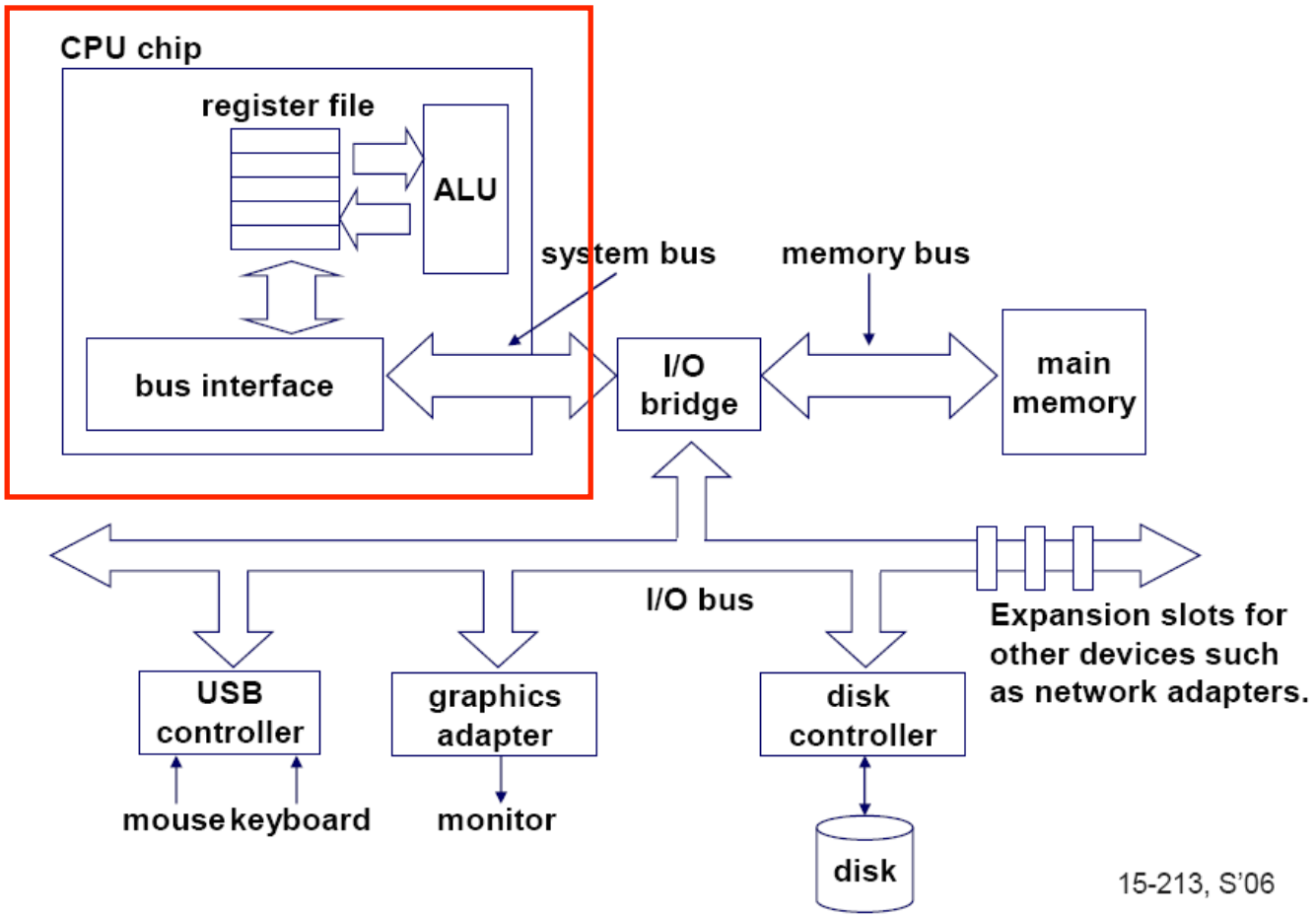
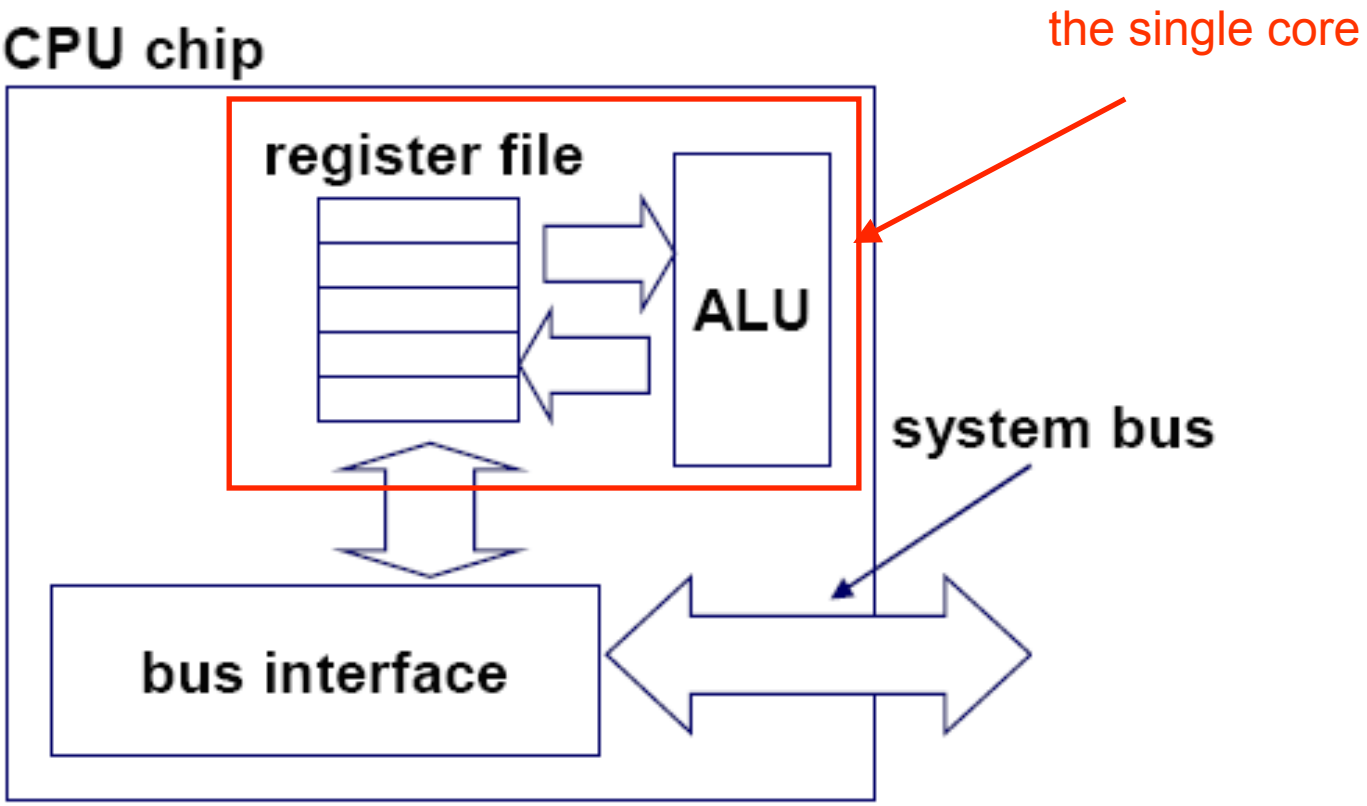
Multi-Core Architecture

