Chapter 2: Process Synchronization

2.5: Deadlocks

GV: Nguyễn Thị Thanh Vân

Outline

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

Chapter Objectives

- Illustrate how deadlock can occur when mutex locks are used
- Define the four necessary conditions that characterize deadlock
- Identify a deadlock situation in a resource allocation graph
- Evaluate the four different approaches for preventing deadlocks
- Apply the banker's algorithm for deadlock avoidance
- Apply the deadlock detection algorithm
- Evaluate approaches for recovering from deadlock

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$
  - *CPU cycles, memory space, I/O devices*
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **Request:** The thread requests the resource
  - **Use:** The thread can operate on the resource
  - **Release:** The thread releases the resource**.**

Deadlock Example with Semaphores

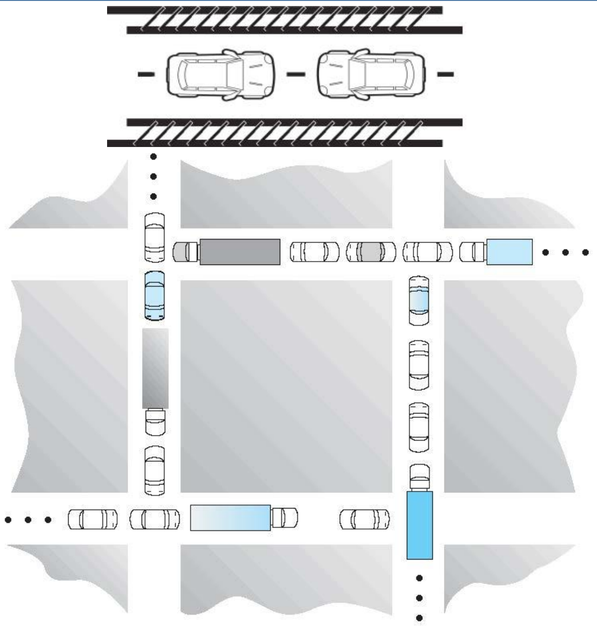
- Data:
  - A semaphore `S1` initialized to 1
  - A semaphore `S2` initialized to 1
- Two processes P1 and P2
- **P1:**

```
  wait(s1)
  wait(s2)
```

- **P2:**

```
  wait(s2)
  wait(s1)
```

Deadlock Example on a one-way Bridge



Deadlock Characterization

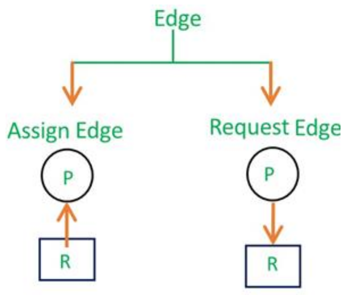
Deadlock can arise if four conditions hold simultaneously.

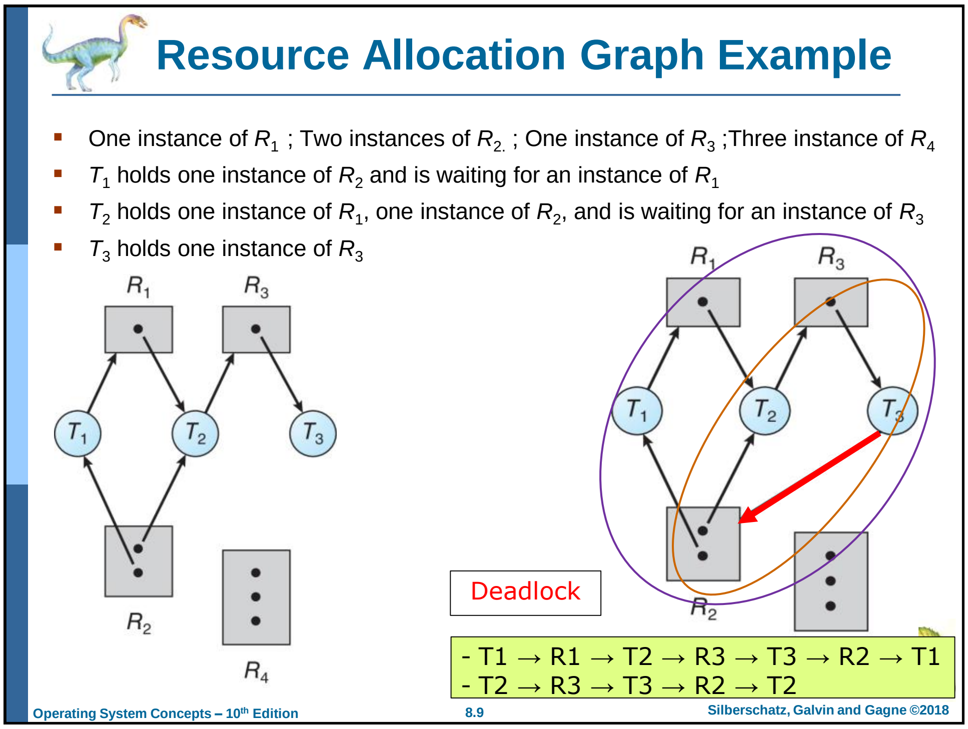
- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **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.

