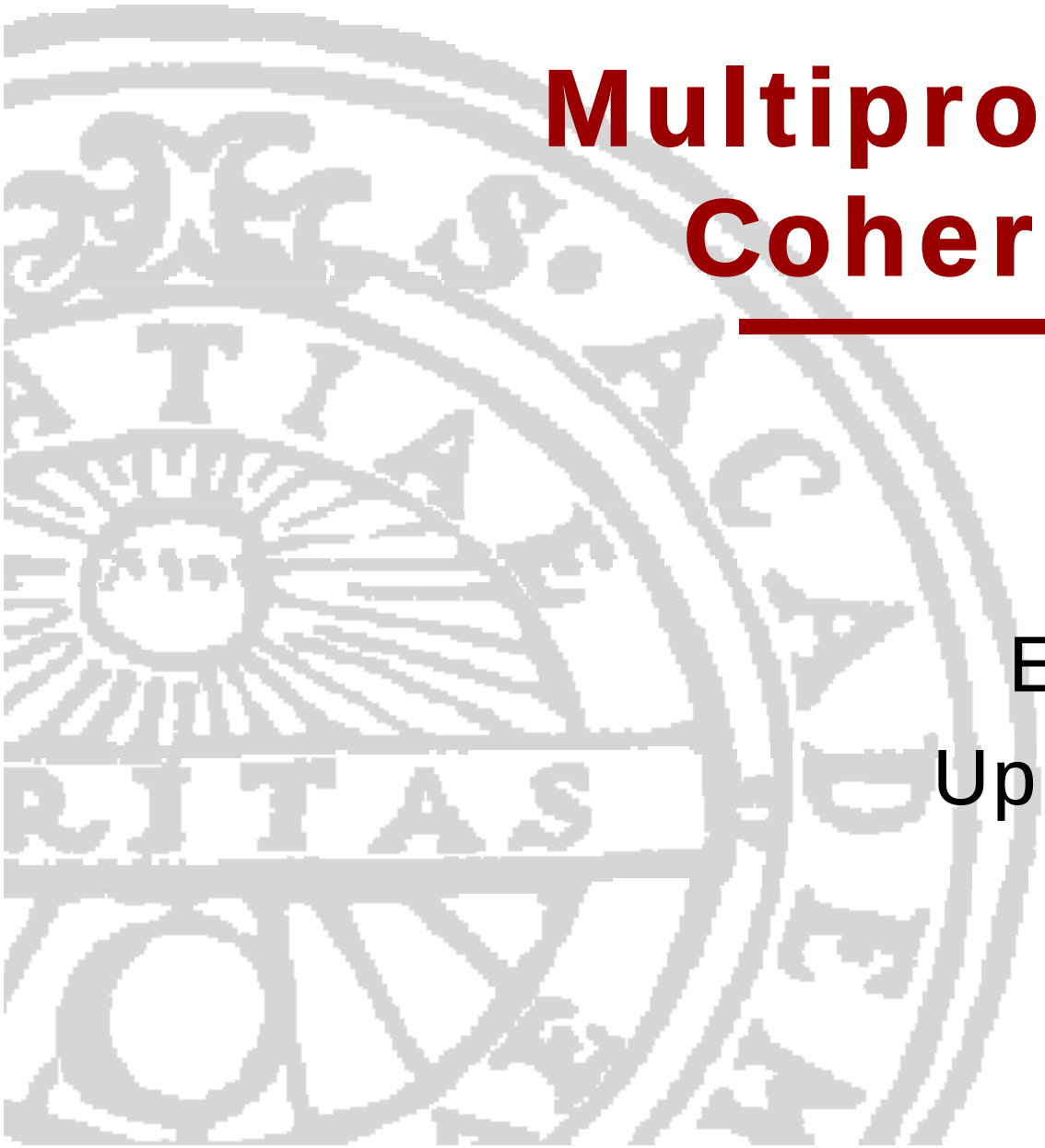


Multiprocessors and Coherent Memory

Erik Hagersten
Uppsala University

UPPSALA UNIVERSITET

DARK2 in a nutshell

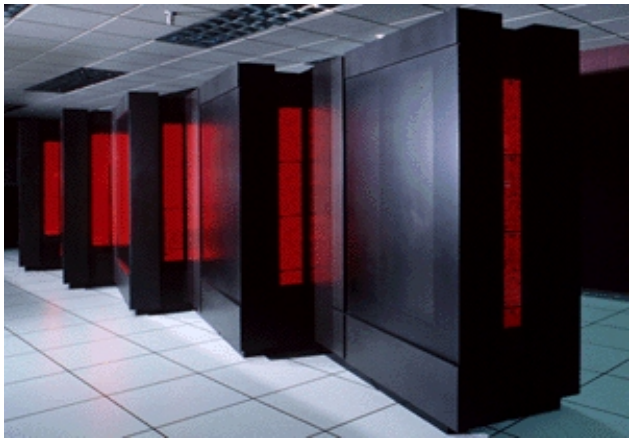
1. Memory Systems (caches, VM, DRAM, microbenchmarks, …)
2. Multiprocessors (TLP, coherence, memory models, interconnects, scalability, …)
3. CPUs (pipelines, ILP, scheduling, Superscalars, VLIWs, embedded, …)
4. Future: (physical limitations, TLP+ILP in the CPU,…)

DARK 2008

UPPSALA UNIVERSITET

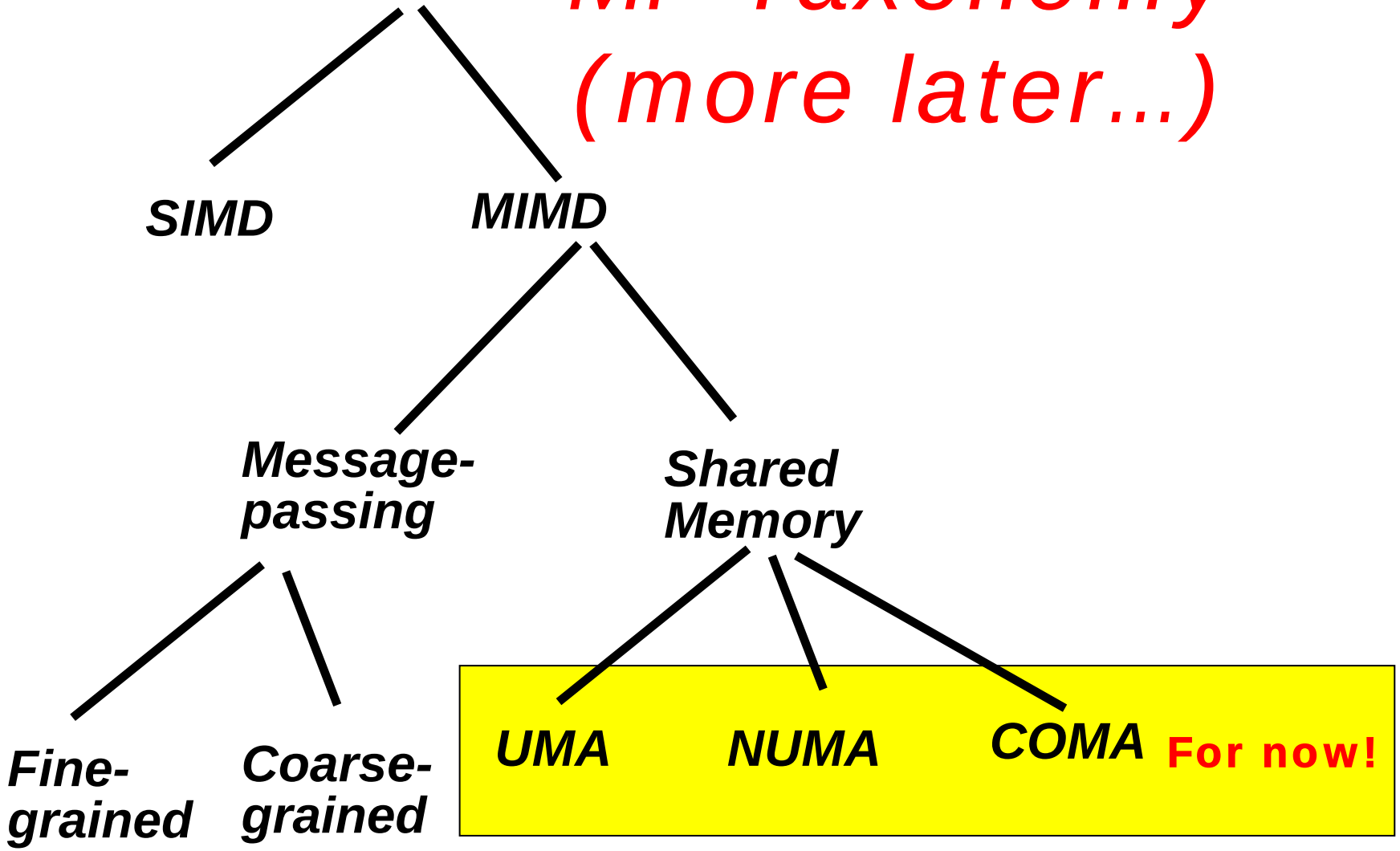
The era of the ”Rocket Science Supercomputers” 1980-1995

- The one with the most blinking lights wins
- The one with the niftiest language wins
- The more different the better!



DARK 2008

MP Taxonomy (more later…)

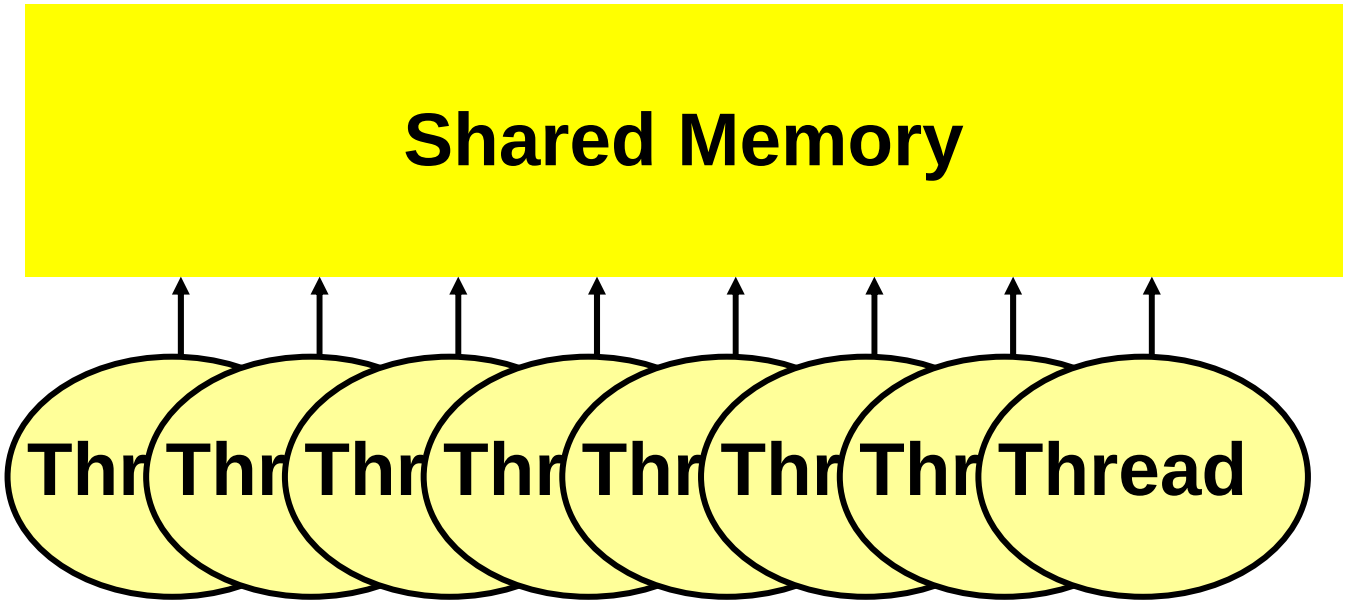


UPPSALA UNIVERSITET

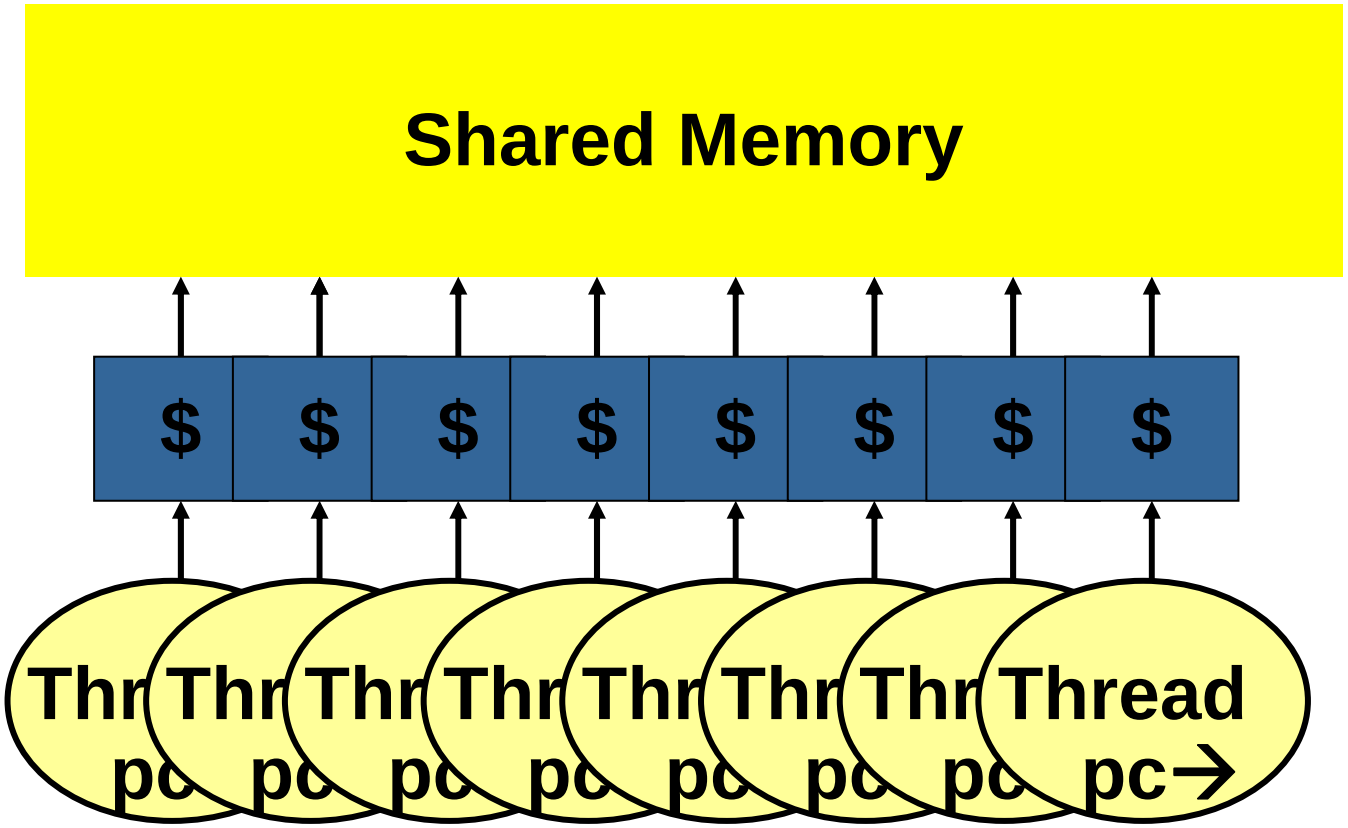
Models of parallelism

- Processes (**`fork`** or **`&`** in UNIX)
  - A parrallel execution, where each process has its own process state, e.g., memory mapping
- Threads (**`thread_chreate`** in POSIX)
  - Parallel threads of control inside a process
  - There are some thread-shared state, e.g., memory mappings.
- Sverker will tell you more…

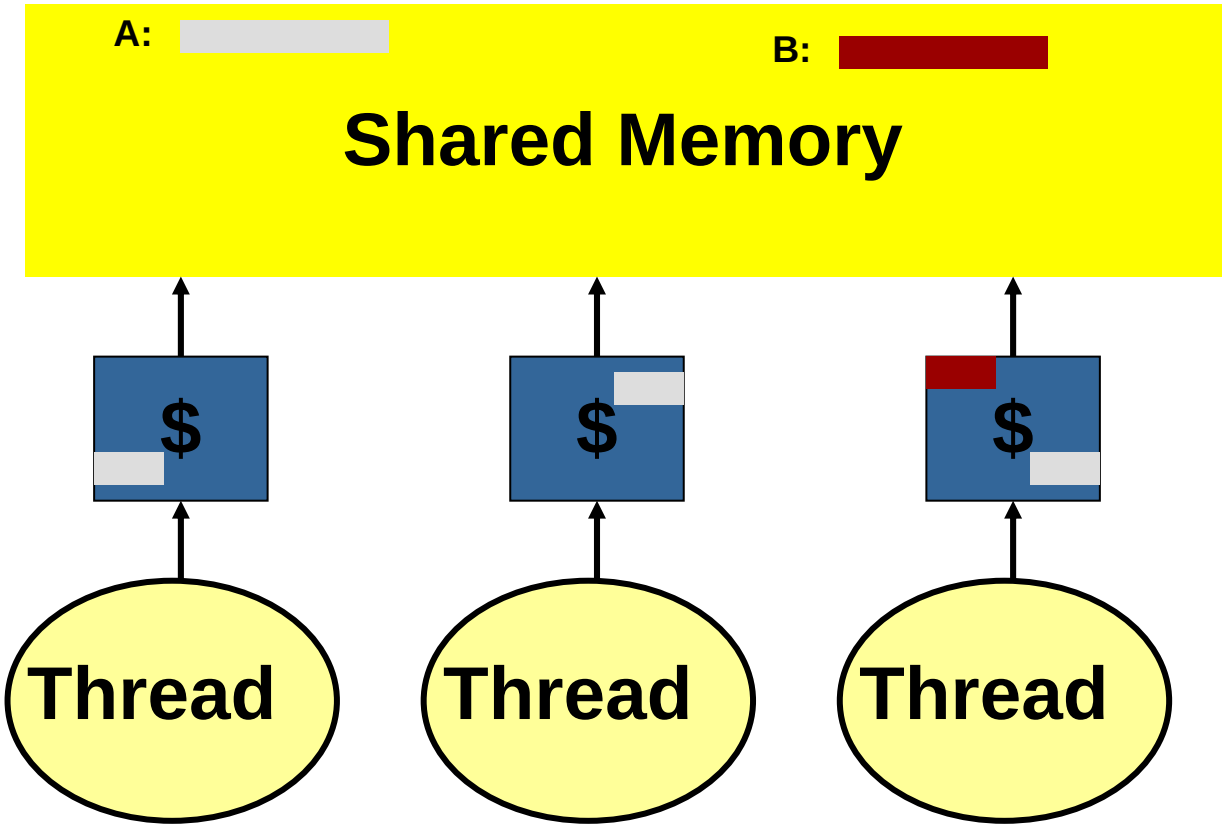
Programming Model:



Adding Caches: More Concurrency



Caches: Automatic Replication of Data

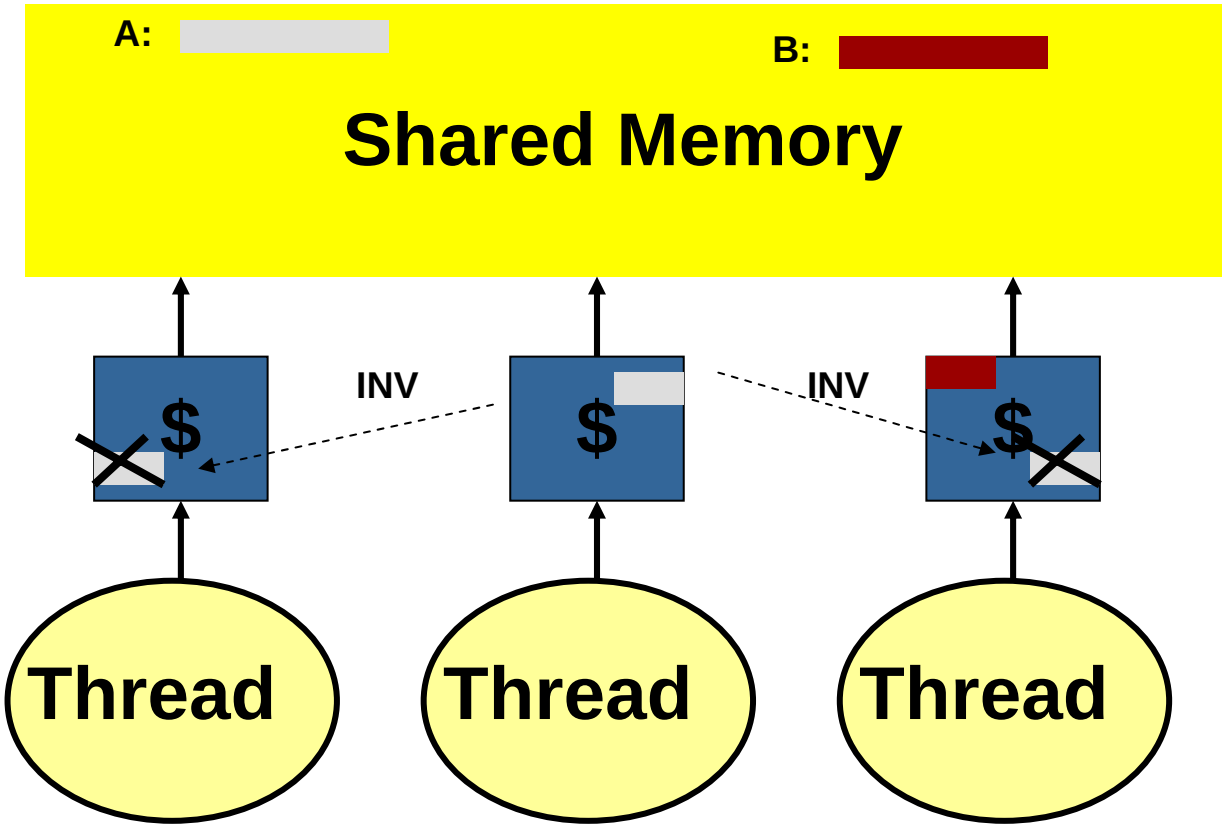


Read A
Read A
…
…
Read A

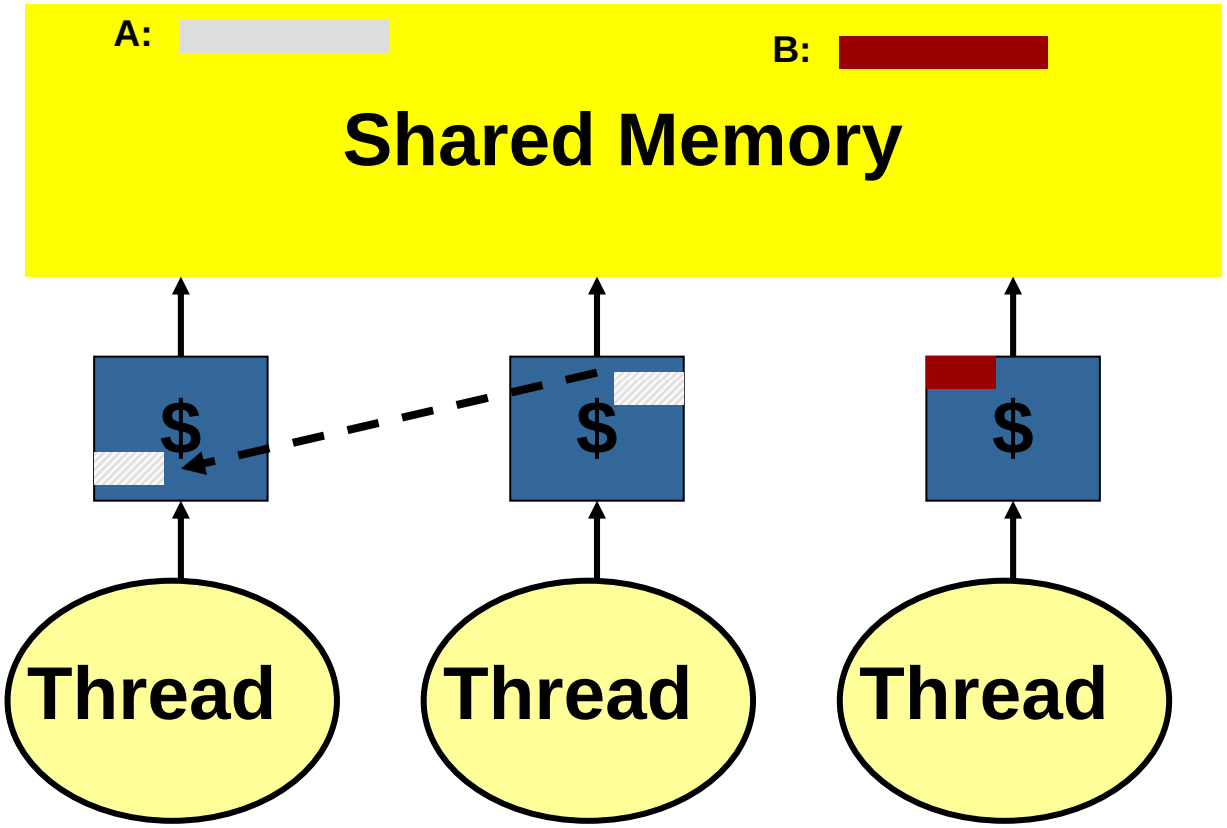
…
Read A
…

Read B
…
Read A

The Cache Coherent Memory System



The Cache Coherent 2



Summing up Coherence

There ca[illegible]y copies of a datu[illegible]one value

Too strong definition!

There is a single global order of value changes to each datum

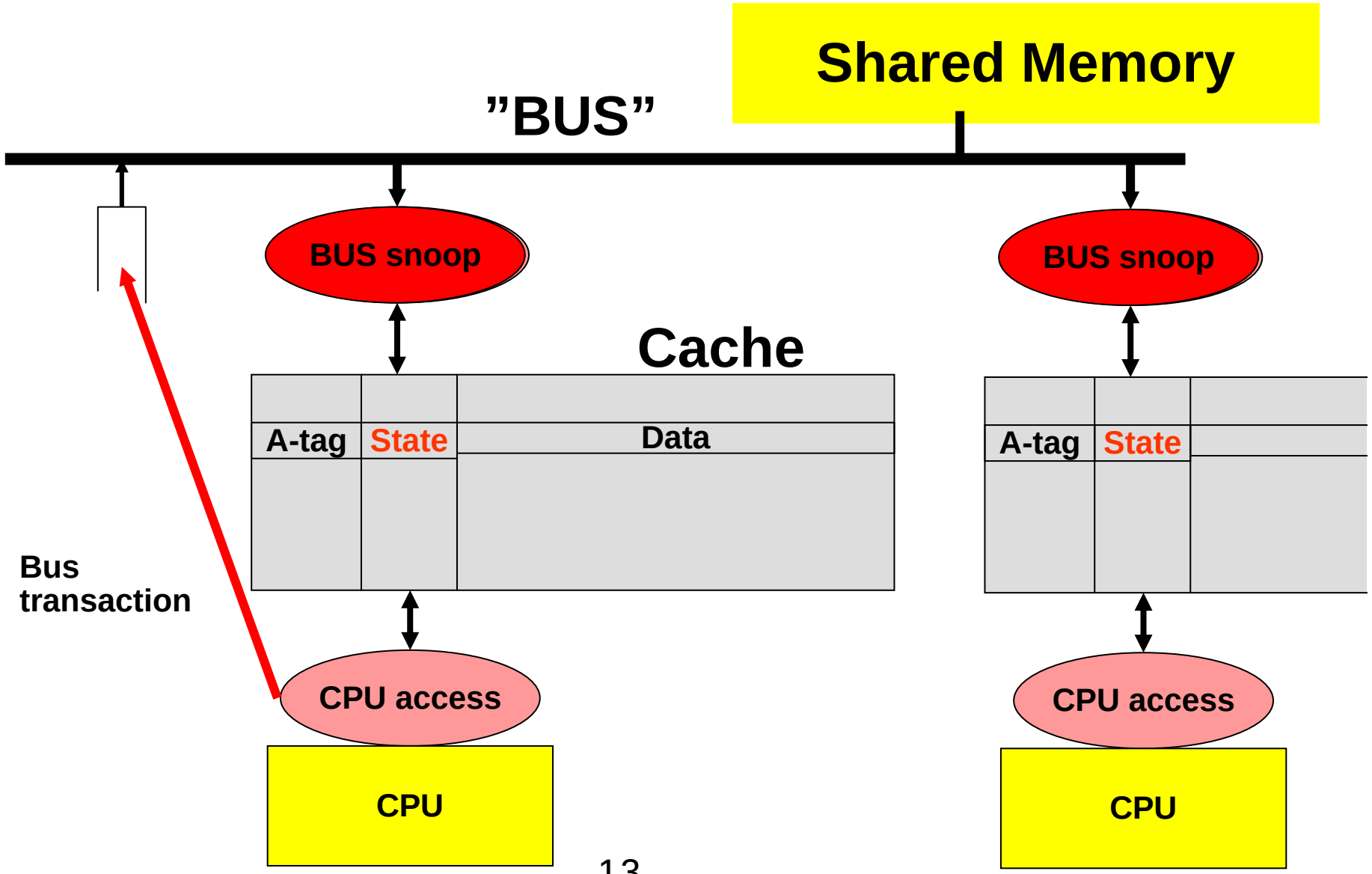
UPPSALA UNIVERSITET

Implementation options for memory coherence

- Two coherence options
  - Snoop-based (”broadcast”)
  - Directory-based (”point to point”)
- Different memory models
- Varying scalability

DARK 2008

Snoop-based Protocol Implementation



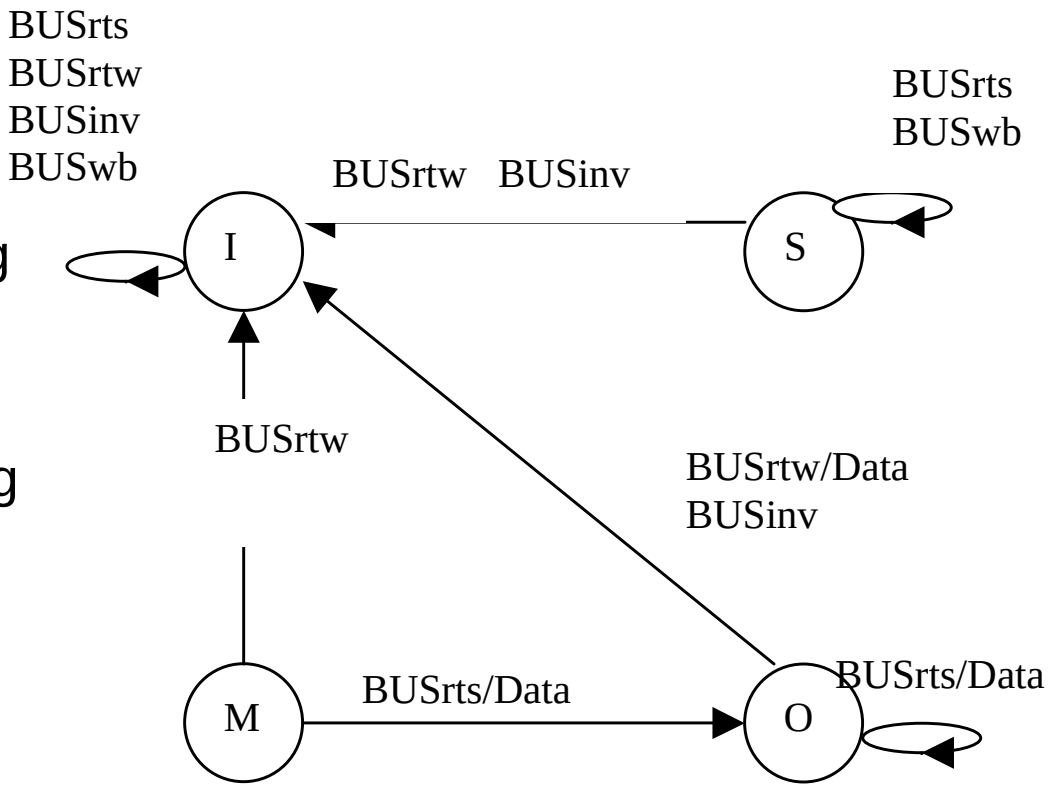
Example: Bus Snoop MOSI

BUSrts: ReadtoShare (reading the data with the intention to read it)

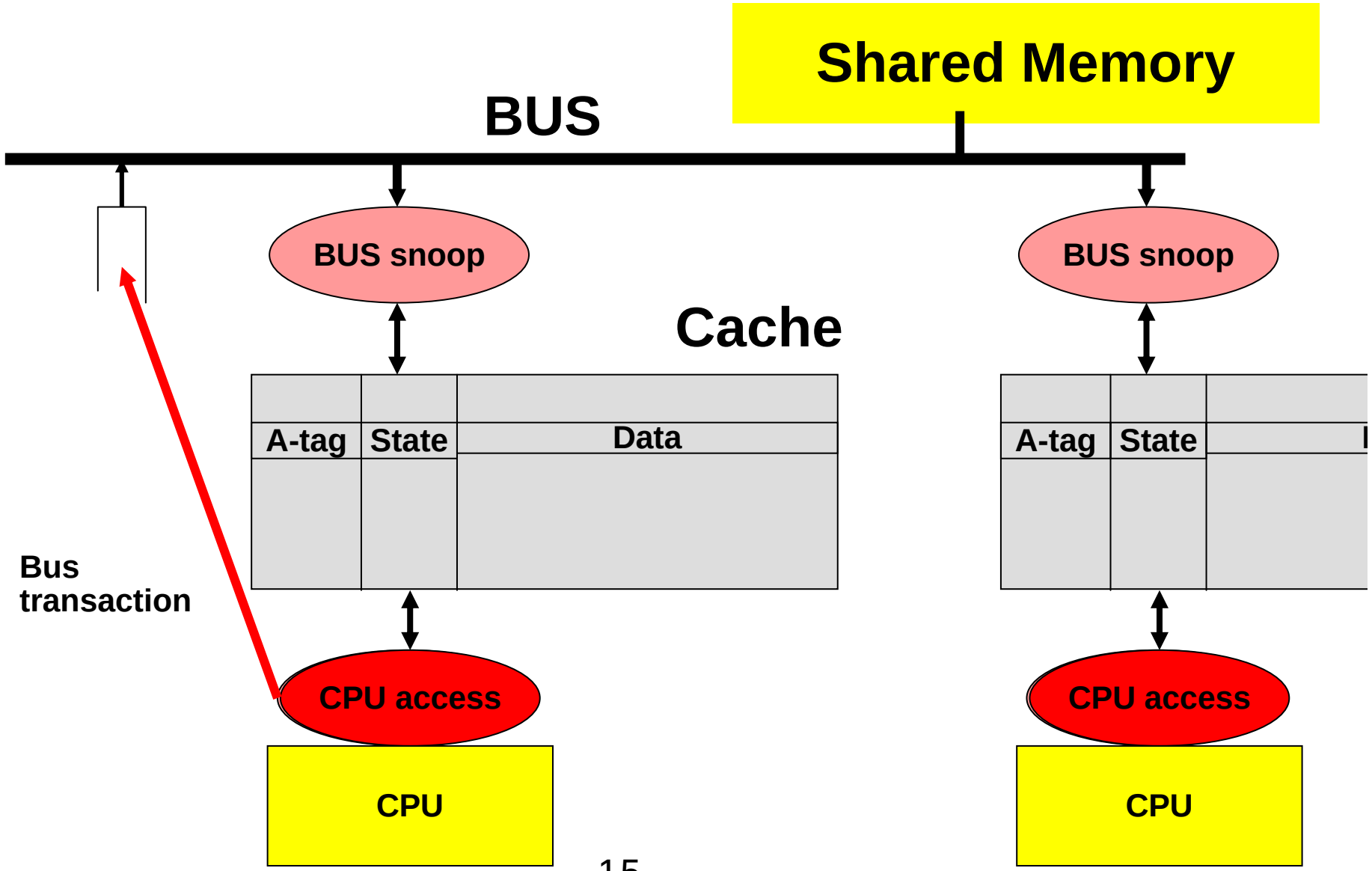
BUSrtw, ReadToWrite (reading the data with the intention to modify it)

