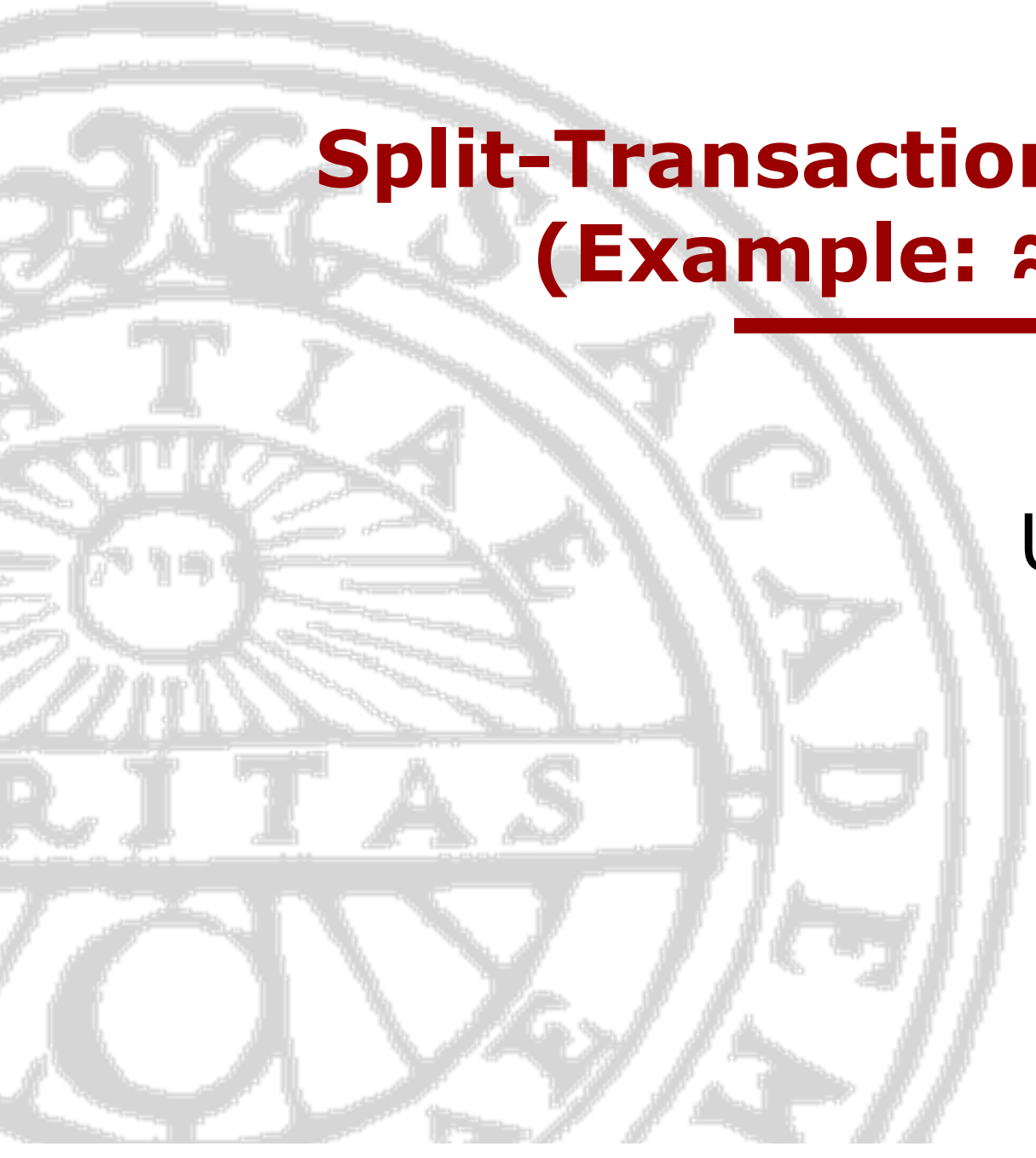
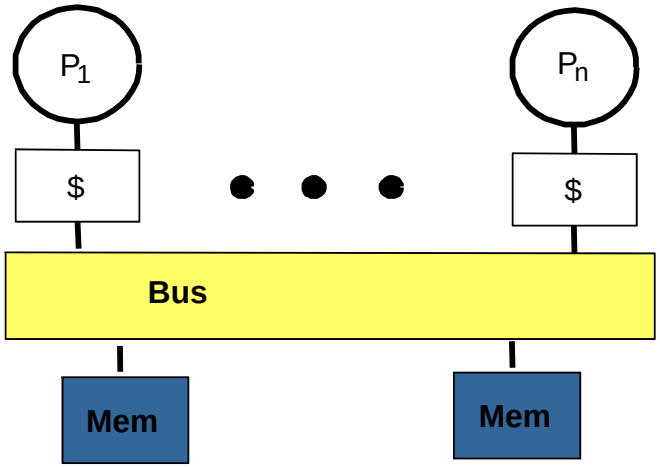


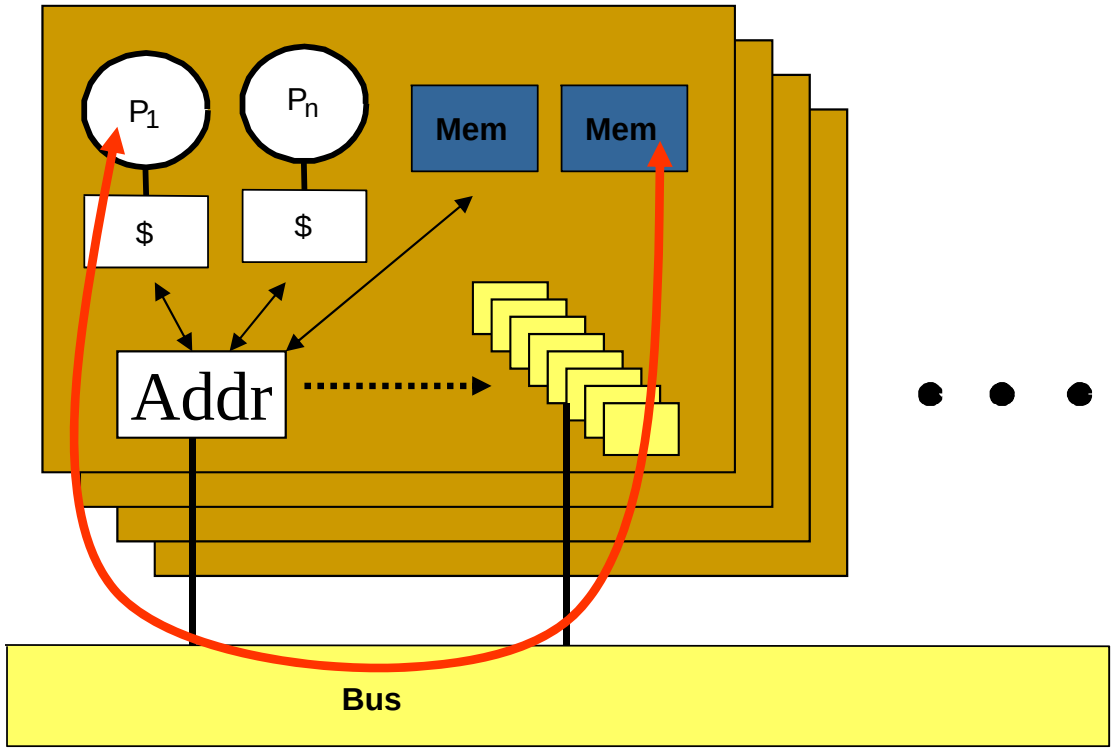
Split-Transaction Bus Coherence (Example: ≈ E6000 and P6)

Erik Hagersten
Uppsala University

E6000 -- Looks like a NUMA but drives like a UMA



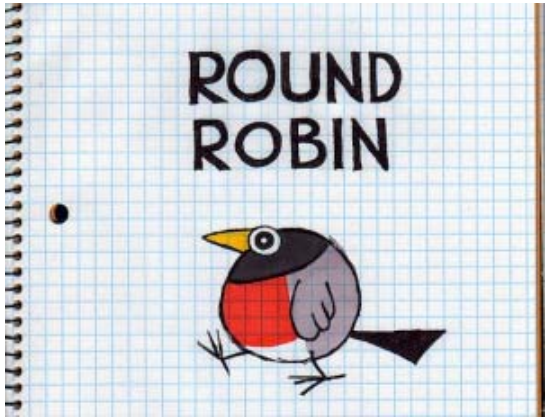
Logical View



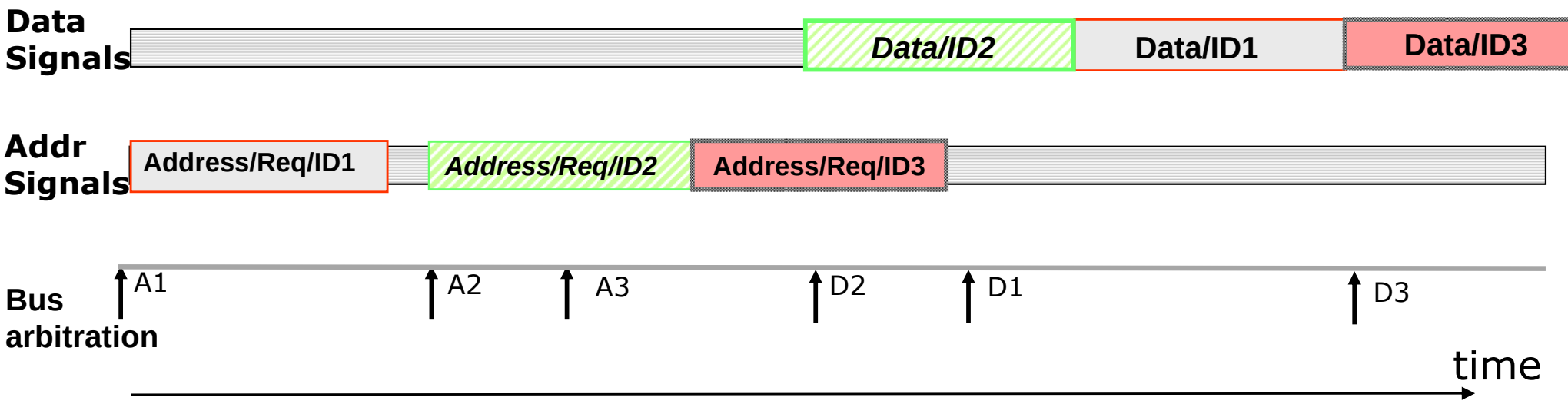
Physical Viewl

- Up to 16 CPU modules (each with 2xCPU + 2xMem)
- Memory bandwidth scales with the processor count,
... bus bandwidth does not.
- Optimize for the UMA case (no memory shortcut)

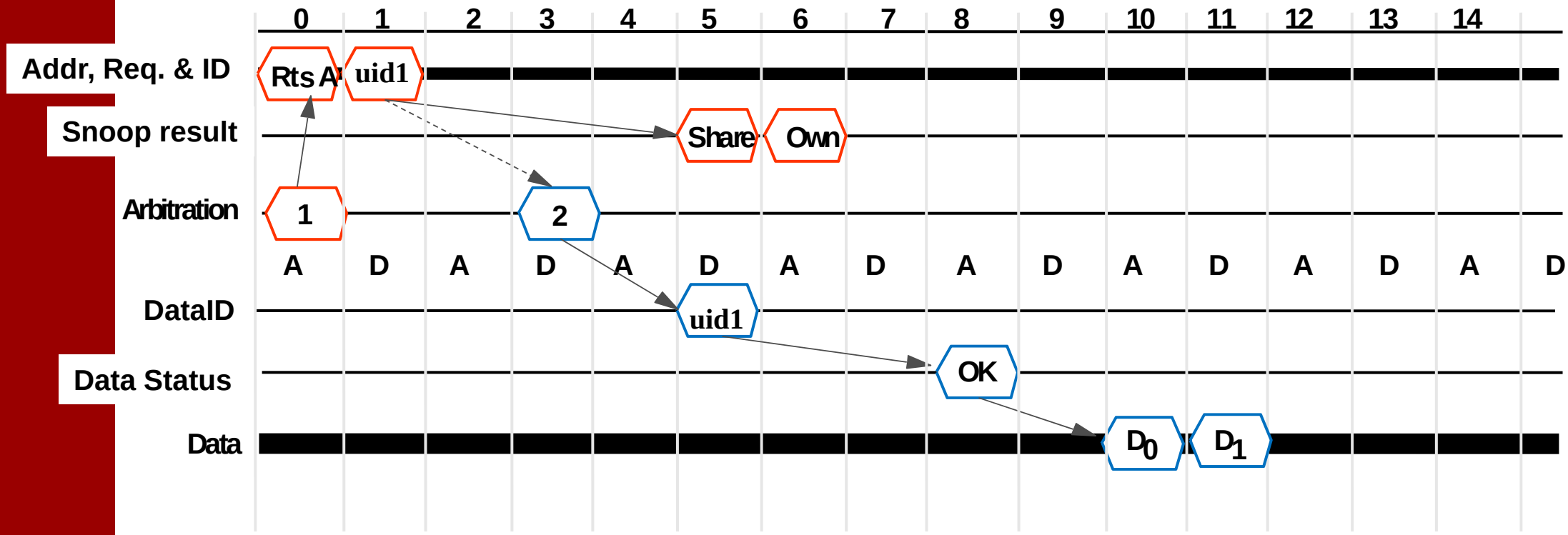
Split-Transaction Bus



- Split bus transaction
  - Request transaction / Data transaction
  - Separate arbitration for addr/data transaction (round robin)
- Out-of-order response
  - Response is matched to request using unique identifiers
  - Buffering requests between bus and snoop state machines



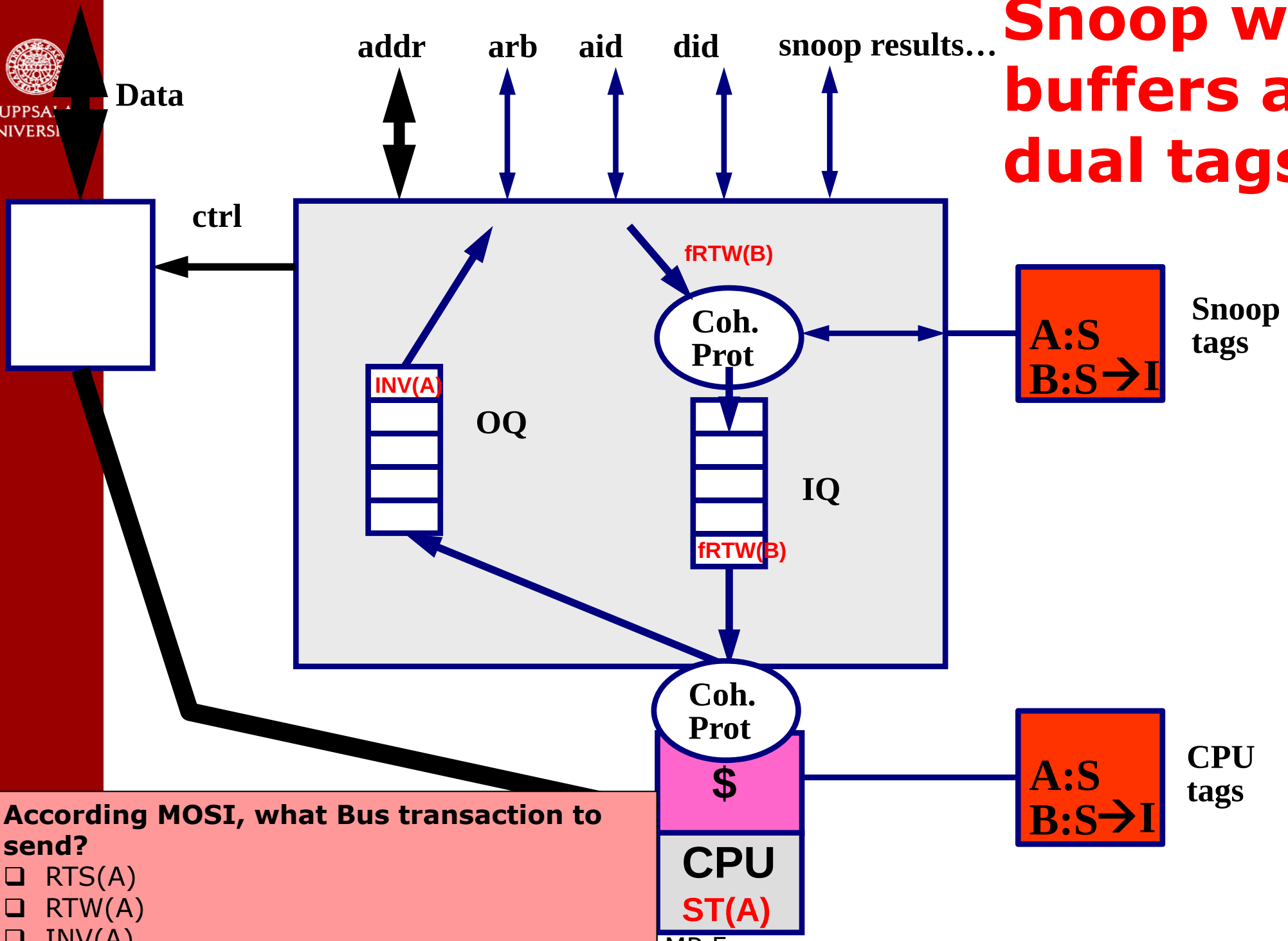
Timing of a single read trans
Board 1 reading from mem 2



