Chapter 4

프로세스 동기화 & 상호배제

Process Synchronization and Mutual Exclusion

Process Synchronization (동기화)

- **다중 프로그래밍 시스템**
  - 여러 개의 프로세스들이 존재
  - 프로세스들은 서로 독립적으로 동작
  - 공유 자원 또는 데이터가 있을 때, 문제 발생 가능

- **동기화 (Synchronization)**
  - 프로세스 들이 서로 동작을 맞추는 것
  - 프로세스 들이 서로 정보를 공유 하는 것

Asynchronous and Concurrent P's

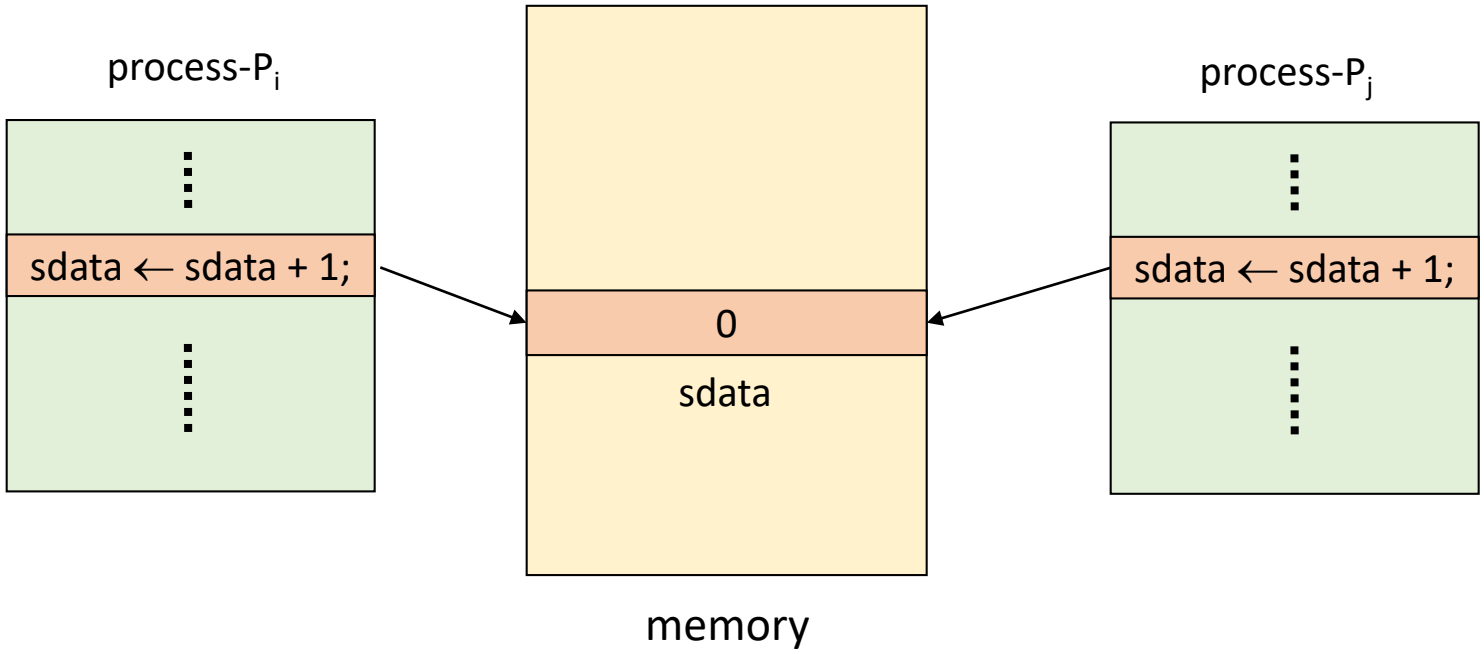
- **비동기적(Asynchronous)**
  - 프로세스들이 서로에 대해 모름
- **병행적 (Concurrent)**
  - 여러 개의 프로세스들이 동시에 시스템에 존재

- **병행 수행중인 비동기적 프로세스들이 공유 자원에 동시 접근 할 때 문제가 발생 할 수 있음**

Terminologies

- **Shared data (공유 데이터)** or Critical data
  - 여러 프로세스들이 공유하는 데이터
- **Critical section (임계 영역)**
  - 공유 데이터를 접근하는 코드 영역(code segment)
- **Mutual exclusion (상호배제)**
  - 둘 이상의 프로세스가 동시에 critical section에 진입하는 것을 막는 것

Critical Section (example)



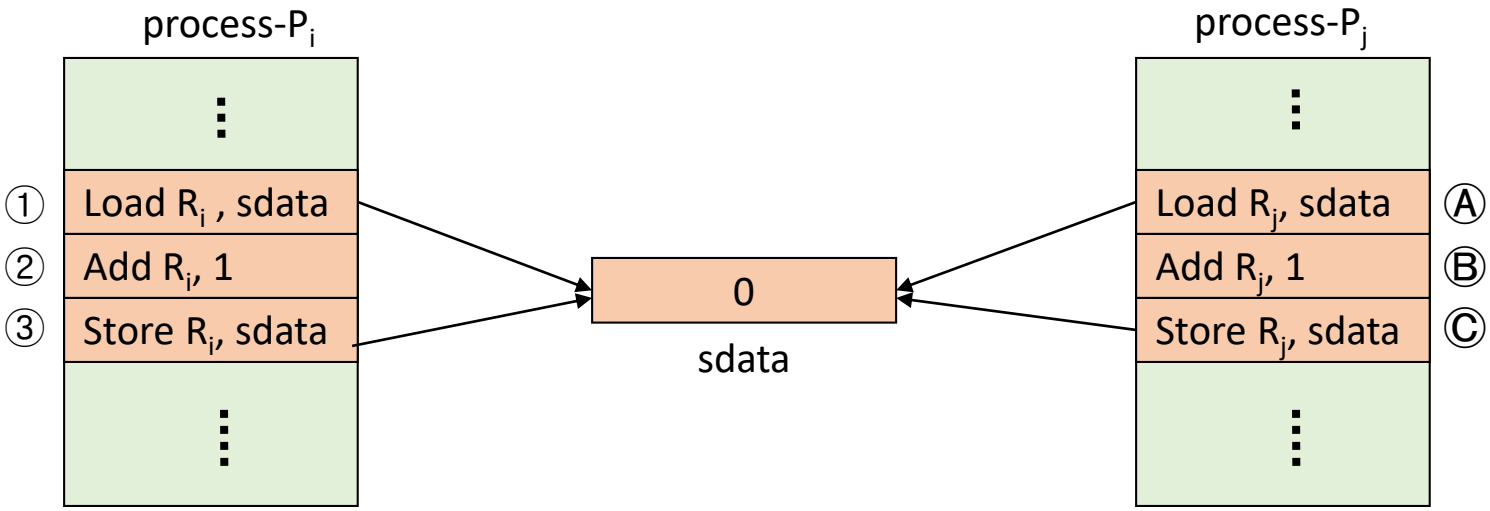
Note

기계어 명령(machine instruction)의 특성

- Atomicity (원자성), Indivisible (분리불가능)
- **한 기계어 명령의 실행 도중에 인터럽트 받지 않음**

Base images from Prof. Seo's slides

Critical Section (example)



- **명령 수행 과정 (1)**
  - ① → ② → ③ → Ⓐ → Ⓑ → Ⓒ 또는 Ⓐ → Ⓑ → Ⓒ → ① → ② → ③
  - 결과 sdata = 2
- **명령 수행 과정 (2)**
  - ① → ② → Ⓐ → Ⓑ → Ⓒ → ③
  - 결과 sdata = 1

Race condition

Base images from Prof. Seo's slides

Mutual Exclusion (상호배제)

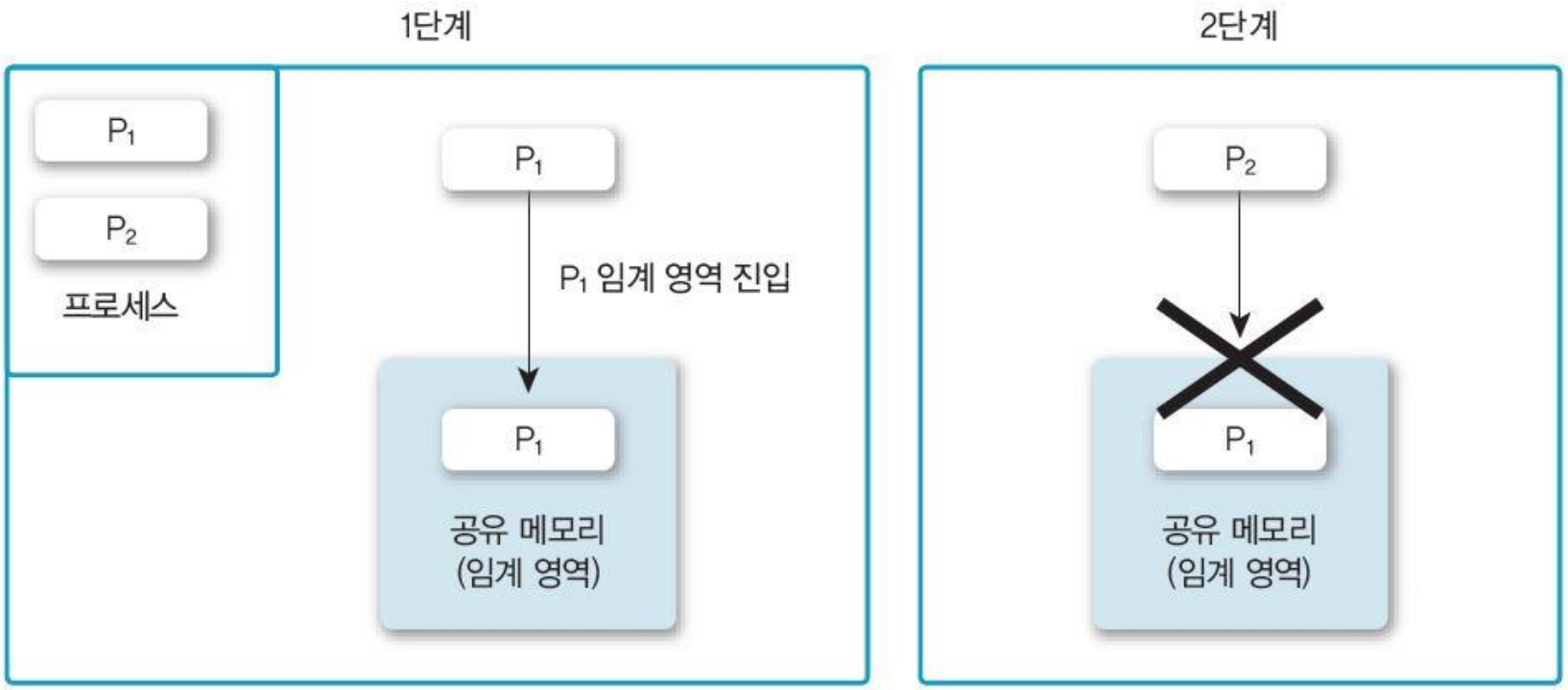
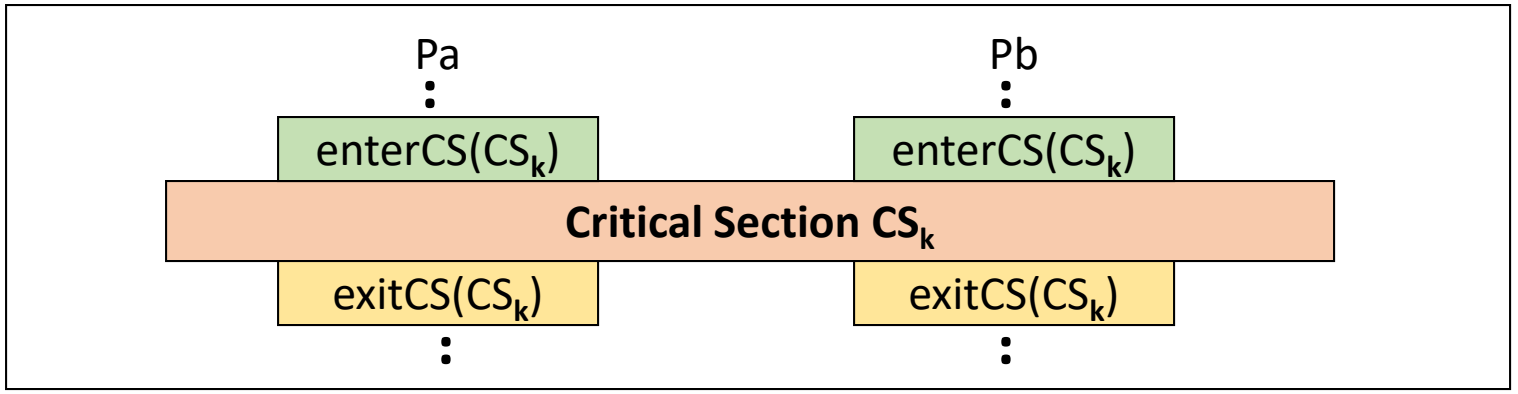


그림 4-13 상호배제의 개념

Images from the textbook

Mutual Exclusion Methods

- **Mutual exclusion primitives**
  - **enterCS()** primitive
    - Critical section 진입 전 검사
    - 다른 프로세스가 critical section 안에 있는지 검사
  - **exitCS()** primitive
    - Critical section을 벗어날 때의 후처리 과정
    - Critical section을 벗어남을 시스템이 알림