BUSwb: Writing data back to memory

BUSinv: Invalidating other caches copies



Snoop-based Protocol Implementation

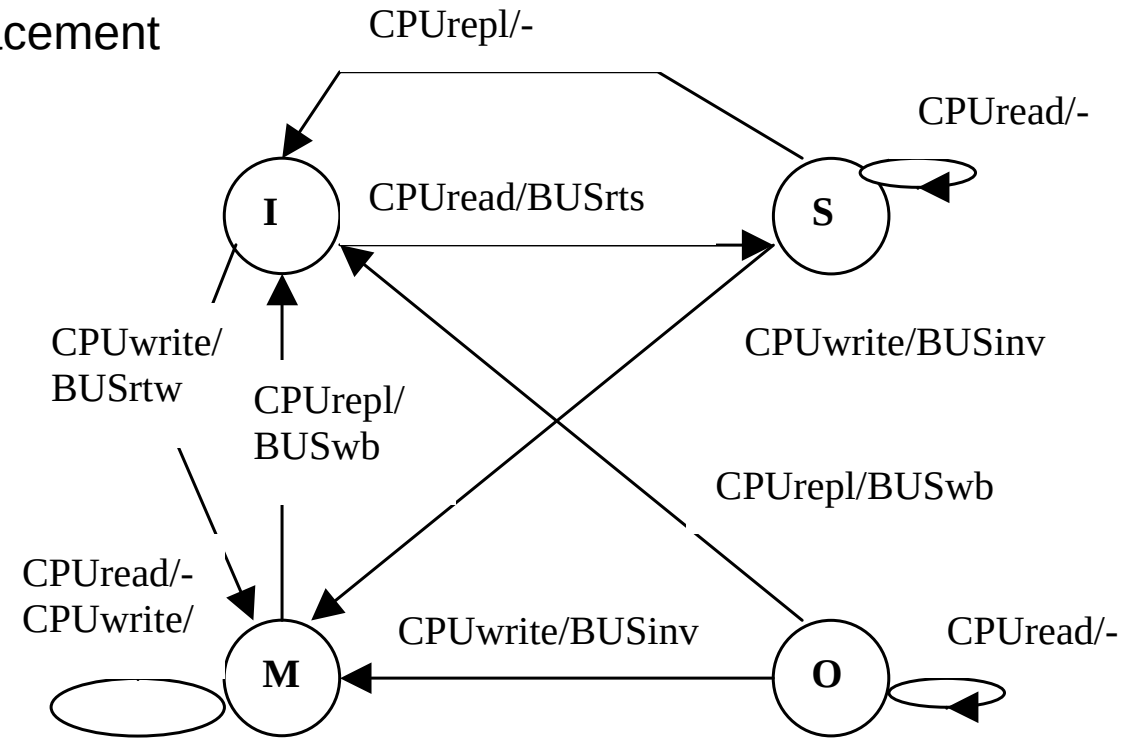


Example: CPU access MOSI

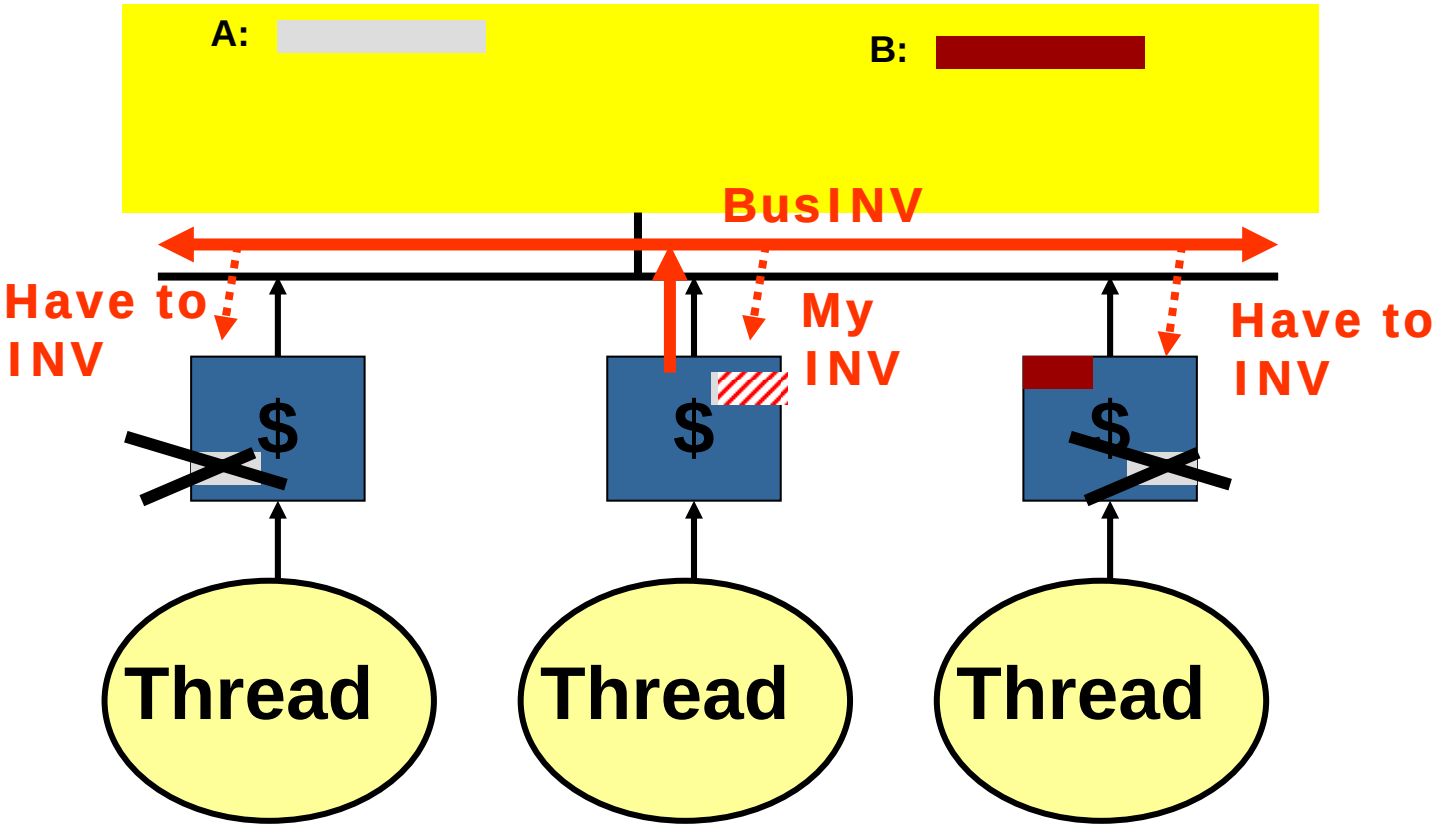
CPUwrite: Caused by a store miss

CPUread Caused by a loadmiss

CPUrepl: Caused by a replacement

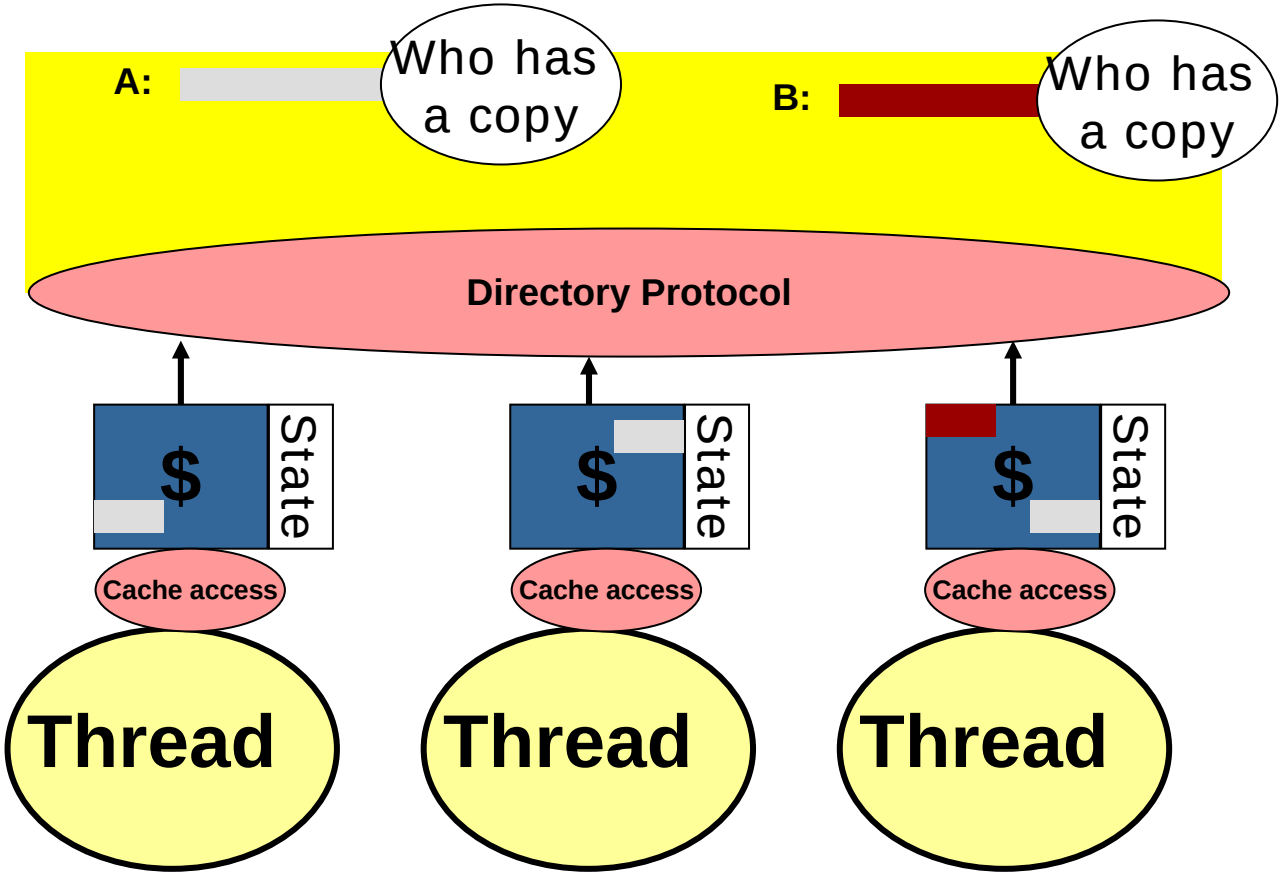


”Upgrade” in snoop-based

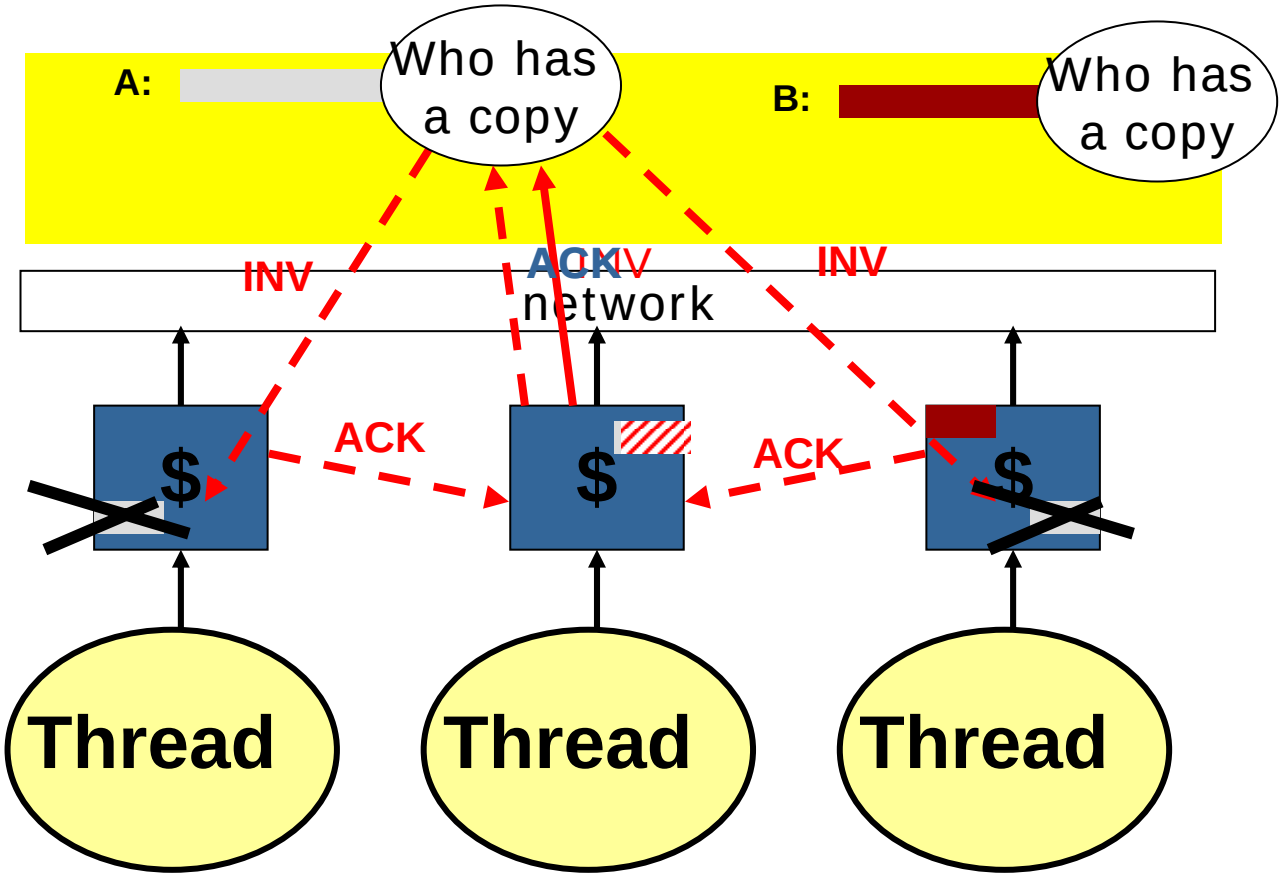


Read A	...	Read B
Read A	Read A	...
...	...	Read A
...	**Write A**	

Directory-based coherence: per-cachline info in the memory

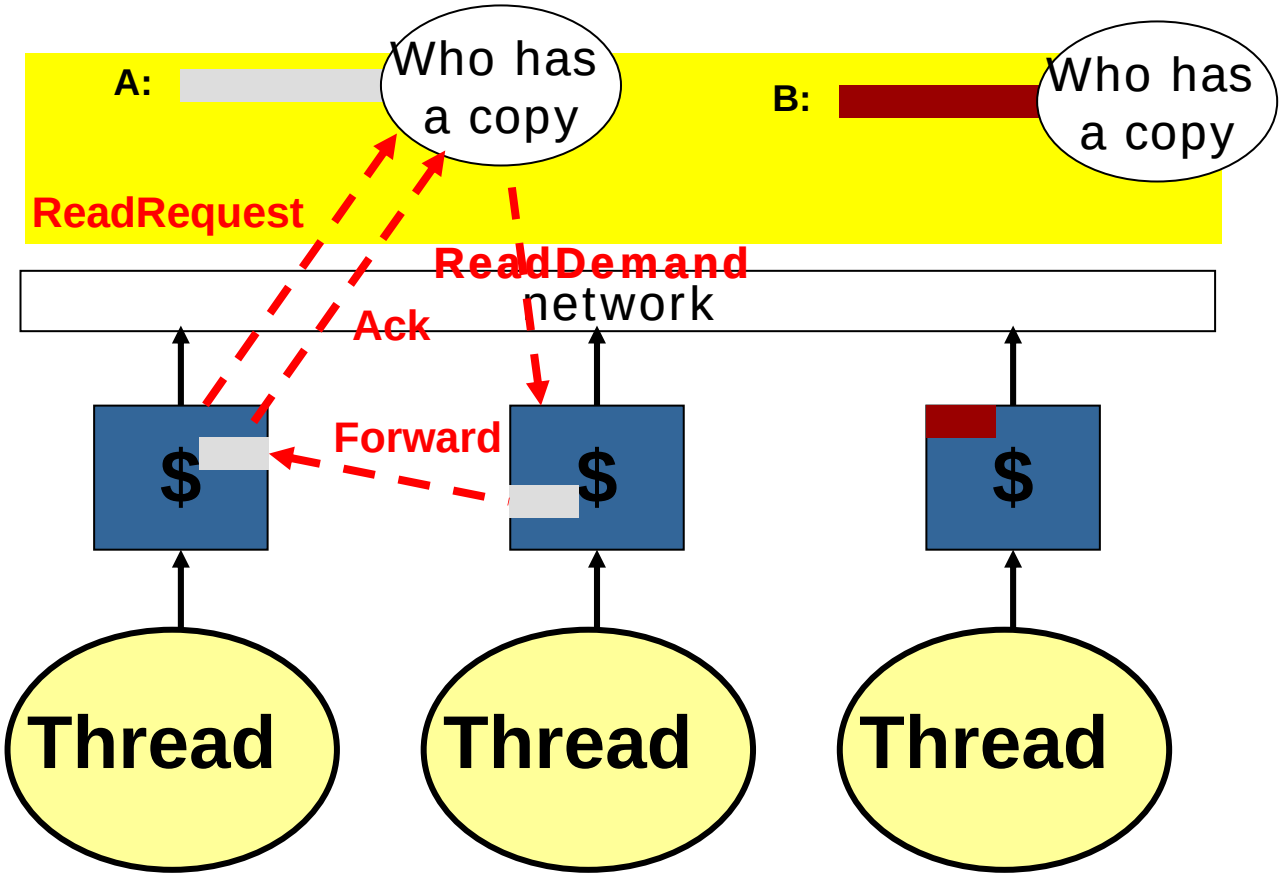


”Upgrade” in dir-based

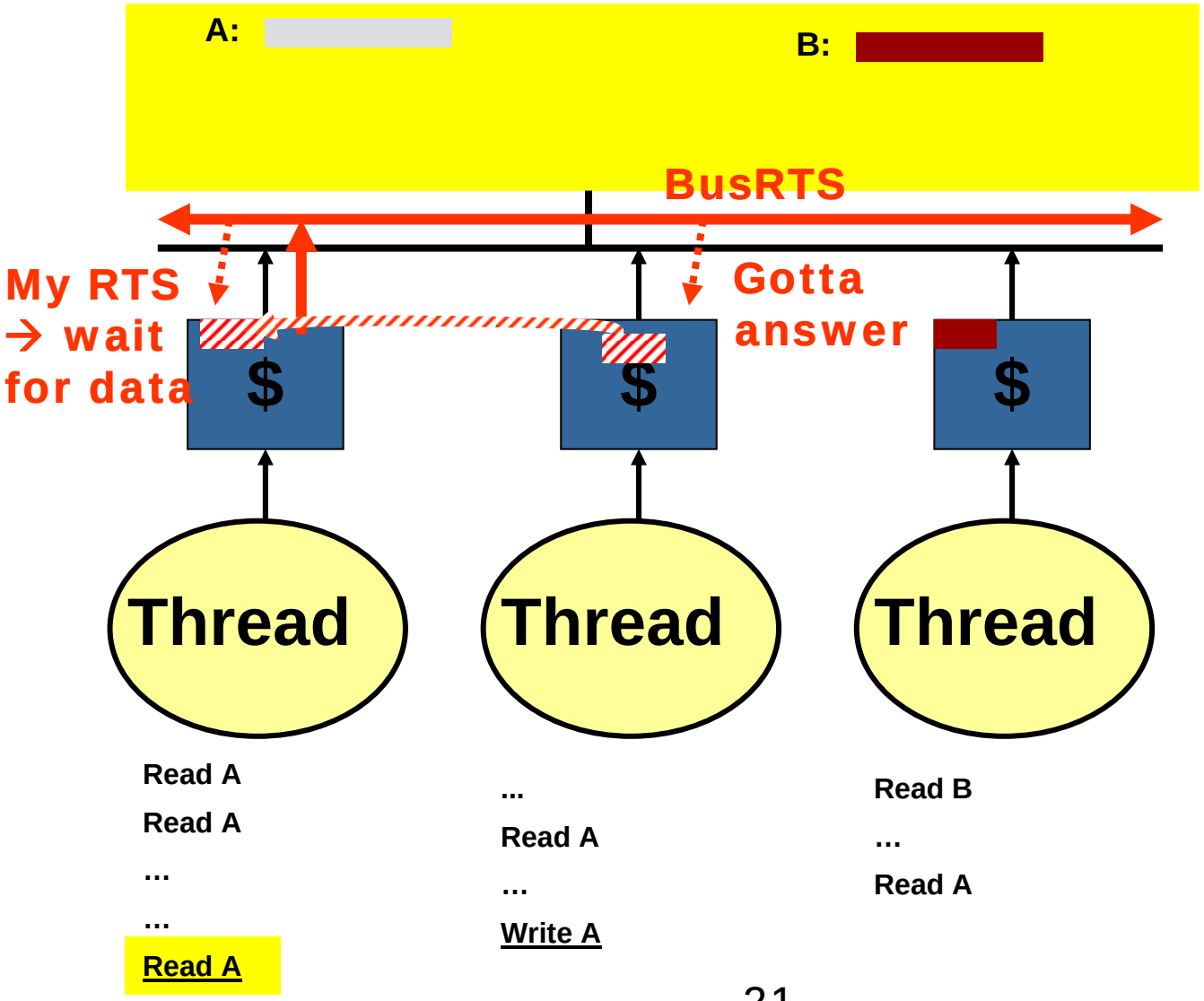


Read A	...	Read B
Read A	Read A	...
...	...	Read A
...	**Write A**	

Cache-to-cache in dir-based



Cache-to-cache in snoop-based



A New Kind of Cache Miss

- Capacity – too small cache
- Conflict – limited associativity
- Compulsory – accessing data the first time

- Communication (or ”Coherence”) [Jouppi]
  - Caused by downgrade (modified→shared)

    ”A store to data I had in state M, but now it’s in state S” ☹
  - Caused my invalidation (shared→invalid)

    ”A load to data I had in state S, but now it’s been invalidated” ☹

UPPSALA UNIVERSITET

Why snoop?

- A ”bus”: a serialization point helps coherence and memory ordering
- Upgrade is faster [producer/ consumer and migratory sharing]
- Cache-to-cache is **much** faster [i.e., communication…]
- Synchronization, a combination of both
- …but it is hard to scale the bandwidth☹

UPPSALA UNIVERSITET

Why directory-based

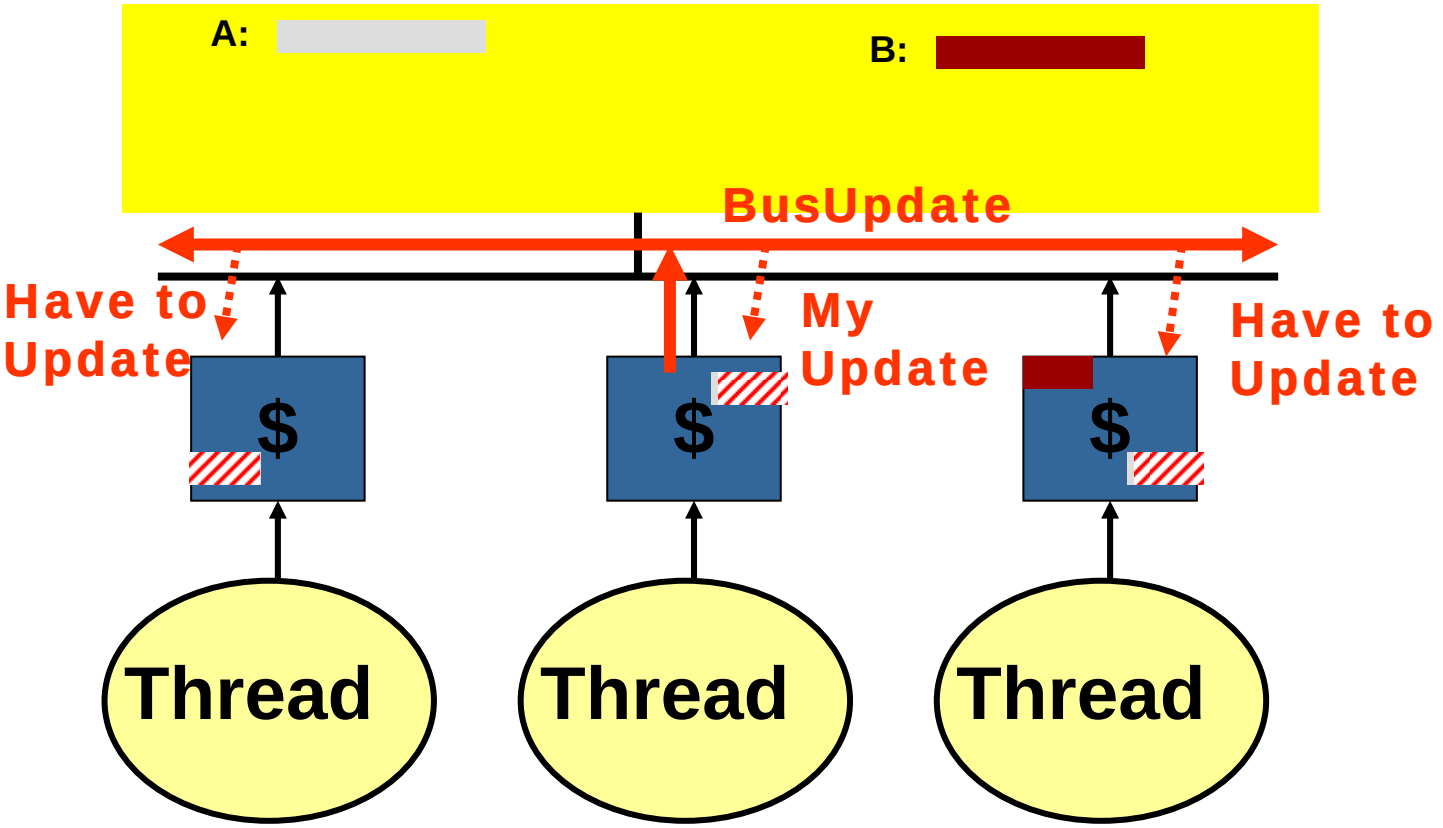
- P2P messages → **high** bandwidth
- Suits out-of-the-box coherence
- Note:
  - Dir-based can be used to build a uniform-memory architecture (UMA)
  - Bandwidth will be great!!
  - Memory latency will be OK
  - Cache-to-cache latency will not!

UPPSALA UNIVERSITET

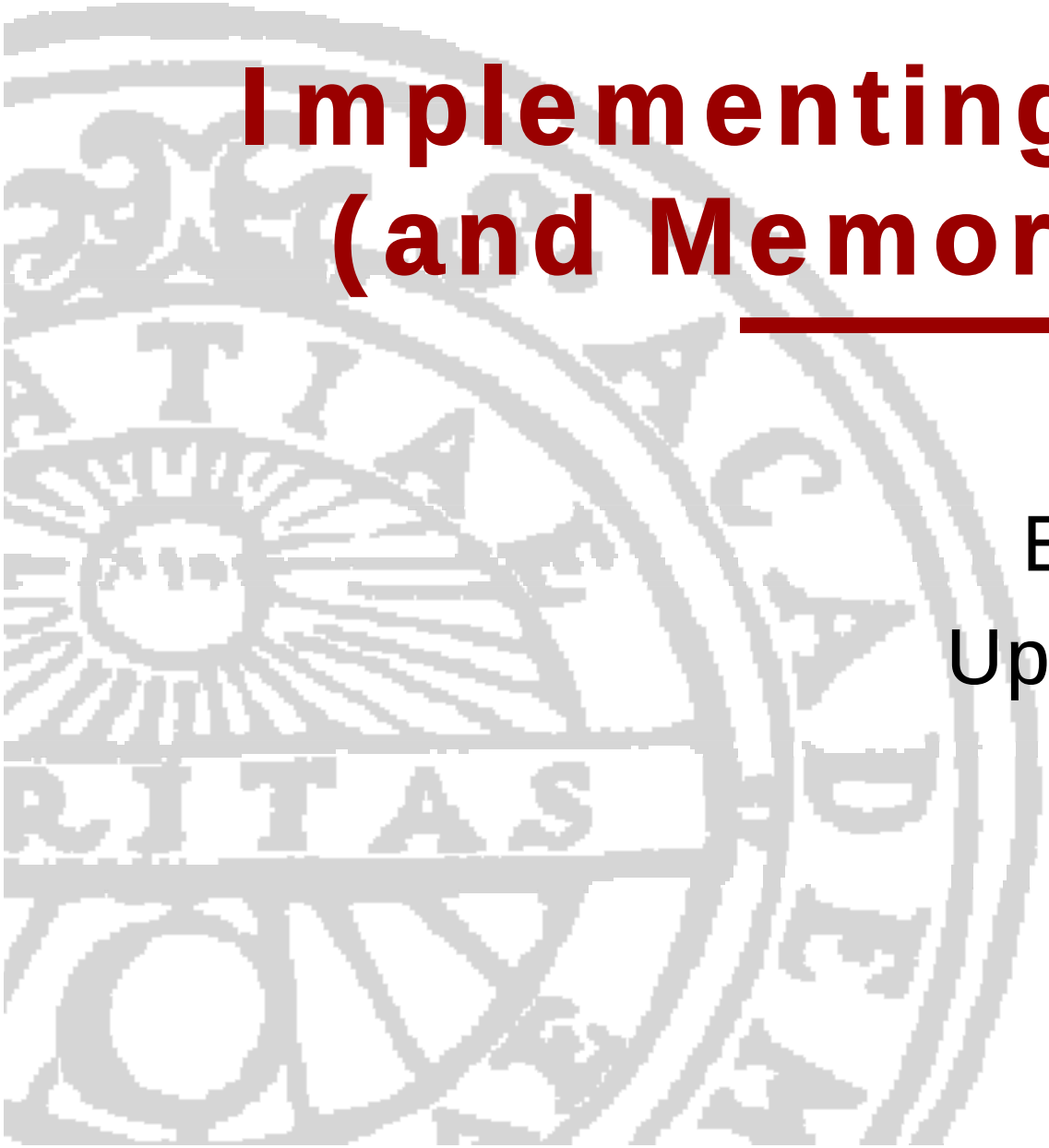
Update Instead of Invalidate?

- Write the new value to the other caches holding a shared copy (instead of invalidating...)
- Will avoid coherence misses
- Consumes a large amount of bandwidth
- Hard to implement strong coherence
- Few implementations: SPARCCenter2000, Xerox Dragon

Update in MOSI snoop-based



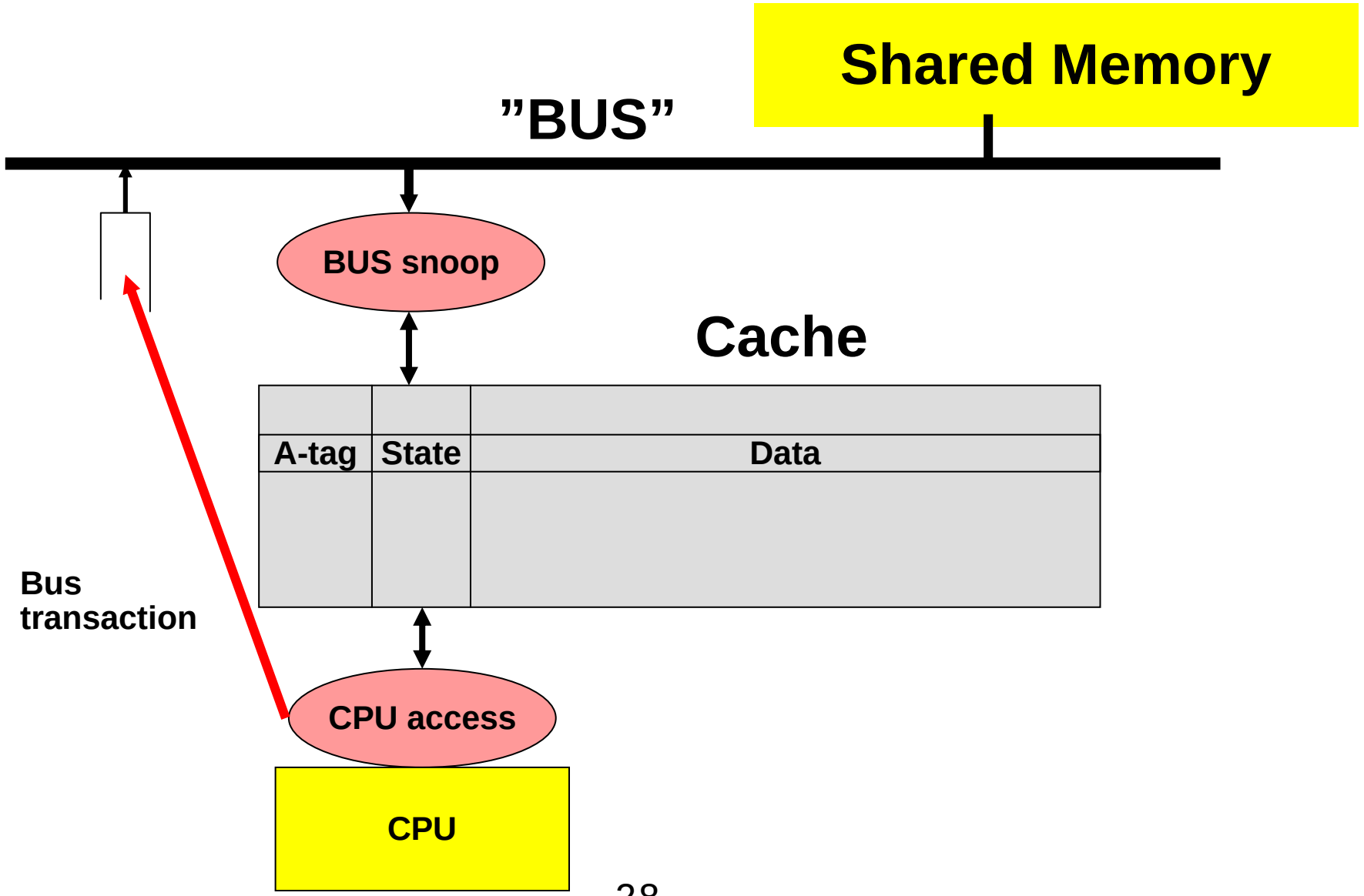
Read A	...	Read B
Read A	Read A	...
...	...	Read A
...	Write A	
Read A → HIT		



Implementing Coherence (and Memory Models…)

Erik Hagersten
Uppsala University
Sweden

Snoop-based Protocol Implementation



UPPSALA UNIVERSITET

Common Cache States

- M – Modified

  My dirty copy is the only cached copy
- E – Exclusive

  My clean copy is the only cached copy
- O – Owner

  I have a dirty copy, others may also have a copy
- S – Shared

  I have a clean copy, others may also have a copy
- I – Invalid

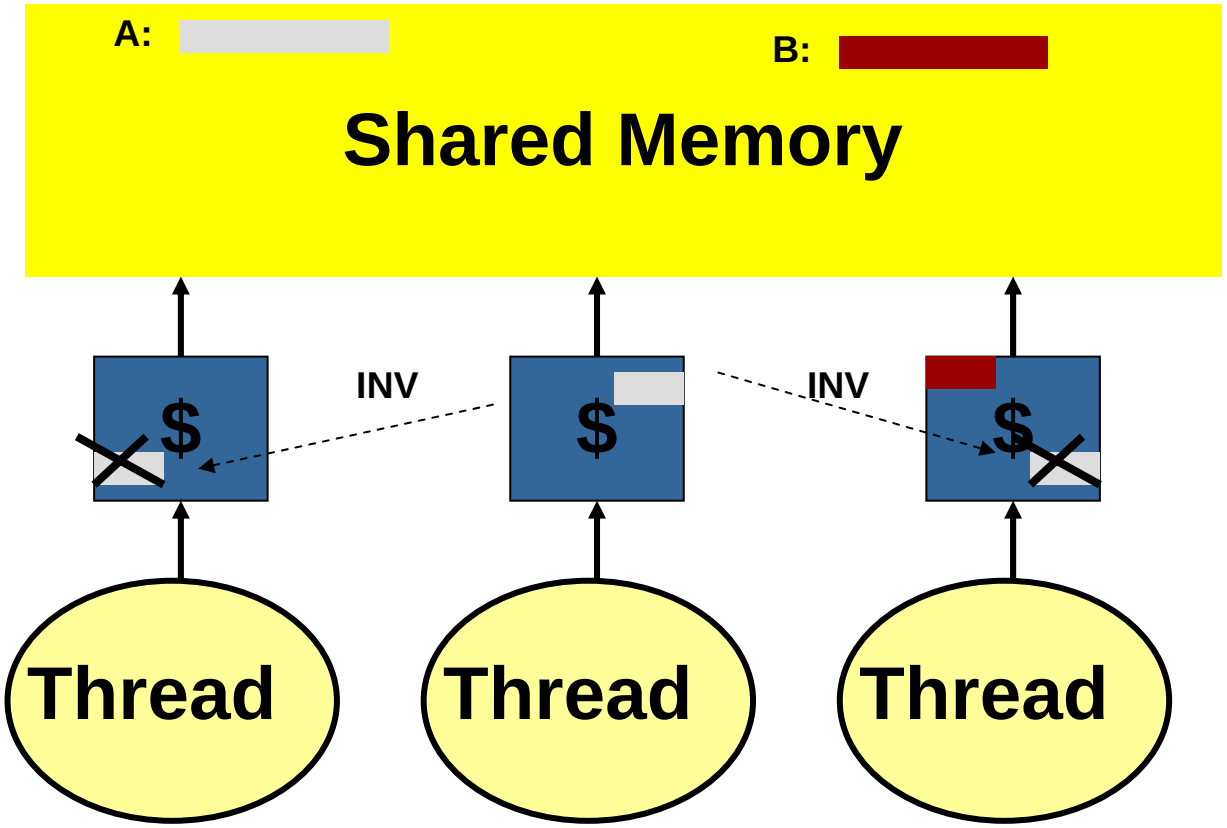
  I have no valid copy in my cache

UPPSALA UNIVERSITET

Some Coherence Alternative

- MSI
  - Writeback to memory on a cache2cache.
- MOSI
  - Leave one dirty copy in a cache on a cache2cache
- MOESI
  - The first reader will go to E and can later write cheaply

