Lecture 3 Threads and Locks

Distributed Systems

Semester 1 2022/2023

Dr. Ayman Khallel

Content

- Process
- Threads
- Multithreading
- Locks
- Synchronization
- Deadlock
- Summary

What is a Process?

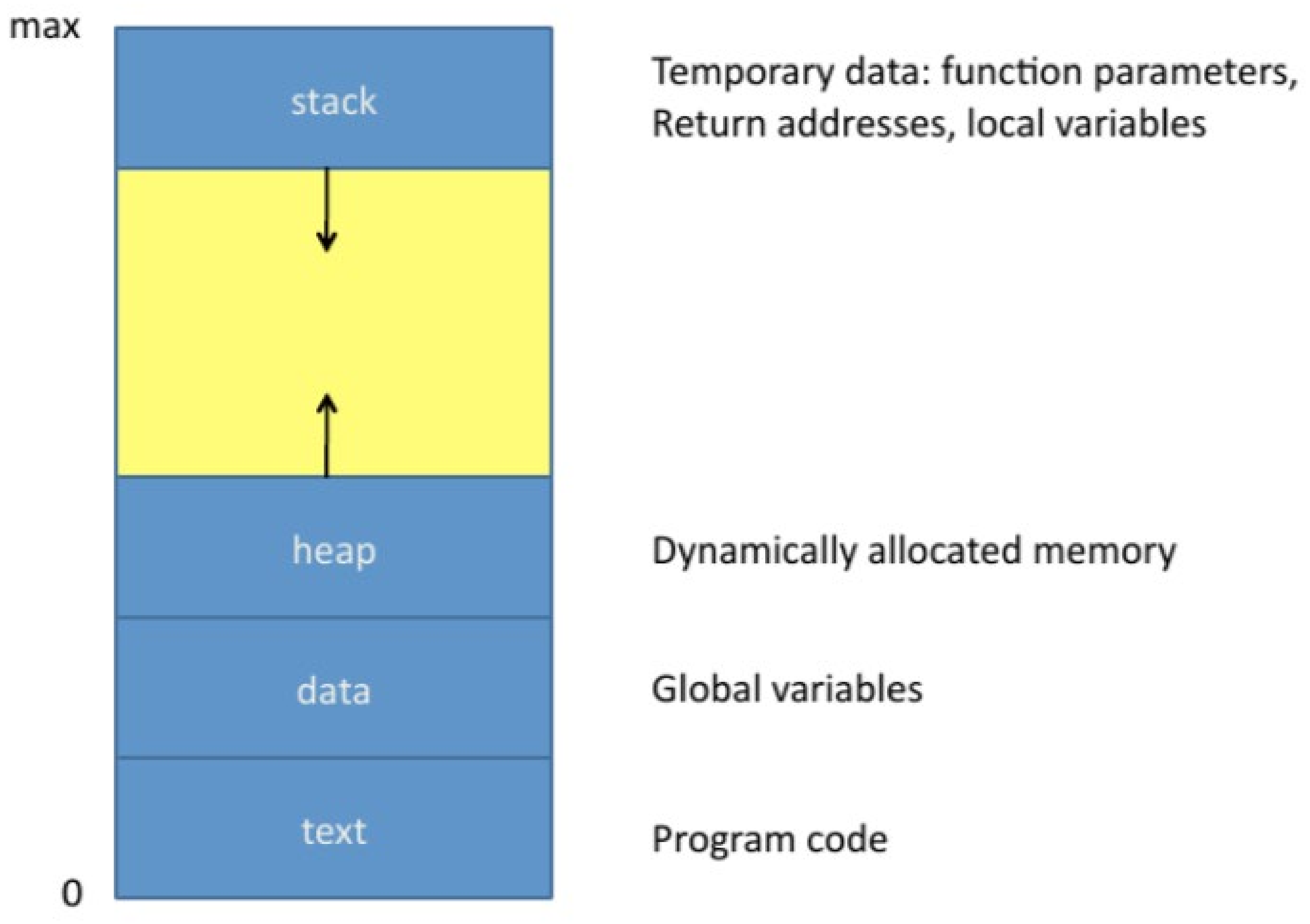
- Traditionally, a process is considered an instance of a computer program that is being executed.
- A process contains
  - System resources: program code, user data, buffers, devices, I/O channels, files
  - Current activity: CPU, registers, state, execution path, “On the clock”, interleaved with other processes
- Can resources and CPU activity be treated independently
  - Unit of resource ownership $\rightarrow$ process or task
  - Unit of execution $\rightarrow$ thread or lightweight process

Processes (Heavy)

- Resources owned by a process:
  - code ("text"),
  - data (VM),
  - stack,
  - heap,
  - files,
  - tables (signals, semaphores, buffers, I/O,…)
- Context switching processes has a significant amount of overhead:
  - Tables have to be flushed from the processor when context switching.
  - Processes share information only through pipes and shared memory (IPC).