Resource-Allocation Graph

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

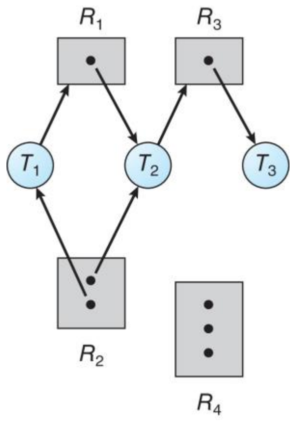
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

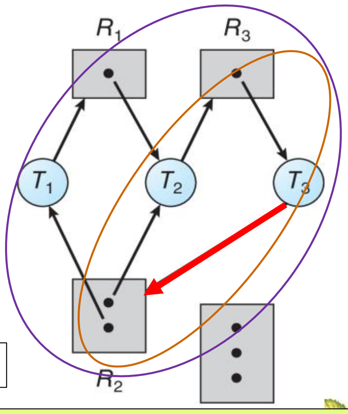




Resource Allocation Graph Example

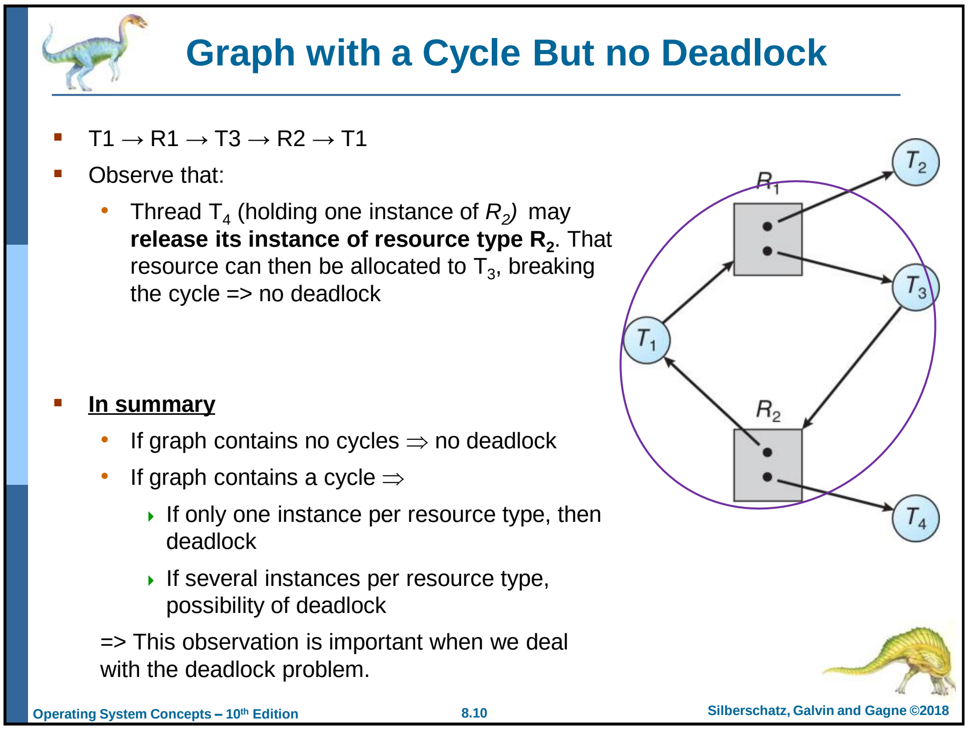
- One instance of R_1 ; Two instances of $R_{2.}$; One instance of R_3 ;Three instance of R_4
- T_1 holds one instance of R_2 and is waiting for an instance of R_1
- T_2 holds one instance of R_1, one instance of R_2, and is waiting for an instance of R_3
- T_3 holds one instance of R_3





