Lecture 15:

A Basic Snooping-Based Multi-processor

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2014

Tunes

"Stompa" (Serena Ryder)

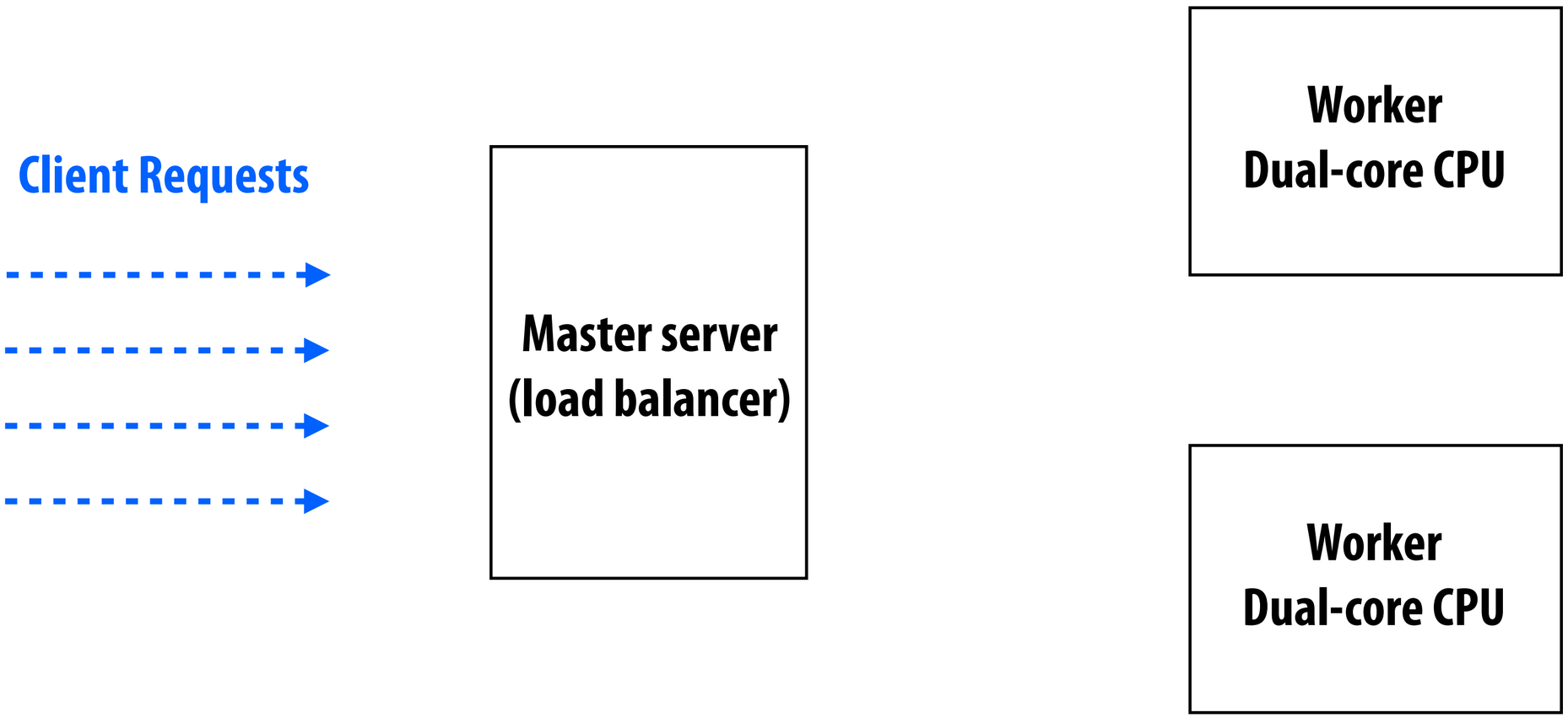
"I wrote Stompa because I just get so excited about writing load balancers."

- Serena Ryder

Spring break photo competition (vote on Piazza)

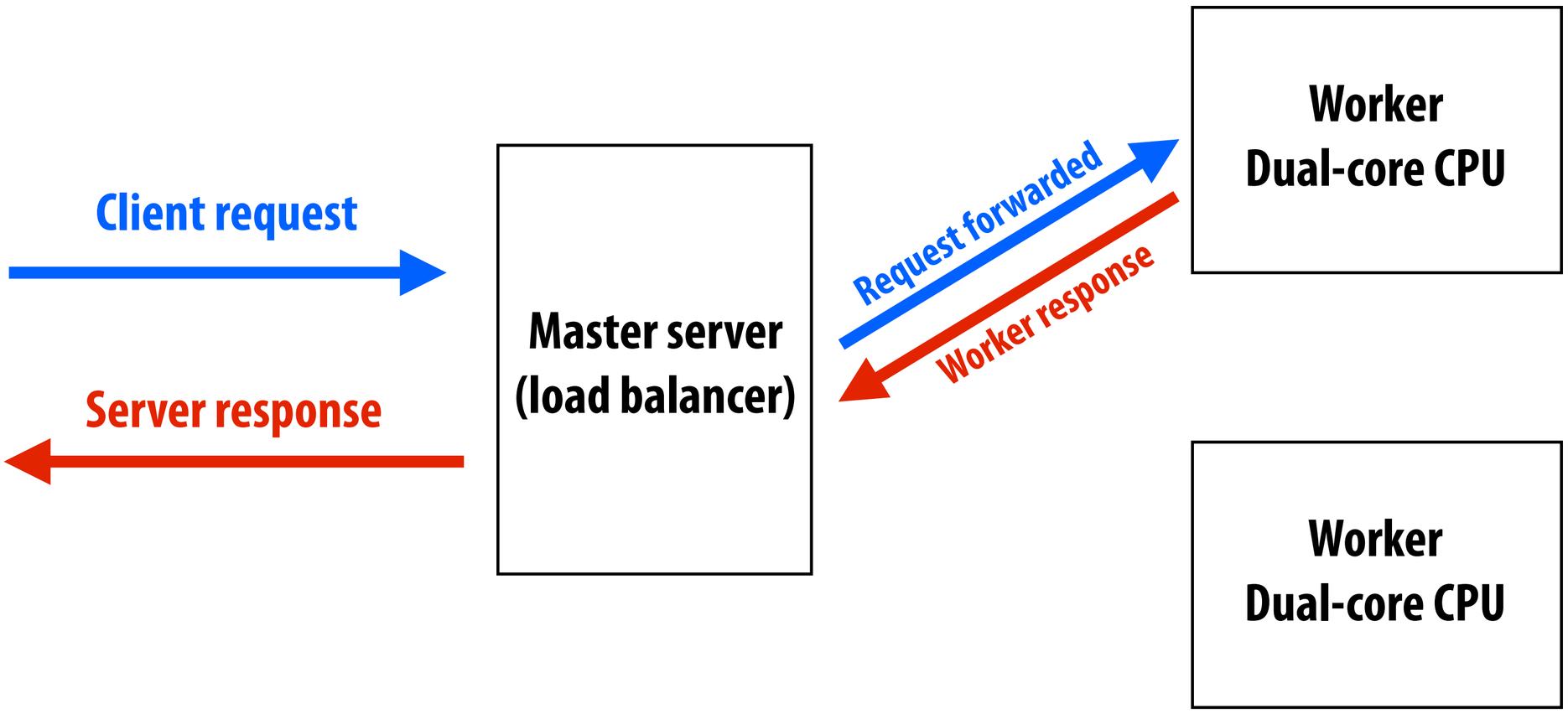
Assignment 4

- Due Thurs, March 27th
- You will implement a simple web site that efficiently handles a request stream



Assignment 4

- You will implement a load balancer/scheduler to efficiently handle a request stream



Assignment 4: the master node

- The master is a load balancer
- The master is structured as an event-driven system
  - The master has only one thread of control, but processes client requests concurrently

Master server (load balancer)

You implement:

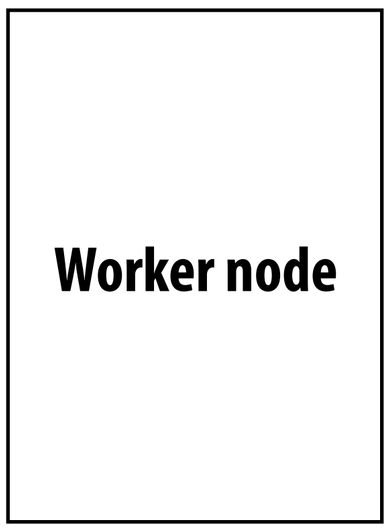
```
// take action when a request comes in
void handle_client_request(Client_handle client_handle, const RequestMsg& req);

// take action when a worker provides a response
void handle_worker_response(Worker_handle worker_handle, const ResponseMsg& resp);
```

We give you:

```
// sends a request to a worker
void send_job_to_worker(Worker_handle worker_handle, const RequestMsg& req);

// sends a response to the client
void send_client_response(Client_handle client_handle, const ResponseMsg& resp);
```

Assignment 4: the worker nodes

- The worker nodes are responsible for the "heavy lifting" (executing the specified requests)



You implement:

```
// take action when a request comes in
void worker_handle_request(const RequestMsg& req);
```

We give you:

```
// send a response back to the master
void worker_send_response(const ResponseMsg& resp);


// black-box logic to actually do the work (and populate a response)
void execute_work(const RequestMsg& req, ResponseMsg& resp);
```

Assignment 4: challenge 1

- There a number of different types of requests with different workload characteristics
  - Compute intensive requests (both long and short)
  - Disk intensive requests

```
{"time": 0,  "work": "cmd=highcompute;x=5", "resp": "42"}
{"time": 10, "work": "cmd=highcompute;x=10", "resp": "59"}
{"time": 20, "work": "cmd=highcompute;x=15", "resp": "78"}
{"time": 21, "work": "cmd=popular;start=2013-02-13;end=2013-03-23", "resp": "lecture/cachecoherence1 -- 856 views"}
{"time": 22, "work": "cmd=highcompute;x=20", "resp": "10"}
{"time": 23, "work": "cmd=highcompute;x=20", "resp": "10"}
{"time": 24, "work": "cmd=highcompute;x=20", "resp": "10"}
{"time": 30, "work": "cmd=popular;start=2013-02-13;end=2013-03-23", "resp": "lecture/cachecoherence1 -- 856 views"}
{"time": 40, "work": "cmd=popular;start=2013-02-13;end=2013-03-23", "resp": "lecture/cachecoherence1 -- 856 views"}
{"time": 50, "work": "cmd=popular;start=2013-02-13;end=2013-03-23", "resp": "lecture/cachecoherence1 -- 856 views"}
```

Assignment 4: challenge 2

- **The load varies over time! Your server must be elastic!**

```
{"time": 0,  "work": "cmd=highcompute;x=5", "resp": "42"}
{"time": 10, "work": "cmd=highcompute;x=10", "resp": "59"}
{"time": 20, "work": "cmd=highcompute;x=15", "resp": "78"}
{"time": 21, "work": "cmd=popular;start=2013-02-13;end=2013-03-23", "resp": "lecture/cachecoherence1 -- 856 views"}
{"time": 22, "work": "cmd=highcompute;x=20", "resp": "10"}
{"time": 23, "work": "cmd=highcompute;x=20", "resp": "10"}
{"time": 24, "work": "cmd=highcompute;x=20", "resp": "10"}
{"time": 30, "work": "cmd=popular;start=2013-02-13;end=2013-03-23", "resp": "lecture/cachecoherence1 -- 856 views"}
{"time": 40, "work": "cmd=popular;start=2013-02-13;end=2013-03-23", "resp": "lecture/cachecoherence1 -- 856 views"}
{"time": 50, "work": "cmd=popular;start=2013-02-13;end=2013-03-23", "resp": "lecture/cachecoherence1 -- 856 views"}
```

We give you:

```
// ask for another worker node
void request_boot_worker(int tag);

// request a worker be shut down
void kill_worker(Worker_handle worker_handle);
```

You implement:

```
// notification that the worker is up and running
void handle_worker_boot(Worker_handle worker_handle, int tag);
```

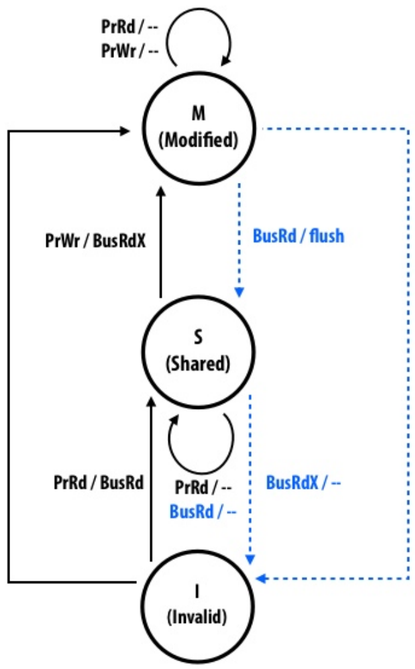
Assignment 4

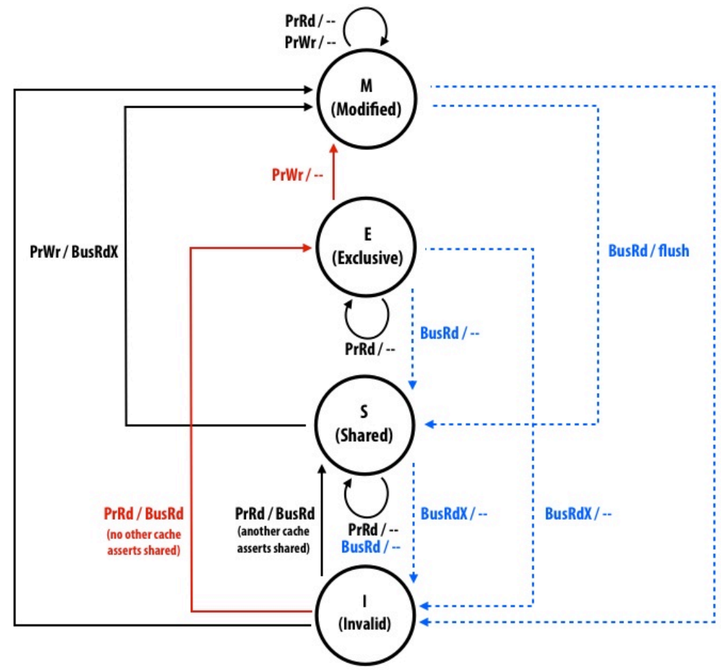
- Goal: service the request stream as efficiently as possible (low latency response time) using as few workers as possible (low website operation cost)

- Ideas you might want to consider:
  - What is a smart assignment of jobs (work) to workers?
  - When to [request more/release idle] worker nodes?
  - Can overall costs be reduced by caching?

Today's topic: A basic implementation of cache-coherence

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





- Before spring break we talked about cache coherence <u>protocols</u>
  - 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 the desired protocol (in a real machine… behavior is more complex)

Our implementation goals

1. Correct
   - Implements cache coherence
   - Adheres to specified memory consistency model
2. High performance
3. Minimize cost (e.g., minimize extra hardware)

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

What you should know (from this lecture)

- 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! (next Monday's lecture)
- Understand why maintaining coherence is challenging to implement, even when operating under simple machine design parameters
  - Mental model of hardware: many components operating in parallel
  - How do performance optimizations make correctness challenging?

