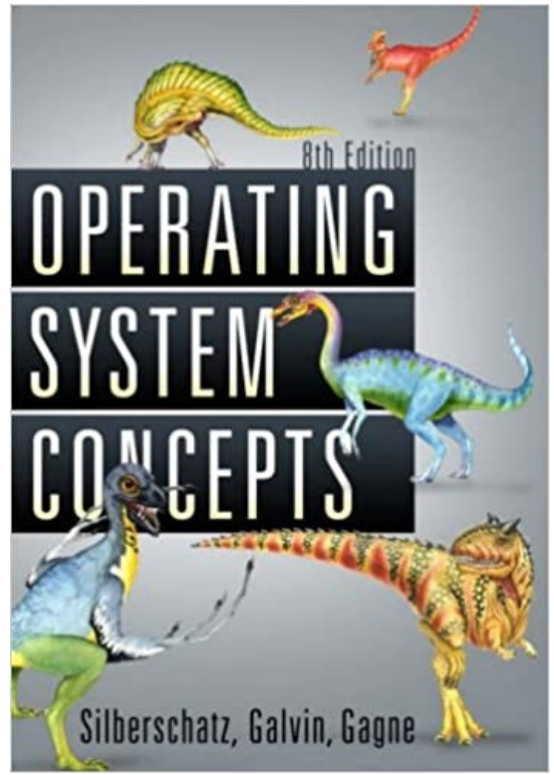
Chapter 8: Deadlocks

Dept. of Software, Gachon Univ.
Joon Yoo



Gachon University
School of Computing

Chapter Objectives

- To learn the concept of **deadlock**
- To develop a **characterization of deadlocks**, which prevent sets of concurrent processes (threads) from completing their tasks
- To present a number of different methods for **preventing** or **avoiding** deadlocks in a computer system

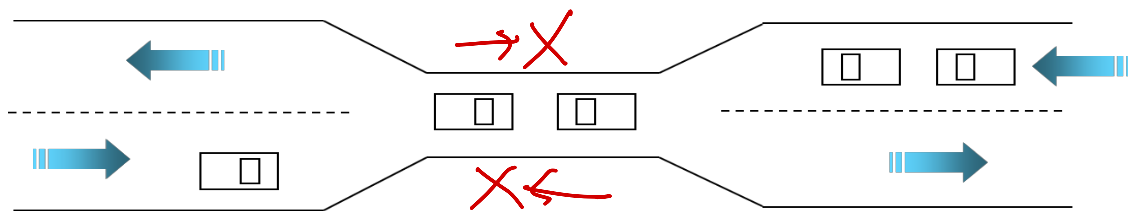
Chapter 8: Deadlocks

- Deadlock Concept
- System Model: Resource Allocation Graph
- Deadlock Prevention
- Deadlock Avoidance

Example 1: Bridge Crossing

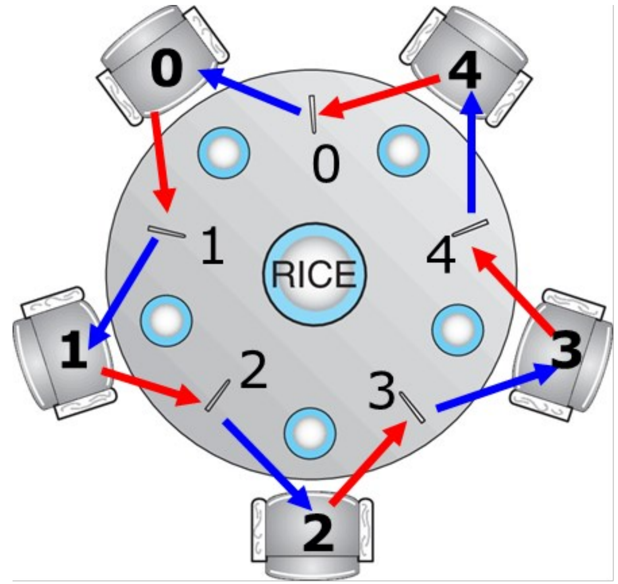
- One-lane Bridge Example
  - Traffic only in one direction.
  - Each section of a bridge can be viewed as a resource.

https://commons.wikimedia.org/wiki/File:One_lane_bridge_over_West_River_on_Rice_Farm_Road.JPG



Example 2: Dining-Philosophers Problem (Ch.7)

- Deadlock
  - All five philosophers become hungry at the same
  - Each philosopher grabs the left chopstick – all chopstick semaphore will become 0
  - Each philosopher tries to grab her right chopstick, she will be delayed forever! – **deadlock**



Example 3: Semaphores (Ch. 6)

- A set of blocked processes each holding a shared resource and waiting to acquire a resource held by another process in the set.