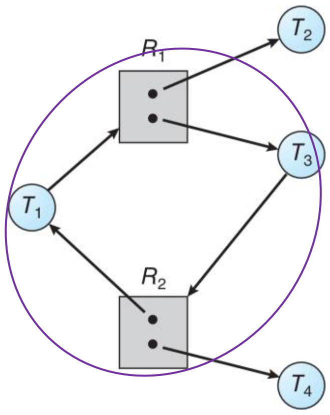
Deadlock

- T1 → R1 → T2 → R3 → T3 → R2 → T1
- T2 → R3 → T3 → R2 → T2



Graph with a Cycle But no Deadlock

- T1 → R1 → T3 → R2 → T1
- Observe that:
  - Thread T_4 (holding one instance of R_2) may **release its instance of resource type R_2**. That resource can then be allocated to T_3, breaking the cycle => no deadlock



- **<u>In summary</u>**
  - If graph contains no cycles ⇒ no deadlock
  - If graph contains a cycle ⇒
    - If only one instance per resource type, then deadlock
    - If several instances per resource type, possibility of deadlock

=> This observation is important when we deal with the deadlock problem.

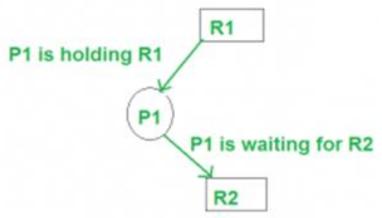
Methods for Handling Deadlocks

- Ensure that the system will **never** enter a deadlock state:
  - Deadlock prevention
    - ***Mutual Exclusion:*** One or more than one resource are non-shareable (Only one process can use at a time)
    - ***Hold and Wait:*** A process is holding at least one resource and waiting for resources.
    - ***No Preemption:*** A resource cannot be taken from a process unless the process releases the resource.
    - ***Circular Wait:*** A set of processes are waiting for each other in circular form.
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system.
  - What is the consequence?

Deadlock Prevention

Invalidate one of the four necessary conditions for deadlock:

- **Mutual Exclusion** –
  - sharable resources (e.g., read-only files): not required
  - non-sharable resources (e.g: printer): do not
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require processes to request and be allocated all its resources before it begins execution, or
  - Allow processes to request resources only when the process has none allocated to it.
  - Limit: Low resource utilization; starvation possible



Deadlock Prevention (Cont.)

- **No Preemption**:
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all the resources currently being held by the process are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be **restarted** only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait:**
  - Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

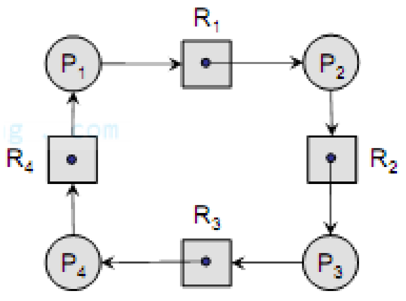
Circular Wait

- Cách 1: mỗi process yêu cầu thực thể của tài nguyên theo thứ tự tăng dần (định nghĩa bởi hàm F) của loại tài nguyên. Ví dụ
  - Chuỗi yêu cầu thực thể hợp lệ: tape drive -> disk drive -> printer
  - Chuỗi yêu cầu thực thể không hợp lệ: disk drive -> tape drive
- Cách 2: Khi một process yêu cầu một thực thể của loại tài nguyên Rj thì nó phải trả lại các tài nguyên Ri với F(Ri) > F(Rj).
- Ví dụ

"Chứng minh" cho cách 1: phản chứng

- P1: F(R4) < F(R1)
- P2: F(R1) < F(R2)
- P3: F(R2) < F(R3)
- P4: F(R3) < F(R4)

- Vậy F(R4) < F(R4), mâu thuẫn!

