VI. 프로세스 동기화 (Process Synchronization)

Topics

1. 배경
2. Critical Section 문제
3. 동기화 장치 (1): Mutex Lock
4. 동기화 장치 (2): Semaphore
5. 고전적인 동기화 문제들
6. 동기화 장치 (3): Monitor
7. 동기화 사례

1. Background

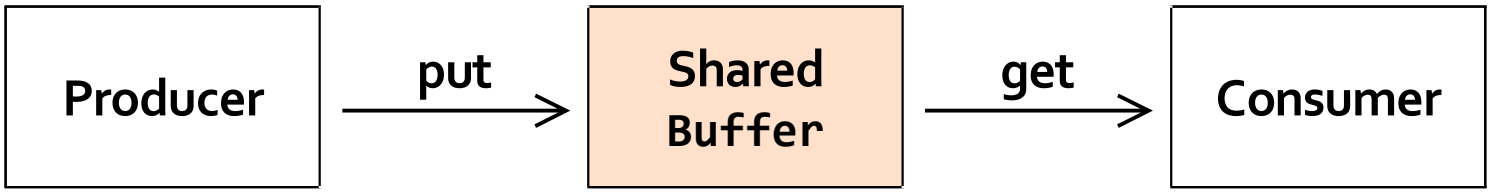
❑ 협력적 프로세스 (Cooperating Process)

(정의) 다른 프로세스에게 영향을 주거나,
다른 프로세스로부터 영향을 받는 프로세스

- 협력적 프로세스들은 보통 자원을 공유하며, 동시에 접근함
(Concurrent access to shared data)

(예) 생산자-소비자 문제 (Producer-Consumer Problem)

생산자는 ***shared buffer***에 item을 **put(write)**하며,
소비자는 ***shared buffer***에서 item을 **get(read)**함.



❑ **경쟁 상황** (Race Condition) 공유자원 접근時 실행결과가 접근순서에 의존하는 상황

< 유한 버퍼 문제 Bounded-Buffer Problem >

Producer	Shared Buffer	Consumer

while (true) {
  item 생산;
 	
  //wait until not full
  while(isFull() == true);
 	
  buffer[in] = item;
  in = (in+1) % n;
  count ++;
}
	int buffer[n];
	int count = 0;
	int in = 0, out = 0;
	
	count: 5.
	
	producer: 1개 생산.
	consumer: 1개 소비.
	
	⇨
	count should be 5.
		while (true) {
		  //wait until not empty
		  while(isEmpty() == true);
		
		  item = buffer[out];
		  out = (out+1)%n;
		  count --;
		
		  item 소비;
		}

Producer		Consumer	
R1 ← count (5)	T_0		경쟁 상황 발생. ⇨ *Data Consistency* 깨어짐: 실제 5개 items, count는 4.
R1 ← R1 + 1 (6)	T_1		
	T_2	R2 ← count (5)	
	T_3	R2 ← R2 - 1 (4)	
count ← R1 (6)	T_4		
	T_5	count ← R2 (4)	

❑ 프로세스 동기화 (Process Synchronization)

(정의) 공유자원의 일관성(*Consistency*) 유지를 위해
프로세스 間 실행 순서를 제어하는 것.

(note) 순차 접근과 동일한 결과를 산출하도록 실행 순서를 제어해야 함.

- 동기화 유형 - 경쟁적(*Competitive*) 및 협력적(*Cooperative*) 동기화.

Mutual Exclusion (상호 배타적 접근)	**Coordination: Wait & Signal.** (협력 or 조정)
한 순간 오직 하나의 프로세스만이 공유자원에 접근	• **Wait (대기):** - 버퍼가 full일 때 생산자는 대기. - 버퍼가 empty일 때 소비자는 대기. • **Signal or Notify (통지):** - put item 후 not empty 임을 소비자에게 통지. - get item 후 not full 임을 생산자에게 통지.

2. 임계 구역 문제

- **Critical Section** - 공유 자원을 접근하는 **code segment**

- 한 프로세스가 자신의 임계구역 內에 있을 때
  다른 프로세스들은 자신들의 임계구역으로 들어갈 수 없음.

- **Critical Section Problem** - 임계구역 전후에서 프로세스間 협력 규약을 설계하는 것.

(예) Producer의 임계구역

```
while (true) {
    item 생산;
    while(isFull()); //
    buffer[in] = item;
    in = (in+1) % n;
    count ++;
}
```

(예) Consumer의 임계구역

```
while (true) {
    while(isEmptyl()); //
    item = buffer[out];
    out = (out+1) % n;
    count --;
    item 소비;
}
```

나머지 구역
진입 구역
Critical Section
퇴출 구역
나머지 구역

- **critical section problem**의 해결 방안은 다음 3가지 **요구 조건**을 충족해야 함:

상호 배제 Mutual Exclusion	한순간 오직 하나의 프로세스만이 공유의 지정 임계구역에 들어갈 수 있음. (공유자원을 접근할 수 있음)
진행 Progress	임계구역을 실행중인 프로세스가 없고 임계구역을 진입하려는 프로세스들이 존재할 경우, 다음 프로세스의 선정은 무한히 연기되지 않아야 함. **(Note) Deadlock 방지**
유한 대기 Bounded Waiting	프로세스의 진입 요청은 유한한 시간 내에 허락되어야 함. **(Note) Starvation 방지**

동기화 장치	
하드웨어 장치	소프트웨어 장치
• Interrupt disabling • Special Atomic Instruction ◦ Test-and-Set instruction ◦ Compare-and-Swap instruction	• Mutex lock • Semaphore • Monitor
동기화 프로그래밍이 복잡해짐.	

3. Mutex Lock

Mutex Lock	동기화 (사용)방법

available : boolean // lock의 가용 여부를 나타냄.
 	
acquire() { // lock 획득, atomic operation.
while (!available) ; // *busy waiting – 계속 while*
available = false;
}
release() { // lock 반환, atomic operation.
available = true;
}
	do {
	…
	acquire( ); // lock 획득
	임계 구역
	release( ); // lock 반환
	…
	}

- (단점) 프로세스는 lock이 가용할 때까지 spin. (note) mutex lock을 *spinlock* 이라 함.
- (장점) context switching 필요 없음. spin: 돌다, 뱅뱅 돌다.
- 적용
  - 프로세스들이 짧게 lock을 소유하는 경우.
  - 다중 처리기 시스템:
    한 처리기에서 thread가 임계구역 실행하는 동안
    다른 처리기에서 다른 thread는 spin.

