Semaphores

Semaphore proposed by Edsger Dijkstra, is a technique to manage concurrent processes by using a simple integer value, which is known as a semaphore.

- Semaphore is simply a variable which is non-negative and shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.

- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait ( ) and signal ( ).

wait ( ) → P [from the Dutch word proberen, which means "to test"]

signal ( ) → V [from the Dutch word verhogen, which means " to increment"]

wait ( ) → P [from the Dutch word proberen, which means "to test"]

signal ( ) → V [from the Dutch word verhogen, which means " to increment"]

Definition of wait ( ):

```
P (Semaphore S) {
        while (S <= 0)
          ; // no operation
        S-- ;
}
```

wait ( ) → P [from the Dutch word proberen, which means "to test"]

signal ( ) → V [from the Dutch word verhogen, which means " to increment"]

Definition of wait ( ):

```
P (Semaphore S) {
        while (S <= 0)
          ; // no operation
        S-- ;
}
```

Definition of signal ( ):

```
V (Semaphore S) {
        S++ ;
}
```

All the modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly.
That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

wait ( ) → P [from the Dutch word proberen, which means "to test"]

signal ( ) → V [from the Dutch word verhogen, which means " to increment"]

Definition of wait ( ):

```
P (Semaphore S) {
        while (S <= 0)
          ; // no operation
        S-- ;
}
```

Definition of signal ( ):

```
V (Semaphore S) {
        S++ ;
}
```

All the modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly.
That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

wait ( ) → P [from the Dutch word proberen, which means "to test"]

signal ( ) → V [from the Dutch word verhogen, which means " to increment"]

Definition of wait ( ):

```
P (Semaphore S) {
        while (S <= 0)
          ; // no operation
        S-- ;
}
```

Definition of signal ( ):

```
V (Semaphore S) {
        S++ ;
}
```

All the modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

Definition of wait ( ):

```
P (Semaphore S) {
        while (S <= 0)
          ; // no operation
        S-- ;
}
```

Definition of signal ( ):

```
V (Semaphore S) {
        S++ ;
}
```

All the modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly.
That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

Types of Semaphores:

1. Binary Semaphore:

The value of a binary semaphore can range only between 0 and 1. On some systems, binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion.

0 - wait 1 → can enter CS

Definition of wait ( ):

```
P (Semaphore S) {
        while (S <= 0)
          ; // no operation
        S-- ;
}
```

Definition of signal ( ):

```
V (Semaphore S) {
        S++ ;
}
```

All the modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly.
That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

Types of Semaphores:

1. Binary Semaphore:

The value of a binary semaphore can range only between 0 and 1. On some systems, binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion.

0 - wait till free 1 → can enter CS

Initial - 1

Definition of wait ():

```
P (Semaphore S) {
        while (S <= 0)
          ; // no operation
        S-- ;
}
```

Definition of signal ():

```
V (Semaphore S) {
        S++ ;
}
```

All the modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

Types of Semaphores:

1. Binary Semaphore:

The value of a binary semaphore can range only between 0 and 1. On some systems, binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion.

2. Counting Semaphore:

Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Disadvantages of Semaphores

- The main disadvantage of the semaphore definition that was discussed is that it requires busy waiting.
- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.
- Busy waiting wastes CPU cycles that some other process might be able to use productively.
- This type of semaphore is also called a spinlock because the process "spins" while waiting for the lock.

wasting CPU

To overcome the need for busy waiting, we can modify the definition of the wait () and signal () semaphore operations.

- When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait.

- However, rather than engaging in busy waiting, the process can block itself.
- The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state.

To overcome the need for busy waiting, we can modify the definition of the wait () and signal () semaphore operations.

