DEADLOCKS

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; **if the resources** are busy, the process enters a **waiting state**. Sometimes, a waiting process is never again able to change state, because the resources it has requested are **held** by other waiting processes. This situation is called a **deadlock**.

**7.1 System Model**

* A system consists of a **finite** number of resources to be distributed among a number of competing processes.
* The resources may be partitioned into several types (or **classes**).
* Each class consisting of some number of identical **instances**. CPU cycles, files, and I/0 devices (such as printers and DVD drives) are examples of resource types.

If a process requests an **instance** of a resource type, the **allocation** of any instance of the type should **satisfy** the request or it’s **not identical**.

* Mutex locks and semaphores are also considered system resources, and they are a **common source of deadlock**.

**Under the normal mode of operation, a process may utilize a resource in only the following sequence:**

1. **Request**. The process requests the resource. If the request **cannot be granted** immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

2. **Use**. The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).

3. **Release**. The process releases the resource.

* The request and release of resources may be system calls.
* Examples are the **request ()** and **release ()** device, **open ()** and **close ()** file, and **allocate ()** and **free ()** memory system calls.
* **Wait ()** and **signal ()** operations on semaphores or through **acquire ()** and **release ()** of a mutex lock.

**A system table** records whether each resource is free or allocated.

For each resource that is allocated, the table **also** records the process to which it is allocated.

A **set of processes is in a deadlocked state** when every process in the set is waiting for an event that can be caused only by another process in the set.

The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example, semaphores, mutex locks, and files).

**7.2 Deadlock Characterization**

7.2.1 Necessary Conditions

**A deadlock situation** can arise if the following **four conditions** hold simultaneously in a system:

1. **Mutual exclusion**. At least one resource must be held in a **nonsharable** mode; that is, only **one** **process** at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait**. A process must be **holding at least one resource** and waiting to acquire additional resources that are currently being held by other processes.
3. **No preemption**. Resources **cannot be preempted**; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. **Circular wait.** A set {P0, P1, ... , P,} of **waiting processes must exist** such that Po is waiting for a resource held by P,, P1 is waiting for a resource held by P2, ... , P,\_, is waiting for a resource held by P,, and Pn is waiting for a resource held by Po.

* All four conditions must hold for a deadlock to occur.

7.2.2 Resource-Allocation Graph

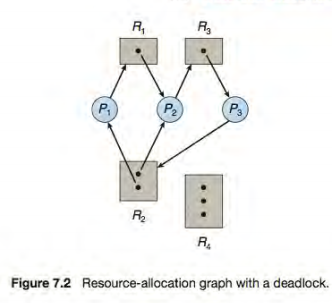
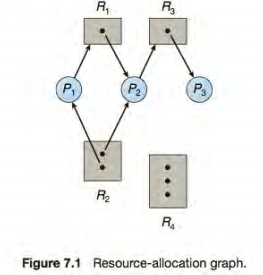
Deadlocks can be described by a **system resource-allocation graph.**

This graph consists of a set of vertices V and a set of edges E.

* The set of vertices V is partitioned into two different types of nodes:
* P = {P1, P2,…, Pn}, the set of active processes in the system.
* R = {R1, R2,... Rm}, the set consisting of all resource types in the system.
* **Request edge:** A directed edge from process Pi to resource type Rj is denoted by Pi 🡪 Rj.
* **Assignment edge**: A directed edge from resource type Rj to process Pi is denoted by Rj 🡪Pi. It signifies that an instance of resource type RJ has been **allocated to process** Pi.
* We represent each **process** Pi as a **circle** and each **resource** type Rj as a **rectangle**.
* We represent each such **instance** as a **dot** within the rectangle.

Given the definition of a resource-allocation graph, it can be shown that,

* If the graph contains no cycles, then no process in the system is deadlocked.
* If the graph does contain a cycle, then a deadlock may exist.
* Deadlock occurs:
* If the cycle has only a set of resource types, each of which **has only a single instance**. Each process involved in the cycle is deadlocked.
* In this case, a cycle in the graph is both a **necessary** and a **sufficient** condition for the existence of deadlock.

**7.3 Methods for Handling Deadlocks**

* We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
* We can allow the system to enter a deadlocked state, detect it, and recover.
* We can ignore the problem altogether and pretend that deadlocks never occur in the system.

The third solution is the one used by most operating systems, including Linux and Windows.

**To ensure that deadlocks never occur**, the system can use either **a deadlock prevention** or a **deadlock-avoidance scheme.**

**Deadlock prevention** provides a set of methods to ensure that at least one of the necessary conditions (Section 7.2.1) cannot hold.

**Deadlock avoidance** requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime.

* If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may arise.

