# Concurrency and Deadlocks

The core of operating system designs focuses on the management of processes and threads, such as the following (Stallings, 2018):

- **Multiprogramming** – It is the management of multiple processes within a uniprocessor system.
- **Multiprocessing** - It is the management of multiple processes within a multiprocessor.
- **Distributed processing** – It is the management of multiple processes executing on multiple distributed computer systems.

The basic requirement to support concurrent processes is the ability of a program to enforce **mutual exclusion**, which pertains to the ability to execute all other processes from a course of action while one (1) process is granted that ability. Any facility or capability that is to provide support for mutual exclusion should meet the following requirements (Stallings, 2018):

1. Only one process at a time is allowed into its critical section, among all processes that have critical sections for the same resource or shared object.
2. A process that halts in its noncritical section must do so without interfering with other processes.
3. No deadlock of starvation. Processes requiring access to a critical section must not be delayed indefinitely.
4. When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.
5. No assumptions must be made about the relative process speeds or number of processors.
6. A process shall remain inside the critical section for a finite time only.

## Principles of Concurrency

**Concurrency** is an application performance technique that encompasses the ability to load and execute multiple runnable programs. Each of these programs may be an application process that does not necessarily execute on the central processing unit (CPU) at the same instance; even their runtimes may overlap (Gregg, 2021).

A concurrent system supports more than one (1) task by allowing all the tasks to make progress. Concurrency involves an array of design issues, including process communication, accessing and sharing of resources, synchronization of activities of multiple processes, and allocation of processor time to different processes. Concurrency can be viewed based on the following contexts:

1. Multiple applications
2. Structured applications
3. Operating system structure

Below are some important terminologies related to concurrency:

- **Atomic operation** – It is a function or an action implemented as a sequence of one (1) or more instructions that appear to be indivisible, wherein no other process can see an intermediate state or interrupt the operation. In this operation, the sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on the system state. Note that the concept of **atomicity** guarantees isolation from concurrent processes.
- **Critical section** – It is a section of code within a process that requires access to shared resources, and must not be executed while other process is in a corresponding section of code.
- **Race condition** – It is a situation in which multiple threads or processes read and write a shared data item, and the final result depends on the relative timing of their execution.
- **Starvation** – It is a situation in which a runnable process is overlooked indefinitely by the scheduler. Although the process is able to proceed, it is never chosen.

In a single-processor multiprogramming system, processes are interleaved in time to yield the appearance of simultaneous execution. Even actual parallel processing is not achieved, and there is an overhead in switching back and forth between processes, interleaved execution provides a major benefit in processing efficiency and program structuring.

It is possible to classify how processes interact based on the degree of their awareness about the other existing processes (Stallings, 2018).

- **Competition: Processes unaware of each other** – This comprises independent processes that are not intended to work together, which can either be batch jobs, interactive sessions, or a mixture of both. Some of the potential control problems in this process interaction set-up are mutual exclusion, deadlock, and starvation. The best example of this situation is the multiprogramming of multiple independent processes.
- **Cooperation by sharing: Processes indirectly aware of each other** – This involves processes that are not necessarily aware of each other by their respective process identifications but shares access to some objects. The result of one (1) process may depend on the information obtained

from the other processes. Some of the potential control problems in this process interaction set-up are mutual exclusion, deadlock, starvation, and data coherence.

- **Cooperation by communication: Processes directly aware of each other** – This encompasses processes that are able to communicate with each other by process identification and that are designed to work jointly on some activity. Some of the potential control problems in this process interaction set-up are deadlock and starvation.

Below are some operating system (OS) concerns that are raised by the existence of concurrency (Stallings, 2018):

1. The OS must be able to keep track of the various processes through the utilization of process control blocks.
2. The OS must allocate and deallocate different resources for each active process.
3. The OS must protect the data and physical resources of each process against unintended interference by other processes.
4. The operation of a process, and the output it produces, must be independent of the speed at which its execution is carried out relative to the speed of other concurrent processes.

## Common Concurrency Mechanisms (Stallings, 2018)

The first major advancement in dealing with concurrency-related problems came in around 1965 with Dijkstra's written formal works. Dijkstra's work focuses on operating system (OS) design as a collection of cooperating sequential processes, with the development of an efficient and reliable mechanism for supporting cooperation.

Below are the common OS and programming language mechanisms that are used to provide or support concurrency:

- **Counting Semaphore** – This involves an integer value that is used for signaling processes. Only three (3) operations may be performed on a semaphore, all of which are atomic: *initialize*, *decrement*, and *increment*. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. This concurrency mechanism is also known as the *general semaphore*.
- **Binary Semaphore** – This is a semaphore that only takes the values zero (0) and one (1).
- **Mutual Exclusion (Mutex) Lock** – This mechanism is similar to a binary semaphore. The key difference between the two is that the process that locks the mutex must be the one to unlock it, and only the holder of the lock can operate.
- **Condition Variable** – This is a data type that is used to block a process or a thread until a specific condition is true.
- **Monitor** – This is a programming construct that encapsulates variables, access procedures, and initialization code within an abstract data type. It is easier to control and has been implemented in a number of programming languages such as Java and Pascal-Plus. The monitor's variable may only be accessed via its access procedures, and that only one (1) process can actively access the monitor at any given time.
- **Event Flag** – It is a memory word used as a synchronization mechanism. A specific application code is associated with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one (1) or multiple bits in the corresponding flag.
- **Mailbox or Message Passing** – This mechanism is considered as a means for two (2) processes to exchange information, and that may be used for process synchronization.
- **Spinlock** – This is a mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability.

