Operating Systems

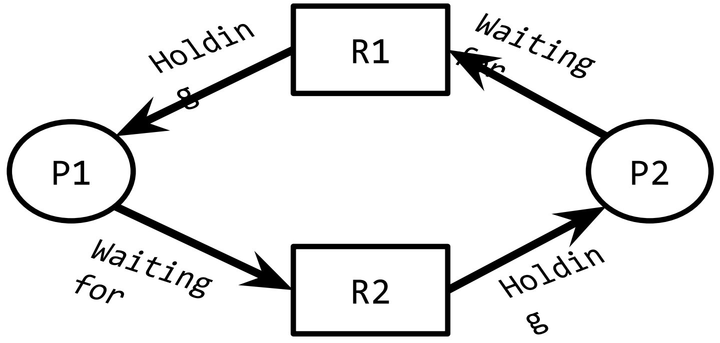
Deadlock

Deadlock

A set of processes is in a deadlocked state if every process in the set is **waiting for an event** that can be caused only by another process in the set.

Example:

- Two process : P1, P2
- Two resource: R1, R2
- P1 is holding R1 and waiting for R2
- P2 is holding R2 and waiting for R1



System Model

- A system contains finite number of resources
    - These resources are distributed among competing processes
    - Examples: CPU cycle, File, I/O devices (printer, DVD drive) etc.
- Each resource type may have identical number of instances
- A process may request as many resource as it requires.
    - But, it should not exceed the total number of available resources of the system
- A process may utilize a resource only in the following sequence
    1. **Request**: requests the resource
    2. **Use**: operate on the resource
    3. **Release**: releases the resource

Necessary Conditions for Deadlock

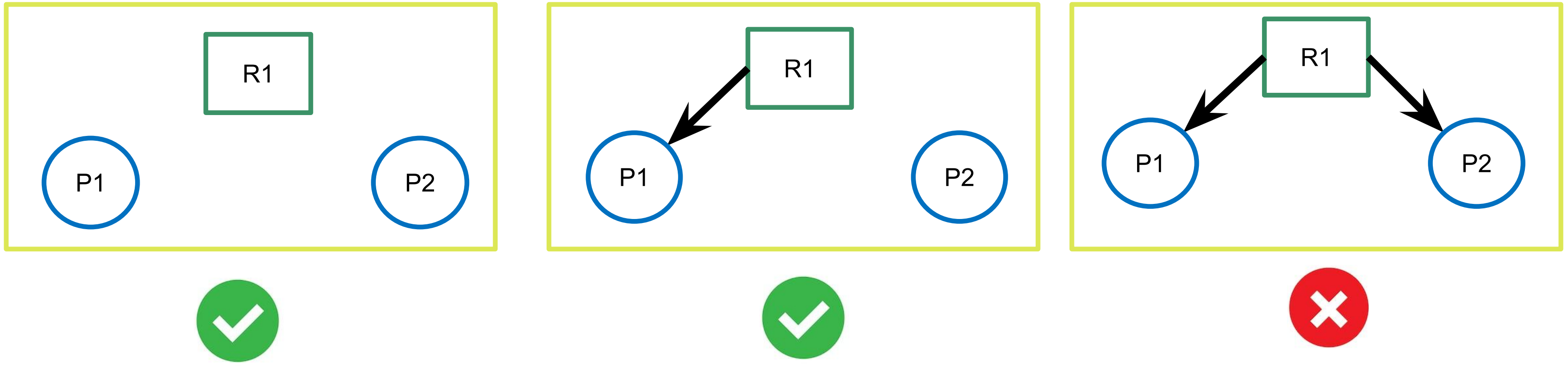
A deadlock situation can arise if the following four conditions hold simultaneously in a system -

1. **Mutual exclusion**
2. **Hold and wait**
3. **No preemption**
4. **Circular wait**

If one of them is not present in a system, no deadlock will arise

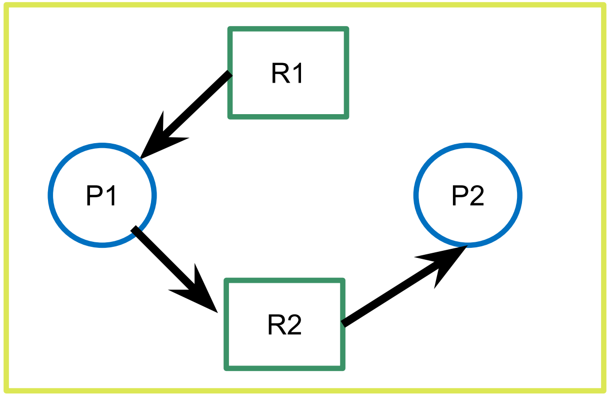
Mutual Exclusion

At least one resource must be held in a nonsharable mode; that is, only one process at a time can use the resource



Hold and Wait

A process must be holding at least one resource, and waiting to acquire additional resources that are currently being held by other processes.



