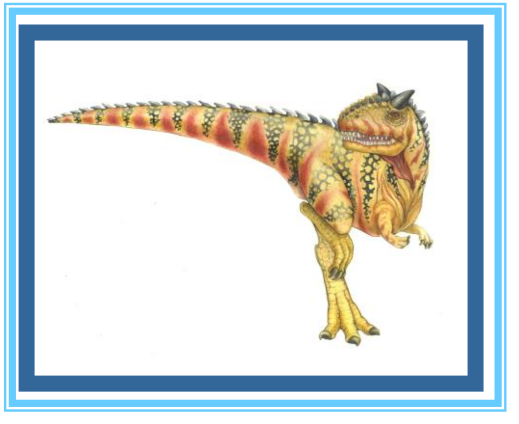
# Chapter 6: Synchronization



# Synchronization

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

# Objectives

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity

# Background

Concurrent access to shared data may result in data inconsistency

Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Suppose that we want to provide a solution to the consumer-producer problem that fills all the buffers.

We can do so by having an integer count that keeps track of the number of full buffers.

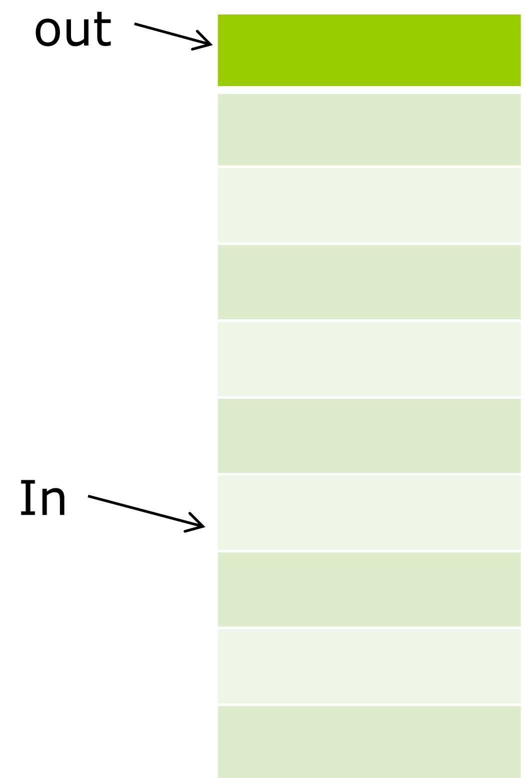
Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

# Producer

```
while (true) {

        /*  produce an item and put in nextProduced  */
         while (count == BUFFER_SIZE)
                 ; // do nothing
             buffer [in] = nextProduced;
             in = (in + 1) % BUFFER_SIZE;
             count++;
}
```

# Consumer




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

                     /*  consume the item in nextConsumed
}
```

# Race Condition

**count++** could be implemented as

register1 = count
register1 = register1 + 1
count = register1

**count--** could be implemented as

register2 = count
register2 = register2 - 1
count = register2

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}

# Critical-Section Problem

do {

*entry section*

critical section

*exit section*

remainder section

} while (TRUE);

**General structure of a typical Process $P_i$**

# Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process $P_i$ is executing in its critical section, then no other processes can be executing in their critical sections

do {

*entry section*

critical section

*exit section*

remainder section

} while (TRUE);

do {

*entry section*

critical section

*exit section*

remainder section

} while (TRUE);

# Solution to Critical-Section Problem

2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then **the selection of the processes** that will enter the critical section next **cannot be postponed indefinitely**

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

# Solution to Critical-Section Problem

3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the N processes

# 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[2]

The variable turn indicates whose turn it is to enter the critical section.

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!

# Algorithm for Process $P_i$

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
        critical section
    flag[i] = FALSE;
        remainder section
} while (TRUE);
```

# Prove this algorithm is correct

1. Mutual exclusion is preserved
2. The progress requirement is satisfied.
3. The bounded waiting requirement is met

# Prove this algorithm is correct

## 1. Mutual exclusion is preserved

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
        critical section
    flag[i] = FALSE;
        remainder section
} while (TRUE);
```

```
do {
    flag[j] = TRUE;
    turn = i;
    while (flag[i] && turn == i);
        critical section
    flag[j] = FALSE;
        remainder section
} while (TRUE);
```

# Prove this algorithm is correct

## 2. The progress requirement is satisfied.

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
        critical section
    flag[i] = FALSE;
        remainder section
} while (TRUE);
```

```
do {
    flag[j] = TRUE;
    turn = i;
    while (flag[i] && turn == i);
        critical section
    flag[j] = FALSE;
        remainder section
} while (TRUE);
```

# Prove this algorithm is correct

## 3. The bounded waiting requirement is met

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
        critical section
    flag[i] = FALSE;
        remainder section
} while (TRUE);
```

```
do {
    flag[j] = TRUE;
    turn = i;
    while (flag[i] && turn == i);
        critical section
    flag[j] = FALSE;
        remainder section
} while (TRUE);
```

# Synchronization Hardware

**Any solution to the critical-section problem requires a simple tool – a lock.**

**Race conditions are prevented by requiring that critical regions be protected by locks**

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

# 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

# TestAndndSet Instruction

**Definition:**

```
boolean TestAndSet (boolean *target)
{
→    boolean rv = *target;   /* Test */
→    *target = TRUE;         /* Set */
     return rv:
}
```

# Solution using TestAndSet

Shared boolean variable lock., initialized to false.

Solution (Mutual-Exclusion):

lock


```
do {
    while ( TestAndSet (&lock ))
            ;   // do nothing
        //    critical section
    lock = FALSE;
        //    remainder section
} while (TRUE);
```

# Swap Instruction

**Definition:**

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

# Solution using Swap

Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key

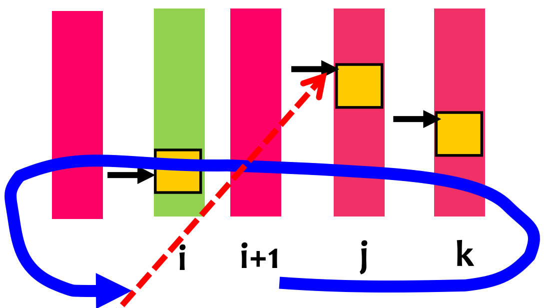
Solution(Mutual-Exclusion):

```
do {
        key = TRUE;
        while ( key == TRUE)
                Swap (&lock, &key );
                //   critical section
        lock = FALSE;
                //     remainder section
} while (TRUE);
```

lock

# Bounded-waiting Mutual Exclusion with TestandSet()

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
            key = TestAndSet(&lock);
    waiting[i] = FALSE;
            // critical section
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])   (Find next waiting process)
            j = (j + 1) % n;
    if (j == i)
            lock = FALSE; (No one is waiting)
    else
            waiting[j] = FALSE; (process j enters next)
            // remainder section
} while (TRUE);
```



# Prove this algorithm is correct

1. **Mutual exclusion** is preserved
2. The **progress** requirement is satisfied.
3. The **bounded waiting** requirement is met

# Semaphores

The hardware-based solutions for the CS problem are complicated for application programmers to use.

