UNIT 5. DEADLOCK AND STARVATION

5.1 PRINCIPLES OF DEADLOCK

Deadlock can be defined as the *permanent* blocking of a set of processes that either compete for system resources or communicate with each other. A set of processes is deadlocked when each process in the set is blocked awaiting an event (typically the freeing up of some requested resource) that can only be triggered by another blocked process in the set. Deadlock is permanent because none of the events is ever triggered.

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

1. **Request**: If the request cannot be granted immediately (for example, 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.
3. **Release**: The process releases the resource.

But, if every process in the set is waiting for other processes to release the resource, then the deadlock happens. Two general categories of resources:

- **Reusable**: can be safely used by only one process at a time and *is not depleted* (that is not reduced) by that use. For example, Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores. Here, deadlock occurs if each process holds one resource and requests the other.
- **Consumable**: one that can be created (*produced*) and destroyed (*consumed*). For example, interrupts, signals, messages, and information in I/O buffers. Here, deadlock may occur if a Receive message is blocking.

5.1.1 Conditions (or Characterization) for Deadlock

Following are the conditions for deadlock to present:

- **Mutual exclusion:** only one process can use a resource at a time
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, and so on, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

5.1.2 Resource Allocation Graphs

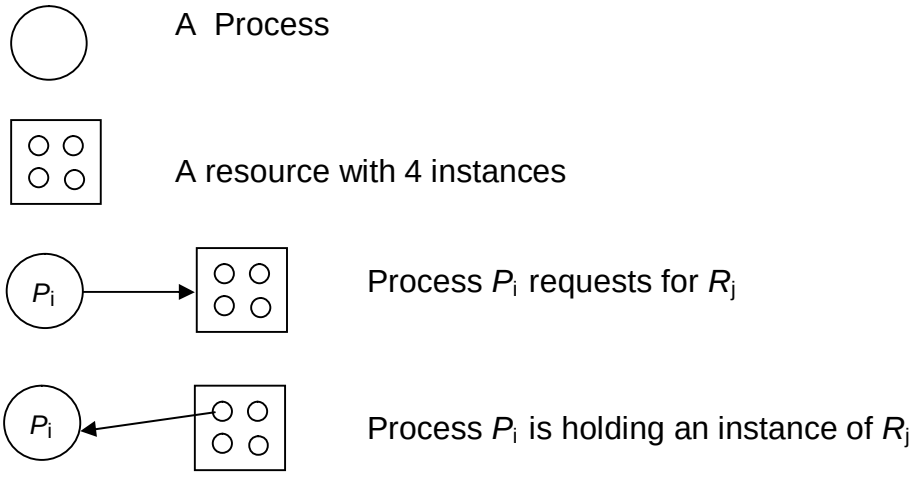
The resource allocation graph is a directed graph that depicts a state of the system of resources and processes with each process and each resource represented by a node. It is a graph consisting of a set of vertices *V* and a set of edges *E* with following notations:

- V is partitioned into two types:

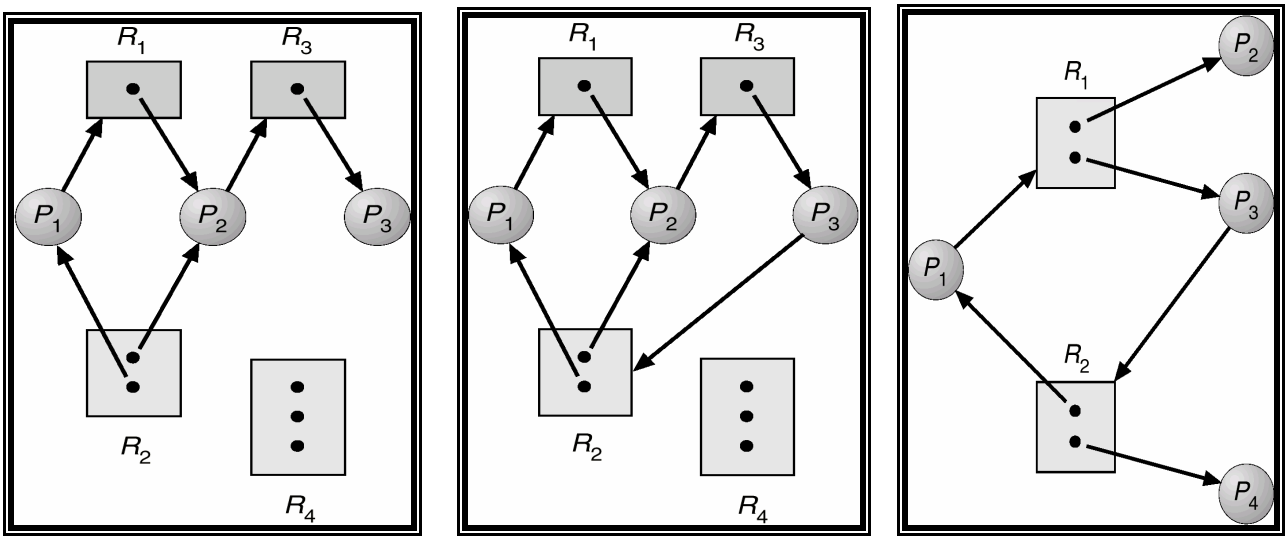
- $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
- $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.