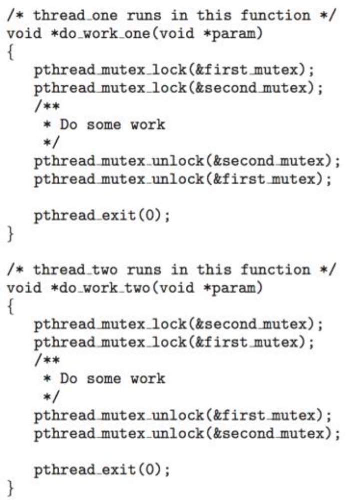


Circular Wait

- Invalidating the circular wait condition is most commonly used.
- Simply assign each resource (i.e., mutex locks) a unique number.
- Resources must be acquired in order.
- Example: two mutex locks

```
first_mutex = 1
second_mutex = 5
```

code for **`thread_two`** could not be written as follows:



```
/* 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 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 sequence

  $P_1, P_2, \ldots, P_n$

  of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < I$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Safe State ex

- Ví dụ: Hệ thống có **12** tape drive và 3 quá trình P0, P1, P2
- Tại thời điểm t0, dữ liệu cho như bảng,
  - => hệ thống còn 3 tape drive sẵn sàng (12-5-2-2).
  - Sequence ⟨P1, P0, P2 ⟩: safe:
    - P1: get and return => avail: 3+2=5
    - P0: get and return => avail: 5+5=10
    - P2: get and return =>avail: 10+2=**12**

  ⇒ system is in a safe state

Process	Max - need	Alloca-tion
P0	10	5
P1	4	2
P2	9	2

- Tại thời điểm t1, P2 yêu cầu và được cấp phát 1 tape drive còn 2 tape drive sẵn sang
    - P1: get and return => avail: 2+2=4
    - P0: get 5: wait
    - P2: get

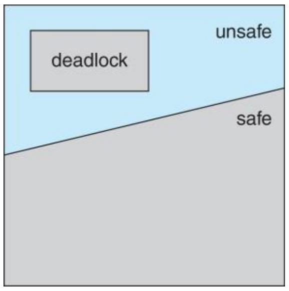
  ⇒ Sequence ⟨P1, P0, P2 ⟩: no safe

  ⇒ system is in a no safe state

Process	Max - need	Alloca-tion
P0	10	5
P1	4	2
P2	9	**3**

Safe, Unsafe, Deadlock State

- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance algorithm:
  - Ensure that a system will never enter an unsafe state



Avoidance Algorithms

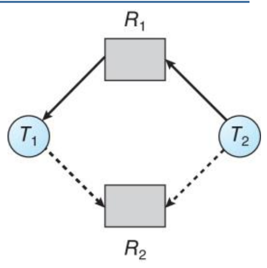
- Single instance of a resource type
  - Use a modified resource-allocation graph
- Multiple instances of a resource type
  - Use the Banker's Algorithm

Modified Resource-Allocation Graph Scheme

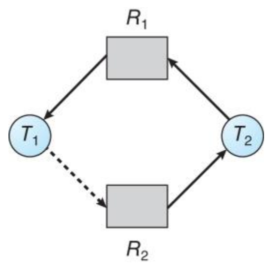
