Lecture 07: Resource allocation and Deadlock Management

Lecture 3 started the discussion of process management, with the emphasis on CPU scheduling. This lecture continues that discussion but with a slightly different focus. Here, we will concentrate on how the operating system allocates resources for multiple processes. This must be done in a way that is efficient and useful. The lecture proceeds under the following captions:

- Introduction
- deadlock Definition and Characterization
- Resource Allocation Graph
- Controlling Deadlock
- Innovation — Printer Spooling
- Summary and concluding Remarks

7.1 Introduction

As we have seen so far, an operating system is really a complex software system that manages resource allocation in a computer system. Among the resources managed are processes, memory, the CPU, printers and other peripheral devices, etc. Some resources are non-sharable, while others are sharable. For instance, except for read-only data, memory locations are non-sharable — two executing processes cannot access the same RAM location as this would cause a fatal error in execution. Peripheral devices such as printers are traditionally non-sharable, but as we will soon see, thanks to spooling, this is no longer the case.

It may happen that the waiting processes never change state, because other waiting processes are holding the resources they require. This situation is called deadlock.

Example 1:

Consider a system with one line printer (LP) and one tape drive (TD). There are two jobs running as follows:

- Job A allocates TD and awaits the LP
- Job B allocates LP and awaits the TD

Result: Neither A nor B can continue. They are in a state of deadlock.

Example 2:

A highly congested communication network could be deadlocked if there is not a strict communication protocol about sending messages over a common access medium.

Magnetic disks are designed to be shared. Hence, without controls to regulate disk usage, competing processes could send conflicting requests to the device manager and thereby cause deadlock.

7.2 The Deadlock Definition and Characterization

A set of processes is in a state of deadlock when the processes are in a state of waiting for some event which can only be caused by some other process(es) in the set, such event not being likely to occur. Deadlock need not affect all jobs in the system at once.

7.2 The Deadlock Definition and Characterization (continued)

The effect of deadlock is that a number of the process in the set cannot be completed. The following conditions are necessary (but not sufficient) for deadlock to occur:

a. **Mutual Exclusion:** At least one of the resources involved is non-shareable.
b. A **hold and wait** condition must exist: One process is holding at least one resource, while waiting to acquire additional resource(s) being held by other process(es).
c. **No Preemption** of resources (e.g. a resource can only be released voluntarily by the process holding it *after the process has completed its task*).
d. **Circular Wait:** Processes (P_0 P_1 P_2 … P_n) order unimportant, such that P_0 waits on P_1, P_1 waits on P_2…and P_n waits on P_0.

The conditions are necessary but not sufficient: Occurrence of deadlock implies satisfaction of conditions. Satisfaction of conditions does not necessarily imply that deadlock will occur; rather, it implies that deadlock may occur.

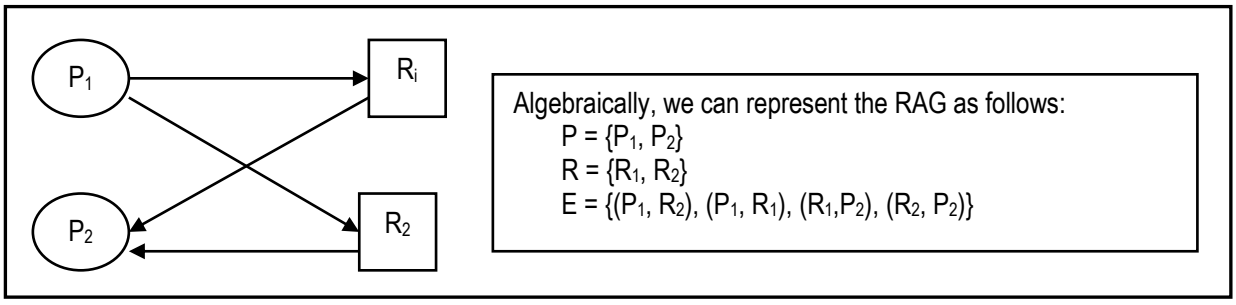
7.3 Resource Allocation Graph

It is customary to use a system resource allocation graph (RAG) to describe resource allocation status of a system. The graph consists of vertices and edges as follows:

- Vertices represent processes or resource types.
- Edges represent either process-to-resource (a request) or resource-to-process (an allocation), determined by the direction of the arrow.
- Circles represent processes; squares (boxes) represent resource types; no. of dots in resource box indicates the number of instance of that type.

