**ARYAMAN MISHRA**

**19BCE1027**

1. **Deadlock Problem**

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

• Example #1: System has 2 tape drives. P1 and P2 each hold one tape drive and each needs another one.

• Example #2: semaphores A and B, initialized to 1

P0 P1

wait (A); wait (B)

wait(B) ; wait(A)

**2. Assumptions**

• Resource types R1, R2, . . ., Rm (e.g., CPU cycles, memory space, I/O devices)

• Each resource type Ri has Wi instances.

• Each process utilizes a resource as follows: o request o use o release

**Deadlock Situation**

Deadlock can arise if the following four conditions hold simultaneously:

• **Mutual exclusion**: only one process at a time can use a resource.

• **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 {P1, P2, …, Pn} of waiting processes such that P1 is waiting for a resource that is held by P2, P2 is waiting for a resource that is held by P3, …, Pn–1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P1.

1. **Method for Handling Deadlocks**

**Choice #1:** Ensure that the system will never enter a deadlock state

a. Deadlock Prevention

b. Deadlock Avoidance

**Choice #2**: Allow the system to enter a deadlock state and then recover

c. Deadlock Detection & Recovery

**Choice #3:** Ignore and don’t care

**Deadlock Prevention:** ensure that the system will never enter a deadlock state.

**Technique #1**: Ensure that “Mutual Exclusion” condition never holds Impossible

**Technique #2**: Ensure that “Hold-and-Wait” condition never holds

• Protocol #1: A new process requests every required resources before it begins execution, OR

• Protocol #2: OS allows a process to request resources only when it has no resource

• Problems: low resource utilization, starvation

**Technique #3**: Ensure that “No Preemption” condition never holds

• Protocol #1: Pi requests a resource at point in time. If the resource is not available then all resources currently being held by Pi are preempted, OR

• Protocol #2: Pi request a resource at anytime. If the resource is held by Pj waiting for another resource then preempt the resource and allocate it to Pi.

**• Problem**: for preemptable resources such as CPU registers and memory space. Not applied to such resources as printers and tape drives.

**Technique #4**: Ensure that “Circular Wait” condition never holds

• Pi can request Rj only after Ri has been allocated to Pi (note, F(Rj) > F(Ri))

• Pi release Ri before it requests Rj (note, F(Rj) >= F(Ri)).

• Problem: Resource ordering

Deadlock Avoidance: Requires that the system has some additional a priori information available.

• Requires each process to declare the maximum number of resources of each type that it may need.

• Safe State: o System is in safe state if there exists a safe sequence of all processes (i.e., the system can allocate all the requested resources in some order). o Safe Sequence for the current allocation state: , for each Pi, Pi resource request can be satisfied by currently available resources + resources held by all the Pj, with j

1. **Banker’s Algorithm**

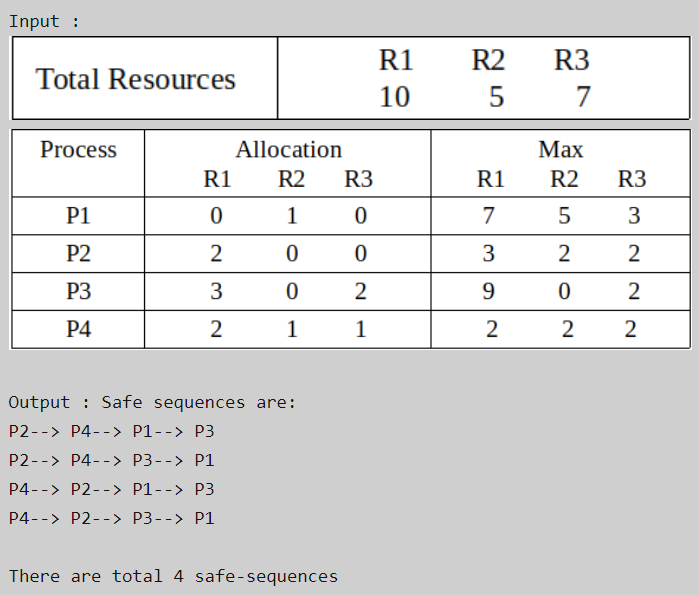
Banker’s Algorithm is a resource allocation and deadlock avoidance algorithm. This algorithm test for safety simulating the allocation for predetermined maximum possible amounts of all resources, then makes an “s-state” check to test for possible activities, before deciding whether allocation should be allowed to continue.

In simple terms, it checks if allocation of any resource will lead to deadlock or not, OR is it safe to allocate a resource to a process and if not then resource is not allocated to that process. Determining a safe sequence(even if there is only 1) will assure that system will not go into deadlock.

Banker’s algorithm is generally used to find if a safe sequence exist or not. But here we will determine the total number of safe sequences and print all safe sequences.

