6장. 프로세스 동기화

목표

- **임계구역**(Critical Region) 문제 소개
  - 이 문제에 대한 해결책은 공유 데이터의 일관성 유지에 사용가능
- 임계구역 문제의 **하드웨어 및 소프트웨어 해결책** 제시
- 전통적인 **프로세스 동기화 문제** 소개
- 프로세스 동기화 문제 해결에 사용되는 **도구** 조사

6.1 배경

- 협력 프로세스(Cooperating process)
  - 다른 프로세스의 실행을 영향을 주거나 받는 프로세스
  - 서로 비동기적으로 수행하면서 데이터를 공유할 수 있음
  - 공유 방법: (1) 직접 공유 - 논리 주소 공간(메모리) 공유
    (2) 간접 공유 – 파일 또는 메시지를 경유
- 공유 데이터에 대한 병행/병렬 접근은 데이터 일관성을 보장하지 못할 수 있음(data inconsistency)
- 병행/병렬 실행 환경
  - **CPU 스케줄링** : 한 프로세스가 일부만 실행한 상태에서 다른 프로세스로 스케줄될 수 있음
  - **인터럽트** : 프로그램의 어떠한 지점에서도 인터럽트가 가능
  - **병렬 실행** : 여러 개의 프로세스가 동시에 실행됨
- 데이터 일관성 유지
  - 협력 프로세스들이 바른 순서로 실행(orderly execution)하는 것을 보장하는 메커니즘이 필요

생산자-소비자 문제 – 공유 데이터 사용

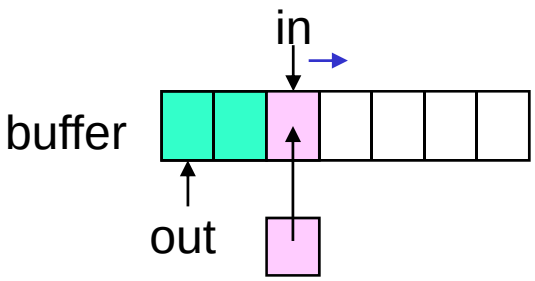
- 유한 버퍼를 사용한 생산자-소비자 문제

Producer

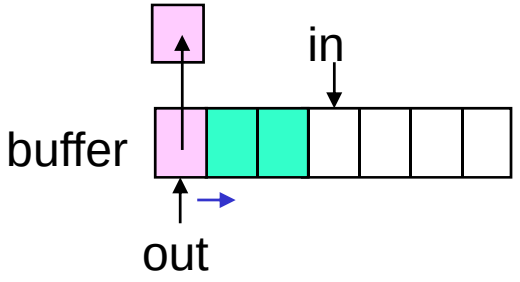
```
while (TRUE) {
    while (count == BUFFER_SIZE)
        ;  // do nothing during full
    // add an item to the buffer
    count = count + 1;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
}
```

Consumer

```
while (TRUE) {
    while (count == 0)
        ; // do nothing during empty
    // remove an item from the buffer
    count = count – 1;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
}
```



shared variable :
count



경쟁 조건(race condition)

- 경쟁 조건
  - 여러 개의 프로세스가 공유 자료를 접근하여 조작하고
  - 그 실행 결과가 자료 접근 순서에 영향을 받는 상황

    → non-deterministic 결과
- 예

Process A

count = count + 1

machine language

```
r1 = count
r1 = r1 + 1
count = r1
```

Process B

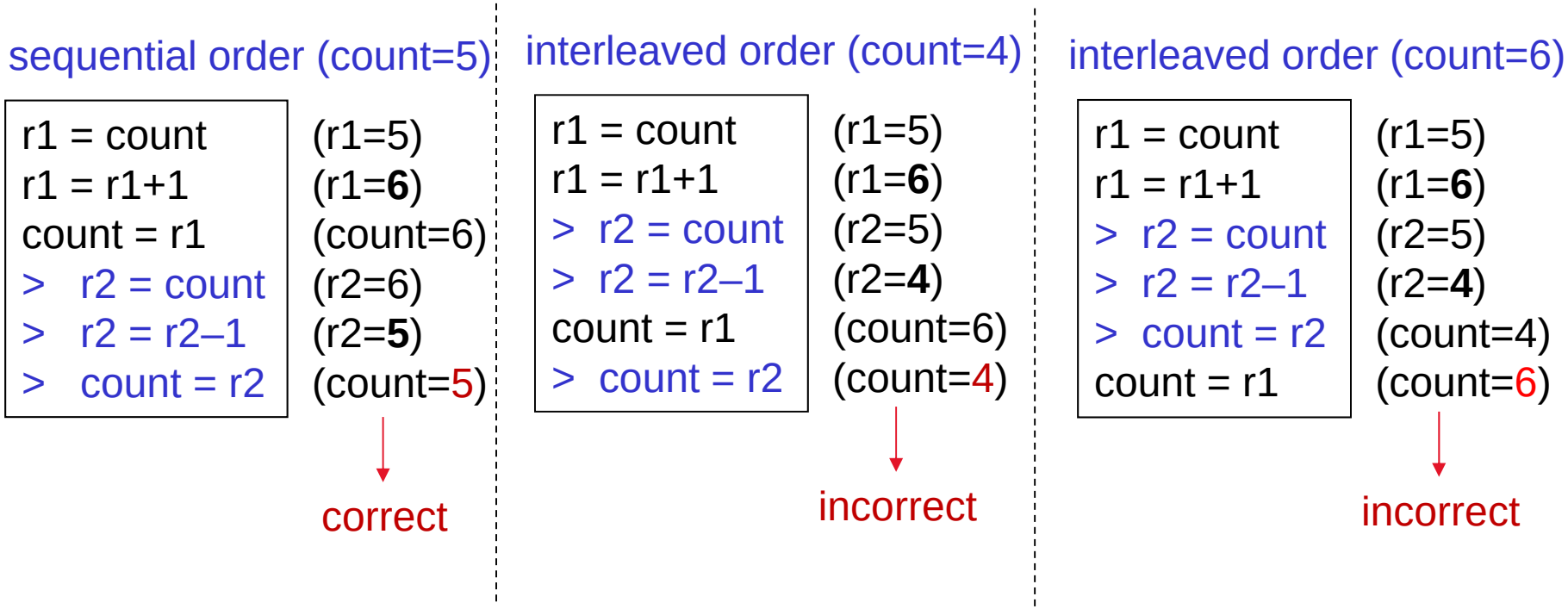
count = count – 1

```
r2 = count
r2 = r2 – 1
count = r2
```