= Fixed latency

= Flexible latency (shortest possible latency shown)

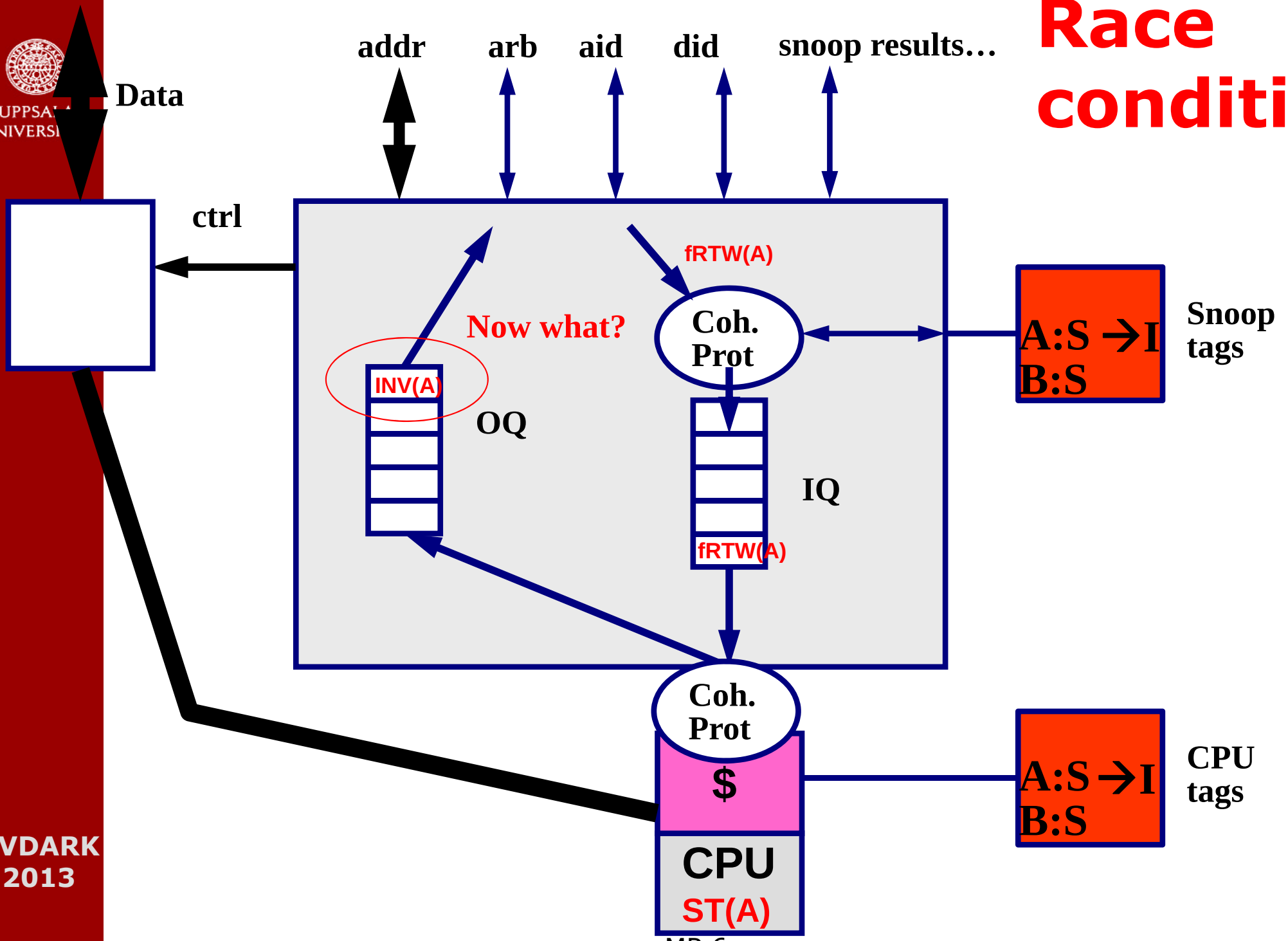
Snoop with buffers and dual tags



According MOSI, what Bus transaction to send?

- ❑ RTS(A)
- ❑ RTW(A)
- ❑ INV(A)

Race condition

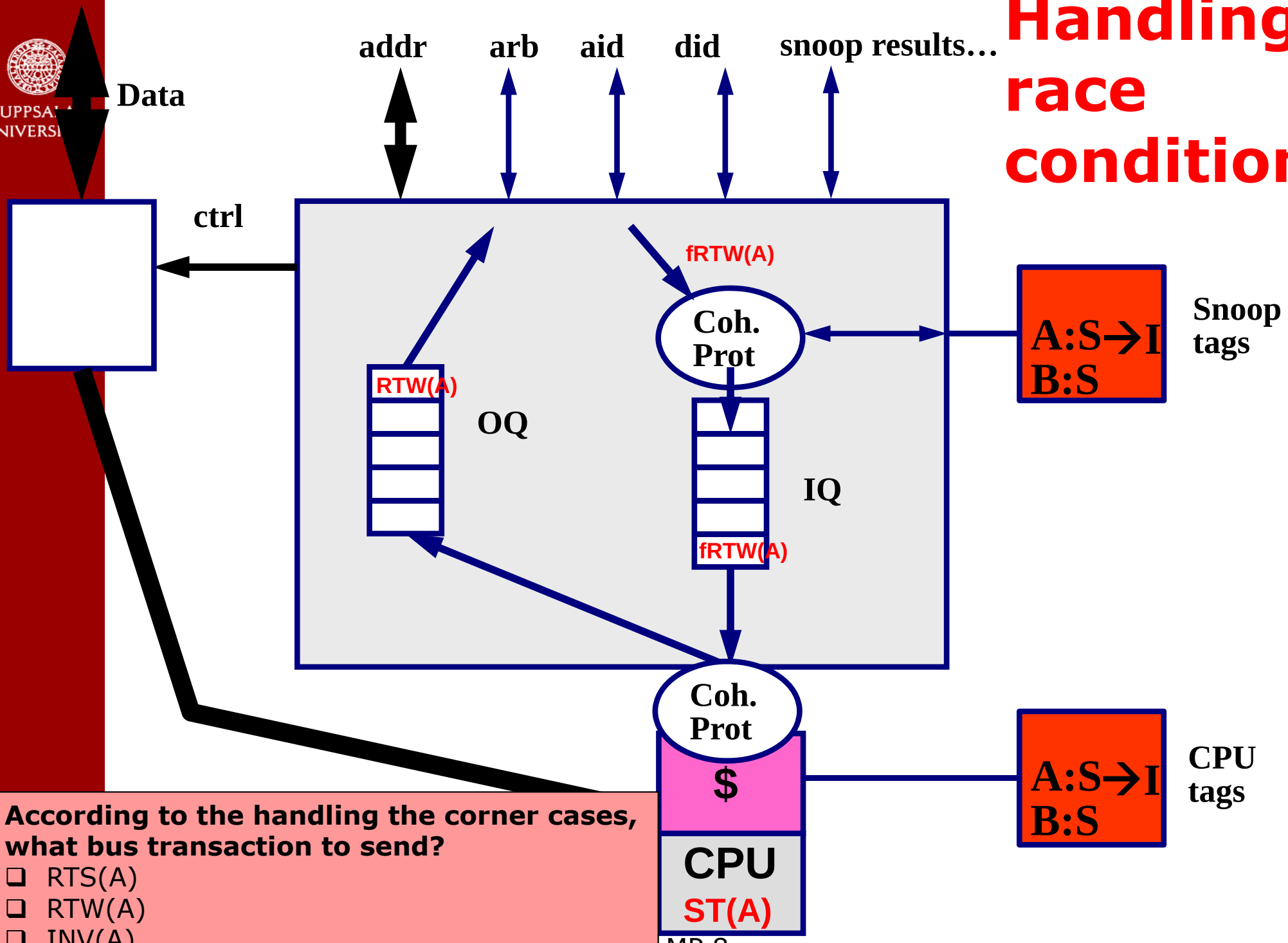


UPPSALA UNIVERSITET

Implementing MOSI (also MOESI) on split transaction, dual tags

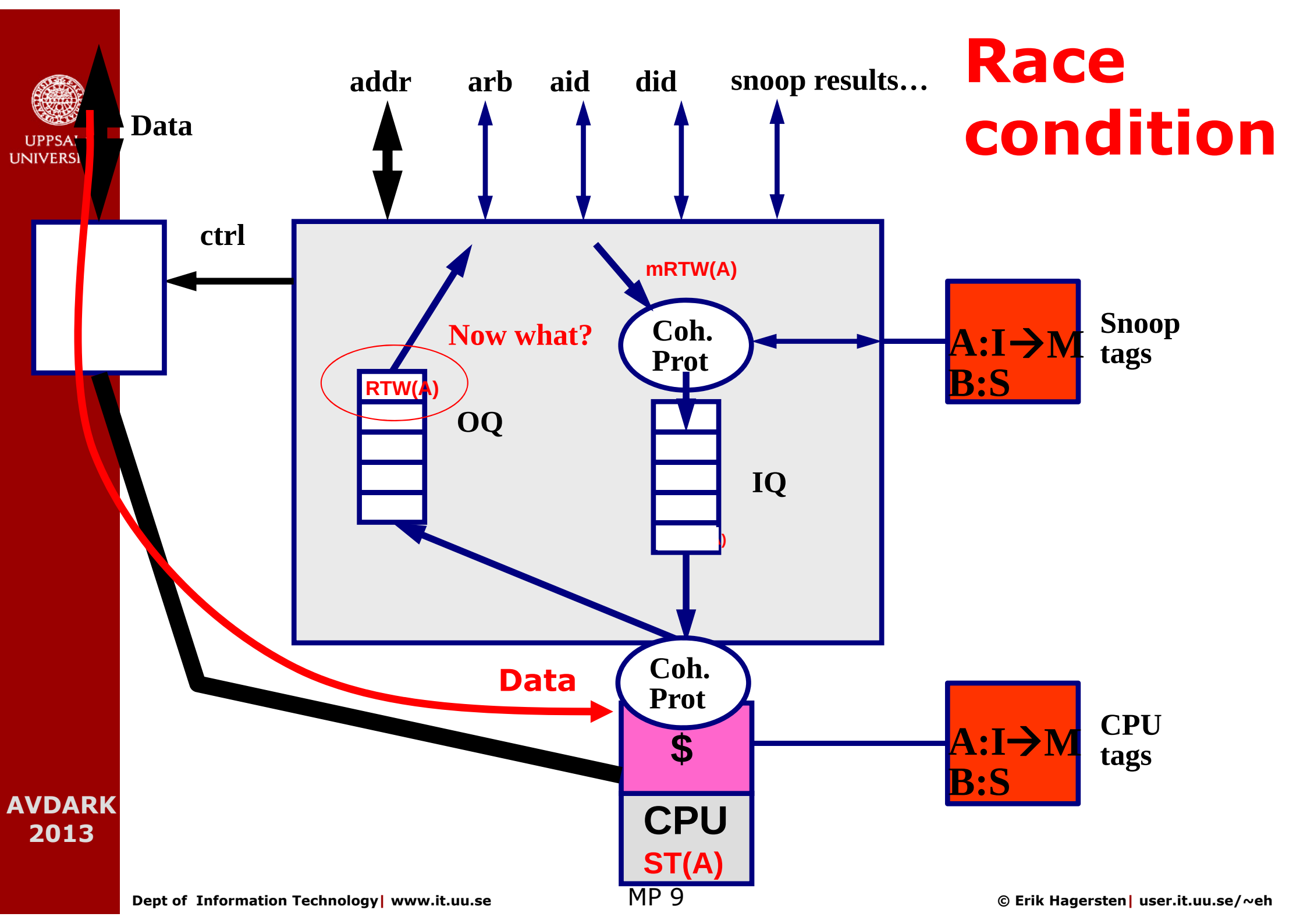
- Need to hande corner cases due to buffering
- Actually, only three bus transactions: RTW, RTS, WB
- On a Store, put RTW in the Output Queue (OQ) even if you have the data
- My RTW(A):
  - Assert ”Shared” if I have the data (cancel data transfer)
- Foreign RTW(A) with Shared asserted
  - ≈ INV(A)
- Foreign RTW(A)
  - Assert ”Owned” if I have the data in M or O (cancel data from Mem)
- Foreign RTS(A):
  - Assert ”Shared” if I have the data (prevents requester → E)
  - Assert ”Owned” if I have the data in M or O (cancel data from Mem)
- My WB(A)
  - Assert ”Owned” if I no longer have the data (cancel WB to Mem)

Handling race conditions

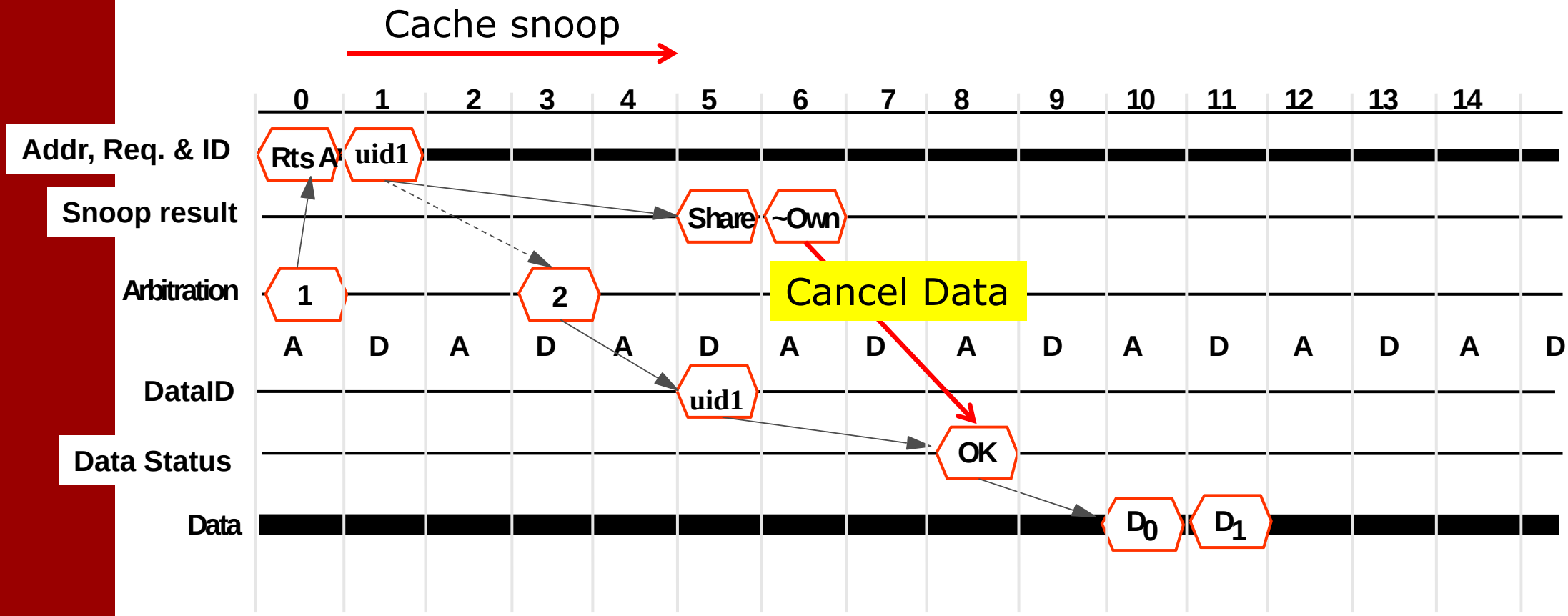


According to the handling the corner cases, what bus transaction to send?

