

# Chapter 7

## Introduction to Shared Memory Multiprocessors

Copyright © 2005-2008 Yan Solihin

*Copyright notice:*

No part of this publication may be reproduced, stored in a retrieval system, or transmitted by any means (electronic, mechanical, photocopying, recording, or otherwise) without the prior written permission of the author.

An exception is granted for academic lectures at universities and colleges, provided that the following text is included in such copy: "Source: Yan Solihin, Fundamentals of Parallel Computer Architecture, 2008".

# Pros/Cons of Providing Shared Memory

- Shared memory = *an abstraction in which any address is the main memory is accessible using load/store instruction from any processor*
- Advantages of Shared Memory Machine:
  - Naturally support shared memory programs
  - Single OS image
  - Can fit larger problems:  $\text{total mem} = \text{mem}_n = n * \text{mem}_1$
  - Allow fine grain sharing
- Disadvantages of Shared Memory Machine:
  - Cost of providing shared memory abstraction
    - Cheap on small-scale multiprocessors (2-32 procs), multicore
    - Expensive on large-scale multiprocessors (> 64 procs)

# Objective of Chapter

- Shared memory abstraction requires hardware support
- Find out what hardware support is needed to construct a shared memory multiprocessor
- What is required to provide a shared memory abstraction?
  - Cache coherence (in systems with caches)
  - Memory consistency models
  - Synchronization support

# Cache Coherence Problem Illustration

- Illustration 1:
- “You (A) met a friend B yesterday and made an appointment to meet her at a place exactly one month from now. Later you found out that you have an emergency and cannot possibly meet on the agreed date. You want to change the appointment to three months from now.”
- Constraints
  - The only mode of communication is mail
  - In your mail, you can only write the new appointment date and nothing else
- Question 1: How can you ensure that you will meet B?
- Principle 1: When a processor performs a write operation on a memory address, the write must be propagated to all other processors
  - The **write propagation** principle

# Cache Coherence Problem

- Illustration 2:
- “You have placed a mail on the mailbox telling B to change the meeting month to 3 months from now. Unfortunately, you find out again that you cannot meet 3 months from now and need to change it to 2 months from now.”
- Question 2: How can you ensure that you will meet B two months from now?
- Principle 2: When there are two writes to the same memory address, the order in which the writes are seen by all processors must be the same.
  - The **write serialization** principle

## Will This Parallel Code Work Correctly?

```
sum = 0;
begin parallel
for (i=0; i<2; i++) {
    lock(id, myLock);
    sum = sum + a[i];
    unlock(id, myLock);
}
end parallel
Print sum;
```

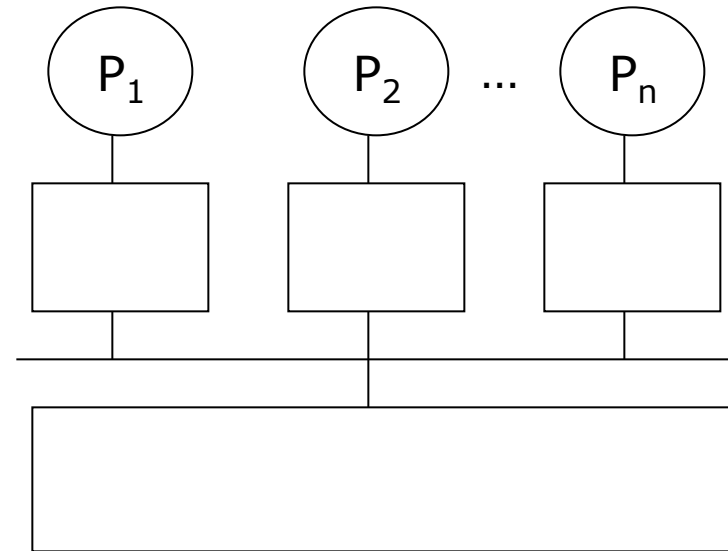
Suppose  $a[0] = 3$  and  $a[1] = 7$

...

Will it print  $\text{sum} = 10$ ?

