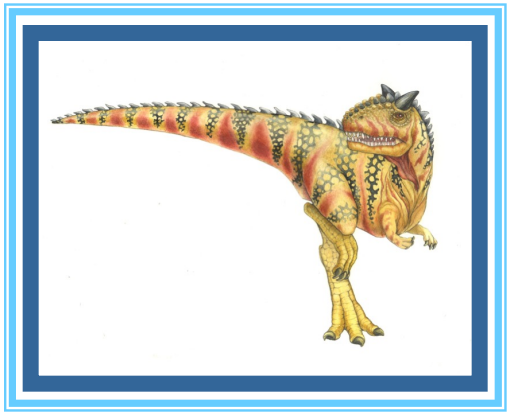
Chapter 6: Synchronization Tools



Objectives

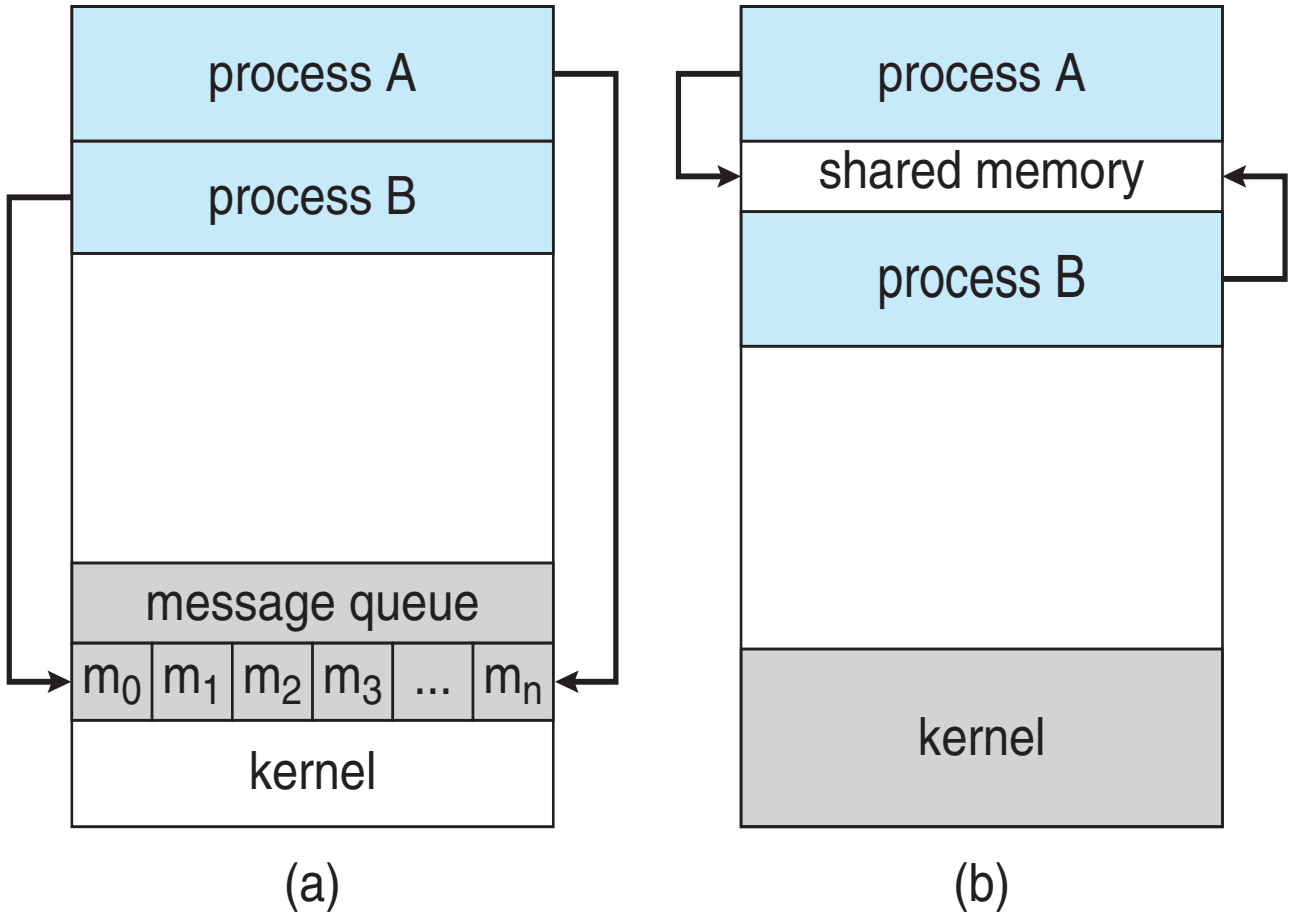
- To present the concept of process synchronization.
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- **Illustration of the problem:**
  - Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers.
  - We can do so by having an integer **`counter`** that keeps track of the number of full buffers. Initially, **`counter`** 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.

