



UNIVERSITÀ  
DEGLI STUDI  
FIRENZE

# Parallel Computing

Prof. Marco Bertini



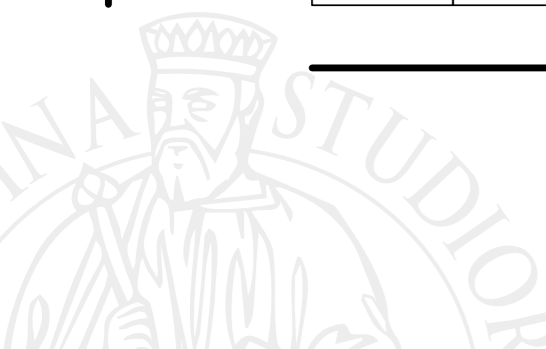
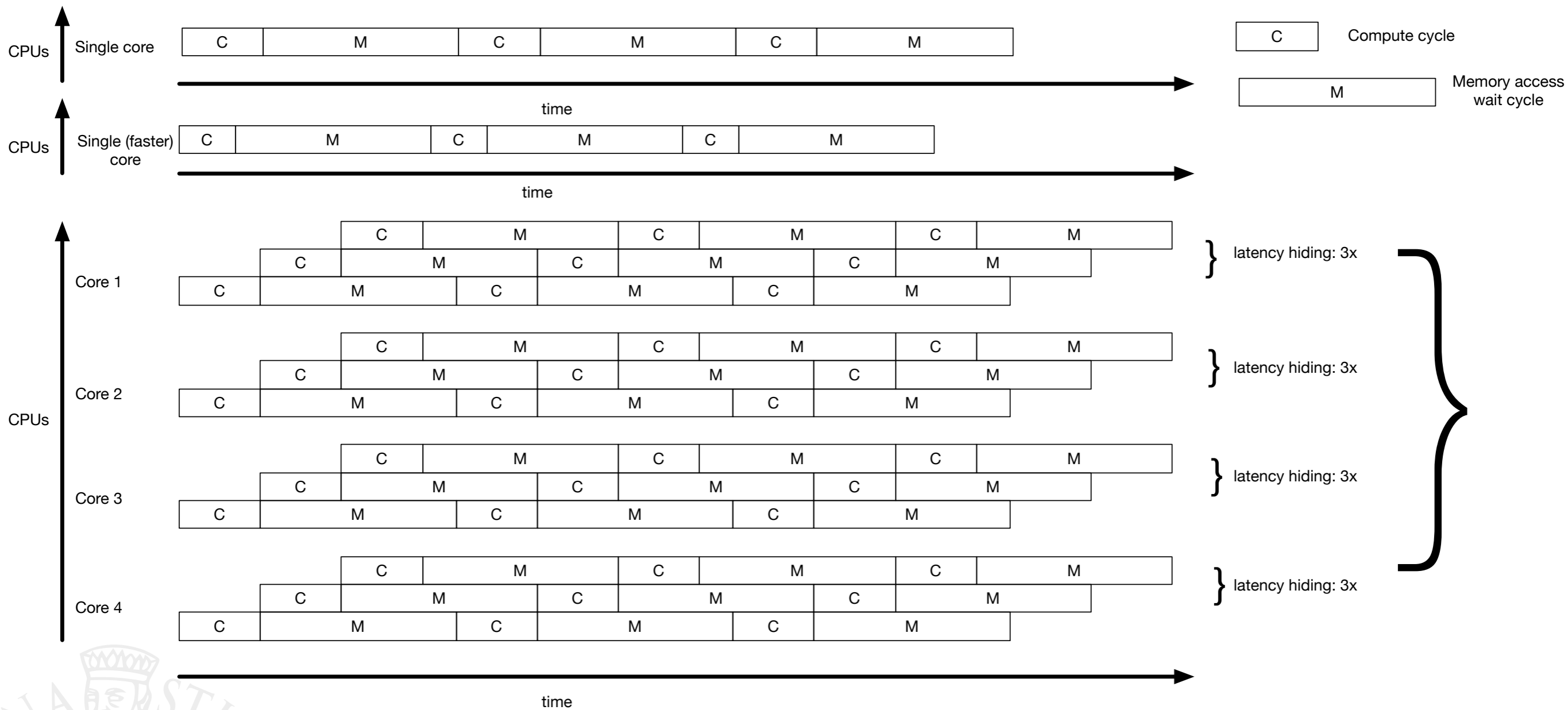


# Shared memory: threads

# Threads: motivations

- Software Portability
  - run on serial and parallel machines
- Latency Hiding
  - While one thread has to wait, others can utilize CPU
  - For example: file reading, message reading, reading data from higher-level memory
- Scheduling and Load Balancing
  - Large number of concurrent tasks System-level dynamic mapping to processors
- Ease of Programming
  - APIs and libraries in many different languages

