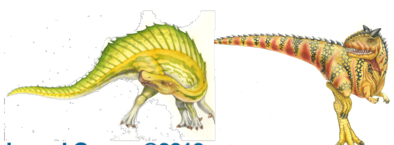
Introduction to Operating System

Deadlocks

葉奕成 I-Cheng (Garrett) Yeh

2020/6, V5 for 10th ed.



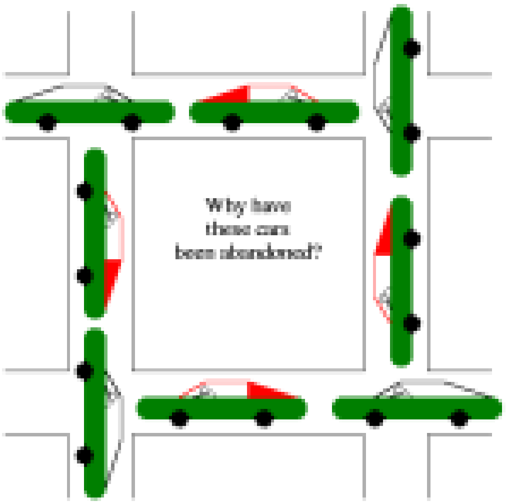
content

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
- Summary

SYSTEM MODEL

Deadlock

- Deadlock
  - A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Example
  - P_1 and P_2 each hold one disk drive and each needs another one



Deadlock vs Starvation

Deadlock	Starvation
一組processes形成circular waiting，導致processes無法往下執行	∵長期無法取得完成工作所需資源某(些)processes形成infinite Blocking
不允許資源preemptive	易發生在不公平/preemptive的環境
CPU utilization及Throughput會大幅下降	CPU utilization正常 與此無關聯

相似點：皆為資源分配及協調出了問題

Deadlock Example with Lock Ordering - Case I

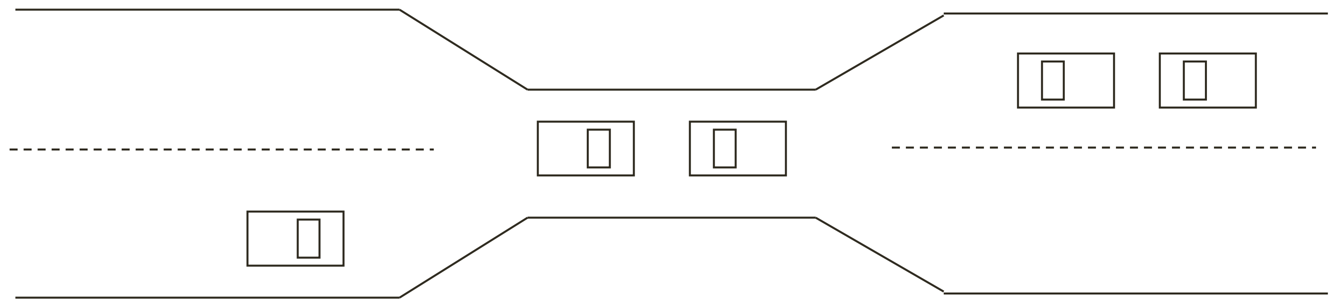
```
/* thread one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);
   pthread_exit(0);
}
/* thread two runs in this function */
void *do_work_two(void *param)
{
   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);
}
```

Deadlock Example with Lock Ordering - Case I I

```
void transaction(Account from, Account to, double amount)
{
   mutex lock1, lock2;
   lock1 = get_lock(from);
   lock2 = get_lock(to);
   acquire(lock1);
      acquire(lock2);
         withdraw(from, amount);
         deposit(to, amount);
      release(lock2);
   release(lock1);
}
```

Transactions 1 and 2 execute concurrently.
Transaction 1 transfers $25 from account A to account B, and
Transaction 2 transfers $50 from account B to account A.

Deadlock



- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible
- Note – Most OSes do not prevent or deal with deadlocks

System Model

- Resources:
  - Physical Resources
    - e.g., CPU, printers, memory, etc.
  - Logical Resources
    - e.g., files, semaphores, etc.

- A Normal Sequence
  - Request
  - Use
  - Release

DEADLOCK CHARACTERIZATION

Deadlock Characterization

deadlock -> conditions
~ conditions -> ~deadlock

- Necessary Conditions
  - **Mutual exclusion:** At least one resource must be held in a non-sharable mode
  - **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
  - **No preemption:** Resources are non-preemptible
  - **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_0.

必要條件當我們說甲是乙的必要條件(necessary condition)時，意思是說沒有甲，乙便不可能存在。

Deadlock Characterization

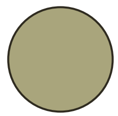
- Remark:
  - Condition 4 implies Condition 2.
  - The four conditions are not completely independent

Resource Allocation Graph

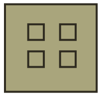
- System Resource-Allocation Graph
  - A set of vertices *V* and a set of edges *E*
  - V is partitioned into two types:
    - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
    - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
  - request edge
    - directed edge P_i >> R_j
  - assignment edge
    - directed edge R_j >> P_i

Resource-Allocation Graph (Cont.)

