고려대학교
KOREA UNIVERSITY

KU-The Future

Operating Systems (10th Ed., by A. Silberschatz)

Chapter 8 Deadlocks

어떤 시스템 환경에서든 항상 발생할 수 있는 문제

Heonchang Yu

Distributed and Cloud Computing Lab.

Contents

- System Model
- Deadlock in Multithreaded Applications
- Deadlock Characterization
- Methods for Handling for Deadlock
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

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.

Introduction

멀티 프로세싱은 서로 다른 코어에서 프로세스를 독립적으로 수행하는 것이고
멀티 프로그래밍은 메모리 내에 여러개의 프로세스들을 로드시켜 동시 또는 *interleaved* 하게 수행할 수 있게 지원하는 것

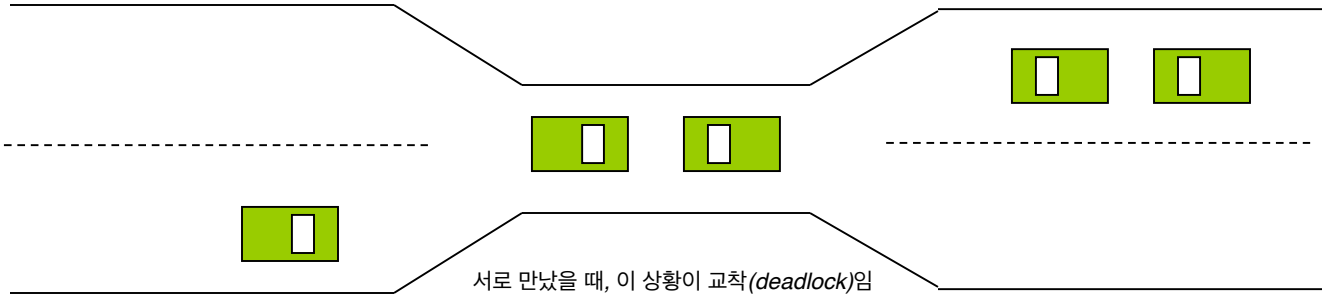
- In a multiprogramming environment
  - A thread requests resources.
    - ✓ If the resources are not available at that time, the thread enters a waiting state.
    - ✓ A waiting thread can never again change state, because the resources it has requested are held by other waiting threads.

      => **Deadlock**

      *hold and wait*하는 상황이 *deadlock*
- Describe methods that application developers as well as operating-system programmers can use to prevent or deal with deadlocks
- Remains the responsibility of programmers to ensure that they design deadlock-free programs

*deadlock-free*한 상황은 쉽게 만들 수 있는 것이 아니다.

Introduction



- A 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). *abort*하는 상황이나 다름 없다.
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

System Model

- A finite number of resources to be distributed among a number of computing threads : memory space, CPU, files
- Under the normal mode of operation, a thread may utilize a resource in only the following sequence:
  - Request
    - ✓ If the request cannot be granted immediately, then the requesting thread must wait until it can acquire the resource.
  - Use
    - ✓ The thread can operate on the resource.
  - Release
    - ✓ The thread releases the resource.
- Examples for request and release of resources
  - `request()` and `release()` of a device, `open()` and `close()` of a file, and `allocate()` and `free()` memory system calls
  - `wait()` and `signal()` operations on semaphores

Deadlock in Multithreaded Applications

- How deadlock can occur in a multithreaded Pthread program using POSIX mutex locks

2개의 *semaphore*를 사용하는 상황
first_mutext & second_mutex

여기서 스레드를 사용하고 있지만, 너무 민감하게 "스레드"라고 못박을 필요는 없고, 프로세스로 이해해도 무방하다

  - Figure 8.1 – Mutex locks are acquired and released using `pthread_mutex_lock()` and `pthread_mutex_unlock()`

```
/* 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 Applications

- Livelock 얘도 진행이 안되는 것은 동일하지만, 다시 전 단계로 넘어가서 진행하는 것처럼 보이게 하는 것
  *deadlock*은 진행이 안되 멈추는 것
  - Whereas deadlock occurs when every thread in a set is blocked waiting for an event that can be caused only by another thread in the set, livelock occurs when a thread continuously attempts an action that fails. 스레드들끼리 간섭하는 상황이 발생하게 된다.
    - ✓ Livelock is similar to what happens when two people attempt to pass in a hallway: One moves to his right, the other to her left, still obstructing each other's progress.
  - Can be illustrated with the Pthreads `pthread_mutex_trylock()` function, which attempts to acquire a mutex lock without blocking
    - ✓ Can lead to livelock if `thread_one` acquires `first_mutex`, followed by `thread_two` acquiring `second_mutex`.
    - ✓ Each thread then invokes `pthread_mutex_trylock()`, which fails, releases their respective locks, and repeats the same actions indefinitely.

Deadlock in Multithreaded Applications

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
      int done = 0;

      while (!done) {
            pthread_mutex_lock(&first_mutex);
            if (pthread_mutex_trylock(&second_mutex)) {
                  /**
                    * Do some work
                    */
                  pthread_mutex_unlock(&second_mutex);
                  pthread_mutex_unlock(&first_mutex);
                  done = 1;
            }
            else
                  pthread_mutex_unlock(&first_mutex);
      }
      pthread_exit(0);
}
```

*first_mutex 1->0*으로 변경하는 과정

여기서 *second_mutex = 0*이기 때문에 *else* 구문 실행

first_mutex 부분을 재실행하게 되면서 *loop*에 빠짐

```
/* thread_two runs in this function */
void *do_work_two(void *param)
{
      int done = 0;

      while (!done) {
            pthread_mutex_lock(&second_mutex);
            if (pthread_mutex_trylock(&first_mutex)) {
                  /**
                    * Do some work
                    */
                  pthread_mutex_unlock(&first_mutex);
                  pthread_mutex_unlock(&second_mutex);
                  done = 1;
            }
            else
                  pthread_mutex_unlock(&second_mutex);
      }

      pthread_exit(0);
}
```

*second_mutex 1->0*으로 변경하는 과정

여기서 *first_mutex = 0*이기 때문에 *else* 구문 실행

second_mutex 부분을 재실행하게 되면서 *loop*에 빠짐

Deadlock Characterization

– A deadlock situation can arise if the following four conditions hold simultaneously in a system.

mutual exclusion : non-shared

- **Mutual exclusion:** At least one resource must be held in a nonsharable mode; that is, only one thread at a time can use the resource.

