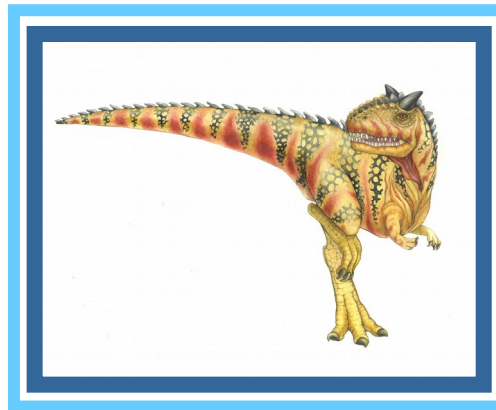


Chapter 6: Synchronization Tools





Chapter 6: Synchronization Tools

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Alternative Approaches

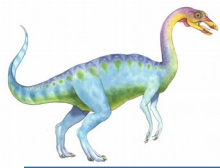




Objectives

- To present the concept of process synchronization.
- 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 examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems





Synchronization

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- The synchronization mechanism is usually provided by both hardware and the operating system
- Illustration of the problem – The producer-Consumer problem, which we introduced in Chapter 3.
- Basic assumption – load and store instructions are atomic.





Producer-Consumer Problem

- The Solution we presented in chapter 3 is correct, but can only use `BUFFER_SIZE - 1` elements.
- The methodology used is to allow only a single process to increment/decrement a particular shared variable.
- There is a solution that fills ***all*** the buffers, using the same methodology:
 - The producer process increments the value on the variable “in” (but not “out”) and the consumer process increments the value on the variable “out” (but not “in”)
 - The solution is more complex. Try and see if you can come up with the algorithm.





Producer-Consumer Problem (Cont.)

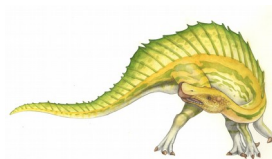
- Suppose that we wanted to provide a solution that fills **all** the buffers where we allow the producer and consumer processes to increment and decrement the same variable.
- We can do so by adding another integer variable -- **counter** that keeps track of the number of full buffers. Initially, **counter** is set to 0.
- The variable **counter** It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.
- Code is shown in next two slides





Producer Process

```
while (true) {  
    /* produce an item in next produced */  
  
    while (counter == BUFFER_SIZE) ;  
        /* do nothing */  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter = counter +1;  
}
```

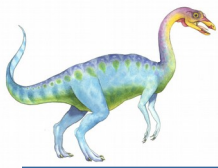




Consumer Process

```
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter = counter - 1;  
  
    /* consume the item in next consumed */  
}
```





Race Condition

- `counter = counter + 1` could be implemented as

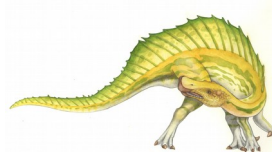
```
register1 = counter
register1 = register1 + 1
counter = register1
```

- `counter = counter - 1` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

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

S0: producer execute	<code>register1 = counter</code>	{register1 = 5}
S1: producer execute	<code>register1 = register1 + 1</code>	{register1 = 6}
S2: consumer execute	<code>register2 = counter</code>	{register2 = 5}
S3: consumer execute	<code>register2 = register2 - 1</code>	{register2 = 4}
S4: producer execute	<code>counter = register1</code>	{counter = 6}
S5: consumer execute	<code>counter = register2</code>	{counter = 4}





Race Condition (Cont.)

- How do we solve the race condition?

- We need to make sure that:

- The execution of

counter = counter + 1

is done as an “atomic” action. That is, while it is being executed, no other instruction can be executed concurrently.

- ▶ actually no other instruction can access **counter**

- Similarly for

counter = counter - 1

- The ability to execute an instruction, or a number of instructions, atomically is crucial for being able to solve many of the synchronization problems.





