Lecture 16:
A Basic Snooping Multi-Processor

CMU 15-418: Parallel Computer Architecture and Programming (Spring 2012)

Announcements

- **Assignment 4**
  - To many of you: just get it done!
  - You only <u>are required</u> to report timings out to 32 CPUs (although it would be great to see how code scales to larger processor counts)
  - Keep track of all the things you try. Describe your thought process in the writeup.

- **Special queue that should decrease wait time on Blacklight**
  - Add option `-q pri_3` to your qsub command
  - PSC's comment:

    "The limits for the queue are 256 cores and 30 minutes of walltime, although jobs that ask for 64 or fewer cores will be better turnaround. because of how we have structured the queue."

Final projects

- Key deadlines:
  - Mon, April 2: project proposal: 1-2 page writeup
  - Fri, April 20: project checkpoint: 1-2 page writeup
  - Thurs, May 10: final presentations + final writeup

Your choice of computing resources

- Quad-core Intel Xeon CPUs in GHC 5205
- NVIDIA GPUs in GHC 5205
- Blacklight
- If you have a machine you want to try, that's a possibility too

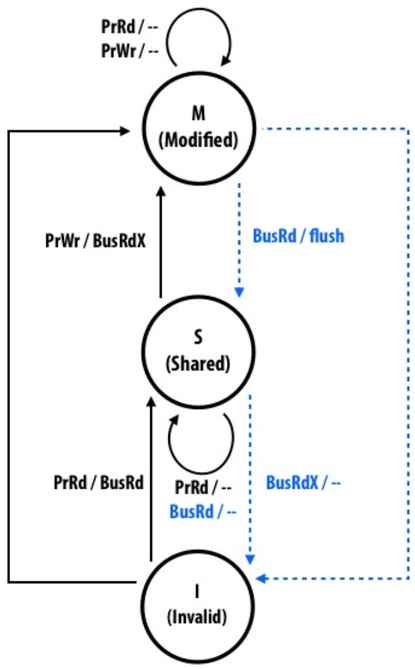
Your choice of project: some suggestions

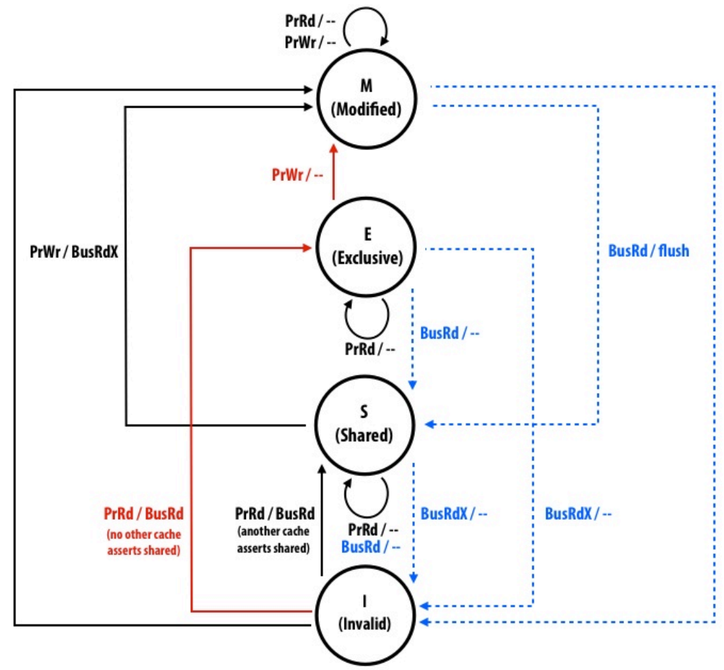
- A great source of application-oriented project ideas are projects you've done in other classes (now make them fast)
  - machine learning
  - A.I.
  - graphics / physical simulation
  - computational photography
  - computer vision

- But systems projects are also encouraged
  - Workload analysis:
    - comparing performance on GPU vs CPU
    - simulating behavior of a bunch of SIMD widths
  - Characterize performance of GPU using microbenchmarks
  - Modify ISPC compiler

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 designing a machine that efficiently implements the desired protocol (in a real machine… behavior is more complex)

Our implementation goals

1. Correct
   - Implements cache coherence
   - Adheres to specified consistency model
2. High performance
3. Minimize cost (for us: minimize extra hardware)

As you will see: tricks to gain high performance tend to make ensuring correctness tricky.

What you should know

- The concept of pipelining
- Deadlock, livelock, starvation
- Basic understanding of how a bus works
- Understand why maintaining coherence is challenging to implement, even when operating under simple machine design parameters
  - Mental model of hardware: many components, many simultaneous transactions
  - How do performance optimizations make correctness challenging?

