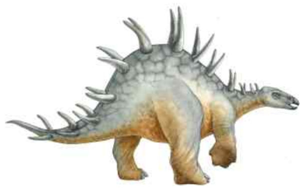
Chapter 7: Deadlocks

Chapter 7: Deadlocks

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

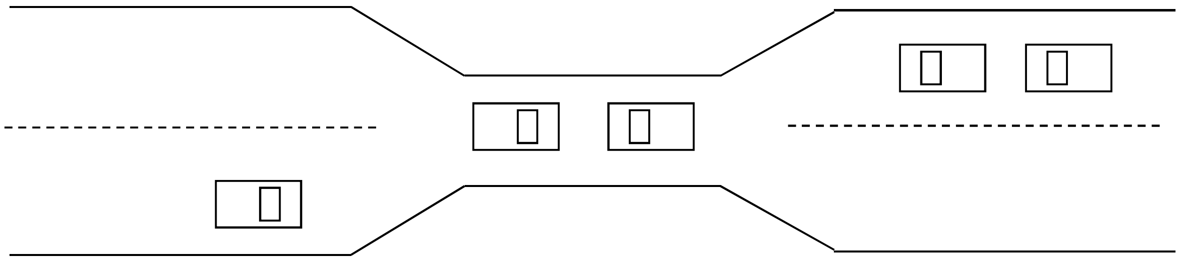


Deadlock 문제

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
  - System has 2 disk drives.
  - P_1 and P_2 each hold one disk drive and each needs another one.
- Example
  - semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)

Bridge Crossing Example



- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

Deadlock의 실제 예

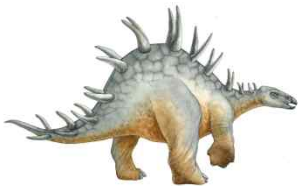
- 교착상태의 예 - 스풀링 시스템에서의 교착상태
  - 프로그램의 출력 ⇒ 디스크, 디스크 ⇒ 프린터
  - 출력이 완전히 끝난 후 실제 프린트 시작
  - 부분출력이 디스크를 완전히 채울 경우 **recovery** 가 어려움
  - 방지 방안 : **saturation(포화상태) threshold** 설정

Deadlock의 특성

다음 네 개의 조건이 모두 만족할 때만 Deadlock이 발생(필요조건)

- **Mutual exclusion(상호배제):**
  - only one process at a time can use a resource.
- **Hold and wait(점유하며 대기):**
  - a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption(비선점):**
  - a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait(순환대기):**
  - there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by

    $P_2, \ldots, P_{n-1}$ is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_0.



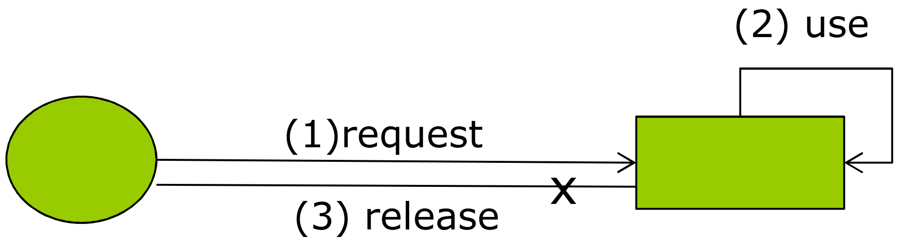
System Model

- 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.

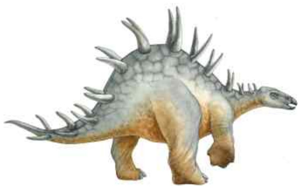
- 각 프로세스의 자원 사용 순서
  - request
  - use
  - release



Resource-Allocation Graph

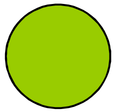
A set of vertices *V* and a set of edges *E*.

- 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 $R_j \rightarrow P_i$

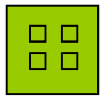


Resource-Allocation Graph (Cont.)

