

# 운영체제

## Lecture 07: Deadlock

---



전남대학교 인공지능학부  
박태준 ([taejune.park@jnu.ac.kr](mailto:taejune.park@jnu.ac.kr))

# 지난 시간 복습

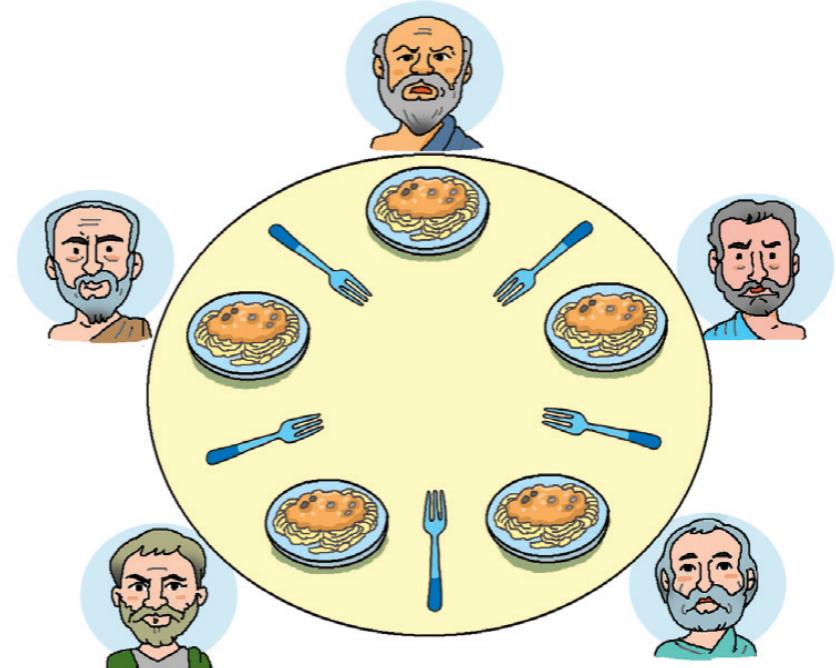
---

- ▶ 프로세스간 통신
  - 프로세스들 간에는 기본적으로 통신이 안됨! (가상메모리) → 별도의 수단이 필요
  - Pipe, Named Pipe, Message queue, Shared memory, Memory map, Socket ...
- ▶ 동기화 기법들
  - 공유 자원의 문제: 하나의 자원을 여럿이 쓴다!
    - 데이터를 읽고, 처리하고, 쓰고... → 이 순서 때문에 데이터가 오염된다!
  - Mutual exclusion and critical section
- ▶ 생산자-소비자 문제
  - 비어있는데 읽으면 안되고, 차있는데 쓰면 안된다. → 생산과 소비의 동기화
  - 두개의 세마포어를 이용해서 해결

# Goal

---

- ▶ 식사하는 철학자 문제
  - 서로 포크 기다리기만 하면? 아무도 밥을 못먹는다!
- ▶ 교착상태 Deadlock
  - 서로 자원을 요구하기만 하고 처리하지를 못하는 상태
  - 어떠한 상황에서 발생하는가?
- ▶ 교착상태의 해결방법
  - 예방, 회피, 감지 및 복구, 무시



# 교착상태

---

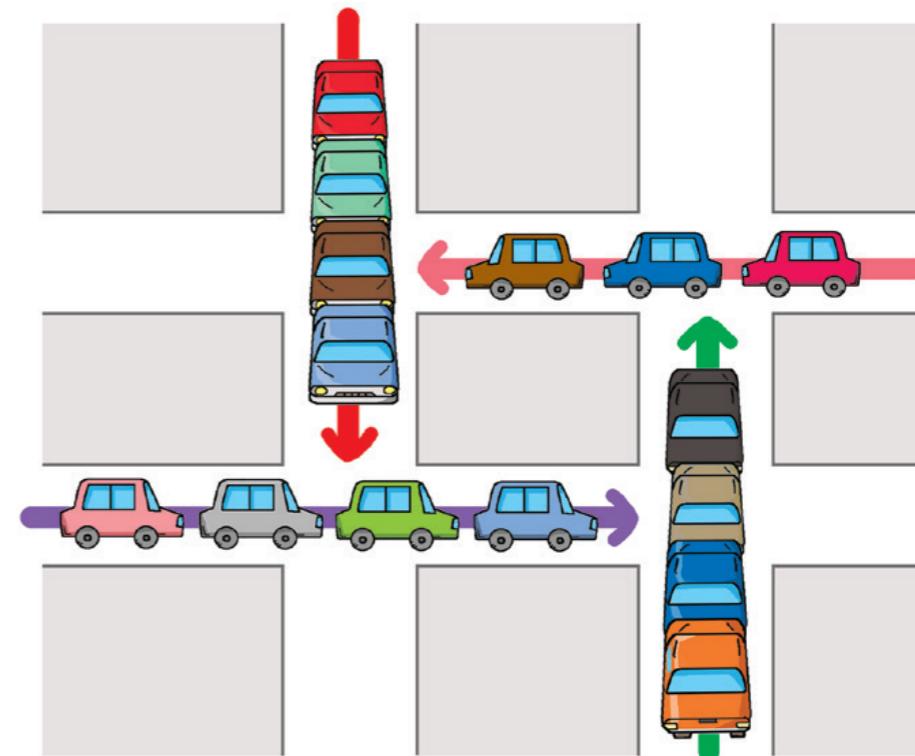
# 교착상태의 정의

- ▶ 자원을 소유한 채, 모두 상대방이 소유한 자원을 기다리면서 무한 대기



한 사람이 밥을 먹기 위해서는 숟가락,  
젓가락이 모두 필요한 상황.

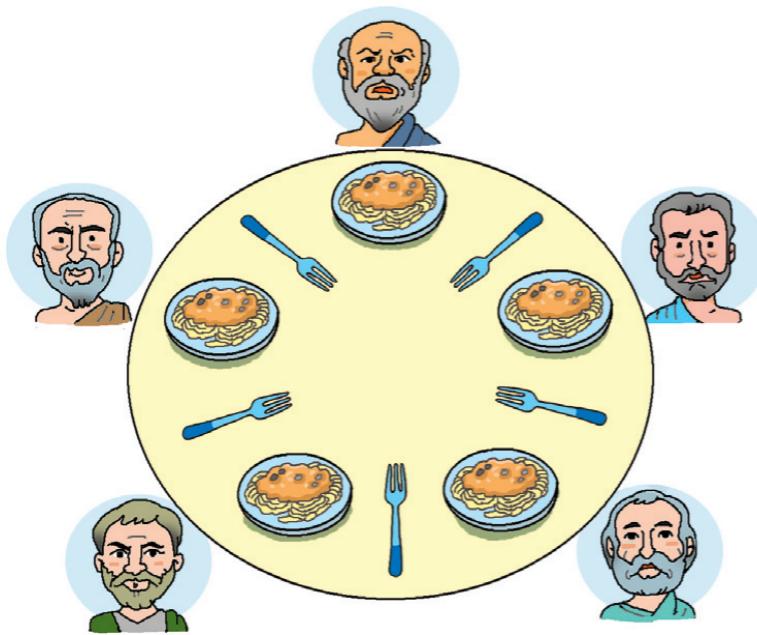
A는 숟가락을 들고, B는 젓가락을 들고,  
A는 젓가락을 사용할 수 있을 때까지 대기,  
B는 숟가락을 사용할 수 있을 때까지 대기,  
무한 대기 발생



자동차들이 한 길을 점유하고  
다른 길을 막고 있는 경우

# 식사하는 철학자 문제 (Dining Philosophers Problem)

- ▶ 교착상태의 문제화 by Edsger Dijkstra 1965.

