Systems Programming and Operating Systems	**Induprakas Kodukula**
CS414, Summer 1996	**Class#7, 7/17/96**

1 Class #7

Deadlocks are a natural consequence of mutual exclusion. Informally, a deadlock is said to have happened when certain processes cannot proceed any further. For example, consider the following simple program that uses two semaphores for accessing a critical region. The mutexes are S and Q

```
Process 1                                Process 2

  P (S)                                    P (Q)
  P (Q)                                    P (S)

  V (S)                                    V (Q)
  V (Q)                                    V (S)
```

Suppose process 1 executes P(S) and process 2 executes P(Q). Subsequently, process 1 executes P(Q) and blocks. Similarly, process 2 executes P(S) and blocks. Since the V(S) in process 1 and V(Q) in process 2 will never be executed, these two processes will be blocked forever, without being able to make any progress. This is a typical instance of a deadlock.

Deadlocks typically happen if processes are waiting for multiple *resources*. These resources can be broadly divided into two classes: *permanent* resources, which are typically hardware resources, such as tape drives and the like, and *temporary* resources which are typically produced by one process and consumed by another. A typical resource acquisition strategy used by a process is the following:

Request a resource

Wait until resource (or required units) are available

Use resource

Release resource

For example, suppose we have 4 units of a resource available and there are two processes which need 3 units of this particular resource. Let us assume that each process makes a request of 3 units, but this is processed as a request of 1 unit at a time. In other words, in one single attempt, each process can get only one unit of the resource. Since no atomicity is guarenteed regarding the requests of the processes, it is possible that each of the processes ends up with 2 units of the resource. At this stage, since no more units of the resource are available, both processes will block. This particular deadlock could have been avoided if we had processed the entire request of 3 units by each process in one shot.

A deadlock situation can arise if all the four following conditions are satisfied at any given time

Mutual Exclusion At least one resource must be held in a nonshareable mode, that is only one process at a time can use that resource. If a process requests the use of resource while another is holding it, the requesting process must be delayed until the holding process releases the resource.

Hold and Wait There must be at least one process that is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes.

No preemption Resources cannot be preempted. In other words, if part (or whole) of a resource being held by a process cannot be taken away from it. It can only be relinquished voluntarily.

circular waiting There must exist a set $P_0, \ldots, P_n$ such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, ..., and P_n is waiting for a resource held by P_0.

In other words, to avoid a deadlock, at least one of the above conditions must be false at all times. It is easy to see that the *circular waiting* condition implies the *Hold and wait* condition, which implies that the conditions above are not independent. Exploring each of these possibilities yields the following:

Mutual exclusion is needed in general for system integrity, so it is not really dispensable.

Preemption is in general quite expensive. This is because transferring resources is quite time consuming in general. Sometimes, for output devices, it is not even practical. For example, it is not a practical solution to interrupt a line printer in the middle of a printing job to start printing something else.

Complete allocation is not really a very acceptable solution either. Complete allocation would imply that the combined resource requirement of the maximum number of processes that can run concurrently is limited by the resources available to the system.

This leaves the following options to deal with deadlocks.

Detect deadlocks. When a deadlock is detected, special purpose code must be written to recover from it. This is known as *deadlock recovery.* Since it is not known in advance when a deadlock may occur, this approach could be quite expensive.

Prevent the possibility of deadlocks by using special resource allocation policies. This approach is known as *deadlock prevention.* The resource allocation policies ensure that all the four conditions needed for a deadlock to happen are **never** satisfied simultaneously.

The third approach requires programs to give more information about the resources they require. Essentially, a program gives advance notice to the operating system of all the resources it requires. Depending on all the information available, for each actual request made by a process, the operating system decides whether the request should be granted or not, and whether the process should block or not.

Each of the above approaches of dealing with deadlocks is presented next.

1.1 Deadlock detection

Deadlocks can be detected in one of two ways.

One could wait until all activity of the system ceased. This is potentially very expensive, and is not very practical. An example of this approach is in a spooling system, where memory eventually becomes unavailable and the system cannot produce any more output.

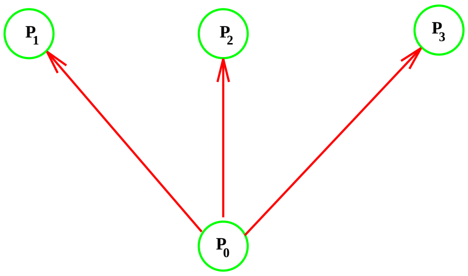

Figure 1: : Wait-for graph approach to the detection of deadlocks

> A second approach for detecting deadlocks uses a model to keep track of resource allocation. Using this model, the operating system can keep track of whether all the conditions necessary for a deadlock to happen are satisfied. Typically, such models are used to detect the *circular wait* condition.

At this stage, studying of a couple of models that are used to detect deadlocks will be useful to provide insight into the nature of these models.

1.1.1 Wait-for-graph

The first approach we will look at will be called the wait-for graph approach, illustrated in Figure 1. The wait-for graph is defined as a tuple (V, E), where V is the set of nodes and E is the set of edges.

> V = process, i.e. V contains a node for all the processes that can request a resource.
>
> $E = V_iV_j$, such that V_i waits for V_j to release a resource before it can run
>
> Every edge is labeled with the type of the resource that is being waited upon.

Given this data structure, the following procedure is used to determine if there is the possibility of a deadlock. A deadlock exists in the system if and only if the *wait-for* graph has a cycle. So, the detection of a deadlock boils down to the detection of a cycle in the *wait-for* graph. This can be done as follows:

> Check for a node that has no outgoing edges. Delete this node. In addition, all edges going into the node are also deleted.
>
> In addition, delete all edges labeled with the same label as any of the edges being deleted. Intuitively, this corresponds to the case of waiting for one of many units of a resource, where the various units are indistiguishable from each other.
>
> This process is repeated as long as possible. If eventually, the graph is empty, then there is no deadlock.

1.1.2 Holt-approach

Holt uses a slightly different graph for modeling the acquisition of resources and deadlock. The *Holt* graph for a set of processes trying to acquire a set of resources can be defined as a tuple (V, E) as follows:

V consists of two types of nodes: process, resource, i.e. V contains one node for every process and for every type of resource available.

All edges are of the form $P_i R_j$ or $R_i P_j$. There is an edge from P_i to R_j if process i is waiting for resource j. Similarly, there is an edge from R_i to P_j if resource i is allocated to process j.

It is fairly easy to see that there is a deadlock if and only if there is a cycle in the *Holt* graph. The advantage of this approach over the *wait-for* graph is that when an edge is deleted, we don't have to worry about deleting other edges with the same label.