스마트시스템 운영체제
(LD01600)

김준철
정보시스템공학과
greensday@sungshin.ac.kr

6주차 강의

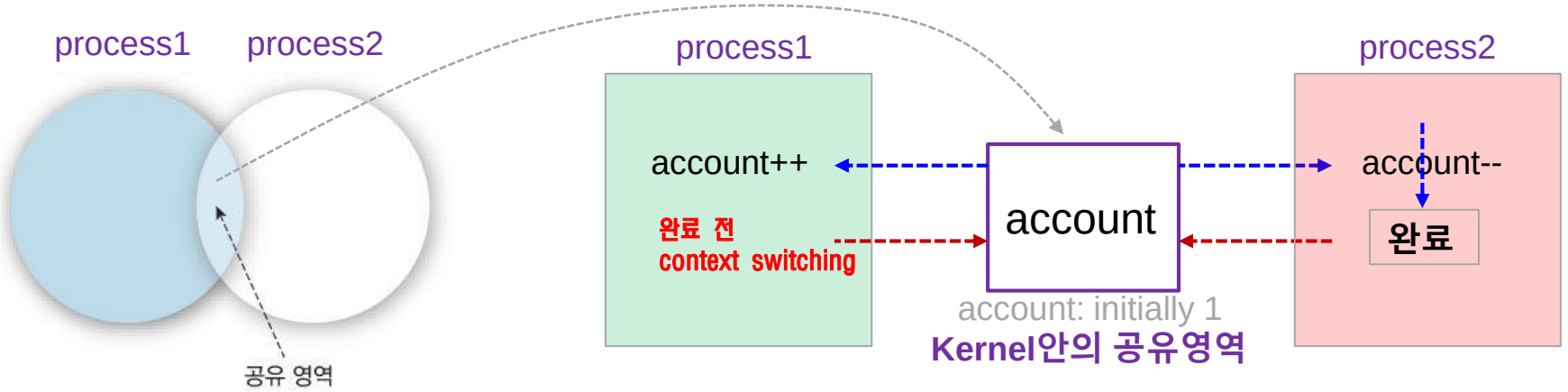
휴강(9.30)
(추석)

	주차	강의 목차
9.2	1	과목소개 / 운영체제 개요
9.9	2	컴퓨터 시스템 구조
9.16	3	process와 스레드1
9.22	4	process와 스레드2, CPU스케쥴링1
10.7	5	CPU스케쥴링2
10.14	6	process 동기화
10.21	7	교착 상태
10.28	8	중간고사
11.4	9	물리 메모리 관리
11.11	10	가상메모리 기초
11.18	11	가상메모리 관리
11.25	12	입출력시스템1
12.2	13	입출력시스템2, 파일시스템1
12.9	14	파일시스템2
12.16	15	기말고사

ch.05 Process Synchronization

공유 자원(shared resource)의 접근

- **공유 자원(shared resource)**
  - 여러 thread(process)가 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 thread(process)가 공유 자원을 동시에 접근(병행적으로 읽거나 쓰는) 하는 상황
  - Process들 끼리는 data를 공유하지 않아도, system call시 OS내부 data를 수정하는 과정에서 문제 발생 가능
  - 경쟁 조건이 발생하면 공유 자원 접근 순서에 따라 실행 결과가 달라질 수 있음



[그림] race condition

Concurrency Problem

dependency가 있는 여러 thread가 실행 될 때, 각 thread를 어떻게 control 하느냐에 따라 결과가 달라지는 문제

- **Independent threads / cooperating threads**
  - Independent threads: thread의 state(상태)가 다른 thread들과 공유되지 않음
  - Cooperating threads: thread의 state(상태)가 다른 thread들과 공유됨

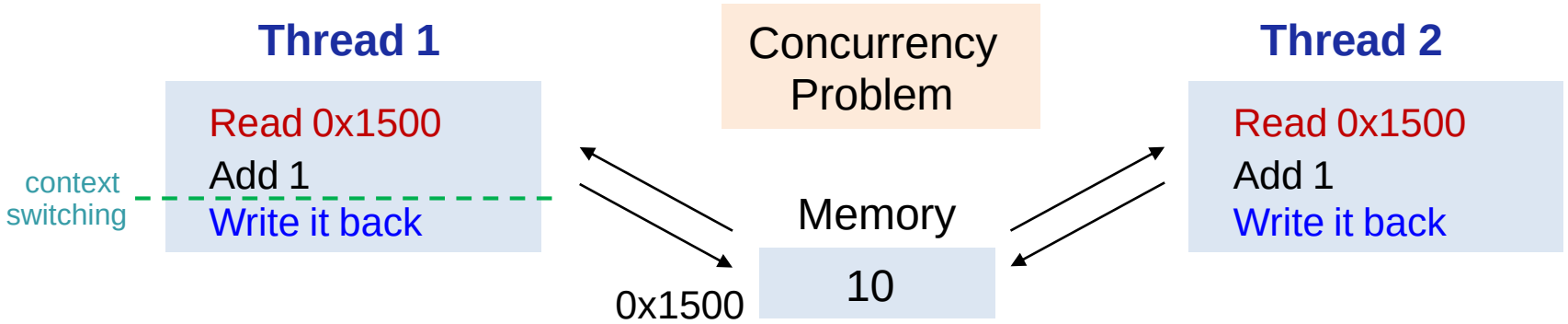
Synchronization 필요

- **Synchronization(동기화) :**
  - thread간 cooperation을 문제없이 진행하게 하는 것 (atomic operation을 사용)

- **Critical section**
  - 한 개의 thread만 실행할 수 있는 code section
- **Mutual exclusion**
  - 한 process가 critical section에 들어가면 다른 process는 critical section에 들어갈 수 없는 것

Atomic Operations

- **Atomic Operations**: An operation that always runs to completion
  - Atomic operation은 실행 중간에 멈출 수 없음(끝까지 실행)
  - 실행 중간에 state(상태)를 수정할 수 없음
- **Thread들의 동시 작업** : atomic operations을 통해서 가능
- **Examples**
  - Memory접근명령(load/store)과 값의 연산명령의 실행은 atomic operation



Critical Section(임계 구역)

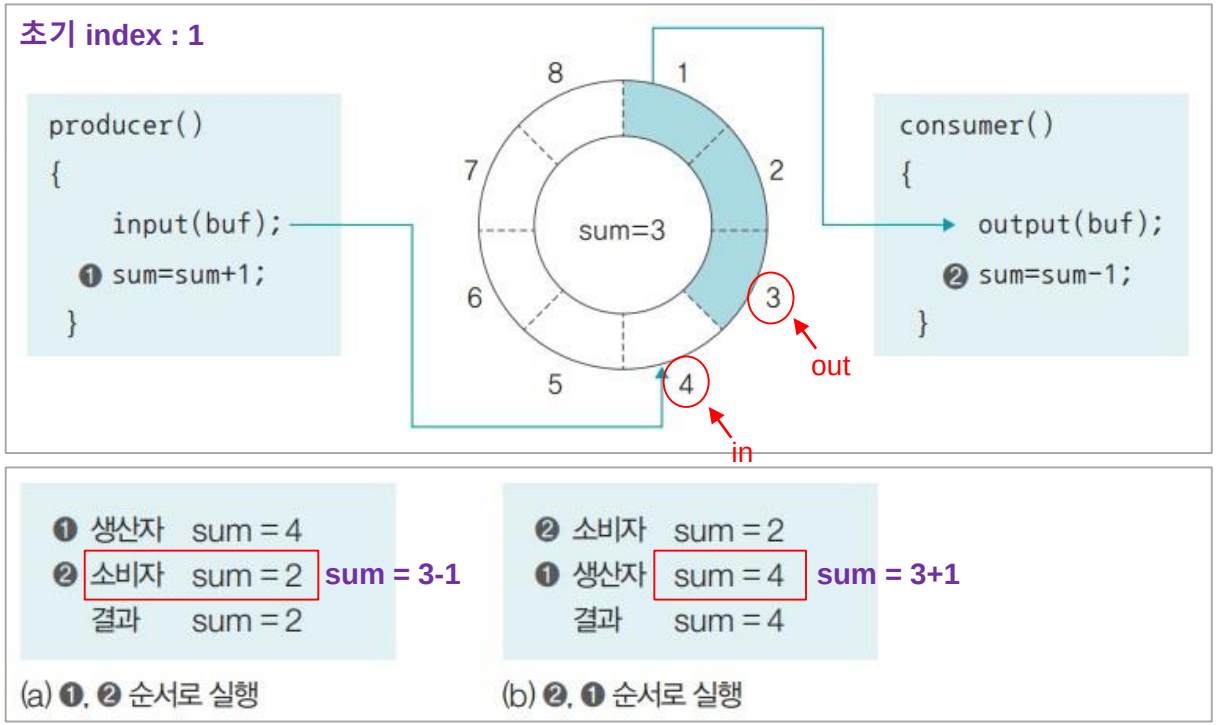
- Critical Section
  - 공유 자원 접근 순서에 따라 실행 결과가 달라지는 프로그램 부분
    - 공통 자원(변수)를 update하는 프로그램 부분
  - Critical Section에서는 process들이 **동시에 작업하면 안 됨**
  - 어떤 process가 Critical Section에 들어가면 다른 process는 Critical Section 밖에서 기다려야 하며 Critical Section의 process가 나와야 들어갈 수 있음