Process

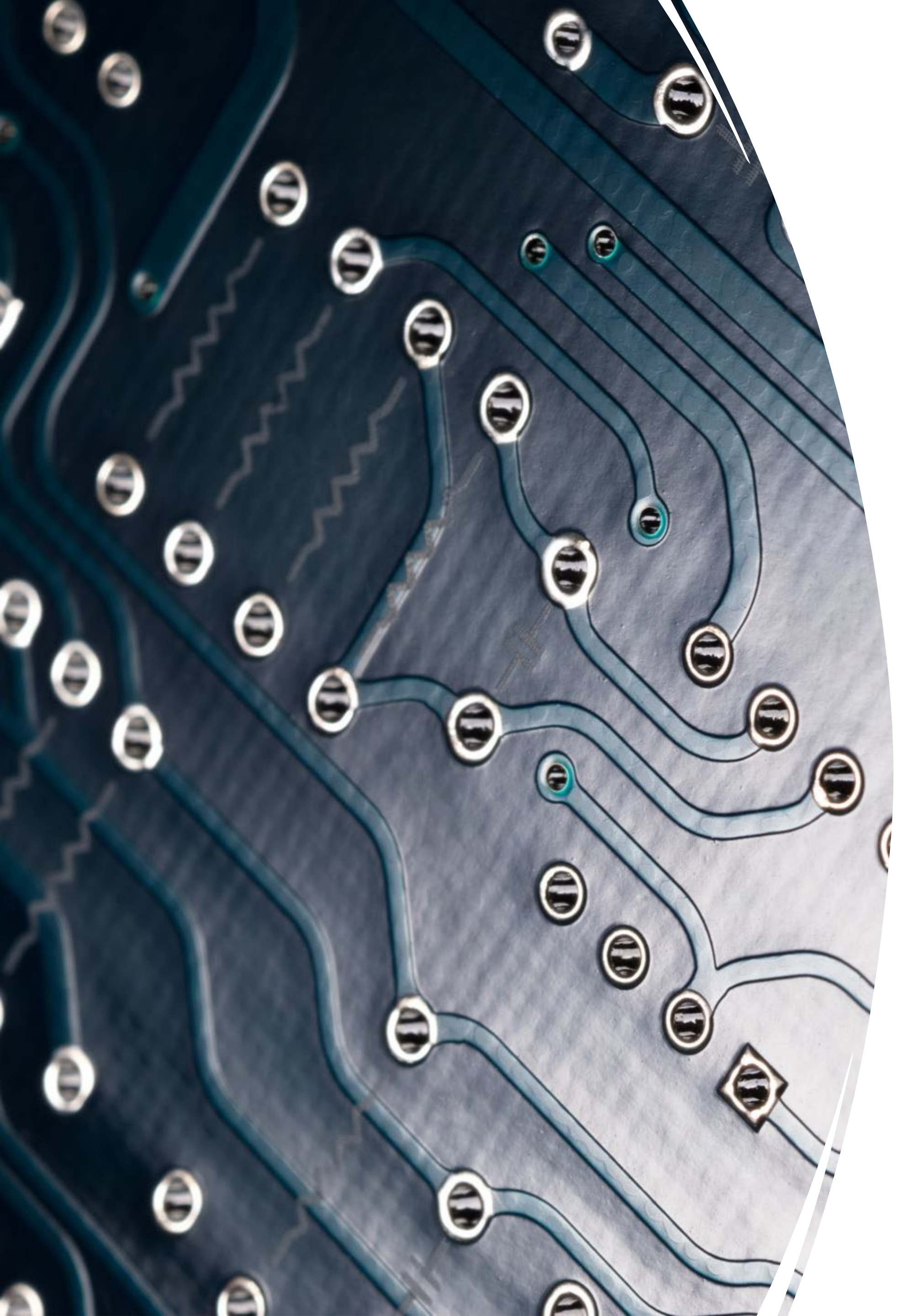
- Associated address space
  - Program itself (text section)
  - Program's data (data section)
  - Stack, heap



max
stack
heap
data
text
0
Temporary data: function parameters, Return addresses, local variables
Dynamically allocated memory
Global variables
Program code

What is a Thread?

- A thread of execution
  - Smallest unit of processing (context) that can be scheduled by an operating system
- Threads reduce overhead by sharing the resources of a process.
  - Switching can happen more frequently and efficiently.
  - Sharing information is not so "difficult" anymore - everything can be shared.
- A Thread is an independent program counter and stack operating within a process.
  - Sometimes called a lightweight process (LWP)
  - A smaller execution unit than a process.

