**Georgia Southern University**

**Allen E. Paulson College of Engineering and Computing**

**Information Technology Department**

**Dr. Atef Shalan**

**Course: IT2530-Operating Systems**

**Module 3-** CPU scheduling and Process Synchronization– reading material

Contents and Reading Material

[**2** Module 2-CPU scheduling and Process Synchronization 1](#_Toc95638249)

[2.1 CPU Scheduling 1](#_Toc95638250)

[2.1.1 Uniprocessor Scheduling 2](#_Toc95638251)

[2.1.2 Multiprocessor scheduling 4](#_Toc95638252)

[2.2 Process Synchronization and deadlocks 5](#_Toc95638253)

[2.2.1 Key Definitions 6](#_Toc95638254)

[2.2.2 Process Interaction relationships 7](#_Toc95638255)

[2.2.3 Message Passing 8](#_Toc95638256)

# Module 2-CPU scheduling and Process Synchronization

Processor scheduling aims to assign processes to be executed by the processor or processors over time, in a way that meets system objectives, such as response time, throughput, and processor efficiency. In many systems, the CPU scheduling activity is broken down into three separate functions: long-, medium-, and short-term scheduling. The names suggest the relative time scales with which these functions are performed. When a computer system contains more than a single processor, several new issues are introduced into the design of the scheduling function. We begin with a brief overview of multiprocessors and then look at the rather different considerations when scheduling is done at the process level and the thread level. As we build bigger systems, we will see how we have to synchronize them.  In the OS, processes have to be synchronized as well. Process synchronization describes the cooperation and interaction relationships among processes during execution. In this part, we learn about the ideal conditions and the possible problems with process synchronization. We also learn how processes cooperate and interact by emphasizing message passing methodology. Part 1 in this module will discuss CPU scheduling and part 2 discusses Process synchronization.

## CPU Scheduling

CPU Scheduling mechanism depends on the scheduling environment. There are two processing environments for CPU scheduling: uniprocessor and multiprocessor scheduling. [Uniprocessor scheduling](https://spin.atomicobject.com/2015/02/26/uniprocessor-scheduling-policies/) focuses on single processor multiprogramming and multitasking systems by multiplexing several processes on a single processor through a timesharing mechanism. Uniprocessor scheduling then supports concurrent execution where multiple processes make progress at the same time. In this execution model, each process alternates between using a processor (running and ready states) and waiting for some event to occur (blocked state), e.g., I/O operation. One of the goals of uniprocessor scheduling is to make the highest utilization of the processor and system resources. [Multiprocessor and multicore scheduling](https://www.javatpoint.com/multiple-processors-scheduling-in-operating-system) discuss scheduling processes in multiprocessor/multicore environments. Thus, this type of scheduling is concerned with the structure/design of the multiprocessor system, The main concerns of multiprocessor and multicore scheduling are the synchronization granularity, loose or tightly processor coupling, and the distribution and clustering of processors. The following section discusses uniprocessor scheduling and the next section will cover the scheduling in multiprocessor environments.

### Uniprocessor Scheduling

Multiprogramming and multitasking systems on uniprocessor environments aim to assign processes to be executed by the processor while satisfying some system objectives, e.g., response time, throughput, and resource efficiency. Uniprocessor scheduling is managed in operating systems at three different stages using three separate functions:

* Long-term scheduling: this function adds processes to the pool of processes ready to execute.
* Medium-term scheduling:add to the pool of processes in the main memory and ready to run on the processor.
* Short term scheduling: make a process to take its turn to run on the processor

#### Uniprocessor scheduling and process state transitions

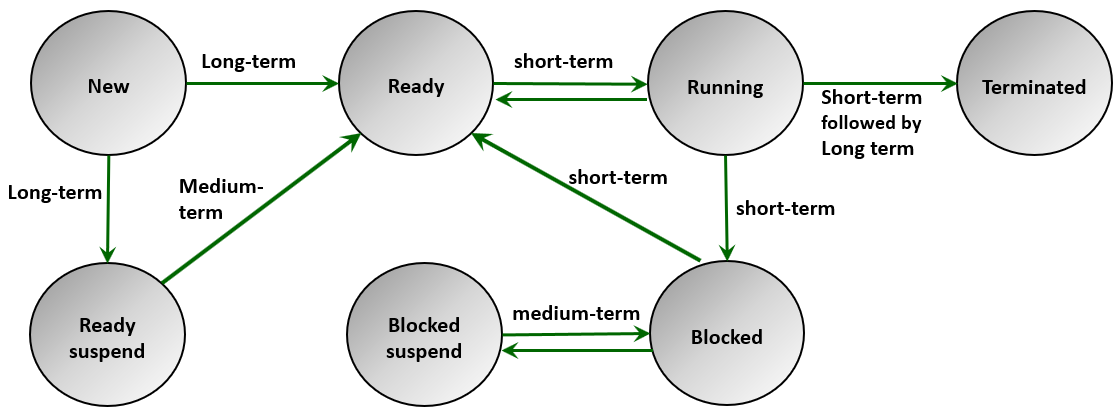


Figure 1-Uniprocessor scheduling and process transitions.