4. Semaphore

Semaphore – Mutex 보다 더 정교한 동기화 장치	동기화 (사용)방법

S : integer – 가용한 자원의 개수를 나타냄
S >= 1 : S개 자원이 가용함.
S == 0 : 가용한 자원 없음.
S < 0 : 기다리고 있는 프로세스의 개수.
 	
wait(S) { // 공유 자원 획득, atomic operation.
while (S <= 0) ; // busy waiting
S --;
}
signal(S) { // 공유 자원 반환 , atomic operation.
S ++;
}
	do {
	…
	wait(s); //공유 자원 획득
	임계 구역
	signal(s); //공유 자원 반환
	…
	}

- 유형: Counting Semaphore: **unbounded domain**, 유한 개의 공유 자원 접근 時 사용.
  Binary Semaphore (≈Mutex): **domain = {0, 1}**, **Mutual exclusion** 보장 時 사용.
- 다양한 동기화 문제에도 사용됨
  (예) S1 실행 후 S2 실행: Process1: S1; signal(sync); Process2: wait(sync); S2;

Semaphore **with** Busy waiting **(Spinlock)**	Semaphore **w/o** Busy waiting **(Semaphore with Wait queue)**

// 가용 자원 개수
semaphore : integer
	typedef struct {
	  int value; // 가용 자원 개수
	  struct process *waitQ; // 대기 큐
	} Semaphore;

wait(s) {
  while (s <= 0)
  ; // *busy waiting*
  s --;
}
	wait(s) {
	  s.value --;
	  if (s.value < 0) {
	  현행 프로세스를 대기 큐에 추가;
	  block(); // 프로세스 일시 중지
	  }
	}

signal(s) {
  s ++;
}
	signal(s) {
	  s.value ++;
	  if (s.value <= 0) {
	  대기 큐에서 한 프로세스(P) 제거;
	  wakeup(P); // 프로세스 P 실행 재개
	  }
	}

❑ Note

Semaphore with **busy waiting**	Semaphore with **wait queue**
- multi-processor 환경에서 유리	- uni-processor 환경에서 유리
	- wait(), signal()의 atomicity는 interrupt disabling으로 쉽게 해결 - starvation과 deadlock 발생 가능 (예) LIFO 대기 큐에서의 deadlock: P0: wait(S) 後 wait(Q) ⇨ wait on Q P1: wait(Q) 後 wait(S) ⇨ wait on S

우선순위 전도 (*Priority Inversion*)	(해결책) 우선순위 상속 규약 (*Priority-Inheritance Protocol*)
① 낮은 순위 프로세스(L)가 공유자원 사용 중 ② 높은 순위 프로세스(H)가 공유자원 요청➜대기 ③ L이 중간 순위 프로세스(M)에 의해 preempted. ⇨ M 때문에 H의 대기 시간이 더 길어짐.	L이 공유자원을 사용하는 동안 H가 공유자원 요청하면 L의 우선순위를 H의 우선순위로 일시 변경함.

5. 고전적인 동기화 문제들

- **Bounded buffer problem (*or* Producer-consumer problem)**
- **Readers-Writers problem**
- **Dining philosophers problem**

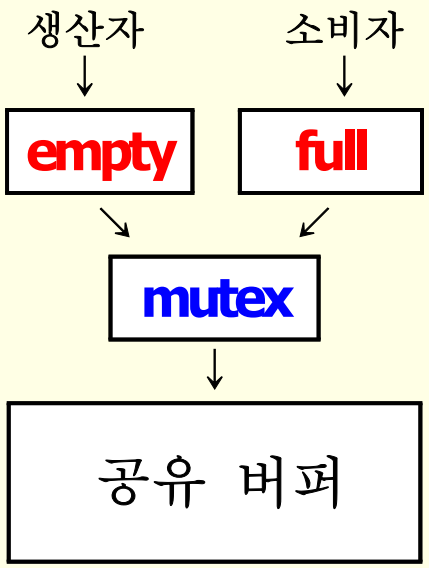
❑ Bounded Buffer Problem (or Producer-consumer problem)

- 공유 버퍼 (크기 n≧1), 다수의 생산자, 다수의 소비자.
- Semaphore
  - **mutex** 공유버퍼 가용성 (initially 1) for *mutual exclusion*.
  - **empty** empty slot 개수 (initially n) for producer's *coordination*.
  - **full** full slot 개수 (initially 0) for consumer's *coordination*.

생산자 프로세스

```
do {
    item 생산;
    wait(empty); // 협력
    wait(mutex); // 독점
    버퍼에 추가 // 임계구역
    signal(mutex); // 깨움
    signal(full);  // 깨움
} while (true)
```

공유자원 & 세마포



소비자 프로세스

```
do {
    wait(full);      // 진입
    wait(mutex);     // 구역
    버퍼에서 제거 // 임계구역
    signal(mutex); // 퇴출
    signal(empty); // 구역
    item 소비;
} while (true)
```

(note) 다수의 생산자, 다수의 소비자

❑ Readers-Writers Problem

- 공유 data set, 다수의 reader, 다수의 writer.
  - **reader) read** 요청 時 **동시 접근**이 허용됨. (note) 유한버퍼와의 차이점
  - **writer) write** 요청 時 상호 배타적 접근 만 허용됨.

  (예) **concurrent process**들에 의해 공유되는 데이터베이스

- 일부 운영체제(ex. Solaris)는 ***reader-writer lock***을 제공함:
  - **read** 요청 時 동시 접근 허용
  - **write** 요청 時 상호 배타적 접근

- **Readers-Writers Problem의 변형들**

	reader, writer 間 우선순위	starvation 가능성?
변형 1	writer가 data set 접근 허가를 얻지 못한 경우 어떤 reader도 다른 reader를 기다리게 하지 않아야 함.	writer
변형 2	새로운 writer는 최대한 빨리 수행시킴 (대기 중인 writer는 새로운 readers를 우선함)	reader

(예 1) 도착순서: R1 W2 R3 R4

R1: data set 접근; W2: blocked; 새로 도착한 R3, R4: ??

(예 2) 도착순서: W1 R2 R3 R4

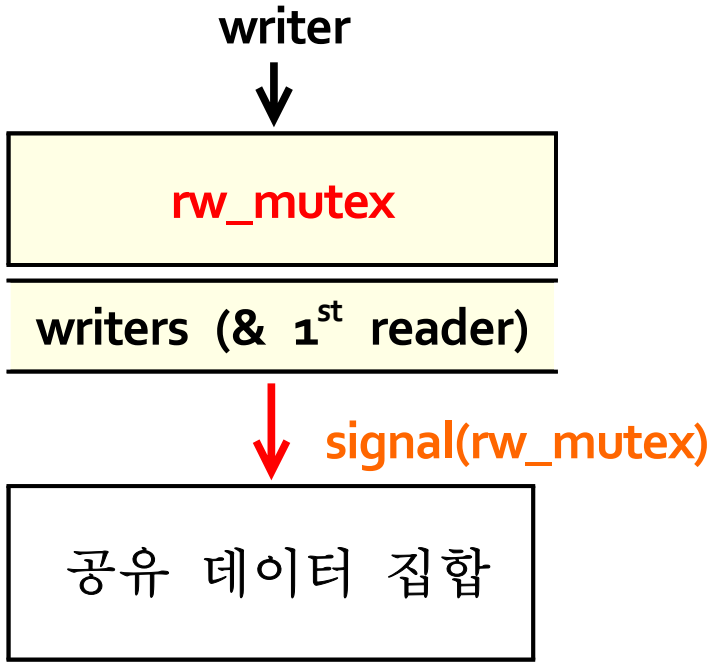
W1: data set 접근; 새로 도착한 R2, R3, R4: blocked.

W1 종료, R2가 선택된 경우 이미 대기 중인 R3, R4: ??

- 변형 1

Writer

```
do {
        wait(rw_mutex);
        쓰기 수행
        signal(rw_mutex);
} while (true)
```



(note)

- signal(rw_mutex) 時 대기 중인 writers와 1st reader는 CPU scheduler에 의해 선택됨.
- 대기 중인 1st reader 보다 나중에 도착한 writer가 먼저 선택될 수 있음.

