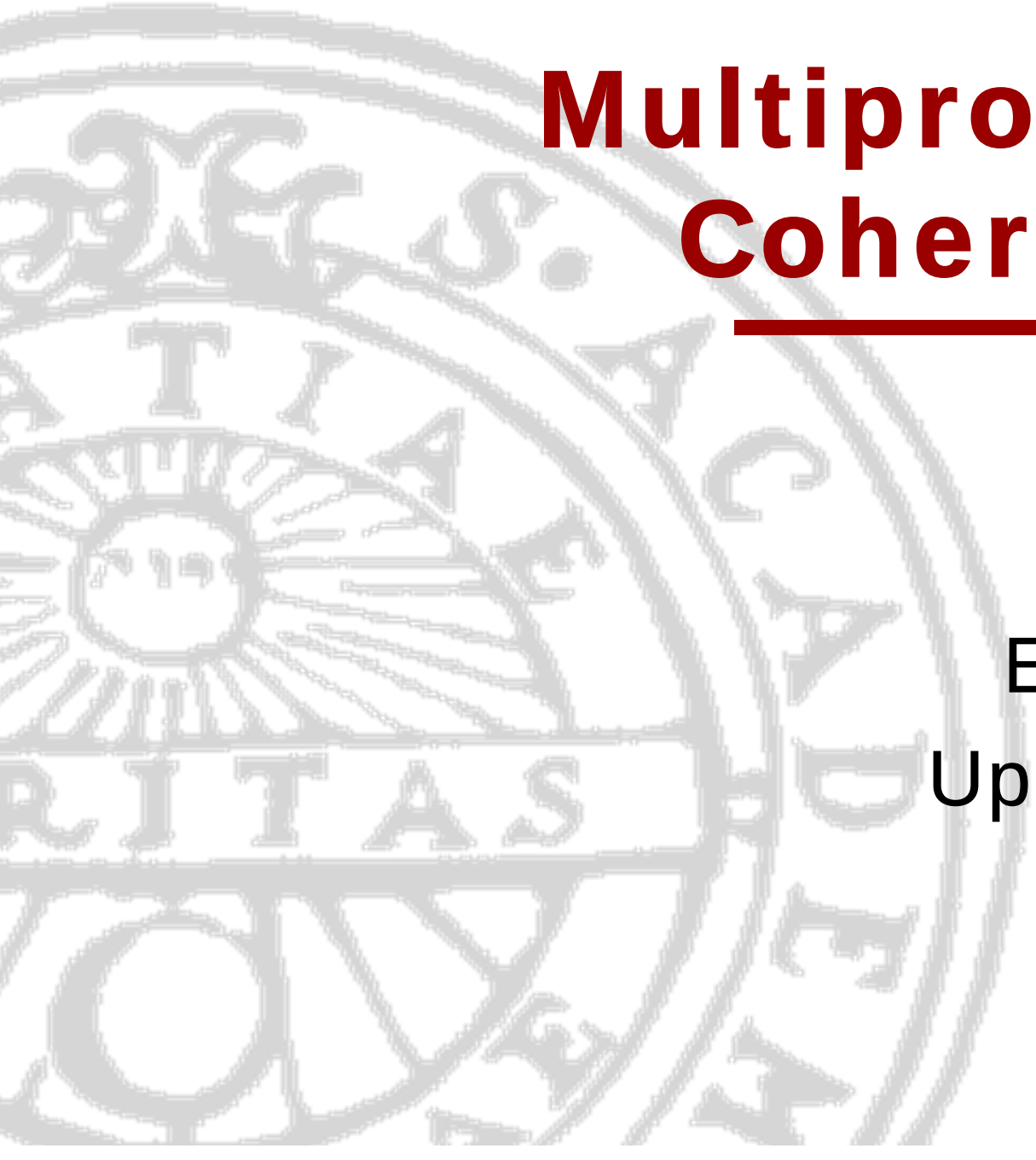
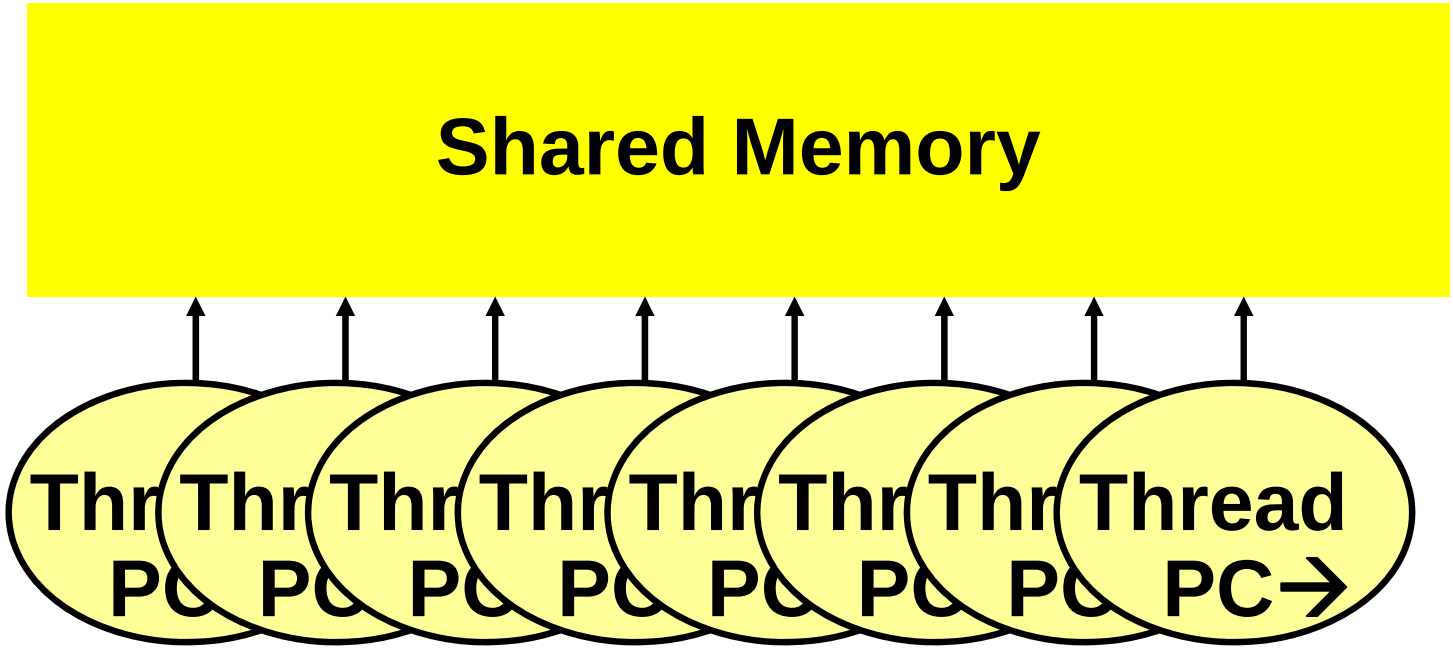