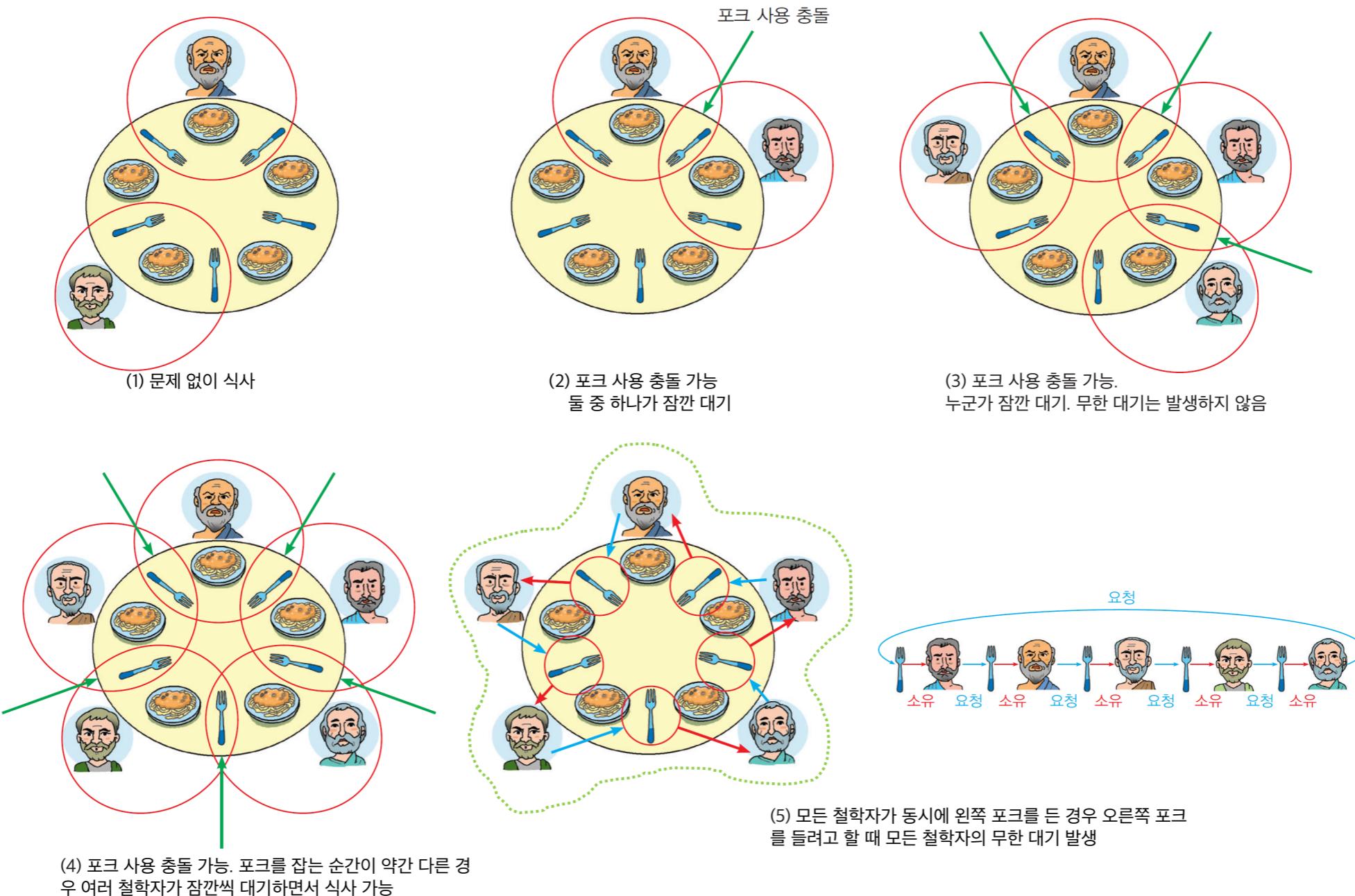


## 식사 규칙(?)

- 5명의 철학자가 원탁에서 식사. 식사 시간은 서로 다를 수 있음
- 자리마다 스파게티 1개와 양 옆에 포크 있음
- 각 철학자는 옆의 철학자에게 말할 수 없음
- 식사를 하기 위해서는 양 옆의 포크가 동시에 들려 있어야 함
- 왼쪽 포크를 먼저 들고, 다음 오른쪽 포크를 드는 순서
- 포크가 사용 중이면 대기

모두가 이상 없이 식사를 잘 할 수 있을까?

# 경우의 수



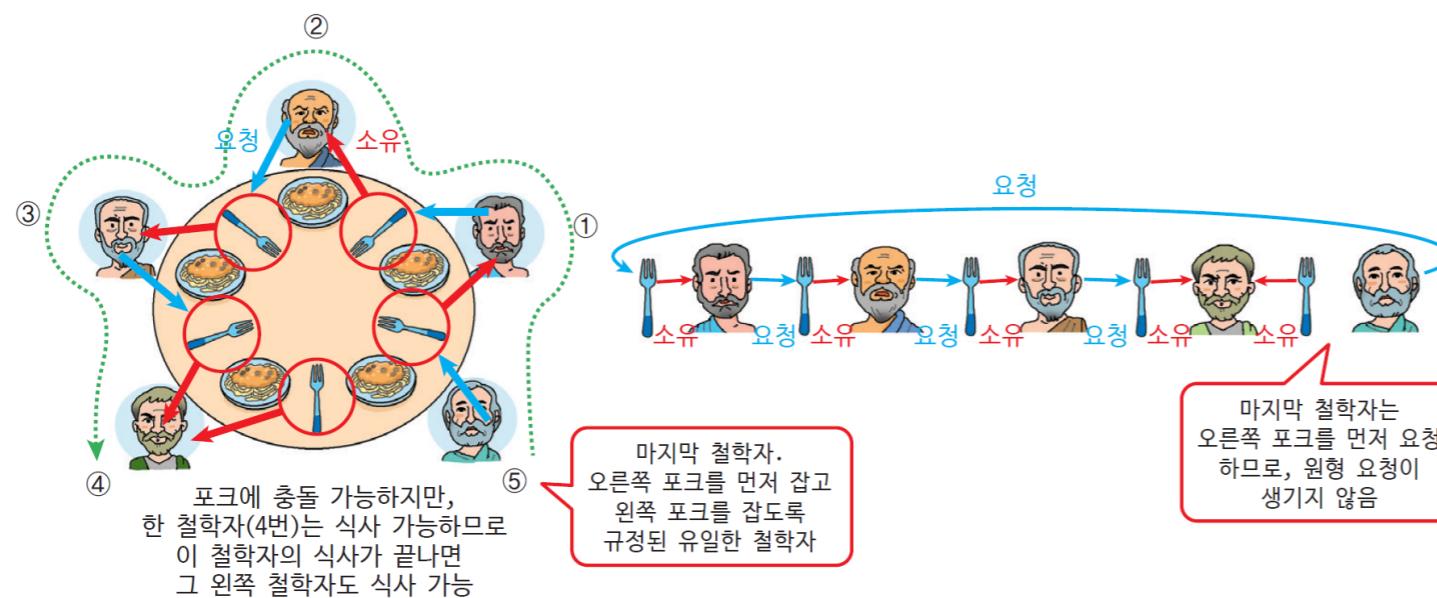
# 왜 식사를 못하는 걸까?