- **Request edge:** A directed edge $P_i \rightarrow R_j$ indicates that the process P_i has requested for an instance of the resource R_j and is currently waiting for that resource.
- **Assignment edge:** A directed edge $R_j \rightarrow P_i$ indicates that an instance of the resource R_j has been allocated to the process P_i

The following symbols are used while creating resource allocation graph:



Examples of resource allocation graph are shown in Figure 5.1. Note that, in Figure 5.1(c), the processes P2 and P4 are not depending on any other resources. And, they will give up the resources R1 and R2 once they complete the execution. Hence, there will not be any deadlock.



(a) Resource allocation Graph (b) With a deadlock (c) with cycle but no deadlock

Figure 5.1 Resource allocation graphs

Given the definition of resource allocation graph, we can understand that, if there is no cycle in the graph, then there will not be a deadlock. If there is a cycle, there is a chance of deadlock.

There are three general approaches exist for dealing with deadlock.

- Prevent deadlock: Ensure that the system will *never* enter a deadlock state.
- Avoid deadlock: Make appropriate dynamic choices based on the current state of resource allocation.
- Detect Deadlock: Allow the system to enter a deadlock state and then recover.

5.2 DEADLOCK PREVENTION

The strategy of deadlock prevention is to design a system in such way that the possibility of deadlock is excluded. This is possible if we ensure that one of the four conditions (mutual exclusion, hold & wait, No preemption and circular wait) cannot hold. We will examine these conditions now.

- **Mutual Exclusion:** The mutual-exclusion condition must hold for nonsharable resources. For example, a printer cannot be simultaneously shared by several processes. On the other hand, sharable resources do not require mutually exclusive access, and thus cannot be involved in a deadlock. For example, simultaneous access can be granted for read-only file. A process never needs to wait for a sharable resource. In general, we cannot prevent deadlocks by denying the mutual-exclusion condition: Some resources are intrinsically nonsharable.
- **Hold and Wait:** To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. That is, a process should request all its required resources at one time and block that process until all its requests can be granted simultaneously. This approach has certain problems:
    - A process may be held up for a long time waiting for all its required resources. Actually, it would have proceeded with only few of the processes.
    - Resources allocated to one process may remain unused for some time, during which they are denied to other processes.
    - A process may not know in advance all the resources that it requires.
- **No Preemption:** This condition can be prevented in this way: If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held are preempted. In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- **Circular Wait:** The circular-wait condition can be prevented by defining a linear ordering of resources. That is, every process must request the resources in an increasing order. Let there by two processes P1 and P2. Let there are resources Ri and Rj such that i<j. Now, P1 has acquired Ri and requested Rj. And, P2 has acquired Rj and requesting Rj. This condition is impossible, because it implies i<j and j<i. But, here also, the problems seen in hold-and-wait prevention are seen.

5.3 DEADLOCK AVOIDANCE

Deadlock prevention methods seen in the previous section requires at least one condition should not hold. But, these methods result in 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. For this purpose, a simplest and most useful model is designed: each process must declare the maximum number of resources of each type that it may need. Given a priori information about the maximum number of resources of each type that may be requested for each process, it is possible to construct an algorithm that ensures that the system will never enter a deadlock state. This algorithm defines the deadlock-avoidance approach. A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist. The resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

5.3.1 Safe State

A state is **safe** if the system can allocate resources to each process 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 $<P_1, P_2, \ldots, P_n>$ is safe if for each P_i, the resources that *Pi* can still request can be satisfied by currently available resources + resources held by all the P_j, with $j<i$.