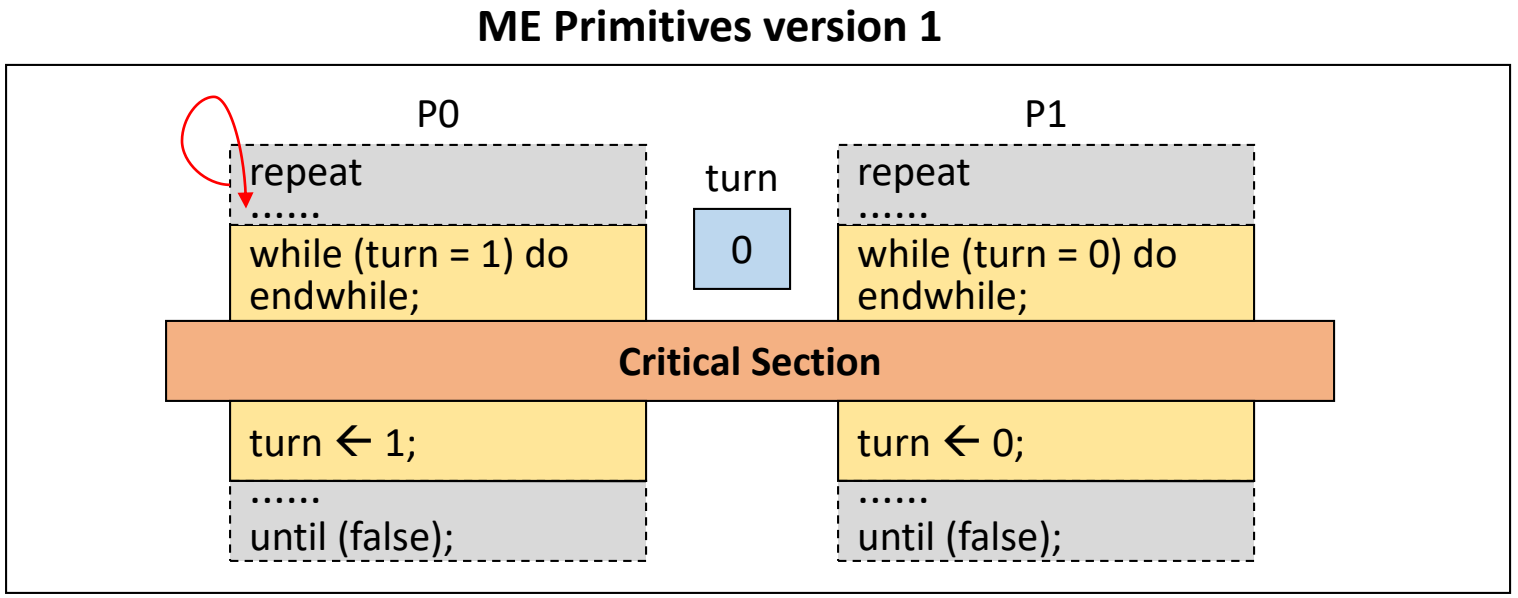


Base images from Prof. Seo's slides

Requirements for ME primitives

- **Mutual exclusion (상호배제)**
  - Critical section (CS) 에 프로세스가 있으면, 다른 프로세스의 진입을 금지
- **Progress (진행)**
  - CS 안에 있는 프로세스 외에는, 다른 프로세스가 CS에 진입하는 것을 방해 하면 안됨
- **Bounded waiting (한정대기)**
  - 프로세스의 CS 진입은 유한시간 내에 허용되어야 함

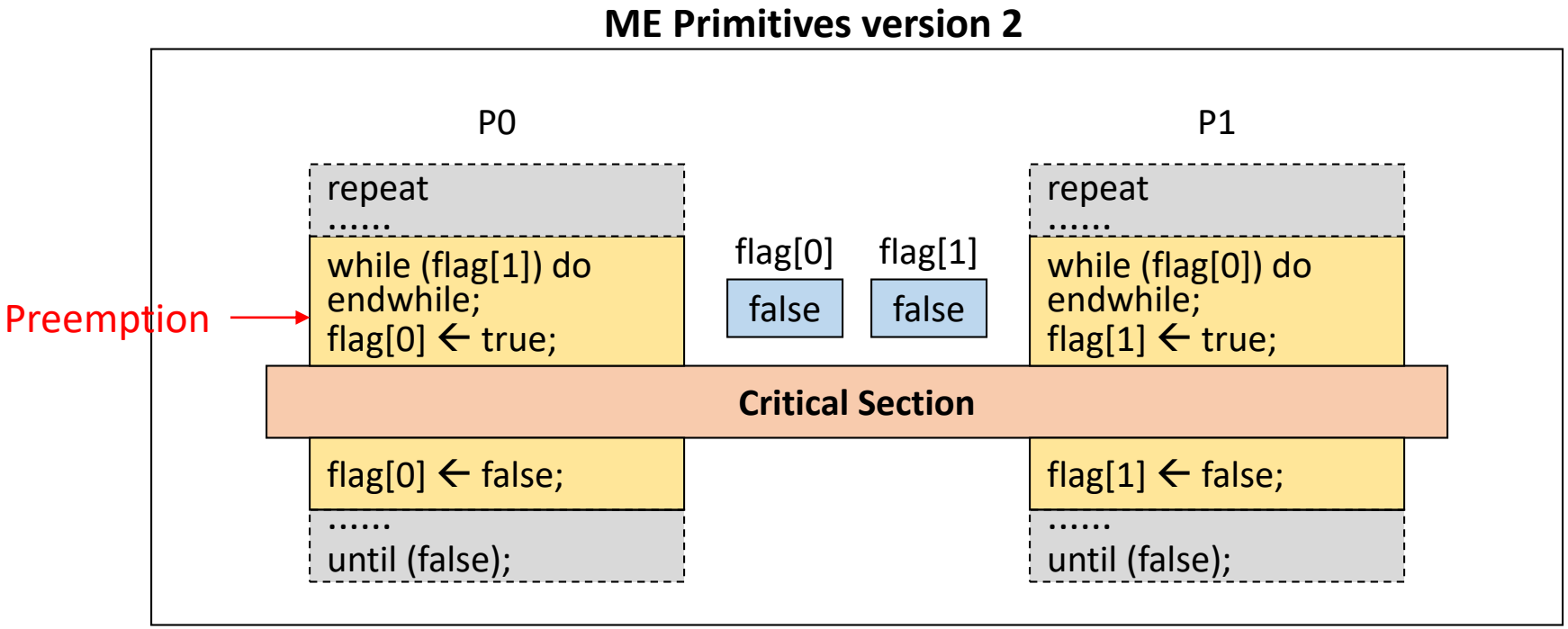
Two Process Mutual Exclusion



- **Progress 조건 위배**
  - Why?
    - P0이 critical section에 진입 하지 않는 경우
    - 한 Process가 두 번 연속 CS에 진입 불가

Base images from Prof. Seo's slides

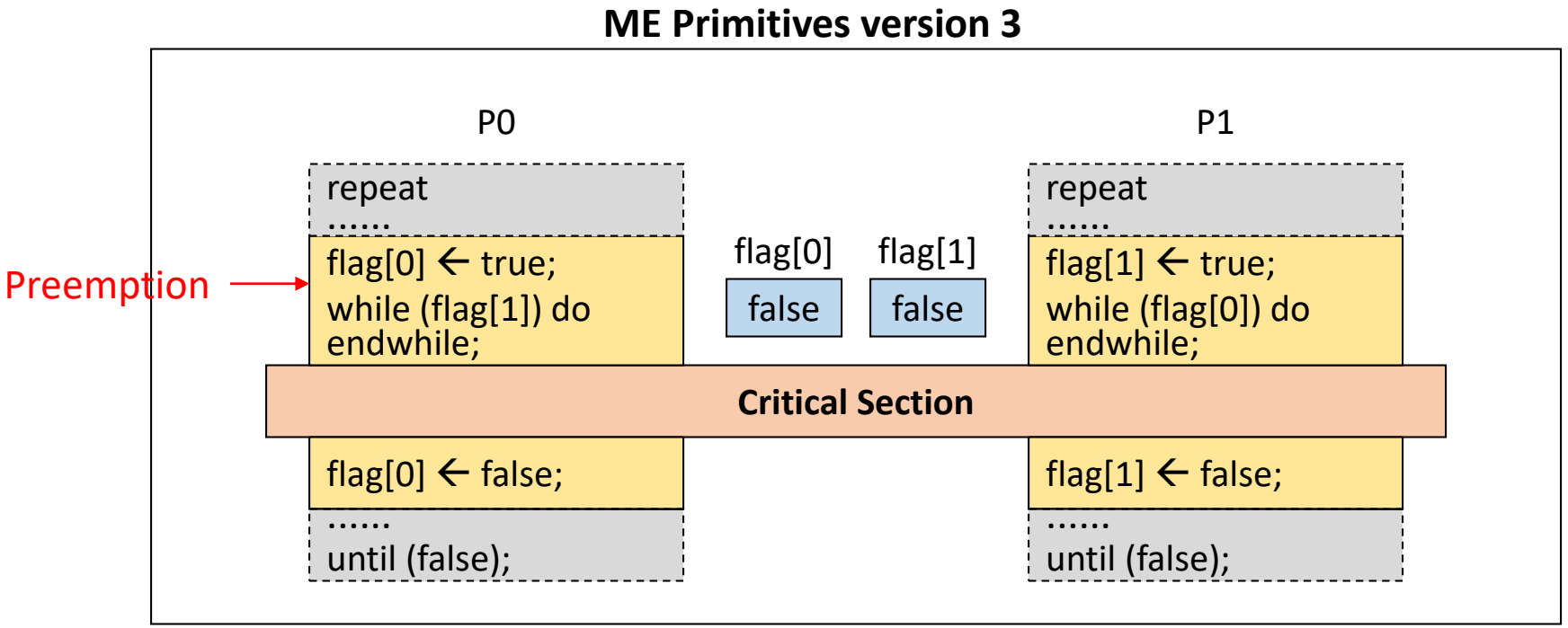
Two Process Mutual Exclusion



- **Mutual exclusion 조건 위배**
  - Why?

Base images from Prof. Seo's slides

Two Process Mutual Exclusion



- **Progress, Bounded waiting 조건 위배**
  - **Why?**

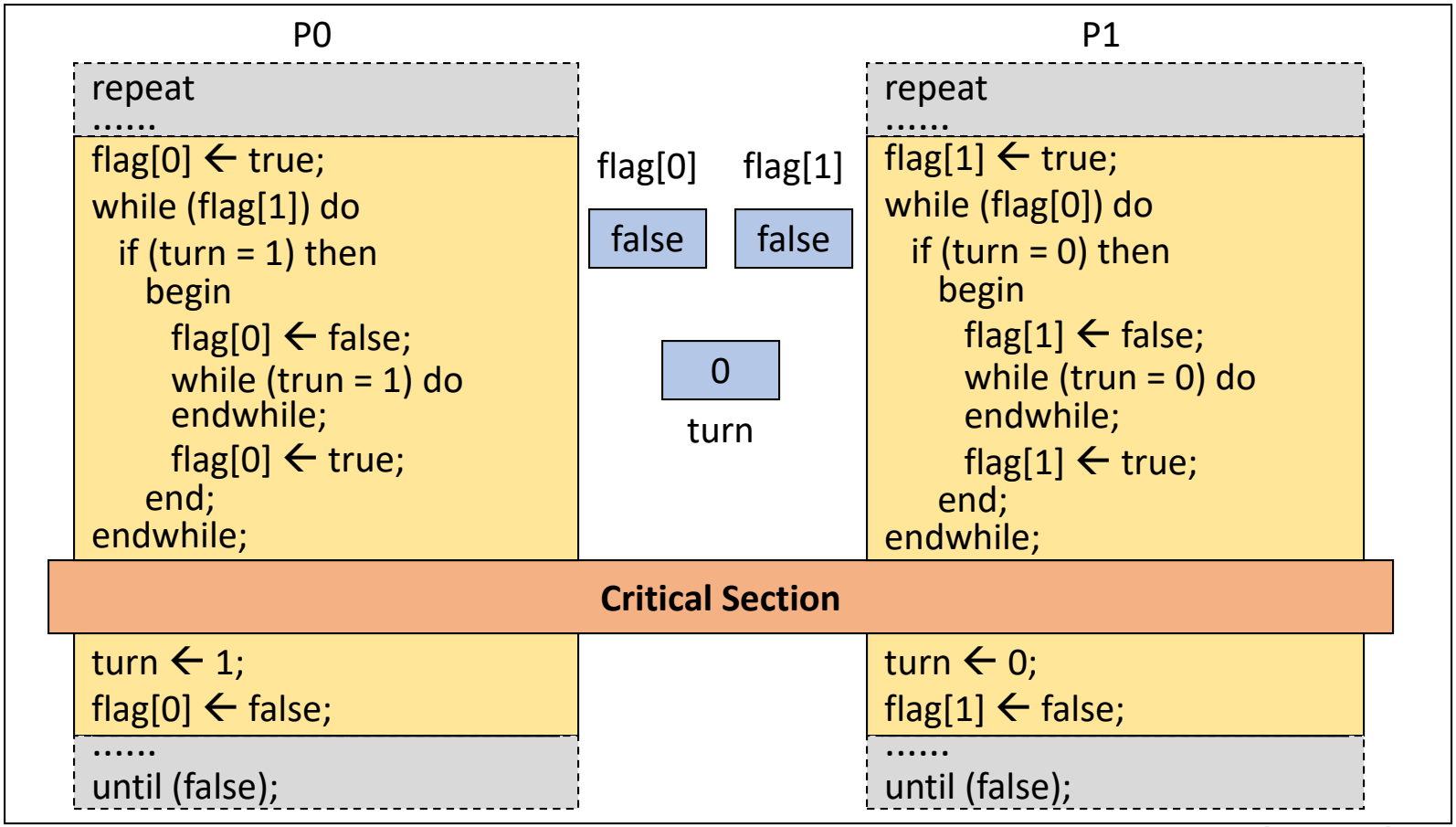
Base images from Prof. Seo's slides

Mutual Exclusion Solutions

- **SW solutions**
  - **Dekker's algorithm** (Peterson's algorithm)
  - **Dijkstra's algorithm**, Knuth's algorithm, Eisenberg and McGuire's algorithm, Lamport's algorithm
- **HW solution**
  - **TestAndSet (TAS) instruction**
- **OS supported SW solution**
  - Spinlock
  - Semaphore
  - Eventcount/sequencer
- **Language-Level solution**
  - Monitor