Figure 7.1 provides an illustration of a resource allocation graph. According to the figure, process P_2 is running with R_1 and R_2 allocated to it; process P_1 is waiting on R_1 and R_2.

Figure 7.1 Illustration of a Resource Allocation Graph



7.3 Resource Allocation Graph (continued)

Note: The RAG cannot predict what resources a process will need or what requests will be made; it simply represents a current state.

Algebraically, the RAG may also be represented as two sets V, E:

V is the set of vertices consisting of:
$P = (P_1\ P_2 \ldots P_n)$, the set of processes
$R = (R_1\ R_2 \ldots R_n)$, the set of resource types.

E is the set of edges, each edge being an ordered pair $(P_i\ R_i)$ or $(R_j\ P_j)$, the former representing a request, the latter representing an allocation. We therefore have the sets P, R, E.

Resource allocation graphs are quite handy in representing complex situations. If a complete closed path (a cycle) can be traced in the graph, deadlock may exist.

Figure 7.2: RAG Indicating Likelihood of Deadlock

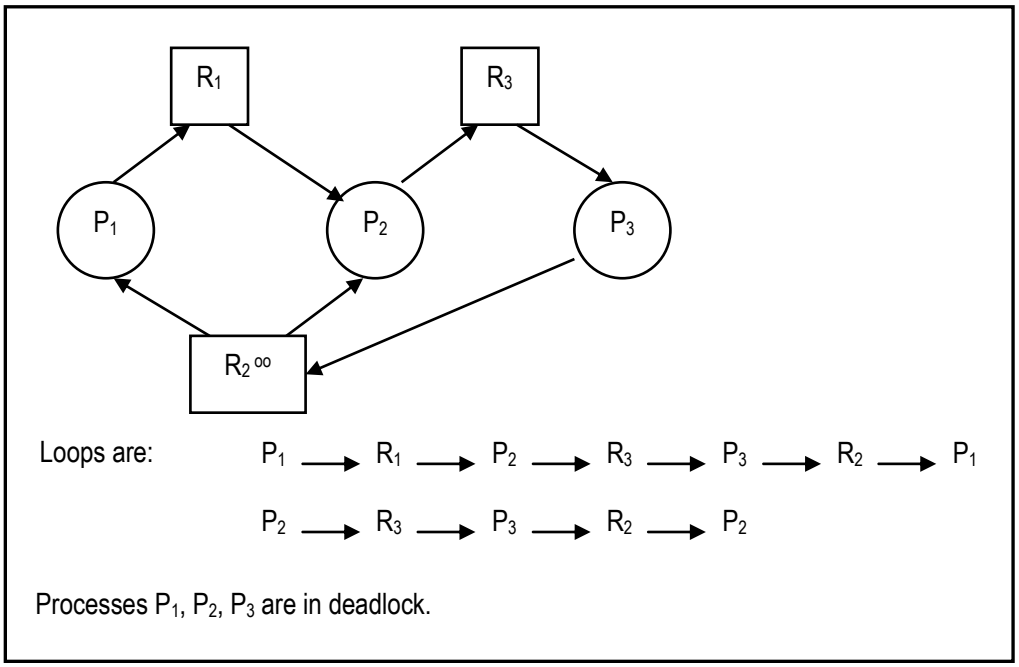
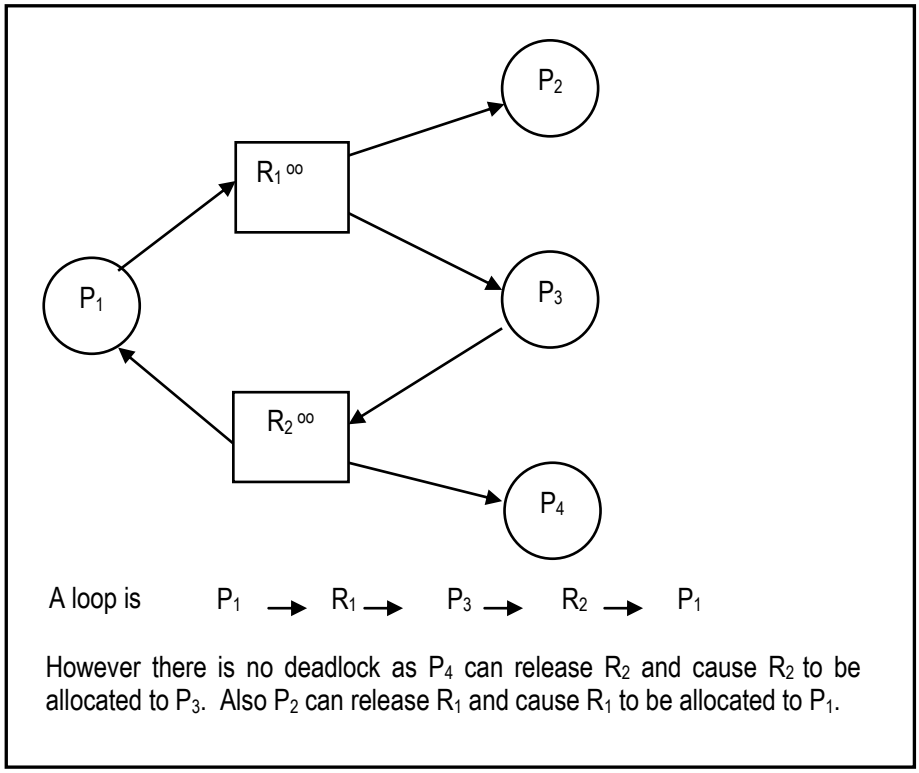


