



# 4강. 병행 프로세스 I

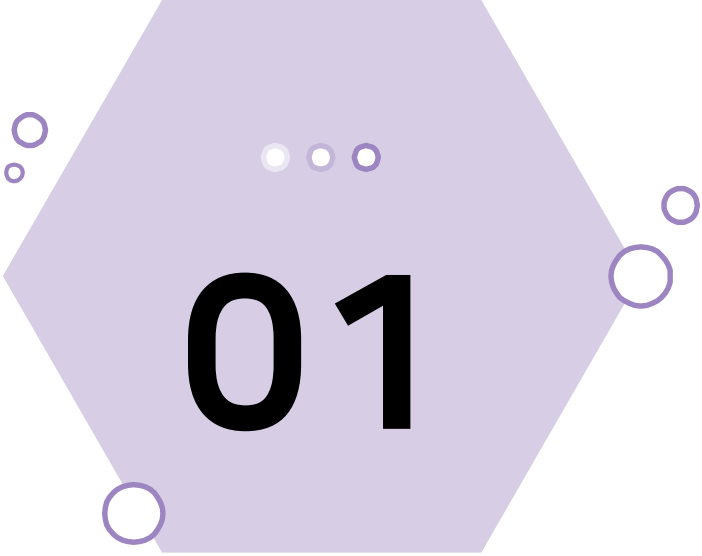
방송대 컴퓨터과학과  
김진욱 교수



# 목차

01 병행 프로세스의 개념

02 동기화와 임계영역










01

# 병행 프로세스의 개념

# 병행성

## ■ 병행성(concurrency)

- 여러 개의 프로세스 또는 스레드가 동시에 실행되는 시스템의 특성

이름	상태	16% CPU	41% 메모리	0% 디스크
앱 (7)				
>  Google Chrome(32 bit)		0%	46.9MB	0.1MB/s
>  Hancom Office Hanword 2010(...)		0%	13.9MB	0MB/s
>  Microsoft PowerPoint(32 bit)(2)		9.3%	106.6MB	0.1MB/s
>  Windows Media Player(32 bit)		0.6%	20.1MB	0MB/s
>  Windows 탐색기		0.1%	35.3MB	0MB/s
 영화 및 TV		0.4%	93.2MB	2.4MB/s
>  작업 관리자		0.8%	12.4MB	0MB/s

# 병행성

## ■ 병행성(concurrency)

- 여러 개의 프로세스 또는 스레드가 동시에 실행되는 시스템의 특성

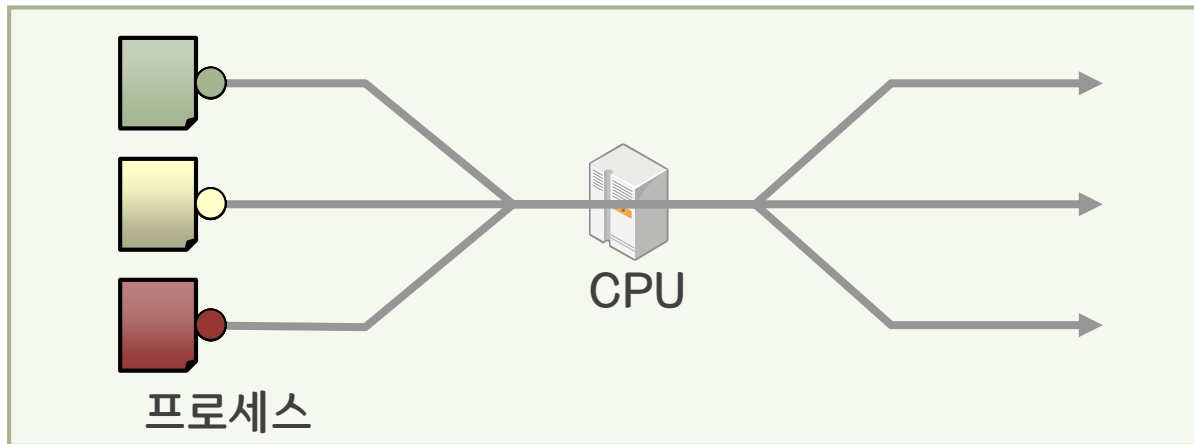
이름	상태	16% CPU	41% 메모리	디스크
앱 (7)				
> Google Chrome(32 bit)		0%	46.9MB	0.1MB/s
> Hancom Office Hanword 2010(...		0%	13.9MB	0MB/s
> Microsoft PowerPoint(32 bit)(2)		9.3%	106.6MB	0.1MB/s
> Windows Media Player(32 bit)		0.6%	20.1MB	0MB/s
> Windows 탐색기		0.1%	35.3MB	0MB/s
영화 및 TV		0.4%	93.2MB	2.4MB/s
> 작업 관리자		0.8%	12.4MB	0MB/s

### ★ 병행 프로세스

동시에 실행되는 여러 개의 프로세스 또는 스레드

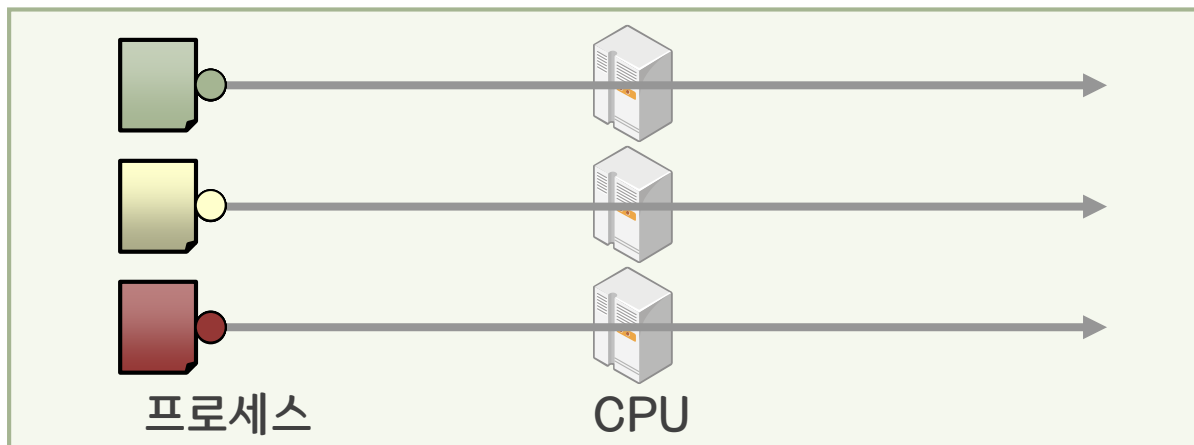
## 병행 프로세스의 실행 형태

- CPU의 개수에 따른 병행 프로세스의 실행 형태
  - 하나의 CPU에서 인터리빙 형식으로 실행



# 병행 프로세스의 실행 형태

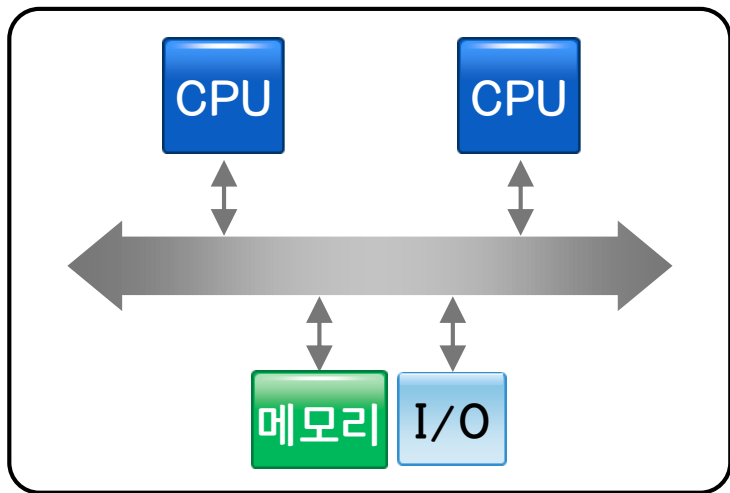
- CPU의 개수에 따른 병행 프로세스의 실행 형태
  - 여러 개의 CPU에서 병렬 처리 형식으로 실행



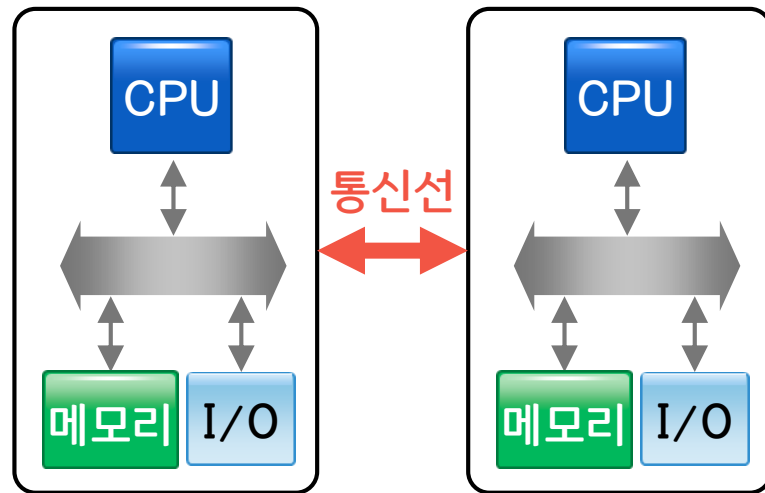
# 병행 프로세스의 실행 형태 – 여러 개의 CPU

## ■ 메모리 구조에 따른 병행 프로세스의 실행 형태

- 강결합 멀티프로세서 시스템
  - 공유 메모리 구조



- 약결합 멀티프로세서 시스템
  - 분산 메모리 구조





# 병행성 문제

- 병행 프로세스들이 상호작용 하는 경우 발생
  - 공유자원 점유 문제
  - 동기화 문제
  - 통신 문제
- 상황에 따른 구분
  - 단일 프로세스 내의 병행성
  - 프로세스 간의 병행성

# 단일 프로세스 내의 병행성

```
S1: a := x + y;  
S2: b := z + 1;  
S3: c := a + b;  
S4: write(c);
```



우선순위 그래프



Fork/Join 구조



병행문

# ○ 단일 프로세스 내의 병행성

## ■ 우선순위 그래프


- 정점: 문장
- 방향 있는 간선: 우선순위 관계

$S_1: a := x + y;$

$S_2: b := z + 1;$

$S_3: c := a + b;$

$S_4: \text{write}(c);$

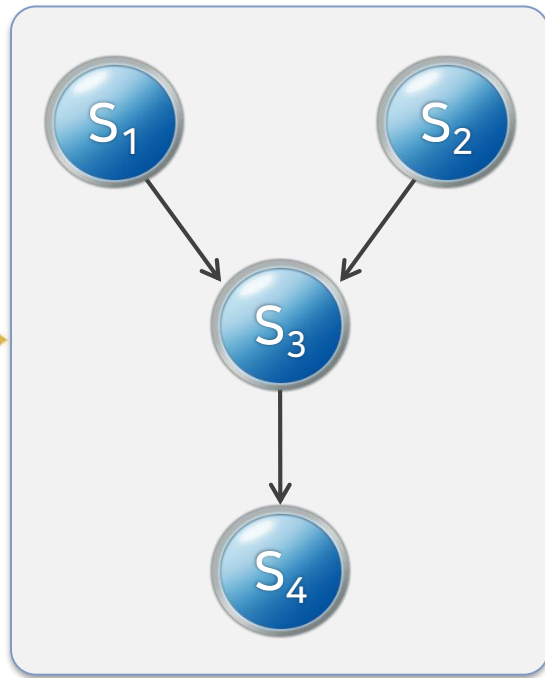


# 단일 프로세스 내의 병행성

## ■ 우선순위 그래프

- 정점: 문장
- 방향 있는 간선: 우선순위 관계

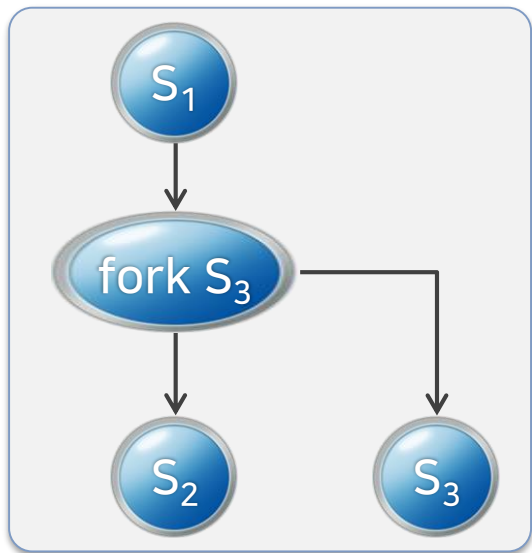
```
S1: a := x + y;  
S2: b := z + 1;  
S3: c := a + b;  
S4: write(c);
```



# 단일 프로세스 내의 병행성

## ■ Fork/Join 구조

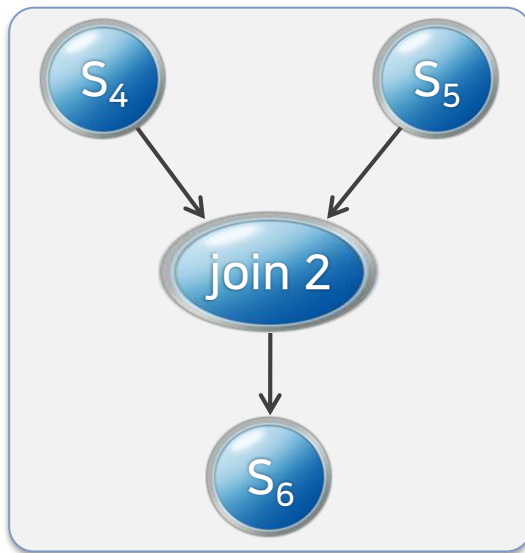
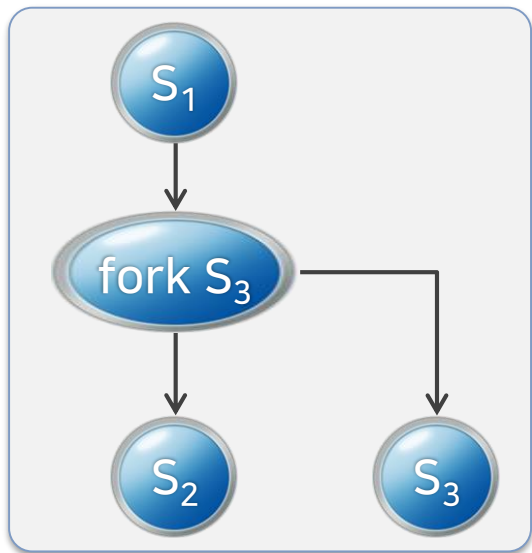
- fork L : 2개의 병행 수행을 만듦(레이블 L 위치, fork 명령어 다음)
- join n : 병행하는 n개의 흐름을 하나로 재결합



