


# Contents

1. Introduction
2. System Structures
3. Process Concept
4. Multithreaded Programming
5. Process Scheduling
6. Synchronization
-  7. Deadlocks
8. Memory-Management Strategies
9. Virtual-Memory Management
10. File System
11. Mass-Storage Structures
12. I/O Systems
13. Protection, Security, Distributed Systems 57

# Chapter 7 Deadlocks

# Deadlocks

e.g. DiningPhilosopher 每個人都在等別人的筷子

- A set of process is in a *deadlock* state when every process in the set is waiting for an event that can be caused by only another process in the set.
- A System Model
  - Competing processes – distributed?
  - Resources:
    - Physical Resources, e.g., CPU, printers, memory, etc.
    - Logical Resources, e.g., files, *semaphores*, etc.

e.g. Two processes each wait for the semaphore of each other.

# Deadlocks

- A Normal Sequence
  - Request: Granted or Rejected
  - Use e.g. block()
  - Release
- Remarks
  - No request should exceed the system capacity!
  - Deadlock can involve different resource types!
    - Several instances of the same type!

# Deadlocks

```
void *do_work_one(void *param) {  
1.  pthread_mutex_lock(&first_mutex);  
3.  pthread_mutex_lock(&second_mutex);  
    /* Do some work */  
    pthread_mutex_unlock(&second_mutex);  
    pthread_mutex_unlock(&first_mutex);  
    pthread_exit(0); }  
  
void *do_work_two(void *param) {  
2.  pthread_mutex_lock(&second_mutex);  
    pthread_mutex_lock(&first_mutex);  
    /* Do some work */  
    pthread_mutex_unlock(&first_mutex);  
    pthread_mutex_unlock(&second_mutex);  
    pthread_exit(0); }  
  
...  
pthread_mutex_init(&first_mutex, NULL);  
pthread_mutex_init(&second_mutex, NULL);
```

當 P2 打算鎖住  
first\_mutex 時，發現  
已經被 P1 鎖住，所以  
切回 P1，P1 又發現  
second\_mutex 已經被  
P2 鎖住，於是產生  
deadlock。

# Deadlock Characterization

- Necessary Conditions

(deadlock  $\rightarrow$  conditions or  $\neg$  conditions  $\rightarrow \neg$  deadlock)

- Mutual Exclusion – At least one resource must be held in a non-sharable mode! 若 resource is sharable, 不可能會有 mutual exclusion
- Hold and Wait –  $P_i$  is holding at least one resource and waiting to acquire additional resources that are currently held by other processes!

# Deadlock Characterization

- **No Preemption** – Resources are nonpreemptible!
- **Circular Wait** – There exists a set  $\{P_0, P_1, \dots, P_n\}$  of waiting process such that  $P_0 \xrightarrow{\text{wait}} P_1, P_1 \xrightarrow{\text{wait}} P_2, \dots, P_{n-1} \xrightarrow{\text{wait}} P_n$ , and  $P_n \xrightarrow{\text{wait}} P_0$ .

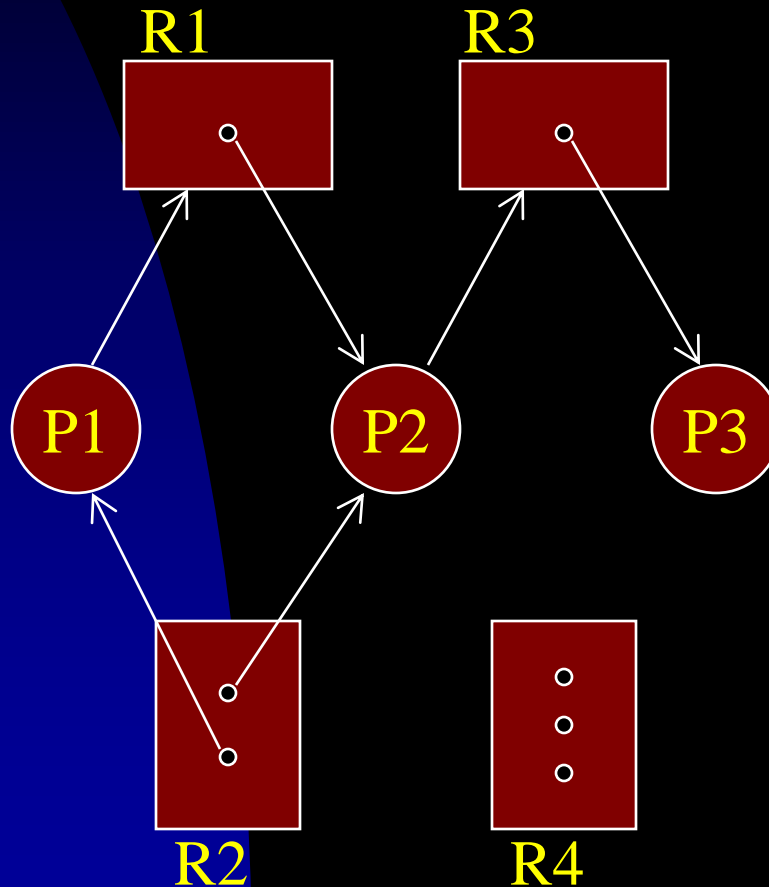
if  $A \rightarrow B$ , 這 4 個 condition 只要有一個 fail, 就不會有 deadlock!