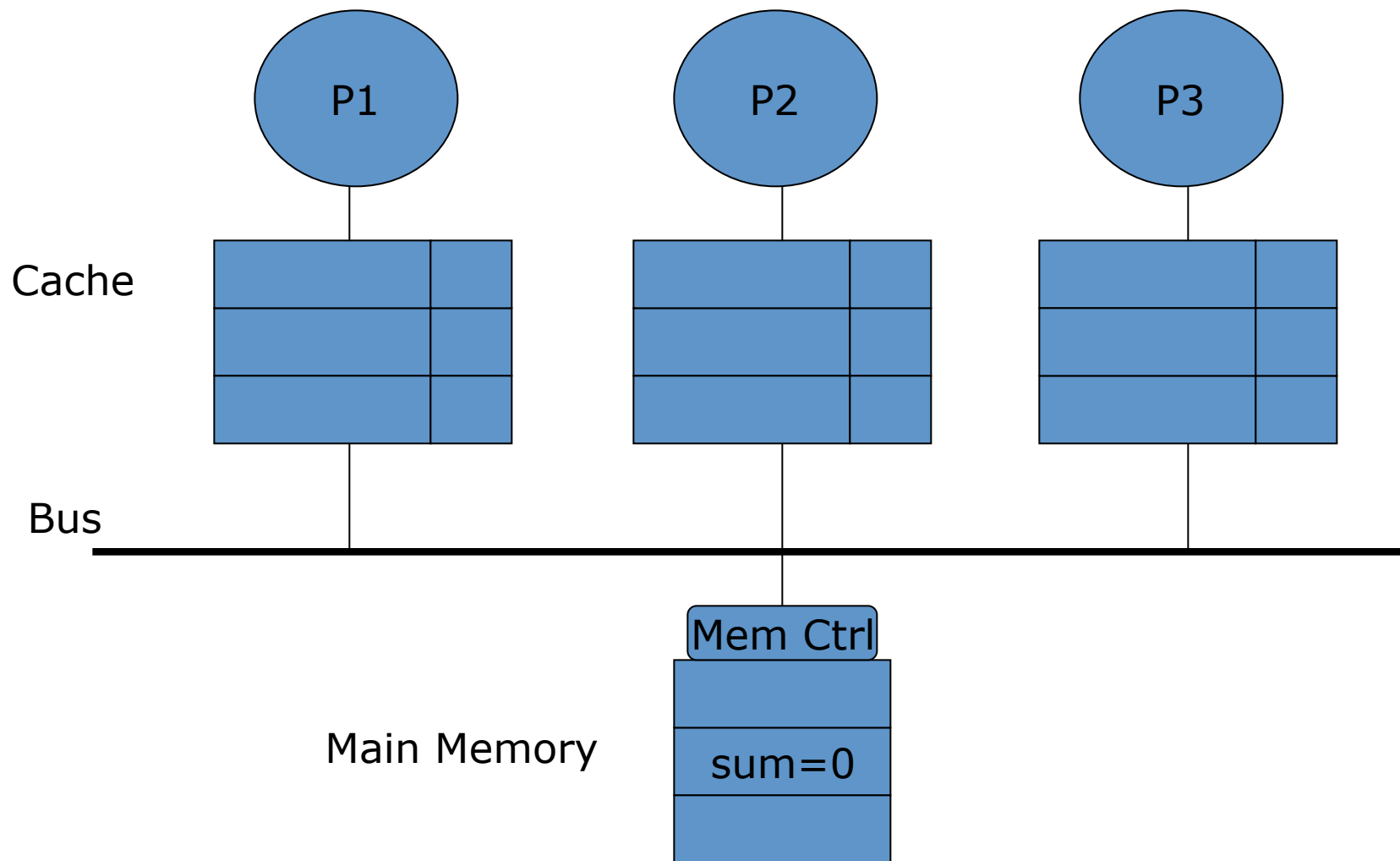
## Example 1: Cache Coherence Problem

```
sum = 0;  
begin parallel  
for (i=0; i<2; i++) {  
    lock(id, myLock);  
    sum = sum + a[i];  
    unlock(id, myLock);  
}  
end parallel  
Print sum;  
  
Suppose a[0] = 3 and a[1] = 7
```