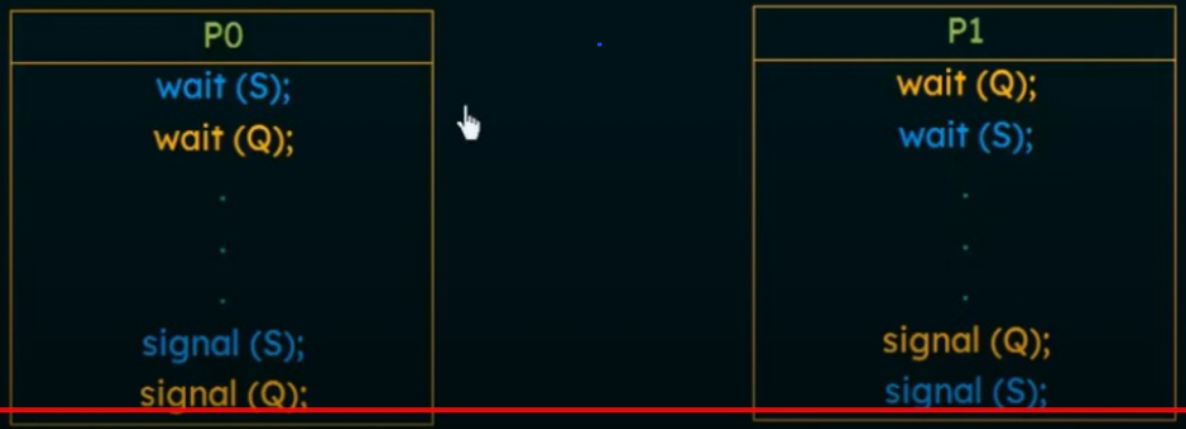
- When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait.
- However, rather than engaging in busy waiting, the process can block itself.
- The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state.
- Then control is transferred to the CPU scheduler, which selects another process to execute.

CPU not wasted

Deadlocks and Starvation

- The implementation of a semaphore with a waiting queue may result in a situatio where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.

- The event in question is the execution of a signal() operation. When such a state i reached, these processes are said to be deadlocked.



Classic Problems of Synchronization

(The Bounded-Buffer Problem)

The Bounded Buffer Problem **(Producer Consumer Problem)**, is one of the classic problems of synchronization.

There is a buffer of n slots and each slot is capable of storing one unit of data.

There are two processes running, namely, Producer and Consumer, which are operating on the buffer.

There is a buffer of n slots and each slot is capable of storing one unit of data.

There are two processes running, namely, Producer and Consumer, which are operating on the buffer.

- The producer tries to insert data into an empty slot of the buffer.
- The consumer tries to remove data from a filled slot in the buffer.
- The Producer must not insert data when the buffer is full.
- The Consumer must not remove data when the buffer is empty.
- The Producer and Consumer should not insert and remove data simultaneously.

Solution to the Bounded Buffer Problem using Semaphores:

We will make use of three semaphores:

1. m (mutex), a binary semaphore which is used to acquire and release the lock.
2. empty, a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.
3. full, a counting semaphore whose initial value is 0.

1. m (mutex), a binary semaphore which is used to acquire and release the lock.
2. empty, a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.
3. full, a counting semaphore whose initial value is 0.

Producer

```
do {
  wait (empty); // wait until empty>0
                and then decrement 'empty'
  wait (mutex); // acquire lock
  /* add data to buffer */
  signal (mutex); // release lock
  signal (full);  // increment 'full'
} while(TRUE)
```

Consumer

```
do {
  wait (full); // wait until full>0 and
               then decrement 'full'
  wait (mutex); // acquire lock
  /* remove data from buffer */
  signal (mutex); // release lock
  signal (empty);  // increment 'empty'
} while(TRUE)
```

Classic Problems of Synchronization

(The Readers-Writers Problem)

- A database is to be shared among several concurrent processes.
- Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.
- We distinguish between these two types of processes by referring to the former as **Readers** and to the latter as **Writers**.
- Obviously, if two readers access the shared data simultaneously, no adverse affects will result.
- However, if a writer and some other thread (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database.

This synchronization problem is referred to as the readers-writers problem.

Solution to the Readers-Writers Problem using Semaphores:

We will make use of two semaphores and an integer variable:

1. mutex, a semaphore (initialized to 1) which is used to ensure mutual exclusion when readcount is updated i.e. when any reader enters or exit from the critical section.
2. wrt, a semaphore (initialized to 1) common to both reader and writer processes.
3. readcount, an integer variable (initialized to 0) that keeps track of how many processes are currently reading the object.

