Chapter : Process Synchronization

Background

- **Co-Operating Process: that can affect or be affected by other processes executing in system**
- Concurrent access to shared data may result in data inconsistency
- **Process Synchronization: Ensures coordination among processes and maintains Data Consistency**

- **Process P1** **Process P2**

1. X=5

2. X=5+2

1. read(x);

2. x=x+5;

3. Printf(x);

Producer Consumer Problem

There can be two situations:

1. **Producer Produces Items at Fastest Rate Than Consumer Consumes**

2. **Producer Produces Items at Lowest Rate Than Consumer Consumes**

Producer Consumer Problem

Producer Produces Items at Fastest Rate Than Consumer Consumes:

If Producer produces items at fastest rate than Consumer consumes Then Some items will be lost

Eg. Computer → Producer

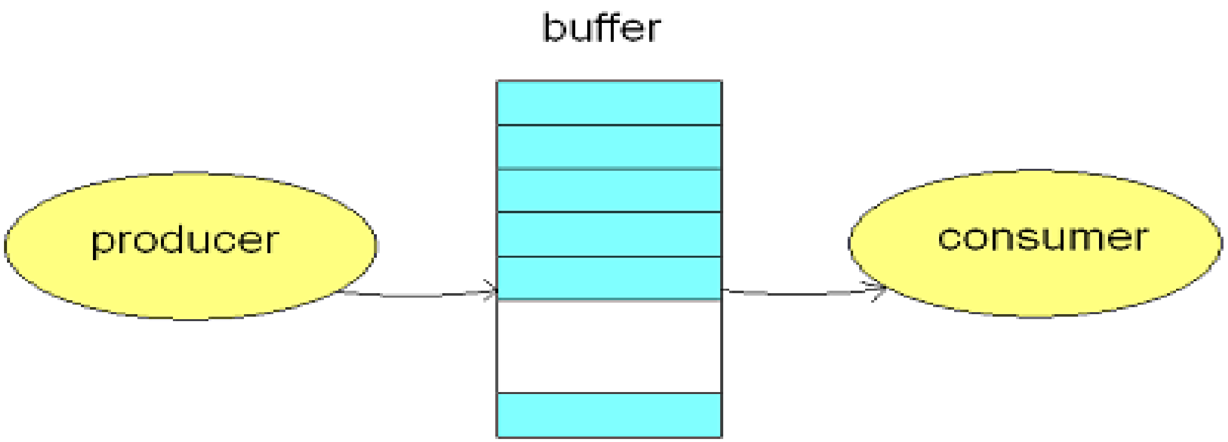
Printer → Consumer

Producer Consumer Problem

Solution:

To avoid mismatch of items Produced or Consumed → Take Buffer

Idea is: Instead of sending items from Producer to Consumer directly→ Store items into buffer



Producer Consumer Problem

Buffer Can be:

1. Unbounderd Buffer:
    1. No buffer size limit
    2. Any no. of items can be stored
    3. Producer can produce on any rate, there will always be space in buffer

2. Bounded Buffer:
    1. Limited buffer size

Producer Consumer Problem

Bounderd Buffer:

If rate of Production > rate of Consumption:

Some items will be unconsumed in buffer

If rate of Production < rate of Consumption:

At some time buffer will be empty

Producer

```
while (true) {

    /*  produce an item and put in nextProduced  */

    while (count == BUFFER_SIZE)
        ; // do nothing
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
}
```

Consumer

```
while (true)  {
        while (count == 0)            // buffer empty
              {
              ; // do nothing
               }
              nextConsumed =  buffer[out];
               out = (out + 1) % BUFFER_SIZE;
                 count- -;

              /*  consume the item in nextConsumed
 }
```

Race Condition

- When multiple processes access and manipulate the same data at the same time, they may enter into a race condition.
- **Race Condition: When output of the process is dependent on the sequence of other processes.**
- Race Condition occurs when processes share same data

- **Process P1** **Process P2**

1. reads i=10
2. i=i+1 =11

1. P2 reads i=11 from memory
2. i=i+1 = 12

3. Stores i=11 in memory
3. Stores 12 in memory

Critical Section Problem

- Section of code or set of operations, in which process may be changing shared variables, updating common data.
- **A process is in the critical section if it executes code that manipulate shared data and resources.**
- Each process should seek permission to enter its critical section→ **Entry Section**
- **Exit Section**
- **Remainder section:** Contains remaining code

Structure of a process

Repeat

{

Locks are set here

// Entry Section

Critical Section (a section of code where processes work with shared data)

Critical Section

// Exit Section

Locks are released here

Remainder Section

} until false.