- **Claim edge** P_i ----> R_j indicates that process P_i may request resource R_j
- **Request edge** $P_i \rightarrow R_j$ indicates that process P_i requests resource R_j
  - Claim edge converts to request edge when a process requests a resource
- **Assignment edge** $R_j \rightarrow P_i$ indicates that resource R_j was allocated to process P_i
  - Request edge converts to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed ***a priori*** in the system

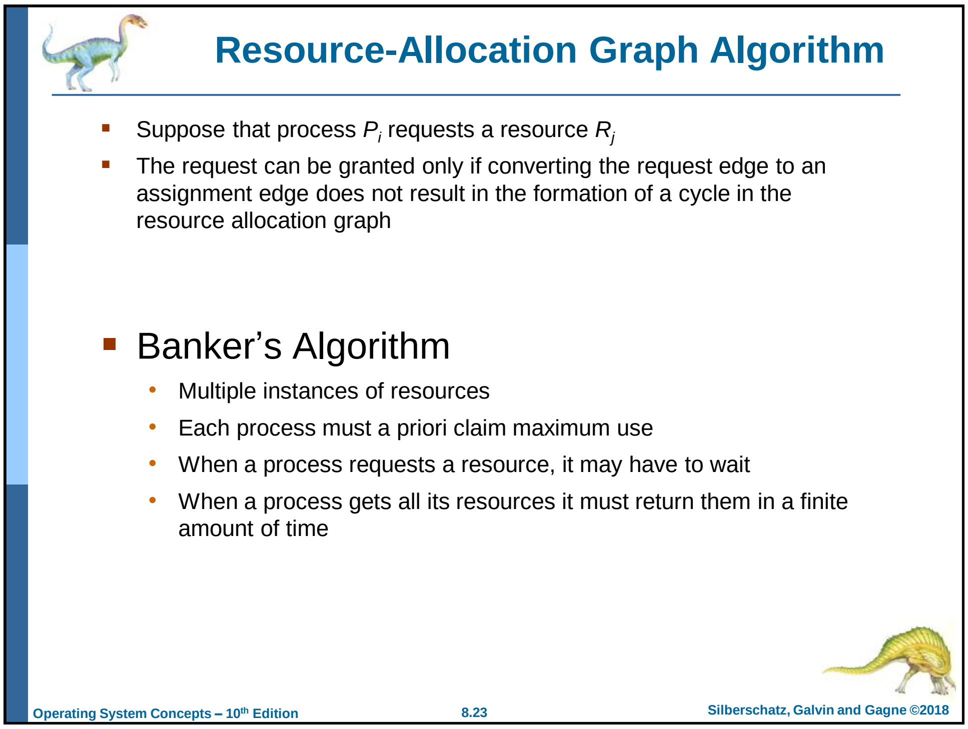
Resource-Allocation Graph

- Modified Resource-Allocation Graph Example



- Unsafe State In Resource-Allocation Graph

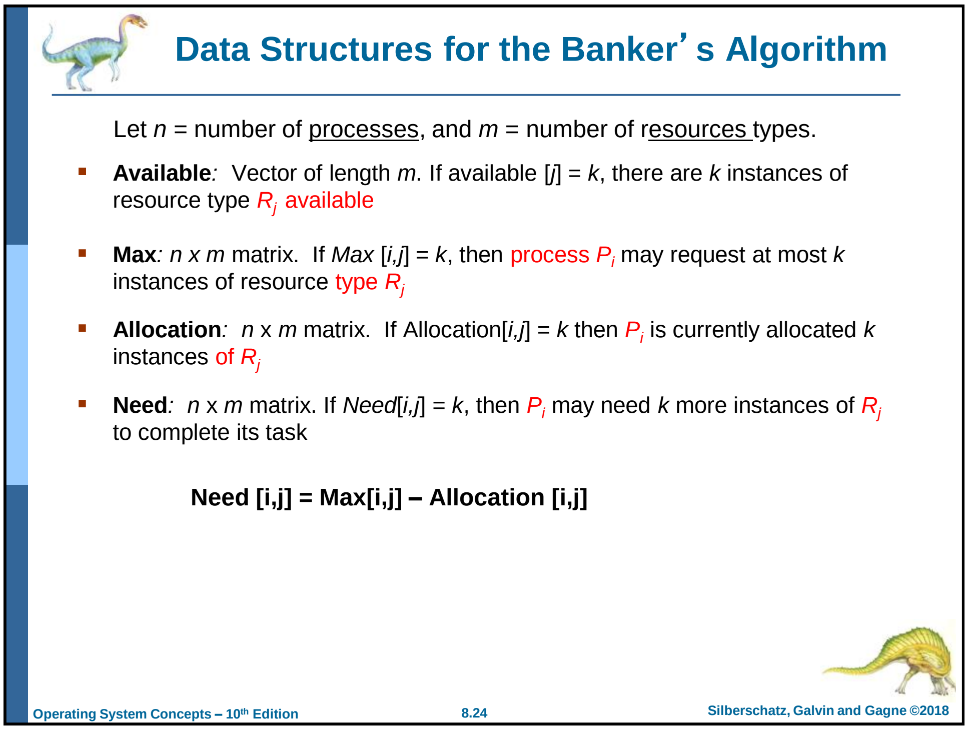




Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

- Banker's Algorithm
  - Multiple instances of resources
  - 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
- **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
- **Allocation**: *n* x *m* matrix. If Allocation[*i,j*] = *k* then P_i is currently allocated *k* instances of R_j
- **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

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

Safety Algorithm

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

   Work = Available

   ***Finish* [*i*] = *false* for *i* = 0, 1, …, *n*- 1**

2. Find an ***i*** such that both:

   (a) ***Finish* [*i*] = *false***

   (b) ***$Need_i$ ≤ Work***

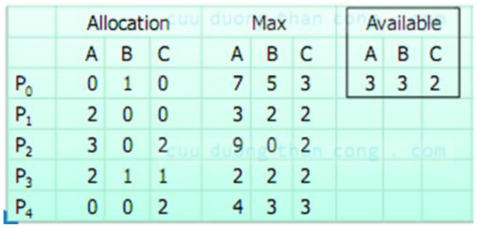
   If no such ***i*** exists, go to step 4

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

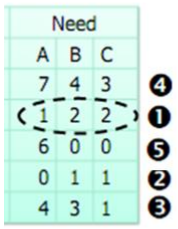
4. If ***Finish* [*i*] == *true*** for all ***i***, then the system is in a safe state

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			



Need			
A	B	C	
7	4	3	❹
1	2	2	❶
6	0	0	❺
0	1	1	❷
4	3	1	❸

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

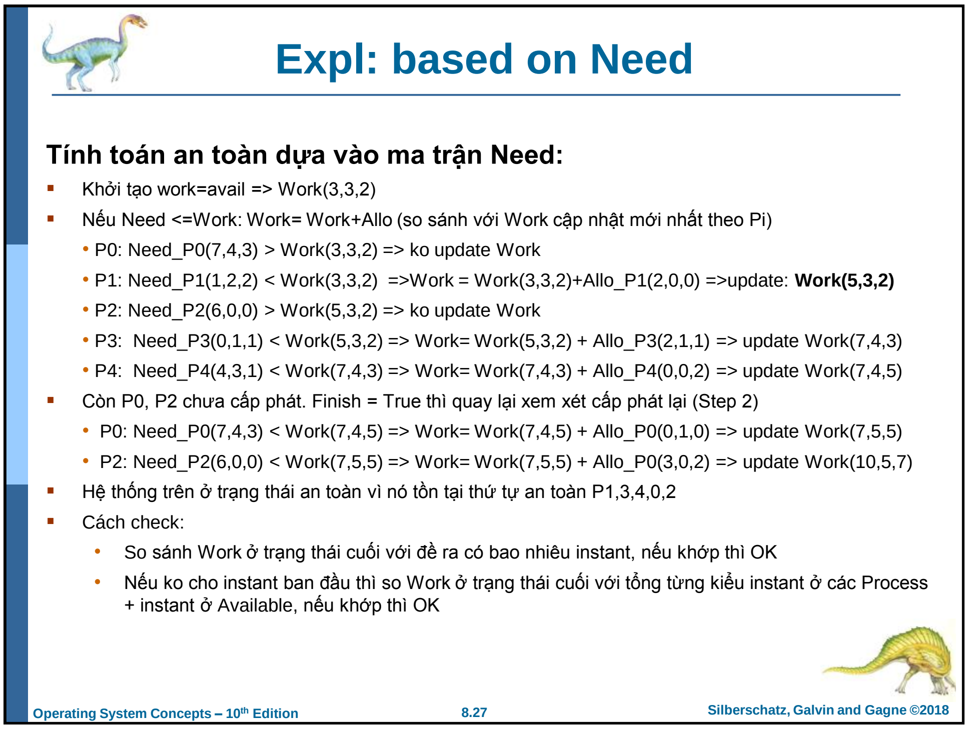
	Allocation	Need
	A B C	*A B C*