Figure 7.3: RAG with Cycle but no Deadlock



7.4 Control of Deadlock

There are three approaches to managing deadlock.

- Deadlock prevention
- Deadlock avoidance
- Deadlock detection and recovery

7.4.1 Deadlock Prevention

In this approach, at least one of the conditions necessary for deadlock is denied (prevented) occurrence. Let us examine each condition.

Condition 1 — Mutual Exclusion

There is not much future in attempting to prevent the existence of certain non-shareable resources. However, the operating system could provide a spool files for devices such as printers.

7.4.1 Deadlock Prevention (continued)

Condition 2 —Hold and Wait

The operating system could be made to satisfy the request of a process before the resource(s) is (are) needed. For example, if a process will need a maximum of n RAM locations, assign them at the start of execution. This approach is wasteful and expensive.

Alternatively, the job may be partitioned into stages. The resources required by a stage are requested prior to the running of that stage.

Two problems may occur from this approach:
- Starvation: A process never gets resources because other process(es) have allocated them.
- Resource utilization would be low since many resources may be allocated to a process but not used at the same time.

Condition 3 — No Preemption

The no preemption condition could be denied by allowing some process to be able to preempt resource(s) from others. One approach is as follows:
- If a process p_1 requests some resources and they are available, allocate them.
- If resources requested are not available, check to see whether they are allocated to some other process p_2 that is waiting on other resources. If this is the case, preempt the resources from p_2 and allocate them to p_1.

This approach would apply to resources whose status can easily be saved and restored, such as CPU registers, memory space. The approach could not be applied to resources such as; printers, card readers, tape drives etc. For such devices, a spooling system is usually in place.

Condition 4 — Circular Wait

The circular wait condition can be denied by imposing some ordering on the resources types. A nominal value is applied to each resource type. One of two strategies could then be applied:
- Each process must request resources in increasing order of enumeration; for instance, R_I would be allocated before R_j where the nominal order of R_j is greater than the nominal order of R_i.
- Alternately, a process in requesting resource type R_j, must first release resource type R_I,

The main drawback is that a resource may sit idle for a long time. It is therefore not very efficient.

7.4.2 Deadlock Avoidance

In deadlock avoidance, we allow the necessary conditions but avoid the circumstances causing deadlock. Each request is examined. If granting a request will improve the likelihood of deadlock, the request is denied (until some later stage). Looking ahead to determine the probability of deadlock is an approximate method.

