

Chapter 05

프로세스 동기화

Contents

- 01** 프로세스 간 통신
- 02** 공유 자원과 임계구역
- 03** 임계구역 해결 방법
- 04** [심화학습] 파일, 파이프, 소켓 프로그래밍

학습목표

- 프로세스 간 통신의 개념을 이해하고 종류를 파악한다.
- 공유 자원 사용 시의 임계구역 문제를 알아본다.
- 임계구역 문제를 해결하기 위한 조건과 해결 방법을 알아본다.

1. 병행 프로세스의 개념

■ 병행 프로세스 Concurrent Process 종류

■ 독립 프로세스

- 단일 처리 시스템에서 수행하는 병행 프로세스, 다른 프로세스에 영향 주고받지 않으면서 독립 실행

■ 협력 프로세스

- 다른 프로세스와 상호작용(통신)하며 특정 기능 수행하는 비동기적 프로세스
- 제한된 컴퓨터 자원의 효율성 증대, 계산 속도 향상, 모듈적 구성 강화, 개별 사용자의 여러 작업 동시에 수행 편의성 제공에 사용

■ 병행 프로세스들이 입출력장치, 메모리, 프로세서, 클록 등 자원을 서로 사용 시 충돌 발생(경쟁 조건 race condition 발생)

2. 병행 프로세스의 과제

■ 병행성(Concurrency)

- 여러 프로세스를 이용하여 작업을 수행하는 것
- 시스템 신뢰도 높이고 처리 속도 개선, 처리 능력 높이는 데 중요

■ 병행 프로세스의 문제

- 공유 자원 상호 배타적 사용(프린터, 통신망 등은 한순간에 프로세스 하나만 사용)
 - 병행 프로세스 수행 과정에서 발생하는 상호배제 보장
- 병행 프로세스 간의 협력이나 동기화(상호배제도 동기화의 한 형태)
- 두 프로세스 간 데이터 교환을 위한 통신
- 동시에 수행하는 다른 프로세스의 실행 속도와 관계 없이 항상 일정한 실행 결과 보장(결정성^{determinacy}) 확보
- 교착 상태 해결, 병행 프로세스들의 병렬 처리 능력 극대화

1 프로세스 간 통신의 개념

■ 프로세스 간 통신의 종류

- 프로세스 내부 데이터 통신
 - 하나의 프로세스 내에 2개 이상의 스레드가 존재하는 경우의 통신
 - 프로세스 내부의 스레드는 전역 변수나 파일을 이용하여 데이터를 주고받음
- 프로세스 간 데이터 통신(IPC Inter-process Communication)
 - 같은 컴퓨터에 있는 여러 프로세스끼리 통신하는 경우
 - 공용 파일 또는 운영체제가 제공하는 파이프를 사용하여 통신
- 네트워크를 이용한 데이터 통신
 - 여러 컴퓨터가 네트워크로 연결되어 있을 때 통신
 - 원격 프로시저 호출(RPC Remote Process Communication)과 소켓(Socket)을 이용하여 데이터를 주고받음

1 프로세스 간 통신의 개념

■ 프로세스 간 통신의 종류

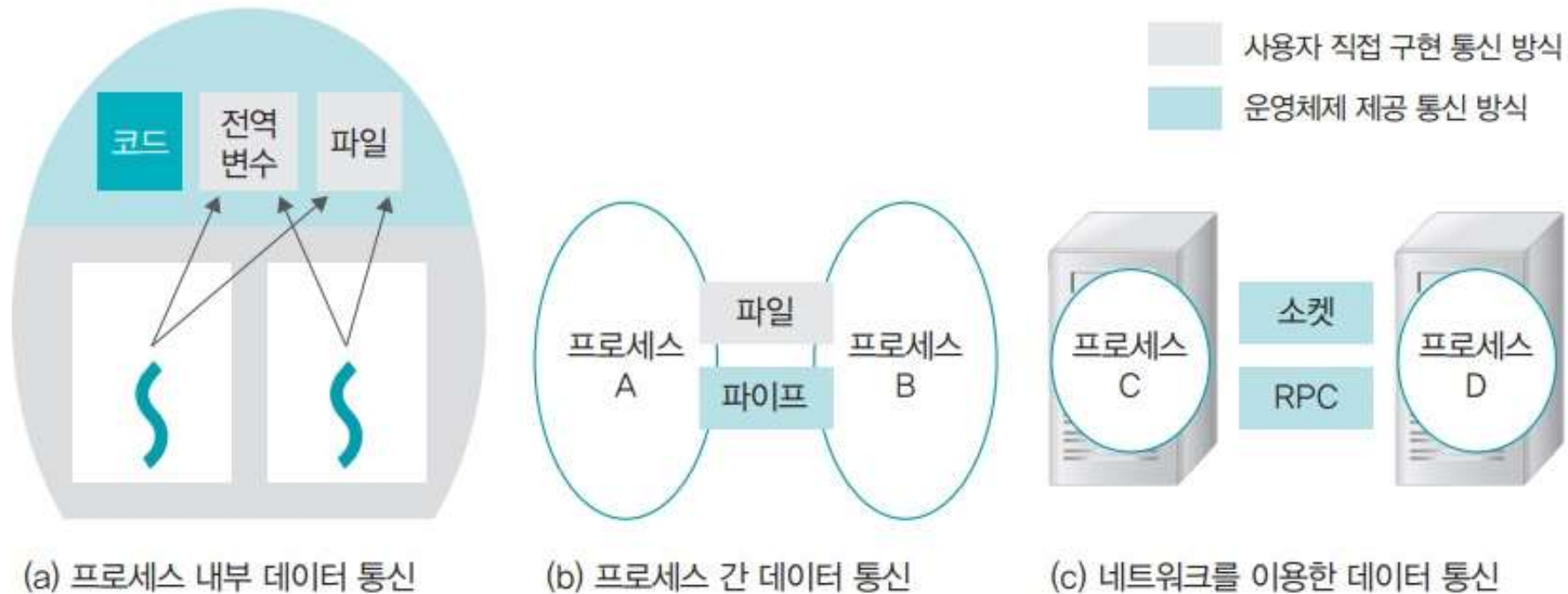


그림 5-1 프로세스 간 통신의 종류

1 프로세스 간 통신의 개념

■ 통신 방식의 이해

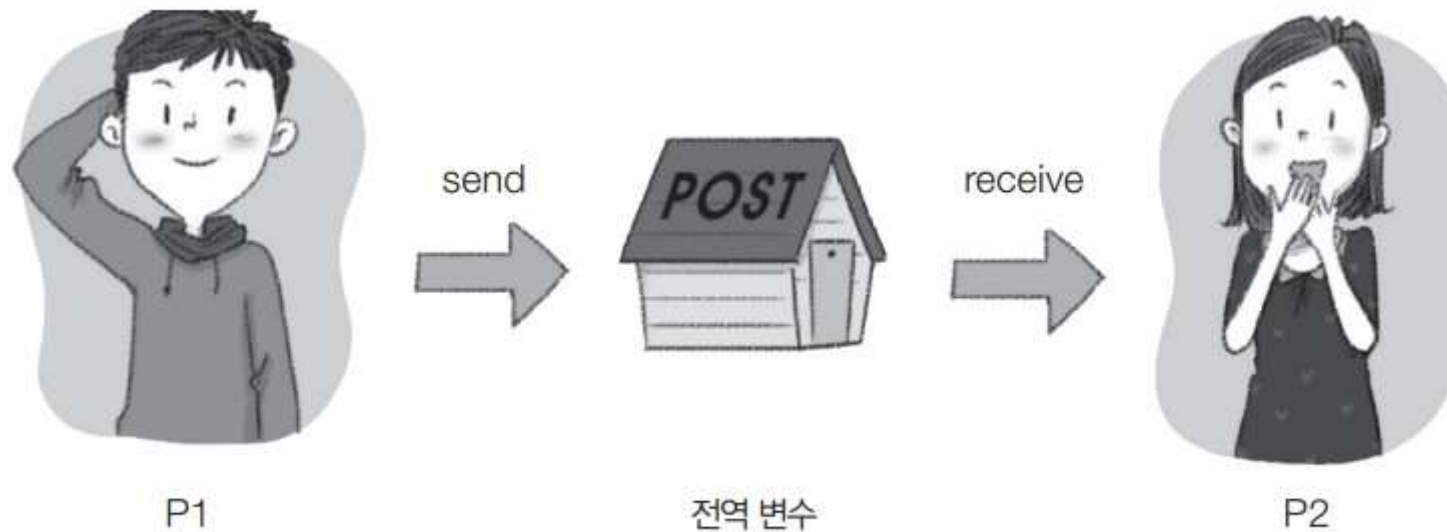


그림 5-2 통신 방식의 이해

1 공유 자원의 접근

■ 공유 자원 **Sharing Resources**

- 여러 프로세스가 공동으로 이용하는 변수, 메모리, 파일 등을 말함
- 공동으로 이용되기 때문에 누가 언제 데이터를 읽거나 쓰느냐에 따라 그 결과가 달라질 수 있음

