Lecture 11:

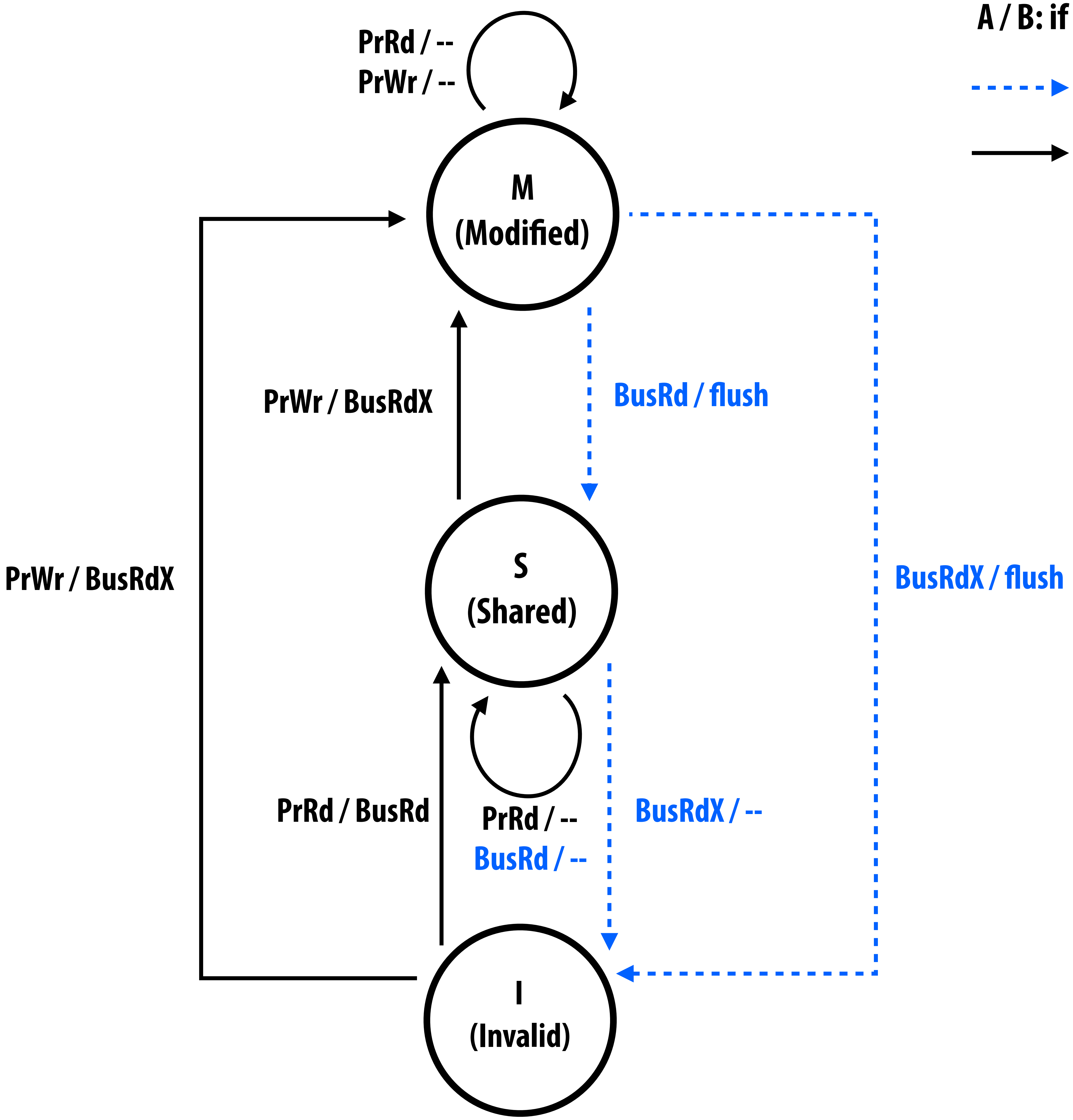
A Basic Snooping-Based Multi-Processor Implementation

Parallel Computer Architecture and Programming

CMU / 清华大学, Summer 2017

Tsinghua has its own ice cream! Wow!

Review: MSI state transition diagram



A / B: if action A is observed by cache controller, action B is taken

- - - - -▶ Remote processor (coherence) initiated transaction

——▶ Local processor initiated transaction

flush = flush dirty line to memory

Review:

P0: LD X

P0: LD X

P0: ST X ← 1

P0: ST X ← 2

P1: ST X ← 3

P1: LD X

P0: LD X

P0: ST X ← 4

P1: LD X

P0: LD Y

P0: ST Y ← 1

P1: ST Y ← 2

Consider this sequence of loads and stores to addresses X and Y by processors P0 and P1

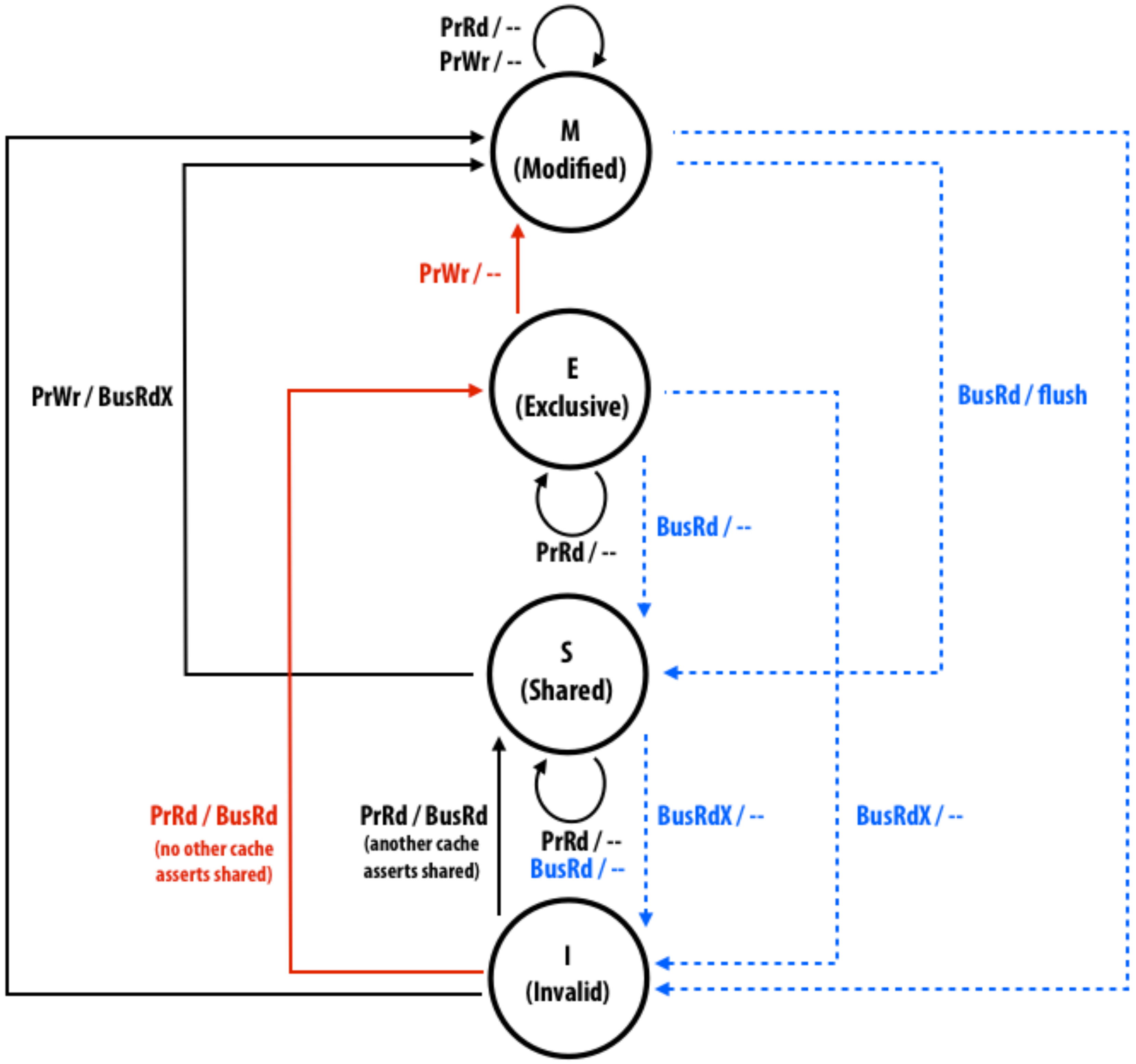
Assume that X and Y contain value 0 at start of execution.

Today: implementing cache coherence

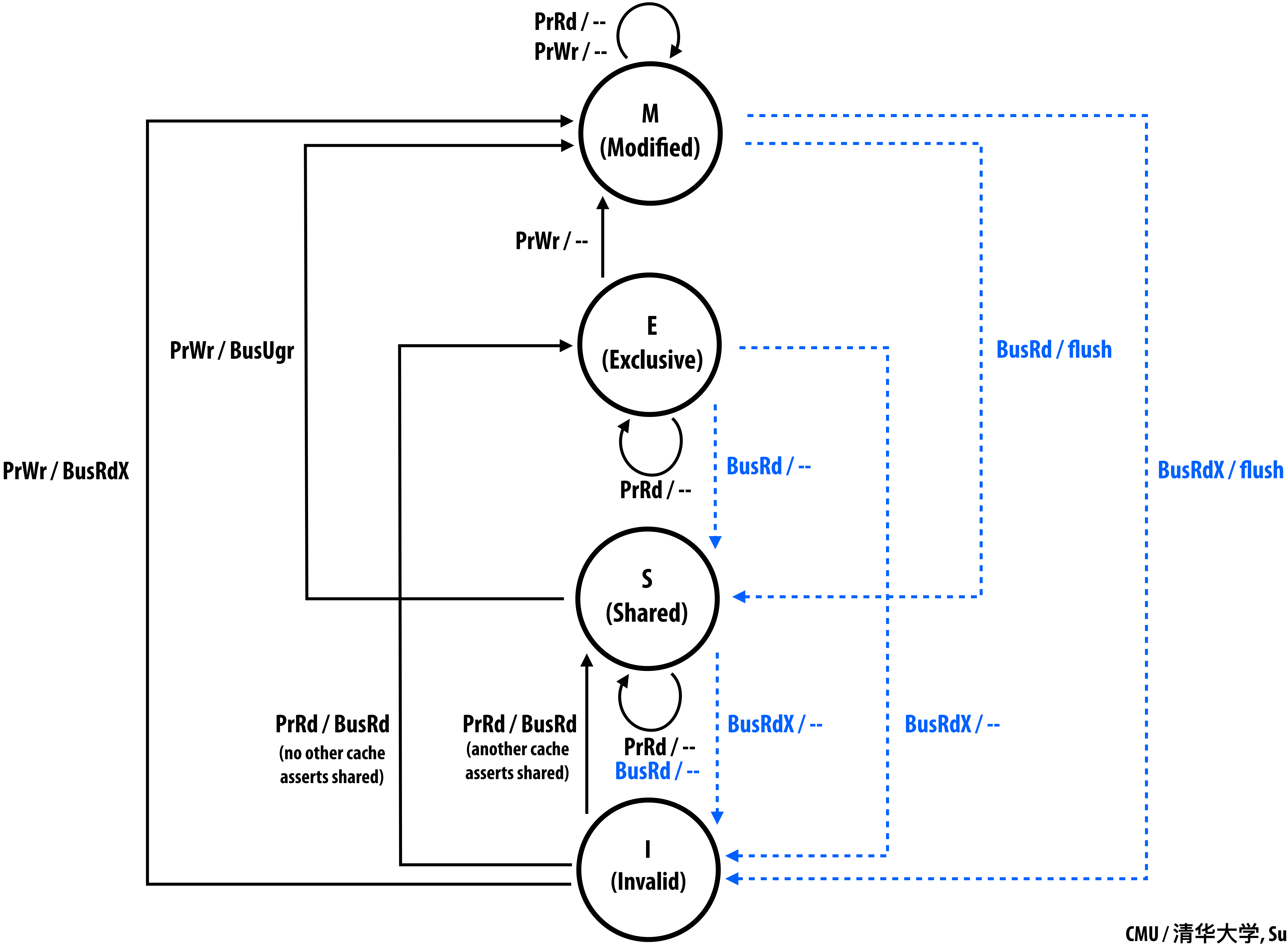
- Wait… wasn't this the topic of last class?
- In the previous lectures we talked about cache coherence protocols
  - But our discussion was very abstract
    - We described what messages/transactions needed to be sent
    - We assumed messages/transactions were atomic