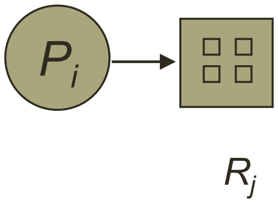
- Process



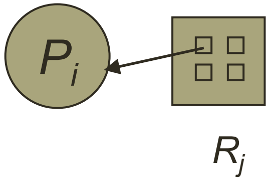
- Resource Type with 4 instances



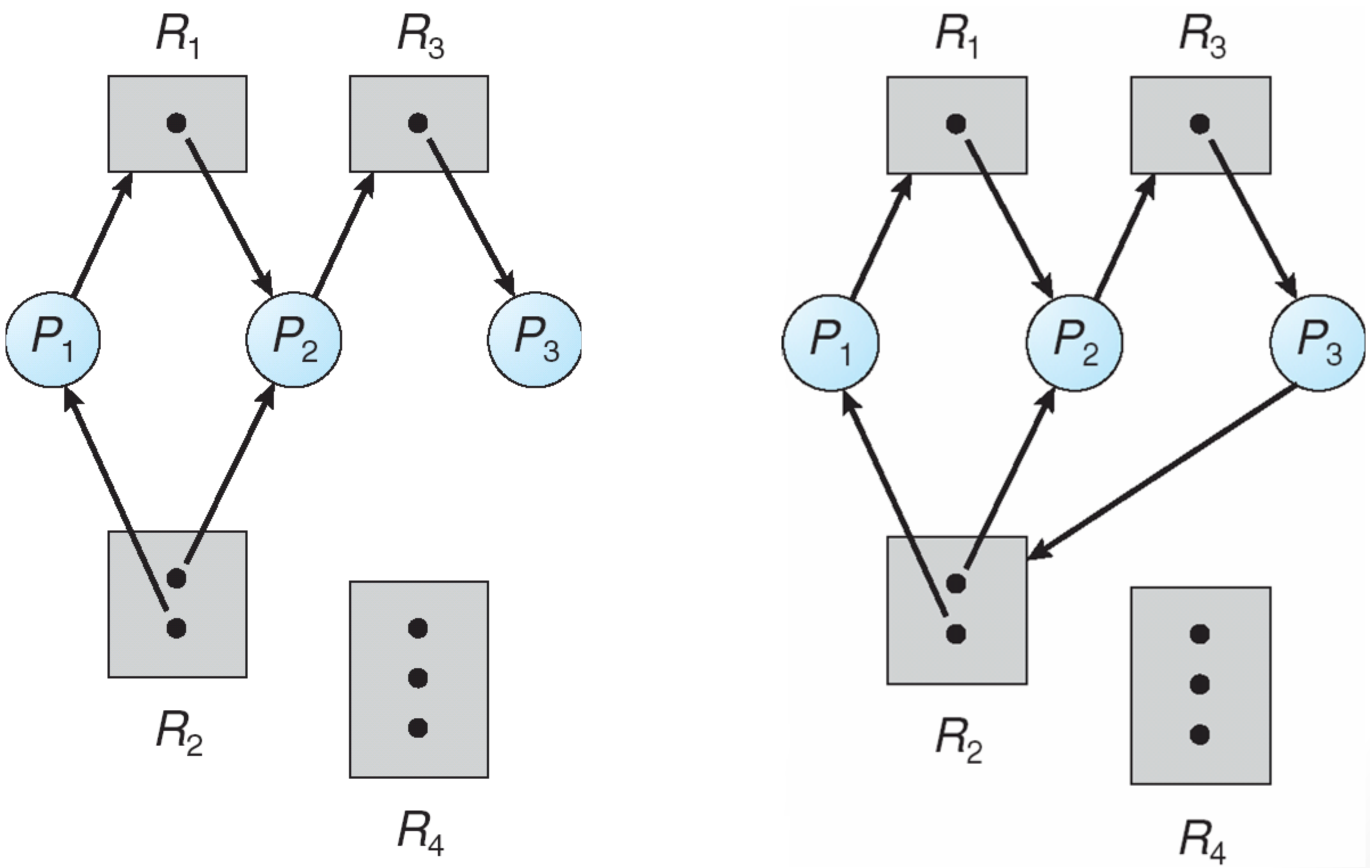
- P_i requests instance of R_j



- P_i is holding an instance of R_j

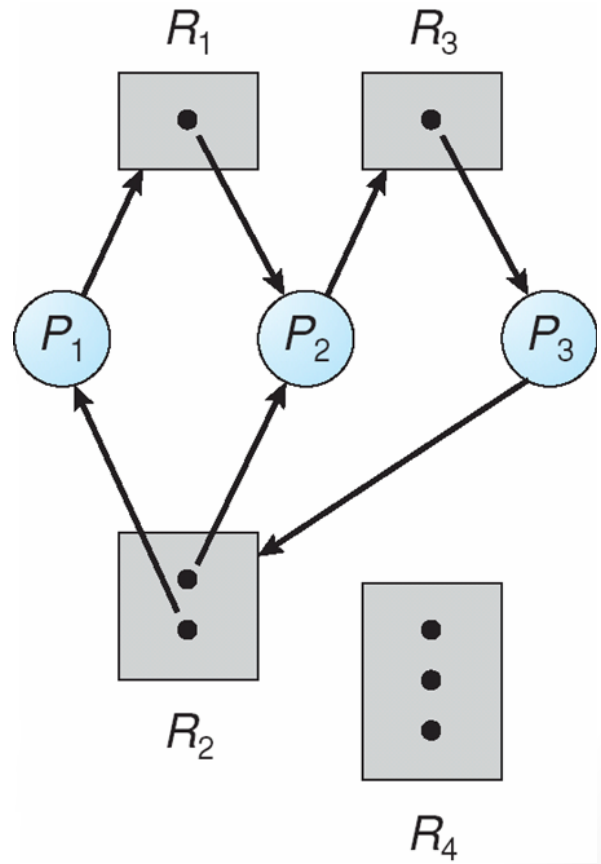


Example of a Resource Allocation Graph



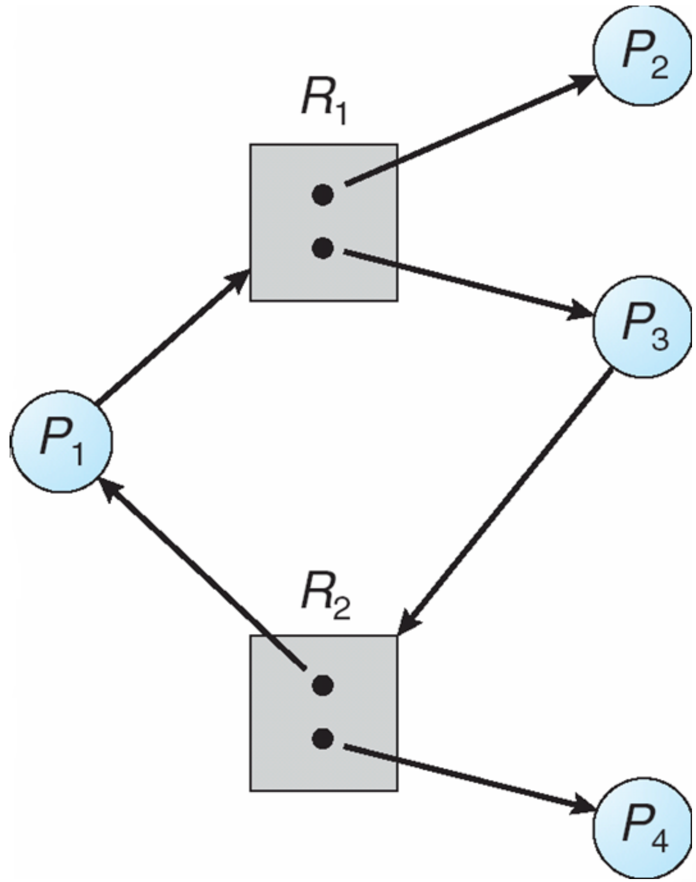
Resource Allocation Graph With A Deadlock

- Mutual exclusion
- Hold and wait
  - R2 -> P1 -> R1
  - R1 -> P2 -> R3
- No preemption
- Circular wait
  - P1 -> R1 -> P2 -> R3 -> P3 ->R2 -> P1 (P2)



Graph With A Cycle But No Deadlock

- Mutual exclusion
- Hold and wait
  - R2 -> P1 -> R1
  - R1 -> P3 -> R2
- No preemption
- Circular wait
  - After P2/P4 finish, it would be fixed.



Basic Facts

- The existence of a cycle
  - One Instance per Resource Type
    - Deadlock
  - Otherwise
    - Has possibility of deadlock
- No cycles
  - no deadlock

METHODS FOR HANDLING DEADLOCKS

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state
  - Deadlock Prevention (ch 8.5) – static
    - Fail at least one of the necessary conditions
