7장. 교착상태(deadlock)

목표

- 병행 프로세스들의 집합에 속한 프로세스들이 작업을 완료할 수 없도록 하는 **교착 상태**에 대한 기술 방법 소개
- 컴퓨터 시스템에서 교착상태를 **예방하거나 회피하는** 여러 방법을 제시함

7.1 시스템 모델

- 시스템 모델
  - 자원(resource) - 유한한 개수의 자원
  - 프로세스(process) - 자원 사용을 경쟁하는 프로세스들
- 자원
  - 시스템의 자원은 여러 유형(type)으로 구분됨
    - 물리적 자원: CPU 사이클, 메모리 공간, I/O 장치
    - 논리적 자원: 파일, 세마포, 뮤텍스 락
  - 각 유형의 자원(R_i)은 여러 개(W_i)의 인스턴스(instance)를 가질 수 있음
- 프로세스의 자원 이용 과정
  1. 요청(request) – 요청 자원을 즉시 사용할 수 없으면,
자원을 획득할 때까지 대기해야 함
  2. 사용(use)
  3. 방출/해제(release)
  - 이와 관련된 시스템 호출 : open() – close(), request() – release(),
allocate() – free(), wait() – signal()

교착상태 문제

- 교착상태(deadlock)
  - 프로세스 집합에 있는 모든 프로세스가 한 자원을 점유하고,
    같은 집합의 다른 프로세스가 점유한 자원의 획득을 기다리는 상태
    → 모든 프로세스가 진행되지 못함
- 예1
  - 자원: DVD, Printer
  - P1은 DVD를 점유하고 프린터 요청
    P2는 프린터를 점유하고 DVD 요청
- 예2
  - 자원: 1로 초기화된 두 개의 Semaphore A, B

P_0	P_1
P(A);	*P(B);*
P(B);	*P(A);*

7.2 교착상태의 특징

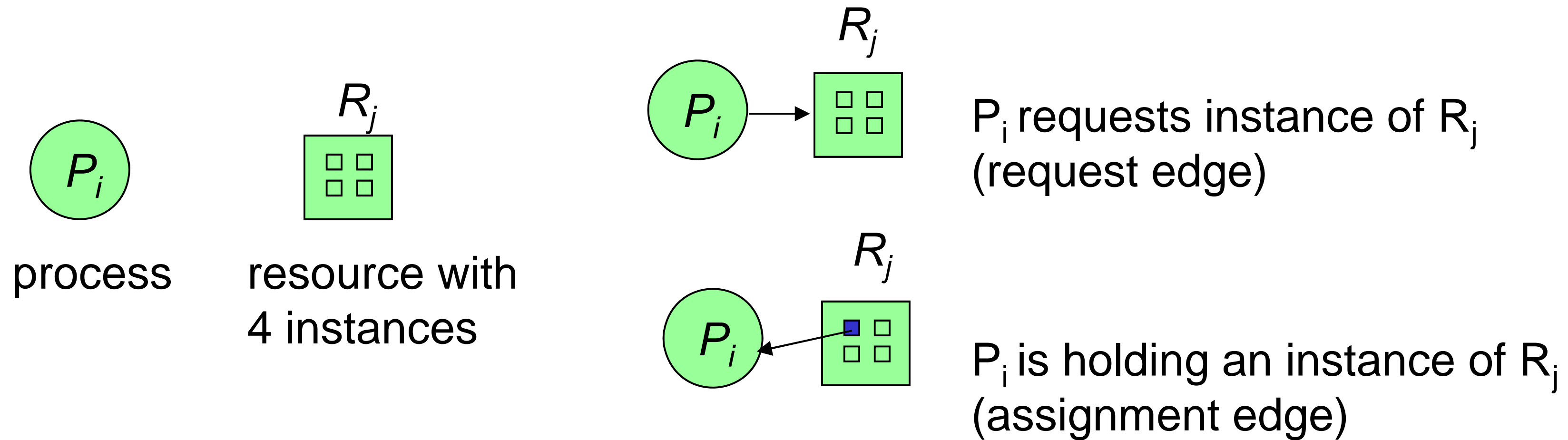
- 교착상태가 발생하는 필요조건
  - 다음 4가지 조건이 동시에 성립할 때에 발생함

1. **상호배제**(Mutual exclusion)
   - 한 번에 한 프로세스만 사용할 수 있는 자원(비공유 모드로 점유되는 자원)이 적어도 하나 존재
2. **점유하며 대기**(Hold and wait)
   - 프로세스가 최소 하나의 자원을 점유한 상태에서 다른 프로세스가 점유한 자원을 대기
3. **비선점**(No preemption)
   - 자원이 강제로 방출될 수 없고, 자발적으로만 방출될 수 있음
4. **순환대기**(Circular wait)
   - 프로세스 집합 $\{P_0, P_1, \ldots, P_n\}$에서,
     P_0 는 P_1 점유자원 대기
     P_1 는 P_2 점유자원 대기,
     …
     P_{n-1} 는 P_n 점유자원 대기
     P_n 는 P_0 점유자원 대기