Today we will talk about implementing an invalidation-based protocol

Today's point: in a real machine… efficiently ensuring coherence is complex



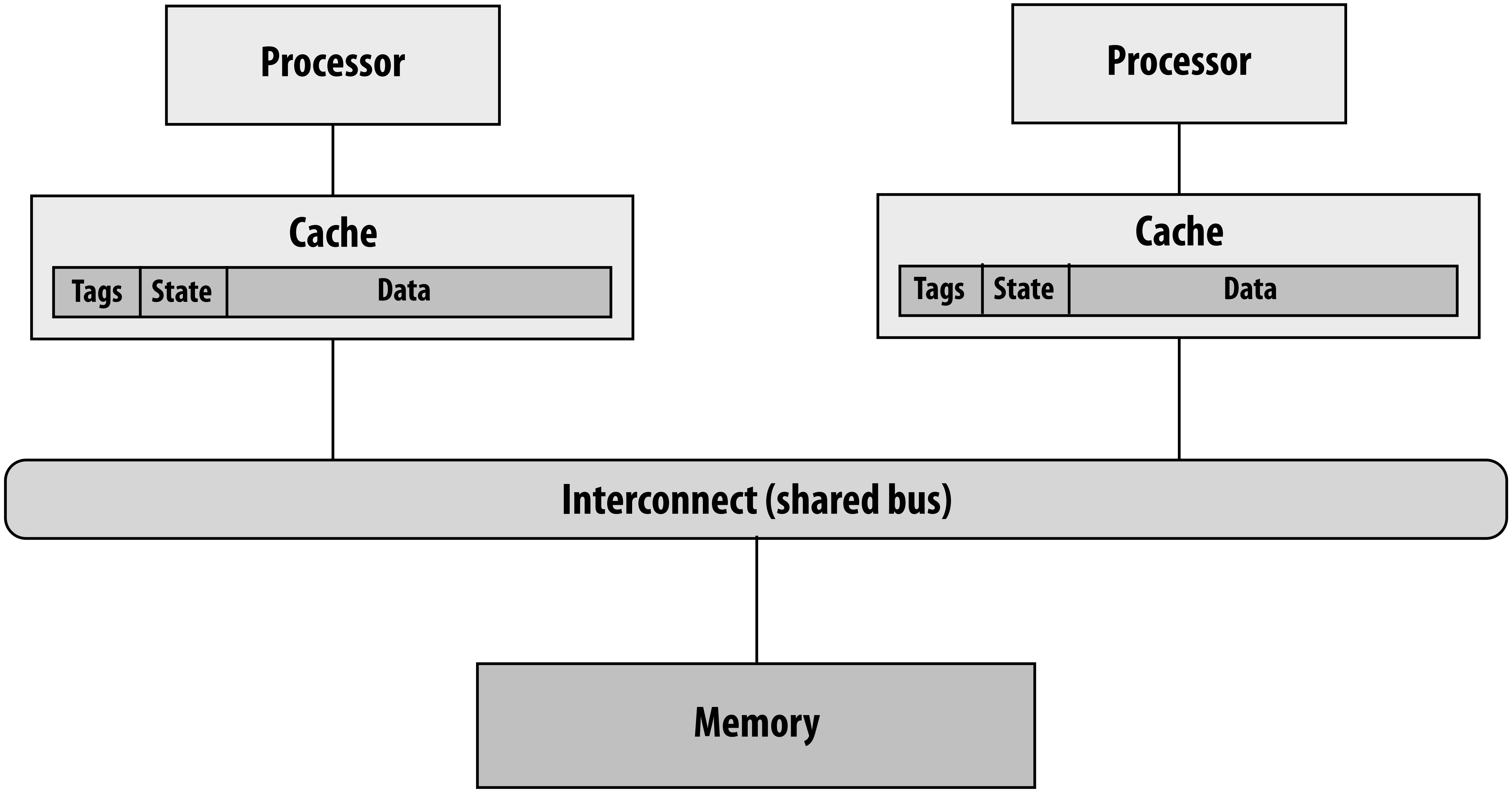
Review: MESI state transition diagram



Part 1:
A basic implementation of snooping
(assuming an “atomic” bus)

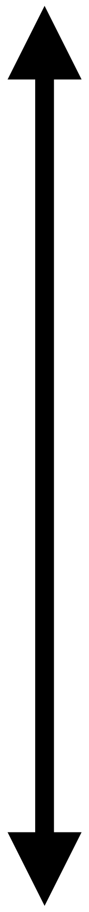
Consider a basic system design

- One outstanding memory request per processor
- Single level, write-back cache per processor
- Cache can stall processor as it is carrying out coherence operations
- System interconnect is an atomic shared bus (one cache can send data over bus at a time)



Transaction on an atomic bus

1. Client is granted bus access (result of arbitration)
2. Client places command on bus (may also place data on bus)



3. Response to command by another bus client placed on bus
4. Next client obtains bus access (arbitration)

Cache miss logic on a single core processor

1. Determine location in cache (using appropriate bits of address)
2. Check cache tags (to determine if line is in cache)

   [Let's assume no matching tags, so must read data from memory]
3. Request access to bus
4. Wait for bus grant (bus "arbitrator" manages access to bus)
5. Send address + command on bus
6. Wait for command to be accepted
7. Receive data on bus

Address
Data

In a multi-core processor:

For BusRd, BusRdX: no other bus transactions allowed between issuing address and receiving data

Flush: address and data sent simultaneously, received by memory before any other transaction allowed on bus

Reporting snoop results

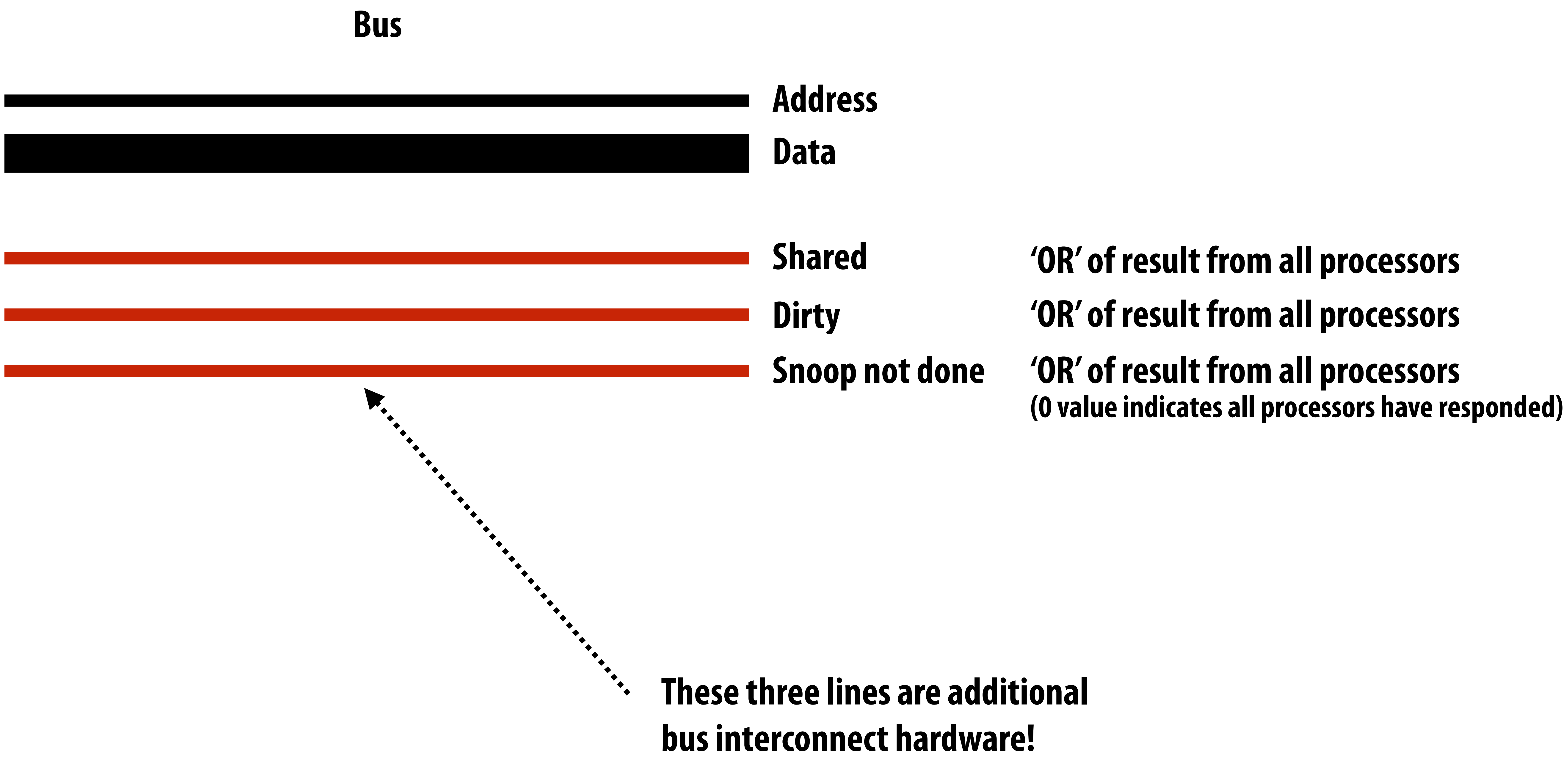
- Let's assume a cache read miss (BusRd command on bus)
- Response of all caches must appear on bus
  - Is line dirty in some cache? If so, memory should not respond
  - Is line shared? If so, cache should load into S state, not E

Memory needs to know what to do

Loading cache needs to know what to do

How are snoop results communicated?

How to report snoop results



Bus

Address

Data

Shared — 'OR' of result from all processors

Dirty — 'OR' of result from all processors

Snoop not done — 'OR' of result from all processors
(0 value indicates all processors have responded)

These three lines are additional bus interconnect hardware!

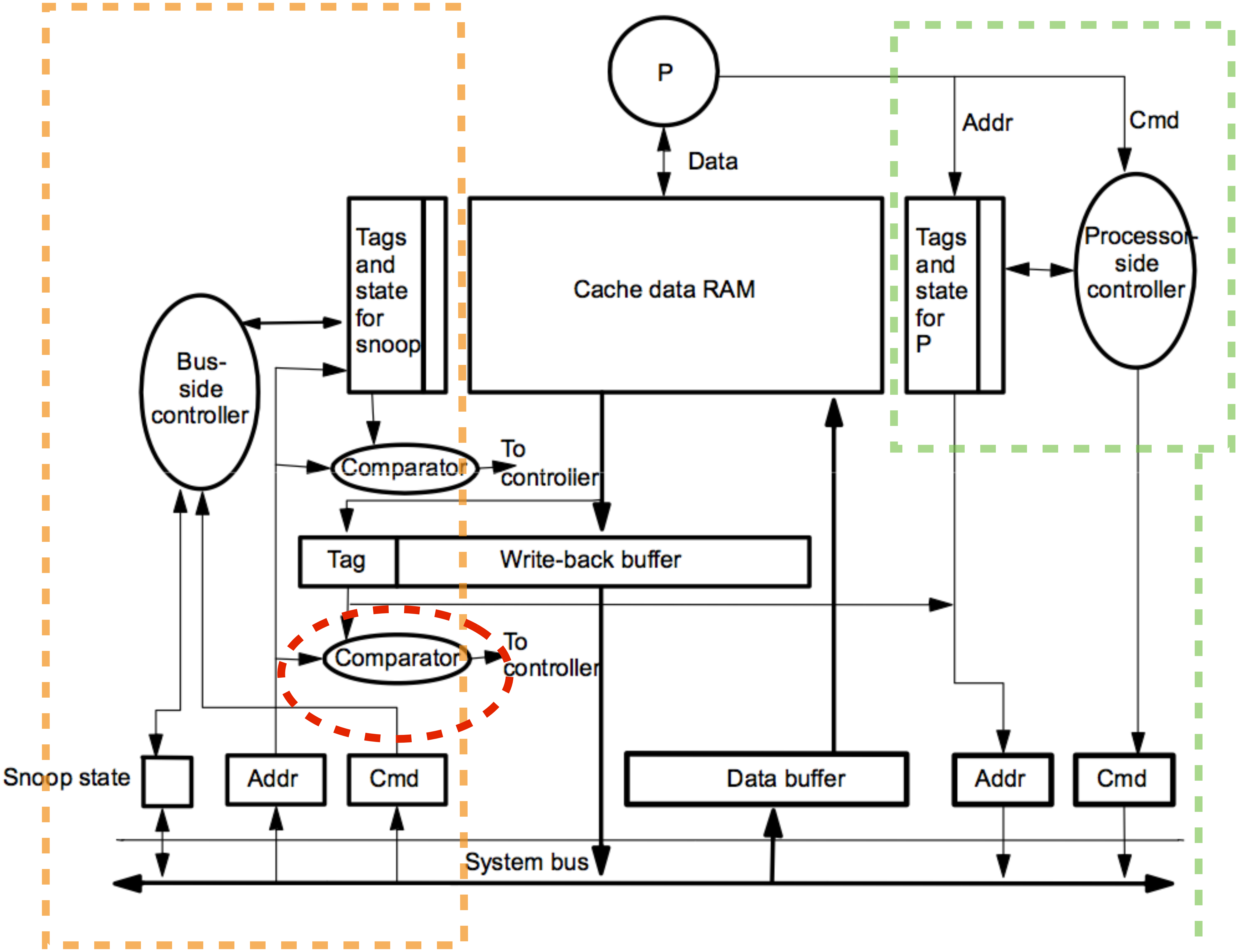
Handling write back of dirty cache lines

- **Replacing a dirty cache line involves two bus transactions**
  1. Read incoming line (line requested by processor)
  2. Write outgoing line (evicted dirty line in cache that must be flushed)

- **Ideally would like the processor to continue as soon as possible (it should not have to wait for the flush of the dirty line to complete)**

- **Solution in modern processors: write-back buffer**
  - Stick line to be evicted (flushed) in a "write-back buffer"
  - Immediately load requested line (allows processor to continue)
  - Flush contents of write-back buffer at a later time

Cache with write-back buffer



What if a request for the address of the data in the write-back buffer appears on the bus?

Snoop controller must check the write-back buffer addresses in addition to cache tags.

