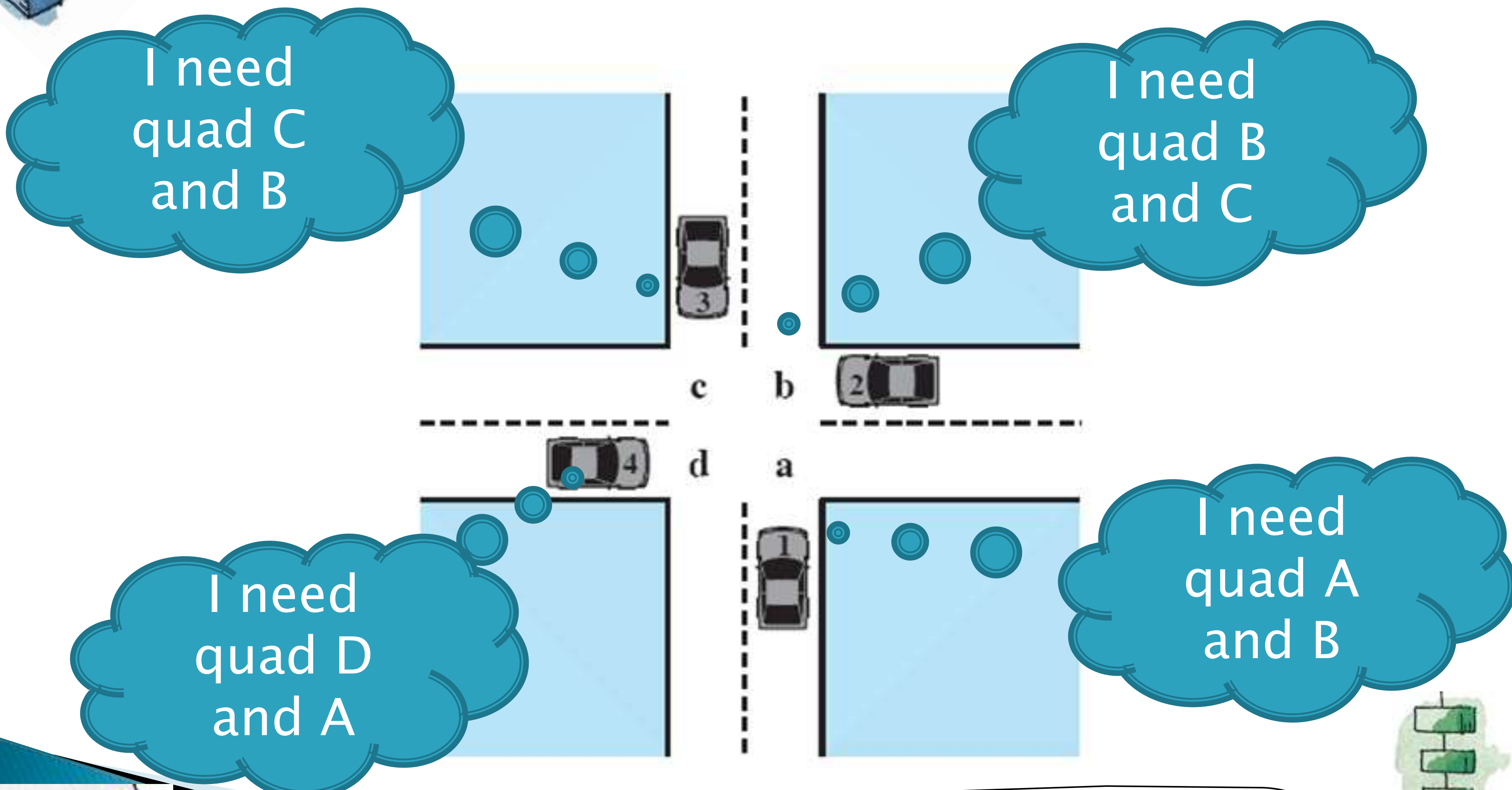
# Deadlock

Course Instructor: Nausheen Shoaib

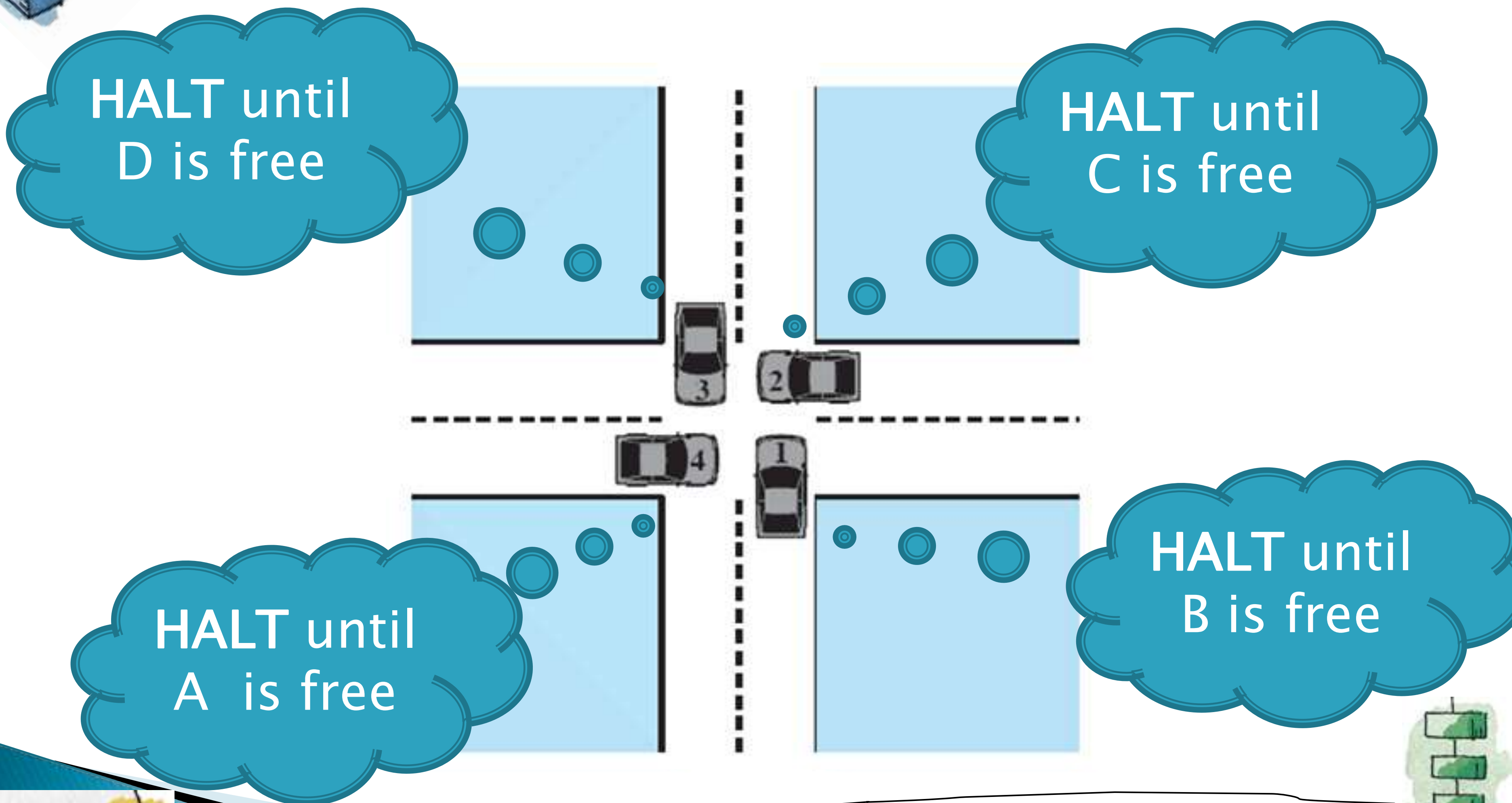
# Deadlock

- A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set
  - Typically involves processes competing for the same set of resources
- No efficient solution