Solution to: Critical Section

1. Mutual Exclusion:

It states that if one process is executing in its critical section, then no other process can execute in its critical section.

2. Bounded Wait:

It states that if a process makes a request to enter its critical section and before that request is granted, there is a limit on number of times other processes are allowed to enter that critical section.

3. Progress:

It states that process cannot stop other process from entering their critical sections, if it is not executing in its CS.

Peterson's Solution

- Two process solution (Software-based)
- 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!

- Only **2 processes**,
- General structure of process P_i (other process P_j)

do {

$\boxed{\text{entry section}}$

critical section

$\boxed{\text{exit section}}$

reminder section

} **while (1)**;

Algorithm for Process P_i

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

            CRITICAL SECTION

        flag[i] = FALSE;

            REMAINDER SECTION
```

```
do {

    acquire lock

        critical section

    release lock

        remainder section

} while (TRUE);
```

Critical Section Problem solution

- Now prove that this solution is correct. We need to show that:
  - Mutual exclusion is preserved.
  - The progress requirement is satisfied.
  - The bounded-waiting requirement is met.

Synchronization Hardware

Hardware Solution to C.S.

- Many systems provide hardware support for critical section code

- Uni-processors – could disable interrupts
  - Currently running code would execute without preemption

- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

Hardware Solution to C.S.

1. Interrupt Disabling

1. Process leaves control of CPU when it is interrupted.
2. Solution is:
    1. To have each process disable all interrupts just after entering to the critical section.
    2. Re-enable interrupts after leaving critical section

Hardware Solution to C.S.

- **Interrupt Disabling**

Repeat

Disable interrupts

C.S

Enable interrupts

Remainder section

Synchronization Hardware

- Having the support of some simple hardware instructions, the CS problem can be solved very easily and efficiently.
- The CS problem occurs because the modification of a shared variable of a process may be interrupted.
- Two common hardware instructions that execute **atomically**
  - **`Test-and-Set`**
  - **`Swap`**

Synchronization Hardware

- Test and modify the content of a word atomically.

```
boolean TestAndSet(boolean &target) {
    boolean rv = target;
    target = true;

    return rv;
}
```

Mutual Exclusion with Test-and-Set

- Shared data:

  boolean lock = false;

- Process P_i

```
do {
    while (TestAndSet(lock)) ;
        critical section
    lock = false;
        remainder section
}
```

Synchronization Hardware

- Atomically swap two variables.

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

Mutual Exclusion with Swap

- Shared data (initialized to **false**):

boolean lock; /*global variable

- Process P_i

```
do {
    key = true;
    while (key == true)
        Swap(lock, key);
        critical section
    lock = false;
remainder section
}
```

Semaphore

- Synchronization tool that maintains concurrency using variables
- Semaphore S is a integer variable which can take positive values including 0. It is accessed from 2 operations only.

Operations On Semaphore:

- wait() and signal()

1. Wait Operation is also known as P() which means to test
2. Signal Operation is also known as V() which means to increment

- Entry to C.S. is controlled by wait()
- Exit from C.S. is signaled by signal()

Semaphore

- Can only be accessed via two indivisible (atomic) operations and S=1
- For critical Section problem semaphore value is always 1
- P(S) and V(S):

```
wait (S) {
        while (S <= 0)
        do skip ; // no-op}
          S- -;
C.S
signal (S) {
      S++;      }
```

Semaphore as General Synchronization Tool

Types of Semaphores:

- Counting semaphore – when integer value can be any non-negative value
- Binary semaphore – integer value can range only between 0 and 1
  - Also known as mutex locks

Semaphore and Busy Waiting

Disadvantage of Semaphore: Busy Waiting

- When a process is in C.S. and any other process that wants to enter C.S. loops continuously in entry section.
- Wastes CPU cycles
- Semaphore that implements busy waiting is known as: **Spin Lock**

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

Semaphore Implementation with no Busy waiting

Instead of waiting, a process blocks itself.

- Two operations:
  - block – place the process invoking the operation on the waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.

Semaphore Implementation with no Busy waiting

- Implementation of wait:

S=1

```
wait (S){
      value--;
      if (value < 0)  {
  add process P to waiting queue
   block();
 }
C.S
}
```

Semaphore Implementation with no Busy waiting

- Implementation of signal:

```
Signal (S){
        value++;            //"value" has -ve value i.e -1 here
         if (value <= 0) {
    remove a process P from the waiting queue
     wakeup(P);  }
}
```

Deadlocks and Starvation

- The using of semaphores may cause **deadlocks**

(Initially, A=B=1)

P_0

```
wait(A);
wait(B);
   S0
signal(A);
signal(B);
```

P_1

```
wait(B);
wait(A);
   S1
signal(B);
signal(A);
```

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

  For example: waiting queues are implemented in LIFO order.

Drawbacks of Semaphores

- Semaphores provide a convenient and effective mechanism for process synchronization.
- However, **incorrect use** may result in timing errors.

