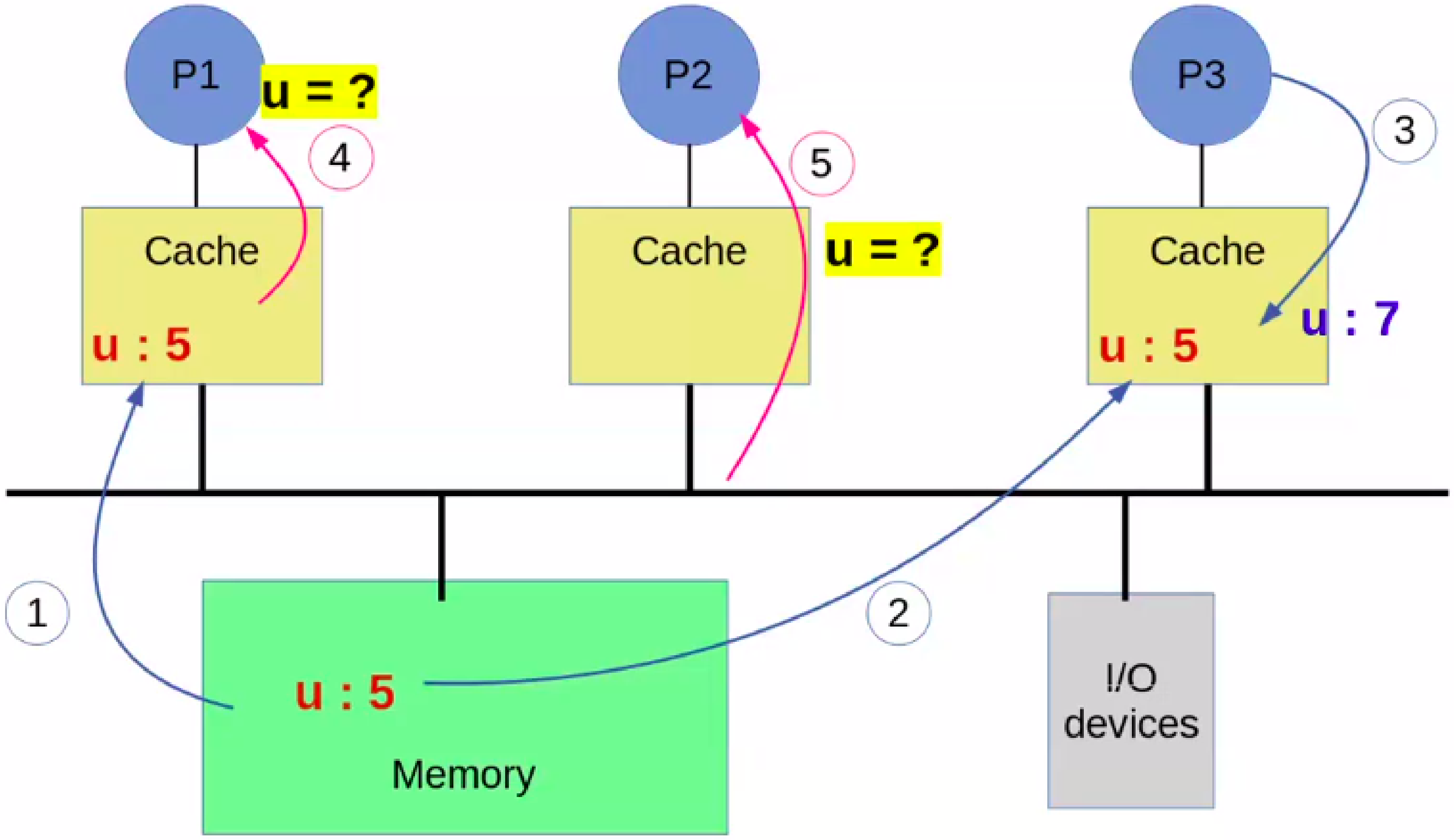
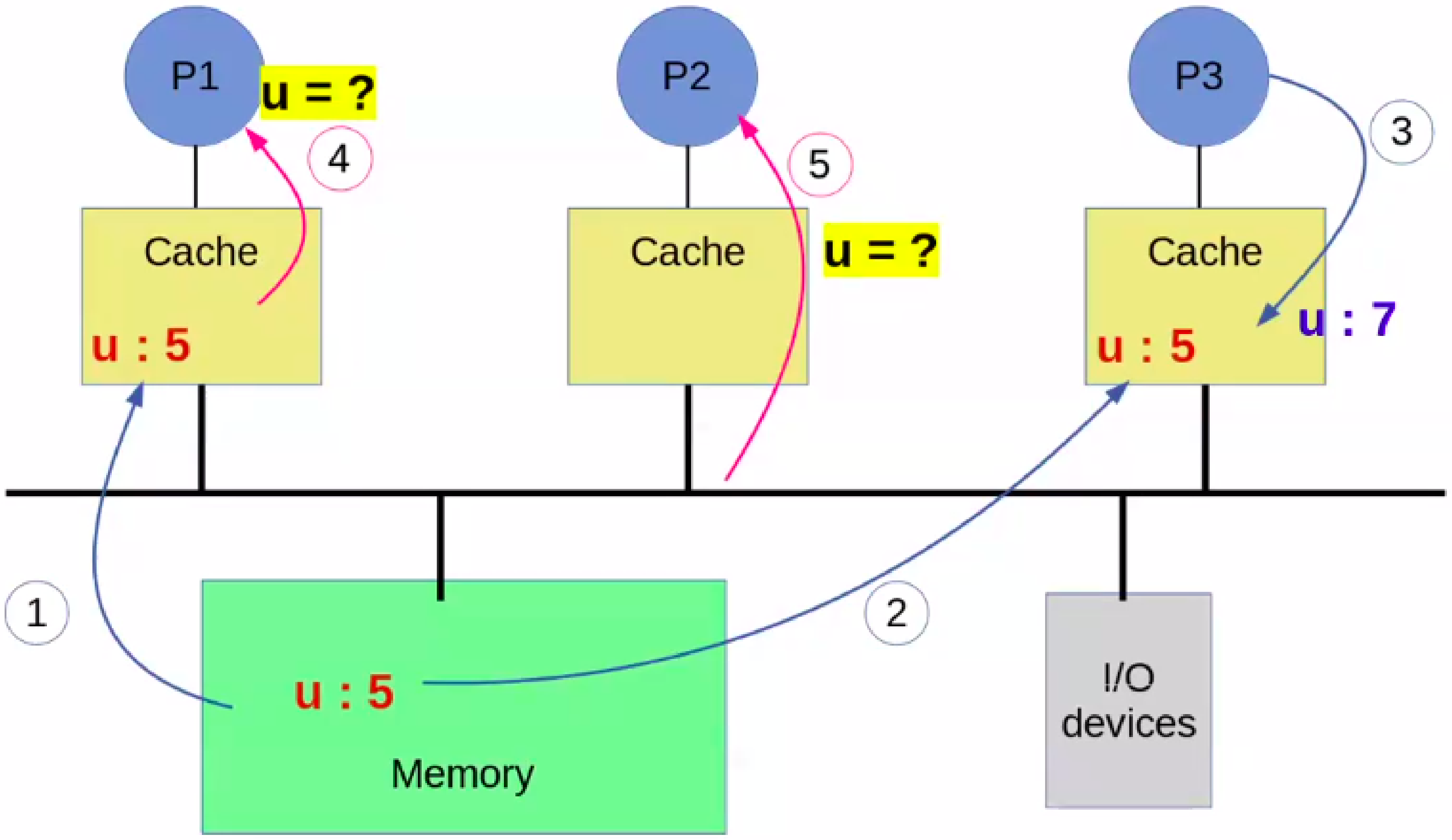
Cache Coherence