믹서는 공유가 불가능한 자원으로서 주방의 Critical Section

[그림] Critical Section에 대한 비유 설명 예 (가스레인지와 믹서)

example 생산자-소비자 문제(producer-consumer problem)

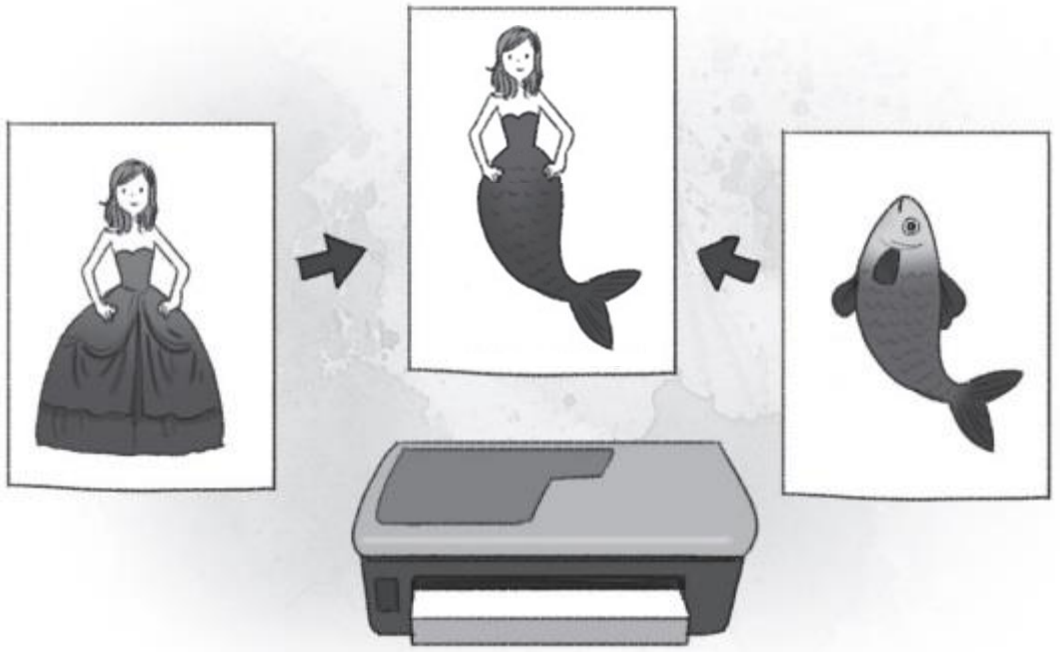
- code 및 실행 순서에 따른 결과
  - 생산자는 수를 증가시켜가며 물건을 채우고 소비자는 생산자를 쫓아가며 물건을 소비
  - 생산자 code와 소비자 code가 동시에 실행되면 문제가 발생



[그림] 생산자-소비자 문제 : 실행 순서에 따른 결과 차이 예

example 하드웨어 자원을 공유

- 하드웨어 자원을 공유에 대한 (극단적인) 문제 발생의 예



[그림] 하드웨어 자원을 공유하면 발생하는 문제

Critical Section 기본 code

- Critical Section 해결 방법을 설명하기 위한 기본 code

```
#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;
}
```

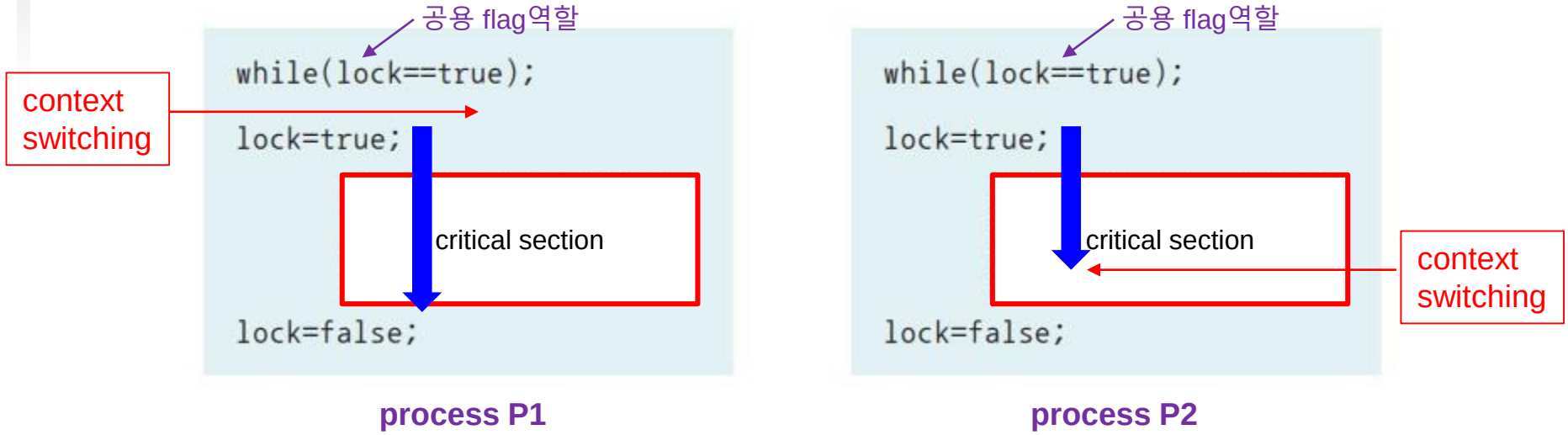
[그림] Critical Section 해결 방법을 설명하기 위한 기본 code

Critical Section 해결 조건을 고려한 코드 설계

- 알고리즘 1

mutual exclusion(mutex) 문제

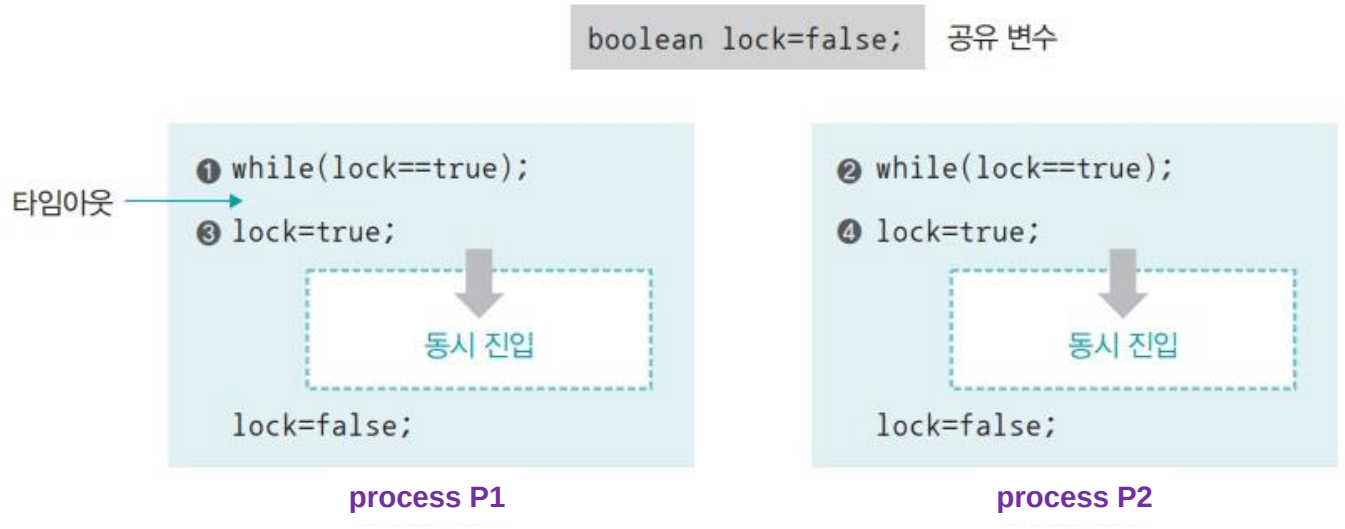
```
boolean lock=false;
```
공유 변수



[그림] 전역 변수로 잠금을 구현한 code

Critical Section 해결 조건을 고려한 코드 설계

- 알고리즘 1



[그림] 동시 진입 상황 (상호 배제 조건을 충족하지 않은 경우)

