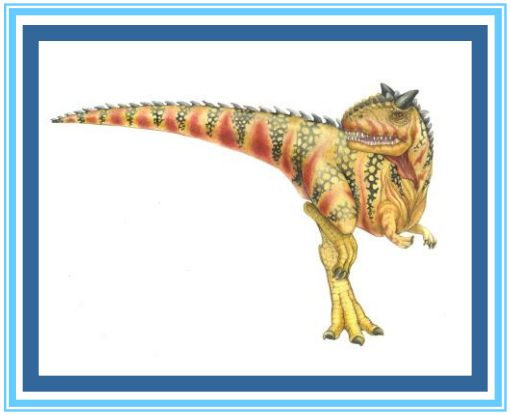
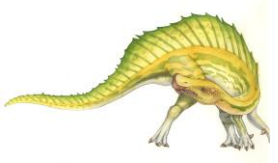
Chapter 5: Process Synchronization



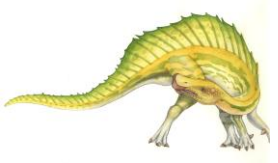
Chapter 5: Process Synchronization

- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches



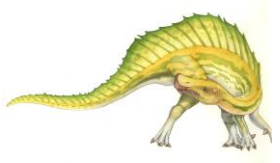
Objectives

- 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



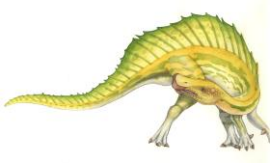
Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- **Concurrent access** to **shared** data may result in **data inconsistency**
- **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.



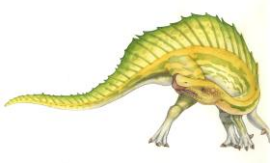
Producer

```
while (true) {
        /* produce an item in next produced */

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



Consumer

```
while (true) {
        while (counter == 0)
                ; /* 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

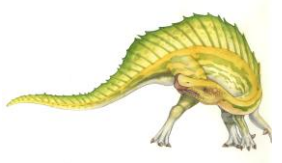
```
register1 = counter
register1 = register1 + 1
counter = register1
```

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

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

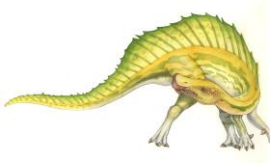
- Consider this execution interleaving with "count = 5" initially:

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}



Critical Section Problem

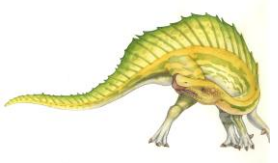
- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design a **protocol** that the processes can use to cooperate.
- Each process must ask **permission** to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**



Critical Section

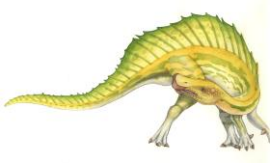
- General structure of process P_i

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



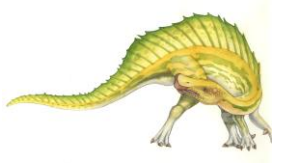
Algorithm for Process P_i

```
do {

    while (turn == j);

          critical section

    turn = j;

          remainder section

 } while (true);
```



Solution to Critical-Section Problem

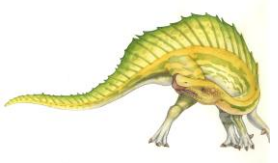
1. **Mutual Exclusion** - If process P_i is executing in its critical section, then **no other processes** can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that **wish to enter** their critical section, then 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

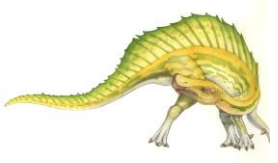
Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive kernels**– allows preemption of process when running in kernel mode
  - **More responsive** but it **must be carefully designed**
- **Non-preemptive kernels** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - **Essentially free of race conditions in kernel mode**



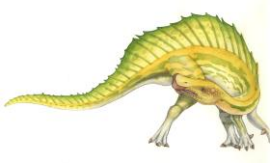
Peterson's Solution

- 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);
```

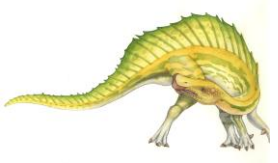


Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

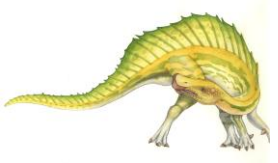
Synchronization Hardware

- 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);
```

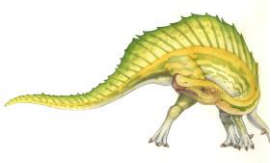


test_and_set Instruction

Definition:

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

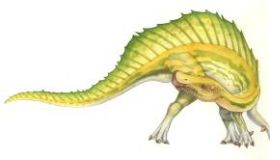
1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.



