

# 7장. 교착 상태 (Deadlock)

## 순천향대학교 컴퓨터공학과 이 상 정

순천향대학교 컴퓨터공학과

1

운영체제

## 강의 목표 및 내용

### □ 목표

- 병행 프로세스의 집합이 작업을 완료할 수 없도록 하는 **교착 상태**의 표현과 기술 방법 소개
- 컴퓨터 시스템에서의 교착 상태를 **예방**하거나 **회피**하는 여러 방법들 소개

### □ 내용

- 시스템 모델
- 교착 상태의 특징
- 교착 상태 처리 방법
- 교착 상태 예방
- 교착 상태 회피
- 교착 상태 탐지
- 교착 상태에서부터 회복

순천향대학교 컴퓨터공학과

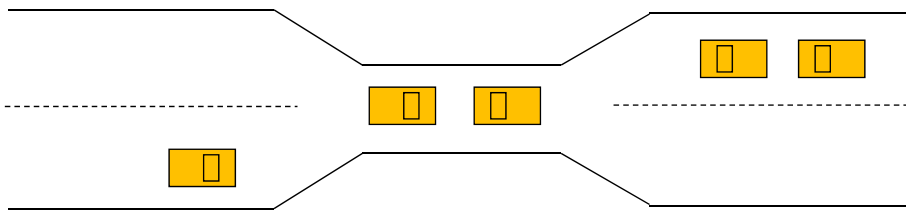
2

7. 교착 상태

## 교착 상태 문제 (Deadlock Problem)

- 프로세스가 한 **자원을 점유**하고 **봉쇄되고(blocking)**, 다른 프로세스가 이 자원을 획득하기 위해 요청 후 기다린다면 이 프로세스들은 **교착 상태(deadlock)**에 있고 함
- 예
  - 하나의 프린터와 하나의 DVD 드라이브가 있는 시스템
  - 프로세스  $P_i$ 는 DVD 드라이브를 점유하고, 프로세스  $P_j$ 는 프린터를 점유한다고 가정
  - $P_i$ 가 프린터를 요청하고,  $P_j$ 가 DVD 드라이브를 요청한다면 교착 상태가 발생
  - 각각은 프린터와 DVD를 방출(release)하는 사건을 기다리는데, 이 사건은 다른 대기 중인 프로세스들 중의 어느 하나에 의해서만 발생이 가능

## 교량 통과 예



- 교량에서는 한 방향으로만 운행
- 교량의 각 부분은 자원으로 간주
- 교착 상태 발생 시 한 자동차가 후진해야만 해결 (**자원의 선점과 롤백(rollback)**)
- 교착 상태 발생 시 여러 대의 차량이 후진할 수도 있음
- **기아 상태(starvation)** 발생 가능성
- 대부분의 운영체제가 교착 상태를 방지하거나 다루지 못함

## 시스템 모델 (System Model)

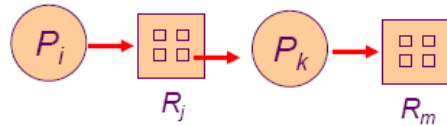
- 시스템의 **자원(resource)**은 다수의 유형으로 분할
  - 메모리 공간, CPU 주기, 파일, 입/출력 장치(프린터, DVD 드라이브 등) 등이 자원 유형들의 예
- 각 자원의 유형은 동등한 다수의 **인스턴스(instance)**들로 구성
  - 한 시스템이 두 개의 CPU를 가진다면, 자원 유형 CPU는 두 개의 인스턴스를 가짐
- 프로세스는 다음 순서로만 자원을 사용
  - **요청(request)**: 요청이 즉시 허용되지 않으면(예를 들어, 자원이 다른 프로세스에 의해 사용될 경우), 요청 프로세스는 자원을 얻을 때까지 대기
  - **사용(use)**: 프로세스는 자원에 대해 작업을 수행
  - **방출(release)**: 프로세스가 자원을 방출

## 교착 상태의 특징 (Deadlock Characterization)



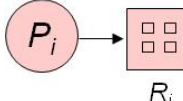
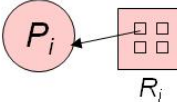
- 교착 상태는 한 시스템에 다음 네 가지 조건이 동시에 성립될 때 발생
  - **상호 배제 (mutual exclusion)**
    - 한 번에 오직 한 프로세스만이 한 자원을 사용
  - **점유하며 대기(hold and wait)**
    - 최소한 하나의 자원을 점유한 프로세스가 다른 프로세스에 의해 점유된 자원을 추가로 얻기 위해 대기해야 함
  - **비선점 (no preemption)**
    - 자원이 강제로 방출될 수 없고, 점유하고 있는 프로세스가 태스크를 종료한 후 그 프로세스에 의해 자발적으로만 방출
  - **순환 대기 (circular wait)**
    - 대기하고 있는 프로세스의 집합  $\{P_0, P_1, \dots, P_n\}$  에서  $P_0$ 는  $P_1$ 이 점유한 자원을 대기하고,  $P_1$ 은  $P_2$ 가 점유한 자원을 대기하고,  $P_2, \dots, P_{n-1}$ 은  $P_n$ 이 점유한 자원을 대기하며,  $P_n$ 은  $P_0$ 가 점유한 자원을 대기함

