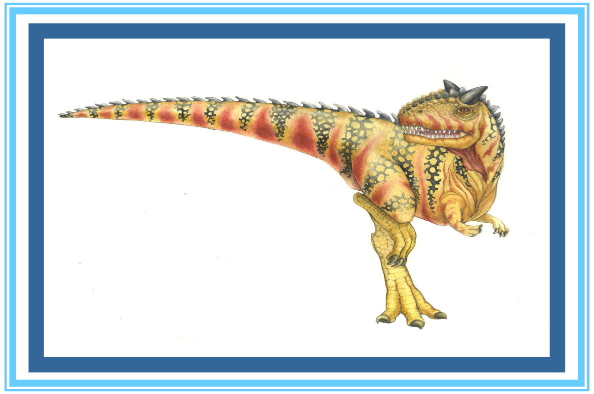
Chapter 7: Deadlocks

School of Computing, Gachon Univ.
Jungchan Cho



Most slides from "Operating System Concepts – 10th Edition".
Many slides are taken from lecture notes of Prof. Joon Yoo.

Chapter Objectives

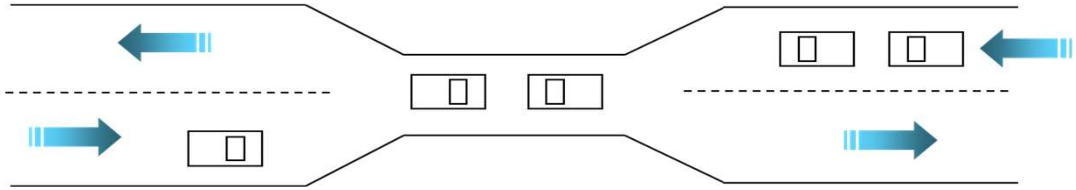
- To learn the concept of **deadlock**
- To develop a **characterization of deadlocks**, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for **preventing** or **avoiding** deadlocks in a computer system

Chapter 7: Deadlocks

- Deadlock Concept
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
  - Deadlock Prevention
  - Deadlock Avoidance

Bridge Crossing Example

- One-lane Bridge Example
  - A real-world example would be traffic, which is going only in one direction.
  - Here, a bridge is considered a resource.



https://commons.wikimedia.org/wiki/File:One_lane_bridge_over_West_River_on_Rice_Farm_Road.JPG

Deadlock Problems

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

Example1: semaphores *A* and *B*, initialized to 1

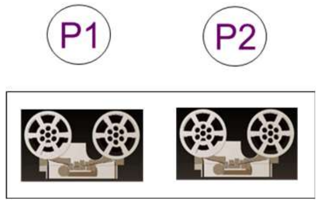
*P*0	*P*1
A0 wait (A);	B0 wait(B);
A1 wait (B);	B1 wait(A);

- Execution order : A0→B0→A1→B1

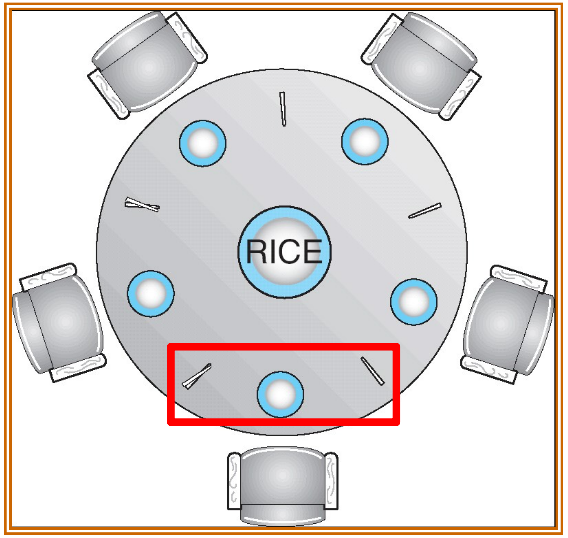
P0	P1
P (A);	*P(B)*
take one	*take one*
P (B);	*P(A)*
take another	*take another*

Example 2:

- System has a disk drive and a network card
- *P*1 holds the disk drive and needs the network card, *P*2 holds the network card and needs the disk drive
  - *P*1 : disk → network, *P*2; network → disk



