Chapter 4: Process Synchronization

Process Synchronization

- ✓ Cooperating Process
- ✓ The Critical-Section Problem
- ✓ Synchronization Hardware
- ✓ Semaphores
- ✓ Classical Problems of Synchronization
- ✓ Monitors
- ✓ Synchronization examples

Cooperating Process

- ✓ 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 or be allowed to share data only through files or messages
- ✓ Concurrent access to shared data may result in data inconsistency.
- ✓ Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Bounded Buffer – Shared Memory Solution

Shared data

```
#define    BUFFERSIZE   10
Typedef  struct { … } item;
item   buffer[ BUFFERSIZE ];        int  in = 0, out = 0 ;
```

Producer process

```
item   nextproduced;
while (TRUE)
{ /* produce an item in nextproduced */
 while((in+1) % BUFFERSIZE == out);
     buffer [in] := nextproduced ;
     in = ( in + 1 ) % BUFFERSIZE;
}
```

Consumer process

```
item   nextconsumed;
while (TRUE)
{  while ( in == out ) ;
  nextconsumed = buffer [out];
  out=(out+1)%BUFFERSIZE;
/* Consume an item in next
   consumed */
}
```

Solution is correct, but can only fill up BUFFERSIZE–1 buffer

Bounded Buffer

- ✓ Shared-memory solution to bounded-buffer problem allows at most $n - 1$ items in buffer at the same time. A solution, where all n buffers are used is not simple.
- ✓ One possibility is to modify the producer-consumer code by adding a variable *counter*, initialized to 0
- ✓ Counter is incremented each time a new item is added to the buffer
- ✓ Counter is decremented every time we remove one item from the buffer

Bounded Buffer – modified solution

Shared data

```
#define     BUFFERSIZE    10
Typedef  struct { … } item;
item   buffer[ BUFFERSIZE ];        int  in = 0, out = 0, counter = 0 ;
```

Producer process

```
item   nextproduced;
while (TRUE)
{ /* produce an item in nextproduced */
 while ( counter == BUFFERSIZE );
     buffer [in] := nextproduced ;
     in = ( in + 1 ) % BUFFERSIZE;
     counter++;
}
```

Consumer process

```
item   nextconsumed;
while (TRUE)
{  while ( counter == 0 ) ;
  nextconsumed = buffer [out];
  out=(out+1)%BUFFERSIZE;
  counter--;
/* Consume the item in
   nextconsumed
}
```

Bounded Buffer

The statements

counter++;   and   counter--;

must be performed *atomically*.

- ✓ Atomic operation means an operation that completes in its entirety without interruption.

Bounded Buffer

The statement “counter++” may be implemented in machine language as:

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

The statement “counter--” may be implemented as:

```
register2 = counter
register2 = register2 – 1
counter = register2
```

Bounded Buffer

- ✓ **If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.**

- ✓ **Interleaving depends upon how the producer and consumer processes are scheduled.**

Bounded Buffer

Assume counter is initially 5. One interleaving of statements is:

T0: producer execute register1 = counter (*register1* = 5)

T1: producer execute register1 = register1 + 1 (*register1* = *6*)

T2: consumer execute register2 = counter (*register2 = 5)*

T3: consumer execute register2 = register2 – 1 (*register2* = *4*)

T4: producer execute counter = register1 (*counter* = *6*)

T5: consumer execute counter = register2 (*counter* = *4*)

The value of counter may be either 4 or 6, where the correct result should be 5.

