06. Process Synchronization & Mutual Exclusion

Process Synchronization (동기화)

- 다중 프로그래밍 시스템
  - 서로 독립적으로 동작하는 여러 개의 프로세스가 존재하는 시스템 ➡️ 현재 사용하는 컴퓨터
  - 프로세스들이 동시에 공유 자원 또는 데이터를 사용하려고 할 때, 문제 발생 가능
- **동기화 (Synchronization)**
  - 문제가 발생하지 않도록 프로세스 간 서로 정보를 공유하고, 동작을 맞추는 것

Asynchronous & Concurrent Processes

- 비동기적 (Asynchronous)
  - 프로세스 간 서로 어떻게 동작하는지 모름
- 병행적 (Concurrent)
  - 여러 프로세스가 동시에 시스템에 존재, 동시에 동작
- **병행 수행 중인 비동기적 프로세스들이 공유 자원에 동시 접근할 때, 문제 발생 가능!**

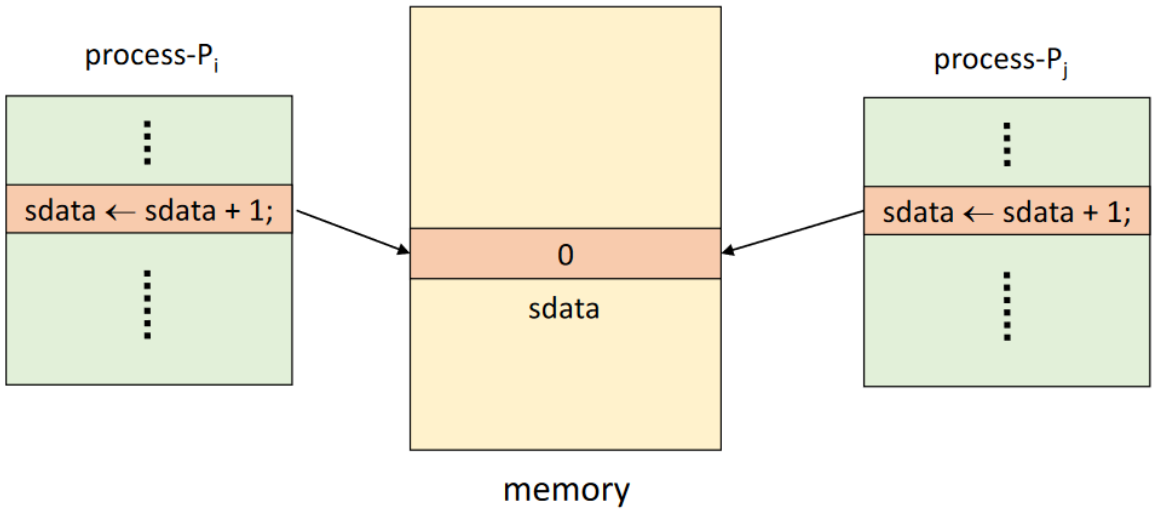
Terminologies

- Shared Data/Critical Data (공유 데이터)
  - 여러 프로세스가 공유하는 데이터
- Critical Section (임계 영역)
  - 공유 데이터를 접근하는 코드 영역 (code segment)
- Mutual Exclusion (상호 배제) ⭐
  - 둘 이상의 프로세스 동시에 임계 영역에 진입하는 것을 막는 것

Critical Section (example)

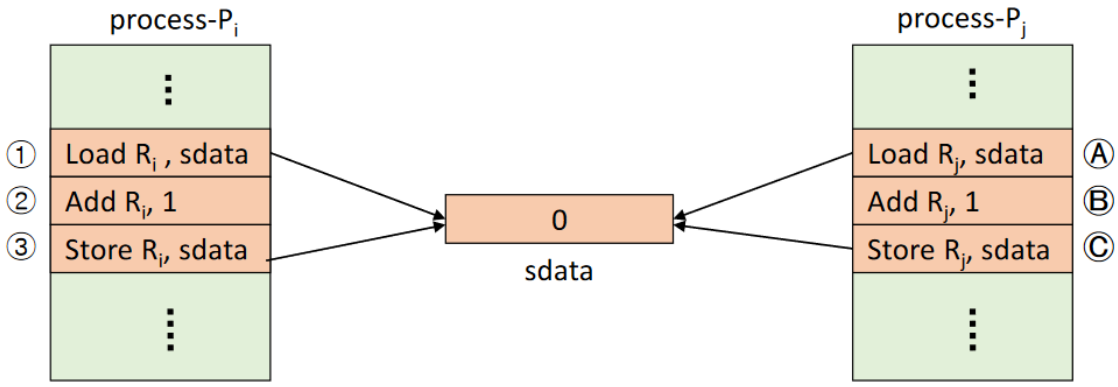
> 💡 **기계어 명령의 특성**
>
> **Atomicity (원자성), Indivisible (분리 불가능)**: 한 기계어 명령 실행 도중 인터럽트 받지 않음

[상황]



- 같은 코드를 가진 프로세스 P_i와 P_j가 공유 데이터 `sdata`를 사용해 작업 수행
  - P_i와 P_j 모두 `sdata` 값을 가져와 `+1` 한 값을 다시 `sdata`에 저장하는 코드
  - 두 프로세스가 작업을 마치면 `sdata` 값은 반드시 2가 될까? ➡️ 아님!

[기계어로 코드를 컴파일했을 때]



- 기계어 명령 번역 결과
  1. 공유 데이터 `sdata`를 읽어서 레지스터 R_i에 저장

2. R_i 값에 1 더하기
3. R_i 값을 `sdata`에 저장

- **Race Condition:** 명령 실행 순서가 달라지면 명령 수행 결과가 달라지는 상태
  - 각 기계어 명령은 atomic하기 때문에 중간에 방해를 받을 수 없지만, 각 명령 사이에 방해를 받을 수 있음
  - ex) 1 → 2 → A → B → C → 3 순서로 실행되면 `sdata` 값은 1이 됨

Mutual Exclusion (상호 배제)

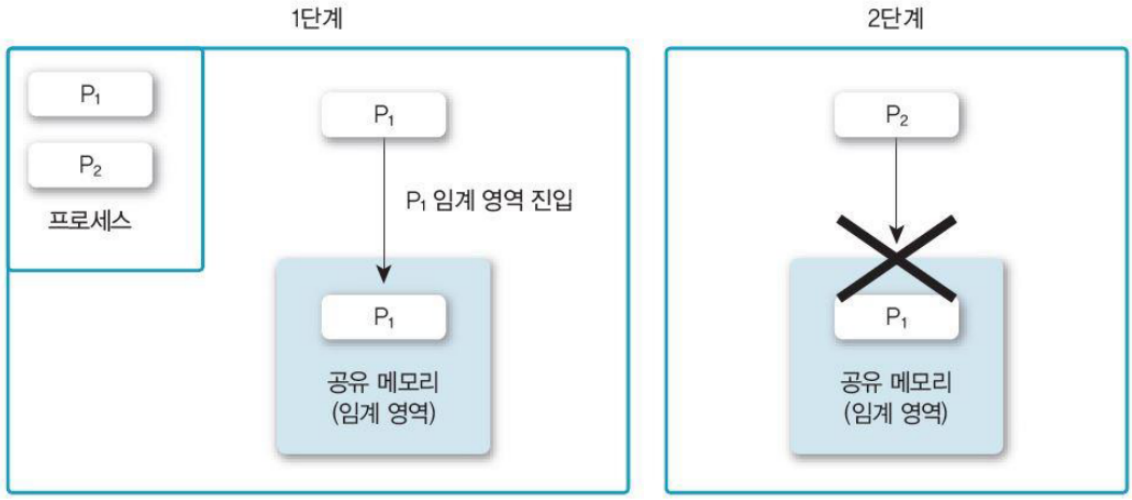
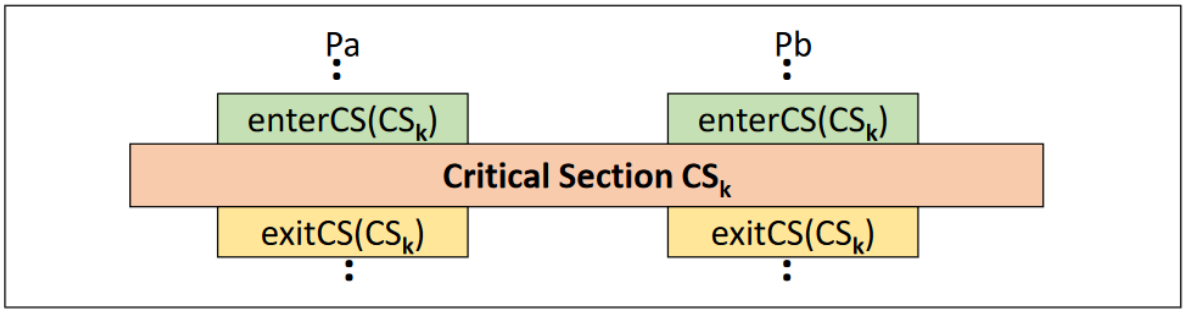