Dekker's Algorithm

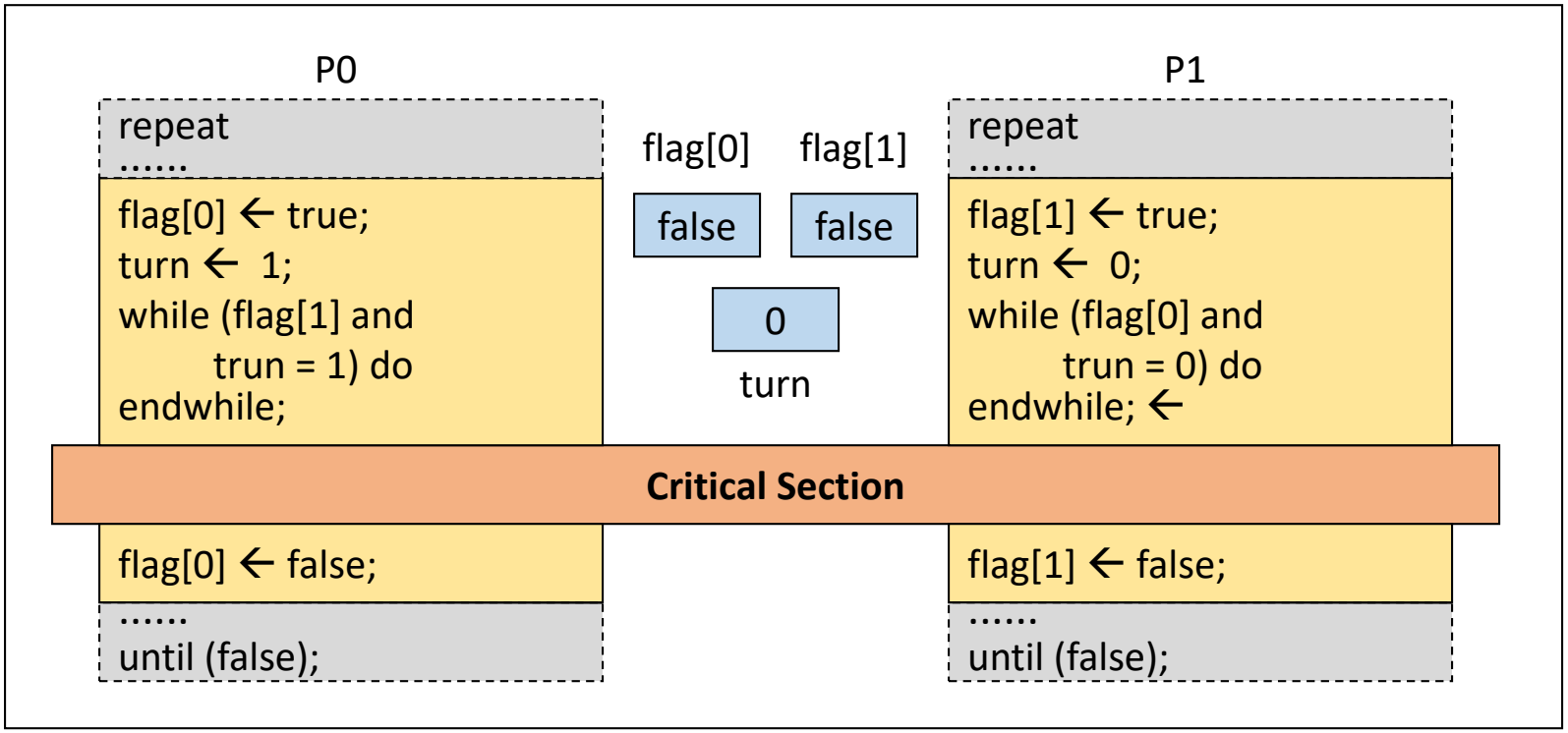
- **Two process ME을 보장하는 최초의 알고리즘**



Base images from Prof. Seo's slides

Peterson's Algorithm

- **Dekker's algorithm 보다 간단하게 구현**



Base images from Prof. Seo's slides

N-Process Mutual Exclusion

- **다익스트라Dijkstra**
  - 최초로 프로세스 n개의 상호배제 문제를 소프트웨어적으로 해결
  - 실행 시간이 가장 짧은 프로세스에 프로세서 할당하는 세마포 방법, 가장 짧은 평균 대기시간 제공
- **크누스knuth**
  - 이전 알고리즘 관계 분석 후 일치하는 패턴을 찾아 패턴의 반복을 줄여서 프로세스에 프로세서 할당
  - 무한정 연기할 가능성을 배제하는 해결책을 제시했으나, 프로세스들이 아주 오래기 다려야 함
- **램포트lamport**
  - 사람들로 붐비는 빵집에서 번호표 뽑아 빵 사려고 기다리는 사람들에 비유해서 만든 알고리즘 (Bakery algorithm)
  - 준비 상태 큐에서 기다리는 프로세스마다 우선순위를 부여하여 그 중 우선순위가 가장 높은 프로세스에 먼저 프로세서를 할당함
- **핸슨brinch Hansen**
  - 실행 시간이 긴 프로세스에 불리한 부분을 보완하는 것
  - 대기시간과 실행 시간을 이용하는 모니터 방법
  - 분산 처리 프로세서 간의 병행성 제어 많이 발표

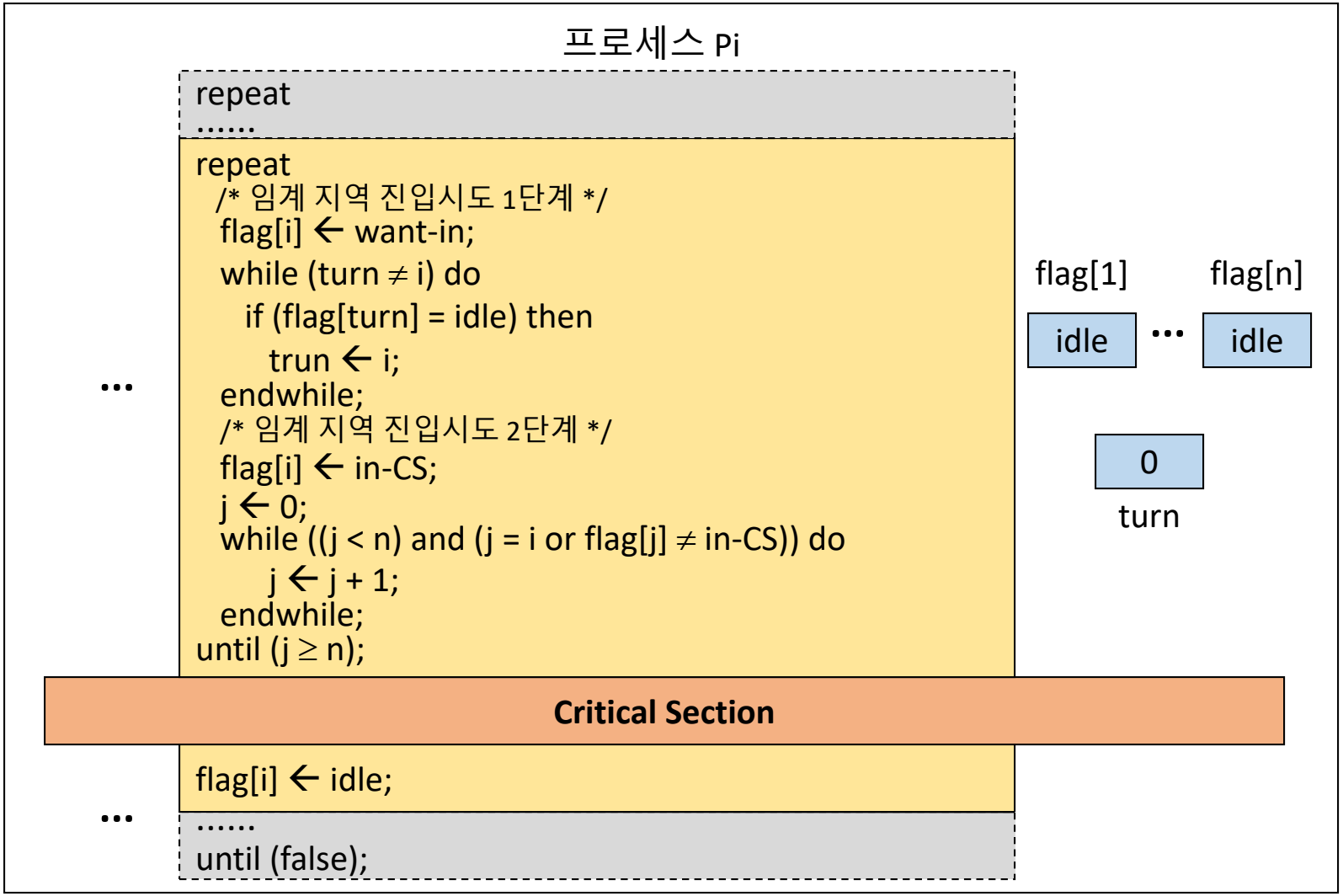
From the text book

Dijkstra's Algorithm

Dijkstra 알고리즘의 flag[] 변수

flag[] 값	의 미
idle	프로세스가 임계 지역 진입을 시도하고 있지 않을 때
want-in	프로세스의 임계 지역 진입 시도 1단계일 때
in-CS	프로세스의 임계 지역 진입 시도 2단계 및 임계 지역 내에 있을 때

Dijkstra's Algorithm



Base images from Prof. Seo's slides

SW Solutions

- **SW solution들의 문제점**
  - 속도가 느림
  - 구현이 복잡한
  - ME primitive 실행 중 preemption 될 수 있음
    - 공유 데이터 수정 중은 interrupt를 억제 함으로서 해결 가능
      - Overhead 발생
  - Busy waiting
    - Inefficient

Mutual Exclusion Solutions

- **SW solutions**
  - **Dekker's algorithm** (Peterson's algorithm)
  - **Dijkstra's algorithm**, Knuth's algorithm, Eisenberg and McGuire's algorithm, Lamport's algorithm
- **HW solution**
  - **TestAndSet (TAS) instruction**
- **OS supported SW solution**
  - Spinlock
  - Semaphore
  - Eventcount/sequencer
- **Language-Level solution**
  - Monitor

Synchronization Hardware

- **TestAndSet (TAS) instruction**
  - Test 와 Set을 한번에 수행하는 기계어
  - Machine instruction
    - Atomicity, Indivisible
    - **실행 중 interrupt를 받지 않음 (preemption 되지 않음)**
  - Busy waiting
    - Inefficient

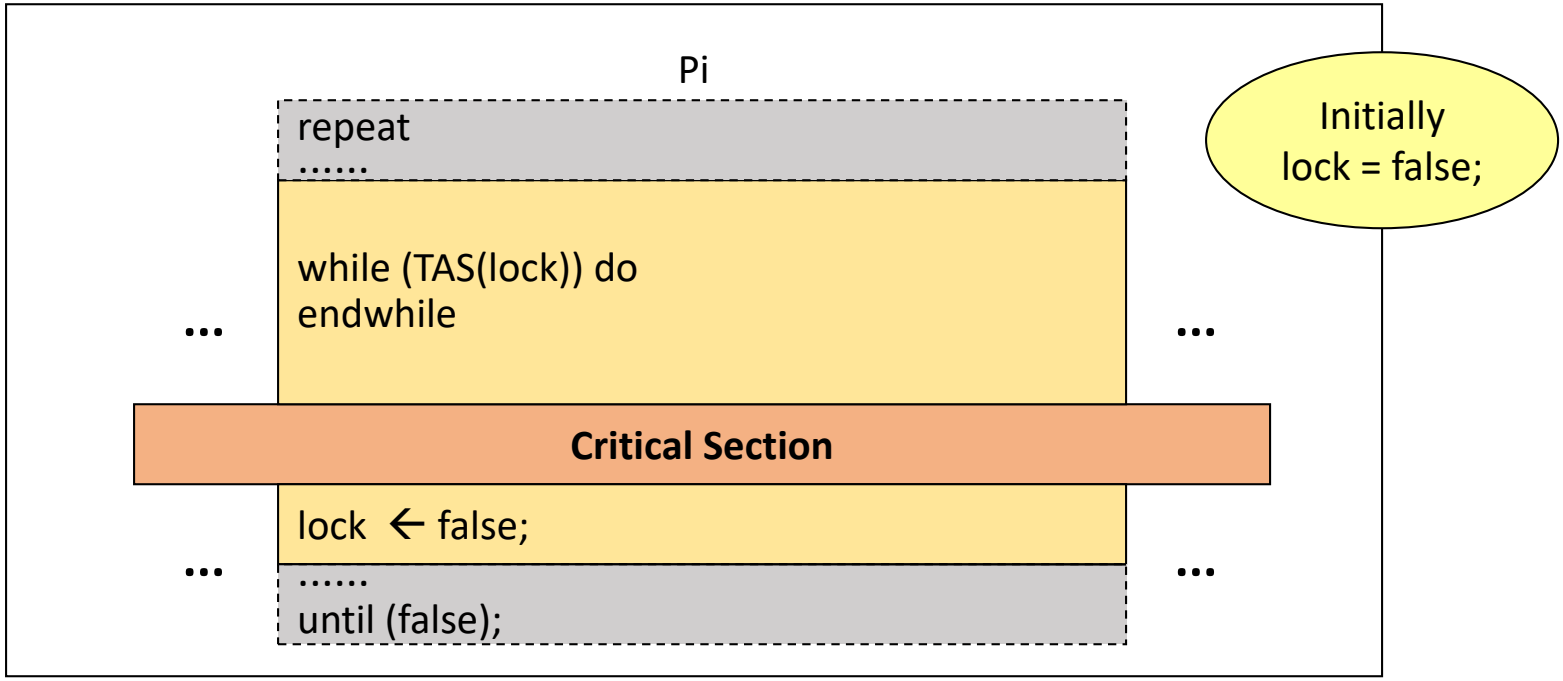
TAS Instruction

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)

ME with TAS Instruction



- **3개 이상의 프로세스의 경우, Bounded waiting 조건 위배**
  - **Why?**

Base images from Prof. Seo's slides

ME with TAS Instruction

- **N-Process mutual exclusion**

```
❶ do                                          // 프로세스 Pi의 진입 영역
  {
    ❷ waiting[i] = true;
      key = true;
    ❸ while (waiting[i] && key)
        ❹ key = TestAndSet(&lock);
    ❺ waiting[i] = false;
          // 임계 영역
          // 탈출 영역
    ❻ j = (i + 1) % n;
    ❼ while ((j != i) && !waiting[j])         // 대기 중인 프로세스를 찾음
          j = (j + 1) % n;
    ❽ if (j == i)                             // 대기 중인 프로세스가 없으면
          lock = false;                       // 다른 프로세스의 진입 허용
    ❾ else                                    // 대기 프로세스가 있으면 다음 순서로 임계 영역에 진입
          waiting[j] = false;                 // Pj가 임계 영역에 진입할 수 있도록
          // 나머지 영역
  } while (true);
```

HW Solution

- **장점**
  - 구현이 간단
- **단점**
  - **Busy waiting**
    - Inefficient
- **Busy waiting 문제를 해소한 상호배제 기법**
  - **Semaphore**
    - 대부분의 os들이 사용하는 기법

