8. Deadlocks

* **Def: 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 – distributed or not.**

– Competing process

– Resources: CPU, files, printer, memory …..
Several instances of the same type!

– A normal operation mode
1.Request: granted or kept waiting (*open*)
2.Use
3.Release (*close*)

* Remarks

  - No request exceeds the system capacity!

  - Deadlock can involve <u>different resource types</u>!

* Deadlock: A situation in which one or more processes in a system are blocked forever because of requirements that can never be satisfied.

  A set of processes is deadlocked if each process in the set is waiting for an event that only another process in this set can cause.

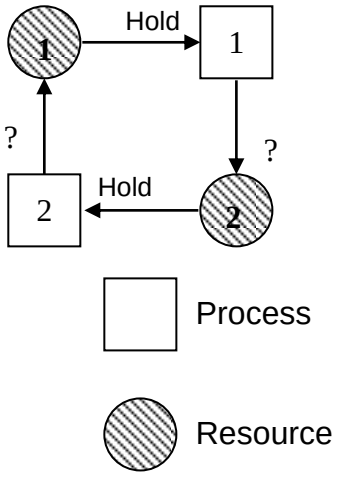
  Permanent blocking of a set of processes that either compete for system resources or communication with each other.

  Non-preemptable resource: one that can not be taken away from its current owner without causing the computation to fail. Ex. Printer.

  Deadlock occurs if each process holds one resource and requests another.

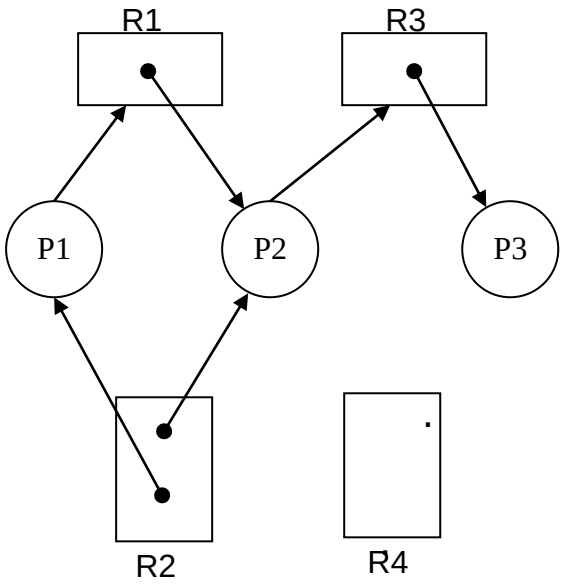
  A deadlocked process waits for resource (held by another process) that will never be released.
  (* different from starvation: Starvation occurs when some process waits for resources that periodically become available but are never allocated to that process due to some scheduling policy. *)

* For Example
  Process: P1, P2.
  Resource: R1, R2.
  R1→P1; R2→P2.
  P1? R2; P2? R1.
  ⇨P1 and P2 are deadlocked.



* Deadlock Examples

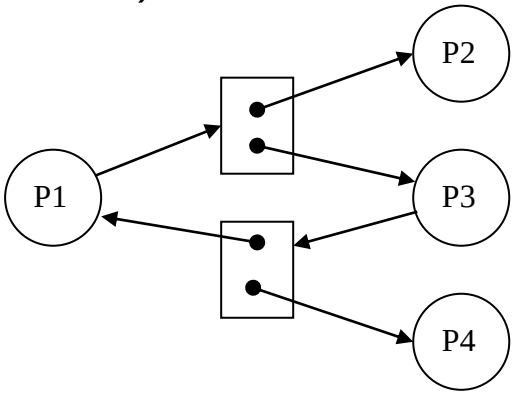
Example: (No deadlock)



* Vertices 2 Edges
  - P= { P1, P2, P3 }
  - R= { R1, R2, R3, R4 }
  - E= { P1→R1, P2→R3, R1→P2, R2→P2, R2→P1, R3→P3 }

* Resources
  R1:1 R2:2 R3:1 R4:3

* Deadlock ↔ The existence of a cycle
  - One instance / Resource type ⇨ Yes!!
  - Else ⇨ Only a necessary condition!! (but not sufficient)



A cycle but no deadlock

No cycle → no deadlock
deadlock → 必存在 cycle
有 cycle → ?

(1) P1 P2

request (D) request (T)

request (T) request (D)

release (T) release (D)

request (D) release (T)

(* resource deadlock *)

(2) Total memory size= 20 KB

P1 P2

request 8KB request 7KB

request 6KB request 8KB

(* but memory is preemptable→use swaping to solve *)

(3) P1 P2

receive (P2, M) receive (P1, M)

send (P2, M') send (P1, M')

(* communication deadlock *)

* One easy solution→time-out.

* Deadlock policies

(1) Detection and recovery.

(2) Prevention.