■ 경쟁 조건 **Race Condition**

- 2개 이상의 프로세스가 공유 자원을 병행적으로 읽거나 쓰는 상황
- 경쟁 조건이 발생하면 공유 자원 접근 순서에 따라 실행 결과가 달라질 수 있음

■ 상호배제 **Mutual Exclusion**의 개념

- 특정 공유 자원을 한 순간에 하나의 프로세스만 사용할 수 있을 때, 프로세스 하나가 공유 데이터를 접근하는 동안 다른 프로세스들은 해당 데이터를 접근할 수 없게 하는 것
- 동기화 **Synchronization** 필요함
 - 공유 자원을 동시에 사용하지 못하게 실행을 제어하는 기법
 - 동기화로 상호배제 보장할 수 있지만, 이 과정에서 교착 상태와 기아 상태가 발생할 수 있음

1 공유 자원의 접근

■ 공유 자원의 접근 예

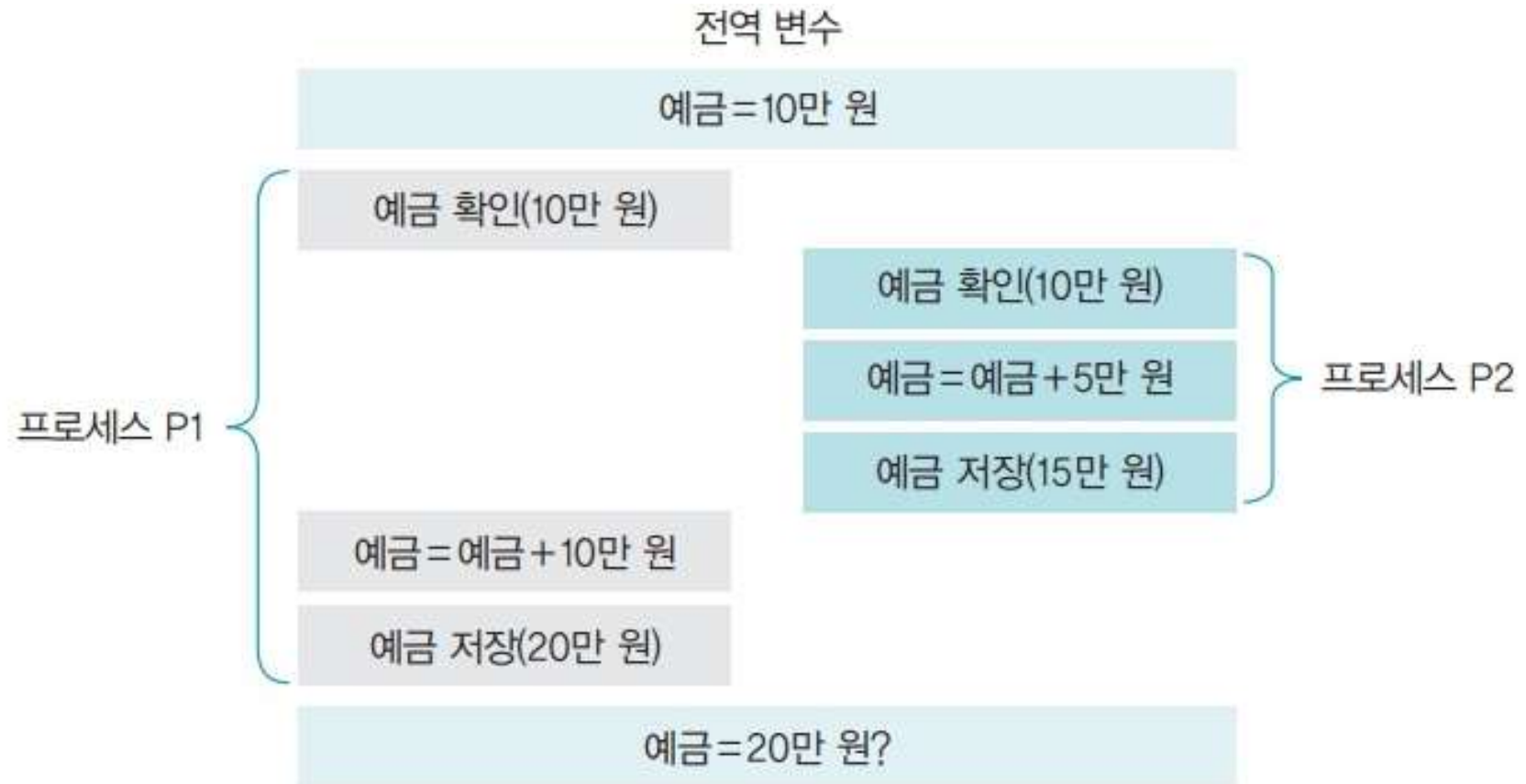


그림 5-10 공유 자원의 접근

2 임계 구역

■ 임계 구역 Critical Section/ Critical Region

- 둘 이상의 프로세스가 공유할 수 없는 자원을 **임계 자원(Critical Resource)**이라고 함
- 프로그램에서 임계 자원을 이용하는 부분을 **임계 영역(Critical Section)**이라고 함
- 다수의 프로세스 접근 가능하지만, 어느 한 순간에는 프로세스 하나만 사용 가능
- 공유 자원 접근 순서에 따라 실행 결과가 달라지는 프로그램의 영역
- 임계구역에서는 프로세스들이 동시에 작업하면 안 됨
 - 어떤 프로세스가 임계구역에 들어가면 다른 프로세스는 임계구역 밖에서 기다려야 하며, 임계구역의 프로세스가 나와야 들어갈 수 있음



