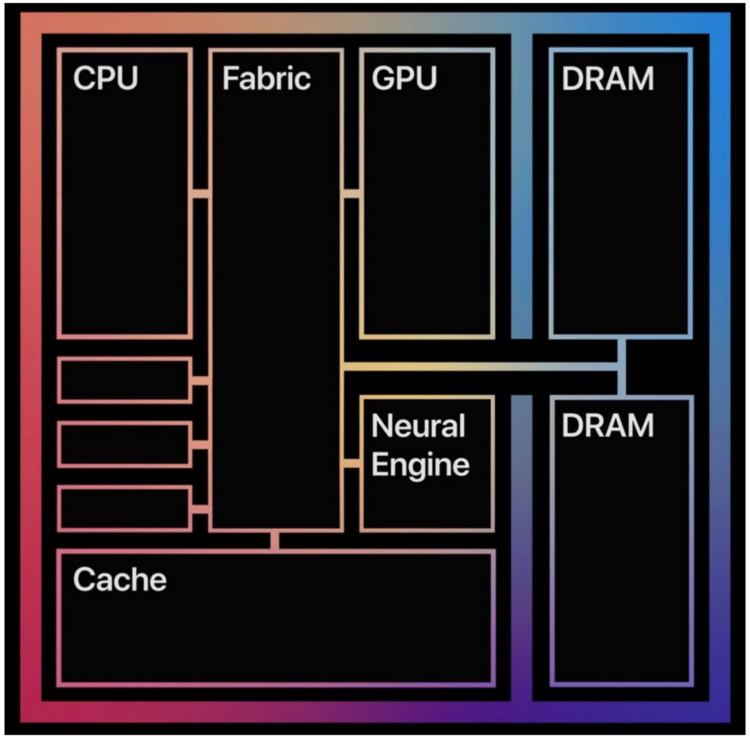
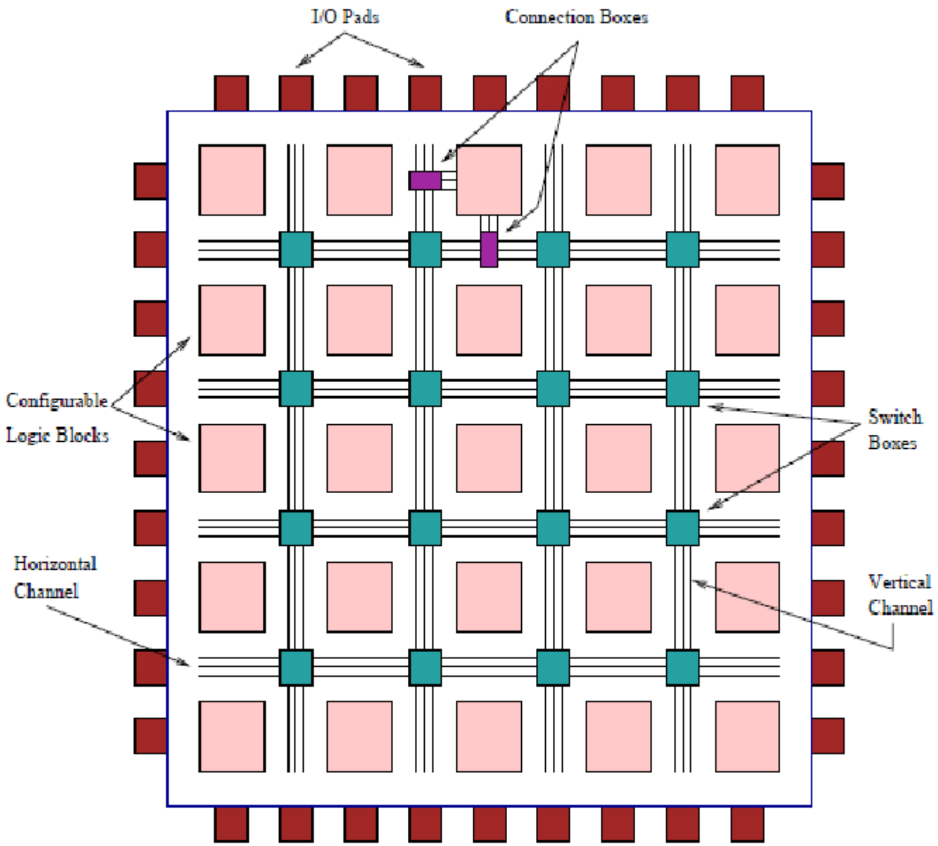
Distributed Systems
Subheadline

- Stefan Henkler **E-Mail: stefan.henkler@hshl.de**

Motivation

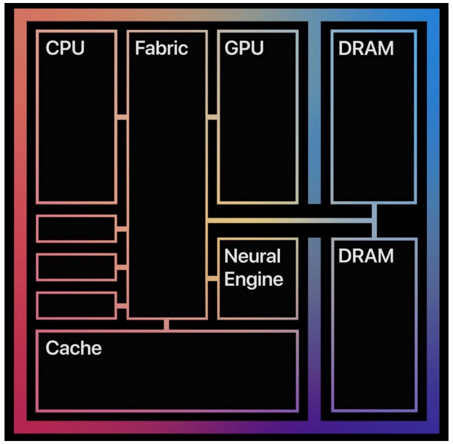
- What is that?

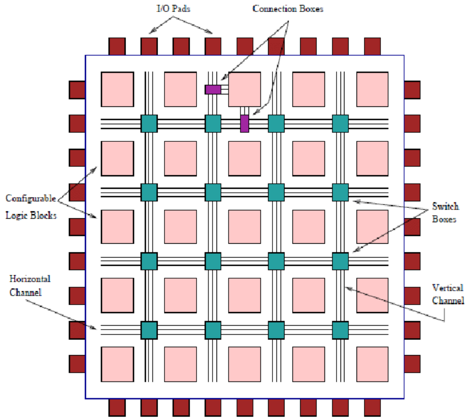




Motivation

Enabler for





Artificial intelligence

Million-dollar babies — The Economist

As Silicon Valley fights for talent, universities struggle to hold on to their stars

WIRED

ARTIFICIAL INTELLIGENCE FINALLY ENTERED OUR EVERYDAY WORLD

Toyota Invests $1 Billion in Artificial Intelligence in U.S.

Autonomes Fahren in Deutschland

Bosch schickt bald Robo-Taxis auf die Straße

Bloomberg Business

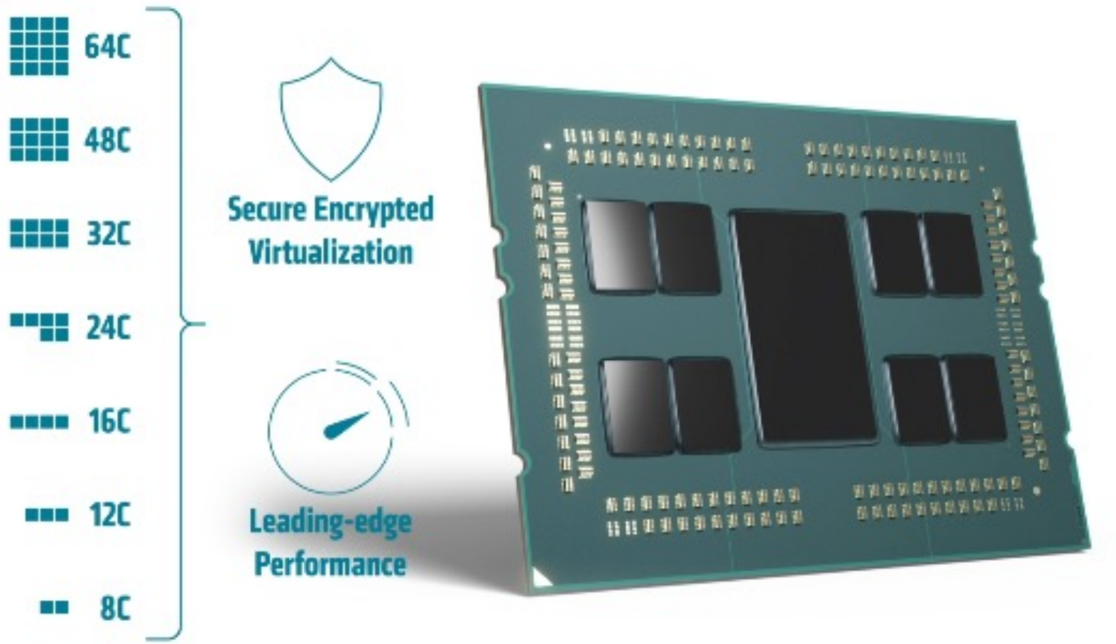
The First Person to Hack the iPhone Built a Self-Driving Car. In His Garage

Microsoft and Baidu partner to bring AI powered vehicles to the automotive market