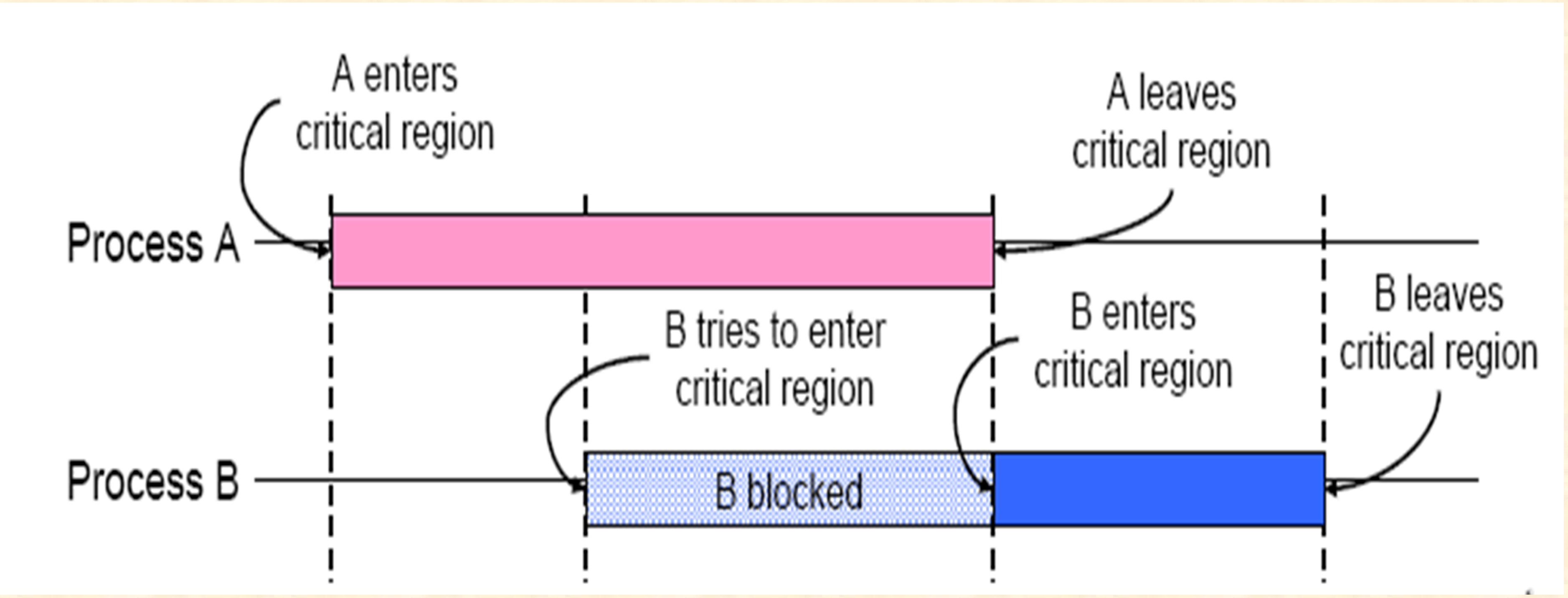
Race Condition

- ✓ Race condition: **The situation where several processes access and manipulate same shared data concurrently. The final value of the shared data depends on the particular order in which the access to shared data takes place.**

- ✓ **How to prevent race conditions?**
  - **Ensure that Only one process at a time can be manipulating the shared variable (ex. counter)**
  - **Concurrent processes must be synchronized.**

Critical-Section Problem

- *n* processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Goal – when one process is executing in its critical section, no other process is allowed to execute in its critical section.



Critical-Section Problem contd….

- ✓ General structure of a process

```
do {
        entry section
        critical section
        exit section
        reminder section
   }  while ( 1 ) ;
```

- ✓ Processes may share some common variables to synchronize their actions.

Solution to Critical-Section Problem

A solution to the Critical Section problem must satisfy THREE requirements

1. Mutual Exclusion
2. Progress
3. Bounded Waiting

1. **Mutual Exclusion :** Mutually exclusive execution of Critical Section.

   If process P_i is executing in its critical section, then no other processes can be executing in their critical sections

Solution to Critical-Section Problem

2. **Progress:** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision on which will enter its critical section next, and 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. It Prevents starvation.

Peterson's Solution

- Two process solution - allows just two processes to share a single-use resource without conflict
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int turn;

  /* indicates whose turn it is to enter the critical section. */

  - boolean ready[2]

  /* Used to indicate whether a process is ready to enter the critical section. ready[i] = 1 implies that process P_i is ready */

Peterson's Solution

Process P_i

```
int turn;
boolean ready[2];

        do {

            ready [i]:= 1;
            turn = j;
            while (ready [j] && turn == j) ;

            critical section

            ready [i] = 0;

                remainder section

            }   while (1);
```

Meets all the THREE requirements;

Peterson's Solution……example

```
int turn=0;
boolean ready[ 2 ]={ 0 };
```

Process P0

```
do
{   ready [0]:= 1;
      turn = 1;
      while (ready [1] &&
                        turn == 1) ;
    critical section
    ready [0] = 0;
    remainder section
} while (1);
```

Process P1

```
do
{  ready [1]:= 1;
      turn = 0;
      while (ready [0] &&
                        turn == 0) ;
    critical section
    ready [1] = 0;
    remainder section
 } while (1);
```

Peterson’s Solution

Meets all the THREE requirements;

- ✓ Mutual exclusion : P0 and P1 can never be in the critical section at the same time. If P0 is in its critical section, then either ready[1] is 0 or turn is 0. In both cases, P1 cannot be in its critical section.
- ✓ Progress requirement : If process P0 does not want to enter its critical section, P1 can enter it without waiting. There is no strict alternating between P0 and P1.
- ✓ Bounded waiting : A process will not wait longer than one turn for entry into the critical section. After giving priority to the other process, this process will run to completion and set its ready to 0, thereby allowing the other process to enter the critical section.

Lamport's Bakery Algorithm

- ✓ Critical section solution for n processes

Analogy

- ✓ Lamport envisioned a bakery with a numbering machine at its entrance so each customer is given a unique number.
- ✓ Numbers increase by one as customers enter the store.
- ✓ A global counter displays the number of the customer that is currently being served.
- ✓ All other customers must wait in a queue until the baker finishes serving the current customer and the next number is displayed.
- ✓ When done shopping, the customer loses the number and can then do whatever he/she want, except for shopping without getting a new number.

Lamport's Bakery Algorithm

- ✓ In the computer world, the 'customers' will be threads, identified by the letter *i*, obtained from a global variable
- ✓ Before entering its critical section, process receives a number.
- ✓ Holder of the smallest number enters the critical section.
- ✓ If processes Pi and Pj receive the same number, if i < j, then Pi is served first; else Pj is served first.
- ✓ The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...