❶ process P1은 while(lock==true); 문을 실행

❷ process P2는 while(lock==true); 문을 실행

❸ process P1은 lock=true; 문을 실행하여 Critical Section에 잠금을 걸고 진입

❹ process P2도 lock=true; 문을 실행하여 Critical Section에 잠금을 걸고 진입
(결국 둘 다 Critical Section에 진입)

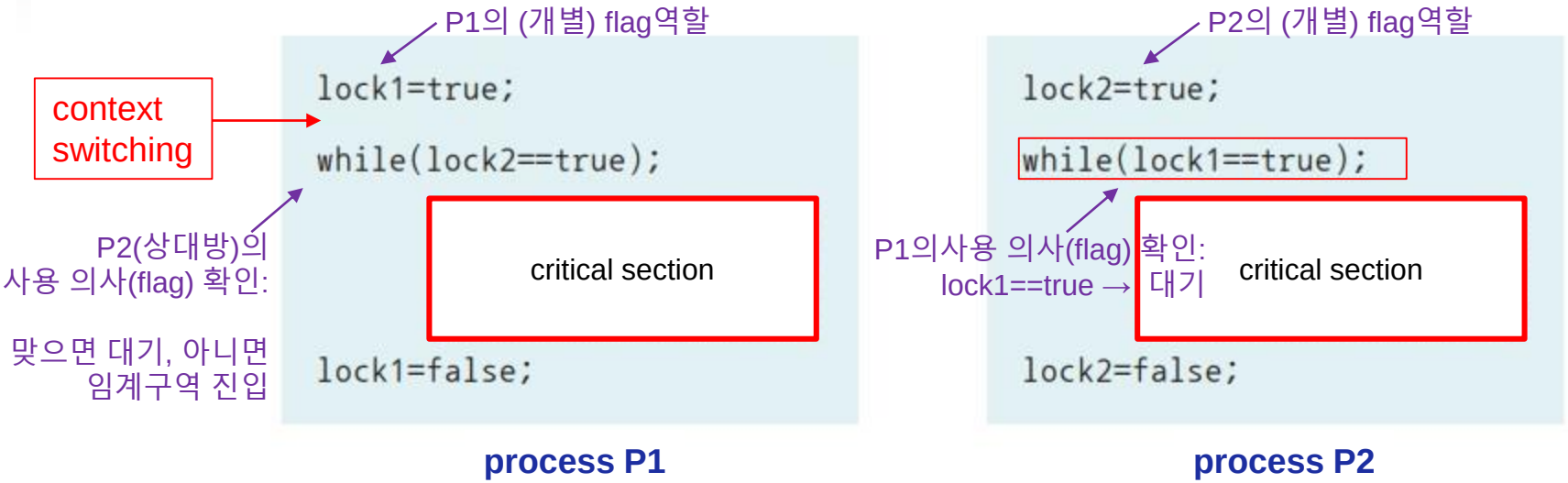
Critical Section 해결 조건을 고려한 코드 설계

- 알고리즘 2

bounded waiting 조건 위반

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

공유 변수



[그림] 상호 배제 조건을 충족하는 code

Critical Section 해결 조건을 고려한 코드 설계

- 알고리즘 2

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

공유 변수

[그림] 무한 대기 상황(한정 대기 조건을 충족하지 않음)

❶ process P1은 lock1=true; 문을 실행한 후 자신의 CPU 시간을 다 씀(타임아웃) 문맥 교환이 발생하고 process P2가 실행 상태로 바뀜

❷ process P2도 lock2=true; 문을 실행한 후 자신의 CPU 시간을 다 씀(타임아웃) 문맥 교환이 발생하고 process P1이 실행 상태로 바뀜

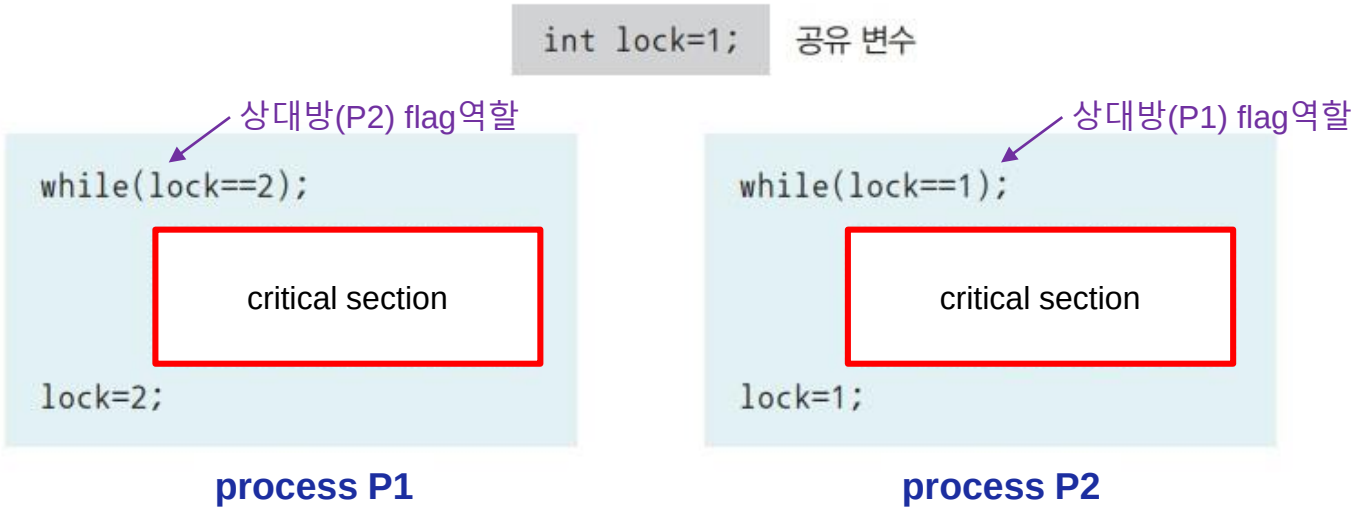
❸ process P2가 lock2=true; 문을 실행했기 때문에 process P1은 while(lock2==true); 문에서 무한 루프에 빠짐

❹ process P1이 lock1=true; 문을 실행했기 때문에 process P2도 while(lock1==true); 문에서 무한 루프에 빠짐

Critical Section 해결 조건을 고려한 코드 설계

- 알고리즘 3

progress flexibility 조건 위반



[그림] 상호 배제 조건과 한정 대기 조건을 충족하는 code (진행의 융통성 조건을 충족하지 않음)