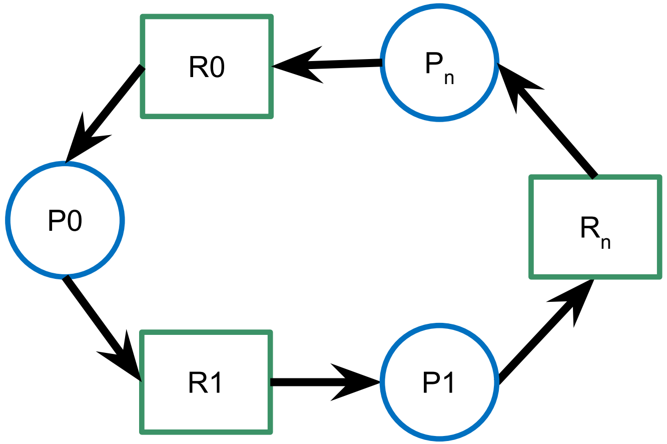
No Preemption

a resource can be released only voluntarily by the process holding it, after that process has completed its task.

❌

Circular Wait

A set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes must exist such that
P0 is waiting for a resource held by P_1,
P_1 is waiting for a resource held by P_2, …, P_{n-1} is waiting for a resource held by P_n, and
P_n is waiting for a resource held by P_0

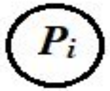
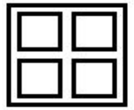
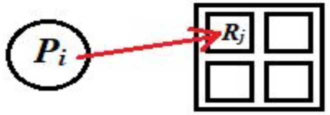
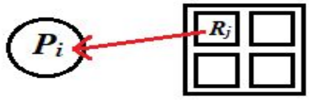


Resource Allocation Graph

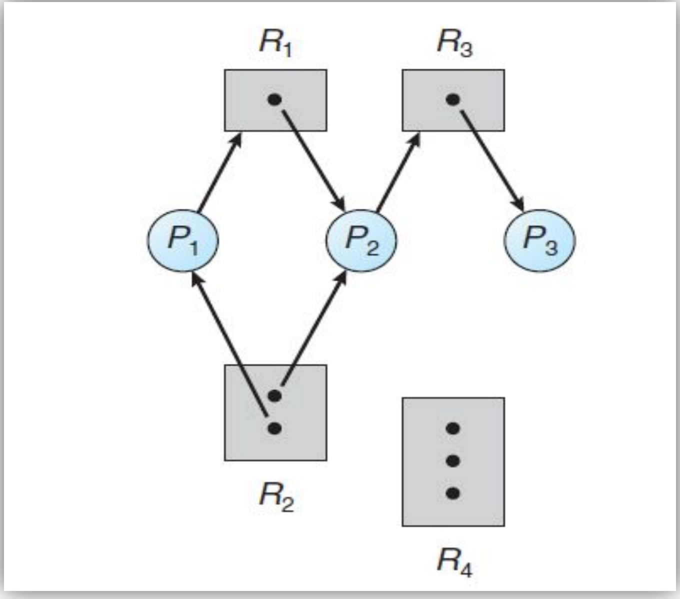
Resource Allocation Graph

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

- V is partitioned into two types:
  - P = {P_1, P_2, …, P_n}, the set consisting of all the processes in the system.
  - R = {R_1, R_2, …, R_m}, the set consisting of all resource types in the system.
- Request edge: directed edge from a process to a resource
  - $P_1 \rightarrow R_2$
- Assignment edge: directed edge from a resource to a process
  - $R_1 \rightarrow P_2$

Process	
Resource Type with 4 instances	
P_i requests instance of R_j	
P_i is holding an instance of R_j	

