Lecture 15:

A Basic Snooping-Based Multi-Processor Implementation

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2015

Tunes

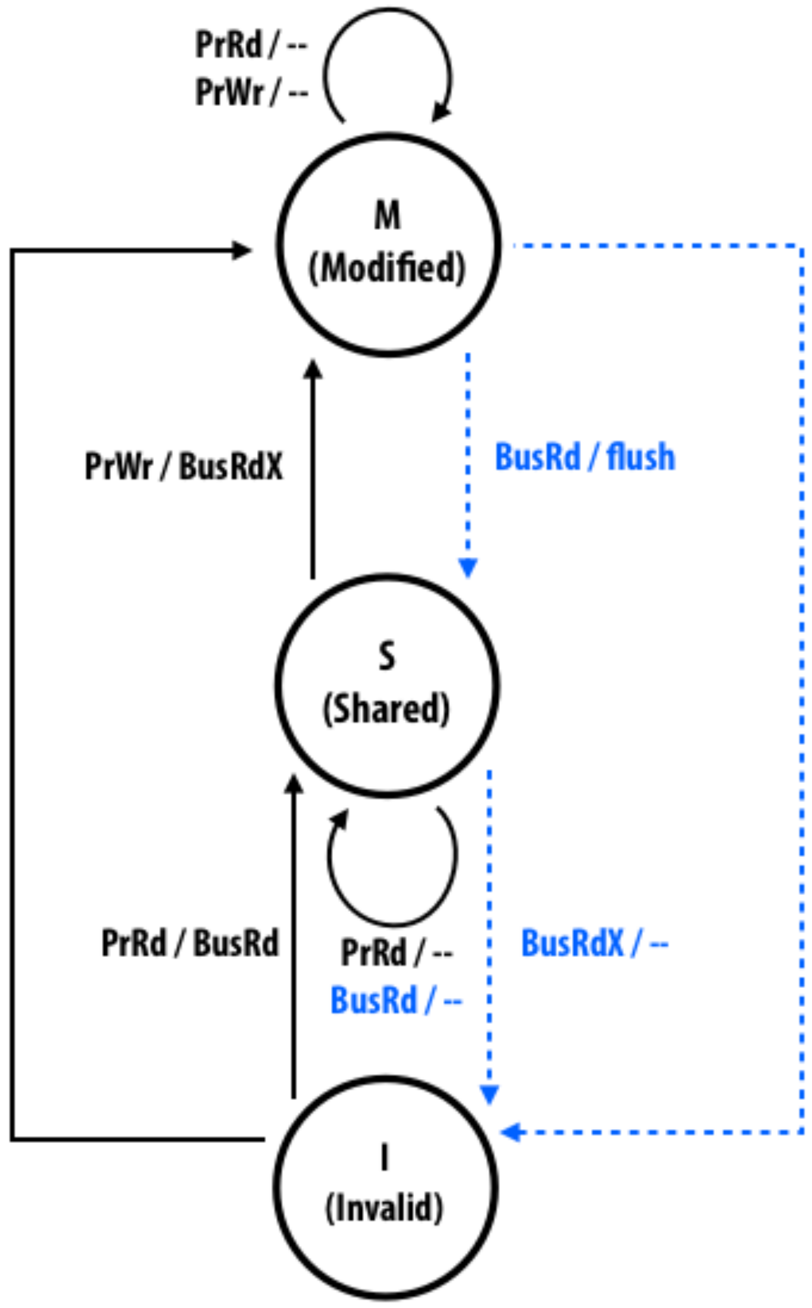
"Pushing On"
(Oliver $ & Jimi Jules)

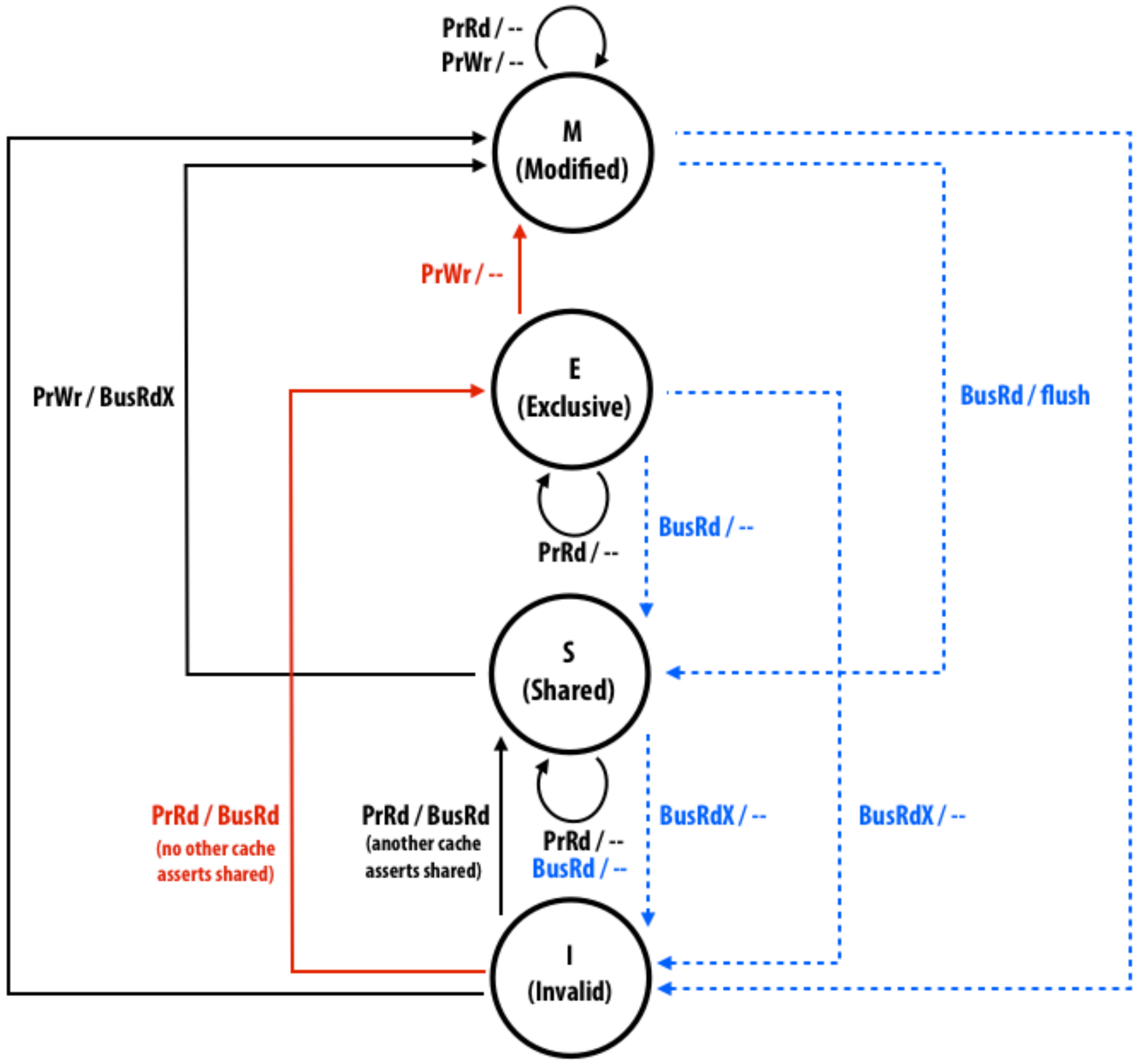
"Time for the second half of the semester."

- Oliver $

Today's topic: implementing cache coherence

- Wait... haven't we talked about this before?





- Before spring break we talked about cache coherence protocols
  - But our discussion was very abstract (a protocol is an abstraction)
    - We described what messages/transactions needed to be sent
    - We assumed messages/transactions were atomic
  - Today we will talk about efficiently implementing an invalidation-based protocol (today's point: in a real machine... efficiently ensuring coherence is complex)

The goals of our implementation

1. Be correct
   - Implements cache coherence
   - Adheres to specified memory consistency model

2. Achieve high performance

3. Minimize "cost" (e.g., minimize amount of extra hardware needed to implement coherence)

As you will see...
Techniques that pursue high performance tend to make ensuring correctness tricky.

What you should know

- **Concepts of deadlock, livelock, and starvation**

- **Have a basic understanding of how a bus works**
  - **But keep in mind most modern interconnects are NOT buses! (we'll have a whole lecture on interconnects soon)**

- **Understand why maintaining coherence is challenging to implement, even when operating under simple machine design parameters**
  - **Your mental model of hardware should be: there are many components operating in parallel**
  - **How do performance optimizations make correctness challenging? (e.g., how can deadlock, livelock, and starvation occur in coherence implementations, and how are these problems avoided)**

Terminology

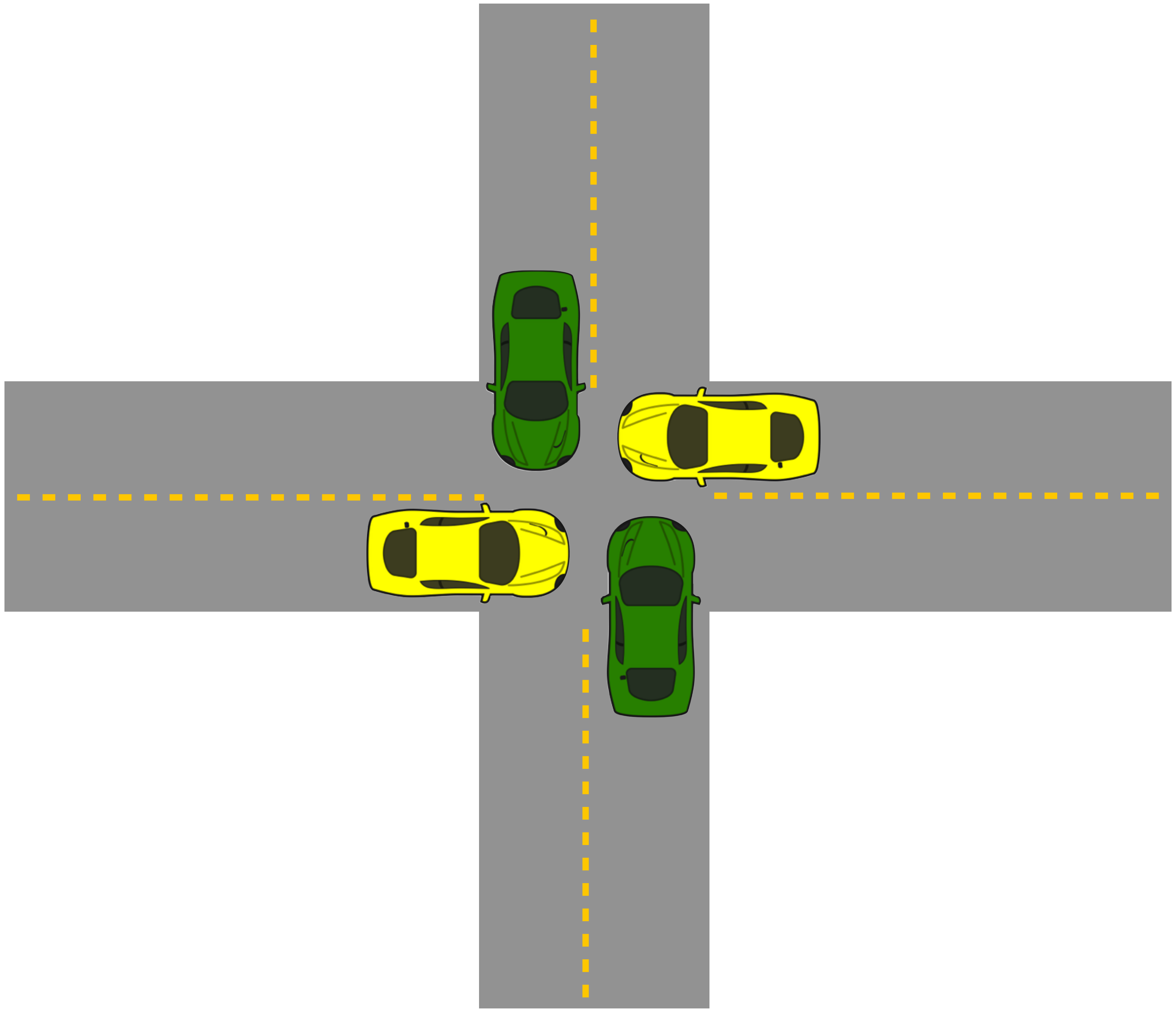
Deadlock

Livelock

Starvation

(Deadlock and livelock concern program correctness. Starvation is really an issue of fairness)

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”)

Yinzer deadlock

Non-technical side note for car-owning students:
Deadlock happens in Pittsburgh all the %$***## time

(However, deadlock can be amusing when a bus driver decides to let another driver know he has caused deadlock... "go take 418 you fool!")

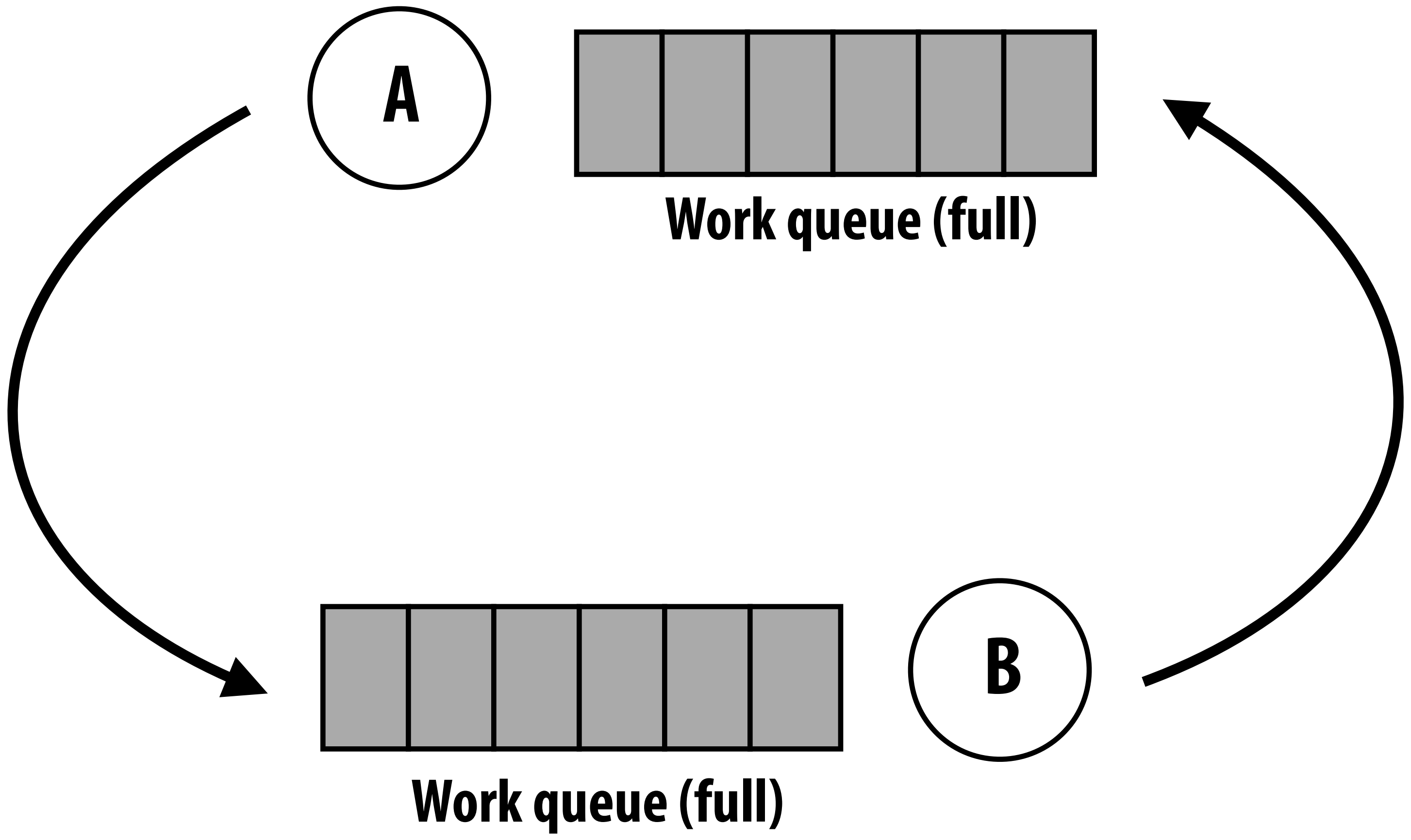
More illustrations of deadlock

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:

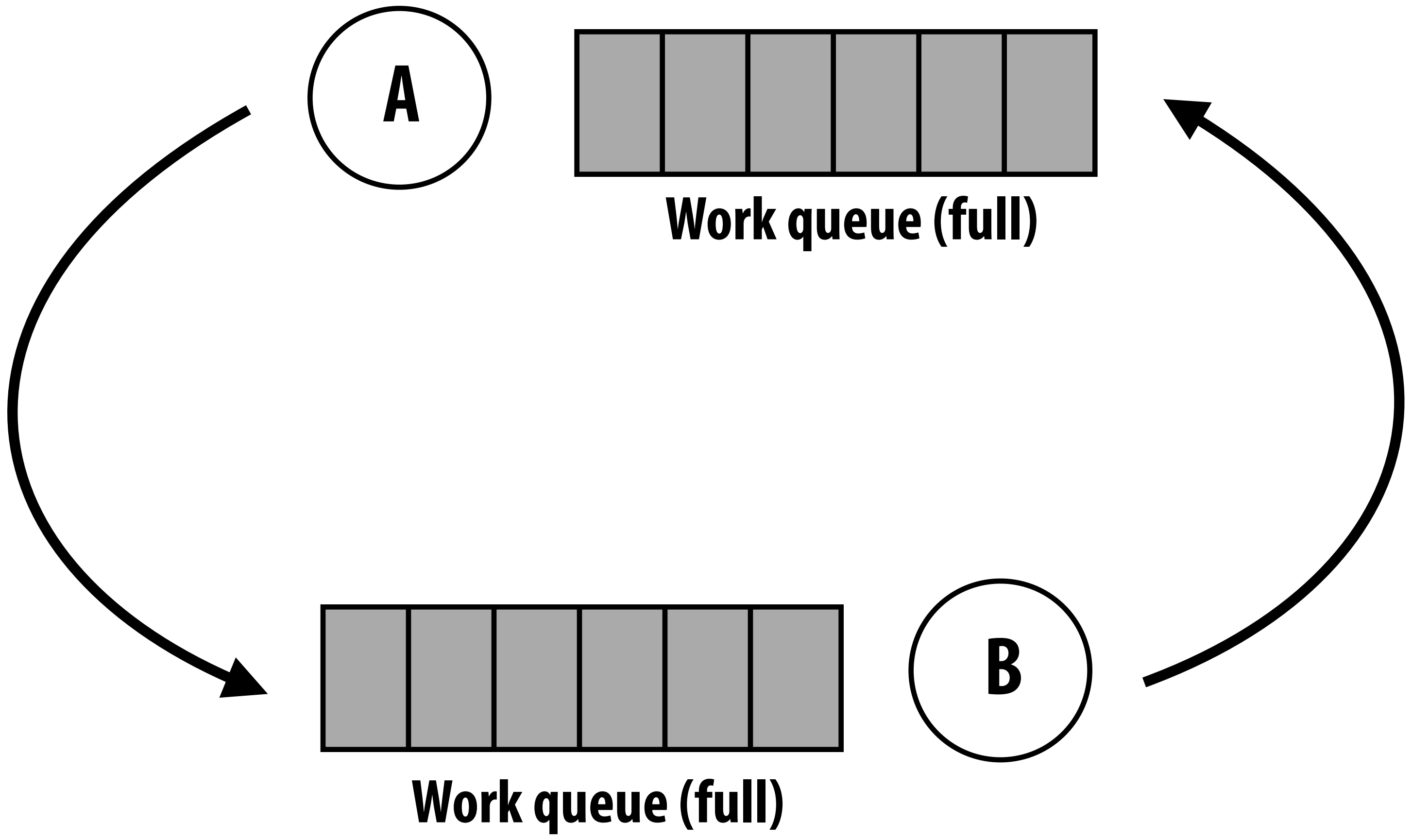
```
const int numEl = 1024;
float msgBuf1[numEl];
float msgBuf2[numEl];

int processId;
MPI_Comm_rank(MPI_COMM_WORLD, &processId);

... do work ...


MPI_Send(msgBuf1, numEl, MPI_INT, processId+1, ...
MPI_Recv(msgBuf2, numEl, MPI_INT, processId-1, ...
```

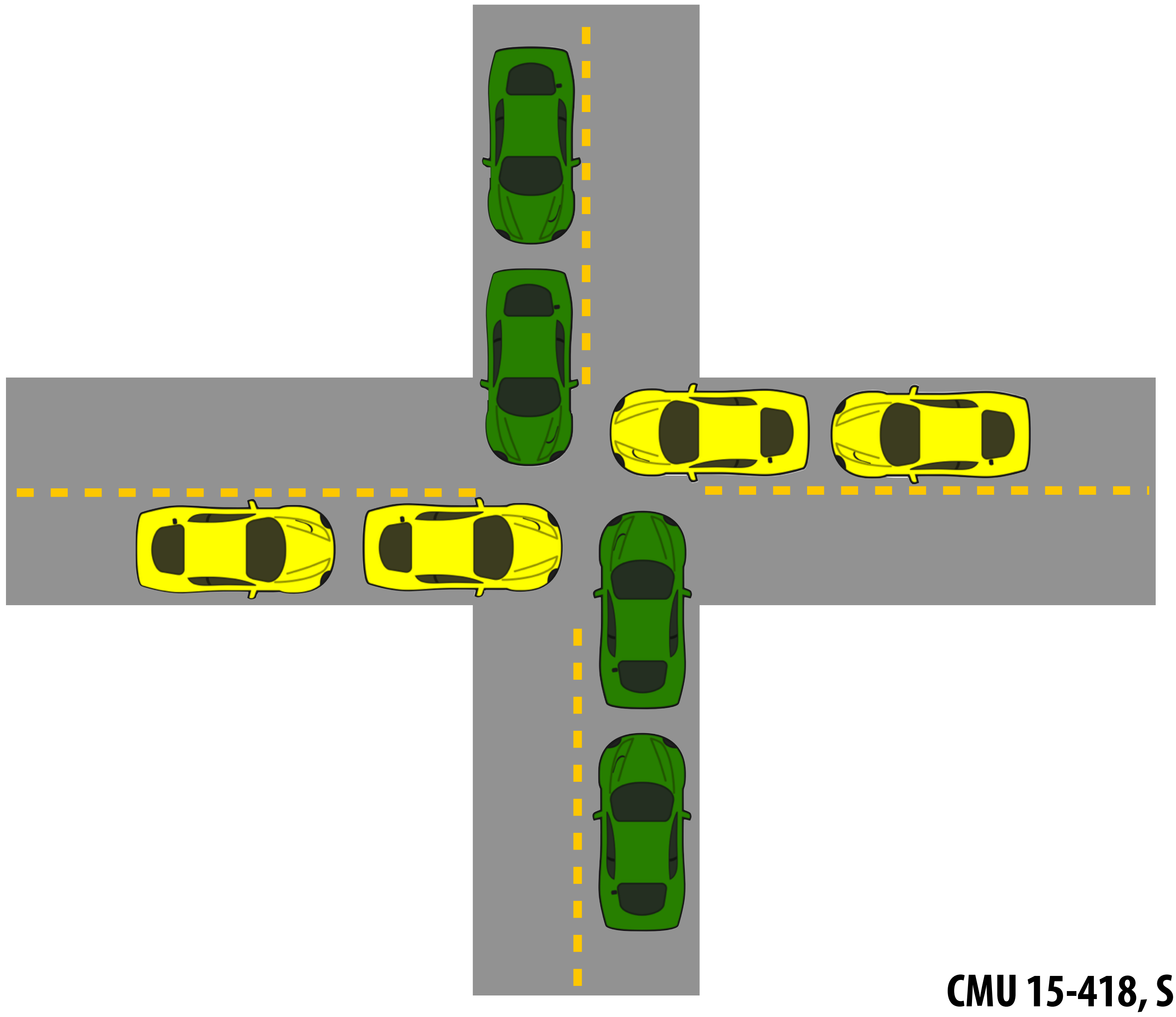
Every process sends a message (blocking send) to the processor with the next higher id

Then receives message from processor with next lower id.

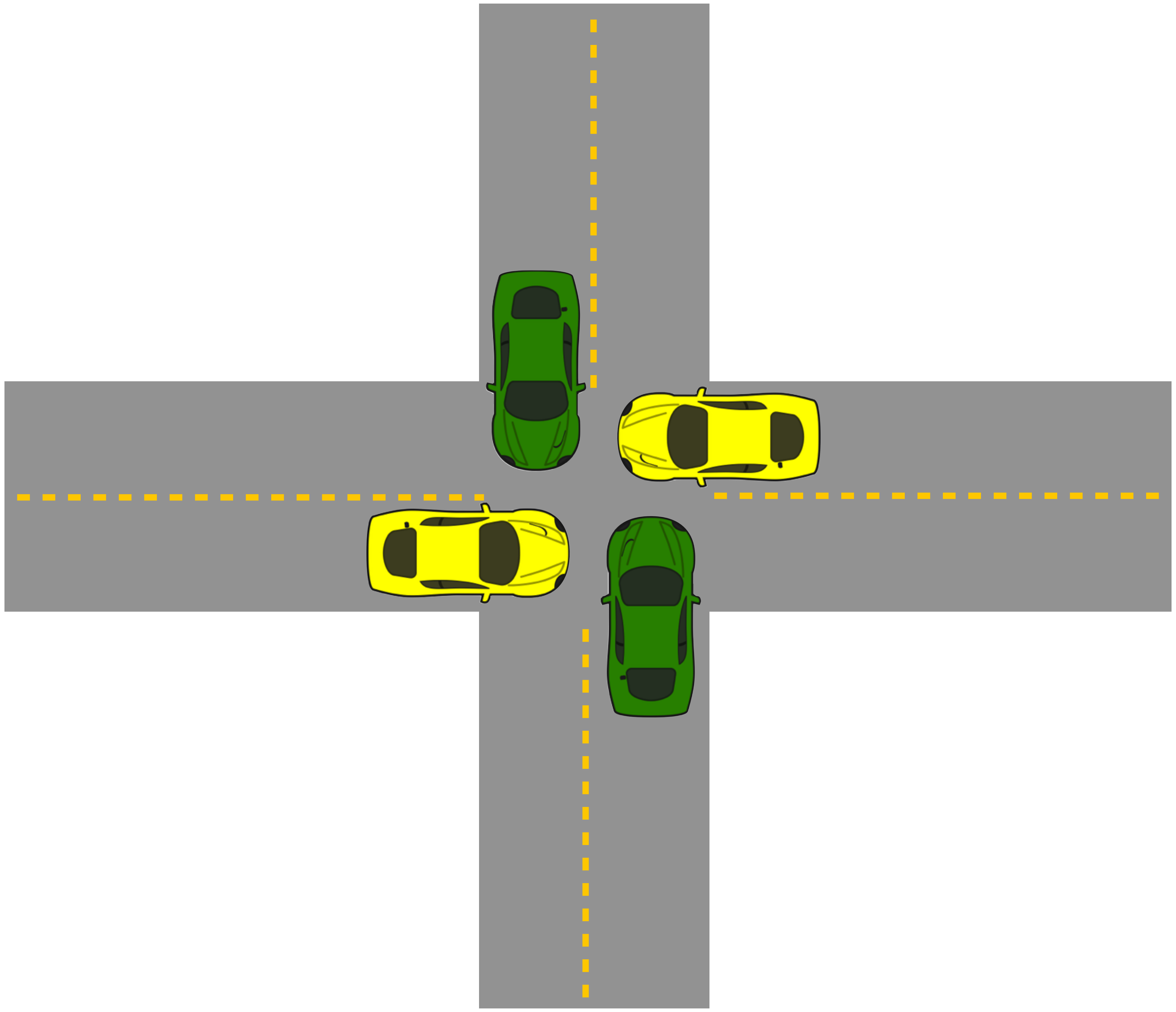
Required conditions for deadlock

1. Mutual exclusion: one processor can hold a given resource at once
2. Hold and wait: processor must hold the resource while waiting for other resources needed to complete an operation
3. No preemption: processors don't 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)

