

# Chapter 6: Process Synchronization

---



# Module 6: Process Synchronization

---

- ❑ Background
- ❑ The Critical-Section Problem
- ❑ Peterson's Solution
- ❑ Synchronization Hardware
- ❑ Semaphores
- ❑ Classic Problems of Synchronization
- ❑ Monitors
- ❑ Synchronization Examples
- ❑ Atomic Transactions



# Background

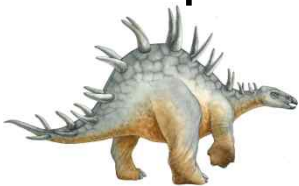
- ❑ *Concurrent access* to *shared data* may result in data *inconsistency*.



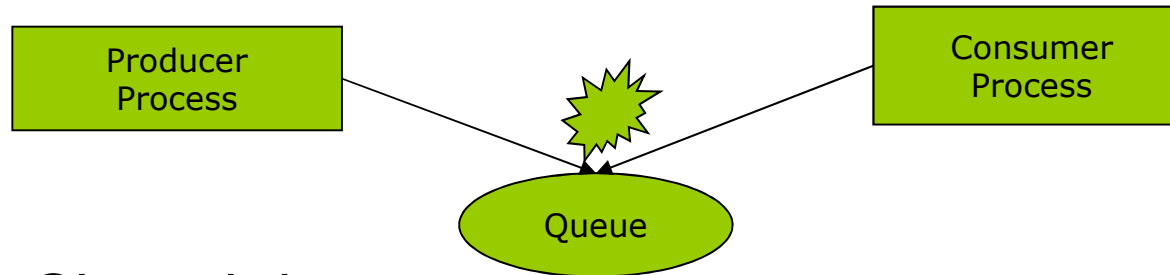
- ❑ **Race condition**

- 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*.

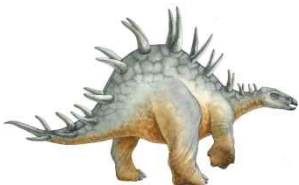


# Background : Queue!



□ Shared data

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



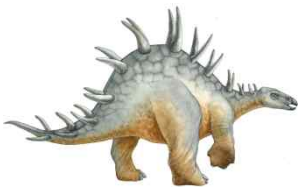
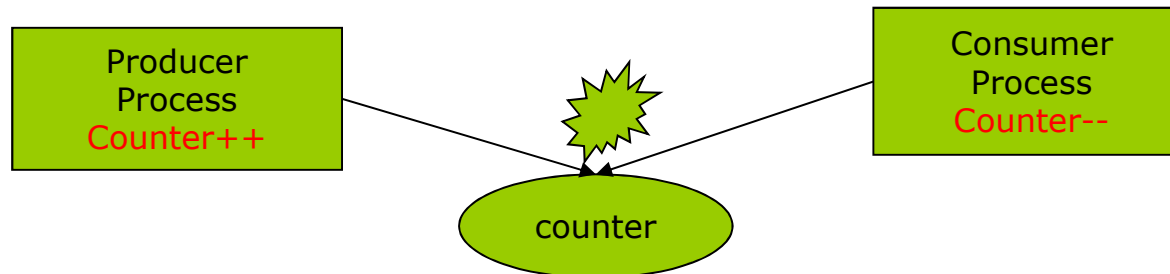
# Background : Queue!

## □ Producer process

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

## ■ Consumer process

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



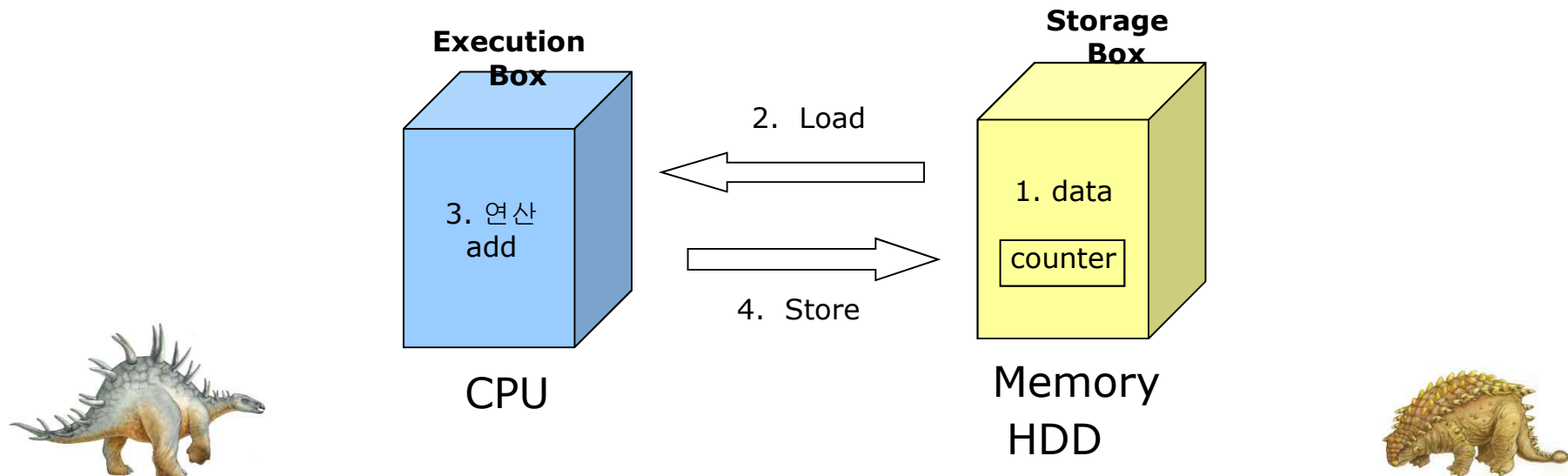
# Bounded Buffer

- 문제점 :  
**counter++;**  
**counter--;**

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

Atomic한 연산으로 보이지만, 실제로는 Atomic하지 않다!