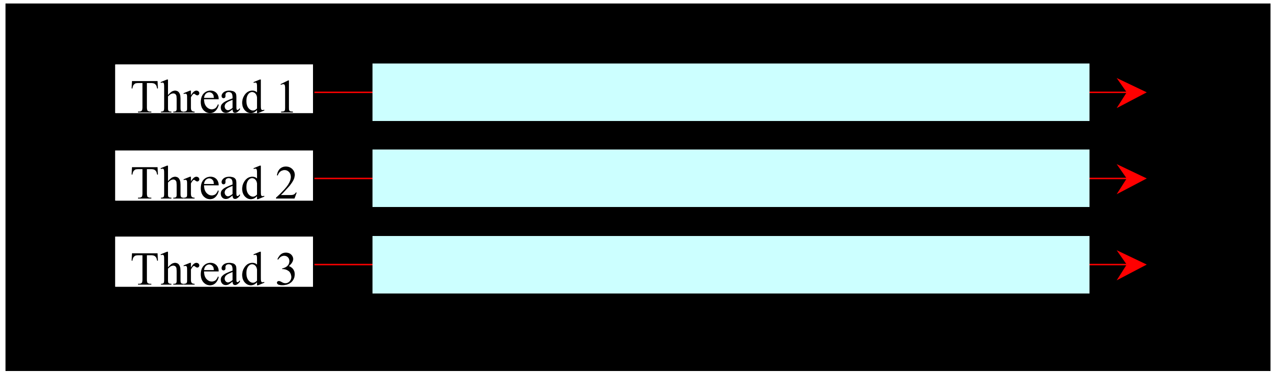


Thread concepts

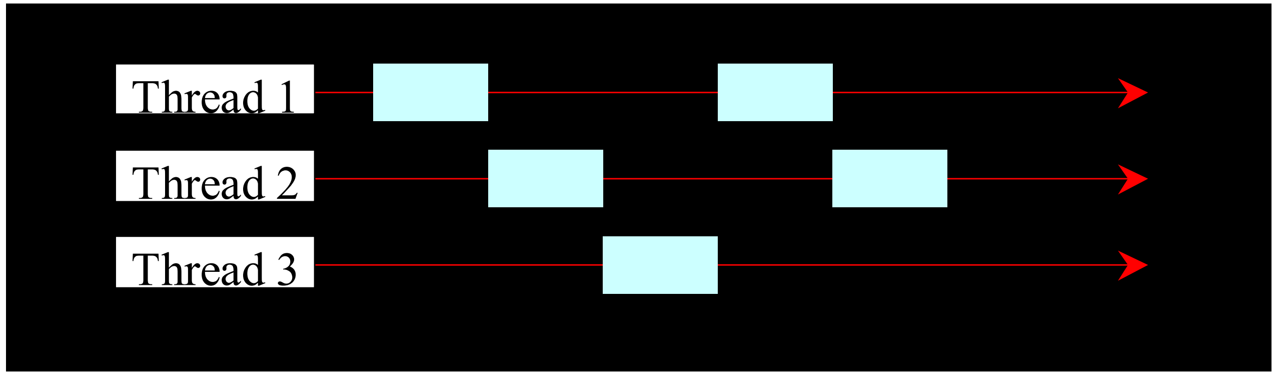
- Basic unit of CPU utilization
  - Flow of control within a process
- A thread includes
  - Thread ID Program counter
  - Register set
  - Stack
- Shares resources with other threads within the same process
  - Text section
  - Data section
  - Other OS resources (open files, ...)