- ❑ RTS(A)
- ❑ RTW(A)
- ❑ INV(A)


Race condition
addr
arb
aid
did
snoop results…
Data
ctrl
mRTW(A)
Now what?
Coh. Prot
A:I→M
B:S
Snoop tags
RTW(A)
OQ
IQ
Coh. Prot
Data
$
CPU
ST(A)
A:I→M
B:S
CPU tags

Timing of a single read trans
Board 1 reading from mem 2



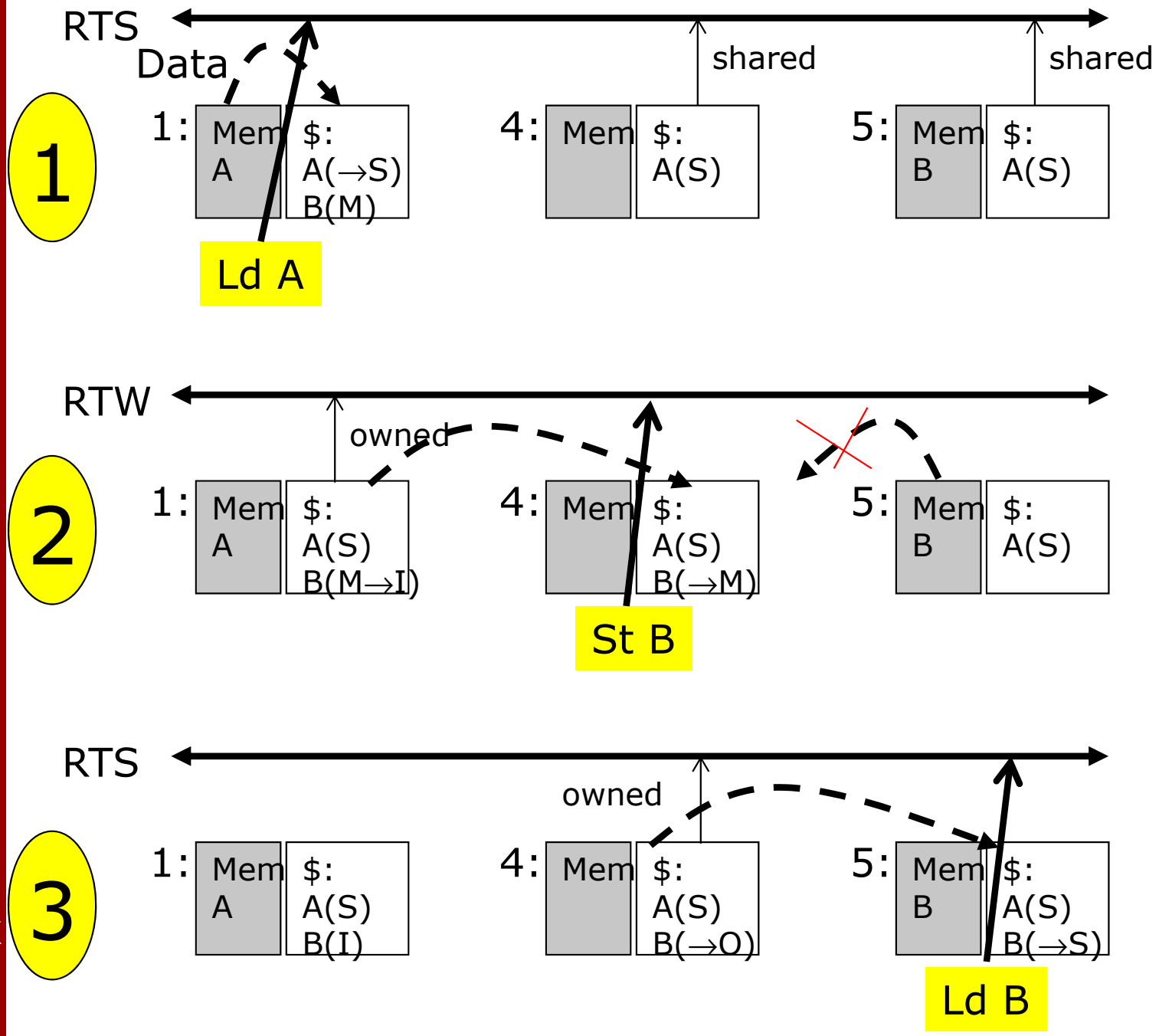
= Fixed latency

= Flexible latency (shortest possible latency shown)

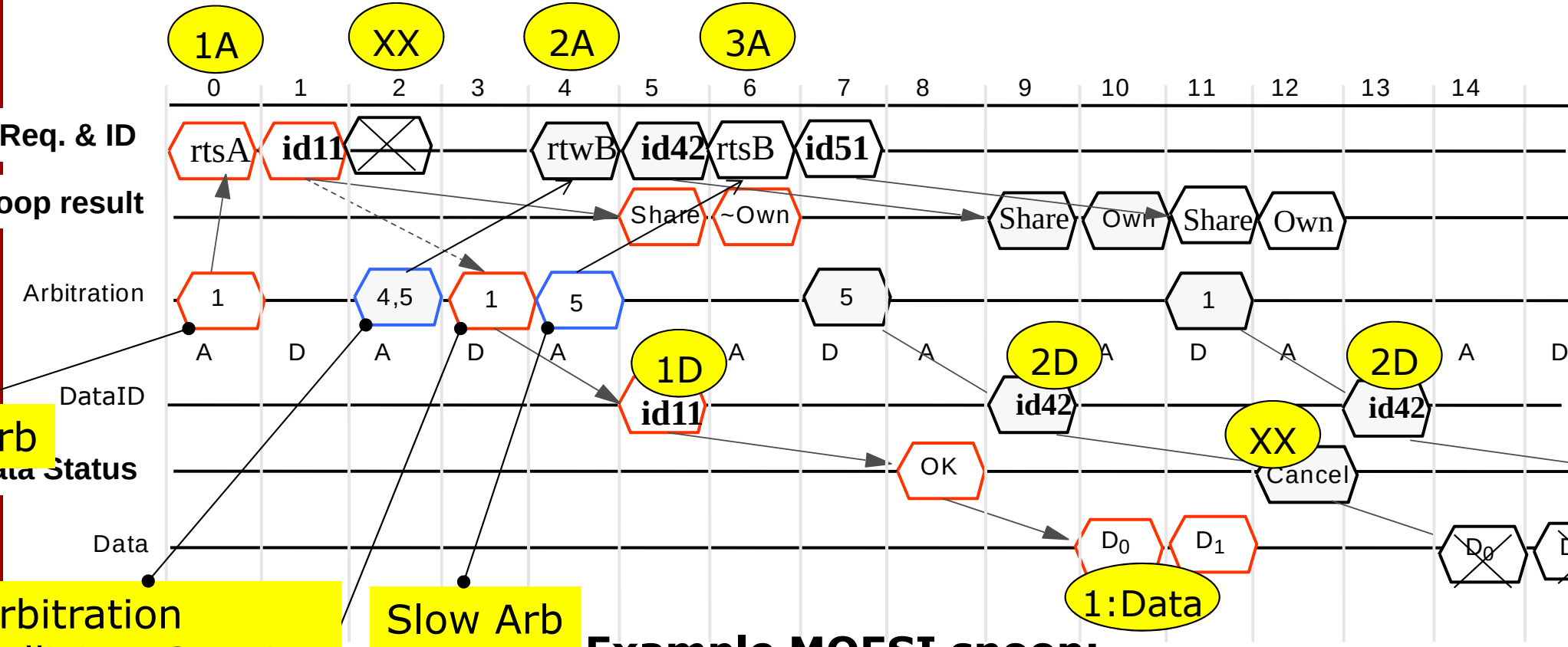
Handling multiple transactions

Erik Hagersten
Uppsala University

EXAMPLE: MOESI snoop



Gigaplane Bus Timing (MOESI)



Fast Arb

Arbitration Collision, 2A wins

Slow Arb by "3A"

MEM2 arbiting for Data Earliest possible time (may also be later)

Example MOESI snoop:

1: CPU1 sends RTS (A), data in MEM1 ➔ state S

2: CPU4 sends RTW(B), in MEM5, owned by CPU1

3: CPU5 sends RTS (B), owned by CPU4

UPPSALA UNIVERSITET

A cascade of five stores to address A

Initially, A resides in CPU7's cache in state M

On the bus:

- **CPU1: RTW, ID=a**
- **CPU2: RTW, ID=b**
- **…**
- **CPU5: RTW, ID=f**

After the five bus transactions have been sent:

CPU tags		Snoop tags
I	**IQ1 = <mRTW$_{IDa}$, fRTW$_{IDb}$>**	*I*
I	**IQ2 = <mRTW$_{IDb}$, fRTW$_{IDc}$>**	*I*
	…	
I	**IQ5 = <mRTW$_{IDf}$>**	*M*
	…	
M	**IQ7= < fRTW$_{IDa}$>**	*I*

This cache initially had the data

AVDARK 2013

UPPSALA UNIVERSITET

A cascade of five stores to address A

A initially resides in CPU7's cache in state M

On the bus:

- **CPU1: RTW, ID=a**
- **CPU2: RTW, ID=b**
- **…**
- **CPU5: RTW, ID=f**

Servicing the transaction in IQ7

CPU tags		Snoop tags
M	IQ1 = < fRTW$_{IDb}$>	I
I	IQ2 = <mRTW$_{IDb}$, fRTW$_{IDc}$>	I
	…	
I	IQ5 = <mRTW$_{IDf}$>	M
	…	
I	IQ7= < >	I

Data

This cache initially had the data

UPPSALA UNIVERSITET

A cascade of five stores to address A

A initially resides in CPU7's cache in state M

On the bus:

- **CPU1: RTW, ID=a**
- **CPU2: RTW, ID=b**
- **…**
- **CPU5: RTW, ID=f**

Servicing fRTW in IQ1

CPU tags	Data	Snoop tags
I	IQ1 = < >	*I*
M	IQ2 = < $fRTW_{IDc}$ >	*I*
	…	
I	IQ5 = <$mRTW_{IDf}$>	*M*
	…	
I	IQ7= < >	*I*

This cache initially had the data

What memory model is implemented assuming that everything else is ideal?

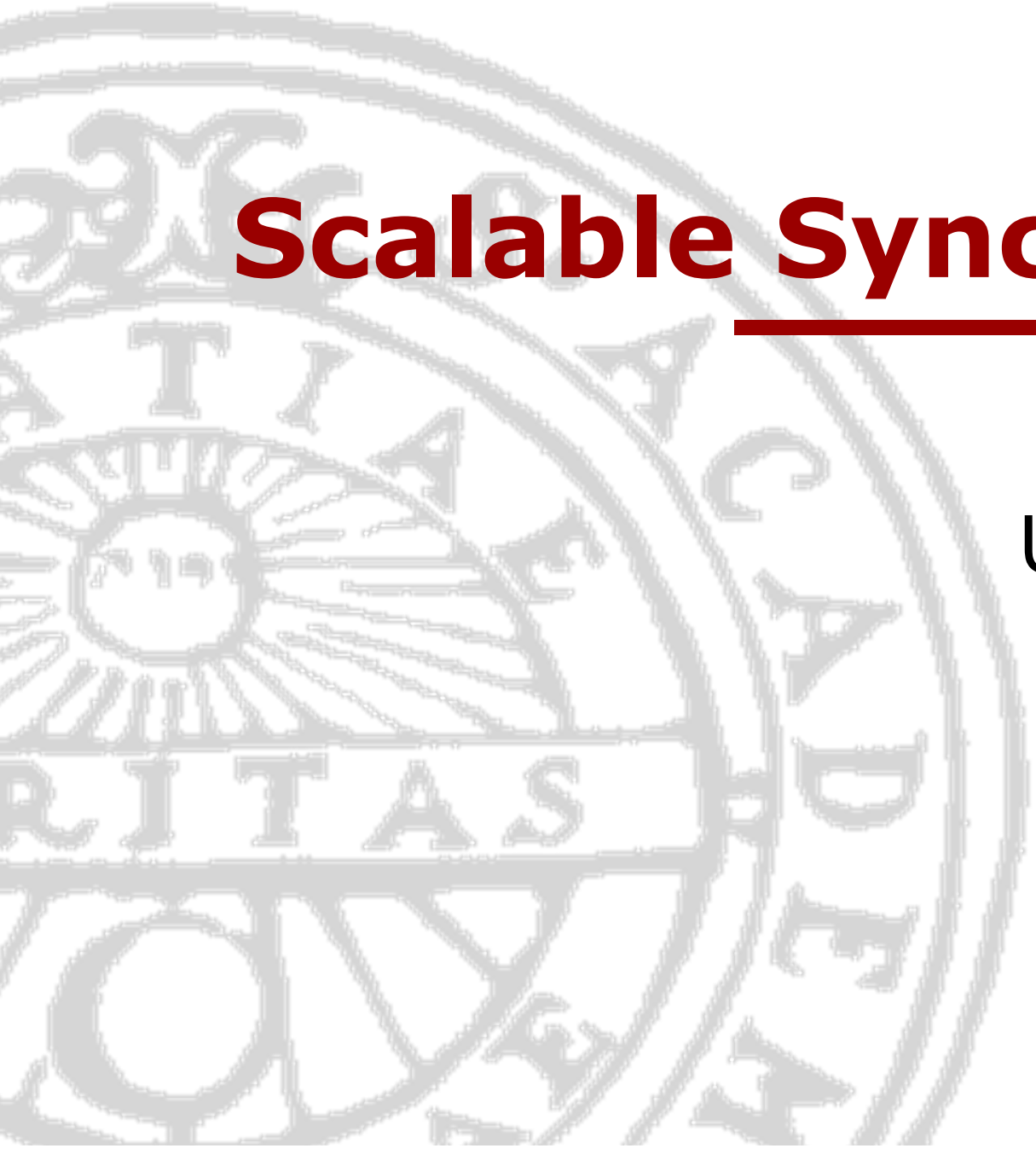
- ❑ Sequential Consistency
- ❑ Total Store Order
- ❑ Weak Consistency

What memory model is implemented assuming the CPUs have store buffers

- ❑ Sequential Consistency
- ❑ Total Store Order
- ❑ Weak Consistency

Summary

- Transactions are serviced by each cache in the order they occur on the bus (if at all)
- Each IQ contains that partial bus order
- A cache maintains its access rights until it services a request from its IRQ
- Reading mRTW-Shared (≈mINV) from IRQ gives write permission right away
- mRTW and mRTS that requires data can only be serviced once data has arrived



Scalable Synchronization

Erik Hagersten
Uppsala University

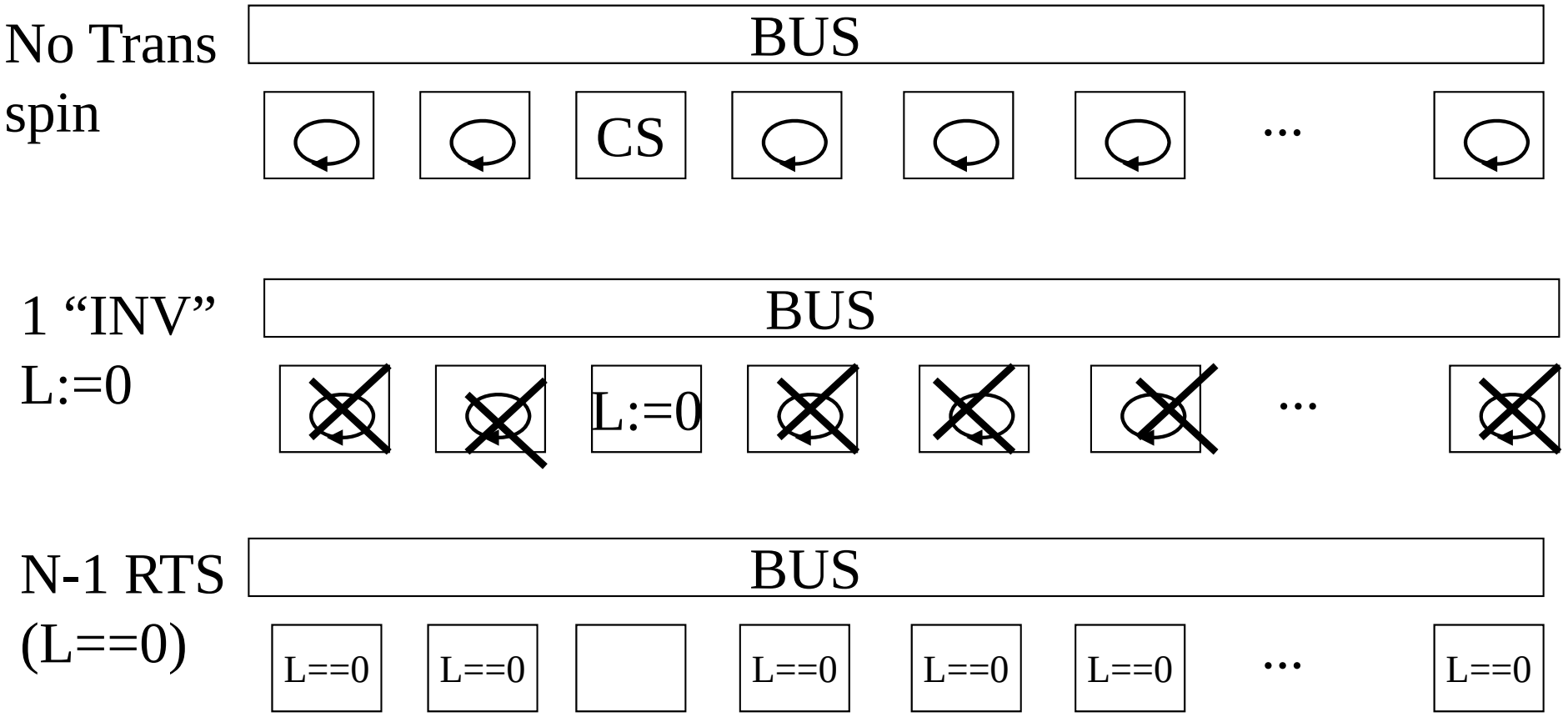
UPPSALA UNIVERSITET

Optimistic Test&Set Lock ”spinlock”

