University College Dublin
An Coláiste Ollscoile, Baile Átha Cliath

Operating Systems Deadlock and Starvation

Dr. Vivek Nallur (vivek.nallur@ucd.ie)

Deadlock and Starvation

Problems

- Mutual exclusion mechanisms for synchronisation guarantee that processes do not clash when used **shared resources**
- But there can still be problems

Deadlock

- A set of processes is in a deadlock state when every process is **blocked** forever, waiting for the availability of resources held by **other processes**

Starvation

- When a process waits for resources that periodically become available, but are never allocated to that process due to some scheduling policy

Resources

- Processes are assigned resources when they request for them
- There are two types of resources:
  - Reusable resources
  - Consumable resources

Reusable resources

- Resources used by only **one** process at a time, and not depleted by that use
- After use, they are released for **reuse** by other processes

Reusable resources

- Examples:
  - Processors,
  - Main and secondary memory,
  - Devices,
  - Data structures such as databases and semaphores

Consumable Resources

- Resources **created** (produced) by one process and **destroyed** (consumed) by another
- Infinite number of instances
- No need to release them

Consumable Resources

- Examples
  - Signals
  - Interrupts
  - Messages

Resources

- Deadlock and starvation are possible with **both** types of resources

Reusable Resource Example

- There are 200MB of memory **available** for use
- Processes P1 and P2 both request some memory

Reusable Resource Example

P1	**P2**
…	…
Request 80MB	Request 70MB
…	…
Request 60MB	Request 80MB
…	…

Reusable Resource Example

- Both processes are designed correctly, and hence **neither requests more** than the total available
- What happens if they both process get to the second request at the same time?
  - The initial memory request is **not released**

Reusable Resource Example

- The result is **deadlock:**
  - There is **not enough memory** for either request to be satisfied
  - No process will **release** the memory it has until it has completed its task

Consumable Resource Example

- Consumable resource: **message**
  - We may send and receive as many as we want: they are produced and consumed
- Assume that the receive operation is **blocking**

Consumable Resource Example

```
P1                      P2
…                       …
receive(mbox_2,M)       receive(mbox_1,N)
…                       …
send(mbox_1,N)          send(mbox_2,M)
…                       …
```

Consumable Resource Example

- Both processes request and release resources
- What happens if both processes call **receive** at the same time?

Consumable Resource Example

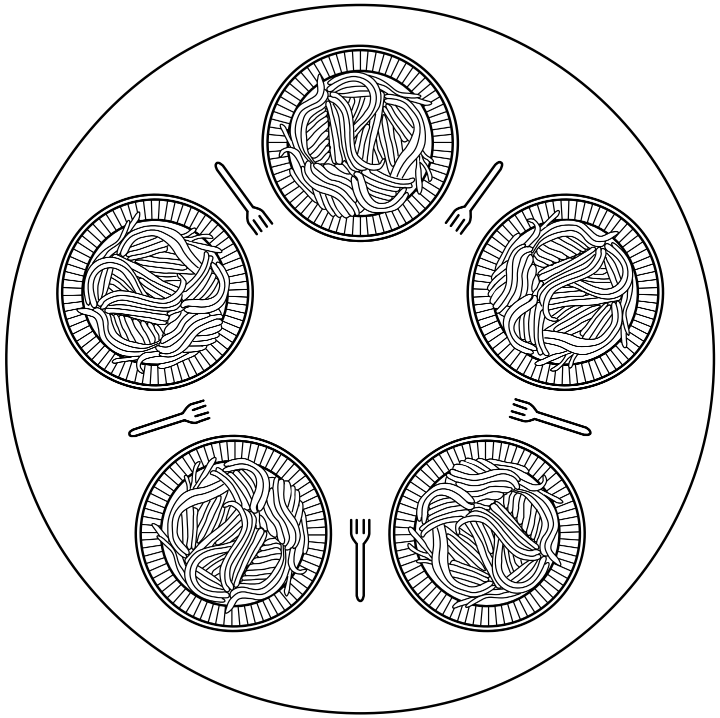
- The result is **deadlock**
  - P1 cannot proceed until P2 sends its message
  - P2 cannot proceed until P1 sends its message

Dining Philosophers

- Classic example proposed by Dijkstra to illustrate **deadlock** & **starvation**

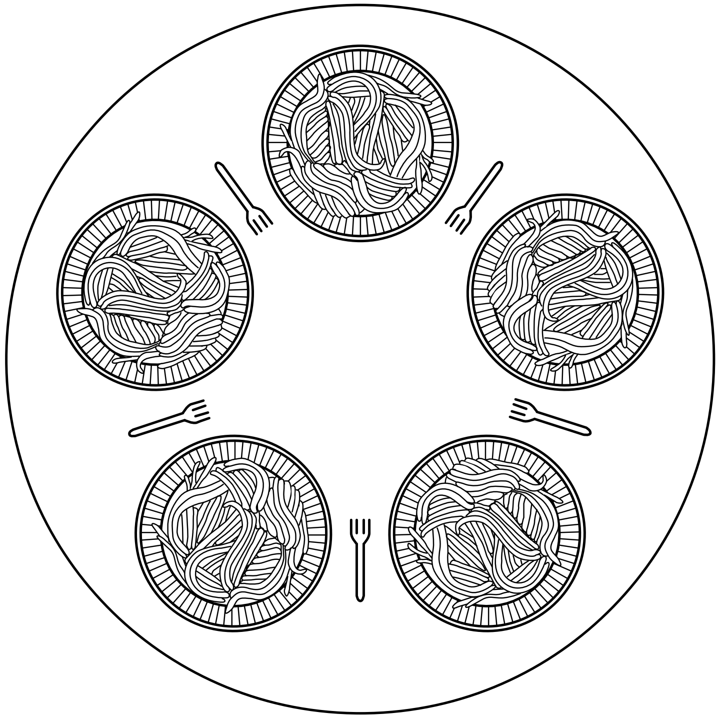
Dining Philosophers

- **5** philosophers are living together
- Philosophers can either be **eating** or **thinking**
- Each philosopher has a seat at a round table