Recap: Cooperating Processes

(a) Message passing. (b) shared memory.



Producer

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

Consumer

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

Race Condition

- **`counter++`** could be implemented as

```
register1 = counter //load counter to register 1
register1 = register1 + 1
counter = register1 //store register 1 to counter
```

- **`counter--`** could be implemented as

```
register2 = counter // load counter to register 2
register2 = register2 - 1
counter = register2 // store register 2 to counter
```

- Consider this execution interleaving with "counter = 5" initially:

Time 0: producer execute `register1 = counter` {register1 =5}
Time 1: producer execute `register1 = register1 + 1` {register1 =
Time 2: interrupt on Producer
Time 3: consumer execute `register2 = counter` {register2 = 5}
Time 4: consumer execute `register2 = register2 - 1` {register2 = 4}
Time 5: interrupt on Consumer
Time 6: producer execute `counter = register1` {counter = 6 }
Time 7: consumer execute `counter = register2` {counter = 4}

Let's Trace The Race Condition on the Producer Consumer example

Suppose counter is located at Memory #100 and counter = 5
1) Producer executes counter++
2) Consumer executes counter--

Time		Producer	Consumer	Register 1 (R1)	Register 2 (R2)	Memory #100 (M100)
0		load R1 M100		5		**5**
1		add R1 by 1		6		
2	interrupt					
3			load R2 M100		5	
4			sub R1 by 1		4	
5	interrupt					
6		storen R1 M100				6
7	interrupt					
8			storen R2 M100			4

since there is producer+consumer, so there is inconsistence

Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … 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**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i

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

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 only one process can executing in critical section at once
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

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

Peterson's Solution

- Good algorithmic description of solving the problem
- 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);
```

meaning i is ready to enter critical section

pi will wait until it is false, otherwise it would not enter critical section

Algorithm for Process

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

P_i

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

P_j

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**

     Process P_i can be prevented from entering its critical section only if it is stuck in while loop. If P_j does not want to be in critical section it will set flag[j]=False allowing P_i to enter.

  3. **Bounded-waiting requirement is met**

     Since P_i does not change the value of the variable turn while in its while statement, P_i will enter the critical section (progress) after at most one entry by P_j

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

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

test_and_set Instruction

Definition:

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

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

Solution using test_and_set()

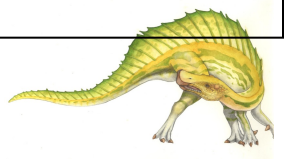
```
boolean test_and_set (boolean *target)
        {
                boolean rv = *target;
                *target = TRUE;
                return rv:
        }
```

P_1

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

P_2

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



Test your understanding

- In the producer and consumer programs, what are the critical sections?

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

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

Solution to Critical-section Problem Using Locks

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

S

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

acquire() and release()

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

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

acquire() and release()

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

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

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

Test your understanding

- There is the shared global variable `double balance` that represents the balance in a banking account. There are multiple threads that may concurrently call the `deposit()` and `withdraw()` functions.