Thread concepts

Multiple threads on multiple CPUs



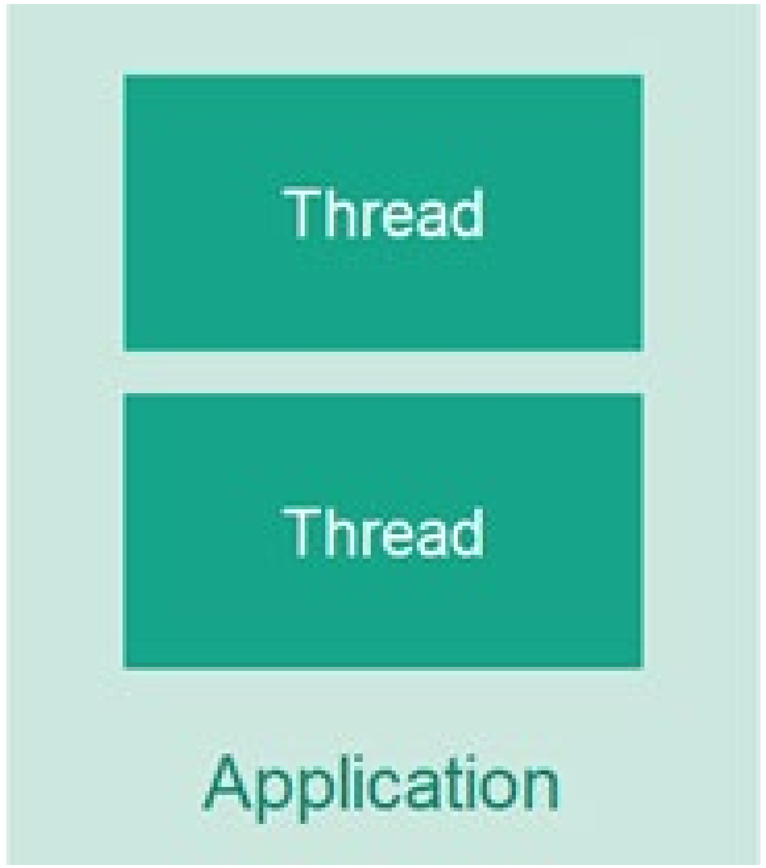
Multiple threads sharing a single CPU



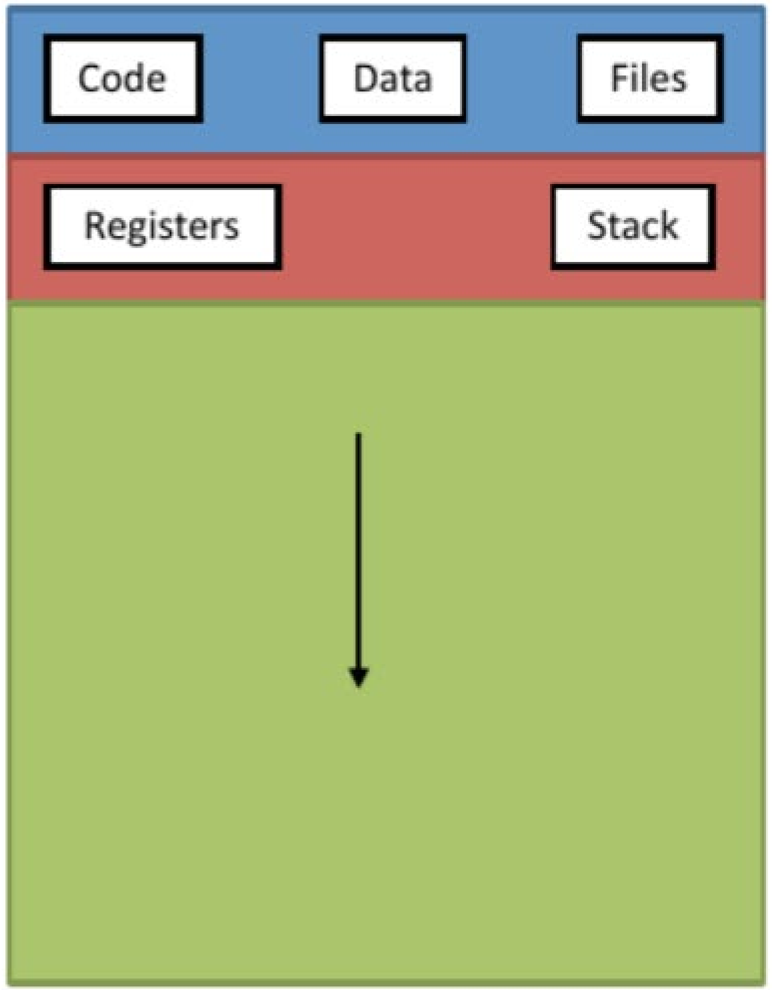
*Source: Liang, Introduction to Java Programming, Seventh Edition