(3) Avoidance.

* How to deal with the deadlock problem?

Solutions:

1. Make sure that the system never enter a deadlock state!

   – Deadlock Prevention: fail any one of the necessary conditions.

   – Deadlock Avoidance: Processes provide information regarding their resource usage. Make sure that the system stays at a “safe” state!

2. Do recovery if the system is deadlocked.

   – Deadlock Detection

   – Recovery

3. 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! (Starvation)

* Deadlock characterization

– Necessary Conditions
(deadlock→conditions or ¬conditions→¬deadlock)└─

1. Mutual Exclusion – At lease one resource must be held in a nonsharable mode!

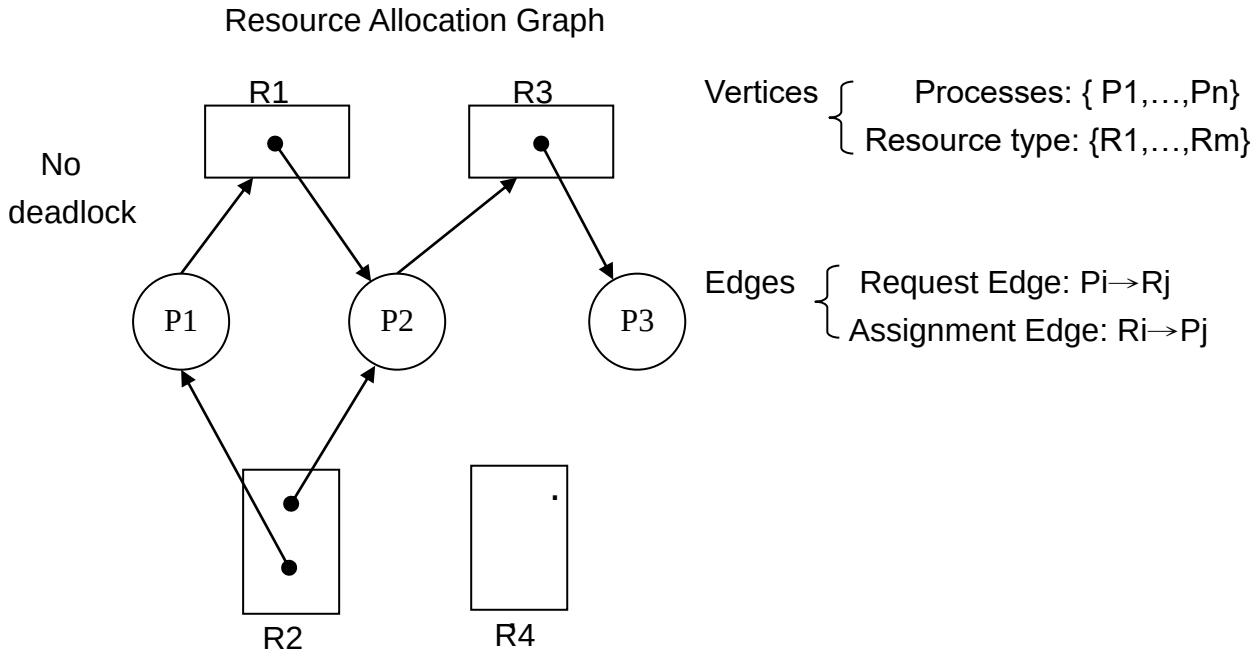
2. Hold and Wait – Pi is holding one resource & waiting to acquire additional resources that are currently held by other process.

3. No preemption – Resources are nonpreemptible!

4. Circular Wait – There exists a set { P0, P1, …, Pn } of waiting process such that P0 $\xrightarrow{want}$ P1, P1 $\xrightarrow{want}$ P2, …,Pn-1 $\xrightarrow{want}$ Pn, and Pn $\xrightarrow{want}$ P0.

* Condition 4 implies condition 2, and conditions are not completely independent!

– Resource Allocation – a description!



Try to fail anyone of the necessary condition!

∵¬ (∧condition i) →¬deadlock

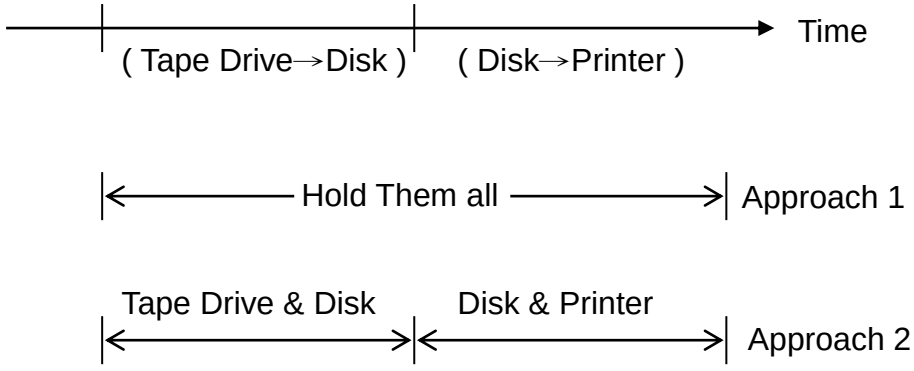
– Mutual Exclusion
  ?? Some resources such as a printer are intrinsically nonsharable. ?? (depend on resource types; memory, Ok; shared files for read, ok.)