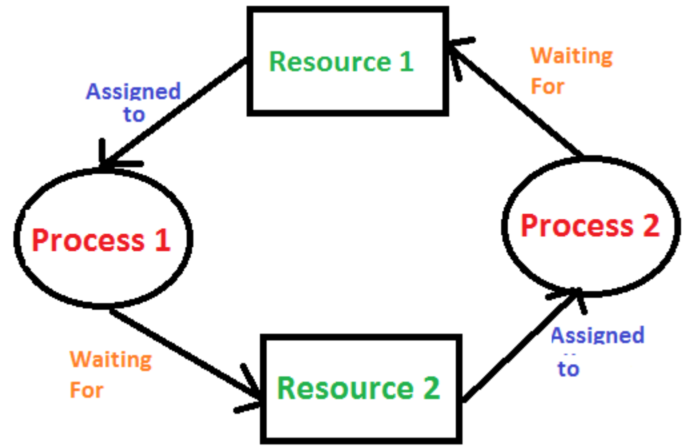
Example 1: semaphores *R1* and *R2*, initialized to 1

	Process 1		*Process 2*
A1	wait (R1);	B1	wait(R2);
A2	wait (R2);	B2	wait(R1);

- Execution order : A1→B1→B2→A2

서로가 서로를 기다리는 ....

= Deadlock!



Example 4: Deadlock in Multithreaded Application

- Two mutex locks are created an initialized:

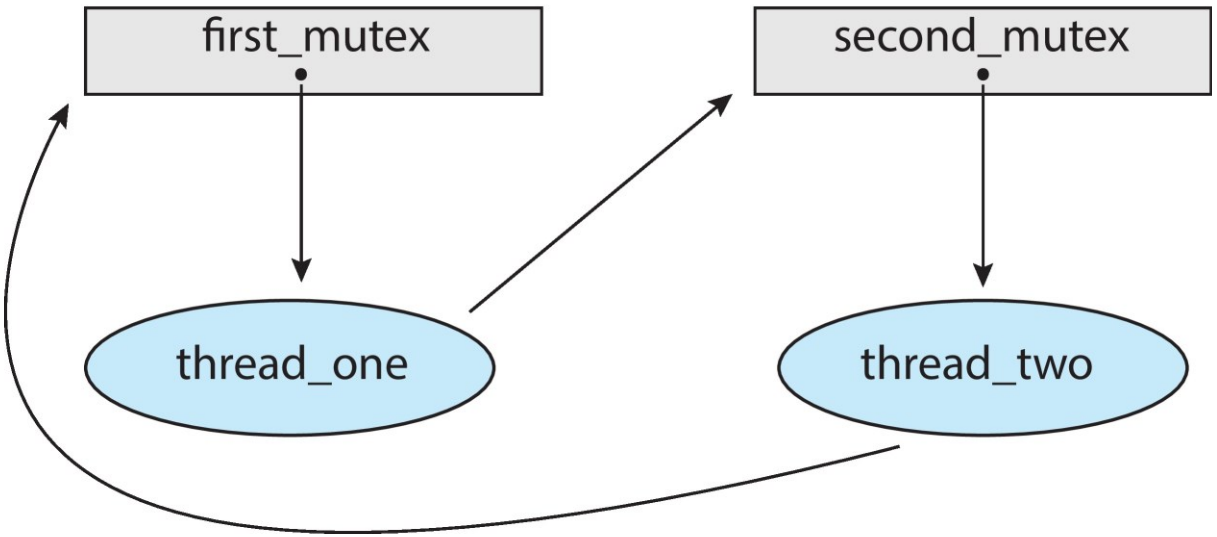
```
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex,NULL);
pthread_mutex_init(&second_mutex,NULL);
```

1 2 3 4 Deadlock!

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);   1
   pthread_mutex_lock(&second_mutex);  4
   /**
    * 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);  2
   pthread_mutex_lock(&first_mutex);   3
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

Deadlock in Multithreaded Application

- Deadlock is possible if thread 1 acquires **`first_mutex`** and thread 2 acquires **`second_mutex`**. Thread 1 then waits for **`second_mutex`** and thread 2 waits for **`first_mutex`**.
- Can be illustrated with a **resource allocation graph**:



Chapter 8: Deadlocks

- Deadlock Concept
- System Model: Resource Allocation Graph
- Deadlock Prevention
- Deadlock Avoidance

System Model: Resource Type

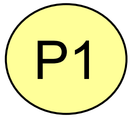
- System consists of ***finite*** number of **resource types**: $R_1, R_2, \ldots, R_m$
  - **Physical resource types**: I/O devices (e.g., printers), memory space, CPU cycles
  - **Logical resource types**: semaphores, mutex locks, and files
- Each resource type R_i can have W_i **identical instances**
  - A system may have four identical printers: Printer resource type has four instances

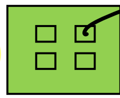
resource type that has ~ instances

모든 instance는 동일하다.

- A process must **request an instance** of a **resource type** before **using** it
  - Any instance of a resource type should satisfy the request
- The process must **release** the instance of a resource type after using it

Resource-Allocation Graph

- Graph model
  - A directed graph: sets of nodes (vertices) *V* and edges *E*
  - Nodes *V*: Process node(P_i) ∪ resource node (R_j)
    - **Process** : 

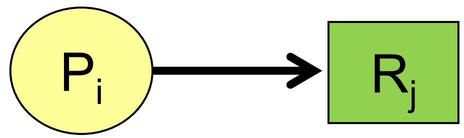
    - **Resource type** with 4 **instances** : 

각 instance는 정확히 같다!