- Counter++ is Not Atomic!



# Bounded Buffer

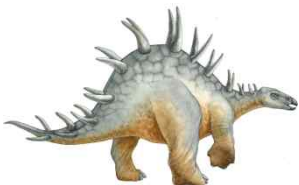
---

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

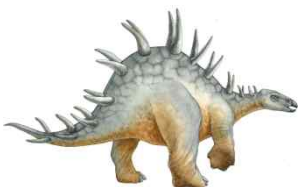
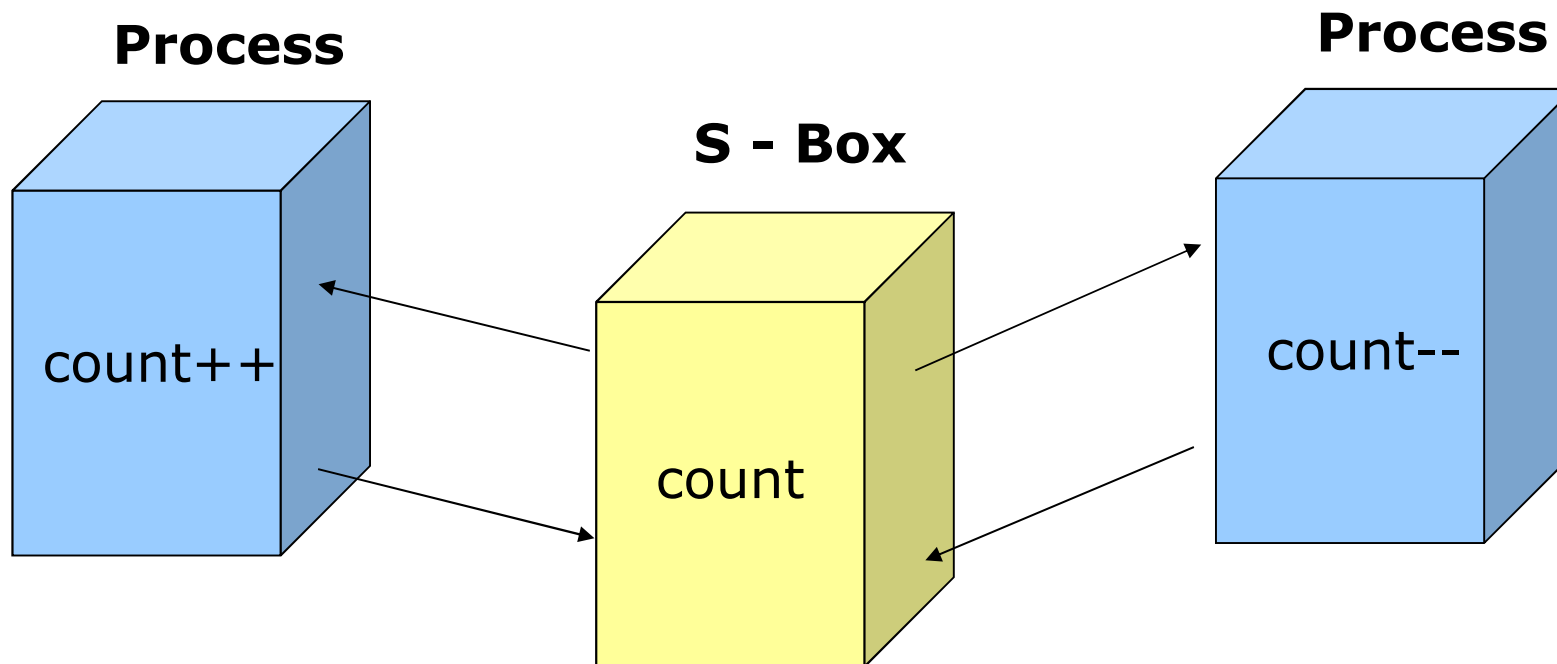
<b>register1 = count</b>	<b>(LOAD R1, COUNT)</b>
<b>register1 = register1 + 1</b>	<b>(ADD R1, 1)</b>
<b>count = register1</b>	<b>(STORE R1, COUNT)</b>

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

<b>register2 = count</b>	<b>(LOAD R2, COUNT)</b>
<b>register2 = register2 - 1</b>	<b>(SUB R2, 1)</b>
<b>count = register2</b>	<b>(STORE R2, COUNT)</b>



# Bounded Buffer : 동시 수행!!

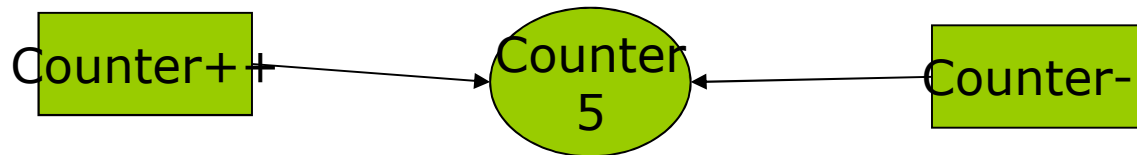




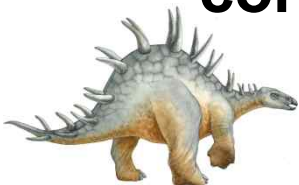
# Bounded Buffer

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

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

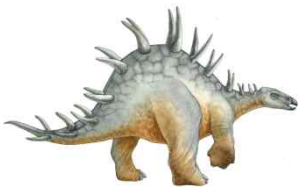
---

- $n$  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 문제 해결의 충족조건

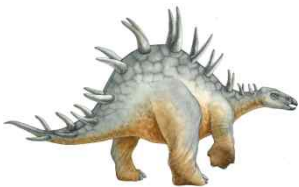
1. **Mutual Exclusion.** 한 프로세스가 임계 구역을 실행 중일 때, 다른 어떤 프로세스도 임계 구역을 실행할 수 없다
2. **Progress.** 임계 구역을 실행하는 프로세스가 없고 여러 개의 프로세스들이 임계 구역에 들어오고자 하는 상황에서는, 반드시 하나의 프로세스를 선택하여 진입시키는 올바른 결정 기법이 있어야 하고, 이러한 결정은 무한정 미루어져서는 안 된다.
3. **Bounded Waiting.** 한 프로세스가 임계 구역에 대한 진입 요청 후부터 요청의 수락까지의 기간 내에, 다른 프로세스가 임계 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.



