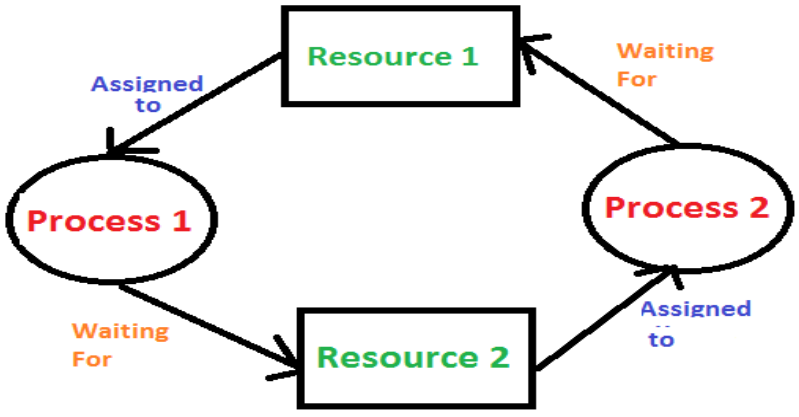
The Deadlock Problem

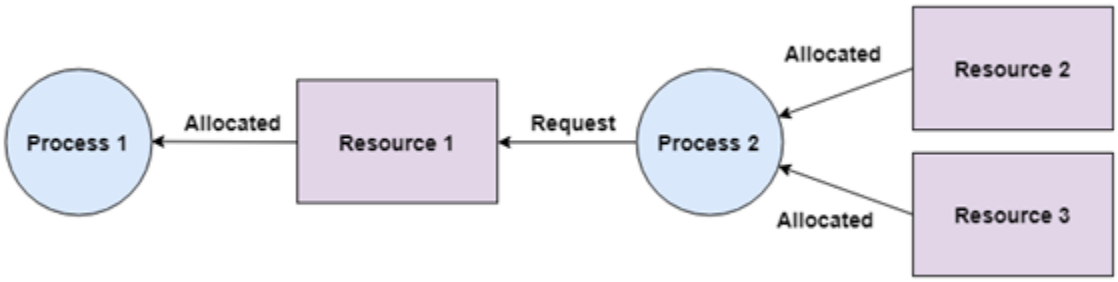
A process in operating systems uses different resources and uses resources in following way.
1) Requests a resource
2) Use the resource
2) Releases the resource

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.



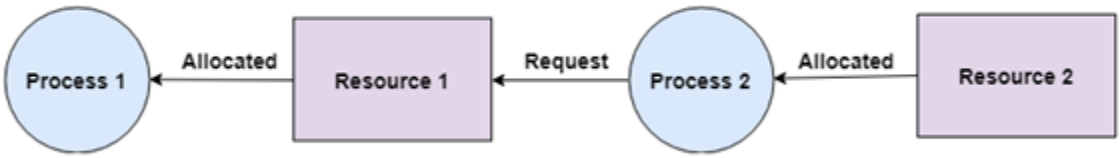
Hold and Wait

A process can hold multiple resources and still request more resources from other processes which are holding them. In the diagram given below, Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.



No Preemption

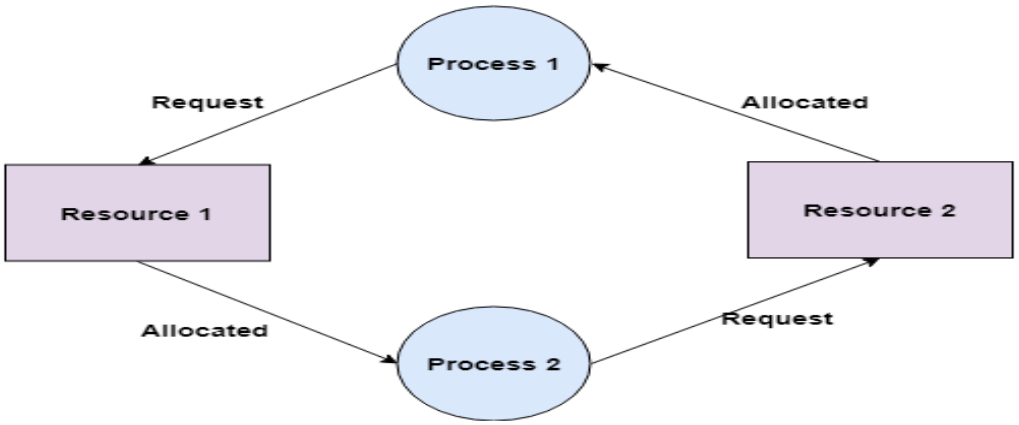
A resource cannot be preempted from a process by force. A process can only release a resource voluntarily. In the diagram below, Process 2 cannot preempt Resource 1 from Process 1. It will only be released when Process 1 relinquishes it voluntarily after its execution is complete.



