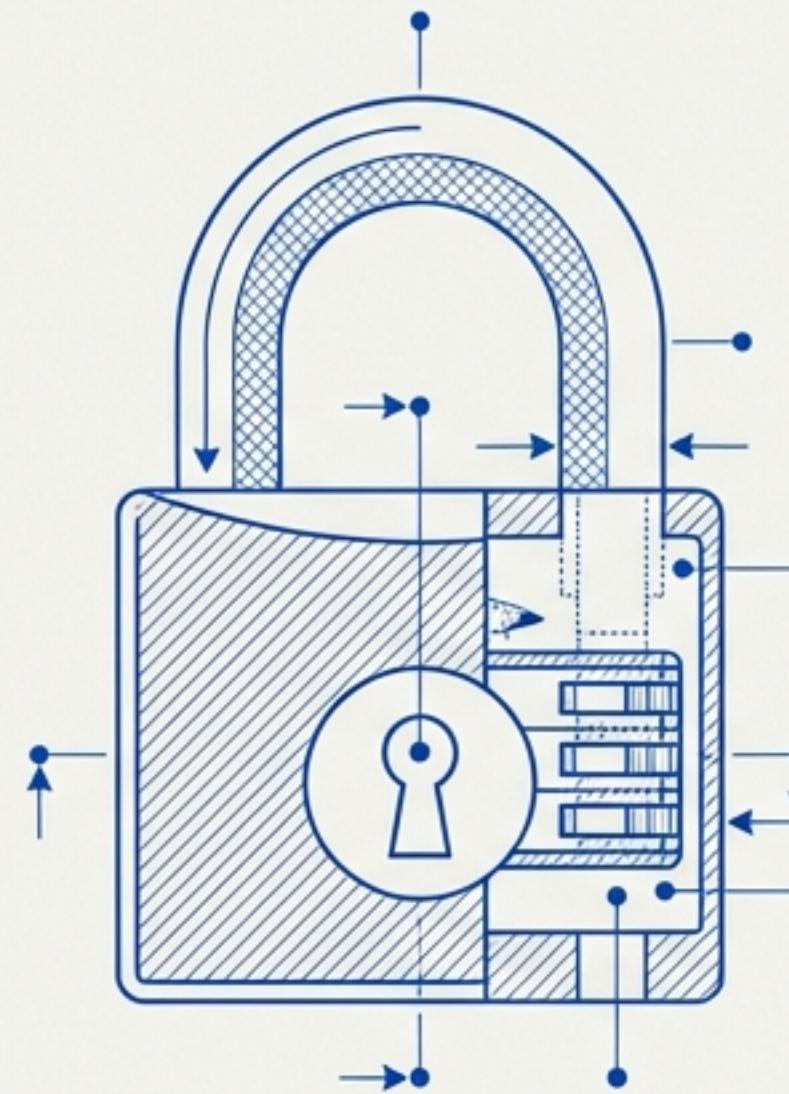
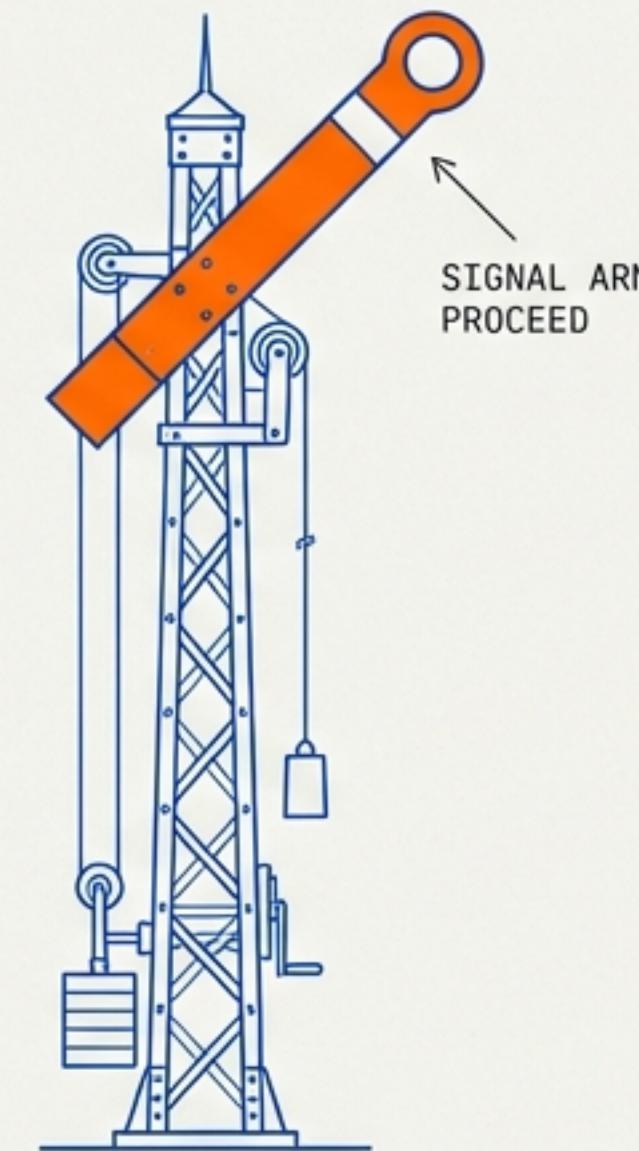