Terminology

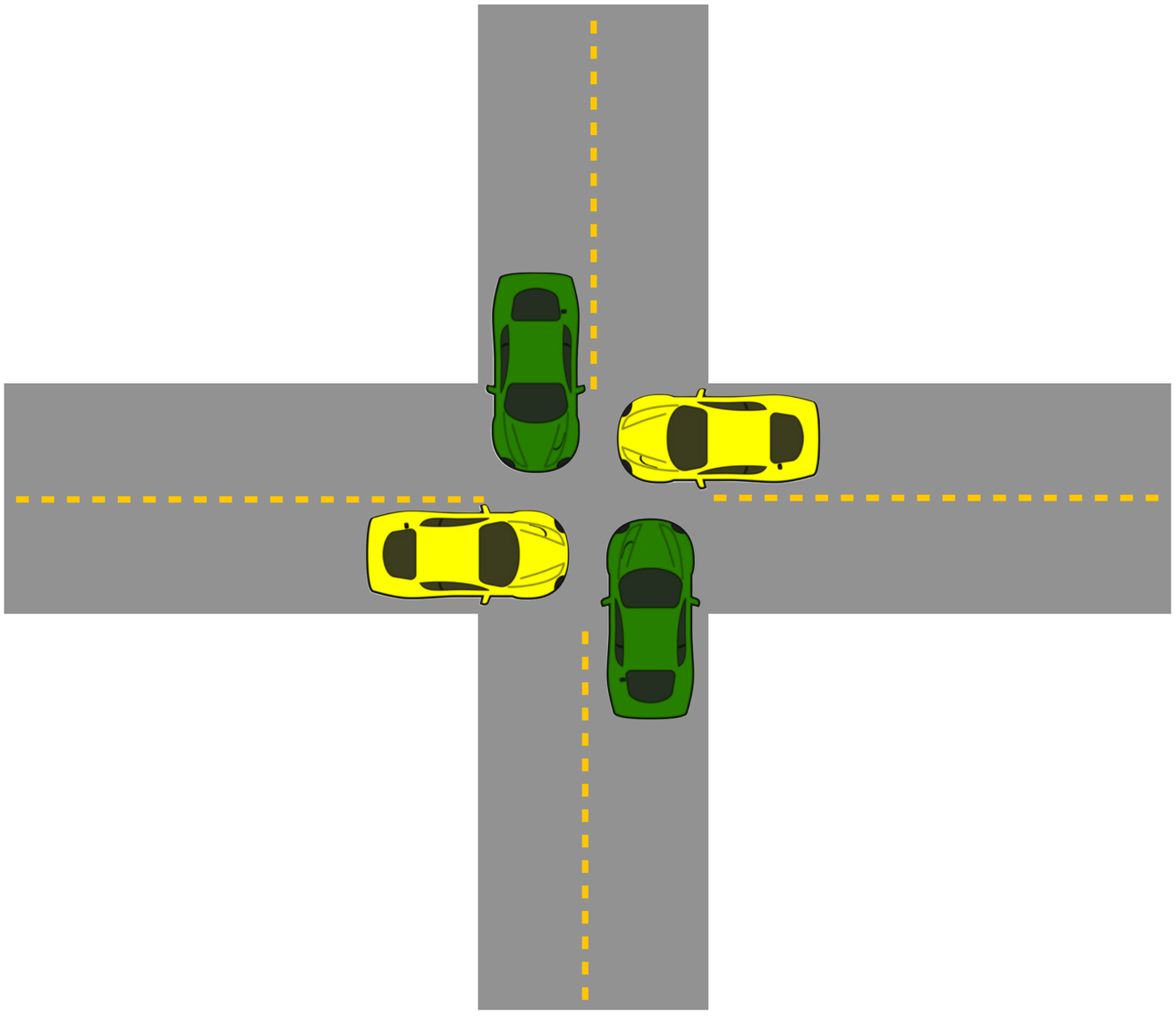
Deadlock

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.

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!")

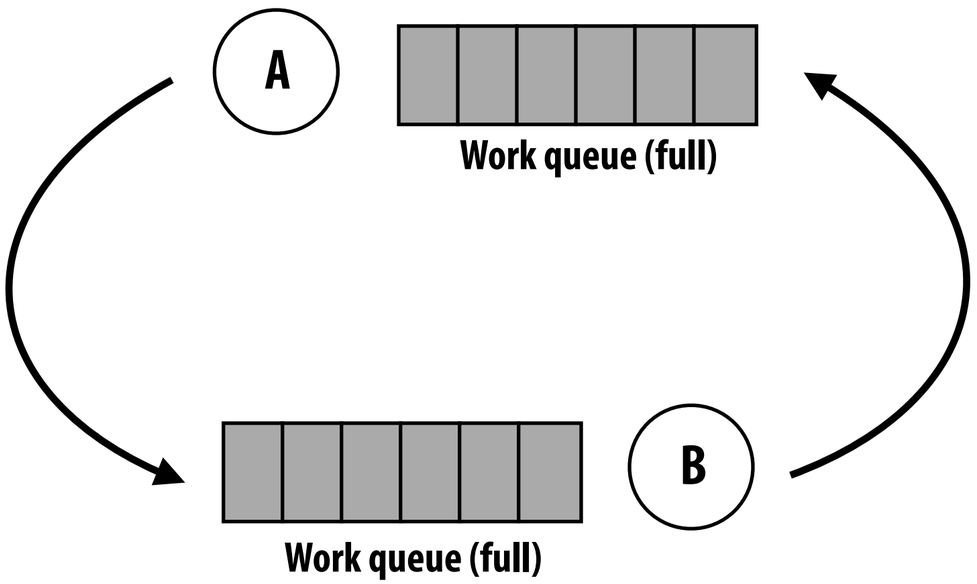
More deadlock

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

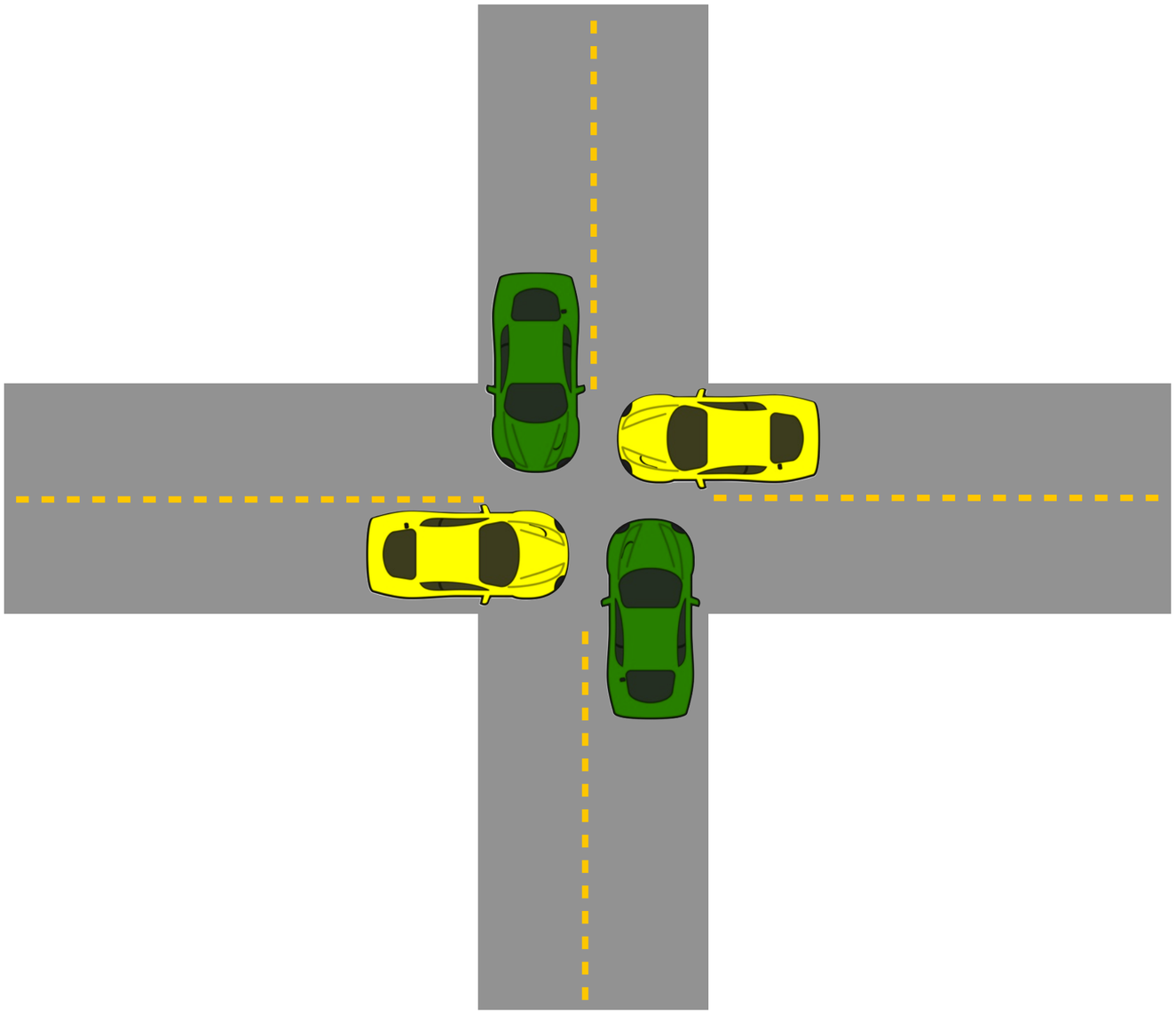
Example 2:

```
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, ...
```

