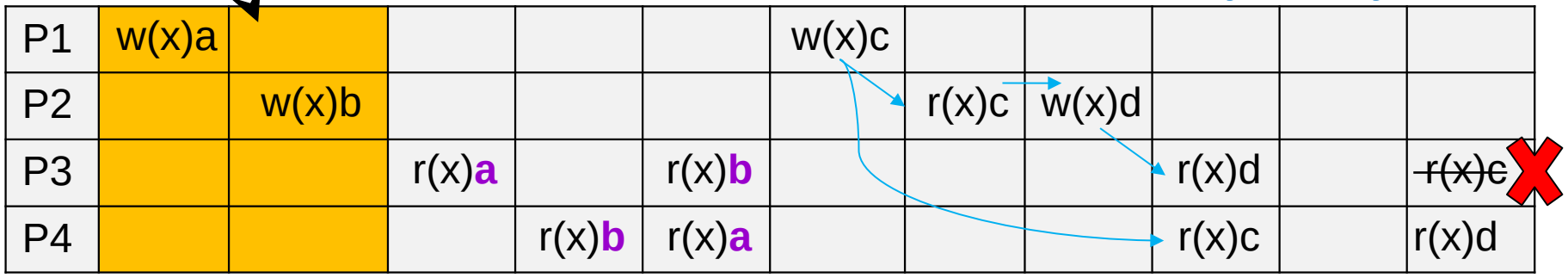
Another consistence models

Causal consistency *(Hutto, Ahamad, 1990)*

- Writes that are potentially causally bound must be seen by all processes in the same order. Concurrent writes may be seen in different order.
- Distinguishing events that are potentially dependent and which are not
  - Reading on a given P is causally ordered before writing (even to another address) – the written value may depend on the read value
  - Reading is causally ordered after an earlier write to the same address if the read has received data written by that write
  - Writes at the same address by a given P are causally ordered as they were performed
- **Weaker than sequential consistency**

Writes are not causally bound – simultaneous writes

Write w(x)d on P2 is causally bound to earlier r(x)c, which is causally bound to write w(x)c on P1. That is why these writes are causally bound as well and systems has to ensure their order: w(x)c < w(x)d. This ensures that on P3 r(x) last read cannot return c because d has been already seen by P3.



Another consistence models

- ***PRAM consistency*** *(*pipelined random access memory consistency*) = FIFO consistency,* (*Lipton, Sandberg (1988)*
  - Writes executed by one process are seen by other processes in the order in which they were performed, but the writes executed by different processes can be seen by different processes differently (permuted).
  - **Weaker than sequential consistency**

Writes by different processors can be seen in different order

Does not obey causality principle. Writes originate on different processors and that is why P3 can observe these in an order different to P4.

P1	w(x)a					w(x)c					
P2		w(x)b					r(x)c	w(x)d			
P3			r(x)**a**		r(x)**b**				r(x)d		r(x)c ✓
P4				r(x)**b**	r(x)**a**				r(x)c		r(x)d

To recall

- In a sequential program exists this fragment of code:

```
Instr.1:    load  R1, A   // read of value A from memory to R1
Instr.2:    load  R2, B
Instr.3:    store R3, C   // value of R3 into C
Instr.4:    load  R4, D
…
Instr.N:    store R5, A
```

Question No 1

- Is there problem to finish (execute) instruction No 2 before No 1?

Question No 2

- Is there problem to finish (execute) instruction N before No 1?