Reader

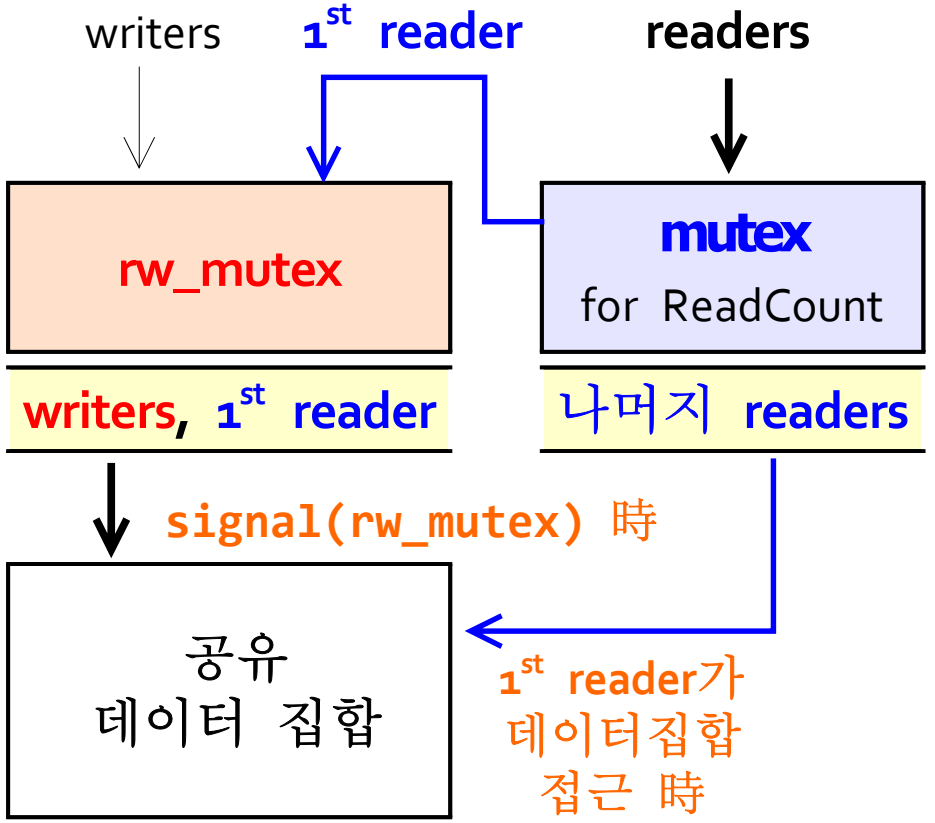
```
do {
    ④ wait(mutex);

    if (++readcount==1) // first reader
            ① wait(rw_write);
    // 1st reader가 다음 reader를 깨움
    ② signal(mutex); // recursive

    ③ 읽기 수행 // critical section

    wait(mutex);
    if (--readcount==0)  // last reader
            ⑤ signal(rw_write);
    signal(mutex);

} while (true)
```

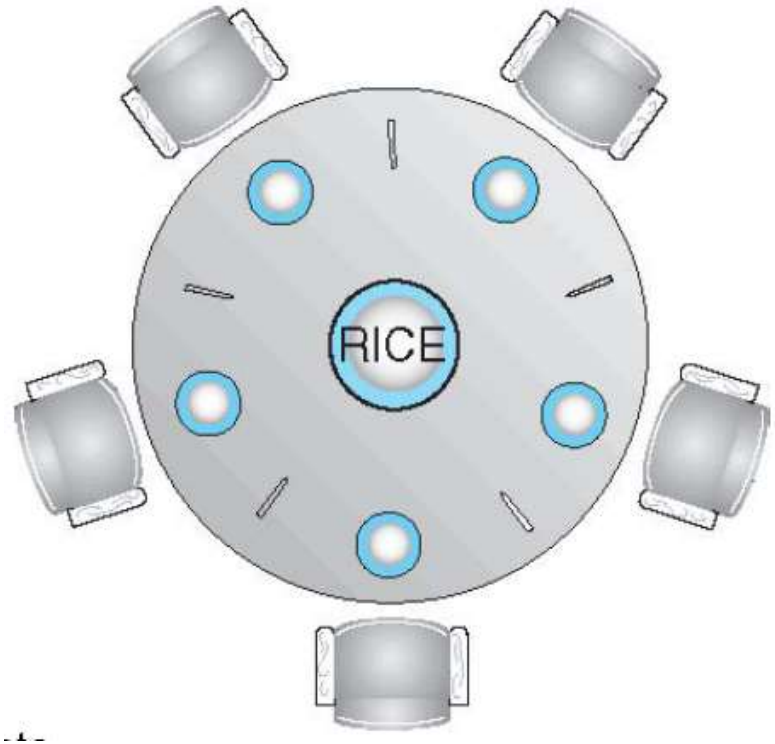


1st reader:

① 공유데이터집합 접근권한 획득
② mutex에서 대기 중인 다음 reader 깨운 後
③ 읽기 수행
④에서 깨어난 reader는 차례로 ②, ③
⑤ last reader는 rw_mutex 상의 process 깨움

❑ Dining-Philosophers Problem

- 공유 자원: 5개 젓가락
- 세마포: chopstick[5]



i 번째 Philosopher 프로세스

```
do {
    wait( chopstick[i] );
    wait( chopstick[(i+1)%5] );

    eat

    signal( chopstick[i] );
    signal( chopstick[(i+1)%5] );
} while (true)
```

- 상호 배제 보장 ○, **deadlock** 보장 ×
- **deadlock** 해결 방안)

  교재 **p271**, 강의노트 **p25** 참고.
- **starvation** 방지도 고려해야 함

6. 모니터 (Monitor)

- **Semaphore**의 문제점

  Error 유발 가능성이 높음:

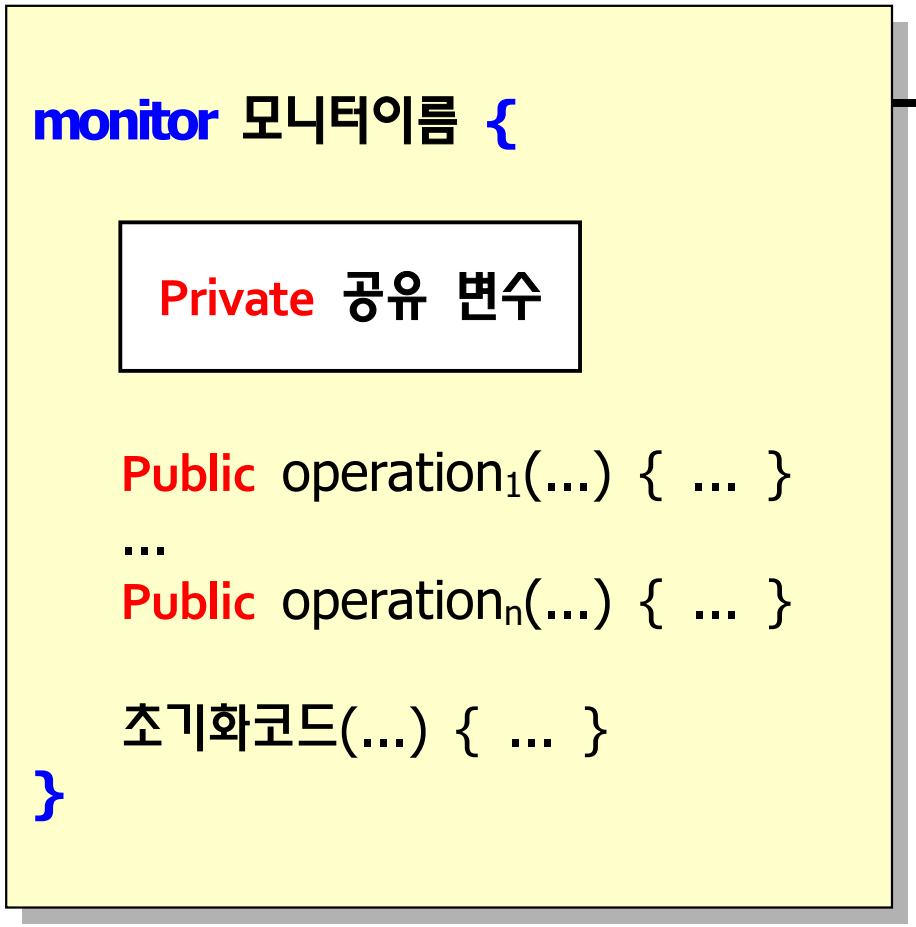
  - **mutual exclusion** 보장 못함.
  - **deadlock** 발생 가능.

상호배제 ×	Deadlock 발생	상호배제 × 또는 Deadlock 발생	

signal(mutex);
임계 구역
wait(mutex);
	wait(mutex);
	임계 구역
	wait(mutex);
		임계 구역
		signal(mutex);
			wait(mutex);
			임계 구역

❑ Monitor

- Abstract data type of shared resource.
- mutual exclusion을 자동 보장함. (cf) Java class with synchronized methods.



진입 큐

```
class SharedBuffer { //상호배제 보장함.
    private int[] buf = new int[100];

    public synchronized int get() {
        ...
    }
    public synchronized void set(int x) {
        ...
    }
} // class + synchronized methods = monitor
```

❑ 조건 변수 Condition Variable - *coordination*을 프로그래밍하기 위한 장치.

condition 조건변수;

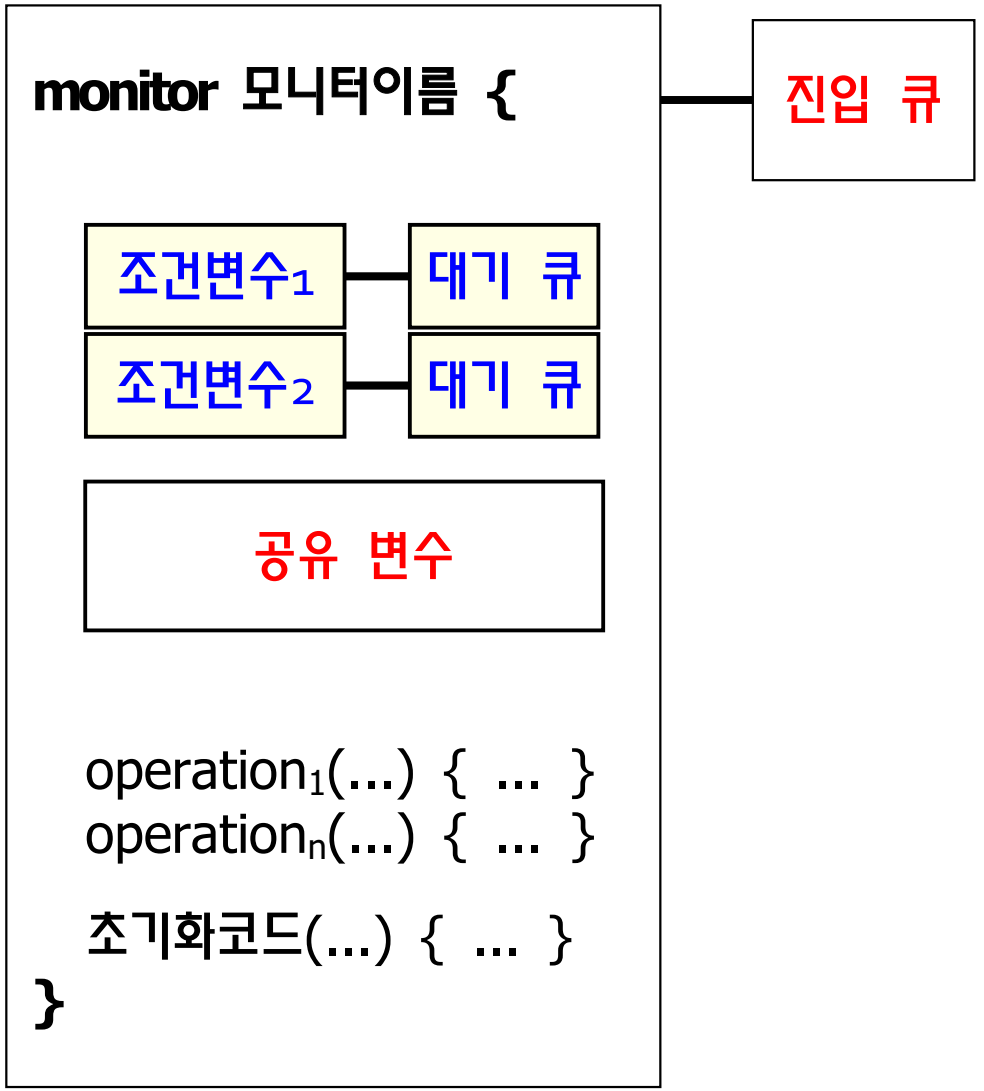
허용되는 연산:

wait() :

프로세스를 일시 중지시킴.

signal() :

```
if (대기 중인 프로세스 있음) {
    대기 큐에서 하나의 프로세스를
    선택하고 실행 재개 시킴;
} else {
    Do noting;
}
```



