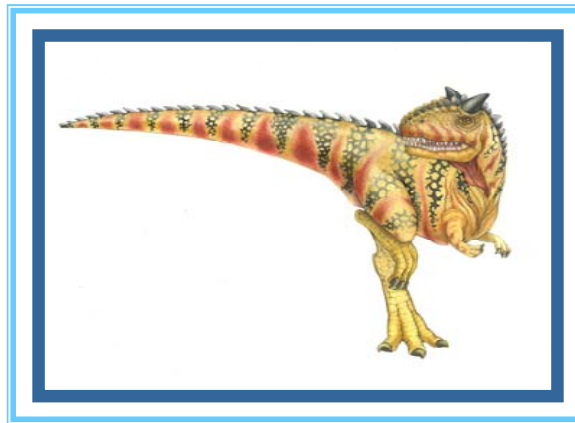


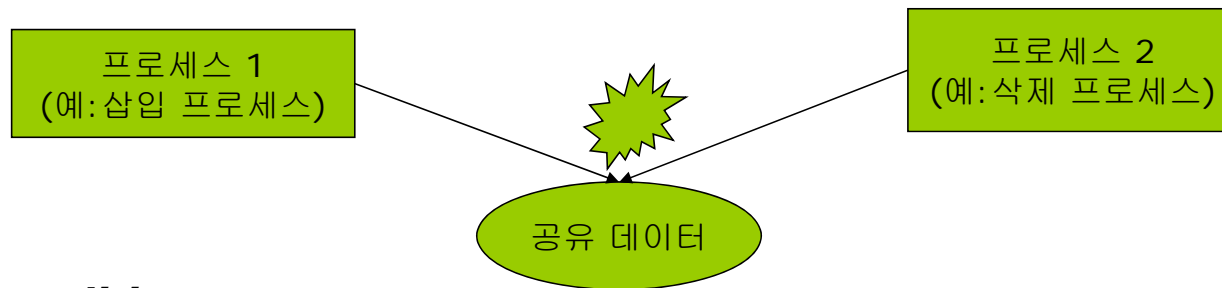
Chapter 6: Process Synchronization





Background

- **Concurrent access** to **shared data** may result in data **inconsistency**.

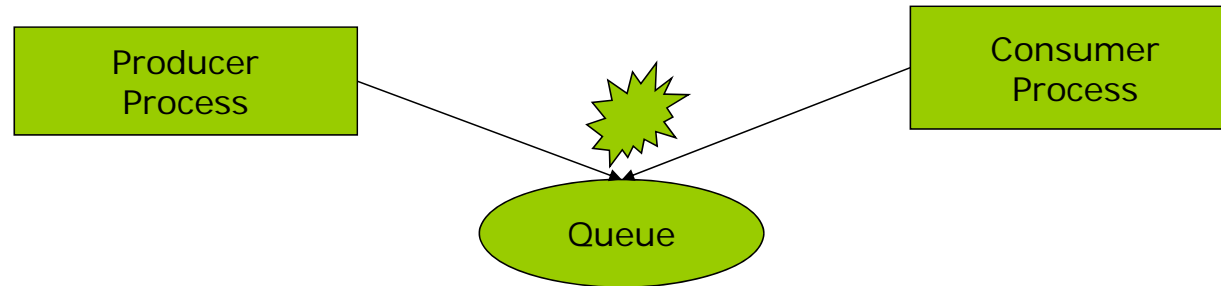


- **Race condition**
 - The situation where several processes access and manipulate shared data concurrently.
 - The final value of the shared data
 - depends upon which process finishes last.
- To prevent race conditions,
concurrent processes must be **synchronized**.





Background : Queue!



■ Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    ...
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```





Background : Queue!

■ Producer process

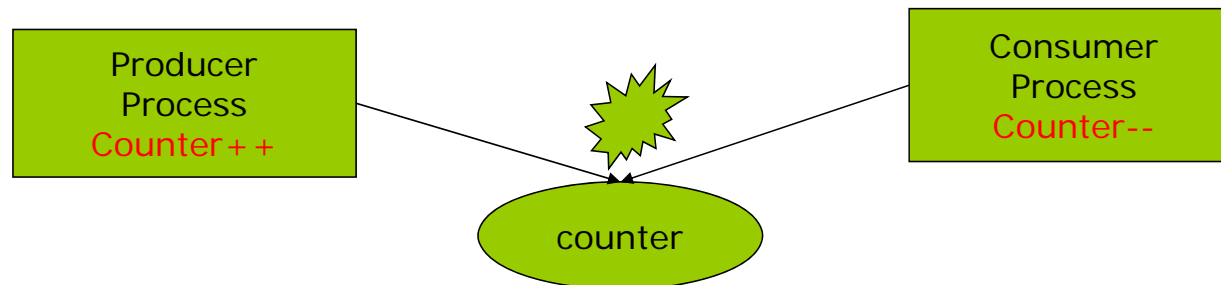
item nextProduced;

```
while (true) {  
    while (counter == BUFFER_SIZE)  
        ; /* do nothing */  
    buffer[in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```

■ Consumer process

item nextConsumed;