- Remark:
  - Condition 4 implies Condition 2.
  - The four conditions are not completely independent!

他們之間存在關係！

# Resource Allocation Graph

## System Resource-Allocation Graph



### Vertices

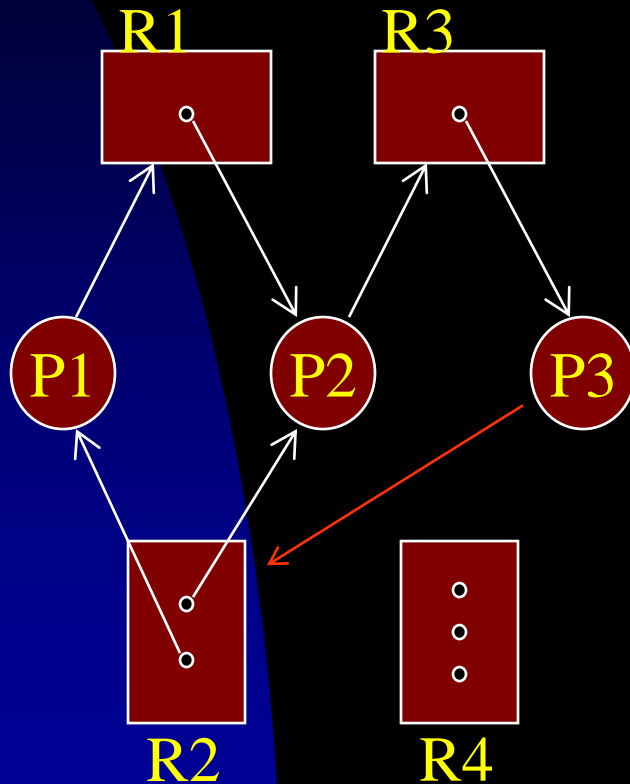
- Processes:  
 $\{P1, \dots, Pn\}$
- Resource Type :  
 $\{R1, \dots, Rm\}$

### Edges

- Request Edge:  
 $P_i \rightarrow R_j$
- Assignment Edge:  
 $R_i \rightarrow P_j$



# Resource Allocation Graph



## ■ Example

### ■ No-Deadlock

#### ■ Vertices

■  $P = \{ P1, P2, P3 \}$

■  $R = \{ R1, R2, R3, R4 \}$

#### ■ Edges

■  $E = \{ P1 \rightarrow R1, P2 \rightarrow R3, R1 \rightarrow P2, R2 \rightarrow P2, R2 \rightarrow P1, R3 \rightarrow P3 \}$

### ■ Resources

■  $R1:1, R2:2, R3:1, R4:3$

■ → results in a deadlock.