Background concepts

- Pipelining

- System correctness/fairness issues:
  - Deadlock
  - Livelock
  - Starvation

Pipelining

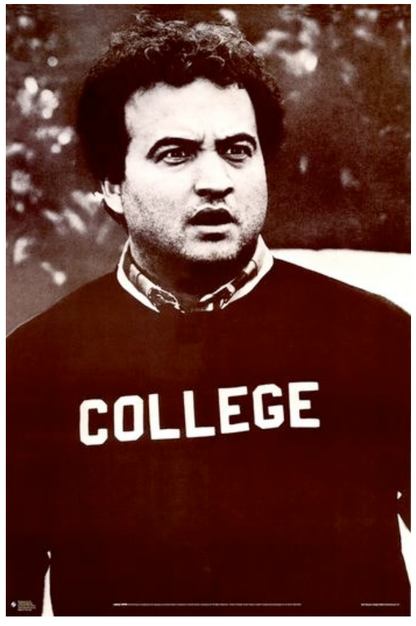
Doing your laundry

Operation: do your laundry

1. Wash clothes
2. Dry clothes
3. Fold clothes

Washer
45 min

Dryer
60 min



College Student
15 min

Latency of completing 1 load of laundry = 2 hours

Increasing laundry throughput

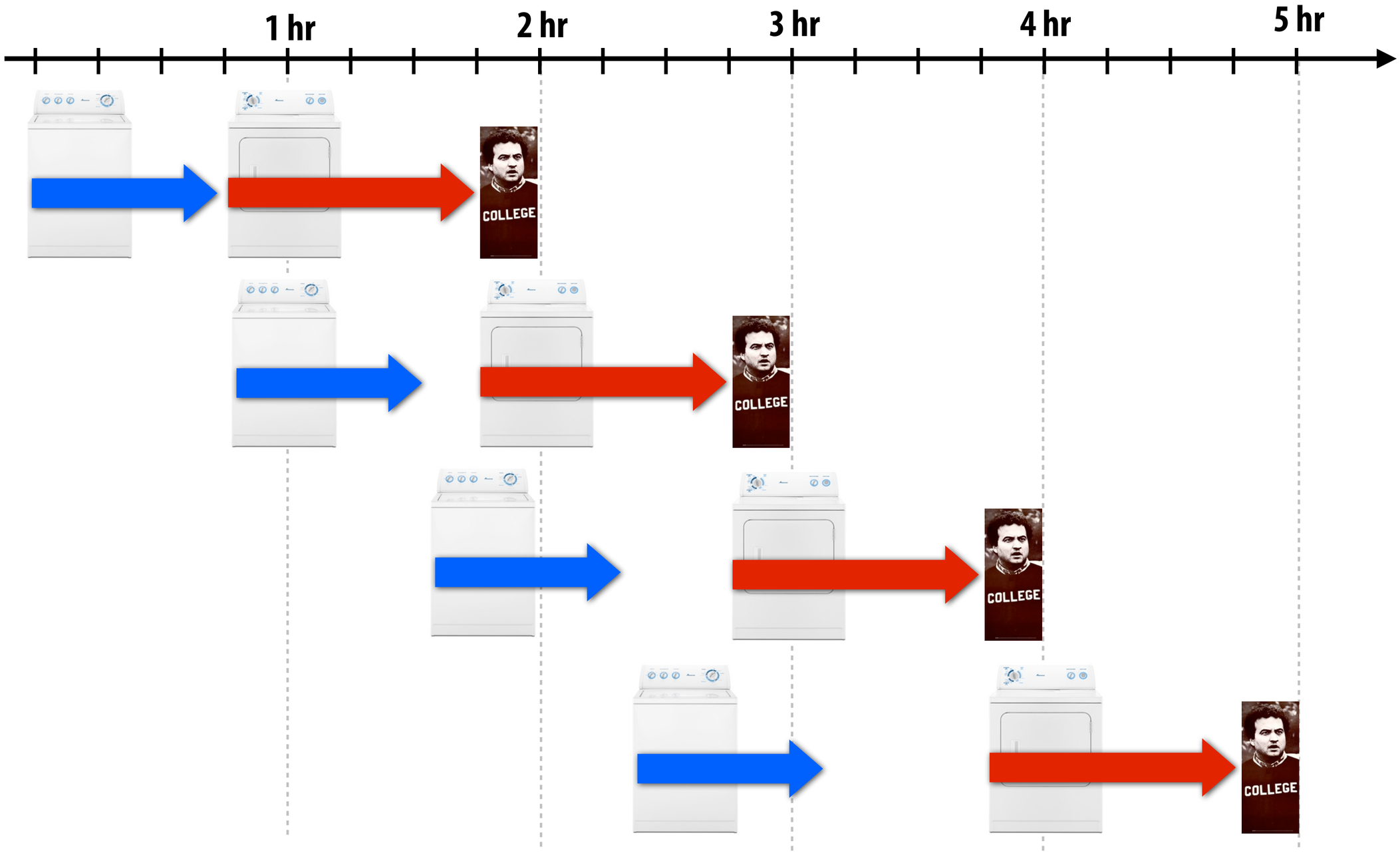
Goal: maximize throughput of many loads of laundry

On approach: duplicate execution resources:
use two washers, two dryers, and call a friend

Latency of completing 2 loads of laundry = 2 hours
Throughput increases by 2x: 1 load/hour
Cost increases by 2x: use twice the resources

Pipelining

Goal: maximize throughput of many loads of laundry



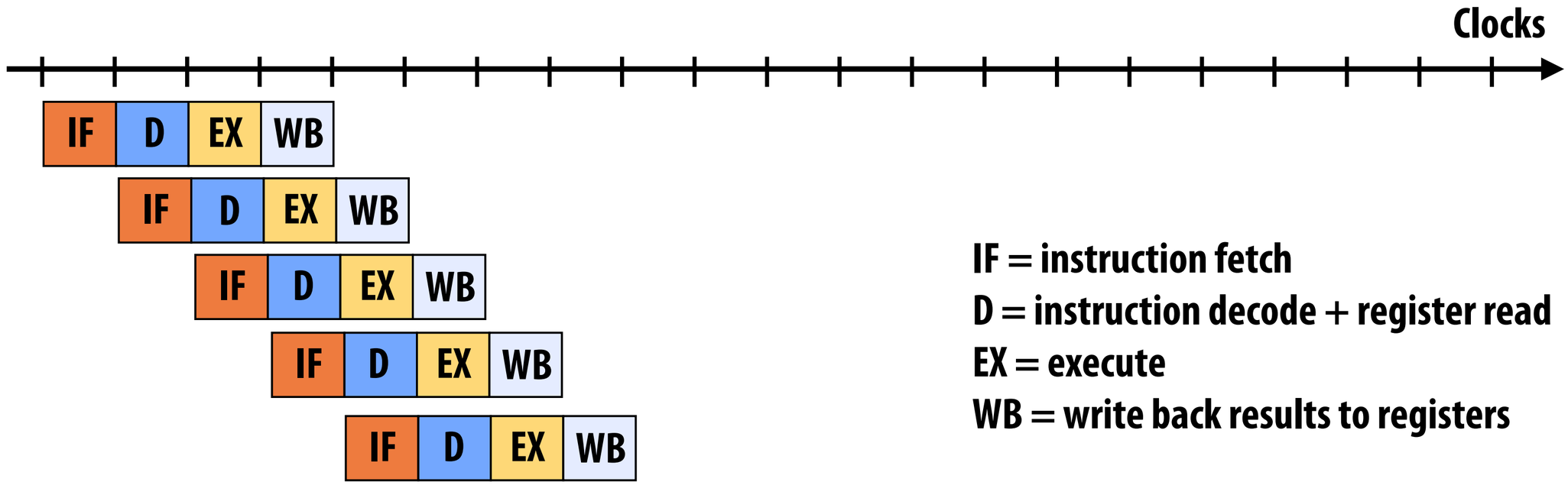
Latency: 1 load takes 2 hours

Throughput: 1 load/hour

An instruction pipeline

Break instruction execution down into many steps

Key to scaling CPU clock frequency (each clock, a simple short operation is done by each unit)



IF = instruction fetch
D = instruction decode + register read
EX = execute
WB = write back results to registers

Latency: 1 instruction takes 4 cycles
Throughput: 1 instruction per cycle
(Important: special care must be taken to ensure correctness in case of dependent instructions)

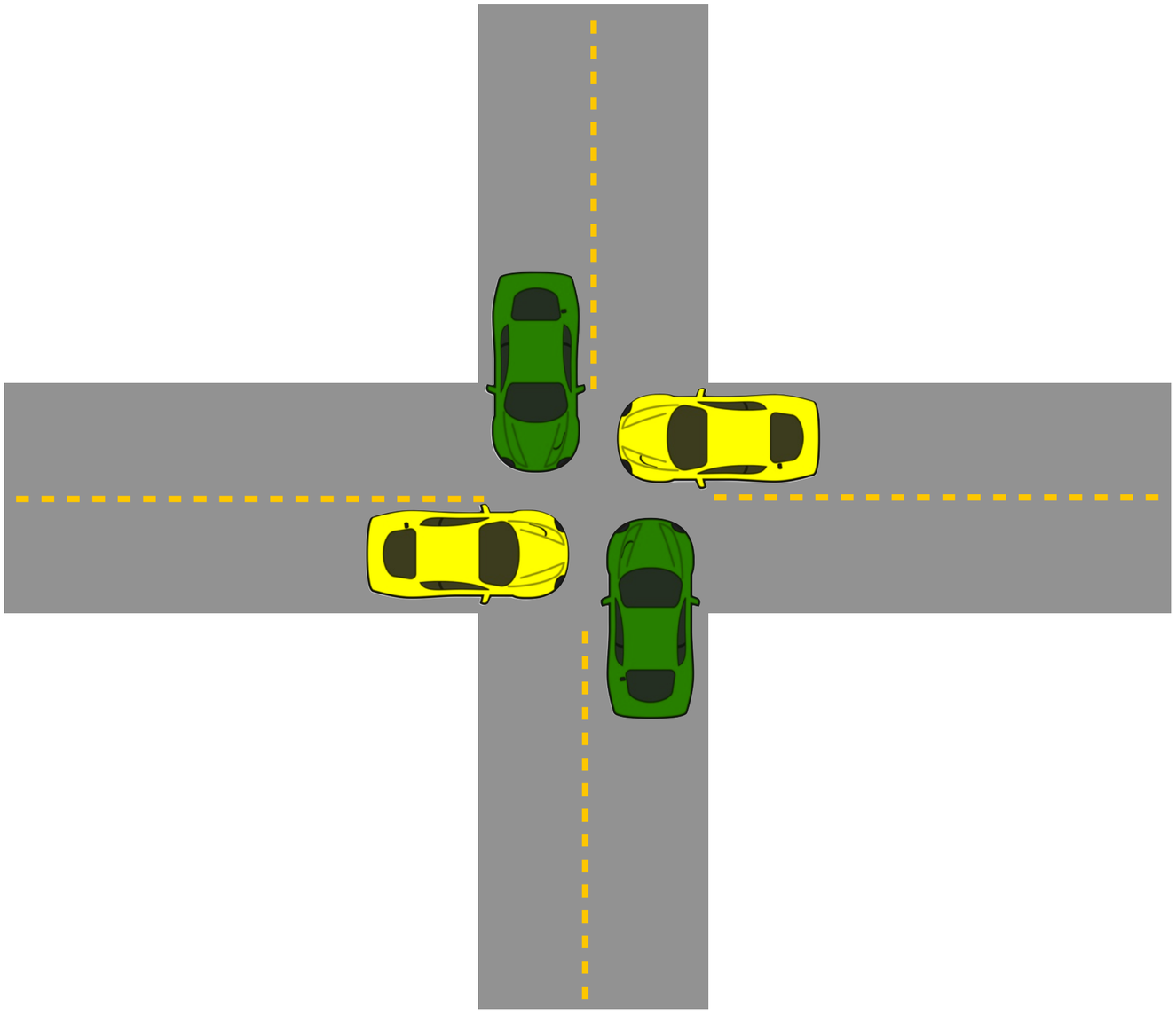
Modern Intel Core i7 pipeline is variable length ~15-20 stages

