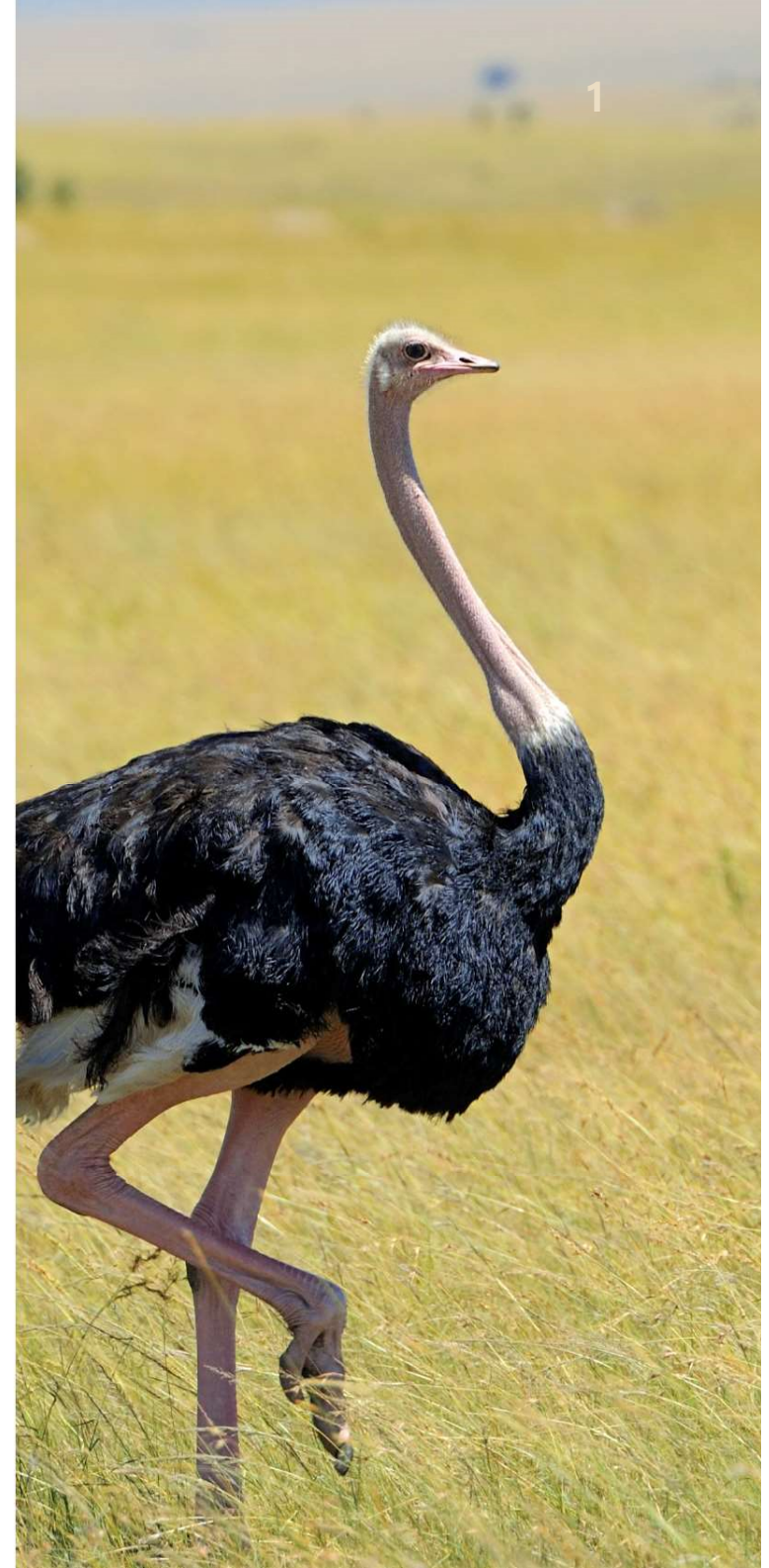


Chapter

# 07

## 교착상태

1. 교착상태 문제 제기
2. 교착상태
3. 교착상태 해결



# 강의 목표

1. 식사하는 철학자 문제를 통해 처음으로 제기된 교착상태 개념을 이해한다.
2. 컴퓨터 시스템에서의 교착상태 정의와 교착상태를 판단할 수 있는 자원 할당 그래프를 이해한다.
3. 교착상태가 유발 가능한 코프만의 4가지 조건을 이해한다.
4. 교착상태의 해결 방법을 이해한다.
  - 교착상태 예방, 회피, 감지 및 복구, 무시
5. 현대에서 교착상태 해결 방법은 교착상태 무시 방법임을 안다.

3

# 1. 교착상태 문제 제기

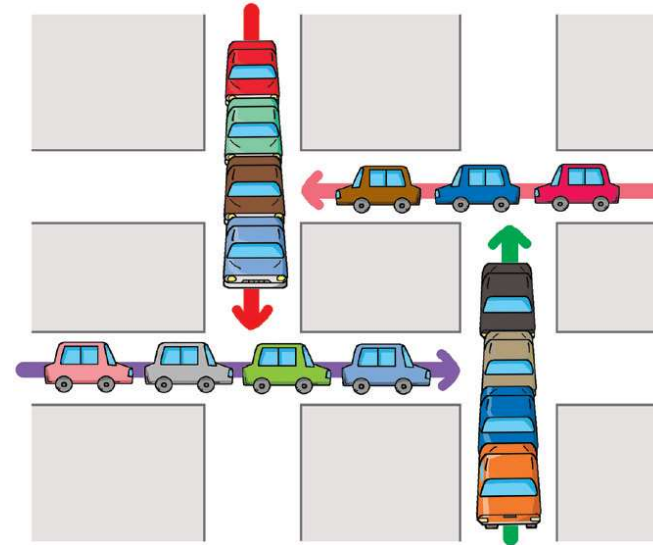
# 실생활에서 발생하는 교착상태 사례

4



한 사람이 밥을 먹기 위해서는 숟가락,  
젓가락이 모두 필요하다는 규칙.