Mutual Exclusion Solutions

- **SW solutions**
  - **Dekker's algorithm** (Peterson's algorithm)
  - **Dijkstra's algorithm**, Knuth's algorithm, Eisenberg and McGuire's algorithm, Lamport's algorithm
- **HW solution**
  - **TestAndSet (TAS) instruction**
- **OS supported SW solution**
  - Spinlock
  - Semaphore
  - Eventcount/sequencer
- **Language-Level solution**
  - Monitor

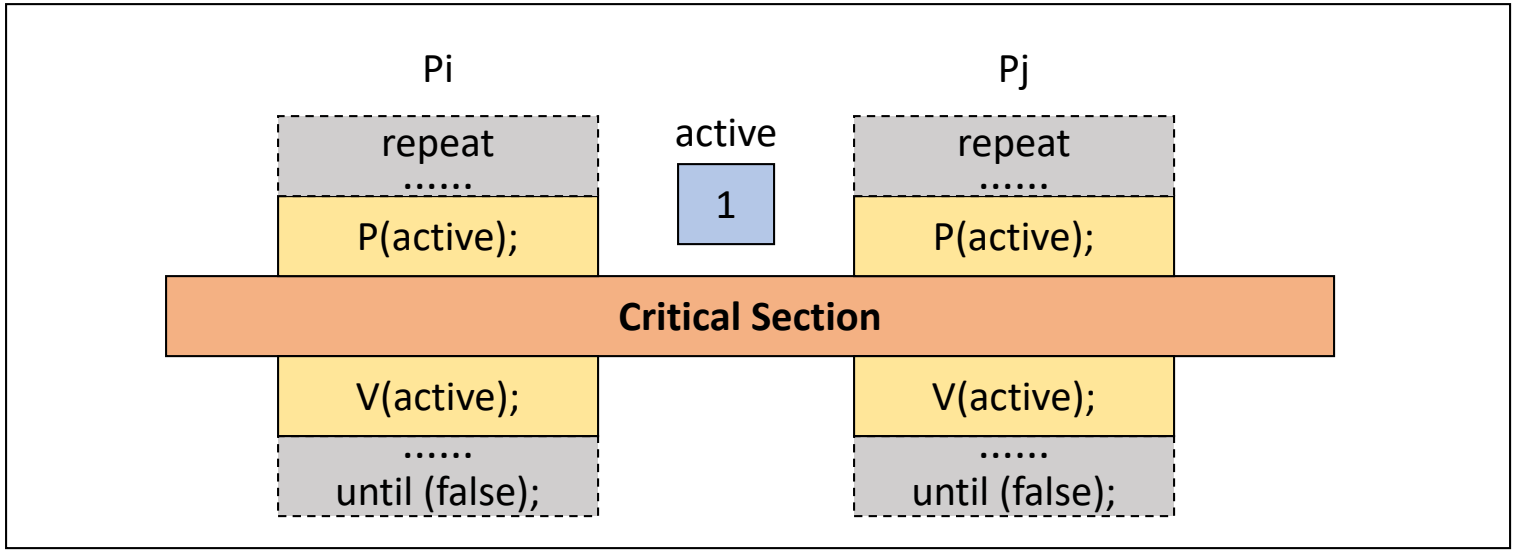
Spinlock

- **정수 변수**
- **초기화, P(), V() 연산으로만 접근 가능**
  - 위 연산들은 indivisible (or atomic) 연산
    - OS support
    - 전체가 한 instruction cycle에 수행 됨

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

```
V(S) {
  S ← S + 1;
}
```

Spinlock



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

Base images from Prof. Seo's slides

Spinlock

- 멀티 프로세서 시스템에서만 사용 가능
- **Busy waiting!**

Semaphore

- 1965년 Dijkstra가 제안
- Busy waiting 문제 해결

- 음이 아닌 정수형 변수(S)
  - 초기화 연산, P(), V()
    로만 접근 가능
    - P: Probern (검사)
    - V: Verhogen (증가)
- 임의의 s 변수 하나에
  ready queue 하나가 할당 됨

Semaphore

- **Binary semaphore**
  - S가 0과 1 두 종류의 값만 갖는 경우
  - 상호배제나 프로세스 동기화의 목적으로 사용
- **Counting semaphore**
  - S가 0이상의 정수값을 가질 수 있는 경우
  - Producer-Consumer 문제 등을 해결하기 위해 사용
    - 생산자-소비자 문제

Semaphore

- **초기화 연산**
  - S 변수에 초기값을 부여하는 연산
- **P() 연산, V() 연산**

P(S)연산

if (S > 0)
 then S ← S - 1;
 else wait on the queue Q_s ;

V(S)연산

if (∃ waiting processes on Q_s)
 then wakeup one of them;
 else S ← S + 1;

- **모두 indivisible 연산**
  - OS support
  - 전체가 한 instruction cycle에 수행 됨

Semaphore in OSs

- **Windows**
  - MSDN : https://msdn.microsoft.com/en-us/library/windows/desktop/ms685129(v=vs.85).aspx

```C++
HANDLE WINAPI CreateSemaphore(
  _In_opt_ LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
  _In_     LONG                  lInitialCount,
  _In_     LONG                  lMaximumCount,
  _In_opt_ LPCTSTR               lpName
);
```

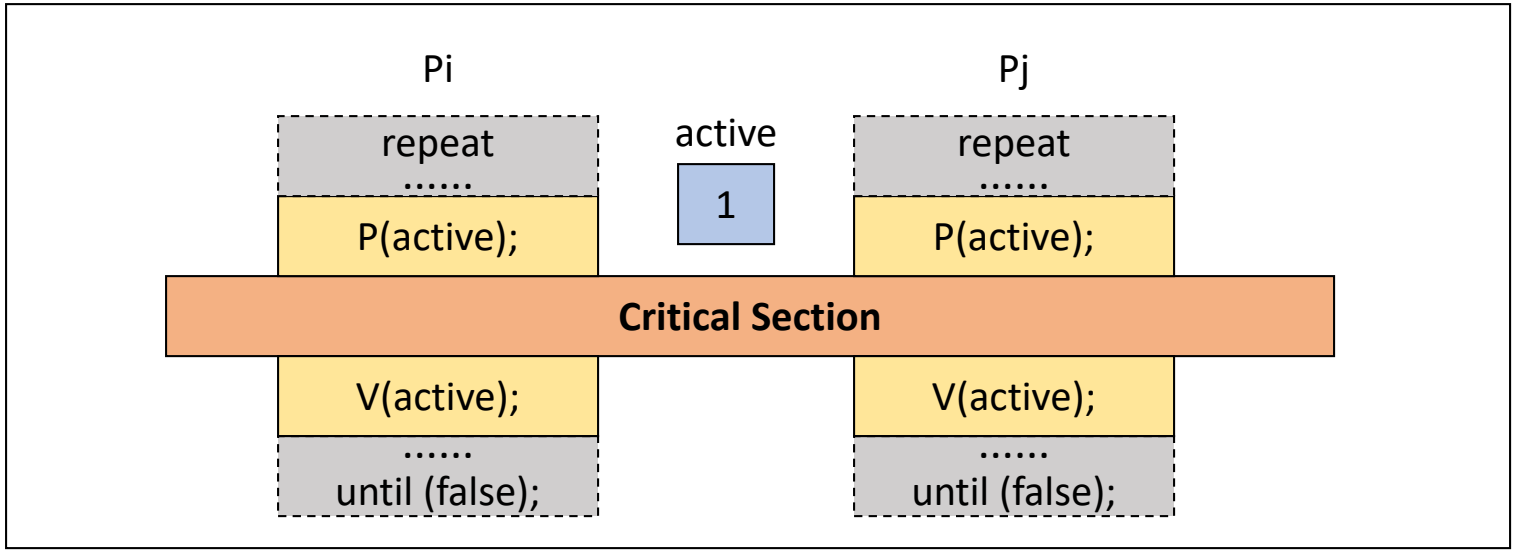
- **Unix/Linux**
  - System V : https://docs.oracle.com/cd/E19683-01/816-5042/auto32/index.html

Semaphore

- **Semaphore로 해결 가능한 동기화 문제들**
  - 상호배제 문제
    - Mutual exclusion
  - 프로세스 동기화 문제
    - process synchronization problem
  - 생산자-소비자 문제
    - producer-consumer problem
  - Reader-writer 문제
  - Dining philosopher problem
  - 기타

Semaphore

- **Mutual exclusion**

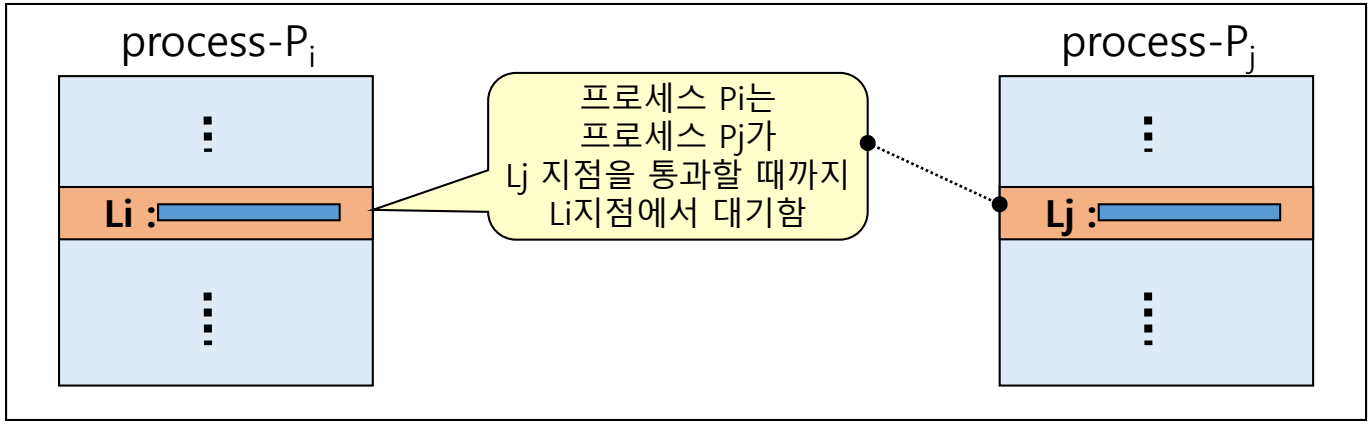


- active = 1 : 임계 지역을 실행중인 프로세스 없음
- active = 0 : 임계 지역을 실행중인 프로세스 있음

Base images from Prof. Seo's slides

Semaphore

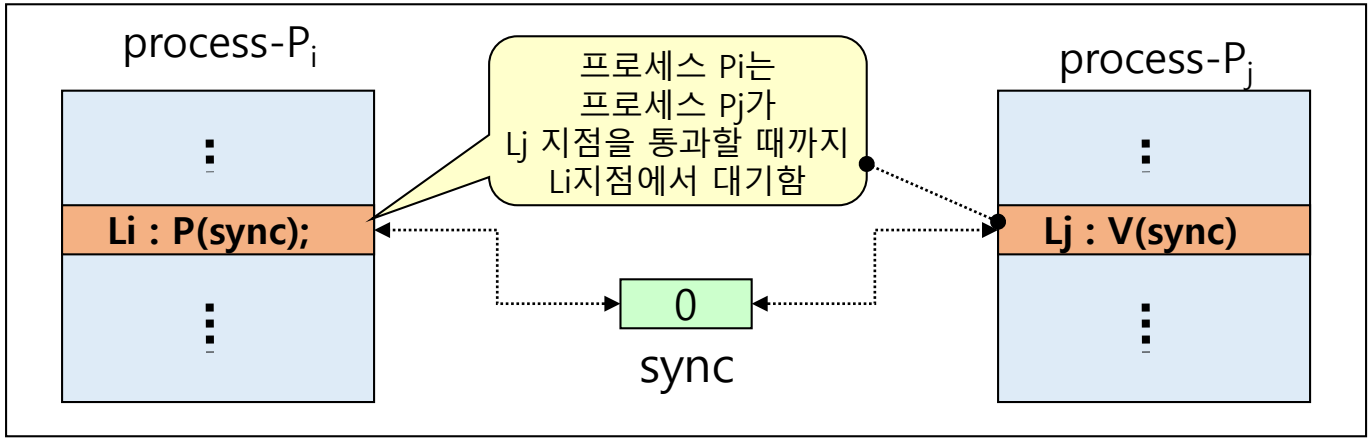
- **Process synchronization**
  - Process들의 실행 순서 맞추기
    - 프로세스들은 병행적이며, 비동기적으로 수행



Base images from Prof. Seo's slides

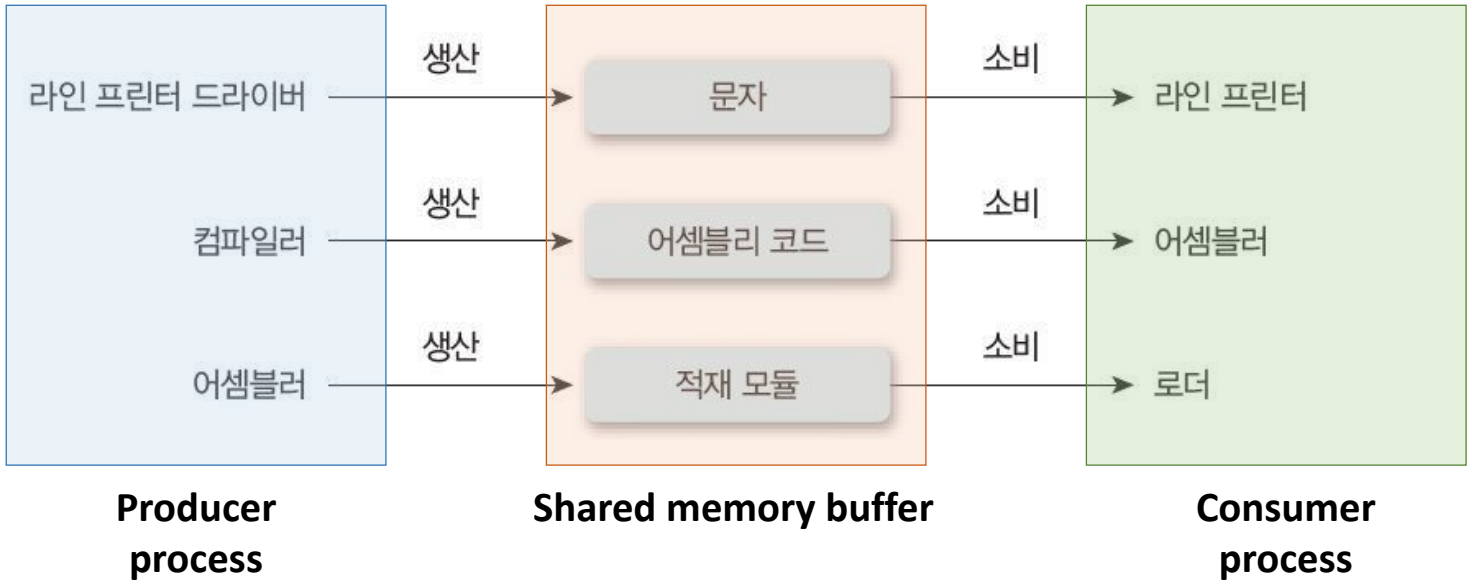
Semaphore

- **Process synchronization**
  - Process들의 실행 순서 맞추기
    - 프로세스들은 병행적이며, 비동기적으로 수행