Correctness

Deadlock
Livelock
Starvation

(deadlock and livelock are clearly about correctness. Starvation is really an issue of fairness)

Deadlock



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 process to make progress unless some process relinquishes a resource ("backs up")

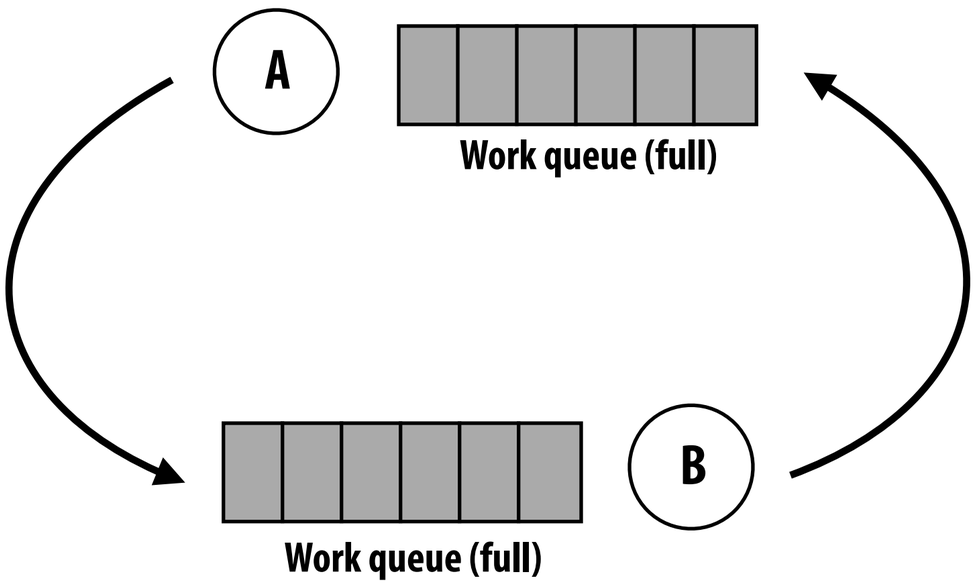
Non-technical side note:
Deadlock happens in Pittsburgh all the time
A great example is a bus turning right onto Morewood from 5th when cars on Morewood have creeped up past the white line.
(Deadlock can be amusing when a bus driver decides to let another driver know he has caused deadlock... "go take 418 you fool")

Deadlock

Credit: David Maitland, National Geographic

Deadlock in computer systems

Example 1:



A produces work for B's work queue

B produces work for A's work queue

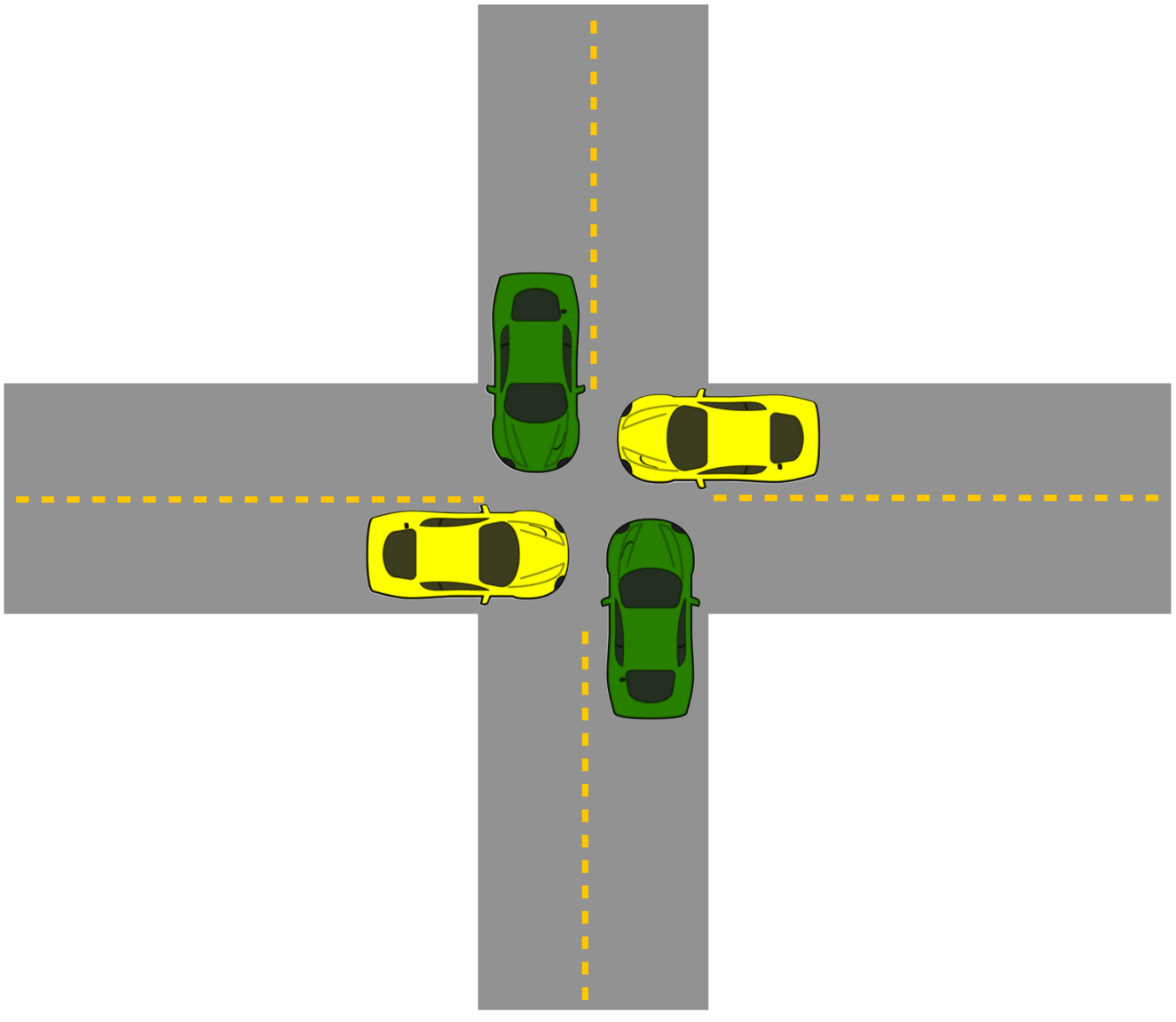
Queues are finite. Workers wait if no output space is available

