CS343: Operating System

Deadlock: Avoidance, Prevention, Detection and Recovery

Lect25 : 05th Oct 2023

Dr. A. Sahu

Dept of Comp. Sc. & Engg.

Indian Institute of Technology Guwahati

Outline

- Deadlock
  - Conditions (Why deadlock happens)
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection and Recovery

Resource-Allocation Graph

A set of vertices V and a set of edges E.

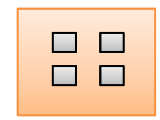
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge

$$R_j \rightarrow P_i$$

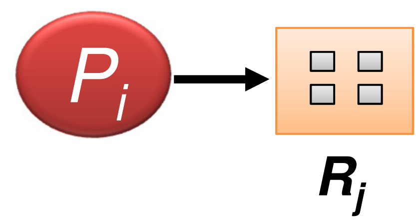
Resource-Allocation Graph (Cont.)

- Process

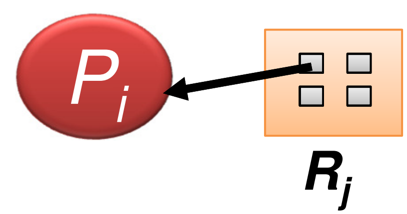
- Resource Type with 4 instances



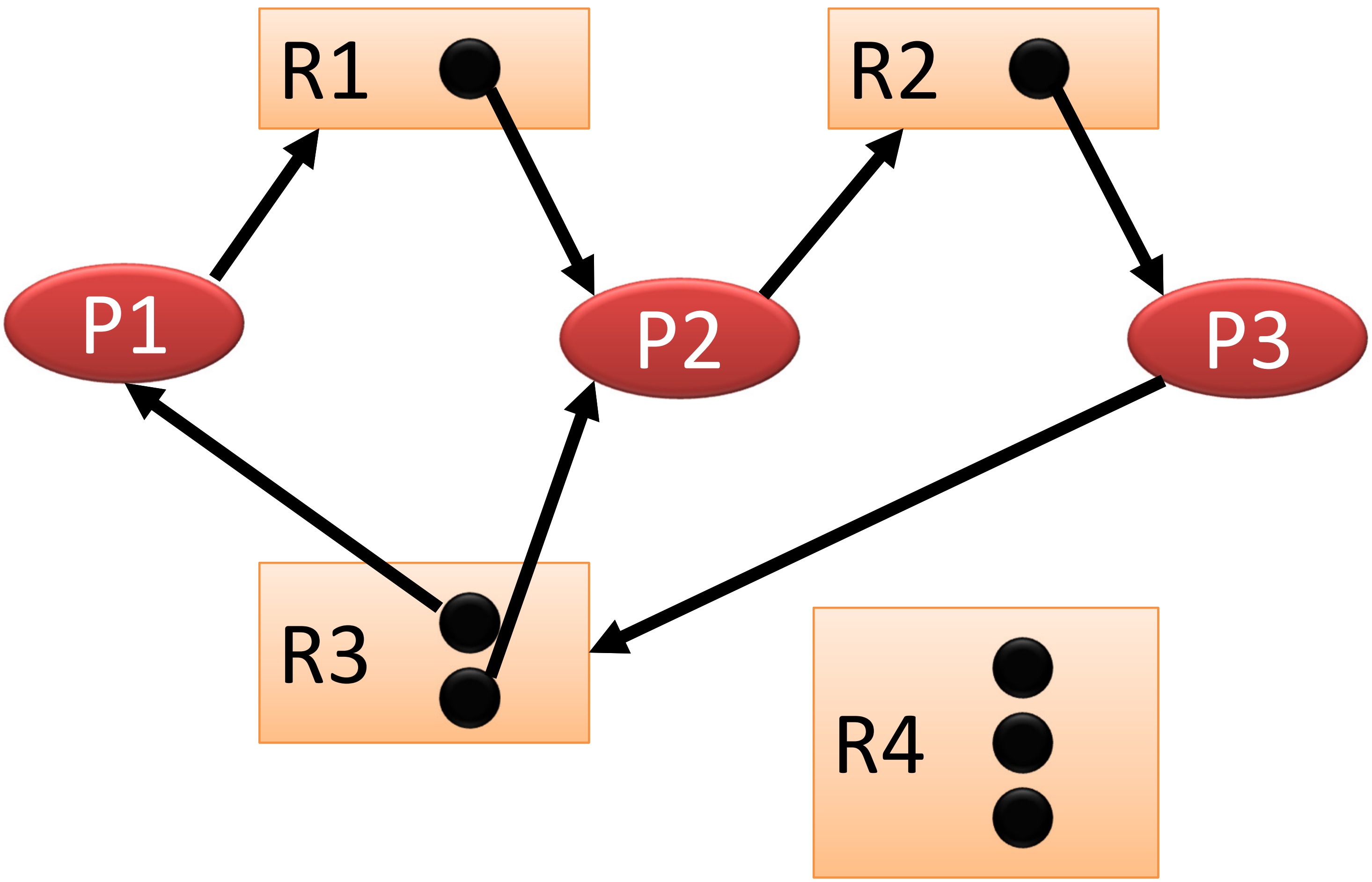
- P_i requests instance of R_j



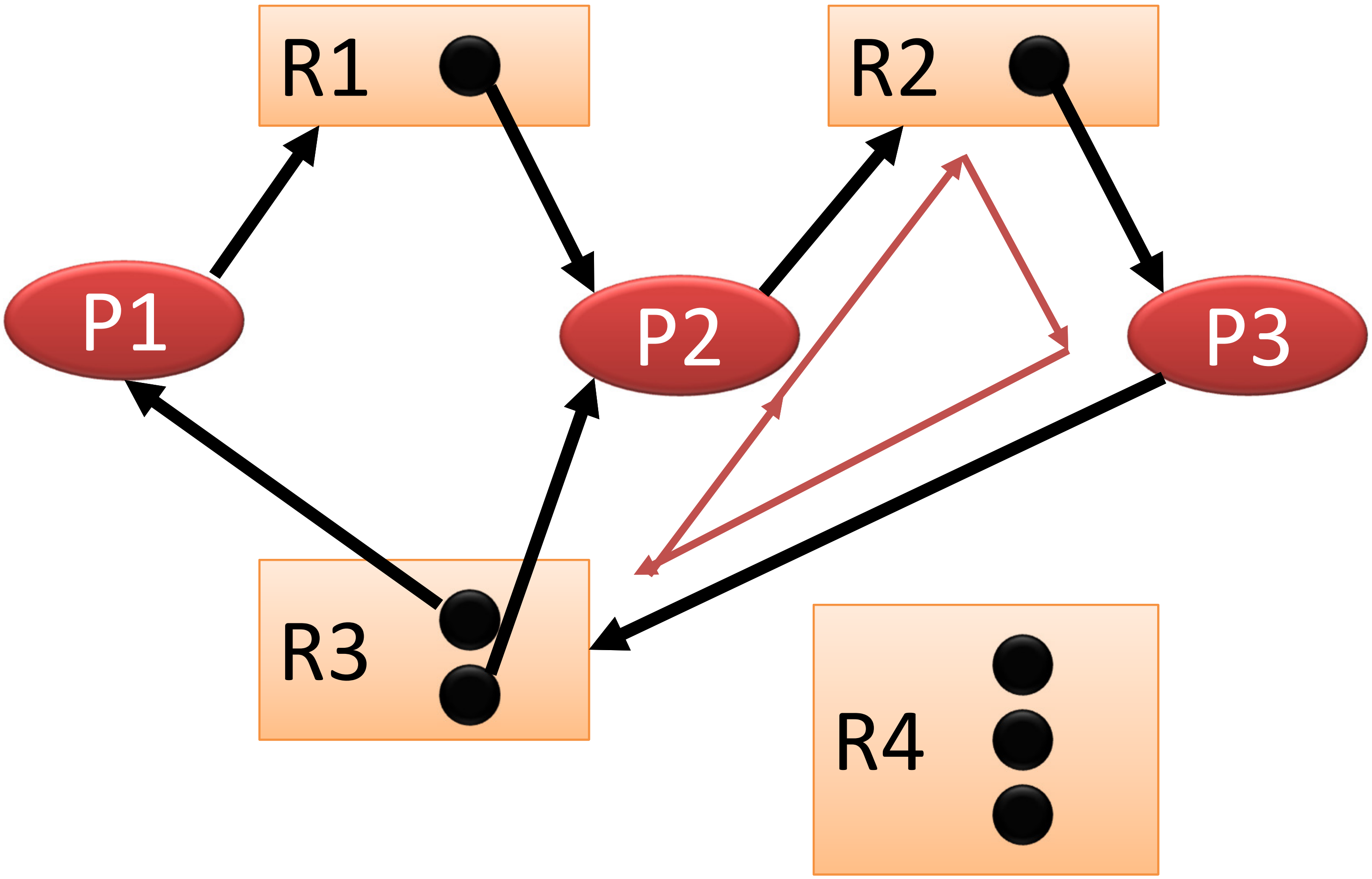
- P_i is holding an instance of R_j



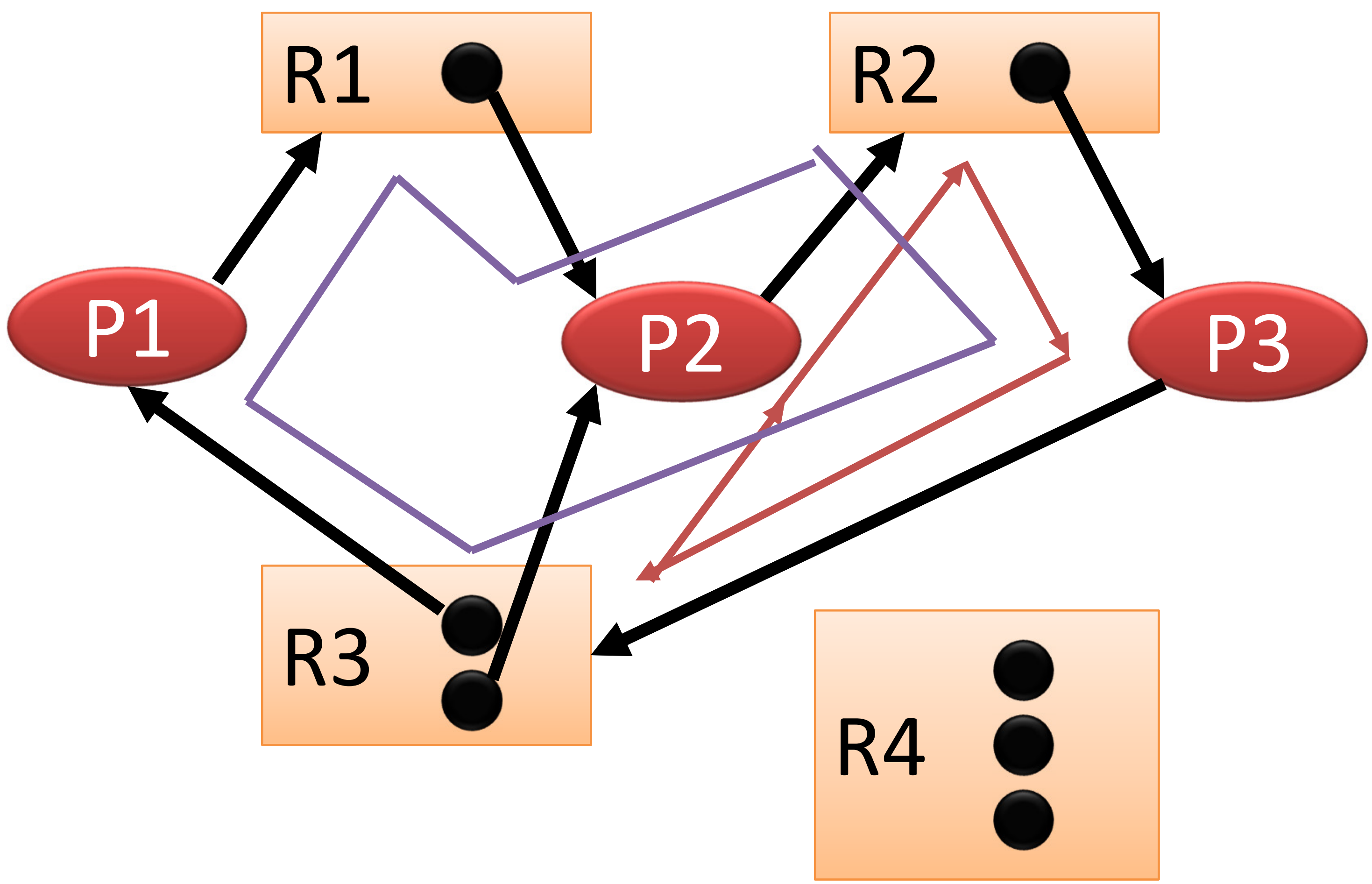
Example of a Resource Allocation Graph with a Deadlock



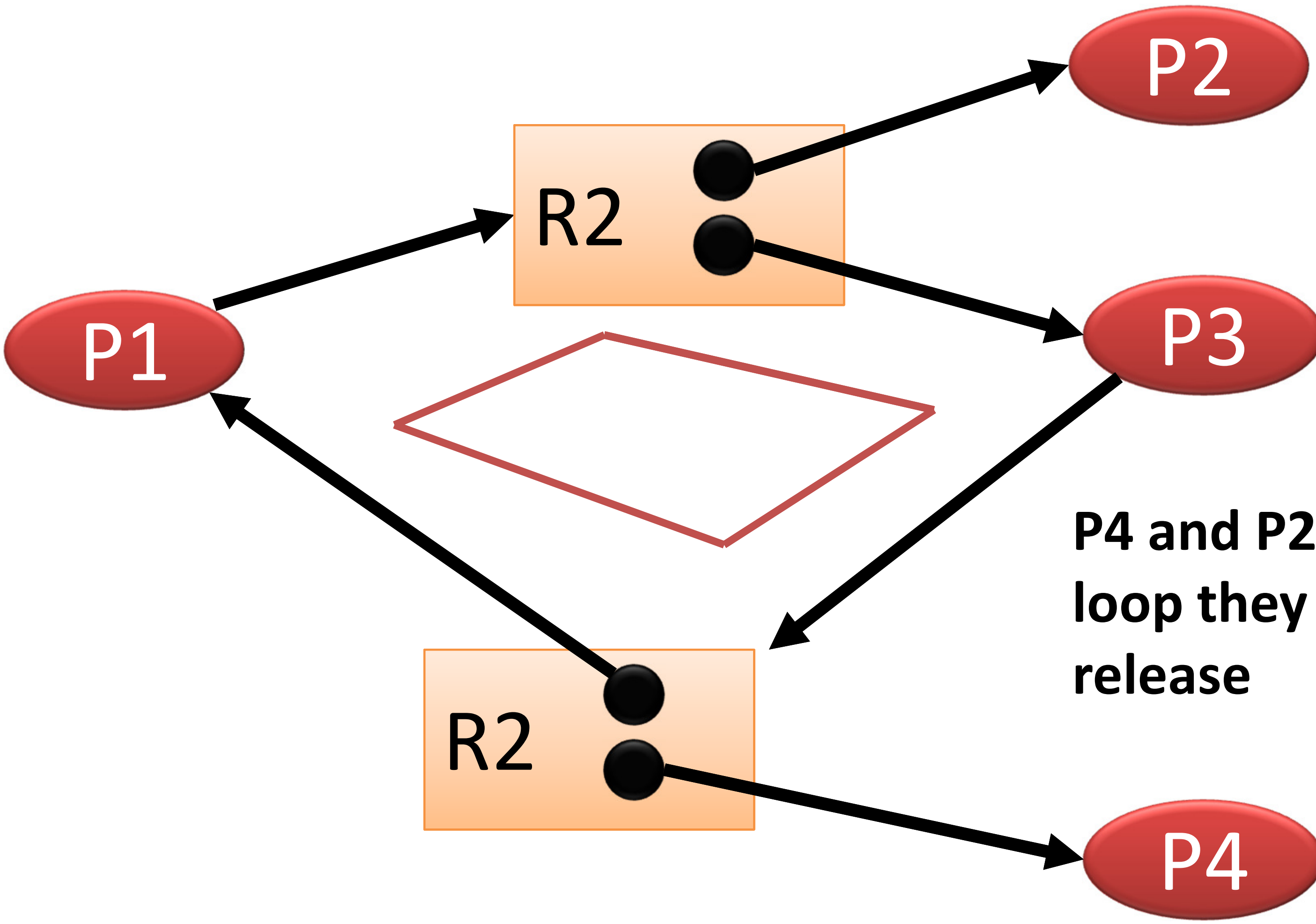
Example of a Resource Allocation Graph with a Deadlock



Example of a Resource Allocation Graph with a Deadlock



Graph with a Cycle but No Deadlock



P4 and P2 are not in loop they will release

Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, **then deadlock**
  - if several instances per resource type, **possibility of deadlock**

Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- ☺ ☺ ☺ Ignore the problem and pretend that deadlocks never occur in the system
  - Used by most operating systems, including UNIX

Prevention, Avoidance & Detection

- Cold wave in December
- Prevention
  - Don't go outside: it is too restrictive
- Avoidance
  - Go to outside but wear sweeter/jacket
- Recovery : Got cold : Take medicine

Deadlock Prevention

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion**
  - Not required for sharable resources (e.g., read-only files);
  - Must hold for non-sharable resources

Deadlock Prevention

Restrain the ways request can be made

- **Hold and Wait**
  - Must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution
  - Or allow process to request resources only when the process has none allocated to it.
  - Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

Deadlock Avoidance

Deadlock Avoidance

- Requires that the system has some additional ***a priori*** information available
- Simplest and most useful model
- It requires
  - Each process declare the ***maximum number*** of resources of each type that it may need

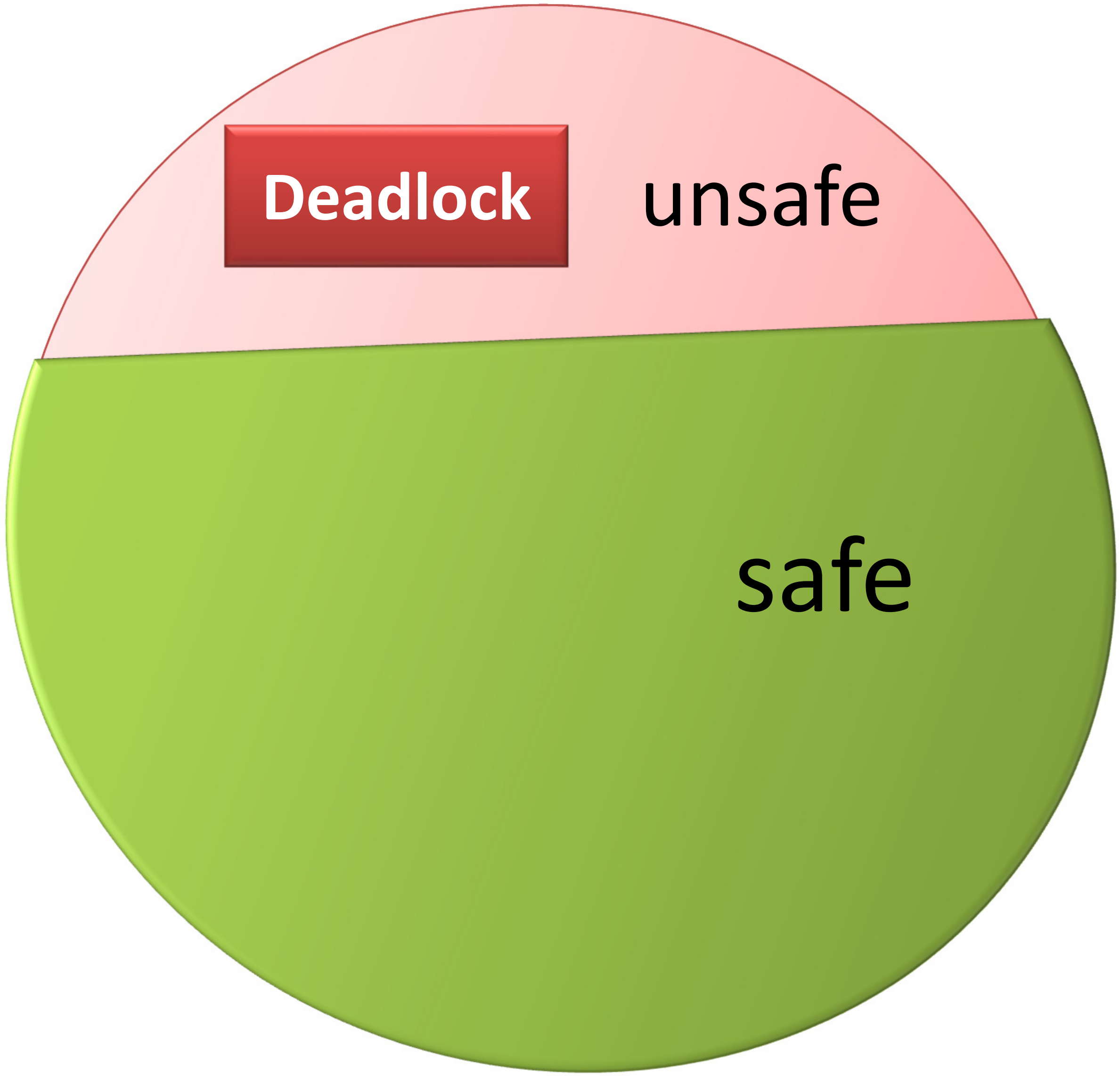
Deadlock Avoidance

- 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
  - The number of allocated resources
  - And the maximum demands of the processes

Basic Facts

- 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.

