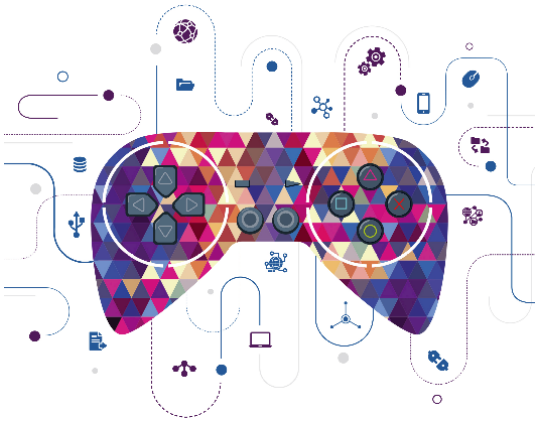


그림으로 배우는 구조와 원리

운영체제 개정 3판

Chatpter 05
교착 상태와 기아 상태

- 교착 상태의 개념과 발생 원인을 이해
- 교착 상태를 해결하는 예방, 회피, 탐지, 회복에 대한 이해
- ‘식사하는 철학자 문제’를 통해 기아 상태 해결

Section 01 교착 상태의 개념과 발생 원인

교착 상태deadlock의 개념

- 다중 프로그래밍 시스템에서 프로세스가 결코 일어나지 않을 사건event을 기다리는 상태
- 프로세스가 교착 상태에 빠지면 작업 정지되어 명령 진행 불가
- 운영체제가 교착 상태 해결 못하면, 시스템 운영자나 사용자는 작업 교체, 종료하는 외부 간섭으로 해결해야 함
- 하나 이상의 작업에 영향을 주어 무한 대기, 기아 상태보다 더 심각한 문제 야기
- 두 프로세스가 사용하는 자원(비공유) 서로 기다리고 있을 때 발생
- 자원 해제 요청 받아들일 때까지 프로세스들은 작업 진행 불가
- 자원 해제 수신 때까지 현재 보유 자원도 해제 불가
- 예

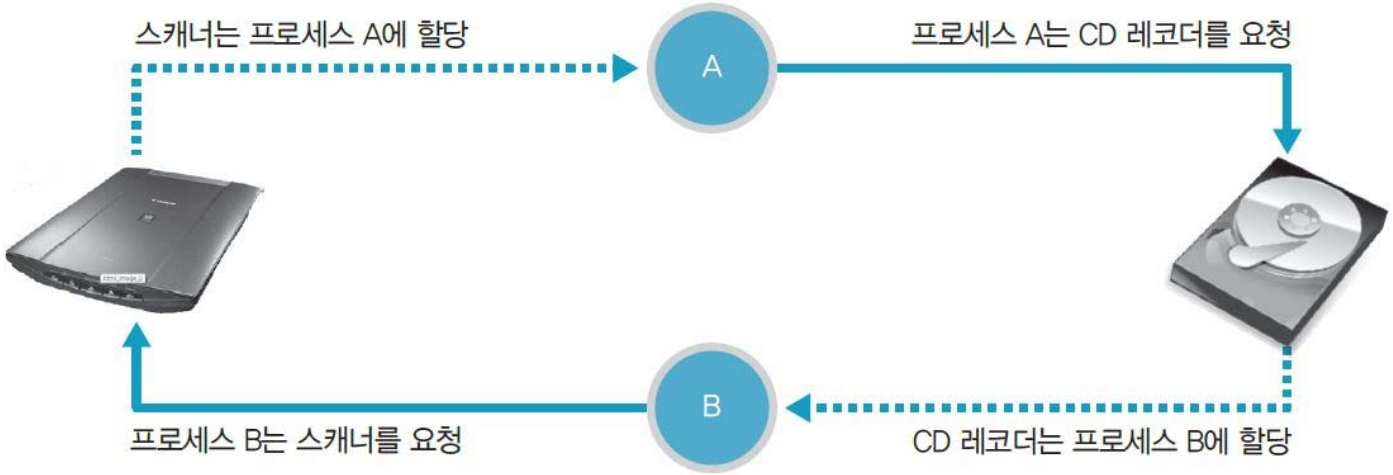


그림 5-1 교착 상태 예

1. 교착 상태의 개념

- 일상 생활에서 교착 상태

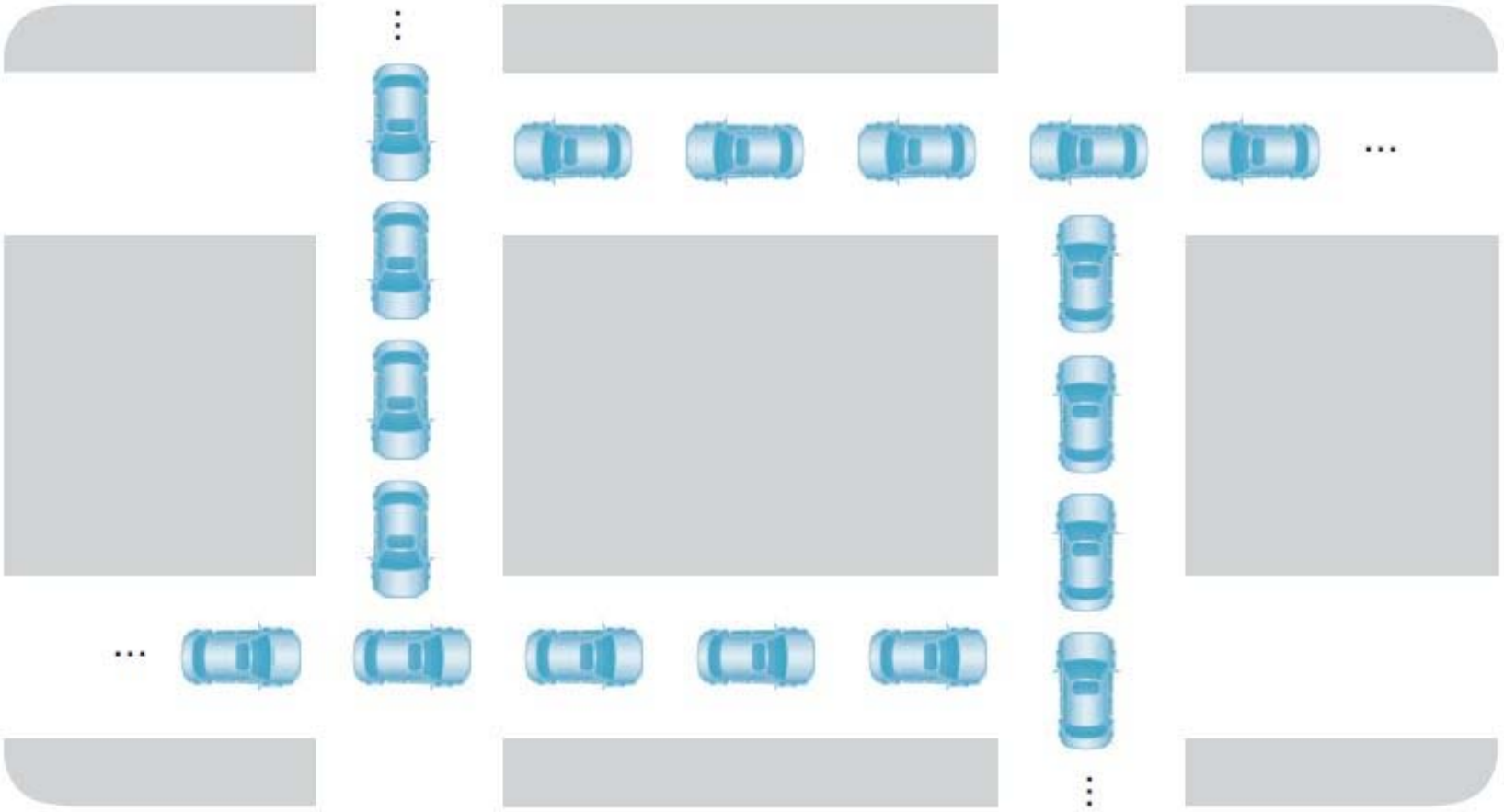


그림 5-2 교착 상태 예 : 교통마비 상태

교착 상태를 해결하기 위한 외부 간섭 필요

1. 교착 상태의 개념

■ 프로세스의 자원 사용 순서

❶ 자원 요청 : 프로세스가 필요한 자원 요청
해당 자원 다른 프로세스가 사용 중이면 요청을 수락 때까지 대기

❷ 자원 사용 : 프로세스가 요청한 자원 획득하여 사용

❸ 자원 해제 : 프로세스가 자원 사용 마친 후 해당 자원 되돌려(해제) 줌

2.교착 상태의 예

■ 컴퓨터 시스템에서 교착 상태의 발생 예

▪ 스풀링 시스템에서 발생하는 교착 상태

- 스풀링 시스템 쉽게 교착 상태에 빠짐. 디스크에 할당된 스풀 공간에서 출력 완료하지 않은 상태에서 다른 작업이 스풀 공간 모두 차지하면 교착 상태 발생
- 스풀링 처리부에 공간이 넉넉하면 교착 상태 발생률 감소하나 비용 많이 듬. 이때는 스풀링 파일의 일정 포화 임계치saturation threshold 설정하여 교착 상태 예방 가능

