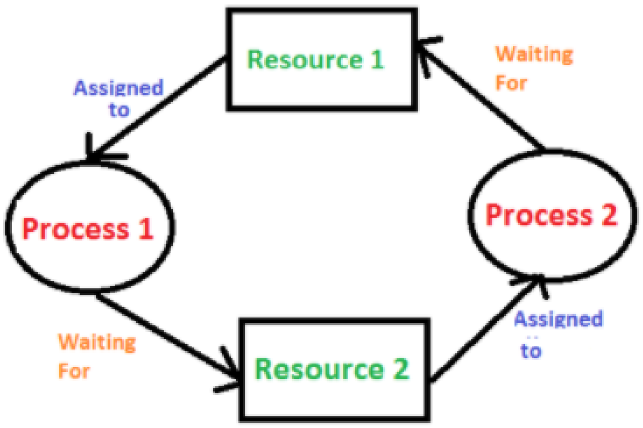
Unit-4

Deadlocks

Introduction

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process. Consider an example when two trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other. Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.



System Model

A system model or structure consists of a fixed number of resources to be circulated among some opposing processes. The resources are then partitioned into numerous types, each consisting of some specific quantity of identical instances. Memory space, CPU cycles, directories and files, I/O devices like keyboards, printers and CD-DVD drives are prime examples of resource types. When a system has 2 CPUs, then the resource type CPU got two instances.

Under the standard mode of operation, any process may use a resource in only the below-mentioned sequence:

1. Request: When the request can't be approved immediately (where the case may be when another process is utilizing the resource), then the requesting job must remain waited until it can obtain the resource.

2. Use: The process can run on the resource (like when the resource is a printer, its job/process is to print on the printer).
3. Release: The process releases the resource (like, terminating or exiting any specific process).

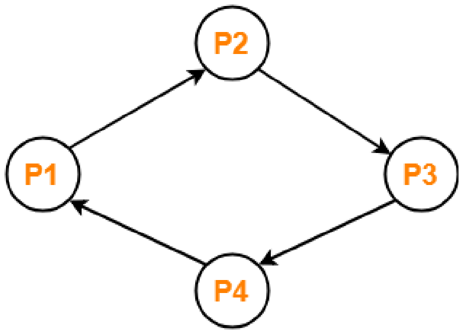
System Resources: Preemptable and Nonpreemptable Resources

A preemptable resource is one that can be taken away from the process with no ill effects. Memory is an example of a preemptable resource. On the other hand, a nonpreemptable resource is one that cannot be taken away from process (without causing ill effect). For example, CD resources are not preemptable at an arbitrary moment. Reallocating resources can resolve deadlocks that involve preemptable resources. Deadlocks that involve nonpreemptable resources are difficult to deal with.

Conditions for Resource Deadlocks

A deadlock state can occur when the following four circumstances hold simultaneously within a system:

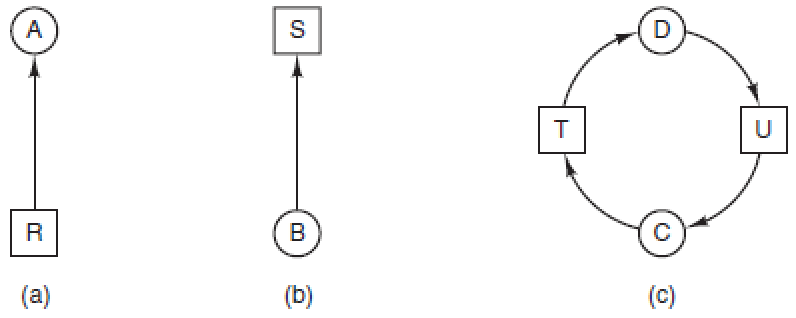
1. **Mutual exclusion:** At least there should be one resource that has to be held in a non-sharable manner; i.e., only a single process at a time can utilize the resource. If other process demands that resource, the requesting process must be postponed until the resource gets released. Printer is an example of a resource that can be used by only one process at a time.
2. **Hold and wait:** There must exist a process which holds some resource and waits for another resource held by some other process.
3. **No preemption:** Once the resource has been allocated to the process, it can not be preempted. It means resource can not be snatched forcefully from one process and given to the other process.
4. **Circular wait:** All the processes must wait for the resource in a cyclic manner where the last process waits for the resource held by the first process.



Circular Wait

Deadlock Modeling

Holt (1972) showed how these four conditions can be modeled using directed graphs. The graphs have two kinds of nodes: processes, shown as circles, and resources, shown as squares. A directed arc from a resource node (square) to a process node (circle) means that the resource has previously been requested by, granted to, and is currently held by that process. In Fig (a), resource *R* is currently assigned to process *A*.
A directed arc from a process to a resource means that the process is currently blocked waiting for that resource. In Fig.(b), process *B* is waiting for resource *S*. In Fig.(c) we see a deadlock: process *C* is waiting for resource *T*, which is currently held by process *D*. Process *D* is not about to release resource *T* because
it is waiting for resource *U*, held by *C*. Both processes will wait forever. A cycle in the graph means that there is a deadlock involving the processes and resources in the cycle (assuming that there is one resource of each kind). In this example, the cycle is *C – T – D –U –C*.