그림 5-11 가스레인지와 믹서

3 생산자-소비자 문제

■ 코드 및 실행 순서에 따른 결과

- 생산자는 수를 증가시켜가며 물건을 채우고 소비자는 생산자를 쫓아가며 물건을 소비
- 생산자 코드와 소비자 코드가 동시에 실행되면 문제가 발생

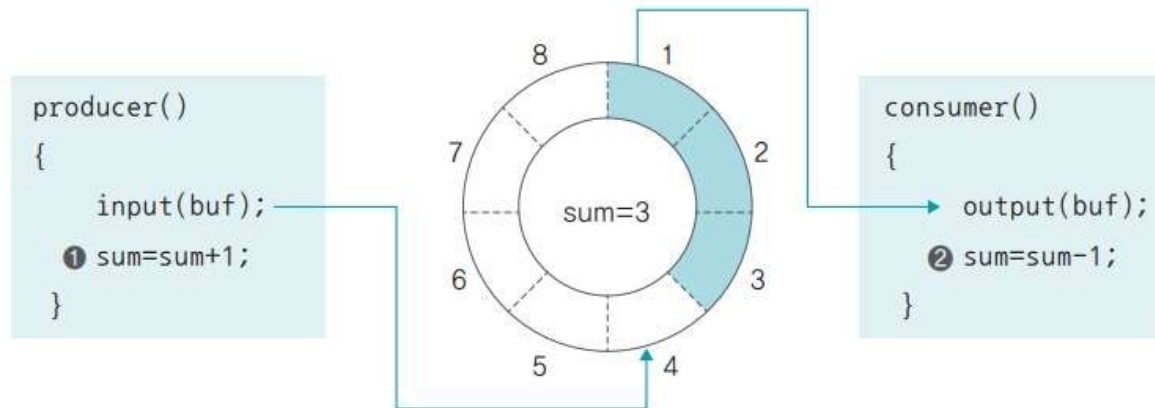


그림 5-12 생산자-소비자 문제

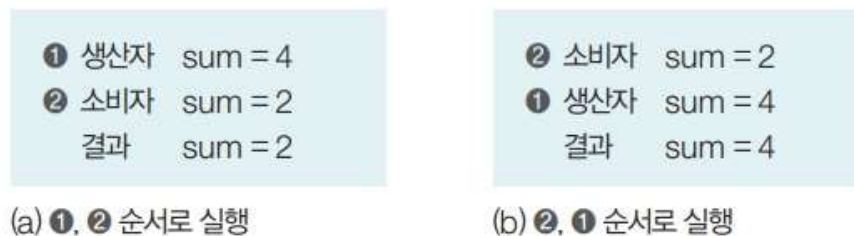


그림 5-13 실행 순서에 따른 결과 차이

3 생산자-소비자 문제

■ 하드웨어 자원을 공유하면 발생하는 문제

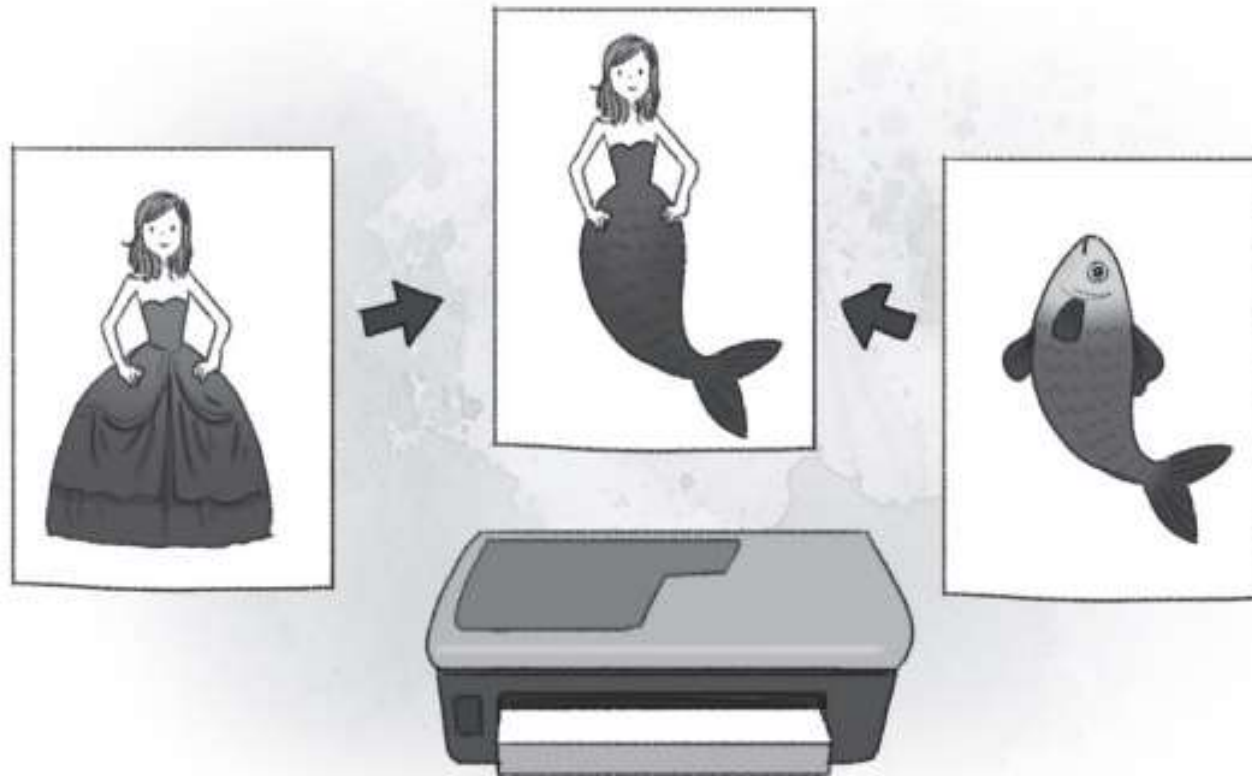


그림 5-14 하드웨어 자원을 공유하면 발생하는 문제

4 임계구역 해결 조건

■ 상호 배제 **mutual exclusion**

- 한 프로세스가 임계구역에 들어가면 다른 프로세스는 임계구역에 들어갈 수 없는 것

■ 진행(의 융통성) **progress flexibility**

- 한 프로세스가 다른 프로세스의 진행을 방해해서는 안 된다는 것
- 임계 영역에 프로세스가 없는 상태에서는 어떤 프로세스든 진행할 수 있어야 함

■ 한정 대기 **bounded waiting**

- 어떤 프로세스도 무한 대기하지 않아야 함

1 기본 코드 소개

■ 임계구역 해결 방법을 설명하기 위한 기본 코드

```
#include <stdio.h>

typedef enum {false, true} boolean;
extern boolean lock=false;
extern int balance;

main() {
    while(lock==true);
    lock=true;
    balance=balance+10;    /* 임계구역 */
    lock=false;
}
```