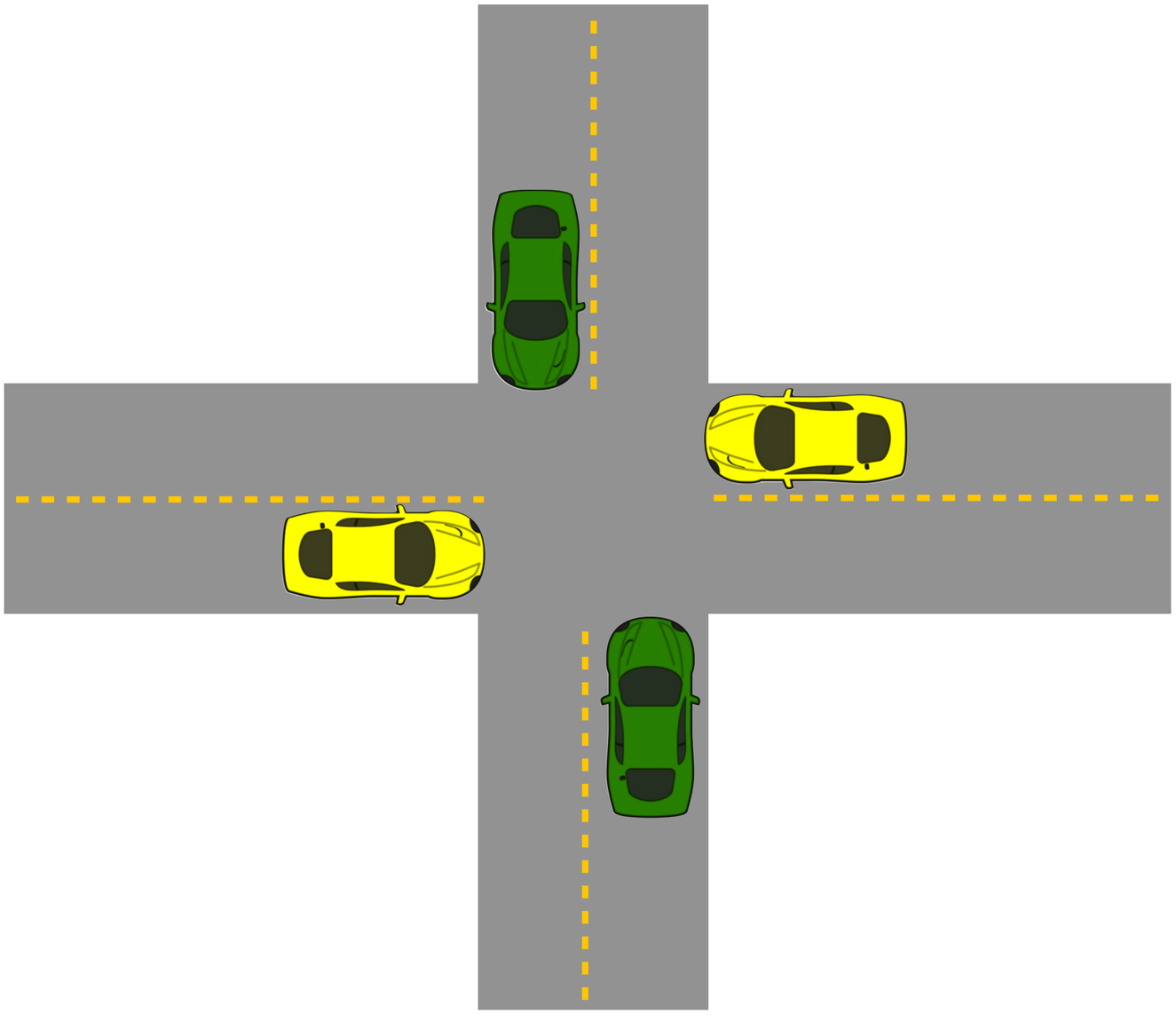
Every process sends a message (blocking send) to the processor with the next higher id

Then receives message from processor with next lower id.