– Hold and Wait

  * Acquire all needed resources before its exec.

  * Release allocated resources before request additional resources.

  * Eg-

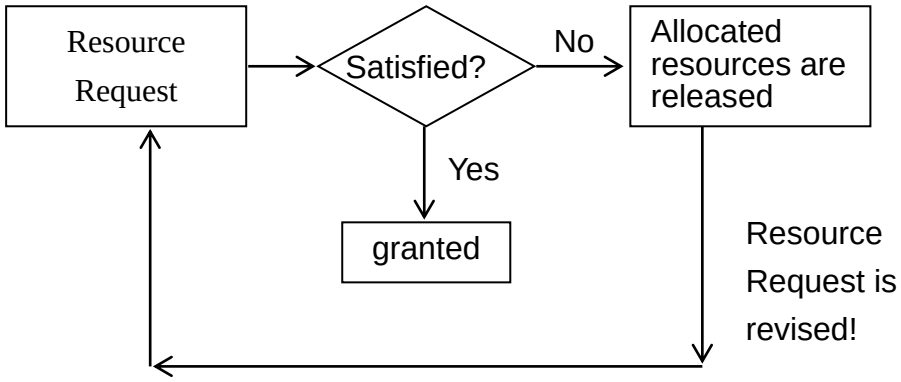


  * Disadvantage:
    - low resource utilization
    - starvation ~ may never get all you want at a time!

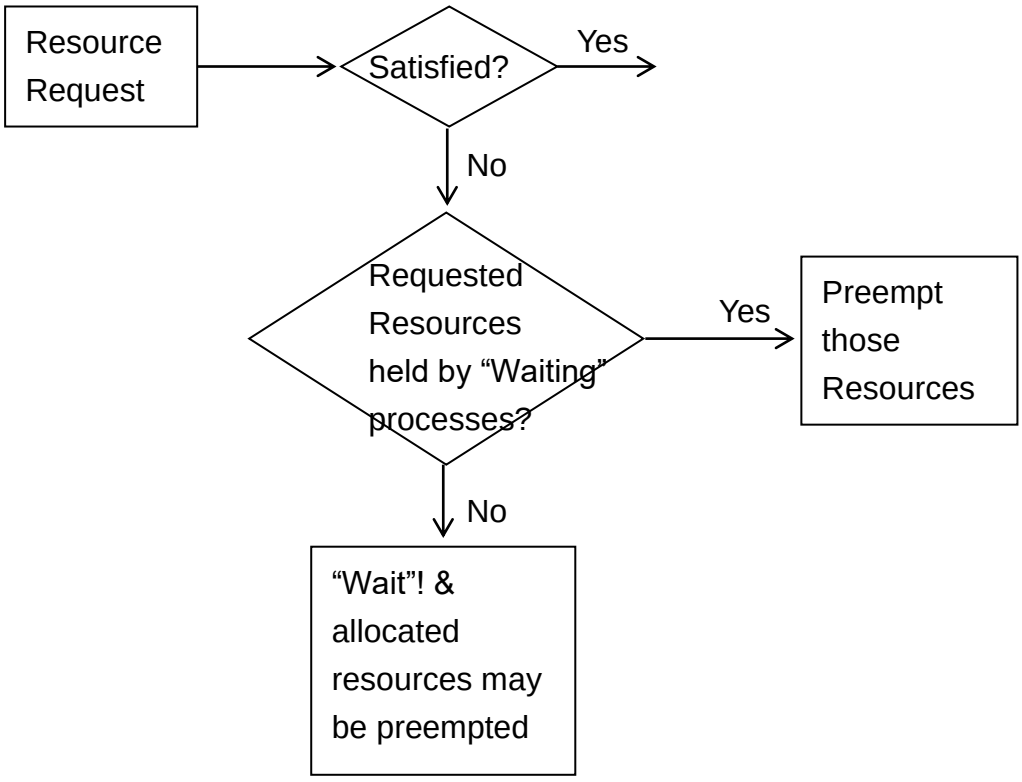
– No Preemption

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

* Approach 1:



* Approach 2:



– Circular Wait

A resource-ordering approach: (impose a total order on the resource)

F: R→N
Resource requests must be made in an increasing order of enumeration.

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

– A single request must be issued for all needed instance of the same resources.