Circular Wait

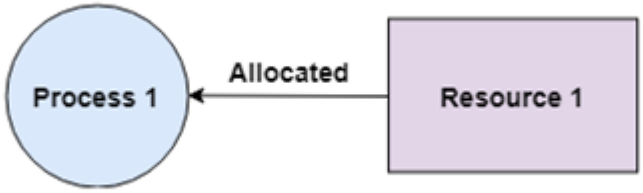
A process is waiting for the resource held by the second process, which is waiting for the resource held by the third process and so on, till the last process is waiting for a

resource held by the first process. This forms a circular chain. For example: Process 1 is allocated Resource2 and it is requesting Resource 1. Similarly, Process 2 is allocated Resource 1 and it is requesting Resource 2. This forms a circular wait loop.



Mutual Exclusion (No Sharing of Resources)

There should be a resource that can only be held by one process at a time. In the diagram below, there is a single instance of Resource 1 and it is held by Process 1 only.



Deadlock Prevention and Avoidance

Deadlock Characteristics

As discussed in the previous Section, deadlock has following characteristics.

1. Mutual Exclusion
2. Hold and Wait

3. No preemption
4. Circular wait

Deadlock Prevention

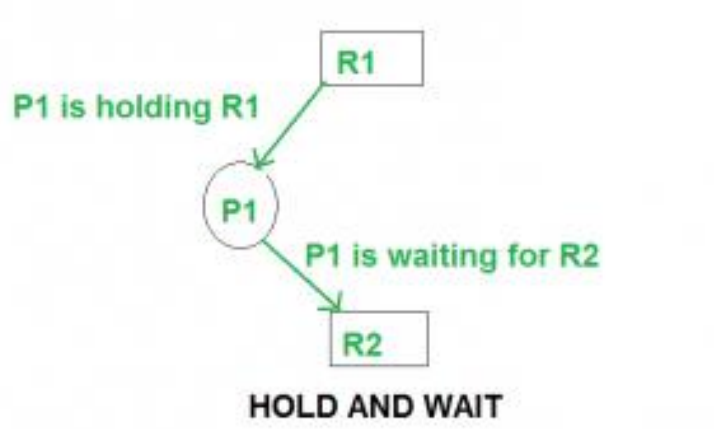
We can prevent Deadlock by eliminating any of the above four conditions.

Eliminate Mutual Exclusion

It is not possible to dis-satisfy the mutual exclusion because some resources, such as the tape drive and printer, are inherently non-shareable.

Eliminate Hold and wait

1. Allocate all required resources to the process before the start of its execution, this way hold and wait condition is eliminated but it will lead to low device utilization. for example, if a process requires printer at a later time and we have allocated printer before the start of its execution printer will remain blocked till it has completed its execution.
2. The process will make a new request for resources after releasing the current set of resources. This solution may lead to starvation.



Eliminate No Preemption
Preempt resources from the process when resources required by other high priority processes.

Eliminate Circular Wait
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 time 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.

Deadlock Avoidance

Deadlock avoidance can be done with Banker's Algorithm.

Banker's Algorithm

Bankers' Algorithm is resource allocation and deadlock avoidance algorithm which test all the request made by processes for resources, it checks for the safe state, if after granting request system remains in the safe state it

allows the request and if there is no safe state it doesn't allow the request made by the process.

Inputs to Banker's Algorithm:

1. Max need of resources by each process.
2. Currently allocated resources by each process.
3. Max free available resources in the system.

The request will only be granted under the below condition:

1. If the request made by the process is less than equal to max need to that process.
2. If the request made by the process is less than equal to the freely available resource in the system.

Example:

Total resources in system:

A B C D

6 5 7 6

Available system resources are:

A B C D

3 1 1 2

Processes (currently allocated resources):

	A	B	C	D
P1	1	2	2	1
P2	1	0	3	3
P3	1	2	1	0

Processes (maximum resources):

	A	B	C	D
P1	3	3	2	2

P2 1 2 3 4

P3 1 3 5 0

Need = maximum resources - currently allocated resources.

Processes (need resources):

   A B C D

P1 2 1 0 1

P2 0 2 0 1

P3 0 1 4 0

Note: Deadlock prevention is stricter that Deadlock Avoidance.

Detection Algorithm

Deadlock Detection Algorithm helps decide if in scenario of multi instance resources for various processes are in deadlock or not.

In cases of single resource instance we can create wait-for graph to check deadlock state. But, this we can't do for multi instance resources system.