Multiprocessors and Coherent Memory

Erik Hagersten
Uppsala University

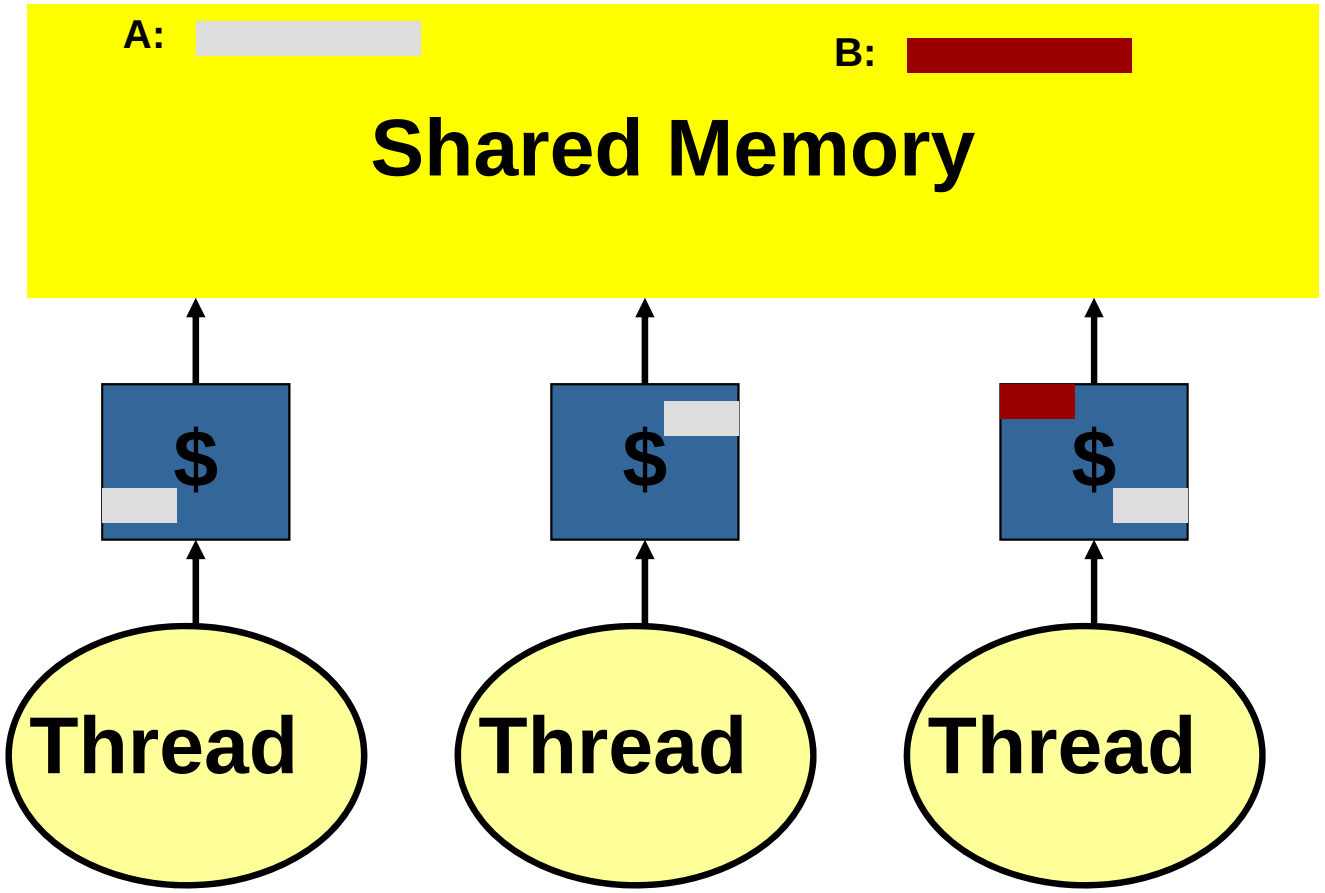
UPPSALA UNIVERSITET

Programming Model: Coherent shared memory

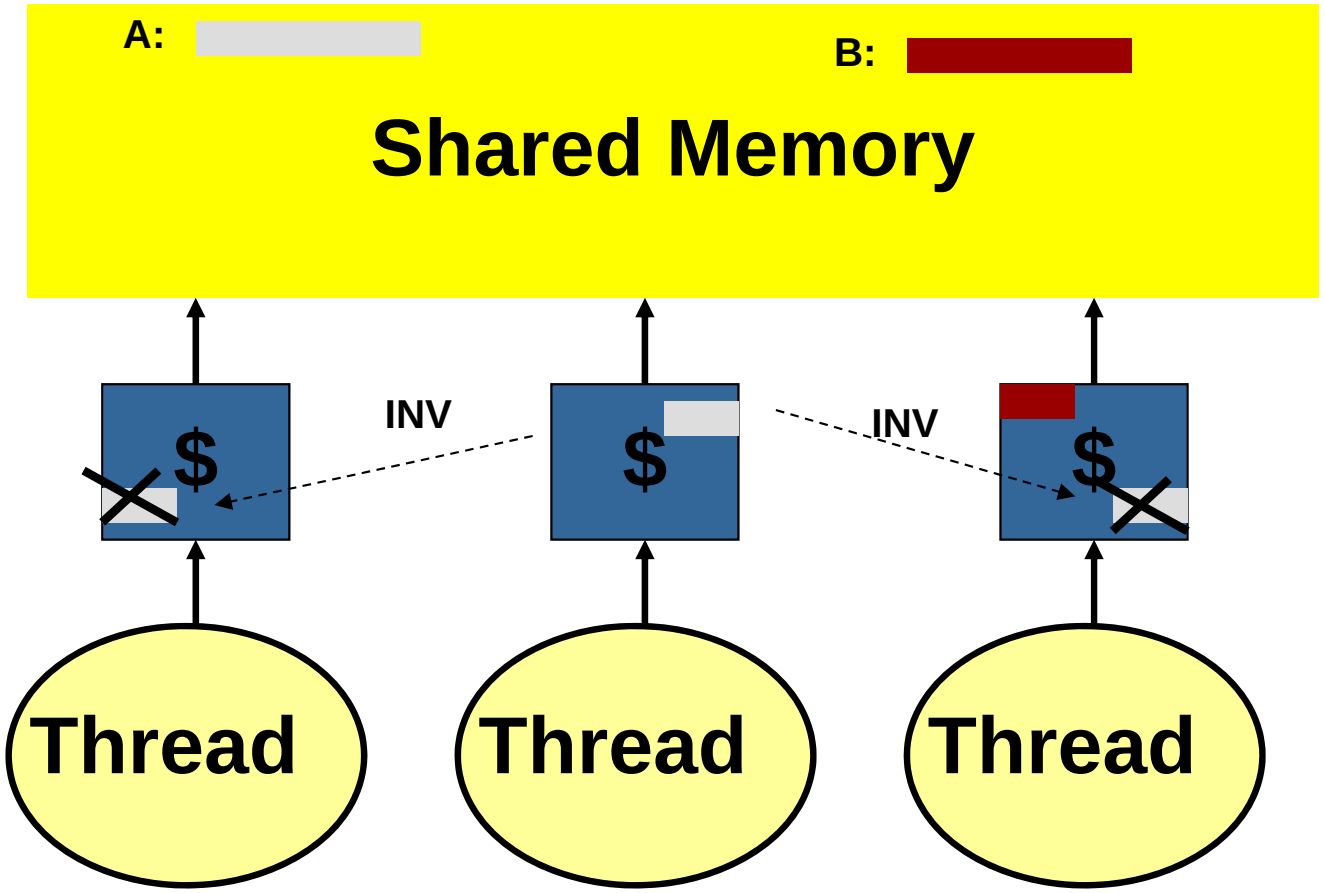


AVDARK 2013

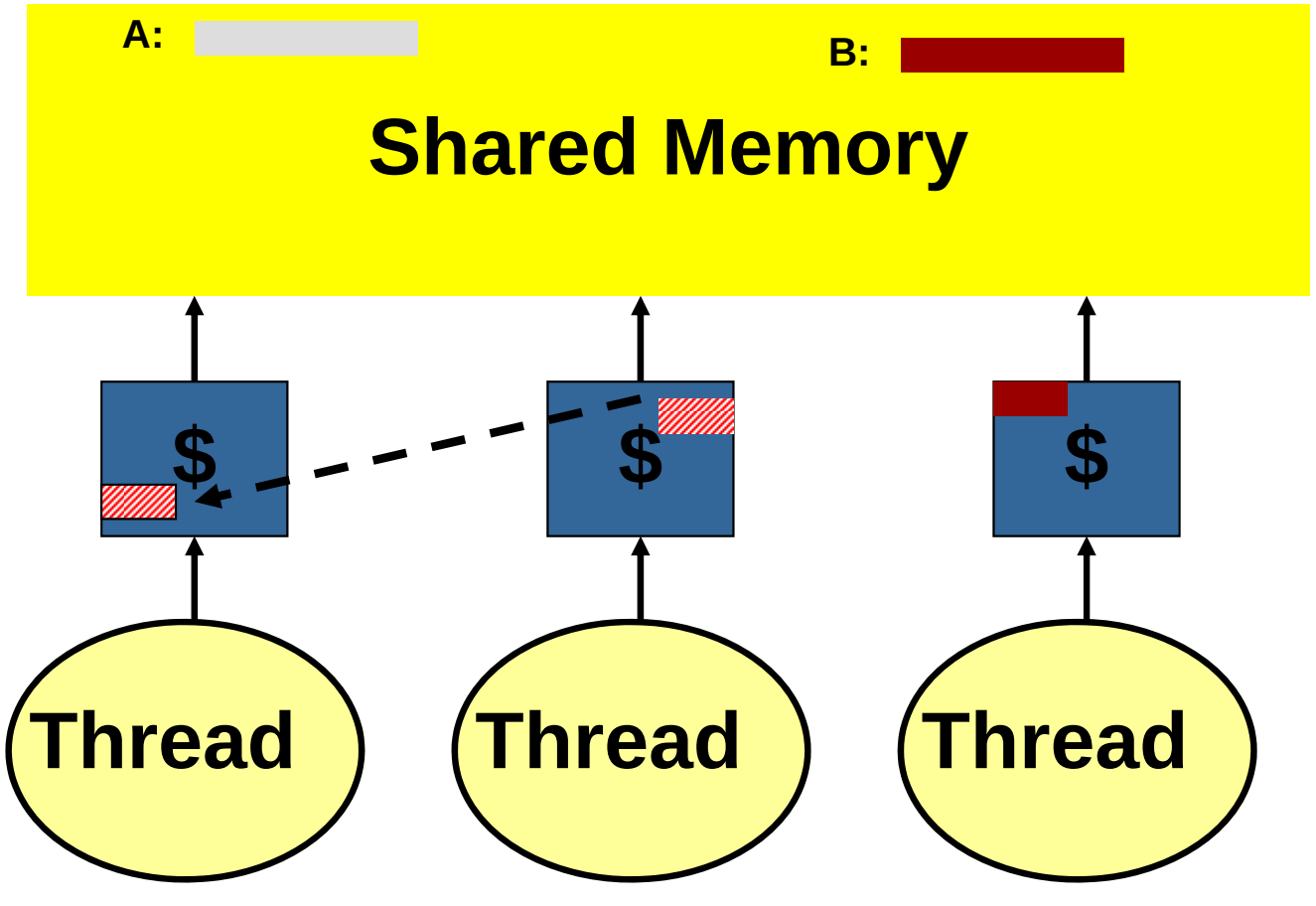
Automatic Replication of Data



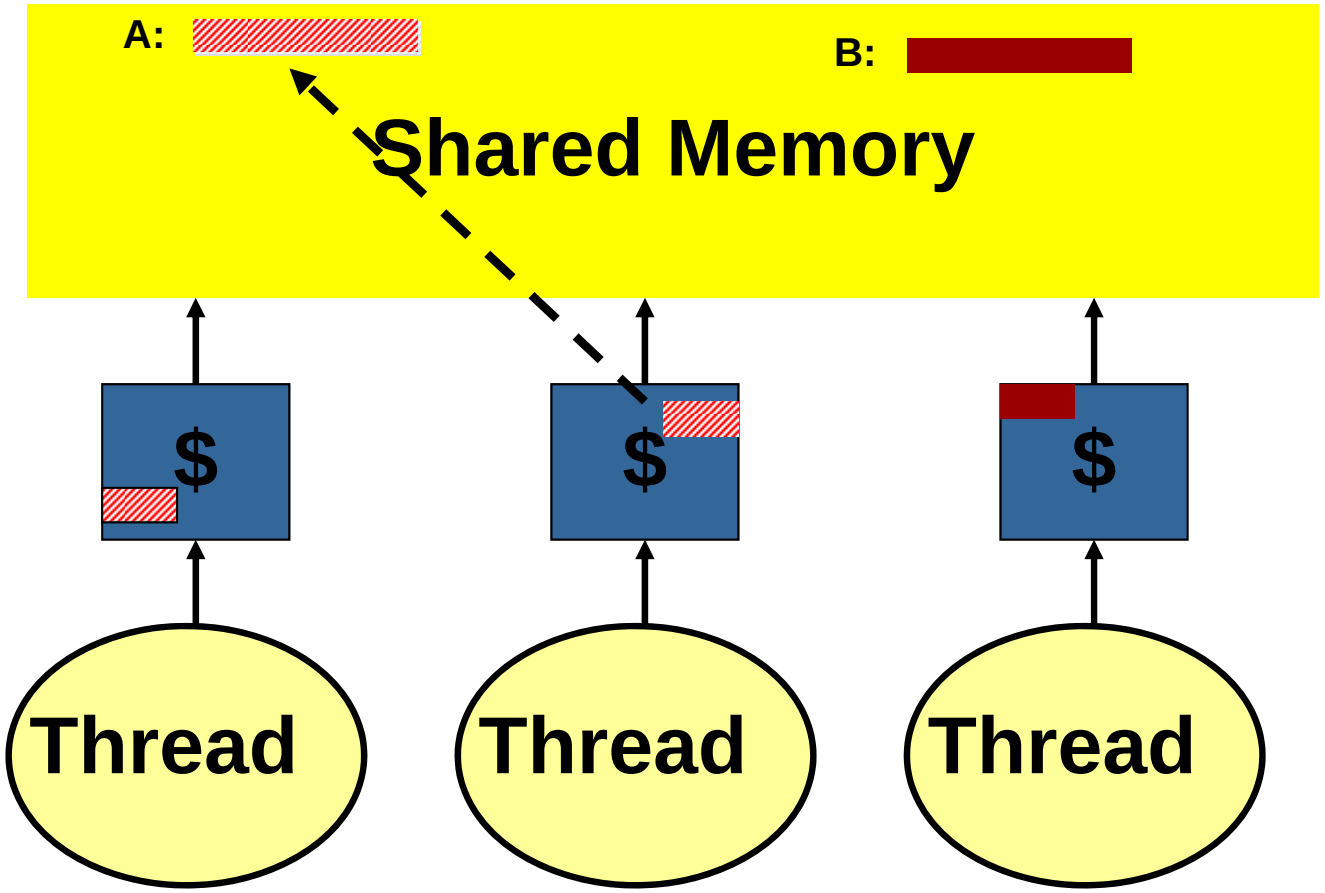
The Cache Coherent Memory System
Coherent Write (Here: Write invalidate)