7.4.2 Deadlock Avoidance (continued)

A set of processes is in a *safe state* if there exists a safe sequence $[P_1\ P_2 \ldots P_n]$ such that:

- For each P_j , the resources which P_j can still request can be satisfied by the currently available resources plus the resources held by P_i ($_i < _j$).
- Thus, if P_j needs resources not immediately available, then P_j could wait until P_i is finished.
- The safe sequence defines the sequence in which processes are guaranteed to finish.

For each resource request, the OS must determine whether a safe sequence can be found, before granting the request. To illustrate how this works, consider the following processes in a system which has twelve (12) TD's available as illustrated below.

Figure 7.4: Illustrating the Concept of Safe Sequence

Process	Required Max	Holds	
P1	10	5	
P2	4	2	
P3	9	2	
		9 Held	Therefore 3 TD's free

Note:

1. A safe sequence is $[P_2\ P_1\ P_3]$.
2. The safe sequence could not start with either P_1 or P_3.
3. P_2 will complete and release 4 TD's to add to the extra 1 TD that was unallocated. P_1 will use the 5 TD's released from P_2, to complete and then release 10 TD's. P_3 will then execute.

We shall examine two deadlock avoidance algorithms: the **Banker's Algorithm** and the **Safety Algorithm**. The required data structures along with the two algorithms are provided in figure 7.5.

Figure 7.5a: Data Structures Required for the Banker's Algorithm

Available: A vector (1D array) of length m, indicating the number of available resources of each resource type. Abbreviated as Avl.
Avl [$_j$] = k => there are k instances of R_j available.

Max: An n x m matrix (2D array) indicating the maximum requirement of processes.
Rows = processes ($P_1 \ldots P_n$).
Columns = entries for resource types $R_1 \ldots R_m$.
Max [i, j] = k => process P_i may request a maximum of k instances of R_j.

Allocation: An n x m matrix (2D array) indicating the current allocation of resources to processes.
The interpretation is similar to Max.
Alc[$_{i\ j}$] = k => process P_i has k instances of R_j allocated to it.

Need: An n x m matrix (2D array) indicating the current resource needs of processes.
Need = Max – Allocation.

Additional Notations:

If X, Y are two vectors of length n, then X<=Y if X [$_i$] <=Y [$_i$] for $_i$ = 1,2…n
Example: if X = [1 1 1 1] and Y = [4 3 2 4] then X<=Y.

We can treat rows of matrices Allocation, Max, and Need as Vectors and refer to them as Alc[i], Max[i], and Need[i], respectively. Thus, Need[i] refers to all the resource needs of P[i].

Figure 7.5b: Banker's Algorithm

Let Request[i] be the request vector for P[i].

1. If Request[i] <= Need[i] then proceed to step 2. Otherwise we have an error (*process requesting more than it needs*).

2. If Request[i] <= Available then proceed to step 3. Otherwise the resources are not available and p_i must wait.

3. Pretend to allocate the requested resources:
   Available := Available – Request[i];
   Allocation[i] := Allocation[i] + Request[i];
   Need[i] := Need[i] – Request[i];

Call the Safety Algorithm to determine if the new state is safe. If the new state is safe, proceed with allocation. Otherwise Pi must wait and the old allocation state be restored.

Figure 7.5c: Safety Algorithm (determines whether system is safe)

Let Work be a vector of length m (an entry for each resource type).
Let Finish be a vector of length n (a boolean flag for each of n processes).

1. Initialize Work := Available;
   Finish[i] := false for $_i$ = 1 to n;
2. Find an i such that
   (Finish[i] = false) and (Need[i] <= Work). If none exist, go to step 4.
3. Work := Work + Allocation[i]
   Finish[i] := true; Go to step 2.
4. If Finish[i] = true for $_i$ = 1 to n, then the system is in a safe state. Otherwise the system is not in a safe state.

Note: With the Banker's algorithm, the programmer must specify what the maximum number of each resource will be.

Example 3: Consider a system with five processes (P_1 P_2 P_3 P_4 P_5) and three resource types (ABC). There are 10, 5 and 7 instances of resource types A, B, C respectively.

Figure 7.6 provides a snapshot of the system at time t_o. We shall use the above two algorithms to analyze resource allocation for the jobs.