Example 2:

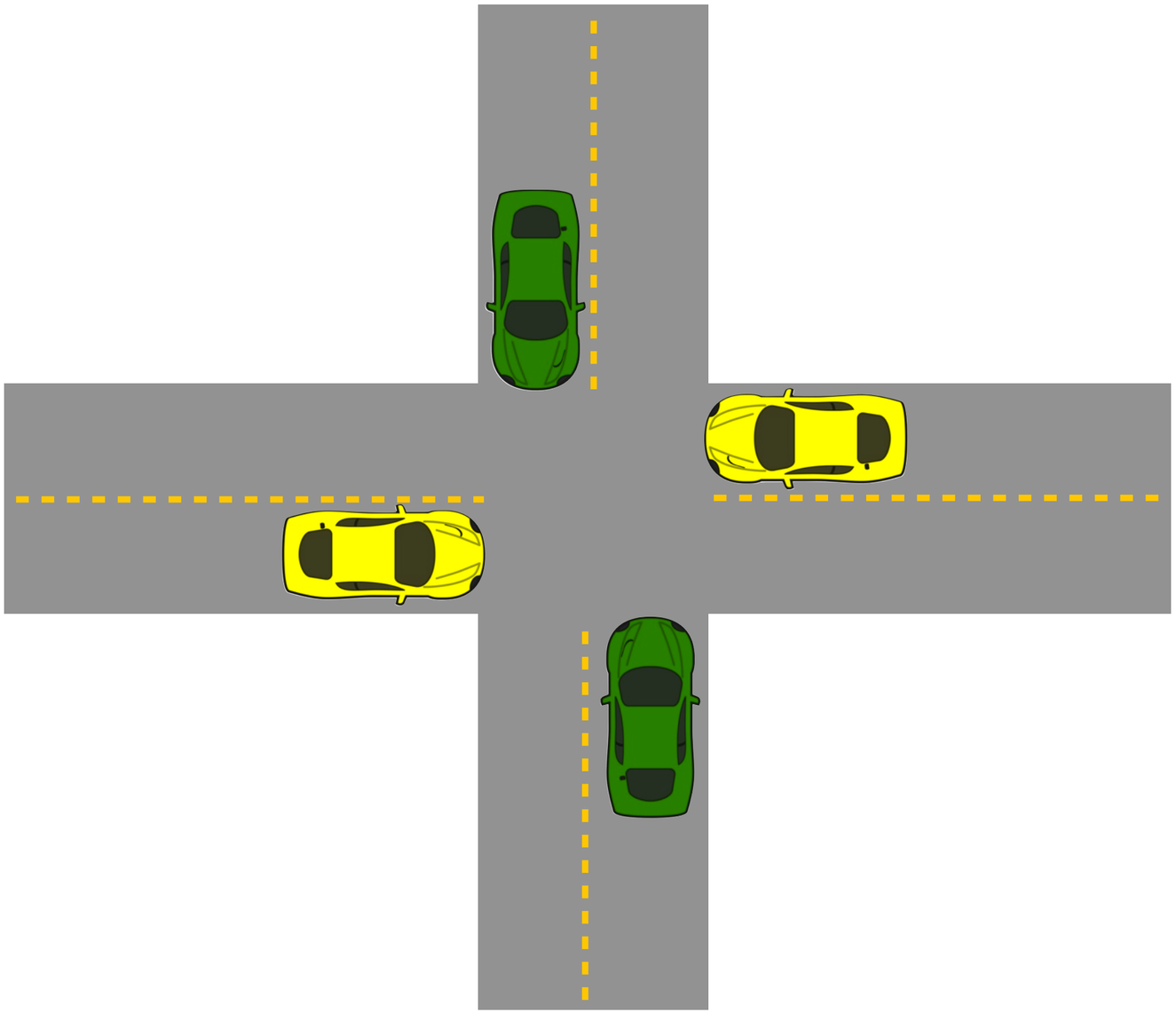
```
const int numEl = 1024;
float msgBuf1[numEl];
float msgBuf2[numEl];

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 right neighbor
Then receives message from left neighbor.

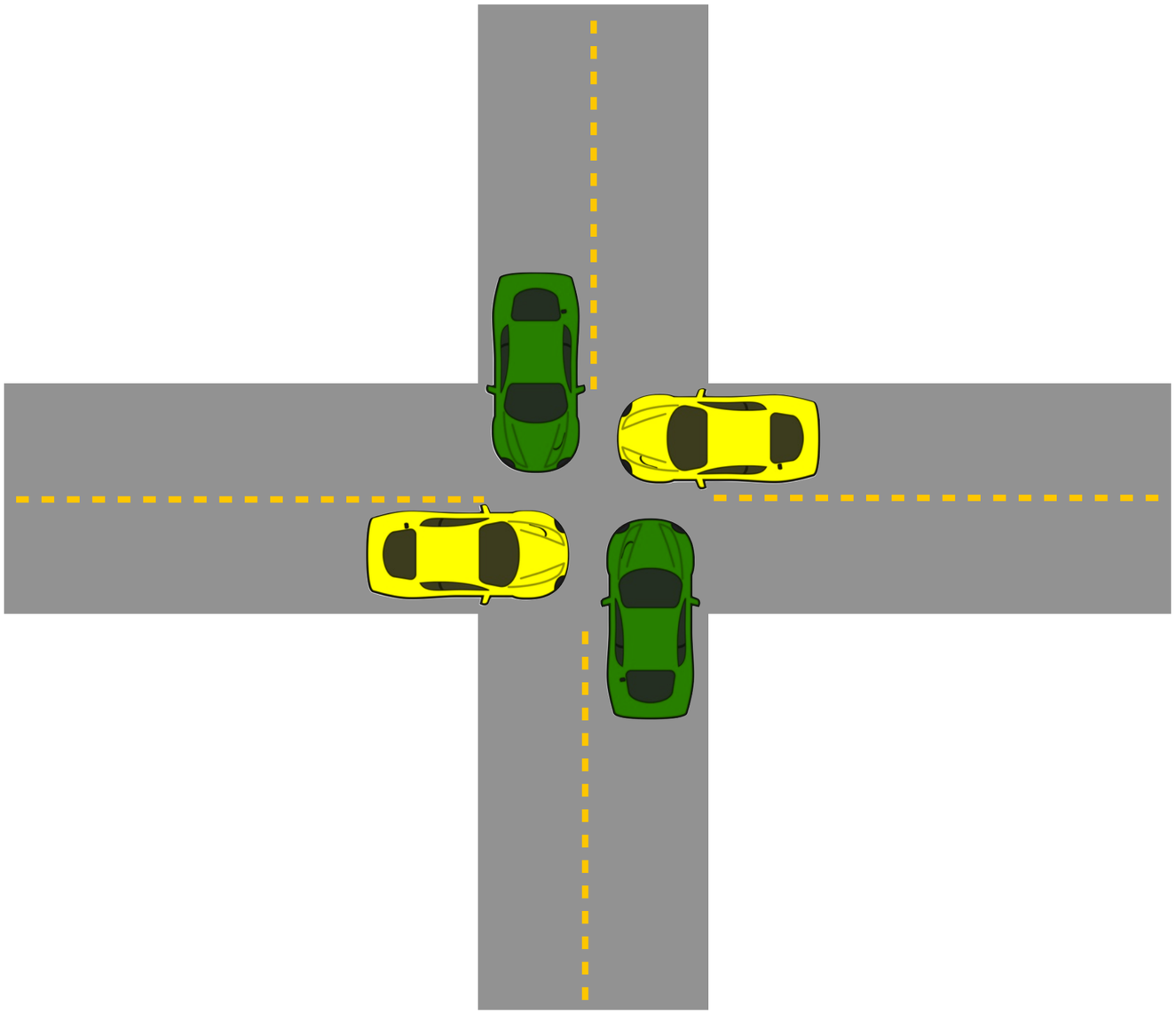
Livelock



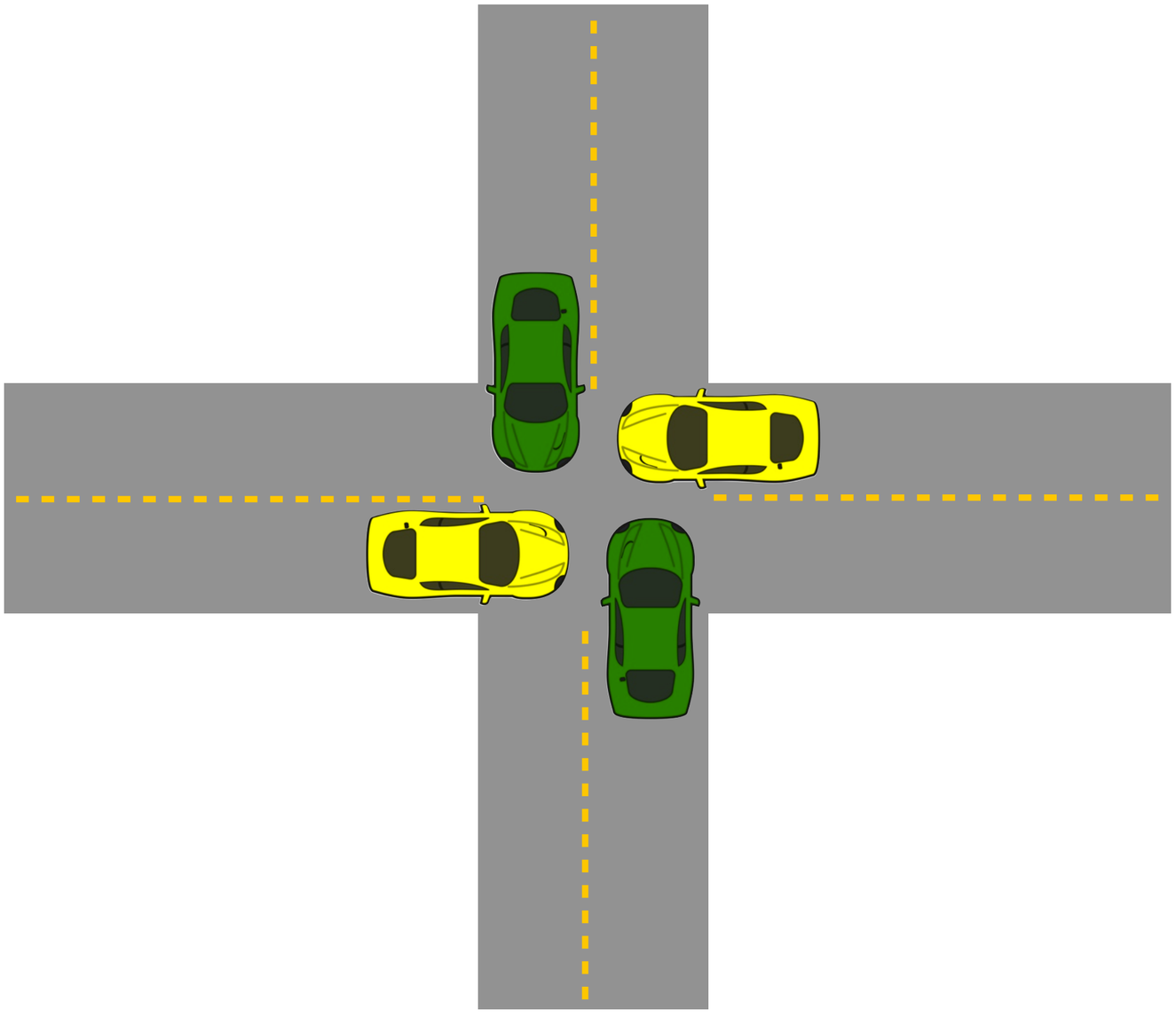
Livelock



Livelock



Livelock

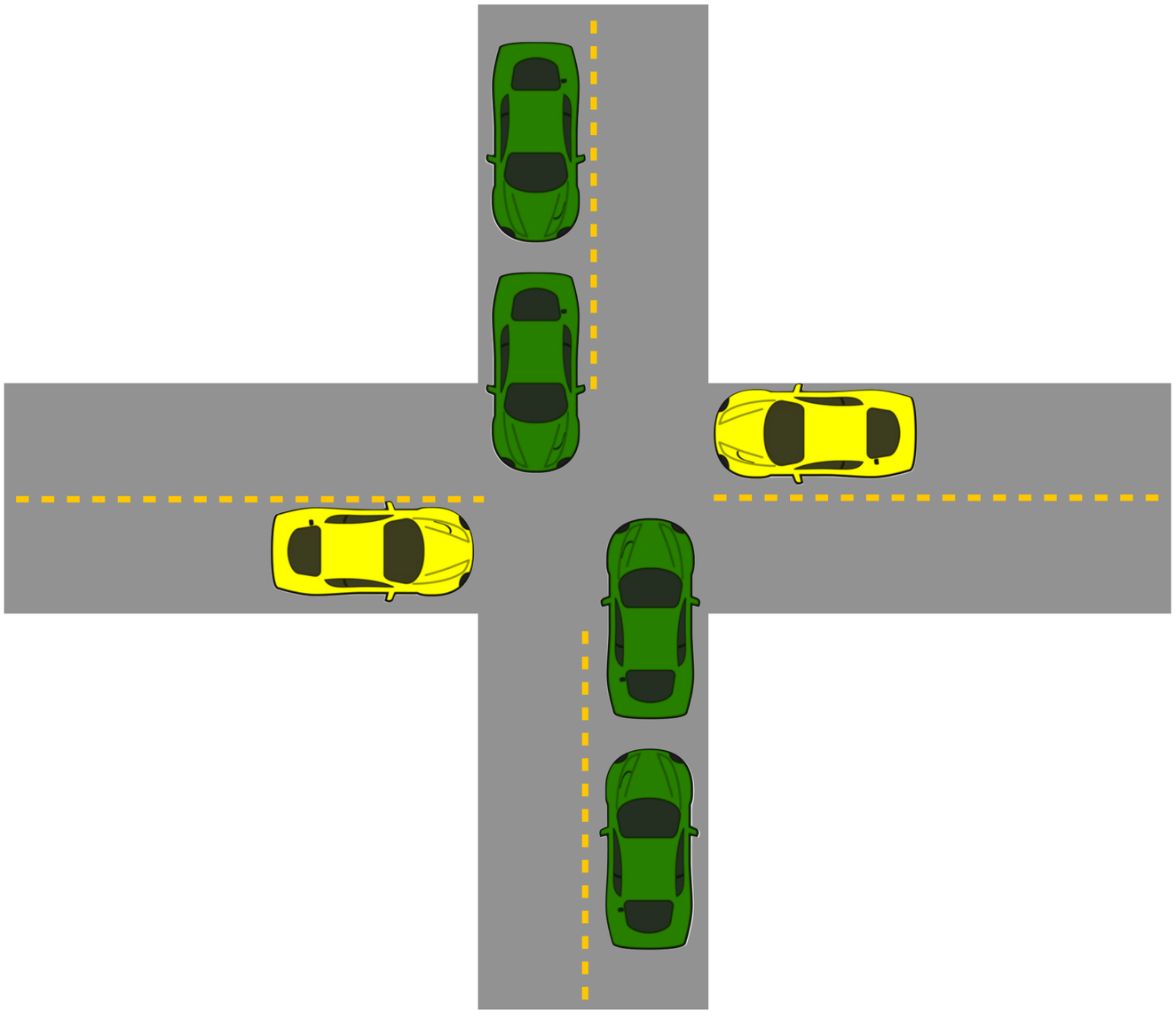


State where a system is executing many transactions/operations, but no process is making meaningful progress.

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

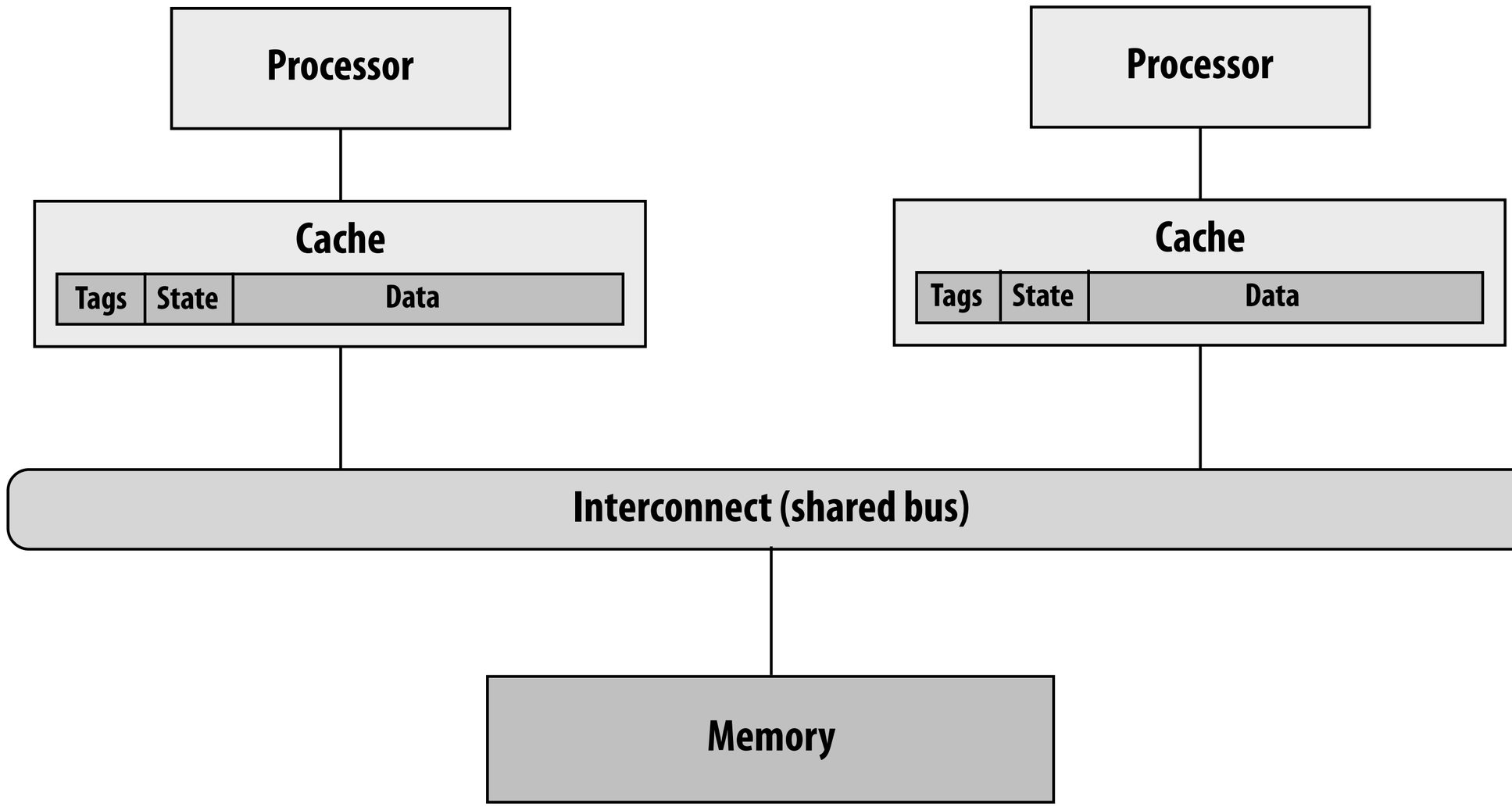
(as soon as green cars pass, yellow cars can go)

