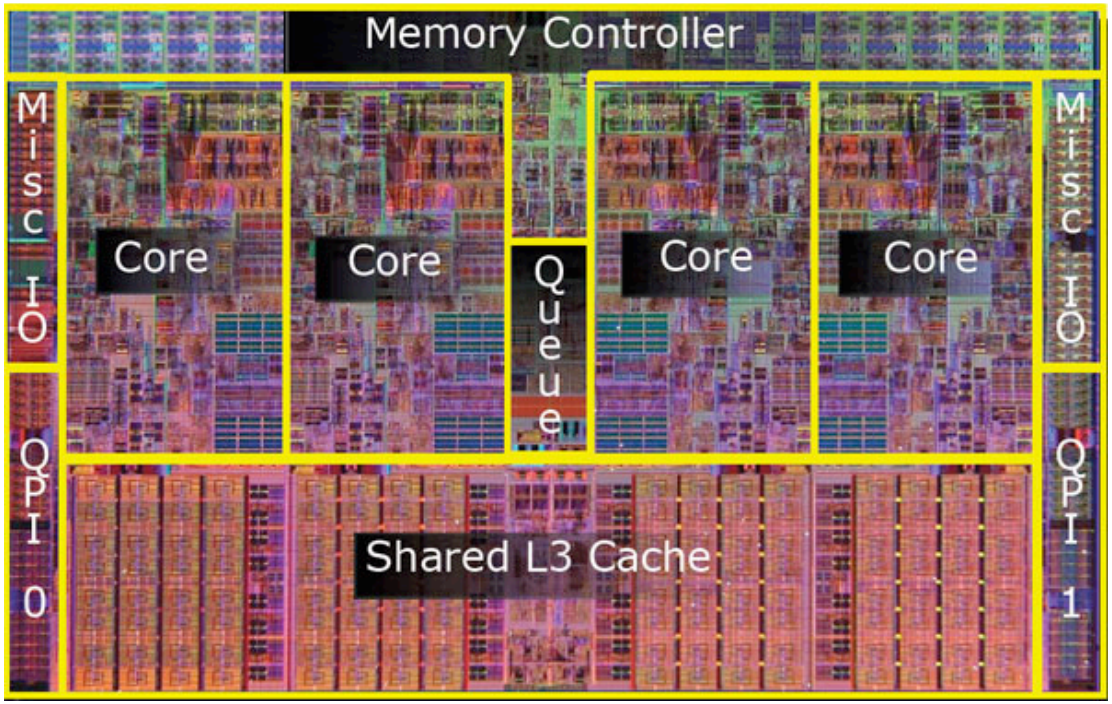
Threads and Cache Coherence in Hardware

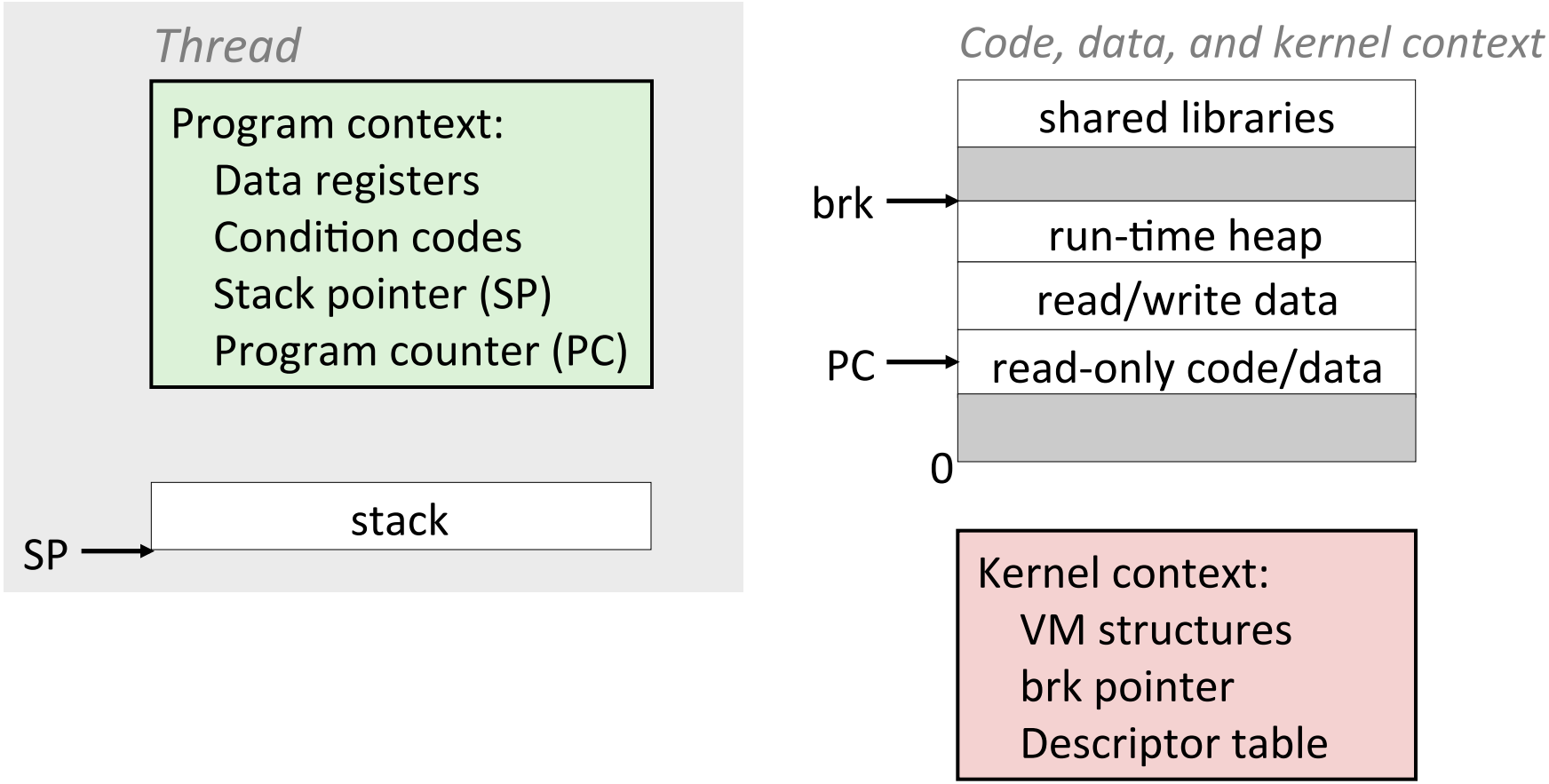
- Previously, we introduced multi-cores.
  - Today we'll look at issues related to multi-core memory systems:
    1. Threads
    2. Cache coherence