▪ 디스크를 공유할 때 발생하는 교착 상태

- 디스크 사용에 제어가 없으면 프로세스들이 서로 충돌하는 명령 요청할 때 교착 상태 발생
- 예

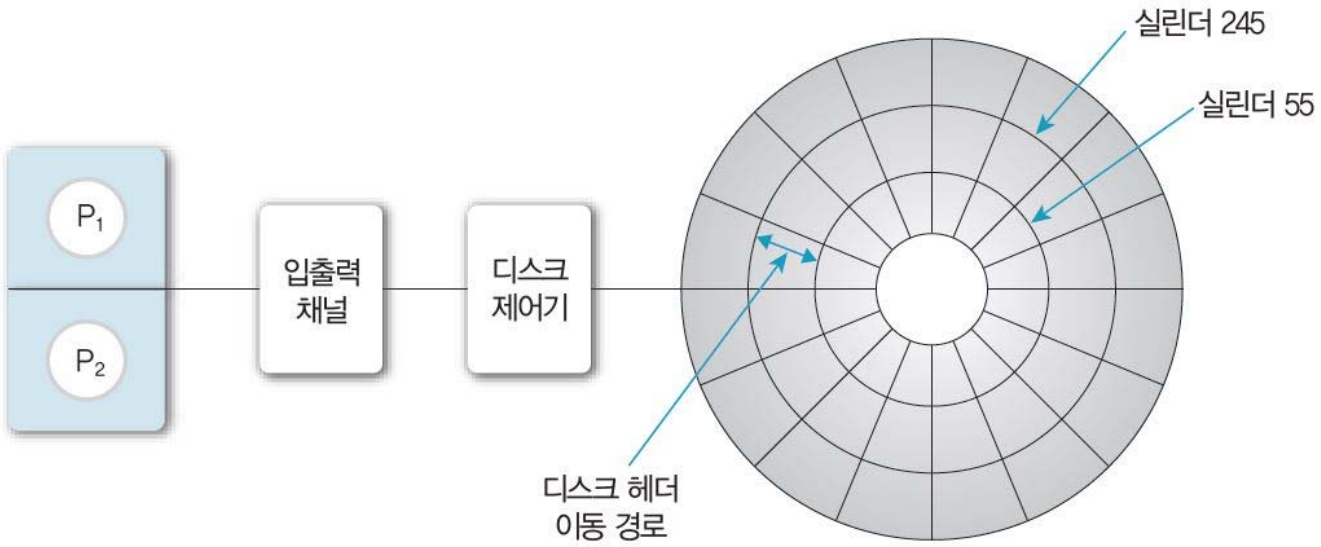


그림 5-3 디스크 제어기가 프로세서와 독립적으로 작동할 때 발생하는 교착 상태

2. 교착 상태의 예

- 네트워크에서 발생하는 교착 상태
  - 네트워크가 붐비거나 입출력(I/O) 버퍼 공간이 부족한 네트워크 시스템에 메시지 흐름 제어하는 적절한 프로토콜 없으면, 교착 상태가 발생
  - 예

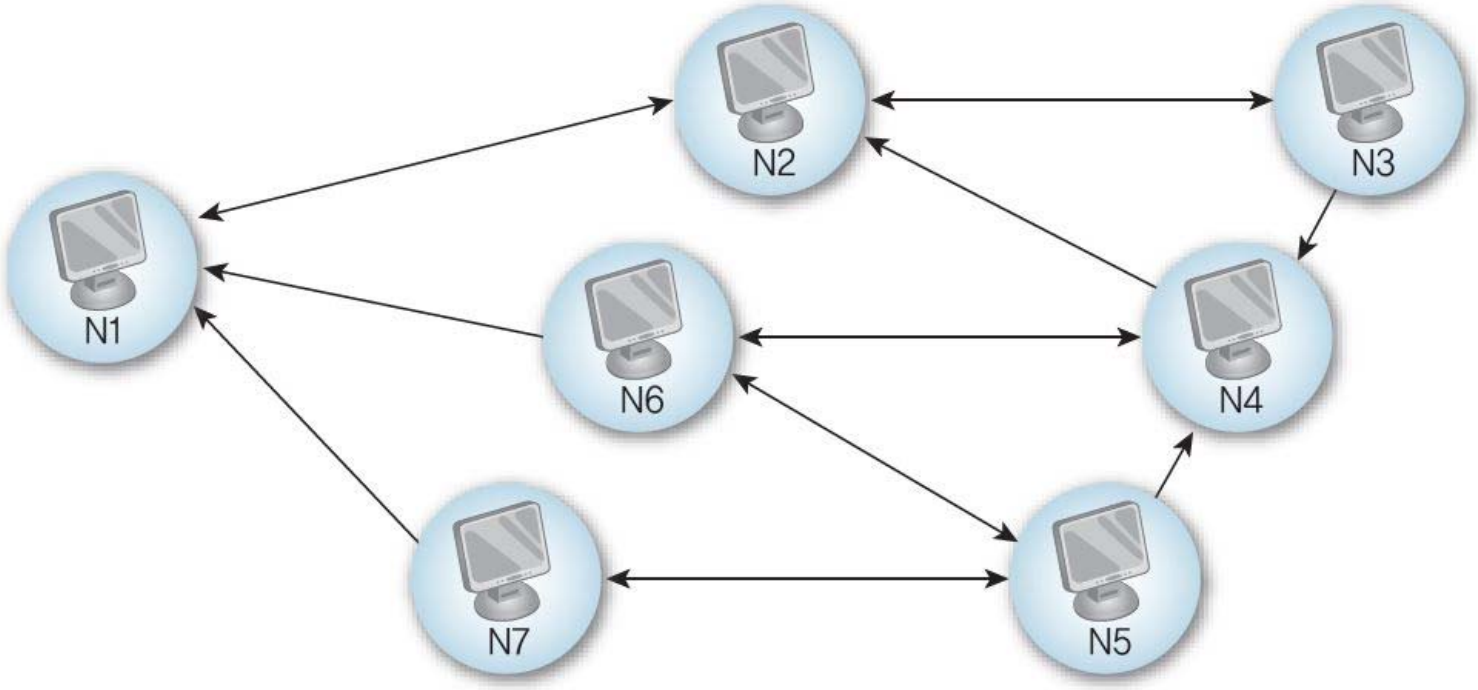


그림 5-4 네트워크에서 발생하는 교착 상태

3. 교착 상태의 발생 조건

■ 교착 상태 발생의 네가지 조건

❶~❸만 만족해도 교착 상태가 발생 가능, 발생하지 않을 수도 있음
❹는 ❶~❸ 조건을 만족할 때 발생 할 수 있는 결과, 점유와 대기 조건을 포함하므로 네 가지 조건이 모두 독립적인 것은 아님

❶ 상호배제

- 자원을 최소 하나 이상 비공유. 즉, 한 번에 프로세스 하나만 해당 자원 사용할 수 있어야 함
- 사용 중인 자원을 다른 프로세스가 사용하려면 요청한 자원 해제될 때 까지 대기