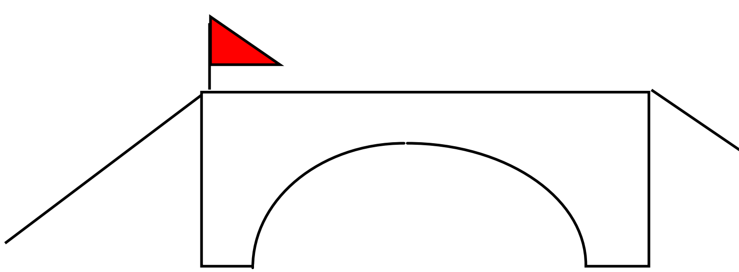
To overcome this difficulty, we use a synchronization tool called a semaphore.

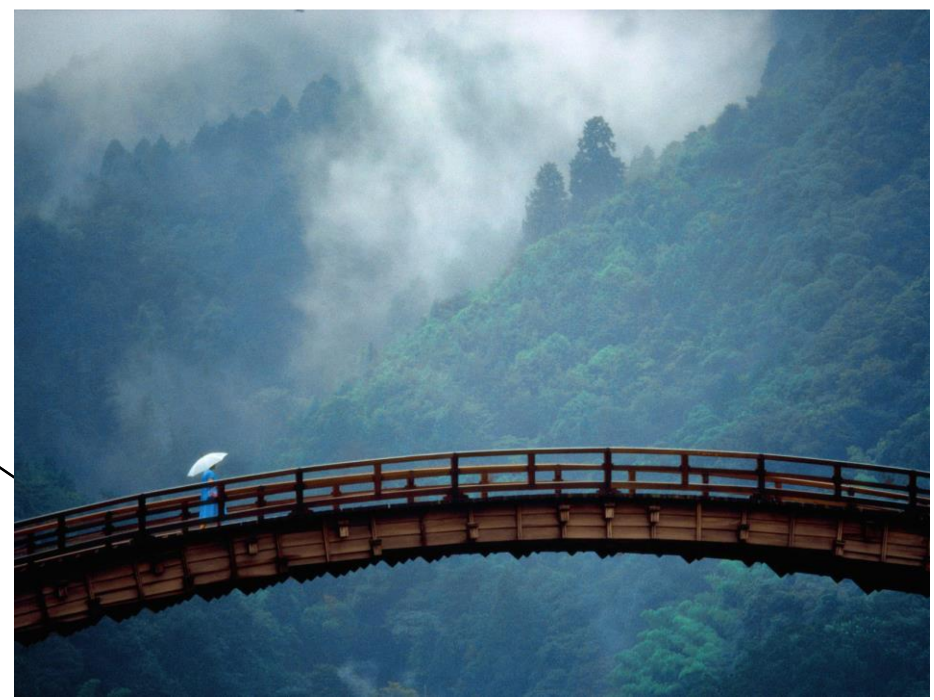
Semaphore *S* – integer variable

Two standard operations modify S: wait() and signal()

- Originally called P() and V()

Less complicated





# Semaphores

Can only be accessed via two indivisible (atomic) operations

```
wait (S) {
    while S <= 0        /* Semaphore S is occupied */
        ; // no-op
      S--;              /* Semaphore S is available, get it */
 }
signal (S) {
    S++;                /* Release the semaphore S */
 }
```

# Semaphore Usage

- Counting semaphore – integer 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 as they are locks that provide mutual exclusion.
- We can use binary semaphore to deal with the CS problem for multiple processes.
- The n processes share a semaphore, mutex, initialized to 1

# Mutual-Exclusion Implementation with semaphores

**Provides mutual exclusion (for Process Pi)**

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

# Semaphore Usage

Counting semaphore can be used to control access to a given resource consisting of a finite number of instances.

The semaphore is initialized to the number of resources available.

To use a resource, wait()

To release a resource, signal()

Semaphores can be used to solve various synchronization problem.

For example, we have two processes P1 and P2. Execute S1 and then S2: Synch = 0

| P1 | P2 |
| --- | --- |
| S1; | wait(synch); |
| signal(synch); | S2; |

# Semaphore Implementation

The main disadvantage of previous mutual-exclusion solution is the *busy waiting* (CPU is wasting).

This type of semaphore is called a *spinlock.*

To overcome this, we can use the concept of *block* and *wakeup* operations.

```
Typedef struc  {
      int value;
      struct  process *list;
} semaphore
```

# Semaphore Implementation with no Busy waiting

With each semaphore there is an associated **waiting queue**. Each entry in a waiting queue has two data items:

- **value** (of type integer)
  - Value > 0 indicates semaphore is still available
  - Value = 0 indicates semaphore is just occupied and no waiting process
  - Value < 0 indicates the number of waiting processes
- **pointer** to next record in the list /* waiting list */

Two operations:

- **block** – place the process invoking the operation on the appropriate waiting queue.
- **wakeup** – remove one of processes in the waiting queue and place it in the ready queue.

# Semaphore Implementation with no Busy waiting

**Implementation of wait:**

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

# Semaphore Implementation with no Busy waiting

**Implementation of signal:**

```
signal(semaphore *S) {
        S.value ++;
        if (S.value <= 0) {
                remove a process P from S.list;
                wakeup(P);
        }
}
```

# Semaphore Implementation with no Busy waiting

Note that the semaphore value may be negative.

Its magnitude is the number of processes waiting on that semaphore.

The list of waiting processes can be easily implemented by a link field in each process control block (PCB).

# 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

| P0 | P1 |
|---|---|
| *wait(S);* | *wait(Q);* |
| *wait(Q);* | *wait(S);* |
| ⋮ | ⋮ |
| *signal(S);* | *signal(Q);* |
| *signal(Q);* | *signal(S);* |

**Deadlock !!**

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

# Priority Inversion

Priority Inversion - Scheduling problem when lower-priority process holds a lock needed by higher-priority process.