The Cache Coherent Memory System

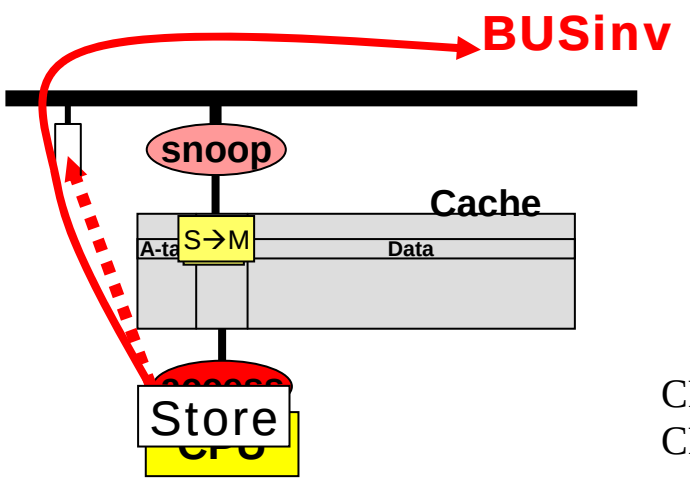


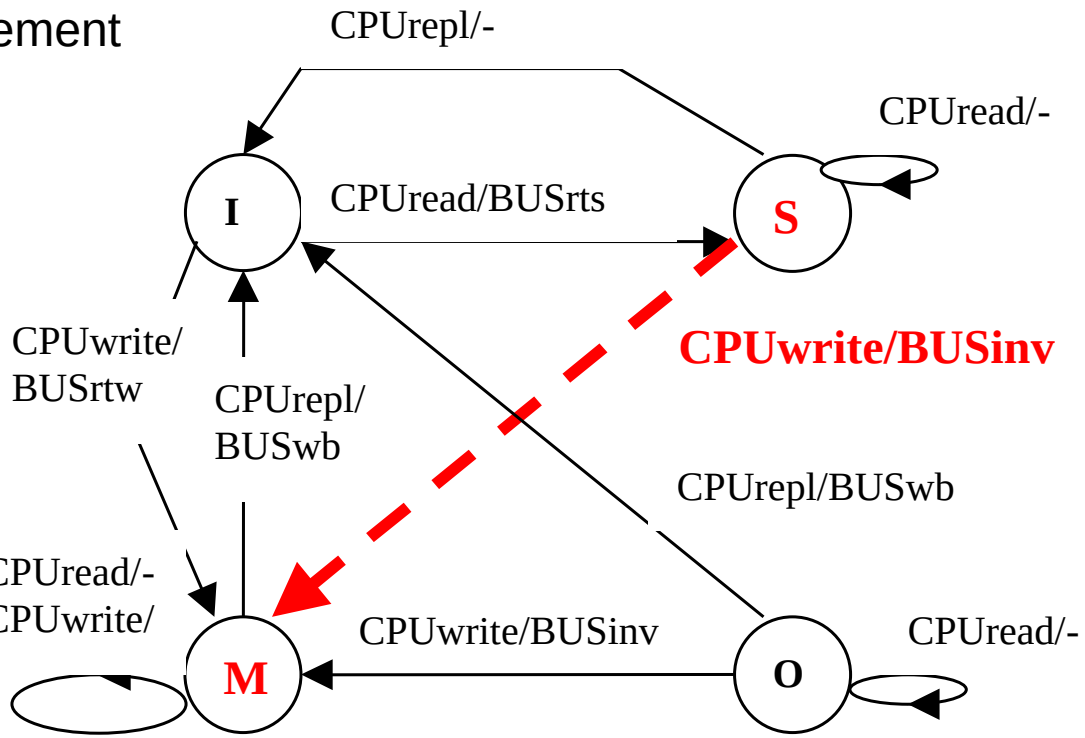
Upgrade – the requesting CPU

CPUwrite: Caused by a store miss

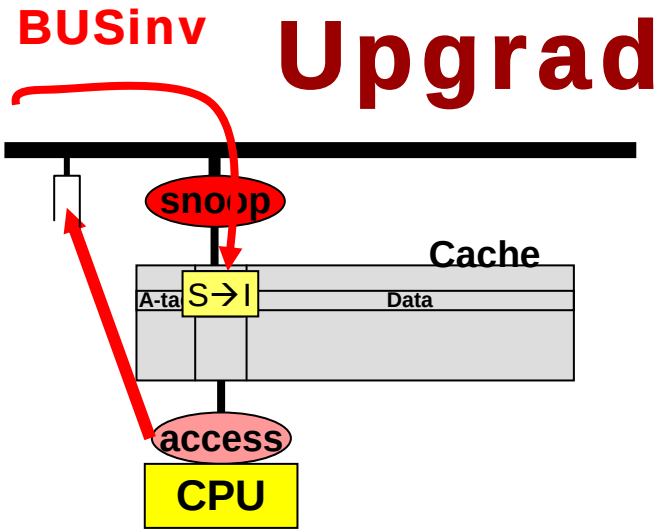
CPUread Caused by a loadmiss

CPUrepl: Caused by a replacement





Upgrade – the other CPUs

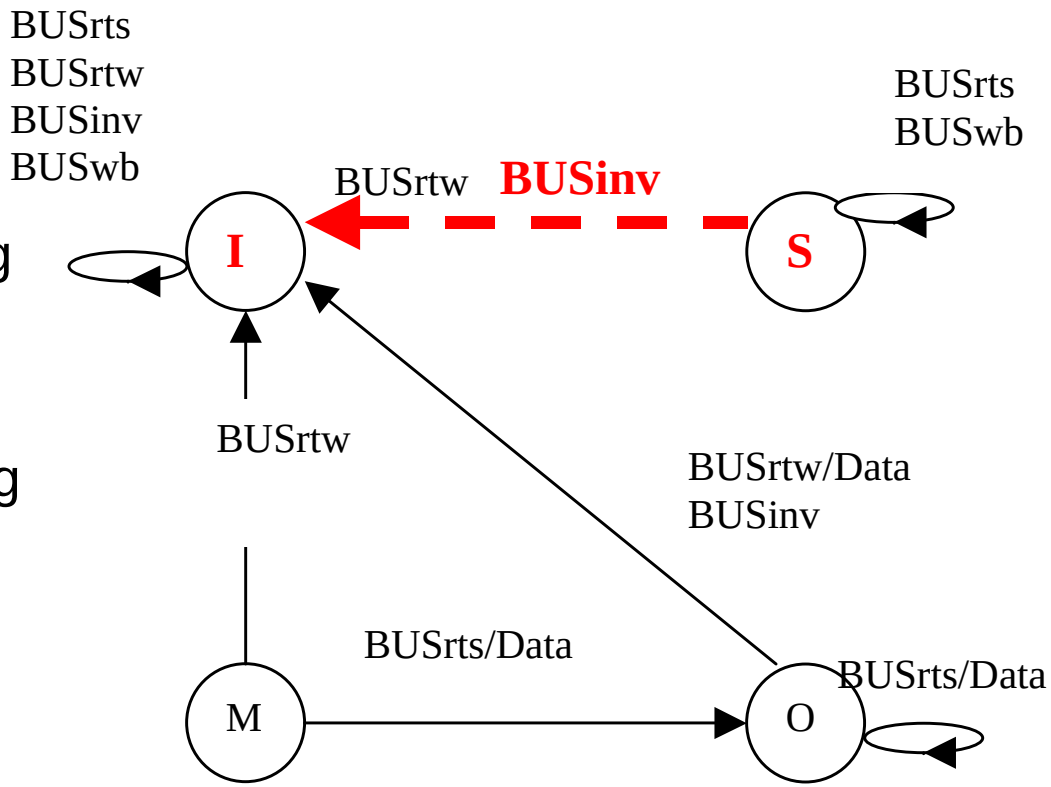


BUSrts: ReadtoShare (reading the data with the intention to read it)

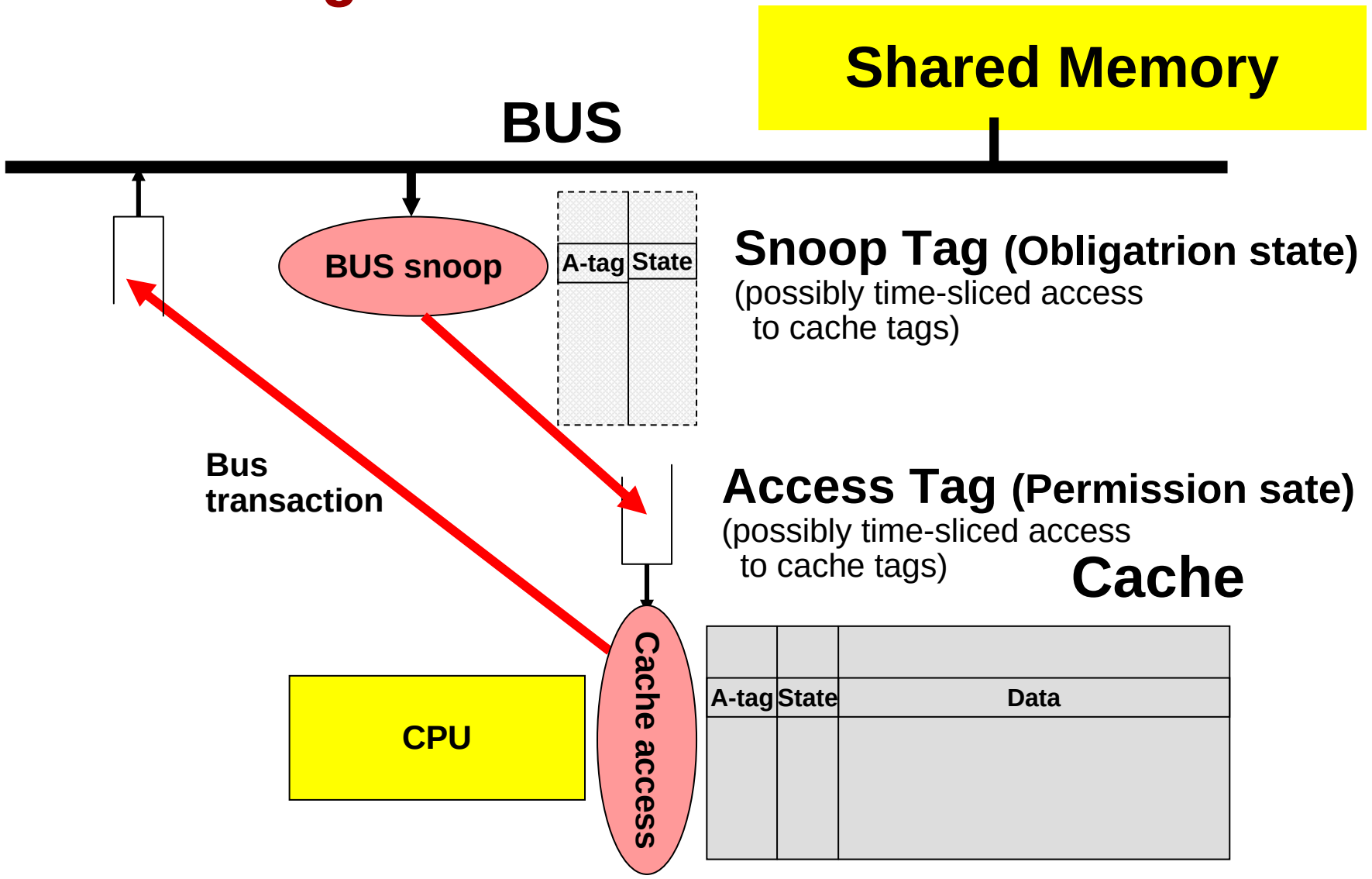
BUSrtw, ReadToWrite (reading the data with the intention to modify it)

BUSwb: Writing data back to memory

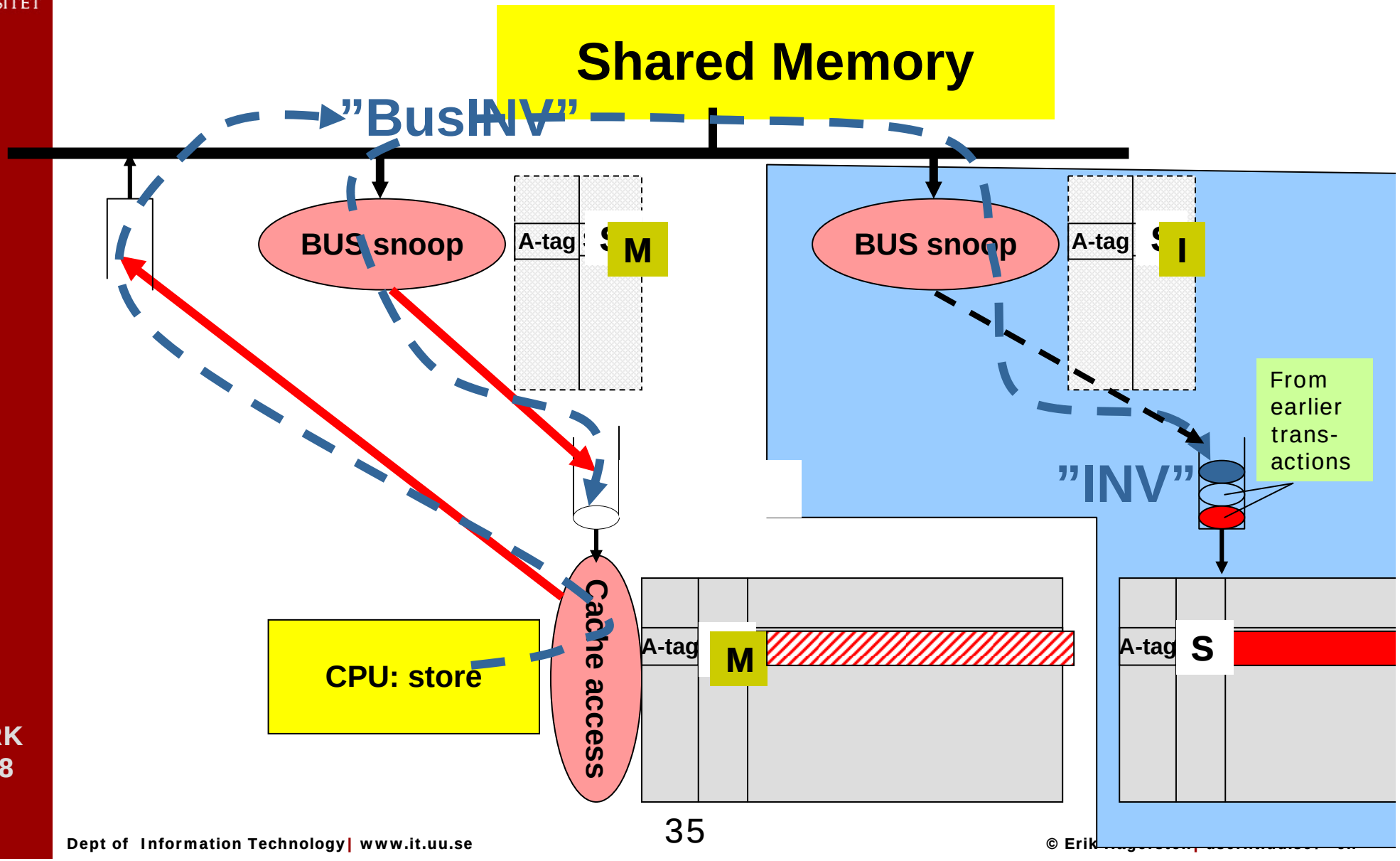
BUSinv: Invalidating other caches copies



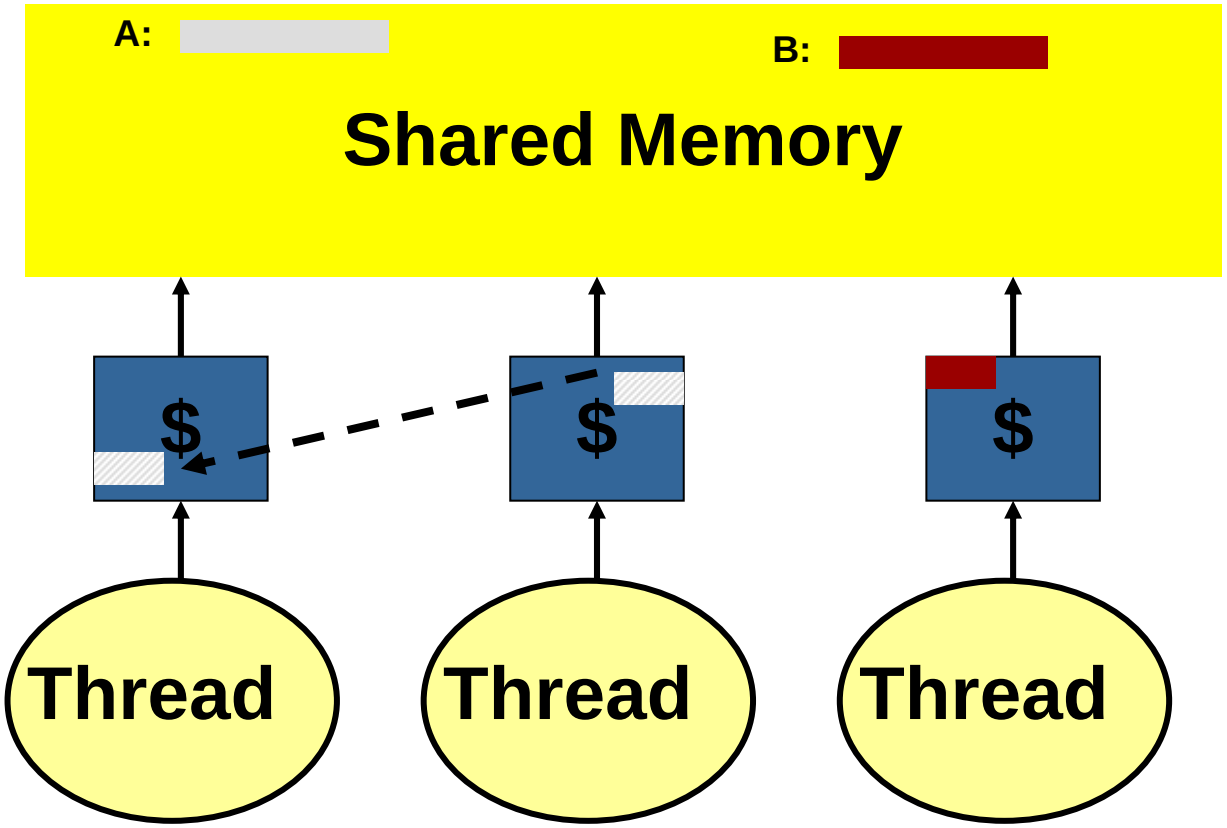
Modern snoop-based architecture -- dual tags



”Upgrade” in snooped-based



The Cache Coherent Cache-to-cache



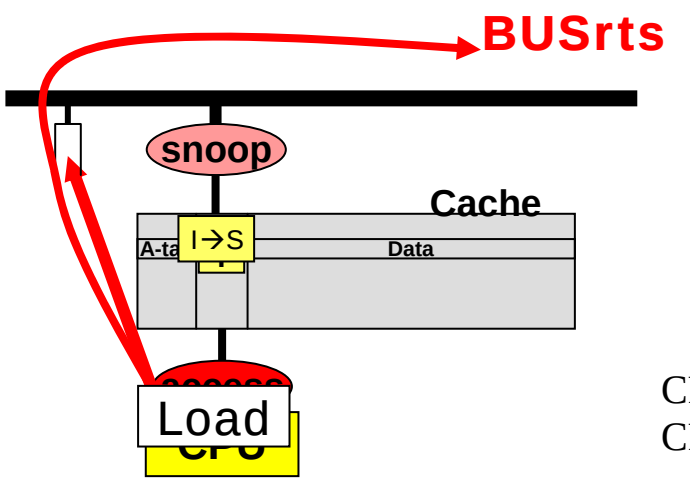
Read A	...	Read B
Read A	Read A	...
...	...	Read A
...	Write A	
Read A		

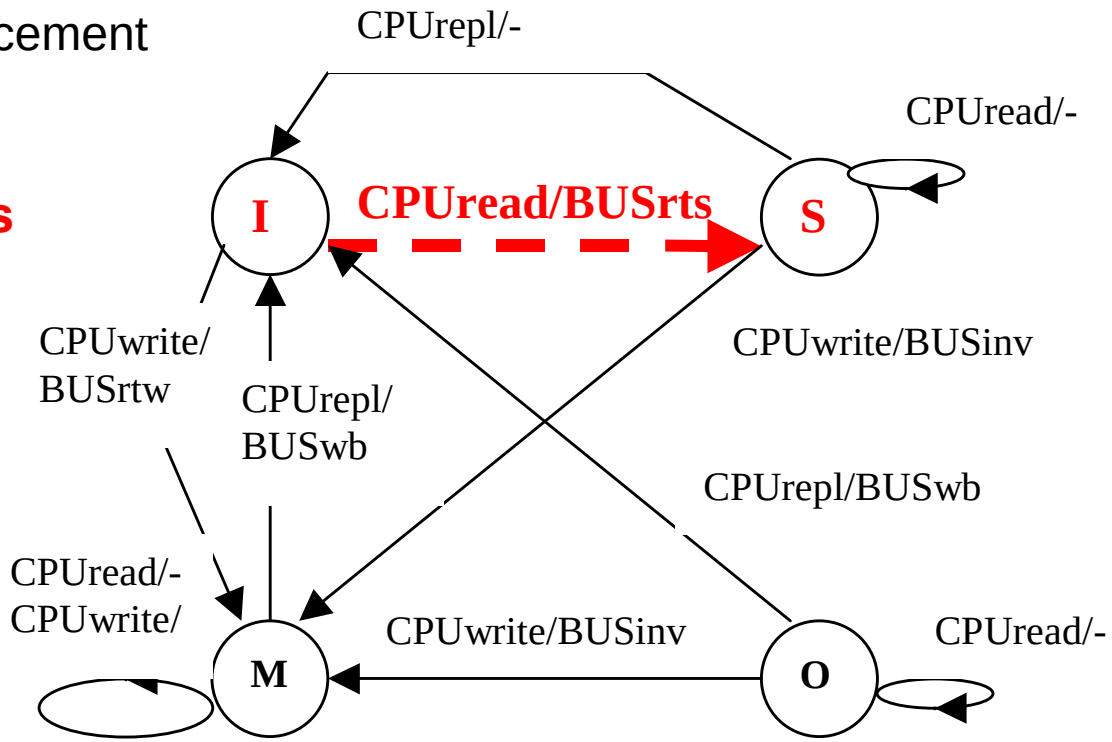
Cache2cache – the requesting CPU

CPUwrite: Caused by a store miss

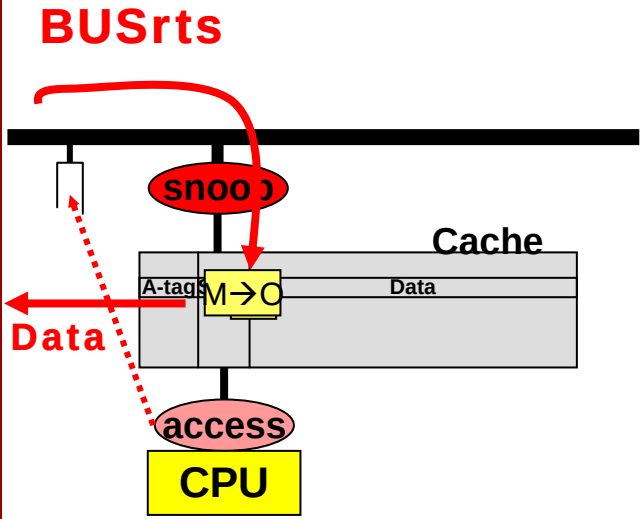
CPUread Caused by a loadmiss

CPUrepl: Caused by a replacement





Cache-to-cache – the other CPU

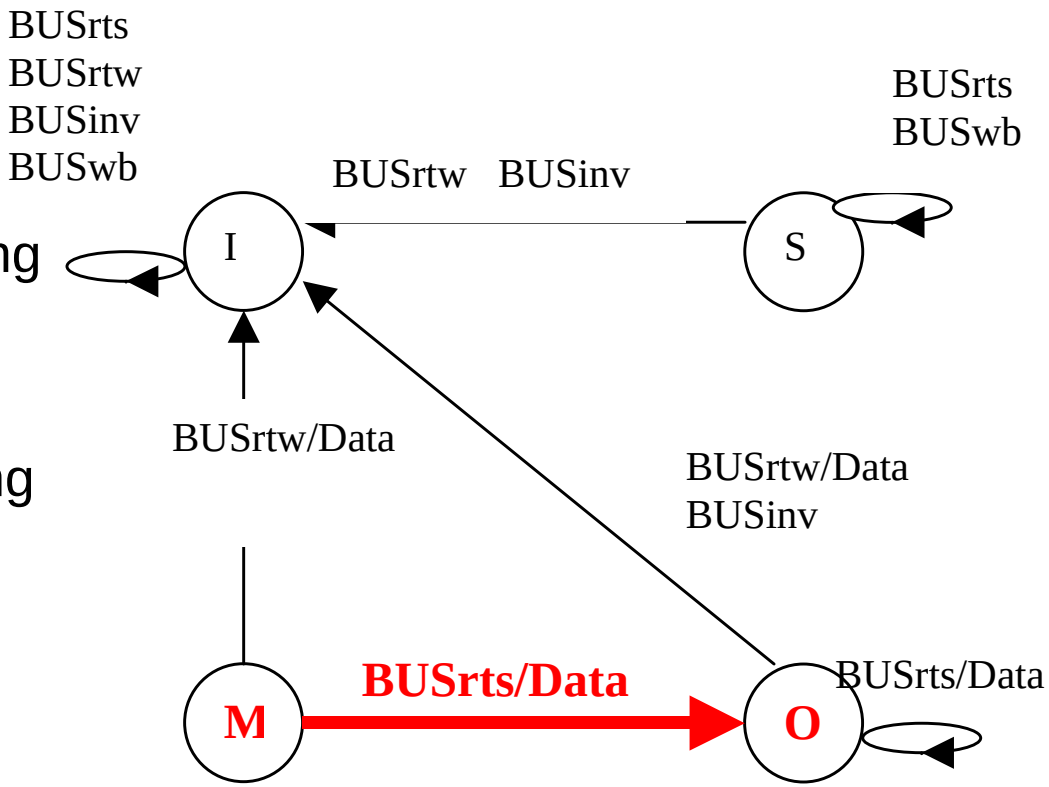


BUSrts: ReadToShare (reading the data with the intention to read it)

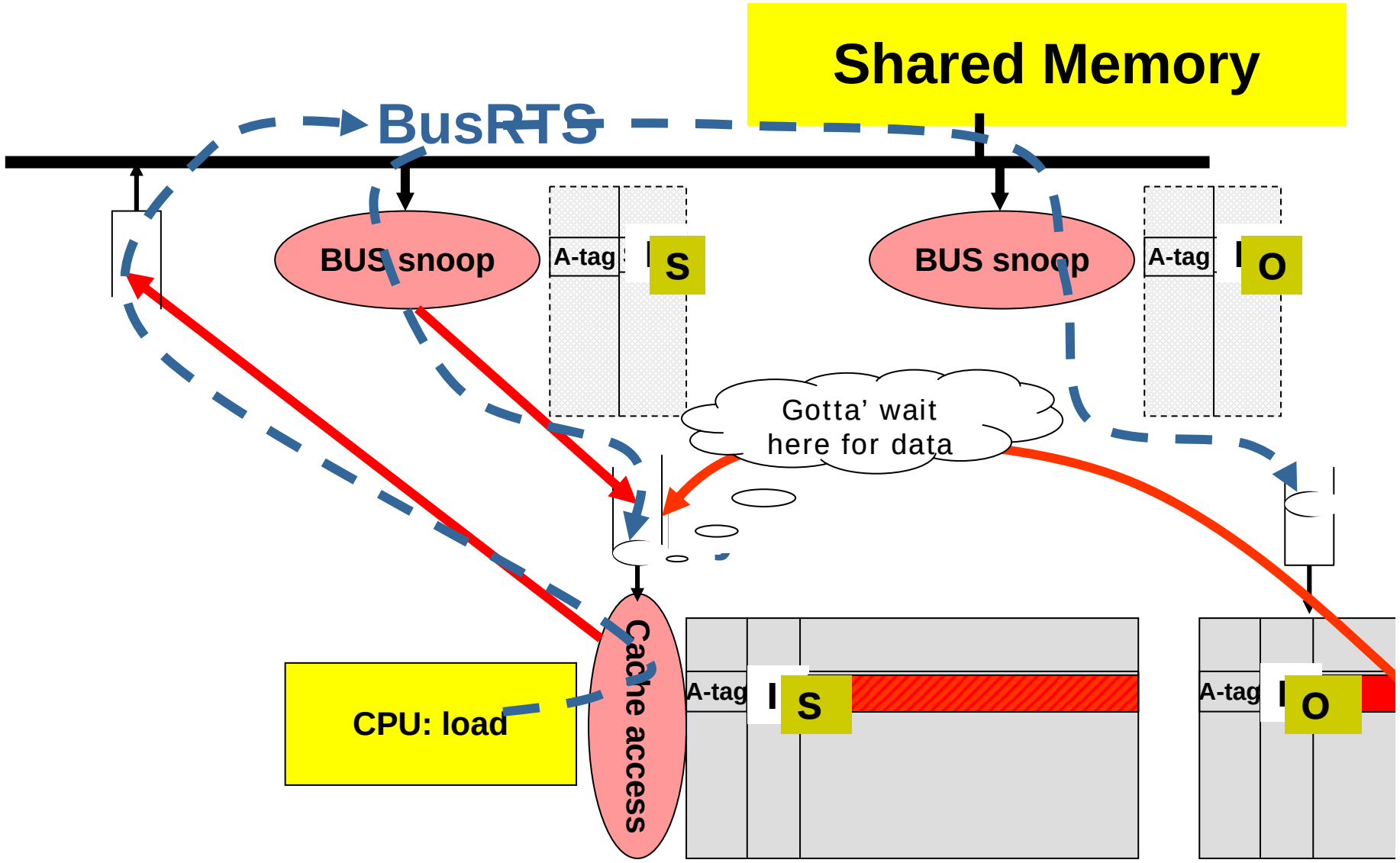
BUSrtw, ReadToWrite (reading the data with the intention to modify it)

BUSwb: Writing data back to memory

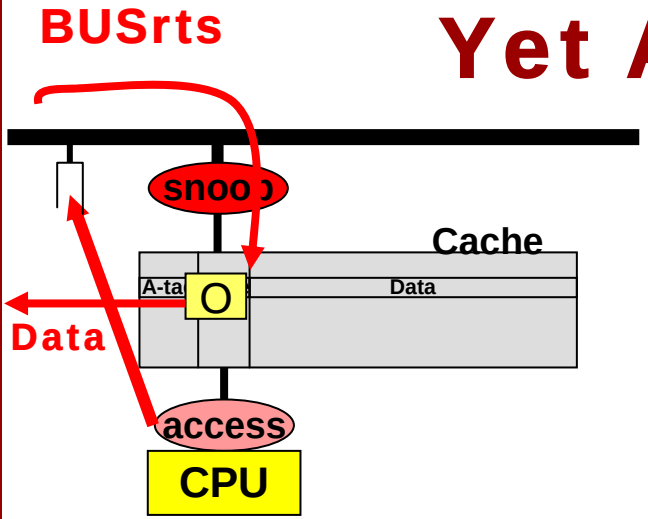
BUSinv: Invalidating other caches copies



Cache-to-cache in snoope-based



Yet Another Cache-to-cache

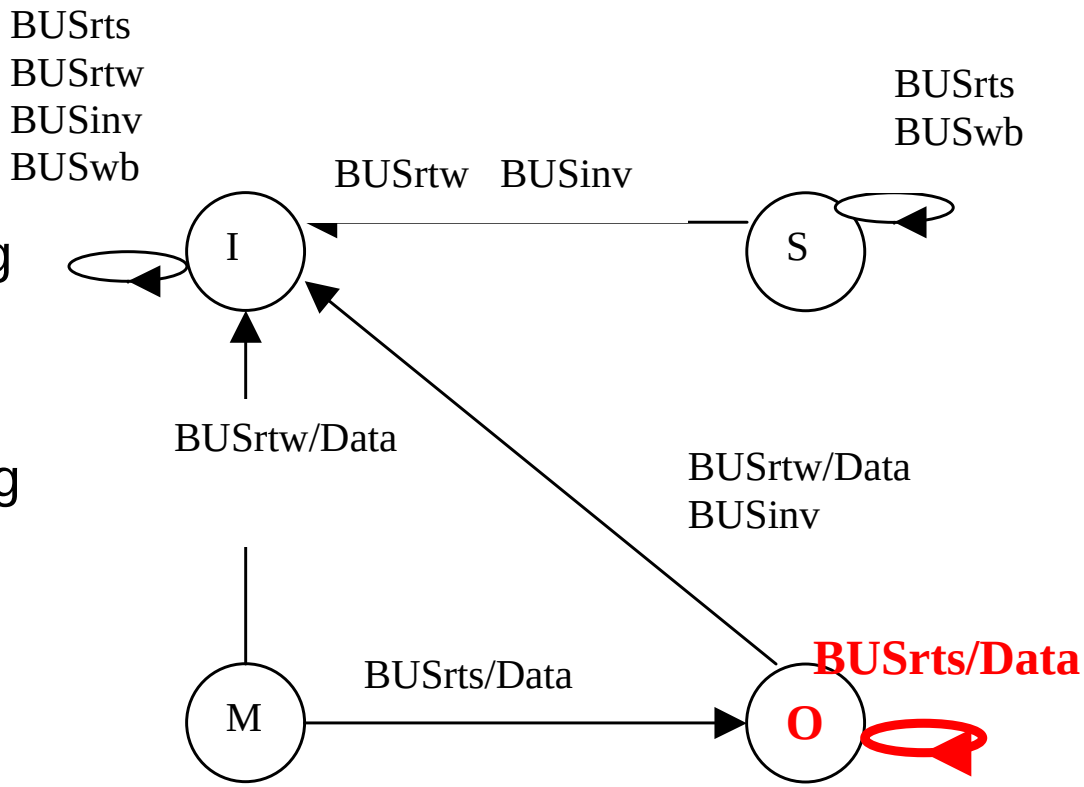


BUSrts: ReadtoShare (reading the data with the intention to read it)

BUSrtw, ReadToWrite (reading the data with the intention to modify it)

BUSwb: Writing data back to memory

BUSinv: Invalidating other caches copies



UPPSALA UNIVERSITET

All the three RISC CPUs in a **MOSI** shared-memory sequentially consistent multiprocessor executes the following code almost at the same time:

```
while(A != my_id){};   /* this is a primitive kind of lock */
B := B + A * 2;
A := A + 1;            /* this is a primitive kind of unlock */
while (A != 4) {};  /* this is a primitive kind of barrier*/
<after a long time>
<some other execution replaces A and B from the caches, if still
  present>
```

Initially, CPU1 has its local variable my_id=1, CPU has my_id=2 and CPU3 has my_id=3 and the globally shared variables A is equal to 1 and B is equal to 0. CPU2 and 3 are starting slightly ahead of CPU1 and will execute the first while statement before CPU1. Initially, both A and B only reside in memory.

The following four bus transaction types can be seen on the snooping bus connecting the CPUs:

- **RTS:** ReadtoShare (reading the data with the intention to read it)
- **RTW,** ReadToWrite (reading the data with the intention to modify it)
- **WB:** Writing data back to memory
- **INV:** Invalidating other caches copies

Show every state change and/or value change of A and B in each CPU's cache according to one possible interleaving of the memory accesses. After the parallel execution is done for all of the CPUs, the cache lines still in the caches will be replaced. These actions should also be shown. For each line, also state what bus transaction occurs on the bus (if any) as well as which device is providing the corresponding data (if any).

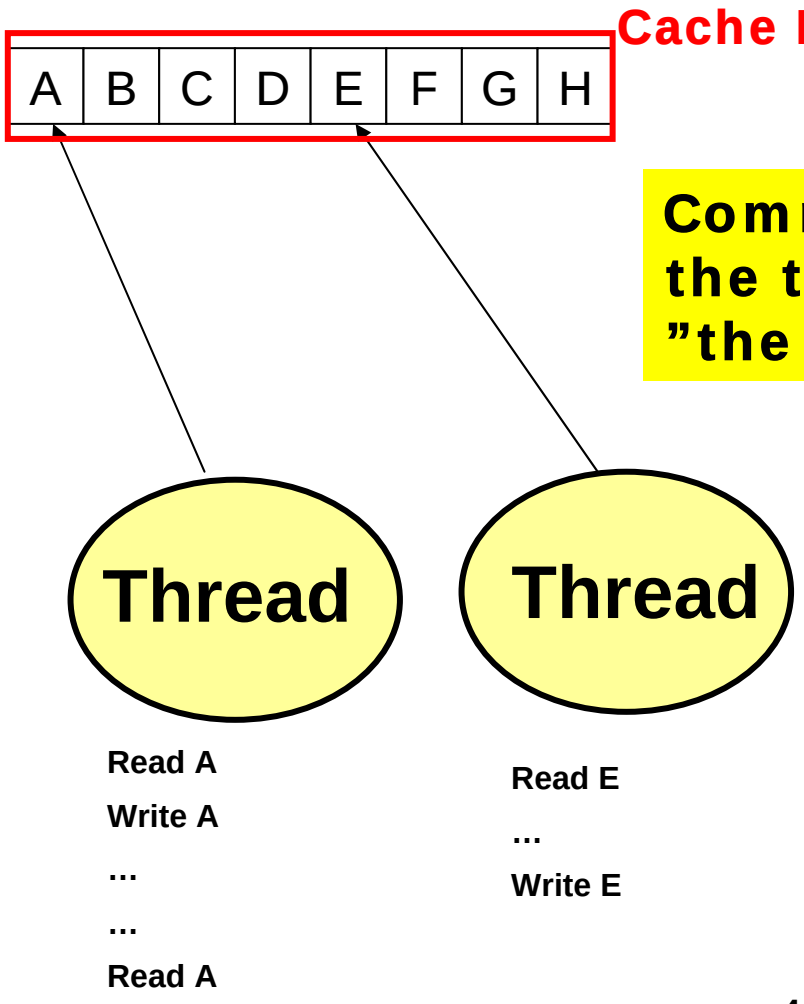
UPPSALA UNIVERSITET

Example of a state transition sheet:

CPU action	**Bus Transaction (if any)**	**State/value after the CPU action**						**Data is provided by** [CPU 1, 2, 3 or Mem] (if any)
		CPU1		CPU2		CPU3		
		A	B	A	B	A	B	
Initially		I	I	I	I	I	I	
CPU1: LD A	RTS(A)	S/1						Mem
CPU2: LD B	RTS(B)				S/0			Mem
…some time elapses .								
CPU1: replace A	-	I						-
CPU2: replace B	-				I			-

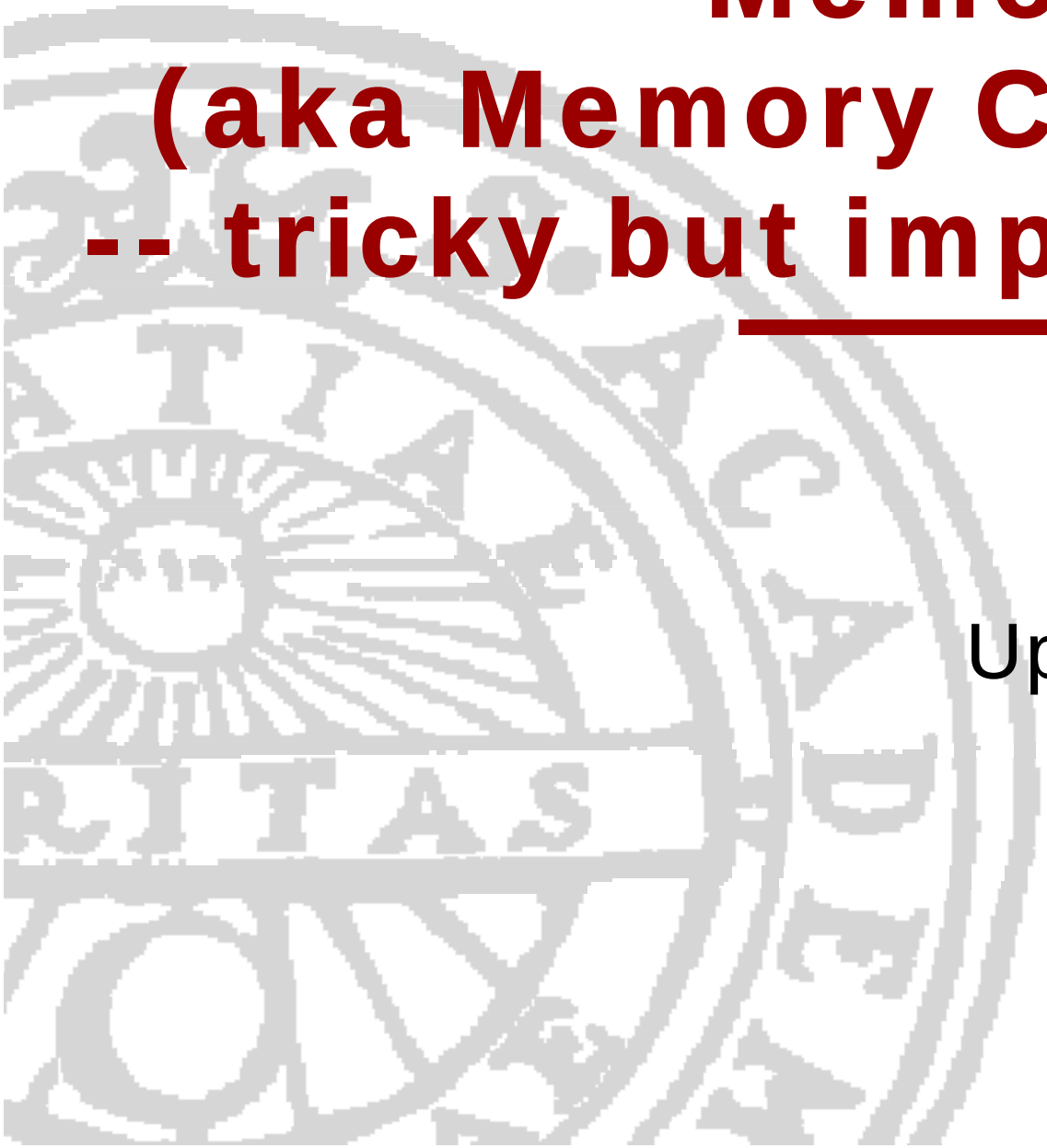
DARK 2008

False sharing



Communication misses even though the threads do not share data
”the cache line is too large”

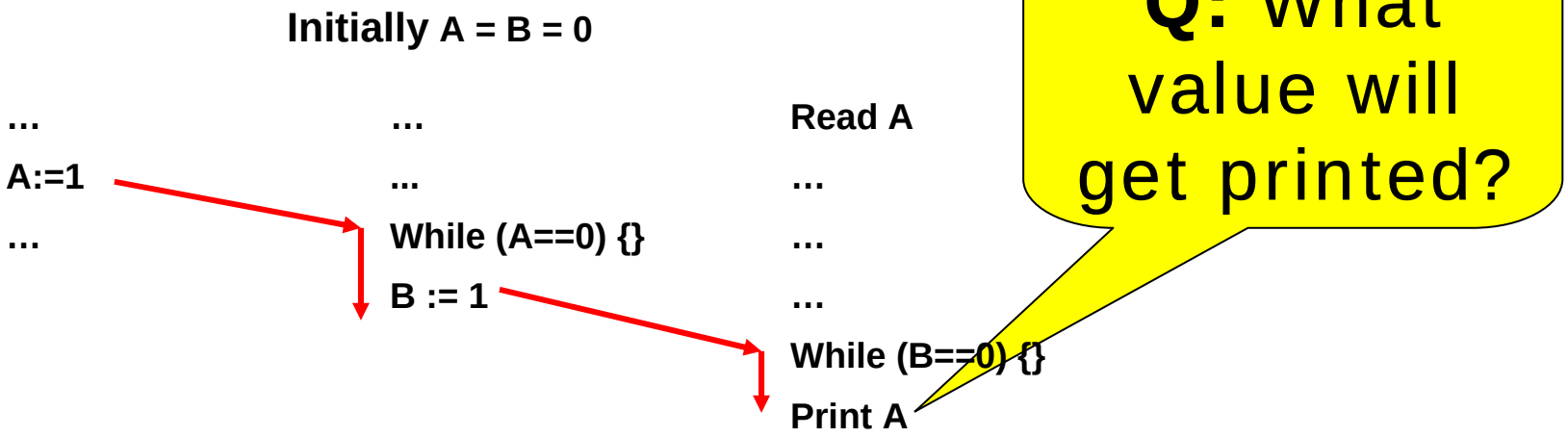
Memory Ordering (aka Memory Consistency) -- tricky but important stuff



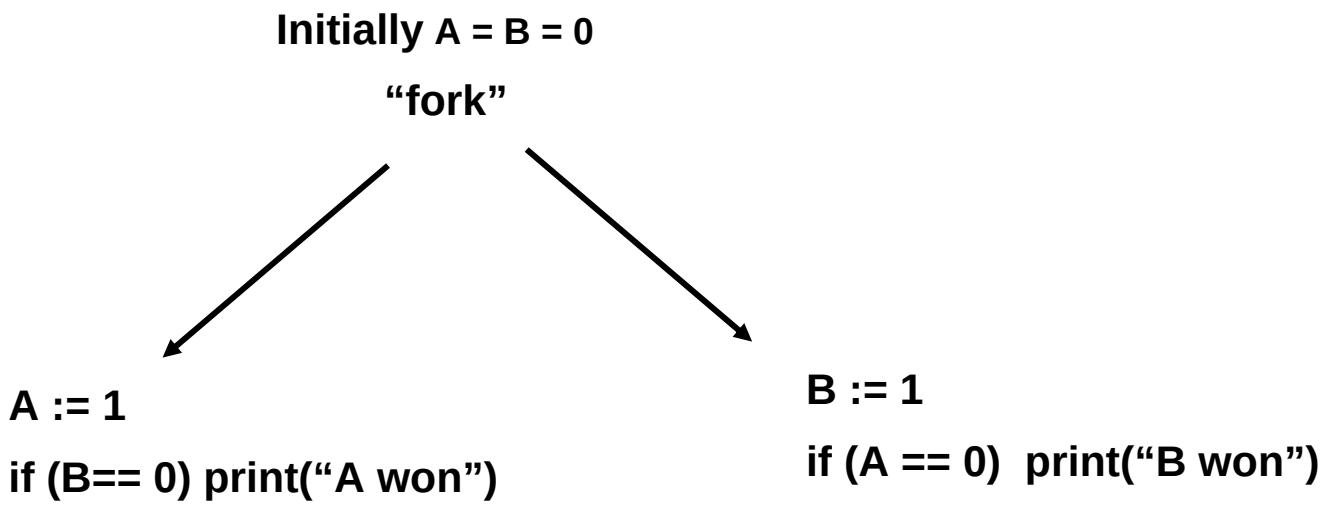
Erik Hagersten
Uppsala University
Sweden

Memory Ordering

- Coherence defines a per-datum valuechange order
- Memory model defines the valuechange order for all the data.