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

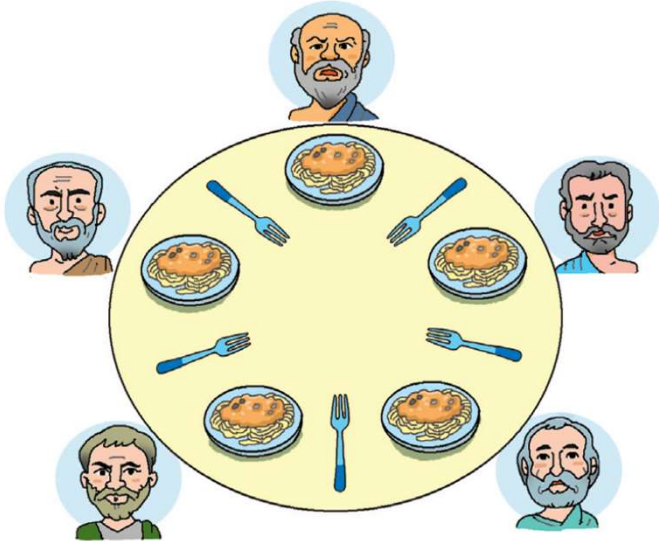


자동차들이 한 길을 점유하고  
다른 길을 기다리고 있는 경우

교착상태 : 자원을 소유한 채, 모두 상대방이 소유한 자원을 기다리면서 무한 대기

# 식사하는 철학자 문제 (Dining Philosophers Problem)

5



1965 년 [Edsger Dijkstra](#)에 의해 처음으로 문제화  
병렬처리(concurrent programming)에서의 동기화 이슈와 해결  
방법을 설명하고자 낸 학생 시험 문제(네델란드의 [Eindhoven University of Technology](#))

- 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 포크를 확보하여야 함
- 왼쪽 포크를 먼저 들고, 다음 오른쪽 포크를 드는 순서
- 포크가 사용 중이면 대기

식사하는 데 어떤 문제가 있을까?

문제가 발생하는 경우 - 모두 거의 동시에 왼쪽 포크를 든 후 오른쪽 포크를 들려고 할 때, 모두 상대가 가진 포크를 기다리면서 먹을 수 없는 교착 상태 발생

**해결 방법 중 하나** – 원형 상태로 요청이 생기지 않도록 함

5명 중 마지막 사람을 제외한 4명이 왼쪽 포크를 먼저 잡고 오른쪽 포크를 잡는 순서로 진행하고, 마지막 사람은 오른쪽 포크를 잡고 왼쪽 포크를 잡도록 함

# 식사하는 철학자 문제

6

Communicating Sequential Processes, C. A. R. Hoare

1985년 Prentice Hall 출판사에서 출간한 논문

- [Dijkstra](#)의 문제를 공식화한 내용 포함

고대에 부유한 박애주의자가 저명한 다섯 명의 철학자를 위해 대학을 기증하였습니다. 각 철학자는 자신의 전문적인 사고 활동을 할 수 있는 방과 공동으로 식사를 할 수 있는 식당이 제공되었습니다.

이 식당에는 5개의 의자로 둘러싸인 원형 테이블이 구비되어 있고, 테이블에는 앉을 철학자의 이름이 붙어 있습니다. 철학자의 이름은 PHIL0, PHIL1, PHIL2, PHIL3, PHIL4였으며, 테이블을 따라 반 시계 방향으로 배치되었습니다. 각 철학자의 왼쪽에는 황금 포크가, 중앙에는 한 개의 그릇에 스파게티가 놓여 있습니다. 스파게티는 끊임없이 보충됩니다.

철학자는 대부분의 시간을 사고로 보내지만, 배고플 때는 식당에 가서 자기 의자에 앉아, 자신의 왼쪽에 있는 포크를 들어 스파게티 속에 넣습니다. 그러나 스파게티를 입으로 가져가기 위해서는 두 번째 포크가 반드시 필요하므로, 철학자는 그의 오른쪽에 있는 포크도 가져야만 합니다. 먹기가 끝나면 두 포크를 내려 놓고 의자에서 일어나 계속 생각을 할 것입니다.

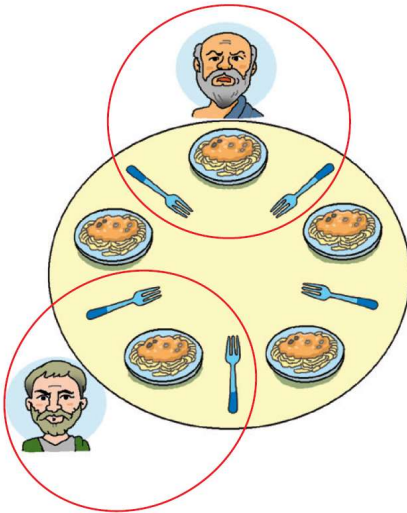
물론 하나의 포크는 한 번에 한 철학자만 사용할 수 있습니다. 다른 철학자가 포크를 사용하려면 사용가능하게 될 때까지 기다려야합니다. (저자의 번역)

철학자들이 식사하는 동안 언제 어떤 문제가 생길까요?

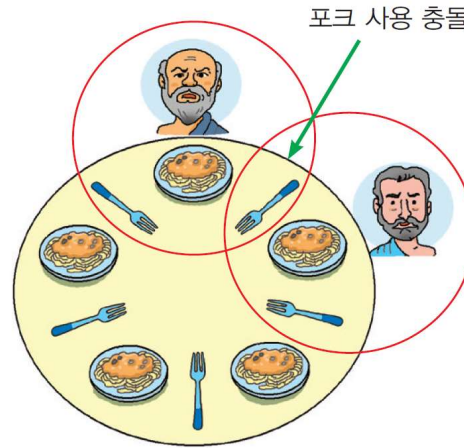
갑자기 모든 철학자들이 동시에 자리에 앉아 식사를 하려고 할 때 문제가 생깁니다.