# 자원 할당 그래프 (1) (Resource-Allocation Graph)

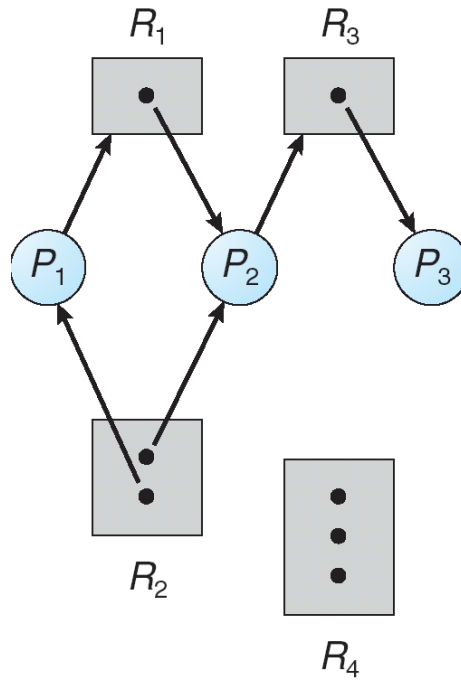
- 정점(vertex)  $V$ 의 집합과 간선(edge)  $E$ 의 집합으로 구성된 그래프
- $V$ 는 두 가지 유형으로 구별
  - $P = \{P_1, P_2, \dots, P_n\}$ , 시스템 내의 모든 활성 프로세스의 집합
  - $R = \{R_1, R_2, \dots, R_m\}$ , 시스템 내의 모든 자원 유형의 집합
- 방향 간선(directed edge)은  $P_i \rightarrow R_j$ 
  - 프로세스  $P_i$ 가 자원 유형  $R_j$ 의 인스턴스를 하나 요청
- 방향 간선은  $R_i \rightarrow P_k$ 
  - 자원 유형  $R_i$ 의 한 인스턴스가 프로세스  $P_k$ 에 할당



# 자원 할당 그래프 (2)

- 프로세스 
- 4개의 인스턴스들을 갖는 자원 
- $P_i$ 가  $R_j$ 의 인스턴스를 요청 
- $P_i$ 가  $R_j$ 의 인스턴스를 할당 받아 점유 

# 자원 할당 그래프 예

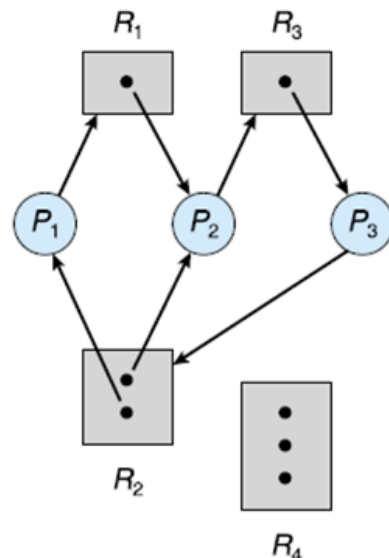


# 교착 상태를 갖는 자원 할당 그래프

□ 그래프가 사이클(cycle)을 포함하면 교착 상태

- 각 자원 유형이 여러 개의 인스턴스를 가지면, 사이클이 반드시 교착 상태가 발생했음을 의미하지는 않음

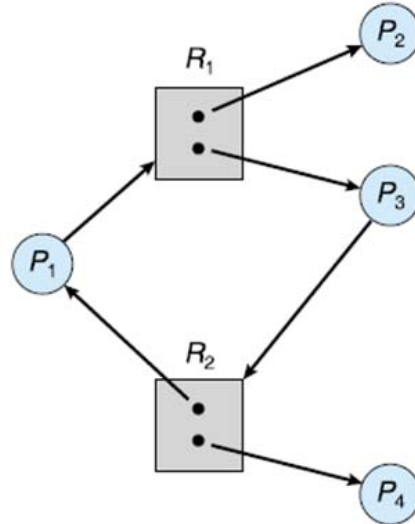
- $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$