# Critical-Section Problem

---

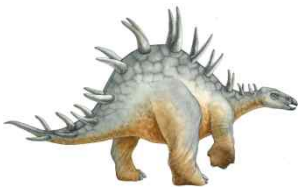
1. **Race Condition** - When there is concurrent access to shared data and the final outcome depends upon order of execution.
2. **Critical Section** - Section of code where shared data is accessed.
3. **Entry Section** - Code that requests permission to enter its critical section.
4. **Exit Section** - Code that is run after exiting the critical section



# Structure of a Typical Process

---

```
while (true) {  
    entry section  
    critical section  
    exit section  
    remainder section  
}
```



# 소프트웨어기반 상호배제 구현방법

- 두개의 프로세스간 상호배제
  - 데커(Dekker) 알고리즘
    - 상호배제의 최초 소프트웨어 해결법
  - Peterson's 알고리즘
    - Dekker 알고리즘의 향상
- 다수의 프로세스간 상호배제
  - Lamport의 Bakery(제과점, 빵집) 알고리즘
    - 분산처리 환경에 유용한 알고리즘
    - 부여받은 번호에 의해 수행



# 소프트웨어 상호배제 : Dekker 알고리즘

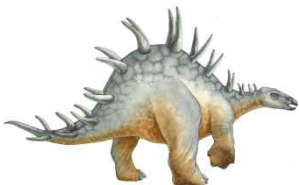
- 데커(Dekker) 알고리즘
  - 상호배제의 최초 소프트웨어 해결법
    - flag, turn 2개의 공유변수 사용

```
P0 :
flag[0] = true;
while (flag[1] == true) {
    if (turn ≠ 0) {
        flag[0] = false;
        while (turn ≠ 0) { }
        flag[0] = true;
    }
}
```

// critical section

...

```
turn  = 1;
flag[0] = false;
// remainder section
```



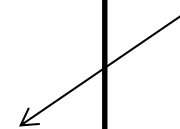
```
P1:
flag[1] = true;
while (flag[0] == true) {
    if (turn ≠ 1) {
        flag[1] = false;
        while (turn ≠ 1) { }
        flag[1] = true;
    }
}
```

// critical section

...

```
turn  = 0;
flag[1] = false;
// remainder section
```

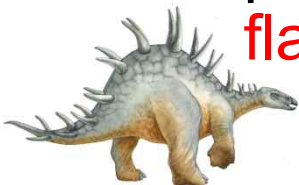
2중 while



# 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<sub>i</sub>** is ready!





# 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);
- Problems
  - Busy Waiting! (계속 CPU와 memory 를 쓰면서 wait)
  - Software 적인 해결책은 느림



# Bakery 알고리즘

---

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

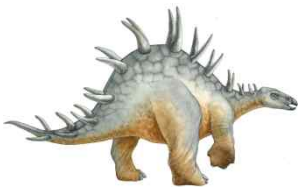


```

// "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 }

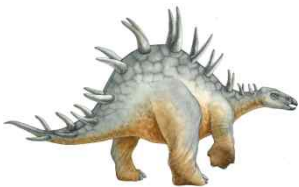
```



# Critical Section Using Locks

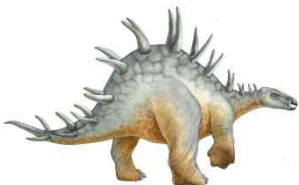
---

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



# 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-interruptible**
    - Either test memory word and set value
    - Or swap contents of two memory words



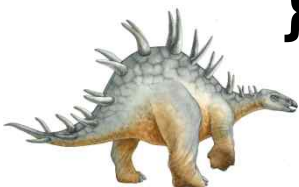
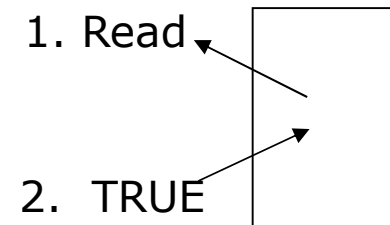
# Synchronization Hardware

상호배제(Mutual Exclusion)의 구현

- Lock에 대한 testAndSet를 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)

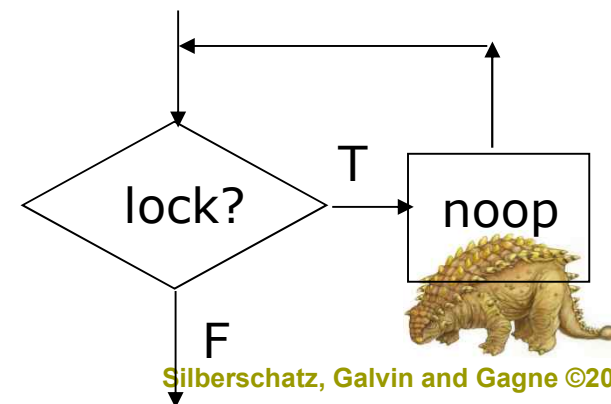


# Mutual Exclusion with Test-and-Set

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 { };



# Synchronization Hardware : Swap 이용

상호배제(Mutual Exclusion)의 구현

- Atomically swap two variables.
  - swap 명령은 CPU에서 지원할 경우가 많음

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



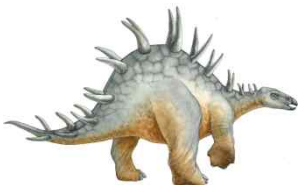


# Mutual Exclusion with Swap

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;



**until** false;

잔류 구역

process의 순서대로 lock을 줌

다른 프로세스가 요청 후 기다리고 있는지 차례로 검사

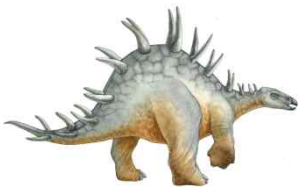
요청 후 기다리고 있는 프로세스가 하나도 없으면...