Figure 7.6: Deadlock Avoidance Example (a system snapshot at time t_0)

	Allocation ABC	Max ABC	Available ABC
P1	0 1 0	7 5 3	3 3 2
P2	2 0 0	3 2 2	
P3	3 0 2	9 0 2	**Resource instances**
P4	2 1 1	2 2 2	ABC
P5	0 0 2	4 3 3	10 5 7

We can calculate the Need matrix as follows:

Need = Max – Allocation =

$$\begin{pmatrix} 7 & 4 & 3 \\ 1 & 2 & 2 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}$$

$$Avl[j] = ResInstance[j] - \sum_{i=1}^{n} ALc[i,j] \quad \text{(for } j=1\ldots.m\text{)}$$

7.4.2 Deadlock Avoidance (continued)

From Banker's Algorithm, any process, $Request_i$ (for $_i$ = 1 to 5) must satisfy the condition (Request[i] <= Need[i]) and (Request[i] <= Available) in order to be considered for execution.

Suppose that Request[i] = Need[i] Then P_2, P_4 may be tested for starting a safe sequence. Suppose that we start with P_2. Then by applying step 3 of the Bankers Algorithm will yield the result shown in figure 7.7

Figure 7.7: Banker's Algorithm Applied to Figure 7.6

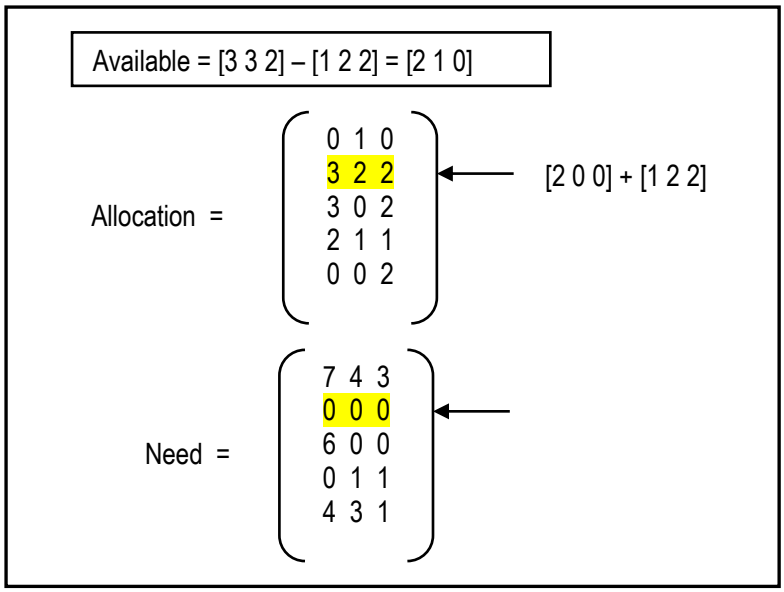
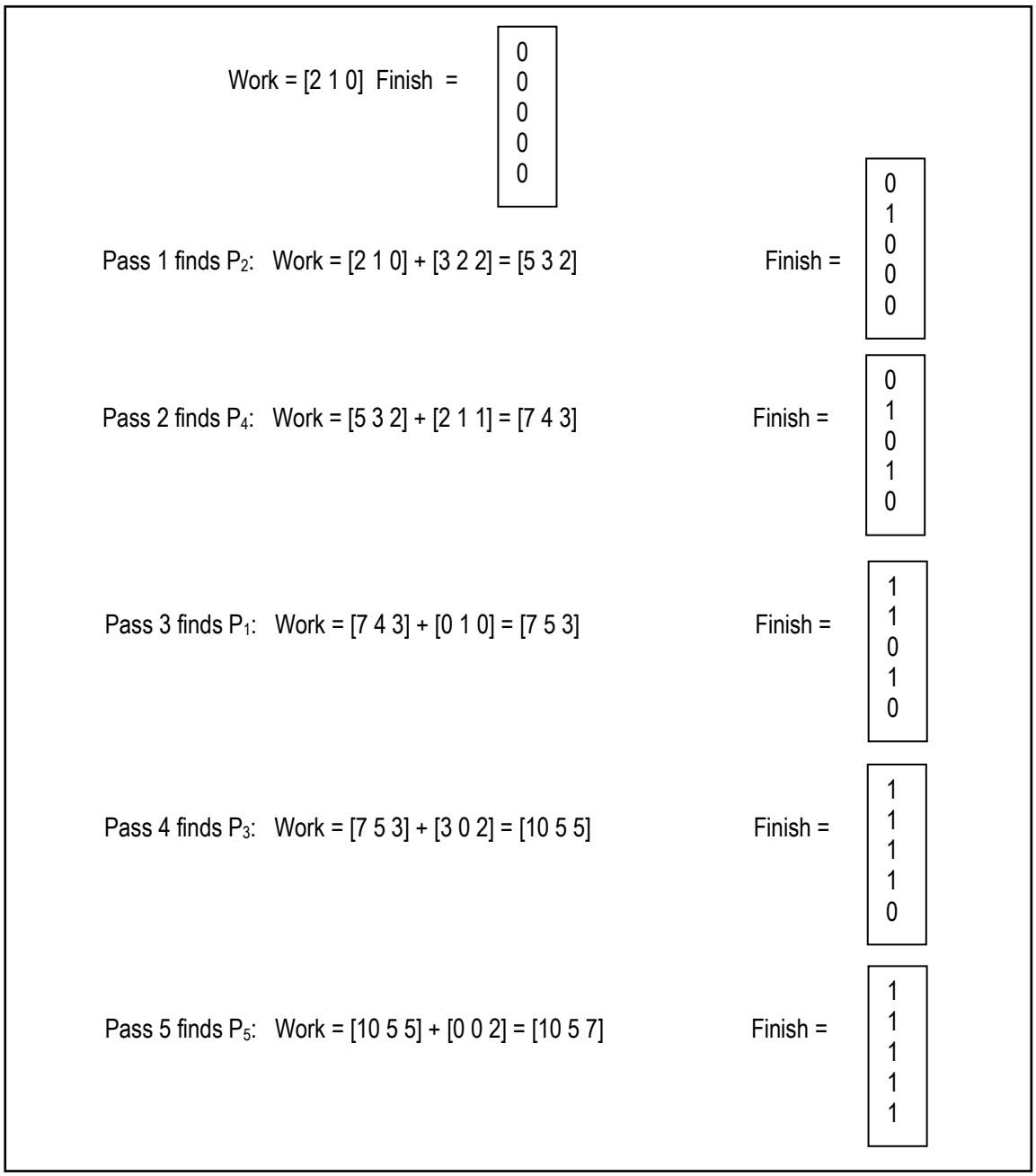