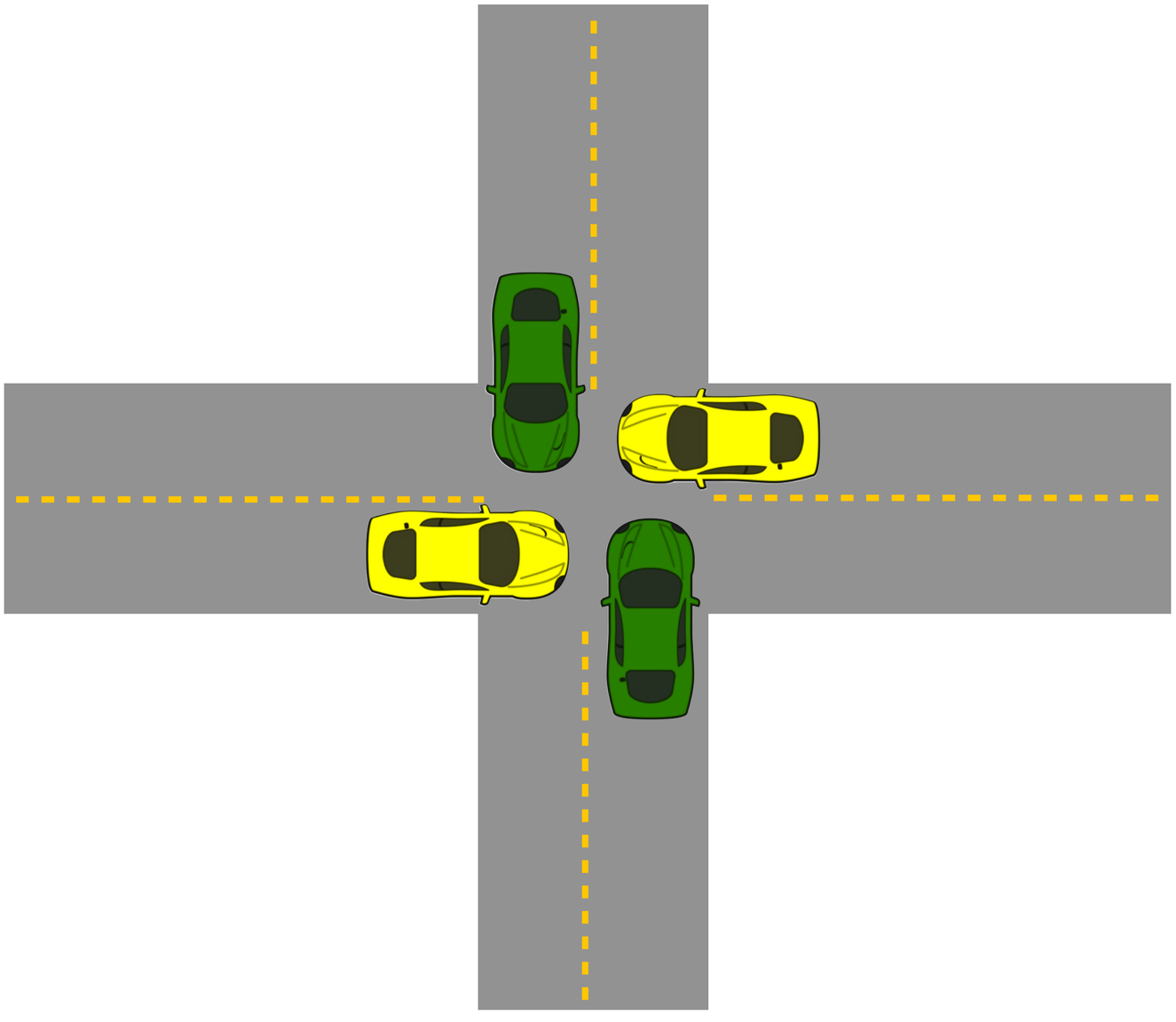
Livelock



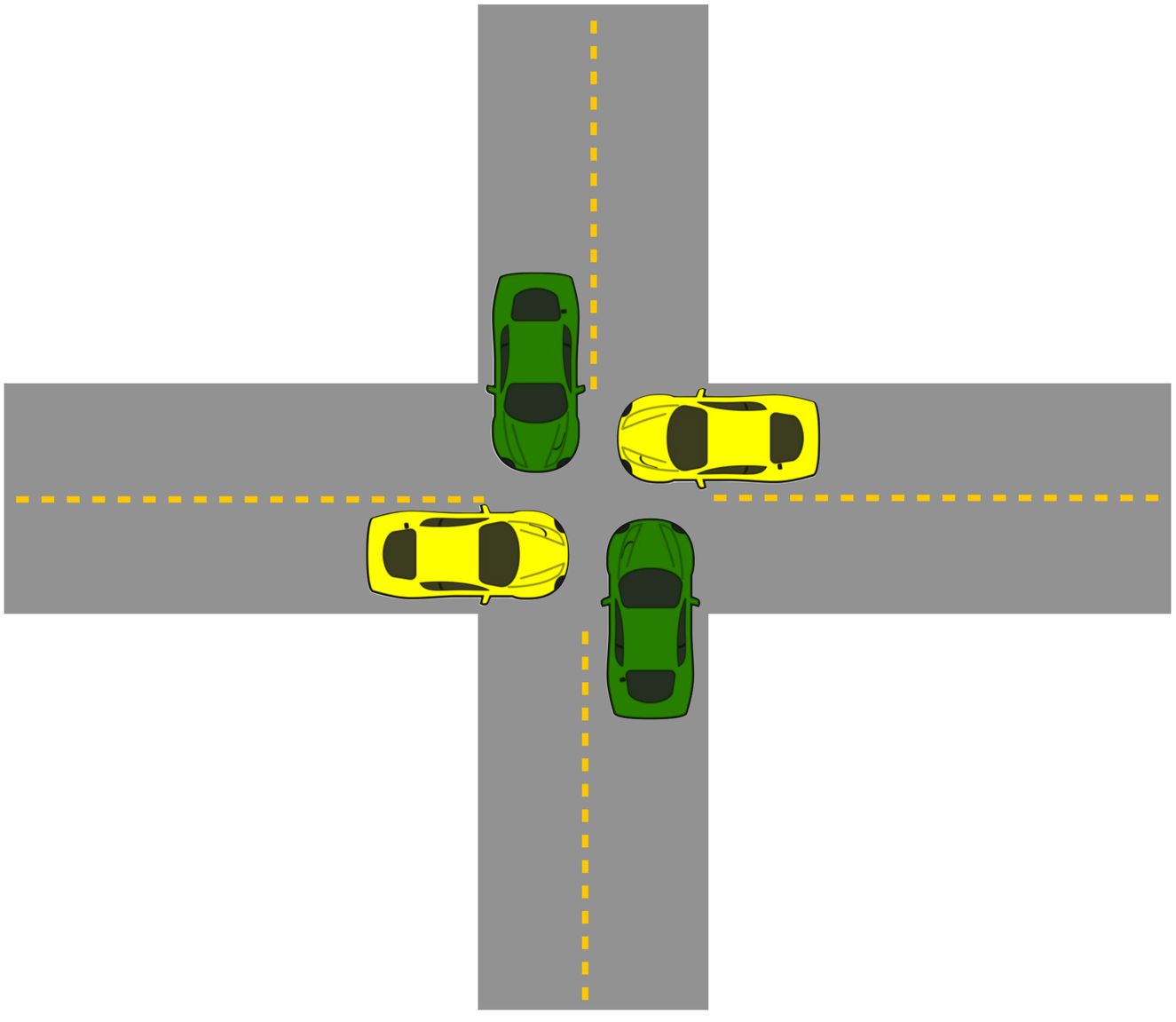
Livelock



Livelock



Livelock



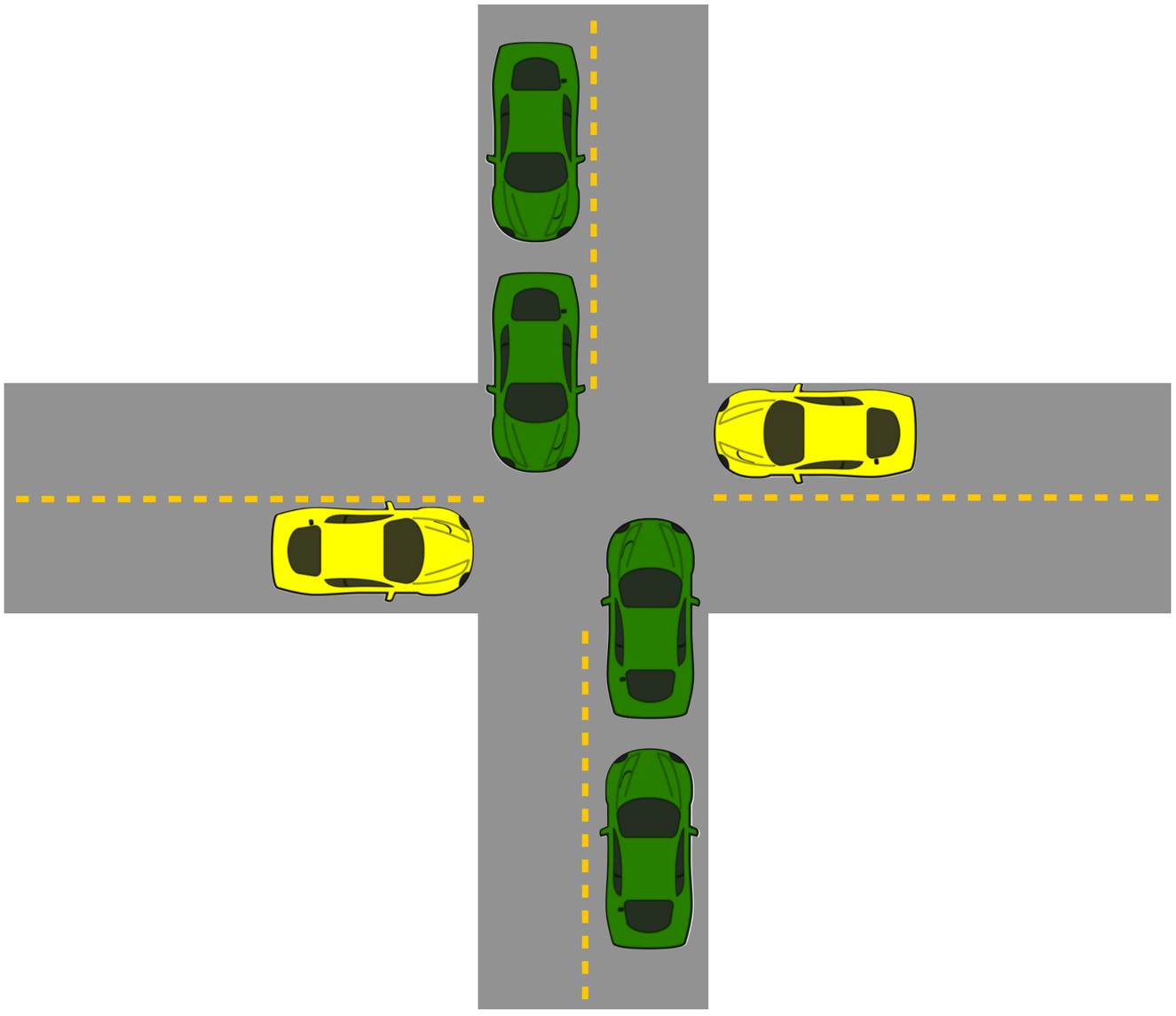
Livelock is a state where a system is executing many operations, but no thread is making meaningful progress.