Single-core computer

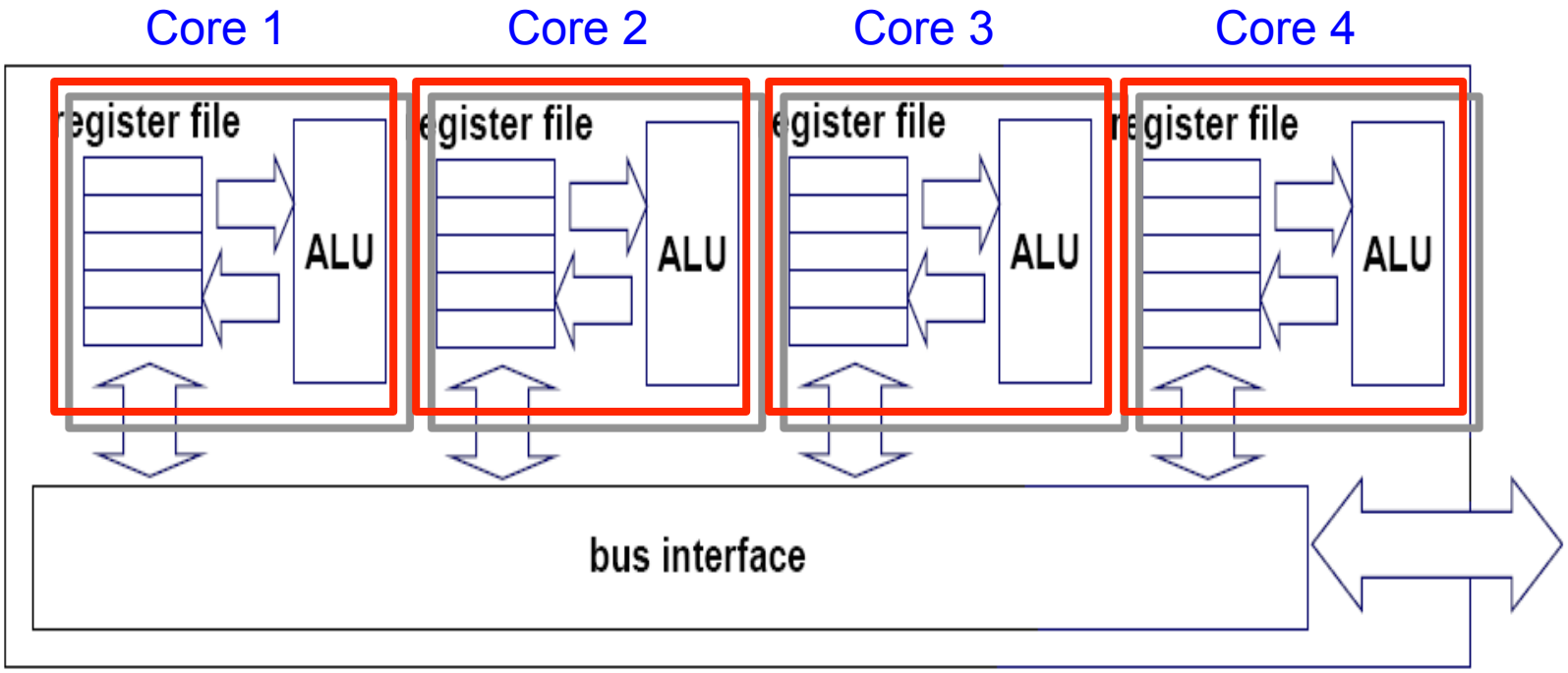


Single-core CPU chip



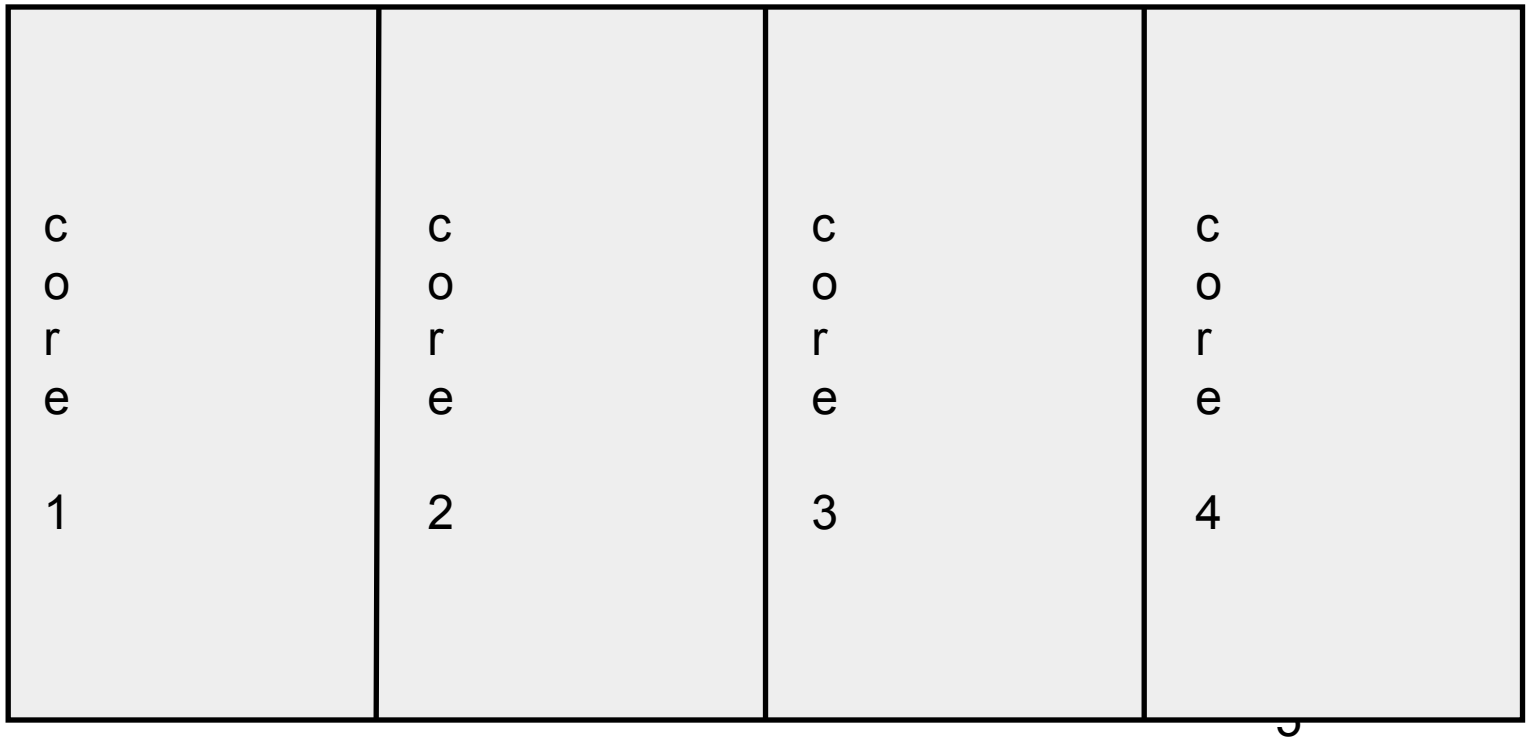
Multicore Architecture

- Replicate multiple processor cores on a single die

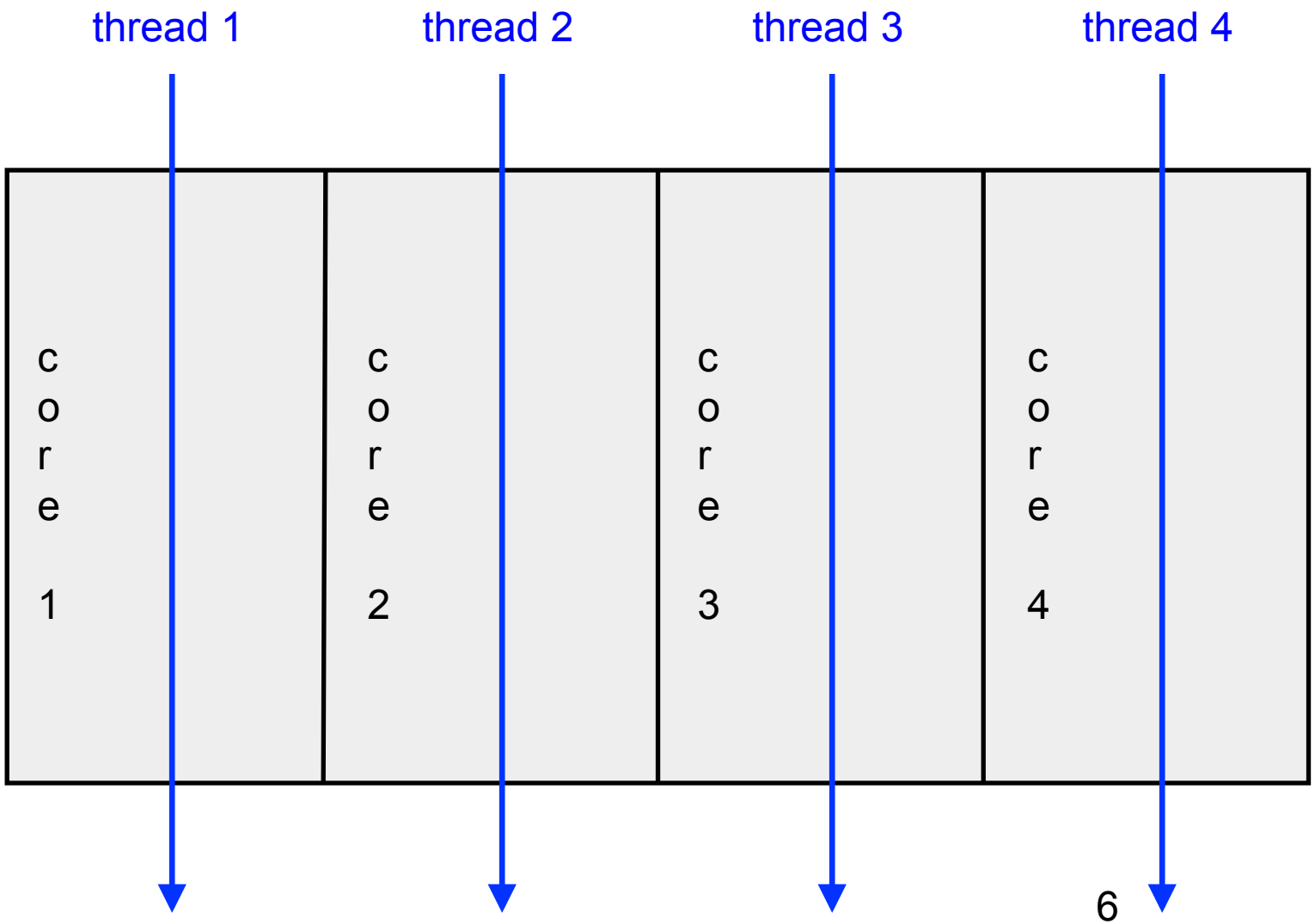


Multi-core CPU chip

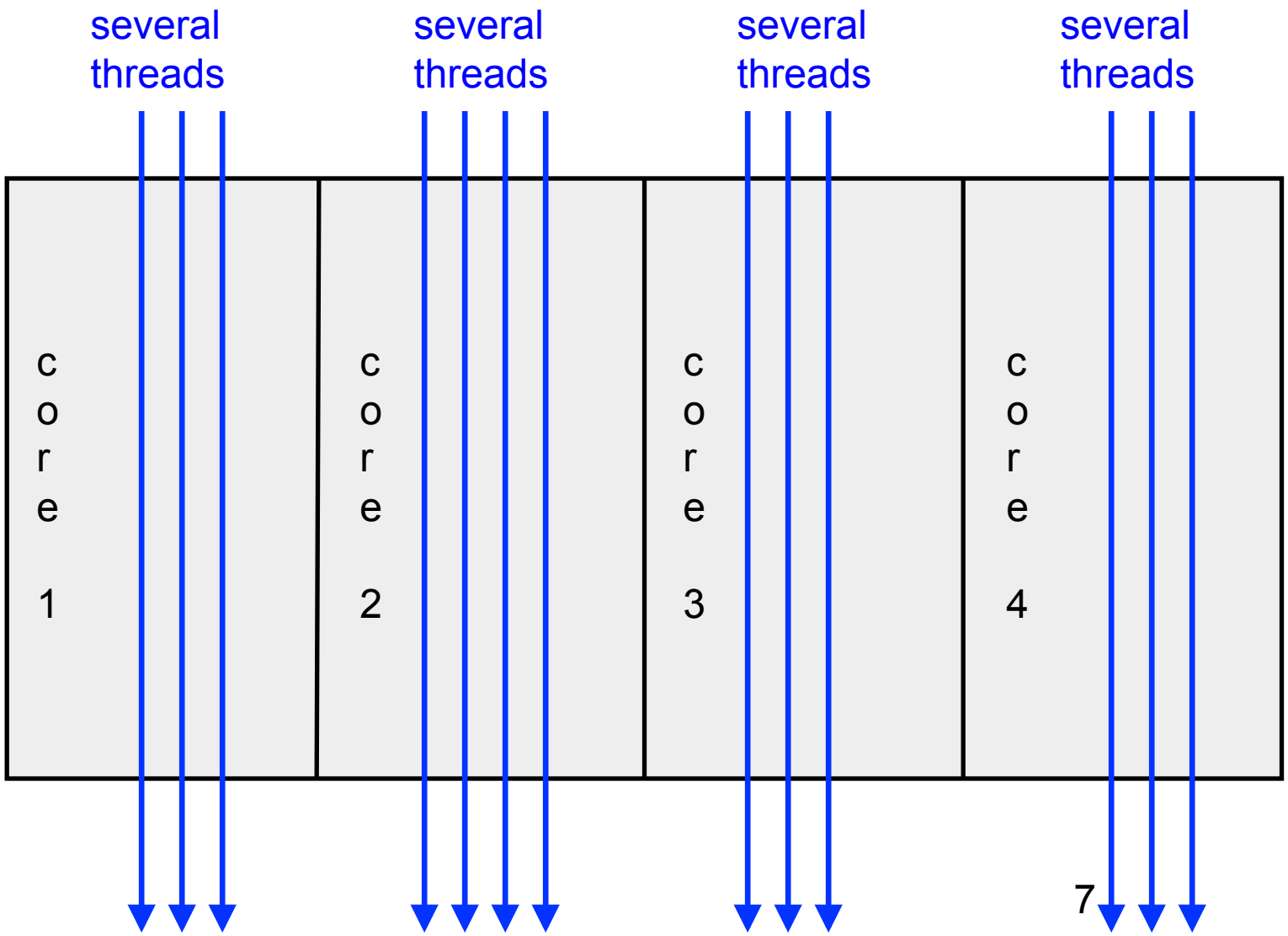
- The cores fit on a single processor socket
- Also called CMP (Chip Multi-Processor)



The cores run in parallel



Within each core, threads are time-sliced (just like on a uniprocessor)



Interaction with the Operating System

- OS perceives each core as a separate processor
- OS scheduler maps threads/processes to different cores
- Most major OS support multi-core today: Windows, Linux, Mac OS X, …

Why multi-core ?

