Deadlock

System Model

- **Computer systems** consist of **resources** (for example CPU, memory, and I/O devices).
- We also consider tools like **locks, and semaphores** to be **resources**.
- Each resource of type R_i has W_i instacnes.
- When a **process** utilizes a resource the following happens:
  1. The process **requests** the resource.
  2. The process **uses** the resource.
  3. The process **releases** the resource.

  - Processes can request as many resources as they need, it is not restricted. However, the number of resources requested cannot exceed the amount of resources available.
  - Processes may need to wait for resources to become available before their request is granted.
- The operating system uses a **system table** to record the status of **resources** (unallocated / allocated). If a resource is allocated, the process / thread allocating it is also recorded.

Deadlock

- A set of **threads** are said to be **in a deadlocked state** when **every thred** in the set is **waiting for an event** that can be caused only by **another thread in the set**.
- The following is an example of how deadlock can occur:
  1. Process one acquires a lock on resource 1.
  2. Process two acquires a lock on resource 2.
  3. Process wants to acquire a lock on resource two, but must block until it is available.
  4. Process two wants to acquire a lock on resource one, but must block until it is available.
  5. Both processes have entered deadlock.
- A set of **threads** are said to be **in a livelocked state** when **every thread** in the set is **waiting for an event** that can be caused only by **another thread in the set**. Livelock occurs when a the thread that the others are waiting for **continuously attempts an action that fails**.

Deadlock Conditions

- **Deadlock** can arise if **four conditions** occur at the same time:
  1. **Mutual exclusion**: Only one process at a time an use the resource.
  2. **Hold and wait**: A process holding at least one resource is waiting to acquire additional resources held by other processes.
  3. **No preemption**: A resource can be released only voluntarily by the process holding it.
  4. **Circular wait**: There exists a set of waiting processes such that P_0 is waiting for P_1, and P_1 is waiting for P_2 and ... and P_n is waiting for P_0.
- These conditions do **not guarantee deadlock** will occur, they can **potentially cause deadlock**.

Resource-Allocation Graphs

- A **resource-allocation graph** is a **directed graph**, $G = (V, E)$.
- V is partitioned into two sets:
  1. $P = \{P_1, P_2, ..., P_n\}$ consisting of all of the processes in the system.
  2. $R = \{R_1, R_2, ..., R_m\}$ consisting of all of the resource types in the system.
- E consists of two types of edges:
  1. A **request edge** is a directed edge of the form $P_i \rightarrow R_j$.
  2. An **assignment edge** is a directed edge of the form $R_j \rightarrow P_i$.
- If G has a **cycle**, **deadlock** may exist.
- If G has **no cycles**, **deadlock** does not exist.

Methods for Handling Deadlock

- One way to handle deadlocks is to ensure the system will never enter a deadlocked state (prevention / avoidence).
- Another way to handle deadlocks is to allow the system to enter a deadlocked state, and then recover.
- The last way is to ignore the problem, and pretend that deadlocks never occur in the system (this option is used by most systems).

Deadlock Prevention

- To prevent **deadlock**, we must **restrain** the ways **requests can be made**.
- To prevent deadlock, we must prevent at-least one of the conditions required for deadlock. We can do the following:
  1. **Mutual Exclusion** — Only one thread can request a resource at any given time.
     - Not realistic, this would put locks on shared resources which is inefficient.
  2. **Hold and Wait** — Must guarantee that whenever a thread requests a resource, it does not hold any other resources. Either we can have the thread request all of it's resources at the start of execution, or only when it does not hold any other resources.
     - Not realistic, it is inefficient, and often difficult to predict the exact amount of resources required.
  3. **No Preemption** — If a thread is holding some resources, and requests another resource that cannot be immediately allocated, then all resources currently being held are released. The process is then restarted when the resoruces it requested become available.
     - Not reasitic, this cannot be used for resources like semaphores, and mutexes.
  4. **Circular Wait** — Impose a total ordering of all resource types, and require that each thread requests resources in an increasing order of enumeration.
     - This is realistic, all we have to enforce is the order threads allocate resources.

Deadlock Avoidance

- To avoid **deadlocks**, we can give the operating system **information** about the **resources a thread intends to request**.
- The simplest and most useful model requires that **each process** declares the **maximum number** of **resource**s of each type that **it may need**.
- The **deadlock-avoidance algorithm** dynamically examines the **resource-allocation state** to ensure that **circular-wait conditions do not happen**.
- **Resource-allocation state** is defined by the **number of available and allocated resources**, as well as the **maximum demands of the processes**.

Safe State

- When a **process requests** an **available resource**, the system must decided if **immediate allocation** leaves the system in a **safe state**.
- This happens as follows:
  1. If P_i resource needs are not immediately available, the P_i can wait until all of the resources of P_{i-1} are deallocatd.
  2. When they are available, P_i receives them, performs it's required tasks, and deallocates them.
  3. When P_i deallocates the resources, P_{i+1} can allocate them.
- If the system is in a **safe state**, there are no deadlocks.
- If the system is in an **unsafe state**, there is a possibility that deadlocks can occur.
- **Avoidance** ensures that a system **never enteres an unsafe state.**

Deadlock-Avoidance Algorithms

- There are two main **deadlock-avoidance algorithms**:
  1. **Resource-Allocation Graphs** — Use when resources have a single instance.
  2. **Banker's Algorithm** — Used when resources have multiple instances.

Deadlock Recovery — Process Termination

- One way to **recover from deadlock** is to **abort all deadlocked processes**.
- The processes should be **aborted one at a time** until the **deadlock cycle is eliminated**.
- Things to consider when choosing the order of aborting processes:
  1. Process priority.
  2. The amount of time the process has been running, and how much longer it has to run until it is completed.
  3. Resources the process has allocated.
  4. Resource the process is attempting to allocate.
  5. Is the process interactive or batch.

Deadlock Recovery — Resource Preemption

- Another way to **recover from a deadlock** is to **select a victim** that will minimize the **cost**, and rollback that process to a **safe state**.
- This may cause **starvation** if the same process is always selected.
- This also requires a lot overhead, you have to maintain previous states of all processes.