Process Synchronization

Chapter 6

Adapted from the slides given by Silberschatz, Galvin, and Gagne
In "Operating System Concepts", 10th edition
John Wiley and Sons, Inc 2018.

Outline of this lecture

In this lecture, we will discuss the following:

- Background
- The Critical-Section Problem
- Solutions to the Critical-Section Problem
- Classical Problems of Synchronization

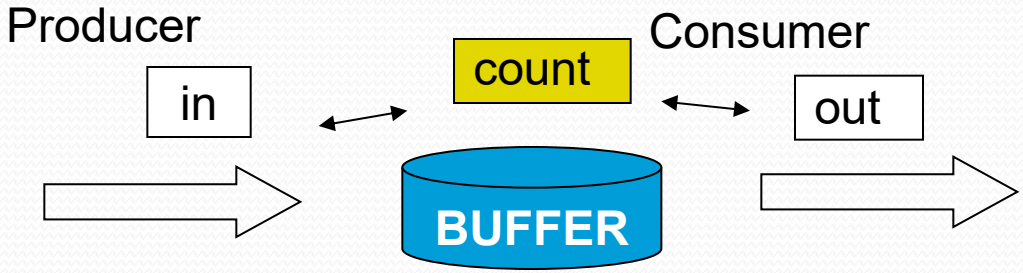
Objectives

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

Background

- A **cooperating process** can affect or be affected by the execution of other processes (e.g. Parent waits for child; parent communicates with child).

- Producer-Consumer problem- To illustrate the concept of cooperating processes *Producer process* produces info that is consumed by a *Consumer process*.



- Concurrent access to shared data may result in data inconsistency.

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

Producer

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

while (count == BUFFER SIZE) ;
       /* do nothing */

buffer[in] = next produced;
in = (in + 1) % BUFFER SIZE;
count++;
}
```

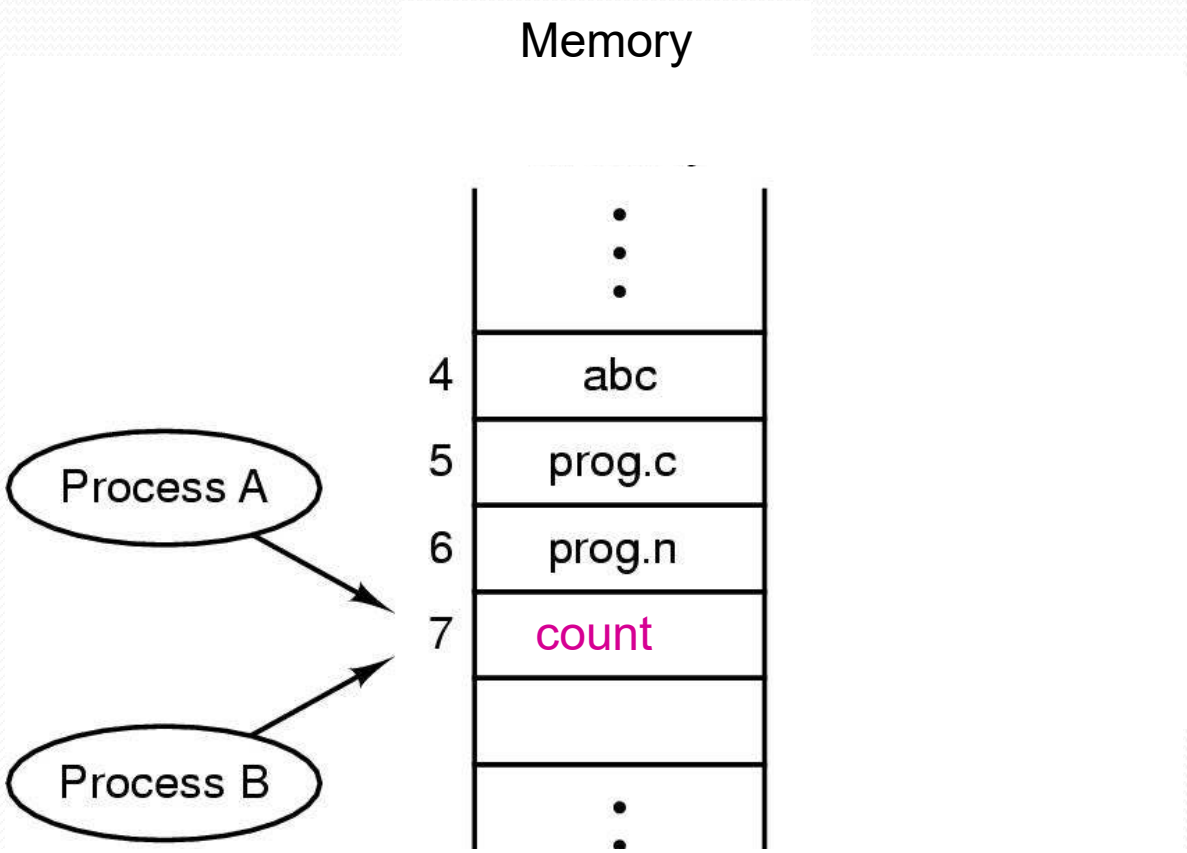
Consumer