Algorithm Example

Do not confuse the deadlock detection algorithm with Banker's algorithm which is completely different.
Learn Banker's Algorithm here.

The deadlock detection algorithm uses 3 data structures –

- Available

    - Vector of length m
    - Indicates number of available resources of each type.
- Allocation
    - Matrix of size n*m
    - A[i,j] indicates the number of j th resource type allocated to i th process.
- Request
    - Matrix of size n*m
    - Indicates request of each process.
    - Request[i,j] tells number of instance Pi process is request of jth resource type.

Steps

Step 1

1. Let **Work(vector)** length = m
2. **Finish(vector)** length = n
3. Initialize Work= Available.
    1. For i=0, 1, …., n-1, if $Allocation_i$ = 0, then Finish[i] = true; otherwise, Finish[i]= false.

Step 2

1. Find an index i such that both
    1. Finish[i] == false
    2. $Request_i$ <= Work

If no such i exists go to step 4.

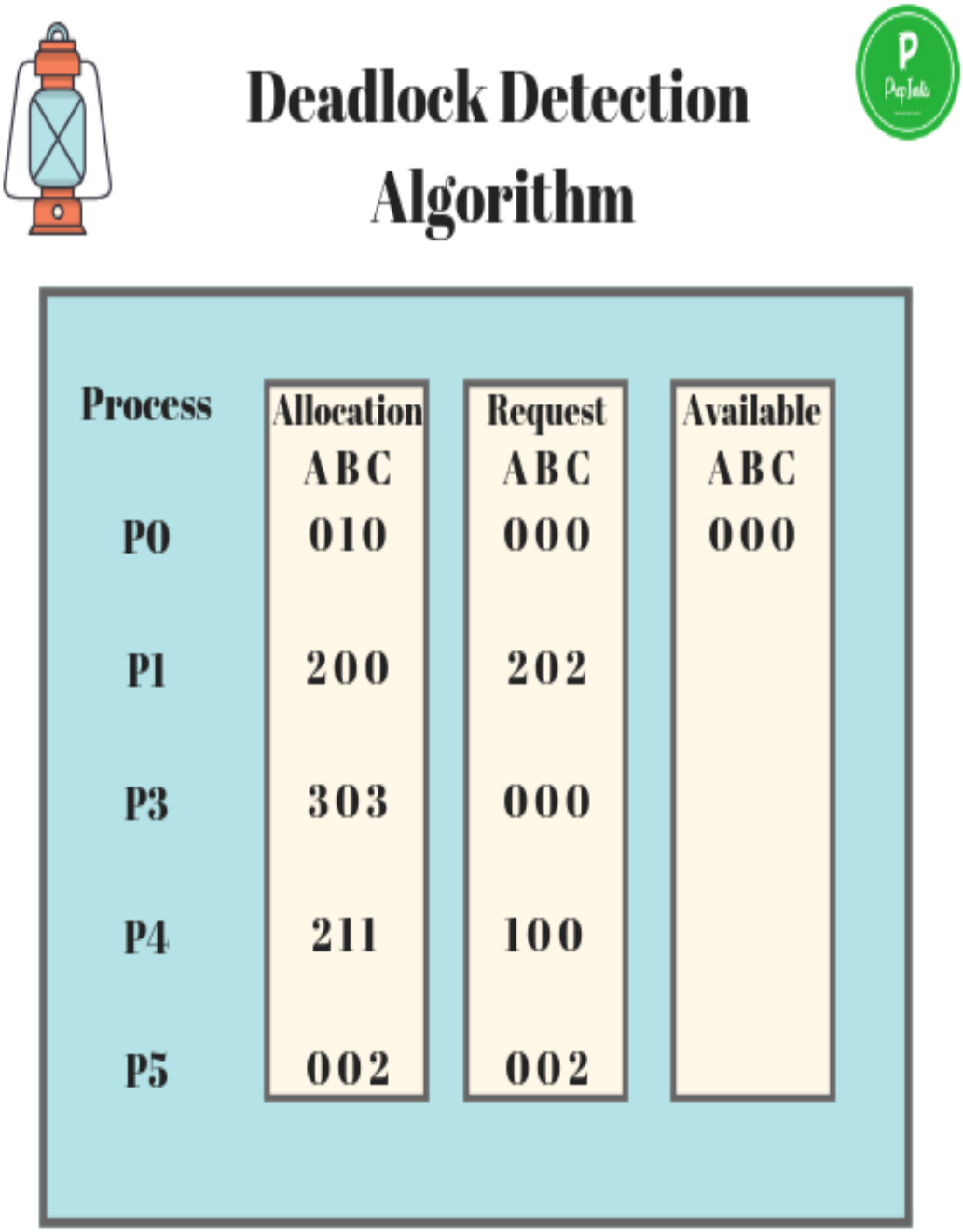
Step 3

1. Work= Work+ Allocation
2. Finish[i]= true

Go to Step 2.