If there is a write-back buffer match:

1. Respond with data from write-back buffer rather than cache
2. Cancel outstanding bus access request (for the write back)

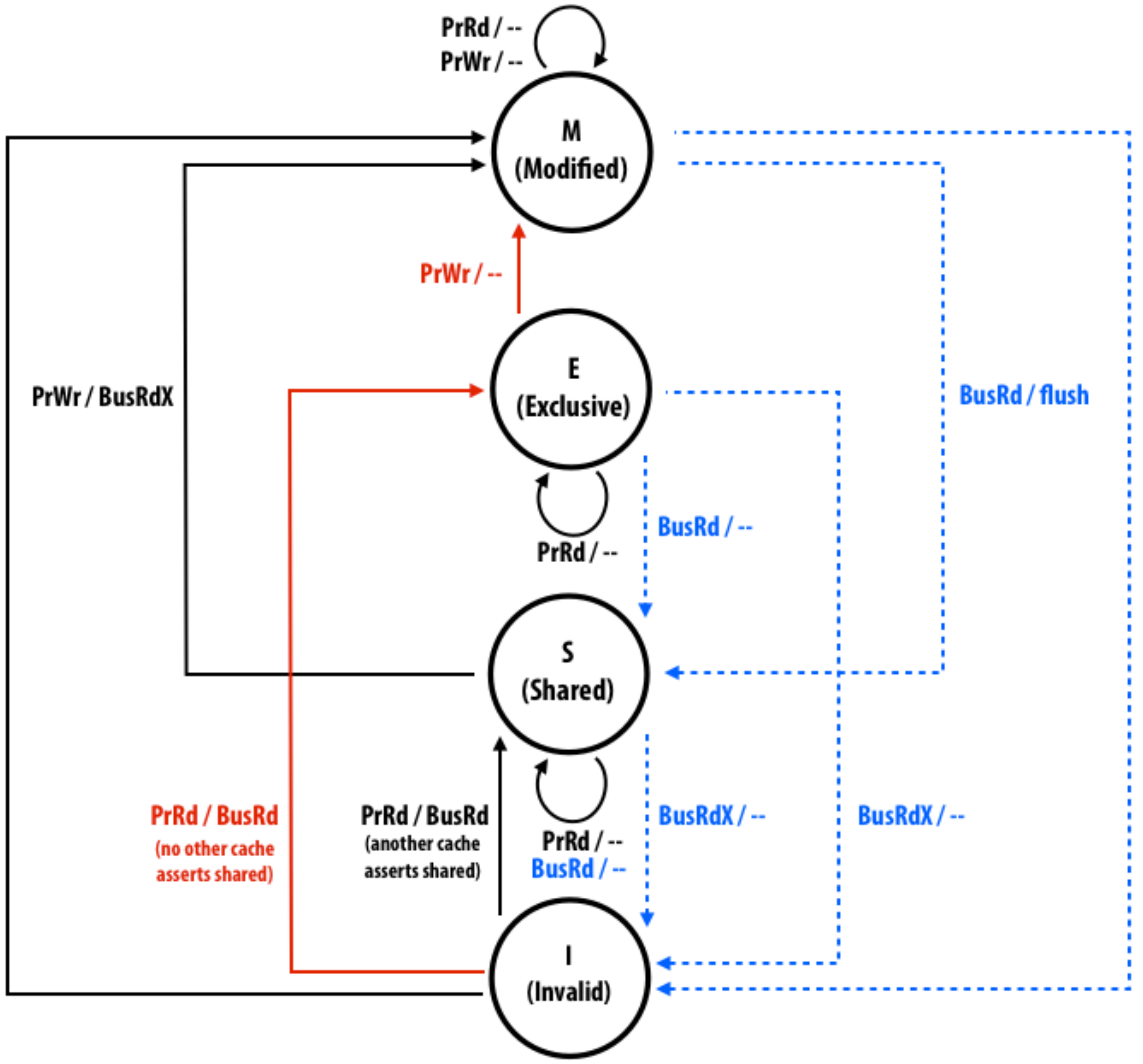
these hardware components handle snooping related tasks

these hardware components handle processor-related requests

Figure credit: Culler, Singh, and Gupta

Main point: in practice, state transitions are not atomic (they involve many operations in a modern computer)

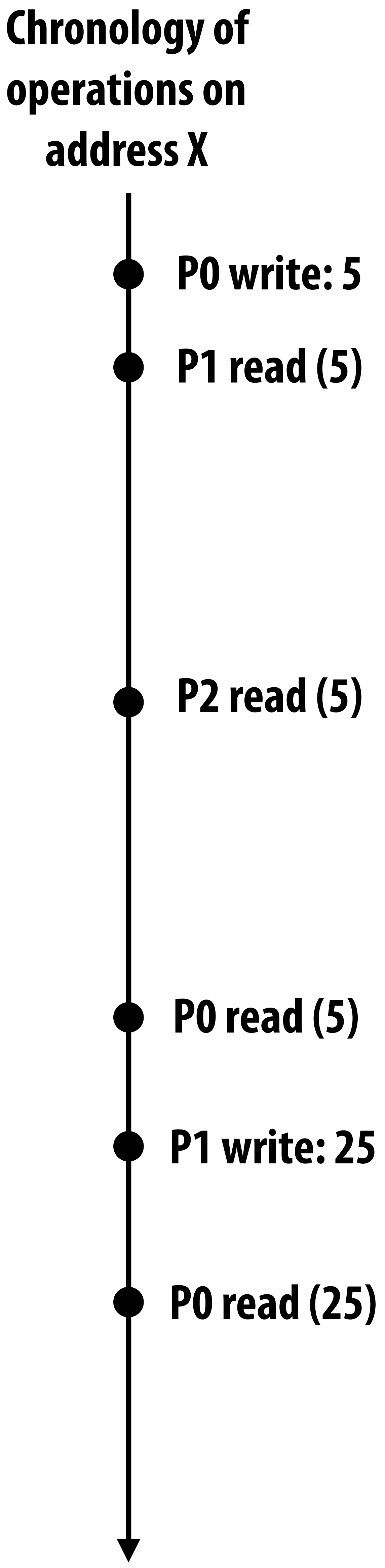
- Coherence protocol state transition diagrams (like the one below) assumed that transitions between states were atomic
- But in reality there is a sequence of operations the system performs as a result of a memory operation (look up cache tags, arbitrate for bus, wait for actions by other controllers, ...)



So when is a write “done”?

- Remember, memory coherence says there must be some serial order (a timeline) of all read and write operations to the same address that is consistent with the results observed by all processors during program execution

- So given what we know about how coherence is implemented, at what point is the write “committed” to the timeline? (when do all processors agree that it has occurred?)



Self check: when does a write "commit?"

- Consider a sequence of operations a machine performs when carrying out a write (consider write miss scenario)

1. Core issues STORE X ← R0 instruction
2. Look up line in cache (assume line not in cache or not in M state)
3. Arbitrate for bus
4. Place BusRdX on bus / other processors snoop request
5. Memory responds with data for cache line containing X
6. Contents of R0 written to appropriate bytes of cache line

- When does the write "commit"?
  - In other words, at what point are we guaranteed that the write will be "visible" to other processors?

Self check: when does a write "commit?"

- A write commits when a read-exclusive transaction appears on bus and is acknowledged by all other caches
  - At this point, the write is "committed"
  - All future reads will reflect the value of this write (even if data from P has not yet been written to P's dirty cache line, or to memory)
    - Why is this?
  - Key idea: order of transactions on the bus defines the global order of writes in the parallel program (write serialization requirement of coherence)

- Commit != complete: a write completes when the store instruction is done (updated value has been put in the cache line)

- Why does a write-back buffer not affect when a write commits?

First-half summary: parallelism and concurrency in real hardware implementation of coherence

- Processor, cache, and bus all are hardware resources operating in parallel!
  - Often contending for shared resources:
    - Processor and bus contend for cache
    - Difference caches contend for bus access

- “Memory operations” are <u>abstracted</u> by the architecture as atomic (e.g., loads, stores) are <u>implemented</u> via multiple operations involving all of these hardware components

Part 2:

Building the system around non-atomic bus transactions

Review: transaction on an atomic bus

1. Client is granted bus access (result of arbitration)
2. Client places command on bus (may also place data on bus)

Problem: bus is idle while response is pending
(this decreases effective bus bandwidth)

This is bad, because the interconnect is a limited, shared resource in a multi-processor system.
(So it is important to use it as efficiently as possible)

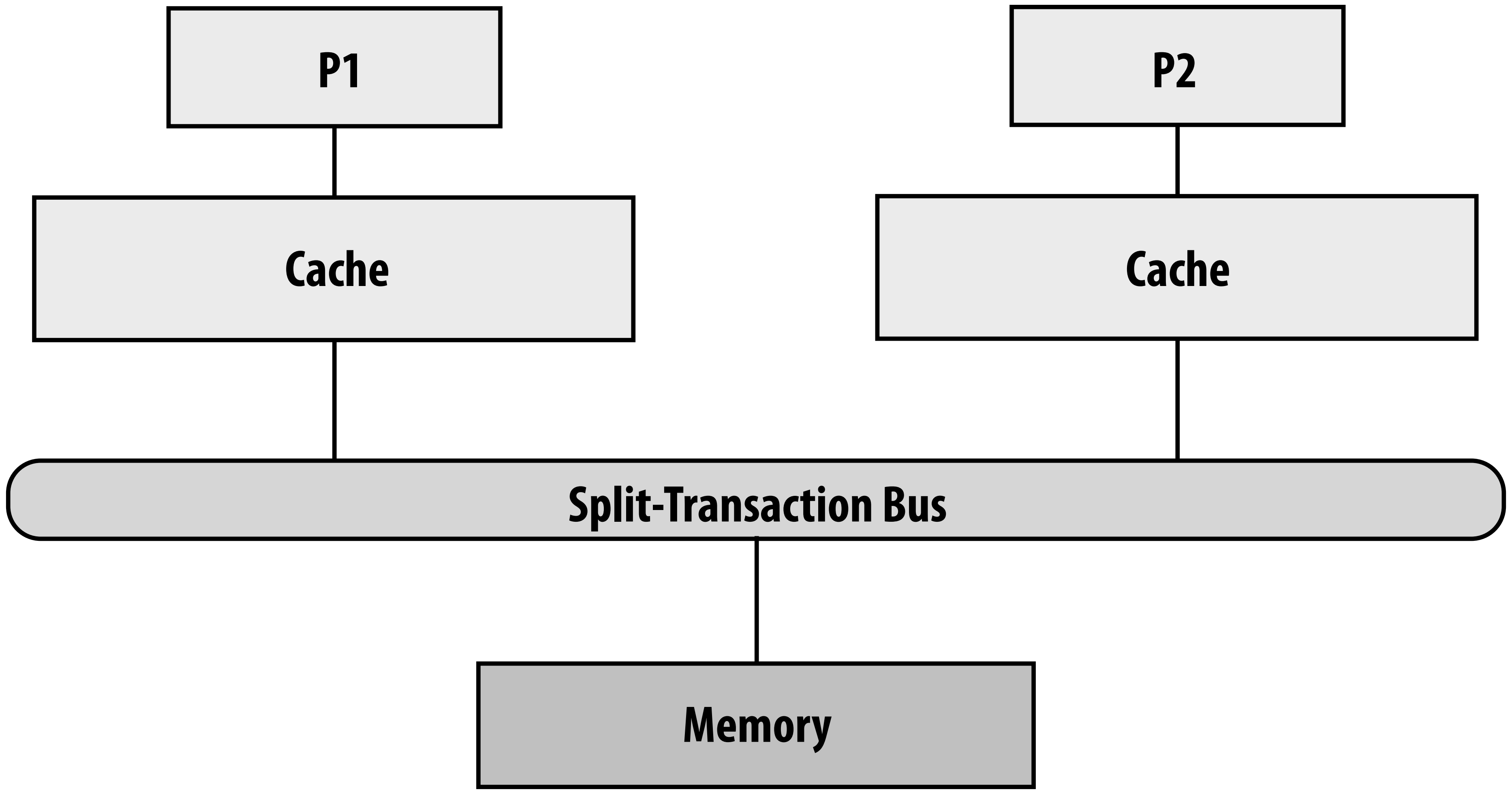
3. Response to command by another bus client placed on bus
4. Next client obtains bus access (arbitration)

Split-transaction bus

Bus transactions are split into two transactions:

1. The request
2. The response

Other transactions can use the bus in between a transaction's request and response.



Consider this scenario:

Read miss to A by P1

Bus upgrade of B by P2

Possible timeline of events on a split-transaction bus:

P1 gains access to bus

P1 sends BusRd A command
[memory starts fetching data now…]

P2 gains access to bus

P2 sends BusUpg command

Memory gains access to bus

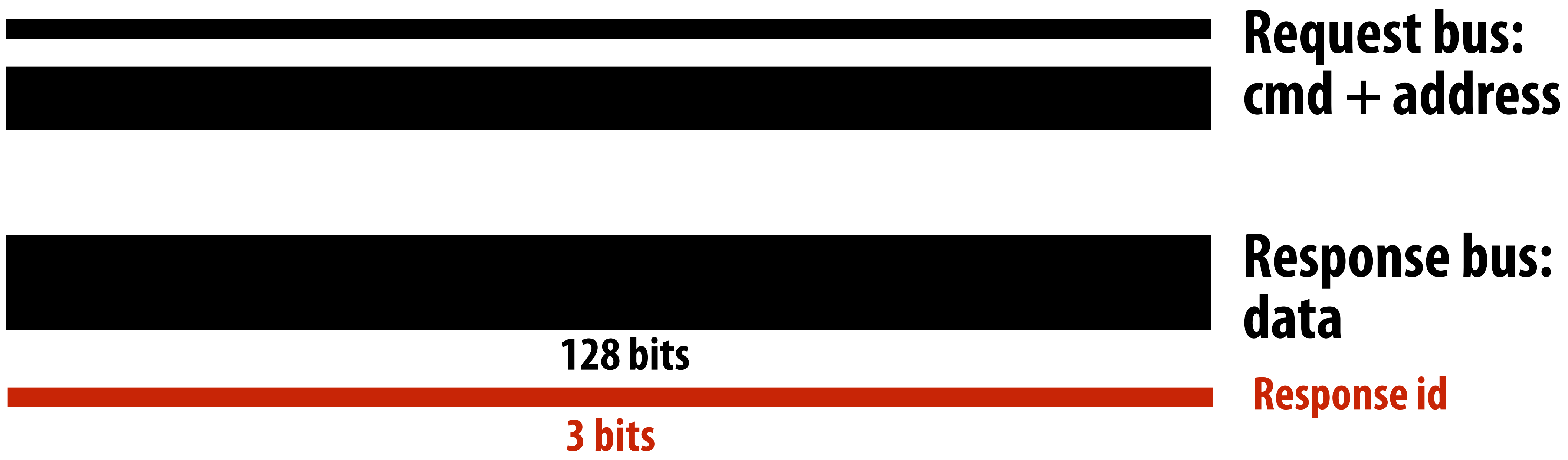
Memory places A on bus (response)

A basic design

- Up to eight outstanding requests at a time (system wide)
- Responses <u>need not</u> occur in the same order as requests
  - But request order establishes the total order for the system
- Flow control via negative acknowledgements (NACKs)
  - Operations can be aborted, forcing a retry

Initiating a request

Can think of a split-transaction bus as two separate buses: a request bus and a response bus.