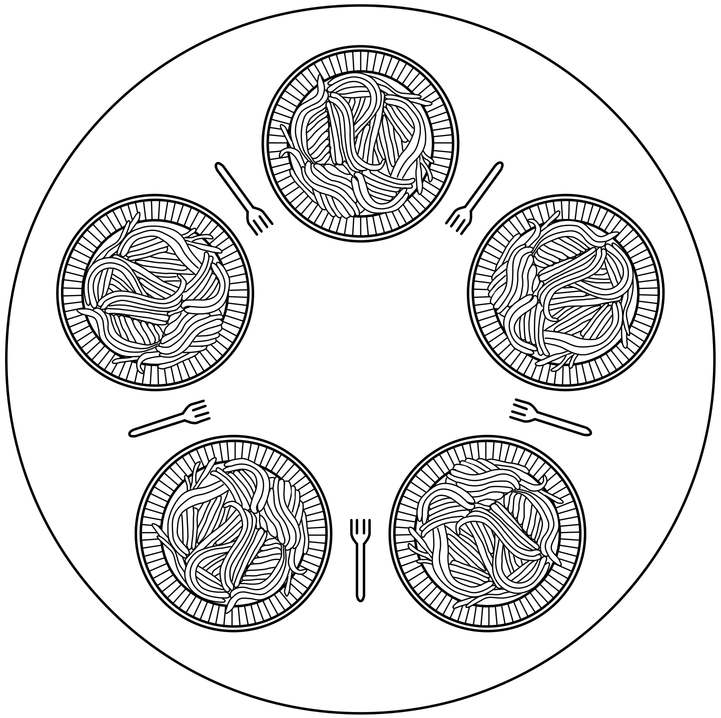
Dining Philosophers

- On the table there are **5** plates of spaghetti (noodles) and **5** forks
- A philosopher needs **both** forks beside their plate to eat



Dining Philosophers

- Two philosophers cannot use the **same fork** at the **same time**
- A philosopher is a **process**
- A fork is a **shared resource**



Semaphore Solution

- Semaphore for each resource
  - semaphore fork**[5]**(1,NULL)
- Try your fork first and then the fork beside you

```
Philosopher(int i)

 while(true) {
     think();
     P(fork[i]);
     P(fork[(i+1) mod 5]);
     eat spaghetti();
     V(fork[(i+1) mod 5]);
     V(fork[i]);
 }
```

Semaphore Solution

- Not a real solution:
- It may happen that all philosophers are simultaneously hungry and grab their left forks at the **same time**
- This leads to **deadlock**

Solving the deadlock

- Make the philosophers **release** their left fork if after having grabbed if they detect that the right fork is in use
- After waiting for some **fixed time** the philosopher would try again to grab both forks

Solving the deadlock

- Deadlock is solved, but **starvation** is still possible
  - If all philosophers start the algorithm at the **same time**, no philosopher ever grabs both forks
  - The philosophers are caught in an **endless cycle**

Solving the deadlock

- Starvation can be solved by having a **random wait** time before we try to pick up the forks again
- Someone will usually get there first and eat
  - No endless cycle

Semaphore for Four Philosophers

- Only allow 4 philosophers at the table at a time

- One philosopher will have access to two forks

```
semaphore fork[5](1,NULL);
semaphore table(4,NULL)
```

Philosopher(int I)

```
while(true) {
     think();
     P(table);
     P(fork[i]);
     P(fork[(i+1) mod 5]);
     eat spaghetti();
     V(fork[(i+1) mod 5]);
     V(fork[i]);
     V(table);
}
```

Semaphore for Four Philosophers

- No deadlock or starvation
- Every philosopher will **eventually** get both forks
    - They may have to wait a while

Other possible solutions

- Many **ad-hoc** solutions to deadlock and starvation are possible:

1. Limit the number of philosophers at the table
    - allow at most four philosophers at a time at the table
    - pass a token around the table so that only the philosopher holding the token can eat

Other possible solutions

2. Asymmetric solution:
   - even philosophers try left fork first
   - odd philosophers try right fork first

3. Use five counting semaphores, each counting the number of available forks per philosopher
   - Careful initialization needed

Other possible solutions

- We need some sort of formal model to handle the complexity of the problem

Resource-Allocation Graph

Resource-Allocation Graph

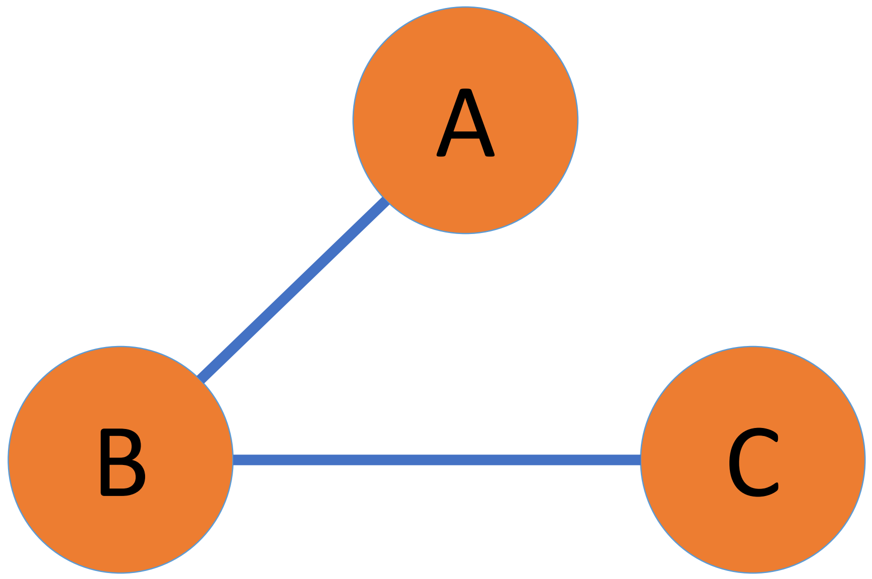
- Deadlocks can be described more formally in terms of a directed graph called **resource-allocation graph** (RAG)
- A RAG completely describes the state of a system in terms of
  - what resources are allocated to what processes
  - what processes are waiting for what resources

Graphs

- Graphs are a the combination of
    - A set of vertices (nodes)
    - A set of edges (lines connecting vertices)