## ▶ 원인: 환형 요청/대기(circular wait)

- 모두 거의 동시에 왼쪽 포크를 든 후 오른쪽 포크를 들려고 할 때, 모두 상대가 가진 포크를 기다리면서 먹을 수 없는 **교착상태** 발생
- 5명 모두 왼쪽 포크를 가지고 오른쪽 포크를 요청하는 환형 고리
- 환형 고리는 스스로 인식이나 해체 불가

## ▶ 해결: 원형 상태로 요청이 생기지 않도록 해야함.

- 5명 중 마지막 사람은 제외한 4명은 왼쪽의 포크를 잡고 오른쪽 포크를 잡는 순서로 진행, 마지막 사람은 오른쪽 포크를 잡고 왼쪽 포크를 잡음



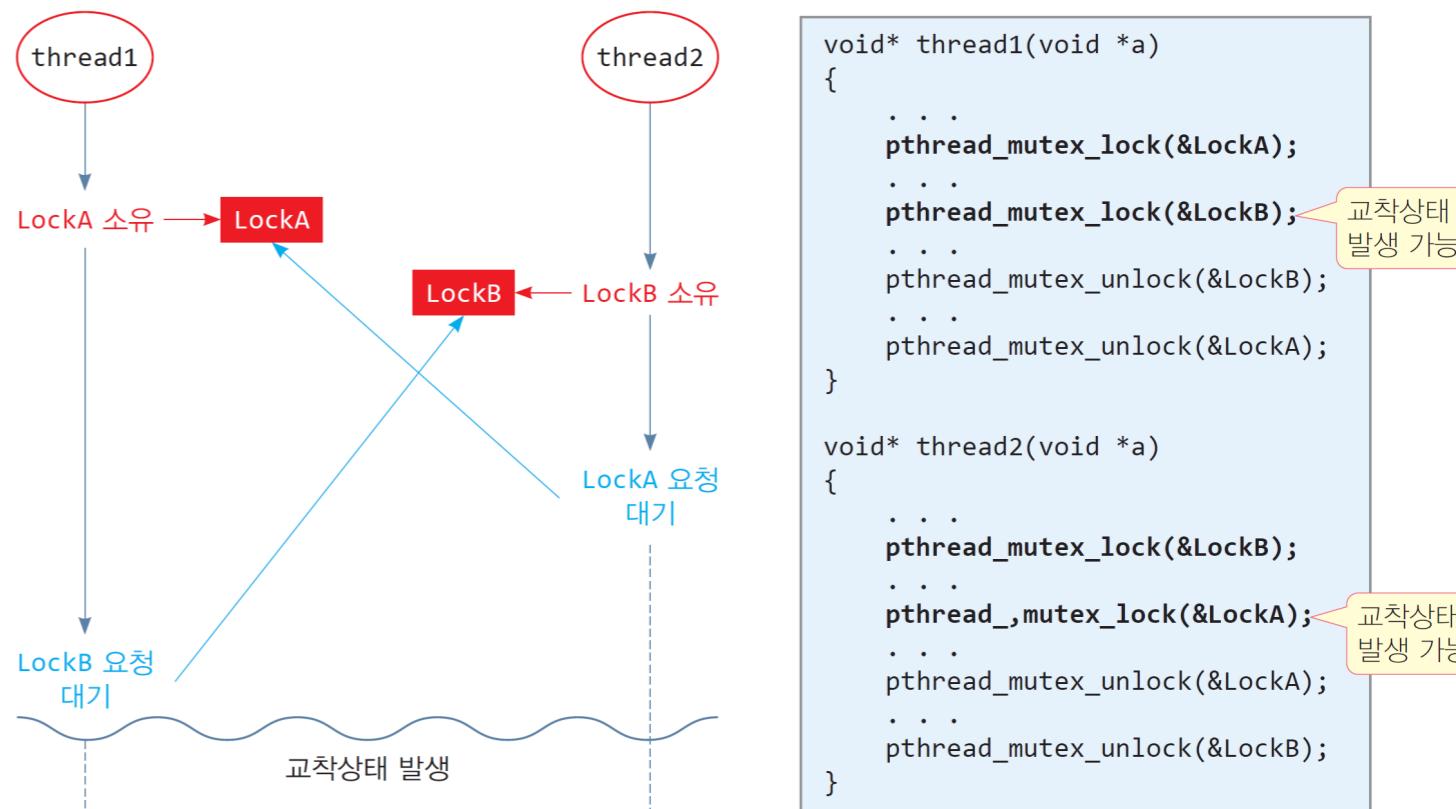
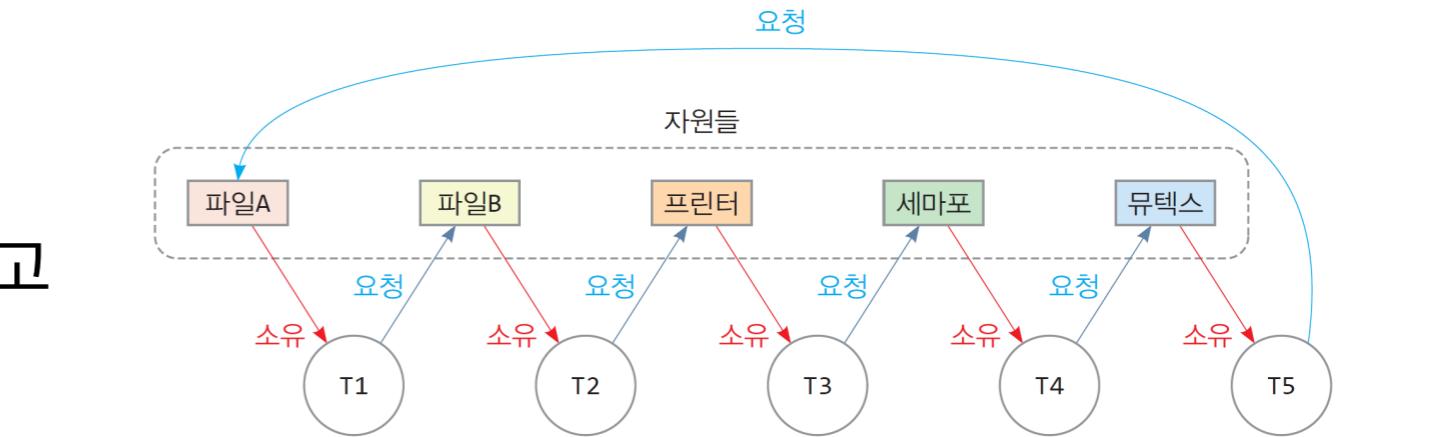
← 이것을 알고리즘으로!