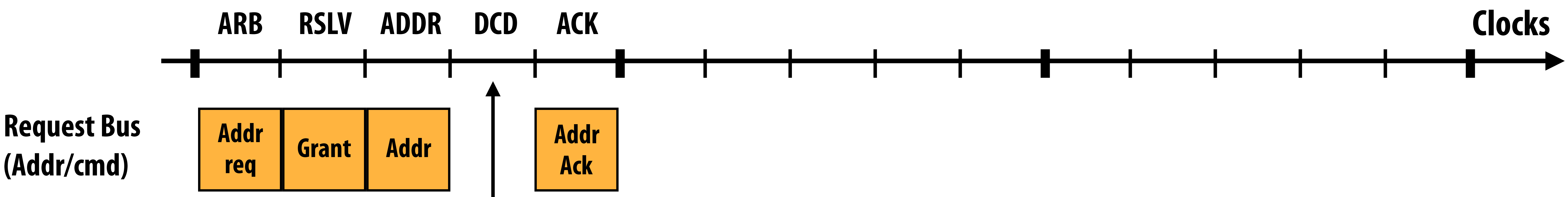


Step 1: Requestor asks for request bus access

Step 2: Bus arbiter grants access, assigns transaction an id

Step 3: Requestor places command + address on the request bus

Read miss: cycle-by-cycle bus behavior (phase 1)



Caches acknowledge this snoop result is ready
(or signal they could not complete snoop in time here (e.g., NACK transaction)

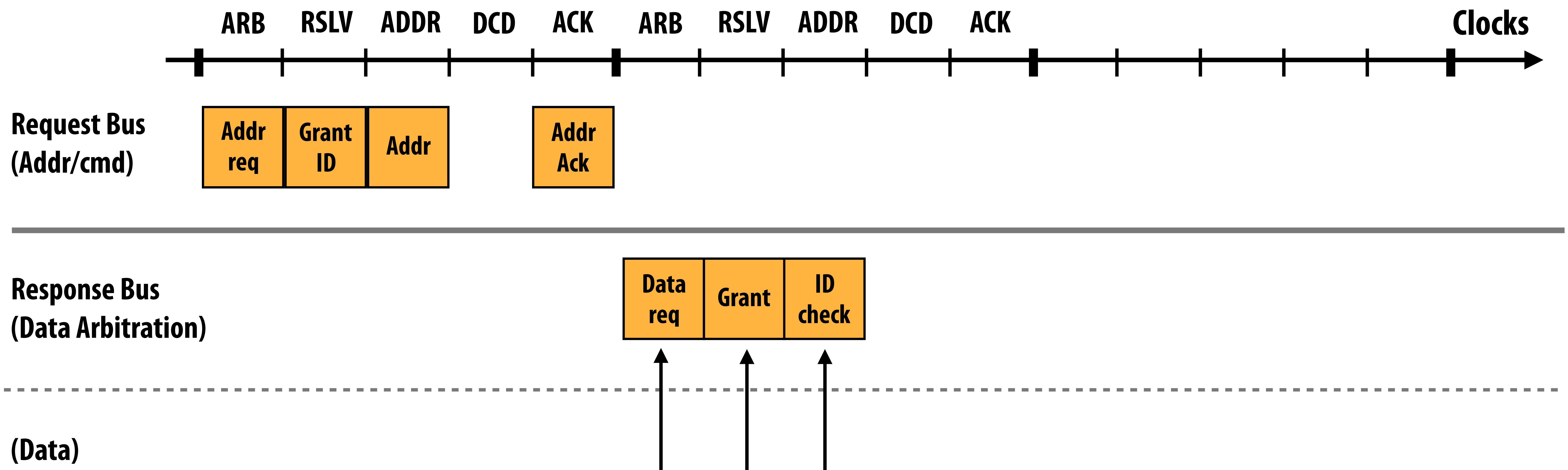
Caches perform snoop: look up tags, update cache state, etc.
Memory operation commits here!
(NO BUS TRAFFIC)

Bus "winner" places command/address on the bus

Request resolution: address bus arbiter grants access to one of the requestors
Special arbitration lines indicate what id is assigned to request

Request arbitration: cache controllers present request for address bus
(many caches may be doing so in the same cycle)

Read miss: cycle-by-cycle bus behavior (phase 2)

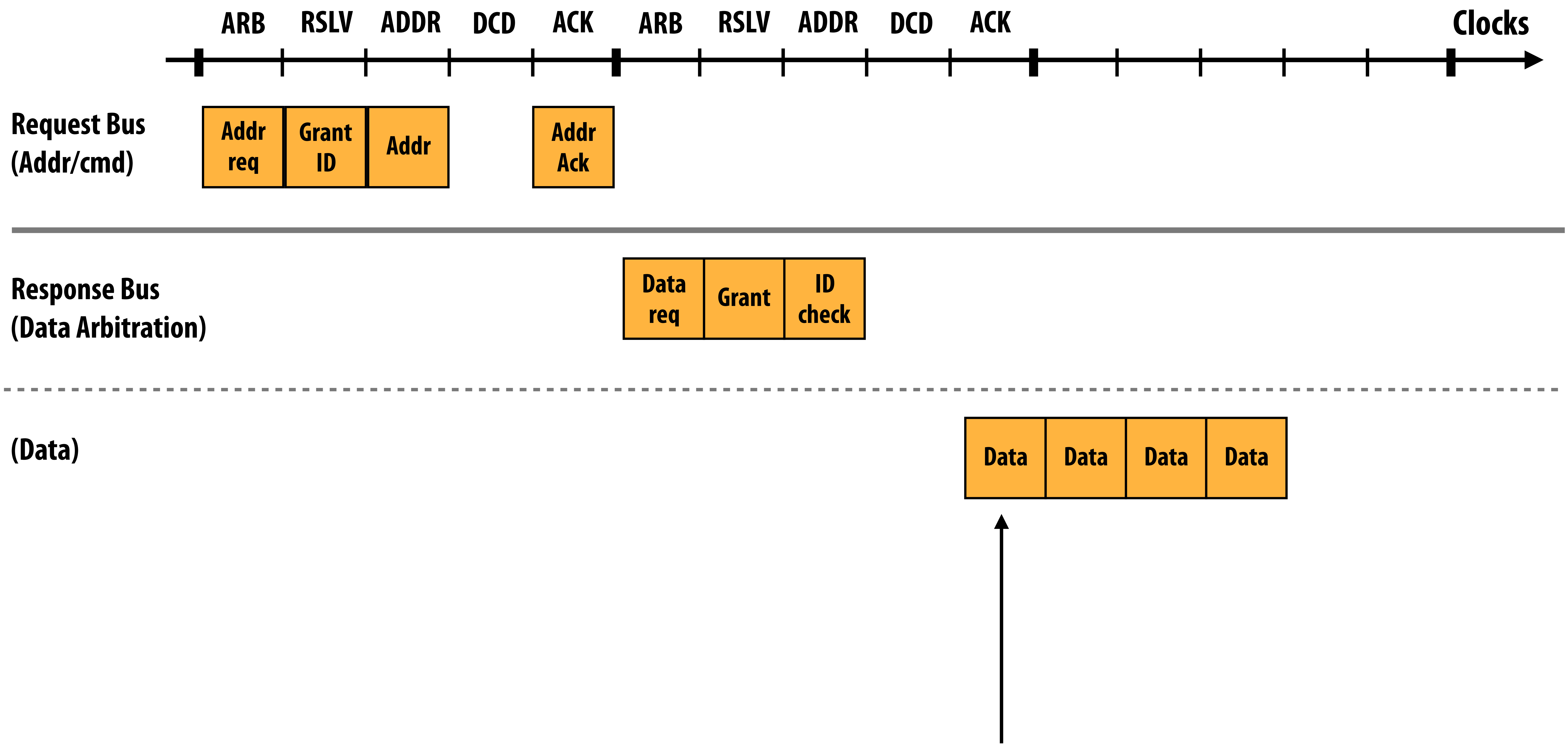


Original requestor signals readiness to receive response
(or lack thereof: requestor may be busy at this time)

Data bus arbiter grants one responder bus access

Data response arbitration: responder presents intent to respond
to request with specific ID
(many caches --or memory-- may be doing so in the same cycle)

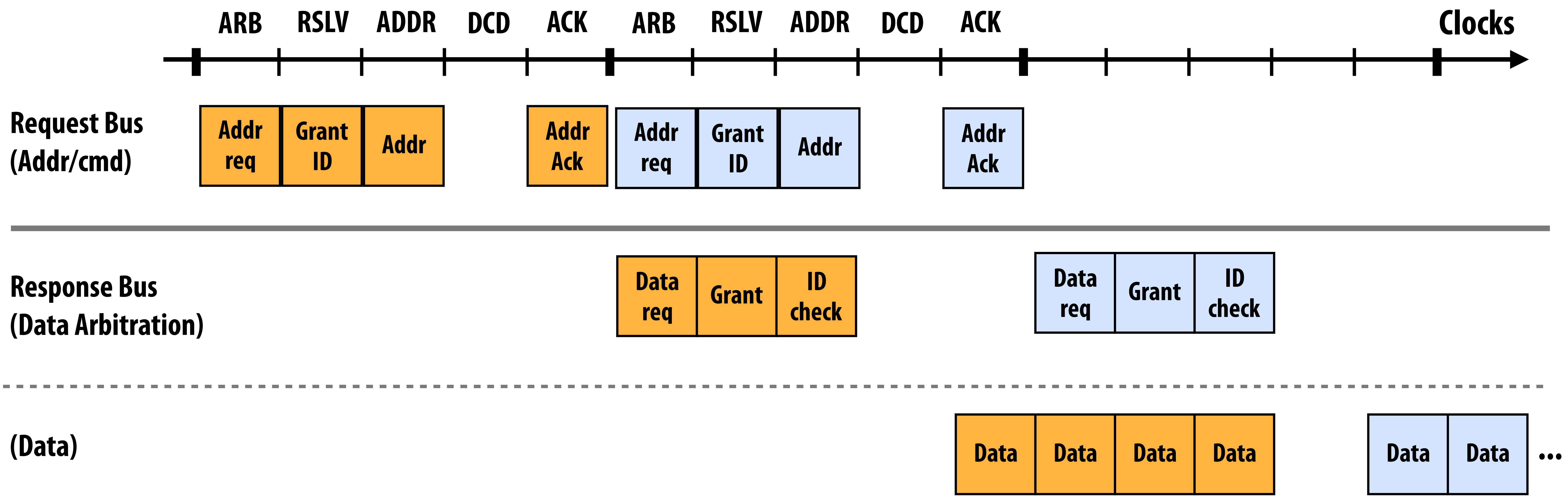
Read miss: cycle-by-cycle bus behavior (phase 3)



Responder places response data on data bus
Caches present snoop result for request with the data
Request table entry is freed

Here: assume 64 byte cache lines → 4 cycles on 128 bit bus

Pipelined transactions on bus



ARB RSLV ADDR DCD ACK ARB RSLV ADDR DCD ACK Clocks

Request Bus (Addr/cmd)

Addr req | Grant ID | Addr | Addr Ack | Addr req | Grant ID | Addr | Addr Ack

Response Bus (Data Arbitration)

Data req | Grant | ID check | Data req | Grant | ID check

(Data)

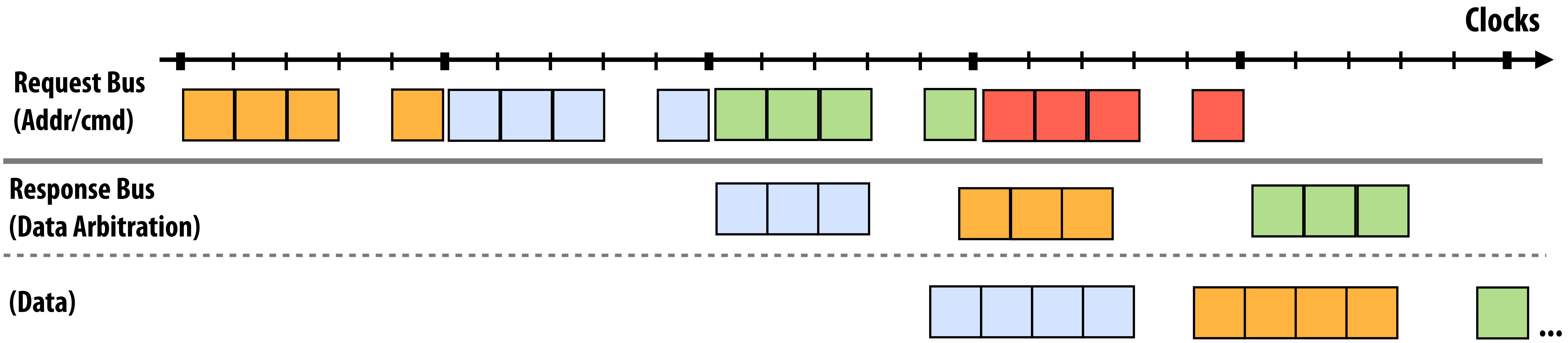
Data | Data | Data | Data | Data | Data ...