Figure 1- describes the process state transitions as parts of the three uniprocessor scheduling functions. From the figure, we notice that new processes are scheduled first through long-term scheduler works with the new processes, it takes a process to the ready state or ready suspended. We see also that any state transitions that swap a process in or out of memory are taken care of by the medium-term scheduler (swapper). Dispatching processes, timing them out, blocking is controlled by the short-term scheduler.

Therefore, the long-term scheduler decides which programs are admitted to the system for processing and controls the degree of system multiprogramming and the number of time slices allowed for every process to execute. The long scheduler takes also into consideration the process execution request time, running processes, available and allocated resources, processor utilization, process priority, expected execution time, I/O requirements, system security, access rights, and other factors.

Medium-term scheduling is part of the memory management component of Operating systems. The medium-term scheduler is called a swapper and it is concerned with swapping processes in and out between the main memory and the virtual memory. Therefore, the swapper decision depends on the system’s capability to manage the multiprogramming and the memory requirements of the swapped in or out processes.

The short-term scheduler is known as a process dispatcher. It is a small-size process that is always running in the system and that is responsible for multiplexing multiple processes to execute on the system CPUs. Relative to the other scheduler, the dispatcher work most frequently. The short-term scheduler is also invoked when the events may lead to process blocking. Examples of these events are clock interrupt I/O operations, and system calls.

#### CPU Scheduling Criteria and policy

In this part, we focus on the different criteria and algorithms considered by the CPU scheduling to decide which process to be scheduled next. The main objective of CPU scheduling is to allocate processor time to optimize certain aspects of system behavior. [CPU scheduling criteria](https://www.geeksforgeeks.org/cpu-scheduling-criteria/) can be categorized into two types:

* **Process-based criteria**
  + Factors that are focused on user satisfaction with the system and that are aimed to improve the system behavior as perceived by processes
  + **Examples:**
    - **Response time**: the time taken for the process to start responding to the user. I. e., the time interval between execution request submission and the receival of the first response.
    - **Turnaround time**: the time interval between execution request submission and the process execution completion.
    - **Waiting time**: the time spent by a process waiting in the ready queue.
    - **Predictability**: a measure that indicates consistency with the expected run time and resource usage regardless of the system load.
* **System-based criteria**
  + Factors that are focused on the utilization of the system resources to serve the user and system processes efficiently.
  + **Examples:**
    - **Throughput:** the number of completed processes per unit of time.
    - **Processor utilization**: the percentage of time the processor is not idle.
    - **Fairness**: equal treatment processes and assurance of starvation avoidance.
    - **Enforcing priorities**: higher priority processes run first.
    - **Balancing resources**: fairness of distributing system resource utilization.

Different [CPU scheduling algorithms](https://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm) are used by operating systems to maintain the aforementioned criteria based on the computing environments and quality of service requirements. To understand these algorithms, we first need to distinguish between two different modes of scheduling: **non-preemptive and preemptive.** In non-preemptive scheduling algorithms, once a process starts to run (enters the running state), it cannot be preempted until it completes its allotted time. In preemptive scheduling, a scheduler may preempt a low priority running process when a high priority process becomes ready (i.e., enters a ready state). Scheduling algorithms are listed below.

* **First Come First Serve (FCFS)**
  + Processes are executed on a first-come, first-serve basis.
  + Both are non-preemptive and preemptive.
  + Implementation as a FIFO queue.
  + Average wait time can become high with this algorithm
  + Check [the example in TutorialPoint](https://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm).
* **Shortest Process time Next (SPN)**
  + The process with the shortest expected processing time is scheduled next.
  + Both non-preemptive and pre-emptive.
  + The processer should anticipate the processing time in advance.
  + Check [the example in TutorialPoint](https://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm).
* **Priority-based scheduling:** 
  + The process with the highest priority is scheduled first, otherwise, first come first serve.
  + Common in batch systems.
  + Non-preemptive algorithm
  + Check [the example in TutorialPoint](https://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm).
* **Shortest Remaining Time (SRT)**
  + A preemptive version of the SPN.
  + Processes closest to completion are scheduled and processes can be preempted by a newer ready job with a shorter completion time.
  + Check [the example in TutorialPoint](https://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm).
* **Round Robin (take turns)**
  + A preemptive process scheduling based on a clock.
  + Each process is provided a time slice (called a quantum) to execute, and then be preempted, and another process executes for another time slice, and so on.
  + Check [the example in TutorialPoint](https://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm).