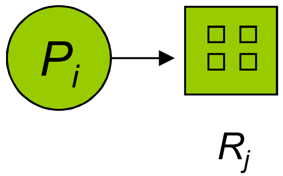
- Process



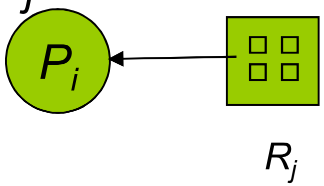
- Resource Type with 4 instances

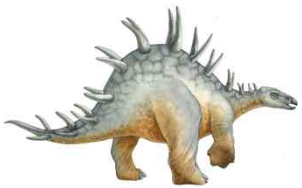


- P_i requests instance of R_j

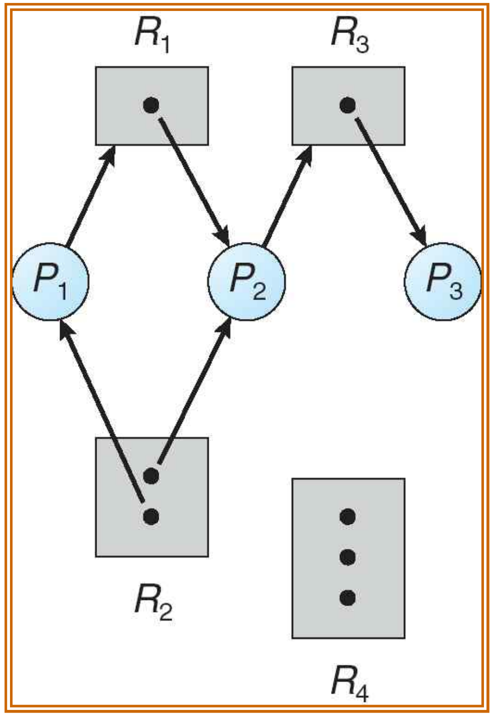


- P_i is holding an instance of R_j

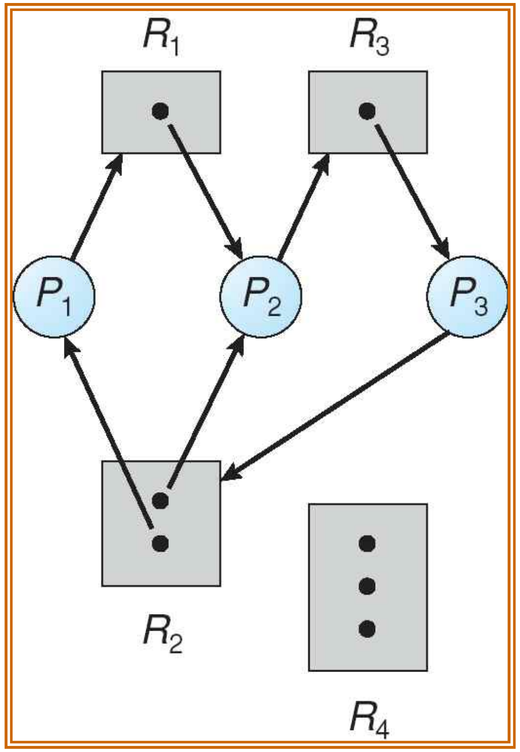




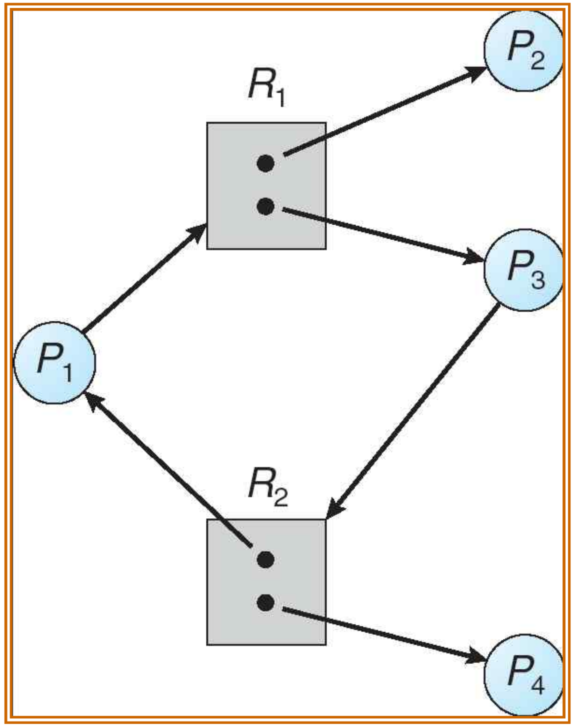
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock

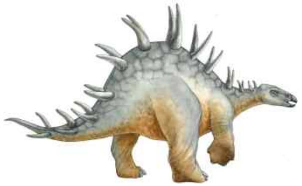


Graph With A Cycle But No Deadlock



Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock.
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.



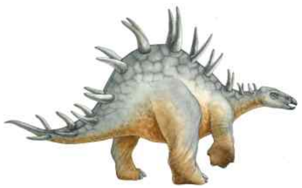
Java Deadlock Example

```
class A implements Runnable
{
   private Lock first, second;

   public A(Lock first, Lock second) {
      this.first = first;
      this.second = second;
   }

   public void run() {
      try {
         first.lock();
         // do something
            second.lock();
            // do something else
      }
      finally {
         first.unlock();
         second.unlock();
      }
   }
}
```

Thread A

```
class B implements Runnable
{
   private Lock first, second;

   public A(Lock first, Lock second) {
      this.first = first;
      this.second = second;
   }

   public void run() {
      try {
         second.lock();
         // do something
            first.lock();
            // do something else
      }
      finally {
         second.unlock();
         first.unlock();
      }
   }
}
```

Thread B



Java Deadlock Example

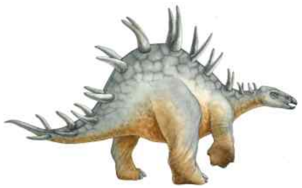
```
public static void main(String arg[]) {
    Lock lockX = new ReentrantLock();
    Lock lockY = new ReentrantLock();

    Thread threadA = new Thread(new A(lockX,lockY));
    Thread threadB = new Thread(new B(lockX,lockY));

    threadA.start();
    threadB.start();
}
```

Deadlock is possible if:

threadA -> lockY -> threadB -> lockX -> threadA

Methods for Handling Deadlocks

- 예방(prevention) 또는 회피(avoidence) : Deadlock 상태가 절대 발생하지 않도록 함
- 회복 : Deadlock 상태가 되는 것을 일단 허용한 후, 걸리면 회복.
- 무시 : 시스템 상에서 deadlock이 거의 발생하지 않는다고 가정; used by most operating systems, including UNIX.



Deadlock Prevention

교착상태가 발생하려면, 네 가지 필요조건 각각이 만족해야 하므로, 이들 조건중 최소한 하나가 성립하지 않도록 보장

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources.(강제불가)

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - 1) Require process to request and be allocated all its resources before it begins execution(실행전 모두 확보),
  - 2) or allow process to request resources only when the process has none. (확보된 것이 아무것도 없을 때 요청)
  - Low resource utilization; starvation possible
  - 예) DVD로 부터 Disk로 파일 복사 후 정렬, 프린트하는 프로그램

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 resources currently being held are released.
    - (한 리소스를 확보하고 있으면서, 다른 리소스가 즉각적으로 확보될수 없으면 바로 release)
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
    - 모든 리소스들이 모두 재확보가능할 때 restart

- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.
  - 오름차순 순차적 할당
  - 또는 아랫 순서의 자원 release후 할당

Deadlock Avoidance

시스템에 대한 선제적인 정보를 활용하여 회피

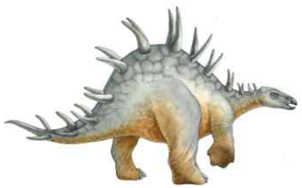
- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.
  - 가장 단순하면서 유용한 모델은 각 프로세스가 필요한 각 타입의 자원의 최대 숫자를 선언하도록 요구하는 것
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
  - circular-wait 상태가 절대 발생하지 않도록 resouce-allocation state를 유지함
- 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.