machine language

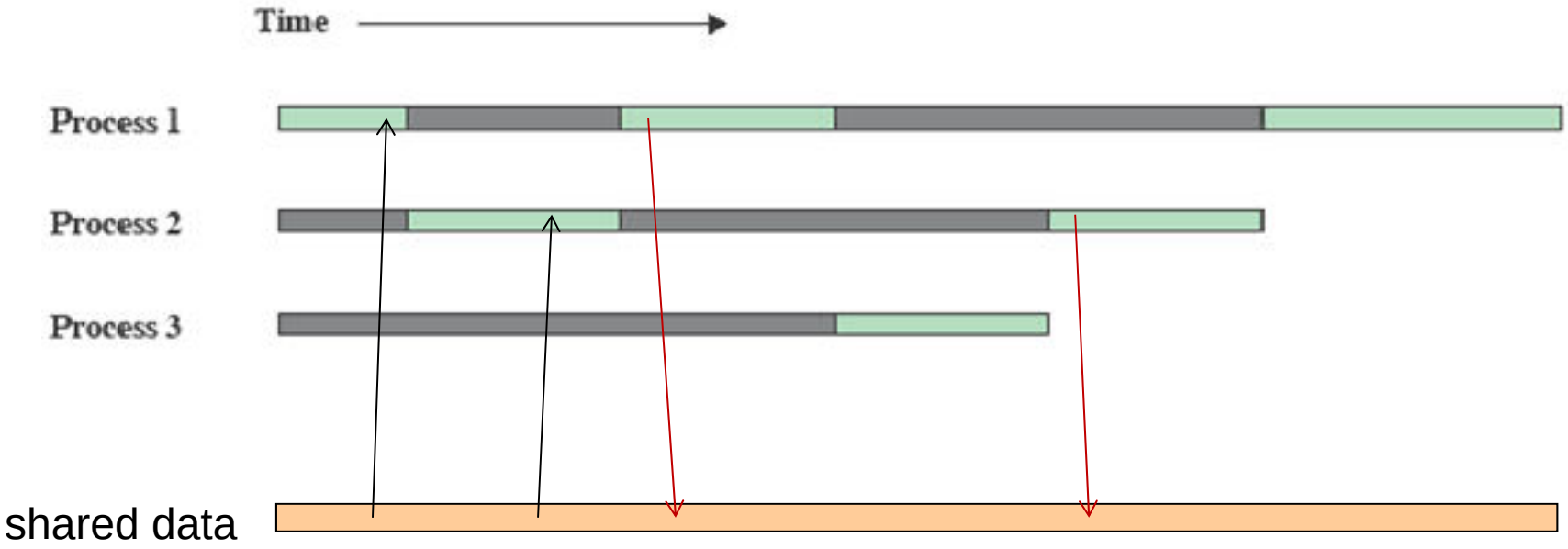
경쟁 조건에서의 가능한 결과

- 3가지 결과가 가능함



병행 접근(Concurrent Access)