Dekker's Algorithm



Q: Is it possible that both A and B win?

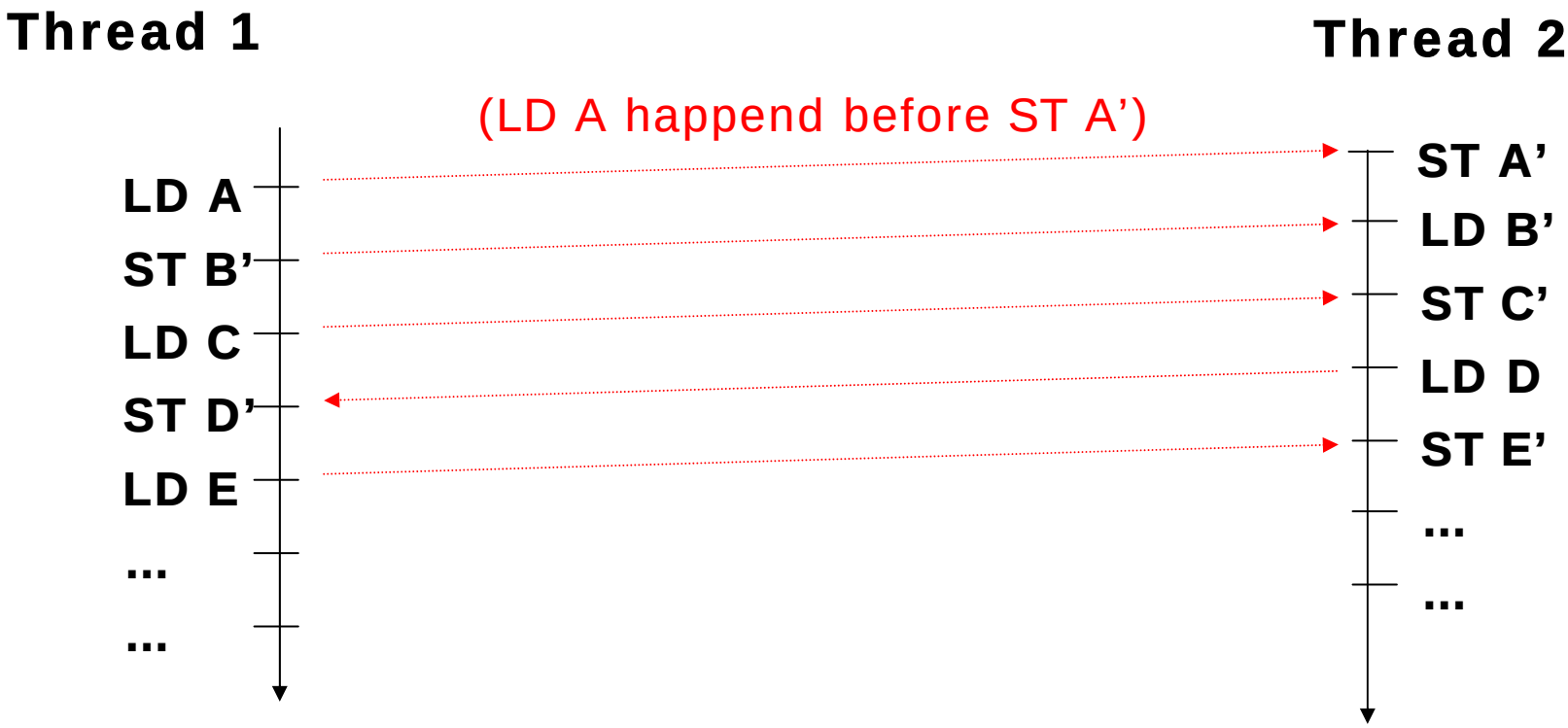
UPPSALA UNIVERSITET

Memory Ordering

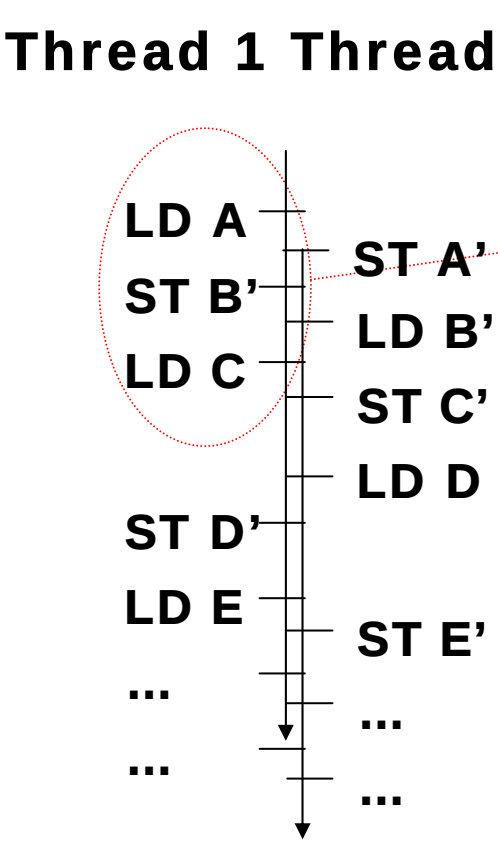
- Defines the guaranteed memory ordering
- Is a ”contract” between the HW and SW guys
- Without it, you can not say much about the result of a parallel execution

In which order were these threads executed?

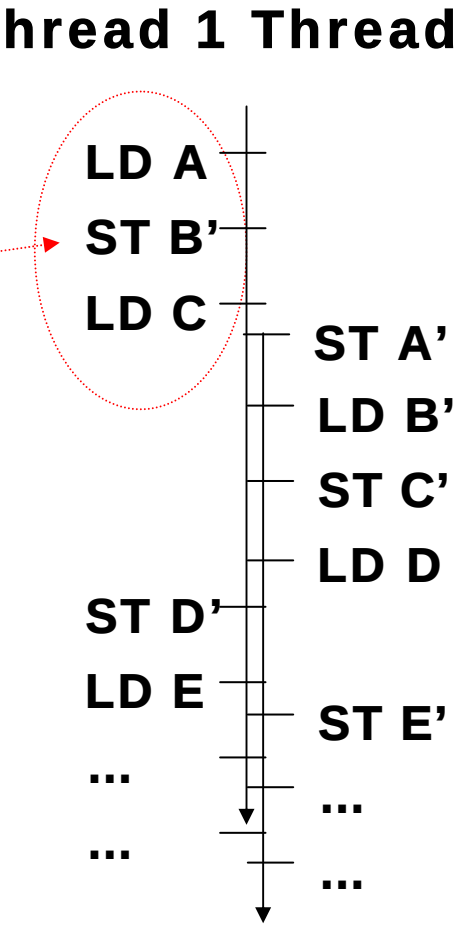
(A' denotes a modified value to the data at addr A)



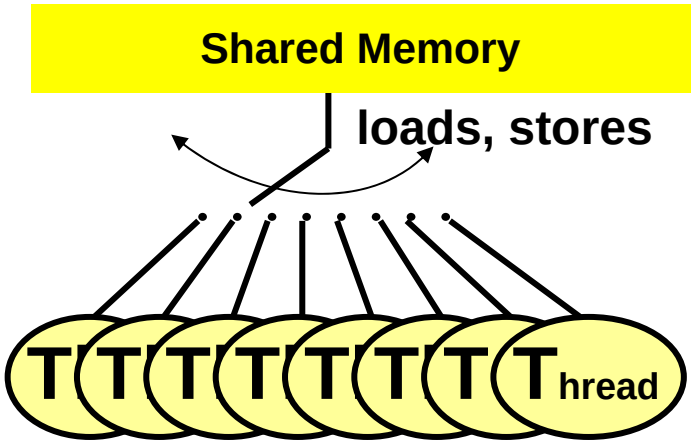
One possible observed order



Another possible observed order

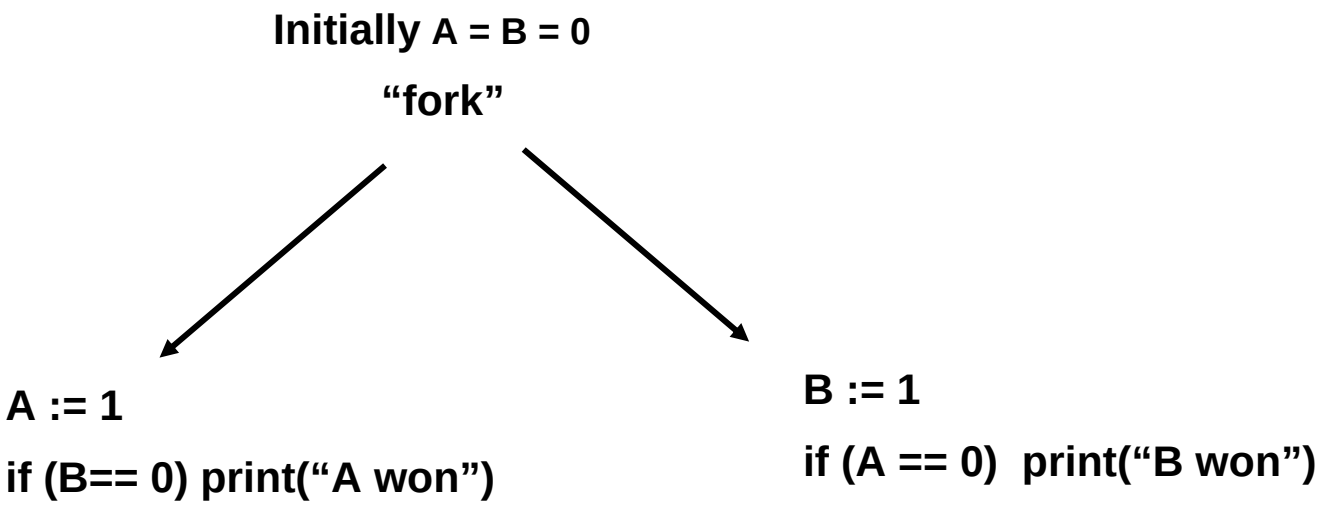


"The intuitive memory order" Sequential Consistency (Lamport)



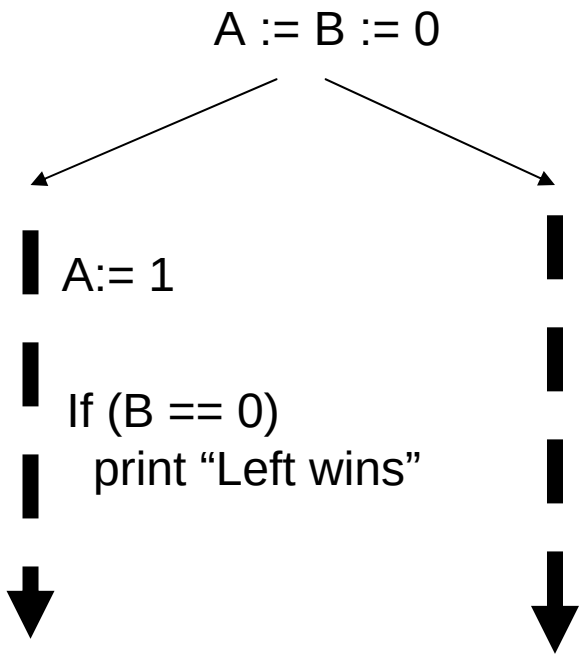
- Global order achieved by *interleaving* all memory accesses from different threads
- "Programmer's intuition is maintained"
  - Store causality? Yes
  - Does Dekker work? Yes
- Unnecessarily restrictive ==> performance penalty

Dekker's Algorithm



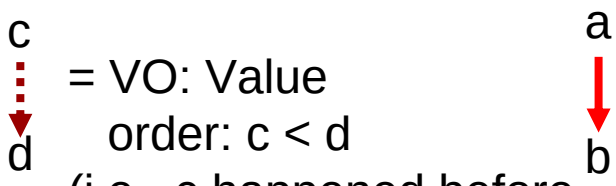
Q: Is it possible that both A and B win?

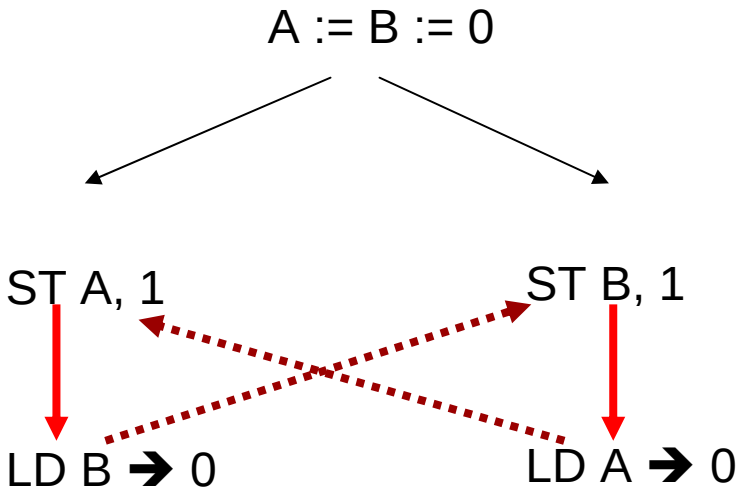
Sequential Consistency (SC) Violation → Dekker: both wins



Both Left and Right wins → SC violation

Acess graph

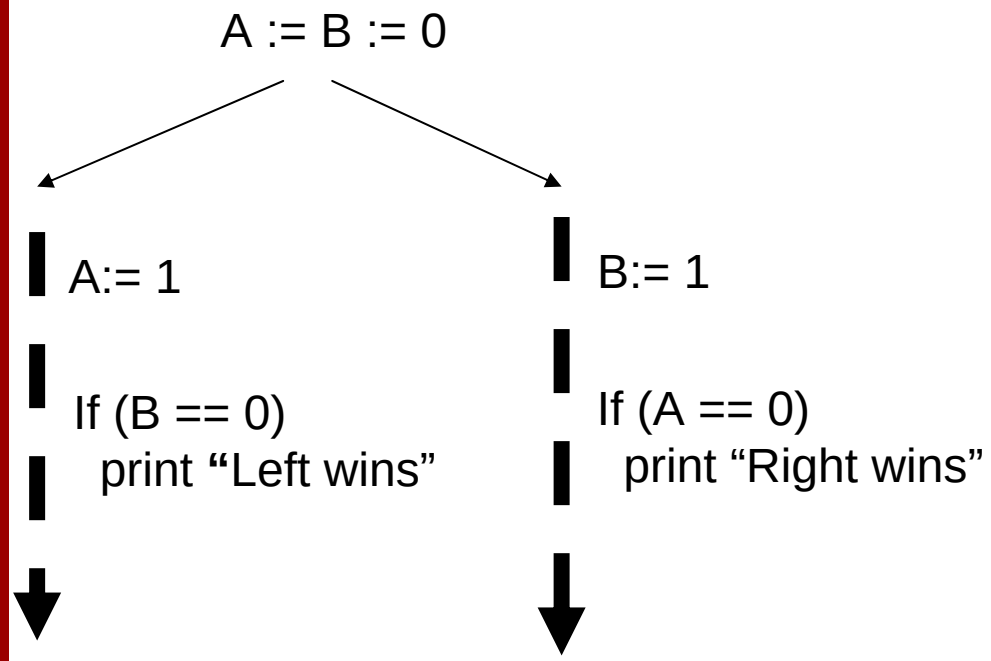


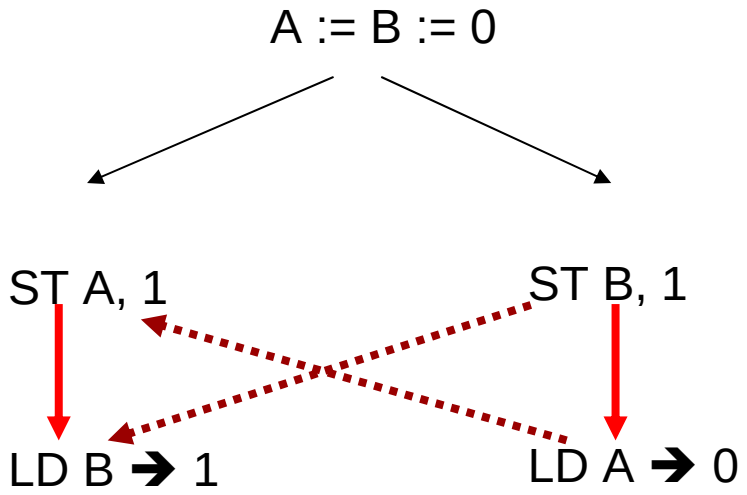


Cyclic access graph → Not SC (there is no global order)

SC is OK if one thread wins

Only Right wins ➔ SC is OK



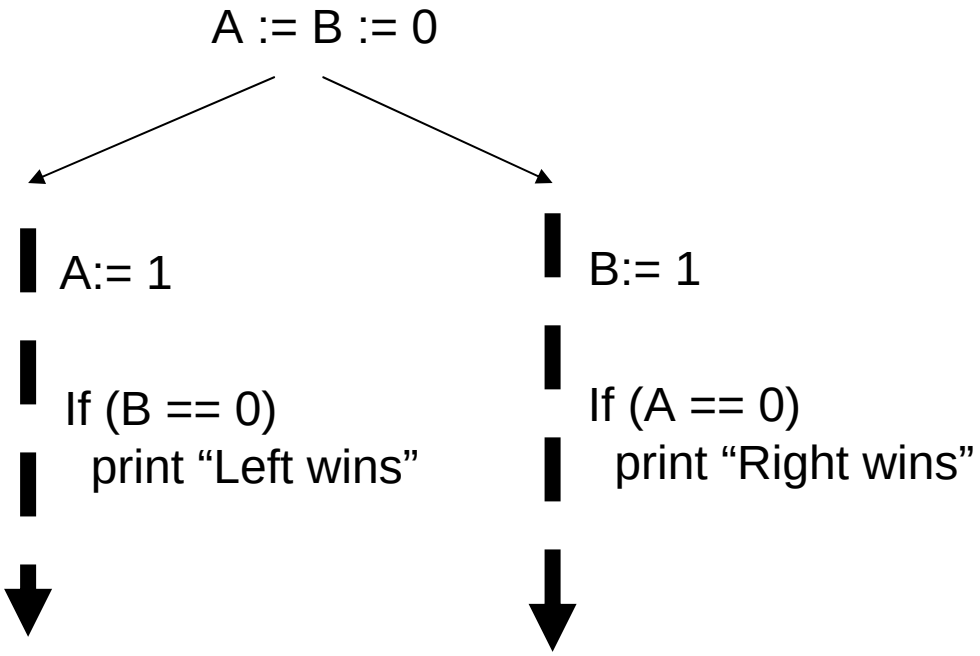


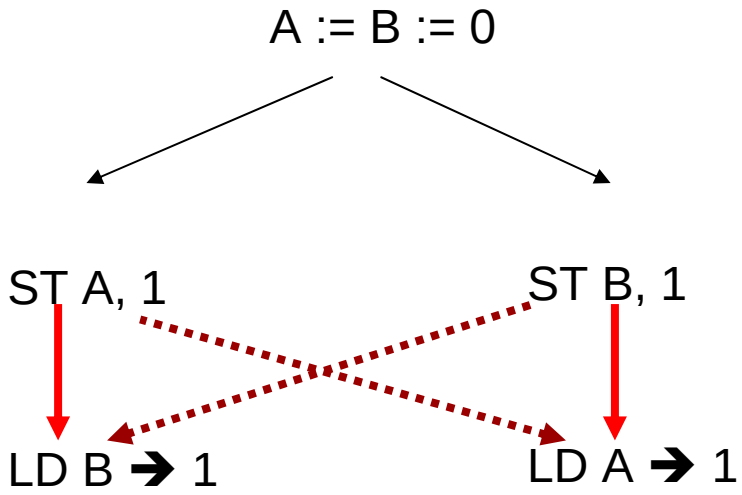
Not cyclic graph ➔ SC

One global order:
STB < LDA < STA <LDB

SC is OK if no thread wins

No thread wins ➔ SC is OK



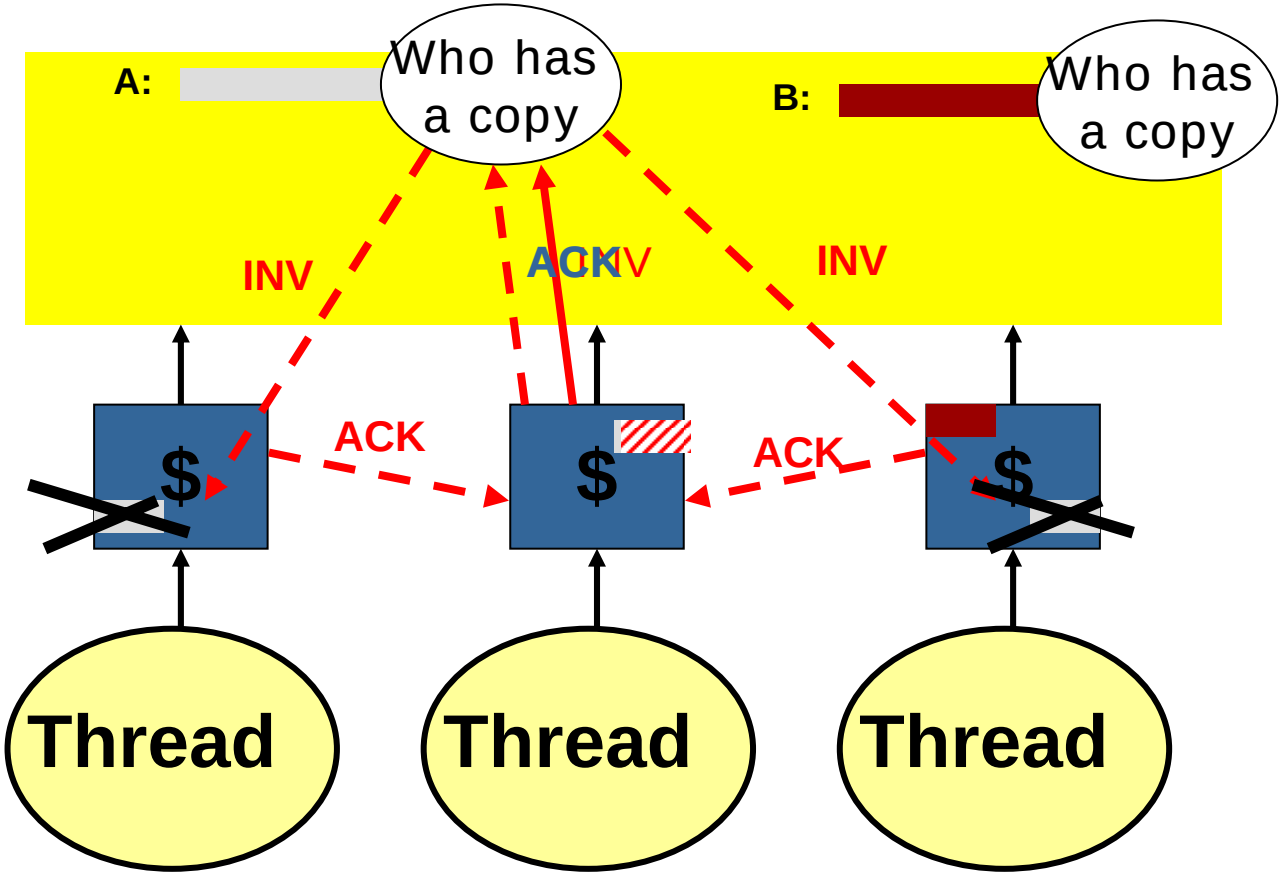


Not cyclic graph ➔ SC

Four Partial Orders, still SC

STB < LDA ; STA < LDA; STB < LDB ; STA < LDA

One implementation of SC in dir-based (....without speculation)



Read A
Read A
...
...

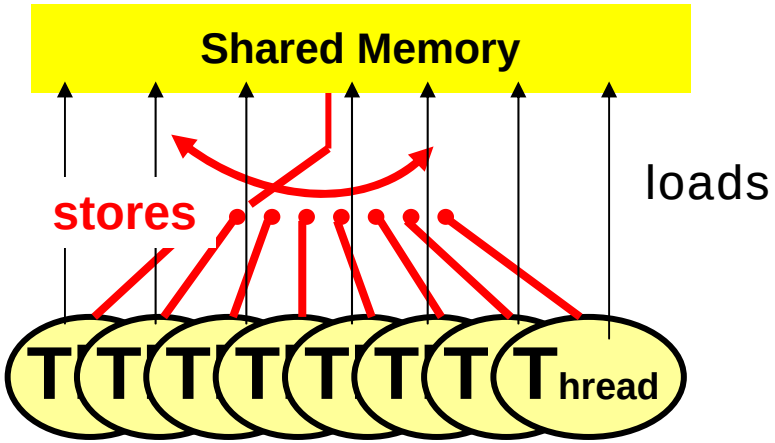
Read X
Read A
...
Write A
Read C

Read A

Read X must complete before starting Read A

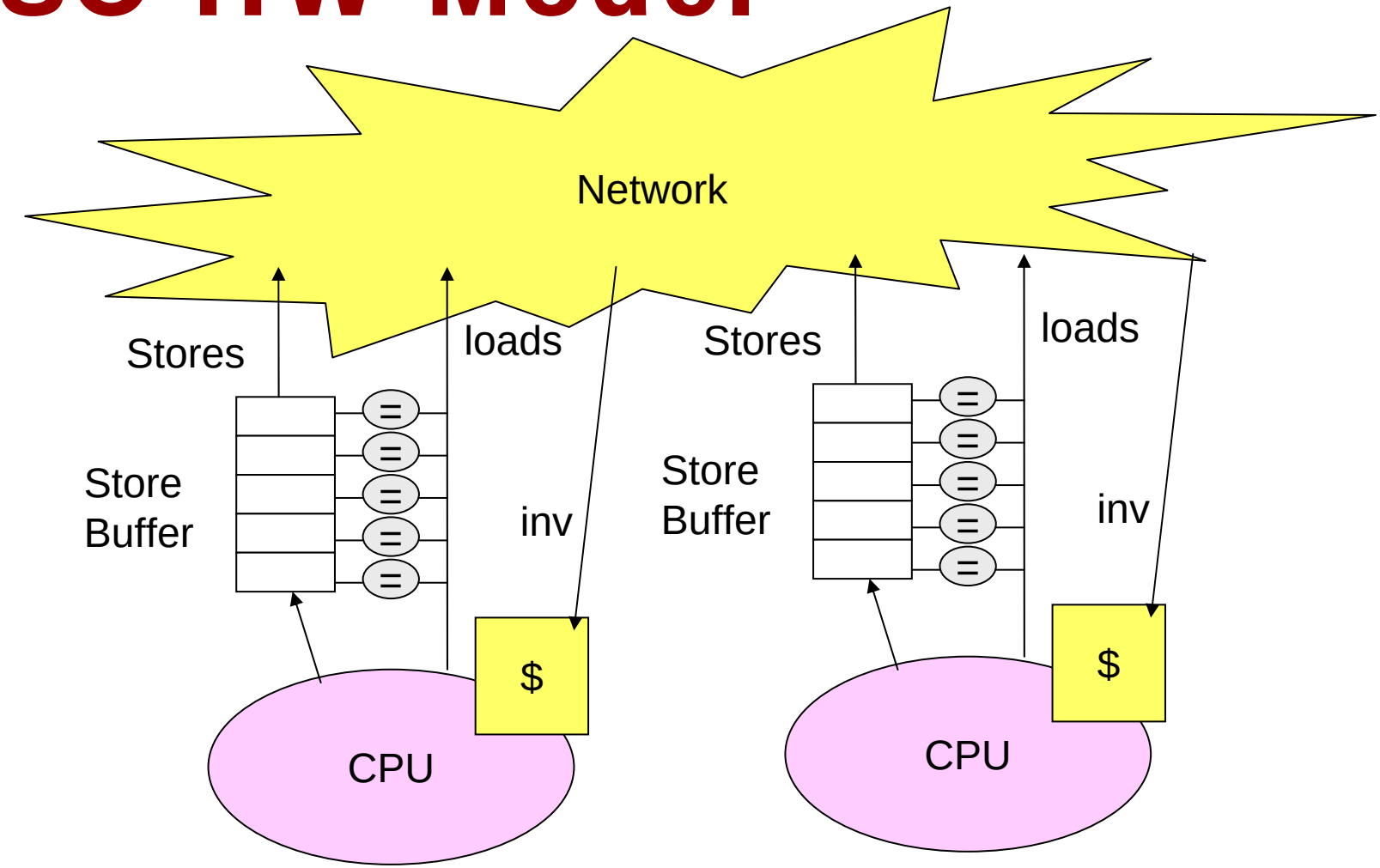
Must receive all ACKs before continuing

"Almost intuitive memory model" Total Store Ordering (P. Sindhu)



- Global order achieved by *interleaving* <u>all</u> store accesses from different threads
- "Programmer's intuition is maintained"
  - Store causality? Yes
  - Does Dekker work? No
- Unnecessarily restrictive ==> performance penalty

TSO HW Model



➔Stores are moved off the critical path

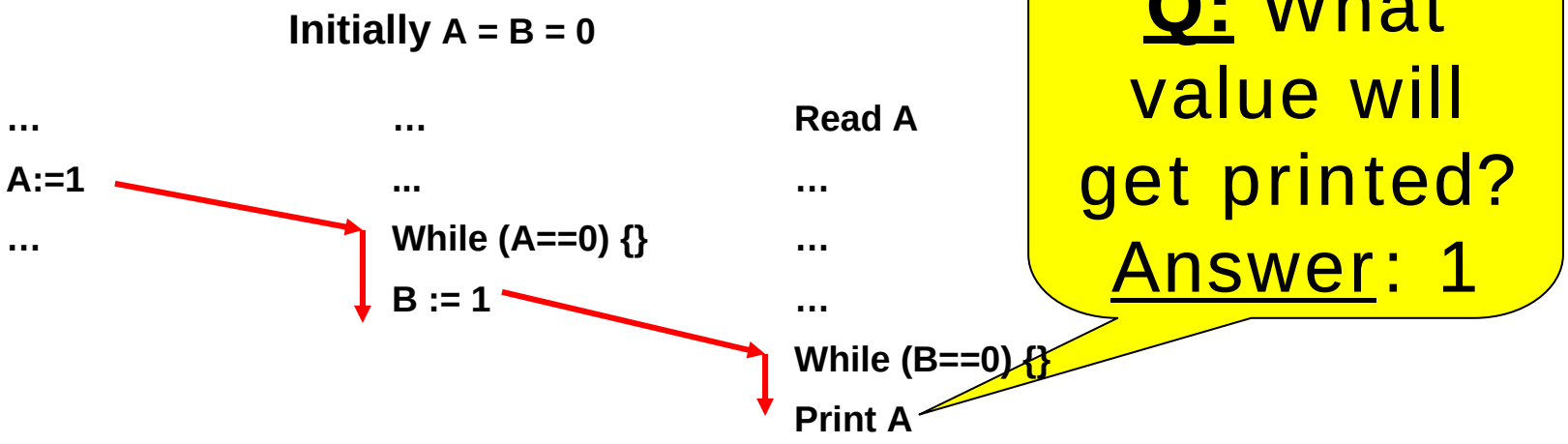
Coherence implementation can be the same as for SC

TSO

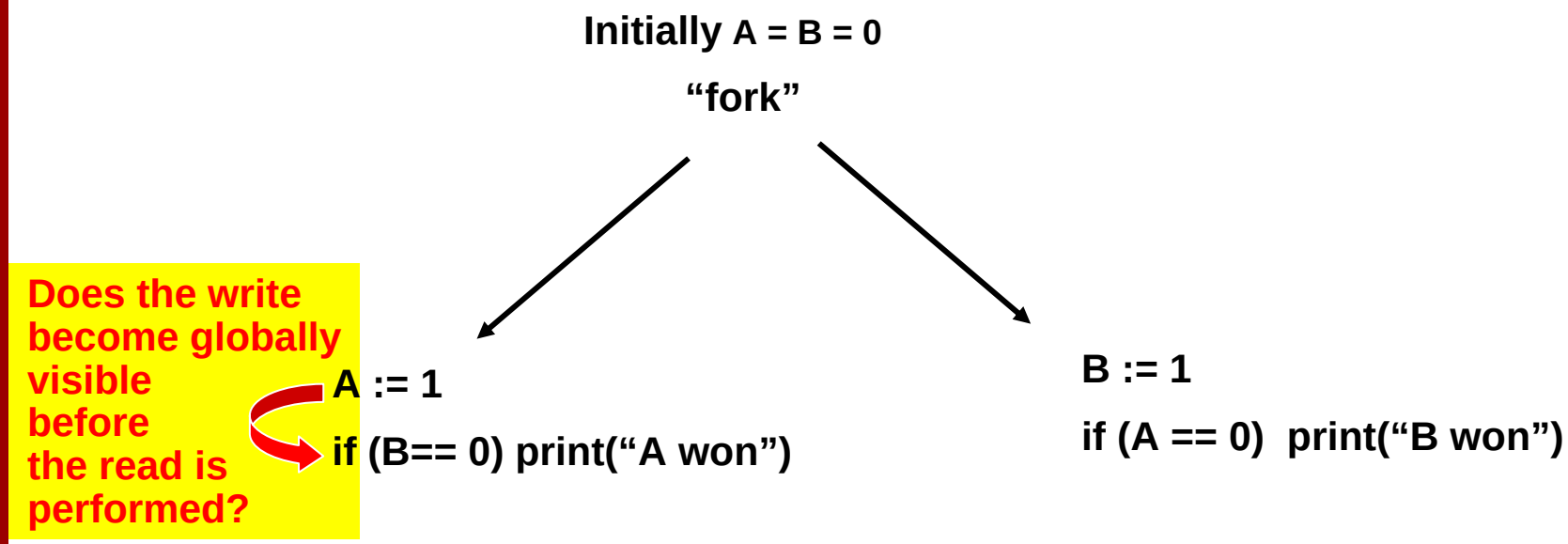
- Flag synchronization works

A := data	while (flag != 1) {};
flag := 1	X := A

- Provides causal correctness



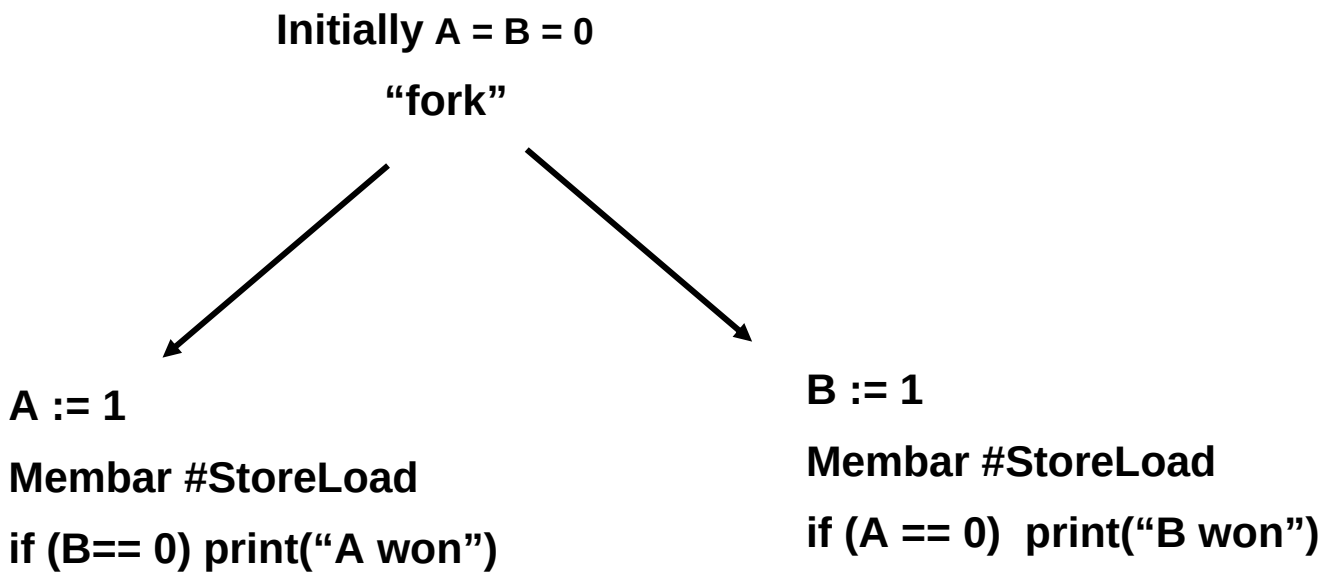
Dekker's Algorithm



Q: Is it possible that both A and B wins?