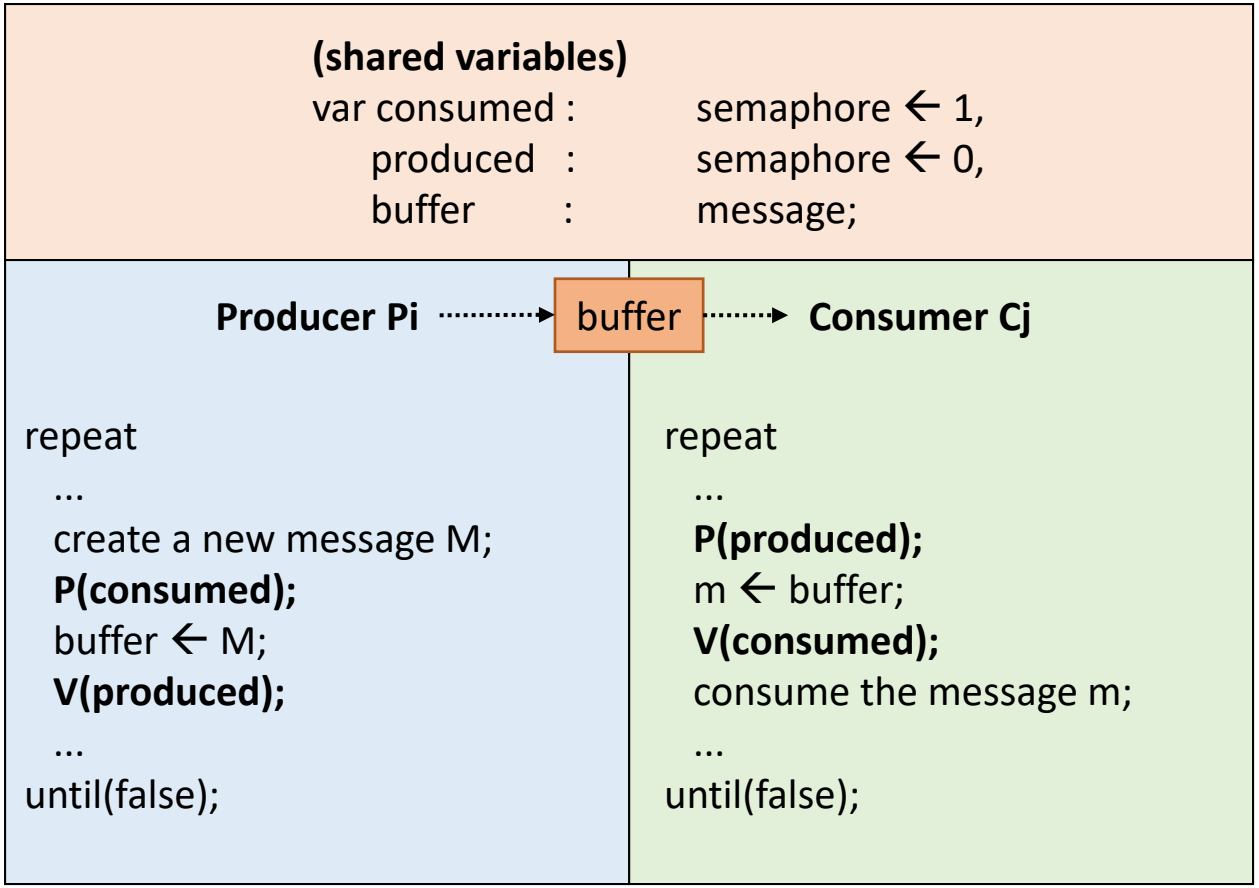
Semaphore

- **Producer-Consumer problem**
  - 생산자(Producer) 프로세스
    - 메시지를 생성하는 프로세스 그룹
  - 소비자(Consumer) 프로세스
    - 메세지 전달받는 프로세스 그룹



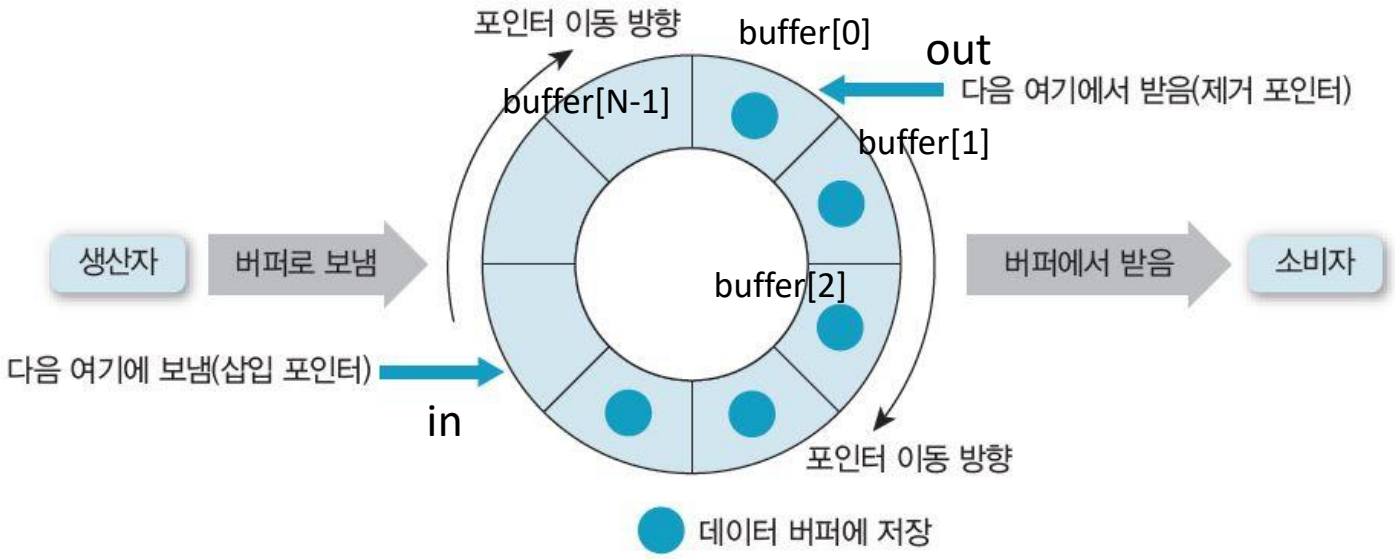
Semaphore

- **Producer-Consumer problem with single buffer**



Semaphore

- **Producer-Consumer problem with N-buffers**



Semaphore

- **Producer-Consumer problem with N-buffers**

```
(shared variables)
var nrfull   :   semaphore ← 0,
    nrempty  :   semaphore ← N,
    mutexP   :   semaphore ← 1,
    mutexC   :   semaphore ← 1,
    buffer   :   array[0..N-1] of message,
    in, out  :   0..N-1 ← 0,0;
```

Producer Pi ⋯▸ buffers ⋯▸ **Consumer Cj**

```
repeat
  ...
  create a new message M;
  P(mutexP);
  P(nrempty);
  buffer[in] ← M;
  in ← (in + 1) mod N;
  V(nrfull);
  V(mutexP);
  ...
until(false);
```

```
repeat
  ...
  P(mutexC);
  P(nrfull);
  m ← buffer[out];
  out ← (out + 1) mod N;
  V(nrempty);
  V(mutexC);
  ...
until(false);
```

Semaphore

- **Reader-Writer problem**
  - **Reader**
    - 데이터에 대해 읽기 연산만 수행
  - **Writer**
    - 데이터에 대해 갱신 연산을 수행
  - **데이터 무결성 보장 필요**
    - Reader들은 동시에 데이터 접근 가능
    - Writer들(또는 reader와 write)이 동시 데이터 접근 시, 상호배제(동기화) 필요
  - **해결법**
    - reader / writer 에 대한 우선권 부여
      - reader preference solution
      - writer preference solution

Semaphore

- **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);
```

Semaphore

- **No busy waiting**
  - 기다려야 하는 프로세스는 block(asleep)상태가 됨
- **Semaphore queue에 대한 wake-up 순서는 비결정적**
  - Starvation problem

Eventcount/Sequencer

- 은행의 번호표와 비슷한 개념

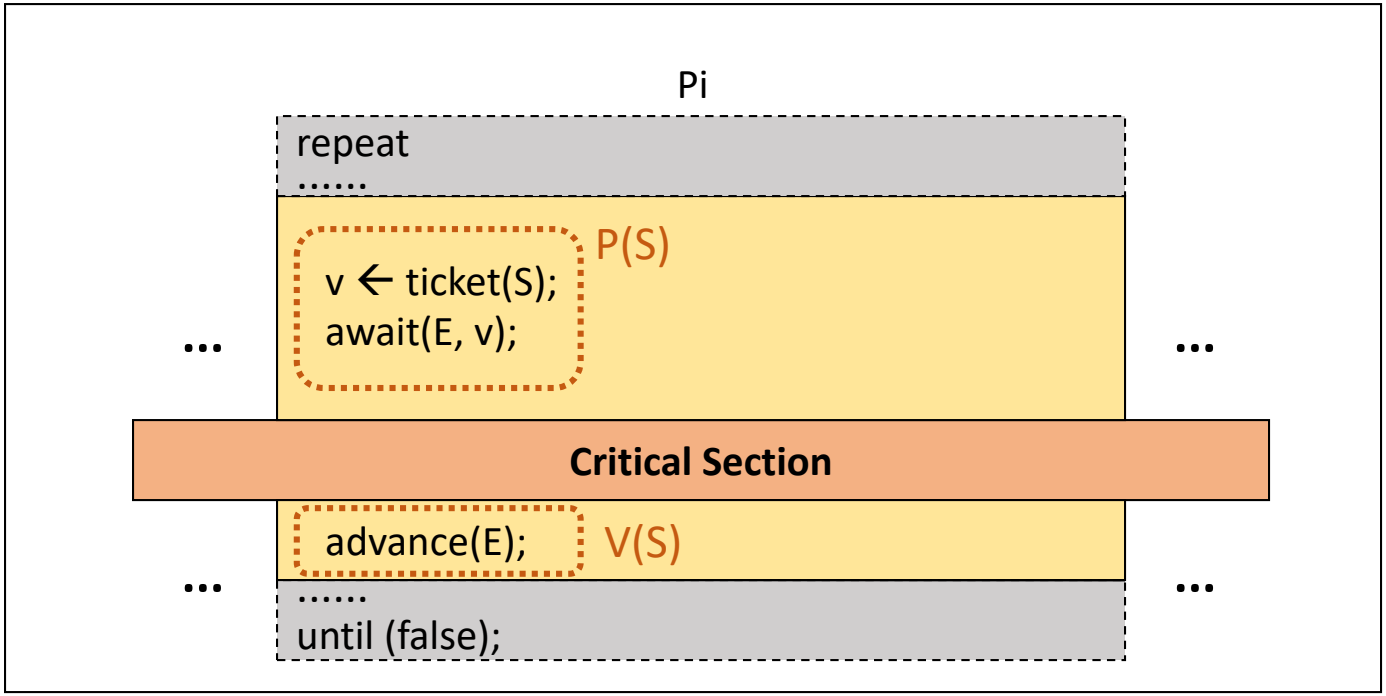
- **Sequencer**
  - 정수형 변수
  - 생성시 0으로 초기화, 감소하지 않음
  - 발생 사건들의 순서 유지
  - ticket() 연산으로만 접근 가능
- **ticket(S)**
  - 현재까지 ticket() 연산이 호출 된 횟수를 반환
  - Indivisible operation

Eventcount/Sequencer

- **Eventcount**
  - 정수형 변수
  - 생성시 0으로 초기화, 감소하지 않음
  - 특정 사건의 발생 횟수를 기록
  - read(E), advance(E), await(E, v) 연산으로만 접근 가능
- **read(E)**
  - 현재 Eventcount 값 반환
- **advance (E)**
  - E ← E + 1
  - E를 기다리고 있는 프로세스를 깨움 (wake-up)
- **await(E, v)**
  - V는 정수형 변수
  - if (E < v) 이면 E에 연결된 Q_E에 프로세스 전달(push) 및 CPU scheduler 호출

Eventcount/Sequencer

- **Mutual exclusion**



Base images from Prof. Seo's slides

Eventcount/Sequencer

- **Producer-Consumer problem**

(shared variables)

```
var Pticket, Cticket   : sequencer,
    In, Out            : eventcount,
    buffer             : array[0..N-1] of message;
```

Producer Pi

```
var t : integer;

repeat
  ...
  Create a new message M;
  t ← ticket(Pticket);
  await(In, t);
  await(Out, t - N + 1);
  buffer[t mod N] ← M;
  advance(In);
  ...
until(false)
```

Consumer Cj