* Type 2
  ~ Processes must release all Ri's when they request any instance of Rj if F(Ri) ≧ F(Rj)

* F : R→N must be defined according to the <u>normal order</u> of resource usages in a system, eg.,

F(tape drive)= 1
F(disk drive)=5
F(printer) =12

?? feasible ??

(常用的，放前面號碼)

* Deadlock Prevention

Design the system so that deadlock is not possible.
A system which is not secure from deadlock can sometimes be made **secure** by prohibiting operations that may lead to deadlock.
if Secure ⇨ no deadlock; if deadlock ⇨ no secure.

* Denying at least one of the four following conditions, all of which are necessary for deadlock to occur:

  (1) Mutual exclusion: process hold resources exclusively, making them unavailable to other processes.
  (resource in a nonsharable mode)
  (attack ⇨ some resource are sharable)

  (2) Partial allocation: processes may hold some resources when they request additional units of the same of other resource.
  (hold and wait)
  (attack ⇨ require all initially)
  (attack ⇨ release all old resources before ask for new resources)
  (disadv: low system utilization, starvation)

  (3) Nonpreemption: resources are not taken away from a process holding them; only processes can release resource they hold.
  (attack ⇨ allow preempting the resources; some resources are preemptable.)
  (when waiting occur→release all you have.)

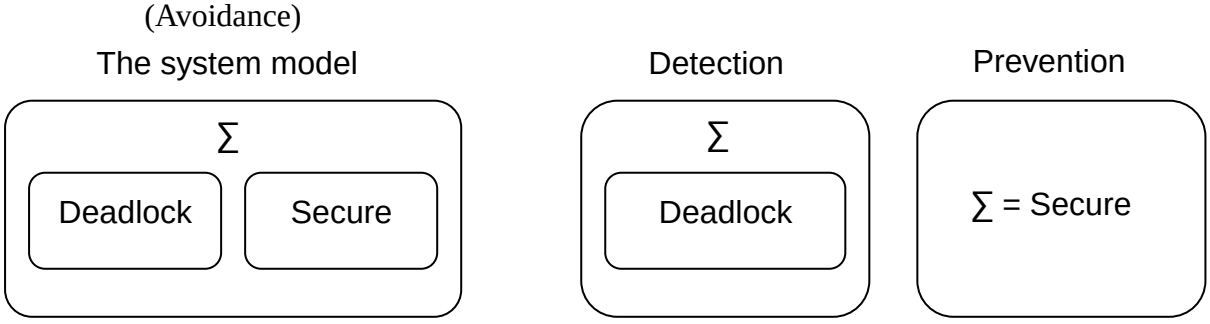
(4) Resource waiting: process that request unavailable units of resources block until they become available. (circular waiting)
(attack ⇨ impose a total ordering of all resource types, require that in an increasing order of enumeration.)

* Deadlock is prevented by designing the resource management sections of an operating so that one of the conditions cannot occur.

* Degrade utilization of system resource, but is appropriate in systems for which deadlock carries a heavy penalty (ex. In a real-time system).

* **Deadlock Detection**

Design the system assuming that deadlock can occur.
Employ methods to detect deadlocks when they occur.
(and then do recovery)
Can recover by terminating the deadlocked processes or by preempting the resources. (choose which one ?)
Advantage: allow higher resource utilization compared to when deadlock is absolutely prevented.
Should be used when deadlock is not too frequent and recovery is not too expensive.
(the inverse case for deadlock prevention)



* Deadlock Avoidance

* Motivation:
  Deadlock prevention can cause low device utilization & reduced system throughput.
  ⇨Acquire additional info. about how resources are to be requested & have better resource allocation!

Methods that relay on some knowledge of future process behavior to constrain the pattern of resource allocation.
(how resources are to be requested)
Each process declares the maximum number of resources of each type that it may need, (* required *)

Deadlock Avoidance: dynamically examines the resource allocation state to ensure that there can never be a circular-wait condition.
*

* A simple model

  – Processes declare the max # of resources of each type that it may need. (不能>系統有)

  – A deadlock-avoidance algorithm dynamically examines the resource-allocation state and make sure that it is safe! (如果不 safe →只是可能進入 deadlock, 但不一定會有)
    (e.g., never enter a circular-wait condition) ⇨Reject 其 request

  – A resource-allocation state
    < # of available resources,
    # of allocated resources,
    max demands of processes >

* Safe State

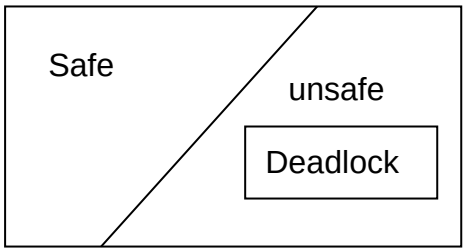
– Motivation: Deadlocks are avoided if the system can allocate resources to each process up to its maximum request in some order. If so, the system is in a safe state! (exist a safe sequence →exist a safe state)