The Cache Coherent Memory System
Coherent Read & Write-back
(Here: Cache to Cache Transfer)



The Cache Coherent Memory System
Coherent Read & Write-back
(Here: Write Back)



UPPSALA UNIVERSITET

Summing up Coherence

Good intuition, but too strong definition!

There can be many copies of a datum, but only one value

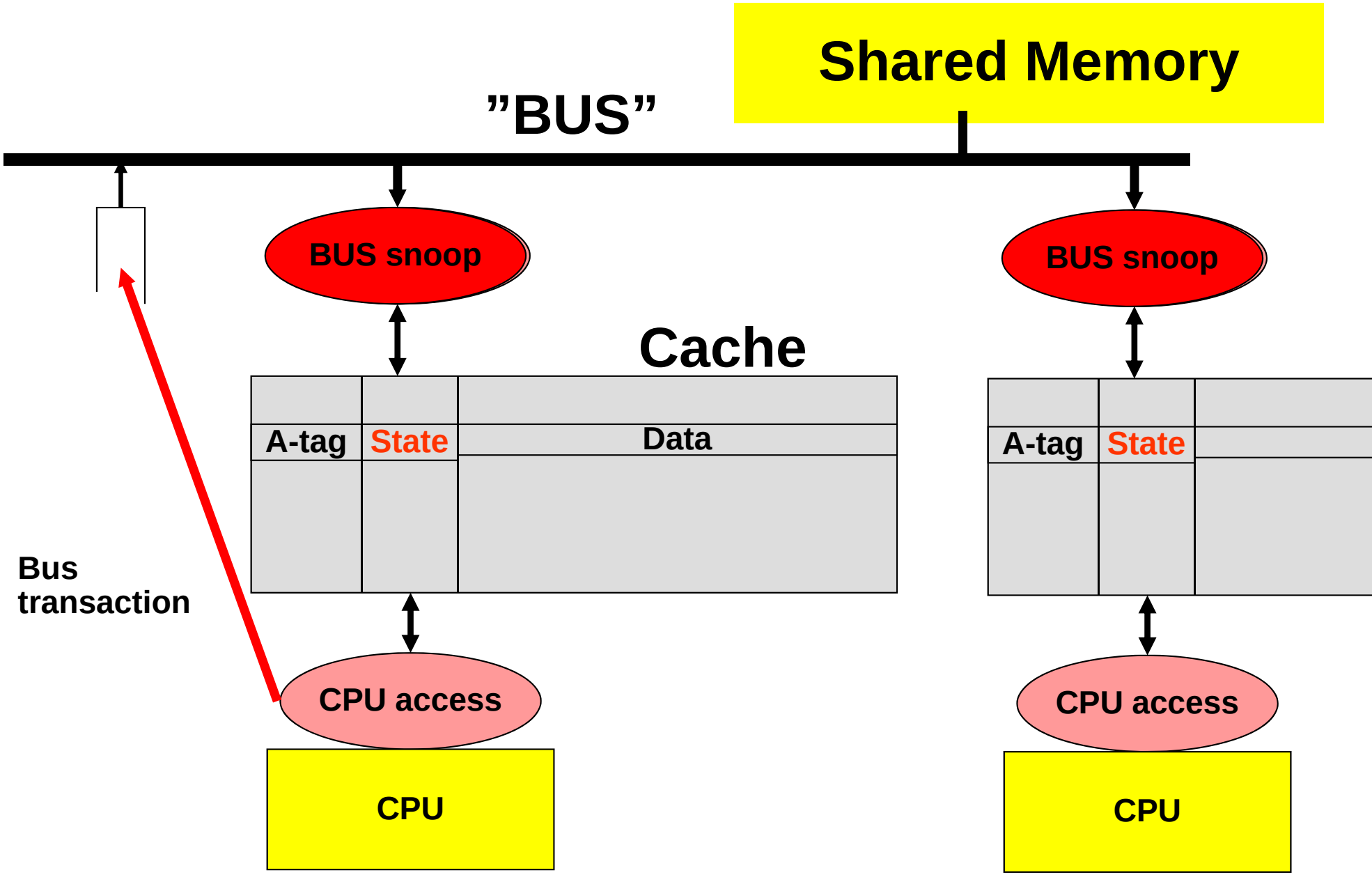
There is a single global order of value changes to each datum

After the computer stops, all copies should have the same value

Thread1={1,2,3,4,5,6,7…} Thread2={1,4,7…} ~~Thread3={1,8,7…}~~

AVDARK 2013

Snoop-based Protocol Implementation



Example: MOSI Bus Snoop

UPPSALA UNIVERSITET

STATES:

M – Modified: My dirty* copy is the only cached copy

S – Shared: I have a clean copy, others may also have a copy

O – Owner: I have a dirty copy, others may also have a copy

I – Invalid: I have no valid copy in my cache (including cache miss)

BUS TRANSACTIONS FROM OTHERS:

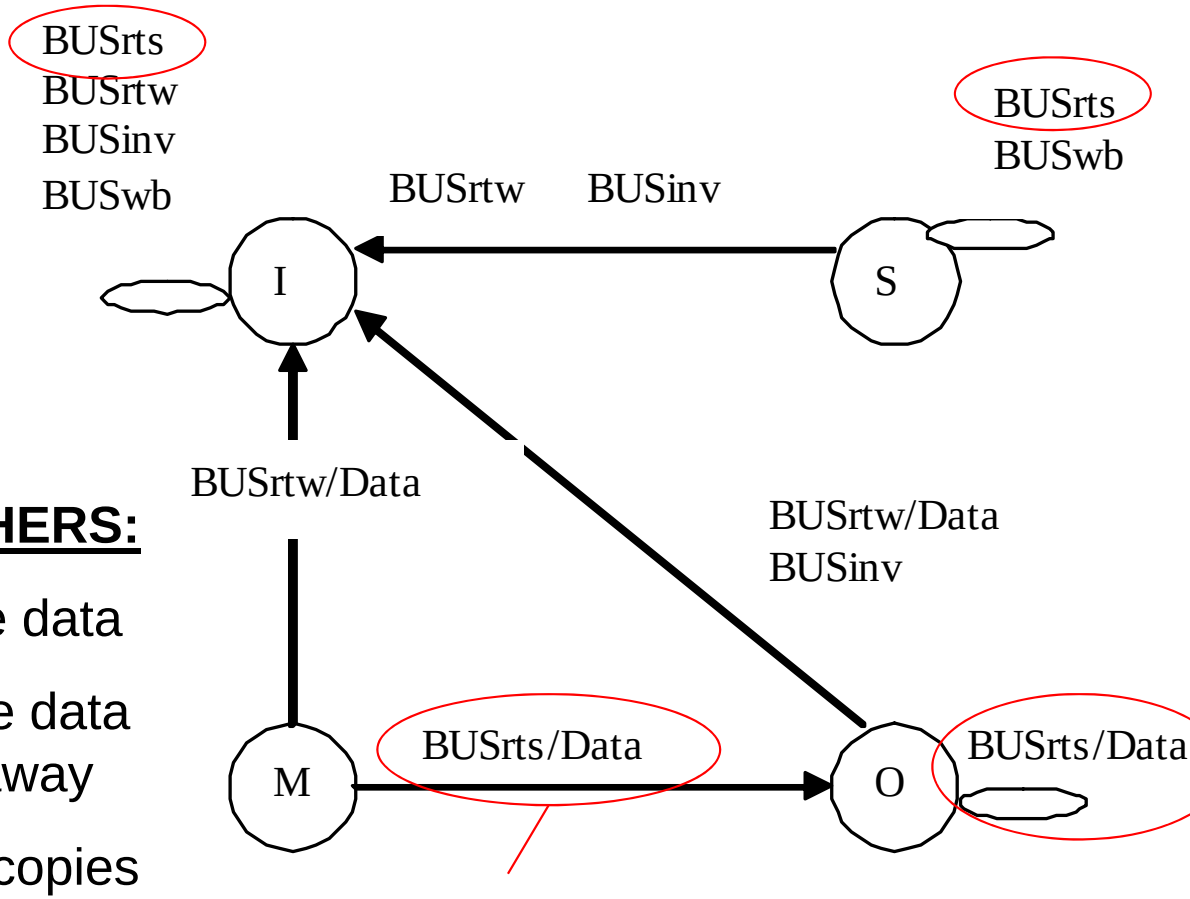
BUSrts ReadtoShare. Reading the data

BUSrtw: ReadToWrite. Reading the data with the intention to modify it right away

BUSinv: Invalidating other caches copies

BUSwb: Writing data back to memory

**Dirty: my value differs from the old value in mem*

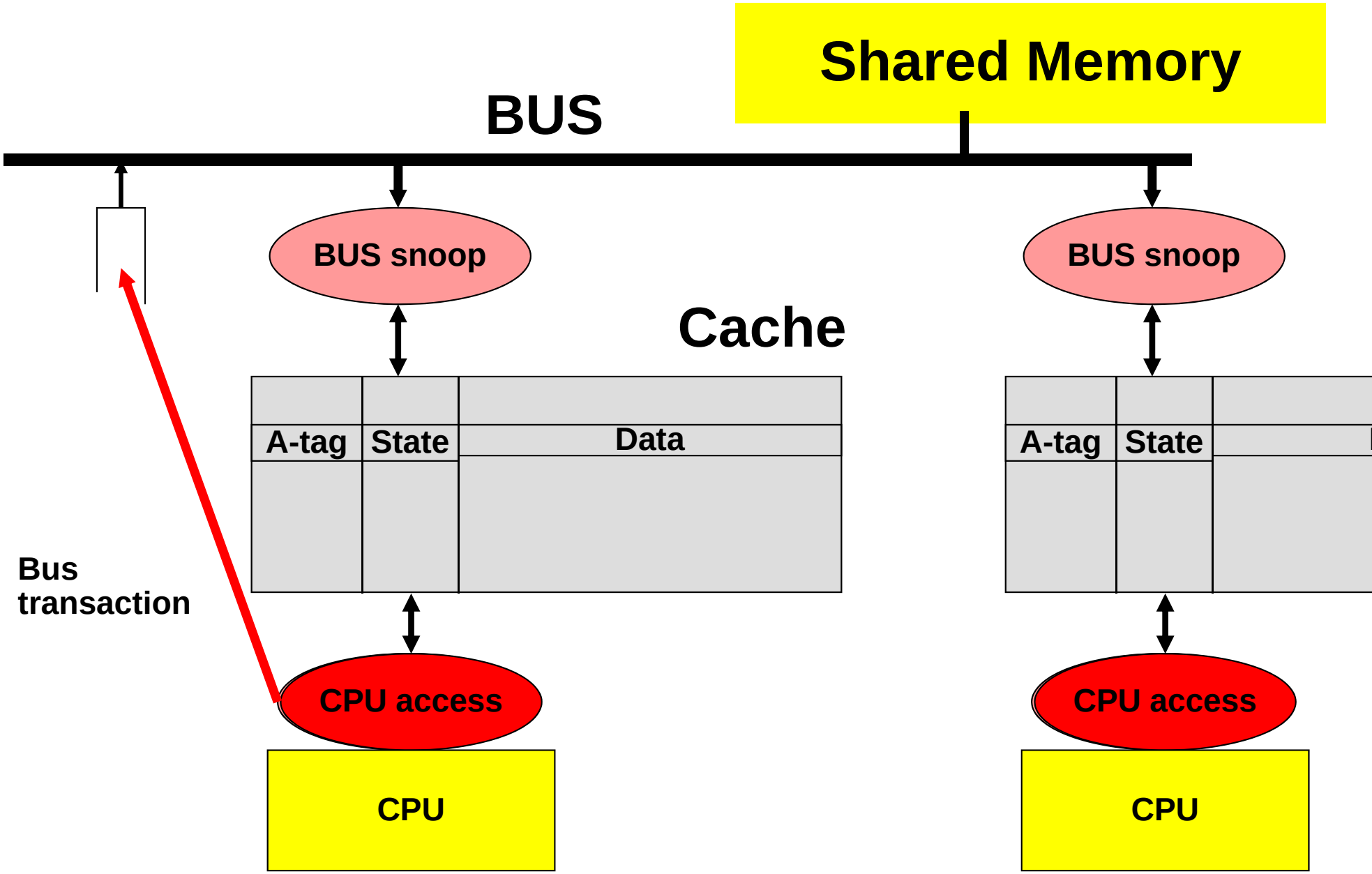


Input-signal / Reply-signal

Meaning: If you are in state M and see BUSrts, goto state O and reply with Data

AVDARK 2013

Snoop-based Protocol Implementation



Example: CPU access MOSI

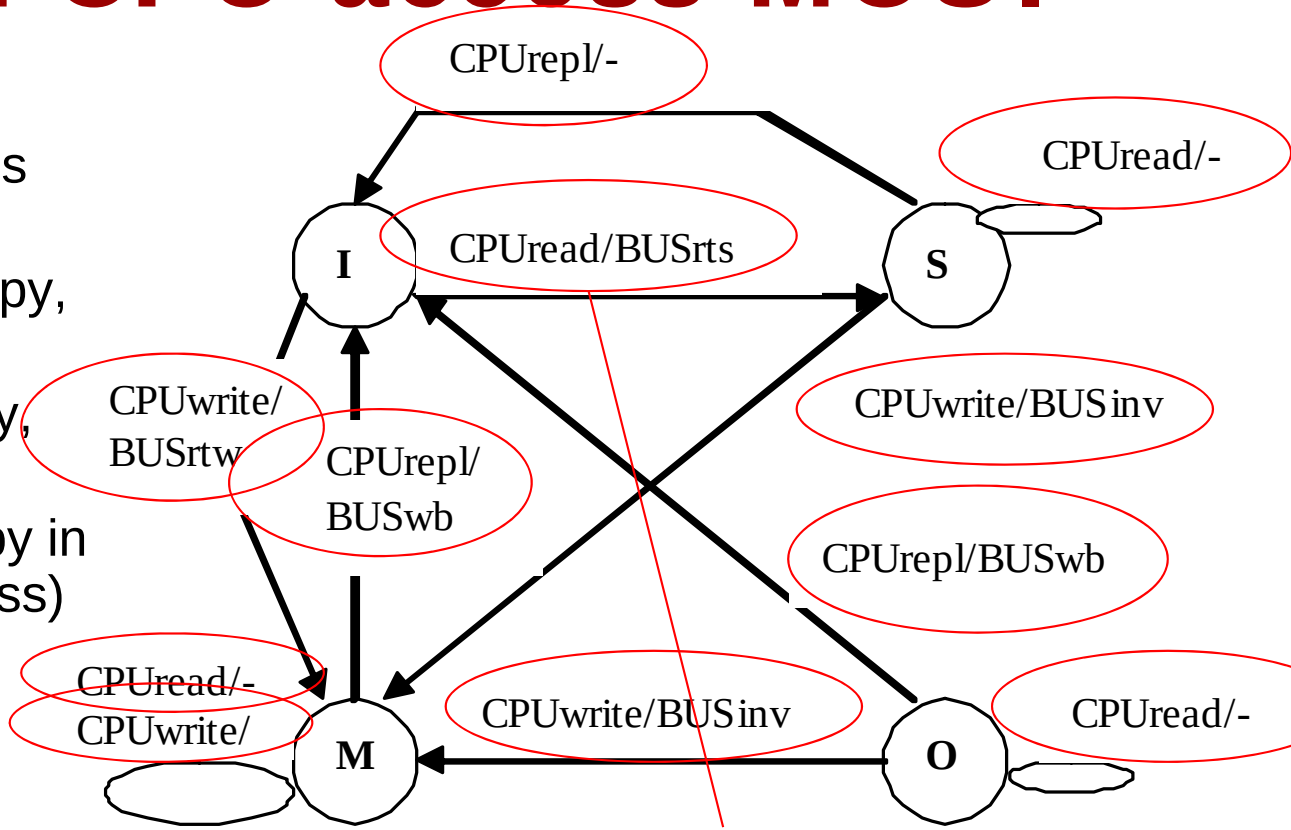
STATES:

M – Modified: My dirty* copy is the only cached copy

S – Shared: I have a clean copy, others may also have a copy

O – Owner: I have a dirty copy, others may also have a copy

I – Invalid: I have no valid copy in my cache (may be a cache miss)



Input-signal/Reply-signal
Meaning: If you are in state I and see CPURead, send a BUSrts and goto S

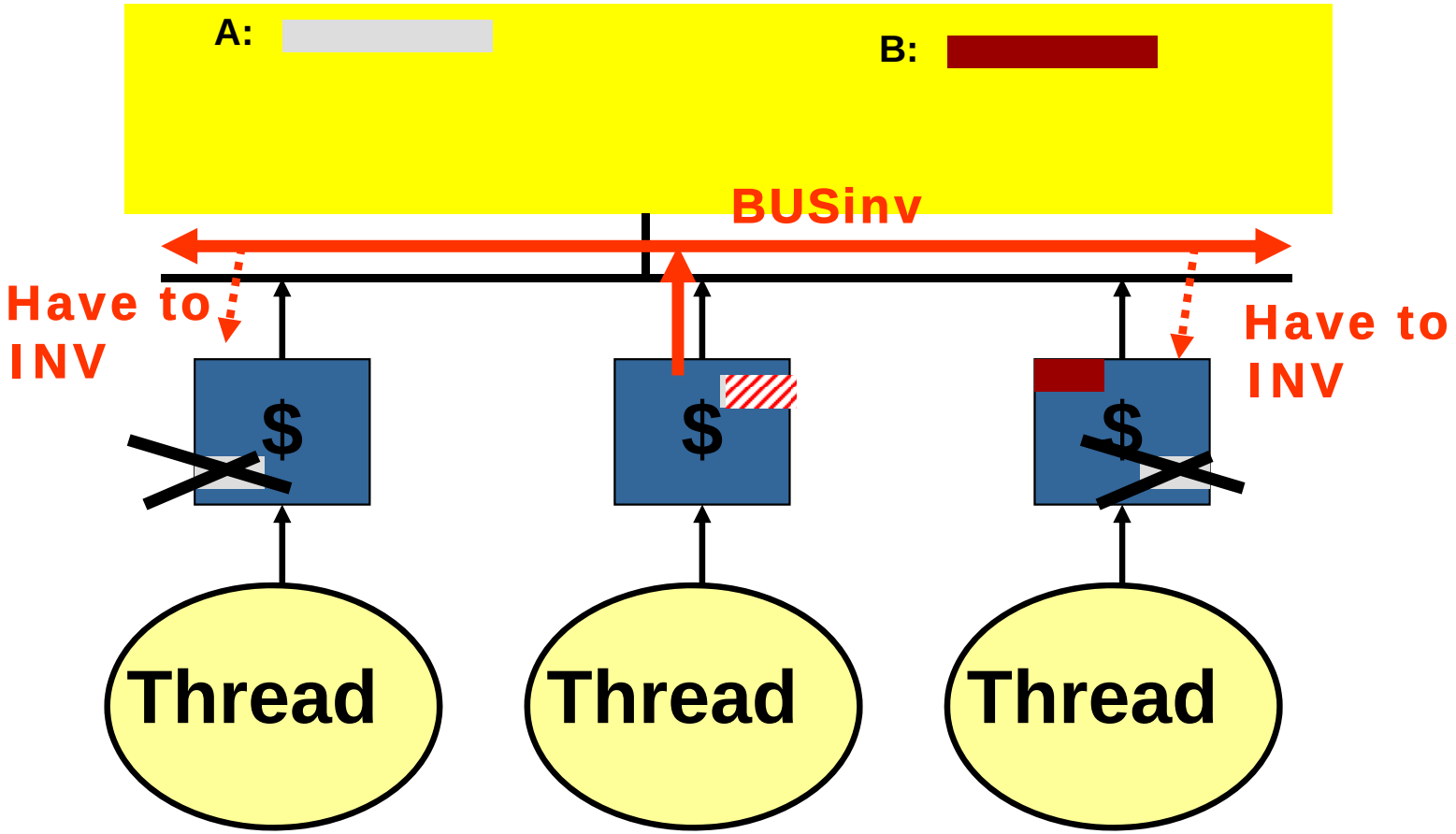
FROM MY CPU:

CPURead Caused by a Load instruction

CPUwrite: Caused by a Store or Atomic instruction

CPURepl: Caused by a replacement of this cachline (caused by murphy ☺)

”Upgrade” in snoop-based



UPPSALA UNIVERSITET

More Cache Lingo

- **Capacity miss** – too small cache
- **Conflict miss** – limited associativity
- **Compulsory miss** – accessing data the first time
- **Coherence miss** – The cache would have had the data unless it had been invalidated by someone else
- **Upgrade miss:** (only for writes) – The cache would have had a writable copy, but answered a read request and "downgraded" itself to read-only state
- **False sharing:** Coherence/downgrade is caused by a shared cacheline and not by shared data:

False sharing example:

Read A	…
…	Read D
Write A	…
…	Write D
Read A	

cacheline:

A, B, C, D

AVDARK 2013

UPPSALA UNIVERSITET

Example in Class:

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;
A = A + 1;              /* this is a primitive kind of unlock */
while (A != 4) {};      /* this is a primitive kind of barrier sync */
<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.
Assume that CPU3, 2 and 1 first make one memory reference (i.e, a load or a store) each and then repeats that interleaving.

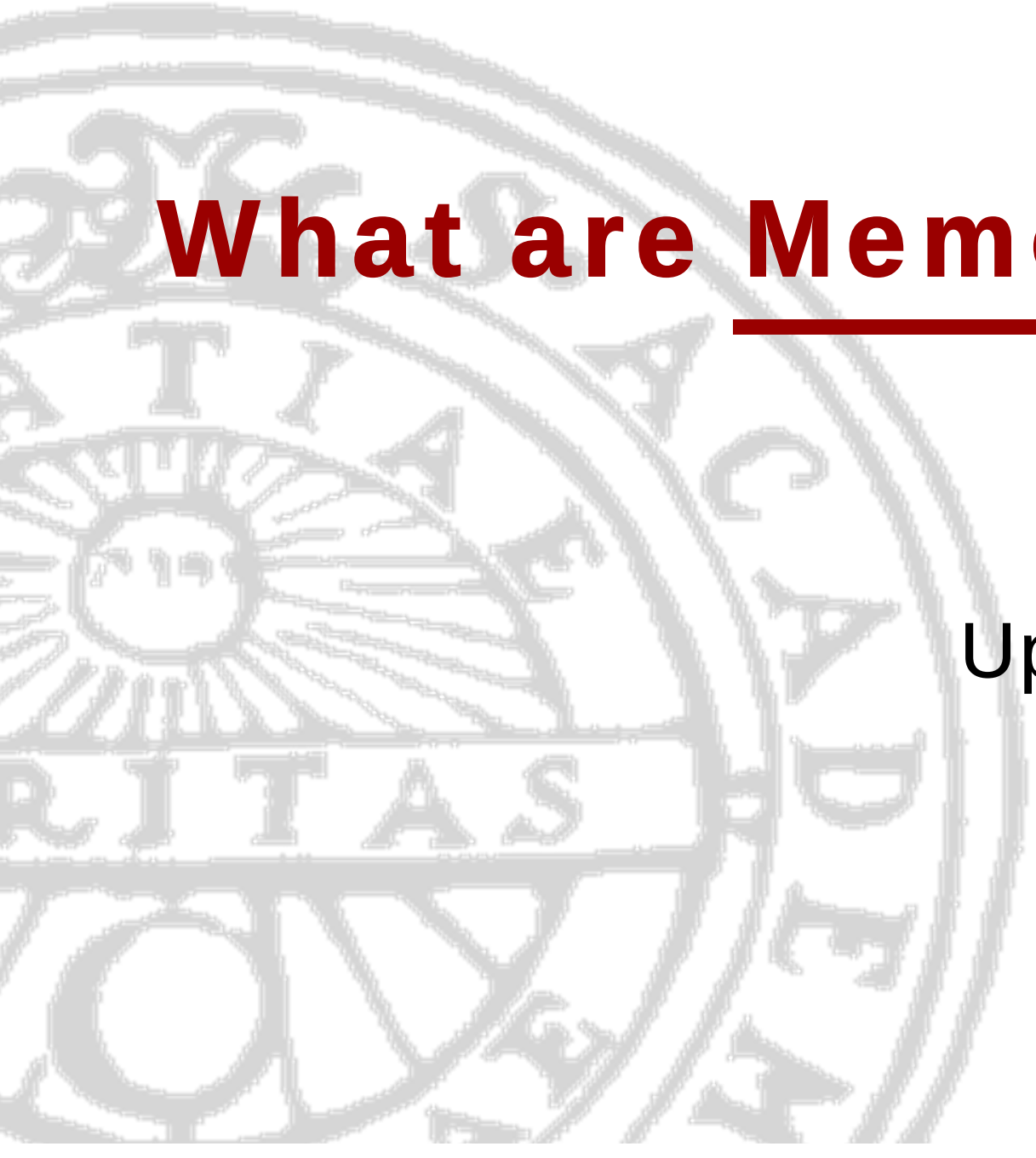
The following four bus transaction types can be seen on the snooping bus connecting the CPUs:

- **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).

Example of a state transition sheet:

CPU action	Bus Transaction (if any)	State/value after the CPU action						Data is provided by [Cache 1, 2, 3 or Mem] (if any)
		CPU1		CPU2		CPU3		
		A	B	A	B	A	B	
Initially		I	I	I	I	I	I	
CPU3: LD A	RTS(A)					S/1		Mem
CPU2: LD A	RTS(A)			S/1				Mem
CPU1: LD A	RTS(A)	S/1						Mem
CPU3: LDA	—							—



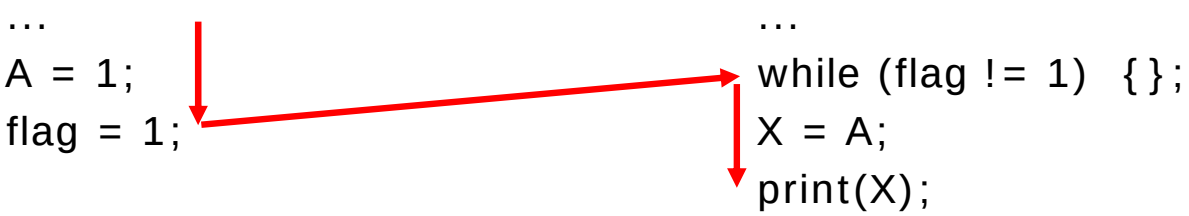
What are Memory Models?

Erik Hagersten
Uppsala University
Sweden

Where Memory Models Matters

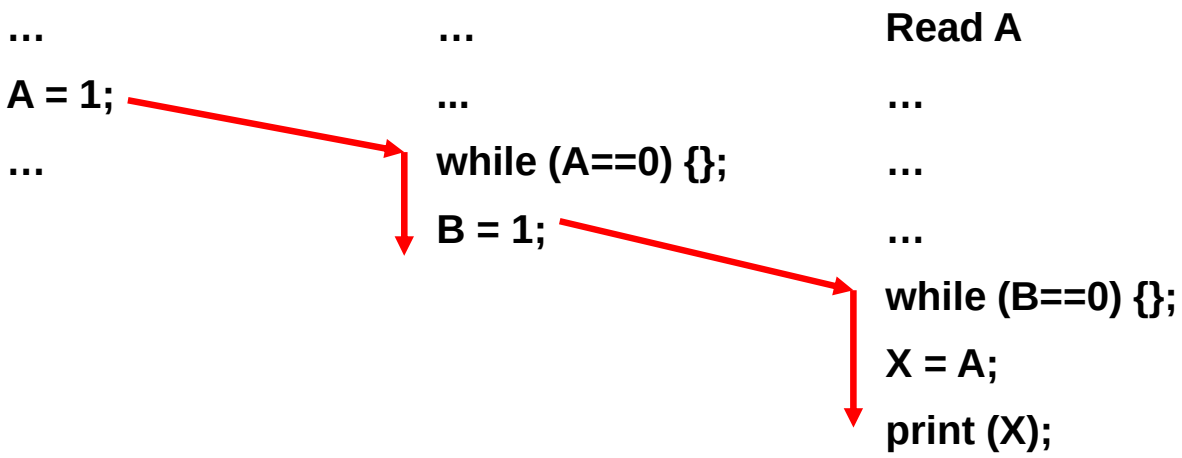
- Flag synchronization

(initially flag = 0 and A = 0)



- Causality (Causal correctness)

(Initially A = 0 and B = 0)



UPPSALA UNIVERSITET

Dekker's Algorithm (mutual exclusion)

Initially A = B = 0

"fork"

A := 1

if (B== 0) print("A won")

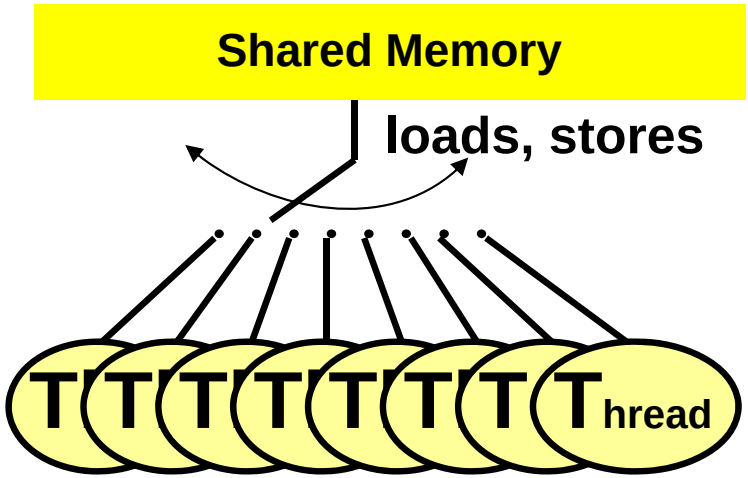
B := 1

if (A == 0) print("B won")

Does the write become globally visible before the read is performed?