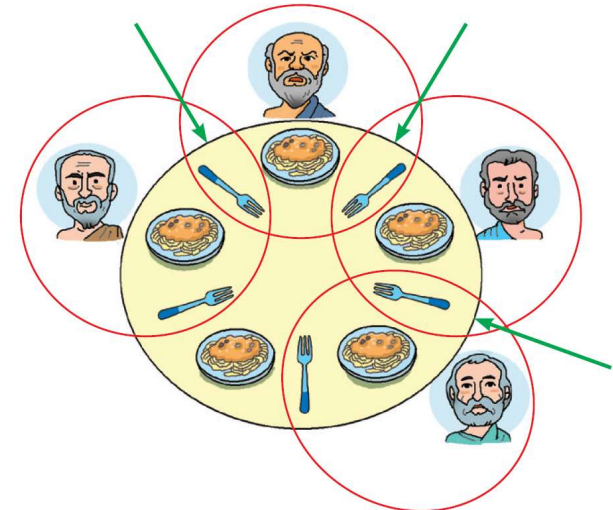
# 철학자가 식사하는 모든 경우 분석



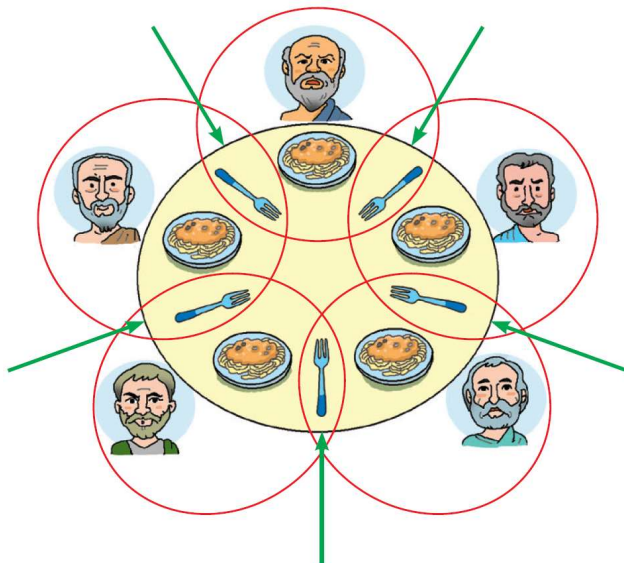
(1) 문제 없이 식사



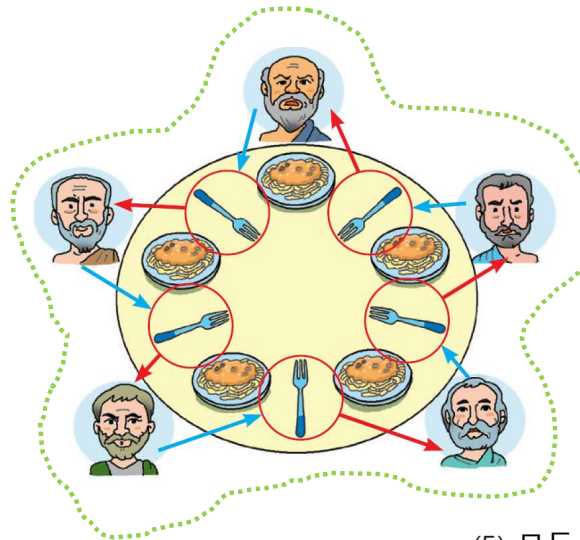
(2) 포크 사용 충돌 가능.  
둘 중 하나가 잠깐 대기



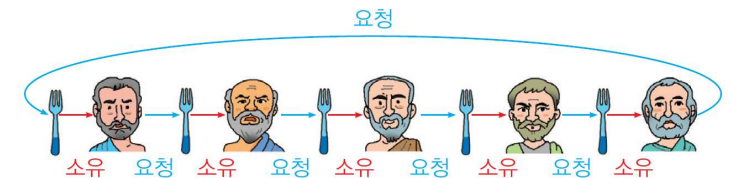
(3) 포크 사용 충돌 가능.  
누군가 잠깐 대기. 무한 대기는 발생하지 않음



(4) 포크 사용 충돌 가능. 포크를 잡는 순간이  
약간 다른 경우 여러 철학자가 잠깐씩 대기하  
면서 식사 가능



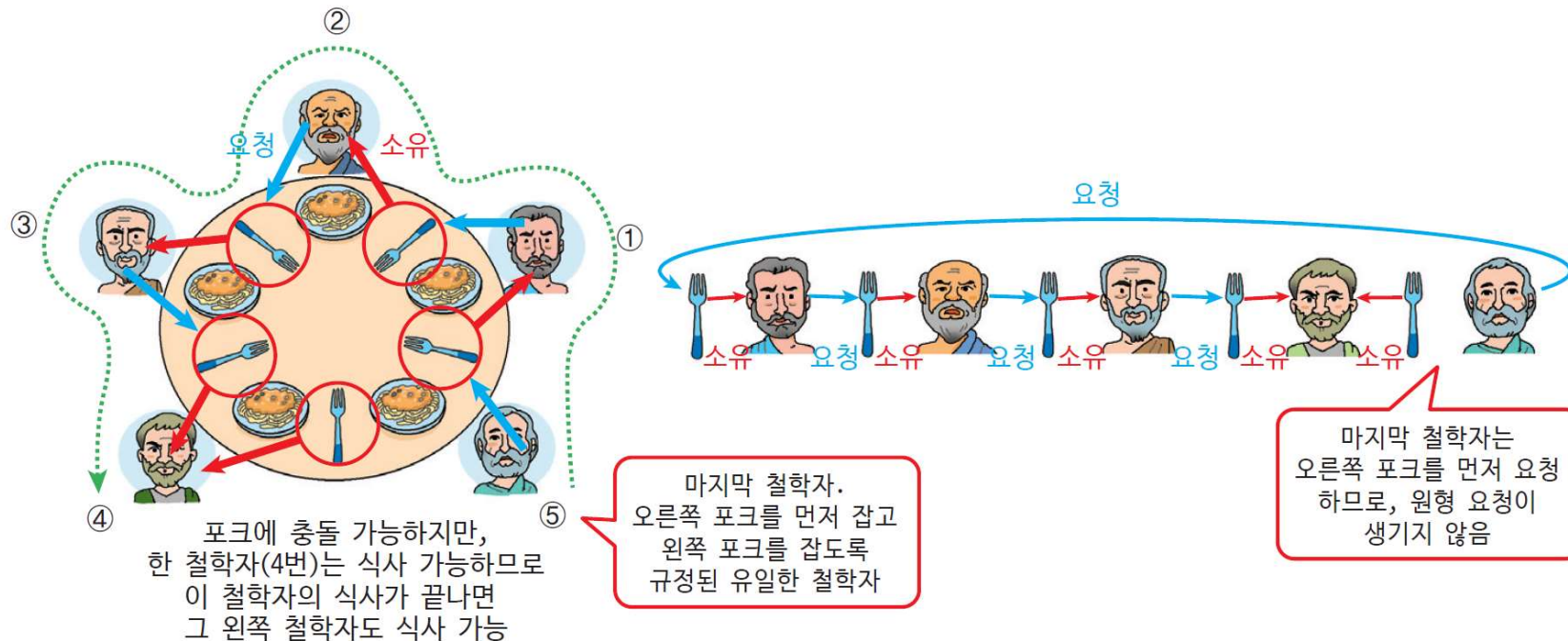
(5) 모든 철학자가 동시에 왼쪽 포크를 든 경우,  
오른쪽 포크를 들려고 할 때 모든 철학자의 무  
한 대기 발생



# 철학자들의 교착상태 원인과 해결

8

- 교착상태 원인 – 환형 요청/대기(circular wait)
  - 5명 모두 왼쪽 포크를 가지고 오른쪽 포크를 요청하는 환형 고리
  - 환형 고리는 스스로 인식하거나 해체 불가
- 교착상태 해결 – ‘환형 대기’가 생기지 않도록
  - 마지막 철학자(5번)만 오른쪽 포크를 먼저 잡고 왼쪽을 잡도록 규칙 수정