```
while (true) {

while (count == 0)
; /* do nothing */

next consumed = buffer[out];
out = (out + 1) % BUFFER SIZE;
count--;
/* consume the item in next
consumed */
}
```

Race condition example



Suppose that count=5

Producer & consumer processes concurrently execute the statement count++ and count--.

Following the execution of both statements, what's the value of the variable count will be? 4,5 or 6!

Race Condition

- `count++` could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

- `count--` could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

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

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

Race condition

- Shared-memory solution to bounded-buffer problem (see lecture 4) has a **race condition** on the class data ***count*.**

- A situation where several processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which access takes place.
- Two or more processes reading and writing some shared data and final result depends on who runs precisely when.

- To prevent race conditions, concurrent processes must be **synchronized**

The Critical Section (CS) Problem

- CS is the part of process code that manipulates shared data or resources.
- Execution of the CS should be *mutually exclusive*- If a process is in its CS, then no other process can be in the same CS.
- Each process must request permission to enter the CS in **entry section**, may follow critical section with **exit section**, then **remainder section**.
- The *Critical section problem* is to design a protocol that the processes can use to cooperate.

Critical Section

- General structure of process p_i is

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

Solution to Critical-Section Problem

Solution to the Critical Section Problem must meet three conditions…

1. **Mutual Exclusion**. If process *Pi* 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 sections, then only those processes that are not executing in their RS can participate in the decision on which will enter its CS next, & this selection 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. This bound prevents **Starvation**

Types of Solutions

- Software solutions
  - algorithms who's correctness does not rely on any other assumptions (OS or hardware)
- Hardware solutions
  - rely on special machine instructions
- Operating System solutions
  - provide special functions and data structures to the programmer

Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **`load`** and **`store`** instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **`int turn;`**
  - **`Boolean flag[2]`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- The **`flag`** array is used to indicate if a process is ready to enter the critical section. **`flag[i] = true`** implies that process $\mathbf{P_i}$ is ready!

Algorithm for Process P_i

```
do {
    flag[i] = true;//process Pi is ready to enter its CS
    turn = j; //process Pj is allowed to execute in its CS
    while (flag[j] && turn == j);
          critical section
    flag[i] = false;
          remainder section
} while (true);
```

- Provable that

1. Mutual exclusion is preserved (the value of turn can be either i or j but cannot be both)
2. Progress requirement is satisfied (A process cannot immediately re-enter the critical section if the other process has set its flag to true)
3. Bounded-waiting requirement is met (a process will not wait longer than one turn for entrance to the critical section)

Drawbacks of Software Solutions

- **Complicated** to program
- Processes that are requesting to enter in their critical section are **busy waiting** (consuming processor time needlessly)
- If CS are long, it would be more efficient to block processes that are waiting (just as if they had requested I/O).

Synchronization Hardware

- Many systems provide hardware support for critical section code
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
  - Design of locks can be sophisticated

- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems (must tell all CPUs)
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions: Atomic = non-interruptable
  - Two types:
    - 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:
    }
```

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);
```

swap Instruction

- Definition:

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

Solution using swap

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

```
do {
        key = TRUE;
        while ( key == TRUE)
              Swap (&lock, &key );

                //    critical section

         lock = FALSE;

                //      remainder section

       } while ( TRUE);
```

Semaphore

- Synchronization tool that does not require busy waiting.
- Semaphore *S* – integer variable
- can only be accessed via two indivisible (atomic) operations
- Two standard operations modify ***S***: **`wait()`** and **`signal()`**
  - Originally called **`P()`** and **`V()`**
- Less complicated

```
  wait (S) {
     while (S <= 0)
        ; // busy wait
     S--;
}
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; can be simpler to implement.
  - Also known as ***mutex locks*** –locks that provide mutual exclusion.