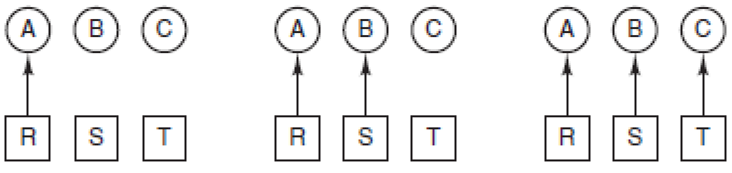
Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

The operating system is free to run any unblocked process at any instant, so it could decide to run A until A finished all its work, then run B to completion, and finally run C.
This ordering does not lead to any deadlocks (because there is no competition for resources) but it also has no parallelism at all. In addition to requesting and releasing resources, processes compute and do I/O. When the processes are run sequentially, there is no possibility that while one process is waiting for I/O, another can use the CPU. Thus, running the processes strictly sequentially may not be optimal. On the other hand, if none of the processes does any I/O at all, shortest job first is better than round robin, so under some circumstances running all processes sequentially may be the best way.

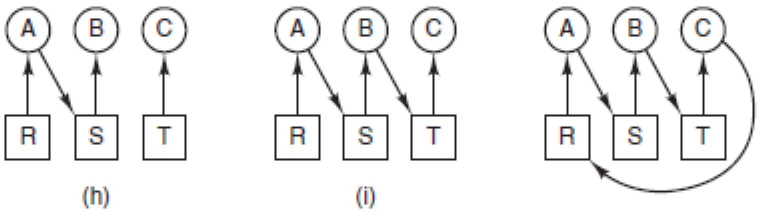
A	B	C
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R
(a)	(b)	(c)

1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
   deadlock

(d)



(e) (f) (g)

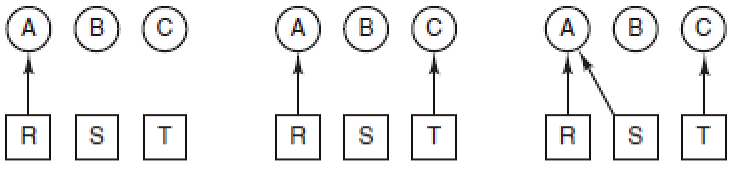


(h) (i) (j)

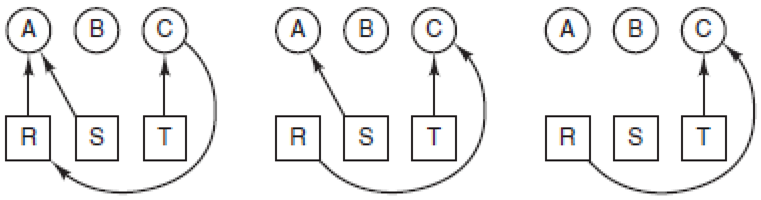
The resource requests might occur in the order of Fig.(d). If these six requests are carried out in that order, the six resulting resource graphs are as shown in Fig.(e)–(j). After request 4 has been made, *A* blocks waiting for *S*, as shown in Fig.(h). In the next two steps *B* and *C* also block, ultimately leading to a cycle and the deadlock of Fig.(j).

1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S
   no deadlock

(k)



(l) (m) (n)



(o) (p) (q)

The operating system is not required to run the processes in any special order. In particular, if granting a particular request might lead to deadlock, the operating system can simply suspend the process without granting the request (i.e., just not schedule the process) until it is safe. In above

Fig. if the operating system knew about the impending deadlock, it could suspend *B* instead of granting it *S*. By running only *A* and *C*, we would get the requests and releases of Fig.(k) instead of Fig.(d). This sequence leads to the resource graphs of Fig.(l)–(q), which do not lead to deadlock. After step (q), process *B* can be granted *S* because *A* is finished and *C* has everything it needs. Even if *B* blocks when requesting *T*, no deadlock can occur. *B* will just wait until *C* is finished.

Deadlock ignorance (Ostrich Algorithm)

When storm approaches, an ostrich puts his head in the sand (ground) and pretend (imagine) that there is no problem at all. If deadlocks occur on the average once every five years, but system crashes due to hardware failures and operating system bugs occur once a week, most engineers would not be willing to pay a large penalty in performance or convenience to eliminate deadlocks. In these types of systems, the user has to simply restart the computer in the case of deadlock. Windows and Linux are mainly using this approach.

Method of Handling Deadlocks

There are mainly four methods for handling deadlock.

1. Deadlock prevention
2. Deadlock avoidance
    - Banker's algorithm
3. Deadlock detection
    - Resource allocation graph
4. Recovery from deadlock

1. Deadlock prevention

Deadlock has following characteristics

1. Mutual Exclusion
2. Hold and Wait
3. No preemption
4. Circular wait

We can prevent Deadlock by eliminating any of the above four conditions.

Eliminate Mutual Exclusion Condition

Mutual section from the resource point of view is the fact that a resource can never be used by more than one process simultaneously which is fair enough but that is the main reason behind the deadlock. If a resource could have been used by more than one process at the same time then the process would have never been waiting for any resource.

Eliminate Hold and Wait Condition

Hold and wait condition lies when a process holds a resource and waiting for some other resource to complete its task. Deadlock occurs because there can be more than one process which are holding one resource and waiting for other in the cyclic order.

However, we have to find out some mechanism by which a process either doesn't hold any resource or doesn't wait. That means, a process must be assigned all the necessary resources before the execution starts. A process must not wait for any resource once the execution has been started.

Eliminate No-Preemption Condition

One protocol is “if a process that is holding some resources requests another resource and that resource cannot be allocated to it, then it must release all resources that are currently allocated to it”.

Another protocol is “when a process requests some resources, if they are available, allocate them. If a resource it requested is not available, then we check whether it is being used or it is allocated to some other process waiting for other resources. If that resource is not being used, then the OS preempts it from the waiting process and allocate it to the requesting process. If that resource is used, the requesting process must wait”. This protocol can be applied to resources whose states can easily be saved and restored. It cannot be applied to resources like printers.