Graph - Undirected

- Vertices = {A,B,C}
- Edges = {(A,B),(B,C)}

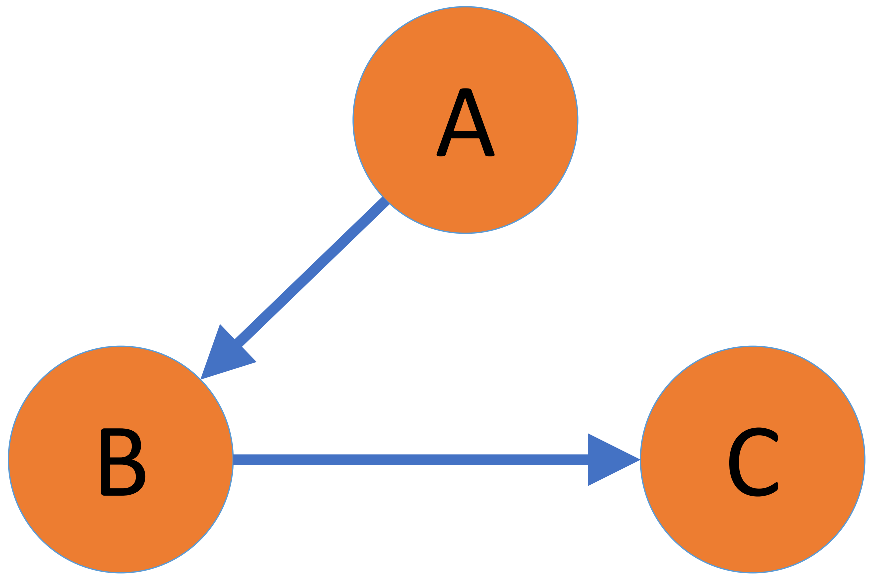


Directed Graphs

- Directed graphs are graphs in which the edges have a direction
- Edges are usually represented using arrows to show direction

Directed Graph Example

- Vertices = {A,B,C}
- Edges = {(A→B),(B→C)}

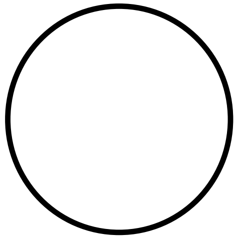


Resource-Allocation Graph

- A resource-allocation graph is a **special directed graph**, where both vertices and directed edges are partitioned into **two sets**

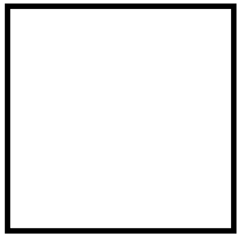
Vertices

- Vertices are split into to groups
  - Processes
    - **$P = \{P_1, . . . , P_n\}$**, all active processes in the system
    - Represented by:

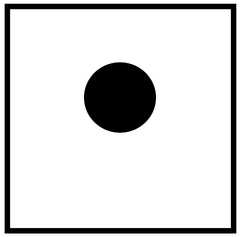


Vertices

- Resources
  - **$R = \{R_1, \ldots, R_m\}$**, all resource types in the system
  - Represented by:



  - Where there are a number of the same resource, each is represented by a

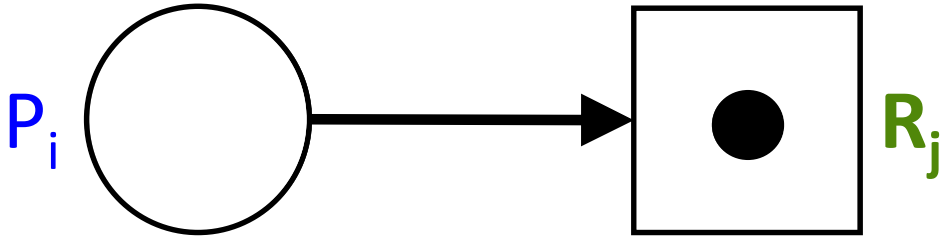


Edges

- There are also two sets of edges
  - Request edges
    - These represent a **request** for a resource by a process
  - Assignment edges
    - These represent the **allocation** of a resource to a process

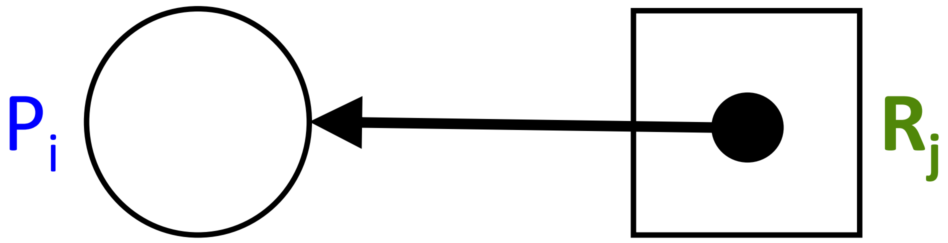
Request edges

- A request by process `i` for resource `j` looks like this
  - $P_i \rightarrow R_j$



Assignment edges

- The assignment of a resource `j` to a process `i` looks like this
  - $R_j \rightarrow P_i$



Resource-Allocation Graph Example

- Two processes **P_1** and **P_2**, two shared resources **R_1** and **R_2**
  - Each resource has one instance only

Resource-Allocation Graph Example

P_1

```
while(true) {
1:      request(R1);
2:      …
3:      request(R2);
4:      …
         release(R2);
5:      …
         release(R1);
}
```

P_2

```
while(true) {
1:      request(R2);
2:      …
3:      request(R1);
4:      …
         release(R1);
5:      …
         release(R2);
}
```

System States

State	Situation of P_1
0	Holds no Resources
1	Holds none, Requests R_1
2	Holds R_1
3	Holds R_1, requests R_2
4	Holds R_1 and R_2
5	Holds R_1, R_2 released