- Intuitive model of what a memory should do?
  - It should provide a set of locations that holds values
  - When a location is read it should return the latest value written to that location
  - Our shared address space based inter process communication also depends on this
- Private cache with processor creates problem
  - Copies of variable may be present in multiple caches
  - A write by one processor, may not become visible to others, in that others keep accessing stale data
- This is the cache coherence problem
- What to do about it?
  - Organise memory hierarchy to make it go away -or-
  - Detect and take actions to eliminate the problem

Ex: cache coherence problem



Ex: cache coherence problem



Cache coherence

- Processors see different values of ‘u’ after event-3
- If write-through cache then
  - <P3, Mem> updated ;
  - <P1> old value ; <P2> may get new value
- If write-back cache then
  - memory also has old value
  - <P1, P2, Mem> = old value
- In case write back of value happens after P1, P2 read value from memory
  - They get the old value
- In case two processor perform write back, then the final value that will reach the memory will depend on the order in which the processors replace the block containing u
- Unacceptable for programming, but this is frequent !

Cache coherence

- How to solve the coherence problem?
  - We don’t want to disallow caching
  - We don’t want to invoke OS at each reference to shared data
- Cache coherence has to be addressed as a basic hardware design issue
  - Can be done by eliminating other copies of the data when one copy is getting modified
  - Or by updating the other copies with new value
- Each read should return the latest value. How do we define latest value in a parallel system?

How to define latest value?

- Two processors might write to same location at same instant
- Or one processor might read so soon after the write by another processor that there is no time to propagate invalidation or update message
- Even in sequential program, “last” is not chronological or physical, it refers to last in the program order
- In parallel case we need to make sense of the collection of program orders

Formalisms

Concepts of program order and serialisation in sequential and parallel case

- We will discuss the concepts in
  - A uniprocessor - sequential
  - In multiprocessor setup - parallel

Prog-order + serial…uniprocessor

- Memory operations = read/write within an instruction are assumed to execute atomically with respect to each other in a specified order
- Memory operation issues when it leaves the processor
  - But goes through the cache, write-buffer, memory-modules
  - i.e. it takes long time to complete
- **Only way processor observes state of memory is by issuing memory operations, e.g. Reads**
- Ex: Processor knows that write operation is performed if subsequent read returns the written value or value of a later write
- Ex: Processor completes a read operation means that subsequent writes to that location cannot affect the read value
- "**Subsequent**" is well defined in sequential process => i.e. they are in program order

Prog-order + serial...parallel case

- Memory operations are same as sequential
- “subsequent” means something more here
- As we do not have one program order
- Rather several program orders interacting with the memory system
- To sharpen our idea of coherence
  - ASSUME: multi-cores, single memory, no caches
- All reads/writes will be directly performed by memory for all processors
- The memory imposes a serial order on accesses
- Also, the read/write acesses of individual process should be in program-order within this overall serial order