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

$\forall$ *Pi*, need(Pi) $\leq$ Available + Σ allocated(Pj) (j <i)

Avail – Need + Max = Avail – (Max – Allocated) + Max
= Avail + Allocated

⇒ A system state is { safe: if there exists a safe state.
unsafe: otherwise



A deadlock is an unsafe state

– Example: (total: 12 tapes)

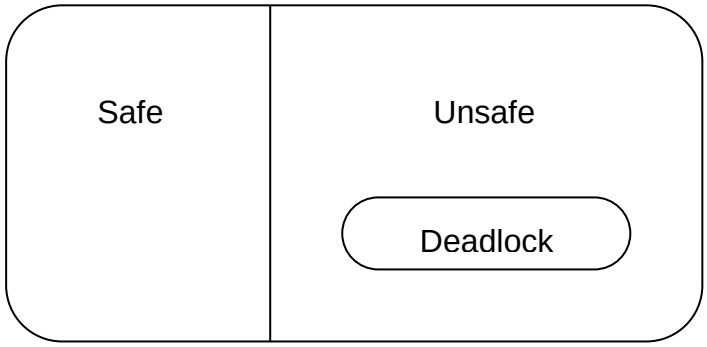
	Max needs	Allocated	Available
P0	10	5	3
P1	4	2	<P1, P0, P2>
P2	9	2	safe sequence

* If P2 got one more, the system state is unsafe. (reject!)
∵((P0, 5), (P1, 2), (P2, 3), (available, 2))

A “**safe**” state: the system can allocate resource to each process (up to its maximum) in some order and still avoid a deadlock (some processes must wait if it cannot be granted).

Safe sequence <P_{i1}, P_{i2}, …, P_{ij}, P_{ik}, …, P_{in}>
P_{iq} ($1 \le q \le n$), Req# (P_{iq}) ≤ Available # + Allocate # (P_{iw}) ($1 \le w \le q$)
(* for single resource type *)

If can find such a safe sequence → safe state.



* **How to ensure that the system will always remain in a safe state? (*The request is granted only if the allocation leaves the system in a safe state.)**

* <u>One instance / Resource type</u> ~ Resource-Allocation Graph

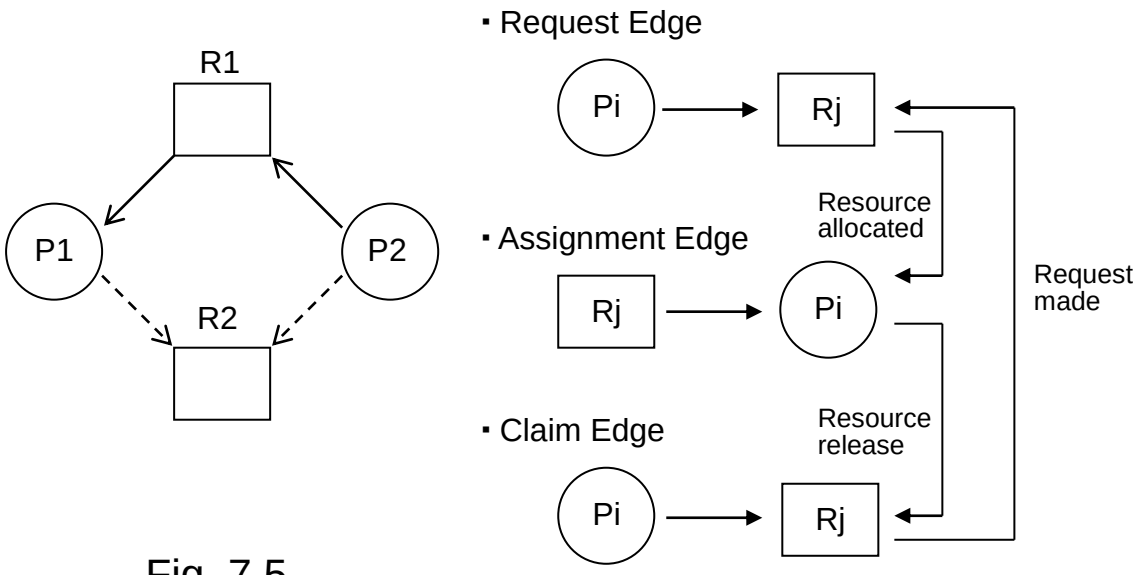
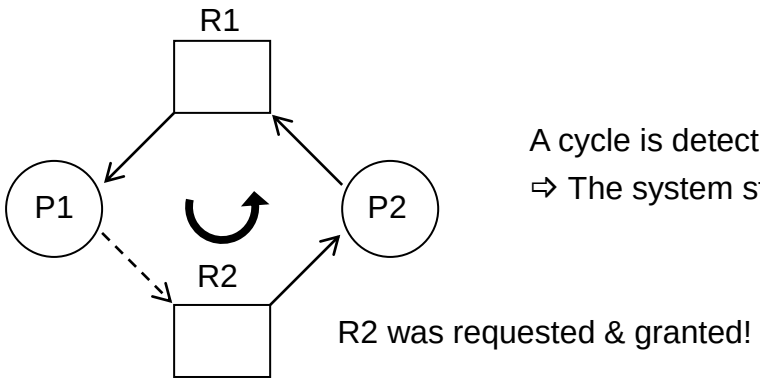


Fig. 7.5

(→, claim edge, 可能會要求)



A cycle is detected!

