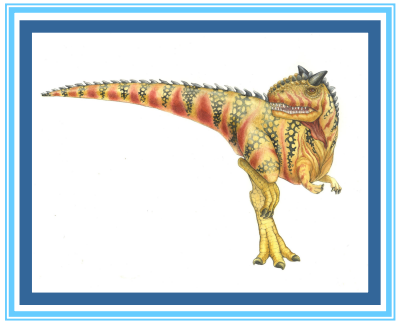
Chapter 6: Synchronization Tools



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

- A **cooperating process** is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages.
- Concurrent access to shared data may result in data inconsistency
- various mechanisms to maintain the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.
- The synchronization mechanism is usually provided by both hardware and the operating system
- Basic assumption – load and store instructions are atomic.
- The basic assumption: the CPU scheduler switches rapidly between processes to provide concurrent execution that is one process may only partially complete execution before another process is scheduled.
- The processes can execute concurrently or in parallel.
- Illustration of the problem – The producer-Consumer problem, which we introduced in Chapter 3.

Producer-Consumer Problem

- Taking an example -> consider a developed a model of a system consisting of cooperating sequential processes or threads, all running asynchronously and possibly sharing data. Consider the producer–consumer problem for this
- we described how a bounded buffer could be used to enable processes to share memory. The original solution allowed at most BUFFER SIZE − 1 items in the buffer at the same time.
- 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.
- Suppose we want to modify the algorithm to remedy this deficiency.

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 integer variable -- **`counter`** that keeps track of the number of full buffers, initially, 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.
- Although the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently.
- As an illustration, suppose that the value of the variable counter is currently 5 and that the producer and consumer processes concurrently execute the statements "counter++" and "counter--". Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6! The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately.
- Code is shown in next 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;
}

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

- We can show that the value of counter may be incorrect as follows.
- The statement "counter++" may be implemented in machine language as follows:

```
register1 = counter   //register1 - local CPU registers
register1 = register1 + 1
counter = register1
```

- Similarly for **counter--** could be implemented as

```
register2 = counter   //register2 - local CPU register
register2 = register2 - 1
counter = register2
```

- The concurrent execution of "counter++" and "counter--" is equivalent to a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order. Consider this execution interleaving with "count = 5" initially:

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

- We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently.

Race Condition (Cont.)

- A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**.
- To guard against the race condition, we ensure that only one process at a time can be manipulating the variable counter.
- To make such a guarantee, we require that the cooperating processes be synchronized and coordinated in some way.
- 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 $\{P_0, P_1, \ldots P_{n-1}\}$
- Each process has a segment of code called **critical section**
  - Process may be changing common variables, updating table, writing file, etc
  - When one process is executing in critical section, no other process is allowed to execute 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 than 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.
- The entry section and exit section are enclosed in boxes to highlight these important segments of 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);
```

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.

- Assume that each process executes at a nonzero speed
- No assumption concerning **relative speed** of the ***n*** processes
- At a given point in time, many kernel-mode processes may be active in the operating system. As a result, the code implementing an operating system (***kernel code***) is subject to several possible race conditions.
- Consider as an example a kernel data structure that maintains a list of all open files in the system. This list must be modified when a new file is opened or closed (adding the file to the list or removing it from the list). If two processes were to open files simultaneously, the separate updates to this list could result in a race condition. Other kernel data structures that are prone to possible race conditions include structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling.

Two general approaches are used to handle critical sections in operating systems:

preemptive kernels - A preemptive kernel allows a process to be preempted while it is running in kernel mode. Preemptive kernels are especially difficult to design for SMP architectures, since in these environments it is possible for two kernel-mode processes to run simultaneously on different processors.

non preemptive kernels - A non preemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU. A non preemptive kernel is free from race conditions on kernel data structures, as only one process is active in the kernel at a time. We cannot say the same about preemptive kernels, so they must be carefully designed to ensure that shared kernel data are free from race conditions.

Why, then, would anyone favor a preemptive kernel over a non preemptive one?

A preemptive kernel may be more responsive, since there is less risk that a kernel-mode process will run for an arbitrarily long period before relinquishing the processor to waiting processes. (Of course, this risk can also be minimized by designing kernel code that does not behave in this way.) Furthermore, a preemptive kernel is more suitable for real-time programming, as it will allow a real-time process to preempt a process currently running in the kernel. Later in this chapter, we explore how various operating systems manage preemption within the kernel.

Solution to Critical-Section Problem must satisfy following conditions

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 only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section and 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
4. Some Assumptions
   - Load, test and store machine-language instructions are atomic - if two such instructions are executed concurrently, the result is equivalent to their sequential execution in some unknown order

Software Solution for 2 Processes P_i (P0 and P1)

- Keep a variable "turn" initialized to 0 (or 1) shared between the processes to indicate which process is next

```
do {
 while (turn != i);
     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" - does not satisfy the progress requirement
- Requires strict alternation of processes in the execution of the critical section. For example, if turn == 0 and ***PI*** is ready to enter its critical section, ***PI*** cannot do so, even though ***Po*** may be in its remainder section.
- Also The problem with algorithm 1 is that it does not retain sufficient information about the state of each process; it remembers only which process is allowed to enter its critical section.