그림 4-13 상호배제의 개념

- Race Condition과 같은 문제를 방지하기 위한 방법
- 어떤 프로세스가 임계 영역에 진입했으면, 다른 프로세스가 들어올 수 없도록 막아주는 것

Mutual Exclusion Methods (Primitives, 기본 연산)



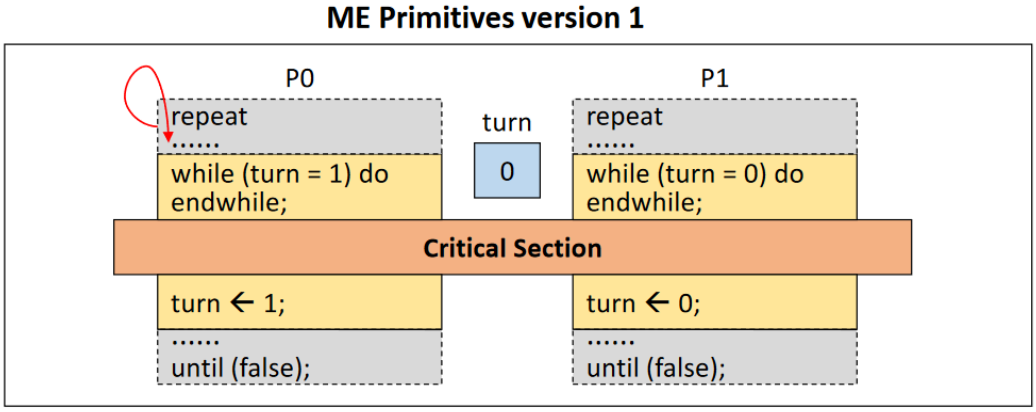
1. enterCS() primitive: 임계 영역 진입 전 검사
    - 다른 프로세스가 임계 영역 내 있는지 검사
2. exitCS() primitive: 임계 영역 퇴장 후 후처리 과정
    - 다른 프로세스가 진입할 수 있도록 시스템에 임계 영역을 벗어난다는 것을 알림

Requirements for Mutual Exclusion Primitives

1. Mutual Exclusion (상호 배제)
    - 임계 영역 내 프로세스가 있으면 다른 프로세스 진입 금지
2. Progress (진행)
    - 임계 영역 내 프로세스 외 다른 프로세스가 임계 영역에 진입하는 것을 방해하면 안 됨
    - 만약 임계 영역이 비어있다면, 어떤 프로세스든 진입할 수 있어야 함
3. Bounded Waiting (한정 대기)
    - 프로세스는 유한 시간 내 임계 영역에 진입할 수 있어야 함 (무한 대기 X)

ME Primitives: Two Process Mutual Exclusion

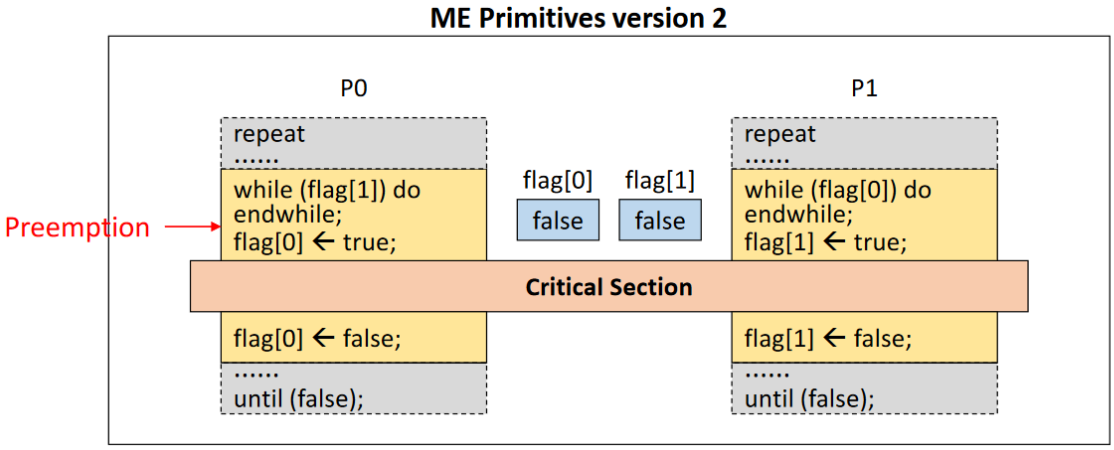
[ME Primitives Version 1]



- 구현
    - `turn`: 임계 영역에 진입할 차례인 프로세스의 인덱스
    - 자기 `turn`이 아닌 경우, while문에서 대기하고 자기 차례가 오면 작업 진행
- Progress 조건 위배

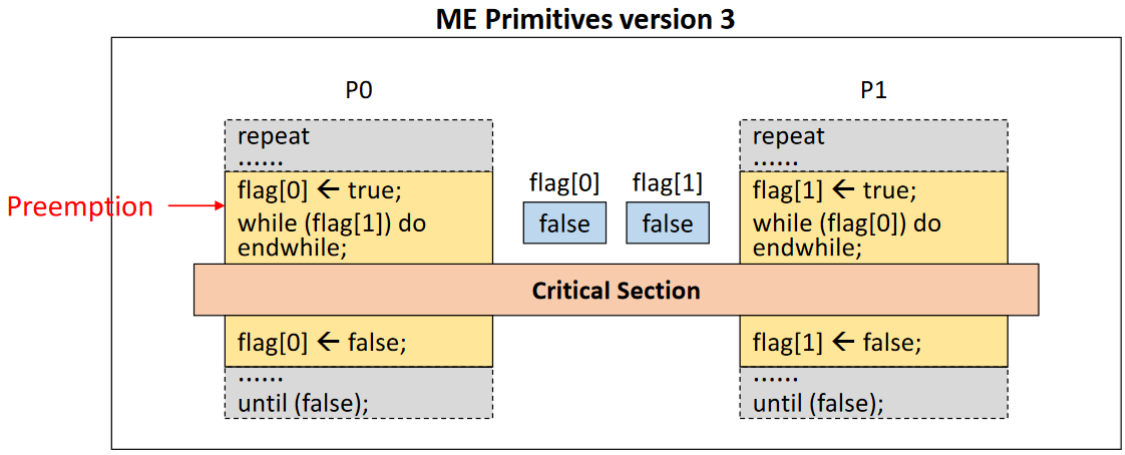
1. `turn = 0`일 때 `P0`이 임계 영역 진입 전 죽는다면, `P1`은 임계 영역에 프로세스가 없지만 자기 차례가 돌아오지 않았기 때문에 `P0`이 임계 영역 들어가기 전까지 진입할 수 없음
   ➡️ Progress 위배
2. `P0`이 작업을 끝내면 `turn = 1`로 초기화하게 됨. 이 때, `P0`이 `P1`보다 임계 영역에 빨리 접근하더라도 자기 차례가 아니기 때문에 진입할 수 없음. 연속 진입 불가 ➡️ Progress 위배

[ME Primitives Version 2]



- 구현
  - `flag`: 해당 프로세스가 임계 영역에 진입했는지 여부 (진입: `true`, 퇴장: `false`)
  - 임계 영역 진입 전, 다른 프로세스의 `flag`를 확인해서 `false`인 경우 진입하고 내 `flag`를 `true`로 설정, 퇴장할 때 `false`로 설정
- Mutual Exclusion 조건 위배
  - `P0`이 `flag[1]`이 `false`인 것을 확인 후 임계 영역에 진입하기 전 Preemption 발생
  - `P1`이 `flag[0]`을 확인했을 때 `false`라 임계 영역 진입해 작업 수행
  - `P0`이 다시 작업 복귀할 때, `flag[1]` 재확인하지 않고 임계 영역 진입
  - 2개의 프로세스가 임계 영역에 존재하게 됨 ➡️ Mutual Exclusion 위배

[ME Primitives Version 3]