- **signal()의 의미**

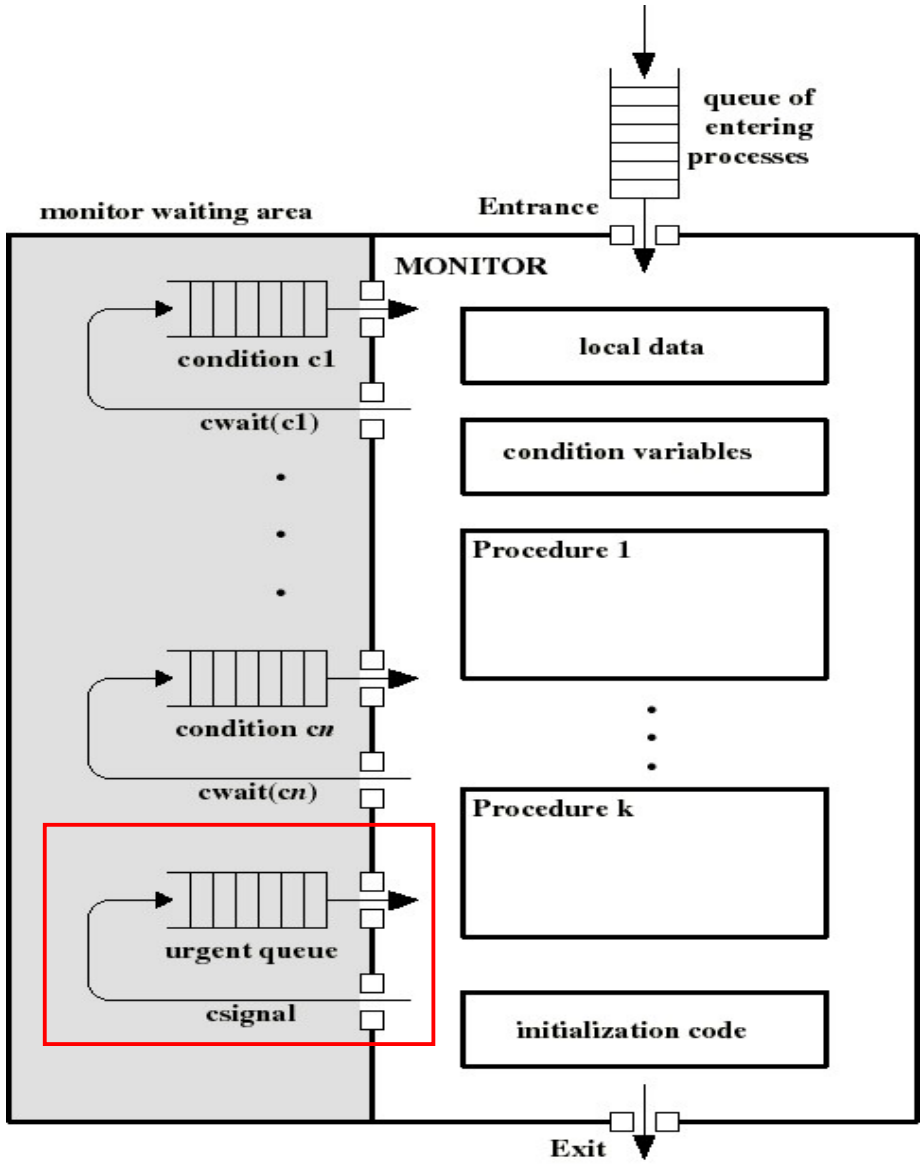
(note) 오직 한 프로세스만이
모니터內에서 활성화 가능.

프로세스 P가 cv.signal() 호출 時,
프로세스 Q가 cv.wait() 상태에 있는 경우

- **Signal & Wait**
  P는 Q가 모니터를 떠날 때 까지
  *urgent queue*에서 대기함.

- **Signal & Continue**
  P는 signal 후 실행을 계속함.
  (Q는 *urgent queue*에서 대기함.)
  ⇨ Q가 재개될 때
  조건변수가 '참'이 아닐 수 있음.

- **Concurrent Pascal의 절충.**
  P는 즉시 모니터를 떠나며,
  Q가 즉시 재계됨



❑ 모니터를 이용한 producers-consumers problem

```
monitor SharedBuffer {
    int[] buf = new int[100]; // 공유 버퍼
    int in, out, count;
    condition full, empty; // 대기 원인: buffer full/empty

    initialization() { in=0; out=0; count=0; }

    void put(int item) {
        if (count==100) full.wait(); // buffer full이면 대기함.
        buffer[in]=item; in=(in+1)%n; count++;
        empty.signal(); // 하나의 consumer를 실행재개 시킴.
    }

    Item get() {
        if (count==0) empty.wait(); // buf. empty이면 대기함.
        int item=buffer[out]; out=(out+1)%n; count--;
        full.signal(); // 하나의 producer를 실행재개 시킴.
        return item;
    }
}
```

Producer:

```
while (true) {
    item 생산;
    put(item);
}
```

Consumer:

```
while (true) {
    item = get();
    item 소비;
}
```

❑ 모니터를 이용한 dining-philosopher problem

- enum {생각중, 배고픔, 식사중} 타입의 상태[5] for deadlock prevention예방:
  양쪽 젓가락 모두 가용할 때만 사용도록 하기 위해 철학자 상태를 표시함.

- condition 자신[5] for coordination

```
monitor DiningPhilosophers {
  enum {생각중, 배고픔, 식사중} 상태[5];
  condition 철학자[5];

  initialization() {
    for (int i=0; i<5; i++) 상태[i] = 생각중;
  }

  void pickup(int i) { // i 번째 철학자
    상태[i] = 배고픔;
    test(i);
    if (상태[i] != 식사중)
        철학자[i].wait(); //자신 대기
  }

  void putdown(int i) {
    상태[i] = 생각중;
    test( (i+4)%5 ); //좌 깨움
    test( (i+1)%5 ); //우 깨움
  }
  test(int i) {
    if( (상태[i] == 배고픔) //데드락 예방
    && (상태[ (i+4)%5 ] != 식사중)
    && (상태[ (i+1)%5 ] != 식사중) ) {
        상태[i]=식사중;
        //putdown() 時 좌or우 철학자 깨움
        철학자[i].signal(); //pickup() 時 do nothing
    }
  }
}
```

```
monitor DiningPhilosophers {
   enum {생각중, 배고픔, 식사중} 상태[5];
   condition 철학자[5];

   initialization() {
      for (int i=0; i<5; i++) 상태[i] = 생각중;
   }

   void pickup(int i) { // i 번째 철학자
      상태[i] = 배고픔;

      //데드락 예방한 후
      if( (상태[ (i+4)%5 ] != 식사중)
      && (상태[ (i+1)%5 ] != 식사중) ) {
            상태[i]=식사중;

      if (상태[i] != 식사중)
            철학자[i].wait(); //자신 대기
   }

   void putdown(int i) {
      상태[i] = 생각중;
      test( (i+4)%5 ); //좌 깨움
      test( (i+1)%5 ); //우 깨움
   }

   test(int i) {
      // "배고픔" 철학자에 대해
      if( (상태[i] == 배고픔)
      //데드락 예방한 후
      && (상태[ (i+4)%5 ] != 식사중)
      && (상태[ (i+1)%5 ] != 식사중) ) {
            상태[i]=식사중;
            철학자[i].signal(); //옆 철학자 깨움
      }
   }
}
```

7. 운영체제 사례

❑ Solaris

semaphore, condition variable, reader-writer lock, adaptive mutex, turnstile 등 제공

Adaptive mutex	요청한 lock을 보유한 스레드가 다른 CPU에서 - 실행 중이면 요청한 스레드는 spin - 아니면 요청한 스레드는 sleep	• 다중 처리기 환경에서 사용됨 • 공유 자원 접근 코드가 짧을 때 (수백 라인) 사용됨
Turnstile 회전문	- adaptive mutex 또는 reader-writer lock을 위한 대기 큐 - *Per-thread* (not per-synchronized object) • 동기화 객체의 turnstile(대기 큐) ⬅ 첫 번째로 block되는 thread의 turnstile • 이 후 봉쇄되는 thread들은 이 turnstile에 추가됨 • 최초 blocked thread가 동기화객체 사용 완료 時 커널은 새로운 turnstile을 할당함 - *priority inheritance protocol* 사용함	

❑ Windows XP

- global resource 접근 時
  단일 처리기에서는 interrupt disabling,
  다중 처리기에서는 spin lock을 사용함.