Overview

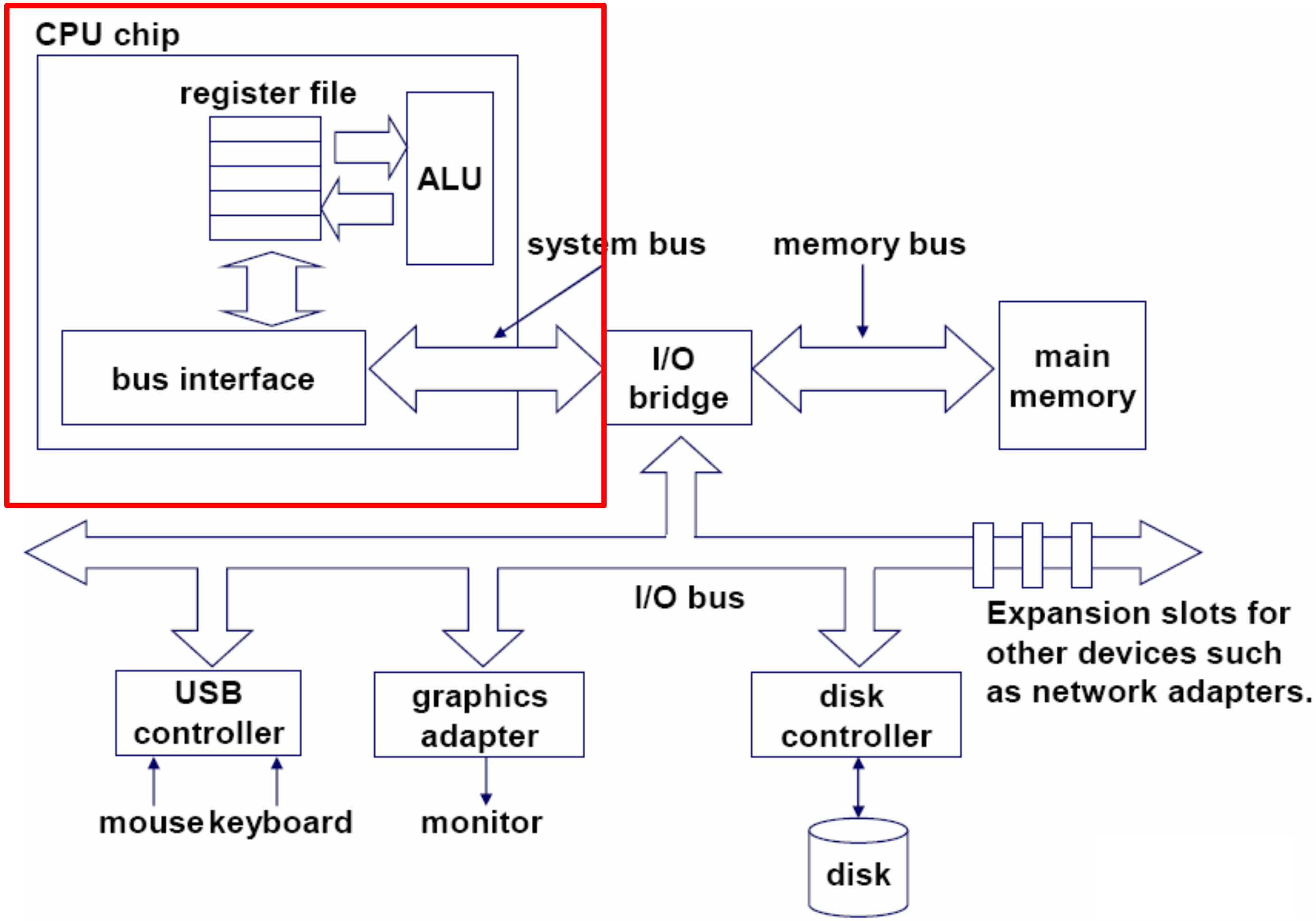
Multi-core

The secret is under the hood

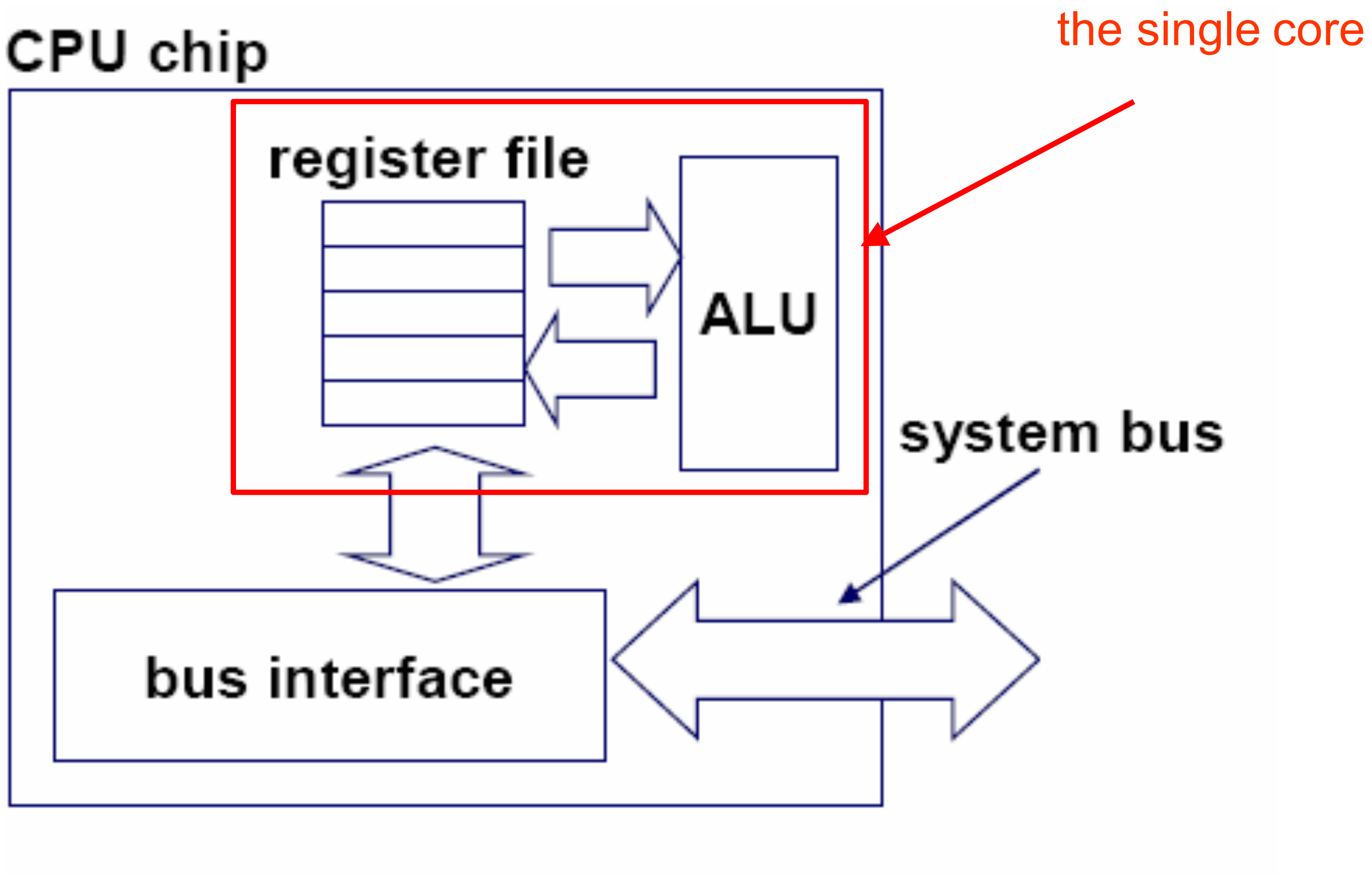


https://www.amd.com/en/processors/epyc-7002-series

Single-core computer

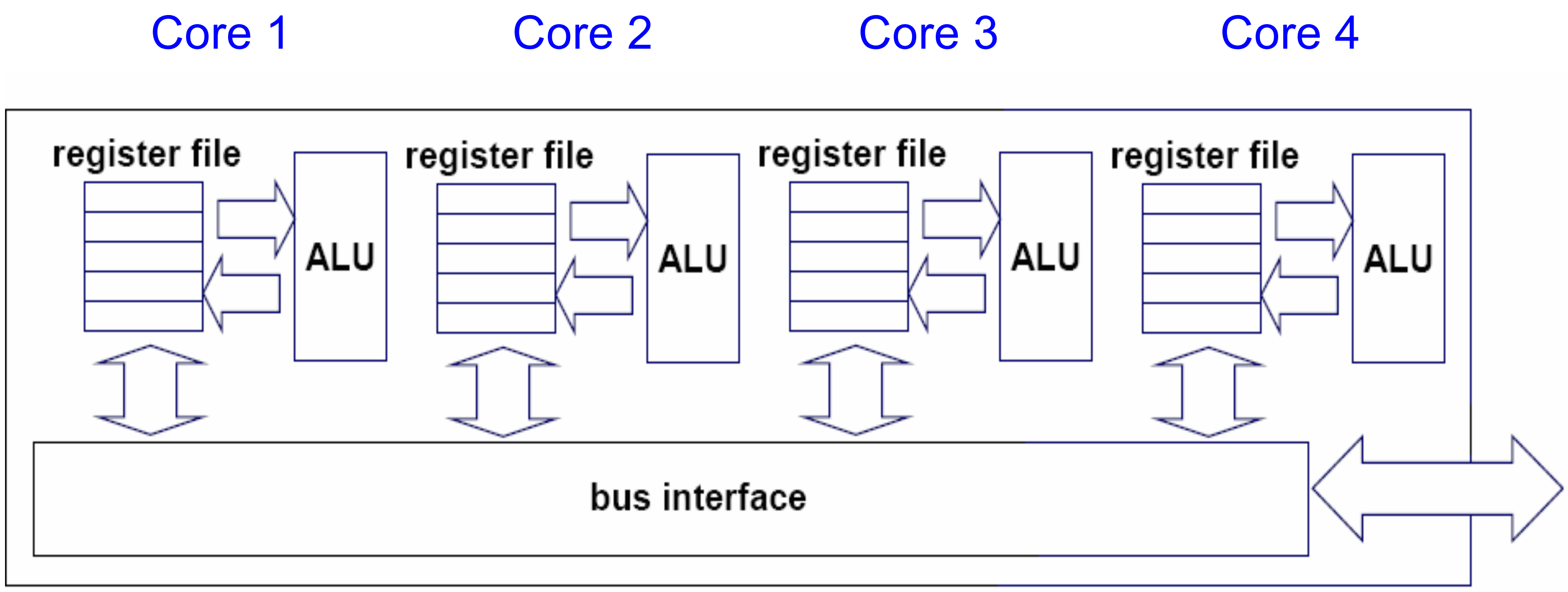


Single-core CPU chip



Multi-core architectures

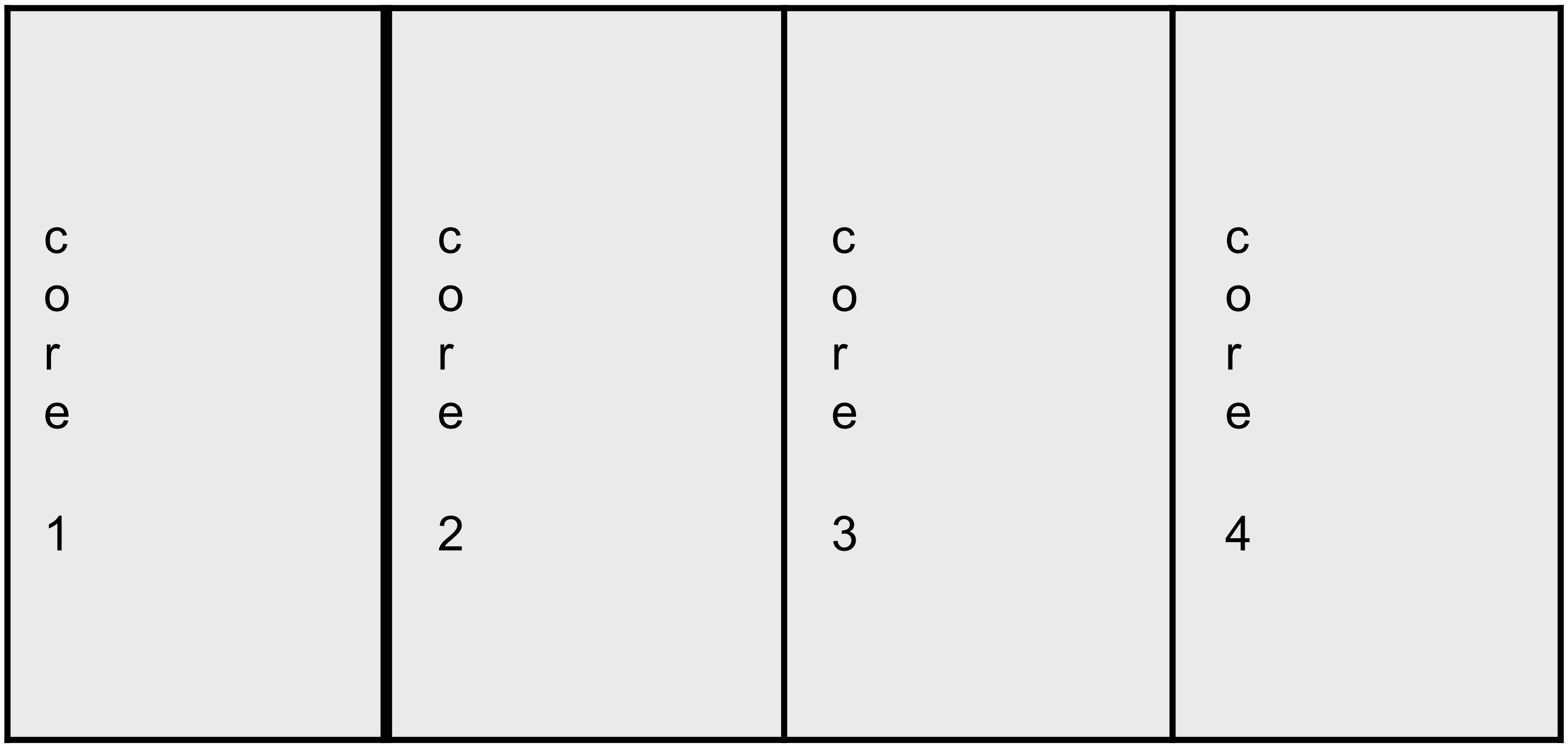
Replicate multiple processor cores on a single die.



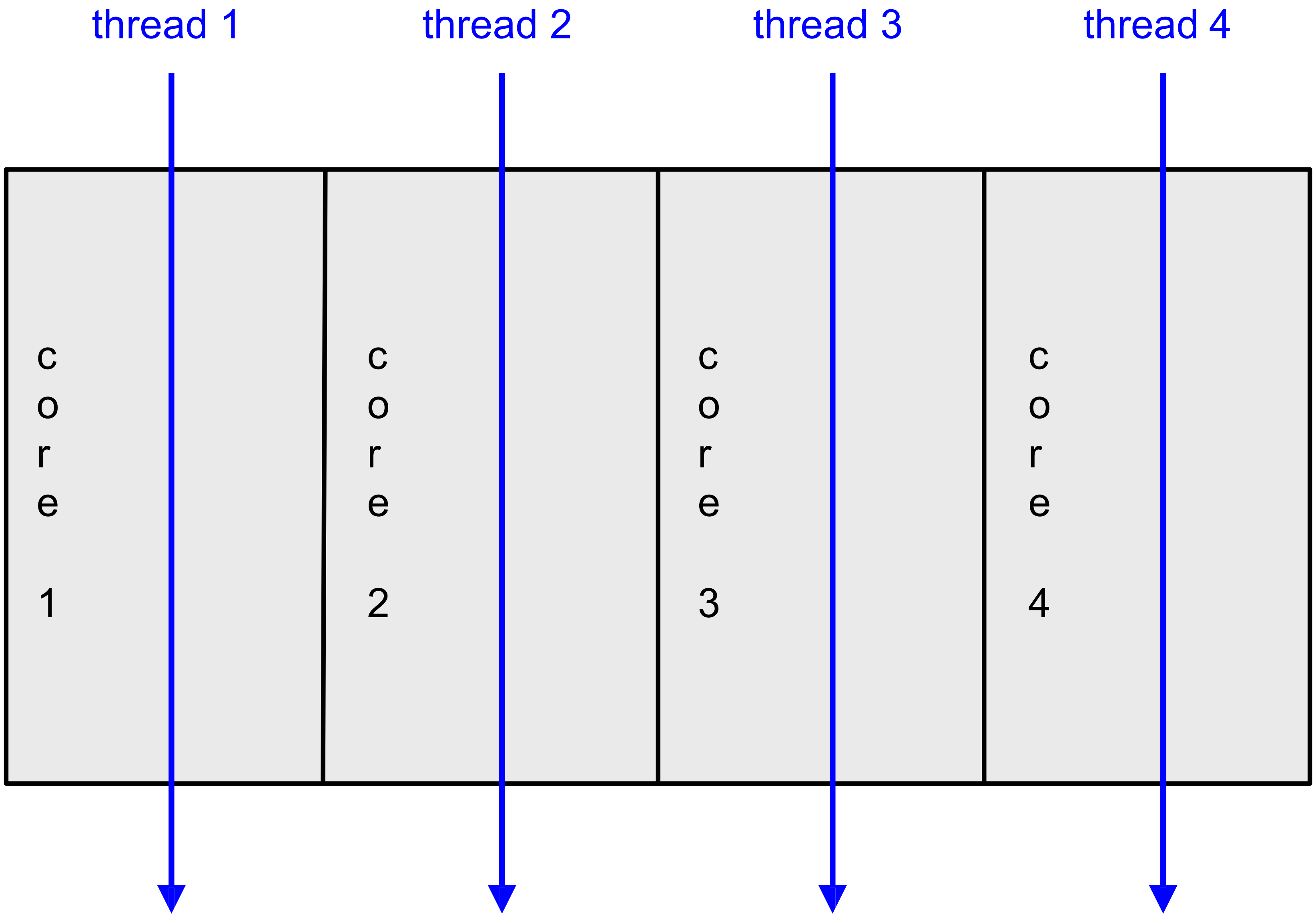
Multi-core CPU chip

Multi-core CPU chip

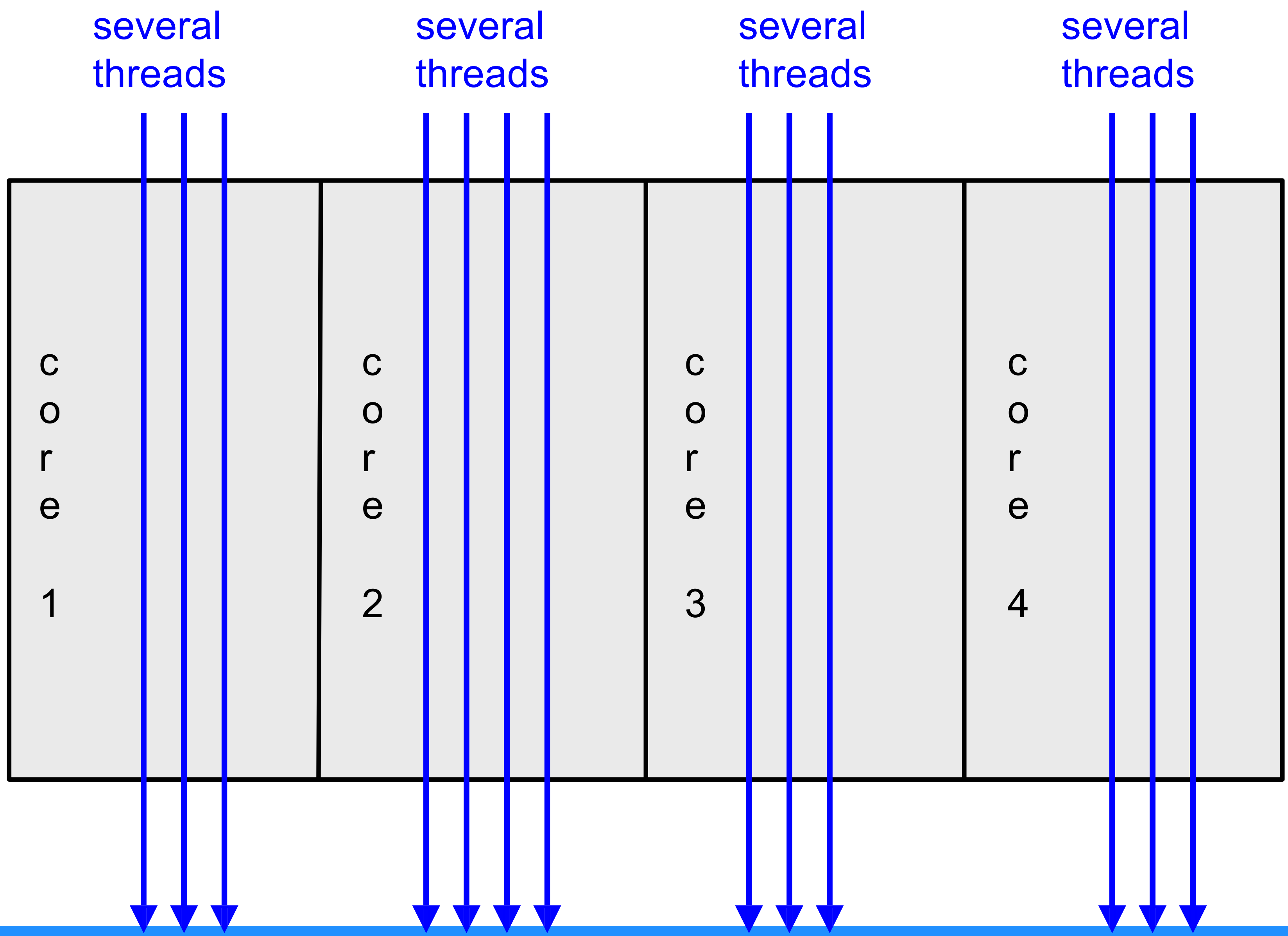
- The cores fit on a single processor socket
- Also called CMP (Chip Multi-Processor)



The cores run in parallel



Multicore and time slicing



Interaction with OS

- OS perceives each core as a separate processor
- OS scheduler maps threads/processes to different cores
- Most major OS support multi-core today

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
- E.g. 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
  - one (large) common shared memory for all processors
- Distributed memory
  - each processor has its own (small) local memory
  - its content is not replicated anywhere else

Multi-core processor

Special kind of a multiprocessor

- All processors are on the same chip

- Multi-core processors are MIMD:
  Different cores execute different threads (Multiple Instructions), operating on different parts of memory (Multiple Data).

- Multi-core is a shared memory multiprocessor:
  All cores share the same memory

Examples

- 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)

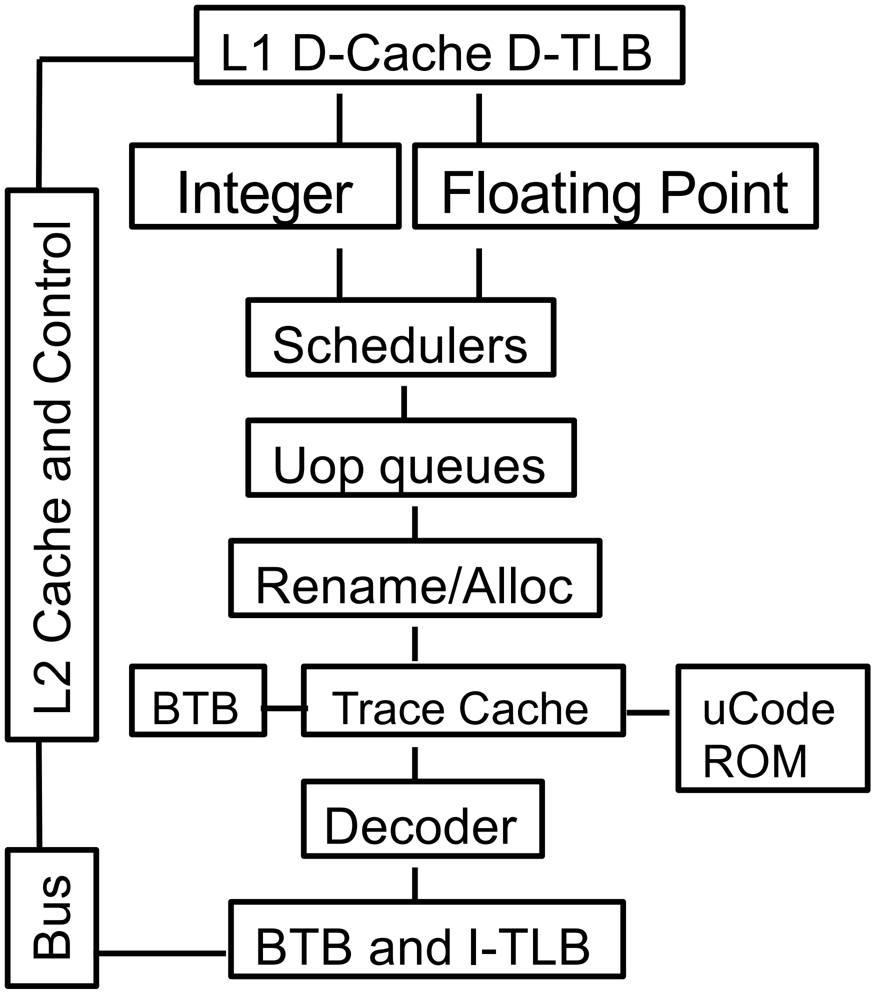
More examples

- Editing a photo while recording a TV show through a digital video recorder
- Downloading software while running an anti-virus program
- “Anything that can be threaded today will map efficiently to multi-core”
- BUT: some applications difficult to parallelize

Simultaneous multithreading

- Problem (example)
  - Waiting for the result of a long floating point (or integer) operation
  - Waiting for data to arrive from memory

  Other execution units wait unused



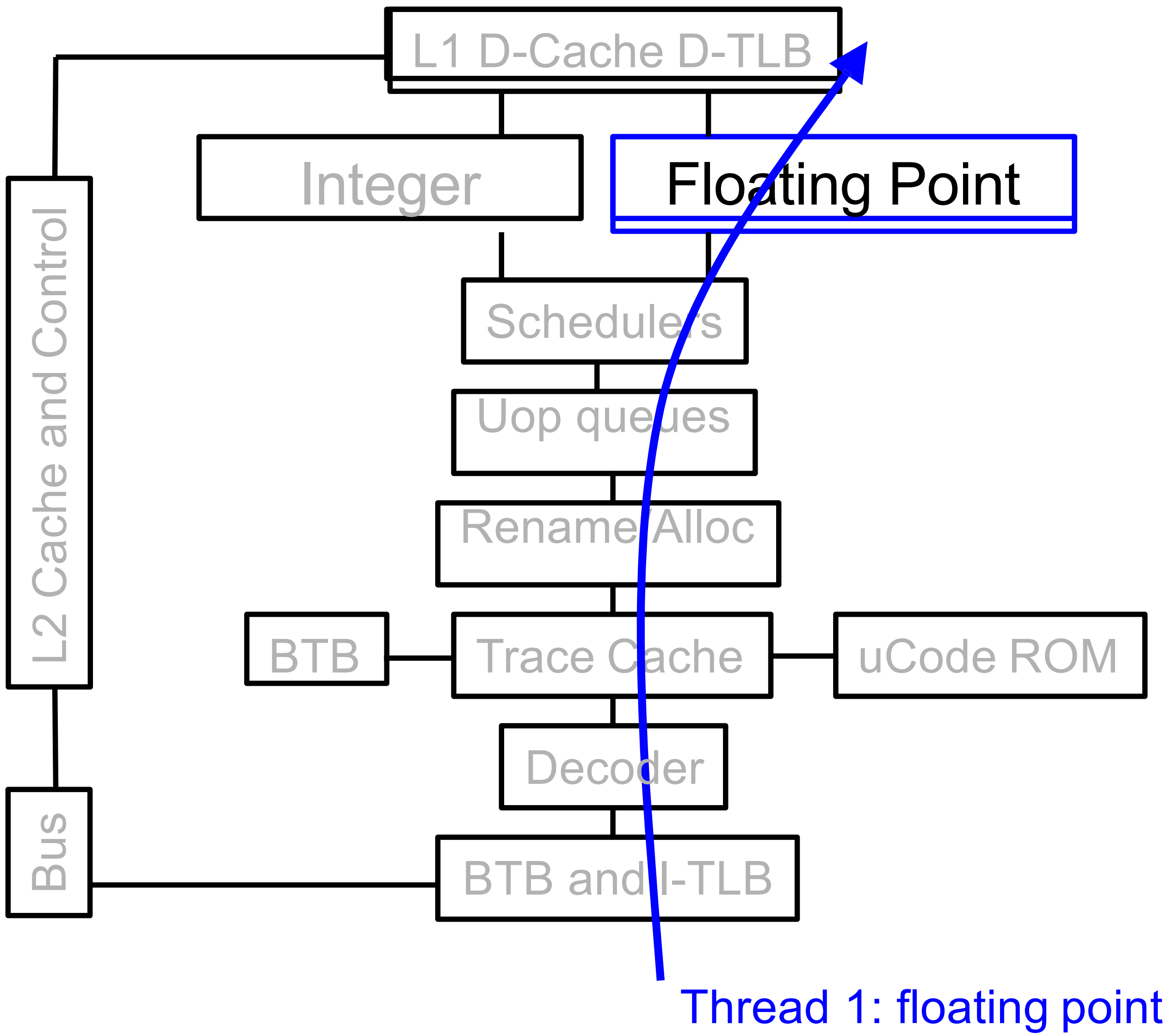
Source: Intel

Simultaneous multithreading (SMT)