Critical Section Problem 해결 조건

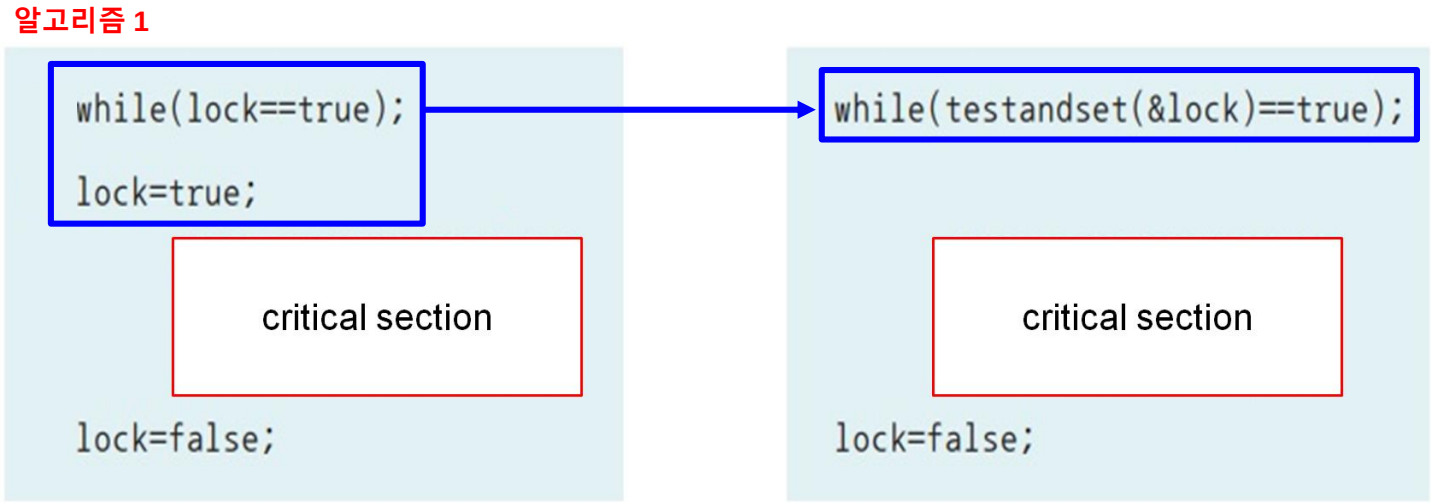
- **mutual exclusion(mutex)** 상호 배제
  - 한 process가 Critical Section에 들어가면 다른 process는 Critical Section에 들어갈 수 없는 것
- **bounded waiting** 한정 대기
  - 한 process가 **critical section을 계속 사용**하여 다른 process들이 critical section에 진입하지 못하는 것 (무한 대기 상태)
- **progress flexibility** 진행의 융통성
  - 한 process가 다른 process의 진행을 방해해서는 안 된다는 것

  (예) process 2개(A, B)가 **critical section을 무조건 번갈아 사용**해야 하는 경우

  → process A는 critical section을 사용 후 B가 사용하지 않으면, 영원히 critical section에 다시 진입하지 못함

Critical Section 해결 조건을 고려한 코드 설계

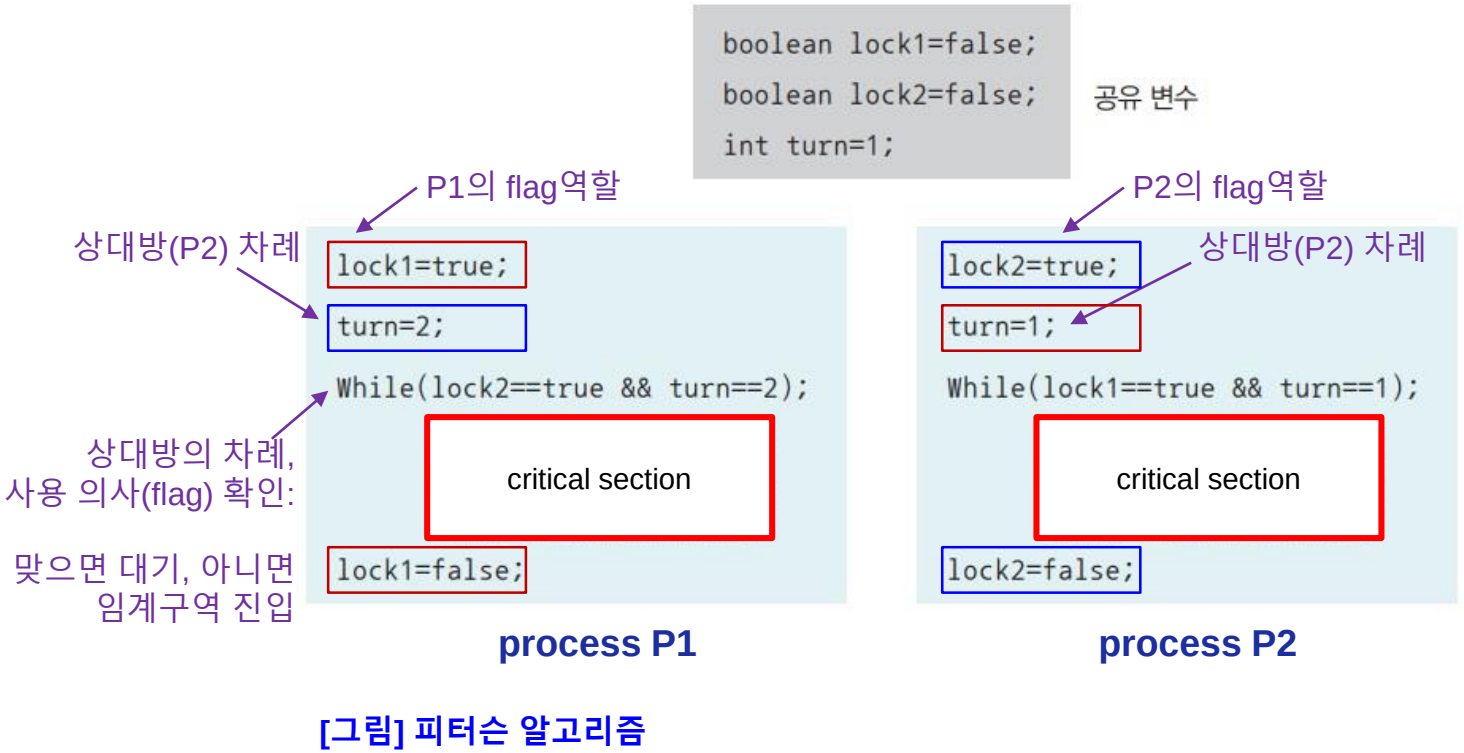
- **하드웨어 지원**
  - 검사와 지정(Test-And-Set) code로 하드웨어의 지원을 받아서 while(lock==true); 문과 lock=true; 문을 한꺼번에 실행
  - 검사와 지정 code를 이용하면 명령어 실행 중간에 타임아웃이 걸려 Critical Section을 보호하지 못하는 문제가 발생하지 않음