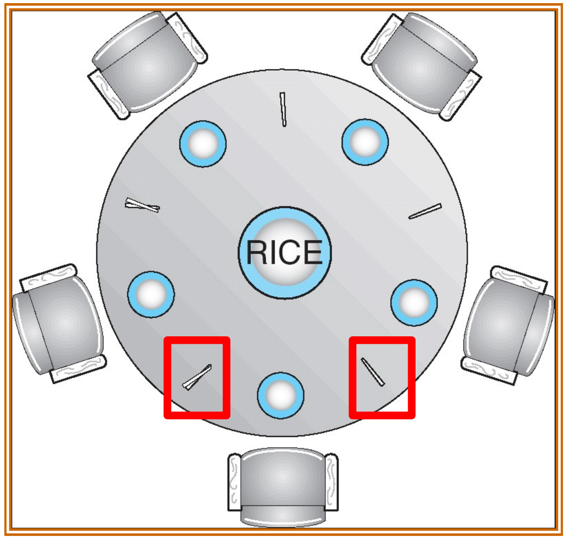
Dining-Philosophers Problem



- The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher
- In the center of table there is a bowl of rice, and the table is laid with 5 single chopsticks
- From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to them
- When a hungry philosopher has both her chopstick s at the same time, she eats without releasing her chopsticks
- When she is finished eating, she puts down both her chopsticks and starts thinking

Dining Philosophers Problem

- *N* philosophers and *N* chopsticks
- Philosophers **eat**, **think**
- Eating needs ***2*** chopsticks
- Pick up one chopstick at a time
  - First pick up left chopstick
  - Next pick up right chopstick
- Each chopstick used by one person at a time

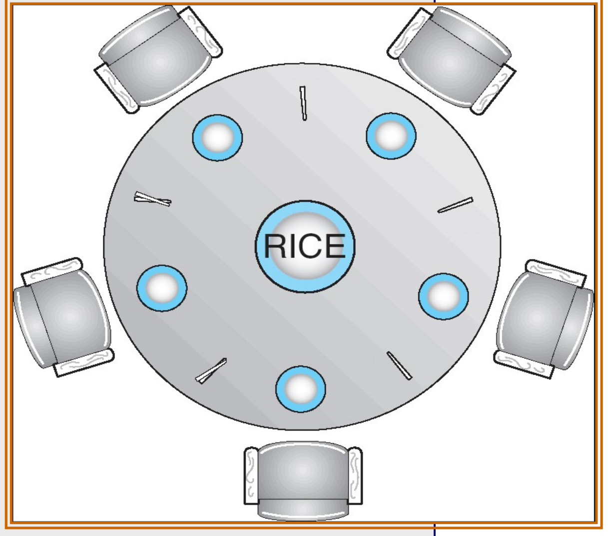


Dining-Philosophers Problem (Cont.)

- The structure of Philosopher *i*:

  - Semaphore chopstick [5] initialized to 1

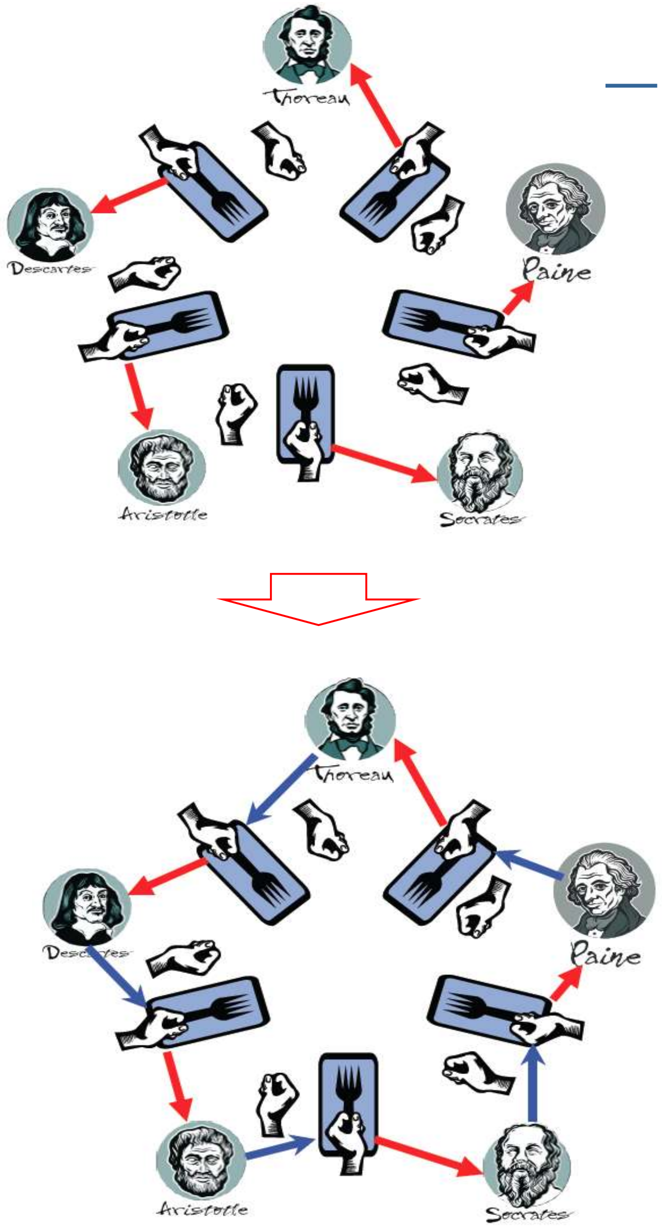
```
do
{
        wait ( chopstick [i] );
        wait ( chopstick [ (i+1)%5 ] );
        …
        //  eat
        …
        signal ( chopstick [i] );
        signal ( chopstick [ (i+1)%5 ] );
        …
        // think
        …
} while (TRUE);
```