Question No 3

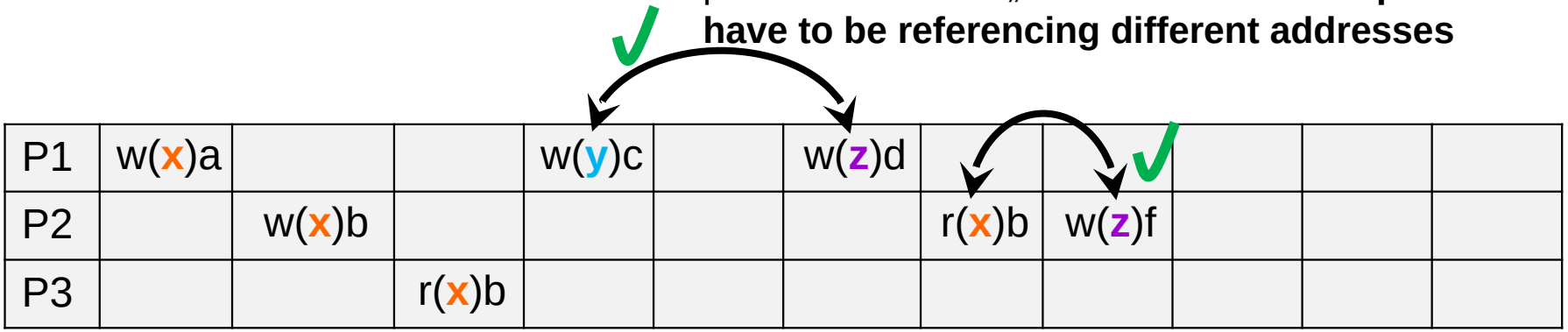
- Is there problem to finish (execute) instruction No 4 before No 3?

More consistency models

Relaxed consistency

- Sequential consistency preserves order of reads and writes:
  1. W → R: write must be finished before the following read
  2. R → R: read must be finished before the following read
  3. R → W: read must be finished before the following write
  4. W → W: write must be finished before the following write
- Relaxed consistency leaves out some of these requirements
- Additionally, we can leave out the requirement of a unique sequence interlace of instructions seen by all processors equally when:
  5. A processor can observe the result of its write before it is seen by other processors
  6. A processor can observe the result of other processor's write before it is seen by others

It is possible to „reorder“ instructions in a given process based on „relaxation“. **But the operations have to be referencing different addresses**



More consistency models

Relaxed consistency – What are the benefits?

- **W → R**: removes Write from critical path – overlap of Write and following Read „reduces“ memory latency for Write. (Write in a coherent NUMA system means not only a write, but also finding of a valid block – queries to home node, distribution of invalidation to all others with block, block reading, etc.)
- **R → R and R → W:** nonblocking cache – it is possible to continue with execution even after read miss; waiting for the miss to be serviced is not necessary – speculative execution
- **W → W**: memory level parallelism
- **Read of own write before others**: Load forwarding – store buffer before cache → speedup of program execution
- **Read of other processor's write before others**: read from memory before the change is distributed to all others

- Thus, the relaxation allows parallel execution. Forced serialization required by sequential consistency is suppressed.