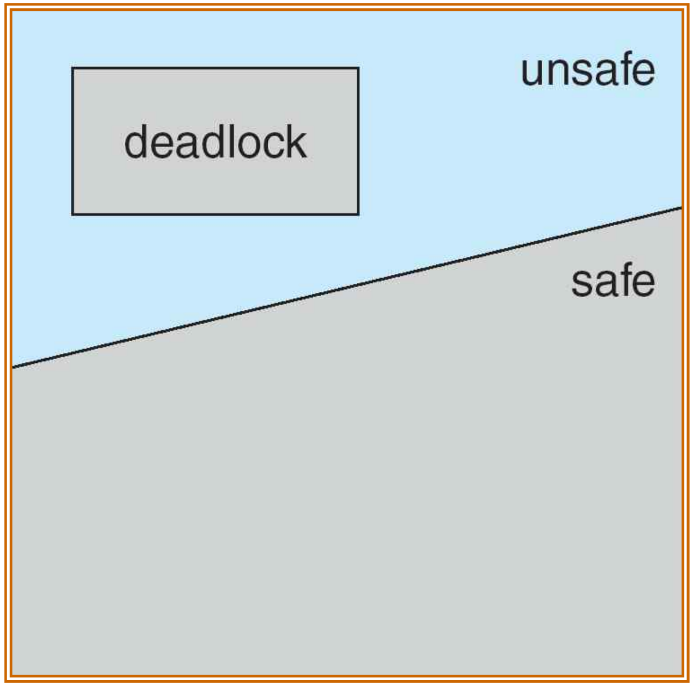
- System is in safe state if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes is the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$.
  - safe sequence(안전순서)

- 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.



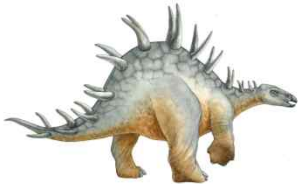
Basic Facts

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



Avoidance algorithms

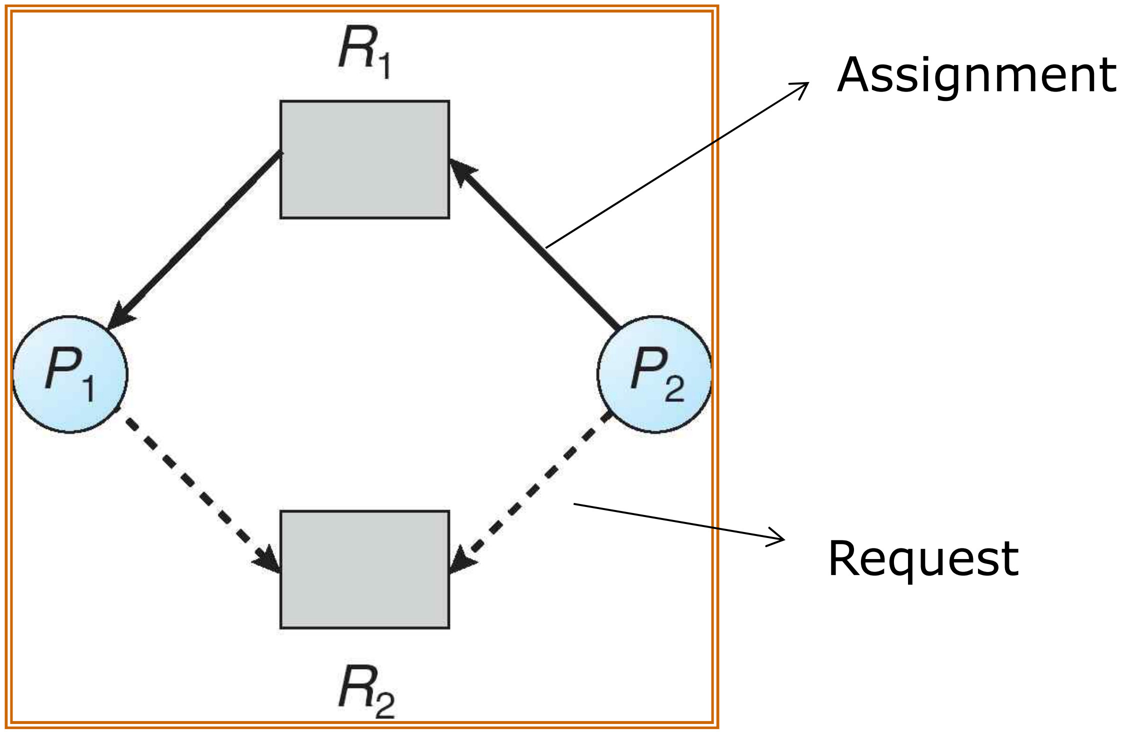
- Single instance of a resource type.
  - resource-allocation graph
- Multiple instances of a resource type.
  - the banker's algorithm



