Classification of Resources

- **Sharable resources.** Can be used by several users at the same time.
  - → Program code (if reentrant)
  - → Data areas if read only.
- **Non-sharable (exclusive) resources.** Can only be used by one user at a time.
  - → Data areas that need to be written.
  - → Most external devices like printers.
  - → The processor.

Non-sharable Resources

- **preemtable.** Use of resource may be preempted and restarted later on.
  - → The processor
  - → The primary memory
- **non-preemptable**
  - → Printers. (A new document cannot be printed until the previous is completed.)

Problems with Resource Administration

1. **Deadlock** - The processes may block each other. Nobody can use the resource.
2. **Starvation** - Some process is prevented from using a resource because there is always another (higher priority) process that is using it.

Deadlock can only occur if the processes need more than 1 resource at the same time.

Starvation is possible in all resource allocation situations unless a suitable policy is used to prevent it.

Starvation can be prevented by assigning the resources in strict FIFO order or by increasing the priority for a process that have been waiting for a long time.

Deadlock

Deadlock may arise if the following four conditions hold simultaneously:

1. **Mutual exclusion.** Non-sharable resources are used.
2. **Hold and wait.** A process holds at least one resource and requests another resource that is held by another process.
3. **No preemption.** Resources that a process holds, can only be voluntarily released by the process itself.
4. **Circular wait.** A circular chain of processes exist, in which each process holds a resource requested by the next process in the chain.

Methods for Handling Deadlocks

- **Deadlock prevention.** Prevents deadlock by ensuring that at least one of the four necessary conditions do not hold.
- **Deadlock avoidance.** Tries to improve the resource utilization by using a less stringent method than deadlock prevention.
- **Deadlock detection.** Methods to detect that deadlock has occurred .
- **Deadlock recovery.** Methods for recovery from deadlock.

Deadlock Prevention

Mutual exclusion.

Sharable recourses do not require mutual exclusion and do not create deadlocks.

In general, avoiding mutual exclusion is not possible because some recourses are intrinsically non-sharable.

Deadlock Prevention

Hold and Wait.

This condition can be prevented by requiring the processes to request all resources at the same occasion.

A variant is that a process that want to allocate additional resources first must release all resources that it already holds and then request both the old and the new resources at the same time.

Disadvantages:

- May lead to poor resource utilization.
- May create starvation.

Deadlock Prevention

No preemption.

The third condition can be prevented with the following method:

If a process holds a resource and requests a new resource that cannot be immediately allocated, then all resources currently being held are preempted. The Process is restarted first then both the new and the old resources are available.

This method is often applied to resources whose state can be saved and restored as for example CPU registers and primary memory. It cannot be used for nonpreemptable resources.

Deadlock Prevention

Circular Wait.

One way to ensure that this condition never holds is to impose a total ordering on the resources. Every resource is given a unique number. The resources must then be requested in an increasing order of enumeration

In order to get good resource utilization, the resources should be numbered in the same order that they are normally needed.

Deadlock Avoidance

- If all four conditions for deadlock is met, deadlock can still be avoided by using additional information about how resources are to be requested.
- The most common method is that each process must declare the maximum number of resources of each type that it may need.
- The processes may demand new resources in an arbitrary order.
- The resource allocator uses a deadlock avoidance algorithm to evaluate each new resource application and grants the allocation only if it leaves the system in a *safe state*.
- The system is in a *safe state*, if the resource allocator can guarantee that all processes will be completed.

Safe and Unsafe States

Assume that the system has 12 units of a resource.

A safe state.

	current need	maximum need
P0	1	4
P1	4	6
P2	5	8

Free: 2

An unsafe state.

	current need	maximum need
P0	1	4
P1	4	6
P2	6	8

Free: 1

Safe and Unsafe States

A system can go from a safe state to an unsafe state.

This stat is safe.

	current need	maximum need
P0	1	4
P1	4	6
P2	5	8

Free: 2

Assume that process P2 requests another resource and that the request is granted. This results in this state.

	current need	maximum need
P0	1	4
P1	4	6
P2	6	8

Free: 1

This state is *unsafe* because there is only one free resource and all processes may need at least 2 extra resources.

The mistake was to grant the last request from P2.

Banker’s Algorithm

Not all unsafe states are deadlocks, but an unsafe state may lead to deadlock. As long as the state is safe the system can avoid unsafe states (and deadlocks).

Banker’s algorithm is the most well-known deadlock avoidance algorithm.

- The processes may demand new resources in arbitrary order but must declare the maximum number of each resource type that they may need.
- The system grants a new application only if it results in a safe state.
- Otherwise, the process must wait until some other process releases sufficiently many resources.

Problems with banker’s algorithm:

- There must be a fixed number of resources to allocate. This may be difficult to meet since external resources may break down and become unavailable.
- The algorithm requires that the processes state their maximum resource needs. This is not always known.

Deadlock Detection

An alternative to preventing deadlock can be to *detect* deadlock and then use some recovery method. Deadlock can be detected with the aid of resource-allocation graphs.

If there only exists one resource of each type, deadlock has occurred if there exists a loop in the resource-allocation graph.

Reduction of resource-allocation graphs

- If the exists several resources of each type, a loop in the graph need not mean deadlock.
- In this case, the method with reduction of the graph can be used.
- If the resource requirements for a certain process can be met, we say that the graph can be reduced with that process. The reduced graph is the graph with this process removed.
- If the graph can be reduced with all its processes there is no deadlock. If certain processes cannot be reduced, a deadlock exists and the the remaining loop in the graph constitutes the deadlocked processes.

Detection Algorithm Usage

Although it is relatively simple to detect a deadlock by looking at the graph, algorithms for deadlock detection are computationally expensive.

An algorithm for detection of a loop in a graph has complexity O(n^2), there n is number of nodes in the graph.

When should we invoke the detection algorithm?

- Deadlock may occur each time a process makes a resource request that cannot be granted immediately.
- In principle, one should therefore run the detection algorithm at each such request.
- Due to the algorithm's high complexity this will use to much CPU time, if there are many processes.
- Therefore, a compromise is needed. How often to run the algorithm is dependent on the judged risk for deadlock an the consequence if deadlock do occur.

Deadlock Recovery

Unfortunately, no good method exists for deadlock recovery

The following methods are possible

Recovery through killing processes.

- Kill one or more processes until the deadlock cycle is broken.
- The killed processes lose all calculated results and have to be restarted.
- Processes that leave permanent changes in the file system, may give incorrect result if they are restarted.

Recovery through preemption.

- With preemtable resources, one can temporary take resources away from a process and give them to another process, in order to resolve a deadlock.
- With non-preemptable resources this is equivalent to killing the process.

Checkpointing

- If check points are used, only part of the process must be rerun if it is killed.
- This must be programmed into the processes.

Deadlock handling in real systems

The handling of deadlock is different for different classes of resources.

Different classes of resources:

- **internal resources.** For example data structures used by the operating system.
- **Primary memory and processors.**
- **External resources.** Resources that may be allocated by the processes, for example files.

Deadlock handling for different types of resources.

- **Internal resources**. In SMP systems, the data structure *locks* have to be combined with a deadlock prevention method. Usually the data structures are enumerated and the processes required to lock them in enumeration order.
- **Primary memory and processor.** Deadlock can be prevented through preemption since these resources are preemptable.
- **External resources.** Most operating systems take no measures for these resources.