```
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
}
```





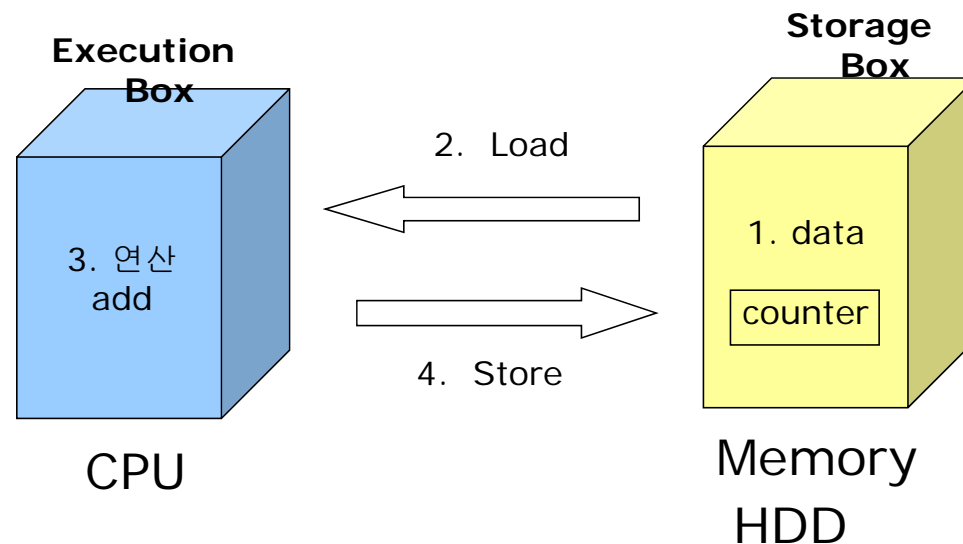
Bounded Buffer

- 문제점 :
counter++;
counter--;

Atomic operation means an operation that completes in its **entirety** without interruption.

Atomic한 연산으로 보이지만, 실제로는 **Atomic**하지 않다!

- **Counter++ is Not Atomic!**





Bounded Buffer

- **count++ could be implemented as**

register1 = count	(LOAD R1, COUNT)
register1 = register1 + 1	(ADD R1, 1)
count = register1	(STORE R1, COUNT)

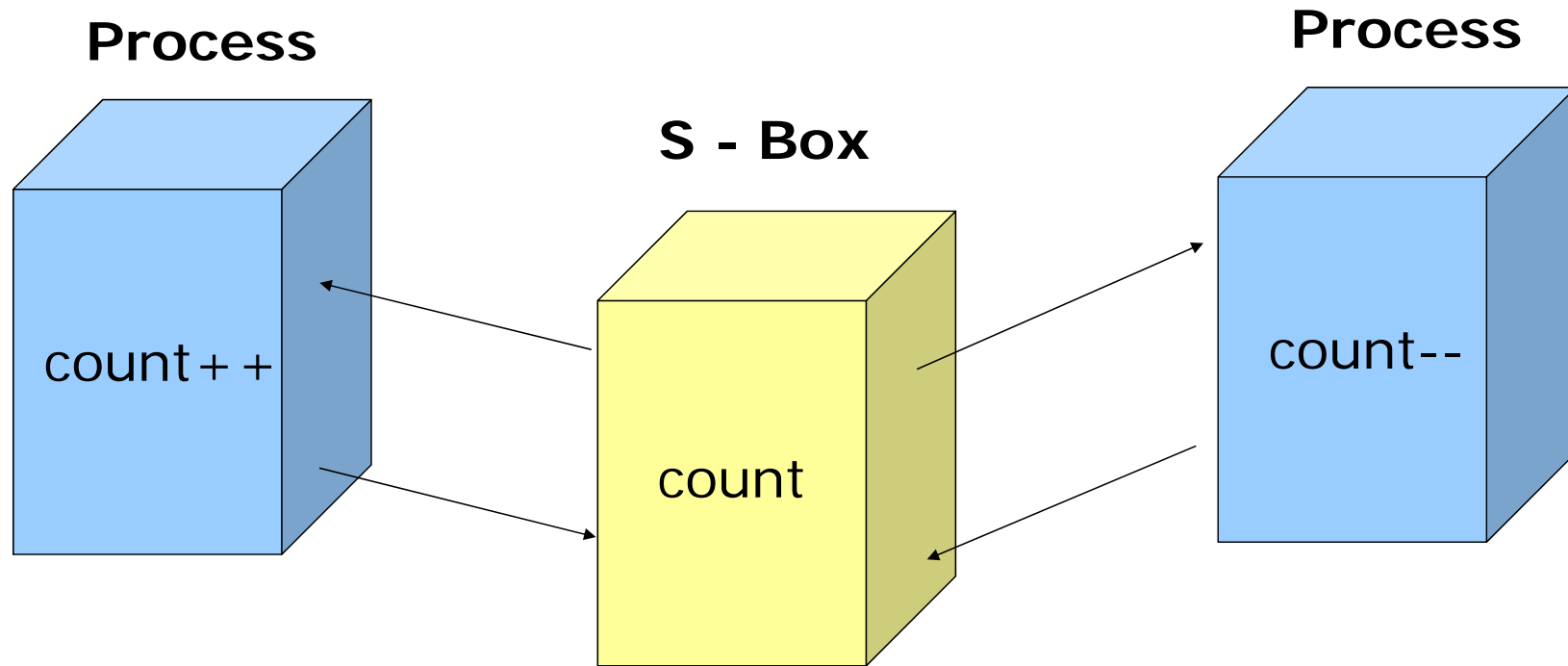
- **count-- could be implemented as**

register2 = count	(LOAD R2, COUNT)
register2 = register2 - 1	(SUB R2, 1)
count = register2	(STORE R2, COUNT)





Bounded Buffer : 동시 수행!!

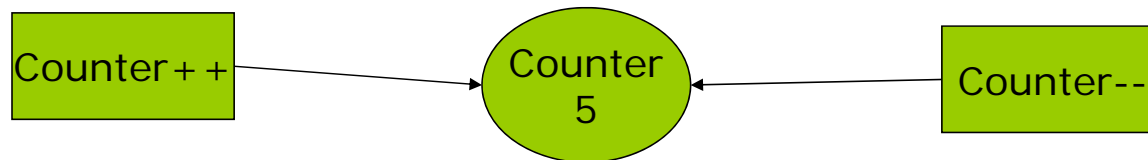




Bounded Buffer

- Consider this execution interleaving with “count = 5” initially:

S0: **producer** execute $\text{register1} = \text{count}$ {register1 = 5}
S1: **producer** execute $\text{register1} = \text{register1} + 1$ {register1 = 6}
S2: **consumer** execute $\text{register2} = \text{count}$ {register2 = 5}
S3: **consumer** execute $\text{register2} = \text{register2} - 1$ {register2 = 4}
S4: **producer** execute $\text{count} = \text{register1}$ {count = 6}
S5: **consumer** execute $\text{count} = \text{register2}$ {count = 4}



- The value of count may be either 4 or 6, where the correct result should be 5.





The Critical-Section Problem(임계구역문제)

- n processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.





Critical Section

- General structure of process p_i is

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (TRUE);
```

Figure 6.1 General structure of a typical process P_i .





Critical-Section 문제 해결의 충족조건

1. **Mutual Exclusion.** 한 프로세스가 임계 구역을 실행 중일 때, 다른 어떤 프로세스도 임계 구역을 실행할 수 없다
2. **Progress.** 임계 구역을 실행하는 프로세스가 없고 여러 개의 프로세스들이 임계 구역에 들어오고자 하는 상황에서는, 반드시 하나의 프로세스를 선택하여 진입시키는 올바른 결정 기법이 있어야 하고, 이러한 결정은 무한정 미루어져서는 안 된다.
3. **Bounded Waiting.** 한 프로세스가 임계 구역에 대한 진입 요청 후부터 요청의 수락까지의 기간 내에, 다른 프로세스가 임계 구역을 실행할 수 있는 회수에는 제한이 있어야 한다.
 - Assume that each process executes at a nonzero speed
 - No assumption concerning relative speed of the n processes.





SW 상호배제 1: Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
 - int **turn**;
 - Boolean **flag[n]**
- **Turn** : The variable **turn** indicates whose turn it is to enter the critical section.
- **Flag** : The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i]** = true implies that process **P_i** is ready!