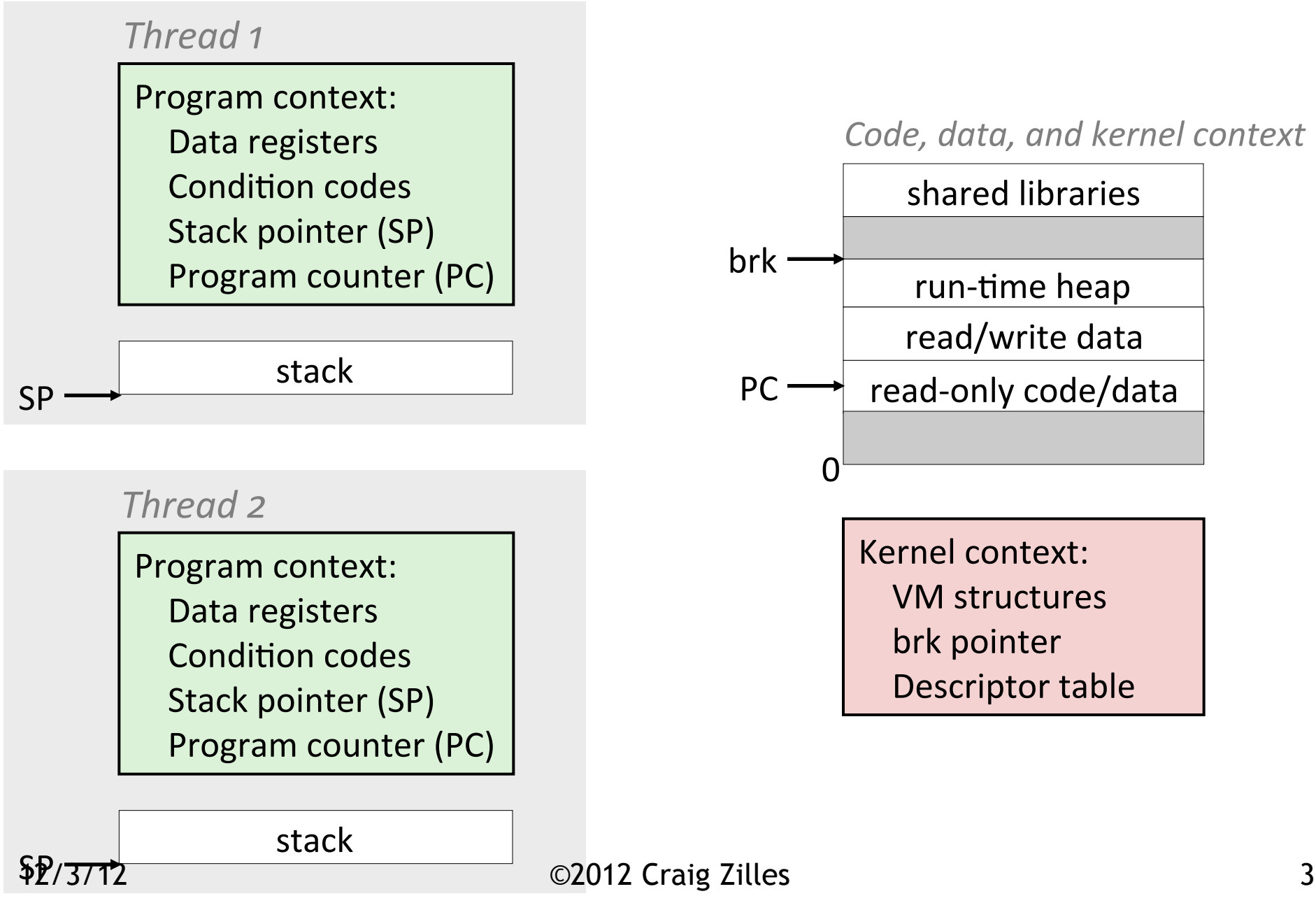
Intel Core i7

Process View

- Process = thread + code, data, and kernel context

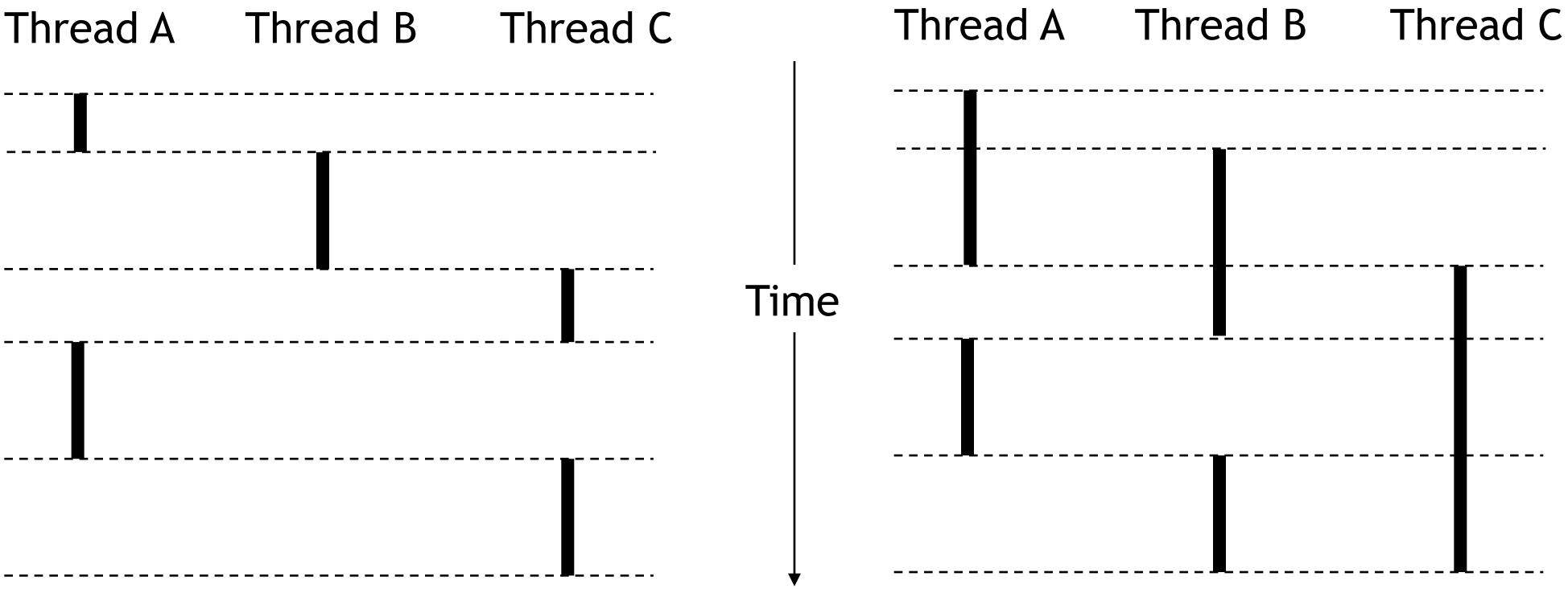


Process with Two Threads



Thread Execution

- Single Core Processor
  - Simulate concurrency by time slicing
- Multi-Core Processor
  - Can have true concurrency



How do threads from the same process communicate?

A) Their registers are kept in sync; when one thread writes a register, the written value is copied to the corresponding register in all cores that are running threads from the same process.

B) Threads have distinct register files, but can read and write each other register files directly.