리퀘스트를 했는데 자원을 바로 할당하지 못하는 경우 —> wait

- **Hold and wait:** A thread must be holding at least one resource and waiting to acquire additional resources that are currently being held by other threads.
- **No preemption:** Resources cannot be preempted.
- **Circular wait:** A set $\{T_0, T_1, \ldots, T_n\}$ of waiting threads must exist such that T_0 is waiting for a resource held by T_1, T_1 is waiting for a resource held by T_2, …, T_{n-1} is waiting for a resource held by T_n, and T_n is waiting for a resource held by T_0.

deadlock을 발생시킬 수 있는 상황이다

Resource–Allocation Graph

- A finite number of resources to be distributed among a number of computing processes : memory space, CPU, files
- Graph G = (V, E)
  - The set of vertices V is partitioned into two different types of nodes:
    - ✓ $P = \{T_0, T_1, \ldots, T_n\}$, the set consisting of all the active threads 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 $T_i \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$

P 대신 *T(Thread)*라 표현하는 것이 맞다 *(*책의 신판*)*

*resource*들의 집합

스레드가 리소스 쪽에 자원을 요청하는 과정

리소스가 스레드에 할당되는 과정

Resource–Allocation Graph

- Represent each thread T_i as a circle and each resource type R_j as a rectangle
- Represent each such instance as a dot within the rectangle

각각의 스레드들 입장에서는 *hold & wait*
전체 상황에선 *circular wait* 상황이라 보면 될 것이다.

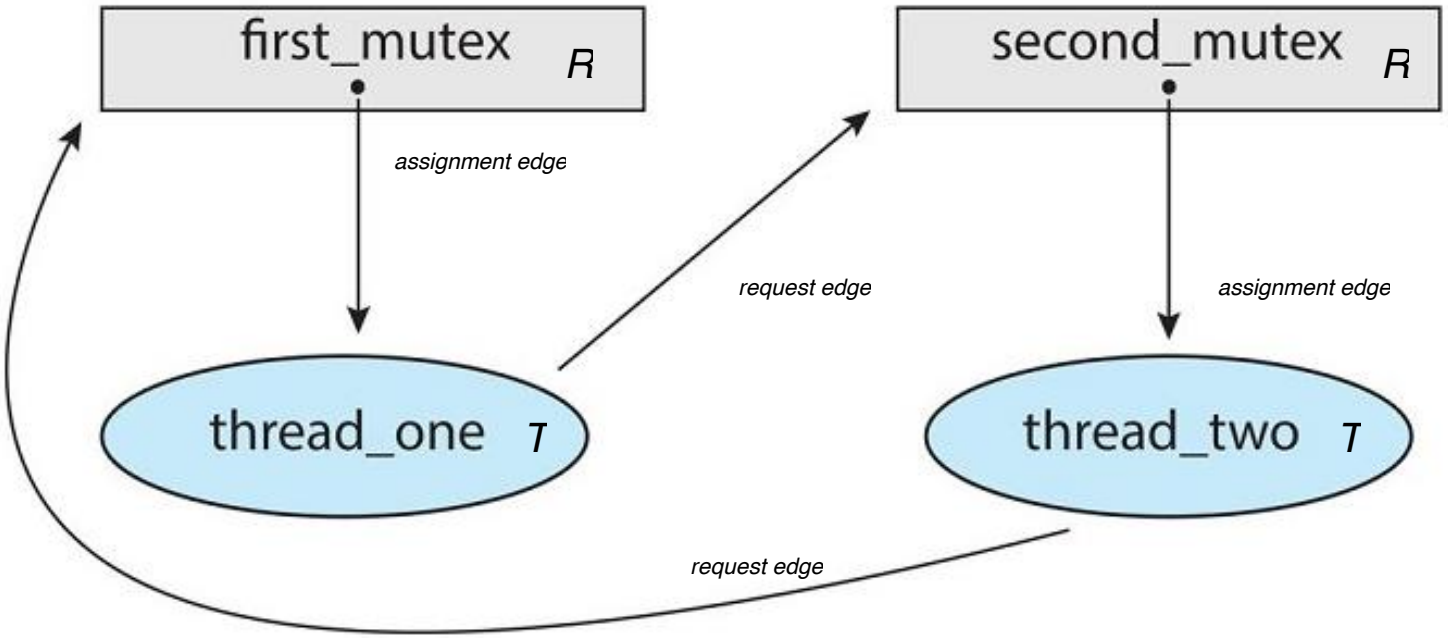
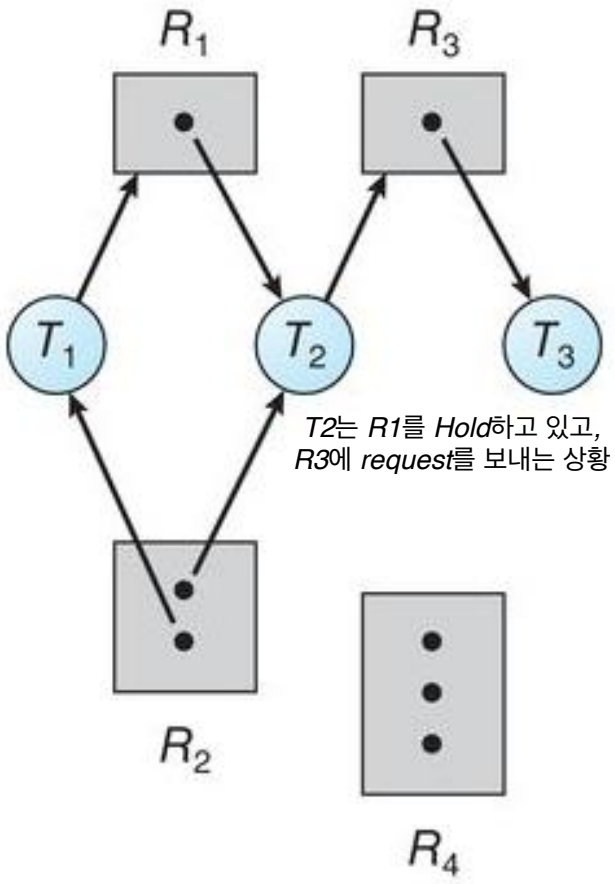