 **No-Operation**

그림 5-15 기본 코드

2 임계구역 해결 조건을 고려한 코드 설계

■ 전역 변수로 잠금을 구현한 코드

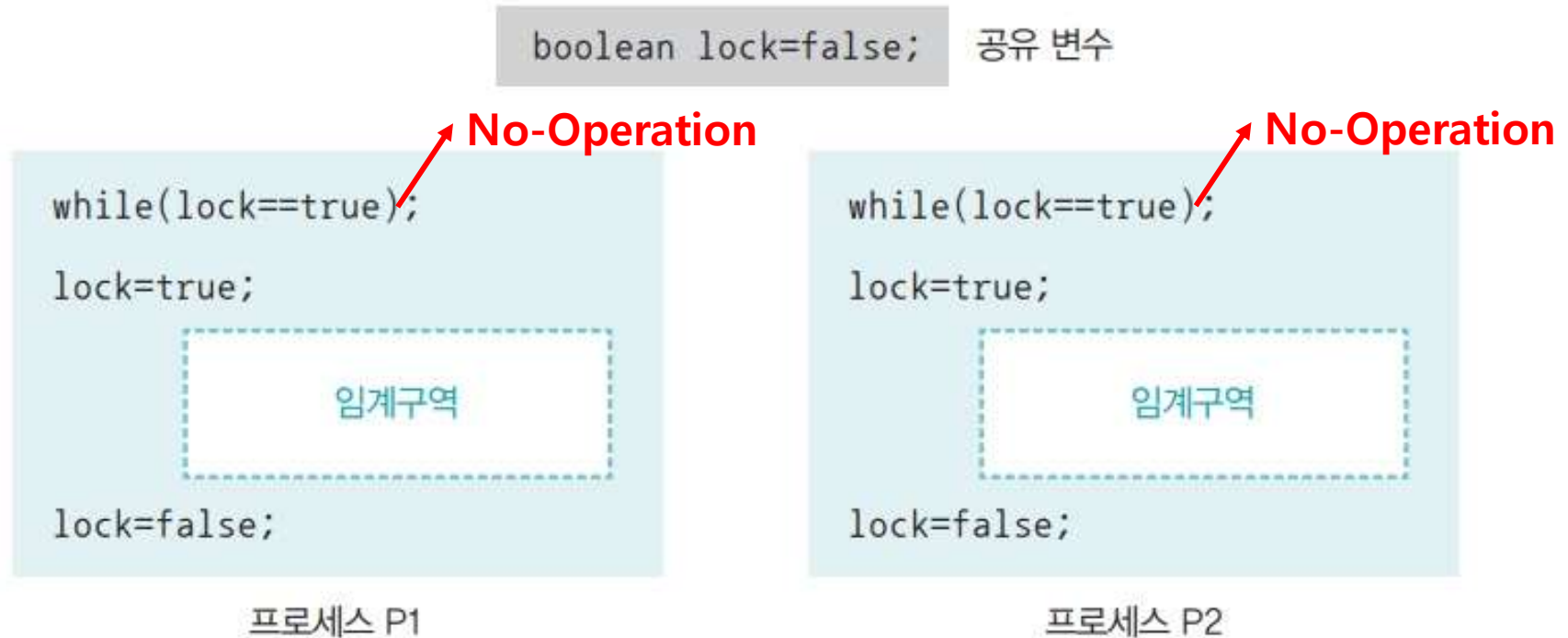


그림 5-16 전역 변수로 잠금을 구현한 코드

2 임계구역 해결 조건을 고려한 코드 설계

■ 전역 변수로 잠금을 구현한 코드의 문제

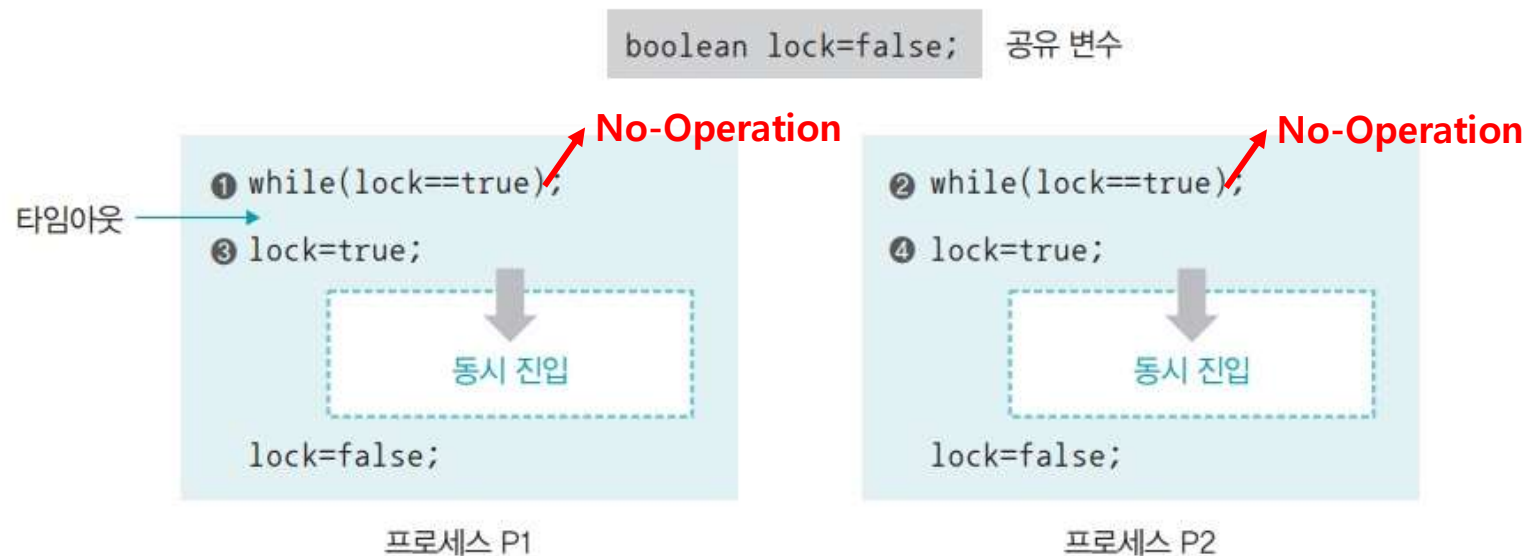


그림 5-17 동시 진입 상황(상호 배제 조건을 충족하지 않는 경우)

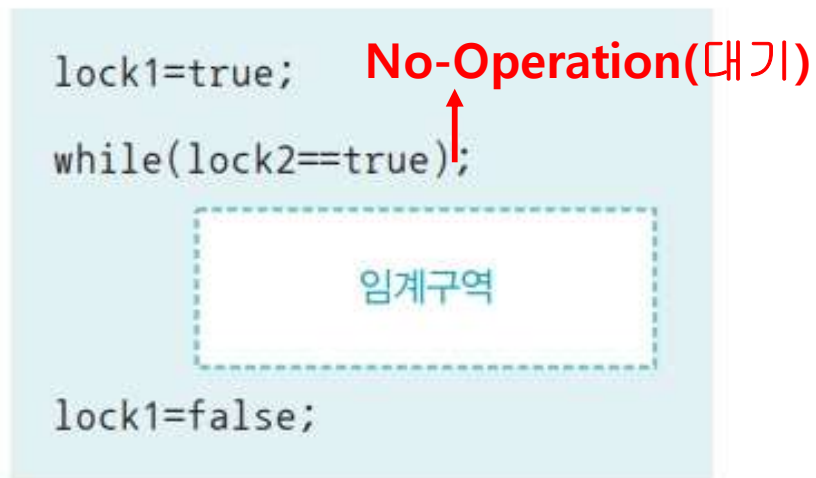
- ❶ 프로세스 P1은 while(lock==true); 문을 실행
- ❷ 프로세스 P2는 while(lock==true); 문을 실행
- ❸ 프로세스 P1은 lock=true; 문을 실행하여 임계구역에 잠금을 걸고 진입
- ❹ 프로세스 P2도 lock=true; 문을 실행하여 임계구역에 잠금을 걸고 진입(결국 둘 다 임계 구역에 진입)

2 임계구역 해결 조건을 고려한 코드 설계

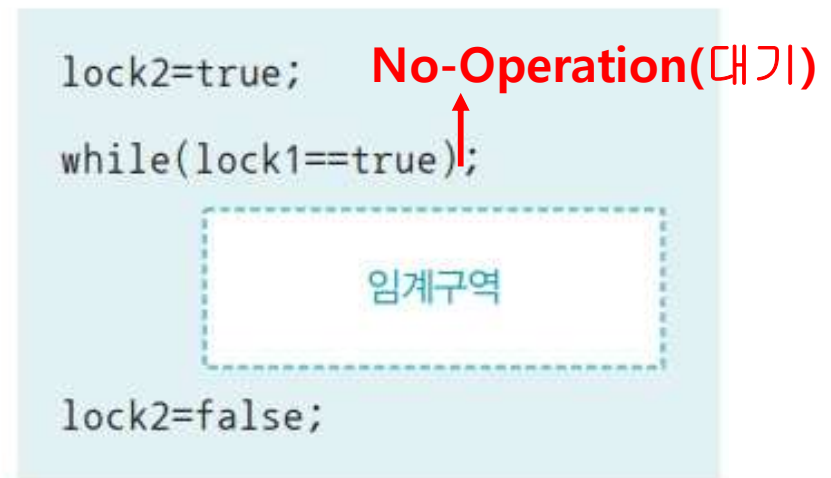
■ 상호 배제 조건을 충족하는 코드

```
boolean lock1=false;
boolean lock2=false;
```