# 단일 프로세스 내의 병행성

## ■ Fork/Join 구조

- fork L : 2개의 병행 수행을 만듦(레이블 L 위치, fork 명령어 다음)
- join n : 병행하는 n개의 흐름을 하나로 재결합



# ○ 단일 프로세스 내의 병행성

## ■ Fork/Join 구조

$S_1$ :  $a := x + y$ ;

$S_2$ :  $b := z + 1$ ;

$S_3$ :  $c := a + b$ ;

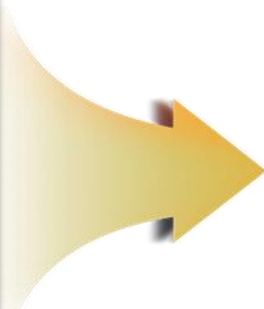
$S_4$ :  $\text{write}(c)$ ;



# 단일 프로세스 내의 병행성

## ■ Fork/Join 구조

```
S1: a := x + y;  
S2: b := z + 1;  
S3: c := a + b;  
S4: write(c);
```



```
count := 2;  
fork L1;  
a := x + y;  
go to L2;  
L1: b := z + 1;  
L2: join count;  
c := a + b;  
write(c);
```

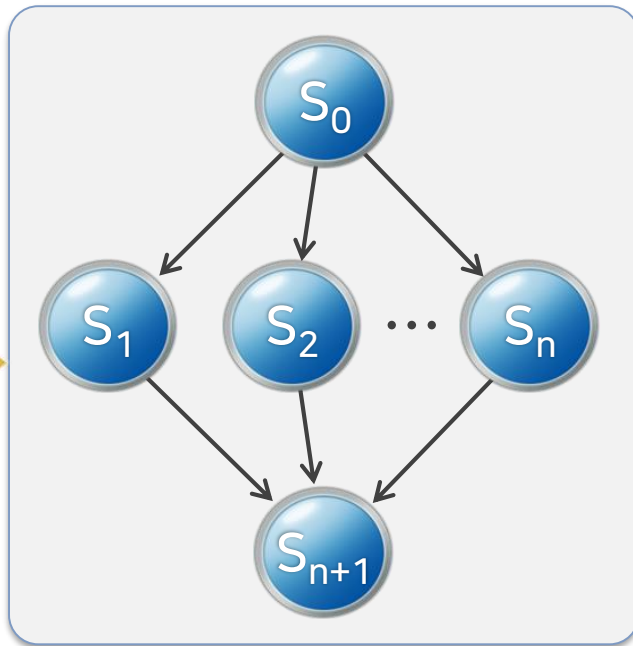


# 단일 프로세스 내의 병행성

## ■ 병행문

- 1개의 프로세스가 여러 가닥의 병렬 프로세스로 분할되었다가 다시 하나로 결합
- `parbegin` / `parend` 문

```
S0;  
parbegin  
    S1;  
    S2;  
    ⋮  
    Sn;  
parend  
Sn+1;
```



# ○ 단일 프로세스 내의 병행성


## ■ 병행문

$S_1: a := x + y;$

$S_2: b := z + 1;$

$S_3: c := a + b;$

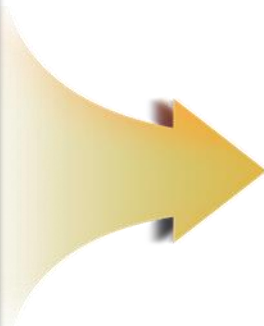
$S_4: \text{write}(c);$



# 단일 프로세스 내의 병행성

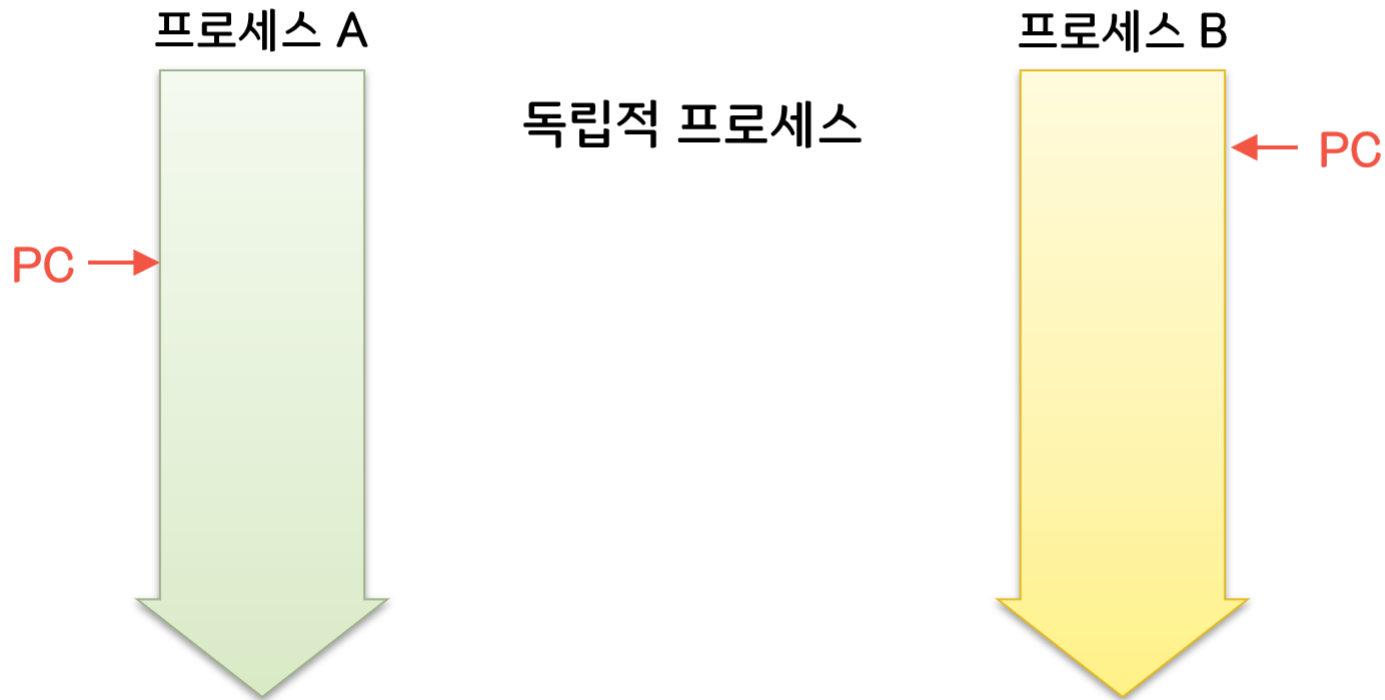
## ■ 병행문

```
S1: a := x + y;  
S2: b := z + 1;  
S3: c := a + b;  
S4: write(c);
```



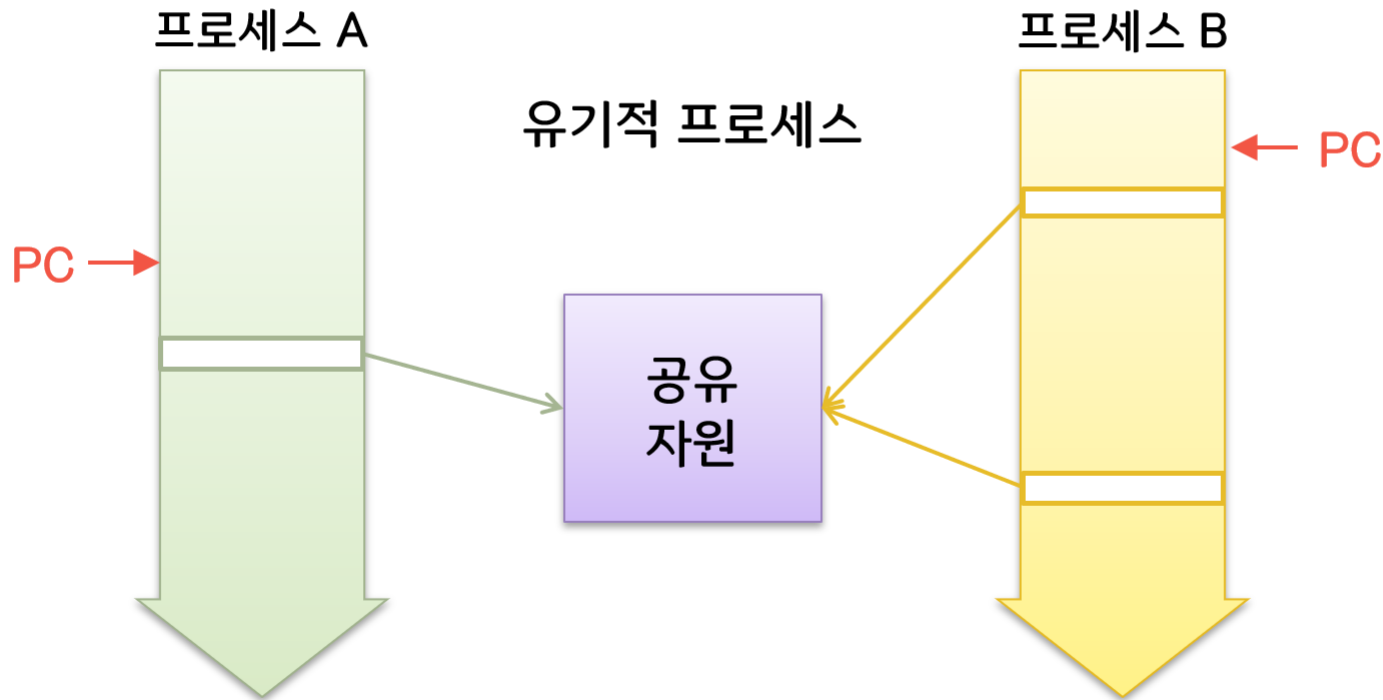
```
parbegin  
    a := x + y;  
    b := z + 1;  
parend  
    c := a + b;  
    write(c);
```