❷ 점유와 대기

- 자원을 최소한 하나 정도 보유, 다른 프로세스에 할당된 자원 얻으려고 대기하는 프로세스 있어야 함

❸ 비선점

- 자원 선점 불가. 즉, 자원은 강제로 빼앗을 수 없고, 자원 점유하고 있는 프로세스 끝나야 해제

❹ 순환(환형) 대기

- 예

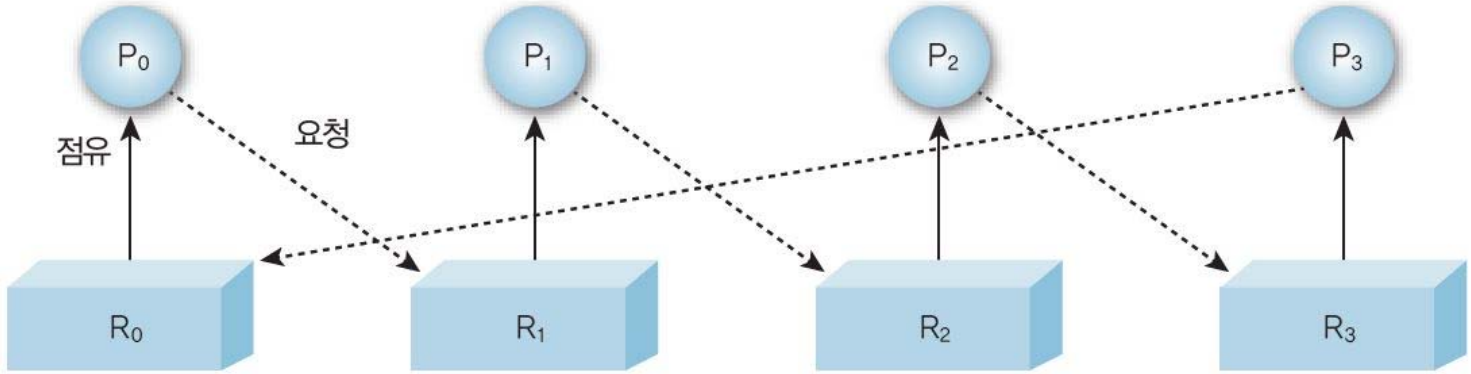


그림 5-5 순환 대기의 교착 상태

3. 교착 상태의 발생 조건

교착 상태의 예

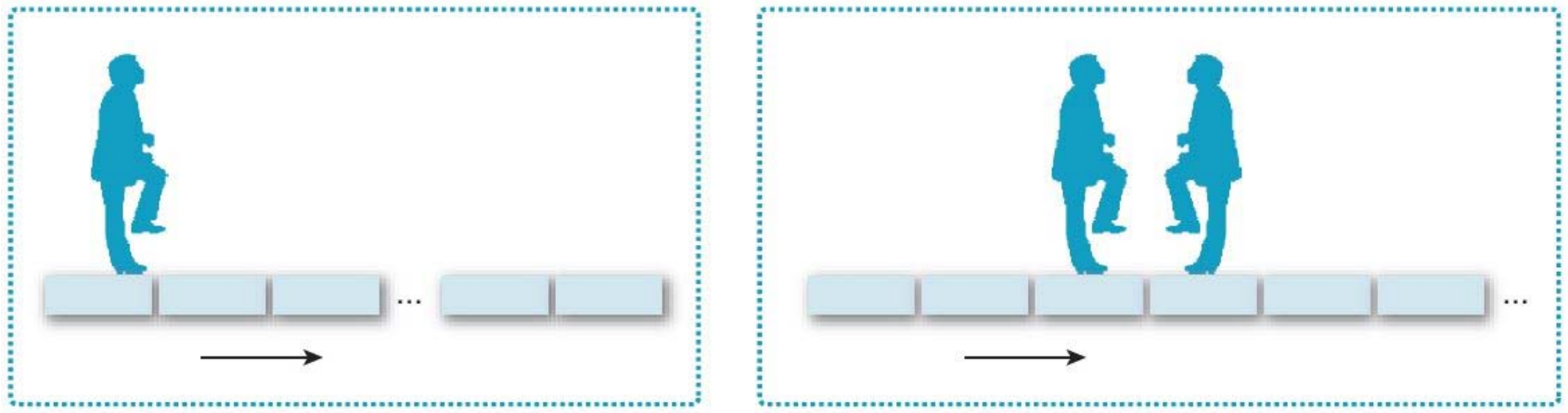


그림 5-6 강 건너기 예로 살펴본 교착 상태

· 상호배제 : 돌 하나를 한 사람만 디딜 수 있음
· 점유와 대기 : 각 사람은 돌 하 나를 딛고 다음 돌을 요구
· 비선점 : 사람이 딛고 있는 돌을 강 제로 제거할 수 없음
· 순환 대기 : 왼쪽에서 오는 사람은 오른쪽에서 오는 사람 기다리고, 오른쪽에서 오는 사람도 왼쪽에서 오는 사람 기다림

교착 상태 해결 방법

❶ 둘 중 한 사람이 되돌아간다(복귀)

❷ 징검다리 반대편을 먼저 확인하고 출발

❸ 강의 한편에 우선순위를 부여

3. 교착 상태의 발생 조건

식사하는 철학자 문제와 교착 상태 필요조건

- **상호 배제** : 포크는 한 사람이 사용하면 다른 사람이 사용할 수 없는 배타적인 자원임
- **비선점** : 철학자 중 어떤 사람의 힘이 월등하여 옆 사람의 포크를 빼앗을 수 없음
- **점유와 대기** : 한 철학자가 두 자원(왼쪽 포크와 오른쪽 포크)을 다 점유하거나, 반대로 두 자원을 다 기다릴 수 없음
- **원형 대기** : 철학자들은 둥그런 식탁에서 식사를 함, 원을 이룬다는 것은 선후 관계를 결정할 수 없어 문제가 계속 맴돈다는 의미(사각형 식탁에서 한 줄로 앉아서 식사를 한다면 교착 상태가 발생하지 않음)

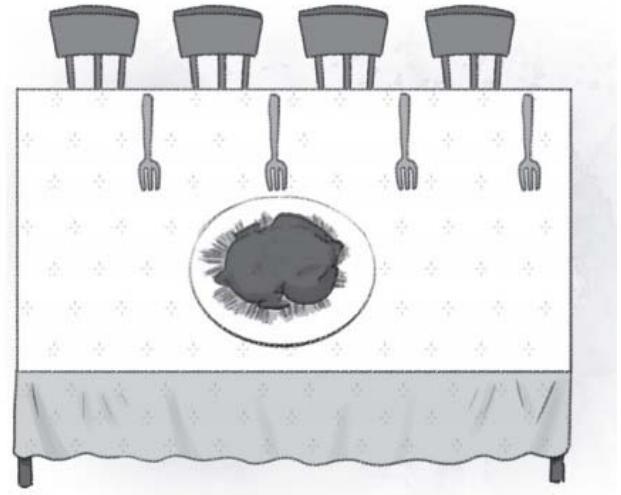


그림 6-10 사각형 식탁과 식사하는 철학자 문제

4. 교착 상태의 표현

교착 상태의 표현

- 시스템 자원 할당 그래프인 방향 그래프 표현

- 자원 할당 그래프
  - G= (V, E)로 구성, 정점 집합 V는 프로세스 집합 P= {P1, P2, ..., Pn}과 자원 집합 R= {R1, R2, ..., Rn} 으로 나뉨. 간선 집합 E는 원소를 (Pi, Rj)나 (Rj, Pi)와 같은 순서쌍으로 나타냄
  - 예