## 사이클이 있으면서 교착 상태가 아닌 자원 할당 그래프

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

- 프로세스  $P_4$ 가 자원 유형  $R_2$ 의 인스턴스를 방출하면 그 자원이  $P_3$ 에 할당될 수 있고, 그 경우 사이클이 없어짐



## 자원 할당 그래프 요약

- 자원 할당 그래프에 사이클이 없다면, 시스템은 교착 상태가 아님
- 사이클이 있는 경우
  - 자원 유형 당 하나의 인스턴스만 있다면 교착상태
  - 자원 유형 당 여러 개의 인스턴스가 있다면 교착 상태의 가능성

# 교착 상태 처리 방법 (Method for Handling Deadlock)

- 원칙적으로 교착 상태 문제를 처리하는 세 가지 방법
  - 시스템이 결코 교착 상태가 되지 않도록 보장하기 위하여 **교착 상태를 예방하거나 회피**하는 프로토콜을 사용
  - 시스템이 교착 상태가 되도록 허용한 다음에 **회복**시키는 방법
  - 문제를 무시하고, 교착 상태가 시스템에서 결코 **발생하지 않는** **척 함**
    - UNIX와 Windows를 포함해 대부분의 운영체제가 사용하는 방법

# 교착 상태 예방 (Deadlock Prevention) - 상호배제

- 교착 상태를 발생시키는 네 가지 조건 중에서 **최소한 하나가 성립하지 않도록 보장**
- 상호 배제 (Mutual Exclusion)
  - **공유 가능한 자원**들은 배타적인 접근을 요구하지 않아서 교착상태 발생시키지 **않음**
    - 읽기 전용 파일 등
  - 상호 배제 조건은 **공유가 불가능한 자원에 대해서는 반드시 성립**해야 함
    - 프린터 등
    - 일반적으로 상호 배제 조건을 거부함으로써 교착 상태를 예방하는 것은 불가능

## 교착 상태 예방 - 점유하며 대기

### □ 점유하며 대기(Hold and Wait)

- 프로세스가 **자원을 요청할 때는, 다른 자원들을 점유하지 않을 것을 반드시 보장**해야 함
- 각 프로세스가 실행되기 전에 사용되는 모든 자원을 요청하고 모두 할당 받을 것을 요구
- 다른 프로세스가 자원을 전혀 갖고 있지 않을 때만 자원을 요청할 수 있도록 허용
- 단점
  - 많은 자원들이 할당된 후 오랜 동안 사용되지 않기 때문에 자원의 이용도가 낮음
  - 기아 (starvation) 상태 유발 가능성

## 교착 상태 예방 - 비선점

### □ 비선점 (No Preemption)

- 이미 **할당된 자원이 선점되지 않아야 한다는 조건이 성립되지 않도록 보장**하는 프로토콜을 사용
- 만일 어떤 자원을 점유하고 있는 프로세스가 즉시 할당할 수 없는 다른 자원을 요청하면(즉, 프로세스가 반드시 대기해야 하면), 현재 점유하고 있는 모든 자원들이 선점
  - 즉, 현재 점유하고 있는 자원들을 방출 (release)
- 선점된 자원들은 기다리고 있는 프로세스 자원들의 리스트에 추가
- 자원이 선점된 프로세스는 자신이 요청하고 있는 새로운 자원은 물론 이미 점유하였던 옛 자원들을 다시 획득할 수 있을 때에만 다시 시작



## 교착 상태 예방 - 순환 대기(Circular Wait)

### □ 순환 대기 (Circular Wait)

- 순환 대기 조건이 성립되지 않도록 모든 자원 유형들에게 전체적인 순서를 부여
- 각 프로세스가 열거된 순서대로 오름차순으로 자원을 요청하도록 요구

### □ 교착 상태 예방 알고리즘은 요청 방법을 제약하여 교착 상태를 예방

- 이 제약은 교착 상태를 위해 필요한 조건 중 최소한 하나가 발생하지 않도록 보장
- 이런 방식으로 교착 상태를 예방할 때 가능한 부수적인 문제는 장치의 이용률이 저하되고 시스템 처리율(throughput)이 감소