# Resource Allocation Graph

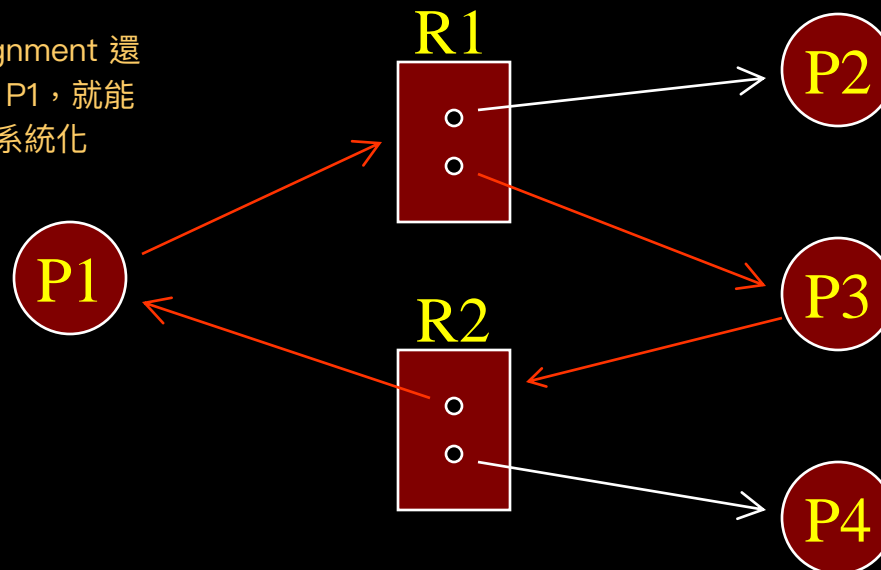
如果給定一個 RAG，若每個 resource 只有一個，一但有 loop，就有 deadlock！

## ■ Observation

### ■ The existence of a cycle

- One Instance per Resource Type → Yes!!
- Otherwise → Only A Necessary Condition!!

在此例中，若將 R1→P2 的 assignment 還回來，而將 P1 → R1 改成 R1 → P1，就能解開 deadlock，但這樣討論不夠系統化



# Methods for Handling Deadlocks

- Solutions:
  - Make sure that the system never enters a deadlock state!
    - Deadlock Prevention: Fail at least one of the necessary conditions  
if  $A \rightarrow B$ , 這 4 個 condition 只要有一個 fail, 就不會有 deadlock!
    - Deadlock Avoidance: Processes provide information regarding their resource usage. Make sure that the system always stays at a “safe” state!

# Methods for Handling Deadlocks

- Do **recovery** if the system is deadlocked.
  - Deadlock Detection
  - Recovery
- Ignore the possibility of deadlock occurrences!
  - Restart the system “manually” if the system “seems” to be deadlocked or stops functioning.
  - Note that the system may be “frozen” temporarily!

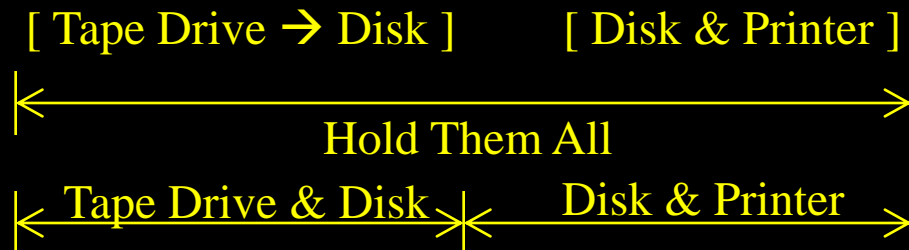
# Deadlock Prevention

- Observation:
  - Try to fail anyone of the necessary condition!  
$$\therefore \neg (\wedge i\text{-th condition}) \rightarrow \neg \text{deadlock}$$
- Mutual Exclusion
  - ?? Some resources, such as a printer, are intrinsically non-sharable??

# Deadlock Prevention

- Hold and Wait
  - Acquire all needed resources before its execution. 拿東西，又想要別人的東西
  - Release allocated resources before request additional resources!