Three processes L, M, H with priority L < M < H

Assume process H requires resource R, which is using by process L. Process H waits.

Assume process M becomes runnable, thereby preempting process L.

Indirectly, a process with lower priority – M – has affected how long H must wait for L to release R.

Priority-inheritance protocol – all processes that are accessing resources needed by a higher priority process inherit the higher priority until they are finished with the resources.

# Classical Problems of Synchronization

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

# Bounded-Buffer Problem

Used to illustrate the power of synchronization primitives.

*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 (Cont.)

The structure of the producer process

```
do {

        //  produce an item in nextp

    wait (empty);
    wait (mutex);

        //  add the item to the  buffer

    signal (mutex);
    signal (full);
 } while (TRUE);
```

The structure of the consumer process

```
do {
    wait (full);
    wait (mutex);

        //  remove an item from  buffer to nextc

    signal (mutex);
    signal (empty);

        //  consume the item in nextc

} while (TRUE);
```

**Initial, Empty = N, Full = 0**

# The Reader and Writers Problem

A data object, such as a file or record, is to be shared among several concurrent processes.

The writers are required to have exclusive access to the shared object.

The readers-writers problem has several variations, all involving priorities.

The First problem -- require no reader will be kept waiting unless a writer has already obtained permission to use the shared object. Thus, no reader should wait for other readers to finish even a writer is waiting.

# The Reader and Writers Problem

**The Second problem -- require once a writer is ready, that writer performs its write as soon as possible, after old readers (or writer) are completed. Thus, if a writer is waiting to access the object, no new readers may start reading.**

**A solution to either problem may result in *starvation*.**

- **The first problem : Writers**
  - **Writers wait, but readers come in one after one**
- **The second problem : Readers**
  - **Readers wait, but writers come in one after one**

# A solution for the first problem

Shared Data

- Semaphore mutex initialized to 1
- Semaphore wrt initialized to 1
- Integer readcount initialized to 0

The *mutex semaphore* is used to ensure mutual exclusion when the variable readcount is updated.

*Readcount* keeps track of how many processes are currently reading the object.

The *wrt semaphore* functions as a mutual exclusion semaphore for the writers.

- It also is used by the *first* or *last* reader that enters or exits the critical section.
- It is not used by the readers who enter or exit while other processes are in their critical sections.

# A solution for the first problem

**If a writer is in the CS and *n* readers are waiting, then one reader is queued on *wrt* and n-1 readers are queued on *mutex*.**

**When a writer executes signal(wrt), we may assume the execution of either the waiting writers or a single reader. The selection is made by the scheduler.**

**First reader**

**Last reader**

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

                // reading is performed

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

```
do {
        wait (wrt) ;
        //  writing is performed
        signal (wrt) ;
} while (TRUE);
```

# A solution for the first problem

First reader

Last reader

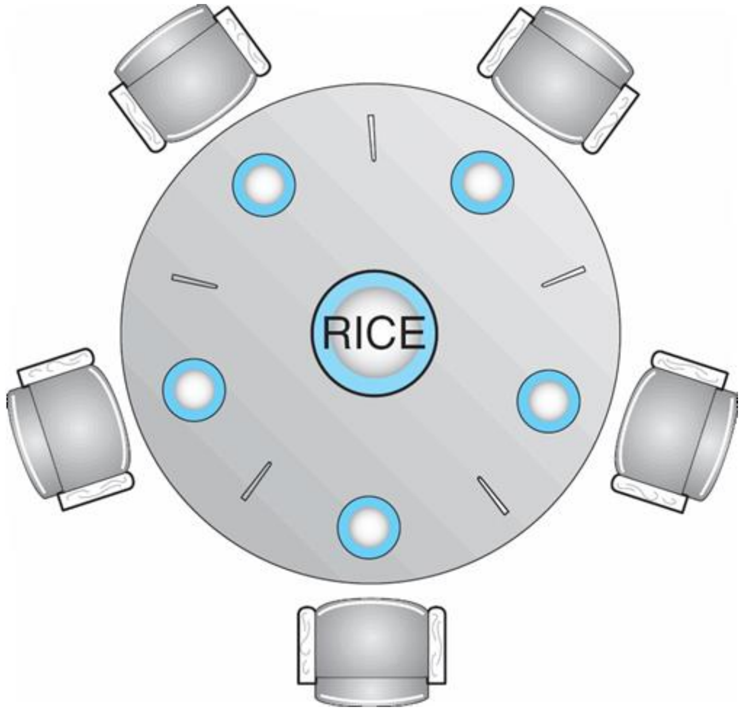
```
do {
    wait (mutex) ;
    readcount ++ ;
    if (readcount == 1)
              wait (wrt) ;
    signal (mutex)

    // reading is performed

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