# 프로세스 간의 병행성



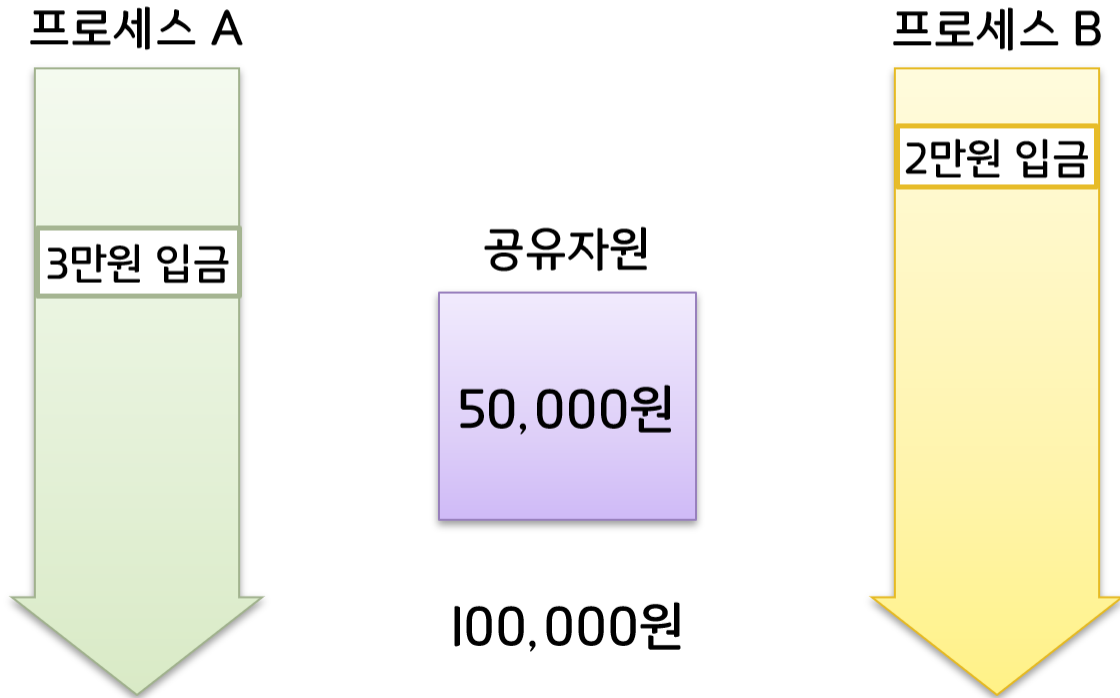
# 프로세스 간의 병행성

## 비동기 병행 프로세스



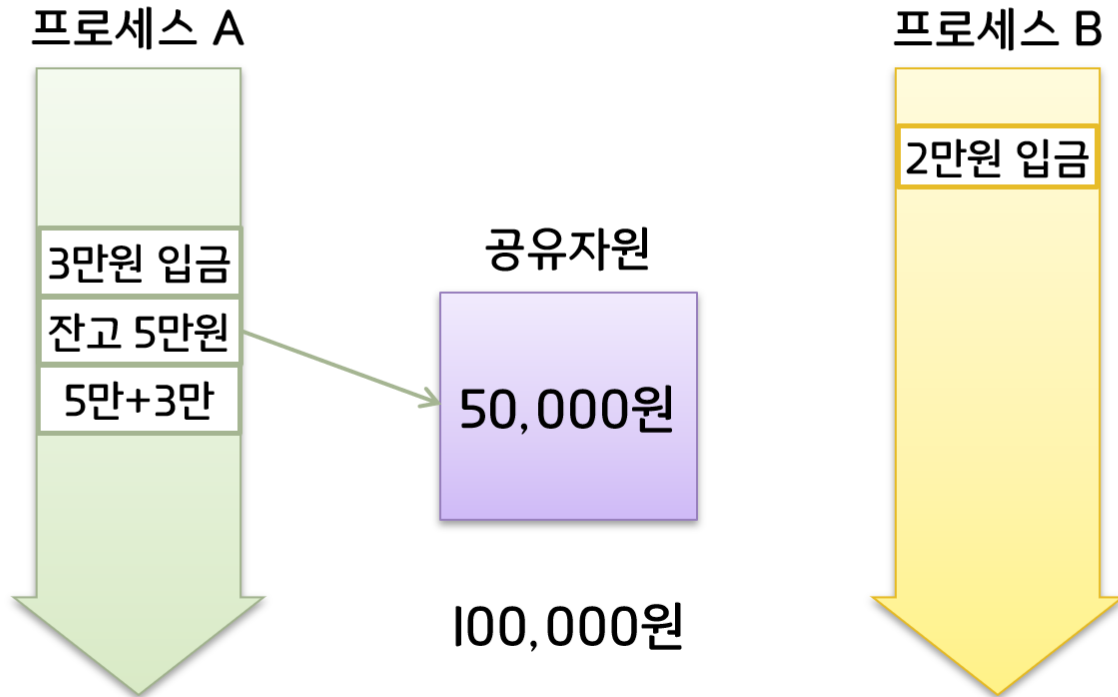
# 프로세스 간의 병행성

## ■ 비동기 병행 프로세스



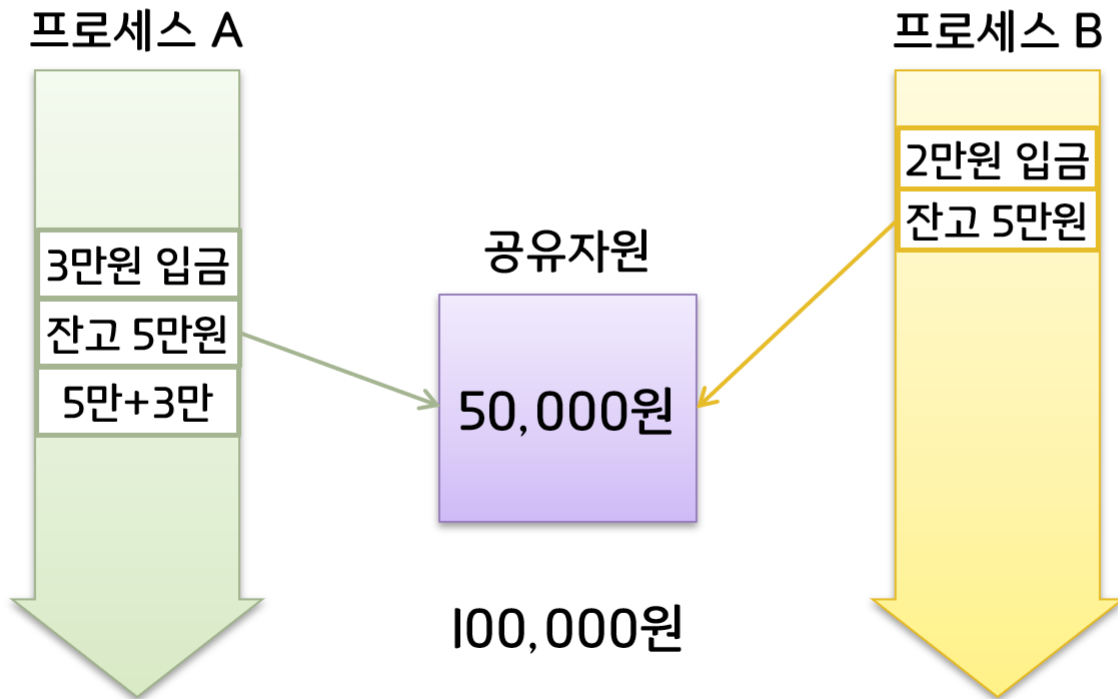
# 프로세스 간의 병행성

## 비동기 병행 프로세스



# 프로세스 간의 병행성

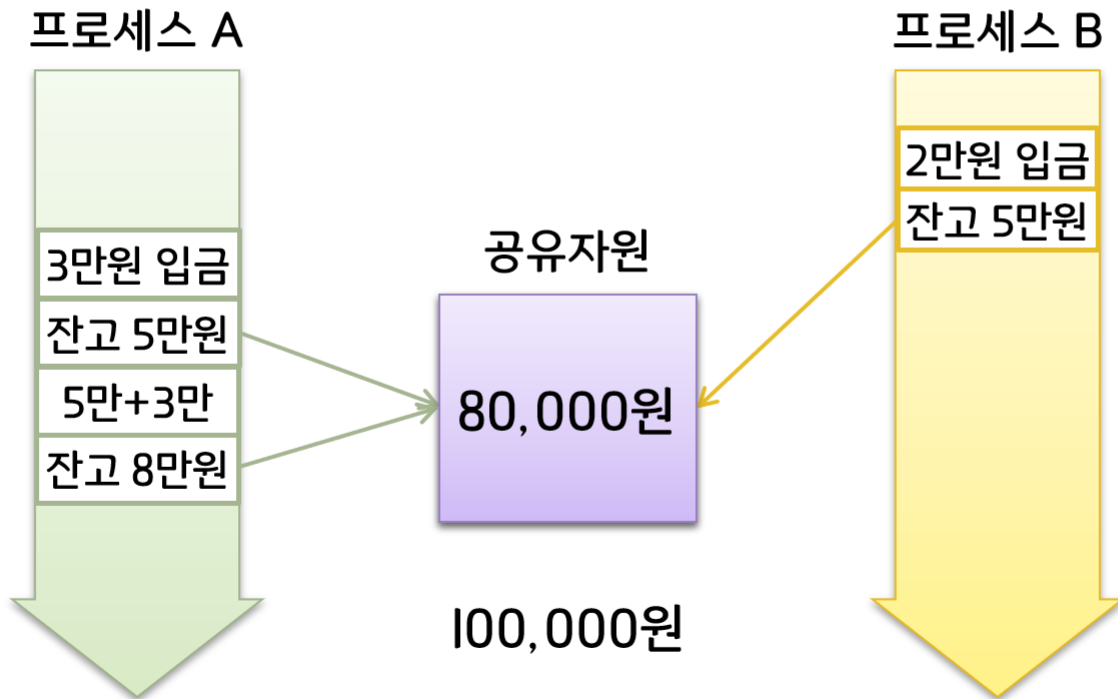
## 비동기 병행 프로세스





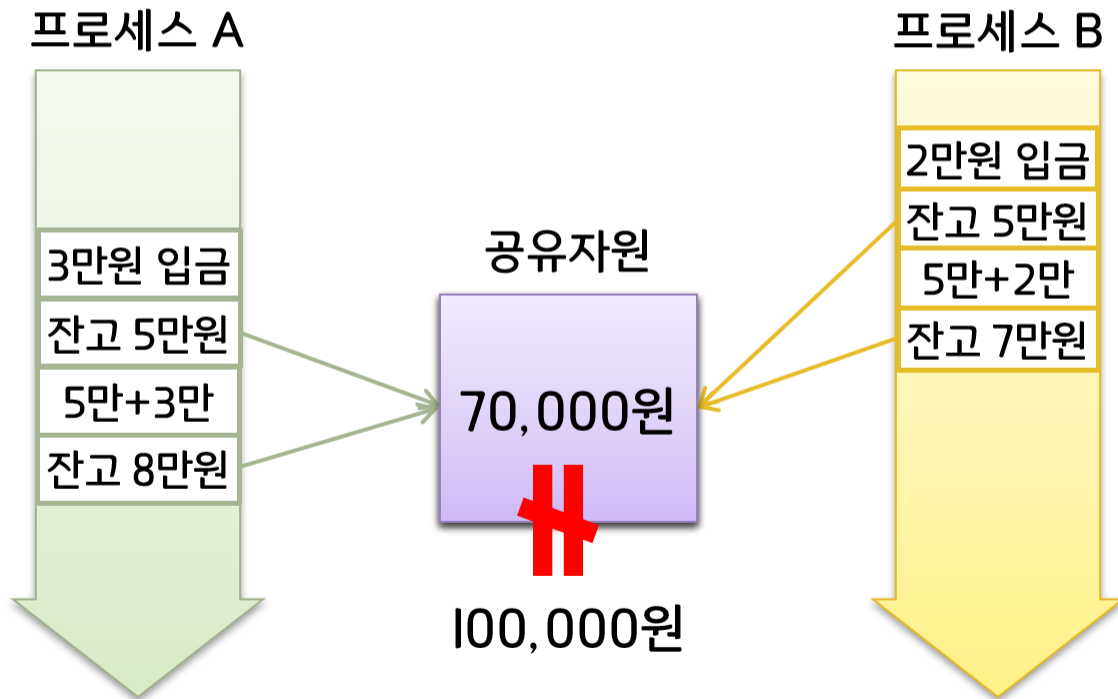
# 프로세스 간의 병행성

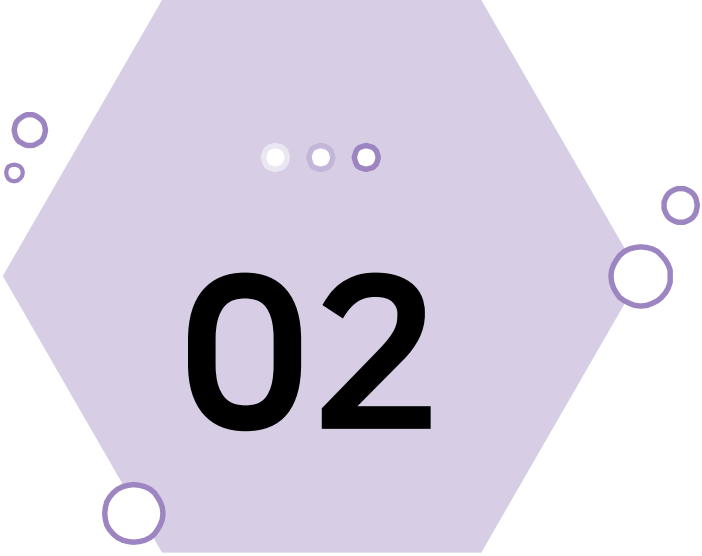
## 비동기 병행 프로세스



# 프로세스 간의 병행성

## 비동기 병행 프로세스





02

# 동기화와 임계영역

# ○ 동기화와 임계영역

## ■ 프로세스 동기화

- 2개 이상의 프로세스에 대한 처리순서를 결정하는 것
- 예: 동시에 사용할 수 없는 공유자원, 한 프로세스의 처리 결과에 따라 다른 프로세스의 처리가 영향을 받는 경우

## ■ 임계영역

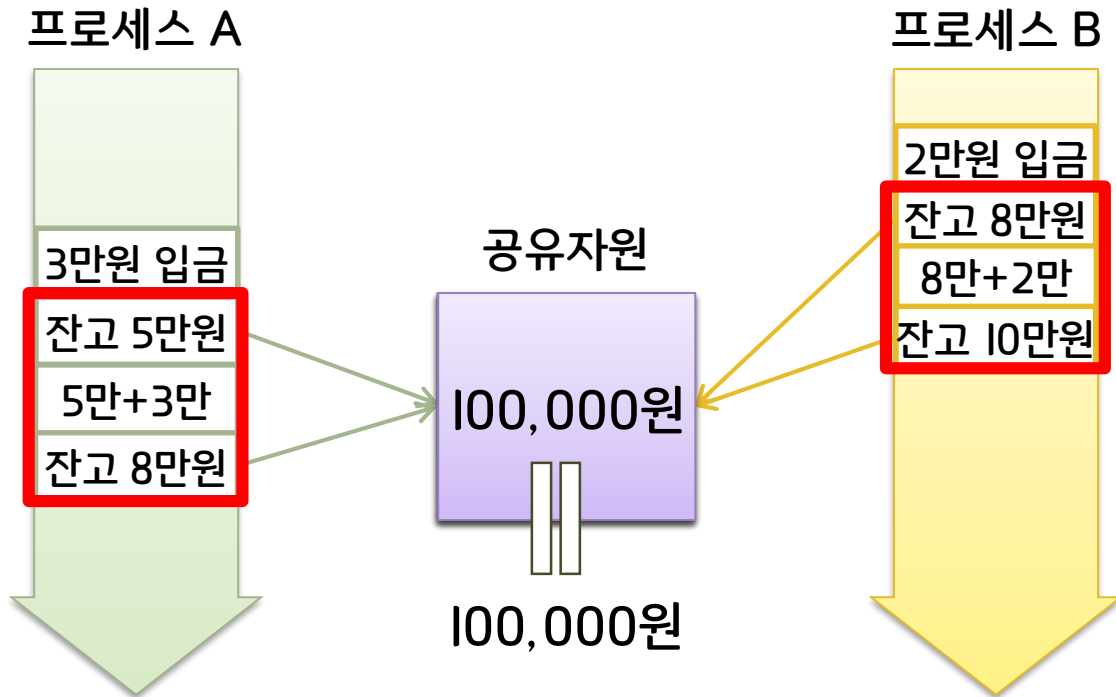
- 2개 이상의 프로세스가 동시에 액세스하면 안 되는 공유자원을 액세스하는 코드 영역

## ■ 상호배제

- 2개 이상의 프로세스가 동시에 임계영역에 진입하지 못하도록 하는 것

# 임계영역의 예

## 상호배제를 통한 프로세스 동기화



# 임계영역을 갖는 프로세스의 일반적 구조

repeat

진입영역

**임계영역**

해제영역

잔류영역

until false;

# ○ 임계영역 문제 해결을 위한 요구조건

## ■ 상호배제

- 한 프로세스가 임계영역에서 실행 중일 때  
다른 어떤 프로세스도 임계영역에서 실행될 수 없음

## ■ 진행

- 임계영역에서 실행 중인 프로세스가 없고 여러 프로세스가  
임계영역에 진입하고자 할 때 그 중에서 적절히 한 프로세스를  
결정해야 하며 이 결정은 무한정 미룰 수 없음

## ■ 제한된 대기

- 한 프로세스가 임계영역 진입 요청을 한 후 수락될 때까지  
다른 프로세스가 임계영역 진입을 허가 받는 횟수는 제한이 있어야 함

# 임계영역 문제 해결을 위한 도구

- Test-and-Set
- 세마포어



# Test-and-Set

## ■ Test-and-Set 명령어 (TS 명령어)

- 상호배제의 하드웨어적 해결 방법
- 분리가 불가능한 단일 기계 명령어(원자적으로 수행)

```
function Test_and_Set(var target: boolean): boolean;  
begin  
    Test_and_Set := target;  
    target := true;  
end
```

# Test-and-Set

## ■ 상호배제의 구현

```
repeat
```

**임계영역**

**잔류영역**

```
until false;
```

# Test-and-Set

## ■ 상호배제의 구현

```
repeat
```

```
    while Test_and_Set(lock) do skip;
```

**임계영역**

**잔류영역**

```
until false;
```

# Test-and-Set

## ■ 상호배제의 구현

- lock의 초깃값은 false

```
repeat
```

```
    while Test_and_Set(lock) do skip;
```