C) Threads have distinct register files and must communicate through memory by one thread writing a memory location and later another thread reading that same memory location.

D) Threads have distinct registers and memory and must communicate through network protocols like TCP/IP.

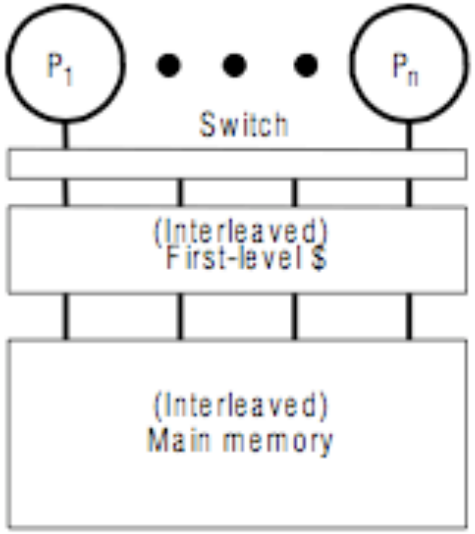
(Hardware) Shared Memory

- The programming model for multi-core CPUs

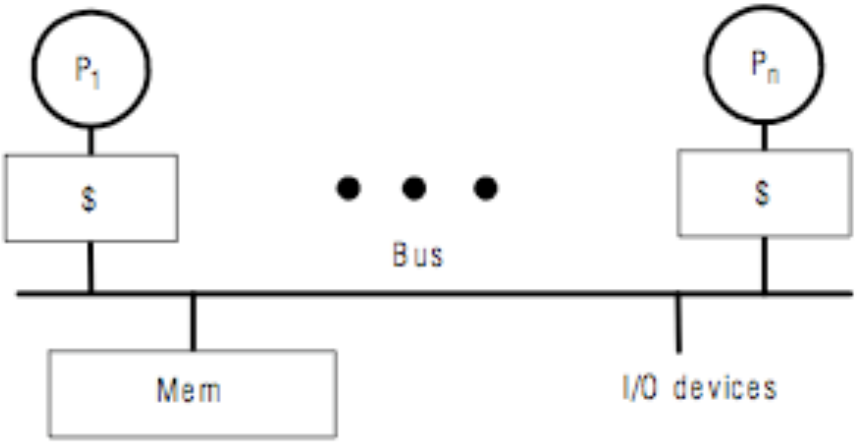
What is it?

- Memory system with a single global physical address space
  - So processors can communicate by reading and writing same memory
- Design Goal 1: Minimize memory latency
  - Use co-location & caches
- Design Goal 2: Maximize memory bandwidth
  - Use parallelism & caches