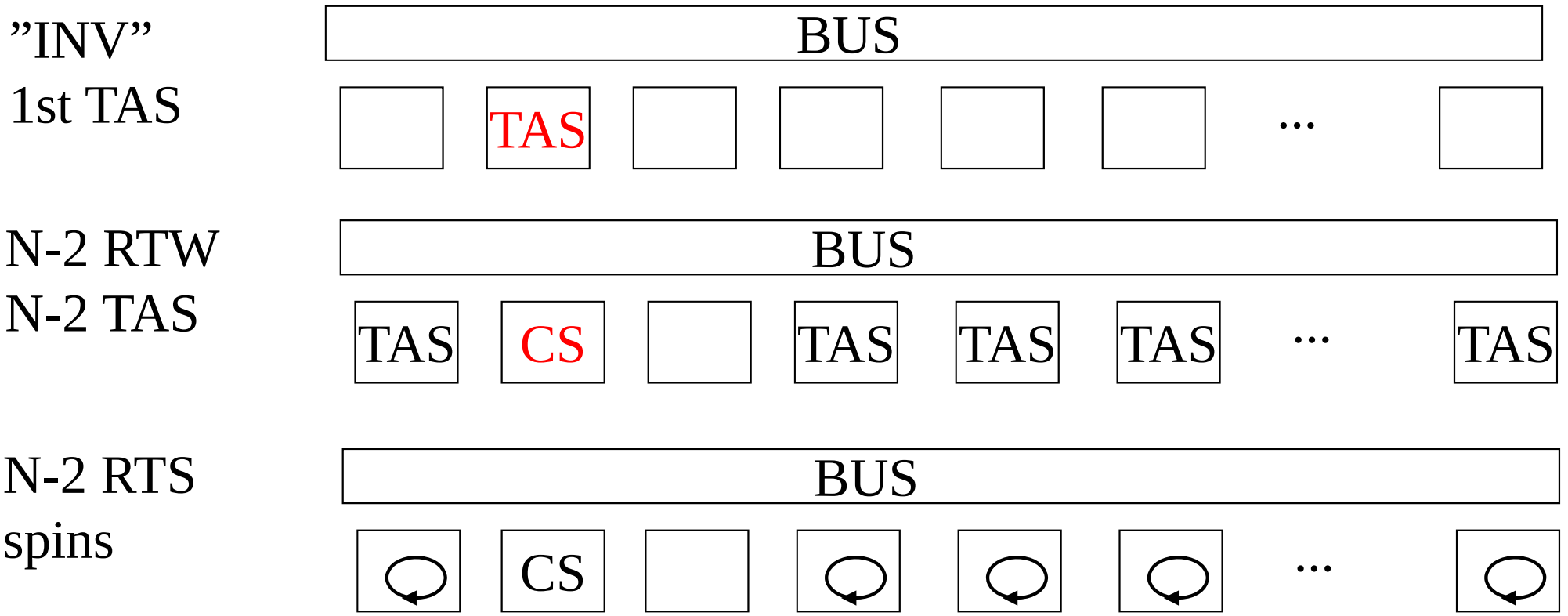
```
proc lock(lock_variable) {
    while true {
      if (TAS[lock_variable] ==0)  break;  /* Pound on the lock once, done if TAS==0 */
      while(lock_variable != 0) {}          /* spin locally in your cache until ”0” observed*
  }
}

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

AVDARK 2013

It could still get messy!
(Assuming round-robin snooping bus)



...messy (part 2)



Problem 1: Snoop activity slows down the hand-over

Problem 2: The new winner in CS is slowed down by more traffic

How many bus transaction between exit CS until a new CS entry (incl. lock/unlock)

- ❑ 1 "INV"
- ❑ (N-1) RTS + (N – 2) RTW
- ❑ "INV" + (N-1) RTS + "INV"

How many total bus transaction are queued up when CS changes "owner"

- ❑ 2 "INV" + (N-1) RTS + (N-2) RTW + (N-2) RTS
- ❑ (N-1) RTS + (N – 1) RTW
- ❑ "INV" + (N-1) RTS + "INV"

Ticket-based queue locks: "ticket"

START: R1 = M(); R2= R1 +1;
Compare&wap if R1==M() swap R2 and M();
ELSE GOTO START:

```
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!

How many bus transaction between exit CS until a new CS entry (incl. lock/unlock)

- 1 "INV" + {1..N-2} RTS
- "INV"
- "INV" + (N-1) RTS + "INV"

How many bus transaction total when CS changes "owner"

- 2 "INV" + (N-1) RTS + (N-2) RTW + (N-2) RTS
- 1 "INV" + (N-1) RTS
- "INV" + (N-1) RTS + (N-1) RTW

More scalable locks

Erik Hagersten
Uppsala University

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!

AVDARK 2013

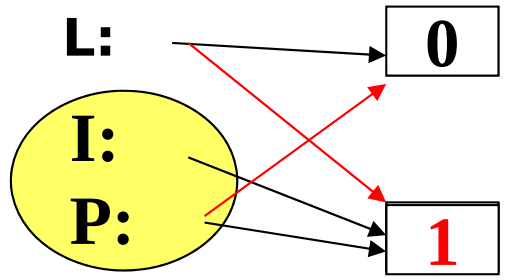
UPPSALA UNIVERSITET

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 Christmas  shopping */
        }
}

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

Even less traffic at lock handover!

Queue-based lock: CLH-lock



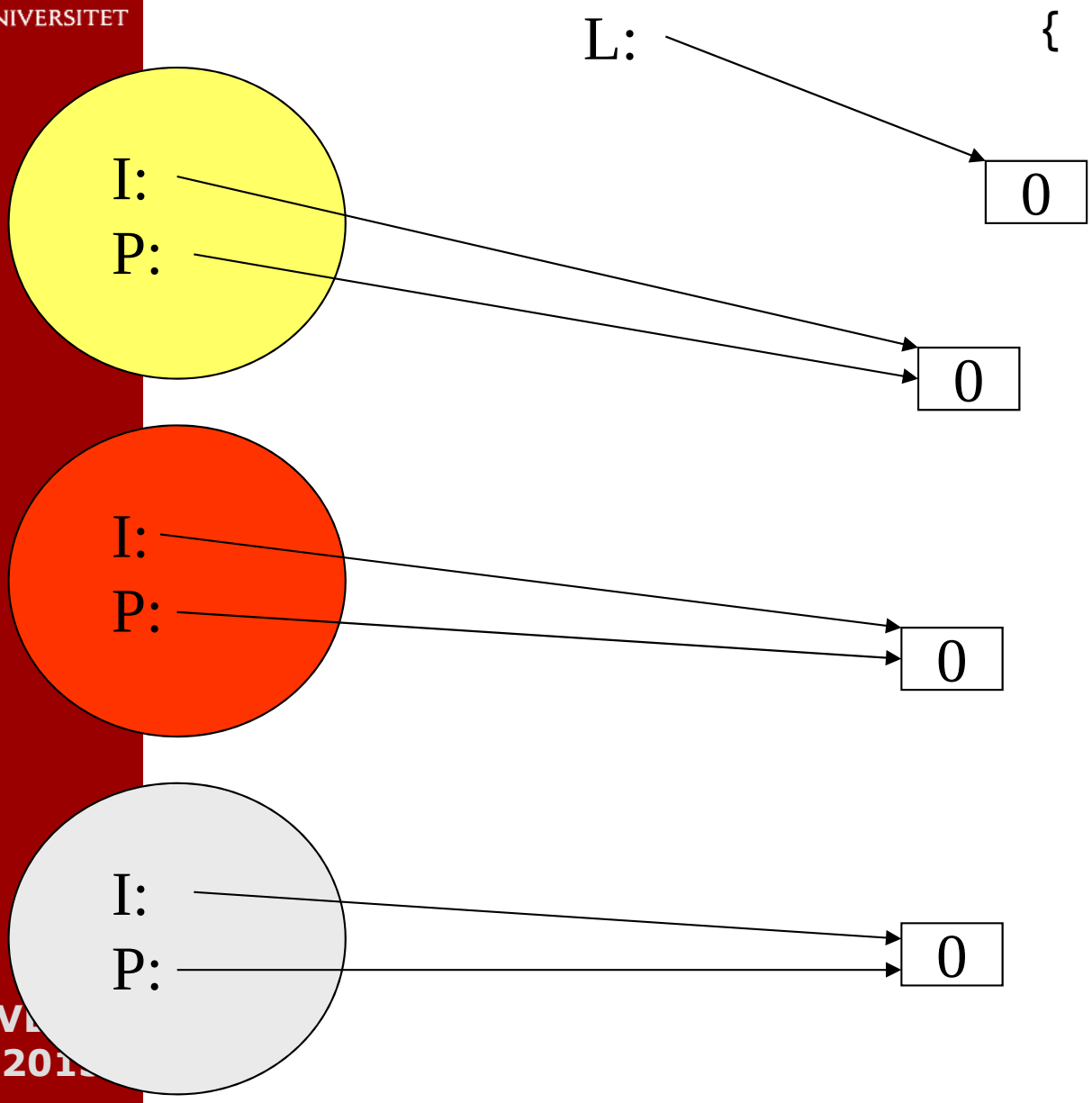
"Initially, each thread starts with one "global cell", pointed to by private *I and *P
Another global cell is pointed to by global *L "lock variable"
Line of thought for implementing **lock(L):**

1) Initialize the *I flag to busy (= "1")
2) Atomically, make *L point to "our" cell and make "our" *P point where *L's cell
3) Wait until *P points to a "0" /* here, the lock is already free */

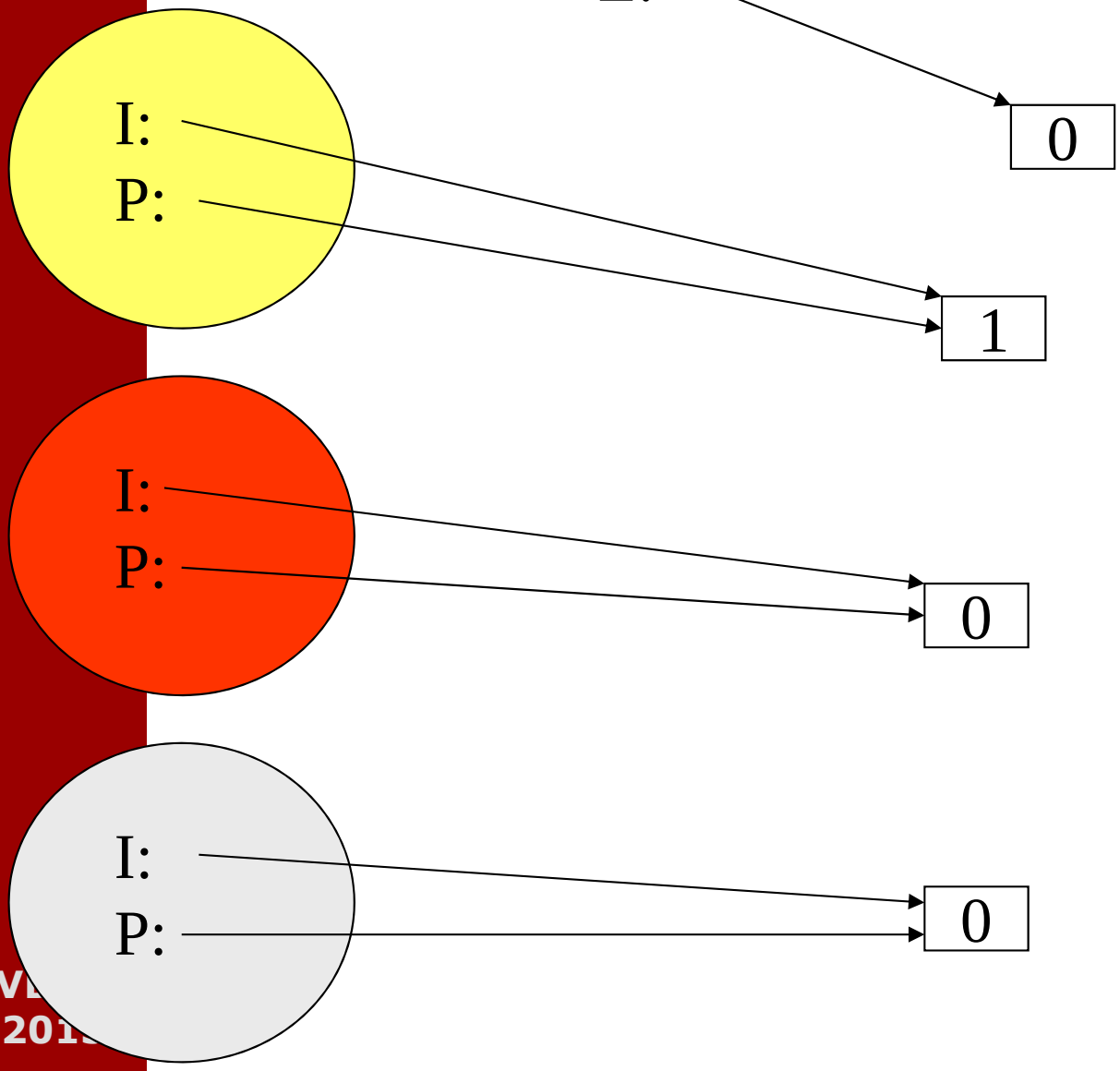
```
DEF lock(int **L, **I, **P)
{   **I =1;                  /*initialized "our" cell as "busy"*/
    atomic_swap {*P =*L; *L=*P}
                 /* P now stores a pointer to the cell L pointed to*/
                 /* L now stores a pointer to our cell */
    while (**P != 0){} }/* keep spinning until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