Lamport's Bakery Algorithm

- ✓ Notation <≡ lexicographical order (ticket #, process id #)
  - ❖ (a,b) < (c,d)   if (a < c) or ((a == c) and (b < d))
  - ❖ max $(a_0,\dots, a_{n-1})$ is a number k, such that

    $k \geq a_i$ for i =0, …, n – 1
- ✓ Shared data

```
boolean   choosing[n];
int     number[n];
```

- ✓ Data structures are initialized to FALSE and 0 respectively

Lamport's Bakery Algorithm

```
boolean choosing[n] = { FALSE };        int number[n] = { 0 };
do { choosing[i] = TRUE;  /* choosing  a number */
       number[i] = max ( number[0], number[1], …,
                                                   number [n – 1] ) + 1;
      choosing[ i ] = FALSE;
      for (j = 0; j < n; j++)     // for all other processes
       {  while (choosing[j]) ;  // wait until process j receives its number
          while ( (number[j] !=0)  &&  (number[j], j) < (number[i], i) ) ;
    /* wait if the process has a number and comes ahead of  process i */
          }
          critical section
          number[i] = 0;
          remainder section
      } while (1);
```

Synchronization Hardware

- ✓ Critical-section problem could be solved easily in a uniprocessor environment
  - Disallow interrupts to occur while a shared variable is being modified

```
Process Pi:
while (1)
{    disable interrupts
         critical section
     enable interrupts
     remainder section
 }
```

Synchronization Hardware

- This solution is not feasible for multiprocessor environment
  - Interrupts are “per-CPU”
  - Disabling interrupts on a multiprocessor system is time consuming as the message is passed to all the processors
  - Message passing delays entry into critical section and system efficiency decreases
  - If the system clock is kept updated by interrupts then system clock will be showing incorrect time

Synchronization Hardware

- ✓ Therefore Modern Computer systems provide special hardware instructions that allow
  - To test and set(modify) the content of a word, or
  - To swap the contents of 2 words, atomically

- ✓ Test and Set instruction is defined as:

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

Swap function

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

Mutual Exclusion implementation with TestandSet

Shared data:

```
boolean lock = FALSE;
```

Process P_i

```
do {
      while ( TestandSet ( &lock ) ) ;
         critical section
      lock = FALSE;
      remainder section
   } while (TRUE);
```

Mutual Exclusion implementation with Swap

Shared data (initialized to FALSE):

```
                boolean lock;

    boolean key; // local variable

    Process P_i

                do {      key = TRUE;
                          while (key == TRUE )
                                      Swap( &lock, &key);
                          critical section
                          lock = FALSE;
                          remainder section
                } while (TRUE);
```

These algorithms does NOT fulfill bounded waiting requirement

Bounded-waiting, mutual exclusion with TestandSet

Shared data structures : (initialized to FALSE)

```
    boolean  waiting [n] ;
    boolean  lock;
do {  waiting [i] = TRUE;
        key = TRUE;
        while( waiting [ i ]  &&  key )
                key = TestandSet ( &lock );
```

/* P_i can enter its CS only if waiting[i] == FALSE or key == FALSE.

key can be FALSE only if the Test-and-Set( ) is executed

The first process to execute this will find key == FALSE, all others must wait.

waiting[i] becomes FALSE only when other process leaves its CS.

Only one waiting[i] is set to FALSE at a time. M.E. is met */

Bounded-waiting, mutual exclusion with Test-and-Set

```
        waiting [i] = FALSE;
        Critical section
        j = ( i + 1) % n;
          while ( ( j != i ) && ( ! waiting [ j ] ) )
                    j = ( j + 1 )% n;   /* Give chance to next process
which is waiting to enter CS */
       if( j == i )  lock = FALSE;  // No process is waiting
       else  waiting [ j ] = FALSE; // Enable the waiting process
       Remainder section
}  while ( TRUE );
```

A process exiting CS scans the array waiting[ ] in cyclic order. It either sets lock to FALSE or sets waiting[j] to FALSE enabling a waiting process to enter its CS. Thus satisfying Progress & Bounded-waiting requirement.

Properties of Hardware Approaches

- ✓ Advantages:
  - Any number of processes on any number of processors sharing memory.
  - Simple and easy to verify.
  - Support multiple critical sections; each critical section can be defined by its own variable.
- ✓ Disadvantages:
  - Busy waiting. While a process is waiting for access to a critical section, it continues to consume processor time.
  - Starvation is possible. The selection of a waiting process is arbitrary. Thus, some process could indefinitely be denied access.
  - Hardware based solutions are complicated for the application programmers to use

Semaphores

- ✓ Synchronization tool (provided by the OS)
- ✓ A Semaphore S is an integer variable that, apart from initialization, can be accessed only through two *atomic and mutually exclusive* operations:

1. wait (S)   (or sleep or down or P)
2. signal (S)   (or wakeup or up or V)

definition

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

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

Usage of Semaphores

Types of semaphores

1. Binary Semaphore (or mutex locks)
2. Counting Semaphore (or general semaphores)

- ✓ Binary Semaphores can assume only the value 0 or the value 1
- ✓ Counting semaphores can assume value over an unrestricted domain