### Multiprocessor scheduling

[Multiprocess scheduling](https://www.javatpoint.com/multiple-processors-scheduling-in-operating-system) allows various processes to run simultaneously over more than one processor or core. The complexity of multiprocessor scheduling as compared to single processor scheduling is mainly caused by the requirements to balance the execution load over multiple processors. These processers can be tightly coupled (multicore systems), i.e., they are capable of communicating through a common bus, memory, I/O channels. They also can be loosely coupled (e.g., multiprocessor systems, distributed multiprocessor, and clustered systems), where each processor is having its main memory and I/O channels. The processors can be also identical (symmetric multiprocessing), or heterogeneous (asymmetric multiprocessing). The design issues of multiprocessor scheduling include some basic concepts: processor affinity, load balancing, synchronization granularity.

[Processor affinity](https://www.javatpoint.com/multiple-processors-scheduling-in-operating-system) is a design concept that associates processes with the processors to run on. Based on the processor affinity concept, the system tends to avoid migrating processes to other processors different from where they are executed. The process affinity concept aims to mitigate the overheads of invalidating process data in the first processor (where the process ran first) cache and repopulating them again on the new processor cache. There are two types of processor affinity: Soft and hard. In soft affinity, the operating systems choose but do not guarantee to keep a process running on the same processor (*dynamic scheduling*). In hard affinity, a process can specify a subset of processors on which it may run (*dedicated processor assignment*).

[Load Balancing](https://www.javatpoint.com/multiple-processors-scheduling-in-operating-system) is a design concept that distributes the workload evenly across all processors. In general, systems immediately extract a runnable process from the common run queue once a processor becomes idle. If a processor has its private queue of processes eligible to execute, based on the processor affinity concept, the load balancing concept will be necessary to assure equal distribution of execution load. There are two approaches for load balancing: push and pull migration. In *push migration*, a system process routinely checks the load on each processor and moves processes from overloaded to idle or less busy processors. In *pull migration*, an idle processor pulls a waiting task from a busy processor and executes it.

[Synchronization granularity](https://en.wikipedia.org/wiki/Granularity_(parallel_computing)) is a measure that indicates the frequency of communication between multiple process threads in a multithreading system relative to the execution time of these threads. Depending on the synchronization granularity, executing a multithreaded process on multiple processors (i.e., parallelism) can be classified into three categories: fine-grained, medium-grained, and coarse-grained parallelism.

* **Fine-Grained Multithreading**: The switching between threads may take place at an instruction level. The system may execute up to 20 instructions before it needs to communicate with other threads.
* **Medium-Grained Multithreading**: thread execution on a processor takes a longer time (~ between 20 - 2000 instructions) before communicating with other threads.
* **Coarse-Grained Multithreading**: thread execution on a processor takes a longer time (~ >2000 instructions) before communicating with other threads.

## Process Synchronization and deadlocks

Processes running on a system or a network of connected systems may cooperate to achieve some goals and may need to share some common repository. These processes may also need to share some common resources or compete on their usage. If a group of processes shares some common resource, Then their order of execution is a very critical factor and any change in this order may result in incorrect or inappropriate system outputs. [Process synchronization](https://en.wikipedia.org/wiki/Synchronization_(computer_science)) is an operating systems procedure that preserves the order of execution of a group of cooperating processes by joining them up (i.e., handshaking) at specific points of execution. In the following subsections, we overview some key definitions related to process synchronization and then we discuss the methodology for process communication.

### Key Definitions

This part introduces the key terms related to process synchronization and concurrent execution.

* [Atomic operation](https://www.techopedia.com/definition/3466/atomic-operation#:~:text=Explains%20Atomic%20Operation-,What%20Does%20Atomic%20Operation%20Mean%3F,systems%20and%20parallel%20processing%20systems.): a program operation or action implemented as a sequence of one or more instructions either executed entirely or not executed at all.
* [Critical section](https://www.tutorialspoint.com/critical-section-problem): a part of process code that if executed by more than one process at the same time, may cause inappropriate or incorrect output.
* [Deadlock](https://www.geeksforgeeks.org/deadlock-starvation-and-livelock/): a situation where some processes or threads are blocked because each process is holding a resource and waiting for another resource held by some other process.
* [Livelock](https://www.geeksforgeeks.org/deadlock-starvation-and-livelock/): a situation where a set of processes or threads keep changing their states in response to changes in the other processes while not doing any real work.
* [Starvation](https://www.geeksforgeeks.org/deadlock-starvation-and-livelock/): a situation where a process or a thread can proceed but it never gets its turn to run by the system scheduler.
* [Race condition](https://en.wikipedia.org/wiki/Race_condition): a situation where multiple processes or threads access some shared data with read and write operations and the final result depends on the timing of their execution.
* [Mutual exclusion](https://www.definitions.net/definition/mutual+exclusion): a requirement for executing a critical code section that if one process is in that section, no other process can be executing this section at the same time.