- Can implement a counting semaphore ***S*** as a binary semaphore
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

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

Both processes are running concurrently – statement S2 must be executed only after executing statement S1
Synch semaphore shared between P1 and P2 is initialized to 0;

Semaphore Implementation

- Must guarantee that no two processes can execute **`wait ()`** and **`signal ()`** on the same semaphore at the same time

- Thus, 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
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied

- 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

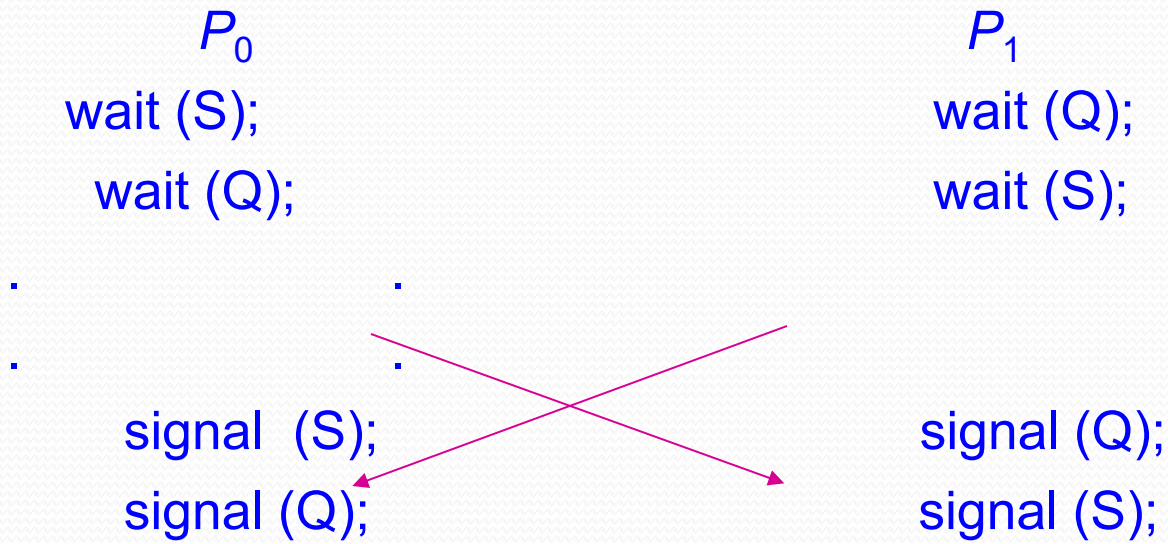
- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

Semaphore Implementation with no Busy waiting (Cont.)

```
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);
    }
}
```

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



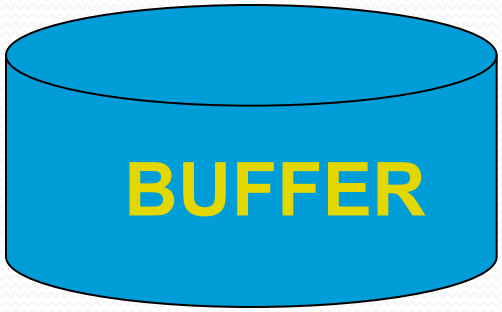
- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.
- When Po executes Wait(Q), it must wait until P1 executes Signal (Q) and when P1 executes Wait(S)), it must wait until Po executes Signal(S) .

Classical Problems of Synchronization

- 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); // synchronize the use of the bounded buffer
    wait(mutex); //protect the CS for a shared value
        ...
      /* 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 treated – all involve priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1 // common to both reader and writer
  - Semaphore **`mutex`** initialized to 1 // to ensure mutual exclusion
  - Integer **`read_count`** initialized to 0 //keeps track of how many processes are currently reading the object.