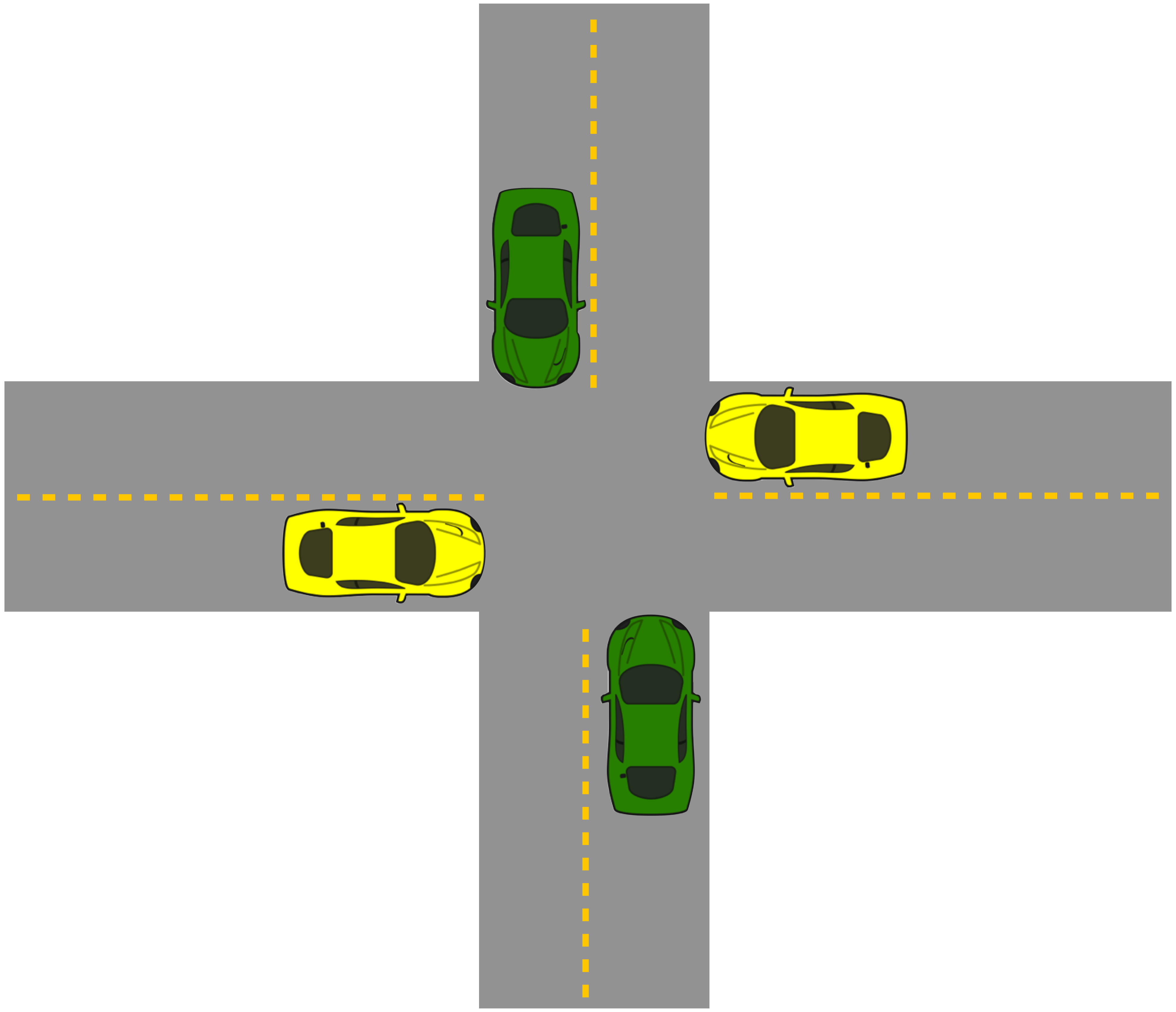




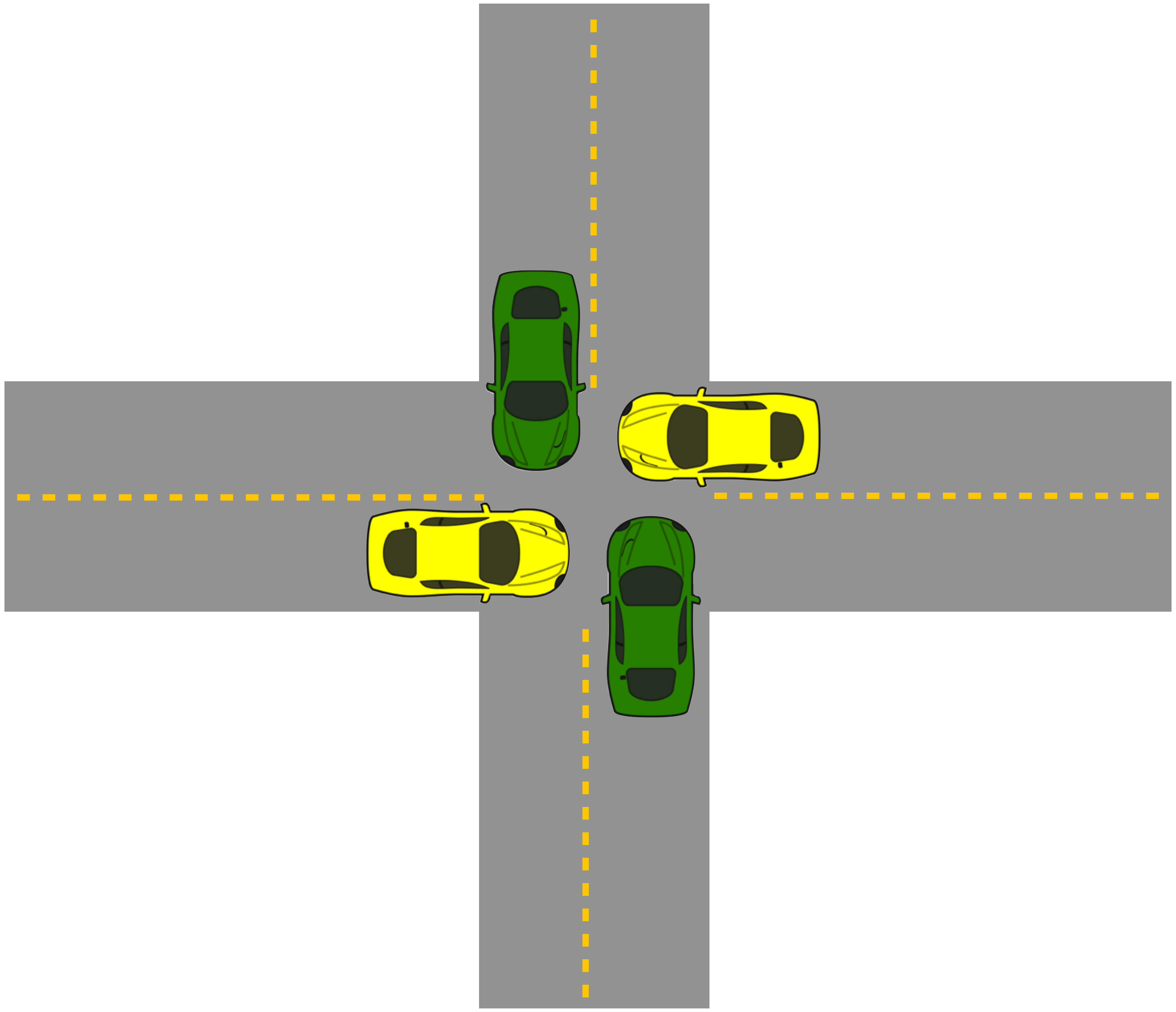
Livelock



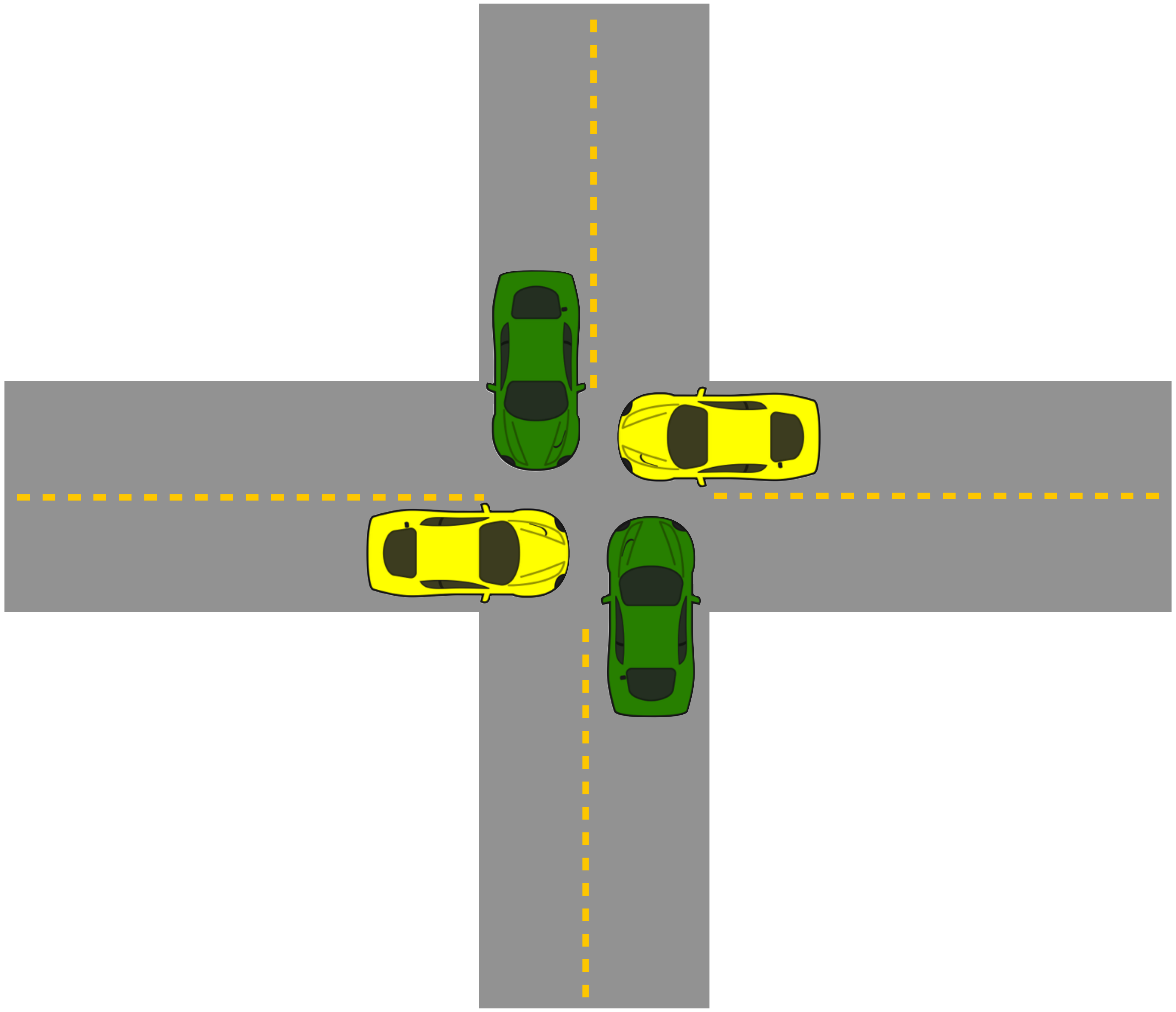
Livelock



Livelock



Livelock



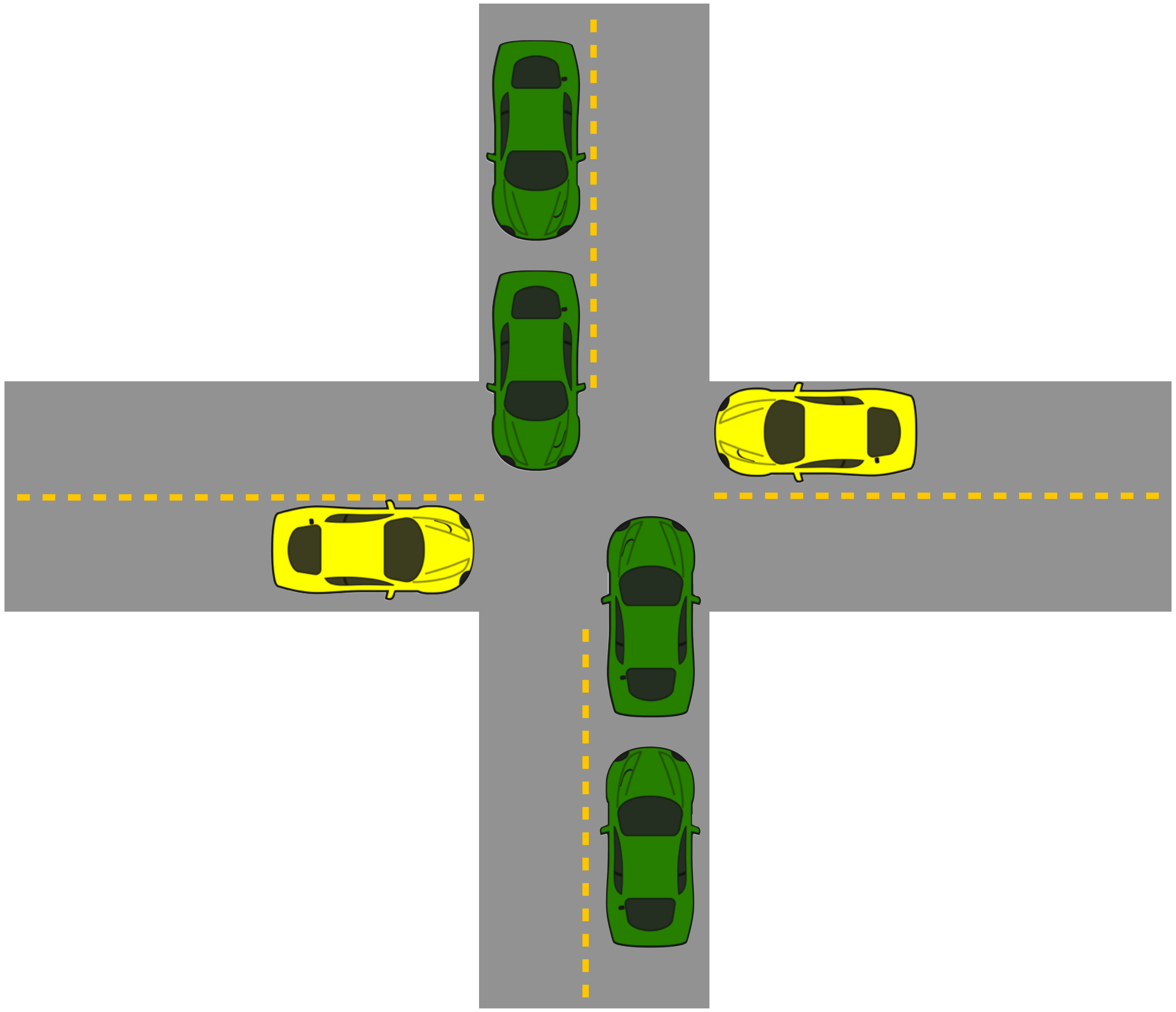
Livelock is a state where a system is executing many operations, but no thread is making meaningful progress.

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

Computer system examples:

Operations continually abort and retry

Starvation



State where a system is making overall progress, but some processes make no progress.

(green cars make progress, but yellow cars are stopped)

Starvation is usually not a permanent state

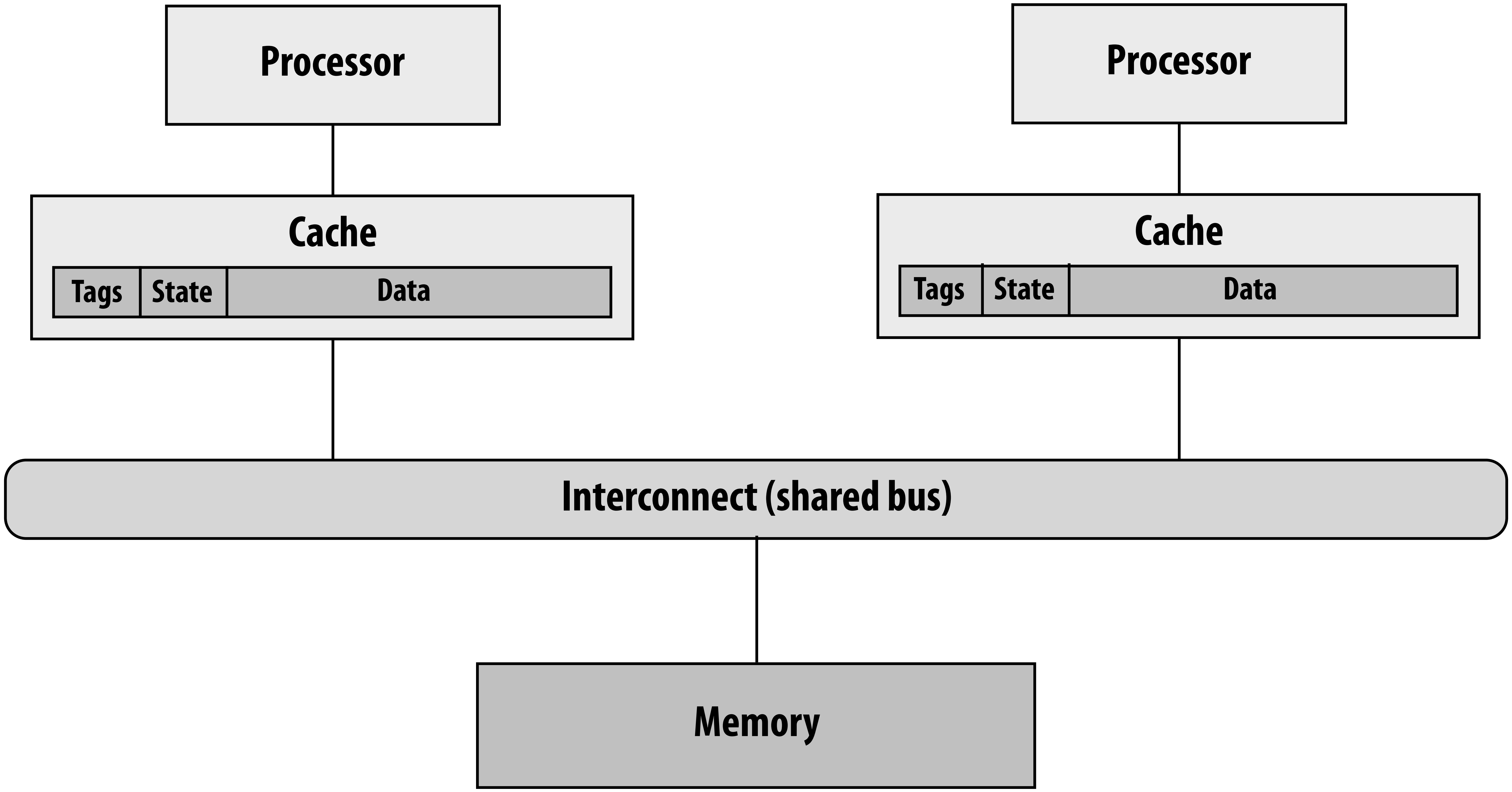
(as soon as green cars pass, yellow cars can go)