# 프로세스 동기화 기법

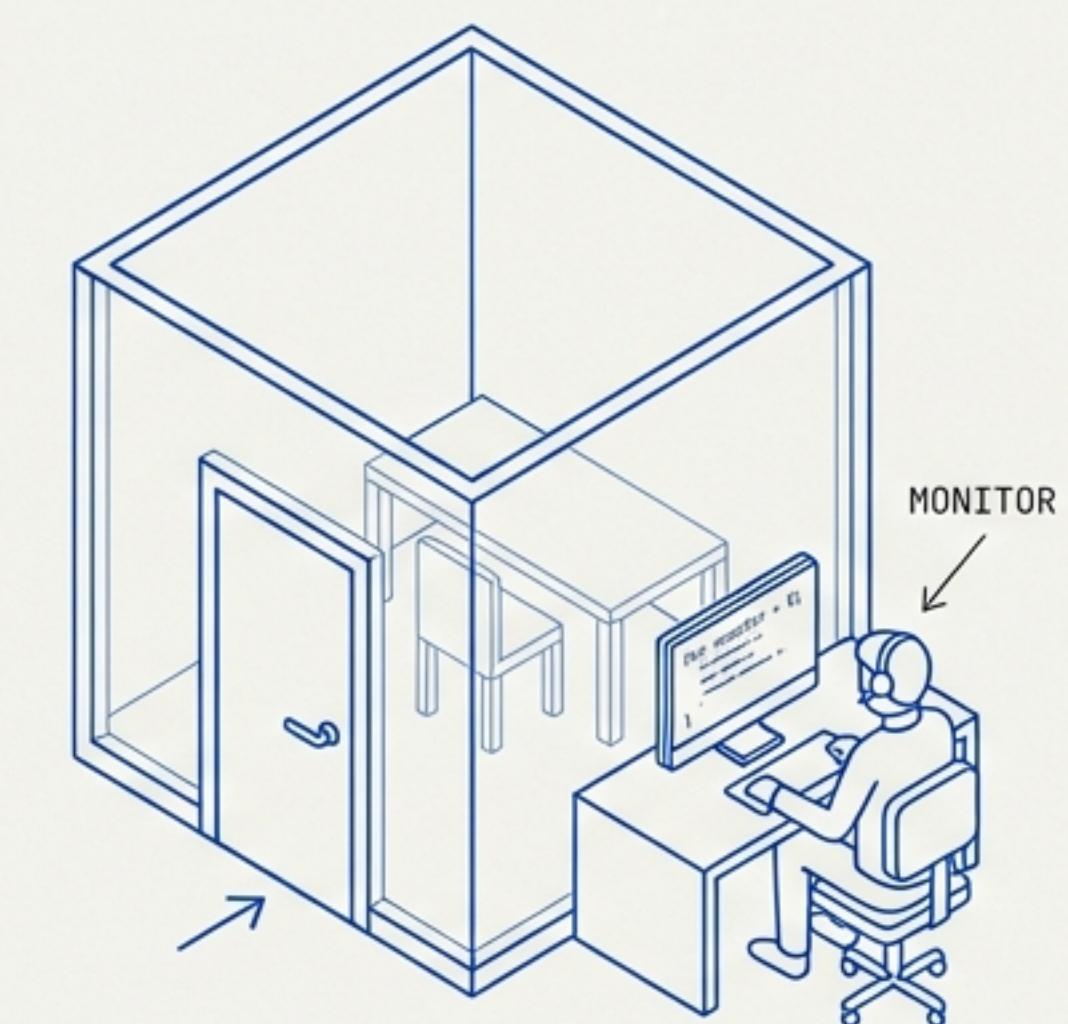
Mutex Lock, Semaphore, Monitor: 혼돈 속의 질서를 만드는 3가지 도구



Mutex



Semaphore

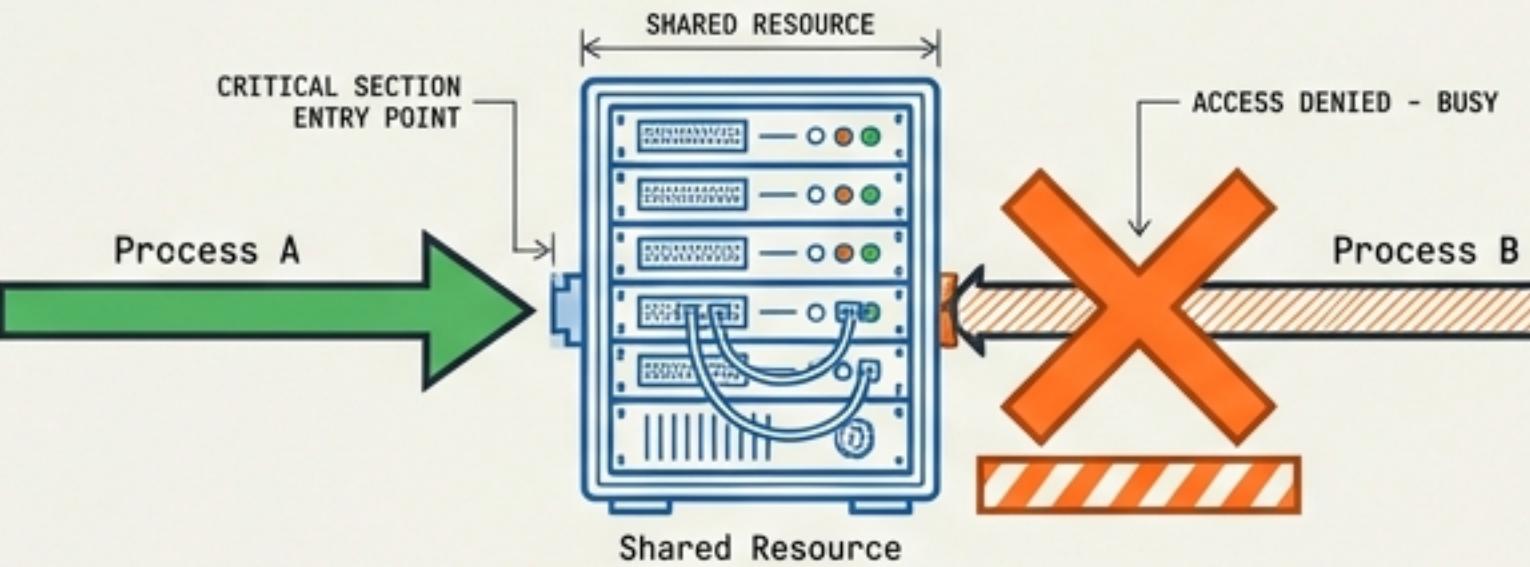


Monitor

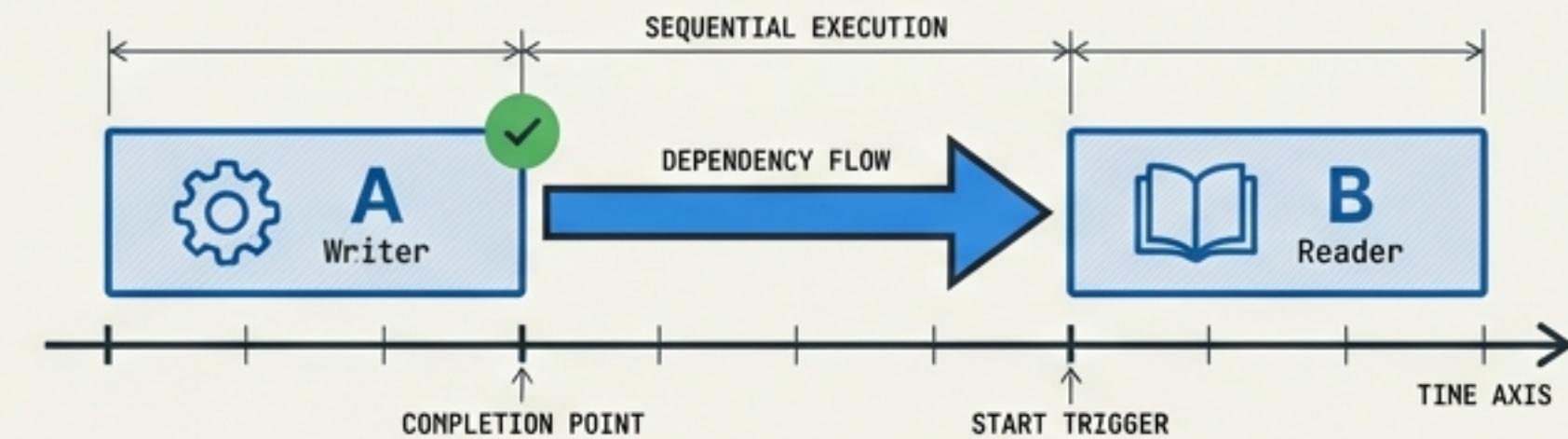
# 동기화의 두 가지 핵심 목표

프로세스 동기화가 실패하면 공유 자원(데이터)의 무결성이 훼손됩니다.  
우리는 다음 두 가지를 반드시 보장해야 합니다.