Algorithm 2

- replace the variable turn with the following array: boolean flag[2], initialized to false. If flag[i] is true, this value indicates that ***Pi*** is ready to enter the critical section.

```
do {

    flag[i] = true;
    while (flag[j]);

        critical section

    flag[i] = false;

        remainder section

} while (1);
```

Software Solution: Peterson's Algorithm

- The two processes share two variables: **`int turn; and 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 to enter its critical section!

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

     critical section

    flag[i] = false;

     remainder section

} while (true);
```

Peterson's Solution (Cont.)

- To enter the critical section, process *Pi* first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. But one of these assignments will last; the other will occur but will be overwritten immediately. The eventual value of turn determines which of the two processes is allowed to enter its critical section first.
- What about a solution to N > 2 processes
- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

- To prove property 1,
  - each *Pi* enters its critical section only if either flag[j] == false or turn == i.
  - Let both processes are executing in their critical sections at the same time, then flag[0] == flag[1] == true => that *P*0 and *P*1 could not have successfully executed their while statements at about the same time,
  - turn can be either 0 or 1 but cannot be both. Hence, one of the processes —say, *Pj*—must have successfully executed the while statement, whereas *Pi* had to execute at least one additional statement ("turn == j"). However, at that time, flag[j] == true and turn == j, and this condition will persist as long as *Pj* is in its critical section; as a result, mutual exclusion is preserved.

- To prove properties 2 and 3,
  - A process *Pi* can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] == true and turn == j;
  - If *Pj* is not ready to enter the critical section, then flag[j] == false, and *Pi* can enter its critical section.
  - If *Pj* has set flag[j] to true and is also executing in its while statement, then either turn == i or turn == j. If turn == i, then *Pi* will enter the critical section.
  - If turn == j, then *Pj* will enter the critical section.
  - However, once *Pj* exits its critical section, it will reset flag[j] to false, allowing *Pi* to enter its critical section.
  - If *Pj* resets flag[j] to true, it must also set turn to i.

  Thus, since *Pi* does not change the value of the variable turn while executing the while statement, *Pi* will enter the critical section (progress) after at most one entry by *Pj* (bounded waiting).

```
do {
    choosing[i] = true;
    number[i] = max(number[0], number[1], ... , number[n-1]) + 1;
    choosing[i] = false;
    for (j=0; j < n; j++) {
        while (choosing[j]);
        while ((number[j] != 0) && (number[j,j] < number[i,i]));
    }

        critical section

    number[i] = 0;

        remainder section

} while(1);
```

Hardware Synchronization – Solution to Critical-section Problem Using Locks

- Peterson's are not guaranteed to work on modern computer architectures
- Many systems provide hardware support for implementing the critical section code.
- All solutions are based on idea of **locking -** that is, protecting critical regions through the use of locks.
  - 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 that can be used effectively in solving the critical-section problem.
- easier and improve system efficiency.
  - **Atomic** = non-interruptible
    - The critical-section problem could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified. In this way, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption.
    - Unfortunately, this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, as the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases.
    - Also consider the effect on a system's clock if the clock is kept updated by interrupts.
- special hardware instructions are provided that allow us either to test and modify the content of a word or to swap the contents of two words **atomically- relatively simpler in use**
  - 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 - Thus, if two **`test and set()`** instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order. If the machine supports the **`test and set()`** instruction, then we can implement mutual exclusion by declaring a boolean variable **`lock`**, initialized to **`false`**.
  - 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?
- This Solution results in busy waiting.

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
- operates on three operands;
- The swap takes place only under this condition
- Regardless, always returns the original value of passed parameter value

- Mutual exclusion can be provided as follows:
  - a global variable `(lock)` is declared and is initialized to 0. The first process that invokes `compare and swap()` will set `lock` to 1 and enter its critical section,
  - Subsequent calls to `compare and swap()` will not succeed, because `lock` now is not equa to the expected value of 0. When a process exits its critical section, it sets `lock` back to 0, which allows another process to enter its critical section.

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.

Bounded-waiting Mutual Exclusion with test_and_set

- another algorithm using the test and set() instruction that satisfies all the critical-section requirements. The common data structures are
  - boolean waiting[n]; boolean lock;
  - These data structures are initialized to false.