  - Edges *E*
    - **Request edge** (P_i → R_j)

      The process P_i requested (is waiting) for the resource R_j

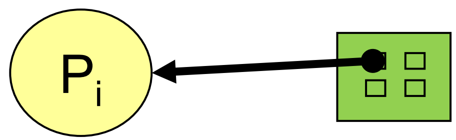
Rj라는 리소스를 얻기위해 요청.



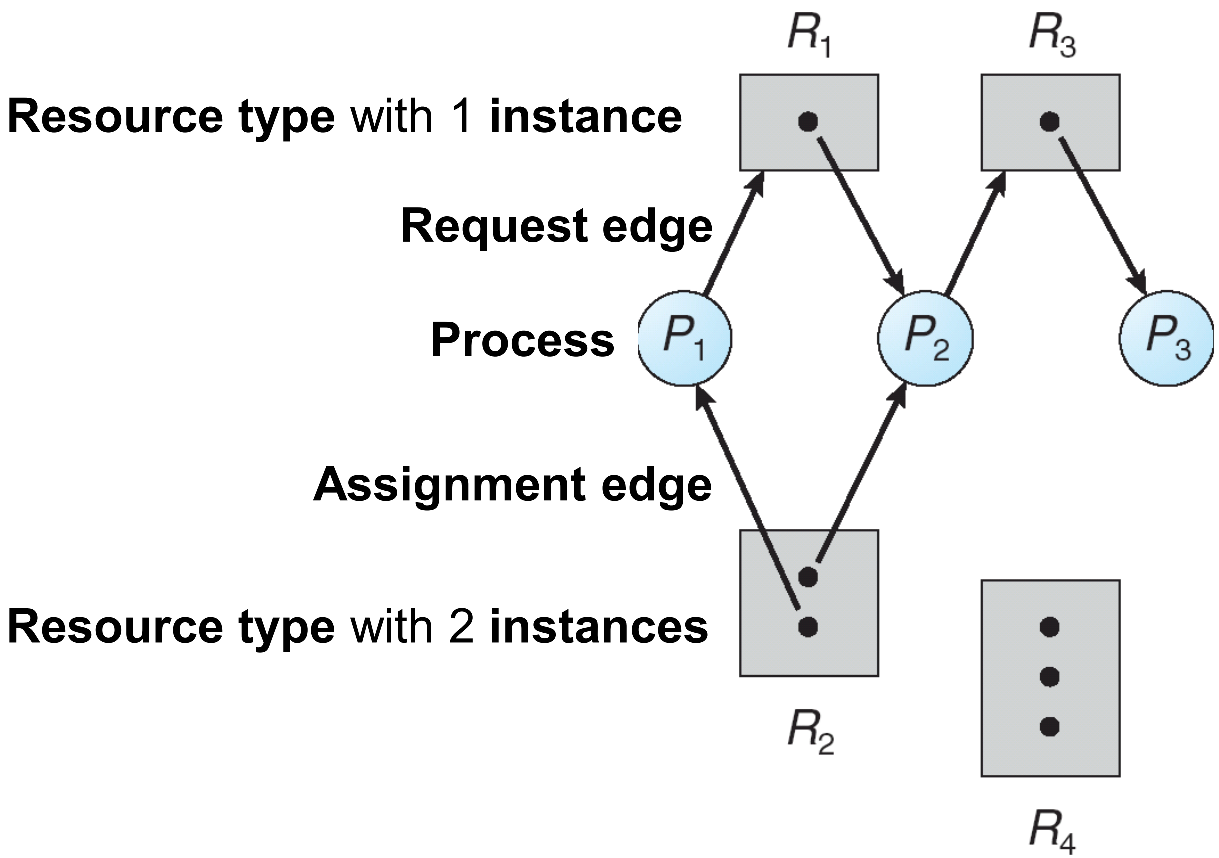
    - **Assignment edge** (R_j → P_i)

      : P_i is holding an instance of R_j

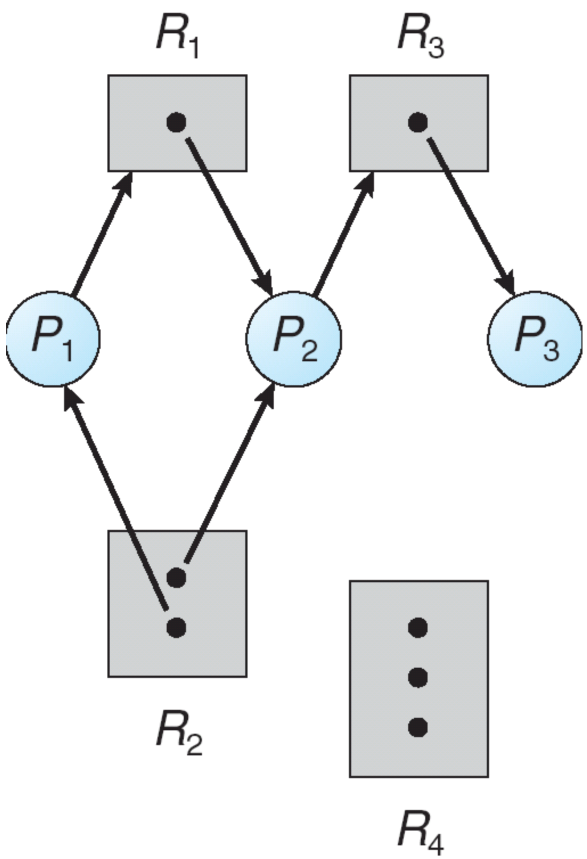
요청한 리소스를 주기 위해 리소스의 특정한 인스턴스 하나를 Pi에게 주게된다.