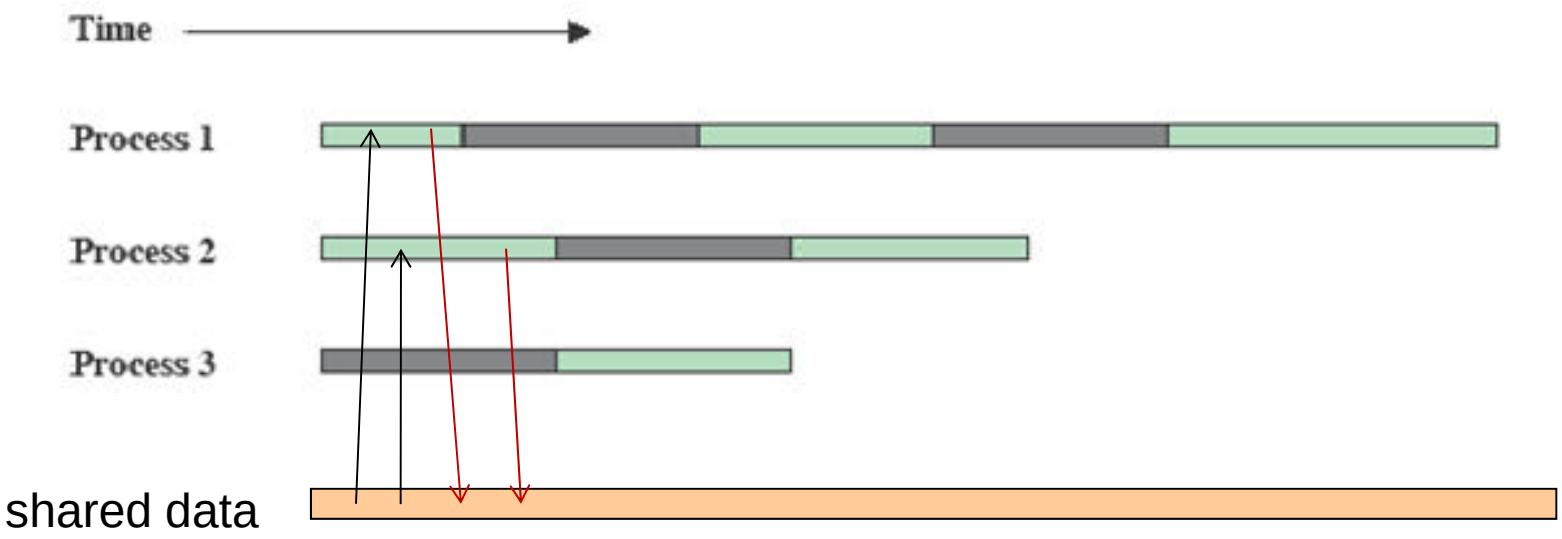
- 단일 프로세서 시스템에서



선점(preemptive) 스케줄링을 사용할 때에, 경쟁 조건이 발생할 수 있음

병렬(Parallel) 접근

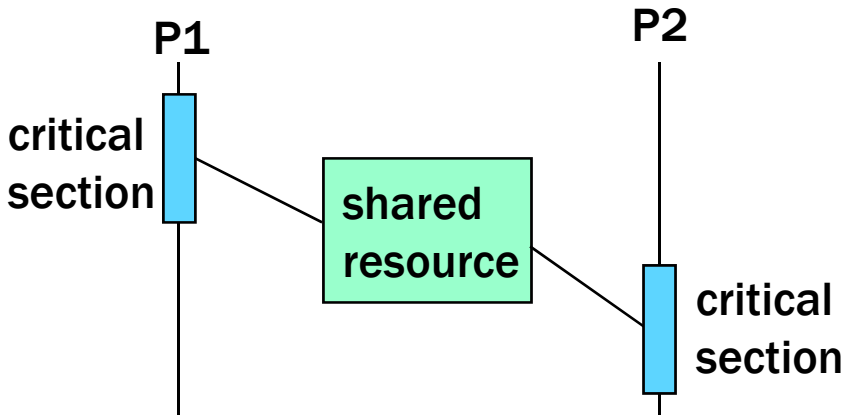
- 다중 프로세서 시스템에서



스케줄링 알고리즘에 관계없이, 경쟁 조건이 발생할 수 있음

6.2 임계 구역(Critical-Section) 문제

- 임계 구역(Critical section: **CS**)
  - 프로세스(쓰레드)가 공유 자원을 변경할 수 있는 코드 부분
    - 공유 자원 – 공유 변수, 테이블, 파일 등



- 임계 구역 문제
  - 프로세스들이 임계 구역에서 경쟁조건이 발생하지 않도록
    서로 협력하기 위해 사용할 수 있는 프로토콜(대화 규약)을 설계하는 것
    → 프로세스 **동기화**(synchronization)와 **조정**(coordination)

임계 구역 문제의 해결책

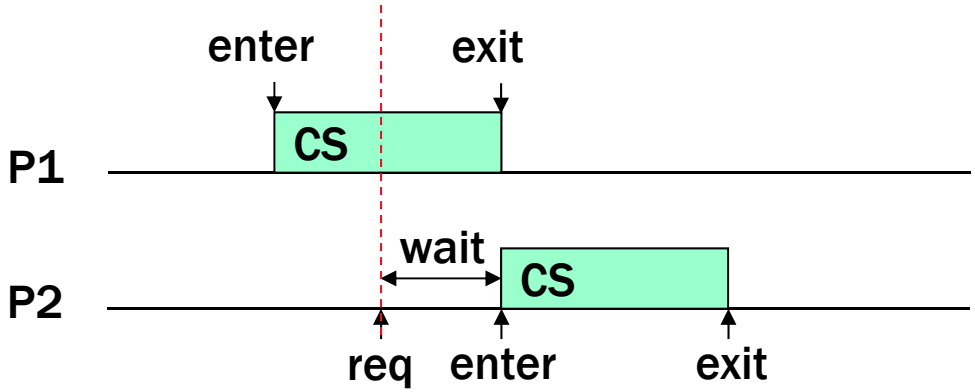
- 프로세스의 일반 구조

```
while (true) {
        ... remainder section
        entry section          // request permission to enter CS
        … critical section
        exit  section
        … remainder section
};
```

- 3가지 필요조건
  1. 상호배제(Mutual Exclusion)
  2. 진행(Progress)
  3. 한정 대기(Bounded Waiting)
- 가정
  - 프로세스들의 상대 속도를 가정하지 않음.
  - 프로세스들은 0이 아닌 속도로 실행됨

상호 배제(Mutual Exclusion)

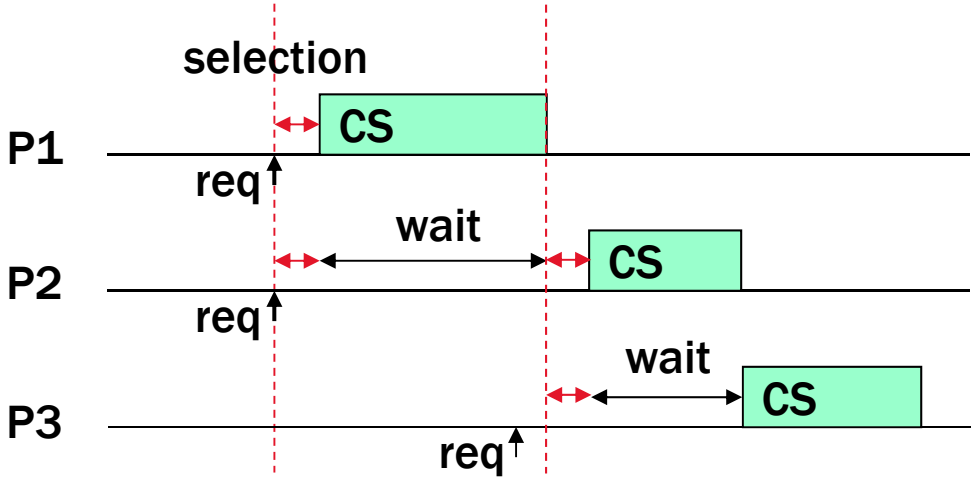
- 프로세스가 자신의 CS에서 실행 중이라면
  다른 프로세스들은 자신의 CS에서 실행될 수 없음



- 동시에 두 개 이상의 프로세스가 임계 구역(CS)에서 실행될 수 없음

진행(Progress)