Eliminate Circular Wait Condition

Each resource will be assigned with a numerical number. A process can request the resources increasing/decreasing. order of numbering. For Example, if P1 process is allocated R5 resources, now next tim’e if P1 ask for R4, R3 lesser than R5 such request will not be granted, only request for resources more than R5 will be granted.

2. Deadlock Avoidance

This approach to the deadlock problem anticipates deadlock before it actually occurs. This approach employs an algorithm to access the possibility that deadlock could occur and acting accordingly. This method differs from deadlock prevention, which guarantees that deadlock cannot occur by denying one of the necessary conditions of deadlock.

If the necessary conditions for a deadlock are in place, it is still possible to avoid deadlock by being careful when resources are allocated. So named because the process is analogous to that used by a banker in deciding if a loan can be safely made.

A deadlock avoidance algorithm dynamically examines the resources allocation state to ensure that a circular wait condition case never exists. Where, the resources allocation state is defined by available and allocated resources and the maximum demand of the process. There are 3 states of the system:

- Safe state
- Unsafe state
- Deadlock state

A state is *safe* if the system can allocate resources to each process (up to its maximum) 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 of processes <*P*1, *P*2, ..., *Pn*> is a safe sequence for the current allocation state if, for each *Pi*, the resource requests that *Pi* can still make can be satisfied by the currently available resources plus the resources held by all *Pj*, with *j* <*i*.

If a safe sequence does not exist, then the system is in an unsafe state, which may lead to deadlock. All safe states are deadlock free, but not all unsafe states lead to deadlocks.

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1
(b)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5
(c)

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0
(d)

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7
(e)

Fig: Demonstration that the state in (a) is safe.

In Fig.(a) we have a state in which *A* has three instances of the resource but may need as many as nine eventually. *B* currently has two and may need four altogether, later. Similarly, *C* also has two but may need an additional five. A total of 10 instances of the resource exist, so with seven resources already allocated, three there are still free.

The state of Fig. (a) is safe because there exists a sequence of allocations that allows all processes to complete. Namely, the scheduler can simply run *B* exclusively, until it asks for and gets two more instances of the resource, leading to the state of Fig.(b). When *B* completes, we get the state of Fig.(c). Then the scheduler can run *C*, leading eventually to Fig.(d). When *C* completes, we get Fig.(e). Now *A* can get the six instances of the resource it needs and also complete. Thus, the state of Fig.(a) is safe because the system, by carefull scheduling, can avoid deadlock.

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2
(b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0
(c)

	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4
(d)

Fig: Demonstration that the state in (b) is not safe.

Now suppose we have the initial state shown in Fig.(a), but this time *A* requests and gets another resource, giving Fig.(b). Can we find a sequence that is guaranteed to work? Let us try. The scheduler could run *B* until it asked for all its resources, as shown in Fig.(c). Eventually, *B* completes and we get the state of Fig.(d). At this point we are stuck. We only have four instances of the resource free, and each of the active processes needs five. There is no sequence that guarantees completion. Thus, the allocation decision that moved the system from Fig. (a) to Fig. (b) went from a safe to an unsafe state.

Method for deadlock avoidance

Banker’s Safety Algorithm

Step1. If need <=Available then

Execute process

Calculate new available as,

Available = Available + Allocation

Step2. Otherwise

Do not execute and go forward

Example 1

Assume that there are 5 processes, P0 through P4, and 4 types of resources. At T0 we have the following system state:

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	1	1	0	0	2	1	0	1	5	2	0
P_1	1	2	3	1	1	6	5	2				
P_2	1	3	6	5	2	3	6	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

1. Create the need matrix (Max-Allocation)

	A	B	C	D
P_0	0	1	0	0
P_1	0	4	2	1
P_2	1	0	0	1
P_3	0	0	2	0
P_4	0	6	4	2

2. Use the safety algorithm to test if the system is in a safe state.
   We will first define work and finish.

Work vector	Finish matrix	
1	P_0	False
5	P_1	False
2	P_2	False
0	P_3	False
	P_4	False

Check to see if need0 (0,1,0,0) is less than or equal to work. It is, so let's set finish to true for that process and also update work by adding the allocated resources (0,1,1,0) for that process to work.

Work vector	Finish matrix	
1	P_0	True
6	P_1	False
3	P_2	False
0	P_3	False
	P_4	False

Now, let's check to see if need1 (0,4,2,1) <= work. Remember that we have to check each element of the vector need1 against the corresponding element in work. Because 1 is not less than 0 (the fourth element), we need to move on to P2. Need2 (1,0,0,1) is not less than work, so must move on to P3. Need3 (0,0,2,0) is less than work, so we can update work and finish.

Work vector	Finish matrix	
1	P_0	True
12	P_1	False
6	P_2	False
2	P_3	True
	P_4	False

Next, let's look at P4. Need4 (0,6,4,2) is less than work, so we can update work and finish as follows:

Work vector	Finish matrix	
1	P_0	True
12	P_1	False
7	P_2	False
6	P_3	True
	P_4	True

Now we can go back up to P1. Need1 (0,4,2,1) is less than work, so let's update work and finish.