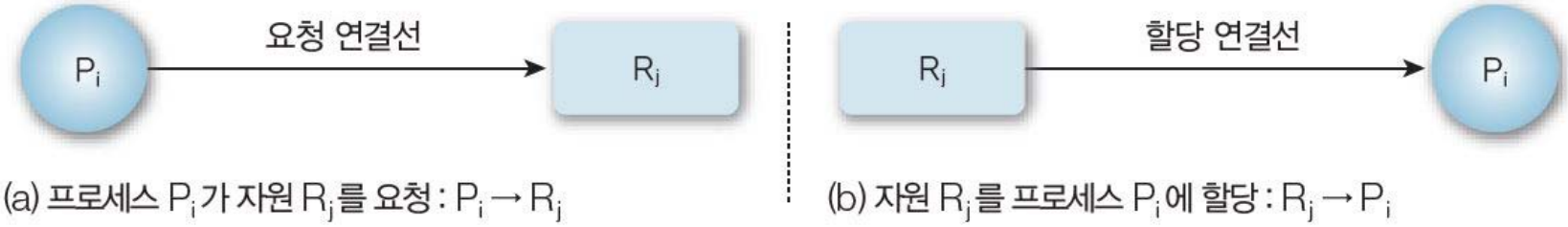


그림 5-7 자원 할당 그래프

4. 교착 상태의 표현

- 사이클이 있어 교착 상태인 자원 할당 그래프

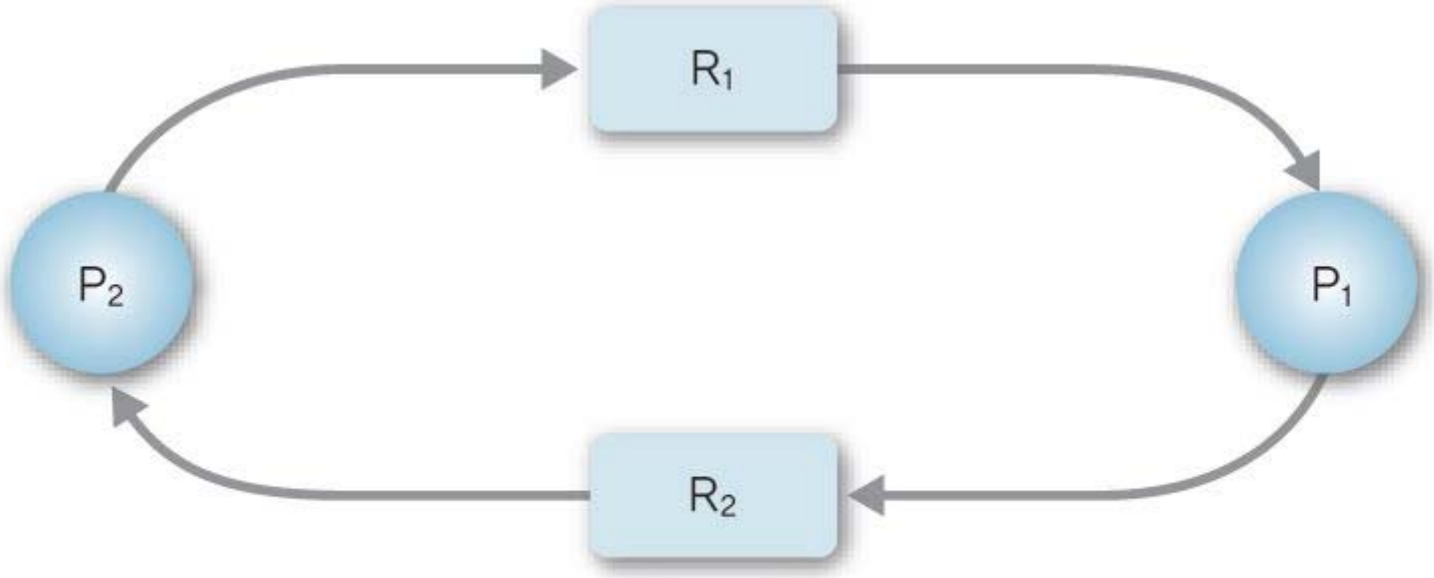
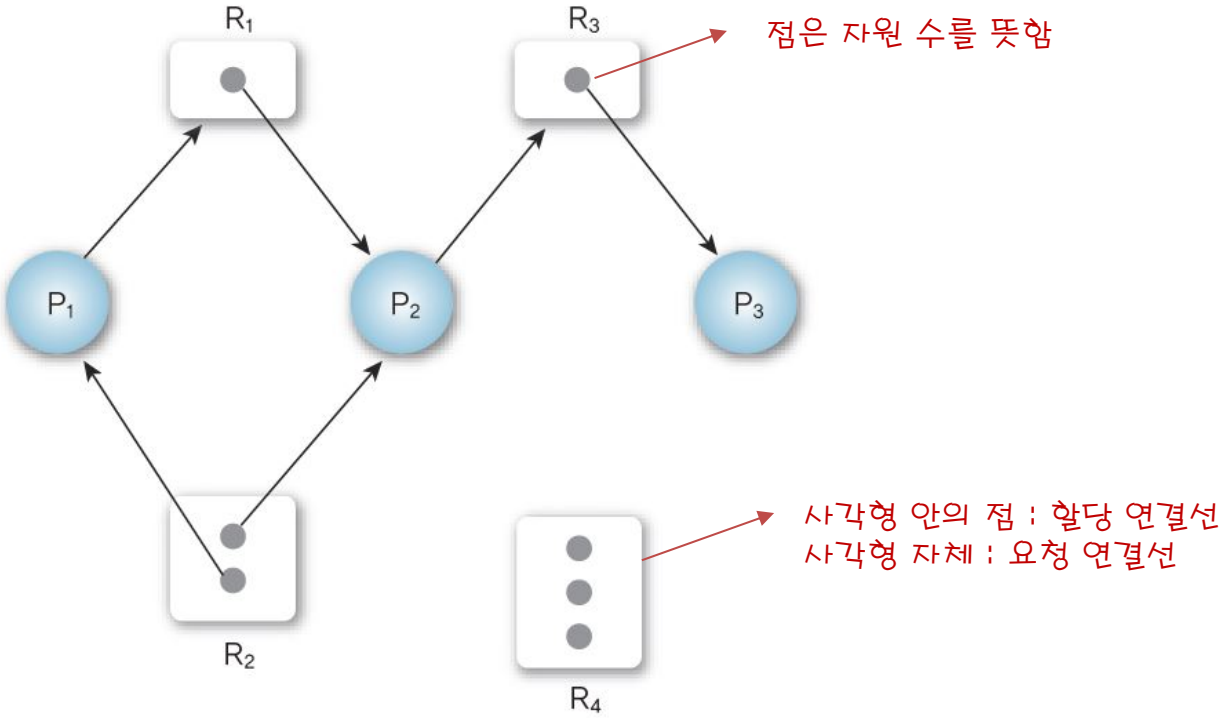


그림 5-8 사이클이 있어 교착 상태인 자원 할당 그래프 : 프로세스 P_1, P_2가 교착 상태

4. 교착 상태의 표현

- 자원 할당 그래프의 예



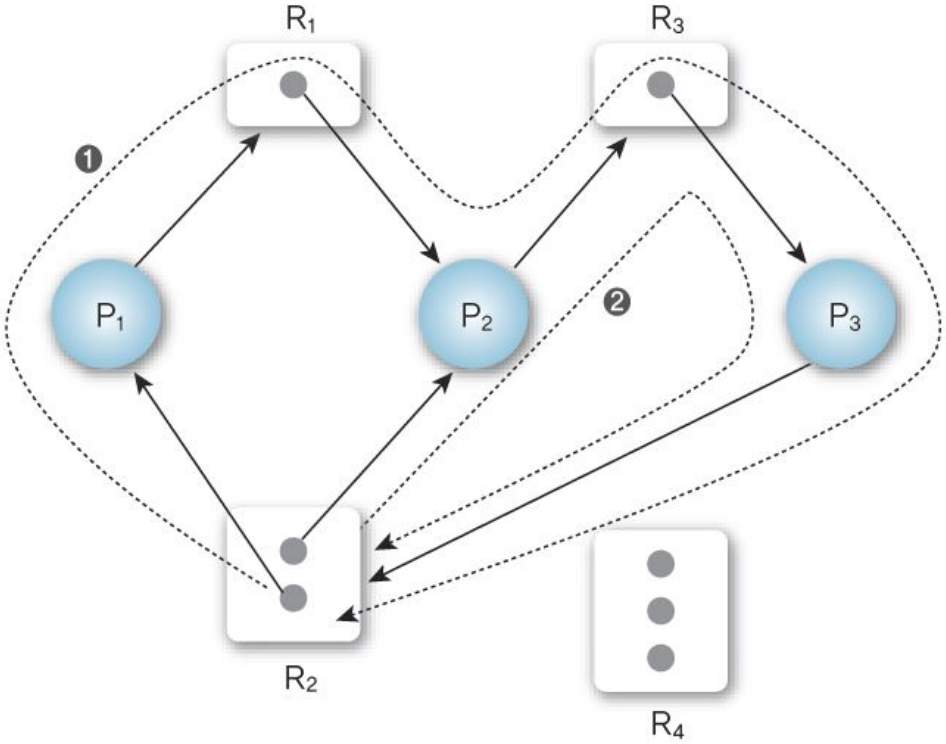
❶ 집합 P, R, E
- $P = \{P_1, P_2, P_3\}$
- $R = \{R_1, R_2, R_3, R_4\}$
- $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

❷ 프로세스의 상태
- 프로세스 P_1은 자원 R_2의 자원을 하나 점유하고, 자원 R_1을 기다린다.
- 프로세스 P_2는 자원 R_1과 R_2의 자원을 각각 하나씩 점유하고, 자원 R_3을 기다린다.
- 프로세스 P_3은 자원 R_3의 자원 하나를 점유 중이다.

그림 5-9 **자원 할당 그래프 예와 프로세스 상태**

4. 교착 상태의 표현

- 교착 상태의 할당 그래프와 사이클



(a) 교착 상태의 할당 그래프

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

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

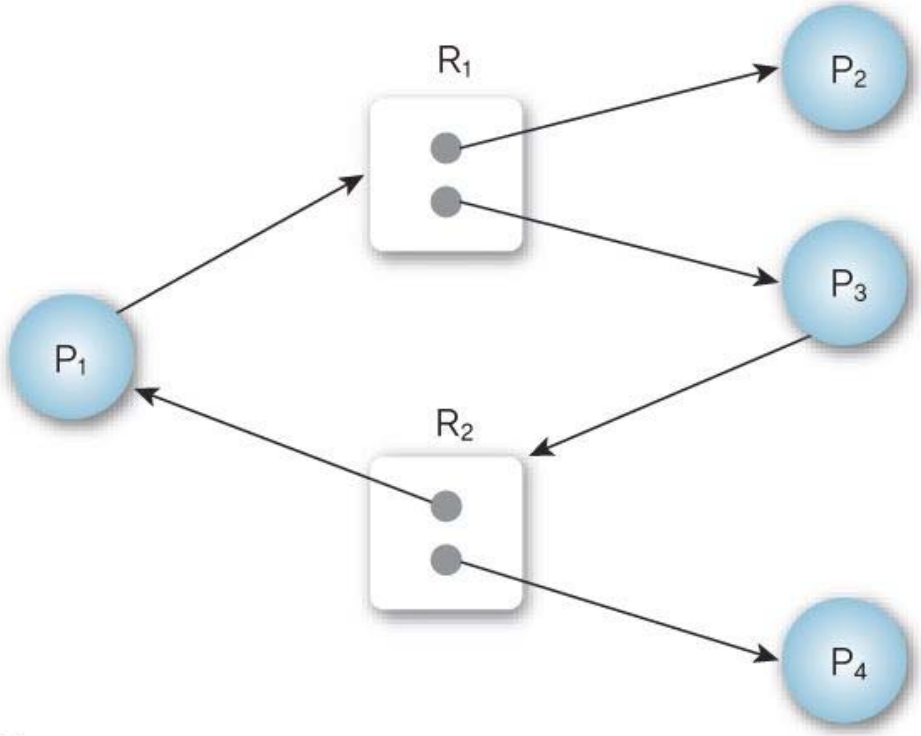
(b) (a)에 있는 사이클

그림 5–10 교착 상태의 할당 그래프와 사이클

그래프에 있는 사이클은 교착 상태 발생의 필요 조건이지 충분조건 아님

4. 교착 상태의 표현

- 사이클이 있으나 교착 상태가 아닌 할당 그래프



(a) 교착 상태의 할당 그래프

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

(b) (a)에 있는 사이클

그림 5-11 사이클이 있으나 교착 상태가 아닌 할당 그래프

자원 할당 그래프에 사이클이 없다면 교착 상태 아님
그러나 사이클이 있다면 시스템은 교착 상태일 수도 있고 아닐 수도 있음

Section 02 교착 상태의 해결 방법

■ 교착 상태 해결 방법 세 가지

❶ 예방prevention

❷ 회피avoidance

❸ 탐지detection, 회복

1. 교착 상태 예방

하벤더Havender의 교착 상태 예방 방법

- 각 프로세스는 필요한 자원 한 번에 모두 요청해야 하며, 요청한 자원을 모두 제공받기 전까지는 작업 진행 불가
- 어떤 자원을 점유하고 있는 프로세스의 요청을 더 이상 허용하지 않으면 점유한 자원을 모두 반납하고, 필요할 때 다시 자원 요청
- 모든 프로세스에 자원 순서대로 할당