- Difficult to make single-core clock frequencies even higher
- Deeply pipelined circuits:
  - heat problems
  - speed of light problems
  - difficult design and verification
  - large design teams necessary
  - server farms need expensive air-conditioning

- Many new applications are multithreaded
- General trend in computer architecture (shift towards more parallelism)

Instruction-level parallelism

- Parallelism at the machine-instruction level
- The processor can re-order, pipeline instructions, split them into microinstructions, do aggressive branch prediction, etc.
- Instruction-level parallelism enabled rapid increases in processor speeds over the last 15 years

Thread-level parallelism (TLP)

- This is parallelism on a more coarser scale
- Server can serve each client in a separate thread (Web server, database server)
- A computer game can do AI, graphics, and physics in three separate threads
- Single-core superscalar processors cannot fully exploit TLP
- Multi-core architectures are the next step in processor evolution: explicitly exploiting TLP

General context: Multiprocessors

- Multiprocessor is any computer with several processors

Lemieux cluster, Pittsburgh supercomputing center

- SIMD
  - Single instruction, multiple data
  - Modern graphics cards
- MIMD
  - Multiple instructions, multiple data

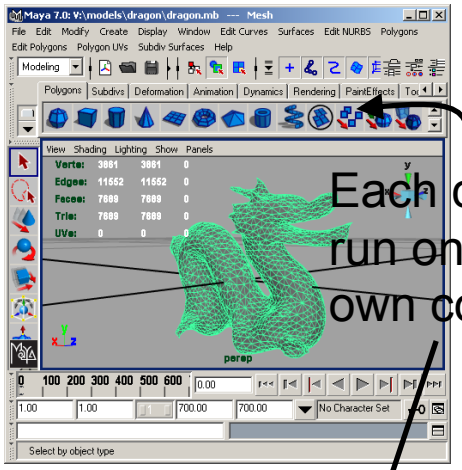
Multiprocessor memory types

- Shared memory:
  In this model, there is one (large) common shared memory for all processors

- Distributed memory:
  In this model, each processor has its own (small) local memory, and its content is not replicated anywhere else

What applications benefit from multi-core?

- Database servers
- Web servers (Web commerce)
- Compilers
- Multimedia applications
- Scientific applications, CAD/ CAM
- In general, applications with *Thread-level parallelism* (as opposed to instruction-level parallelism)

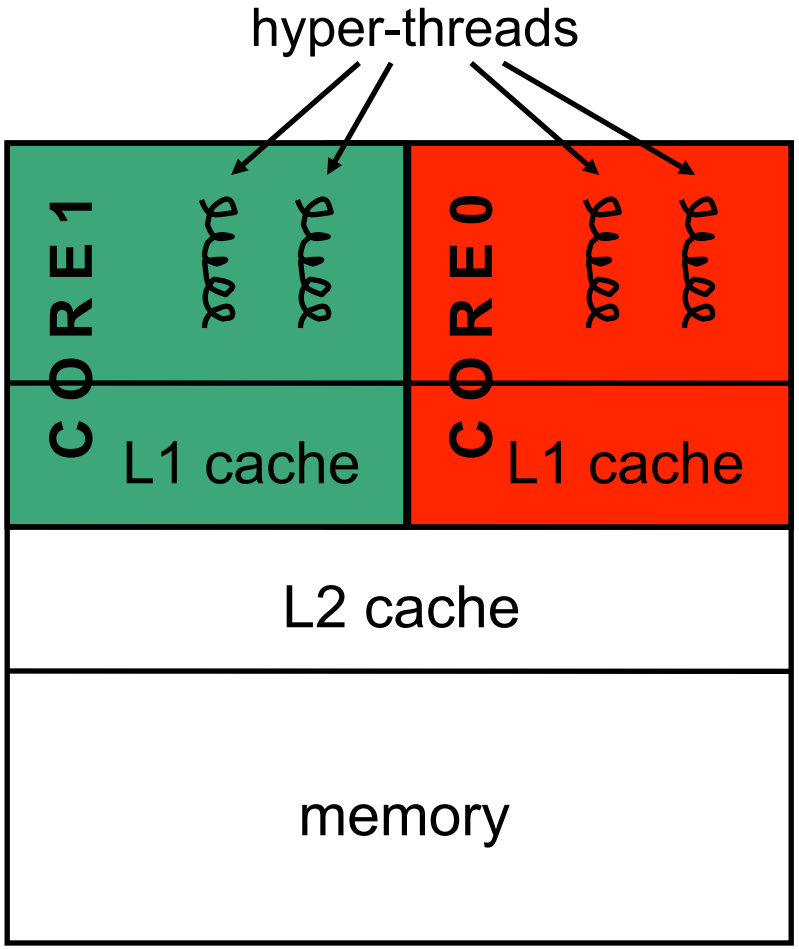


The memory hierarchy

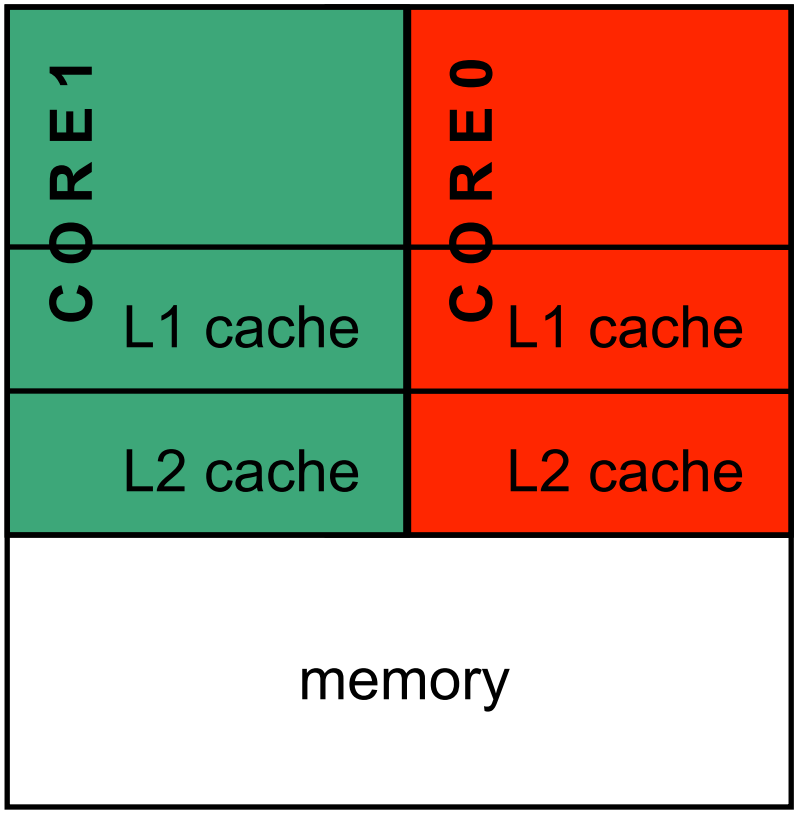
- Multi-core chips:
  - L1 caches private
  - L2 caches private in some architectures and shared in others
- Memory is always shared

“Fish” machines

- Dual-core
  Intel Xeon processors
- Each core is
  hyper-threaded
- Private L1 caches
- Shared L2 caches