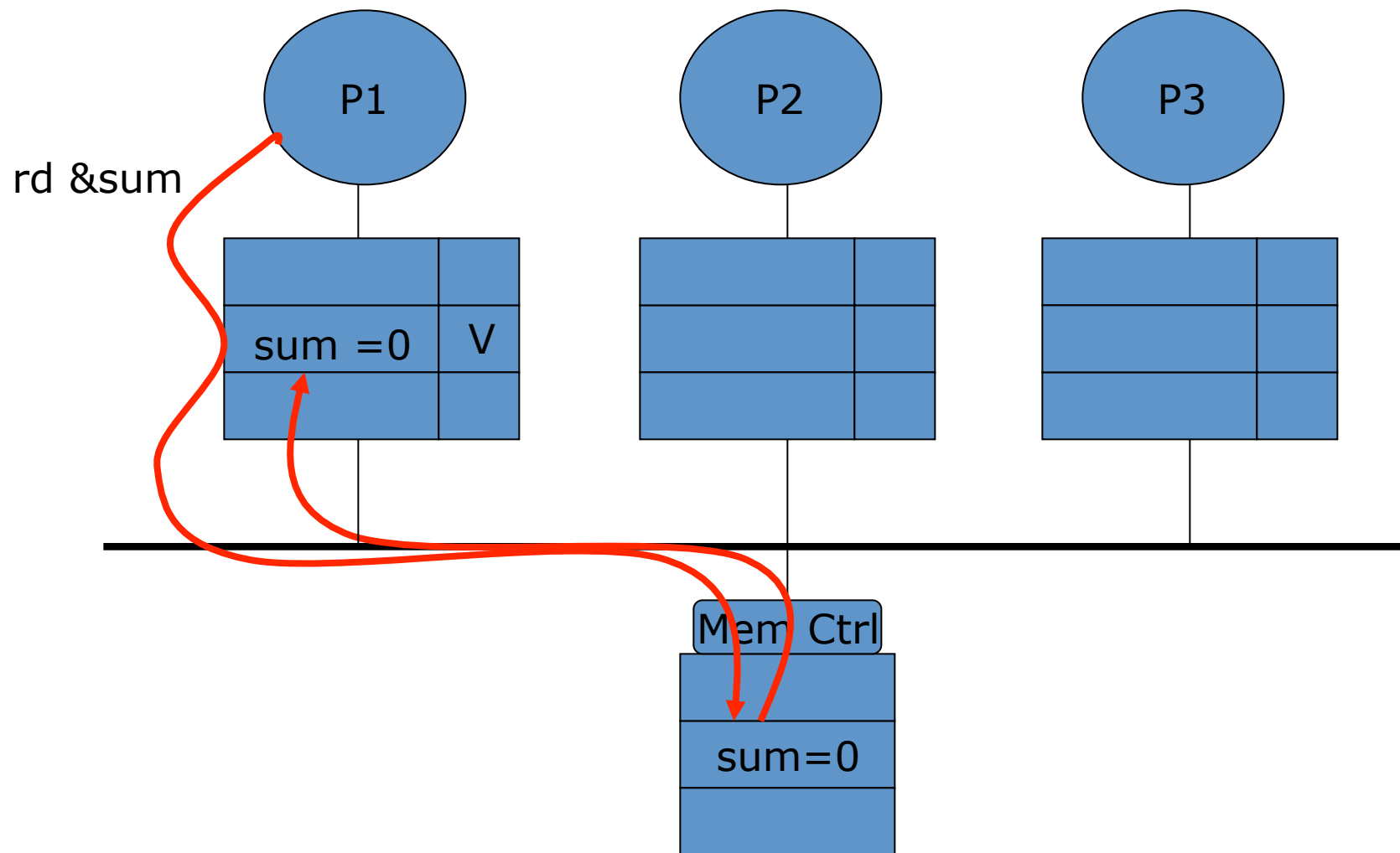
- Will it print sum = 10?