```
signal(mutex);
   ...
  critical section
   ...
wait(mutex);
```

incorrect order
(not mutual exclusive)

```
wait(mutex);
   ...
  critical section
   ...
[        ]
```

forgotten

```
[        ]
   ...
  critical section
   ...
signal(mutex);
```

```
wait(mutex);
   ...
  critical section
   ...
wait(mutex);
```

typing error
(deadlock)

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

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.
- Shared Data
  - Data set
  - For Readers: Semaphore mutex initialized to 1.
  - For Writers: Semaphore wrt initialized to 1.
  - Integer readcount initialized to 0.

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
while (true) {
        wait (wrt) ;

            //    writing is performed

        signal (wrt) ;
}
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
while (true) {
 wait (mutex) ;
 readcount ++ ;
 if (readcount == 1)
wait (wrt) ;    //( don't allow writers)
 signal (mutex);  //(allow other readers to come)

                // reading is performed

 wait (mutex) ;     // (one by one readers leave the C.S.)
 readcount  - - ;
 if (readcount  == 0)
{ signal (wrt) ; } // last reader will do this
 signal (mutex) ; // mutex=1 (for readers to exit and writer to come)
}
```

writer

```
wait(wrt);
 ...
    writing is performed
 ...
signal(wrt);
```

reader

first-in

```
wait(mutex);
  readcount++;
  if (readcount == 1) wait(wrt);
signal(mutex);
    ...
    reading is performed
    ...
wait(mutex);
  readcount--;
  if (readcount == 0)  signal(wrt);
signal(mutex);
```