# 컴퓨터 시스템에서의 교착상태

- ▶ 자원을 소유한 쓰레드(프로세스)들 사이에서, 각 쓰레드는 다른 쓰레드가 소유한 자원을 요청하여 무한정 대기하고 있는 현상  
→ 다중프로그래밍 시스템 초기에

## 노출된 문제점!

- 철학자: 프로세스
- 포크: 자원
- 스파게티: 프로세스가 처리할 작업



## 교착상태 확인해보기

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

int x = 0; // shared
int y = 0; // shared
pthread_mutex_t lock1; // mutex lock
pthread_mutex_t lock2; // mutex lock

void* thread1(void* arg) { // thread1
    pthread_mutex_lock(&lock1); // lock1 for x
    printf("lock thread1 lock1\n");
    x++;
    sleep(2); // sleep

    pthread_mutex_lock(&lock2); // lock2 for y
    printf("lock thread1 lock2\n");
    y++;
    pthread_mutex_unlock(&lock2); // unlock lock2
    printf("unlock thread1 lock2\n");

    pthread_mutex_unlock(&lock1); // unlock lock1
    printf("unlock thread1 lock1\n");
}

void* thread2(void* arg) { // thread2
    pthread_mutex_lock(&lock2); // lock2 for y
    printf("lock thread2 lock2\n");
    y++;
    sleep(2); // sleep

    pthread_mutex_lock(&lock1); // lock1 for x
    printf("lock thread2 lock1\n");
    x++;
    pthread_mutex_unlock(&lock1); // unlock lock1
    printf("unlock thread2 lock1\n");

    pthread_mutex_unlock(&lock2); // unlock lock2
    printf("unlock thread2 lock2\n");
}
```

```
int main() {
    pthread_t tid[2];

    pthread_mutex_init(&lock1, NULL);
    pthread_mutex_init(&lock2, NULL);

    pthread_create(&tid[0], NULL, thread1, NULL);
    pthread_create(&tid[1], NULL, thread2, NULL);

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);

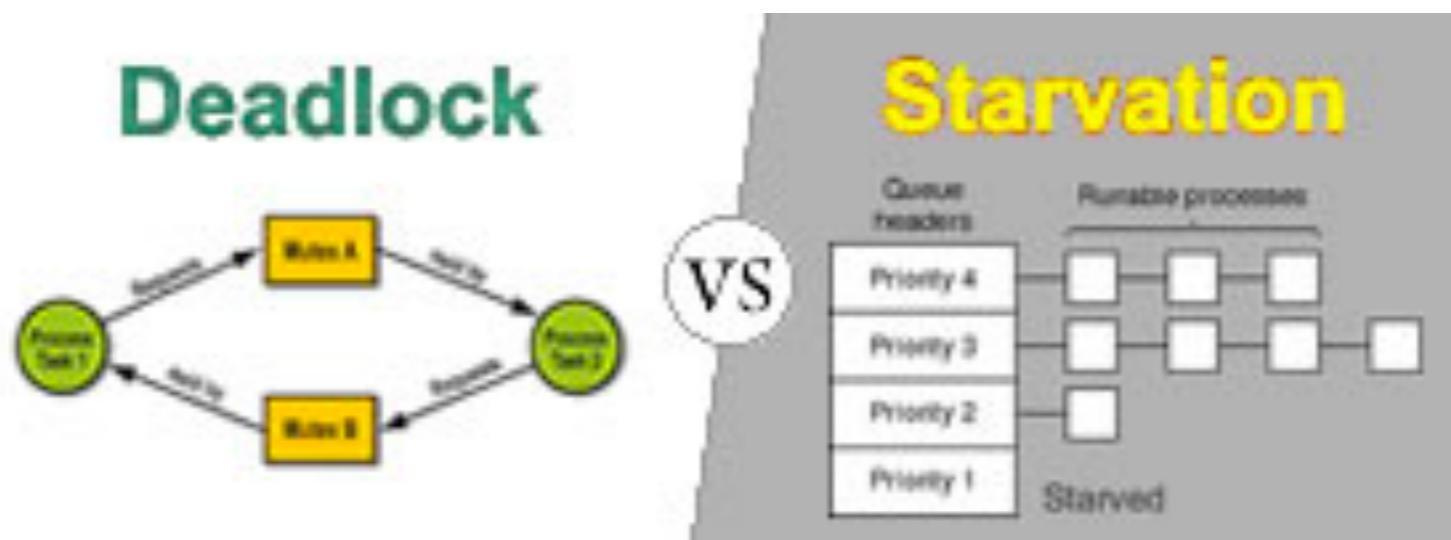
    pthread_mutex_destroy(&lock2);
    pthread_mutex_destroy(&lock1);

    printf("x = %d, y = %d\n", x, y);