Example of Resource Allocation Graph

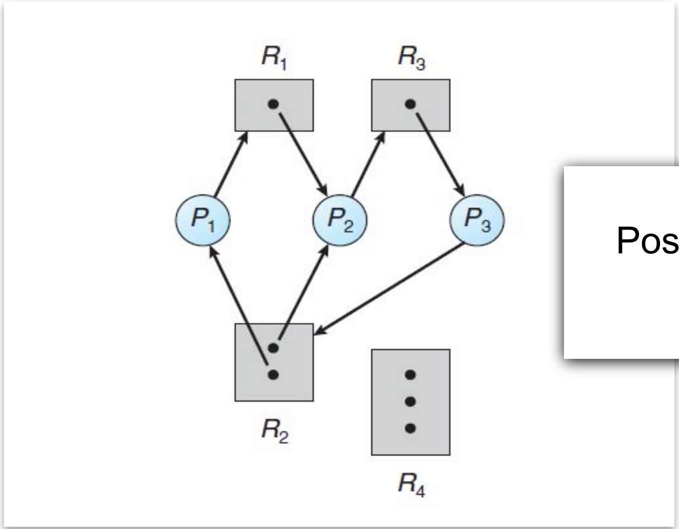


P ={P1, P2, P3}

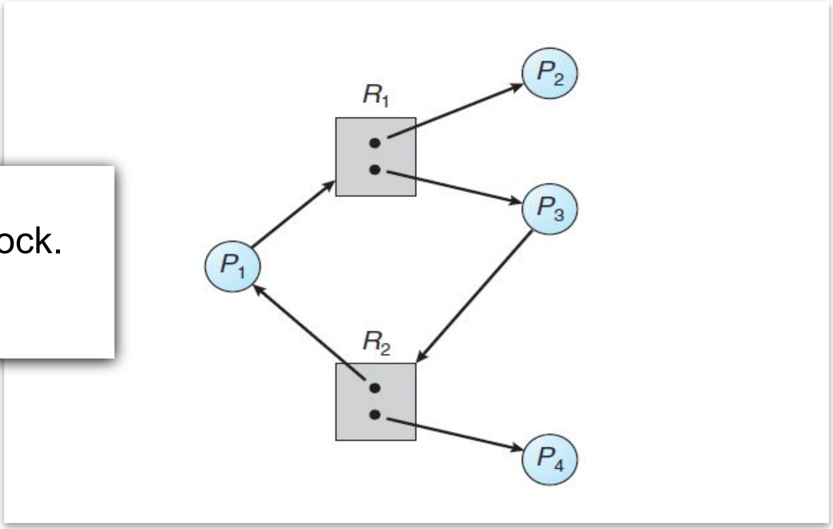
R ={R1, R2, R3, R4}

E ={P1 → R1, P2 → R3, R1 → P2,
R2 → P2, R2 → P1, R3 → P3}

Resource Allocation Graph (Deadlock)



Cycle:
Possibility of Deadlock.
No Cycle:
No Deadlock.



Cycle 1: $P1 \rightarrow R1 \rightarrow P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P1$

Cycle 2: $P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P2$

Deadlock!

Cycle: $P1 \rightarrow R1 \rightarrow P3 \rightarrow R2 \rightarrow P1$

No Deadlock!

Methods for Handling Deadlocks

- **Prevention**:
  - Ensure that the system will never enter a deadlock state
- **Avoidance:**
  - request for any resource will be granted if the resulting state of the system doesn't cause deadlock
- **Detection and recovery:**
  - Allow the system to enter a deadlock state and then recover

Operating Systems

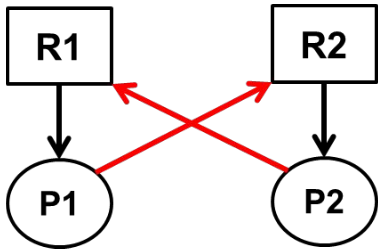
Deadlock Prevention

Deadlock Prevention

Ensure that at least one of the necessary conditions for deadlock cannot hold in the system.

Mutual Exclusion:

- Mutual exclusion condition must hold
- At least one resource must be non-sharable
- Sharable resources do not require mutually exclusive access (example: Read only files)



Hold and Wait:

Guarantee that when a process requests a resource, it does not hold any other resources.

- *Protocol 1:* Request and allocate all its resources before execution.
- *Protocol 2:* Allow process to request resources only when the process has none.