# Cache Coherence Problem Illustration

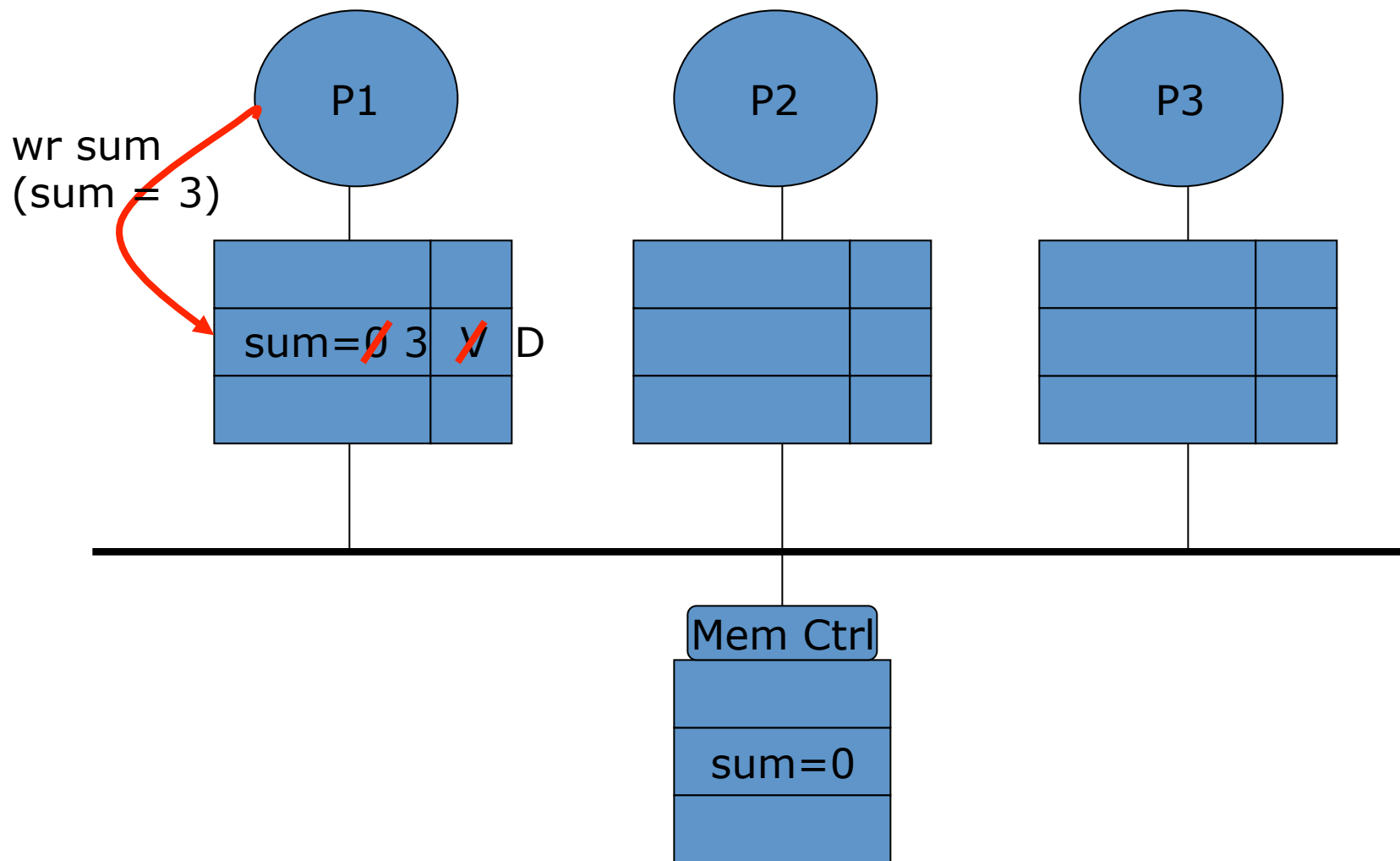




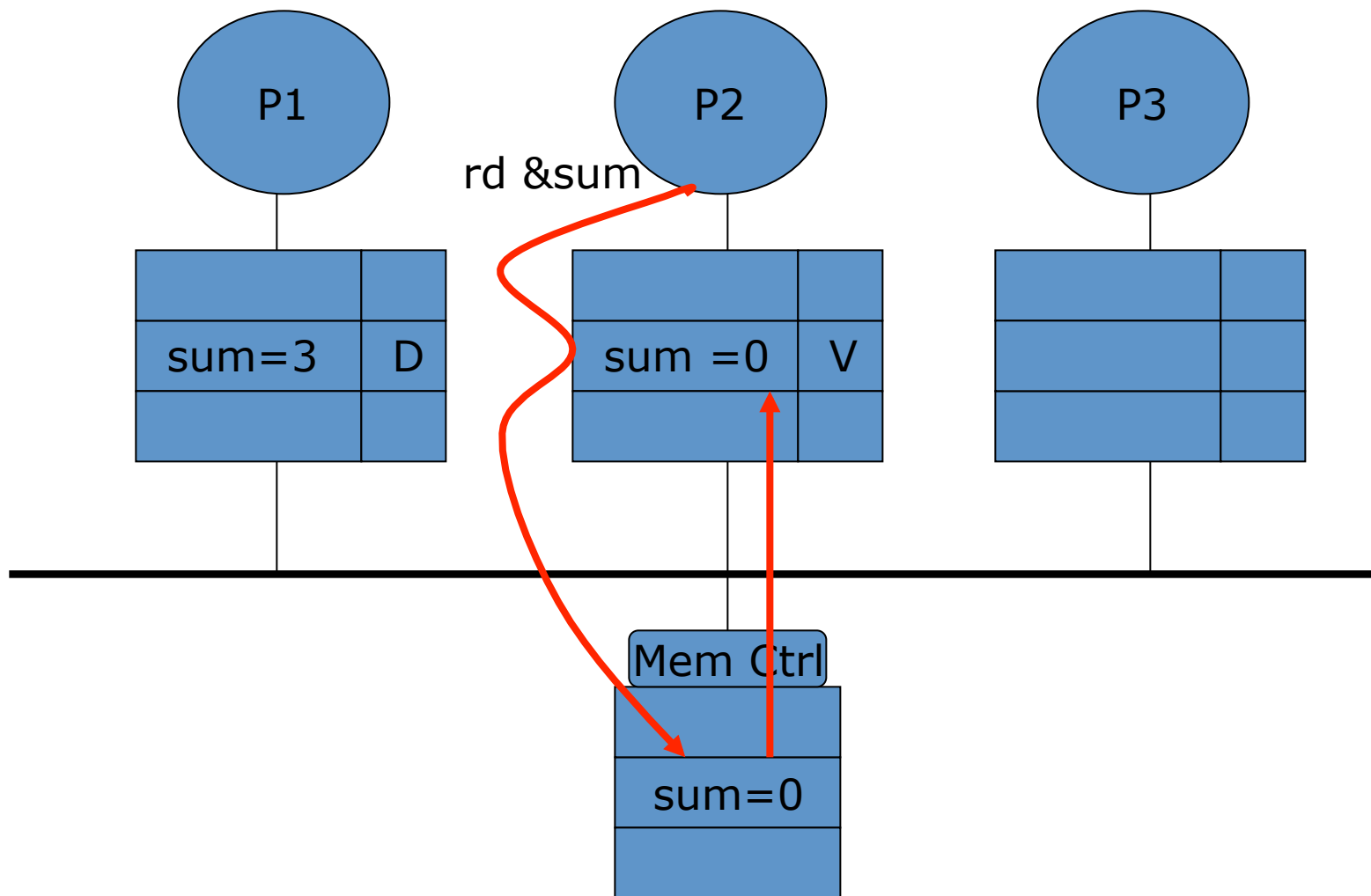
# Cache Coherence Problem Illustration



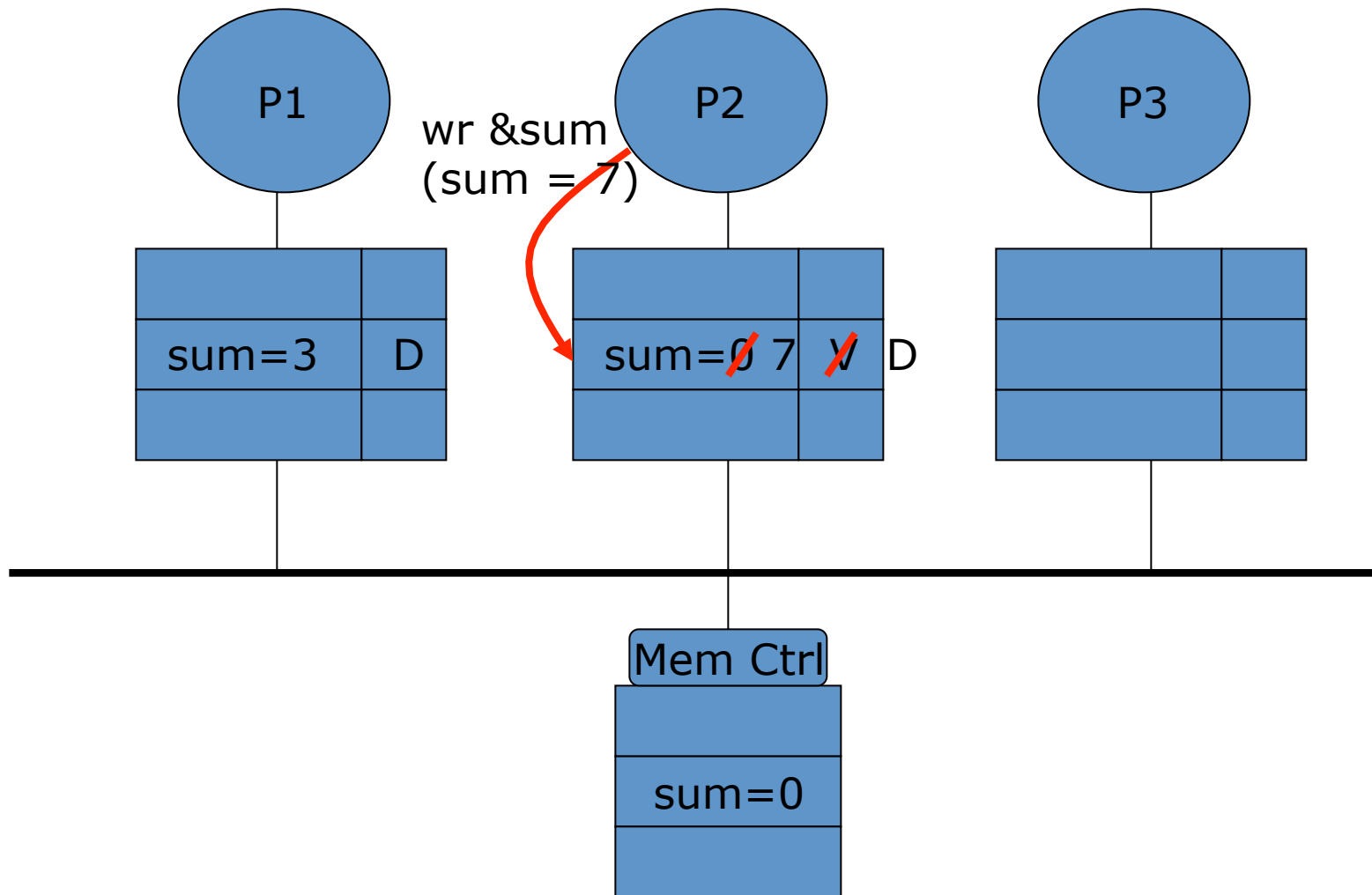
# Cache Coherence Problem Illustration



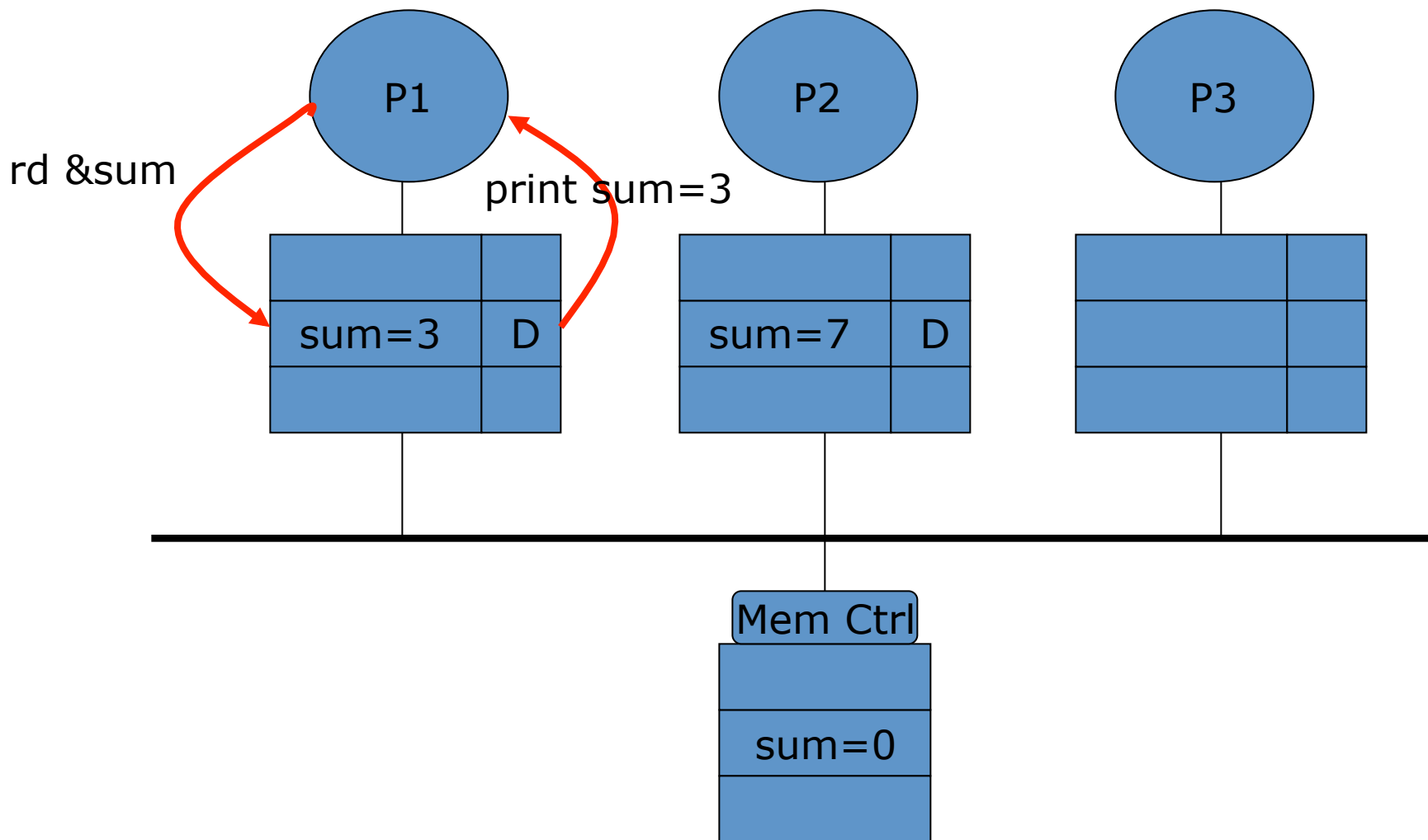
# Cache Coherence Problem Illustration



# Cache Coherence Problem Illustration



# Cache Coherence Problem Illustration



# Questions

- Do P1 and P2 see the same value of sum at the end?
- How is the problem affected by cache organization?  
For example:
  - What happens if we use write-through caches instead?
- What if we do not have caches, or sum is uncachable.  
Will it work?

# Main Observations

- Write propagation: values written in one cache must be propagated to other caches
- Write serialization: values written successively on the same data must be seen in the same order by all caches