## 교착상태 예 (1)

```

/* 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);
}

```

## 교착상태 예 (2)

```

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);
}

```

□ 트랜잭션 1,2가 병행 실행

- 트랜잭션 1은 계좌 A에서 계좌 B로 5만원 이체
- 트랜잭션2는 계좌 B에서 계좌 A로 10만원 이체

## 교착 상태 회피 (Deadlock Avoidance)

□ 교착 상태를 회피하는 다른 대안은 자원이 어떻게 요청될지에 대한 **추가 정보**를 제공하도록 요구

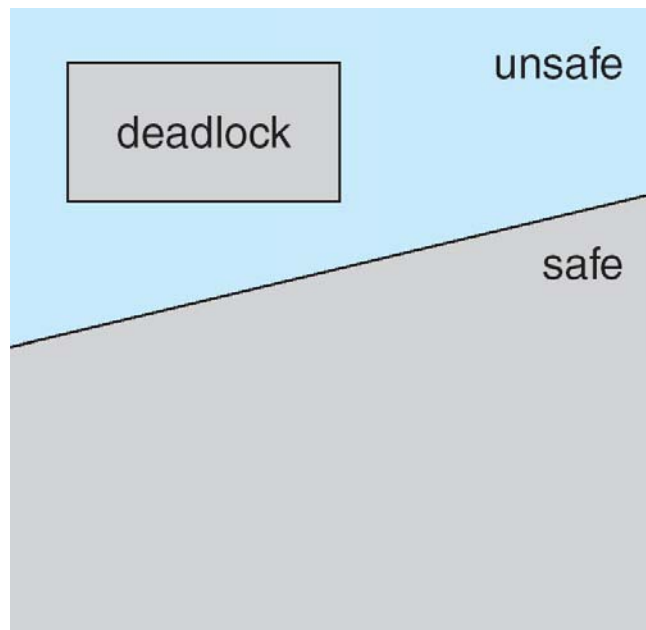
- 가장 단순하고 제일 유용한 모델은 각 프로세스가 자신이 필요로 하는 각 유형의 자원마다 **최대 수**를 선언하도록 요구
- 교착 상태 회피 알고리즘은 시스템에 **순환 대기 상황**이 발생하지 않도록 **자원 할당 상태**를 검사
- 자원 할당 상태는 **가용 자원의 수**, **할당된 자원의 수** 그리고 **프로세스들의 최대 요구 수**에 의해 정의

## 안전 상태 (Safe State)

- 시스템이 모든 프로세스들의 **안전 순서(safe sequence)**를 찾을 수 있다면 시스템은 **안전(safe)**
- $P_i$ 가 요청하는 자원을 시스템에 **현재 남아 있는 자원**과 앞에서 수행을 마칠 모든 프로세스  $P_j$ 들이 ( $j < i$ ) **반납하는 자원들로 만족된다면**  $\langle P_1, P_2, \dots, P_n \rangle$ 과 같은 프로세스 순서 (process sequence)가 **안전**
  - $P_i$ 가 요청할 자원들을 즉시 만족시킬 수 없으면 모든  $P_j$ 들이 마친 후 까지  $P_i$ 는 기다림
  - $P_j$ 가 끝나면  $P_i$ 는 반납한 자원들을 가지고 수행
  - $P_i$ 가 끝났을 때  $P_{i+1}$ 은 필요한 모든 자원들을 얻게 되고 이와 같은 상황이 계속 반복

## 안전, 불안전, 교착상태

- 시스템이 **안전 상태**  
=> **교착 상태 아님**
- 시스템이 **불안전 상태**  
=> **교착 상태 가능성**
- **교착 상태 회피**
  - 시스템이 불안전 상태로 진입하지 않도록 보장



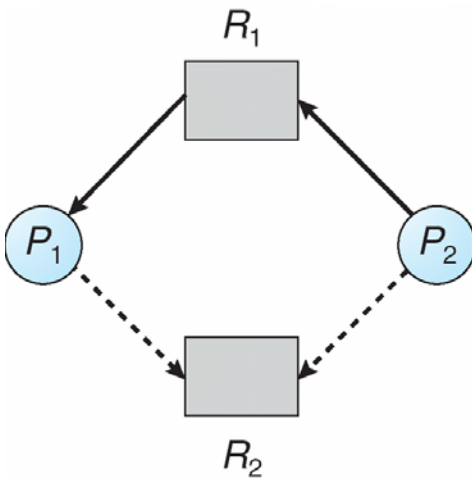
# 교착 상태 회피 알고리즘

- 2가지 유형의 교착 상태 알고리즘
- 단일 인스턴스의 자원 유형  
=> 자원 할당 그래프 알고리즘 적용
- 다수의 인스턴스를 갖는 자원 유형  
=> 은행원 알고리즘 적용

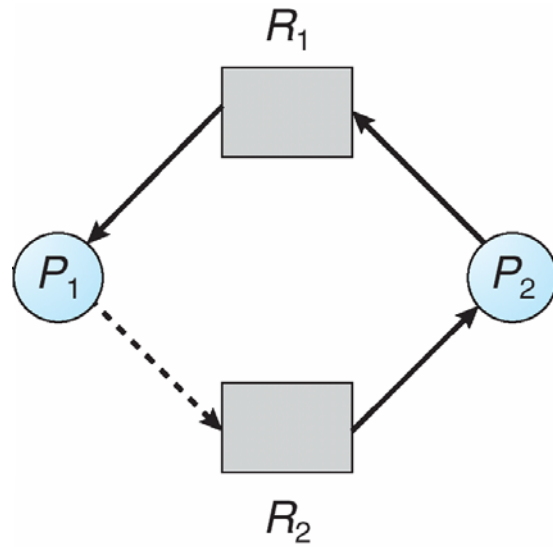
# 자원 할당 그래프 알고리즘 (1) (Resource-Allocation Graph Algorithm)