1. TD + Disk  $\rightarrow$  Disk + Printer
2. TD + Disk + Printer  
(全部 hold 在手上，不要去 wait)



當在使用 TD & Disk 時，  
Printer 沒事；同理，當在使用  
Disk & Printer 時，TD 沒事

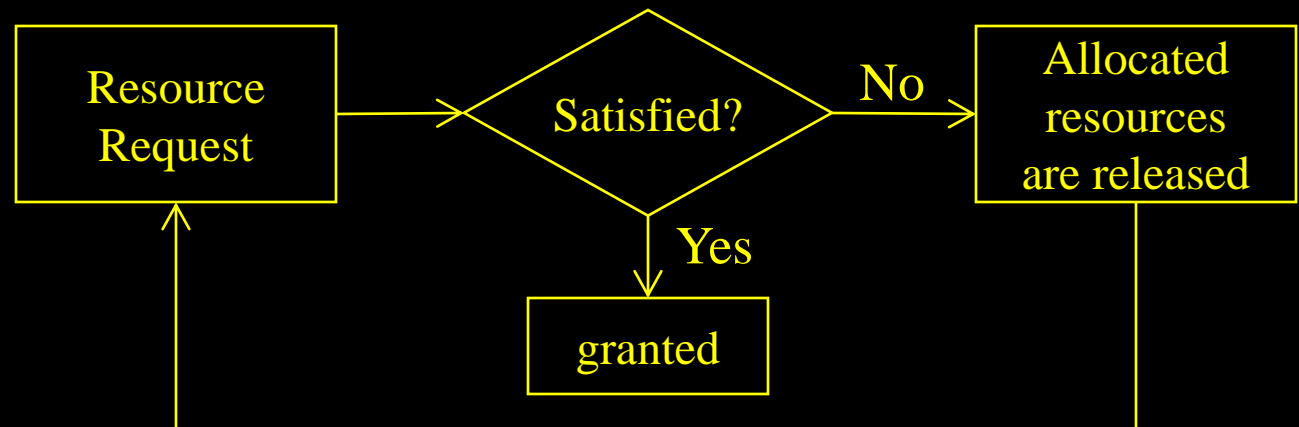
- Disadvantage:
  - Low Resource Utilization
  - Starvation

# Deadlock Prevention

- No Preemption

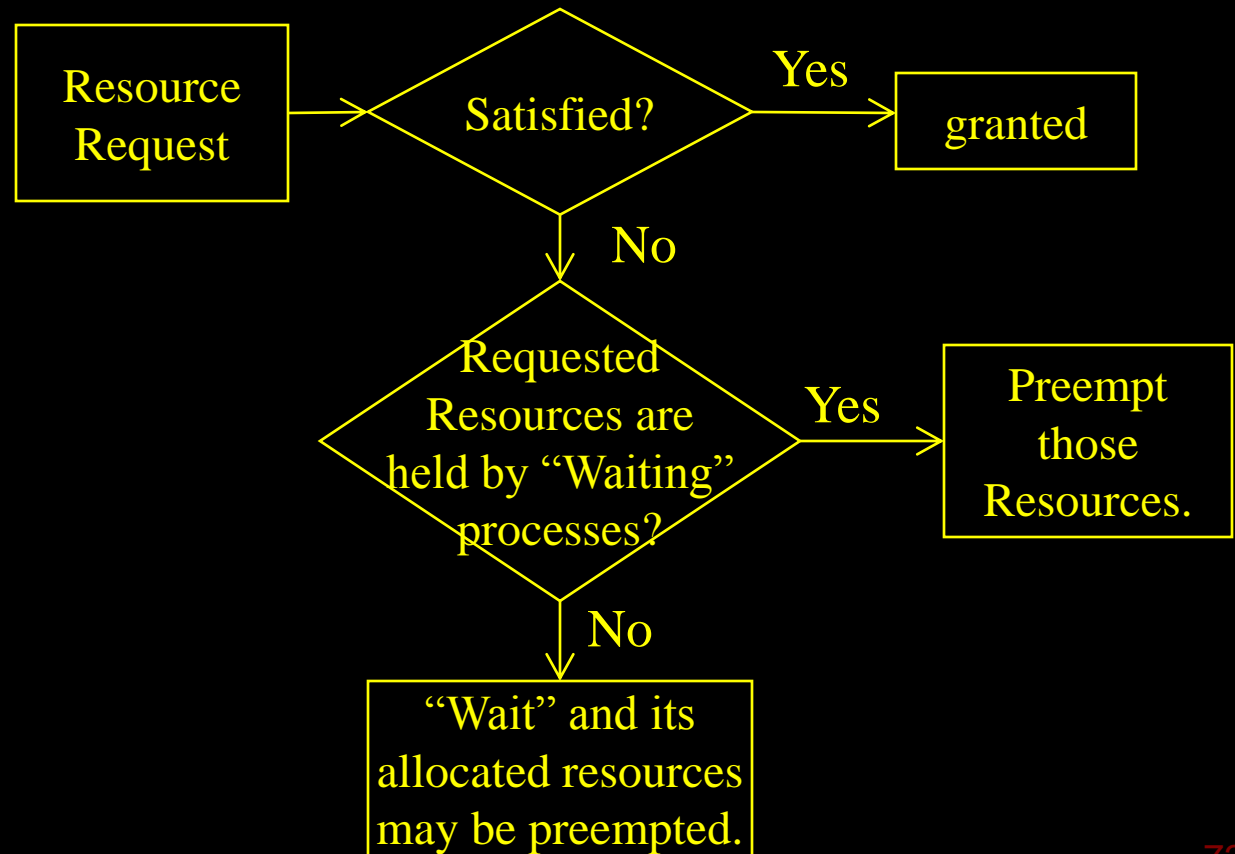
- Resource preemption causes the release of resources.
- Related protocols are only applied to resources whose states can be saved and restored, e.g., CPU register & memory space, instead of printers or tape drives.

