Chapter 7: Deadlocks

現在的OS大多不會管這個(要自己處理)
只有Realtime OS會處理

Prof. Li-Pin Chang

National Chiao Tung University

Chapter 7: Deadlocks

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

Chapter Objectives

- To develop a description 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.

SYSTEM MODEL

System Model

指同一時間只能被一個process使用的東西

- Resource $R_1, R_2, \ldots, R_m$
  - CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.

每種Resource可以有多份給process用

  - For example, DMA channels
- Each process utilizes a resource as follows:
  1. request
  2. use
  3. release

```
/* 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 CHARACTERIZATION

Deadlock 的特性

Deadlock Characterization

單向，有這四個不代表deadlock，但**deadlock**一定會有這四點

- If a deadlock arises, then the four conditions hold *simultaneously* 這四個條件中有一個以上不成立，**Deadlock**就不會發生
  - Mutual exclusion: only one process at a time can use a resource 同時只有一個**process**能用這個資源
  - Hold and wait: a process holding at least one resource is 已經占住某些資源，但 waiting to acquire additional resources held by other processes 又去要其他資源
  - No preemption: a resource can be released only voluntarily by 這個資源被鎖定 the process holding it, after that process has completed its task 後不會被其他人搶走
  - Circular wait: there exists a set $\{P_0, P_1, \ldots, P_n\}$ 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_n is waiting for a resource that is held by P_0. 循環等待

Resource-Allocation Graph

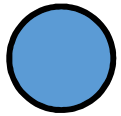
- A set of vertices V and a set of edges E.

- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.

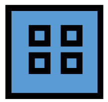
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

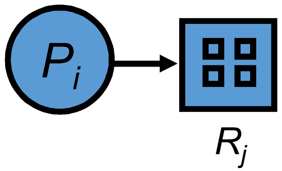
- Process



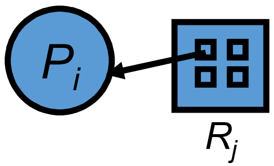
- Resource Type with 4 instances



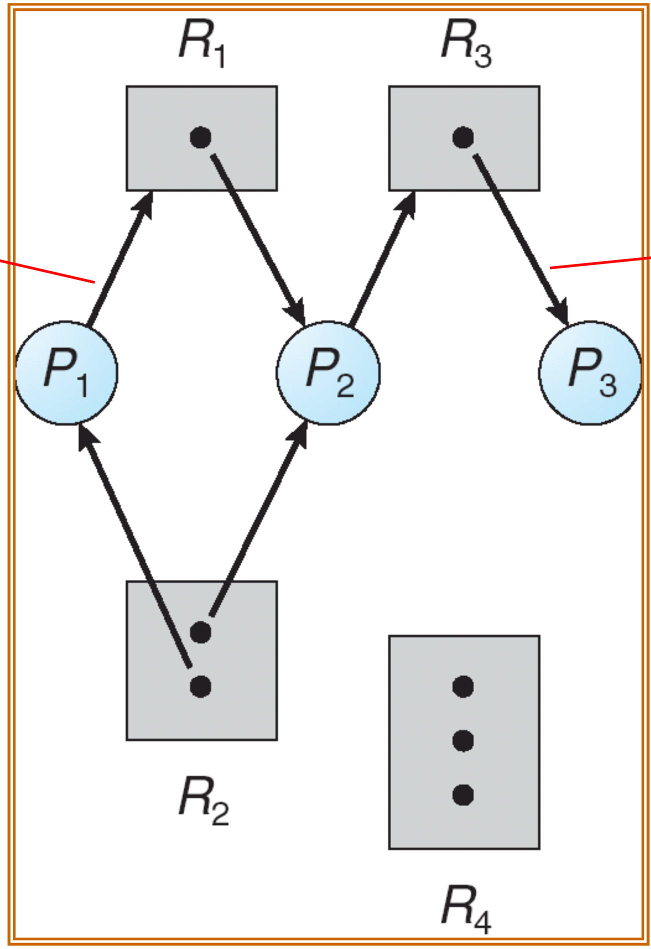
- P_i requests instance of R_j



- P_i is holding an instance of R_j



Example of a Resource Allocation Graph



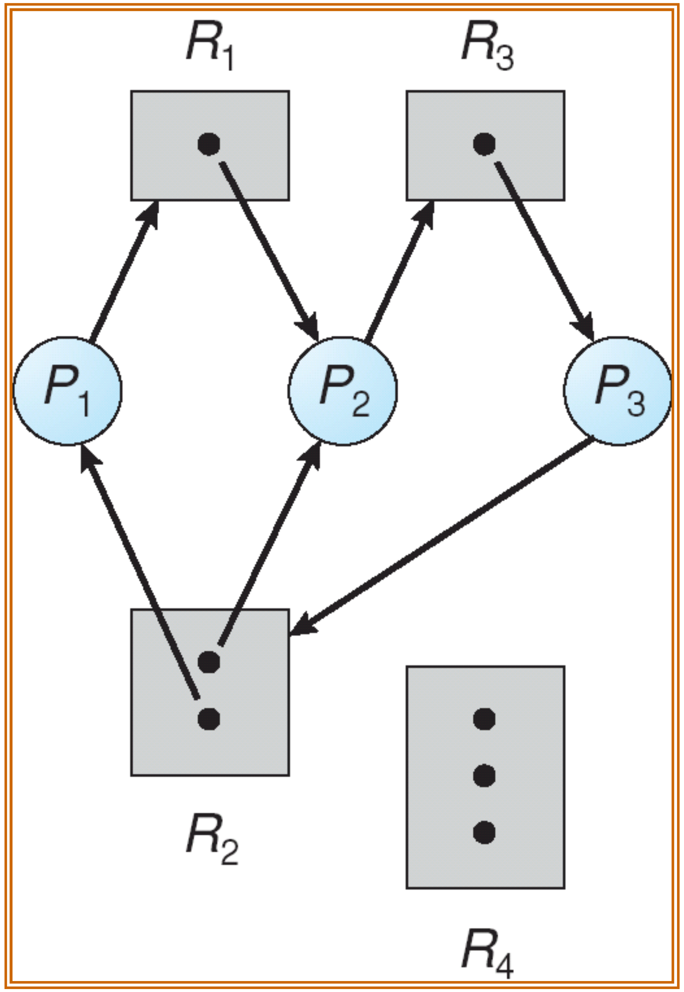
拿不到

該資源被Pi拿走

Resource Allocation Graph With A Deadlock

The system is deadlocked

There is a cycle in the graph

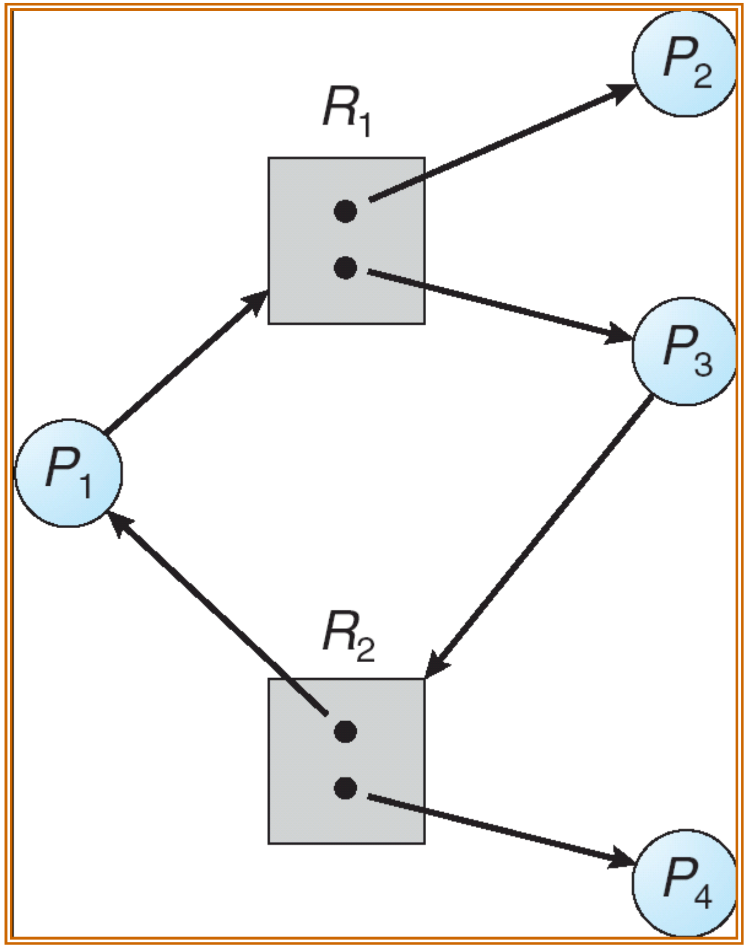


Resource Allocation Graph With A Cycle But No Deadlock

The system is not deadlocked

There is a cycle in the graph

代表有cycle不一定有deadlock



Basic Facts

- Resources have single instance
  - There is a cycle ←→ deadlock 一定會形成deadlock，因為無法解開(要的東西只有一份)
- Resources have multiple instances
  - Deadlock → there is a cycle 一旦環上某個resource被環外的prcess丟回來了Deadlock就會解開

- If graph contains no cycles ⇒ no deadlock
- If graph contains a cycle ⇒
  - Resources have single instance, then deadlock
  - Resources have multiple instances, then *possible* deadlock

METHODS FOR HANDLING DEADLOCKS

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state (prevention or avoidance), 對deadlock發生時的特性，使之永遠不成立
- allow the system to enter a deadlock state and then recover (detection), or 發現thread都睡死起不來→OS進來檢查資源狀況，有沒有deadlock
- ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.