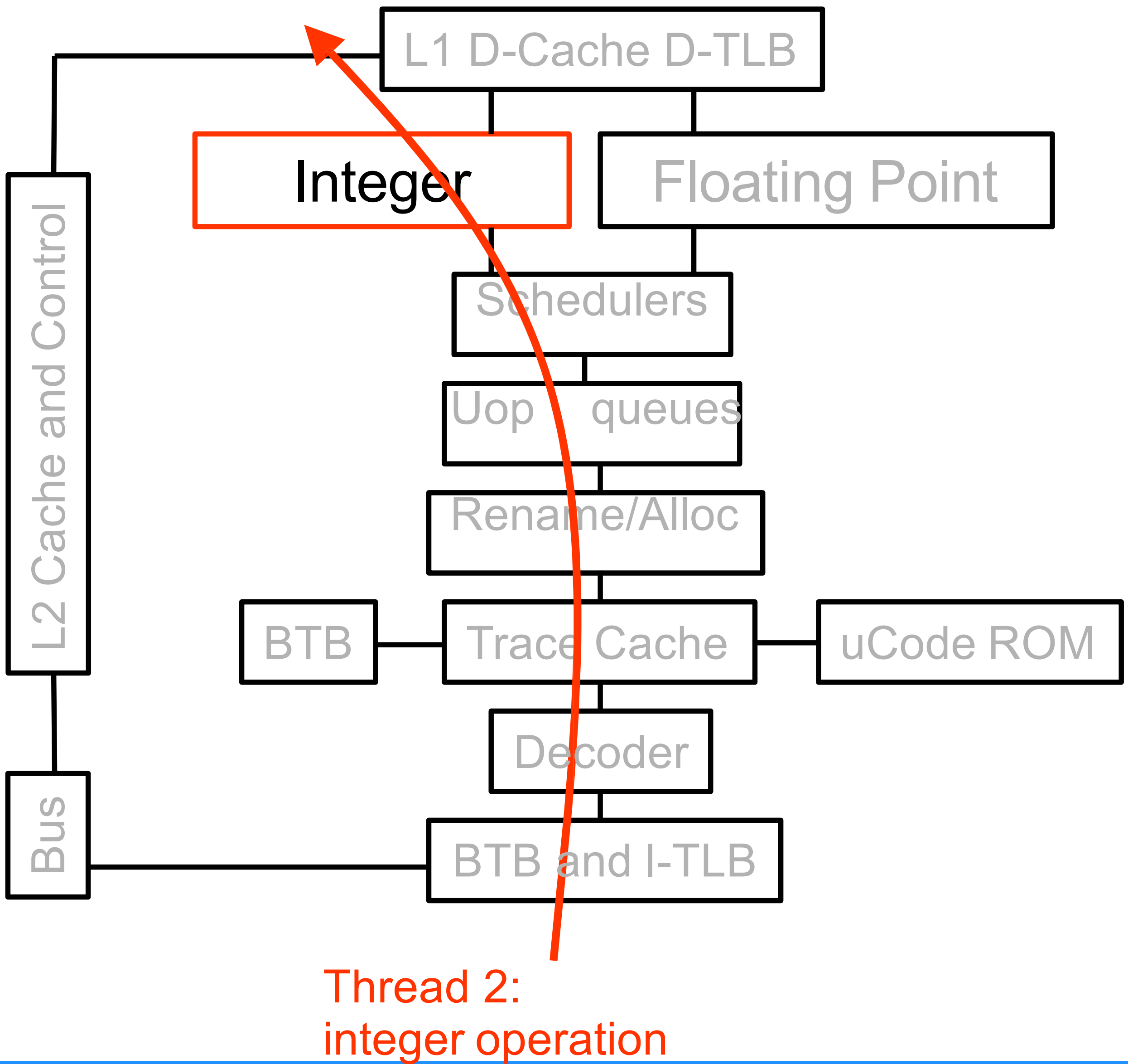
- Permits multiple independent threads to execute SIMULTANEOUSLY on the SAME core
- Weaving together multiple “threads” on the same core

- Example: if one thread is waiting for a floating point operation to complete, another thread can use the integer units

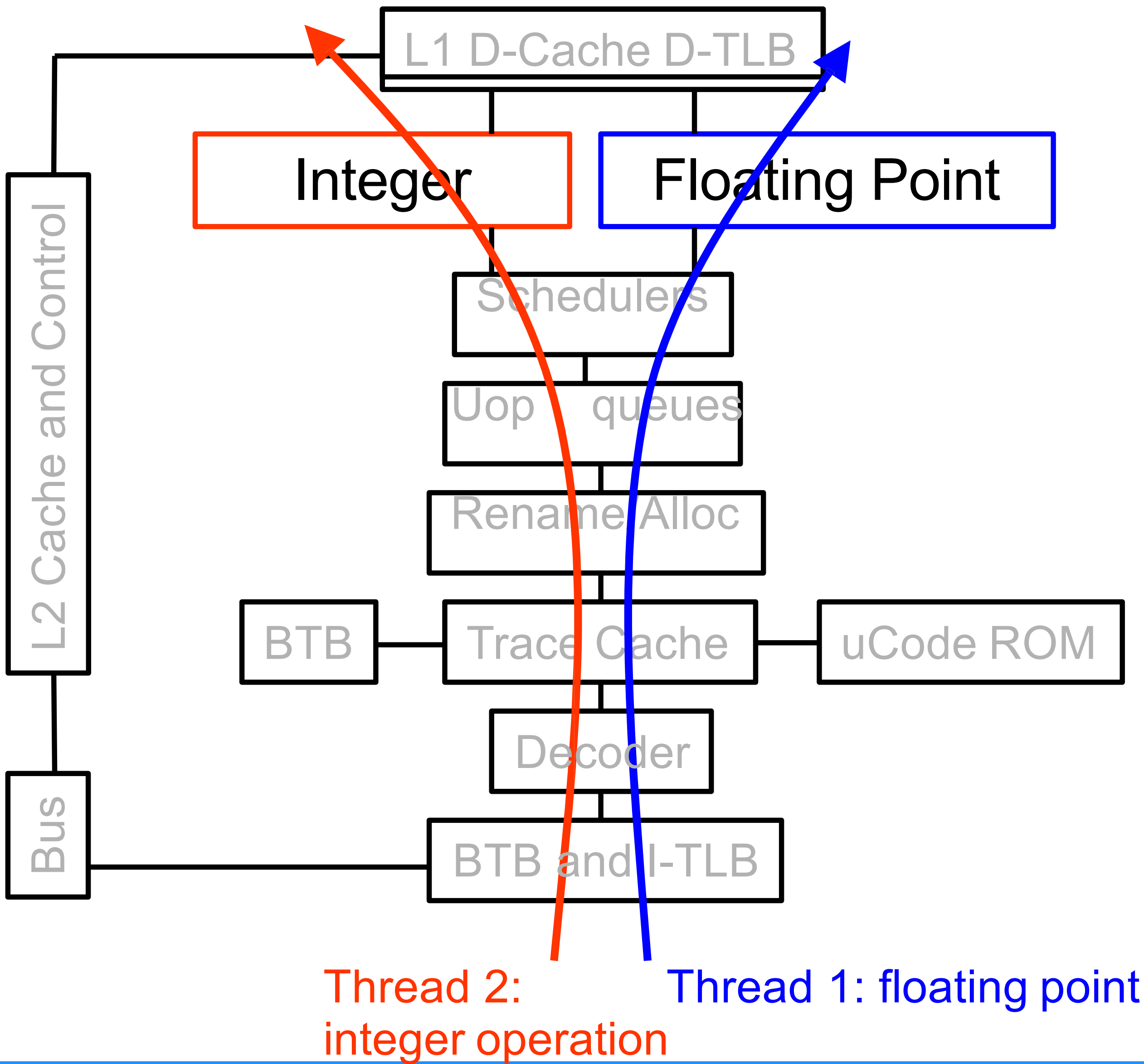
Without SMT, only a single thread can run at any given time



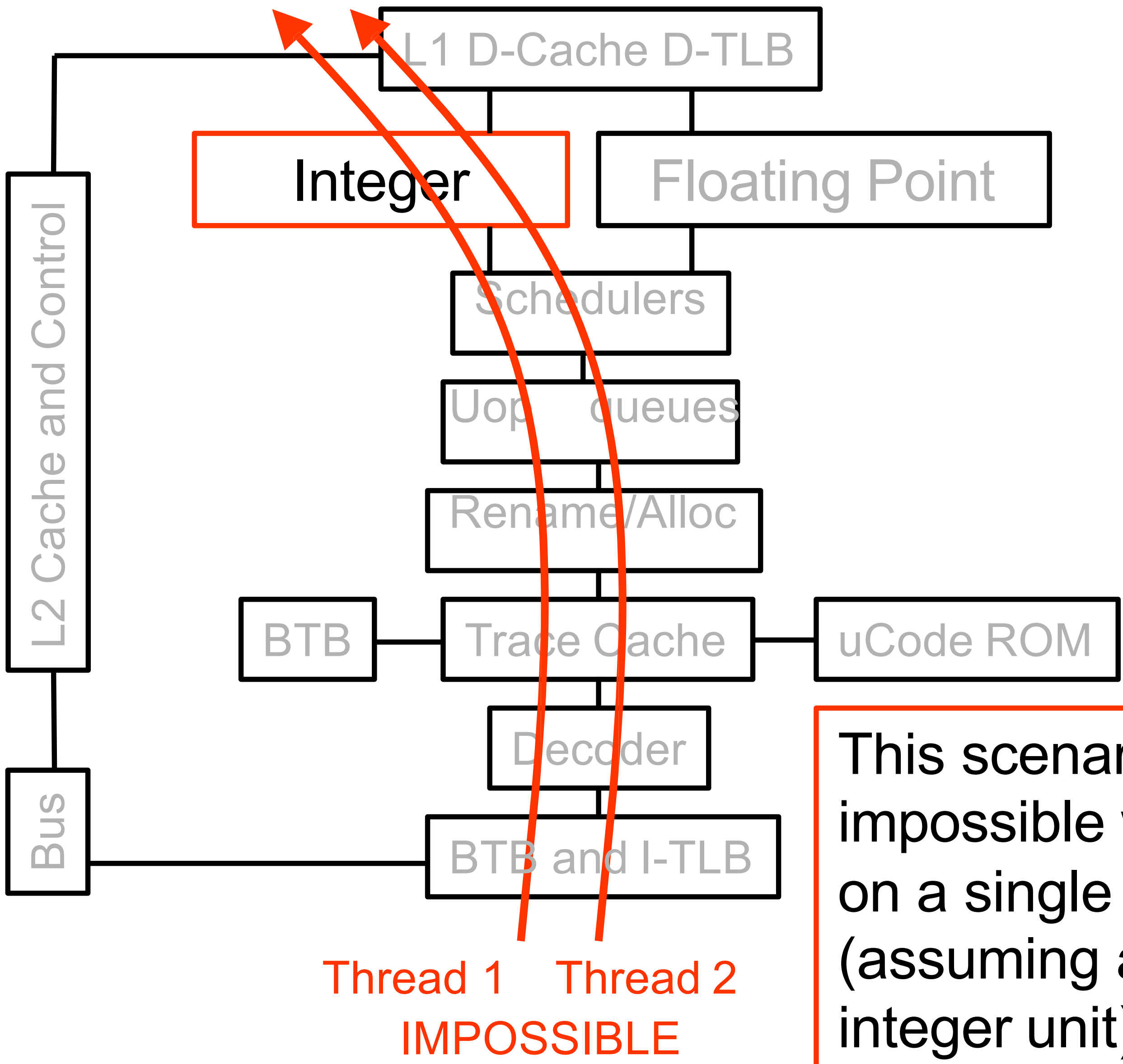
Without SMT, only a single thread can run at any given time



SMT processor: both threads can run concurrently



But: Can't simultaneously use the same functional unit

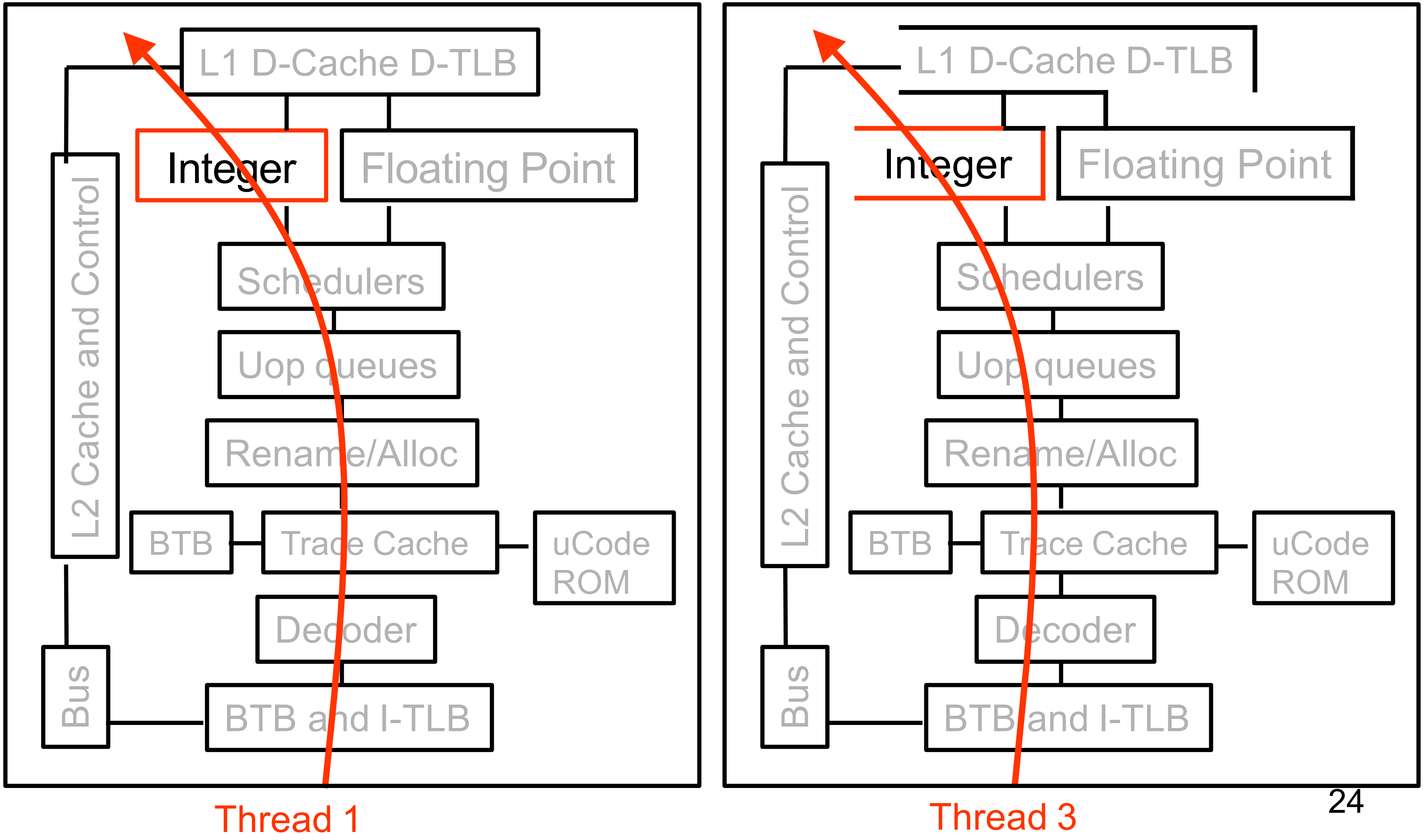


This scenario is impossible with SMT on a single core (assuming a single integer unit)

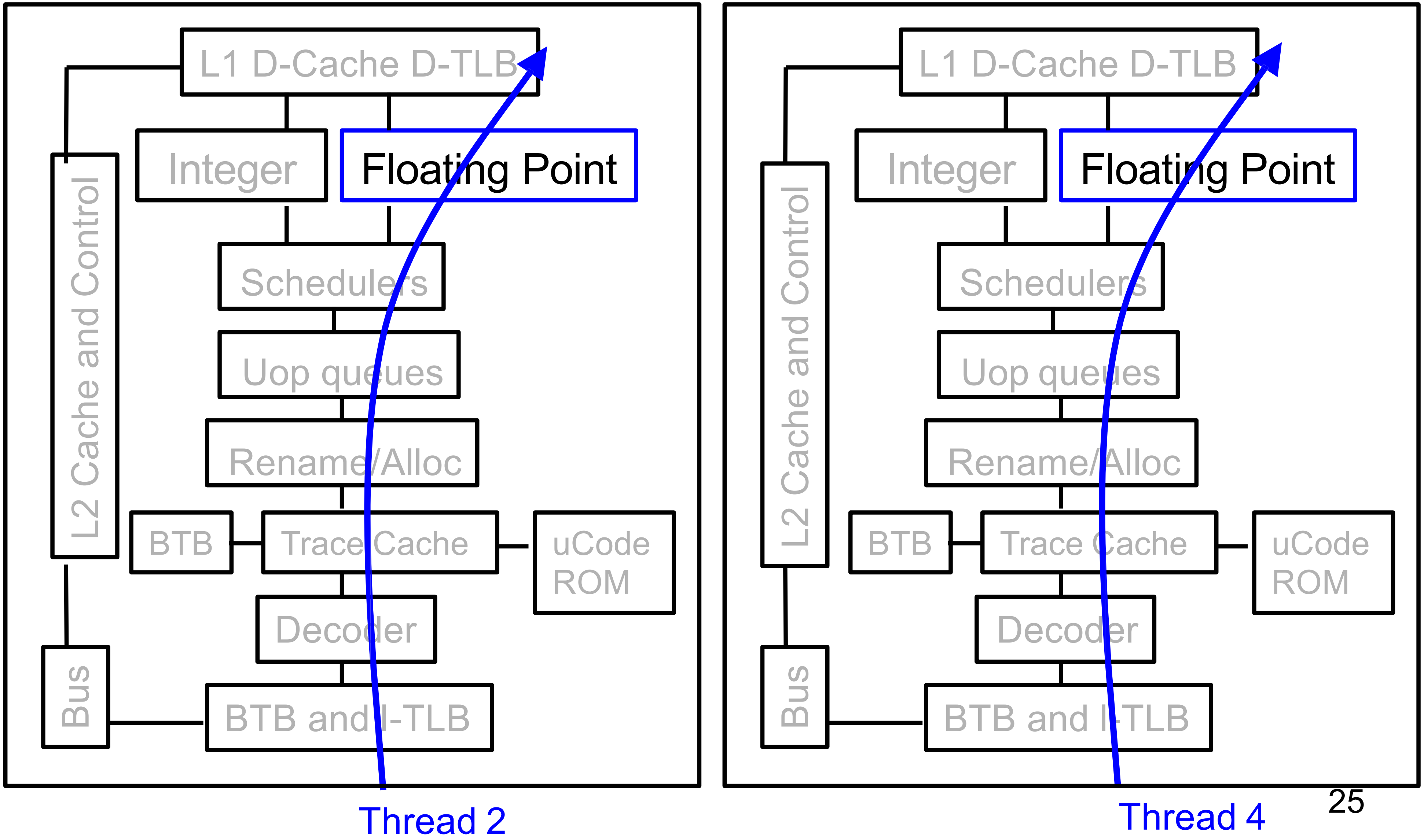
SMT not a “true” parallel processor

- Enables better threading (e.g. up to 30%)
- OS and applications perceive each simultaneous thread as a separate “virtual processor”
- The chip has only a single copy of each resource
- Compare to multi-core:
  each core has its own copy of resources