## 1. Mutual Exclusion (상호 배제)

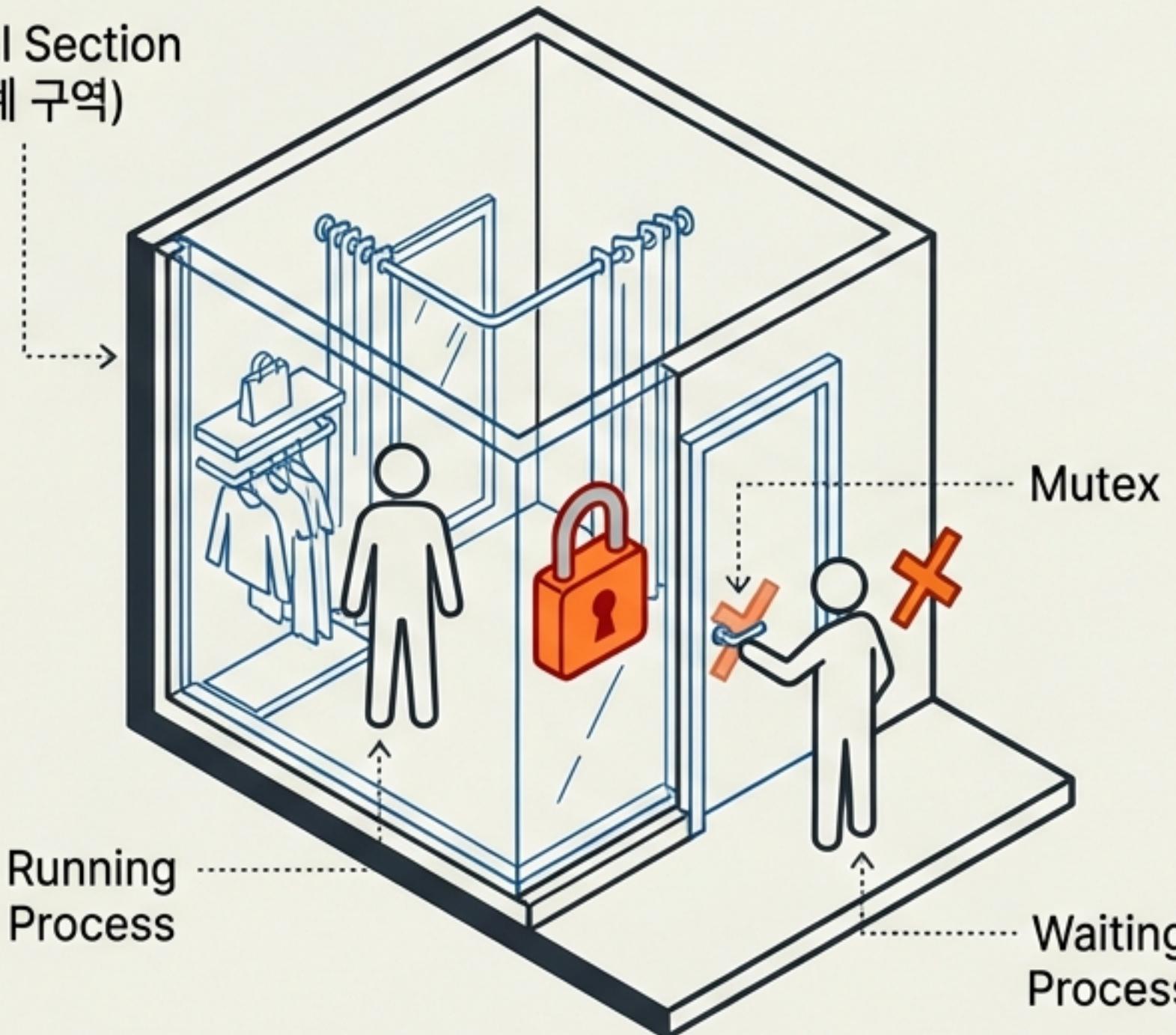


## 2. Execution Order Control (실행 순서 제어)



# Mutex Lock: 탈의실의 자물쇠

Critical Section  
(임계 구역)



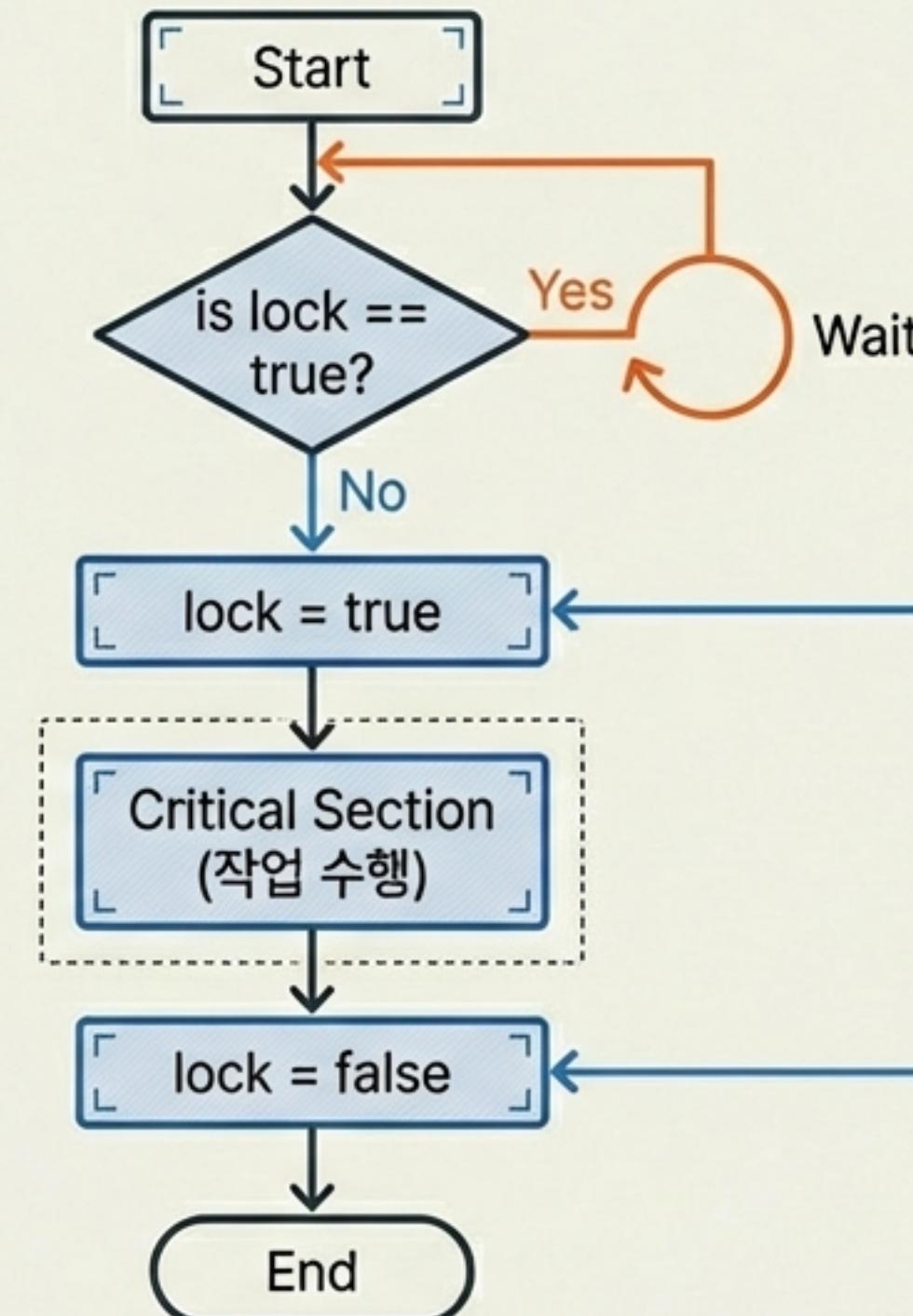
## 개념 (Concept):