  表示Deadlock不是OS該解決的問題

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state 用規則來保證永遠不會發生Deadlock
  - Deadlock prevention: a set of rules that guarantees that one or more the necessary conditions never happens (changing the programming model or the way that the OS manages resources)
  - Deadlock avoidance: to test whether a request to resources is safe or not (request may be delayed even when the requested resources are available) 做一個動作前先測試看看會不會有Deadlock

DEADLOCK PREVENTION

Deadlock Prevention

以下是對Deadlock四特性的prvention

- Mutual Exclusion – must be true for serially reusable resources 不可能預防，Mult-processes program幾乎一定會需要這特性
- Hold and Wait – must guarantee that whenever a process requests a resource, it is not holding any other resources.
  - All or none
  - [R1----[R2----]-----] ➔ [Rv------------------] 這樣會讓一些資源閒置，thread卻不能用

    nested critical sections合併成一個，要使用R1 or R2就要把Rv鎖起來
  - Low concurrency among processes due to long critical sections

可行但不實際

Deadlock Prevention (Cont.)

- No Preemption –
  - If a process (victim) holds a resource R but is waiting for another resource, R will be preempted when another process tries to acquire R
  - The victim process will be restarted when R is available again
  - Requiring a checkpointing mechanism

先對要改的東西做某種程度的備份
當被preempted時，recover回修改前

- Circular Wait – impose a total ordering or partial ordering on all resource types, and require that each process requests resources in an increasing order of enumeration. 一定要先鎖一個資源再鎖另一個，避免環產生
  - E.g., R1→R2 but no R2→R1

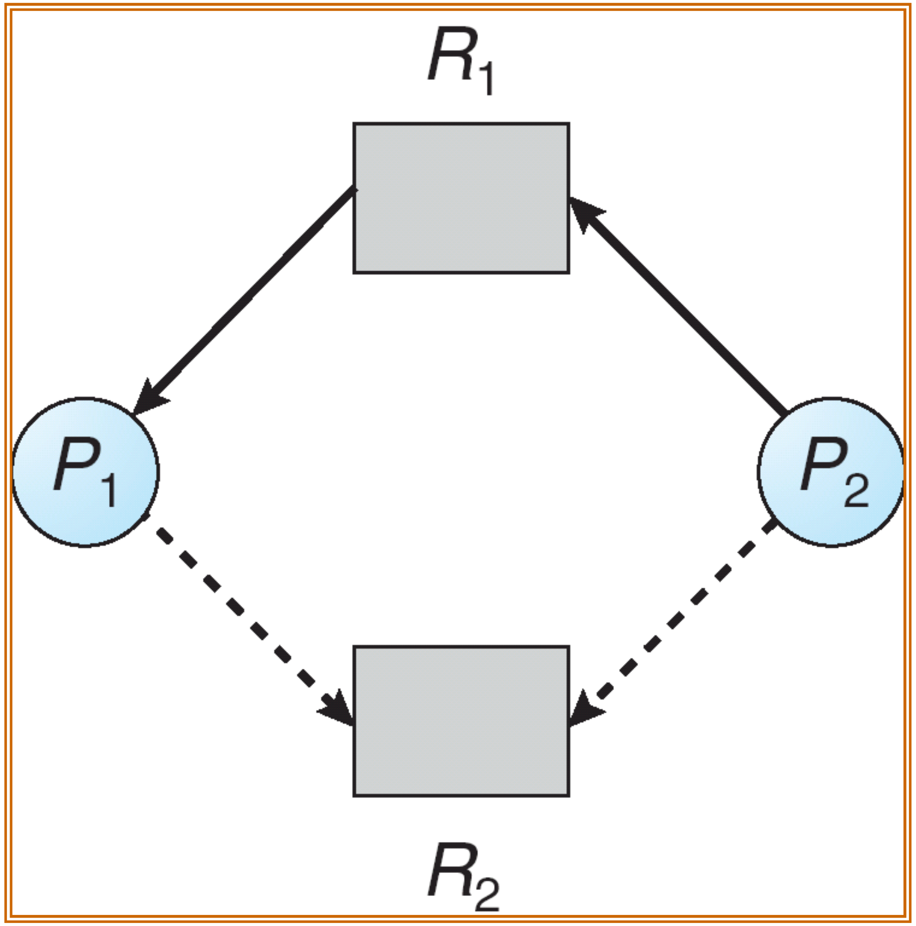
DEADLOCK AVOIDENCE

Deadlock Avoidance

- 1 instance per resource
  - Deadlock ←→ cycle (s) 獲得
  - Resource acquisition must not create cycle(s) in the resource allocation graph
- Deadlock avoidance based on cycle detection in resource allocation graphs

假裝先給資源，看一下有沒有形成cycle再決定要不要真的給

Resource-Allocation Graph For Deadlock Avoidance



示意圖，有bug

Claim edge: may use a resource at some time
Request edge: is requesting a resource
Assignment edge: is holding a resource

Resource-Allocation Graph Algorithm

在真的分配資源給Process前，預先測一下這樣會不會產生cycle

- Claim edge Pi → Rj indicated that process Pj may request resource Rj; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- When a resource is released by a process, assignment edge reconverts to a claim edge

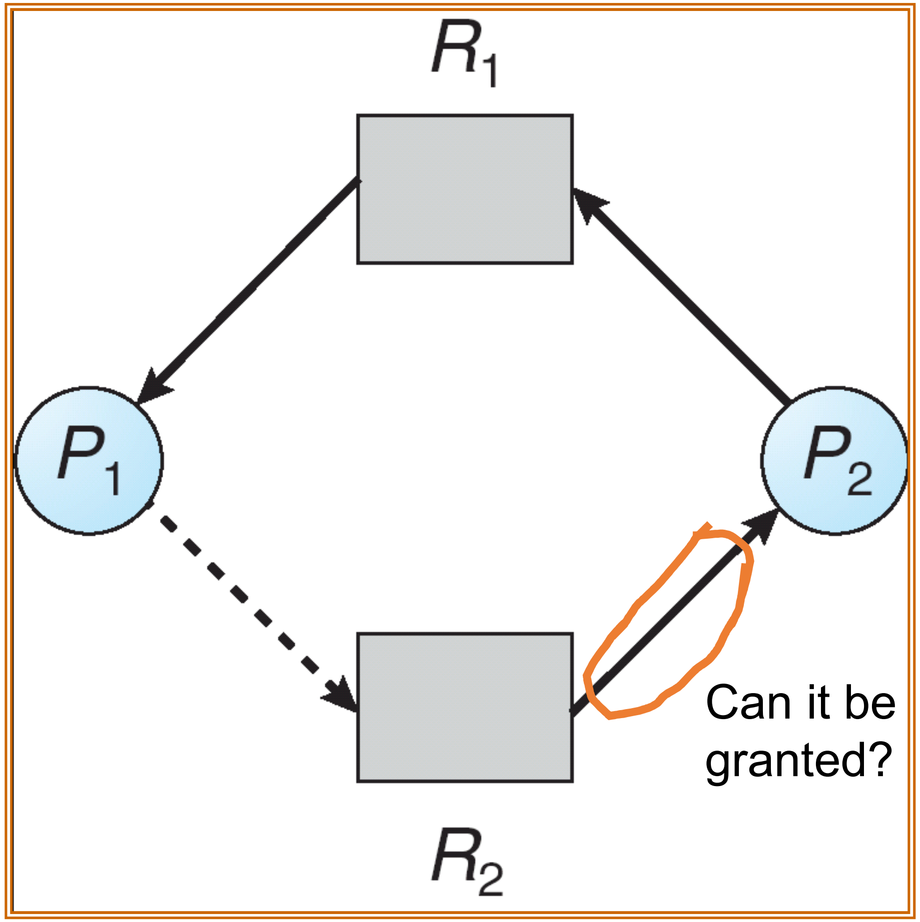
由因及果的

- Resources must be claimed *a priori* in the system.