**임계영역**

```
    lock := false;
```

**잔류영역**

```
until false;
```

# Test-and-Set

## ■ 상호배제의 구현

프로세스1

repeat

```
while Test_and_Set(lock)
do skip;
```

**임계영역**

```
lock:= false;
```

**잔류영역**

until false;

프로세스2

repeat

```
while Test_and_Set(lock)
do skip;
```

**임계영역**

```
lock:= false;
```

**잔류영역**

until false;

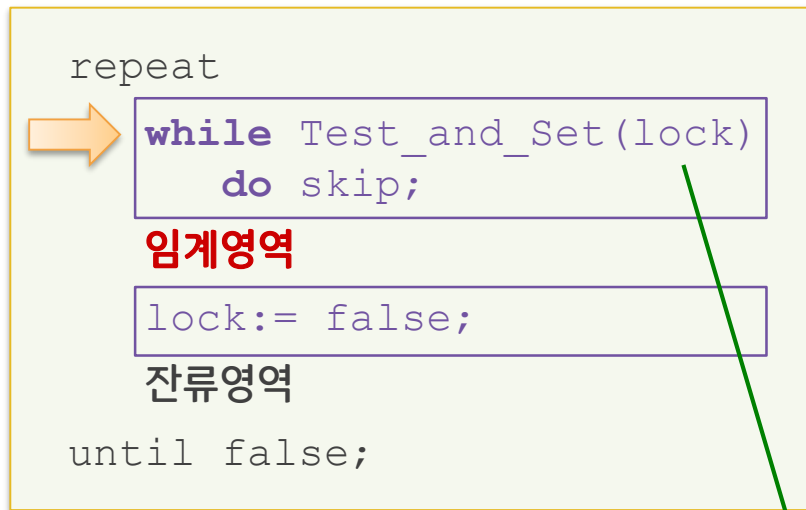
lock

*false*

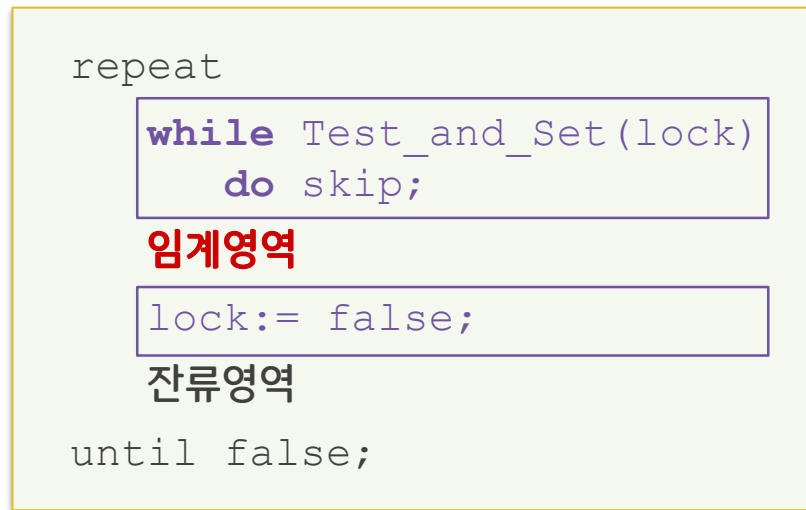
# Test-and-Set

## ■ 상호배제의 구현

프로세스1



프로세스2

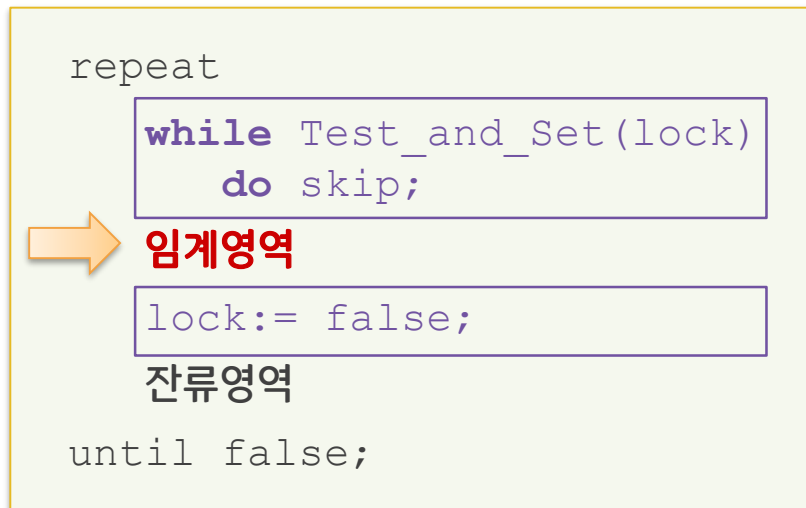


lock  
*true*

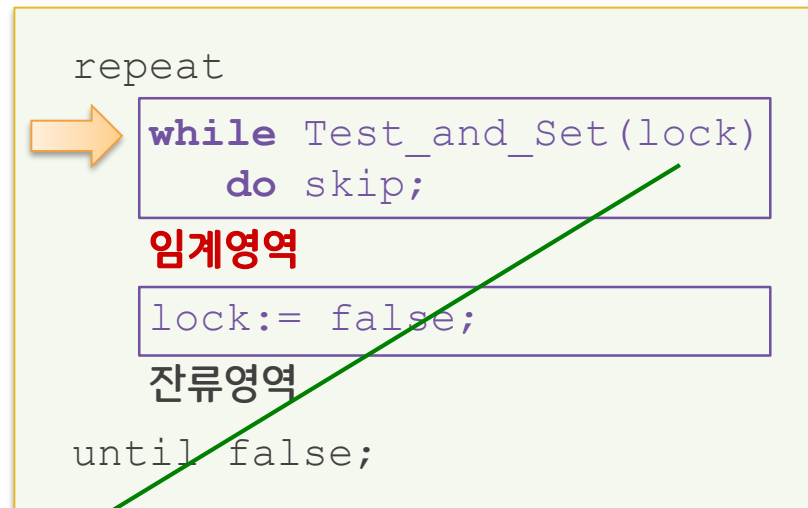
# Test-and-Set

## ■ 상호배제의 구현

프로세스1



프로세스2

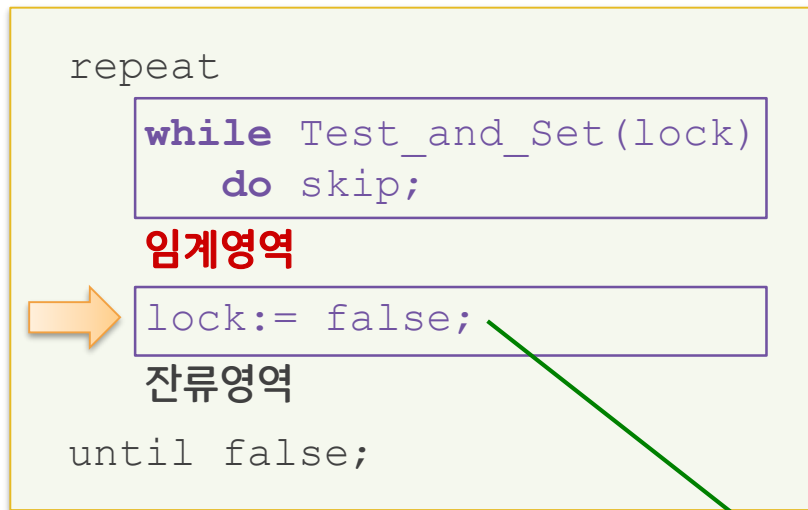


lock  
*true*

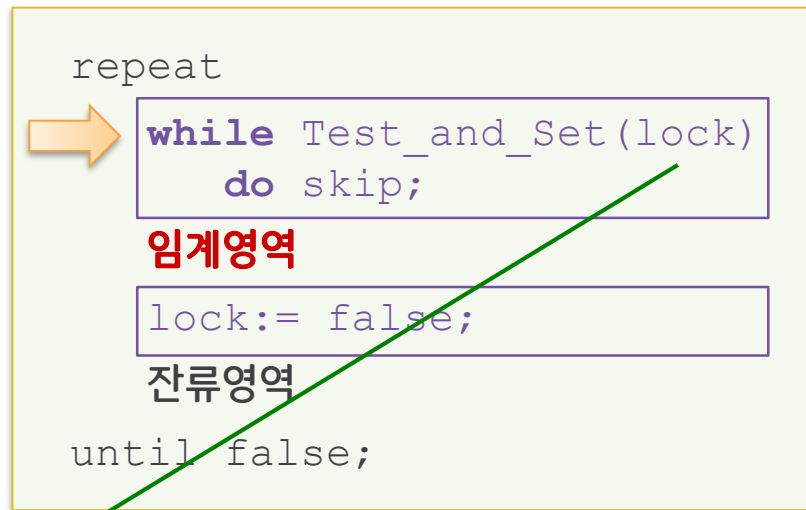
# Test-and-Set

## ■ 상호배제의 구현

프로세스1



프로세스2



lock

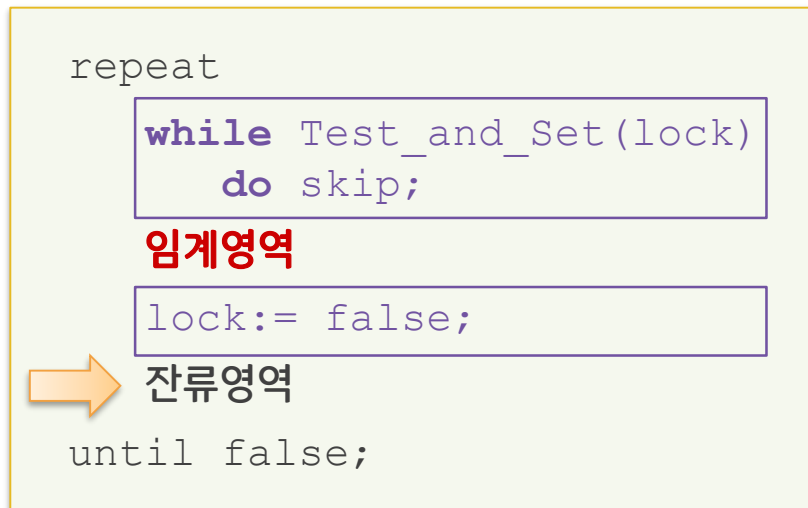
*false*



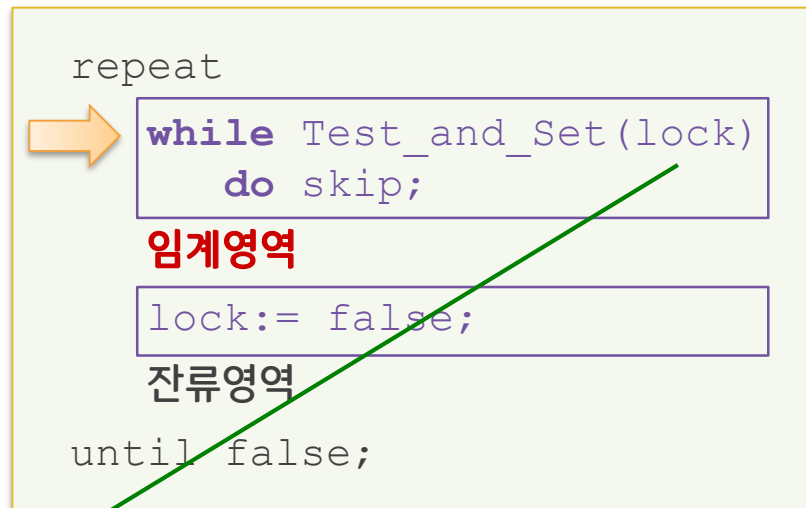
# Test-and-Set

## ■ 상호배제의 구현

프로세스1



프로세스2

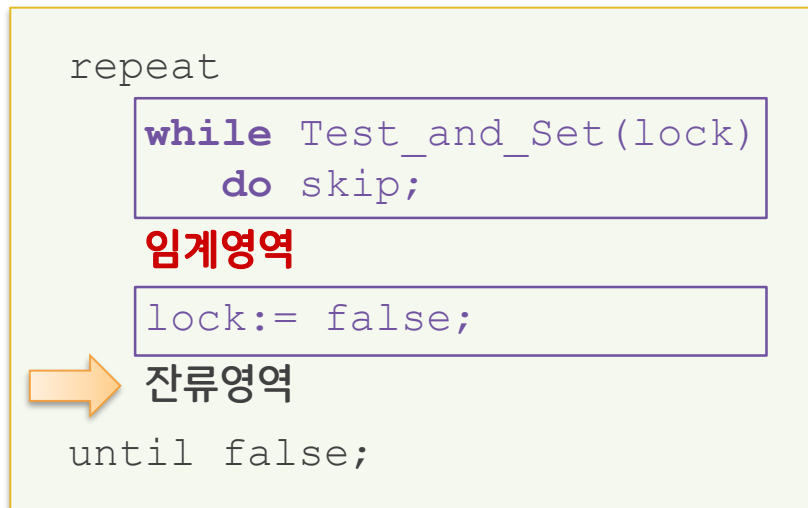


lock  
*false*

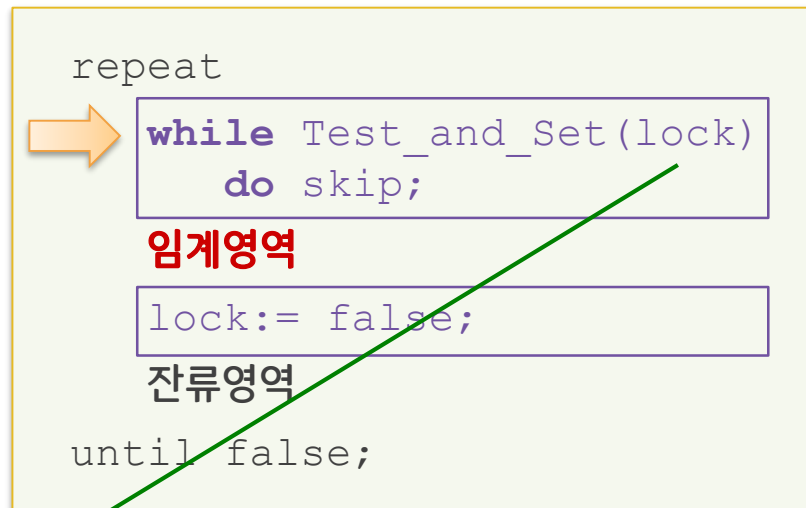
# Test-and-Set

## 상호배제의 구현

프로세스1



프로세스2



lock

*true*

# Test-and-Set

## ■ 상호배제의 구현

프로세스1

repeat

```
while Test_and_Set(lock)
do skip;
```

**임계영역**

```
lock:= false;
```



**잔류영역**

until false;