Figure 8.3 Resource-allocation graph for program in Figure 8.1.

Resource–Allocation Graph

- The sets T, R, and E:
  $T = \{T_1, T_2, T_3\}$
  $R = \{R_1, R_2, R_3, R_4\}$
  $E = \{T_1 \rightarrow R_1, T_2 \rightarrow R_3, R_1 \rightarrow T_2, R_2 \rightarrow T_2, R_2 \rightarrow T_1, R_3 \rightarrow T_3\}$

*T1*은 *R2*를 *Hold*하고 있고,
*R1*에 *request*를 보내는 상황



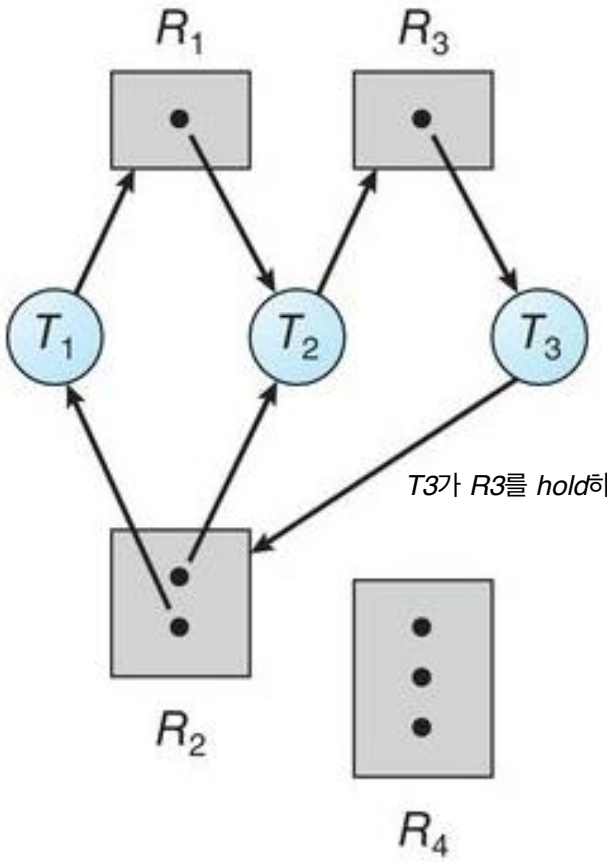
*T2*는 *R1*를 *Hold*하고 있고,
*R3*에 *request*를 보내는 상황

Figure 8.4 Resource-allocation graph.

Resource–Allocation Graph

– Given the definition of a resource-allocation graph
  - If the graph contains no cycles, then no thread in the system is deadlocked. *cycle*이 없으면 스레드 사이에 *deadlock*이 없다
  - If the graph does contain a cycle, then a deadlock may exist. *cycle*자체가 항상 *deadlock*을 표현하는 건 아니다
  - If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. 인스턴스가 한개만 존재할 수도, 그 이상 존재할 수도 있음 여기서 *cycle*은 무조건 *deadlock* 상황을 의미함
  - If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. *cycle*이 항상 *deadlock*을 만든다고 볼 수 없음
  - In figure 8.4, suppose that thread T_3 requests an instance of resource type R_2.
    - ✓ Since no resource instance is currently available, a request edge $T_3 \rightarrow R_2$ is added to the graph. (figure 8.5)
    - ✓ Two cycles (T_1, T_2, and T_3 are deadlocked.) *13*페이지의 그림에서 없었던 부분을 추가
      - ❖ $T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$
      - ❖ $T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_2$

Resource–Allocation Graph



*T3*가 *R3*를 *hold*하고, *R2*에 *request*하는 상황

Figure 8.5 Resource-allocation graph with a deadlock.

Resource–Allocation Graph

- In figure 8.6, 인스턴스가 한 개만 있으면 그 *cycle*은 무조건 *deadlock*
  여러개를 갖는 구조에서는 *cycle*이 있더라고 *deadlock*이 있을 수도, 없을 수도 있음
  - ✓This figure shows a cycle, 지금은 *deadlock*처럼 보이지만, *deadlock*이 추후에 해결될 수 있는 상황임 *T2* 작업이 마무리되면, 본인에게 할당된 자원을 *release*하면서 *T1*에 자원을 재할당 할 수 있음

$$T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$$

  - ✓However, there is no deadlock.
- If a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state.
- If there is a cycle, then the system may or may not be in a deadlocked state.