Deadlock Avoidance for 1-Instance Resources

1. Initially, put all claim edges
2. When a process requests a resource, convert the claim edge into request edge
3. If the resource is available, 試驗性地 tentatively change the request edge into assignment edge and check if there are any new cycles(s) in the resource-allocation graph
4. If new cycle(s) exist, revert the allocation edge back to request edge and put the process waiting; Otherwise, the resource is allocated to the process

Unsafe State In Resource-Allocation Graph



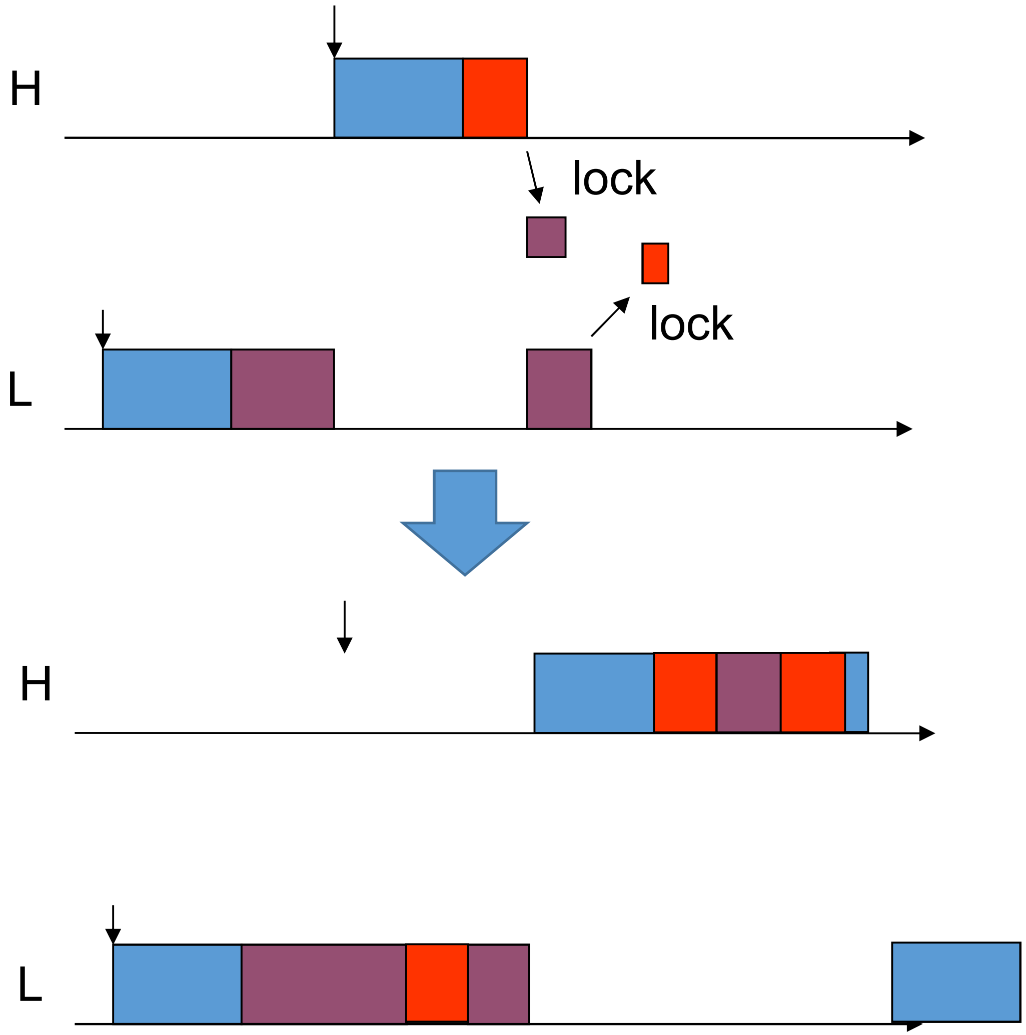
To detect cycles before an request can be granted

How to detect cycle(s) in the resource-allocation graph?