Multithreading

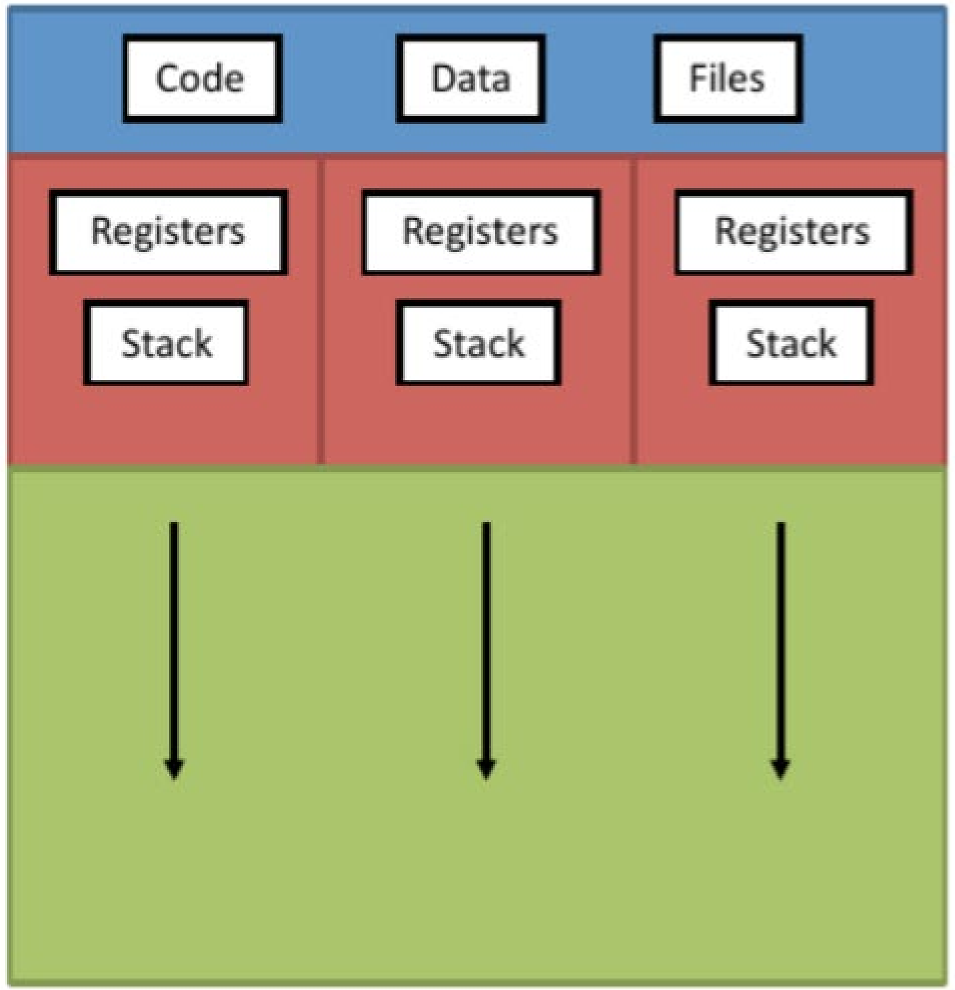
- Multiple **threads** of execution inside the same application.
  - A thread is like a separate CPU executing your application.
- Thus, a multithreaded application is like an application that has multiple CPUs executing different parts of the code at the same time.
- Multithreading is a technique that allows for concurrent (simultaneous) execution of two or more parts of a program for maximum utilization of a CPU
  - basic example - multithreading allows you to write code in one program and listen to music in another