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

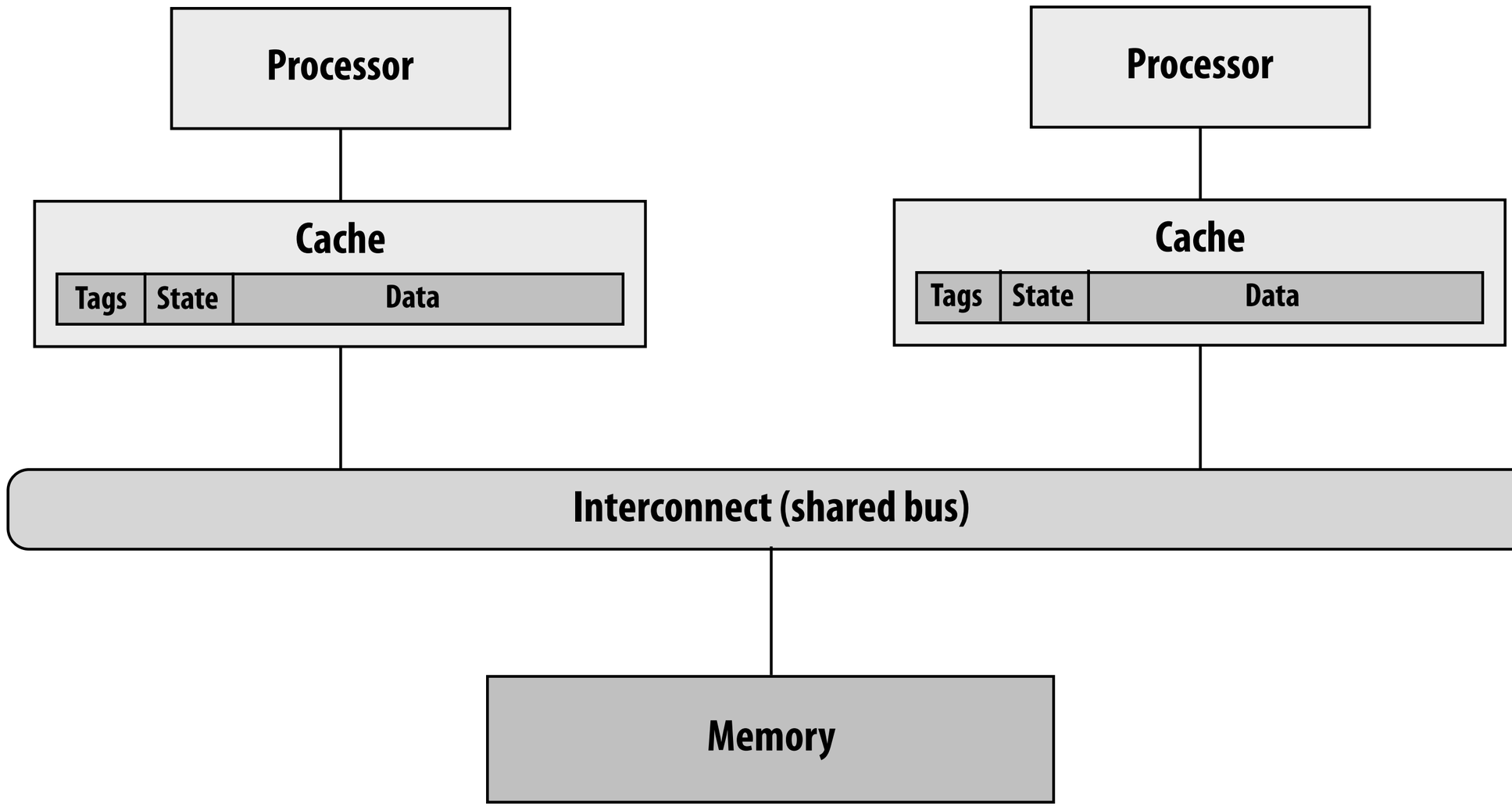
(as soon as green cars pass, yellow cars can go)

Example: assume left-right traffic must yield to top-bottom traffic.

A basic implementation of snooping

Basic system design

- One outstanding memory request per processor
- Single level, write-back cache per processor
- 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

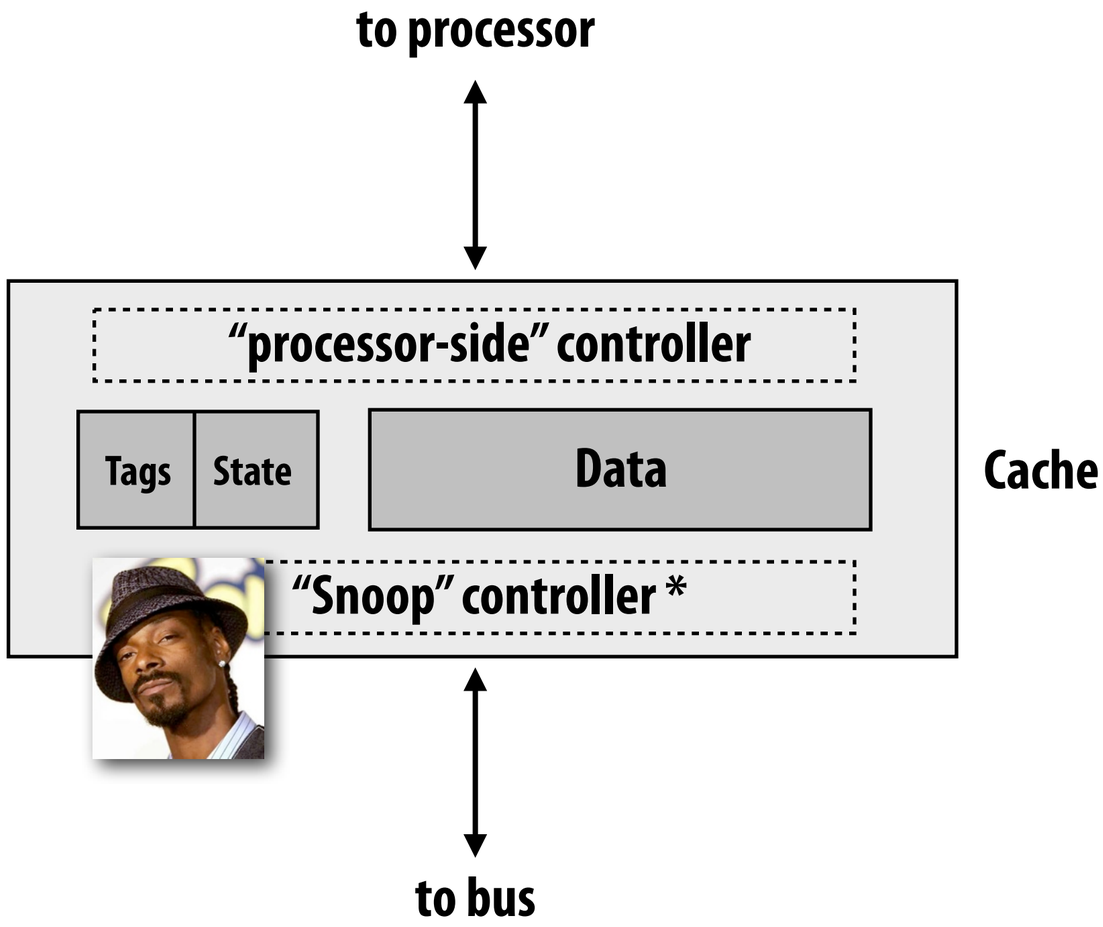
Multi-processor atomic bus:

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

BusWr: 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

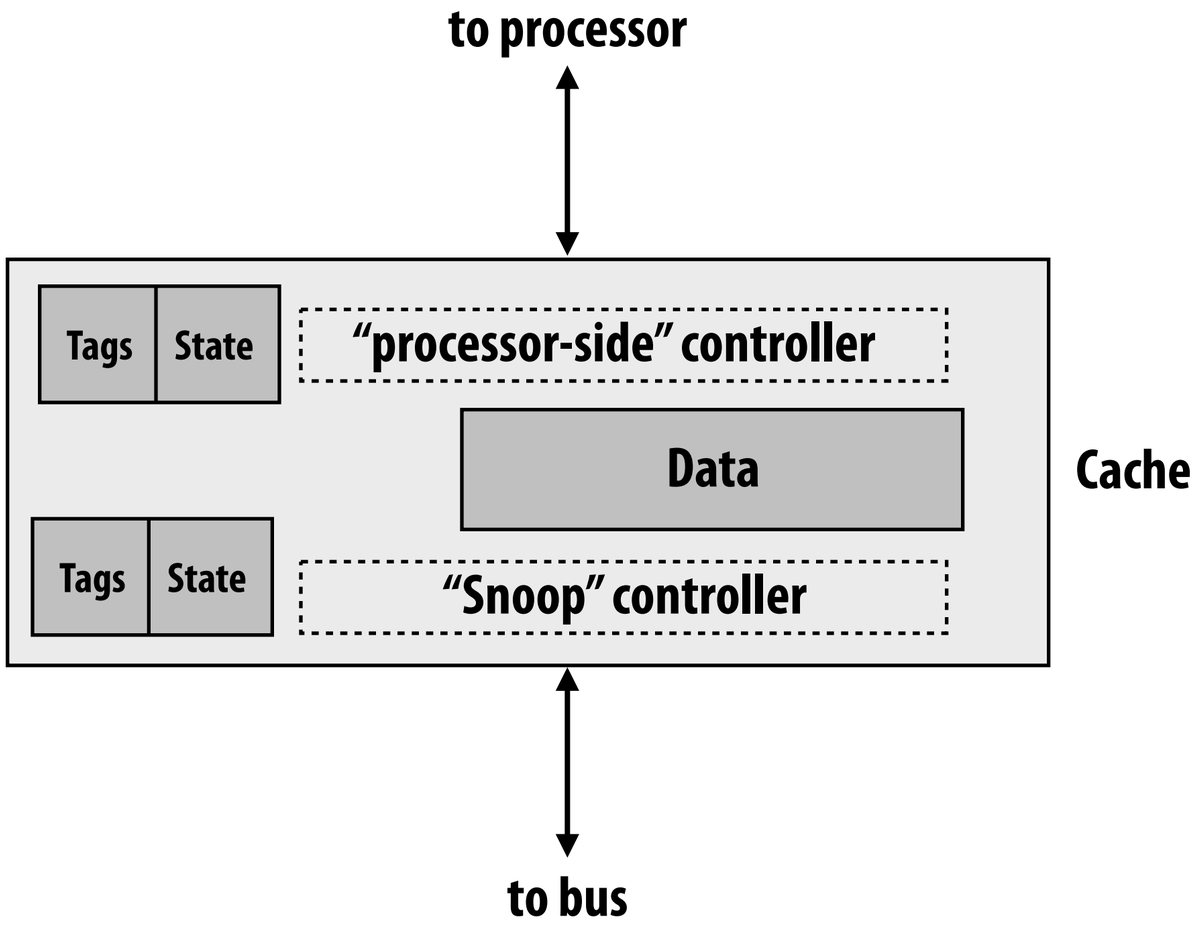


If bus receives priority:
During bus transaction, processor is locked out from cache.

If processor receives priority:
During processor cache accesses, cache can't respond with snoop (delaying 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**

Allow simultaneous access by processor-side and snoop controllers



Option 1: 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)

Reporting snoop results protocol in MESI

- Assume a cache read miss
- 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

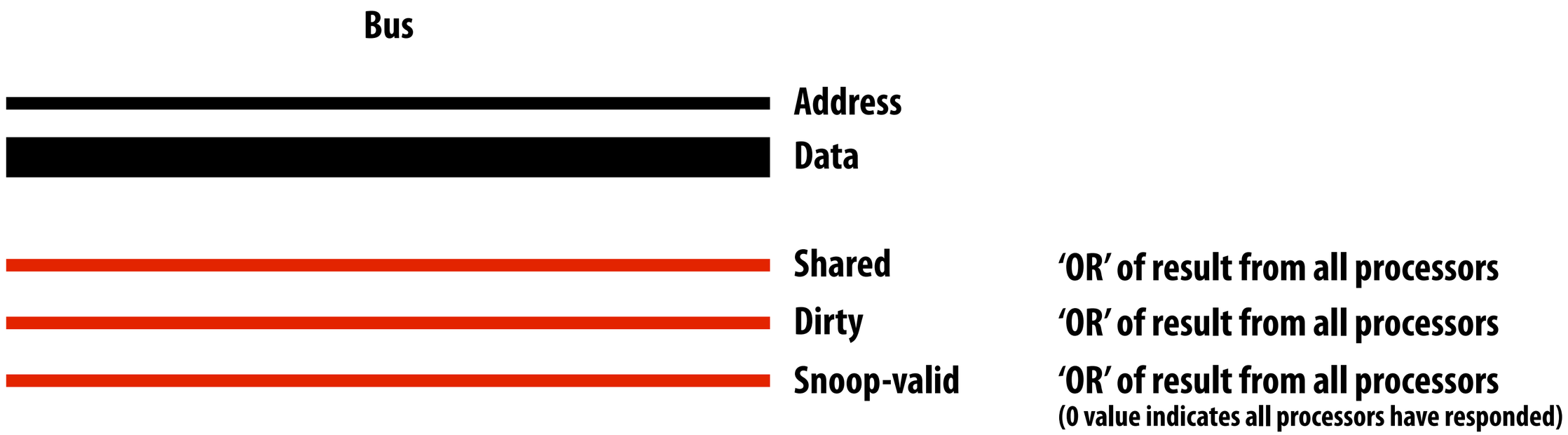
Memory needs to know what to do

Loading cache needs to know what to do

HOW?

WHEN?