Topological Sort or DFS
可以檢測有沒有cycle

There’s a bug in this example…

Another Deadlock Avoidance Strategy: Highest Locker's Protocol in RTOS



priority會隨著process鎖到什麼資源而改變
→避免cycle(需要證明，研究所)

A process's priority is boosted to the highest among the lockers' priorities

Deadlock Avoidance

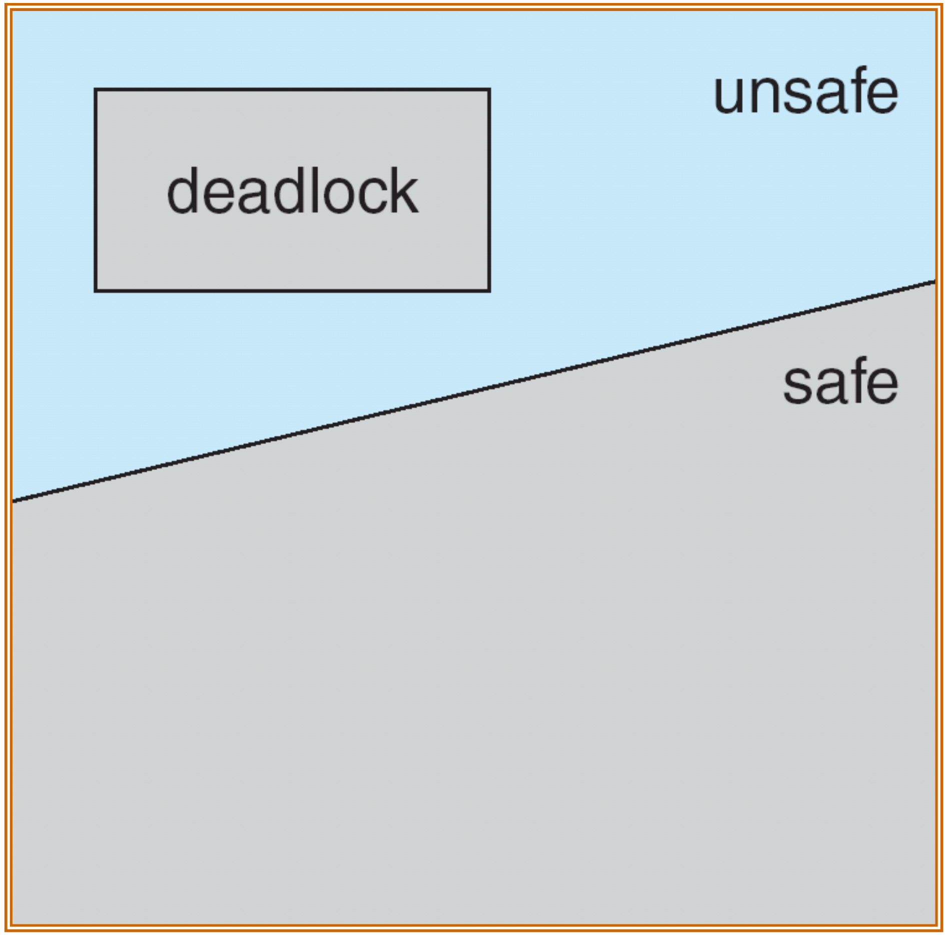
- N instances per resource
  - The graph-based approach is still applicable

- A more general approach
  - Safe/unsafe-state method
  - A system is safe → the system has no deadlock
  - The system must always be in a safe state; resource acquisition cannot put the system in a unsafe state
  - Need a definition on “safe state”

Basic Facts

- 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

Safe, Unsafe , Deadlock State in Banker's Alogorithm



Deadlock Avoidance

- Requires that the system has some additional *a priori* information available
- Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in safe state if there exists a safe sequence of all processes
- Sequence <P1, P2, …, Pn> is safe if for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the Pj, with $j<i$
  - If Pi resource needs are not immediately available, then Pi can wait until all Pj have finished
  - When Pj is finished, Pi can obtain needed resources, execute, return allocated resources, and terminate
  - When Pi terminates, Pi+1 can obtain its needed resources, and so on

P_i要求的資源，可以從前面的P或可用的資源中拿到

若存在這條sequence，P_i就必定能透過上面的狀況拿到資源

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. 一條vector記錄每種type有多少可用
- *Max:* n x m matrix. If *Max* [i,j] = k, then process P_i may request at most k instances of resource type R_j. 一個矩陣紀錄P_i對R_j最多可能的要求
- *Allocation:* n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j. 一個矩陣紀錄P_i對R_j現在已經收到的instance
- *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. 一個矩陣紀錄P_i對R_j還需要多少instance

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

Safety Algorithm

n: process #; m: resource #

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively.