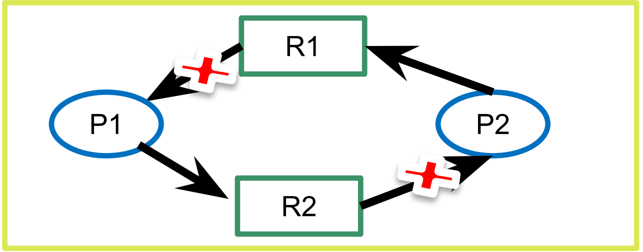
Disadvantages:

- Low utilization of resources
- Starvation

Deadlock Prevention

No Preemption:

- If a process is holding some resources and requesting for another resource, which is held by another process, then this process needs to wait, and the resources held by it will be preempted or 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.



Circular Wait:

- Impose a total ordering of all resource types
- Each process requests resources in an increasing order of enumeration

$R = \{R_1, R_2, \ldots, R_m\}$ and we assign unique number to each of the resource type.

For example, $F(R_1) = 1$, $F(R_2) = 3$, $F(R_3) = 5$.

- A process can initially request any number of instances of a resource type, R_i
- After that, the process can request instances of resource type R_j if and only if $F(R_j) > F(R_i)$.
- A process requesting an instance of resource type R_j must have released any resources R_i such that $F(R_i) \geq F(R_j)$

Operating Systems

Deadlock Avoidance

Deadlock Avoidance

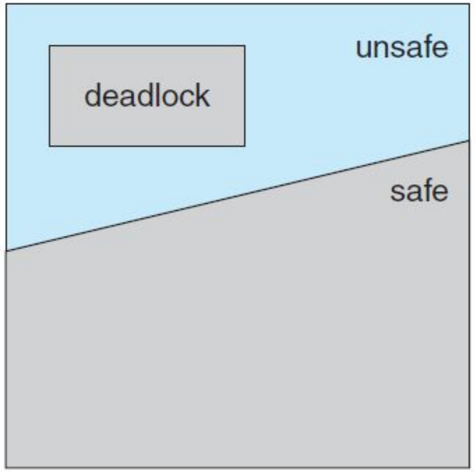
- System has a priori information, such as, maximum number of resources of each type that every process need
- Based on these information construct an algorithm that ensures that the system will never enter a deadlocked state
- Dynamically examines the r*esource-allocation state* to ensure that a circular-wait condition can never exist
- **Resource-allocation state:** number of available and allocated resources, and the maximum demands of the processes.

Safe State

- System is in safe state if can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.
- Safe state = there exists a safe sequence
- Sequence $<P_1, P_2, \ldots, P_n>$ is safe if, 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 $j<i$)
  - 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.

Safe State, Unsafe State & Deadlock

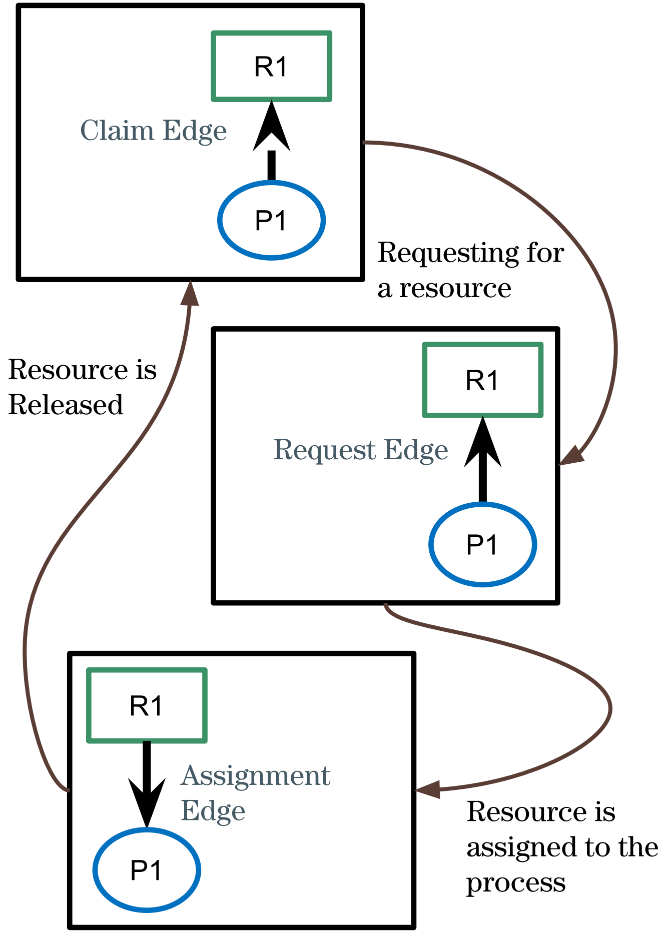
- Safe State => No Deadlock!
- Unsafe State => Possibility of Deadlock!
- To avoid deadlock, always ensure that system is in safe state.