P_0	0 1 0	7 4 3
P_1	2 0 0	1 2 2
P_2	3 0 2	6 0 0
P_3	2 1 1	0 1 1
P_4	0 0 2	4 3 1

Work

A B C

$$\begin{array}{ccc} 3 & 3 & 2 \\ & \downarrow & \\ 5 & 3 & 2 \\ & \downarrow & \\ 7 & 4 & 3 \\ & \downarrow & \\ 7 & 4 & 5 \\ & \downarrow & \\ 10 & 4 & 7 \end{array} \longrightarrow 10 \quad 5 \quad 7$$

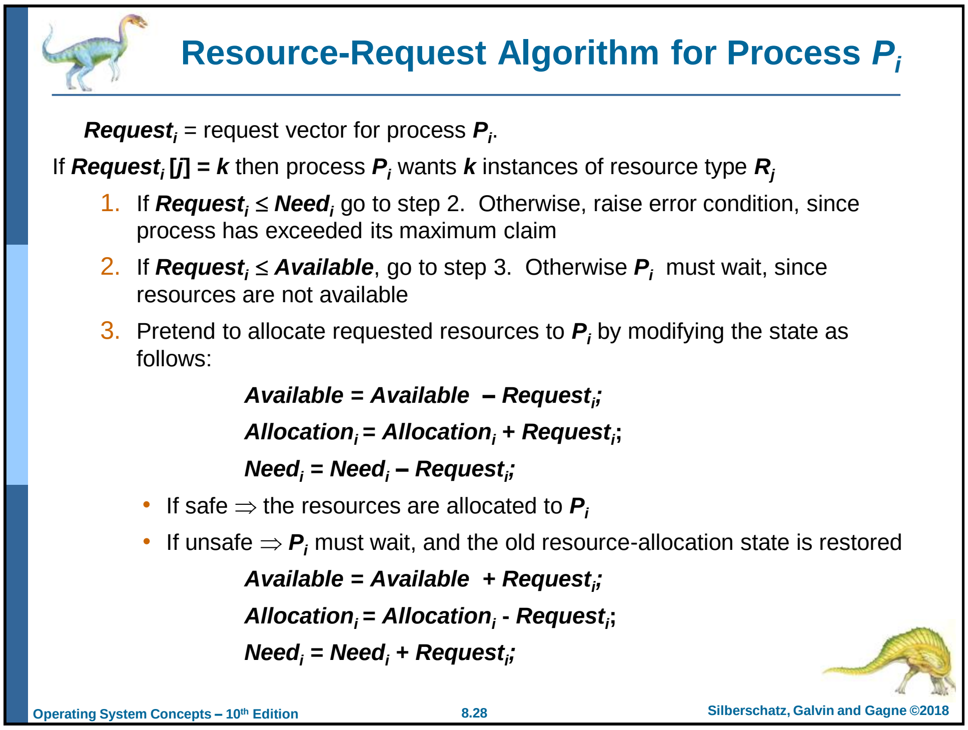
The system is in a safe state since the sequence < P_1, P_3, P_4, P_0, P_2 > satisfies safety criteria



Expl: based on Need

Tính toán an toàn dựa vào ma trận Need:

- Khởi tạo work=avail => Work(3,3,2)
- Nếu Need <=Work: Work= Work+Allo (so sánh với Work cập nhật mới nhất theo Pi)
  - P0: Need_P0(7,4,3) > Work(3,3,2) => ko update Work
  - P1: Need_P1(1,2,2) < Work(3,3,2) =>Work = Work(3,3,2)+Allo_P1(2,0,0) =>update: **Work(5,3,2)**
  - P2: Need_P2(6,0,0) > Work(5,3,2) => ko update Work
  - P3: Need_P3(0,1,1) < Work(5,3,2) => Work= Work(5,3,2) + Allo_P3(2,1,1) => update Work(7,4,3)
  - P4: Need_P4(4,3,1) < Work(7,4,3) => Work= Work(7,4,3) + Allo_P4(0,0,2) => update Work(7,4,5)
- Còn P0, P2 chưa cấp phát. Finish = True thì quay lại xem xét cấp phát lại (Step 2)
  - P0: Need_P0(7,4,3) < Work(7,4,5) => Work= Work(7,4,5) + Allo_P0(0,1,0) => update Work(7,5,5)
  - P2: Need_P2(6,0,0) < Work(7,5,5) => Work= Work(7,5,5) + Allo_P0(3,0,2) => update Work(10,5,7)
- Hệ thống trên ở trạng thái an toàn vì nó tồn tại thứ tự an toàn P1,3,4,0,2
- Cách check:
  - So sánh Work ở trạng thái cuối với đề ra có bao nhiêu instant, nếu khớp thì OK
  - Nếu ko cho instant ban đầu thì so Work ở trạng thái cuối với tổng từng kiểu instant ở các Process + instant ở Available, nếu khớp thì OK



Resource-Request Algorithm for Process P_i

Request$_i$ = 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 P_i
   - If unsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is restored

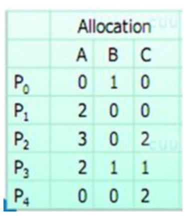