SW 상호배제 1: Peterson's Solution

- Combined shared variables of algorithms 1 and 2.

- Process P_i

do {

flag [i] := true; /* My intention is to enter */

turn = j; /* Set to his turn-빠를수록 양보 */

while (flag [j] and turn = j) ; /* wait only if ... */

critical section

flag [i] = false;

remainder section

} while (1);

Provable that

1. Mutual exclusion is preserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

- Problems

- **Busy Waiting!** (계속 **CPU**와 **memory** 를 쓰면서 **wait**)
- **Software** 적인 해결책은 느낌





SW 상호배제 2: Bakery 알고리즘(일명 빵집)

- 분산처리 환경에서 유용한 알고리즘
- 자료구조
 - **choosing[]**과 **number[]** 변수들을 이용
 - 각 프로세스들은 번호표를 부여받고, 낮은 번호순으로 수행





```
// "declaration and initial values of global variables"
Entering: array [1..NUM_THREADS] of bool = {false};
Number: array [1..NUM_THREADS] of integer = {0};

1 lock(integer i) {
2   Entering[i] = true;
3   Number[i] = 1 + max(Number[1], ..., Number[NUM_THREADS]);
4   Entering[i] = false;
5   for (j = 1; j <= NUM_THREADS; j++) {
6     // "Wait until thread j receives its number":
7     while (Entering[j]) { /* nothing */ }
8     // "Wait until all threads with smaller numbers or with the same"
9     // "number, but with higher priority, finish their work":
10    while ((Number[j] != 0) && ((Number[j], j) < (Number[i], i))) {
11      /* nothing */
12    }
13  }
14 }
15
16 unlock(integer i) {
17   Number[i] = 0;
18 }
19
20 Thread(integer i) {
21   while (true) {
22     lock(i);
23     // "The critical section goes here..."
24     unlock(i);
25     // ""non-critical section..."
26   }
27 }
```





Synchronization Hardware

- Many systems provide **hardware support for critical section code**
- Uniprocessors – could disable interrupts
 - Currently running code would execute without preemption
 - Generally too inefficient on multiprocessor systems
 - ▶ Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
 - ▶ **Atomic = non-interruptable**
 - Either test memory word and set value
 - Or swap contents of two memory words





Critical Section Using Locks

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





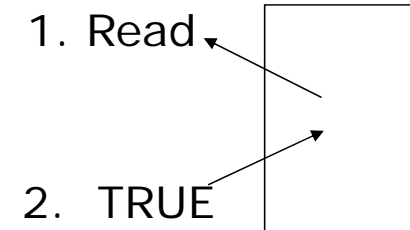
HW 상호배제 1: TestAndSet

상호배제 (Mutual Exclusion)의 구현

- Lock에 대한 **testAndSet**를 CPU의 명령어로 제공 (**atomic instruction**)
- Test and modify the content of a word atomically

```
boolean TestAndSet(boolean *target) {  
    boolean rv = *target;  
    *target = true;  
  
    return rv;  
}
```

TestAndSet(a)



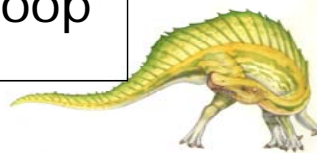
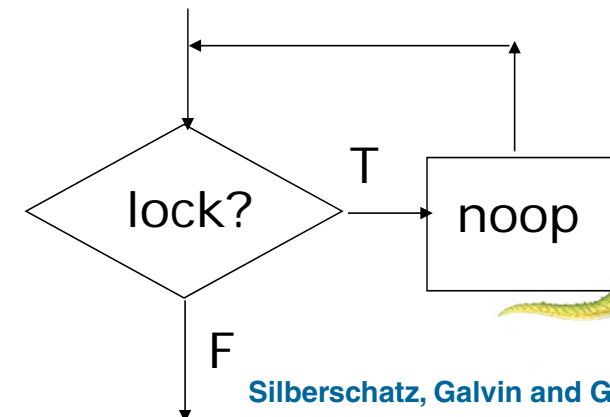


HW 상호배제 1: TestAndSet

상호배제 (Mutual Exclusion)의 구현

- Shared data:
boolean lock = false;
- Process P_i
do {
 while (TestAndSet(&lock)) ;
 critical section
 lock = false;
 remainder section
}

while(cond) do { };





HW 상호배제 2: Swap 이용

상호배제 (Mutual Exclusion)의 구현

- **Atomically swap two variables.**
 - **swap** 명령은 **CPU**에서 지원할 경우가 많음

```
void Swap(boolean *a, boolean *b) {  
    boolean temp = *a;  
    *a = *b;  
    *b = temp;  
}
```





HW 상호배제 2: Swap

상호배제 (Mutual Exclusion)의 구현

- Shared data (initialized to false):
 boolean lock = false ;
 boolean waiting[n];
- Process P_i
 do {
 key = true; /* My intention */
 while (key == true)
 Swap(&lock,&key);

critical section

lock = false;

remainder section

} while(true);

bounded waiting
문제는 어떻게 해결?





상호배제와 한정된 대기조건을 만족하는 lock

상호배제 (Mutual Exclusion) + 한정된 대기 (Bounded Waiting)

repeat

waiting[i] := true;

key := true;

while waiting[i] **and** key **do**

key := Test-and-Set(lock);

waiting[i] := false;

// 임계 구역

j := i+1 **mod** n;

while (j ≠ i) **and** (not waiting[j]) **do**

j := j+1 **mod** n;

if j = i **then** lock := false

else waiting[j] := false;

잔류 구역

until false;

process의 순서대로 lock을 줌

다른 프로세스가 요청 후 기다리고 있는지 차례로 검사

요청 후 기다리고 있는 프로세스가 하나도 없으면...

