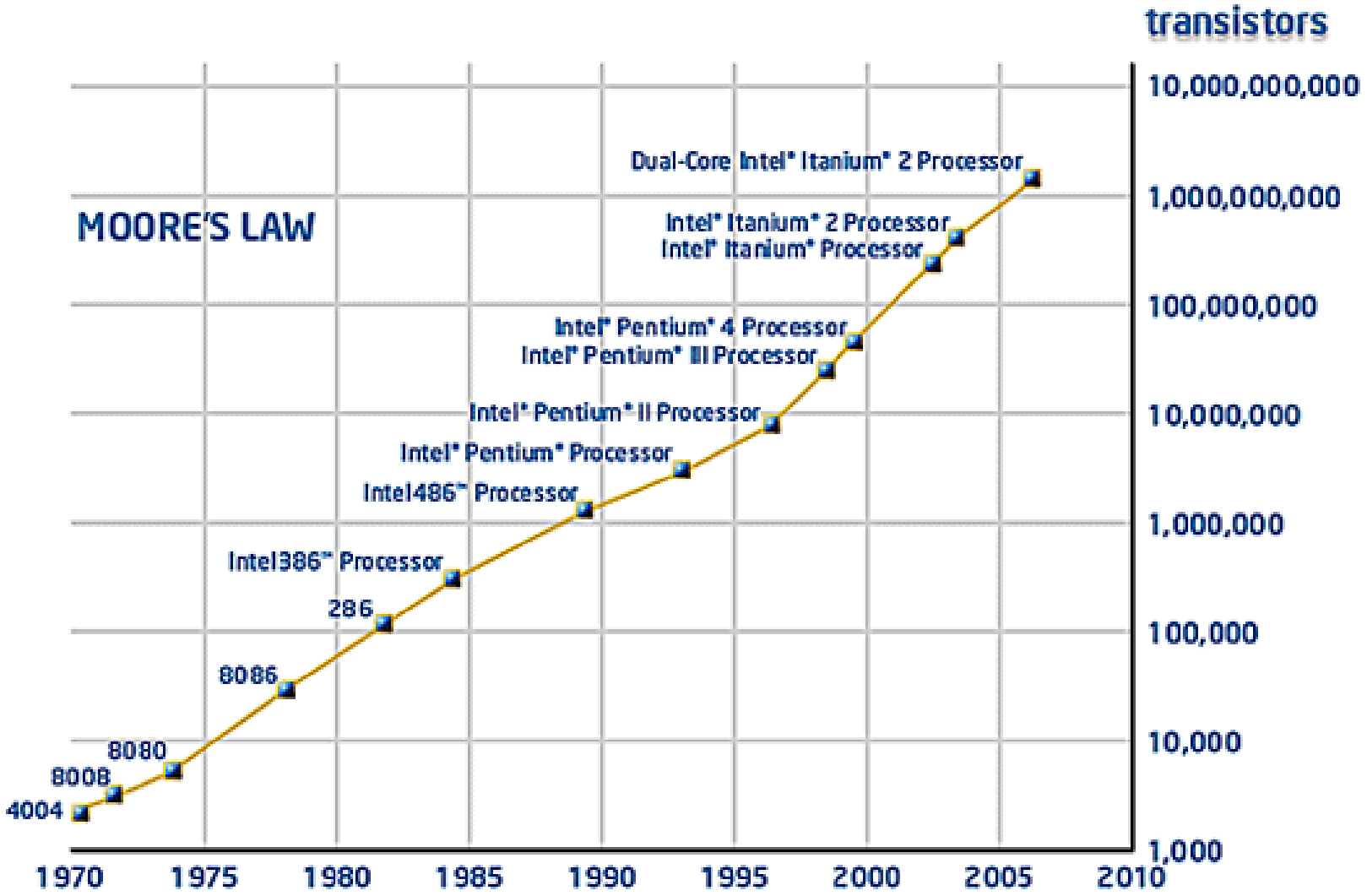
L11 Multi-core

Zonghua Gu, 2018

Acknowledgement: some slides taken from UC Berkeley CS61C

Moore's Law

- Number of transistors on a chip doubles every 18 to 24 Months

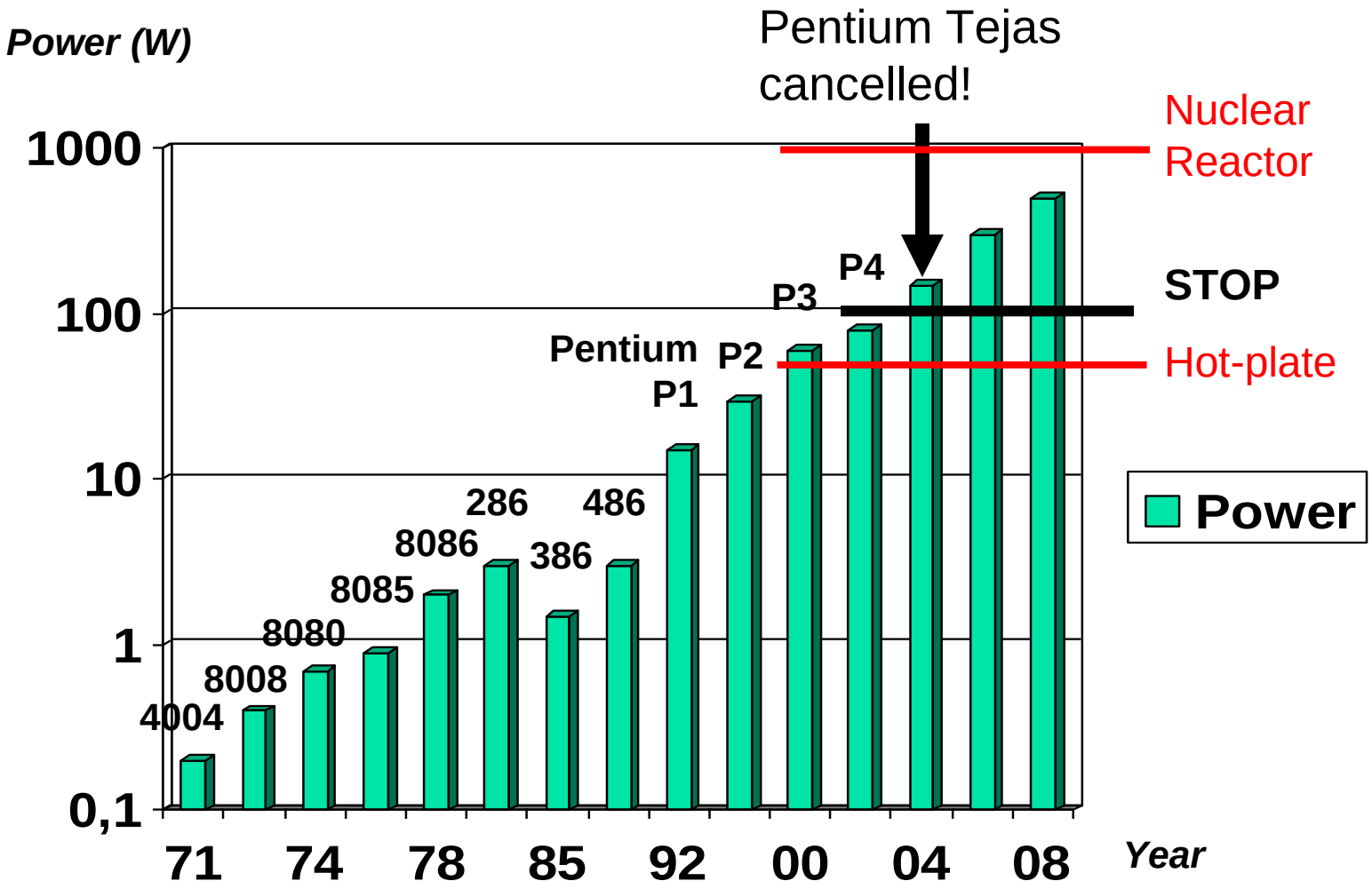


What to do with all the transistors?

- Up to ~2005, CPU clock frequency had been increasing along with Moore's law
  - Increased # transistors are used to design more sophisticated CPU (deeper pipelines, more complex control unit, super-scalar…)
- But CPU clock freq stopped increasing after 2005
  - Highest CPU freq is ~4 GHz
  - Mainly due to power and heating issues

CPU clock freq stopped increasing

- Power is proportional to CPU clock freq cubed (f^3)

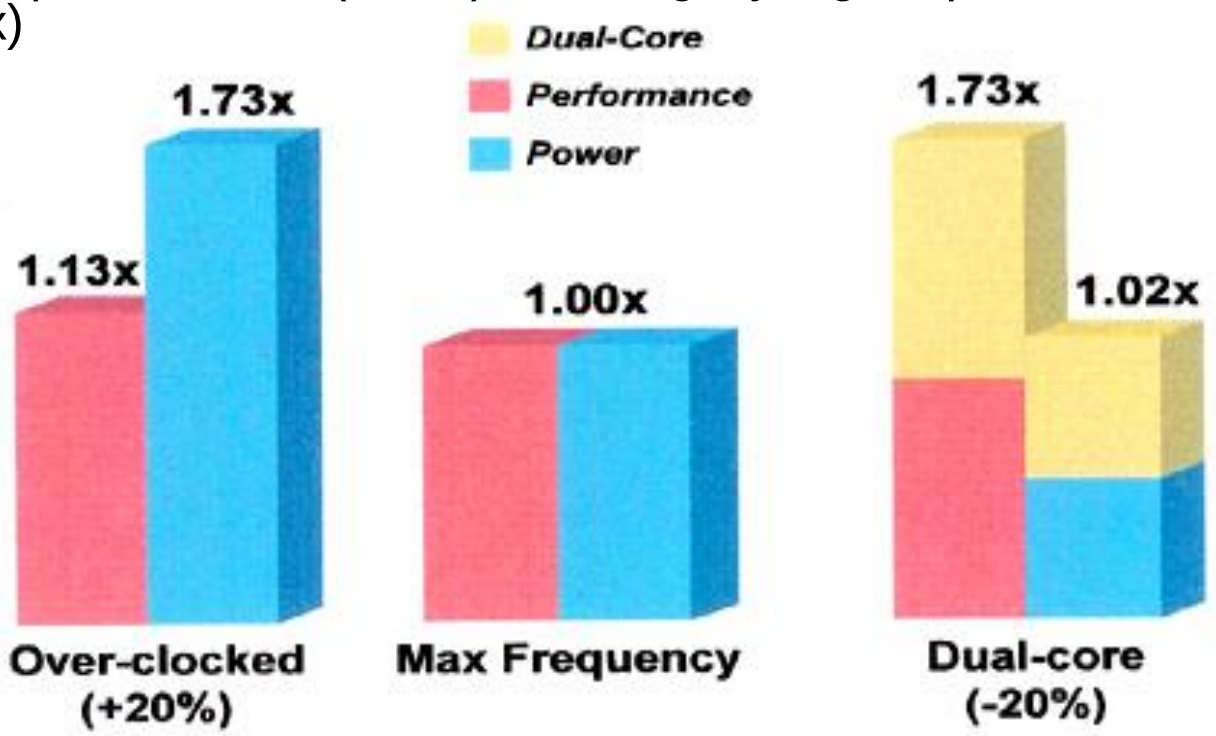


Motivation for multicore

- # transistors on a chip is still increasing with Moore's law; What to do with all the transistors now?
- Instead of making a single-core faster by increasing its CPU clock frequency, put multiple (slow) cores on the same chip to increase overall chip performance with parallelism
  - Multicore processors!

