PART 2. PROCESS MANAGEMENT

Chapter 7: Deadlocks

WHAT'S AHEAD:

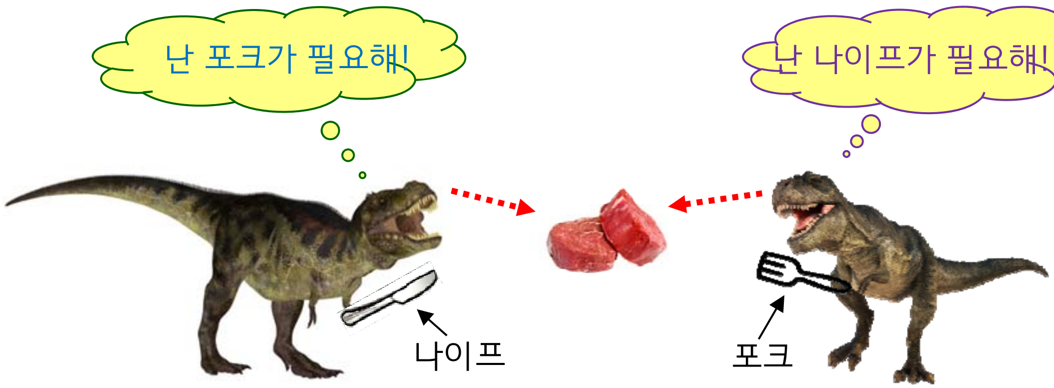
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

WE AIM:

- To develop a description of deadlocks
- To present a number of different methods for preventing or avoiding deadlocks in a system

Note: These lecture materials are based on the lecture notes prepared by the authors of the book titled *Operating System Concepts*, 9e (Wiley)

핵심•요점　교착상태 = Deadlock



교착상태는 왜 치명적일까?

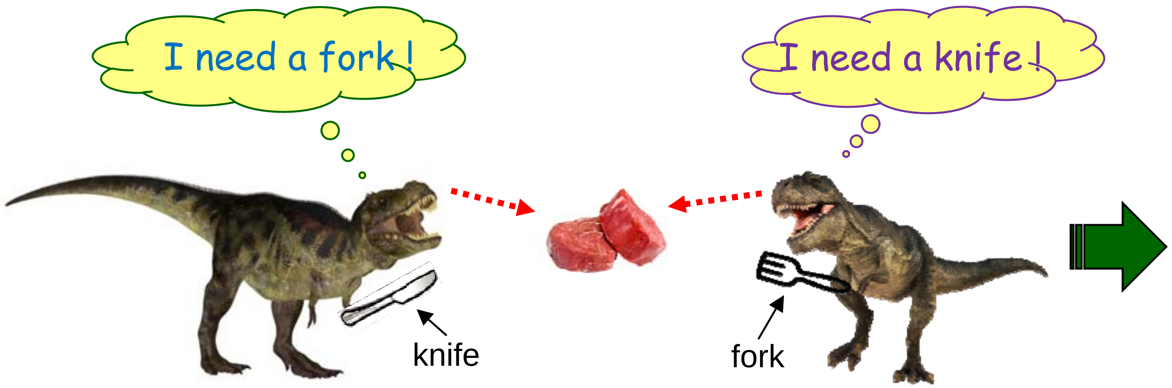
- 컴퓨터 동작 멈춤!
- 스스로 해결 불가!

이 데드락 문제를 어떻게 해결할까?

1. 예방하라!
2. 회피하라!
3. 발견해서 조치를 취하라!
4. 무시하고 가만두라!

Core Ideas

Deadlock = Deadly Embrace



Why so “deadly”?