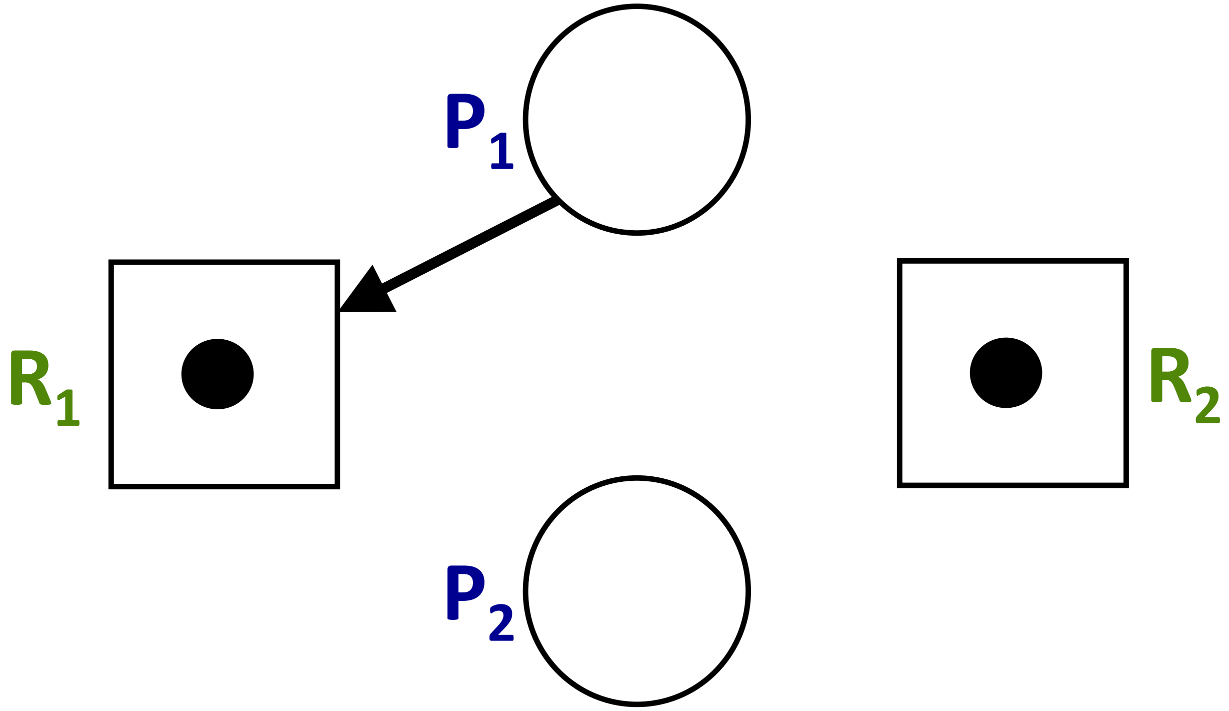
State	Situation of P_2
0	Holds no Resources
1	Holds none, Requests R_2
2	Holds R_2
3	Holds R_2, requests R_1
4	Holds R_1 and R_2
5	Holds R_2, R_1 released

System States

- Each possible pair of states is one resource-allocation graph
  - One from each process
- Some state pairs are not possible
  - State 4 from both process cannot happen at the same time