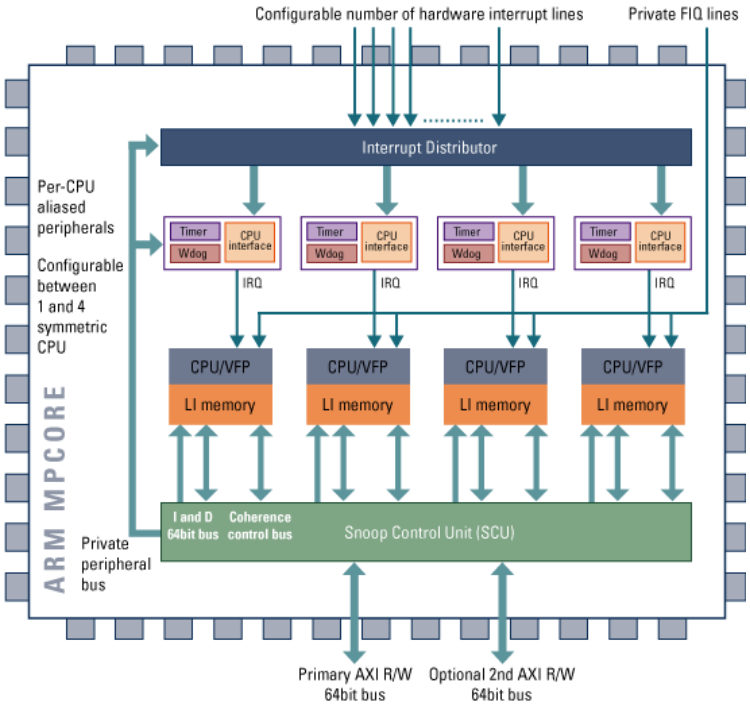
Performance vs. Power

- **Middle**: Uni-core processor with standard clock freq (= f)
- **Left**: Uni-core processor over-clocked 20% (clock freq = 1.2*f)
  - Higher performance (1.13x), but much higher power consumption (1.73x)
- **Right**: A dual-core processor, with each core under-clocked 20% (clock freq = 0.8*f)
  - Higher performance (1.73x), with slightly higher power consumption (1.02x)



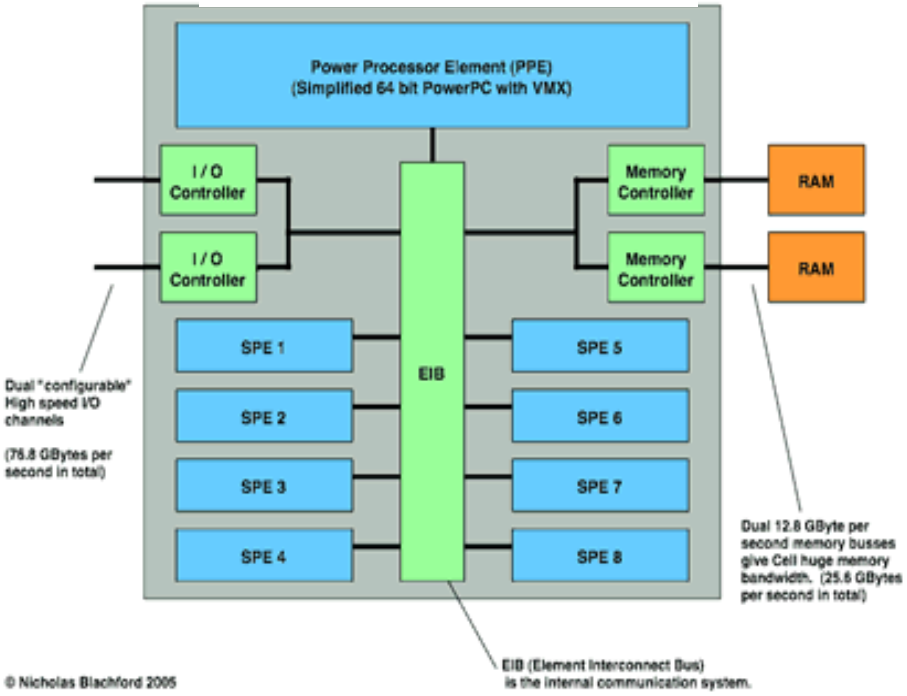
Homogeneous vs Heterogenous Multicores

ARM's MPCore



- 4 identical ARMv6 cores

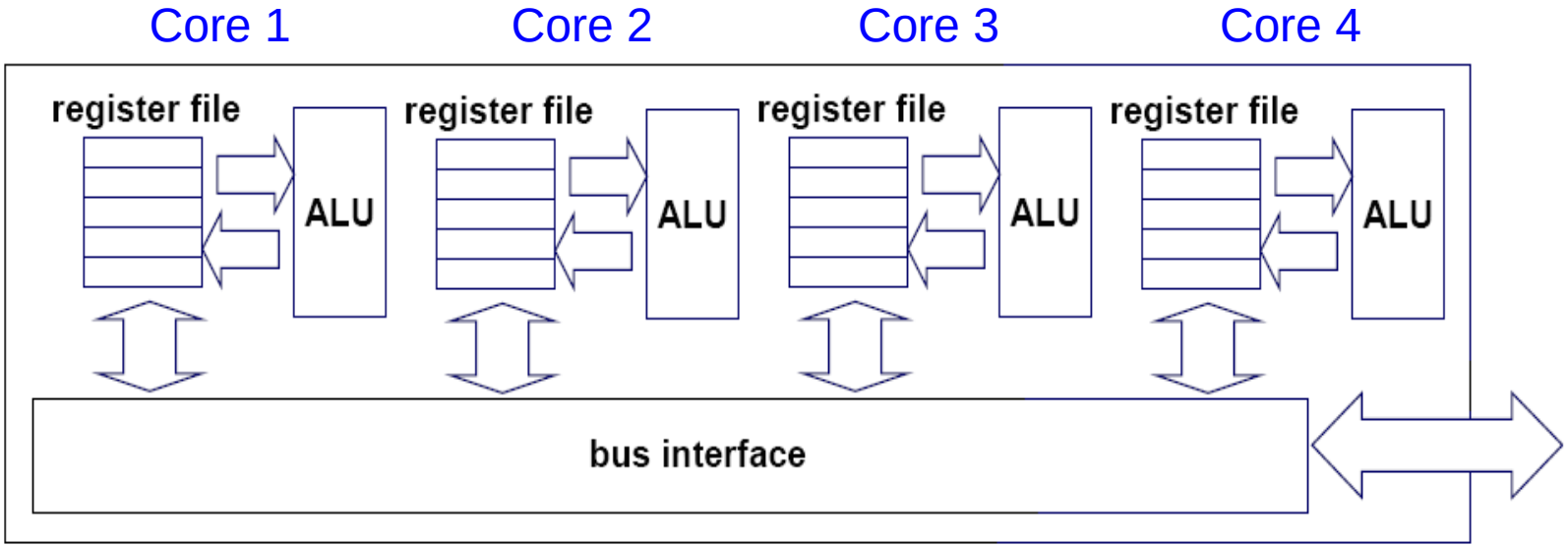
IBM Cell Processor



- One Power Processor Element (PPE)
- 8 Synergistic Processing Element (SPE)

Multi-core architectures

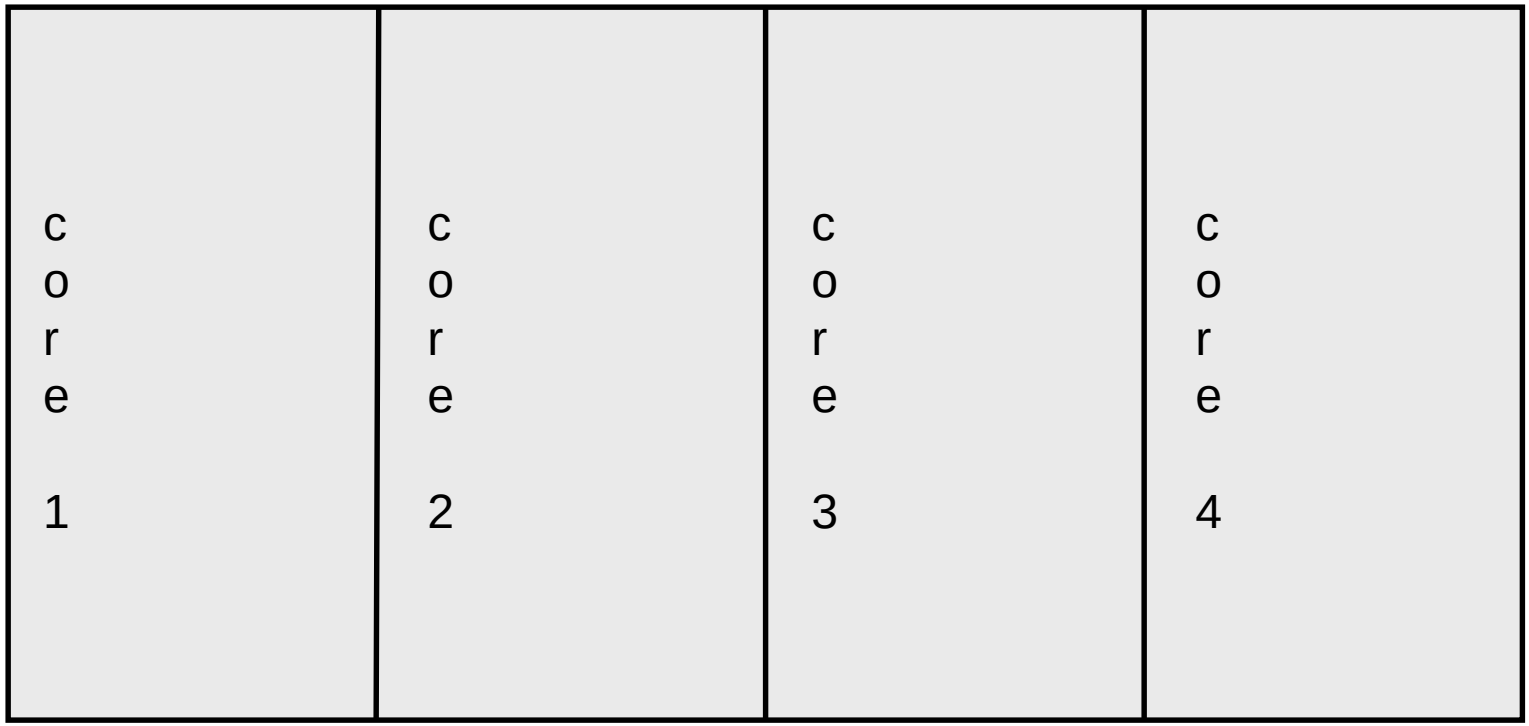
- This lecture is about a new trend in computer architecture:
  Replicate multiple processor cores on a single die.