   Initialize:

   Work = Available

   Finish [*i*] = *false* for *0~n*

2. Find and *i* such that both: O(n)

   (a) *Finish* [*i*] = *false* 把空閒resource給P_i

   (b) $Need_i \leq Work$ O(m)

   If no such *i* exists, go to step 4.

3. $Work = Work + Allocation_i$ P_i完成
   Finish[*i*] = *true*
   go to step 2. O(n)

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

對每一個process開頭 O(n)
然後找下一個process O(n)
然後確認每一個資源符合 O(m)

O(m*n²)

為什麼要O(m*n²)?

Resource-Request Algorithm for Process P_i

Request = 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 Pi.*
- *If unsafe $\Rightarrow$ Pi must wait, and the old resource-allocation state is restored*

Example of Banker's Algorithm

- 5 processes P_0 through P_4; 3 resource types A (10 instances), B (5instances, and C (7 instances).
- Snapshot at time T_0:

pro最多拿多少

	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 (Cont.)

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

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria.

safe：Need[i] < Σ(j=1~i-1)Allocation[j] + Avaliable

Example (Cont.)

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

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria.

Example P_1 Request (1,0,2) (Cont.)

- Check that Request ≤ Available (that is, R_1(1,0,2) ≤ (3,3,2) ⇒ true.

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

- Executing safety algorithm shows that sequence <P1, P3, P4, P0, P2> satisfies safety requirement.
- Can request for (3,3,0) by P4 be granted?
- Can request for (0,2,0) by P0 be granted?

If P0 (0,2,0) was made…

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

Discussions: Safe State

- Why all processes make their largest resource requersts in the check?
  - Rationale: if processes do not request the largest amount of resources, the problem becomes easier

    要更少資源→更好解決

DEADLOCK DETECTION

Deadlock Detection& Recovery

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

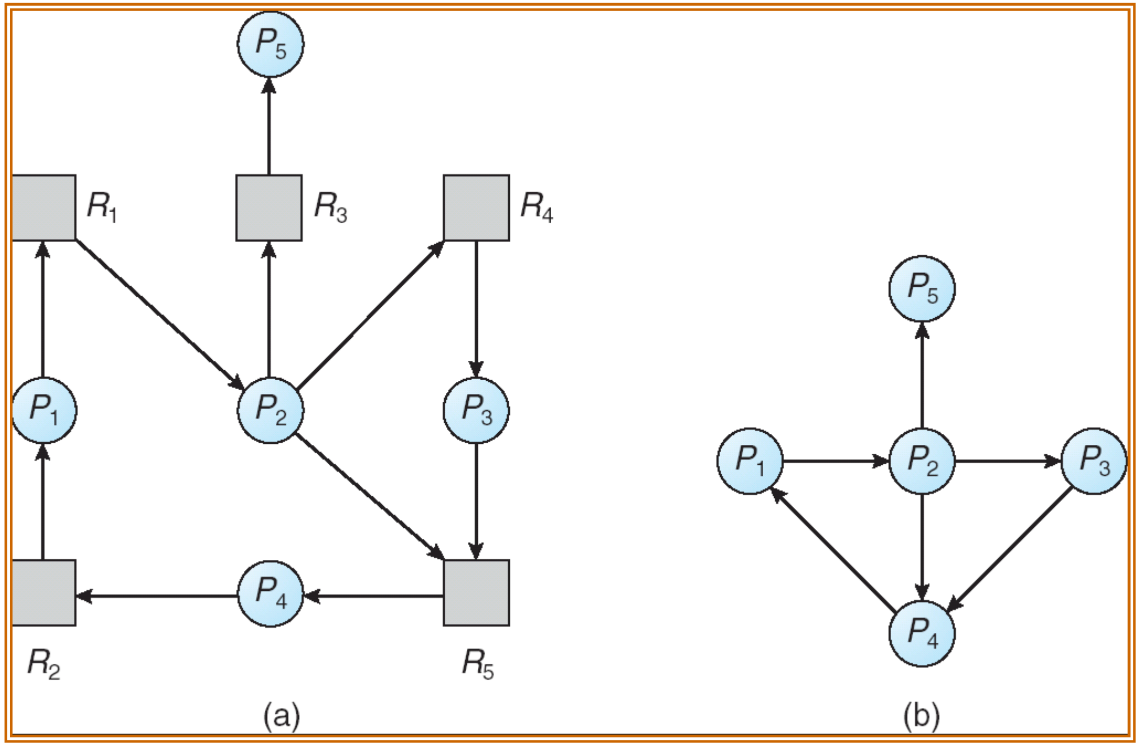
Single Instance of Each Resource Type

- Maintain wait-for graph
  - Nodes are processes.