🟧 = memory transaction 1

🟦 = memory transaction 2

Note: write backs and BusUpg transactions do not have a response component
(write backs acquire access to both request address bus and data bus as part of "request" phase)

Pipelined transactions



🟧 = memory transaction 1

🟦 = memory transaction 2

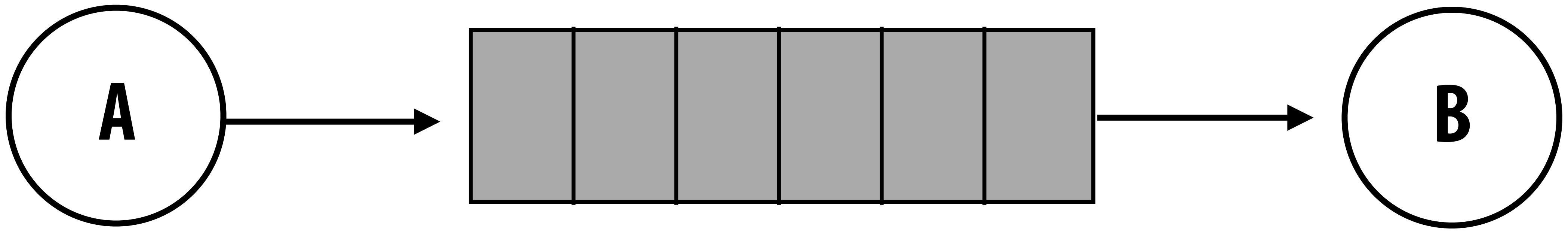
🟩 = memory transaction 3

🟥 = memory transaction 4

Note out-of-order completion of bus transactions

(in this figure, transaction 2 completes before 1)

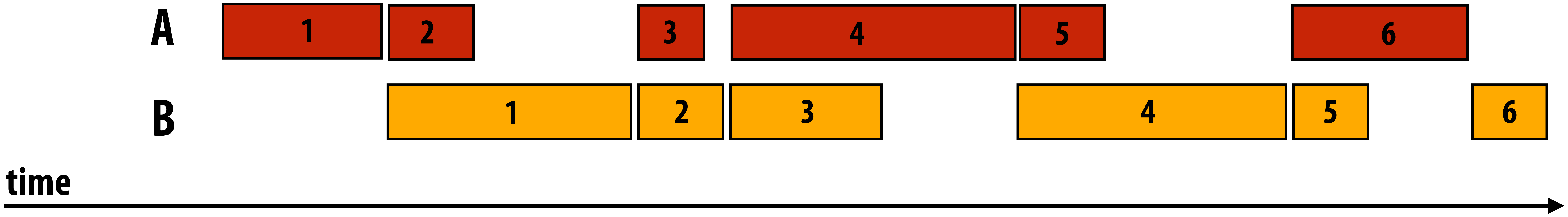
Why do we have queues in a parallel system?



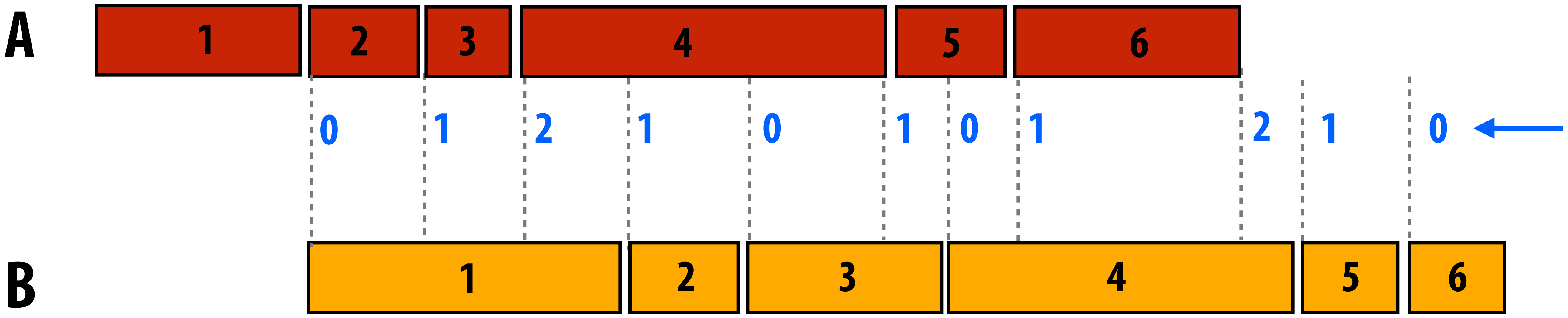
Answer: to accommodate variable (unpredictable) rates of production and consumption.

As long as A and B produce and consume data at the <u>same rate on average</u>, both workers can run all the time!

No queue: notice A stalls waiting for B to accept new input (and B sometimes stalls waiting for A to produce new input).



With queue of size 2: A and B never stall



Size of queue when A completes a piece of work (or B begins work)

Multi-level cache hierarchies

Numbers indicate steps in a cache miss from processor on left. Serviced by cache on right.

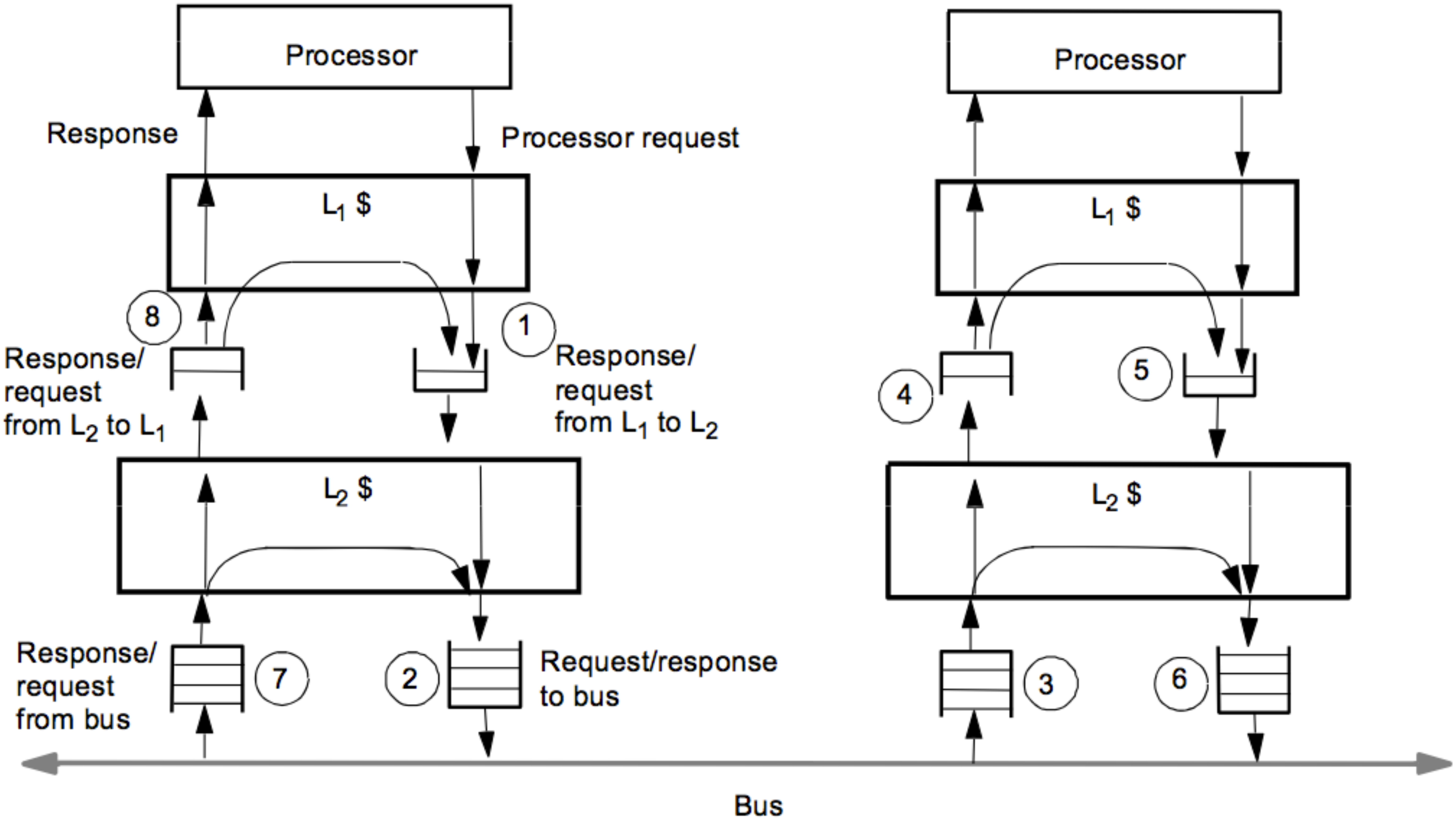
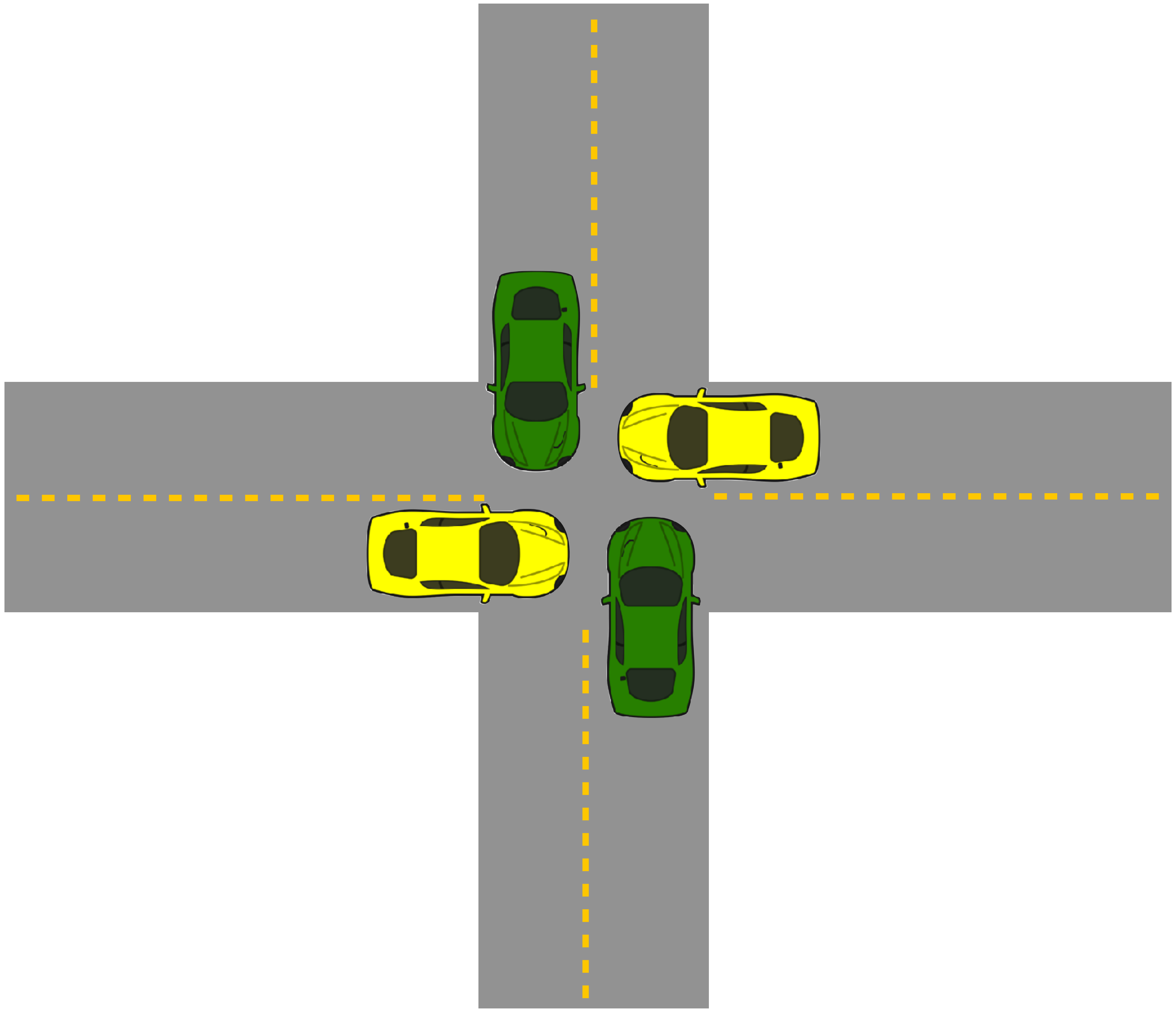


Figure credit: Culler, Singh, and Gupta

Deadlock / Livelock

Deadlock



Deadlock is a state where a system has outstanding operations to complete, but no operation can make progress.

Can arise when each operation has acquired a shared resource that another operation needs.