만약 있다면, **lock**을 풀지 않은 채로 대기 중인 프로세스를 임계 구역으로 진입시킴

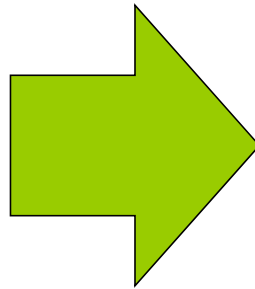




Synchronization in Java

■ keyword 'synchronized' in Java

```
class Counter {  
    private int c = 0;  
  
    public void increment() {  
        c++;  
    }  
  
    public void decrement() {  
        c--;  
    }  
  
    public int value() {  
        return c;  
    }  
}
```



```
public class SynchronizedCounter {  
    private int c = 0;  
  
    public synchronized void increment() {  
        c++;  
    }  
  
    public synchronized void decrement() {  
        c--;  
    }  
  
    public synchronized int value() {  
        return c;  
    }  
}
```





Synchronization in Java

■ Lock in Java

```
public class MsLunch {  
    private long c1 = 0;  
    private long c2 = 0;  
    private Object lock1 = new Object();  
    private Object lock2 = new Object();  
  
    public void inc1() {  
        synchronized(lock1) {  
            c1++;  
        }  
    }  
  
    public void inc2() {  
        synchronized(lock2) {  
            c2++;  
        }  
    }  
}
```





Semaphores(세마포어)

- 소프트웨어 해결 및 **Test-and-Set** 등은 모두 “**busy waiting**” 알고리즘
- 임계 구역 진입 시 이미 다른 프로세스가 진입해 있으면 **busy-waiting loop** 실행 -> 타임 슬라이스 낭비
- 세마포어 (**Dijkstra**) : **block/wakeup** 알고리즘
 - 진입 불가능 시에는 대기 상태로 전환
 - 임계 구역을 진출하는 프로세스가 대기 프로세스를 준비 상태로 깨워줌





Semaphores

- Synchronization tool that does not require busy waiting.
- Semaphore S – integer variable
- can only be accessed via two indivisible (atomic) operations

wait (S):

*while $S \leq 0$ do no-op;
 $S--$;*

signal (S):

$S++$;





Semaphore as General Synchronization Tool

■ 세마포어의 종류

- **Counting** semaphore – integer value can range over an unrestricted domain
- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement
 - ▶ Also known as **mutex locks**

■ Provides mutual exclusion

```
Semaphore mutex; // initialized to 1
do {
    wait (mutex);
    // Critical Section
    signal (mutex);
    // remainder section
} while (TRUE);
```

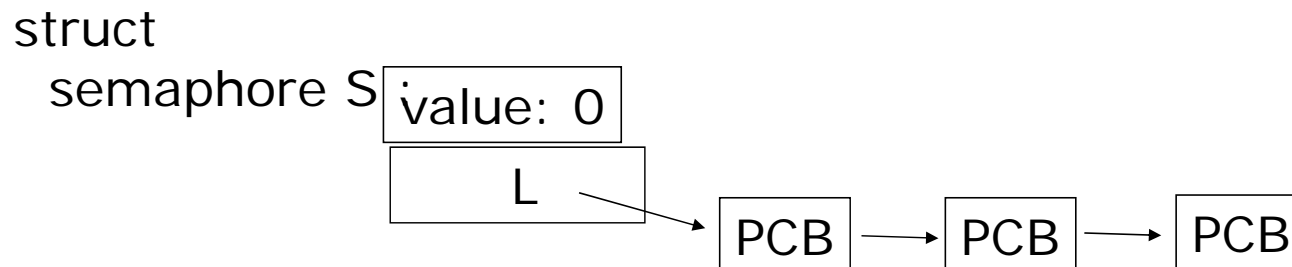




Semaphore Implementation

- Define a semaphore as a record

```
typedef struct {  
    int value;  
    struct process *L;  
} semaphore;
```
- Assume two simple operations:
 - **block** kernel suspends the process that invoked P(S) itself.
 Put this process' PCB into wait queue (semaphore)
 - **wakeup(P)** V(S) resumes the execution of a blocked process P.
 (Put this process' PCB into ready queue)





Implementation

- Semaphore operations now defined as

wait(semaphore *S):

S->value--;

if (S->value < 0) {

add this process to S->L;

block();

}

signal(semaphore *S):

S->value++;

if (S->value <= 0) {

remove a process P from S->L;

wakeup(P);

}





Semaphore in Java

- **Java SE5, SE6**에서 기본 지원
 - url :
<http://java.sun.com/javase/6/docs/api/java/util/concurrent/Semaphore.html>

```
Semaphore S = new Semaphore();  
  
S.acquire();  
  
    // critical section  
  
S.release();  
  
    // remainder section
```





Semaphore in Java : example

```
public class Worker implements Runnable
{
    private Semaphore sem;
    private String name;

    public Worker(Semaphore sem, String name) {
        this.sem = sem;
        this.name = name;
    }

    public void run() {
        while (true) {
            sem.acquire();
            MutualExclusionUtilities.criticalSection(name);
            sem.release();
            MutualExclusionUtilitiesremainderSection(name);
        }
    }
}
```





Semaphore in Java : example

```
public class SemaphoreFactory
{
    public static void main(String args[]) {
        Semaphore sem = new Semaphore(1);
        Thread[] bees = new Thread[5];

        for (int i = 0; i < 5; i++)
            bees[i] = new Thread(new Worker
                (sem, "Worker " + (new Integer(i)).toString() ));

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





Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let **S** and **Q** be two semaphores initialized to 1

P_0
wait (S);
wait (Q);
.
.
.
signal (S);
signal (Q);

P_1
wait (Q);
wait (S);
.
.
.
signal (Q);
signal (S);

- **Starvation** – indefinite blocking
 - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
 - Solved via **priority-inheritance protocol**





Semaphore as a General Synchronization Tool

프로세스 Sync를 위해 Semaphore 사용하기

- Execute B in P_j only after A executed in P_i
- Use semaphore $flag$ initialized to 0
- Code:

P_i	P_j
\vdots	\vdots
A	$wait(flag)$
$signal(flag)$	B





Classical Problems of Synchronization

- **Classical problems used to test newly-proposed synchronization schemes**
 - **Bounded-Buffer Problem**
 - **Readers and Writers Problem**
 - **Dining-Philosophers Problem**





Bounded-Buffer Problem

- N buffers, each can hold one item
- Semaphore **mutex** initialized to the value 1
- Semaphore **full** initialized to the value 0
- Semaphore **empty** initialized to the value N





Bounded-Buffer Problem

New data arrived

Any empty buf?
Fill it
Produce full buf

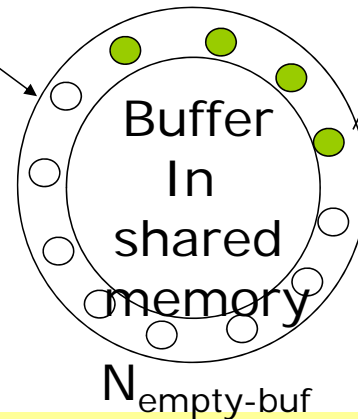
If yes, but ...

Can I access
shared variable now?

Producer
in

$N_{\text{full-buf}}$

Consumer
out



Any full buf exist?
Get it
Produce empty buf

If yes, but ...

Can I access
shared variable now?

Shared variable: buf, count ==> Need binary semaphore

Resource count: # of full buf ==> Need integer semaphore
of empty buf





Bounded-Buffer Problem

- 생산자-소비자 문제

- Shared data

semaphore full, empty, mutex;

Initially:

full = 0, empty = n, mutex = 1





Bounded-Buffer Problem: Producer Process

Producer Process

Before

■ Producer process

item nextProduced;