In this example: assume traffic moving left/right (yellow cars) must yield to traffic moving up/down (green cars)

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
- System interconnect is an atomic shared bus
- Cache can stall processor as it is carrying out coherence operations



Cache miss logic on a uniprocessor

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

   [Assume no matching tags, must read data from memory]

3. Assert request for bus
4. Wait for bus grant (as determined by bus arbitrator)
5. Send address + command on bus
6. Wait for command to be accepted
7. Receive data on bus

Address
Data

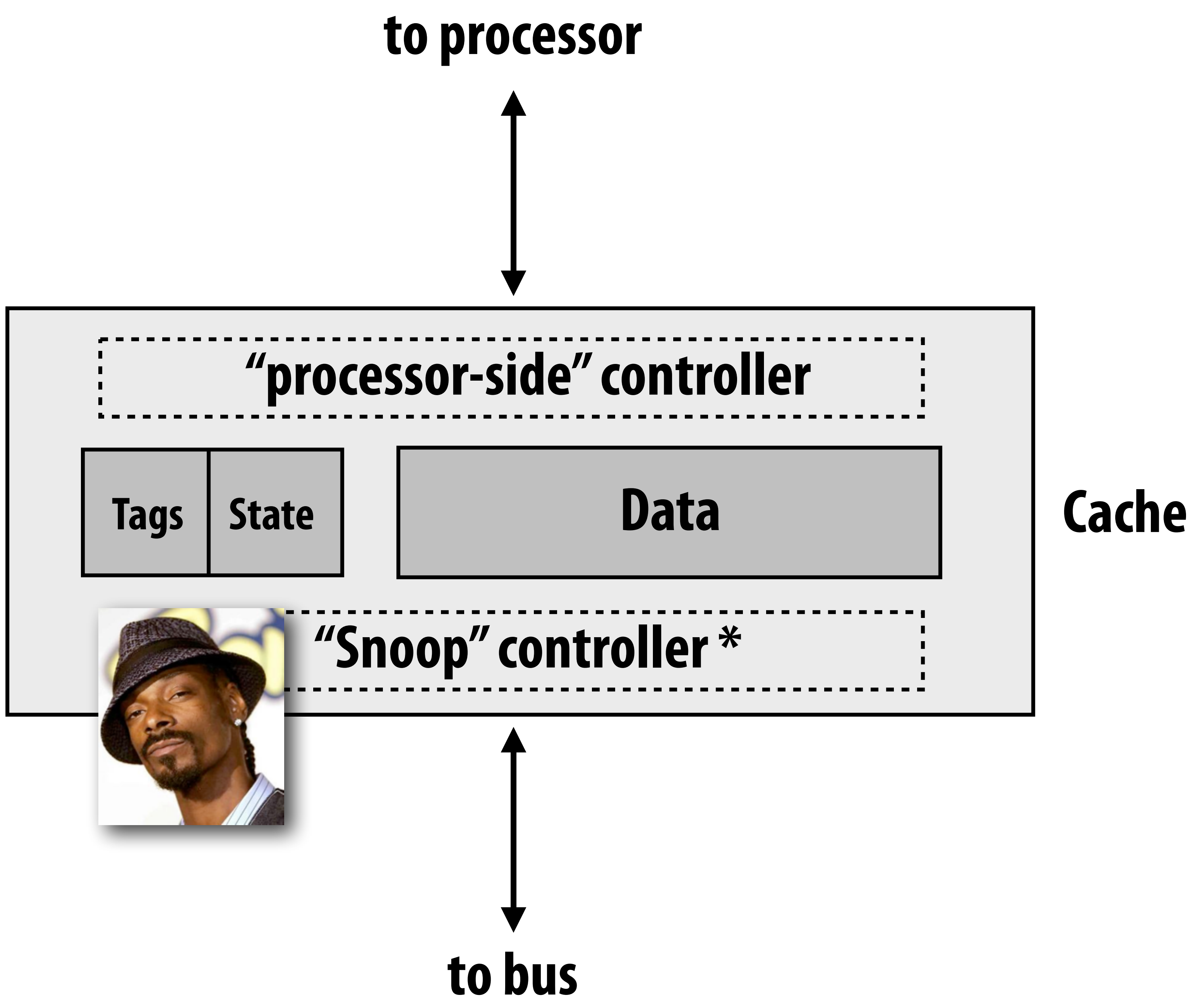
What does atomic bus mean in a multi-processor scenario?

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

Multi-processor cache controller behavior

Challenge: both requests from processor and bus require tag lookup

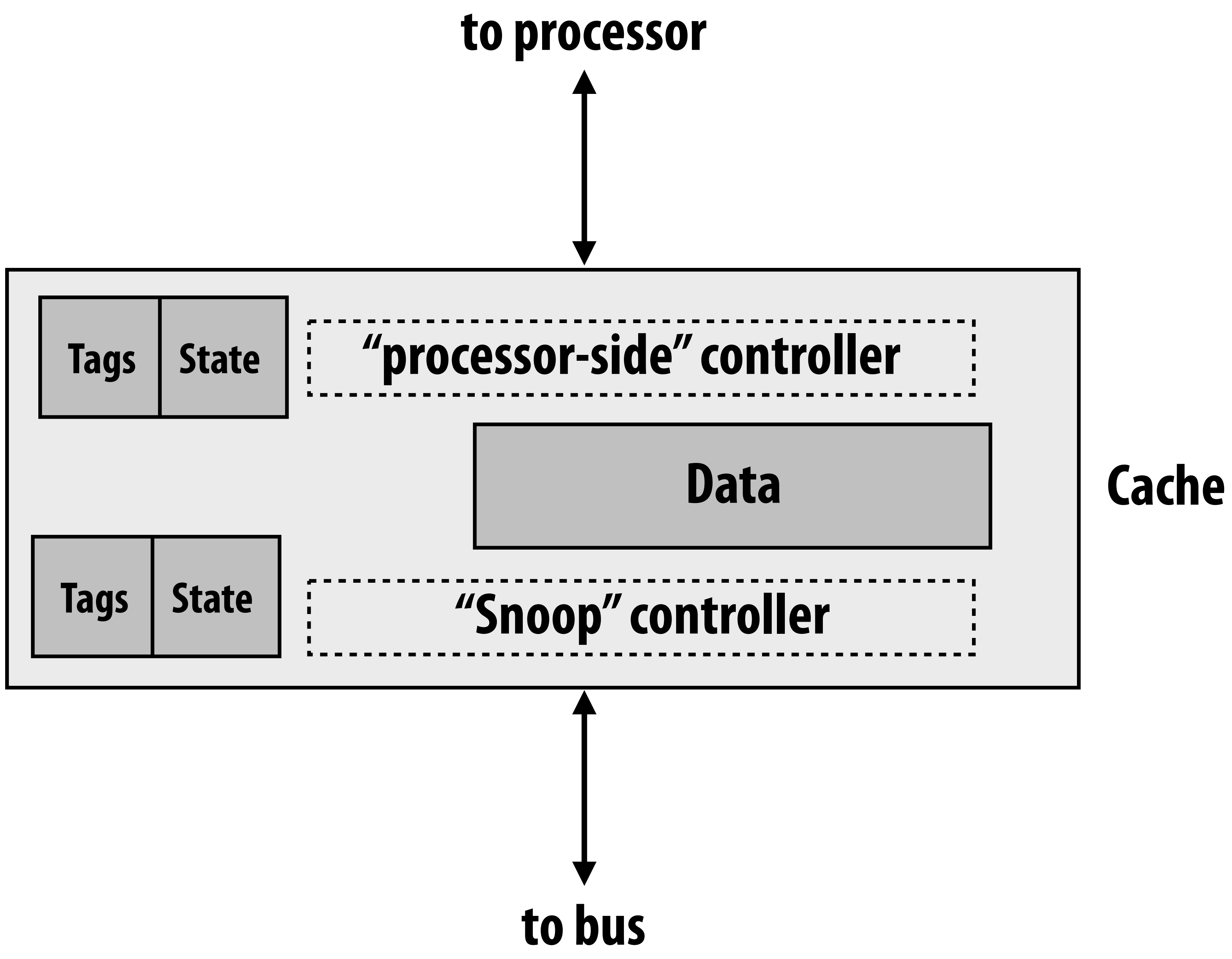


If bus receives priority:
During bus transaction, processor is locked out from its own cache.

If processor receives priority:
During processor cache accesses, cache cannot respond with it's snoop result (so it delays other processors even if no sharing of any form is present)

* Snoop controller has its mind on the bus and the bus on its mind

Allowing simultaneous access by processor-side and snoop controllers



Option 1: cache duplicate tags

Option 2: multi-ported tag memory

Note: tags must stay in sync for correctness, so tag update by one controller will still need to block the other controller (but modifying tags is infrequent compared to checking them)

Keep in mind: in either case cost of the additional performance is additional hardware resources.

Reporting snoop results protocol in MESI

- Assume a cache read miss (BusRd)
- Collective response of caches must appear on bus
  - Is line dirty? If so, memory should not respond
  - Is line shared? If so, cache should load into S state, not E

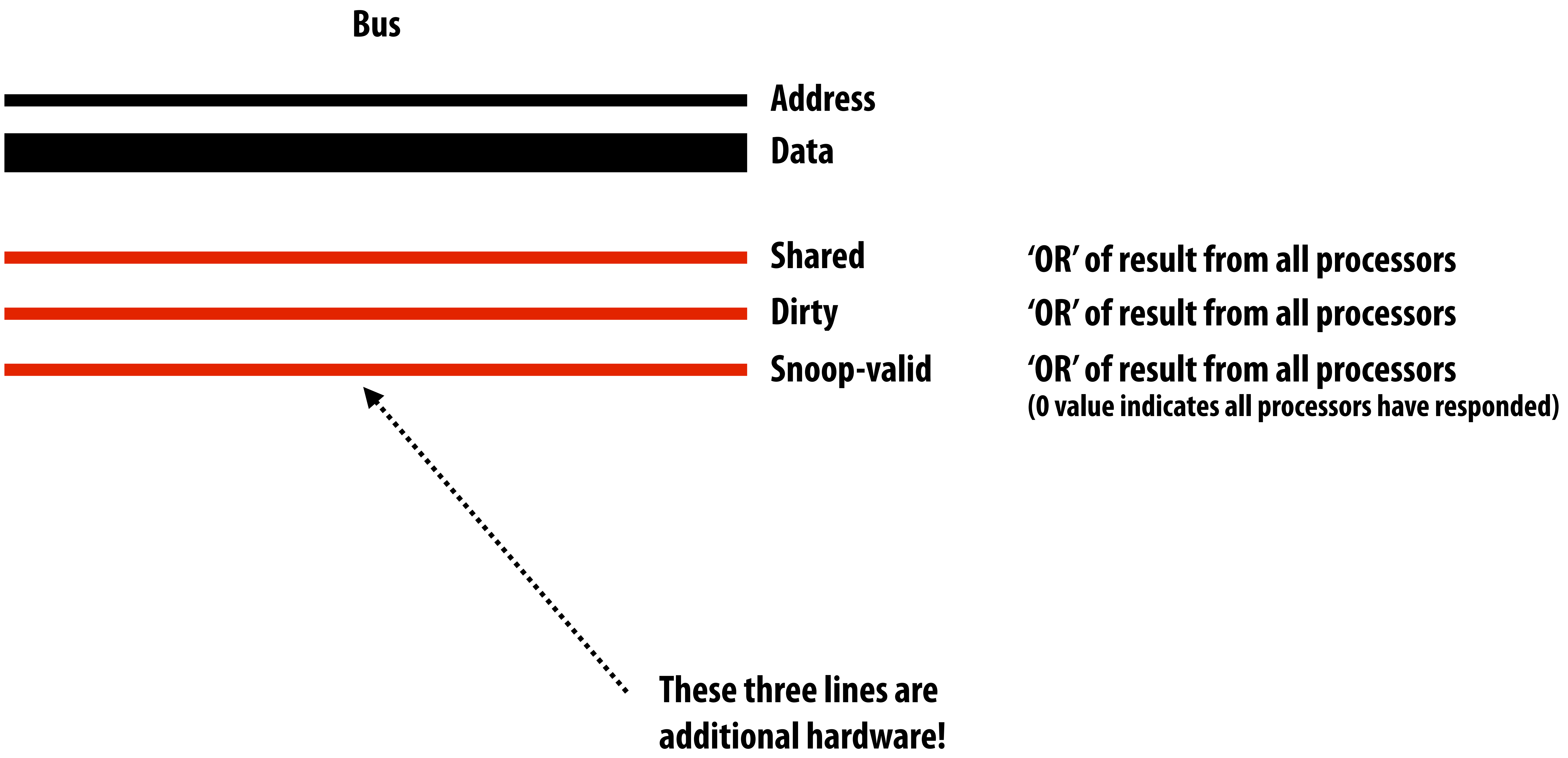
Memory needs to know what to do

Loading cache needs to know what to do

How are snoop results communicated?

When are snoop results communicated?

Reporting snoop results: how


Bus
Address
Data
Shared
Dirty
Snoop-valid
'OR' of result from all processors
'OR' of result from all processors
'OR' of result from all processors
(0 value indicates all processors have responded)
These three lines are additional hardware!