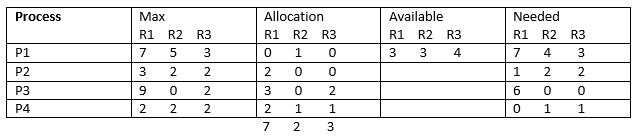
The data structure used are:

* Available vector
* Max Matrix
* Allocation Matrix
* Need Matrix

****

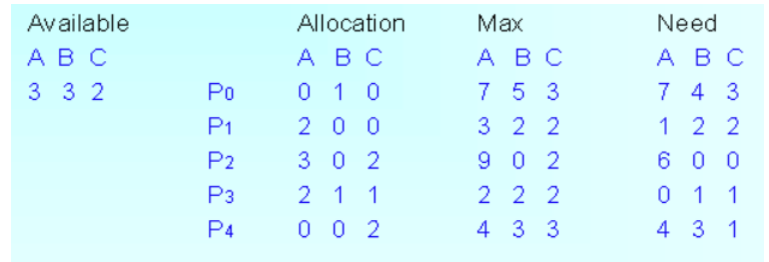
**Explanation –**  
Total resources are R1 = 10, R2 = 5, R3 = 7 and allocated resources are R1 = (0+2+3+2 =) 7, R2 = (1+0+0+1 =) 2, R3 = (0+0+2+1 =) 3. Therefore, remaining resources are R1 = (10 – 7 =) 3, R2 = (5 – 2 =) 3, R3 = (7 – 3 =) 4.

Remaining available = Total resources – allocated resources  
and  
Remaining need = max – allocated



So, we can start from either P2 or P4. We can not satisfy remaining need from available resources of either P1 or P3 in first or second attempt step of Banker’s algorithm. There are only four possible safe sequences. These are:  
P2–>P4–>P1–>P3  
P2–>P4–>P3–>P1  
P4–>P2–>P1–>P3  
P4–> P2–> P3–> P1

Write a C Program to find all possible safe sequences for this set of process.



#include <stdio.h>

int main()

{

int n, m, i, j, k;

n = 5;

m = 3;

int alloc[5][3] = { { 0, 1, 0 },

{ 2, 0, 0 },

{ 3, 0, 2 },

{ 2, 1, 1 },

{ 0, 0, 2 } };

int max[5][3] = { { 7, 5, 3 },

{ 3, 2, 2 },

{ 9, 0, 2 },

{ 2, 2, 2 },

{ 4, 3, 3 } };

int avail[3] = { 3, 3, 2 };

int f[n], ans[n], ind = 0;

for (k = 0; k < n; k++) {

f[k] = 0;

}

int need[n][m];

for (i = 0; i < n; i++) {

for (j = 0; j < m; j++)

need[i][j] = max[i][j] - alloc[i][j];

}

int y = 0;

for (k = 0; k < 5; k++) {

for (i = 0; i < n; i++) {

if (f[i] == 0) {

int flag = 0;

for (j = 0; j < m; j++) {

if (need[i][j] > avail[j]){

flag = 1;

break;

}

}

if (flag == 0) {

ans[ind++] = i;

for (y = 0; y < m; y++)

avail[y] += alloc[i][y];

f[i] = 1;

}

}

}

}

printf("Following is the SAFE Sequence\n");

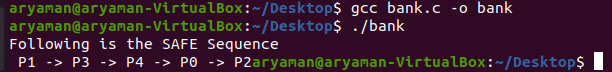
for (i = 0; i < n - 1; i++)

printf(" P%d ->", ans[i]);

printf(" P%d", ans[n - 1]);

return (0);