   $Available = Available + Request_i;$

   $Allocation_i = Allocation_i - Request_i;$

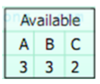
   $Need_i = Need_i + Request_i;$

Example: P_1 Request (1,0,2)

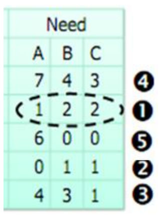
- P_1 Request (1,0,2): Check that Request $\leq$ Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ true
- **Before:**



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



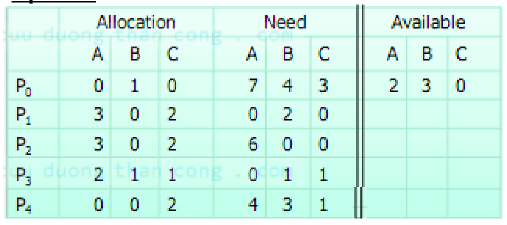
Available		
A	B	C
3	3	2



Need			
A	B	C	
7	4	3	❹
1	2	2	❶
6	0	0	❺
0	1	1	❷
4	3	1	❸

Avail = Avail – Request_i;
Alloc_i = Alloc_i + Request_i;
Need_i = Need_i – Request_i;

- **Update:**



	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	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

Work:

Pro_cess	A	B	C
	2	3	0
P1	5	3	2
P3	7	4	3
P4	7	4	5
P0	7	5	5
P2	10	5	7

- Sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement
  - => R1 can be allocated resource

Example:

- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

Deadlock Detection

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

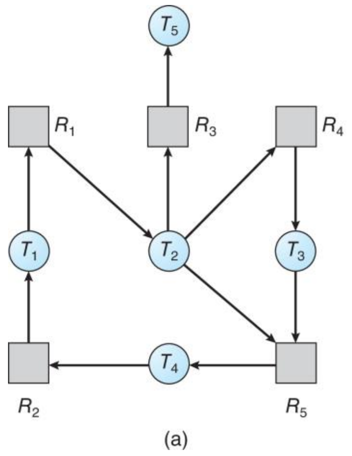
Single Instance of Each Resource Type

- a deadlock detection algorithm that uses a variant of the resource-allocation graph, called **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j to release a resource P_i needs

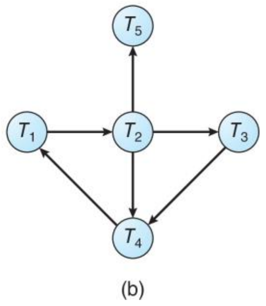
We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges

- To detect deadlocks:
  - Maintain the wait-for graph
  - Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- 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

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* [*i*][*j*] = *k***, then process $\boldsymbol{P_i}$ is requesting ***k*** more instances of resource type $\boldsymbol{R_j}$.

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 $\boldsymbol{Allocation_i \neq 0}$, then