Multi-core:
threads can run on separate cores



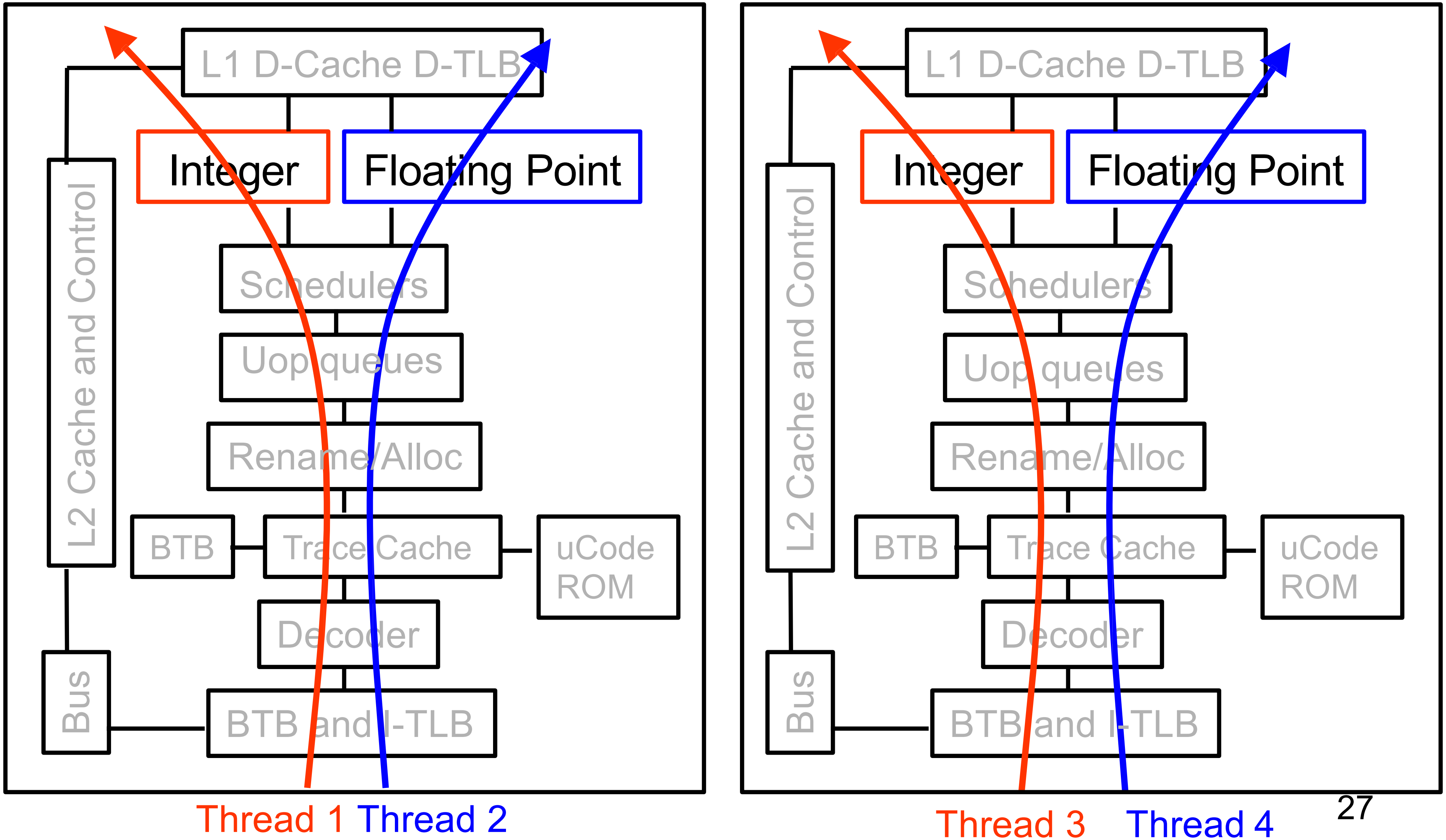
Multi-core: threads can run on separate cores



Combining Multi-core and SMT

- Cores can be SMT-enabled (or not)
- The different combinations:
  - Single-core, non-SMT: standard uniprocessor
  - Single-core, with SMT
  - Multi-core, non-SMT
  - Multi-core, with SMT: our fish machines
- The number of SMT threads:
  2, 4, or sometimes 8 simultaneous threads
- Intel calls them "hyper-threads"

SMT Dual-core: all four threads can run concurrently



Comparison: multi-core vs SMT

- Advantages/disadvantages?

Comparison: multi-core vs SMT

- Multi-core:
  - Since there are several cores,
    each is smaller and not as powerful
    (but also easier to design and manufacture)
  - However, great with thread-level parallelism
- SMT
  - Can have one large and fast superscalar core
  - Great performance on a single thread
  - Mostly still only exploits instruction-level parallelism

The memory hierarchy

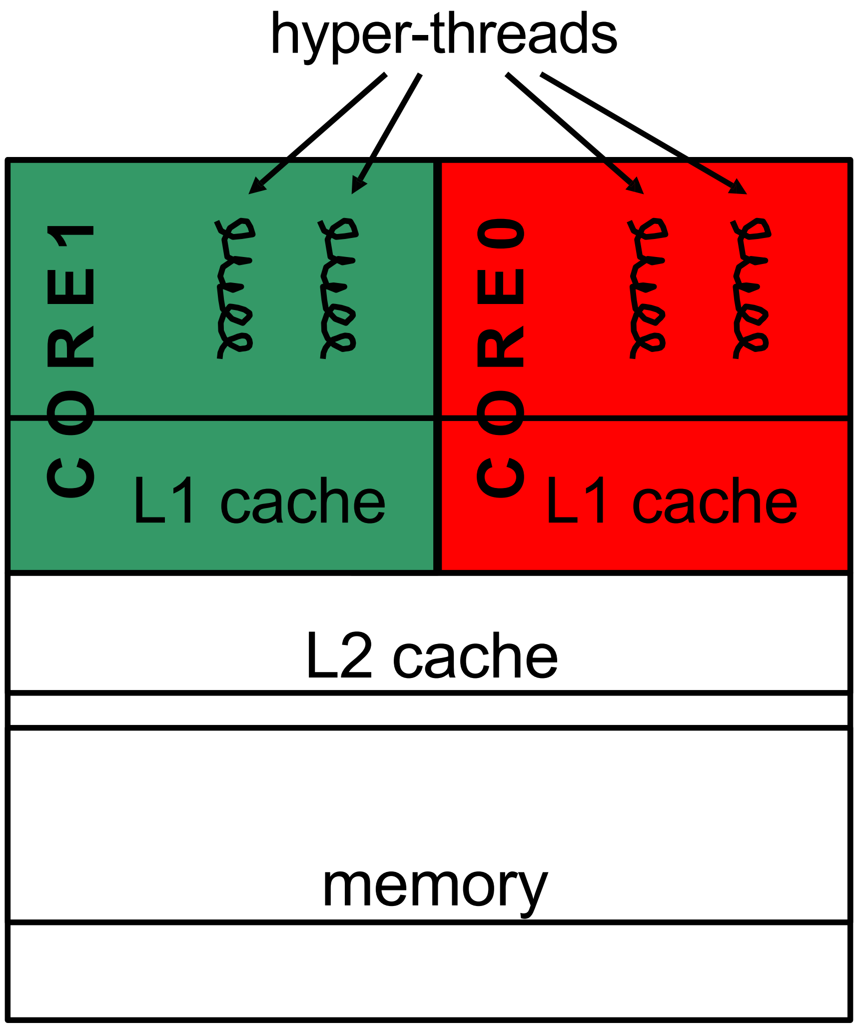
- If simultaneous multithreading only:
  – all caches shared
- Multi-core chips:
  – L1 caches private
  – L2 caches private in some architectures and shared in others
- Memory is always shared

Example

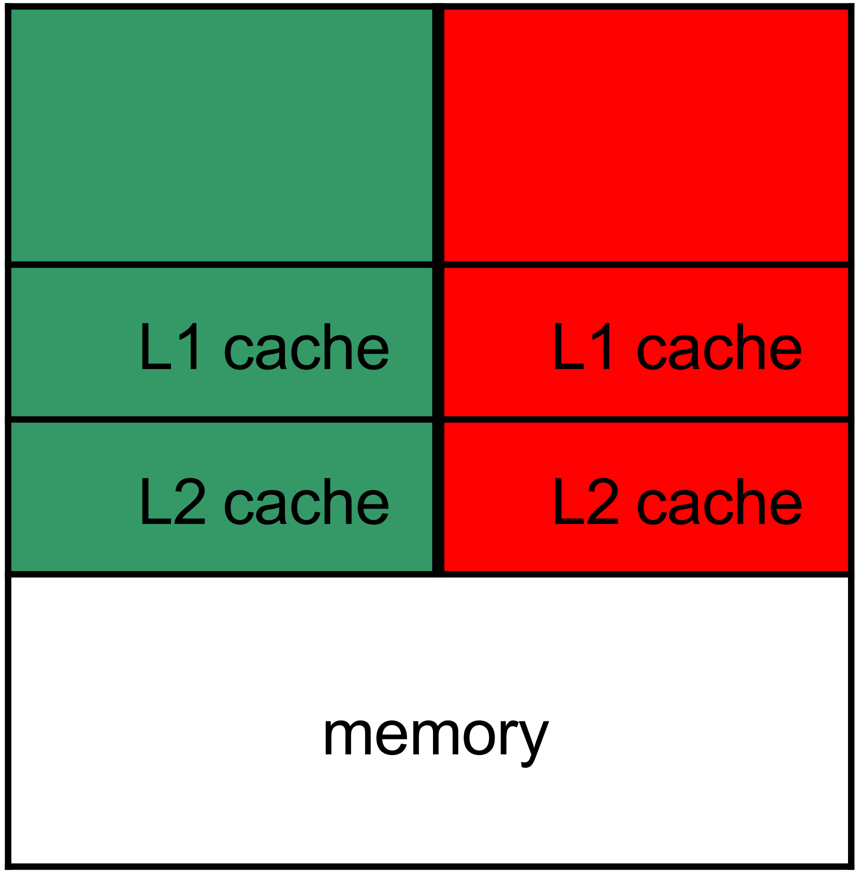
- Dual-core
  Intel Xeon processors

- Each core is
  hyper-threaded

- Private L1 caches
- Shared L2 caches

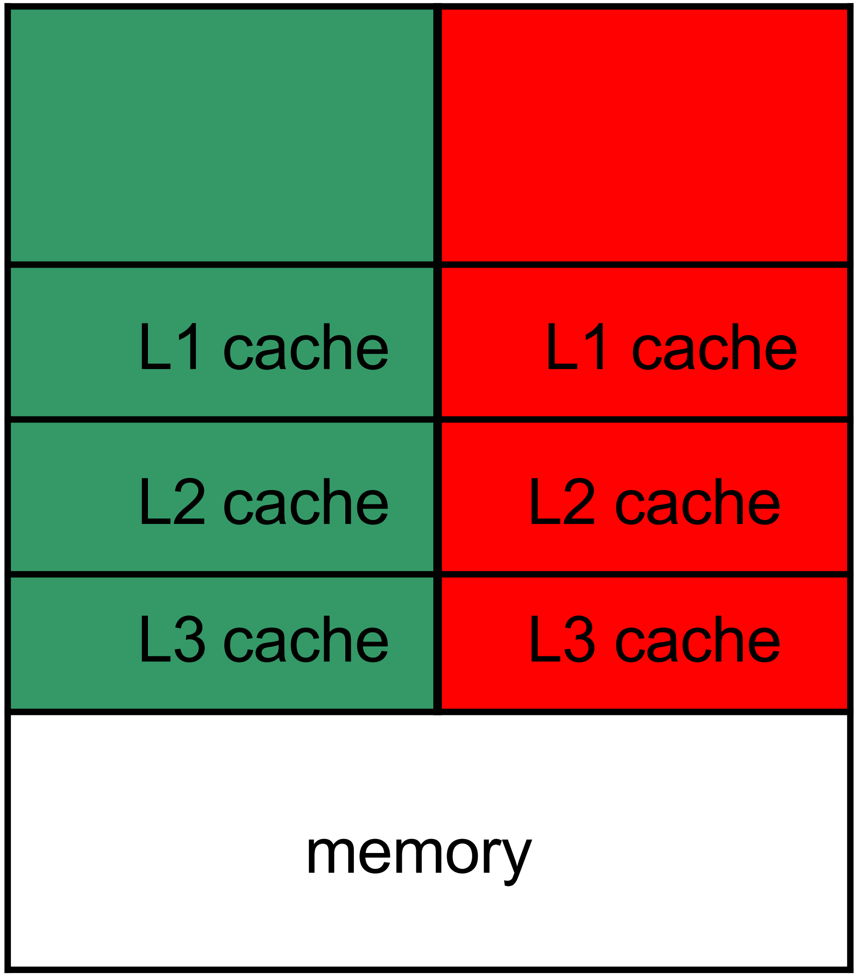


Designs with private L2 caches



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?

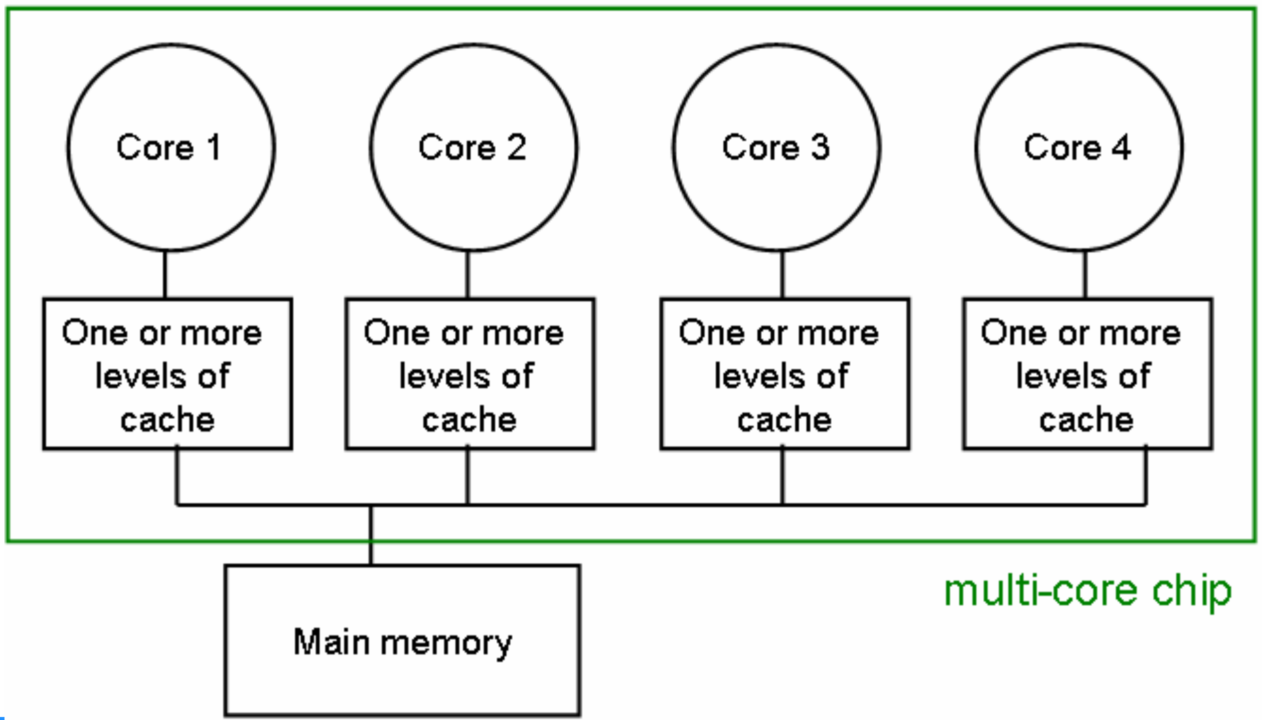
- Advantages/disadvantages?