}



1. **Resource Allocation Graph**

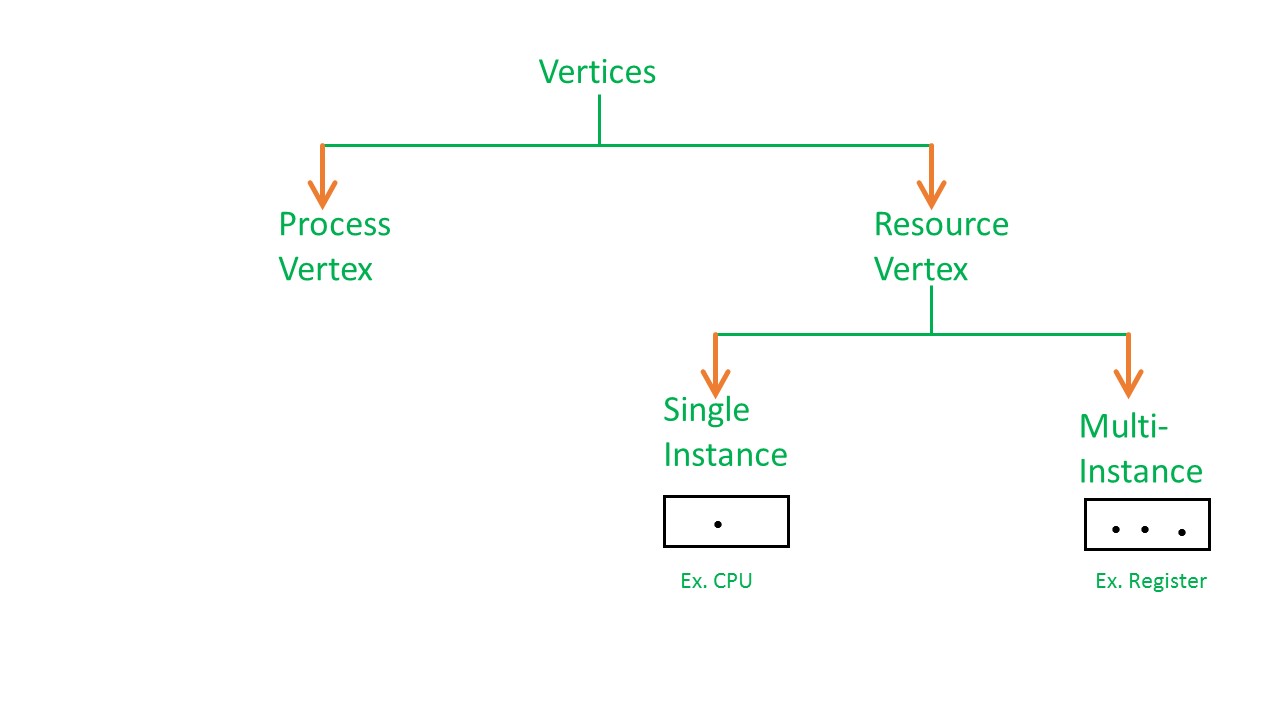
As [Banker’s algorithm](https://www.geeksforgeeks.org/operating-system-bankers-algorithm/) using some kind of table like allocation, request, available all that thing to understand what is the state of the system. Similarly, if you want to understand the state of the system instead of using those table, actually tables are very easy to represent and understand it, but then still you could even represent the same information in the graph. That graph is called **Resource Allocation Graph (RAG)**.

So, resource allocation graph is explained to us what is the state of the system in terms of **processes and resources**. Like how many resources are available, how many are allocated and what is the request of each process. Everything can be represented in terms of the diagram. One of the advantages of having a diagram is, sometimes it is possible to see a deadlock directly by using RAG, but then you might not be able to know that by looking at the table. But the tables are better if the system contains lots of process and resource and Graph is better if the system contains less number of process and resource.  
We know that any graph contains vertices and edges. So RAG also contains 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 –

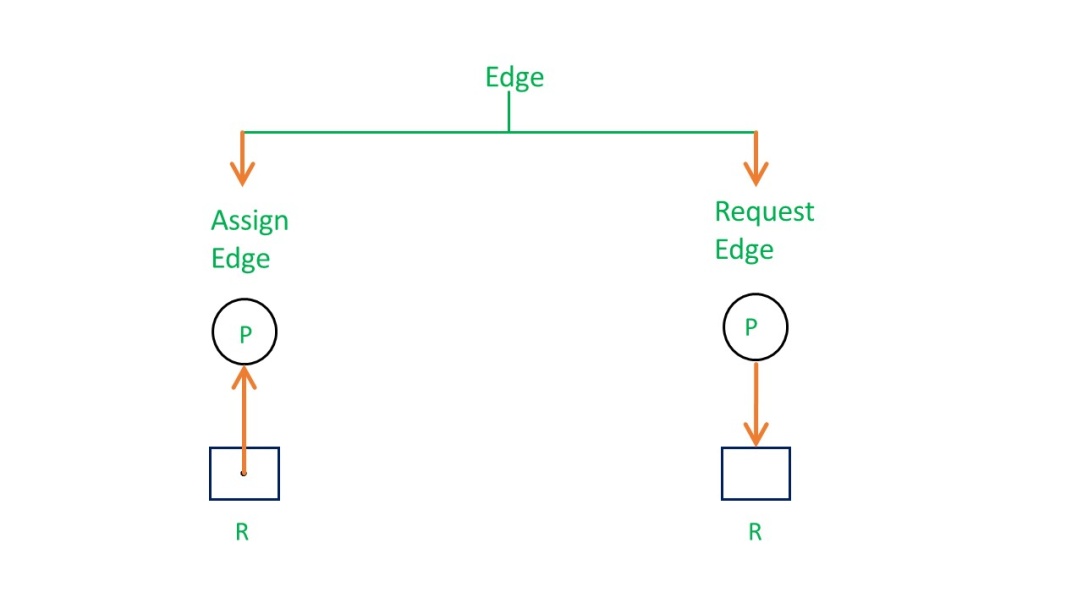
**Single instance type resource –** It represents as a box, inside the box, there will be one dot.So the number of dots indicate how many instances are present of each resource type.

**Multi-resource instance type resource –** It also represents as a box, inside the box, there will be many dots present.