More consistency models

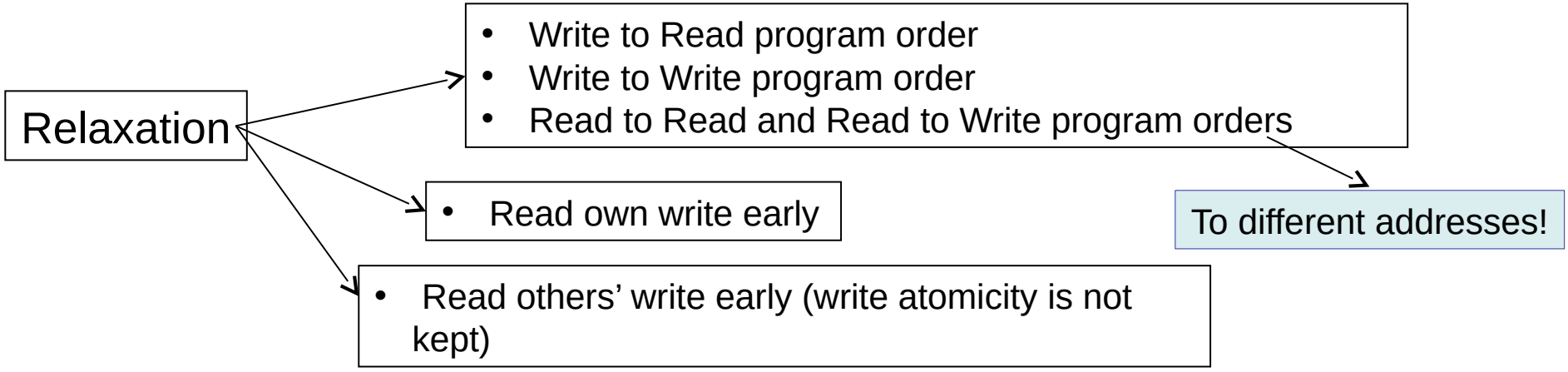
Relaxed consistency

Following models falls into the relaxed consistency category:

- **Total Store Ordering** (TSO) – IBM 370: a read operation can be completed before an earlier write to another address, but the read cannot return the written value until the write is visible to all the other nodes
- **Total Store Ordering** (TSO) – SPARC: a read operation can be completed before an earlier write to another address. Read cannot return a value written by another processor until the write is visible to all the other processors. But the processor can return own written value before this write is visible to the other processors.
- **Processor Consistency** (PC): read can be completed before an earlier write (arbitrary processor to arbitrary place) is visible to all. That is, a read executed on some of the processors can return the new value while a read executed on other processors still returns the old value.
- **Partial Store Ordering** (PSO) – similar to TSO. Difference: PSO preserves only the order of writes to the same address; writes to different locations can be reordered.
- And more…

More consistency models

Relaxed consistency



	W->R	W->W	R->R,W	Read own write before others	Read others write before others
TSO – IBM 370	x				
TSO – SPARC: SPARC, IA-32, Intel64, AMD64	x			x	
PC	x			x	x
PSO	x	x		x	
Weak consistency: PowerPC, ARMv7, IA-64	x	x	x	x	x

Consistency model of IA-32 and Intel64

Intel Core i5, Core i7, Intel Xeon, Intel Core2 Extreme

- Read in respect to read and write in respect to write on a given processor are not reordered (exception are special long string store and string move write operations) – that is, R->R and W->W is not relaxed
- Write cannot precede an earlier read – that is, R->W is not relaxed
- Read can precede an earlier write to different address – relaxed W->R, Dekker's algorithm can fail to protect critical section

P1:	**P2**:
X=1;	Y=1;
R1=Y;	R2=X;

For initial values X=Y=0, it can return P1.R1=0 and simultaneously P2.R2=0.

- Read cannot precede an earlier write to the same address
- Load-forwarding inside a given processor is allowed – that is, read of own write before others

P1:	**P2**:
X=1;	Y=1;
R1=X;	R3=Y;
R2=Y;	R4=X;

For initial values X=Y=0, it can return P1.R2=0 and simultaneously P2.R4=0.

Consistency model of IA-32 and Intel64

Intel Core i5, Core i7, Intel Xeon, Intel Core2 Extreme

- Writes are visible transitively – writes which are causally bound are seen by all the other processors in the same order

P1:	**P2**:	**P3**:
X=1;	R1=X;	R2=Y;
	Y=1;	R3=X;

For initial values X=Y=0, it cannot return P2.R1=1, P3.R2=1 and simultaneously P3.R3=0.

- Writes are seen by **all other** processors in same order – processor executing write can see different order

P1:	**P2**:	**P3**:	**P4**:
X=1;	Y=1;	R1=X;	R3=Y;
		R2=Y;	R4=X;

For initial values X=Y=0, it cannot return P3.R1=1, P3.R2=0, P4.R3=1 and simultaneously P4.R4=0.

- IA-32 and Intel64 architectures comply with TSO – SPARC consistency.

Which behavior can be expected for these code fragments?

Example A:

P1:
```
A=1;
flag=1;
```

P2:
```
while(flag==0);
print(A);
```

Example B:

P1:
```
A=1;
B=1;
```

P2:
```
print(B);
print(A);
```

Example C:

P1:
```
A=1;
```

P2:
```
while(A==0);
B=1;
```

P3:
```
while(B==0);
print(A);
```

Example D:

P1:
```
A=1;
print(B);
```

P2:
```
B=1;
print(A);
```

Would the code be executed with conformance to sequential consistency?