만약 있다면, lock을 풀지 않은 채로 대기 중인 프로세스를 임계 구역으로 진입시킴



# Semaphores

- 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

---

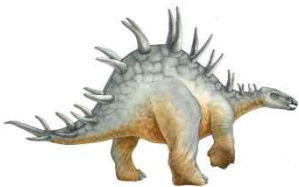
- ❑ Synchronization tool that does not require busy waiting.
- ❑ Semaphore  $S$  – integer variable
- ❑ can only be accessed via two indivisible (atomic) operations

*wait* ( $S$ ):

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

*signal* ( $S$ ):

**$S++$ ;**



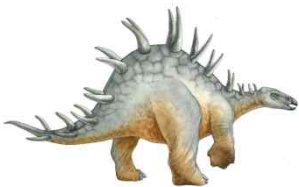
# Critical Section of $n$ Processes

---

- Shared data:  
**semaphore mutex;** //initially *mutex* = 1

- Process  $P_i$ :

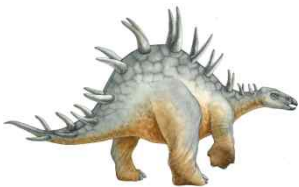
```
do {  
    wait(mutex);  
  
    critical section  
  
    signal(mutex);  
  
    remainder section  
} while (1);
```



# Semaphores 사용 예

---

```
Semaphore Printer;  
init(Printer, 3);  
:  
-----  
wait (Printer);  
-----  
    use a printer;  
-----  
    signal (Printer);  
-----  
    remainder section;
```



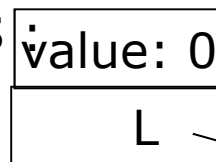
# Semaphore Implementation

- 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)

struct

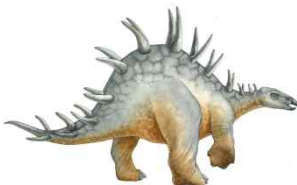
semaphore S



PCB

PCB

PCB



# 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, SE6에서 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();
            MutualExclusionUtilities.remainderSection(name);
        }
    }
}
```



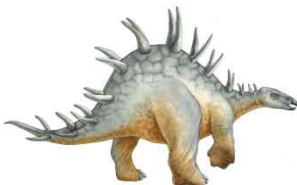
# Semaphore in Java : example

---

```
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() ));

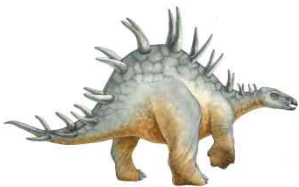
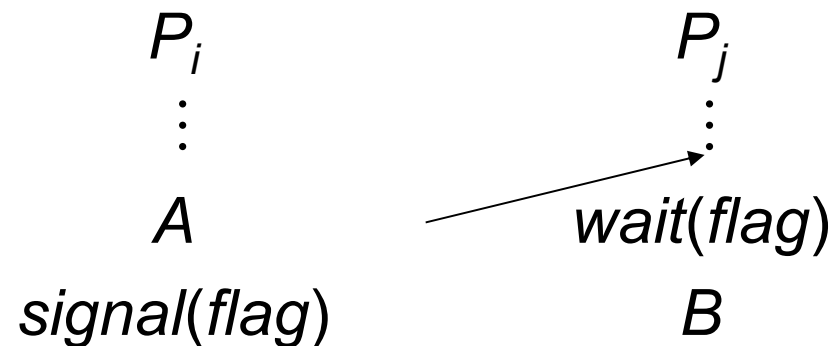
        for (int i = 0; i < 5; i++)
            bees[i].start();
    }
}
```



# Semaphore as a General Synchronization Tool

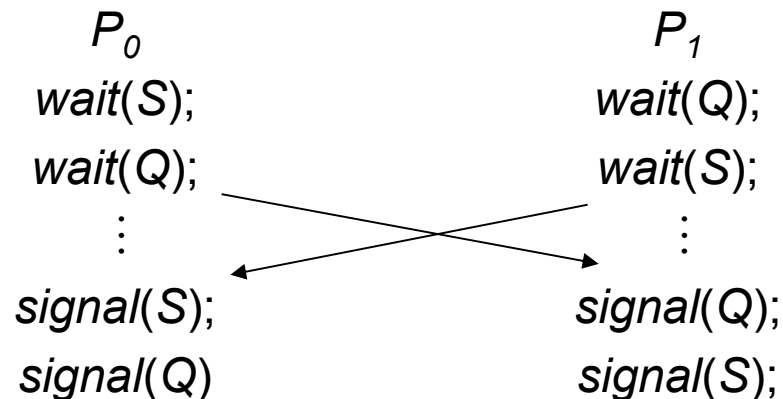
프로세스 Sync를 위해 Semaphore 사용하기

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

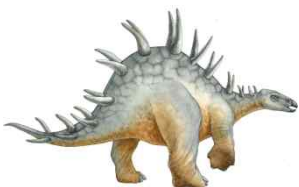


# Deadlock and Starvation

- ❑ **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



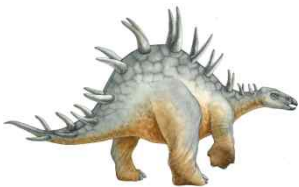
- ❑ **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.



# Two Types of Semaphores

---

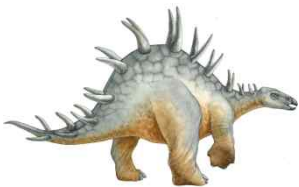
- ❑ *Counting* semaphore – integer value can range over an unrestricted domain.
- ❑ *Binary* semaphore – integer value can range only between 0 and 1; can be simpler to implement.
- ❑ Can implement a counting semaphore  $S$  as a binary semaphore.



# Classical Problems of Synchronization

---

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem



# Bounded-Buffer Problem

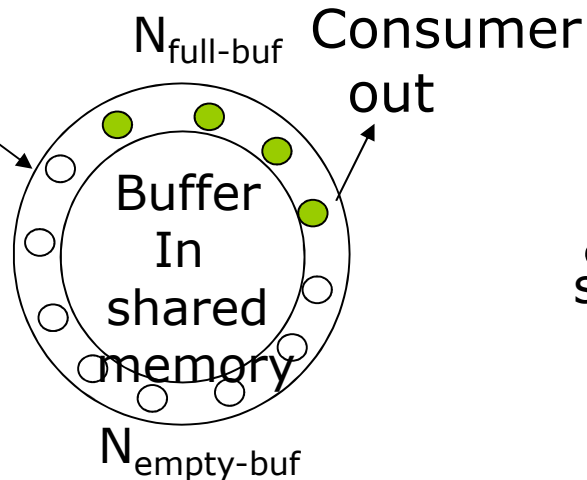
New data arrived

Any empty buf?  
Fill it  
Produce full buf

If yes, but ...

Can I access  
shared variable now?

Producer  
in



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 # of empty buf	==>	Need integer semaphore





# Bounded-Buffer Problem

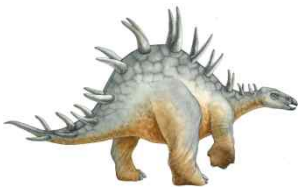
---

- Shared data

**semaphore full, empty, mutex;**

Initially:

**full = 0, empty = n, mutex = 1**



# Bounded-Buffer Problem Producer Process

---

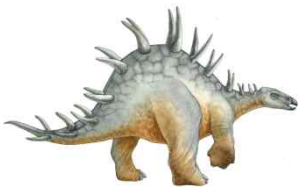
```
do {  
    ...  
    produce an item in nextp  
    ...  
    wait(empty);  
    wait(mutex);  
    ...  
    add nextp to buffer  
    ...  
    signal(mutex);  
    signal(full);  
} while (1);
```



# Bounded-Buffer Problem Consumer Process

---