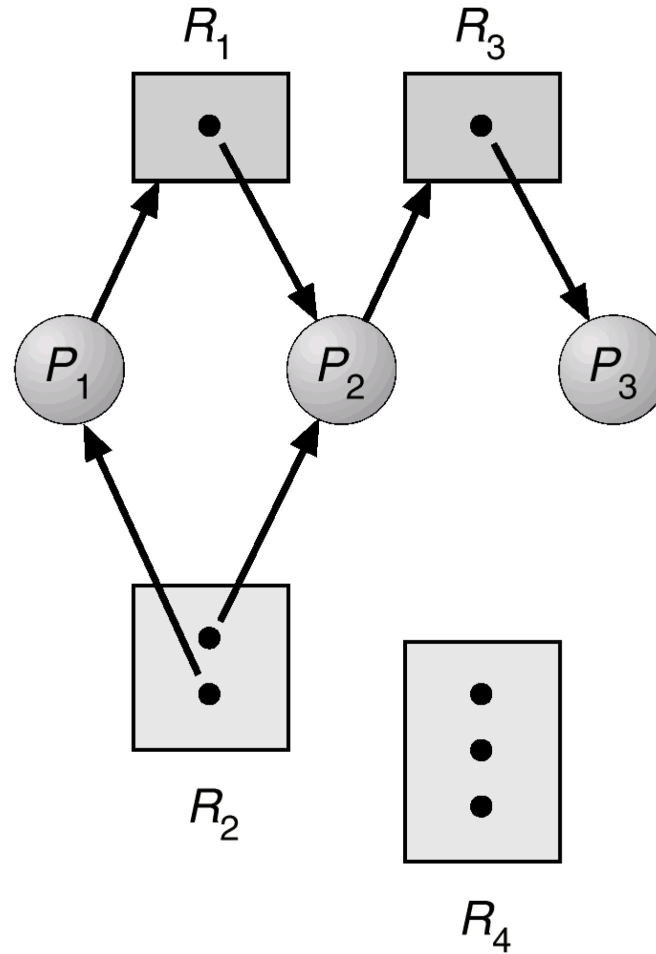
(예) 식사하는 철학자 문제

자원 할당 그래프

- 자원할당 그래프
  - V: vertex 집합, E: edge 집합
  - vertex 집합 V = 두 종류의 집합 P와 R로 구분됨
    - $P = \{P_1, P_2, \ldots, P_n\}$: 프로세스 집합
    - $R = \{R_1, R_2, \ldots, R_m\}$: 자원의 집합
  - edge – 방향성이 있음
    - request edge : $P_i \rightarrow R_j$ (프로세스가 자원 요청)
    - assignment edge : $R_j \rightarrow P_i$ (자원이 프로세스에 할당됨)



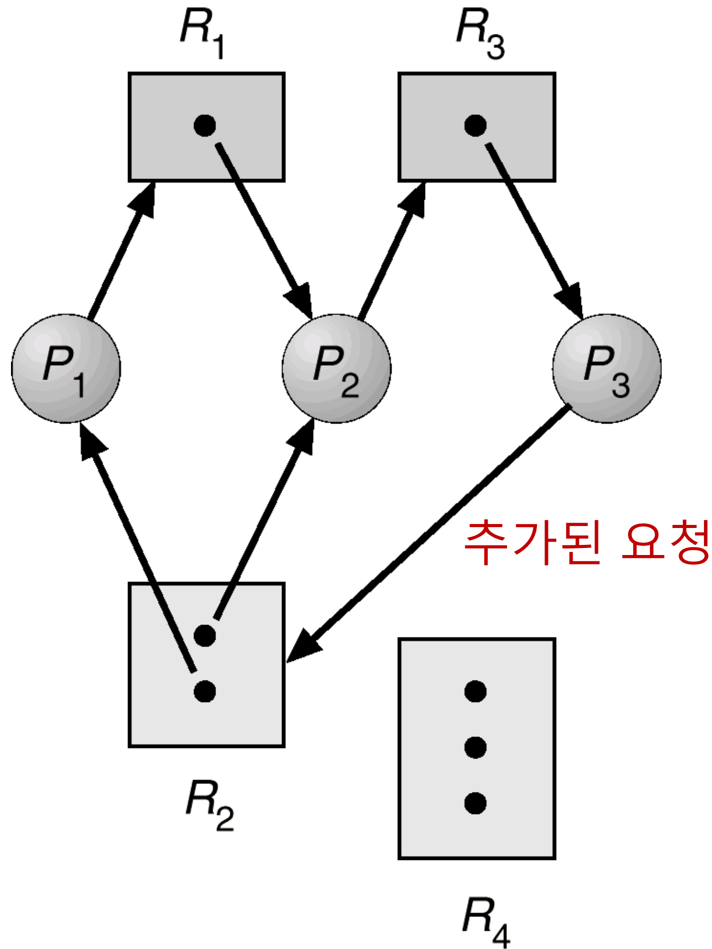
자원 할당 그래프의 예



자원 할당 그래프와 deadlock

- 자원 할당 그래프가 cycle이 없음 ➔ no deadlock
- 자원 할당 그래프가 cycle 포함
  - 자원 유형 당 1개의 instance를 보유 ➔ deadlock
  - 자원 유형 당 여러 개의 instance를 보유 ➔ deadlock 가능성
    (cycle 수가 instance 수보다 작으면 no deadlock 가능)