- If the needs of P_i are not immediately available, then P_i can wait until all P_j have finished.
- When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
- When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

If no such sequence exists, then the system state is said to be unsafe.

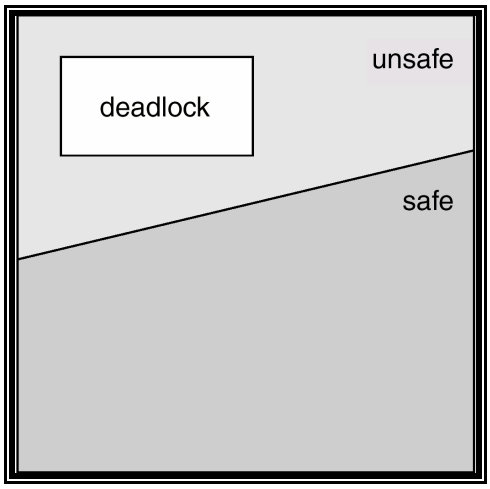


Figure 5.2 Safe, unsafe and deadlock state spaces

A safe state is not a deadlock state. A deadlock state is an unsafe state. But, not all unsafe states are deadlocks as shown in Figure 5.2. An unsafe state may lead to a deadlock. As long as the state is safe, the OS can avoid unsafe (and deadlock) states. In an unsafe state, the OS cannot prevent processes from requesting resources such that a deadlock occurs: The behavior of the processes controls unsafe states.

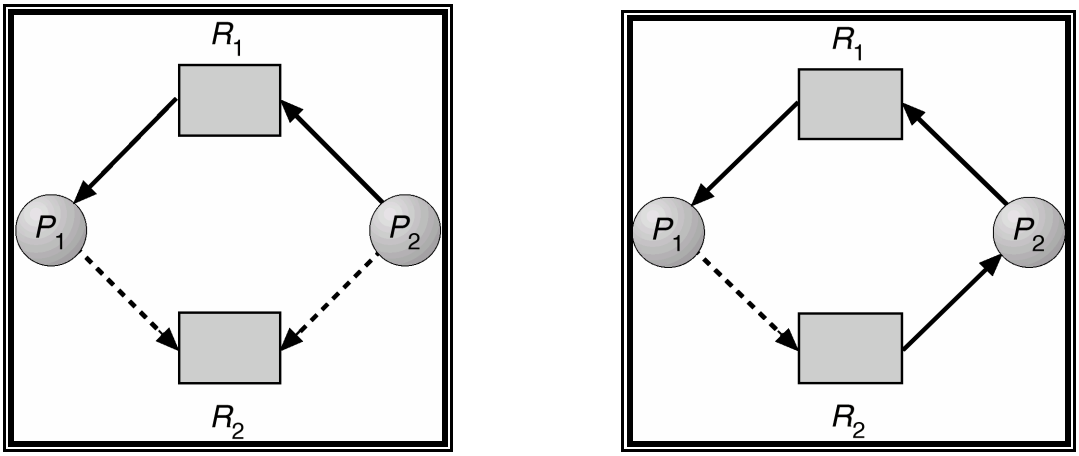
5.3.2 Resource Allocation Graph Algorithm

One of the techniques for avoiding a deadlock is using resource allocation graph with an additional edge called as ***claim edge.*** Claim *edge* $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j at some time in future. This edge is represented by a dashed line. The important steps are as below:

- When a process P_i requests a resource R_j, the claim edge $P_i \rightarrow R_j$ is converted to a request edge.
- Similarly, when a resource R_j is released by the process P_i, the assignment edge $R_j \rightarrow P_i$ is reconverted as claim edge $P_i \rightarrow R_j$.
- The request for R_j from P_i can be granted only if the converting request edge to assignment edge do not form a cycle in the resource allocation graph.

To apply this algorithm, each process P_i must know all its claims before it starts executing. If no cycle exists, then the allocation of the resource will leave the system in a safe state. If the cycle is found, system is put into unsafe state and may cause a deadlock.

An illustration: Consider a resource allocation graph shown in Figure 5.3(a). Suppose P2 requests R2. Though R2 is currently free, we cannot allocate it to P2 as this action will create a cycle in the graph as shown in Figure 5.3(b). This cycle will indicate that the system is in unsafe state: because, if P1 requests R2 and P2 requests R1 later, a deadlock will occur.