**7.4 Deadlock Prevention**

By ensuring that **at least one of these conditions cannot hold**, we can prevent the occurrence of a deadlock.

7.4.1 Mutual Exclusion

* The mutual exclusion condition must **hold**. That is, **at least one resource must be non-sharable.**
* **Read-only files** are a good example of a sharable resource.
* **Sharable resources** do not require mutually exclusive access and thus cannot be involved in a deadlock.
* We cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsicallyبجوهرها nonsharable.
* For example, a **mutex** lock cannot be simultaneously **shared** by several processes.

7 .4.2 Hold and Wait

**To ensure** that the hold-and-wait condition never occurs, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.

* **One protocol that we can use** requires each process to request and be **allocated all its resources** before it begins execution.
* **An alternative protocol** allows a process to request resources only **when it has none.**
* Before it can request any additional resources, it must release all the resources that it is currently allocated.

Both these protocols have **two main disadvantages.**

* **First**, **resource utilization** may be **low**, since resources may be allocated but unused for a long period.
* **Second**, **starvation** is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

7.4.3 No Preemption

To ensure that this condition **does not hold**, we can use the following protocol.

1-If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait),

2-Then all resources the process is currently holding are preempted (implicitly released).

3-The preempted resources are added to the list of resources for which the process is waiting.

4-The process will be restarted only when it can regain(تستعيد) its old resources, as well as the new ones that it is requesting.

7 .4.4 Circular Wait

One way to ensure that this condition never holds is **to impose a total ordering** of all resource types and to require that **each process requests resources in an increasing order of enumeration**.

* **1-** That is, a process can **initially** request any number of instances of a resource type -say, Ri. **After that,** the process can request instances of resource type Rj if **and only if F (Rj) > F(Ri).**
* We define a one-to-one function F: R🡪 N, where N is the set of natural numbers.
* For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:
* F(tape drive) = 1 F(disk drive)= 5 F(printer) = 12
* **A process that wants to use the** tape drive and printer at the same time must first request the tape drive and then request the printer.
* **2-** **Alternatively**, we can require that a process requesting an instance of resource **type Rj** must have released any resources **Ri** **such that F(Ri) F(Rj).** Note also that if several instances of the same resource type are needed, a single request for all of them must be issued.
* **If these two protocols** are used, then the circular-wait condition **cannot hold.**

More info look p321

**7.5 Deadlock Avoidance**

Deadlock-prevention algorithms, **prevent deadlocks by limiting how requests can be made.**

The limits ensure that **at least one** of the necessary conditions for deadlock cannot occur.

* Possible side effects are **low device utilization** and **reduced system throughput**.

**An alternative method for avoiding deadlocks** is to require additional information about how resources are to be requested.

* Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.
* The various algorithms that use this approach differ in the amount and type of information required.
* The simplest and most useful model requires that each process declare the **maximum number** of resources of each type that it may need.
* A deadlock avoidance algorithm **examine** the resource-Allocation state.
* **The resource-allocation state** is defined by the number of available and allocated resources and the maximum demands of the processes.

7.5.1 Safe State

A state is **safe** if the system can allocate resources to each process (**up to its maximum**) in some order and still avoid a deadlock.

* More formally, a system is in a safe state only if there exists a safe sequence.
* A sequence of processes <P1, P2, ... , Pn> is a **safe sequence** for the current allocation state if, for each Pi the resource requests that Pi can still make can be satisfied by the currently available resources plus the resources held by all Pi, with **j < i**.
* When Pi terminates, Pi+1 can obtain its needed resources, and so on.
* If no such sequence exists, then the system state is said to be **unsafe**.
* A deadlocked state is an unsafe state.
* As long as the state is **safe**, the operating system can **avoid** unsafe (and deadlocked) states.

The idea is simply to ensure that the system will always remain in a safe state.

* **Initially, the system is in a safe state.**
* Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait.

7 .5.2 Resource-Allocation-Graph Algorithm

In addition to the request and assignment edges already described, we introduce a new type of edge, called a **claim edge**.

* **Claim edge:** indicates that process Pi may request resource Rj at some time in the future.
* This edge resembles a request edge in direction but is represented in the graph by a **dashed line**.
* When process Pi **requests** resource Rj, **the claim edge** Pi 🡪 Rj is **converted** to a **request edge**. Similarly, when a resource Rj is **released** by Pi, the **assignment edge** Rj 🡪 Pi is **reconverted** to a **claim edge** Pi🡪Rj.
* Note that the resources must be claimed a **priori** in the system.
* That is, before process Pt starts executing, all its claim edges must already appear in the resource-allocation graph.