사이클을 갖는 자원 할당 그래프 - deadlock



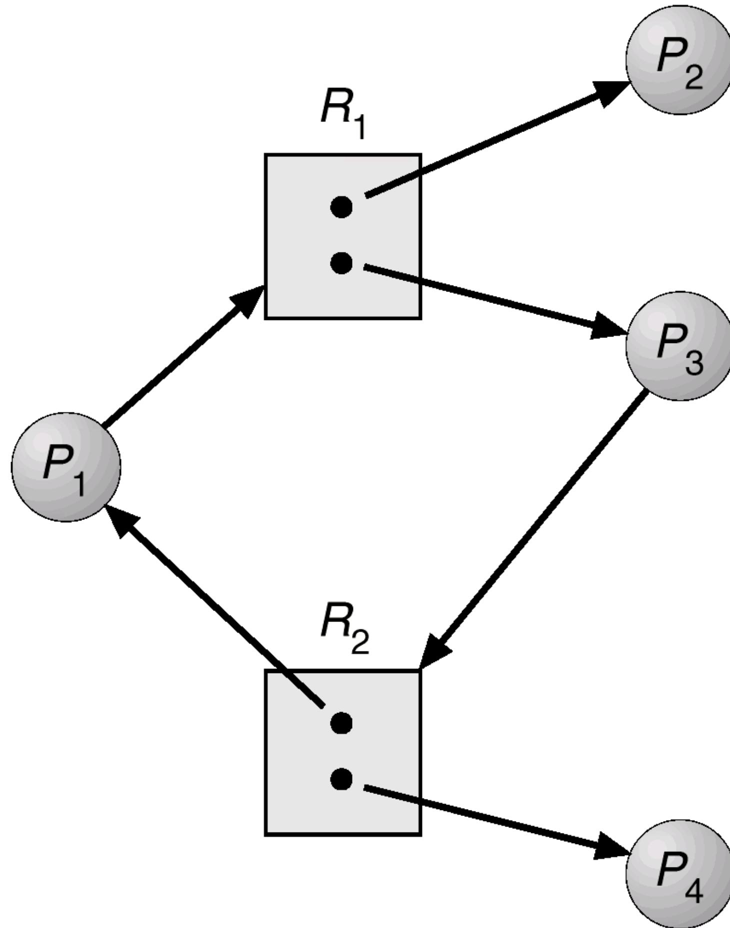
- 교착상태에 있는 자원할당 그래프
  - cycle이 존재 (순환 대기)
- 두 개의 cycle 존재

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

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

  - R_2의 인스턴스가 2개이어도 2개의 cycle을 존재하므로 교착상태 발생

사이클을 갖는 자원 할당 그래프 – no deadlock



- One cycle

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

  $R_2 \rightarrow P_4$

  $R_1 \rightarrow P_2$

  - R_2가 2개의 인스턴스를 가지므로 교착상태가 발생하지 않음

7.3 교착상태 처리 방법

- 교착상태 예방(prevention) 또는 회피(avoidance)
  - 시스템이 deadlock 상태가 되지 않도록 보장하는 방법
  - **deadlock prevention** :
    - 4가지 필요조건 중 하나 이상을 만족하지 않게 함
  - **deadlock avoidance** :
    - 부가적인 정보를 사용하여 자원 요청을 위해 프로세스가 대기할 지 여부를 결정
- 교착상태 탐지(detection) 후 복구(recovery)
  - deadlock 상태가 되는 것을 허용
  - deadlock detection 알고리즘과 deadlock recovery 알고리즘으로 구성
- 교착상태 무시
  - deadlock이 자주 발생하지 않기 때문에 예방/회피나 탐지를 하지 않음
  - 수작업으로 복구
  - 대부분의 운영체제에서 사용 – 응용프로그램의 deadlock 방지는 응용 프로그래머 책임

7.4 교착상태 예방

- 4가지 필요조건들 중 적어도 한 개를 만족하지 않게 함
- 상호배제 조건 거부
  - 동시 공유 가능 자원은 상호배제 하지 않음 (예) 읽기 전용 파일
  - 동시 공유 불가능 자원은 상호배제를 해야 함
- 점유 대기 조건 거부
  - 자원을 요청할 때에 다른 자원을 점유하지 않은 상태에서 자원을 요청
  - 두 가지 방법
    1. 실행하기 전에 모든 자원을 요청하여 할당 받도록 함
    2. 점유 자원을 반납한 후에 이 자원과 추가 자원을 요청함
  - 단점
    - 자원 이용률이 낮음 - 자원들이 할당된 후 일부 자원이 오랫동안 사용되지 않을 수 있음
    - 기아 가능성