```
DEF unlock(int **I, **P)
{    **I = 0;      /* release the lock */
     *I =*P; }     /* next time *I to reuse the previous guy's cell*/
```

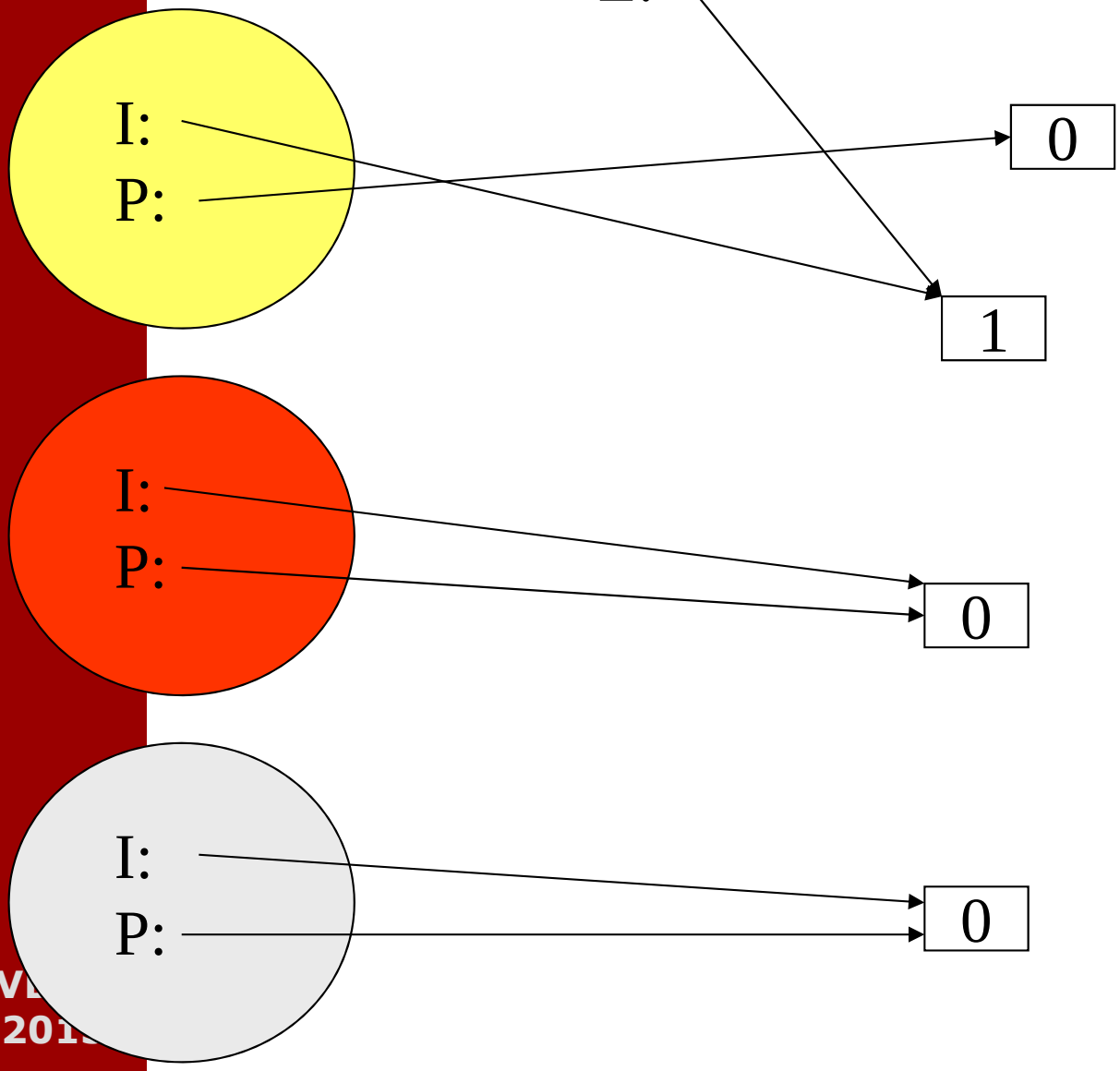
CLH lock



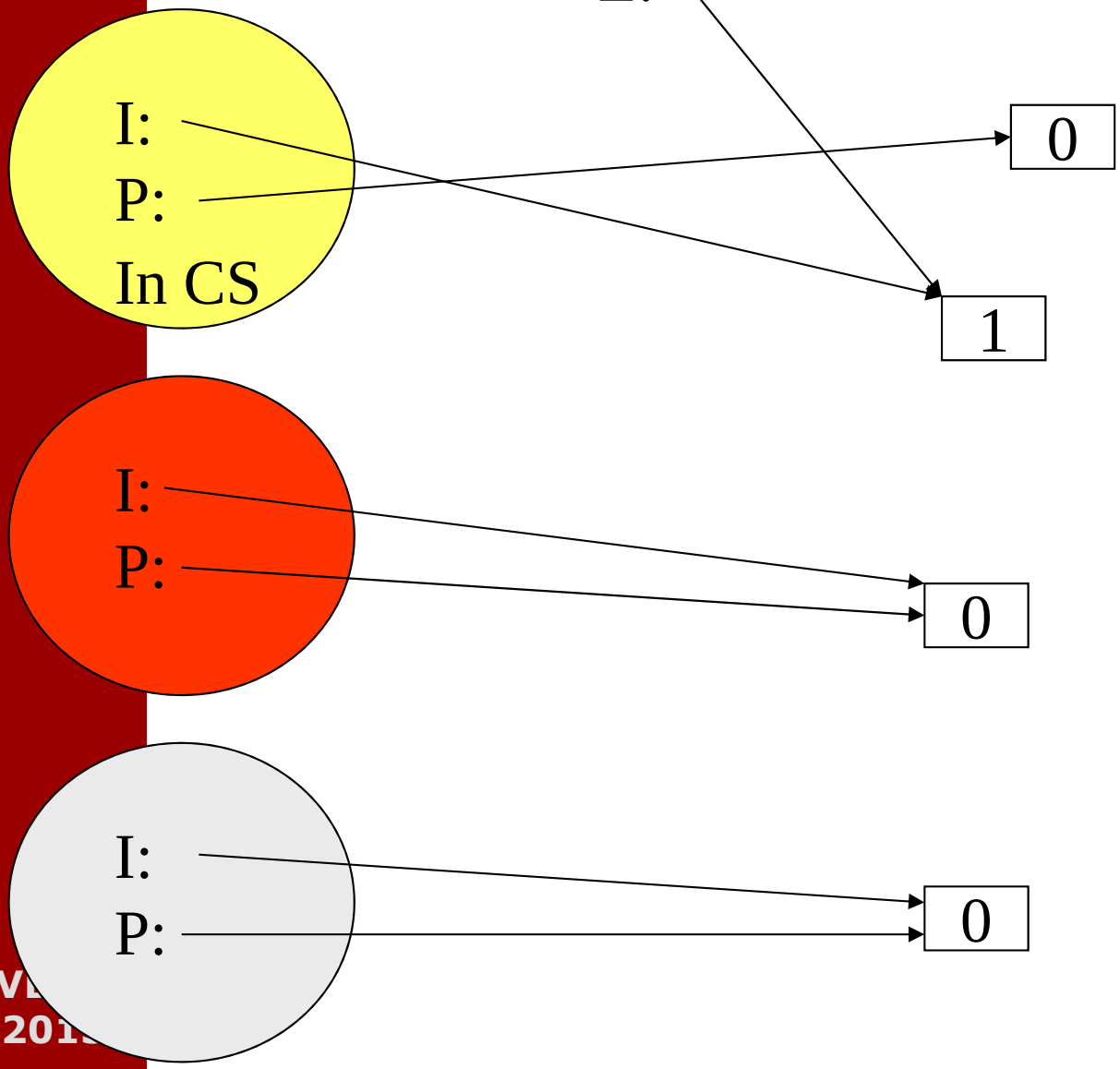

```
proc lock(int **L, **I, **P)
{       **I =1 /* init to "busy"*/
        atomic_swap {*P =*L; *L=*P}
/* *L now points to our I* */
        while (**P != 0){}  }
/* spin unit prev is done */
```



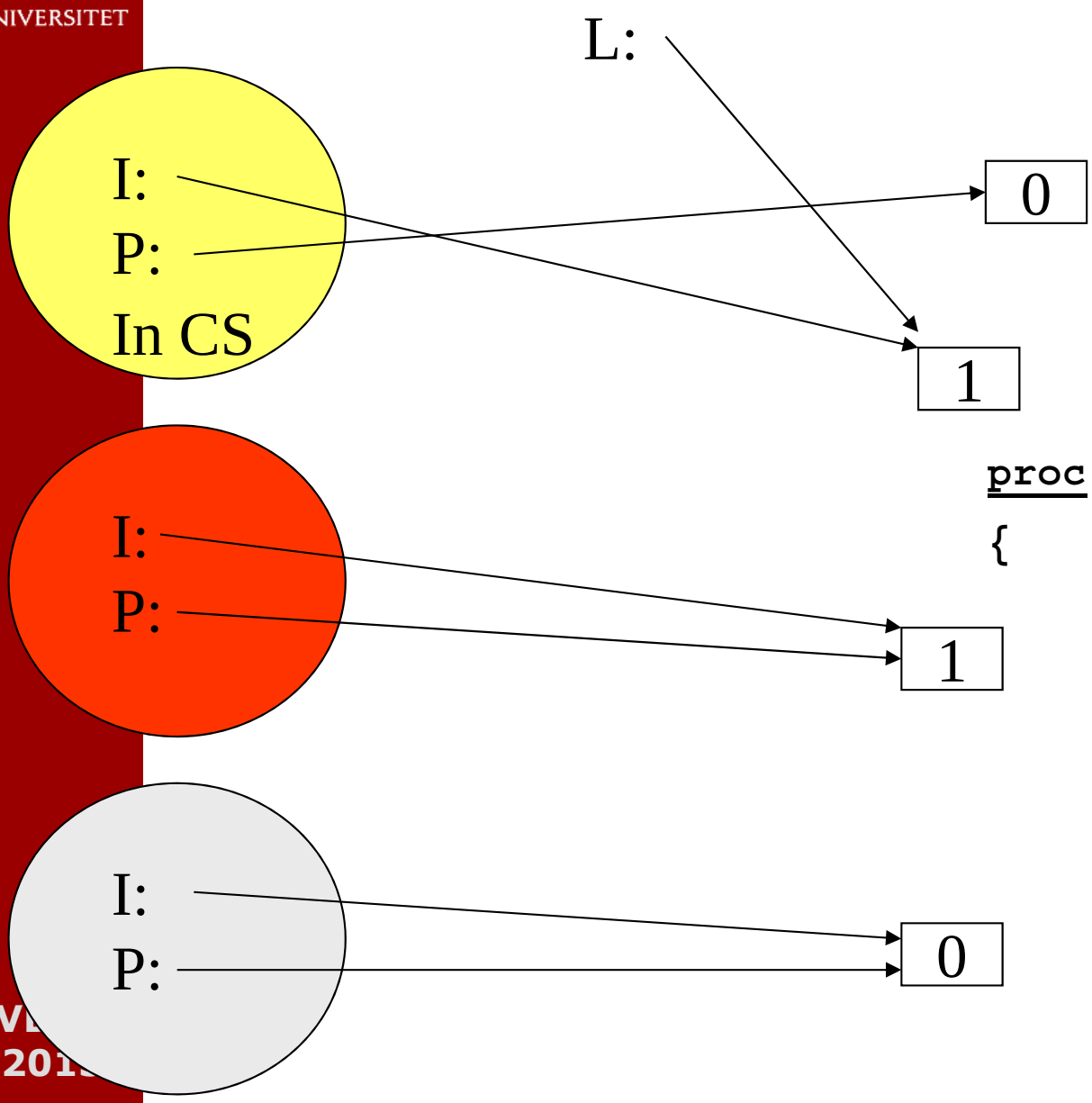

```
proc lock(int **L, **I, **P)
{       **I =1 /* init to "busy"*/
        atomic_swap {*P =*L; *L=*P}
/* *L now point to our I* */
        while (**P != 0){} }
/* spin unit prev is done */
```



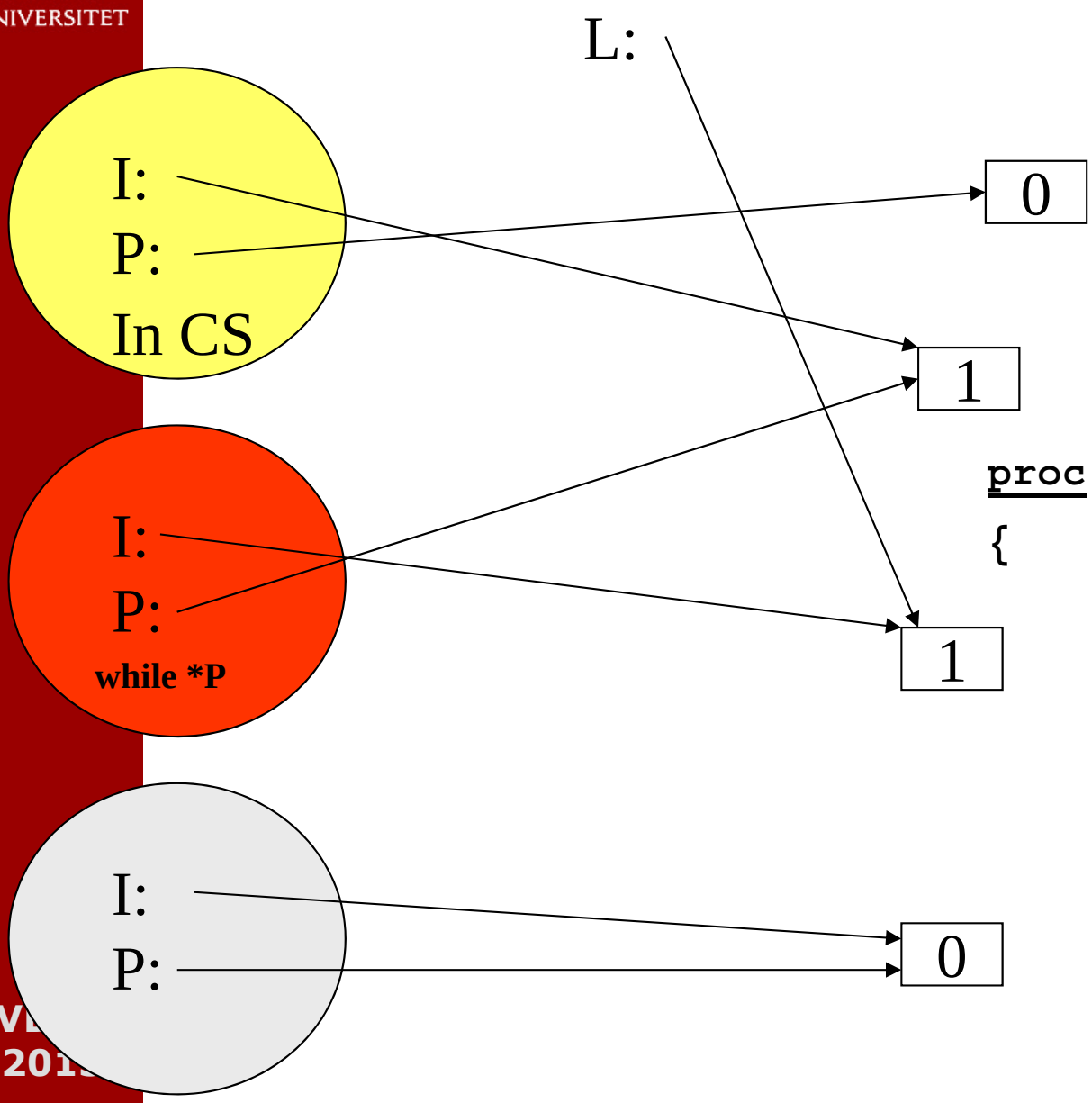

```
proc lock(int **L, **I, **P)
{       **I =1 /* init to "busy"*/
        atomic_swap {*P =*L; *L=*P}
/* *L now point to our I* */
        while (**P != 0){} };
/* spin unit prev is done */
```



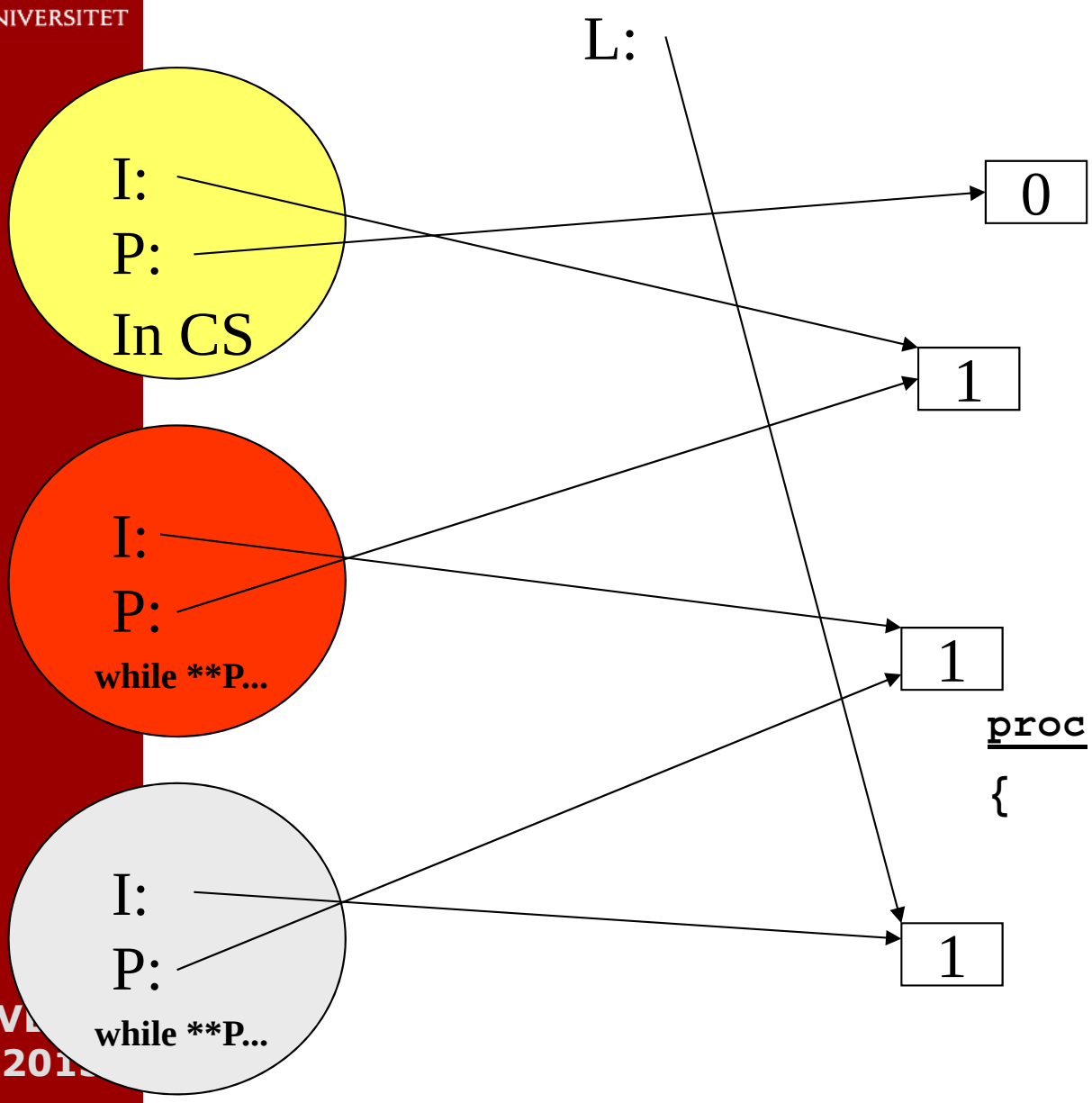