- Approach 1: e.g. printer 印到一半，還有 100 頁：先等我把這頁印完，印完再給你印。



# Deadlock Prevention

- Approach 2





# Deadlock Prevention

- Circular Wait

A resource-ordering approach:

$$\left\{ \begin{array}{l} F : R \rightarrow N \\ \text{Resource requests must be made in} \\ \text{an increasing order of enumeration.} \end{array} \right.$$

- Type 1 – strictly increasing order of resource requests.
  - Initially, order any # of instances of  $R_i$
  - Following requests of any # of instances of  $R_j$  must satisfy  $F(R_j) > F(R_i)$ , and so on.
  - \* A single request must be issued for all needed instances of the same resources.

# Deadlock Prevention

- Type 2
  - Processes must release all  $R_i$ 's when they request any instance of  $R_j$  if  $F(R_i) \geq F(R_j)$
- $F : R \rightarrow N$  must be defined according to the normal order of resource usages in a system, e.g.,

$$\left. \begin{array}{l} F(\text{tape drive}) = 1 \\ F(\text{disk drive}) = 5 \\ F(\text{printer}) = 12 \end{array} \right\} \text{?? feasible ??}$$

# Deadlock Avoidance

- Motivation:
  - Deadlock-prevention algorithms can cause low device utilization and reduce system throughput!
- ➔ Acquire additional information about how resources are to be requested and have better resource allocation!
  - Processes declare their maximum number of resources of each type that it may need.

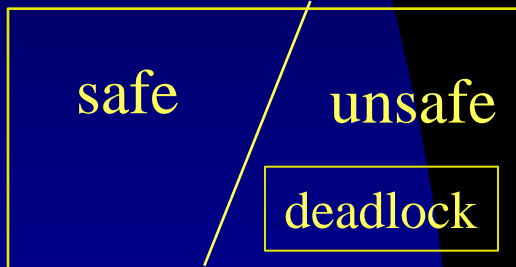
# Deadlock Avoidance

- A Simple Model
  - A resource-allocation state  
<# of available resources,  
# of allocated resources,  
max demands of processes>
- A deadlock-avoidance algorithm dynamically examines the resource-allocation state and make sure that it is safe.
  - e.g., the system never satisfies the circular-wait condition.

# Deadlock Avoidance

- Safe Sequence
  - A sequence of processes  $\langle P_1, P_2, \dots, P_n \rangle$  is a safe sequence if

$$\forall P_i, need(P_i) \leq Available + \sum_{j < i} Allocated(P_j)$$



Deadlocks are avoided if the system can allocate resources to each process up to its maximum request in some order. If so, the system is in a safe state!

- Safe State
  - The existence of a safe sequence
- Unsafe need(P<sub>i</sub>) 是 maximum 的 need，所以不代表若 >，就一定有 deadlock

# Deadlock Avoidance

## ■ Example:

	Max Needs	Allocated	Available
P0	10	5	3
P1	4	2	
P2	9	2	

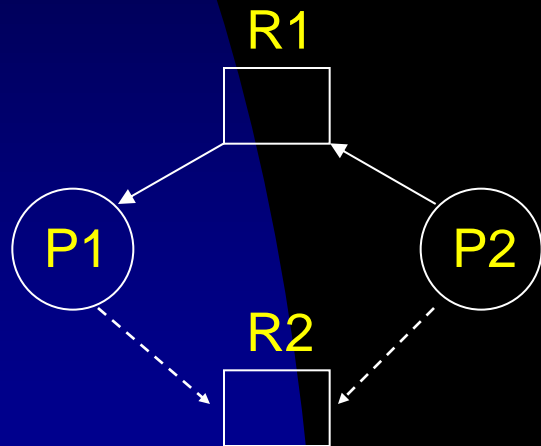
- The existence of a safe sequence  $\langle P1, P0, P2 \rangle$ .
- If P2 got one more, the system state is unsafe.  
(P0, 5), (P1, 2), (P2, 3), (available, 2))

How to ensure that the system will always remain in a safe state?