리소스 내의 점들은 인스턴스를 의미함

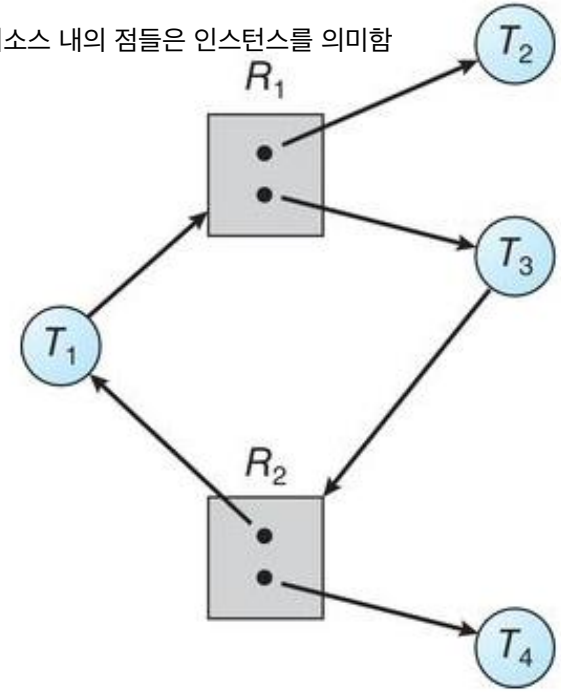


Figure 8.6 Resource-allocation graph with a cycle but no deadlock.

Methods for Handling Deadlocks

- Dealing with the deadlock problem
  - Using a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state
  - Allowing the system to enter a deadlocked state, detect it, and recover
  - Ignoring the problem and pretending that deadlocks never occur in the system – used by most operating systems, including UNIX and Windows

Deadlock Prevention

*deadlock*이 발생할 수 있는 *4*개의 상황을 방지하면, *deadlock*이 발생하는 것을 막을 수 있을 것이다.

- Mutual Exclusion

*mutual exclusion*을 반드시 써야하는 애플리케이션이 아닌 경우,
*mutual exclusion*을 사용하지 않음으로써 *deadlock*을 방지할 수 있다.
리소스에 접근할 수 있는 것을 *2*개 이상 만들 수 있는 환경이 된다면?

  - Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock.
  - Read-only files are a good example of a sharable resource.

- Hold and Wait

*(deadlock*이 되는 상황에서는 *2*개 다 만족되어야한다.*)*
hold 또는 *wait* 중 하나만 만족하게 하게 하는 상황을 만든다

  - must guarantee that, whenever a thread requests a resource, it does not hold any other resources.

실행 전에 모든 필요로하는 리소스를 미리 다 할당해주는 방식 *(*미리 *hold*만 하는 상황*)*

    - ✓ Requires each thread to request and be allocated all its resources before it begins execution.
    - ✓ allows a thread to request resources only when it has none.

*wait*만 하는 상황

  - Disadvantages

이용률이 떨어질 수 있다

    - ✓ Resource utilization may be low.
    - ✓ Starvation is possible.

Deadlock Prevention

완벽한 상황은 *deadlock*을 *0*으로 만드는 것인데, 사실상 불가능하다.

첫번째, 두번째 기준은 관점만 다르지
내용이 거의 비슷함

– No Preemption

내가 리소스를 갖고 있으면서, 추가적인 리소스를 요청했음
그런데 이 요청 시, 내가 갖고 있는 모든 자원을 *Preemptive*로 뺏김. 내 자원이 *preemptive*된다.

- If a thread is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources the thread is currently holding are preempted.
  - ✓ The preempted resources are added to the list of resources for which the thread is waiting.
  - ✓ The thread will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- Alternatively, if a thread requests some resources, we first check whether they are available. 내가 요청한 자원을 또다른 스레드가 갖고 있는데, 해당 스레드로부터 리소스를 뺏어오는 것
  - ✓ If they are, we allocate them.
  - ✓ If they are not, we check whether they are allocated to some other thread that is waiting for additional resources. If so, we preempt the desired resources from the waiting thread and allocate them to the requesting thread. *some other thread*가 *waiting*하고 있는 상황이다

Deadlock Prevention

– Circular Wait *Non-Circular wait == Linear Wait*으로 만들어주는 것임

- impose a total ordering of all resource types and require that each thread requests resources in an increasing order of enumeration.
- The set of resource types, R={R_1, R_2, …, R_m}
  - ✓ Assign to each resource type a unique integer number
  - ✓ Define a one-to-one function $F: R \rightarrow N$ *integer number N* 부여
  - ✓ A thread can initially request any number of instances of a resource R_i.
    - ❖ After that, the thread can request instances of resource R_j if and only if $F(R_j) > F(R_i)$. 갖고 있는 리소스보다 큰 것에 대해 요청하는 형태
    - ❖ A thread requesting an instance of resource type R_j must have released any resources R_i such that $F(R_i) > F(R_j)$.

*Rj*보다 큰 *Ri* 값들을 모두 *release*
*Cycle*을 파괴함

Deadlock Avoidance

교통 체증 시간 때 해당 도로를 피해가는 것처럼
*deadlock*을 피하기 위해서는 미리 알아야할 정보들이 있다
미리 알고 피하는 것은 이론적으로 정의할 수 있어도, 실제 적용하기는 매우 어렵다

- Require additional information about how resources are to be requested
- With knowledge of the complete sequence of requests and releases for each thread, the system can decide for each request whether or not the thread should wait in order to avoid a possible future deadlock.
  - ✓ In making this decision the system consider the resources currently available, the resources currently allocated to each thread, and the future requests and releases of each thread.