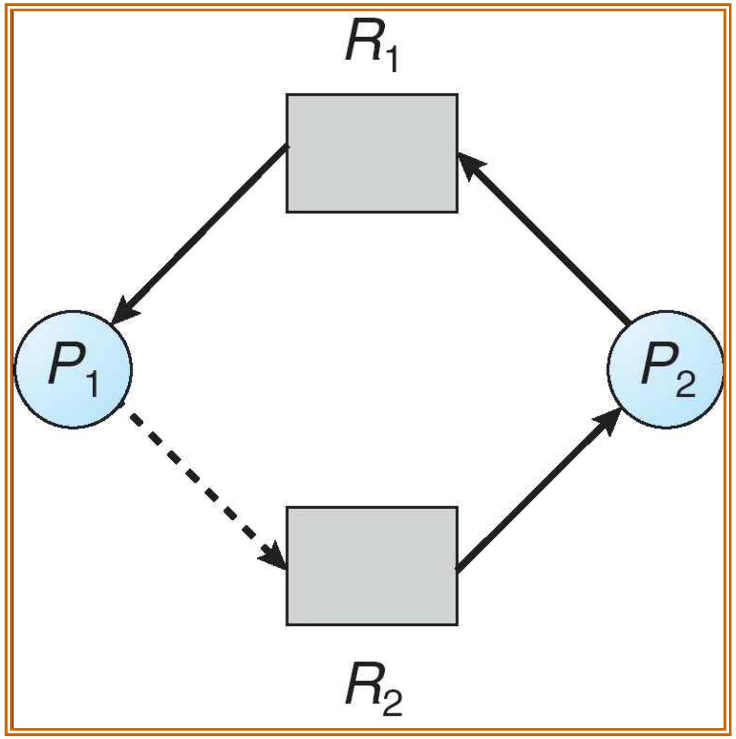
Resource-Allocation Graph Scheme

- *Request edge* $P_i \rightarrow R_j$
  - indicated that process P_j may request resource R_j; represented by a dashed line.
  - Claim edge converts to request edge when a process requests a resource.
- 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



Unsafe State In Resource-Allocation Graph

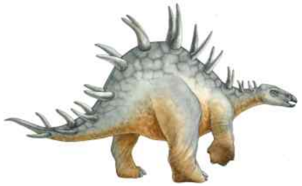


Resource-Allocation Graph Algorithm

- 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(은행원 알고리즘)

- 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.
  - 현재 프로세스에 할당되고 남아있는 자원의 양
- *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.
  - 각 프로세스 당 최대 요구량
- *Allocation:* n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j.
  - 현재 할당된 양
- *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.
  - 최대 프로세스 활성화 시에 추가로 필요로 되는 양

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

Safety Algorithm

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

   $Work = Available$

   $Finish[i] = false$ for $i = 0, 1, \ldots, n-1$.

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.

Resource-Request Algorithm for Process P_i

리소스의 요청을 처리했을 때 safe state를 유지할 수 있는지 처리

Request = request vector for process P_i. If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j.