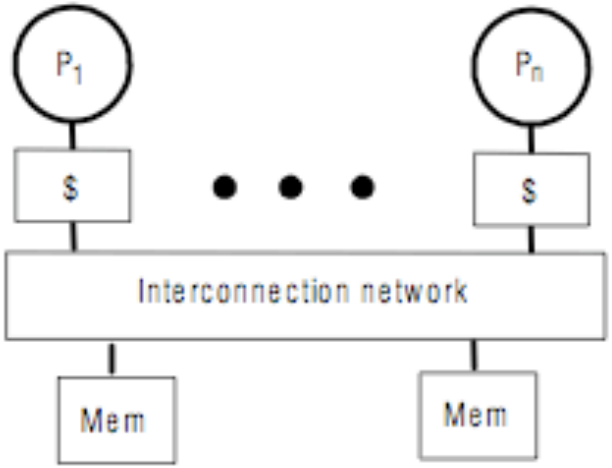
Some example memory-system designs



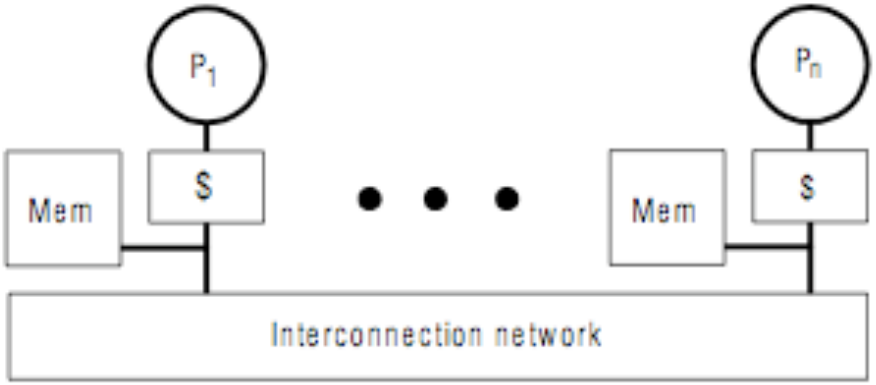
(a) Shared cache



(b) Bus-based shared memory

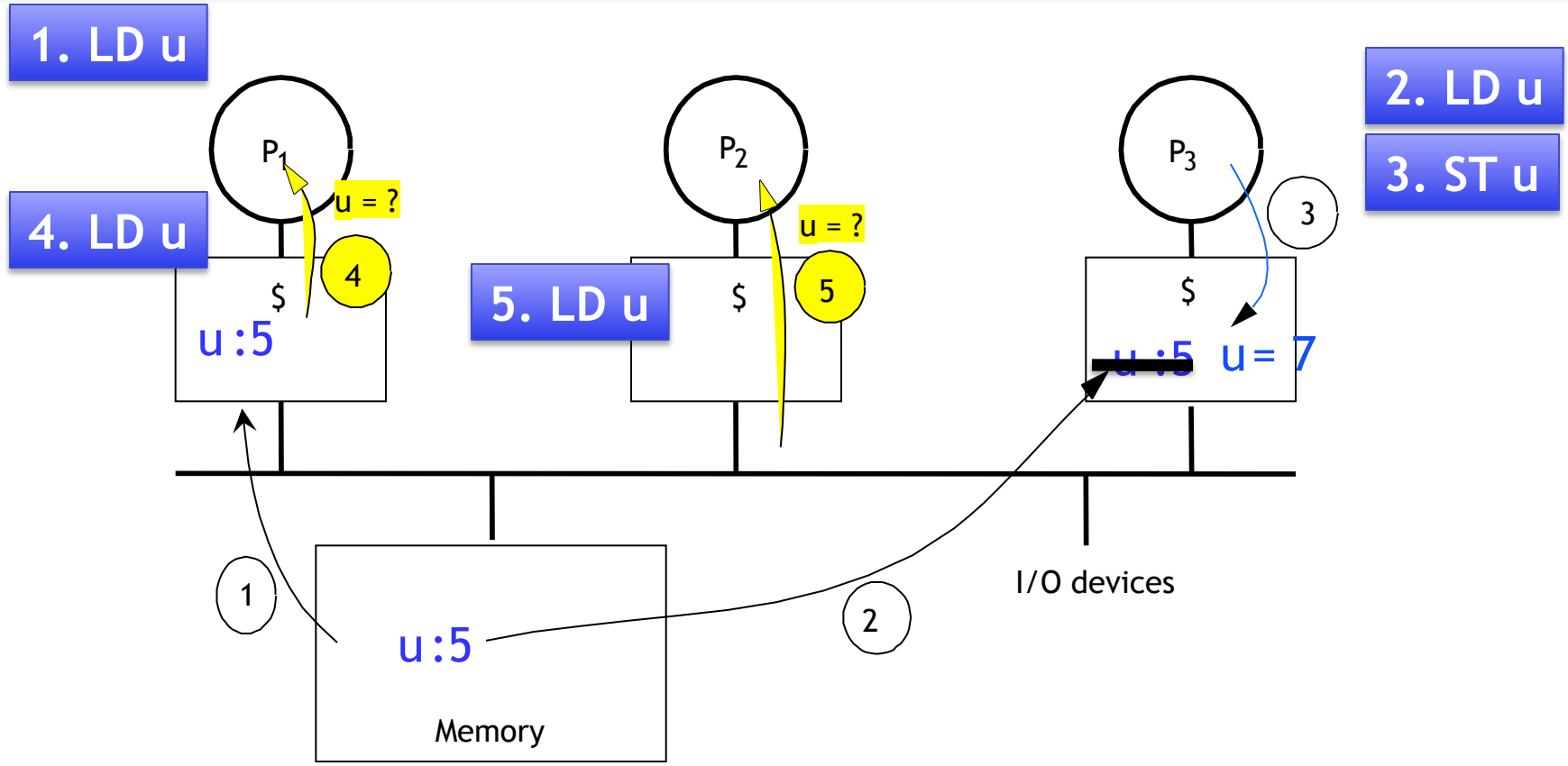


(c) Dancehall



(d) Distributed-memory

The Cache Coherence Problem



- Multiple copies of a block can easily get inconsistent
  - Processor writes; I/O writes
- Processors could see different values for u after event 3

Cache Coherence

- According to Webster's dictionary …
  - Cache: a secure place of storage
  - Coherent: logically consistent

- **Cache Coherence:** keep storage logically consistent
  - Coherence requires enforcement of 2 properties

1. Write propagation
   - All writes eventually become visible to other processors
2. Write serialization
   - All processors see writes to same block in same order