Safe, Unsafe, Deadlock State



Safe State

- When a process requests an available resource
  - System must decide if immediate allocation leaves the system in a safe state
- **Safe state** If there exists a sequence < $P_1, P_2, \ldots, P_n$ > of all the processes in the systems
  - Such that 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 $j < I$**

Total Order Execution of processes < $P_1, P_2, \ldots, P_n$ > is possible in Safe State

Safe State

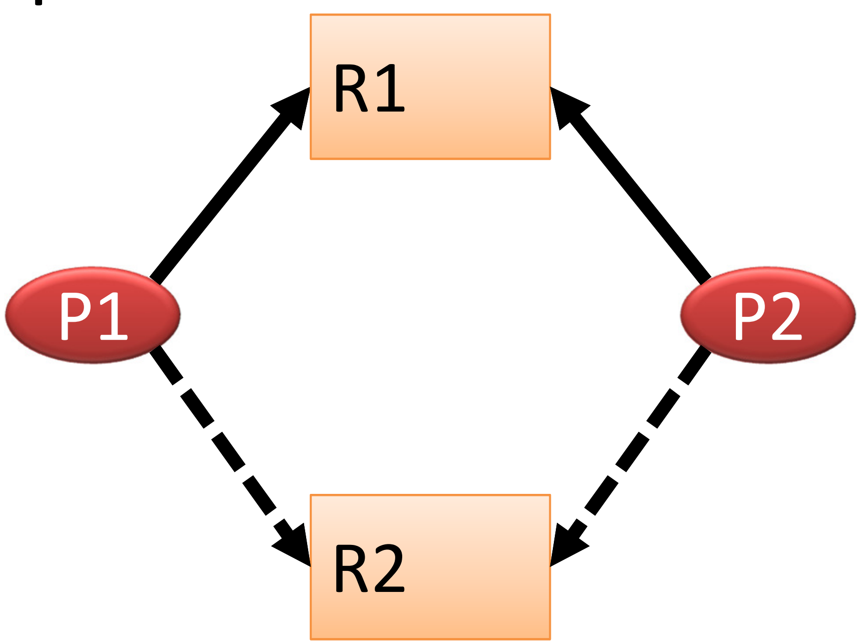
- If there exists a sequence <$P_1, P_2, \ldots, P_n$> of all the processes in the systems
  - Such that for each P_i, the resources that
    - **P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$**
- That is:
  - If P_i resource needs 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

Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph (RAG)
- Multiple instances of a resource type
  - Use the Banker's algorithm
  - **Credit card issued by bank**: If you use X amount , you need to pay X at the end of the month.

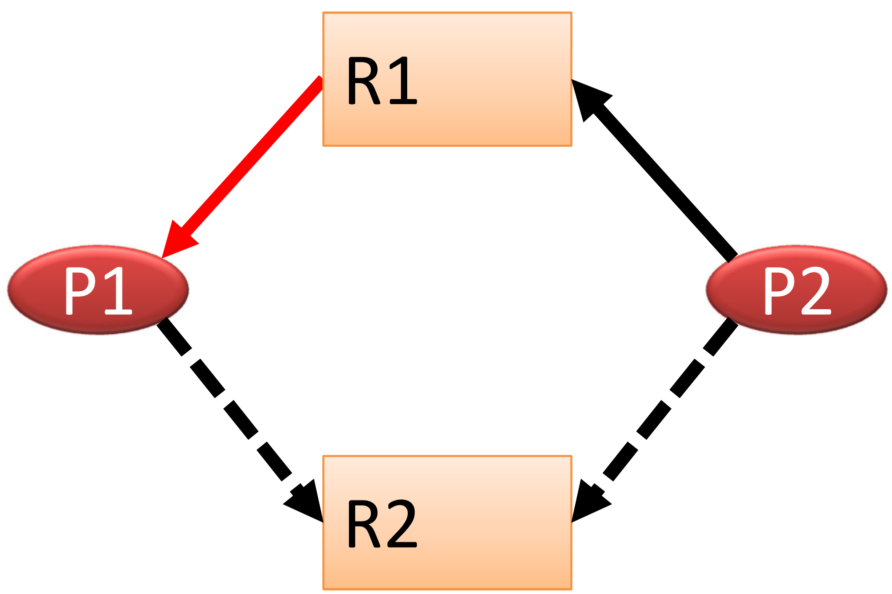
Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource

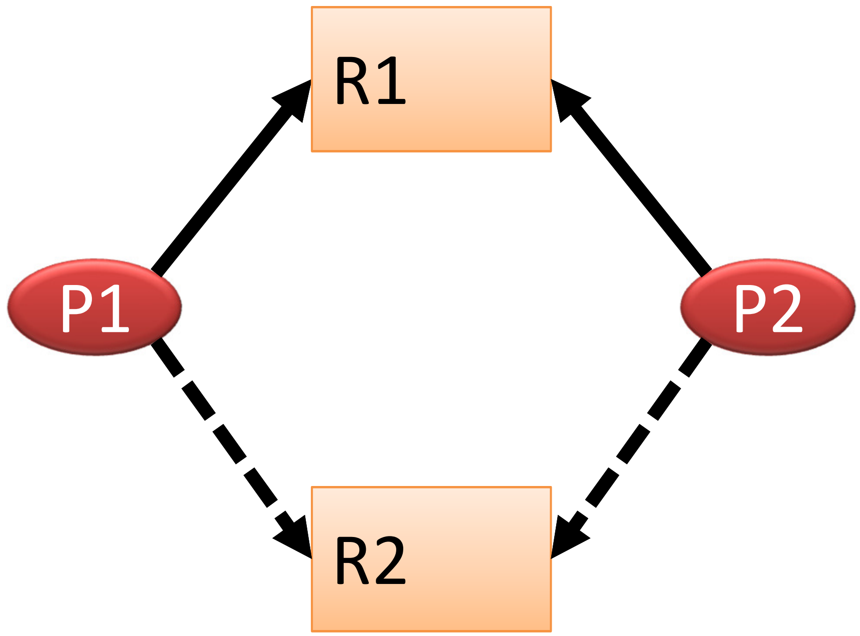


Resource-Allocation Graph Scheme

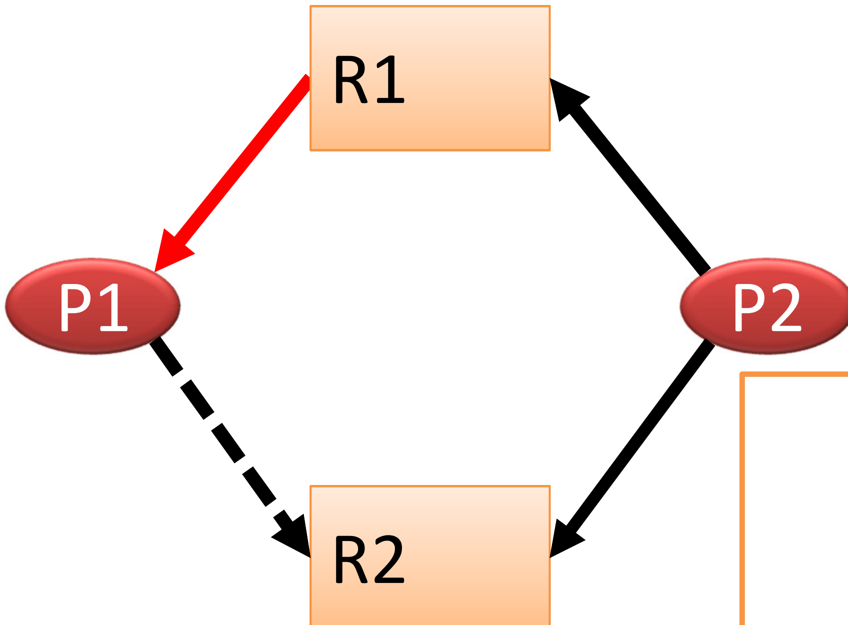
- Request edge converted to an assignment edge when the resource is allocated to the process Red Edge
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system



Resource-Allocation Graph



Resource-Allocation Graph

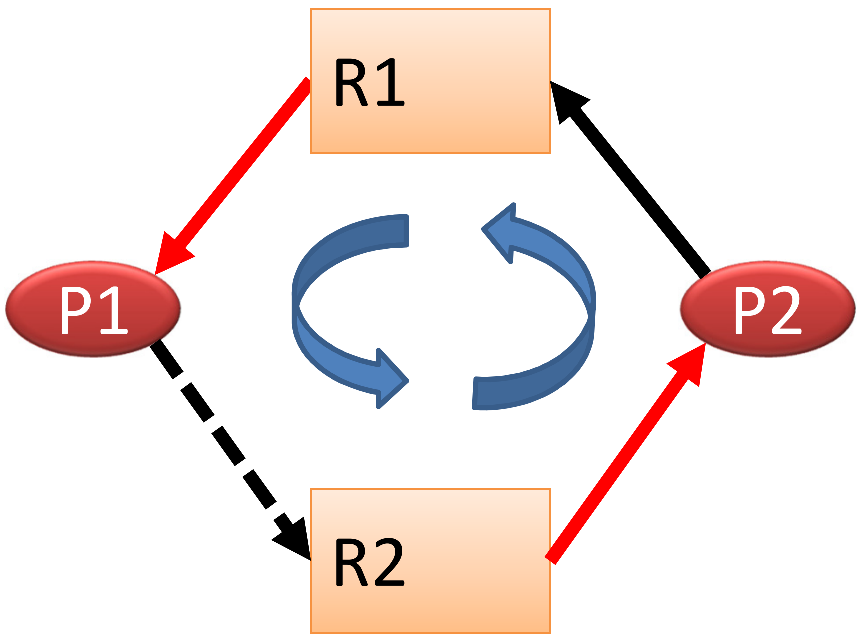


Suppose P2 request R2

....