	Example A	Example B	Example C	Example D
TSO – SPARC	Yes	Yes	Yes	No
PC	Yes	Yes	No	No
PSO	No	No	No	No
Weak consistency	No	No	No	No

Assuming that the compiler follows the order of lines/operation… Initial values: A=flag=0.

How to achieve desired behavior of program

Use a **memory barrier** (more types, consider full for now)

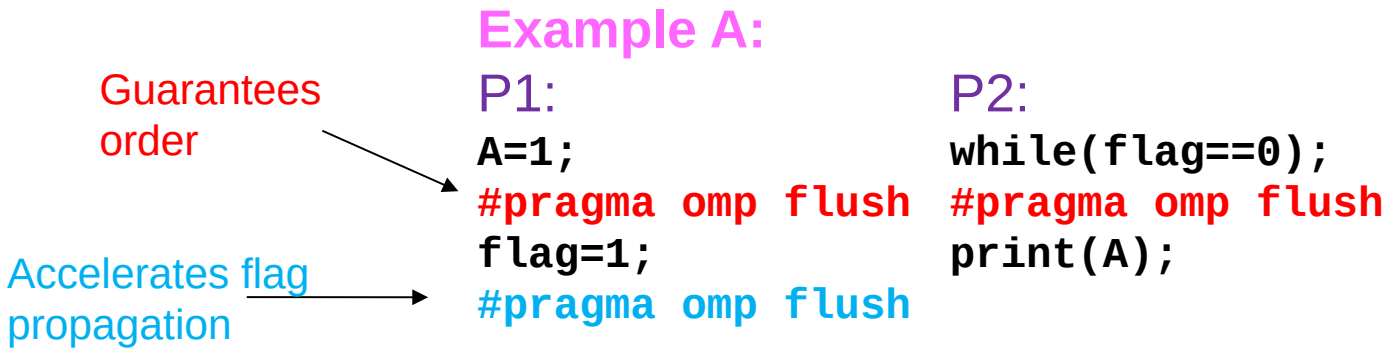
- All data operations (instructions) BEFORE the barrier have to complete
- All data operations (instructions) AFTER the barrier have to wait until the barrier instruction is completed
- Barrier instructions are processed in program order

Programmer has to accept that **memory operations working with shared variables can be arbitrarily reordered** in each code sequence block. These blocks are separated by barriers.

- IA-32, Intel64 defines three barrier instructions: **sfence, lfence, mfence**
  - Sfence – all store operations before the barrier have to be completed before the first store after the barrier instruction is executed
  - Lfence – all load instructions before the barrier have to be completed before the first load after the barrier is executed
  - Mfence – all memory operations have to be finished (be globally visible) before the first memory operation after the barrier instruction is executed
- PowerPC ISA defines **sync** instruction
- OpenMP defines **flush** directive

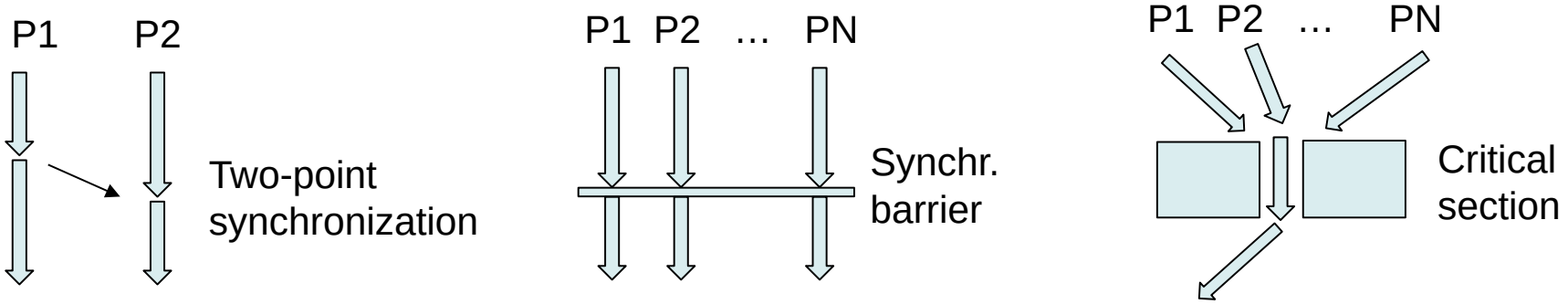
How to achieve desired behavior of program