Step 4

If Finish[i]== false for some i, 0<=i<n, then the system is in a deadlocked state. Moreover, if Finish[i]==false the process P_i is deadlocked.



DeadLock Avoidance Results

Step 1
In this, Work = [0, 0, 0] &
Finish = [false, false, false, false, false]

Step 2
i=0 is selected as both Finish[0] = false and [0, 0, 0]<=[0, 0, 0].

Step 3
Work =[0, 0, 0]+[0, 1, 0] =>[0, 1, 0] &
Finish = [true, false, false, false, false].

Step 4
i=2 is selected as both Finish[2] = false and [0, 0, 0]<=[0, 1, 0].

Step 5
Work =[0, 1, 0]+[3, 0, 3] =>[3, 1, 3] &
Finish = [true, false, true, false, false].

Step 6
i=1 is selected as both Finish[1] = false and [2, 0, 2]<=[3, 1, 3].

Step 7
Work =[3, 1, 3]+[2, 0, 0] =>[5, 1, 3] &
Finish = [true, true, true, false, false].

Step 8
i=3 is selected as both Finish[3] = false and [1, 0, 0]<=[5, 1, 3].

Step 9
Work =[5, 1, 3]+[2, 1, 1] =>[7, 2, 4] &
Finish = [true, true, true, true, false].

Step 10
i=4 is selected as both Finish[4] = false and [0, 0, 2]<=[7, 2, 4].

Step 11
Work =[7, 2, 4]+[0, 0, 2] =>[7, 2, 6] &
Finish = [true, true, true, true, true].

Since Finish is a vector of all true it means there is no

Concept of Fork and Join methods.

The fork-join is a basic method of expressing concurrency within a computation. This is implemented in the UNIX operating system as system calls.

The fork() call creates a new child process which runs concurrently with the parent. The child process is an exact copy of the parent except that it has a new process id. The fork() creates concurrency.

The join() is called by both the parent and the child. The child process calls join() after it has finished execution. This operation is done implicitly. The parent process waits until the child joins and continues later.

Advertisements

The join() call breaks the concurrency because the child process exits. The join() inform the parent that child operation is finished.

There are two join scenarios:

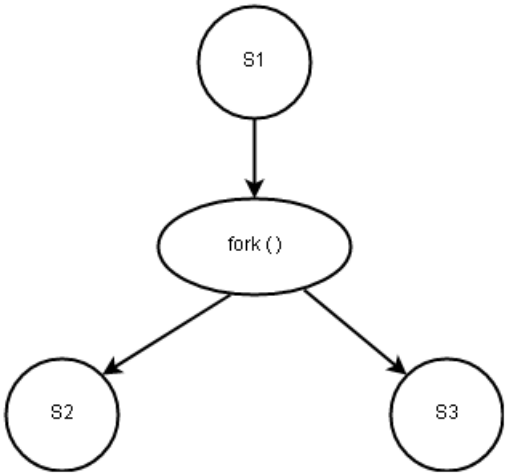
1. The child joins first and then the parent joins without waiting for any process.
2. The parent process joins first and wait, the child process joins, and the parent continues thereafter.

Example:

The parent executes "fork L" statement. It creates a child process which runs concurrently- one process (parent)

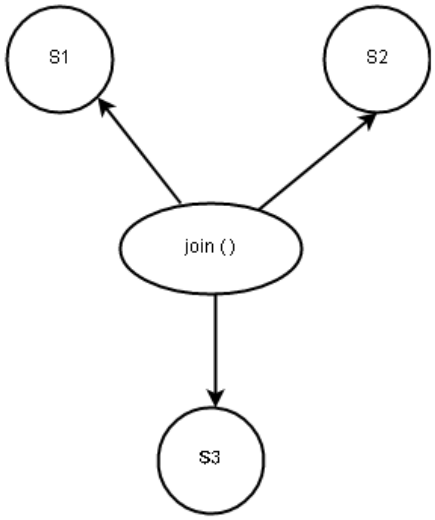
continues after the fork operation and other (child) statement runs at statement labeled L.

```
S1;

fork L;

S2     /* Statement right after fork operation */

...

...

L: S3     /* statement labeled L */
```



The fork splits the single operation into two independent operation – S2 and S3.

The join call combines the two concurrent process into one. The process which executes join first will be terminated and the other process continues. If it is parent S2 that terminates first, then it will wait for S3 to finish.



If S3 join first, then S2 will join later after finishing the execution without waiting.