프로세스2

repeat

```
while Test_and_Set(lock)
do skip;
```



**임계영역**

```
lock:= false;
```

**잔류영역**

until false;

lock

*true*

# Test-and-Set

## ■ 문제점

- 많은 프로세스가 임계영역에 들어가기를 원할 때 기아가 발생할 수 있음
- Busy waiting을 함으로써 다른 작업이 사용할 수 있는 CPU 사이클을 낭비

### ★ 기아(starvation)

프로세스가 필요한 자원할당을 받지 못하고 계속적으로 대기하게 되는 상황

# ○ 세마포어

## ■ 세마포어(semaphore)

- Dijkstra가 제안한 동기화 도구
- 세마포어  $s$ : 사용 가능한 자원의 수 또는 잠김/열림 등의 상태를 나타내는 값을 저장하는 정수형 공용변수
- 세마포어  $s$ 는 두 표준단위 연산  $P$ 와  $V$ 에 의해서만 접근됨

»  $P(s)$ : 검사, 감소시키려는 시도

```
if (  $s > 0$  ) then
```

```
     $s := s-1$ ;
```

```
else
```

현재의 프로세스 대기;

»  $V(s)$ : 증가

```
if (1개 이상의 프로세스가 대기 중) then
```

```
    그 중 1개의 프로세스만 진행;
```

```
else
```

```
     $s := s+1$ ;
```

# 세마포어

## ■ 상호배제의 구현

- 세마포어 `mutex`의 초깃값은 1

repeat

`P(mutex) ;`

**임계영역**

`V(mutex) ;`

**잔류영역**

until false;

» **P(s)**

if ( s > 0 ) then

    s := s-1;

else

    현재의 프로세스 대기;

» **V(s)**

if (1개 이상의 프로세스가 대기 중) then  
    그 중 1개의 프로세스만 진행;

else

    s := s+1;

# 세마포어

## 상호배제의 구현

프로세스1

repeat

P (mutex) ;

임계영역

V (mutex) ;

잔류영역

until false;

프로세스2

repeat

P (mutex) ;

임계영역

V (mutex) ;

잔류영역

until false;

프로세스3

repeat

P (mutex) ;

임계영역

V (mutex) ;

잔류영역

until false;