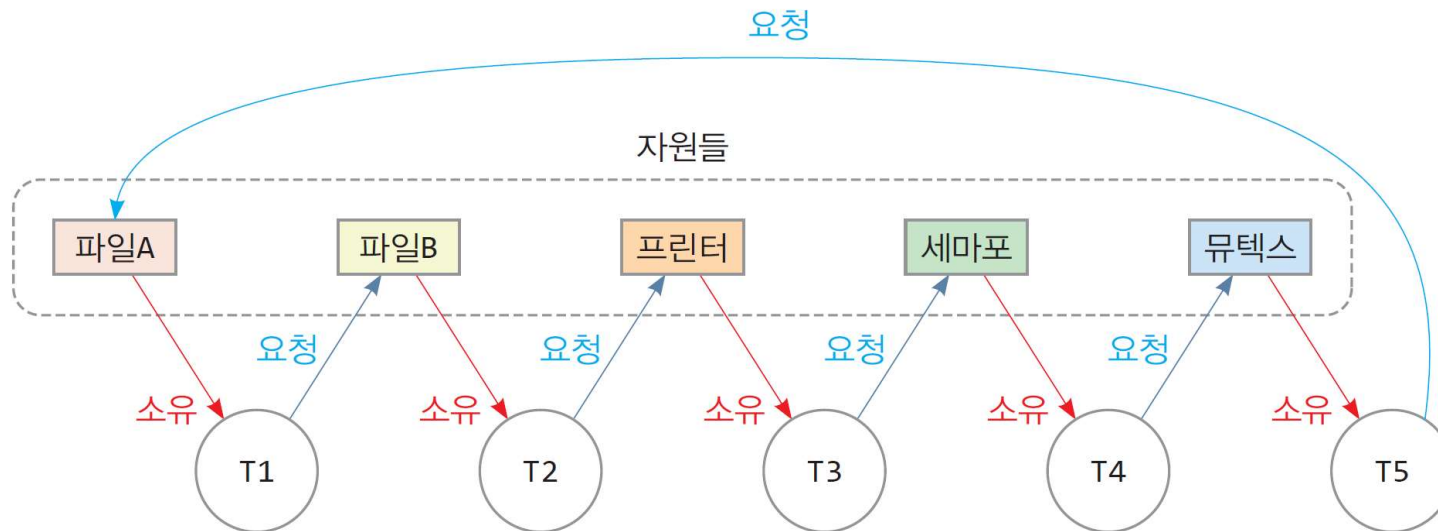




# 식사하는 철학자와 컴퓨터 시스템

9

- 식사하는 철학자 문제는 교착 상태를 비유
  - 교착상태는 다중프로그래밍 시스템 초기에 노출된 문제점
- 철학자 : 스레드
- 포크 : 자원
- 스파게티 : 스레드가 처리할 작업



T1~T5의 5개의 스레드들 사이에서 발생한 교착상태 모습

10

## 2. 교착상태

# 컴퓨터 시스템에서의 교착상태 정의

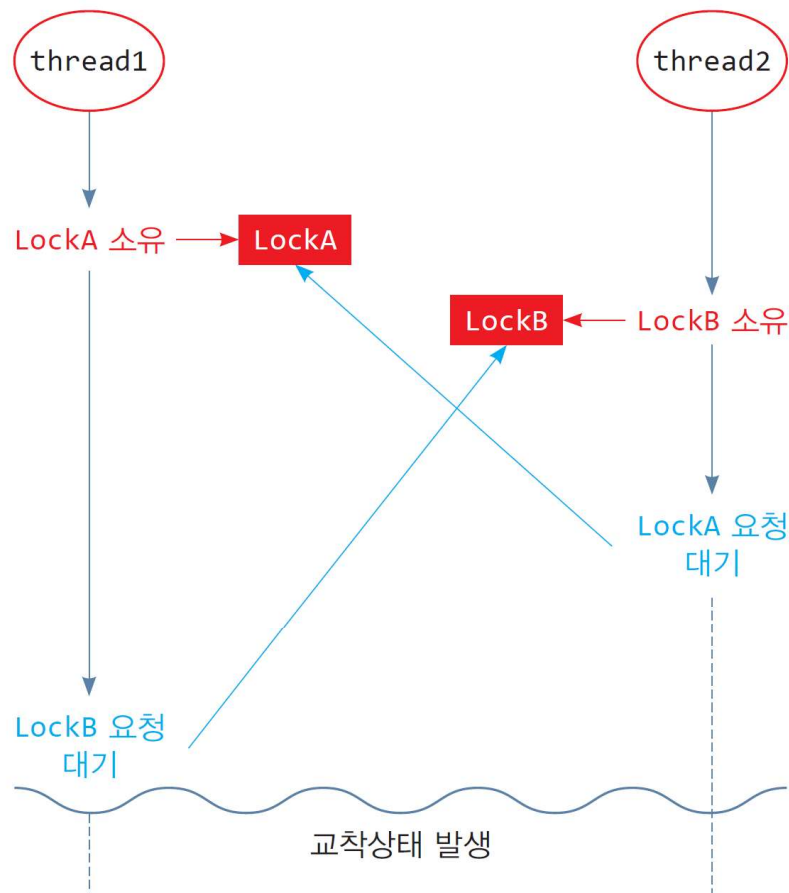
11

- 교착상태(deadlock)
  - ▣ 자원을 소유한 스레드들 사이에서,
  - ▣ 각 스레드는 다른 스레드가 소유한 자원을 요청하여 무한정 대기하는 현상
    - deadly embrace : 풀지못하는 포옹
    - 교착상태 문제는 1965년 Dijkstra의 banker's algorithm research에서 처음 제기
- 교착상태 발생 위치
  - ▣ 사용자가 작성한 멀티스레드 응용프로그램에서 주로 발생
    - 정교하지 못한 코딩에서 비롯
  - ▣ 커널 내에서도 발생
    - 하지만, 거의 발생하지 않음. 매우 정교하게 작성되기 때문
  - ▣ 교착상태를 막도록 운영하는 컴퓨터 시스템은 거의 없는 실상
    - 막는데 많은 시간과 공간 비용 때문. 그러면 어떻게 해결?
    - 교착상태가 발생하도록 두고, 교착상태가 발생한 것 같으면, 시스템 재시작, 혹은 의심스러운 몇몇 프로그램 종료

# 전형적인 멀티스레드 교착 상태 사례

12

- 교착상태의 전형적인 발생 사례
  - ▣ 2개의 스레드가 각각 락 **소유**, 상대가 가진 락 요청하고 **기다릴 때**
    - 단일CPU/다중CPU 모두에서 발생, T1과 T2가 서로 다른 CPU에서 실행될 때도 발생
  - ▣ 락과 자원에 대한 경쟁이 있는 한 교착상태는 언제든지 발생 가능



```
void* thread1(void *a)
{
    . . .
    pthread_mutex_lock(&LockA);
    . . .
    pthread_mutex_lock(&LockB);
    . . .
    pthread_mutex_unlock(&LockB);
    . . .
    pthread_mutex_unlock(&LockA);
}
```

교착상태  
발생 가능

```
void* thread2(void *a)
{
    . . .
    pthread_mutex_lock(&LockB);
    . . .
    pthread_mutex_lock(&LockA);
    . . .
    pthread_mutex_unlock(&LockA);
    . . .
    pthread_mutex_unlock(&LockB);
}
```

교착상태  
발생 가능

# 컴퓨터 시스템에 잠재된 교착상태 유발 요인

13

## 1. 자원은 교착 상태의 발생지

- 교착상태는 멀티스레드가 자원을 동시에 사용하려는 충돌이 요인
- ▣ 컴퓨터 시스템에는 많은 자원 존재
  - 소프트웨어 자원 – 뮤텍스, 스핀락, 세마포, 파일, 데이터베이스, 파일 락
  - 하드웨어 자원 – 프린터, 메모리, 프로세서 등

## 2. 자원과 스레드

- ▣ 한 스레드가 여러 자원을 동시에 필요로 하는 상황이 요인

## 3. 자원과 운영체제

- ▣ 한 번에 하나씩 자원을 할당하는 운영체제 정책이 요인
  - 만일 스레드가 필요한 자원을 한 번에 모두 요청하도록 한다면,
  - 교착상태가 발생하지 않게 할 수 있다.