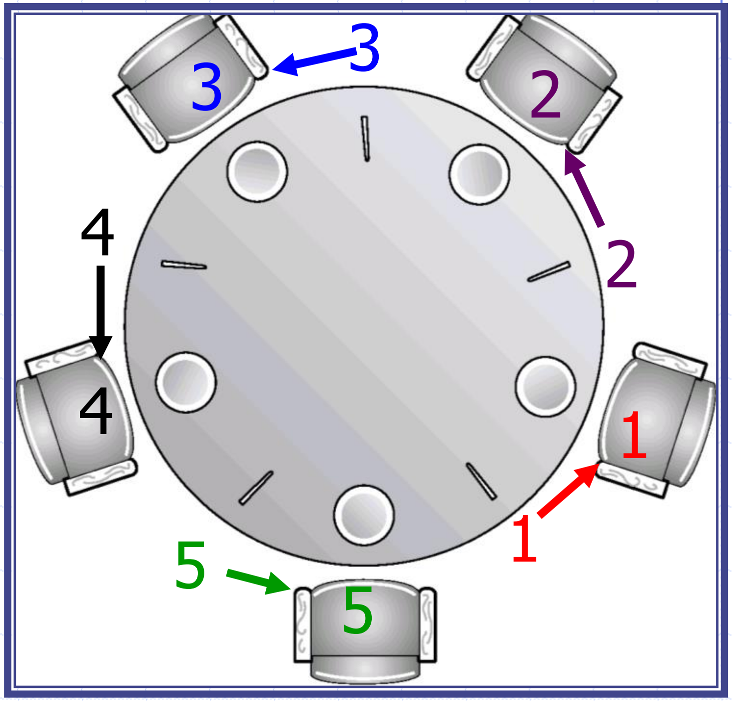
update readcount

Dining-Philosophers Problem

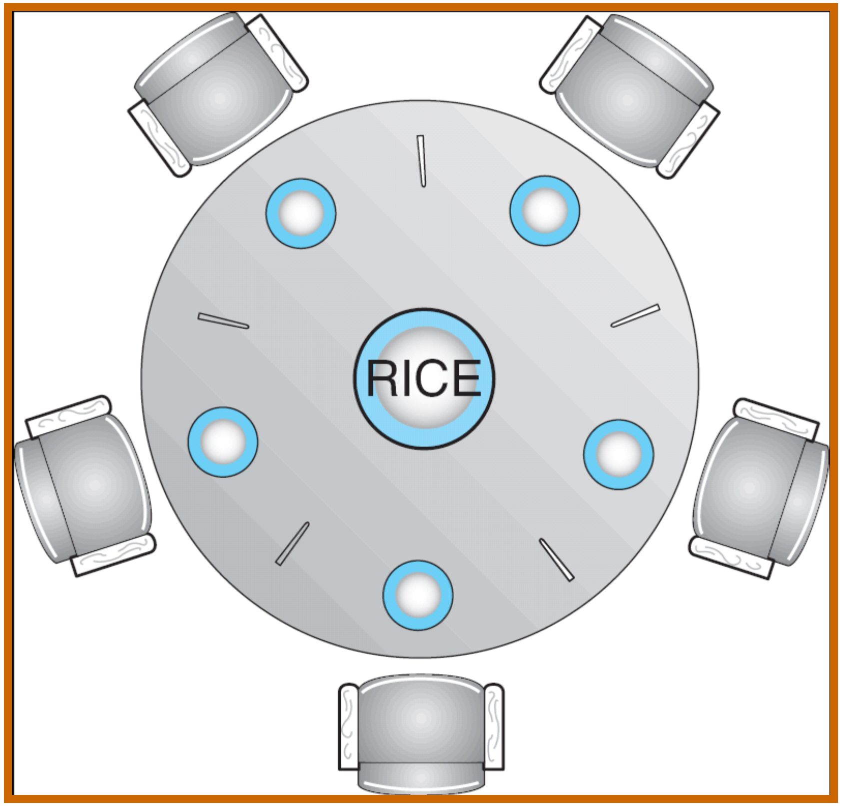
- Five philosophers, either thinking or eating
- to eat, two chopsticks are required
- taking one chopstick at a time
- Shared data

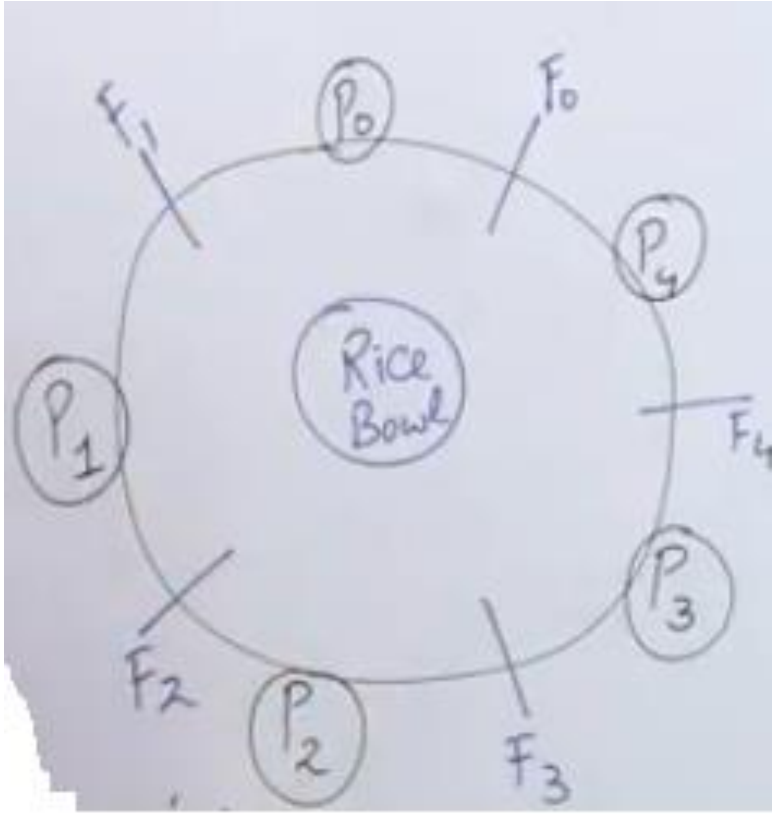
semaphore chopstick[5];

Initially all values are 1



Dining-Philosophers Problem





Dining-Philosophers Problem

philosopher *i*

```
do {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
        …
            eat
        …
    signal(chopstick[i]);
    signal (chopstick[(i+1) % 5]);
        …
     think
        …
} while(1);
```

get chopsticks
- left
- right

free chopsticks
- left
- right

deadlock !