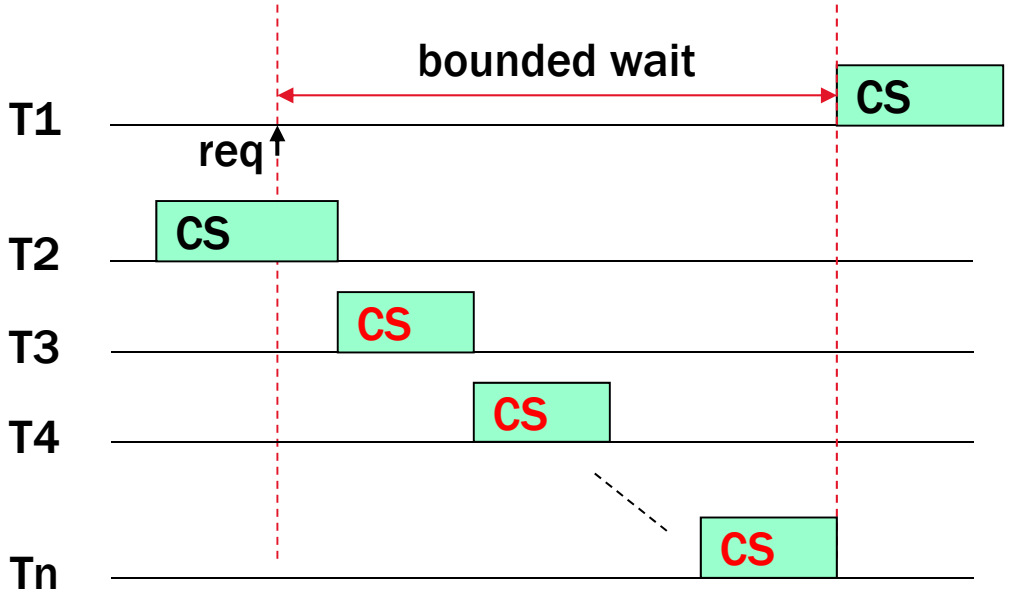
- CS에서 실행되는 프로세스가 없고, 자신의 CS로 진입하려는 프로세스가 있다면
  - 나머지 구역(remainder section)에서 실행하지 않는 프로세스들 만이 CS에 진입하는 프로세스 결정에 참여하고
  - 이 선택이 무한히(indefinitely) 지연될 수 없음 → progress



- CS 밖에서 수행중인 프로세스는 다른 프로세스를 block할 수 없음

한정 대기(Bounded Waiting)

- 프로세스가 CS 진입을 요청한 후에 요청이 허용될 때까지 다른 프로세스가 CS 진입이 허용되는 횟수에 **제한**이 있어야 함



- 어떤 프로세스도 CS 진입을 영원히 기다리지 않아야 함

커널에서의 경쟁조건과 선점/비선점 커널

- 커널에서의 경쟁 조건
  - 커널에서 여러 개의 커널 모드 프로세스(루틴)이 활성화될 수 있으며 커널 자료구조를 공유할 수 있음

    → 경쟁조건 발생 가능
- 커널은 경쟁조건이 발생하지 않도록 설계해야 함.
- 임계 구역을 다루기 위한 운영체제 커널의 두 가지 방식
  - **선점형 커널** – 커널모드에서 프로세스가 선점되는 것을 허용
    - 경쟁조건이 발생하지 않도록 설계해야 함
    - SMP 시스템에서는 특히 어려움
  - **비선점형 커널** – 커널모드에서 프로세스가 선점되는 것을 허용하지 않음
    - 경쟁조건이 발생하지 않도록 다음 상황까지 프로세스를 계속 실행

      (1) 커널모드 종료 (2) 봉쇄(block) (3) 자발적 양보(yield)

6.3 Peterson의 해결안

- Peterson 알고리즘
  - 임계 구역 문제에 대한 고전적인 소프트웨어 기반 해결방안
    - **두 프로세스에 대해서**만 적용 가능 (P_0 , P_1)
- 동기화 변수
  - 프로세스들의 동작을 동기화하는 데 사용하는 공유 변수
- Peterson 알고리즘에서의 동기화 변수

```
int  turn;              (initially turn=0)
boolean  flag[2];       (initially, flag[0]=flag[1]=false)
```

  - if (turn==i) then $\mathbf{P_i}$ can enter its critical section (Pi 차례)
  - if (flag[i] == true) then $\mathbf{P_i}$ ready to enter its critical section (Pi 준비)

if flag[0] is true and flag[1] is false → P_0's turn (하나만 CS 진입 요청)
if both flag[0] and flag[1] are true → P_{turn}'s turn (둘 다 CS 진입 요청)

Peterson 알고리즘

- Process P_i 의 구조 (다른 프로세스: P_j , j = 1 – i)

```
do {
    flag[i]:= true;
    turn = j;
    while (flag[j] and turn == j)
          ;
    … critical section
    flag [i] = false;
    … remainder section
} while (1);
```

entry CS → turn = j;

while (flag[j] and turn == j) ; ⟷ if (flag[j]==false or turn==i) then enter CS

exit CS → flag [i] = false;

- 세 필요조건을 충족
  - 상호 배제 준수 – 둘 다 진입 불가 (turn에 의해서 하나만 진입 가능)
  - 진행 – 상대장의 flag가 false이면 바로 CS진입, 두 flag 모두 true이면 turn이 0 또는 1의 값을 가지므로 둘 중 하나는 CS로 진입
  - 한정대기 – 상대방의 수행이 끝나면 flag가 false가 되므로 CS진입. 상대방 flag가 다시 true가 되어도, turn을 넘겨주므로 CS 진입

임계 구역을 구현하는 소프트웨어 알고리즘

- 2 프로세스 알고리즘
  - Peterson 알고리즘
  - Dekker 알고리즘 (Exercise 6.2)
- *n*-프로세스 알고리즘 → 훨씬 복잡
  - Eisenberg and McGuire 알고리즘 (Exercise 6.3)
    - Dekker 알고리즘의 확장
  - Bakery 알고리즘 – Lamport

- 알고리즘 방식은 실제로 널리 사용되지 않음
  - 매우 복잡함
  - 정확성(올바르게 동작함)에 대한 증명이 복잡함

6.4 동기화 하드웨어(Synchronization Hardware)

- **lock**을 사용하는 임계구역 문제 해결책

```
while (true) {
    acquire lock
    ... critical section
    release lock
    ... remainder section
}
```