Use a **memory barrier**



- It is guaranteed that P2 will read 1 from variable A. Memory operations in the block before, between and after the barriers can be reordered by the compiler and/or hardware, but that does not influence the program result.
- The barrier instruction implementation must ensure that shared variables (thread-visible) are visible to all threads/processors after this directive $\rightarrow$ the compiler must ensure that for all such variables the values from registers are written to memory (Write/SW instructions are inserted), processor flushes write-buffers, etc.
- **Memory barrier ensures sequentially consistent view of memory only in defined instants of time – the operation has to be considered by all the participating threads/processors.**

Synchronization events types



The following synchronization types are distinguished in parallel programming:

- **Two-point synchronization**: ensures safe data passing between two processes (threads). The first one can eventually continue in execution without the need to wait – see the previous slide (or can be implemented by **semaphore**)
- **Synchronization barrier:** all processes from the given process group must wait at this point until the last one reaches the barrier; then they can continue (Warning: do not confuse this barrier with the term memory barrier by mistake)
- **Mutual exclusion** – critical section: Only one of the processes can acquire access to the marked code block and the others need to wait until it exits the block (often implemented by **mutex**)

Synchronization events types defined by OpenMP

Two-point synchronization:

#pragma omp flush
as has been shown already

Synchronization barrier:

…
#pragma omp barrier
…

Critical section:

```
#pragma omp critical
{
    … // A = A+1;
}
```

Notice: **Flush** operation (memory barrier) is inserted by two-point synchronization directive as well for the synchronization barrier and at critical section entry and exit – it is important for ensuring sequentially consistent memory view at the given location.

The already introduced and described instructions for memory synchronization and atomic operations (test-and-set, pair ll-sc), together with memory barrier instructions (enforcing sequentially consistent view in a given instant of time) are the building blocks for implementation of the synchronization events described above.

Using synchronization only in minimal required form

- Programming models using full memory barriers are too restrictive for efficient use of processor cores.
- It is useful to define memory models which allow to precisely define the purpose of shared variable access/modification. There are the following use cases:
  - atomic operation concerning only the specified variable (**relaxed** model, i.e. A+=2)
  - confirmation of data availability in some other variables (**release**)
  - for checking that data protected by the read/modified variable are ready (**consume**)
  - for overtaking of control/lock for related resources (**acquire**)
  - combined (**acq-rel**)
  - complete synchronization (**seq_cst**)

  Only last one, the most expensive (Sequentially-consistent ordering), corresponds to the synchronization events introduced before.
- The most sophisticated model of these operations is probably the one defined in the C++11 language standardhttp://en.cppreference.com/w/cpp/atomic/memory_order

Ticket-lock based on C++ memory model

- Ticket-lock is a spinlock implementation, critical section with busy waiting
- Peter Cordes – analysis of question on StackOverflow about implementation optimization for GCC

  https://stackoverflow.com/questions/33284236/implementing-a-ticket-lock-with-atomics-generates-extra-mov

```
#include <atomic>

struct atom_ticket { std::atomic<uint32_t> next_ticket,now_serving;};

void lock_acquire(atom_ticket* tkt) {
    const auto my_ticket =
        tkt->next_ticket.fetch_add(1,std::memory_order_acquire);
    while (tkt->now_serving.load(std::memory_order_acquire) !=
           my_ticket) {
        _mm_pause(); /* x86 specific, #include <immintrin.h> */
    }
}

void lock_release(atom_ticket* tkt) {
    tkt->now_serving++; // variable data type ensures atomic increment
}                       // the strongest memory_order_seq_cst model is used
```