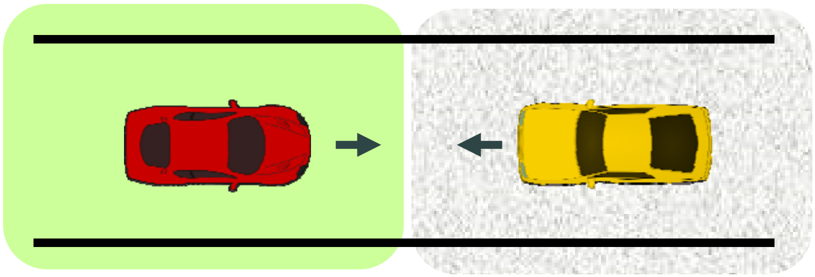
- Impossible to keep running
- Impossible to work themselves out

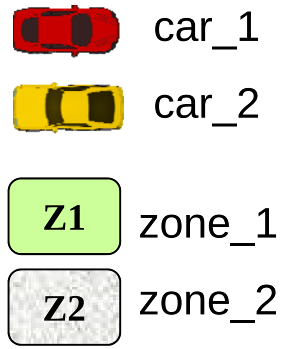
How to resolve this deadlock problem?

1. Prevent it!
2. Avoid it!
3. Detect it and take action!
4. Do nothing! Let it be!

Illustration of Deadlock

- Two-way single-lane road analogy





- We can interpret this diagram as follows:
  - Zone_1(Z1) and zone_2(Z2) are the resources both of which a car must obtain to pass through this road
  - Car_1 is holding Z1 and waiting for Z2 to be available
  - Car_2 is holding Z2 and waiting for Z1 to be available
  - Thus, car_1 and car_2 are waiting for the resource that is held by the other party at the same time
  - ➔➔ Deadlock!! Deadly embrace!!

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$
  - Examples: CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

Deadlock Characterization

- [Necessary conditions] *Deadlock can arise if the following 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.

Example: Deadlock with Mutex Locks

- Deadlocks can occur via system calls, locking, etc
- See example box in text page 313 (Deadlock with mutex locks) for a multithreaded Pthread program using mutex locks

- Scenario
  - two mutex locks
  - two threads, each locking the same locks
  1. thread 1: acquires 1st lock
  2. thread 2: acquires 2nd lock
  3. thread 1: tries 2nd lock
  4. thread 2: tries 1st lock

  --- deadlock ! ---

```
/* Create/initialize the mutex locks */
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;
pthread_mutex_init(&first_mutex,NULL);
pthread_mutex_init(&second_mutex,NULL);

/* 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);
}
```

Resource-Allocation Graph

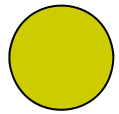
- A set of vertices V and a set of edges E

- 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

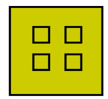
- **request edge** – directed edge $P_i \rightarrow R_j$

- **assignment edge** – directed edge $R_j \rightarrow P_i$

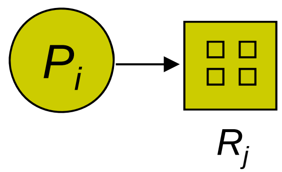
- Process



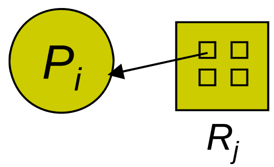
- Resource Type with 4 instances



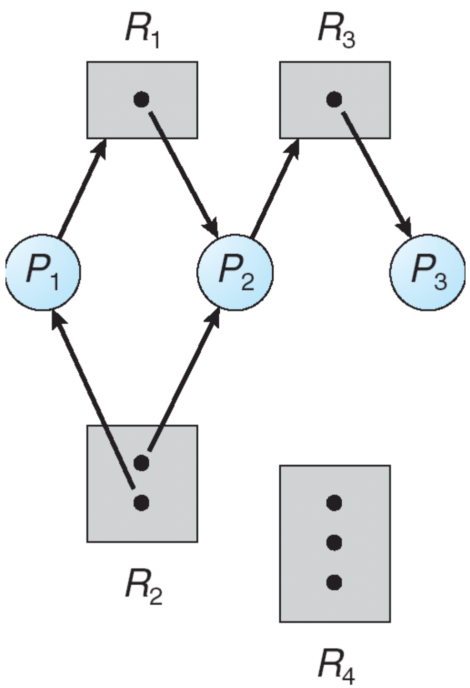
- P_i requests instance of R_j



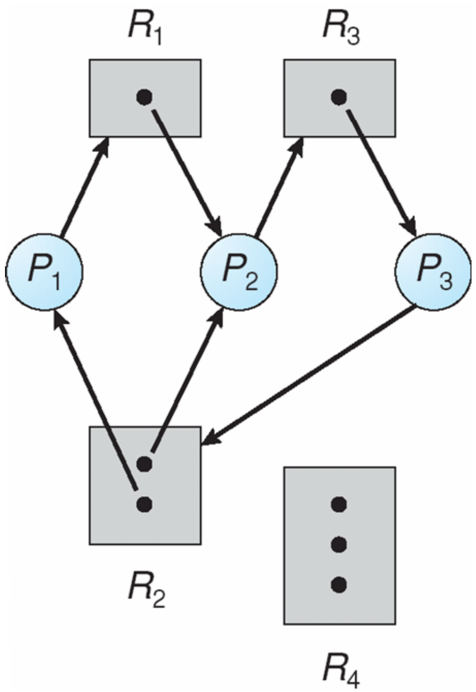
- P_i is holding an instance of R_j



Example of a Resource Allocation Graph



No deadlock

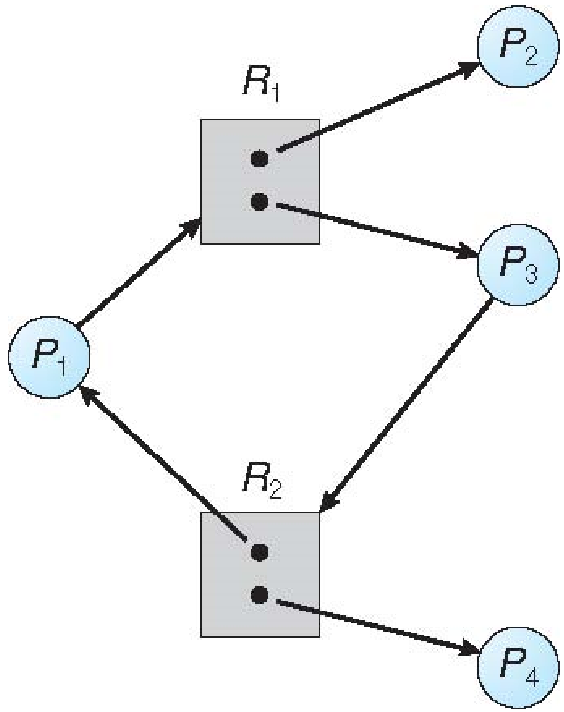


Deadlock

- [Directed] Cycle in a [Directed] Graph
  - A sequence of vertices starting and ending at the same vertex
  - Absence of a cycle in the resource allocation graph indicates deadlock-free state

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



Graph with a cycle but no deadlock

Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state
  - Prevent a system from entering a deadlock state
  - Let a system avoid a deadlock state

- Allow the system to enter a deadlock state and then recover

- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

- Restrain the ways request can be made.

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated ***all*** its resources before it begins execution, or allow process to request resources only when the process has none
  - 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 resources currently being held 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

Deadlock Example

```
int A, B;
lock_t LA, LB; /* Locks for A and B */
```

```
/* Thread T1 */
   lock(LA); /* (1) */
   A -= 100;
      /* At this point,
          T2 starts */
   lock(LB); /* (4) */
      /* waiting */

   /* -- Deadlock -- */

   B += 200;
   A += B;
   unlock(LB);
   A *= 2;
   unlock(LA);
```

```
/* Thread T2 */
   lock(LB); /* (2) */
   B += 100;
   lock(LA); /* (3) */
     /* Now, waiting */
     /* T1 resumes */

   /* -- Deadlock -- */

   A -= 200;
   A += B;
   unlock(LA);
   B *= 5;
   unlock(LB);
```

Deadlock Example with Lock Ordering

```
void transaction(Account from, Account to, double amount)
{
   mutex lock1, lock2;
   lock1 = get_lock(from);
   lock2 = get_lock(to);
   acquire(lock1);
      acquire(lock2);
         withdraw(from, amount);
         deposit(to, amount);
      release(lock2);
   release(lock1);
}
```

- What if two threads invoke this function simultaneously as follows:

```
transaction(checking_account, savings_account, 25); /* thread 1 */
transaction(savings_account, checking_account, 50); /* thread 2 */
```