*future request*를 알기 정말 어렵다

- The simplest and most useful model requires that each thread declare the maximum number of resources of each type that it may need.
- A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist.
- The resource-allocation state is defined by the number of available and allocated resources and the maximum demands of the threads.

Deadlock Avoidance

– Safe state

*safe*하다 —> 데드락이 발생하지 않는 안전한 상황이다.

- A system is in a safe state only if there exists a safe sequence.
  - ✓ A sequence of threads $< T_1, T_2, \ldots, T_n >$ is a safe sequence for the current allocation state if, for each T_i, the resource requests that T_i can still make can be satisfied by the currently available resources plus the resources held by all T_j, with $j < i$.
  - ✓ If the resources that T_i needs are not immediately available, then T_i can wait until all T_j have finished.
- A safe state is not a deadlocked state.
- A deadlocked state is an unsafe state.
- An unsafe state may lead to a deadlock.

unsafe 상황이 항상 *deadlock*을 유발하는 것은 아니다

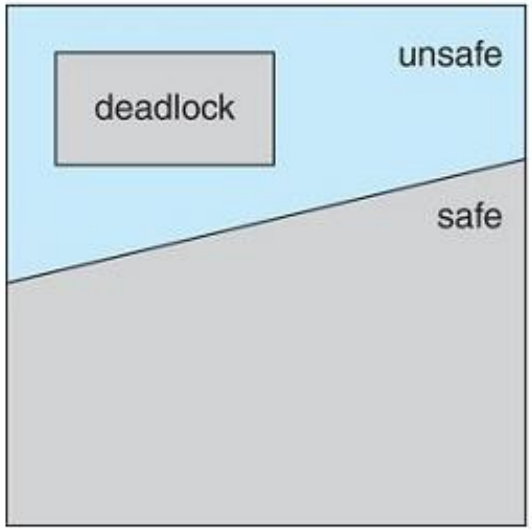


Figure 8.8 Safe, unsafe, and deadlocked state spaces.

Deadlock Avoidance

– Resource-Allocation-Graph Algorithm 단순한 경우에만 그래프 알고리즘으로 그려본다
*request & assignment edge*는 동일하게 존재

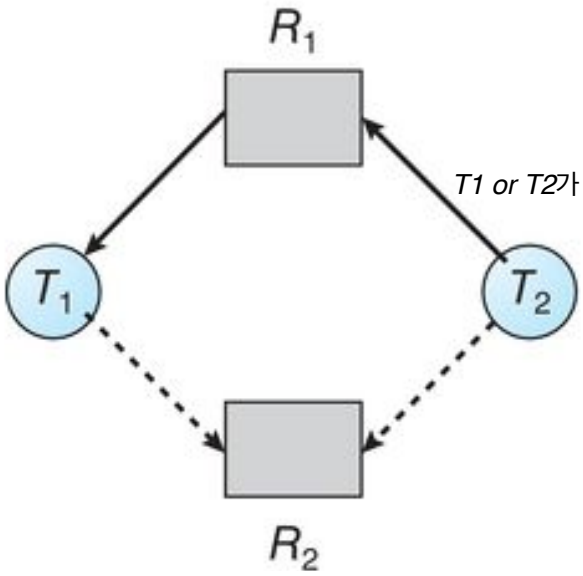
- A claim edge $T_i \rightarrow R_j$: thread T_i may request resource R_j at some time in the future. *claim edge*는 점선 형태로 요청하는 형태다.
- A claim edge resembles a request edge in direction but is represented in the graph by a dashed line.
- When thread T_i requests resource R_j, the claim edge $T_i \rightarrow R_j$ is converted to a request edge.
- When a resource R_j is released by T_i, the assignment edge $R_j \rightarrow T_i$ is reconverted to a claim edge $T_i \rightarrow R_j$.
- The resources must be claimed a priori in the system.
- Suppose that thread T_i requests resource R_j.
  - ✓ The request can be granted only if converting the request edge $T_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow T_i$ does not result in the formation of a cycle in the resource-allocation graph.

*Request edge -> assignment edge*로 허용하느냐?
여기서 중요한 건 *cycle*을 형성하면 안된다는 점

Deadlock Avoidance

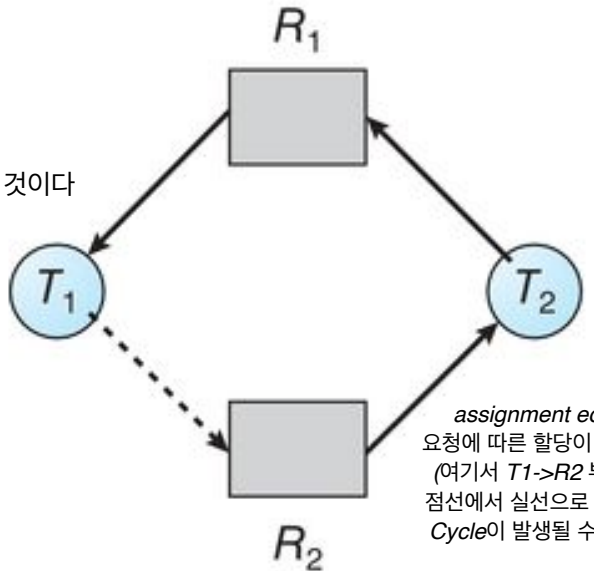
- Suppose that T_2 requests R_2 .
- Although R_2 is currently free, we cannot allocate it to T_2, since this action will create a cycle in the graph.
- A cycle indicates that the system is in an unsafe state.