## 4. 자원 비선점

- ▣ 할당된 자원은 스레드가 자발적으로 내놓기 전에 강제로 뺏지 못하는 정책이 요인
  - 운영체제는 스레드가 가진 자원을 강제로 뺏지 못함
  - 만일 강제로 빼앗을 수 있다면,
  - 교착상태가 발생하지 않게 할 수 있다.



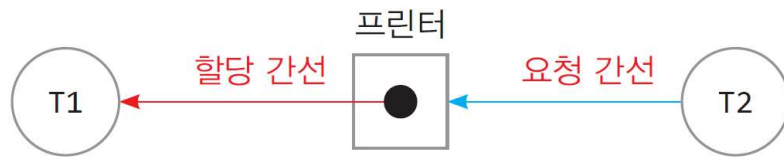
# 교착상태 모델링

14

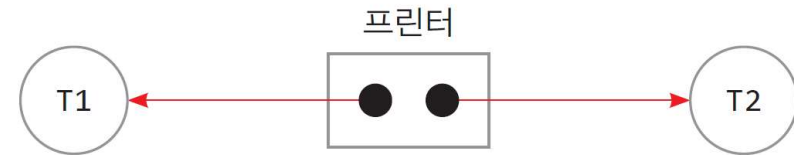
- 자원 할당 그래프(Resource Allocation Graph, RAG)
  - ▣ 그래프의 요소
    - 꼭지점(vertex) – 스레드(원), 자원(사각형)
    - 간선(edge) – 소유/요청 관계. 할당 간선과 요청 간선
      - 할당 간선 : 자원에서 스레드로 향하는 화살표. 할당 받은 상태 표시
      - 요청 간선 : 스레드에서 자원으로 향하는 화살표. 요청 표시
  - ▣ 시스템내 자원들의 상태를 나타내는 방향성 그래프
    - 컴퓨터 시스템에 실행 중인 전체 스레드와 자원의 개수
    - 각 자원의 총 인스턴스 개수와 할당 가능한 인스턴스 개수
    - 각 스레드가 할당받아 소유하고 있는 자원의 인스턴스 개수
    - 각 스레드가 실행에 필요한 자원 유형과 인스턴스 개수
- 자원 할당 그래프를 통해 교착상태 판단
  - ▣ 교착상태 예방, 회피, 감지를 위한 알고리즘 개발에 필요

# 자원 할당 그래프 사례

15



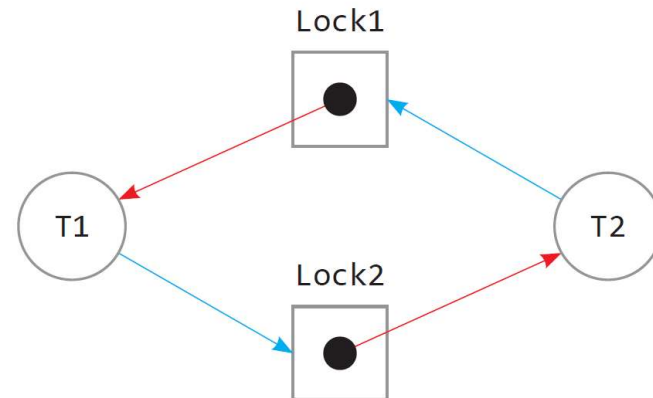
(a) 시스템에는 2개의 스레드 T1과 T2 그리고 프린터 1개 있음. T1은 프린터 소유, T2는 프린터 요청 대기



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



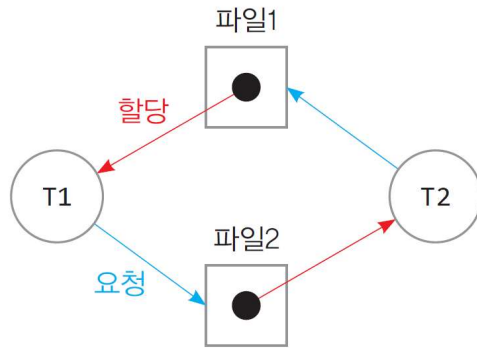
(c) 시스템에는 T1 스레드와 2개의 프린터 있음. T1이 프린터 1개 소유



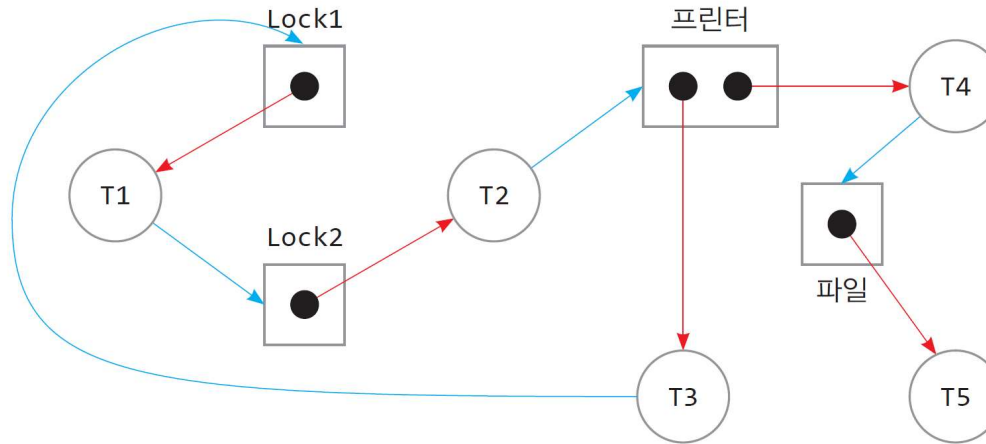
(d) 시스템에는 2개의 스레드 T1과 T2 그리고 Lock1과 Lock2의 두 자원이 있음. T1은 Lock1을 소유하고 Lock2를 요청, T2는 Lock2를 소유하고 Lock1을 요청