Convert this Request to Alloc and test for Safety

Resource-Allocation Graph (Unsafe State)



Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if
  - Converting the request edge to an assignment edge **does not form a cycle**

Multiple Instance of Resources: Banker's Algorithm

- **Proposed by Dijkstra, 1965**
  - Works for Multiple Instance of Resources
  - **Same guy Proposed Bakery Algorithms**
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait

Banker algorithm

- For each request
  - The system checks if granting this request may lead to a deadlock in the worse case
    - If yes, do not grant the request
    - If no, grant the request
- When a process gets all its resources
  - it must return them in a finite amount of time

Relation to Bank

- **Relation to Bank**
  - Each customer tells banker the maximum number of resources it needs
  - Customer borrows resources from banker
  - Customer returns resources to banker
  - Customer eventually pays back loan
  - Banker only lends resources if the system will be in a *safe* state after the loan
- ***Safe state*** - there is a lending sequence such that all customers can take out a loan
- ***Unsafe state*** - there is a possibility of deadlock

Safe State and Unsafe State

- Safe State
  - there is *some* scheduling order in which every process can run to completion even if all of them request their maximum number of resources immediately
  - From safe state, the system can guarantee that all processes will finish
- Unsafe state: no such guarantee
  - Not deadlocked state
  - Some process may be able to complete

An Example
of
Deadlock Avoidance

An Example of Deadlock Avoidance

- 5 Processes : P0, P1, P2, P3 and P4
- Three Resource type A, B and C
- Many instances of resources
  - A : 10, B: 5 and C:5
- Allocation: Already allocated
- Max : Maximum need
- Need/request for : Current need

Is Allocation (1 0 2) to P1 Safe?

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	10	5	5
P1	2	0	0	3	2	2	1	2	2	3	3	2
P2	3	0	0	9	0	2	6	0	2			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

If P1 requests max resources, can P1 complete?

Run Safety Test

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	10	5	5
P1	3	0	2	3	2	2	0	2	0	2	3	0
P2	3	0	0	9	0	2	6	0	2			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

If P1 requests max resources, can P1 complete?

Allocate to P1, Then

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	10	5	5
P1	3	2	2	3	2	2	0	0	0	2	1	0
P2	3	0	0	9	0	2	6	0	2			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

If P1 requests max resources, can P1 complete? **YES**

Release - P1 Finishes

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	10	5	5
P1	0	0	0	3	2	2	3	2	2	5	3	2
P2	3	0	0	9	0	2	6	0	2			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

Now P3 can acquire max resources and release

Release - P3 Finishes

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	10	5	5
P1	0	0	0	3	2	2	3	2	2	7	4	3
P2	3	0	0	9	0	2	6	0	2			
P3	0	0	0	2	2	2	2	2	2			
P4	0	0	2	4	3	3	4	3	1			

Now P4 can acquire max resources and release

Release - P4 Finishes

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	10	5	5
P1	0	0	0	3	2	2	3	2	2	7	4	5
P2	3	0	0	9	0	2	6	0	2			
P3	0	0	0	2	2	2	2	2	2			
P4	0	0	0	4	3	3	4	3	3			

Now P2 can acquire max resources and release

Release - P2 Finishes

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	10	5	5
P1	0	0	0	3	2	2	3	2	2	10	4	5
P2	0	0	0	9	0	2	9	0	2			
P3	0	0	0	2	2	2	2	2	2			
P4	0	0	0	4	3	3	4	3	3			

Now P0 can acquire max resources and release

So P1 Allocation (1 0 2) Is Safe

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	10	5	5
P1	3	0	2	3	2	2	0	2	0	2	3	0
P2	3	0	0	9	0	2	6	0	2			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

Got a sequence P1, P3, P4, P2 and P2, can be completed even if all request their maximum need

YES, P1 Allocation(1 0 2) Is Safe

Is allocation (0 2 0) to P0 Safe?

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	10	5	5
P1	3	0	2	3	2	2	0	2	0	2	3	0
P2	3	0	0	9	0	2	6	0	2			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

Try to Allocate 2 B to P0

Run Safety Test

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	3	0	7	5	3	7	2	3	10	5	5
P1	3	0	2	3	2	2	0	2	0	2	1	0
P2	3	0	0	9	0	2	6	0	2			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

No Processes may get max resources and release

So Unsafe State- Do Not Enter

Process	Alloc				Max				Need				Available		
	A	B	C		A	B	C		A	B	C		A	B	C
P0	0	3	0		7	5	3		7	2	3		10	5	5
P1	3	0	2		3	2	2		0	2	0		2	1	0
P2	3	0	0		9	0	2		6	0	2				
P3	2	1	1		2	2	2		0	1	1				
P4	0	0	2		4	3	3		4	3	1				

Return to Safe State and do not allocate resource

P0 Suspended Pending Request

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	10	5	5
P1	3	0	2	3	2	2	0	2	0	2	3	0
P2	3	0	0	9	0	2	6	0	2			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

When enough resources become available, P0 can awake

Data Structures for the Banker's Algorithm

- N=# process, m = # resource type

```
int AVL[M];
int Max[N][M],
int Alloc[N][M],
int Need[N][M];
```

State

- **AVL[j]** instances of resource type $\mathbf{R_j}$ available
- Process $\mathbf{P_i}$ may request at most **Max[i][j]** instances of resource type $\mathbf{R_j}$
- $\mathbf{P_i}$ is currently allocated **Alloc [i][j]** instances of $\mathbf{R_j}$
- P_i may need **Need[i][j]** more instances of $\mathbf{R_j}$ to complete its task

Need [i][j] = Max[i][j] – Alloc[i][j]

Safety Algorithm

```
Let Work[M] and Finish[N]; Found = true;
for (i = 0; i<N; i++)  { // Initialize
    Work[1:m] = Available[1:m] ;  Finish [i] = false;
}
while (Found ==true) {
    Find an  i such that both
        Finish [i] = false  &&  Need_i [1:m] ≤ Work[1:m]
    If (Found ==false ) Break;
    Work[1:m] += Allocation_i [1:m];  Finish[i] = true;
    }
if (Finish [i] == true for all i)
        The system is in a safe state
```

Resource-Request Algorithm for Process P_i

$Request_i$ [1:m]= Req vector for process P_i.

If ($Request_i$ *[1:m]*> $Need_i$ *[1:m]*)

Raise error (process has exceeded its max claim)

If ($Request_i$ [1:M]> *Available[1:M]*)

P_i must wait, since resources are not available

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_allocation()) **//Previous Page/Slide**

The requested resources are allocated to P_i

else

P_i must wait, and the old resource-allocation state is restored

Deadlock Detection

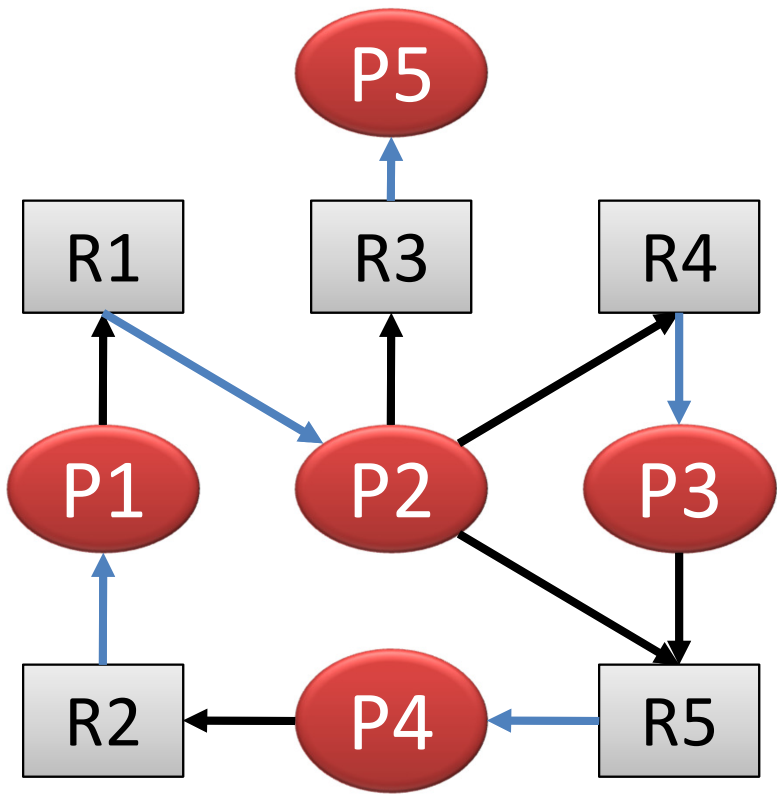
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

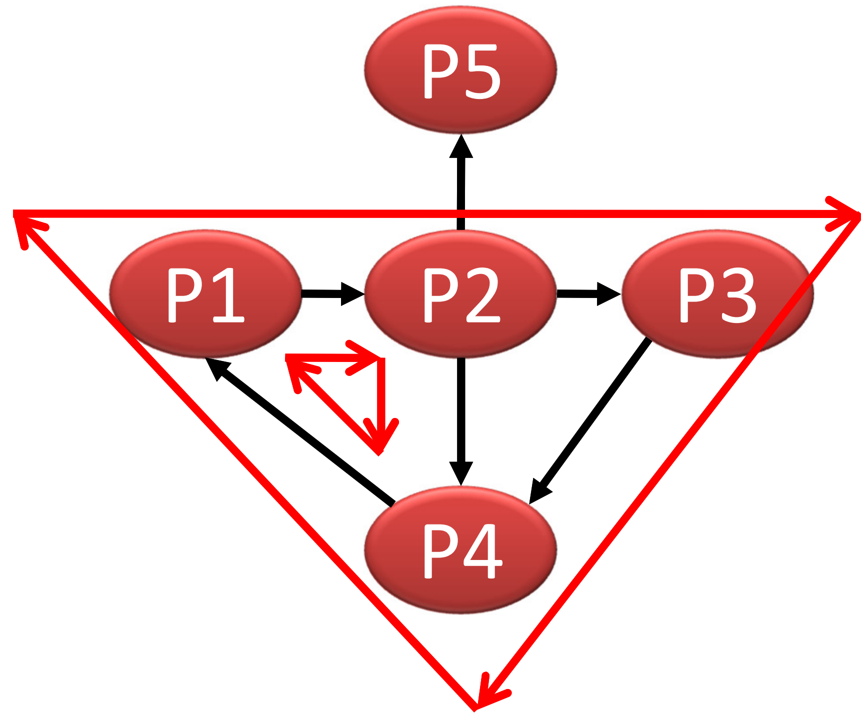
Single Instance of Each Resource Type

- Maintain **wait-for** graph
  - Nodes are processes
  - $\boldsymbol{P_i} \rightarrow \boldsymbol{P_j}$ if $\boldsymbol{P_i}$ is waiting for $\boldsymbol{P_j}$
- Periodically searches for a cycle in WFG
  - If there is a cycle, there exists a deadlock
- Detect a cycle in a graph
  - $\boldsymbol{O(n^2)}$ operations, where $\boldsymbol{n}$ =number of vertices