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 cache per CPU (write back)
- Interconnect is an atomic shared bus
- Cache can stall processor as its carrying out coherence operations



A basic cache miss on a uniprocessor

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

   [No matching tags, must read 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

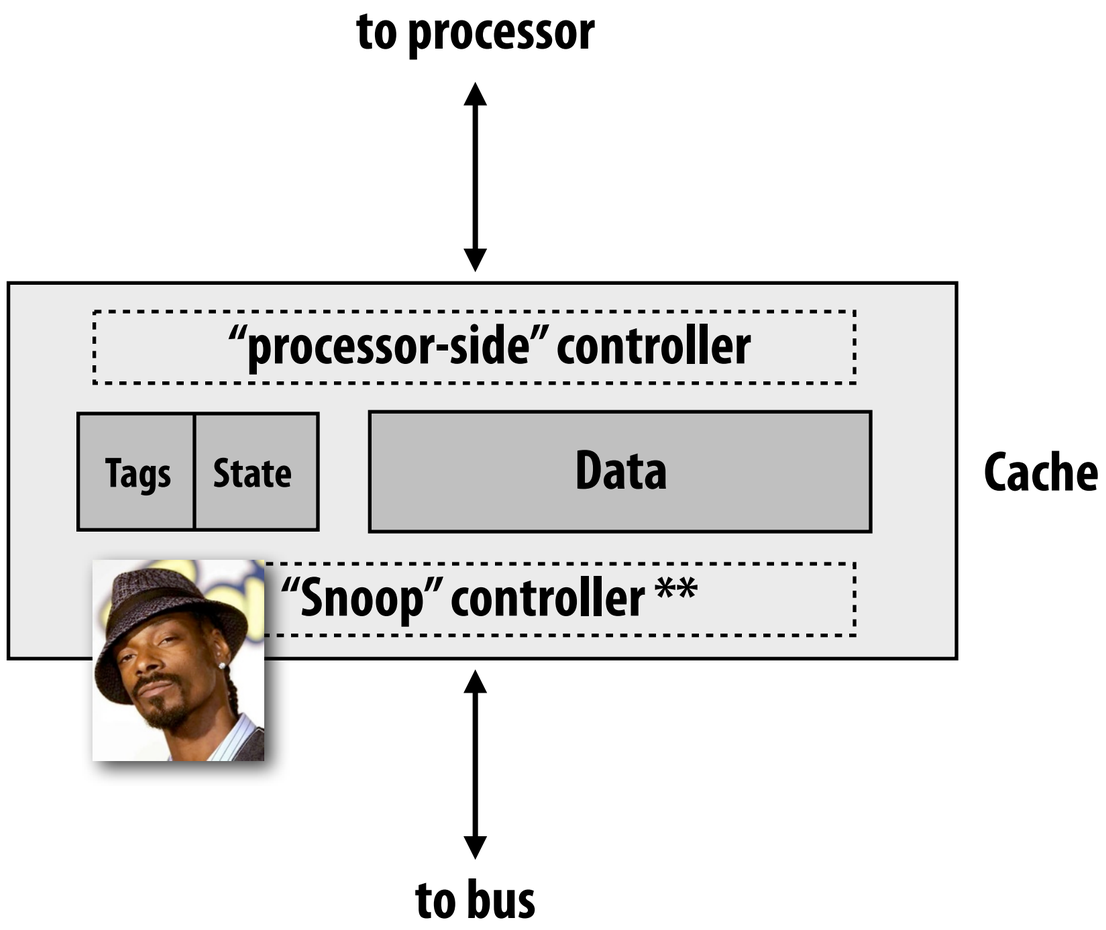
Multi-processor atomic bus:

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

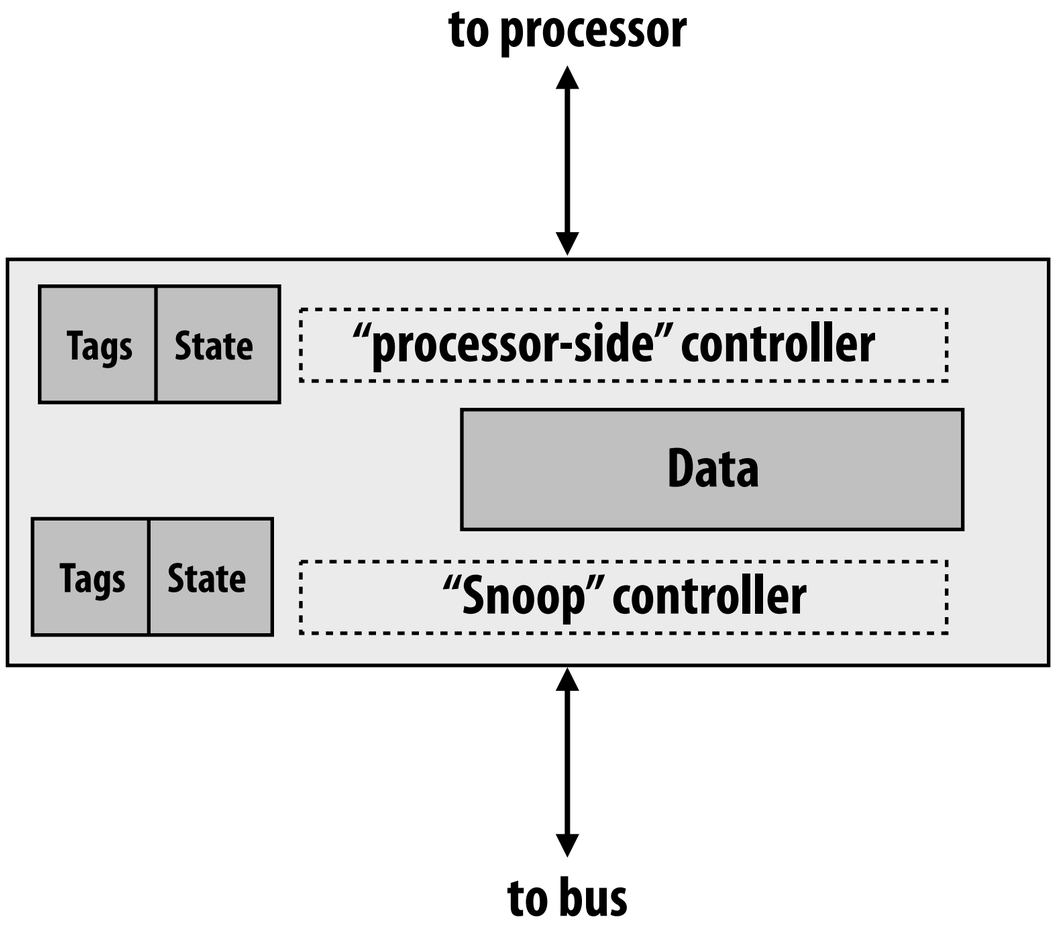


If bus receives priority:
During bus transaction, processor is locked out from cache.

If processor receives priority:
During processor cache access, cache can't respond with snoop

