3주차 Process Synchronization and Mutual Exclusion

Process Synchronization (동기화)

- 다중 프로그래밍 시스템
  - 여러 개의 프로세스들이 존재
  - 프로세스들은 서로 독립적으로 동작 (동시에 동작)
  - 공유 자원 또는 데이터가 있을 때, 문제 발생 가능
- 동기화 (Synchronization)
  - 프로세스들이 서로 동작을 맞추는 것
  - 프로세스들이 서로 정보를 공유 하는 것

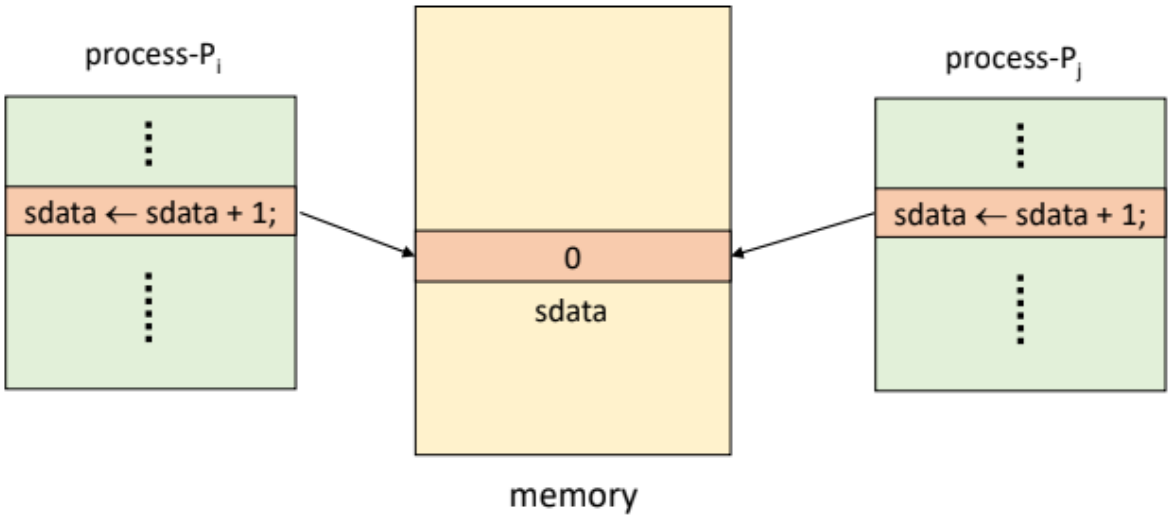
Asynchronous and Concurrent P’s

- 비동기적(Asynchronous) : 프로세스들이 서로에 대해 모름
- 병행적 (Concurrent) : 여러 개의 프로세스들이 동시에 시스템에 존재
- 병행 수행 중인 비동기적 프로세스들이 공유 자원에 동시 접근 할 때 문제가 발생 할 수 있음

Terminologies

- Shared data (Critical data, 공유 데이터) : 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

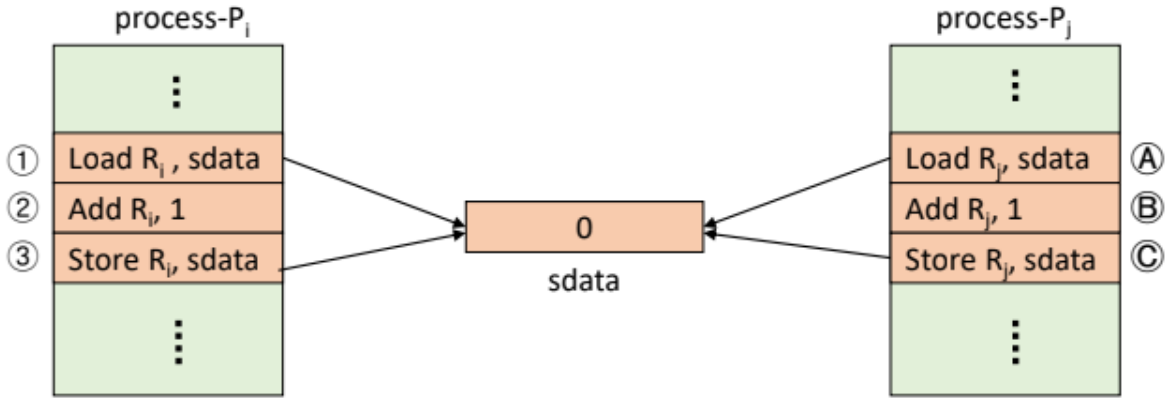
Critical Section



Note

기계어 명령(machine instruction)의 특성

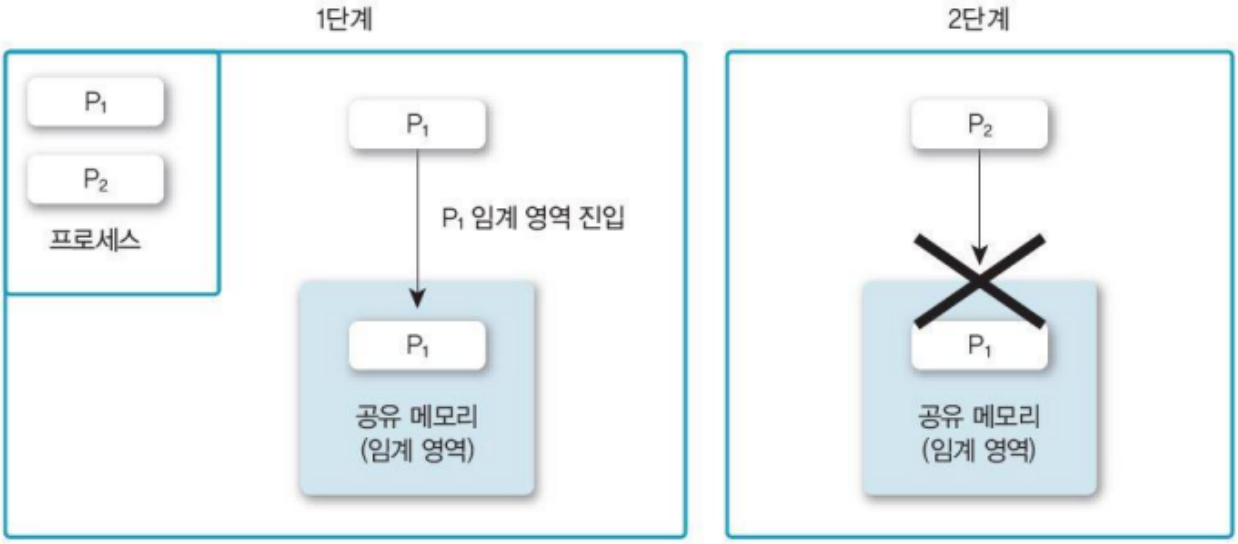
- Atomicity (원자성), Indivisible (분리불가능)
- **한 기계어 명령의 실행 도중에 인터럽트 받지 않음**



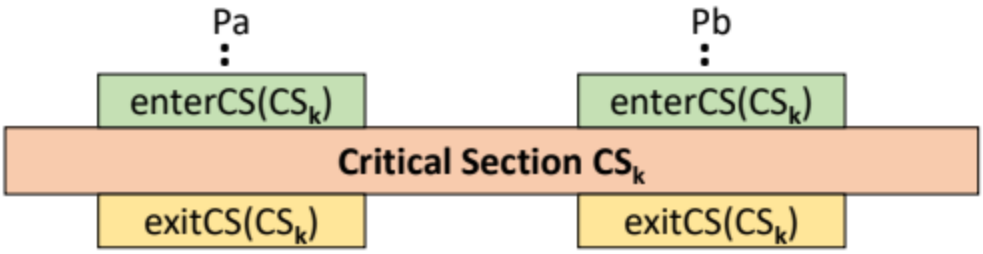
- **명령 수행 과정 (1)**
  - ① → ② → ③ → Ⓐ → Ⓑ → Ⓒ 또는 Ⓐ → Ⓑ → Ⓒ → ① → ② → ③
  - 결과 sdata = 2
- **명령 수행 과정 (2)**
  - ① → ② → Ⓐ → Ⓑ → Ⓒ → ③
  - 결과 sdata = 1

Race condition

Mutual Exclusion (상호배제)



- Mutual exclusion primitives - ME을 구현하기 위한 기본 연산
  - **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