  - Deadlock Avoidance (ch 8.6) – dynamic / on-the-fly
    - Processes provide information regarding their resource usage. Make sure that the system always stays at a “safe” state!

Methods for Handling Deadlocks

- Allow the system to enter a deadlock state and then recover
  - Deadlock Detection (ch 8.7)
  - Recovery (ch 8.8)

- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX
  - Restart the system "manually" if the system "seems" to be deadlocked or stops functioning.
    - Note that the system may be "frozen" temporarily

DEADLOCK PREVENTION

Deadlock Prevention

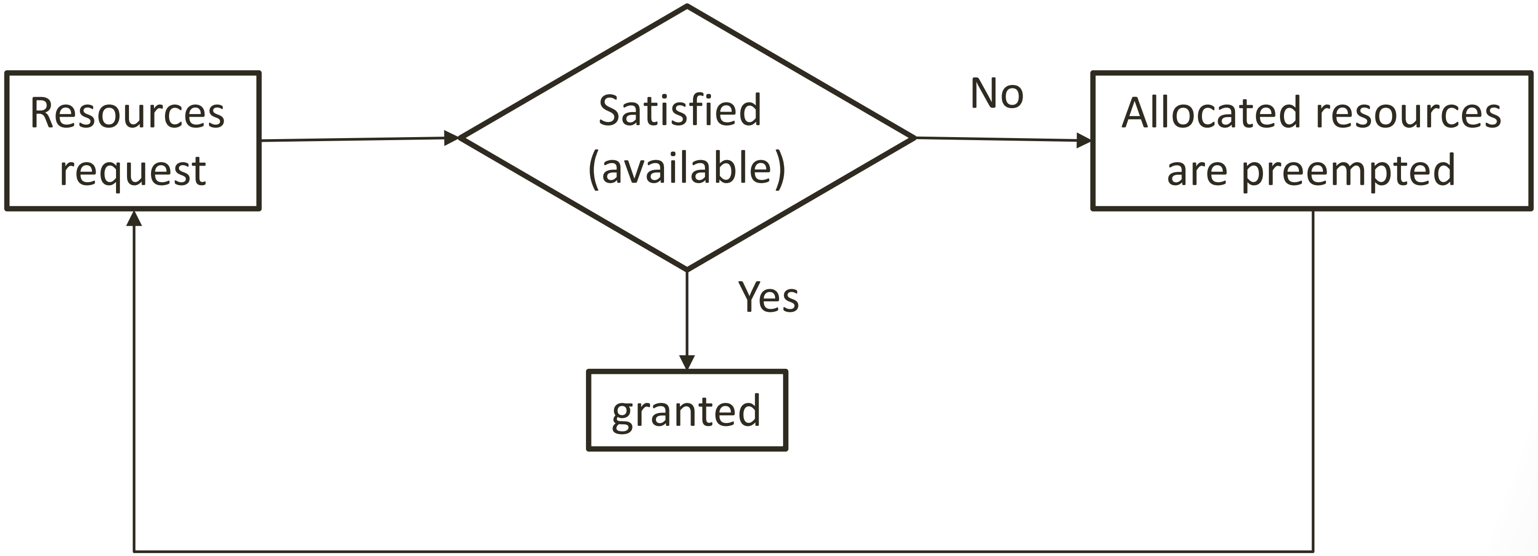
- **Mutual Exclusion**
  - not required for sharable resources
    - e.g., Read-only file
  - must hold for non-sharable resources
    - e.g., printer
- **Hold and Wait**
  - Acquire all needed resources before its execution.
  - Release allocated resources before request additional resources
  - Disadvantage
    - 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
    - O : CPU register, memory space
    - X : printers , tape drives.