```
var u : integer;

repeat
  ...
  u ← ticket(Cticket);
  await(Out, u);
  await(In, u + 1);
  m ← buffer[u mod N] ;
  advance(Out);
  Consume the message m;
  ...
until(false)
```

Eventcount/Sequencer

- **No busy waiting**
- **No starvation**
  - FIFO scheduling for Q_E
- **Semaphore 보다 더 low-level control이 가능**

Mutual Exclusion Solutions

- **SW solutions**
  - **Dekker's algorithm** (Peterson's algorithm)
  - **Dijkstra's algorithm**, Knuth's algorithm, Eisenberg and McGuire's algorithm, Lamport's algorithm
- **HW solution**
  - **TestAndSet (TAS) instruction**
- **OS supported SW solution**
  - Spinlock
  - Semaphore
  - Eventcount/sequencer

Low-level mechanisms

- Felxible
- Difficult to use
  - Error-prone

- **Language-Level solution**
  - Monitor

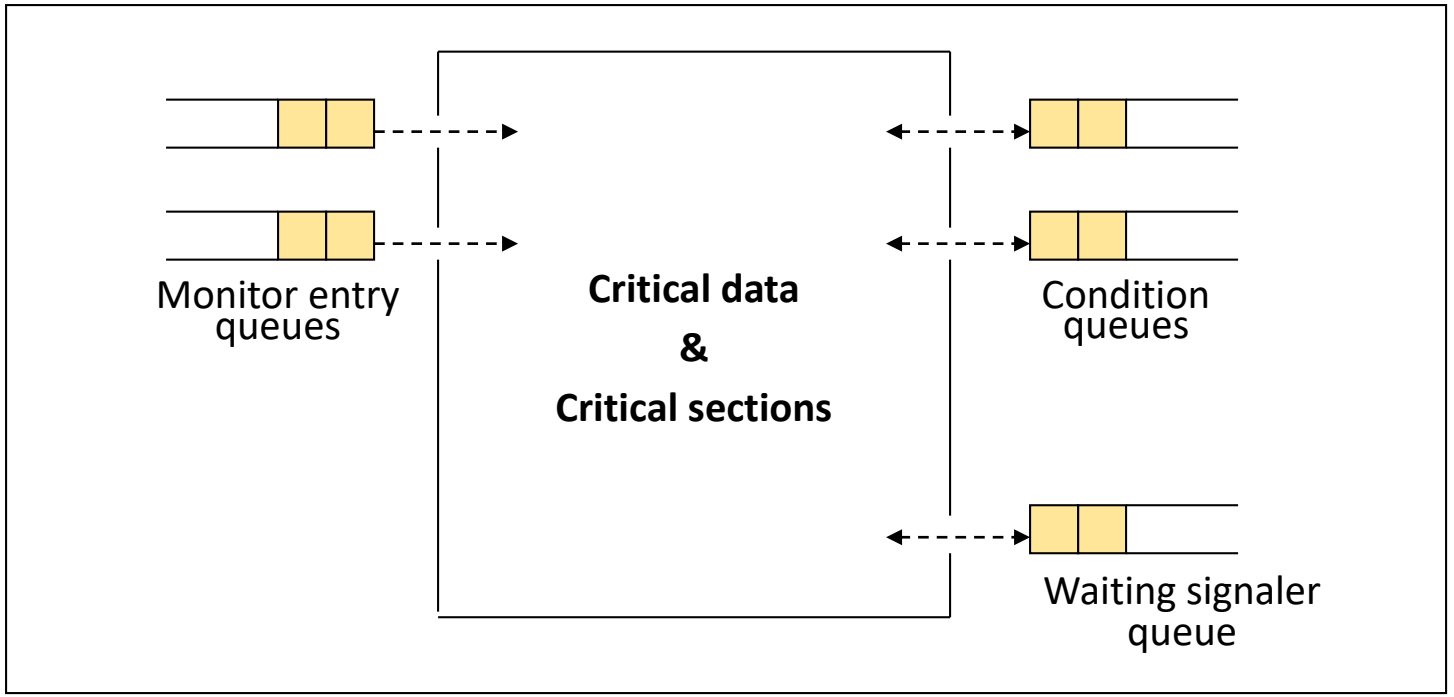
High-level Mechanism

- **Monitor**
- Path expressions
- Serializers
- Critical region, conditional critical region

- **Language-level constructs**
- **Object-Oriented concept과 유사**
- **사용이 쉬움**

Monitor

- 공유 데이터와 Critical section의 집합
- **Conditional variable**
  - wait(), signal() operations



Base images from Prof. Seo's slides

Monitor의 구조

- **Entry queue (진입 큐)**
  - 모니터 내의 procedure 수만큼 존재
- **Mutual exclusion**
  - 모니터 내에는 항상 하나의 프로세스만 진입 가능
- **Information hiding (정보 은폐)**
  - 공유 데이터는 모니터 내의 프로세스만 접근 가능
- **Condition queue (조건 큐)**
  - 모니터 내의 특정 이벤트를 기다리는 프로세스가 대기
- **Signaler queue (신호제공자 큐)**
  - 모니터에 항상 하나의 신호제공자 큐가 존재
  - signal() 명령을 실행한 프로세스가 임시 대기

Monitor in Languages

Monitor Class

.NET Framework (current version) | Other Versions

Note

The .NET API Reference documentation has a new home. Visit the .NET API Browser on docs.microsoft.com to see the new experience.

Provides a mechanism that synchronizes access to objects.

Namespace: System.Threading
Assembly: mscorlib (in mscorlib.dll)

Inheritance Hierarchy

System.Object
  System.Threading.Monitor

Syntax

C# | C++ | F# | VB

```
[ComVisibleAttribute(true)]
[HostProtectionAttribute(SecurityAction.Link
        ExternalThreading = true)]
public static class Monitor
```

Class Monitor

java.lang.Object
  javax.management.NotificationBroadcasterSupport
    javax.management.monitor.Monitor

All Implemented Interfaces:

MBeanRegistration, MonitorMBean, NotificationBroadcaster, NotificationEmitter

Direct Known Subclasses:

CounterMonitor, GaugeMonitor, StringMonitor

```
public abstract class Monitor
extends NotificationBroadcasterSupport
implements MonitorMBean, MBeanRegistration
```

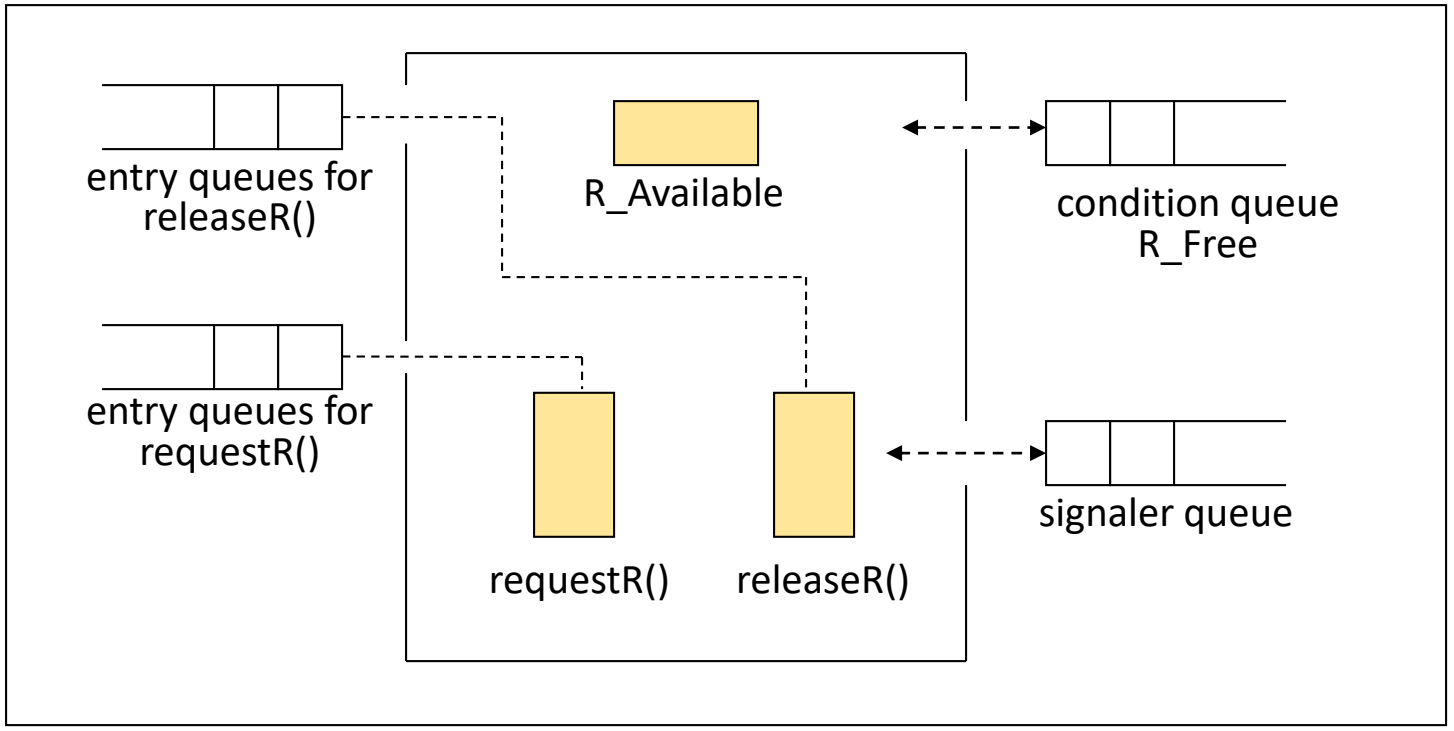
Defines the part common to all monitor MBeans. A monitor MBean monitors values of an attribute common to a set of observed MBeans. The observed attribute is monitored at intervals specified by the granularity period. A gauge value (derived gauge) is derived from the values of the observed attribute.

Since:

1.5

Monitor

- 자원 할당 문제



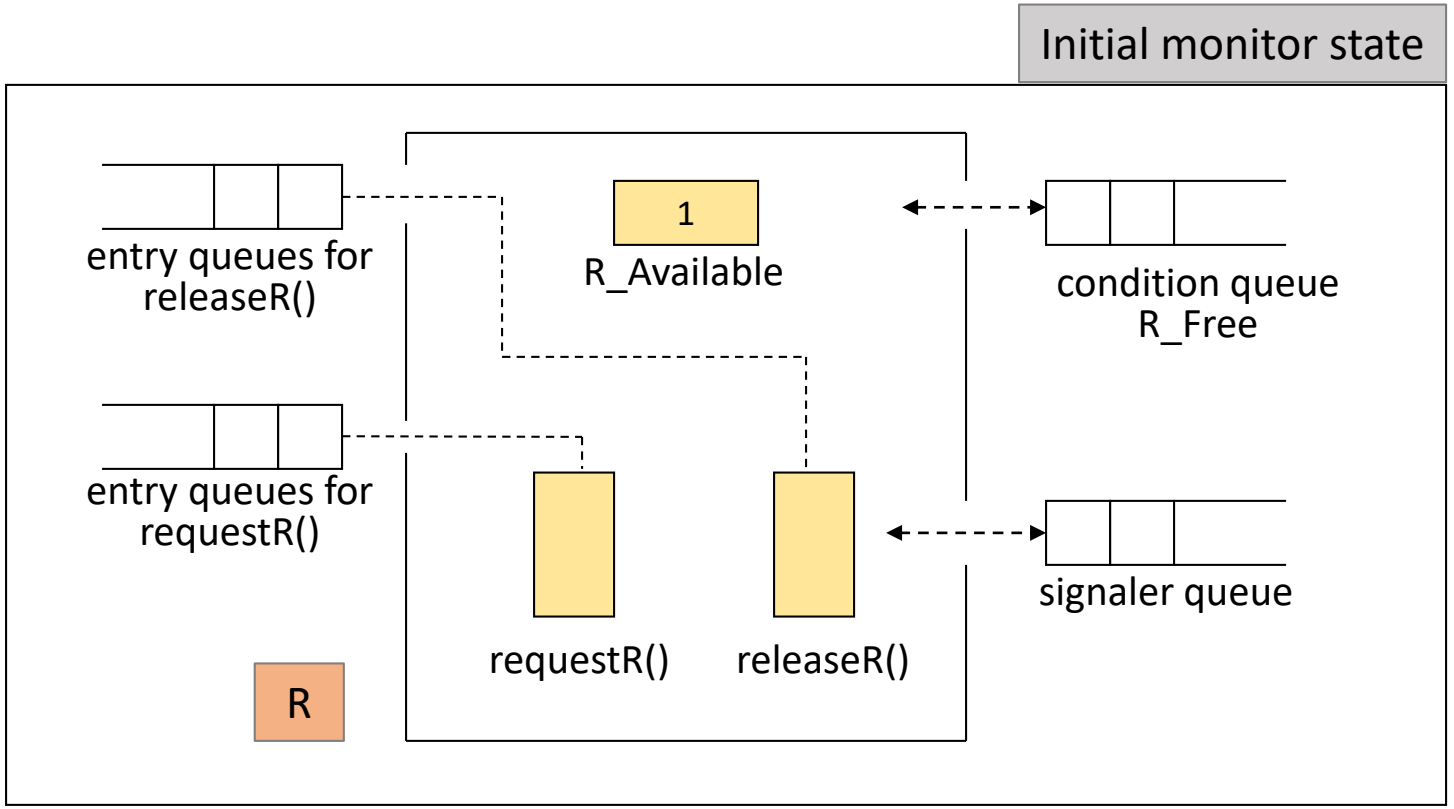
Monitor

- 자원 할당 문제