      Finish*[i] = *false; otherwise, ***Finish*[i] = *true***

2. Find an index ***i*** such that both:
   a) ***Finish*[*i*] == *false***
   b) $\boldsymbol{Request_i \leq Work}$

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

3. $\boldsymbol{Work = Work + Allocation_i}$
   Finish*[*i*] = *true
   go to step 2

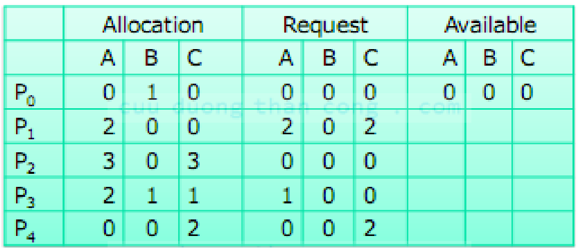
4. If ***Finish[i] == false***, for some ***i***, $1 \leq \boldsymbol{i} \leq \boldsymbol{n}$, then the system is in deadlock state. Moreover, if ***Finish*[*i*] == *false***, then $\boldsymbol{P_i}$ is deadlocked

   If ***Finish[i] == true,*** then the system is in safe

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

Example of Detection Algorithm

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



	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			

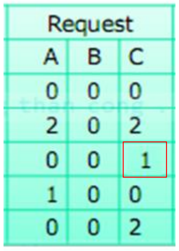
- Sequence <$\boldsymbol{P_0, P_2, P_3, P_4, P_1}$> or
- Sequence <$\boldsymbol{P_0, P_2, P_3, P_1, P_4}$>
- will result in ***Finish[i] = true*** for all ***I***

=> The system is in safe

Pro_ cess	A	B	C
	0	0	0
P0	0	1	0
P2	3	1	3
P3	5	2	4
P4	5	2	6
P1	7	2	6

Example (Cont.)

- P_2 requests an additional instance of type C



Request		
A	B	C
0	0	0
2	0	2
0	0	1
1	0	0
0	0	2

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - One for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

When Deadlock

- When a detection algorithm determines that a deadlock exists, several alternatives are available.
  - One possibility is to inform the operator
  - Another possibility is to let the system recover from the deadlock automatically.

- recover from breaking a deadlock
  - Abort all deadlocked processes
  - Abort one process at a time until the deadlock cycle is eliminated

Recovery from Deadlock: Process Termination

- 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

End of Chapter 2