# Multi thread latency



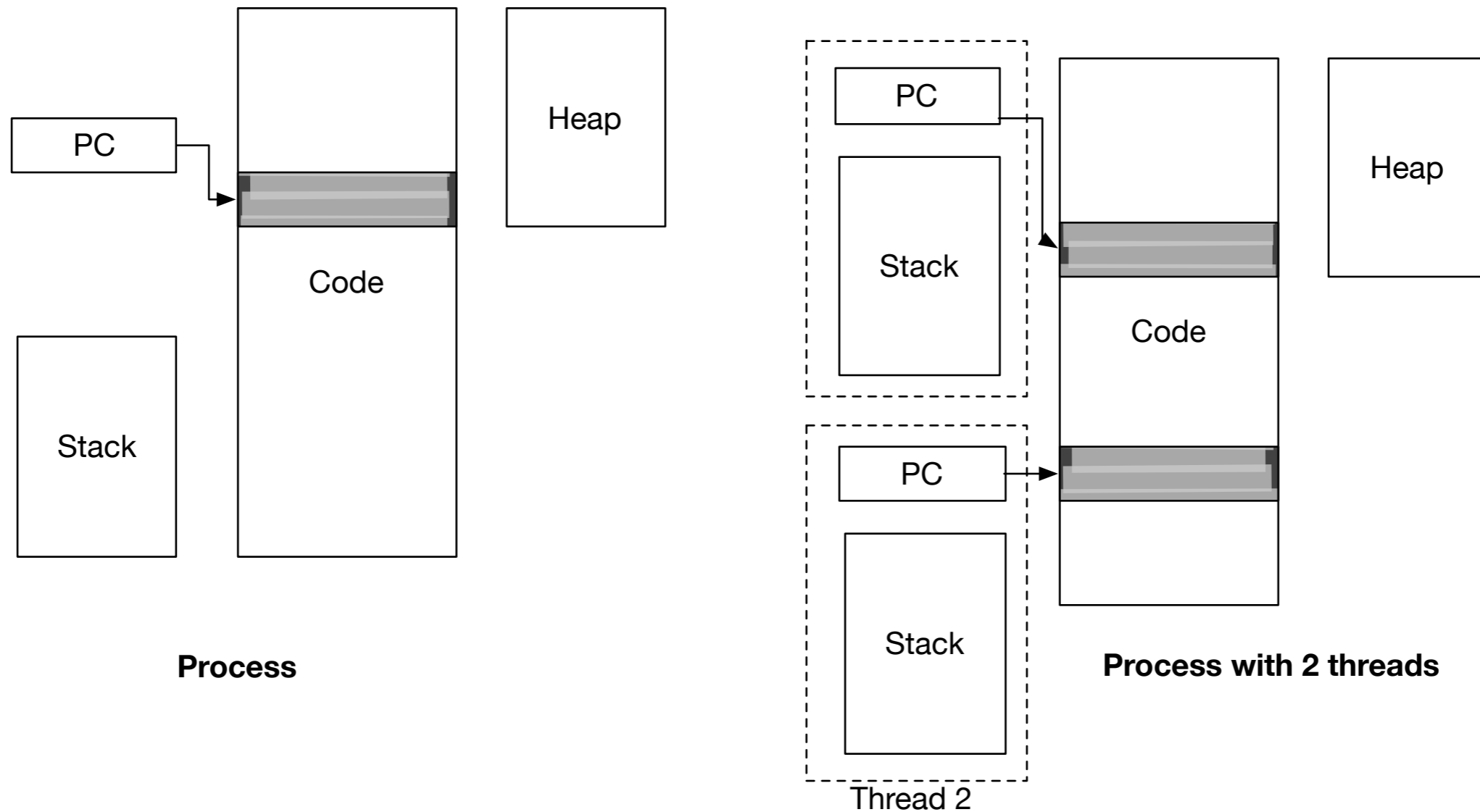
# Processes and threads

- Parallel programming models are often based on processors or threads.
- A **process** is defined as a program in execution.
  - The process comprises the executable program along with all information that is necessary for the execution of the program:
    - program data on the runtime stack or the heap, the current values of the registers, as well as the content of the program counter
    - Each process has its own address space, i.e., the process has exclusive access to its data. When two processes want to exchange data, this has to be done by explicit communication.
- In the thread model, each process may consist of multiple independent control flows which are called threads
- In the thread model, each process may consist of multiple independent control flows which are called **threads**.
  - Threads of one process share the address space of the process, i.e., they have a common address space.

# Processes and threads

- Parallel programming models are often based on processors or threads.
- A **process** is defined as a program in execution.
  - The process comprises the executable program along with all information that is necessary for the execution of the program:
    - program data on the runtime stack or the heap, the current values of the registers, as well as the content of the program counter
  - **Threads** are also called **light-weight processes**. The term thread comes from the concept of “thread of control”, i.e. a sequence of statement in a program.
- In the thread model, each process may consist of multiple independent control flows which are called threads
- In the thread model, each process may consist of multiple independent control flows which are called **threads**.
- Threads of one process share the address space of the process, i.e., they have a common address space.

# Process vs. Thread



**Process**

**Process with 2 threads**

Thread 2

There are several thread APIs: Windows, Java, Linux...  
There's also a standard: POSIX (Pthreads)



# Threads: H/W and S/W

- Software threads: scheduling and context switching performed by O.S. or library
- Hardware threads: scheduling and context switching performed by hardware
  - H/W must support separate registers and logic for each thread
  - cheap context switching
  - e.g. Intel Hyperthreading: each thread appears as logical processor.
  - Common in GPUs



# Threads

- When a thread stores a value in the shared address space, another thread of the same process can access this value afterwards.
  - Very practical in multi-core programming: fast information exchange.
- Fast to create: no need to create a new context, therefore reduced cost for context switch.
- Threads can be provided by the runtime system as **user-level** threads or by the operating system as **kernel threads**.
  - O.S. is aware and manages kernel threads, optimizing their allocation to cores/CPU

# Threads: states

- A thread can be in one of the following states:
  - **newly generated**: the thread has just been generated, but has not yet performed any operation;
  - **executable**: the thread is ready for execution, but is currently not assigned to any execution resources;
  - **running**: the thread is currently being executed by an execution resource;
  - **waiting**: the thread is waiting for an external event to occur; the thread cannot be executed before the external event happens;
  - **finished**: the thread has terminated all its operations.
- The transitions between the states **executable** and **running** are determined by the scheduler. **Waiting** is entered because of blocking operations (e.g. I/O) or synchronization operation. The transition from the state **waiting** to **executable** may be caused by conclusion of blocking operation or release (by another thread) of a required resource.