```
double deposit(double amount) {
  balance += amount;
  return balance;
}

double withdraw(double amount) {
  if (amount <= balance) {
    balance -= amount;
  }
  return balance;
}
```

There is a race condition on the shared variable balance.

Test your understanding

check it on Thur

Which of the following code statements correctly fixes the race condition in the deposit() function using a mutex lock with the acquire() and release() operations?

a

```
double deposit(double amount) {

  balance += amount;

  return balance;

}
```

b

```
double deposit(double amount) {

  acquire();

  balance += amount;

  return balance;
}
```

c

```
double deposit(double amount) {

  acquire();

  balance += amount;

  release();

  return balance;
}
```

d

```
double deposit(double amount) {

  acquire();

  balance += amount;

  return balance;

  release();
}
```

Test your understanding

Which of the following code statements correctly fixes the race condition in the withdraw() function using a mutex lock with the acquire() and release() operations?

a

```
double withdraw(double amount) {

  if (amount <= balance) {
    acquire();
    balance -= amount;
    release();
  }

  return balance;
}
```

b

```
double withdraw(double amount) {

  acquire();

  if (amount <= balance) {
    balance -= amount;

    release();
  }

  return balance;
}
```

c

```
double withdraw(double amount) {

  if (amount <= balance) {
    acquire();
    balance -= amount;
  }

  release();

  return balance;
}
```

d

```
double withdraw(double amount) {

  acquire();

  if (amount <= balance) {
    balance -= amount;
  }

  release();

  return balance;
}
```

Semaphore

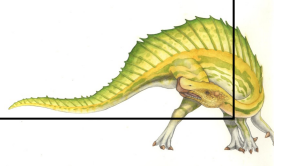
- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** **and** **`V()`**
    - **`P for 'proberen' (to test)`**
    - **`V() for 'verhogen' (to increment)`**

Definition of the **`wait()` operation**

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

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

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



Semaphore Usage

- **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 solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "`synch`" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

- Can implement a counting semaphore ***S*** as a binary semaphore
- Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal(synch), which is after statement S1 has been executed

Test your understanding

To use a binary semaphore to solve the critical section problem, the semaphore must be initialized to 1 and _____ must be called before entering a critical section and _____ must be called after exiting the critical section.

a ○ `wait(), signal()`

b ○ `signal(), wait()`

c ○ Mutex locks - not semaphores - are used to solve the critical section problem.

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
  - Could now have **busy waiting** in critical section implementation
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- Busy waiting process can be suspended
- The suspend operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state.
- CPU scheduler can then bring another process into execution
- When some other process execute Signal, the waiting process can restart.
- 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

- Each semaphore has two data items:
  - value (of type integer)
  - pointer to next record in the list

- ```
  typedef struct{
   int value;
   struct process *list; // a pointer to a list of PCBs.
   } semaphore;
  ```


  ```

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 (Example)

```
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);
    }
}
```

Suppose there are 5 Processes:
A, B, C, D, E
that share a global variable
They require a semaphore, with initial value = 3
init S->value = 3

Implementation with no Busy waiting (Example)

```
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);
    }
}
```

When A is ready to enter its CS:
S->value = 2

When B is ready to enter its CS:
S->value = 1

When C is ready to enter its CS:
S->value = 0

When D is ready to enter its CS:
S->value = -1
S->list: D
Block D

When E is ready to enter its CS:
S->value = -2
S->list: D -> E
Block E

Implementation with no Busy waiting (Example)

```
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);
    }
}
```

When A exits its CS
S->value = -1
S->list: E
Wakeup (D)

When B exits its CS
S->value = 0
Wakeup(E)

When C exits its CS
S->value = 1

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
- **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); … critical section … wait(mutex);
    - In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement.
  - wait(mutex); … critical section … wait(mutex);
    - In this case, the process will permanently block on the second call to wait(), as the semaphore is now unavailable.
  - A process omits the wait(mutex), or the signal(mutex), or both
- Deadlock and starvation are possible.