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

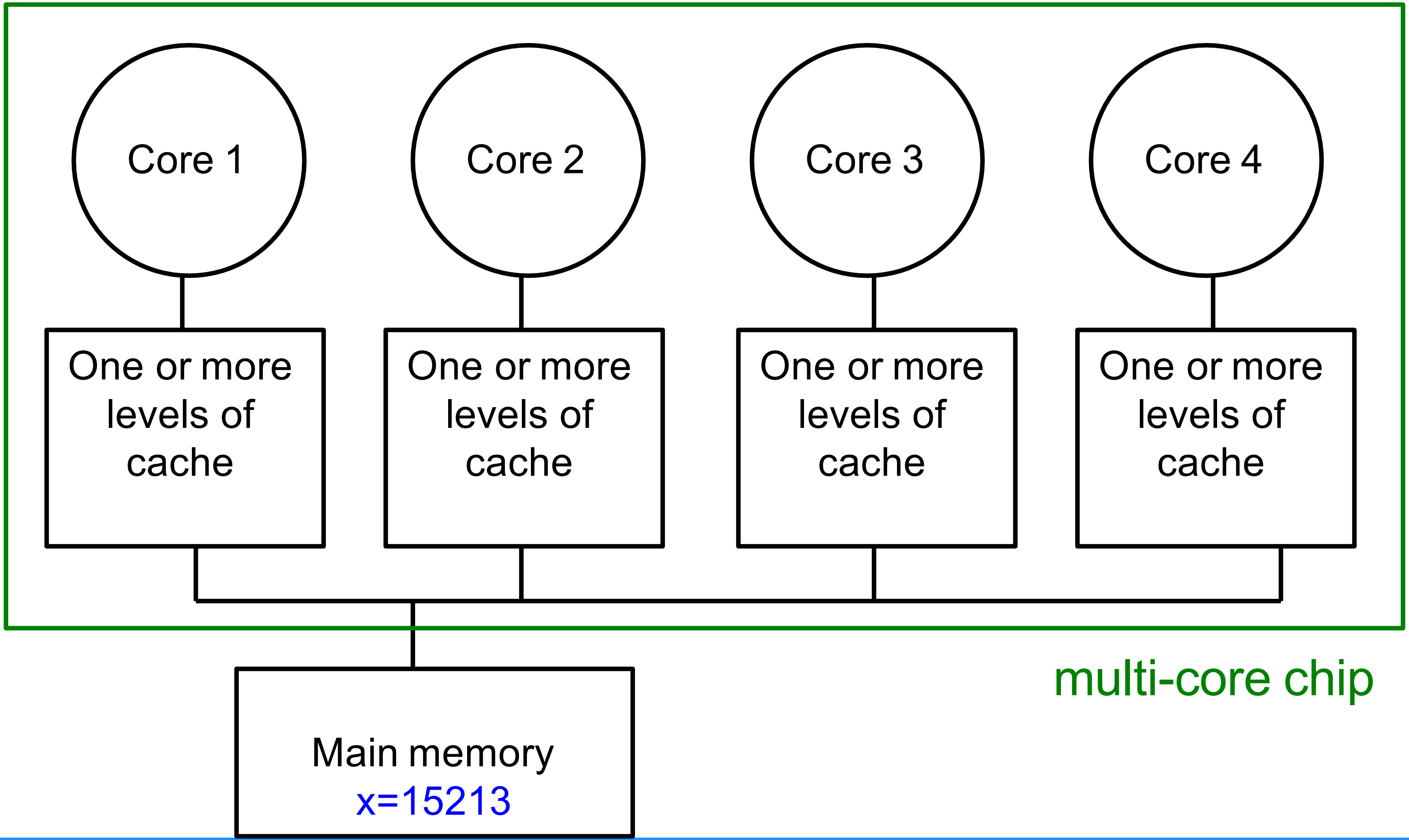
The cache coherence problem

- Since we have 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



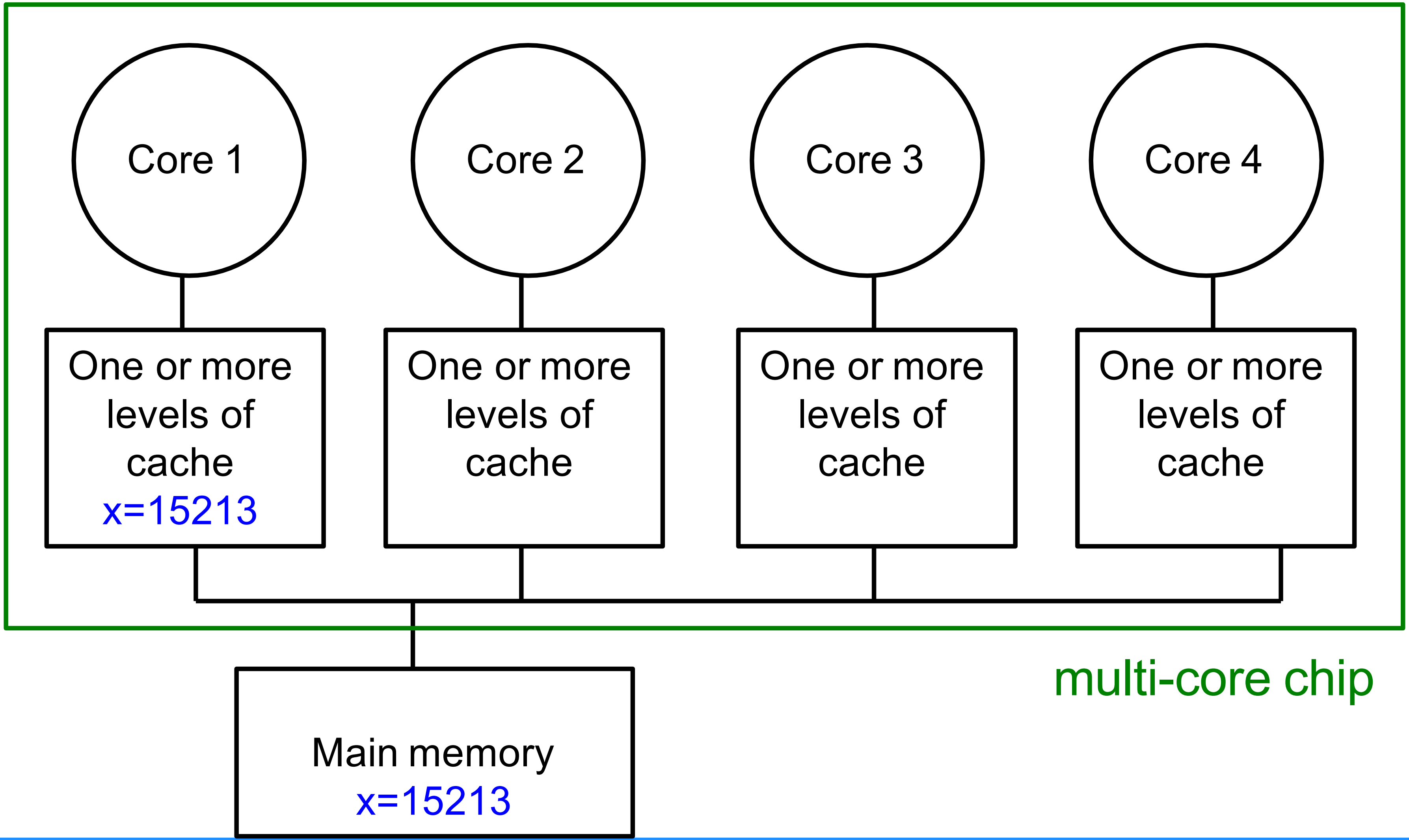
The cache coherence problem

Suppose variable x initially contains 15213



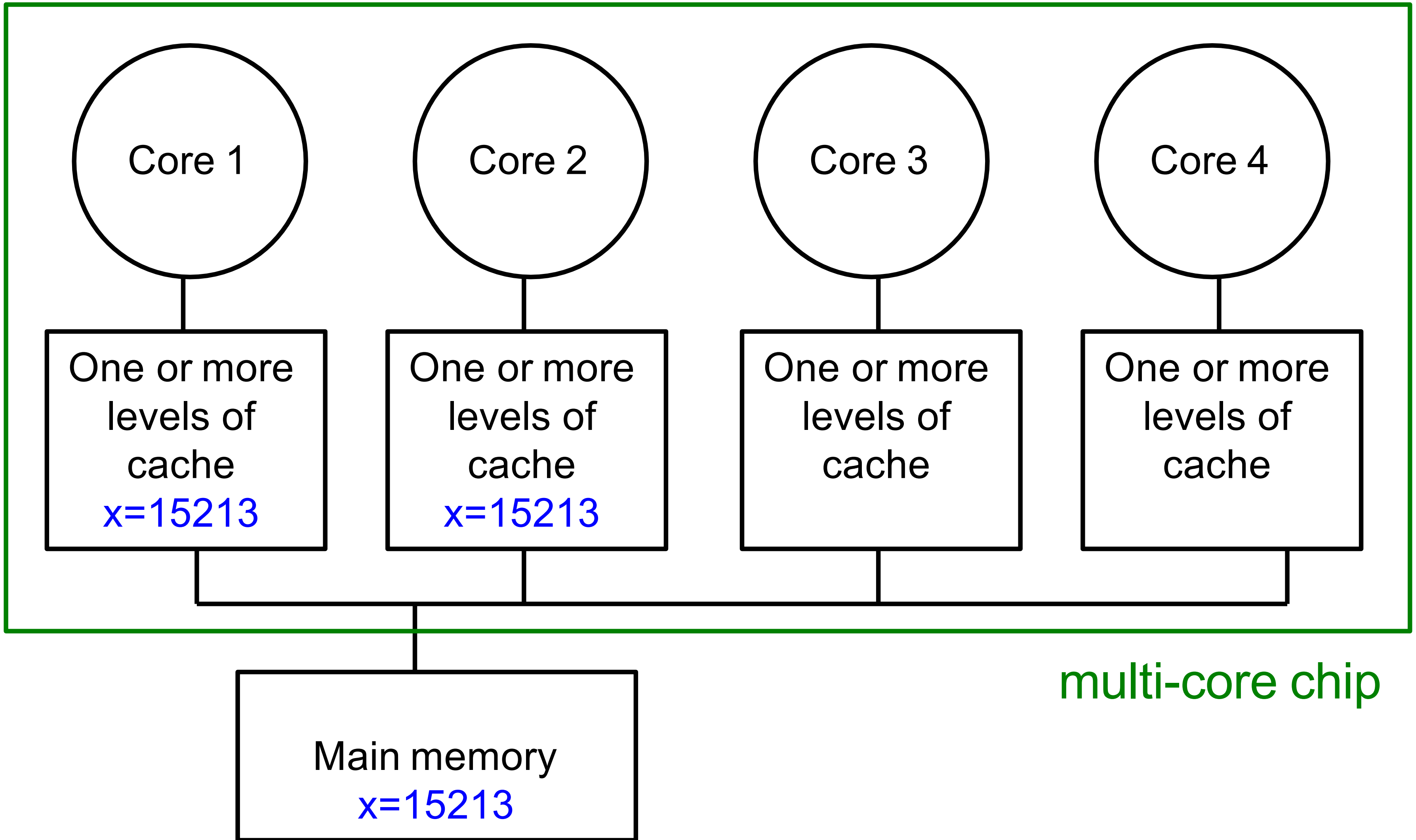
The cache coherence problem

Core 1 reads x



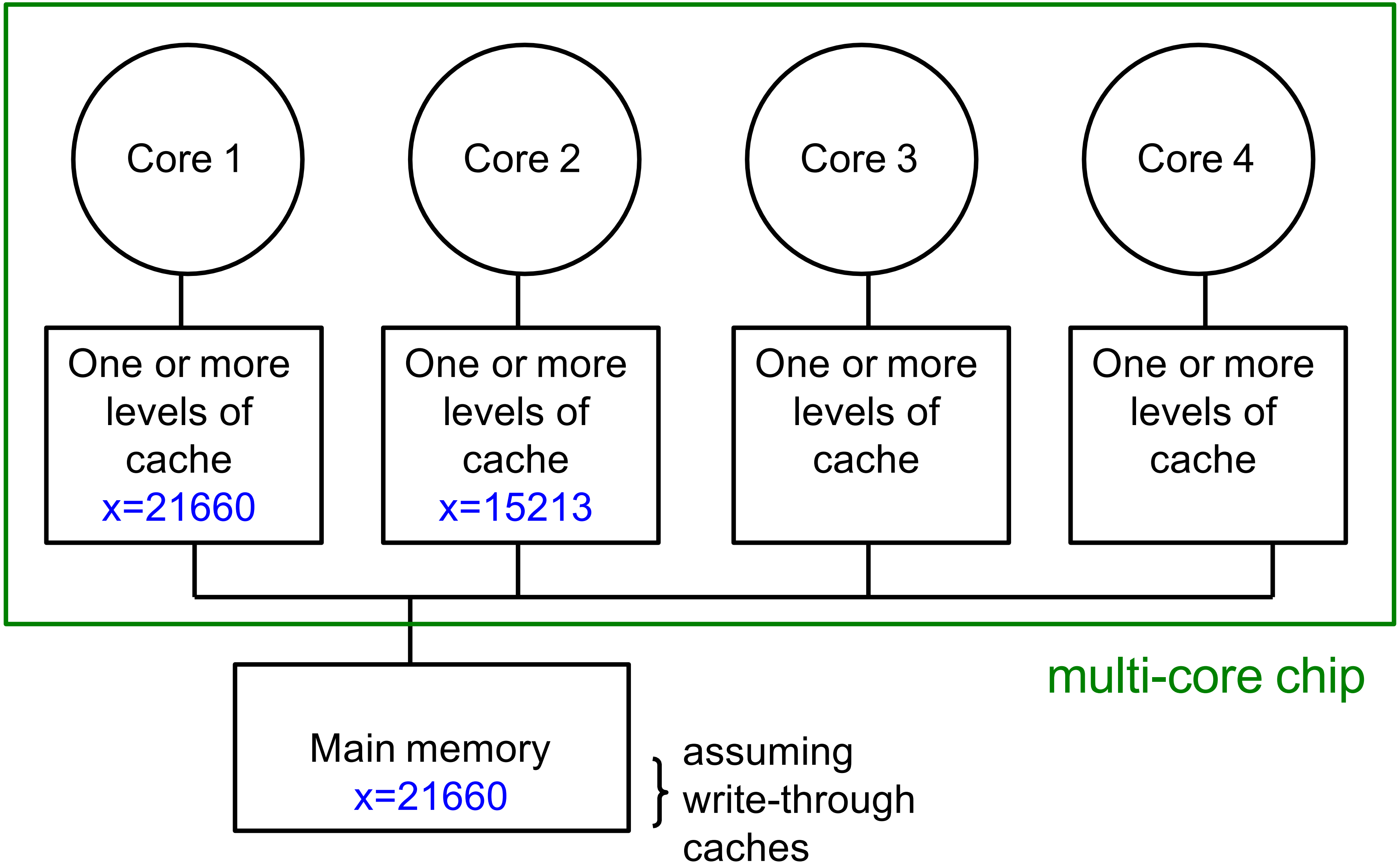
The cache coherence problem

Core 2 reads x



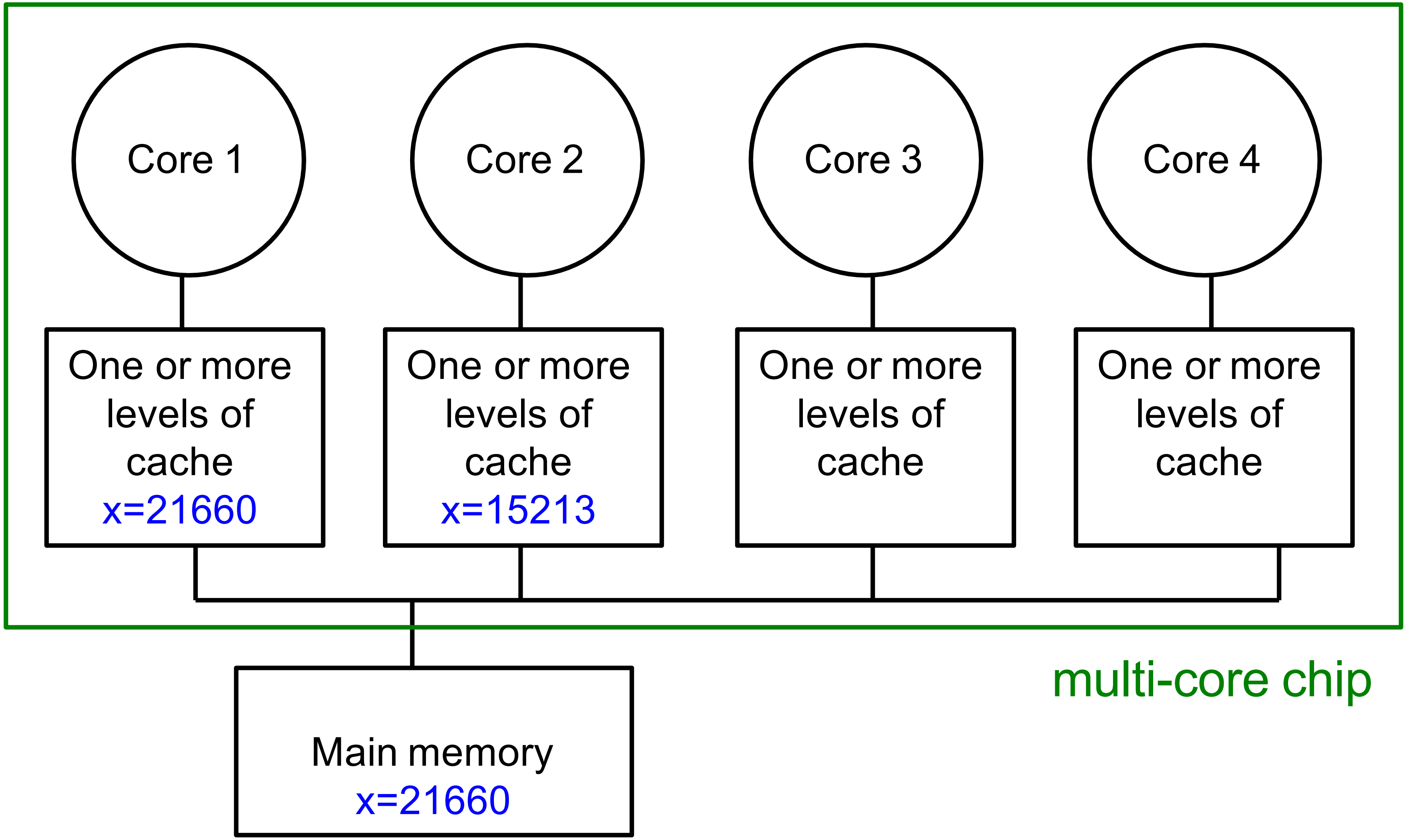
The cache coherence problem

Core 1 writes to x, setting it to 21660



The cache coherence problem

Core 2 attempts to read x… gets a stale copy

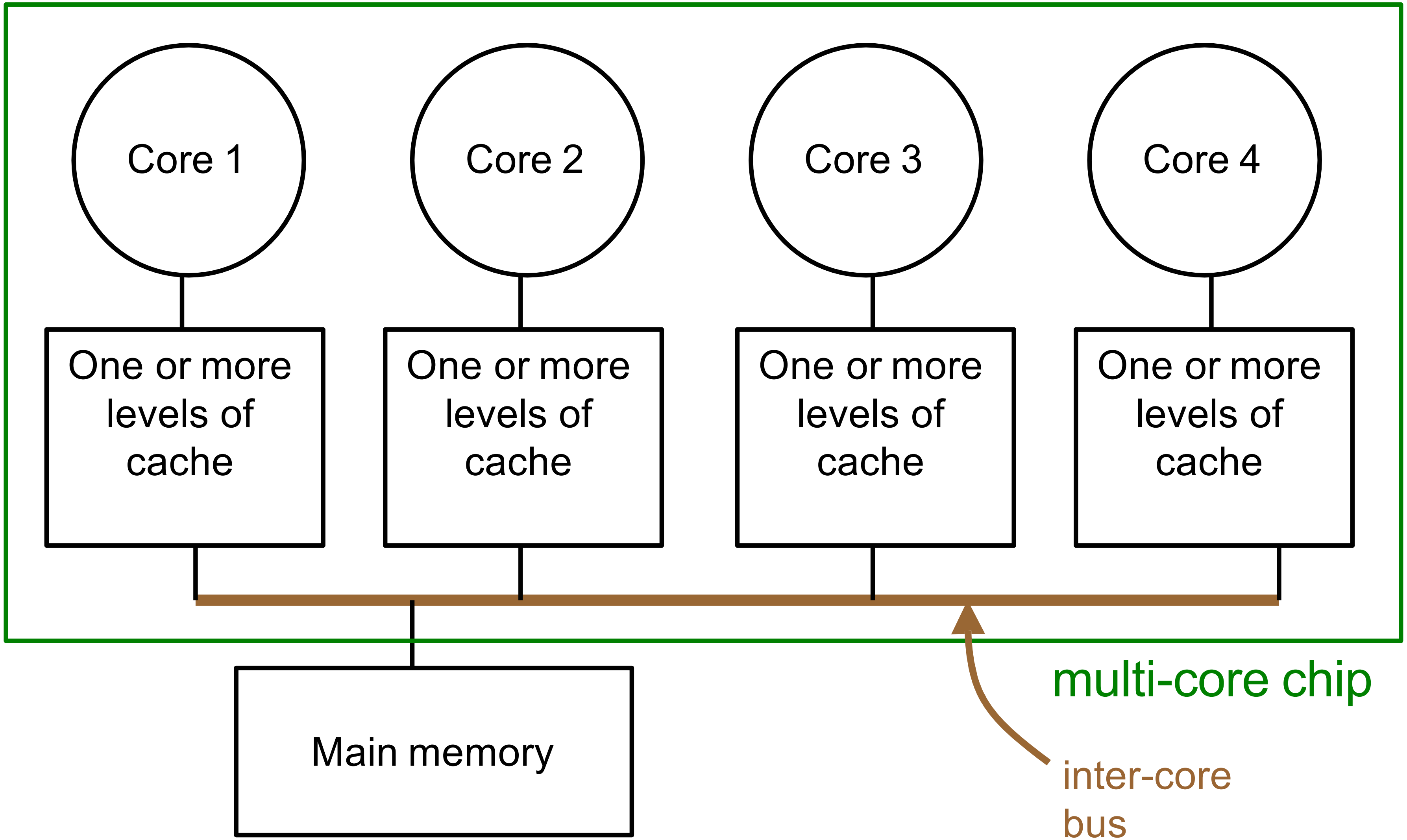


Solutions for cache coherence

- 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

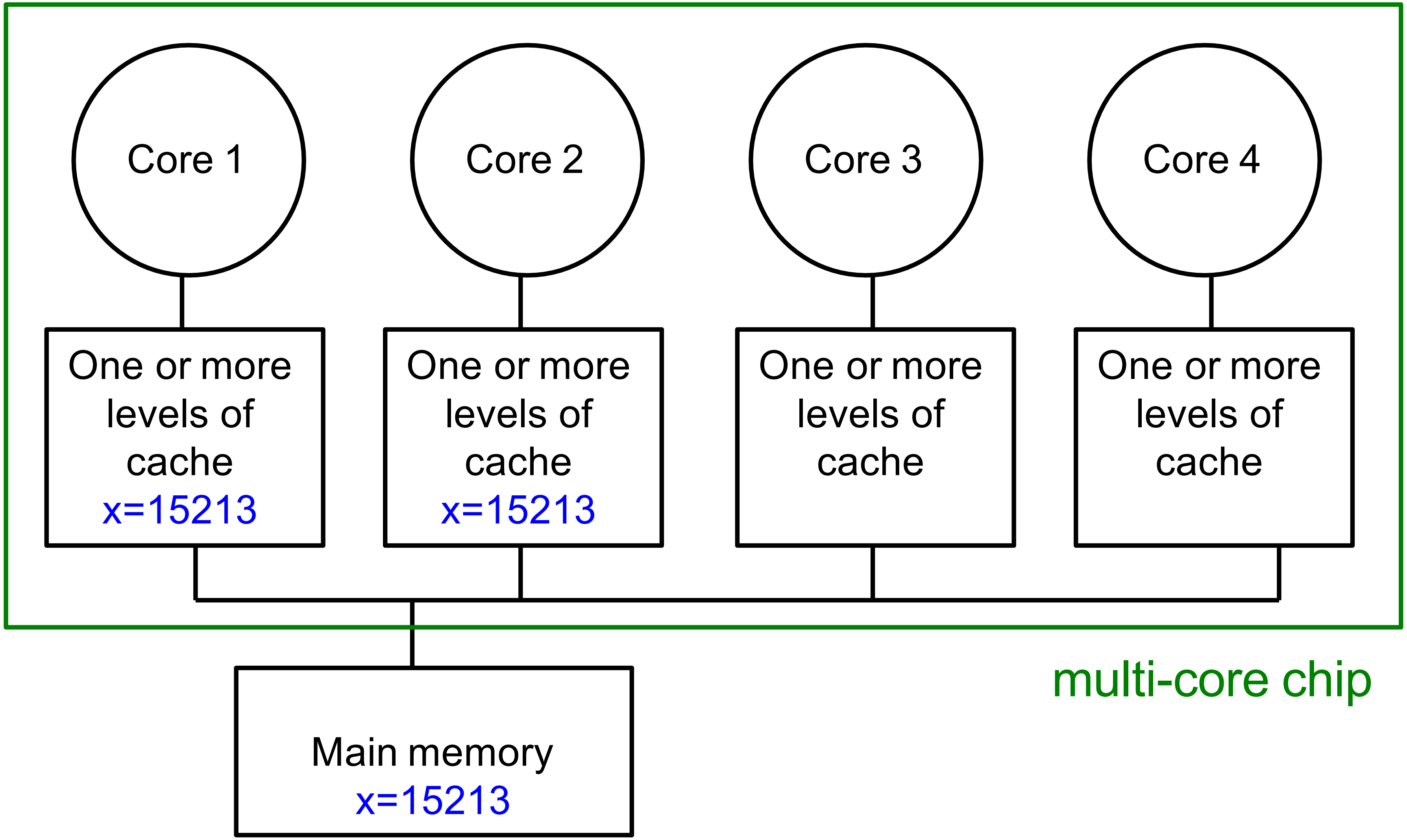


Invalidation protocol with snooping

- 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.