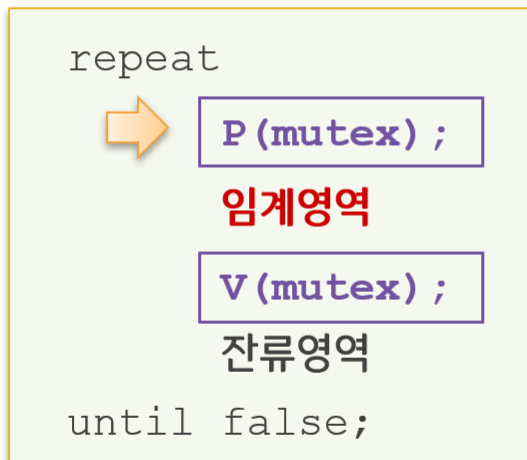
mutex

1

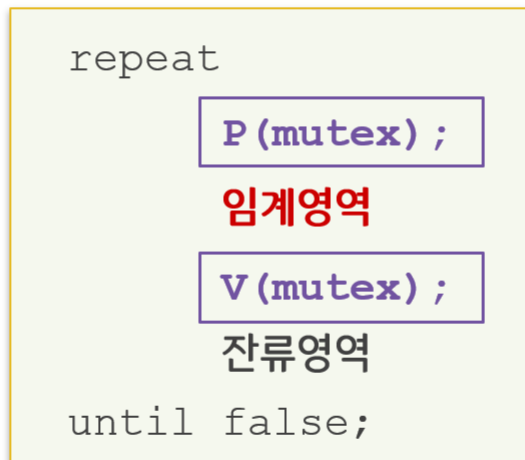
# 세마포어

## 상호배제의 구현

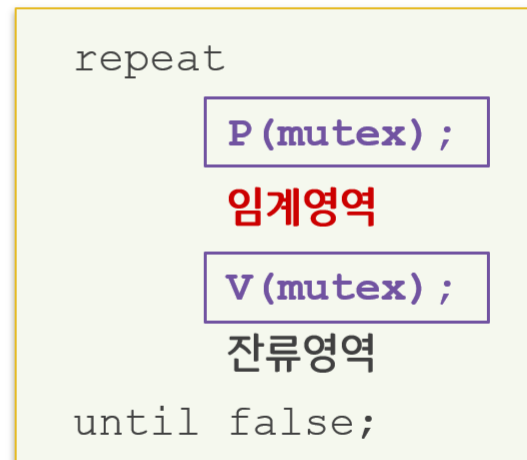
프로세스1



프로세스2



프로세스3

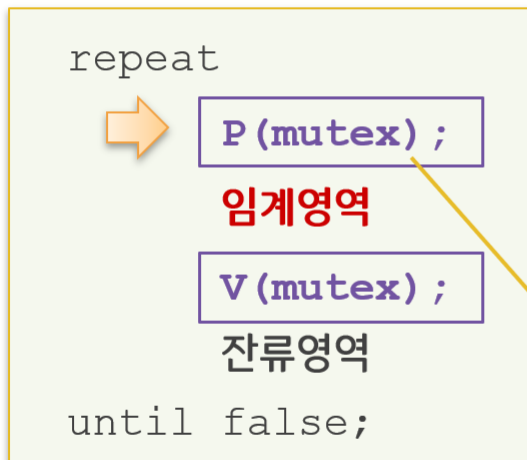




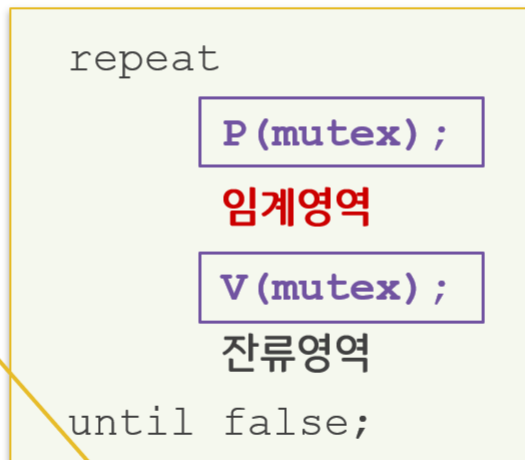
# 세마포어

## 상호배제의 구현

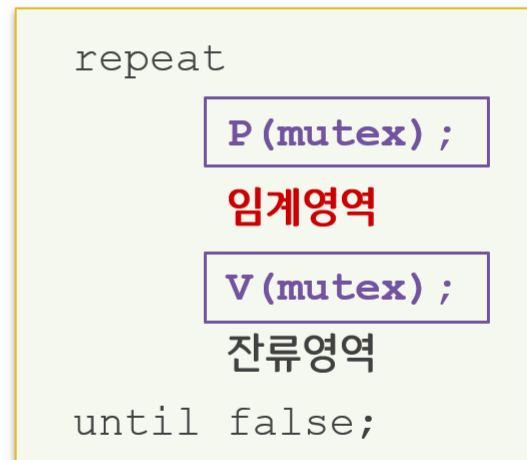
프로세스1



프로세스2



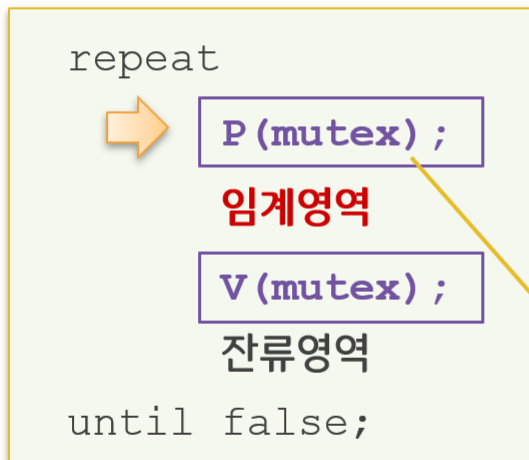
프로세스3



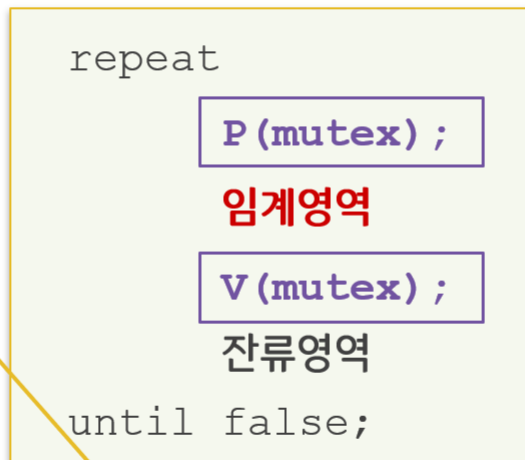
# 세마포어

## 상호배제의 구현

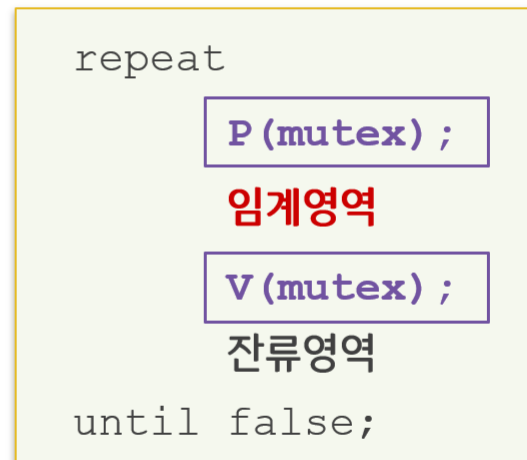
프로세스1



프로세스2



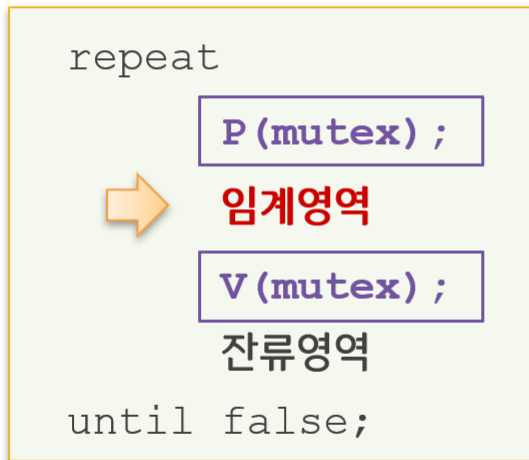
프로세스3



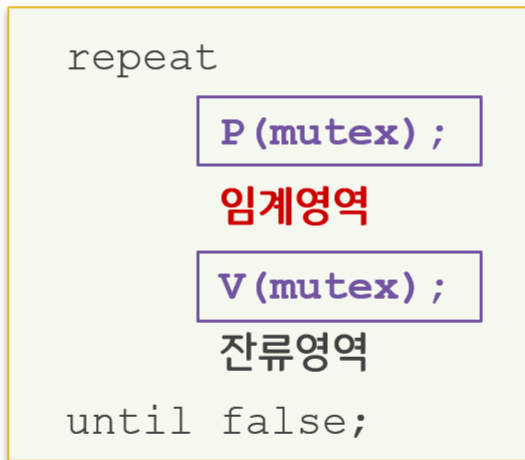
# 세마포어

## 상호배제의 구현

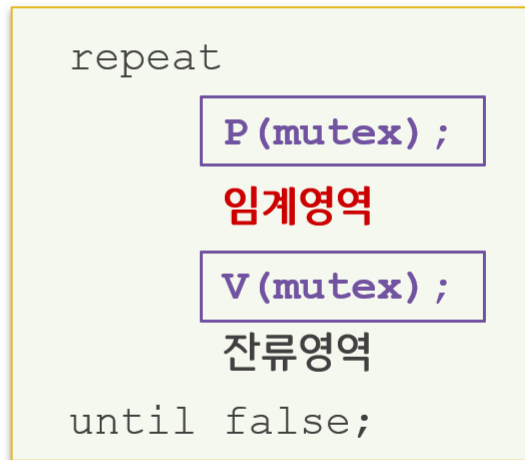
프로세스1



프로세스2



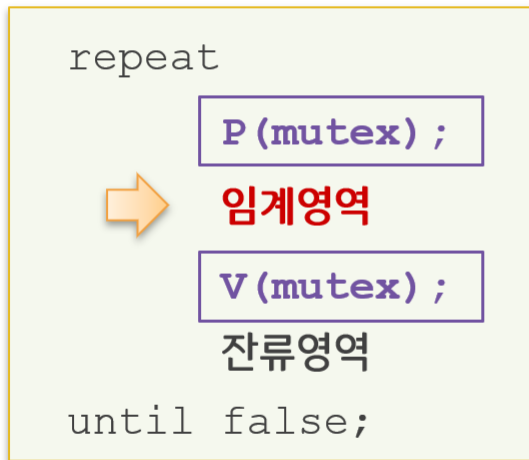
프로세스3



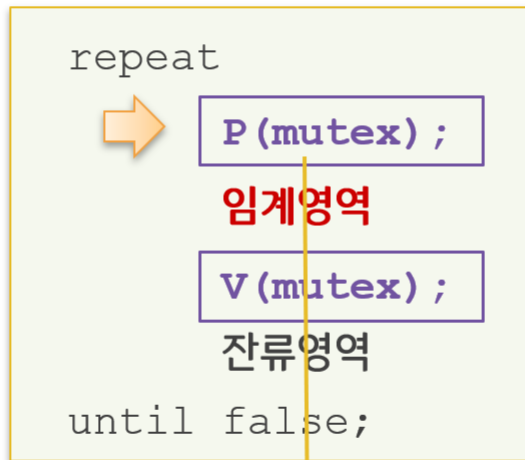
# 세마포어

## 상호배제의 구현

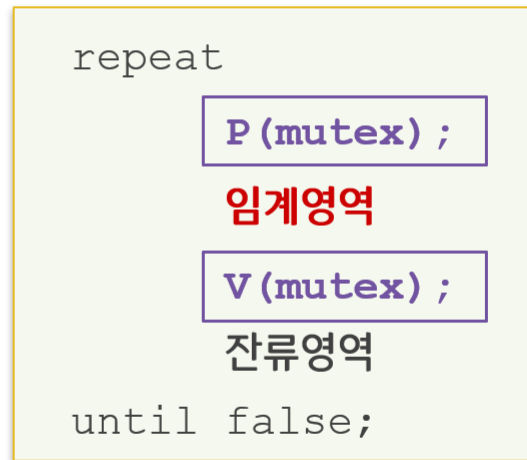
프로세스1



프로세스2



프로세스3



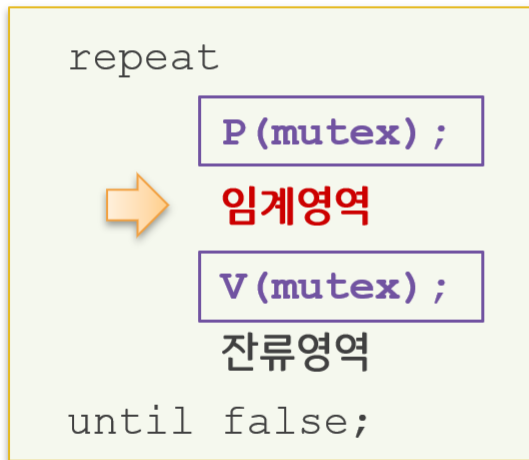
mutex

0

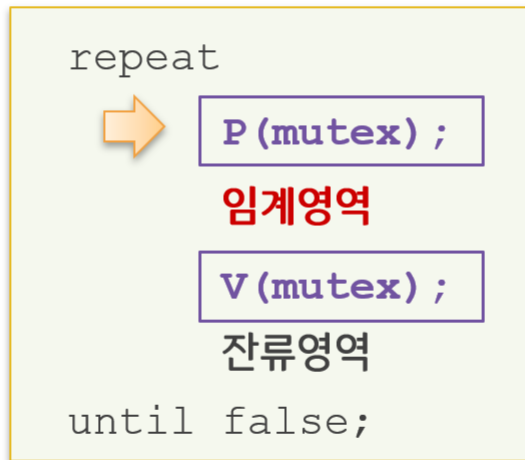
# 세마포어

## 상호배제의 구현

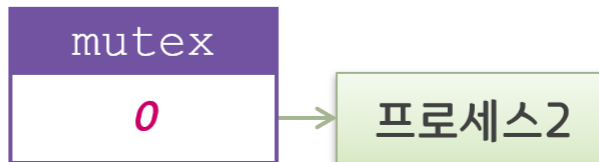
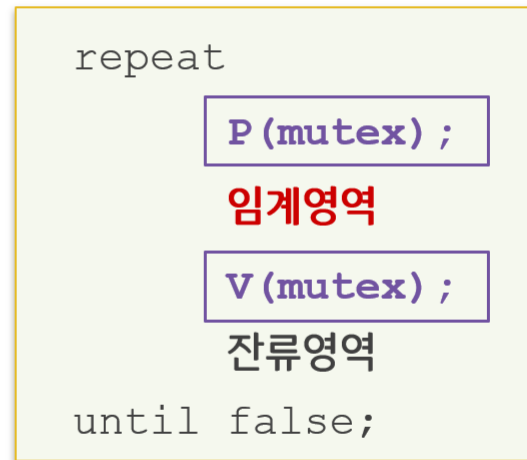
프로세스1



프로세스2



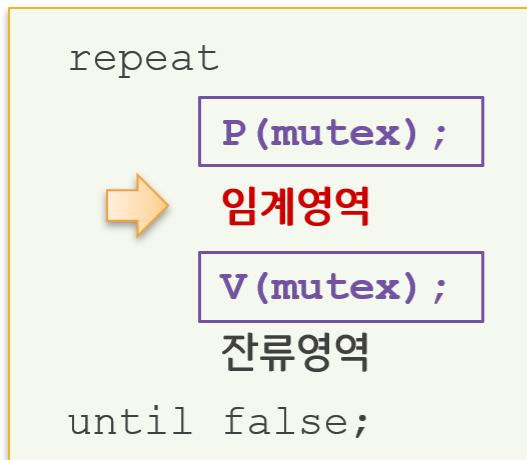
프로세스3



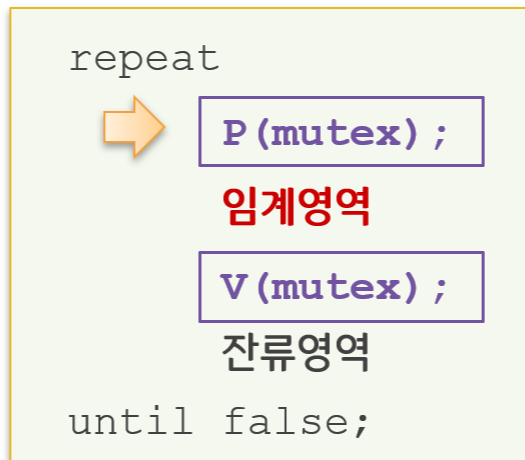
# 세마포어

## 상호배제의 구현

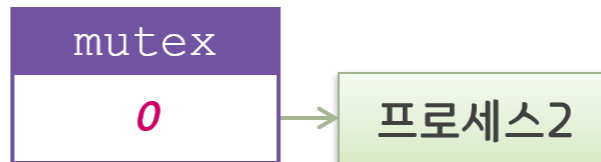
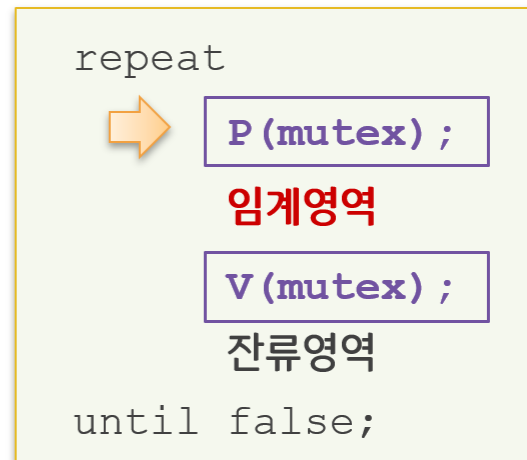
프로세스1



프로세스2

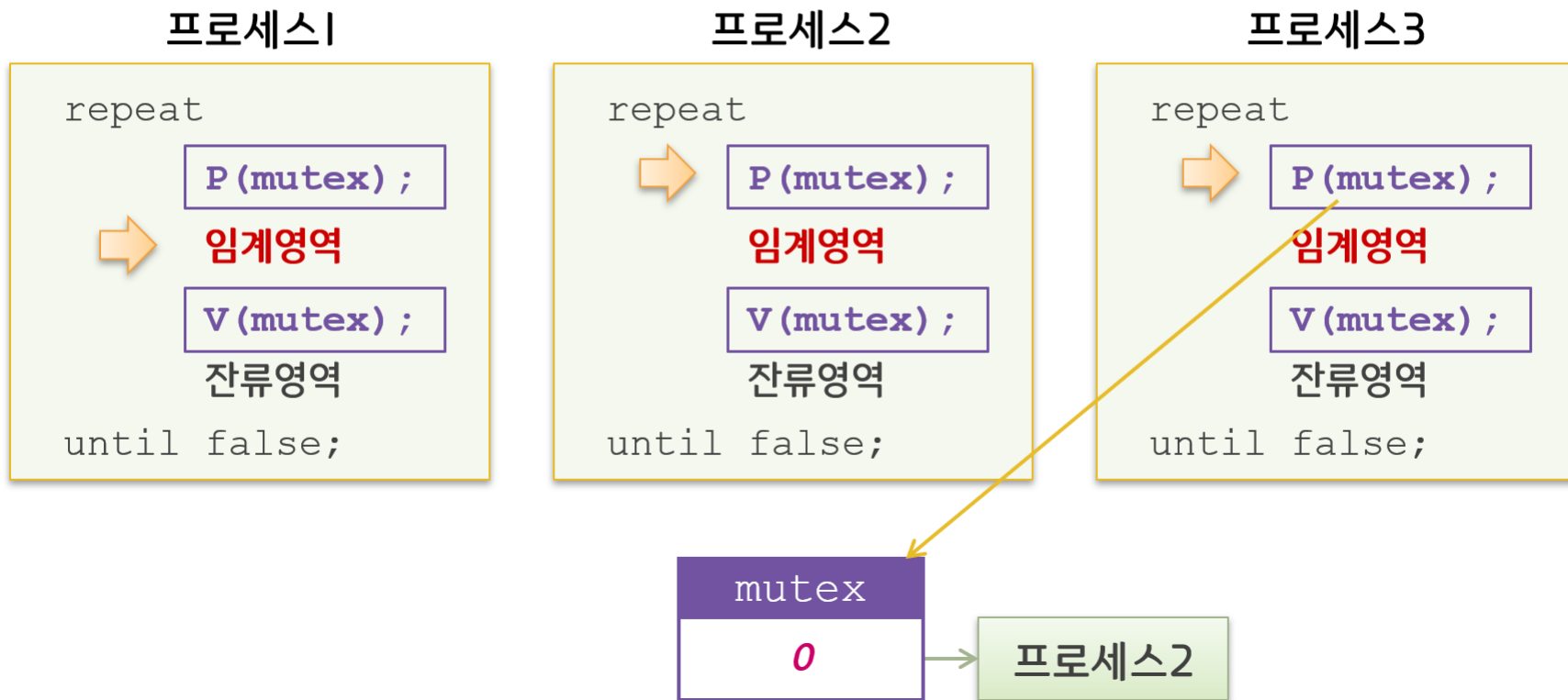


프로세스3



# 세마포어

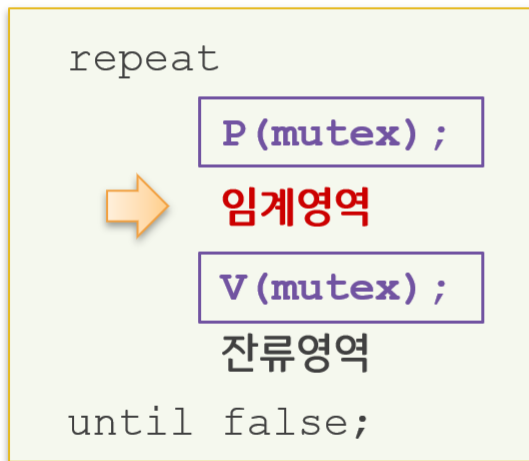
## 상호배제의 구현



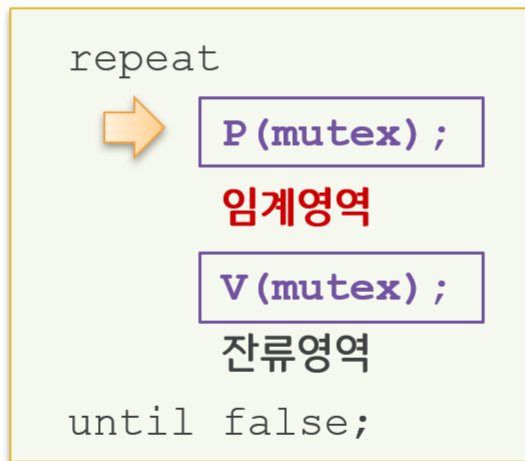
# 세마포어

## 상호배제의 구현

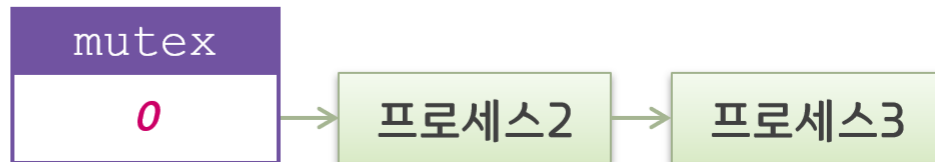
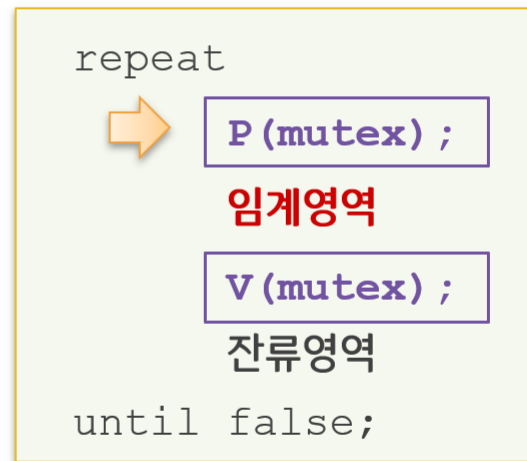
프로세스1



프로세스2



프로세스3

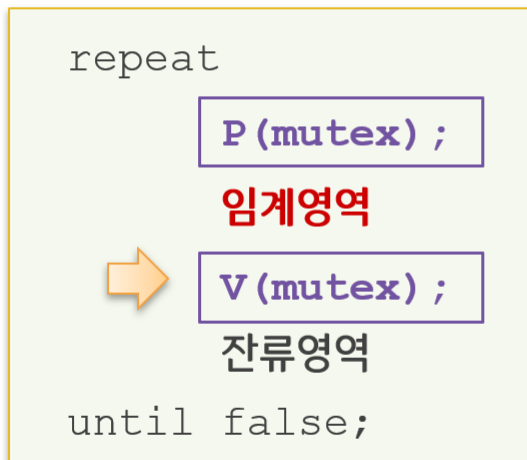




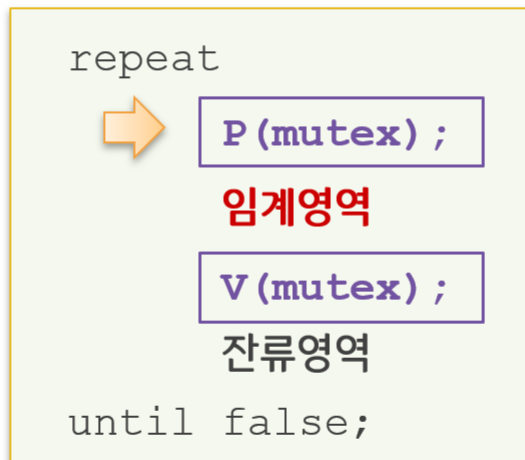
# 세마포어

## 상호배제의 구현

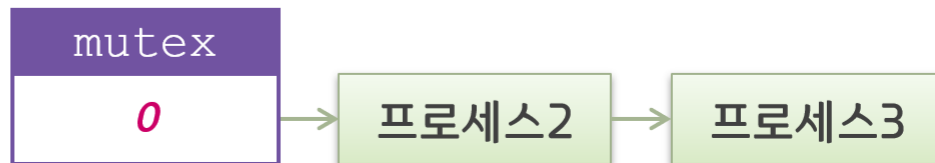
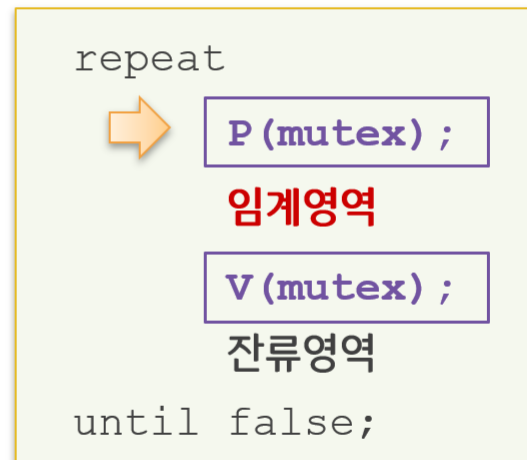
프로세스1