Reporting snoop results: when

Mainly an issue of when memory should react to the BusRd request

- **Option 1: memory responds in a fixed number of clocks after address appears on bus (delay set by worst case scenario for cache to respond)**
    - Design of caches guarantees they can respond with their snoop results in a fixed number of clocks
    - Note: importance of duplicated tags (to meet guarantee)

- **Option 2: variable delay**
    - Memory assumes one of the caches will service request until snoop results are valid (if no dirty bit set, then memory must respond)
    - More complex logic, but lower latency if snoops are completed quickly

Handling write backs

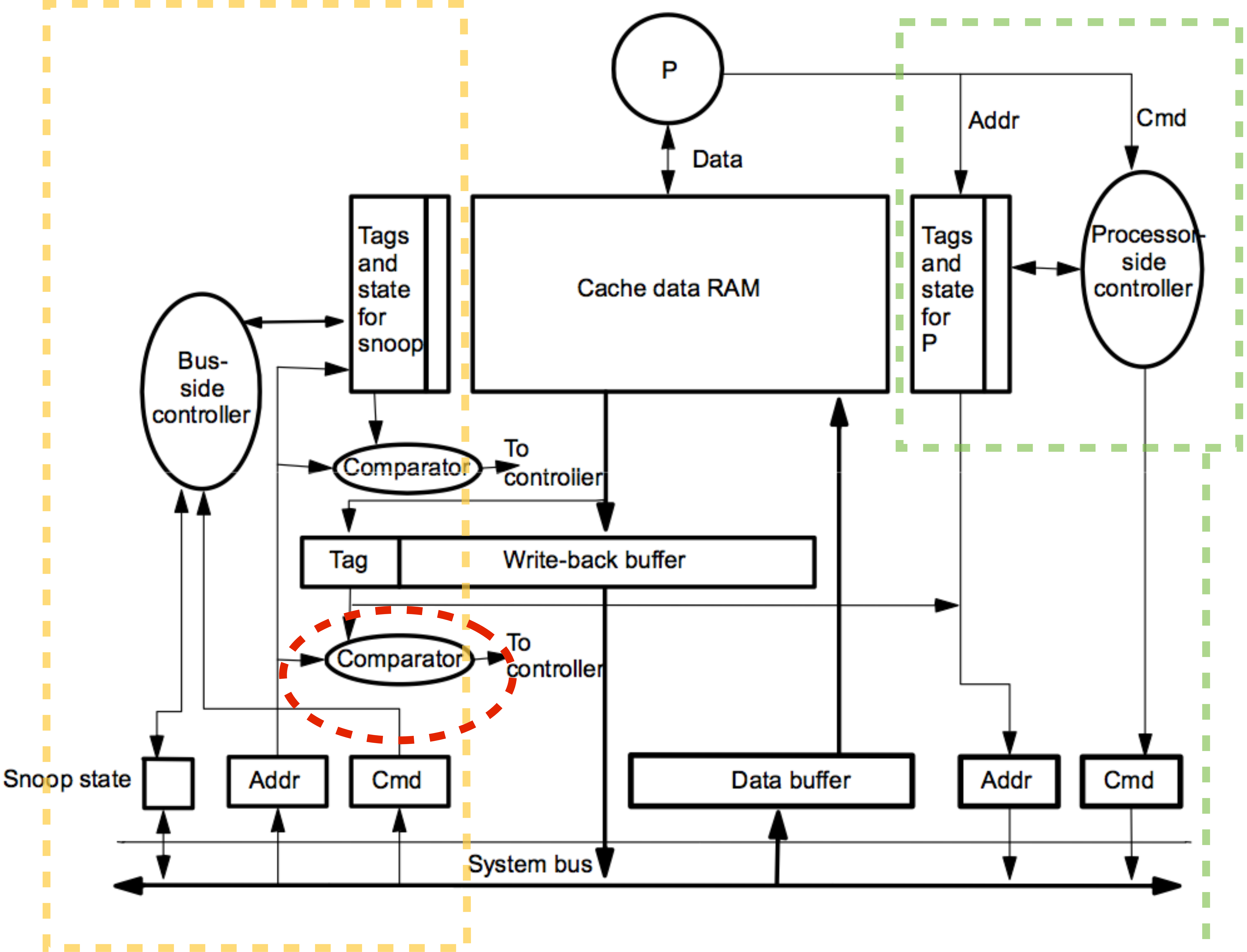
- **Write backs involve two bus transactions**
  1. **Incoming line (line requested by processor)**
  2. **Outgoing line (evicted dirty line in cache that must be flushed)**

- **Ideally would like the processor to continue as soon as possible (it shouldn’t have to wait for the flush to complete)**

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

*** Do not confuse a cache’s write-back buffer (discussed here) with the processor’s write buffer (discussed in the memory consistency lecture). Both buffers exist to hide the latency of memory operations. However, the write buffer holds writes that have been issued by the processor, but not completed. The write-back buffer holds dirty cache lines that must be flushed to memory so memory stays up to date. The lines are dirty because there was some write to them completed by the processor a long time ago. (This is a good distinction to discuss in comments.)**

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)

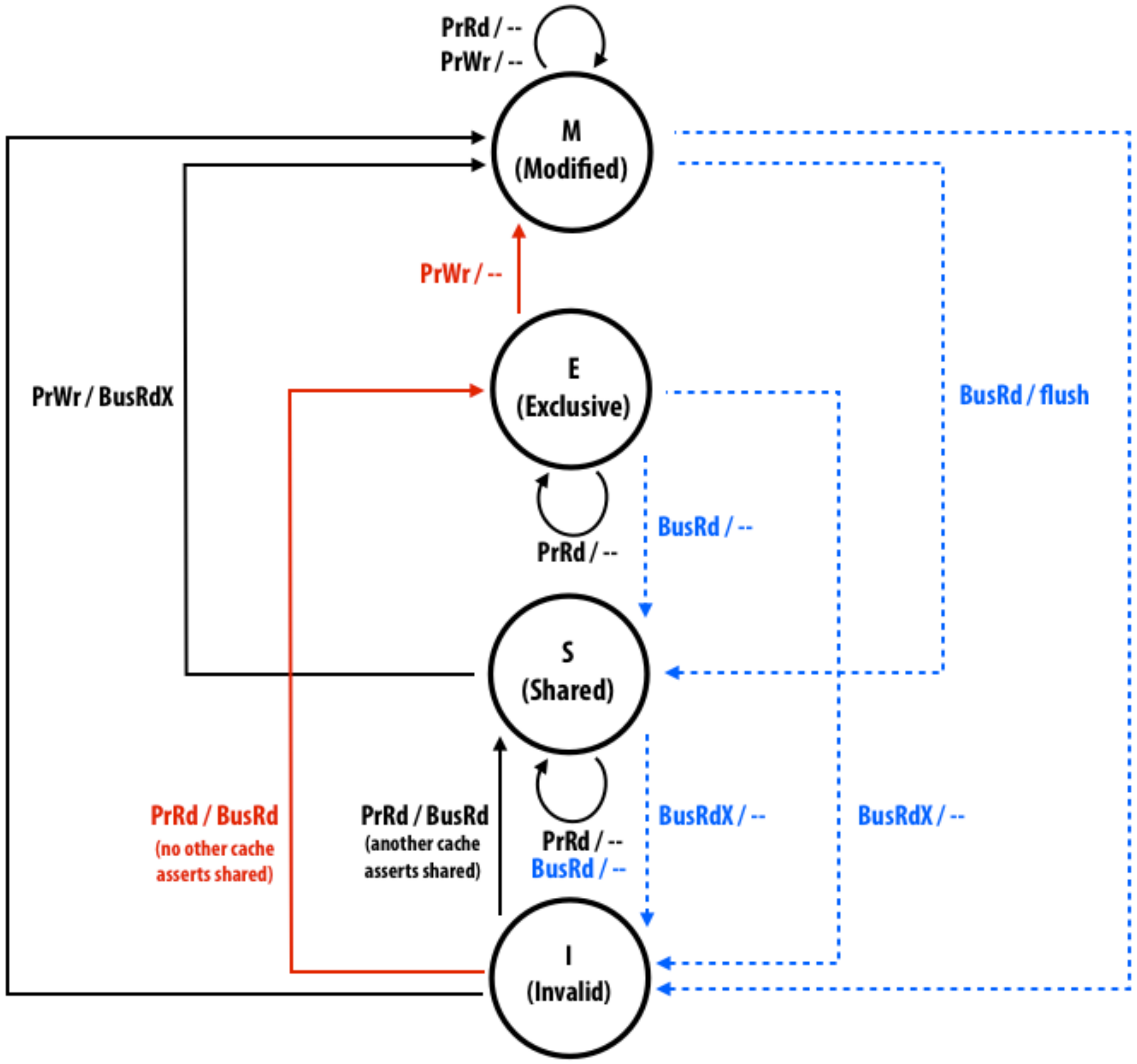
these hardware components handle snooping related tasks.

these hardware components handle processor-related requests

Figure credit: Culler, Singh, and Gupta

Non-atomic state transitions

- Coherence protocol state transition diagrams (like the one below) assumed that transitions between states were atomic
- We've assumed the bus transaction itself is atomic, but <u>all</u> the operations the system performs as a result of a memory operation are not
  - e.g., look up tags, arbitrate for bus, wait for actions by other controllers, ...
- Must be careful to handle race conditions appropriately



An example race condition

Processors P1 and P2 write to valid (and shared) cache line A simultaneously
(both need to issue BusUpg to move line from S state to M state)

P1 “wins” bus access (as determined by arbiter), P1 sends BusUpg

P2 is waiting for bus access (to send its own BusUpg), can’t proceed because P1 has bus

P2 receives BusUpg, must invalidate line A (as per MESI protocol)

P2 must also change its pending BusUpg request to a BusRdX

Cache must be able to handle requests while waiting to acquire bus AND be able to modify its own outstanding requests

Maintaining write serialization

- Consider this tempting optimization: on processor write, update cache line, allow processor to proceed prior to sending BusRdX (or BusUpg) transaction out to bus (to obtain exclusive access)

- This violates coherence. Why?
  - Why does a write-back buffer not cause this problem? (it sounds like a similar optimization, right?)

- To ensure write serialization, cache cannot allow processor to proceed until read-exclusive transaction appears on bus
  - At this point, the write is "committed"
  - Key idea: order of transactions on the bus defines the global order of writes in the parallel program
    - THIS MAINTAINS WRITE SERIALIZATION!

Fetch deadlock

P1 has a modified copy of cache line B

P1 is waiting for the bus to issue BusRdX on cache line A

BusRd for B appears on bus while P1 is waiting

To avoid deadlock, P1 must be able to service incoming transactions while waiting to issue requests

Livelock

Two processors writing to cache line B

P1 acquires bus, issues BusRdX

P2 invalidates

Before P1 performs cache line update, P2 acquires bus, issues BusRdX

P1 invalidates

and so on...

To avoid livelock, a write that obtains exclusive ownership must be allowed to complete before exclusive ownership is relinquished.

Starvation

- Multiple processors competing for bus access
  - Must be careful to avoid (or minimize likelihood of) starvation
- Example policies:
  - FIFO arbitration
  - Priority-based heuristics (frequent bus users have priority drop)

Design issues

- **Design of cache controller and tags**

  (to support access from processor and bus)
- **How and when to present snoop results on bus**
- **Dealing with write backs**
- **Dealing with non-atomic state transitions**
- **Avoiding deadlock, livelock, starvation**

These issues arose even though we only implemented a few optimizations on a very basic invalidation-based, write-back system!

(atomic bus, one outstanding memory request per processor, single-level caches)

First-half summary: parallelism and concurrency in coherence implementation are sources of complexity

- **Processor, cache, and bus all are resources operating in parallel**
  - **Often contending for shared resources:**
    - **Processor and bus contending for cache**
    - **Caches contending for bus access**

- **"Memory operations" that are abstracted by the architecture as atomic (e.g., loads, stores) are implemented via multiple transactions involving all of these hardware components**

- **Performance optimization often entails splitting operations into several, smaller transactions**
  - **Splitting work into smaller transactions reveals more parallelism (recall pipelining example)**
  - **Cost: more hardware needed to exploit additional parallelism**
  - **Cost: more care needed to ensure abstractions still hold (the machine is correct)**

Part 2:

Building the system around non-atomic bus transactions.

What you should know

- What is the major performance issue with atomic bus transactions that motivates moving to a more complex non-atomic system?

- You should know the main components of a split-transaction bus, and how transactions are split into requests and responses

- How deadlock and livelock might occur in both atomic bus and non-atomic bus-based systems (what are possible solutions for avoiding it?)

- The role of queues in a parallel system (today is yet another example)

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)

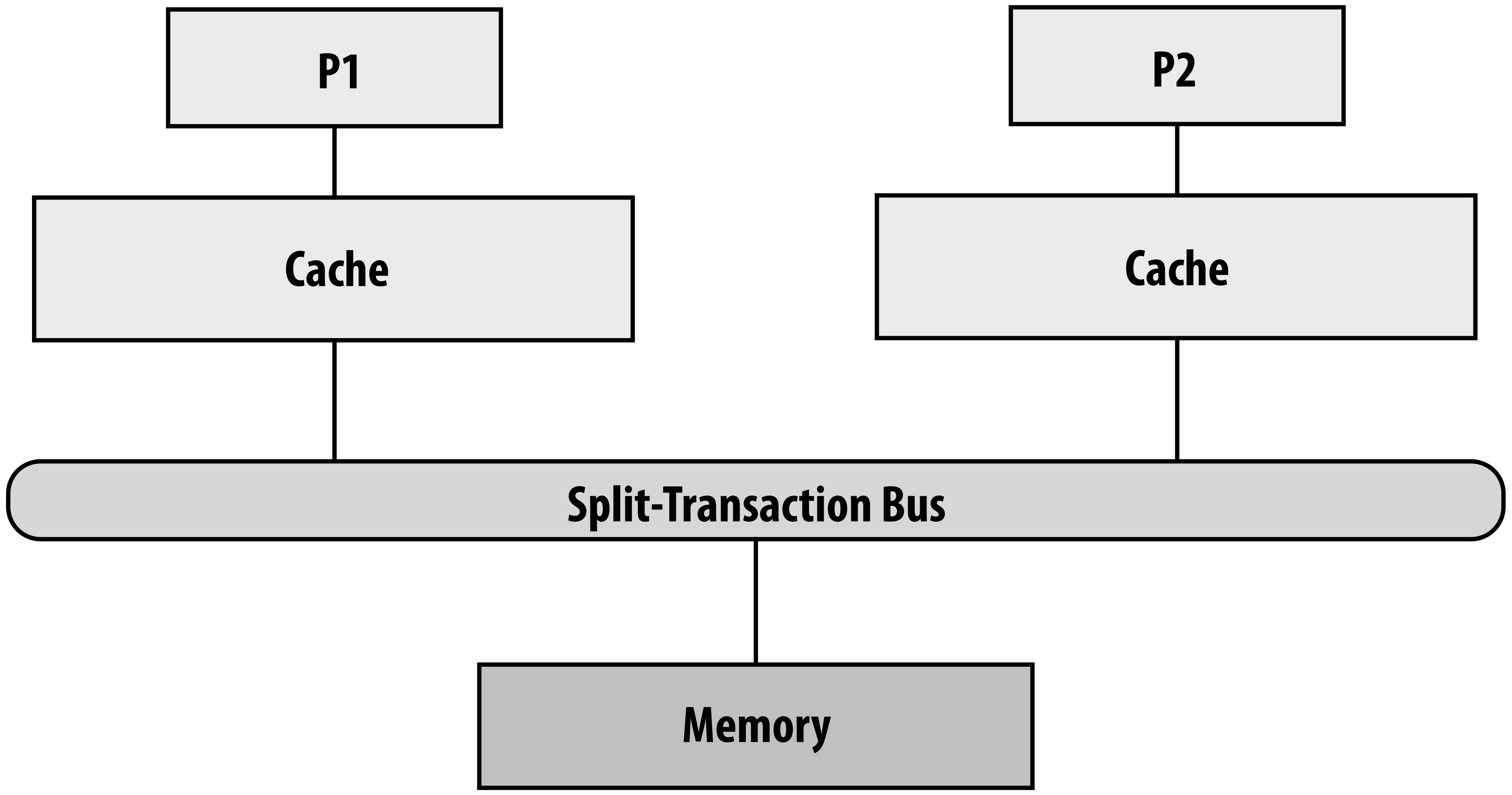
3. Response to command by another bus client placed on bus
4. Next client obtains bus access (arbitration)

Split-transaction bus

Bus transactions are split into two transactions:

1. The request
2. The response

Other transactions can intervene 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 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

New issues arise due to split transactions

1. **How to match requests with responses?**

2. **How to handle conflicting requests on bus?**

   Consider:

   - **P1 has outstanding request for line A**
   - **Before response to P1 occurs, P2 makes request for line A**

3. **Flow control: how many requests can be outstanding at a time, and what should be done when buffers fill up?**