Resource-Allocation-Graph Algorithm

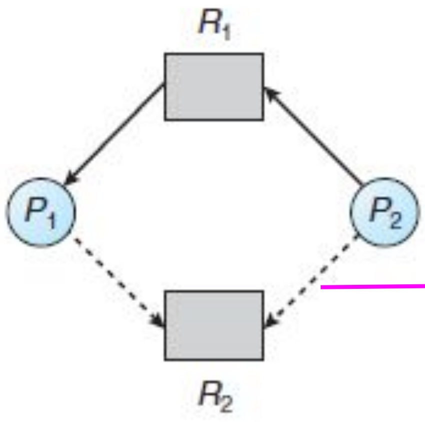
(Deadlock Avoidance Algorithm)

- Used when the system has only one instance of each resource type.
- Request Edge and Assignment Edges exist here similar to Resource allocation graph
- Claim Edge: claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at some time in the future
- When process P_i requests resource R_j, the claim edge $P_i \rightarrow R_j$ is converted to a request edge.
- When a resource R_j is released by P_i, the assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$
- Resources must be claimed a priori in the system. i.e. before process P_i starts executing, all its claim edges must appear in the graph



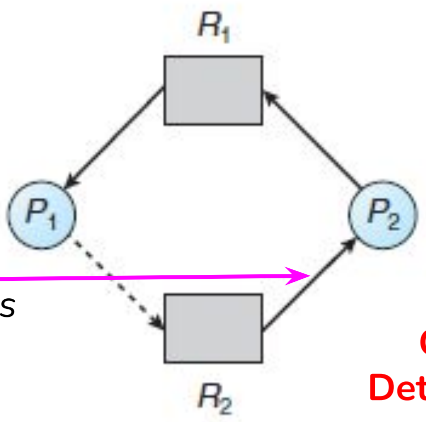
Resource-Allocation-Graph Algorithm

(Deadlock Avoidance Algorithm)



Resource-allocation graph for deadlock avoidance

Assigning the resource to the requesting process



Cycle Detected !!!

An unsafe state in a resource-allocation graph

Operating Systems

Bankers Algorithm

Banker’s Algorithm (Deadlock Avoidance Algorithm)

- Multiple instances of each resource type.
- Each process must a priori claim maximum use.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time.

Data Structures for the Banker's Algorithm

Need the following data structures, where **n** is the number of processes in the system and **m** is the number of resource types.

- ❑ `Available:` Array of length **m,** indicates the number of available resources of each type.
- ❑ `Max:` **n** by **m** matrix, defines maximum demand of each process for each type.
- ❑ `Allocation:` **n** by **m** matrix, defines number of assigned resources to each process for each type.
- ❑ `Need:` **n** by **m** matrix, defines number of remaining resources of each type needed by each process.

```
Need [i,j] = Max[i,j] – Allocation [i,j]
```

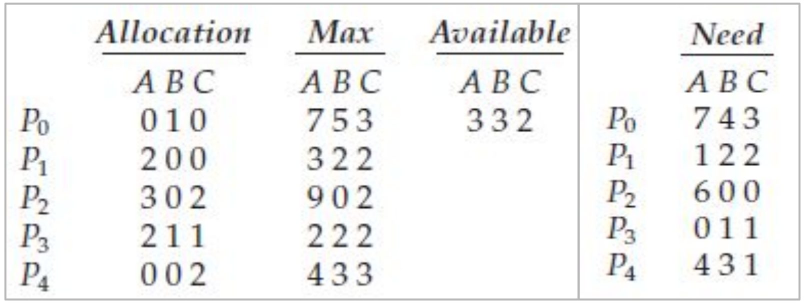
Banker's Safety Algorithm

Algorithm for finding out whether or not a system is in a safe state

1. Let **Work** and **Finish** be vectors of length m and n, respectively. Initialize
   `Work = Available` and
   `Finish[i] = false` for i = 0, 1, …, n – 1.
2. Find an index *i* such that both of the following meets,
   `Finish[i] == false`
   $\texttt{Need}_i \leq \texttt{Work}$
   if no such *i* exists, go to step 4.
3. $\texttt{Work} = \texttt{Work} + \texttt{Allocation}_i$
   `Finish[i] = true`
   Go to step 2.
4. If `Finish[i] == true` for all ***i***, then the system is in a safe state.