교착상태 예방(계속)

- 비선점 조건 거부
  - 어떤 자원을 점유한 프로세스가 다른 자원 요청하여 즉시 할당 받을 수 없으면,
  - **방법 1**: 현재 점유한 모든 자원을 반납하고 대기 상태가 된다. 프로세스는 반납한 자원과 추가 요청 자원들의 대기 리스트에 추가
    - 반납한 자원과 요청한 자원을 할당 받을 수 있을 때에 재개됨
  - **방법 2**: 요청한 자원을 이용할 수 없거나 다른 대기 프로세스가 점유하지 않았다면 대기 상태가 됨
    - 점유한 자원은 다른 프로세스가 요청하는 경우에만 선점되어 반납
    - 선점된 자원과 요청한 자원을 할당 받을 수 있을 때에 재개됨
  - 단점:
    - CPU 레지스터나 메모리 공간처럼 저장과 복원이 용이한 자원에만 선점 적용가능

교착상태 예방(계속)

- 순환 대기 조건 거부
  - 모든 자원들에 전체적인 순서(total ordering) 부여
    - One-to-one mapping function: F(r) = n   (r: resource, n: integer)
  - 여러 개의 자원이 필요한 경우에 오름차순으로 자원들을 요청함
    - R_i을 점유한 프로세스는 $F(R_j) > F(R_i)$ 일 때만 R_j 를 요청할 수 있음
    - $F(R_j) \le F(R_i)$ 인 자원 R_i 은 반납해야 함
  - 위의 두 가지 프로토콜을 사용하면 순환 대기가 발생하지 않음
- (예) 식사하는 철학자 문제 – 세마포를 오름차순으로 요청

```
if ( i < (i+1)%5 ) {
        P(chopStick[i]);
        P(chopStick[(i+1) % 5]);
} else {
        P(chopStick[(i+1) % 5]);
        P(chopStick[i]);
}
```

7.5 교착상태 회피

- 교착상태 예방의 단점
  - 장치 이용률(utilization) 저하
  - 시스템 처리율(throughput) 감소
- 교착상태 회피(avoidance) 방법
  - 자원이 어떻게 요청되는 지에 대한 추가 정보를 필요로 함
    - 필요로 하는 각 유형별 자원에 대한 최대 수 등.
  - 자원 할당 상태를 조사하여 순환 대기 조건이 발생하지 않도록 자원 요청을 제어함
- 자원-할당 상태(resource-allocation state) – 다음 값으로 정의
  - 가용(available) 자원 수, 할당(allocated) 자원 수
  - 프로세스의 최대 요구(demand) 수

안전 상태(safe state)

- 안전 상태
  - 시스템이 **안전 순서(safe sequence)**가 존재하하면, 안전 상태에 있다.
- 안전 순서(safe sequence)
  - 프로세스 시퀀스 <P_1, P_2, …, P_n>에 대해서
  - P_i 가 요청하는 자원이

    (1) 현재의 가용 자원과

    (2) $j < i$인 모든 P_j 가 점유한(즉, 사용 후 반납하는) 자원

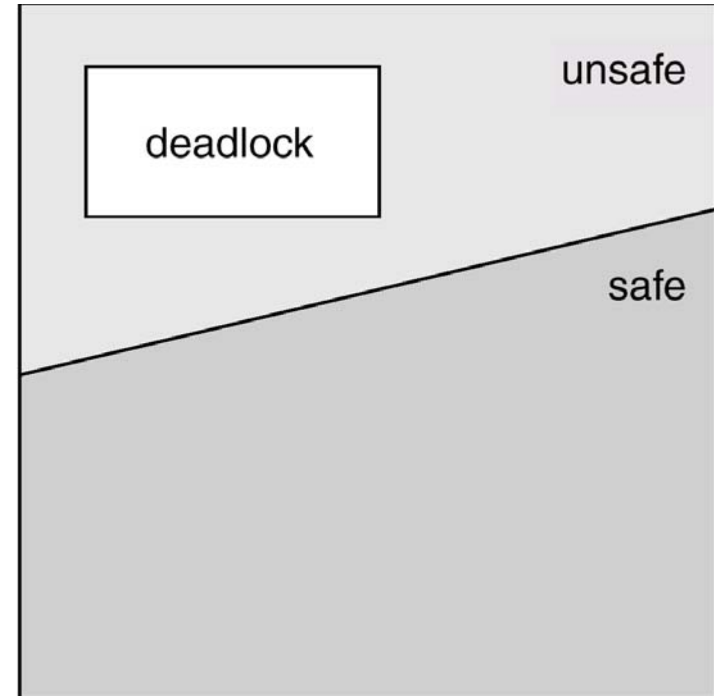
    에 의해서 만족된다면
  - 프로세스 시퀀스는 안전 순서라고 함

$$P_i \text{ 요구량} \leq \text{ 가용 자원 수} + \sum_{j=1}^{i-1} (P_j \text{ 반납자원 수})$$

- 안전한 프로세스 시퀀스에 대해서
  - P_i는 자원 요구량이 즉시 충족되지 못할 때에는 가용자원과 시퀀스의 앞에 있는 모든 프로세스가 끝나서 반납한 자원들과 가지고 수행할 수 있음.