- Need a protocol to ensure both properties
  - Called *cache coherence protocol*
- Cache coherence protocol can be implemented in hardware or software
  - Overheads of software implementations are often very high

# Memory Consistency Problem

- Cache coherence deals with propagation and ordering of memory accesses to a single datum
- Are there any needs for ordering memory accesses to multiple data items?



# Memory Consistency Illustration

P0:

```
S1: datum = 5;  
S2: datumIsRead = 1;
```

P1:

```
S3: while (!datumIsRead) {};  
S4: print datum
```

- Recall that point-to-point synchronization is common in DOACROSS and DOPIPE parallelism
- Correctness of execution is conditional upon P0 the ordering of S1 and S2, and the ordering of S3 and S4
  - What happens if S2 is executed before S1?
  - What happens if S4 is executed before S3?
- In multiprocessors, even if P0 executes S1 before S2, P1 does not necessarily see the same order!

# Global Ordering of Memory Accesses

- Full ordering: all memory accesses issued by a processor are executed in program order, and they are seen by all processors in that order
- However, full ordering is expensive
- Current solution: programmers' manual of what types of ordering are enforced by the processor, and what types are not
  - Referred to as *memory consistency model*
- Programmers' responsibility to write programs that conform to that model
  - Incorrect understanding is a source of non-trivial program errors
  - Contributes to non-portability of code

# How to Ensure Ordering

- Need cooperation of compiler and hardware
- Compiler may reorder S1 and S2
  - No data dependence between S1 and S2
  - So, it must be explicitly told not to reorder them