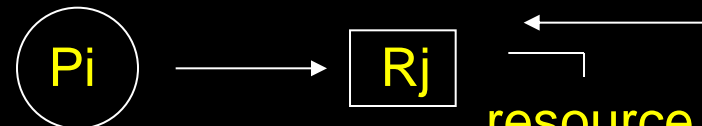
不 safe 還不一定有 deadlock，但代表不安全

# Deadlock Avoidance – Resource-Allocation Graph Algorithm

- One Instance per Resource Type



•Request Edge



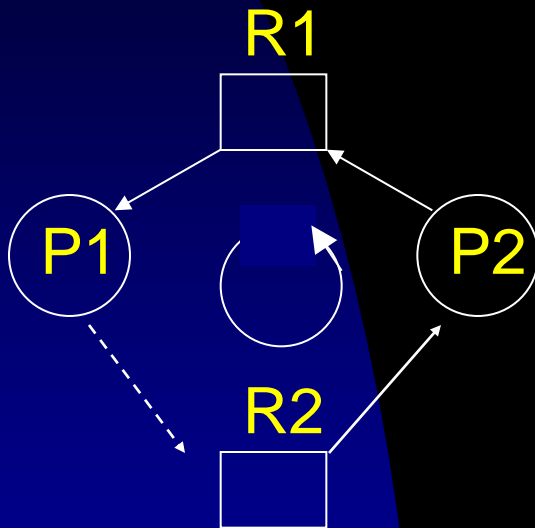
•Assignment Edge



•Claim Edge



# Deadlock Avoidance – Resource-Allocation Graph Algorithm



A cycle is detected!

➔ The system state is unsafe!

- R2 was requested & granted!

如果有一天，P1 說：我要 request R2，這時我們剛才把 R2 assign 給 P2 就可能產生 deadlock，所以這是 unsafe 的！

Safe state: no cycle

Unsafe state: otherwise

Cycle detection  
can be done  
in  $O(n^2)$



# Deadlock Avoidance – Banker's Algorithm

*n*: # of processes,  
*m*: # of resource types

- Available [*m*]
  - If Available [*i*] = *k*, there are *k* instances of resource type *R<sub>i</sub>* available.
- Max [*n*,*m*]
  - If Max [*i*,*j*] = *k*, process *P<sub>i</sub>* may request at most *k* instances of resource type *R<sub>j</sub>*.
- Allocation [*n*,*m*]
  - If Allocation [*i*,*j*] = *k*, process *P<sub>i</sub>* is currently allocated *k* instances of resource type *R<sub>j</sub>*.
- Need [*n*,*m*]
  - If Need [*i*,*j*] = *k*, process *P<sub>i</sub>* may need *k* more instances of resource type *R<sub>j</sub>*.



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

# Deadlock Avoidance – Banker's Algorithm

$n$ : # of processes,  
 $m$ : # of resource types

- Safety Algorithm – A state is safe??
  1.  $Work := Available$  &  $Finish[i] := F, 1 \leq i \leq n$
  2. Find an  $i$  such that both
    1.  $Finish[i] = F$
    2.  $Need[i] \leq Work$**If no such  $i$  exist, then goto Step4**
  3.  $Work := Work + Allocation[i]$   
 $Finish[i] := T$ ; **Goto Step2**
  4. **If  $Finish[i] = T$  for all  $i$ , then the system is in a safe state.**

Where  $Allocation[i]$  and  $Need[i]$  are the  $i$ -th row of  $Allocation$  and  $Need$ , respectively, and  
 $X \leq Y$  if  $X[j] \leq Y[j]$  for all  $j$ ,  
 $X < Y$  if  $X \leq Y$  and  $Y \neq X$

# Deadlock Avoidance – Banker's Algorithm

- Resource-Request Algorithm

Request<sub>i</sub> [j] = k: P<sub>i</sub> requests k instance of resource type R<sub>j</sub>

1. If Request<sub>i</sub> ≤ Need<sub>i</sub>, then Goto Step2; otherwise, Trap
2. If Request<sub>i</sub> ≤ Available, then Goto Step3; otherwise, P<sub>i</sub> must wait.
3. Have the system pretend to have allocated resources to process P<sub>i</sub> by setting

Available := Available – Request<sub>i</sub>;

Allocation<sub>i</sub> := Allocation<sub>i</sub> + Request<sub>i</sub>;

Need<sub>i</sub> := Need<sub>i</sub> – Request<sub>i</sub>;

Execute “Safety Algorithm”. If the system state is safe, the request is granted; otherwise, P<sub>i</sub> must wait, and the old resource-allocation state is restored!