```
while (true) {  
    while (counter == BUFFER_SIZE)  
        ; /* do nothing */  
    buffer[in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```

After using Semaphore

```
do {  
    ...  
    produce an item in nextp  
    ...  
    wait(empty);  
    wait(mutex);  
    ...  
    add nextp to buffer  
    ...  
    signal(mutex);  
    signal(full);  
} while (1);
```





Bounded-Buffer Problem : Consumer Process

Consumer Process

Before

```
item nextConsumed;  
  
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
}
```

After using Semaphore

```
do {  
    wait(full)  
    wait(mutex);  
    ...  
    remove an item from buffer to nextc  
    ...  
    signal(mutex);  
    signal(empty);  
    ...  
    consume the item in nextc  
    ...  
} while (1);
```





Bounded-Buffer : Java 구현 예

```
public class BoundedBuffer implements Buffer
{
    private static final int BUFFER_SIZE = 5;
    private Object[] buffer;
    private int in, out;
    private Semaphore mutex;
    private Semaphore empty;
    private Semaphore full;

    public BoundedBuffer() {
        // buffer is initially empty
        in = 0;
        out = 0;
        buffer = new Object[BUFFER_SIZE];

        mutex = new Semaphore(1);
        empty = new Semaphore(BUFFER_SIZE);
        full = new Semaphore(0);
    }

    public void insert(Object item) {
        // Figure 6.9
    }

    public Object remove() {
        // Figure 6.10
    }
}
```





Bounded-Buffer : Java 구현 예

insert() Method

```
public void insert(Object item) {  
    empty.acquire();  
    mutex.acquire();  
  
    // add an item to the buffer  
    buffer[in] = item;  
    in = (in + 1) % BUFFER_SIZE;  
  
    mutex.release();  
    full.release();  
}
```





Bounded-Buffer : Java 구현 예

remove() Method

```
public Object remove() {  
    full.acquire();  
    mutex.acquire();  
  
    // remove an item from the buffer  
    Object item = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
  
    mutex.release();  
    empty.release();  
  
    return item;  
}
```

--





Bounded-Buffer : Java 구현 예

■ The structure of the producer process

```
public class Producer implements Runnable
{
    private Buffer buffer;

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

    public void run() {
        Date message;

        while (true) {
            // nap for awhile
            SleepUtilities.nap();
            // produce an item & enter it into the buffer
            message = new Date();
            buffer.insert(message);
        }
    }
}
```





Bounded-Buffer : Java 구현 예

■ The structure of the consumer process

```
public class Consumer implements Runnable
{
    private Buffer buffer;

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

    public void run() {
        Date message;

        while (true) {
            // nap for awhile
            SleepUtilities.nap();
            // consume an item from the buffer
            message = (Date)buffer.remove();
        }
    }
}
```





Bounded-Buffer : Java 구현 예

■ The Factory

```
public class Factory
{
    public static void main(String args[]) {
        Buffer buffer = new BoundedBuffer();

        // now create the producer and consumer threads
        Thread producer = new Thread(new Producer(buffer));
        Thread consumer = new Thread(new Consumer(buffer));

        producer.start();
        consumer.start();
    }
}
```





Bounded-Buffer : Java 구현 예

■ ConcurrentLinkedQueue in Java

```
import java.util.concurrent.*;

public class ConcurrentLinkedQueueExample {
    public static void main(String[] args) {
        ConcurrentLinkedQueue<String> queue = new ConcurrentLinkedQueue<String>();
        Thread producer = new Thread(new Producer(queue));
        Thread consumer = new Thread(new Consumer(queue));
        producer.start();
        consumer.start();
    }
}
```

```
// the consumer removes strings from the queue
class Consumer implements Runnable {

    ConcurrentLinkedQueue<String> queue;
    Consumer(ConcurrentLinkedQueue<String> queue){
        this.queue = queue;
    }
    public void run() {
        String str;
        System.out.println("Consumer Started");
        for (int x = 0; x < 10; x++) {
            while ((str = queue.poll()) != null) {
                System.out.println("Removed: " + str);
            }
            try {
                Thread.currentThread().sleep(500);
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

// the consumer removes strings from the queue
class Consumer implements Runnable {

```
    ConcurrentLinkedQueue<String> queue;
    Consumer(ConcurrentLinkedQueue<String> queue){
        this.queue = queue;
    }
    public void run() {
        String str;
        System.out.println("Consumer Started");
        for (int x = 0; x < 10; x++) {
            while ((str = queue.poll()) != null) {
                System.out.println("Removed: " + str);
            }
            try {
                Thread.currentThread().sleep(500);
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

ArrayBlockingQueue
PriorityBlockingQueue





Readers-Writers Problem

- **Shared data**

semaphore mutex, wrt;

Initially

mutex = 1, wrt = 1, readcount = 0





Readers-Writers Problem Writer Process

wait(wrt);

...

writing is performed

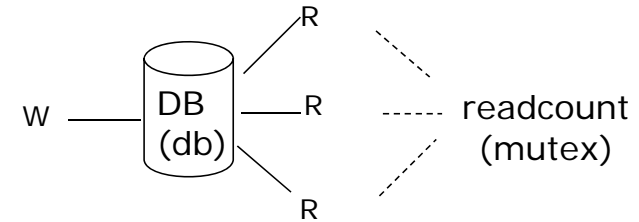
...

signal(wrt);





Readers-Writers Problem Reader Process



```
wait(mutex);  
readcount++;  
if (readcount == 1)  
    wait(wrt);  
signal(mutex);
```

다른 프로세스가 쓰는 중이면
기다림

읽기 카운트를
증가 시키는중에
다른 프로세스가
Readcount를 증가시키지
못하도록 함

...
reading is performed
...

```
wait(mutex);  
readcount--;  
if (readcount == 0)  
    signal(wrt);  
signal(mutex);
```





Readers-Writers Problem : Java 구현 예

Interface for read-write locks

```
public interface RWLock
{
    public abstract void acquireReadLock();
    public abstract void acquireWriteLock();
    public abstract void releaseReadLock();
    public abstract void releaseWriteLock();
}
```





Readers-Writers Problem : Java 구현 예

Methods called by writers.

```
public void acquireWriteLock() {  
    db.acquire();  
}  
  
public void releaseWriteLock() {  
    db.release();  
}
```





Readers-Writers Problem : Java 구현 예

■ The structure of a writer process

```
public class Writer implements Runnable
{
    private RWLock db;

    public Writer(RWLock db) {
        this.db = db;
    }

    public void run() {
        while (true) {
            // nap for awhile
            SleepUtilities.nap();

            db.acquireWriteLock();

            // you have access to write to the database
            SleepUtilities.nap();

            db.releaseWriteLock();
        }
    }
}
```