# Threads: states

- A thread can be in one of the following states:

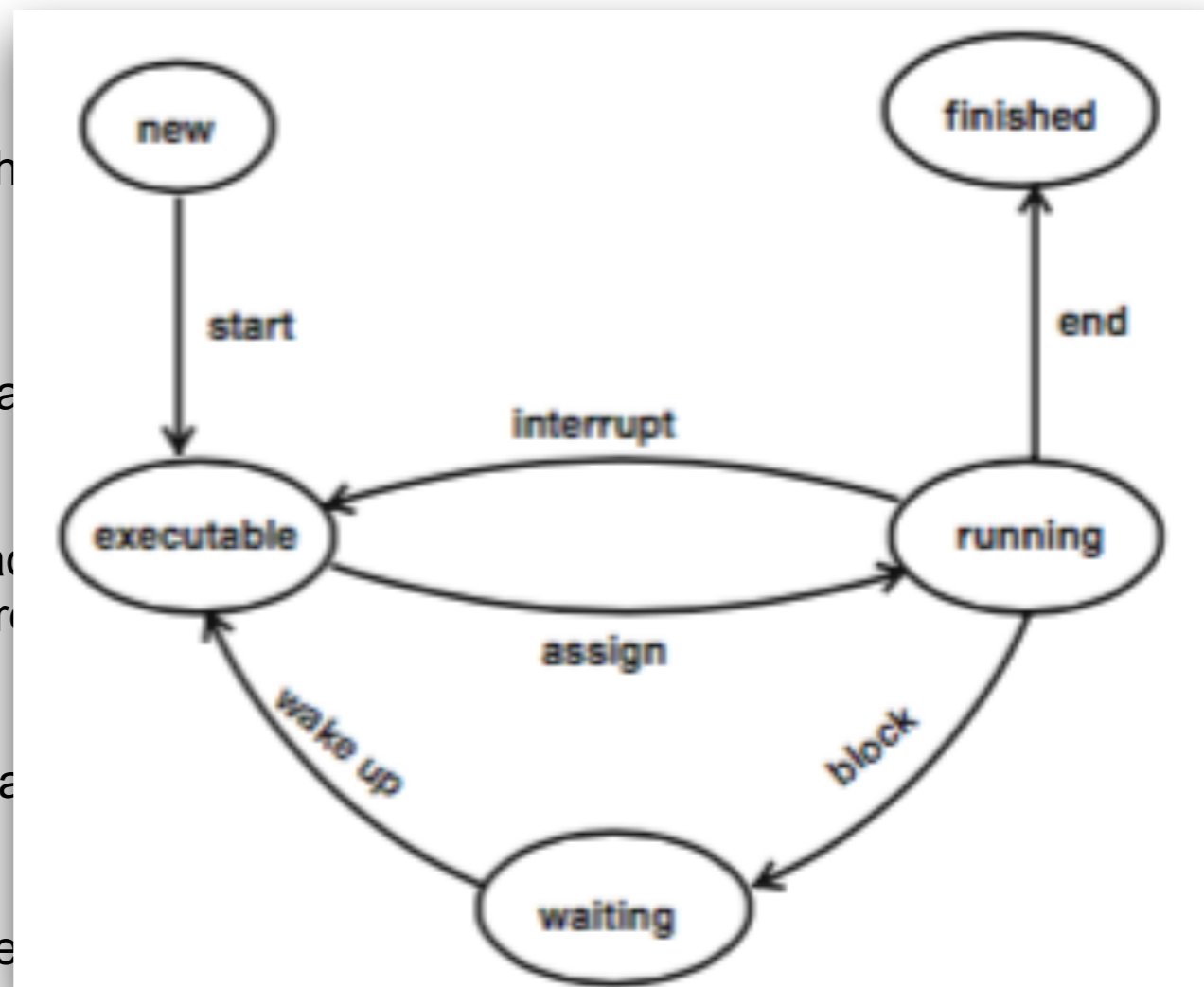
- **newly generated**: the thread has just been generated, but has not yet performed any operation;

- **executable**: the thread has been assigned resources;

- **running**: the thread is currently executing;

- **waiting**: the thread is waiting to be executed before it can start;

- **finished**: the thread has completed its execution.