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph



Corresponding wait-for graph

Several Instances of a Resource Type

- Similar to Safety Algorithm of Deadlock Avoidance

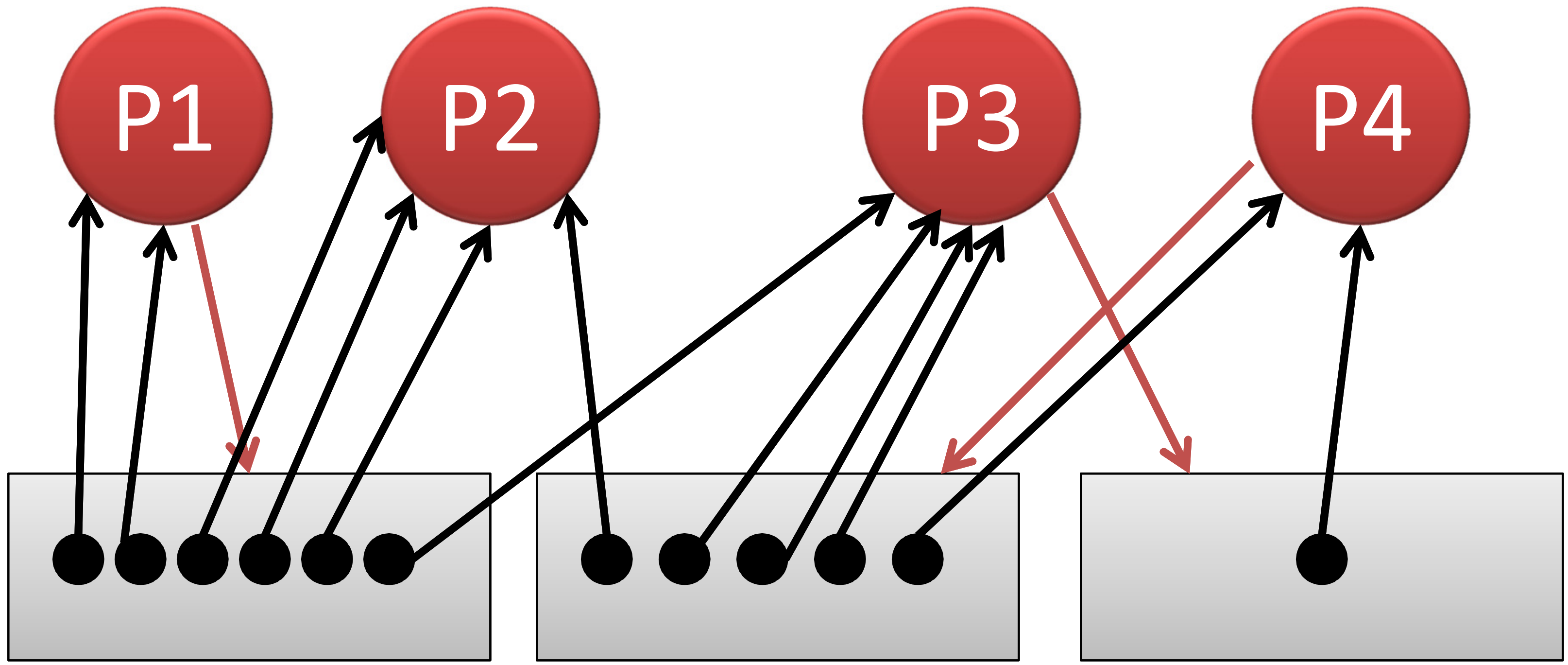
- **Available[1:M] :** Currently Available
- **Allocation[N:M]:** currently allocated
- **Request[N:M]:** current request

Deadlock Detection Graphical Approach

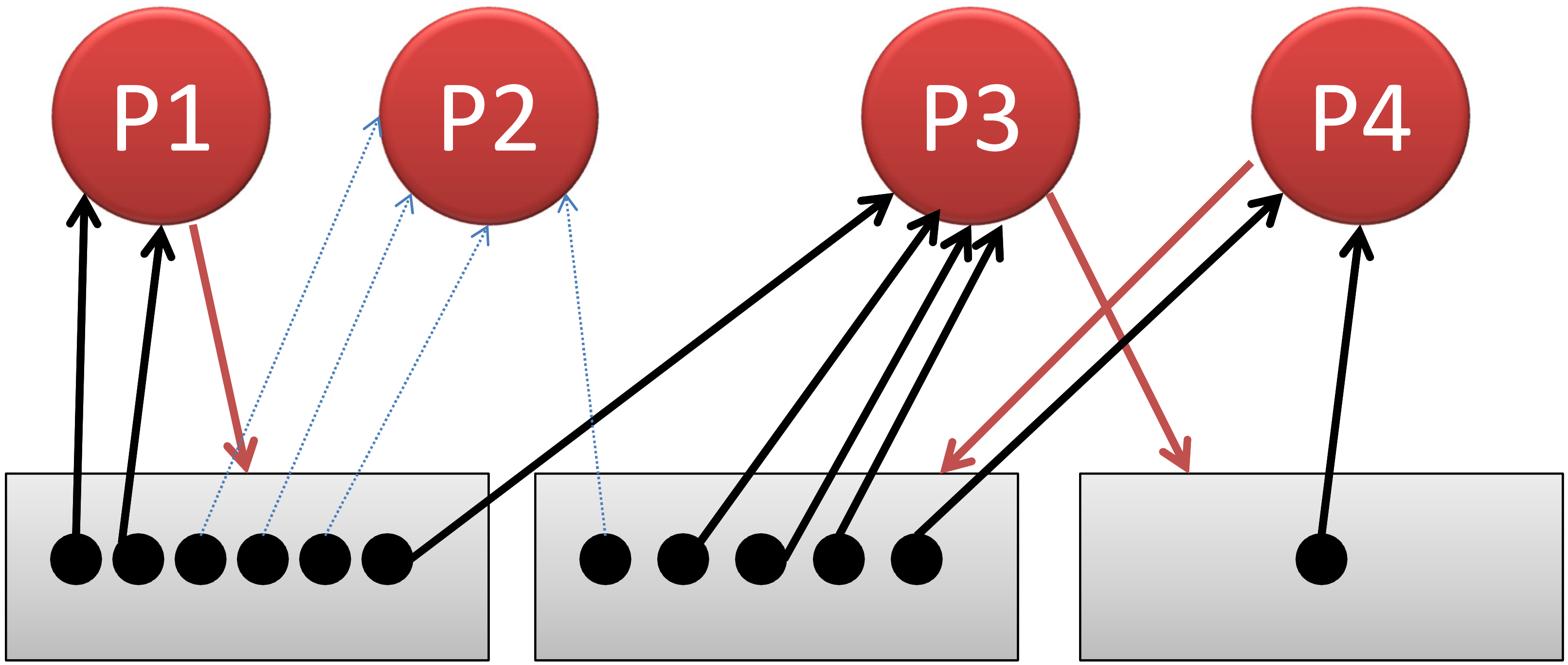
- Resource Allocation Graph
- Reduction (Erase)
  - If a resource has only arrow away from it
    - No request pending to resource
  - If a process has only arrow pointing towards it
    - All the request granted
  - If a process has arrows pointing away from it but each such request arrow there is an available dot in resource : Erase all the process arrow