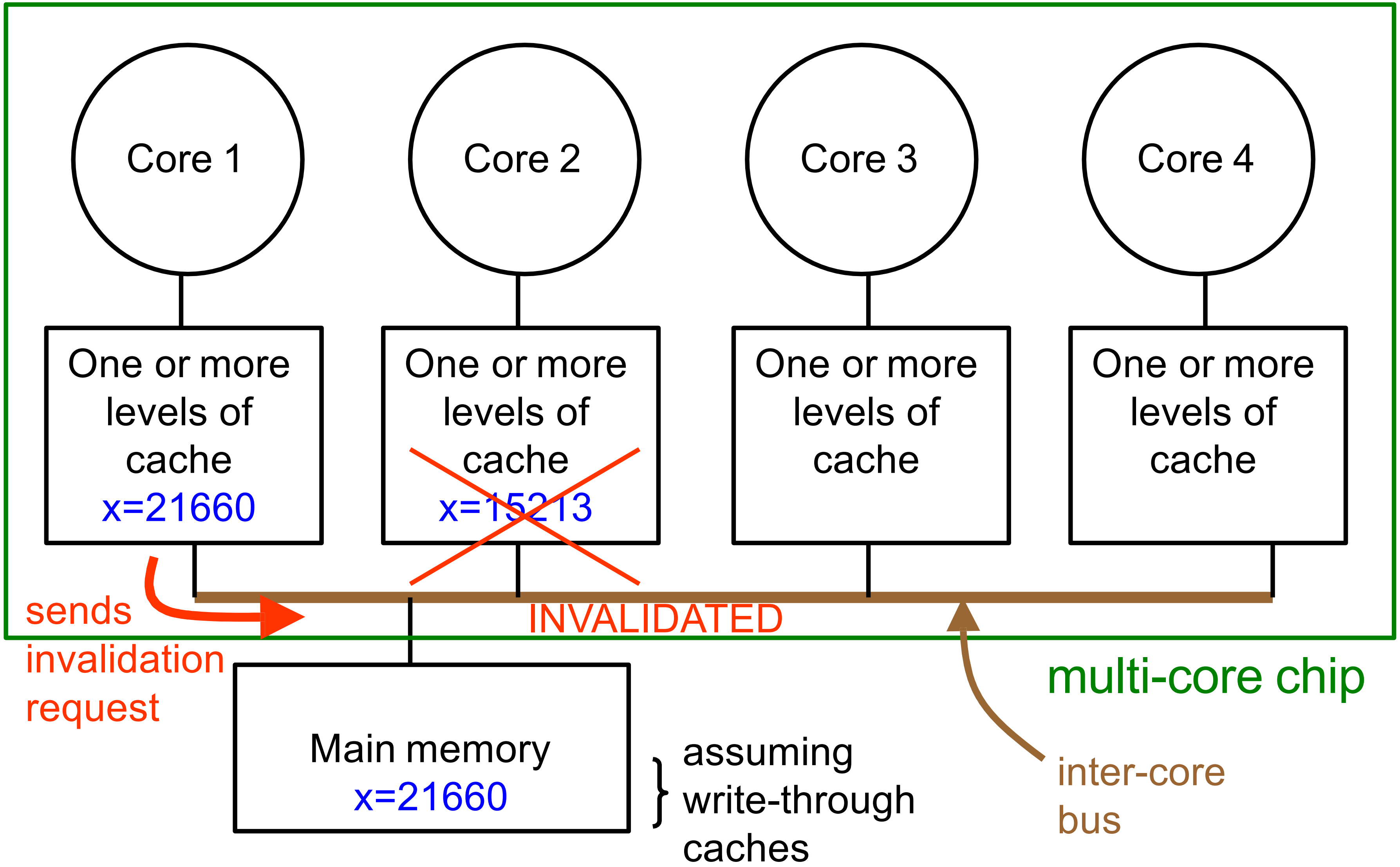
The cache coherence problem

Revisited: Cores 1 and 2 have both read x



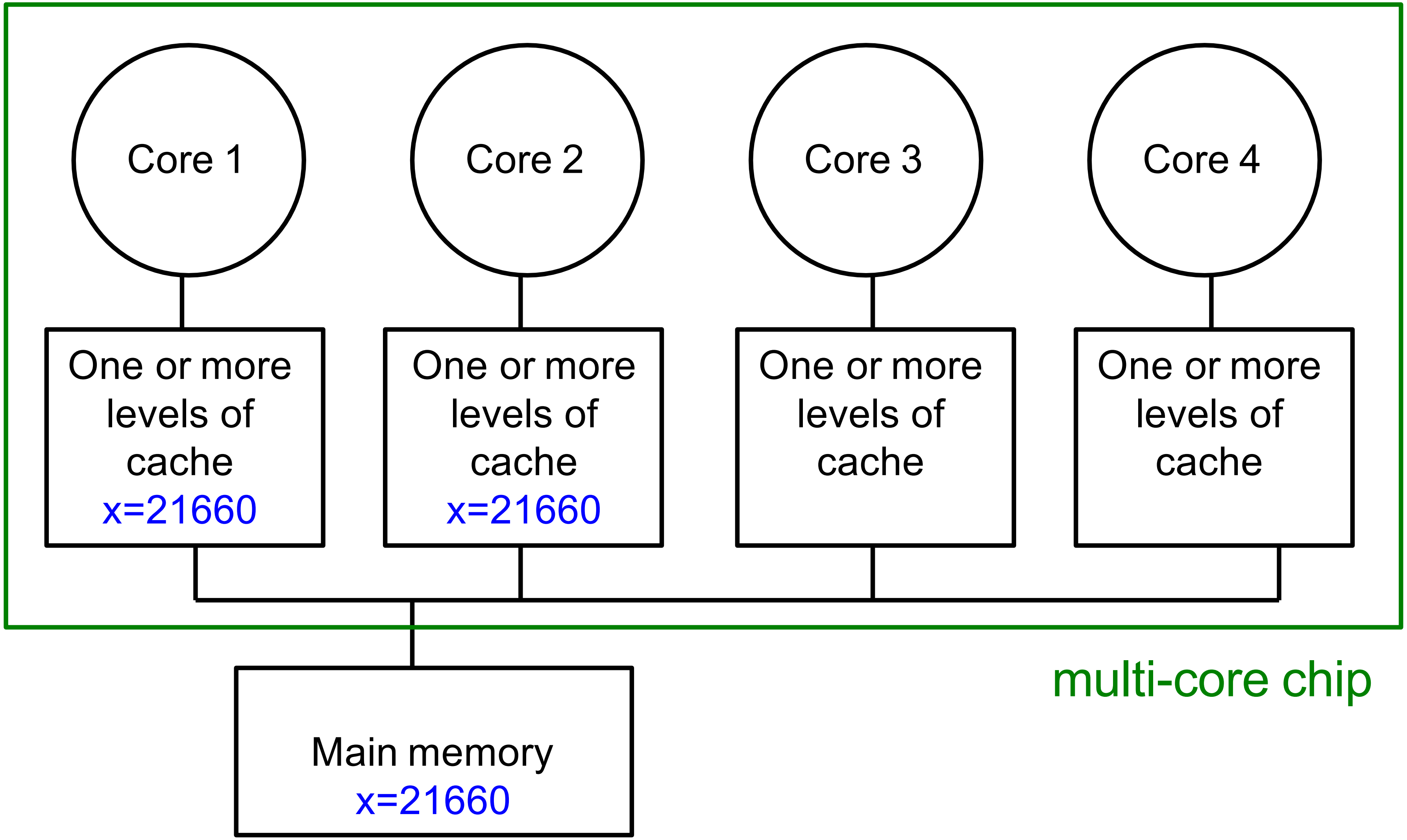
The cache coherence problem

Core 1 writes to x, setting it to 21660



The cache coherence problem

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



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

However: Need to use synchronization even if only time-slicing on a uniprocessor

```
int counter=0;

void thread1() {
 int temp1=counter;
 counter = temp1 + 1;
}

void thread2() {
 int temp2=counter;
 counter = temp2 + 1;
}
```

Need to use synchronization even if only time-slicing on a uniprocessor

```
temp1=counter;
counter = temp1 + 1;
temp2=counter;
counter = temp2 + 1
```

gives counter=2

```
temp1=counter;
temp2=counter;
counter = temp1 + 1;
counter = temp2 + 1
```

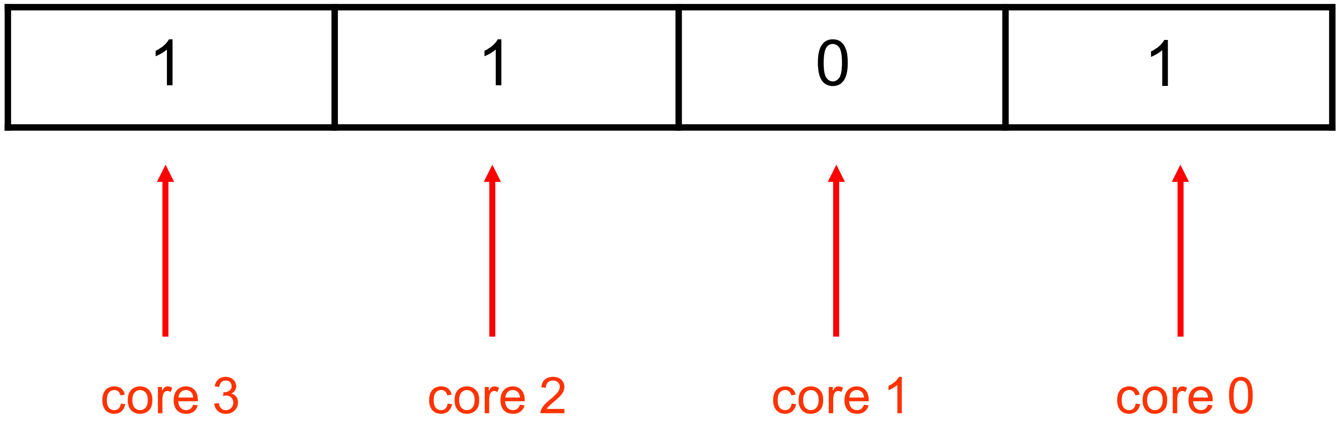
gives counter=1 or 2

Assigning threads to the cores

- Each thread 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

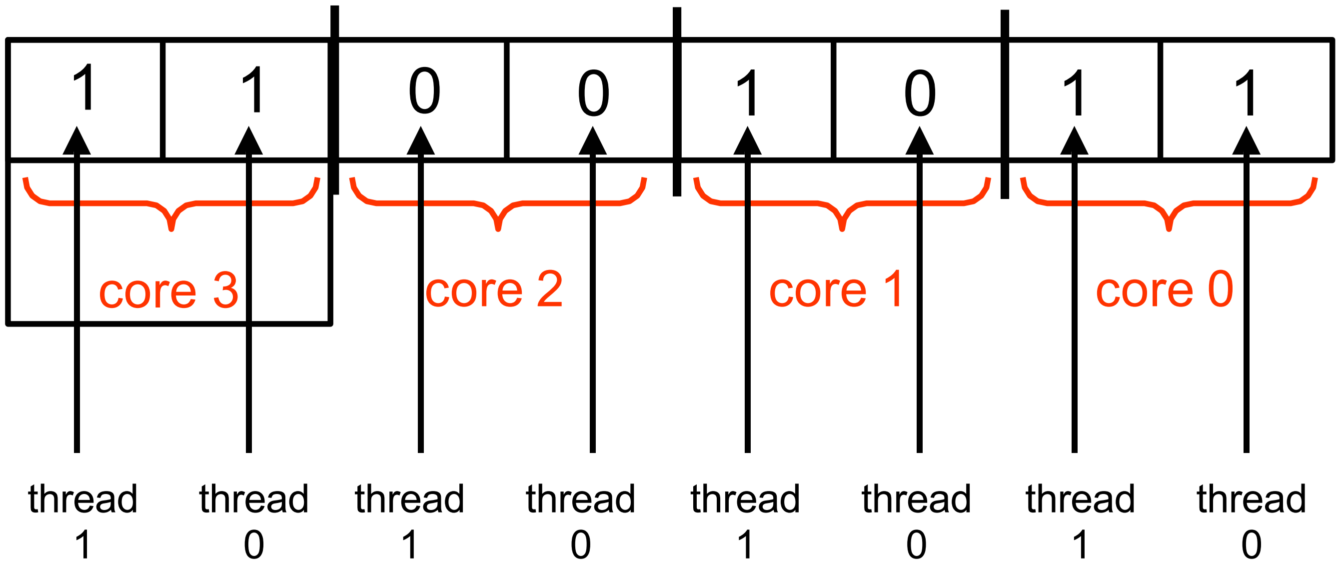
- Example: 4-way multi-core, without SMT



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

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

When to set your own affinities

- Two (or more) threads share data-structures in memory
  – map to same core so that can share cache
- Real-time threads:
  Example: a thread running
  a robot controller:
  - must not be context switched,
    or else robot can go unstable
  - dedicate an entire core just to this thread

Source: Sensable.com

Kernel scheduler API

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

Retrieves the current affinity mask of process ‘pid’ and stores it into space pointed to by ‘mask’.

‘len’ is the system word size: sizeof(unsigned int long)

Kernel scheduler API

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

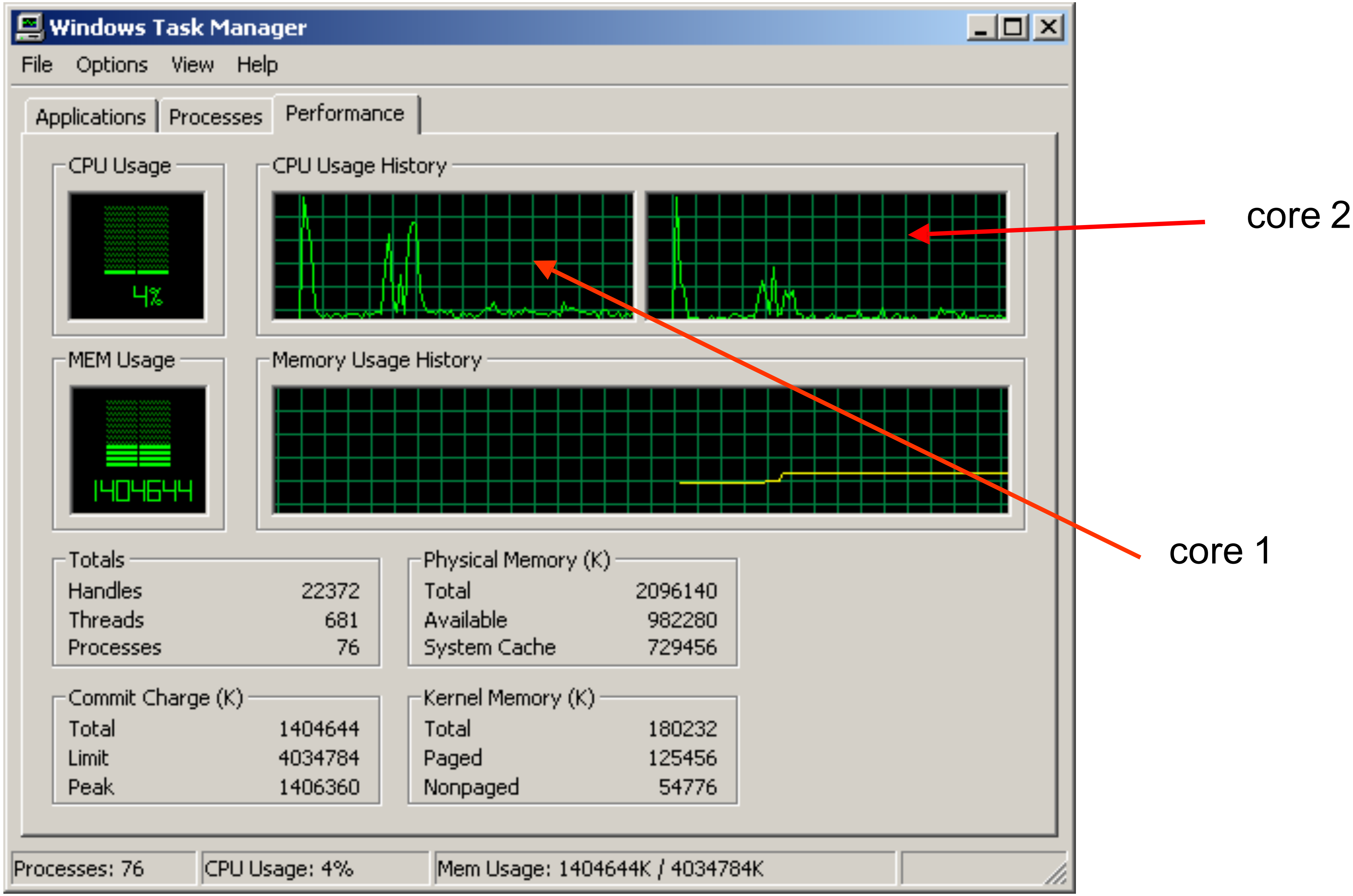
Sets the current affinity mask of process ‘pid’ to *mask