공유 변수



프로세스 P1



프로세스 P2

그림 5-18 상호 배제 조건을 충족하는 코드

2 임계구역 해결 조건을 고려한 코드 설계

■ 상호 배제 조건을 충족하는 코드의 문제

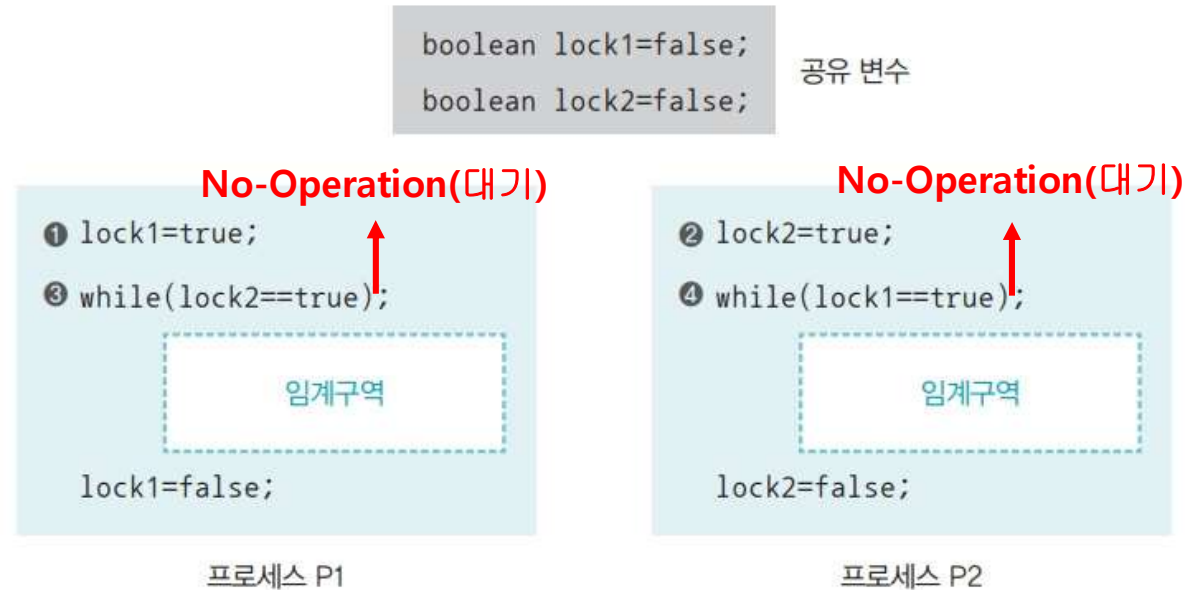


그림 5-19 무한 대기 상황(한정 대기 조건을 충족하지 않는 경우)

- 프로세스 P1은 lock1=true; 문을 실행한 후 자신의 CPU 시간을 다 씀(타임아웃) 문맥 교환이 발생하고 프로세스 P2가 실행 상태로 바뀜
- 프로세스 P2도 lock2=true; 문을 실행한 후 자신의 CPU 시간을 다 씀(타임아웃) 문맥 교환이 발생하고 프로세스 P1이 실행 상태로 바뀜
- 프로세스 P2가 lock2=true; 문을 실행했기 때문에 프로세스 P1은 while(lock2==true); 문에서 무한 루프에 빠짐
- 프로세스 P1이 lock1=true; 문을 실행했기 때문에 프로세스 P2도 while(lock1 ==true); 문에서 무한 루프에 빠짐

2 임계구역 해결 조건을 고려한 코드 설계

■ 상호 배제와 한정 대기 조건을 충족하는 코드

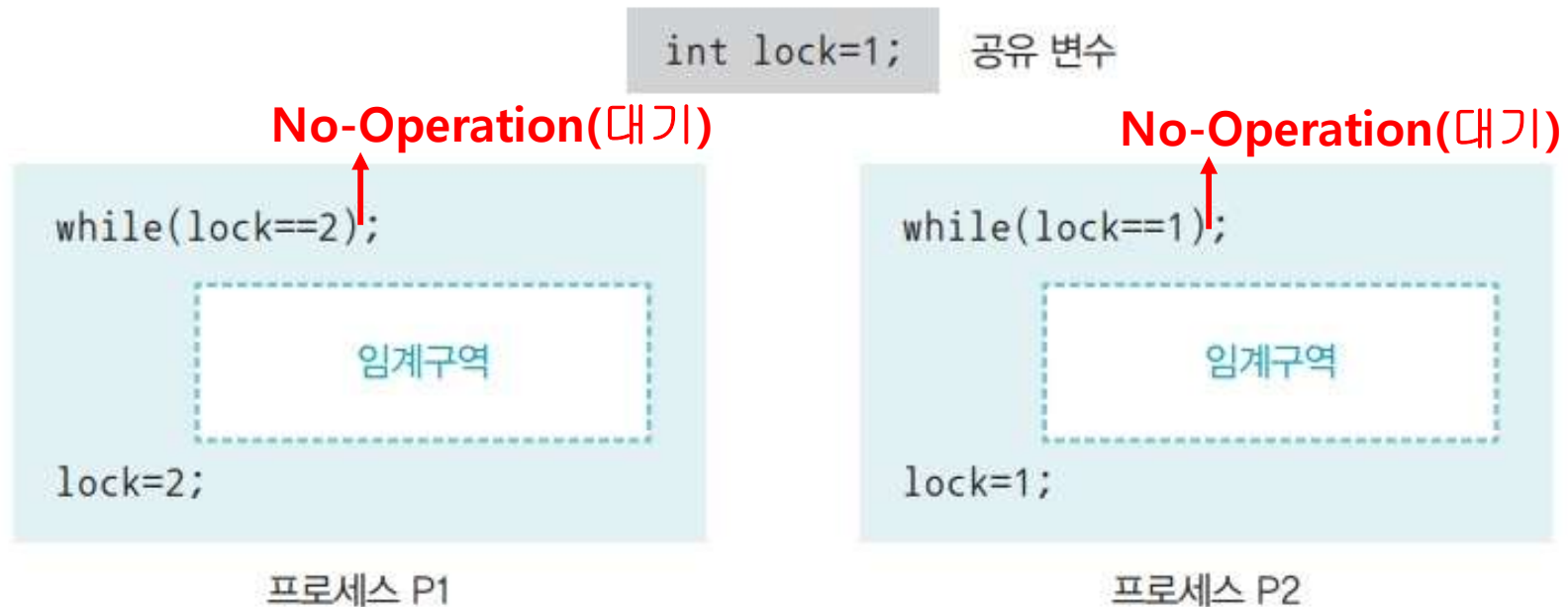


그림 5-20 상호 배제와 한정 대기 조건을 충족하는 코드(진행의 융통성 조건을 충족하지 않는 경우)

2 임계구역 해결 조건을 고려한 코드 설계

■ 임계구역 문제의 하드웨어적인 해결 방법

- 검사와 지정 test-and-set 코드로 하드웨어의 지원을 받아 while(lock==true); 문과 lock=true; 문을 한꺼번에 실행
- 검사와 지정 코드를 이용하면 명령어 실행 중간에 타임아웃이 걸려 임계구역을 보호하지 못하는 문제가 발생하지 않음