- 교착 상태 회피를 위해 자원 할당 그래프를 변형
- 요청 간선(request edge)과 할당 간선(assignment edge)에 추가하여 예약 간선(claim edge)을 도입
  - 예약 간선  $P_i \rightarrow R_j$ 는  $P_i$ 가 미래에 자원  $R_j$ 를 요청할 것이라는 의미
  - 점선(dashed line)으로 표시
  - 프로세스  $P_i$ 가 자원  $R_j$ 를 요청하면, 예약 간선  $P_i \rightarrow R_j$ 는 요청 간선으로 변환
  - $P_i$ 가 자원  $R_j$ 를 방출할 때, 할당 간선  $R_j \rightarrow P_i$ 는 예약 간선  $P_i \rightarrow R_j$ 로 다시 변환
- 시스템에서 자원이 반드시 미리 예약되어야 함
  - 프로세스  $P_i$ 가 실행되기 전에 프로세스의 모든 예약 간선들이 자원 할당 그래프에 표시되어야 함

## 자원 할당 그래프 예



- $P_2$ 가  $R_2$ 를 요청하면?
  - $R_2$ 를 할당할 수 없음
  - 할당하면 사이클이 발생하여 교착 상태



불안정한 상태의 자원 할당 그래프

## 자원 할당 그래프 알고리즘 (2)

- $P_i$ 가 자원  $R_j$ 를 요청하는 경우 가정
- 요청 간선을 할당 간선으로 변환해도 자원 할당 그래프에서 사이클이 발생하지 않으면 요청을 수락

## 은행원 알고리즘 (Banker's Algorithm)

- 은행원 알고리즘은 자원 종류(유형) 당 여러 개의 인스턴스를 갖는 경우에도 적용
- 각 프로세스는 필요한 자원의 최대 개수를 자원 종류마다 미리 신고
- 프로세스가 자원들을 요청 시 요청을 수락하면 시스템이 계속 안전 상태에 머무르게 되는지 여부를 판단
  - 계속 안전하게 된다면 그 요청을 수락
  - 안전하지 않으면 프로세스의 요청은 수락되지 않고 다른 프로세스가 끝나기까지 기다림

## 은행원 알고리즘의 자료 구조 (1)

- $n$ 은 프로세스의 수이고,  $m$ 이 자원 종류의 수
- Available
  - 각 종류의 자원이 현재 몇 개가 사용 가능한지를 나타내는 벡터로 크기가  $m$
  - $Available[j] = k$  라면 현재  $R_j$ 를  $k$  개 사용할 수 있다는 의미
- Max
  - 각 프로세스가 최대 필요로 하는 자원의 개수를 나타내는 행렬로 크기가  $n \times m$
  - $Max[i,j] = k$  라면  $P_i$ 가  $R_j$ 를 최대  $k$  개까지 요청할 수 있음을 의미
- Allocation
  - 각 프로세스에 현재 할당된 자원의 개수를 나타내는 행렬로 크기가  $n \times m$
  - $Allocation[i,j] = k$  라면 현재  $P_i$ 가  $R_j$ 를  $k$  개 사용중임을 의미

## 은행원 알고리즘의 자료 구조 (2)

### □ Need

- 각 프로세스가 향후 요청할 수 있는 자원의 개수를 나타내는 행렬로 크기가  $n \times m$
- $Need[i,j] = k$ 라면  $P_i$ 가 향후  $R_j$ 를  $k$  개까지 더 요청할 수 있음을 의미
- $Need[i,j] = Max[i,j] - Allocation[i,j]$

## 안전성 알고리즘 (Safety Algorithm)

1.  $Work = Available$ 로 초기화.  $i = 0, 1, \dots, n-1$ 에 대해서  $Finish[i] = false$ 로 초기화  
( $Work$ 와  $Finish$ 는 크기가  $m$ 과  $n$ 인 벡터)
2. 아래 두 조건을 만족시키는  $i$  값을 탐색
  - $Finish[i] == false$
  - $Need_i \leq Work$
 위 조건을 만족하는  $i$  값을 찾을 수 없다면 step 4로 이동
3.  $Work = Work + Allocation_i$   
 $Finish[i] = true$   
Go to step 2.
4. 모든  $i$  값에 대해  $Finish[i] == true$ 이면 이 시스템은 안전 상태에 있음