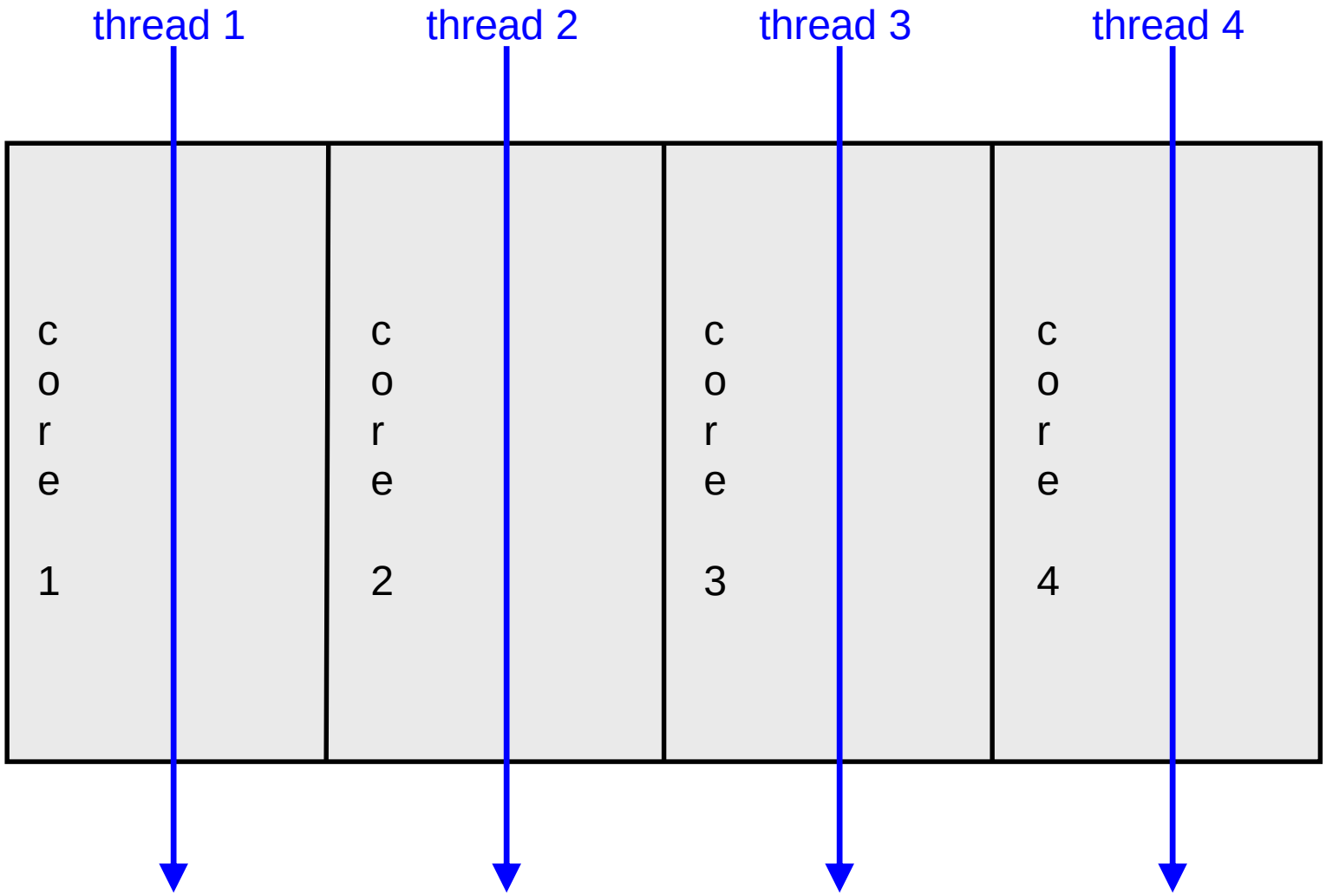
Multi-core CPU chip

Multi-core CPU chip

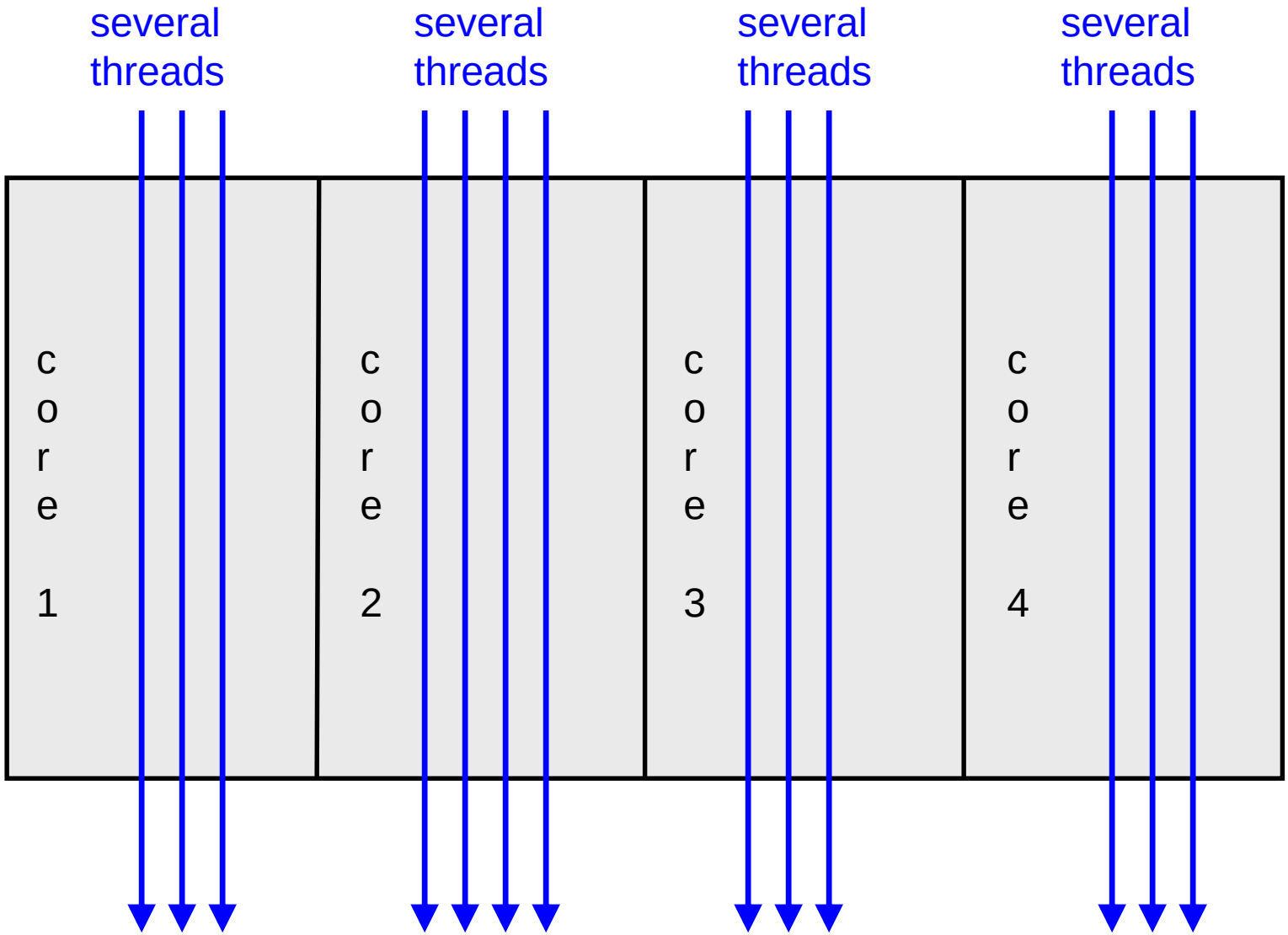
- The cores fit on a single processor socket
- Also called CMP (Chip Multi-Processor)



Threads on different cores run in parallel



On each core, threads are scheduled by OS with time-slicing



Memory architecture

- Shared memory
  - One common shared memory shared by all cores
  - Most common, and focus of this lecture
- Distributed memory
  - Each core has its own local memory
  - Only appears in manycore processors (up to a few hundred cores)

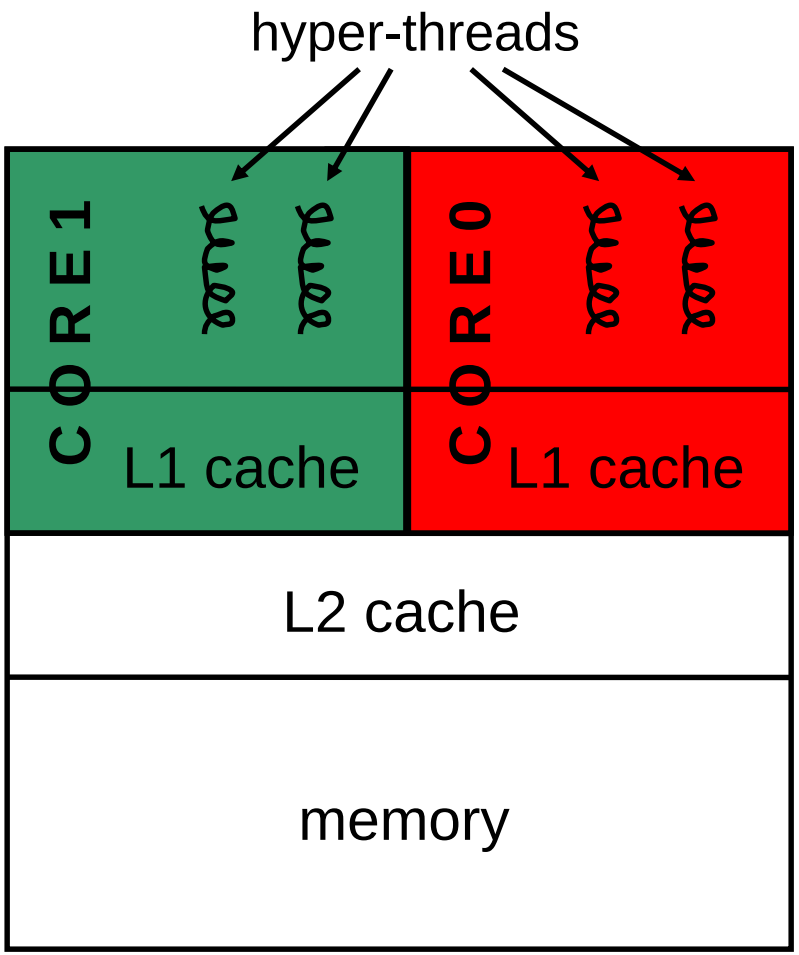
The memory hierarchy

- L1 caches always private to each core
- L2/L3 caches private in some architectures, and shared in others
- Memory is always shared