Does this work ?

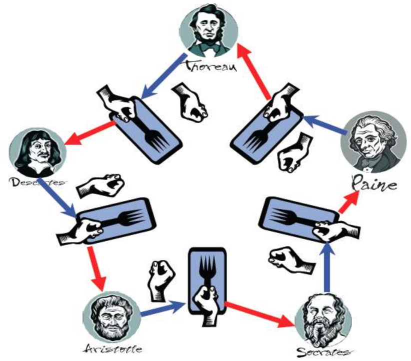
NO? What is the problem?

```
do
{
    wait ( chopstick [i] );
    wait ( chopstick [ (i+1)%5 ] );
    …
    //  eat
    …
    signal ( chopstick [i] );
    signal ( chopstick [ (i+1)%5 ] );
    …
    // think
    …
} while (TRUE);
```



Dining-Philosophers Problem (Cont.)

- Deadlock
  - All five philosophers become hungry at the same
  - Each grab her left chopstick – all chopstick semaphore will become 0
  - Each philosopher tries to grab her right chopstick, she will be delayed forever! – **deadlock**
- How can we solve this?
  - Deadlock Prevention
  - Deadlock Avoidance, ...

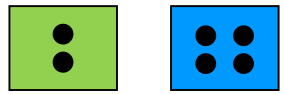


Chapter 7: Deadlocks

- Deadlock Concept
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
  - Deadlock Prevention
  - Deadlock Avoidance

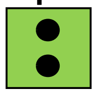
Deadlock: System Model

- System consists of ***finite*** number of **resource types**
  - **Physical resource types**: I/O devices (e.g., printers), memory space, CPU cycles
  - **Logical resource types**: semaphores, mutex locks, and files

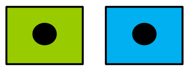
- Each resource type can have **identical instances** 
  - A system may have five identical printers: Printer resource type has five instances

Deadlock: System Model

- A Process requests an instance of a resource type
  - if instances are ***identical***, allocation of *any* instance of the type should satisfy the request
  - e.g., A system may have two printers
    - Case 1: two printers (e.g., instances) are defined to be in the same resource type - no one cares which printer is printing – so any printer instance can be allocated



    - Case 2: one printer is in 9th floor, and one printer is in basement – which printer would the 9th floor people would like to use? don't care? – This case two printer resource types.