```
do {
    wait (wrt) ;
    //  writing is performed
    signal (wrt) ;
} while (TRUE);
```

# 6.6.3 Dining-Philosophers Problem




Shared data

Bowl of rice (data set)

Semaphore chopstick [5] initialized to 1

# Dining-Philosophers Problem (Cont.)

Represent each chopstick by a semaphore.

Wait and Signal on the semaphores.

Var *chopstick:* array [0..4] of *semaphores;*

The structure of Philosopher *i*:

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

Deadlock !!

## Several possible solutions to the deadlock problem

Allow at most four philosophers to be sitting simultaneously at the table.

Allow a philosopher to pick up her chopsticks only if both chopsticks are available (note that she must pick them up in a critical section).

Use an asymmetric solution. Thus,

an odd philosopher picks up first her left chopstick and then her right chopstick, whereas

an even philosopher picks up her right chopstick and then her left chopstick.

# Problems with Semaphores

Correct use of semaphore operations: Otherwise, some problems may happen

signal (mutex) …. wait (mutex)

wait (mutex) … wait (mutex)

Omitting of wait (mutex) or signal (mutex) (or both)

# 6.7 Monitors

A high-level abstraction that provides a convenient and effective mechanism for process synchronization

Only one process may be active within the monitor at a time

monitor monitor-name

{ // shared variable declarations

procedure P1 (…) { …. }

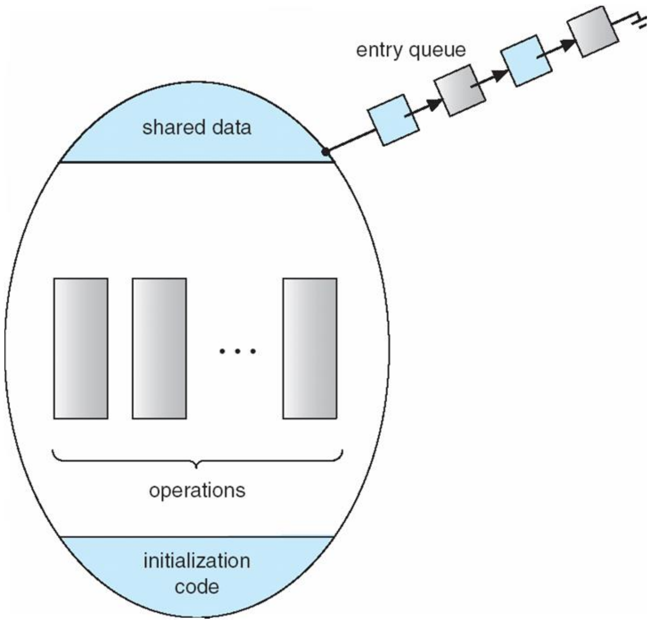
…

procedure Pn (…) {……}

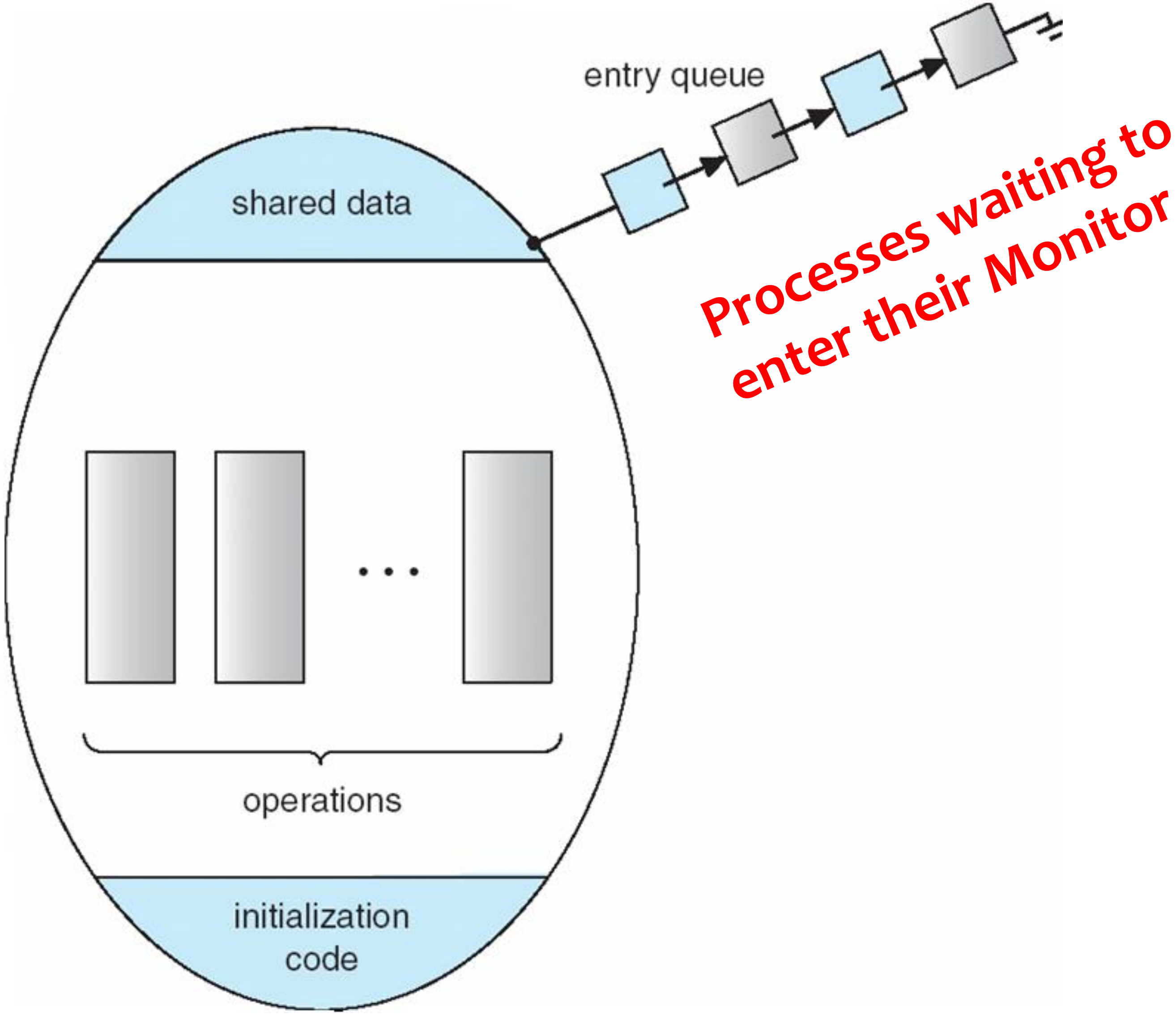
Initialization code ( ….) { … }

…

}



# Schematic view of a Monitor



# Monitors

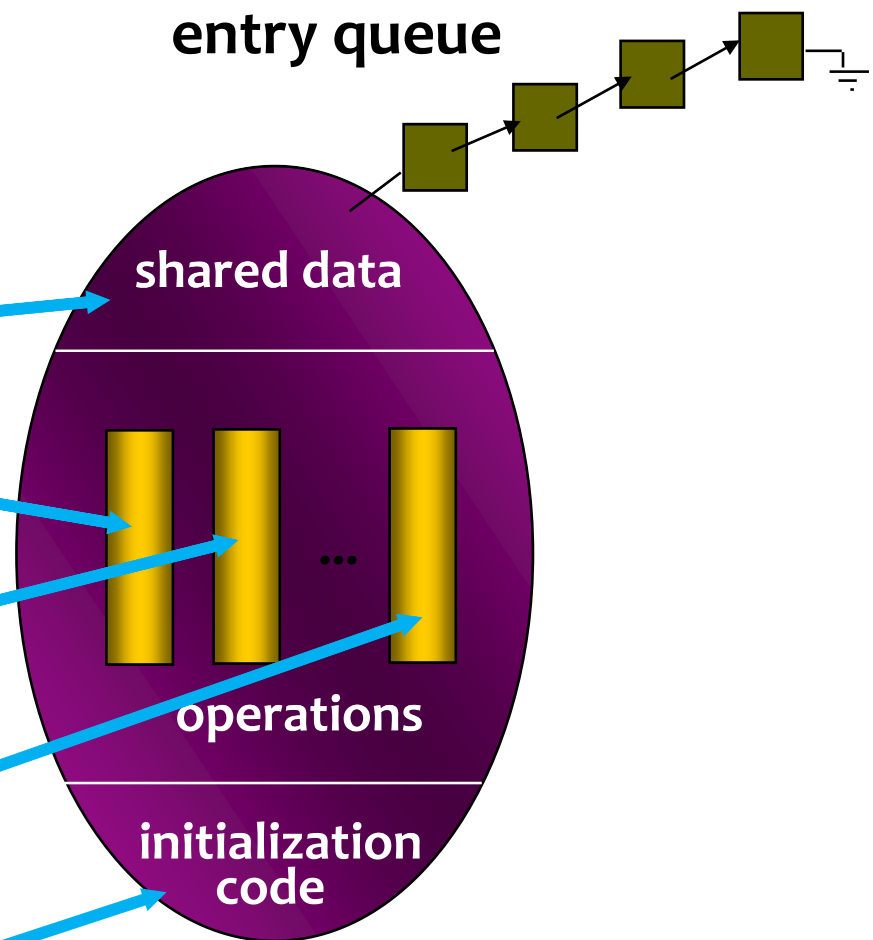
- Syntax of a monitor