## 자원 요청 알고리즘 (Resource-Request Algorithm)

- Request<sub>i</sub>는 프로세스 P<sub>i</sub>의 요청 벡터
  - Request<sub>i</sub>[j] == k 라면 P<sub>i</sub>가 R<sub>j</sub>를 k 개 요청하고 있음을 의미
- 1. 만일 Request<sub>i</sub> ≤ Need<sub>i</sub>이면 step 2로 이동하고, 아니면 시스템에 있는 개수보다 더 많이 요청했으므로 오류로 처리
- 2. 만일 Request<sub>i</sub> ≤ Available이면 step 3으로 간다. 아니면 요청한 자원이 당장은 없으므로 P<sub>i</sub>는 기다림
- 3. 마치 시스템이 P<sub>i</sub>에게 자원을 할당해준 것처럼 시스템 상태 정보를 아래처럼 변경 조사
  - Available = Available - Request<sub>i</sub>;
  - Allocation<sub>i</sub> = Allocation<sub>i</sub> + Request<sub>i</sub>;
  - Need<sub>i</sub> = Need<sub>i</sub> - Request<sub>i</sub>;
 만일 이렇게 바뀐 상태가 안전하다면 P<sub>i</sub>에게 자원을 할당  
 새로운 상태가 불안정하다면, 위의 자원 할당 상태는 원상태로 복원  
 되고 P<sub>i</sub>는 Request<sub>i</sub>가 만족되기까지 기다림

## 은행원 알고리즘 적용 예 (1)

- P<sub>0</sub>부터 P<sub>4</sub>의 5개 프로세스
- A, B, C 세 가지 종류의 자원
  - A 자원이 10개, B 자원이 5개, C 자원이 7개
- 임의의 시간 T<sub>0</sub>에 시스템은 아래와 같은 상태

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>	<u>Need</u>
	A B C	A B C	A B C	
P <sub>0</sub>	0 1 0	7 5 3	3 3 2	?
P <sub>1</sub>	2 0 0	3 2 2		
P <sub>2</sub>	3 0 2	9 0 2		
P <sub>3</sub>	2 1 1	2 2 2		
P <sub>4</sub>	0 0 2	4 3 3		



	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P <sub>0</sub>	0	1	0	7	5	3	3	3	2			
P <sub>1</sub>	2	0	0	3	2	2						
P <sub>2</sub>	3	0	2	9	0	2						
P <sub>3</sub>	2	1	1	2	2	2						
P <sub>4</sub>	0	0	2	4	3	3						

## 은행원 알고리즘 적용 예 (2)

- Need 행렬의 값은 (Max - Allocation)로 계산

	Need			Available		
	A	B	C	A	B	C
P <sub>0</sub>	7	4	3	3	3	2
P <sub>1</sub>	1	2	2			
P <sub>2</sub>	6	0	0			
P <sub>3</sub>	0	1	1			
P <sub>4</sub>	4	3	1			

What if P1 request (1,0,2)

- <P<sub>1</sub>, P<sub>3</sub>, P<sub>4</sub>, P<sub>2</sub>, P<sub>0</sub>> 순서가 안전성 기준을 만족하기 때문에 안전

## 은행원 알고리즘 적용 예 - P<sub>1</sub> 이 (1,0,2) 요청

- P<sub>1</sub>이 A 자원 한 개와 C 자원 두 개를 추가로 요청하는 경우
  - Request<sub>1</sub> = (1, 0, 2), Request<sub>1</sub> ≤ Available (1, 0, 2) ≤ (3, 3, 2)

	Allocation			Max			Available				Allocation			Need			Available			
	A	B	C	A	B	C	A	B	C		A	B	C	A	B	C	A	B	C	
P <sub>0</sub>	0	1	0	7	5	3	3	3	2		P <sub>0</sub>	0	1	0	7	4	3	2	3	0
P <sub>1</sub>	2	0	0	3	2	2					P <sub>1</sub>	3	0	2	0	2	0			
P <sub>2</sub>	3	0	2	9	0	2					P <sub>2</sub>	3	0	2	6	0	0			
P <sub>3</sub>	2	1	1	2	2	2					P <sub>3</sub>	2	1	1	0	1	1			
P <sub>4</sub>	0	0	2	4	3	3					P <sub>4</sub>	0	0	2	4	3	1			

- <P<sub>1</sub>, P<sub>3</sub>, P<sub>4</sub>, P<sub>0</sub>, P<sub>2</sub>> 순서가 안전성 기준을 만족하기 때문에 안전
- P<sub>4</sub>의 (3,3,0) 요청이 수락 가능?
- P<sub>0</sub>의 (0,2,0) 요청이 수락 가능?