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

```
monitor monitor name
{
  /* shared variable declarations */

  function P1 ( . . . ) {
    . . .
  }

  function P2 ( . . . ) {
    . . .
  }

        .
        .
        .
  function Pn ( . . . ) {
    . . .
  }

  initialization_code ( . . . ) {
    . . .
  }
}
```

Monitors

- A procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.
- The local variables of a monitor can be accessed by only the local procedures.
- Only one process may be active within the monitor at a time

```
monitor monitor name
{
  /* shared variable declarations */

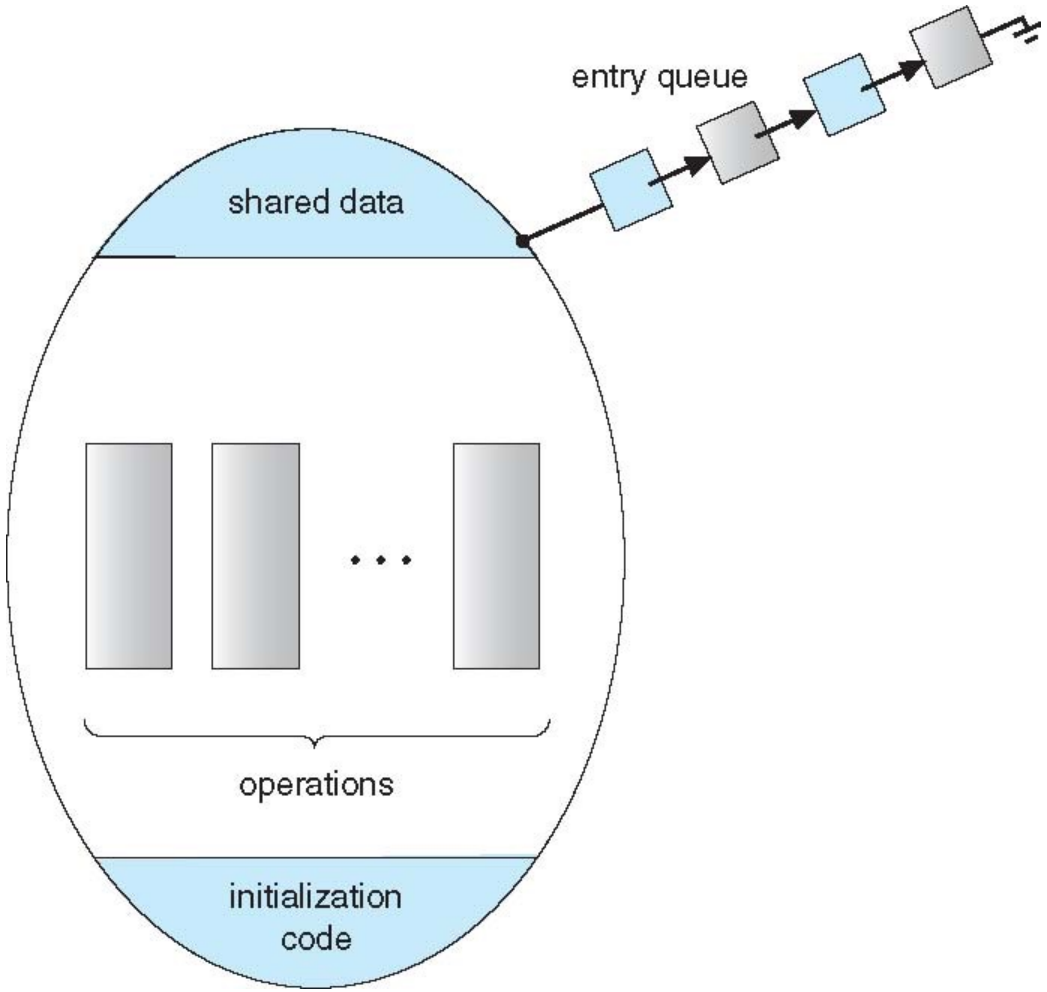
  function P1 ( . . . ) {
    . . .
  }

  function P2 ( . . . ) {
    . . .
  }

        .
        .
        .
  function Pn ( . . . ) {
    . . .
  }

  initialization_code ( . . . ) {
    . . .
  }
}
```