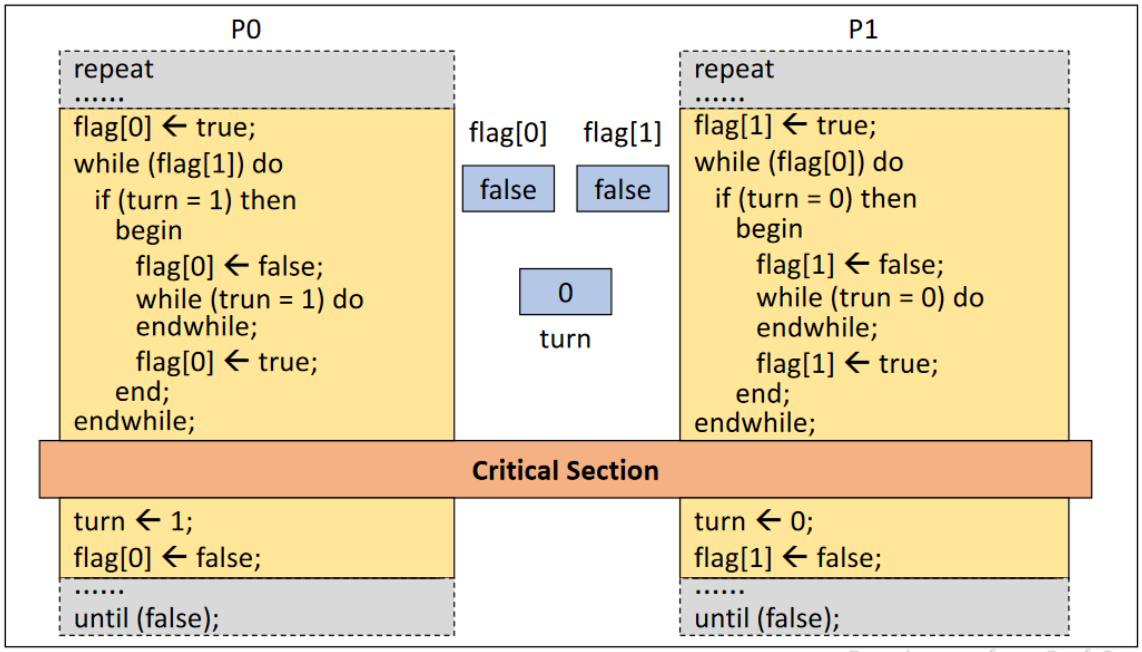
- 구현
  - Version 2에서 `flag` 설정하는 순서 변경
- Progress, Bounded Waiting 조건 위배
  - `P0`이 진입하기 위해 `flag[0] = true` 설정 후 Preemption 발생
  - `P1`이 진입하기 위해 `flag[1] = true` 설정, `flag[0] = true`이기 때문에 대기
  - `P0`이 다시 작업 복귀할 때, `flag[1] = true`이기 때문에 대기
  - 임계 영역이 비었는데도 아무 프로세스도 못 들어감 ➡️ Progress 위배
  - 두 프로세스 모두 무한 대기하게 됨 ➡️ Bounded Waiting 위배

Mutual Exclusion: SW Solution

1. Two Process Mutual Exclusion
   - Dekker's Algorithm
   - Peterson's Algorithm
2. N-Process Mutual Exclusion
   - Dijkstra's Algorithm
   - Knuth's Algorithm, Eisenberg and McGuire's Algorithm, Lamport's Algorithm

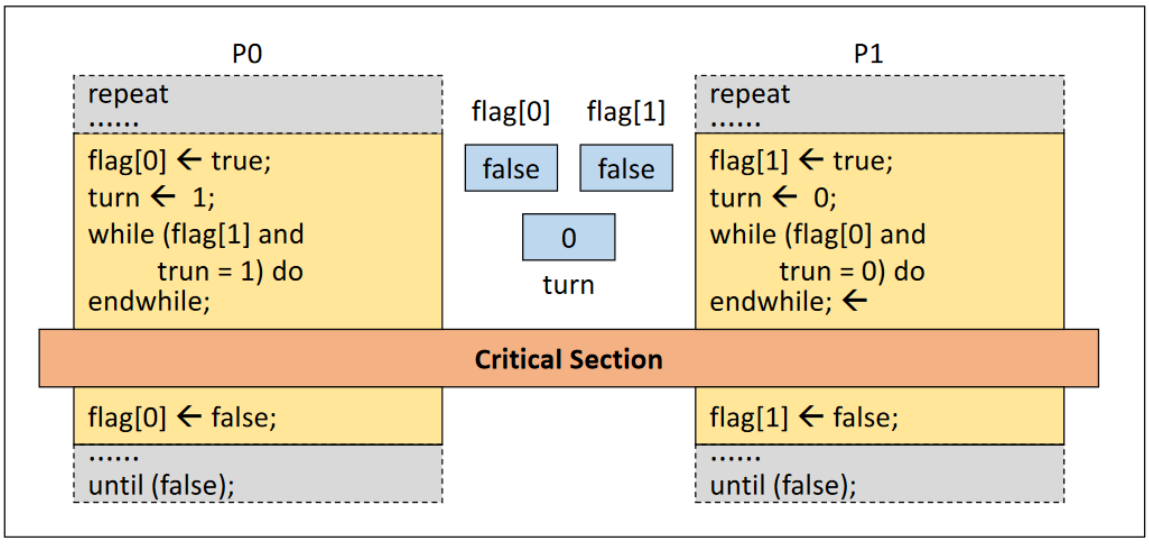
SW Solution: Two Process Mutual Exclusion

Dekker's Algorithm



- Two Process ME를 보장하는 최초의 알고리즘
- 구현: `flag`와 `turn` 둘 다 사용
  - 임계 영역 진입 전, `flag = true`로 변경
    - 상대편 `flag`가 `false`라면 곧장 진입
    - 상대편 `flag`가 `true`라면 누구 차례인지 확인하고 차례인 프로세스가 진입
  - 임계 영역 진입 후, `turn` 및 `flag` 변경
  - Mutual Exclusion, Progress, Bounded Waiting 모두 보장

Peterson's Algorithm



- Dekker’s Algorithm과 동일, 보다 간단하게 구현
- 구현
  - 임계 영역 진입 전, `flag = true`로 변경 및 `turn` 값도 변경하면서 양보

SW Solution: N-Process Mutual Exclusion

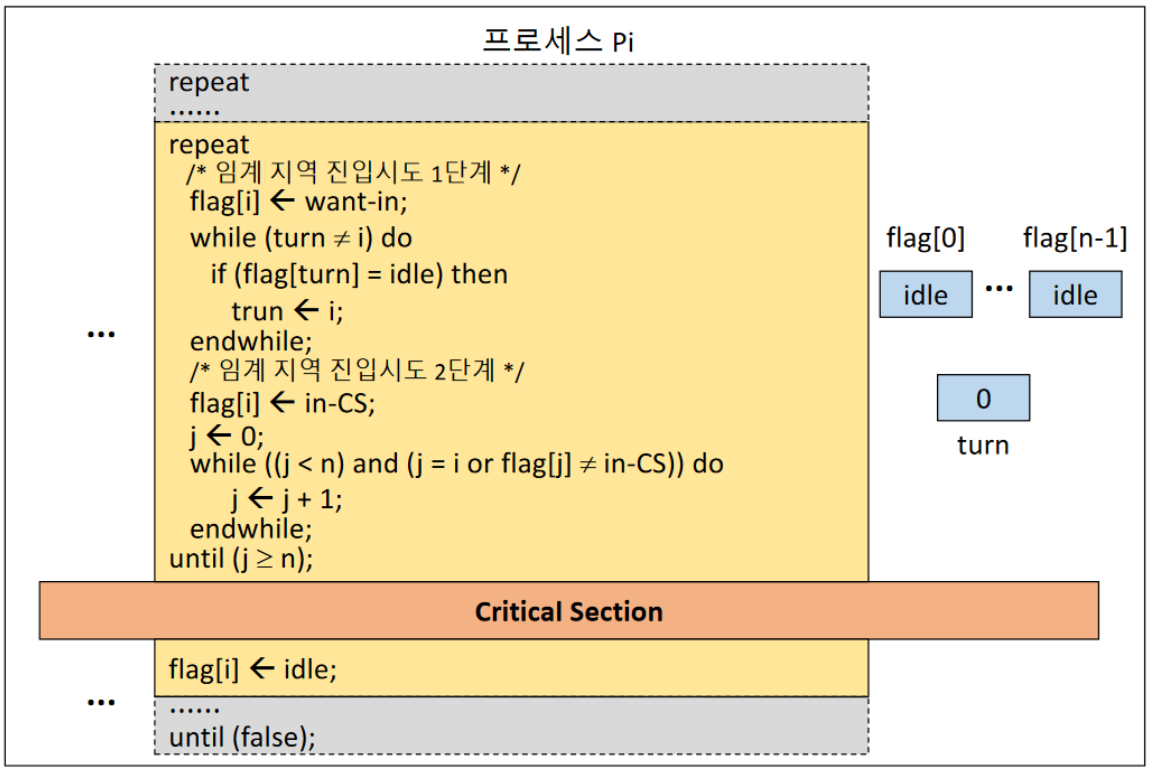
Dijkstra’s Algorithm

- 최초로 프로세스 N개의 ME 문제를 SW적으로 해결
- 실행 시간이 가장 짧은 프로세스에 프로세서를 할당하는 세마포 방법
  - 가장 짧은 평균 대기 시간 제공