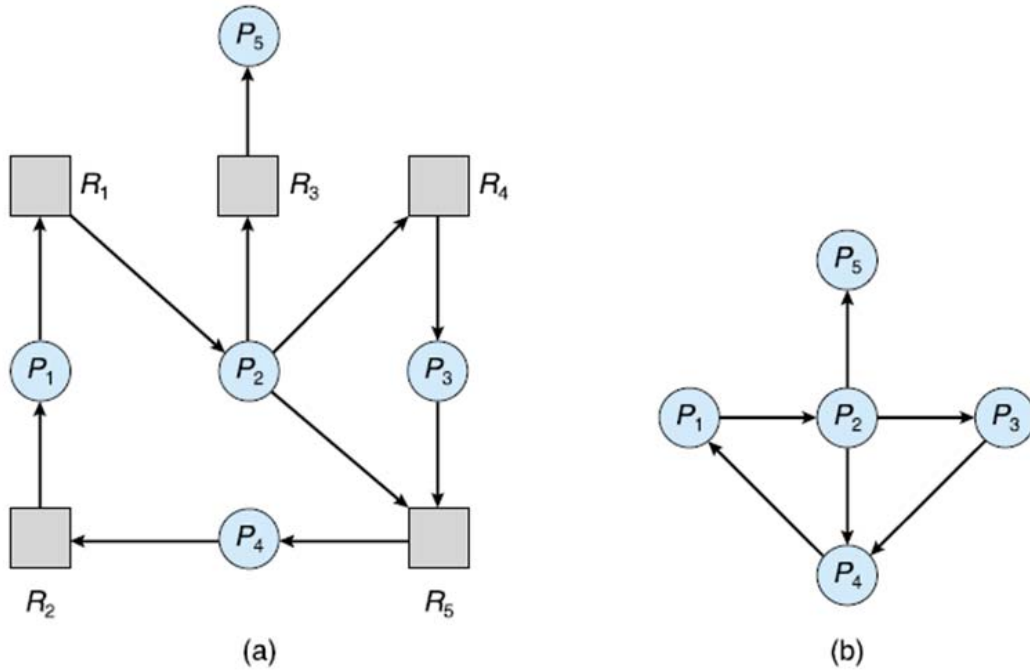
## 교착 상태 탐지(Deadlock Detection)

- 교착 상태 발생이 가능한 시스템에서는 다음들을 지원해야 함
  - 교착 상태가 발생했는지 결정하기 위해 시스템의 상태를 검사하는 알고리즘 (detection algorithm)
  - 교착 상태에서부터 회복하는 방식 (recovery scheme)
- 2가지 유형 논의
  - 각 자원 유형마다 하나의 인스턴스를 갖는 시스템
  - 자원 유형마다 다수의 인스턴스를 갖는 시스템

## 각 자원 유형이 한 개씩 있는 경우 (Single Instance of Each Resource Type)

- 대기 그래프(wait-for graph) 사용
  - 자원 할당 그래프로부터 자원 유형의 노드를 제거
  - $P_i \rightarrow P_j$ 
    - 프로세스  $P_j$ 가 가지고 있는 자원을 프로세스  $P_i$ 가 요청하여 기다림
- 주기적으로 그래프에서 사이클을 조사하여 교착상태 탐지
- 그래프에서 사이클을 탐지하는 알고리즘은  $O(n^2)$ 의 연산을 요구
  - $n$ 은 그래프에 있는 정점들(vertices)의 수이다.

## 자원 할당 그래프와 대기 그래프



## 각 유형의 자원을 여러 개 가진 경우 (Several Instances of a Resource Type)

- 은행원 알고리즘과 같이 시시각각 그 내용이 달라지는 자료 구조를 사용,  $O(m \times n^2)$
- 가용(Available)
  - 각 종류의 자원이 현재 몇 개가 가용한지를 나타내는 벡터로 크기가  $m$
- 할당(Allocation)
  - 각 프로세스에게 현재 할당되어 있는 자원의 개수를 나타내는 행렬로 크기가  $n \times m$
- 요청(Request)
  - 각 프로세스가 현재 요청중인 자원의 개수를 나타내는 행렬로 크기가  $n \times m$
  - $Request[i, j] == k$ 라면 현재  $P_i$ 가  $R_j$ 를  $k$  개 요청중임을 의미

## 탐지 알고리즘 (Detection Algorithm)

1.  $Work = Available$ 로 초기화.  $i = 0, 1, \dots, n-1$ 에 대해서  $Allocation_i \neq 0$ 이면  $Finish[i] = false$ , 아니면  $Finish[i] = true$  ( $Work$ 와  $Finish$ 는 크기가  $m$ 과  $n$ 인 벡터)
2. 아래 두 조건을 만족시키는  $i$  값을 탐색
  - $Finish[i] == false$
  - $Request_i \leq Work$
 위 조건을 만족하는  $i$  값을 찾을 수 없다면 step 4로 이동
3.  $Work = Work + Allocation_i$   
 $Finish[i] = true$   
 step 2로 이동
4. 어떠한  $i$  값에 대해 ( $0 \leq i \leq n$ )  $Finish[i] == false$ 이면  $P_i$ 가 교착 상태 (즉, 시스템이 교착 상태)