## Principles of Deadlocks (Stallings, 2018)

**Deadlocks** can be defined as permanent blocking of a set of processes that either compete for system resources or communicate with each other. A set of processes is deadlocked when each process in the set is blocked waiting for an event, typically freeing up some requested resource that can only be triggered by another blocked process in the set. Note that all deadlocks involve conflicting needs for resources by two (2) or more processes.

The following are the general resource categories:

- **Reusable resources** – These resources can be used by only one process at a time and are not depleted by usage. As an example of a deadlock involving reusable resources, consider two (2) programs that compete for exclusive access to a disk file D and a tape drive T. Deadlock occurs if each process holds one (1) resource and requests the other.
- **Consumable resources** – These are resources that can be created (produced) and destroyed (consumed). An unblocked producing process may create any number of resources. Then, when a resource is acquired by a consuming process, the resource ceases to exist. As an example of a deadlock involving consumable resources, consider a pair of processes in which each process attempts to receive a message from the other

process, then sends a message to the other. Deadlock occurs if the receiving process is blocked until the message is received.

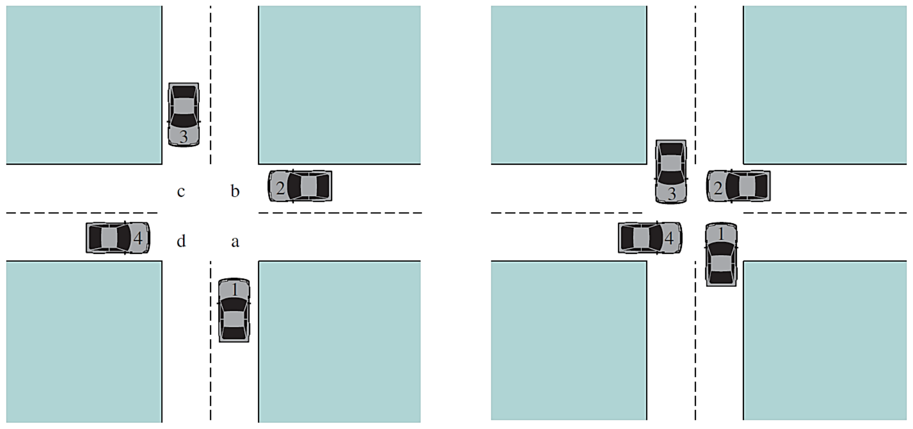
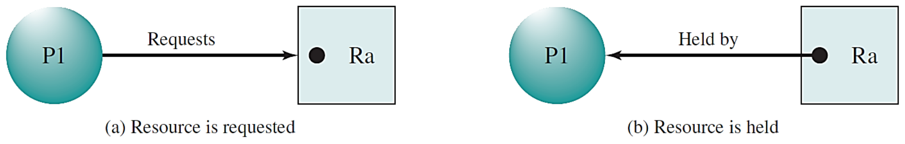


***Figure 1***. Example of a deadlock.
*Source*: Operating Systems: Internal and Design Principles (9th ed.), 2018 p. 291

The **resource allocation graph**, which was introduced by *Richard Holt*, is a useful tool in characterizing the allocation of resources to processes. It is a directed graph that depicts the state of system resource processes, wherein processes and resources are represented by nodes connected by edges. A resource allocation graph can be described as follows:

- An edge directed from a process to a resource indicates a resource that has been requested by the process but not yet granted.
- A dot within the resource node represents an instance of a resource.
- An edge directed from a reusable resource node dot to a process indicates a request that has been granted, and one (1) unit of the resource has been assigned to the process.
- An edge directed from a consumable resource node dot to a process indicates the process is the procedure of that resource.