Mutual Exclusion(상호 배제)의 약자.

## 메커니즘 (Mechanism):

1. 탈의실(공유 자원)에 들어갈 때 자물쇠를 잠금니다 (Lock = True).
2. 나올 때 자물쇠를 엽니다 (Lock = False).
3. 밖에서 기다리는 사람은 자물쇠가 잠겨 있는지 계속 확인합니다.

# Mutex의 작동 원리와 코드 구현



## Pseudo-code

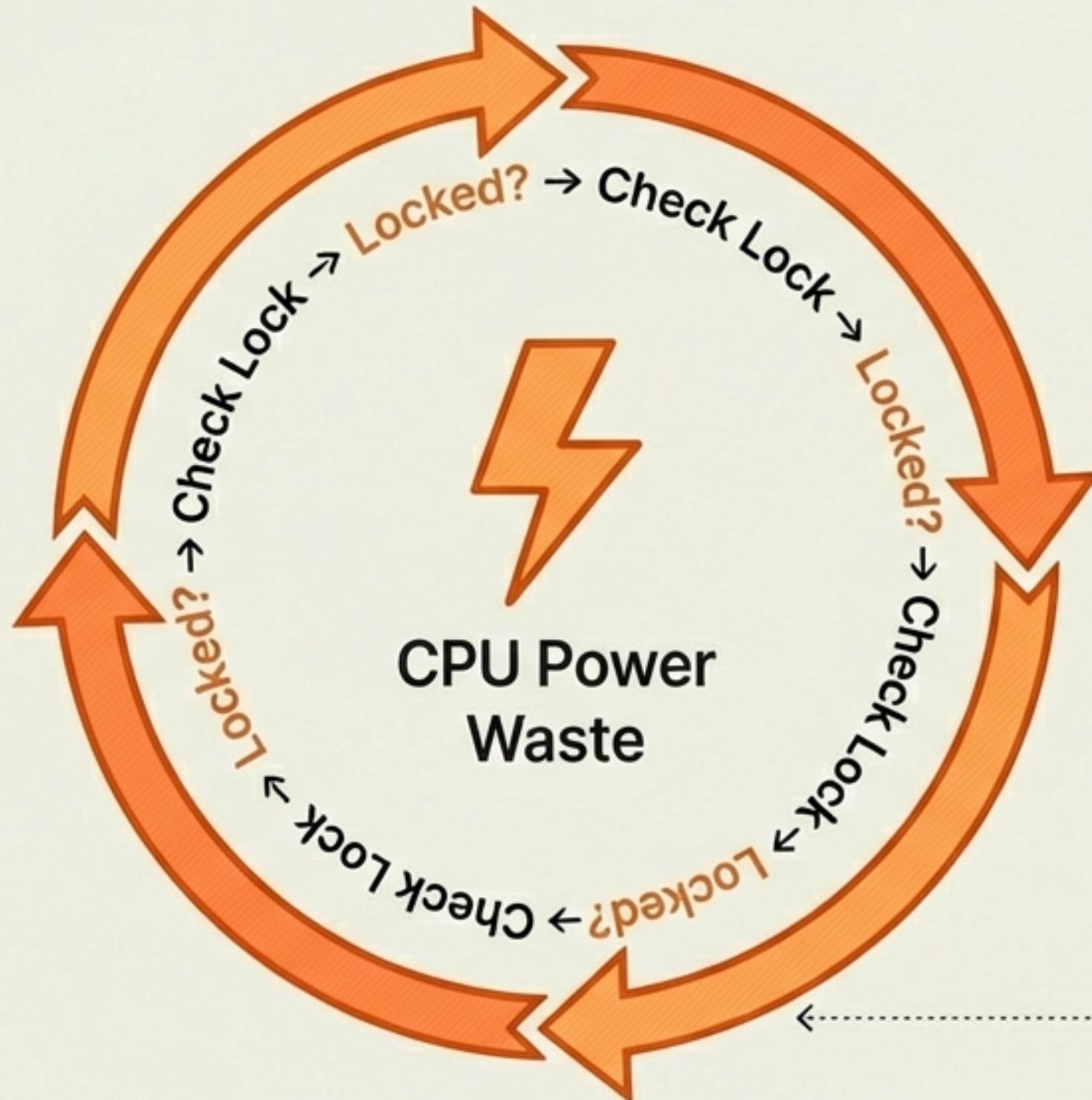
```

1 // Global Variable
2 boolean lock = false;
3
4 void acquire() {
5   while (lock == true) {
6     /* 대기 (Wait) */
7   }
8   lock = true; // 잠금 설정
9 }
10
11 void release() {
12   lock = false; // 잠금 해제
13 }
  