(a) For deadlock avoidance (b) an unsafe state

Figure 5.3 Resource Allocation graphs

5.3.3 Banker's Algorithm

The resource-allocation graph algorithm is not applicable when there are multiple instances for each resource. The banker's algorithm addresses this situation, but it is less efficient. The name was chosen because this algorithm could be used in a banking system to ensure that the bank never allocates its available cash such that it can no longer satisfy the needs of all its customers.

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Data structure for Banker's algorithms is as below –
Let n be the number of processes in the system and m be the number of resource types.

- ***Available****:* Vector of length *m* indicating number of available resources. If Available[*j*] = *k*, there are *k* instances of resource type R_j available.
- ***Max****:* An *n x m* matrix defines the maximum demand of each process. If *Max* [*i,j*] = *k*, then process P_i may request at most *k* instances of resource type R_j.
- ***Allocation****:* An *n x m* matrix defines the number of resources currently allocated to each process. If Allocation[*i, j*] = *k* then P_i is currently allocated *k* instances of R_j.
- ***Need****:* An *n x m* matrix indicates remaining resource need of each process. If *Need*[*i,j*] = *k*, then P_i may need *k* more instances of R_j to complete its task. Note that,

  Need [*i,j*] = *Max*[*i,j*] – *Allocation* [*i,j*].

The Banker's algorithm has two parts:

1. **Safety Algorithm:** It is for finding out whether a system is in safe state or not. The steps are as given below –
   1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:
      Work = *Available*
      Finish [*i*] = *false* for *i* = 1, 2, 3, …, *n.*
   2. Find an *i* such that both:
      (a) *Finish* [*i*] = *false*
      (b) $Need_i \leq Work$
      If no such *i* exists, go to step 4.
   3. $Work = Work + Allocation_i$
      Finish[*i*] = *true*
      go to step 2.
   4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state.

2. **Resource – Request Algorithm:** Let $Request_i$ be the request vector for process P_i. If $Request_i[j] = k$ then process P_i wants *k* instances of resource type R_j.

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state as follows:
   $Available = Available - Request_i$;
   $Allocation_i = Allocation_i + Request_i$;
   $Need_i = Need_i - Request_i$;

   If the resulting resource allocation is safe, then the transaction is complete and the process P_i is allocated its resources. If the new state is unsafe, then P_i must wait for $Request_i$, and the old resource-allocation state is restored

Example for Banker's algorithm:
Consider 5 processes P_0 through P_4 and 3 resources *A* (10 instances), *B* (5 instances), and *C* (7 instances). Snapshot at time T_0 the snapshot of the system is as given in Table 5.1.

Table 5.1 Snapshot of the system at time To

Process	Allocation (A B C)	Max (A B C)	Available (A B C)
P0	(0 1 0)	(7 5 3)	(3 3 2)
P1	(2 0 0)	(3 2 2)	
P2	(3 0 2)	(9 0 2)	
P3	(2 1 1)	(2 2 2)	
P4	(0 0 2)	(4 3 3)	

The matrix *Need* = *Max* – *Allocation*. It is given by the Table 5.2.

Table 5.2 Need Matrix

Process	Need (A B C)
P0	(7 4 3)
P1	(1 2 2)
P2	(6 0 0)
P3	(0 1 1)
P4	(4 3 1)

Table 5.3 New State

Process	Allocation (A B C)	Need (A B C)	Available (A B C)
P0	(0 1 0)	(7 4 3)	(2 3 0)
P1	(3 0 2)	(0 2 0)	
P2	(3 0 2)	(6 0 0)	
P3	(2 1 1)	(0 1 1)	
P4	(0 0 2)	(4 3 1)	

We can apply Safety algorithm to check whether the system is safe. We can find that the sequence <P1, P3, P4, P2, P0> is one of the safety sequences.

Suppose, now the process P1 makes a request (1, 0, 2). To check whether this request can be immediately granted, we can apply Resource-Request algorithm. If we assume that this request is fulfilled, the new state would be as shown in Table 5.3. Now, by checking using safety algorithm, we see that the sequence <P1, P3, P4, P0, P2> is in safe state. Hence, this request can be granted.