Priority here Reader

Writer Process

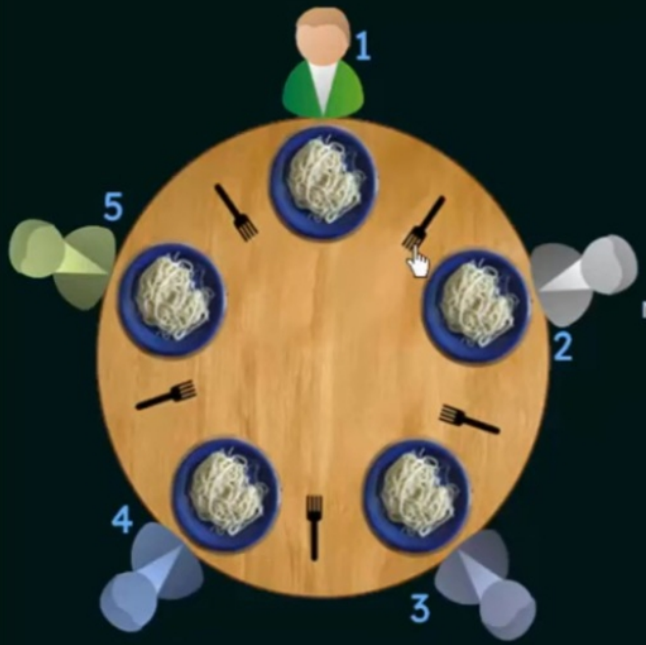
```
do {
/* writer requests for critical
section */
   wait(wrt);
  /* performs the write  */
   // leaves the critical section
   signal(wrt);
} while(true);
```

Reader Process

```
do {
  wait (mutex);
 readcnt++;  // The number of readers has now increased by 1
 if (readcnt==1)
    wait (wrt); // this ensure no writer can enter if there is even one reader
   signal (mutex);  // other readers can enter while this current reader is
                                inside the critical section
 /* current reader performs reading here */
  wait (mutex);
  readcnt--; // a reader wants to leave
  if (readcnt == 0)      //no reader is left in the critical section
     signal (wrt);       // writers can enter
     signal (mutex); // reader leaves
} while(true);
```

Classic Problems of Synchronization

(The Dining-Philosophers Problem)



Philosopher either

Thinks

When a philosopher thinks, he does not interact with his colleagues

Eats

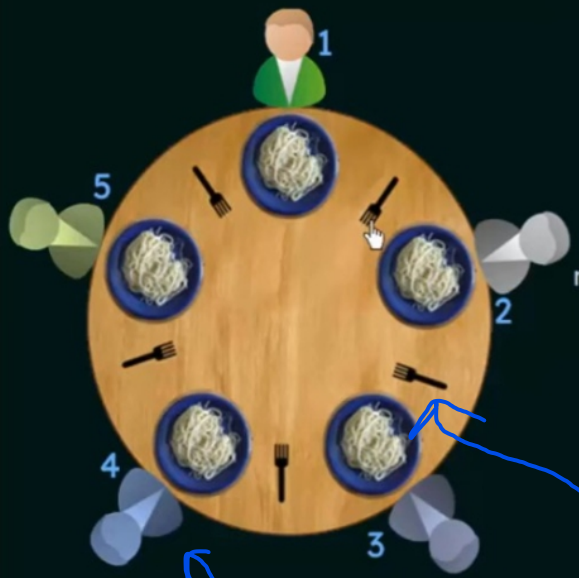
When a philosopher gets hungry he tries to pick up the two forks that are closest to him (left & right). A philosopher may pick up only one fork at a time.

One cannot pick up a fork that is already in the hand of a neighbour.

When a hungry philosopher has both his forks at the same time, he eats without releasing his forks. When he has finished eating, he puts down both of his forks and starts thinking again.

Classic Problems of Synchronization

(The Dining-Philosophers Problem)



Philosopher either

Thinks

When a philosopher thinks, he does not interact with his colleagues

Eats

When a philosopher gets hungry he tries to pick up the two forks that are closest to him (left & right). A philosopher may pick up only one fork at a time.

One cannot pick up a fork that is already in the hand of a neighbour.

When a hungry philosopher has both his forks at the same time, he eats without releasing his forks. When he has finished eating, he puts down both of his forks and starts thinking again.

process

resource

where all the elements of chopstick are initialized to 1.

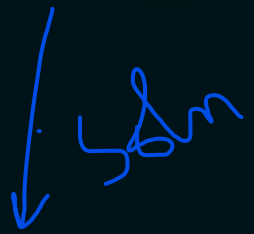
The structure of philosopher i

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

Although this solution guarantees that no two neighbors are eating simultaneously, it could still create a deadlock.

Suppose that all five philosophers become hungry simultaneously and each grabs their left chopstick. All the elements of chopstick will now be equal to 0.

When each philosopher tries to grab his right chopstick, he will be delayed forever.



Some possible remedies to avoid deadlocks:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up his chopsticks only if both chopsticks are available (to do this he must pick them up in a critical section).
- Use an asymmetric solution; that is, an odd philosopher picks up first his left chopstick and then his right chopstick, whereas an even philosopher picks up his right chopstick and then his left chopstick.