- A process must **request an instance** of a **resource type** before using it
  - Any instance of a resource type should satisfy the request
- The process must **release** the instance of a resource type after using it

Deadlock: System Model

- Each process utilizes a resource as follows:
  - **request**
    - The process requests a resource
    - e.g., open(), malloc()   wait(), acquire(), …
  - **use**
    - The process operates on the resource
    - e.g., the process can print on the printer
    - e.g., the process can use critical section
  - **release**
    - The process releases the resource
    - close(), free()   signal(), release(), …

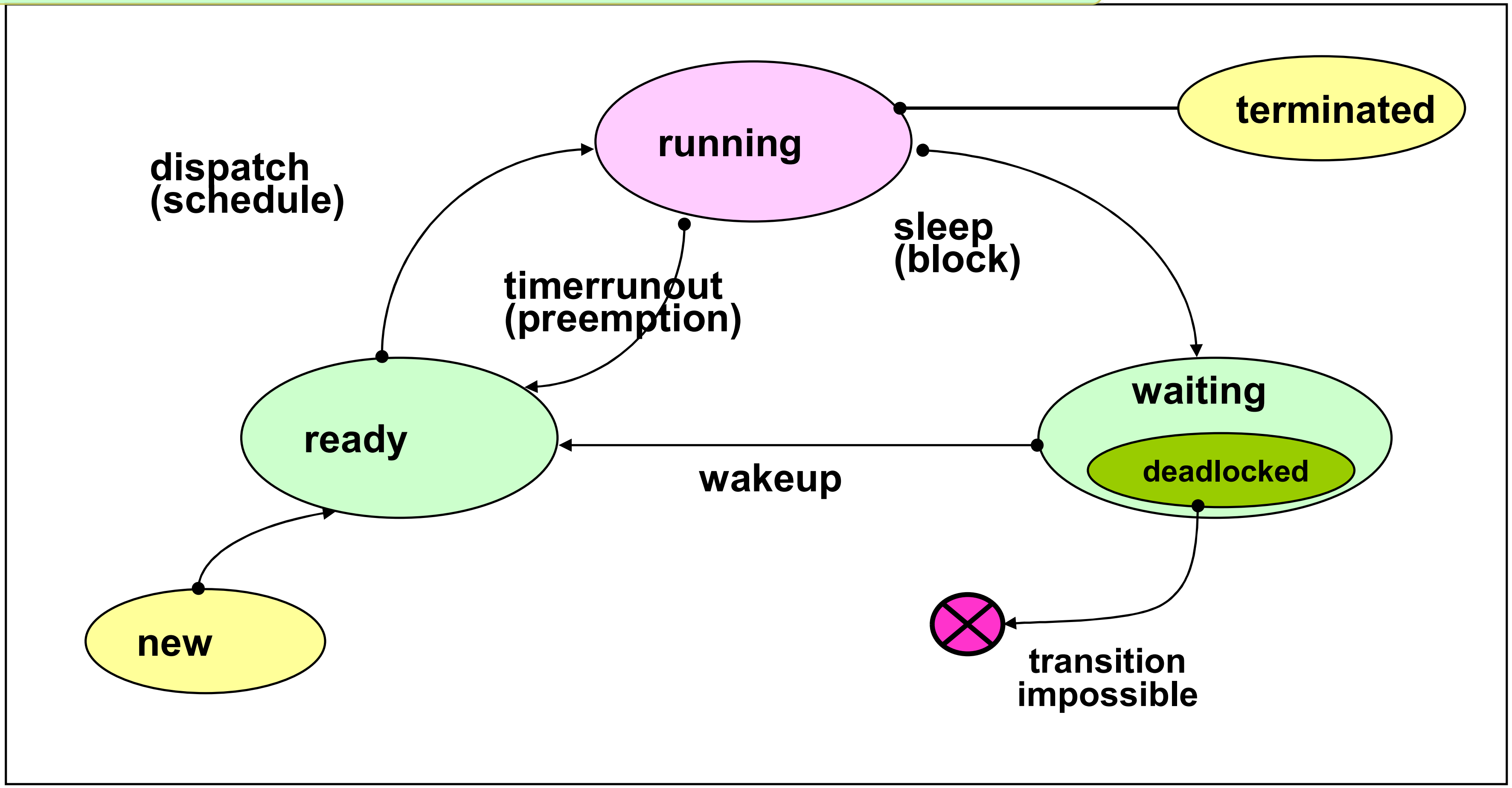
System Call!