5.4 DEADLOCK DETECTION

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

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

Note that a detection-and-recovery scheme has some system overhead and run-time cost is more.

5.4.1 Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock detection algorithm that is similar to 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. That is, an edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs.

Consider Figure 5.4 showing a resource allocation graph and its respective wait-for graph.

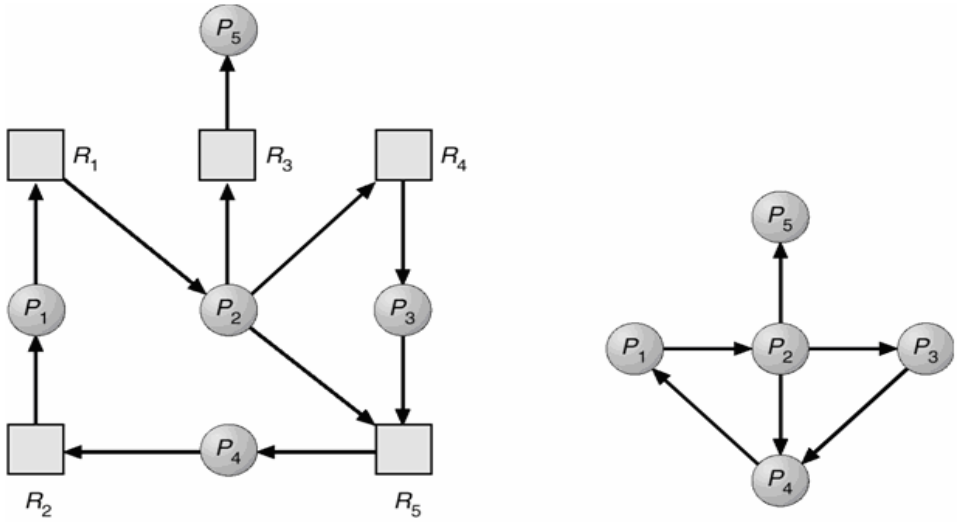


Figure 5.4 (a) Resource Allocation graph (b) corresponding wait-for graph

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 wait-for graph and periodically to invoke an algorithm that searches for a cycle in the graph.

5.4.2 Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. If resources have many instances, we use another algorithm which is similar to Banker's algorithm. The data structures used are:

- ***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] = *k*, then process P_i is requesting *k* more instances of resource type R_j.

The **detection algorithm** given below investigates every possible allocation sequence for the processes that remain to be completed.

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:
   (a) Work = Available
   (b) For *i* = 1,2, …, *n*
   if $Allocation_i \neq 0$, then
   Finish[i] = false;
   Else
   Finish[i] = *true*.

2. Find an index *i* such that both:
   (a) *Finish*[*i*] = *false*
   (b) $Request_i \leq Work$
   If no such *i* exists, go to step 4.

3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   go to step 2.
4. If *Finish*[*i*] == false, for some *i*, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked.

5.4.3 Detection Algorithm Usage

The detection algorithm should be invoked based on following factors:

- How often is a deadlock likely to occur?
- 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. So, we could invoke the deadlock detection algorithm every time a request for allocation cannot be granted immediately. In this case, we can identify not only the set of processes that is deadlocked, but also the specific process that "caused" the deadlock. Another alternative is to invoke the algorithm in periodic intervals, say, once in an hour or whenever CPU utilization drops below certain level.

5.5 RECOVERY FROM DEADLOCK

When a detection algorithm determines that a deadlock exists, several alternatives available. One possibility is to inform the operator that a deadlock has occurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.

5.5.1 Process Termination

Processes can be aborted for eliminating deadlock in two different ways:

- **Abort all deadlocked processes:** This method clearly will break the deadlock cycle. But, these processes may have computed for a long time, and the results of these partial computations must be discarded and probably recomputed later.
- **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.

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on the printer, the system must reset the printer to a correct state before printing the next job. Many factors may determine which process is chosen to abort:

- Priority of the process.
- How long process has computed, and how much longer to completion.
- How many and what type of resources the process has used.
- How many more resources process needs to complete.
- How many processes will need to be terminated?
- Is process interactive or batch?