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
        wait (rw_mutex) ;

          //    writing is performed

        signal (rw_mutex) ;
    } while (true)
```

Note: if a writer is in the CS and *n* readers are waiting, then 1 reader is queued on **rw_mutex** and *n-1* readers are queued on **mutex**.

Readers-Writers Problem (Cont.)

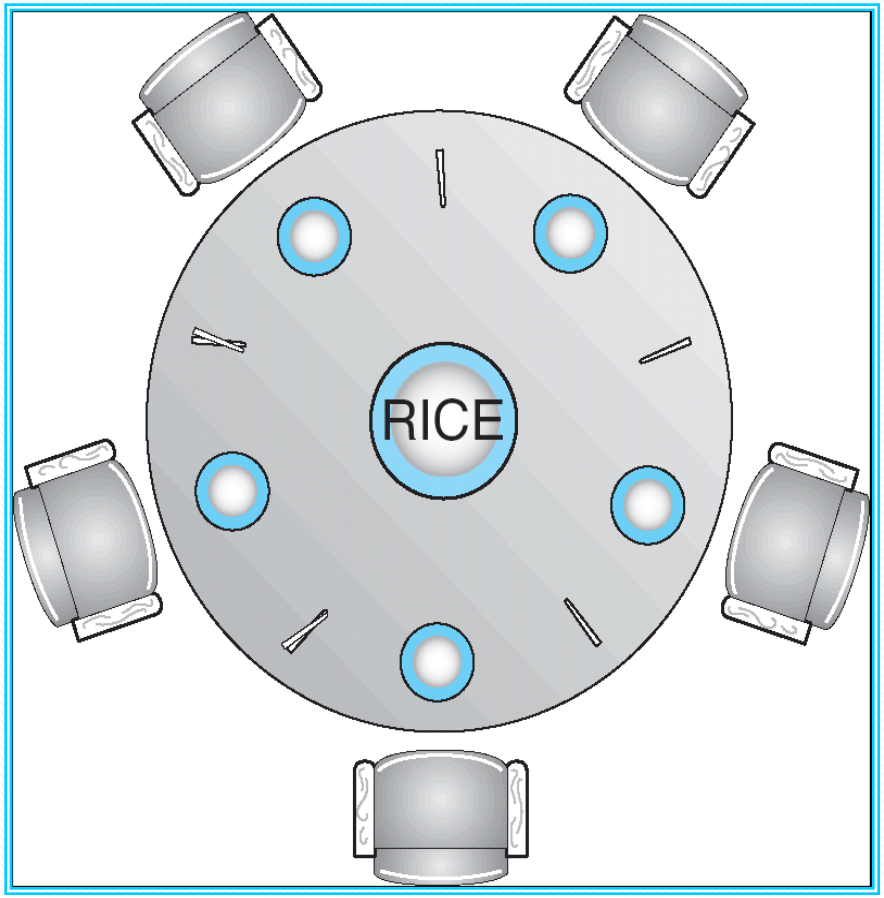
- The structure of a reader process

```
do {
    wait(mutex); //Mutex protects READCOUNT from all the READERS
     read count++;
    if (read count == 1) //1st reader captures rw mutex for all readers

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

- 5 philosophers who only eat and think
- each need to use 2 chopsticks for eating
- we have only 5 chopsticks
- A classical synchronization problem
- Illustrates the difficulty of allocating resources among process without deadlock and starvation



Dining-Philosophers Problem (Cont.)

- Each philosopher is a process
- One semaphore per chopstick:
  - chopstick: array[0..4] of semaphores initialized to 1

```
do {
        wait ( chopstick[i] );
          wait ( chopStick[ (i + 1) % 5] );

                // eat

          signal ( chopstick[i] );
          signal (chopstick[ (i + 1) % 5] );

              // think

} while (TRUE);
```

- Deadlock if each philosopher start by picking his left chopstick

Problems with Semaphores

- semaphores provide a powerful tool for enforcing mutual exclusion and coordinate processes
- But wait(S) and signal(S) are scattered among several processes. Hence, difficult to understand their effects
- Usage must be correct in all the processes
- 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

Operating System Solutions: Monitors

- Are high-level language constructs that provide equivalent functionality to that of semaphores but are easier to control.
- But not powerful enough to model some synchronization schemes