Deadlock state

- In multiprogramming environment, several processes compete for shared resources
  - A process request resources; if resource is not available, process goes to _________ state.
  - Sometimes, waiting process can ***never*** again able to change state, because resource is ***held*** by other waiting process.
  - This situation is called **deadlock**.
- In which state does a process get deadlocked? - _______ state

Deadlock state

Process state transition diagram and deadlocked state



Deadlock state

- **Deadlock state**
  - The process is waiting for a particular event that will never occur in the future
  - The waiting process is never able to change state, because the resources that have been requested are held by other waiting processes, it is called deadlock

Definition : Deadlock

When a process waits for an event that will never occur
- That **process** is said to be in **deadlock state**
When there are one or more deadlocked process in the system
- The **system** is said to be in **deadlock state**

Chapter 7: Deadlocks

- Deadlock Concept
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
  - Deadlock Prevention
  - Deadlock Avoidance

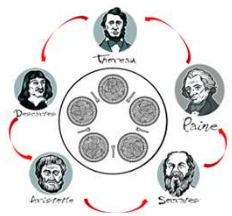
Deadlock Characterization

- **4 necessary conditions** for Deadlock

  = All 4 conditions ***must*** hold for deadlock to occur

1. **Mutual exclusion**
   - only one process at a time can use a resource instance
2. **Hold and wait**
   - a process holding at least one resource is waiting to acquire additional resources held by other processes
3. **No preemption**
   - a resource can be released only voluntarily by the process holding it, after that process has completed its task
4. **Circular wait**
   - $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots P_N \rightarrow P_0$

P0	*P1*
A0 wait (A);	B0 wait(B);
A1 wait (B);	B1 wait(A);



Exercise

- **4 necessary conditions** for Deadlock
  1. **Mutual exclusion**
     - only one process at a time can use a resource instance
  2. **Hold and wait**
     - a process holding at least one resource is waiting to acquire additional resources held by other processes
  3. **No preemption**
     - a resource can be released only voluntarily by the process holding it, after that process has completed its task
  4. **Circular wait**
     - $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots P_N \rightarrow P_0$

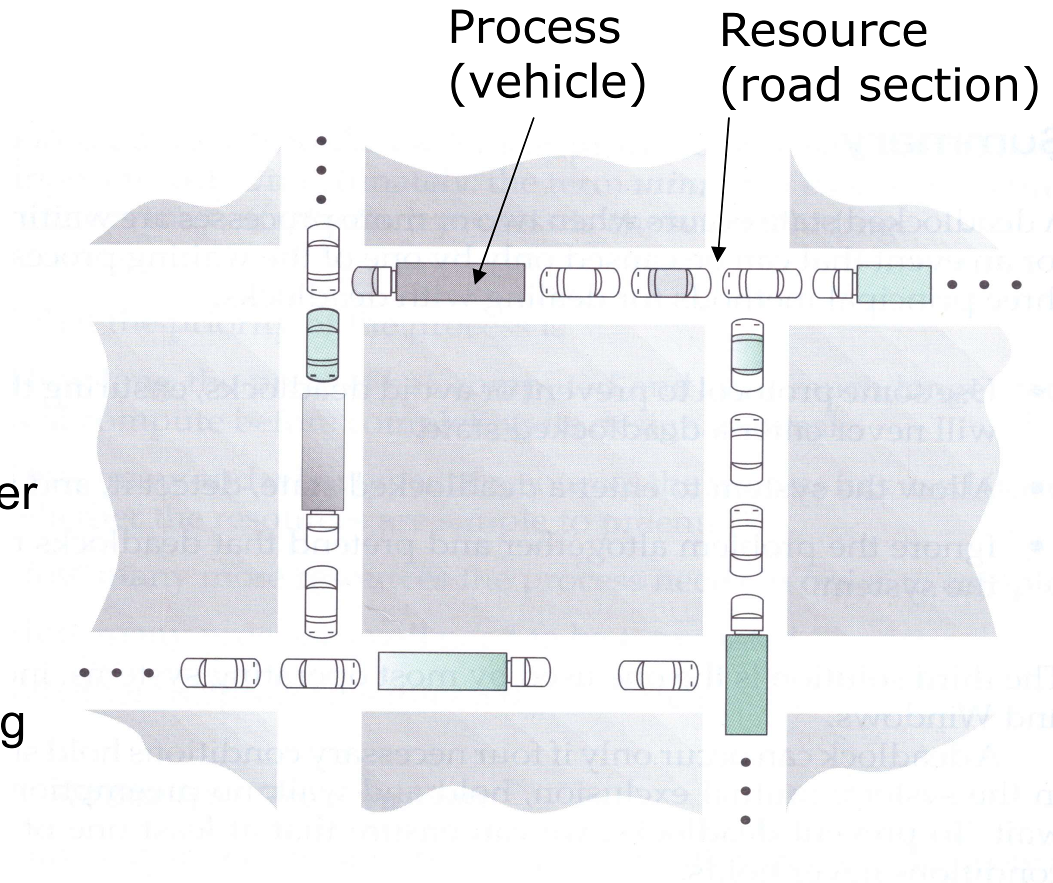
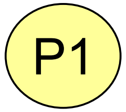


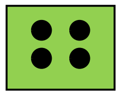
Figure 7.10 Traffic deadlock for Exercise 7.11.