to any execution

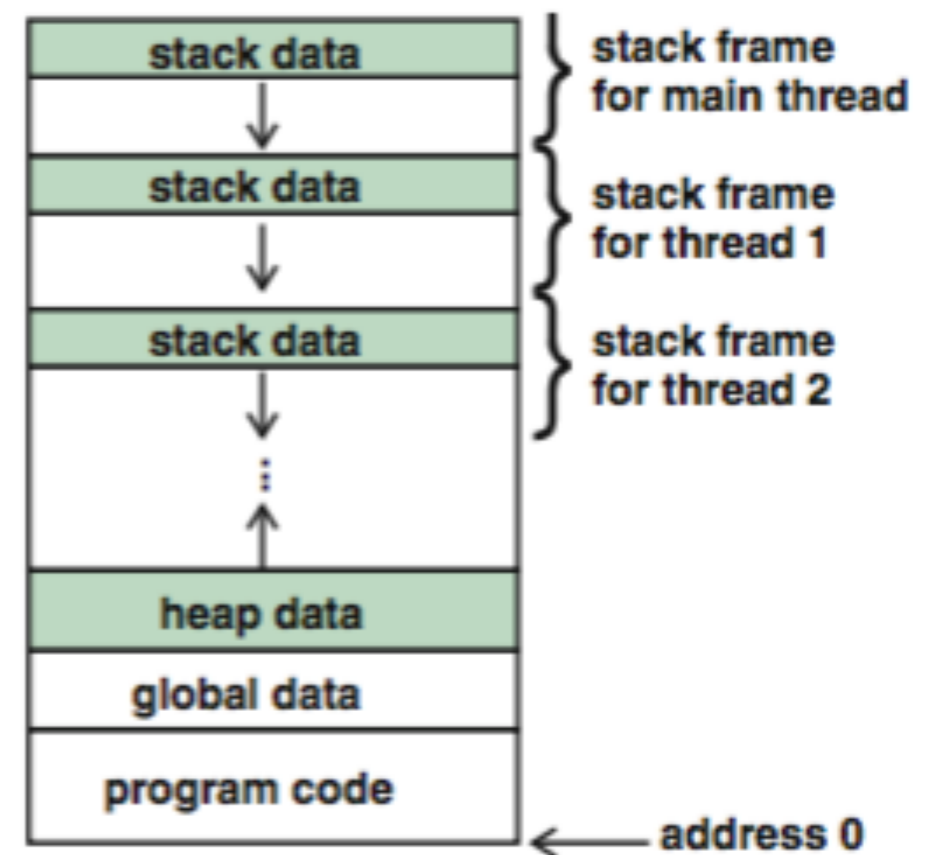
not

the scheduler.

- The transitions between **executable** and **running** are managed by the scheduler. **Waiting** is entered because of blocking operations (e.g. I/O) or synchronization operation. The transition from the state **waiting** to **executable** may be caused by conclusion of blocking operation or release (by another thread) of a required resource.

# Threads: data

- Different threads share a common address space of the process they belong to: static and dynamically allocated data can be accessed by all threads
- Each thread has also a private runtime stack for controlling function calls of this thread and to store the local variables of these functions.
  - It exists only as long as the thread is active: it's freed as soon as the thread is terminated.



# Threads: synchronization

- Execution of threads must be coordinated to avoid race conditions:
  - The term **race condition** describes the effect that the result of a parallel execution of a program part by multiple execution units depends on the order in which the statements of the program part are executed by the different units.  
In the presence of a race condition it may happen that the computation of a program part leads to different results, depending on whether thread  $T_1$  executes the program part before  $T_2$  or vice versa.

# Threads: synchronization

Remind: when 2 threads run simultaneously, we cannot determine which one is first or which one is faster...

- The term **race condition** describes the effect that the result of a parallel execution of a program part by multiple execution units depends on the order in which the statements of the program part are executed by the different units.  
In the presence of a race condition it may happen that the computation of a program part leads to different results, depending on whether thread  $T_1$  executes the program part before  $T_2$  or vice versa.

# Threads: synchronization

- Program parts in which concurrent accesses to shared variables by multiple threads may occur, thus holding the danger of the occurrence of inconsistent values, are called **critical sections**.
- An error-free execution can be ensured by letting only one thread at a time execute a critical section. This is called **mutual exclusion**.
- The programmer identifies critical sections in the program and protects them with a synchronization mechanism that is locked when the critical section is entered and locked when the critical section is left.
- This lock mechanism guarantees that the critical section is entered by one thread at a time, leading to mutual exclusion.



# Threads: synchronization

- Lock/unlock operations must be atomic instructions, like test-and-set, fetch-and-add or compare-and-swap.
- Intel processors can run atomically with a lock prefix, several instructions. The lock will perform a cache lock, and if required also a bus lock.
- The programmer identifies critical sections in the program and protects them with a synchronization mechanism that is locked when the critical section is entered and locked when the critical section is left.
- This lock mechanism guarantees that the critical section is entered by one thread at a time, leading to mutual exclusion.



# Threads: synchronization

- Synchronization mechanisms are provided to:
  - enable a coordination, e.g., to ensure a certain execution order of the threads or to control access to shared data structures;
  - avoid a concurrent manipulation of the same (shared) variable by different threads, which may lead to non-deterministic behavior.
- Different synchronization mechanisms are provided for different situations.

# Lock synchronization: lock

- **Lock / mutex variables:** a lock variable  $l$  can be in one of two states: **locked** or **unlocked**.  
Two operations are provided to influence this state:  $lock(l)$  and  $unlock(l)$ .  
The execution of  $lock(l)$  locks  $l$  such that it cannot be locked by another thread; after the execution,  $l$  is in the locked state and the thread that has executed  $lock(l)$  is the owner of  $l$ .  
The execution of  $unlock(l)$  unlocks a previously locked lock variable  $l$ ; after the execution,  $l$  is in the unlocked state and has no owner.
- To avoid race conditions for the execution of a program part, a lock variable  $l$  is assigned to this program part and each thread executes  $lock(l)$  before entering the program part and  $unlock(l)$  after leaving the program part. To avoid race conditions, each of the threads must obey this programming rule.
- Using this lock mechanism leads to sequentialization of the execution: a lock can allow only one thread to execute a part of a program.