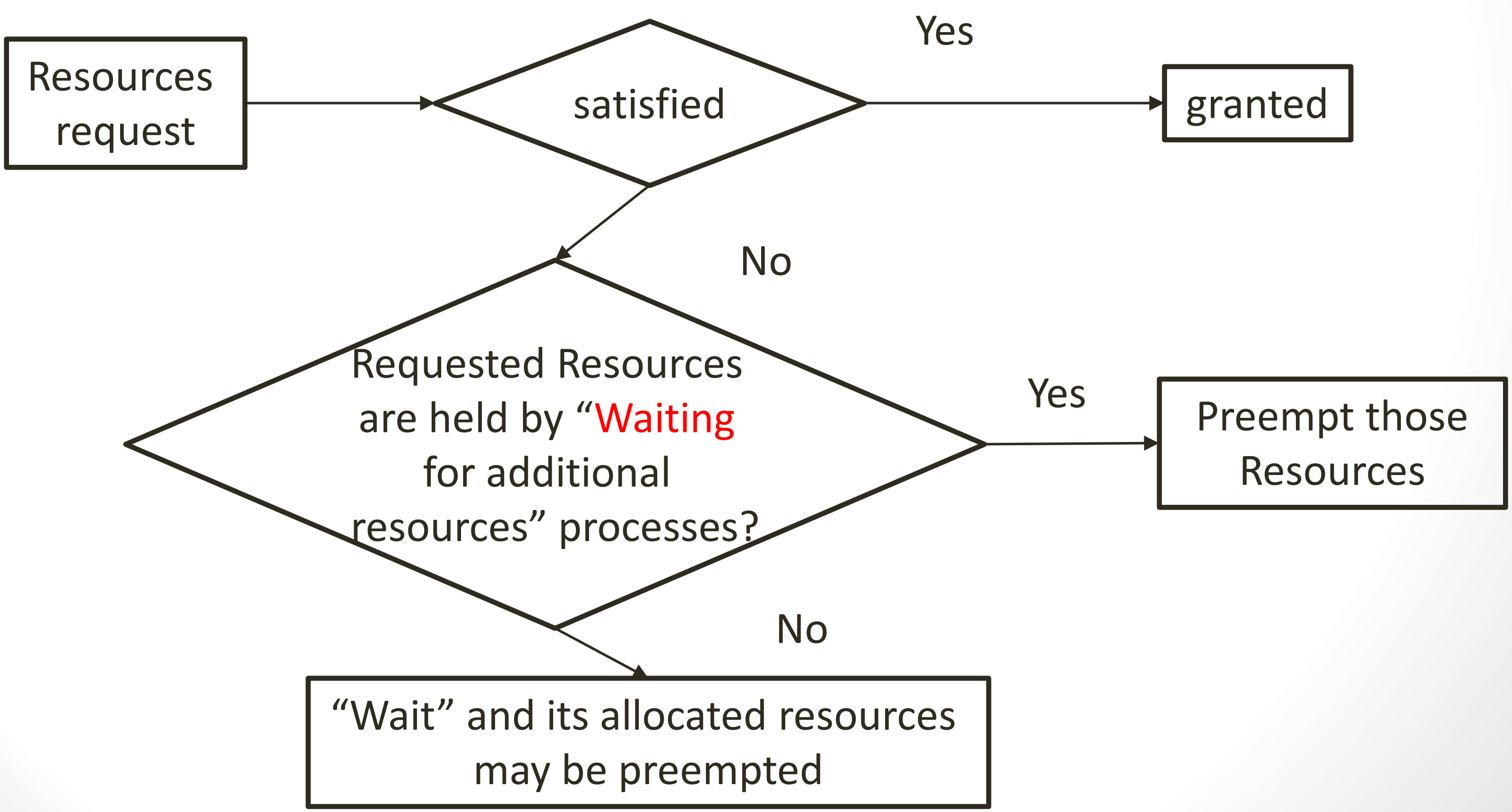
Deadlock Prevention

- **No Preemption**
  - Approach 1



Deadlock Prevention

- **No Preemption**
  - Approach 2



Deadlock Prevention

- **Circular Wait**
  - Resource requests must be made in an increasing order of enumeration
  - F : R -> N
    - R : set of resources
    - N: positive integer
    - e.g.
      - F(tape drive) = 1
      - F(disk drive) = 5
      - F(printer) = 12

Deadlock Prevention

- **Circular Wait**
  - Type 1: strictly increasing order of resource requests.
    - Initially, order any # of instances of Ri
    - Following requests of any # of instances of Rj must satisfy F(Rj) > F(Ri), and so on.