Safe, Unsafe, Deadlock 상태

- Safe 상태 → No deadlock
- Unsafe 상태 → Deadlock 가능성

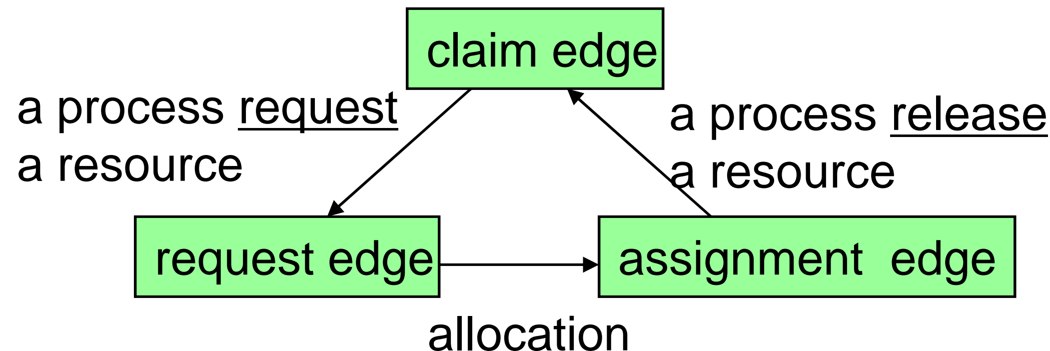


- 교착상태 회피 알고리즘
  - 시스템이 **Unsafe 상태**로 진입하지 않도록 보장해야 함.

- 교착상태 회피 알고리즘 유형
  - 단일 인스턴스 자원 유형
    → 자원 할당 그래프 사용
  - 다수의 인스턴스를 갖는 자원 유형
    → **은행원(banker's) 알고리즘** 사용 (Dijkstra 제안)

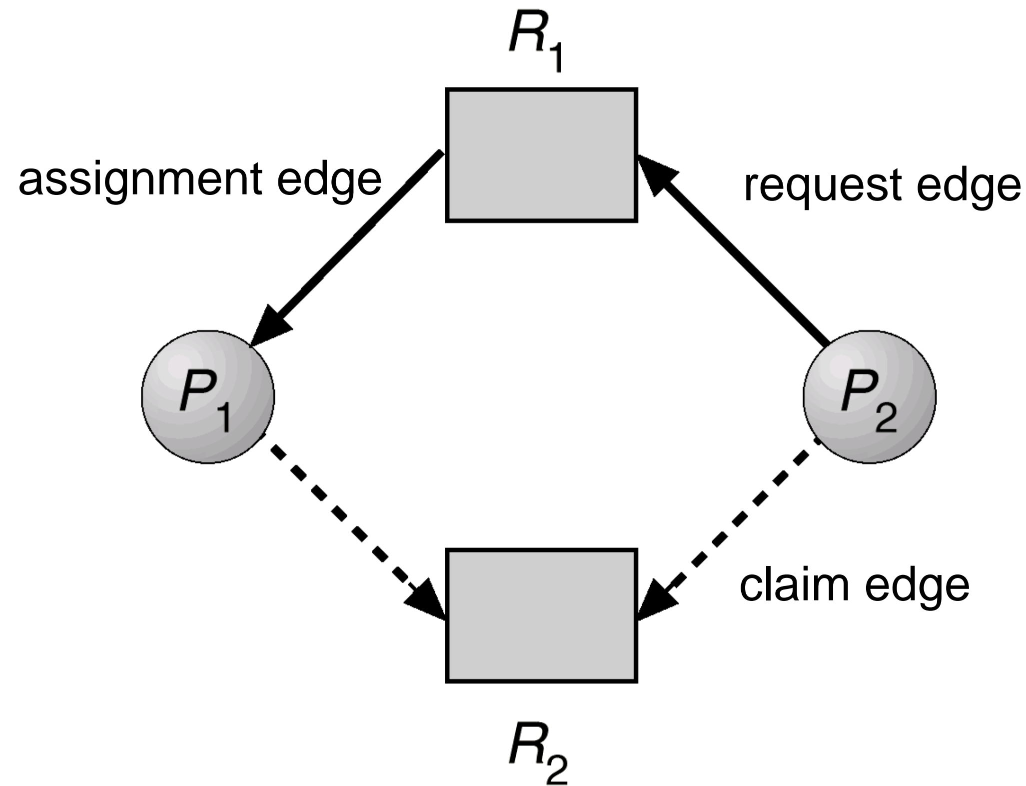
자원 할당 그래프 알고리즘

- 예약 에지(claim edge)
  - 프로세스 Pi는 미래에 Rj를 요청할 수 있음 (Pi → Rj)
    - 즉 claim edge는 request edge로 변환될 수 있음
  - 점선으로 표시