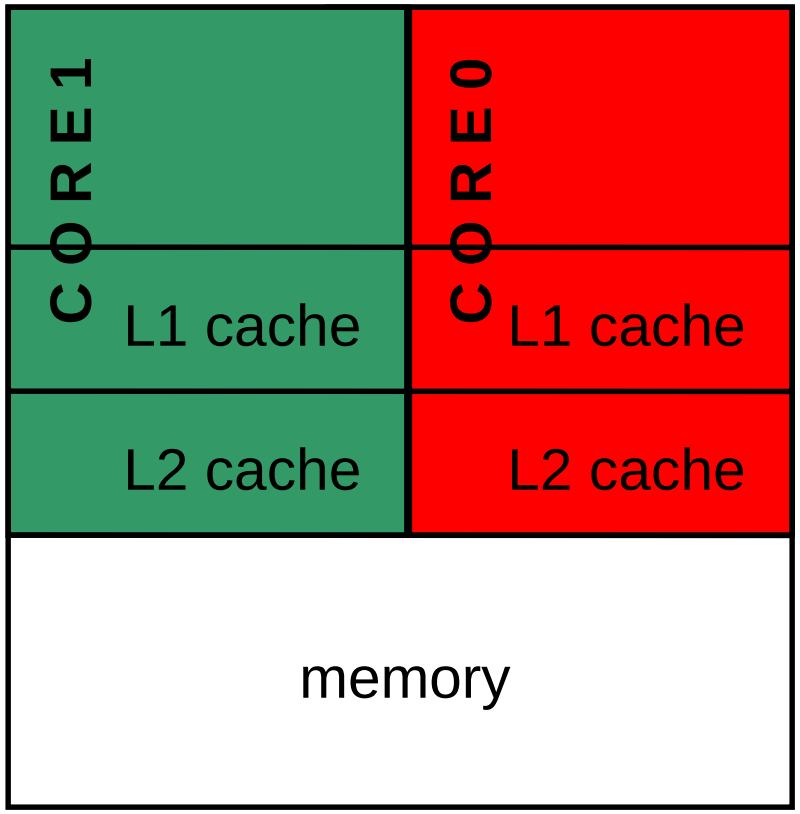
Intel Xeon processors

- Dual-core
  Intel Xeon processor

- L1 caches are private
- L2 cache is shared

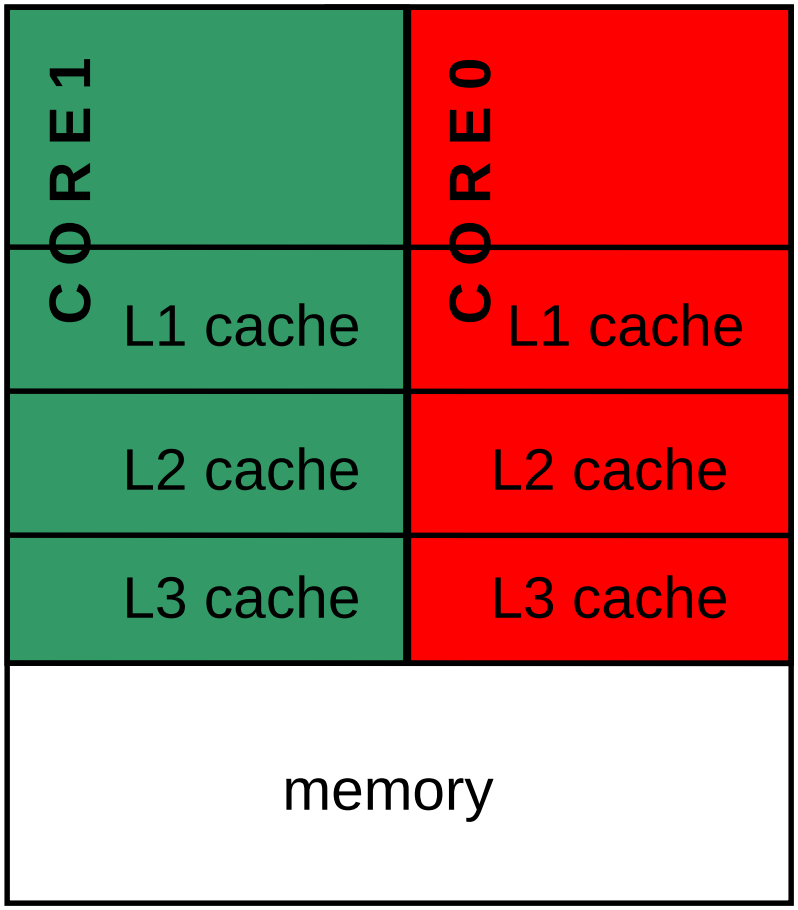


Designs with private L2 caches



L1/L2 caches are private

Examples: AMD Opteron, AMD Athlon, Intel Pentium D



L1/L2/L3 caches are all private

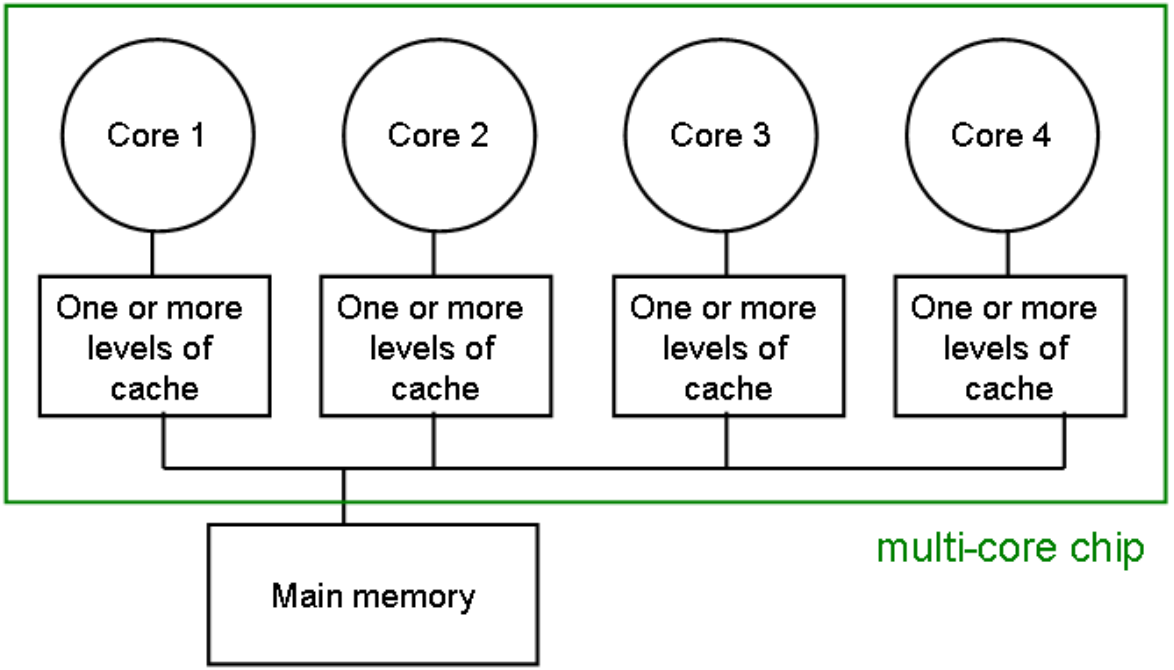
Example: Intel Itanium 2

Private vs shared caches

- Advantages of private caches:
  - Better isolation and less contention among threads on different cores
- Advantages of a shared cache:
  - Better sharing of cache space among threads on different cores: inter-thread communication can go through cache and not touch memory

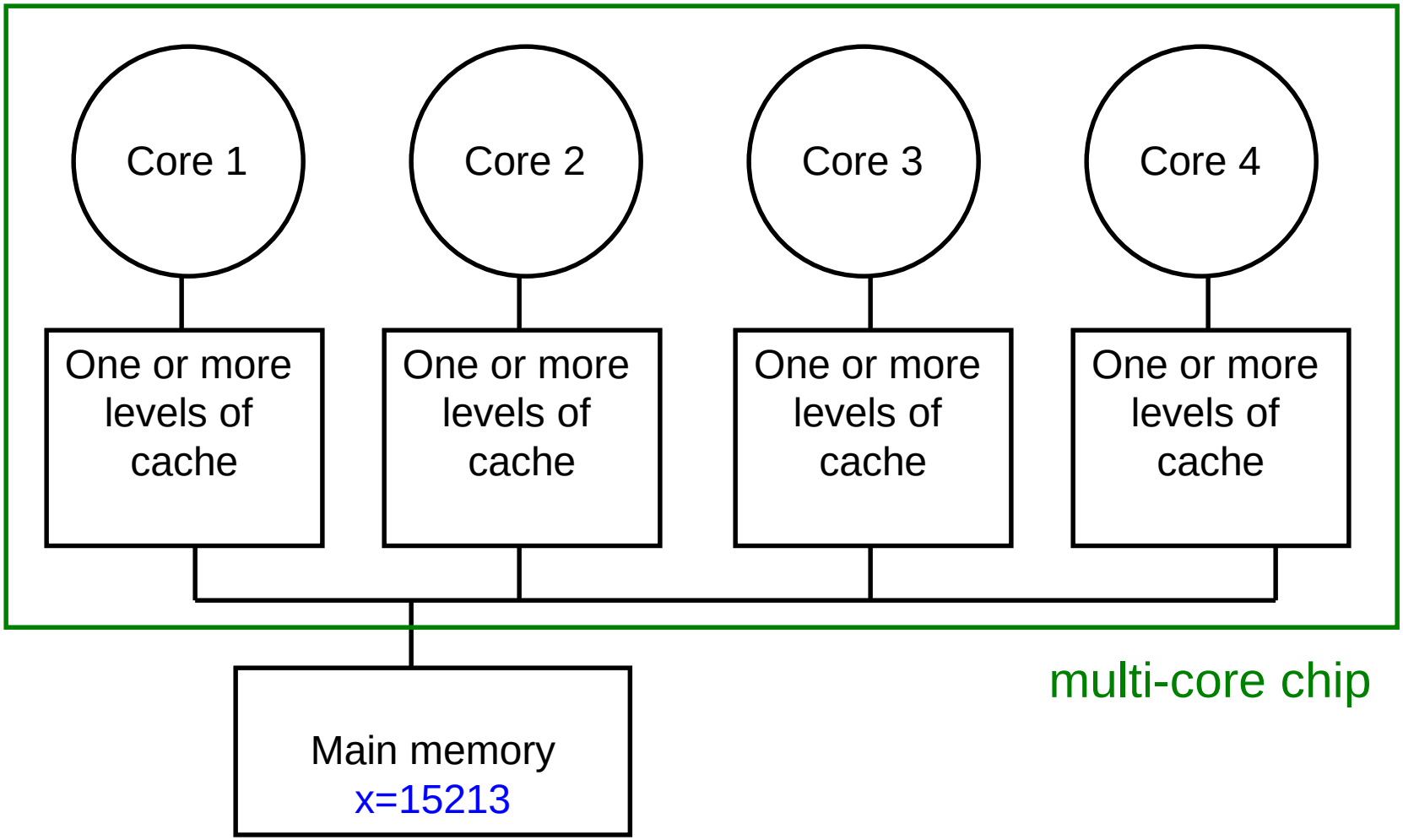
The cache coherence problem

- Since we have private caches:
  How to keep the data consistent across caches on different cores?