[Dijkstra 알고리즘의 `flag[]` 변수]

1. `idle`: 프로세스가 임계 지역 진입을 시도하고 있지 않을 때
2. `want-in`: 프로세스의 임계 지역 진입 시도 1단계일 때
   - 임계 영역에 진입하고 싶다는 의사를 밝히는 단계
3. `in-CS`: 프로세스의 임계 지역 진입 시도 2단계 및 임계 지역 내에 있을 때
   - 임계 영역 진입 직전 단계

[Dijkstra 알고리즘 구현]



1. 임계 지역 진입 시도 1단계
   - 임계 지역 진입 의사 밝힘: `flag[i] = want-in`
   - 만약 자기 차례가 아니라면, 현재 차례인 프로세스가 `idle`이 될 때까지 대기
     - `idle`이 되었을 때, `turn = i`로 설정
     - `turn` 설정하고 while 문 종료하는 사이에 다른 프로세스가 `turn`을 가져갈 수도 있음
2. 임계 지역 진입 시도 2단계
   - while문을 빠져나와 `flag[i] = in-CS`로 설정
   - `flag[j] = in-CS`인 다른 프로세스가 존재하는지 확인
     - 다른 프로세스가 존재한다면, 임계 지역 진입 시도 1단계로 돌아감
     - 다른 프로세스가 없다면, 임계 지역 진입

- ME 조건 만족
  - 임계 지역에 언제나 한 프로세스만 진입 ➡️ Mutual Exclusion
  - 임계 지역에 프로세스가 없다면 곧장 진입 가능 ➡️ Progress
  - 운이 없어서 `in-CS` 확인할 때, 확인하던 모든 프로세스가 1단계로 돌아갈 수도 있지만, 언젠가 임계 지역에 진입할 것임 ➡️ Bounded Wait

SW Solution 문제점

- 비효율적: 반복문이 많아서 속도가 느리고, 구현 복잡
- ME Primitive 실행 중 Preemption 될 수 있음
  - 공유 데이터 수정 중에 발생하는 Preemption은 인터럽트를 억제해서 해결 가능하지만, 오버헤드가 발생한다는 문제가 있음
- Busy Waiting: 임계 영역을 진입할 수 있는지 여부를 확인하기 위해서 계속 반복(while 문)해서 확인하며 대기해야 하기 때문에 비효율적

Mutual Exclusion: HW Solution

TestAndSet (TAS) Instruction

TAS 정의

- Test와 Set을 한 번에 수행하는 기계어 명령
- 기계어의 특징 (Atomicity, Indivisible) 때문에 실행 중 인터럽트를 받지 않음 (Preemption ❌)

TAS Instruction

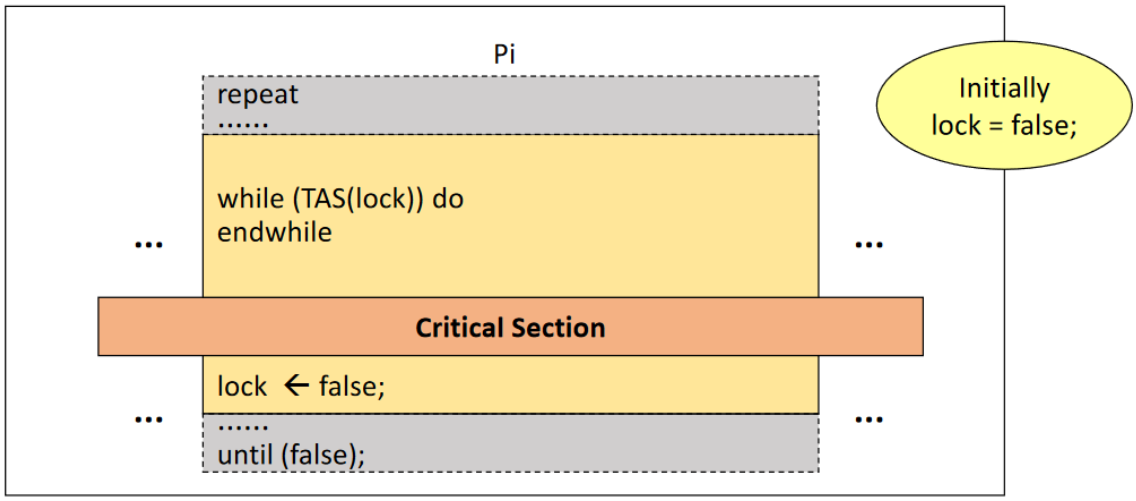
예제 4-6 TestAndSet 명령어

```
// target을 검사하고, target 값을 true로 설정
boolean TestAndSet (boolean *target) {
    boolean temp = *target;     // 이전 값 기록
    *target = true;             // true로 설정
    return temp;                // 값 반환
}
```

한번에 수행
(Machine instruction)

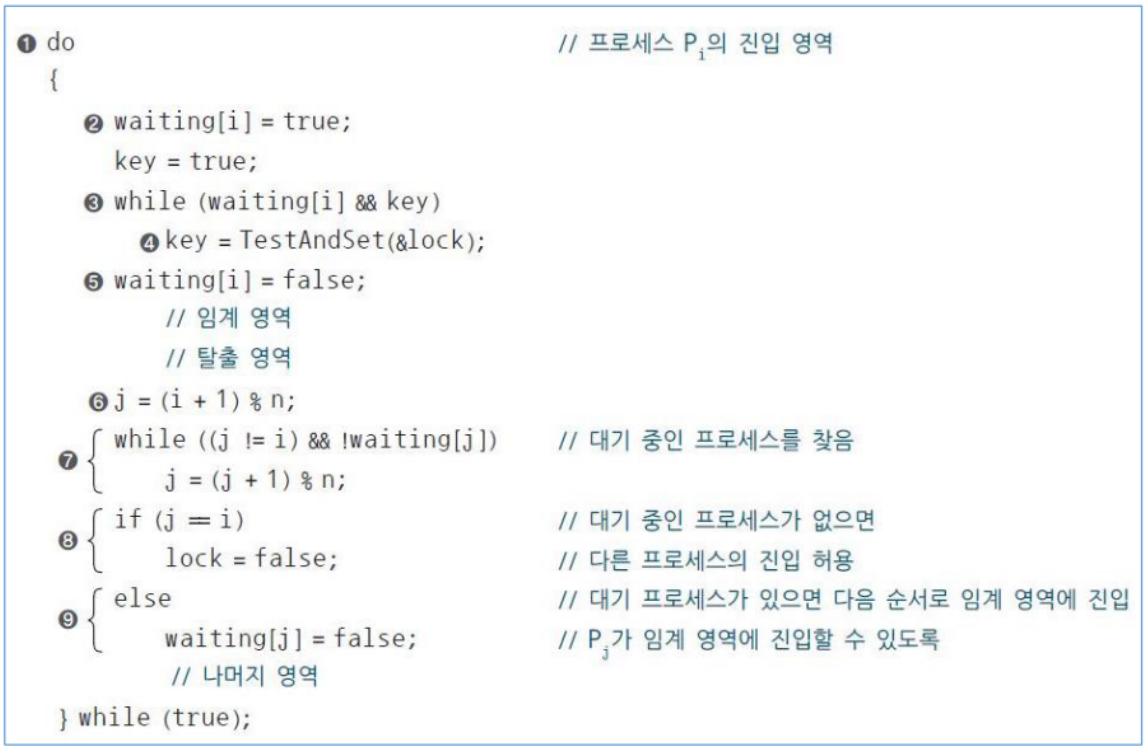
- 위 명령어가 인터럽트 없이 한 번에 수행됨을 보장
  - `lock`이 `false` 일 때, `TAS(lock)`은 `false` 반환 및 `lock` 값은 다시 `true`로 설정