Example of a Resource Allocation Graph

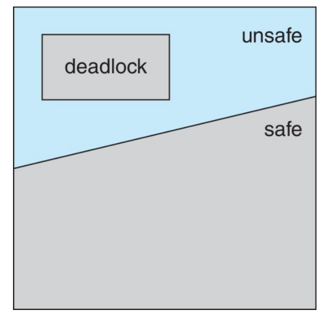


Example of a Resource Allocation Graph



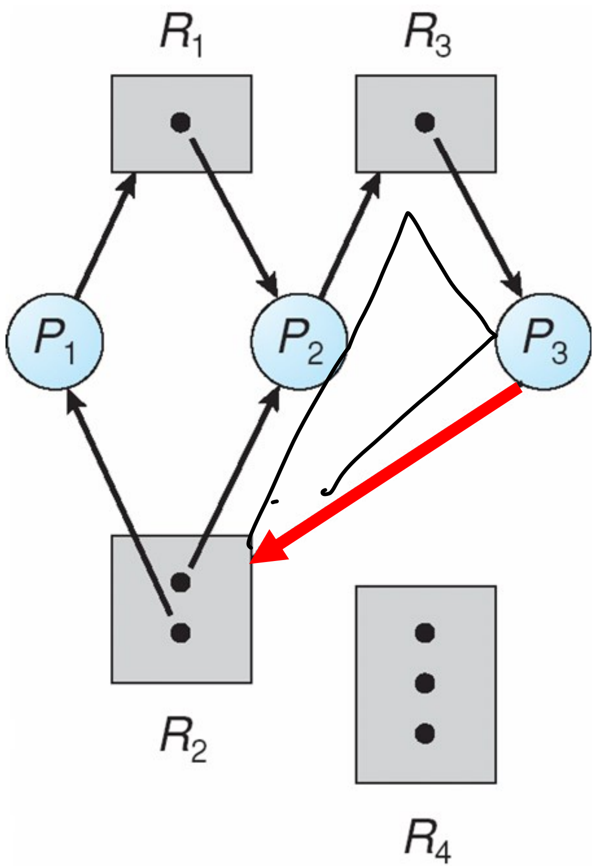
- If graph contains *no* cycles
  $\Rightarrow$ *not* deadlock (**safe**)
- If graph contains a cycle (**unsafe**) $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, may or may not be deadlock

o instance가 많으면 deadlock 방지 가능



- Deadlock ?
  - Circular wait (cycles)? : No

Resource Allocation Graph With A Deadlock



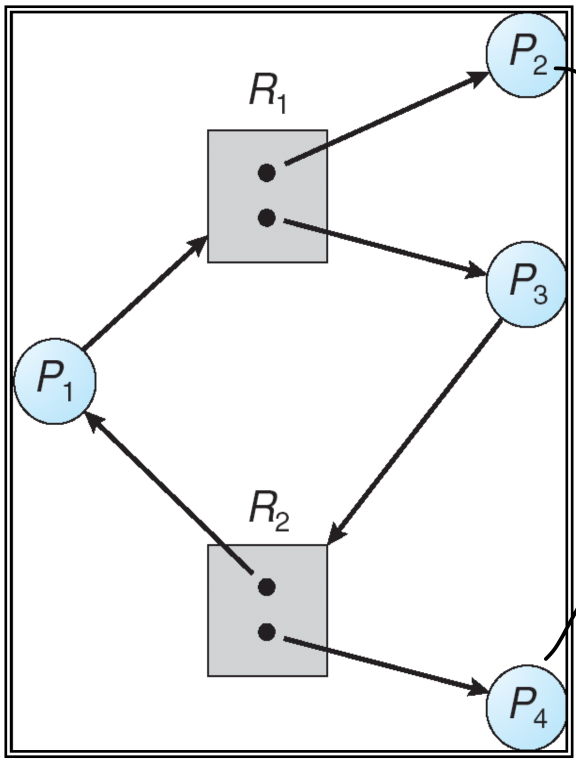
- **Deadlock ?**
  - Circular wait (cycles)? Yes

Two minimal cycles:

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

Graph With A Cycle But No Deadlock



minimal cycle:

$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

- Deadlock ? X
  - Circular wait (cycle): YES

request하는 대상이 없으므로
언젠가는 release 된다.
따라서 $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$ 같은 circle에서
deadlock가 발생하지 않는다!