Single-threaded vs multi-threaded



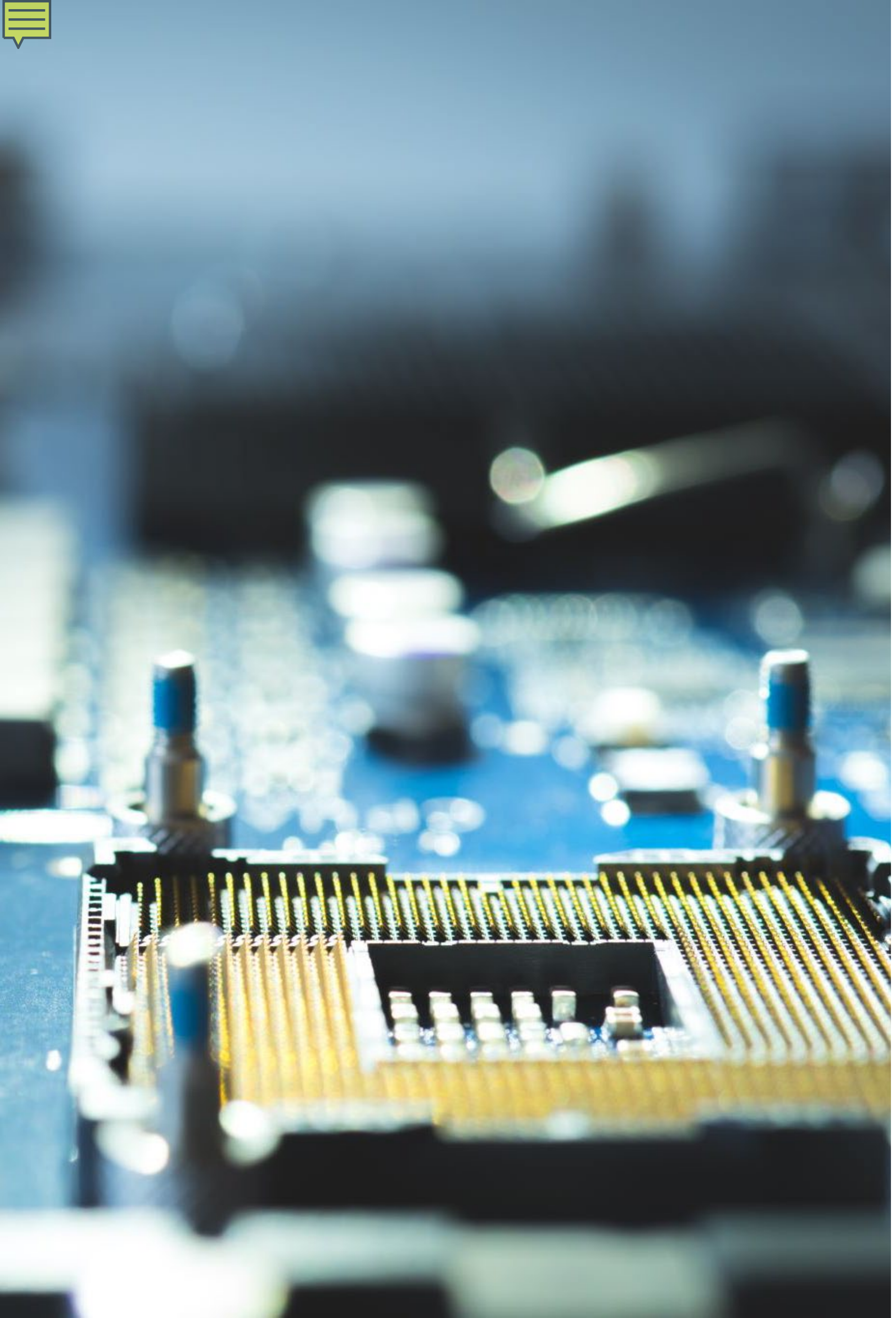
Single-threaded



Multi-threaded

Multithreading example

- Think about a single processor running your programming language IDE
    - You are executing the code that process (read) a large image to be (recognized) using some deep learning algorithm
    - This process is expensive and may took lots of memory and time dealing with the IO
    - At this time, the UI may freeze if there is no thread. However, with IO is still in process, IO thread switched out and give to UI that the IDE and the computer is still responsive and not frozen.
    - This is multitasking in concept – where each thread allocated with a time to be executed on CPU – and switching is very fast (new multicore machine nowadays)



Why multithreading?

- Multiple threads in a single core would have to switch back and forth to give the illusion of multitasking – not efficient
- In multiple cores – multithreading has become extremely important in terms of the efficiency of your application
- common reasons for multithreading are
  - Better utilization of a single CPU.
  - Better utilization of multiple CPUs or CPU cores.
  - Better user experience with regards to responsiveness.
  - Better user experience with regards to fairness.

Cost of Multithreading

- Performance overhead:
  - Synchronization
  - Access to shared resources
- Hazards:
  - **Deadlocks**: A thread enters a waiting state for a resource held by another one, which in turn is waiting for a resource by another (possible the first one).
  - **Race conditions**: Two or more threads read/write shared data and the result depends on the actual sequence of execution of the threads.
- Not deterministic: Harder to debug (hard to reproduce errors)

Synchronization

- Concurrent access to shared data can result in data inconsistency.
- To ensure orderly execution of processes, the operating system provides mechanisms for job synchronization and communication.
- A deadlock state is a state of indefinite wait by one or more processes for an event that can be triggered only by one of the waiting processes.
- Operating system also provides mechanisms to ensure that jobs do not get stuck in a deadlock, forever waiting for each other.

Synchronization - Example

- Consider two threads t1 and t2:
  - Thread t1 increments the counter
    - counter := counter + 1
  - Thread t2 decrements the counter
    - counter := counter – 1

  If both are run one after the other, the final value of the counter (counter could be a memory location for example) should remain the same regardless of the order in which they are run. One increments it, the other decrements it. No change.

The increment and decrement statements of t1 and t2 translate to machine code as shown on the following slide.

Synchronization - Example

t1 (increment)

1. reg_1 := counter
2. reg_1 := reg_1 +1
3. counter := reg_1

t2 (decrement)

4. reg_2 := counter
5. reg_2 := reg_2 −1
6. counter := reg_2

Synchronization - Example

- Consider the situation in which the statements are run in the following order: 1, 2, 3, 4, 5, 6.
- The corresponding sequence of code thus, becomes:

1. reg_1 := counter
2. reg_1 := reg_1 + 1
3. counter := reg_1
4. reg_2 := counter
5. reg_2 := reg_2 – 1
6. counter := reg_2

Counter =5

1. reg_1 := 5
2. reg_1 := 5 + 1
3. counter := 6
4. reg_2 := 6
5. reg_2 := 6 – 1
6. counter := 5

Synchronization - Example

- Now consider the two threads run concurrently. Consider the situation in which the statements are run in the following order: 1, 2, 4, 3, 5, 6.
- The corresponding sequence of code thus, becomes:

1. reg_1 := counter
2. reg_1 := reg_1 + 1
4. reg_2 := counter
3. counter := reg_1
5. reg_2 := reg_2 – 1
6. counter := reg_2

$\xrightarrow{\text{Counter =5}}$

1. reg_1 := 5
2. reg_1 := 5 + 1
4. reg_2 := 5
3. counter := 6
5. reg_2 := 5 – 1
6. counter := 4

Synchronization - An Example

- Thus, we see from the preceding sequence of code that the counter will not be unchanged upon completion but will be rather decremented by 1.
- The above stated problem arose because both sections of code constitute critical sections and these critical sections interleaved. The critical sections must be run atomically.