‘len’ is the system word size: sizeof(unsigned int long)

To query affinity of a running process:

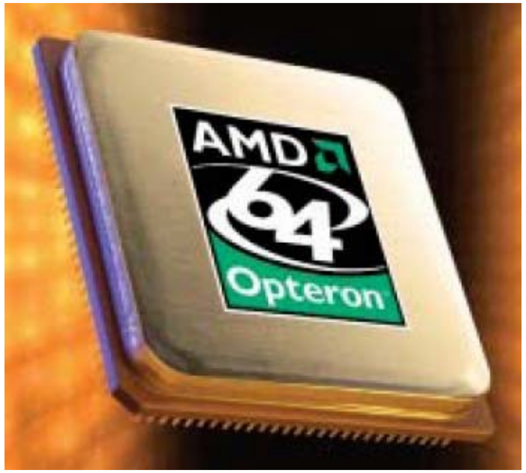
```
[~]$ taskset -p 3935  pid 3935's
current affinity mask: f
```

Windows Task Manager



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

Further readings

A. S. Tanenbaum, T. Austin, and B. R. Chandavarkar, *Structured computer organization*, 6. ed ., international ed. Boston, Mass.: Pearson, 2013 – Chapter 8

On-Chip Paralellism

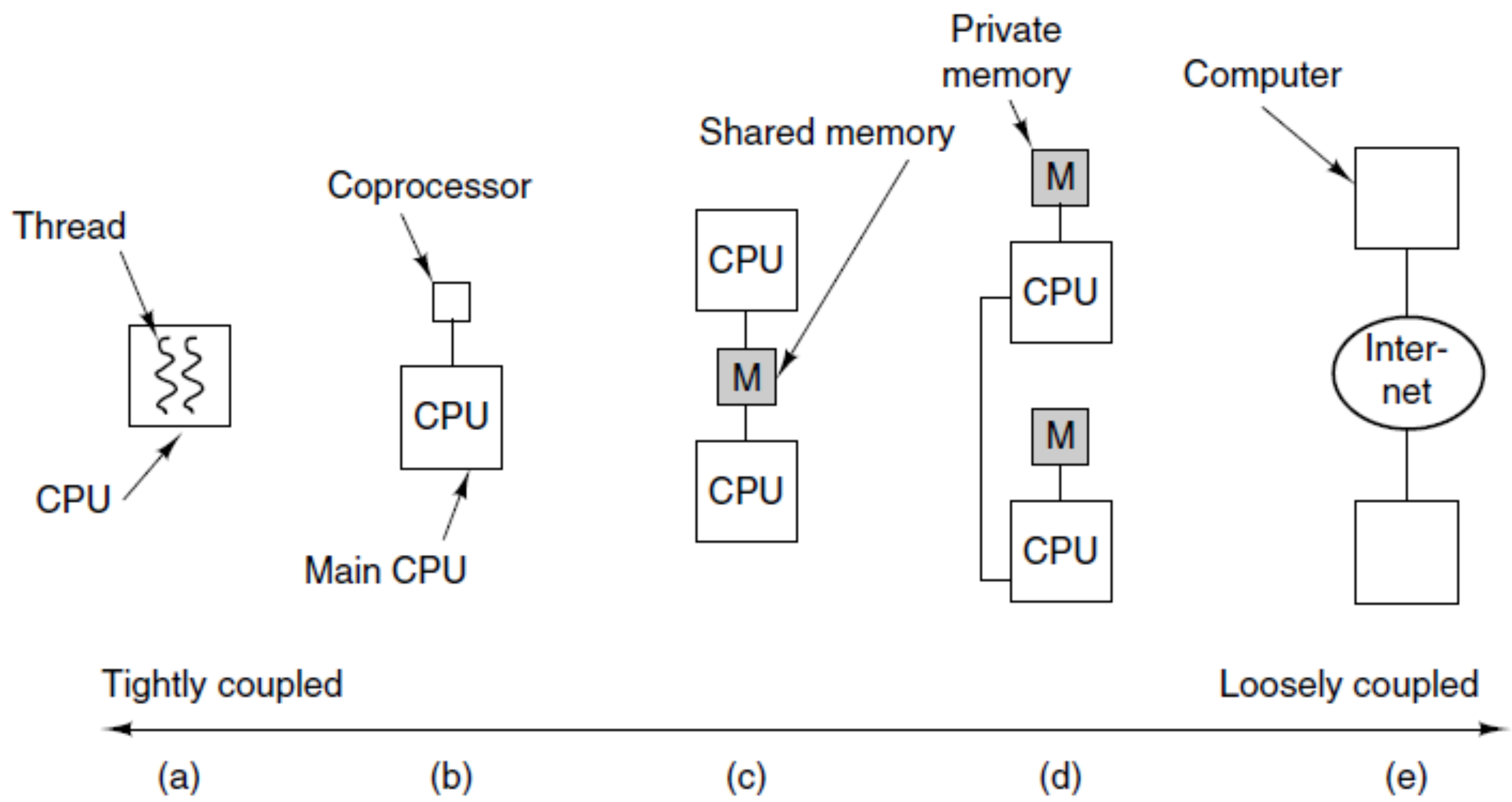


Figure 8-1. (a) On-chip parallelism. (b) A coprocessor. (c) A multiprocessor. (d) A multicomputer. (e) A grid.

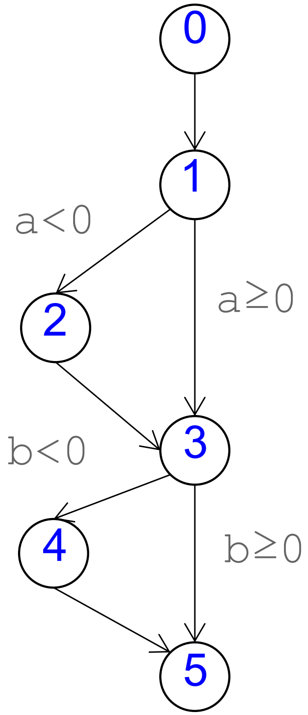
Programming

C/C++ Threads – short recap

Sequential program

– A sequential program can be presented as a control flow graph

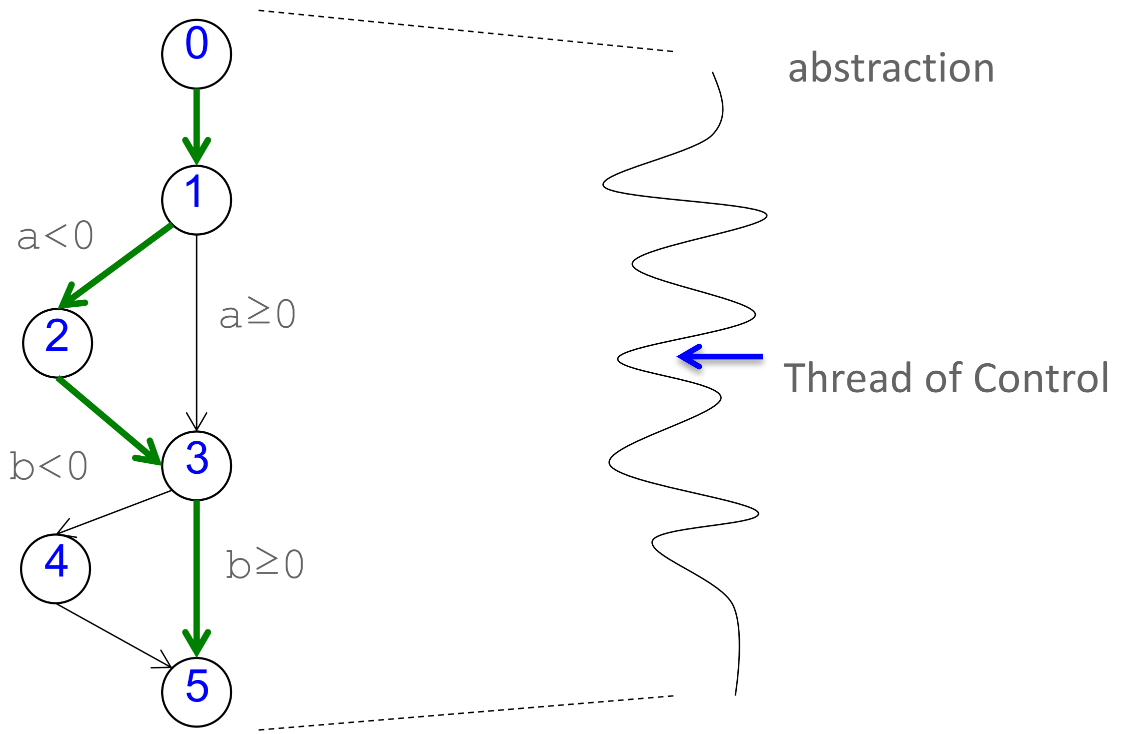
```
0:      public static int manhattan(int a, int b){
1:         if (a < 0)
2:            a = -a;
3:         if (b < 0)
4:            b = -b;
5:         return a+b;
        }
```



Thread

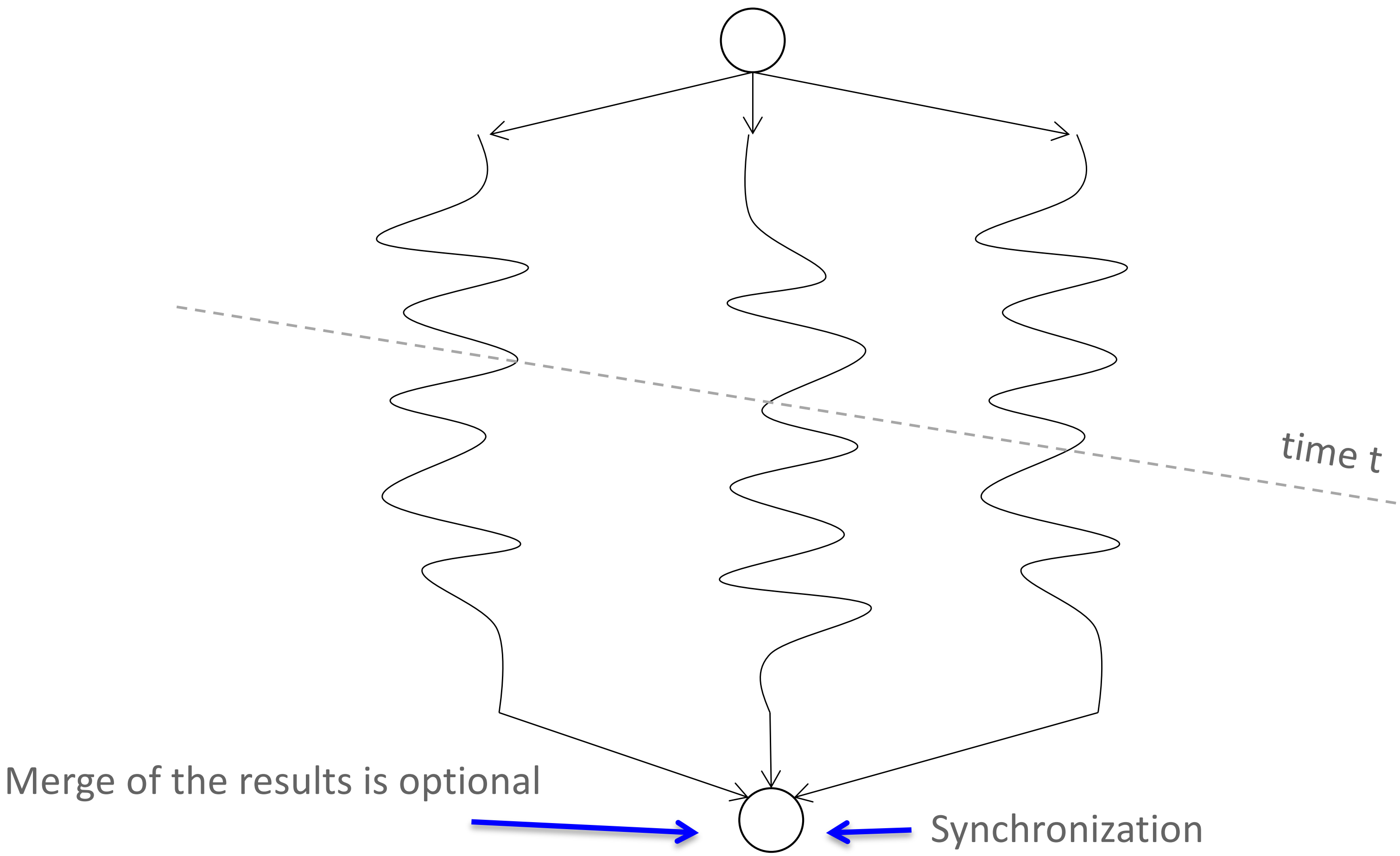
- For a specific input
  - a path of the control graph is run through

`manhattan(-1,1)`



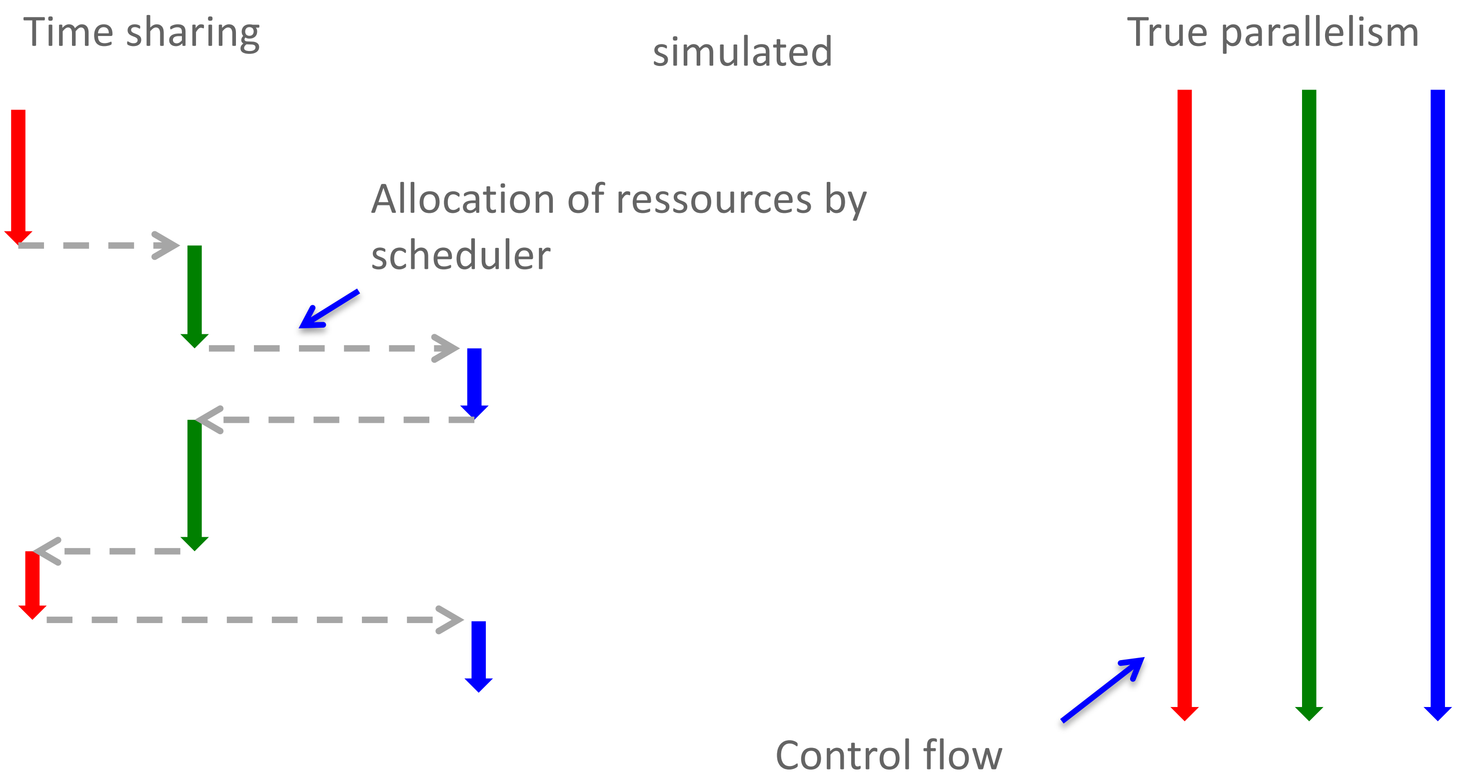
Concurrent

- Program (process) are called concurrent if they do not depend on each other
  - The associated control flows can be run through in parallel



Concurrent

- Concurrent control flows can be processed in parallel if the required resources (CPU, printer, ...) are available for each control flow.
- Otherwise, the resources are temporarily allocated to the control flow by a scheduler.