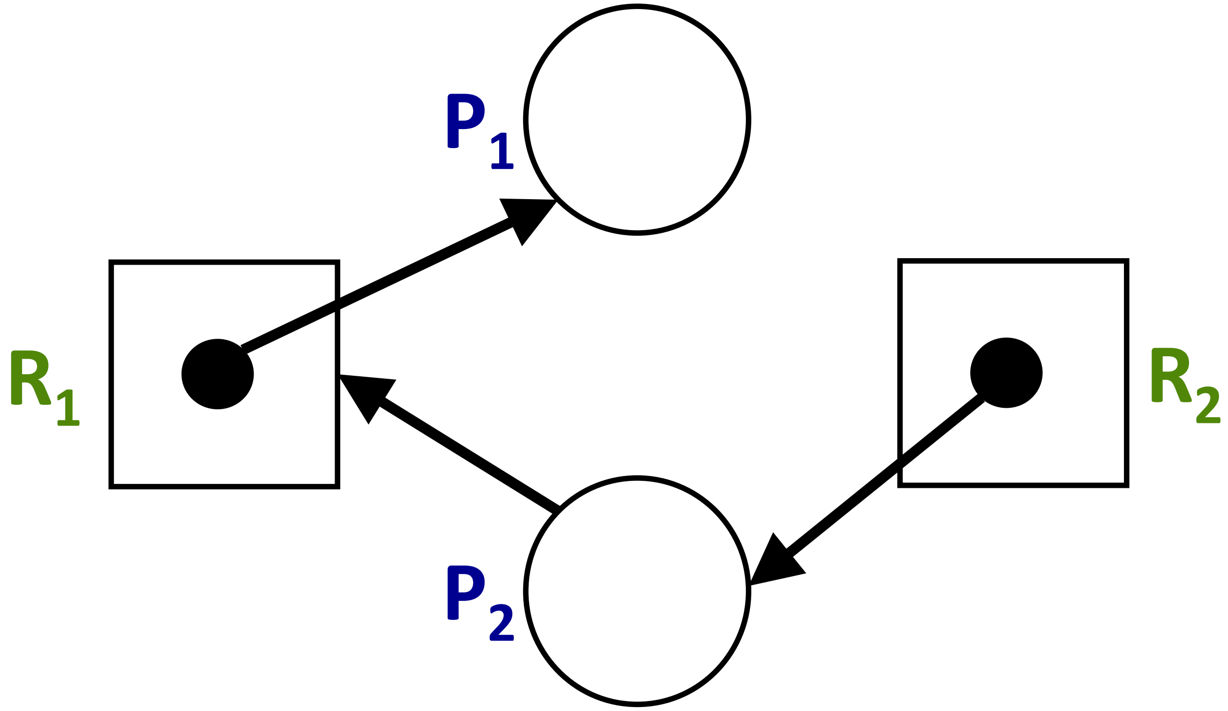
Resource-Allocation Graph Example

- (State P_1, State P_2)
- (1,0): **P_1** asks for **R_1** which is free



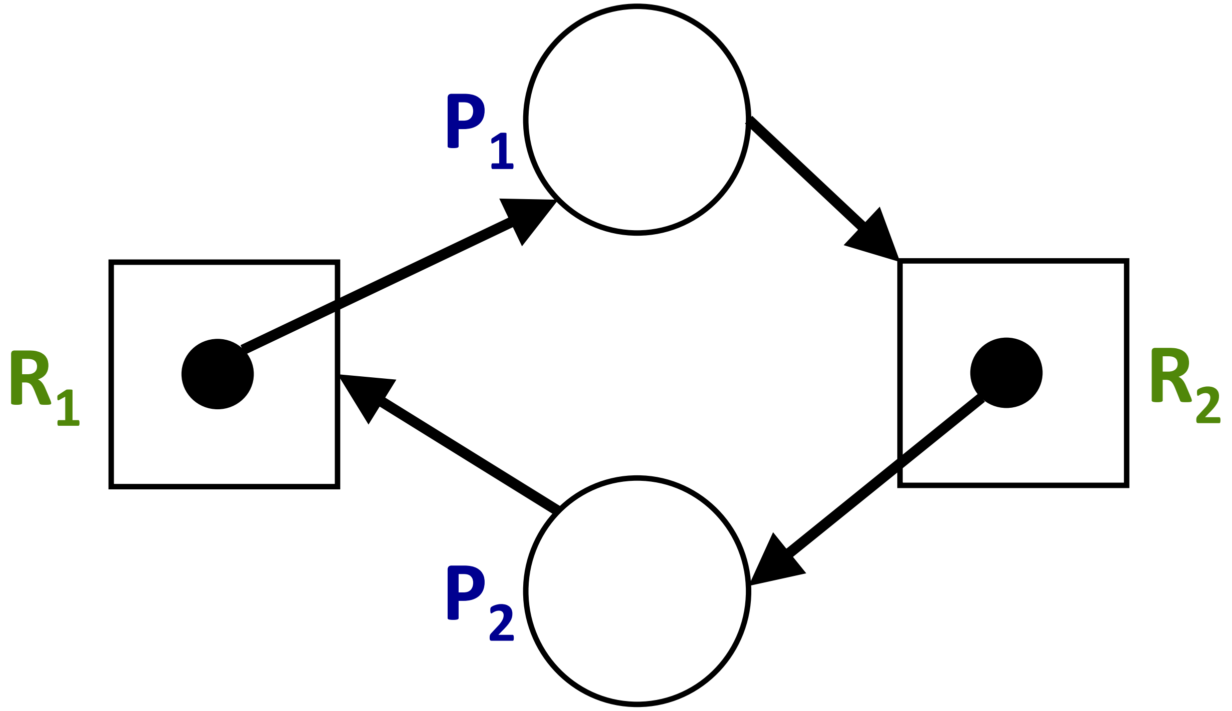
Resource-Allocation Graph Example

- (2,3): Not deadlock, but next step in P_1 leads to deadlock



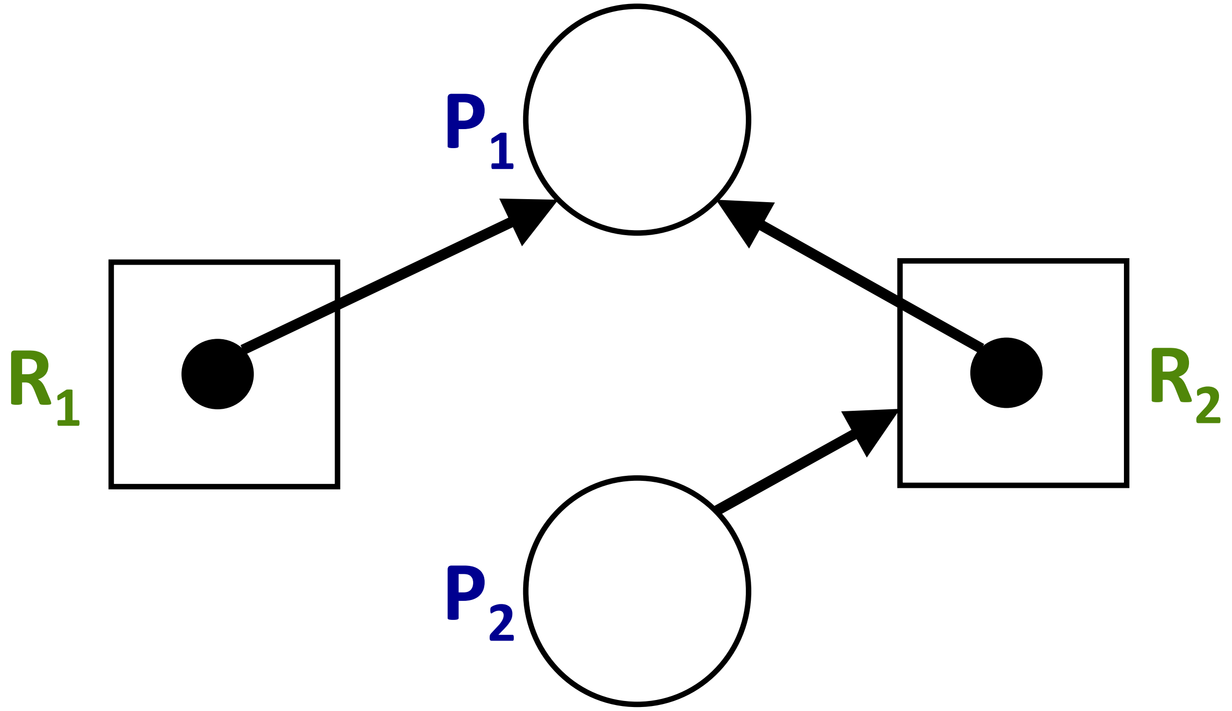
Resource-Allocation Graph Example

- (3,3): deadlock



Resource-Allocation Graph Example (alt if P2 slower)

- (4,1): $\mathbf{P_2}$ blocked (waiting for P_1 to release $\mathbf{R_2}$)



Necessary Conditions for Deadlock

1. Mutual Exclusion
   - At least one resource may be acquired exclusively by only one process at a time

2. Hold-and-wait
   - Processes may ask for resources while holding other resources

Necessary Conditions for Deadlock

3. No preemption
    - once allocated, a resource cannot be taken away from a process by the system or by other processes

4. Circular chain of request

Circular Chain of Request

- Two or more processes locked in a circular chain in which each process is **waiting** for one or more resources that the next process in the chain is holding
- The same as a **cycle** in the resource-allocation graph

Cycles in RAG and Deadlock

- If there are **no cycles** in the RAG: then there is **no deadlock**
  - A cycle is a necessary condition for a deadlock