```
type monitor-name = monitor
    variable declarations

    procedure entry P1(...);
        begin ... end;

    procedure entry P2(...);
        begin ... end;

    procedure entry Pn(...);
        begin ... end;
    begin
        initialization code
    end.
```




- Schematic view of a monitor

# Condition Construct

A programmer who needs to write her own tailor-made synchronization scheme can define one or more variables of type *condition.*

Var *x,y : condition;*

The only operations that can be invoked on a condition variable are *wait* and *signal*. For example, *x.wait, x.signal.*

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

Suppose P invokes x.signal and Q is suspended with x. Two possibilities exist:

P either waits Q leaves or another condition

Q either waits P leaves or another condition

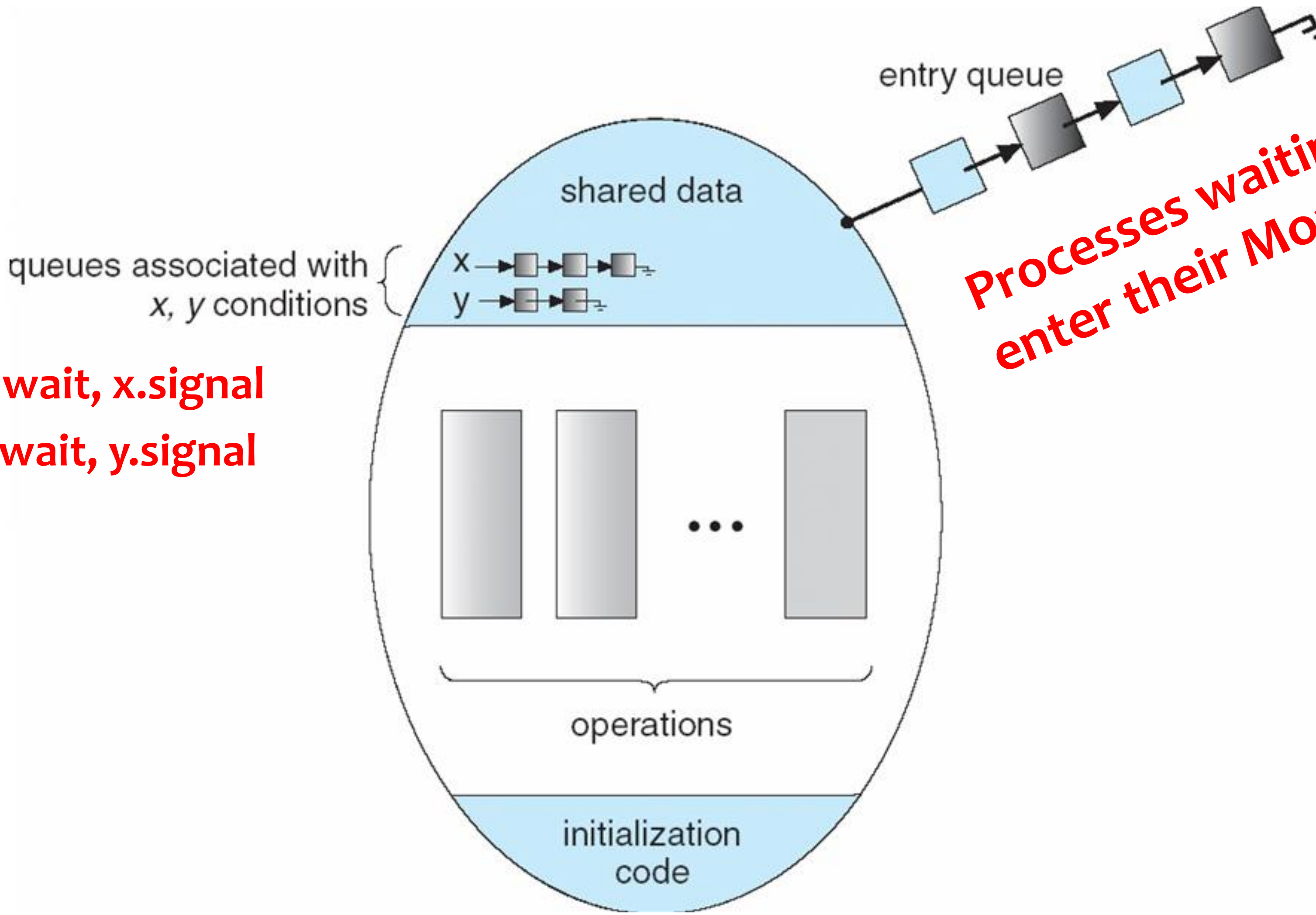
# Condition Variables

condition x, y;

Two operations on a condition variable:

- x.wait () – a process that invokes the operation is suspended.
- x.signal () – resumes one of processes (if any) that invoked x.wait ()

# Monitor with Condition Variables




x.wait, x.signal

y.wait, y.signal

Processes waiting to enter their Monitor

# A Deadlock-free Monitor Solution for the Dining-Philosophers Problem

A philosopher is allowed to pick up her chopsticks only if both of them are available.

Data structure:

Var *state*: array [0..4] of (*thinking, hungry, eating);*

Var *self*: array [0..4] of *condition*;

Philosopher *i* can delay herself when she is hungry, but is unable to obtain the chopsticks she needs.

Operations:

*pickup* and *putdown* on the instance *dp* of the *dining-philosophers* monitor

# Solution to Dining Philosophers (cont)

Each philosopher *i* must invoke the operations pickup() and putdown() in the following sequence:

**var dining-philosophers: dp**

dining-philosophers.*pickup(i);*

...

eat

...

dining-philosophers.*putdown(i);*

Process *i*

# A Deadlock-free Monitor Solution for the Dining-Philosophers Problem