⇨ The system state is unsafe!

R2 was requested & granted!

Fig. 7.6

{ Safe state: no cycle ; unsafe state: otherwise } Cycle detection can be done in $O(n^2)$

* Multiple instance per resource type

– Banker's algorithm

  * Every new process must declare the maximum usage of each resource type.

  * A resource request is granted if the resource allocation will leave the system in a safe state.

Data structures (n: # of processes, m: # of resources type)

  * Available[m]
    If Available[i]= k, there are k instances of resource type Ri available.

  * Max[n, m]
    If Max[i, j]= k, process Pi may request at most k instances of resource type Rj.

  * Allocation[n. m]
    If Allocation[i, j]= k, process Pi is currently allocated k instances of resource type Rj.

  * Need[n, m]
    If Need[i, j]= k, process Pi may need k more instances of resource type Rj.

  * Need[i, j] = Max[i, j] – Allocation[i, j]

* Deadlock Avoidance: ensure that the system will always be in a safe state.
  Because of waiting → low utilization.

* Several instances of a resource type
  Banker's algorithm

  Declare the maximum number of instances of each resource type that it may need.

  This number can not exceed total number of resources in the system.

  n: # process
  m: # resource types
  $X \leq Y$ iff $X[i] \leq Y[i]$ for all i= 1, 2, …, n
  $(0, 3, 2, 1) \leq (1, 7, 3, 2)$

Available: A vector of length m indicating the number of available resources of each type. If Available[j]= k, there are k instances of resource type R_j available.

* Max: An n × m matrix defining the maximum demand of each process. If Max[i,j]= k, then process P_i may request at most k instances of resource type R_j.

* Allocation: An n × m matrix defining the number of resources of each type currently allocated to each process. If Allocation[i,j]= k, then process P_i is currently allocated k instances of resource type R_j.

* Need: An n × m matrix indicating the remaining resource need of each process. If Need[i,j]= k, then process P_i may need k more instances of resource type R_j in order to complete its task. Note that Need[i,j]= Max[i,j] – Allocation[i,j]

* Banker's algorithm

Let $Request_i$ be the request vector for process Pi. If $Request_i[j] = k$, then process P_i want k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

2. If $Request_i \leq Available$, go to step 3. Otherwise, P_i must wait, since the resources are not available.

3. The system pretends to have allocated the requested resources to process P_i by modifying the state as follows:

$$\begin{cases} Available := Available - Request_i; \\ Allocation_i := Allocation_i + Request_i; \\ Need_i := Need_i - Request_i; \end{cases}$$

   If the resulting resource-allocation state is safe, the transaction is completed and process P_i is allocated its resources. However, if the new state is unsafe, then P_i must wait for $Request_i$ and the old resource-allocation state is restored.

– Safe Algorithm (Safety: O(mxn^2))
/* Find out whether or not a system is in a safe state */

1. Work := Available; Finish[i] := F, $1 \leq i \leq n$

2. Find an i such that both
   a. Finish[i]= F
   b. Need $i \leq$ Work
   If no such i exists, goto stp 4.

3. Work := Work + Allocation i;
   Finish[i] := T
   goto step 2

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

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

(* m resources; n processes *)

* Safety algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize *Work := Available* and *Finish*[i] := *false* for i= 1, 2, …, n.

2. Find an i such that both
   a. *Finish*[i]= *false*
   b. $Need_i \leq Work$
   If no such i exists, go to step 4.

3. *Work := Work + Allocation*$_i$
   Finish[i] := *true*
   go to step 2

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

	Allocation	*Max*	*Available*	(* initial: 10, 5, 7*)
	A B C	A B C	A B C	
P0	0 1 0	7 5 3	3 3 2	
P1	2 0 0	3 2 2		
P2	3 0 2	9 0 2		
P3	2 1 1	2 2 2		
P4	0 0 2	4 3 3		

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

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

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 !! ∵ <P1, P3, P4, P2, P0> is a safe sequence of the state.

Let P1 make a request Request i = (1, 0, 2)

1. Request i ≤ Available ((1, 0, 2) ≤ (3, 3, 2))
2. “New” state

	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 ∵ <P1, P3, P4, P0, P2> is its safe state!
⇒ granted!