프로세스2

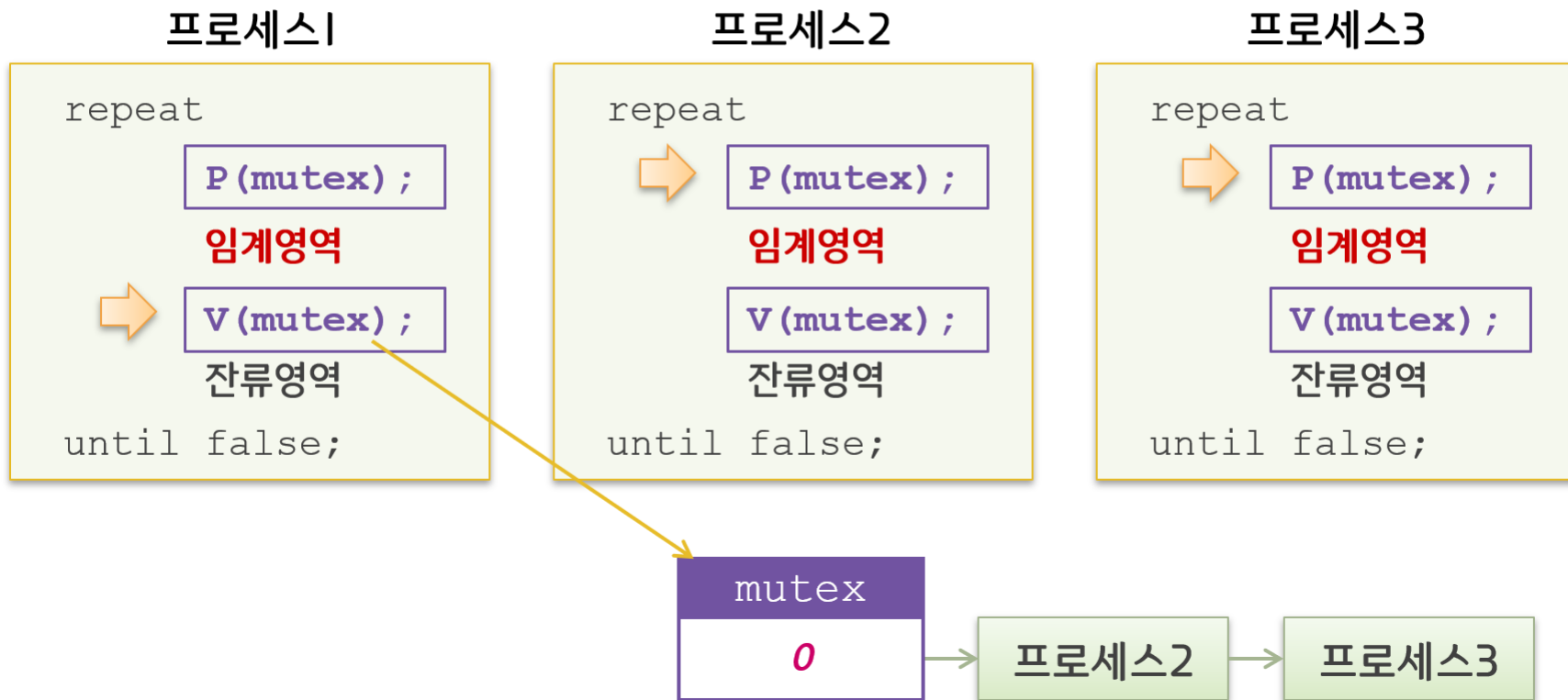


프로세스3



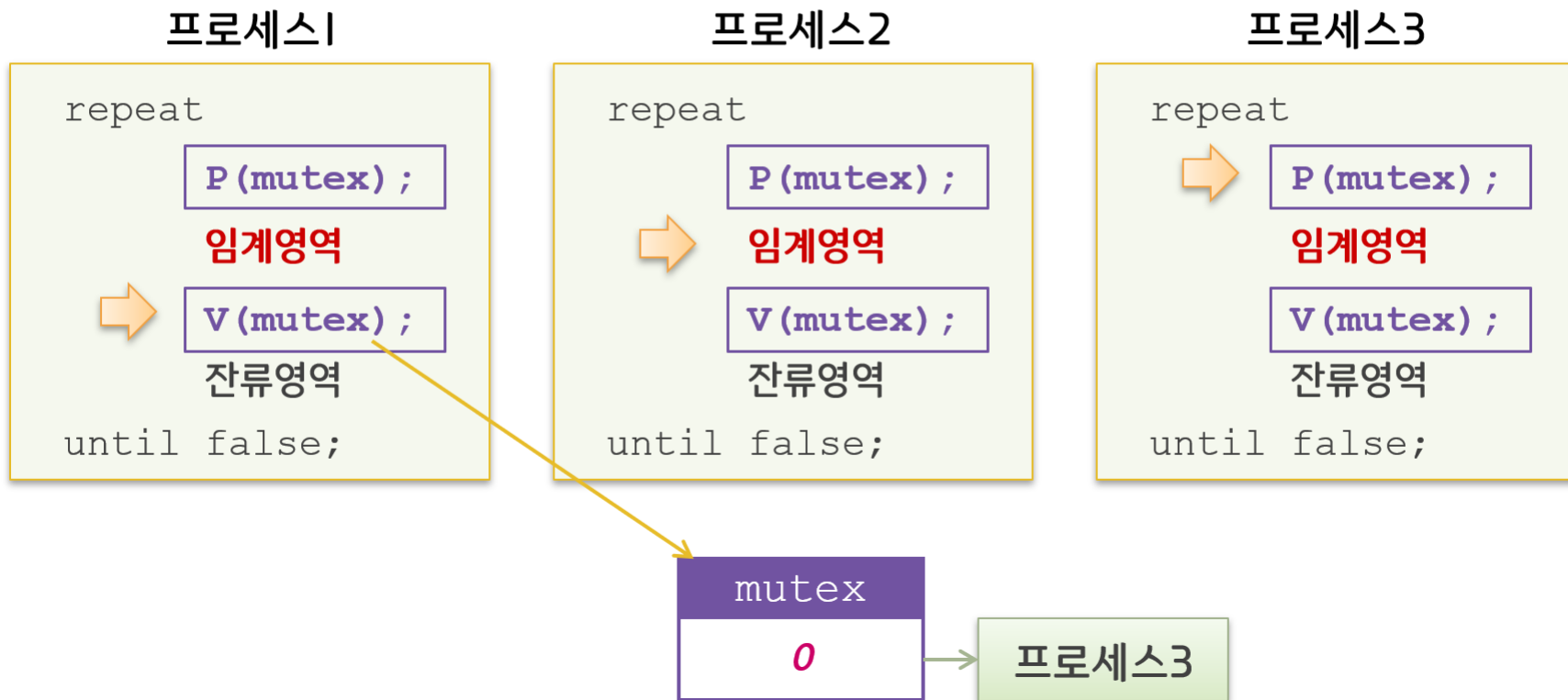
# 세마포어

## 상호배제의 구현



# 세마포어

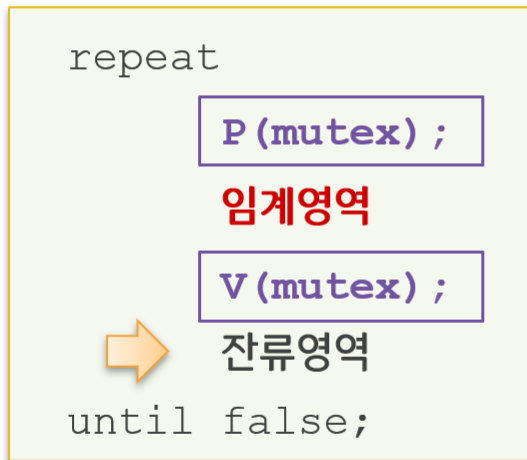
## 상호배제의 구현



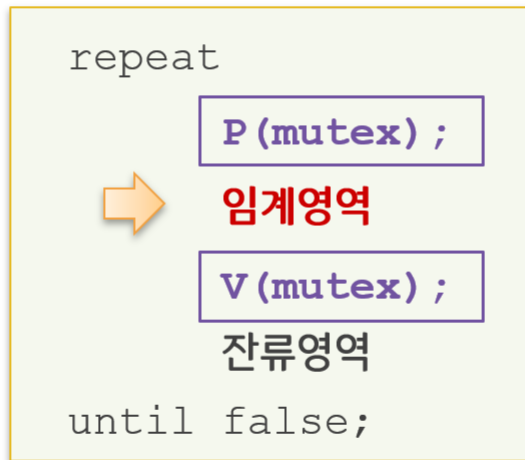
# 세마포어

## 상호배제의 구현

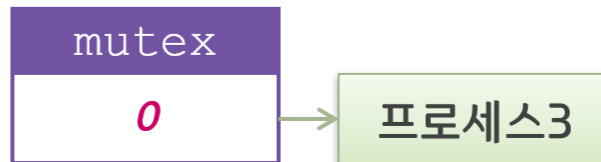
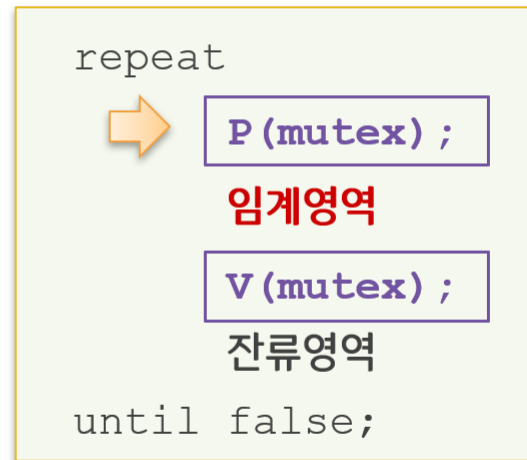
프로세스1



프로세스2



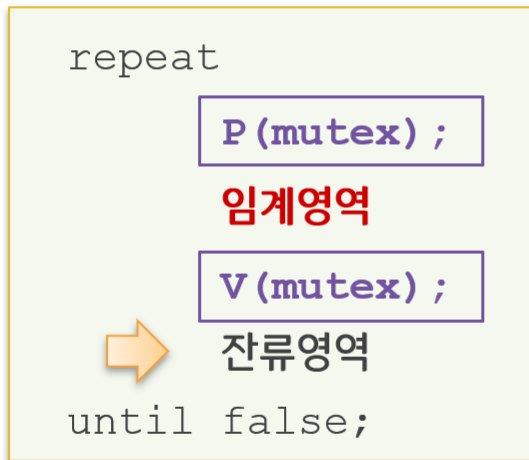
프로세스3



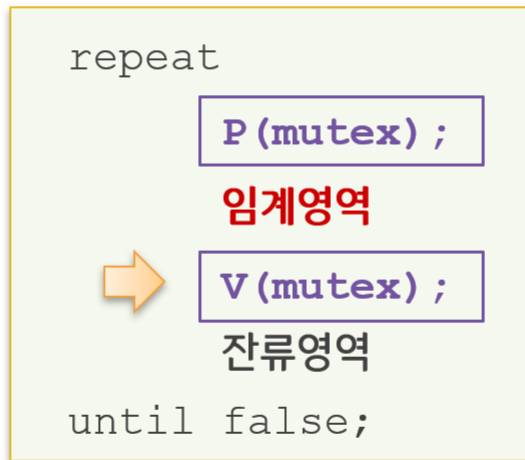
# 세마포어

## 상호배제의 구현

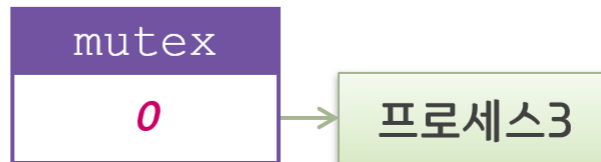
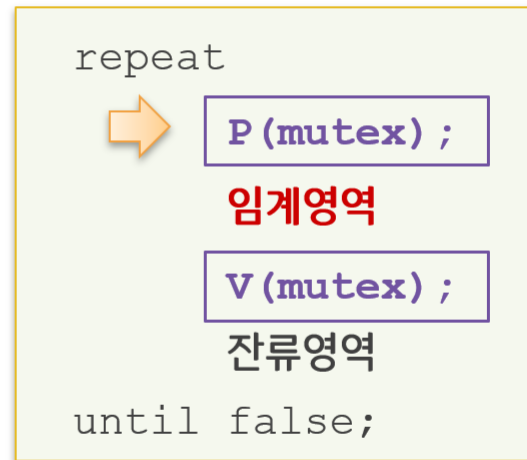
프로세스1