Work vector	Finish matrix	
2	P_0	True
14	P_1	True
10	P_2	False
7	P_3	True
	P_4	True

Finally, let’s look at P2. Need2 (1,0,0,1) is less than work, so we can then say that the system is in a safe state and the processes will be executed in the following order: P0,P3,P4,P1,P2

3. If the system is in a safe state, can the following requests be granted, why or why not? Please also run the safety algorithm on each request as necessary. a. P1 requests (2,1,1,0) We cannot grant this request, because we do not have enough available instances of resource A. b. P1 requests (0,2,1,0) There are enough available instances of the requested resources, so first let’s pretend to accommodate the request and see what the system looks like:

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	1	1	0	0	2	1	0	1	3	1	0
P_1	1	4	4	1	1	6	5	2				
P_2	1	3	6	5	2	3	6	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

Need matrix

	A	B	C	D
P_0	0	1	0	0
P_1	0	2	1	1
P_2	1	0	0	1
P_3	0	0	2	0
P_4	0	6	4	2

Now we need to run the safety algorithm:

Work vector	Finish matrix	
1	P_0	False
3	P_1	False
1	P_2	False
0	P_3	False
	P_4	False

Let’s first look at P0. Need0 (0,1,0,0) is less than work, so we change the work vector and finish matrix as follows:

Work vector	Finish matrix	
1	P_0	True
4	P_1	False
2	P_2	False
0	P_3	False
	P_4	False

Need1 (0,2,1,1) is not less than work, so we need to move on to P2.
Need2 (1,0,0,1) is not less than work, so we need to move on to P3.
Need3 (0,0,2,0) is less than or equal to work. Let's update work and finish:

Work vector	Finish matrix	
1	P_0	True
10	P_1	False
5	P_2	False
2	P_3	True
	P_4	False

Let's take a look at Need4 (0,6,4,2). This is less than work, so we can update work and finish:

Work vector	Finish matrix	
1	P_0	True
10	P_1	False
6	P_2	False
6	P_3	True
	P_4	True

We can now go back to P1. Need1 (0,2,1,1) is less than work, so work and finish can be updated:

Work vector	Finish matrix	
1	P_0	True
14	P_1	True
10	P_2	False
7	P_3	True
	P_4	True

Finally, Need2 (1,0,0,1) is less than work, so we can also accommodate this. Thus, the system is in a safe state when the processes are run in the following order: P0,P3,P4,P1,P2. We therefore can grant the resource request.

Example 2
Assume that there are three resources, A, B, and C. There are 4 processes P0 to P3. At T0 we have the following snapshot of the system:

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

1. Create the need matrix

	Need		
	A	B	C
P_0	1	1	0
P_1	3	3	2
P_2	0	1	1
P_3	0	1	0

2. Is the system in a safe state? Why or why not? In order to check this, we should run the safety algorithm. Let's create the work vector and finish matrix:

Work vector	Finish matrix	
2	P_0	False
1	P_1	False
1	P_2	False
	P_3	False

Need0 (1,1,0) is less than work, so let's go ahead and update work and finish:

Work vector	Finish matrix	
3	P_0	True
1	P_1	False
2	P_2	False
	P_3	False

Need1 (3,3,2) is not less than work, so we have to move on to P2.
Need2 (0,1,1) is less than work, let's update work and finish:

Work vector	Finish matrix	
6	P_0	True
1	P_1	False
2	P_2	True
	P_3	False

Need3 (0,1,0) is less than work, we can update work and finish:

Work vector	Finish matrix	
7	P_0	True
1	P_1	False
3	P_2	True
	P_3	True

We now need to go back to P1. Need1 (3,3,2) is not less than work, so we cannot continue. Thus, the system is not in a safe state.

Example 3

Assume that there are 5 processes, P0 through P4, and 4 types of resources. At T0 we have the following system state:

	Allocation A B C D	Max A B C D	Available A B C D
P0	0 0 1 2	0 0 1 2	1 5 2 0
P1	1 0 0 0	1 7 5 0	
P2	1 3 5 4	2 3 5 6	
P3	0 6 3 2	0 6 5 2	
P4	0 0 1 4	0 6 5 6	

1. What is the content of the matrix need?
2. Is the system in a safe state?
3. If a request from process p1 arrives for (0 4 2 0) can the request be granted immediately?

Problem with Banker's Algorithm

1. An algorithm requires fixed number of resources, some processes dynamically change the number of resources.
2. An algorithm requires the number of resources in advance, it is very difficult to predict the resources in advanced.
3. Algorithms predict all process returns within finite time, but the system does not guarantee it.

3. Deadlock Detection

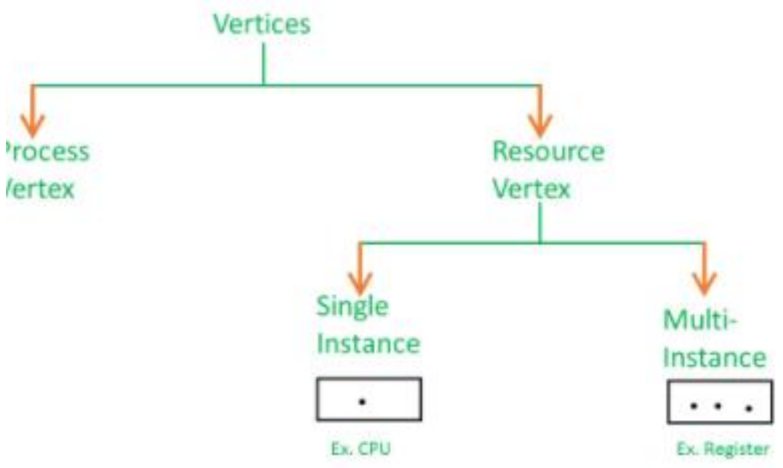
Deadlock detection is the process of actually determining that a deadlock exists and identifying the processes and resources involved in the deadlock. The basic idea is to check allocation against resource availability for all possible allocation sequences to determine if the system is in deadlocked state a. Of course, the deadlock detection algorithm is only half of this strategy. Once a deadlock is detected, there needs to be a way to recover several alternatives exists:

- Temporarily prevent resources from deadlocked processes.
- Back off a process to some check point allowing preemption of a needed resource and restarting the process at the checkpoint later.
- Successively kill processes until the system is deadlock free.