# Lock synchronization: lock

```
// Note: edx register contains address of lock variable.
```

```
// Lock variable is free if it's 0
```

```
// move 1 into eax register
```

```
mov 1, eax
```

```
// xchg 1 with value contained in dereferenced edx
```

```
lock xchg eax, [edx] ; xchg is atomic on Intel CPU
```

```
// test if zero: it's bitwise AND so the zero flag ZF is 0 only if eax is 0
```

```
test eax, eax
```

```
// jump if not zero (i.e. if [edx] was not 0)
```

```
jne Target
```

# Lock synchronization: semaphore

- A **semaphore** is a data structure which contains an integer counter  $s$  and to which two atomic operations  $P(s)$  and  $V(s)$  can be applied. A binary semaphore  $s$  can only have values 0 or 1. For a counting semaphore,  $s$  can have any positive integer value.  
The operation  $P(s)$ , also denoted as *wait(s)*, waits until the value of  $s$  is larger than 0. When this is the case, the value of  $s$  is decreased by 1, and execution can continue with the subsequent instructions.  
The operation  $V(s)$ , also denoted as *signal(s)*, increments the value of  $s$  by 1.

# Lock synchronization: semaphore

- A **semaphore** is a data structure which contains an integer counter and two atomic operations  $P(s)$  and  $V(s)$ .  
A binary semaphore is a *critical section* semaphore with values 0 or 1.  
For a counting semaphore,  $s$  can have any positive integer value.  
 $wait(s)$   
 $signal(s)$ .

The operation  $P(s)$ , also denoted as  $wait(s)$ , waits until the value of  $s$  is larger than 0. When this is the case, the value of  $s$  is decreased by 1, and execution can continue with the subsequent instructions.

The operation  $V(s)$ , also denoted as  $signal(s)$ , increments the value of  $s$  by 1.

# Lock synchronization: monitor

- A **monitor** is a language construct which allows the definition of data structures and access operations.
- These operations are the only means by which the data of a monitor can be accessed.
- The monitor ensures that the access operations are executed with mutual exclusion, i.e., at each point in time, only one thread is allowed to execute any of the access methods provided.



# Execution control: barrier

- A **barrier synchronization** defines a synchronization point where each thread must wait until all other threads have also reached this synchronization point.
  - No thread executes any statement after the synchronization point until all other threads have also arrived at this point.
- A barrier synchronization also has the effect that it defines a global state of the shared address space in which all operations specified before the synchronization point have been executed.
  - Statements after the synchronization point can be sure that this global state has been established.

# Execution control: condition

- In this approach a thread  $T_1$  is blocked until a given condition has been established (**condition synchronization**).
- Condition synchronization can be supported by **condition variables**. Could/must be used together with a lock variable to avoid race condition when evaluating the condition.





# # threads and sequentialization

- A parallel program should create a sufficiently large number of threads to provide enough work for all cores of an execution platform, but if the number is too large we pay an overhead for thread creation, management, termination and access to shared resources (e.g. cache).
- We need to use synchronization to ensure correct behaviour, but too many synchronizations result in sequentialization, i.e. we do not achieve parallelism.



# Deadlock

- Another possible issue with locks is **deadlock**: when program execution comes into a state where each thread waits for an event that can only be caused by another thread, but this thread is also waiting.
- **Dining philosophers problem**

Five silent philosophers sit at a round table with bowls of rice. Chopsticks are placed between each pair of adjacent philosophers. Each philosopher must alternately think and eat. A philosopher can only eat when he has both left and right chopsticks. Each chopstick can be held by only one philosopher, a philosopher can use the chopstick only if it is not being used by another one. After he finishes eating, he needs to put down both chopsticks so they become available to others. A philosopher can take the chopstick on his right or the one on his left as they become available, but cannot start eating before getting both of them.

*The philosophers do not speak to each other and there is no arbiter organizing the resources.*



# Deadlock

- Another possible issue with locks is **deadlock**: when program execution comes into a state where each thread waits for an event that can only be caused by another thread, but this thread is also waiting.

- **Dining philosophers problem**

Five silent philosophers sit at a round table with chopsticks and plates of rice in front of them. Each philosopher has two chopsticks. A philosopher can only eat if he has two chopsticks: one on his left and one on his right. After he finishes eating, he puts down the chopsticks and the philosopher to his left can start eating. *The philosopher has no arbiter or*

Another example of deadlock  
(similar in spirit):

Thread  $T_1$

```
lock(s1);  
lock(s2);  
do work();  
unlock(s2);  
unlock(s1);
```

Thread  $T_2$

```
lock(s2);  
lock(s1);  
do work();  
unlock(s1);  
unlock(s2);
```



# Deadlocks

- There are four conditions associated to the creation of deadlocks:
  1. Mutual exclusion
  2. Hold and wait: threads hold some resources and request other
  3. No preemption: resource can only be released by the thread that holds it
  4. Circular wait: cycle in waiting of a thread for a resource of another