ME with TAS Instruction: Two Process ME



- 구현
  - 임계 영역 진입 전, `TAS(lock)`이 `false`임을 확인, `false`면 진입
    - `TAS(lock)`이 `true`면 `false` 될 때까지 대기
  - 임계 영역 퇴장 시, `lock = false`로 변경
- 3개 이상의 프로세스의 경우, Bounded Waiting 조건 위배
  - `TAS(lock) = true`에서 `false`로 바뀔 때, 대기하고 있던 프로세스 중 먼저 while문이 끝난 프로세스가 임계 영역에 진입하게 됨
  - 운이 안 좋은 프로세스의 경우, 계속 임계 영역에 진입하지 못하고 무한 대기 가능
    ➡️ Bounded Waiting 위배

ME with TAS Instruction: N-Process ME



- 변수
  - `waiting`: 프로세스가 대기 중인지 여부
  - `key`: `TAS(lock)` 수행 결과. `lock` 초기값은 `false`
- 구현
  - 임계 영역 진입 전, while문에서 `TAS(lock)` 값을 계속 확인
    - 자신의 `waiting` 값 또는 `TAS(lock)`값이 `false`일 때 임계 영역 진입 가능. 진입 전 `waiting = false` 로 변경
  - 임계 영역 퇴장 시, 자기 자신 이후에 대기 중인 프로세스 찾음 (`j = (i + 1) % n`)
    - 대기 중인 프로세스가 없다면 `lock = false`로 변경, 다른 프로세스 접근 허용
    - 대기 중인 프로세스가 있다면 해당 프로세스가 진입할 수 있도록 그 프로세스의 `waiting` 값을 `false`로 변경

HW Solution 장단점

- 장점
  - 구현 간단
- 단점

- Busy Waiting 문제 해결 X ➡️ OS Supported SW Solution

Mutual Exclusion: OS Supported SW Solution

Spinlock

Spinlock 정의

> 정수 변수. 초기화, `P()`, `V()` 연산으로만 접근 가능

- 위 초기화, `P()`, `V()` 연산은 indivisible (or atomic) 연산
  - OS에서 전체가 한 instruction cycle에 수행됨을 보장

[`P()`, `V()`]

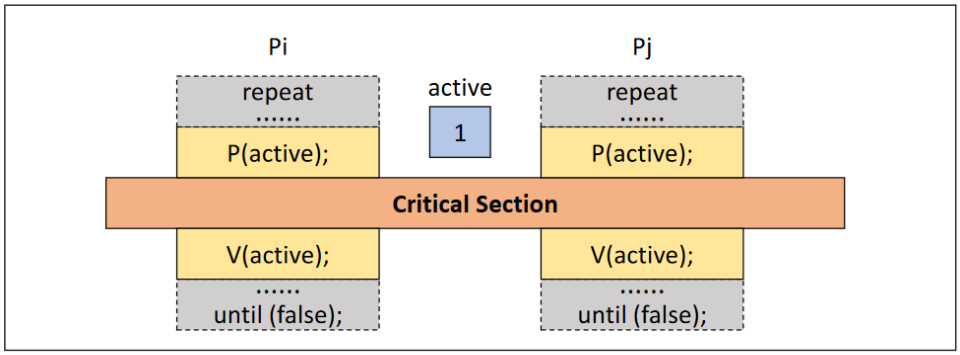
```
P(S) {
  while (S ≤ 0) do
  endwhile;
  S ← S – 1;
}
```

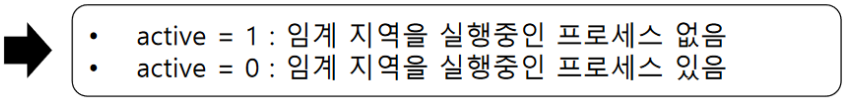
```
V(S) {
  S ← S + 1;
}
```

S: 물건의 개수

1. `P(S)`: 물건 꺼내기, 자물쇠 잠그기
   - 물건이 없다면, 물건이 생길 때까지 대기. 있다면 꺼내감
2. `V(S)`: 물건 넣기, 자물쇠 풀기
   - 물건 반납

Spinlock: Mutual Exclusion





- 구현
  - 임계 영역 진입 전, `P()` 연산으로 임계 영역 진입 가능 여부 확인
    - 만약 임계 영역 진입 가능하다면, `P()` 연산 내 `active = 0`으로 변경
  - 임계 영역 퇴장 시, `V()` 연산으로 임계 영역 퇴장 알림
- OS에서 `P()` 연산 중 Preemption 되지 않는 것을 보장, Mutual Exclusion 가능

Spinlock 문제점

- 멀티 프로세서 시스템에서만 사용 가능
  - 프로세스 P_i가 `P()` 연산 내에서 대기 중이라면 프로세스 P_j는 해당 프로세서 사용 불가. OS는 `P()` 연산 중 쫓아내지 않기 때문에 대기 시간이 길어짐
  - 멀티 프로세서 시스템인 경우에만 위 그림처럼 프로세스 P_i와 P_j가 공존할 수 있음
- Busy Waiting 문제 해결 X
  - `P()`연산 내 반복문

Semaphore

Semaphore 정의

> 음이 아닌 정수형 변수(S). 초기화, `P()`, `V()` 연산으로만 접근 가능

- 1965년 Dijkstra가 제안, Busy Waiting 문제 해결
- Spinlock과 유사

- 비슷하게 임계 영역 진입 전 연산 `P()`와 퇴장 후 연산 `V()` 존재

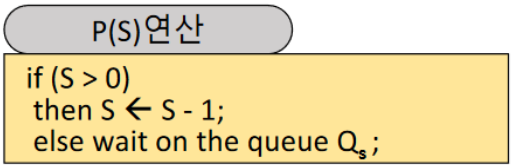
- ⭐임의의 S 변수 하나에 Ready Queue 하나가 할당됨

Semaphore 종류

1. Binary Semaphore
   - S가 0과 1 두 종류의 값만 갖는 경우
   - 상호 배제나 프로세스 동기화 목적으로 사용
2. Counting Semaphore
   - S가 0 이상의 정수값을 가질 수 있는 경우
   - Producer-Consumer 문제 등 해결하기 위해 사용

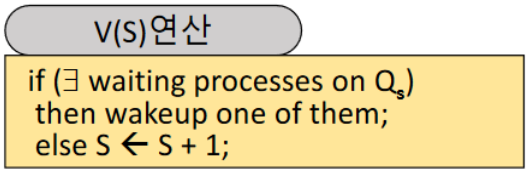
Semaphore 연산

1. 초기화 연산
   - S 변수에 초기값을 부여하는 연산
2. P() 연산



- 물건 S가 있다면, 물건을 꺼내감
- 물건이 없다면, **S에 할당된 ready queue Q에서 대기**

3. V() 연산



- Q에 대기하고 있는 프로세스가 있다면, 그 중 한 프로세스를 wakeup
- 대기하고 있는 프로세스가 없다면, 반납

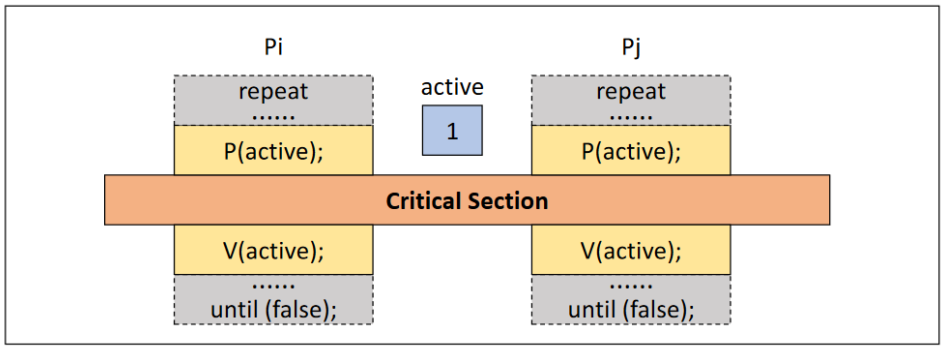
- 모두 indivisible한 연산
  - OS 보장, 전체가 한 instruction cycle에 수행됨