    return 0;
}
```

# 기아와 교착의 차이

- ▶ 비슷해 보이지만 다릅니다...!
- ▶ **Starvation**
  - 운영체제가 잘못된 정책을 사용하여 특정 프로세스의 작업이 지연되는 문제
- ▶ **Deadlock**
  - 교착 상태는 여러 프로세스가 작업을 진행하다 보니 자연적으로 일어나는 문제이다.



# 교착상태의 잠재적 요인들

---

- ▶ 1. 자원
  - 교착상태의 발생지
  - 교착상태는 멀티쓰레드가 자원을 동시에 사용하려는 충돌이 요인
  - 컴퓨터 시스템에는 많은 자원 존재
    - 소프트웨어 자원 – 뮤텍스, 스피너, 세마포, 파일, 데이터베이스
    - 하드웨어 자원 – 프린터, 메모리, 프로세서 등
- ▶ 2. 자원과 쓰레드
  - 한 쓰레드가 여러 자원을 동시에 필요로 하는 상황이 요인
- ▶ 3. 자원과 운영체제
  - 한 번에 하나씩 자원을 할당하는 운영체제 정책이 요인
  - 만일 쓰레드가 필요한 자원을 한 번에 모두 요청하도록 한다면? → 교착상태가 발생하지 않게 할 수 있다.
- ▶ 4. 자원 비선점
  - 할당된 자원은 쓰레드가 자발적으로 내놓기 전에 강제로 뺏지 못하는 정책이 요인
  - 운영체제는 쓰레드가 가진 자원을 강제로 뺏지 못함
  - 만일 강제로 빼앗을 수 있다면? → 교착상태가 발생하지 않게 할 수 있음!

# 의외로... 자주 발생합니다!

- ▶ 특히 공유변수와 관련하여  
코딩을 잘못한다면.....!
- ▶ 커널에서도 정말 가끔 발생함.
  - 다만 코딩을 엄청 신경써서 잘 하기 때문에,  
우려할 정도는 아님.

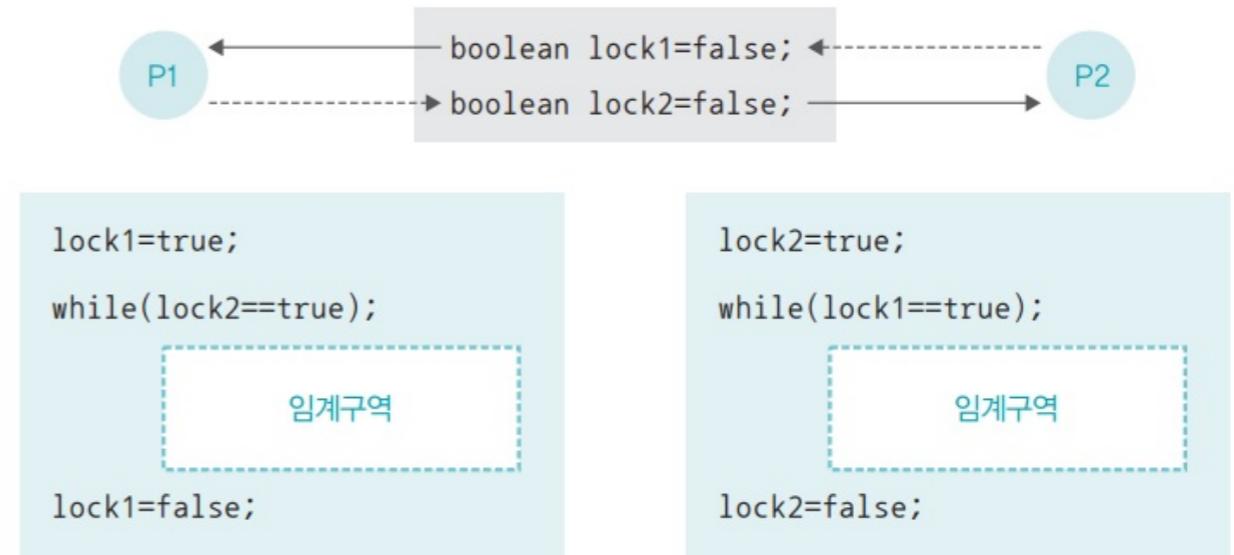


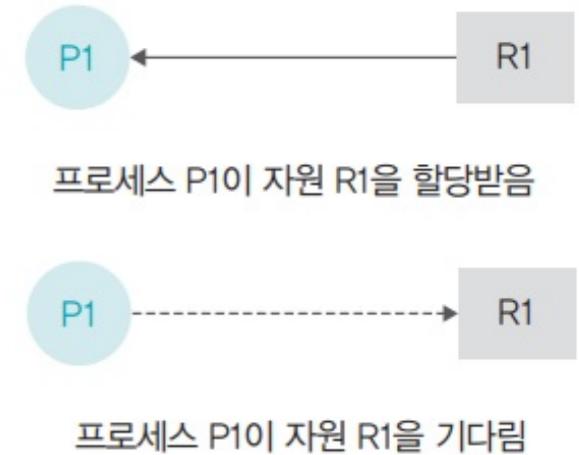
그림 6-4 교착 상태를 발생시키는 임계구역 코드

- ▶ 그렇지만 교착상태를 직접적으로 막는 시스템은 거의 없다!
  - 막는데 많은 시간과 공간의 비용이 들기 때문.
  - 교착상태가 발생하도록 두고, 교착상태가 발생한 것 같으면,  
시스템 재시작, 혹은 의심스러운 몇몇 프로그램 종료

# 교착상태 모델링: 자원 할당 그래프(Resource Allocation Graph, RAG)

## ▶ 그래프의 요소

- Vertex – 프로세스/쓰레드(원), 자원(사각형)
- Edge – 소유/요청 관계. 할당 간선과 요청 간선
  - 할당 간선 : 자원에서 쓰레드로 향하는 화살표. 할당 받은 상태 표시
  - 요청 간선 : 쓰레드에서 자원으로 향하는 화살표. 요청 표시



## ▶ 자원에 대한 시스템의 상태를 나타내는 방향성 그래프

- 컴퓨터 시스템에 실행 중인 전체 쓰레드와 자원의 개수
- 각 자원의 총 인스턴스 개수와 할당 가능한 인스턴스 개수
- 각 쓰레드가 할당받아 소유하고 있는 자원의 인스턴스 개수
- 각 쓰레드가 실행에 필요한 자원 유형과 인스턴스 개수

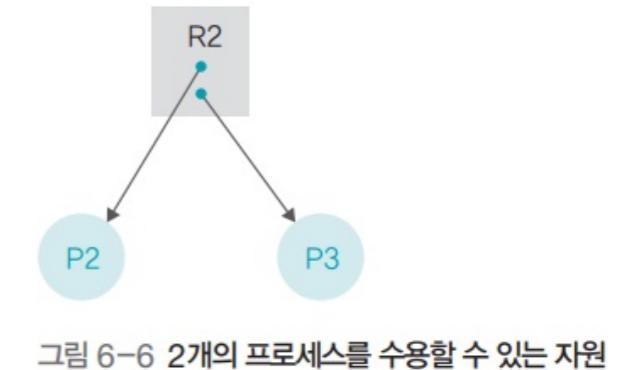
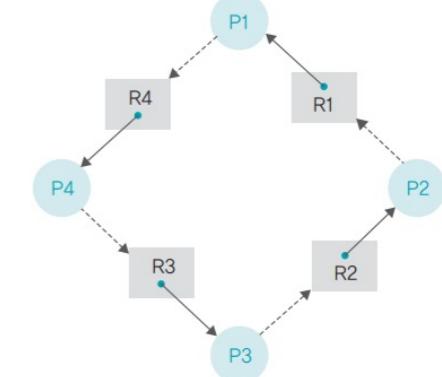


그림 6-6 2개의 프로세스를 수용할 수 있는 자원

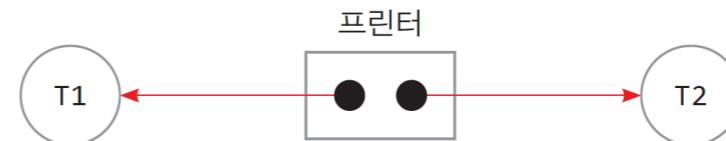
## ▶ 자원할당그래프를 통해 교착상태 판단



# 자원 할당 그래프



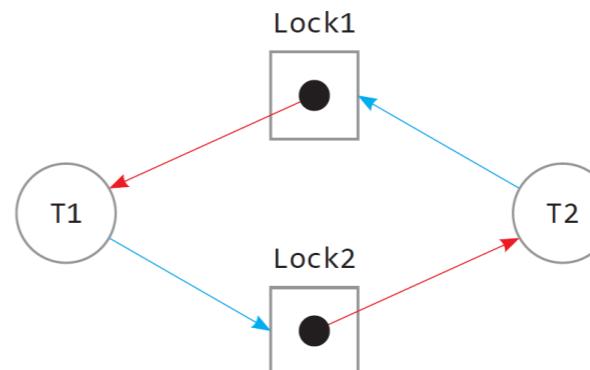
(a) 시스템에는 2개의 스레드 T1과 T2 그리고 프린터 1개 있음. T1은 프린터 소유. T2는 프린터 요청 대기



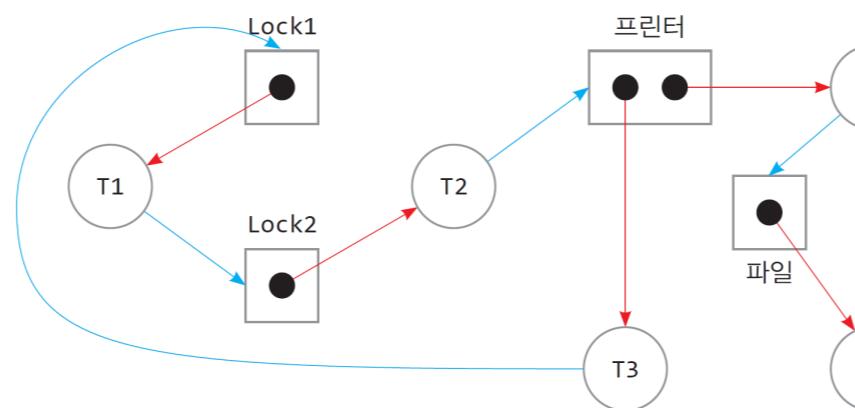
(b) 시스템에는 2개의 스레드 T1과 T2 그리고 2개의 프린터 있음. T1과 T2 각각 프린터 1개씩 소유



(c) 시스템에는 T1 스레드와 2개의 프린터 있음. T1이 프린터 1개 소유



(d) 시스템에는 2개의 스레드 T1과 T2 그리고 Lock1과 Lock2의 두 자원이 있음. T1은 Lock1을 소유하고 Lock2를 요청. T2는 Lock2를 소유하고 Lock1을 요청



T1, T2, T3의 교착 상태

# 교착상태 해결

---