UPPSALA UNIVERSITET

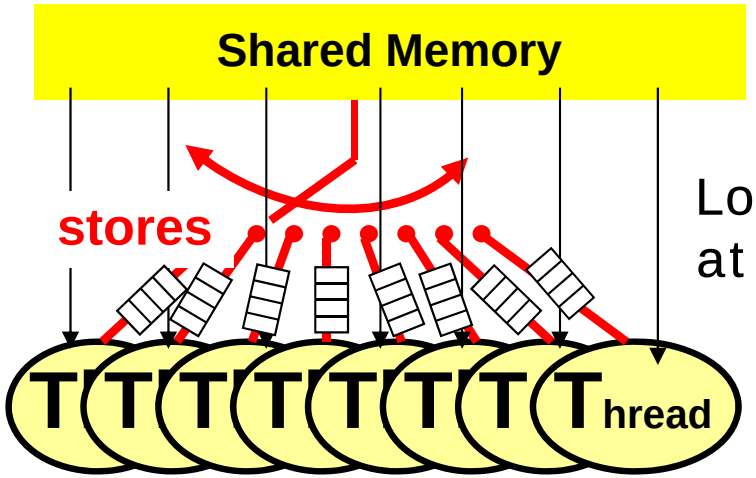
"The intuitive memory order" Sequential Consistency (Lamport)



- Global order achieved by *interleaving* <u>all</u> memory accesses from different threads
- SW should not be able to detect contradictive orders
- "Programmer's intuition is maintained"
- Unnecessarily restrictive ==> performance penalty

AVDARK 2013

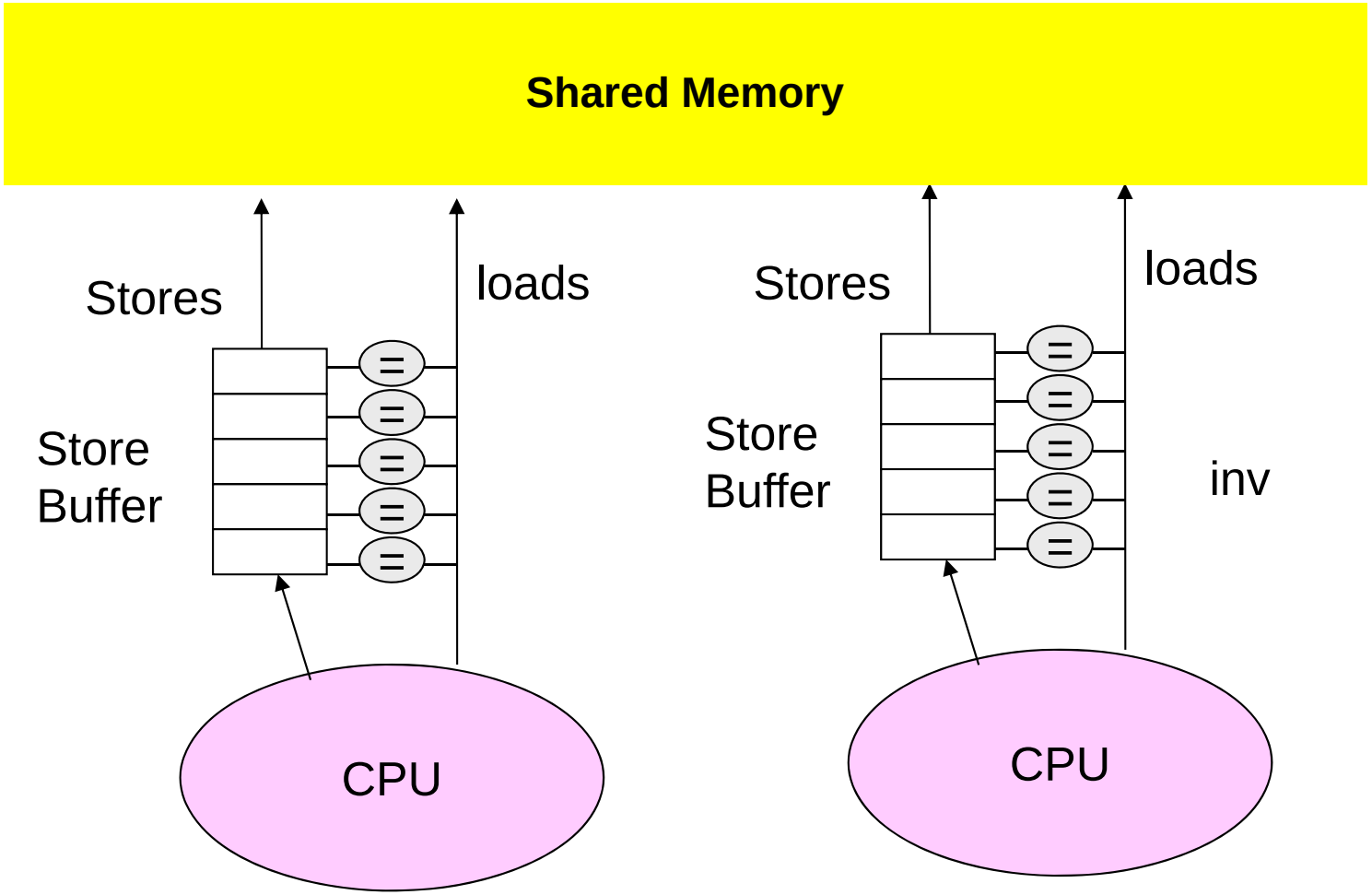
"Almost intuitive memory model" Total Store Ordering [TSO] (P. Sindhu)



Loads(each thread can have at most one outstanding load)

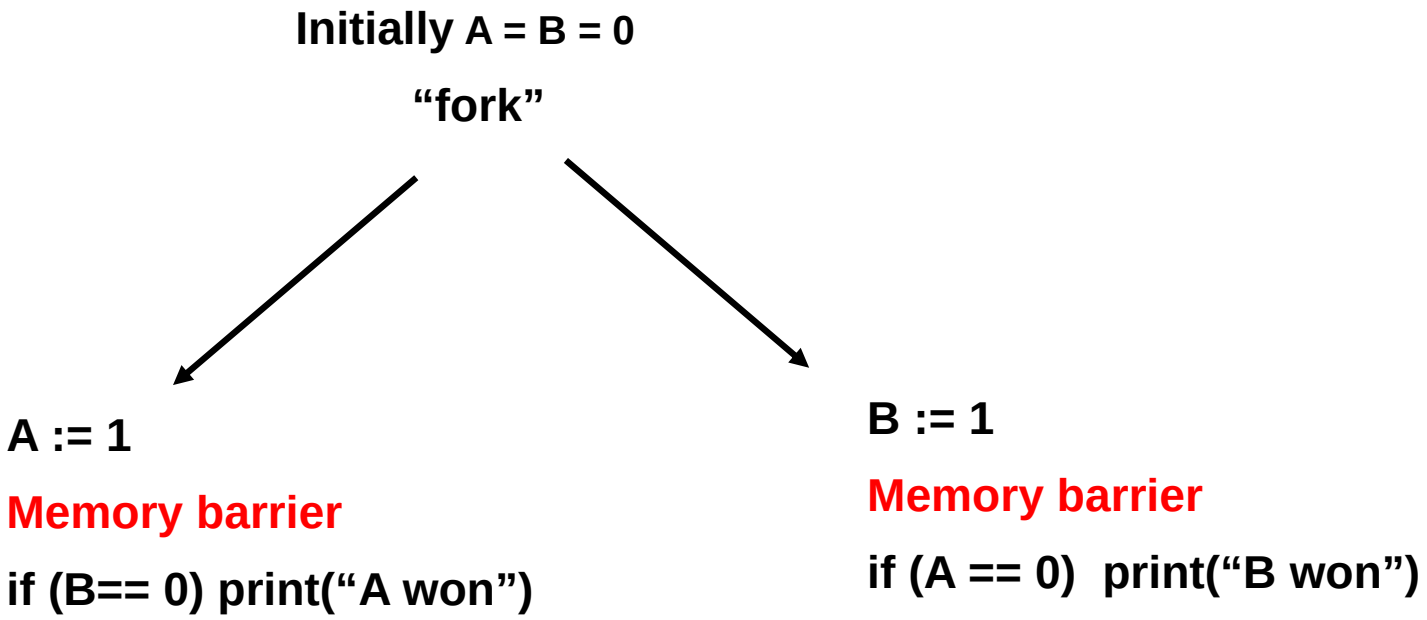
- Global *interleaving* [order] for <u>all</u> stores from different threads (own stores excepted)
- "Programmer's intuition is almost maintained"
  - Flag synchronization? Yes
  - Store causality? Yes
  - Does Dekker work? No
- Unnecessarily restrictive ==> performance penalty

TSO HW Model



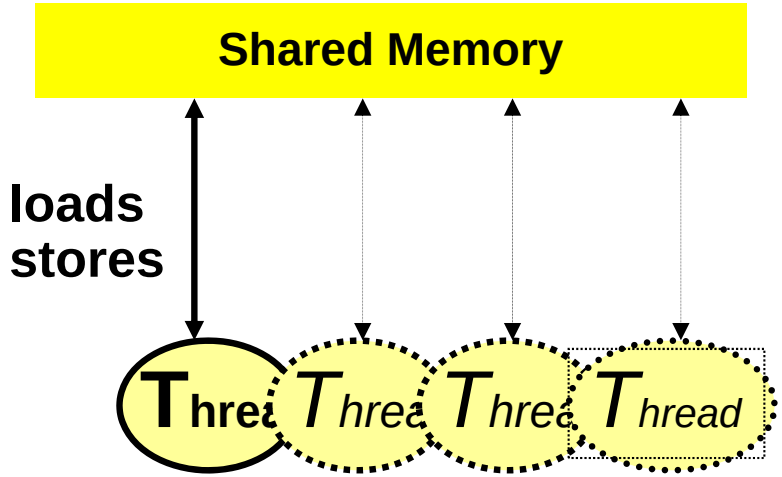
➔Stores are moved off the critical path
Coherence implementation can be the same as for SC

Dekker's Algorithm for TSO



UPPSALA UNIVERSITET

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



- **Most accesses are unordered**
- **"Programmer's intuition is not maintained"**
  - Flag synchronization? No
  - Causal correctness? No
  - Dekker? No
- **Global order only established when the programmer explicitly inserts memory barrier instructions**

++ Better performance!!