Deadlock Detection Example

Example



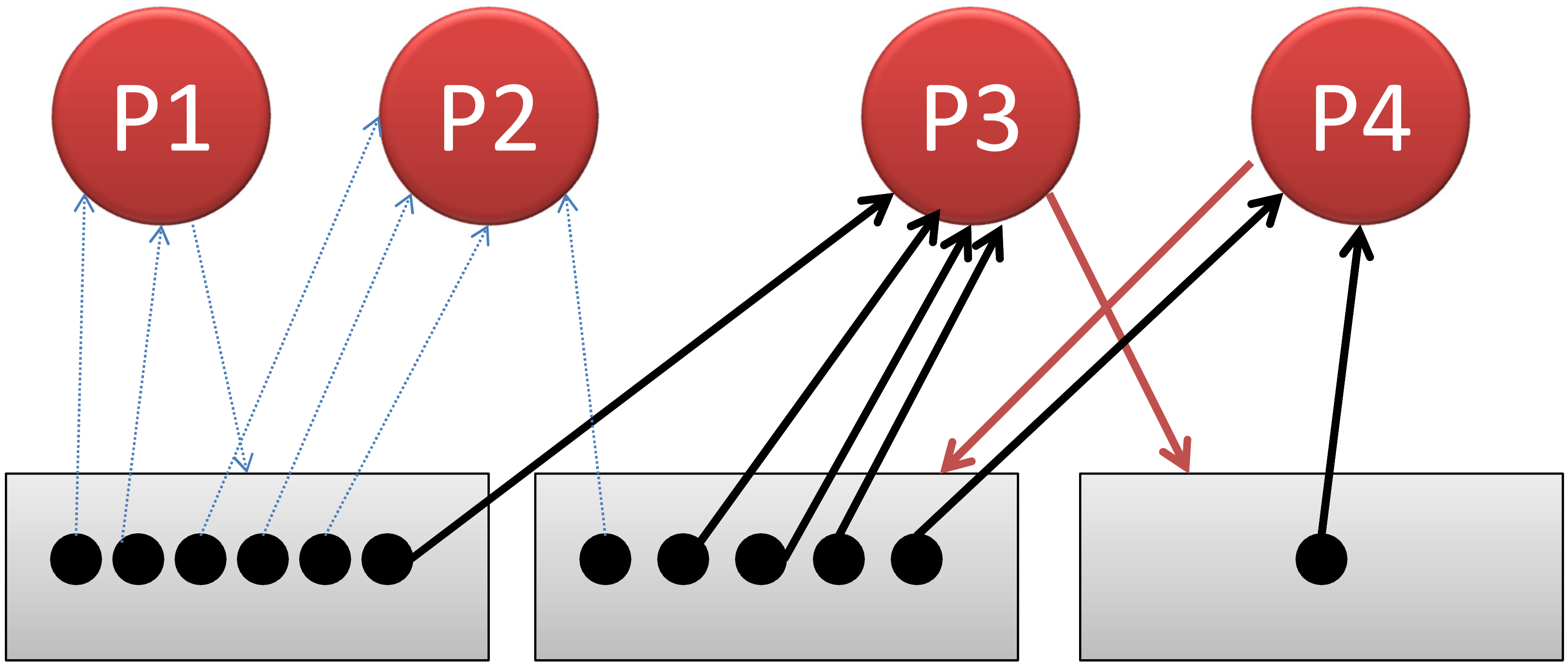
Example



If a process has only arrow pointing towards it

All the request granted

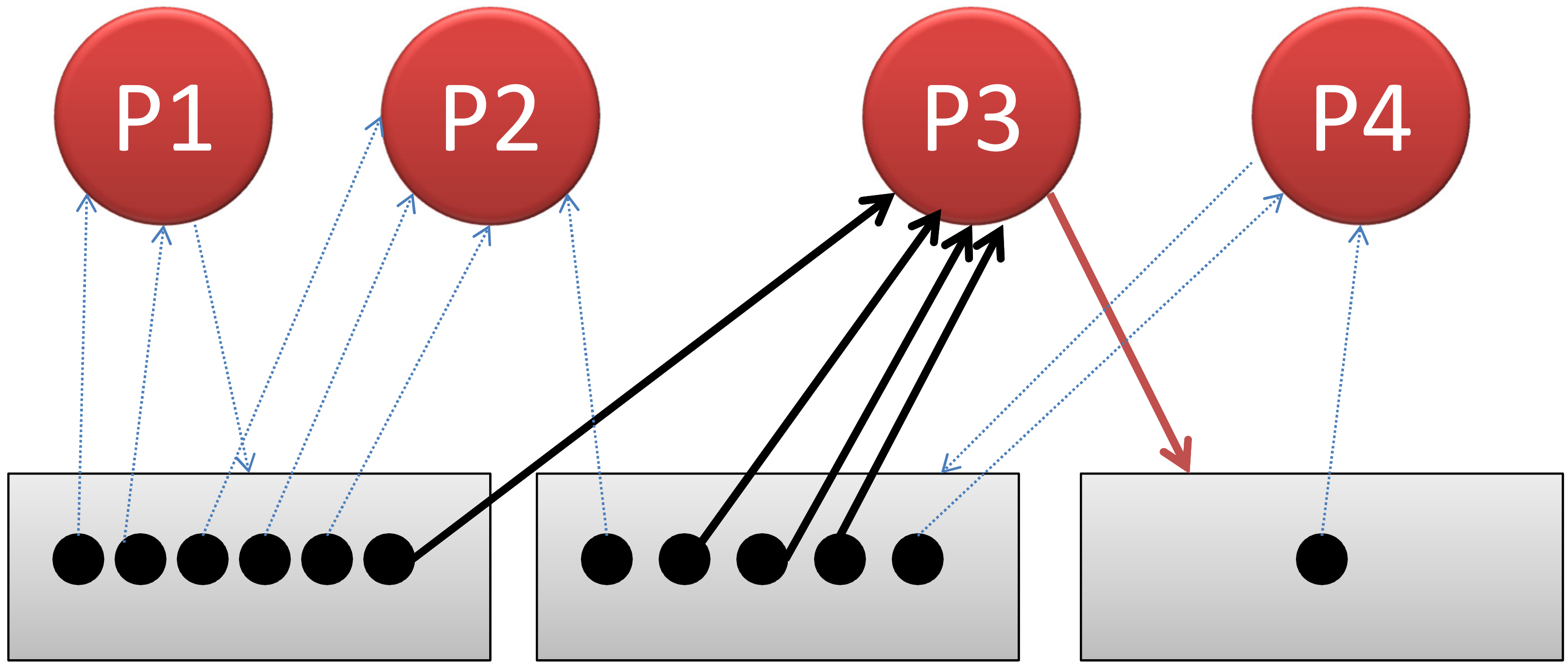
Example



If a process has arrows pointing away from it but each such request arrow there is an available dot in resource

Erase all the process arrow

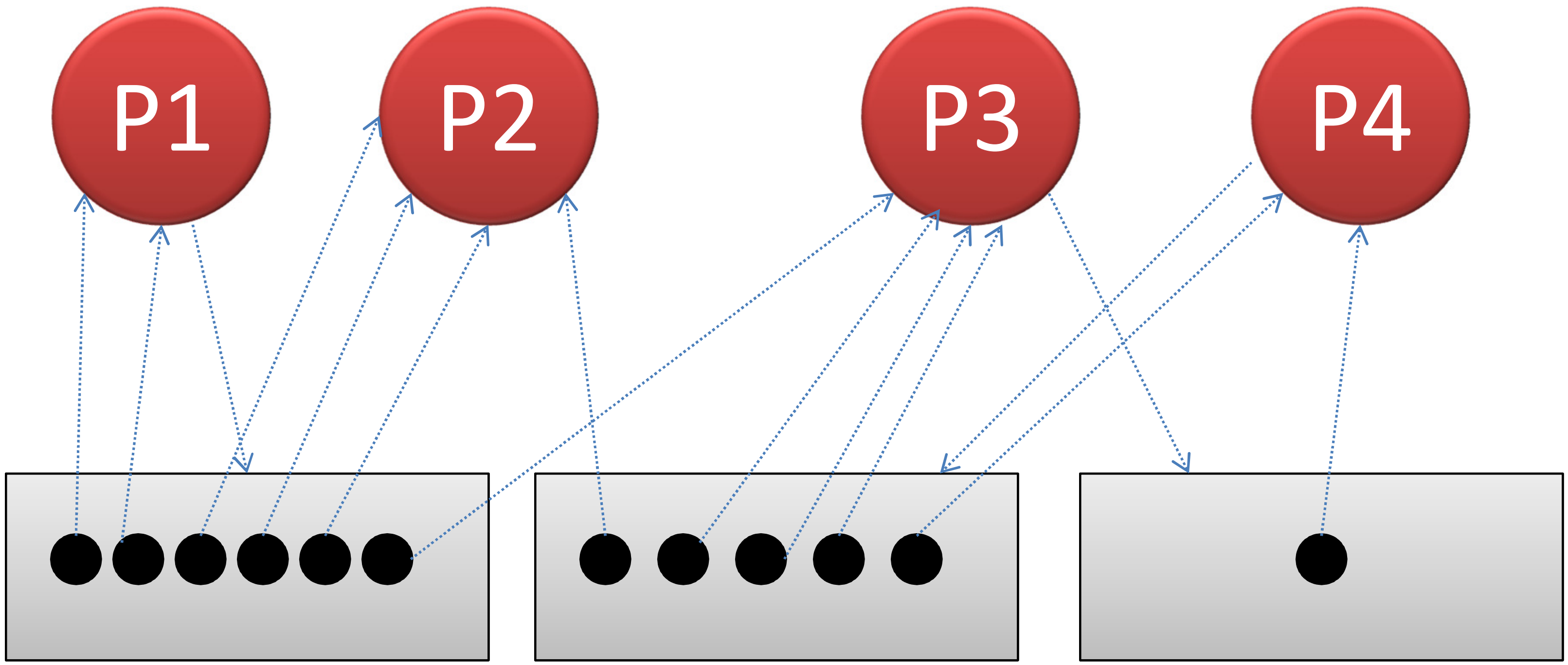
Example



If a process has arrows pointing away from it but each such request arrow there is an available dot in resource

Erase all the process arrow

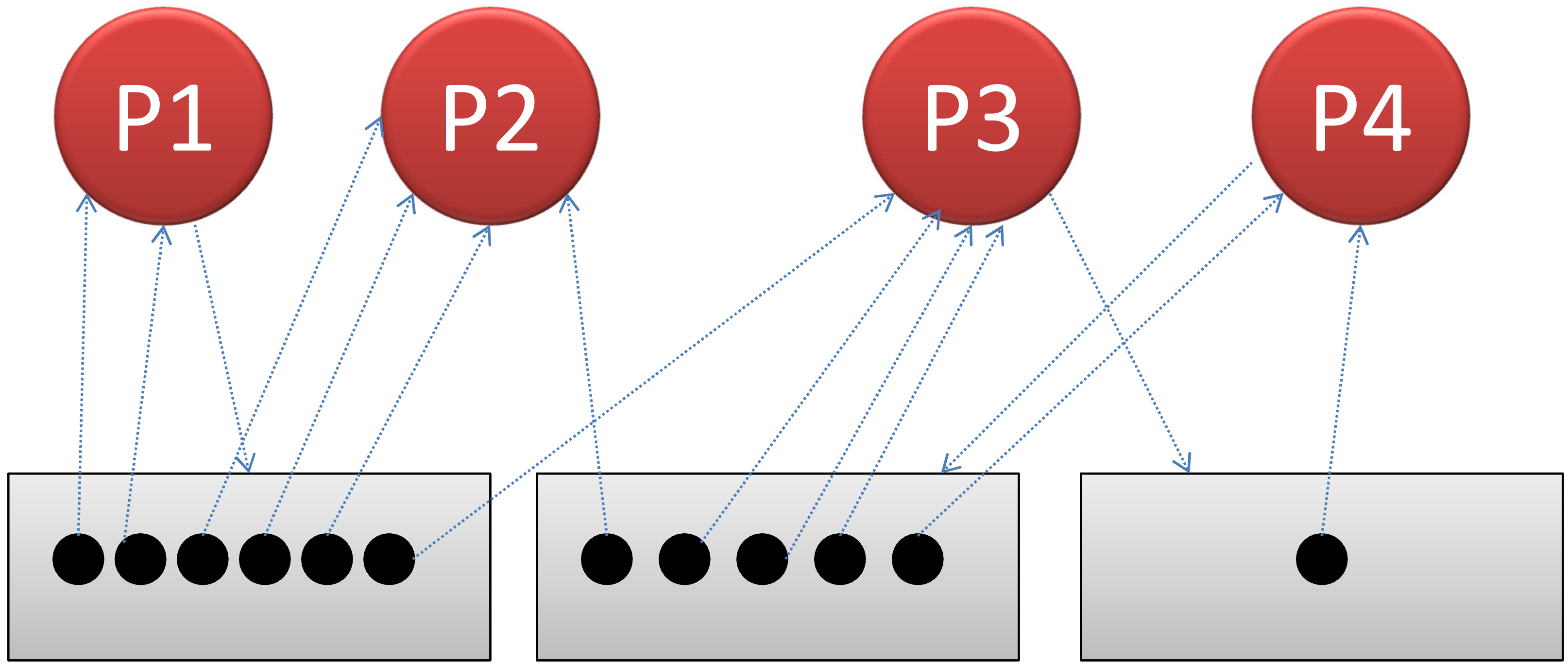
Example



If a process has arrows pointing away from it but each such request arrow there is an available dot in resource