  - 임계구역을 lock으로 보호하여, 경쟁조건을 방지함
- Lock의 구현
  - 소프트웨어 알고리즘 – 복잡하고 비효율적
  - 하드웨어 지원
    - 인터럽트 금지(disable)
    - Atomic 명령어

인터럽트 금지(Interrupt Disable)

- 인터럽트 금지(disable)
  - 임계구역 실행 동안 인터럽트를 방지하여 **선점을 허용하지 않음**
    - 현재 수행중인 코드가 선점되지 않고 계속 수행됨
- 단일 프로세서 시스템에서는 임계구역 문제 **해결 가능**

```
cli                 ; interrupt disable
  ... critical section
sti                 ; interrupt enable
```

(80x86)

- 멀티 프로세서 시스템에서는 **적용 불가능**
  - 한 CPU의 인터럽트를 금지시켜도, 다른 CPU는 여전히 임계구역에 진입할 수 있음.
  - 모든 CPU의 임계구역 진입을 방지하기 위해서, 모든 CPU의 인터럽트를 금지시키는 것은 시간이 소요되므로 비효율적임.
- 시스템 클록(시계)에 대한 영향
  - 시스템 클록이 인터럽트에 의해서 갱신되는 시스템에서, 인터럽트 금지는 시간에 영향을 줄 수 있음

원자적 명령어(Atomic Instruction)

- Atomic 명령어 = Indivisible Instruction
  - 연속되는 여러 개 동작(**read-modify-write** 동작)을 원자적으로 (인터럽트 되지 않고/분리되지 않고) 수행하는 기계어 명령어

- Atomic 명령어의 예
  - **Test and Set (TAS)** instruction:
    - word 내용을 검사(test)하고 변경(set)하는 동작을 원자적으로 수행

    (ex) 80386: bit test and set (test, set)

    BTS [100], 2 ; CF ← $M[100]_2$, $M[100]_2$ ← 1

    68000: test and set (test, set)

    TAS $5000 ; N ← $M[5000]_7$, $M[5000]_7$ ← 1
  - **Swap** instruction:
    - 두 word의 내용의 교환(swap)을 원자적으로 수행

    (ex) 80386: exchange

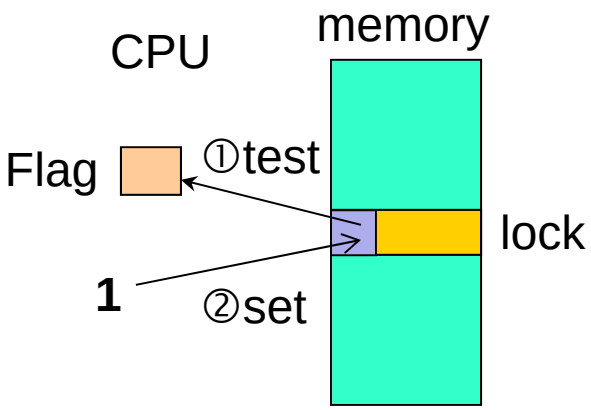
    XCHG AX, [BX] ; AX ↔ M[BX]

Lock 변수 사용하기 – Atomic 명령어 사용

- **Lock** 변수의 사용
  - 두 상태: 0 (unlock, open, 사용 중이 아님), 1 (lock, close, 사용 중)
  - atomic 명령어를 사용하여 lock 변수를 조작함
- TAS 또는 SWAP 명령어를 사용한 lock 변수의 조작

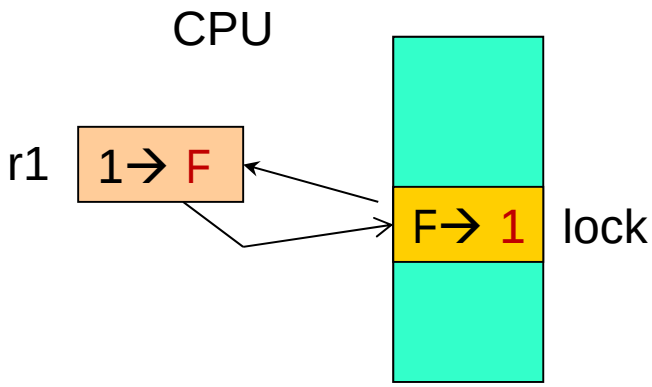
Test and Set (TAS) instruction

TAS lock



SWAP instruction

mov r1, 1   // set값 설정
swap r1, lock



Test-and-Set을 사용한 상호 배제

- 공유 데이터
  - **lock** = 0; // **global** shared data – unlock

- Process P_i

entry section

```
while ( TestAndSet(&lock) )
        ;
```

… critical section

exit section

```
lock = 0;
```

… remainder section

→ **busy waiting**
wait loop

TestAndSet() 연산은
TAS 또는 Swap 명령어로
구현 가능함

- 필요조건 충족 여부
  - 이 알고리즘은 상호 배제 및 진행 조건을 만족시킴
  - 한정대기 조건은 만족시키지 못함

Test&Set를 사용한 한정 대기를 만족하는 상호 배제

- 공유 자료구조
  - boolean **waiting**[n]; // initialized to false(0)
  - boolean lock; // initialized to false(0)
- Process Pi

entry section

```
waiting[i] = TRUE;  key = TRUE;
while (waiting[i] && key)
        key = TestAndSet(&lock);
waiting[i] = FALSE;
```

→ unlock상태이면 key=0이 됨

... critical section

exit section

```
j = (i + 1) % n;
while ( (j != i) && ! waiting[j] )
        j = (j + 1) % n;
if (j == i) lock = FALSE;
else waiting[j] = FALSE;
```