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

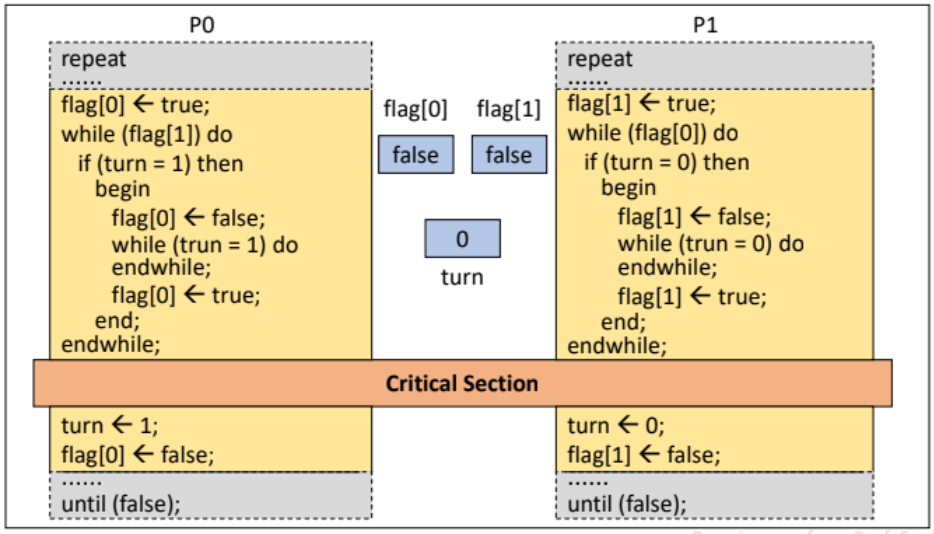
- Bounded waiting (한정대기) : 프로세스의 CS 진입은 유한시간 내에 허용되어야 함

Mutual Exclusion Solutions

- SW solutions
  - 2개일 때
    - Dekker’s algorithm (Peterson’s algorithm)
  - n개일 때
    - 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

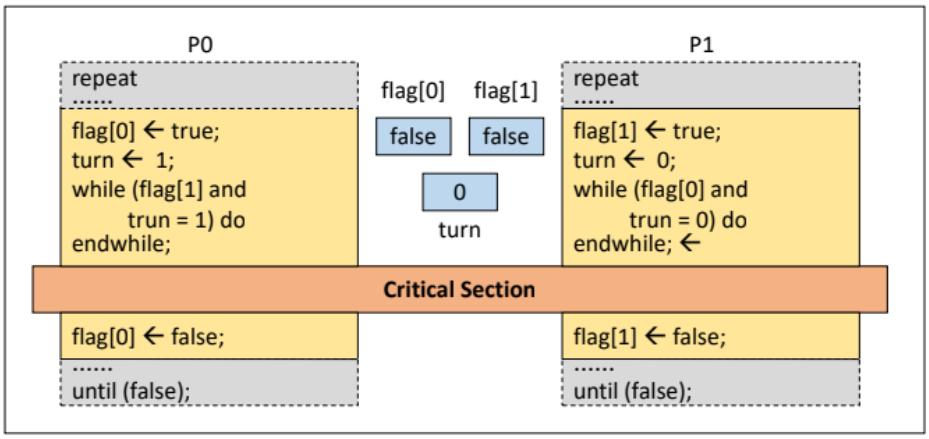
Dekker’s algorithm

• **Two process ME을 보장하는 최초의 알고리즘**



Peterson's algorithm

• **Dekker's algorithm 보다 간단하게 구현**

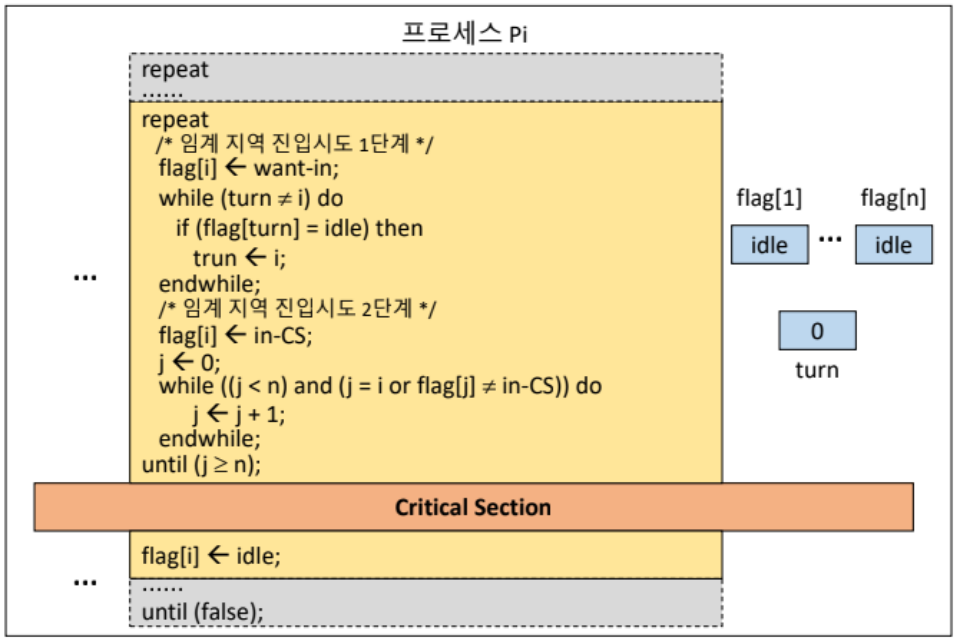


Dijkstra's algorithm

- 최초로 프로세스 n개의 상호배제 문제를 소프트웨어적으로 해결
- 실행 시간이 가장 짧은 프로세스에 프로세서 할당하는 세마포 방법, 가장 짧은 평균 대기시간 제공

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



SW solution 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 : 기다리면서도 뭔가를 수행하는 상태

TestAndSet (TAS) instruction

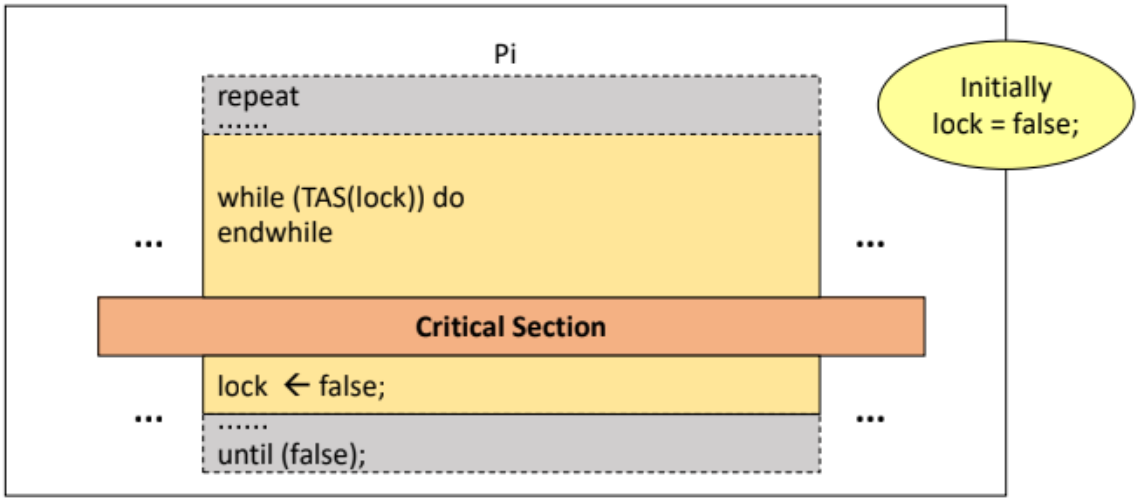
- Test 와 Set을 한번에 수행하는 기계어
- Machine instruction
  - Atomicity, Indivisible
  - **실행 중 interrupt를 받지 않음 (preemption 되지 않음)**
- Busy waiting
- 명령어

예제 4-6 TestAndSet 명령어

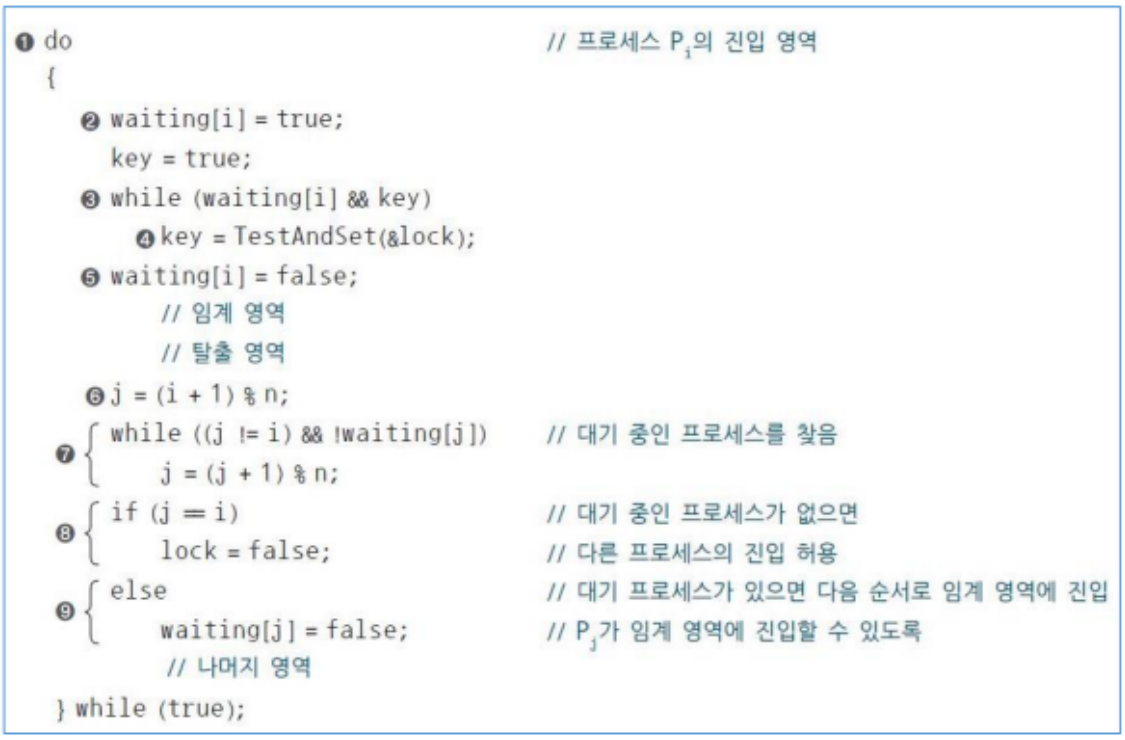
```
// target을 검사하고, target 값을 true로 설정
boolean TestAndSet (boolean *target) {
    boolean temp = *target;     // 이전 값 기록
    *target = true;             // true로 설정
    return temp;                // 값 반환
}
```