```
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);
```

- To prove that the mutual exclusion requirement is met, we note that process *Pi* can enter its critical section only if either waiting[i] == false or key == false. The value of key can become false only if the test and set() is executed. The first process to execute the test and set() will find key == false; all others must wait. The variable waiting[i] can become false only if another process leaves its critical section; only one waiting[i] is set to false ,maintaining the mutual-exclusion requirement.
- To prove that the progress requirement is met, we note that the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets lock to false or sets waiting[j] to false. Both allow a process that is waiting to enter its critical section to proceed. To prove that the bounded-waiting requirement ismet,we note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering ($i + 1, i + 2, ..., n - 1, 0, ..., i - 1$). It designates the first process in this ordering that is in the entry section (waiting[j] == true) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within $n - 1$ turns.

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest tools is the **Mutex lock,** which has a Boolean variable "available" associated with it to indicate if the lock is available or not.
- We use the mutex lock to protect critical regions and thus prevent race conditions.
- That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section.
- The acquire()function acquires the lock, and the release() function releases the lock
- Two operations available to access a Mutex Lock:

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

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

- If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable. A process that attempts to acquire an unavailable lock is blocked until the lock is released.

Mutex Locks (Cont.)

- Calls to **`acquire()`** and **`release()`** are performed 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**

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"
- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire(). In fact, this type of mutex lock is also called a **spinlock** because the process "spins" while waiting for the lock to become available. the test and set() instruction and the compare and swap() instruction. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes.
- Busy waiting is a pure waste of CPU cycles that some other process might be able to use productively
- Spinlocks do have an advantage, however, in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful. They are often employed on multiprocessor systems where one thread can "spin" on one processor while another thread performs its critical section on another processor.

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

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

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

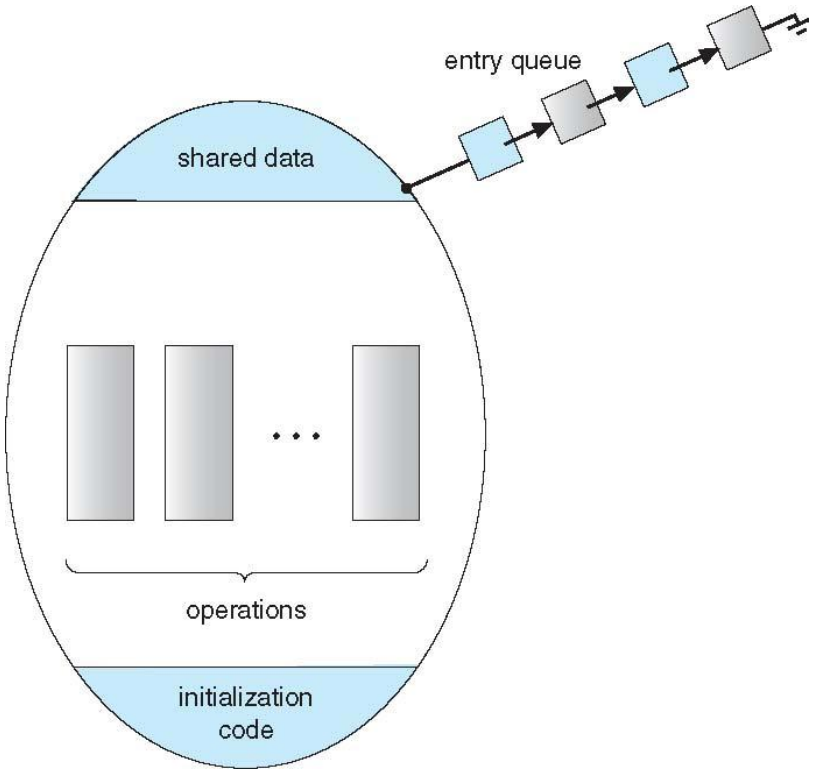
```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (…) { …. }