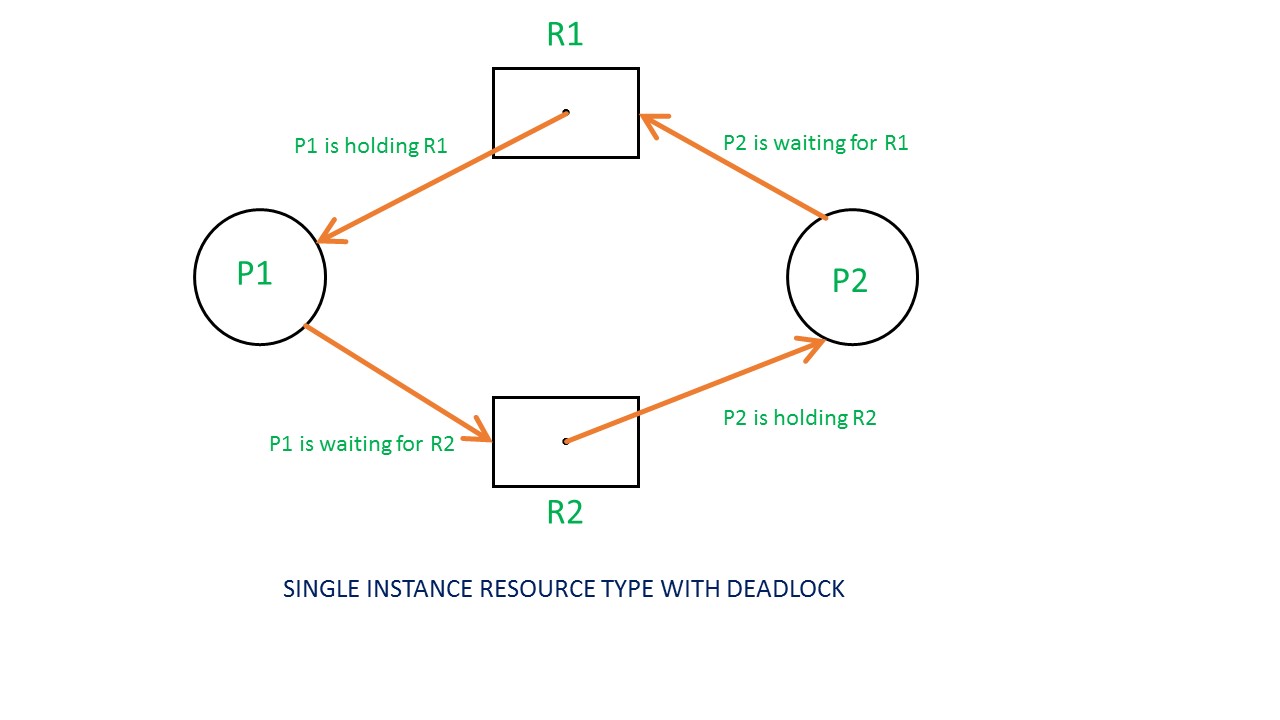
Now coming to the edges of RAG.There are two types of edges in RAG –