한번에 수행
(Machine instruction)

- ME with TAS Instruction



  - 3개 이상의 프로세스의 경우, Bounded waiting 조건 위배
    ⇒ 1,2번만 수행되고 3번이 수행되지 못하는 경우가 발생할 수 있음
  - 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);
```

- 구현이 간단하지만 Busy waiting 문제를 해결하지 못함

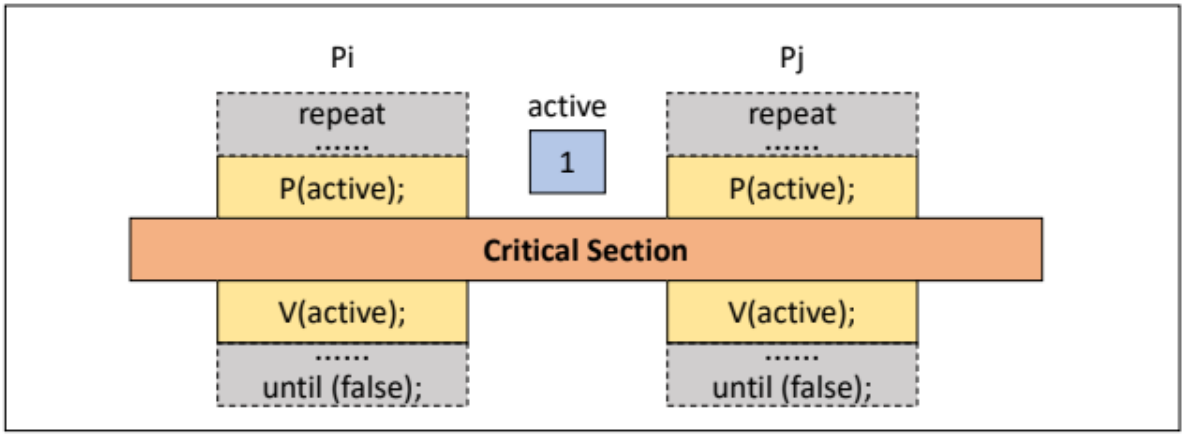
Spinlock

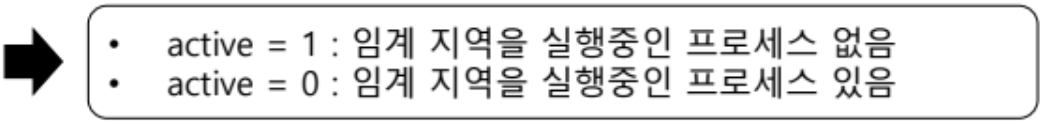
- 정수 변수 (S)
  - 초기화, P(), V() 연산으로만 접근 가능
    - 위 연산들은 indivisible (or atomic) 연산 (OS support)
    - 전체가 한 instruction cycle에 수행 됨

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