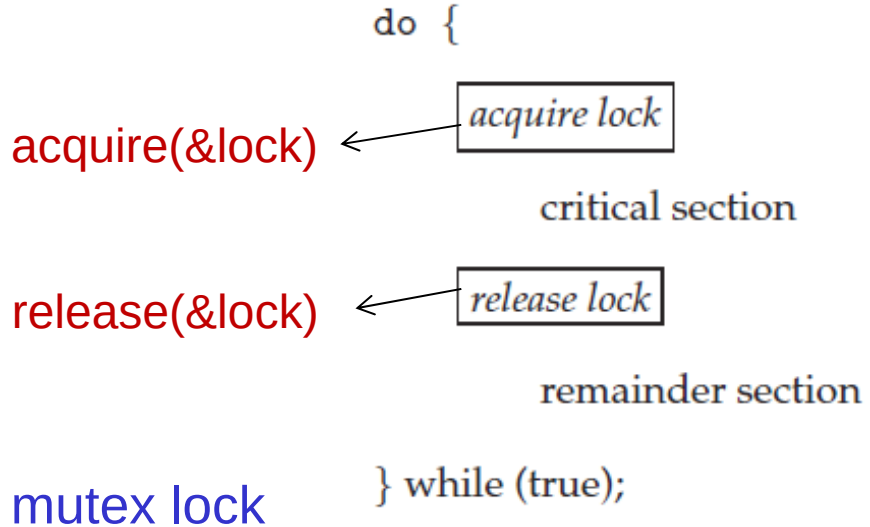
... remainder section

cyclic ordering :

- Pi 다음 프로세스부터 대기 여부 검사(waiting[j]=true)
- 임계구역 진입을 기다리는 프로세스는 최대 n-1회 내에 진입 가능

6.5 뮤텍스 락(Mutex Locks)

- 임계 구역 문제에 대한 하드웨어 기반 해결안
  - 응용 프로그래머에게는 복잡하며 일반적으로 직접 사용할 수 없음
  - 해결방법 – 운영체제에서 응용프로그래머에게 이를 위한 소프트웨어 도구를 제공 → Mutex Lock
- Mutex Lock (상호배제 락)
  - **Mut**ual **Ex**clusion의 줄임말
  - 두 함수 제공
    - **acquire()** – lock 획득
    - **release()** – lock 반환



- 스핀락(Spinlock) : busy-waiting mutex lock
  - 루프를 반복 실행하면서 lock 획득을 기다림
  - CPU cycle이 낭비됨 → 단일 프로세서 시스템에서 특히 문제임
  - 멀티프로세서 시스템에서 짧은 시간 내에 lock 획득이 예상되면 spinlock이 유용함 - context switching이 없음

Busy-waiting이 없는 Mutex Lock

- Busy-waiting이 없는 Test and Set을 사용한 상호 배제
  - lock을 획득하지 못하면 프로세스는 lock을 기다리는 대기상태로 전환하고 CPU를 내어 놓음
- 알고리즘
  - **lock** = 0;
  - Process P_i

entry section = acquire( )

```
while ( TestAndSet(&lock) )
        block();
```

→ **nobusy waiting**

… critical section

exit section = release( )

```
lock = 0;
```

… non-critical section

}

다른 프로세스가 release()를 호출하여 unlock이 될 때에 **wakeup()**이 호출되어 wait 상태에 있는 block된 프로세스를 깨움

6.6 세마포(Semaphores)

사전 뜻: 수기 신호

- 세마포(Semaphore)
  - Mutex Lock보다 더 정교하고 강력한 프로세스 동기화 도구
  - **세마포 S**
    - 특별한 **표준 동기화 연산**을 통해서만 접근 할 수 있는 정수 변수.
    - 세마포 값은 대개 사용 가능한 특정 자원의 수를 나타냄
- Semaphore 연산
  - 초기화 연산 – semaphore S 값 초기화
  - 두 개의 원자적(*atomic*) 연산 (by Dijkstra) 다른 표기
    - P operation (**P**roberen = test) → wait(S), acquire(S)
    - V operation (**V**erhogen = increment) → signal(S), release(S)

P(S)

```
while (S ≤ 0)
      ;        // busy-wait
// now S > 0
S = S - 1;
```

V(S)

```
S = S + 1;
```

세마포 S값 (검사 및) 변경은 원자적으로 실행되어야 함

Semaphore 용도

- 용도
  - 상호 배제(Mutual exclusion) → 이진 세마포(binary semaphore)
  - 유한 개수의 자원 접근, 한정된 concurrency → counting semaphore
  - 프로세스 동기화: signaling
- Binary semaphore ⇒ 상호배제 구현 (mutex lock과 유사)
  - semaphore 값: 0 or 1 (1로 초기화)
  - 임계구역 접근 제어에 사용

Semaphore **S** = **1**; // initialize

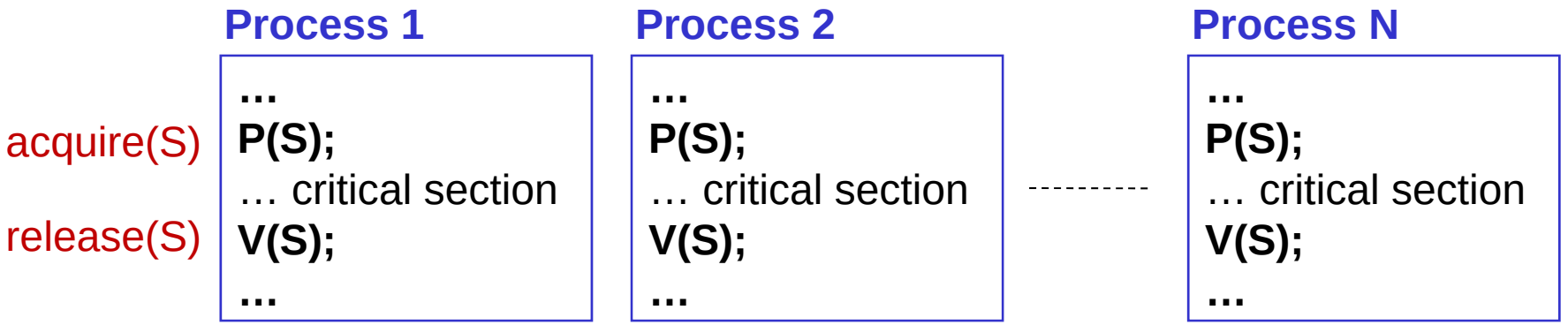
S=1 : unlock
S=0 : lock

	Process 1	Process 2	
	…	**…**	
wait(S)	**P(S);**	**P(S);**	acquire(S)
	… critical section	… critical section	
signal(S)	**V(S);**	**V(S);**	release(S)
	…	**…**	