**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.

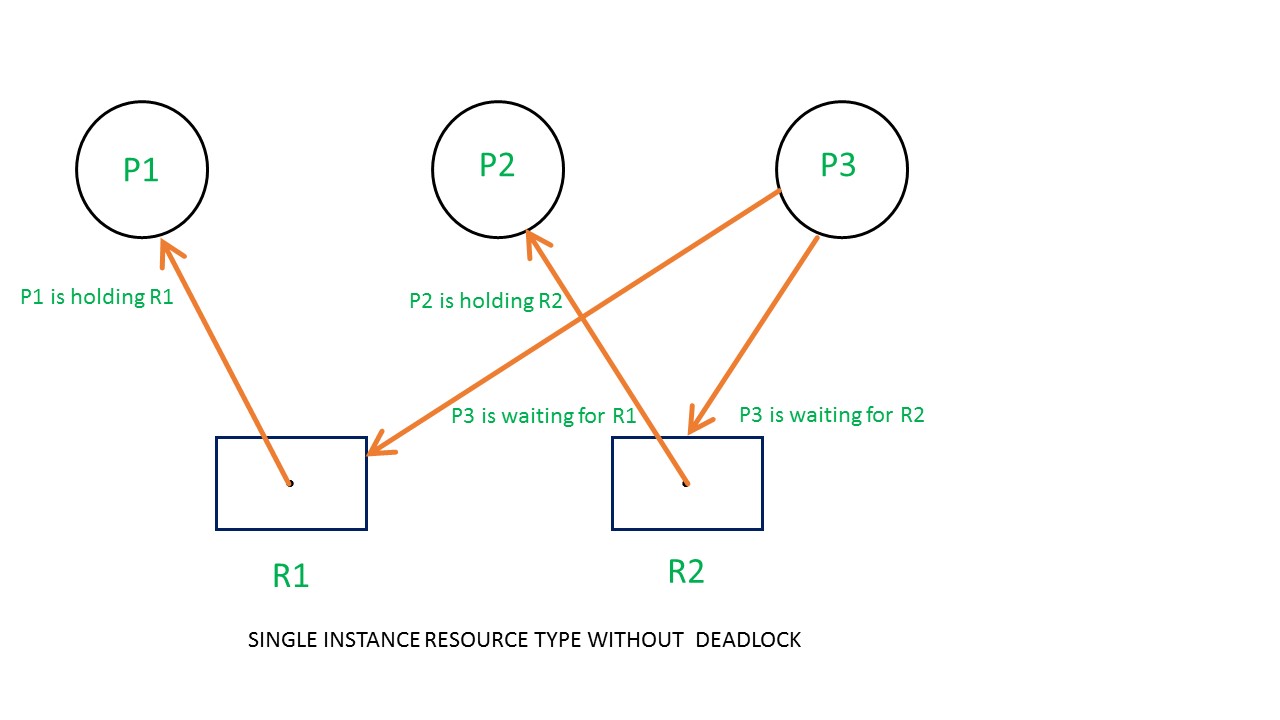


So, if a process is using a resource, an arrow is drawn from the resource node to the process node. If a process is requesting a resource, an arrow is drawn from the process node to the resource node.

* **Example 1 (Single instances RAG) –**

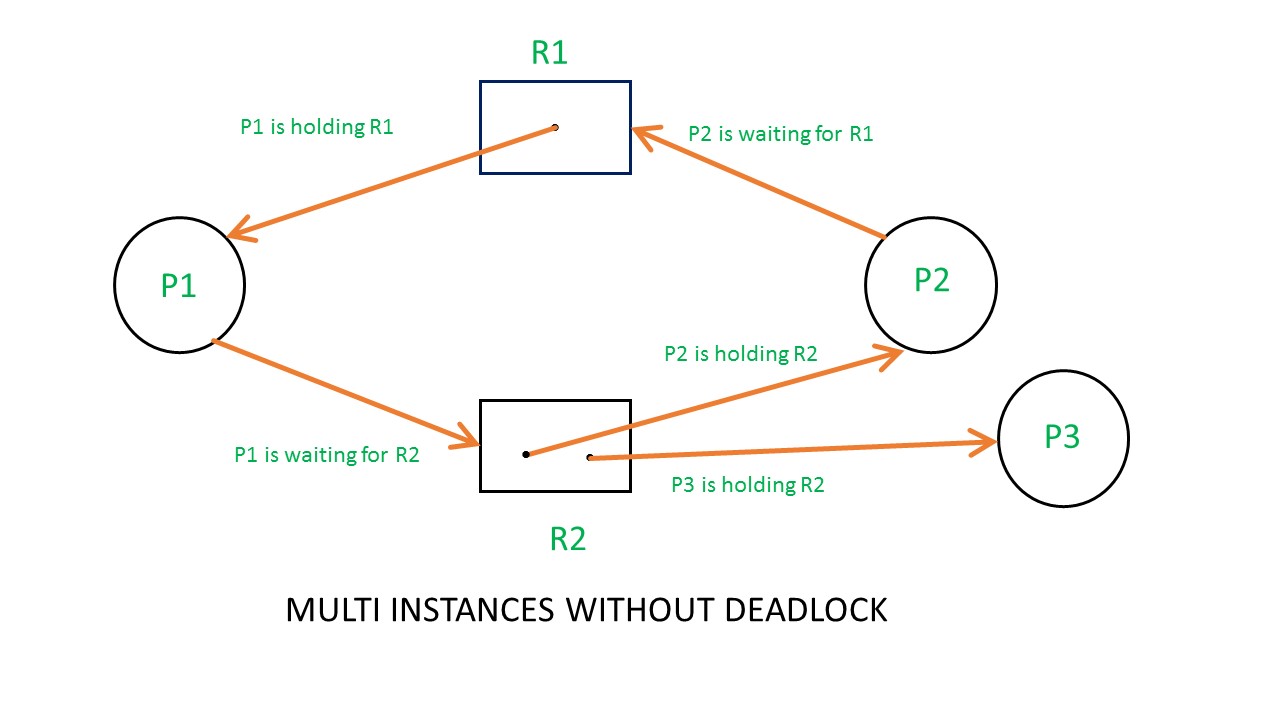


* If there is a cycle in the Resource Allocation Graph and each resource in the cycle provides only one instance, then the processes will be in deadlock. For example, if process P1 holds resource R1, process P2 holds resource R2 and process P1 is waiting for R2 and process P2 is waiting for R1, then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