These methods are expensive in the sense that each iteration calls the detection algorithm until the system proves to be deadlock free. The complexity of algorithm is $O(N^2)$ where N is the number of proceeds. Another potential problem is starvation; same process killed repeatedly.

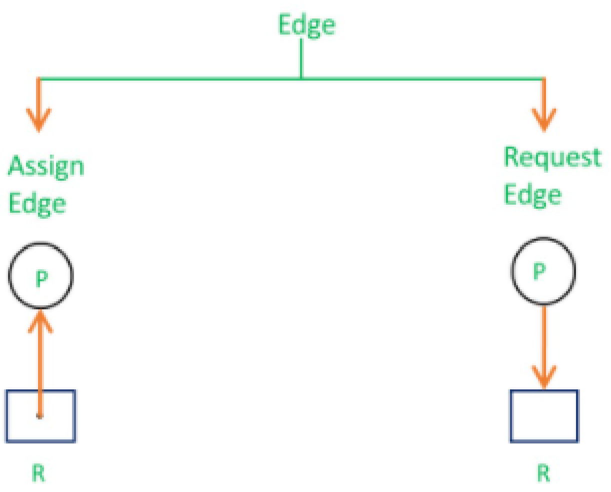
Resource allocation graph (RAG)
We use the resource allocation graph for the pictographic representation of the state of a system. The resource allocation graph contains all the information related to the processes that are holding some resources and also waiting for some more resources.
Resource allocation graph consists of all the information which is related to all the instances of the resources means the information about available resources and the resources which the process is being using.
In the Resource Allocation Graph, we use a circle to represent the process and rectangle to represent the resource. There are two components of the resource allocation graph vertices and edges.
In RAG vertices are two type –

1. **Process vertex –** Every process will be represented as a process vertex. Generally, the process will be represented with a circle.
2. **Resource vertex –** Every resource will be represented as a resource vertex. It is also two type.
   - **Single instance type resource –** It represents as a box, inside the box, there will be one dot. So the number of dots indicate how many instances are present of each resource type.
   - **Multi-resource instance type resource –** It also represents as a box, inside the box, there will be many dots present.



There are two types of edges in RAG-

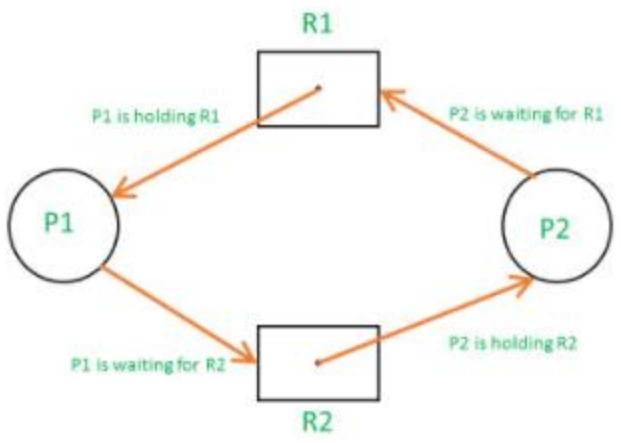
1. Assign Edge – If you already assign a resource to a process then it is called Assign edge.
2. Request Edge – It means in future the process might want some resource to complete the execution, that is called request edge.



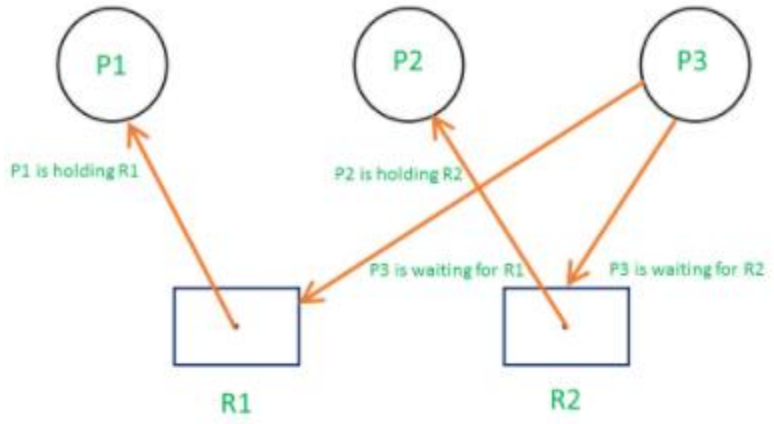
So, if a process is using a resource, an arrow is drawn from the resource node to the process node. If a process is requesting a resource, an arrow is drawn from the process node to the resource node.

Example 1 (Single instances RAG)

If there is a cycle in the Resource Allocation Graph and each resource in the cycle provides only one instance, then the processes will be in deadlock. For example, if process P1 holds resource R1, process P2 holds resource R2 and process P1 is waiting for R2 and process P2 is waiting for R1, then process P1 and process P2 will be in deadlock.