Left: The read (i.e,, test if B==0) can bypass the store (A:=1)
Right: The read (i.e,, test if A==0) can bypass the store (B:=1)
➔both loads can be performed before any of the stores
➔yes, it is possible that both wins
➔➔ Dekker's algorithm breaks

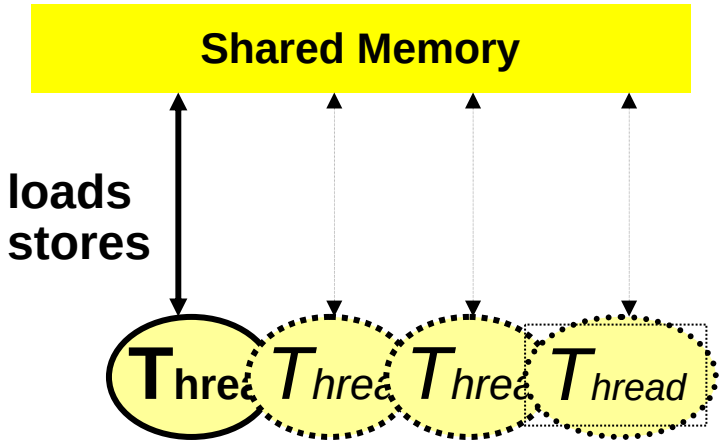
Dekker's Algorithm for TSO



Q: Is it possible that both A and B win?

Membar: The read is stared after all previous stores have been "globaly ordered"
➔behaves like SC
➔ Dekker's algorithm works!

Weak/release Consistency (M. Dubois, K. Gharachorloo)



- **Most accesses are unordered**
- **"Programmer's intuition is not maintained"**
  - Store causality? No
  - Does Dekker work? No
- **Global order only established when the programmer explicitly inserts memory barrier instructions**

++ Better performance!!

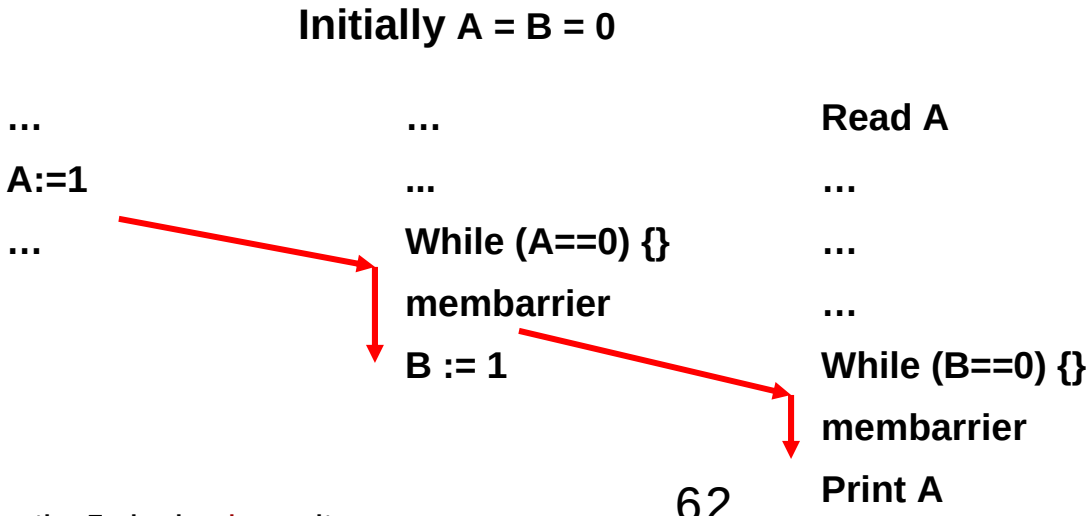
--- Interesting bugs!!

Weak/Release consistency

- New flag synchronization needed

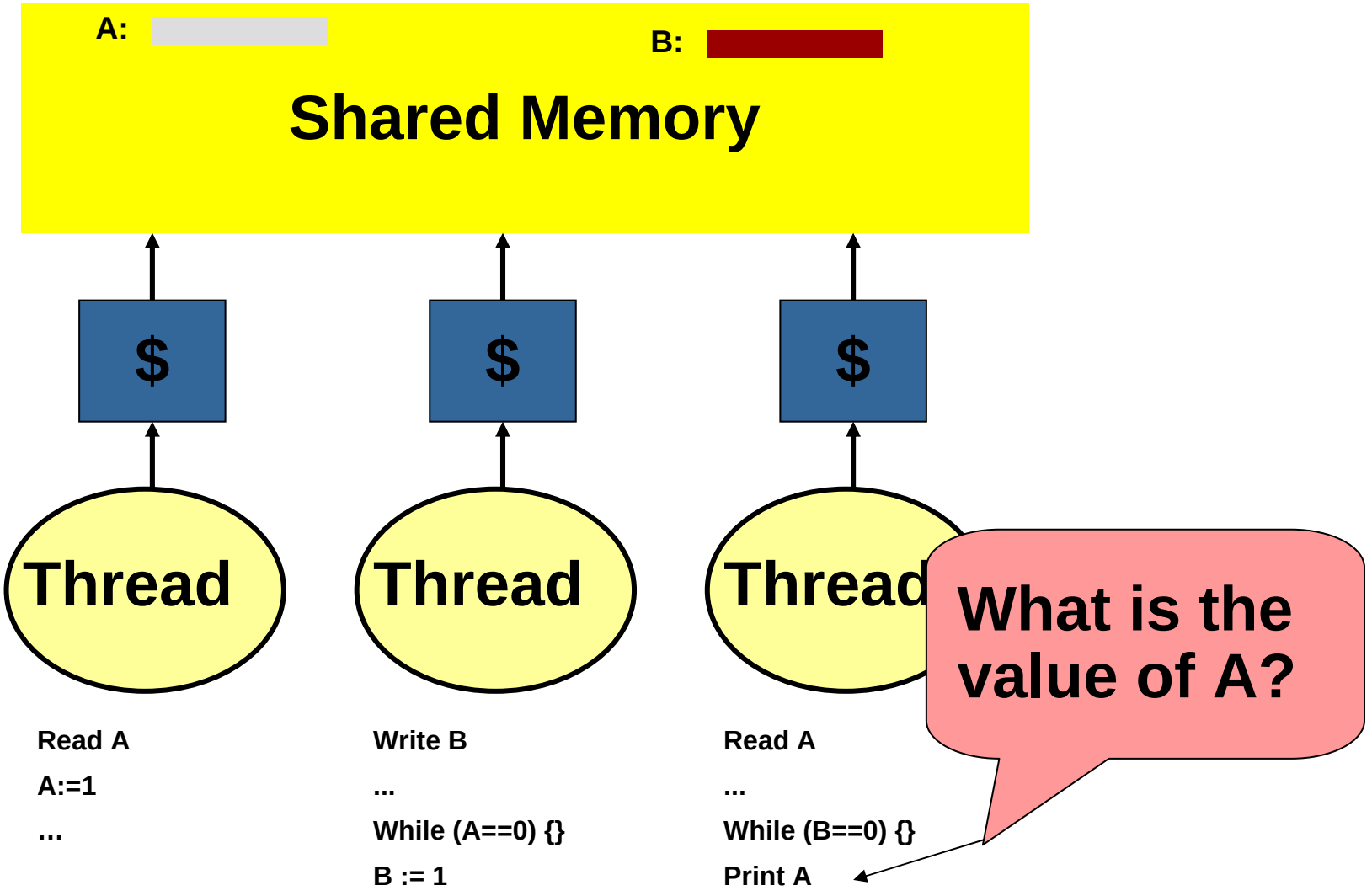
```
A := data;                  while (flag != 1) {};
membarrier;                 membarrier;
flag := 1;                  X := A;
```

- Dekker's: same as TSO
- Causal correctness provided for this code

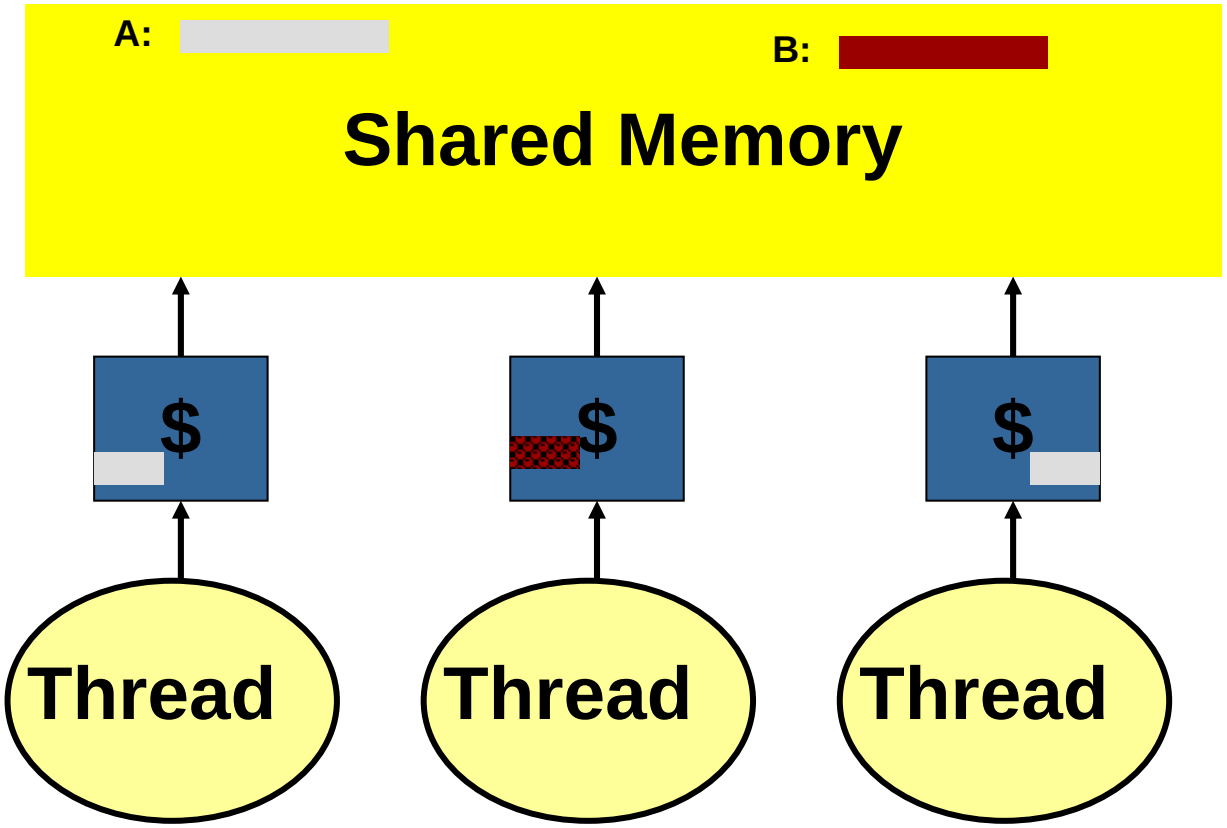


Q: What value will get printed?
Answer: 1

Example1: Causal Correctness Issues

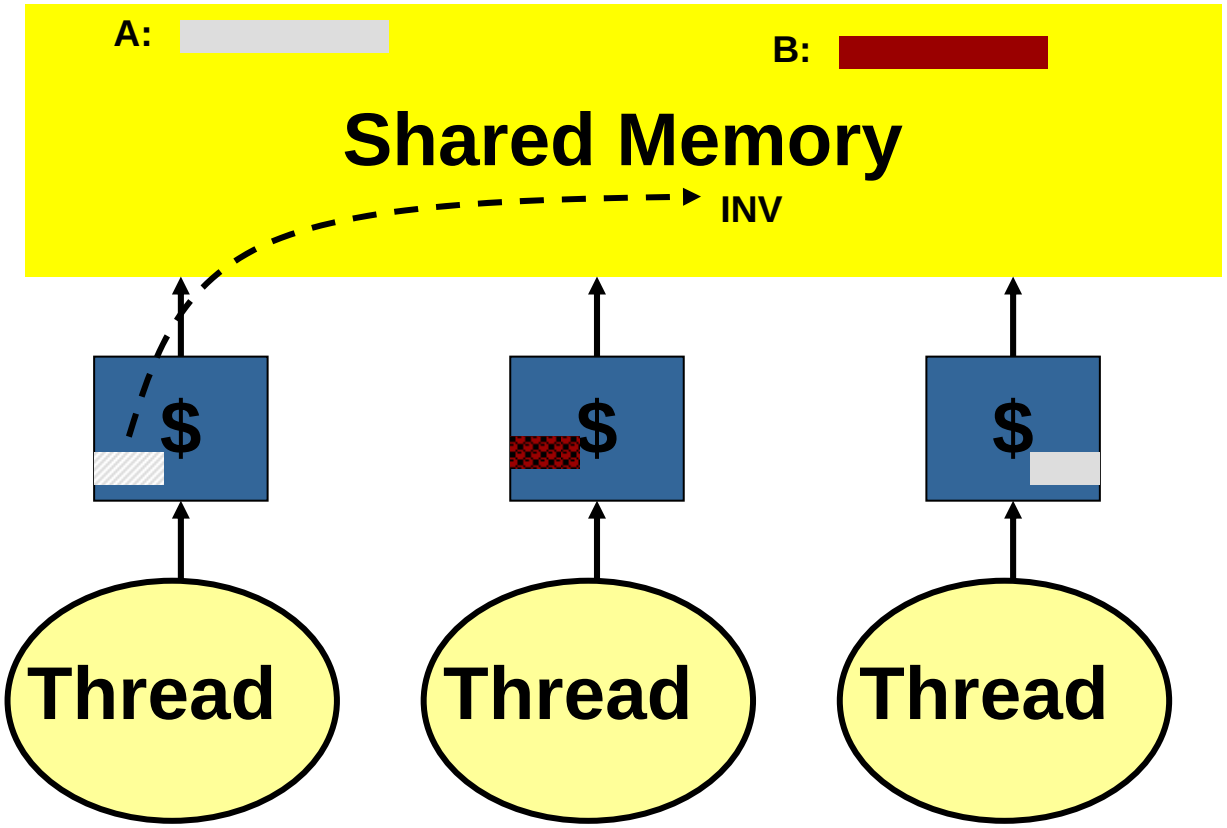


Example1: Causal Correctness Issues



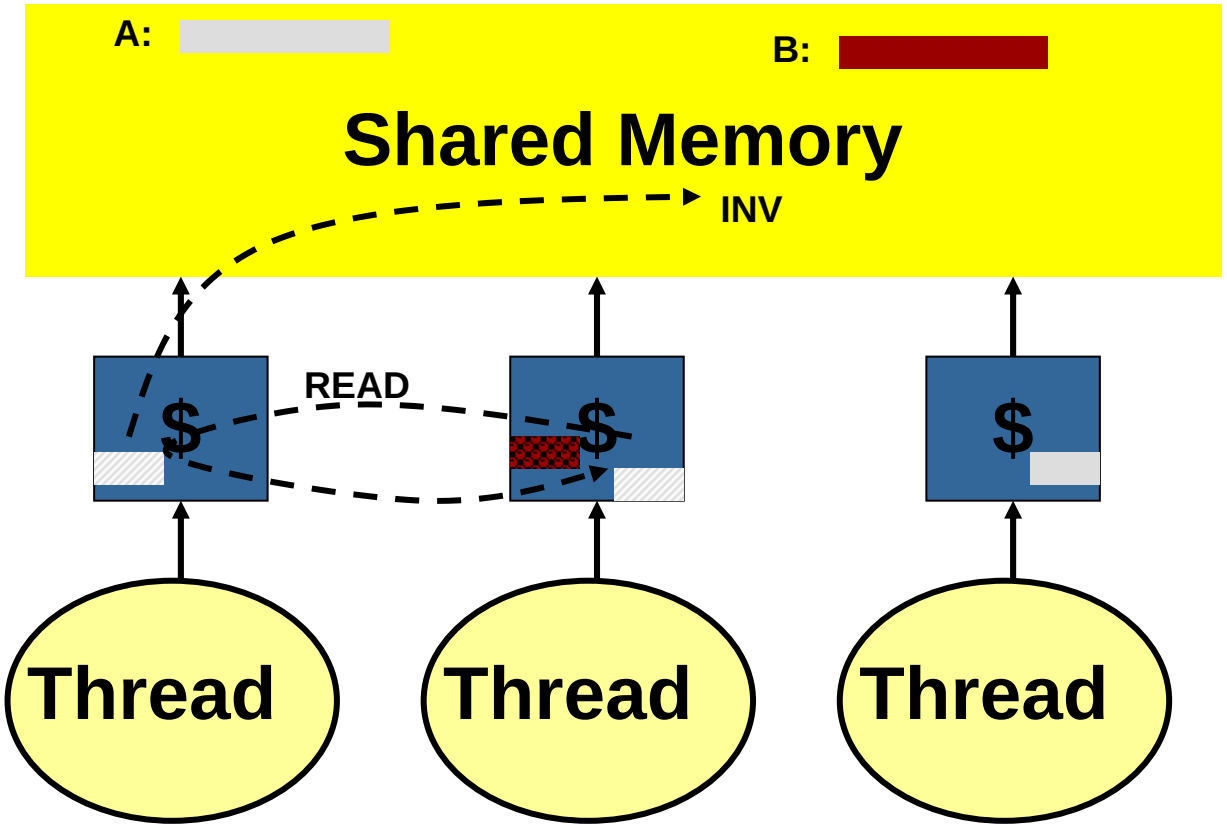
Read A	Write B	Read A
A:=1	...	...
...	While (A==0) {}	While (B==0) {}
	B := 1	Print A

Example1: Causal Correctness Issues



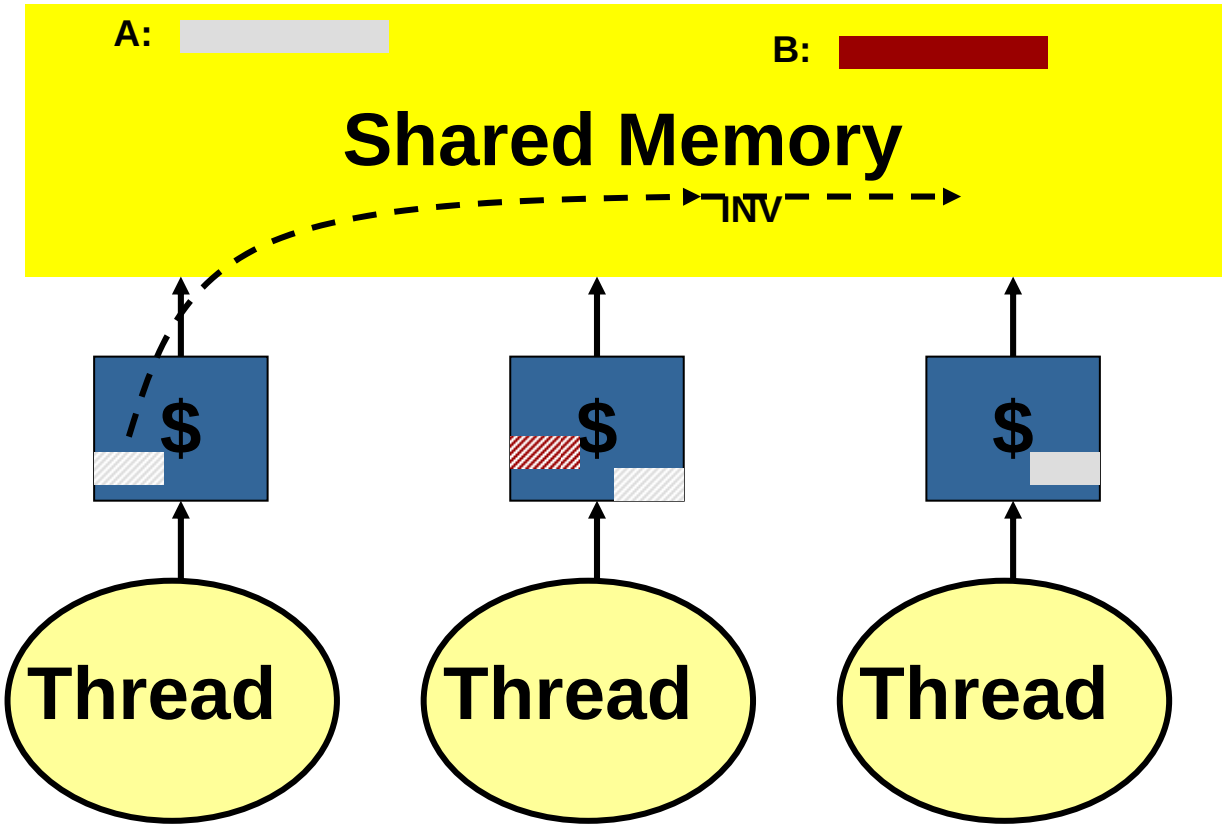
Read A	Write B	Read A
A:=1	...	...
...	While (A==0) {}	While (B==0) {}
	B := 1	Print A

Example1: Causal Correctness Issues



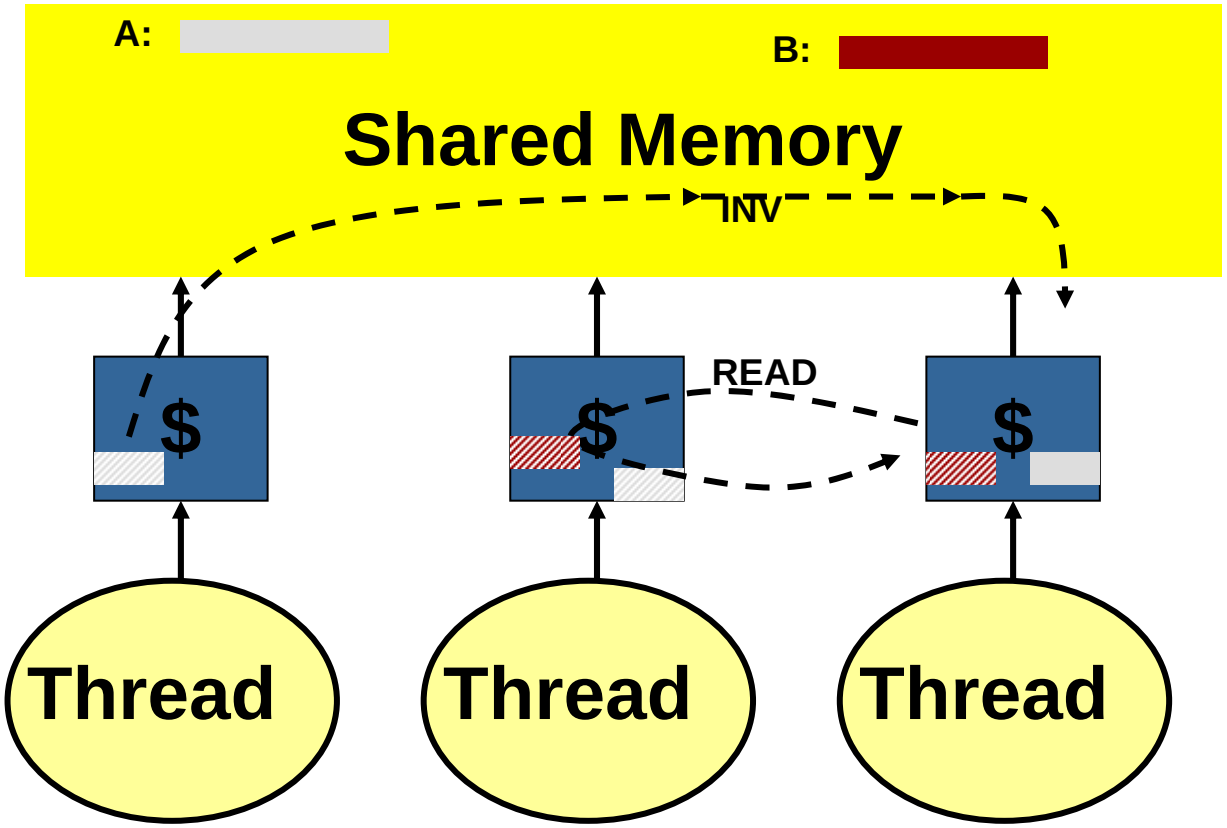
Thread	Thread	Thread
Read A	Write B	Read A
A:=1	...	...
...	While (A==0) {}	While (B==0) {}
	B := 1	Print A

Example1: Causal Correctness Issues

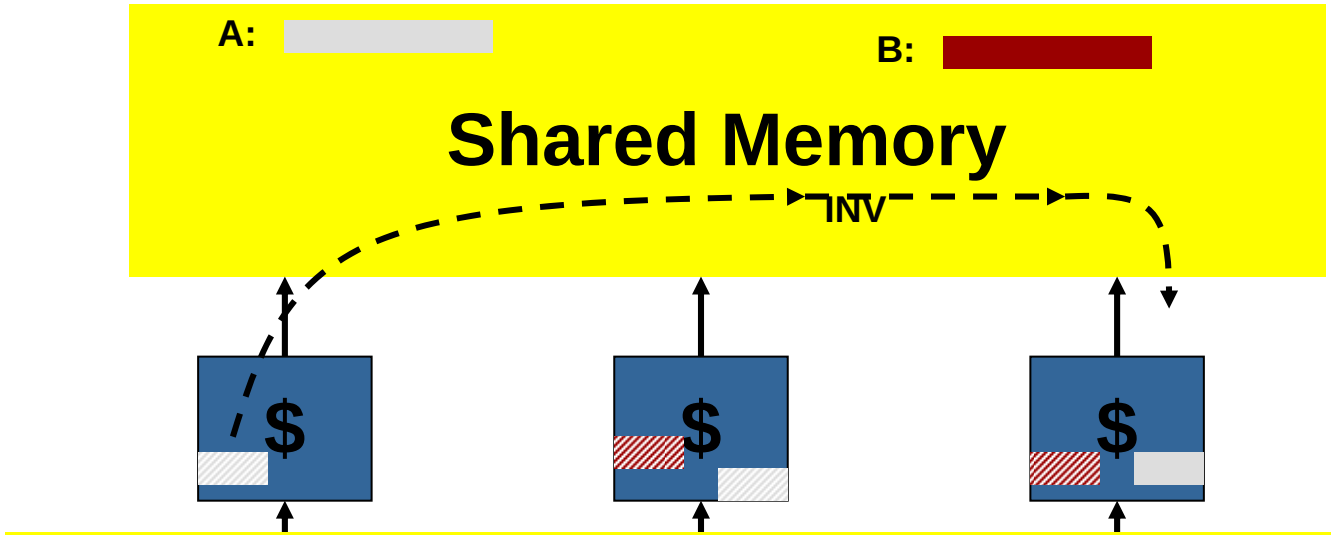


Thread	Thread	Thread
Read A	Write B	Read A
A:=1	...	...
...	While (A==0) {}	While (B==0) {}
	B := 1	Print A

Example1: Causal Correctness Issues



Example1: Causal Correctness Issues



A: if store causality ➔ ”1” will be printed

Thread

Read A

A:=1

…

Thread

Write B

…

While (A==0) {}

B := 1

Thread

Read A

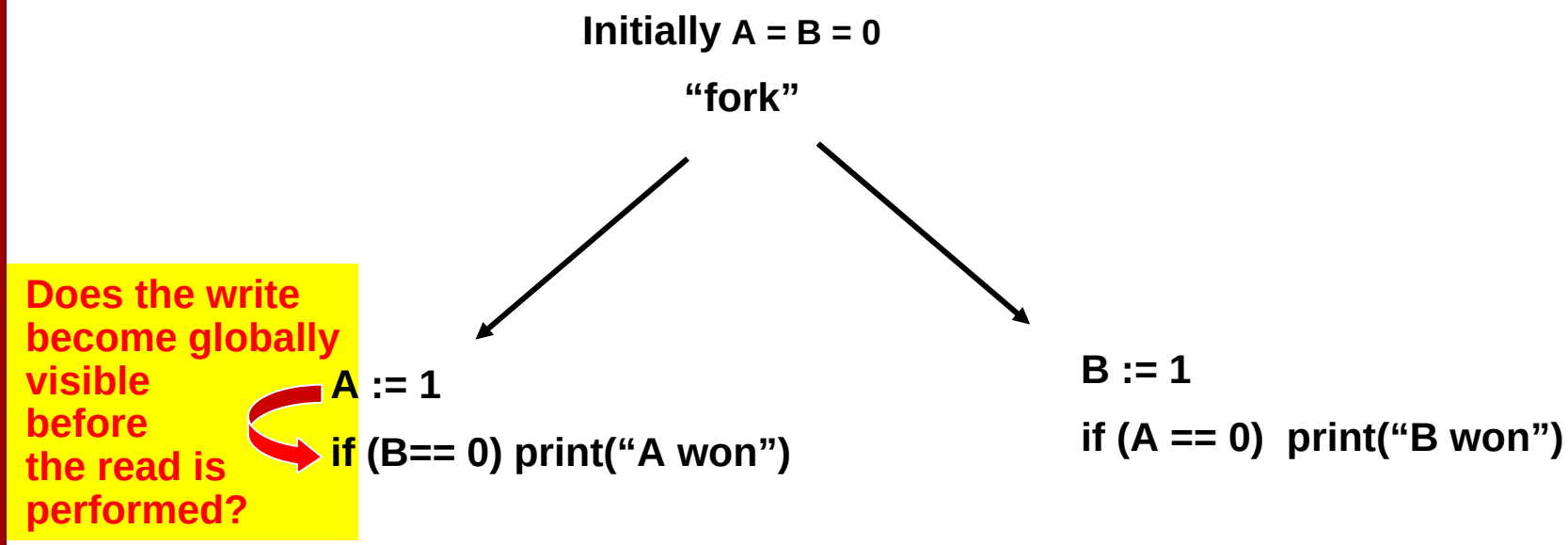
…

While (B==0) {}

Print A

What is the value of A?
It depends...

Dekker's Algorithm



Q: Is it possible that both A and B win?

A: Only known if you know the memory model

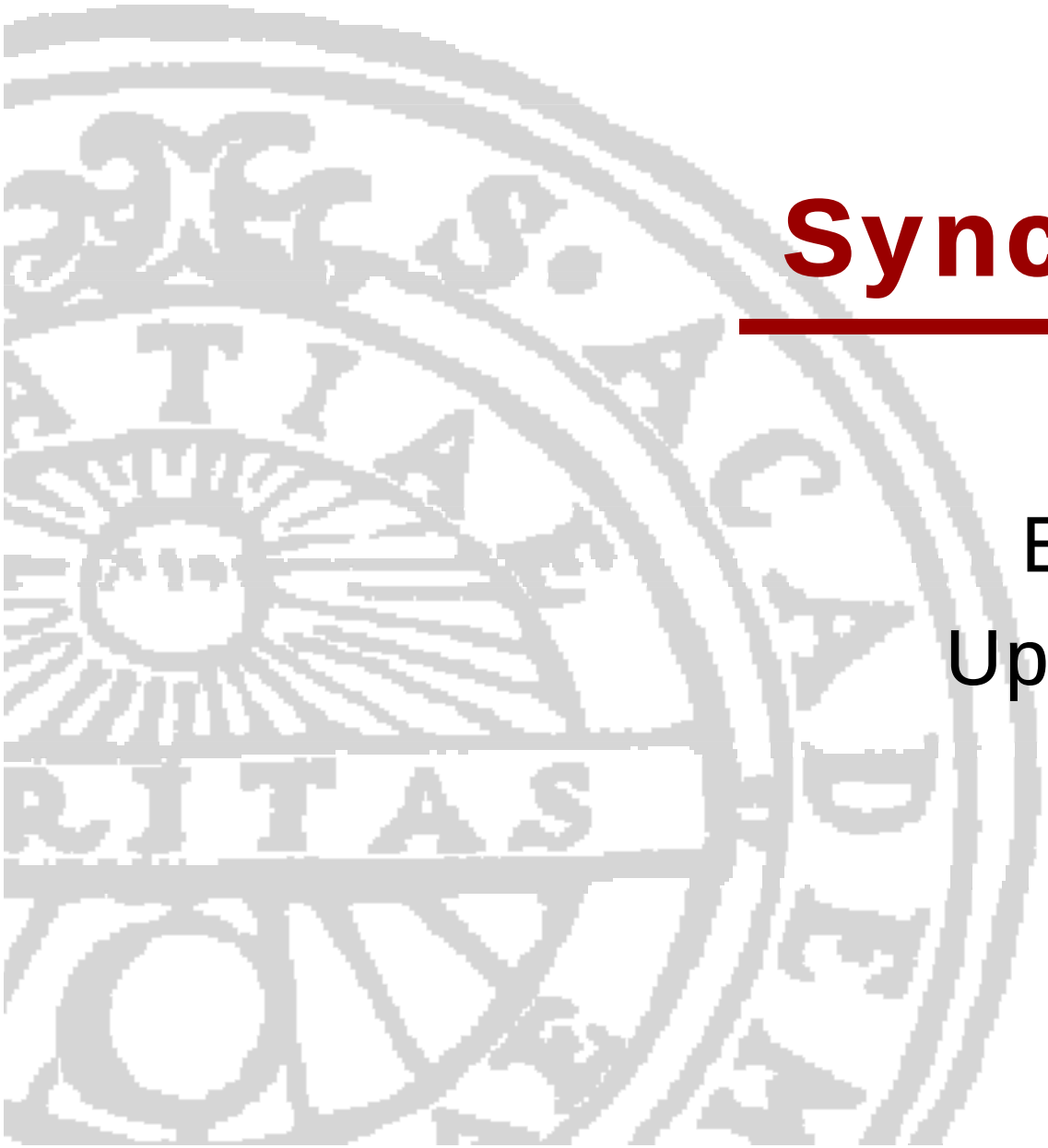
UPPSALA UNIVERSITET

Learning more about memory models

Shared Memory Consistency Models: A Tutorial
by Sarita Adve, Kouroush Gharachorloo
in IEEE Computer 1996 **(in the ”Papers” directory)**

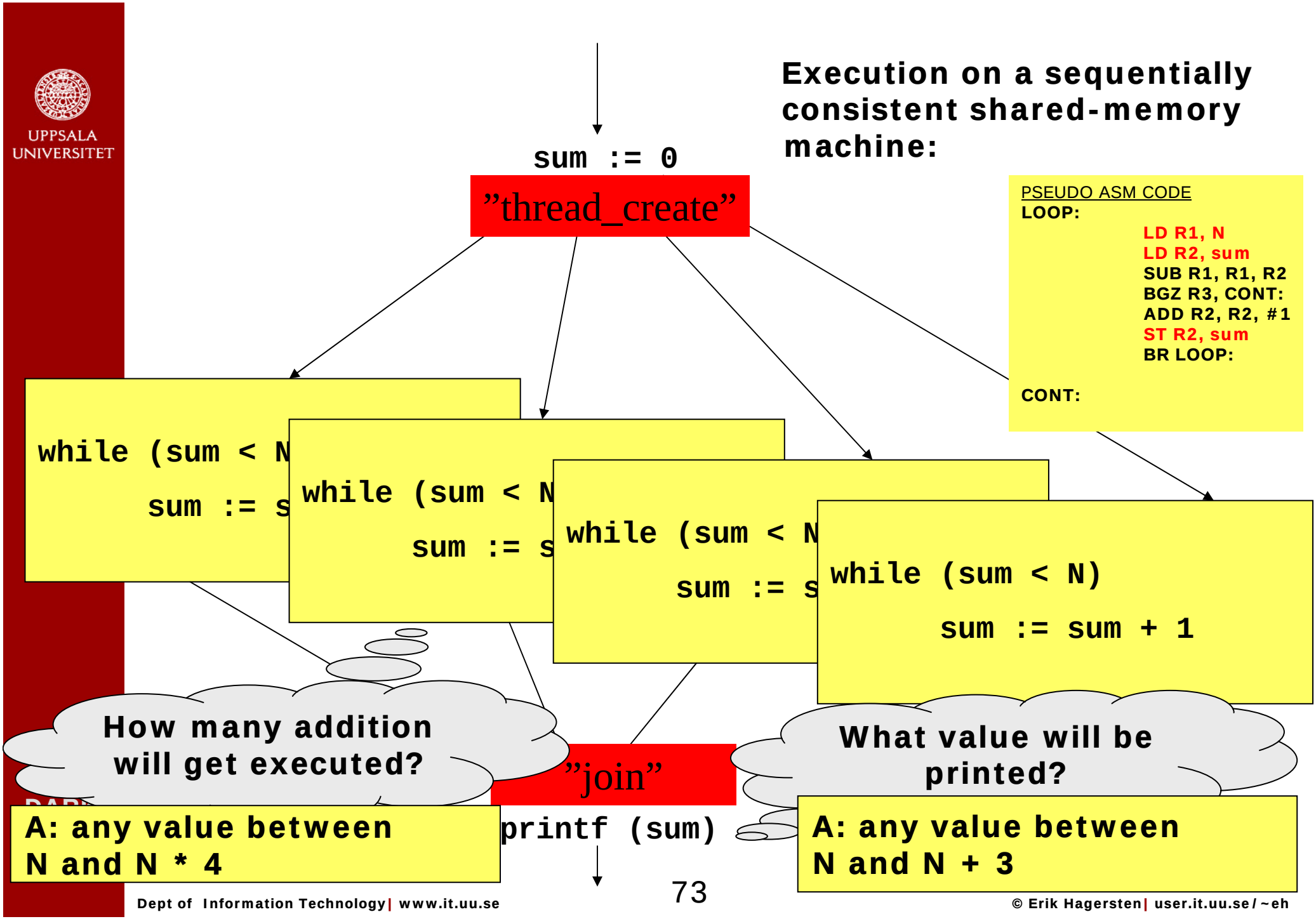
RFM: Read the F*****n Manual of the system you are working on!
(Different microprocessors and systems supports different memory models.)