# 교착상태가 발생 사례

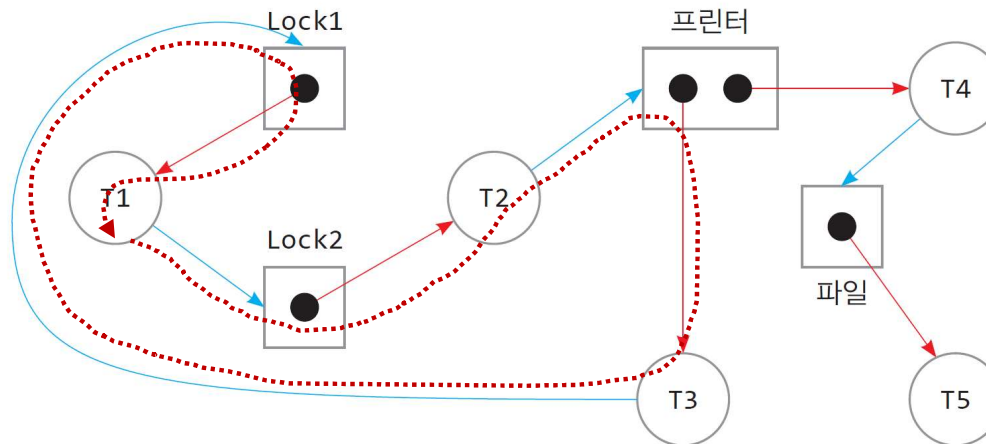
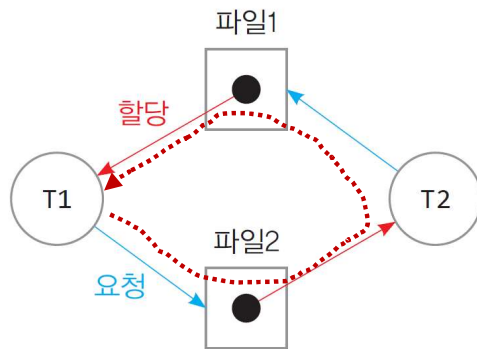
16



(a) T1, T2의 교착 상태



(b) T1, T2, T3의 교착 상태



# 탐구 7-1: 교착상태에 빠진 응용프로그램 사례

## deadlock.c

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

int x = 0; // 공유 변수
int y = 0; // 공유 변수
pthread_mutex_t lock1; // 뮤텍스 락 변수
pthread_mutex_t lock2; // 뮤텍스 락 변수

void* worker1(void* arg) { // 스레드 코드
    pthread_mutex_lock(&lock1); // x를 독점 사용하기 위해 lock1 잠그기
    printf("%s lock1 잠금\n", (char*)arg);
    x++;
    sleep(2); // 2초 잠자기

    pthread_mutex_lock(&lock2); // y를 독점 사용하기 위해 lock2 잠그기
    printf("%s lock2 잠금\n", (char*)arg);
    y++;
    pthread_mutex_unlock(&lock2); // lock2 풀기
    printf("%s lock2 해제\n", (char*)arg);

    pthread_mutex_unlock(&lock1); // lock1 풀기
    printf("%s lock1 해제\n", (char*)arg);
}

void* worker2(void* arg) { // 스레드 코드
    pthread_mutex_lock(&lock2); // y를 독점 사용하기 위해 lock2 잠그기
    printf("%s lock2 잠금\n", (char*)arg);
    y++;
    sleep(2); // 2초 잠자기

    pthread_mutex_lock(&lock1); // x를 독점 사용하기 위해 lock1 잠그기
    printf("%s lock1 잠금\n", (char*)arg);
    x++;
    pthread_mutex_unlock(&lock1); // lock1 풀기
    printf("%s lock1 해제\n", (char*)arg);

    pthread_mutex_unlock(&lock2); // lock2 풀기
    printf("%s lock2 해제\n", (char*)arg);
}
```

```
int main() {
    char *name[] = {"황기태", "이찬수"};
    pthread_t tid[2];

    pthread_mutex_init(&lock1, NULL); // 뮤텍스 락 변수 lock1 초기화
    pthread_mutex_init(&lock2, NULL); // 뮤텍스 락 변수 lock2 초기화

    pthread_create(&tid[0], NULL, worker1, name[0]); // worker1 스레드 생성
    pthread_create(&tid[1], NULL, worker2, name[1]); // worker2 스레드 생성

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

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

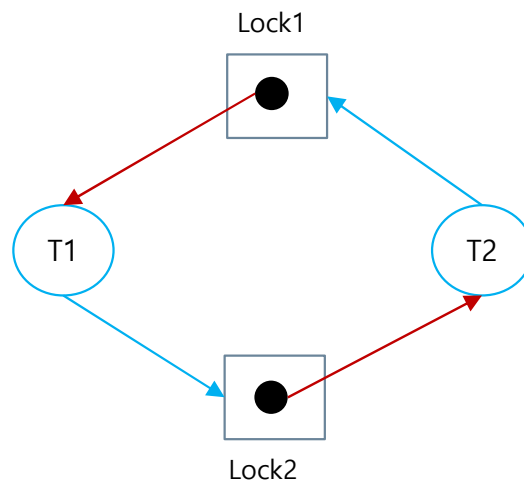
    printf("x = %d, y = %d\n", x, y);