```
proc lock(int **L, **I, **P)
{       **I =1 /* init to "busy"*/
        atomic_swap {*P =*L; *L=*P}
/* *L now point to our I* */
        while (**P != 0){} };
/* spin unit prev is done */
```



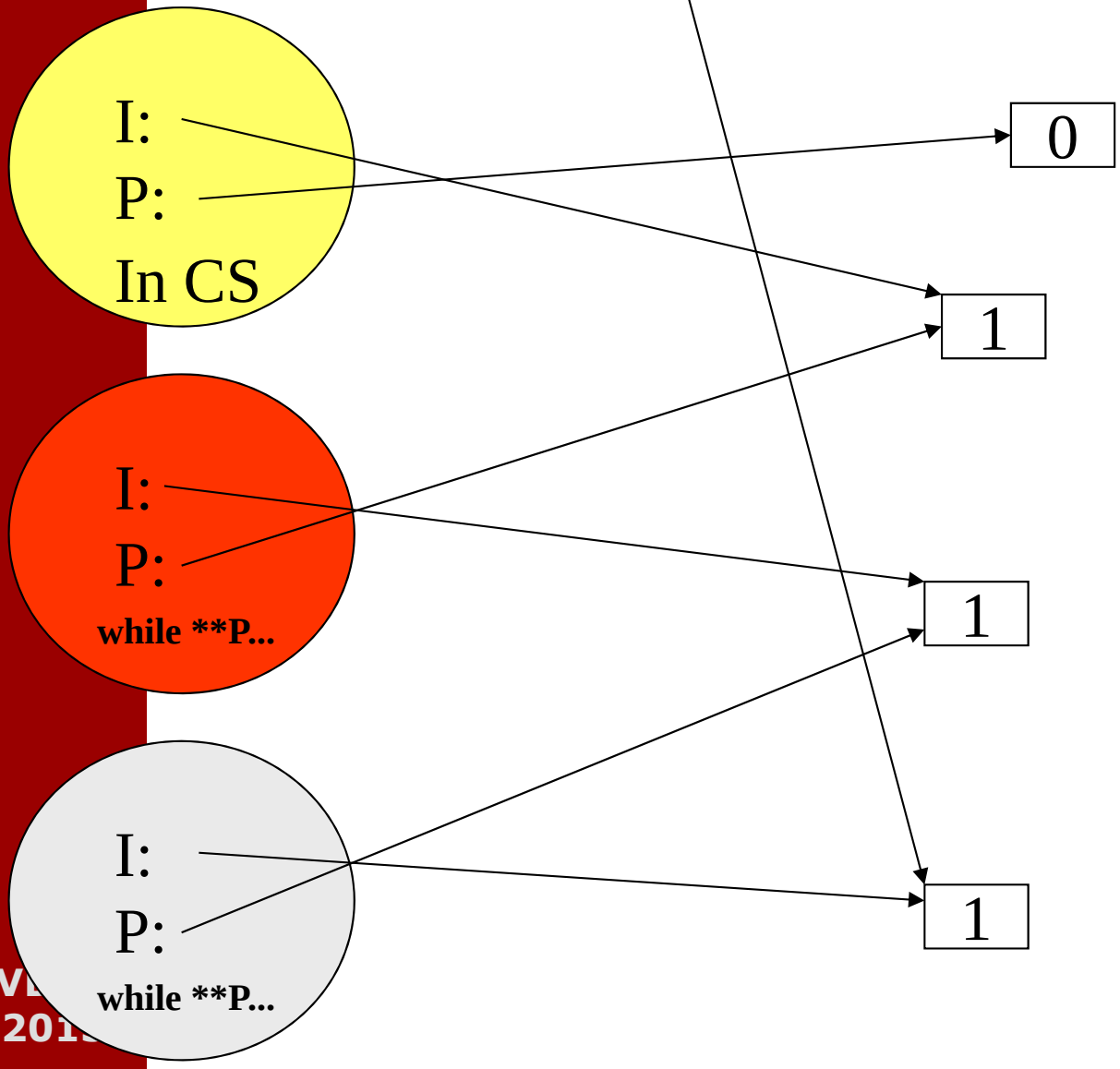

```
proc lock(int **L, **I, **P)
{       **I =1 /* init to "busy"*/
        atomic_swap {*P =*L; *L=*P}
       /* *L now point to our I* */
        while (**P != 0){} };
        /* spin unit prev is done */
```



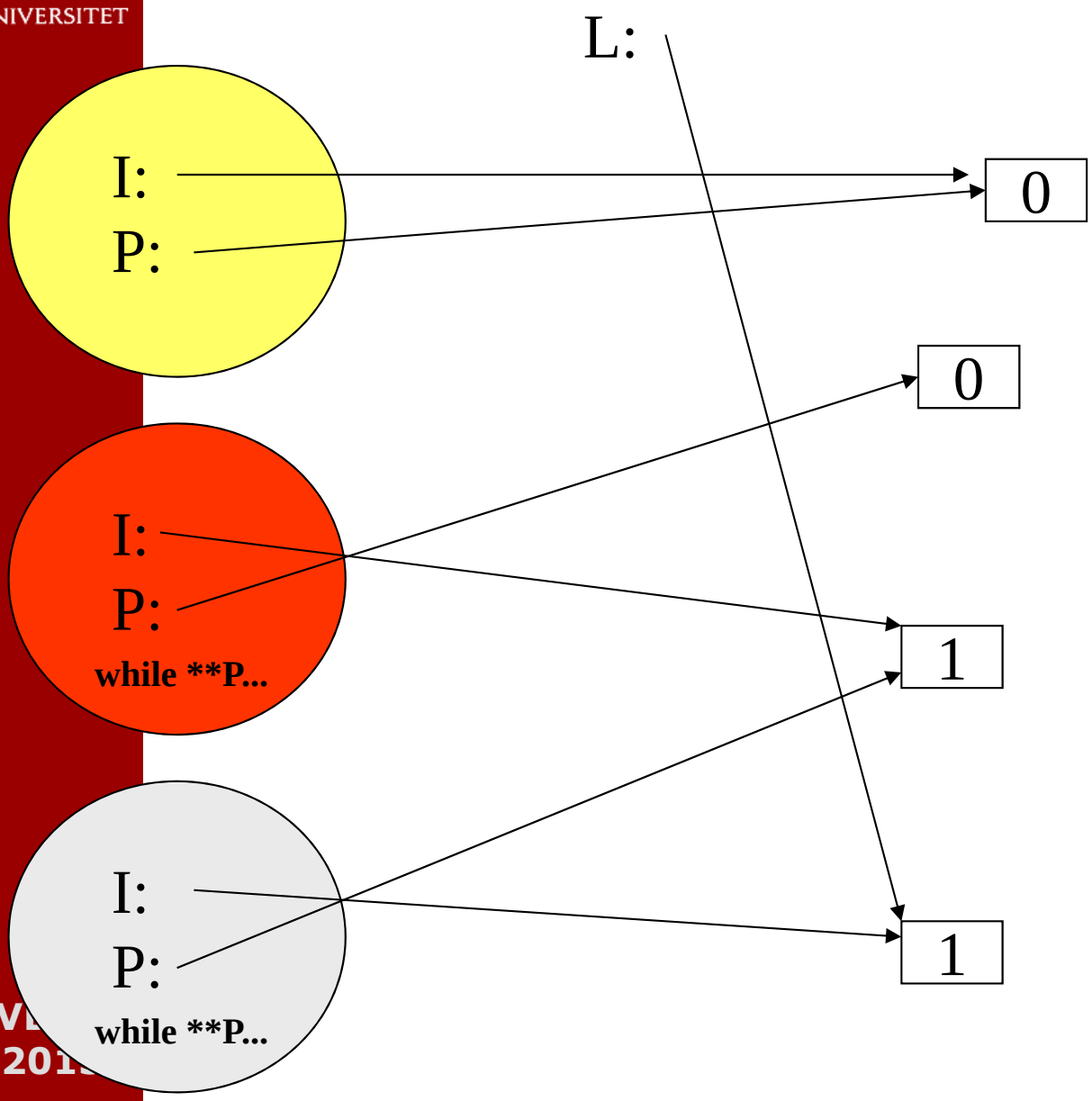

```
proc lock(int **L, **I, **P)
{       **I =1 /* init to "busy"*/
        atomic_swap {*P=*L; *L=*P;}
        /* *L now point to our I* */
        while (**P != 0){} };
        /* spin unit prev is done */
```



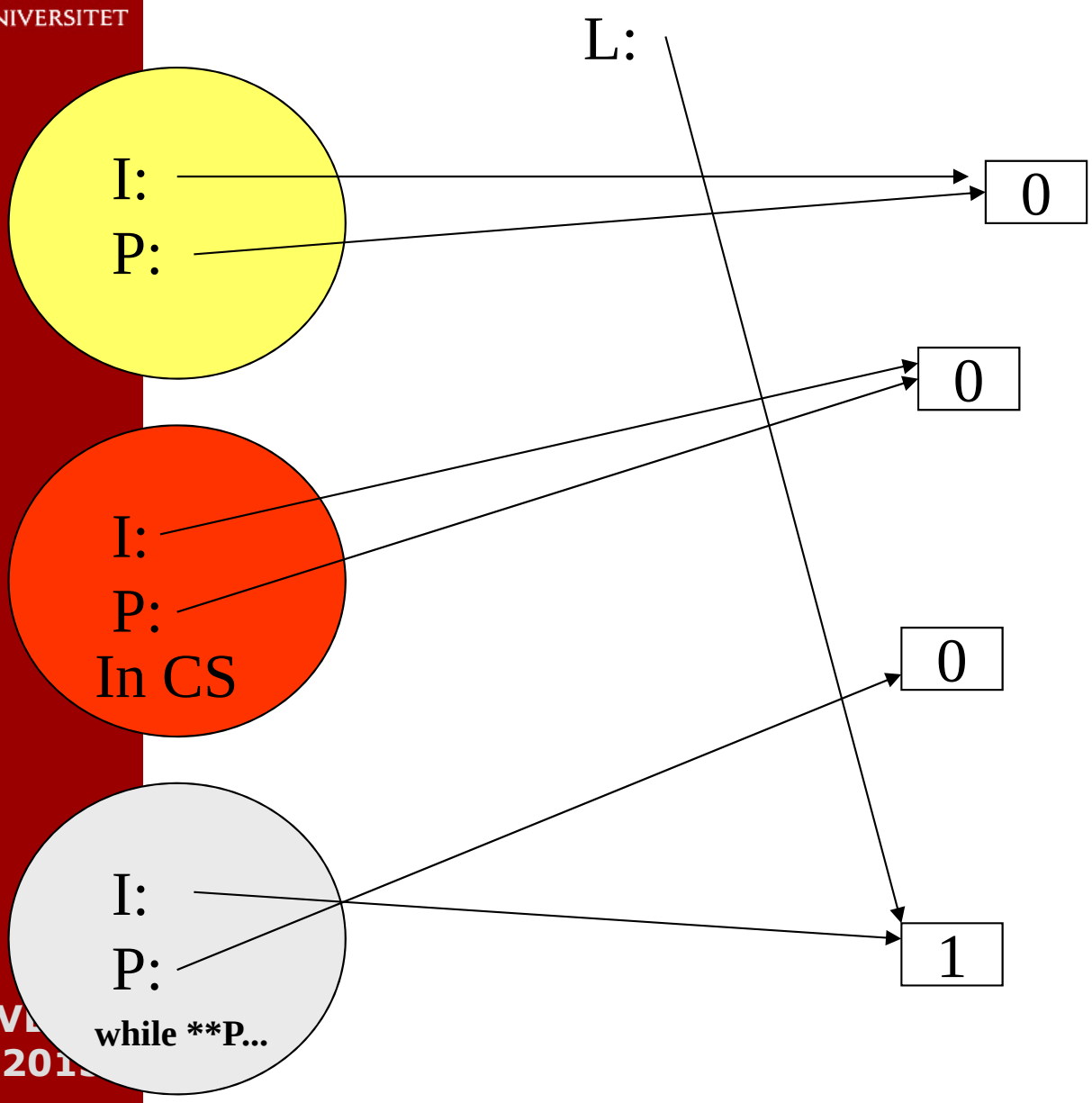

```
proc lock(int **L, **I, **P)
{       **I =1 /* init to "busy"*/
        atomic_swap {*P =*L; *L=*P;}
       /* *L now point to our I* */
        while (**P != 0){} };
        /* spin unit prev is done */
```




```
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*/
```

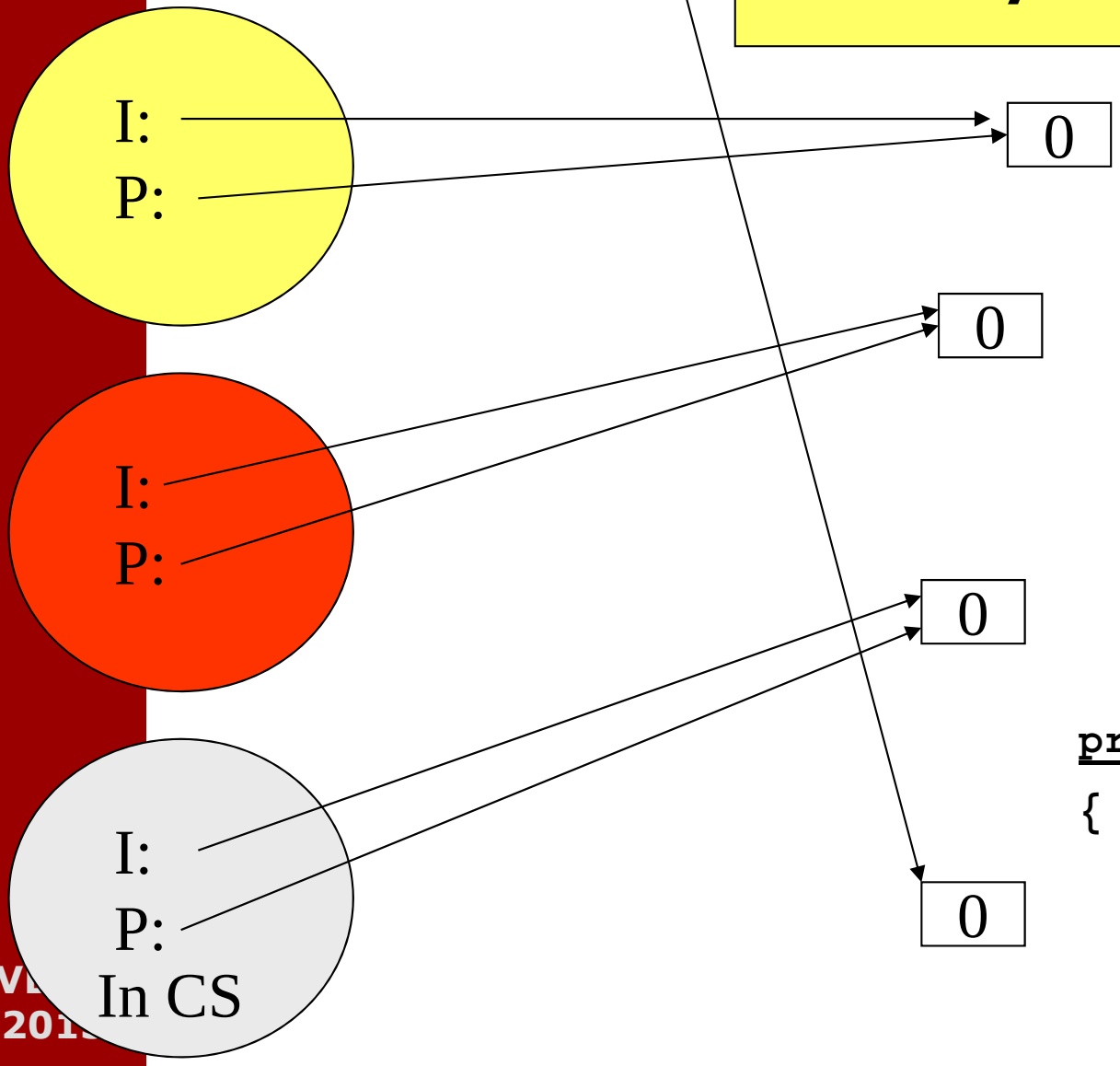