Binary Semaphore solution to Critical Section of *n* Processes

Shared data:

```
semaphore    mutex;    //initially mutex = 1
```

Process *Pi:*

```
do {    wait (mutex);
        critical section

        signal (mutex);
        remainder section
   } while (1);
```

Usage of Semaphores

- Synchronization problems
  - P1 with statement S1 and P2 with statement S2 are 2 concurrently running processes
  - Suppose that we require S2 be executed only after S1
    - We can implement as follows

Shared data:

```
      semaphore synch      // initially 0
              S1;
              signal (synch);     // in process P1


              wait (synch);
               S2;                  // in process P2
```

- Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal (synch), which is after S1

Semaphore Implementation

- ✓ previous wait & signal definition has busy waiting
  - wastes CPU cycles
    - In a multiprogramming environment another process might have used CPU productively
  - Previous definition of Semaphore is called spinlocks - procss spins while waiting for lock
  - Advantage of spinlock is no context switches
    - When locks are expected to be held for short durations process must wait on a lock rather than wasting more time in context switch

- ✓ How to overcome Busy waiting?

Semaphore Implementation

- ✓ Define a semaphore as a record

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

When a process must wait on a semaphore it is added to the List

Assume two simple operations:

- Block( )   suspends the process that invokes it
- wakeup(P) resumes the execution of a blocked process P
- A signal( ) operation removes one process from the list of waiting processes and awakens that process.

Semaphore Implementation

- ✓ Semaphore operations now defined as

```
wait( semaphore  *S )
{       S.value--;
        if ( S.value < 0 )
                  {    add this process to S.List;
/* if the semaphore value is negative, its magnitude represents
 the number of processes waiting on that semaphore */
                        block( );     //suspend the process
                   }
}
```

Implementation

```
signal( semaphore  *S )
{       S.value++;
        if ( S.value <= 0 )
         {   remove a process P from S.List;
             wakeup ( P );    //resume the process
         }
  }
```

Deadlock and Starvation in Semaphores

- ✓ 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

```
        Process P0                  Process P1
          wait(S);      then          wait(Q);
         wait(Q);                      wait(S);  //P0, P1 deadlocked
              ⋮                        ⋮
        signal(S);                  signal(Q);
        signal(Q)                   signal(S);
```

- ✓ Starvation – A process may never be removed from the semaphore queue in which it is suspended if it is a LIFO queue.

Classical Problems of Synchronization

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

Bounded-Buffer Problem

- a pool of *n* buffers, each capable of holding one item
- producer:
  - grab an empty buffer
  - place an item into the buffer
  - waits if no empty buffer is available
- consumer:
  - grab a buffer and retracts the item
  - place the buffer back to the free pool
  - waits if all buffers are empty

Bounded-Buffer Problem

Shared data

```
semaphore   full, empty, mutex;
```

// mutex provides mutual exclusion for accesses to the buffer pool

// empty and full count the number of empty and full buffers

Initially:

```
full = 0, empty = n, mutex = 1
```

Bounded-Buffer Problem Producer Process

```
do {
     …
     //produce an item in nextp
     …
    wait ( empty); //wait if number of empty buffers is 0
    wait ( mutex );
     …
      //add nextp to buffer
      …
      signal ( mutex );
      signal ( full );
  } while (1);
```

Bounded-Buffer Problem Consumer Process

```
do {
        wait(full); //wait if number of full buffers is 0
        wait(mutex);
                …
        // remove an item from buffer to nextc
                …
        signal(mutex);
        signal(empty);
                …
        // consume the item in nextc
                …
    } while (1);
```

Readers-Writers Problem

- a set of shared data objects
- a group of processes
  - reader processes (read shared objects)
  - writer processes (update shared objects)
  - a writer process has exclusive access to a shared object
- *First RW problem*: no reader will be kept waiting unless a writer is updating a shared object
- *Second RW problem*: once a writer is ready, it performs the updates as soon as the shared object is released.

Readers-Writers Problem

Shared data

```
    semaphore   mutex,  wrt;
    int   readcount ;
```

/* readcount : how many processes are currently reading the object, mutex : ensures mutual exclusion for updatation of readcount, wrt : mutual exclusion semaphore for writers */