* If Request 4 = (3, 3, 0) is asked later, it must be rejected.

* Request 0 = (0, 2, 0) must be rejected because it results in an unsafe state.

* Available: A vector of length m indicating the number of available resources of each type.

* Allocation: An n × m matrix defining the number of resources of each type currently allocated to each process.

* Request: An n × m matrix defining 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.

1. Let *Work* and *Finish* be vector of length m and n, respectively. Initialize *Work* := *Available*. 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:
   a. *Finish*[i]= *false*, and
   b. $Request_i \leq Work$.
   If no such i exists go to step 4.

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 a deadlock state. Moreover, if *Finish*[i]= false then process P_i is deadlocked.

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

	Request
	A B C
P0	0 0 0
P1	2 0 2
P2	0 0 1
P3	1 0 0
P4	0 0 2

* Ex. (A, B, C)= (10, 5, 7) initially
  (sequence <P1, P3, P4, P2, P0> satisfies the safety criteria)

(3 3 2) →P1 (- 1 2 2) (2 1 0) (+ 3 2 2)
(5 3 2) →P3 (- 0 1 1) (5 2 1) (+ 2 2 2)
(7 4 3) →P4 (- 4 3 1) (3 1 2) (+ 4 3 3)
(7 4 5) →P2 (- 6 0 0) (1 4 5) (+ 9 0 2)
(10 4 7) →P0 (-7 4 3) (3 0 4) (+ 7 5 3)
(10 5 7)

However, if P1 requests one more A and two more C,
Request1= (1, ,0, 2) →
find safe sequence <1, 3, 4, 0, 2> → safe, OK, grant resource to P1.

Next, if P4 ask for (3, 3, 0) → NO! (> available).
However, if P0 asks for (0, 2, 0) → NO → unsafe state.
(now available (2, 1, 0))

Performance: $O(mn^2)$

– Resource Allocation Algorithm
/* Request i is the request vector for Pi Request[i] = k :
Pi request k instances of resource type Rj */

1. If Request i ≤ Need i, goto step2, otherwise, reject.

2. If Request i ≤ Available, goto step3, otherwise, Pi must wait.

3. Let the system pretend to have allocated resources to process Pi by setting .

   Available := Available – Request i;
   Allocation i := Allocation i + Request i;
   Need i := Need i – Request i;

   Exec “Safety Algorithm”. If the system state is safe, the request is granted; otherwise, Pi must wait; and the old resource-allocation state is restored!

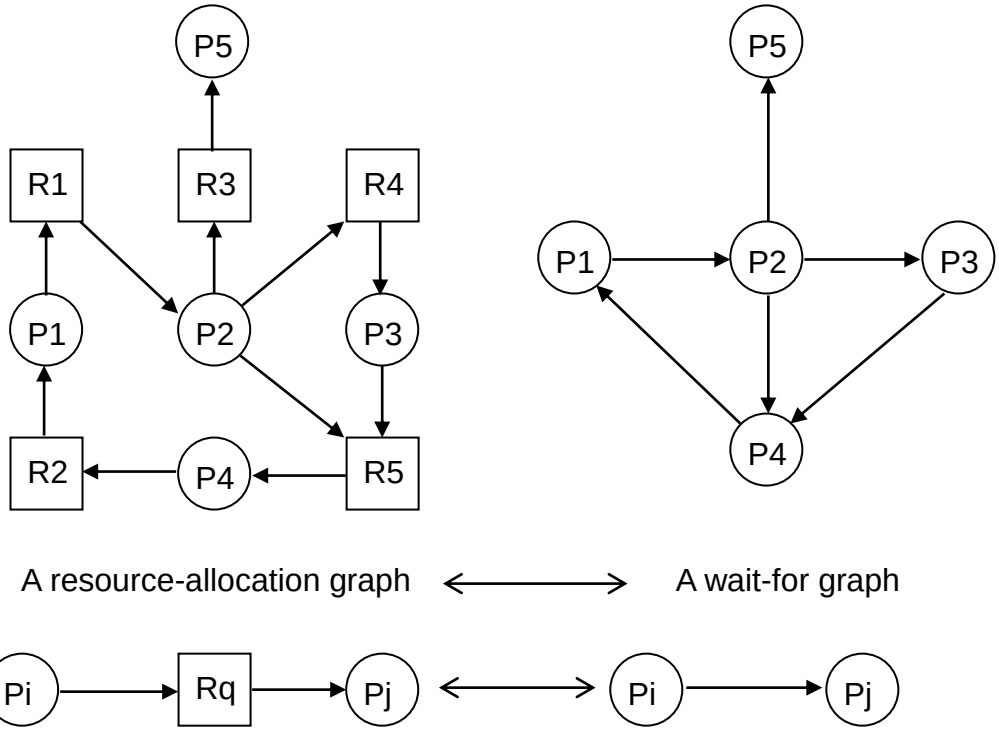
* **Deadlock Detection**

* Motivation: Higher resource utilization and “may be” lower possibility of deadlock occurrence.