- Spinlock) 짧은 코드에서만 사용, 효율성 제고 위해 not preempted.

- Dispatcher object
  - mutex, semaphore, event(≈조건 변수), timer처럼 행동할 수 있음.
  - 상태:

    signaled state (객체가 가용함)

    non-signaled state (가용하지 않음)

8. Java의 동기화 장치 - **Java 1.5** 이전

	Java 1.5 이전
상호배제	Object's Intrinsic Lock 동기화 문장: Synchronized Method Synchronized Block
협력	wait() signal()

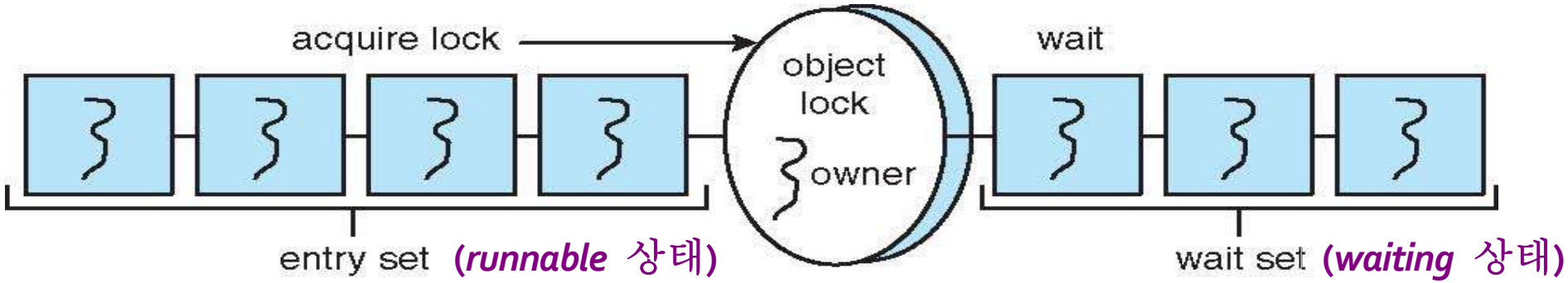
(1) Java monitor = Any object + Synchronized methods.

```
public class BoundedBuffer<E>
implements Buffer<E> {
    private static final int SIZE=5;
    private int count, in, out;
    private E[] buffer; // 공유 버퍼
    // 초기화 코드
    public BoundedBuffer() {
        count=0; in=0; out=0;
        buffer = (E[]) new Object[SIZE];
    }
    // synchronized methods
    public synchronized void insert(E item) {
        while (count == SIZE) wait(); // entry
        buffer[in] = item;            // critical
        in = (in+1)%SIZE; ++count;    // section
        notify();                     // exit
    }
    public synchronized E remove() { ... }
}
```

- 여러 thread가
  Producer p1 = new Producer();
  Consumer c2 = new Consumer();
- 어떤 (공유) 객체의
  sharedObj = new BoundedBuffer<>();
- ***synchronized method*** 호출 時
  p1: sharedObj.insert(3)
  c2: sharedObj.remove()
- mutual exclusion 자동 보장됨.
- **wait(), notify()**: cooperation 장치.

(2) Object Lock과 Synchronized method

모든 자바 객체는 자신의 lock을 가지며, 오직 하나의 thread에 의해 소유됨.



<하나의 진입 큐와 오직 하나의 대기 큐를 가지는 object lock>

synchronized method() 호출 時	synchronized method 탈출(exit) 時
thread는 lock을 획득 하거나 진입 큐에 들어감. (mutual exclusion이 자동 보장됨)	owner thread는 lock을 방출함; 이때 (진입 큐가 empty가 아니면) JVM은 임의의 thread를 선택함.

(3) wait()와 notify() - coordination 장치

```
public synchronized void insert(E item) {
```

while (count == SIZE) {
  wait();
}
	버퍼 full 時
	① lock을 방출하고 ② 대기 큐에 들어감
	✶ 다른 스레드가 **notify()**를 호출하면 진입 큐로 이동 가능.

buffer[in] = item;
in = (in+1)%SIZE;
++count;
	임계 구역

notify(); // notifyAll();	대기 큐 內 임의의/모든 스레드를 선택하여 진입 큐로 옮김

```
}
```

(note on while)

- 자바 모니터는 오직 하나의 대기 큐를 가짐. (모든 thread가 하나의 대기 큐에서 기다림)
- 즉, 자바 모니터는 하나의 무명 조건 변수를 가지는 모니터.
- 따라서 자바 모니터에서는 thread가 자신의 대기 조건을 나타낼 수 없음.

⇨ 깨어난 thread는 다시 (반복하여) 자신의 대기조건 만족 여부를 검사해야 함.

(예) Bounded Buffer Problem 프로그래밍 과제 (1)

```
public interface Buffer<E> { public void insert(E item); public E remove(); }
```

```
public class BoundedBuffer<E> implements Buffer<E> {

    private static final int SIZE=5; private int count, in, out; private E[] buffer;

    public BoundedBuffer() { count=0; in=0; out=0; buffer = (E[]) new Object[SIZE]; }

    public synchronized void insert(E item) {
        while (count==SIZE) wait(); // wait() 호출 時 InterruptedException 처리 필수.
        buffer[in] = item; in = (in+1)%SIZE; ++count;
        notify();
    }

    public synchronized E remove() {
        E item;
        while (count == 0) wait();
        item = buffer[out]; out = (out-1)%SIZE; --count;
        notify();
        return item;
    }
}
```

```
public class Producer
implements Runnable
{
    private Buffer<Date> buffer;

    public Producer(Buffer<Date> buffer) {
        this.buffer = buffer;
    }

    public void run() {
        Date message;
        while (true) {
            message = new Date();
            buffer.insert(message);
        }
    }
}
```

```
public class Consumer
implements Runnable
{
    private Buffer<Date> buffer;

    public Consumer(Buffer<Date> buffer) {
        this.buffer = buffer;
    }

    public void run() {
        Date message;
        while (true) {
            message =
                (Date)buffer.remove();
        }
    }
}
```

(4) Block Synchronization

- synchronized method는
  *noncritical section*을 포함한
  메소드 전체가 동기화됨.

- **Synchronized Block**
  - 메소드 일부를 동기화 함.
  - 명시적 **lock** 생성이 요구됨.

```
// mutual exclusion이 보장됨
syncronized(mutexLock) {
      ...임계구역...
}
```

- 동기화 단위가 보다 세밀해짐.

⇨ 동기화 비용이 낮아짐.

⇨ logic error 가능성↑

```
class Aclass {
    // 임의의 객체를 lock으로 사용
    Object lock = new Object();
    ...
    public void someMethod() {
        noncritical section
        syncronized(lock) {
            lock.wait();
            임계구역
            lock.notify();
        }
        noncritical section
    }
}
```

프로그래밍 과제 (2) - 유한버퍼문제

- 모든 클래스는 자신의 **a single class lock**을 가짐.