Concurrent

Typical applications for concurrent programming

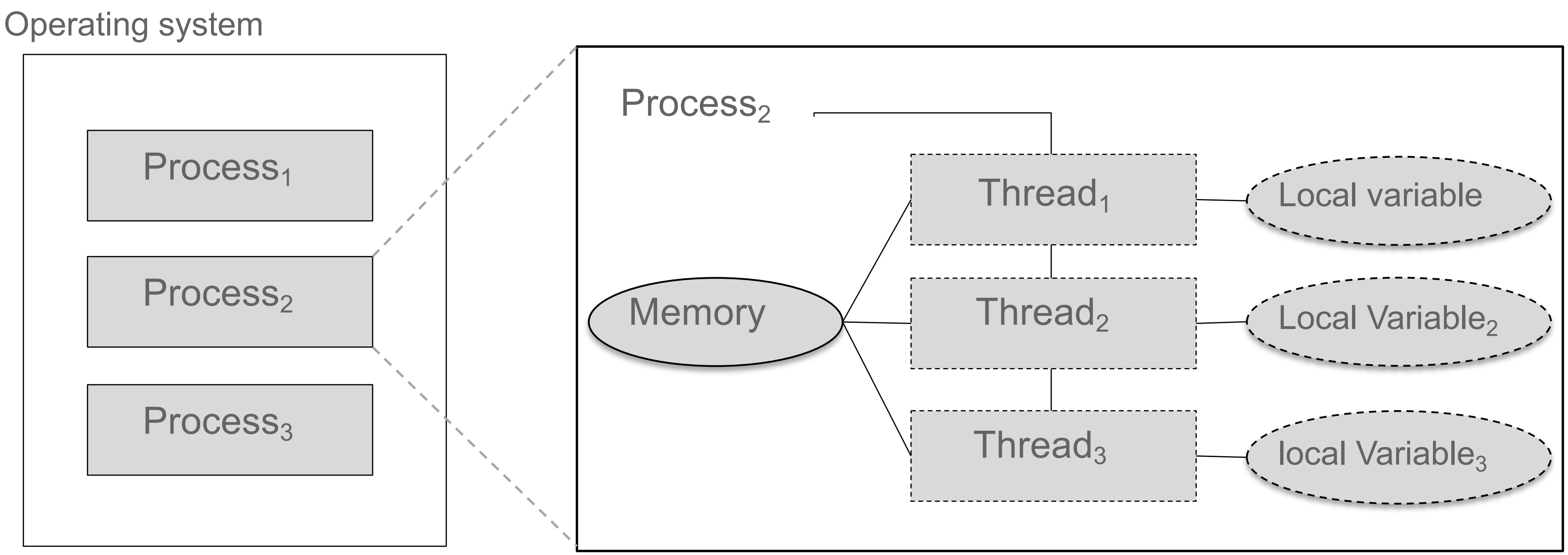
- Concurrency is given by the task in a natural way (e.g. motor control)
- Better utilization of resources (hardware)
  - Modern processors typically have multiple CPU cores
- Increasing the throughput of a system
  - Time-consuming I/O request blocks only one control flow and not the entire application
- Avoidance of blockades
  - Improvement of the interactivity of user interfaces
  - Reduction of the influence of blocking interfaces (e.g. network connections)

Processes vs. threads

- A process is an executed program
  - A process receives resources from the operating system for a certain period of time.
    - Processor
    - Own main memory area (address space)

- A thread is a lightweight process
  - Belongs to a process
  - Shares address space (data) and resources with other threads of the same process
  - Has its own register set and stack (e.g. for local variables)
  - can be stopped and continued by a scheduler
  - can wait for an event without CPU load

Processes vs. threads



Threads

- Threads typically require less administrative effort
  - The scheduler can switch between threads faster
  - Data exchange between threads is accelerated by the common address space.
    - but also holds dangers...
- Application area of threads: subtasks of a program should be executed in parallel
  - GUI
  - technical logic
  - DB and network accesses

Multithreaded communication

- Serial execution:
  - All our programs so far has had a single thread of execution: main thread.
  - Program exits when the main thread exits.
- Multithreaded:
  - Program is organized as multiple and concurrent threads of execution.
  - The main thread *spawns* multiple threads.
  - The threads **may** communicate with one another.
  - Advantages:
    - Improves performance
    - Improves responsiveness
    - Improves utilization
    - less overhead compared to multiple processes

Multithreaded programming

- Even in C, multithread programming may be accomplished in several ways
  - Pthreads: POSIX C library.
  - OpenMP
  - CUDA (GPU)
  - OpenCL (GPU/CPU)

Example

```
int balance =500;
void deposit ( int sum ) {
  int currbalance=balance ; /* read balance */
  ...
  currbalance+=sum ;
  balance=currbalance ; /* write balance */
}
void withdraw ( int sum ) {
  int currbalance=balance ; /* read balance */
  if ( currbalance >0)
   currbalance-=sum ;
 balance=currbalance ; /* write balance */
}
.. deposit (100); /* thread 1*/
.. withdraw (50);/ thread 2*/
.. withdraw (100); /* thread 3*/
...
```

- minimize use of global/static memory
- Scenario: T1(read),T2(read,write),T1(write) ,balance=600
- Scenario: T2(read),T1(read,write),T2(write) ,balance=450

Pthread

- Unlike Java, multithreading is not supported by the language standard.
- POSIX (Portable Operating System Interface, specified by IEEE)
- Threads (or Pthreads) is a POSIX standard for threads.
  - Included in the gcc compiler implementation

Creating threads

```c
int pthread_create(pthread_t * thread,
                   const pthread_attr_t * attr,
                   void *(*start_routine)(void*), void * arg);
```

- creates a new thread with the attributes specified by attr.
- Default attributes are used if attr is NULL.
  - E.g. set private stack size
- calls function start_routine(arg) on a separate thread of execution.
- returns zero on success, non-zero on error.

```c
void pthread_exit(void *value_ptr);
```

- called implicitly when thread function exits.
-

Example

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
  long tid;
  tid = (long)threadid;
  printf("Hello World! It's me, thread #%ld!\n", tid);
  pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;
  for(t=0;t<NUM_THREADS;t++){
    printf("In main: creating thread %ld\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
      }
    }

  /* Last thing that main() should do */
  pthread_exit(NULL);
}
```

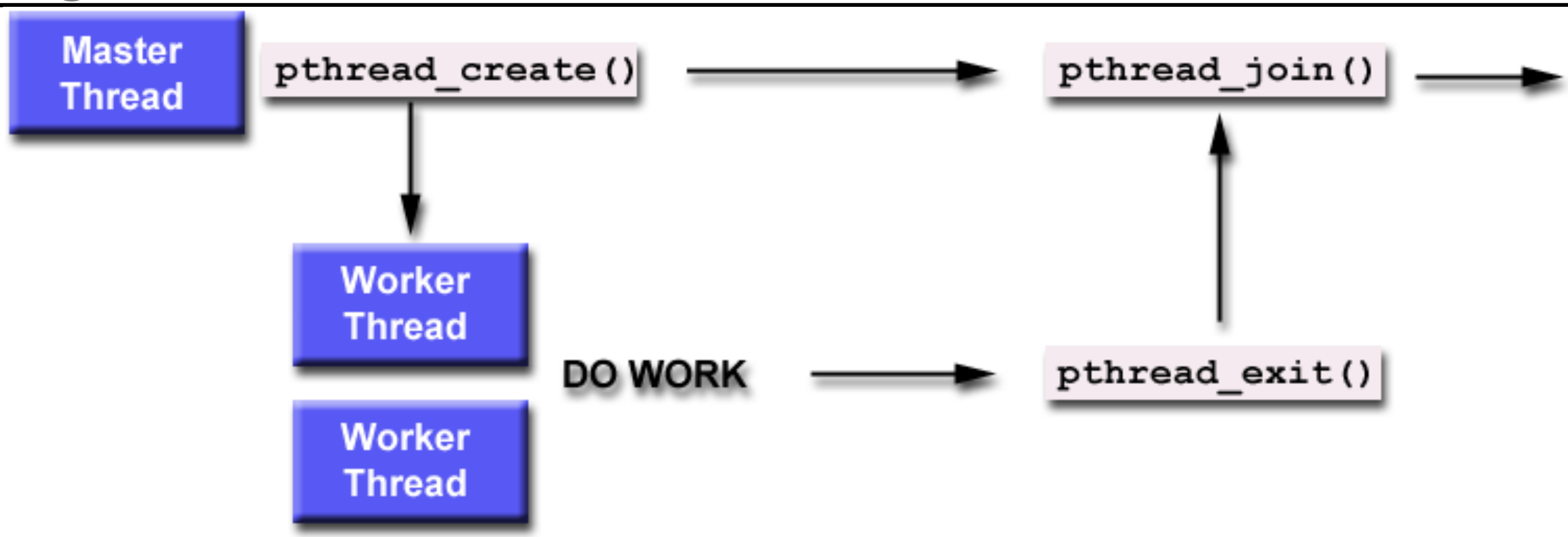
Possible output

```
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
Hello World! It's me, thread #1!
In main: creating thread 2
In main: creating thread 3
Hello World! It's me, thread #2!
Hello World! It's me, thread #3!
In main: creating thread 4
Hello World! It's me, thread #4!
```

```
In main: creating thread 0
Hello World! It's me, thread #0!
In main: creating thread 1
Hello World! It's me, thread #1!
In main: creating thread 2
Hello World! It's me, thread #2!
In main: creating thread 3
Hello World! It's me, thread #3!
In main: creating thread 4
Hello World! It's me, thread #4!
```

Synchronization

Joining




```
int pthread_join(pthread_t thread, void **value_ptr);
```

- pthread_join() blocks the calling thread until the specified thread terminates.
- If value_ptr is not null, it will contain the return status of the ending thread.
- Other ways to synchronize: mutex,condition variables

```
#define _OPEN_THREADS
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

void *thread(void *arg) {
  char *ret;
  printf("thread() entered with argument '%s'\n", arg);
  if ((ret = (char*) malloc(20)) == NULL) {
    perror("malloc() error");
    exit(2);
  }
  strcpy(ret, "This is a test");
  pthread_exit(ret);
}

main() {
  pthread_t thid;
  void *ret;

  if (pthread_create(&thid, NULL, thread, "thread 1") != 0) {
    perror("pthread_create() error");
    exit(1);
  }

  if (pthread_join(thid, &ret) != 0) {
    perror("pthread_create() error");
    exit(3);
  }

  printf("thread exited with '%s'\n", ret);
}
```

```
   printf("Thread %ld done. Result = %e\n",tid, result);
   pthread_exit((void*) t);
}
```

Example

```
int main (int argc, char *argv[])
{
   pthread_t thread[NUM_THREADS];
   pthread_attr_t attr;
   int rc;
   long t;
   void *status;

   /* Initialize and set thread detached attribute */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   for(t=0; t<NUM_THREADS; t++) {
      printf("Main: creating thread %ld\n", t);
      rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
      if (rc) {
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
         }
      }

   /* Free attribute and wait for the other threads */
   pthread_attr_destroy(&attr);
   for(t=0; t<NUM_THREADS; t++) {
      rc = pthread_join(thread[t], &status);
      if (rc) {
         printf("ERROR; return code from pthread_join() is %d\n", rc);
         exit(-1);
         }
      printf("Main: completed join with thread %ld having a status of %ld\n",t,(long)status);
      }

printf("Main: program completed. Exiting.\n");
pthread_exit(NULL);
}
```

Mutex

- Mutex (mutual exclusion) acts as a "lock" protecting access to the shared resource.
- Only one thread can "own" the mutex at a time. Threads must take turns to lock the mutex.

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t * mutex,
                       const pthread_mutexattr_t * attr);
thread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- pthread_mutex_init() initializes a mutex. If attributes are NULL, default attributes are used.
- The macro PTHREAD_MUTEX_INITIALIZER can be used to initialize static mutexes.
- pthread_mutex_destroy() destroys the mutex.
- Both function return return 0 on success, non zero on error.

Mutex

```c
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- pthread_mutex_lock() locks the given mutex. If the mutex is locked, the function is blocked until it becomes available.
- pthread_mutex_trylock() is the non-blocking version. If the mutex is currently locked the call will return immediately.
- pthread_mutex_unlock() unlocks the mutex.

Example revisted

```
int balance=500;
void deposit(int sum){
  int currbalance=balance;/*read balance*/
  ...
  currbalance+=sum;
  balance=currbalance;/*write balance*/
}

void withdraw(int sum){
  int currbalance=balance;/*read balance*/
  if(currbalance>0)
    currbalance-=sum;
  balance=currbalance; /*write balance*/
}
  ..
  deposit(100);/*thread 1*/
  ..
  withdraw(50);/thread 2*/
  ..
  withdraw(100);/*thread 3*/
    ...
```

- Scenario: T1(read),T2(read,write),T1(write),balance=600
- Scenario: T2(read),T1(read,write),T2(write),balance=450

Using Mutex

```
int balance=500;
pthread_mutex_t mutexbalance=PTHREAD_MUTEX_INITIALIZER;

void deposit(int sum){
  pthread_mutex_lock(&mutexbalance);
  {
  int currbalance=balance;/*read balance*/
  ...
  currbalance+=sum;
  balance=currbalance;/*write balance*/
  }
  pthread_mutex_unlock(&mutexbalance);
}
void withdraw(int sum){
  pthread_mutex_lock(&mutexbalance);
  {
  int currbalance=balance;/*read balance*/
  if(currbalance>0)
    currbalance-=sum;
  balance=currbalance; /*write balance*/
  }
  pthread_mutex_unlock(&mutexbalance);
}
..  deposit(100);/*thread 1*/
..  withdraw(50);/thread 2*/
..  withdraw(100);/*thread 3*/
```

- Scenario: T1(read,write),T2(read,write),balance=450
- Scenario: T2(read),T1(read,write),T2(write),balance=450

Condition variables

- Sometimes locking or unlocking is based on a run-time condition.
  - Without condition variables, program would have to poll the variable/condition continuously.

Consumer:

(a) lock mutex on global item variable

(b) wait for (item>0) signal from producer (mutex unlocked automatically).

(c) wake up when signaled (mutex locked again)

- automatically), unlock mutex and proceed.

Producer:

(1) produce something

(2) Lock global item variable, update item

(3) signal waiting (threads)

(4) unlock mutex

Condition variables

```c
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t * cond,const pthread_condattr_t * attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- pthread_cond_init() initialized the condition variable. If attr is NULL, default attributes are set.
- pthread_cond_destroy() will destroy (uninitialize) the condition variable.
- destroying a condition variable upon which other threads are currently blocked results in undefined behavior.
- macro PTHREAD_COND_INITIALIZER can be used to initialize condition variables. No error checks are performed.
- Both function return 0 on success and non-zero otherwise.

Condition variables

`int pthread_cond_wait(pthread_cond_t *cond,pthread_mutex_t *mutex);`

- blocks on a condition variable.
- must be called with the mutex already locked otherwise behavior undefined.
- automatically releases mutex
- upon successful return, the mutex will be automatically locked again.

`int pthread_cond_broadcast(pthread_cond_t *cond);`

`int pthread_cond_signal(pthread_cond_t *cond);`

- unblocks threads waiting on a condition variable.
- pthread_cond_broadcast() unlocks all threads that are waiting.
- pthread_cond_signal() unlocks one of the threads that are waiting.
- both return 0 on success, non zero otherwise.

Example

```
#include<pthread.h>
pthread_cond_t cond_recv=PTHREAD_COND_INITIALIZER;
pthread_cond_t cond_send=PTHREAD_COND_INITIALIZER;
pthread_mutex_t cond_mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t count_mutex=PTHREAD_MUTEX_INITIALIZER;

int full=0;
int count=0;
```

```
void* produce(void*)
{
  while(1)
  {
      pthread_mutex_lock(&cond_mutex);
    while(full)
    {
    pthread_cond_wait(&cond_recv,
                &cond_mutex);
    }
      pthread_mutex_unlock(&cond_mutex);
      pthread_mutex_lock(&count_mutex);
      count++; full=1;
      printf("produced(%d):%d\n",
      pthread_self(), count);
      pthread_cond_broadcast(&cond_send);
      pthread_mutex_unlock(&count_mutex);
      if(count>=10) break;
  }
```

```
void* consume(void*)
{
  while(1)
    {
      pthread_mutex_lock(&cond_mutex);
      while(!full)
    {
    pthread_cond_wait(&cond_send,
    &cond_mutex);
    }
      pthread_mutex_unlock(&cond_mutex);
      pthread_mutex_lock(&count_mutex);
      full=0;
      printf("consumed(%ld):%d\n",
      pthread_self(), count);
      pthread_cond_broadcast(&cond_recv);
      pthread_mutex_unlock(&count_mutex);
      if(count>=10) break;
    }
```

Example

```
int main ()
{
   pthread_t cons_thread , prod_thread ;
   pthread_create (&prod_thread ,NULL,produce ,NULL);
   pthread_create (&cons_thread ,NULL,consume ,NULL);

   pthread_join(cons_thread ,NULL);
   pthread_join( prod_thread ,NULL);
   return 0;
}
```

Output:

```
produced(3077516144):1
consumed(3069123440):1
produced(3077516144):2
consumed(3069123440):2
produced(3077516144):3
consumed(3069123440):3
produced(3077516144):4
consumed(3069123440):4
produced(3077516144):5
consumed(3069123440):5
produced(3077516144):6
consumed(3069123440):6
produced(3077516144):7
consumed(3069123440):7
```

Race conditions

- Race conditions occur when multiple threads share a variable, without proper synchronization
- Synchronization uses special variables, like a mutex, to ensure order of execution is correct
- Example: thread T1 needs to do something before thread T2
  - condition variable forces thread T2 to wait for thread T1
  - producer-consumer model program
- Example: two threads both need to access a variable and modify it based on its value
  - surround access and modification with a mutex
  - mutex groups operations together to make them atomic – treated as one unit

- Consider the following program `race.c`:
  - What is the value of `cnt`?

```
unsigned int cnt = 0;

void *count(void *arg){/* thread body */
  int i;
  for (i = 0; i < 100000000; i ++)
     cnt ++;
  return NULL;
}

int main(void) {
  pthread_t tids[4];
  int i;
  for (i = 0; i < 4; i++)
   pthread_create(&tids[i], NULL, count, NULL);
  for (i = 0; i < 4; i ++)
   pthread_join(tids[i], NULL); printf("cnt=%u\n", cnt
  );
  return 0;
}
```

- Ideally, should increment `cnt` 4×100000000 times, so `cnt` = 400000000.
- However, running our code gives:
- `./race.o   cnt=137131900`
- `./race.o   cnt=163688698`
- `./race.o   cnt=163409296`
- `./race.o   cnt=170865738`
- `./race.o   cnt=169695163`
- So, what happened?

Race conditions

On assembly level

- C not designed for multithreading
- No notion of atomic operations in C
- Increment cnt++; maps to three assembly operations:
  - load cnt into a register
  - increment value in register
  - save new register value as new cnt
- So what happens if thread interrupted in the middle?

- Race condition!

Race conditions

Fixed example

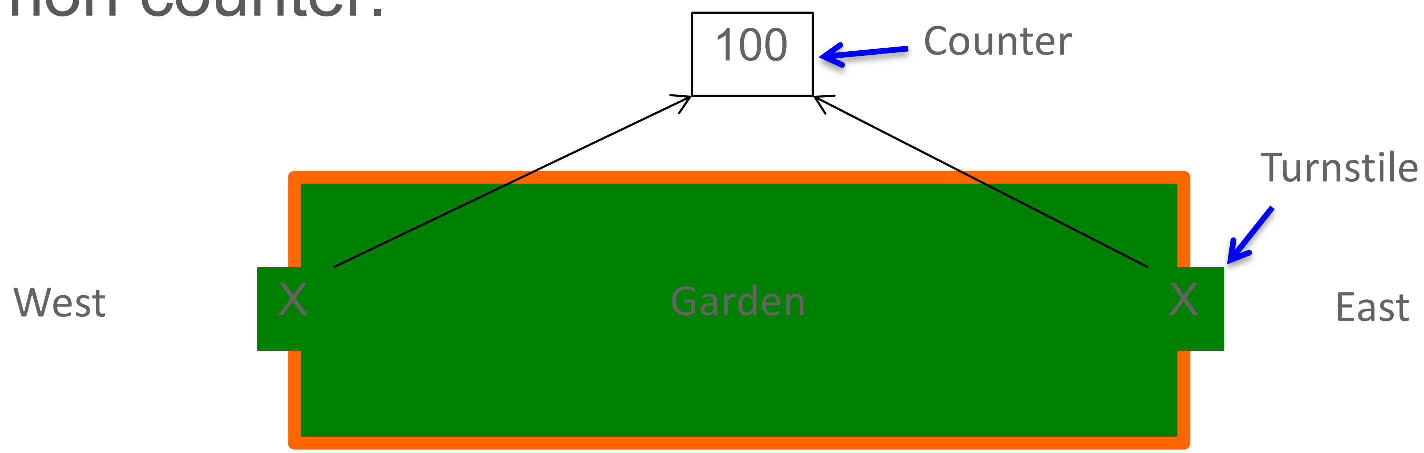
```
pthread_mutex_t mutex;
unsigned int cnt = 0;

void *count(void *arg) { /* thread body */
  int i;
  for (i = 0; i < 100000000; i++) {
    pthread_mutex_lock(&mutex);
    cnt++;
    pthread_mutex_unlock(&mutex);
  }
  return NULL;
}

int main(void) {
  pthread_t tids[4];
  int i;
  pthread_mutex_init(&mutex, NULL);
  for (i = 0; i < 4; i++)
    pthread_create(&tids[i], NULL, count, NULL);
  for (i = 0; i < 4; i++)
    pthread_join(tids[i], NULL); pthread_mutex_destroy(&mutex);
    printf("cnt=%u\n", cnt);
  return 0;
}
```

Race conditions

- Note that new code functions correctly, but is much slower
- C statements are not atomic – threads may be interrupted at assembly level, in the middle of a C statement
- Atomic operations like mutex locking must be specified as atomic using special assembly instructions
- Ensure that all statements accessing/modifying shared variables are synchronized

Example

- We consider a botanical garden that can be entered through two entrances (West and East)
- For the sake of simplification, visitors cannot leave the garden again.
- At each entrance, visitors must pass a turnstile
- Each turnstile has a counting device that counts up a common counter.



- Develop a C Program including Pthreads!