```
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
    }
}
```

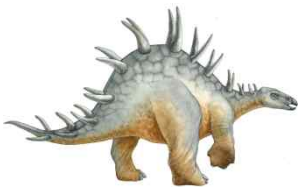


# Bounded-Buffer : Java 구현 예

---

## 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();  
}
```

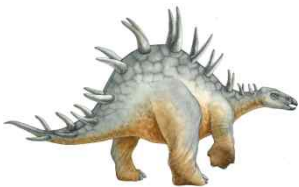


# Bounded-Buffer : Java 구현 예

---

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

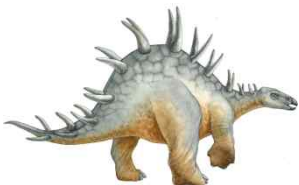
## □ The structure of the producer process

```
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

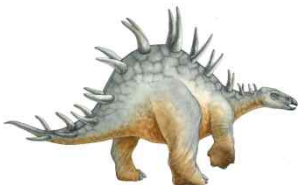
## □ The structure of the consumer process

```
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();
        }
    }
}
```





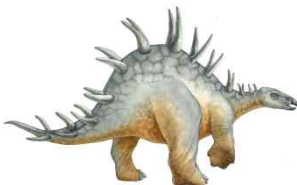
# Bounded-Buffer : Java 구현 예

## □ 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();
    }
}
```



# Readers-Writers Problem

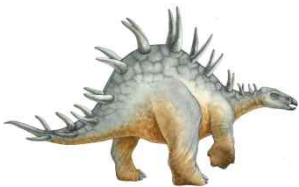
---

- Shared data

**semaphore mutex, wrt;**

Initially

**mutex = 1, wrt = 1, readcount = 0**



# Readers-Writers Problem Writer Process

---

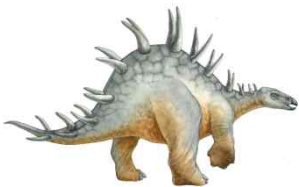
**wait(wrt);**

...

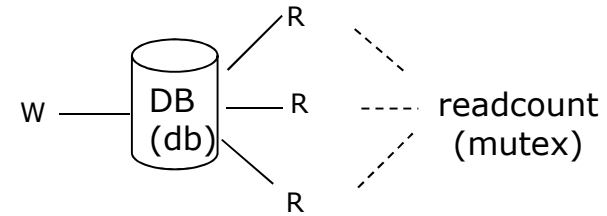
writing is performed

...

**signal(wrt);**



# Readers-Writers Problem Reader Process



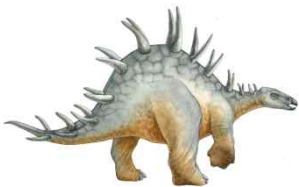
```
wait(mutex);
readcount++;
if (readcount == 1)
    wait(wrt);
signal(mutex);
```

다른 프로세스가 쓰는 중이면  
기다림

...  
reading is performed

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

읽기 카운트를  
증가 시키는중에  
다른 프로세스가  
Readcount를 증가시키지  
못하도록 함

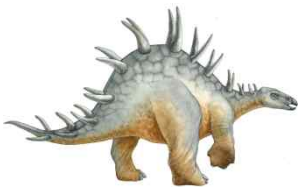


# Readers-Writers Problem : Java 구현 예

---

Interface for read-write locks

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

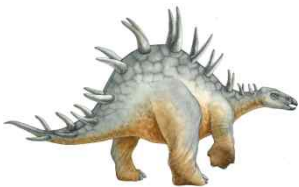


# Readers-Writers Problem : Java 구현 예

---

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;