See the [code examples in geeksforgeeks](https://www.geeksforgeeks.org/deadlock-starvation-and-livelock/) for extended learning about the above key terms.

The following is a code example that illustrates the race condition situation. If the producer is faster than the consumer, the producer will overload the buffer, and if the consumer is faster than the producer, the consumer will come to a situation where it draws from an empty buffer.

|  |  |
| --- | --- |
| Producer code | Consumer code |
| // The producer cannot overload the buffer  //Process: producer  while (true) {  buffer[in] = next\_produced;  in = (in + 1) % BUFFER\_SIZE;  counter++; //produce item in next produced  } | // The consumer cannot draw from empty buffer  //Process: consumer  while (true) {  next\_consumed = buffer[out];  out = (out + 1) % BUFFER\_SIZE;  counter--; //consume the item in next consumed  } |

In the following version, we add a dummy while loop that makes the producer wait if the buffer is full, and makes the consumer waits if the buffer is empty.

|  |  |
| --- | --- |
| Producer code | Consumer code |
| // The producer cannot overload the buffer  //Process: producer  while (true) {  while (counter == BUFFER\_SIZE) ; //do nothing  buffer[in] = next\_produced;  in = (in + 1) % BUFFER\_SIZE;  counter++; //produce item in next produced  } | // The consumer cannot draw from empty buffer  //Process: consumer  while (true) {  while (counter == 0) ; // do nothing  next\_consumed = buffer[out];  out = (out + 1) % BUFFER\_SIZE;  counter--; //consume the item in next consumed  } |

### Process Cooperation and competition

In operating systems, executing processes share data, system I/O resources, and microprocessor computing. They also participate in the whole system behavior and convenience. The interaction relationships between a set of the process can be either competition or cooperation.

[Process competition](https://en.wikipedia.org/wiki/Deadlock) is a relationship among processes sharing some data or system resource and are not aware of each other. Therefore, these processes do not share any data and do not communicate, but they only run on the same system and share some of the resources. Depending on the system scope: computing device, local area network, wide area network, etc, competing processes can exist on a single computer system and can be distributed over remote devices. Competing processes can influence each other by delaying their execution and consequently, affecting the result timing of these processes. Resource sharing by competition may result in deadlocks and starvation of some processes. Mutual exclusion of the shared resources is an important access requirement to avoid race conditions and incorrect results.

If the processes are indirectly aware of each other by sharing some data or directly aware of each other by communicating, we call these them [cooperating processes](https://www.tutorialspoint.com/cooperating-process). Cooperation by sharing gives processes a degree of awareness through their effects on the shared objects. Therefore, processes cooperating by sharing can affect each other’s results and timing. Cooperation by sharing may also result in deadlock and starvation of processes. Also, mutual execution is an important resource access requirement to avoid race conditions. Processes communication allow processes to exchange message and synchronize their execution and avoid race conditions and the need for mutual execution requirements. However, cooperation by communication still can affect each other’s results and timing and may result in deadlock and starvation of processes.

### Message Passing

Message passing is the main communication technique among cooperating processes. Message passing allows processes to send/receive data and to synchronize their execution. The message passing function is provided by using two main primitives: **send** and **receive** functions. The communication among processes also requires a communication channel or link: e.g., shared memory, hardware bus, and network. The logical attributes of the communication among processes include the following

* **Synchronization:** blocking or non-blocking message passing
  + Blocking message passing is synchronous communication in which, the sender is blocked until the message is received (blocking send) or the receiver is blocked until a message is available (blocking receive), or both blocking send and receive (rendezvous message passing).
  + Non-Blocking message passing is asynchronous communication in which the sender sends a message and continues executing and the receiver can receive either a message or none, and then continues its execution.
* **Addressing**: Direct or indirect.
  + In the direct addressing, the message send primitive must specify the destination process and the receiving process must identify or only accept messages from an explicitly designated sender.
  + In the indirect addressing, messages are sent and received to/from a structure data store: e.g., message queue: FIFO, and mailbox: one to one, one to many, many to one, and many to many.
* **Buffering:** limited, or unlimited buffer capacity.
* Limited buffer can be zero capacity (i.e., the sender and the receiver must come to a rendezvous), it also can be bounded capacity where limited buffer for a fixed number of messages are allowed. If the buffer is full, the sender must wait until a free buffer is available.
* The unlimited capacity allows an infinite buffer for senders and receivers.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_