In a deadlock situations, there is no way for any thread (or, in this illustration, a car) to make progress unless some thread relinquishes a resource ("backs up")

Deadlock in Pittsburgh :-(

Deadlock in Beijing

http://www.chicagonow.com/cheaper-than-therapy/2010/08/worlds-worst-traffic-jam-and-you-thought-chicago-traffic-was-bad

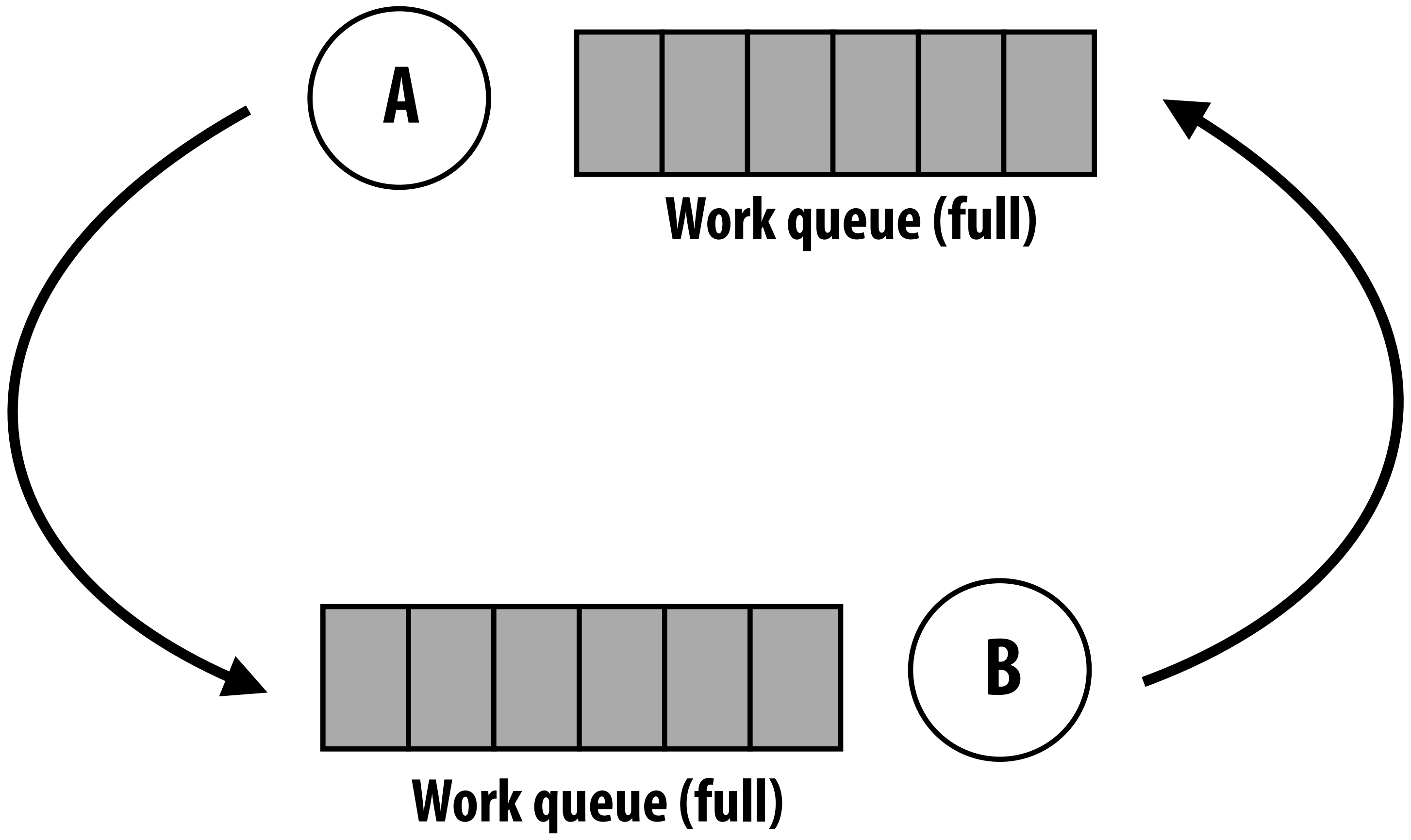
More deadlock

Credit: David Maitland, National Geographic

Why are these examples of deadlock?

Deadlock in computer systems

Example 1:



A produces work for B's work queue

B produces work for A's work queue

Queues are finite and workers wait if no output space is available

Example 2:

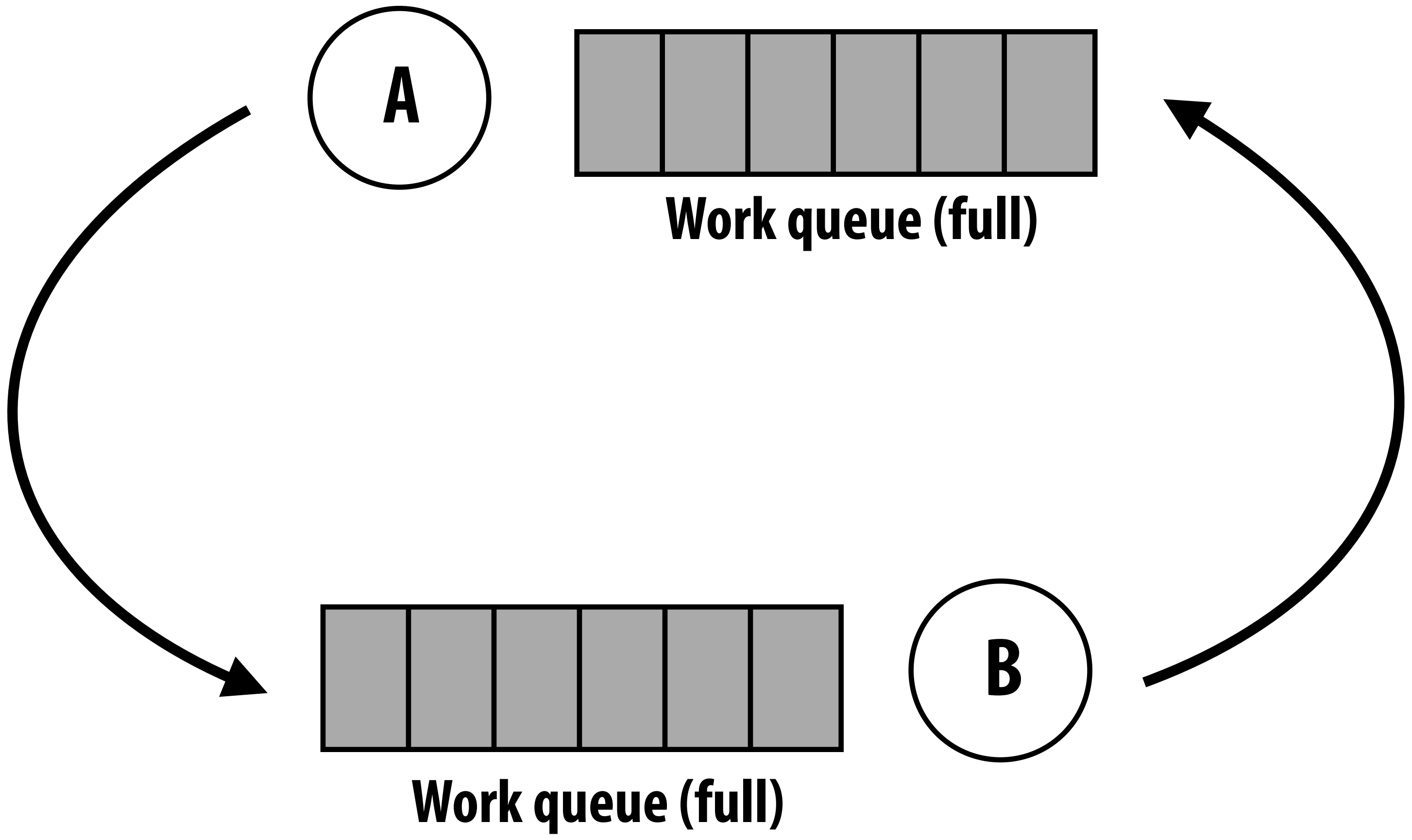
```
float msgBuf1[1024];
float msgBuf2[1024];

int threadId = getThreadId();

// send data to “neighbor” threads
MsgSend(msgBuf1, threadId+1, ...
MsgSend(msgBuf1, threadId-1, …

// receive data from “neighbor” threads
MsgRecv(msgBuf2, threadId+1, ...
MsgRecv(msgBuf2, threadId-1, ...
```

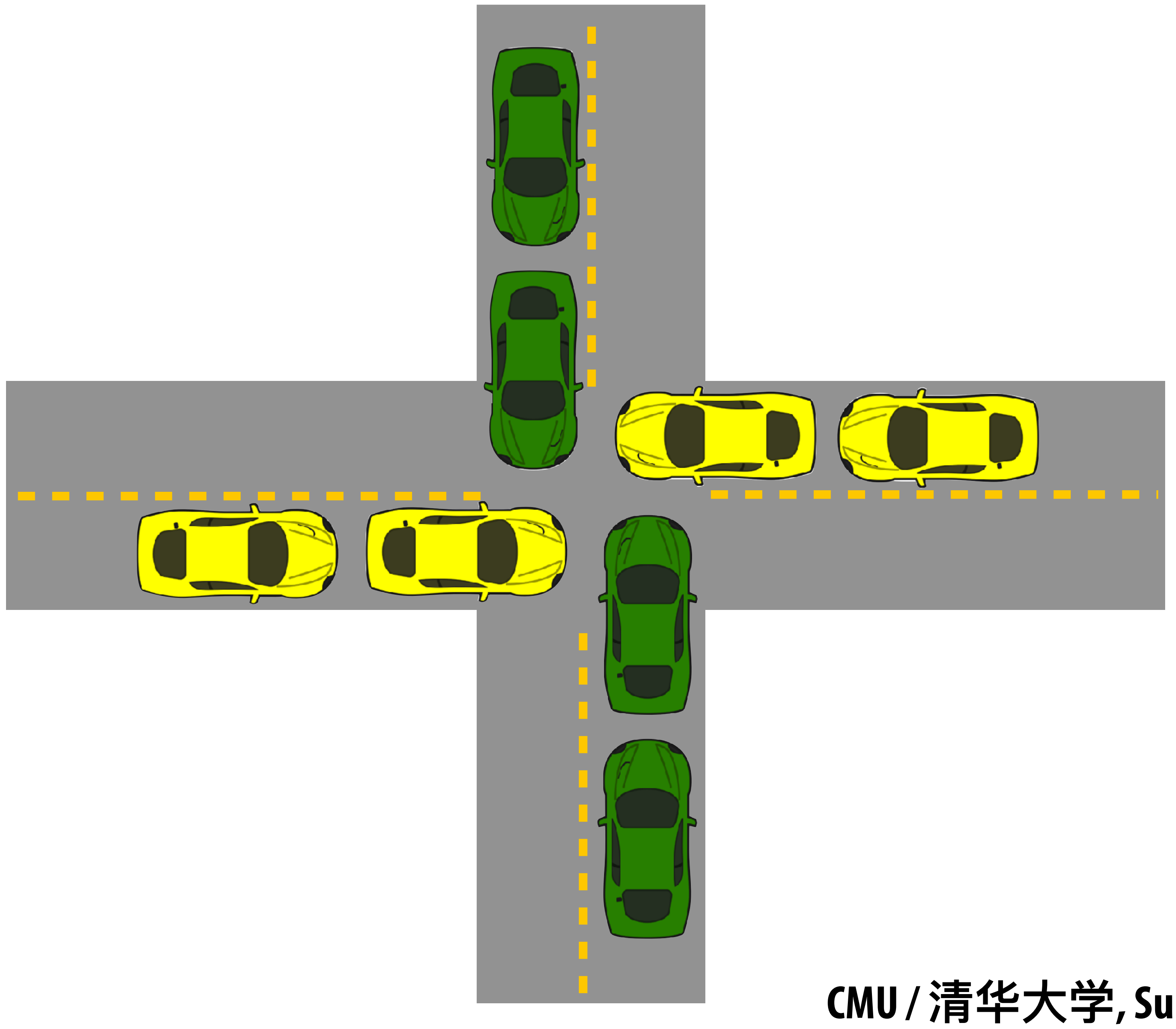
Recall our message passing example:
Every thread sends a message (using blocking send) to the processor with the next higher id

Then receives message from processor with next lower id.