    - e.g. declaring `datum` and `datumIsReady` as `volatile`
  - This prevents compiler from using aggressive optimizations used in sequential programs
- The architecture may reorder S1 and S2
  - Write buffers, memory controller
  - Network delay for statement A
  - Many performance enhancing features for exploiting Instruction Level Parallelism affect ordering of instructions
  - Need to ensure ordering while allowing ILP to be exploited

# How to support critical section?

- How to guarantee mutual exclusion in a critical section?
- Let us see how a lock can be implemented in software

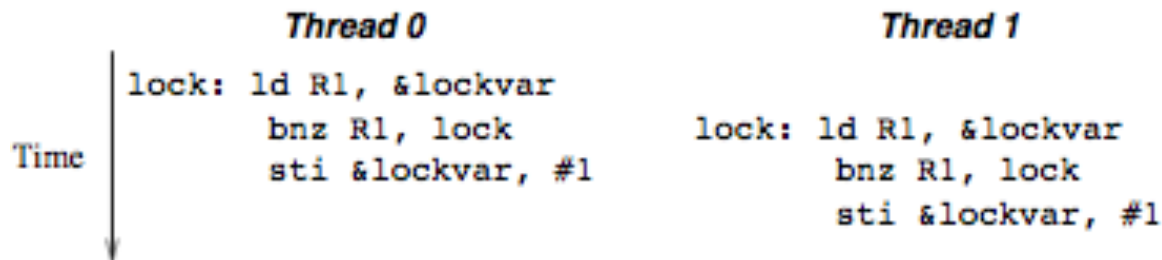
```
void lock (int *lockvar) {  
    while (*lockvar == 1) {} ;    // wait until released  
    *lockvar = 1;                // acquire lock  
}  
  
void unlock (int *lockvar) {  
    *lockvar = 0;  
}
```