Initially

```
    mutex = 1, wrt = 1, readcount = 0
```

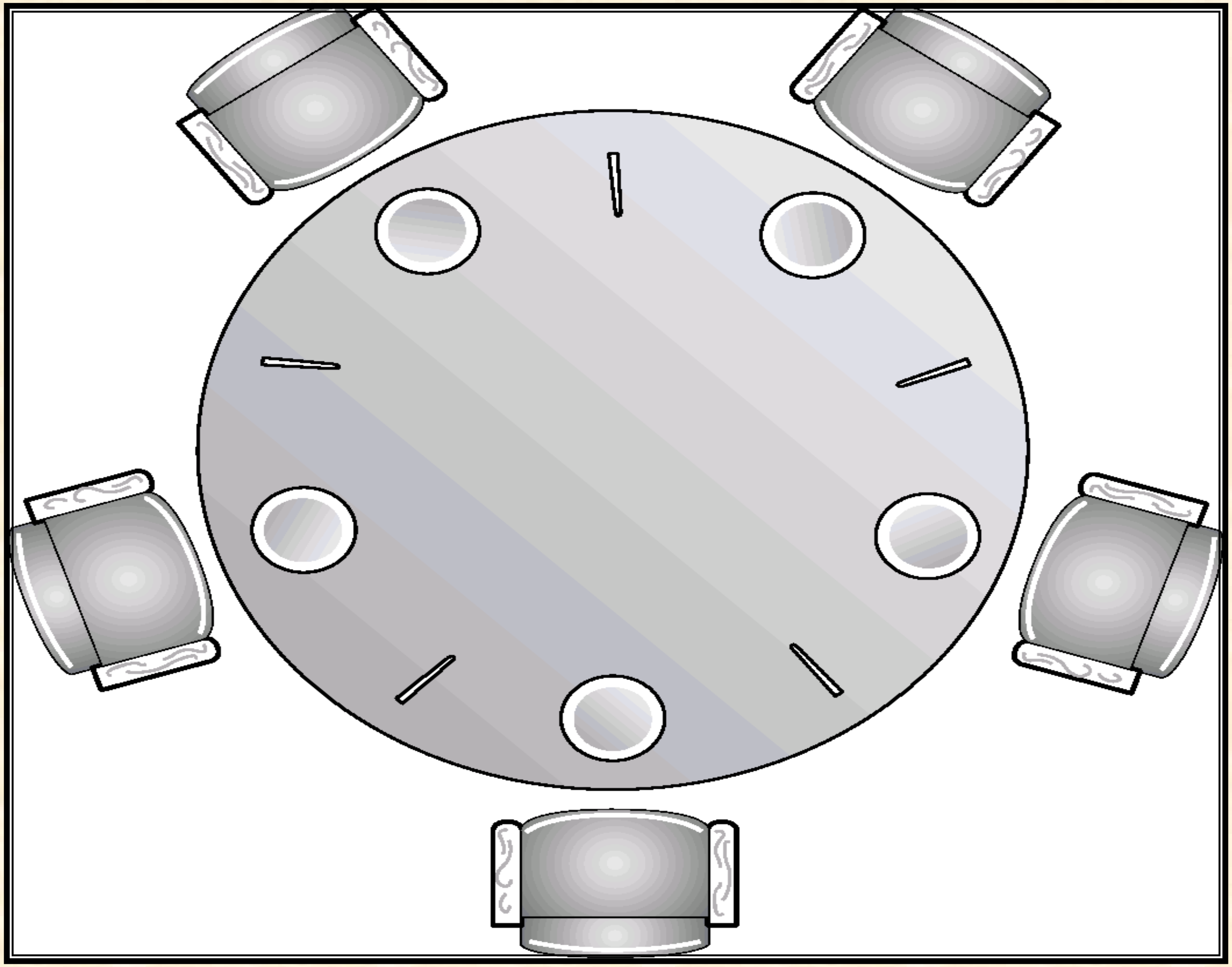
Writer Process

```
        do {   wait(wrt);
                ...
                writing is performed
                ...
                signal(wrt);
           } while(1);
```

Readers-Writers Problem Reader Process

```
do {    wait(mutex);
        readcount++;
        if (readcount == 1)   wait(wrt);
        signal(mutex);
            …
        // reading is performed
            …
        wait(mutex);
        readcount--;
        if (readcount == 0)   signal(wrt);
        signal(mutex):
    }  while(1);
```

Dining-Philosophers Problem



Dining-Philosophers Problem

- 5 philosophers sitting on 5 chairs share a circular table
- Table is laid with 5 single chopsticks
- A philosopher is either thinking or eating
  - thinking: no interaction with the rest 4 persons
  - eating: need 2 chopsticks at hand
  - a philosopher picks up 1 chopstick at a time
  - After eating: put down both chopsticks
- deadlock problem
  - one chopstick as one semaphore
- starvation problem