Ticket-lock – compilation for x86_64

- Compilation by https://gcc.godbolt.org/
- x86_64 gcc 5.2 -std=gnu++1y -Wall -O3 -ffast-math -fverbose -asm -march=native -mtune=native

```
lock_acquire(atom_ticket*):
        mov         edx, 1
        lock xadd   DWORD PTR [rdi], edx
        add         rdi, 4
.L2:
        mov         eax, DWORD PTR [rdi]
        cmp         edx, eax
        jne         .L2
        rep ret

lock_release(atom_ticket*):
        lock add    DWORD PTR [rdi+4], 1
        ret
```

Ticket-lock – compilation for MIPS

- Compilation by https://gcc.godbolt.org/
- MIPS gcc 5.4 -std=gnu++1y -Wall -O3 -ffast-math

```
lock_acquire(atom_ticket*):
1:  ll      $3,0($4)
    addiu   $1,$3,1
    sc      $1,0($4)
    beq     $1,$0,1b
    nop
    sync
    addiu   $4,$4,4
2:  lw      $2,0($4)
    sync
    bne     $3,$2,2b
    nop
    jr      $31
    nop
```

```
lock_release(atom_ticket*):
    sync
1:  ll      $1,4($4)
    addiu   $1,$1,1
    sc      $1,4($4)
    beq     $1,$0,1b
    nop
    sync
    jr      $31
    nop
```

Ticket-lock – compilation for ARM Aarch64

- Compilation by https://gcc.godbolt.org/
- ARM64 gcc 6.3 -std=gnu++1y -Wall -O4

```
lock_acquire(atom_ticket*):
.L4:ldaxr   w2, [x0]
    add     w1, w2, 1
    stxr    w3, w1, [x0]
    cbnz    w3, .L4
    add     x0, x0, 4
.L2:ldar    w1, [x0]
    cmp     w2, w1
    bne     .L2
    ret
```

```
lock_release(atom_ticket*):
    add     x0, x0, 4
.L7:ldaxr   w1, [x0]
    add     w1, w1, 1
    stlxr   w2, w1, [x0]
    cbnz    w2, .L7
    ret
```

Ticket-lock – compilation for ARM 32-bit

- Compilation by https://gcc.godbolt.org/
- ARM gcc 6.3.0 -std=gnu++1y -Wall -O4

```
lock_acquire(atom_ticket*):
    push {r4, r5, r6, lr}
    mov  r1, #1
    mov  r5, r0
    bl   __sync_fetch_and_add_4
    mov  r6, r0
    add  r5, r5, #4
.L2:ldr  r4, [r5]
    bl   __sync_synchronize
    cmp  r6, r4
    bne  .L2
    pop  {r4, r5, r6, lr}
    bx   lr
```

```
lock_release(atom_ticket*):
    push {r4, lr}
    add  r0, r0, #4
    mov  r1, #1
    bl __sync_fetch_and_add_4
    pop  {r4, lr}
    bx   lr
```

Ticket-lock – compilation for ARM Cortex-A7

- Compilation by https://gcc.godbolt.org/
- ARM gcc 6.3 -std=gnu++1y -Wall -O4 -march=armv7-a

```
lock_acquire(atom_ticket*):
.L4:ldrex   r2, [r0]
    add     r3, r2, #1
    strex   r1, r3, [r0]
    cmp     r1, #0
    bne     .L4
    add     r0, r0, #4
    dmb     ish
.L2:ldr     r3, [r0]
    dmb     ish
    cmp     r2, r3
    bne     .L2
    bx      lr
```

```
lock_release(atom_ticket*):
    add     r0, r0, #4
    dmb     ish
.L7:ldrex   r3, [r0]
    add     r3, r3, #1
    strex   r2, r3, [r0]
    cmp     r2, #0
    bne     .L7
    dmb     ish
    bx      lr
```

Ticket-lock – compilation for RISC-V RV64G (I.e with AMO)

