# (OSD)Report on deadlock detection technique

**Deadlock :**  A set of two or more processes are deadlocked if they are blocked (i.e., in the waiting state) each holding a resource and waiting to acquire a resource held by another process in the set. Simply we can say that if it is waiting for an event which is never going to happen.

There are four conditions that are necessary to achieve deadlock:

1. **Mutual Exclusion** - At least one resource must be held in a non-sharable mode; If any other process requests this resource, then that process must wait for the resource to be released.
2. **Hold and Wait** - A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process.
3. **No preemption** - Once a process is holding a resource ( i.e. once its request has been granted ), then that resource cannot be taken away from that process until the process voluntarily releases it.
4. **Circular Wait** - A set of processes { P0, P1, P2, . . ., PN } must exist such that every P[ i ] is waiting for P[ ( i + 1 ) % ( N + 1 ) ].

**Resource-Allocation Graph**

* In some cases deadlocks can be understood more clearly through the use of **Resource-Allocation Graphs**, having the following properties:
* A set of resource categories, { R1, R2, R3, . . ., RN }, which appear as square nodes on the graph. Dots inside the resource nodes indicate specific instances of the resource. ( E.g. two dots might represent two laser printers. )
* A set of processes, { P1, P2, P3, . . ., PN }
* **Request Edges -** A set of directed arcs from Pi to Rj, indicating that process Pi has requested Rj, and is currently waiting for that resource to become available.
* **Assignment Edges -** A set of directed arcs from Rj to Pi indicating that resource Rj has been allocated to process Pi, and that Pi is currently holding resource Rj.
* Note that a **request edge** can be converted into an **assignment edge** by reversing the direction of the arc when the request is granted.
* If a resource-allocation graph contains no cycles, then the system is not deadlocked. ( When looking for cycles, remember that these are directed graphs. )
* If a resource-allocation graph does contain cycles **AND** each resource category contains only a single instance, then a deadlock exists.
* If a resource category contains more than one instance, then the presence of a cycle in the resource-allocation graph indicates the possibility of a deadlock, but does not guarantee one.

Note:

1. Create a directed graph for resource allocation.
2. This graph will have edges from processes to resources and resource to processes nodes.
3. There can be multiple edges emerging from one process to a resource. This means that many processes need same resource to make progress.
4. There would be only one emerging edge from a resource node to a process node. This would mean that this process has been assigned to a single process.
5. If there are two edges emerging out from a single resource node than that is an invalid edge as a single resource can only be used by single process at a time.

**Algorithm:**

The algorithm tries to find out a cycle in graph. If there is a cycle present in graph than it would mean that there is some cyclic dependency which is resulting into a deadlock.

In order to find out a cycle following approach is taken.

1. Pick up any random to start.
2. Now visit all nodes reachable from this node in depth first manner.
3. Keep all nodes pushing into a stack.
4. If a node is visited again which is already pushed onto a stack than it would mean that, that node is reachable from some other node in depth first manner which can only be possible if there is some loop. Hence, it would lead to detection of a cycle/deadlock.
5. Keep on marking a node as visited when a node is visited.
6. Keep repeating this algorithm for all unvisited nodes.
7. If all nodes are visited and there is no loop then it would mean that there is no cyclic dependency or no deadlock.

**Improvements:**

The program uses a stack whose maximum capacity is assigned in compilation phase. This can be made flexible which can grow/shrink as per the growing needs.

1. This program makes use of recursive program for depth first search. This can lead to stack overflow if there is a large number of process/resource in system. This can be handled if dfs is written in iterative manner and dynamically allocating memory during runtime.
2. In order to check for reachable node a loop is run. This can be improved if adjacency list is used in place of adjacency matrix.
3. The program makes use of adjacency matrix which is good if there is large number of edges in system or we can say request/allocation branches. In case these branches is less then it is better to use adjacency list to represent this graph.

**How to run:**

**Input:**

Enter number of Processes :5

Enter number of Resources :5

0 0 0 0 0 1 0 1 0 0

0 0 0 0 0 0 1 0 0 0

0 0 0 0 0 0 0 1 0 0

0 0 0 0 0 0 0 1 0 1

0 0 0 0 0 0 0 0 1 0

0 0 1 0 0 0 0 0 0 0

1 0 0 0 0 0 0 0 0 0

0 1 0 0 0 0 0 0 0 0

0 0 0 1 0 0 0 0 0 0

0 0 0 0 1 0 0 0 0 0

**Output:**

Deadlock detected : (1 -> 6 -> 3 -> 8 -> 2 -> 7 -> 1)

Deadlock detected : (4 -> 10 -> 5 -> 9 -> 4)

**For our program,the inputs have to be given one after the other in the console as shown in the screen shots.**

The first five rows are processes and the next five rows are resources.If there is an edge from process to resource we also take the possibility of a reverse edge into consideration.If there is an edge from process to resource or from resource to a process it is indicated as 1 in the corresponding place in the adjacency matrix.Otherwise,it is indicated as zero.

**How algorithm works?**

**Processes**

1 2 3 4 5

6 7 8 9 10

**Resources**

This shows the diagrammatic representation of graph given in run example. The algorithm starts from process 1 and push it into a stack. It looks for all nodes reachable from here (6, 8) in this case. Then it visits each of them one by one.

Stack : 1

It visits 6 and pushes it onto stack. And looks for another node reachable from 6 which is 3.

Stack : 1, 6

It visits 3 and pushes it onto stack. And looks for another node reachable from 3 which is 8

Stack: 1, 6, 3

It visits 8 and pushes it onto stack. And looks for another node reachable from 8 which is 2

Stack: 1, 6, 3, 8

It visits 2 and pushes it onto stack. And looks for another node reachable from 2 which is 7

Stack: 1, 6, 3, 8, 2

It visits 7 and pushes it onto stack. And looks for another node reachable from 7 which is 1

Stack: 1, 6, 3, 8, 2, 7

It visits 1 and finds that 1 is already on stack. So, that means there is some cycle or we can say a deadlock has been detected.Thus, deadlock is detected.While visiting all nodes, all nodes are marked to be visited. Later same process is repeated on all non-visited nodes.

Sample input and output Screenshots:

