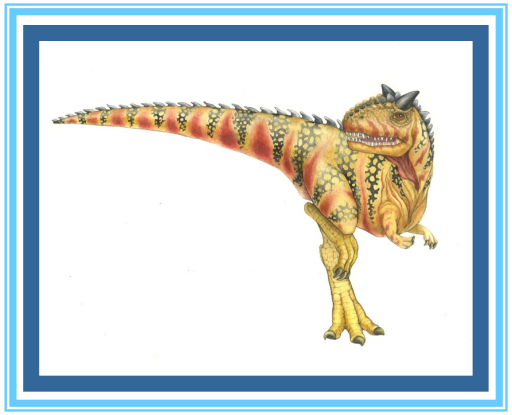
Chapter 8: Deadlocks



Chapter 8: Deadlocks

- System Model
- Deadlock in Multithreaded Applications
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention (방지)
- Deadlock Avoidance (회피)
- Deadlock Detection (탐지)
- Recovery from Deadlock (회복)

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, . . ., 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**
  - **use**
  - **release**

Deadlock in Multithreaded Application

- Two mutex locks are created an initialized:

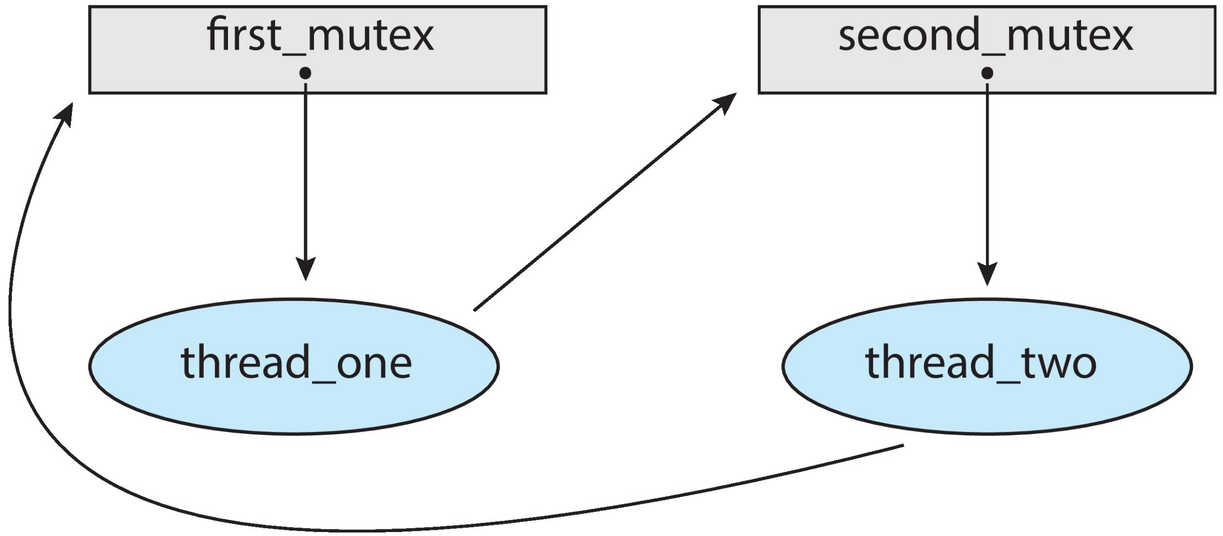
```
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

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

Deadlock in Multithreaded Application

```
/* 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 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**:



Livelock: 스레드가 정체된 상태는 아니지만 계속 시도해도 진행이 안 되는 경우. Ex) 마주 오던 두 사람이 같은 방향으로 계속 피할 때, 또는 CSMA/CD

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously. (필요조건)

- **Mutual exclusion**: only one process at a time can use a resource $\rightarrow$ 적어도 하나의 자원이 공유불가
- **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 $\rightarrow$ Task가 끝나야 자원을 놓는다
- **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.

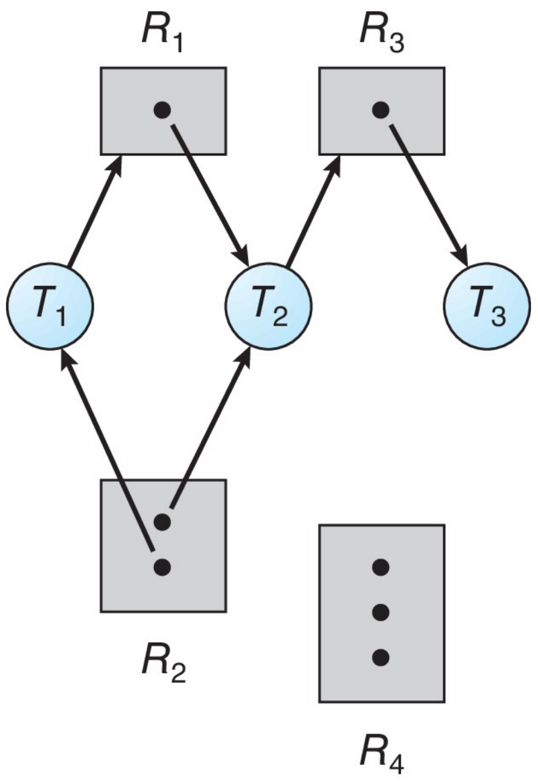
Resource-Allocation Graph

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 R1
- Two instances of R2
- One instance of R3
- Three instance of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- T3 is holds one instance of R3

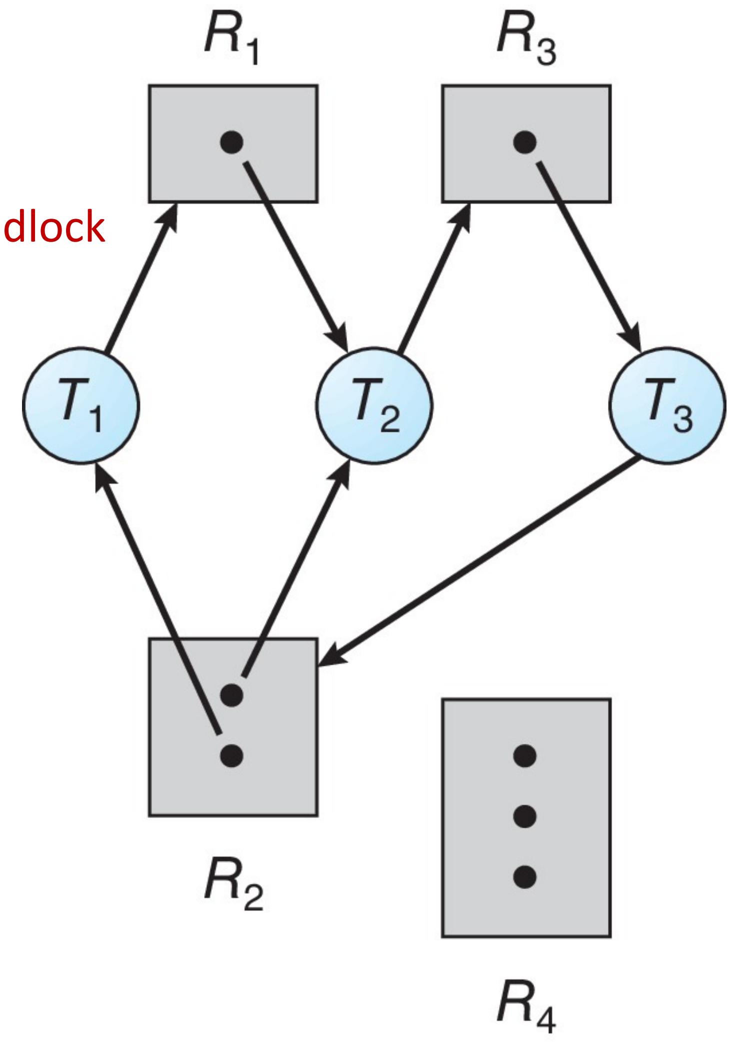


Resource Allocation Graph With A Deadlock

no cycle $\rightarrow$ no deadlock

deadlock $\rightarrow$ cycle

cycle $\rightarrow$ may or may not deadlock



Graph With A Cycle But No Deadlock

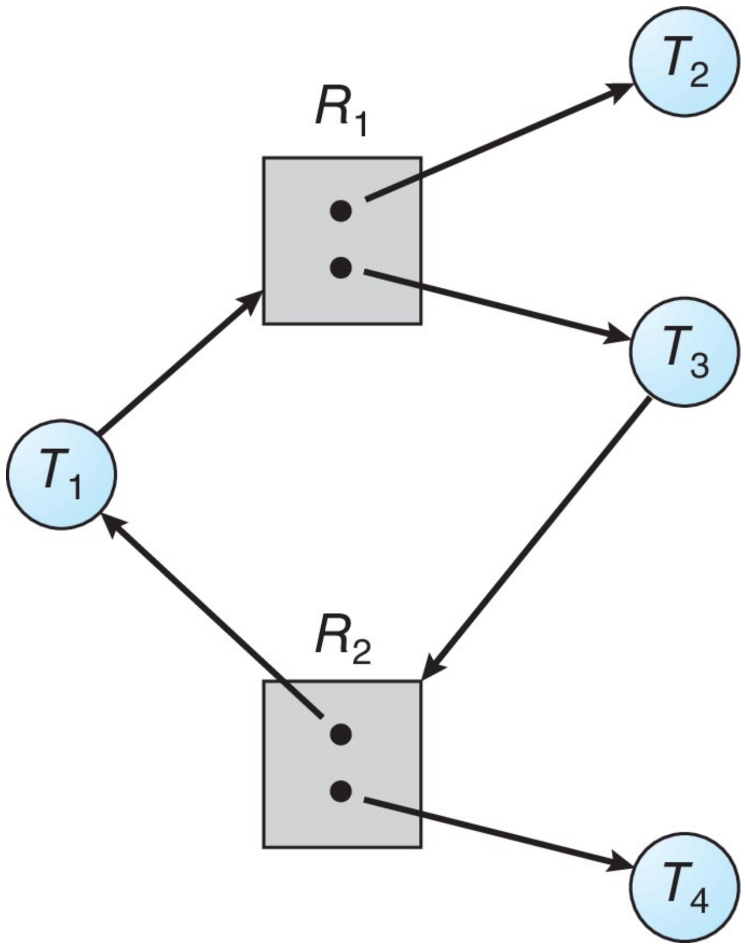
① single instance

cycle $\leftrightarrow$ deadlock

② multiple instances

cycle $\leftarrow$ deadlock

cycle $\nrightarrow$ deadlock



Basic Facts

- 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

Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state:
  - Deadlock prevention 필요조건 4개 중 적어도 하나 이상 성립이 안되게 함
  - Deadlock avoidance 필요한 자원에 대한 정보를 이용
- Allow the system to enter a deadlock state and then recover Ex) 데이터베이스 시스템에서 사용
- Ignore the problem and pretend that deadlocks never occur in the system. 대부분의 OS가 사용. 왜냐하면 비용이 저렴

Deadlock Prevention

Invalidate one of the four necessary conditions for deadlock:

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable 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 allocated to it.
  - Low resource utilization; starvation possible

공유 자원이 너무 많아서 이 조건을 막는 것은 어렵다

① 시작하기 전에 필요한 자원을 한꺼번에 할당하거나
② 잡고 있는 자원이 없을 때만 자원을 요청함

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

① 자원 할당이 불가능하면 가지고 있는 자원을 다 내 놓음
② 대기 중인 프로세스의 자원을 반납하게 함
③ 자원을 다 할당 받을 수 있을 때 프로세스를 다시 시작함

"가장 현실적인 방법"
어떤 자원을 요청할 때 그 자원보다 높은 순서의 자원을 가질 수 없다.

Circular Wait

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

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

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

순서대로 lock을 획득해야 하는 규칙을 어김

```
/* 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

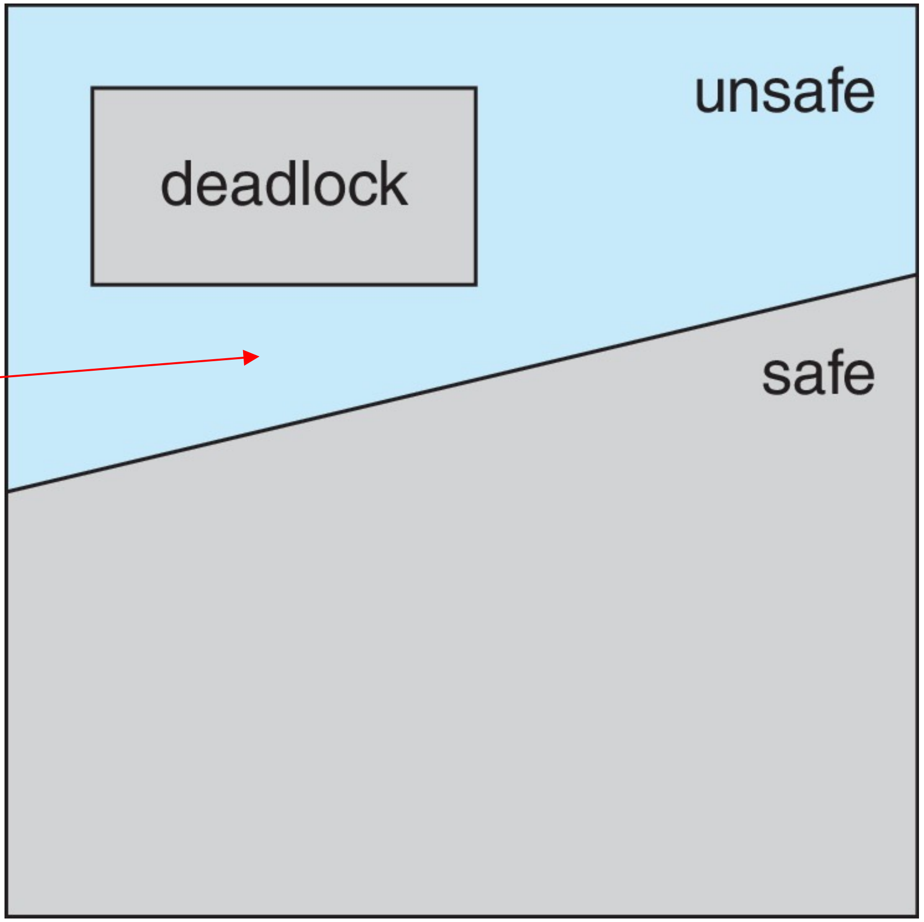
시스템에 있는 모든 프로세스가 각자 필요한
자원을 사용할 수 있는 실행 순서가 존재할 때
우리는 시스템이 안전한 상태에 있다고 말한다.

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.

시스템이 항상 안전한 상태에 남아 있도록 유지함. 즉,
시스템이 안전한 상태를 유지할 수 있을 때만 자원을 할당함.

Safe, Unsafe, Deadlock State



Unsafe state라고 해서 반드시 deadlock이 일어나는 것은 아니다.

Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph

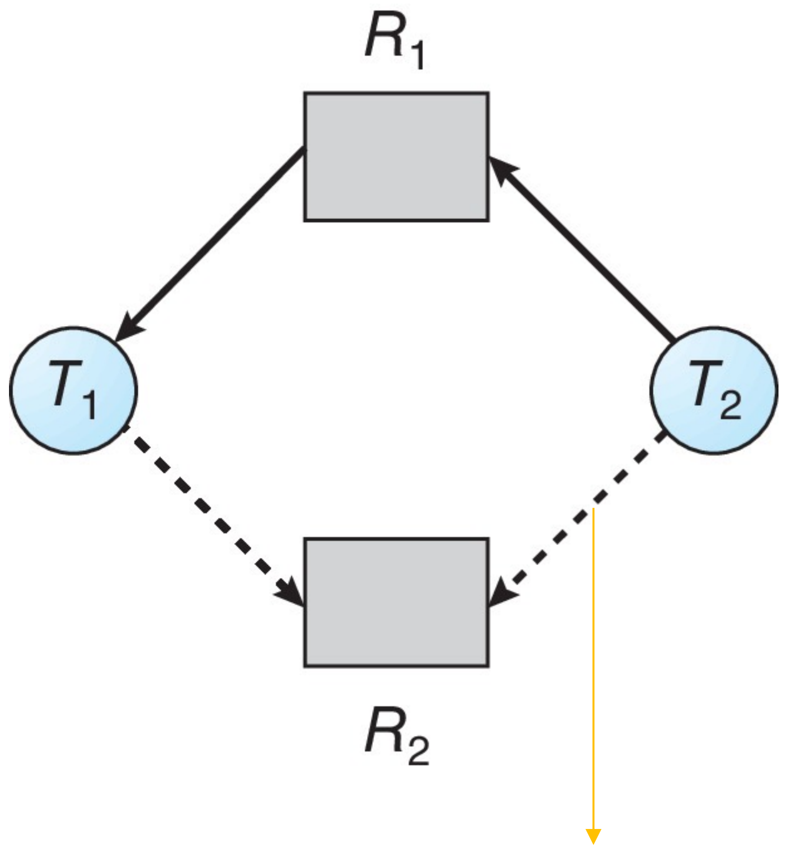
- Multiple instances of a resource type
  - Use the Banker's Algorithm

Resource-Allocation Graph Scheme

P_i가 앞으로 R_j를 요청할지도 모른다

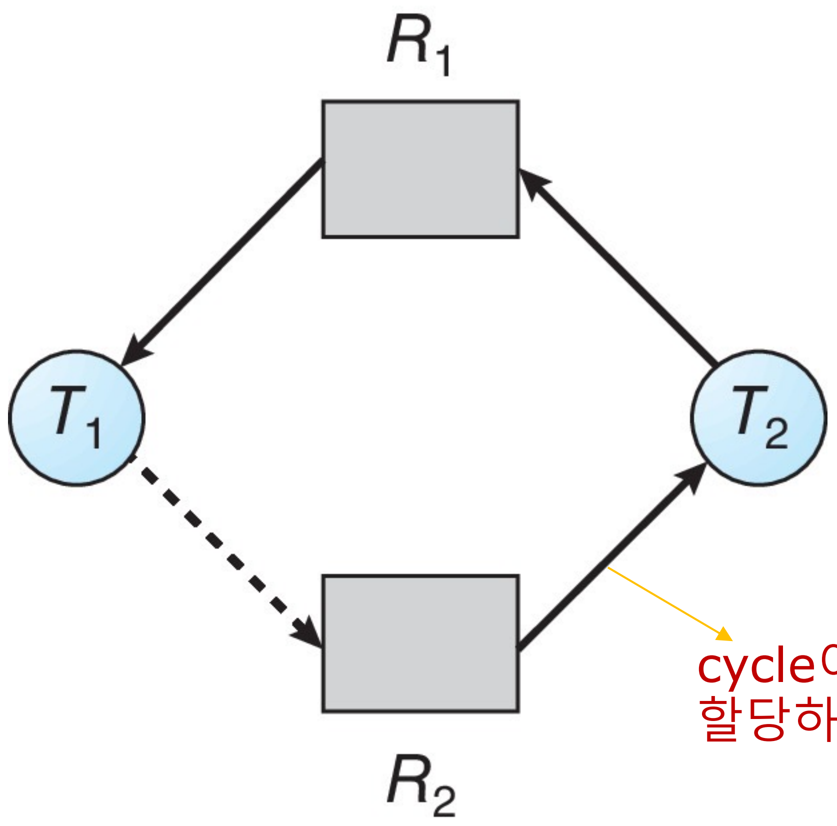
- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system

Resource-Allocation Graph



T_2가 R_2를 요청할 때, 이를 할당하면
cycle이 발생하므로 할당하지 않는다.

Unsafe State In Resource-Allocation Graph



cycle이 발생하므로
할당하지 않는다.

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

각 프로세스는 사용할 자원의 최대치를 사전에 선언해야 한다.
프로세스가 원하는 모든 자원을 확보하면 작업을 마친 후에 모두 반납한다.

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) $\boldsymbol{Need_i \le Work}$

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

프로세스 i가 아직 안 끝났고
필요한 자원이 충분히 있으면
작업을 마칠 수 있으니까
종료 후에 자원을 전부 반납함

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

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

모든 프로세스의 작업을 마칠 수 있으면
시스템은 안전한 상태에 있다

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$_j$***

필요이상? 1. If ***Request$_i$*** $\leq$ ***Need$_i$*** go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

충분한가? 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:

요청한 것을 할당해도 안전한 상태를 유지하나?

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

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

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

안전하면 할당
아니면 대기

- 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

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	

Max와 Allocation으로부터 Need를 구한 후, 이 상태가 안전한 상태인지 검증

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

Available
A B C
3 3 2 $\rightarrow$ P1
5 3 2 $\rightarrow$ P3
7 4 3 $\rightarrow$ P4
7 4 5 $\rightarrow$ P0
7 5 5 $\rightarrow$ P2
10 5 7

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

Example: P_1 Request (1,0,2)

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

앞 상태에서 P1의 요청을 받아들일 수 있나?

	Allocation	*Need*	*Available*	
	A B C	*A B C*	*A B C*	
P_0	0 1 0	7 4 3	2 3 0	→ P1
P_1	3 0 2	0 2 0	5 3 2	→ P3
P_2	3 0 2	6 0 0	7 4 3	→ P4
P_3	2 1 1	0 1 1	7 4 5	→ P0
P_4	0 0 2	4 3 1	7 5 5	→ P2
			10 5 7	

- 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? No, (3,3,0) > (2,3,0)
- Can request for (0,2,0) by P_0 be granted? No, the state is unsafe.

Deadlock Detection

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

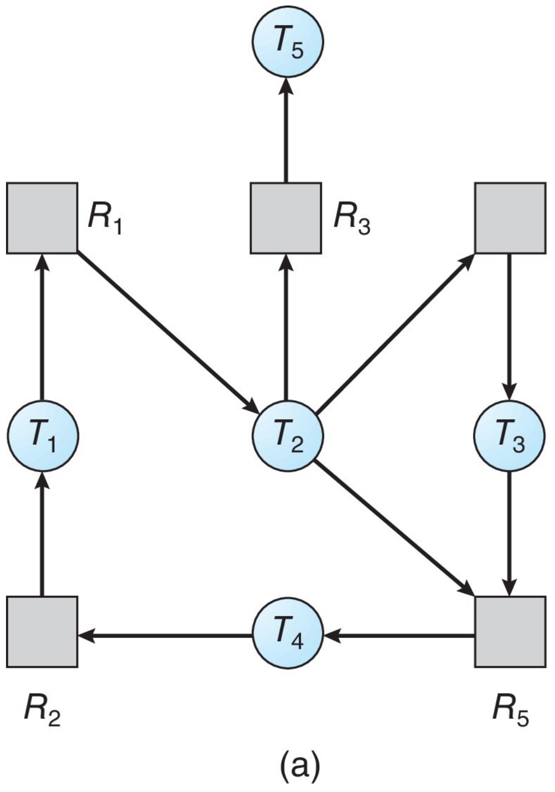
Single Instance of Each Resource Type

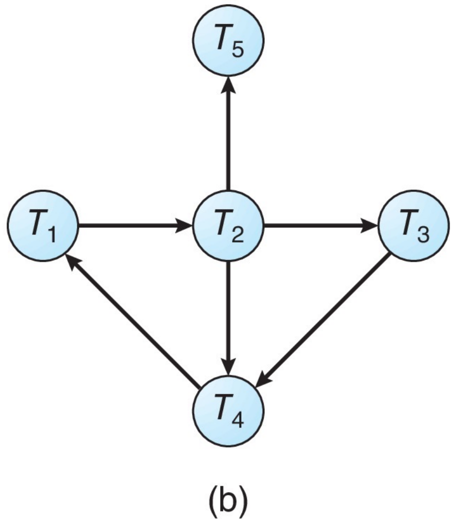
- Maintain wait-for graph → Resource-allocation graph에서 resource를 뺀 것
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j → $P_i \rightarrow P_j$ iff $P_i \rightarrow R_k \wedge R_k \rightarrow P_j$

- 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





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

Deadlock Avoidance와 차이점

Max가 없는 대신 현재 Request를 Need로 간주하고 Banker's 알고리즘을 실행 → unsafe state이면 deadlock으로 판정

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

Detection Algorithm (Cont.)

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

P_i가 더 이상 자원이 필요치 않을 것이라는 낙관적인 가정을 전제함

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

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 → P0
P_1	2 0 0	2 0 2	0 1 0 → P2
P_2	3 0 3	0 0 0	3 1 3 → P3
P_3	2 1 1	1 0 0	5 2 4 → P4
P_4	0 0 2	0 0 2	5 2 6 → P1
			7 2 6

- Sequence <P_0, P_2, P_3, P_4, P_1> 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 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

Avaiable
A B C
0 0 0 → P0
0 1 0 → 더 이상 진행 불가

- 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.

Deadlock detection 알고리즘을 아무 때나 수행하면
deadlock을 유발한 스레드를 식별하지 못할 수 있다.
반면에 자원 요청이 실패할 때 deadlock detection 알고리즘을
수행하면 deadlock을 유발한 스레드를 식별하는데 도움이 된다.

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?

프로세스를 강제로 종료하면 시스템 불일치
상태가 발생할 수 있음에 유의해야 한다.

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 8