Monitors

- A high level abstraction that provides a convenient and effective mechanism for process synchronization.
- A monitor type presents a set of programmer-defined operations that provide mutual exclusion within the monitor.
- The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables.

Syntax of a Monitor

```
monitor monitor_name
{
// shared variable declarations
procedure P1 ( . . . ) {
. . .
}
procedure P2 ( . . . ) {
. . .
}
.
.
procedure Pn ( . . . ) {
. . .
}
initialization code ( . . . ) {
. . .
}
}
```

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

Condition Construct- **condition x, y;**

The only operations that can be invoked on a condition variable are 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.

Syntax of a Monitor

```
monitor monitor_name
{
// shared variable declarations
procedure P1 ( . . . ) {
. . .
}
procedure P2 ( . . . ) {
. . .
}
.
.
procedure Pn ( . . . ) {
. . .
}
initialization code ( . . . ) {
. . .
}
}
```

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

Condition Construct- **condition x, y ;**

The only operations that can be invoked on a condition variable are 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.

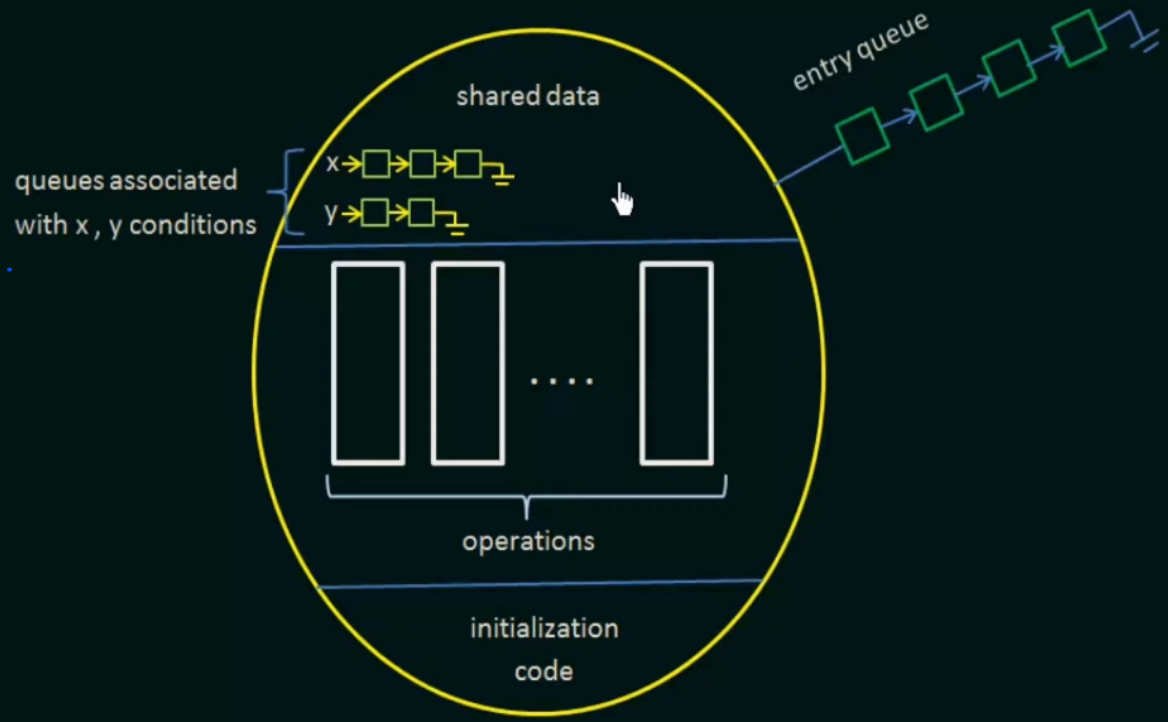


Fig: Schematic view of a monitor

Dining-Philosophers Solution Using Monitors

- We now illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem.
- This solution imposes the restriction that a philosopher may pick up his chopsticks only if both of them are available.
- To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

```
enum { thinking, hungry, eating } state [5] ;
```

- Philosopher i can set the variable state [i] = eating only if his two neighbors are not eating: (state [(i+4) % 5] != eating) and (state [(i+1)% 5] != eating).
- We also need to declare `condition self [5] ;` where philosopher i can delay himself when he is hungry but is unable to obtain the chopsticks he needs.

A monitor solution to the dining-philosopher problem

```
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 ((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 ;
                }
}
```