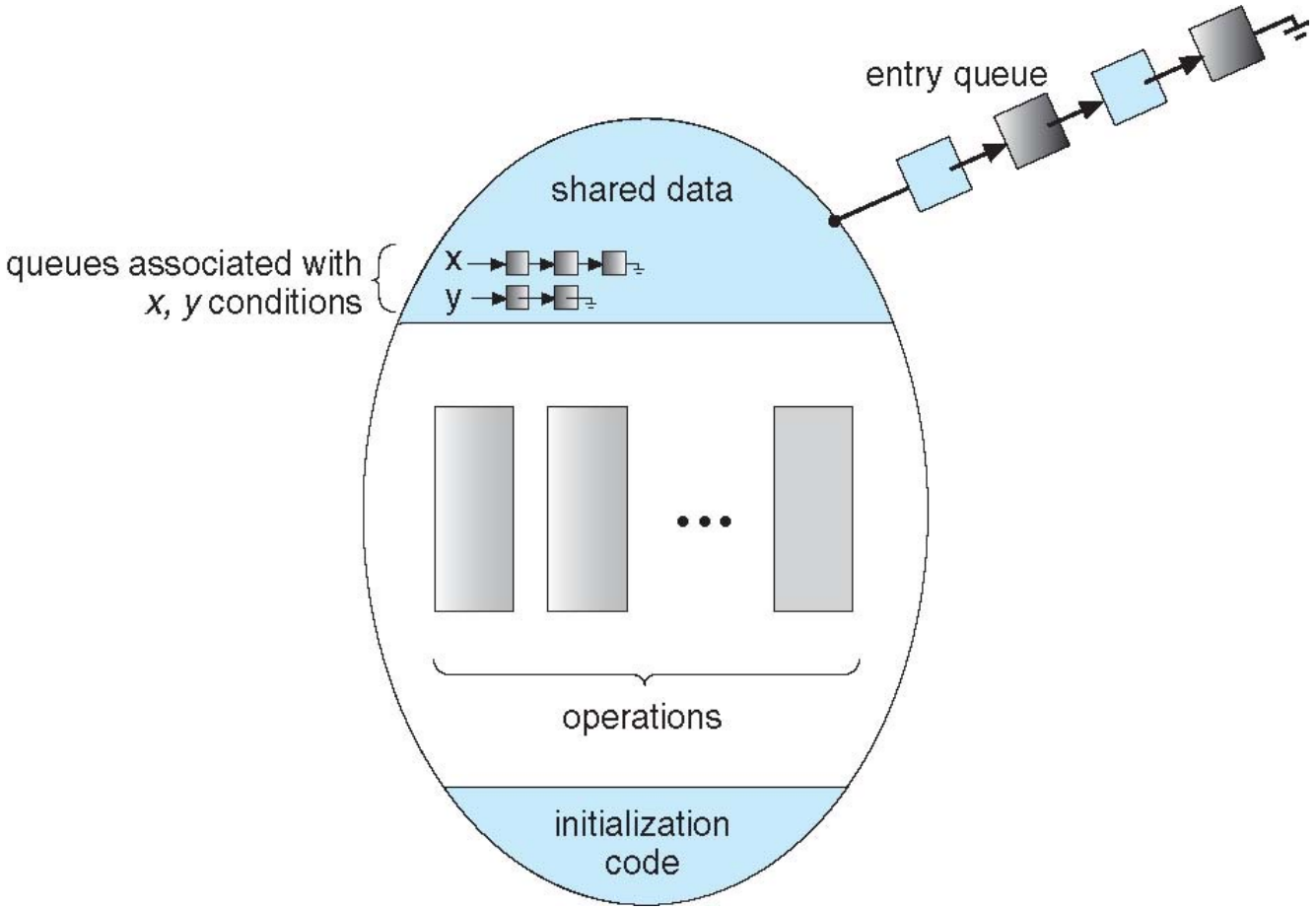
Schematic view of a Monitor



Condition Variables

- `condition x, y;`
- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is suspended until **`x.signal()`**
  - **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`**
    - If no **`x.wait()`** on the variable, 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 paralel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

Example: A monitor to allocate a 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;
  }
}
```

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

End of Chapter 6