# Deadlock Avoidance

## ■ An Example

	Allocation			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	3	3	2
P1	2	0	0	3	2	2	1	2	2			
P2	3	0	2	9	0	2	6	0	0			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

- A safe state

∴  $\langle P1, P3, P4, P2, P0 \rangle$  is a safe sequence.

# Deadlock Avoidance

Let P1 make a request  $\text{Request}_i = (1,0,2)$   
 $\text{Request}_i \leq \text{Available}$  (i.e.,  $(1,0,2) \leq (3,3,2)$ )

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3	2	3	0
P1	3	0	2	0	2	0			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

→ Safe  $\because \langle P1, P3, P4, P0, P2 \rangle$  is a safe sequence!

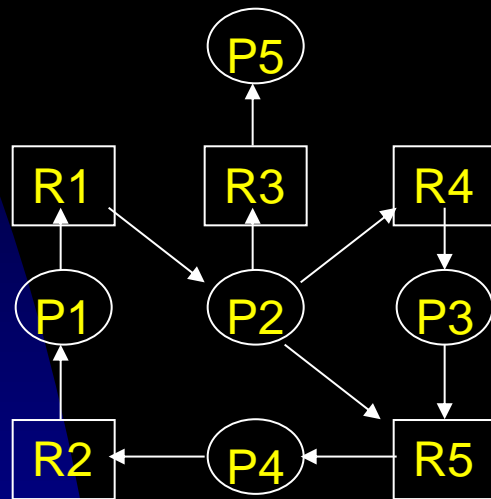
$(3, 3, 0) > (2, 3, 0)$

- If  $\text{Request}_4 = (3,3,0)$  is asked later, it must be rejected.
- $\text{Request}_0 = (0,2,0)$  must be rejected because it results in an unsafe state.  $(2, 3, 0) - (0, 2, 0) = (2, 1, 0) < \text{其它所有人's need}$

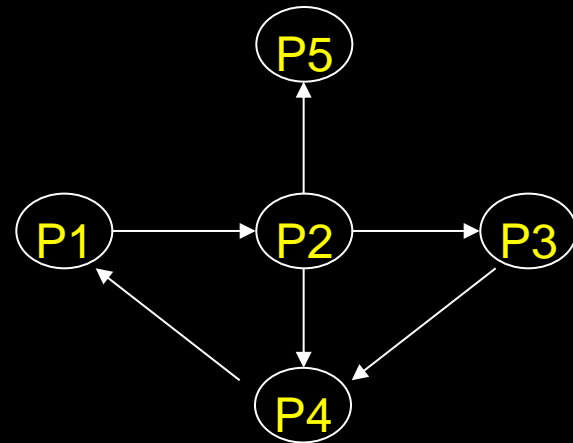
# Deadlock Detection

- Motivation:
  - Have high resource utilization and “maybe” a lower possibility of deadlock occurrence.
- Overheads:
  - Cost of information maintenance
  - Cost in executing a detection algorithm
  - Potential loss inherent from a deadlock recovery

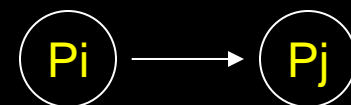
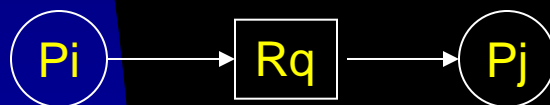
# Deadlock Detection – Single Instance per Resource Type



A Resource-Allocation Graph



A Wait-For Graph



- Detect an cycle in  $O(n^2)$ .
- The system needs to maintain the wait-for graph

# Deadlock Detection – Multiple Instance per Resource Type

$n$ : # of processes,  
 $m$ : # of resource types

- Data Structures
  - Available[1.. $m$ ]: # of available resource instances
  - Allocation[1.. $n$ , 1.. $m$ ]: current resource allocation to each process
  - Request[1.. $n$ , 1.. $m$ ]: the current request of each process
    - If Request[ $i$ ,  $j$ ] =  $k$ ,  $P_i$  requests  $k$  more instances of resource type  $R_j$