```

JetBrains Rider

DOC. 00- ATSC-SPSE-AB1

# 단순함의 대가: 바쁜 대기 (Busy Waiting)



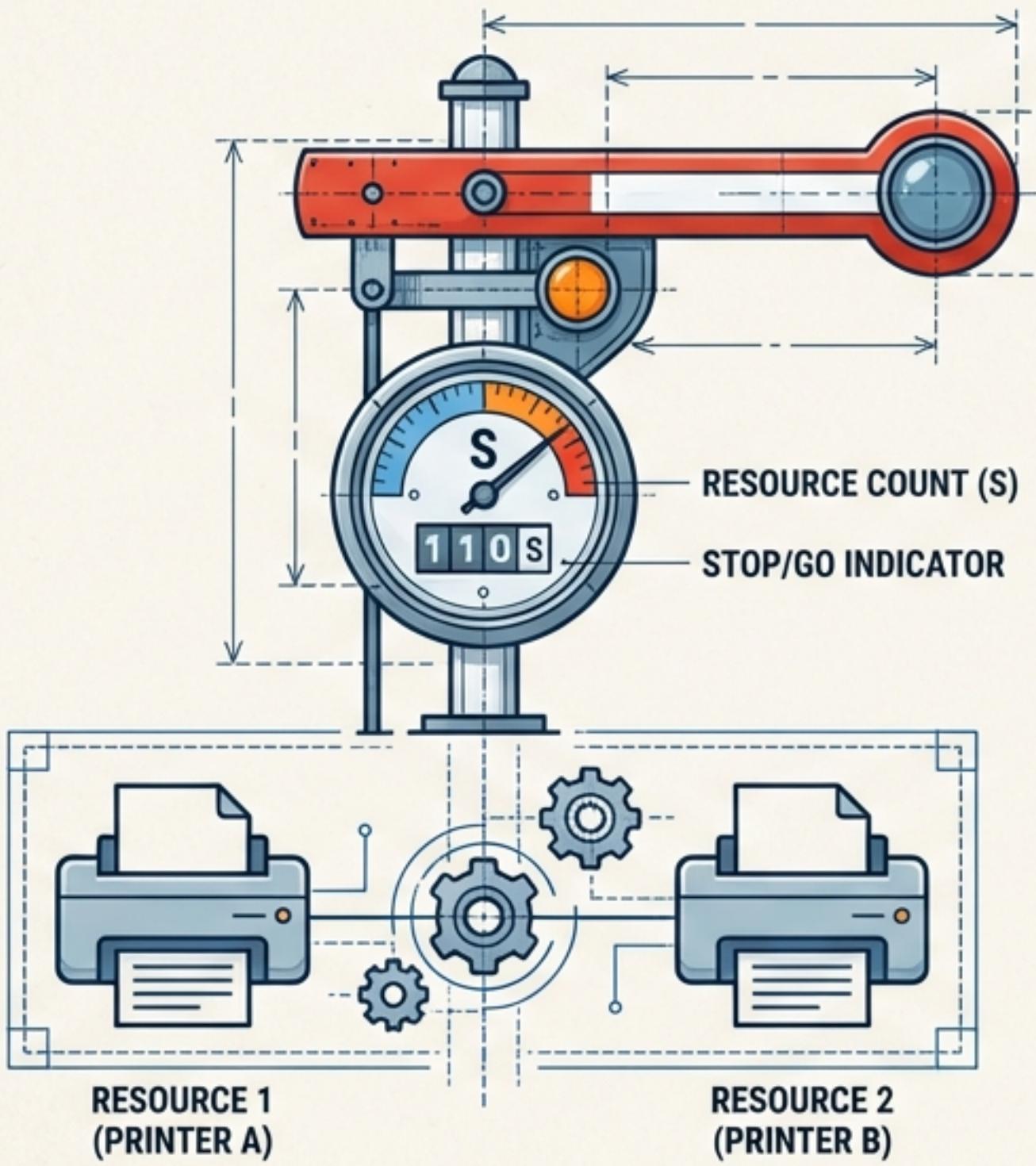
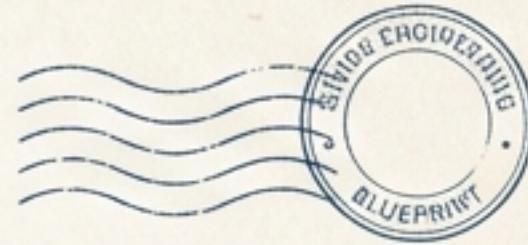
## CPU 자원의 낭비

acquire() 함수의 while 루프는 프로세스가 문이 열릴 때까지 쉴 새 없이 자물쇠를 확인하게 만듭니다.

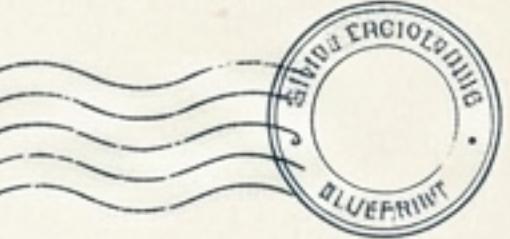
마치 탈의실 문고리를 1초마다 덜컥거리며 확인하는 것과 같으며, 실제 작업을 하지 않으면서 CPU 점유율을 소모합니다.

해결책이 필요함: 쉴 새 없이 확인하는 대신, 자리가 나면 '알림'을 받을 수는 없을까?

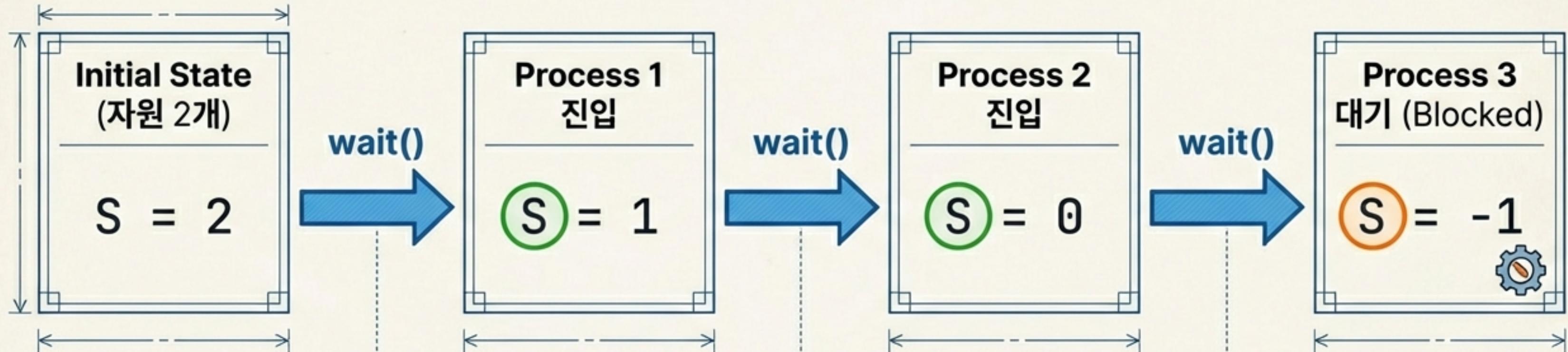
# Semaphore: 철도 신호기



- 신호(Signal) 기반의 관리 도구
- 철도 신호기(Semaphore)에서 유래.
- Mutex는 오직 1개의 자원(Lock)만 관리하지만, Semaphore는 여러 개의 공유 자원을 관리할 수 있습니다.
- Global Variable 'S' : 사용 가능한 자원의 개수를 나타내는 정수 변수.
  - $S > 0$  : 진입 가능 (Go) ■
  - $S \leq 0$  : 대기 (Stop) ■