Erase all the process arrow

Example



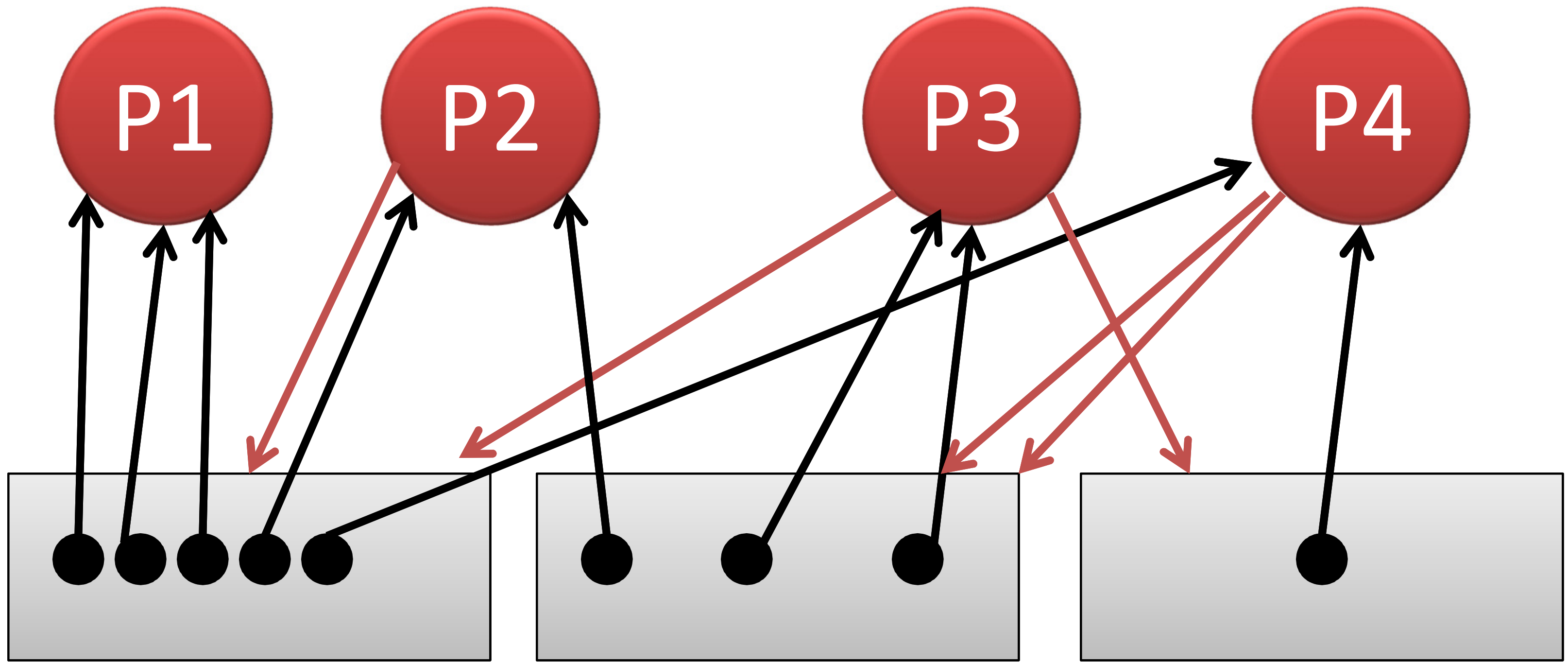
If a process has arrows pointing away from it but each such request arrow there is an available dot in resource