```
monitor dp
  {
     enum { THINKING; HUNGRY, EATING) state [5] ;
     condition self [5];

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);
}
```

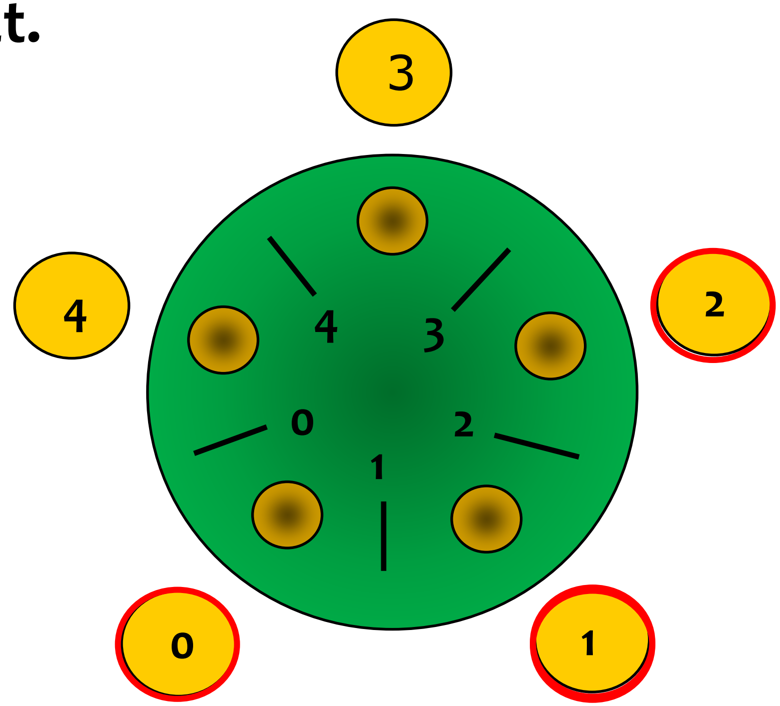
Test if both chopsticks are available

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

 initialization_code() {
        for (int i = 0; i < 5; i++)
        state[i] = THINKING;
 }
}
```

# Illustration of the algorithm

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




Self[2].wait --> Eating

Self[0].wait --> Eating

Eating --> self[1].signal (No effect)

Pushdown
test (0) -> self[0].signal,
test (2) -> self[2].signal

# Monitor Implementation Using Semaphores

- A possible implementation of the monitor mechanism using semaphores.
- For each monitor, a semaphore mutex (init to 1) is provided.
- A process must execute wait (mutex) before entering the monitor and must execute signal (mutex) after leaving the monitor
- Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, next, is introduced (init to 0).
- The signaling processes can use *next* to suspend themselves.
- An integer variable next_count is also provided to count the number of processes suspended on next.

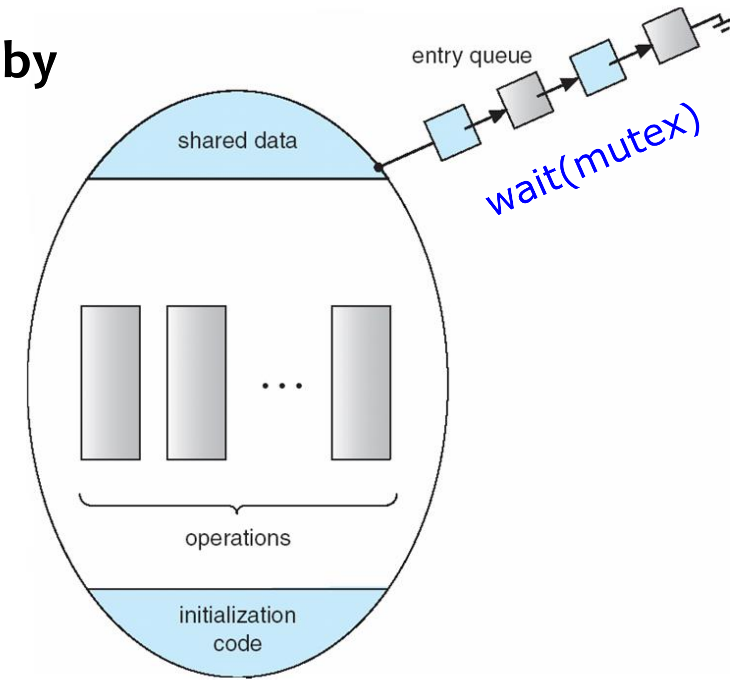
# Monitor Implementation Using Semaphores

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.

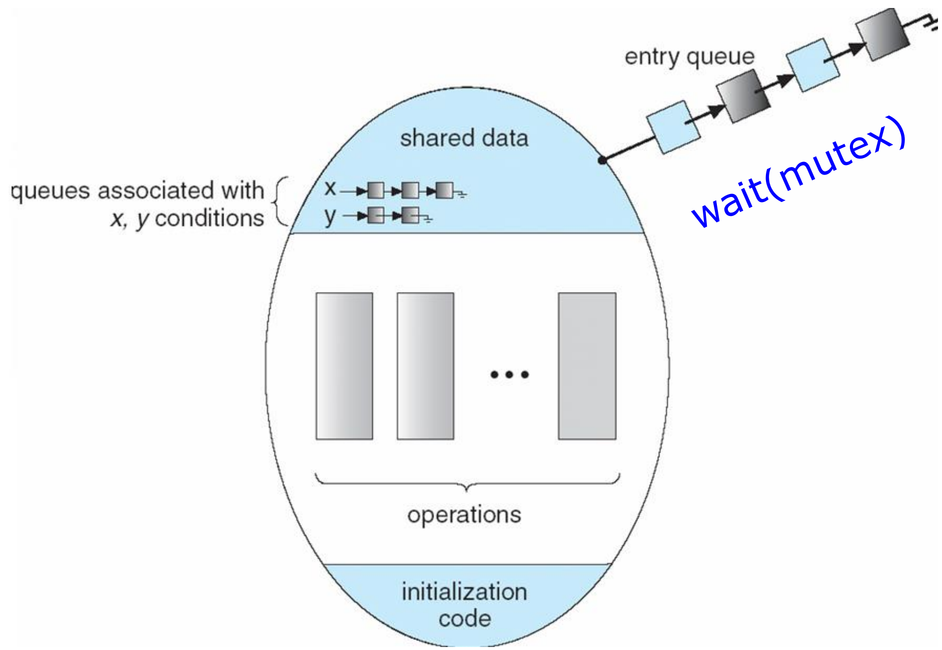
# Monitor (Condition Variable) Implementation Using Semaphores