7.5.3 Banker's Algorithm

Please look at it in the textbook p326

**7.6 Deadlock Detection**

If a system **does not** employ either a deadlock-prevention or a deadlock avoidance algorithm, **then** a deadlock situation may occur. In this environment, the system may provide:

• An algorithm that examines the state of the system to determine whether a deadlock has occurred

• An algorithm to recover from the deadlock.

**7.6.1 Single Instance of Each Resource Type**

If all resources have only a **single instance**, then we can define a **deadlock detection algorithm** that uses a variant of the resource-allocation graph, called **a wait-for graph.**

* We obtain this graph from the resource-allocation graph **by removing the resource nodes and collapsing the appropriate edges.**
* **As before, a deadlock** exists in the system if and only if the wait-for graph contains **a cycle.**
* To detect deadlocks, the system needs to maintain the graph and periodically invoke an algorithm that searches for a cycle in the graph.
* An algorithm requires an order of **n2** operations, where **n** is the number of vertices in the graph.

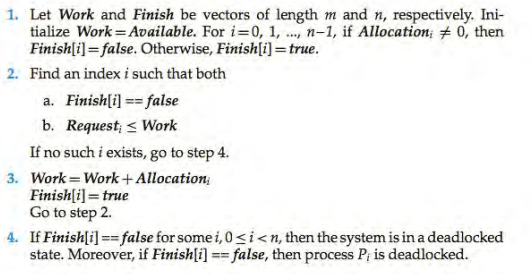
**7.6.2 Several Instances of a Resource Type**

The wait-for graph scheme is not applicable with multiple instances of each resource type.

We turn to **deadlock-detection algorithm** that is applicable to such a system.

The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm:

* Available. A vector of length m indicates the number of available resources of each type.
* Allocation. An **n x m** matrix defines the **number of resources** of each type currently allocated to each process.
* Request. An **n** x **m** matrix indicates the **current request** of each process.
* If Request[i](j] equals k, then process Pi is requesting k more instances of resource type Rj.



For more info please check out textbook p331-332

**7.6.3 Detection-Algorithm Usage**

When should we invoke the detection algorithm? The answer depends on two factors:

1. How often is a deadlock likely to occur?

2. How many processes will be affected by deadlock when it happens?

* If deadlocks occur **frequently**, then the **detection algorithm should be invoked frequently**.
* Resources allocated to deadlocked processes will be **idle** until the deadlock can be broken.
* Deadlocks **occur only** when some process makes a request that **cannot** be granted **immediately**.
* Then, we can invoke the deadlock detection algorithm every time a request for allocation cannot be granted immediately.
* In this case, we can identify not only **the deadlocked set of processes** but also the specific **process that "caused" the deadlock**.
* Of course, invoking the deadlock-detection algorithm for every resource request will incur **considerable overhead** in computation time.
* A less expensive **alternative** is simply to invoke the algorithm at **defined interval**.

**7. 7 Recovery from Deadlock**

* **One possibility** is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
* **Another possibility** is to let the system recover from the deadlock automatically.

**There are two options for breaking a deadlock:** aborting processes –preempting resources.

**7.7.1 Process Termination**

To eliminate deadlocks by aborting a process, we use one of two methods.

In both methods, the system reclaims all resources allocated to the terminated processes.

* **Abort all deadlocked processes:** **break the deadlock cycle, but at great expense.** The deadlocked processes will lose its partial results of these partial computations (If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state.)
* **Abort one process at a time until the deadlock cycle is eliminated.** This method incurs considerable overhead, **since after each process is aborted, a deadlock-detection algorithm must be invoked** to determine whether any processes are still deadlocked.
* Using this method: we must determine which deadlocked process (or processes) should be terminated.
* **Many factors may affect which process is chosen, including**:
* 1. What the priority of the process is
* 2. How long the process has computed and how much longer the process will compute before completing its designated task
* 3. How many and what types of resources the process has used (for example, whether the resources are simple to preempt)
* 4. How many more resources the process needs in order to complete
* 5. How many processes will need to be terminated
* 6. Whether the process is interactive or batch

**7.7.2 Resource Preemption**

To eliminate deadlocks using resource preemption, we successively **preempt** some resources from processes **and give** these resources to other processes until the deadlock cycle is broken.

Three issues need to be addressed:

1. **Selecting a victim.** Which resources and which processes are to be preempted? Determine the order of preemption to minimize cost.

2. **Rollback**. If we preempt a resource from a process, what should be done with that process? We must roll back the process to some safe state and restart it from that state.

In general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it.

3. **Starvation**. , how can we guarantee that resources will not always be preempted from the same process? We must ensure that a process can be picked as a victim only a (small) finite number of times.

**The most common solution** is to include the number of rollbacks in the cost factor.