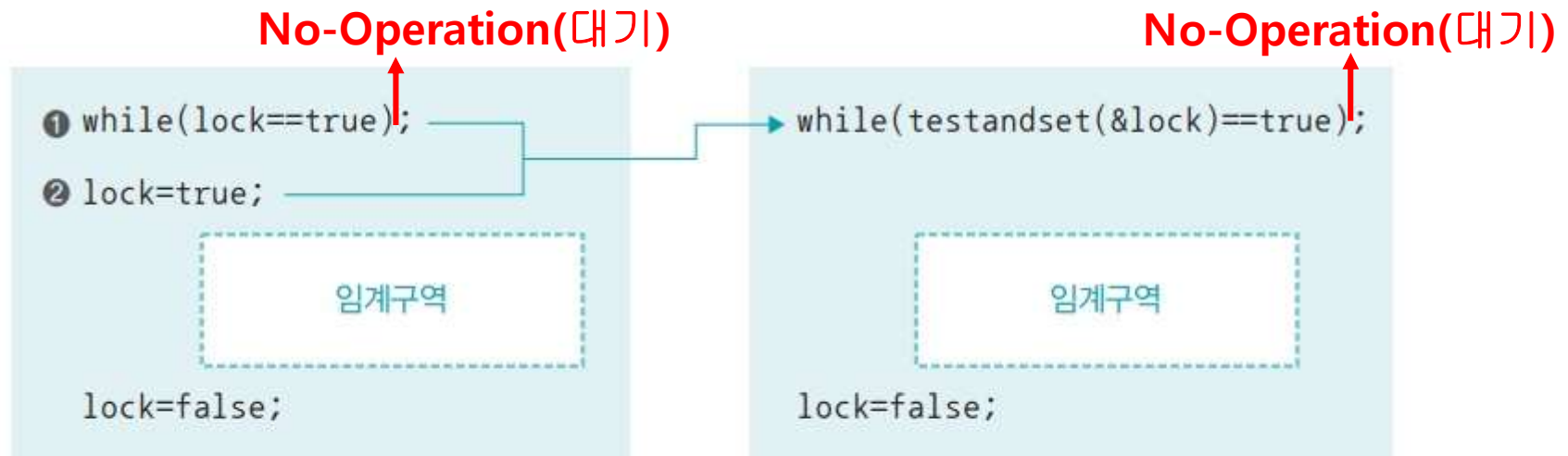


그림 5-21 검사와 지정을 이용한 코드

3 피터슨 알고리즘

■ 피터슨 알고리즘

- 임계구역 해결의 세 가지 조건을 모두 만족
- 2개의 프로세스만 사용 가능하다는 한계가 있음

```
boolean lock1=false;
boolean lock2=false;
int turn=1;
```

공유 변수

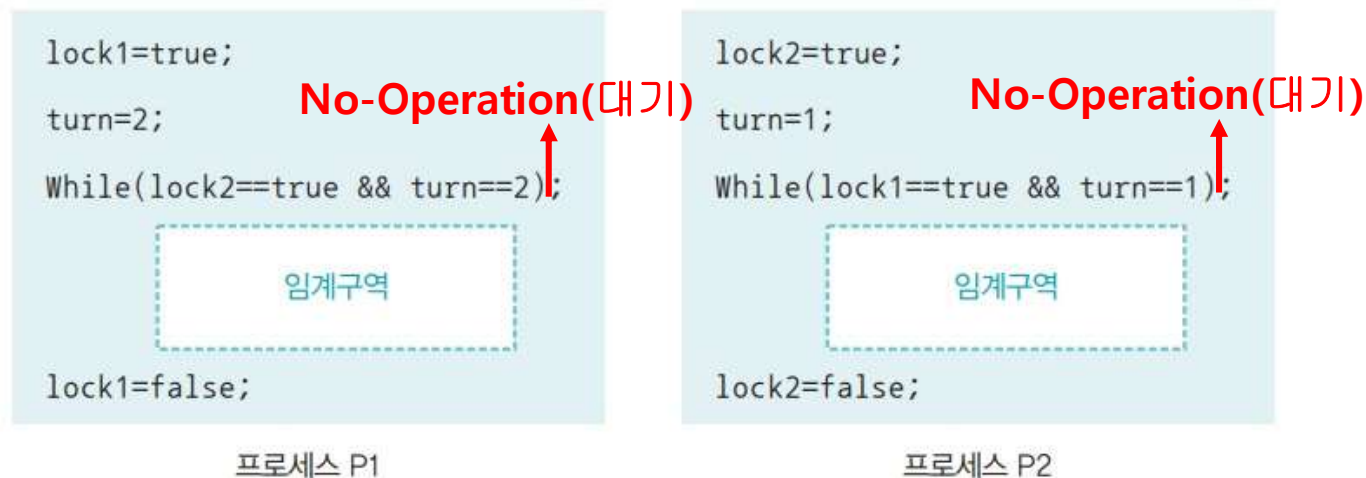


그림 5-22 피터슨 알고리즘

4 데커 알고리즘

데커 알고리즘

```
boolean lock1=false;
boolean lock2=false;
int turn=1;
```

공유 변수

```
lock1=true;
while(lock2==true)
{ if(turn==2) {
    lock1=false;
    while(turn==2);
    lock1=true; } /* end if */
} /* end while */



임계구역



turn=2;
lock1=false;
```

프로세스 P1

```
lock2=true;
while(lock1==true)
{ if(turn==1) {
    lock2=false;
    while(turn==1);
    lock2=true; } /* end if */
} /* end while */



임계구역



turn=1;
lock2=false;
```

프로세스 P2

그림 5-23 데커 알고리즘

4 데커 알고리즘

■ 데커 알고리즘의 동작

- ❶ 프로세스 P1은 우선 잠금을 걸(lock1=true;)
- ❷ 프로세스 P2의 잠금이 걸렸는지 확인[while(lock2==true)]
- ❸ 만약 프로세스 P2도 잠금을 걸었다면 누가 먼저인지 확인[if(turn ==2)]
만약 프로세스 P1의 차례라면(turn =1) 임계구역으로 진입
만약 프로세스 P2의 차례라면(turn =2) ❹로 이동
- ❹ 프로세스 P1은 잠금을 풀고(lock1 =false;) 프로세스 P2가 작업을 마칠 때까지 기다림
[while(turn ==2);]
프로세스 P2가 작업을 마치면 잠금을 걸고(lock1 =true;) 임계구역으로 진입

5 세마포어

■ 세마포어

- 임계구역에 진입하기 전에 스위치를 사용 중으로 놓고 임계구역으로 들어감
- 이후에 도착하는 프로세스는 앞의 프로세스가 작업을 마칠 때까지 기다림
- 프로세스가 작업을 마치면 다음 프로세스에 임계구역을 사용하라는 동기화 신호를 보냄



그림 4-23 세마포 예: 열차 차단기

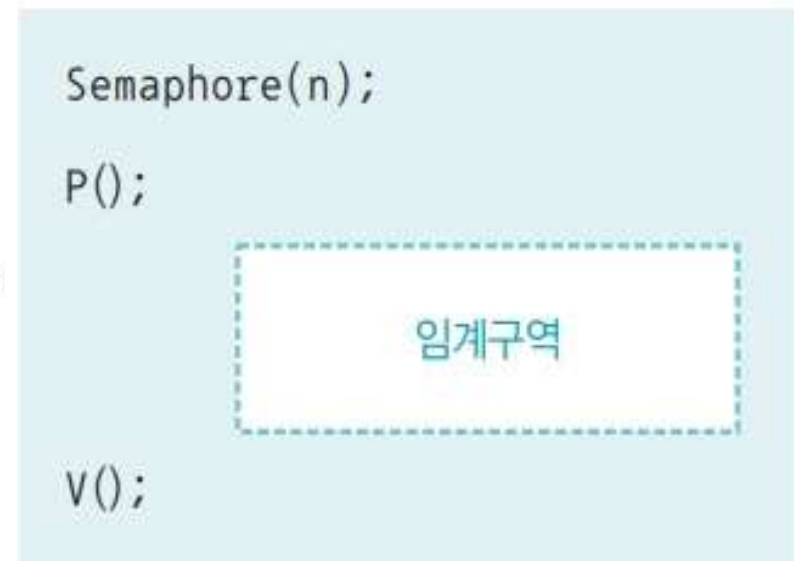


그림 5-25 세마포어 코드

5 세마포어

■ 세마포어 내부 코드

- Semaphore(n) : 전역 변수 RS를 n으로 초기화, RS에는 현재 사용 가능한 자원의 수가 저장
 - 네덜란드어로 P는 검사 **Proberen**, V는 증가 **Verhogen**
- P() : 잠금을 수행하는 코드로 RS가 0보다 크면(사용 가능한 자원이 있으면) 1만큼 감소시키고 임계구역에 진입, 만약 RS가 0보다 작으면(사용 가능한 자원이 없으면) 0보다 커질 때까지 기다림
- V() : 잠금 해제와 동기화를 같이 수행하는 코드로, RS 값을 1 증가시키고 세마포어에서 기다리는 프로세스에게 임계구역에 진입해도 좋다는 wake_up 신호를 보냄