Semaphore로 해결 가능한 동기화 문제들

1. 상호 배제 문제

2. 프로세스 동기화 문제
3. 생산자-소비자 문제
4. Reader-Writer 문제
5. Dining Philosopher 문제

Semaphore: Mutual Exclusion Problem

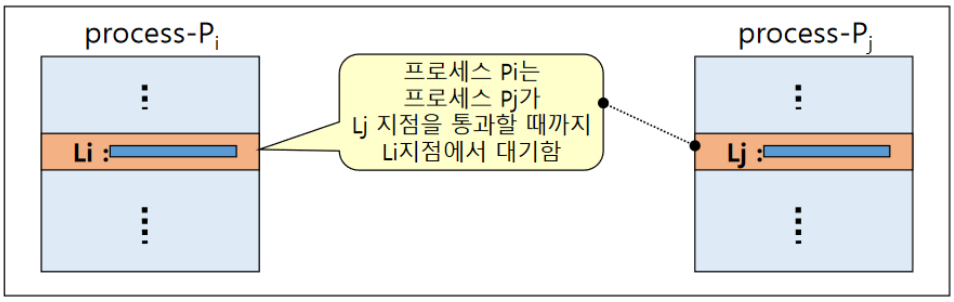


- active = 1 : 임계 지역을 실행중인 프로세스 없음
- active = 0 : 임계 지역을 실행중인 프로세스 있음

- Spinlock과 동일하게 해결 가능
- Spinlock과 다른 점은 `P()` 연산에서 busy waiting 문제 발생 X
  - 프로세스가 직접 계속 확인하는 것이 아니라 ready queue에 들어가 있으면 `V()` 연산이 wake up 해줌

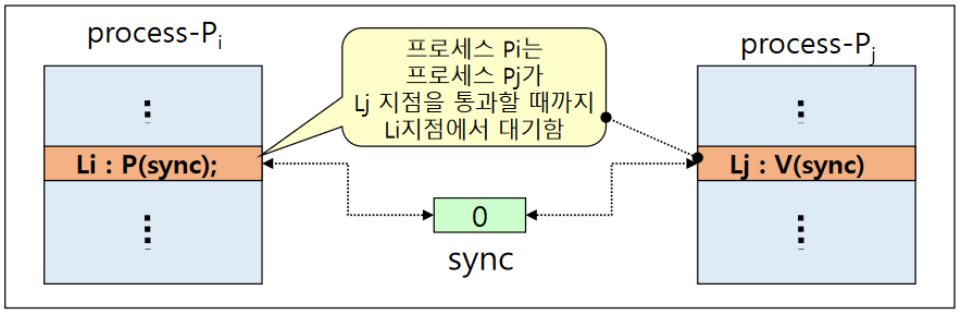
Semaphore: Process Synchronization Problem

- 프로세스들의 실행 순서를 맞추는 문제
  - 프로세스는 병행적이며 비동기적으로 수행



프로세스 P_j가 먼저 실행되고 있을 때

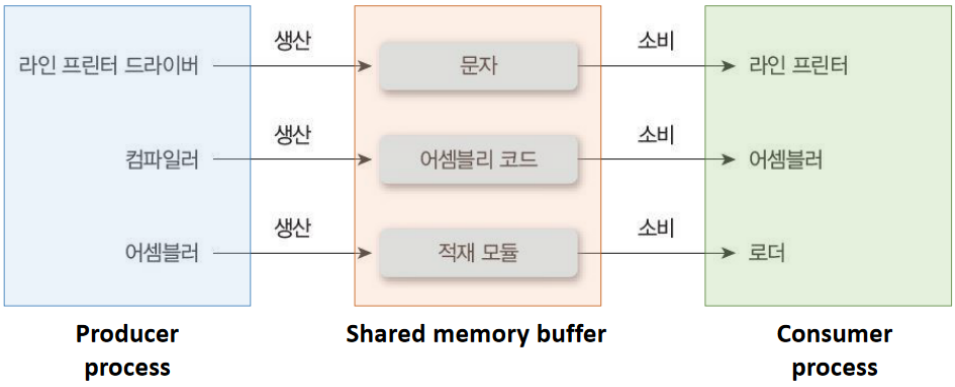
[Semaphore를 사용한 해결 방법]



- P_i가 L_i 지점에서 대기할 때: 프로세스 P_j가 L_j 지점을 통과하기 전
  - `P(sync)` 연산에서 ready queue에 들어감
- P_j가 L_j 지점을 통과할 때
  - `V(sync)` 연산에서 ready queue에 있는 P_i를 wakeup 함

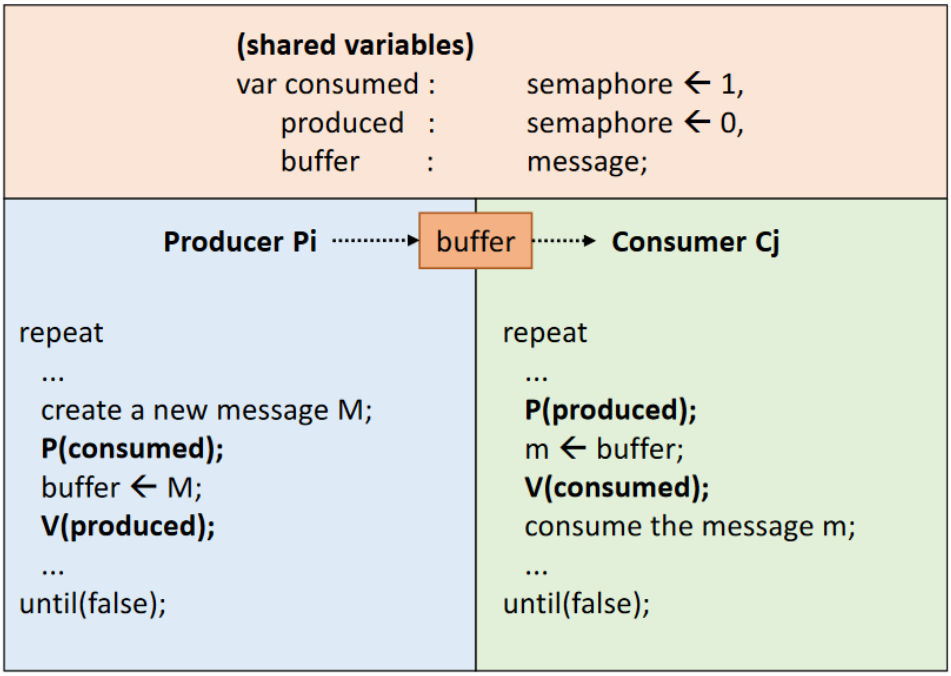
Semaphore: Producer-Consumer Problem

[Producer-Consumer Problem 정의]



- Actors
  1. 생산자 프로세스: 메시지를 생성하는 프로세스 그룹
  2. 소비자 프로세스: 메시지를 전달받는 프로세스 그룹
- 문제 ➡️ 동기화 필요
  - 메시지 생산 중 버퍼에 다른 메시지가 생산되면 안 됨
  - 메시지 생산 중 소비되면 안 됨

[Producer-Consumer Problem With Single Buffer]

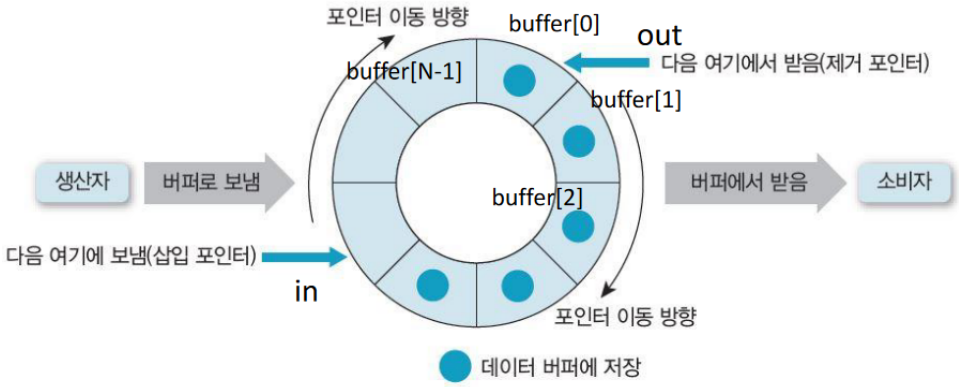


buffer = 1인 경우. 한 번에 한 프로세스만 버퍼에 접근해야 함

- 변수
  - `consumed`: 소비되었는지 여부. `produced`의 반대값
  - `produced`: 생산되었는지 여부. `consumed`의 반대값
- Producer P_i
  1. 새로운 메시지 M 생산
  2. `P(consumed)`: 버퍼가 비었는지(소비되었는지) `consumed` 변수로 확인
     - 아직 메시지가 소비되지 않았다면 (`consumed = 0`), ready queue에서 대기
     - 메시지가 소비되었다면 (`consumed = 1`), `consumed = 0`으로 변경
  3. 버퍼가 비었다면 생산한 메시지 M 놓기
  4. `V(produced)`: 메시지를 생산했다는 것을 알림
     - `produced = 1`로 변경
     - `produced` queue에 대기하는 Consumer 프로세스 wakeup
- Consumer C_j
  1. `P(produced)`: 버퍼에 메시지가 있는지 `produced` 변수로 확인
     - 아직 메시지가 생산되지 않았다면 (`produced = 0`), ready queue에서 대기
     - 메시지가 생산되었다면 (`produced = 1`), `produced = 0`으로 변경