In machine language, it looks like this:

```
lock: ld R1, &lockvar    // R1 = MEM[&lockvar]  
      bnz R1, lock       // jump to lock if R1 != 0  
      sti &lockvar, #1    // MEM[&lockvar] = 1  
      ret                // return to caller  
unlock: sti &lockvar, #0  // MEM[&lockvar] = 0  
       ret                // return to caller
```

# Problem with the Solution Just Shown

- Unfortunately, the solution is incorrect
- The sequence of `ld`, `bnz`, and `sti` are not atomic
  - Several threads may be executing it at the same time
- It allows several threads to enter the critical section simultaneously



# Peterson's algorithm

```
int turn;
int interested[n]; // initialized to 0


void lock (int process, int lvar) {      // process is 0 or 1
    int other = 1 - process;
    interested[process] = TRUE;
    turn = process;
    while (turn == process && interested[other] == TRUE) {} ;
}
// Post: turn != process or interested[other] == FALSE

void unlock (int process, int lvar) {
    interested[process] = FALSE;
}
```

- Exit from lock() happens only if:
  - interested[other] == FALSE: either the other process has not competed for the lock, or it has just called unlock()
  - turn != process: the other process is competing, has set the turn to itself, and will be blocked in the while() loop


# No Race

```
// Proc 0
interested[0] = TRUE;
turn = 0;
while (turn==0 && interested[1]==TRUE)
    {};
// since interested[1] == FALSE,
// Proc 0 enters critical section
```



```
// Proc 1
interested[1] = TRUE;
turn = 1;
while (turn==1 && interested[0]==TRUE)
    {};
// since turn==1 && interested[0]==TRUE
// Proc 1 waits in the loop until Proc 0
// releases the lock
```

```
// unlock
Interested[0] = FALSE;
```



```
// now can exit the loop and acquire the
// lock
```

# There is race

```
// Proc 0
interested[0] = TRUE;
turn = 0;
```


```
while (turn==0 && interested[1]==TRUE)
    {};
// since turn == 1,
// Proc 0 enters critical section
```

```
// unlock
Interested[0] = FALSE;
```

```
// Proc 1
interested[1] = TRUE;
```

```
turn = 1;
```

```
while (turn==1 && interested[0]==TRUE)
    {};
// since turn==1 && interested[0]==TRUE
// Proc 1 waits in the loop until Proc 0
// releases the lock
```



```
// now can exit the loop and acquire the
// lock
```



# Will Peterson's Algorithm Work?

- Yes, but correctness depends on the global order of

```
A: interested[process] = TRUE;
B: turn = process;
```

- So, it's related to the memory consistency problem

The diagram illustrates a memory consistency problem by showing two parallel code snippets for Proc 0 and Proc 1. Red arrows indicate the global order of memory operations, which may differ from the program order. The word "reordered" is written in bold on the right, with a red arrow pointing to the sequence of operations.

```
// Proc 0                                // Proc 1
interested[0] = TRUE;                      turn = 1;
                                           reordered
turn = 0;
while (turn==0 && interested[1]==TRUE)
    {};
// since interested[1] == FALSE,
// Proc 0 enters critical section

                                           interested[1] = TRUE;
                                           while (turn==1 && interested[0]==TRUE)
                                           {};
```

Additional code for Proc 1 shown below:

```
// since turn==0,
// Proc 1 enters critical section
```

# Performance Problem

- Assuming full program ordering is guaranteed:
  - Peterson's algorithm can achieve mutual exclusion
- However, it is very inefficient
  - For  $n$  threads, it needs an `interested[n]` array
  - Before a thread can enter the critical section, it must ensure that all elements of `interested[.]` array are `FALSE`
  - $O(n)$  number of loads and branches
- If non-atomicity is the problem, hardware can provide atomic sequence of load, modify, and store
- Typically in the form of an atomic instruction

# Conclusion

- To support shared memory abstraction in multiprocessors, we need to provide
  - Cache coherence
  - Memory Consistency Model
  - Support for synchronization