[그림] Test-And-Set 을 이용한 code

Peterson(피터슨) 알고리즘

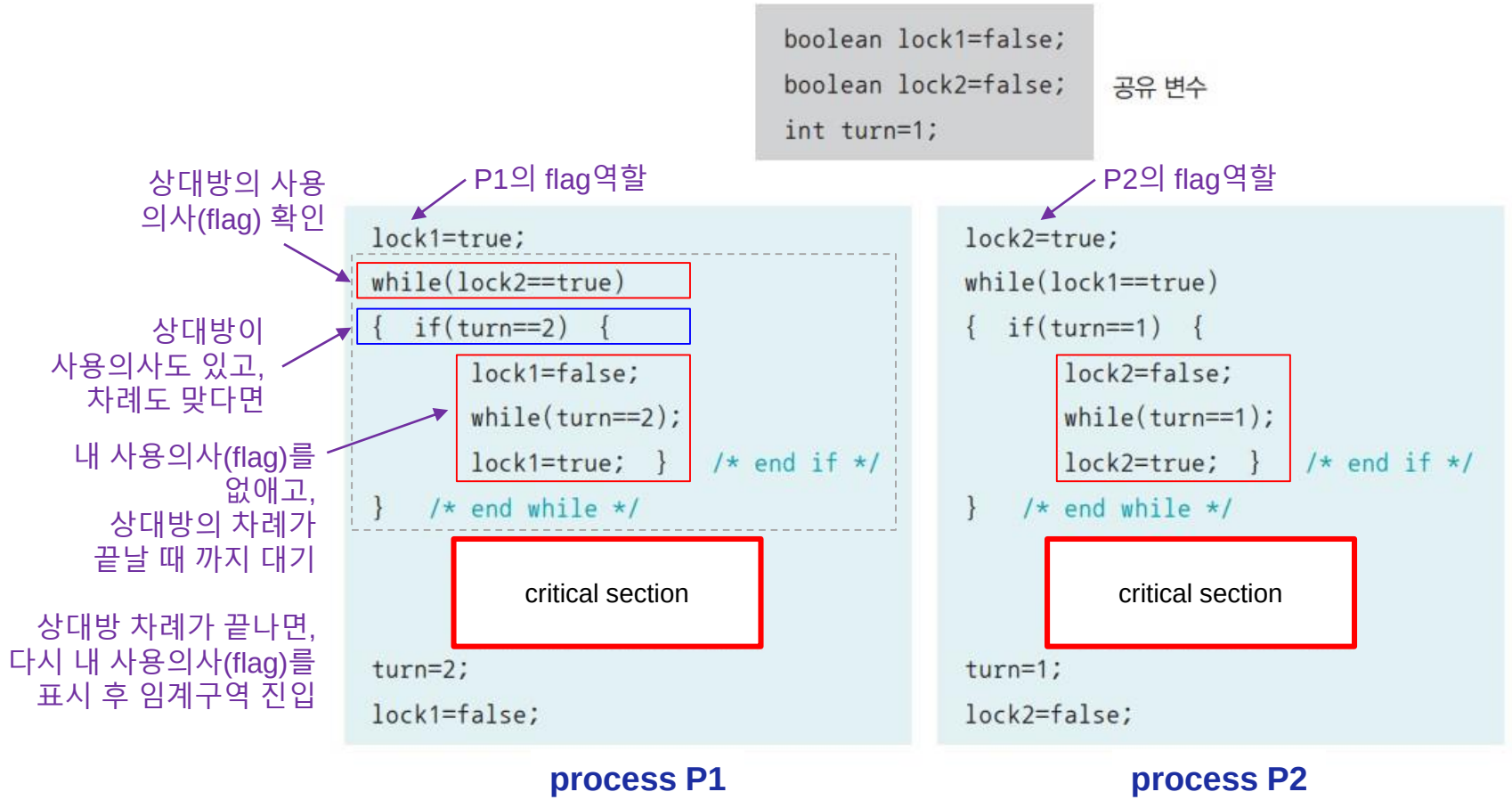
- **피터슨 알고리즘** - 공유변수 2가지(lock, turn)를 확인하여 critical section 진입을 가능여부를 판단
  - Critical Section 해결의 세 가지 조건을 모두 만족
  - 2개의 process만 사용 가능하다는 한계가 있음



[그림] 피터슨 알고리즘

Dekker(데커) 알고리즘

- **데커 알고리즘** - 공유변수 2가지(lock, turn)를 확인하여 critical section 진입을 가능여부를 판단



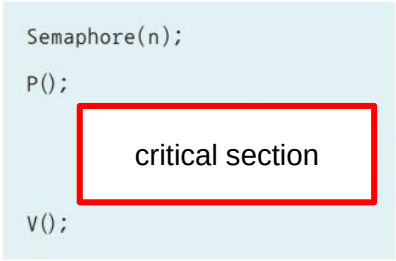
[그림] 데커 알고리즘

Busy-waiting 문제가 있음

Semaphore (세마포)

- **Semaphore**
  - Synchronization 문제에 대한 해결을 위해 만들어진 도구
  - Semaphore 변수 S는 표준 단위 연산 P와 연산 V로만 접근하는 정수 변수
    - **P()** : process를 critical section에 진입 또는 대기(wait)하게 하는 동작
    - **V()** : process를 critical section에서 나오는 연산,
      대기 중인 process 깨우려고 신호 보내는(signal) 동작
  - Semaphore 동작:
    - **Critical Section에 진입하기 전에 사용 중으로 표시**하고 Critical Section 진입
    - 이후에 도착하는 process는 **앞의 process가 작업을 마칠 때까지 대기**
    - 작업을 마치면 **다음 process에 Critical Section을 사용하라는 Synchronization 신호를 보냄**

[그림] Semaphore와 토글 스위치



[그림] Semaphore code

Semaphore (세마포)

Semaphore 내부 code

- **Semaphore(n)** : 전역 변수 S를 n으로 초기화, S에는 현재 **사용 가능한 자원의 수**가 저장

Atomic Operations

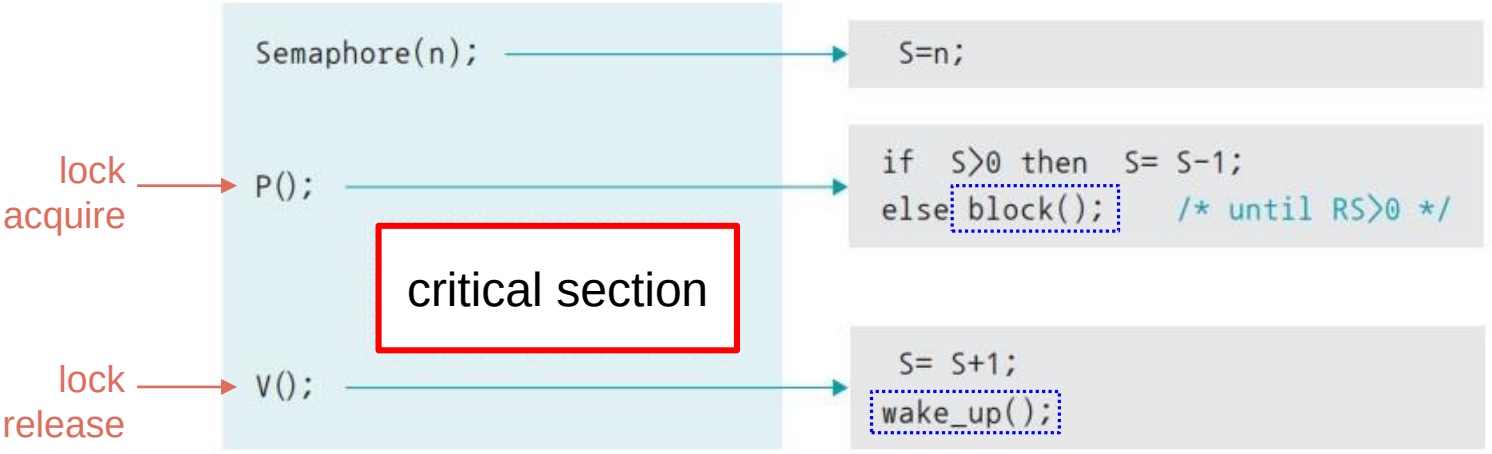
- **P()** : 잠금을 수행하는 code로 S가 0보다 크면(사용 가능한 자원이 있으면) **1만큼 감소** 시키고 Critical Section에 진입, 만약 S가 0보다 작으면(사용 가능한 자원이 없으면) 0보다 커질 때까지 기다림

acquire 동작

Atomic Operations

- **V()** : 잠금 해제와 Synchronization를 같이 수행하는 code로, S **값을 1 증가시키고** Semaphore에서 기다리는 process에게 Critical Section에 진입해도 좋다는 **wake_up 신호**를 보냄