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

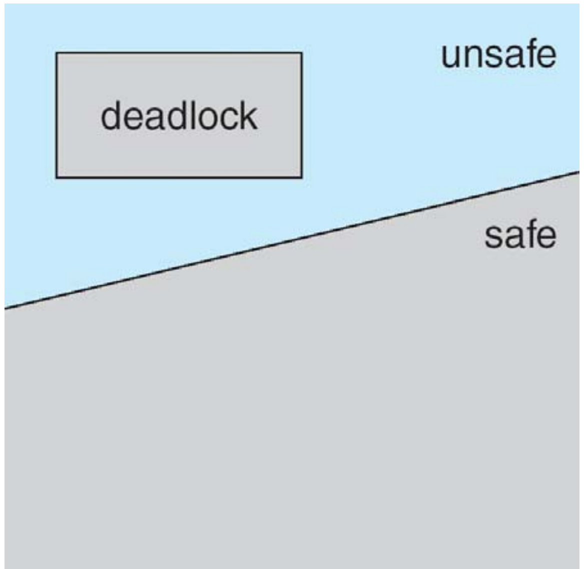
Safe State

- 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 the resources that P_i needs are not immediately available, then P_i can wait until all P_j have finished
  - When all P_j are 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
  - If no such sequence exists, then the system state is said to be unsafe

Basic Facts

- If a system is in safe state $\Rightarrow$ no deadlocks
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.
- Avoidance algorithms
  - Single instance of a resource type → Use a **resource-allocation graph**
  - Multiple instances of a resource type → Use the **banker's algorithm**