카운팅 세마포(Counting Semaphore)

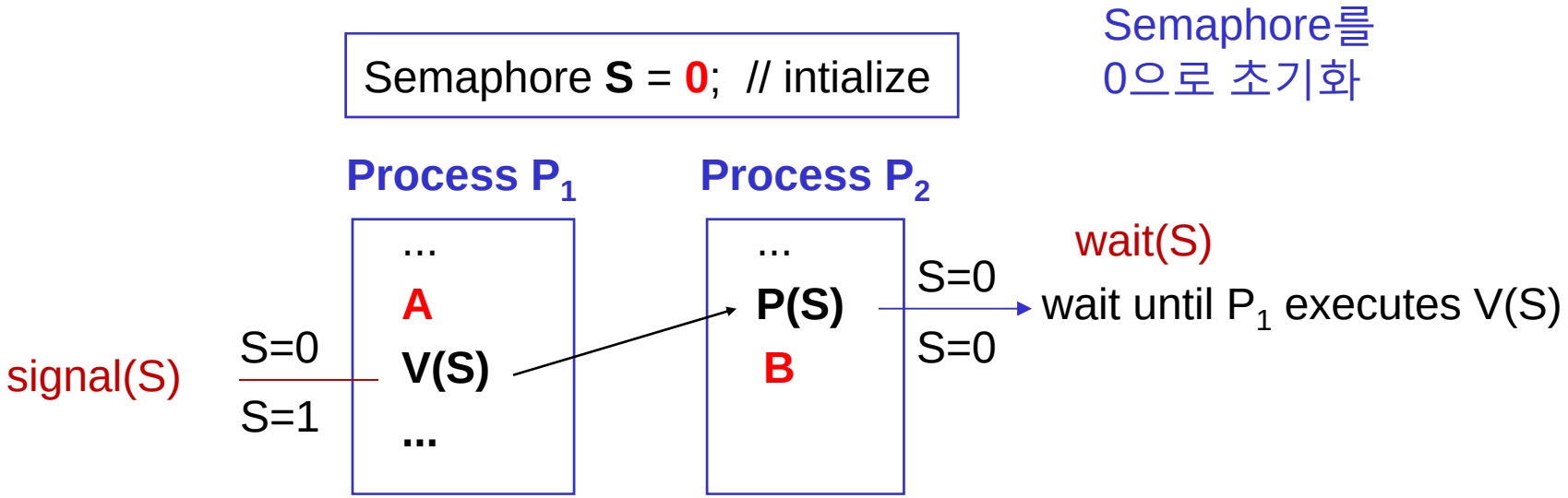
- Counting semaphore ⇒ 한정된 concurrency, 유한 개수 자원 접근
  - semaphore 값: 가용 자원 개수를 의미(최대 가용 자원 개수로 초기화)
  - 유한 개수의 자원의 접근 제어에 사용

Semaphore **S** = **n**; // initialize ⟶ 최대 ***n*** 프로세스가 병행하여 수행될 수 있음



일반적인 동기화

- 동기화(synchronization)
  - (예) 프로세스 **P_1의 A 부분** 실행 후에 프로세스 **P_2의 B 부분**이 실행되어야 함
  - Code:



Semaphore 구현

- 바쁜 대기(Busy waiting) Semaphore
  - spinlock과 유사하게 구현

- 바쁜 대기 없는(No Busy waiting) Semaphore
  - 프로세스의 block-wakeup 방법 이용

  - semaphore를 획득할 수 없을 때 semaphore 대기 큐에 자기 프로세스를 추가하고 자신을 **block** 시킴
  - semaphore를 사용 가능하게 되면, semaphore 대기 큐에서 한 프로세스를 꺼내어 ready queue로 이동하여, 대기 중인 하나의 프로세스를 **wakeup**시킴

  - 코드 → 다음 slide

No busy-waiting 세마포 구현

- Sempahore **S** – 다음 두 item으로 자료구조가 구성됨
  - **value** – 세마포 값
  - **list** – 이 세마포를 대기하는 프로세스 리스트 (세마포 대기 큐)
- Semaphore 연산

```
P(S)
{
    S.value - -;
    if (S.value < 0) {
        add this process to S.list
        block()
    }
}
```

```
V(S)
{
    S.value++;
    if (S.value <= 0) {
        remove a process Q from S.list
        wakeup(Q);
    }
}
```

- 세마포 값을 먼저 감소 → 음수 가능
- 음수의 크기는 세마포를 대기하는 프로세스들의 개수임
  (세마포 검사와 감소의 순서가 busy-waiting 방법과 반대)
- 두 프로세스가 같은 세마포에 대해 P 또는 S 연산이 중첩되어 실행되지 않도록 해야 함. (atomic 실행)

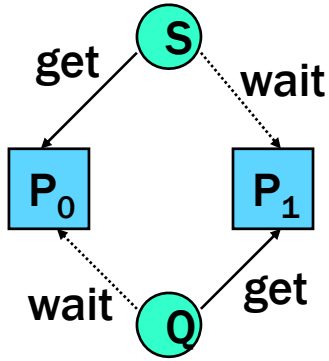
교착상태(Deadlock)와 기아(Starvation)

- Semaphore를 사용한 코드는 교착상태와 기아가 발생할 수 있음
- 교착상태(Deadlock)
  - 두 개 이상의 프로세스들이 그들 중 한 프로세스에 의해서만 발생될 수 있는 사건을 무한정 기다리고 있어서, 진행이 되지 않는 상황

(ex) 두 binary semaphore S, Q: S=1, Q=1로 초기화 됨

둘째 P 연산들이 교착상태가 됨

P_0	P_1
P(S);	P(Q);
P(Q);	**P(S);**
…	…
V(S);	V(Q);
V(Q)	V(S);