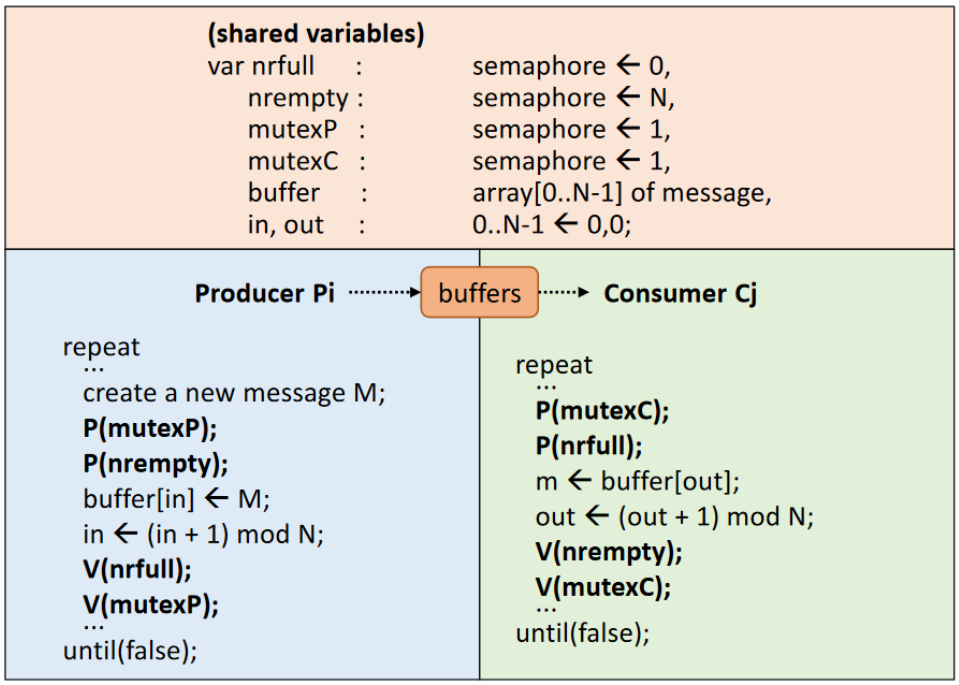
2. 버퍼에 메시지가 있다면 메시지 m 가져오기
3. `V(consumed)`: 메시지를 소비했다는 것을 알림
    - `consumed = 1`로 변경
    - `consumed` queue에 대기하는 Producer 프로세스 wakeup
4. 가져온 메시지 m 소비

[Producer-Consumer Problem With N-Buffers]



Circular Queue 사용한다고 가정

- `in`: 생산자가 물건을 놓는 지점
- `out`: 소비자가 물건을 가져가는 지점



- 변수
  - `mutexP`, `mutexC`: 한 번에 한 Producer/Consumer 프로세스만 실행되도록 사용하는 변수
  - `nrfull`, `nrempty`: 채워진/비어있는 버퍼 수
    - `nrfull + nrempty = N`
- Producer P_i
  1. 새로운 메시지 M 생산
  2. `P(mutexP)`, `V(mutexP)`: 한 번에 한 Producer 프로세스만 메시지 생산하도록 함
  3. `P(nrempty)`: 비어있는 버퍼가 있는지 `nrempty`변수로 확인
     - 비어있는 버퍼가 없다면 대기
     - 비어있는 버퍼가 있다면 `nrempty` 값 감소
  4. 비어있는 버퍼가 있다면 해당 자리 `buffer[in]`에 생산한 메시지 M 놓기
  5. `in` 값 갱신: Circular Queue에서 다음 메시지를 놓을 자리 갱신
  6. `V(nrfull)`: 메시지를 생산했다는 것을 알림
     - `nrfull` 값 증가
- Consumer C_j
  1. `P(mutexV)`, `V(mutexV)`: 한 번에 한 Consumer 프로세스만 메시지 소비하도록 함
  2. `P(nrfull)`: 버퍼에 메시지가 있는지 `nrfull`변수로 확인
     - 버퍼가 비어있다면 대기
     - 버퍼에 메시지가 있다면 `nrfull` 값 감소
  3. 버퍼에 메시지가 있다면 해당 자리 `buffer[out]`에 있는 메시지 m 가져오기
  4. `out` 값 갱신
  5. `V(nrempty)`: 메시지를 소비했다는 것을 알림
     - `nrempty` 값 증가

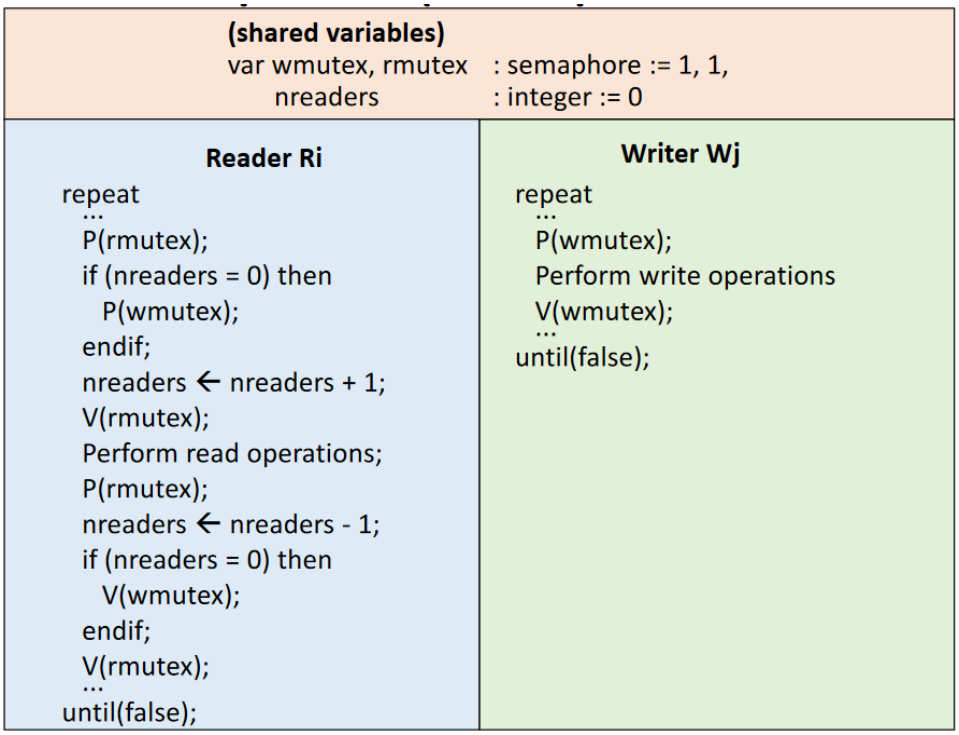
Semaphore: Reader-Writer Problem

[Reader-Writer Problem 정의]

- Actors

    - Reader: 데이터에 대해 읽기 연산만 수행
    - Writer: 데이터에 대해 갱신 연산을 수행
- 문제: 데이터 무결성 보장 필요
    - Reader는 동시에 데이터 접근 가능
    - 여러 Writer (또는 Reader와 Writer)가 동시에 데이터 접근 시, 상호 배제(동기화) 필요
- 해결법: Reader/Writer에 대한 우선권 부여
    - Reader Preference Solution
    - Writer Preference Solution

[Reader-Writer Problem: Reader Preference Solution]



(shared variables)
var wmutex, rmutex : semaphore := 1, 1,
nreaders : integer := 0

Reader Ri

```
repeat
  ...
  P(rmutex);
  if (nreaders = 0) then
    P(wmutex);
  endif;
  nreaders ← nreaders + 1;
  V(rmutex);
  Perform read operations;
  P(rmutex);
  nreaders ← nreaders - 1;
  if (nreaders = 0) then
    V(wmutex);
  endif;
  V(rmutex);
  ...
until(false);
```