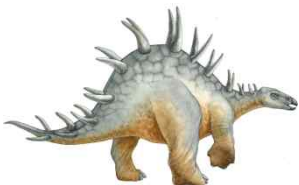
    public Writer(RWLock db) {
        this.db = db;
    }

    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;

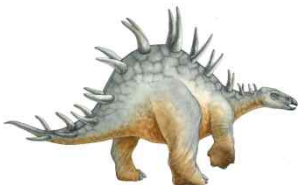
    public Reader(RWLock db) {
        this.db = db;
    }

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

            db.acquireReadLock();

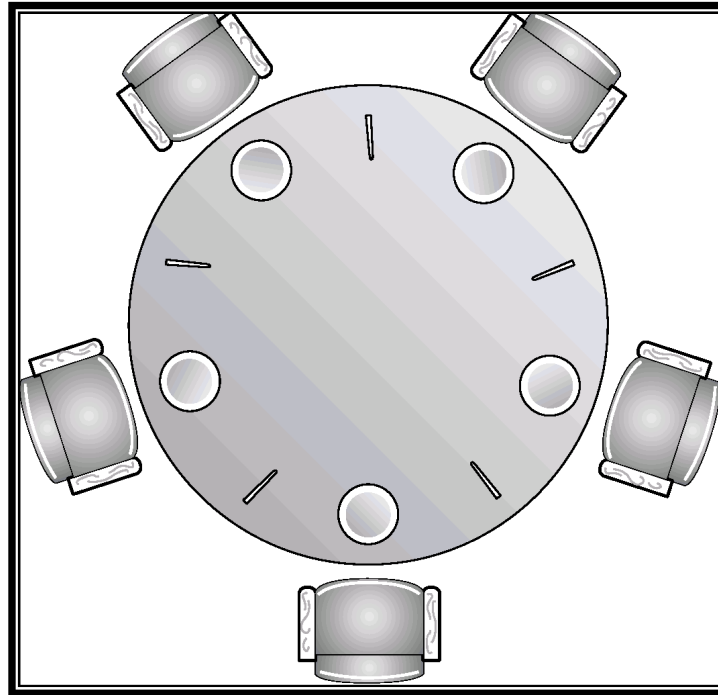
            // you have access to read from the database
            SleepUtilities.nap();

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





# Dining-Philosophers Problem



□ Shared data

**semaphore chopstick[5];**

Initially all values are 1



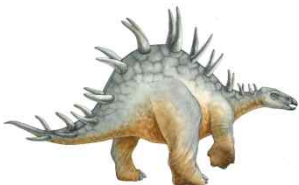
# Dining-Philosophers Problem

□ 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!

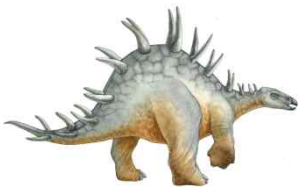
왼쪽 숟가락과  
오른쪽 숟가락을  
모두 반납!



# Semaphores의 문제점

---

- Difficult to code
- Difficult to prove correctness
  - \*\* errors are not reproducible
  - \*\* error are observed rarely
- Requires voluntary cooperation
- Single misuse affect entire system



# Critical Regions

---

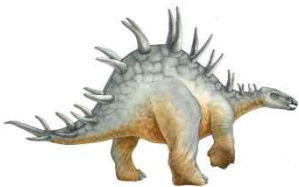
- High-level synchronization construct
- A shared variable  $v$  of type  $T$ , is declared as:

**$v$ : shared  $T$**

- Variable  $v$  accessed only inside statement  
**region  $v$  when  $B$  do  $S$**

where  $B$  is a boolean expression.

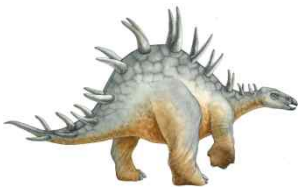
- While statement  $S$  is being executed, no other process can access variable  $v$ .



# Critical Regions

---

- ❑ Regions referring to the same shared variable exclude each other in time.
- ❑ When a process tries to execute the region statement, the Boolean expression  $B$  is evaluated. If  $B$  is true, statement  $S$  is executed. If it is false, the process is delayed until  $B$  becomes true and no other process is in the region associated with  $v$ .

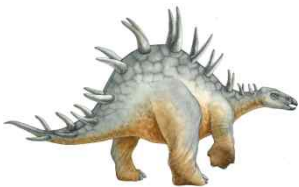


# Example – Bounded Buffer

---

□ Shared data:

```
struct buffer {  
    int pool[n];  
    int count, in, out;  
}
```

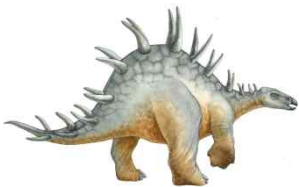


# Bounded Buffer Producer Process

---

- Producer process inserts **nextp** into the shared buffer

```
region buffer when( count < n) {  
    pool[in] = nextp;  
    in:= (in+1) % n;  
    count++;  
}
```

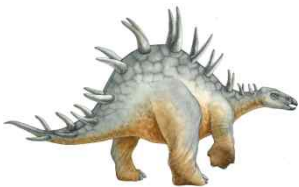


# Bounded Buffer Consumer Process

---

- Consumer process removes an item from the shared buffer and puts it in **nextc**

```
region buffer when (count > 0) {  
    nextc = pool[out];  
    out = (out+1) % n;  
    count--;  
}
```

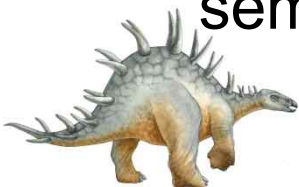




# Implementation region $x$ when $B$ do $S$

---

- ❑ Associate with the shared variable  $x$ , the following variables:  
**semaphore mutex, first-delay, second-delay;**  
**int first-count, second-count;**
- ❑ Mutually exclusive access to the critical section is provided by **mutex**.
- ❑ If a process cannot enter the critical section because the Boolean expression **B** is false, it initially waits on the **first-delay** semaphore; moved to the **second-delay** semaphore before it is allowed to reevaluate  $B$ .