Critical Section Problem

- Consider system of n processes $\{P_0, P_1, \dots P_{n-1}\}$
- Each process has **critical section** segment of code
 - Process may be changing common variables, updating table, writing file, etc
 - When one process in critical section, no other may be in its critical section
- **Critical section problem** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section** code; it then executes in the critical section; once it finishes executing in the critical section it enters the **exit section** code. The process then enters the **remainder section** code.



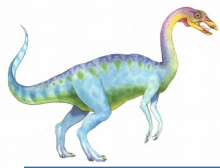


General structure of Process Entering the Critical Section

- General structure of process P_i

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

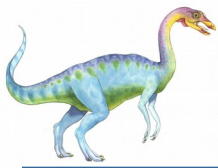




Hardware Solution

- Entry section – first action is to “disable interrupts”.
- Exit section – last action is to “enable interrupts”.
- Must be done by the OS. Why?
- Implementation issues:
 - Uniprocessor systems
 - ▶ Currently running code would execute without preemption
 - Multiprocessor systems.
 - ▶ Generally too inefficient on multiprocessor systems
 - ▶ Operating systems using this not broadly scalable
- Is this an acceptable solution?
 - This is impractical if the critical section code is taking a long time to execute.





Software Solution for Process P_i

- Keep a variable “turn” to indicate which process is next

```
do {  
    while (turn == j);  
        critical section  
    turn = j;  
        remainder section  
} while (true);
```

- Algorithm is correct. Only one process at a time in the critical section. But:
- Results in “busy waiting”.
- What if $\text{turn} = j$; P_i wants to enter the critical section and P_j does not want to enter the critical section?





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
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
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





Software Solution: Peterson's Algorithm

- Good algorithmic software solution
- Two process solution
- Assume that the **load** and **store** machine-language 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 {

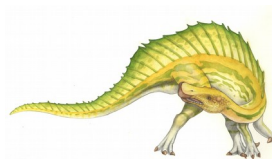
```
flag[i] = true;  
turn = j;  
while (flag[j] && turn == j);
```

critical section

```
flag[i] = false;
```

remainder section

} while (true);





Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:

1. Mutual exclusion is preserved

P_i enters CS only if:

either `flag[j] = false` or `turn = i`

2. Progress requirement is satisfied

3. Bounded-waiting requirement is met

- What about a solution to $N > 2$ processes





Busy Waiting

- All the software solutions we presented employee “busy waiting”
 - A process interested in entering the critical-section is stuck in a loop asking continuously
“can I get into the critical-section”
- Busy waiting is a pure waste of CPU cycles





Solution to Critical-section Problem Using Locks

- Many systems provide hardware support for implementing the critical section code.
- All solutions are based on idea of **locking**
 - Two processes can not have a lock simultaneously.
- Code:

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

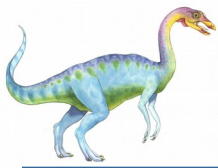




Synchronization Hardware

- Modern machines provide special atomic hardware instructions to implement locks
 - **Atomic** = non-interruptible
- Two types instructions:
 - Test memory word and set value
 - Swap contents of two memory words





test_and_set Instruction

■ Definition:

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

■ Properties:

- Executed atomically
- Returns the original value of passed parameter
- Set the new value of passed parameter to “TRUE”.





Solution using test_and_set()

- Shared Boolean variable **lock**, initialized to FALSE
- Each process, wishing to execute critical-section code:

```
do {  
    while (test_and_set(&lock))  
        ; /* do nothing */  
        /* critical section */  
    lock = false;  
        /* remainder section */  
} while (true);
```

- What about bounded waiting?
- Solution results in busy waiting.