  - Pi $\rightarrow$ Pj if Pi is waiting for Pj.

- Periodically invoke an algorithm that searches for a cycle in the graph
  - Cycle detection is more efficient than cycle search

- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph
  - Topological sort

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph Corresponding wait-for graph

Several Instances of a Resource Type

- Available: A vector of length m indicates the number of available resources of each type.

- Allocation: An n x m matrix defines the number of resources of each type currently allocated to each process.

- Request: An n x m matrix indicates the current request of each process. If Request [ij] = k, then process Pi is requesting k more instances of resource type. Rj.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:
   (a) *Work* = *Available*
   (b) For i = 1,2, …, n, if $Allocation_i \neq 0$, then
   Finish[i] = false;otherwise, *Finish*[i] = *true*.
2. Find an index i such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$

   If no such i exists, go to step 4.

Multiple-Instance Resources Deadlock Detection

- 用banker’s algorithm 判斷系統是否已經在 unsafe state
  - 不是檢查是否已經有 deadlock （因為沒有充分必要條件）沒有足夠的抽象條件去檢查是否有**deadlock**
- 如果已經在unsafe state→ kill some processes

Detection Algorithm (Cont.)

3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   go to step 2.

4. If $Finish[i]$ == false, for some i, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if $Finish[i]$ == *false*, then P_i is deadlocked.

Algorithm requires an order of O($m \times n^2$) operations to detect whether the system is in deadlocked state.

The same as the step to check the existence of a safe sequence. If there is no safe sequence, then deadlock "**may**" occur.

Because: deadlocks → not in a safe state

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances).
- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	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	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*.

Example (Cont.)

- P_2 requests an additional instance of type *C*.

	Allocation	*Request*	*Available*
	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	

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes' requests.
  - Deadlock exist~~s, consisting~~ of processes P_1, P_2, P_3, and P_4.

"may exists"

Detection-Algorithm Usage

- When to run the deadlock detection algorithm?
  - Upon a resource request is granted 確保safe
  - When the system throughput drops 可能有deadlock發生
- If a deadlock is detected,
  - roll back some processes until the deadlock is removed

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes.

- Abort one process at a time until the deadlock cycle is eliminated.

- In which order should we choose to abort?
  - Priority of the process.
  - How long process has computed, and how much longer to completion.
  - Resources the process has used.
  - Resources process needs to complete.
  - How many processes will need to be terminated.
  - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process for that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.

避免starvation發生

SUMMARY

- If there are deadlocks, then (→)
  - Mutual exclusion && hold and wait && non-preemptible && circular wait
    - Necessary conditions (must hold simultaneously)
    - p→(q1 && q2 && q3 && q4)
  - If any one of the four is invalid, then there is no deadlocks
    - ~(q1 && q2 && q3 && q4) →~p
    - ~q1 || ~q2 || ~q3 || ~q4 →~p

- Deadlock prevention
  - Rules to guarantee that one or more of the 4 necessary conditions are invalid, so that deadlocks are impossible
- Deadlock avoidance
  - No rules on resource usages
  - Check if the system may possibility have a deadlock when locking a resource
    - Reject a request that may cause deadlocks

- If each resource has only one instance
  - Deadlock $\leftrightarrow$ cycles exist in the resource allocation graph
- If a resource have >= 1 instance
  - Deadlocks $\rightarrow$ cycles

- If every resource has exact one instance
  - Safety check: cycle detection in the resource-allocation graph
  - The system is in a safe state $\leftarrow\rightarrow$ no deadlocks

- If a resource has > 1 instance
  - Safety check: banker's algorithm
  - The system is in a safe state → no deadlocks
  - The system is in an unsafe state → possible deadlocks

  - Deadlock→unsafe

End of Chapter 7