    return 0;
}
```

```
$ gcc -o deadlock deadlock.c -lpthread
$ ./deadlock
황기태 lock1 잠금
이찬수 lock2 잠금
(무한정 대기 상태)
^C
$
```

# 탐구 7-1 코드의 교착 상태 발생 과정

| worker1 스레드                            | 공유 변수 x            | 공유 변수 y            | worker2 스레드                            |
|--|--------------------|--------------------|--|
| <b>pthread_mutex_lock(&amp;lock1);</b> | <b>lock1 소유</b>    |                    |  |
| x++;                                   | x 독점사용가능           |                    |  |
| sleep(2);                              |                    |                    |  |
| [blocked]                              |                    | <b>lock2 소유</b>    | <b>pthread_mutex_lock(&amp;lock2);</b> |
| [blocked]                              |                    | y 독점사용가능           | y++;                                   |
| [blocked]                              |                    |                    | sleep(2);                              |
| [blocked]                              |                    |                    | [blocked]                              |
| [blocked]                              |                    |                    | [blocked]                              |
| [blocked]                              |                    |                    | [blocked] 2초 경과                        |
| <b>pthread_mutex_lock(&amp;lock2);</b> | <b>lock2 요청/대기</b> |                    | [blocked]                              |
| [waiting lock2]                        |                    | <b>lock1 요청/대기</b> | <b>pthread_mutex_lock(&amp;lock1);</b> |
| [waiting lock2]                        |                    |                    | [waiting lock1]                        |
| [waiting lock2]                        | <b>교착상태 발생!</b>    |                    | [waiting lock1]                        |
| [waiting lock2]                        |                    |                    | [waiting lock1]                        |
| [waiting lock2]                        |                    |                    | [waiting lock1]                        |



교착 상태 발생 조건을 모두 보여줌

- 상호 배제
- 소유와 대기
- 선점 없이 대기
- 원형 대기



19

## 3. 교착상태 해결

# 교착상태가 발생하는 4가지 필요충분 조건

20

- 코프만 조건(Coffman condition)
  - ▣ 교착상태가 발생하는 4가지 필요충분 조건
    - Computing Survey, Vol. 3, No. 2, June, 1971에 실린 논문
  - ▣ 다음 4가지 상황이 허용되는 시스템은 언제든지 교착상태 발생 가능
    - 상호 배제(Mutual Exclusion)
      - 각 자원은 한 번에 하나의 스레드에게만 할당
    - 소유하면서 대기(Hold & Wait)
      - 스레드가 한 자원을 소유하면서 다른 자원을 기다리기
    - 강제 자원 반환 불가(No Preemption)
      - 스레드에게 할당된 자원을 강제로 빼앗지 못함
    - 환형 대기(Circular Wait)
      - 한 그룹의 스레드들에 대해, 각 스레드는 다른 스레드가 요청하는 자원을 소유하는 환형 고리 형성
  - ▣ 4가지 조건 중 하나라도 성립되지 않으면, 교착상태 발생 않음

# 교착상태 해결 방법

21

## 1. 교착상태 예방(prevention)

- ▣ 교착상태 발생 여지를 차단하여 예방
- ▣ 교착상태에 빠지는 4가지 코프만 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 불안정한 상태로 시스템 상태 분류, 안전한 상태인 경우에만 자원 할당
  - banker's algorithm
  - 자원 할당 시마다 교착 상태 가능성을 검사하므로 시스템 성능 저하

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

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

## 4. 교착상태 무시

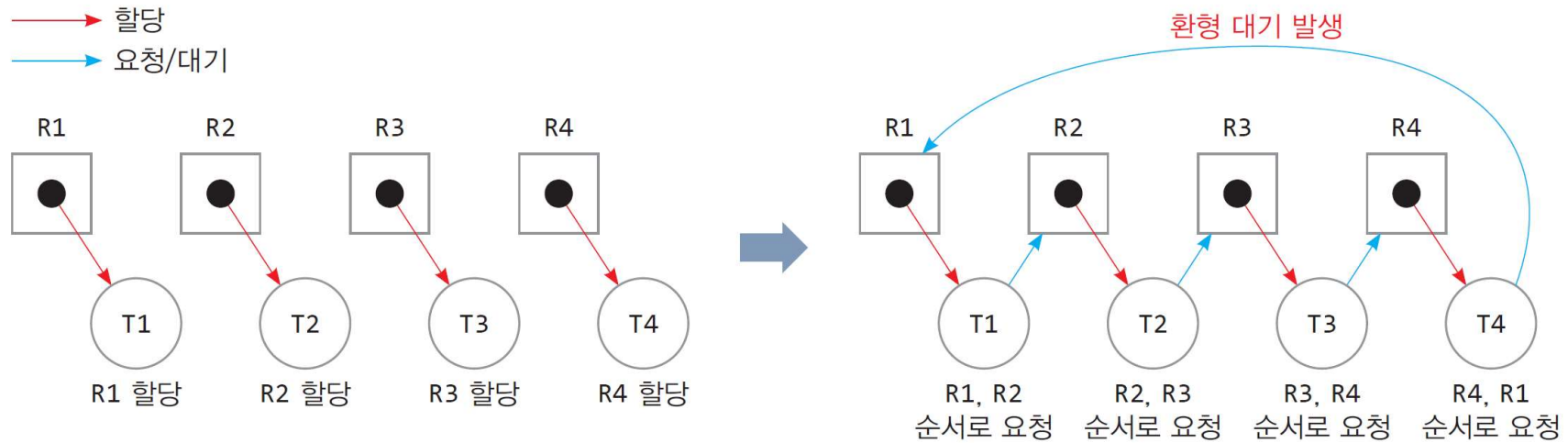
- ▣ 교착상태는 없다고 단정, 아무런 대비책 없음
- ▣ 사용자나 관리자가 이상을 느끼면 재실행할 것이라고 믿는 방법