Writer Wj

```
repeat
  ...
  P(wmutex);
  Perform write operations
  V(wmutex);
  ...
until(false);
```

- 변수
    - `wmutex`, `rmutex`: Writer/Reader 프로세스 상호 배제를 위한 변수
    - `nreaders`: Reader 수
- Reader R_i
    - 읽기 수행 사전 작업

1. `P(rmutex)`, `V(rmutex)`: 읽는 것은 여러 Reader가 할 수 있지만 사전/사후 작업은 한 Reader만 가능
2. `P(wmutex)`: 읽는 도중 write 작업이 발생하지 않도록 `wmutex` 설정
    - `nreaders > 0`일 때 실행한다면, `wmutex` 값 음수 가능
3. `nreaders` 값 증가

- 읽기 수행
- 읽기 수행 사후 작업

1. `nreaders` 값 감소
2. `V(wmutex)`: 더 이상 실행되고 있는 Reader가 없다면 `wmutex` 값 변경

- Writer W_i
    - `P(wmutex)`, `V(wmutex)`: 동시 실행 불가하기 때문에 lock 걸어주기

Semaphore 장단점

- 장점
    - No Busy Waiting: Ready Queue를 사용, 기다려야 하는 프로세스는 block(asleep) 상태가 됨
- 단점
    - Starvation Problem: Queue에 대한 wake up 순서는 비결정적이기 때문에 계속 대기하고 있는 프로세스가 발생할 수 있음 ➡️ Eventcount/Sequencer

Eventcount/Sequencer

- 은행의 번호표와 비슷한 개념

Sequencer

> 정수형 변수. 발생 사건들의 순서(sequence) 유지
>
> - 생성 시 0으로 초기화, 감소 X
> - `ticket()` 연산으로만 접근 가능

- `ticket(S)`
    - 현재까지 `ticket()` 연산이 호출된 횟수 반환

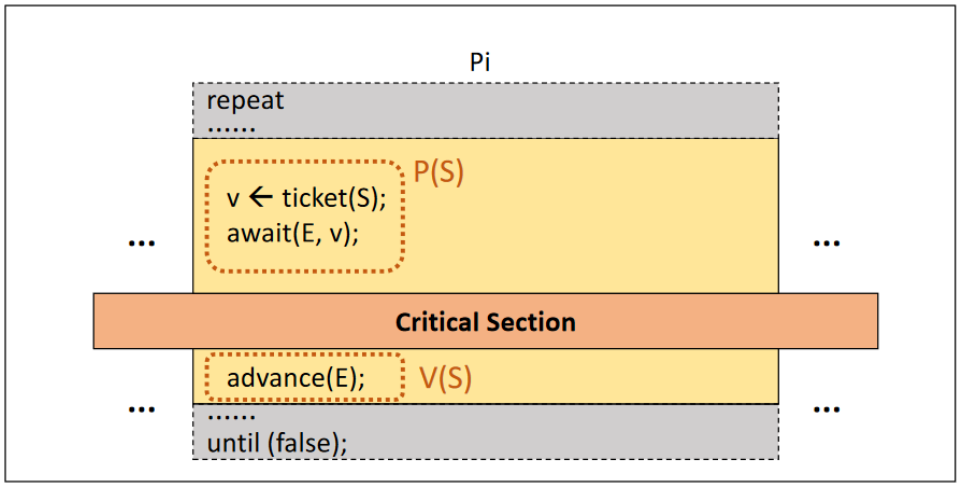
- Indivisible Operation

Eventcount

> 정수형 변수. 특정 사건의 발생 횟수 기록
>
> - 생성 시 0으로 초기화, 감소 X
> - `read(E)`, `advance(E)`, `await(E, v)` 연산으로만 접근 가능

- `read(E)`
  - 현재 Eventcount 값 반환
- `advance(E)`
  - `E` 값 증가
  - `E`를 기다리고 있는 프로세스를 깨움(wake up)
- `await(E, v)`
  - `v`는 정수형 변수. 번호표
  - `E < v`면, `E`에 연결된 Queue에 프로세스 전달(push) 및 CPU Scheduler 호출

Eventcount/Sequencer: Mutual Exclusion

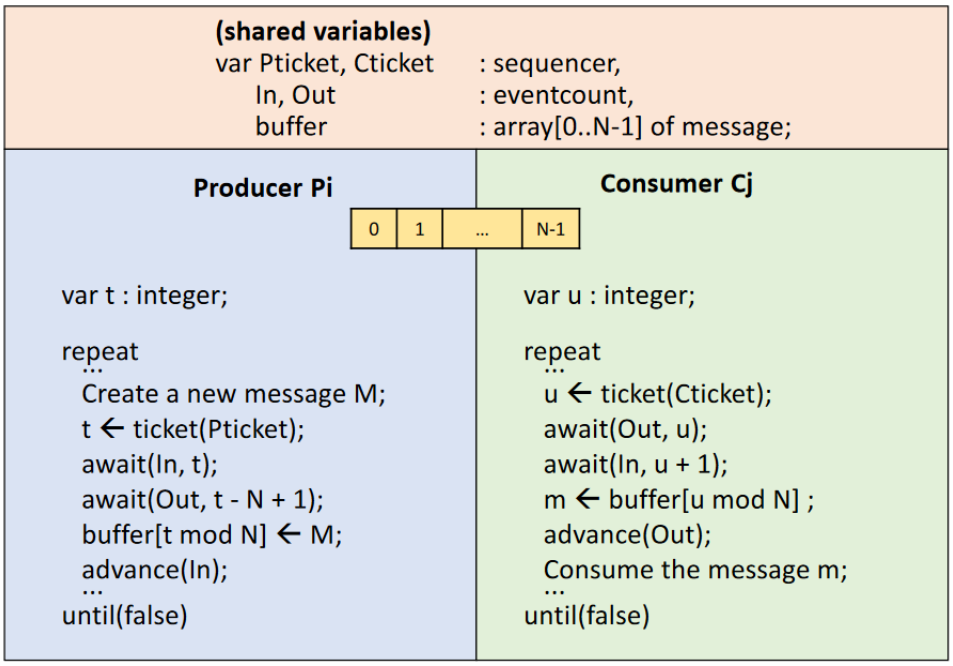


- 초기 설정: `S = 0`, `E = 0`

1. `ticket(S)`
   - 번호표 뽑기. 순서를 의미하는 숫자를 받아 `v`에 저장

2. `await(E, v)`
   - 차례가 오지 않았다면(`E < v`) Queue에서 대기
   - 차례가 왔다면(`E == v`)면 임계 영역 진입
3. `advance(E)`
   - `E` 증가
   - Queue 내 대기하고 있는 프로세스 wake up

Eventcount/Sequencer: Producer-Consumer Problem



- 변수
  - `Pticket`, `Cticket`: Producer/Consumer 순서 관리 위한 Sequencer
  - `In`, `Out`: 메시지를 생산/소비하는 이벤트에 대한 Eventcount
- Producer P_i
  1. 새로운 메시지 M 생산
  2. `P()` 연산과 동일: 한 번에 한 Producer만 가능
     - `ticket(Pticket)`: 번호표 뽑기
     - `await(In, t)`: 차례 돌아올 때까지 기다렸다 진입
  3. 생산한 메시지를 버퍼에 놓기 위한 작업: 임계 구역

    - `await(Out, t - N + 1)`: 버퍼에 빈 공간이 있는지 확인
        - `(N - t + Out) >= 1`일 때 버퍼에 빈 공간 존재
    - `buffer[t mod N]`: 메시지를 버퍼에 넣음
4. `V()` 연산과 동일
    - `advance(In)`: 메시지 넣은 후 `In` 증가

- Consumer C_j
1. `P()` 연산과 동일: 한 번에 한 Producer만 가능
    - `ticket(Cticket)`: 번호표 뽑기
    - `await(Out, u)`: 차례 돌아올 때까지 기다렸다 진입
2. 생산한 메시지를 버퍼에 놓기 위한 작업: 임계 구역
    - `await(In, u + 1)`: 메시지가 존재하는지 확인
        - `(In - 1) >= 1`일 때 메시지 존재
    - `buffer[u mod N]`: 메시지를 읽어들임
3. `V()` 연산과 동일
    - `advance(Out)`: 메시지 가져간 후 `Out` 증가
4. 메시지 m 소비

Eventcount/Sequencer 특징

- Busy Waiting X
- Starvation Problem X: Queue의 FIFO 성질
- Semaphore보다 더 low-level control 가능: 순서 제어 가능