# Deadlock Detection – Multiple Instance per Resource Type

1.  $Work := Available$ . For  $i = 1, 2, \dots, n$ , **if**  $Allocation[i] \neq 0$ , **then**  $Finish[i] = F$ ; **otherwise**,  $Finish[i] = T$ .
2. Find an  $i$  such that both
  - a.  $Finish[i] = F$
  - b.  $Request[i] \leq Work$**If** no such  $i$ , **Goto** Step 4
3.  $Work := Work + Allocation[i]$   
 $Finish[i] := T$   
**Goto** Step 2
4. **If**  $Finish[i] = F$  for some  $i$ , **then** the system is in a deadlock state. **If**  $Finish[i] = F$ , then process  $P_i$  is deadlocked.

Complexity =  
 $O(m * n^2)$

# Deadlock Detection – Multiple Instances per Resource Type

## ■ An Example

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	2	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	2			

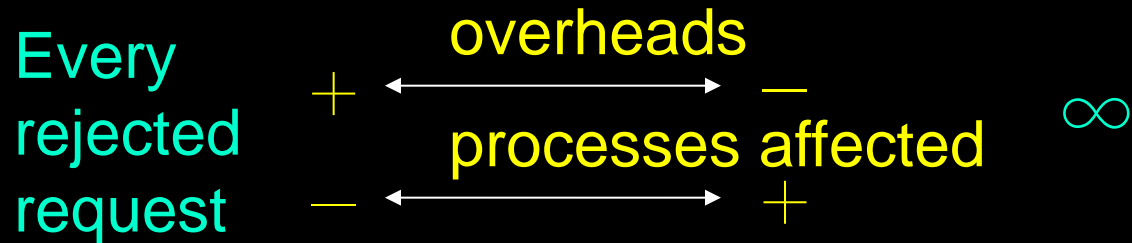
→ Find a sequence  $\langle P0, P2, P3, P1, P4 \rangle$  such that  $Finish[i] = T$  for all  $i$ .

If Request<sub>2</sub> = (0,0,1) is issued, then P1, P2, P3, and P4 are **deadlocked**.

P0: Available = (0, 3, 0) < 其它所有人's Request

# Deadlock Detection – Algorithm Usage

- When should we invoke the detection algorithm?
  - How often is a deadlock likely to occur?
  - How many processes will be affected by a deadlock?



- Time for Deadlock Detection?
  - CPU Threshold? Detection Frequency? ...

# Deadlock Recovery

- Whose responsibility to deal with deadlocks?
  - Operator deals with the deadlock manually.
  - The system recover from the deadlock automatically.
- Possible Solutions
  - Abort one or more processes to break the circular wait.
  - Preempt some resources from one or more deadlocked processes.

# Deadlock Recovery – Process Termination

- Process Termination
  - Abort all deadlocked processes!
    - Simple but costly!
  - Abort one process at a time until the deadlock cycle is broken!
    - Overheads for running the detection again and again.
    - The difficulty in selecting a victim!

But, can we abort any process?  
Should we compensate any  
damage caused by aborting?

# Deadlock Recovery – Process Termination

- What should be considered in choosing a victim?
  - Process priority
  - The CPU time consumed and to be consumed by a process.
  - The numbers and types of resources used and needed by a process
  - Process's characteristics such as “interactive or batch”
  - The number of processes needed to be aborted.

# Deadlock Recovery – Resource Preemption

- Goal: Preempt some resources from processes and give them to other processes until the deadlock cycle is broken!
- Issues
  - Selecting a victim:
    - It must be cost-effective!
  - Roll-Back
    - How far should we roll back a process whose resources were preempted?
  - Starvation
    - Will we keep picking up the same process as a victim?
    - How to control the # of rollbacks per process efficiently?

# Deadlock Recovery – Combined Approaches

- Partition resources into classes that are hierarchically ordered.  
⇒ No deadlock involves more than one class
- Handle deadlocks in each class independently



# Deadlock Recovery – Combined Approaches

Examples:

- Internal Resources: Resources used by the system, e.g., PCB
  - Prevention through resource ordering
- Central Memory: User Memory
  - Prevention through resource preemption
- Job Resources: Assignable devices and files
  - Avoidance ∴ This info may be obtained!
- Swappable Space: Space for each user process on the backing store
  - Pre-allocation ∴ the maximum need is known!