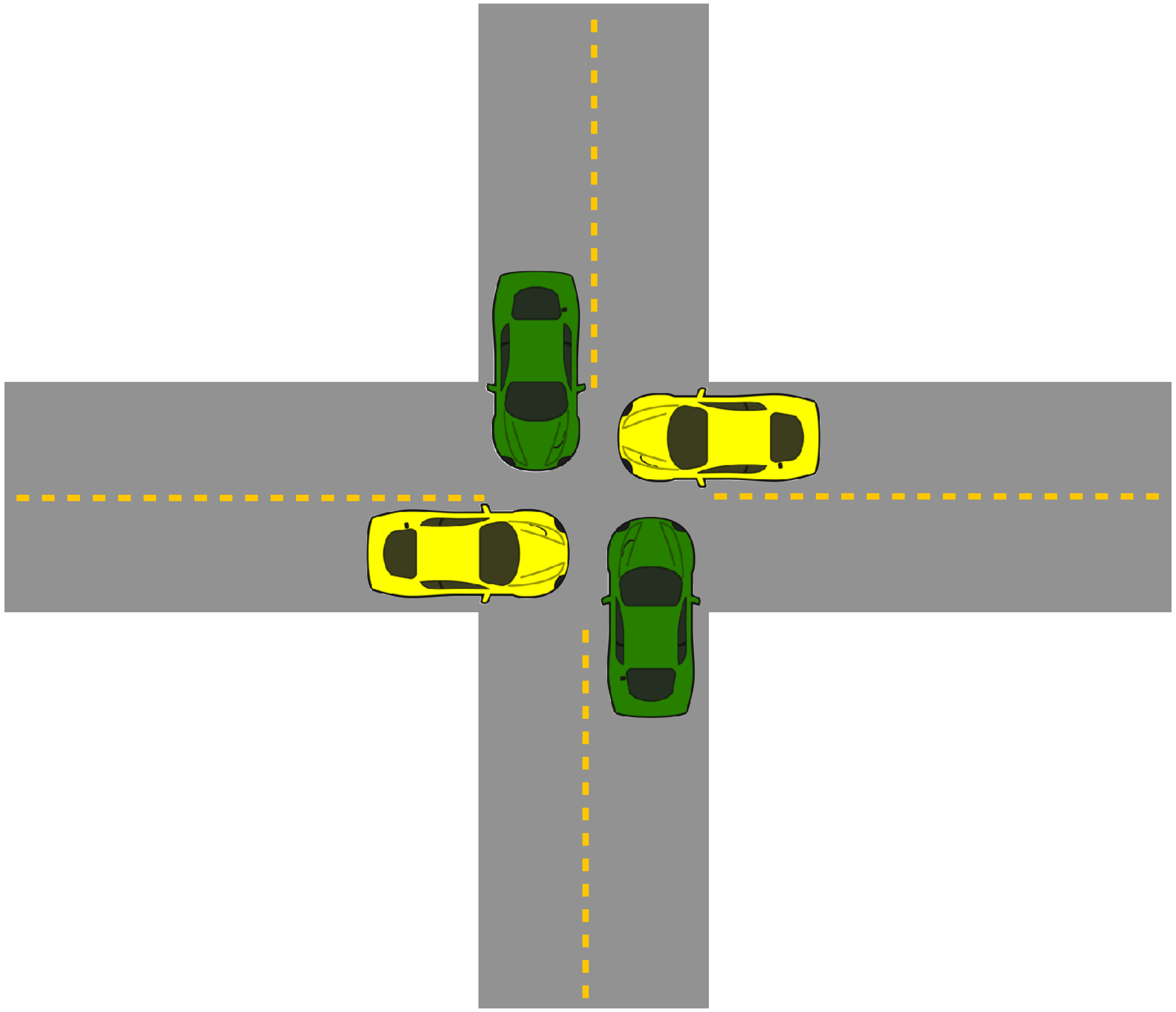
Required conditions for deadlock

1. Mutual exclusion: only one processor can hold a resource at once
2. Hold and wait: processor must hold the resource while waiting for other resources it needs to complete an operation
3. No preemption: processors do not give up resources until operation they wish to perform is complete
4. Circular wait: waiting processors have mutual dependencies (a cycle exists in the resource dependency graph)