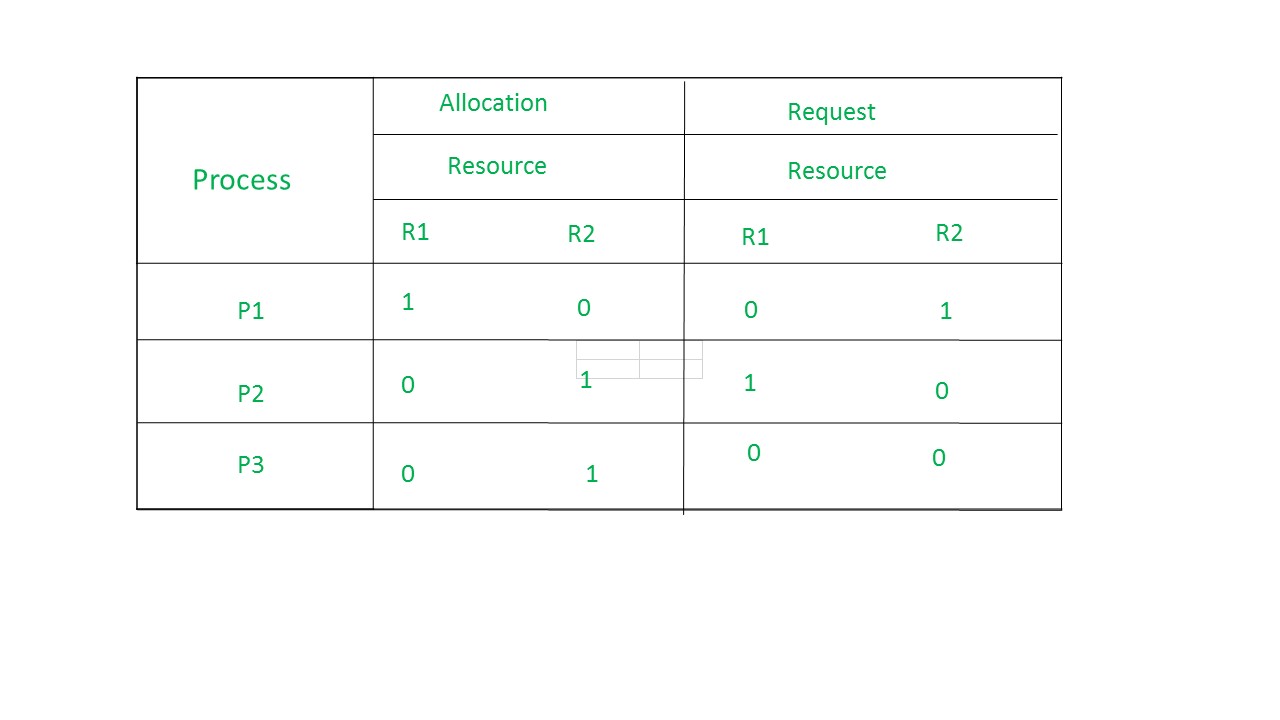


Here’s another example, that shows Processes P1 and P2 acquiring resources R1 and R2 while process P3 is waiting to acquire both resources. In this example, there is no deadlock because there is no circular dependency.  
So cycle in single-instance resource type is the sufficient condition for deadlock.

* **Example 2 (Multi-instances RAG) –**



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.



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.