-- Interesting bugs!!

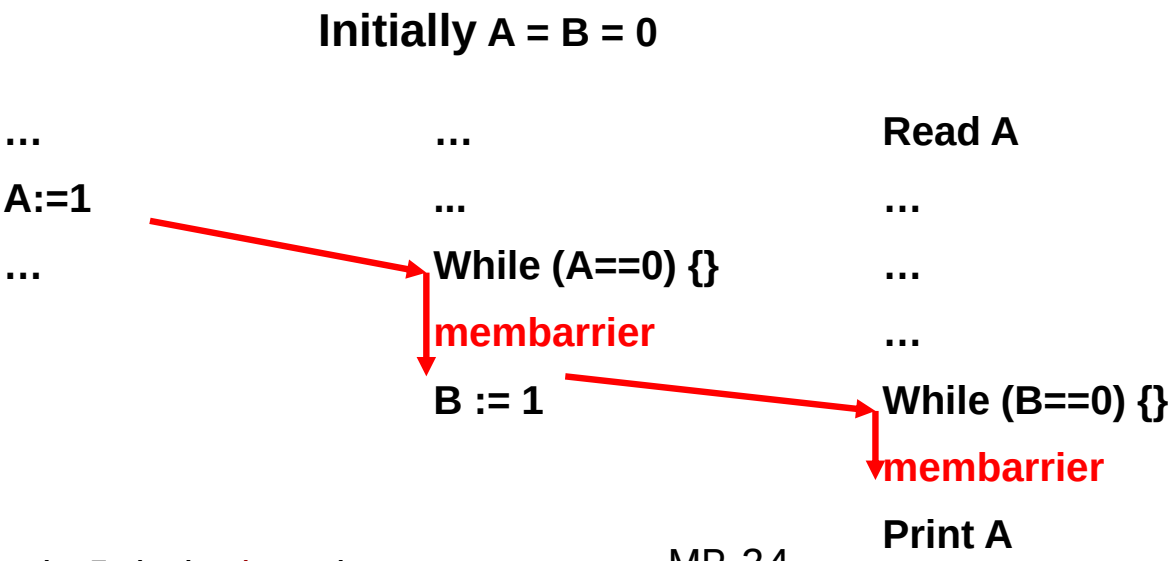
AVDARK 2013

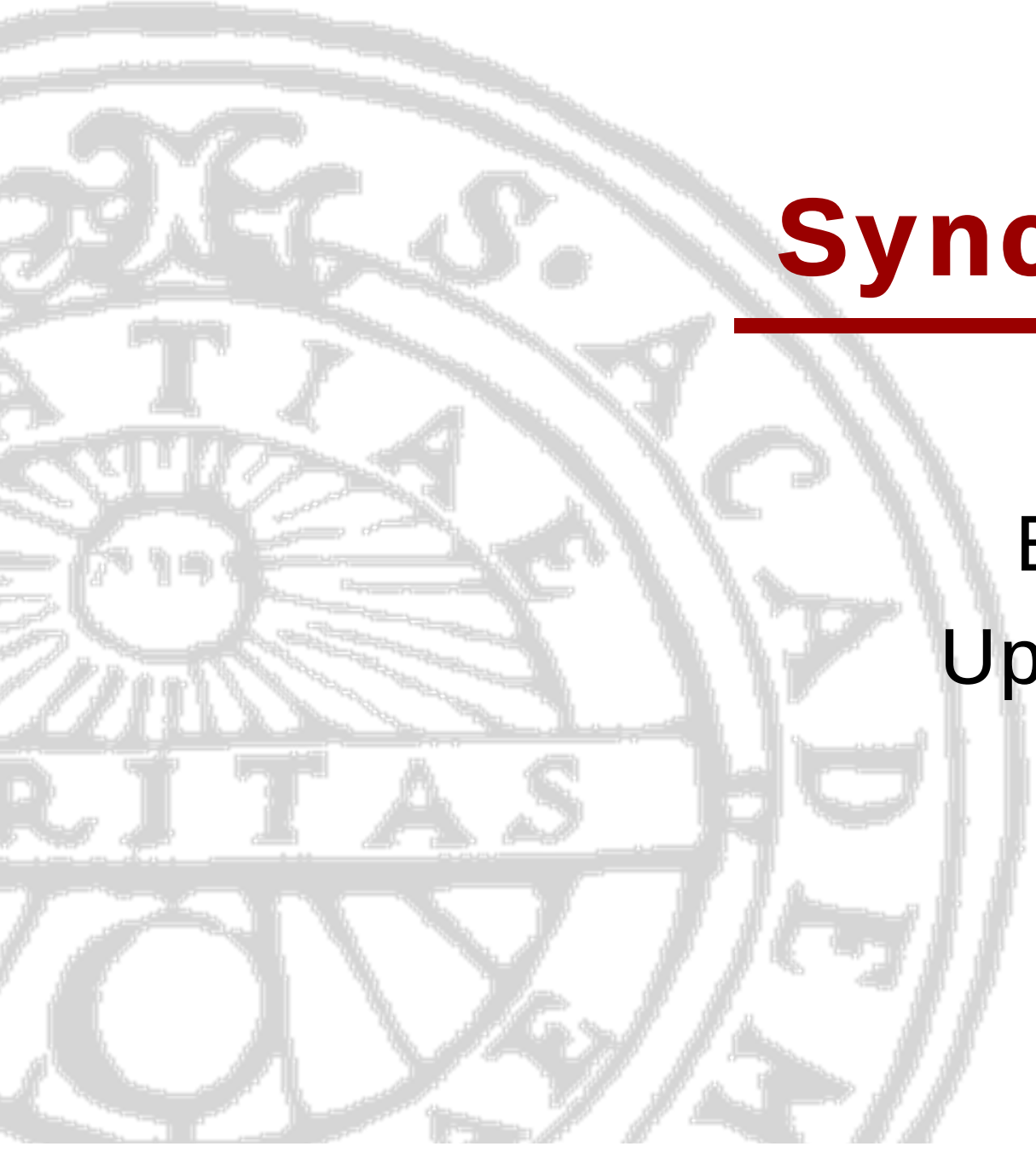
Weak/Release consistency

- 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





Synchronization

Erik Hagersten
Uppsala University
Sweden

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 synchronization (enter critical section, go past sync 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

A Bad Example: ”POUNDING”

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

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

Assume: The function TAS[addr] returns the current memory value at addr and ***atomically*** *writes the busy pattern “1” to the memory*

Spinning threads produce traffic!