- **Static** synchronized block
  - ✶ **class lock**이 요구됨.

```
public void someMethod() {
    noncriticalSection
    static syncronized(SomeObject.class) {
        임계구역
    }
    remainderSection
}
```

- **Synchronized block** using **Class block**

  syncronized(SomeObject.class) { 임계구역 }

(5) InterruptedException의 처리

무시

```
public void someMethod() {
    ...
    try {
        wait(); // (note)
    } catch (InterruptedException) {
        /* ignore */
    }
    ...
}
```

전파

```
public void someMethod()
throws InterruptedException // propagate
{
    ...
    wait();
    ...
}
```

```
class caller() {
    try {
        someMethod();
    } catch (InterruptedException) {
        handles the propagated exception
    }
}
```

(note) wait()
- checks thread's interruption status.
- If set, throws an InterruptedException. (이때 status는 clear.)

  This allows interruption of a "blocked" thread in the wait set.

(6) 자바 동기화 문제

❑ ***Thread-safe*** **Application**

다수의 thread가 동시 접근하는 데이터의 consistency를 보장하는 application.

❑ **Deadlock과 Livelock**

- **Deadlock**

  Every thread in a thread set is blocked
  waiting for an event caused only by another blocked thread in the set.

- **Livelock**

  A thread continuously attempts an action that fails.

- **yield()**와 ***busy waiting***은 `livelock`을 야기할 수 있음.
  yield()와 **notify()**는 `deadlock`을 야기할 수 있음.

❑ yield() and busy waiting may lead to a Livelock.

```
public synchronized void insert(E item) {

    while (isFull()) Thread.yield() ;
    or
    while (isFull()) ; /*busy waiting*/

    buffer[in] = item;
    in = (in+1) % BUF_SIZE;
    ++count;
}
```

- JVM: 우선순위 기반 scheduling.
- Let: Priority$_{생산자}$ > Priority$_{소비자}$.

buffer full 時 :

⇨ yield() 실행

⇨ ready 상태로 전환됨

⇨ scheduling(dispatching) 됨

⇨ 이 과정을 무한 반복함.

✶ *busy waiting*의 경우도 동일함.

❑ yield() may lead to a Deadlock.

```
public synchronized void insert(E item) {

    while (isFull()) Thread.yield() ;

    buffer[in] = item;
    in = (in+1) % BUF_SIZE;
    ++count;
}
```

Assume:
버퍼 full, 모든 소비자 sleep.

buffer full 時 :
⇨ **yield()** 실행:
생산자는 lock을 보유한 채로 ready 상태로 전환됨.
⇨ 깨어난 소비자는 다시 block 됨.
⇨ 생산자, 소비자 모두 **can't proceed**.

(note)

- **wait()**는 owner thread의 **lock을 회수한 後** waiting 상태로 전환시킴.
- **wait()와 signal()**은 deadlock을 방지할 수 있음.

❑ notify() may lead to *another* Deadlock.

```
public synchronized void doWork(int me)
{
    while (turn != me) wait();
    do some work for a while …
    turn = (turn+1)%5;
    notify();
}
```

Assume: 5 threads. 현재 T3 실행 중.

- T3 종료시 turn = 4,
  wait set: { T0, T1, T2, T4 }
- T0, T1, T2가 wake up 時
  ⇨ 자신의 차례가 아니므로
  다시 wait set에 추가됨.
- **notify()**는 임의의 thread를 선택하므로 결국 T4가 선택되지 않으면 **deadlock**이 발생함.

(note)
notifyAll() prevents this kind of deadlock.

Rules of Synchronized

1. A thread that owns the lock for an object can enter another synchronized block (or method) for the same object. This is known as a recursive or reentrant lock.

2. A thread can nest synchronized method calls for different objects. Thus, a thread can simultaneously own the locks for several different objects.

3. If a method is not declared synchronized, then it can be invoked regardless of lock ownership, even while another synchronized method for the same object is executing.

4. If the wait set is empty, then a call to notify()/notifyAll() has no effect.

5. wait()/notify()/notifyAll() may only be invoked from synchronized method/block; otherwise an IllegalMonitorStateException is thrown.

Rules of Synchronized

1. A thread that owns the lock for an object can enter another synchronized block (or method) for the same object. This is known as a recursive or reentrant lock.

2. A thread can nest synchronized method calls for different objects. Thus, a thread can simultaneously own the locks for several different objects.

3. If a method is not declared synchronized, then it can be invoked regardless of lock ownership, even while another synchronized method for the same object is executing.

4. If the wait set is empty, then a call to notify()/notifyAll() has no effect.

5. wait()/notify()/notifyAll() may only be invoked from synchronized method/block; otherwise an IllegalMonitorStateException is thrown.

9. Java 1.5 동기화 장치

	Java 1.5 이전	Java 1.5 부터	
상호배제	Object Lock 동기화 문장: Synchronized Method Synchronized Block	Reentrant Mutex Lock Counterpart of synchronized statements	Semaphore
협력	wait() signal()	Condition Variable Counterpart of wait() and signal()	

자바 모니터 = 자바 객체 + object lock with 1 진입 큐 and only 1 대기 큐
+ 공유 데이터
+ synchronized methods.

자바 모니터 = Monitor with a single unnamed condition variable.

(1) Reentrant Mutex Lock - mutual exclusion 장치

재진입 가능한 mutex lock :

이미 자신을 소유하고 있는 thread가 다시 소유하는 것을 허용하는 lock.

✶ 허용되는 연산: lock(), unlock()

```
lock() {
    if (이미 소유 or 가용) {
        스레드에게 lock 소유권을 줌;
        return;
    } else {
        스레드를 block 시킴;
    }
}
```

사용 방법

```
Lock lock = new ReentrantLock();
            lock    진입 큐
aMethod() {
    ...
    lock.lock();
    try {
        임계 구역 // 상호배타적 실행
    } finally {
    // owner thread는 예외 발생 여부와
    // 무관하게 항상 lock 방출.
        lock.unlock();
    }
    ...
}
```

synchronized statement와 유사한 기능.

(2) Condition Variable 조건 변수 - Coordination 장치

생성 방법

① Lock lock = new ReentrantLock();

② Condition cv = lock.newCondition();

reentrant lock과 연관된
조건변수와 자신의 대기 큐

허용되는 연산: await(), signal()

await():

스레드는 lock을 방출하고,
대기 큐로 들어감 (waiting 상태가 됨).

signal():

대기 중인 스레드 중에서 하나를 깨움
(runnable 상태가 됨).

(note) 이때 lock은 방출하지 않음.
(note) lock은 unlock()에 의해 방출됨.

wait(), signal()과 유사한 기능.