```
V(S) {
  S ← S + 1;
}
```

- S : 물건의 개수
  P : 물건 꺼내감
  V : 물건 반납





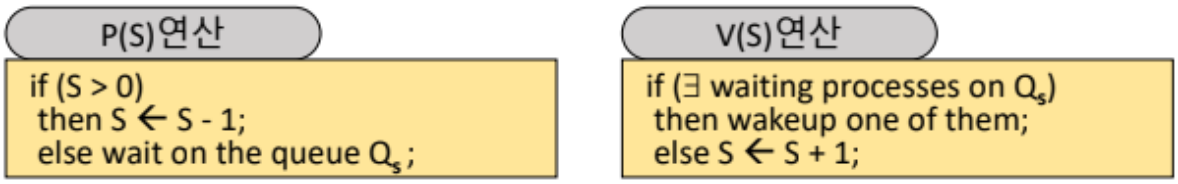
물건을 꺼내가면서 1이 0으로 바뀌면 다른 건 물건을 꺼내갈 수 없으므로 기다리게 됨

- 멀티 프로세서 시스템에서만 사용 가능 ⇒ P_i와 P_j가 동시에 돌아야 함
- Busy waiting

Semaphore

- Busy waiting 문제 해결
- 음이 아닌 정수형 변수(S) - 초기화 연산, P(), V()로만 접근 가능
- **임의의 S 변수 하나에 ready queue 하나가 할당 됨**
- 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

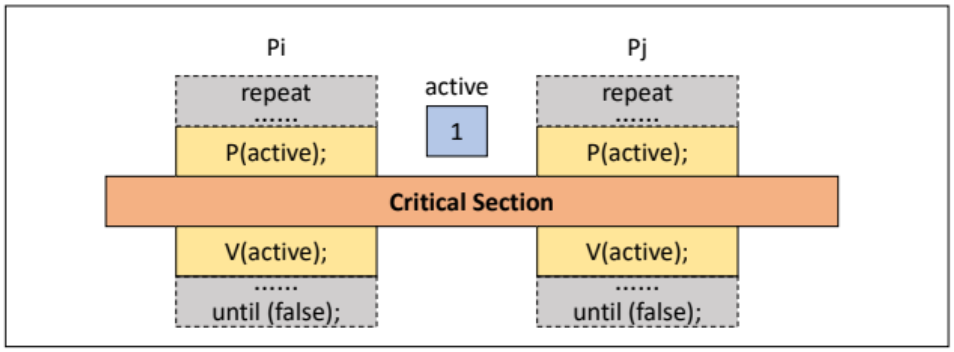
- P() 연산, V() 연산

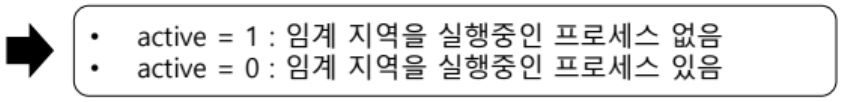


  - P : 실행할 수 없을 경우 S가 ready queue에서 대기
  - V : ready queue에 대기 중인 프로세스가 있다면 깨우고 나감

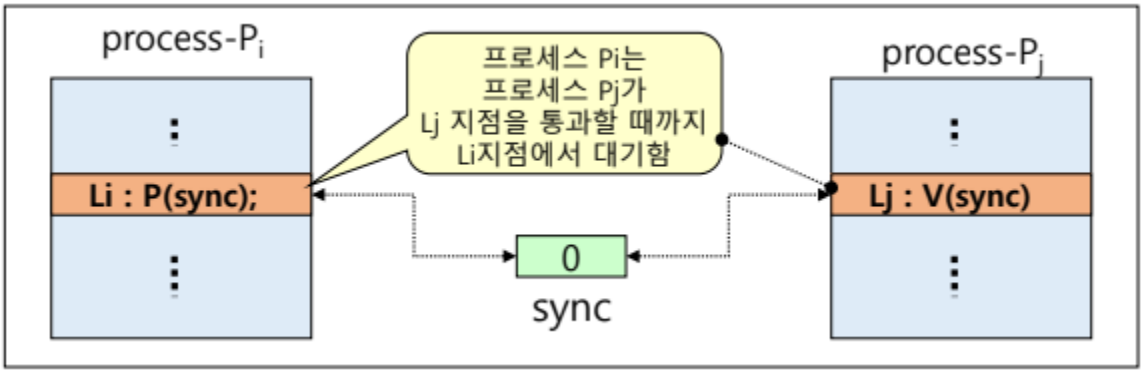
Semaphore로 해결 가능한 동기화 문제들

- 상호배제 문제

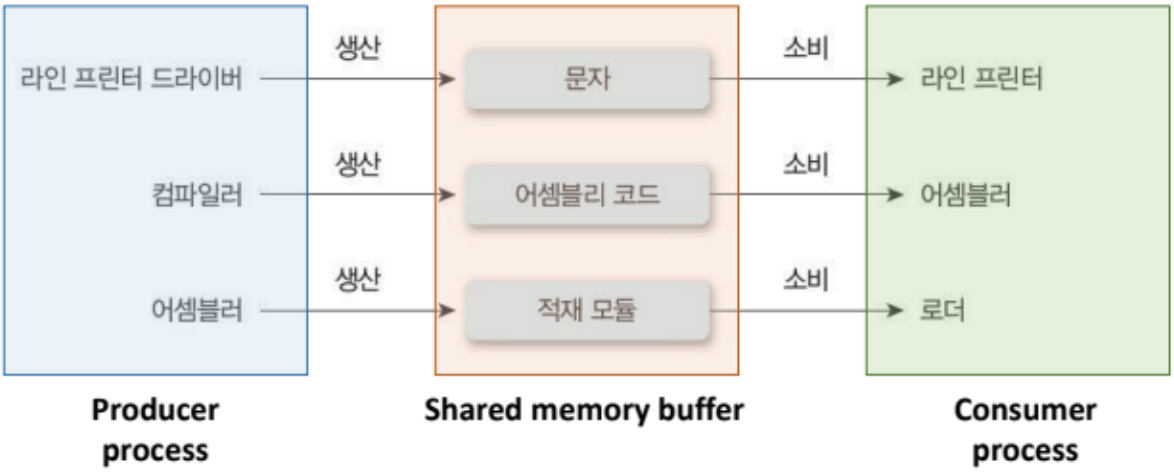




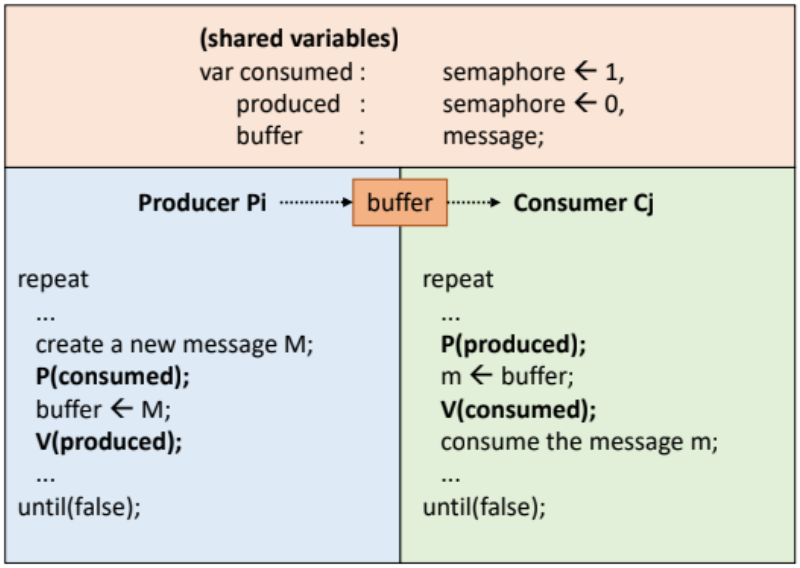
  - spinlock과 비슷하지만 Busy waiting 문제가 해결됨
- 프로세스 동기화 문제
  - Process들의 실행 순서 맞추기
  - 프로세스들은 병행적이며, 비동기적으로 수행



- 생산자-소비자 문제
  - 생산자(Producer) 프로세스 : 메시지를 생성하는 프로세스 그룹
  - 소비자(Consumer) 프로세스 : 메시지 전달받는 프로세스 그룹

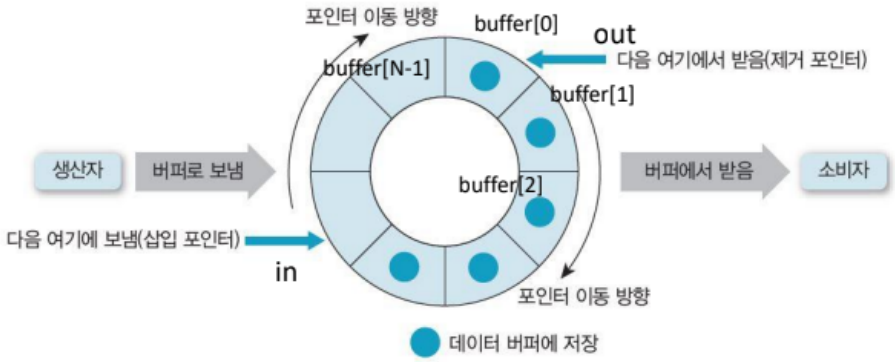


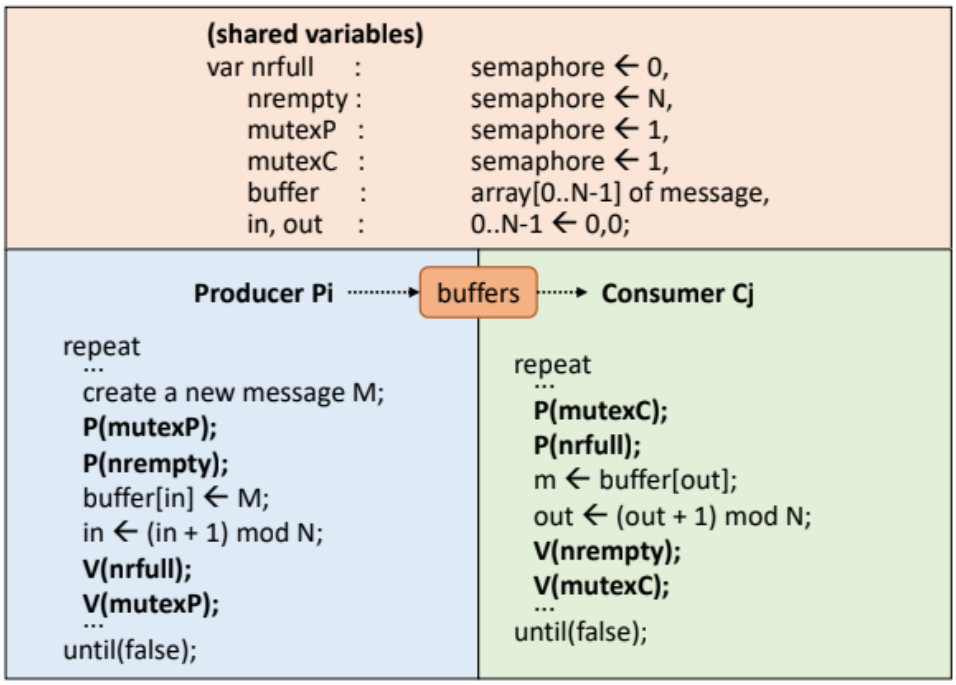
• Producer-Consumer problem with single buffer



- 메시지를 놓는 중에는 꺼내가면 안되고, 꺼내가는 중에는 놓으면 안된다. → 한 번에 한 명만 접근
- 변수 2개 설정 : consumed, produced
- 생산자 : 비었는지 확인 > consumed를 0으로 변경, produced를 1로 변경
- 소비자 : 생산되었는지 확인 > 생산 안됐다면 ready queue에서 기다림, 생산된 후에 produced를 0으로 변경, consumed를 1로 변경

• Producer-Consumer problem with N-buffers



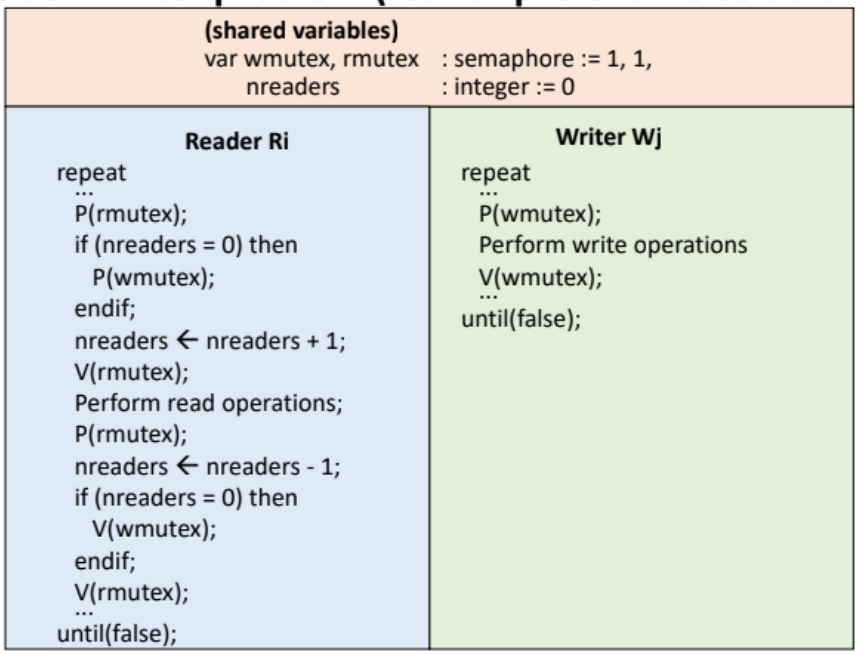


  - nrfull : 채워진 buffer 수
    nrempty : 빈 buffer 수
    mutexP : 생산자에 대한 mutual exclusion
    mutexP : 소비자에 대한 mutual exclusion
  - 생산자 : nrempty가 0보다 크면 생산, circular queue를 사용하므로 N으로 나눈 나머지를 활용해서 다음 위치 지정, 나오면서 nrfull 증가
  - 소비자 : nrfull이 0보다 크면 소비, 다음 위치 지정, 나오면서 nrempty 증가

- Reader-writer 문제
  - 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

- **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);
```