5.5.2 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. Following issues have to be considered:

- **Selecting a victim**: Which resources and which processes are to be preempted? We must determine the order of pre-emption to minimize cost. Cost factors may include parameters like the number of resources a deadlock process is holding, and the amount of time a deadlocked process has thus far consumed during its execution.
- **Rollback:** If we preempt a resource from a process, it cannot continue its normal execution and hence we must roll back the process as far as necessary to break the deadlock. This method requires the system to keep more information about the state of all the running processes.
- **Starvation:** In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process

never completes its designated task, a starvation situation that needs to be dealt with in any practical system. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times.

5.6 AN INTEGRATED DEADLOCK STRATEGY

There are strengths and weaknesses to all of the strategies for dealing with deadlock. Rather than attempting to design an OS facility that employs only one of these strategies, it might be more efficient to use different strategies in different situations. One of the approaches could be –

- Group resources into a number of different resource classes.
- Use the linear ordering strategy for the prevention of circular wait to prevent deadlocks between resource classes.
- Within a resource class, use the algorithm that is most appropriate for that class.

The resource classes can be

- **Swappable space:** Blocks of memory on secondary storage for use in swapping processes
- **Process resources:** Assignable devices, such as tape drives, and files
- **Main memory:** Assignable to processes in pages or segments
- **Internal resources:** Such as I/O channels

5.7 DINING PHILOSOPHERS PROBLEM

The dining philosopher's problem can be explained as below: Five philosophers live in a house, where a table is laid for them. The life of each philosopher consists of thinking and eating. The food they eat is spaghetti. Each philosopher requires two forks to eat spaghetti. The eating arrangements are simple as shown in Figure 5.5. There is a round table on which is set a large serving bowl of spaghetti, five plates, one for each philosopher, and five forks. A philosopher wishing to eat goes to his or her assigned place at the table and, using the two forks on either side of the plate, takes and eats some spaghetti.

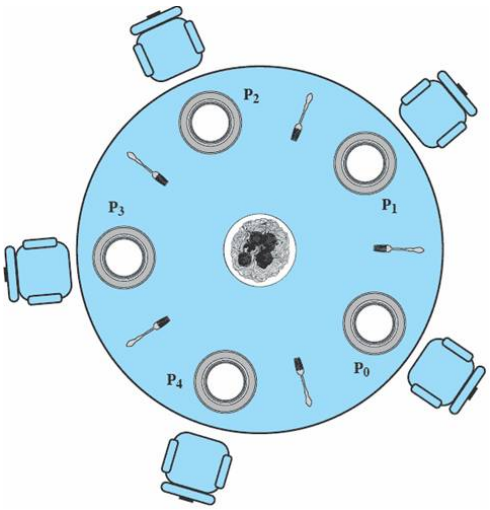


Figure 5.5 Dining arrangement for philosophers

The problem: Devise an algorithm that will allow the philosophers to eat. The algorithm must satisfy mutual exclusion (no two philosophers can use the same fork at the same time) while avoiding deadlock and starvation.

Key points of dining philosopher's problem:

- illustrates basic problems in deadlock and starvation
- reveal many of the difficulties in concurrent programming
- deals with the coordination of shared resources, which may occur when an application includes concurrent threads of execution
- it is a standard test case for evaluating approaches to synchronization

5.7.1 Solution using Semaphores

The dining philosopher's problem can be solved using semaphores as shown in the code (Figure 5.6).

Each philosopher picks up the fork on the left side first and then the fork on the right. After the philosopher is finished eating, the two forks are replaced on the table. This solution leads to deadlock: If all of the philosophers are hungry at the same time, they all sit down, they all pick up the fork on their left, and they all reach out for the other fork, which is not there. In this undignified position, all philosophers starve.

To overcome the risk of deadlock, we could buy five additional forks or teach the philosophers to eat spaghetti with just one fork. As another approach, we could consider adding an attendant who only allows four philosophers at a time into the dining room. With at most four seated philosophers, at least one philosopher will have access to two forks.

```
/* program     diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
     while (true) {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1), philosopher
(2),
          philosopher (3), philosopher (4));
     }
```

Figure 5.6 Solution to Dining Philosopher's problem

5.7.2 Solution using Monitors

Here, some condition variables are set to see that each philosopher must wait for the availability of the fork. A function (for entering a monitor) has to be written so that, a philosopher must seize two forks on his left and right sides. If anyone fork is not available, he must wait. Now, another philosopher can enter the monitor and try his luck. Another function has to be written for releasing the forks when a philosopher finishes eating. In this solution, deadlock will not occur.