```
monitor resourceRiAllocator;
var RiIsAvailable : boolean,
    RiIsFree      : condition;
procedure requestR();
begin
  if (¬R_Available) then
     R_Free.wait();
  R_Available ← false;
end;
procedure releaseR():
begin
  R_Available ← true;
  R_Free.signal();
end;

begin
  RiIsAvailable ← true;
end.
```

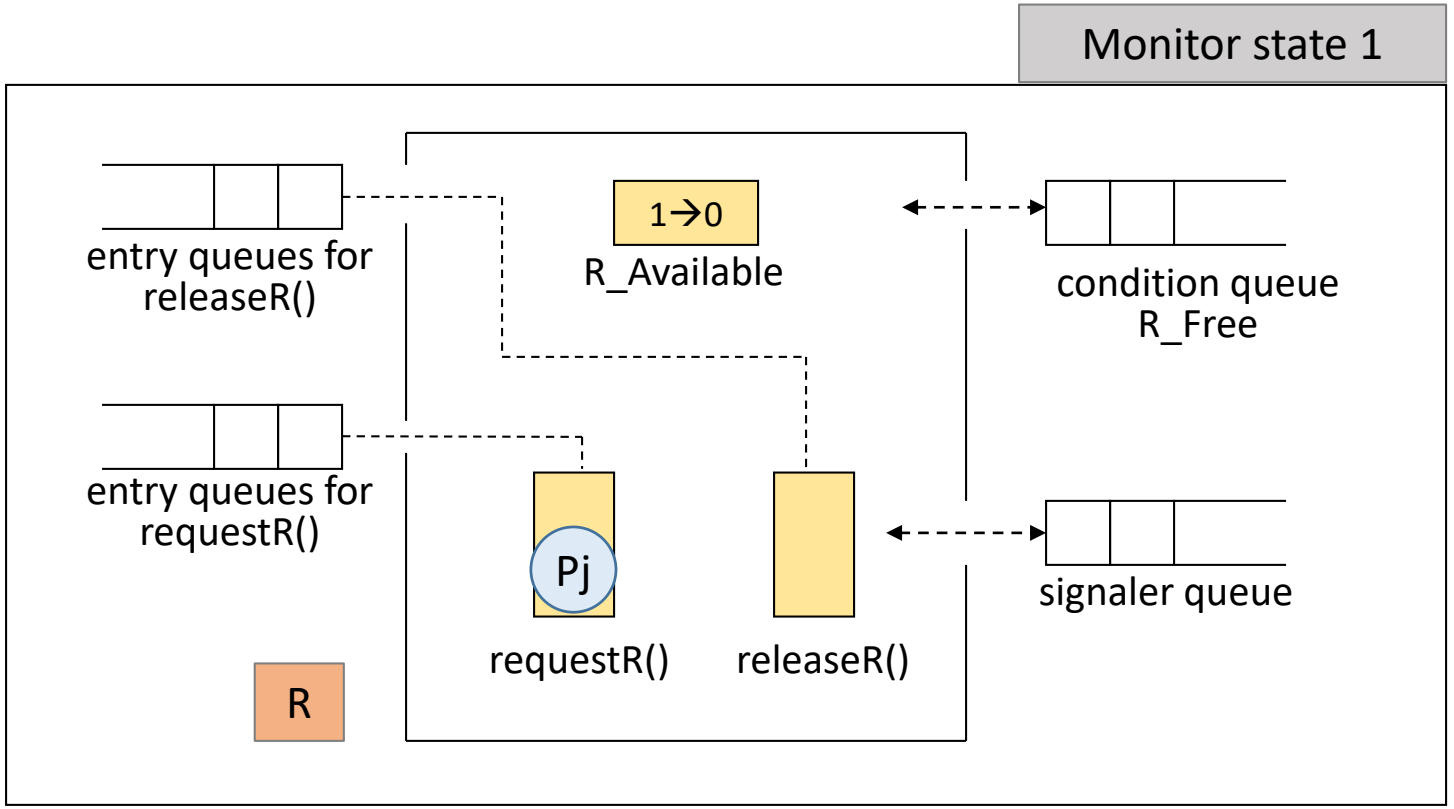
Monitor

- 자원 할당 시나리오
  - 자원 R 사용 가능
  - Monitor 안에 프로세스 없음



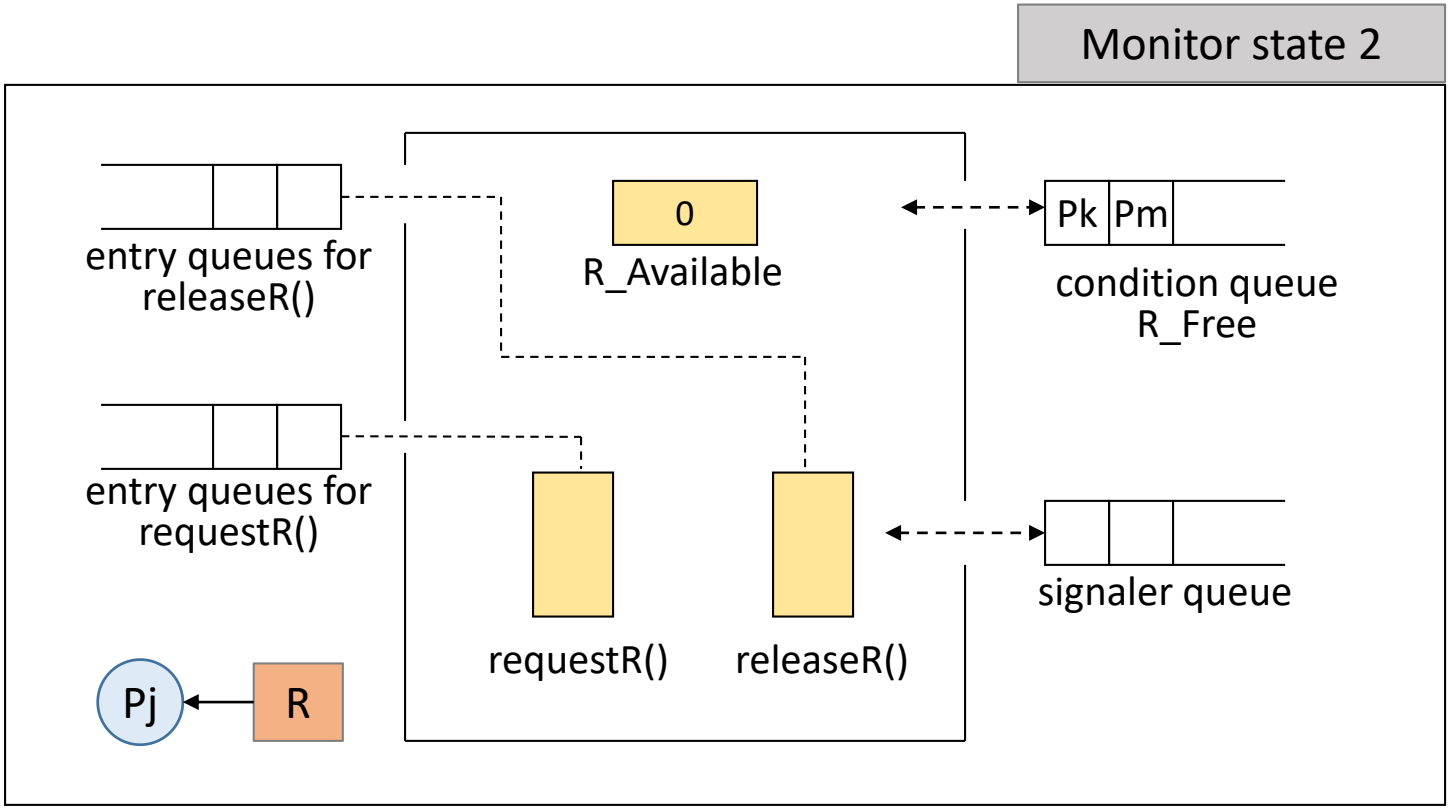
Monitor

- 자원 할당 시나리오
  - 프로세스 Pj가 모니터 안에서 자원 R을 요청



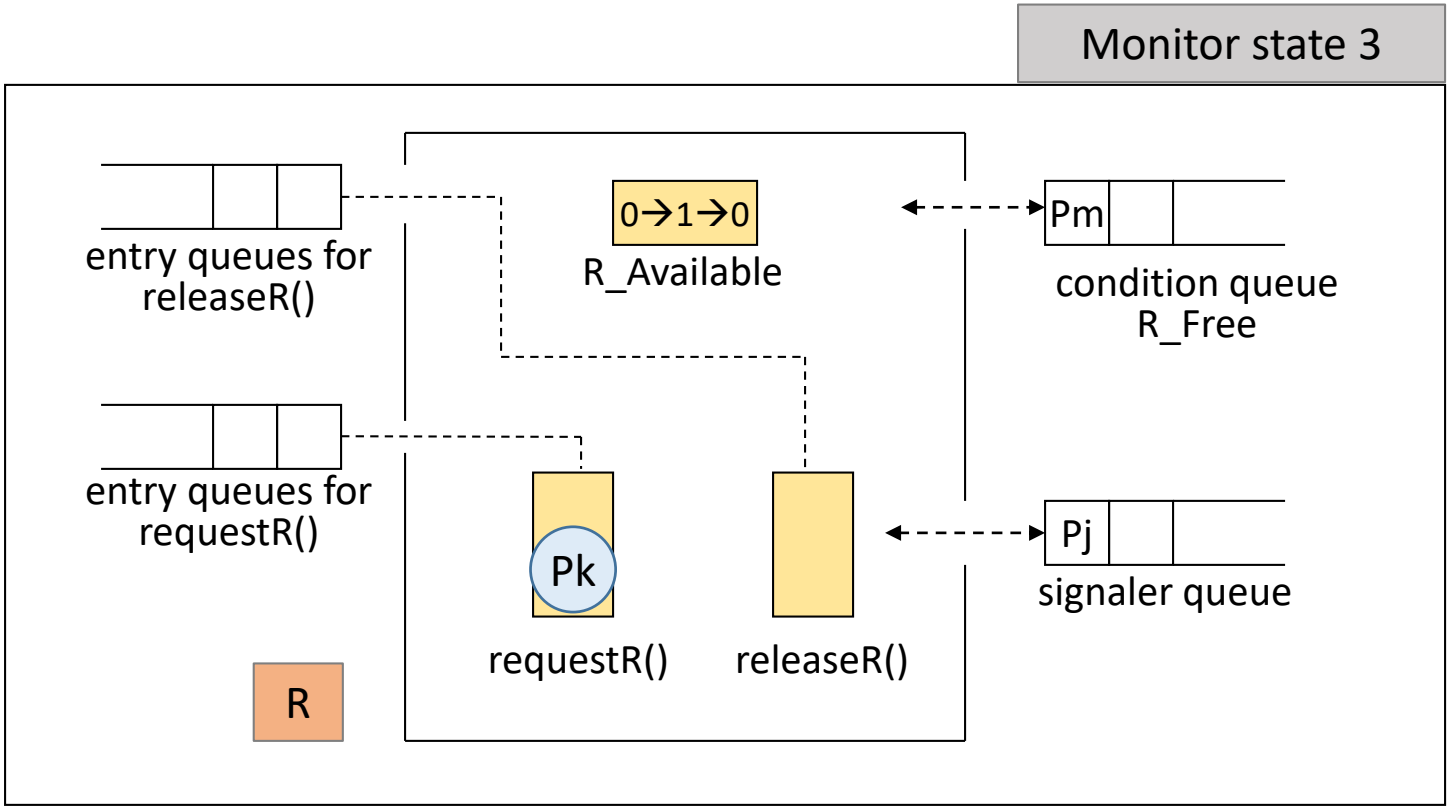
Monitor

- 자원 할당 시나리오
  - 자원 R이 Pj에게 할당 됨
  - 프로세스 Pk 가 R 요청, Pm 또한 R요청



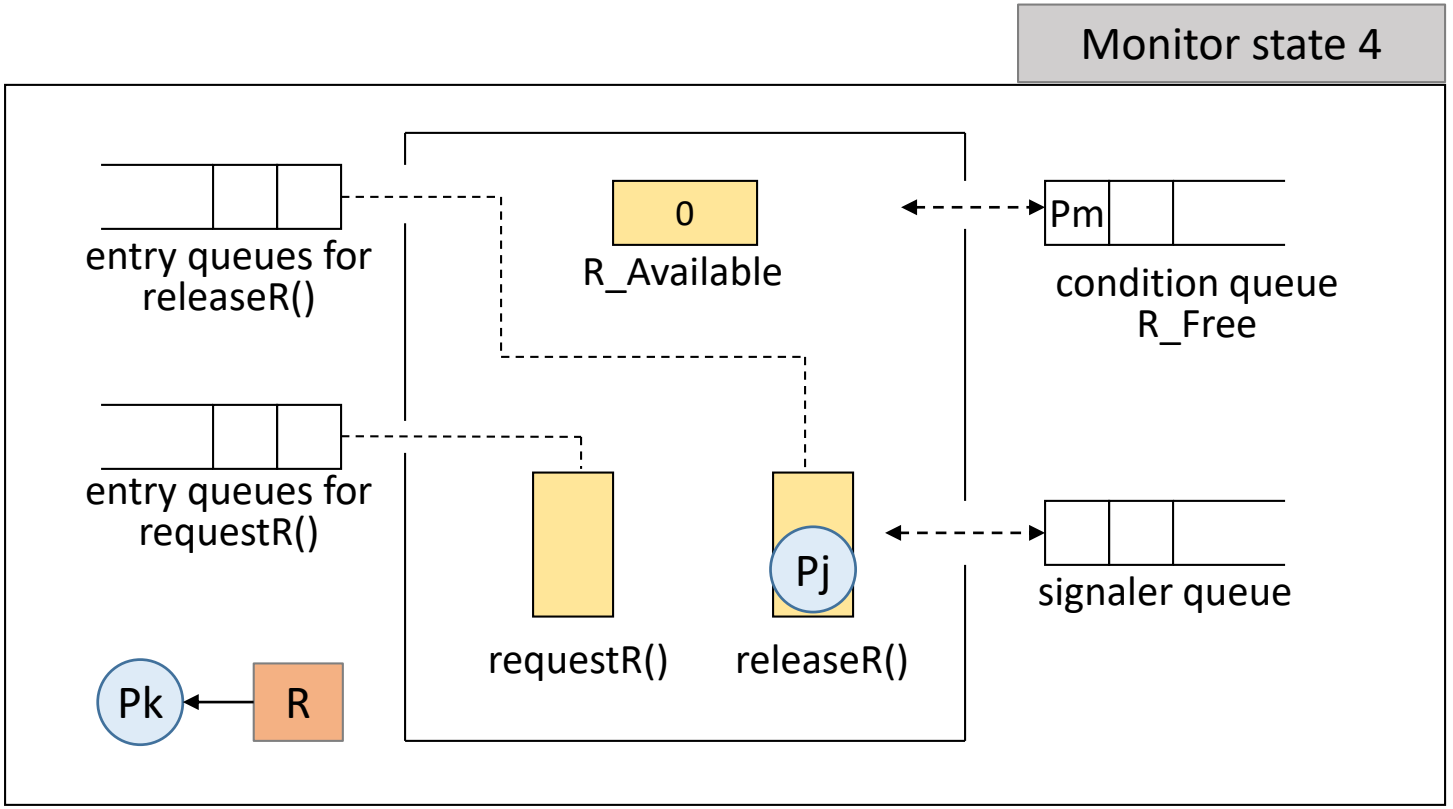
Monitor

- 자원 할당 시나리오
  - Pj가 R 반환
  - R_Free.signal() 호출에 의해 Pk가 wakeup



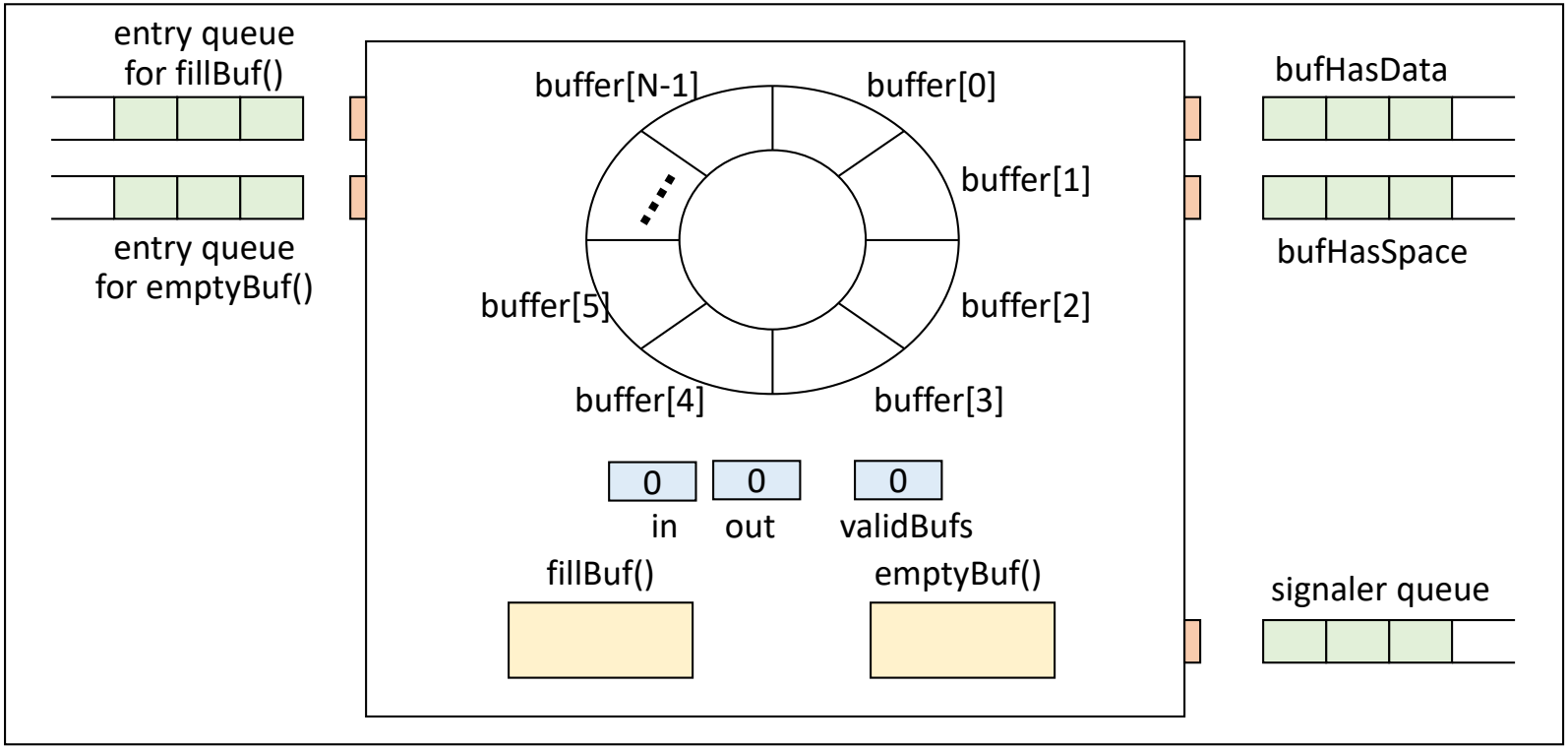
Monitor

- 자원 할당 시나리오
  - 자원 R이 Pk에게 할당 됨
  - Pj가 모니터 안으로 돌아와서, 남은 작업 수행



Monitor

- **Producer-Consumer Problem**



Monitor (Producer-Consumer Problem)