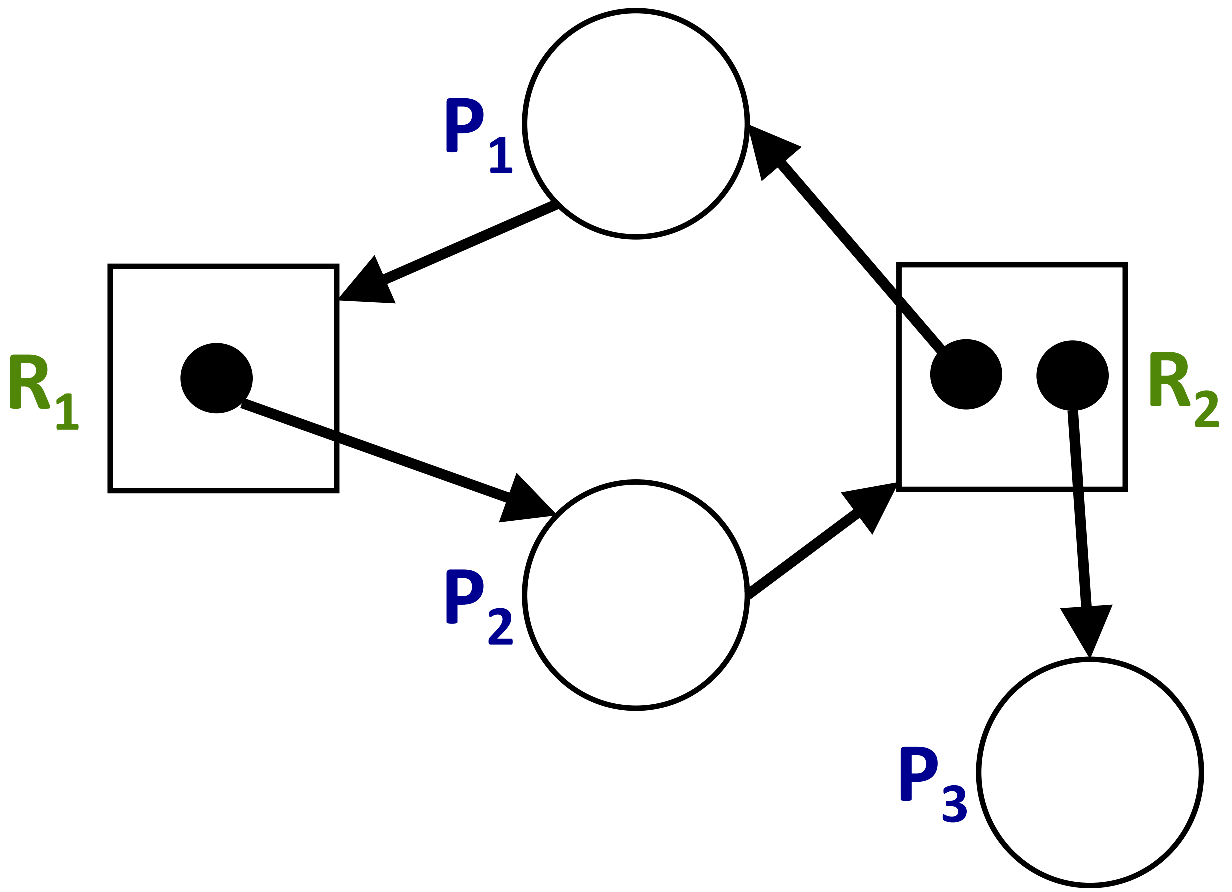
Cycles in RAG and Deadlock

- If there is a cycle in the RAG and there is a **single instance** of each resource in the cycle, then there is a deadlock

Cycles in RAG and Deadlock

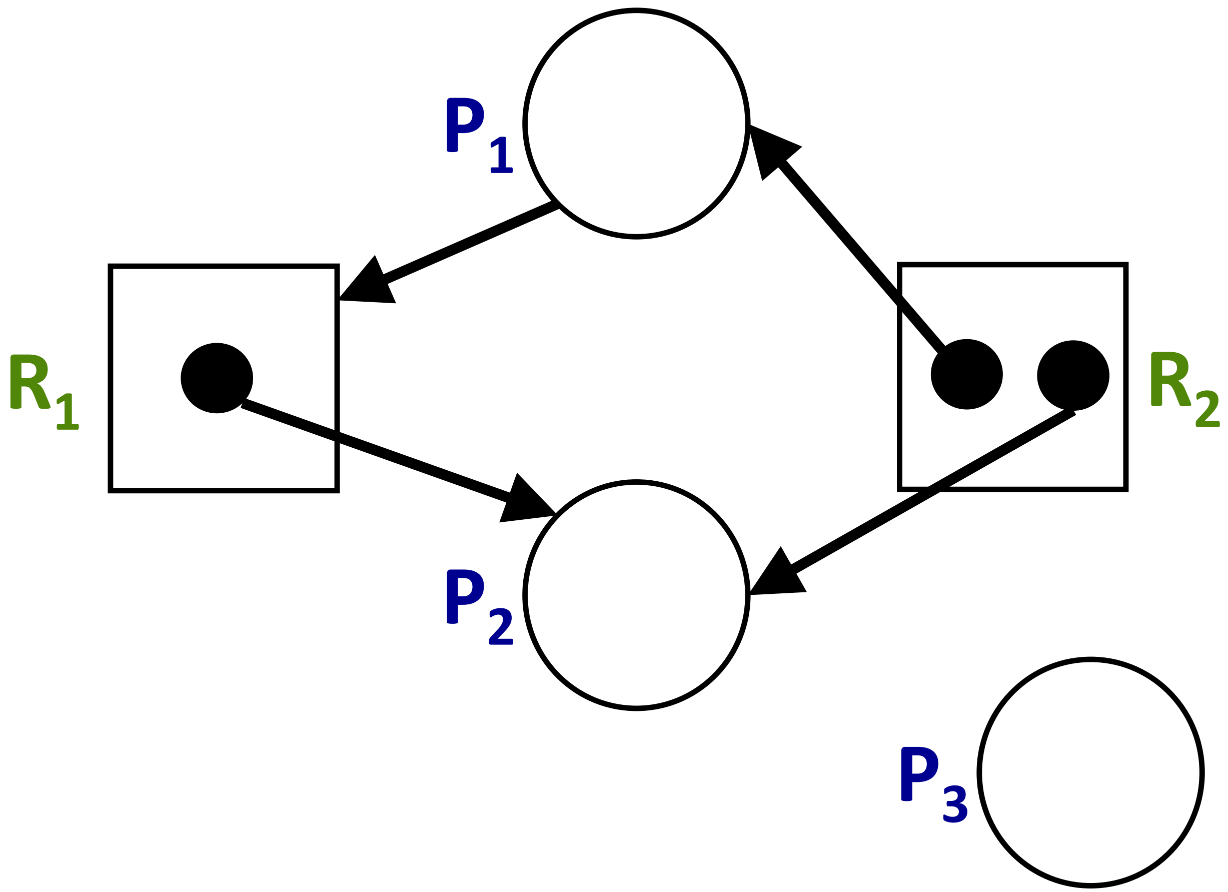
- If there is a cycle in the RAG and there are several instances of at least one resource in the cycle, then there **may** be a deadlock
- A cycle does not always mean there is a deadlock