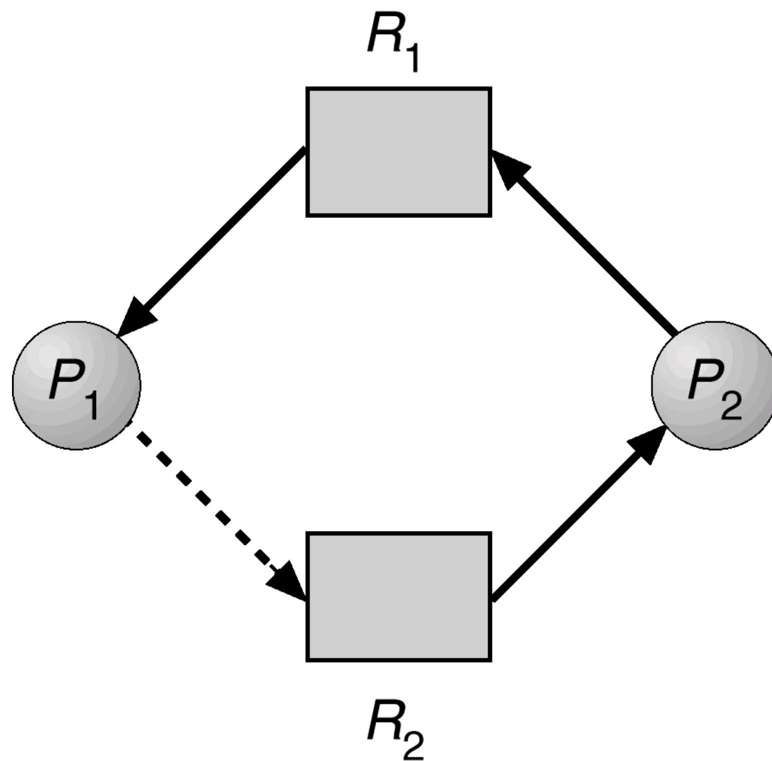


- 요청 승인(grant)
  - 자원 요청은 **요청 에지**(request edge) Pi → Rj가 **할당 에지**(assignment edge) Rj → Pi로 변환하는 것이 자원 할당 그래프에 cycle을 형성하지 않을 때에만 승인한다.

자원 할당 그래프 - 교착상태 회피를 위한



자원 할당 그래프 – Unsafe 상태로 되는 경우



- P2가 R2를 요청
  (claim edge가 request edge가 됨)
- request edge P2→R2가 assignment edge R2→P2로 변환되면 unsafe 상태가 됨
- P1이 R2를 요청할 때에 deadlock
- 따라서 P2의 R2요청은 승인하지 않음

은행원(Banker's) 알고리즘

- 여러 개의 인스턴스를 갖는 자원 유형을 사용할 때에 적용
- 필요 조건
  - 프로세스가 시작할 때 자원 유형마다 필요한 최대 개수를 선언해야 함
    - 최대 개수는 시스템이 보유한 개수 이하이어야 함
  - 자원을 요청할 때, 자원 할당이 <u>safe 상태를 유지하지 못하면</u> 다른 프로세스가 끝나기를 기다려야 함
  - 자원을 할당 받으면 유한한 시간 후에 반납해야 함

Banker's 알고리즘을 위한 자료구조

- n : 프로세스 수, m : 자원 유형의 수
- Available : (크기 m인 벡터) 자원 유형별 가용 자원 수
  - **Available[j] = k** : 자원 Rj 를 k개 사용 가능
- Max, Allocation, Need : (n x m 행렬, 행은 프로세스, 열은 자원 번호)
  - **MAX[i, j] = k** : 프로세스 Pi가 자원 Rj를 최대 k개 필요로 함
  - **Allocation[i, j] = k** : 프로세스 Pi가 자원 Rj를 k개 할당 받음(사용 중)
  - **Need[i,j] = k** : 프로세스 Pi가 자원 Rj를 향후에 k개 더 필요로 함

Need[i,j] = Max[i,j] – Allocation[i,j]

- **Max_i, $Allocation_i$, $Need_i$** : 행렬의 i 행
  (프로세스 Pi의 필요자원, 할당자원 정보)

안전성 알고리즘(Safety Algorithm)

1. **Work** : 길이 m의 벡터 (자원)
   Finish: 길이 n의 벡터 (프로세스 종료 여부)
   $Work = Available$
   $Finish[i] = false$ for $i = 0, 1, \ldots, n-1$.
2. Find an ***i*** such that both
   (a) $Finish[i] = false$
   (b) $Need_i \leq Work$
   If no such *i* exists, go to step 4.

3. $Work = Work + Allocation_i$ // Pi는 할당 자원을 반납, 가용자원 증가
   $Finish[i] = true$
   go to step 2.