- Compilation by https://gcc.godbolt.org/
- RISC-V (64-bits) gcc 12 -std=gnu++1y -Wall -O4 -march=rv64g

```
lock_acquire(atom_ticket*):
    li      a5, 1
    amoadd.w.aq a4,a5,0(a0)
    sext.w  a4, a4
    addi    a0, a0, 4
.L2:
    lw      a5, 0(a0)
    fence   iorw, iorw
    bne     a4, a5, .L2
    ret
```

```
lock_release(atom_ticket*):
    li      a5,1
    addi    a4,a0,4
    fence iorw,ow
    amoadd.w.aq zero,a5,0(a4)
    ret
```

- Atomic Memory Operations (AMOs) – two bits – 4 combinations
  - Acquire (aq) – AMO before any following loads or stores
  - Release (rl) – AMO after any earlier loads or stores
- AMOs only order the AMO w.r.t. other loads/stores/AMOs
- FENCEs order every load/store/AMO before/after FENCE

Memory model for parallel programming and Linux

- Paul E. McKenney, IBM
- Memory Ordering in Modern Microprocessors,
  http://www2.rdrop.com/users/paulmck/scalability/paper/whymb.2010.07.23a.pdf
- Is Parallel Programming Hard, And, If So, What Can You Do About It?
  https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html
- SMP Scalability Papers
  http://www2.rdrop.com/users/paulmck/scalability/
- Read-Copy-Update (RCU) papers
  http://www2.rdrop.com/users/paulmck/RCU/

Conclusion and summary

- Definition of sequentially consistent memory system is eligible for synchronization in parallel computers.
- Today computer systems support some of the weaker models of memory consistency, where the sequential consistency behavior may be specified only in defined locations of programs with the use of some of the synchronization operations.
- Synchronization operations are mutual exclusion, conditional/semaphore synchronization (two-point synchronization), and synchronization barrier.
- Implementation of synchronization operations is based on atomic RMW primitives and memory barriers.
  - Processors ISA includes RMW instructions T&S, SWAP, F&I, C&S
  - Newer processors support building of RMW primitives by inclusion of LL and SC instructions, which allow an efficient implementation of synchronization operations in systems with cache memories
- Memory barrier ensures separation/ordering of memory operations before and after the barrier. These instructions are found in different variants in ISA of contemporary processors as well.

Literature and references

1. Bečvář M: Přednášky Architektury paralelních počítačů II: Sekvenční konzistence paměti, Implementace synchronizačních událostí, s použitím slajdů Prof. Ing. Pavla Tvrdíka, CSc.
2. Shen, J.P., Lipasti, M.H.: Modern Processor Design : Fundamentals of Superscalar Processors, First Edition, New York, McGraw-Hill Inc., 2005
3. https://www.cs.utexas.edu/~pingali/CS395T/2009fa/lectures/mesi.pdf
4. D.E.Culler, J.P. Singh,A.Gupta: Parallel Computer Architecture: A HW/SW Approach,Morgan Kaufmann Publishers, 1998.
5. Einar Rustad: Numascale. Coherent HyperTransport Enables the Return of the SMP
6. Intel Itanium Processor 9300 Series and 9500 Series - Datasheet http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/itanium-9300-9500-datasheet.pdf
7. Daniel Molka et al.: Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. https://tu-dresden.de/zih/forschung/ressourcen/dateien/abgeschlossene-projekte/benchit/2015_ICPP_authors_version.pdf?lang=de
8. Michael R. Marty: Cache Coherence Techniques for Multicore Processors, 2008.
9. Brian Railing: Synchronization: Basics. 15-213: Introduction to Computer Systems https://www.cs.cmu.edu/afs/cs/academic/class/15213-m16/www/lectures/24-sync-basic.pdf
10. http://www.cs.cmu.edu/afs/cs/academic/class/15418-s12/www/lectures/14_relaxedReview.pdf
11. Adve and Gharachorloo: Shared Memory Consistency Models, WRL Research Report.
12. Synchronizing Instructions for PowerPC™ Instruction Set Architecture http://cache.freescale.com/files/32bit/doc/app_note/AN2540.pdf