리소스에서 인스턴스 표현은 불필요하다
R1 리소스가 *T1*에 할당된 상태



*T1 or T2*가 미래에 *R2*에 자원을 요청할 것이다

Figure 8.9 Resource-allocation graph for deadlock avoidance.



assignment edge
요청에 따른 할당이 되었음
*(*여기서 *T1->R2* 부분이
점선에서 실선으로 바뀌면
*Cycle*이 발생될 수 있음*)*

Figure 8.10 An unsafe state in a resource-allocation graph.

Deadlock Avoidance

– Banker's Algorithm

인스턴스가 2개 이상이 되면 그래프로 관계를 표현하는 과정이 매우 어려워지고 복잡하다.
그래서 간단하게 표현하기 위해 인스턴스를 1개로 가정해 그래프 작성하게 된다.

- The resource-allocation-graph algorithm is not applicable to a resource-allocation system with multiple instances of each resource type.
- When a new thread enters the system, it must declare the maximum number of instances of each resource type that it may need.
- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state.
  - ✓ If it will, the resources are allocated.
  - ✓ Otherwise, the thread must wait until some other thread releases enough resources.

Deadlock Avoidance

– Banker's Algorithm

- Let *n* be the number of threads and *m* be the number of resource types.

n : 스레드의 수
m : 리소스의 수

각 키워드들이 변수 이름이다

✓ **Available** : A vector of length *m* indicates the number of available resources of each type. If available[*j*] equals *k*, then *k* instances of resource type R_j are available.

✓ **Max** : An *n x m* matrix defines the maximum demand of each thread. If *Max*[*i*][*j*] equals *k*, then thread T_i may request at most *k* instances of resource type R_j .

각 프로세스에서 요구하는 최대 요구치

현재*allocated*된 *(=hold*하고 있는*)* 리소스를 얼마나 들고 있느냐?

✓ **Allocation** : An *n* x *m* matrix defines the number of resources of each type currently allocated to each thread. If *Allocation*[*i*][*j*] equals *k*, then *thread* T_i is currently allocated *k* instances of R_j .

✓ **Need** : An *n* x *m* matrix indicates the remaining resource need of each thread. If *Need*[*i*][*j*] equals *k*, then thread T_i may need *k* more instances of resource type R_j to complete its task.

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

추가적으로 얼마나 더 필요한가를 따지는 게 *need* 값

Deadlock Avoidance

– Safety Algorithm : the algorithm for finding out whether or not a system is in a safe state

"현재" 할당된 리소스 정보들을 기준으로 *deadlock*이 발생할지, 아니면 *deadlock*을 우회하기 위한 순서를 정하는 알고리즘

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively.
   Initialize *Work* = *Available* and *Finish*[*i*] = *false* for *i* = 0,1,..., *n-1*.
2. Find an index *i* such that both

   work : 가용할 수 있는 인스턴스 수로 초기화
   Finish : 각 스레드들이 작업을 끝낼 수 있는지 판단

   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.

*safety algorithms*에서는 *safe sequence*를 구하는 과정이다.

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

Deadlock Avoidance

- Resource-Request Algorithm

추가적으로 어떤 스레드가 특정 리소스(자원)를 요청했을 때, 할당을 해줄지 말지를 결정하는 알고리즘

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the thread has exceeded its maximum claim.

이 조건을 만족 못하면 에러이기 때문에 종료

2. If $Request_i \leq Available$, go to step 3. Otherwise, T_i must wait, since resources are not available.
3. Have the system pretend to have allocated the requested resources to thread T_i by modifying the state as follows:

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

요청한 값을 만족한다고 가정하고 *27*페이지의 *safety algorithm*을 수행하게 된다

- If its state is safe, the transaction is completed, and thread T_i is allocated its resources.
- Otherwise(unsafe), T_i must wait for $Request_i$, and the old resource-allocation state is restored.

Deadlock Avoidance

- Example of Banker's Algorithm

  n : 스레드 수 -> *5*개
  m : 리소스 수 -> *3*개

  - Consider a system with 5 threads T_0 through T_4 and 3 resource types *A* (10 instances), *B* (5 instances, and *C* (7 instances).

  전체 시스템에서 보유하고 있는 자원의 수는 *10, 5, 7* 개다
  *10, 5, 7*개를 넘어가는 상황이 발생하면 에러다
  이 갯수를 잘 활용해서 스레드들이 작업을 해야한다
  자원 수 = *Allocation* 전체 합 + *available* 전체 합

  - The current state of the system

    ✓ *Need = Max - Allocation*

  available : 현재 남아있는 자원 수
  할당되고 남은 값

  *Max - Allocation*의 결과 값

	Allocation	*Max*	*Available*	*Need*
	A B C	*A B C*	*A B C*	*A B C*
T_0	0 1 0	7 5 3	3 3 2	7 4 3
T_1	2 0 0	3 2 2		1 2 2
T_2	3 0 2	9 0 2		6 0 0
T_3	2 1 1	2 2 2		0 1 1
T_4	0 0 2	4 3 3		4 3 1

T1 : 2 0 0

세로축으로 합은
전혀 관계 없음

각 스레드 별로 최대 필요한
자원의 수를 안다고 가정하고 계산

모든 *i*값에 대해서 *need_i*가
*available_i*보다 작아야한다

여기서 찾아야하는 것
Needi <= Available
만약 없으면 -> *sequence*가
끝나게 된다

  - The system is in a safe state: the sequence < T_1, T_3, T_4, T_2, T_0> satisfies the safety criteria.

*sequence*는 여러가지 상황이 발생할 수 있다
*T3*로 시작하는 *sequence*도 문제 없다

일단 여기서는 각 스레드들의 *Finish* 상태는 모두 *False*라고 가정하고 진행 중이다

모든 *sequence*를 구하면, 결국 전체 시스템이 보유한
자원수의 갯수와 동일해진다.