Cycle with no Deadlock



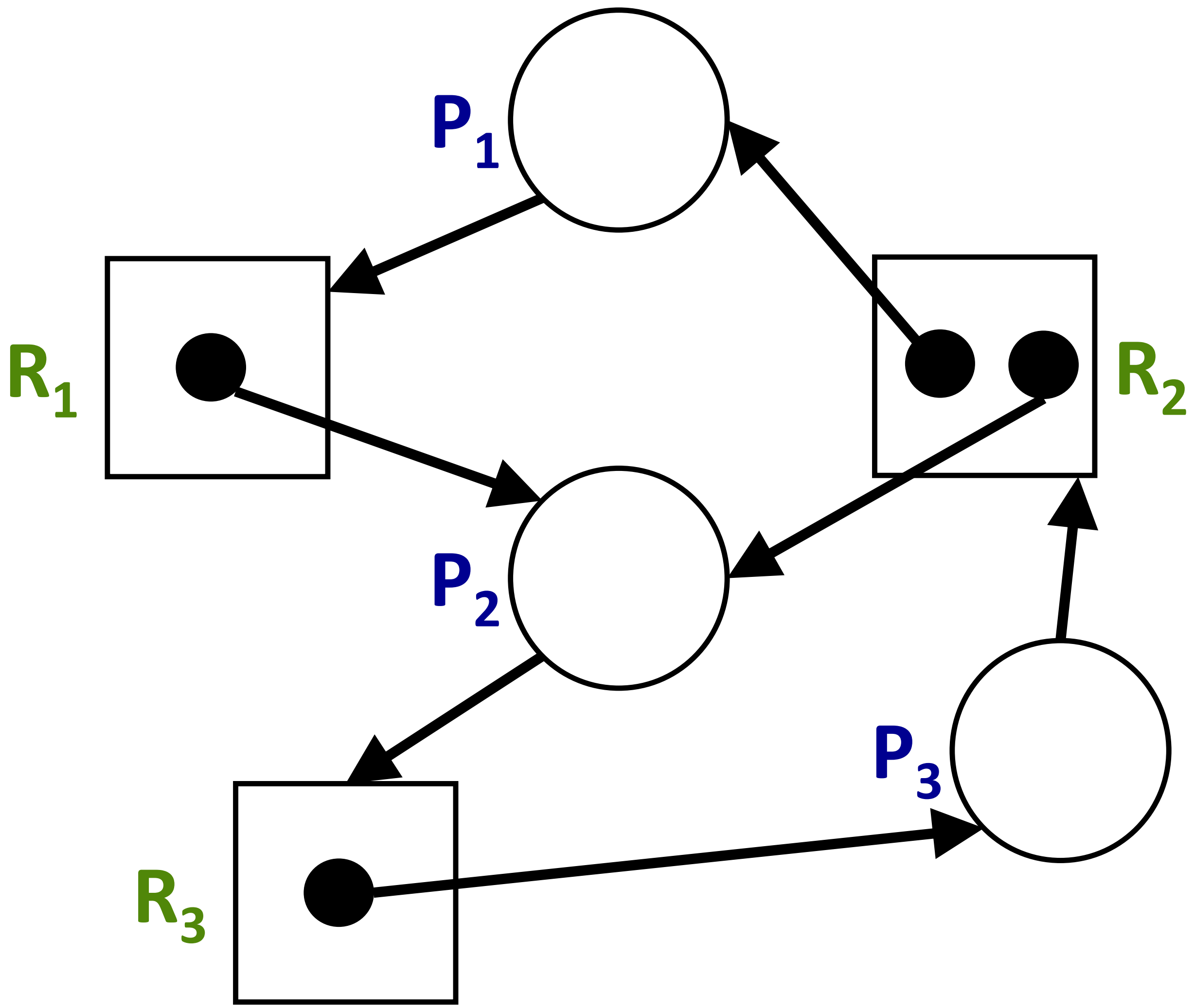
$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_2 \rightarrow P_1$

Cycle with no Deadlock



Cycle broken if P_3 releases R_2

Cycle with Deadlock



Handling Deadlock

Handling Deadlock

- Let deadlock occur, and do something about it afterwards
  - **deadlock detection & recovery**
- Never let deadlock occur
  - **deadlock prevention**
  - **deadlock avoidance**

Handling Deadlock

- Ignore the problem and pretend that deadlock never occurs
  - “strategy” used by many desktop operating systems
  - It can be **efficient**, if the probability of deadlock is low
  - It cannot be tolerated in **mission-critical** or **real-time** systems

Deadlock Detection

- **Detection**: scan the resource-allocation graph to find **cycles**
  - Periodically
  - Or during low system utilisation periods

Deadlock Recovery – Process Termination

- Abort one deadlocked process at a time until deadlock cycle eliminated
  - Costly, and maybe slow
  - deadlock detection after each termination
- Abort all deadlocked processes
  - Very costly, faster
- System reboot
  - Even more costly, but fastest

Deadlock Recovery – Resource Preemption

- This means we successively preempt processes from resources until deadlock cycle broken

- There are some issues

Resource Preemption Issues

- **Victim selection order**: preempt according to cost function
- **Rollback**: victim must be rolled back to a safe prior state
  - information must be kept consistent
- **Starvation**: will resources always be preempted from the same process?
  - We must take this into account in cost function

Deadlock Prevention

- Deadlock prevention policies are based on eliminating the possibility of **at least one** of the necessary conditions for deadlock.