Banker's Algorithm Example

- 5 processes P0 through P4;
- 3 resource types A (10 instances), B (5 instances), and C (7 instances)
- at time T0, the following snapshot of the system:



	Allocation	*Max*	*Available*		*Need*
	A B C	*A B C*	*A B C*		*A B C*
P_0	0 1 0	7 5 3	3 3 2	P_0	7 4 3
P_1	2 0 0	3 2 2		P_1	1 2 2
P_2	3 0 2	9 0 2		P_2	6 0 0
P_3	2 1 1	2 2 2		P_3	0 1 1
P_4	0 0 2	4 3 3		P_4	4 3 1

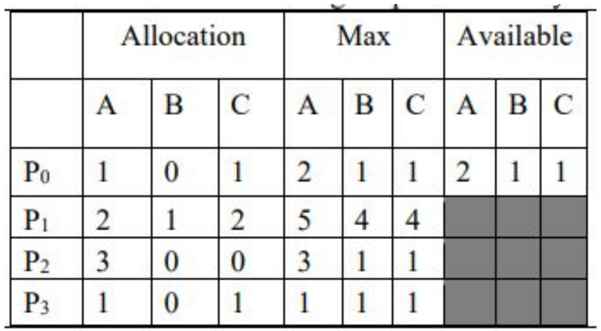
finish:

work:

```
if finish[i] == false & Need_i ≤ Work then
        Work = Work + Allocation_i
        Finish[i] = true
else
        wait
```

The system is in a safe state since the sequence < **P_1, P_3, P_4, P_2, P_0**> satisfies safety criteria

Practice



	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			

Resource-Request Algorithm for Process P_i

Request = request vector for process P_i

If `Request[i][j] = k` then process P_i wants k instances of resource type R_j

1. If $Request_i <= Need_i$ go to step 2.
   Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i <= Available_i$, go to step 3.
   Otherwise P_i must wait, since resources are not available.
3. 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;
```

4. Check Bankers Safety Algorithm, if this new state is safe.
   - ➔ If safe the resources are allocated to P_i
   - ➔ If unsafe P_i must wait, and the old resource-allocation state is restored

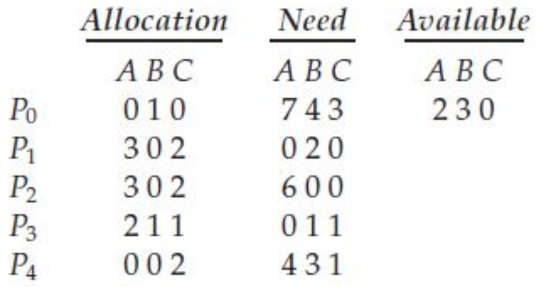
What if P_1 Request (1,0,2) more ?

- To decide whether this request can be immediately granted,
  - we first check that **$\text{Request}_1 \leq \text{Need}_1$** — that is, $(1,0,2) \leq (1,2,2)$? => TRUE
  - Then we check that **$\text{Request}_1 \leq \text{Available}$** — that is, $(1,0,2) \leq (3,3,2)$? => TRUE
  - If both is true,

    $\text{Available} = (3,3,2) - (1,0,2) = (2,3,0)$

    $\text{Allocation}_1 = (2,0,0) + (1,0,2) = (3,0,2)$

    $\text{Need}_1 = (1,2,2) - (1,0,2) = (0,2,0)$
- So, we arrive at the following new state



	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Is the system in safe state?
Yes. Because we find a safe sequence: $<P_1, P_3, P_4, P_0, P_2>$.
Grant the request of P_1.

What about request of (3 3 0) resources for P4?
What about request of (0 2 0) resources for P0?

IT'S OVER
IT'S DONE