```
proc unlock(int **I, **P)
{       **I = 0;
        *I = *P; }
```

How many bus transaction between exit CS until a new CS entry (incl. lock/unlock)

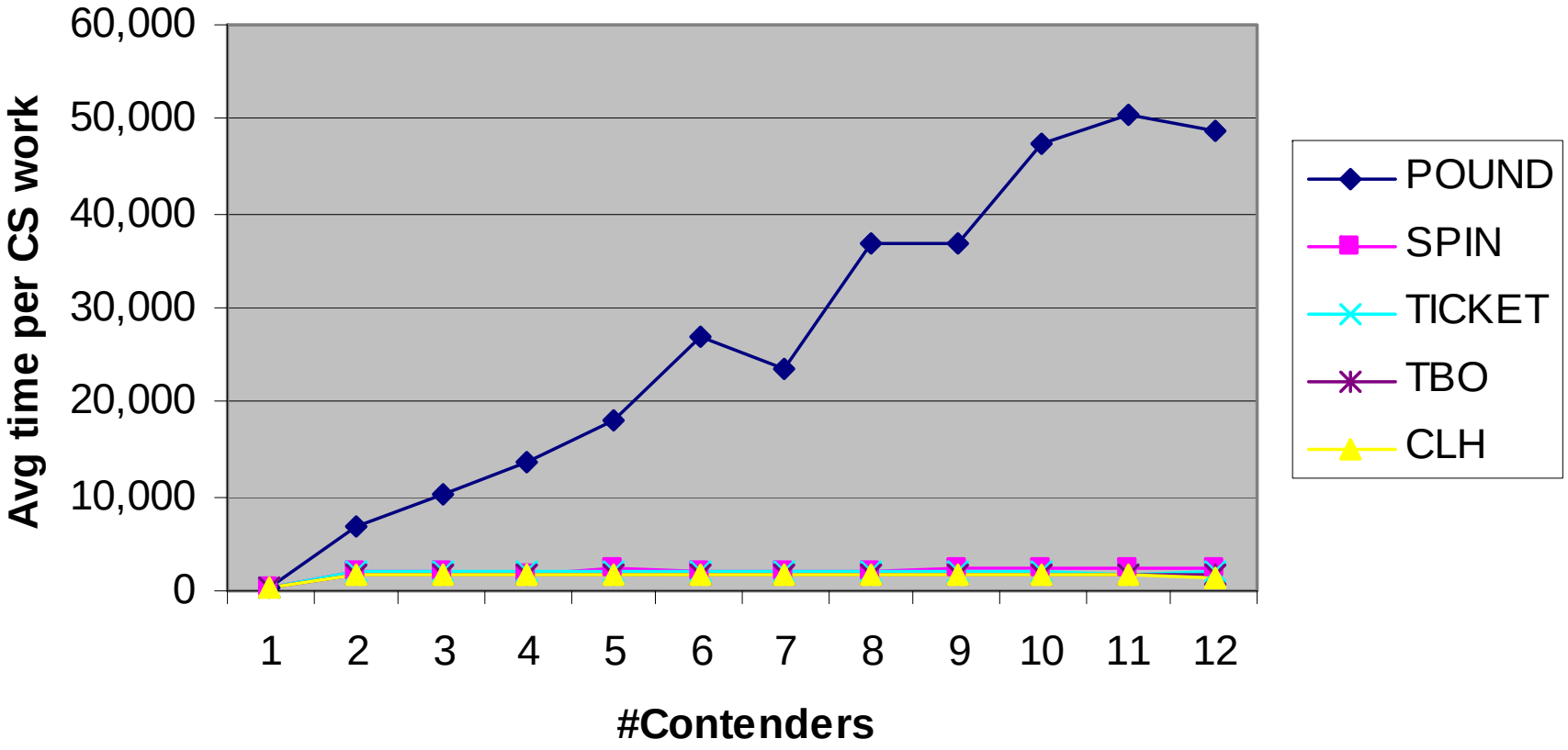
- 1 ”INV” + {1..N-2} RTS
- 1 “INV” + 1 RTS
- “INV” + (N-1) RTS + “INV”

mizes traffic at lock handover!
ay be too fair for NUMAs ….

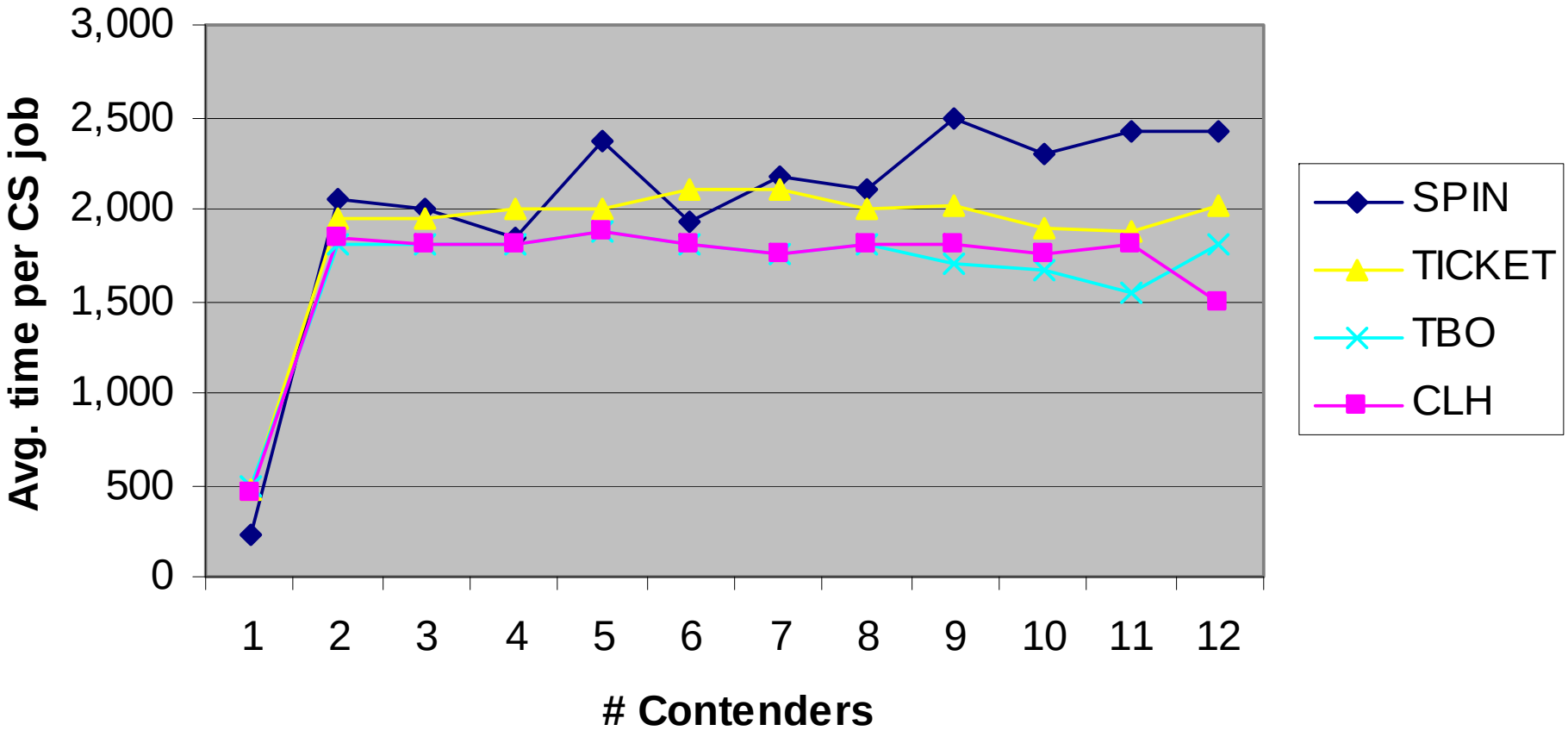