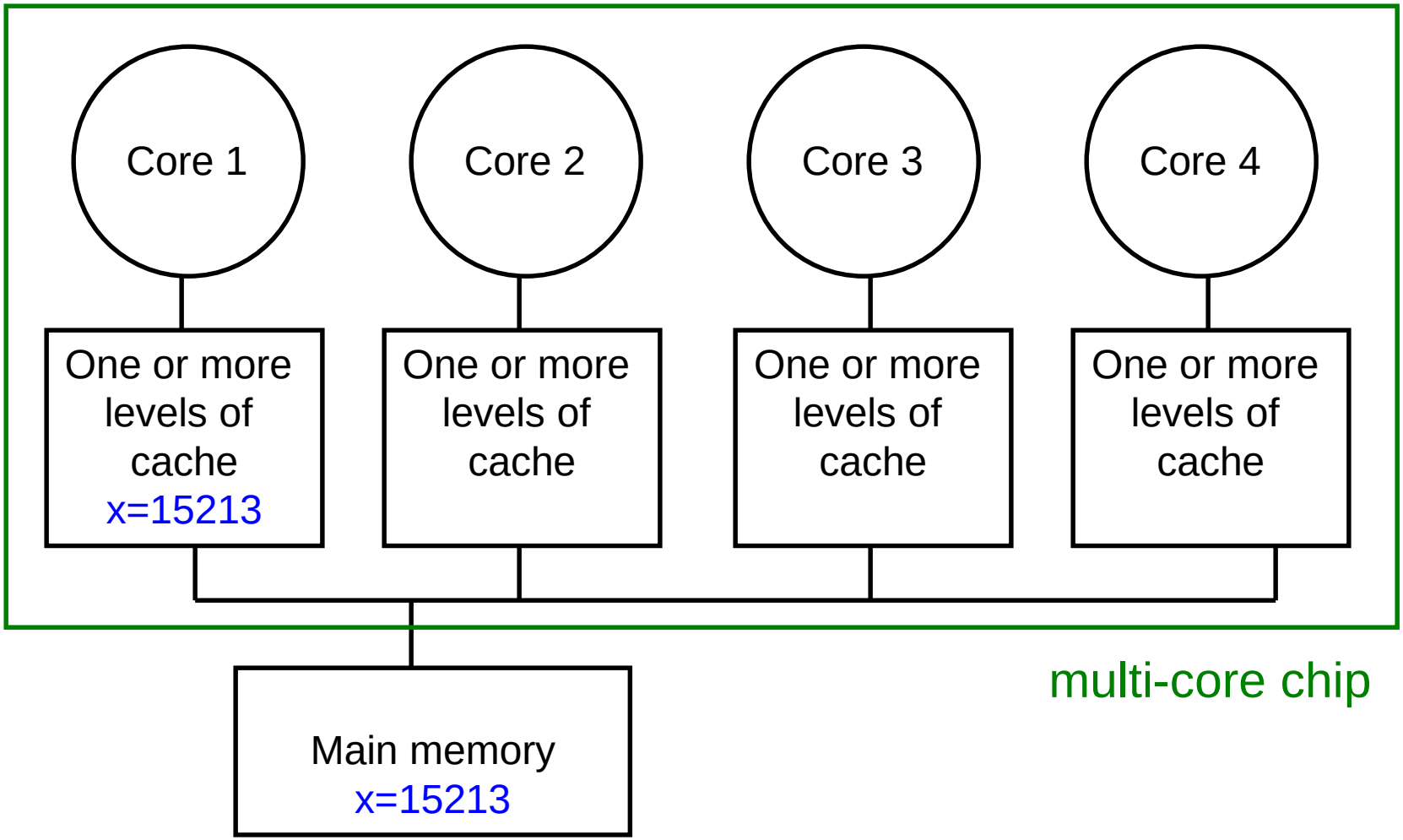
The cache coherence problem

Suppose variable x initially contains 15213



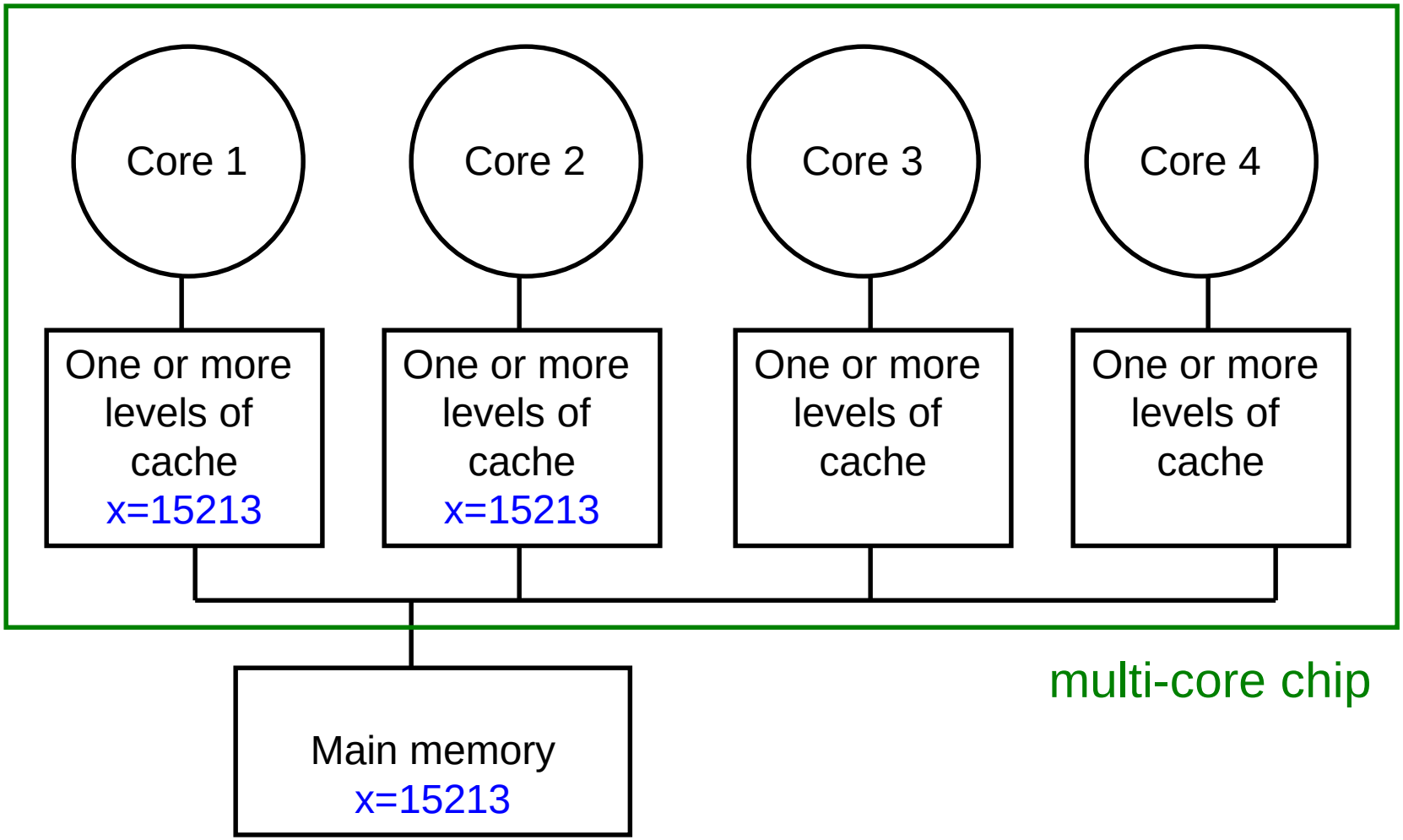
The cache coherence problem

Core 1 reads x



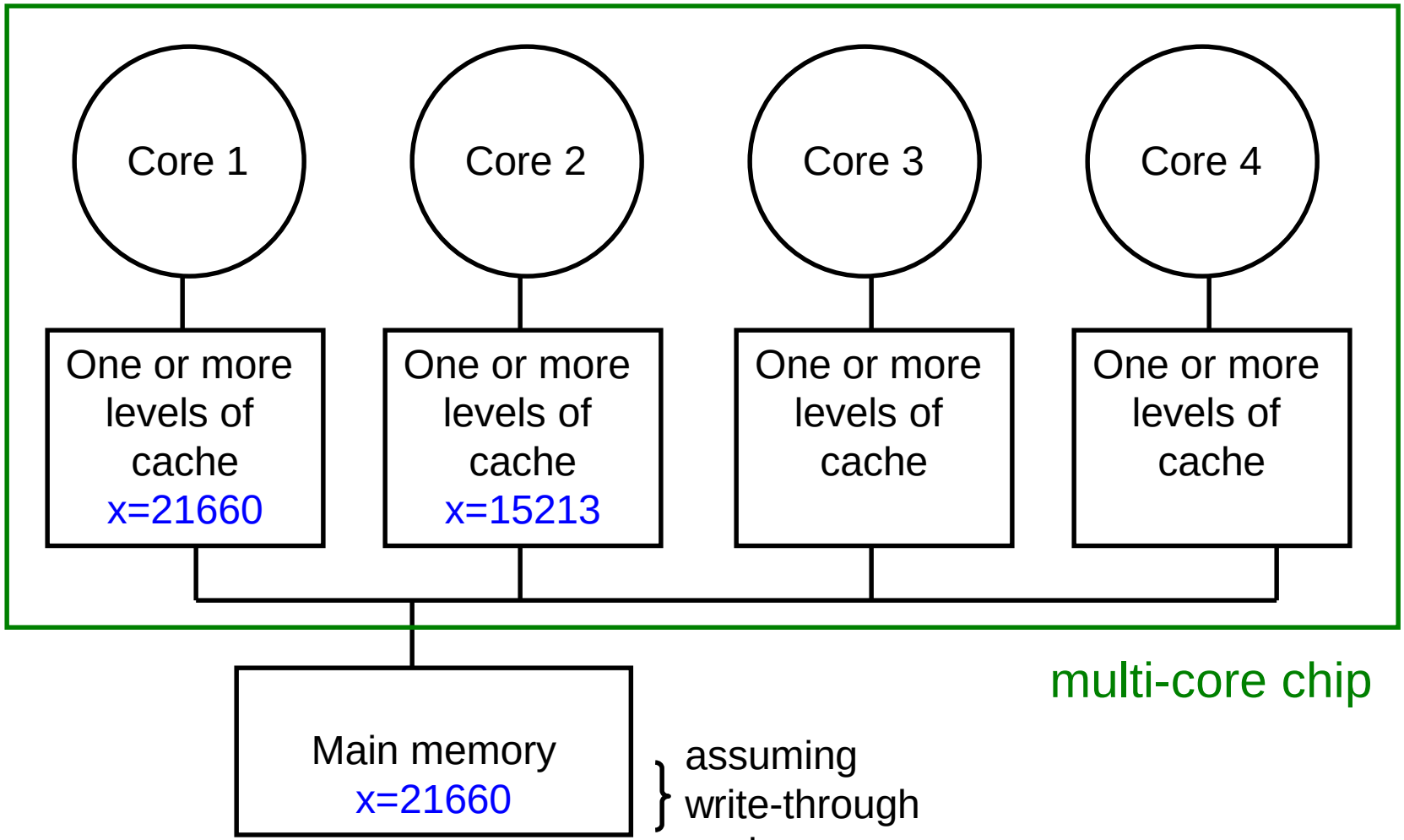
The cache coherence problem

Core 2 reads x



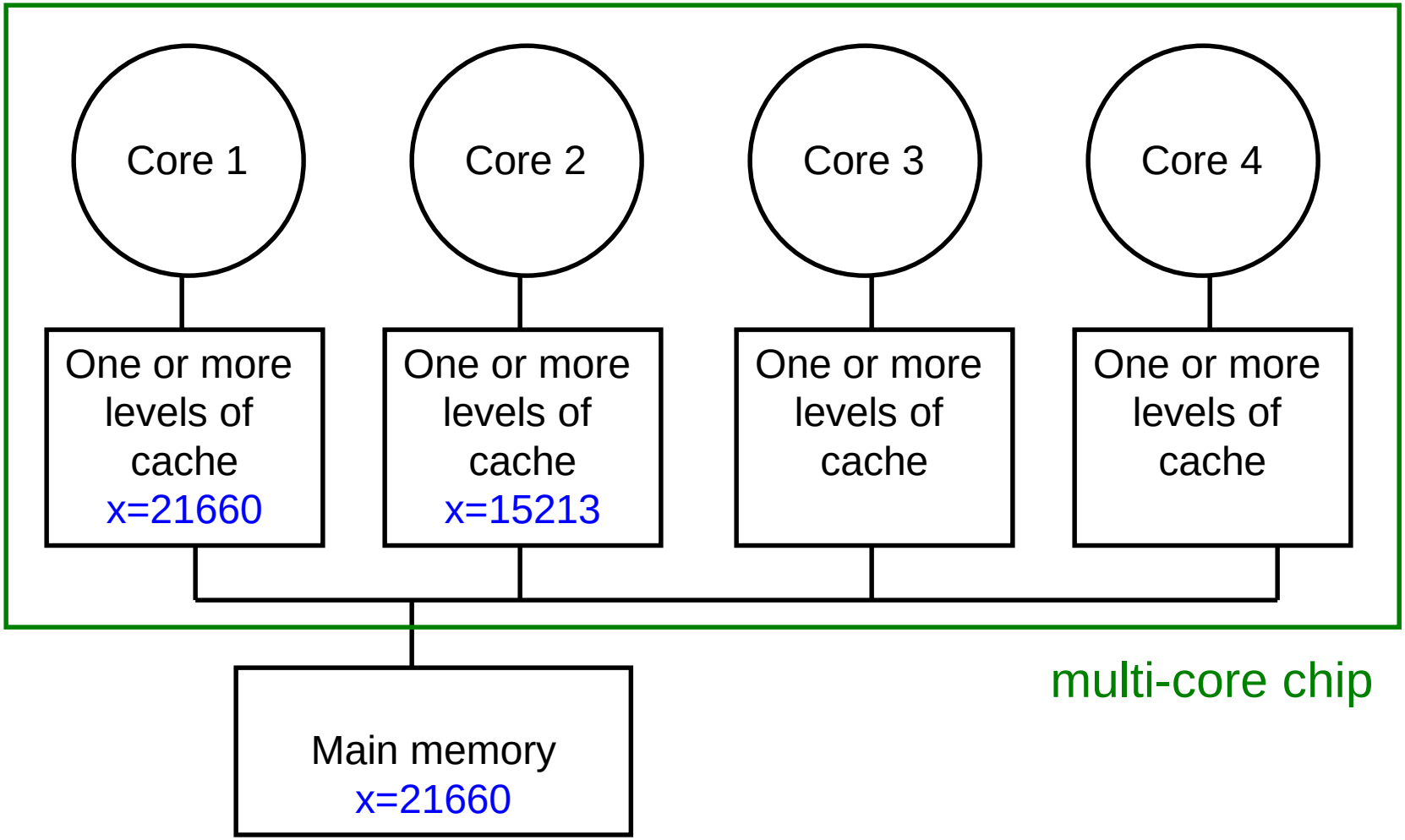
The cache coherence problem

Core 1 writes to x, setting it to 21660