# Semaphore의 두 가지 원자적 연산



## wait():

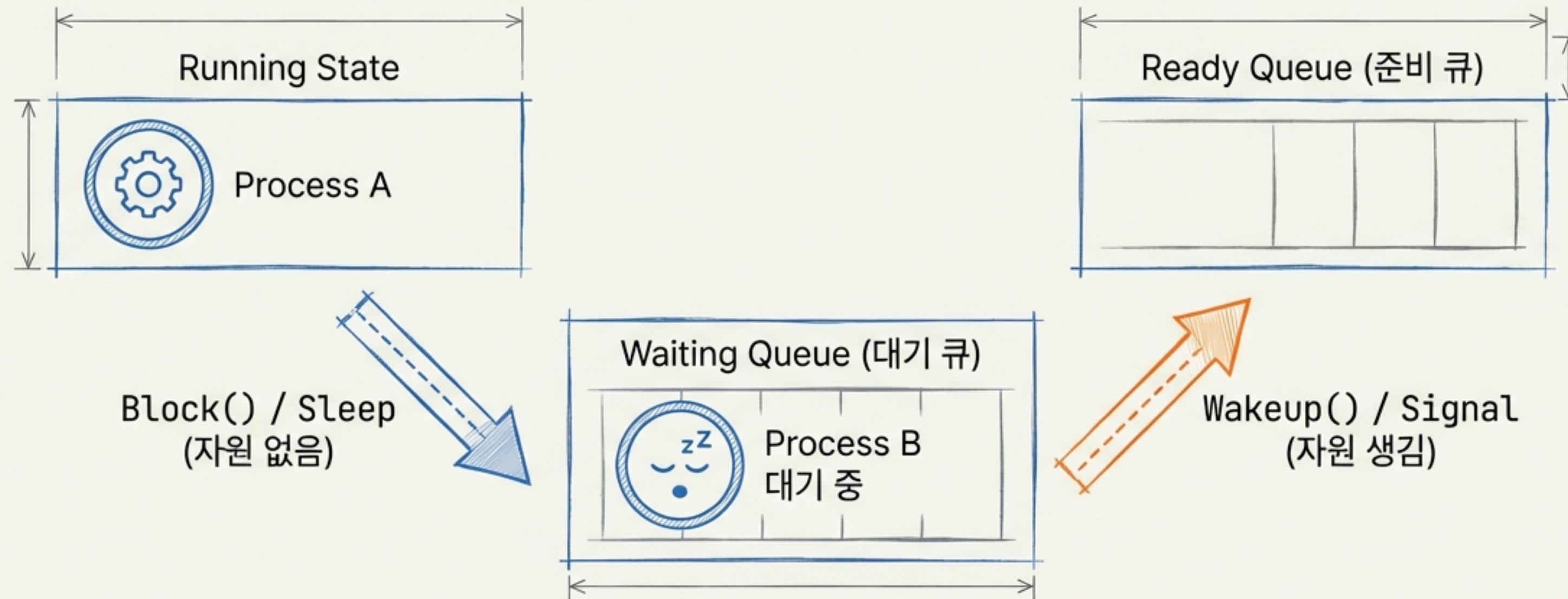
임계 구역 진입 전 호출.  $S$ 를 감소시킴.  
자원이 부족하면 대기.

## signal():

임계 구역 이탈 후 호출.  $S$ 를 증가시킴.  
대기 중인 프로세스에게 신호를 보냄.



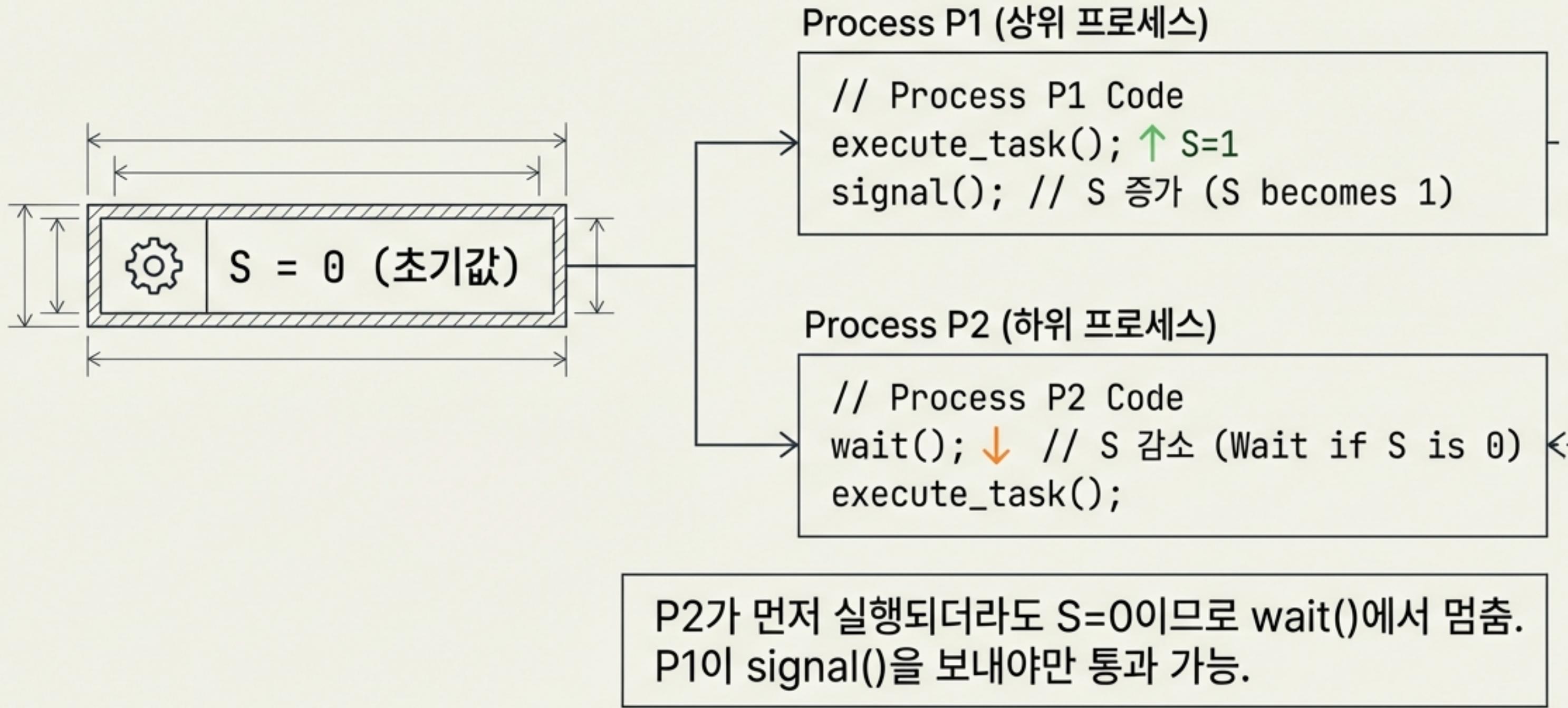
# 바쁜 대기 해결: 재우기와 깨우기 (Sleep & Wake)



- 무한 루프(Busy Waiting)를 도는 대신, 자원이 없을 때 프로세스를 **대기 상태(Block)**로 전환합니다.
- 자원이 생기면 **대기 큐**에 있던 프로세스를 깨워 **준비 큐**로 보냅니다.
- **장점:** 불필요한 CPU 사이클 소모 0.

# Semaphore를 이용한 실행 순서 제어

목표: 무조건 P1이 실행된 후 P2가 실행되어야 함.