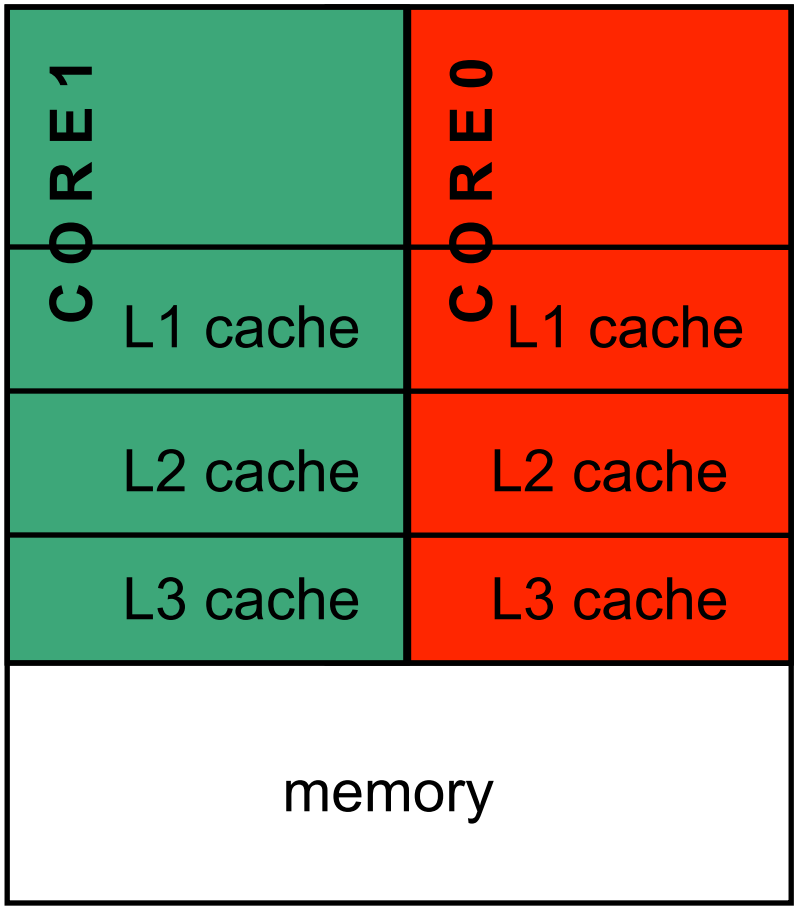


Designs with private L2 caches



Both L1 and L2 are private

Examples: AMD Opteron,
AMD Athlon, Intel Pentium D



A design with L3 caches

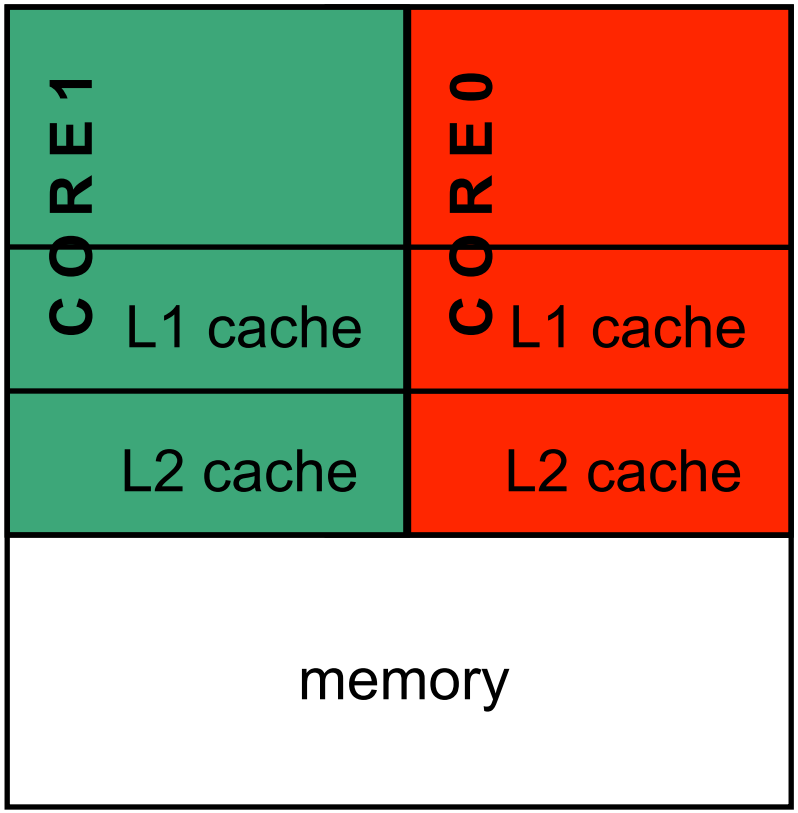
Example: Intel Itanium 2

Private vs shared caches?

- Advantages/disadvantages?

Private vs shared caches

- Advantages of private:
  - They are closer to core, so faster access
  - Reduces contention
- Advantages of shared:
  - Threads on different cores can share the same cache data
  - More cache space available if a single (or a few) high-performance thread runs on the system

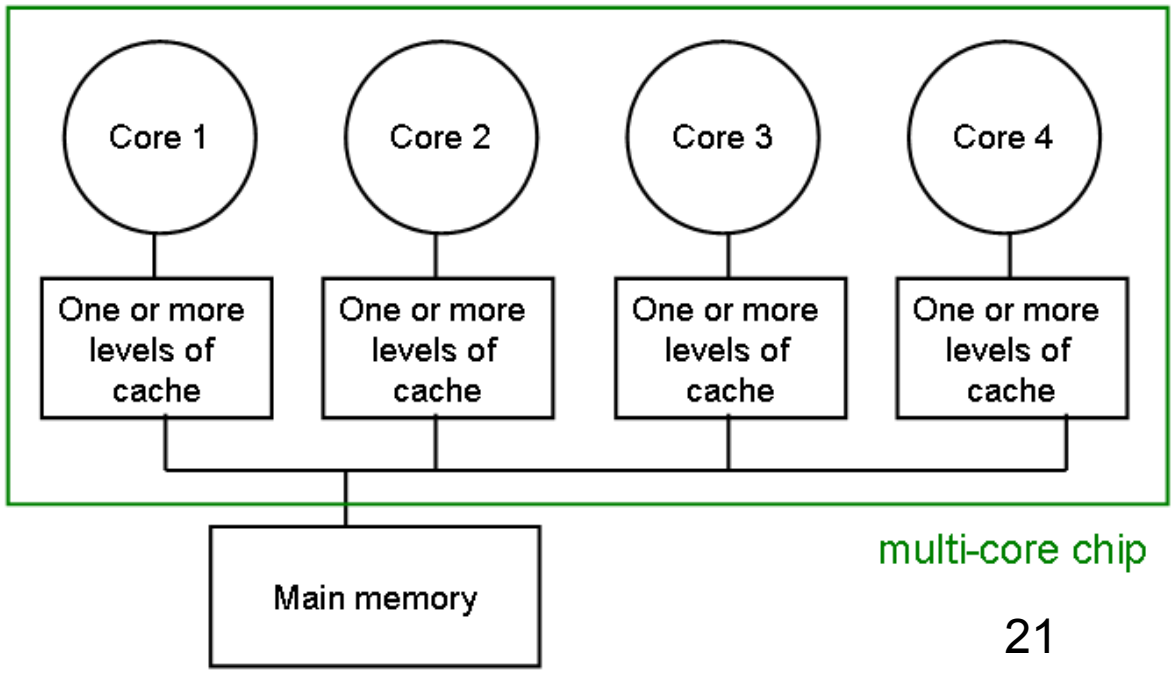


Question:

In multicore environment with private L1/L2 cache, is there any serious issue we should address?

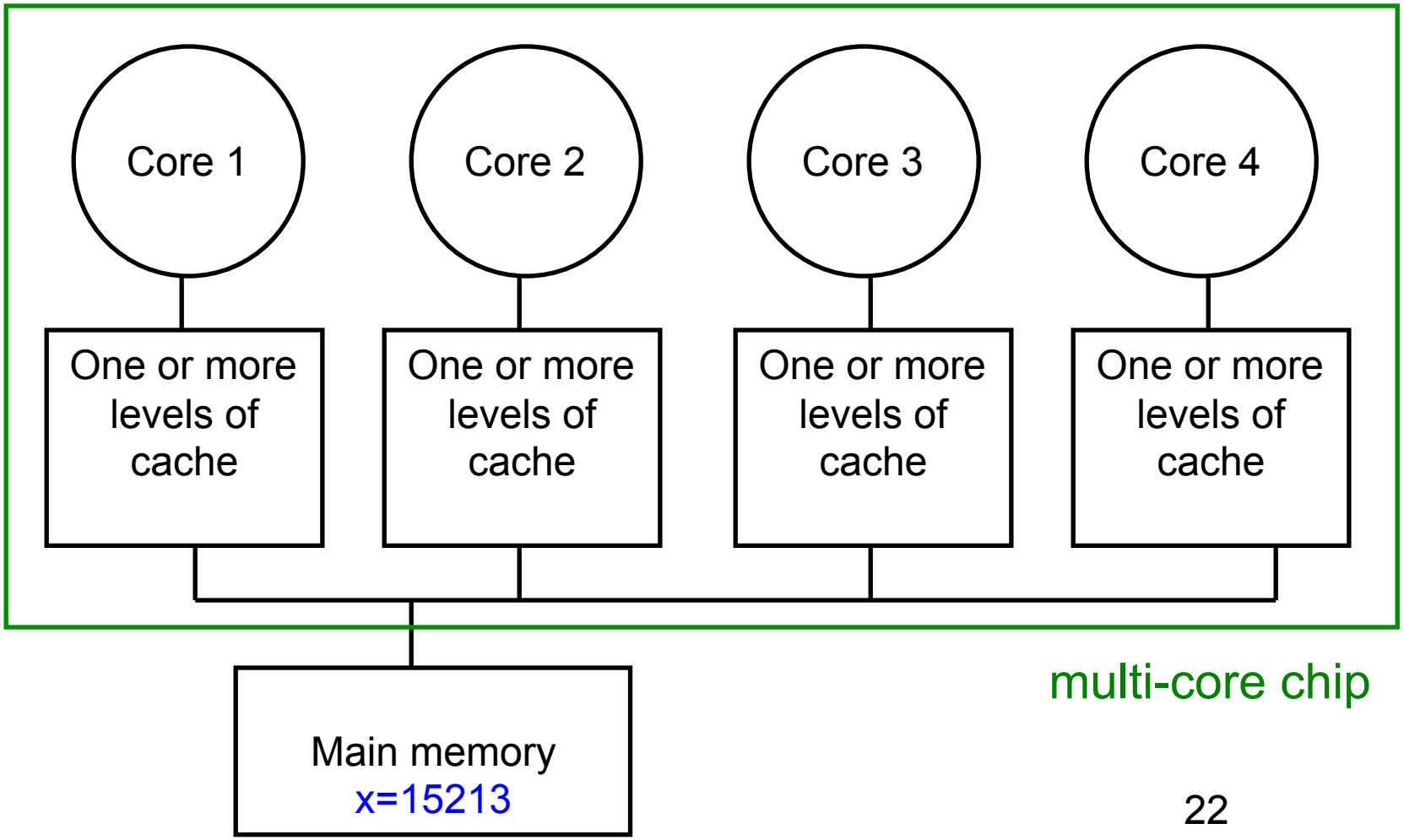
The cache coherence problem

- Since we have multiple private caches:
  How to keep the data consistent across caches?
- Each core should perceive the memory as a monolithic array, shared by all the cores