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

    procedure Pn (…) {……}

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

Syntax of a monitor

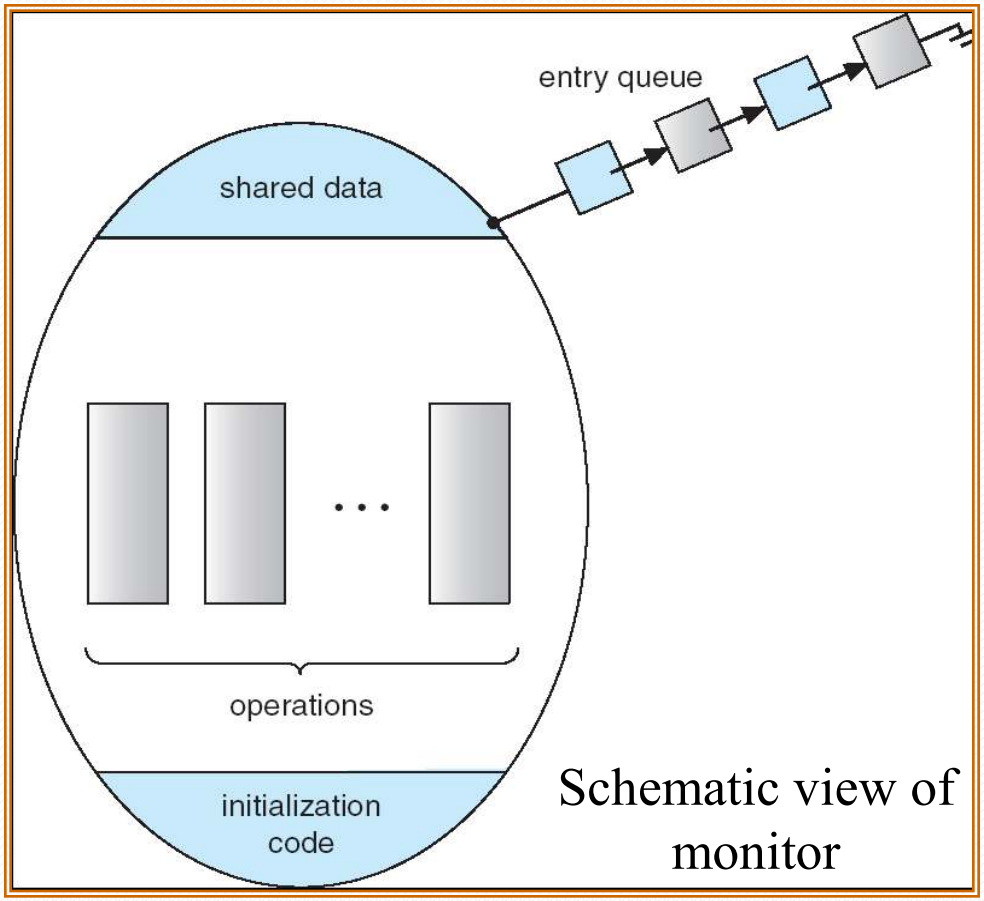
Procedure defined within the monitor can access only those variables declared locally within the monitor

Operating System Solutions: Monitors

- **Is a software module containing:**
  - one or more procedures
  - an initialization sequence
  - local data variables
- **Characteristics:**
  - local variables accessible only by monitor's procedures
  - a process enters the monitor by invoking one of its procedures
  - only **one process** can be in **the monitor** at any given time

Operating System Solutions: Monitors

- The monitor construct ensures only one process is active within the monitor ---→ The programmer does not need to program this synchronization constraint explicitly as shown below:



Schematic view of monitor

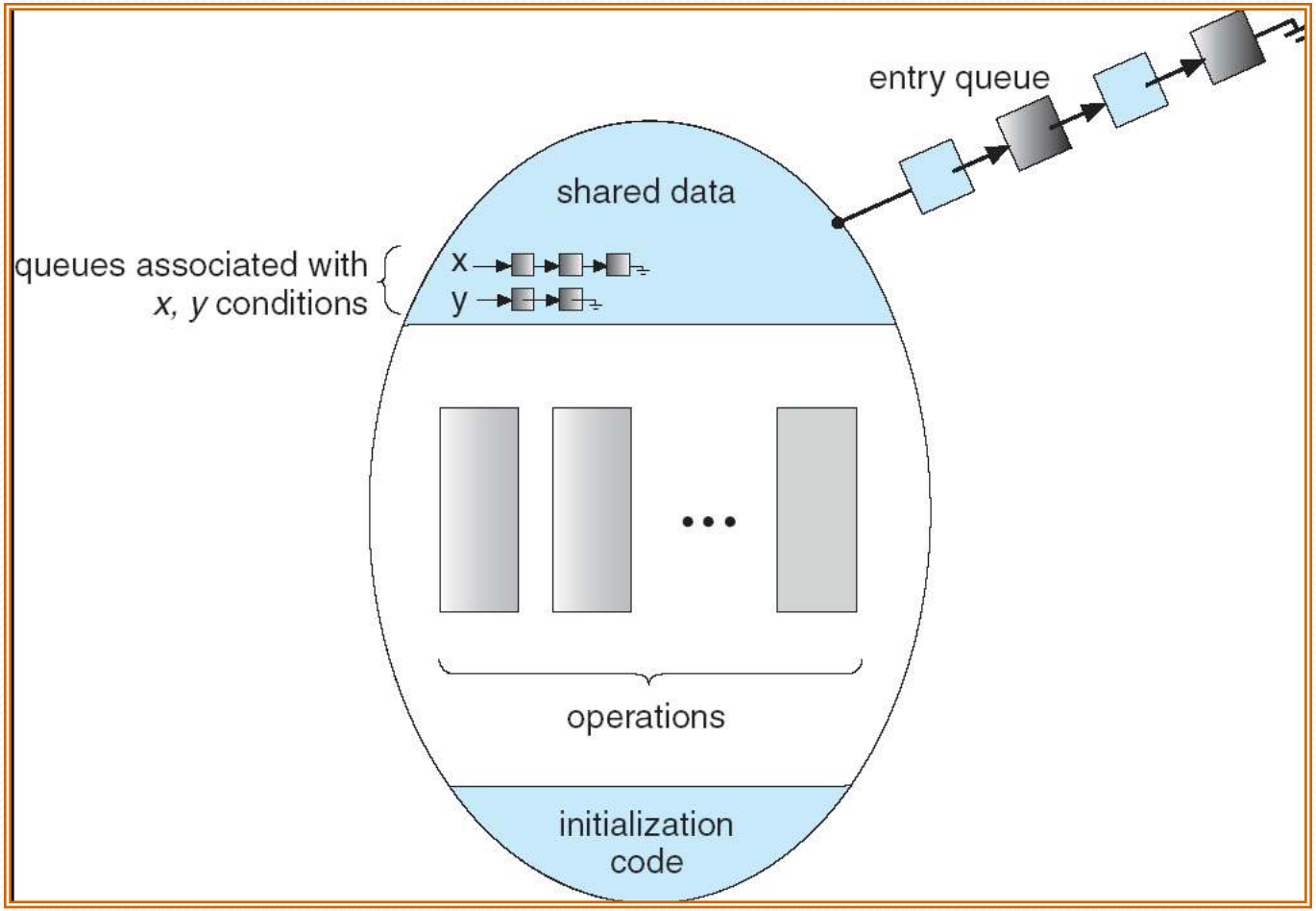
Condition Variables

- The monitor construct is not sufficiently powerful for modeling some synchronization schemes-→ need to define additional mechanisms.
- These mechanisms are provided by the condition construct.
- A programmer who needs to write a tailor-made synchronization scheme can define one or more variable of type *condition*:

- condition x, y;

- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one suspended 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 (), with Q in x.wait () state, what should happen next?
  - If Q is resumed, then P (signaling process) must wait

- Options include
  - **Signal and wait** – P waits until Q leaves monitor or waits for another condition
  - **Signal and continue** – Q waits until P leaves the monitor or waits for another condition

  - Both have pros and cons – language implementer can decide
  - Implemented in other languages including Mesa, C#, Java

Solution to Dining Philosophers

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

   void pickup (int i) {
         state[i] = HUNGRY;
         test(i);
         if (state[i] != EATING) self [i].wait;
   }

     void putdown (int i) {
         state[i] = THINKING;
                // test left and right neighbors
          test((i + 4) % 5);
          test((i + 1) % 5);
      }
```

```
dp.pickup(i);

…

Eat

…

dp.putdown(i);
```

- No deadlock, but starvation is possible

Solution to Dining Philosophers (cont)

```
    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;
    }
}
```

Process Synchronization Conclusion

- A variety of methods exist, and each method has different variations.
- The most commonly used are semaphores and monitors.
- Monitors are the easiest to use, closest to programmer's needs.
- The simpler mechanisms can be used to implement more sophisticated ones.
- No mechanism prevents in principle deadlocks, starvation, or busy waiting but more sophisticated mechanisms make it easier to avoid them.