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

예: Banker's Algorithm 사용한 safety 검사

- 5 processes: P_0 ~ P_4;
  3 resource types: *A* (10 instances), *B* (5 insts), *C* (7 insts).
  - initially, *Available* = (10 5 7)
- 시간 T_0 에서의 시스템 상태 및 *Need* (= *Max* – *Allocation*)

	Allocation	*Max*	*Available*	→ *Need*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	**3 3 2**	7 4 3
P_1	**2 0 0**	**3 2 2**		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

- < P_1, P_3, P_4, P_2, P_0>이 safe sequence이므로 시스템은 safe 상태에 있음
  - Work 변동 과정: (3 3 2) → P1 (5 3 2) → P3 (7 4 3) → P4 (7 4 5) → P2 (10 4 7) →P0 (10 5 7)

자원 요청(Resource-Request) 알고리즘

- ***Request$_i$*** : request vector for process P_i.
  - ***Request$_i$* [*j*] = *k*** : process P_i 가 k개의 자원 R_j 를 요청
- Resource-Request Algorithm
  1. If $Request_i \le Need_i$, go to step 2.
     Otherwise, error (최대 요구량을 초과함)
  2. If $Request_i \le Available$, go to step 3.
     Otherwise P_i wait, (현재 필요자원이 사용 가능하지 않음)
  3. P_i 에 요청 자원을 할당한 것 처럼 상태를 변경하여 safety 검사

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- *If safe* ⇒ P_i 에 자원 할당
- *If unsafe* ⇒ P_i 는 대기하고, 변경 이전의 자원 할당 상태를 복원함

예: P_1 Request (1,0,2)

- Request ≤ Available 여부 확인: (1,0,2) ≤ (3,3,2) ⇒ true.
- 상태 갱신:

	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	

- safety 알고리즘 수행 : < P_1, P_3, P_4, P_0, P_2>는 safe sequence이므로 자원 요청을 승인

  Work: (2 3 0)→P_1(5 3 2) → P_3(7 4 3) → P_4(7 4 5) → P_0(7 5 5) → P_2(10 5 7)
- 질문
  - P_4 request (3,3,0) ➔ 자원 요청 승인?
  - P_0 request (0,2,0) ➔ 자원 요청 승인?

7.6 교착상태 탐지(Deadlock Detection)

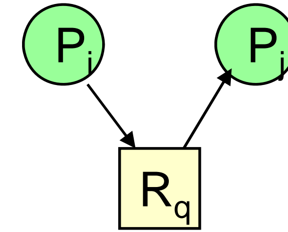
- 교착상태 예방 또는 회피 알고리즘을 사용하지 않으면
  ➔ 교착상태(deadlock) 발생 가능

- 이러한 시스템에서 교착 상태 처리 방법
  - 교착상태 **탐지** 알고리즘(Deadlock Detection algorithm)
  - 교착상태로부터의 **복구** 방법 (Recovery scheme)

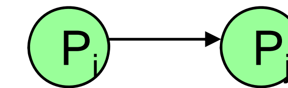
- 2가지 자원 유형
  - 단일 인스턴스 자원 유형
  - 다수의 인스턴스를 갖는 자원 유형

단일 인스턴스 자원 유형

- 대기 그래프(Wait-for graph)
  - 자원 할당 그래프에서 자원 노드를 제거
  - **Nodes**는 프로세스들을 나타냄
  - **edge $P_i \rightarrow P_j$** :
    - P_i 가 P_j의 점유 자원을 대기함의 의미
    - 자원할당 그래프에서 $P_i \rightarrow R_q$, $R_q \rightarrow P_j$ 일 때에 $P_i \rightarrow P_j$ 존재