Dining-Philosophers Problem

Shared data. Initially all values are 1

```
semaphore chopstick[5];
```

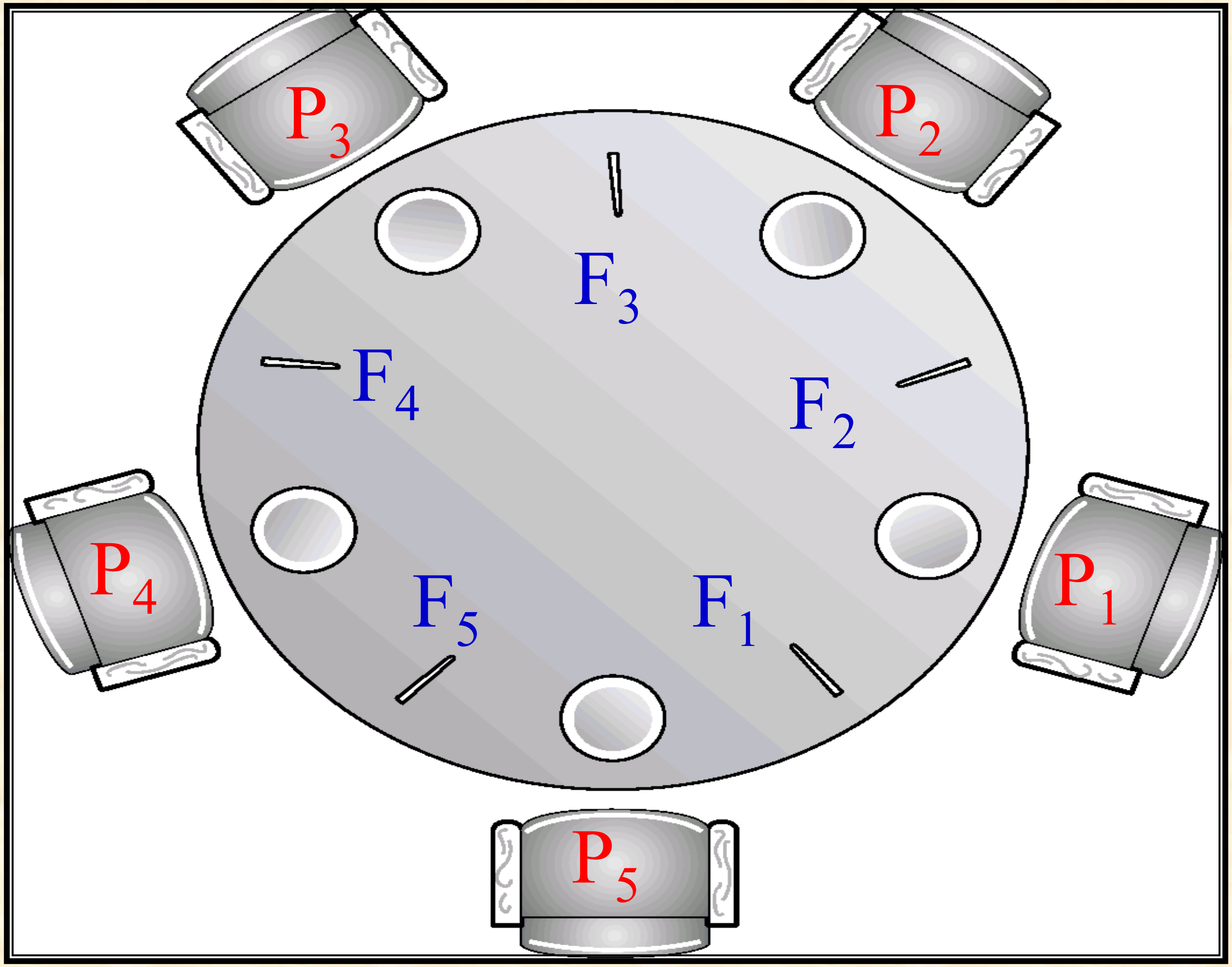
Philosopher *i*:

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

Dining philosophers problem : Solution

- ✓ Philosophers has to pick up the forks in increasing order, which mathematically eliminates the possibility of a deadlock.
- ✓ Label the philosophers P_1, P_2, P_3, P_4, and P_5
- ✓ Label the forks F_1, F_2, F_3, F_4, and F_5.
- ✓ Each philosopher must pick up forks in a prescribed order and cannot pick up a fork another philosopher already has.
- ✓ Upon acquiring two forks, a philosopher may eat.
- ✓ Philosophers P_1 through P_4 follow the rule that P_x must pick up fork F_x first and then may pick up fork F_{x+1}.
- ✓ For example, P_1 must pick up F_1 first and F_2 second.
- ✓ Philosopher P_5 must, conversely, pick up fork F_1 before picking up fork F_5, to respect the deadlock-preventing fork ordering rule.

Dining philosophers problem : Solution



Monitors

- ✓ Monitor is a High-level language synchronization construct that allows safe sharing of an abstract data type among concurrent processes
- ✓ Monitor has a set of programmer-defined operators

A monitor consists of:

- ✓ a set of procedures that allow interaction with the shared resource
- ✓ a mutual exclusion lock
- ✓ the variables associated with the resource
- ✓ a monitor invariant that defines the assumptions needed to avoid race conditions

Monitors

A monitor procedure takes the lock before doing anything else, and holds it until it either finishes or waits for a condition.

If every procedure guarantees that the invariant is true before it releases the lock, then no task can ever find the resource in a state that might lead to a race condition

Syntax of a monitor is :

```
monitor   monitor-name
{     //shared variable declarations
      procedure P1 (…) {
        . . .
            }
          procedure P2 (…) {
                . . .
                }
                  .
        :
```

Monitors

```
    .
    .
    .
    procedure Pn (…) {
                                   . . .
          }

      initialization code ( … ) {
                                         …
             }
}
```