Solution using test_and_set()

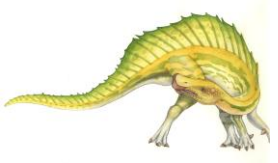
- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */

    /* critical section */

    lock = false;

    /* remainder section */

} while (true);
```



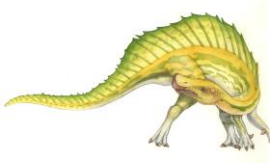
Mutex Locks

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



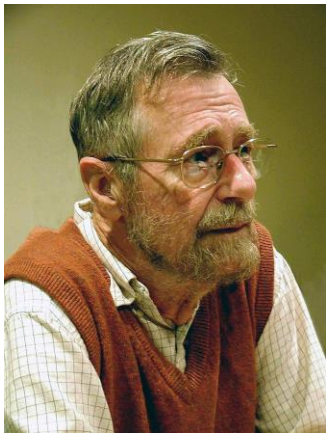
Semaphore

- 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()`**
- 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;
```

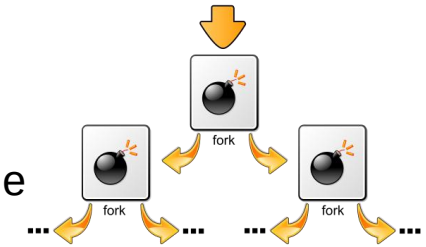
- Can implement a counting semaphore **S** as a binary semaphore

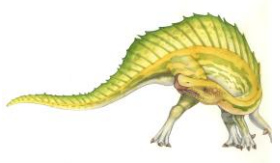
Deadlock and Starvation

- **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**





Semaphore definition

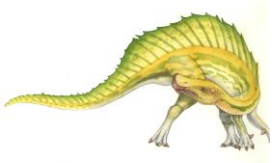
```
typedef struct {
        int value;
        struct process *list;
} semaphore;
```

```
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);
        }
}
```

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem



Bounded-Buffer Problem

- ***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.)

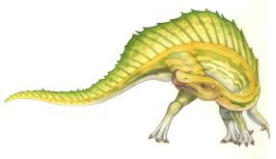
- The structure of the producer process

```
do {
        ...
        /* produce an item in next_produced */
        ...
    wait(empty);
    wait(mutex);
        ...
        /* add next produced to the buffer */
        ...
    signal(mutex);
    signal(full);
} while (true);
```

Bounded Buffer Problem (Cont.)

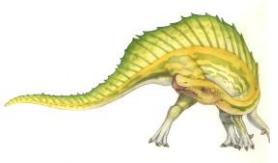
- The structure of the consumer process

```
Do {
    wait(full);
    wait(mutex);
        ...
    /* remove an item from buffer to
next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
    /* consume the item in next consumed */
        ...
} while (true);
```



Readers-Writers Problem

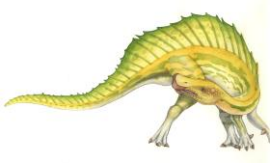
- A data set is shared among a number of concurrent processes
  - Readers – **only read** the data set; they do ***not*** perform any updates
  - Writers – can **both read** and **write**
- **Problem – allow multiple readers to read at the same time**
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0



Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
      wait(rw_mutex);
          ...
      /* writing is performed */
          ...
   signal(rw_mutex);
} while (true);
```



- **Writers**
  - Only one writer writes at a time
  - While writing, reading is not possible
- **Readers**
  - Multiple readers can read concurrently
  - While reading, writing is not possible

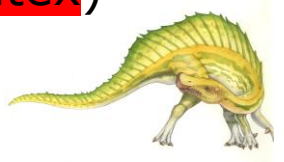
rw_mutex=1
read_count=0
mutex=1

Writer

```
Wait(rw_mutex)
  Write()
Signal(rw_mutex)
```

Reader

```
Wait(mutex)
read_count++;
If(read_count==1)
   Wait(rw_mutex)
Signal(mutex)

Read()