Critical Section and Race Condition

- A Critical Section is a part of a thread that accesses shared resources. Two threads should not be allowed to enter their critical sections at the same time, thus preventing the above problem.
- The situation where several threads access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**.

Thread Synchronization

More on Critical Section

Important points to note about critical sections:

- A critical section must be run atomically.
  - This means that the section is executed either as a whole or not at all. Once the critical section of a process begins, it must be completed or rolled back.
- Mutual exclusion of more than one critical section must be ensured
  - Mutual exclusion is ensured by using pieces of code to block a thread that attempts to run its critical section while the critical section of another thread is being executed. The thread is unblocked when the other thread's critical section completes execution. This is known as Sleep and Wakeup.

Synchronization with lock

- When two or more operations belonging to concurrent threads try to access the shared memory, a race condition can occur.
- The easiest way to get around the race conditions is the use of a lock.
- The operation of a lock is simple when a thread wants to access a portion of shared memory, it must necessarily acquire a lock on that portion prior to using it.
- After completing its operation, the thread must release the lock that was previously obtained.
- The impossibility of incurring races is critical as the need of the lock for the thread.

Advantages of lock

- It allows us to restrict the access of a shared resource to a single thread or a single type of thread at a time
- Before accessing the shared resource of the program, the thread must acquire the lock and must then allow any other threads access to the same resource.
- There are two methods that are used to manipulate the locks: acquire() and release()

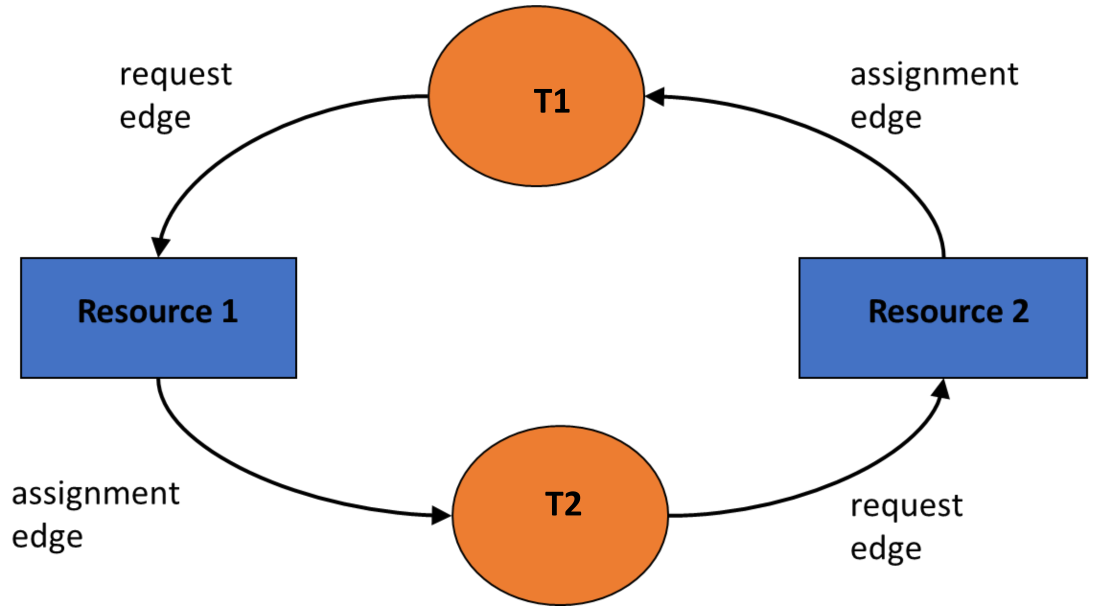
Concurrently means happening at the same time. In the context of computing, concurrency refers to the ability of a system to support multiple threads or processes that can be executed simultaneously. This can be useful for improving the performance and responsiveness of a system by allowing different tasks to be performed in parallel, rather than sequentially.

Disadvantages of lock

- Subject to harmful situations of deadlock
- Limits the scalability of the code and its readability
- The use of a lock is in conflict with the possible need to impose the priority of access to the memory shared by the various processes.
- Difficulties find errors.