# 교착상태의 발생 필요충분조건

- ▶ 다음 4가지 상황이 허용되는 시스템은 언제든 교착상태 발생 가능 (Coffman condition)

- **상호 배제 (Mutual Exclusion)**

- 각 자원은 한 번에 하나의 프로세스/쓰레드에게만 할당
    - 자원이 한 쓰레드에게 할당되면 다른 쓰레드에게는 할당될 수 없음

자원적  
특성

- **비선점 (No Preemption)**

- 프로세스/쓰레드에게 할당된 자원을 강제로 빼앗지 못함

- **점유와 대기 (Hold & Wait)**

- 쓰레드가 한 자원을 소유(lock)하면서 다른 자원을 기다리기

행위적  
특성

- **환형 대기(Circular Wait)**

- 한 그룹의 쓰레드들에 대해, 각 쓰레드는 다른 쓰레드가 요청하는 자원을 소유하는 원형 고리 형성

- ▶ 4가지 조건 중 한 가지라도 성립되지 않으면, 교착상태 발생 않음

- 교착상태 해결은 저 넷중 하나를 깨트리는 것.

# 식사하는 철학자 문제 분석

---

## ▶ 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 없음.
- (사각형 식탁에서 한 줄로 앉아서 식사를 한다면 교착 상태가 발생하지 않음!)

# 교착상태 해결

---

## ▶ 1. 교착상태 예방(prevention)

- 교착상태 발생 여지를 차단하여 예방
- 교착상태에 빠지는 4가지 조건 중 하나 이상의 조건이 성립되지 못하도록 시스템 구성

## ▶ 2. 교착상태 회피(avoidance)

- 미래에 교착상태로 가지 않도록 회피
- 자원 할당 시마다 미래의 교착 상태 가능성을 검사하여 교착 상태가 발생하지 않을 것이라고 확신하는 경우에만 자원 할당
- 안전한 상태와 불안전한 상태로 시스템 상태 분류, 안전한 상태인 경우에만 자원 할당
- 자원 할당 시마다 교착 상태 가능성을 검사하므로 시스템 성능 저하

## ▶ 3. 교착상태 감지 및 복구(detection and recovery)

- 교착상태를 감지하는 프로그램 구동, 발견 후 교착상태 해제
- 백그라운드에서 교착 상태를 감지하는 프로세스가 늘 실행되어야 하는 부담

## ▶ 4. 교착상태 무시

- 아무런 대비책 없음, 교착상태는 없다고 단정
- 사용자가 이상을 느끼면 재실행할 것이라고 믿는 방법
- 리눅스, 윈도우 등 현재 대부분의 운영체제에서 사용하는 가장 일반적인 방법
- 교착상태 예방, 회피, 감지, 복구 등에는 많은 시간과 공간이 필요하며 시스템의 성능을 떨어뜨리기 때문
- 심각하지 않은 작업들에 대해서는 교착상태 무시

# 교착상태 예방 (1)

---

- ▶ 코프만의 4가지 조건 중 최소 하나를 성립하지 못하게 함
- ▶ 1. 상호 배제(Mutual Exclusion) 조건 → 상호 배제 없애기
  - 동시에 2개 이상의 쓰레드가 자원을 활용할 수 있도록 함
  - 컴퓨터 시스템에서 근본적으로 적용 불가능한 방법
- ▶ 2. 비선점(No Pre-emption) 조건 → 선점 허용
  - 모든 자원을 빼앗을 수 있도록 만드는 방법
  - 자원을 강제로 반환하게 된 쓰레드가 자원을 다시 사용하게 될 때 이전 상태로 되돌아갈 수 있도록 상태를 관리할 필요
  - 기아현상이 발생할 수도 있음.
  - 간단치 않고 오버헤드 매우 큼
- ▶ 사실 자원적 특성을 제한하기는 어려움...!

# 교착상태 예방 (2)

## ▶ 3. 점유와 대기(Hold & Wait) 조건 → 기다리지 않게

- 방법 1

- 운영체제는 쓰레드 실행 전 필요한 모든 자원을 파악하고 실행시 한 번에 할당
- 다른 쓰레드는 필요한 자원을 할당 받지 못하고 실행 대기

- 방법 2

- 쓰레드가 새로운 자원을 요청하려면, 현재 할당 받은 모든 자원을 반환하고, 한꺼번에 요청하여 할당

- 문제점

- 프로세스가 자신이 사용하는 모든 자원을 자세히 알기 어려움
- 당장 사용하지 않는 자원을 쓰레드에게 끌어 두기 때문에 자원 활용률이 떨어짐
- 많은 자원을 사용하는 프로세스가 적은 자원을 사용하는 프로세스보다 불리함
- → 결국 Batch-processing화 됨.