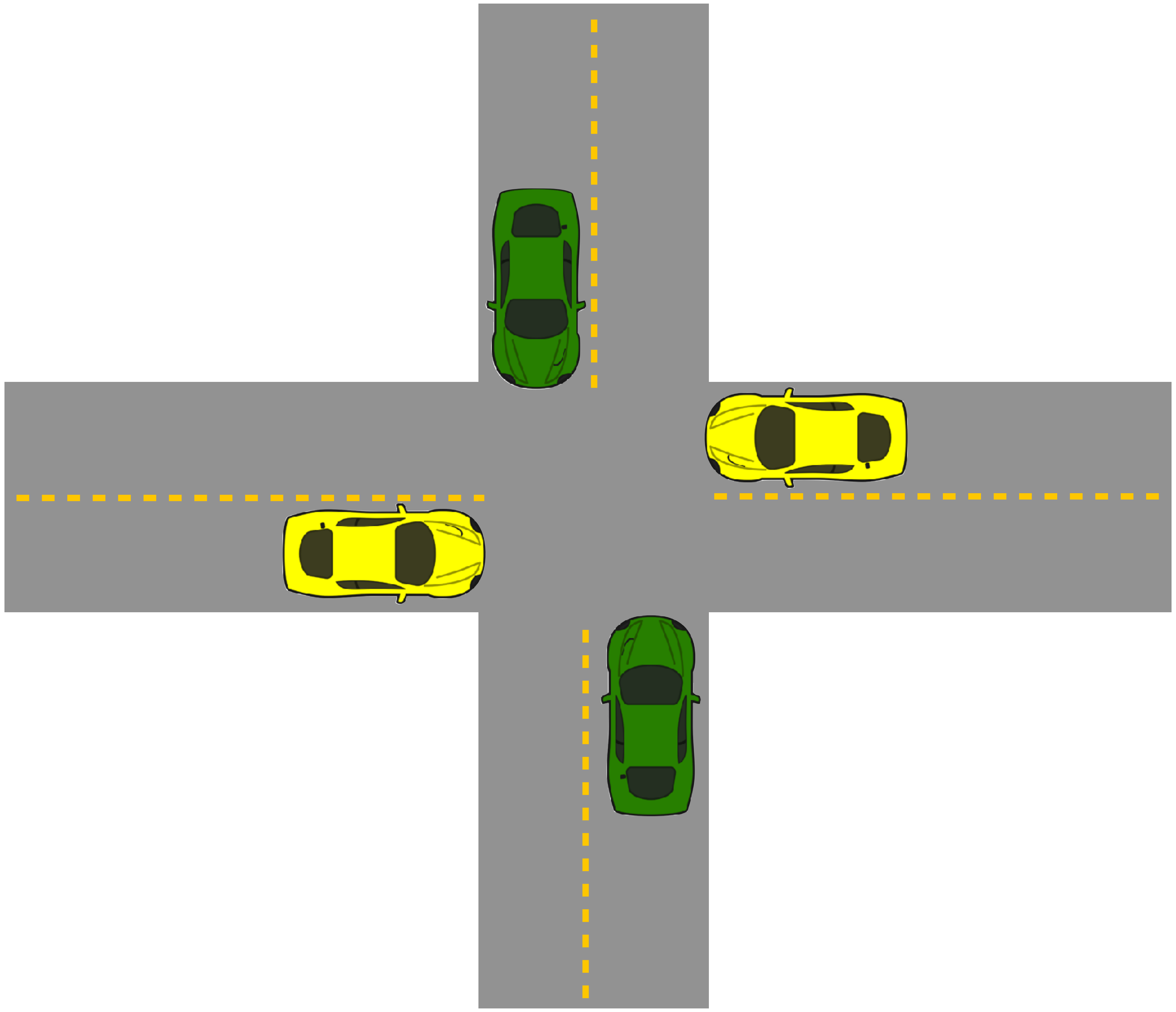




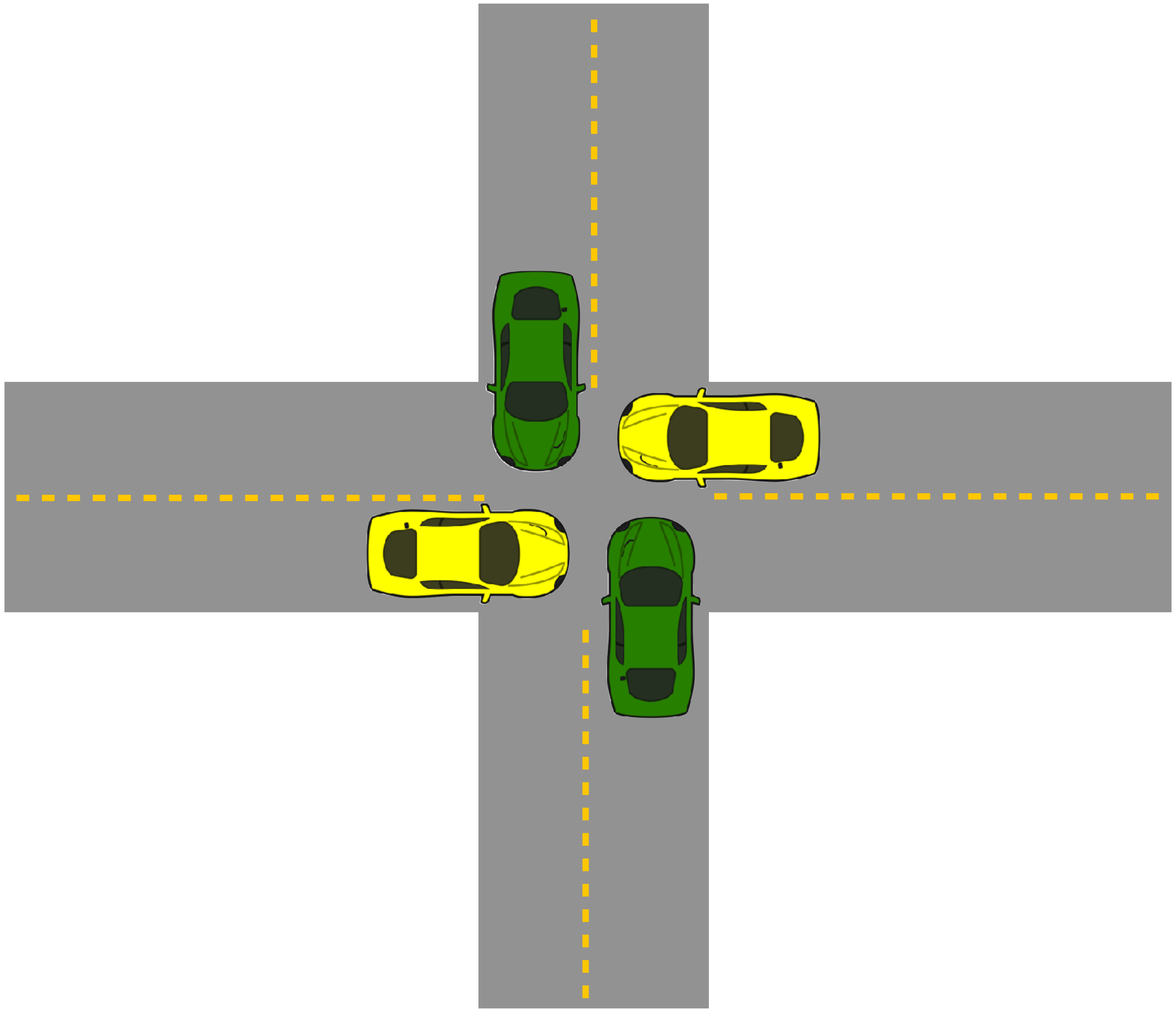
Livelock



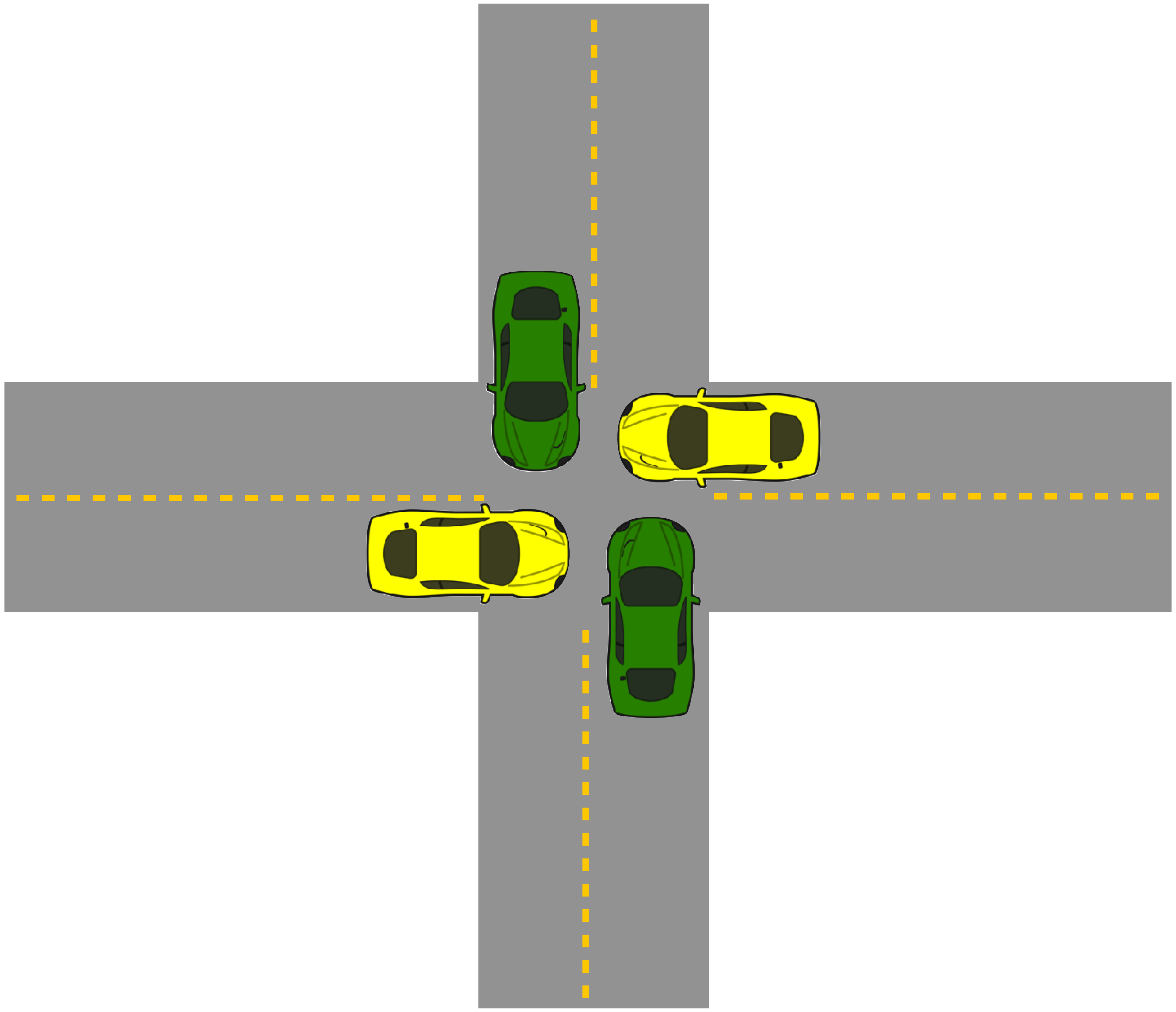
Livelock



Livelock



Livelock



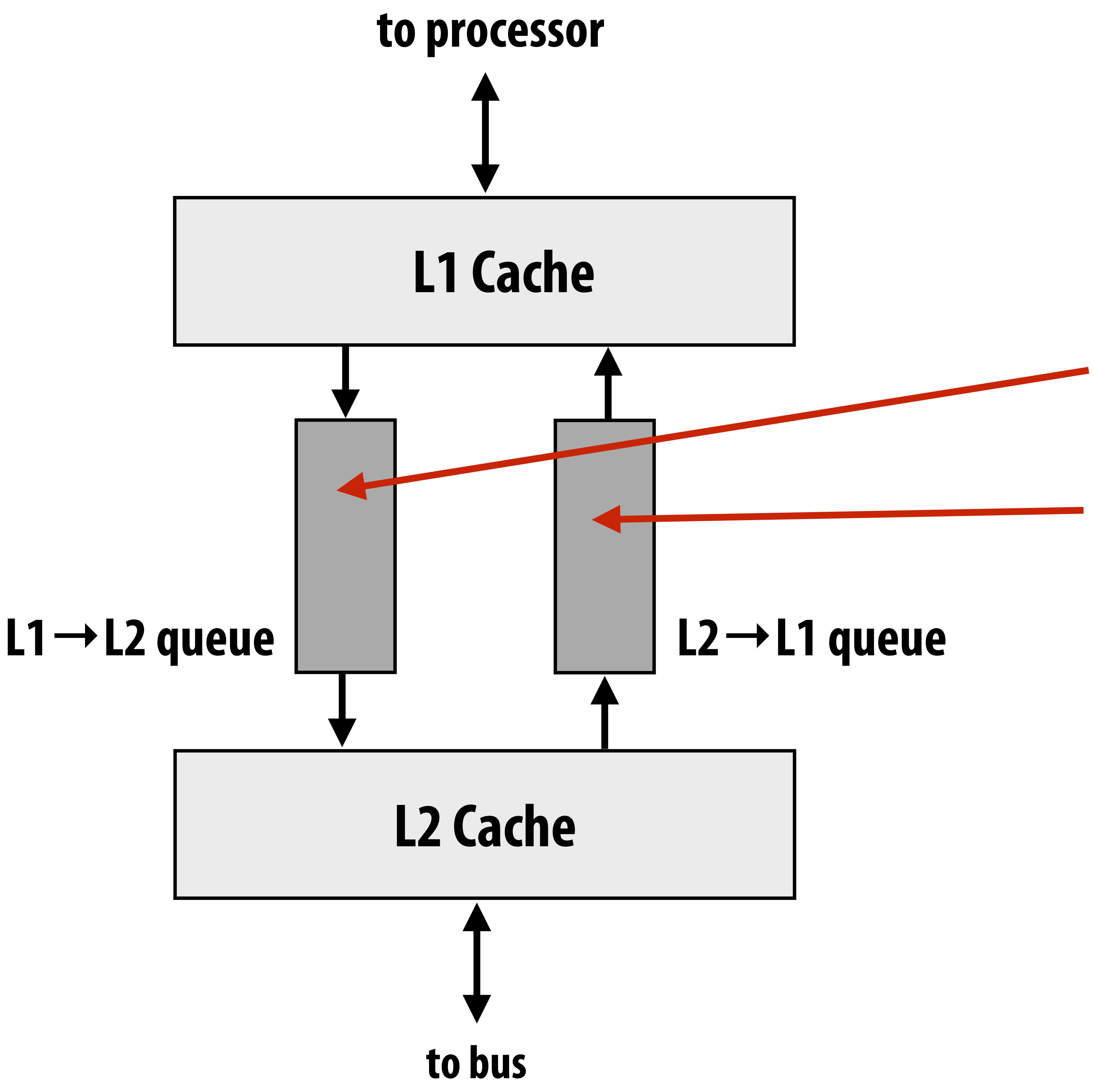
Livelock is a state where a system is executing many operations, but no thread is making meaningful progress.

Can you think of a good daily life example of livelock?

Computer system examples:

Operations continually abort and retry

Deadlock due to full queues



Assume buffers are sized so that the maximum queue size is one message. (buffer size = 1)

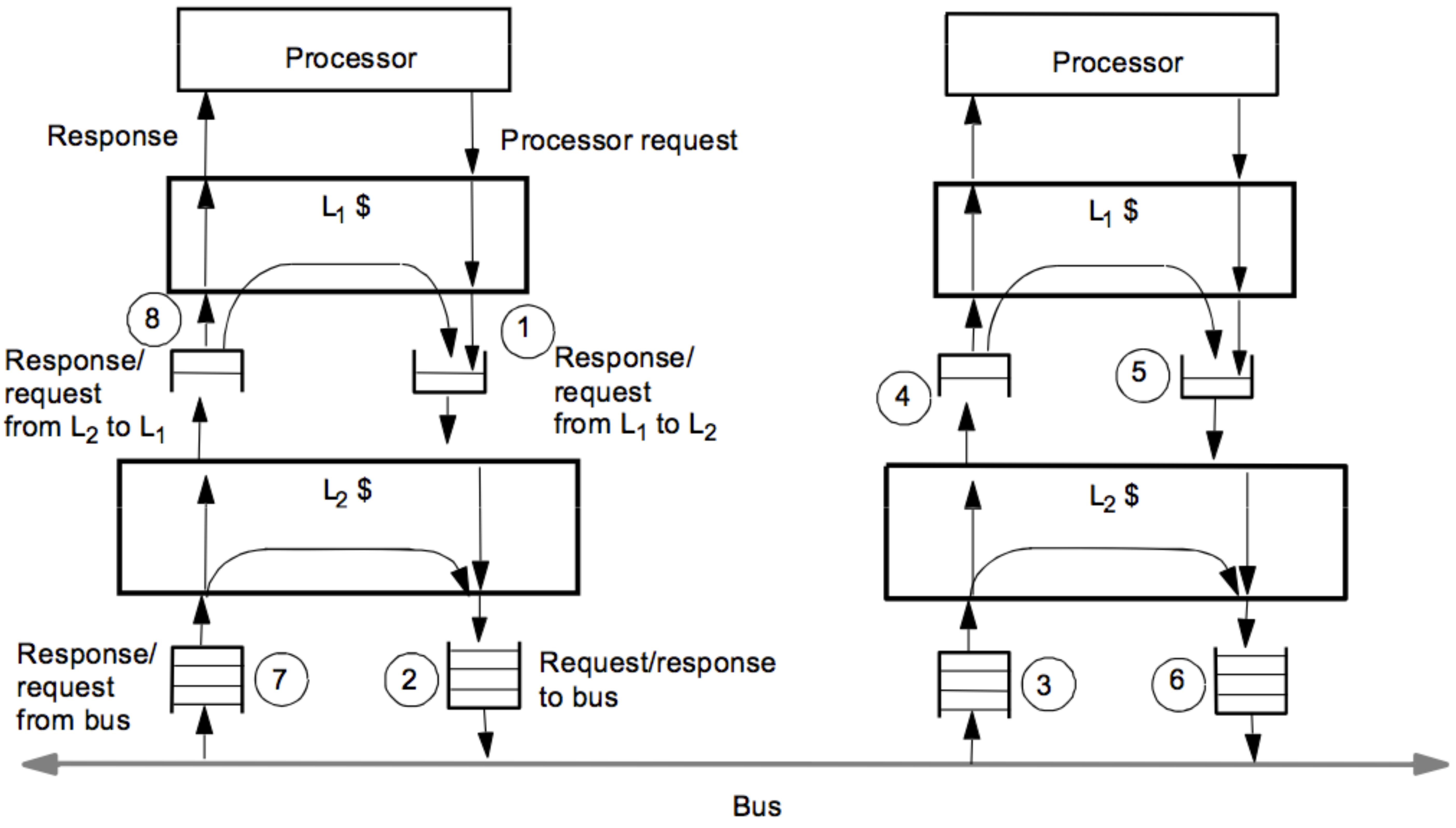
Outgoing read request (initiated by processor)

Incoming read request (due to another cache) **

Both requests generate responses that require space in the other queue (circular dependency)

** will only occur if L1 is write back

Avoiding deadlock by ensuring queues never fill

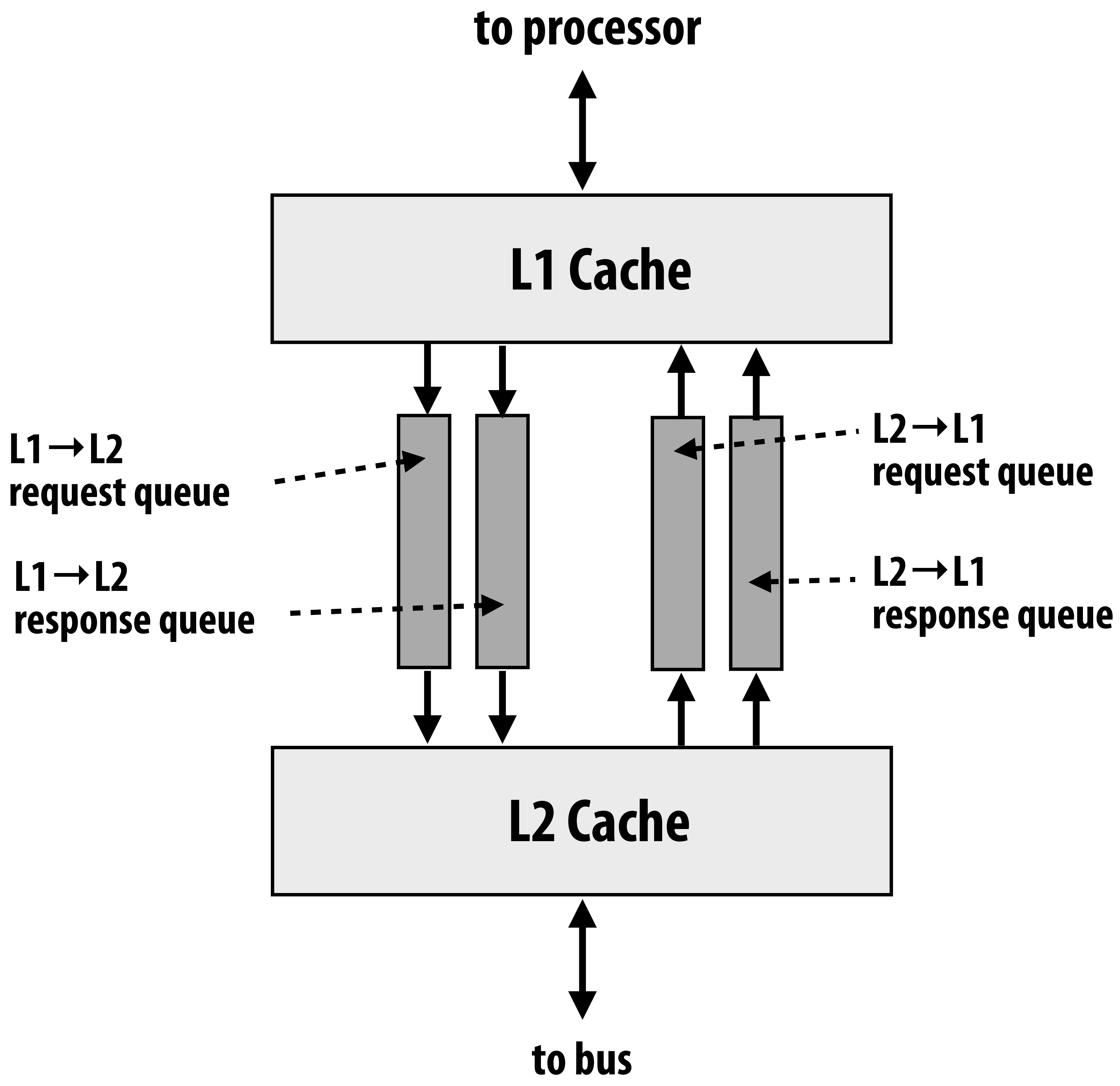


Assume one outstanding memory request per processor (2 cores x 8 requests per core = 16)

Sizing all buffers to accommodate the maximum number of outstanding requests on bus is one solution to avoiding deadlock. But a costly one!

Figure credit: Culler, Singh, and Gupta

Avoiding buffer deadlock with separate request/response queues (prevent circular wait)



System classifies all transactions as requests or responses

Key insight: responses can be completed without generating further transactions!

Requests INCREASE queue length

But responses REDUCE queue length

While stalled attempting to send a request, cache must be able to service responses.

Responses will make progress (they generate no new work so there's no circular dependence), eventually freeing up resources for requests

Putting it all together

Challenge exercise: describe everything that might occur during the execution of this line of code

```
int x = 10;    // assume this is a write to memory (the value
               // is not stored in register)
```

Class exercise: describe everything that might occur during the execution of this statement *

```
int x = 10;
```

* This list is certainly not complete, it's just what I came up with off the top of my head. (I think this would be a great job interview question!)

1. Virtual address to physical address conversion (TLB lookup)
2. TLB miss
3. TLB update (might involve OS)
4. OS may need to swap in page to get the appropriate page table (load from disk to physical address)
5. Cache lookup (tag check)
6. Determine line not in cache (need to generate BusRdX)
7. Arbitrate for bus
8. Win bus, place address, command on bus
9. All caches perform snoop (e.g., invalidate their local copies of the relevant line)
10. Another cache or memory decides it must respond (let's assume it's memory)
11. Memory request sent to memory controller
12. Memory controller is itself a scheduler
13. Memory controller checks active row in DRAM row buffer. (May need to activate new DRAM row. Let's assume it does.)
14. DRAM reads values into row buffer
15. Memory arbitrates for data bus
16. Memory wins bus
17. Memory puts data on bus
18. Requesting cache grabs data, updates cache line and tags, moves line into exclusive state
19. Processor is notified data exists
20. Instruction proceeds