```
proc unlock(int **I, **P)
{       **I = 0;
        *I = *P; }
```

E6000 locks 12 CPUs

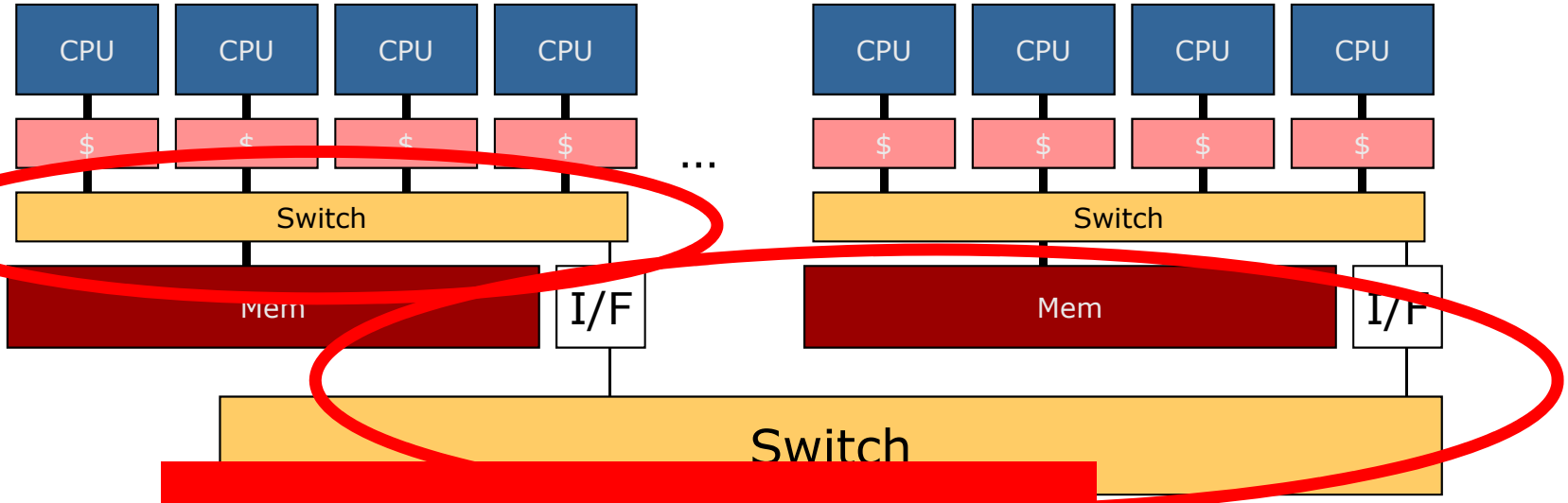


E6000 locks (exluding POUND)



NUMA:

NUCA: Non-uniform Comm Arch.



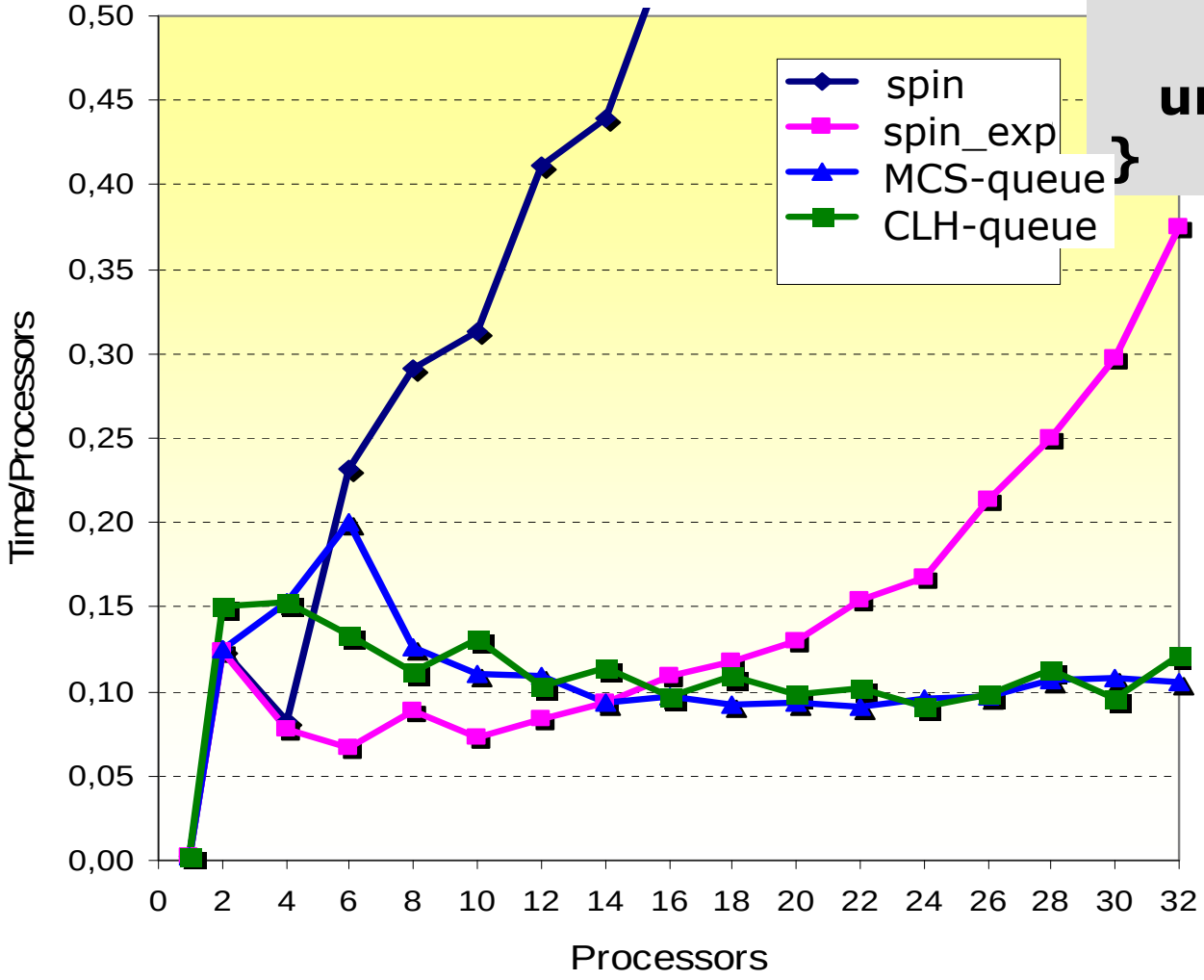
Snoop

Directory-latency = 6x snoop
i.e., roughly CMP NUCA-ness

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

Benchmark:

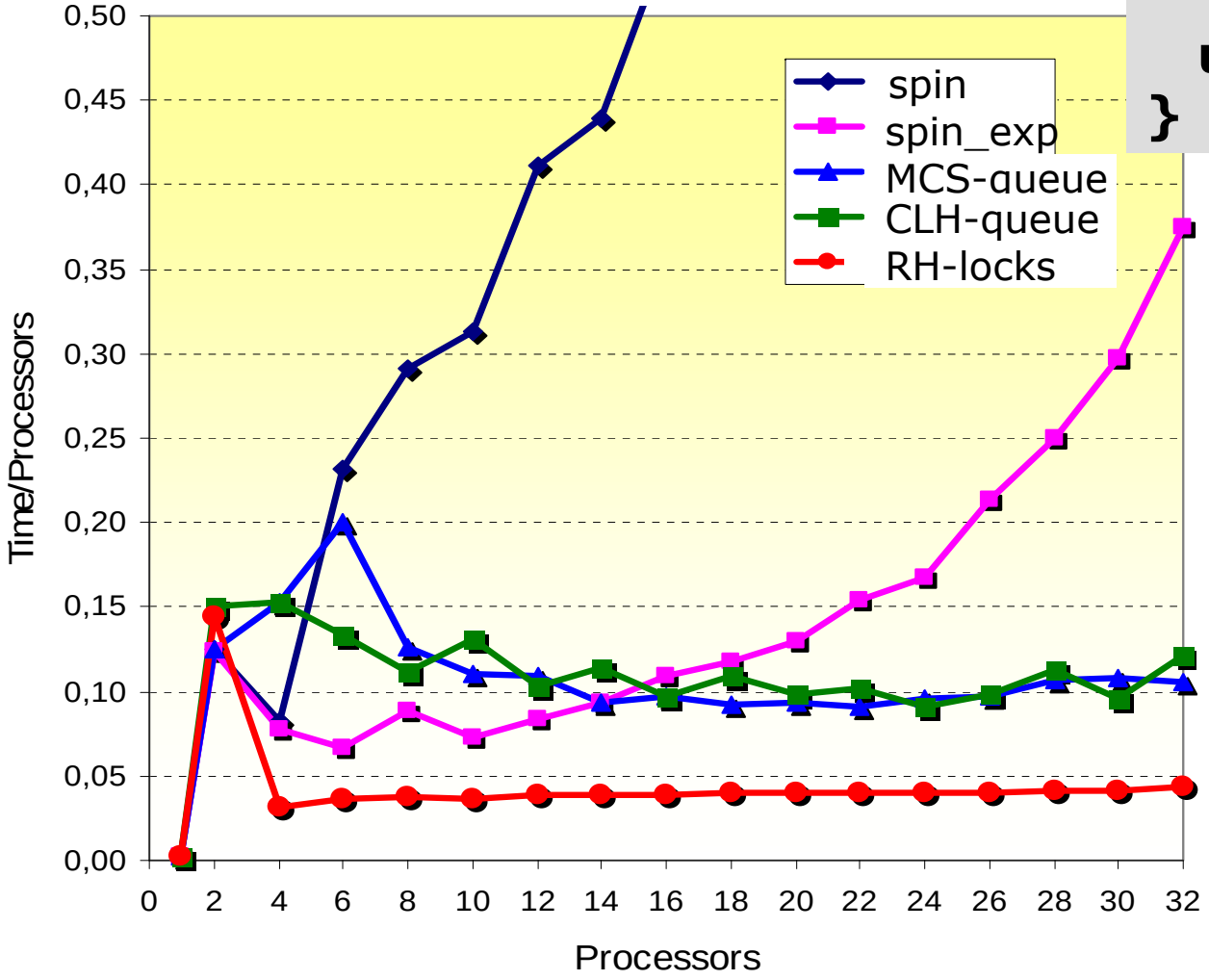
```
for i = 1 to 10000 {
   lock(AL)
      A:= A + 1;
   unlock(AL)
}
```



Introducing RH locks

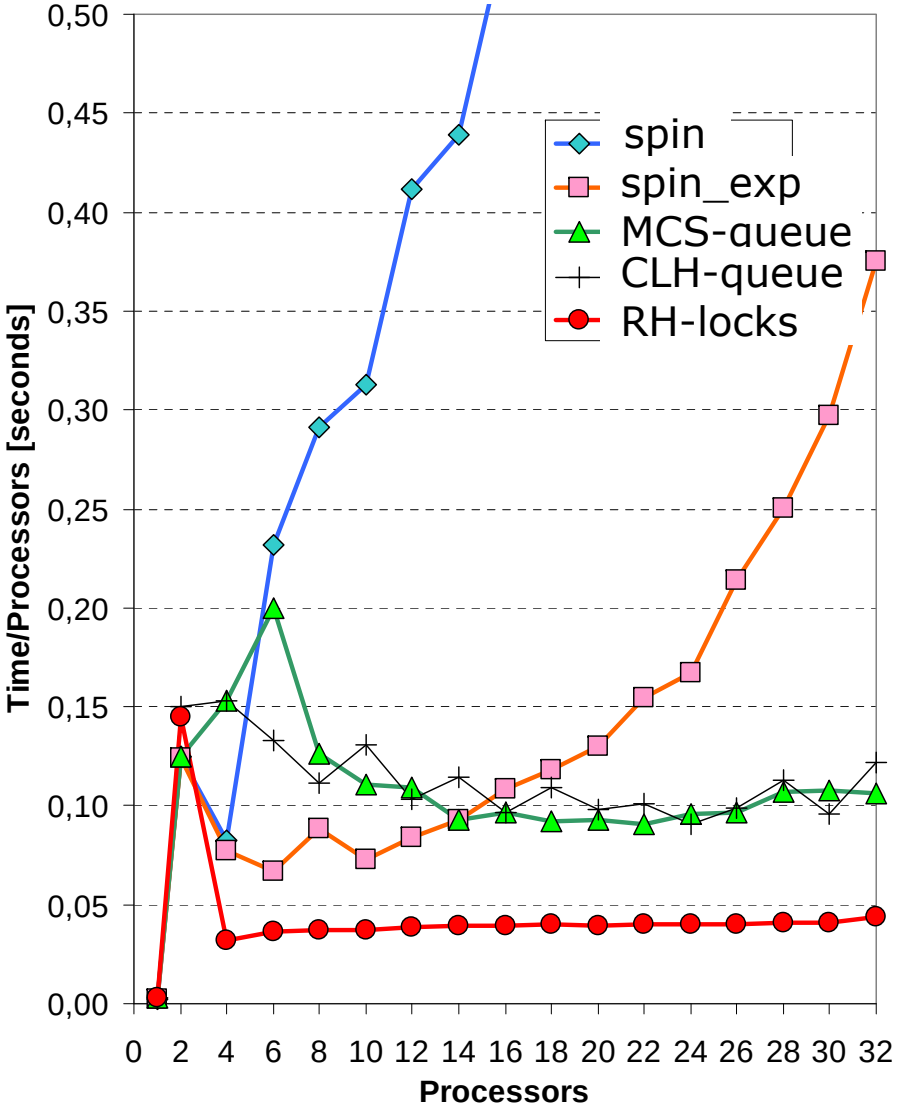
Benchmark:

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

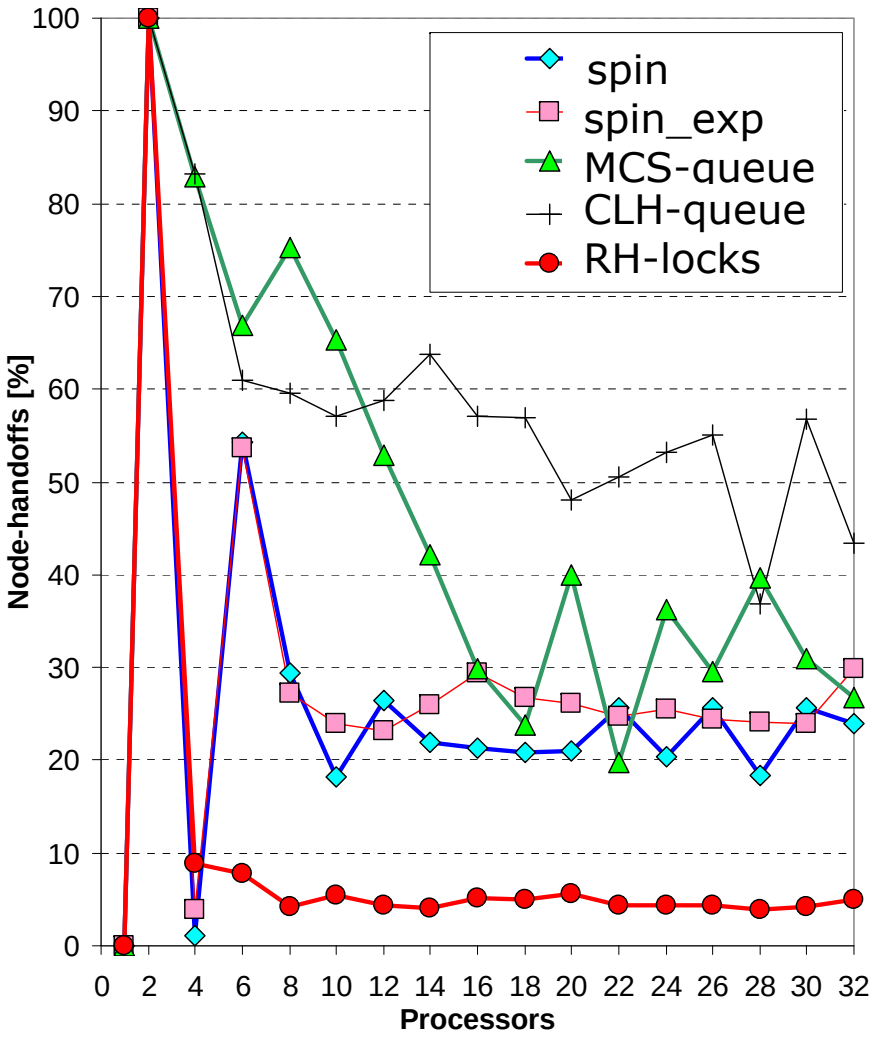


RH locks: encourages unfairness

Time per lock handover



Node migration (%)

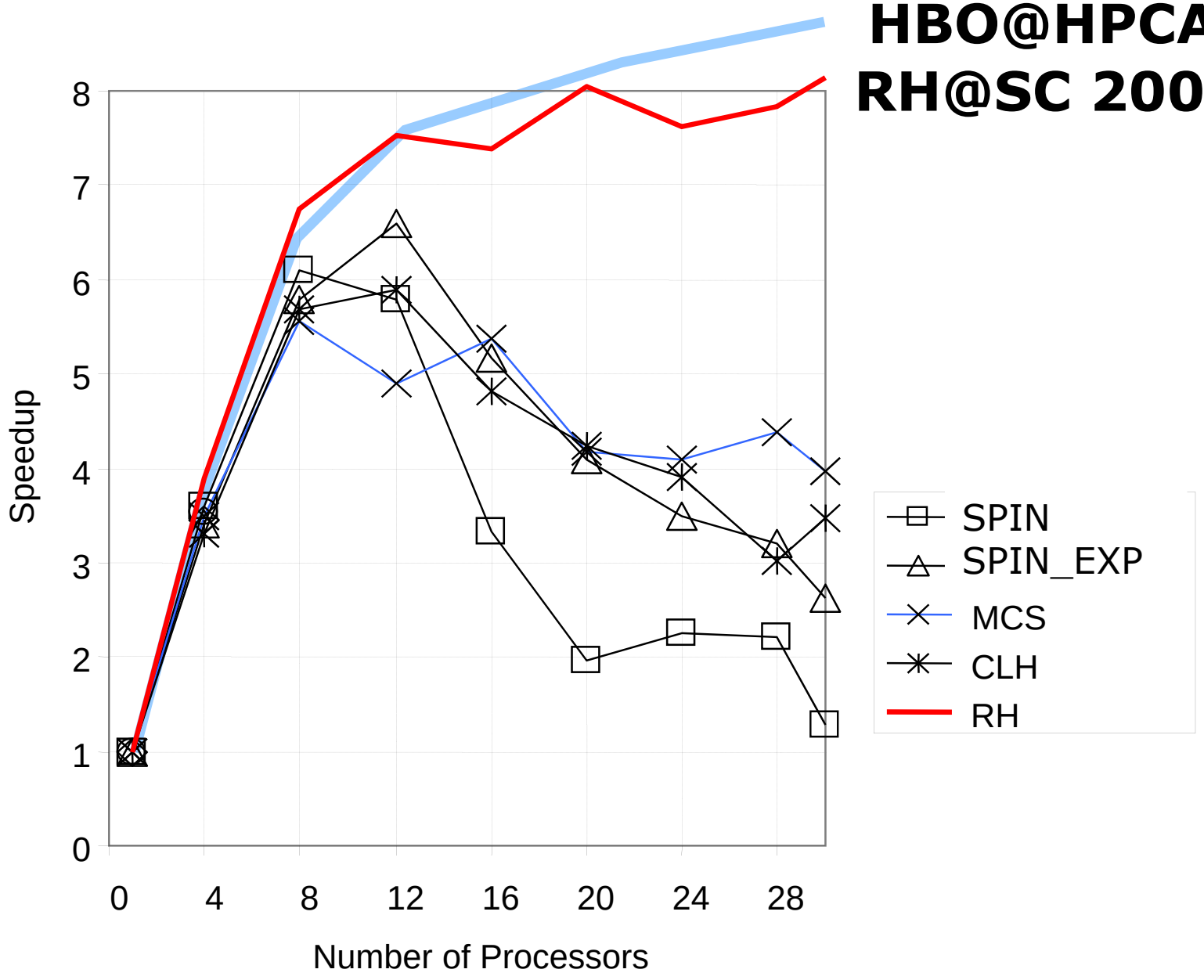


UPPSALA UNIVERSITET

Ex: Splash Raytrace Application Speedup

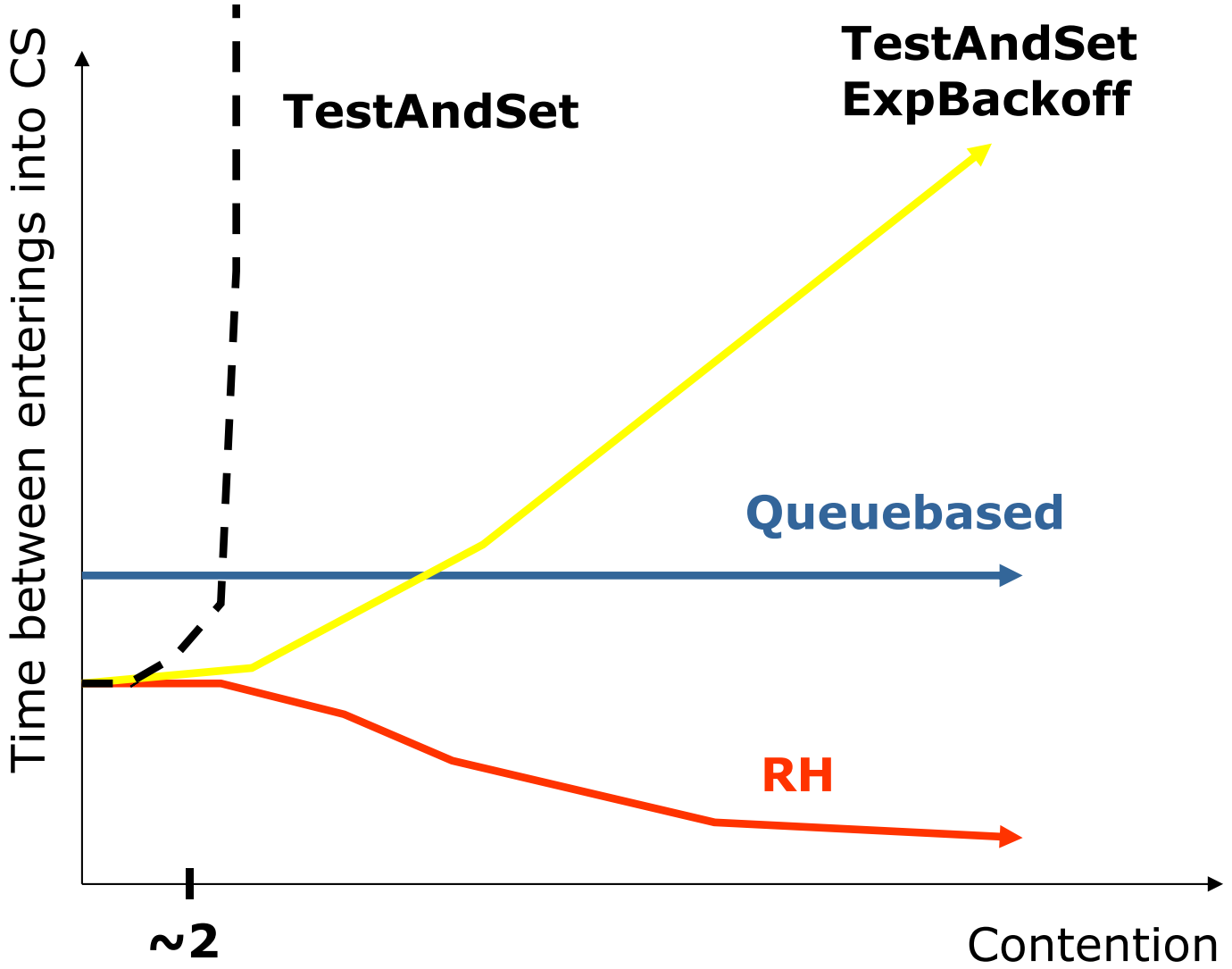
HBO@HPCA 2003

RH@SC 2002



AVDARK 2013

Performance under contention

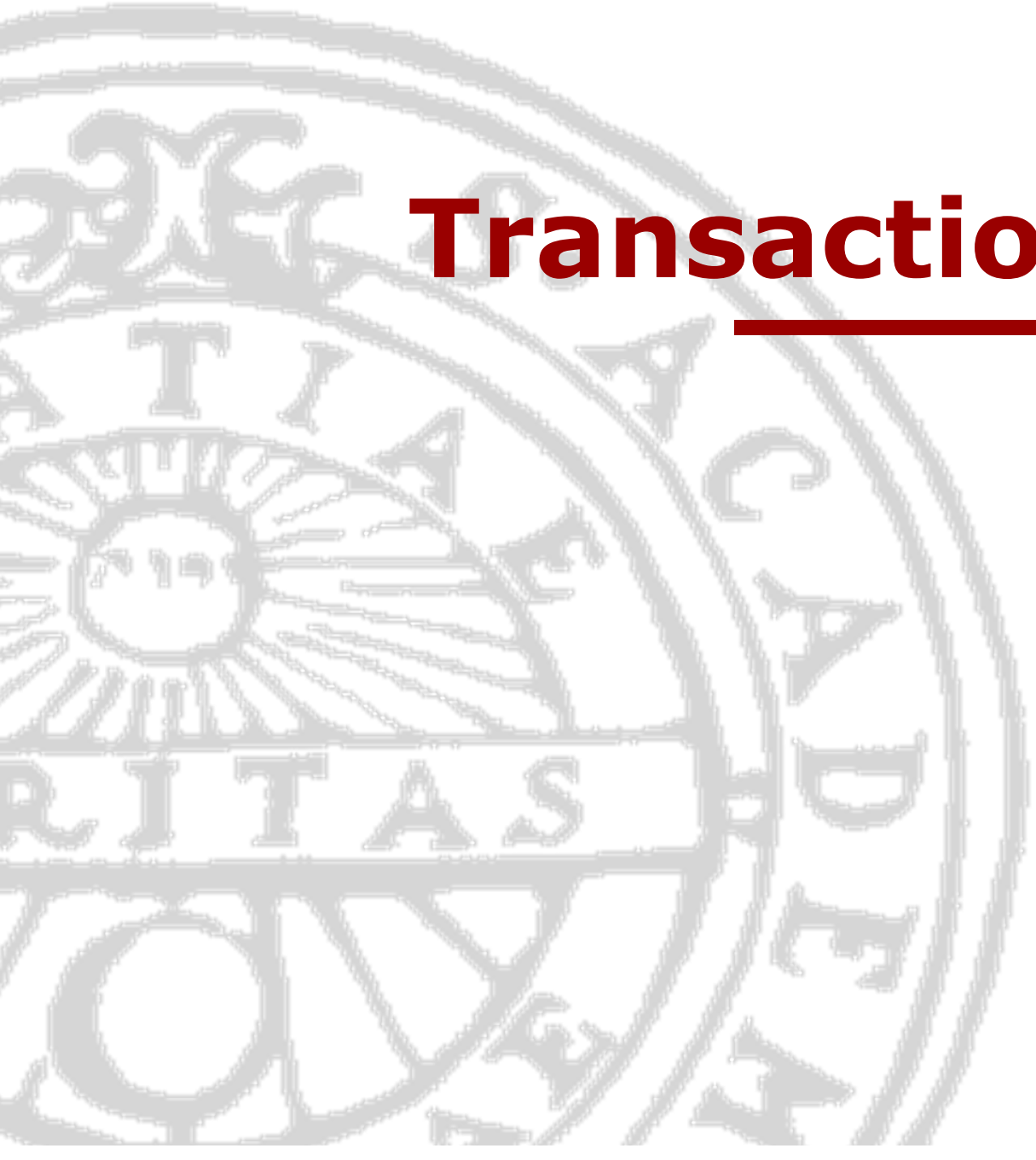


UPPSALA UNIVERSITET

More recent:

- PhD student David Klaftenegger of the compiler/language group is working on a much more scala synchronization primitive
- Stay tuned…

AVDARK 2013



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();
```

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

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…