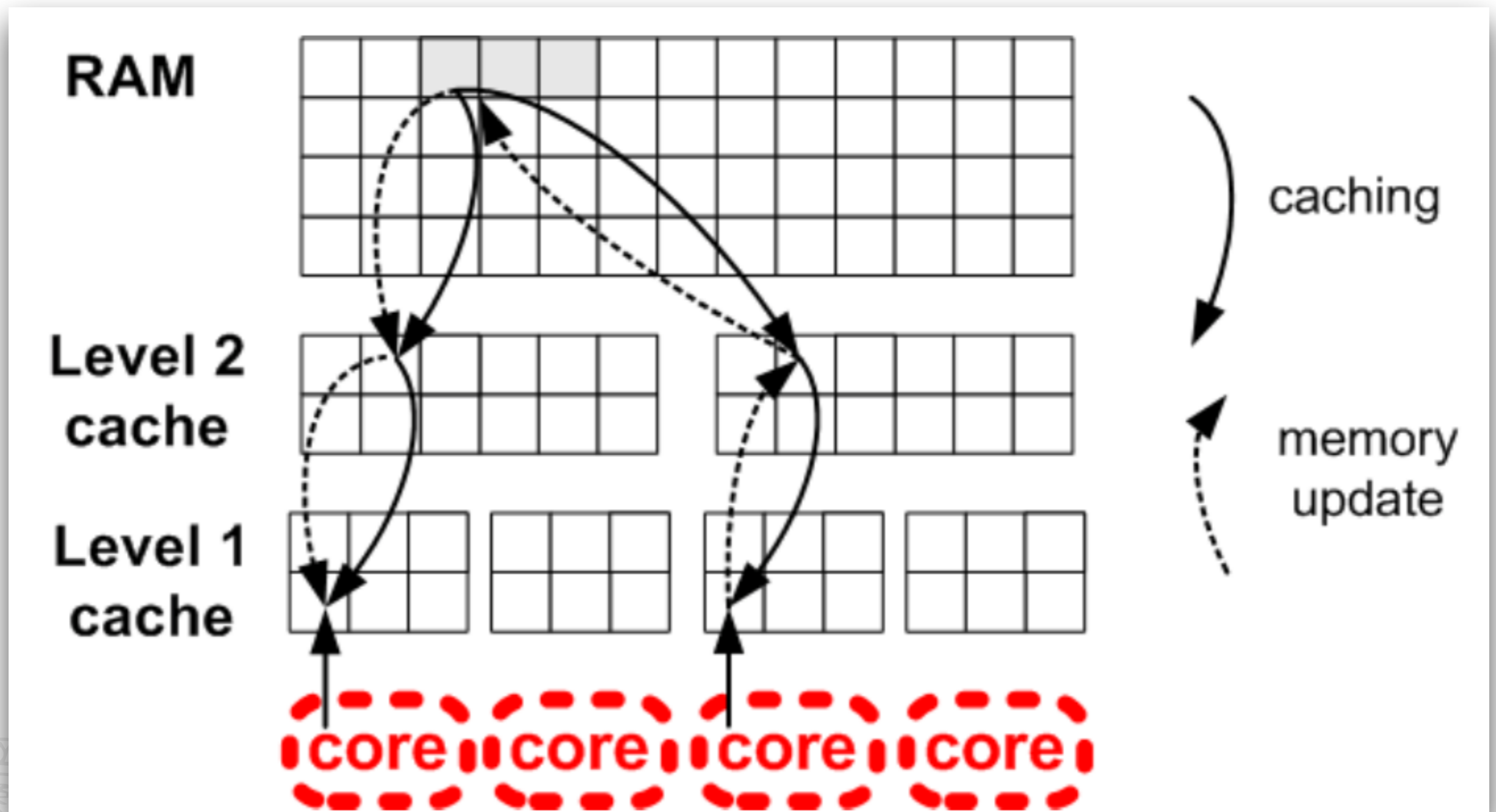
# Livelock

- Similar to a deadlock, except that the states of the processes involved in the **livelock** constantly change with regard to one another, none progressing.
- Real-world example: two people meet in a narrow corridor, each moves aside to let the other pass, but they end up swaying from side to side
- A risk with algorithms that detect and recover from deadlock.

# Memory access

- The transfer within the memory hierarchy can be captured by dependencies between the memory accesses issued by different cores. These dependencies can be categorized as read–read dependency, read–write dependency, and write–write dependency.
- Depending on the specific pattern of read and write operations, not only is there a transfer from *main memory to the local caches* of the cores, but there may also be a transfer *between the local caches of the cores*. Several copies of same data may reside in caches. The exact behavior is controlled by hardware, and the programmer has no direct influence on this behavior.

# Memory access



A cache coherence mechanism updates the copies.



# False sharing

- It is an issue that arises when threads use different objects but those objects happen to be close enough in memory that they fall on the same cache line, and the cache system treats them as a single lump that is effectively protected by a hardware write lock that only one core can hold at a time.
- This causes real but invisible performance contention: the parallel version of the code may become slower than the sequential version.