Wait(mutex)
read_count--;
If (read_count==0)
   Signal(rw_mutex)
Signal(mutex)
```



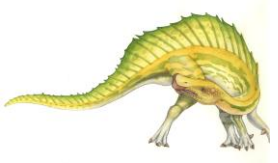
Readers Writers Problem

Writers

Only one writer writes at a time
While writing, reading is not possible

Readers

Multiple readers can read concurrently
While reading, writing is not possible



Readers Writers Problem

mutex=1

Writers

Only one writer writes at a time
While writing, reading is not possible

```
wait(mutex);
write();
signal(mutex);
```

Readers

Multiple readers can read concurrently
While reading, writing is not possible

```
wait(mutex);
read();
signal(mutex);
```

This solution allows only one reader or writer to access shared variable

Readers Writers Problem

mutex=1

Writers

Only one writer writes at a time
While writing, reading is not possible

```
wait(mutex);
write();
signal(mutex);
```

Readers

Multiple readers can read concurrently
While reading, writing is not possible

```
wait(mutex);
read();
signal(mutex);
```

Readers

Multiple readers can read concurrently
While reading, writing is not possible

```
wait(mutex);
read();
signal(mutex);
```

Multiple readers cannot read

Readers Writers Problem

rwmutex=1

Writers

Only one writer writes at a time
While writing, reading is not possible

```
wait(rwmutex);
write();
signal(rwmutex);
```

Readers

Multiple readers can read concurrently
While reading, writing is not possible

Readers

Multiple readers can read concurrently
While reading, writing is not possible

```
readers++;
If (readers==1)
   wait(rwmutex);

read();

readers--;
If(readers==0)
  signal(rwmutex);
```

Multiple readers can read now
However, accessing the «readers» variable is now creating a race condition

Readers Writers Problem

rwmutex=1
mutex=1

Writers
Only one writer writes at a time
While writing, reading is not possible

```
wait(rwmutex);
write();
signal(rwmutex);
```

Readers
Multiple readers can read concurrently
While reading, writing is not possible

Readers
Multiple readers can read concurrently
While reading, writing is not possible

```
wait(mutex);
readers++;
If (readers==1)
   wait(rwmutex);
signal(mutex);
read();

wait(mutex);
readers--;
If(readers==0)
  signal(rwmutex);
signal(mutex);
```

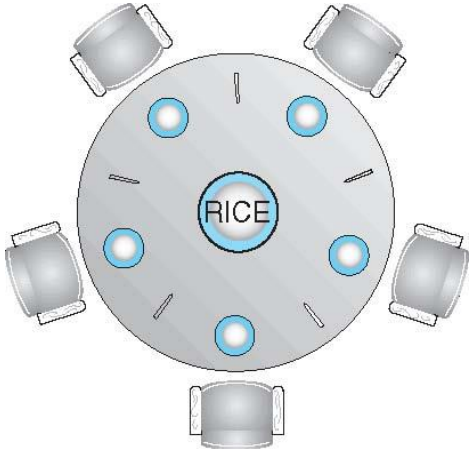
Problem solved

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
        wait(mutex);
            read_count++;
            if (read_count == 1)
                wait(rw_mutex);
        signal(mutex);
          ...
        /* reading is performed */
          ...
        wait(mutex);
            read count--;
            if (read_count == 0)
                signal(rw_mutex);
        signal(mutex);
} while (true);
```