- Safe, Unsafe, Deadlock State

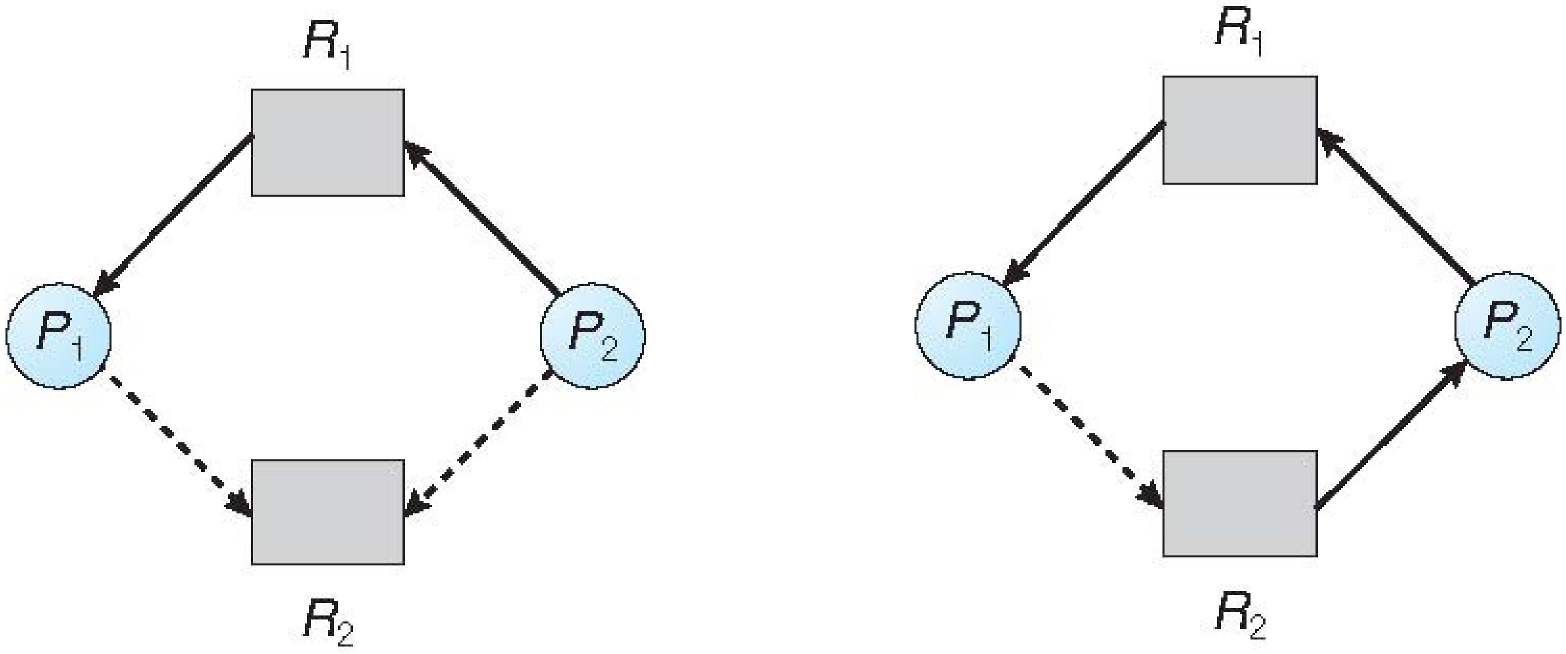


Resource-Allocation Graph Scheme

- **Claim 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
- Request edge converted 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

Unsafe state

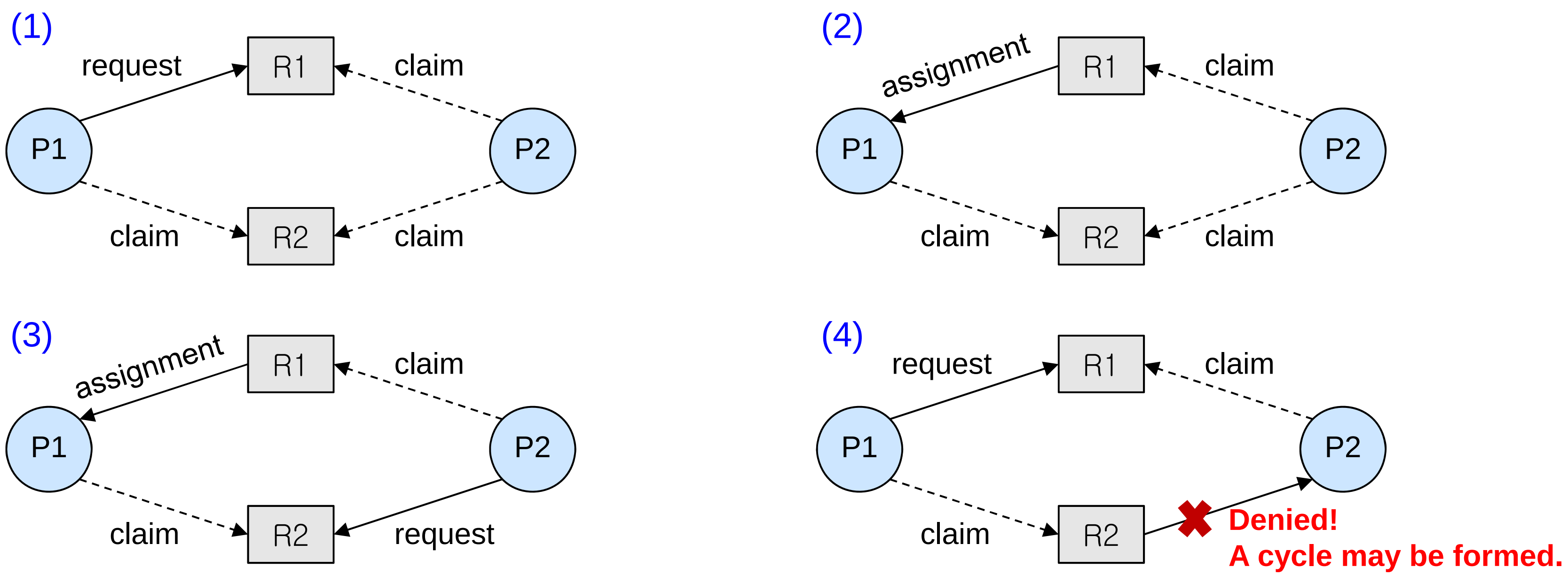


(a) (b)

Although R_2 is currently free as shown in (a), it cannot be allocated to P_2 since this action will create a cycle as shown in the graph (b).

