Cairo University
Faculty of Computers and Artificial Intelligence
Computer Science Department

Concepts of Programming Languages

Concurrency

Dr. Amin Allam

[For more details, refer to "Concepts of Programming Languages" by *Robert Sebesta*]

- *Concurrency* is the simultaneous execution of two or more: *instructions*, *statements*, *subprograms*, or *programs*. In particular, *subprograms concurrency* affects the design of the programming languages which support them. There are two *concurrency* categories:

• *Physical concurrency:* Several subprograms literally execute simultaneously on different physical processors.

• *Logical concurrency:* The programmer and the application software assume that there are multiple processors, but the actual execution of subprograms may take place in interleaved fashion on a single processor.

Multiprocessor architectures fall into the following categories:

• *Single-Instruction Multiple-Data (SIMD):* Multiple processors execute the same instruction simultaneously, each on different data. One processor controls the operation of the other processors.

• *Multiple-Instruction Multiple-Data (MIMD):* Multiple processors operate independently but their operations can be *synchronized*. Each processor executes its own instruction stream.

- Memory can be *shared* among processors or *distributed* such that each processor has its own local memory. Some systems contain both memory configurations.

- A *task* (*process*) is a program unit that can be in concurrent execution with other program units. *Synchronization* is a mechanism that controls the order in which *tasks* execute. There are two *synchronization* types:

• *Cooperation synchronization:* is required when a *task* A must wait for another *task* B to complete some activity before *task* A can continue its execution.

• *Competition synchronization:* is required when two *tasks* require the use of the same resource that cannot be simultaneously used.

- Alternative answers to *concurrency* issues are: *semaphores*, *monitors*, and *message passing*:

• A *semaphore* is an implementation of a *guard* which is a linguistic wrapper around a guarded code to be executed only when a specific condition is true. The guarded code usually accesses a resource (such as a data structure) whose access needs to be controlled. A *semaphore* consists of an integer which stores the number of current *tasks* accessing the resource, and a queue that stores descriptors of *tasks* which need to access the resource but did not gain access yet.

• A *monitor* is logically similar to a *semaphore*, except that *synchronization* responsibilities is transferred to an abstract data type which represents the guarded resource.

• In *message passing*, a *task* can suspend its execution at some point, then announces that it is ready to receive *messages*, then waits for a *message* from another *task* to continue its execution.