# Potential Deadlock

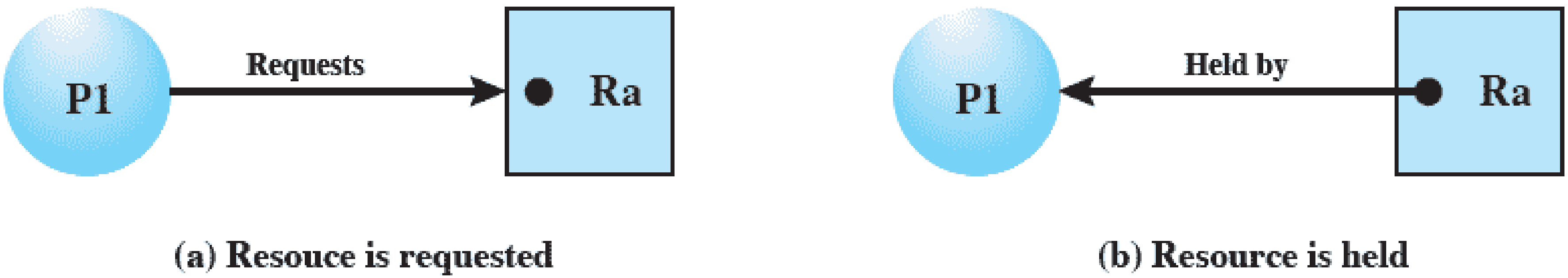


# Actual Deadlock

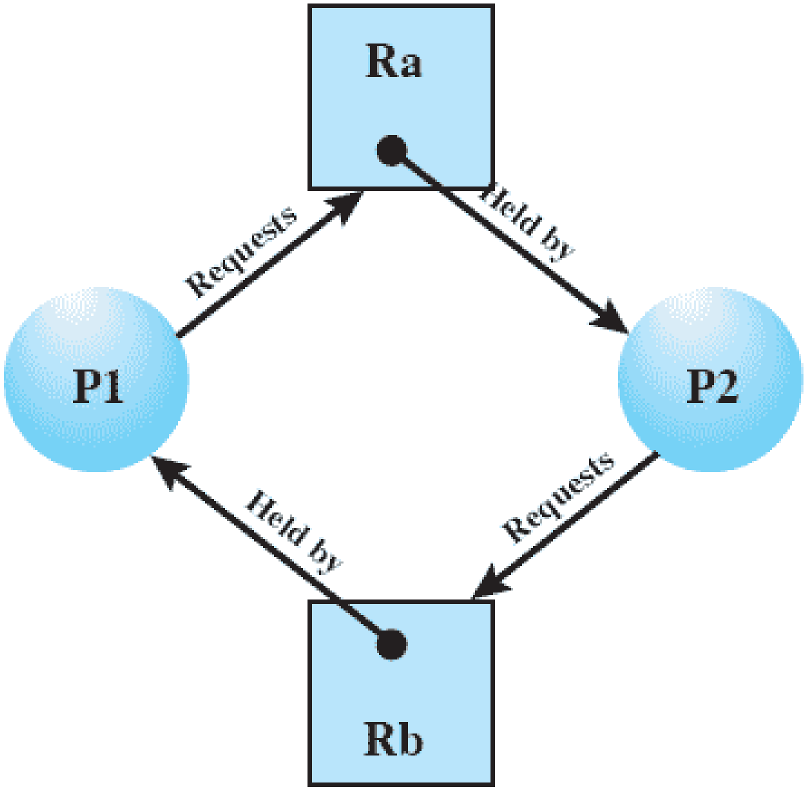


# Resource Allocation Graphs

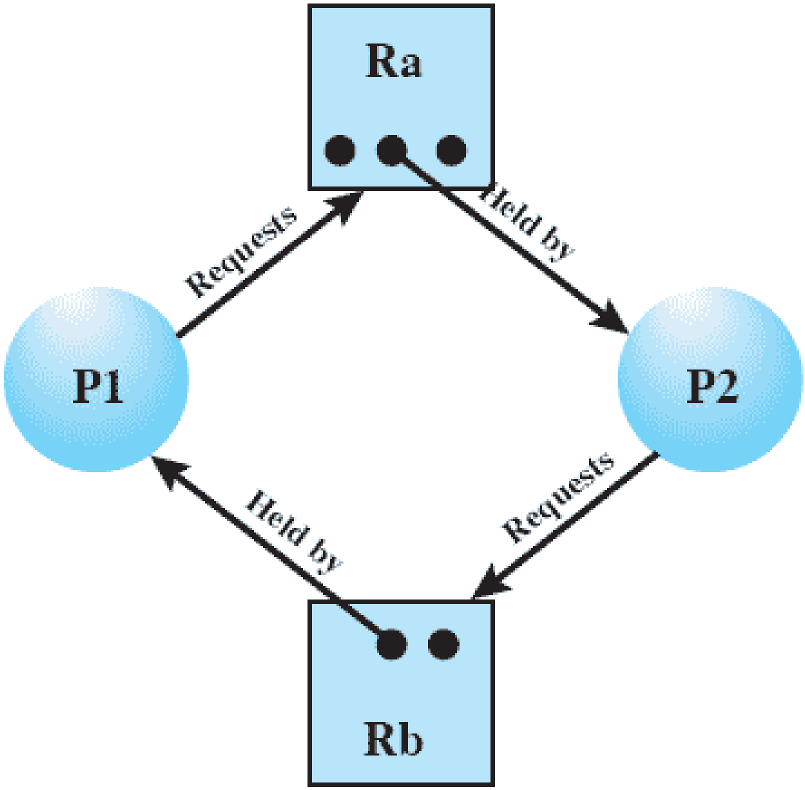
- Directed graph that depicts a state of the system of resources and processes