보통 교착 상태 예방 방법

❶ 자원의 상호배제 조건 방지
❷ 점유와 대기 조건 방지
❸ 비선점 조건 방지
❹ 순환(환형) 대기 조건 방지

1. 교착 상태 예방

자원의 상호배제 조건 방지

- 상호배제는 자원의 비공유가 전제 되어야 함
- 일반적으로 상호배제 조건 만족하지 않으면 교착 상태 예방 불가능

점유와 대기 조건 방지

- 점유와 대기hold and wait 조건 발생 않으려면, 프로세스가 작업 수행 전에 필요한 자원 모두 요청하고 획득해야 함
- 대기 상태에서는 프로세스가 자원 점유 불가능하므로 대기 조건 성립 안됨(최대 자원 할당)
- 점유와 대기 조건 방지 방법
  - 자원 할당 시 시스템 호출된 프로세스 하나를 실행하는 데 필요한 모든 자원 먼저 할당, 실행 후 다른 시스템 호출에 자원 할당
  - 프로세스가 자원을 전혀 갖고 있지 않을 때만 자원 요청할 수 있도록 허용하는 것. 프로세스가 자원을 더 요청하려면 자신에게 할당된 자원을 모두 해제해야 함
- 점유와 대기 조건 방지 방법의 단점
  - 자원 효율성 너무 낮음
  - 기아 상태 발생 가능(대화식 시스템에서 사용 불가)

1. 교착 상태 예방

비선점 조건 방지non-preemption

- 어떤 자원을 가진 프로세스가 다른 자원 요청할 때 요청한 자원을 즉시 할당 받을 수 없어 대기해야 한다면, 프로세스는 현재 가진 자원 모두 해제. 그리고 프로세스가 작업 시작할 때는 요청한 새로운 자원과 해제한 자원 확보해야 함. 이런 방법은 비선점 조건을 효과적으로 무효화시키지만 이미 실행한 작업의 상태를 잃을 수도 있음. 따라서 작업 상태를 쉽게 저장, 복구할 수 있을 때나 빈번하게 발생하지 않을 때만 좋은 방법
- 전용 입출력장치 등을 빼앗아 다른 프로세스에 할당 후 복구하는 과정 간단하지 않음. 대안으로 프로세스가 어떤 자원을 요청할 때 요청한 자원이 사용 가능한지 검사, 사용할 수 있다면 자원 할당. 사용할 수 없다면 대기 프로세스가 요청한 자원을 점유하고 있는지 검사. 요청한 자원을 대기 프로세스가 점유하고 있다면, 자원을 해제하고 요청 프로세스에 할당. 요청한 자원을 사용할 수 없거나 실행 중인 프로세스가 점유하고 있다면 요청 프로세스는 대기. 프로세스가 대기하는 동안 다른 프로세스가 점유한 자원을 요청하면 자원을 해제할 수 있음.
- 또 다른 방법은 두 프로세스에 우선순위 부여하고 높은 우선순위의 프로세스가 그보다 낮은 우선 순위의 프로세스가 점유한 자원 선점하여 해결. 이 방법은 프로세서 레지스터나 기억장치 레지스터와 같이 쉽게 저장되고 이후에 다시 복원하기 쉬운 자원에 사용. 프린터, 카드 판독기, 테이프 드라이버 같은 자원에는 적용 않음

1. 교착 상태 예방

■ 순환(환형) 대기 조건 방지

- 모든 자원에 일련의 순서 부여, 각 프로세스가 오름차순으로만 자원을 요청할 수 있게 함
- 이는 계층적 요청 방법으로 순환 대기circular wait의 가능성 제거하여 교착 상태 예방.
- 예상된 순서와 다르게 자원을 요청하는 작업은 실제로 자원을 사용하기 전부터 오랫동안 자원 할당받은 상태로 있어야하므로, 상당한 자원 낭비 초래
- 자원 집합을 R= {R1, R2, ..., Rn}이라고 하자. 각 자원에 고유 숫자 부여 어느 자원의 순서가 빠른지 알 수 있게 함
- 이것은 1:1 함수 F:R → N으로 정의(여기서 N은 자연수 집합 의미).
- 자원 R의 집합이 CD 드라이브, 디스크 드라이브, 프린터 포함한다면 함수 F는 다음과 같이 정의

F(CD 드라이브) = 2,

F(디스크 드라이브) = 4,

F(프린터) = 7

- 단점) 프로세스 작업 진행에 유연성이 떨어짐, 자원의 번호를 어떻게 부여할 지가 문제

1. 교착 상태 예방

교착 상태 예방 시 고려할 규칙

- 각 프로세스는 오름차순으로만 자원 요청 가능
- 즉, 프로세스는 임의의 자원 Ri 요청할 수 있지만 그 다음부터는 F(Rj) > F(Ri) 일 때만 자원 Rj 요청 가능. 데이터 형태 자원이 여러 개 필요하다면, 우선 요청할 형태 자원 하나 정해야 함. 앞서 정의한 함수 이용하면, CD 드라이브와 프린터를 동시에 사용해야 하는 프로세스는 CD 드라이브 먼저 요청 후 프린터 요청
- 또 다른 해결 방법으로 프로세스가 자원 Rj 요청 때마다 F(Ri)≥F(Rj)가 되도록 Rj의 모든 자원 해제하는 것. 그러면 순환 대기 조건 막을 수 있음
- 순환 대기 성립한다는 가정하에서 사실의 성립(모순에 의한 증명)
  - 순환 대기의 프로세스 집합을 {P0, P1, ..., Pn}이라고 하자. 이때 Pi는 프로세스 Pi +1이 점유 한 자원 Ri 대기(첨자는 모듈러 n. Pn은 P0이 점유한 자원 Rn을 대기. 모듈러 관계이므로 한 바퀴 뒤로 돌면 Pn은 Rn을 기다리고, P0이 Rn을 갖게 됨). 프로세스 Pi +1은 자원 R_i을 요청하는 동안 자원 R를 점유하므로 모든 i에 대하여 F(R) < F(R_i)이 성립 되도록 해야 함. 이는 F(R) < F(R) < ..... < (R) < F(R)이 성립함 의미. 즉, F(R) < F(R)은 불가능. 그러므로 여기서는 순환 대기 불가능. 함수 F는 시스템에 있는 자원의 정상적인 이용 순서에 따라서 정의해야 함
- 계층적 요청은 순환 대기 조건 가능성 제거하여 교착 상태 예방, 반드시 자원의 번호 순서로 요청해야 함. 또 번호 부여할 때 실제로 자원 사용하는 순서를 반영해야 한다

2. 교착 상태 회피

교착 상태 회피의 개념

- 목적 : 덜 엄격한 조건 요구하여 자원 좀 더 효율적 사용
- 교착 상태의 모든 발생 가능성을 미리 제거하는 것이 아닌 교착 상태 발생할 가능성 인정하고(세 가지 필요조건 허용), 교착 상태가 발생하려고 할 때 적절히 회피하는 것
- 예방보다는 회피가 더 병행성 허용

- 교착 상태의 회피 방법
  - 프로세스의 시작 중단
    - 프로세스의 요구가 교착 상태 발생시킬 수 있다면 프로세스 시작 중단
  - 자원 할당 거부(알고리즘Banker's algorithm)
    - 프로세스가 요청한 자원 할당했을 때 교착 상태 발생할 수 있다면 요청한 자원 할당 않음

- 교착 상태의 회피의 문제점
  - 프로세스가 자신이 사용할 모든 자원을 미리 선언해야 함
  - 시스템의 전체 자원 수가 고정적이어야 함
  - 자원이 낭비됨

2. 교착 상태 회피

프로세스의 시작 중단

- 교착 상태 회피를 위해 자원을 언제 요청하는지 추가 정보 필요. 각 프로세스마다 요청과 해제에서 정확한 순서 파악하고 있다면, 요청에 따른 프로세스 대기 여부 결정 가능

- 프로세스의 요청 수락 여부는 현재 사용 가능한 자원, 프로세스에 할당된 자원 등 각 프로세스에 대한 자원의 요청과 해제를 미리 알고 있어야 결정 가능. 다양한 교착 상태 회피 알고리즘 중에서 가장 단순하고 유용한 알고리즘은 각 프로세스가 필요 한 자원의 최대치(할당 가능한 자원 수)를 선언하는 것. 프로세스가 요청할 자원별로 최대치 정보를 미리 파악할 수 있으면 시스템이 교착 상태가 되지 않을 확실한 알고리즘 만들 수 있음