The cache coherence problem

Core 2 attempts to read x… gets a stale copy

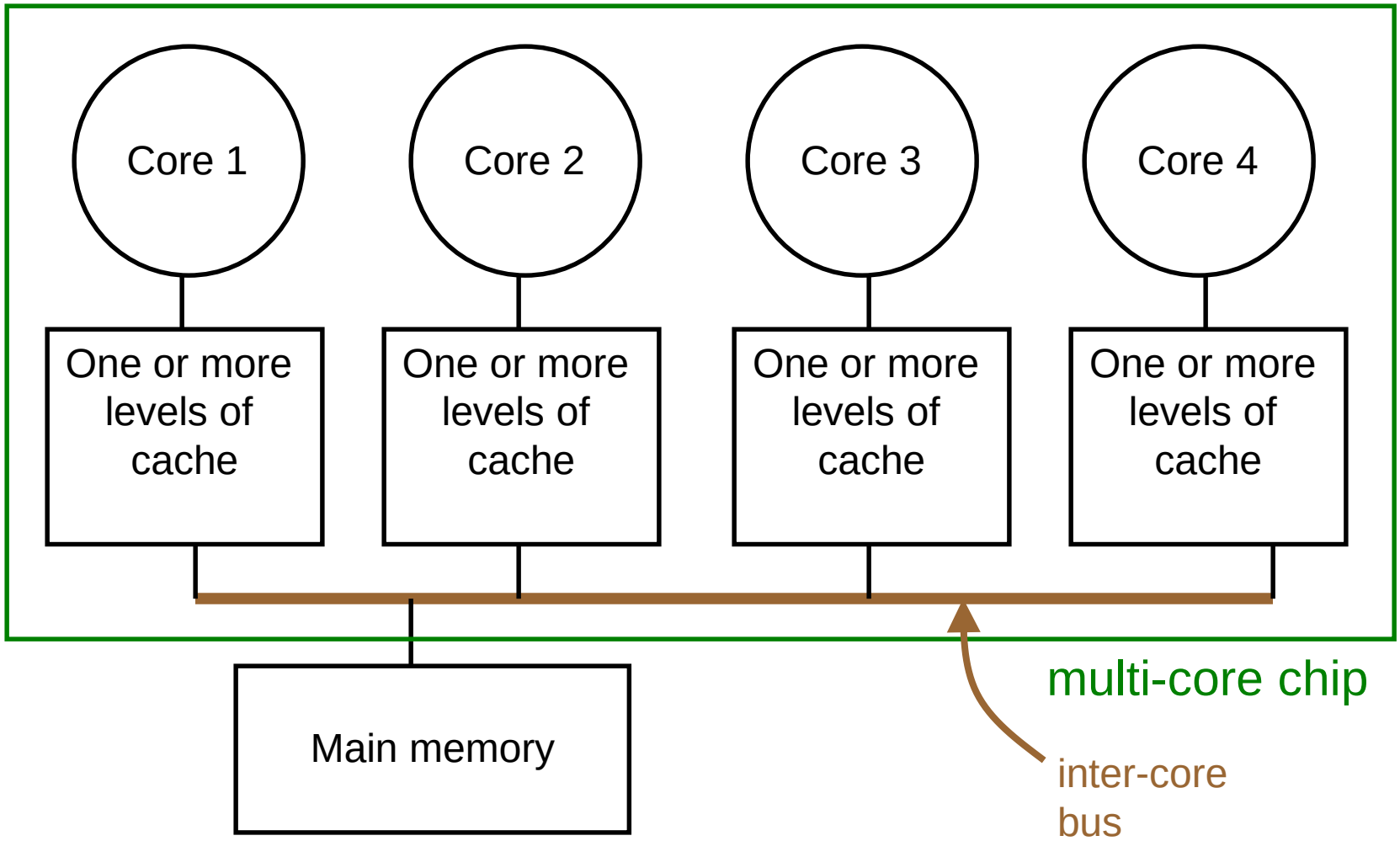


Cache coherence protocols

- Many cache coherence protocols have been designed

- We discuss a simple solution: *invalidation*-based protocol with *snooping*

Inter-core bus

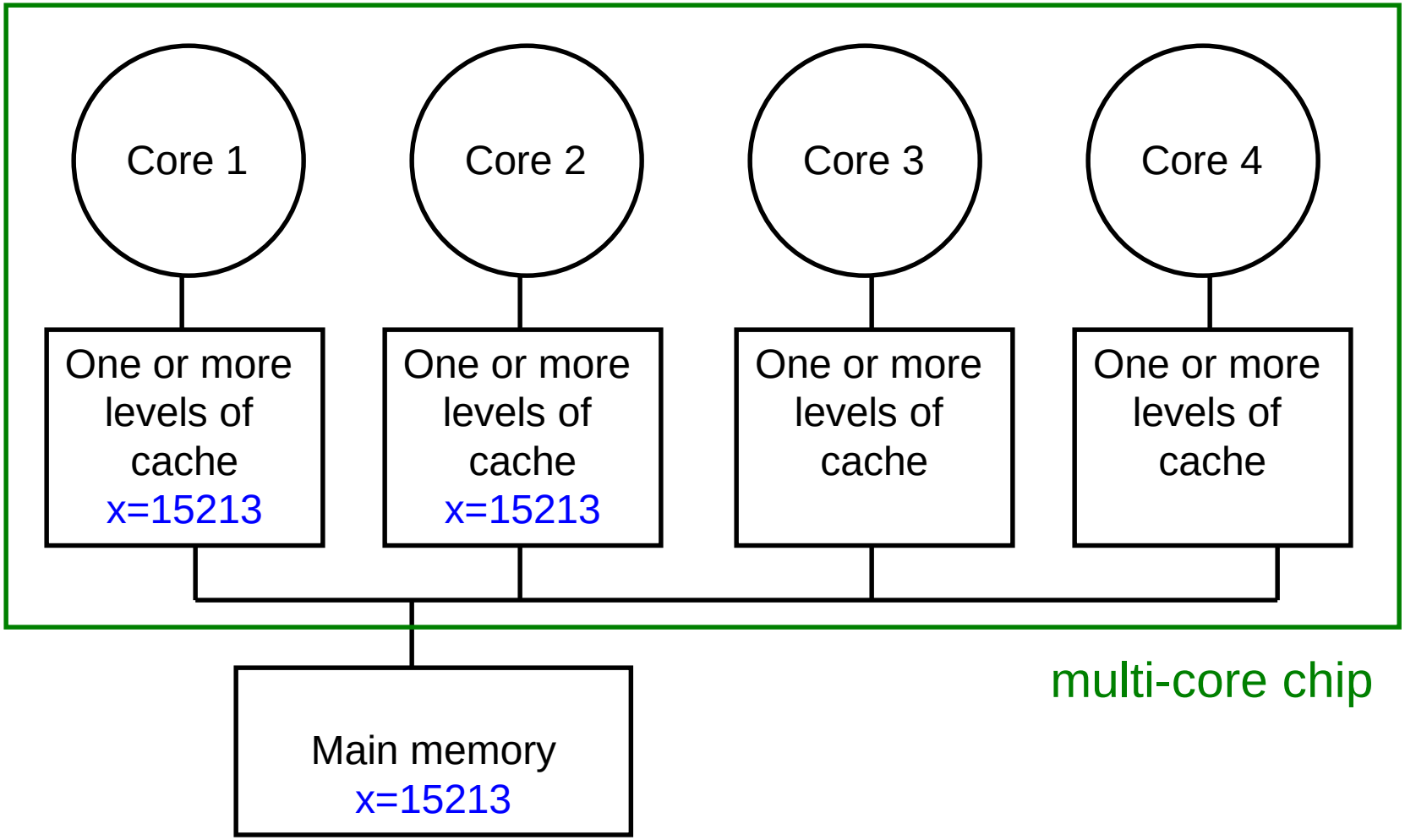


Invalidation protocol with snooping

- 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.