Deadlock Avoidance

- Example of Banker's Algorithm
  - Thread T_1 requests $Request_1=(1, 0, 2)$. Check that $Request_1 \leq$ Available. (that is, (1,0,2) ≤ (3,3,2), which is true)

*need*보다 작은 값을 보통 요청할 것이다. *request* 양도 조심해야한다

전체 시스템의 데드락을 야기할 수 있는 상황이 발생할 수도 있음
무조건적인 *allocate*를 피하자

*2 0 0*에서 *1 0 2* 만큼
추가 요청이 들어와서 할당된 값이 변경됨

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
T_0	0 1 0	7 4 3	2 3 0
T_1	3 0 2	0 2 0	
T_2	3 0 2	6 0 0	
T_3	2 1 1	0 1 1	
T_4	0 0 2	4 3 1	

30 페이지의 *available*에서 *1 0 2* 만큼 빼서
3 3 2 에서 변경된 값이다

*available*을 *work*로 보고 *work*보다 작은 *need*를 찾는다.

*3 2 2*에서 *3 0 2*만큼을 뺐다 *(Max - Allocation* 값*)*

  - We execute our safety algorithm and find that the sequence < T_1, T_3, T_4, T_0, T_2> satisfies the safety requirement.
  - When the system is in this state, a request for (3, 3, 0) by T_4 cannot be granted, since the resources are not available.
  - A request for (0, 2, 0) by T_0 cannot be granted. (unsafe)

Deadlock Detection

- If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then deadlock situation may occur.
  - ✓ An algorithm that examines the state of the system to determine whether a deadlock has occurred.
  - ✓ An algorithm to recover from the deadlock
- Systems

*Deadlock & Fail*이 발생해도 *recovery*는 필요하다.
절대 쉬운 과정은 아니다

  - ✓ With only a single instance of each resource type
  - ✓ With several instances of each resource type

Deadlock Detection

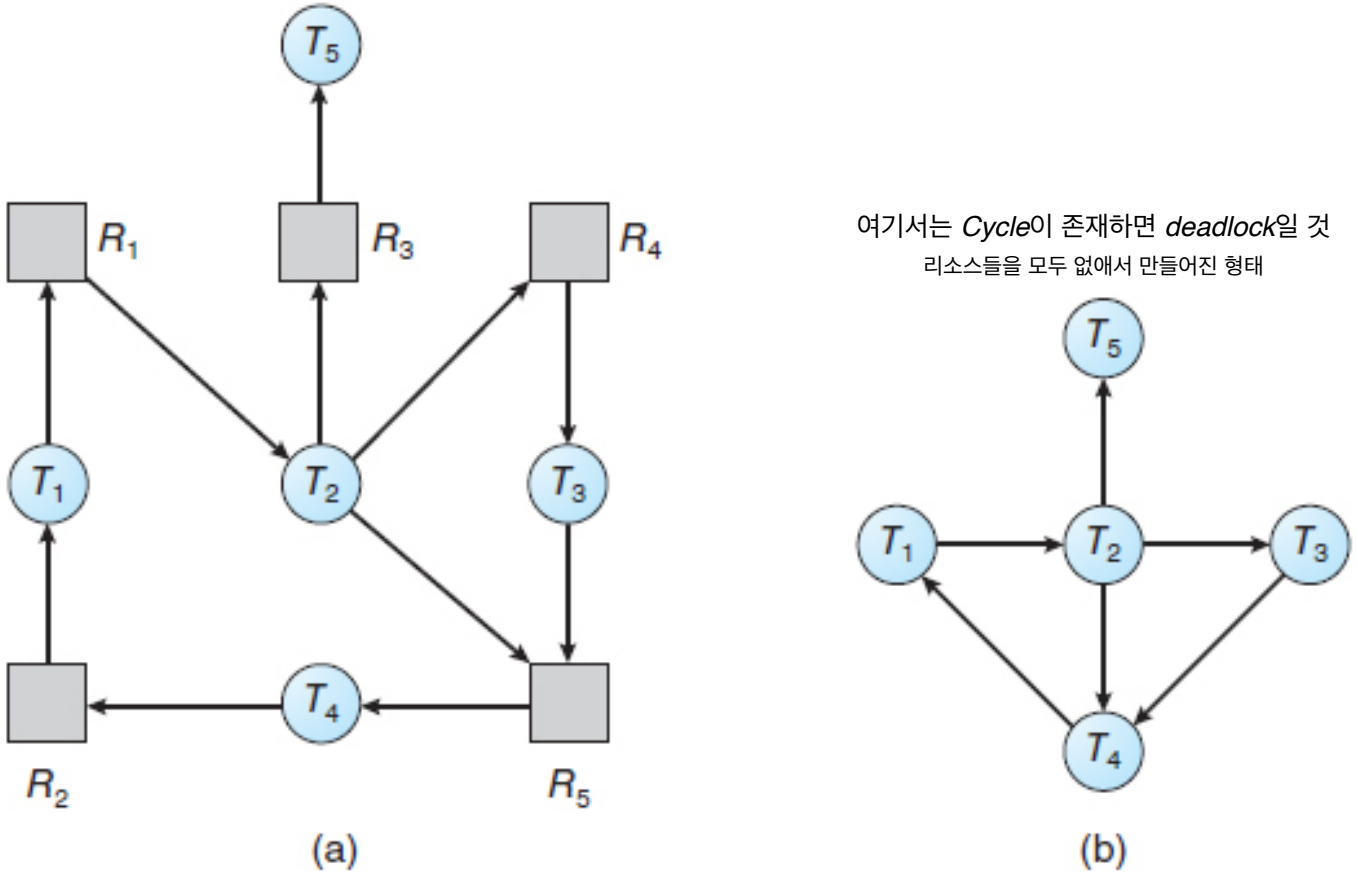
단순히 스레드들 간에 어떤 스레드가 갖고 있는 리소스를 또 다른 어떤 리소스가 갖고 있는지 확인하는 과정이다.