For each condition variable x, we have:

```
semaphore x_sem; // (initially  = 0)
int x-count = 0;
```

The operation x.wait can be implemented as:

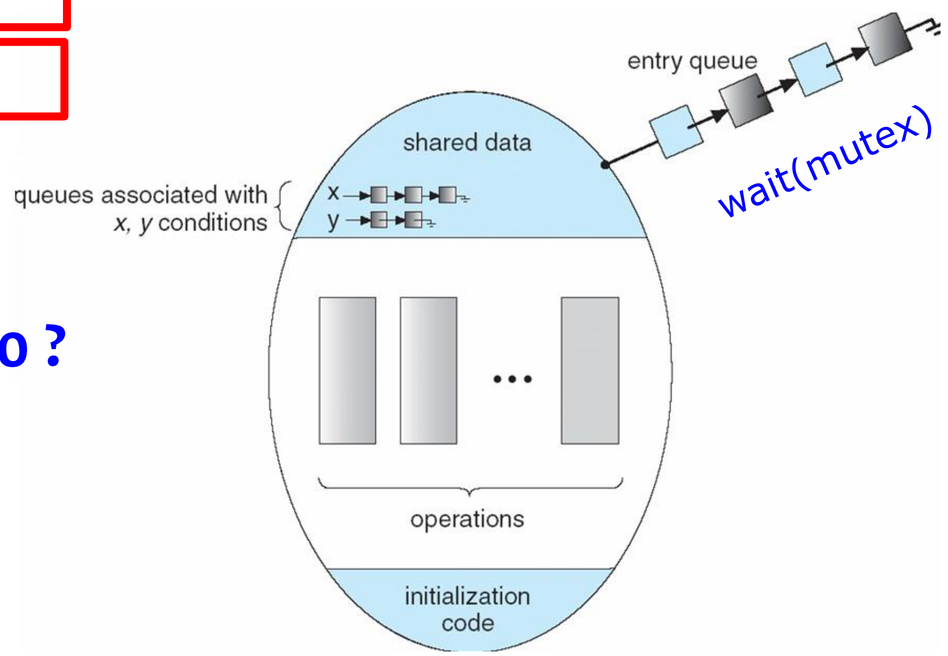
```
x-count++;
if (next_count > 0)
        signal(next);
 else
        signal(mutex);
 wait(x_sem);
  x-count--;
```



# Monitor Implementation Using Semaphores

The operation x.signal can be implemented as:

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




What happen if  x-count <= 0 ?

Nothing will happen !!

# Resuming Processes within a Monitor

- If several processes are suspended on condition x, and an x.signal() operation is executed by some process, how do we determine which of the suspended processes should be resumed next ?
- FCFS ordering is simple, but may not adequate
- Conditional-wait construct
  - x.wait (c)
  - c is an integer expression that is evaluated when the wait() operation is executed.
  - c is called a priority number.
- When x.signal () is executed, the process with smallest priority number is resumed next.