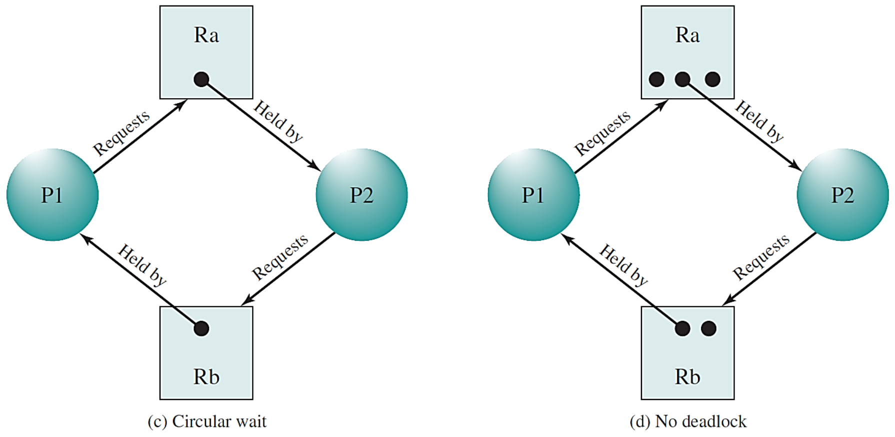


***Figure 2***. Examples of resource allocation graph.
*Source*: Operating Systems: Internal and Design Principles (9th ed.), 2018 p. 297

## Deadlock Prevention, Avoidance, and Detection

The following conditions must be present for a deadlock to occur (Silberschatz, Galvin & Gagne, 2018):

1. **Mutual exclusion**: At least one resource must be held in a non-sharable mode. This means that only one (1) thread at a time can use the resource. If another thread requests that specific resource, the requesting thread must be delayed until the resource has been released.
2. **Hold and wait**: A thread must be holding at least one (1) resource and waiting to acquire additional resources that are currently being held by other threads.
3. **No preemption**: Resources cannot be preempted. This means that a resource can only be released voluntarily by the thread holding it after that thread has completed its task.
4. **Circular wait**: A closed chain of threads exists, such that each thread holds at least one resource needed by the next thread in the chain.

The first three (3) conditions are necessary, but not sufficient, for a deadlock to occur. The fourth condition is required for a deadlock to actually take place. Technically, the fourth condition is the potential consequence of the first three conditions. Deadlock can also be described as an unresolved circular wait.

**Deadlock Prevention: Disallows one of the four conditions for deadlock occurrence** – This strategy of involves the designing of a system in such a way that the possibility of deadlock is excluded (Stallings, 2018).

A. **Indirect method of deadlock prevention** (preventing the first three conditions)
    - *Mutual exclusion*: In general, this condition cannot be disallowed. If access to a resource requires mutual execution, then mutual exclusion must be supported by the operating system.
    - *Hold and wait*: This condition can be prevented by requiring a process to request all of its required resources at once and blocking the process until all resources can be granted simultaneously.
    - *No preemption*: This condition can be prevented through the following ways:
        - If a process is denied of further request, that process must release the resources that it is currently holding;
        - If necessary, request the process again with the additional resources; and
        - Let go of other resources in order to proceed with other process execution.

B. **Direct method of deadlock prevention** (preventing the occurrence of the fourth condition)
    - *Circular wait*: This condition can be prevented by defining a linear ordering of resource types.

**Deadlock Avoidance: Do not grant a resource request if the allocation might lead to a deadlock condition** – This strategy allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached. Thus, allowing more concurrency. With deadlock avoidance, decisions are made dynamically. Hence, knowledge of future process resource requests must be known. Two (2) approaches of deadlock avoidance are as follows (Stallings, 2018):

A. **Process initiation denial**: Do not start a process if its demands might lead to a deadlock; and
B. **Resource allocation denial**: Do not grant an incremental resource request to a process if this allocation might lead to a deadlock.

Below are some restrictions in implementing the deadlock avoidance strategy:

- The maximum resource requirement for each process must be stated in advance;
- The processes under consideration must be unconstrained by any synchronization requirements;
- There must be a fixed number of resources to allocate; and
- No process may exit while holding resources.

**Deadlock Detection: Grant resource requests when possible, but periodically check for deadlock and act to recover** – This strategy does not limit resource access or restricts process executions. Requested resources are granted to processes whenever possible. Then, the operating system periodically executes an algorithm that detects the presence of a circular wait condition. Once the deadlock has been detected, any of the following recovery methods can be performed, whichever is more appropriate for the program (Stallings, 2018):

a. Aborting all deadlock processes is the most common solution in operating systems.
b. Back up each deadlocked process to some previously defined checkpoint, and restart all processes. This requires that the rollback and restart mechanisms are built into the system. However, the original deadlock may recur.
c. Successively abort deadlocked processes until deadlock no longer exists. After each abortion, the detection algorithm must be reinvoked to see whether a deadlock still exists.
d. Successively preempt resources until deadlock no longer exists. A process that encompasses preempted resource must be rolled back to the point prior to its resource acquisition.

The selection criteria in successively aborting deadlocked processes or preempt resources could be one of the following:

- Process with the least amount of processor time consumed so far
- Process with least amount of output produced so far
- Process with the most estimated remaining time
- Process with the least total of resources allocated so far
- Process with the lowest priority

**References:**
Gregg, B. (2021). *System performance: Enterprise and cloud* (2nd ed.). Pearson Education, Inc.
Silberschatz, A., Galvin, P. & Gagne, G. (2018). *Operating systems concepts* (10th ed.). John Wiley & Sons, Inc.
Stallings, W. (2018). *Operating systems: Internal and design principles* (9th ed.). Pearson Education Limited