- 교착 상태 회피 알고리즘은 시스템이 순환 대기 조건이 발생 않도록 자원 할당 상태 검사. 자원 할당 상태는 사용 가능한 자원 수, 할당된 자원 수, 프로세스들 최대 요청 수로 정의. 각 프로세스에 (최대치까지) 자원을 할당할 수 있고 교착 상태를 예방할 수 있으면 안정된 상태. 그리고 시스템에 안정 순서가 있으면 그 시스템은 안정. 순서가 없으면 불안정

2. 교착 상태 회피

■ 시스템의 상태

- 안정 상태와 불안정 상태로 구분
- 교착 상태는 불안정 상태에서 발생
- 모든 사용자가 일정 기간 안에 작업을 끝낼 수 있도록 운영체제가 할 수 있으면 현재 시스템의 상태가 안정, 그렇지 않으면 불안정
- 교착 상태는 불안정 상태. 그러나 모든 불안정 상태가 교착 상태인 것은 아님, 단지 불안정 상태는 교착 상태가 되기 쉬움.
- 상태가 안정하다면 운영체제는 불안정 상태와 교착 상태를 예방 가능
- 불안정 상태의 운영체제는 교착 상태 발생시키는 프로세스의 자원 요청 방지 불가

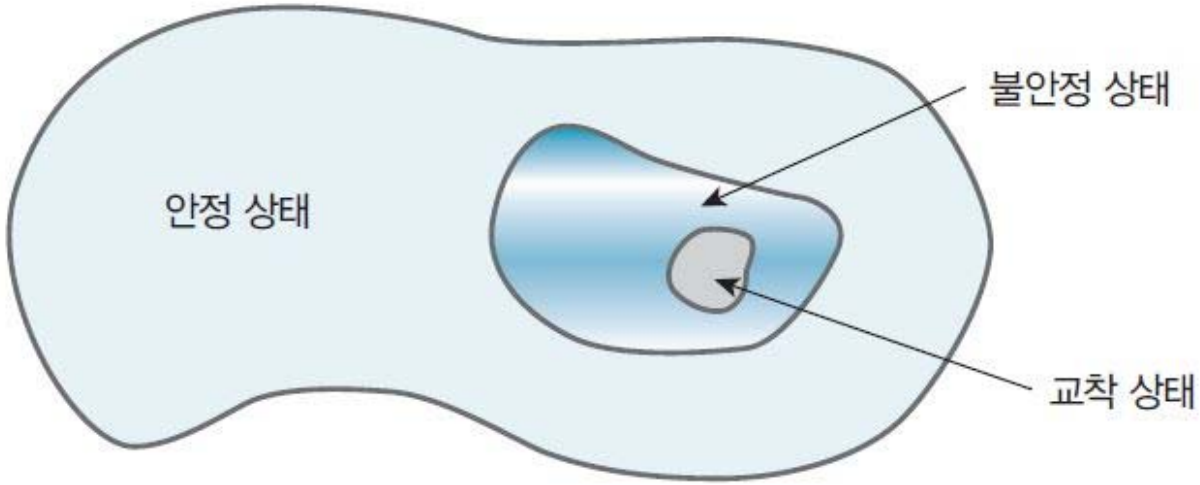


그림 5-12 안정 상태와 불안정 상태, 교착 상태의 공간

2. 교착 상태 회피

■ 안정 상태와 불안정 상태의 자원 예(자원 10개)

프로세스	현재 사용량(t_0 시간)	최대 사용량
P_0	2	7
P_1	1	8
P_2	2	4
P_3	2	10
여분 자원 수		3

(a) 안정 상태의 자원 예

프로세스	현재 사용량(t_0 시간)	최대 사용량
P_0	5	7
P_1	1	8
P_2	2	4
P_3	1	7
여분 자원 수		1

(b) 불안정 상태의 자원 예

그림 5-13 안정 상태와 불안정 상태의 자원 예

불안정 상태는 교착 상태가 발생할 수 있는 가능성이 있다는 의미이지 반드시 교착 상태가 발생한다는 의미는 아님

2. 교착 상태 회피

■ 안정 상태에서 불안정 상태로 변환 예

구분	현재 사용량	최대 사용량
P_0	2	7
P_1	1	8
P_2	2	4
P_3	2	10
사용 가능 자원 수		3

(a) P_0에 자원 할당

안정 $\langle P_2, P_0, P_1, P_3 \rangle$

구분	현재 사용량	최대 사용량
P_0	3	7
P_1	1	8
P_2	2	4
P_3	2	10
사용 가능 자원 수		2

(b) P_1에 자원 할당

불안정

구분	현재 사용량	최대 사용량
P_0	2	7
P_1	2	8
P_2	2	4
P_3	2	10
사용 가능 자원 수		2

P_2는 실행 가능하나 P_0, P_1, P_3은 불안정

그림 5-14 안정 상태에서 불안정 상태로 변화 예

2. 교착 상태 회피

■ 자원 할당 거부(은행가 알고리즘)

▪ 다익스트라의 은행가 알고리즘 이용

- 자원의 할당 허용 여부 결정 전에 미리 결정된 모든 자원의 최대 가능한 할당량maximum을 시뮬레이션 하여 안전 여부 검사. 그런 다음 대기 중인 다른 모든 활동의 교착 상태 가능성 조사하여 '안전 상태' 여부 검사·확인
- 프로세스가 자원 요청 때마다 운영체제로 실행, THE 운영체제의 프로세스 설계 과정에서 개발
- 자원 요청 승낙이 불안전한 상태에서 시스템을 배치할 수 있다고 판단하면 이 요청을 연기, 거부하여 교착 상태 예방. 따라서 각 프로세스에 자원을 어떻게 할당(자원 할당 순서 조정)할 것인지 정보 필요하므로 각 프로세스가 요청하는 자원 종류의 최대 수 알아야 함. 이 정보 이용 교착 상태 회피 알고리즘 정의 가능. 이는 은행에서 모든 고객이 만족하도록 현금 할당하는 과정과 동일

2. 교착 상태 회피

- 은행가 알고리즘 구현을 위한 자료구조
  - Available : 각 형태별로 사용 가능한 자원 수(사용 가능량) 표시하는 길이가 m인 벡터

    Available[j]=k이면, 자원을 k개 사용할 수 있다는 의미
  - Max : 각 프로세스 자원의 최대 요청량(최대 요구량)을 표시하는 n×m 행렬. Max[i, j] =k이면, 프로세스 P는 자원이 R인 자원을 최대 k개까지 요청할 수 있다는 의미
  - Allocation : 현재 각 프로세스에 할당되어 있는 각 형태의 자원 수(현재 할당량) 정의하는 n×m 행렬.

    Allocation[i, j]=k이면, 프로세스 P는 자원이 R인 자원을 최대 k개 할당받고 있다는 의미
  - Need : 각 프로세스에 남아 있는 자원 요청(추가 요구량) 표시하는 n×m 행렬. Need[i, j] =k이면, 프로세스 P는 자신의 작업을 종료하려고 자원 R를 k개 더 요청한다는 의미

N : 시스템의 프로세스 수, m : 자원 수

Need[i, j]=Max[i, j]-Allocation[i, j]라는 식 성립

2. 교착 상태 회피

- 알고리즘 간단 구현을 위한 제약
  - ❶ 시간이 흐르면서 벡터의 크기와 값이 변함
  - ❷ X와 Y는 길이가 n인 벡터
  - ❸ X[i]≤Y[i]이고 i =1, 2, ..., n일 때만 X≤Y
  - ❹ X= (0, 3, 2, 1), Y= (1, 7, 3, 2)이면 X≤Y

2. 교착 상태 회피

■ 프로세스 P_i가 자원 요청 시 일어나는 동작

- 1단계 : $Request_i \le Need_i$이면 2단계로 이동하고, 그렇지 않으면 프로세스가 최대 요청치를 초과하기 때문에 오류 상태가 된다.
- 2단계 : $Request_i \le Available$이면 3단계로 이동하고, 그렇지 않으면 자원이 부족하기 때문에 P_i는 대기한다.
- 3단계 : 시스템은 상태를 다음과 같이 수정하여 요청된 자원을 프로세스 P_i에 할당한다.

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

그림 5-15 은행가 알고리즘

자원 할당이 안정 상태라면 처리가 되고 있는 프로세스 Pi는 자원 할당받음
불안정 상태라면 Pi는 Requesti를 대기하고 이전 자원 할당의 상태로 복귀

2. 교착 상태 회피

■ 안전 알고리즘의 시스템 상태 검사

- 1단계 : Work와 Finish를 각각 길이가 m과 n인 벡터라고 하자. Work = Available, Finish[i] = false, i = 1, 2, ……, n이 되도록 초기화한다.
- 2단계 : 다음 조건을 만족하는 i 값을 찾는다. i 값이 없으면 4단계로 이동한다.

  Finish[i] == false

  $Need_i \leq Work$
- 3단계 : 다음을 수행하고 2단계로 이동한다.

  $Work = Work + Allocation_i$

  Finish[i] = true
- 4단계 : 모든 i에 대하여 Finish[i] == true이면 시스템은 안정 상태이다.