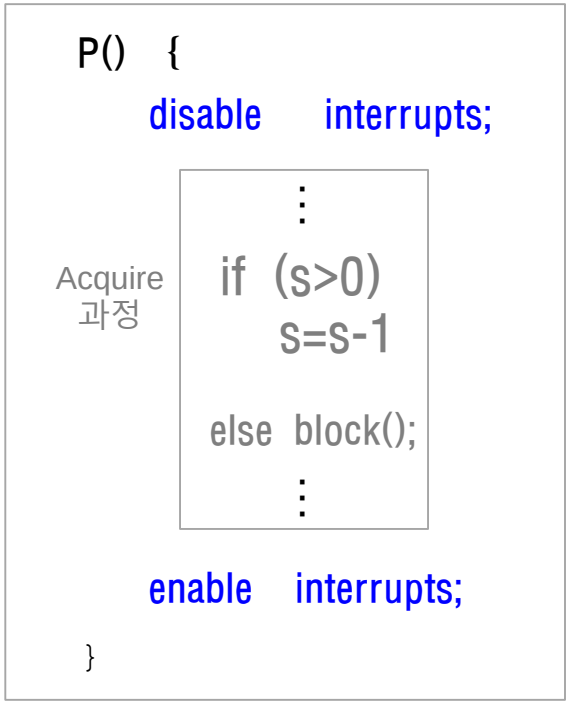
release 동작

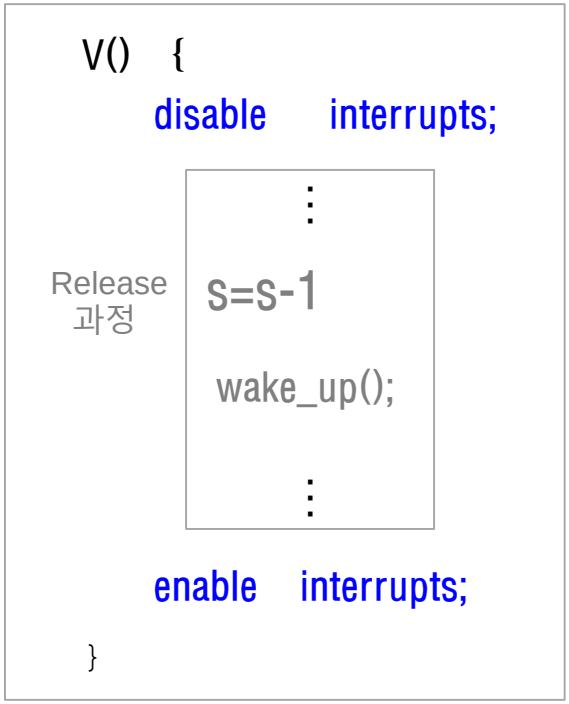


[그림] Semaphore 내부 code

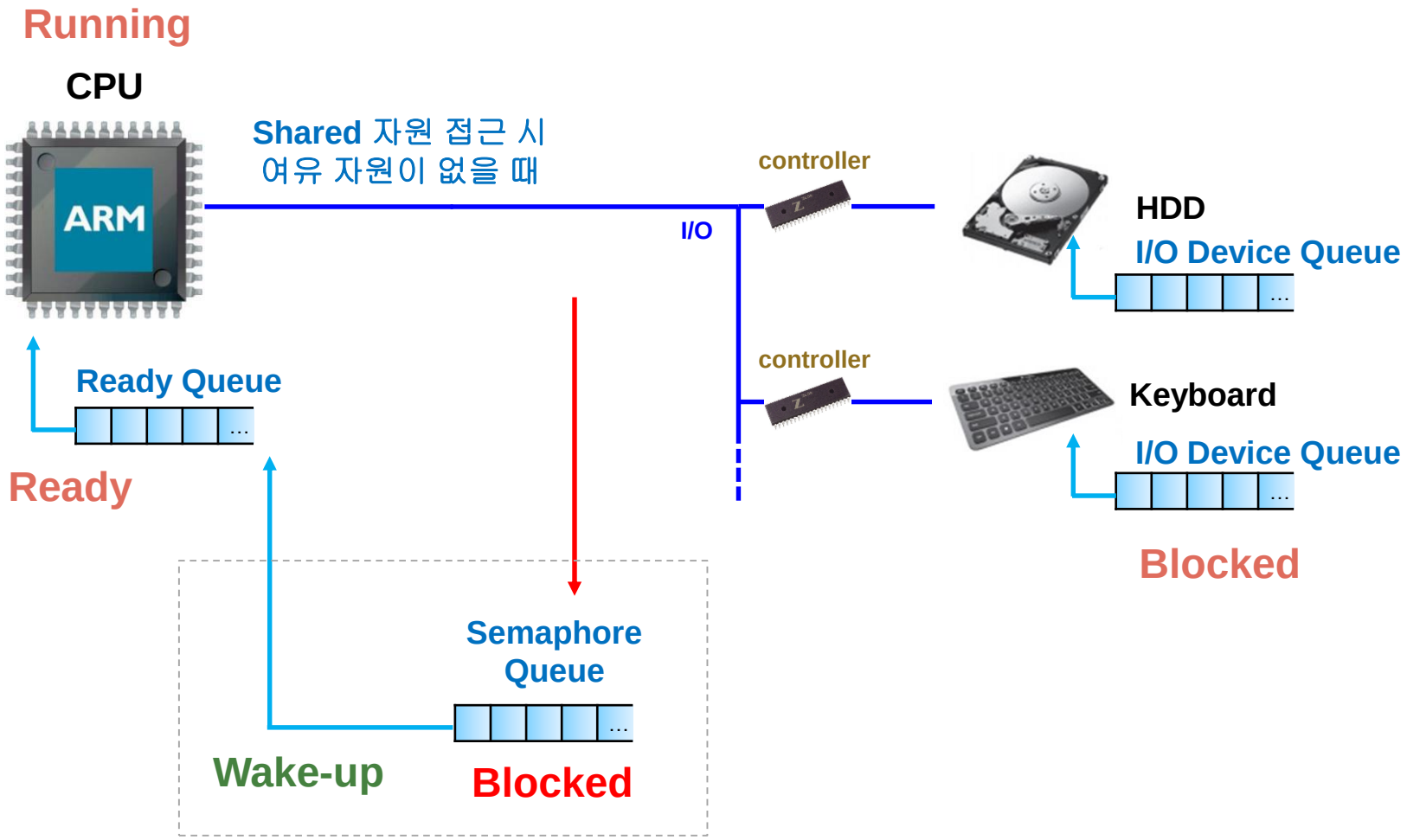
Acquire, Release

- Semaphore에서 synchronization 구현
  - Acquire, Release 과정은 atomic operation으로 구현해야 함