Figure 7.8 illustrates the use of the Safety Algorithm to establish that this allocation results in a safe state. A safe sequence is [P_2 P_4 P_1 P_3 P_5]. Allocation may therefore be made according to the safe sequence.

Figure 7.8: Illustrating use of Safety Algorithm to find a Safe Sequence (see figure 7.7)



Note that for single instance of each resource type it is more economical to use (a variation of) the resource allocation graph to avoid deadlock.

7.4.3 Deadlock Detection and Recovery

The third approach for managing deadlock is to allow the deadlock to occur, but have an algorithm in place for detecting it, and strategies for recovering from it. The deadlock detection algorithm is invoked periodically.

The data structures required are similar to those used in the previous sub-section: The conventions used for the Banker's Algorithm are also adopted here. The detection algorithm is provided in figure 7.9. Notice its similarity to the Safety Algorithm of the previous subsection.

Figure 7.9: Deadlock Detection Algorithm

Required data Structure:
Available: A vector of length m
Allocation: An n x m matrix
Request: An n x m matrix indicating the current request of each process.
Work: A vector of length m
Finish: A vector of length n
Request[i, j] = k => process Pi is requesting k instances of resource R_j.

1. Work := Available;
   For i = 1 to n do the following:
   If Allocation[i] <> 0
   Then Finish[i] := False
   Else Finish[i] :=True
   End-if
   End-for

2. Find an index[i] such that (Finish[i] = False) and (Request[i] <= Work)
   If not found, go to step 4.

3. Work := Work + Allocation[i]
   Finish[i] := True;
   Go to step 2

4. If Finish[i] = False for some $_i$ (* 1 <= $_i$ <= n*)
   then the system is in deadlock. Further, if Finish[i] = False then P[i] is deadlocked.