Bounded-waiting Mutual Exclusion with test_and_set

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* remainder section */
} while (true);
```





compare_and_swap Instruction

■ Definition:

```
int compare_and_swap(int *value, int expected, int new_value) {  
    int temp = *value;  
  
    if (*value == expected)  
        *value = new_value;  
    return temp;  
}
```

■ Properties:

- Executed atomically
- Returns the original value of passed parameter “value”
- Set “value” to “new_value ” but only if “value” == “expected”.
That is, the swap takes place only under this condition.



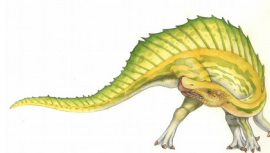


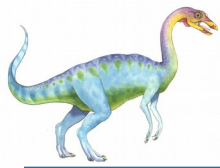
Solution using compare_and_swap

- Shared integer “lock” initialized to 0;
- Solution:

```
do {  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
    /* critical section */  
    lock = 0;  
    /* remainder section */  
} while (true);
```

- What about bounded waiting?
- Solution results in busy waiting.



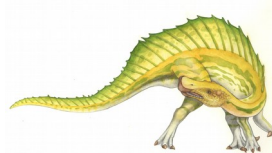


Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest tool is the **Mutex lock**, which has a Boolean variable “available” associated with it to indicate if the lock is available or not.
- Two operations available to access a Mutex Lock:

- ```
acquire() {
 while (!available)
 ; /* busy wait */
 available = false;
}
```

- ```
release() {  
    available = true;  
}
```





Mutex Locks (Cont.)

- Calls to **acquire()** and **release()** are atomic
 - Usually implemented via hardware atomic instructions
- Usage:

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (true);
```
- Solution requires **busy waiting**
 - This lock is therefore called a **spinlock**





Semaphores

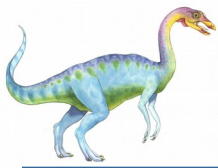
- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations
 - **wait()** and **signal()**
 - ▶ Originally called **P()** and **V()**
- Definition of the **wait()** operation

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S = S - 1;  
}
```

- Definition of the **signal()** operation

```
signal(S) {  
    S = S + 1;  
}
```





Semaphore Usage

Can solve various synchronization problems

■ A solution to the CS problem.

- Create a semaphore “**synch**” initialized to 1

```
wait(synch)
```

```
CS
```

```
signal(synch);
```

■ Consider P_1 and P_2 that require code segment S_1 to happen before code segment S_2

- Create a semaphore “**synch**” initialized to 0

```
P1:
```

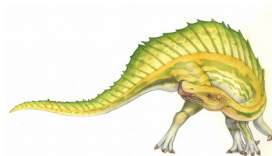
```
S1;
```

```
signal(synch);
```

```
P2:
```

```
wait(synch);
```

```
S2;
```





Types of Semaphores

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
 - Same as a **mutex lock**
- Can implement a counting semaphore **S** as a binary semaphore





Counting Semaphores Example

- Allow at most two process to execute in the CS.
- Create a semaphore “**synch**” initialized to 2

```
wait(synch)
```

```
CS
```

```
signal(synch) ;
```





Semaphore Implementation

- Must guarantee that no two processes can execute the **wait()** and **signal()** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **wait** and **signal** code are placed in the critical section
- This implementation is based on **busy waiting** in critical section implementation (that is, the code for **wait()** and **signal()**)
 - But implementation code is short
 - Little busy waiting if critical section rarely occupied
- Can we implement semaphores with no busy waiting?





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)
- pointer to next record in the list