# Implementation

---

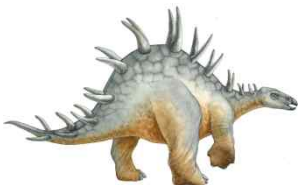
- ❑ Keep track of the number of processes waiting on **first-delay** and **second-delay**, with **first-count** and **second-count** respectively.
- ❑ The algorithm assumes a FIFO ordering in the queuing of processes for a semaphore.
- ❑ For an arbitrary queuing discipline, a more complicated implementation is required.



# 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
    }
}
```

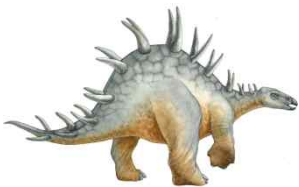


# Monitors

---

## □ 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**.

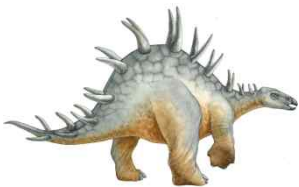
- The operation

**x.wait();**

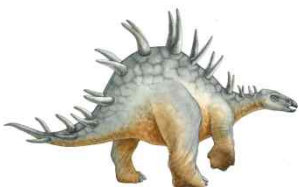
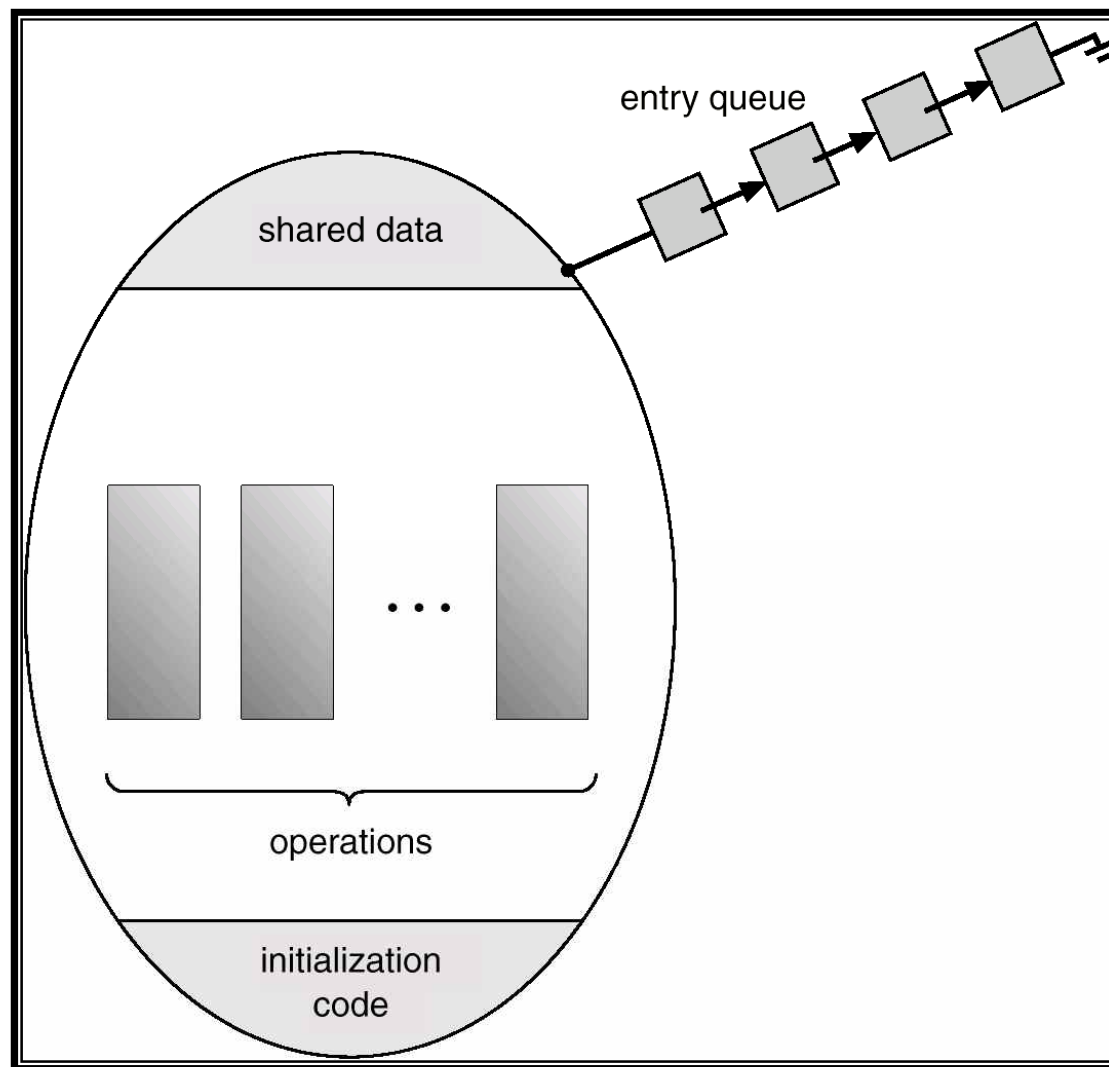
means that the process invoking this operation is suspended until another process invokes

**x.signal();**

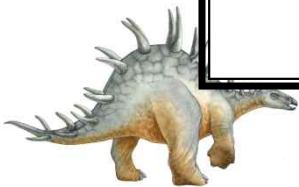
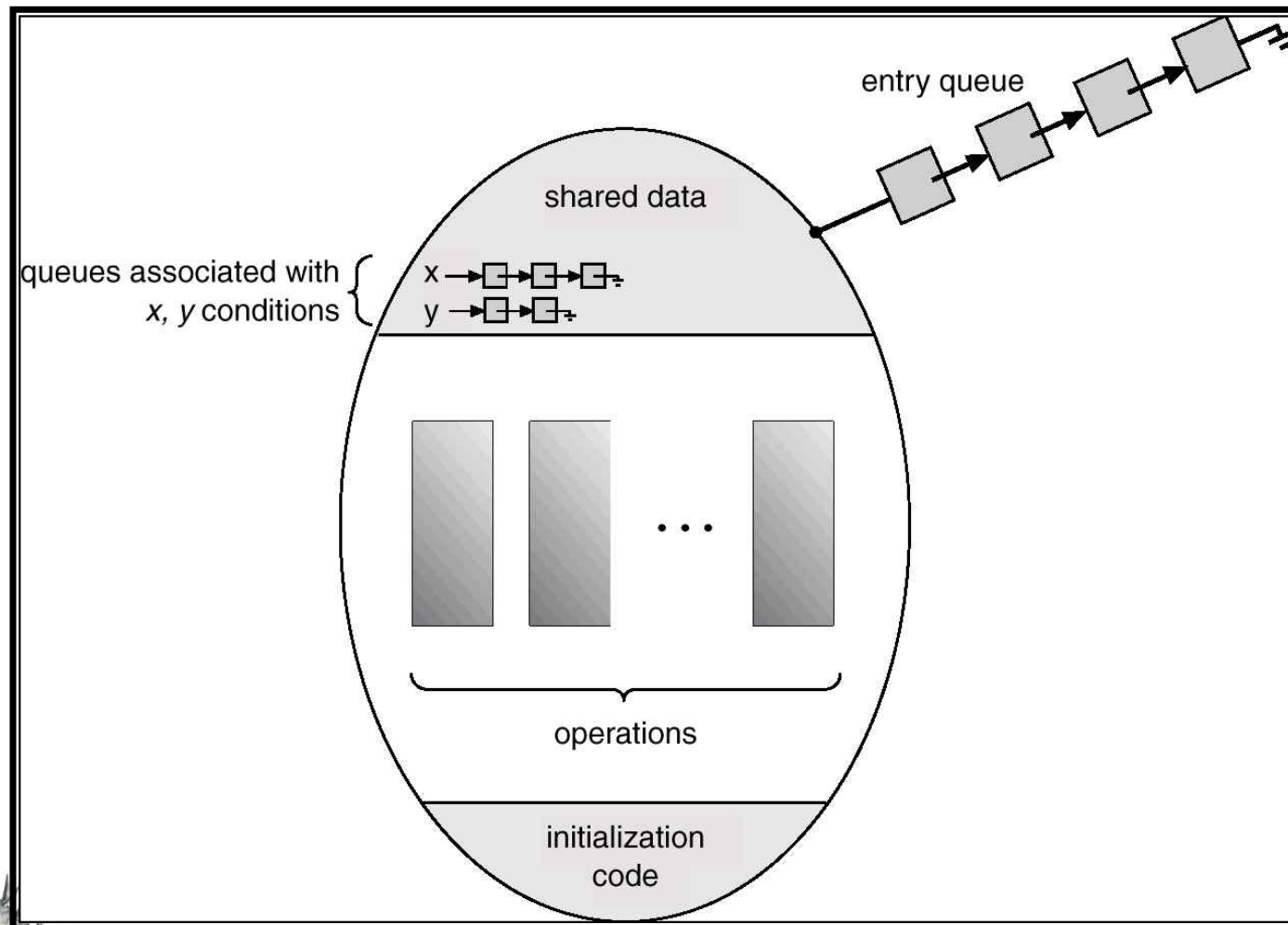
- The **x.signal** operation resumes exactly one suspended process. If no process is suspended, then the **signal** operation has no effect.



# Schematic View of a Monitor



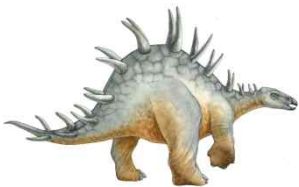
# Monitor With Condition Variables



# 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;
    }
}
```