Deadlock

- A deadlock occurs due to the acquisition of a lock from different threads
- It is impossible to proceed with the execution of operations
- Let's think of a situation wherein there are two concurrent threads



Deadlocks
The four necessary conditions

The four necessary conditions for a deadlock to occur are:

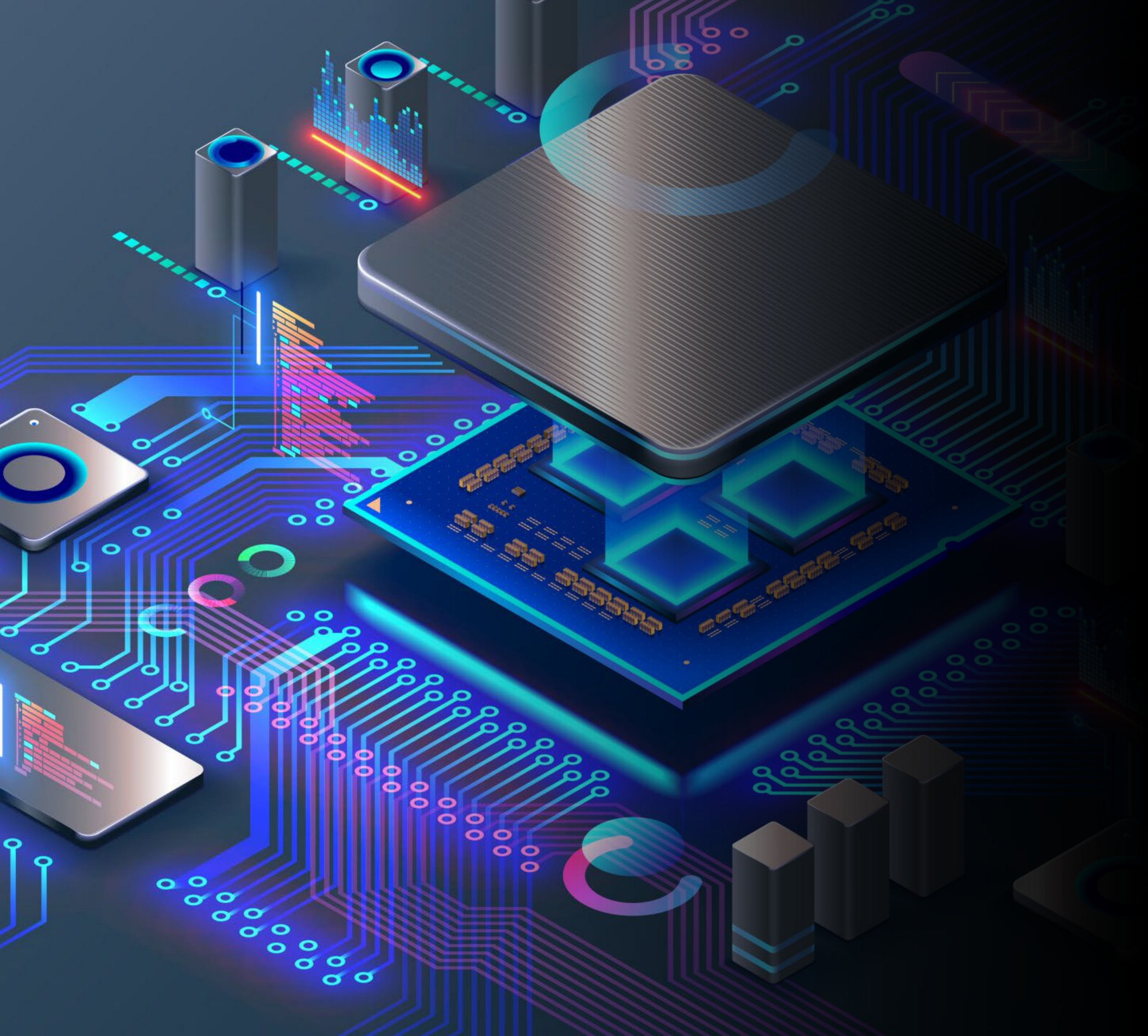
- Mutual exclusion of resources
    - Inability of a resource to be used by more than one thread
- Hold-and-wait
    - A thread holds a resource while waiting for another one
- No pre-emption
    - The system is incapable of grabbing a resource from a thread
- Circular Wait

Deadlock Handling
Four methods for handling deadlocks

- The four method for dealing with the deadlock problem are:
  - Prevention
  - Detection and Recovery
  - Avoidance
  - Ignoring the condition
- Ignoring deadlocks is by far the most widely used method.

References

- Madhavan Nagarajan. Concurrency vs. Parallelism — A brief view.
- http://tutorials.jenkov.com/java-concurrency/concurrency-vs-parallelism.html
- https://www.educative.io/blog/multithreading-and-concurrency-fundamentals
- Umakishore Ramachandran and William D. Leahy Jr., Multithreaded Programming and Multiprocessors
- Prof. Paolo Bientinesi and Diego Fabregat-Traver. Parallel Programming Processes and Threads.
- https://www.tutorialspoint.com/java/java_thread_deadlock.htm



- Thank You
- Next: Functional Parallelism and Concurrency