Resource Queue

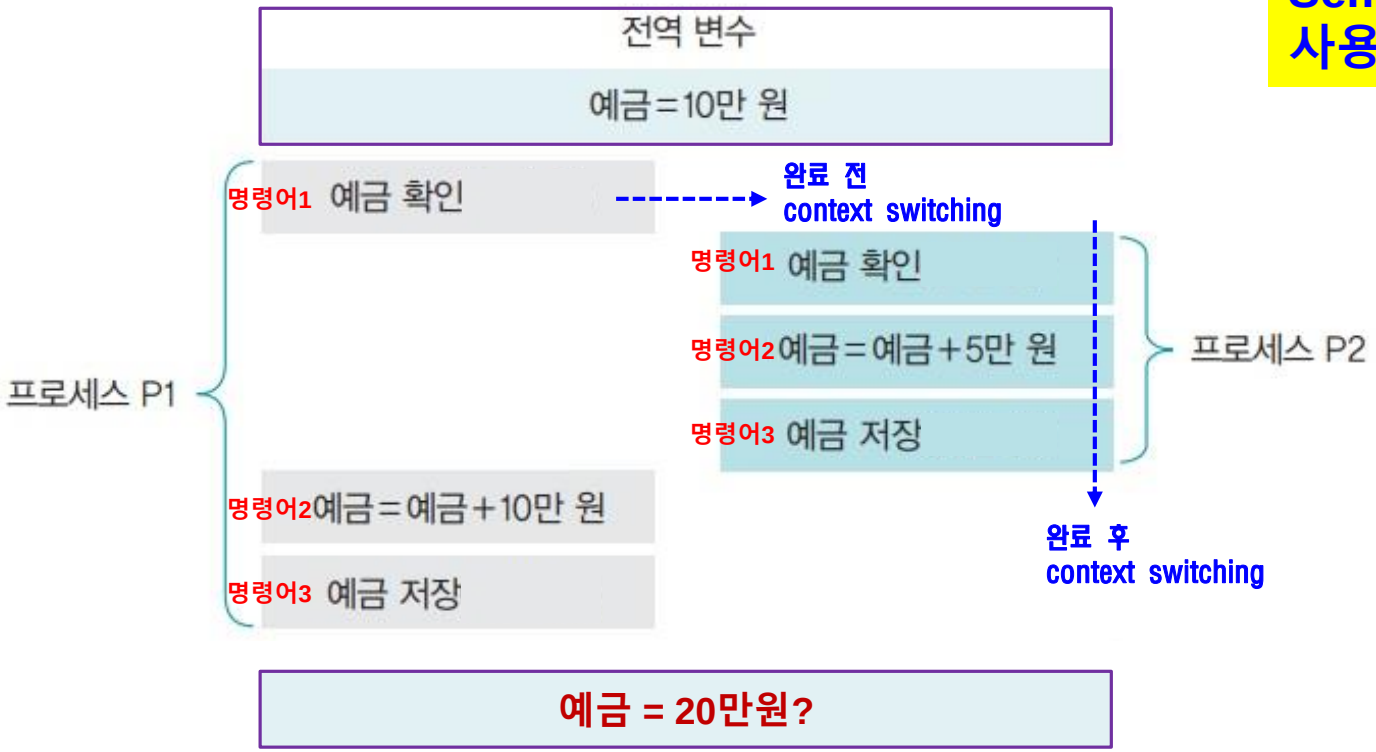


* 그림 상의 Queue는 실제 OS 내부(memory 내부)에 있음

example Semaphore (세마포) -1

- 공유 자원의 접근 예

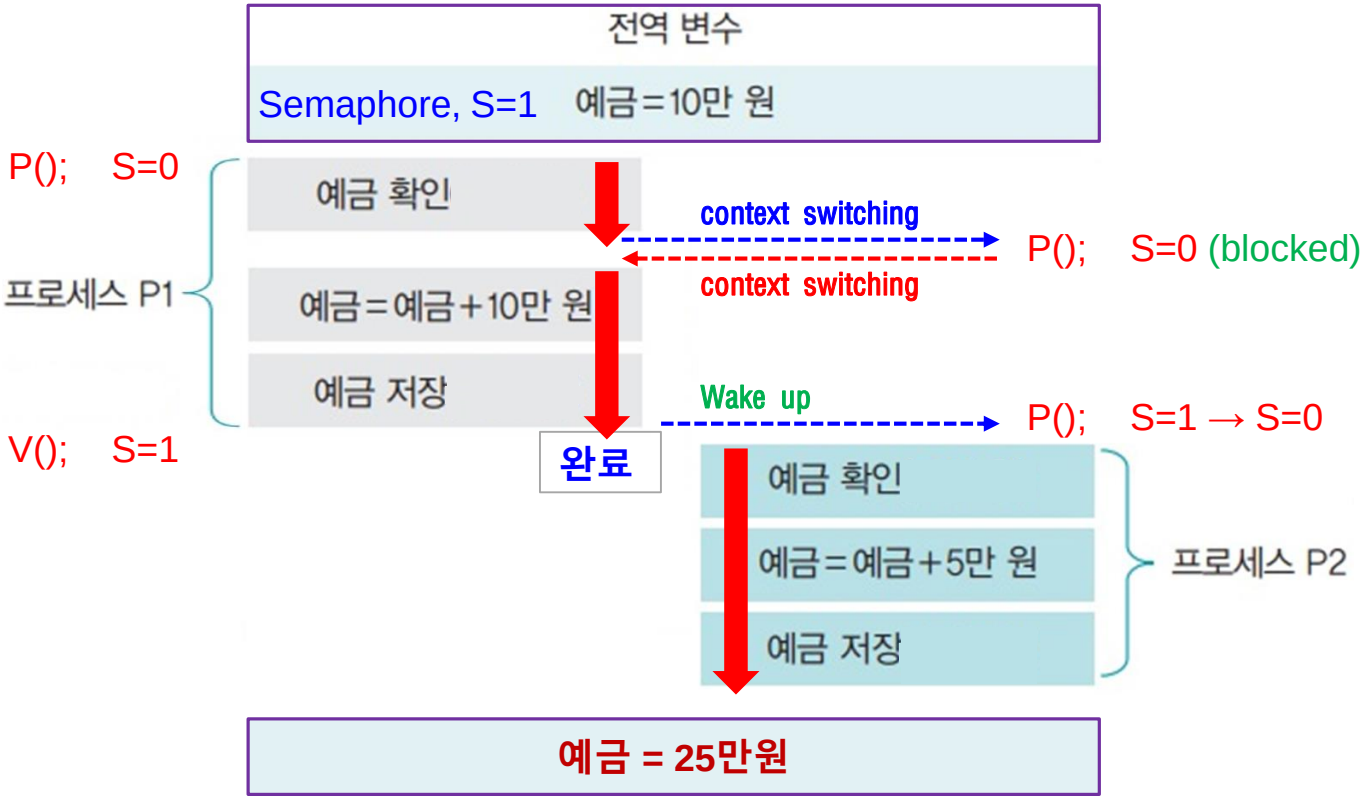
Semaphore를 사용하면?



[그림] 공유 자원의 접근

example Semaphore (세마포) -2

- 공유 자원의 접근 예 – Semaphore 사용



[그림] 공유 자원의 접근 - Semaphore 사용

Semaphore (세마포)

- Semaphore의 잘못된 사용 예

[그림] Semaphore의 잘못된 사용 예

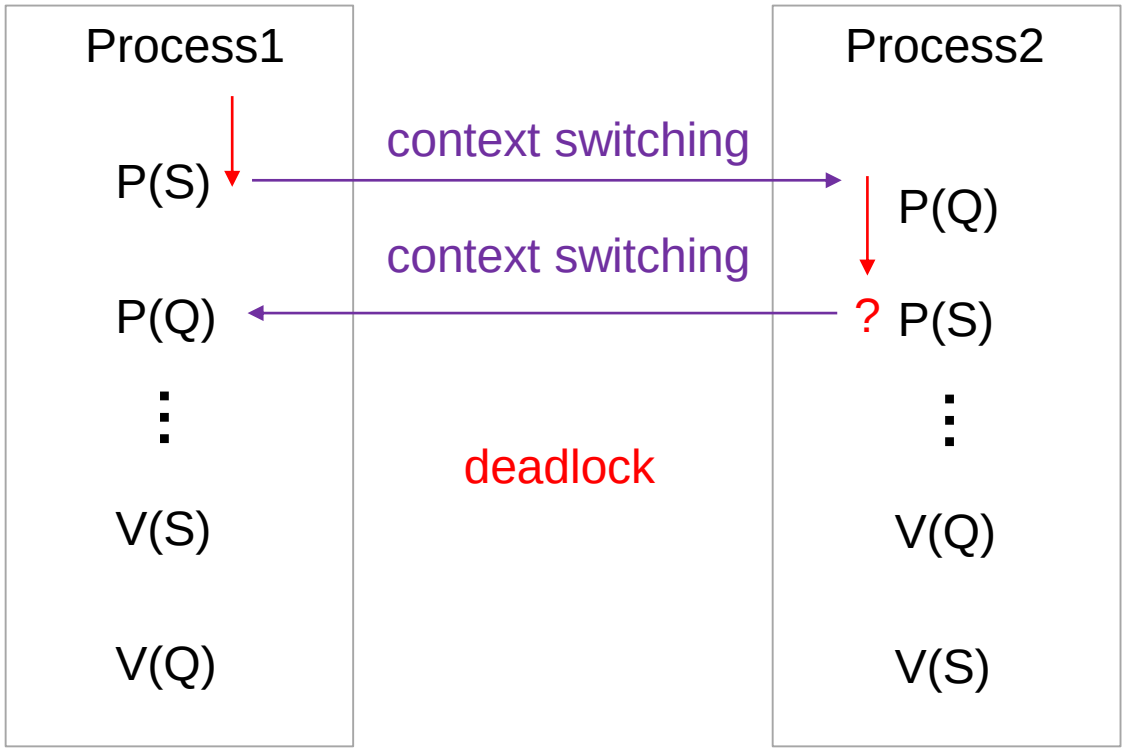
❶ process가 Semaphore를 사용하지 않고 바로 Critical Section에 들어간 경우로 Critical Section을 보호할 수 없음

❷ P( )를 두 번 사용하여 wake_up 신호가 발생하지 않은 경우로 process 간의 Synchronization가 이루어지지 않아 Semaphore 큐에서 대기하고 있는 process들이 무한 대기에 빠짐