# Resource Allocation Graphs of deadlock




(c) Circular wait

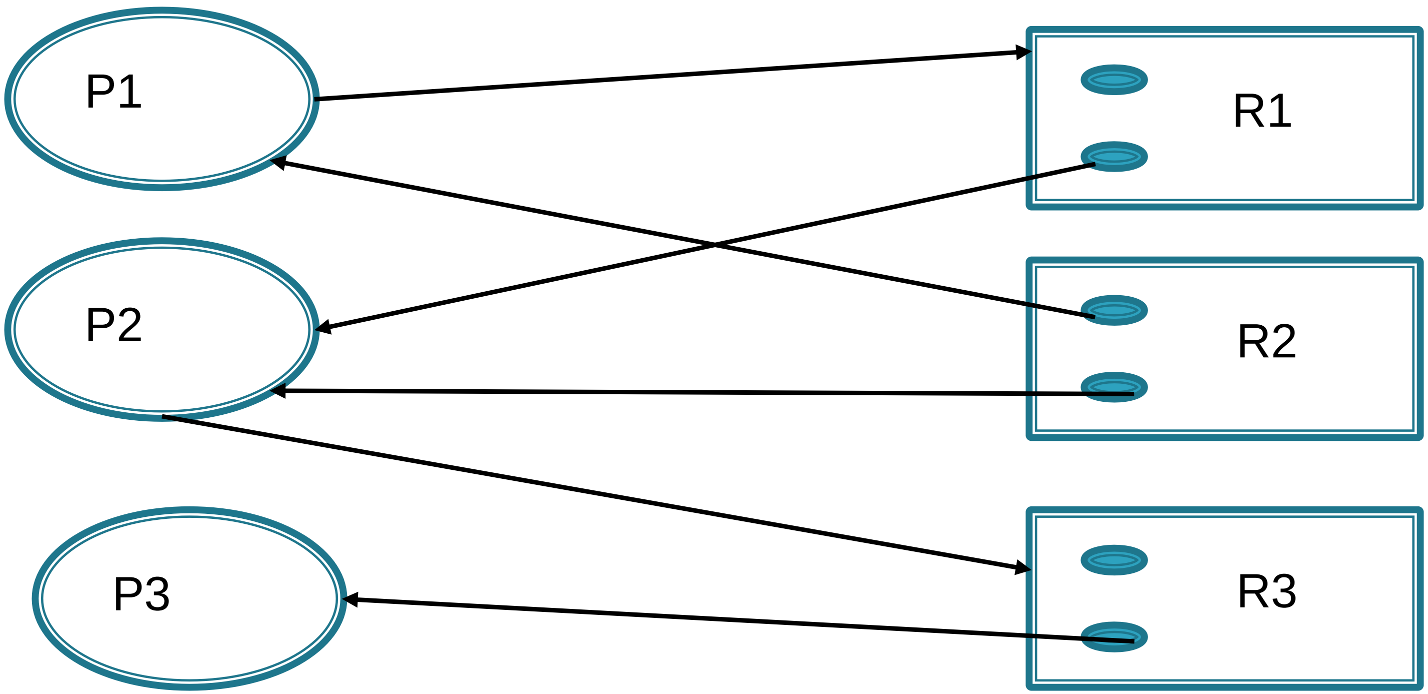



(d) No deadlock

# Resource Allocation Graphs of deadlock

- Deadlocks can be described more precisely in terms of a directed graph called
  a **system resource-allocation graph**