```
monitor ringbuffer;
var buffer : array[0..N-1] of message,
    validBufs : 0..N,
    in : 0..N-1,
    out : 0..N-1,
    vufHasData, bufHasSpace : condition;

procedure fillBuf(data : message);
begin
  if (validBufs = N) then bufHasSpace.wait();
  buffer[in] ← data;
  validBufs ← validBufs + 1;
  in ← (in + 1) mod N;
  vufHasData.signal();
end;

procedure emptyBuf(var data : message):
begin
  if (validBufs = 0) then bufHasData.wait();
  data ← buffer[out];
  validBufs ← validBufs - 1;
  out ← (out + 1) mod N;
  bufHasSpace.signal();
end;

begin
  validBufs ← 0;
  in ← 0;
  out ← 0;
end
```

Monitor

- **Reader-Writer Problem**
  - reader/writer 프로세스들간의 데이터 무결성 보장 기법
  - writer 프로세스에 의한 데이터 접근 시에만 상호 배제 및 동기화 필요함

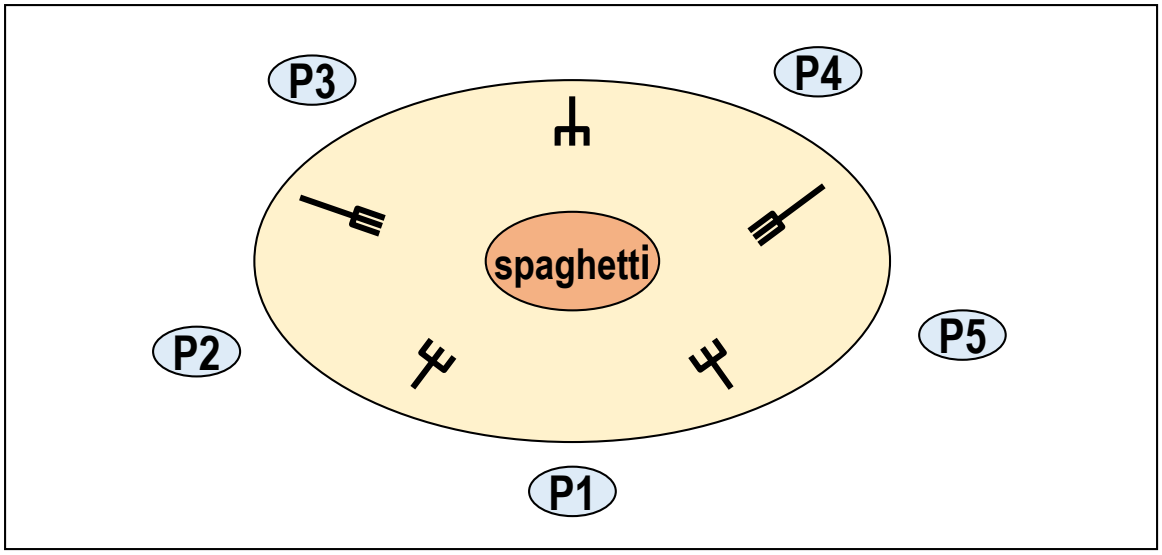
  - **모니터 구성**
    - 변수 2개
      - 현재 읽기 작업을 진행하고 있는 reader 프로세스의 수
      - 현재 writer 프로세스가 쓰기 작업을 진행 중인지 표시
    - 조건 큐 2개
      - reader/writer프로세스가 대기해야 할 경우에 사용
    - 프로시져 4개
      - reader(writer) 프로세스가 읽기(쓰기) 작업을 원할 경우에 호출, 읽기(쓰기) 작업을 마쳤을 때 호출

Monitor (Reader-Writer Problem)

```
monitor readers_and_writers;
var numReaders : integer,
    writing : boolean,
    readingAllowed, writingAllowed : conditon;
procedure beginReading;
begin
  if (writing or queue(writingAllowed)) then readingAllowed.wait();
  numReaders ← numReaders + 1;
  if (queue(readingAllowed)) then readingAllowed.signal();
end;
procedure finishReading:
begin
  numReaders ← numReaders - 1;
  if (numReaders = 0) then writingAllowed.signal();
end;

procedure beginWriting;
begin
  if (numReaders > 0 or writing) then writingAllowed.wait()
  writing ← true;
end;
procedure finishWriting:
begin
  writing ← false;
  if (queue(readingAllowed)) then readingAllowed.signal();
                             else writingAllowed.signal();
end;
begin
  numReaders ← 0;
  writing ← false;
end.
```

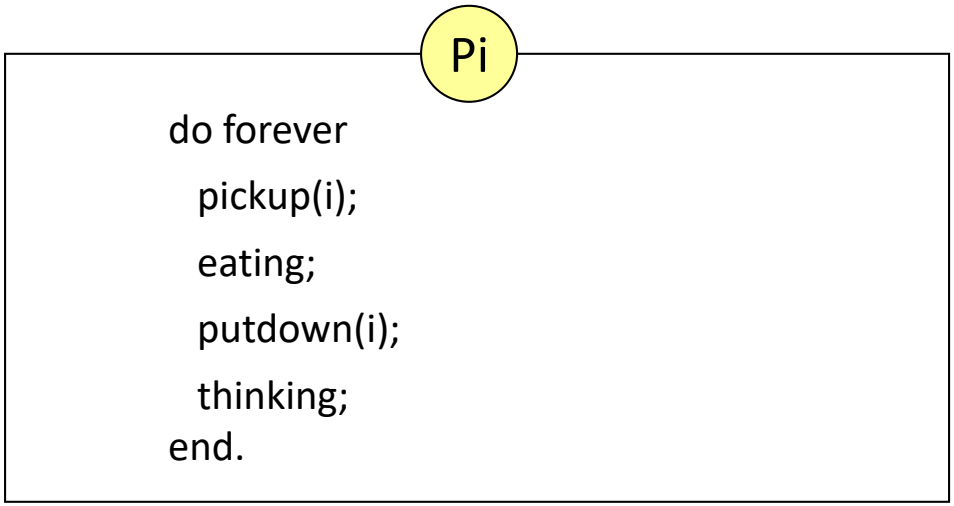
Monitor

- **Dining philosopher problem**
  - 5명의 철학자
  - 철학자들은 생각하는 일, 스파게티 먹는 일만 반복함
  - 공유 자원 : 스파게티, 포크
  - 스파게티를 먹기 위해서는 좌우 포크 2개 모두 들어야 함



Monitor

- **Dining philosopher problem**

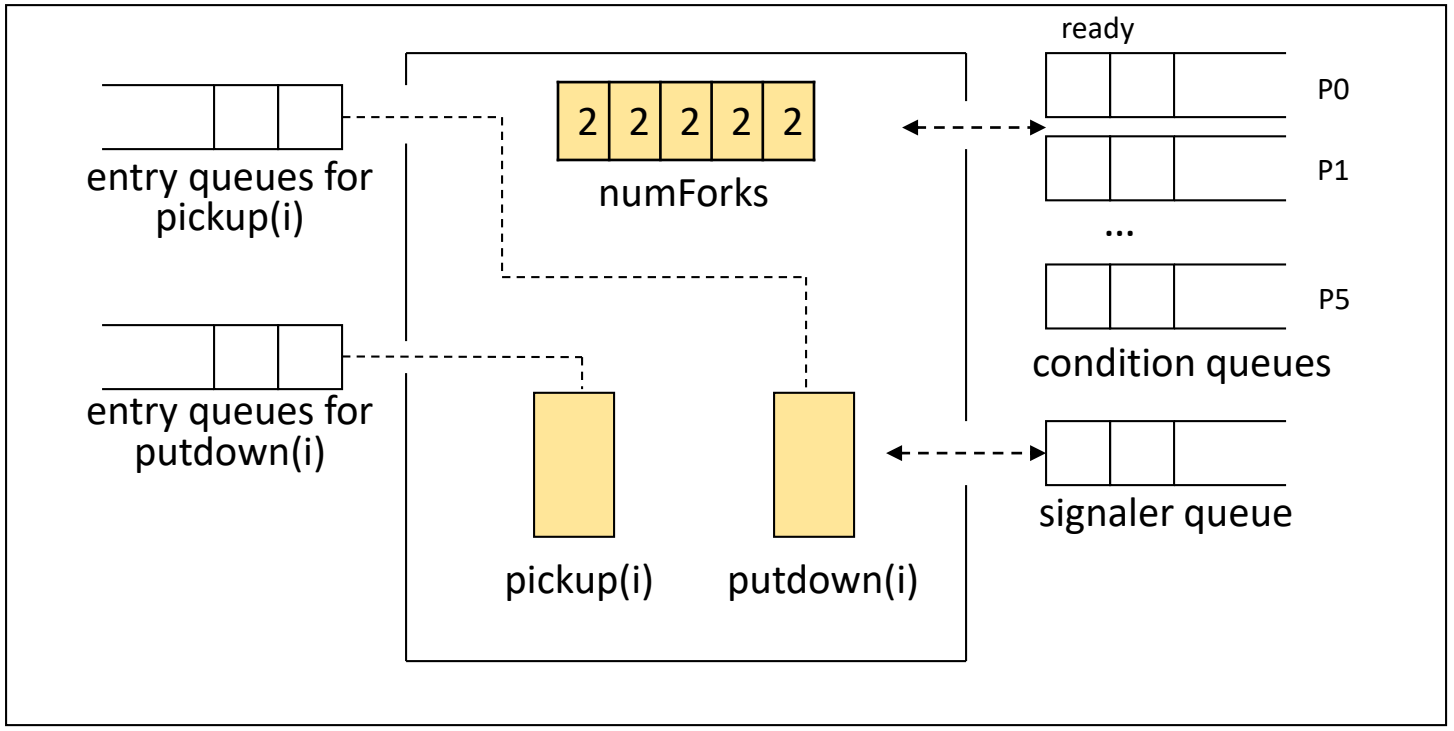


Monitor (Dining philosopher problem)

```
monitor dining_philosophers;
var numForks : array[0..4] of integer,
    ready    : array[0..4] of condition,
    i        : integer;

procedure pickup(me);
var
  me : integer;
begin
  if (numForks[me] ≠ 2) then ready[me].wait();
  numForks[right(me)] ← numForks[right(me)] - 1;
  numForks[left(me)] ← numForks[left(me)]  - 1;
end;

procedure putdown(me):
var
  me : integer;
begin
  numForks[right(me)] ← numForks[right(me)] + 1;
  numForks[left(me)] ← numForks[left(me)]  + 1;
 if (numForks[right(me)] = 2) then ready(right(me)].signal();
 if (numForks[left(me)] = 2) then ready(left(me)].signal();
end;

begin
  for i ← 0 to 4 do
      numForks[i] ← 2;
end.
```

Monitor

- **Dining philosopher problem**



Monitor

- **장점**
  - 사용이 쉽다
  - Deadlock 등 error 발생 가능성이 낮음
- **단점**
  - 지원하는 언어에서만 사용 가능
  - 컴파일러가 OS를 이해하고 있어야 함
    - Critical section 접근을 위한 코드 생성

요약

- **동기화 (Synchronization)**
- **상호배제 (Mutual exclusion)**

- **Low-level mechanism**
  - SW solution, HW solution (TAS operation)
  - OS-supported SW solutions
    - Semaphore, eventcount/sequencer
- **High-level mechanism**
  - Language-level solutions
    - Monitor