❸ P( )와 V( )를 반대로 사용하여 상호 배제가 보장되지 않은 경우로 Critical Section을 보호할 수 없음

example Semaphore (세마포)의 잘못된 사용

- 두 개의 자원 S와 Q가 있고, process는 1과 2가 있음
  두 process 모두 자원 2개를 모두 획득 해야 작업이 가능



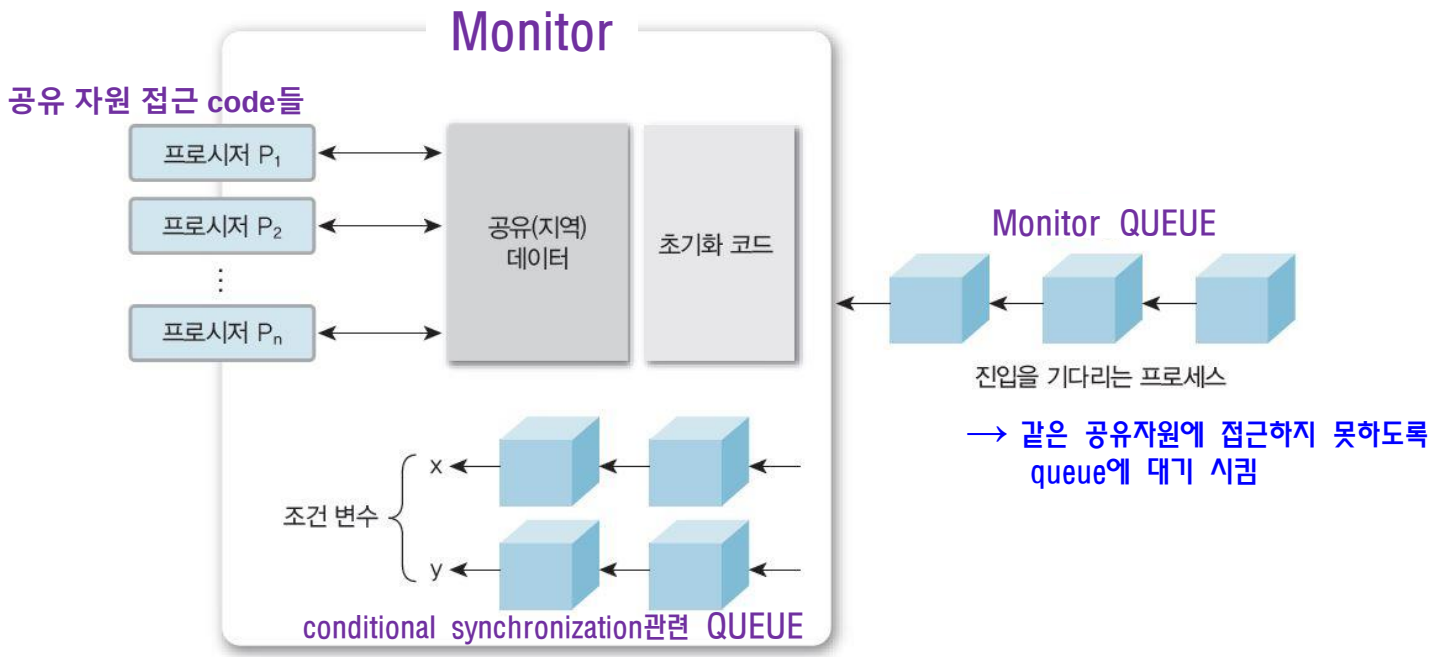
해결 방법?

Monitor (모니터)

- **Monitor**
  - program language에서 제공하는 Mutex 구조체
    - Programmer가 직접 공유데이터 접근에 관한 programming을 하지 않음
      → programmer의 부담이 적어짐
  - 각각의 process는 monitor 안에 있는 procedure(함수)를 통해서만 monitor에 진입 가능
  - Monitor 내부에 정의된 공유data(변수)들은 monitor 내부 code를 통해서만 접근 가능
    - Monitor의 공유 데이터에 접근되는 active process는 1개
      (lock / unlock 필요 없음)
    - Shared/condition variables, functions, shared variable 초기화 code 로 구성

Monitor (모니터)

- Monitor의 구조의 예



[그림] 조건 변수가 있는 Monitor의 구조

❶ Critical Section으로 지정된 변수나 자원에 접근하고자 하는 process는 직접 P( )나 V( )를 사용하지 않고 Monitor에 작업 요청

❷ Monitor는 요청받은 작업을 Monitor Queue에 저장한 후 순서대로 처리하고 그 결과만 해당 process에 알려줌

Monitor (모니터)

- Monitor의 구조

```
Monitor monitor_name
{
                                        // 공유 데이터 변수 선언
    procedure P1(. . . .) {
            . . . .
    }
    procedure P2(. . . .) {
            . . . .
    }
    .
    .
    procedure Pn(. . . .) {
            . . . .
    }

    initialization_code(. . . .) {    // 코드 초기화
            . . . .
    }
}
```

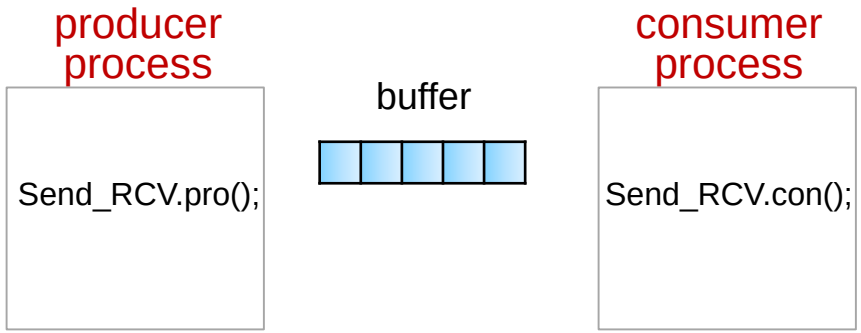
[그림] Monitor의 구조

1. 공유 데이터에 접근
→ 반드시 Monitor 안의 함수를 통해서만 접근가능

2. 실행되는 함수는 Monitor가 하나로 제한
→ Monitor 안의 CODE가 공유데이터에 접근 하도록 하면서 Synchronization가 됨

example Monitor (모니터)

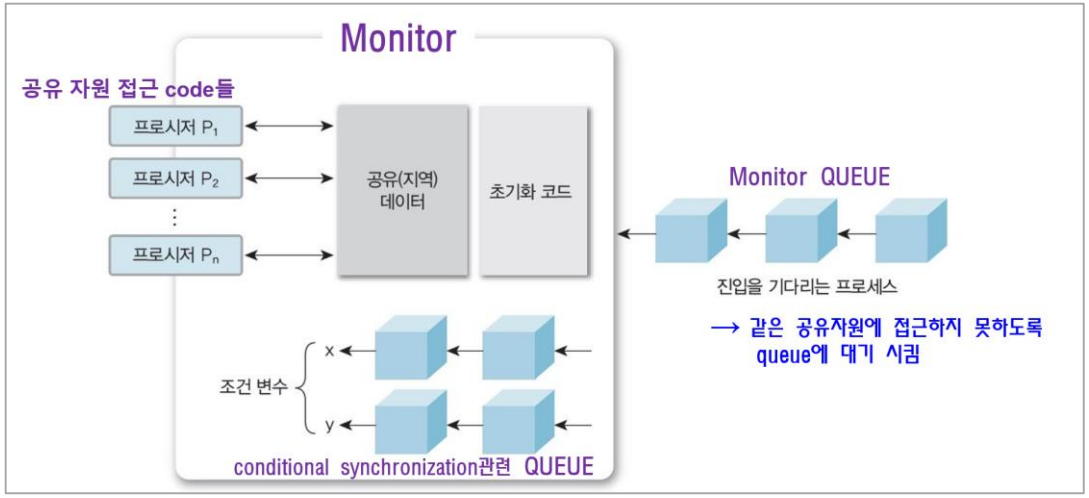
- **생산자, 소비자 문제**



- pro(), con()는 monitor의 critical section에 접근하는 함수

(한 process가 pro()를 수행 중이면 다른 process는 con()를 수행하면 안됨)

➡ Monitor의 공유 데이터에 접근되는 active process는 1개




```
monitor send_RCV
{
  in, out, sum;
  condition full, empty;
   pro() {    // 공유데이터 접근 code1
   if (buffer is full) {
   full.signal();
   full.wait(); }
   sum=sum+1;
   }
   con() {    // 공유데이터 접근 code2
   if (buffer is empty) {
   empty.signal();
   empty.wait(); }
   sum=sum-1;
   }
          ⋮
  initial() {
          ⋮
   }
}
```

empty.signal로 wait() 에서 벗어날 수 있음

full.signal로 wait() 에서 벗어날 수 있음

감사합니다.