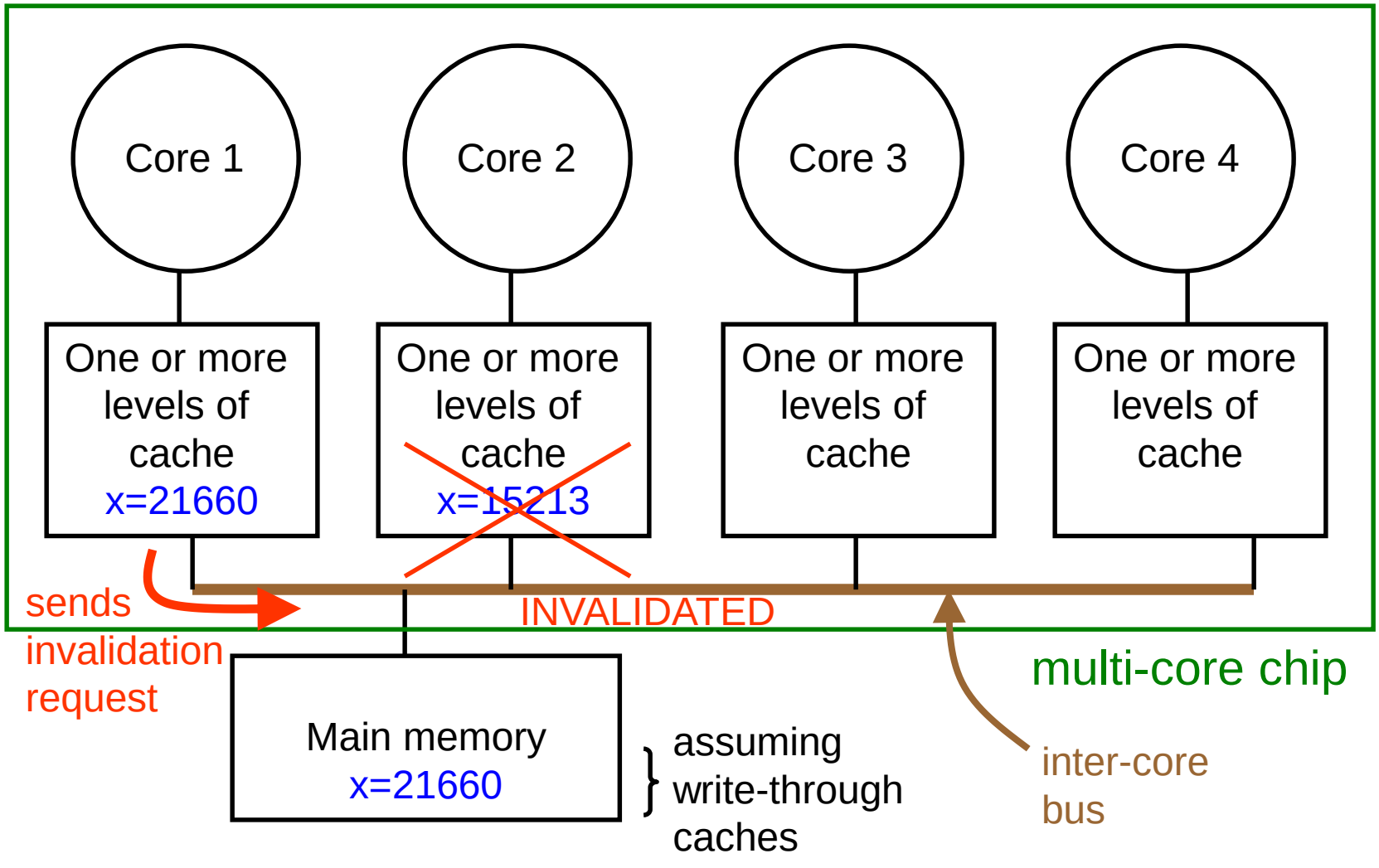
The cache coherence problem

Revisited: Cores 1 and 2 have both read x



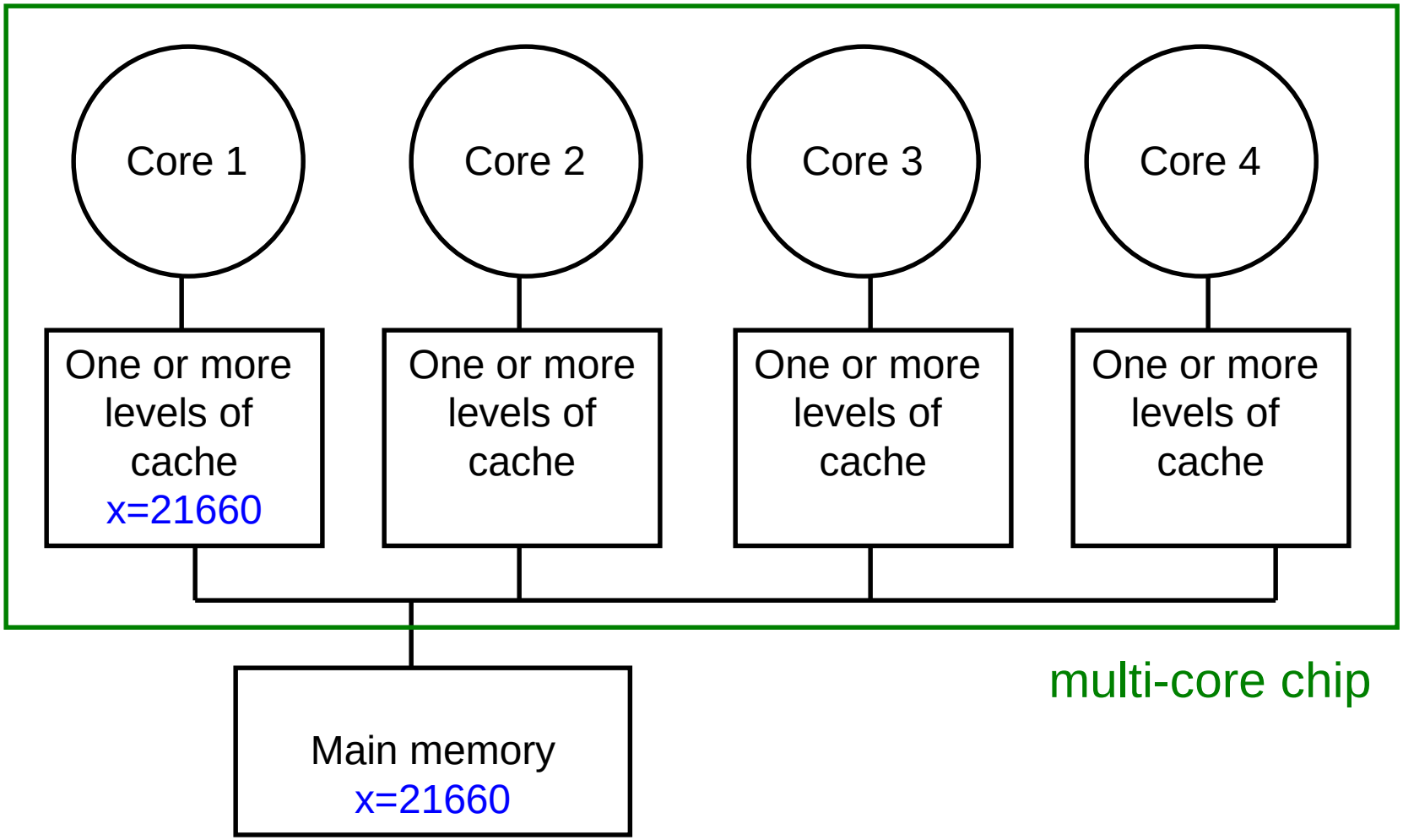
The cache coherence problem

Core 1 writes to x, setting it to 21660



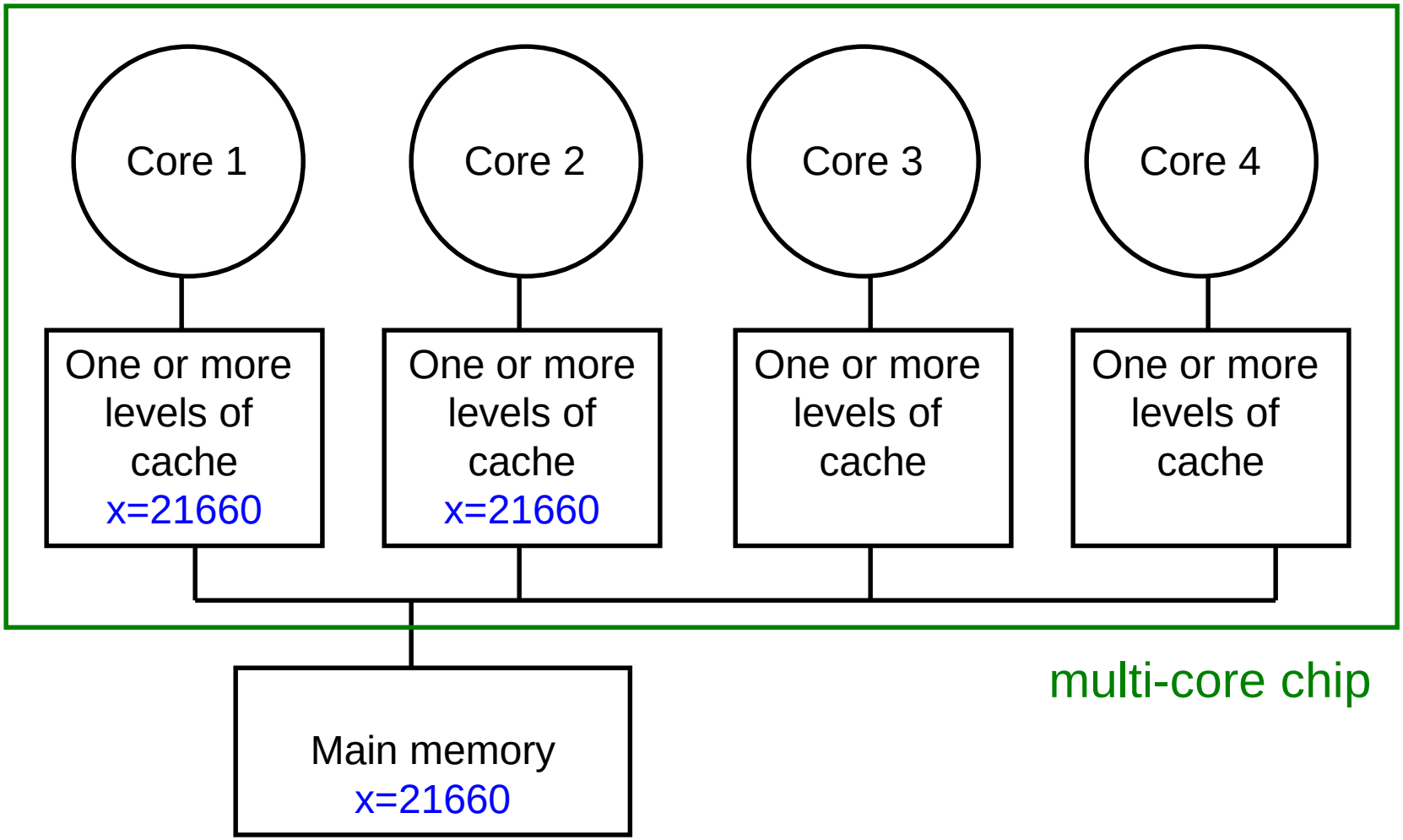
The cache coherence problem

After invalidation:



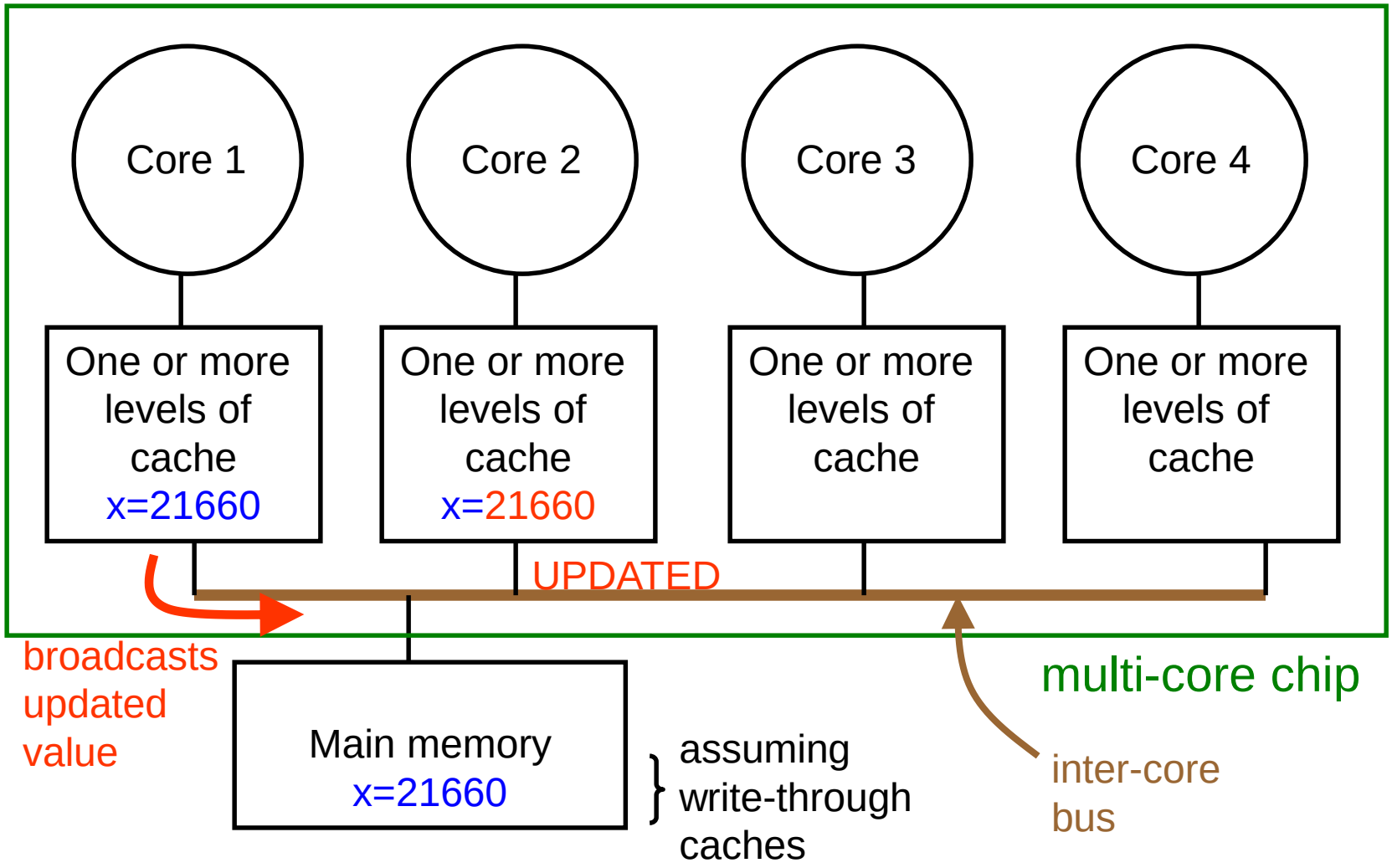
The cache coherence problem

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



Alternative to invalidate protocol: update protocol

Core 1 writes x=21660:



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

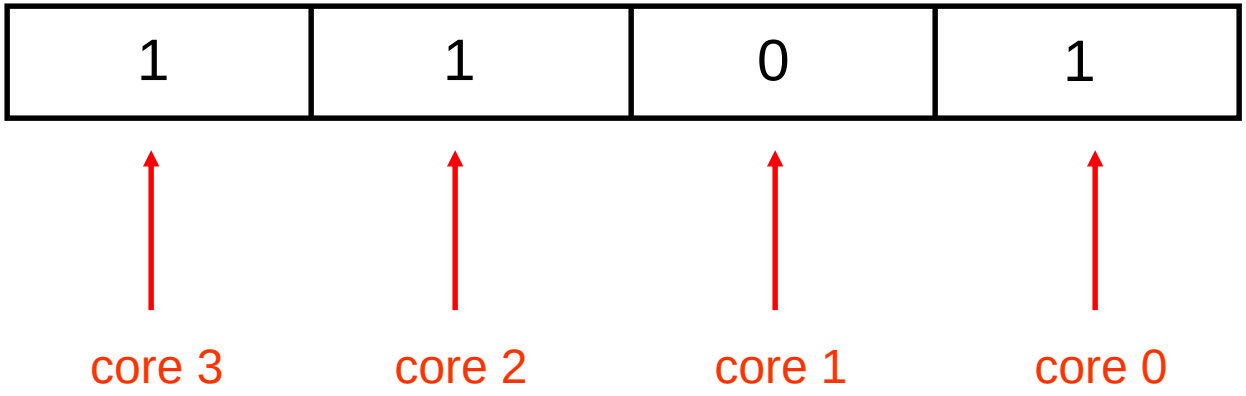
- This was just the basic invalidation protocol
- More sophisticated protocols use extra cache state bits
- MSI, MESI (Modified, Exclusive, Shared, Invalid)

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, without SMT



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

Default affinity mask

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

- The OS scheduler decides what threads run on what core
  - detects skewed workloads and performs load balancing on multiple cores

Soft affinity

- Thread migration between cores is costly
  - The execution pipeline is stopped on the old core, and restarted on the new core
  - The private cache on the new core is “cold” and needs to be “warmed up”
- Therefore, 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 affinity

- If necessary, the programmer can programmatically prescribe hard affinities
- Example use cases:
  - If multiple threads share data and communicate frequently, then map them to same core for more efficient communication
  - If a thread has hard and tight timing constraints, e.g., it runs a real-time robot controller, then assign it to its own dedicated core, and disallow migration of other threads to this core

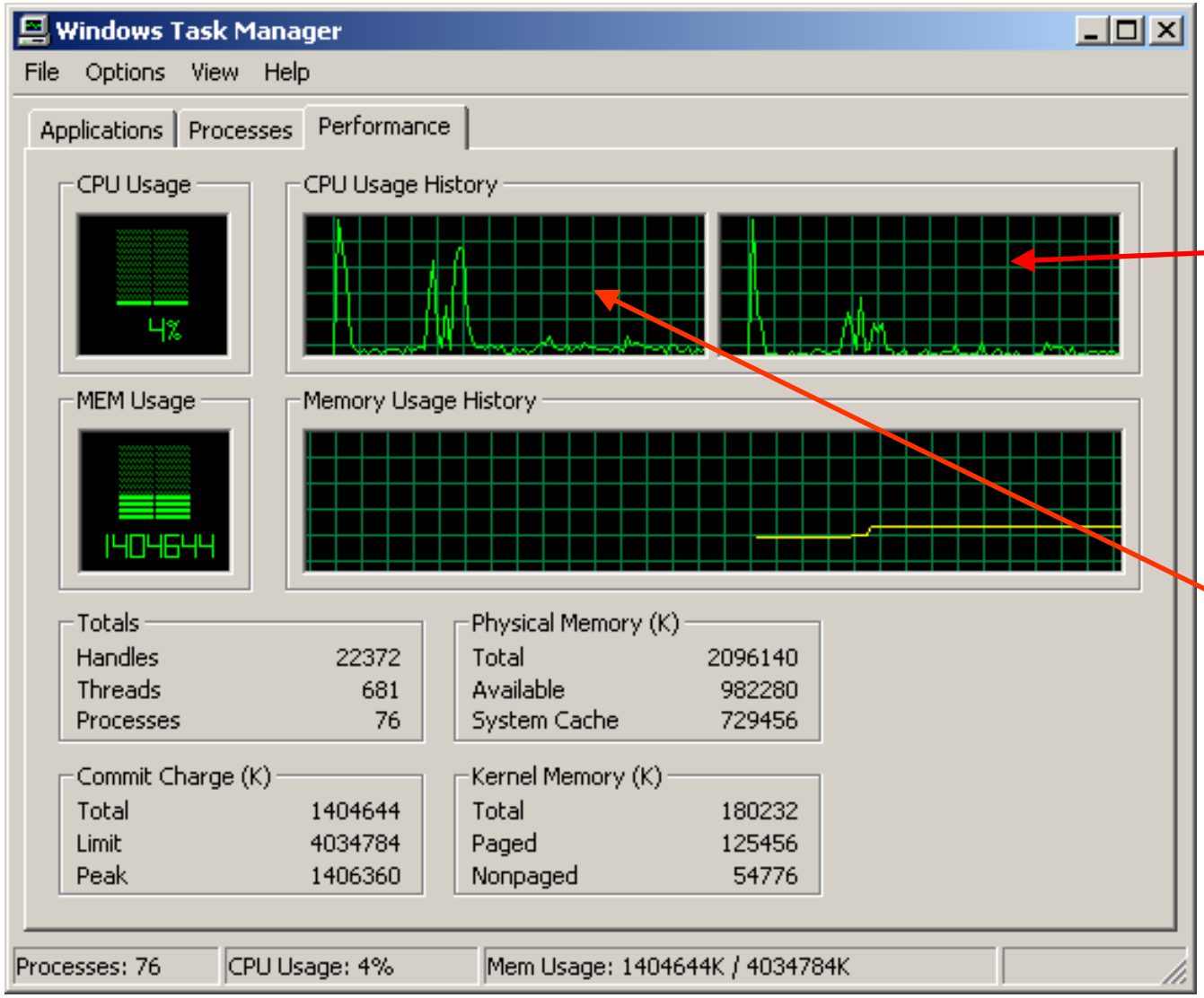
Linux kernel API for hard affinity

```
#include <sched.h>
int sched_getaffinity(pid_t pid,
  unsigned int len, unsigned long * mask);
int sched_setaffinity(pid_t pid,
  unsigned int len, unsigned long * mask);
```

Kernel APIs for getting and setting hard affinities

len=sizeof(unsigned int long)

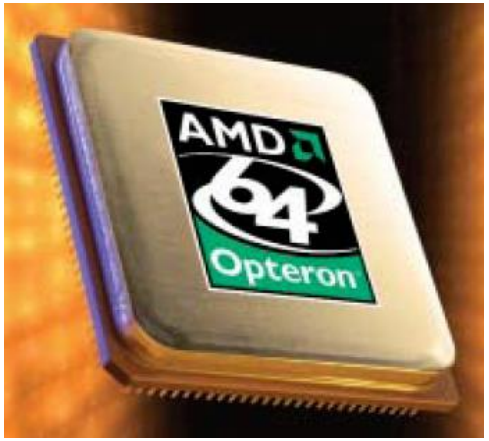
Windows Task Manager



core 2

core 1

Conclusion



- Multi-core chips an important new trend in computer architecture

- Several new multi-core chips in design phases
- Parallel programming techniques likely to gain importance