Necessary Conditions for Deadlock

1. Mutual Exclusion
2. Hold-and-wait
3. No preemption
4. Circular chain of request

Deadlock Prevention

- Always **avoid** mutual exclusion
  - Some resources can be **shared** by unlimited number of processes
    - Read-only files
  - Some resources are not sharable
    - Printers

Deadlock Prevention

- Always avoid hold-and-wait
  - Don't allow **waiting** for a resource while holding resources; or
  - Have each process request and be allocated all its resources **before** execution
- Low resource utilization, starvation possible

Deadlock Prevention

- Always allow preemption
  - E.g. preempt main memory to disk
  - Some resources are not preemptible
- All prevention methods lead to an **inefficient** use of the system

Deadlock Avoidance

Deadlock Avoidance

- Deadlock avoidance policies are based on the system having **a priori** information available

- *A priori information*: processes declare the **maximum** number of resources of each type that they may need at the start

Deadlock Avoidance

- Deadlock-avoidance algorithms dynamically monitor the system state to ensure no circular waits based on this information

Deadlock Avoidance Issues

- Hard to implement, as we need to accurately predict the future
- It assumes processes eventually release their resources, but this could be a long time

Safe State

- Safe state of a system is a state in which resources can be allocated to each process (up to maximum requested) while avoiding deadlock

Safe State

- A state is safe if there exists a safe sequence of processes

  < P1,P2,...,Pn > such that

- The resource requests that P_i can make are satisfiable by
  - Currently available resources,
  - Plus resources held by P_1, P_2,...,P_{i-1}

Safe State

- A safe sequence is an **ordered** arrangement of all processes
- We create it sequentially, starting with one process, then two, etc

Safe State

- If no safe sequence exists, the system state is **unsafe**
- State safety is a worst-case analysis:
  - In safe state deadlock is impossible
  - In unsafe state it is possible (but not sure)

Bankers Algorithm

Bankers Algorithm

- It allows to check whether satisfying a request for resources will put the system in a safe state or not
  - The request is only satisfied if the new state is safe
  - It is a conservative algorithm

Bankers Algorithm

- The banker’s algorithm involves two sub-algorithms:
  - safety algorithm: to verify that a system state is safe
  - resource-request algorithm: to verify whether allocating the requested resources will take the system to a new safe state

Structures & Definitions

- Assume **n** processes and **k** resources
- Data structures required by the algorithm:
  - Availability vector
  - There are v_j instances of R_j currently available

$$\vec{v} = \left[v_1, v_2, \ldots, v_k \right]$$

Structures & Definitions

- **Allocation matrix**

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,k} \\ a_{2,1} & a_{2,2} & \dots & a_{2,k} \\ \dots & \dots & \dots & \dots \\ a_{n,1} & a_{n,2} & \dots & a_{n,k} \end{bmatrix}$$

- P_i is currently allocated $a_{i,j}$ instances of R_j

Structures & Definitions

- **Maximum matrix**:

$$M = \begin{bmatrix} m_{1,1} & m_{1,2} & \dots & m_{1,k} \\ m_{2,1} & m_{2,2} & \dots & m_{2,k} \\ \dots & \dots & \dots & \dots \\ m_{n,1} & m_{n,2} & \dots & m_{n,k} \end{bmatrix}$$

  - P_i may request at most $m_{i,j}$ instances of R_j

Structures & Definitions

- Definition:
  - This is the ith row of A

$$\vec{a}_i = \left[a_{i,1}, a_{i,2}, \ldots, a_{i,k}\right]$$

- For $\vec{x}$ and $\vec{y}$ of same size: $x_j \leq y_j$ means that $\vec{x} \leq \vec{y}$ for all j

Safety Algorithm

- Is a given state safe?
- To answer we need to try to sequentially find a safe sequence of processes

Safety Algorithm

$$\text{Define } \vec{w} = \vec{v}$$

- This means currently available resources, plus resources held by previous processes in the sequence

- Repeat following steps until finished

Safety Algorithm

1. Find P_i such that
   - We have not assigned a place in the sequence to P_i yet and

     $$\vec{m}_i - \vec{a}_i \leq \vec{\mathrm{W}}$$

   - Potential needs of P_i are smaller than or equal to $\vec{\mathrm{W}}$
   - If no such P_i exists, finish

1. Otherwise $\vec{\mathrm{W}} = \vec{\mathrm{W}} + \vec{a}_i$

Safety Algorithm

- If we have assigned a place in the sequence to every P_i then the system is in a safe state

Safety Algorithm Notes

- Based on a direct application of the formal definition of safe state
- The safety algorithm produces a sequence of P_i
  - E.g. with four processes we might obtain
    - $<P_4,P_2,P_1,P_3>$
- Any complete sequence obtained through the algorithm is safe

Resource-Request Algorithm

- Can we safely grant a request for resources?
- Let $\vec{r}_i = \left[r_{i,1}, r_{i,2}, \ldots, r_{i,k} \right]$ be the resource request vector of P_i
    - This means P_i wants $r_{i,j}$ instances of R_j

Resource-Request Algorithm

- First we check $\vec{r}_i + \vec{a}_i \leq \vec{m}_i$
- And $\vec{r}_i \leq \vec{v}_i$
- If either of these are not true then we stop

Resource-Request Algorithm

- $\vec{r}_i + \vec{a}_i > \vec{m}_i$

means that Pi is requesting more that it initially stated

$$\vec{r}_i > \vec{v}_i$$

- means that it is requesting more resources than are available

Step 1

- Consider the **tentative** new system state given by the following:
-
  - What if we allocate the request to P_i

$$\vec{a}_i = \vec{a}_i + \vec{r}_i$$

-
  - What if resources allocated to P_i are no longer available

$$\vec{v} = \vec{v} - \vec{r}_i$$

Step 2

- Check if the tentative state is safe
  - If it is, then P_i is allocated the requested resources
    - The tentative state becomes the new state
  - If it is not, then P_i must wait
    - The system stays in the same state

Next Class

- Task:
    - Review Chapter 7
- Next Lecture will be :
    - Process Scheduling