Readers-Writers Problem : Java 구현 예

■ The structure of a reader process

```
public class Reader implements Runnable
{
    private RWLock db;

    public Reader(RWLock db) {
        this.db = db;
    }

    public void run() {
        while (true) {
            // nap for awhile
            SleepUtilities.nap();

            db.acquireReadLock();

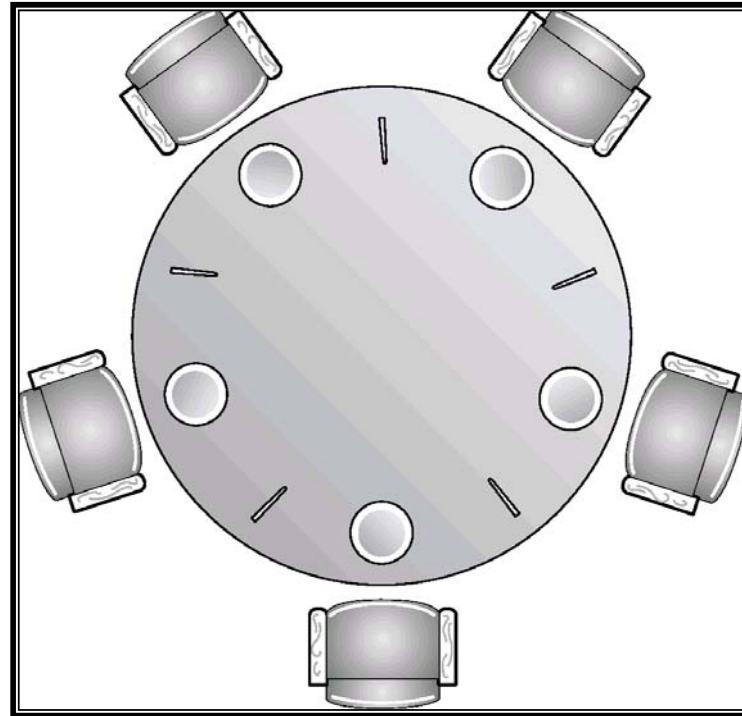
            // you have access to read from the database
            SleepUtilities.nap();

            db.releaseReadLock();
        }
    }
}
```





Dining-Philosophers Problem



- Shared data
semaphore chopstick[5];
Initially all values are 1





Dining-Philosophers Problem

■ Philosopher i :

```
do {  
    wait(chopstick[i])  
    wait(chopstick[(i+1) % 5])  
    ...  
    eat  
    ...  
    signal(chopstick[i]);  
    signal(chopstick[(i+1) % 5]);  
    ...  
    think  
    ...  
} while (1);
```

왼쪽 숟가락과
오른쪽 숟가락을
모두 확보하면
Critical section In!

왼쪽 숟가락과
오른쪽 숟가락을
모두 반납!





Problems with Semaphores

- **Incorrect use of semaphore operations:**
 - **signal (mutex) wait (mutex)**
 - **wait (mutex) ... wait (mutex)**
 - **Omitting of wait (mutex) or signal (mutex) (or both)**
- **Deadlock and starvation**





Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

```
monitor monitor-name
{
    shared variable declarations
    procedure body P1 (...) {
        . . .
    }
    procedure body P2 (...) {
        . . .
    }
    procedure body Pn (...) {
        . . .
    }
    {
        initialization code
    }
}
```





Monitors

■ 모니터의 특징

- 모니터의 공유자원을 사용하려면 반드시 모니터의 진입부를 호출해야함
- 모니터의 외부 프로세스는 모니터내부의 데이터를 직접 접근할 수 없음
- 모니터의 경계에서 상호배제가 시행됨
- 한순간에 하나의 프로세스만 진입을 허용
- 세마포의 **wait/signal**에 의해 구현될수 있음



Monitors

- To allow a process to wait within the monitor, a condition variable must be declared, as

`condition x, y;`
- Condition variable can only be used with the operations `wait` and `signal`.

condition x, y;

- **The operation**

x.wait();

means that the process invoking this operation is suspended until another process invokes

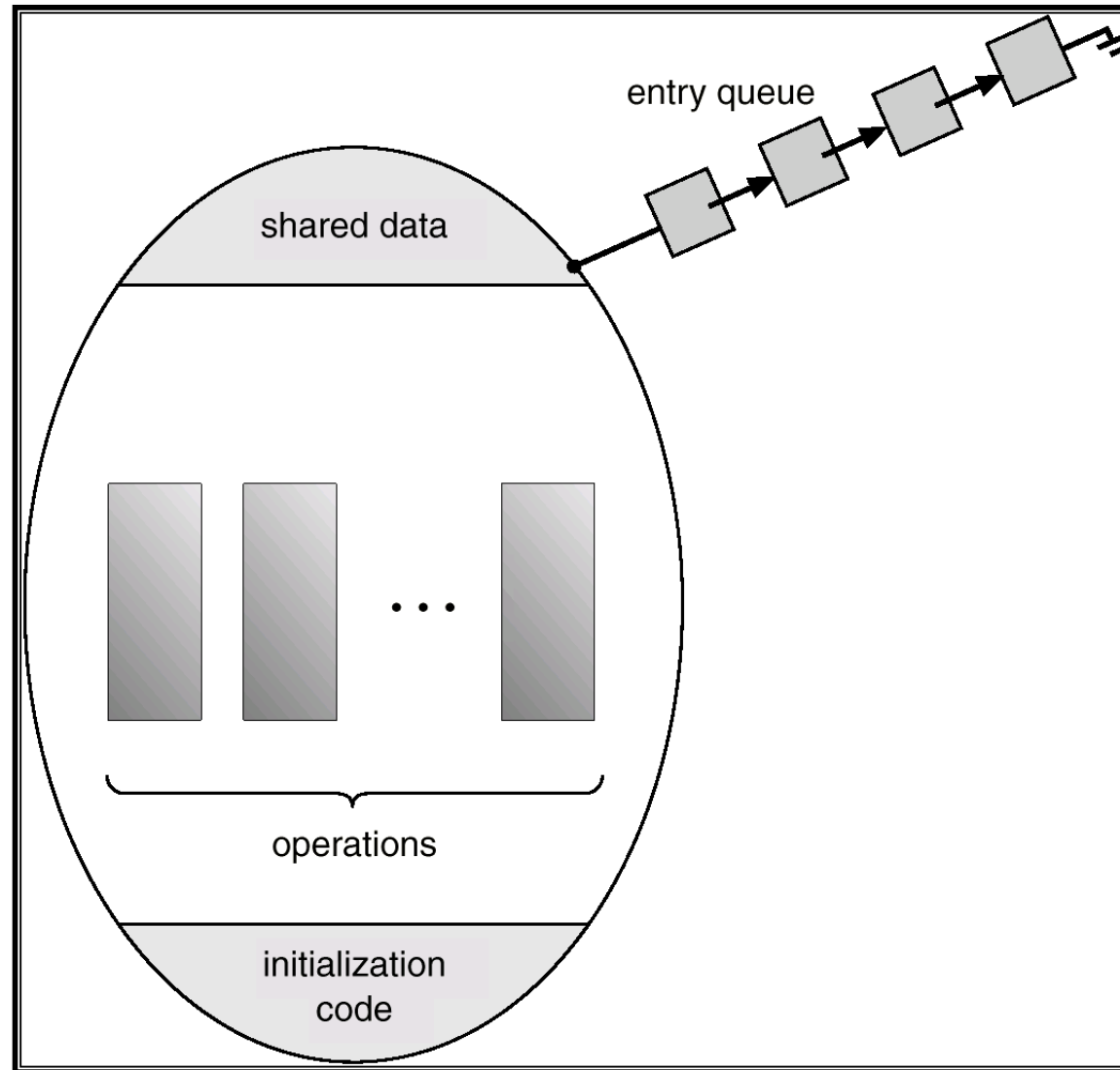
x.signal();