7.4.3 Deadlock Detection and Recovery (continued)

If there is only one instance of each resource type, it is more efficient to use the resource allocation graph discussed earlier, to detect deadlock. You are advised to try examples, similar to the one in the previous section, to consolidate understanding.

The following recovery strategies are possible:

- Preempt resources from some process and give them to others.
  - The victim process could be selected based on priority or some other criteria.
  - The victim process must be rolled back to a safe state for later resumption.

- Deadlocked processes are abandoned.
  - Processes may be killed in stages until the deadlock cycle is eliminated or
  - All deadlocked processes may be killed — quite an expensive option.

  Killing a process may be very costly. Consider for instance a file update process. Killing this process could result in dangling records in the system. These integrity issues must be addressed by the operating system.

7.5 Innovation — Printer Spooling

A classic example of innovation in dealing with deadlock is the use of *virtual devices*, typically implemented as spool files.

Printers have spool files attached to them. The processes that need to output data to a printer, submit their requests to the printer's spool file, instead of requesting the printer as a resource.

The operating system selects print jobs from the spool file, to be run on the printer, based on some scheduling algorithm (typically, but not necessarily FIFO).

The effect of this strategy is that it is no longer possible to have deadlock about a printer in such a system. The non-sharable device is therefore effectively "transformed" to a sharable device. Neither do jobs have to wait on the printer, so that job throughput is also enhanced.

Operating systems such as Unix and IBM i have mastered this strategy for years; however newer systems such as Windows-2000 still struggle with the problem of optimal use of printers in a network.

7.6 Summary and Concluding Remarks

Let us summarize what has been covered in this lecture:

- Deadlock is the situation that occurs when processes in a set are in a state of waiting for some event which can only be caused by some other process(es) in the set, such event not being likely to occur.
- Four conditions necessary for deadlock are the mutual exclusion condition, hold-and-wait condition, the no preemption condition, and the circular wait condition.
- A resource allocation graph (RAG) is a graphical representation of processes and the allocations in a system. The existence of loop(s) in a RAG indicates that deadlock may be likely to occur. However, it is possible for a RAG to have loop(s) and the system is not deadlocked.
- Deadlock prevention strategy involves denying at least one of the four conditions necessary for deadlock.
- Deadlock avoidance strategy involves use of the Banker's algorithm and Safety algorithm to look ahead, and avoid situations that would lead to deadlock.
- Deadlock detection strategy involves identifying the presence of deadlock in the system, and recovering from it.

No single approach (of the three approaches to deadlock management) is best. Usually a combination of the three approaches will ensure that a system is never deadlocked.

Deadlock management also applies to database management, not just hardware resources.

Example:

> Accessing a file for update causes record locking. If a locked record is attempted to be read for update by another user, the second user must wait for a period prescribed by the OS. Recovery will be based on the DBMS or programming language.

Operating systems such as Unix, IBM i and Linux handle the deadlock problem particularly well. However, newer systems such as Windows 2000 still struggle with this problem. Most users of Windows 9x, Windows NT and Windows 2000 are familiar with the scenario where one is running a number of multiple tasks, and then inexplicably, (at least) one of them stops responding, at which point a system request has to be taken (via <Ctrl-Alt-Delete>) and the non-responding job(s) cancelled.

The control of deadlock is of paramount importance in the design and development of an operating system; failure to effectively address deadlock could determine the failure of the system in the market place.

7.7 References and Recommended Readings

[Bacon & Harris 2003] Bacon, Jean & Tim Harris. 2003. *Operating Systems: Concurrent and Distributed Software Design*. Addison-Wesley. See Chapter 4.

[Nutt 2004] Nutt, Gary. 2004. *Operating Systems: A Modern Perspective* 3rd ed. Boston: Addison-Wesley. See Chapter 10.

[Silberschatz 2012] Silberschatz, Abraham, Peter B. Galvin, & Greg Gagne. 2012. *Operating Systems Concepts*, 9th Ed. Update. New York: John Wiley & Sons. See Chapter 7.

[Stallings 2005] Stallings, William. 2005. *Operating Systems* 5th ed. Upper Saddle River, New Jersey: Prentice Hall. See Chapter 6.