* Overhead:
  - Cost of info maintenance.
  - Cost of executing detection algorithm.
  - Potential losses inherent from a deadlock.

* Deadlock Detection

  – Single instance per resource type



  * Detect an cycle in $O(n^2)$ every t time unit.

  * The system needs to maintain the wait-for-graph.

* Deadlock Detection

* Single instance of each resource type.

  A variant of resource graph → wait-for graph.
  (By removing the nodes of type resource and collapsing the appropriate edges).

  A deadlock exists in the system if and only if the wait-for graph contains a cycle.

  To detect deadlock, the system maintains the wait-for graph, and periodically to invoke an cycle-detection algorithm.

  Performance: ($O(n^2)$).

* Detection algorithm usage:
  When should we invoke the detection algorithm?
  Consider:

  1. How often is the deadlock?
     (Deadlock can come only when some process makes a request that cannot be granted immediately.)

  2. How many processes will be affected by deadlock when it happens?

– Multiple instance per resource type

Data Structure
Available[1..m]: # of available resource instance
Allocation[1..n, 1..m]: resource allocation
Request[1..n, 1..m]: the current request of 1
If Request[i,j]= k,Pi requests k more instance of resource type Rj.

1. Work := Available, & for i=1, 2, …, 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] := true
   goto step 2

4. If Finish[i]= F, for some i, then the system is in a deadlock state. If Finish[i]= F, process Pi is deadlock.

Example

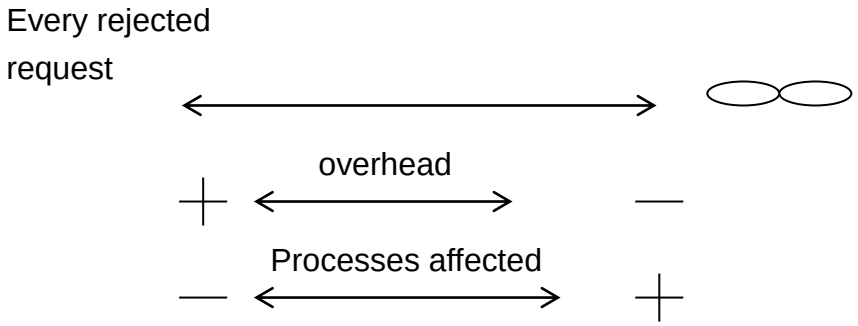
	Allocation	Request	Available
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

⇨ Find a sequence <P0, P2, P3, P1, P4> such that Finish[i]= T for all i.

* How about Request**2**= (0, 0 ,1)
  ⇨ P1, P2, P3, and P4 are deadlocked!

But, when should we invoke the detection algorithm?

– How often



– How many processes will be affected by deadlock?

* Deadlock Detection? When? CPU utilization us a threshold? A detection frequency? …

* Recovery

* Whose responsibility to deal with deadlocks?
  - Operator deals with the deadlock manually.
  - The system recover from the deadlock automatically.
* Solutions
  - Process Termination
    * Abort all deadlocked processes!
      Simple but costly
    * Abort one process at a time until the deadlock cycle is broken!
      Overhead for running the detection again and again, and the difficulty of selecting a victim!

      What is the cost of aborting a victim?
      - Process priority.
      - The CPU time consumed and to be consumed by a process.
      - The (# of) resources used and needed by a process.
      - Process's characteristics such as "interactive or batch".
      - # of processes needed to be aborted.

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

- Resource Preemption
  Preemption some resources from processes and give them to other processes until 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 efficient?

* Recovery

Rollback – abort – restart.
Minimizes the cost of re-executing the rolled-back processes: priority, type and amount of current resource allocation.

1. Abort one or more processes in order to break the circular wait.
   Process termination.
   (Abort it, reclaim all resources allocated to it.)

   (a) Abort all deadlocked processes.

   (b) Abort one process at a time until the deadlock cycle is eliminated.

   Terminate a process may leave a file in an incorrect state. Which process should be terminated? (minimum cost)
   Consider (priority, how long the process has computed, how much longer the process will compute before completing its designated task, how many and what type of resources the process has used, how many more resources the process needs, how many processes will need to be terminated…)

2. Preempt some resources from one or more of the deadlocked processes.

   Preempt some resources from processes and give these resources to other process until the deadlock is broken.
   Select a victim (which resources and which processes are to be preempted?)
   Rollback (when the resource is preempted, what should be done with that process?)
   (total rollback, partial rollback)
   Starvation!! (Avoid affect the same process again!)