프로세스2



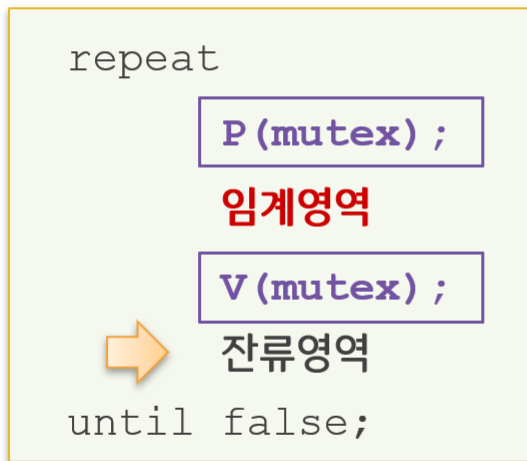
프로세스3



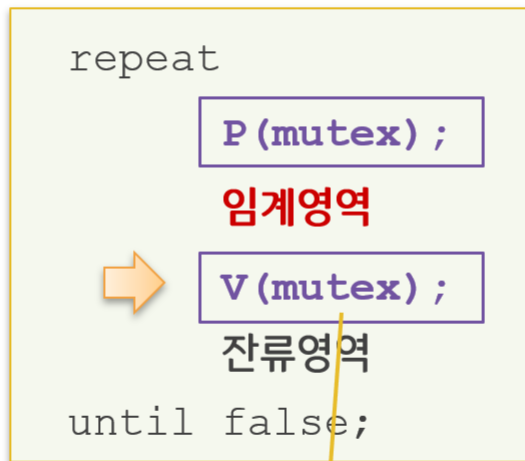
# 세마포어

## 상호배제의 구현

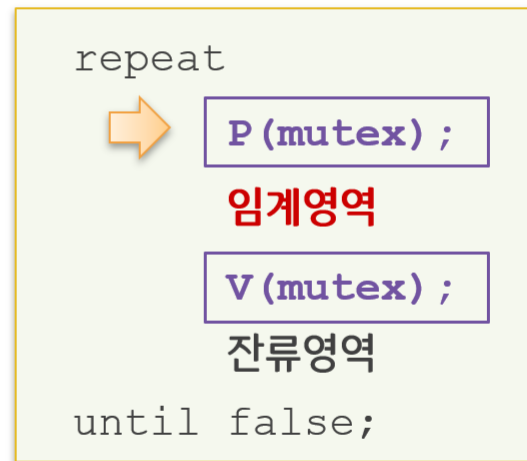
프로세스1



프로세스2



프로세스3

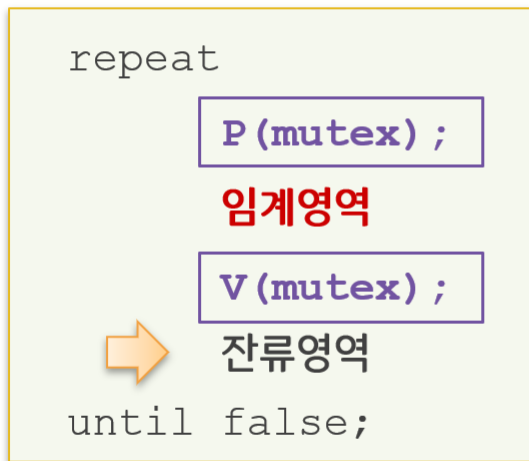


프로세스3

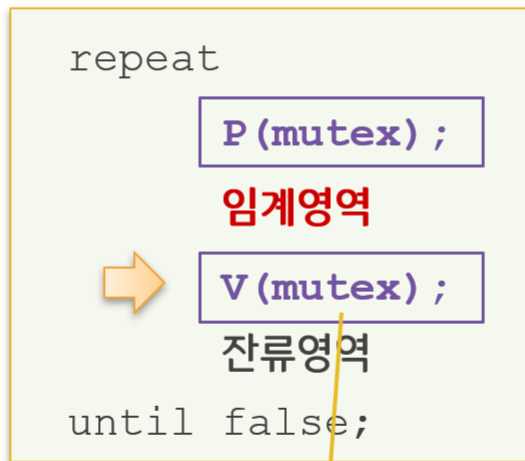
# 세마포어

## 상호배제의 구현

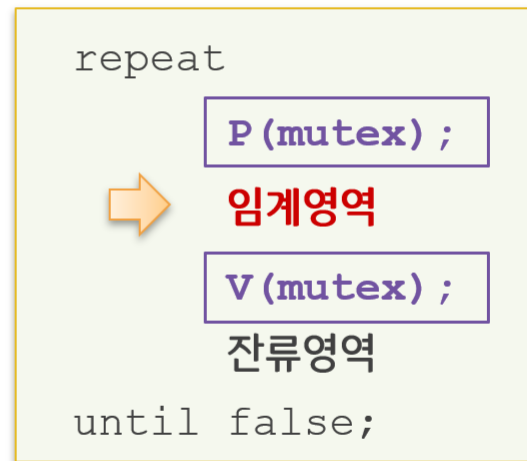
프로세스1



프로세스2



프로세스3



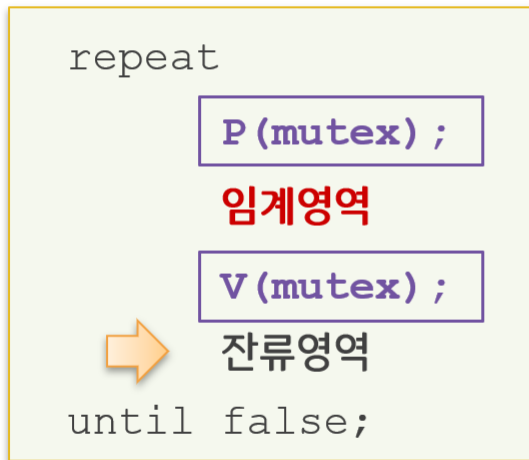
mutex

0

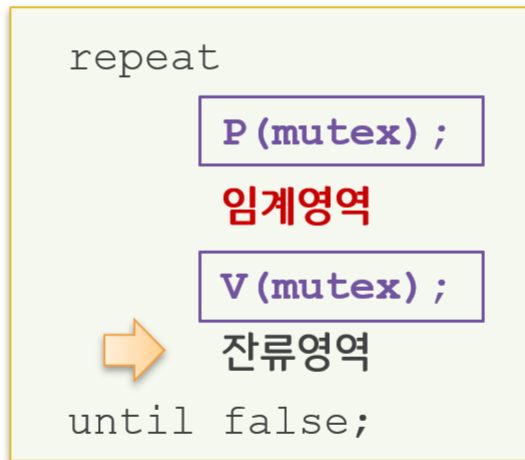
# 세마포어

## 상호배제의 구현

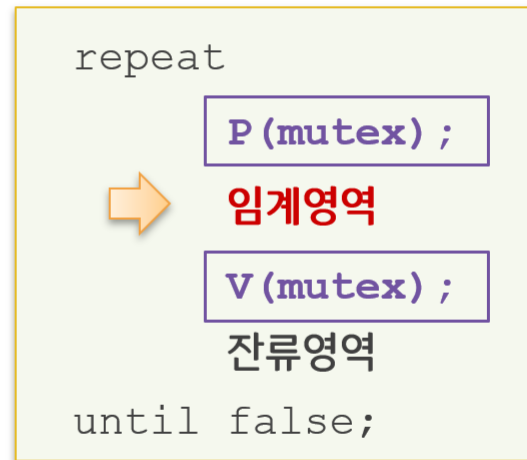
프로세스1



프로세스2



프로세스3

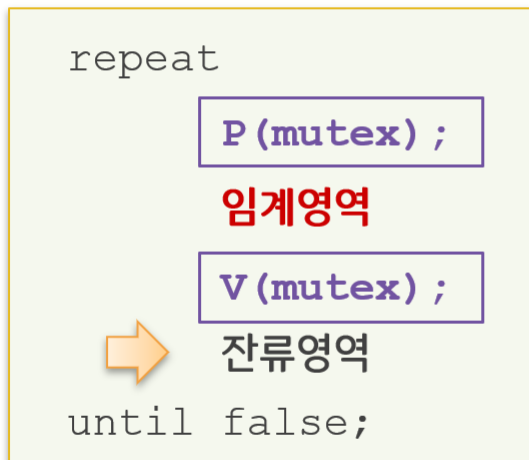




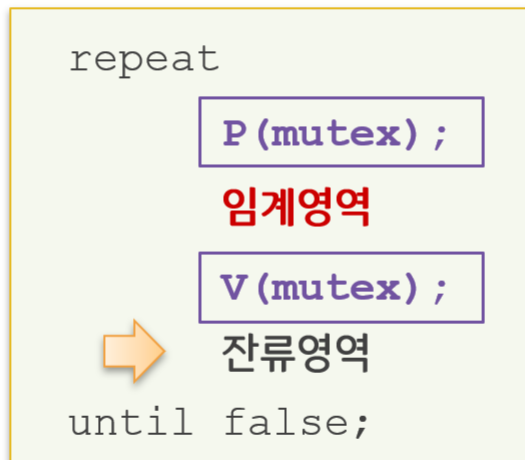
# 세마포어

## 상호배제의 구현

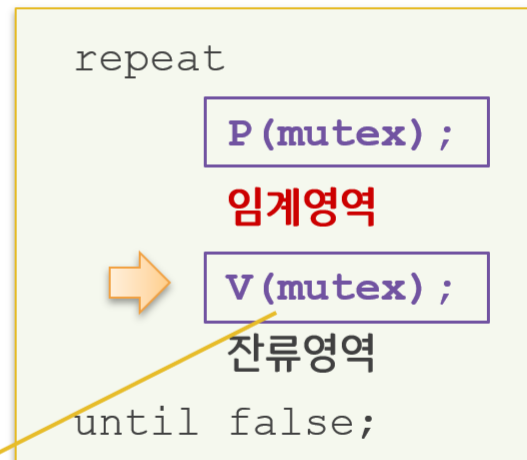
프로세스1



프로세스2



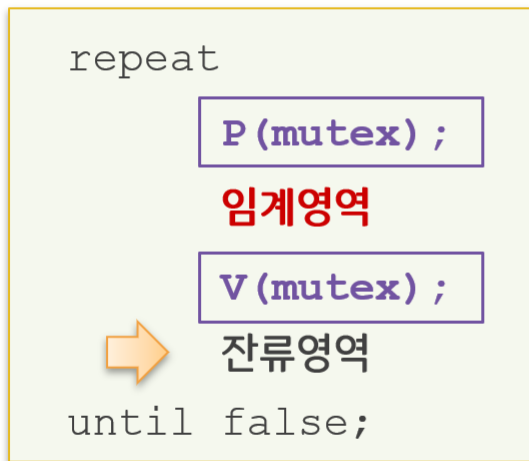
프로세스3



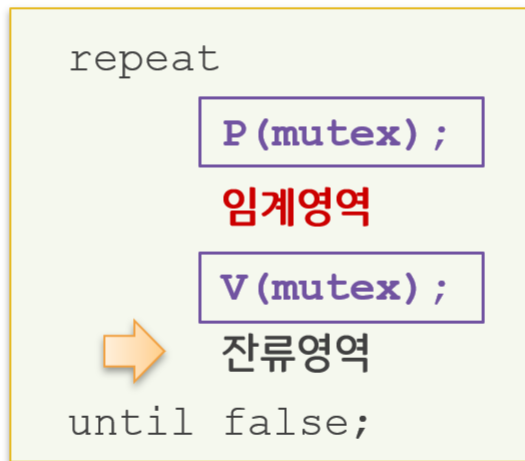
# 세마포어

## 상호배제의 구현

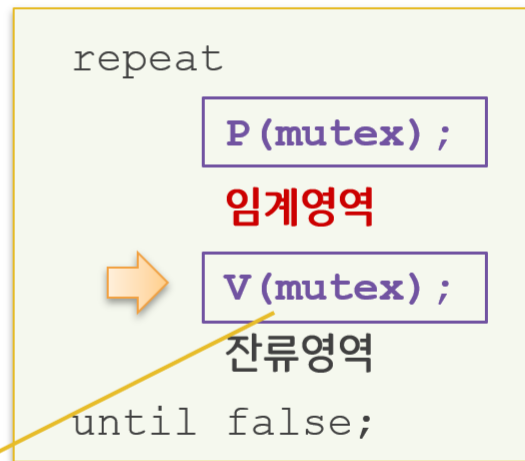
프로세스1



프로세스2



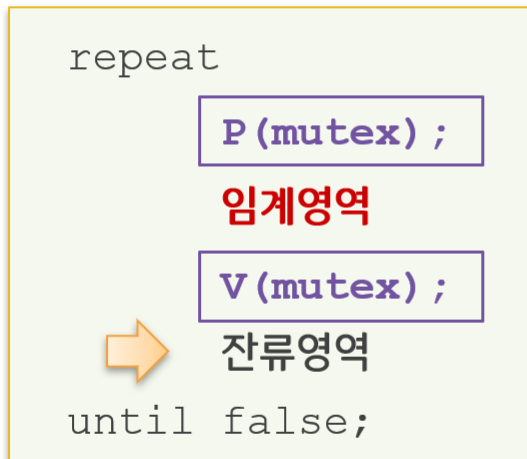
프로세스3



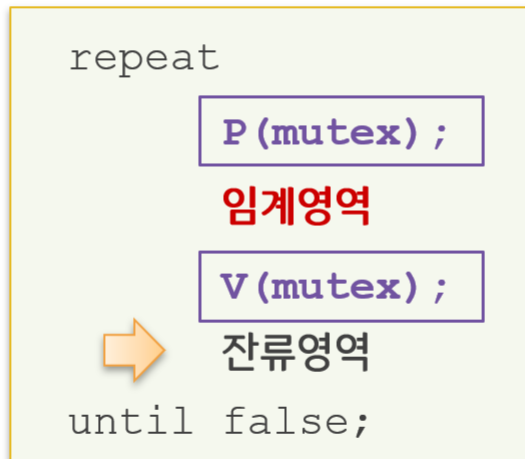
# 세마포어

## 상호배제의 구현

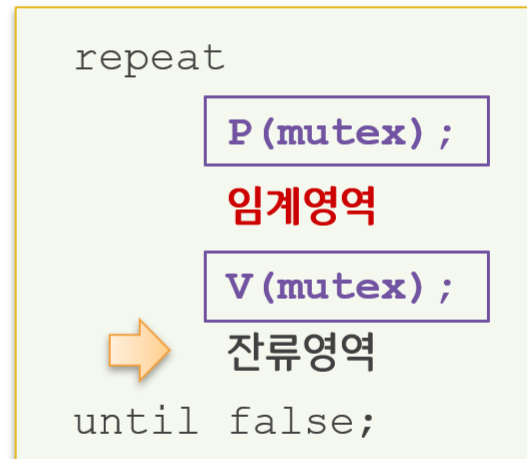
프로세스1



프로세스2



프로세스3



mutex

1

# ○ 세마포어

## ■ 동기화 문제 해결

- 프로세스1이 문장  $s_1$ 을 실행한 후  
프로세스2가 문장  $s_2$ 를 실행하도록 동기화(block/wakeup 프로토콜)

프로세스1

```
.....  
  
S1 ;  
  
V(sync) ;  
  
.....
```

프로세스2

```
.....  
  
P(sync) ;  
  
S2 ;  
  
.....
```

- 세마포어 `sync`의 초깃값은 0

# ○ 세마포어

## ■ 동기화 문제 해결

- 프로세스1이 문장  $s_1$ 을 실행한 후  
프로세스2가 문장  $s_2$ 를 실행하도록 동기화(block/wakeup 프로토콜)

프로세스1

.....

$s_1$ ;

$V(sync)$  ; → wakeup

.....

프로세스2

.....

$P(sync)$  ; → block

$s_2$ ;

.....

- 세마포어  $sync$ 의 초깃값은 0



강의를 마쳤습니다.

다음시간에는  
**5강. 병행 프로세스 II**