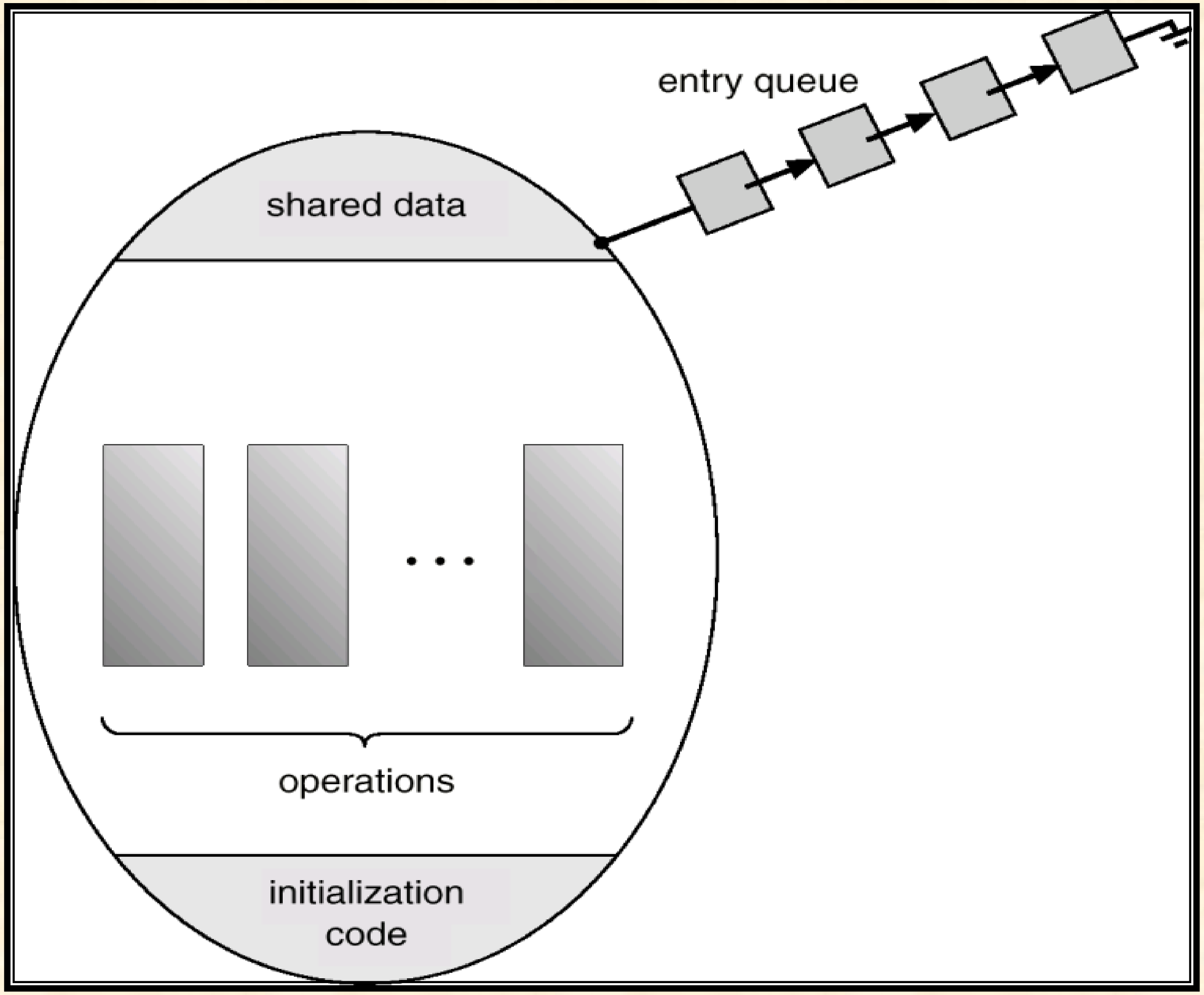
Monitors

- ✓ **To allow a process to wait within the monitor, a** condition **variable must be declared, as**