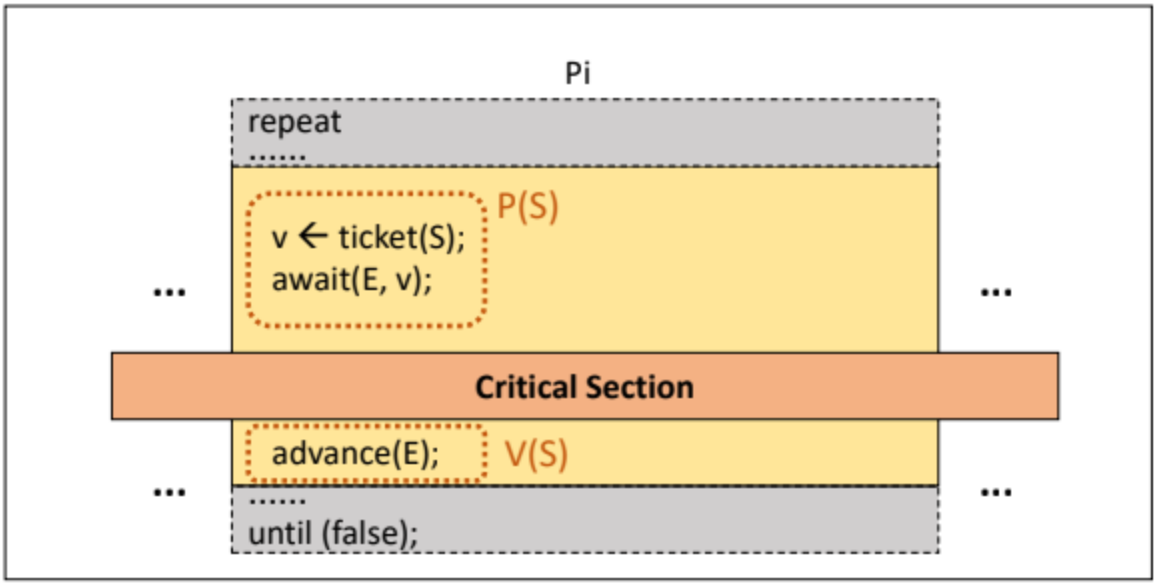
  - reader에 우선권 부여 ⇒ reader가 작업중이면 writer는 접근 불가
  - reader : nreader가 0보다 크면 writer는 작업 불가, nreaders 증가, 읽는 연산은 여러 명이 동시에 가능, 다 읽고 나갈 때 nreaders 감소, nreaders가 0이 된다면 writer가 접근 가능
  - writer : nreader가 0이어야만 접근 가능

Semaphore 특징

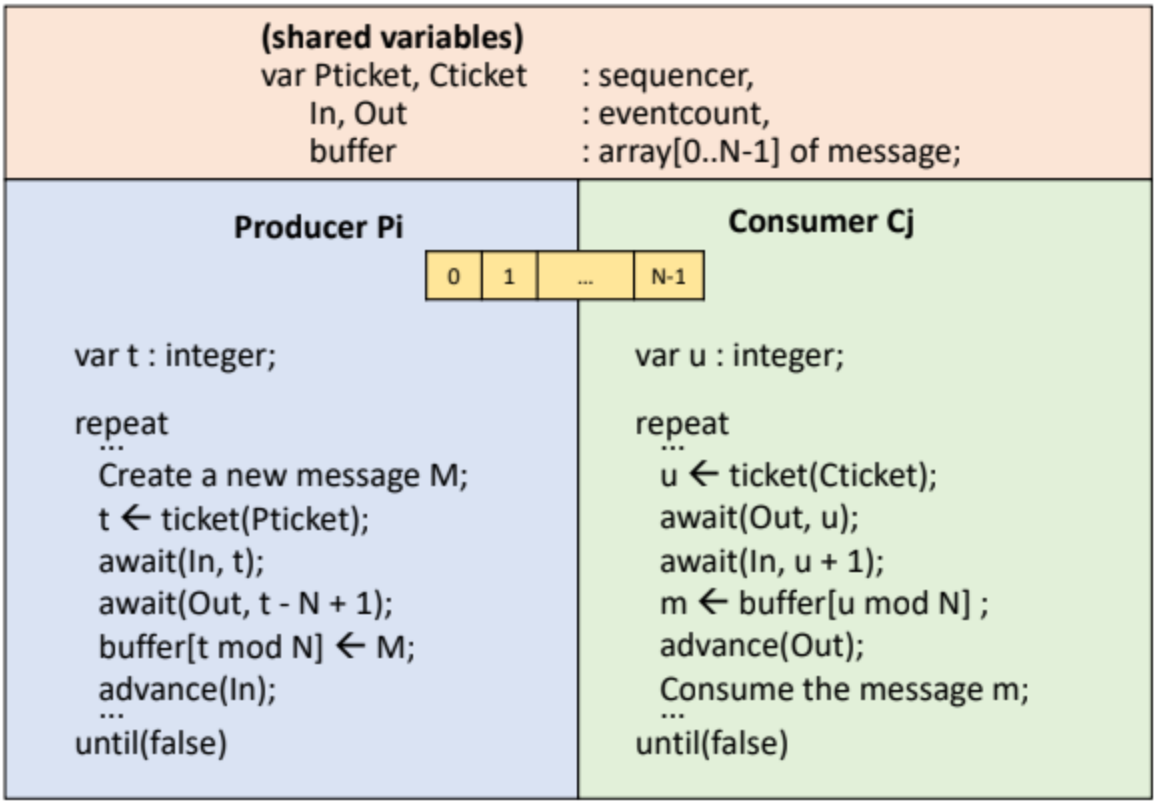
- No busy waiting
  - 기다려야 하는 프로세스는 block(asleep)상태가 됨
- Semaphore queue에 대한 wake-up 순서는 비결정적
  - Starvation problem

Eventcount/Sequencer

- 은행의 번호표와 비슷한 개념
- Sequencer ⇒ 번호표 기계
  - 정수형 변수, 생성 시 0으로 초기화, 감소하지 않음, 발생 사건들의 순서 유지
  - ticket() 연산으로만 접근 가능
- ticket(S)
  - 현재까지 ticket() 연산이 호출 된 횟수를 반환
  - Indivisible operation
- Eventcount ⇒ 번호판
  - 정수형 변수, 생성 시 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
- **Mutual exclusion**



- **Producer-Consumer problem**



  - Pticket : 생산자
    Cticket : 소비자

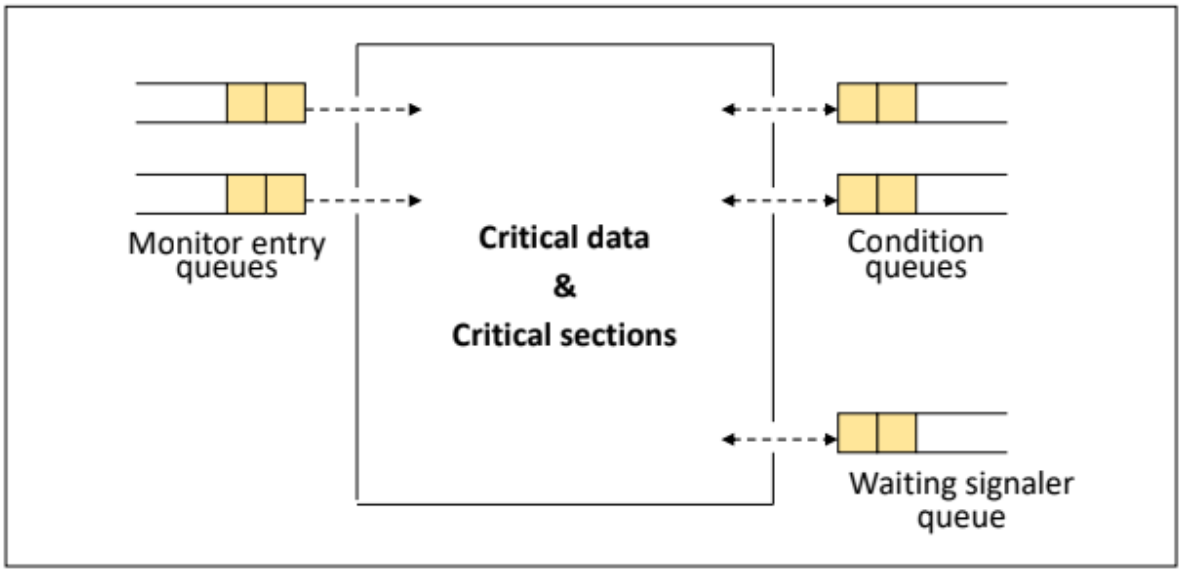
in : 물건 놓기
out : 물건 꺼내기
buffer : 크기 N인 circular buffer

  - 생산자 : await(Out, t - N + 1); ⇒ 공간 있는지 체크 (공간 = N-t+out)
  - 소비자 : await(In, u + 1); ⇒ 물건 수가 0보다 큰 지 체크

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

Monitor

- Language-level constructs
- Object-Oriented concept과 유사
- 사용이 쉬움
- 공유 데이터와 Critical section의 집합
- Conditional variable
  - wait(), signal() operations