Final'15 Problem

4. Which of the following is <u>**true**</u> when the resource allocation graph contains a cycle?

(a) A deadlock will never happen

(b) If there are ~~two~~ one instances per resource type, it is in deadlock

(c) If there is one instance per resource type, then it is ~~never in~~ deadlock

(d) All of the above

반드시 deadlock 걸림

(e) None of the above

Chapter 8: Deadlocks

- Deadlock Concept
- System Model: Resource Allocation Graph
- Deadlock Prevention
- Deadlock Avoidance

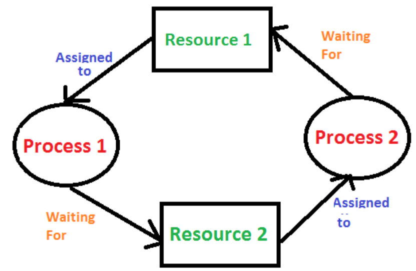
Deadlock Characterization

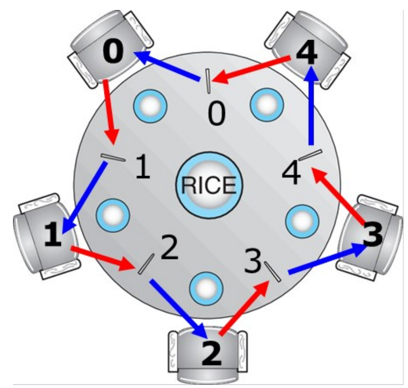
- **4 necessary conditions** for Deadlock

  = All 4 conditions ***must*** hold for deadlock to occur

4가지 조건이 모두 충족해야 deadlock가 발생한다!

1. **Mutual exclusion**
   - only one process at a time can use a resource instance
2. **Hold and wait**
   - a process holding at least one resource is waiting to acquire additional resources held by other processes
3. **No preemption**
   - a resource can be released only voluntarily by the process holding it, after that process has completed its task
4. **Circular wait**
   - Cycle in resource allocation graph
   - $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots P_N \rightarrow P_0$





Deadlock Prevention

- For deadlock to occur, each of the four necessary conditions (mutual exclusion, hold and wait, no preemption, circular wait) must hold
- **Deadlock prevention**
  - Ensure ***at least one*** of above conditions cannot hold – deadlock is **impossible!!**

1. *No* Mutual Exclusion

- Make all resources sharable at same time (e.g., read-only file is OK)
- **Problem:** *Practically impossible*. Some resources are non-sharable (e.g., mutex lock, printers)

- 4 **necessary conditions** for Deadlock
  1. **Mutual exclusion**
  2. **Hold and wait**
  3. **No preemption**
  4. **Circular wait**

■ 4 **necessary conditions** for Deadlock
1. **Mutual exclusion**
2. **Hold and wait**
3. **No preemption**
4. **Circular wait**

Deadlock Prevention (Cont.)

2. No Hold and Wait

$P_i \rightarrow R_j \rightarrow P_j$ → 이런상황

- must guarantee that whenever a process requests a resource, it does not hold any other resources
- **Example 1: Total allocation**: process requests and be allocated all its resources before it begins execution

  모든 리소스를 가져간다 그래서 total allocation

  - If all resources are not available must wait
  - hold all resources until process terminates; then release all resources
- **Example 2:** Request resource only when it holds none
  - A thread may request some resource and use them
  - Before it can request another resource, it must release all resources that it is currently allocated
- **Problem:** Low resource utilization
  - Resources may be allocated but unused for a long period – total allocation

4 necessary conditions for Deadlock
1. Mutual exclusion
2. Hold and wait
3. No preemption
4. Circular wait

Deadlock Prevention (Cont.)

3. *Allow* Preemption

- A lock can be taken away (preempted) from current owner
- Often applied to resources whose **state** can be easily saved and restored
- Example: preemptive CPU scheduling - RR
  - When context switching, CPU registers/PC can be stored in PCB
- **Problem:** OK for CPU but Basically ***impossible*** for some resources
  - E.g., Semaphore – if semaphore is preempted then race condition happens

- 4 **necessary conditions** for Deadlock
  1. **Mutual exclusion**
  2. **Hold and wait**
  3. **No preemption**
  4. **Circular wait**

Deadlock Prevention (Cont.)

4. No Circular Wait

- impose a **total ordering** of all resource types, and require that each process requests resources in an ***increasing order*** of enumeration
- Example

$$F(\text{tape drive}) = 1$$
$$F(\text{disk drive}) = 5$$
$$F(\text{printer}) = 12$$

$1 \to 5 \to 12$ (ok)

$12 \to 5$ (x) 요청을 못함!

wait until 12 terminates

- Each process can request resources only in an increasing order of enumeration - a process requests *Ri* . Then, it requests instances of resource type *Rj* if and only if *F*(*Rj*) > *F*(*Ri*).

- **Problem:** Always have to request resources in some order – e.g., cannot request printer then request disk.