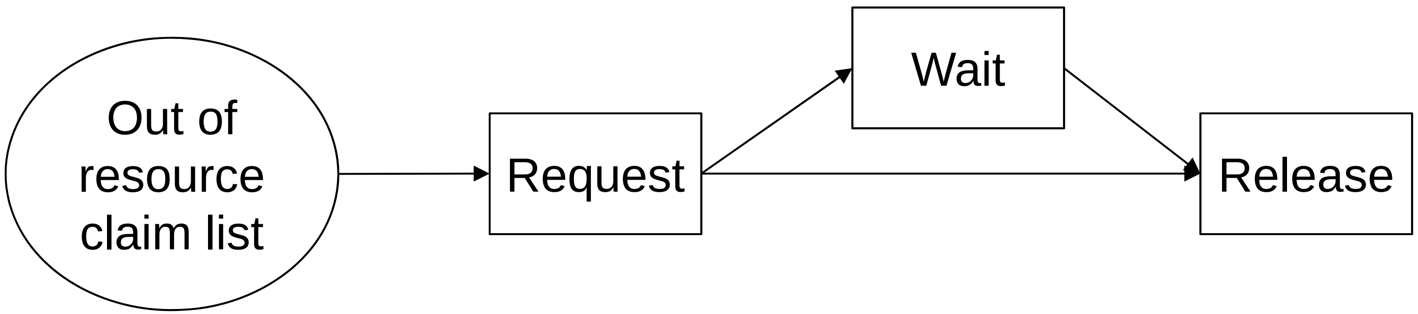
Resource-Allocation Graph Algorithm

- 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
- 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 Banker's Algorithm

- Let n = number of processes, and m = number of resource types.

- **Available**: Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available
- **Max**: $n \times m$ matrix. If $Max[i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation**: $n \times m$ matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j
- **Need**: $n \times m$ 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

   (Upon completion of consumption, release all resources)

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

- ***Request*** = request vector for process P_i.
  If $\textbf{\textit{Request}}_i[j] = k$ then process P_i wants k instances of resource type R_j

1. If $\textbf{\textit{Request}}_i \leq \textbf{\textit{Need}}_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $\textbf{\textit{Request}}_i \leq \textbf{\textit{Available}}$, 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:

   $$\textit{Available} = \textit{Available} - \textit{Request};$$

   $$\textit{Allocation}_i = \textit{Allocation}_i + \textit{Request}_i;$$

   $$\textit{Need}_i = \textit{Need}_i - \textit{Request}_i;$$

   1) If safe $\Rightarrow$ the resources are allocated to P_i
   2) If unsafe $\Rightarrow$ P_i 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** (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			

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

Upon execution of	***Available***		
	A	B	C
P_1	5	3	2
P_3	7	4	3
P_4	7	4	5
P_2	10	4	7
P_0	10	5	7

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

New state after (1,0,2) is allocated to **P_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?