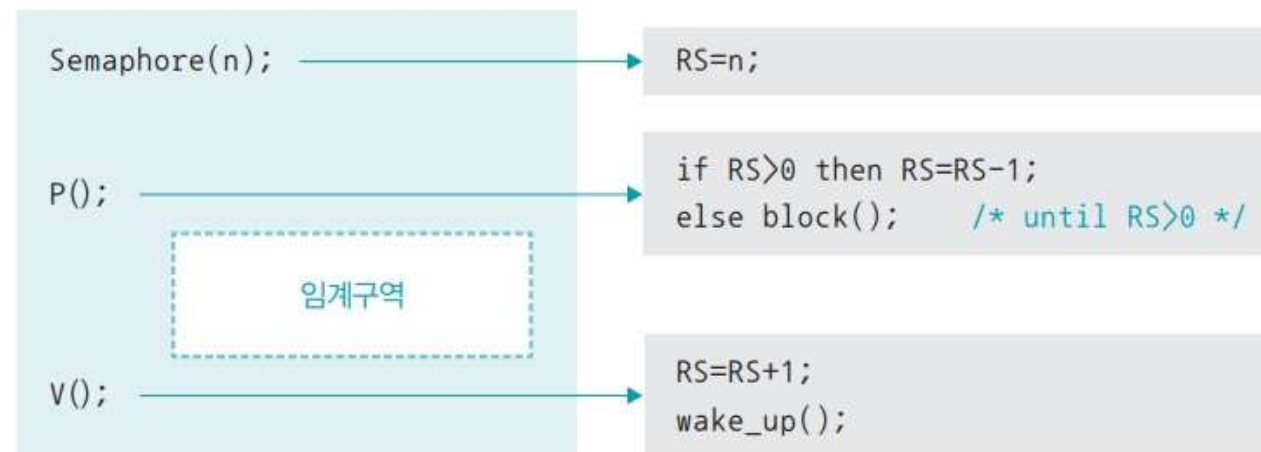


그림 5-26 세마포어 내부 코드

5 세마포어

■ 예금 5만 원이 사라진 문제를 세마포어를 사용하여 해결한 코드

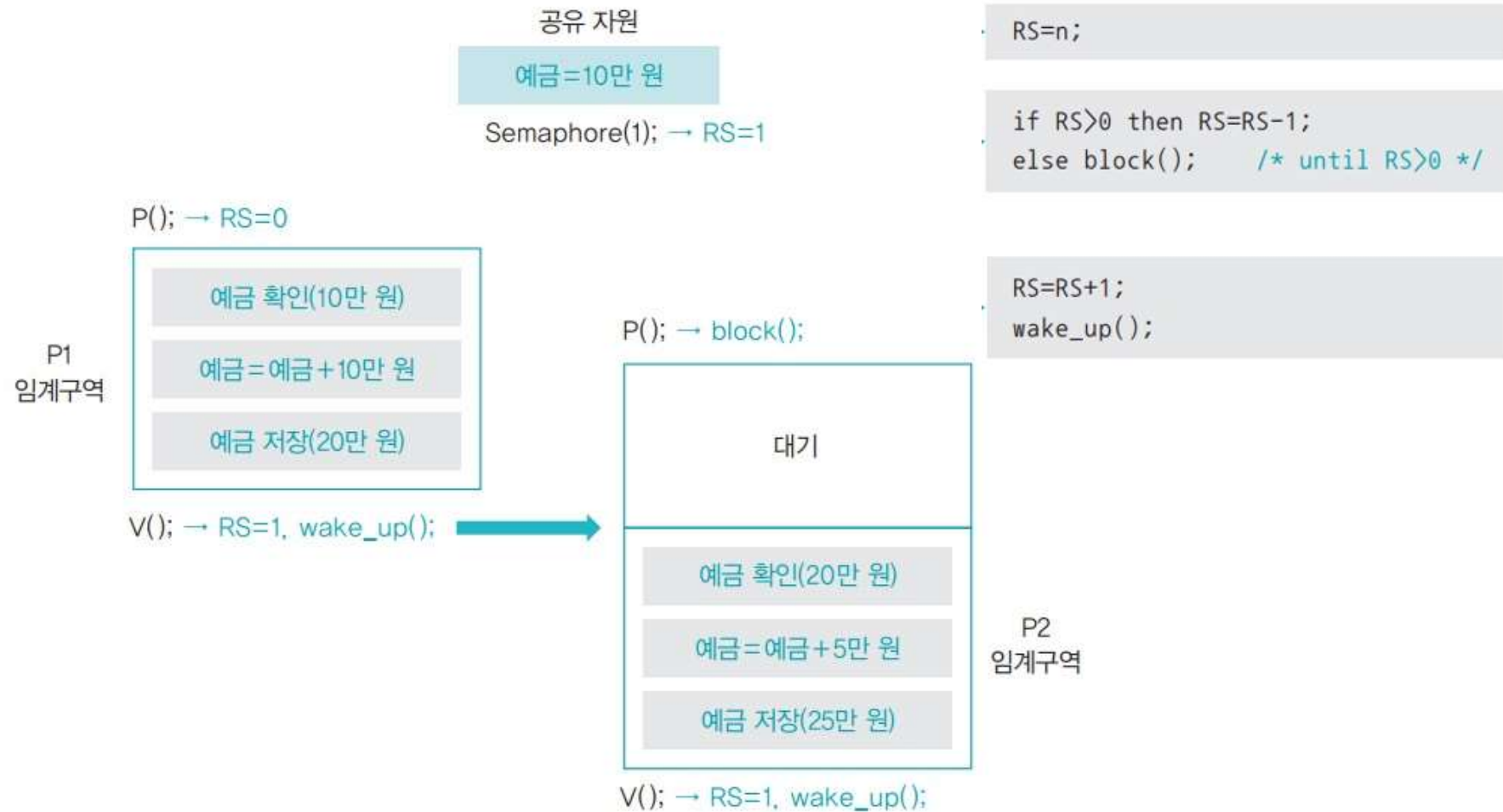


그림 5-27 세마포어 사용 예

5 세마포어

공유 자원이 여러 개일 때 세마포어 사용 예

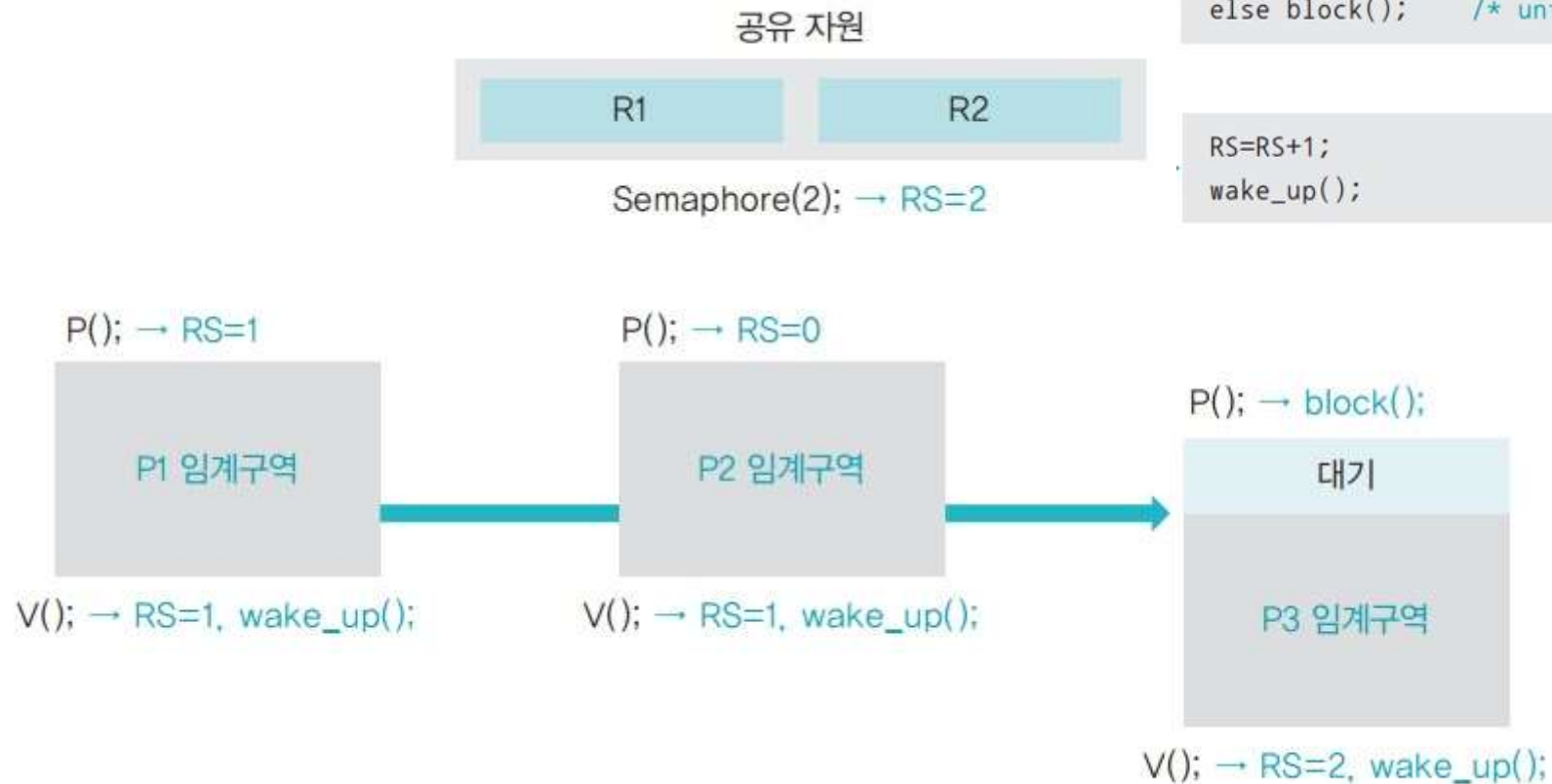


그림 5-28 공유 자원이 여러 개일 때의 세마포어 사용 예

6 모니터

■ 세마포어의 잘못된 사용 예

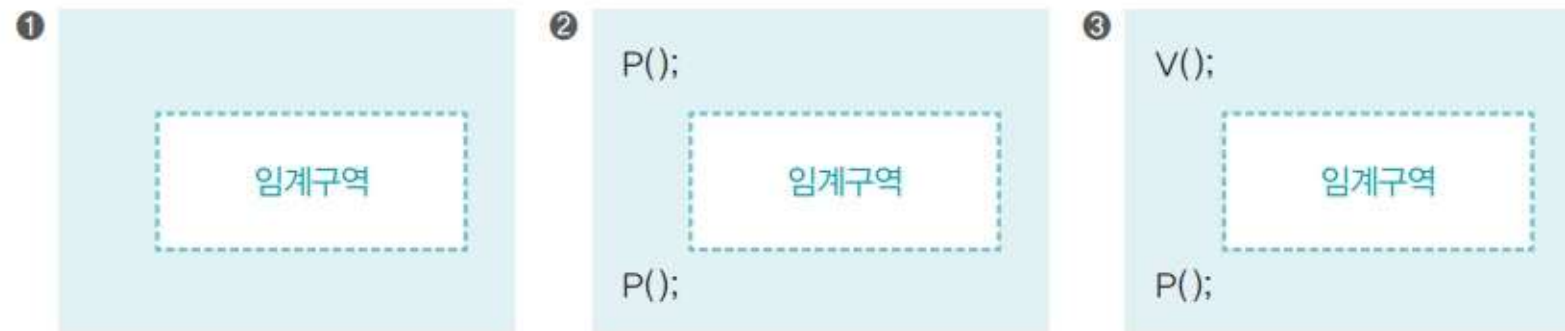


그림 5-29 세마포어의 잘못된 사용 예

- ❶ 프로세스가 세마포어를 사용하지 않고 바로 임계구역에 들어간 경우로 임계구역을 보호할 수 없음
- ❷ `P()`를 두 번 사용하여 `wake_up` 신호가 발생하지 않은 경우로 프로세스 간의 동기화가 이루어지지 않아 세마포어 큐에서 대기하고 있는 프로세스들이 무한 대기에 빠짐
- ❸ `P()`와 `V()`를 반대로 사용하여 상호 배제가 보장되지 않은 경우로 임계구역을 보호할 수 없음

6 모니터

■ 모니터 monitor

- 공유 자원을 내부적으로 숨기고 공유 자원에 접근하기 위한 인터페이스만 제공함으로써 자원을 보호하고 프로세스 간에 동기화를 시킴

■ 모니터의 작동 원리

- ① 임계구역으로 지정된 변수나 자원에 접근하고자 하는 프로세스는 직접 $P()$ 나 $V()$ 를 사용하지 않고 모니터에 작업 요청
- ② 모니터는 요청받은 작업을 모니터 큐에 저장한 후 순서대로 처리하고 그 결과만 해당 프로세스에 알려줌