    procedure Pn (…) {……}

        Initialization code (…) { … }
}
```

- Mutual exclusion is guaranteed by the compiler.

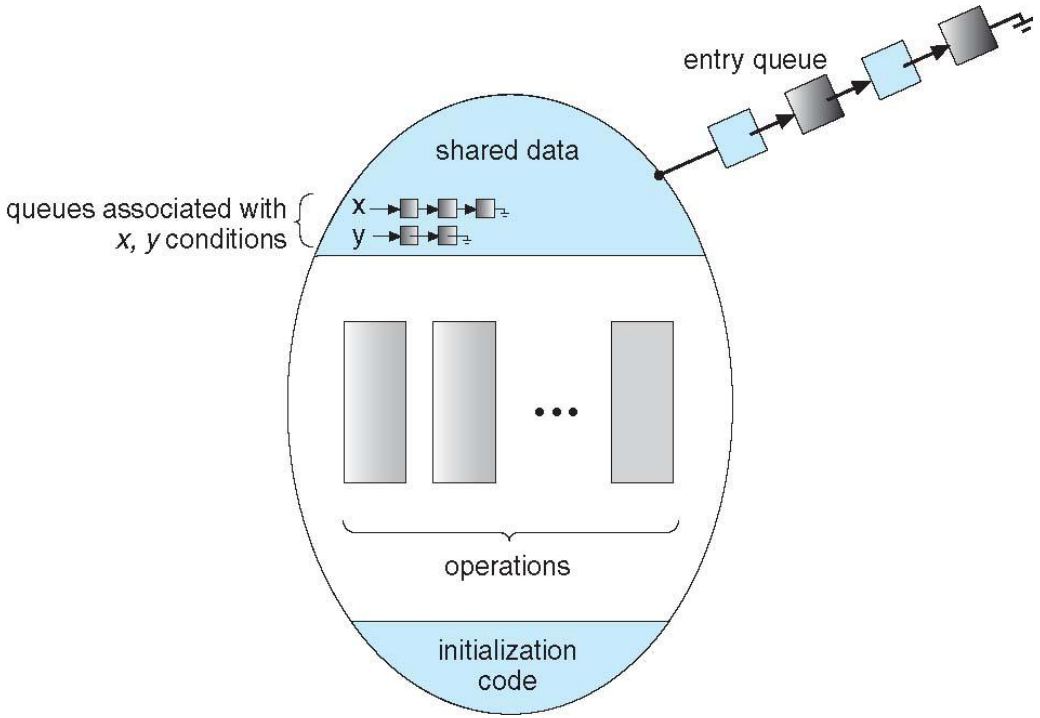
Schematic view of a Monitor



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()**
    4. 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

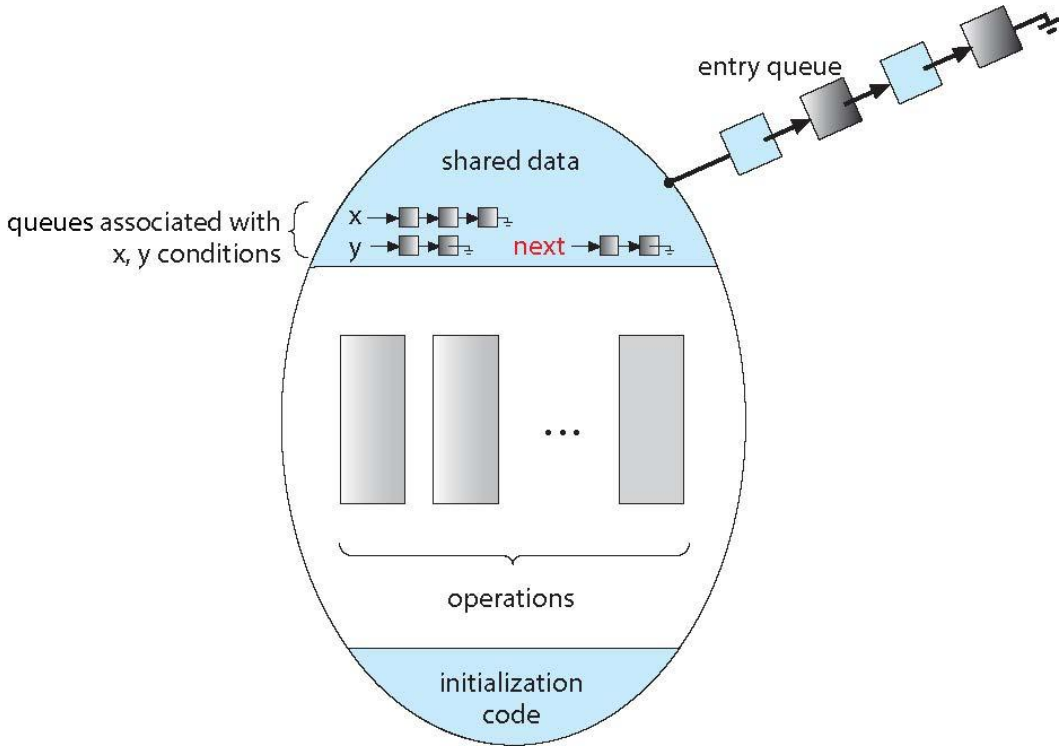
```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0;
```

- 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.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 retunes 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 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