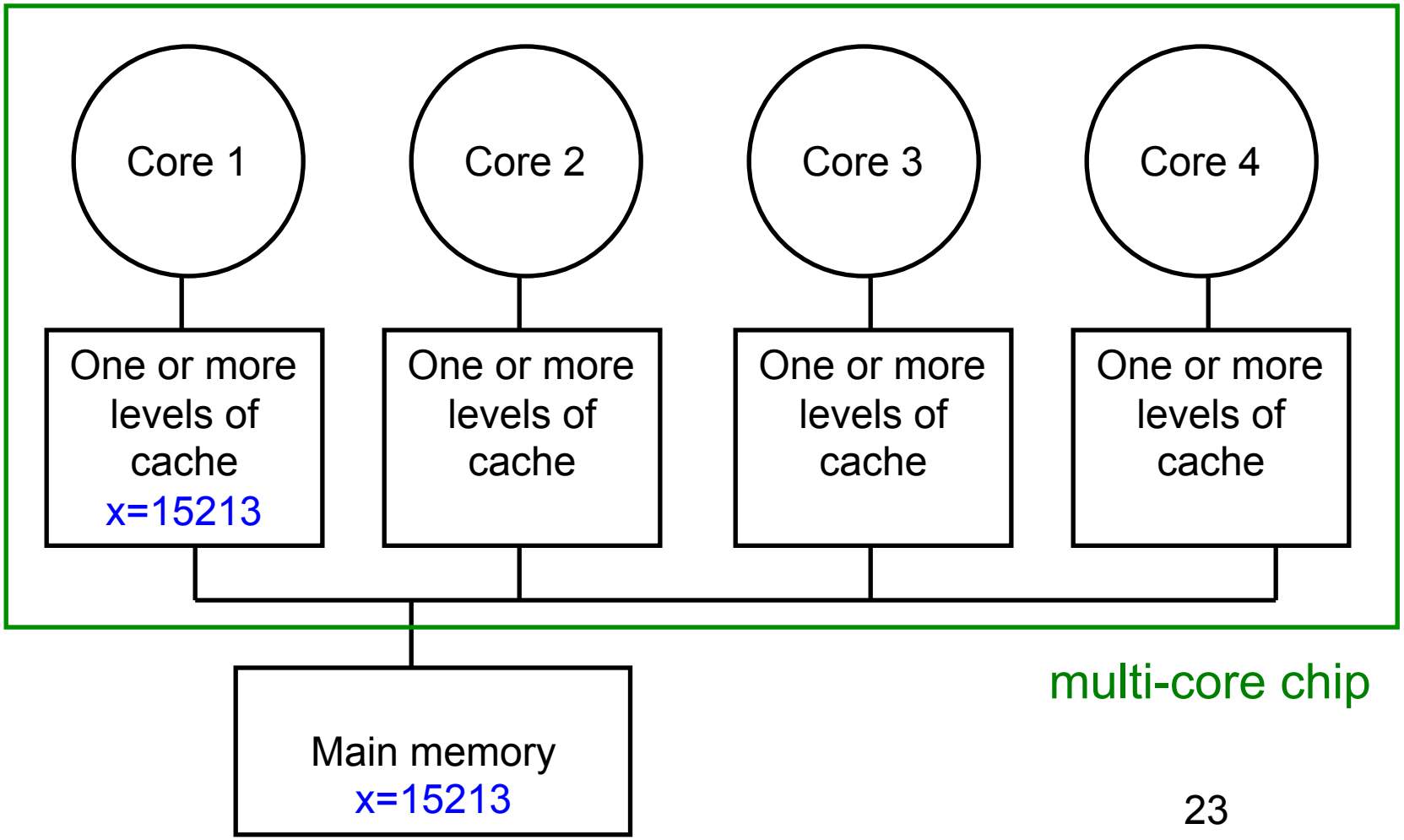
The cache coherence problem

Suppose variable x initially contains 15213



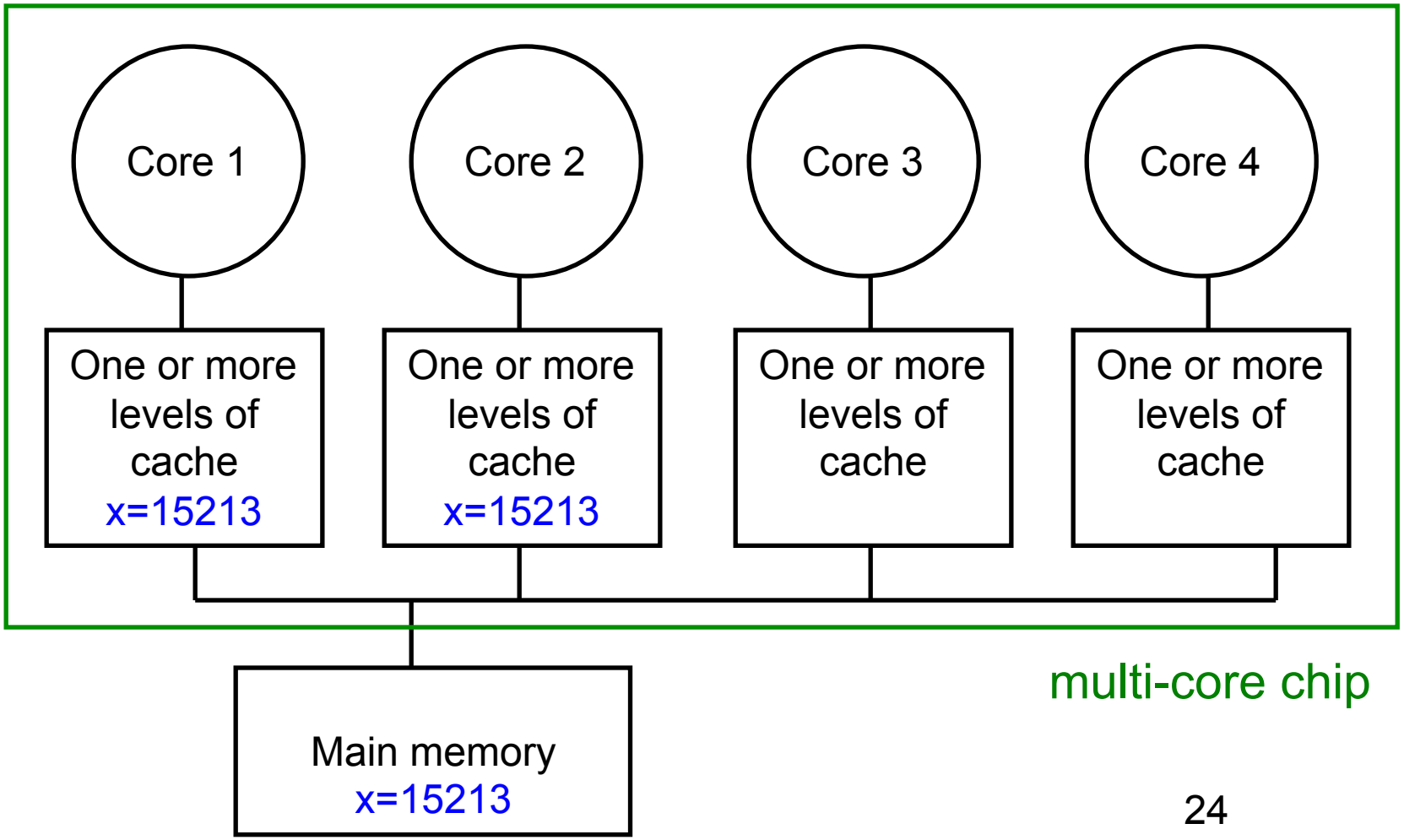
The cache coherence problem

Core 1 reads x



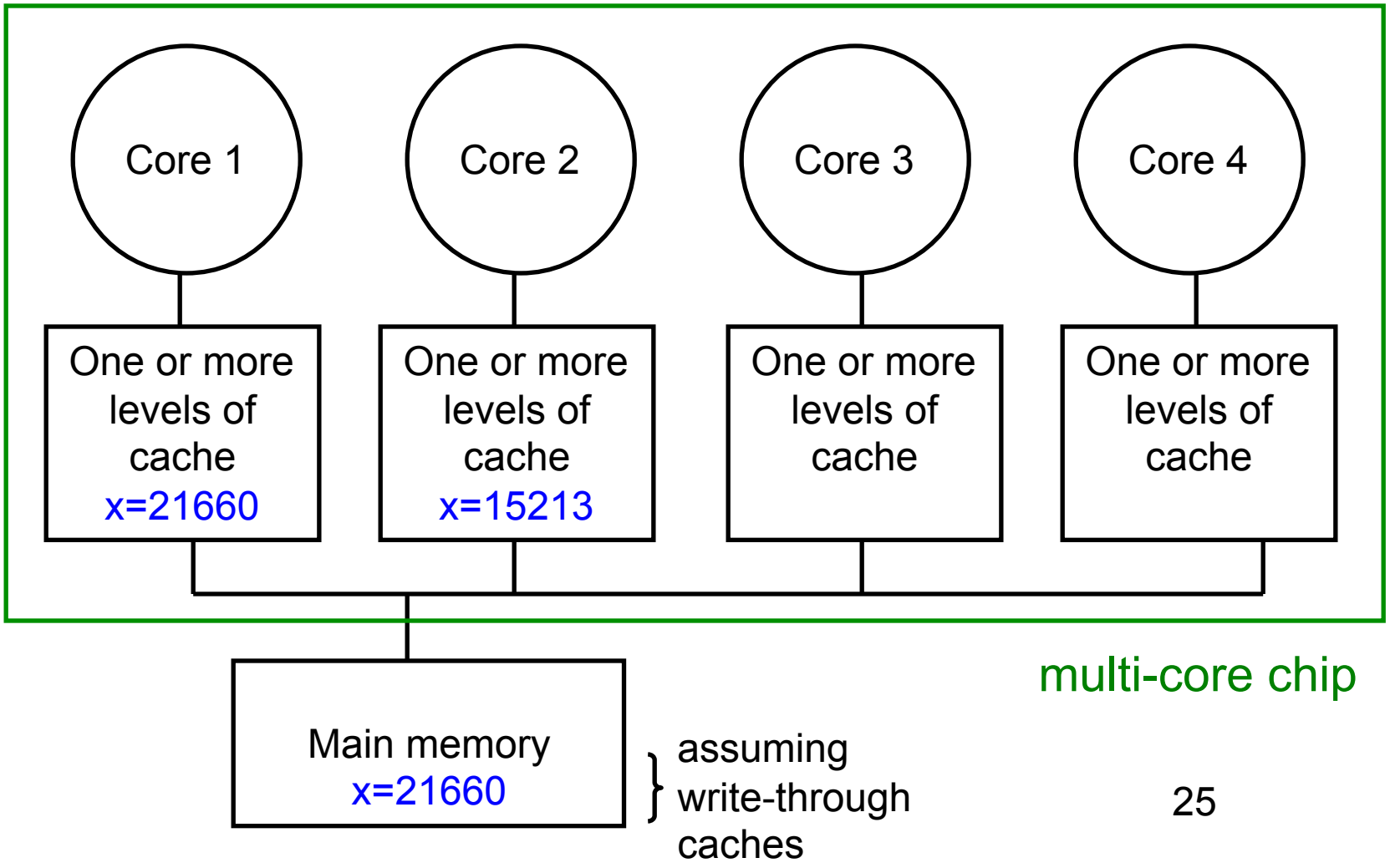
The cache coherence problem

Core 2 reads x



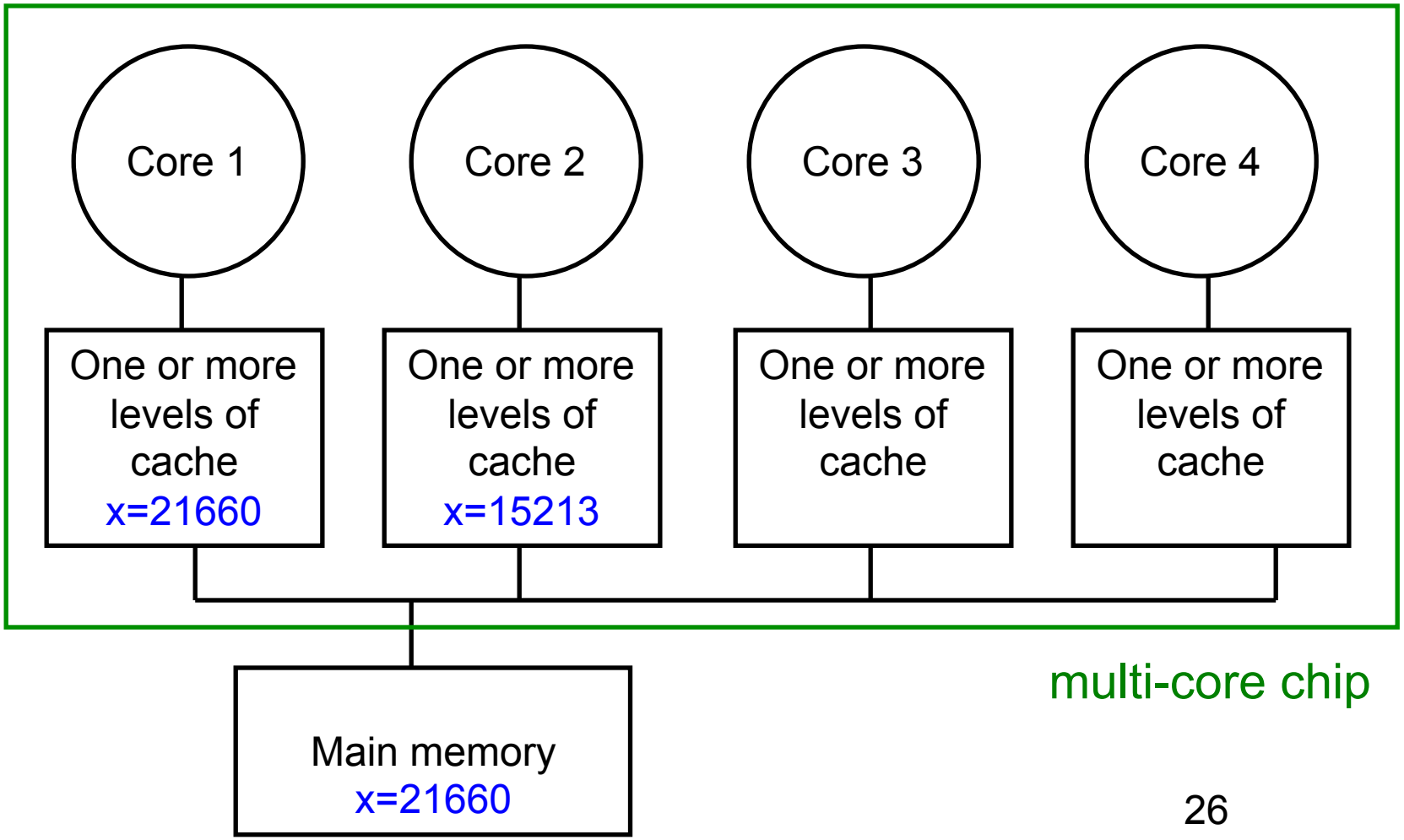
The cache coherence problem

Core 1 writes to x, setting it to 21660



The cache coherence problem

Core 2 attempts to read x… gets a stale copy

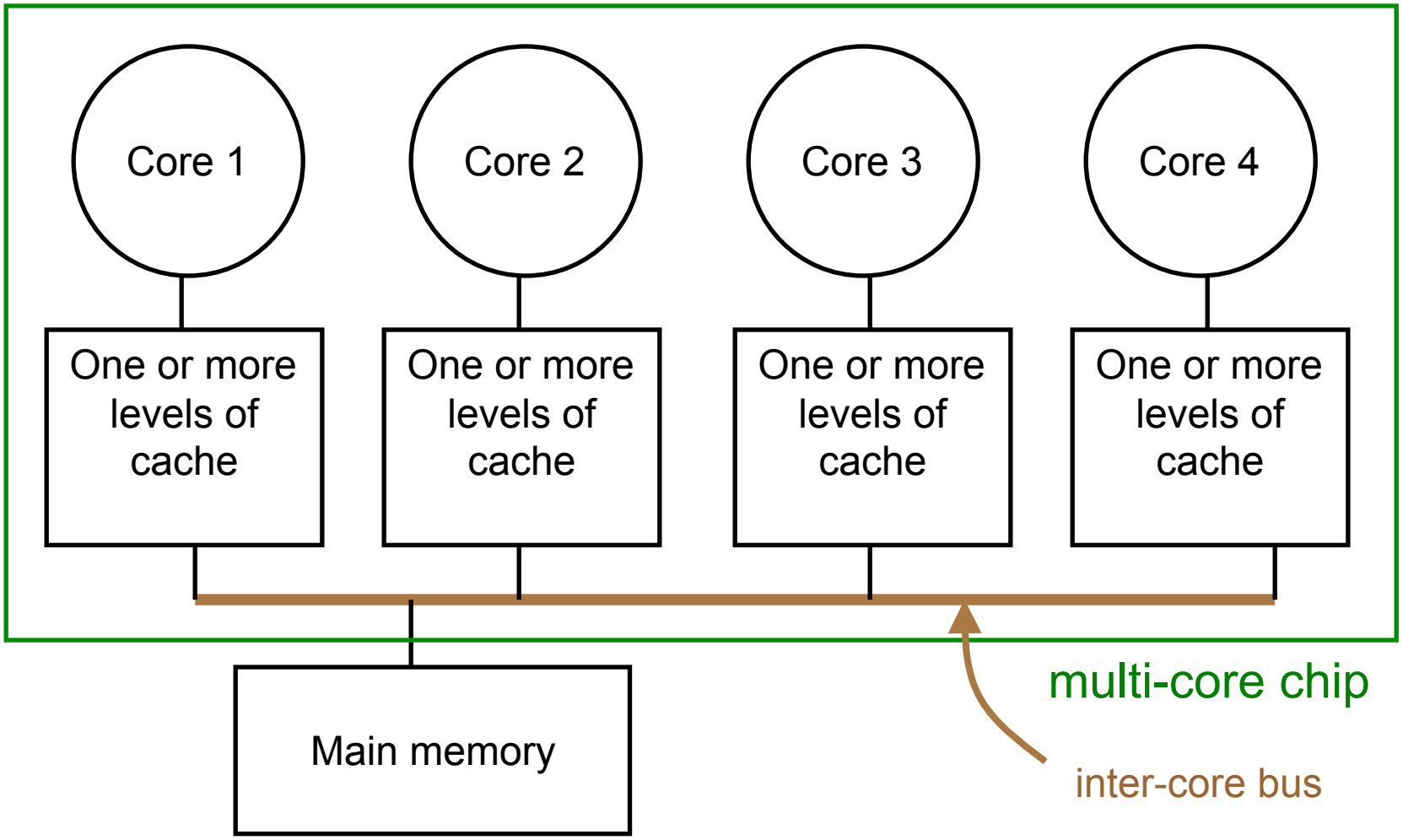