Resource-Allocation Graph

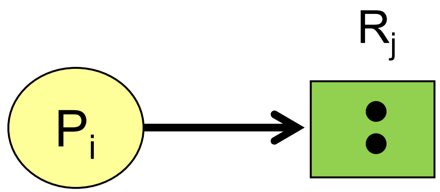
- Graph model
  - A directed graph: sets of nodes (vertices) *V* and edges *E*
  - Nodes *V*: Process node(P_i) ∪ resource node (R_j)
    - **Process** :



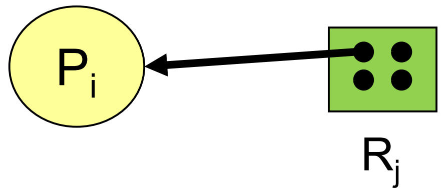
    - **Resource type** with 4 **instances** :



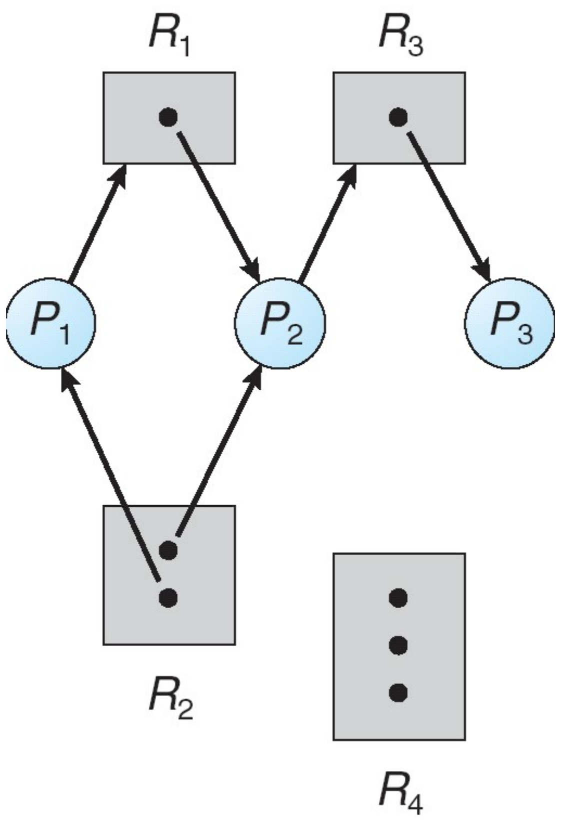
  - Edges *E*
    - ✓ **Request edge** ($P_i \rightarrow R_j$)
      - : The process P_i requested (is waiting)
      - for the resource R_j



    - ✓ **Assignment edge** ($R_j \rightarrow P_i$)
      - : P_i is holding an instance of R_j