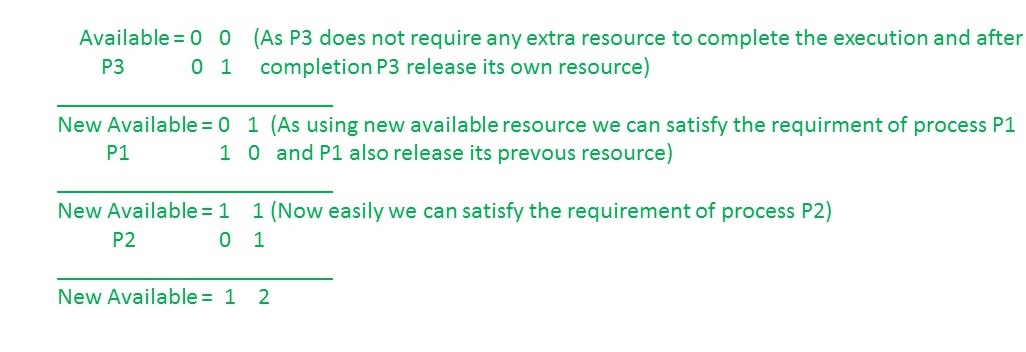
**Request matrix –**

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

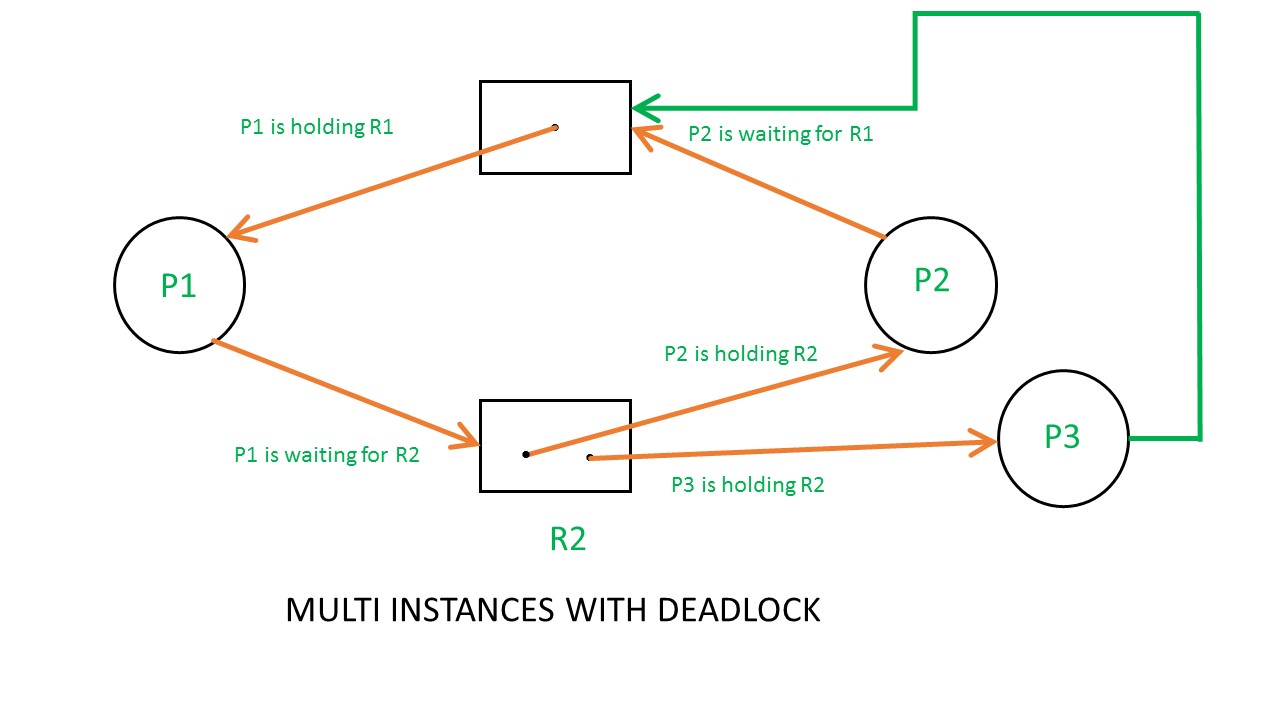
P1 is requesting resource R2, so write 1 in the matrix and similarly, P2 requesting R1 and for the remaining element write 0.

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

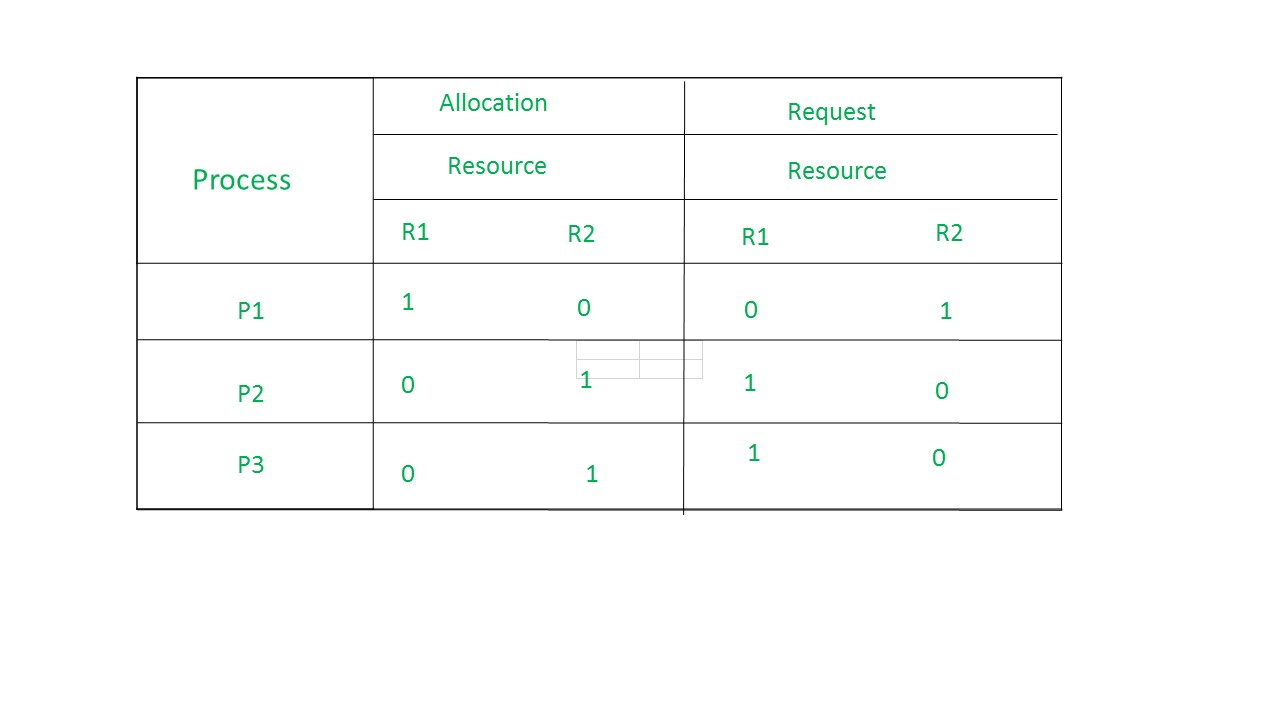
**Checking deadlock (safe or not) –**



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.



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.



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.