** 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

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

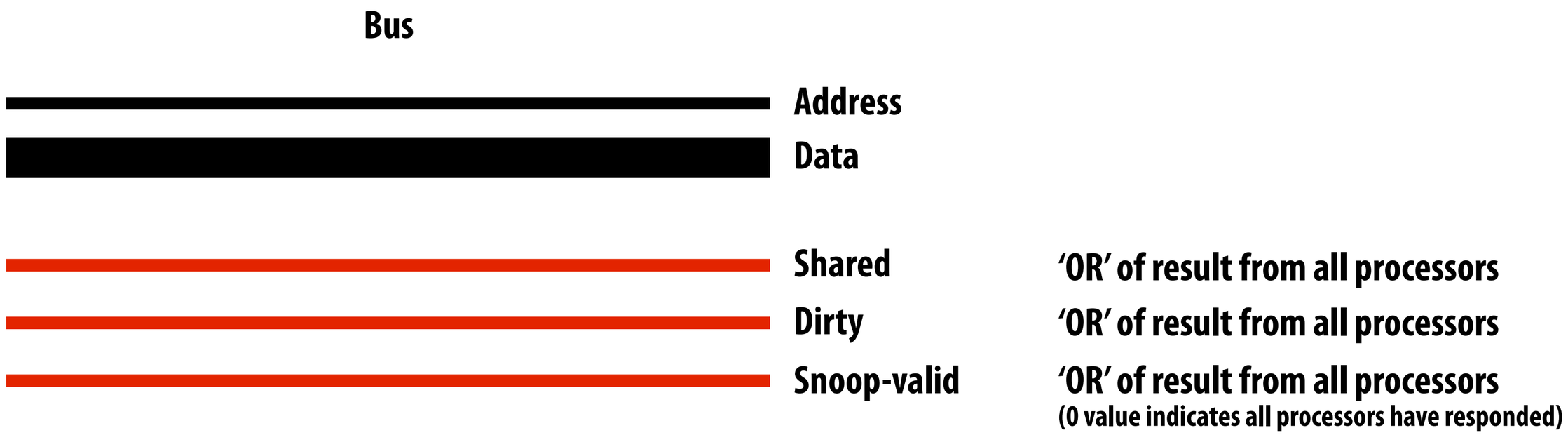
Memory needs to know what to do

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 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

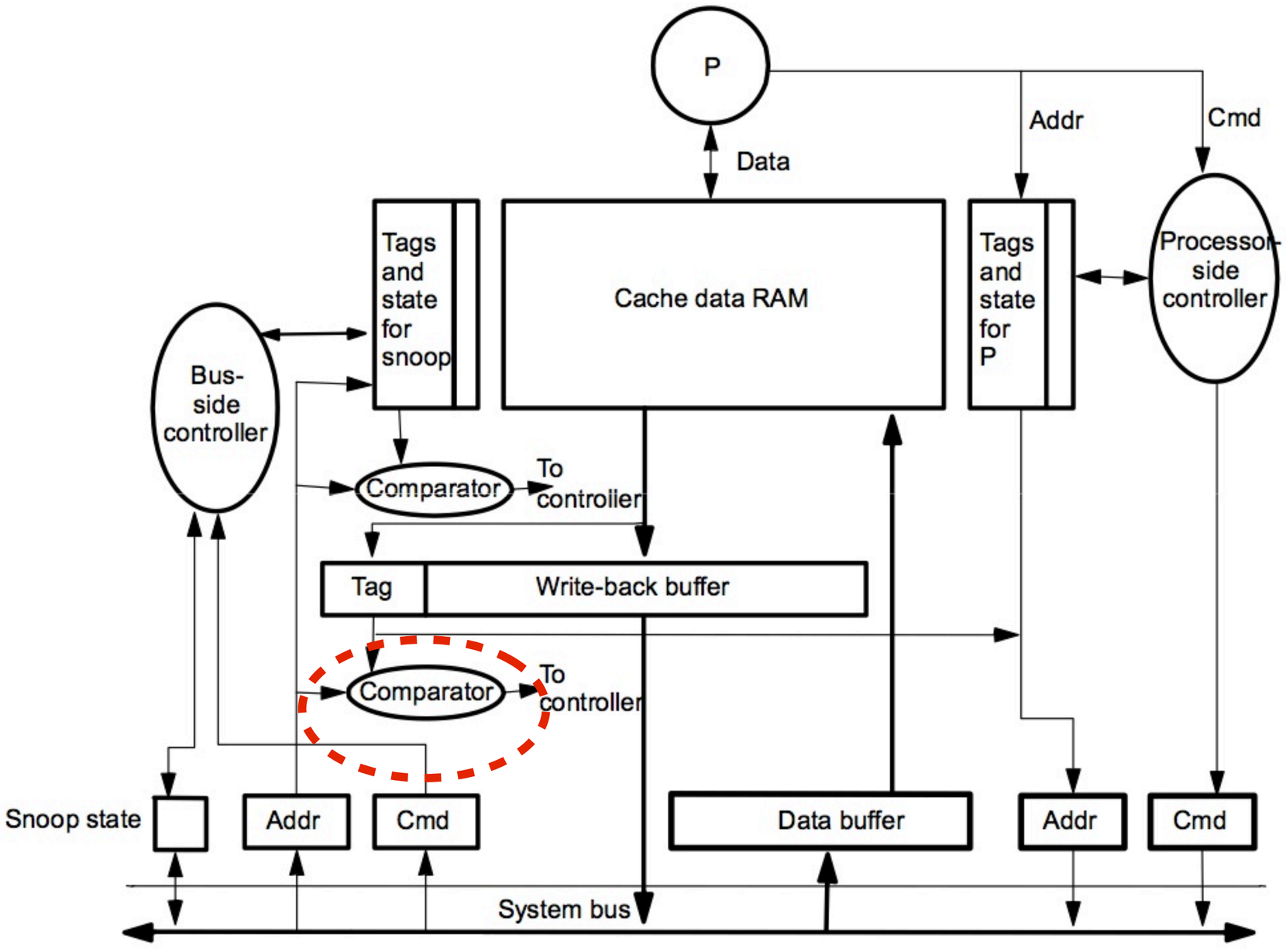
Handling write backs

- Write backs involve two bus transactions
  1. Incoming block (requested by processor)
  2. Outgoing block (dirty block to flush)

- 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 block to flush in buffer
  - Load requested block (allows processor to continue)