型態一：初始時已經擁有Ri，在之後的資源請求Rj，
其順序號碼必須大於手上的資源

  - Type 2
    - Processes must release all Ri's when they request any instance of Rj, if F(Ri) >=F(Rj)

型態二：每次索要新的資源Rj時，
必須釋放所有順序號碼大於等於Rj的資源

*總而言之，資源的索要需要依序從小到大。
因此，每次要違反這個順序時，必須釋放號碼更大的資源

Deadlock Prevention

- **Circular Wait**
  - Let the set of processes involved in the circular wait be $\{P_0, P_1, \ldots, P_n\}$
    - P_i -> R_i ->P_{i+1}
    - P_n -> R_n -> P_0.
  - R_i -> P_{i+1} -> R_{i+1}
    - We have $F(R_i) < F(R_{i+1})$ for all i.
  - $F(R_0) < F(R_1) < \ldots < F(R_n) < F(R_0)$
  - By transitivity, $F(R_0) < F(R_0)$, which is impossible. Therefore, there can be no circular wait.

DEADLOCK AVOIDANCE

Deadlock Avoidance

- Motivation:
  - Deadlock-prevention algorithms can cause low device utilization and reduced 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

- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition

Deadlock Avoidance

- Safe Sequence
  - A sequence of processes <P1, P2, …, Pn> is a safe sequence if

$$\forall Pi, need(Pi) \le Available + \sum_{\boxed{j<i}} allocated(Pj)$$

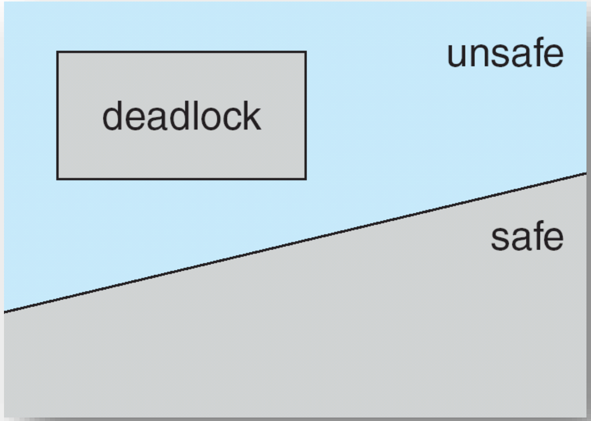
Pi的需求量要小於等於**<u>當下可用的</u>**加上**<u>之前的process</u>**占用的

j < i

- Safe State
  - The existence of a <u>safe sequence</u>

Deadlock Avoidance

- If a system is in safe state
  - no deadlocks
- If a system is in unsafe state
  - Has possibility of deadlock
- Avoidance
  - *Ensure that a system will never enter an unsafe state.



Deadlock Avoidance

- Example:

Total : 12

Process	MAX	Allocated
P_0	10	5
P_1	4	2
P_2	9	2

Deadlock Avoidance

Total : 12

Available = 12 - 5 - 2 - 2 = **3**

Process	MAX	Allocated	need
P_0	10	5	5
P_1	4	2	2
P_2	9	2	7

- The existence of a safe sequence <P1, P0, P2>

  i = 0, 1, 2

  - P1: 2 <= 3 (=12-5-2-2)
  - P0: 5 <= 3+2
  - P2: 7 <= 3+2+5

$$\forall Pi, need(Pi) \leq Available + \sum_{j<i} allocated(Pj)$$

Deadlock Avoidance

- If P2 got one more, the system state is unsafe

```
Total : 12
Available = 12 - 5 - 2 - 3 = 2
```

Process	MAX	Allocated	need
P_0	10	5	5
P_1	4	2	2
P_2	9	3	6

- The existence of a safe sequence <P1, P0, P2>
  - P1: 2 <= 2
  - P0: 5 ?? 2+2
  - P2: 7 <= 2+2+5

Avoidance algorithms

- Single instance of a resource type
  - Use a resource-allocation graph

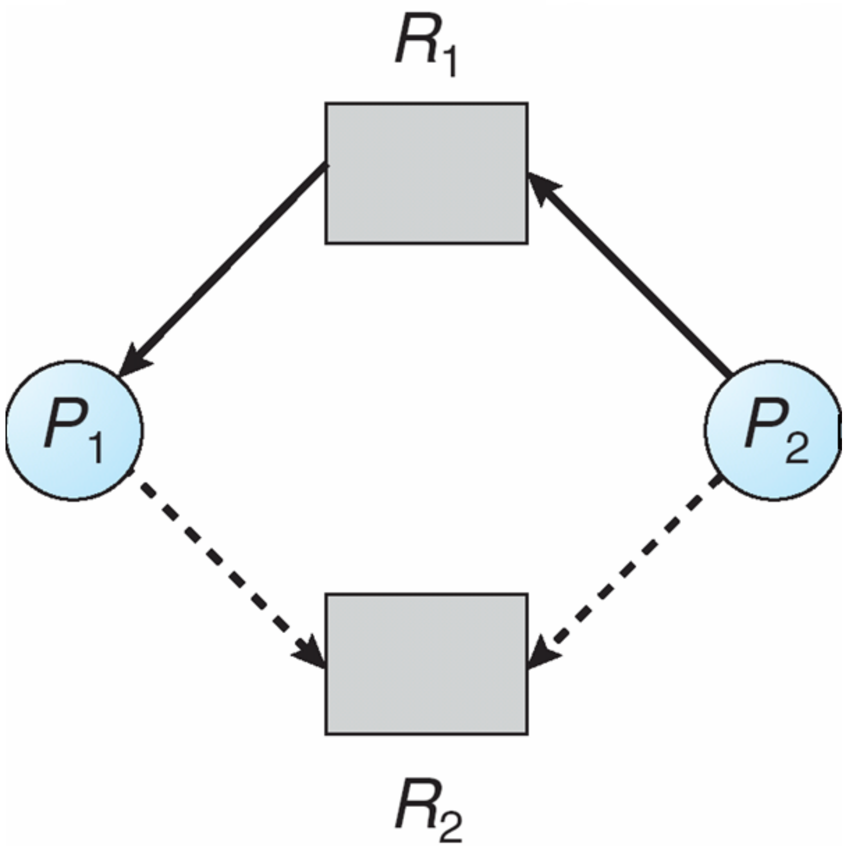
- Multiple instances of a resource type
  - Use the banker’s algorithm