Issue to think about: What code reordering may compilers really do?
Have to use ”volatile” declarations in C.



Synchronization

Erik Hagersten
Uppsala University
Sweden


Execution on a sequentially consistent shared-memory machine:
sum := 0
”thread_create”
PSEUDO ASM CODE
LOOP:
LD R1, N
LD R2, sum
SUB R1, R1, R2
BGZ R3, CONT:
ADD R2, R2, #1
ST R2, sum
BR LOOP:
CONT:
while (sum < N)
sum := sum + 1
How many addition will get executed?
A: any value between N and N * 4
”join”
printf (sum)
What value will be printed?
A: any value between N and N + 3

Need to introduce synchronization

- Locking primitives are needed to ensure that only one process can be in the critical section:

```
LOCK(lock_variable) /* wait for your turn */
if (sum > threshold) {
    sum := my_sum + sum
}
UNLOCK(lock_variable) /* release the lock*/
```

Critical Section

```
if (sum > threshold) {
        LOCK(lock_variable) /* wait for your turn */
        sum := my_sum + sum
        UNLOCK(lock_variable) /* release the lock*/
}
```

Critical Section

UPPSALA UNIVERSITET

Components of a Synchronization Event

- Acquire method
  - Acquire right to the synch (enter critical section, go past event
- Waiting algorithm
  - Wait for synch to become available when it isn't
- Release method
  - Enable other processors to acquire right to the synch

UPPSALA UNIVERSITET

Atomic Instruction to Acquire

Atomic example: test&set "TAS" (SPARC: LDSTB)

- The value at Mem(lock_addr) loaded into the specified register
- Constant "1" atomically stored into Mem(lock_addr) (SPARC: "FF")
- Software can determin if won (i.e., set changed the value from 0 to 1)
- Other constants could be used instead of 1 and 0

Looks like a store instruction to the caches/memory system

Implementation:

1. Get an exclisive copy of the cache line
2. Make the atomic modification to the cached copy

Other read-modify-write primitives can be used too

- Swap (SWAP): atomically swap the value of REG with Mem(lock_addr)
- Compare&swap (CAS): SWAP if Mem(lock_addr)==REG2

UPPSALA UNIVERSITET

Waiting Algorithms

Blocking

- Waiting processes/threads are de-scheduled
- High overhead
- Allows processor to do other things

Busy-waiting

- Waiting processes repeatedly test a lock_variable until it changes value
- Releasing process sets the lock_variable
- Lower overhead, but consumes processor resources
- Can cause network traffic

Hybrid methods: busy-wait a while, then block

UPPSALA UNIVERSITET

Release Algorithm

- Typically just a store ”0”
- More complicated locks may require a conditional store or a ”wake-up”.

DARK 2008

A Bad Example: ”POUNDING”

```
proc lock(lock_variable) {
    while (TAS[lock_variable]==1)  {}      /* bang on the lock until free */
}

proc unlock(lock_variable) {
     lock_variable := 0
}
```

Assume: The function TAS (test and set)

*-- returns the current memory value and **atomically***

writes the busy pattern “1” to the memory

Generates too much traffic!!
-- spinning threads produce traffic!

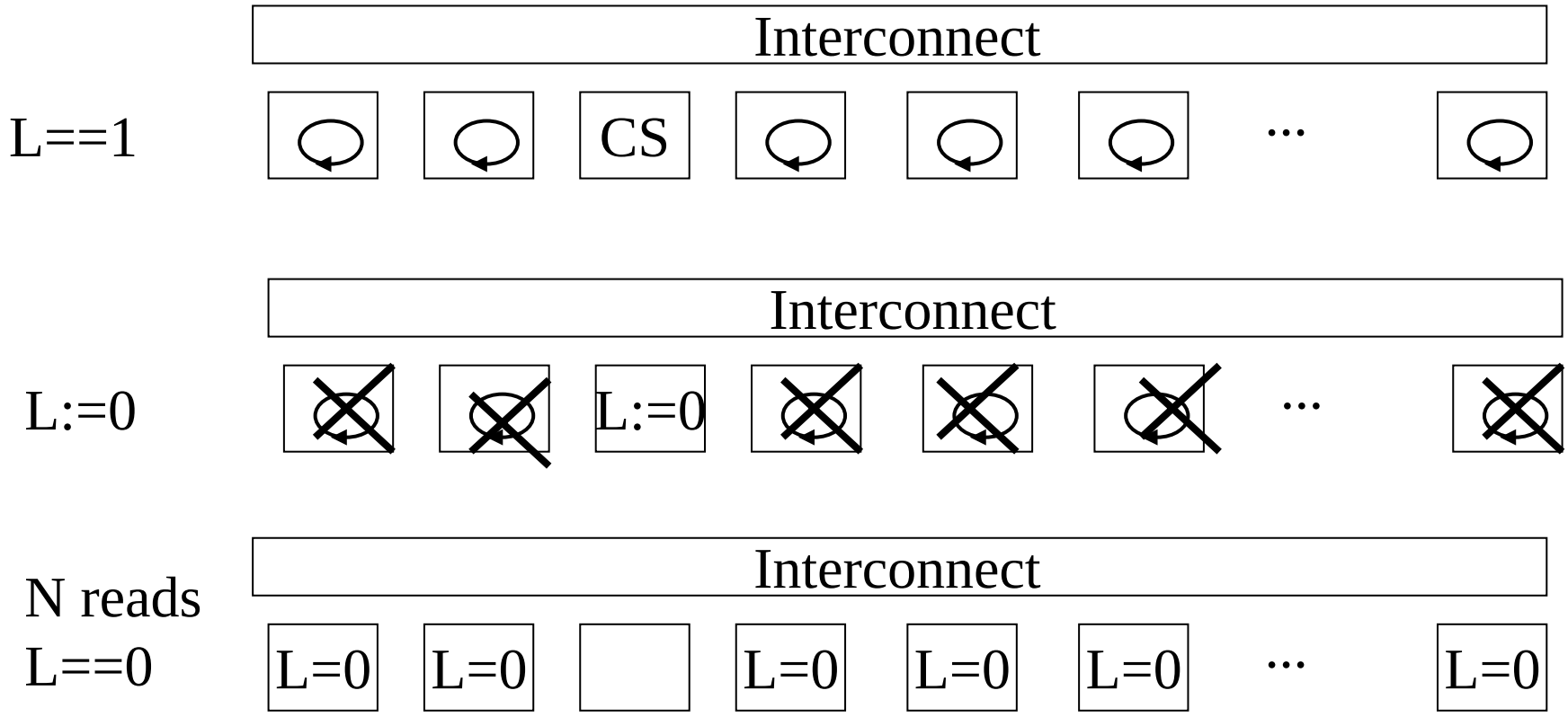
UPPSALA UNIVERSITET

Optimistic Test&Set Lock ”spinlock”

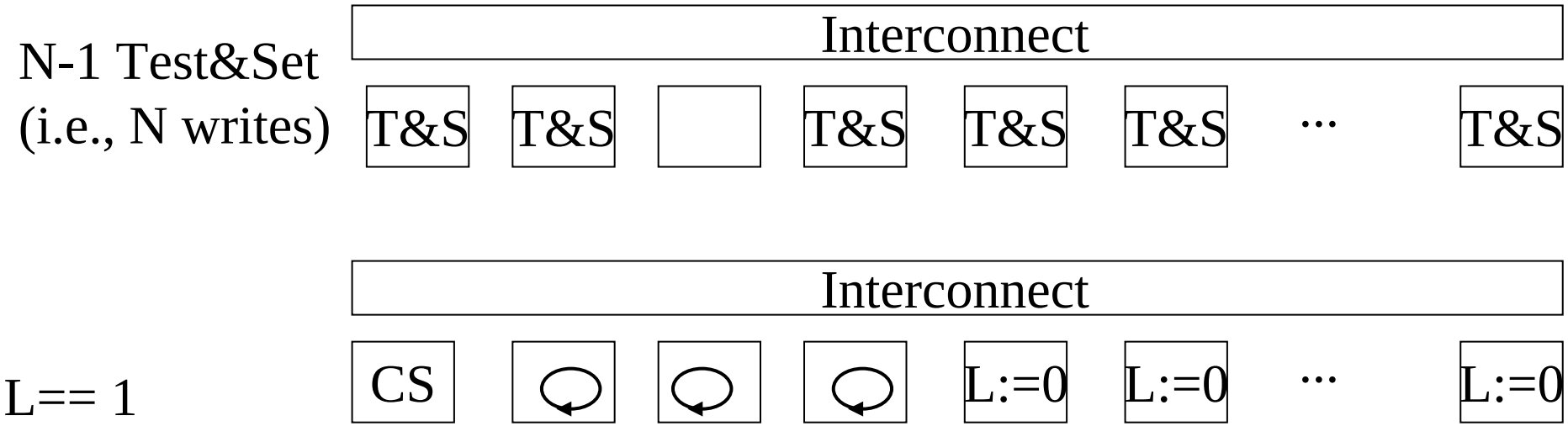
```
proc lock(lock_variable) {
    while true {
      if (TAS[lock_variable] ==0)  break;     /* bang on the lock once, done if TAS==0 */
      while(lock_variable != 0) {}       /* spin locally in your cache until ”0” observed*/
   }
}

proc unlock(lock_variable) {
     lock_variable := 0
}
```

Much less coherence traffic!!
-- still lots of traffic at lock handover!

It could still get messy!



...messy (part 2)



potentially: ~N*N/2 reads :-(

Problem1: Contention on the interconnect slows down the CS proc
Problem2: The lock hand-over time is N*read_throughput
Fix1: Some back-off strategy, bad news for hand-over latency
Fix1: Queue-based locks

UPPSALA UNIVERSITET

Could Get Even Worse on a NUMA

- Poor communication latency
- Serialization of accesses to the same cache line
- WF: added hardware optimization:
  - TAS can bypass loads in the coherence protocol

  ==>N-2 loads queue up in the protocol

  ==> the winner's atomic TAS will bypass the loads

  ==>the loads will return "busy"

UPPSALA UNIVERSITET

Ticket-based queue locks: ”ticket”

```
proc lock(lstruct) {
        int my_num;
        my_num := INC(lstruct.ticket)      /* get your unique number*/
        while(my_num != lstruct.nowserving) {}   /* wait here for your turn */
}

proc unlock(lstruct) {
        lstruct.nowserving++                        /* next in line please */
}
```

Less traffic at lock handover!

Ticket-based back-off ”TBO”

```
proc lock(lstruct) {
        int my_num;
        my_num := INC(lstruct.ticket)                  /* get your number*/
        while(my_num != lstruct.nowserving) {          /* my turn ?*/
            idle_wait(lstruct.nowserving - my_num)     /* do other shopping */
        }
}

proc unlock(lock_struct) {
        lock_struct.nowserving++                       /* next in line please */}
```

Even less traffic at lock handover!

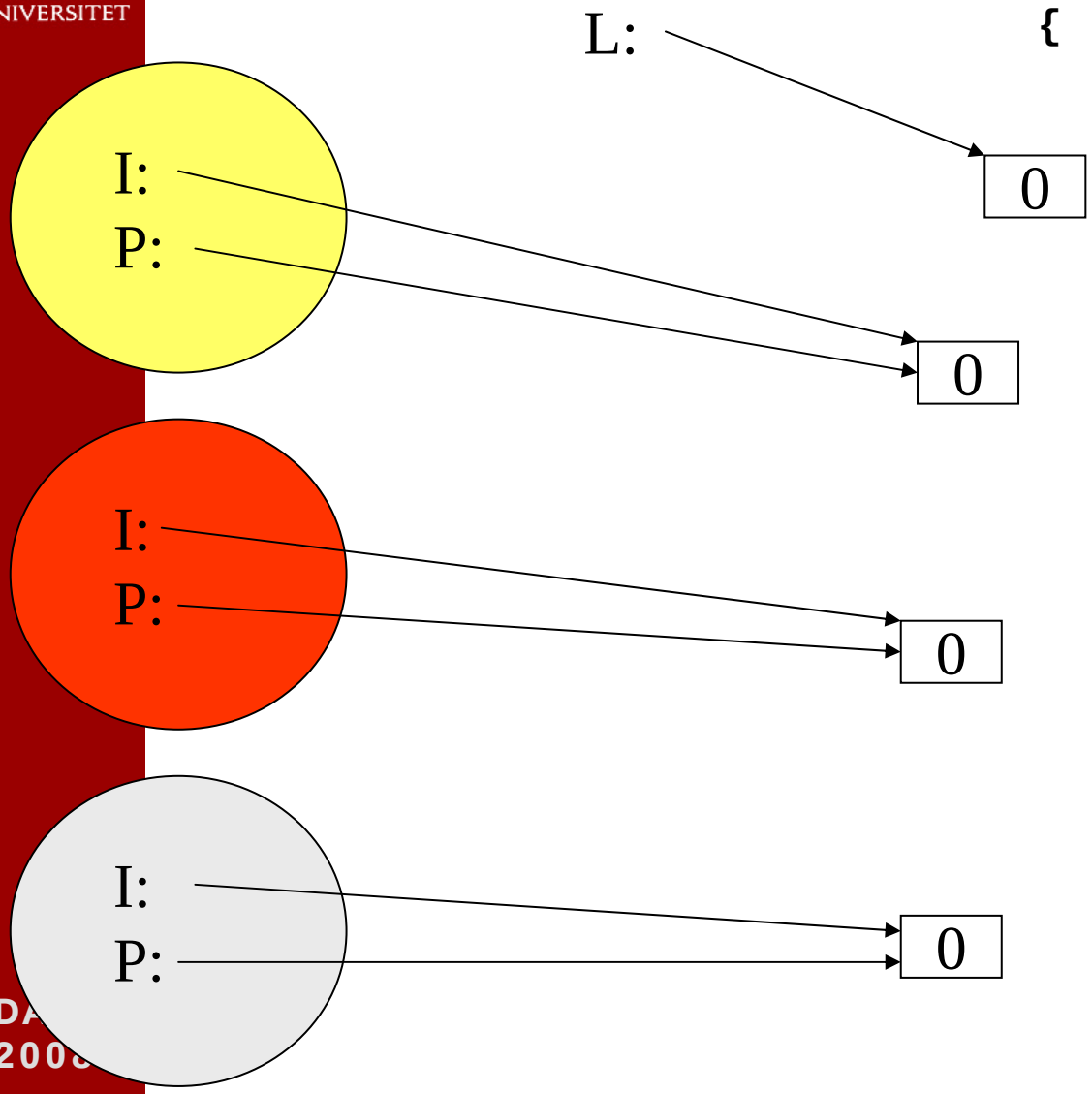
UPPSALA UNIVERSITET

Queue-based lock: CLH-lock -- a variation of the MCS lock

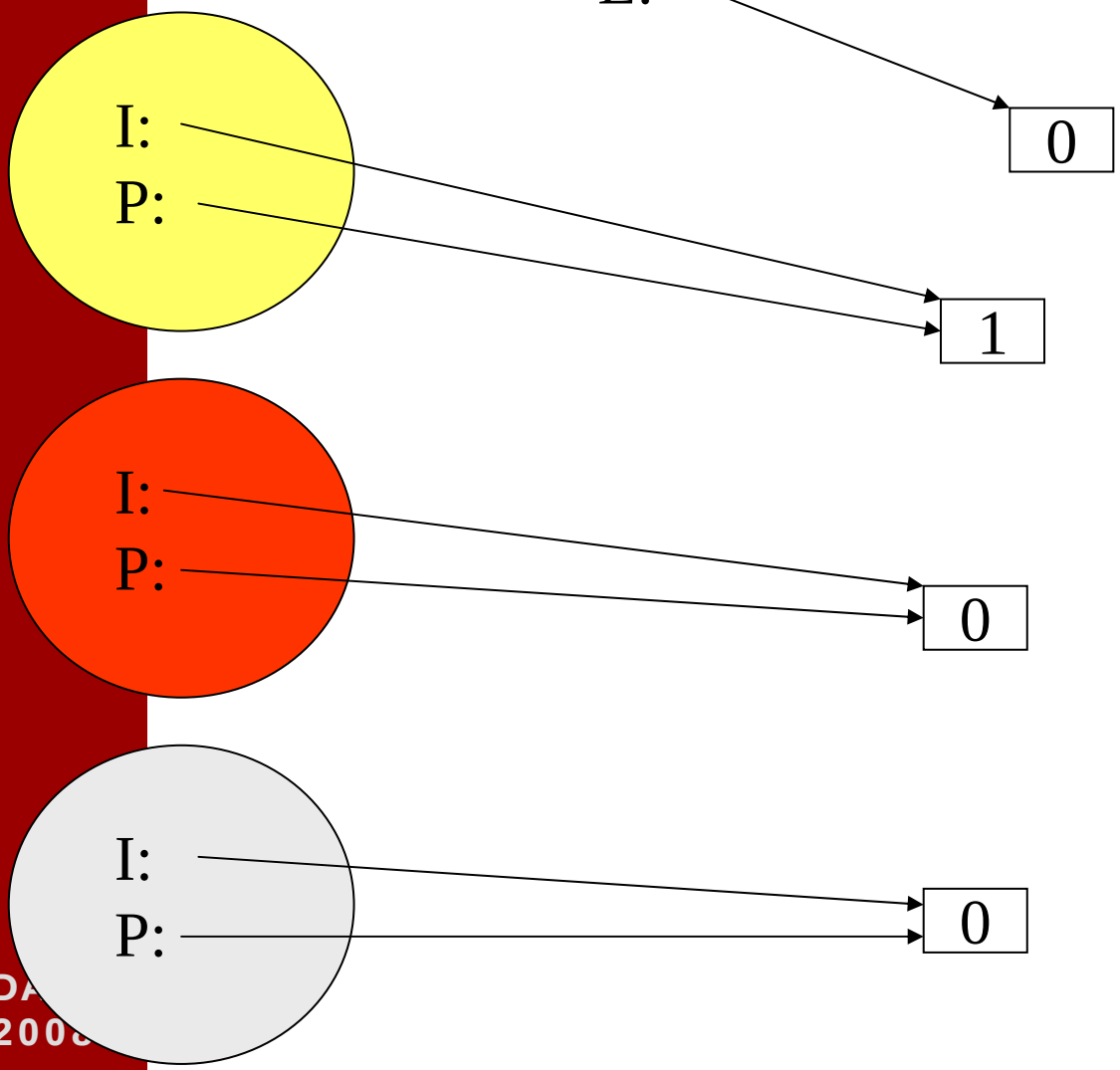
"Initially, each process owns one global flag, pointed to by private *I and *P
The global lock is pointed to by global *L
1) Initialize the *I flag to busy (= "1")
2) Atomically, make *L point our "flag" and make *P point where *L pointed
3) Wait until *P points to a "0"

```
proc lock(int **L, **I, **P)
{   **I =1;                  /*initialized as "busy"*/
    atomic_swap {*P =*L; *L=*P}
                 /* P now stores a pointer to the flag L pointed to*/
                 /* L now stores a pointer to our flag */
    while (**P != 0){} }/* keep spinning until prev owner releases lock*/


proc unlock(int **I, **P)
{    **I =0;       /* release the lock */
     *I =*P; }     /* next time *I to reuse the previous guys flag*/
```

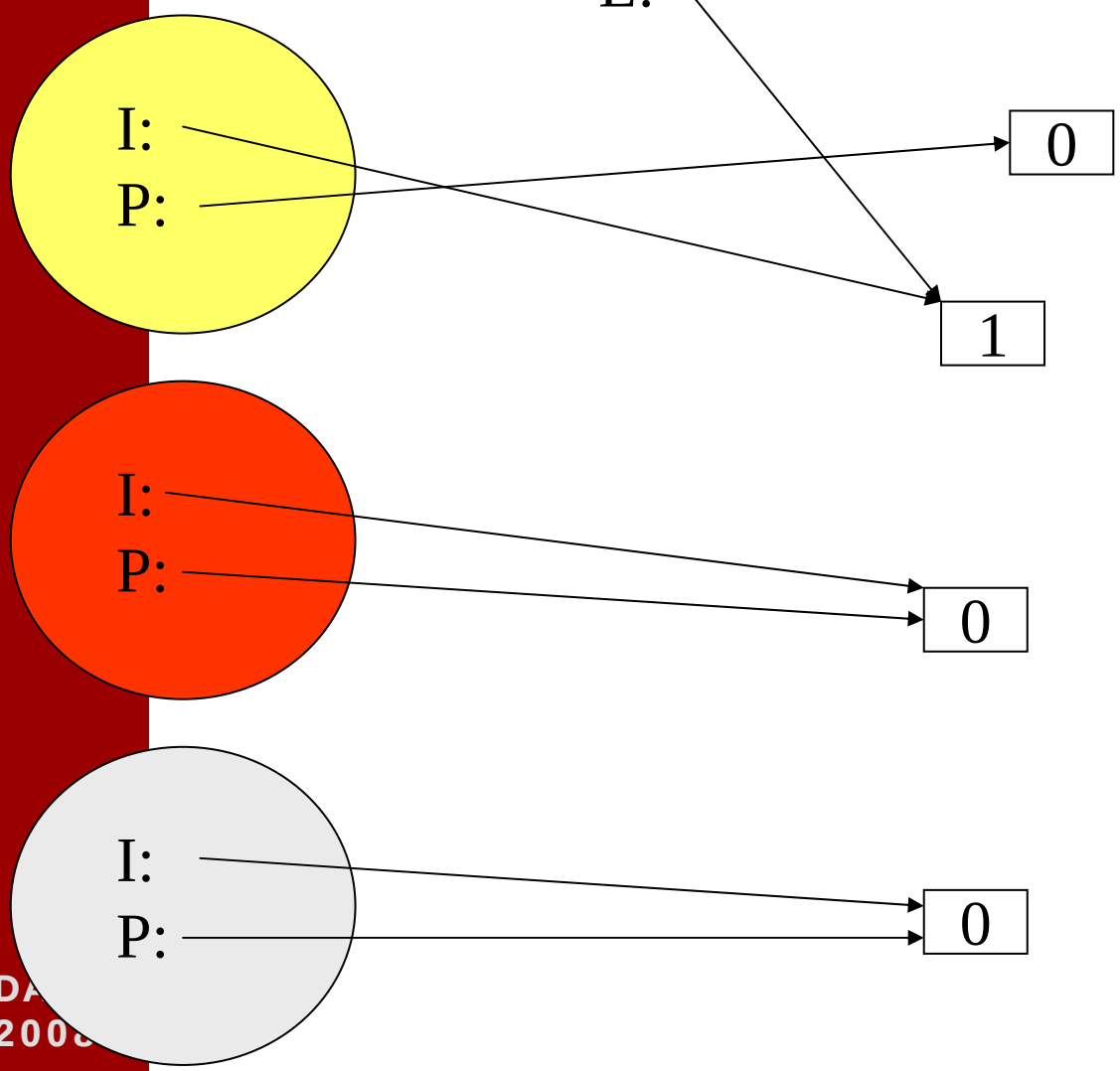
CLH lock



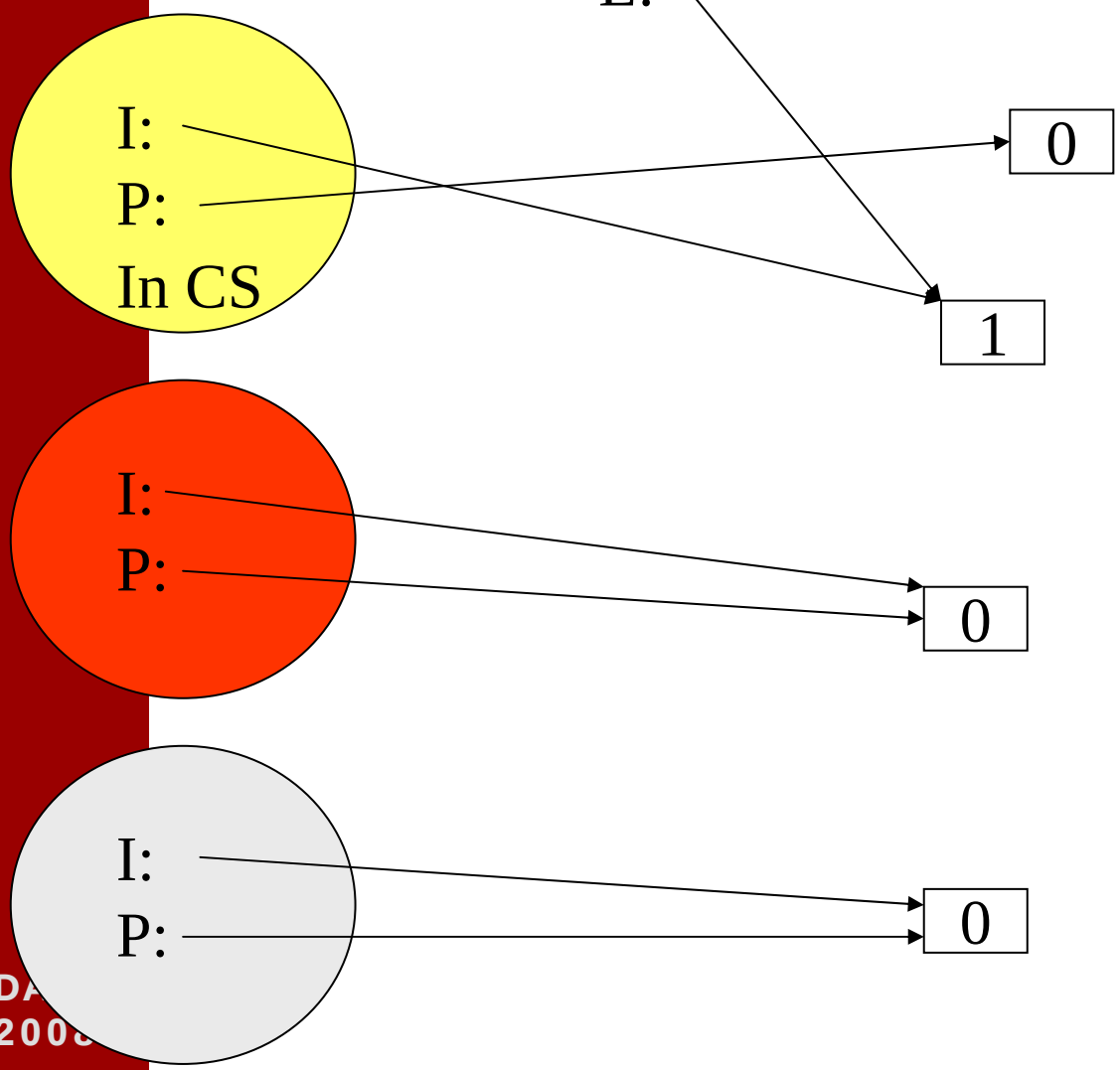

```
proc lock(int **L, **I, **P)
{       **I =1 /* init to "busy"*/
        atomic_swap {*P =*L; *L=*P}
      /* *L now points to our I* */
        while (**P != 0){}  }
      /* spin unit prev is done */
```



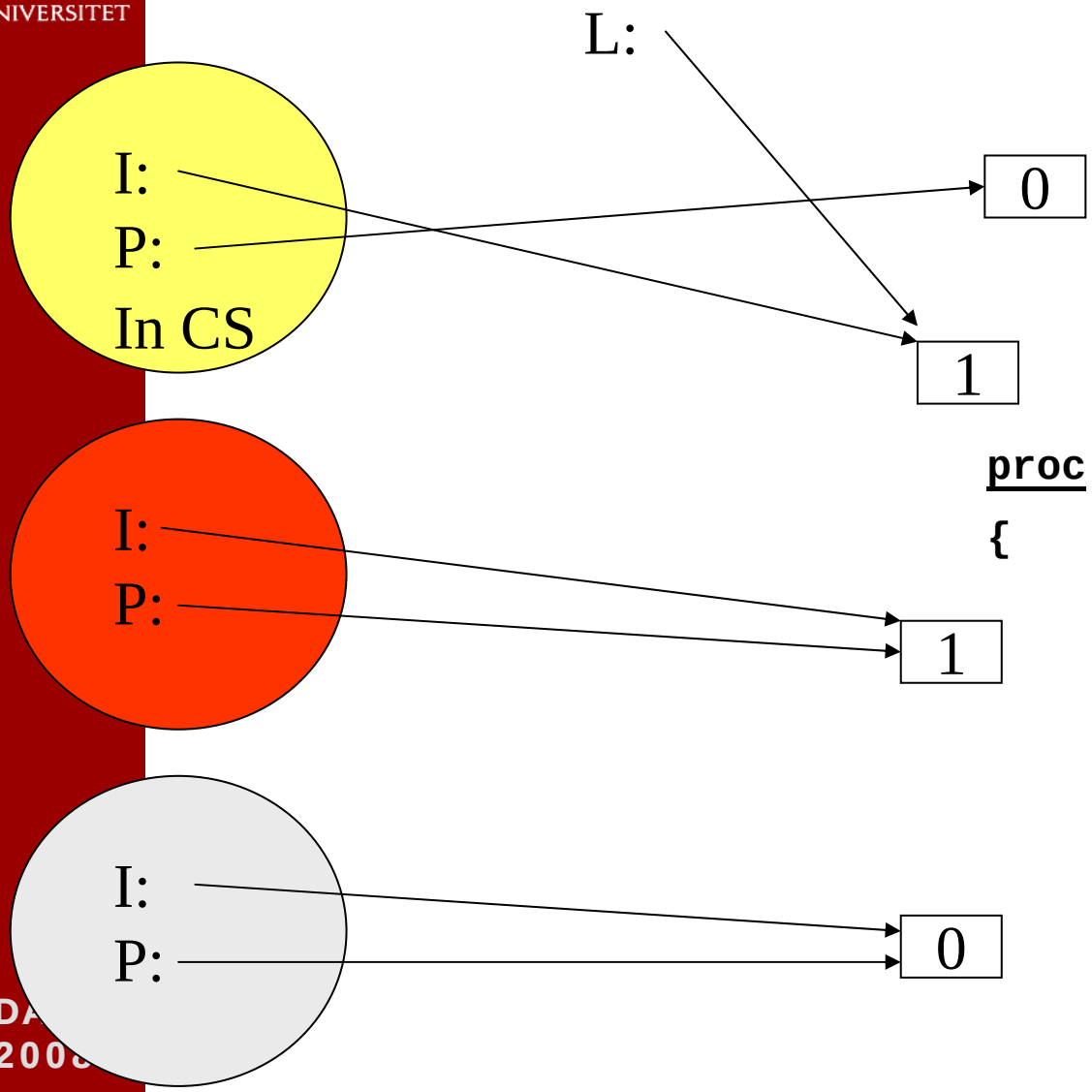

```
proc lock(int **L, **I, **P)
{       **I =1 /* init to "busy"*/
        atomic_swap {*P =*L; *L=*P}
/* *L now point to our I* */
        while (**P != 0){} }
/* spin unit prev is done */
```



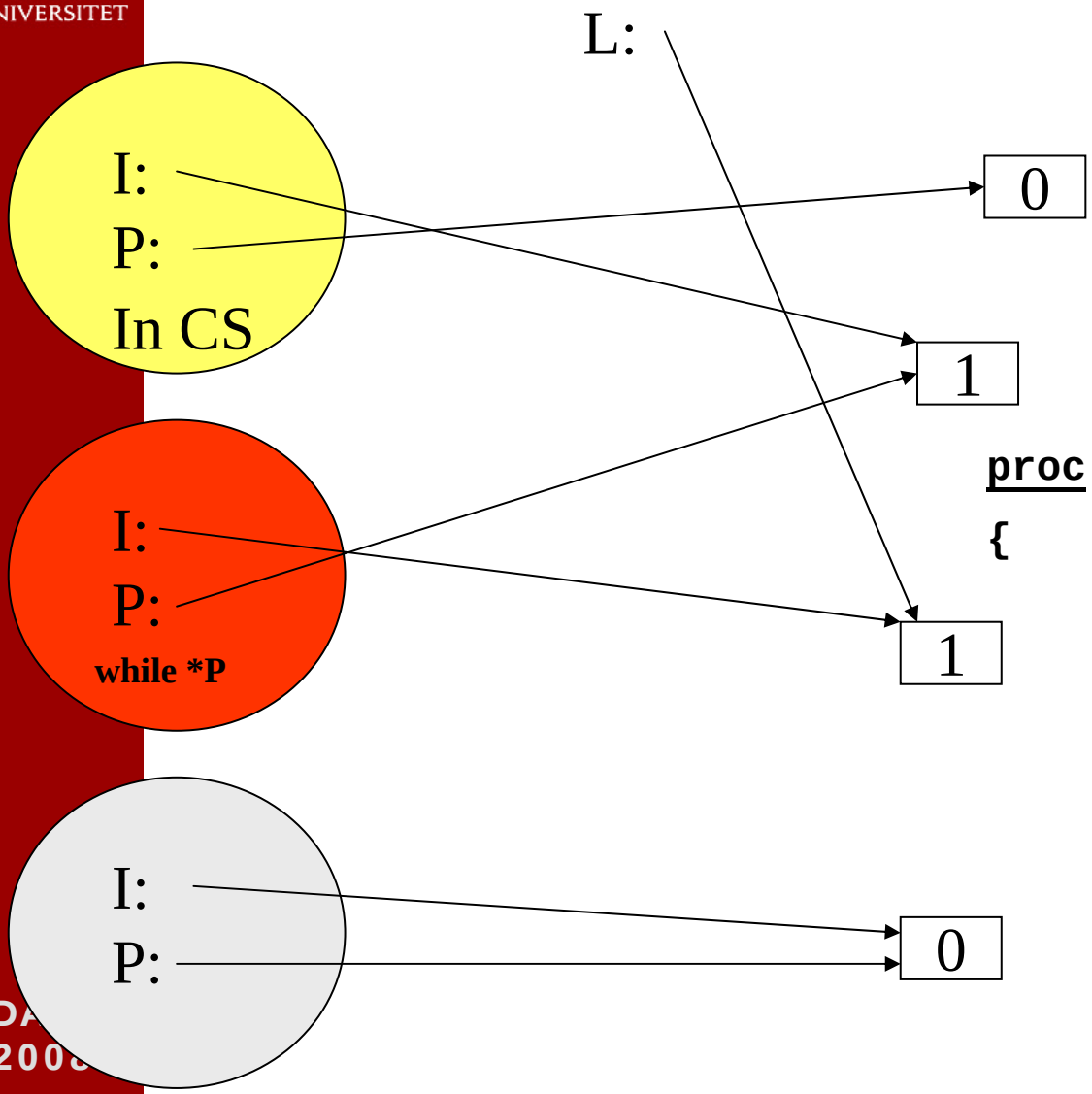

```
proc lock(int **L, **I, **P)
{       **I =1 /* init to "busy"*/
        atomic_swap {*P =*L; *L=*P}
      /* *L now point to our I* */
        while (**P != 0){} };
      /* spin unit prev is done */
```



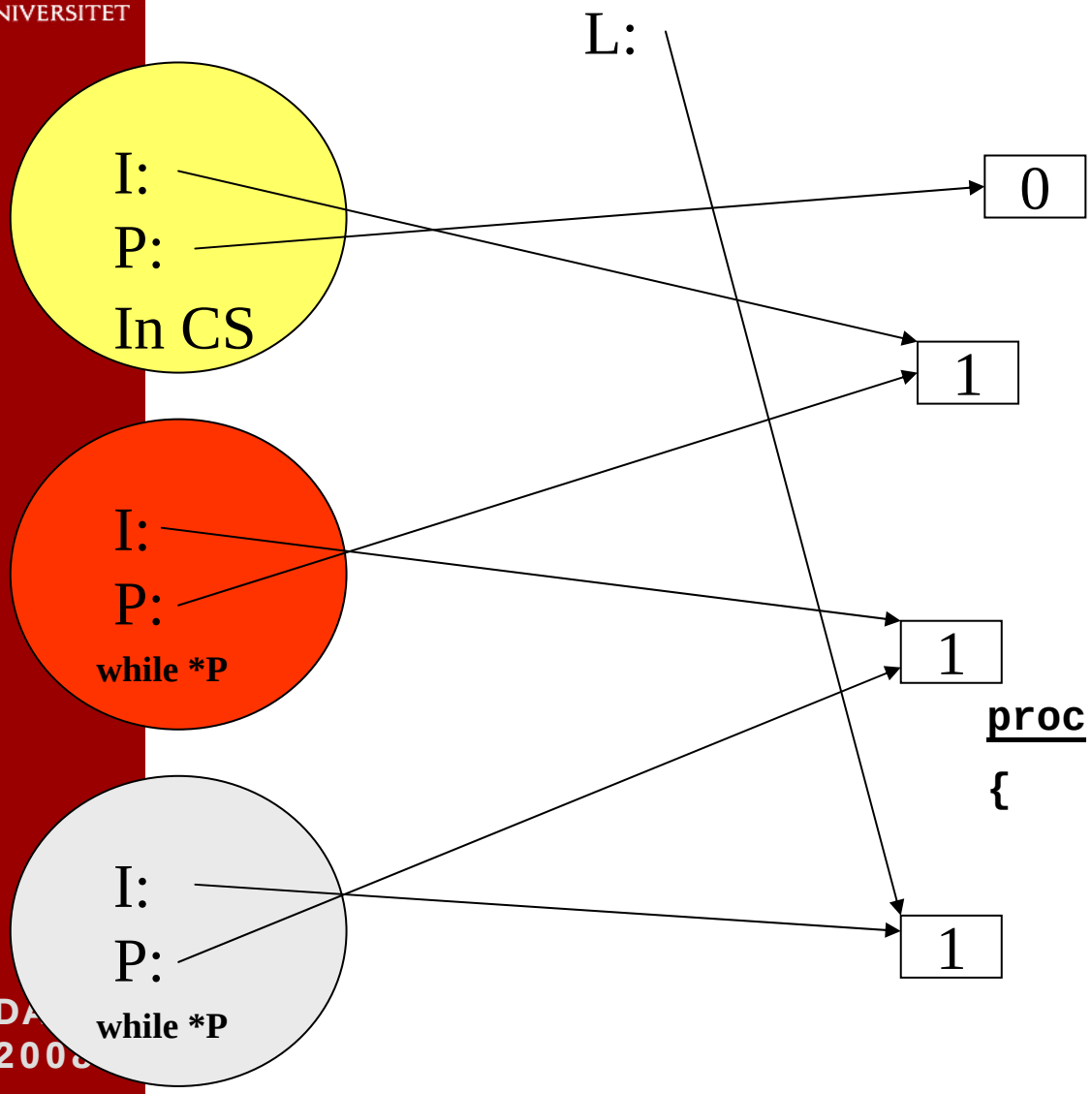

```
proc lock(int **L, **I, **P)
{       **I =1 /* init to "busy"*/
        atomic_swap {*P =*L; *L=*P}
/* *L now point to our I* */
        while (**P != 0){} };
/* spin unit prev is done */
```