(예) 유한 버퍼 문제 **프로그래밍 과제 (3)**

```
class BoundedBuffer {
    Object[] items = new Object[100]; // shared buffer
    int in, out, count;

    final Lock      lock  = new ReentrantLock();     // for mutual exclusion
    final Condition full  = lock.newCondition();     // for Coordination
    final Condition empty = lock.newCondition();     // for Coordination

    public void put(Object x) throws InterruptedException {
        lock.lock();            //① try to acquire the MUTEX lock.
        try {
            if (count == items.length) //② 공유버퍼가 full이면
                full.await();          //    lock을 방출하고, full 대기 큐에서 기다림.
            items[in] = x; if (++in == items.length) in=0; ++count; // critical region
            empty.signal();  //③ empty 대기 큐의 한 consumer를 깨움 (없으면 no op).
        } finally {
            lock.unlock();   //④ release the MUTEX lock.
        }
    }

    public Object get() throws InterruptedException {...}
}
```

Reentrant mutex lock & Condition variable	vs.	synchronized statement & wait()/signal()

```
Lock lock = new ReentrantLock();

// 스레드 전용 조건변수(대기 큐)
Condition[] cv = new Condition[5];
for (i=0; i<5; i++) cv[i] = lock.newCondition();

public void doWork(int me) {
      lock.lock();

      // 대기조건 만족 여부를 검사하고
      // 자신의 대기 큐에서 기다림.
      if (turn != me) cv[me].await();
      임계 구역
      turn = (turn+1)%5; // 다음 순번 설정.
      cv[turn].signal(); // 다음 순번 스레드 깨움

      lock.unlock();
}
```

```
public synchronized void doWork(int me)
{
      // 모든 스레드는 하나의 대기 큐에서
      // 기다리므로, 깨어 날 때마다 자신의
      // 대기조건 만족 여부를 반복 검사함.
      while (turn != me) wait();
      임계 구역
      turn = (turn+1)%5;
      notify(); // or notifyAll();
}
```

class + synchronized methods

⇒ 자바 모니터 with

- 하나의 진입 큐 for mutual exclusion
- 오직 하나의 대기 큐 for coordination.

즉, 자바 모니터는

하나의 무명 조건 변수를 가지는 모니터.

(3) Counting Semaphore - mutual exclusion & Coordination 장치

생성 방법

```
// Binary semaphore for mutual exclusion
Semaphore mutexLock;
mutexLock = new Semaphore(1);

// Counting semaphore for coordination:
// initially 5 available permits (resources)
Semaphore permits;
available = new Semaphore(5);
```

(note)

available = new Semaphore(0);
⇨ 한 번 release() 後 acquire() 가능함.

available = new Semaphore(-3);
⇨ 네 번 release() 後 acquire() 가능함.

허용되는 연산: acquire(), release()

```
acquire() {
    permits--; // 생성자의 실인수
    if (permits < 0) {
        스레드를 대기 큐에 추가함;
        block(); // waiting 상태로 전환.
    }
}

release() {
    permits++;
    if (permits <= 0) {
        대기 큐에서 한 스레드(P)를 제거함;
        awaken(P); // runnable 상태로 전환.
    }
}
```

사용 방법

Binary Semaphore	Counting Semaphore

Semaphore mutexLock;
mutexLock = new Semaphore(1);
 	
try {
  mutexLock.acquire();
  임계 구역
} catch (InterruptedException e) {
} finally {
  mutexLock.release();
}
	int permits = 10; // permits > 1
	Semaphore available;
	available = new Semaphore(permits);
	
	try {
	  available.acquire();
	  임계 구역
	} catch (InterruptedException e) {
	} finally {
	  available.release();
	}

(예) Mutual Exclusion

```
public class ConcurrentThreads {
   public static void main(String[] args) {

      // binary semaphore for mutual exclusion
      Semaphore lock;
      lock = new Semaphore(1);

      Thread[] worker = new Thread[5];
      for (i=0; i<5; i++)
            worker[i] = new Thread(
                  new Worker(lock));

            for (i=0; i<5; i++)
                  work[i].start();
      }
}
```

```
// 공유자원 접근하는 스레드
public class Worker
implements Runnable
{
      private Semaphore lock;
      public Worker(Semaphore s) {
            lock = s;
      }

      public vois run() {
            while (true) {
                  non-critical section
                  lock.acquire();
                  임계 구역
                  lock.release();
                  non-critical section
            }
      }
}
```

(예) 유한 버퍼 문제 프로그래밍 과제 (4)

```
public class BoundedBuffer<E>
implements Buffer<E>
{
   private E[] buffer; // 공유 버퍼
   private static final int SIZE = 5;
   private int in, out;

   private Semaphore mutex; //For mutual exclusion
   private Semaphore empty, full; //For coordination

   public BoundedBuffer() {
      count=0; in=0; out=0;
      mutex = new Semaphore(1);
      empty = new Semaphore(SIZE); // 5 free slots
      full = new Semaphore(0);     // 0 full slot
      buffer = (E[])new Object[SIZE];
   }
```

```
   public void insert(E item) {
      empty.acquire();
      mutex.acquire();
      buffer[in] = item;
      in = (in+1) % SIZE;
      mutex.release();
      full.release();
   }

   public E remove() { ... }

} // end of BoundedBuffer
```

```
// Full code
import java.util.Date;
import java.util.concurrent.Semaphore;

interface Buffer<E> {
        public void insert(E item) throws InterruptedException;
        public E remove() throws InterruptedException;
        public int getIn();
        public int getOut();
}

class BoundedBuffer<E> implements Buffer<E> {
        private static final int BUFFER_SIZE = 5;
        private E[] buffer;
        private int in, out;
        private Semaphore mutex, empty, full;

        public BoundedBuffer() {
                in = 0; out = 0;
                mutex = new Semaphore(1);                   // binary semaphore
                empty = new Semaphore(BUFFER_SIZE);         // initially, 5 free slots
                full = new Semaphore(0);                    // initially, 0 full slots

                buffer = (E[]) new Object[BUFFER_SIZE];
        }

        public int getIn() { return in; }
        public int getOut() { return out; }

        public void insert(E item) throws InterruptedException {
                empty.acquire();
                mutex.acquire();
                buffer[in] = item;
                in = (in + 1) % BUFFER_SIZE;
                mutex.release();
                full.release();
        }
        public E remove()  throws InterruptedException {
                E item;
                full.acquire();
                mutex.acquire();
                item = buffer[out];
                out = (out + 1) % BUFFER_SIZE;
                mutex.release();
                empty.release();

                return item;
        }
} //end of BoundedBuffer
```

```
class Producer implements Runnable {
        private Buffer<Date> buffer;

        public Producer(Buffer<Date> buffer) { this.buffer = buffer; }

        public void run() {
                int n = 0;
                while (true) {
                        SleepUtilities.nap();
                        try {
                                buffer.insert(new Date());
                        } catch (InterruptedException e) {
                        }
                        System.out.println("p::" + n++ + "::" + buffer.getIn());
                }
        }
}

class Consumer implements Runnable {
        private Buffer<Date> buffer;

        public Consumer(Buffer<Date> buffer) { this.buffer = buffer; }

        public void run() {
                int n = 0;
                while (true) {
                        SleepUtilities.nap();
                        try { System.out.println(n++ + "::" + buffer.getOut() + "::" + (Date)buffer.remove());
                        } catch (InterruptedException e) { /* ignore */ }
                }
        }
}

class SleepUtilities {
        private static final int NAP_TIME = 5;
        public static void nap() { nap(NAP_TIME); }
        public static void nap(int duration) {
                int sleeptime = (int) (NAP_TIME * Math.random() );
                try { Thread.sleep(sleeptime*1000); } catch (InterruptedException e) { /* ignore */ }
        }
}

public class Factory {
        public static void main(String[] args) {
                Buffer<Date> buffer = new BoundedBuffer<Date>();
                Thread producer = new Thread(new Producer(buffer));          producer.start();
                Thread consumer = new Thread(new Consumer(buffer));          consumer.start();
        }
}
```