- **The x.signal operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect.**



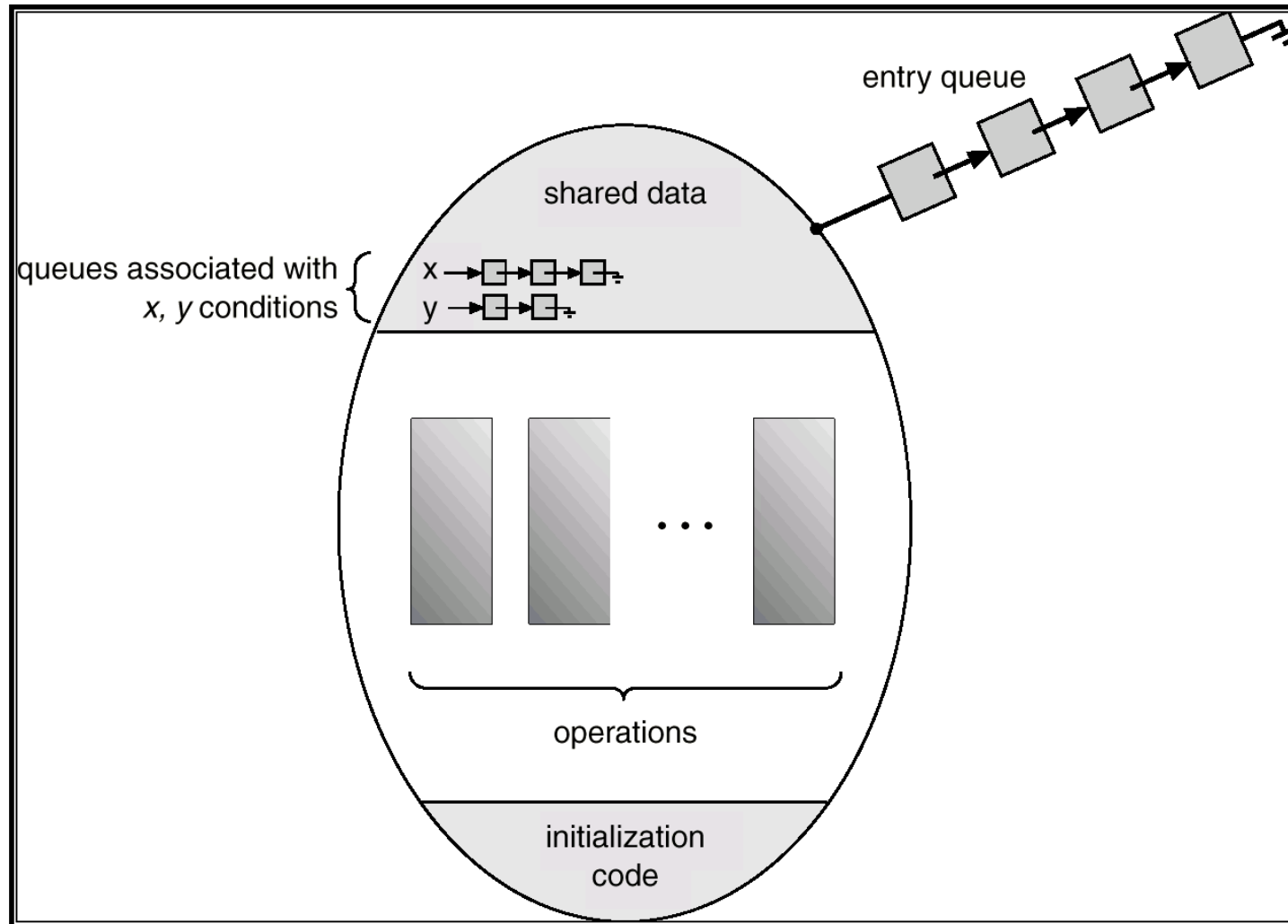


Schematic View of a Monitor





Monitor With Condition Variables





Dining Philosophers Example

```
monitor dp
{
    enum {thinking, hungry, eating} state[5];
    condition self[5];
    void pickup(int i)           // following slides
    void putdown(int i) // following slides
    void test(int i)             // following slides
    void init() {
        for (int i = 0; i < 5; i++)
            state[i] = thinking;
    }
}
```





Dining Philosophers

```
void pickup(int i) {  
    state[i] = hungry;  
    test[i];  
    if (state[i] != eating)  
        self[i].wait();  
}  
  
void putdown(int i) {  
    state[i] = thinking;  
    // test left and right neighbors  
    test((i+4) % 5);  
    test((i+1) % 5);  
}
```





Dining Philosophers

```
void test(int i) {  
    if ( (state[(i + 4) % 5] != eating) &&  
        (state[i] == hungry) &&  
        (state[(i + 1) % 5] != eating)) {  
        state[i] = eating;  
        self[i].signal();  
    }  
}
```





Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex; // (initially = 1)
semaphore next;  // (initially = 0)
int next-count = 0;
```

- Each external procedure F will be replaced by
wait(mutex);

...

body of F ;

...

if (next-count > 0)

signal(next)

else

signal(mutex);

- Mutual exclusion within a monitor is ensured.





Monitor Implementation

- For each condition variable x , we have:
 semaphore $x\text{-sem}$; // (initially = 0)
 int $x\text{-count}$ = 0;
- The operation $x.\text{wait}$ can be implemented as:

```
x-count++;  
if (next-count > 0)  
    signal(next);  
else  
    signal(mutex);  
wait(x-sem);  
x-count--;
```





Monitor Implementation

- The operation `x.signal` can be implemented as:

```
if (x-count > 0) {  
    next-count++;  
    signal(x-sem);  
    wait(next);  
    next-count--;  
}
```





Monitor Implementation

- ***Conditional-wait* construct: `x.wait(c);`**
 - `c` – integer expression evaluated when the wait operation is executed.
 - value of `c` (a *priority number*) stored with the name of the process that is suspended.
 - when `x.signal` is executed, process with smallest associated priority number is resumed next.
- **Check two conditions to establish correctness of system:**
 - User processes must always make their calls on the monitor in a correct sequence.
 - Must ensure that an uncooperative process does not ignore the mutual-exclusion gateway provided by the monitor, and try to access the shared resource directly, without using the access protocols.

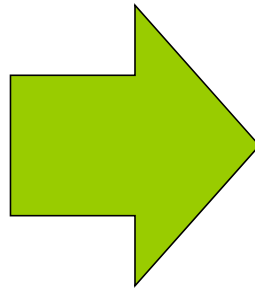




Monitor in Java

■ keyword 'synchronized' in Java

```
class Counter {  
    private int c = 0;  
  
    public void increment() {  
        c++;  
    }  
  
    public void decrement() {  
        c--;  
    }  
  
    public int value() {  
        return c;  
    }  
}
```



```
public class SynchronizedCounter {  
    private int c = 0;  
  
    public synchronized void increment() {  
        c++;  
    }  
  
    public synchronized void decrement() {  
        c--;  
    }  
  
    public synchronized int value() {  
        return c;  
    }  
}
```





Java 5.0 ReadWriteLock

```
public class SyncTest {
    List<String> values = new ArrayList<String>(); ← 공유 데이터

    public void test() {
        Thread writeThread = new Thread() {
            public void run() {
                while(true) {
                    for(int i = 0; i < 10; i++) {
                        add("Val" + i);
                    }
                    try {
                        Thread.sleep(50);
                    } catch (InterruptedException e) {}
                }
            }
        };

        Thread readThread = new Thread() {
            public void run() {
                while(true) {
                    print();
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        };

        writeThread.start();
        readThread.start();

        public void add(String value) {
            values.add(value);
        }

        public void print() {
            for(String value: values) {
                System.out.println(value);
            }
        }

        public static void main(String[] args) {
            (new SyncTest()).test();
        }
    }
};
```

공유 데이터를 이용하는 스레드

Synchronized 를 사용하지 않을 경우 문제를 발생시키는 예





Java 5.0 ReadWriteLock

■ **Synchronized**를 이용한 동기화

```
public synchronized void add(String value) {  
    values.add(value);  
}
```

```
public synchronized void print() {  
    for(String value: values) {  
        System.out.println(value);  
    }  
}
```

- **Synchronized**를 이용하여 메소드를 배타적으로 수행함
- 배타적으로 수행하므로 **Once Write Many Read** 의 경우 성능에 영향을 미칠수 있음





Java 5.0 ReadWriteLock

■ ReadWriteLock을 이용한 Synchronization

```
List<String> values = new ArrayList<String>();  
final ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
```

```
public void add(String value) {  
    lock.writeLock().lock();  
    try {  
        values.add(value);  
    } finally {  
        lock.writeLock().unlock();  
    }  
}
```

```
public void print() {  
    lock.readLock().lock();  
    try {  
        for(String value: values) {  
            System.out.println(value);  
        }  
    } finally {  
        lock.readLock().unlock();  
    }  
}
```

Deadlock의 문제점 내포





Java 5.0 ReadWriteLock

- **ReadWriteLock**에서 **Deadlock**의 회피
 - 일정시간동안 **lock**을 얻지 못하면 **Exception** 발생

```
public void print() {  
    try {  
        if(lock.readLock().tryLock(100, TimeUnit.MILLISECONDS)) {  
            try {  
                for(String value: values) {  
                    System.out.println(value);  
                }  
            } finally {  
                lock.readLock().unlock();  
            }  
        } else {  
            System.out.println("Lock Timeout");  
        }  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}
```





Semaphore in Linux/gcc

```
#include <unistd.h> /* Symbolic Constants */
#include <sys/types.h> /* Primitive System Data Types */
#include <errno.h> /* Errors */
#include <stdio.h> /* Input/Output */
#include <stdlib.h> /* General Utilities */
#include <pthread.h> /* POSIX Threads */
#include <string.h> /* String handling */
#include <semaphore.h> /* Semaphore */
```

```
/* prototype for thread routine */
void handler ( void *ptr );
```

```
void handler ( void *ptr )
{
    int x;
    x = *((int *) ptr);
    printf("Thread %d: Waiting to enter critical region...\n", x);
    sem_wait(&mutex); /* down semaphore */
    /* START CRITICAL REGION */
    printf("Thread %d: Now in critical region...\n", x);
    printf("Thread %d: Counter Value: %d\n", x, counter);
    printf("Thread %d: Incrementing Counter...\n", x);
    counter++;
    printf("Thread %d: New Counter Value: %d\n", x, counter);
    printf("Thread %d: Exiting critical region...\n", x);
    /* END CRITICAL REGION */
    sem_post(&mutex); /* up semaphore */

    pthread_exit(0); /* exit thread */
}
```

```
/* semaphores are declared global so they can be accessed
   in main() and in thread routine,
   here, the semaphore is used as a mutex */
sem_t mutex;
int counter; /* shared variable */
```

```
int main()
{
    int i[2];
    pthread_t thread_a;
    pthread_t thread_b;

    i[0] = 0; /* argument to threads */
    i[1] = 1;
```

```
    sem_init(&mutex, 0, 1); /* initialize mutex to 1 - binary semaphore */
                             /* second param = 0 - semaphore is local */
```

```
    /* Note: you can check if thread has been successfully created by checking return value of
       pthread_create */
    pthread_create (&thread_a, NULL, (void *) &handler, (void *) &i[0]);
    pthread_create (&thread_b, NULL, (void *) &handler, (void *) &i[1]);
```

```
    pthread_join(thread_a, NULL);
    pthread_join(thread_b, NULL);
```

```
    sem_destroy(&mutex); /* destroy semaphore */
```

```
    /* exit */
    exit(0);
} /* main() */
```

<http://www.amparo.net/ce155/sem-ex.html>





Semaphore in Windows/VC++

- 너무 길어서..
 - 다음 **URL**에서 <http://blog.naver.com/dkdaf/90157773148>



End of Chapter 6