```
typedef struct{  
    int value;  
    struct process *list;  
} semaphore;
```

- Two operations:
 - **block ()** – place the process invoking the operation on the appropriate waiting queue
 - **wakeup (P)** – remove one of processes in the waiting queue and place it in the ready queue





Implementation with no Busy waiting (Cont.)

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

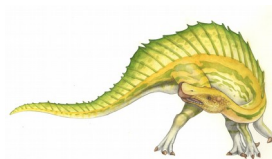
■ signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```





Implementation with no Busy waiting (Cont.)

- Does the implementation ensure the “progress” requirement?
- Does implementation ensure the “bounded waiting” requirement?





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	P_1
<code>wait(S);</code>	<code>wait(Q);</code>
<code>wait(Q);</code>	<code>wait(S);</code>
<code>...</code>	<code>...</code>
<code>signal(S);</code>	<code>signal(Q);</code>
<code>signal(Q);</code>	<code>signal(S);</code>

- **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**





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 are possible.
- Solution – create high-level programming language constructs





Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { ... }

    procedure Pn (...) {.....}

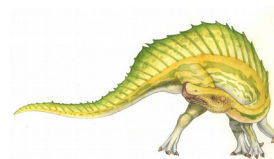
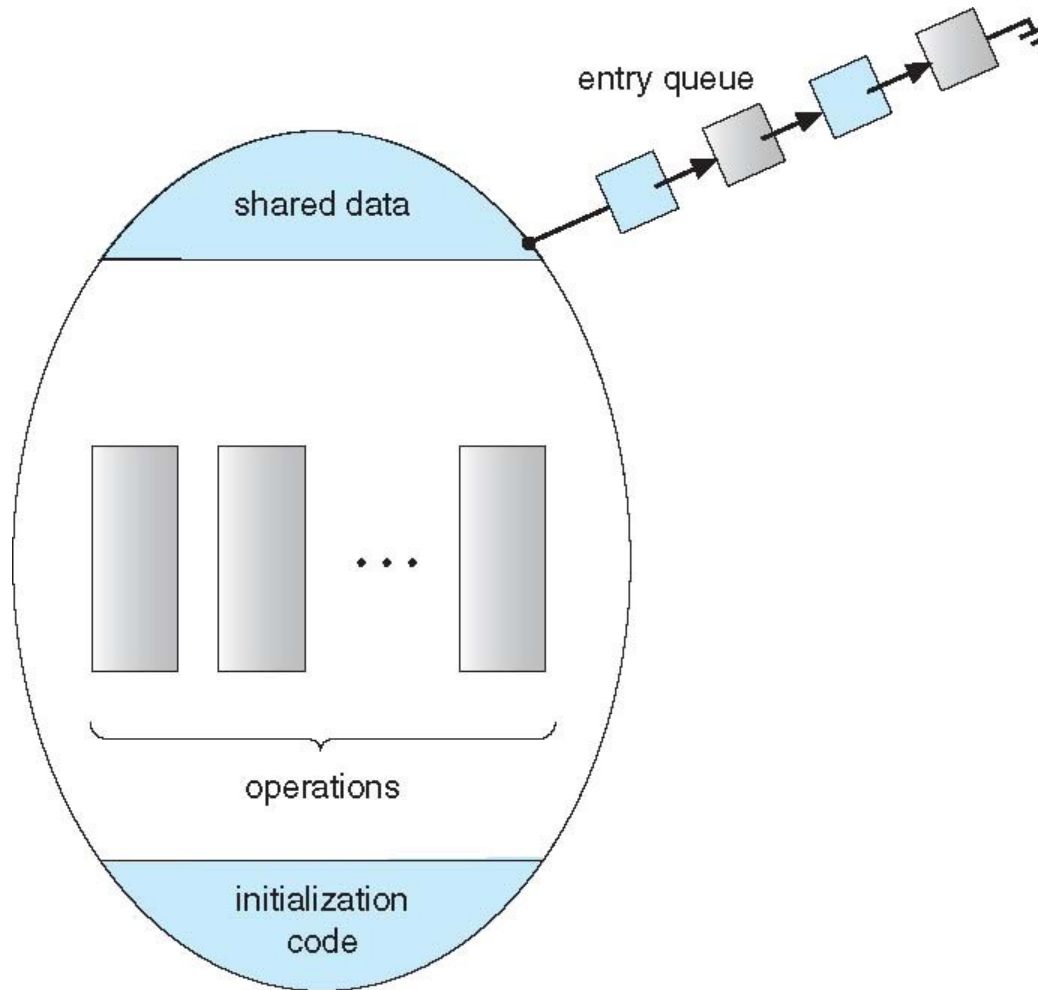
    Initialization code (...) { ... }
}
```

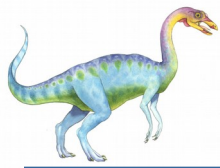
- Mutual exclusion is guaranteed by the compiler.