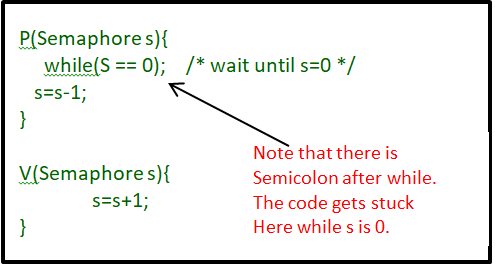
Therefore, every cycle in a multi-instance resource type graph is not a deadlock, if there has to be a deadlock, there has to be a cycle.So, in case of RAG with multi-instance resource type, the cycle is a necessary condition for deadlock, but not sufficient.

1. **Process Semaphores**

Semaphore was proposed by Dijkstra in 1965 which is a very significant technique to manage concurrent processes by using a simple integer value, which is known as a semaphore. Semaphore is simply a variable that is non-negative and shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.   
Semaphores are of two types:

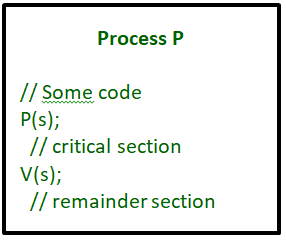
1. **Binary Semaphore –**   
   This is also known as mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes.
2. **Counting Semaphore –**   
   Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Now let us see how it does so.  
First, look at two operations that can be used to access and change the value of the semaphore variable.

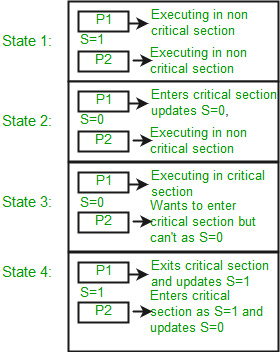


**Some point regarding P and V operation**

1. P operation is also called wait, sleep, or down operation, and V operation is also called signal, wake-up, or up operation.
2. Both operations are atomic and semaphore(s) is always initialized to one. Here atomic means that variable on which read, modify and update happens at the same time/moment with no pre-emption i.e. in-between read, modify and update no other operation is performed that may change the variable.
3. A critical section is surrounded by both operations to implement process synchronization. See the below image. The critical section of Process P is in between P and V operation.



Now, let us see how it implements mutual exclusion. Let there be two processes P1 and P2 and a semaphore s is initialized as 1. Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0. Now if P2 wants to enter its critical section then it will wait until s > 0, this can only happen when P1 finishes its critical section and calls V operation on semaphore s. This way mutual exclusion is achieved. Look at the below image for details which is Binary semaphore.



**Implementation of binary semaphores:**

**CPP**

|  |
| --- |
| struct semaphore {      enum value(0, 1);        // q contains all Process Control Blocks (PCBs)      // corresponding to processes got blocked      // while performing down operation.      Queue<process> q;    } P(semaphore s)  {      if (s.value == 1) {          s.value = 0;      }      else {          // add the process to the waiting queue          q.push(P)              sleep();      }  }  V(Semaphore s)  {      if (s.q is empty) {          s.value = 1;      }      else {            // select a process from waiting queue          s.value = 1;          Process p=q.pop();          wakeup(p);      }  } |

The description above is for binary semaphore which can take only two values 0 and 1 and ensure mutual exclusion. There is one other type of semaphore called counting semaphore which can take values greater than one.

Now suppose there is a resource whose number of instances is 4. Now we initialize S = 4 and the rest is the same as for binary semaphore. Whenever the process wants that resource it calls P or waits for function and when it is done it calls V or signal function. If the value of S becomes zero then a process has to wait until S becomes positive. For example, Suppose there are 4 processes P1, P2, P3, P4, and they all call wait operation on S(initialized with 4). If another process P5 wants the resource then it should wait until one of the four processes calls the signal function and the value of semaphore becomes positive.

**Limitations :**

1. One of the biggest limitations of semaphore is priority inversion.
2. Deadlock, suppose a process is trying to wake up another process which is not in a sleep state. Therefore, a deadlock may block indefinitely.
3. The operating system has to keep track of all calls to wait and to signal the semaphore.

**Problem in this implementation of semaphore :**  
Whenever any process waits then it continuously checks for semaphore value (look at this line while (s==0); in P operation) and waste CPU cycle. To avoid this another implementation is provided below.  
**Implementation of counting semaphore :**

**CPP**

|  |
| --- |
| struct Semaphore {      int value;        // q contains all Process Control Blocks(PCBs)      // corresponding to processes got blocked      // while performing down operation.      Queue<process> q;    } P(Semaphore s)  {      s.value = s.value - 1;      if (s.value < 0) {            // add process to queue          // here p is a process which is currently executing          q.push(p);          block();      }      else          return;  }    V(Semaphore s)  {      s.value = s.value + 1;      if (s.value <= 0) {            // remove process p from queue          Process p=q.pop();          wakeup(p);      }      else          return;  } |

In this implementation whenever the process waits it is added to a waiting queue of processes associated with that semaphore. This is done through system call block() on that process. When a process is completed it calls the signal function and one process in the queue is resumed. It uses wakeup() system call.