# Semaphore의 위험성

Error Case

## 순서 오류 (Wrong Order) ⚠

```
// Wrong Order
signal(S); // S becomes 1
wait(S); // S becomes 0
execute_critical_section();

Mutual Exclusion Failure  

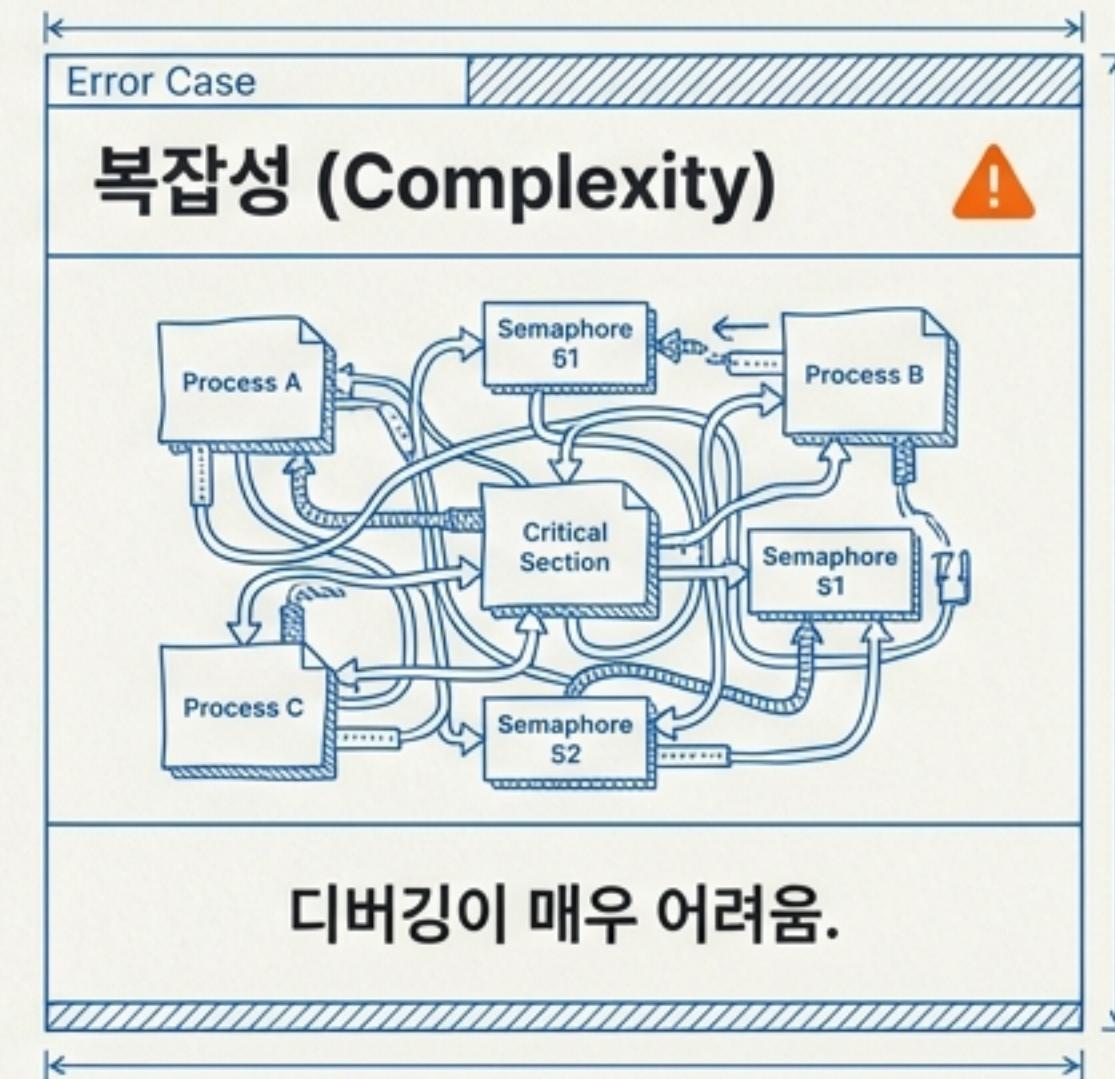
(상호 배제 깨짐).
```

Error Case

## 실수 (Mistake) ⚠

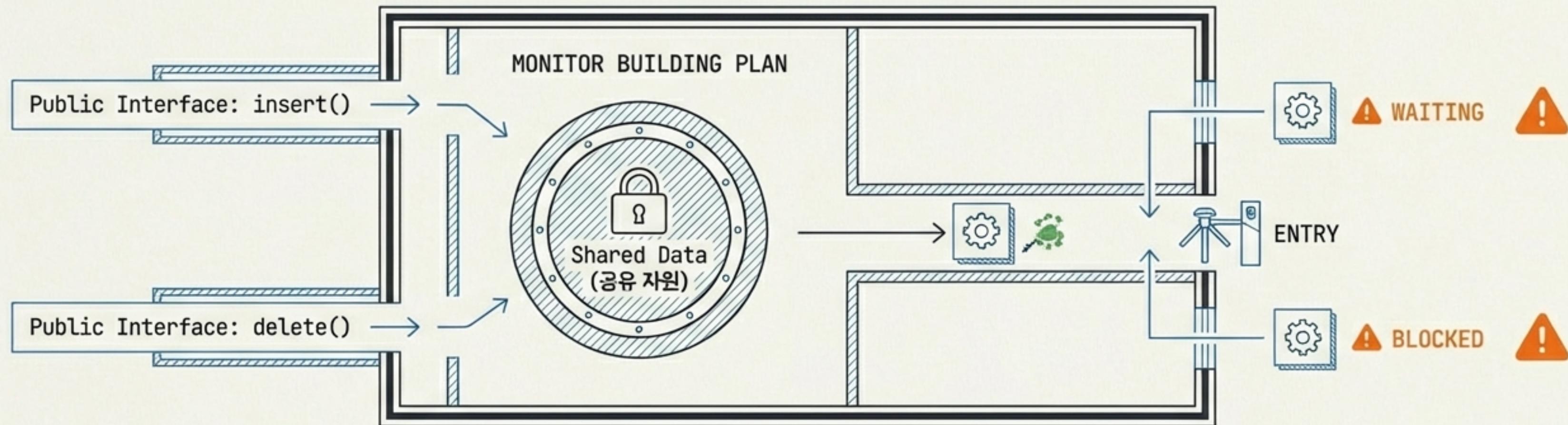
```
// Mistake
wait(S); // S becomes 0
execute_critical_section();
// Forgotten signal(S)!
```

**Deadlock (교착 상태).**



개발자가 직접 락을 관리하는 것은 위험합니다. 더 안전하고 자동화된 도구가 필요합니다.  
 → Monitor의 등장.