Dining-Philosophers Problem

- Possible solutions to the deadlock problem
  - Allow at most **four** philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up her chopsticks only if **both chopsticks are available** (note that she must pick them up in a critical section).
  - Use an asymmetric solution; that is,
    - odd philosopher: **left first, and then right**
    - an even philosopher: **right first, and then left**
- Besides deadlock, any satisfactory solution to the DPP problem must avoid the problem of **starvation**.

Monitors

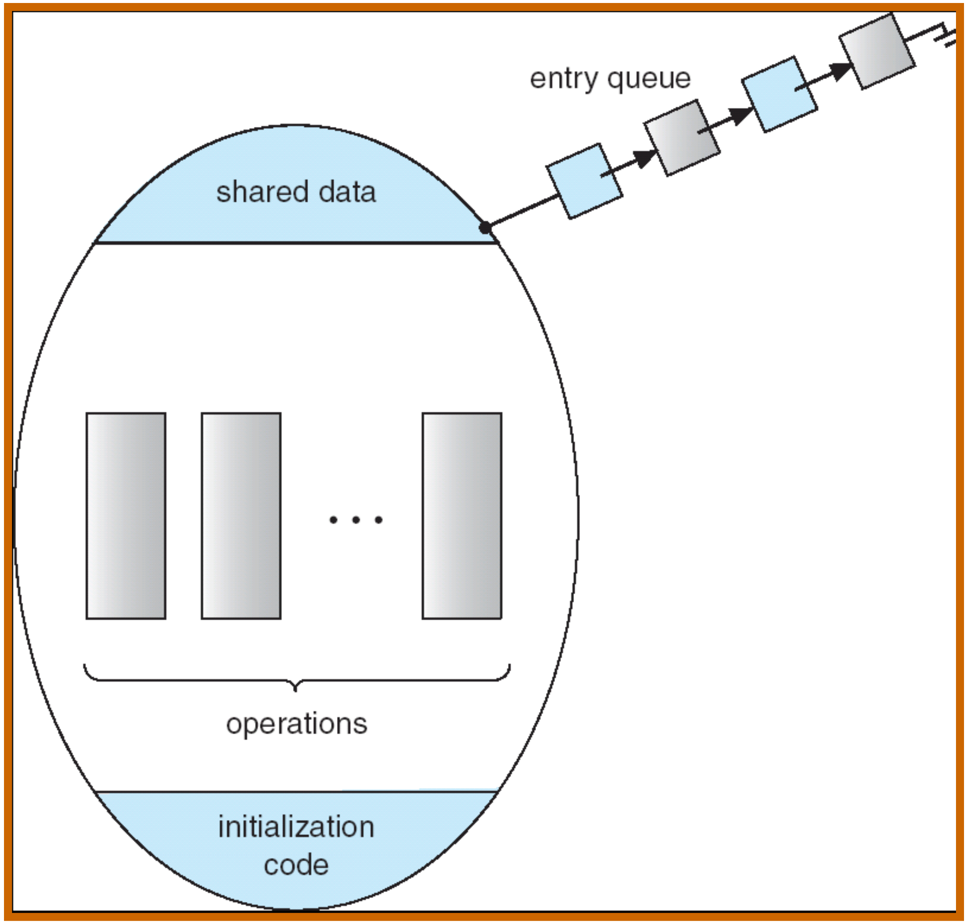
- A way to encapsulate the Critical Section by making class around the critical section and allowing only one process to be active in that class at one time.
- The monitor type is a high-level synchronization construct.
- Only one process may be active within the monitor at a time
  - Name of Monitor
  - Initialization Code Section
  - Procedure to request the Critical Data
  - Procedure to release the Critical Data