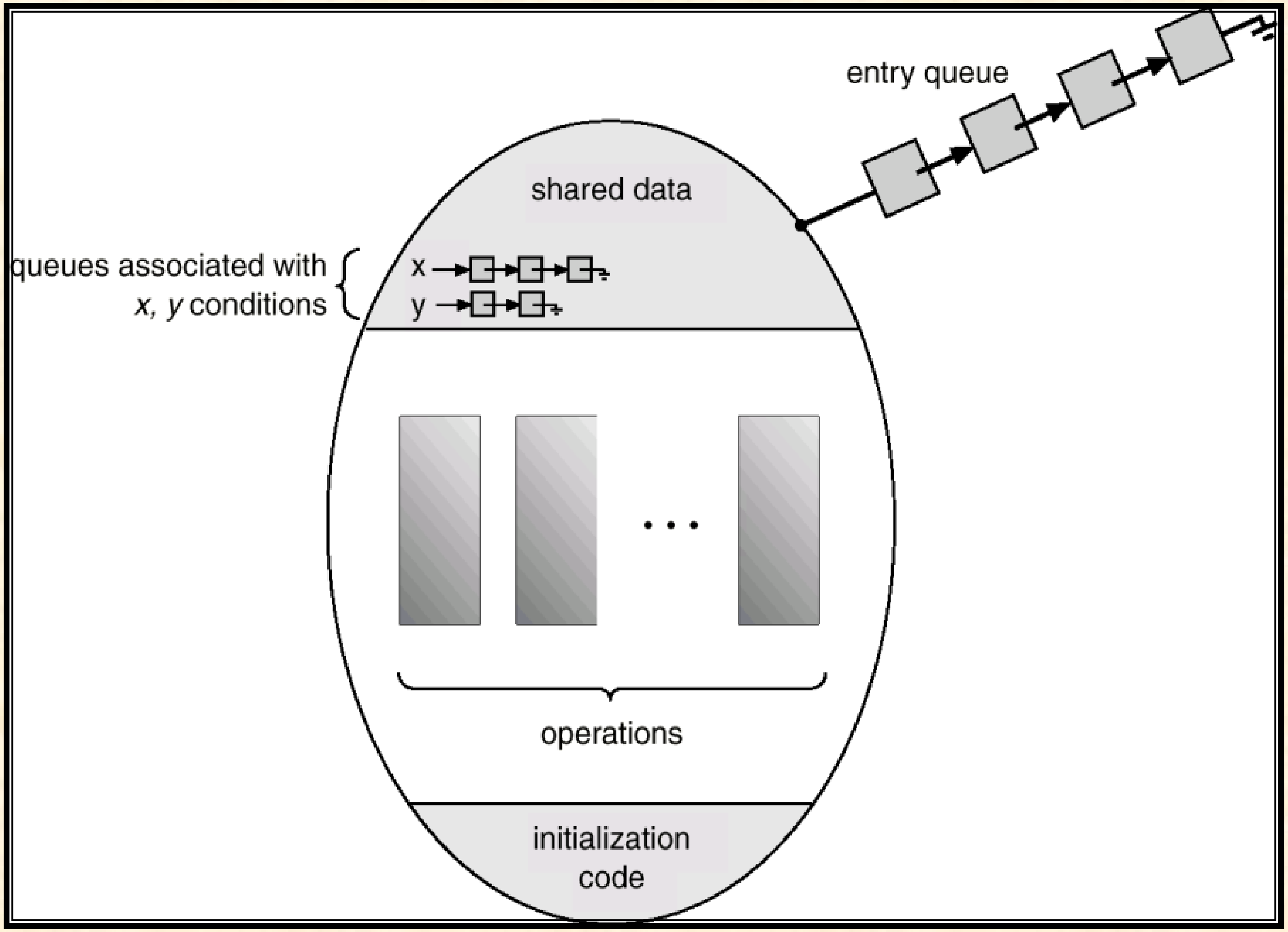
  condition x, y;

- ✓ **Condition variable can only be used with the operations** wait **and** signal**.**
  - **The operation** x.wait( ); **means that the process invoking this operation is suspended until another process invokes** x.signal( );
  - **The** x.signal **operation resumes exactly one suspended process. If no process is suspended, then the** signal **operation has no effect.**

Schematic View of a Monitor



Monitor With Condition Variables



Dining Philosophers Example

```
monitor dp
{     enum { thinking, hungry, eating }  state [ 5 ];
      condition self [ 5 ]; /*philosopher can delay himself when
   hungry but unable to obtain the chopsticks*/
/* Every philosopher, before starting to eat, must invoke
   pickup(). It may result in suspension of the process*/
  void pickup(int i)
   {  state[i] = hungry;
      test ( i ); /* checks the state of  neighbors, if both
   neighbors are not eating, then sets the state to eating */
      if (state[i] != eating)
           self[i].wait( ); /*philosopher is hungry but unable to
   obtain the chopsticks, so delay himself */
   }
```

Dining Philosophers

```
void putdown(int i)
{          state[i] = thinking;
           // test left and right neighbors
           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) )
          {   //  Only if both neighbors are not eating
                state[i] = eating;
                self[i].signal();
          }
  }
```

Dining Philosophers

```
void init( )
      {
                    for (int i = 0; i < 5; i++)
                              state[i] = thinking;
          }
}
```

- ✓ Philosopher i must invoke the the operations pickup( ) and putdo
  ( ) in the following sequence

```
                dp.pickup( ) ;
                  ….eat…...
                dp.putdown ( )
```

Solaris 2 Synchronization

- ✓ Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing.
- ✓ Uses *adaptive mutexes* for efficiency when protecting data from short code segments.
- ✓ Uses *condition variables* and *readers-writers* locks when longer sections of code need access to data.
- ✓ Uses *turnstiles* to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock.

Windows 2000 Synchronization

- ✓ Uses interrupt masks to protect access to global resources on uniprocessor systems.
- ✓ Uses *spinlocks* on multiprocessor systems.
- ✓ Also provides *dispatcher objects* which may act as wither mutexes and semaphores.
- ✓ Dispatcher objects may also provide *events*. An event acts much like a condition variable

Further reading

- ✓ http://greenteapress.com/semaphores/ The Little Book of Semaphores
- ✓ http://vip.cs.utsa.edu/nsf/pubs/starving/starving.html for a simulator to show how monitors are used to solve Dining philosophers problem
- ✓ http://msdn2.microsoft.com/en-us/library/62246a48.aspx for explanation of monitors in C# and Visual Basic
- ✓ http://www.artima.com/insidejvm/ed2/threadsynch.html

Chapter 20 'Thread Synchronization' of 'Inside the Java Virtual Machine' by Bill Venners