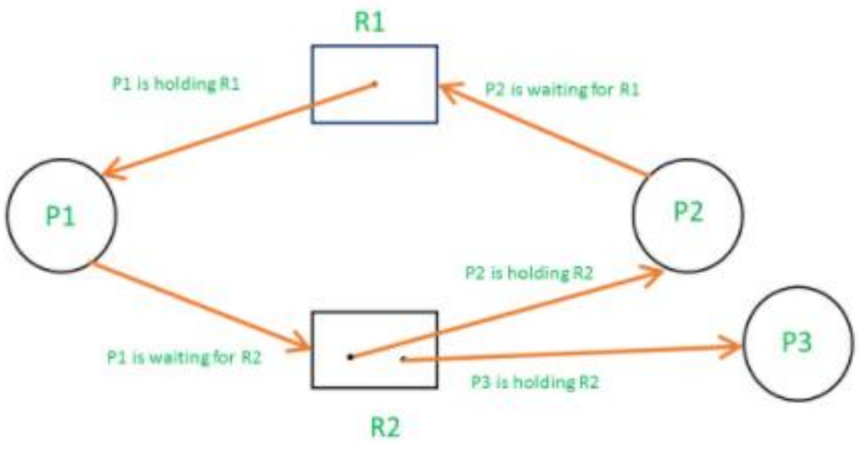
SINGLE INSTANCE RESOURCE TYPE WITH DEADLOCK



SINGLE INSTANCE RESOURCE TYPE WITHOUT DEADLOCK

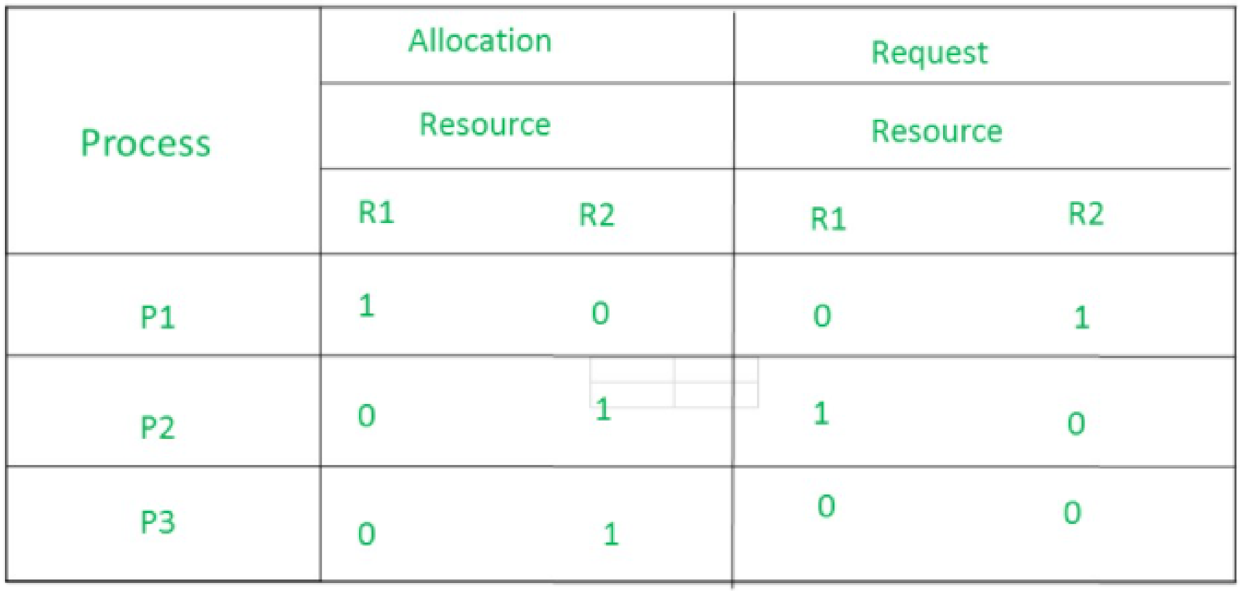
Here’s another example, that shows Processes P1 and P2 acquiring resources R1 and R2 while process P3 is waiting to acquire both resources. In this example, there is no deadlock because there is no circular dependency.So cycle in single-instance resource type is the sufficient condition for deadlock.

Example 2 (Multi-instances RAG) –



MULTI INSTANCES WITHOUT DEADLOCK

From the above example, it is not possible to say the RAG is in a safe state or in an unsafe state. So to see the state of this RAG, let’s construct the allocation matrix and request matrix.



Process	Allocation		Request	
	Resource		Resource	
	R1	R2	R1	R2
P1	1	0	0	1
P2	0	1	1	0
P3	0	1	0	0

- The total number of processes are three; P1, P2 & P3 and the total number of resources are two; R1 & R2.

Allocation matrix –

- For constructing the allocation matrix, just go to the resources and see to which process it is allocated.
- R1 is allocated to P1, therefore write 1 in allocation matrix and similarly, R2 is allocated to P2 as well as P3 and for the remaining element just write 0.

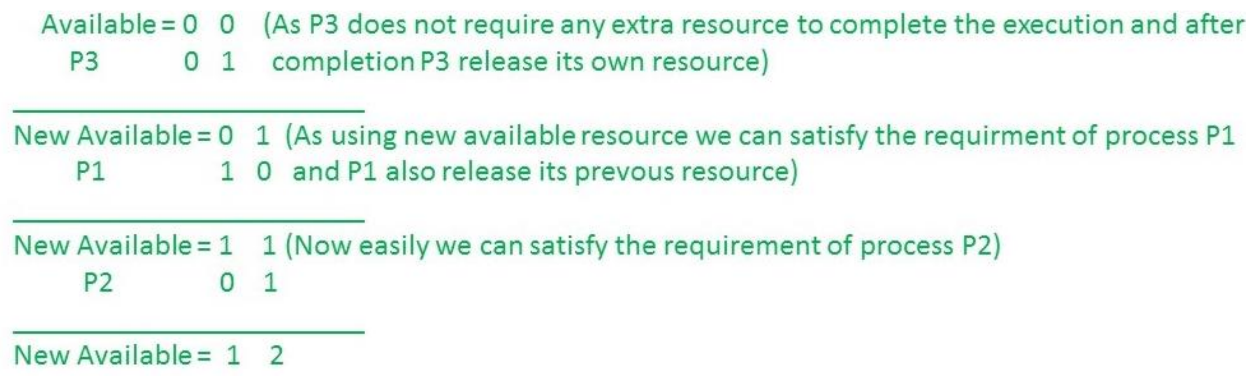
Request matrix –

- In order to find out the request matrix, you have to go to the process and see the outgoing edges.