- ▣ 리눅스, 윈도우 등 현재 대부분의 운영체제에서 사용하는 가장 일반적인 방법
  - 교착상태 예방, 회피, 감지 복구 등에는 많은 시간과 공간이 필요하며 시스템의 성능을 떨어뜨리기 때문
  - 교착상태가 발생해도 파국이 일어나지 않는 범용 시스템에서는 교착상태 무시
  - ostrich 알고리즘

# 교착상태 예방

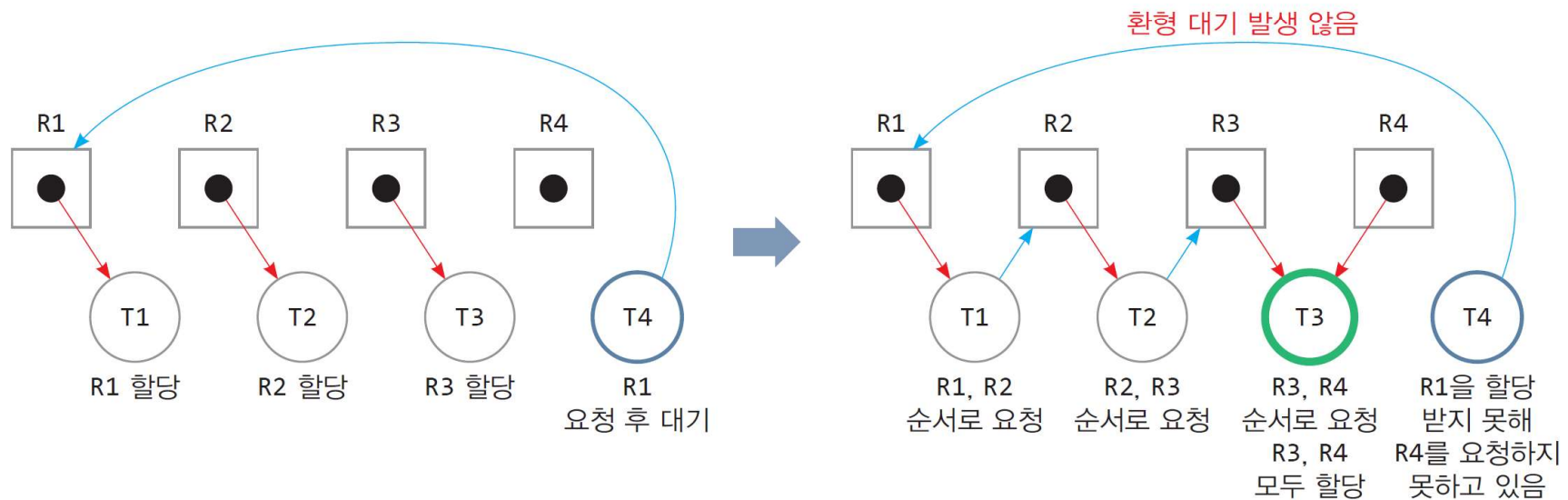
22

- 코프만의 4가지 조건 중 최소 하나를 성립하지 못하게 함
  1. 상호 배제(Mutual Exclusion) 조건 -> 상호 배제 없애기
    - 동시에 2개 이상의 스레드가 자원을 활용할 수 있도록 함
    - > 컴퓨터 시스템에서 근본적으로 적용 불가능
  2. 소유하면서 대기(Hold & Wait) 조건 -> 기다리지 않게
    - 방법 1 : 운영체제는 스레드 실행 전 필요한 모든 자원을 파악하고 실행 시 한 번에 할당
      - >당장 사용하지 않는 자원을 스레드에게 묶어 두므로 자원 활용률 저하
      - >다른 스레드는 필요한 자원을 할당 받지 못하고 실행 대기
    - 방법 2 : 스레드가 새로운 자원을 요청하려면, 현재 할당 받은 모든 자원을 반환하고, 한꺼번에 요청하여 할당
      - >방법1과 방법2 모두 가능하지 않거나 매우 비효율적
  3. 강제 자원 반환 불가(No Pre-emption) 조건 -> 선점 허용
    - 자원을 강제로 반환하게 된 스레드가 자원을 다시 사용하게 될 때 이전 상태로 되돌아갈 수 있도록 상태를 관리할 필요
    - >간단치 않고 오버헤드 매우 큼
  4. 환형 대기(Circular Wait) 조건 -> 환형 대기 제거
    - 모든 자원에 번호를 매기고, 번호순으로 자원 할당

# 환형 대기 발생하지 않도록 번호 순으로 자원 할당



(a) 환형 대기가 발생하는 경우



(b) 스레드가 번호 순으로 자원을 할당 받아 환형 대기가 발생하지 않는 경우.  
T4가 R1과 R4가 필요할 때, R1부터 할당 받으면 환형 대기 발생 않음



# 교착상태 회피

24

- 자원 할당 시, 미래에 환형 대기가 발생할 것으로 판단되면 자원 할당 하지 않는 정책
- banker's 알고리즘으로 해결
  - ▣ [Edsger Dijkstra](#) 에 의해 개발된 알고리즘.
  - ▣ 자원 할당 전에 미래에 교착상태가 발생하지 않을 것인지 안전한지 판단하는 알고리즘
    - 은행에서의 대출 알고리즘(돈을 대출한 사람, 돈을 대출하려고 하는 사람)
  - ▣ 안전한 상태
    - 현재 프로세스들을 어떤 순서로 실행 시켰을 때, 모든 프로세스들이 자신이 요청하는 자원을 가지고 실행할 수 있다면 안전한 상태
  - ▣ 불안정한 상태
    - 환형 대기에 빠질 수 있다면 불안정한 상태
  - ▣ 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 자원의 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

# 교착상태 감지 및 복구

25

- 교착상태를 감지하는 프로그램을 통해, 형성된 교착상태를 푼다.
  - ▣ 백그라운드에서 교착상태를 감지하는 프로그램 늘 실행
- 교착상태를 감지하였을 때의 복구 방법
  - ▣ 자원 강제 선점(preemption)
    - 교착상태에 빠진 스레드 중 하나의 자원을 강제로 빼앗아 다른 스레드에게 할당. 강제로 자원을 뺏긴 스레드는?
  - ▣ 롤백(rollback)
    - 교착상태가 발생할 것으로 예측되는 스레드들의 상태를 주기적으로 저장,
    - 교착상태가 발생하면 마지막 저장 상태로 돌아가도록 하고,
    - 다시 시작하면 스케줄링 등의 여러 요인에 의해 자원 할당 순서가 달라짐
  - ▣ 스레드 강제 종료(kill process)
    - 교착상태에 빠진 스레드 중 하나를 선택하여 강제 종료
    - 가장 간단하면서도 효과적임. 하지만 종료된 스레드의 피해는 누가 보상?
  - ▣ 시간과 메모리 공간(rollback의 경우)에 대한 부담이 크기 때문에 잘 사용하지 않음

# 다음 그림에 등장하는 동물은?

26

## □ 어떤 동물이, 무엇을 하고 있는 걸까?

### ▣ 타조(ostrich)

- 사전적 의미 : 날지 못하는 매우 큰 새
- 1600년대 초 '현실 도피 주의자'
  - 먹이를 찾느라 모래에 머리를 박은 모습을 쫓길 때 모래에 머리를 박는다고 오인해 생긴 뜻
  - 겁쟁이를 치킨이라고 부르는 것과 유사



# 교착상태 무시 : 타조(Ostrich) 알고리즘

27

- 교착상태를 해결할 필요가 있을까?
  - ▣ 교착상태에 대한 통계치는 없다
    - 1년에 한 번, 혹은 10년에 한번?
  - ▣ 교착상태는 반드시 발생한다.
    - 하지만, 교착상태의 발생 가능성이 극히 적은 반면,
    - 교착상태를 피하기 위한 많은 비용 소요
- 타조 알고리즘
  - ▣ Put your head in the sand 접근법
    - 타조가 머리를 모래 속에 박고 자신이 보이지 않는 체하는 것
    - '교착상태는 발생하지 않을 거야' 하고 아무 대책을 취하는 않는 접근법
  - ▣ Unix(리눅스)와 윈도우 등 현재 거의 모든 운영체제에서 사용
    - 사용자나 관리자가 의심 가는 스레드를 종료시키거나 시스템 재시작(reboot)하는 방법으로 해결
    - 거의 발생하지 않거나 아주 드물게 발생하는 것에 비해 교착상태 해결에는 상대적으로 비용이 많이 들기 때문
- 주의
  - ▣ 핵 시스템, 비행기, 미사일 등 시스템 재시작이 파국을 초래할 hard real-time 시스템이나 환자 감시 시스템 등에는 적합하지 않음

# 교착상태를 다루는 현실적인 방안

28

- 대부분의 운영체제 : ostrich 알고리즘 사용
  - ▣ 교착상태가 일어나지 않을 것으로 가정하고, 교착상태에 대한 아무 대책을 세우지 않음
    - 교착상태가 발생할 확률은 극히 작고,
    - 교착상태 예방, 회피, 감지에는 많은 오버헤드가 소모되므로
  - ▣ 교착상태가 발생하면 사용자나 관리자가 시스템 재시작 혹은 특정 프로세스/스레드 강제 종료
    - 관련된 데이터를 잃어버릴 수 있음
    - 하지만 전체적으로 크지 않은 손실



# 교착상태를 대하는 통찰

29

- ▣ "Not everything worth doing is worth doing well"
  - "할만한 가치 있는 일들이 모두 잘할 만큼 가치 있는 것은 아니다."