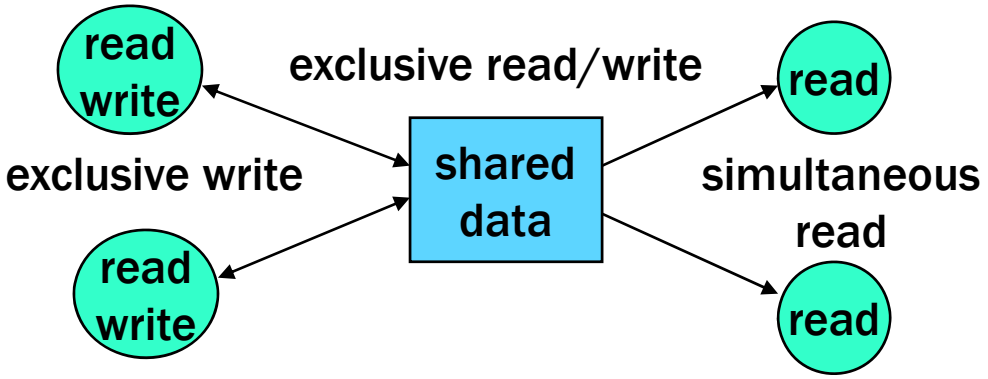
- 기아(Starvation) – 무한 봉쇄(indefinite blocking)
  - 일부 프로세스가 무한정 대기할 수 있는 상황
  - (예) 세마포 대기 리스트가 LIFO 순서로 되어 있으면 리스트 앞쪽에 있는 프로세스는 제거되지 못할 수 있음(세마포가 빈번히 사용되는 경우)

6.7 고전적인 동기화 문제

- 유한 버퍼 문제(Bounded-Buffer Problem)

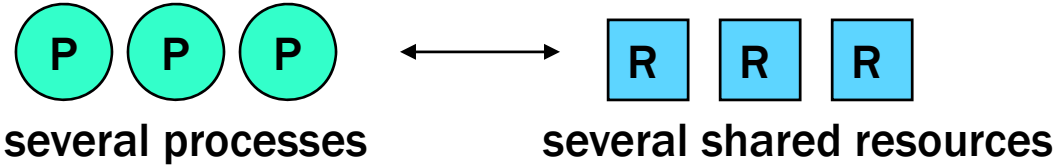
두 프로세스가
하나의 자료 공유

- Readers와 Writers 문제



여러 프로세스가
하나의 자료 공유
- 일부는 읽기만 수행
- 일부는 갱신 수행

- 식사하는 철학자 문제(Dining-Philosophers Problem)



여러 프로세스가
여러 자료 공유

Bounded-Buffer Problem – 세마포 사용

int n;
Semaphore full = **0**;
Semaphore empty = **n**;
Semaphore mutex = **1;**

- Producer

```
do {
    …. produce an item
    P(empty);
    P(mutex);
    ++count;
    buffer[in]=item;
    in=(in+1)%BUFSIZE;
    V(mutex);
    V(full);
}
```

// **n**: buffer size
// **empty, full**: counting semaphore
for synchronization
// **mutex**: semaphore for buffer accesses

- Consumer

```
do {
    P(full);
    P(mutex);
    --count;
    item = buffer[out];
    out=(out+1)%BUFSIZE;
    V(mutex);
    V(empty);
    … consume the item
}
```

Readers-Writers Problem – 세마포 사용

Semaphore mutex = 1
Semaphore rw_mutex=1;
int read_count = 0
// **mutex**: for updating read_count
// **rw_mutex**: for updating shared database (common to reader/writer)

■ Writer

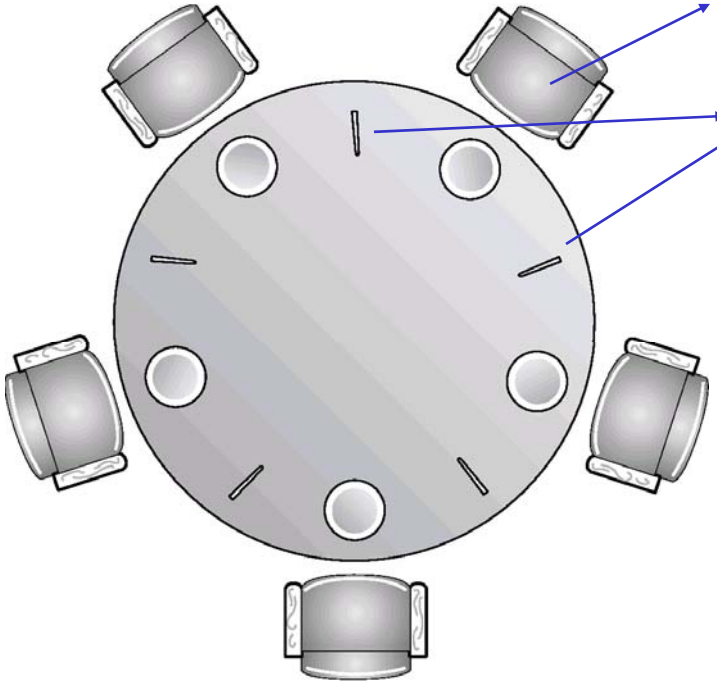
```
P(rw_mutex);        acquireWriteLock
  …
  writing is performed
  …
V(rw_mutex);        releaseWriteLock
```

■ Reader

```
P(mutex);                          acquireReadLock
read_count++;
if (read_count==1)                 // first reader
    P(rw_mutex);
V(mutex);
  …
  reading is performed
  …
P(mutex);                          releaseReadLock
read_count --;
if (read_count==0)                 // last reader
    V(rw_mutex);
V(mutex);
```

일부 시스템에서는 read-writer lock(rwlock)과 이를 위한 연산을 제공한다.
rwlock을 획득하고 반환할 때에 용도(reader용, writer용)를 지정해야 한다.

Dining-Philosophers Problem



5명의 철학자가 thinking과 eating을
반복하며 시간을 보냄

양쪽 젓가락을 내려놓는다.

양쪽 젓가락을 집는다

- 식사하는 철학자 문제
  - 여러 프로세스들에게 여러 자원을 할당할 필요가 있는 병행 제어(concurrency control) 문제를 단순하게 나타낸 예임.
  - 철학자 → 프로세스(process), 젓가락 → 자원(resource)