Schematic view of a Monitor

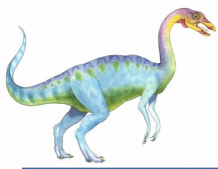




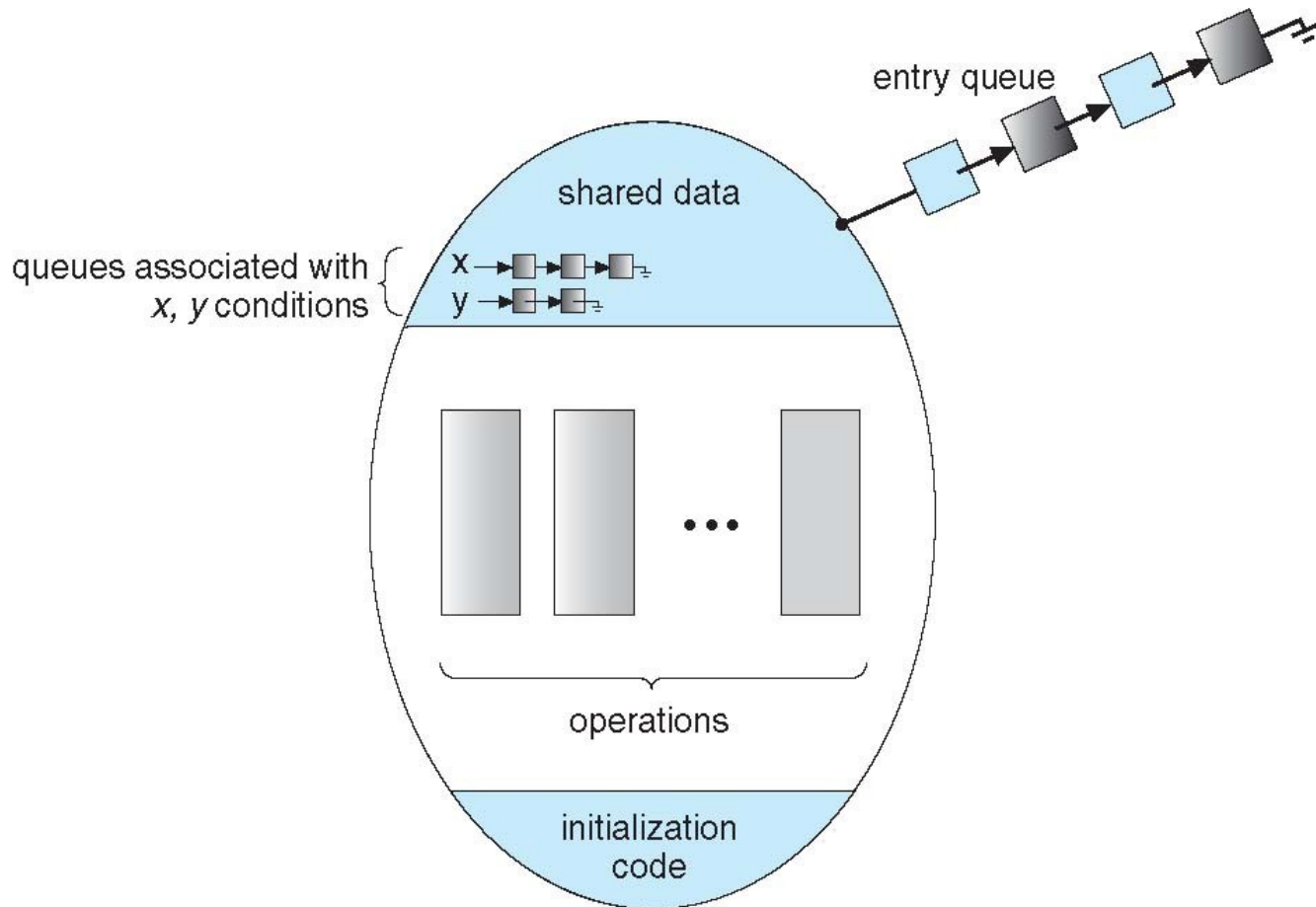
Condition Variables

- Need mechanism to allow a process wait within a monitor
- Provide condition variables.
- A condition variable (say **x**) can be accessed only via two operations:
 - **x.wait()** – a process that invokes the operation is suspended until another process invoked **x.signal()**
 - **x.signal()** – resumes one of processes (if any) that invoked **x.wait()**
 - ▶ If no process is suspended on variable **x** , then it has no effect on the variable





Monitor with Condition Variables





Condition Variables Choices

- If process P invokes **`x.signal()`** , and process Q is suspended in **`x.wait()`** , what should happen next?
 - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include:
 - **Signal and wait** – P either waits until Q leaves the monitor or it waits for another condition
 - **Signal and continue** – Q either waits until P leaves the monitor or it waits for another condition





Condition Variables Choices (Cont.)

- There are reasonable arguments in favor of adopting either option.
 - Since P was already executing in the monitor, the signal-and-continue method seems more reasonable.
 - However, if we allow P to continue, by the time Q is resumed, the logical condition for which Q was waiting may no longer hold.
- A compromise between these two choices was adopted in the language Concurrent Pascal. When P executes the signal operation, it immediately leaves the monitor. Hence, Q is immediately resumed.





Languages Supporting the Monitor Concept

- Many programming languages have incorporated the idea of the monitor as described in this section, including Java and C#.
- Other languages such as Erlang provide concurrency support using a similar mechanism.





Monitor Implementation

- For each monitor, a semaphore **mutex** is provided.
- A process must execute **wait(mutex)** before entering the monitor and must execute **signal(mutex)** after leaving the monitor. This is ensured by the compiler.
- We use the “signal and wait” mechanism to handle the signal operation.
- Since a signaling process must wait until the resumed process either leaves or it waits, an additional semaphore, **next**, is used.
- The signaling processes can use **next** to suspend themselves.
- An integer variable **next_count** is provided to count the number of processes suspended on **next**





Monitor Implementation (Cont.)