- P1 is requesting resource R2, so write 1 in the matrix and similarly, P2 requesting R1 and for the remaining element write 0.

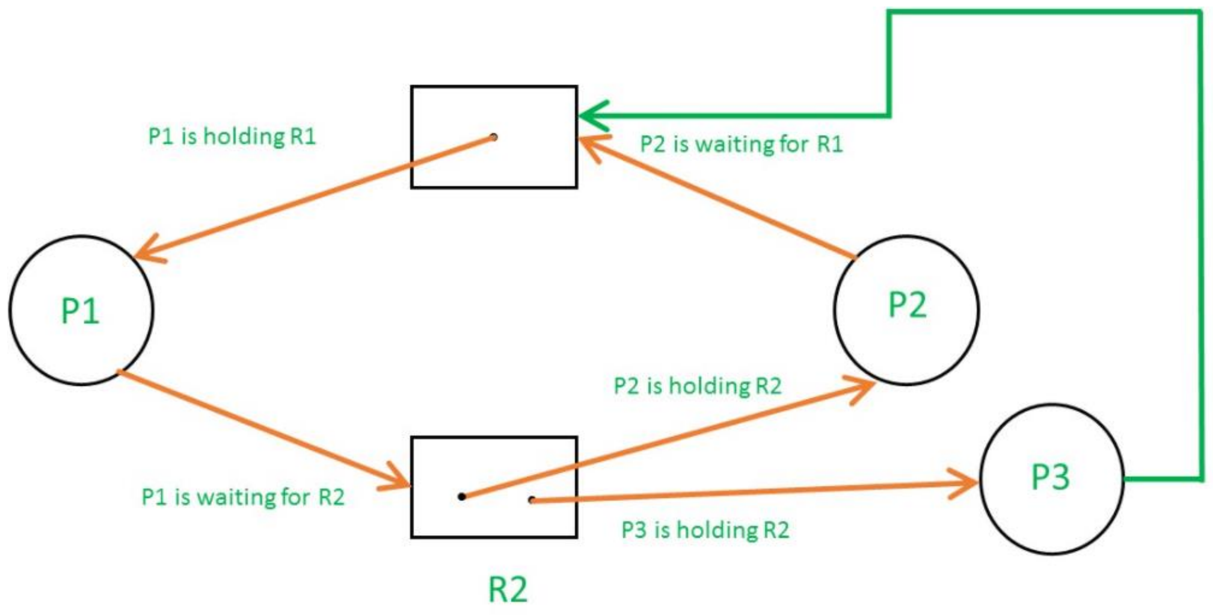
So now available resource is = (0, 0).

Checking deadlock (safe or not) –



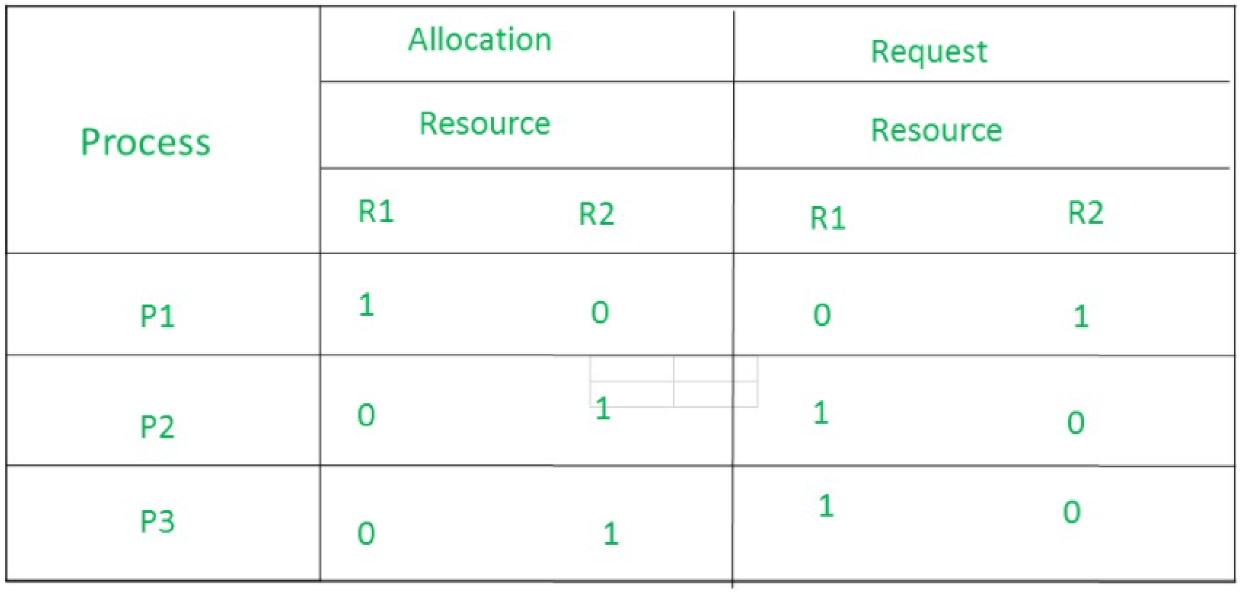
```
  Available = 0  0   (As P3 does not require any extra resource to complete the execution and after
    P3        0  1    completion P3 release its own resource)
_________________________
New Available = 0  1  (As using new available resource we can satisfy the requirment of process P1
     P1         1  0   and P1 also release its prevous resource)
_________________________
New Available = 1  1 (Now easily we can satisfy the requirement of process P2)
      P2        0  1
_________________________
New Available =  1  2
```

So, there is no deadlock in this RAG. Even though there is a cycle, still there is no deadlock. Therefore, in multi-instance resource cycle is not sufficient condition for deadlock.