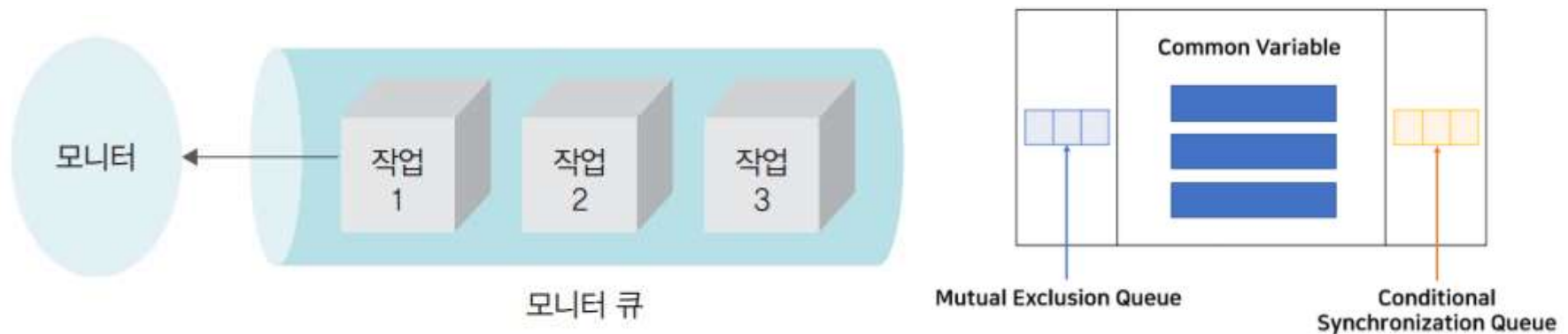


그림 5-30 모니터의 작동 원리

6 모니터

- 예금 5만 원이 사라진 문제를 모니터를 사용하여 해결한 코드

P1 increase(10); P2 increase(5);

그림 5-31 모니터 사용법

- 자바로 작성한 모니터 내부 코드

```
monitor shared_balance {
    private:
        int balance=10;           /* shared data */
        boolean busy=false;
        condition mon;           /* condition variable */

    public:
        increase(int amount) {    /* public interface */
            if(busy==true) mon.wait(); /* waiting in queue */
            busy=true;
            balance=balance+amount;
            mon.signal();         /* wake up next waiting process */
        }
}
```

그림 5-32 자바로 작성한 모니터 내부 코드

Homework

■ 5장 연습문제

- P. 연습문제(단답형)
- P. 심화문제(서술형)
- 제출기한 : 11월 1일(일) 23시
- 제출방법 : hwp/word로 작성한 후 pdf 변환 후 과제함 제출
- 파일이름 : **학번(이름)-5장.pdf**



Thank You