그림 5-16 안전 알고리즘

2. 교착 상태 회피

■ 시간 t_0 일때 시스템의 상태 안정상태

프로세스	Allocation	Max	Need	Available
	ABCD	ABCD	ABCD	ABCD
P_0	2011	3214	1203	1222
P_1	0121	0252	0131	
P_2	4003	5105	1102	
P_3	0210	1530	1320	
P_4	1030	3033	2003	
할당량	7375			

그림 5-17 시간 t_0일 때 시스템의 상태

- 자원 A는 8, B는 5, C는 9, D는 7개 있다 가정
- P2, P0, P4, P1, P3 순서는 안정조건

2. 교착 상태 회피

- 은행가 알고리즘의 단점(교착상태를 회피하려고 가능할 때만 일을 진행시킨다)
  - 할당할 수 있는 자원의 일정량 요청. 자원은 수시로 유지 보수 필요, 고장이나 예방 보수 하기 때문에 일정하게 남아 있는 자원 수 파악 곤란
  - 사용자 수가 일정해야 하지만 다중 프로그래밍 시스템에서는 사용자 수 항상 변함
  - 교착 상태 회피 알고리즘을 실행하면 시스템 과부하 증가
  - 프로세스는 자원 보유한 상태로 끝낼 수 없음. 시스템에서는 이보다 더 강력한 보장 필요
  - 사용자가 최대 필요량을 미리 알려 주도록 요청하지만, 자원 할당 방법이 점점 동적으로 변하면서 사용자의 최대 필요량 파악 곤란. 게다가 최근 시스템은 편리한 인터페이스를 사용자에게 제공하려고 필요한 자원 몰라도 되는 방법 보편화
  - 항상 불안정 상태 방지해야 하므로 자원 이용도 낮음

3. 교착 상태 회복

교착 상태 회복에 필요한 다음 알고리즘

- 시스템 상태 검사하는 교착 상태 탐지 알고리즘
- 교착 상태에서 회복시키는 알고리즘

교착 상태 회복의 특징

- 교착 상태 파악 위해 교착 상태 탐지 알고리즘을 언제 수행해야 하는지 결정하기 어려움
- 교착 상태 탐지 알고리즘 자주 실행하면 시스템의 성능 떨어지지만, 교착 상태에 빠진 프로세스 빨리 발견하여 자원의 유휴 상태 방지 가능. 하지만 자주 실행하지 않으면 반대 상황 발생
- 탐지와 회복 방법은 필요한 정보를 유지하고 탐지 알고리즘을 실행시키는 비용뿐 아니라 교착 상태 회복에 필요한 부담까지 요청

3. 교착 상태 회복

교착 상태 탐지 알고리즘

- 쇼사니Shoshani와 코프만Coffman이 제안

- 은행가 알고리즘에서 사용한 자료구조들과 비슷
  - Available : 자원마다 사용 가능한 자원 수를 표시하는 길이 m 인 벡터
  - Allocation : 각 프로세스에 현재 할당된 각 형태들의 자원 수 표시하는n ×m 행렬
  - Request : 각 프로세스의 현재 요청 표시하는 n×m 행렬. Request[i, j]일 때 프로세스 Pi에 필요한 자원 수가 k개라면, 프로세스 Pi는 자원 Ri의 자원 k개 더 요청

3. 교착 상태 회복

- 교착 상태 탐지 알고리즘 순서 탐지 알고리즘은 남아 있는 프로세스들의 할당 가능 순서 모두 찾음

- 1단계 : Work와 Finish는 각각 길이가 m과 n인 벡터이다.

  Work = Available로 초기화한다. (i = 1, 2, ……, n)일 때 $Allocation_i \neq 0$이면

  Finish[i] = false이고, 아니면 Finish[i] = true이다.

- 2단계 : 다음 조건을 만족하는 색인 i를 찾는다.

  Finish[i] == false

  $Request_i \leq Work$

  조건에 맞는 i가 없으면 4단계로 이동한다.

- 3단계 : 다음 조건과 일치하는지 여부를 판단하여 2단계로 이동한다.

  $Work = Work + Allocation_i$

  Finish[i] = true

- 4단계 : Finish[i] == false라면, $1 \leq i \leq n$인 범위에서 시스템은 교착 상태에 있다.

  또 프로세스 P_i도 교착 상태에 있다.

그림 5-19 교착 상태 탐지 알고리즘

3. 교착 상태 회복

- 시간 t_0일 때 시스템의 상태

현재 이 시스템은 교착 상태에 있지 않음

표 5-1 시간 t_0일 때 시스템의 상태

프로세스	Allocation	Request	Available
	ABC	ABC	ABC
P_0	010	000	000
P_1	200	202	
P_2	303	000	
P_3	211	100	
P_4	002	002	

- 자원 A는 7, B는 2, C는 6개 있다 가정
- P0, P2, P1, P4

3. 교착 상태 회복

- 프로세스 P_2가 자원 C의 자원을 1개 더 요청할 때 현재 이 시스템은 교착 상태

표 5-2 시간 t_0일 때 시스템의 새로운 상태

프로세스	Request
	ABC
P_0	000
P_1	202
P_2	001
P_3	100
P_4	002

3. 교착 상태 회복

- 탐지 알고리즘 호출
  - 교착 상태 발생 빈도수와 교착 상태가 발생했을 때 영향을 받는 프로세스 수에 따라 결정
  - 교착 상태가 자주 발생하면 탐지 알고리즘도 더 자주 호출
  - 일단 교착 상태가 발생하면 해결할 때까지 프로세스에 할당된 자원들은 유휴 상태, 교착 상태의 프로세스 수는 점점 증가
  - 어떤 프로세스라도 허용할 수 없는 요청 하면 즉시 교착 상태. 요청할 때마다 교착 상태 탐지 알고리즘 호출하면 교착 상태 회피 가능하지만 연산 시간 부담. 좀 더 경제적인 방법은 호출 빈도를 줄이는 것, 1시간마다 또는 CPU 이용률이 40%로 떨어질 때마다 호출하는 것

3. 교착 상태 회복

교착 상태 회복 방법

프로세스 중단

- 교착 상태 프로세스 모두 중단 : 교착 상태의 순환 대기를 확실히 해결하지만 자원 사용과 시간면에서 비용 많이 듦. 오래 연산했을 가능성이 있는 프로세스의 부분 결과 폐기하여 다시 연산해야 함
- 한 프로세스씩 중단 : 한 프로세스 중단할 때마다 교착 상태 탐지 알고리즘 호출하여 프로세스가 교착 상태에 있는지 확인. 교착 상태 탐지 알고리즘을 호출하는 부담이 상당히 크다는 것이 단점
- 프로세스 중단이 쉽지 않을 수도 있음. 프로세스가 파일 업데이트하다가 중단된다면 해당 파일은 부정확한 상태. 마찬가지로 프로세스가 데이터를 프린터에 인쇄하고 있을 때 중단하면 다음 인쇄를 진행하기 전에 프린터의 상태를 정상 상태로 되돌려야 함
- 최소 비용으로 프로세스들을 중단하는 우선 순위 선정
  - 프로세스가 수행된 시간과 앞으로 종료하는 데 필요한 시간
  - 프로세스가 사용한 자원 형태와 수(예 : 자원을 선점할 수 있는지 여부)
  - 프로세스를 종료하는 데 필요한 자원 수
  - 프로세스를 종료하는 데 필요한 프로세스 수
  - 프로세스가 대화식인지, 일괄식인지 여부

3. 교착 상태 회복

- 자원 선점
  - 선점 자원 선택 : 프로세스를 종료할 때 비용을 최소화하려면 적절한 선점 순서 결정. 비용 요인에는 교착 상태 프로세스가 점유한 자원 수, 교착 상태 프로세스가 지금까지 실행하는데 소요한 시간 등 매개변수가 포함
  - 복귀 : 필요한 자원을 잃은 프로세스는 정상적으로 실행 불가. 따라서 프로세스를 안정 상태로 복귀시키고 다시 시작해야 함. 일반적으로 안전 상태 결정 곤란, 완전히 복귀시키고(프로세스 중단) 재시작하는 것이 가장 단순한 방법. 프로세스를 교착 상태에서 벗어날 정도로만 복귀시키는 것이 더 효과적. 그러나 이 방법은 시스템이 실 행하는 모든 프로세스의 상태 정보를 유지해야 하는 부담
  - 기아 : 동일한 프로세스가 자원들을 항상 선점하지 않도록 보장하려고 할 때 비용이 기반인 시스템에서는 동일한 프로세스를 희생자로 선택. 그러면 이 프로세스는 작업 완료하지 못하는 기아 상태가 되어 시스템 조치 요청. 따라서 프로세스가 짧은 시간 동안만 희생자로 지정되도록 보장해야 함. 가장 일반적인 해결 방법은 비용 요소에 복귀 횟수를 포함시키는 것

Section 03 기아 상태

■ 기아 상태의 개념

- 작업이 결코 사용할 수 없는 자원을 계속 기다리는 결과(교착 상태)를 예방하려고 자원을 할당할 때 발생(기다림)하는 결과

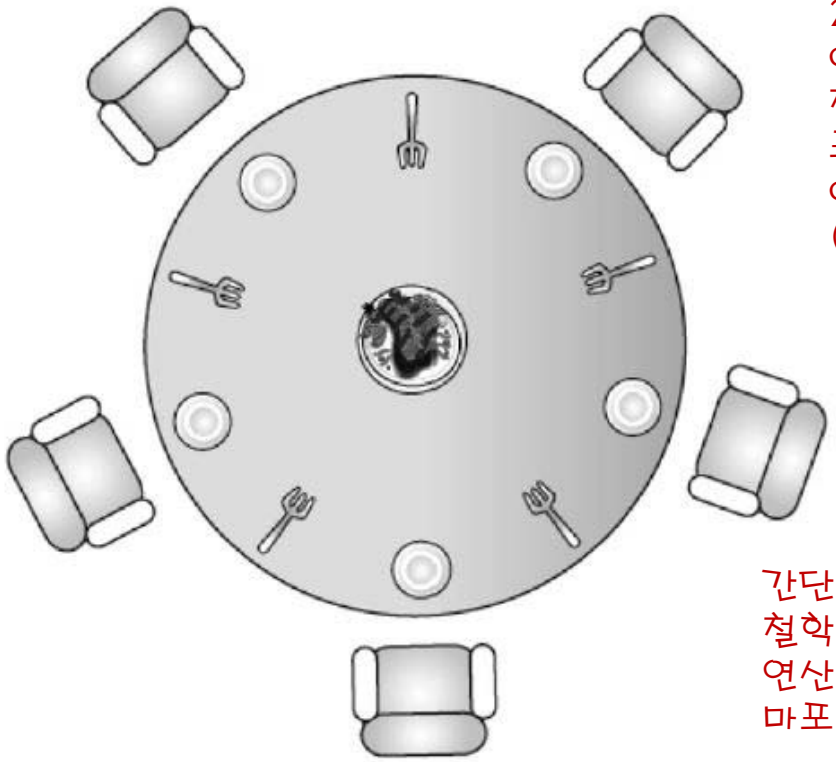


그림 5-20 식사하는 철학자 예

철학자들은 포크 2개로 식사
철학자 5명 동시에 식사 불가(포크가 10개 필요)
2명만이 동시에 식사 가능.
어떤 철학자가 생각 중일 때 다른 철학자 간섭 않음
철학자는 한 번에 포크 한 개만 들 수 있으며, 왼쪽 포크를 먼저 집은 후 오른쪽 포크 집음(공유 자원).
이웃 철학자가 이미 들고 있는 포크는 집을 수 없음
(경쟁 상태)

간단한 해결책 : 각 포크를 세마포로 표시
철학자는 포크에 해당하는 세마포에 대한 P(wait) 연산을 수행하고 나서 포크 집음. 즉, 포크는 해당 세마포에 대한 V(signal) 연산 수행하여 내려놓음

Section 03 기아 상태

식사하는 철학자 문제에서 교착 상태가 발생하는 조건

❶ 철학자들은 서로 포크를 공유할 수 없음

→ 자원을 공유하지 못하면 교착 상태가 발생

❷ 각 철학자는 다른 철학자의 포크를 빼앗을 수 없음

→ 자원을 빼앗을 수 없으면 자원을 놓을 때까지 기다려야 하므로 교착 상태가 발생

❸ 각 철학자는 왼쪽 포크를 잡은 채 오른쪽 포크를 기다림

→ 자원 하나를 잡은 상태에서 다른 자원을 기다리면 교착 상태가 발생

❹ 자원 할당 그래프가 원형

→ 자원을 요구하는 방향이 원을 이루면 양보를 하지 않기 때문에 교착 상태가 발생

Section 03 기아 상태

■ 식사하는 철학자 문제에서 철학자 i의 구조

예제 5-1 식사하는 철학자 문제에서 철학자 i의 구조

```
while (true) {
        wait(fork[i]);                  // 왼쪽 포크를 집음
        wait(fork[(i + 1) % 5);         // 오른쪽 포크를 집음
        ...
                식사한다.
        ...
        signal(fork[i]);                // 왼쪽 포크를 내려놓음
        signal(fork[(i + 1) % 5]);      // 오른쪽 포크를 내려놓음
        ...
                생각한다.
        ...
}
```

- 교착 상태 발생 해결책
  - 철학자 4명만 테이블에 동시에 앉도록 함
  - 철학자가 양쪽 포크 모두 사용할 수 있을 때 포크를 집을 수 있도록 허용(이것은 임계 영역 안에서 해야 함)
  - 비대칭 해결법 사용. 즉, 홀수 번째 철학자는 왼쪽 포크 집은 후 오른쪽 포크 집게 하고, 짝수 번째 철학자는 오른쪽 포크 집은 후 왼쪽 포크 집게 함

Section 03 기아 상태

식사하는 철학자 문제

- 왼쪽에 있는 포크를 잡은 뒤 오른쪽에 있는 포크를 잡아야만 식사 가능

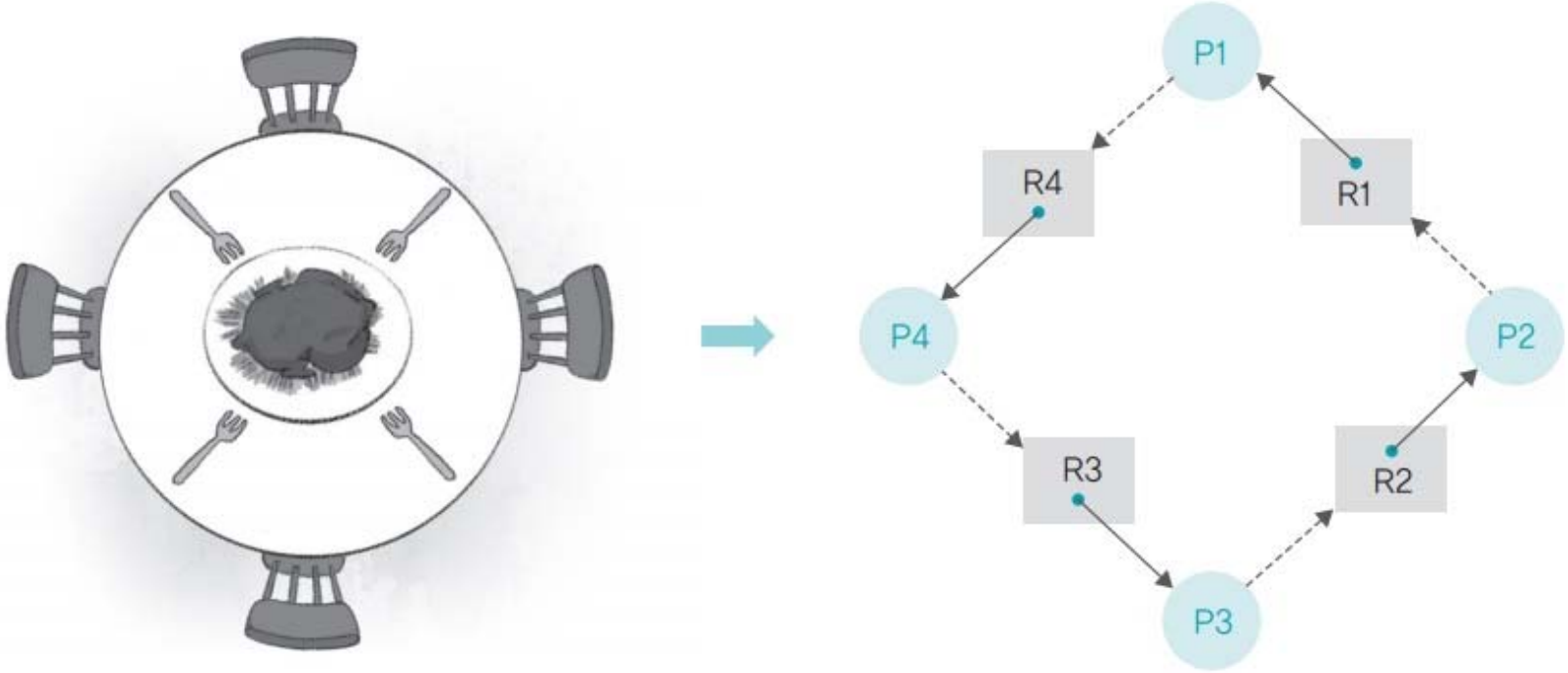


그림 6-8 식사하는 철학자 문제의 자원 할당 그래프

Section 03 기아 상태

■ 식사하는 철학자 문제의 해결 방안

예제 5-2 식사하는 철학자 문제의 해결 방안

```
semaphore fork[5] = {1};
semaphore room = {4};
int i;
while (true) {
        wait(room);
        wait(fork[i]);
        wait(fork[(i + 1) % 5]);
        ...
                식사한다.
        ...
        signal(fork[(i + 1) % 5]);
        signal(fork[i]);
        signal(room);
        ...
                생각한다.
        ...
}
```

철학자 중 한 명이라도 굶어 죽는 일이 없어야 한다는 조건 만족해야 함.
교착 상태의 해결책은 기아 상태starvation의 가능성 제거 못함
기아 상태 해결은 먼저 기다리는 작업 발견하고 각 작업이 기다린 시간 조사 · 추적 해야 함
시스템은 기아 상태 발견하면 즉시 새로운 작업의 시작을 대기 하도록 조치해야 함
빈번한 시스템 대기로 처리량 감소할 수 있으므로 매우 신중한 접근 필요.