Dining-Philosophers Problem



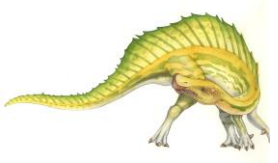
- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (**one at a time**) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- 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);
```

- What is the problem with this algorithm?



Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Problems with Semaphores

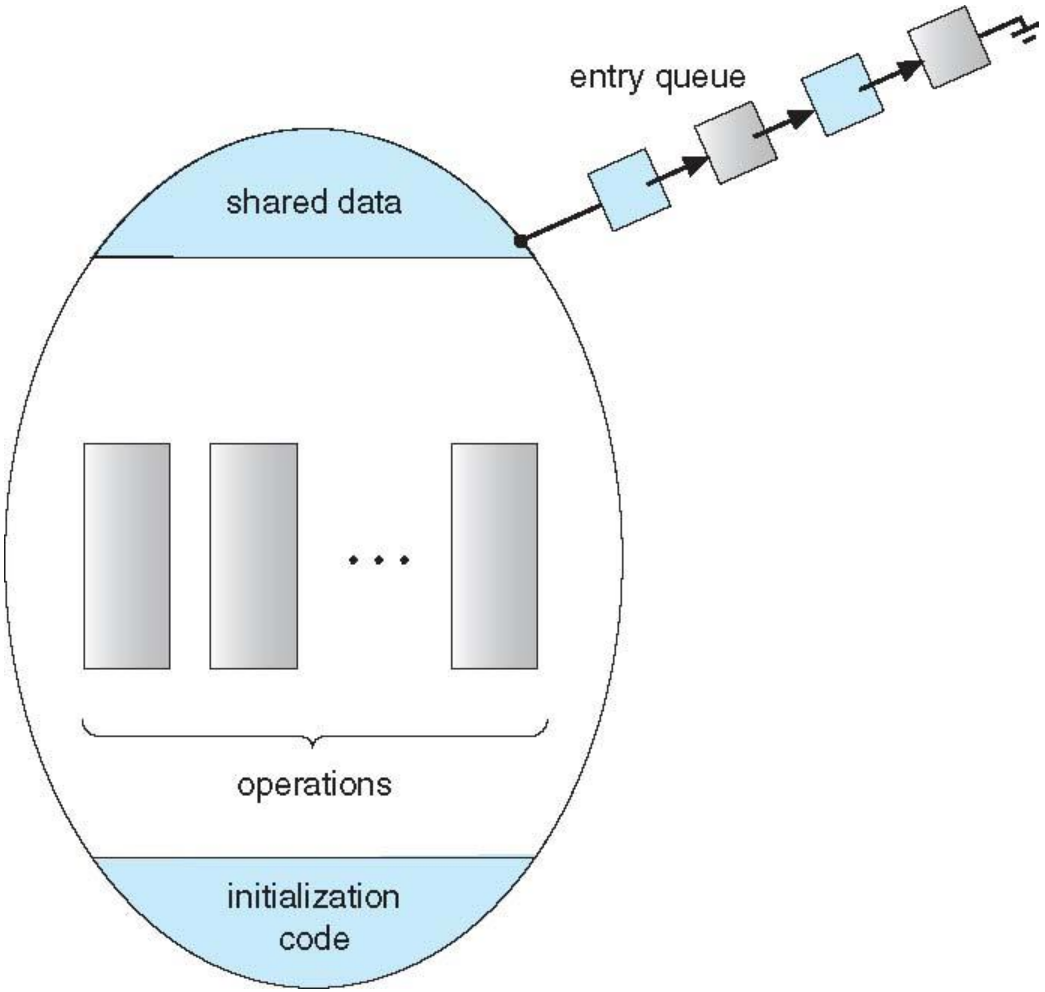
- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex) * unless you want sequential processing
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.

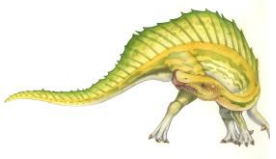
Monitors

- 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
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

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

  procedure Pn (…) {……}

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

Schematic view of a Monitor

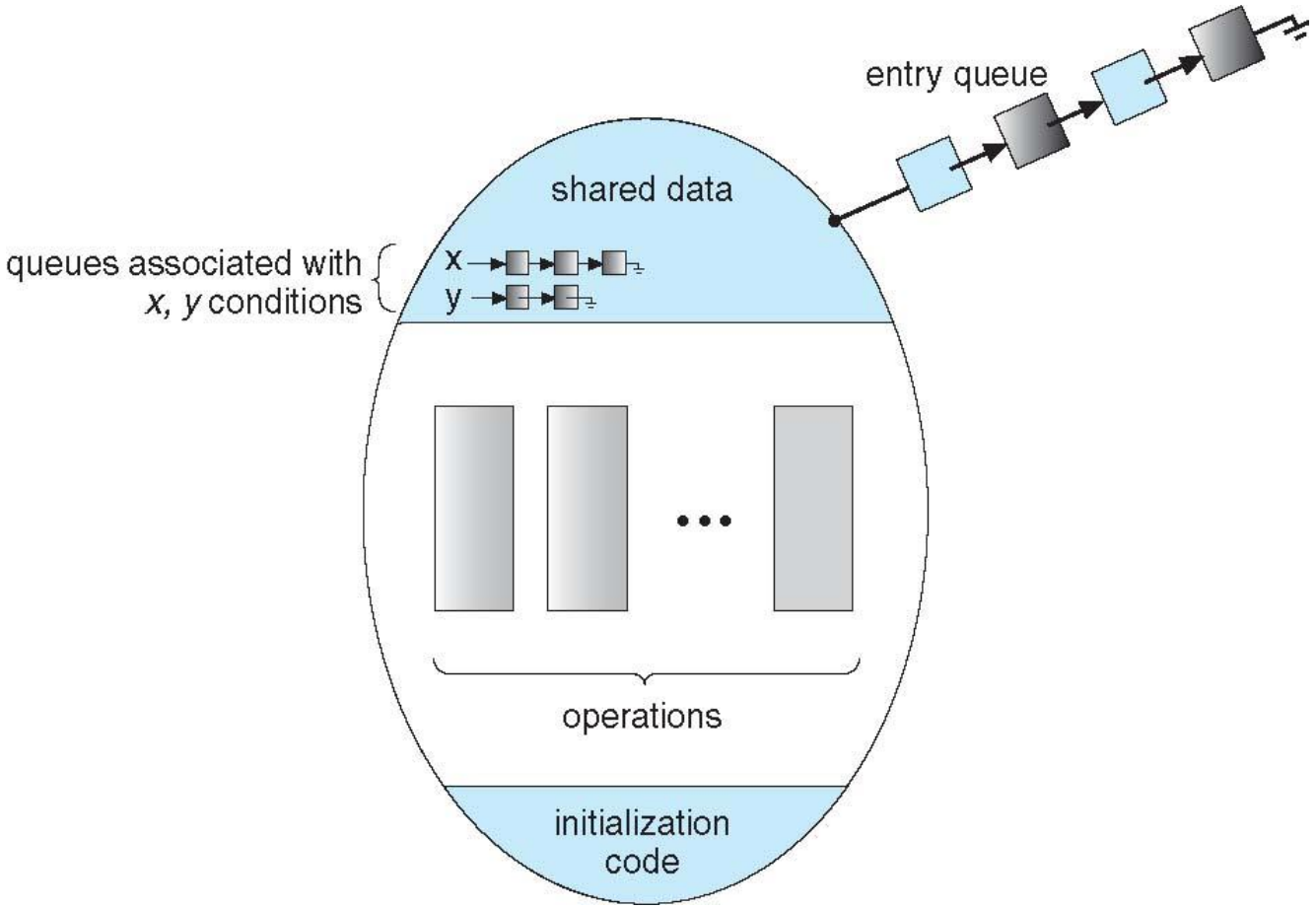




Condition Variables

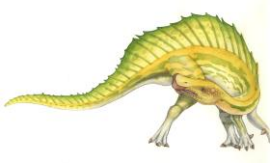
- `condition x, y;`
- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is suspended until **`x.signal()`**
  - **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`**
    - If no **`x.wait()`** on the variable, then it has no effect on the variable

Monitor with Condition Variables



Condition Variables

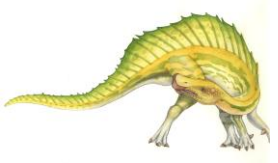
- Condition variables **are not semaphores**. They are different even though they look similar.
  - A condition variable does not count: have no associated integer.