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

   procedure P1 ( . . . ) {
      . . .
   }

   procedure P2 ( . . . ) {
      . . .
   }

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

   initialization code ( . . . ) {
      . . .
   }
}
```

Figure 6.16 Syntax of a monitor.

Schematic view of a Monitor

Only one process at a time can be in monitor



Monitors

- The **monitor** construct is not sufficiently powerful for modeling some synchronization schemes.
- So, we need to define additional synchronization mechanisms.
- These mechanisms are provided by the **condition construct.**
- A programmer can define one or more variables of type **condition**:

  condition x, y;

- The only operations that can be invoked on a **condition variable** are **wait ()** and **signal()**.

Condition Variables

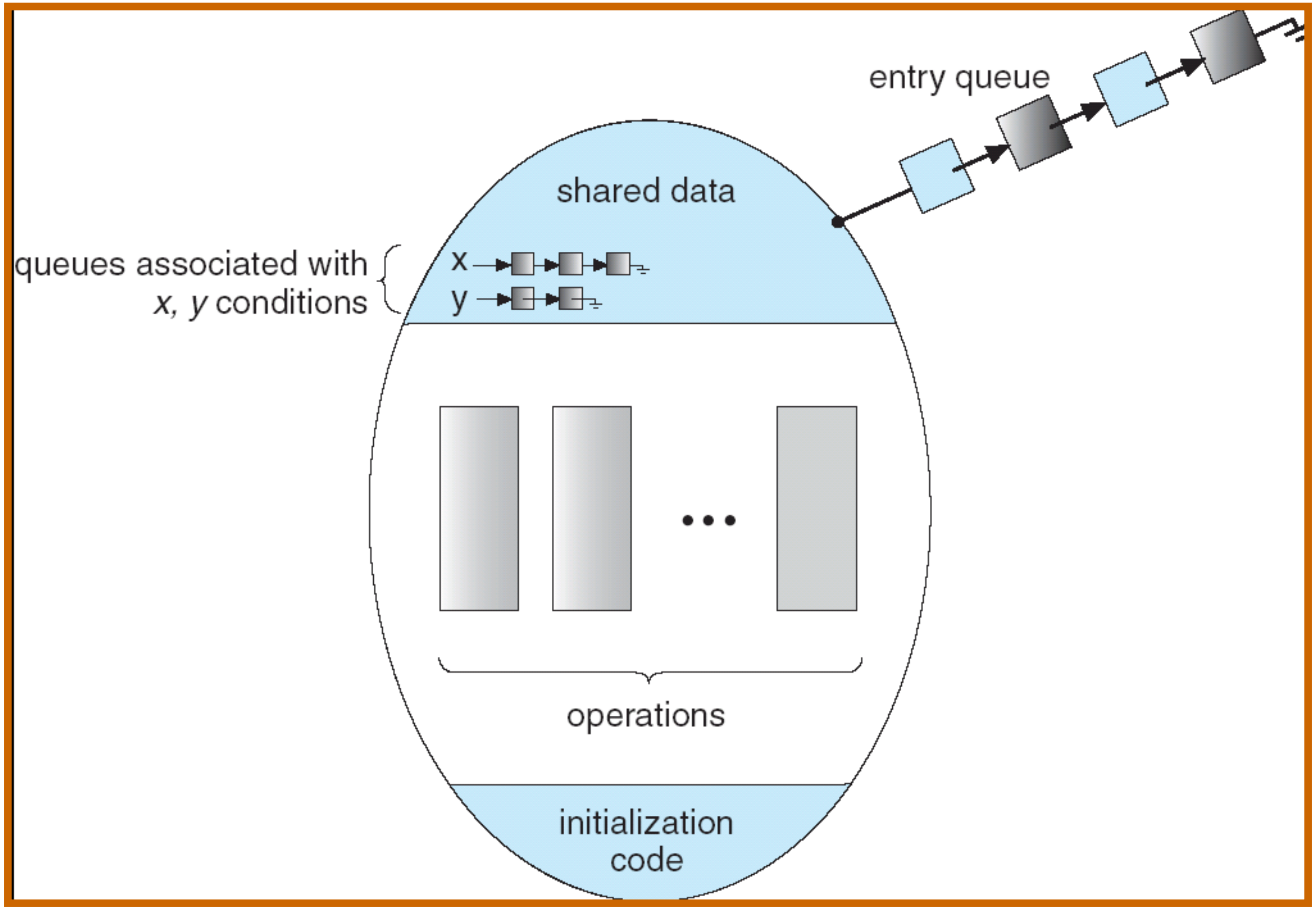
- Two operations on a condition variable:

  - **x.wait ()** – a process that invokes the operation is suspended until another process invokes **x.signal()**

  - **x.signal ()** – resumes one of processes (if any) that invoked **x.wait ()**

There could be different conditions for which a process could be waiting

Monitor with Condition Variables



Solution to Dining Philosophers (cont)

- The distribution of the chopsticks is controlled by the **monitor** DiningPhilosophers

- Each philosopher ‘ i ’ invokes the operations pickup() and putdown() in the following sequence:

    dp.pickup (i)

        EAT

    dp.putdown (i)

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);
      }
```

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 = 1; i <= 5; i++)
      state[i] = THINKING;
 }
}
```