- Variables

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

- Each procedure F will be replaced by

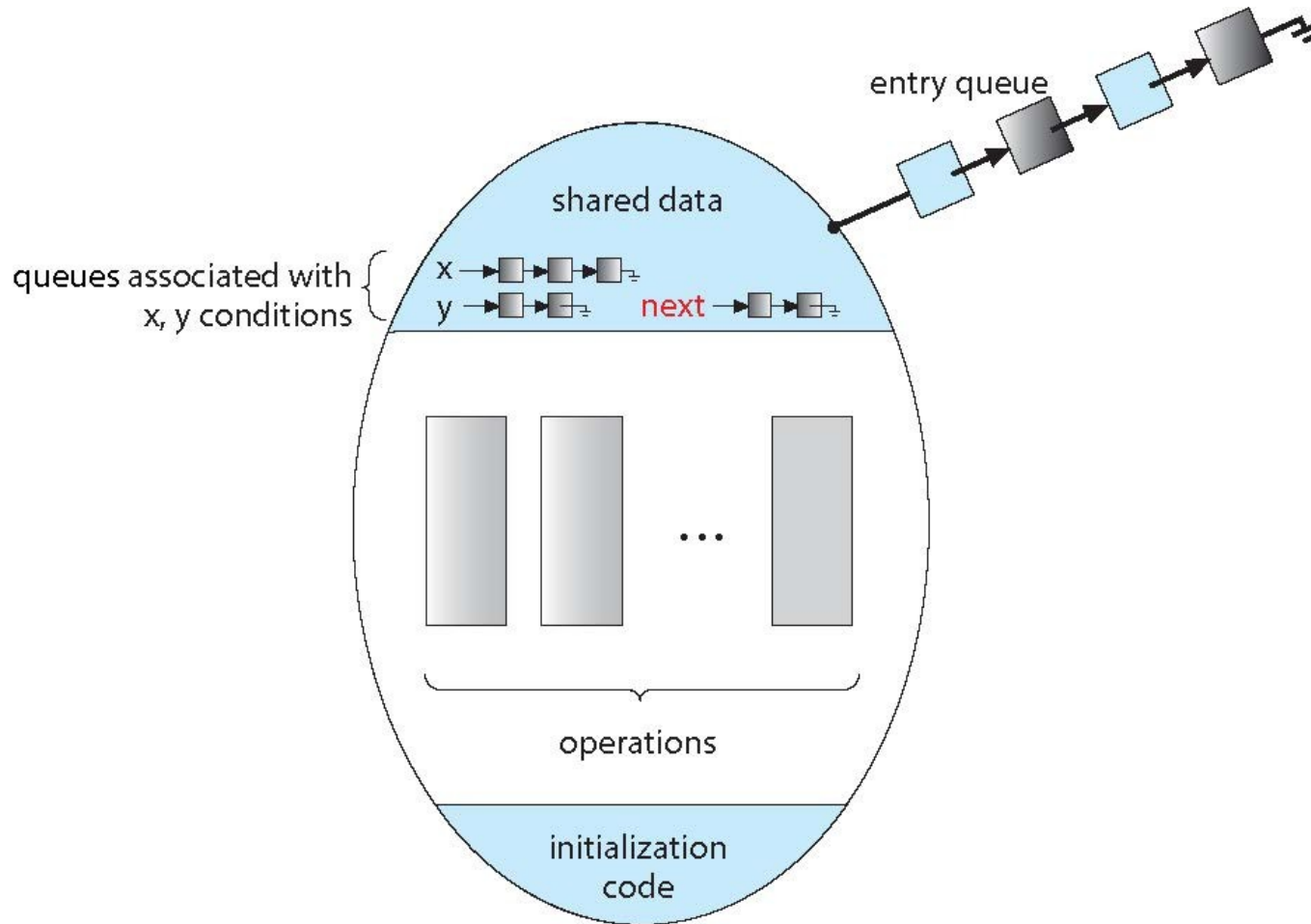
```
wait(mutex) ;
...
body of F;
...
if (next_count > 0)
    signal(next)
else
    signal(mutex) ;
```

- Mutual exclusion within a monitor is ensured





Monitor with Next Semaphore





Condition Variables Implementation

- For each condition variable x , we have:

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





Condition Variables Implementation (Cont.)

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

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





Resuming Processes within a Monitor

- If several processes are queued on condition ***x***, and ***x.signal()*** is executed, which one should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form ***x.wait(c)***
 - Where ***c*** is **priority number**
 - Process with lowest number (low number → highest priority) is scheduled next
- Some languages provide a mechanism to find out the PID of the executing process.
 - In C we have **getpid()**, which returns the PID of the calling process





Resource Allocator Monitor Example

- A monitor to allocate a single resource among competing processes
- Each process, when requesting an allocation of this resource, specifies the maximum time it plans to use the resource. The monitor allocates the resource to the process that has the shortest time-allocation request.

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;

    void acquire(int time) {
        if (busy)
            x.wait(time)
        busy = true;
    }

    void release () {
        busy = false;
        x.signal();
    }

    busy= false;
}
```





Resource Allocator Monitor Example (Cont.)

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

```
R.acquire(t) ;  
    . . .  
    access the resource ;  
    . . .  
R.release() ;
```

where R is an instance of type ResourceAllocator





Observation the Resource Allocator Example

- Incorrect use of the operations:
 - R.release R.acquire(t)
 - R.acquire(t) R.acquire(t)
 - Omitting of acquire and or release (or both)
- Solution exist but not covered in this course



End of Chapter 6