Resource-Allocation Graph Scheme

- Claim edge $P_i > R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge $P_i > R_j$ when a process requests a resource
- Request edge converted to an assignment edge $R_i > P_j$ when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge

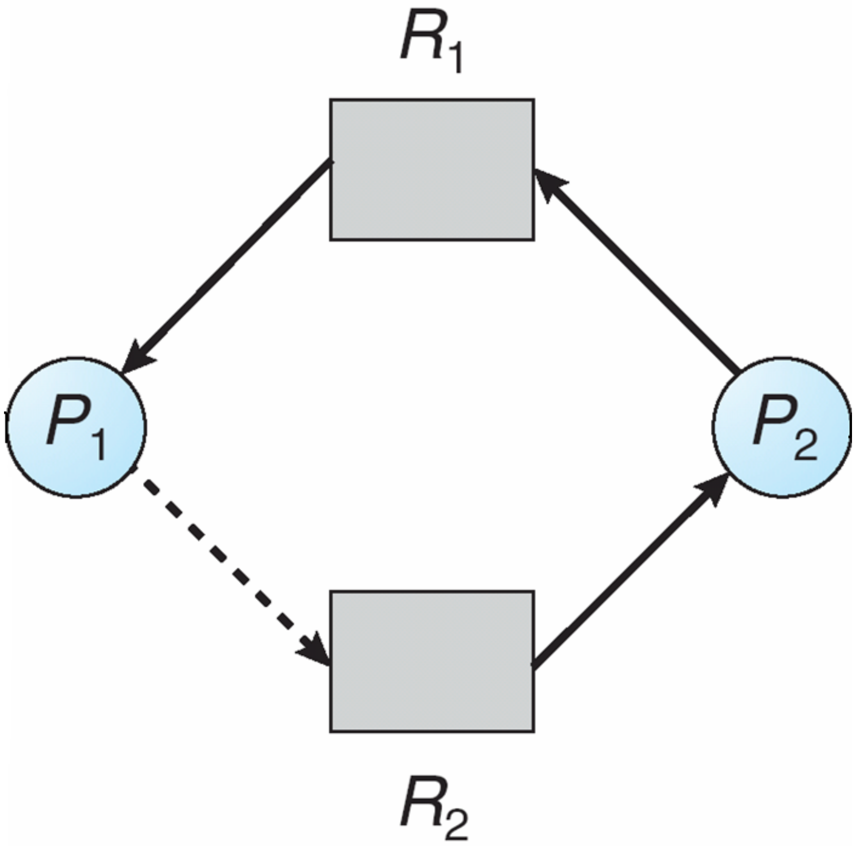
Detect the cycle on the graph

- Safe state : No cycle



Detect the cycle on the graph

- Unsafe state : Maybe have cycle



Resource-Allocation Graph Scheme

- Safe state: no cycle
- Unsafe state: otherwise
- Cycle detection can be done in $O(n^2)$

Banker's Algorithm - notations

- Available [m]
  - Vector of length *m*. If available [*j*] = *k*, there are *k* instances of resource type R_j available

紀錄每個資源還有多少可用

- Max [n,m]
  - If *Max* [*i*,*j*] = *k*, then process P_i may request at most *k* instances of resource type R_j

紀錄每個process，最多需要多少資源

- n: # of processes
- m: # of resources

Banker's Algorithm - notations

- Allocation [n,m]
  - If Allocation[i,j] = k then P_i is currently allocated k instances of R_j

紀錄每個process，已配置多少資源

- Need [n,m]
  - If *Need*[i,j] = k, then P_i may need k more instances of R_j to complete its task

紀錄每個process，還需多少資源

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

Banker's Algorithm

1. 請求比需求少合理
2. 比可用的少才可做
   否則等待
3. 試算
4. 評估

Request[*i,j*] = *k*, Pi 要 k 個 Rj的資源

means P_i wants *k* instances of resource type R_j

1. If *Request[i,j] <= Need[i,j]* *then* go to step 2.
   else Trap;
2. If *Request[i,j] <= Available[j]*, go to step 3.
   else P_i must wait,
3. Pretend to allocate requested resources to P_i by updating (modifying) the state as follows:
   - *Available[j] = Available[j] – Request[i,j];*
   - *Allocation[i,j]= Allocation[i,j]+ Request[i,j];*
   - *Need[i,j]= Need[i,j]– Request[i,j];*
4. Safety Algorithm
   - *If safe >> the resources are allocated to Pi*
   - *If unsafe >> Pi must wait, and the old resource-allocation state is restored*

$$\forall Pi, need(Pi) \le Available + \sum_{j<i} allocated(Pj)$$

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), *B* (5 instances), and *C* (7 instances)

Snapshot at time T_0:

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

Example of Banker's Algorithm

- The content of the matrix *Need* is defined to be *Max – Allocation*

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

Example of Banker's Algorithm

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

Sequence: P1,

Example of Banker's Algorithm

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

Sequence: P1, P3,

Example of Banker's Algorithm

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

Sequence: P1, P3, P4,

Example of Banker's Algorithm

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

Sequence: P1, P3, P4, P2,

Example of Banker's Algorithm

	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	7	4	3	10	4	7
P_1	2	0	0	3	2	2	~~1~~	~~2~~	~~2~~			
P_2	3	0	2	9	0	2	~~6~~	~~0~~	~~0~~			
P_3	2	1	1	2	2	2	~~0~~	~~1~~	~~1~~			
P_4	0	0	2	4	3	3	~~4~~	~~3~~	~~1~~			

Sequence: P1, P3, P4, P2, P0 -> safe

Example of Banker's Algorithm

- P_1 Request (1,0,2)

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

Example of Banker's Algorithm

- P_1 Request (1,0,2)

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

safe sequence P_1, P_3, P_4, P_0, P_2 -> granted

Example of Banker's Algorithm

- P_4 Request (3,3,0)

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

Request > Available ->reject

Example of Banker's Algorithm

- P_0 Request (0,2,0)

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

Example of Banker's Algorithm

- P_0 Request (0,2,0)

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

Example of Banker's Algorithm

- P_0 Request (0,2,0)

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

Sequence ? -> reject

(also, there is no available resource…)

Theorem

- If $(1)\ 1 \le Max_i \le m$ and $(2)\ \sum_{i=1}^{n} Max_i < n + m$ then deadlock free, where n : # of processes and m : # of instance
  - Proof:

    If deadlock exist , then set $\sum_{i=1}^{n} Allocation_i = m$

    By Banker's algorithm

$$\sum_{i=1}^{n} Need_i = \sum_{i=1}^{n} Max_i - \sum_{i=1}^{n} Allocation_i$$

$$\Rightarrow \sum_{i=1}^{n} Need_i = \sum_{i=1}^{n} Max_i - m$$

$$\Rightarrow \sum_{i=1}^{n} Max_i = \sum_{i=1}^{n} Need_i + m$$

Theorem

By $(2)\sum_{i=1}^{n} Max_i < n+m$

$$\sum_{i=1}^{n} Max_i = \sum_{i=1}^{n} Need_i + m < n+m$$

$$=> \sum_{i=1}^{n} Need_i < n$$

$$=> \exists i, Need_i = 0$$ - >< -

By $(1)\, 1 \le Max_i \le m$

when Pi finish, it will release some resource

then others finish those work.

DEADLOCK DETECTION

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance of Each Resource Type

- Maintain *wait-for* graph (V,E)
  - V: processes
  - E: $P_i > P_j$, if P_i is waiting for P_j

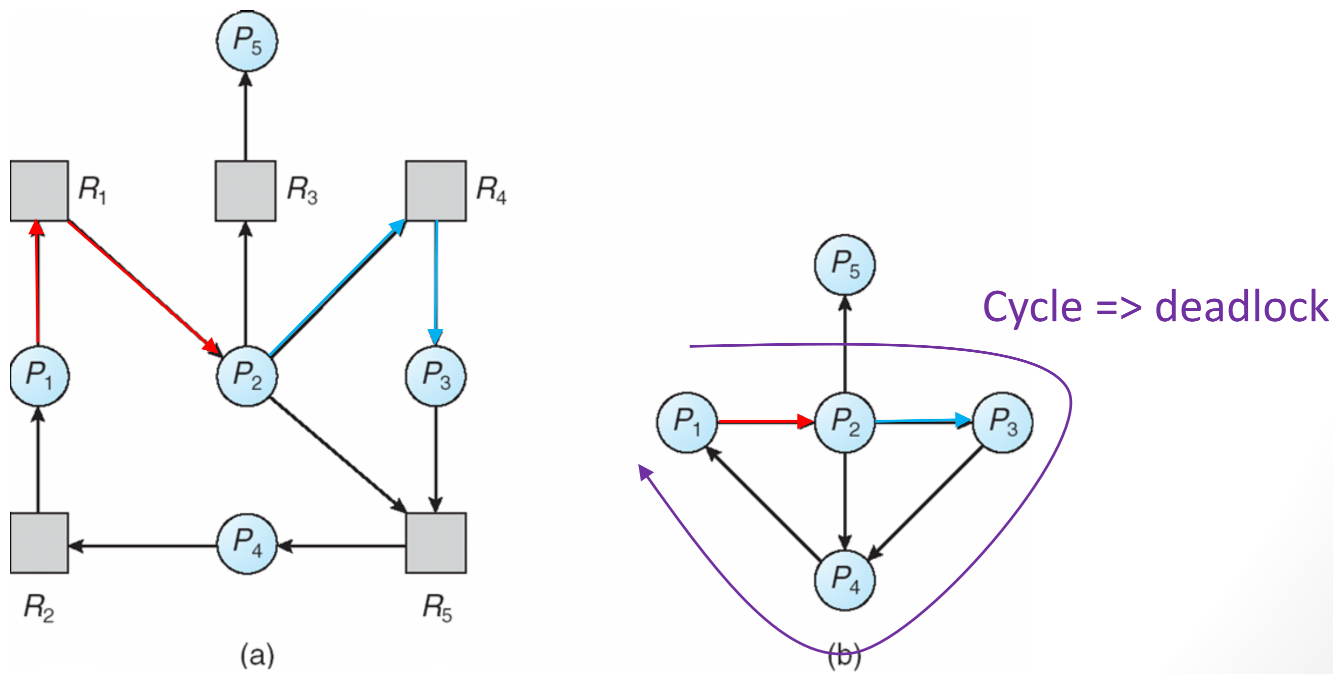
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock