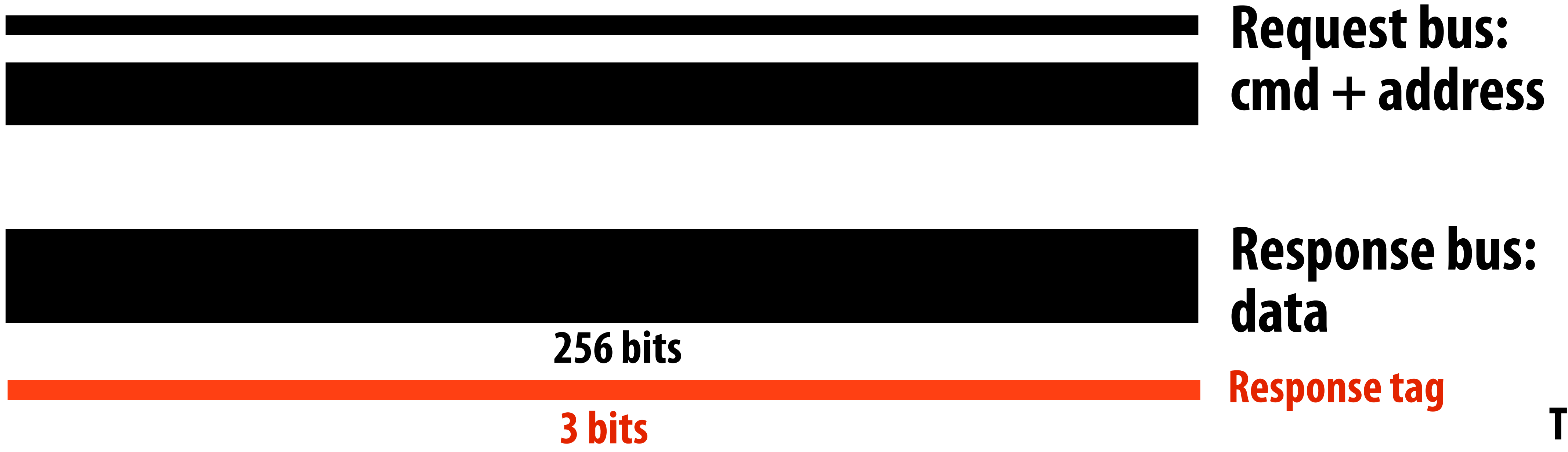
4. **When are snoop results reported? During the request? or during the response?**

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)
  - When a buffer is full, client can NACK a transaction, causing a retry

Initiating a request

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



Request bus:
cmd + address

Response bus:
data

256 bits

Response tag

3 bits

Request Table
(assume a copy of this table is maintained by each bus client: e.g., cache)

Transaction tag is just the index into the request table →

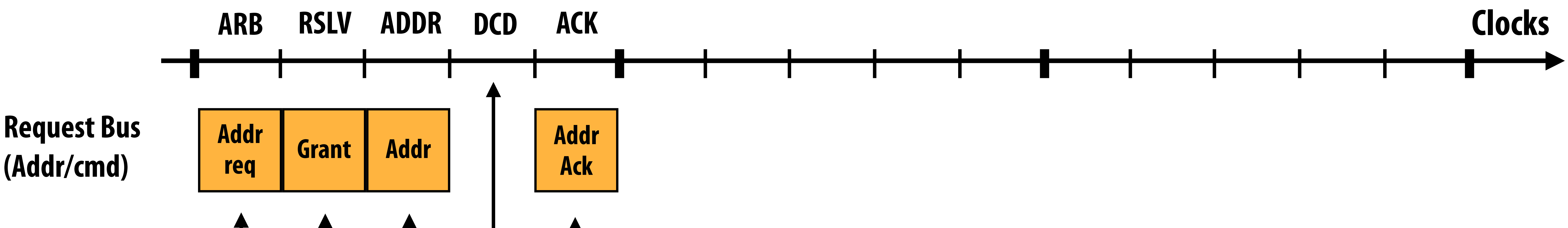
Requestor	Addr	State
P0	0xbeef	

Step 1: Requestor asks for request bus access

Step 2: Bus arbiter grants access, assigns transaction a tag

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., raise inhibit wire)

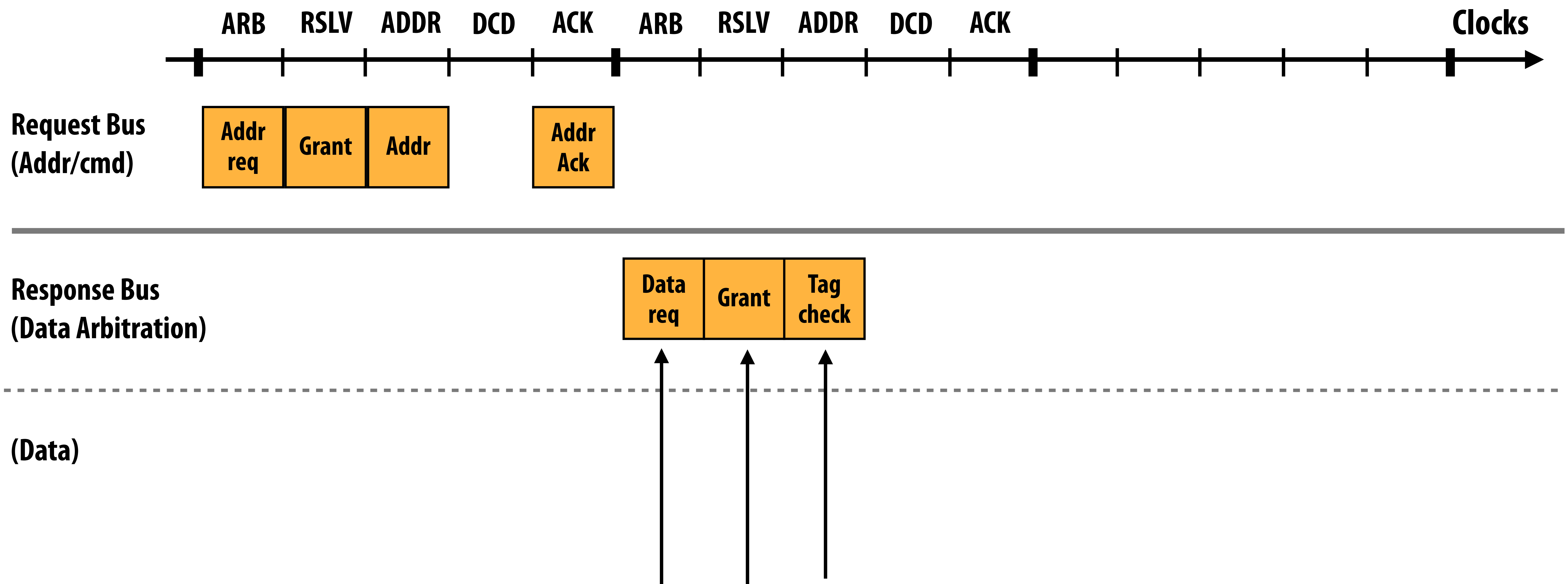
Caches perform snoop: look up tags, update cache state, etc.
Memory operation commits here!
(NO BUS TRAFFIC)

Bus "winner" places command/address on the bus

Request resolution: address bus arbiter grants access to one of the requestors
Request table entry allocated for request (see previous slide)
Special arbitration lines indicate tag assigned to request

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

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

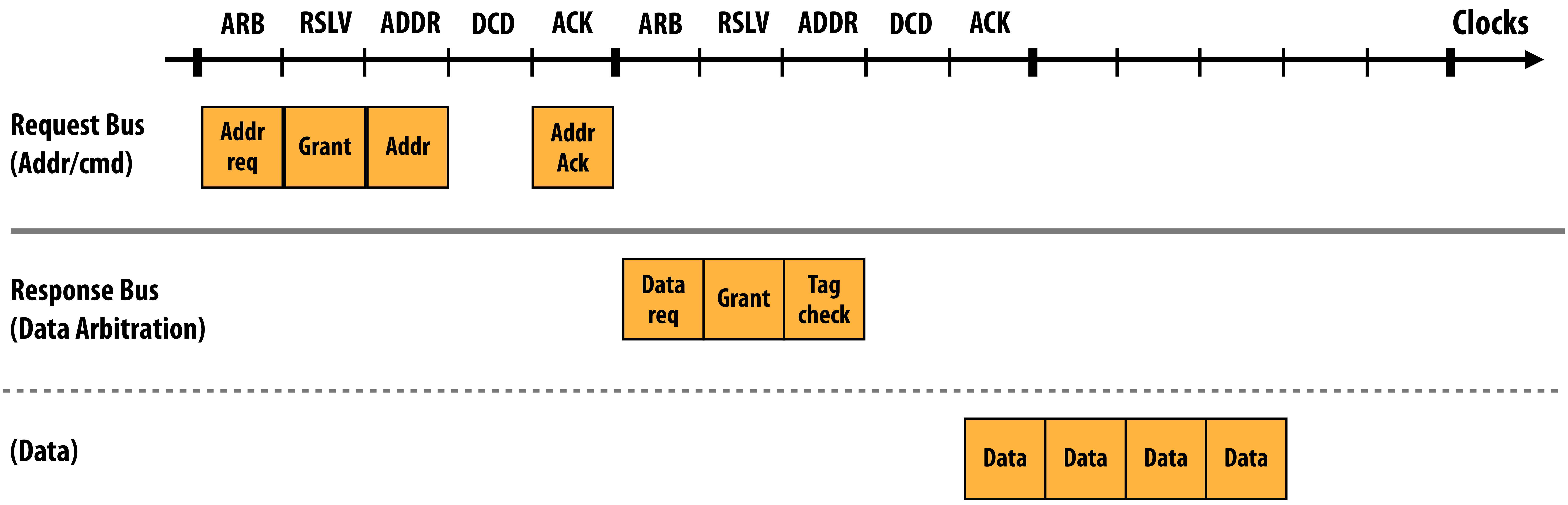


Original requestor signals readiness to receive response
(or lack thereof: requestor may be busy at this time)

Data bus arbiter grants one responder bus access

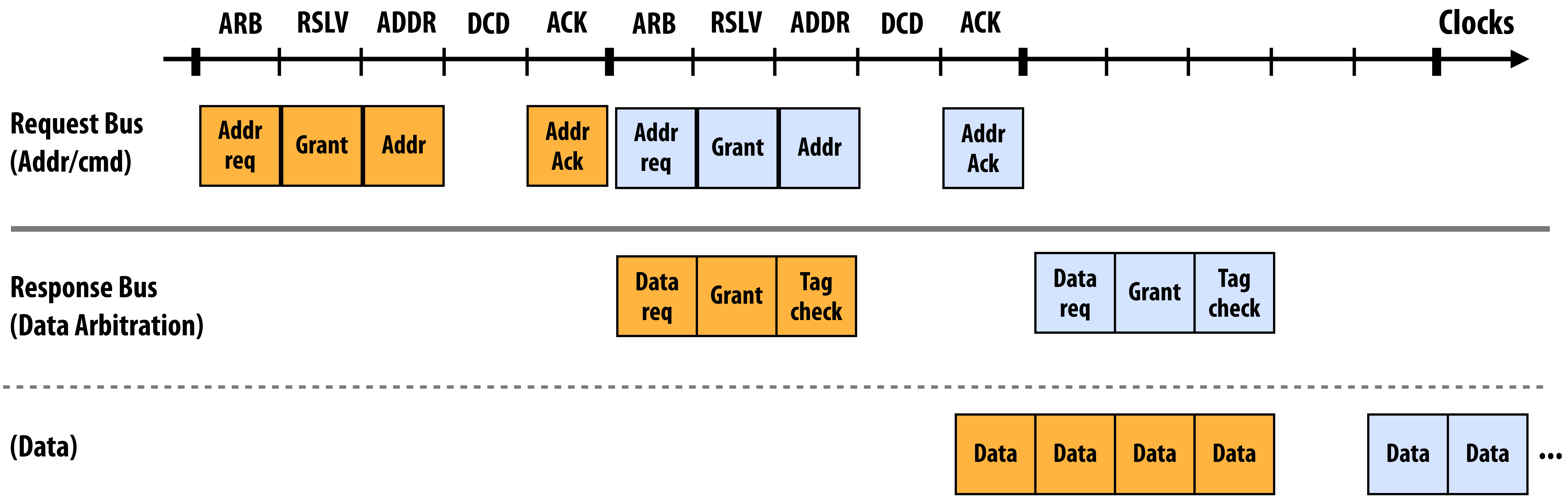
Data response arbitration: responder presents intent to respond to request with tag T
(many caches --or memory-- may be doing so in the same cycle)

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 128 byte cache lines → 4 cycles on 256 bit bus

Pipelined transactions



ARB RSLV ADDR DCD ACK ARB RSLV ADDR DCD ACK

Clocks

Request Bus (Addr/cmd)

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

Response Bus (Data Arbitration)

Data req | Grant | Tag check | Data req | Grant | Tag check

(Data)

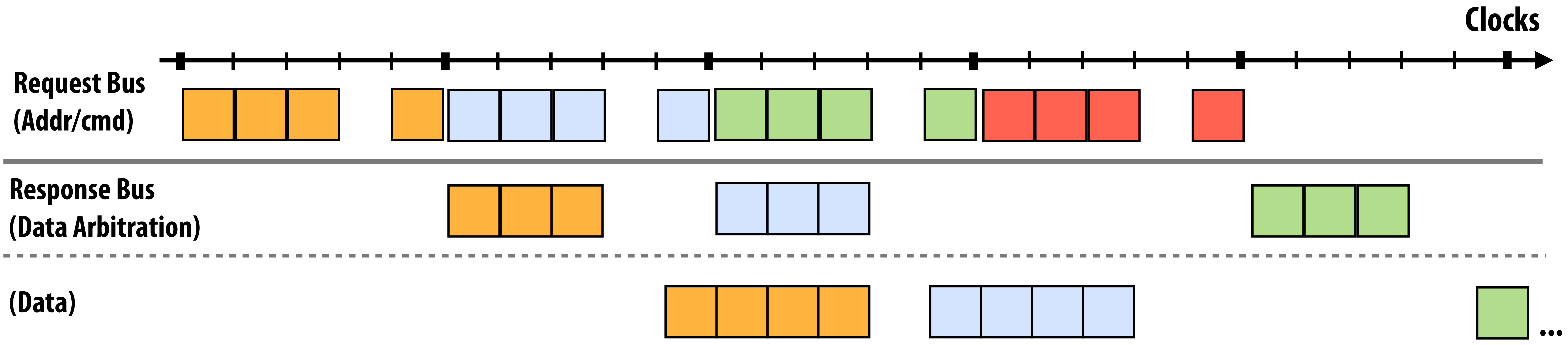
Data | Data | Data | Data | Data | Data ...