Reporting snoop results: how



Reporting snoop results: when

Mainly an issue of when memory should react to the request

1. **Fixed number of clocks (worst case) after address appearing on bus**
   - **All caches guaranteed to respond in a fixed number of clocks**
   - **Note importance of duplicated tags (to meet guarantee)**

2. **Variable delay**
   - **Memory assumes one of the caches will service request until it hears otherwise**
   - **More complex, but lower latency if snoops are completed quickly**

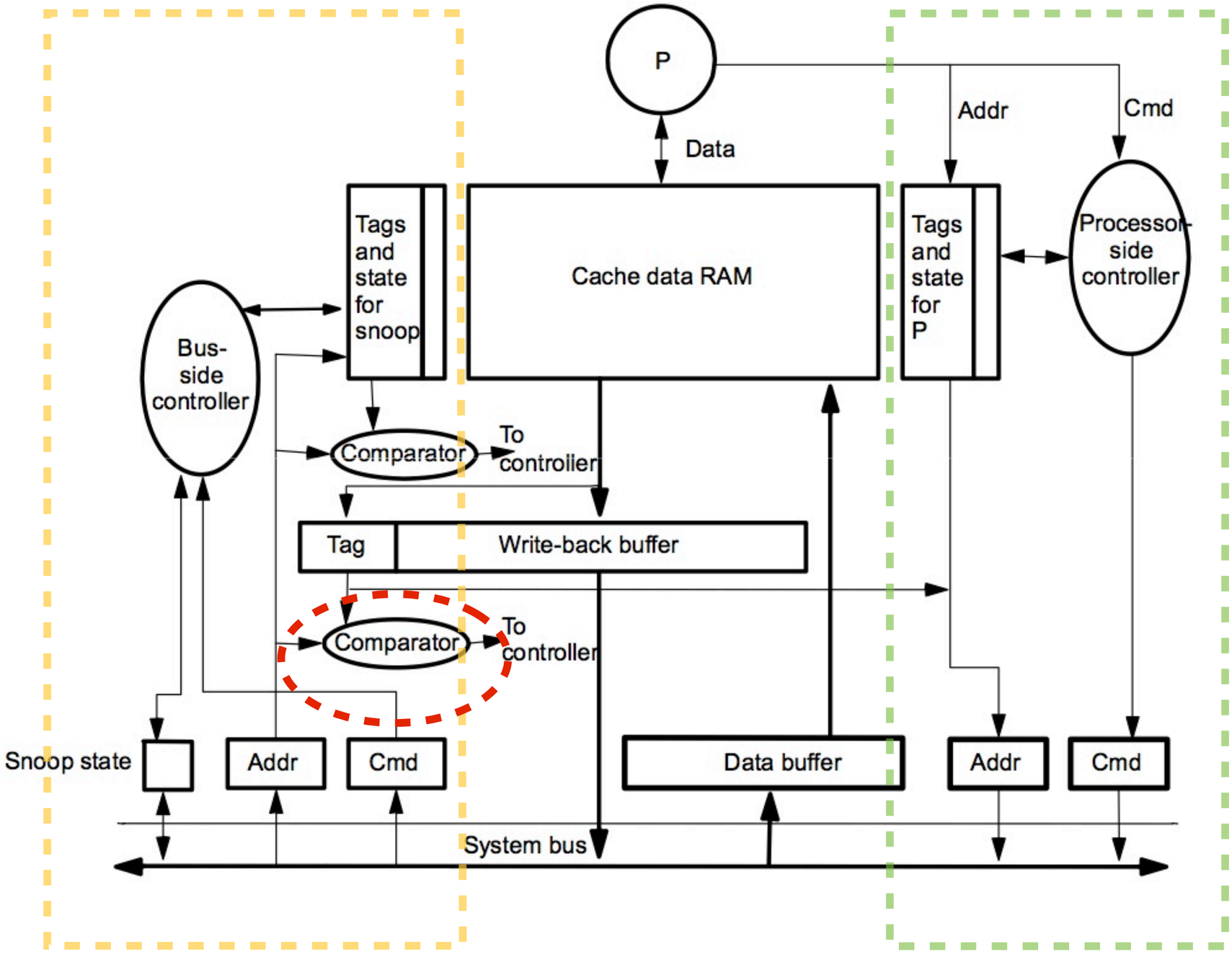
Handling write backs

- **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 (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**

Cache with write-back buffer



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

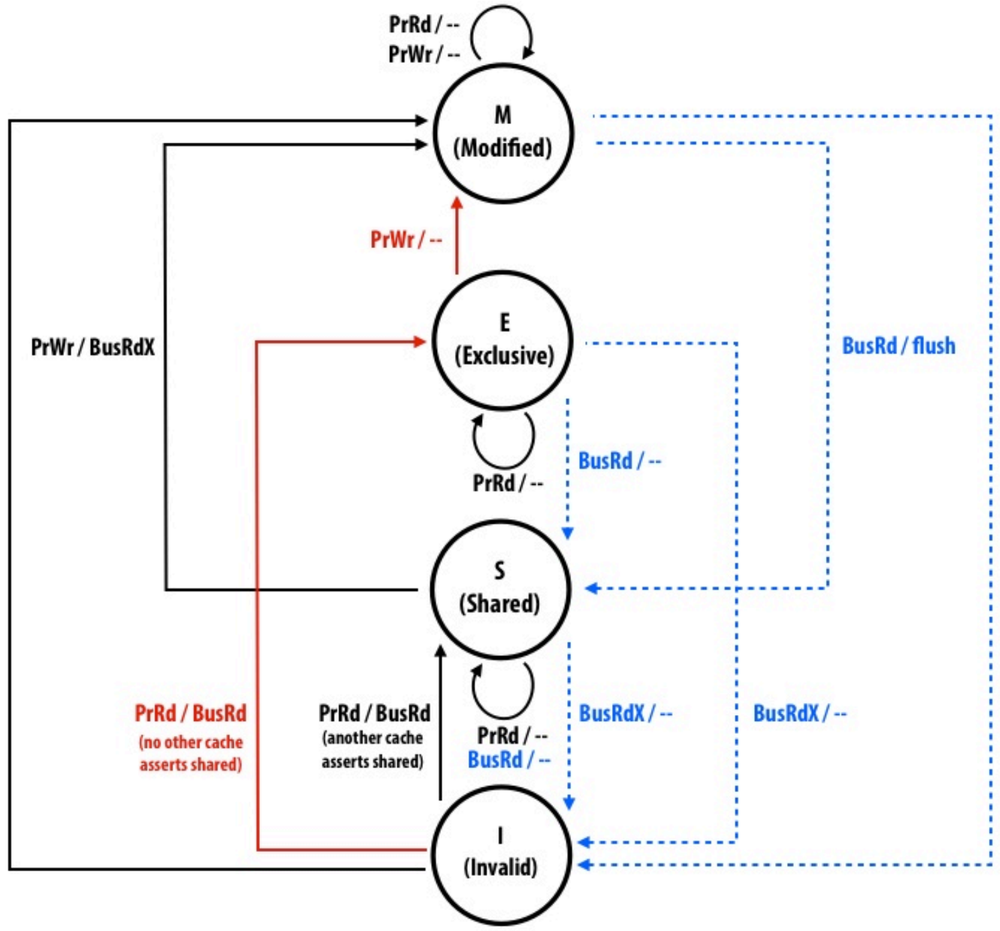
Snoop controller must check write back buffer address in addition to cache tags.

If match:

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

Non-atomic state transitions

- State transition diagrams during protocol discussion assumed that transitions were atomic
- Today we assume the bus transaction is atomic, but <u>all</u> the operations the system performs as a result of a memory operation are not
  - Look up tags, arbitrate for bus, wait for actions by other controllers, etc.
- Must be careful to handle race conditions appropriately



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

Write serialization

- Tempting optimization: on processor write, update cache line, allow processor to proceed prior to sending transaction out to bus (to obtain exclusive access)

- Violates coherence. Why?
  - Why does a write-back buffer not cause this problem?

- 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 writes
    - 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)

Next time: will discuss more advanced (a.k.a. more complex) implementations that strive for higher performance.

Summary:
Source of complexity is parallelism and concurrency

- 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 <u>abstracted</u> by the architecture as atomic (e.g., loads, stores) are <u>implemented</u> 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)