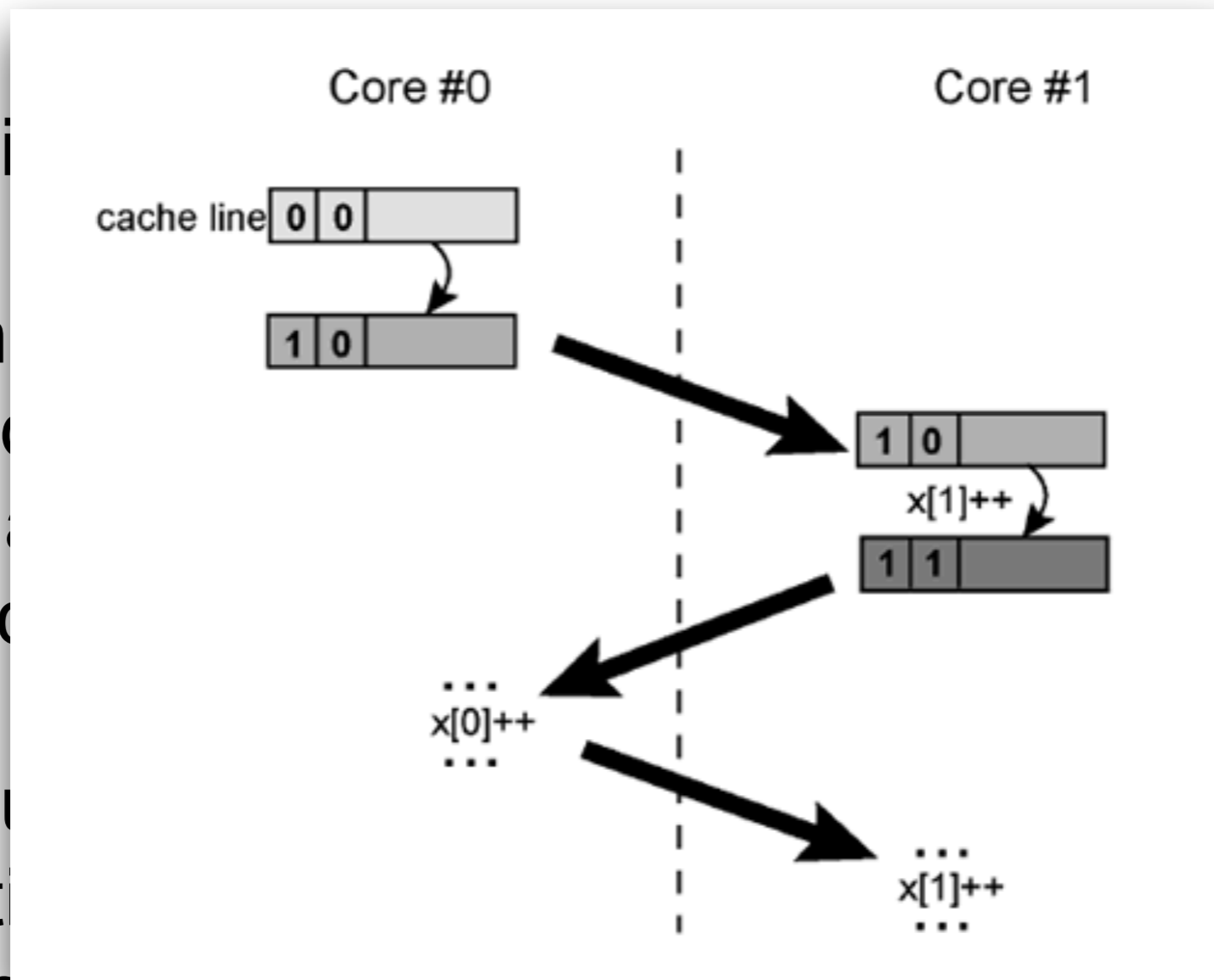
# False sharing

- It is an issue that arises when threads use different objects but those objects happen to be close enough in memory that they fall on the same cache line, and the cache system treats them as a single lump that is effectively protected by a hardware write lock that only one core can hold at a time.
- This causes real but invisible performance contention: the parallel version of the code may become slower than the sequential version.

The program becomes serial since the thread that currently has exclusive ownership, so that it can physically perform an update to the cache line, will silently throttle other threads that are trying to use different data that sits on the same line.

# False sharing

- It is an issue that occurs when different threads access different objects in memory that happen to be on the same cache line, and one thread writes to the cache line, and the other thread has to flush its cache and reload the data from main memory. This can happen even if the threads are working on different data.
- This can happen when threads access different content that happen to be on the same cache line. This can become slower than the sequential version.