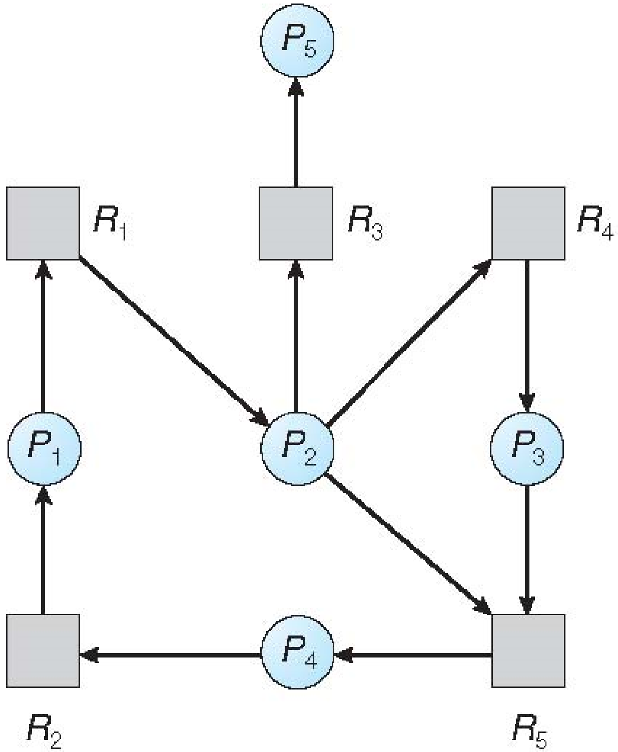
Deadlock Detection

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

Single Instance of Each Resource Type

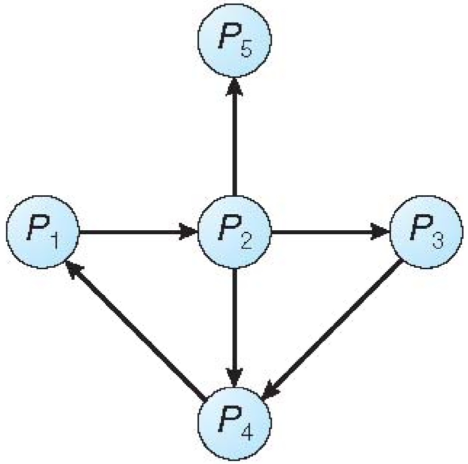
- Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



(a)

Resource-Allocation Graph



(b)

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 P_i is requesting ***k*** more instances of resource type 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 $Allocation_i \neq 0$, 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

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

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

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

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 $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in ***Finish[i] = true*** for all *i*

Upon execution of	*Available*		
	A	B	C
P_0	0	1	0
P_2	3	1	3
P_3	5	2	4
P_1	7	2	4
P_4	7	2	6

Example (Cont.)

- P_2 requests an additional instance of type C

Request

	A	B	C
P_0	0	0	0
P_1	2	0	2
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

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.

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?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. Is process interactive or batch?

Recovery from Deadlock

- Resource Preemption

- Elimination of deadlock
  - Successively preempt some resource(s) from processes and give these resources to other processes until the deadlock cycle is broken.
- Issues
  - **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 if based on cost factors; to avoid starvation, include number of rollback in cost factor

Summary

- 교착상태(deadlock)란?
- 시스템 모델: 프로세스의 자원(resource) 이용 절차. request – use – release
- 교착상태의 특징
  - 필요조건 네가지
  - 자원할당 그래프 구성: cycle → deadlock
- 교착상태의 처리방법
  - 예방, 회피, 검출 및 사후처리, 무시
- 교착상태의 예방(prevention)
  - 필요조건 네 가지가 성립하지 않게 미리 조치함
  - 어느 하나만이라도 성립하지 않으면 OK

- What is deadlock?
- System model: how a process utilize the resource. request – use - release
- Deadlock characterization
  - four necessary conditions
  - construct resource allocation graphs: cycle → deadlock
- Methods for handling deadlocks
  - prevention, avoidance, detection and"post-mortem" actions, no actions
- Deadlock prevention
  - take precautions against necessary conditions
  - any one of the conditions does not hold → no deadlock

Summary (Cont.)

- 교착상태의 회피(avoidance)
  - 안전한 상태(safe state)
  - Banker's algorithm: 다수의 프로세스가 자원을 이용하려고 할 때, 각 프로세스의 요구를 만족할 수 있는 순서를 발견함 (만일 존재한다면)
- 교착상태의 검출(detection)
  - wait-for graph: 싸이클의 발견
  - 검출 알고리즘: 기본 아이디어는 Banker's algorithm과 유사. 어떤 순서대로 실행할 때, 모든 프로세스가 종료되는지 여부 조사
- 교착상태로부터 복구
  - 하나 혹은 복수의 프로세스 실행을 취소하여 싸이클 제거
  - 희생된 프로세스(victim)은 후진(rollback)
  - starvation 가능

- Deadlock avoidance
  - safe state
  - Banker's algorithm: While multiple processes use the resources, find the sequence that satisfies the demands, if any.
- Deadlock detection
  - find a cycle or cycles in the wait-for graph
  - detection algorithm: Basic idea is similar to the Banker's algorithm. Check if all processes terminate normally.
- Recovery from deadlock
  - abort one or more processes to remove a cycle
  - victim process: rollback
  - starvation possible