## 탐지 알고리즘 예 (1)

- $P_0$ 부터  $P_4$ 의 5개 프로세스
- 자원 A (7개 인스턴스), B (2), C (6)
- 임의의 시간  $T_0$ 에 시스템은 아래와 같은 상태

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	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	

- $\langle P_0, P_2, P_3, P_1, P_4 \rangle$  순서와 같이 작업들을 다 끝낼 수 있고, 모든  $i$ 에 대해서  $Finish[i] == true$

## 탐지 알고리즘 예 (2)

- P<sub>2</sub> 가 C 자원을 한 개 더 요청

	<i>Allocation</i>		<i>Available</i>		<i>Request</i>		
	A	B	C		A	B	C
P <sub>0</sub>	0	1	0	0	0	0	0
P <sub>1</sub>	2	0	0	0	2	0	2
P <sub>2</sub>	3	0	3	0	0	0	1
P <sub>3</sub>	2	1	1	0	1	0	0
P <sub>4</sub>	0	0	2	0	0	0	2

- P<sub>0</sub>의 자원을 회수해도 다른 프로세스들이 요구 자원을 충족시켜줄 방법이 없기 때문에 P<sub>1</sub>, P<sub>2</sub>, P<sub>3</sub>과 P<sub>4</sub>가 교착 상태에 연루

## 탐지 알고리즘 사용 (Detection-Algorithm Usage)

- 탐지 알고리즘의 수행 빈도는 다음을 고려
  - 교착 상태가 얼마나 자주 일어나는가?
  - 교착 상태가 일어나면 통상 몇 개의 프로세스가 거기에 연루되는가?
- 프로세스의 요청이 즉시 만족되지 않을 때마다 탐지 알고리즘을 수행하면 교착 상태에 연루된 프로세스들 뿐 아니라 교착 상태를 야기한 장본인 프로세스도 탐지
  - 자원을 요청할 때마다 탐지 알고리즘을 돌리면 너무 오버헤드가 큼
- 탐지 알고리즘을 가끔씩만 돌리면 한꺼번에 여러 개의 사이클이 탐지가 되어 어느 프로세스가 최종적으로 교착 상태를 야기한 장본인인지 알아내기 어려움

## 교착 상태에서부터 회복 - 프로세스 종료

- 프로세스를 중지시킴으로써 교착 상태를 제거
  - 교착 상태 프로세스를 모두 중지
  - 교착 상태가 제거될 때까지 한 프로세스씩 중지
- 부분 종료 방식의 경우 다음을 고려하여 중지될 프로세스의 선택
  - 프로세스의 우선순위
  - 지금까지 프로세스가 수행된 시간과 지정된 일을 종료하는 데 더 필요한 시간
  - 프로세스가 사용한 자원 유형과 수(예를 들어, 자원들을 선점하기가 단순한지 여부)
  - 프로세스가 종료하기 위해 더 필요한 자원의 수
  - 얼마나 많은 수의 프로세스가 종료되어야 하는지,
  - 프로세스가 대화형(interactive)인지 일괄 처리(batch)인지 여부

## 교착 상태에서부터 회복 - 자원 선점

- 자원 선점을 이용해 교착 상태를 제거하려면 다음을 고려
  - 희생자 선택 (selecting a victim)
    - 어느 자원과 어느 프로세스들이 선점될 것인가를 결정
    - 비용을 최소화하기 위해 선점의 순서를 결정
  - 후퇴 (rollback)
    - 중지될 프로세스를 안전한 상태로 후퇴시키고, 그 상태에서부터 다시 시작
    - 안전 상태가 어떤 것인지를 결정하기 어렵기 때문에, 가장 단순한 해결안은 프로세스를 중지(abort)시키고 재시작
  - 기아 상태(Starvation)
    - 동일한 프로세스가 항상 희생자로 선택되어 자신의 태스크를 결코 완료하지 못하는 기아 상태 발생하는 것을 방지
    - 일반적인 해결 방법은 비용 요소에 후퇴의 횟수를 포함

- 연습문제 7.12
- 연습문제 7.13

- 다음 은행원 알고리즘을 프로그램으로 구현하여라.
  - 안전성 알고리즘, 자원 요청 알고리즘
  - 연습문제 7.12, 7.13 적용 테스트
  - 구현 언어는 자유 선택 (C, C++, Java, C#, Python, ....)