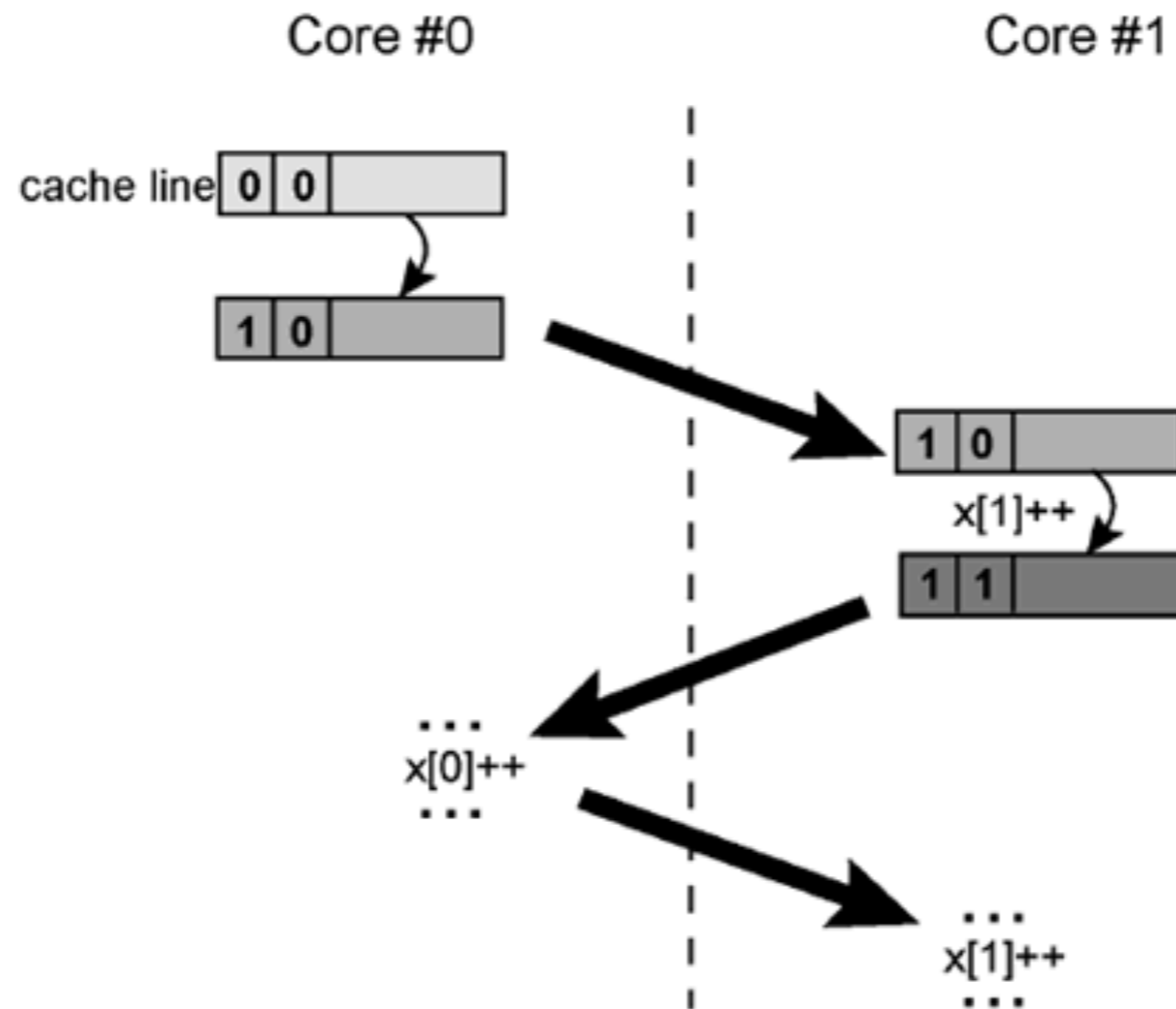
different  
use  
the cache  
a single  
aware  
time.

may

The program becomes serial since the thread that currently has exclusive ownership, so that it can physically perform an update to the cache line, will silently throttle other threads that are trying to use different data that sits on the same line.

Avoiding false sharing may require aligning variables or objects in memory on cache line boundaries. There are a variety of ways to force alignment. Some compilers support alignment pragmas.

- It is an idea to place different objects in memory close enough to each other to share a cache line, and to lump the write locations together.
- This can cause contention for cache content, which may become slower than the sequential version.



different  
use  
the cache  
a single  
aware  
time.

may

The program becomes serial since the thread that currently has exclusive ownership, so that it can physically perform an update to the cache line, will silently throttle other threads that are trying to use different data that sits on the same line.

# False sharing: what to look for

- The general case to watch out for is when you have two objects or fields that are frequently accessed (either read or written) by different threads, at least one of the threads is doing writes, and the objects are so close in memory that they're on the same cache line because they are:
  - objects nearby in the same array;
  - fields nearby in the same object;
  - objects allocated close together in time (C++, Java) or by the same thread (Java);
  - static or global objects that the linker decided to lay out close together in memory;
  - objects that become close in memory dynamically, as when during compacting garbage collection

# False sharing: what to do

1. Reduce the number of writes to the cache line. For example, writer threads can write intermediate results to a scratch variable most of the time, then update the variable in the popular cache line only occasionally as needed.
2. Separate the variables so that they aren't on the same cache line. Typically the easiest way to do this is to ensure an object has a cache line to itself that it doesn't share with any other data. To achieve that, you need to do two things:
  - Ensure that no other object can precede your data in the same cache line by aligning it to begin at the start of the cache line or adding sufficient padding bytes before the object. E.g. Some compilers support alignment pragmas.
  - Ensure that no other object can follow your data in the same cache line by adding sufficient padding bytes after the object to fill up the line.

```
// C++ (using C++11 alignment syntax)
```

```
template<typename T>
```

```
struct alignas(CACHE_LINE_SIZE) cache_line_storage {
```


```
    alignas(CACHE_LINE_SIZE) T data;
```

```
    char pad[ CACHE_LINE_SIZE > sizeof(T)
```

```
        ? CACHE_LINE_SIZE - sizeof(T)
```

```
        : 1 ];
```

```
};
```

1.  Separate the variables so that they aren't on the same cache line. Typically the easiest way to do this is to ensure an object has a cache line to itself that it doesn't share with any other data. To achieve that, you need to do two things:
  - Ensure that no other object can precede your data in the same cache line by aligning it to begin at the start of the cache line or adding sufficient padding bytes before the object. E.g. Some compilers support alignment pragmas.
  - Ensure that no other object can follow your data in the same cache line by adding sufficient padding bytes after the object to fill up the line.



# Credits

- These slides report material from:
  - Prof. Robert van Engelen (Florida State University)
  - Prof. Jan Lemeire (Vrije Universiteit Brussel)





# Books

- Principles of Parallel Programming, Calvin Lyn and Lawrence Snyder, Pearson - Chapt. 6