🟧 = 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 compl

Key issues to resolve

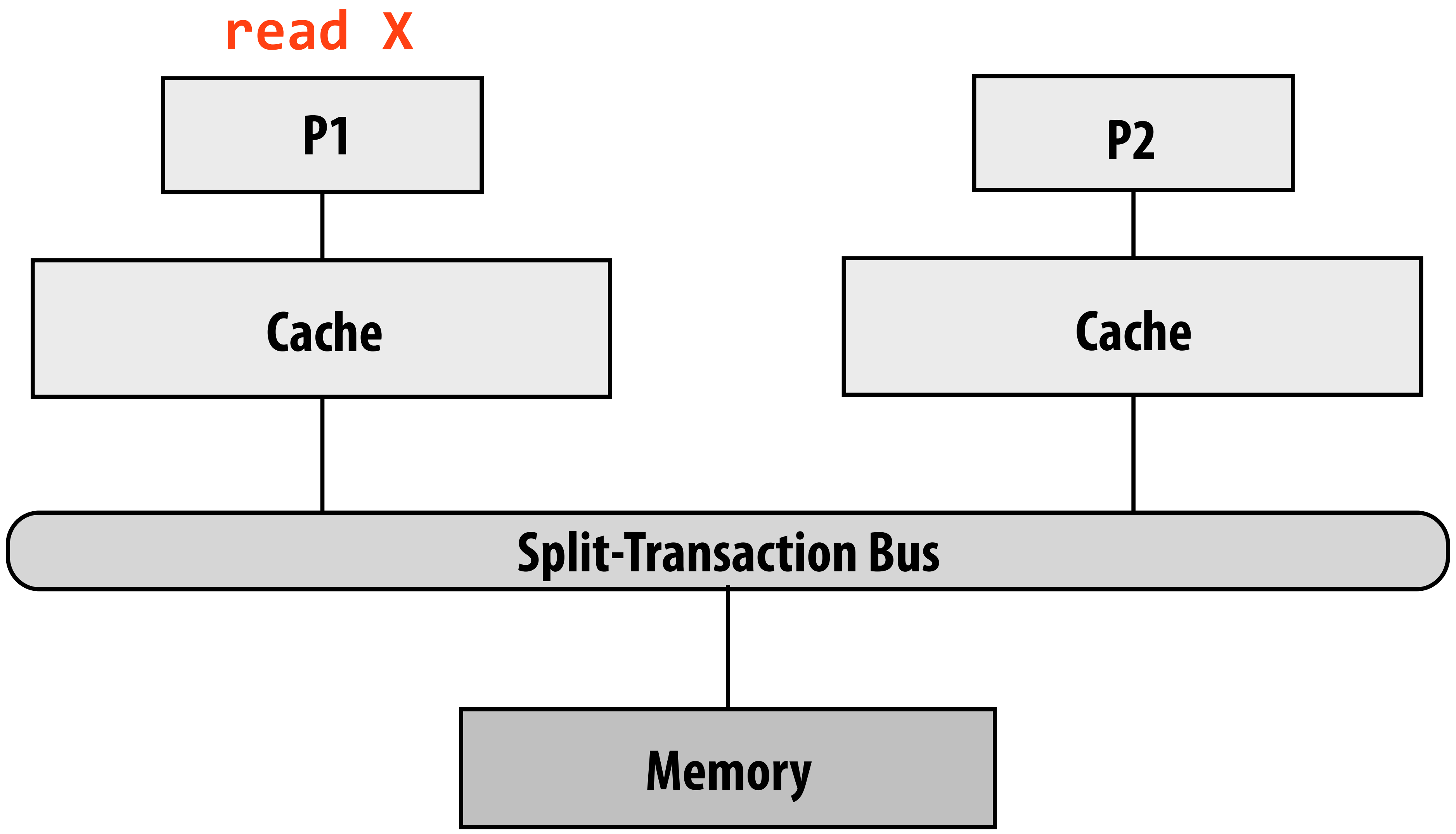
- Conflicting requests
  - Avoid conflicting requests by disallowing them
  - Each cache has a copy of the request table
  - Simple policy: caches do not make requests that conflict with requests in the request table

- Flow control:
  - Caches/memory have buffers for receiving data off the bus
  - If the buffer fills, client NACKs relevant requests or responses (NACK = negative acknowledgement)
  - Triggers a later retry

Situation 1: P1 read miss to X, write transaction involving X is outstanding on bus

P1 Request Table

Requestor	Addr	State
P2	X	Op: BusRdX

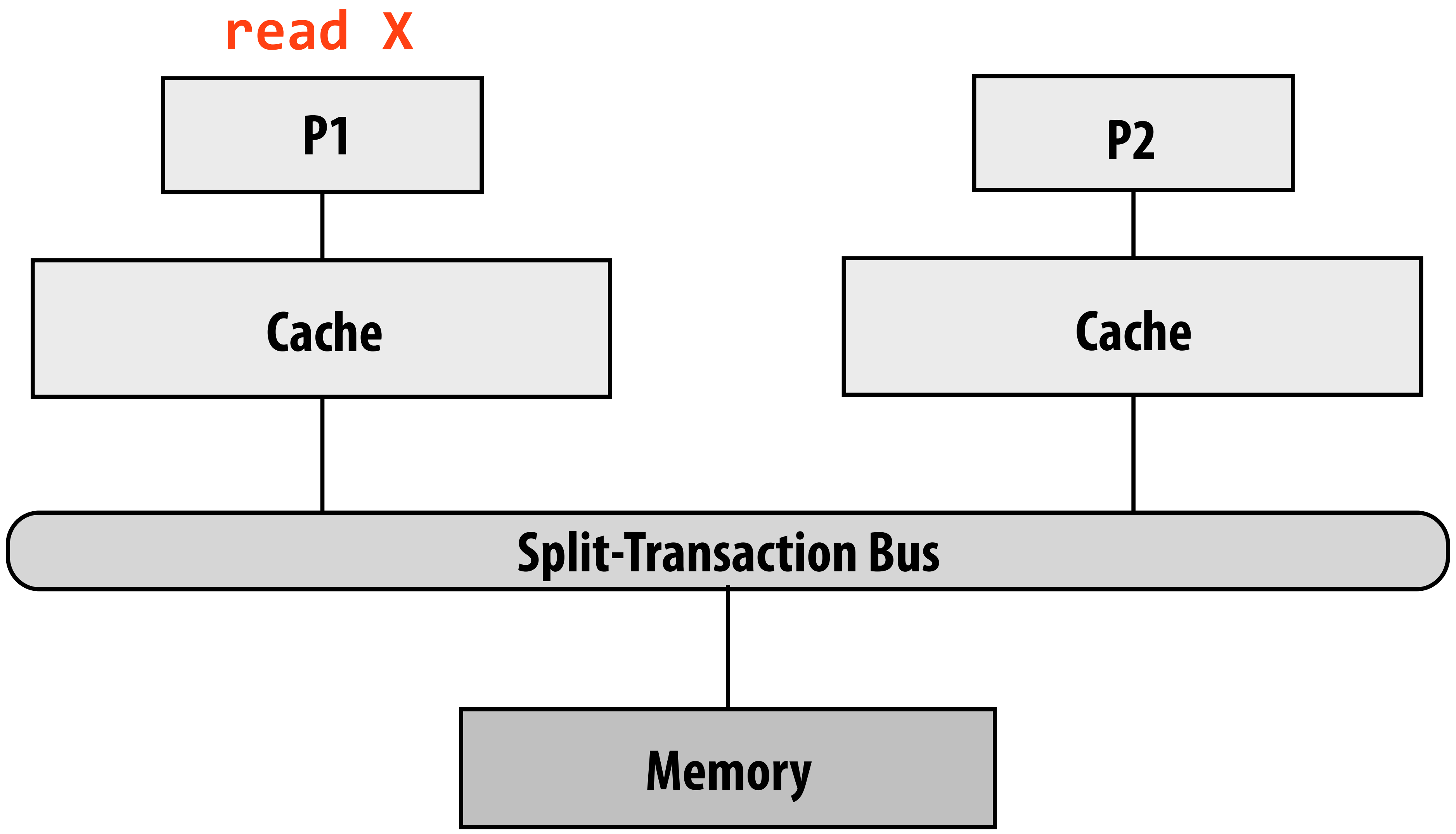


If there is a conflicting outstanding request (as determined by checking the request table), cache must hold request until conflict clears

Situation 2: P1 read miss to X, read transaction involving X is outstanding on bus

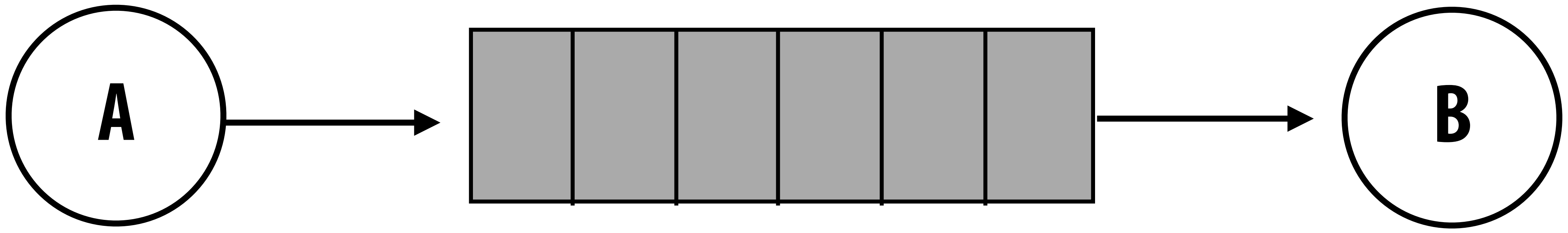
P1 Request Table

Requestor	Addr	State
P2	X	Op: BusRd , share



If outstanding request is a read: there is no conflict. No need to make a new bus request, just listen for the response to the outstanding one.

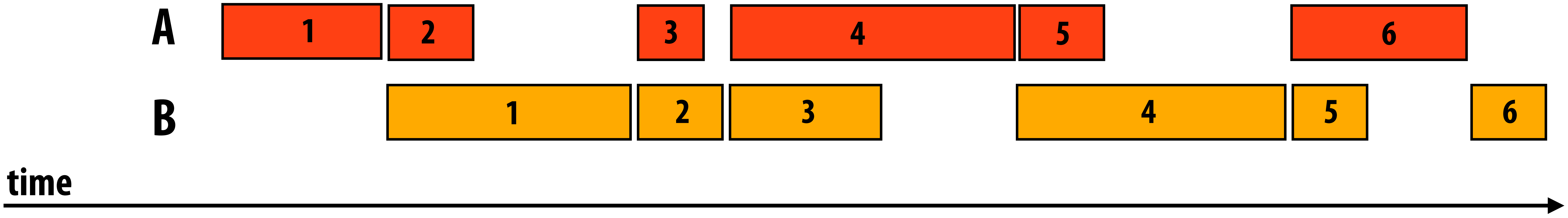
Why do we have queues?



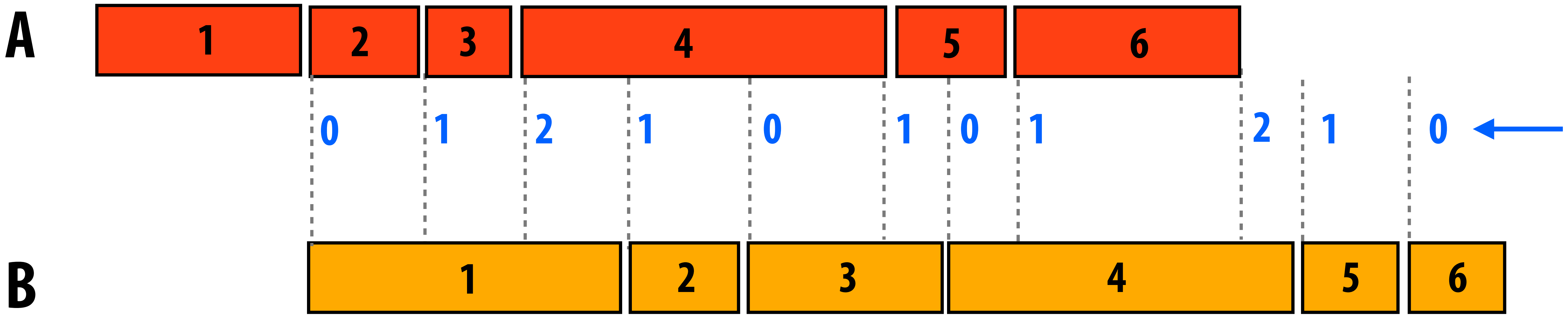
Answer: to accommodate variable (unpredictable) rates of production and consumption.

As long as A and B, on average, produce and consume at the same rate, both workers can run at full rate.

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.

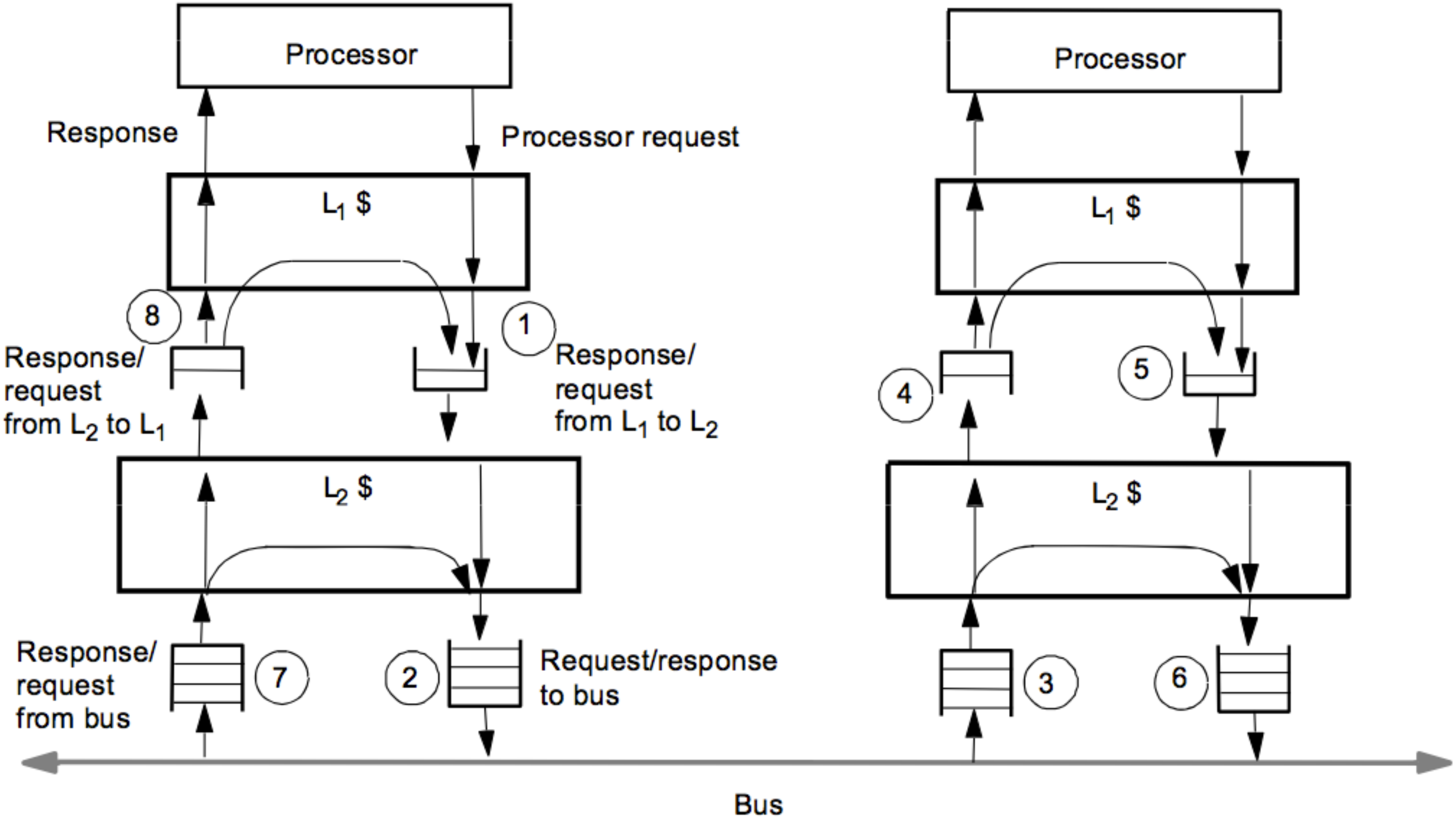
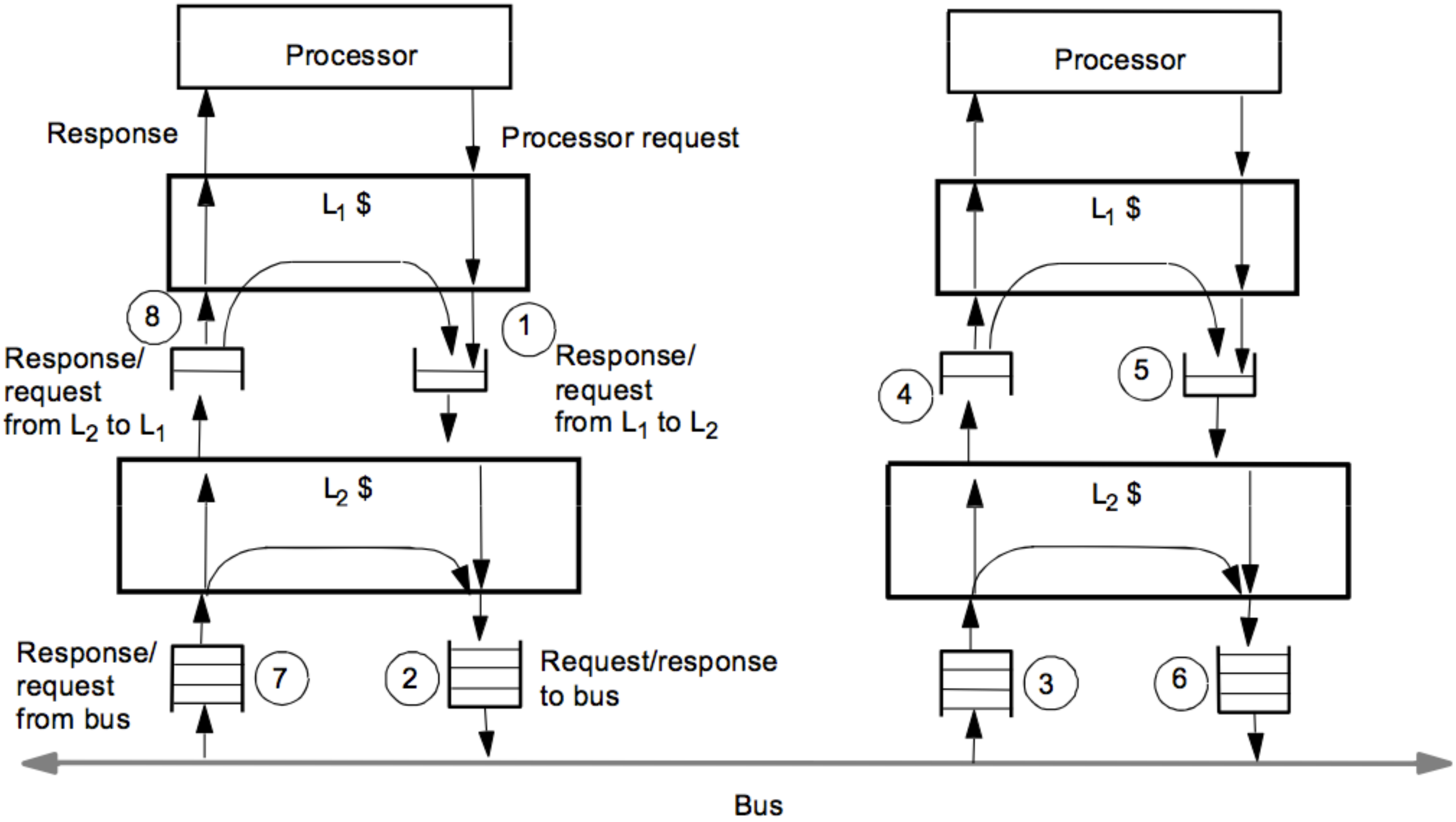