- Single Instance of Each Resource Type
  - A deadlock-detection algorithm that uses a wait-for graph

    리소스 노드를 간단하게 제거하고 표현한 그래프
    - ✓ An edge from T_i to T_j : thread T_i is waiting for thread T_j to release a resource that T_i needs.
    - ✓ $T_i \rightarrow T_j$ in a wait-for graph : $T_i \rightarrow R_q$ and $R_q \rightarrow T_j$ in a resource-allocation graph
  - A deadlock exists in the system if and only if the wait-for graph contains a cycle.
  - An algorithm to detect a cycle in a graph requires $O(n^2)$ operations, where n is the number of vertices in the graph.

    *Worst Case*로 가정 시 연산 속도

Deadlock Detection



여기서는 *Cycle*이 존재하면 *deadlock*일 것

리소스들을 모두 없애서 만들어진 형태

Figure 8.11 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

Deadlock Detection

이론에서 머무르는 것이 아니라 현실에서도 충분히 사용할 수 있을 것으로 보임

– Several Instances of a Resource Type

- The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.
- Data structures 여기서는 *Need*가 아닌 *request* 값을 기준으로 보면 된다
  - ✓ **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 thread.
  - ✓ **Request** : An *n* x *m* matrix indicates the current request of each thread. If *Request*[*i*][*j*] equals *k*, then thread T_i is requesting *k* more instances of resource type R_j.

Deadlock Detection

– Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize *Work = Available*. For $i = 0,1,\ldots, n-1$, 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, $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if $Finish[i] == false$, then thread T_i is deadlocked.

※ This algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked state.

Deadlock Detection

– Example of Detection Algorithm

- Consider a system with 5 threads T_0 through T_4 and 3 resource types A (7 instances), B (2 instances, and C (6 instances). 전체 인스턴스의 갯수
- The current state of the system

가용 자원이 *allocation* 값과 동일함
*A, B, C*의 합보다 작거나 같을 때 성립하기 때문에 남은 가용 자원은 *0*임

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
T_0	0 1 0	0 0 0	0 0 0
T_1	2 0 0	2 0 2	
T_2	3 0 3	0 0 0	
T_3	2 1 1	1 0 0	
T_4	0 0 2	0 0 2	

전부 다 할당된 상태다
*available*이 모두 *0* 이라고 *deadlock*은 아니다
*request <= available*이 있으면 *deadlock*을 피하는 것이 가능

- The system is not in a deadlocked state: the sequence $< T_0, T_2, T_3, T_1, T_4 >$ results in *Finish*[*i*] == *true* for all *i*.

*sequence*는 변경될 수 있음
다만, *sequence*를 구할 수 있는가 없는가를 따지는 것이다

Deadlock Detection

– Example of Detection Algorithm

- Suppose that thread T_2 makes one additional request for an instance of type C.

마찬가지로 추가 *request*가 있다면?

*available*과 비교하지 말고, *request*를 새로 구해 작업하는 과정을 진행하면 된다

	Request
	A B C
T_0	0 0 0
T_1	2 0 2
T_2	0 0 1
T_3	1 0 0
T_4	0 0 2

- The system is deadlocked.
- Although we can reclaim the resources held by thread T_0, the number of available resources is not sufficient to fulfill the requests of the other threads.

Deadlock Detection

– Detection-Algorithm Usage

- When should we invoke the detection algorithm?
  - ✓ How often is a deadlock likely to occur? 요청에 대해 만족을 시켜야하는지, 말아야하는지 만족을 시킨다 –> 자원이 줄어든다
  - ✓ How many threads will be affected by deadlock when it happens? *deadlock* 발생 시, 영향을 받는 스레드는 몇 개나 되는가?
- If deadlocks occur frequently, then the detection algorithm should be invoked frequently.
- Deadlocks occur only when some thread makes a request that cannot be granted immediately.
- In the extreme, we can invoke the deadlock-detection algorithm every time a request for allocation cannot be granted immediately.
- Invoking the deadlock-detection algorithm for every resource request will incur considerable overhead in computation time.

Recovery from Deadlock

작업을 중단시키는 것 등에 대한 개입 정도?

- When a detection algorithm determines that a deadlock exists
  - Is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually
  - Is to let the system recover from the deadlock automatically

Recovery from Deadlock

– Process and Thread Termination *abort : 중단시키다*

결국 *abort*를 시켜야함

- To eliminate deadlocks by aborting a process or thread
  - ✓ Abort all deadlocked processes.
    - ➢ This method clearly will break the deadlock cycle, but at great expense.
  - ✓ Abort one process at a time until the deadlock cycle is eliminated.
    - ➢ After each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.
- Many factors may affect which process is chosen,
  - ✓ What the priority of the process is
  - ✓ How long the process has computed and how much longer the process will compute before completing its designated task
  - ✓ How many and what types of resources the process has used
  - ✓ How many more resources the process needs in order to complete
  - ✓ How many processes will need to be terminated

Recovery from Deadlock

– Resource Preemption
  - Selecting a victim
    - ✓ Which resources and which processes are to be preempted?
    - ✓ We must determine the order of preemption to minimize cost.
  - Rollback
    - ✓ If we preempt a resource from a process, it cannot continue with its normal execution.
    - ✓ So we must roll back the process to some safe state and restart it from that state.
    - ✓ Total rollback
  - Starvation
    - ✓ How can we guarantee that resources will not always be preempted from the same process?