MULTI INSTANCES WITH DEADLOCK

Above example is the same as the previous example except that, the process P3 requesting for resource R1. So the table becomes as shown in below.



Process	Allocation		Request	
	Resource		Resource	
	R1	R2	R1	R2
P1	1	0	0	1
P2	0	1	1	0
P3	0	1	1	0

So, the Available resource is = (0, 0), but requirement are (0, 1), (1, 0) and (1, 0).So you can't fulfill any one requirement. Therefore, it is in deadlock.

Therefore, every cycle in a multi-instance resource type graph is not a deadlock, if there has to be a deadlock, there has to be a cycle. So, in case of RAG with multi-instance resource type, the cycle is a necessary condition for deadlock, but not sufficient.

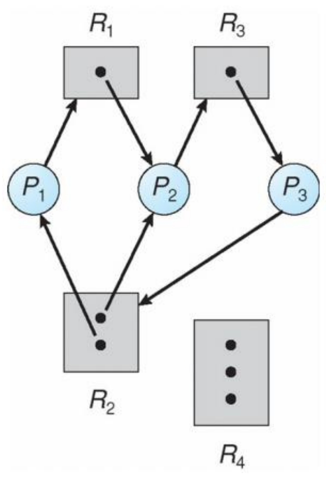


Fig: Resource allocation graph with a deadlock

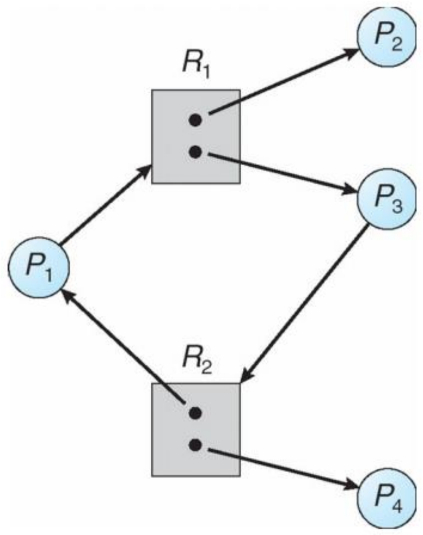


Fig: Resource allocation graph with a cycle but no deadlock

4. Deadlock Recovery

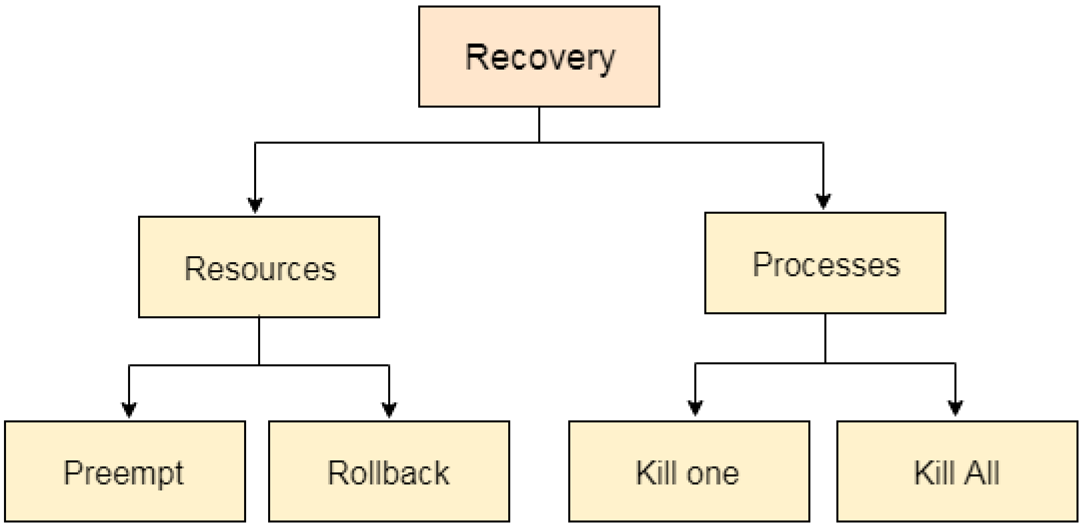
1. Process Termination:
To eliminate the deadlock, we can simply kill one or more processes. For this, we use two methods:

a. **Abort all the Deadlocked Processes:**
   Aborting all the processes will certainly break the deadlock, but with a great expenses. The deadlocked processes may have computed for a long time and the result of those partial computations must be discarded and there is a probability to recalculate them later.
b. **Abort one process at a time until deadlock is eliminated:**
   Abort one deadlocked process at a time, untill deadlock cycle is eliminated from the system. Due to this method, there may be considerable overhead, because after aborting each process, we have to run deadlock detection algorithm to check whether any processes are still deadlocked.

2. Resource Preemption:

To eliminate deadlocks using resource preemption, we preempt some resources from processes and give those resources to other processes. This method will raise three issues –

a. **Selecting a victim:**
   We must determine which resources and which processes are to be preempted and also the order to minimize the cost.
b. **Rollback:**
   We must determine what should be done with the process from which resources are preempted. One simple idea is total rollback. That means abort the process and restart it.


Recovery
Resources
Processes
Preempt
Rollback
Kill one
Kill All