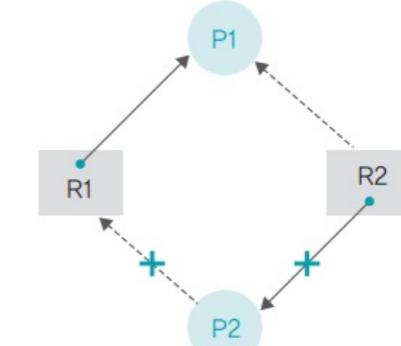


그림 6-11 전부 할당하거나 아니면 아예 할당하지 않는 방식

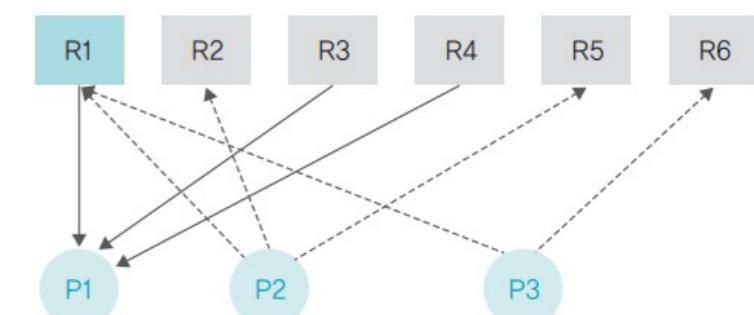


그림 6-12 필요한 자원을 모두 할당한 후 실행

## 교착상태 예방 (3)

### ▶ 4. 환형 대기(Circular Wait) 조건 → 환형 대기 제거

- 모든 자원에 숫자를 부여하고 숫자가 큰 방향으로만 자원을 할당하는 것
  - e.g., 마우스를 할당받은 상태에서 프린터를 할당받을 수는 있지만 프린터를 할당받은 상태에서는 마우스나 하드디스크를 할당받을 수 없음
  - e.g., 프로세스 P2는 자원을 할당받을 수 없어 강제 종료되고 프로세스 P1은 정상적으로 실행

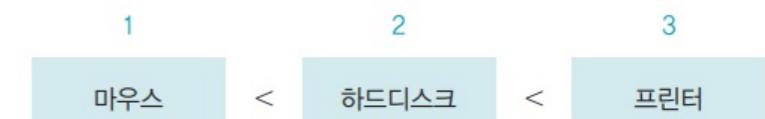


그림 6-13 자원에 대한 번호 매김

- 문제점
  - 프로세스 작업 진행에 유연성이 떨어짐
  - 자원의 번호를 어떻게 부여할 것인지를 문제

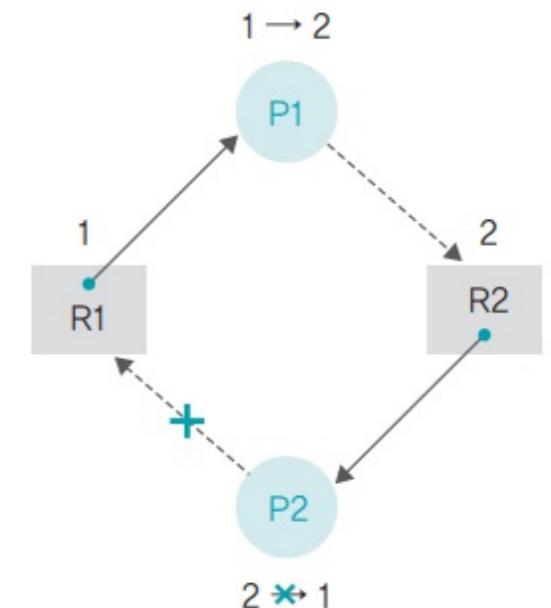


그림 6-14 원형 대기 조건의 부정

▶ 점유와 대기, 원형 대기는 프로세스 작업 방식을 제한하고 자원을 낭비하기 때문에 사용하기 어려움...!

# 교착상태 회피

- ▶ 자원 할당 시, 미래에 환형 대기가 발생할 것으로 판단되면 자원 할당 하지 않는 정책
  - 프로세스에 자원을 할당할 때 어느 수준 이상의 자원을 나누어주면 교착 상태가 발생하는지 파악하여 그 수준 이하로 자원을 나누어주는 방법
- ▶ 교착 상태가 발생하지 않는 범위 내에서만 자원을 할당하고, 교착 상태가 발생하는 범위에 있으면 프로세스를 대기시킴. 즉, 할당되는 자원의 수를 조절하여 교착 상태를 피함
  - 안전한 상태 Safe state
    - 현재 프로세스들을 어떤 순서로 실행 시켰을 때, 모든 프로세스들이 자신이 요청하는 자원을 가지고 실행할 수 있다면 안전한 상태
  - 불안전한 상태 Unsafe state
    - 환형 대기에 빠질 수 있다면 불안전한 상태
  - 할당된 자원이 적으면 안정 상태가 크고,  
할당된 자원이 늘어날수록 불안정 상태가 커짐

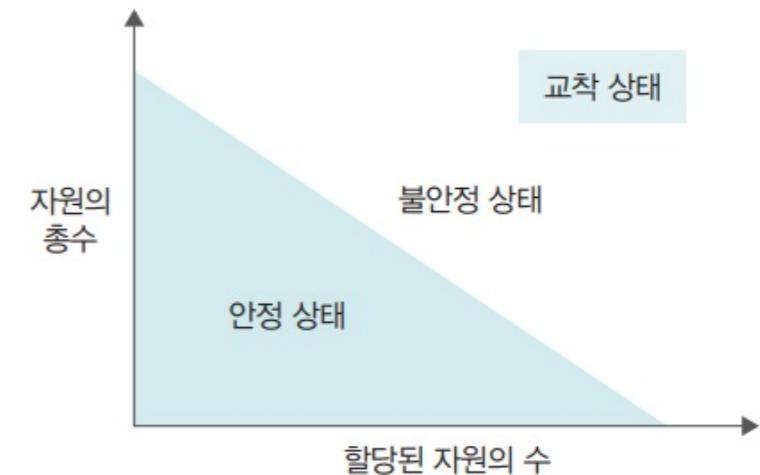


그림 6-15 안정 상태와 불안정 상태

# banker's 알고리즘

- ▶ Edsger Dijkstra 에 의해 개발된 알고리즘.
- ▶ 자원 할당 전에 미래에 교착상태가 발생여부를 판단
  - 은행에서의 대출 알고리즘(돈을 대출한 사람, 돈을 대출하려고 하는 사람)
- ▶ 알고리즘
  - 각 프로세스가 실행 시작 전에 필요한 전체 자원의 수를 운영체제에게 알림
  - 자원을 할당할 때마다, 자원을 할당해주었을 때 교착상태가 발생하지 않을 만큼 안전한 상태인지 판단하여 안전한 상태일 때만 자원 할당
  - 각 프로세스가 필요한 자원의 개수, 현재 각 프로세스가 할당 받은 자원을 개수, 그리고 시스템 내 할당 가능한 자원의 개수를 토대로 현재 요청된 자원을 할당해도 안전한지 판단
- ▶ 비현실적
  - 각 프로세스가 실행 전에 필요한 자원의 개수를 아는 것은 불가능
  - 프로세스의 개수도 동적으로 변하기 때문에, 미리 프로세스의 개수를 정적으로 고정시키는 것 불가능

Total=14		Available=2	
Process	Max	Allocation	Expect
P1	5	2	3
P2	6	4	2
P3	10	6	4

그림 6-17 은행원 알고리즘(안정 상태)

Total=14		Available=1	
Process	Max	Allocation	Expect
P1	7	3	4
P2	6	4	2
P3	10	6	4

그림 6-18 은행원 알고리즘(불안정 상태)

# 교착상태 감지 및 복구

- ▶ 교착상태를 감지하는 프로그램을 통해, 형성된 교착상태를 품
  - 백그라운드에서 교착상태를 감지하는 프로그램 를 실행
- ▶ 자원 할당 그래프를 이용한 교착 상태 검출
  - 단일 자원을 사용하는 경우 자원 할당 그래프에 사이클 있으면 교착 상태
- ▶ 타임아웃을 이용한 교착 상태 검출
  - 일정 시간 동안 작업이 진행되지 않은 프로세스를 교착 상태가 발생한 것으로 간주하여 처리
  - 교착 상태가 자주 발생하지 않을 것이라는 가정하에 사용 (타조 알고리즘)
  - 타임아웃이 되면 프로세스가 종료

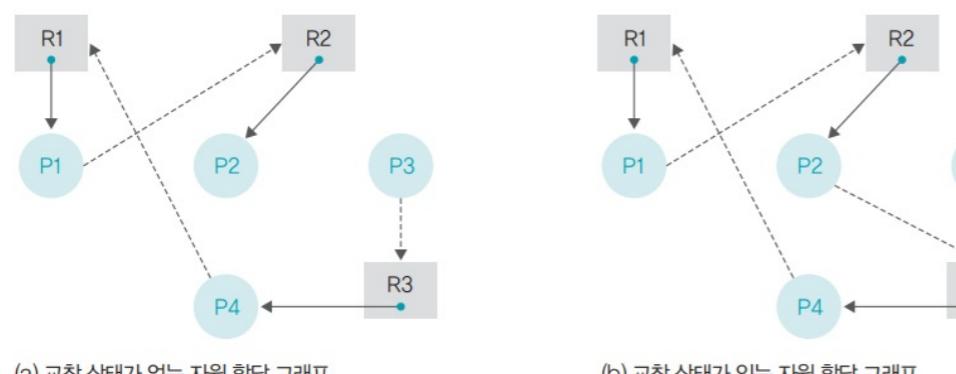
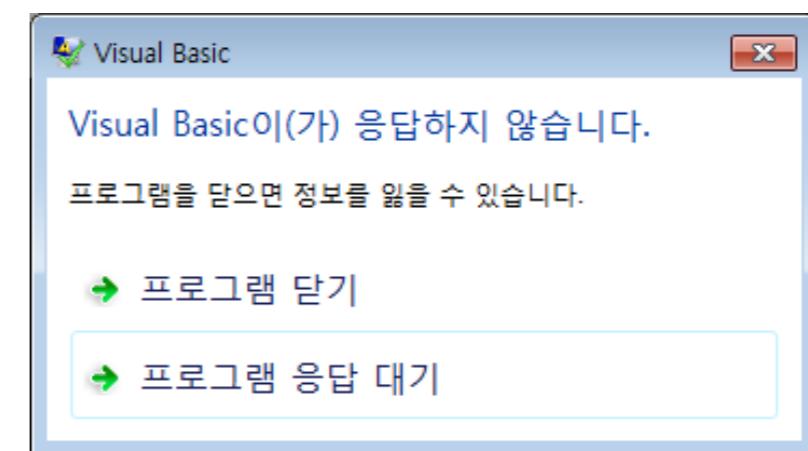


그림 6-22 자원 할당 그래프와 교착 상태



# 교착상태를 감지하였을 때의 복구 방법

---

- ▶ 자원 강제 선점(preemption)
  - 교착상태에 빠진 쓰레드 중 하나의 자원을 강제로 빼앗아 다른 쓰레드에게 할당
- ▶ 롤백(rollback)
  - 운영체제는 주기적으로 교착상태가 발생할 것으로 예측되는 쓰레드의 상태를 저장하여 두고 교착상태가 발생하면 마지막으로 저장된 상태로 돌아가도록 하고, 다시 시작하면서 자원을 다르게 할당
  - 시간과 메모리 공간(rollback의 경우)에 대한 부담이 크기 때문에 잘 사용하지 않음
- ▶ 프로세스/쓰레드 강제 종료(kill process)
  - 교착상태에 빠진 쓰레드 중 하나 강제 종료, 가장 간단하면서도 효과적인 방법
  - 프로세스를 강제로 종료하는 방법
    - 1. 교착 상태를 일으킨 모든 프로세스를 동시에 종료
    - 2. 교착 상태를 일으킨 프로세스 중 하나를 골라 순서대로 종료
      - 우선순위가 낮은 프로세스를 먼저 종료
      - 우선순위가 같은 경우 작업 시간이 짧은 프로세스를 먼저 종료
      - 위의 두 조건이 같은 경우 자원을 많이 사용하는 프로세스를 먼저 종료

# Ostrich algorithm

- ▶ 교착상태를 해결할 필요가 있을까?
  - 얼마나 자주 발생하는 일일까? 1년에 한 번, 혹은 10년에 한번?
- ▶ 교착상태는 반드시 발생!
  - 하지만 발생횟수/확률에 비해 해결에는 상대적으로 비용이 많이 듦
- ▶ 타조 알고리즘: Put your head in sands
  - 타조가 머리를 모래 속에 박고 자신이 보이지 않는 체하는 것
  - 교착상태는 발생하지 않을 거야 하고 아무 대책을 취하는 않는 접근법 (그냥 무시...)
  - Unix와 윈도우 등 현재 거의 모든 운영체제에서 사용
  - 의심 가는 쓰레드를 종료시키거나 시스템 재시작(reboot)



# Ostrich 알고리즘의 문제

- ▶ 교착상태가 발생하면 시스템 재시작 혹은 특정 프로세스/쓰레드 강제 종료  
→ 관련된 데이터를 잃어버릴 수 있음 (하지만 전체적으로 크지 않은 손실)
- ▶ e.g., 데이터베이스
  - 데이터베이스에서 타임아웃으로 프로세스가 종료되면 일부 데이터의 일관성이 깨질 수 있음
  - 데이터의 일관성이 깨지는 문제를 해결하기 위해 체크포인트와 롤백 사용
    - 체크포인트 : 작업을 하다가 문제가 발생하면 저장된 상태로 돌아오기 위한 표시
    - 롤백 : 작업을 하다가 문제가 발생하여 과거의 체크포인트로 되돌아가는 것

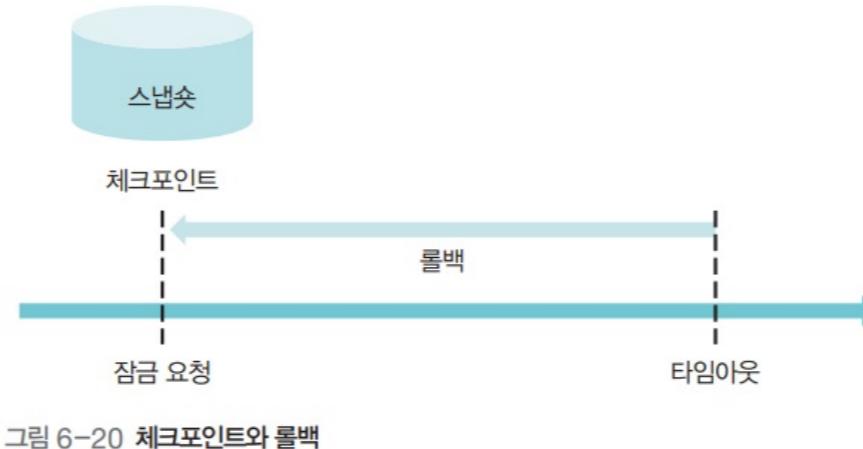


그림 6-20 체크포인트와 롤백

- ▶ 그래서 Mission-critical system에서는 적절한 해결책이 반드시 필요하다!
  - 또는 시스템에서는 설계단계에서 자원에 대한 프로세스의 할당 등에 대해 미리 계획하고 운영

# Summary

---

- ▶ 교착상태
  - 자원을 소유한 채, 모두 상대방이 소유한 자원을 기다리면서 무한 대기하는 상태
  - 기아현상과 다릅니다!
- ▶ 4가지 조건이 만족하게되면 발생함
  - 상호 배제 (Mutual Exclusion)
  - 비선점 (No Preemption)
  - 점유와 대기 (Hold & Wait)
  - 환형 대기(Circular Wait)
- ▶ 교착상태의 회복
  - 예방(prevention), 회피(avoidance), 감지 및 복구(detection and recovery), 무시
  - 그런데 여러 방법들이 비용이 상당히 비쌉니다
  - 그리고 사실 그리 자주 발생하는 문제는 아니라서... 대부분 그냥 종료 후 재시작!