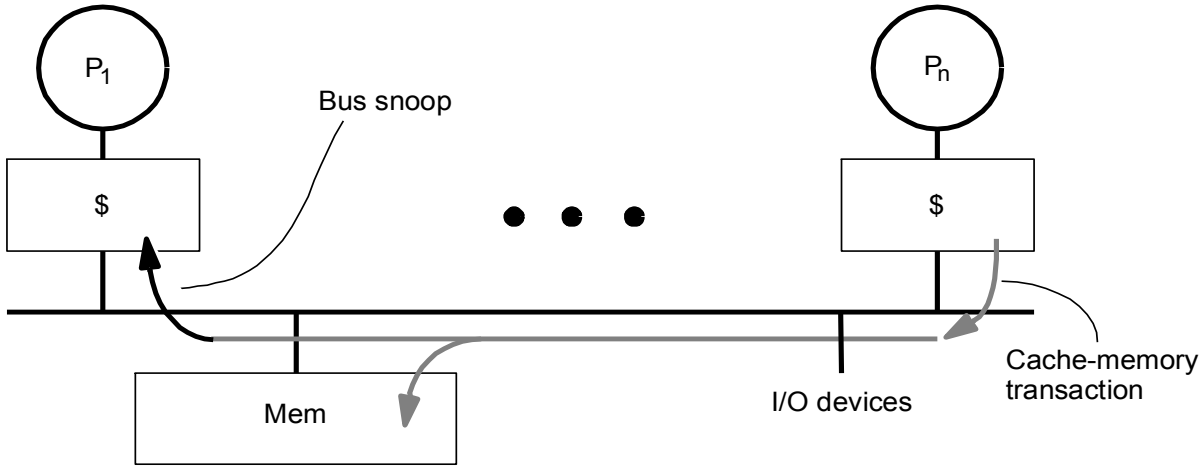
Cache Coherence Invariant

- Each block of memory is in exactly one of these 3 states:

1. **Uncached**: Memory has the only copy
2. **Writable**: Exactly 1 cache has the block and only that processor can write to it.
3. **Read-only**: Any number of caches can hold the block, and their processors can read it.

invariant | inˈve(ə)rēənt |
noun Mathematics
a function, quantity, or property that remains unchanged when a specified transformation is applied.

Snoopy Cache Coherence Schemes



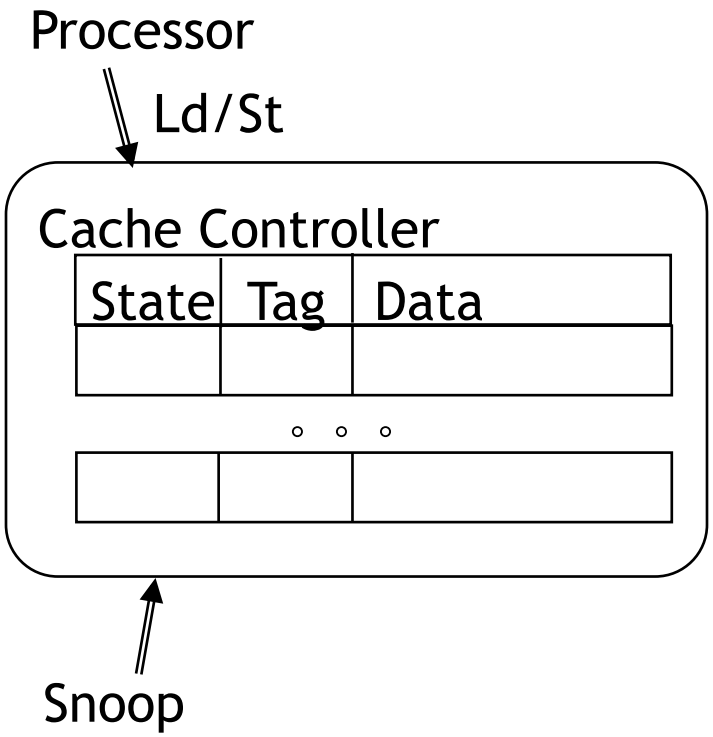
- Bus is a broadcast medium & caches know what they have

- Cache controller "snoops" all transactions on the shared bus
  - Relevant transaction if for a block it contains
  - Take action to ensure coherence
    - Invalidate or supply value

  Depends on state of the block and the protocol

Maintain the invariant by tracking "state"

- Every cache block has an associated state
  - This will supplant the valid and dirty bits
- A **cache controller** updates the state of blocks in response to processor and snoop events and generates bus transactions
- Snoopy protocol
  - set of states
  - state-transition diagram
  - actions



MSI protocol

This is the simplest possible protocol, corresponding directly to the 3 options in our invariant

- **Invalid State**: the data in the cache is not valid.
- **Shared State:** multiple caches potentially have copies of this data; they will all have it in **shared** state. Memory has a copy that is consistent with the cached copy.
- **Dirty or Modified:** only 1 cache has a copy. Memory has a copy that is inconsistent with the cached copy. Memory needs to be updated when the data is displaced from the cache or another processor wants to read the same data.

Actions

Processor Actions:

- **Load**
- **Store**
- **Eviction**: processor wants to replace cache block

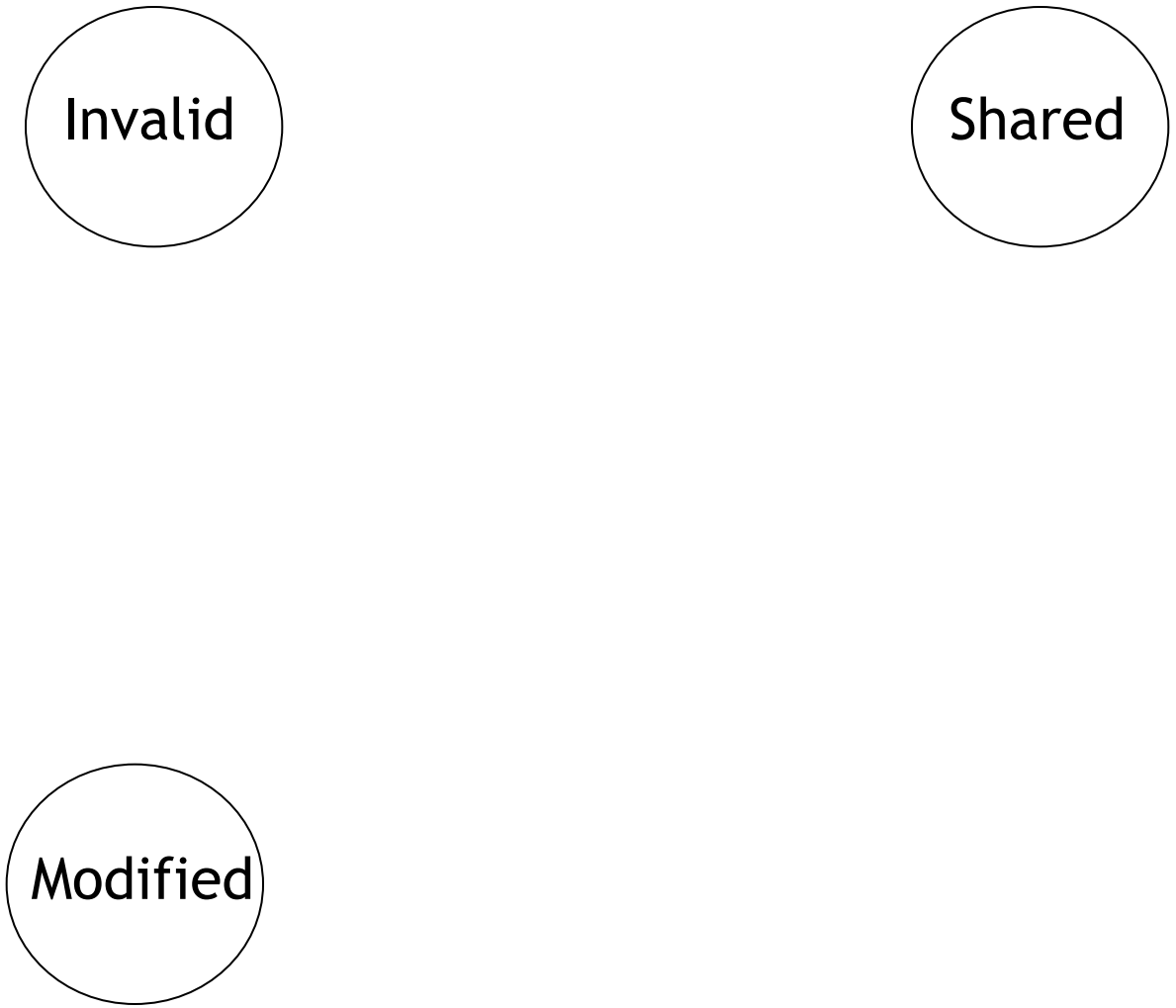
Bus Actions:

- **GETS**: request to get data in shared state
- **GETX**: request for data in modified state (*i.e.*, eXclusive access)
- **UPGRADE**: request for exclusive access to data owned in shared state

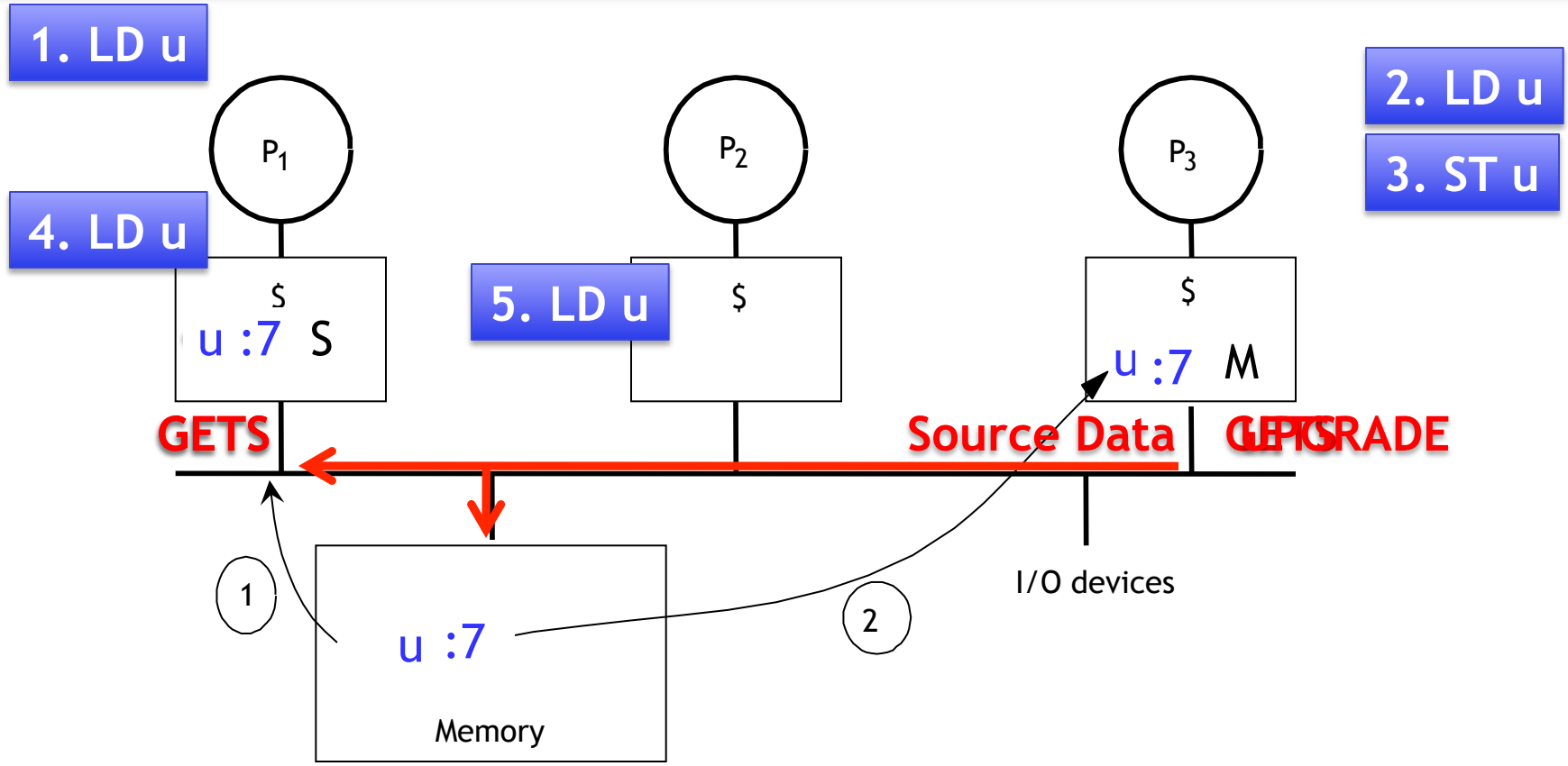
Cache Controller Actions:

- **Source Data**: this cache provides the data to the requesting cache
- **Writeback**: this cache updates the block in memory

MSI Protocol



The Cache Coherence Problem Solved



Real Cache Coherence Protocols

- Are more complex than MSI (see MESI and MEOSI)
- Some modern chips don't use buses (too slow)
  - Directory based: Alternate protocol doesn't require snooping
- But this gives you the basic idea.