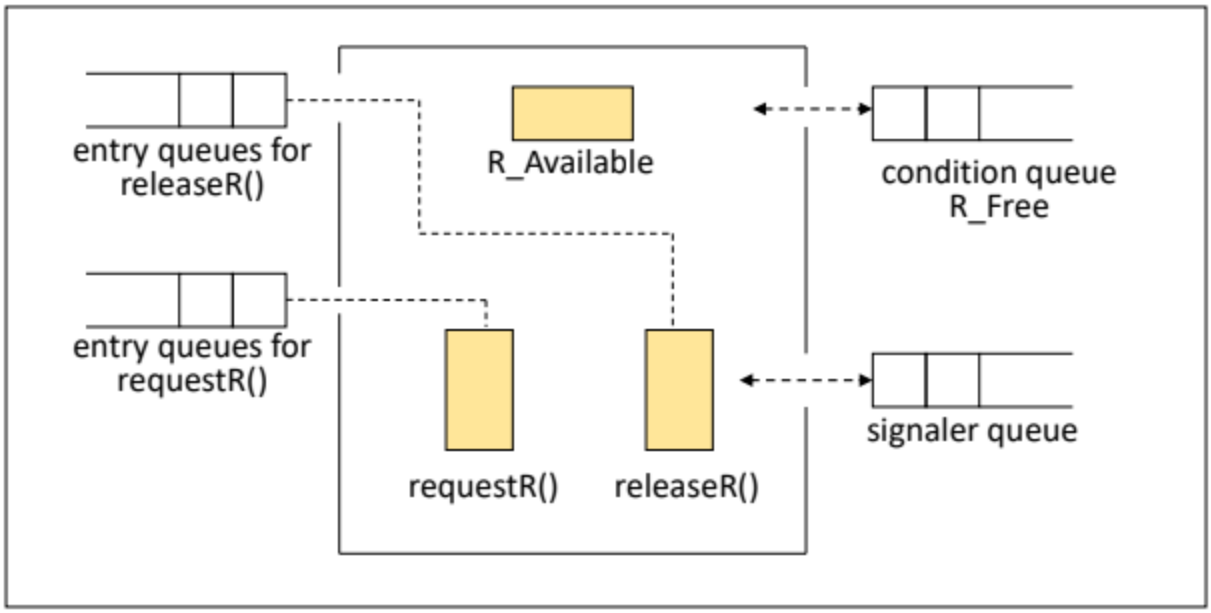


- 한 번에 한 명이 최대인 책방
- ciritical data : 빌리고 싶거나 읽고 싶은 책
- critical sections : 카운터

Monitor의 구조

- Entry queue (진입 큐)
  - 모니터 내의 procedure 수만큼 존재
- Mutual exclusion
  - 모니터 내에는 항상 하나의 프로세스만 진입 가능 ⇒ language가 보장
- Information hiding (정보 은폐)
  - 공유 데이터는 모니터 내의 프로세스만 접근 가능
- Condition queue (조건 큐)
  - 모니터 내의 특정 이벤트를 기다리는 프로세스가 대기하는 대기실
- Signaler queue (신호제공자 큐)
  - 모니터에 항상 하나의 신호제공자 큐가 존재
  - signal() 명령을 실행한 프로세스가 임시 대기
  - 시그널을 보내기 위해 잠시 대기하는 공간

자원 할당 문제



- condition queue : 책 빌릴 수 있는지 체크
- signaler queue : condition queue에 있는 걸 깨우기 위해 잠깐 들어감

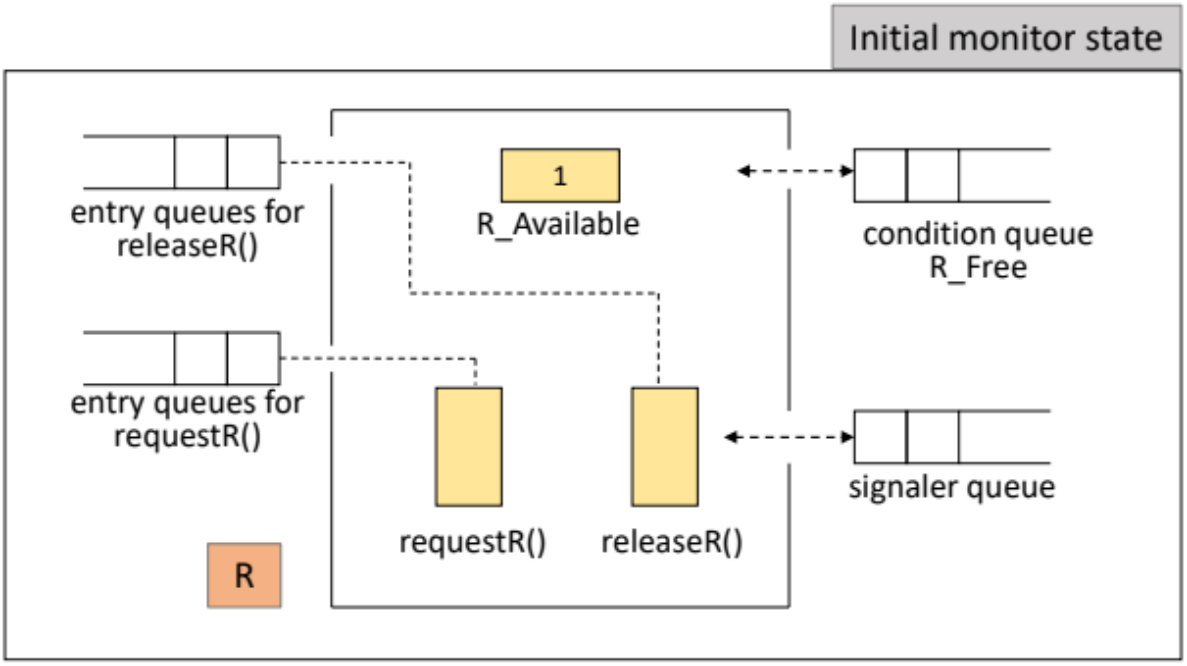
```
monitor resourceRiAllocator;
var RiIsAvailable : boolean,
    RiIsFree      : condition;
procedure requestR();
begin
  if (¬R_Available) then
     R_Free.wait();
  R_Available ← false;
end;
procedure releaseR():
begin
  R_Available ← true;
  R_Free.signal();
end;

begin
  RiIsAvailable ← true;
end.
```

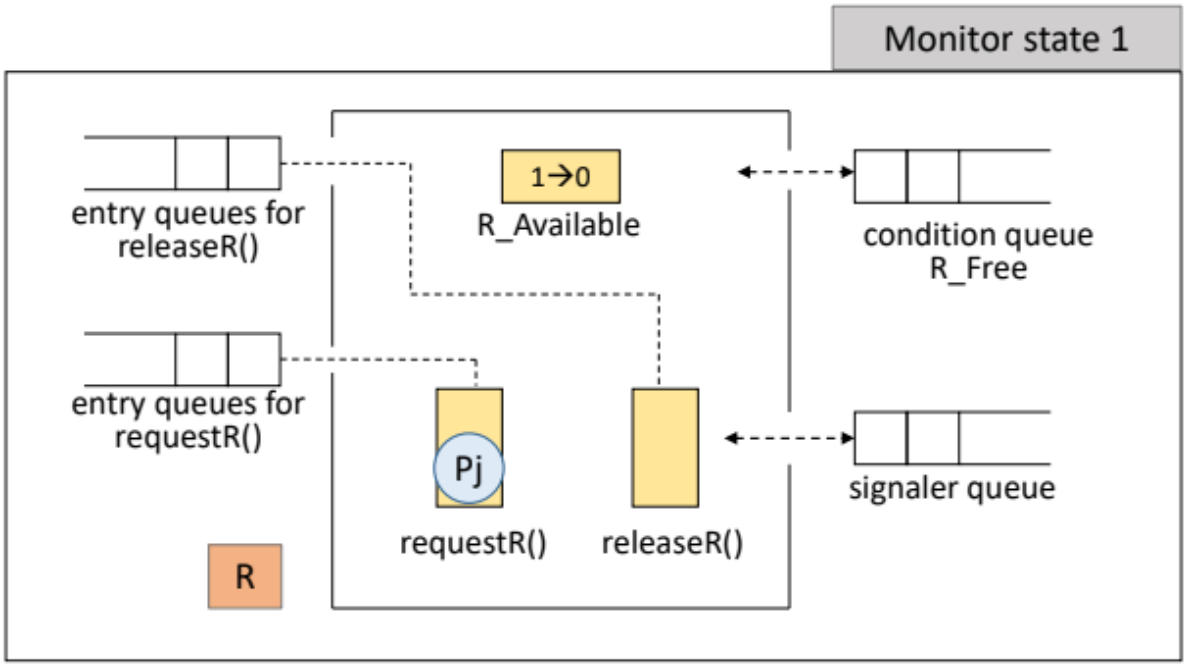
- 대출 : 책이 있는지 확인, 없으면 queue에서 기다림, 있으면 빌려가고 R_Available을 false로 변경
- 반납 : 책 돌려 놓고 queue에 시그널 보냄