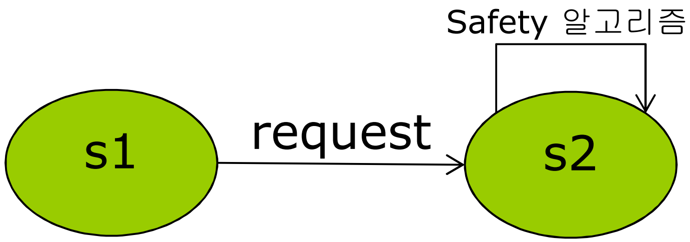
1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

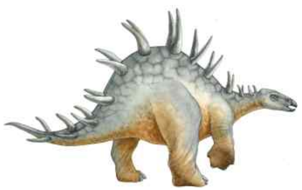
$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- *If safe ⇒ the resources are allocated to Pi.*
- *If unsafe ⇒ 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:

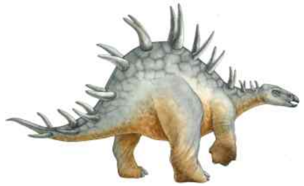
	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

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.

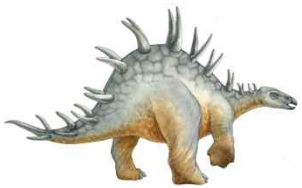


Example: P_1 Request (1,0,2)

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

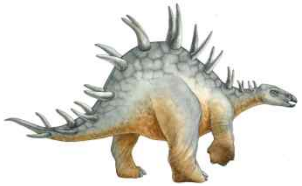
	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence $< P_1, P_3, P_4, P_0, P_2>$ satisfies safety requirement.
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?



회복 기법을 위한 **Deadlock Detection**

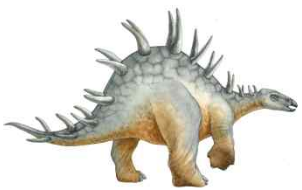
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme



각 자원타입 당 자원이 한 개인 경우

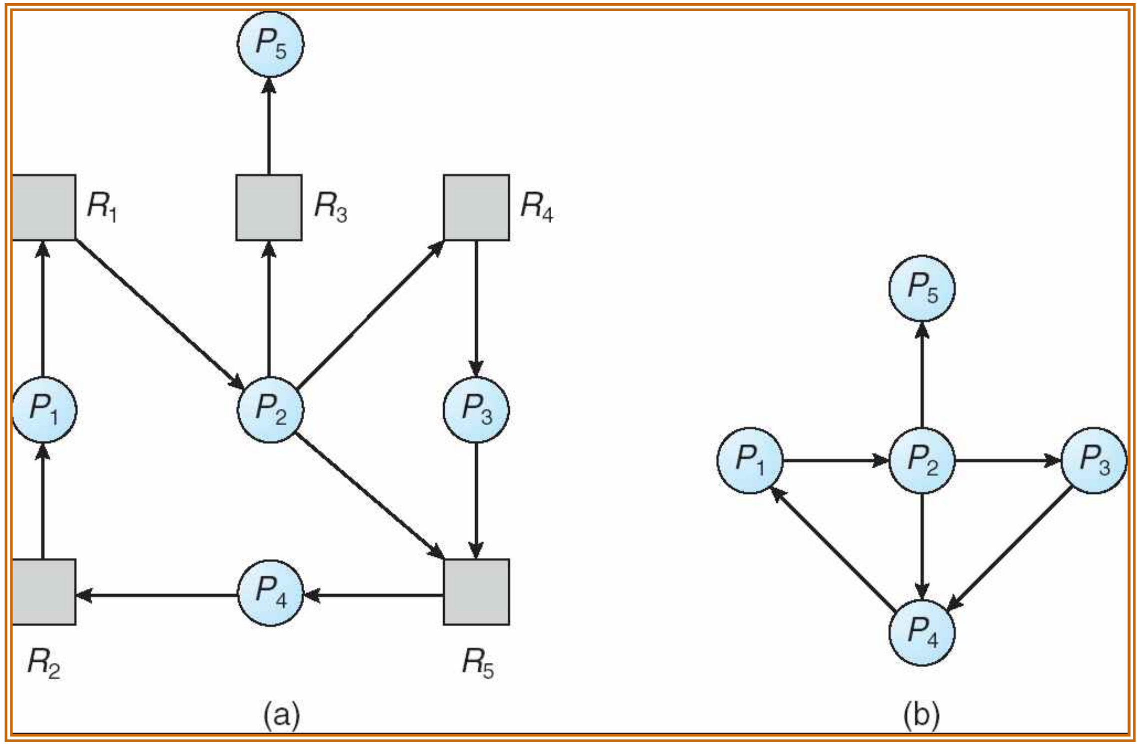
Single Instance of Each Resource Type

- Maintain *wait-for* graph
  - Nodes are processes.
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j.
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock.
  - 주기적으로 그래프 상의 cycle 존재 여부를 검색하는 알고리즘을 실행
- 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.



각 자원타입 당 자원이 한 개인 경우

Resource-Allocation Graph and Wait-for Graph

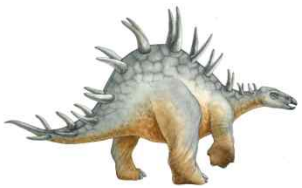


Resource-Allocation Graph　　Corresponding wait-for graph

각 자원타입 당 자원이 여러 개인 경우

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 P_i is requesting *k* more instances of resource type. R_j.

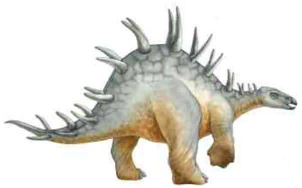


각 자원타입 당 자원이 여러 개인 경우

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.

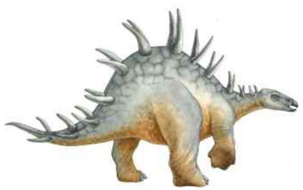


Detection Algorithm (Cont.)

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 x $n^{2)}$ operations to detect whether the system is in deadlocked state.



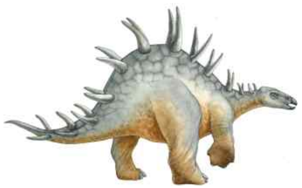
각 자원타입 당 자원이 여러 개인 경우

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*.

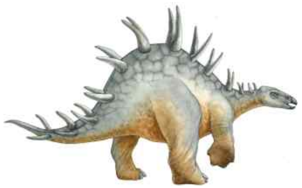


Example (Cont.)

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

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

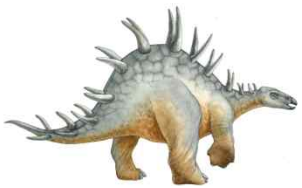
- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests.
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.



각 자원타입 당 자원이 여러 개인 경우

Detection-Algorithm Usage

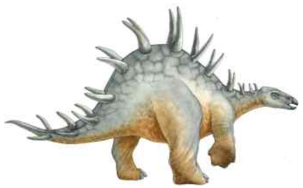
- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to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



각 자원타입 당 자원이 여러 개인 경우

Recovery from Deadlock: Process Termination

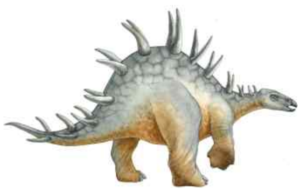
- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- In which order should we choose to abort?
  - Priority of the process.
  - How long process has computed, and how much longer to completion.
  - Resources the process has used.
  - Resources process needs to complete.
  - How many processes will need to be terminated.
  - Is process interactive or batch?



Recovery from Deadlock: Resource Preemption

- 프로세스 종료를 통한 교착상태 회복
  - 교착 프로세스를 모두 중지
  - 교착 상태가 제거 될때까지 한 프로세스씩 종료

- Preemption 상태에서의 회복
  - 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 7