But wait, there's more! (Memory Consistency)

- Cache coherence isn't enough to define shared memory behavior
  - Coherence defines ordering of requests that access same bytes
    - serializes all operations to that location such that,
    - operations performed by any processor appear in program order
      - Program order = order defined by assembly code
    - A read gets the value written by last store to that location

- Memory Consistency
  - **Defines allowed orderings for reads/writes to different locations**
  - Your intuition is wrong.
  - One of the most difficult topics in CS.
  - I'm going to try to make you aware of the issues.

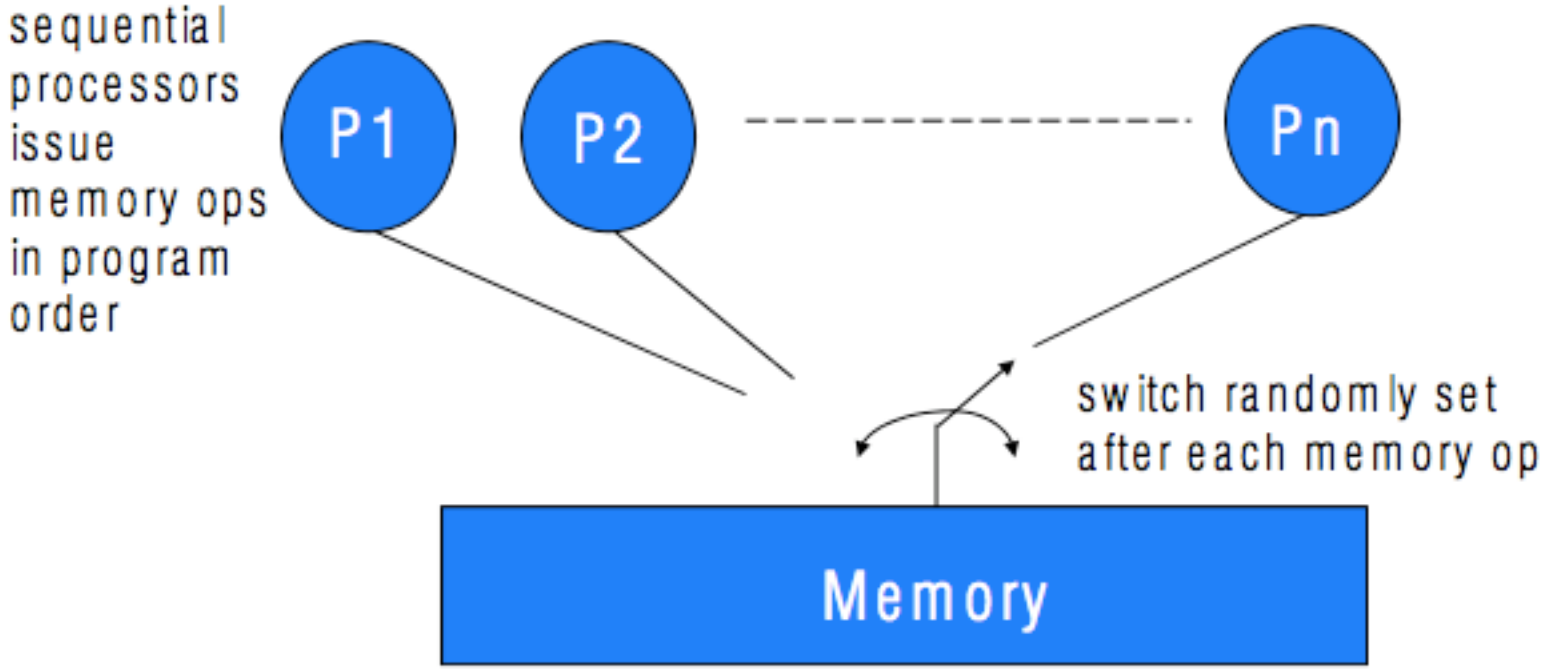
What is the final value of A & B?

/* initially, A = B = flag = 0 */

P1	P2
A = 1;	while (flag == 0); /* spin */
B = 1;	print A;
flag = 1;	print B;

Sequential Consistency

- **Leslie Lamport 1979:**
- "A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program order specified by its program"



- Is not what you get on almost any computer. Why?

Weak Consistency Models

- Done for performance; fewer constraints
- Many such models: e.g., Total Store Ordering, Release Consistency
  - Not necessary for most programmers to know specifics

- Key idea:
  - Fences: allow programmers to specify ordering constraints
    - Hardware instructions
    - E.g., All memory operations before fence must complete before any after the fence.

/* initially, A = B = flag = 0 */

P1	P2
A = 1;	while (flag == 0); /* spin */
B = 1;	fence;
fence;	print A;
flag = 1;	print B;

Fences in software practice

- Most programmers don't need to write their own fences:

- Standard lock libraries include the necessary fences:
  - If you only touch shared data within locked critical sections, the necessary fences will be present.

- If you want to define your own synchronization variables (wait flags)
  - Some languages include keywords that introduce necessary fences
    - Java 5's "volatile" keyword
    - http://www.javamex.com/tutorials/synchronization_volatile_java_5.shtml

- Big takeaway:
  - Be very careful building your own synchronization.

Conclusions

- CPU-based multi-cores use (hardware) shared memory
  - All threads can read/write same physical memory locations
  - Per processor caches for low latency, high bandwidth

- Cache coherence:
  - All writes to a location seen in same order by all processors
  - "Single writer **or** multiple readers" invariant
  - Modified, Shared, Invalid (MSI) protocol

- False sharing:
  - Cache coherence at the granularity of cache blocks
  - Minimize intersection of processors data working set
    - Although read-only sharing is okay

- Memory Consistency:
  - Your intuition is wrong! Beware creating your own synchronization!