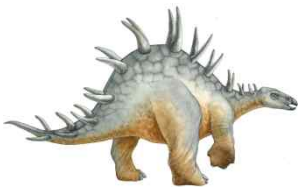




# Dining Philosophers

---

```
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);**

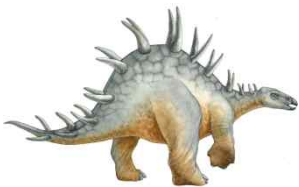
- Mutual exclusion within a monitor is ensured.



# Monitor Implementation

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

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

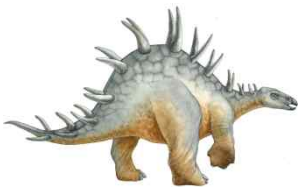


# Monitor Implementation

---

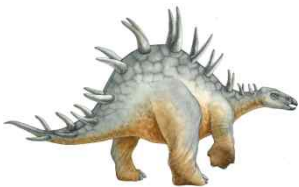
- The operation **x.signal** can be implemented as:

```
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.



# Java 5.0 Synchronization

```
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 Synchronization

## □ Synchronized를 이용한 동기화

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

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

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





# Java 5.0 Synchronization

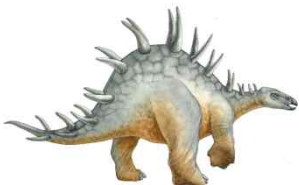
## □ ReadWriteLock을 이용한 Synchronization

```
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 Synchronization

- ReadWriteLock에서 Deadlock의 회피
  - 일정시간동안 lock을 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();  
    }  
}
```