  - Flush contents of write-back buffer at 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?

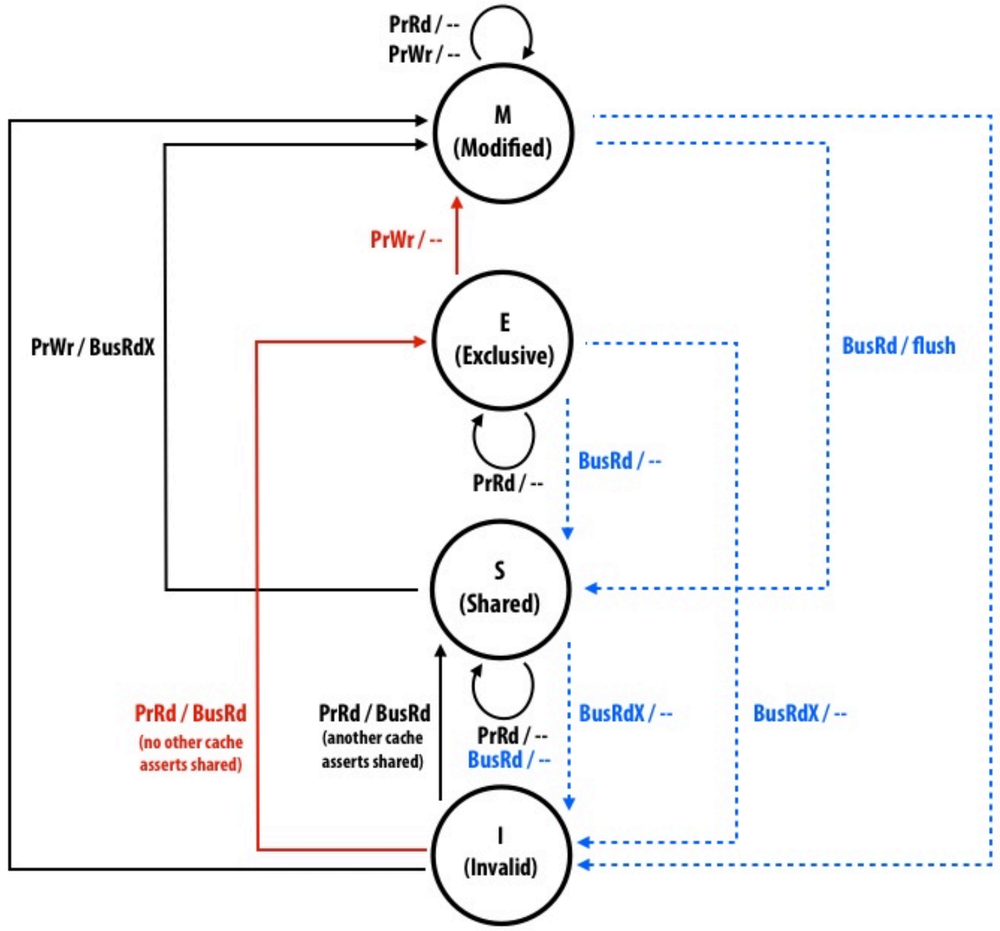
Snoop controller must check write back buffer address in addition to cache tags.

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
- Bus is atomic, but all 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

Example from book (EX: 6.1):

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

P1 wins bus access, 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

Fetch deadlock

P1 has a modified copy of cache block B

P1 is waiting for the bus to issue BusRdX on cache block 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 block B

P1 acquired bus, issues BusRdX

P2 invalidates

Before P1 performs write, P2 acquires bus, issues BusRdX

P1 invalidates

and so on…

To avoid livelock, 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
- FIFO arbitration
- Priority-based heuristics

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 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.

Source of the complexity: parallelism

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

- "Memory operations" that are abstracted by the architecture as atomic are implemented via multiple transactions involving all of these clients

- Performance optimization often entails splitting operations into more, 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)