# A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
            if (busy)
                x.wait(time);
            busy = TRUE;
    }
    void release() {
            busy = FALSE;
            x.signal();
    }
initialization code() {
     busy = FALSE;
    }
}
```

The process with smallest priority number is resumed next

# Resuming Processes within a Monitor

The monitor allocates the resource that has the shortest time-allocation request.

A process that needs to access the resource in question must observe the following sequence:

R.acquire (t); ← Get the resource, or wait for it !!

…. ..

access the resource

… …

R. release();

Where R is an instance of type ResourceAllocator.

# 6.8 Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

# Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments
- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

# Windows XP Synchronization

Uses interrupt masks to protect access to global resources on uniprocessor systems

Uses spinlocks on multiprocessor systems

Also provides dispatcher objects which may act as either mutexes and semaphores

Dispatcher objects may also provide events

An event acts much like a condition variable

# Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - semaphores
  - spin locks

# Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spin locks

# 6.9 Atomic Transactions

Make sure that a critical section forms a single logical unit of work that either is performed in its entirety or is nor performed at all.

Consistency of data, along with storage and retrieval of data, is a concern often associated with database systems.

System Model

Log-based Recovery

Checkpoints

Concurrent Atomic Transactions

# 6.9.1 System Model

- Assures that operations (a collection of instructions) happen as a single logical unit of work, in its entirety, or not at all
- Related to field of database systems
- Challenge is assuring atomicity despite computer system failures
- Transaction - collection of instructions or operations that performs single logical function
  - Here we are concerned with changes to stable storage – disk
  - Transaction is series of read and write operations
  - Terminated by commit (transaction successful) or abort (transaction failed) operation
  - Aborted transaction must be rolled back to undo any changes it performed

# Types of Storage Media

**Volatile storage** – information stored here does not survive system crashes

Example: main memory, cache

**Nonvolatile storage** – Information usually survives crashes

Example: disk and tape

**Stable storage** – Information never lost

Not actually possible, so approximated via **replication** or **RAID** to devices with independent failure modes

Goal is to **assure transaction atomicity** where failures cause loss of information on volatile storage

# 6.9.2 Log-Based Recovery

Record to stable storage information about all modifications by a transaction

Most common is write-ahead logging

- Log on stable storage, each log record describes single transaction write operation, including
  - Transaction name
  - Data item name
  - Old value
  - New value
- <$T_i$ starts> written to log when transaction $T_i$ starts
- <$T_i$ commits> written when $T_i$ commits

Log entry must reach stable storage before operation on data occurs

# Log-Based Recovery Algorithm

- Using the log, system can handle any volatile memory errors
  - Undo($T_i$) restores value of all data updated by $T_i$
  - Redo($T_i$) sets values of all data in transaction $T_i$ to new values
- Undo($T_i$) and redo($T_i$) must be **idempotent**
  - Multiple executions must have the same result as one execution
- If system fails, restore state of all updated data via log
  - If log contains <$T_i$ starts> without <$T_i$ commits>, undo($T_i$)
  - If log contains <$T_i$ starts> and <$T_i$ commits>, redo($T_i$)

# 6.9.3 Checkpoints

Log could become long, and recovery could take long

Checkpoints shorten log and recovery time.

Checkpoint scheme:

1. Output all log records currently in volatile storage to stable storage
2. Output all modified data from volatile to stable storage
3. Output a log record <checkpoint> to the log on stable storage

Now recovery only includes Ti, such that Ti started executing before the most recent checkpoint, and all transactions after Ti

All other transactions already on stable storage

# 6.9.4 Concurrent Atomic Transactions

Must be equivalent to serial execution – serializability

Could perform all transactions in critical section

- Inefficient, too restrictive

Concurrency-control algorithms provide serializability

# Serializability

Consider two data items A and B

Consider Transactions $T_0$ and $T_1$

Execute $T_0$, $T_1$ atomically

Execution sequence called schedule

Atomically executed transaction order called serial schedule

For *N* transactions, there are *N!* valid serial schedules

# Schedule 1: $T_0$ then $T_1$

| $T_0$ | $T_1$ |
|---|---|
| read($A$) | |
| write($A$) | |
| read($B$) | |
| write($B$) | |
| | read($A$) |
| | write($A$) |
| | read($B$) |
| | write($B$) |

# Nonserial Schedule

- Nonserial schedule allows overlapped execute
  - Resulting execution not necessarily incorrect
- Consider schedule S, operations $O_i$ , $O_j$ of Transactions $T_i$ and $T_{j,}$
  - Conflict if access same data item, with at least one write
- If $O_i$, $O_j$ are consecutive and operations of different transactions & $O_i$ and $O_j$ don't conflict
  - Then S' with swapped order $O_j$ $O_i$ equivalent to S

| $T_0$ | $T_1$ |
|---|---|
| read($A$) | |
| write($A$) | |
| | read($A$) |
| read($B$) | |
| | write($A$) |
| write($B$) | |
| | read($B$) |
| | write($B$) |

# Nonserial Schedule

**We say that S is conflict serializable, if it can be transformed into a serial schedule S' by a series of swaps of nonconflicting operations.**

| $T_0$ | $T_1$ |
|---|---|
| read($A$) | |
| write($A$) | |
| | read($A$) |
| read($B$) | |
| | write($A$) |
| write($B$) | |
| | read($B$) |
| | write($B$) |

S

| $T_0$ | $T_1$ |
|---|---|
| read($A$) | |
| write($A$) | |
| read($B$) | |
| write($B$) | |
| | read($A$) |
| | write($A$) |
| | read($B$) |
| | write($B$) |

S'

# Locking Protocol

One way to ensure serializability is to associate **a lock with each data item** and each transaction follows locking protocol for access control.

Locks

- **Shared** – $T_i$ has shared-mode lock (S) on item Q, **$T_i$ can read Q but not write Q**
- **Exclusive** – Ti has exclusive-mode lock (X) on Q, **$T_i$ can read and write Q**

Require every transaction on item Q acquire appropriate lock

If lock already held, new request may have to wait

- Similar to readers-writers algorithm

# Two-phase Locking Protocol

The two-phase locking protocol **ensures conflict serializability**

Each transaction issues lock and unlock requests in two phases

- **Growing** – A transaction may obtain locks but may not release any locks
- **Shrinking** – A transaction may release locks but may not obtain any new locks.

Initially, a transaction is a the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and no more lock requests can be issued

Does not prevent deadlock

# Timestamp-based Protocols

- Select order among transactions in advance – timestamp-ordering
- Transaction $T_i$ associated with timestamp $TS(T_i)$ before $T_i$ starts
  - $TS(T_i) < TS(T_j)$ if $T_i$ entered system before $T_j$
  - TS can be generated from system clock or as logical counter incremented at each entry of transaction
- Timestamps determine serializability order
  - If $TS(T_i) < TS(T_j)$, system must ensure produced schedule equivalent to serial schedule where $T_i$ appears before $T_j$

# Timestamp-based Protocol Implementation

## Data item Q gets two timestamps

- **W-timestamp(Q)** – largest timestamp of any transaction that executed write(Q) successfully
- **R-timestamp(Q)** – largest timestamp of successful read(Q)
- Updated whenever read(Q) or write(Q) executed

**Timestamp-ordering protocol** assures any **conflicting read and write** executed in timestamp order

| $T_0$ | $T_1$ |
|---|---|
| $\boxed{\texttt{read}(A)}$ | |
| $\boxed{\texttt{write}(A)}$ | |
| $\texttt{read}(B)$ | |
| $\texttt{write}(B)$ | |
| | $\boxed{\texttt{read}(A)}$ |
| | $\boxed{\texttt{write}(A)}$ |
| | $\texttt{read}(B)$ |
| | $\texttt{write}(B)$ |

# Timestamp-based Protocol Implementation

**Suppose $T_i$ executes read(Q)**

If $TS(T_i) < W\text{-}timestamp(Q)$, Ti needs to read value of Q that was already overwritten

- **read** operation rejected and $T_i$ rolled back

If $TS(T_i) \geq W\text{-}timestamp(Q)$

- **read** executed, R-timestamp(Q) set to max(R-timestamp(Q), $TS(T_i)$)

# Timestamp-ordering Protocol

Suppose Ti executes write(Q)

If $TS(T_i) < R\text{-timestamp}(Q)$, value Q produced by $T_i$ was needed previously and $T_i$ assumed it would never be produced

- **Write** operation rejected, $T_i$ rolled back

If $TS(T_i) < W\text{-timestamp}(Q)$, $T_i$ attempting to write obsolete value of Q

- **Write** operation rejected and $T_i$ rolled back

Otherwise, **write** executed

A transaction $T_i$ is rolled back as a result of either a read or write operation is assigned a new timestamp and is restarted

# Timestamp-ordering Protocol Example

**Assume a transaction is assigned a timestamp immediately before its first instruction.**

**Thus, $TS(T_2) < TS(T_3)$**

**The following schedule is possible under Timestamp Protocol**

| $T_2$ | $T_3$ |
|---|---|
| $\boxed{\texttt{read}(B)}$ | |
| | $\texttt{read}(B)$ |
| | $\boxed{\texttt{write}(B)}$ |
| $\boxed{\texttt{read}(A)}$ | |
| | $\texttt{read}(A)$ |
| | $\boxed{\texttt{write}(A)}$ |

# Timestamp-ordering Protocol

This algorithm ensures

- conflict serializability – conflicting operations are processed in timestamp order, and
- freedom from deadlock – no transactions ever waits

# End of Chapter 6