Solutions for cache coherence

- This is a general problem with multiprocessors, not limited just to multi-core
- There exist many solution algorithms, coherence protocols, etc.

- A simple solution: *invalidation*-based protocol with *snooping*

Inter-core bus

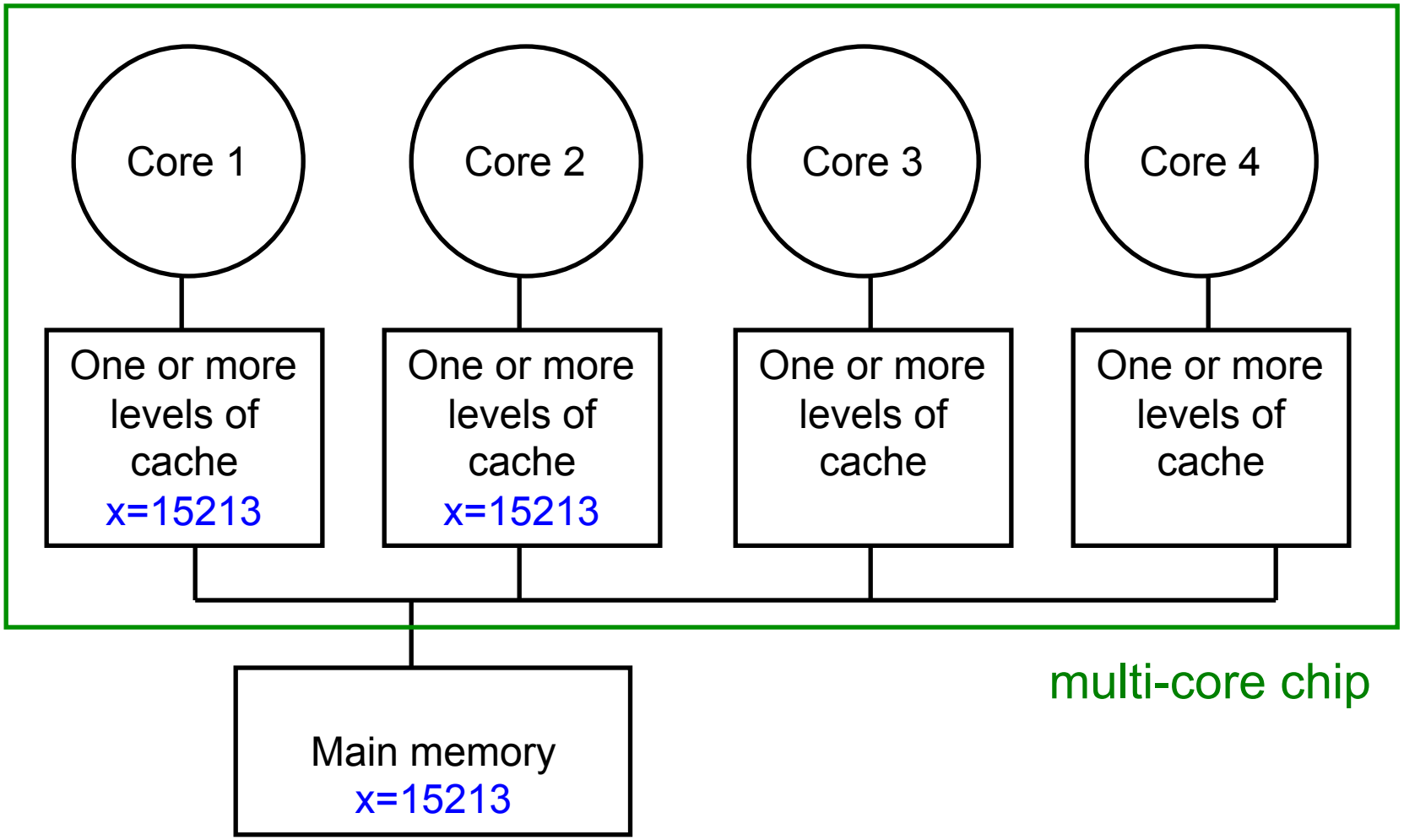


Invalidation protocol with snooping

- Invalidation:
  If a core writes to a data item, all other copies of this data item in other caches are *invalidated*
- Snooping:
  All cores continuously “snoop” (monitor) the bus connecting the cores.