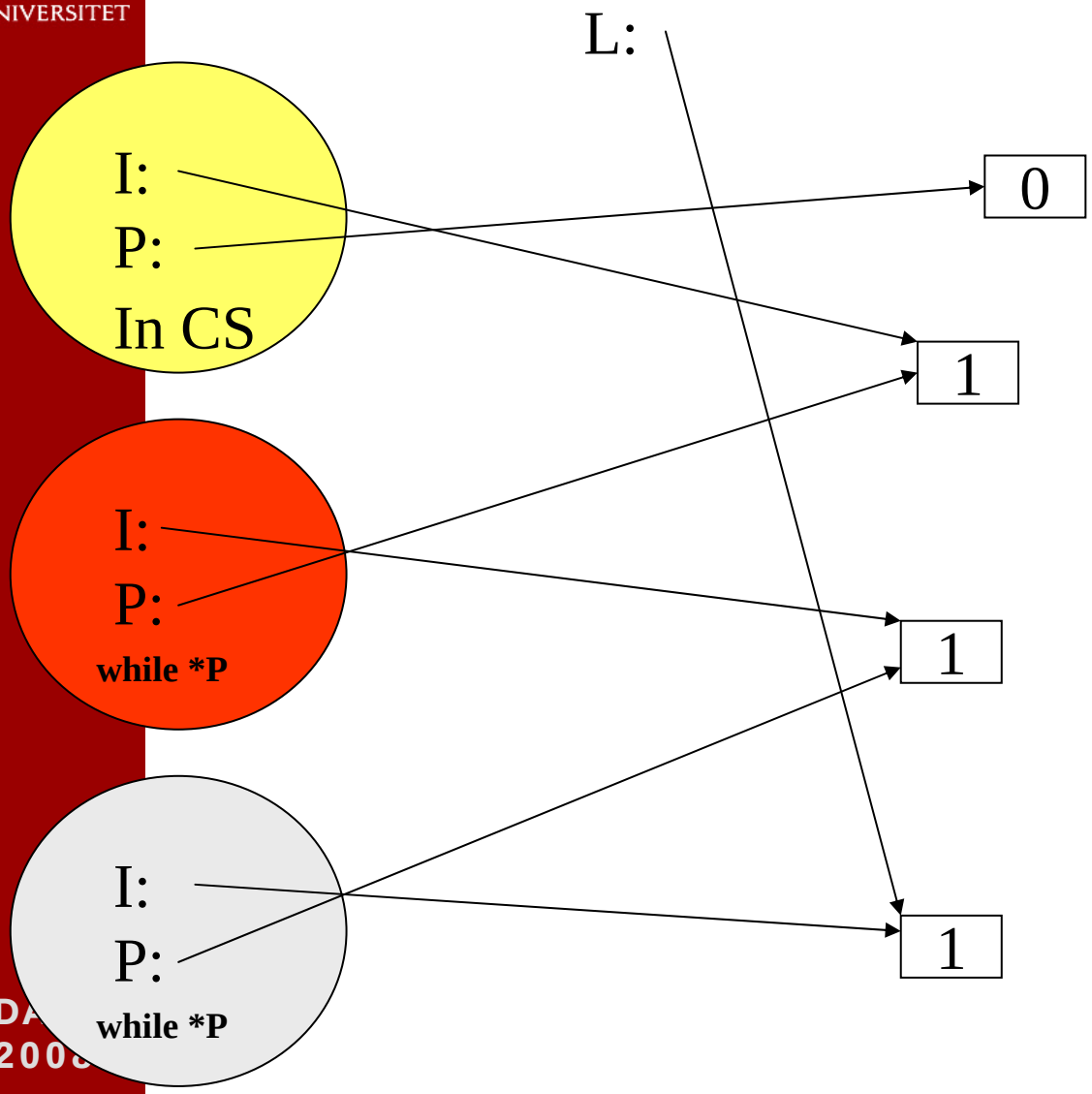

```
proc lock(int **L, **I, **P)
{       **I =1 /* init to "busy"*/
        atomic_swap {*P =*L; *L=*P}
      /* *L now point to our I* */
        while (**P != 0){} };
      /* spin unit prev is done */
```



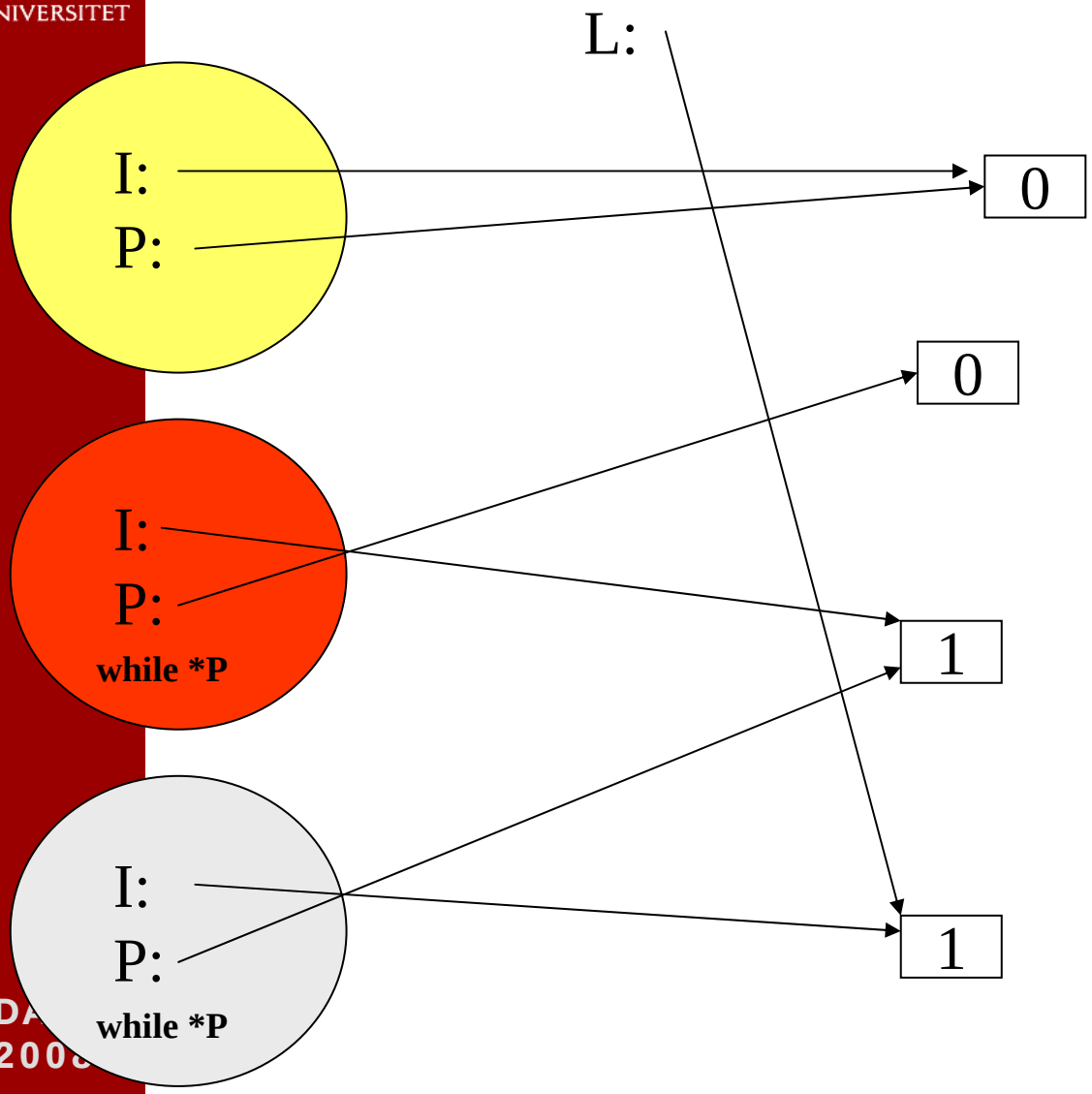

```
proc lock(int **L, **I, **P)
{       **I =1 /* init to "busy"*/
        atomic_swap {*P =*L; *L=*P;}
      /* *L now point to our I* */
        while (**P != 0){} };
      /* spin unit prev is done */
```



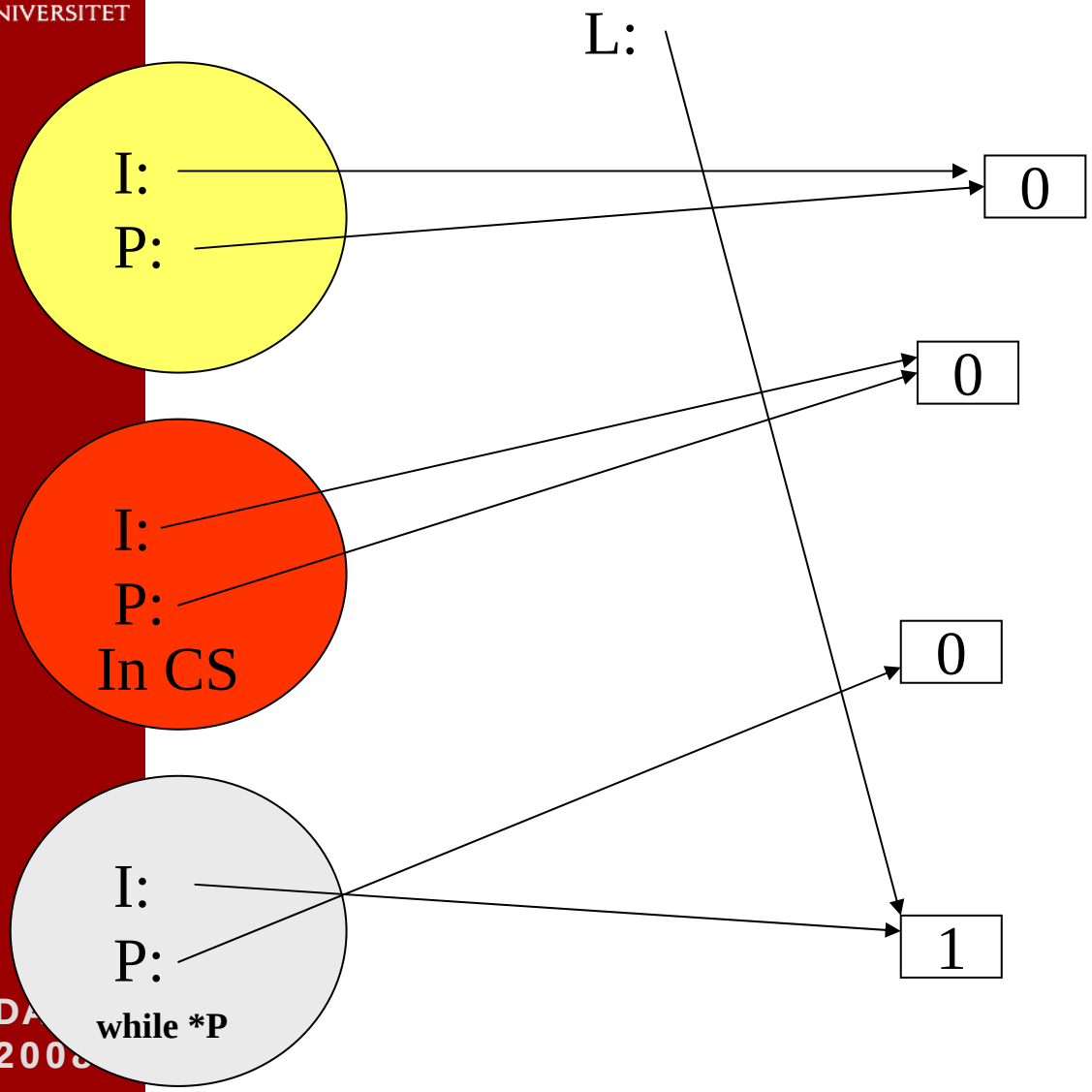

```
proc lock(int **L, **I, **P)
{       **I =1 /* init to "busy"*/
        atomic_swap {*P =*L; *L=*P;}
       /* *L now point to our I* */
        while (**P != 0){} };
       /* spin unit prev is done */
```



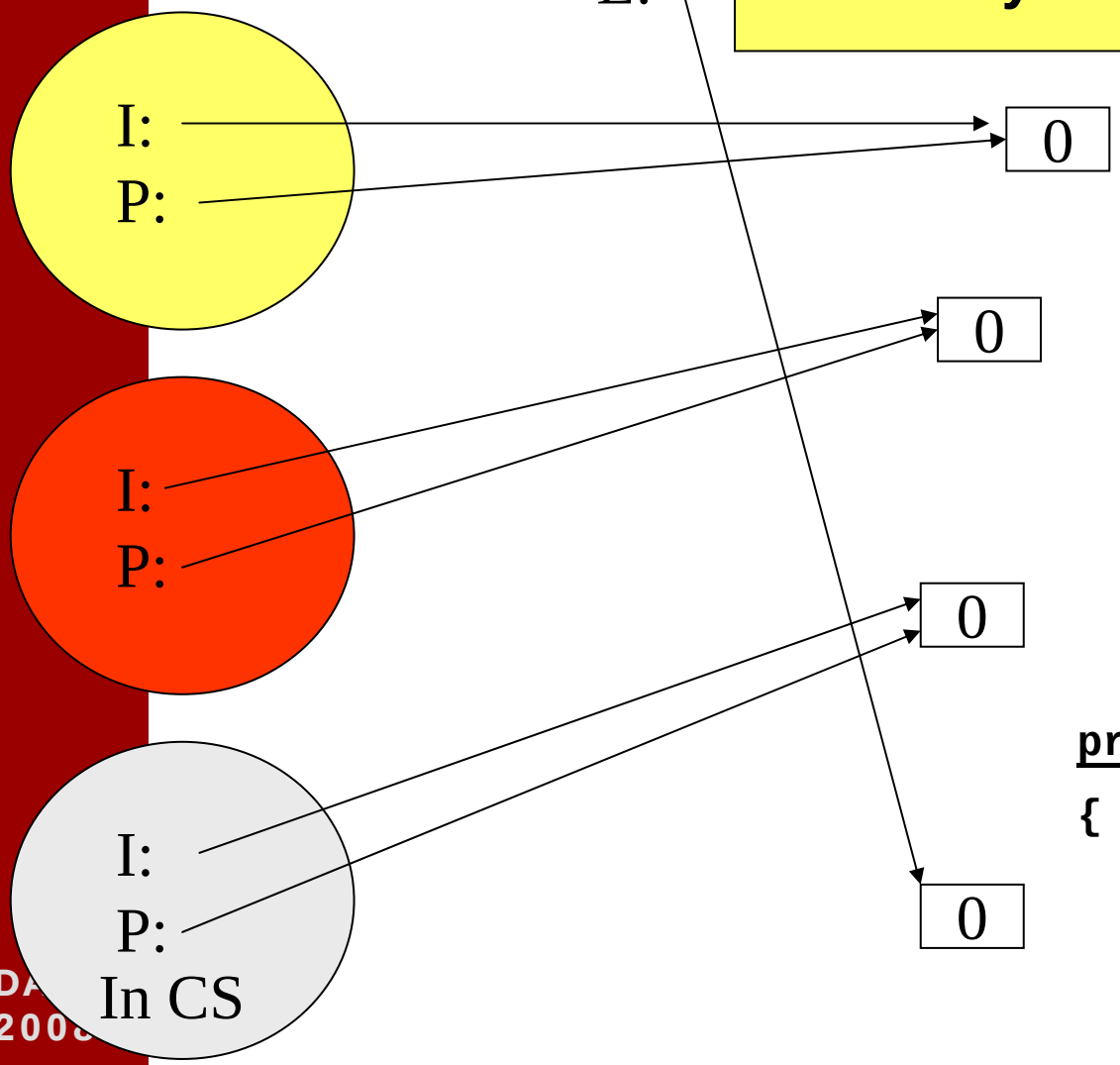

```
proc unlock(int **I, **P)
{       **I = 0;
        /* release the lock */
        *I = *P; }
        /* reuse the previous guy's *P*/
```




```
proc unlock(int **I, **P)
{       **I = 0;
        /* release the lock */
        *I = *P; }
        /* reuse the previous guy's *P*/
```



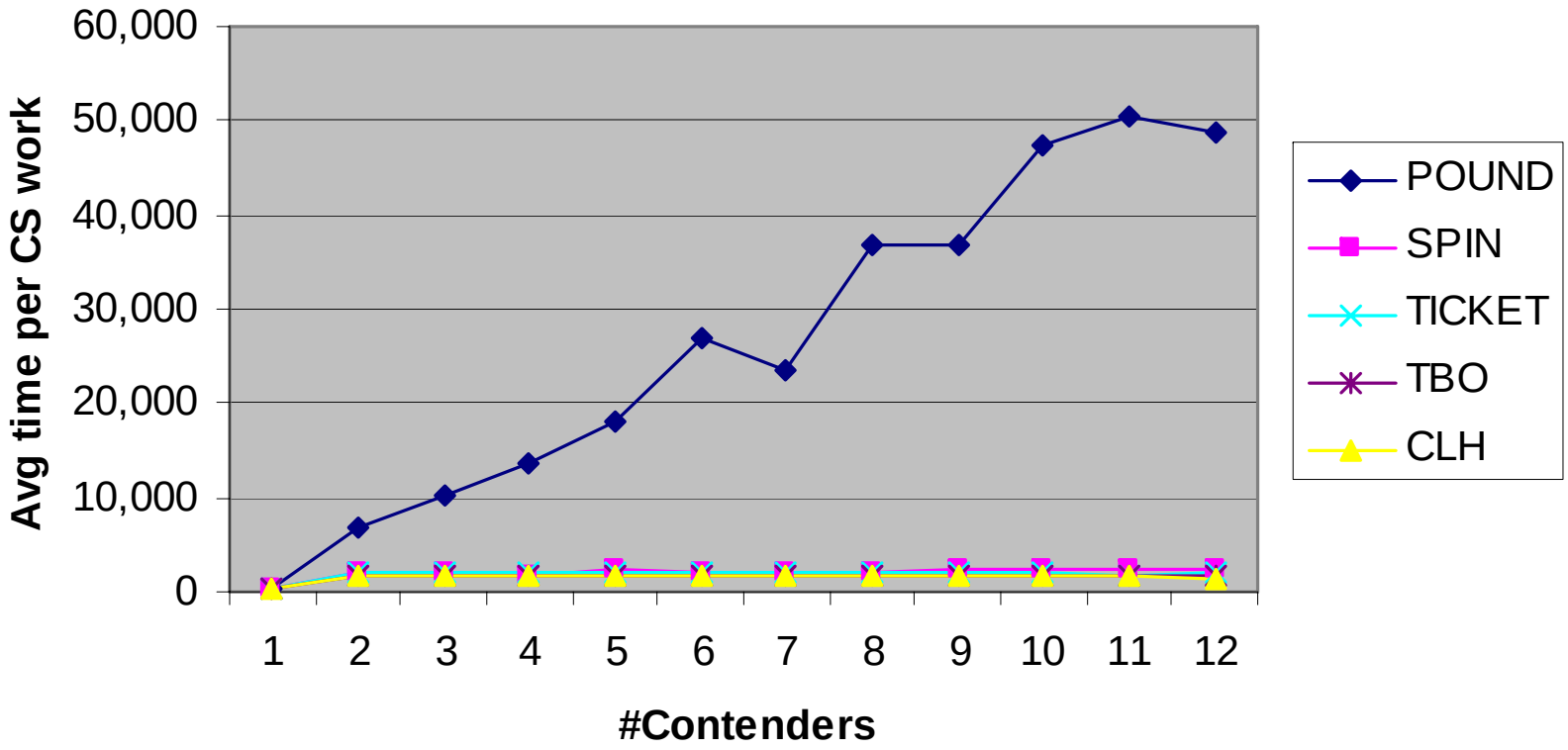

```
proc unlock(int **I, **P)
{       **I = 0;
        *I = *P; }
```



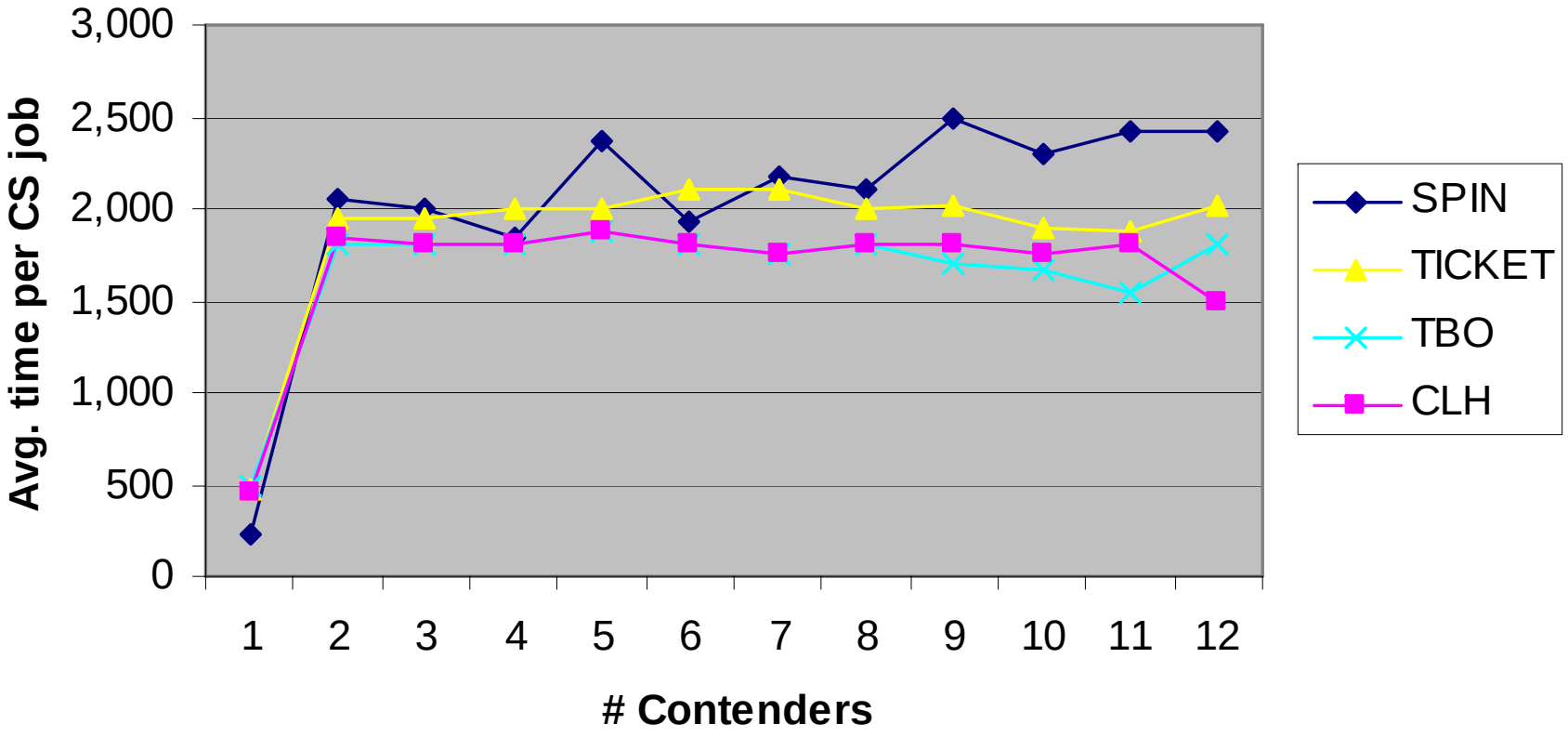
Minimizes traffic at lock handover!
May be too fair for NUMAs ….

```
proc unlock(int **I, **P)
{       **I = 0;
        *I = *P; }
```

E6800 locks 12 CPUs

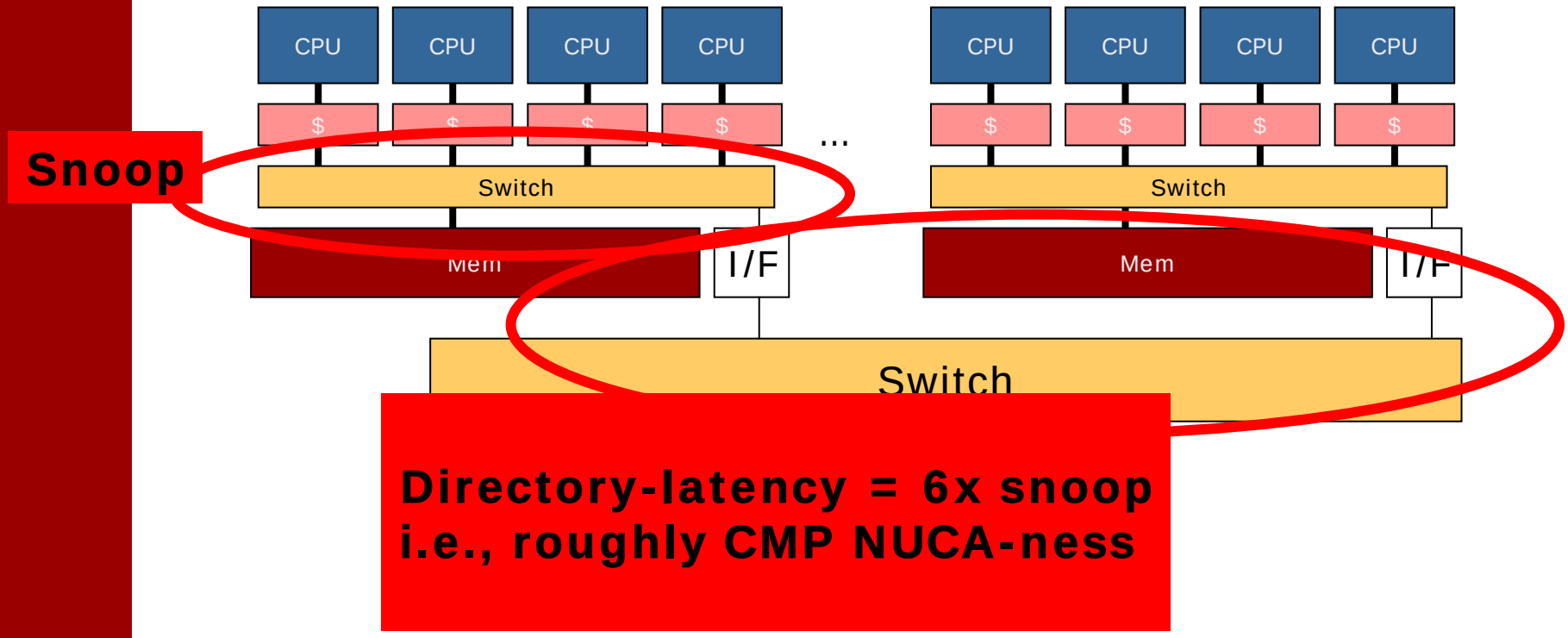


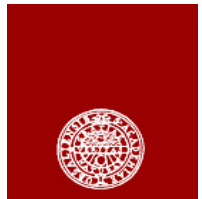
E6800 locks (exluding POUND)



NUMA:

NUCA: Non-uniform Comm Arch.

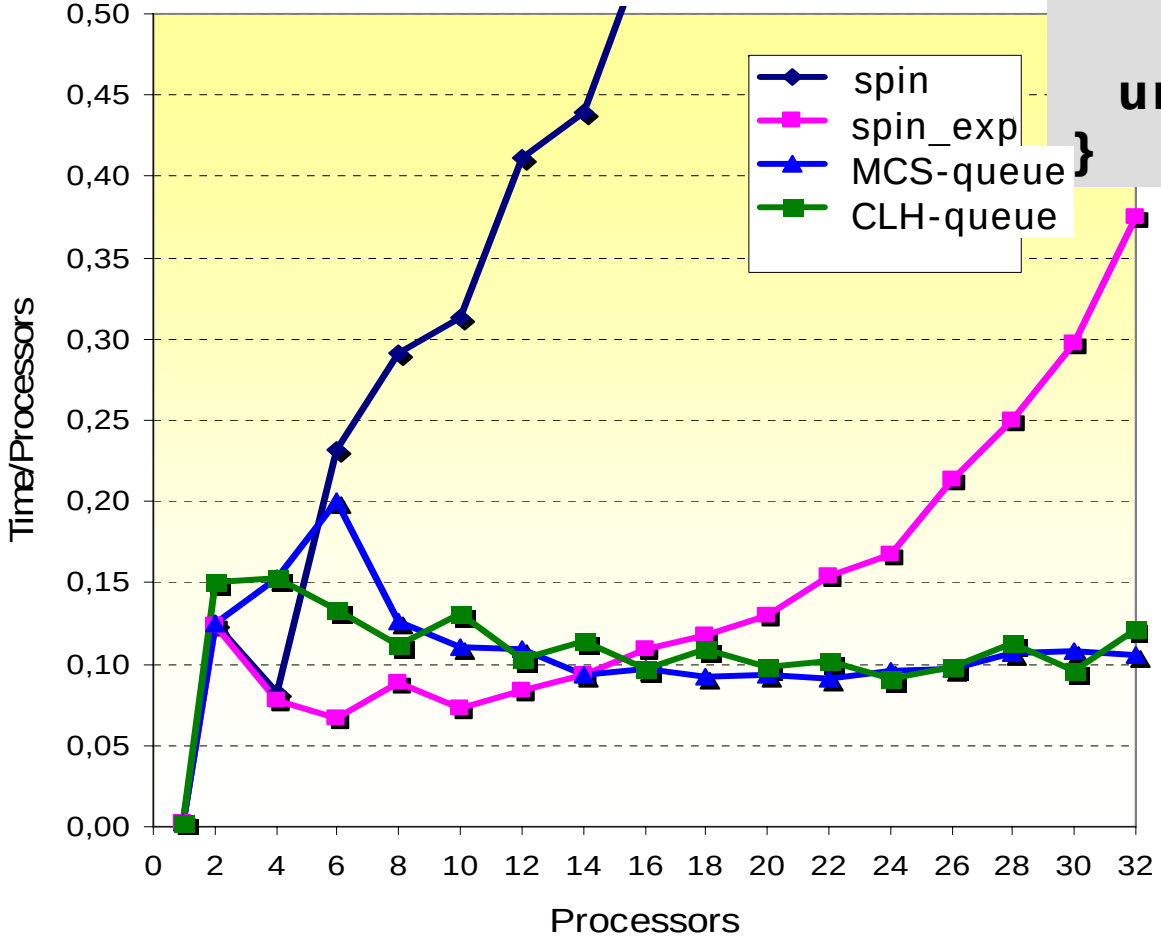




Trad. chart over lock performance on a hierarchical NUMA (round robin scheduling)

Benchmark:

```
for i = 1 to 10000 {
   lock(AL)
      A:= A + 1;
   unlock(AL)
}
```



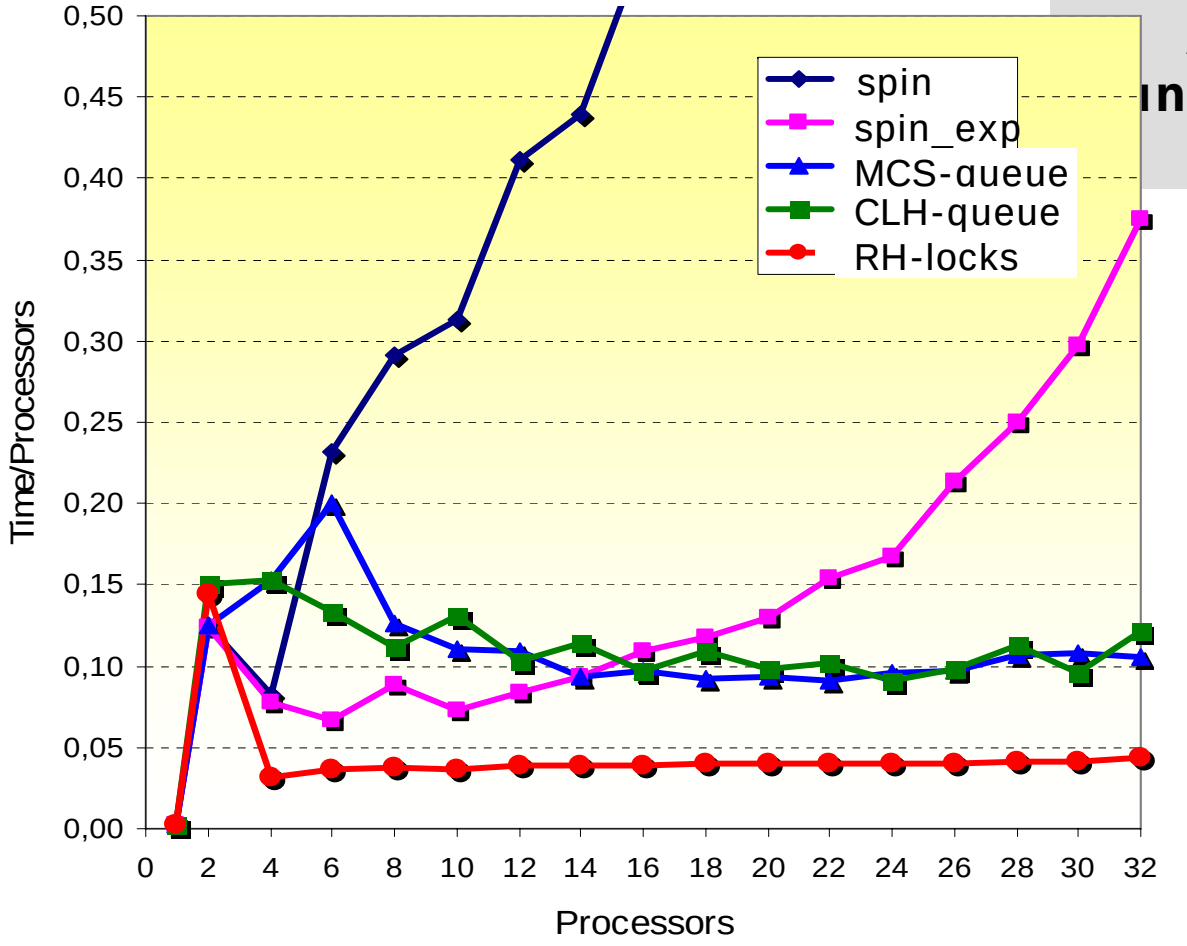
DARK 2008

UPPSALA UNIVERSITET

Introducing RH locks

Benchmark:

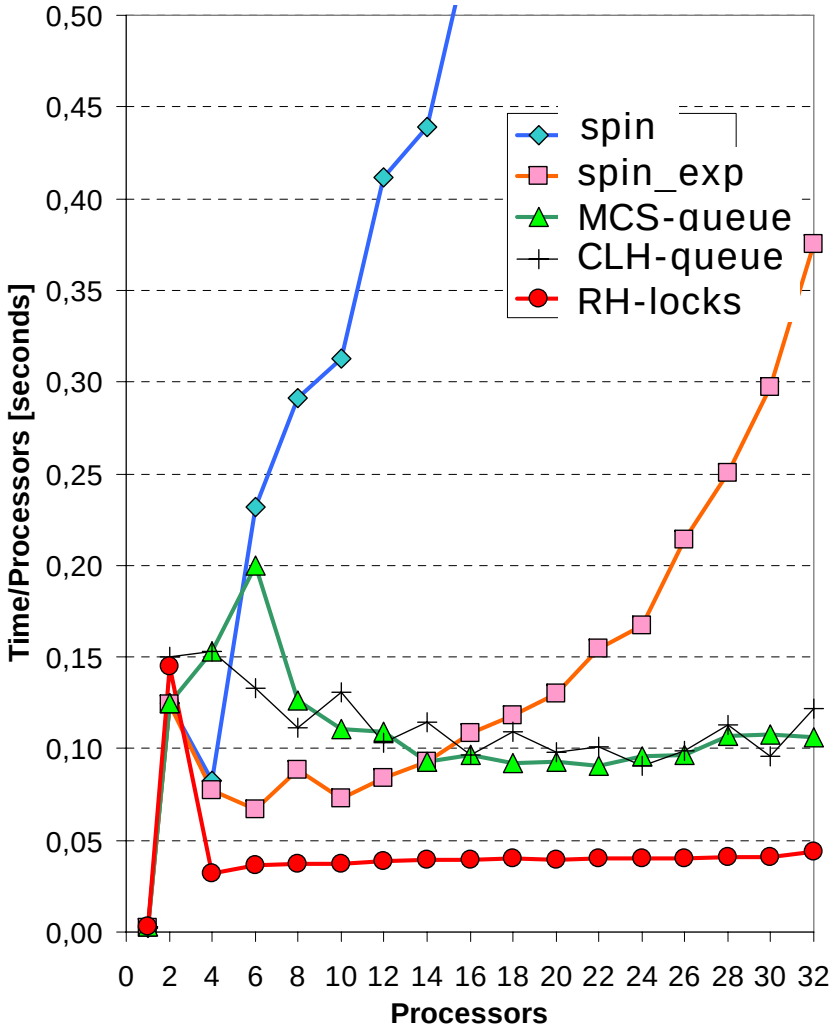
```
for i = 1 to 10000 {
   lock(AL)
      A:= A + 1;
   unlock(AL)
```