Figure credit: Culler, Singh, and Gupta

Recall the fetch-deadlock problem

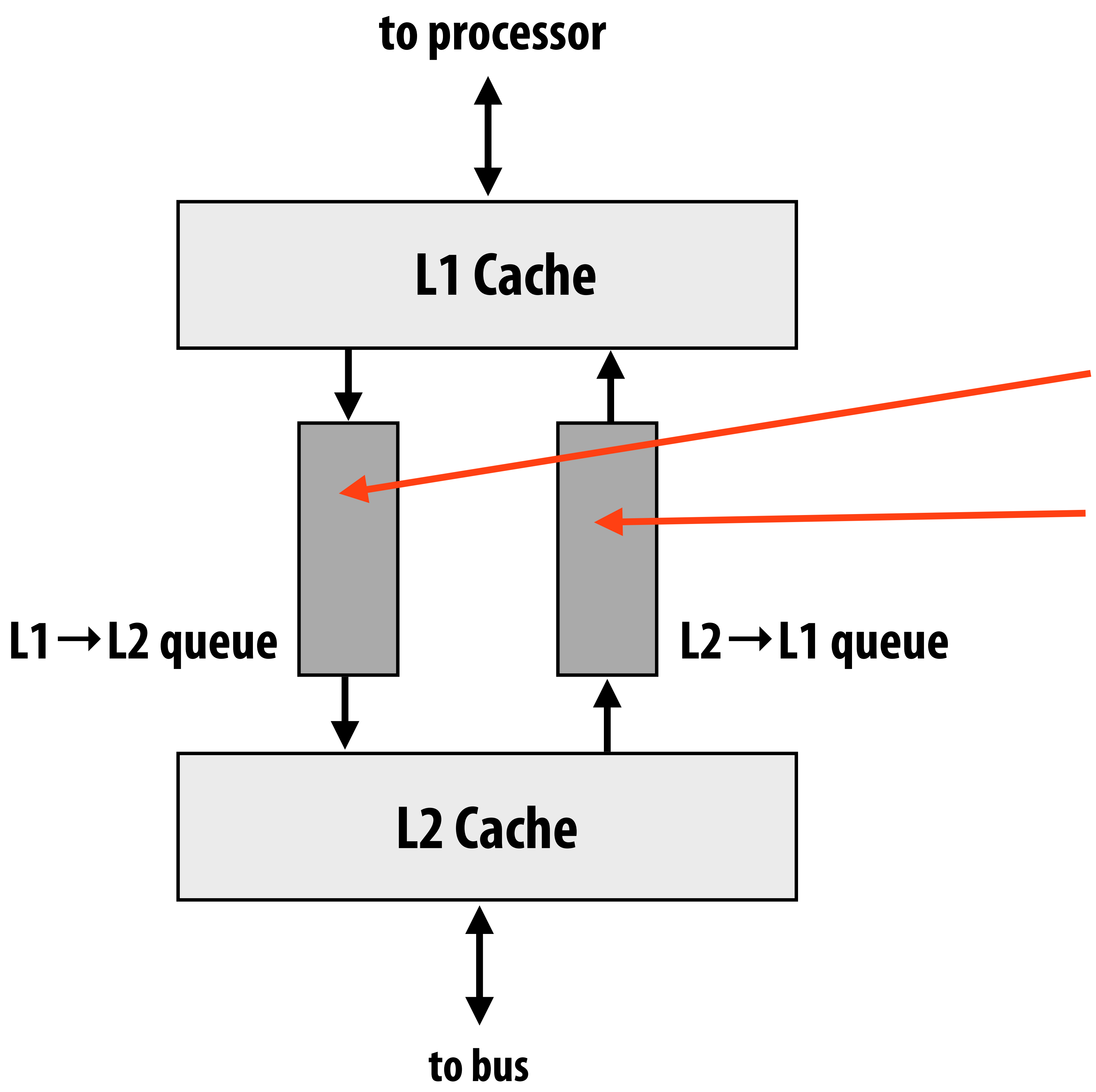


Assume one outstanding memory request per processor.

Consider fetch-deadlock problem: cache must be able to service requests while waiting on response to its own request (hierarchies increase response delay)

Figure credit: Culler, Singh, and Gupta

Deadlock due to full queues



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

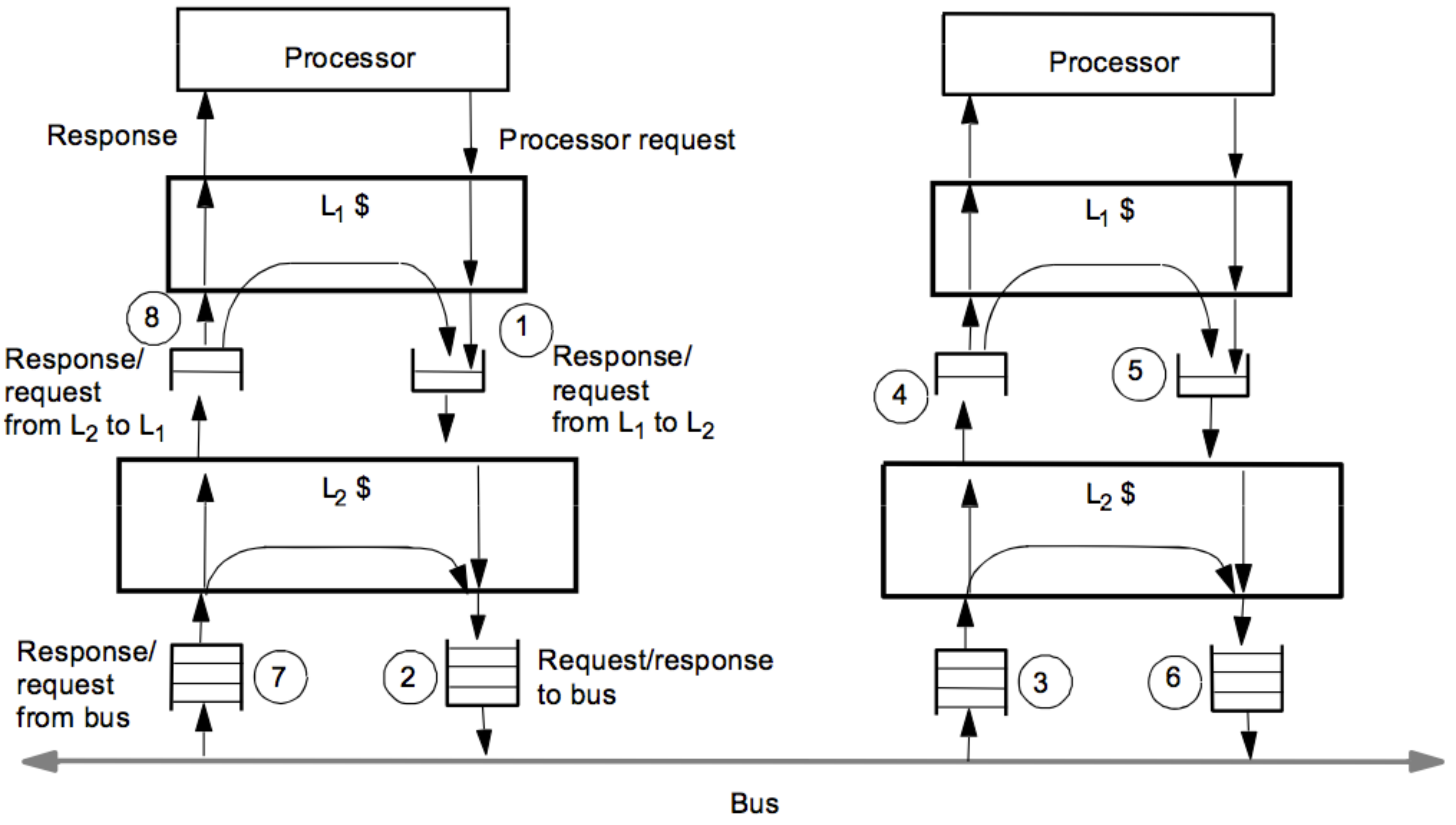
Outgoing read request (initiated by processor)

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

Multi-level cache hierarchies



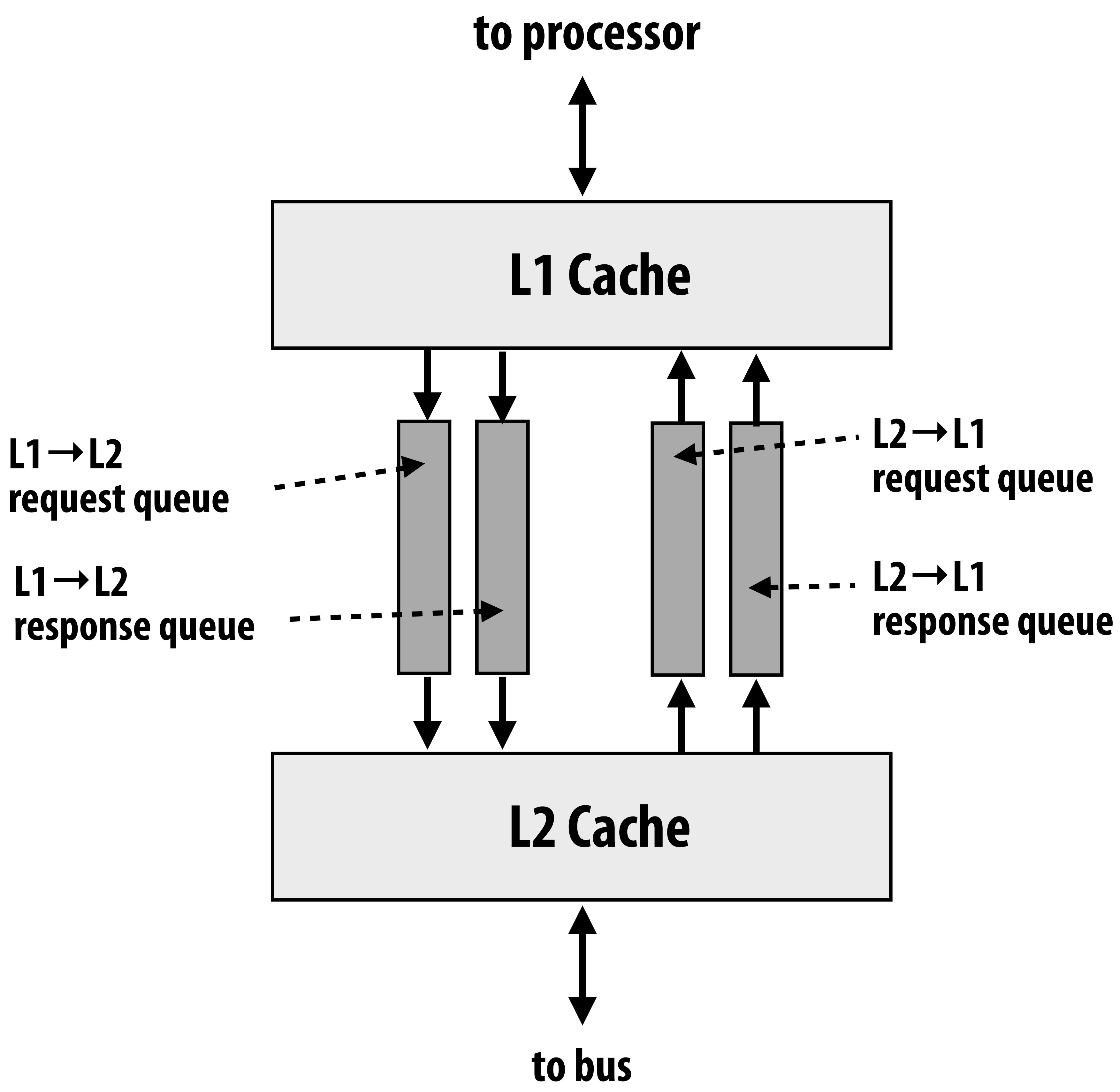
Assume one outstanding memory request per processor.

Consider fetch deadlock problem: cache must be able to service requests while waiting on response to its own request (hierarchies increase response delay)

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



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

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

```
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. (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