Erase all the process arrow

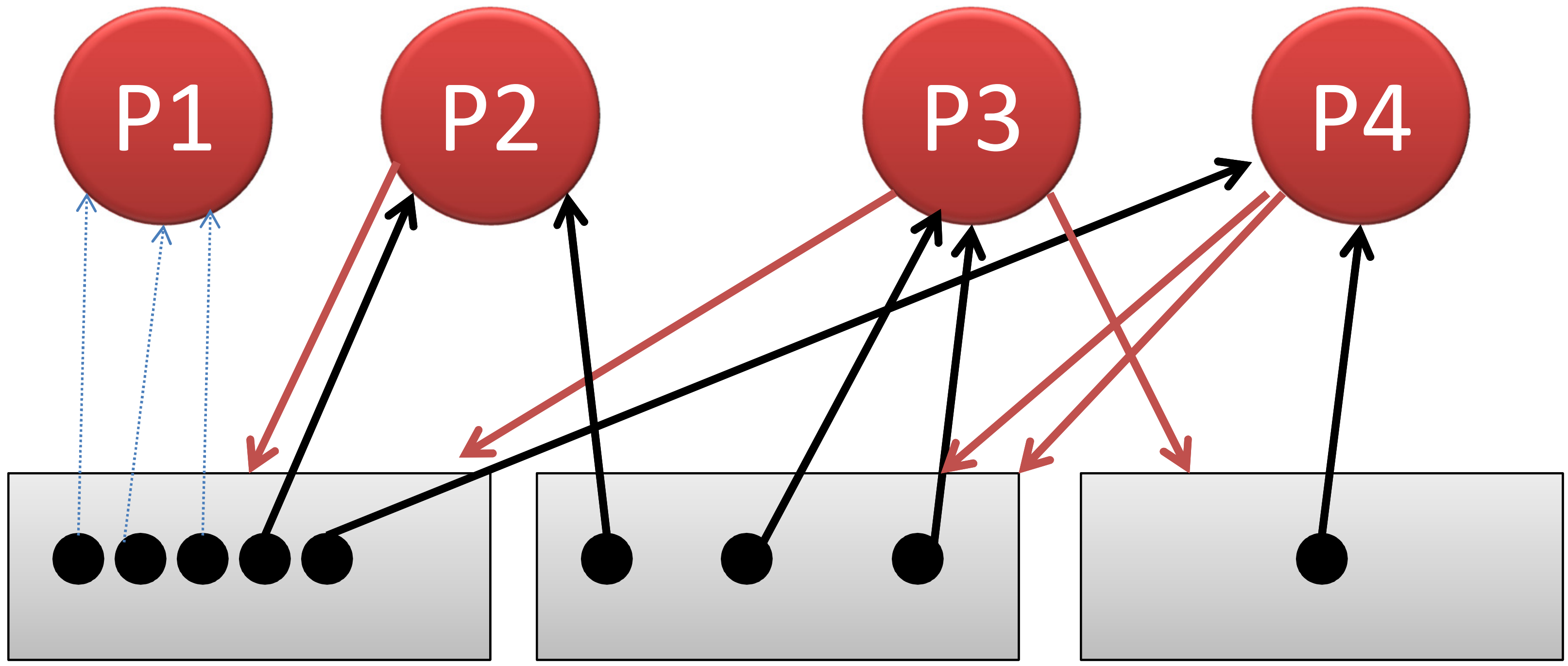
No Deadlock

Deadlock Detection
Another Example

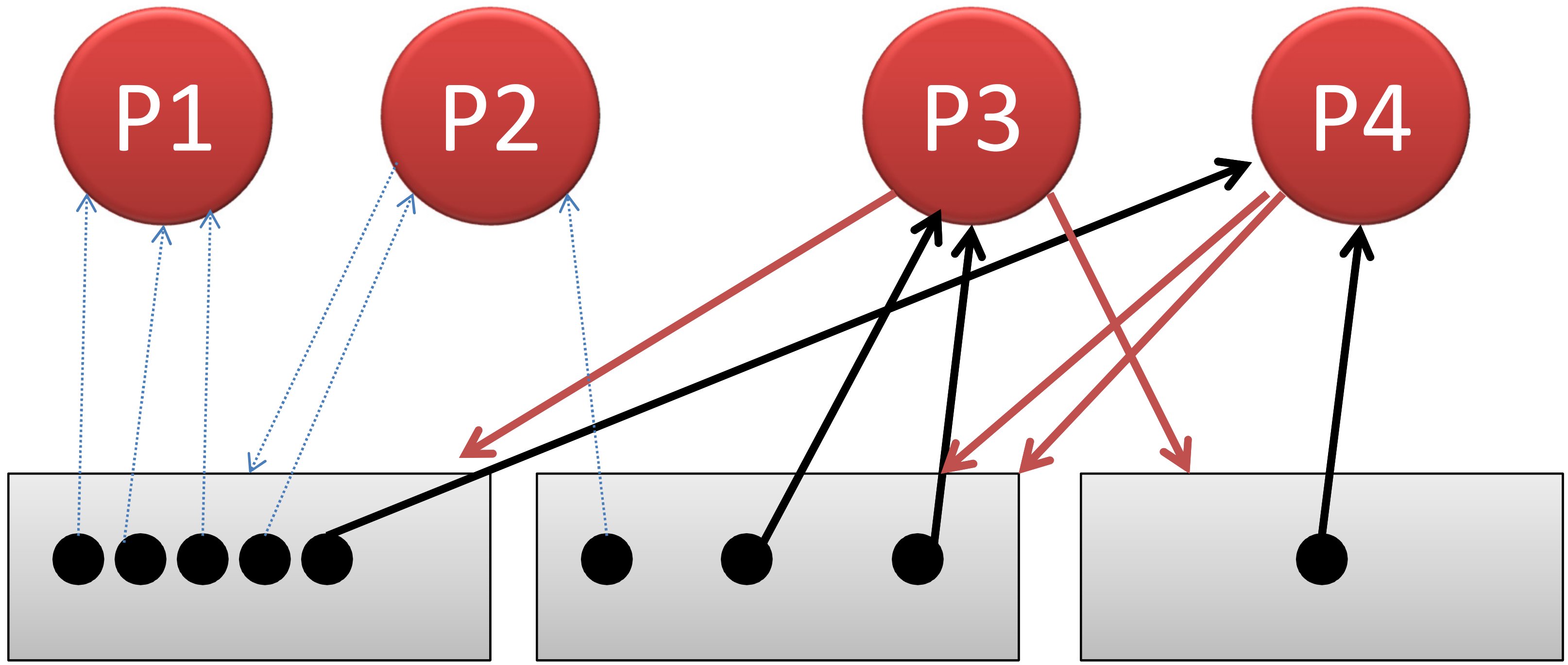
Another Example



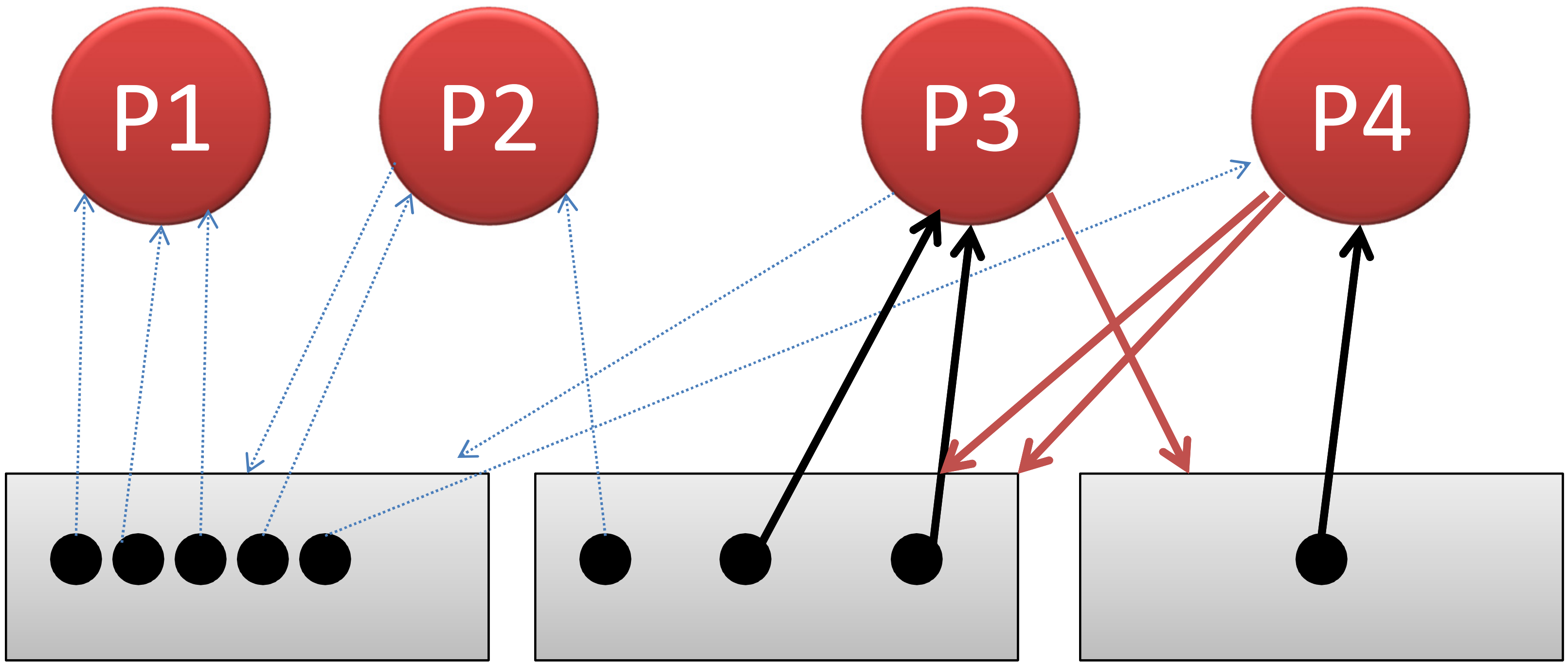
Another Example



Another Example

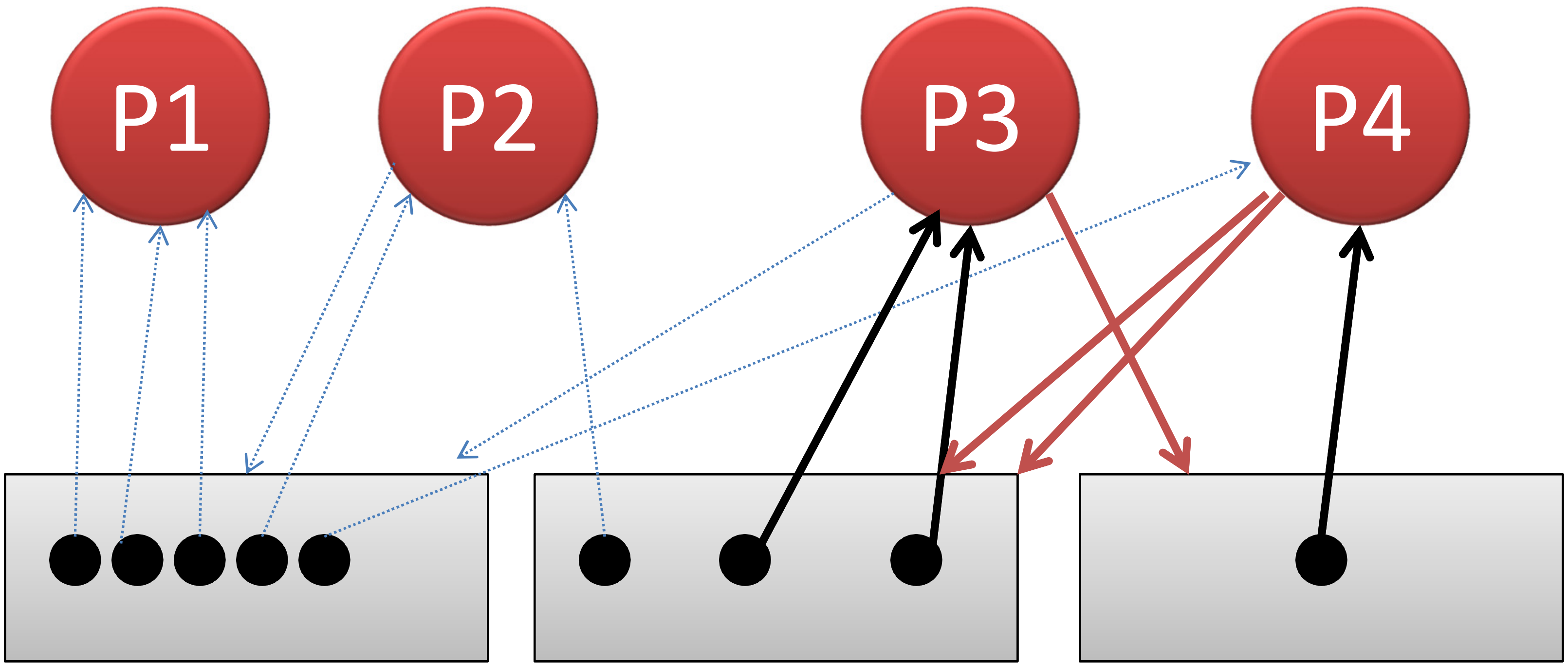


Another Example



If a resource has only arrow away from it
No request pending to resource

Another Example



No-further reduction: There must be cycle

Several Instances of a Resource Type

- Similar to Safety Algorithm of Deadlock Avoidance

- **Available[1:M] :** Currently Available
- **Allocation[N:M]:** currently allocated
- **Request[N:M]:** current request

Detection Algorithm

Let Work[1:M] and Finish[1:M]

Work[1:M] = Available[1:M]; Found=true;

for (i = 0 ; i< n; i++)   { if ($Allocation_i$ [1:M] $\neq$ 0)

Finish[i] = false; else Finish[i] = true; }

while(found==**true)**{

Find an index i such that both:

Finish[i] == false && $Request_i$ [1:M]$\leq$ Work[1:M]

If (Found==**false**) break;

Work = Work + $Allocation_i$ Finish[i] = true

if (Finish[i] == **false**, for some i, 1 $\leq$ i $\leq$ n) {

the system is in deadlock state.

//Moreover, if Finish[i] == false, then P_i is deadlocked

Example of Detection Algorithm

- Five processes P_0 to P_4 Snapshot at time T_0:
- Three resource types A (7 instances), *B* (2), and *C* (6)

	Allocation	Request	Available
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 1	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in ***Finish[i] = true*** for all ***i***

Example of Detection Algorithm

- Five processes P_0 to P_4 Snapshot at time T_0:
- Three resource types A (7 instances), *B* (2), and *C* (6)

	Allocation	Request	Available
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 1 0
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 1	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in ***Finish[i] = true*** for all ***i***

Example of Detection Algorithm

- Five processes P_0 to P_4 Snapshot at time T_0:
- Three resource types A (7 instances), *B* (2), and *C* (6)

	Allocation	Request	Available
	A B C	A B C	A B C
P0	0 1 0	0 0 0	3 1 3
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 1	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in ***Finish[i] = true*** for all ***i***

Example of Detection Algorithm

- Five processes P_0 to P_4 Snapshot at time T_0:
- Three resource types A (7 instances), *B* (2), and *C* (6)

	Allocation	**Request**	**Available**
	A B C	A B C	A B C
P0	0 1 0	0 0 0	5 2 4
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 1	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in ***Finish[i] = true*** for all ***i***

Example of Detection Algorithm

- Five processes P_0 to P_4 Snapshot at time T_0:
- Three resource types A (7 instances), *B* (2), and *C* (6)

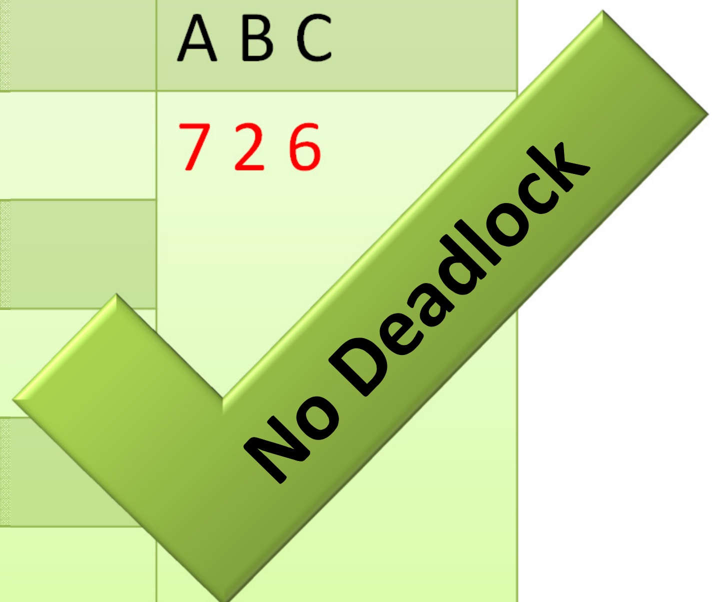
	Allocation	**Request**	**Available**
	A B C	A B C	A B C
P0	0 1 0	0 0 0	7 2 4
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 1	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in ***Finish[i] = true*** for all ***i***

Example of Detection Algorithm

- Five processes P_0 to P_4 Snapshot at time T_0:
- Three resource types A (7 instances), *B* (2), and *C* (6)

	Allocation	**Request**	**Available**
	A B C	A B C	A B C
P0	0 1 0	0 0 0	7 2 6
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 1	



- Sequence $<P_0, P_2, P_3, P_1, P_4>$ will result in ***Finish[i] = true*** for all ***i***

Example of Detection Algorithm

P_2 requests an additional instance of type C

	Allocation	Request	Available
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2	3 0 3	**0 0 1**	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 1	

Deadlock

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily,
  - There may be many cycles in the resource graph
  - And so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated

Recovery from Deadlock: Process Termination

- In which order should we choose to abort?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- **Selecting a victim** – minimize cost
  - Which process to abort?
  - Which minimize the cost.
- **Rollback** – return to some safe state, restart process for that state
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor

Thanks