Gachon University
School of Computing

Deadlock Prevention

- Summary of prevention schemes
  - Deny a (one) necessary condition for deadlocks among the 4 necessary conditions
  - Some cases practically unfeasible
  - Serious resource waste
    - Low device utilization

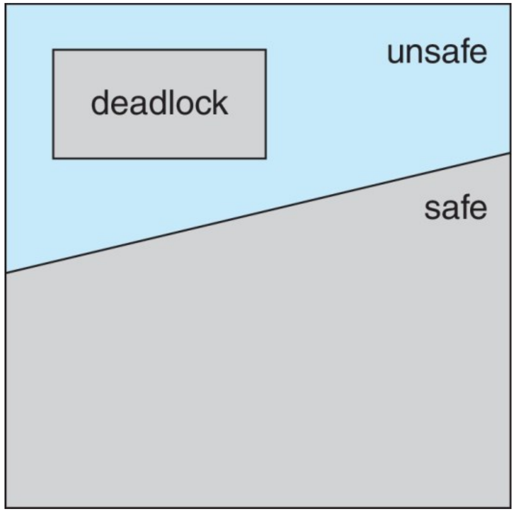
4가지 다 실용적이지 않음! 그냥 이론

Chapter 8: Deadlocks

- Deadlock Concept
- System Model: Resource Allocation Graph
- Deadlock Prevention
- Deadlock Avoidance

Basic Facts ; Safe, Unsafe, Deadlock

- If a system is in safe state $\Rightarrow$ no deadlocks
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock.



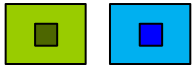
- Deadlock Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.

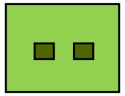
Final'15 Problem

3. Which of the following is **true**?

(a) A safe state may lead to a deadlocked state
(b) An unsafe state may lead to a deadlocked state.
(c) An unsafe state is necessarily always a deadlocked state.
(d) None of the above

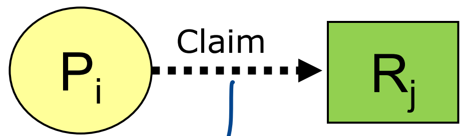
Deadlock Avoidance Methods

- Single instance of a resource type 
  ⇒ Use a Resource-Allocation Graph Algorithm

- Multiple instances of a resource type 
  ⇒ Use the Banker's algorithm

Resource-Allocation Graph Algorithm

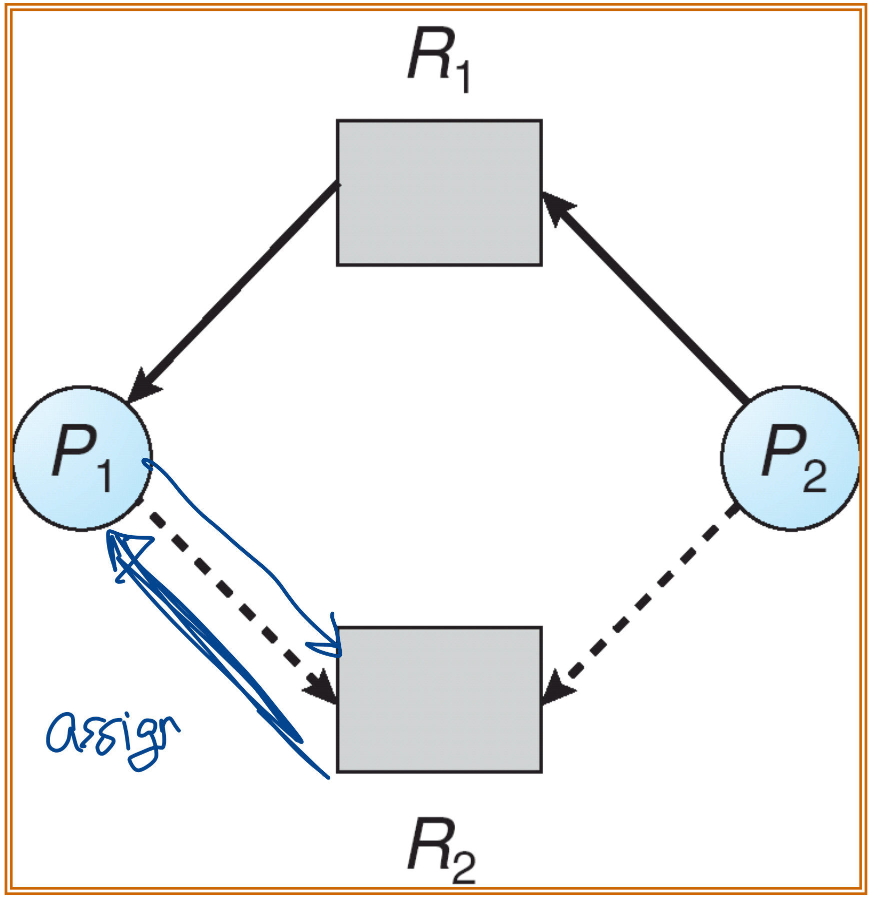
- ***Claim edge*** $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j (at some time in the **future**); represented by a dashed line.