UPPSALA UNIVERSITET

Optimistic Test&Set Lock ”spinlock”

```
proc lock(lock_variable) {
    while true {
      if (TAS[lock_variable] ==0)  break;     /* pound 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!
More on this during Scalable Synchronization

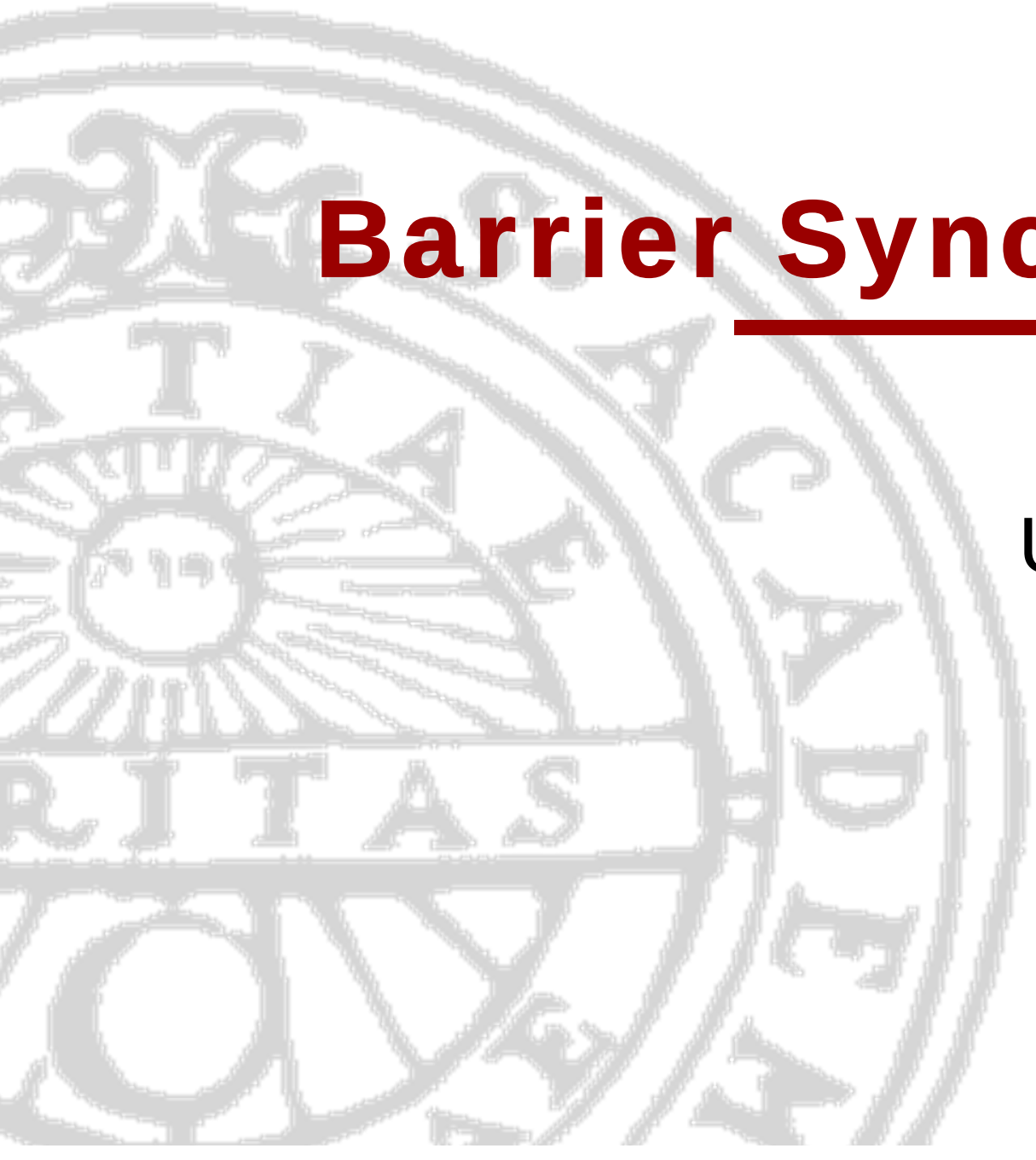
UPPSALA UNIVERSITET

Pesimistic Test&Set Lock ”spinlock”

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

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

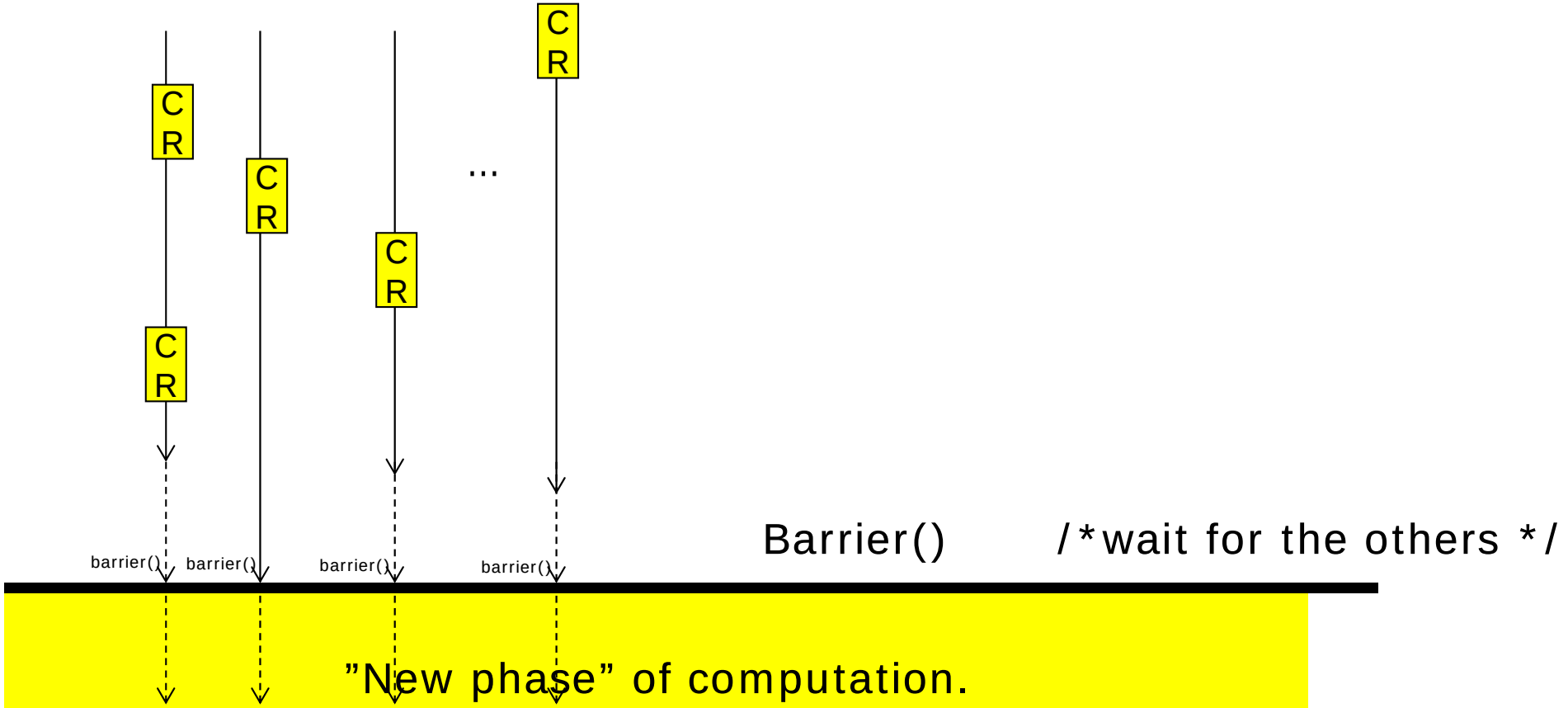
Slightly less traffic than Optimistic for contended locks
-- still lots of traffic at lock handover!
More solutions during Scalable Shared Memory



Barrier Synchronization

Erik Hagersten
Uppsala University

Barrier Synchronization



UPPSALA UNIVERSITET

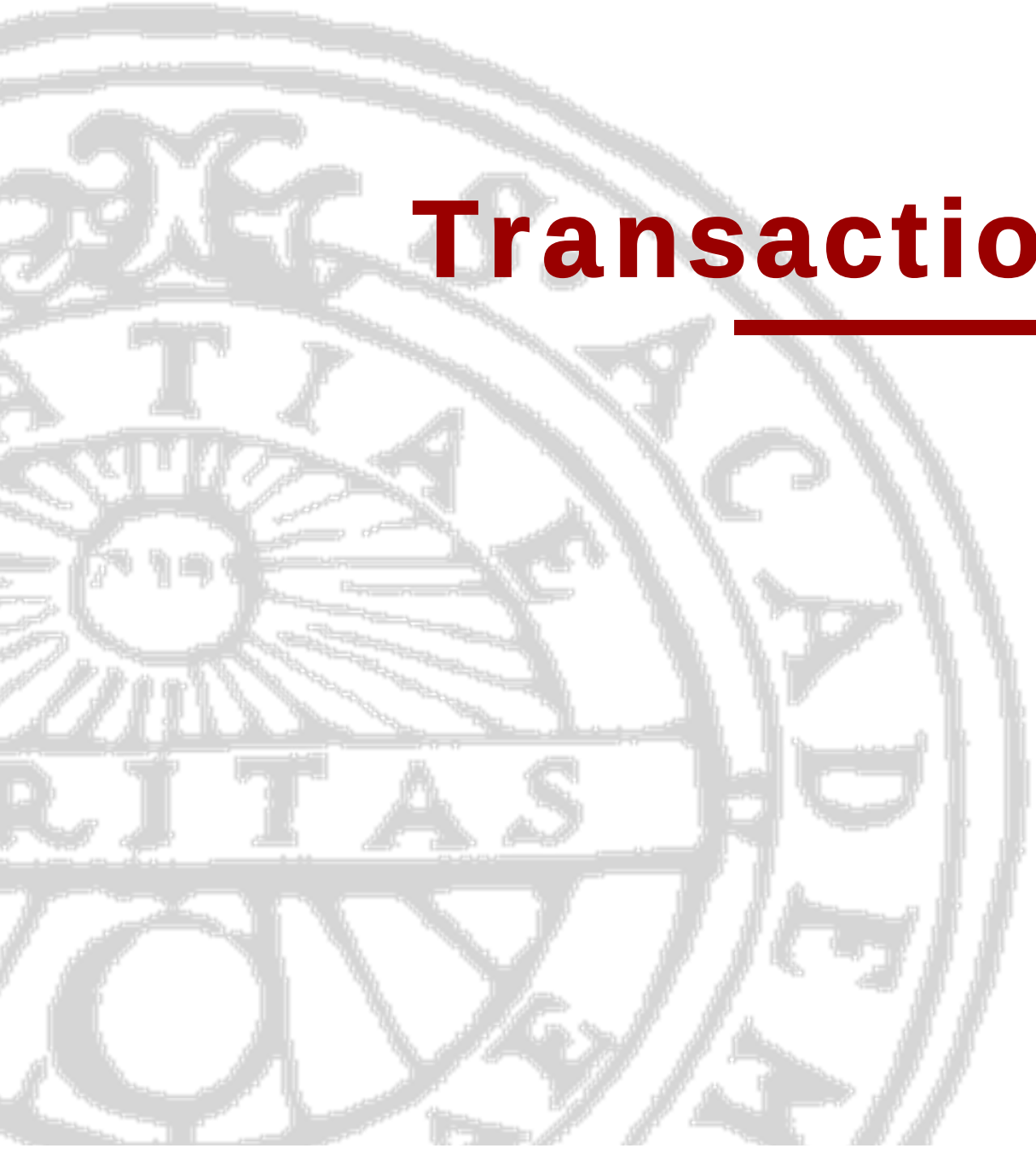
A Naiive Centralized Barrier

```
BARRIER (bar_name, p) {

 LOCK(bar_name.lock) {
    if (bar_name.ctr == p) bar_name.ctr = 0; /* init count*/
    bar_name.ctr++;                          /* globally increment the barrier count */
 }
 UNLOCK(bar_name.lock)

  while (bar_name.ctr < p) {};               /* wait for the last thread */

 }
```



Transactional Memory

Erik Hagersten

UPPSALA UNIVERSITET

```
lock(L);
C = B + 1;
A = A + 1;
unlock(L);

start_transaction();
C = B + 1;
A = A + 1;
end_transaction();
```

AVDARK 2013

UPPSALA UNIVERSITET

New kind of synchronization: Transactional Memory (TM)

```
start_transaction();
C = B + 1;
A = A + 1;
end_transaction();
```

- Traditional critical section: lock(ID); unlock(ID) around critical sections
- TM: start_transaction; end_transaction around ”critical sections” (note: no ID!!)
  - Underlying mechanism to guarantee atomic behavior by rollback mechanisms
  - This is not the same as guaranteeing that only one thread is in the critical action!!
- Suggested by Maurice Herlihy in 1993
- Supported in HW (recent) or in SW (normally very inefficient)

UPPSALA UNIVERSITET

Support for TM

```
start_transaction();
C = B + 1;
A = A + 1;
end_transaction();
```

- Start_transaction:
  - Save original state to allow for rollback (i.e., save register values)
- In critical section
  - Make no global state changes [to memory]
  - Detect ”atomic violations” (others writing data are reading in CS or reading data you are writing in CS)
  - On atomic violation: roll-back to original state
  - Forward progress must be guaranteed
- End_transaction
  - Atomically commit all global state changes performed in the critical section.

UPPSALA UNIVERSITET

Advantage of TM

```
start_transaction();
C = B + 1;
A = A + 1;
end_transaction();
```

- Do not have to ”name” CS
- Less risk for deadlocks (e.g., nested locks)
- Potential performance advantage:
  - 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 potentially small performance penalty
- Performance problems with large ”commit state” and rollback overhead

AVDARK 2013

UPPSALA UNIVERSITET

TM Implementations

- Many suggestion for software TM (STM)
- Implemented in Sun's Rock SPARC (RIP)
- Support for small transactions in AMD x86
- Decent support in IBM's BlueGene-Q
- Better support in Power6
- **Support in Haswell (latest Intel x86)**
- The jury is still out…

AVDARK 2013