자원 할당 시나리오

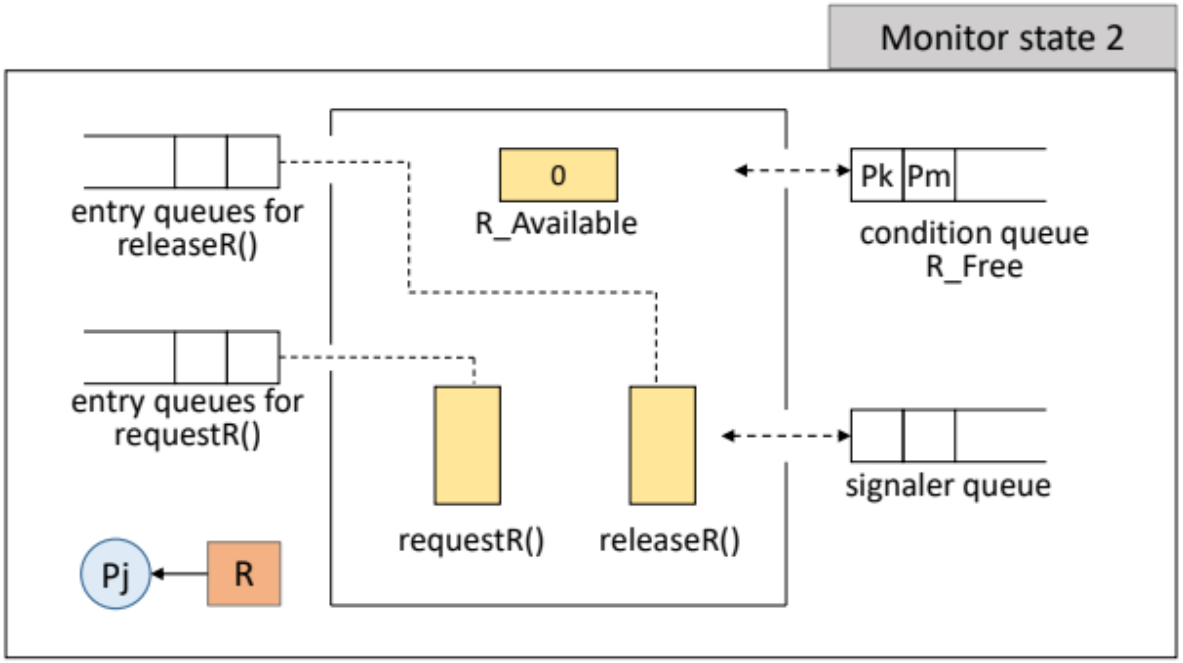
- 자원 R 사용 가능
- Monitor 안에 프로세스 없음



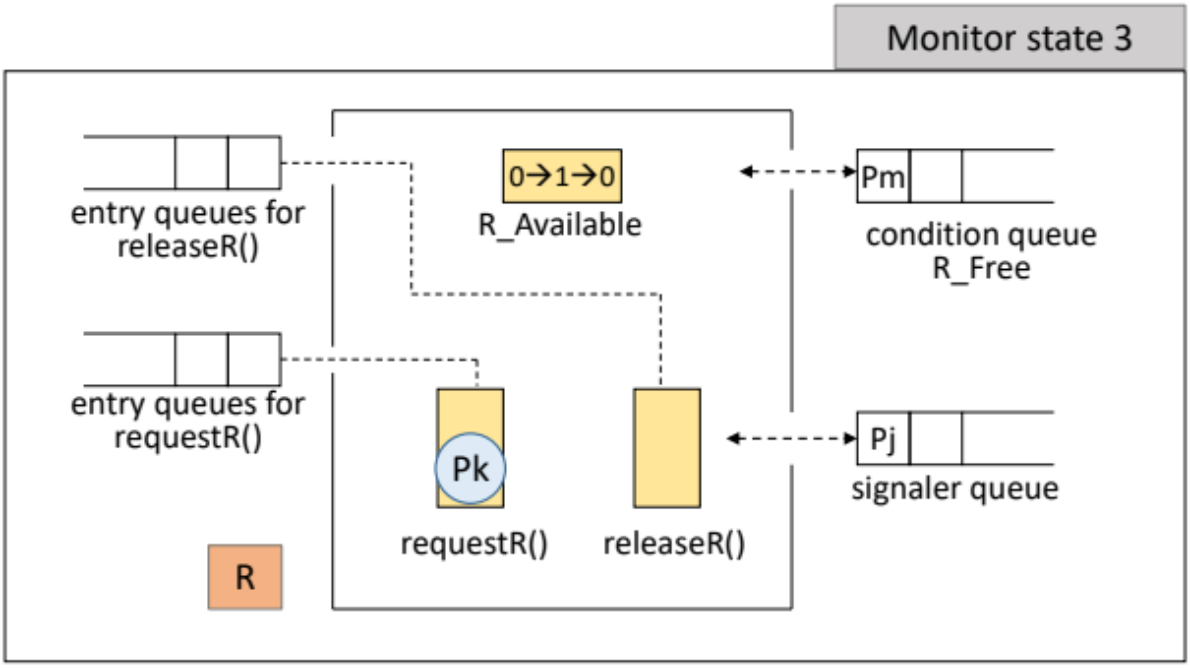
- 프로세스 Pj가 모니터 안에서 자원 R을 요청



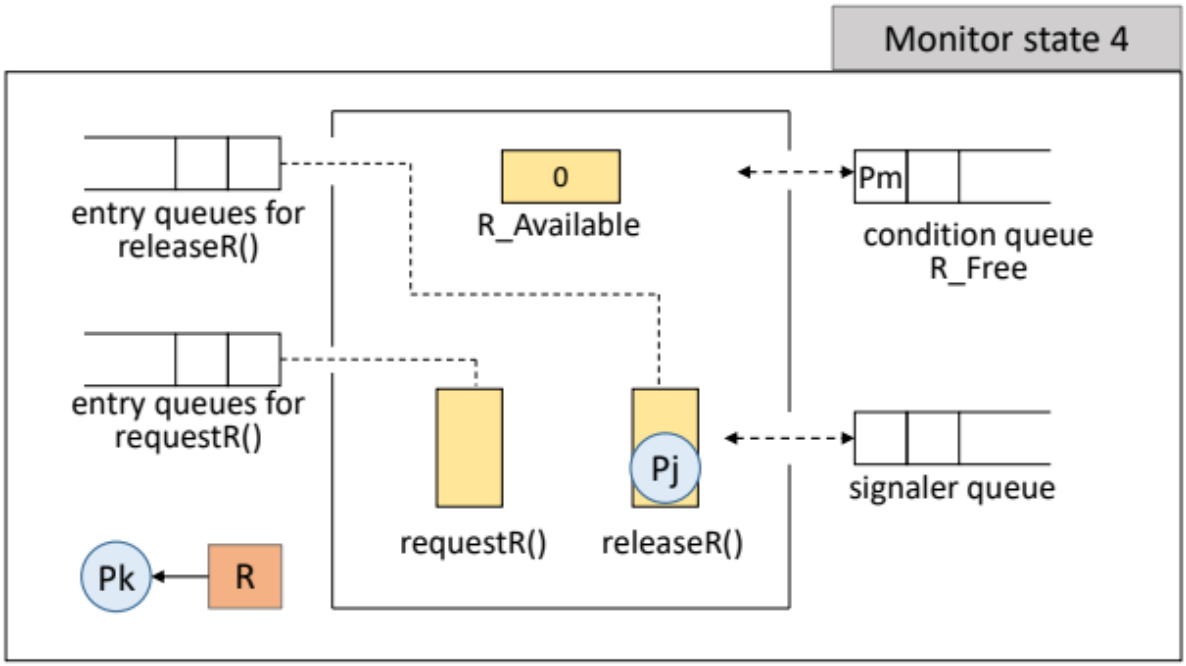
- 자원 R이 Pj에게 할당 됨
- 프로세스 Pk 가 R 요청, Pm 또한 R요청