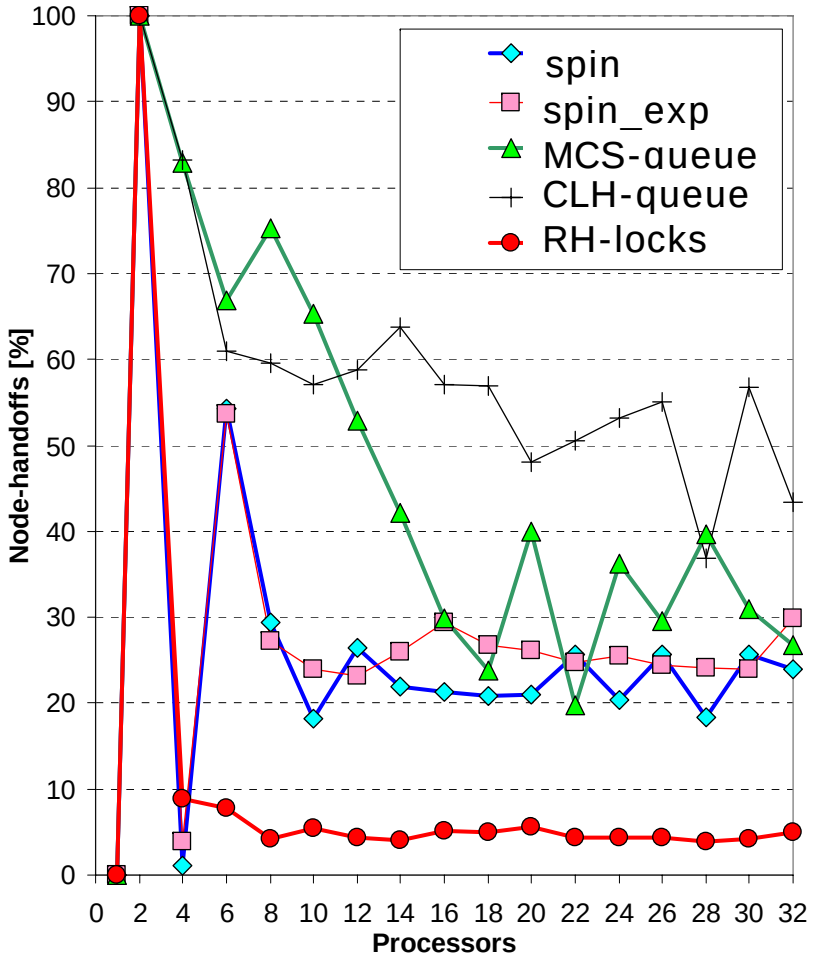
DARK 2008

RH locks: encourages unfairness

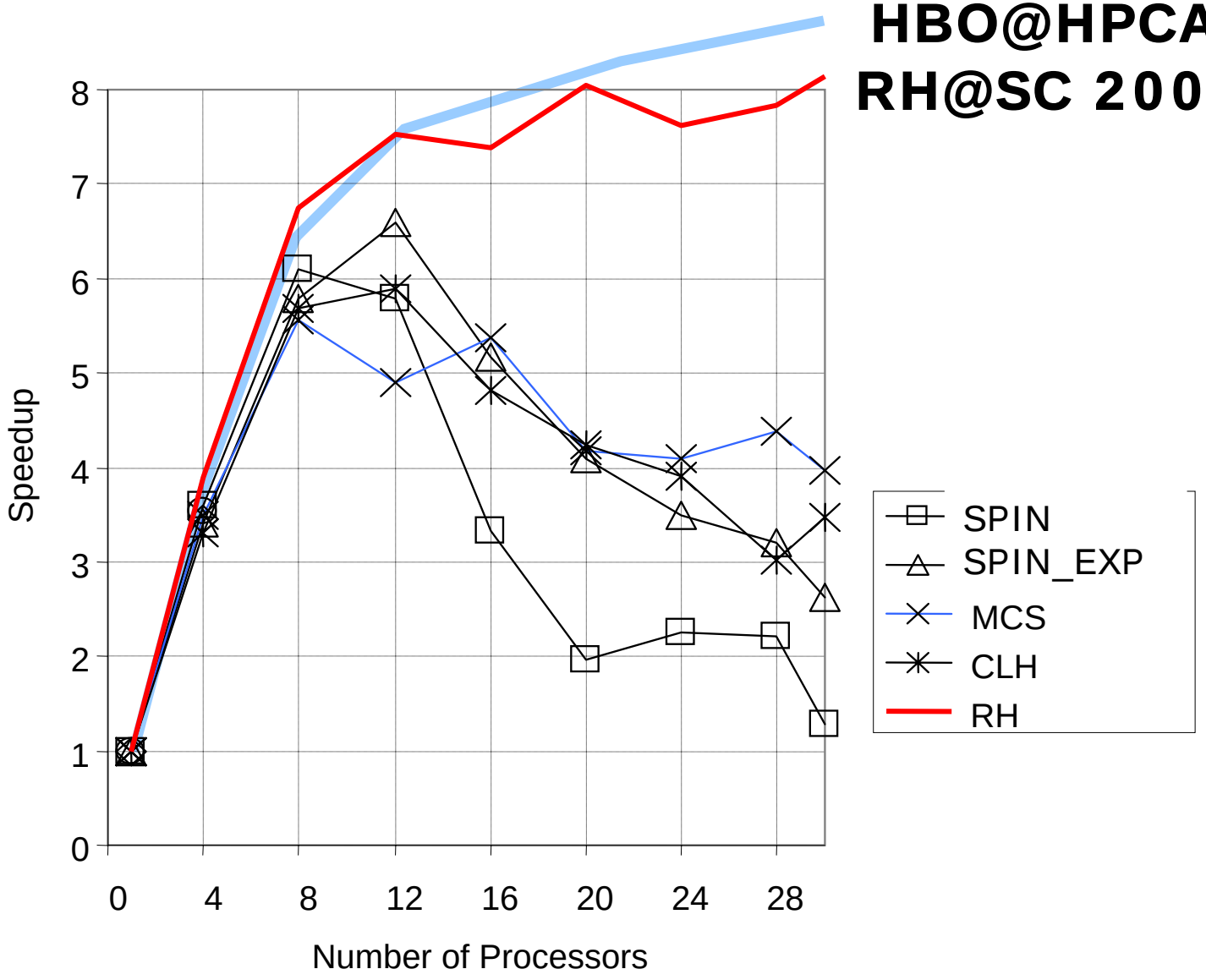
Time per lock handover



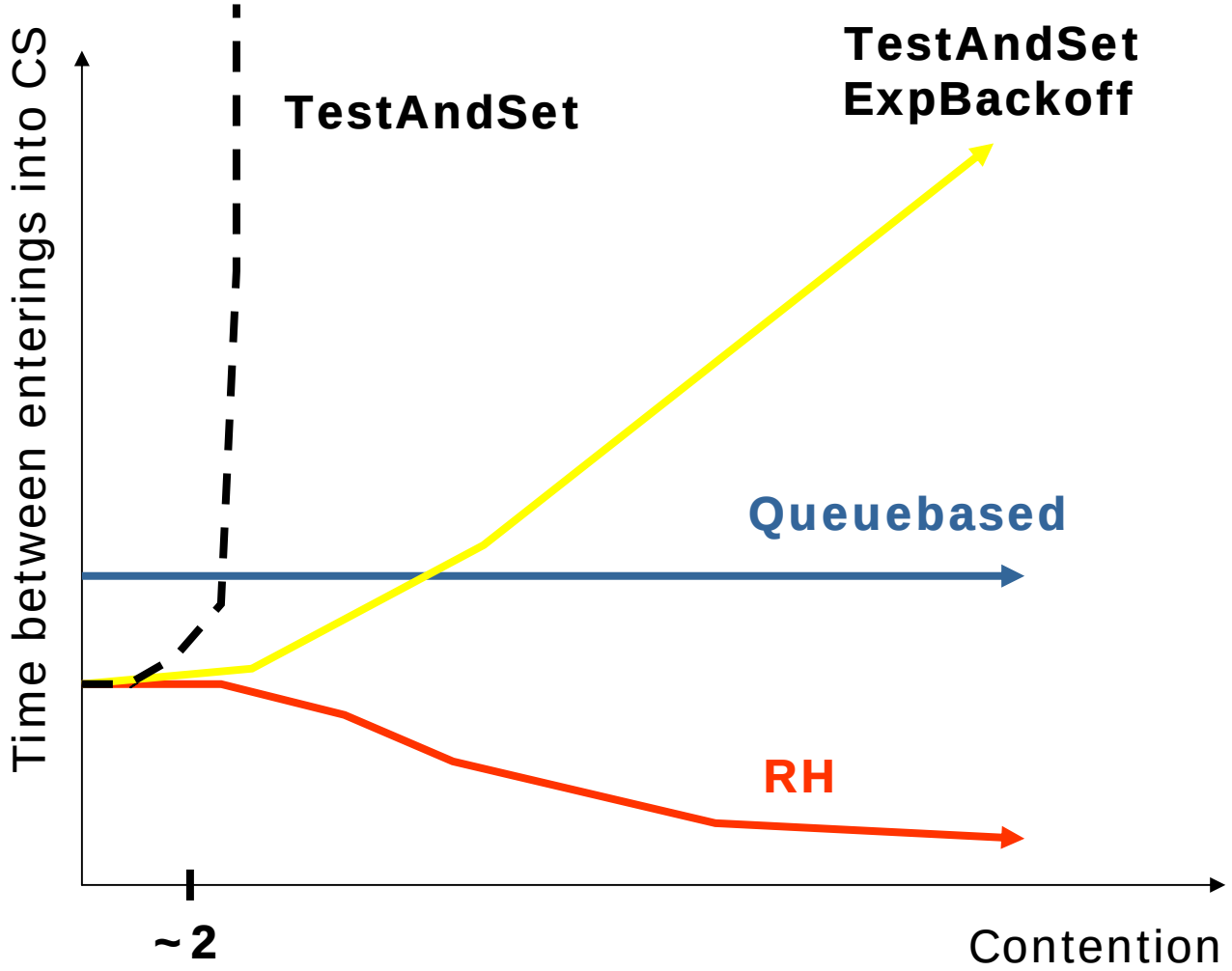
Node migration (%)



Ex: Splash Raytrace Application Speedup



Performance under contention



DARK
2008

UPPSALA UNIVERSITET

Barriers: Make the first threads wait for the last thread to reach a point in the program

1. Software algorithms implemented using locks, flags, counters

2. Hardware barriers
   - Wired-AND line separate from address/data bus
   - Set input high when arrive, wait for output to be high to leave
   - (In practice, multiple wires to allow reuse)
   - Difficult to support arbitrary subset of processors
     - even harder with multiple processes per processor
   - Difficult to dynamically change number and identity of participants
     - e.g. latter due to process migration

DARK 2008

UPPSALA UNIVERSITET

A Centralized Barrier

```
BARRIER (bar_name, p) {
int loops;
loops = 0;

local_sense = !(local_sense);              /* toggle private sense variable
                                             each time the barrier is used */

 LOCK(bar_name.lock);
    bar_name.counter++;                    /* globally increment the barrier count */
    if (bar_name.counter == p)  {          /* everybody here yet ? */
        bar_name.flag = local_sense;       /* release waiters*/
        UNLOCK(bar_name.lock)
     }
    else
      {  UNLOCK(bar_name.lock);
        while (bar_name.flag != local_sense) {     /* wait for the last guy */
          if (loops++ > UNREASONABLE) report_warning(pid)}
     }
```

DARK 2008

UPPSALA UNIVERSITET

Centralized Barrier Performance

- Latency
  - Want short critical path in barrier
  - Centralized has critical path length at least proportional to *p*
- Traffic
  - Barriers likely to be highly contended, so want traffic to scale well
  - About 3*p* bus transactions in centralized
- Storage Cost
  - Very low: centralized counter and flag
- Key problems for centralized barrier are latency and traffic
  - Especially with distributed memory, traffic goes to same node

➔ Hierarchical barriers

UPPSALA UNIVERSITET

New kind of synchronization: Transactional Memory

- OLD: lock(ID); unlock(ID) around critical sections
- NEW: start_transaction; end_transaction around ”critical sections” (note: no ID!!)
  - Underlying mechanism to guarantee atomic behavior
  - This is not the same as guaranteeing that only one thread is in the critical action!!
  - Supported in HW (soon) or in SW (normally very inefficient)
- Suggested by Maurice Herlihy in 1993
- HW support announced by Sun and Intel (??)

UPPSALA UNIVERSITET

Support for TM

- Start_transaction:
  - Save original state to allow for rollback (i.e., save register values)
- In critical section
  - Do not make any global state change
  - Detect ”atomic violations” (others writing data you’ve read in CS or reading data you have written)
  - At atomic violation: roll-back to original state
  - Forward progress must be guaranteed
- End_transation
  - Atomically commit all changes performed in the critical section.

UPPSALA UNIVERSITET

Advantage of TM

- Do not have to ”name” CS
- Less risk for deadlocks
- Performance:
  - Several thread can be in ”the same” CS as long as they do not mess with each other
  - CS can often be large with a small performance penalty

Introduction to Multiprocessors

Erik Hagersten
Uppsala University

UPPSALA UNIVERSITET

Flynn's Taxonomy

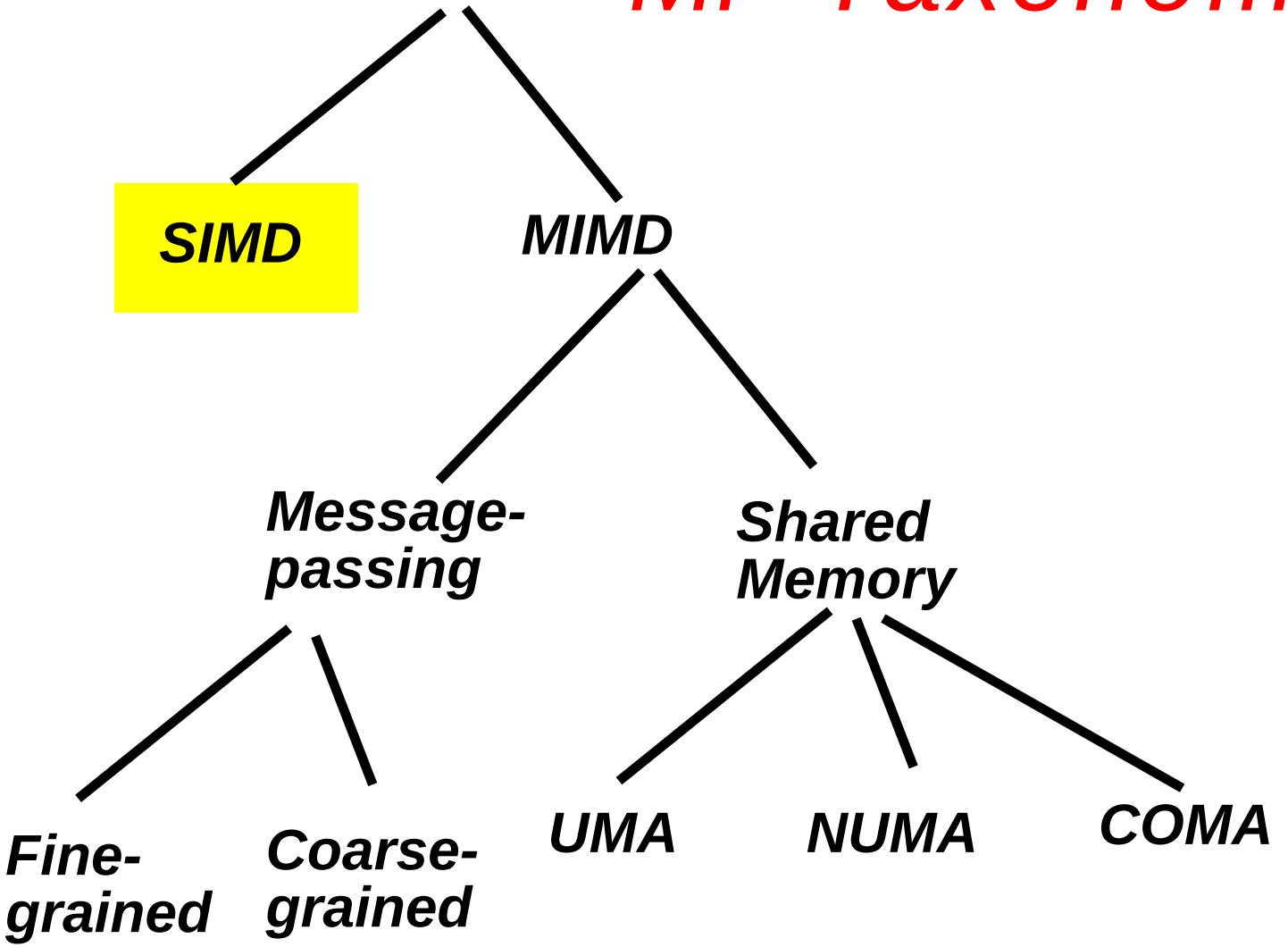
{Single,Multiple}Instruction +
{Single,Multiple}Data

- SISD - Our good old "simple" CPUs
- SIMD – Vectors, "MMX", DSPs, CM-2,...
- MIMD – TLP, cluster, shared-mem MP,...
- MISD – Can't think of any...

DARK 2008

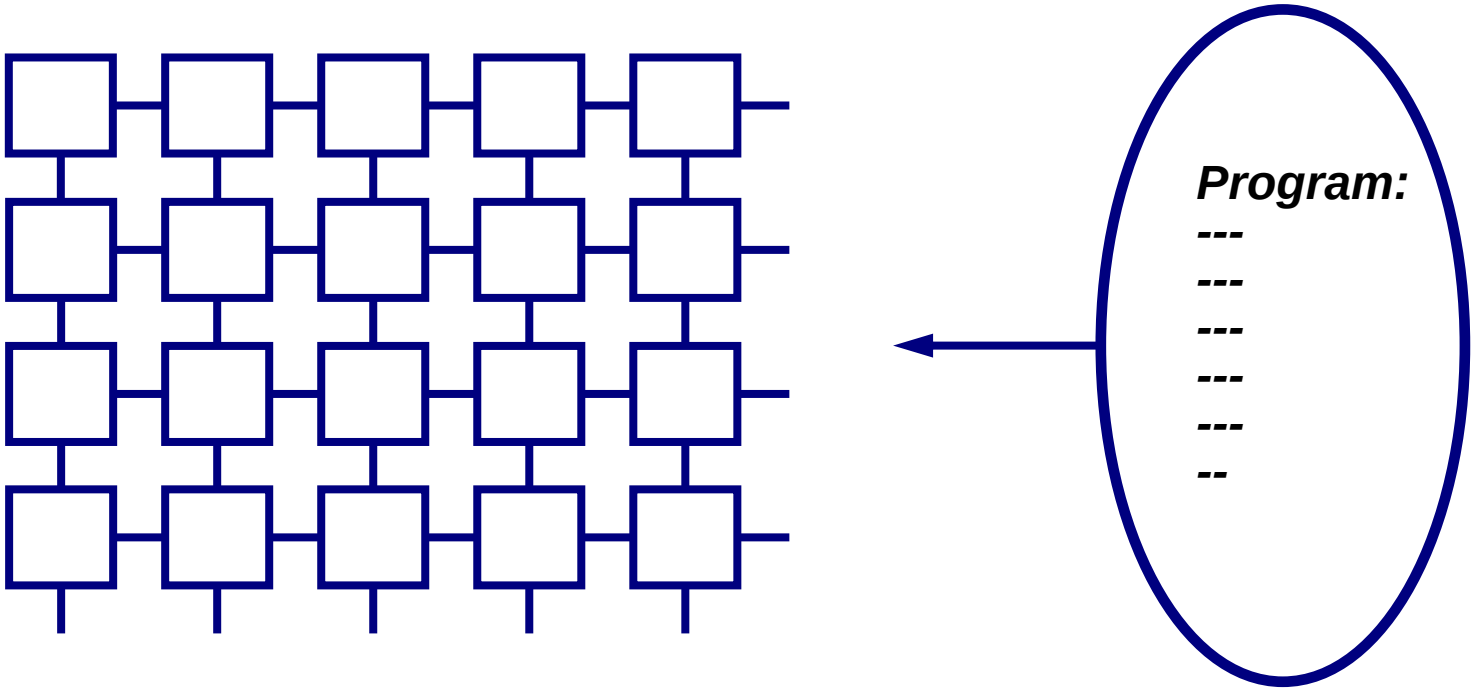
UPPSALA UNIVERSITET

MP Taxonomy



DARK 2008

SIMD = "Dataparallelism"



UPPSALA UNIVERSITET

SIMD: Thinking Machine

- Connection Machine: CM1, CM2, CM200
  (at KTH ~1990: CM200 “Bellman”)
- One-bit ALU and a small local memory
- FP accelerator available
- Programmed in “ASM”, *C and *Lisp
- Hard to program (in my opinion...)

UPPSALA UNIVERSITET

Other Flavors of SIMD

- MMX/AltiVec/VIS instructions/SSE…
  - Divide register content into smaller items (e.g., bytes)
  - Special instructions operate on all items i parallel, e.g., BYTE-COMPARE…
- Some DSPs (Digital Signal Processors)
- Some Image Processors

UPPSALA UNIVERSITET

Vector architectures
CRAY, NEC, Fujitsu, …

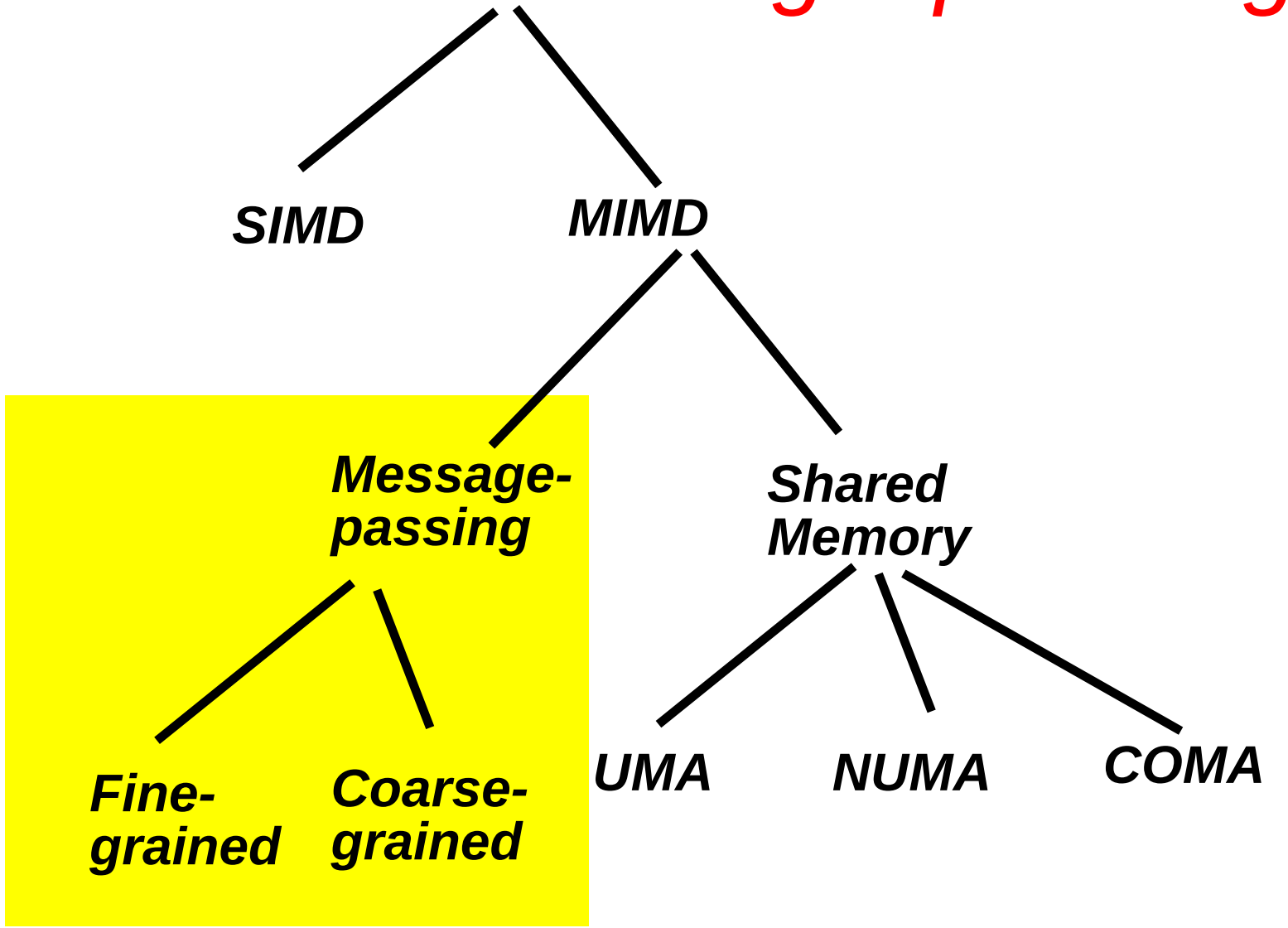
- Vectory Processors
  - LD/ST operate on vectors of data
  - ALU Ops operate on vectors of data

- Example:
  - 8 “vector register” contains 64 vector entries each
  - A single LD/ST instr loads/stores entire vectors
  - A single ALU instr V1 ← V2 op V3
  - 64 bit mask vectors make execution conditional
  - Overlaps Mem and ALU ops
  - One form of ”SIMD” -- Single Instruction Multiple Data

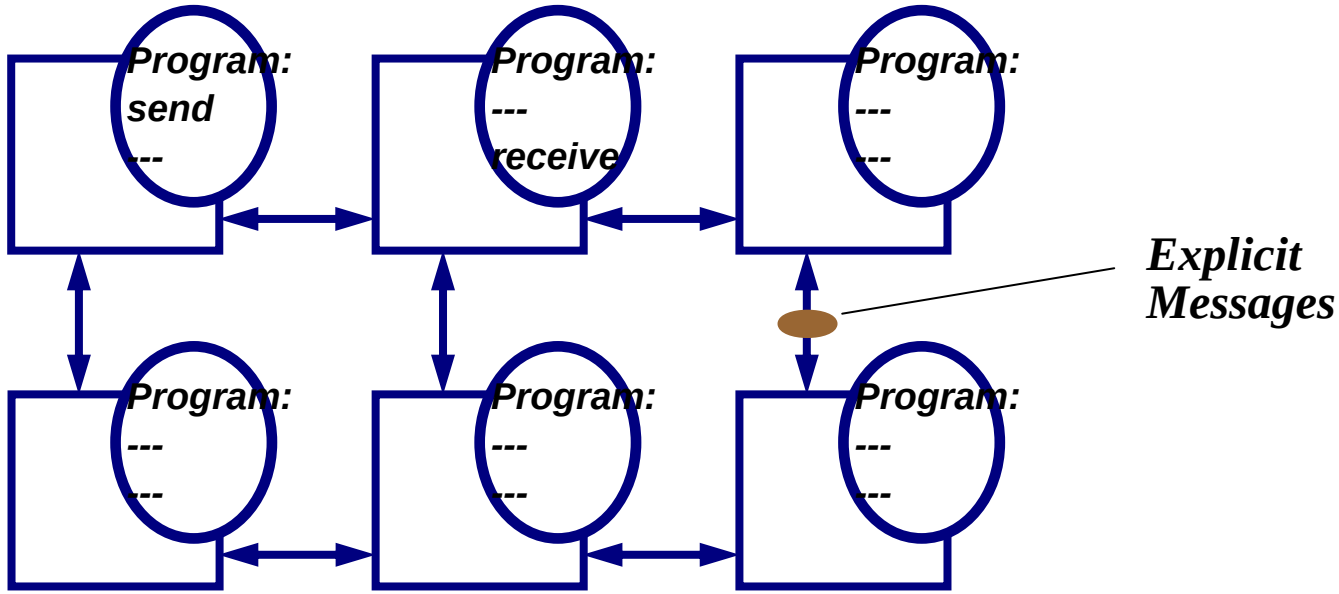
DARK 2008

UPPSALA UNIVERSITET

MIMD: Message-passing



Message-passing Arch MIMD



UPPSALA UNIVERSITET

Message-Passing HW

- Programmed in MPI or PVM (or HPFortran…)
  - Thinking Machines: CM5
  - Intel: Paragon
  - IBM: SP2
  - Meiko (Bristol, UK!!): CS2
  - Today: Clusters with high-speed interconnect (Important today, but not covered in this course)

- Clusters can be used as message-passing HW, put is most often used as capacity computing (i.e., throughput computing)

UPPSALA UNIVERSITET

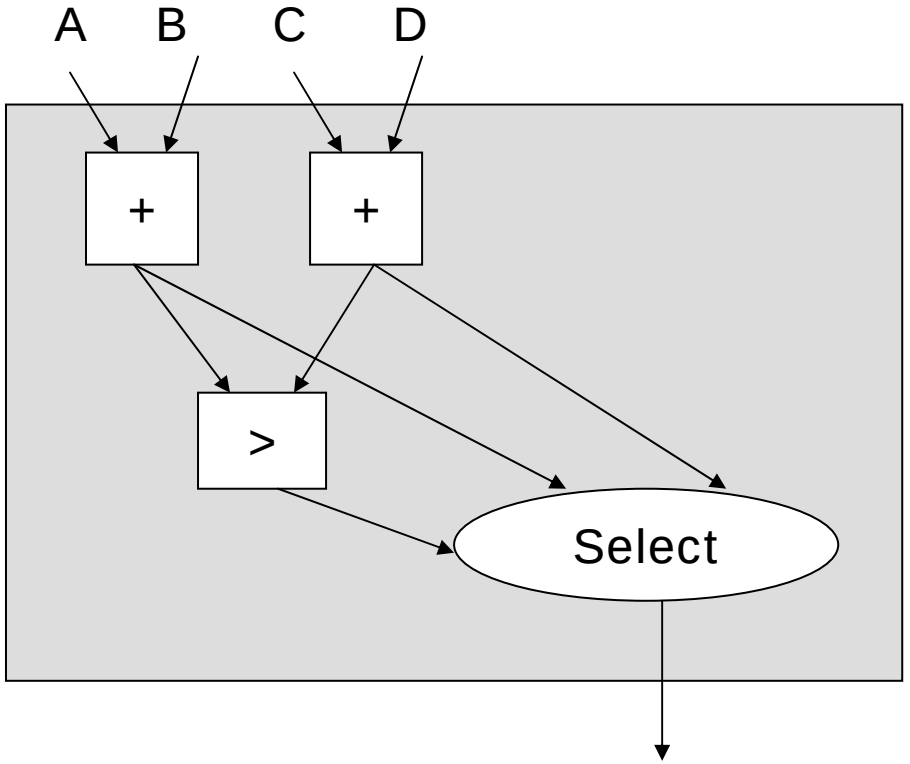
Dataflow

- Often programmed in functional languages (e.g., ID)
- Compile program to Dataflow graph
- Operands + graph = executable
- Operation ready when the source operands are available

UPPSALA UNIVERSITET

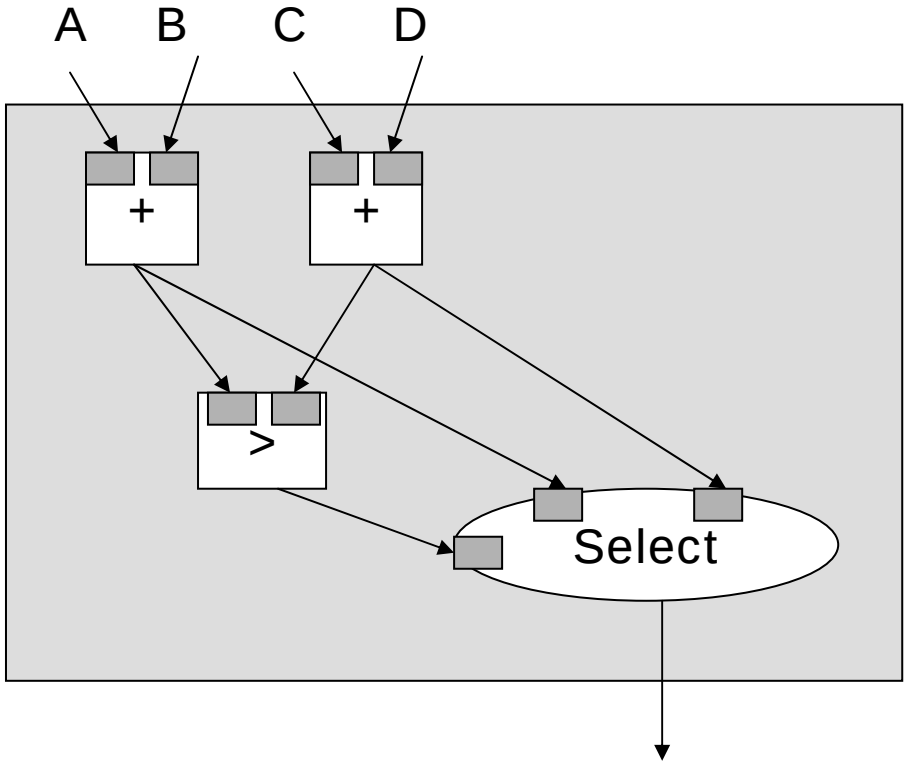
Dataflow Example:

```
X := A + B
Y := C + D
If (X > Y)
  output X
else
  output Y
```



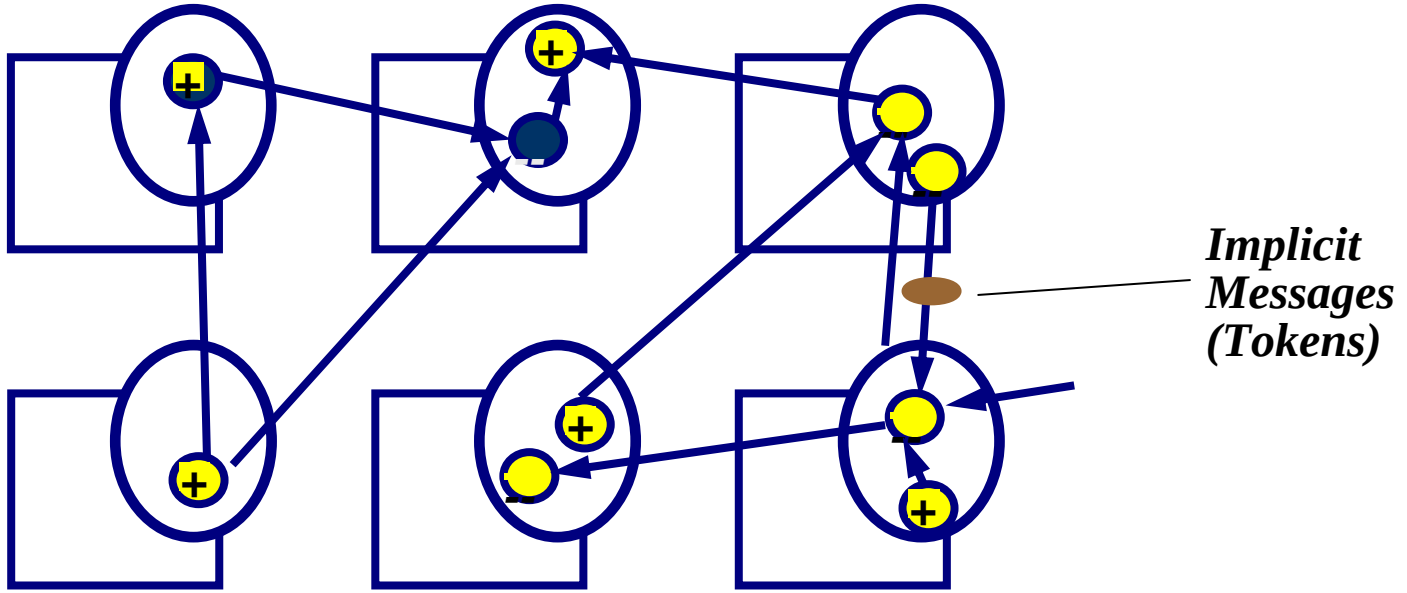
Static Dataflow (Dennis)

```
X := A + B
Y := C + D
If (X > Y)
  output X
else
  output Y
```



Each operand executed exactly once per program
Location assigned for each input data ▭

Fine-grained Message-passing Dataflow ==> Multithreading



UPPSALA UNIVERSITET

Dynamic Dataflow (Arvind)

- Allows for recursion and loops
- Each invocation is assigned a “color”
- Pairs of operands are matched dynamically
  - Based on {Color, Operation}
  - In the Waiting-Matching Section (I.e., a cache)
- One problem: too much parallelism in the wrong place

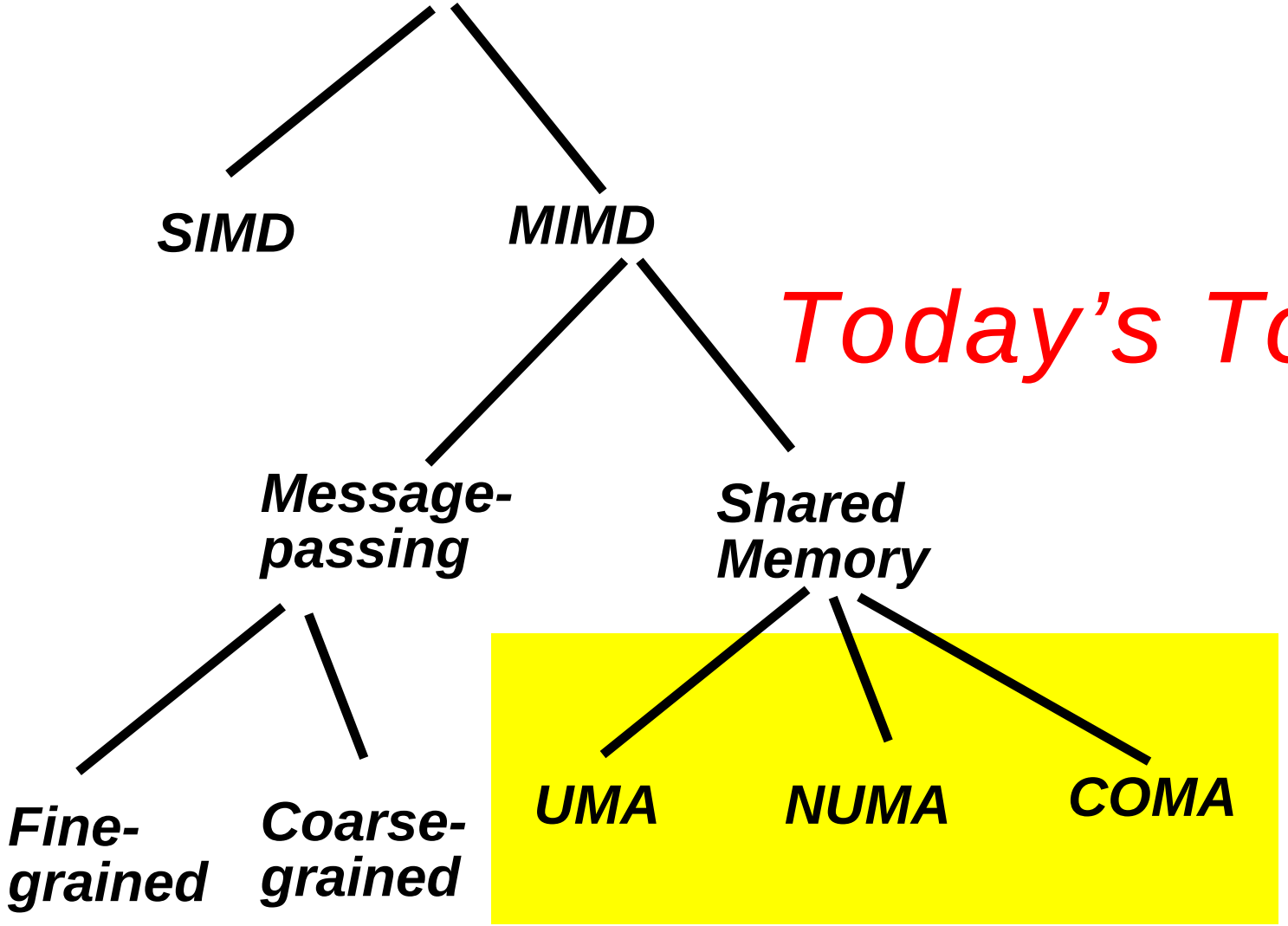
UPPSALA UNIVERSITET

Carlstedts Elektornik

Gunnar Carlstedt, Staffan Truve' et al

- Processor "8601"
  - Gothenburg 1990-1997
  - Functional language "H"
  - Execution performed by a reduction a CAM memory
  - ALU rarely used
  - Many parallel processors on a wafer
    (Wafer-scale integration)
  - ➔ CRT (Carlstedt Research Technology)

DARK 2008


SIMD
MIMD
Today's Topic
Message-passing
Shared Memory
Fine-grained
Coarse-grained
UMA
NUMA
COMA

The server market 1995

Server Size	High-Perf. Computing	Commercial Computing
<$10k	1%	19%
<$50k	5%	25%
<$250k	5%	24%
<$1M	2%	9%
>$1M	3%	8%

UNIX shared-mem servers

The target of the rocket science supercomputers