A simple piece of code

```
unsigned counter1 = 0, counter2 = 0;

void *do_stuff(void * arg) {
  for (int i = 0 ; i < 200000000 ; ++ i) {
     counter1 ++;
     counter2 ++;
  }
  return arg;
}
```

How long does this program take?

How can we make it faster?

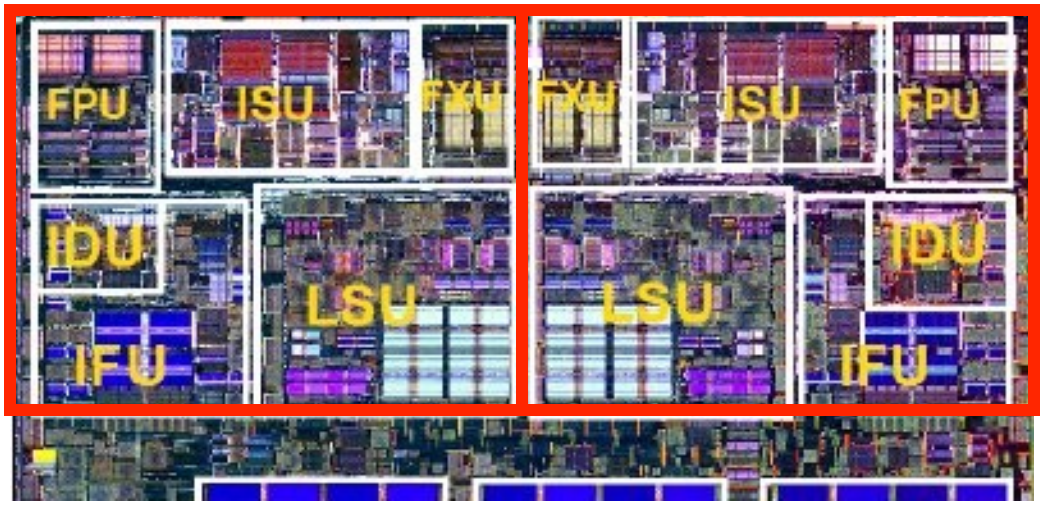
Exploiting a multi-core processor

```
unsigned counter1 = 0, counter2 = 0;

void *do_stuff(void * arg) {
  for (int i = 0 ; i < 200000000 ; ++ i) {
      counter1 ++;
      counter2 ++;
  }
  return arg;
}
```

Split for-loop into two loops, one for each statement.
Execute each loops on separate cores

#1



#2

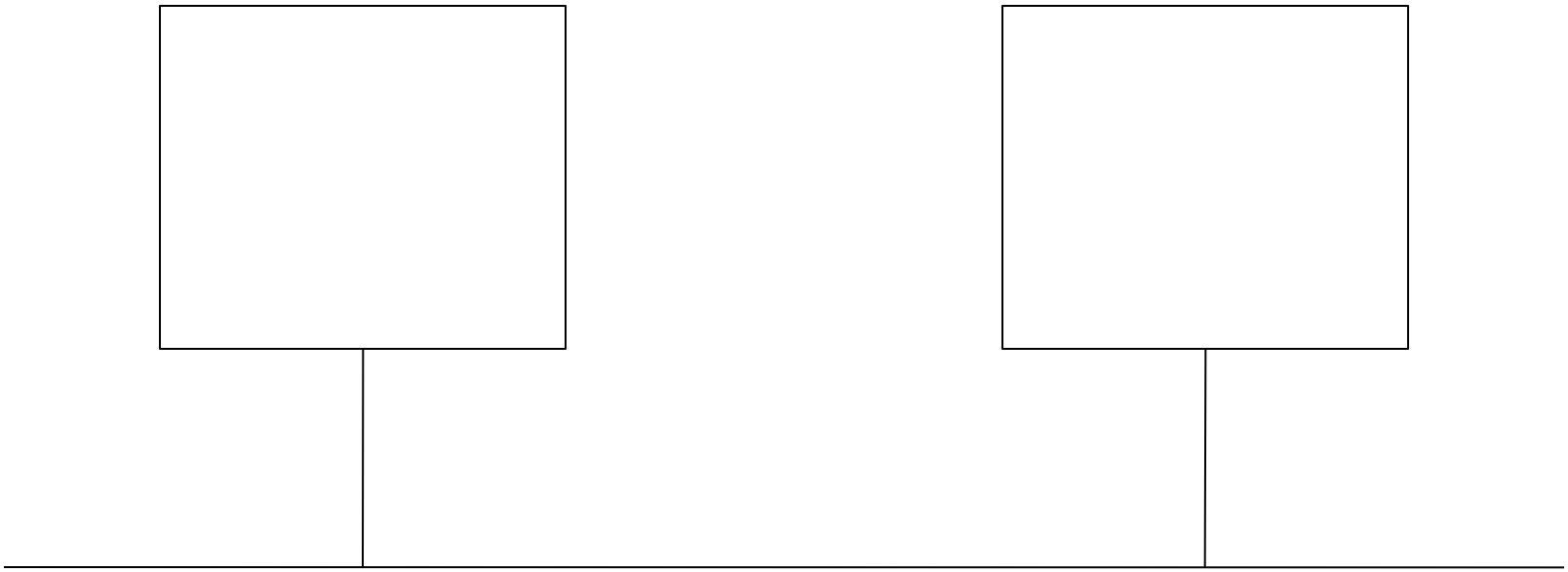
Parallelized.

```
unsigned counter1 = 0, counter2 = 0;

void *do_stuff1(void * arg) {
  for (int i = 0 ; i < 200000000 ; ++ i) {
     counter1 ++;
  }
  return arg;
}
void *do_stuff2(void * arg) {
  for (int i = 0 ; i < 200000000 ; ++ i) {
     counter2 ++;
  }
  return arg;
}
```

How much faster?

What is going on?



Which is better?

- If you wanted to parallelize work on an array for multi-core…
  - Would you rather distribute the work:

1. Round Robin?

2. Chunk-wise?