request할 수도 있다! 라는 의미

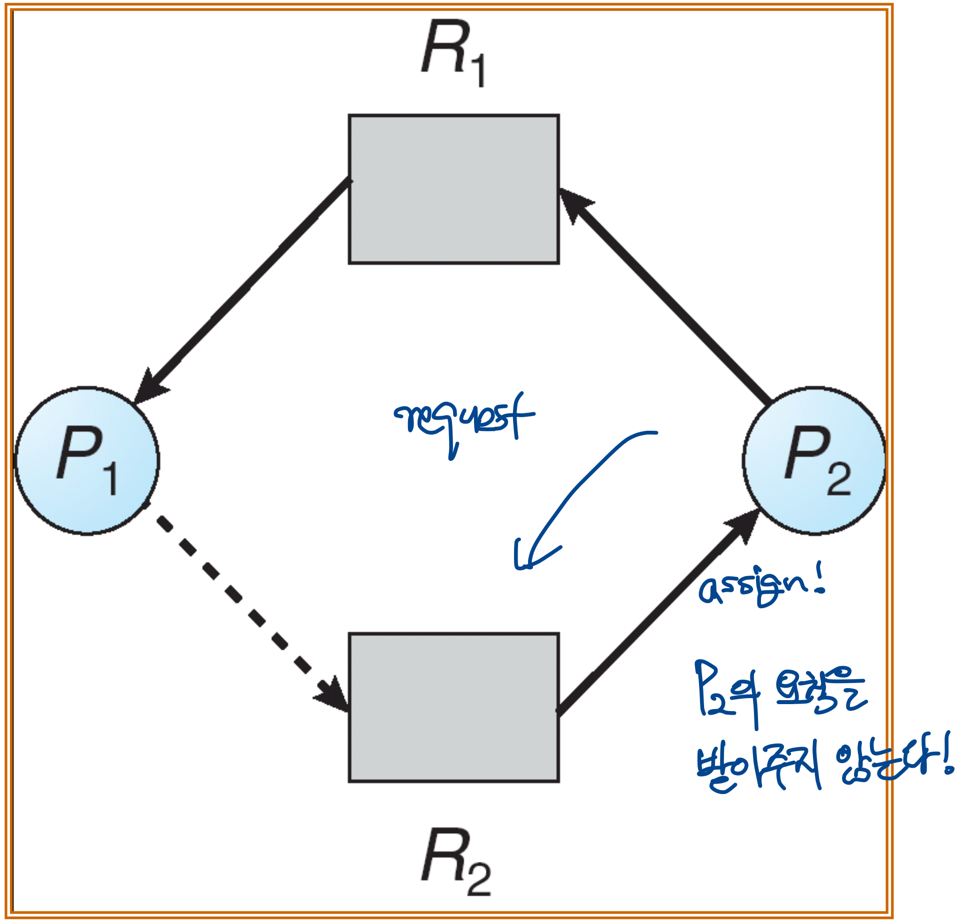
- Claim edge is converted to a request edge when a process actually requests a resource.

Resource-Allocation Graph Algorithm



- P_i requests resource R_j - request is granted *only if*
  - converting request edge $P_i \rightarrow R_j$ to assignment edge $R_j \rightarrow P_i$ does not result in **cycle**
- Example: allocation sequence
  - P_1 requests R_2 – accept?
  - **Safe state!**

Resource-Allocation Graph Algorithm



- Allocation sequence
  - P_2 requests R_2 – accept?
  - **Unsafe state!**
- When will P_2 be able to use R_2?
  - After P_1 is terminated

Banker's Algorithm

- A resource allocation and deadlock avoidance algorithm
- <u>Multiple</u> instances

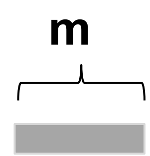
- Assumptions
  - Each process must *a priori* claim *maximum use*.
  - When a process requests a resource it may have to *wait*.
  - When a process gets *all* its resources it must *return* them in a finite amount of time.

- Consists of two sub-algorithms
  - **Safety Algorithm**
  - **Resource-Request Algorithm**

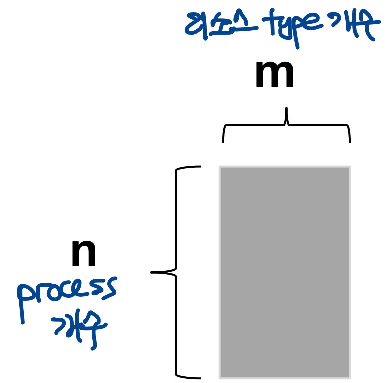
Data Structures for the Banker's Algorithm

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. 현재 할당된 개수
  - 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

Is it currently **safe state**?

- <u>Algorithm for finding out whether a system is in a</u> **<u>safe state</u>**
  - An order of m x n^2 operations may be required

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

   Work = *Available*

   Finish [*i*] = *false* for *i* = 0, 1, 2, …, *n-1.*

2. Find and *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.

Example of Bankers 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:

현재 할당받은 instance

m=3

Available

Finish		Allocation	Max	Need	Work
		A B C	A B C		A B C
F	P_0 x	0 1 0	7 5 3	7 4 3	3 3 2
T	P_1 o	2 0 0	3 2 2	1 2 2	5 3 2
F	P_2 x	3 0 2	9 0 2	6 0 0	7 4 3
F	P_3 o	2 1 1	2 2 2	0 1 1	7 4 5
F	P_4 o	0 0 2	4 3 3	4 3 1	7 4 5

n=5

→ P1의 alloc은 언젠가는 돌려주게 된다.
→ P3 〃
→ P4 〃
→ P2, P0 도 처음에는 안 되었지만 이제는 need<work 이므로 alloc 가능하다!