- $O(n^2)$

Resource-Allocation Graph and Wait-for Graph

- In Resource-Allocation Graph : Pi -> Ri -> Pj
- In Wait-for Graph : Pi -> Pj

Resource-Allocation Graph   Corresponding wait-for graph



Deadlock detection Algorithm

- Available [m]
  - Vector of length *m*. If available [*j*] = *k*, there are *k* instances of resource type R_j available

紀錄每個資源還有多少可用

- Allocation [n,m]
  - If Allocation[*i,j*] = *k* then P_i is currently allocated *k* instances of R_j

紀錄每個process，已配置多少資源

- n: # of processes
- m: # of resources

Deadlock detection Algorithm

- Request [n,m]
  - The current request of each process. If *Request* [*i,j*] = *k*, then process P_i is requesting *k* more instances of resource type R_j

紀錄每個process所提出的資源申請量

Safety Algorithm

Work[1..m]：表示系統目前可用資源數量之累計

1. Let **Work** and **Finish** be vectors of length m and n, respectively. Initialize: 若Pi手中還有資源則Finish[i]為false
   - Work [j]= Available[j]
   - Finish [i] = false for i = 0, 1, …, n- 1

   O(n)

2. Find [i] to satisfied both conditions: Process需求量小於系統可用量才可做
   - Finish[i] = false
   - Need[i,j] <= Work[j]

   n+(n-1)+(n-2)+…+1 = O(n^2)

   *If no such i exists, go to step 4

3. Work[j] = Work[j] + Allocation[i,j]
   - Finish[i] = true
   - go to step 2

   Process可做完，資源可釋放

4. If Finish [i] == true for all i, then the system is in a safe state. (otherwise, the state is unsafe)

   $O(m*n^2)$

Example of Deadlock detection Algorithm

- Five processes P_0 through P_4; three resource types
  A (7 instances), *B* (2 instances), and *C* (6 instances)

	Allocation			*Request*			*Working*		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

Is deadlock?

Example of Deadlock detection Algorithm

	Allocation			*Request*			*Working*		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

Finish :

P0	P1	P2	P3	P4
F	F	F	F	F

所有process手中都有資源，所以都為false

Example of Deadlock detection Algorithm

	Allocation			Request			Working		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	1	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

	P0	P1	P2	P3	P4
Finish :	T	F	F	F	F

Example of Deadlock detection Algorithm

	Allocation			Request			Working		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	~~0~~	~~0~~	~~0~~	3	1	3
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

	P0	P1	P2	P3	P4
Finish :	T	F	T	F	F

Example of Deadlock detection Algorithm

	Allocation			Request			Working		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	~~0~~	~~0~~	~~0~~	5	2	4
P_1	2	0	0	2	0	2			
P_2	3	0	3	~~0~~	~~0~~	~~0~~			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

	P0	P1	P2	P3	P4
Finish :	T	F	T	T	F

Example of Deadlock detection Algorithm

	Allocation			Request			Working		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	~~0~~	~~0~~	~~0~~	5	2	6
P_1	2	0	0	2	0	2			
P_2	3	0	3	~~0~~	~~0~~	~~0~~			
P_3	2	1	1	~~1~~	~~0~~	~~0~~			
P_4	0	0	2	0	0	2			

	P0	P1	P2	P3	P4
Finish :	T	F	T	T	T

Example of Deadlock detection Algorithm

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	~~0 0 0~~	7 2 6
P_1	2 0 0	2 0 2	
P_2	3 0 3	~~0 0 0~~	
P_3	2 1 1	~~1 0 0~~	
P_4	0 0 2	~~0 0 2~~	

	P0	P1	P2	P3	P4
Finish :	T	T	T	T	T

All true => No deadlock

Example of Deadlock detection Algorithm

- P_2 requests an additional instance of type C

	Allocation			*Request*			*Working*		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	1			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

Is deadlock?

Example of Deadlock detection Algorithm

	Allocation			Request			Working		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	1	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	1			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

	P0	P1	P2	P3	P4
Finish :	T	F	F	F	F

Example of Deadlock detection Algorithm

	Allocation			*Request*			*Working*		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
P_0	0	1	0	~~0~~	~~0~~	~~0~~	0	1	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	1			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

	P0	P1	P2	P3	P4
Finish :	T	F	F	F	F

Deadlock : P1, P2, P3, P4

RECOVERY FROM DEADLOCK & COMBINED APPROACHES

Recovery from Deadlock

- Process Termination
  - Abort all deadlocked processes　砍掉所有process
  - Abort one process at a time until the deadlock cycle is eliminated

一次砍一個 process直到deadlock free

Recovery from Deadlock

- Resource Preemption
  - Selecting a victim – minimize cost
    - Rollback – return to some safe state, restart process for that state

選一個犧牲者搶奪其資源
被搶的process回到尚未取得資源的狀態

    - Starvation
      - same process may always be picked as victim, include number of rollback in cost factor

Combined Approaches

- Internal Resources
  - Resources used by the system, e.g., PCB
  - Prevention through resource ordering
- Central Memory
  - User Memory
  - Prevention through resource preemption

Combined Approaches

- Job Resources
  - Assignable devices and files
  - Use “Deadlock Avoidance”

- Swappable Space
  - Space for each user process on the backing store
  - Pre-allocation these resources

SUGGESTION!

OR

OBJECTION?

Let's stop here,

TAKE A BREAK