LAB NO: 8 **Date:**

DEADLOCK MANAGEMENT ALGORITHMS

Objectives:

1. To understand how deadlocks occurs in a computer system.
2. To implement different algorithms for preventing or avoiding deadlocks in a computer system.

The deadlock problem:
A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

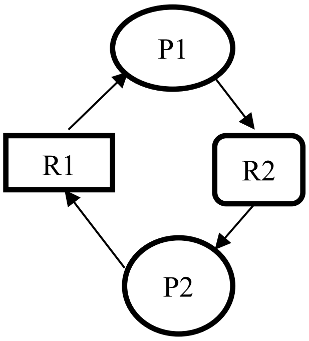


Figure 8.1: Deadlock Situation

Above Fig. 8.1 shows, a situation of deadlock where Process P1 Waiting for resource R2, which is held with process P2 and in the meantime, Process P2 is waiting for resource R1, which is held with process P1. Neither P1 nor P2 can proceed their execution until their needed resources are fulfilled forming a cyclic wait. It is the deadlock situation among processes as both are not progressed. In a single instance of resource type, a cyclic wait is always a deadlock.

Consider Figure 8.2 below, the situation with 4 processes P1, P2, P3 and P4 and 2 resources R1 and R2 both are of two instances. Here, there is no deadlock even though the cycle exists between processes P1 and P3. Once P2 finishes its job, 1 instance of resource will be available which can be accessed by process P1, which turns request edge to assignment edge, thereby removing cyclic-wait. So, in multiple instances of resource type, the cyclic-wait need not be deadlock.

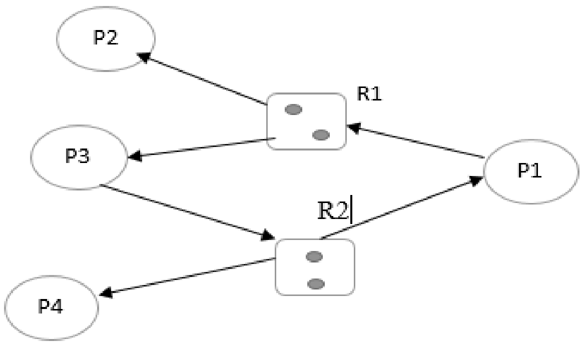


Figure 8.2: Cyclic-wait but no deadlock

Methods for Handling Deadlocks:

(i) Deadlock Avoidance:

The deadlock avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition. Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

Safe State:
System is in safe state if there exists a safe sequence of all processes. **Sequence** of processes $<P_1, P_2, \ldots, P_n>$ is **safe** if for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$.

- If P_i resource needs 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 a system is in safe state $\Rightarrow$ no deadlocks.

If a system is in unsafe state $\Rightarrow$ possibility of deadlock.

Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.

Banker's Algorithm:
Used when there exists **multiple** instances of a resource type. Each process must **declare in advance the maximum** number of instances of each resource type that it may need. When a process requests a resource, it may have to wait. When a process gets all its resources, it must return them in a finite amount of time

Data Structures for the Banker's Algorithm:
Let n = number of processes, and m = number of resources types.

Available: Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available.
Max: n x m matrix. If $Max\,[i, j] = k$, then process P_i may request at most k instances of resource type R_j.
Allocation: n x m matrix. If Allocation$[i, j] = k$ then P_i is currently allocated k instances of R_j.
Need: n x m matrix. If $Need[i, j] = k$, then P_i may need k more instances of R_j to complete its task.
$Need\,[i, j] = Max[i, j] - Allocation\,[i, j]$.

Safety Algorithm:
Allocation$_i$ means resources allocated to process P_i
Need$_i$ means resources needed for the process P_i

```
1. Let Work and Finish be vectors of length m and n, respectively.
Initialize:
Work = Available
Finish [i ] = false for i = 0,1, …, n-1.
2. Find an i such that both:
(a) Finish [i ] = false
(b) $Need_i \le Work$
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.
```

The above algorithm may require an $O(m \times n^2)$ operations to determine whether a state is safe.
Resource-Request Algorithm:
$Request_i$ = request vector for process P_i.
$Request_i\,[j] == k$ means that 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 safe $\rightarrow$ the resources are allocated to P_i.*
- *If unsafe $\rightarrow$ P_i must wait for $Request_i$, and the old resource-allocation state is restored*

(ii) Deadlock Detection:

For deadlock detection, the system must provide

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

Deadlock detection algorithm:

If a resource type can have multiple instances, then an algorithm very similar to the banker's algorithm can be used.

Required data structures:

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.

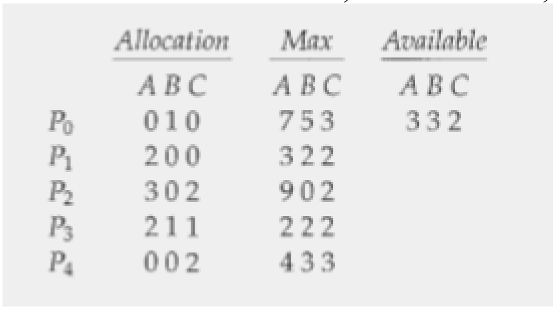
Detection Algorithm:

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize: *Work* = *Available*. For *i* = 0,2, …, *n-1*, if $Allocation_i \neq 0$, then *Finish*[i] = false; otherwise, *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*, $0 \leq i < n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then process P_i is deadlocked

The above algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state.

Lab Exercises

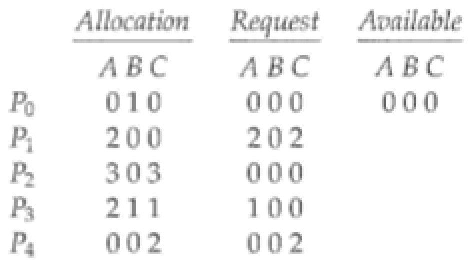
1. Consider the following snapshot of the system. Write C/C++ program to implement Banker's algorithm for deadlock avoidance. The program has to accept all inputs from the user. Assume the total number of instances of A,B and C are 10,5 and 7 respectively.



	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

(a) What is the content of the matrix *Need*?
(b) Is the system in a safe state?
(c) If a request from process P1 arrives for (1, 0, 2), can the request be granted immediately? Display the updated Allocation, Need and Available matrices.
(d) If a request from process P4 arrives for (3, 3, 0), can the request be granted immediately?
(e) If a request from process P0 arrives for (0, 2, 0), can the request be granted immediately?

2. Consider the following snapshot of the system. Write C/C++ program to implement deadlock detection algorithm.
   (a) Is the system in a safe state?
   (b) Suppose that process P2 make one additional request for instance of type C, can the system still be in a safe state?



	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Additional Exercises:

1. Write a multithreaded program that implements the banker's algorithm. Create n threads that request and release resources from the bank. The banker will grant the request only if it leaves the system in a safe state. You may write this program using either Pthreads. It is important that shared data be safe from concurrent access. To ensure safe access to shared data, you can use mutex locks, which are available in the Pthreads libraries.