만족가능

└ 현재의 allocation에서 추가로 요청가능한 최대 resource 개수

- Q : The system is in a safe state ?

현재 work가 need를 만족시킬 수 있는가?

→ 3, 3 2 로 need의 모든 걸 만족가능? 결국 만족 가능해진다!

Example of Bankers 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	Work
	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~~	5 3 2
~~P_2~~	~~3 0 2~~	~~9 0 2~~	7 4 3
~~P_3~~	~~2 1 1~~	~~2 2 2~~	7 4 5
~~P_4~~	~~0 0 2~~	~~4 3 3~~	7 5 5

- Q : The system is in a safe state ?

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

- Q : The system is in a safe state ?

Yes, since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria.

Resource-Request Algorithm for Process P_i

Can we remain in **safe state** after **request**?

Request = request vector for process P_i.

$Request_i[\,j\,] = k$: P_i wants k instances of resource type R_j.

1. If $\boldsymbol{Request_i} \le \boldsymbol{Need_i}$ go to step 2.
   Otherwise, raise error condition, since process has exceeded its maximum claim.

2. If $\boldsymbol{Request_i} \le \boldsymbol{Available}$, 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: P_1 requests (1,0,2) (Cont.)

1. request를 P의 need 보다 작은지
2. 〃 available 〃
3. 빌려주고 나서도 safe 한가를 체크하면 된다!

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

	Allocation		Need		Available	
	A B C		A B C		A B C	(1,0,2)
P_0	0 1 0	(1,0,2) +	7 4 3	(1,0,2)	$\boxed{2\ 3\ 0}$	(3,3,2)
P_1	3 0 2	(2,0,0)	$\boxed{0\ 2\ 0}$	(1,2,2)		
P_2	3 0 1		6 0 0			
P_3	2 1 1		$\boxed{0\ 1\ 1}$			
P_4	0 0 2		4 3 1			

만족가능 → safe

- Executing safety algorithm shows that sequence <P1, P3, P4, P0, P2> satisfies safety requirement.

- Q: Can request for (3,3,0) by P4 be granted? ① Need ok ② available no assign X
- Q: Can request for (0,2,0) by P0 be granted? ① 〃 ② available yes → avail (2,3,0) → (2,1,0)

unsafe!

Example: P_1 requests (1,0,2) (Cont.)

- Check that Request ≤ Available
  that is, $(1,0,2) \le (3,3,2) \Rightarrow$ true.

	Allocation	*Need*	*Work*
	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~~	5 3 2
~~P_2~~	~~3 0 1~~	~~6 0 0~~	7 4 3
~~P_3~~	~~2 1 1~~	~~0 1 1~~	7 4 5
~~P_4~~	~~0 0 2~~	~~4 3 1~~	7 5 5

- Executing safety algorithm shows that sequence <P1, P3, P4, P0, P2> satisfies safety requirement.

- Q: Can request for (3,3,0) by P4 be granted?
- Q: Can request for (0,2,0) by P0 be granted?

Quiz ‘14 Problem

Consider the right system state of resource allocation.
(1) Is this system currently deadlocked, or can any process become deadlocked? Why or why not? If not deadlocked, give an execution order. (5pts)

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	1	1	0	1	2	2	0	1	1	0	1
P1	0	2	3	1	0	6	5	2				
P2	1	0	1	2	2	0	1	3				
P3	0	2	3	0	0	3	5	1				
P4	1	0	1	4	1	6	5	6				

The Need becomes:

	Need			
	A	B	C	D
P0	1	1	1	0
P1	0	4	2	1
P2	1	0	0	1
P3	0	1	2	1
P4	0	6	4	2

It is <u>not deadlocked</u> since it is in safe state, and a safe sequence exists. Using banker's algorithm, the safe sequence is:

P2, P0, P3, P1, P4

(2) If a request from a process P4 arrives for (0, 1, 0, 0), can the request be immediately granted? Why or why not? If yes, show an execution order.

If the request is granted, then the available resource is (1,0,0,1). This meets the need of P2 (1,0,0,1), and then the work becomes (2,0,1,3). Now, the current work cannot meet any of the remaining process needs, thus a safe sequence does not exist.

Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state
  - Deadlock Prevention (Ch. 8.5)
  - Deadlock Avoidance (Ch. 8.6)

deadlock가 그렇게 자주 일어나는 것이 아니기 때문에 ...!

- **Deadlock Ignorance**: Ignore the problem and pretend that deadlocks never occur in the system (**Ostrich Algorithm**)
  - actually used by most operating systems, including Linux and Windows
  - up to the application developer to handle deadlocks
  - Question: Then what happens when deadlock occurs?

Conclusion

- Deadlock Concept: A set of blocked processes each holding a shared resource and waiting to acquire a resource held by another process in the set. – Dining Philosopher's Problem
- 4 necessary conditions for Deadlock: Mutual Exclusion, Hold and Wait, No Preemption, Circular Wait
- Deadlock Prevention: Make sure 4 necessary conditions for Deadlock does not hold
- Deadlock Avoidance: Always keeps safe state
  - Resource allocation graph
  - Banker’s Algorithm