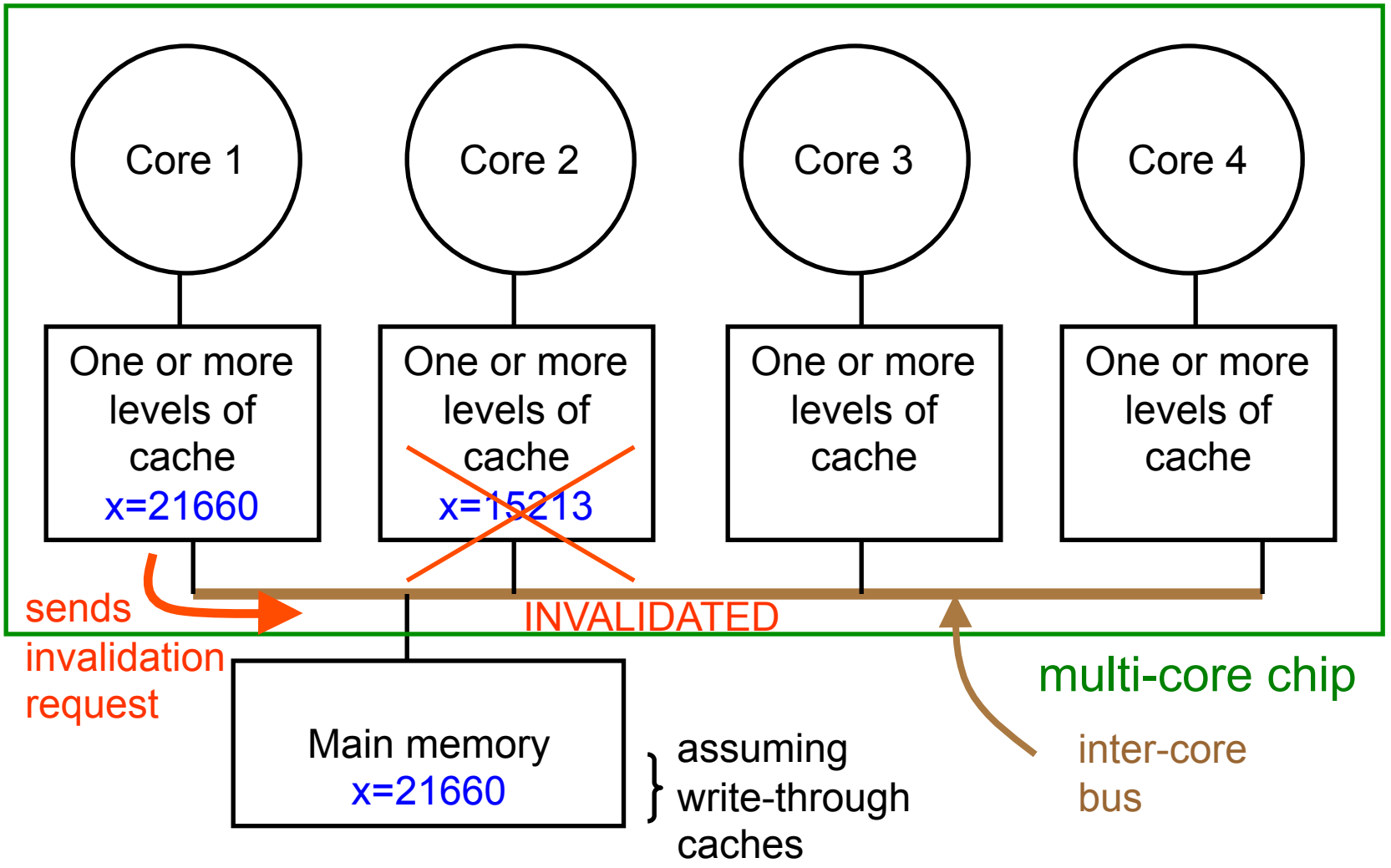
The cache coherence problem

Revisited: Cores 1 and 2 have both read x



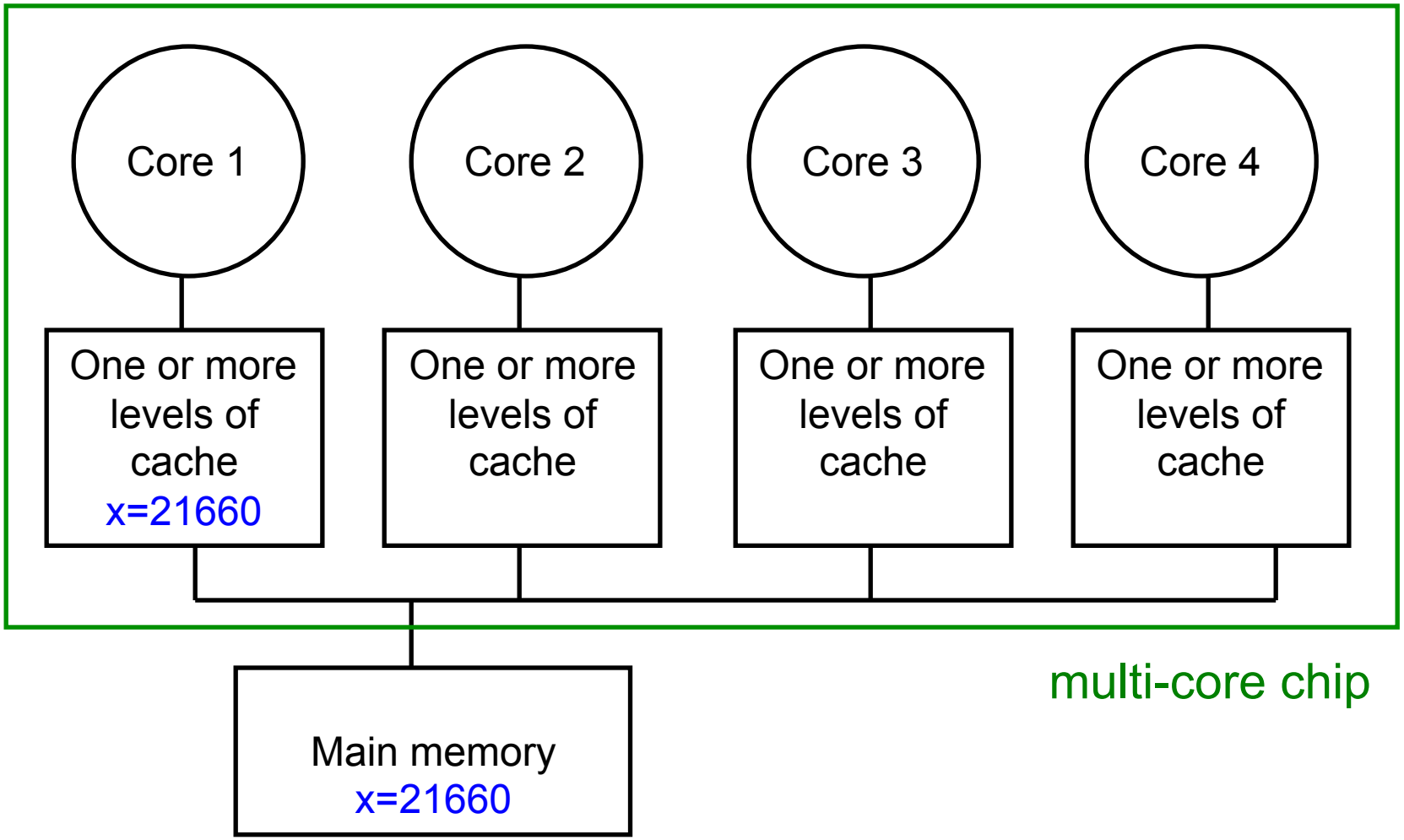
The cache coherence problem

Core 1 writes to x, setting it to 21660



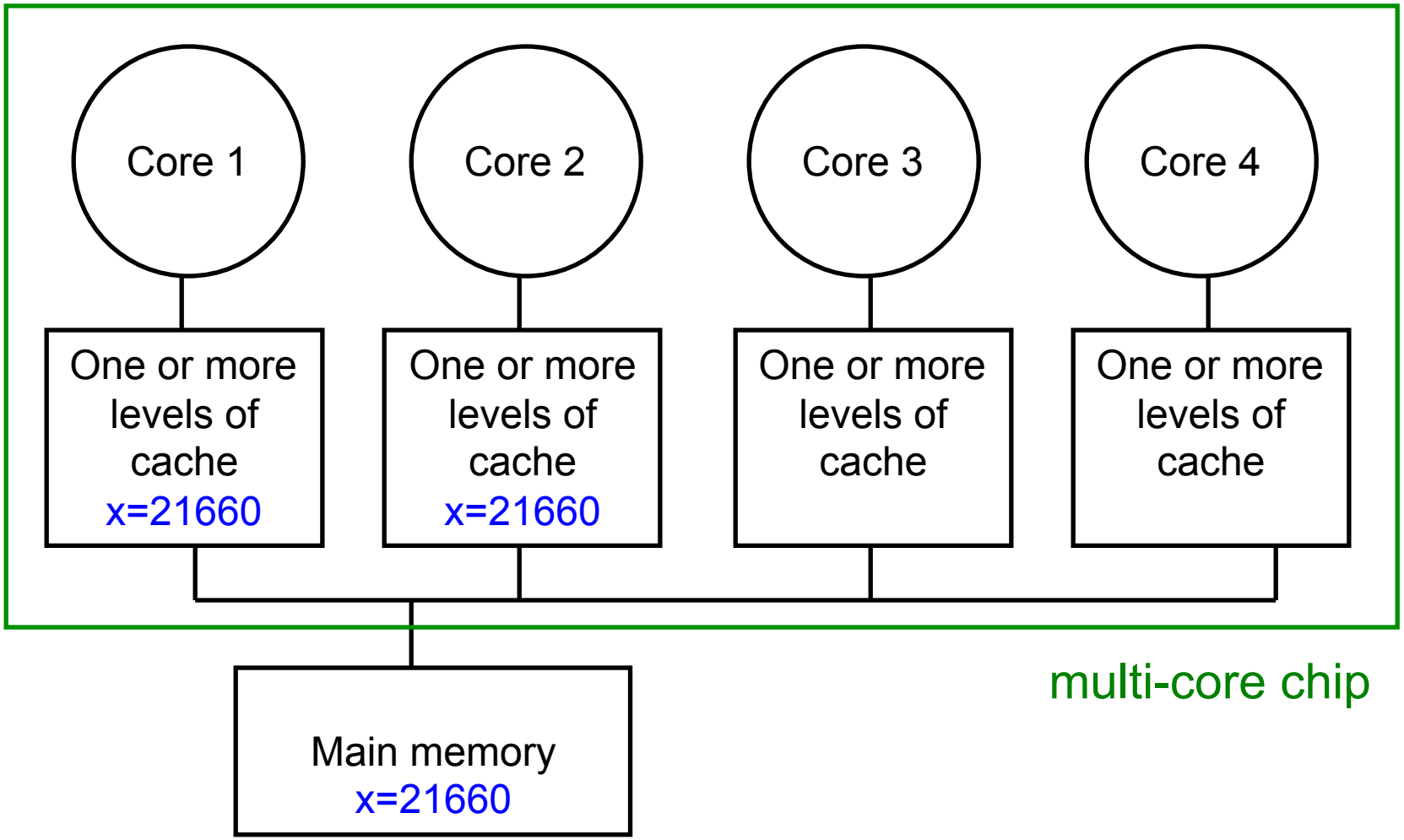
The cache coherence problem

After invalidation:



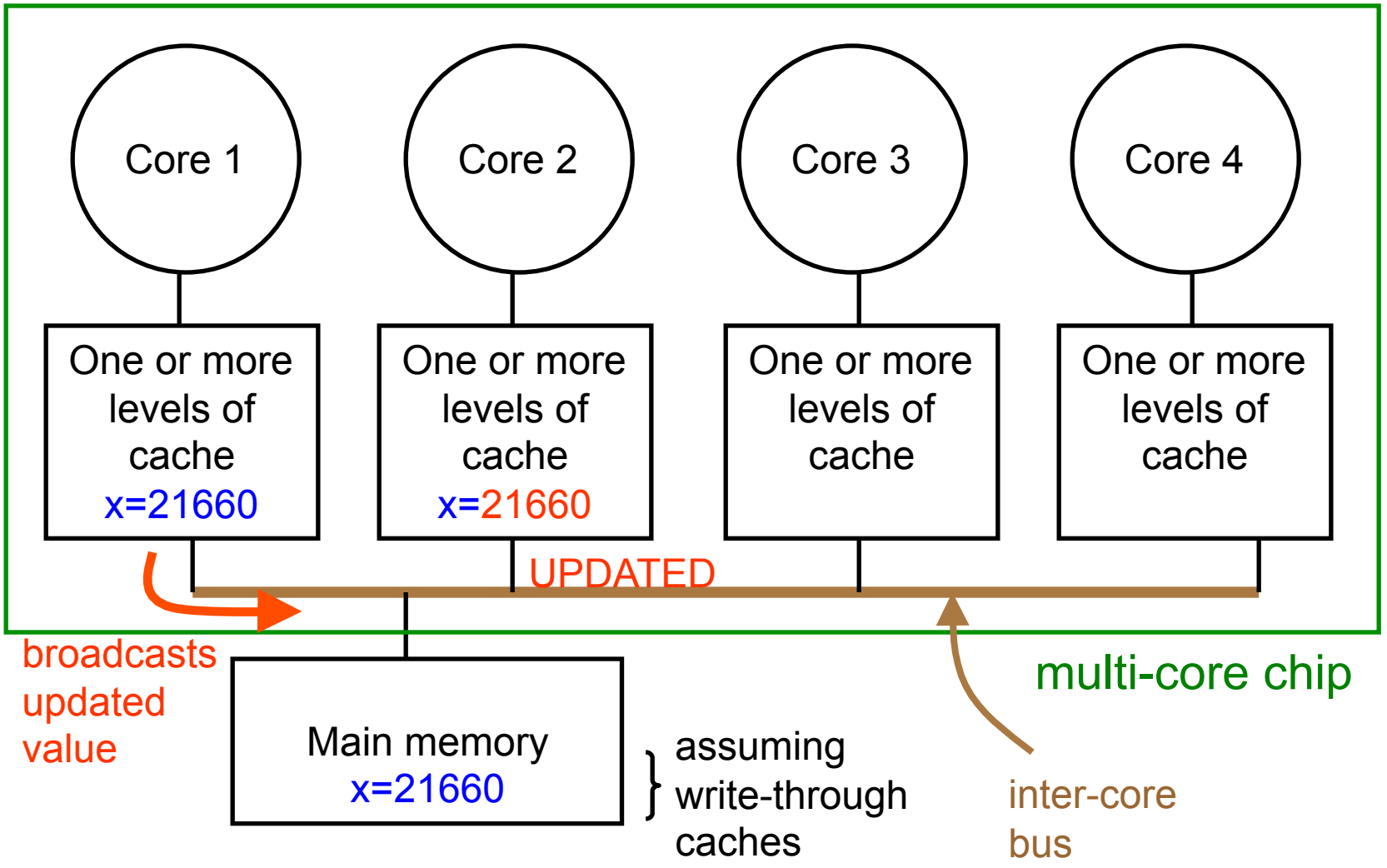
The cache coherence problem

Core 2 reads x. Cache misses, and loads the new copy.



Alternative to invalidate protocol: update protocol

Core 1 writes x=21660:



Question: Which do you think is better? Invalidation or update?

Invalidation vs update

- Multiple writes to the same location
  - invalidation: only the first time
  - update: must broadcast each write
    (which includes new variable value)

- Invalidation generally performs better:
  it generates less bus traffic

Invalidation protocols

- This was just the basic invalidation protocol
- More sophisticated protocols use extra cache state bits
- MSI, MESI (Modified, Exclusive, Shared, Invalid)

Programming for multi-core

- Programmers must use threads or processes
- Spread the workload across multiple cores
- Write parallel algorithms
- OS will map threads/processes to cores

Thread safety very important

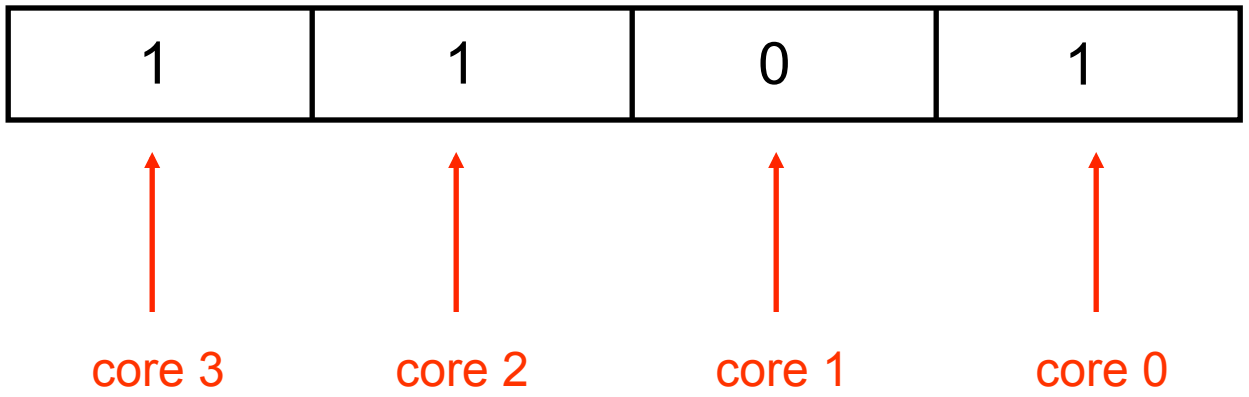
- Pre-emptive context switching:
  context switch can happen AT ANY TIME
- True concurrency, not just uniprocessor time-slicing
- Concurrency bugs exposed much faster with multi-core

Assigning threads to the cores

- Each thread/process has an *affinity mask*
- Affinity mask specifies what cores the thread is allowed to run on
- Different threads can have different masks
- Affinities are inherited across fork()

Affinity masks are bit vectors

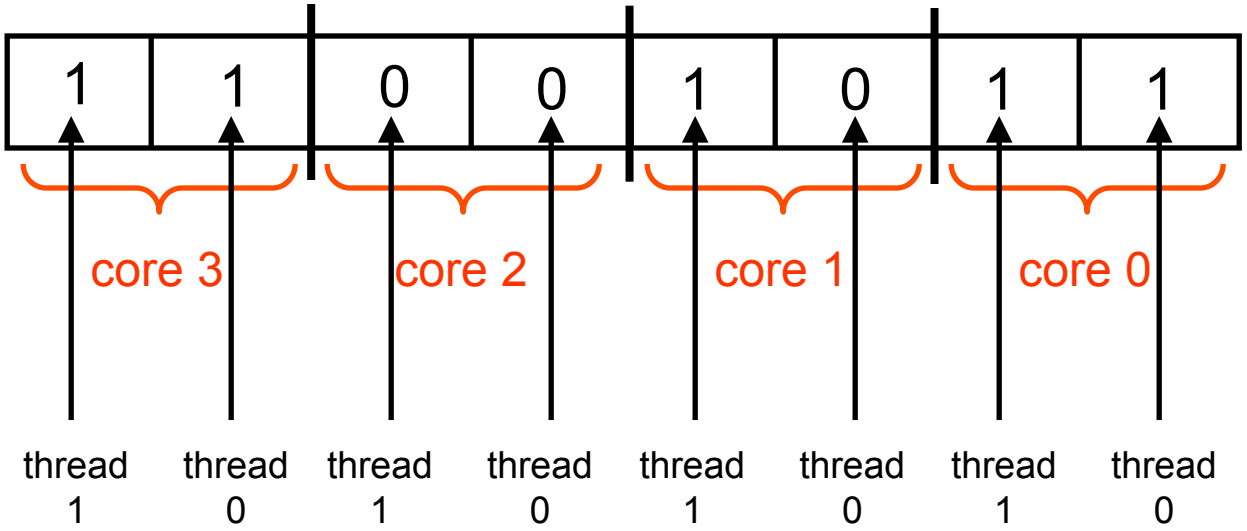
- Example: 4-way multi-core



- Process/thread is allowed to run on cores 0,2,3, but not on core 1

Affinity masks when multi-core and SMT combined

- Separate bits for each simultaneous thread
- Example: 4-way multi-core, 2 threads per core



  - Core 2 can't run the process
  - Core 1 can only use one simultaneous thread

Default Affinities

- Default affinity mask is all 1s:
  all threads can run on all processors

- Then, the OS scheduler decides what threads run on what core

- OS scheduler detects skewed workloads, migrating threads to less busy processors

Process migration is costly

- Need to restart the execution pipeline
- Cached data is invalidated
- OS scheduler tries to avoid migration as much as possible:
  it tends to keeps a thread on the same core
- This is called *soft affinity*

Hard affinities

- The programmer can prescribe her own affinities (hard affinities)
- Rule of thumb: use the default scheduler unless a good reason not to