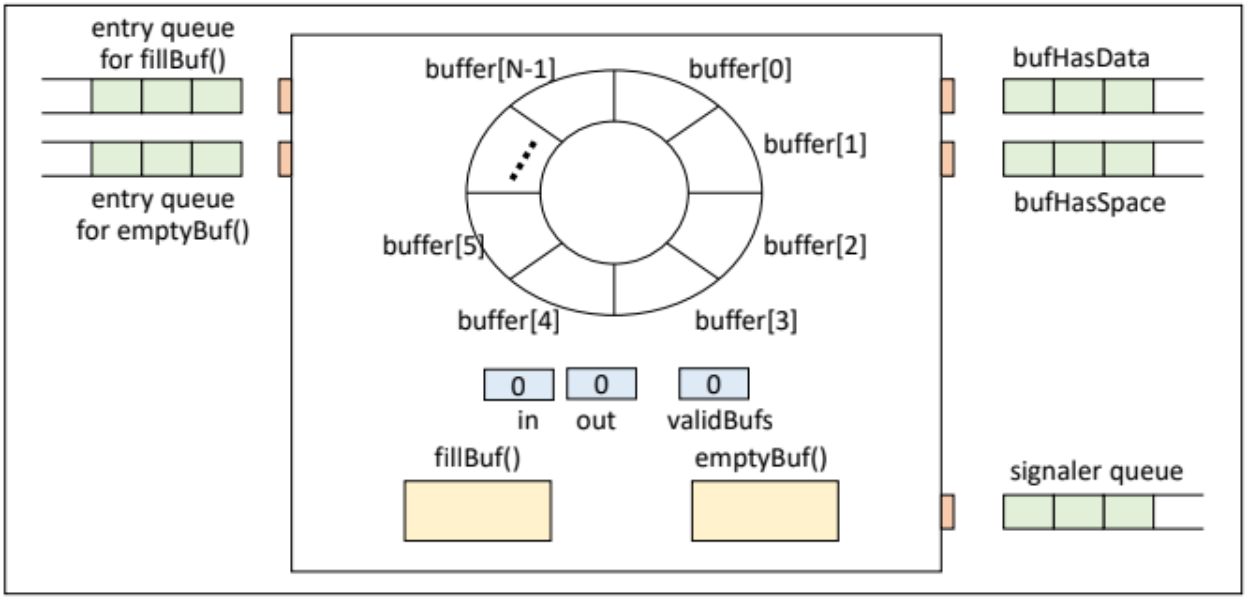
- Pj가 R 반환
- R_Free.signal() 호출에 의해 Pk가 wakeup

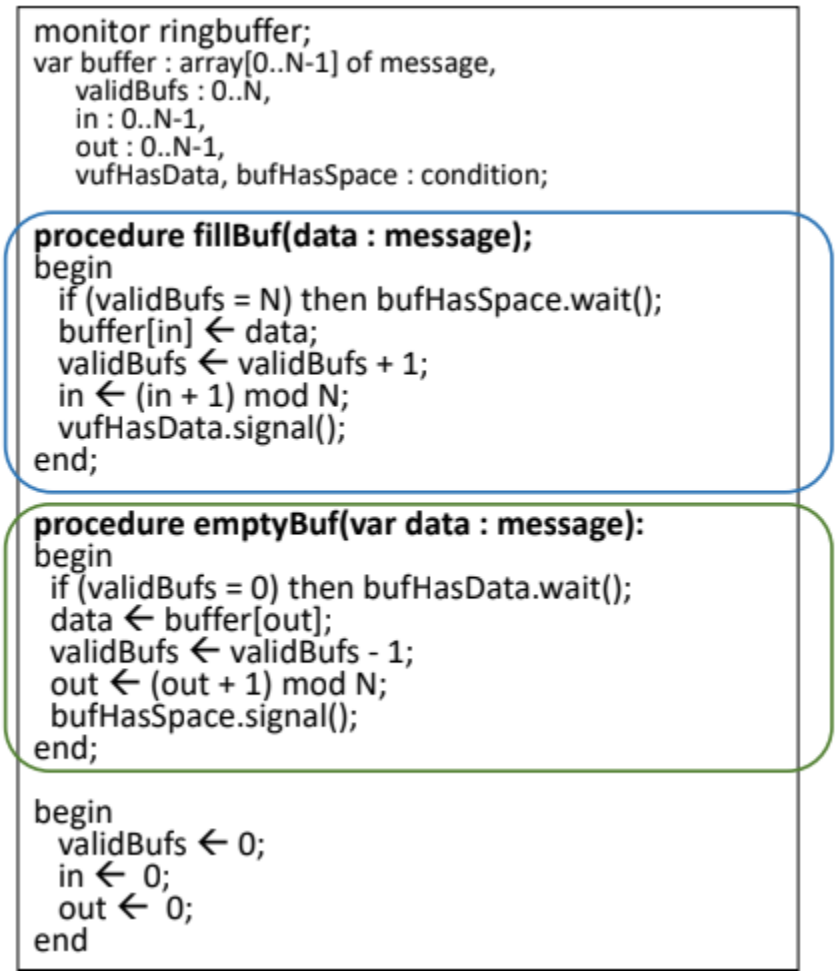


- 자원 R이 Pk에게 할당 됨
- Pj가 모니터 안으로 돌아와서, 남은 작업 수행



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
```

- 생산자 : 공간이 있는지 확인, 물건 수가 N이면 대기, 공간이 있다면 물건 놓고 validBufs 증가 시키고 다음 위치 지정, 시그널 날림
- 소비자 : 물건이 있는지 확인, 물건이 0개면 대기, 물건 꺼내고 validBufs 감소 시키고 다음 위치 지정, 시그널 날림

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

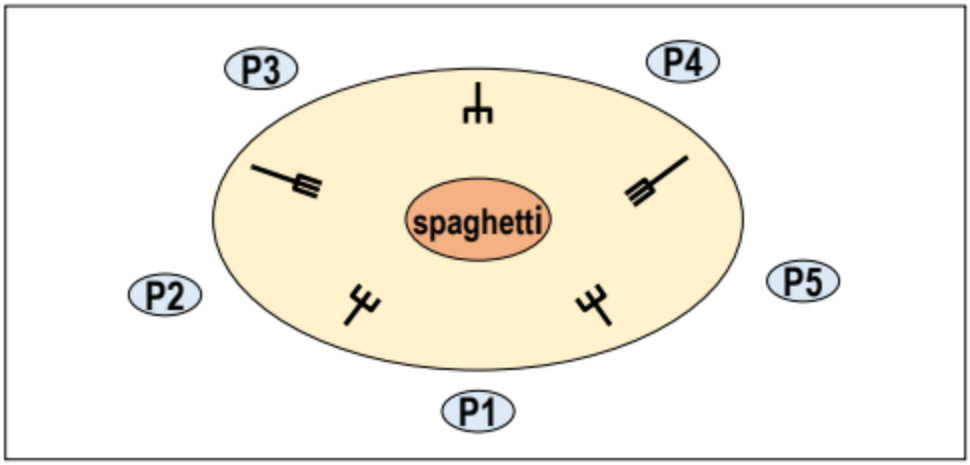
- 조건 큐 2개
  - reader/writer프로세스가 대기해야 할 경우에 사용
- 프로시져 4개
  - reader(writer) 프로세스가 읽기(쓰기) 작업을 원할 경우에 호출, 읽기(쓰기) 작업을 마쳤을 때 호출

```
monitor 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.
```

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
- p : 프로세스, 포크 : 공유 데이터

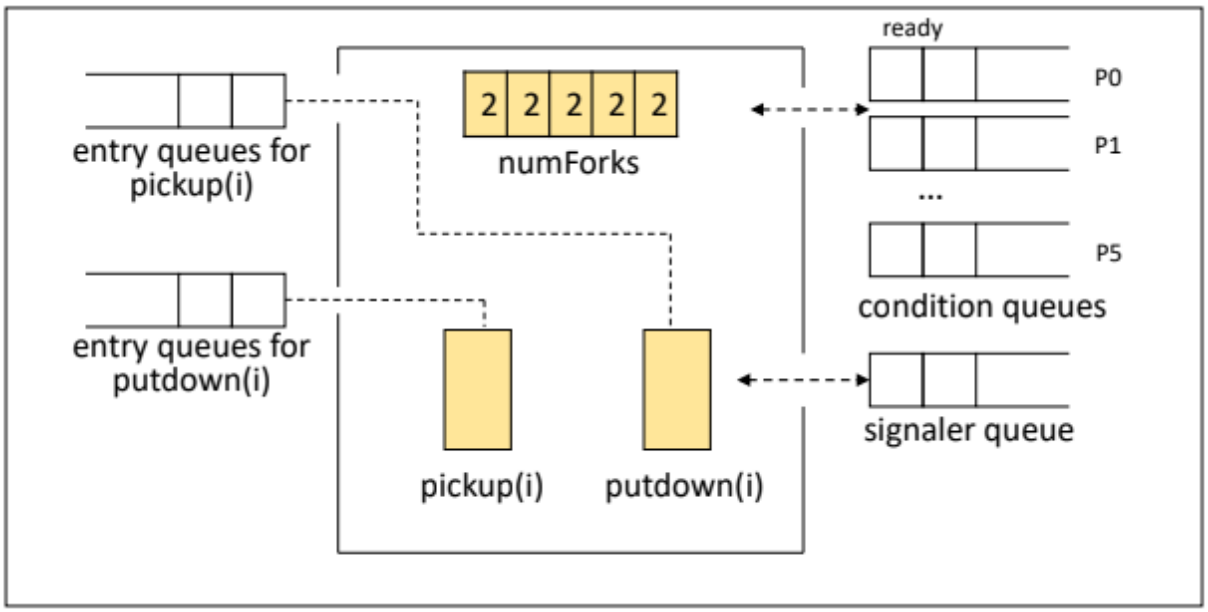


Pi

```
do forever
  pickup(i);
  eating;
  putdown(i);
  thinking;
end.
```

```
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.
```



- pickup : 내가 2개의 포크를 쓸 수 있는지 체크, 없으면 대기, 있으면 포크 집기 ⇒ 내 왼쪽과 오른쪽 철학자의 포크 수 1 감소
- putdown : 포크를 내려 놓음 ⇒ 내 왼쪽과 오른쪽 철학자의 포크 수 1 증가, 시그널 보냄

Monitor 특징

- 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