  - A signal on a condition variable x is **lost** (not saved for future use) if there is no process waiting (blocked) on the condition variable x.
  - The **wait**() operation on a condition variable x will **always cause the caller of wait to block**.
  - The **signal**() operation on a condition variable will wake up a sleeping process on the condition variable, **if any.** It has no effect if there is nobody sleeping.



Condition variables: example

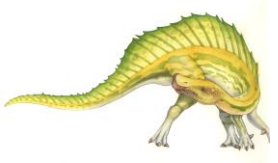
- Assume we have a resource to be accessed by many processes. Assume we have 5 instanced of the resource. 5 processes can use the resource simultaneously.

- We want to implement a monitor that will implement two functions: acquire() and release() that can be called by a process before and after using a resource.



Condition variables: example

```
monitor Allocate
{
        int count = 5;  // we initialize count to 5.
        condition c;

        void acquire () {
                if (count == 0)
                        c.wait();
                count--;
        }

        void release () {
                count++;
                c.signal();
        }
}
```



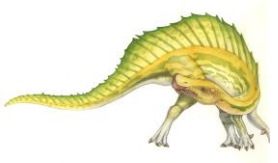
A Monitor to Allocate Single Resource

```
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;
   }
}
```

Dining Philosophers

```
monitor DiningPhilosophers
{    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((i + 4) % 5);
          test((i + 1) % 5);
     }
     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;
}}
```

```
DiningPhilosophers.pickup(i);
...
eat
...
DiningPhilosophers.putdown(i);
```



End of Chapter 5