# RAG



# Dealing with Deadlock

- Three general approaches exist for dealing with deadlock.
  - Prevent deadlock
  - Avoid deadlock
  - Detect Deadlock

# Deadlock Prevention Strategy

- Design a system in such a way that the possibility of deadlock is excluded.

- Two main methods

  - Indirect – prevent one of the three necessary conditions from occurring

  - Direct – prevent circular waits

# Deadlock Prevention Conditions

- Mutual Exclusion
- Hold and Wait
- No Preemption
- Circular Wait

# Deadlock Prevention Conditions

## Mutual Exclusion

- This condition says, "There exist resources in the system that can be used by only one process at a time."
- Examples include printer, write access to a file or record, entry into a section of code
- Best not to get rid of this condition
  - some resources are intrinsically nonsharable

# Deadlock Prevention Conditions

## Hold and Wait (1/2)

- This condition says, "Some process holds one resource while waiting for another."
- To attack the hold and wait condition:
  - Force a process to acquire all the resources it needs before it does anything; if it can't get them all, get none
- Each philosopher tries to get both chopsticks, but if only one is available, put it down and try again later

# Deadlock Prevention Conditions

## No Preemption (1/2)

- This condition says, “Once a process has a resource, it will not be forced to give it up.”
- To attack the no preemption condition:
  - If a process asks for a resource not currently available, block it but also take away all of its other resources
  - Add the preempted resources to the list of resource the blocked process is waiting for

# Deadlock Prevention Conditions

## Circular Wait (1/2)

- This condition says, "A is blocked waiting for B, B for C, C for D, and D for A"
- Note that the number of processes is actually arbitrary
- To attack the circular wait condition:
  - Assign each resource a priority
  - Make processes acquire resources in priority order

# Deadlock Avoidance

- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock

- Requires knowledge of future process requests

# Two Approaches to Deadlock Avoidance

- Process Initiation Denial

- Resource Allocation Denial

# Process Initiation Denial

- A process is only started if the maximum claim of all current processes plus those of the new process can be met.
- Not optimal,
  - Assumes the worst: that all processes will make their maximum claims together.

# Resource Allocation Denial

- Referred to as the banker's algorithm
  - A strategy of resource allocation denial
- Consider a system with fixed number of resources
  - *State* of the system is the current allocation of resources to process
  - *Safe state* is where there is at least one sequence that does not result in deadlock
  - *Unsafe state* is a state that is not safe

# Basic Facts for deadlock avoidance

- 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.

# Banker's Algorithm

- Multiple instances
- 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

Let $n$ = number of processes, and $m$ = number of resources types.

- **Available**: Vector of length $m$. If available [$j$] = $k$, there are $k$ instances of resource type $R_j$ available
- **Max**: $n \times m$ matrix. If *Max* [$i,j$] = $k$, then process $P_i$ may request at most $k$ instances of resource type $R_j$
- **Allocation**: $n$ x $m$ matrix. If Allocation[$i,j$] = $k$ then $P_i$ is currently allocated $k$ instances of $R_j$
- **Need**: $n$ x $m$ matrix. If *Need*[$i,j$] = $k$, then $P_i$ may need $k$ more instances of $R_j$ to complete its task

$$Need\,[i,j] = Max[i,j] - Allocation\,[i,j]$$

# Example of Banker's Algorithm

- Discussed in Class

# Resource-Request Algorithm for Process $P_i$

***Request**$_i$* = 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 $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \leq Available$, 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:

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

- If safe $\Rightarrow$ the resources are allocated to $P_i$
- If unsafe $\Rightarrow$ $P_i$ must wait, and the old resource-allocation state is restored

# Example of Resource Request Algorithm

- Discussed in Class