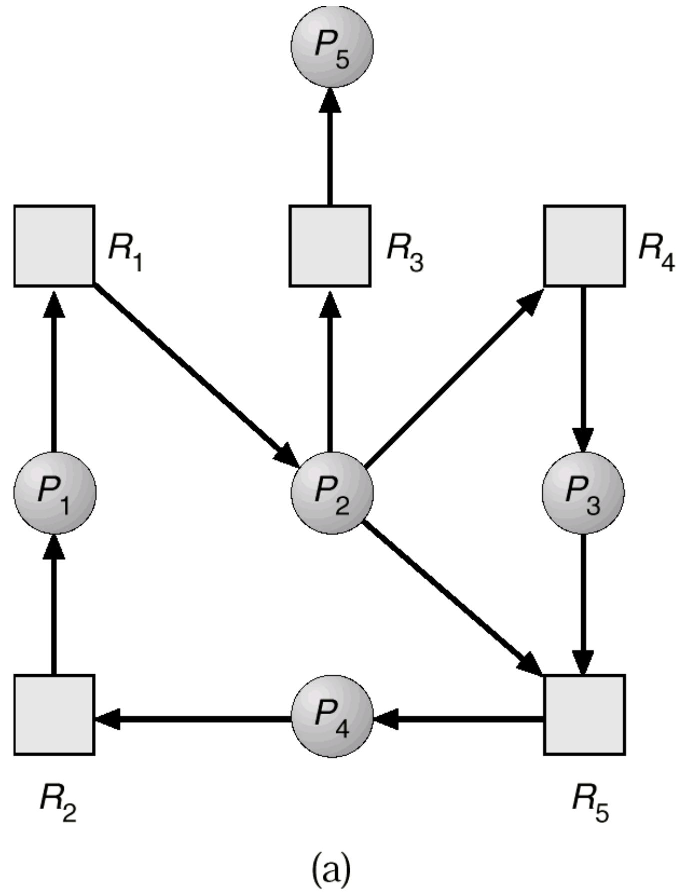
resource allocation graph



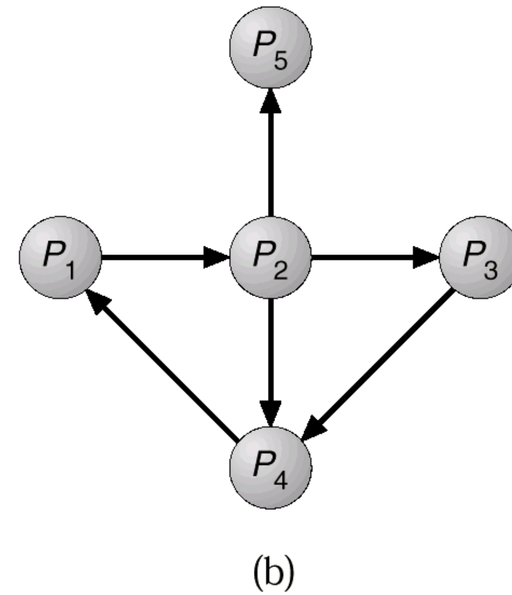
wait-for graph

- 교착상태 탐지 알고리즘
  - wait-for graph에서 cycle이 존재하는 지 검사
  - cycle이 발생하면 deadlock 가능성
  - cycle 존재 여부 검사에 $O(n^2)$ 연산 소요 (n : 노드 수)

자원 할당 그래프와 Wait-for 그래프



Resource-Allocation Graph



Corresponding wait-for graph

여러 개의 인스턴스를 갖는 자원 유형

- 현재의 자원 상태에 safety 를 검사하여 교착상태를 탐지함
  - Banker's algorithm 사용
- 자원 상태
  - **Available**: 길이 m인 벡터 (자원 유형 별 가용 인스턴스 수)
  - **Allocation**: n x m 행렬(각 프로세스에 할당된 유형별 자원 인스턴스 수)
  - **Request**: n x m 행렬 (각 프로세스의 현재 요청)
- 교착상태 탐지 알고리즘
  - Safety 검사 알고리즘에서 Need를 Request로 바꾼 것을 제외하면 기본적으로 같음
  - 현재 상태에 대해서 system이 safe 상태가 아니면(unsafe 상태) 교착상태로 탐지

탐지 알고리즘 사용

- **탐지 알고리즘 수행 빈도**는 다음을 고려하여 결정
  - 교착상태가 얼마나 자주 발생하는가?
  - 교착상태가 발생하면 몇 개의 프로세스가 영향을 받는가?
- 방법 1 : 요청이 즉시 승인되지 않을 때마다 탐지 알고리즘 수행
  - 교착상태를 유발한 특정 프로세스를 식별 가능
  - 오버헤드가 큼
- 방법 2 : 가끔씩 탐지 알고리즘 수행
  (한 시간에 한번, 또는 자원 이용율이 일정 기준 이하가 될 때)
  - 교착상태를 유발한 프로세스를 식별 불가능

7.7 교착상태로 부터의 복구

- 교착상태 복구 방법
  - 프로세스 종료
  - 자원 선점
- 프로세스 종료 – 2가지 방식
  - 교착상태에 있는 모든 프로세스 중지(abort)
  - 교착상태가 제거될 때까지 한 번에 한 프로세스씩 중지
- 부분 종료 방식에서 중지될 프로세스 선택 시 고려 사항
  - 프로세스 우선 순위
  - 지금까지 수행된 시간, 종료에 필요한 시간
  - 프로세서가 사용한 자원
  - 종료에 필요한 자원
  - 종료될 필요가 있는 프로세스 개수
  - 대화형/일괄처리(batch) 여부

자원 선점

- 자원 선점
  - 교착상태가 제거될 때까지 프로세스의 자원을 선점하여 다른 프로세스에게 제공함
- 희생자(victim) 선택
  - 어느 자원과 어느 프로세스가 선점될 것인가?
  - 비용을 최소화하는 선점 순서 결정
- 롤백(Rollback)
  - safe 상태로 되돌린 후, 그 상태에서 다시 시작
  - total rollback 방법 - safe 상태의 결정이 어려우므로 프로세스 중지 후 다시 시작
  - 부분 rollback 방법 – 교착상태를 제거하는 데 필요한 만큼 rollback
- 기아상태(Starvation) 고려
  - 같은 프로세스가 항상 victim으로 선택되는 기아 상태가 발생하지 않도록 해야 함
  - 해결책: rollback 횟수를 비용에 포함