# Monitor: 자동화된 관리 시스템



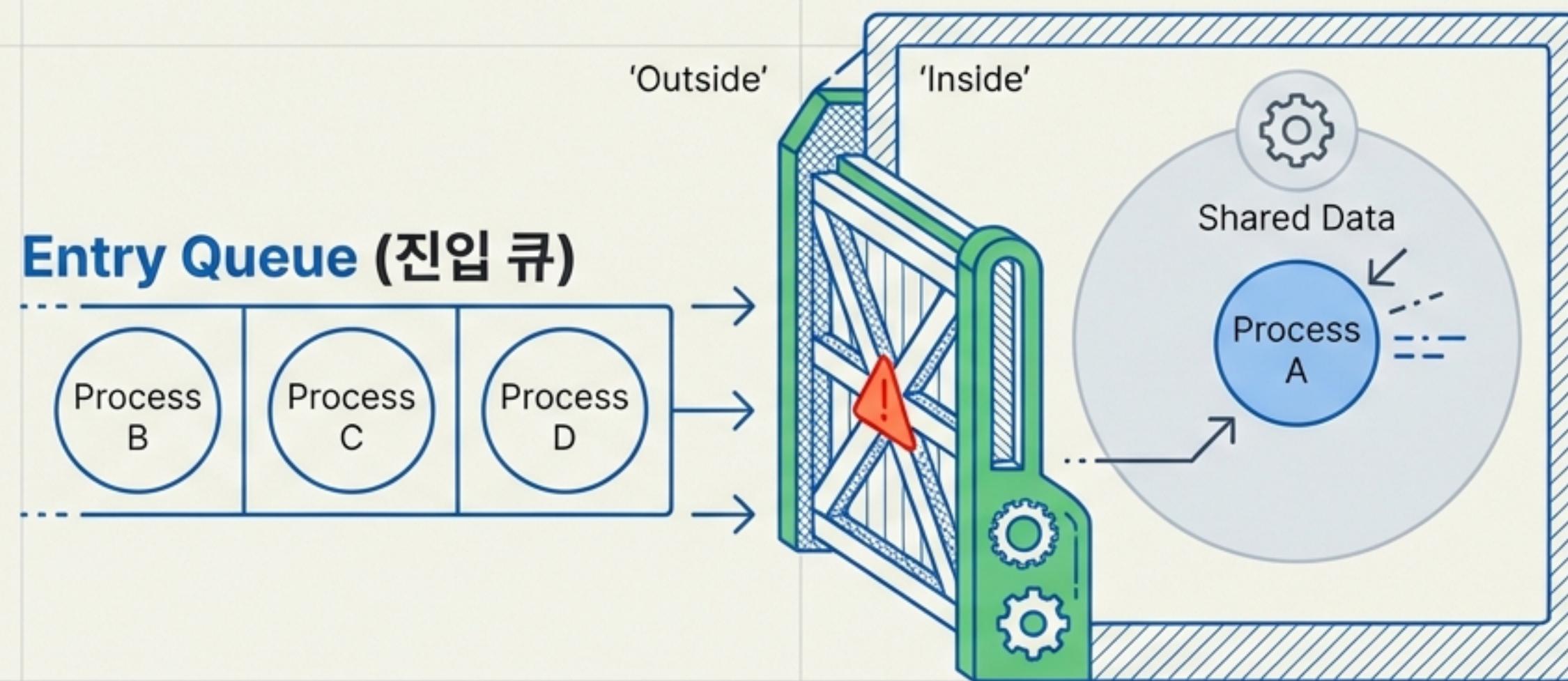
## CONCEPT & FEATURES

**개념\***: 개발자가 락을 직접 다루지 않아도 되는 고수준 동기화 도구 (예: Java).

**특징\***: 상호 배제(Mutual Exclusion)가 자동 지원됨.

한 번에 하나의 프로세스만 Monitor 내부의 코드를 실행 가능.

# Monitor의 구조 1: 상호 배제 (Mutual Exclusion)

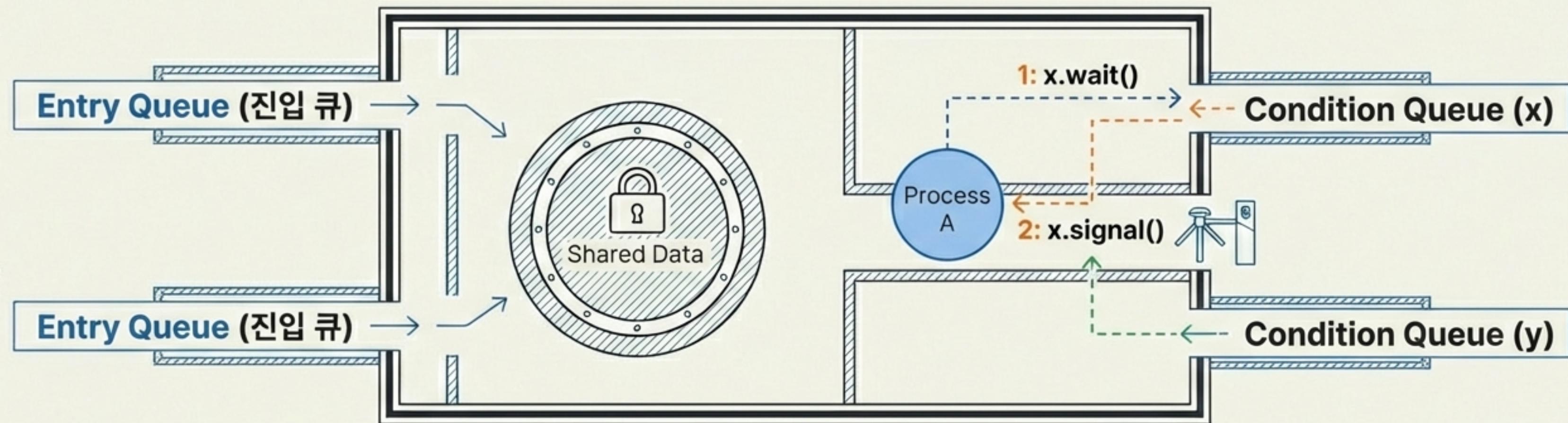


프로세스가 모니터 진입을 시도할 때, 이미 다른 프로세스가 사용 중이라면...

-> 자동으로 **Entry Queue**에서 대기합니다.

-> 개발자는 `acquire`나 `lock`을 명시적으로 작성할 필요가 없습니다.

# Monitor의 구조 2: 실행 순서 제어



**Entry Queue:** 모니터에 들어오기 위해 기다리는 줄.

**Condition Queue:** 모니터 안에서 특정 조건(Condition)이 충족되길 기다리는 줄.

**Condition Variables:** `wait()`와 `signal()`을 통해 내부 프로세스의 실행 순서를 정교하게 제어.

# 동기화 도구의 진화와 비교

	Mutex	Semaphore	Monitor
비유	 자물쇠 (Lock)	 신호기 (Signal)	 안전실 (Safe Room)
방식	단순 잠금 (`lock=true`)	카운터 변수 ('S')	언어 차원의 캡슐화
자원	1개 (상호 배제)	N개 (상호 배제 + 순서)	N개 (자동 상호 배제 + 순서)
장점	가볍고 단순함	강력하고 유연함	사용하기 쉽고 안전함
단점	바쁜 대기 (Busy Waiting)	설계 실수 시 위험 (Deadlock)	지원하는 언어 필요

감사합니다.

Next Lecture: Deadlocks