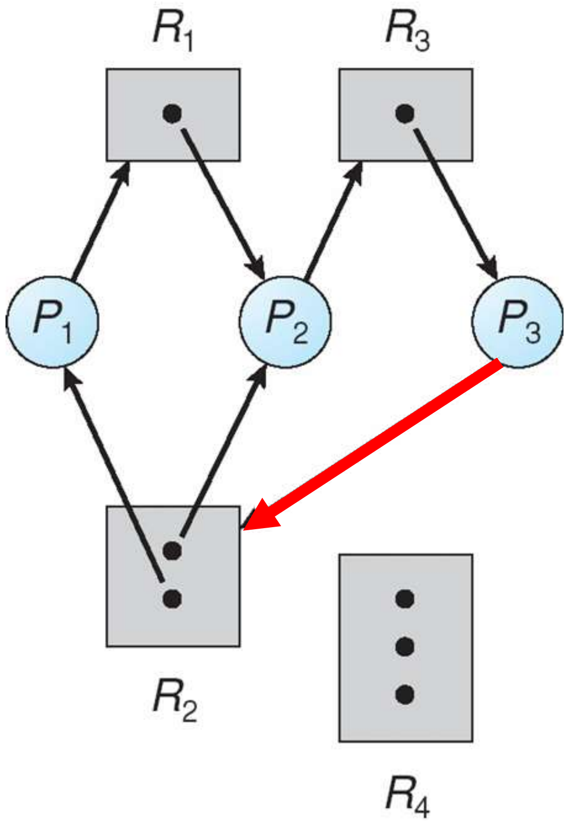


Example of a Resource Allocation Graph



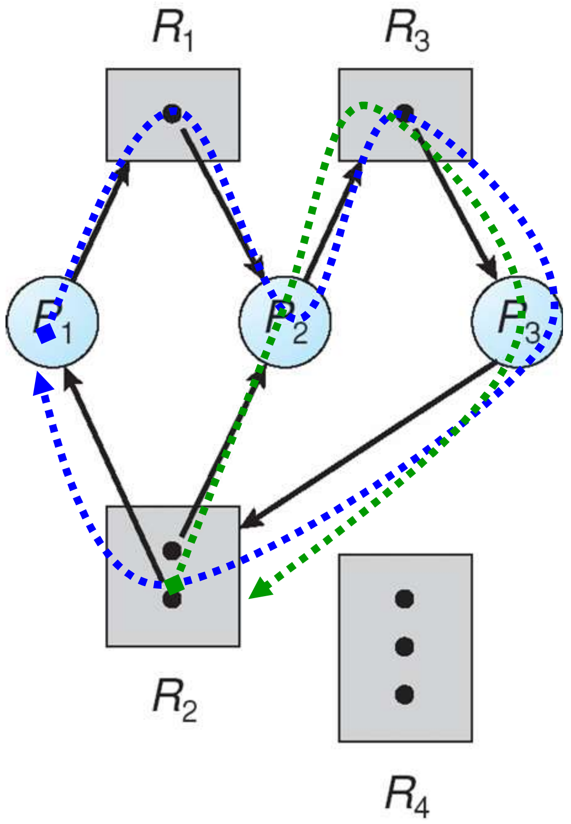
- Deadlock ?
  - Mutual exclusion
    - R_1, R_2, R_3, R_4
  - Hold and wait
    - P_1, P_2
  - No preemption
    - YES
  - Circular wait
    - **NO**

Resource Allocation Graph With A Deadlock



Cycles?

Resource Allocation Graph With A Deadlock

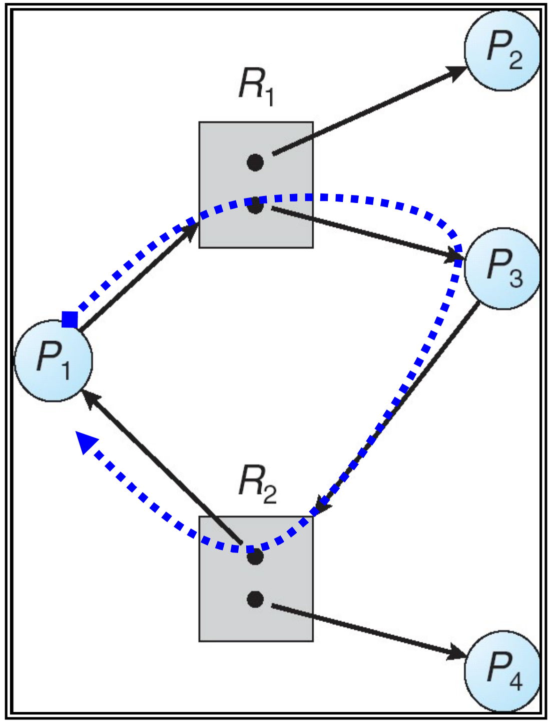


Two minimal cycles:

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

Graph With A Cycle But No Deadlock



minimal cycle:

$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

- Deadlock ?
  - Mutual exclusion
    - R_1, R_2, R_3, R_4
  - Hold and wait
    - P_1, P_3
  - No preemption
    - YES
  - Circular wait
    - YES

Cycle ≡ Deadlock?

- If graph contains **no** cycles $\Rightarrow$ **not** deadlock

- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, may or may not be deadlock