SNS COLLEGE OF ENGINEERING

Kurumbapalayam(Po), Coimbatore – 641 107
Accredited by NAAC-UGC with 'A' Grade
Approved by AICTE, Recognized by UGC & Affiliated to Anna University, Chennai

Department Of Artificial Intelligence and Data Science

Course Code & Name –23ITB203 & Operating Systems

II Year / IV Semester

Unit 2 - The Classical Problems of Synchronization

Classic problems of Synchronization

(The 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

Classic problems of Synchronization

(The Bounded Buffer Problem)

- 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);
```

Classic problems of Synchronization

(The Bounded Buffer Problem)

- 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);
```

Classic problems of Synchronization

(The Bounded Buffer Problem)

- 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);
```

Classic problems of Synchronization

(The Readers-writers problem)

- A database is to be shared among several concurrent processes.
- Some of the processes may want only to read the database, where as 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 has 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 ensure.
- To ensure that these difficulties do not arise, we require that the writer 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 many processes that are currently reading the object.

Writer Process

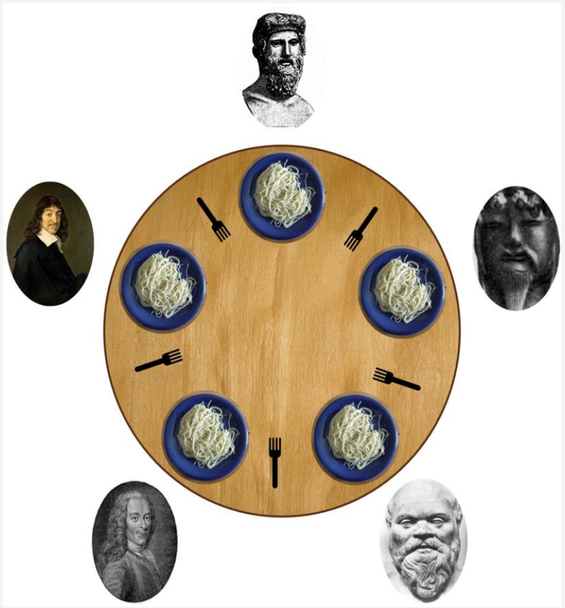
```
do{
/* writer requests for critical section*/
wait(wrt)
/* Perform the write*/

    //leaves the critical section
signal(wrt);
}while(true);
```

Reader Process

```
do{
wait(mutex);
readcnt++; // The number of readers are 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- -; //reader wants to leave
If (readcnt == 0) //no reader is left in the critical section
signal(wrt);
signal(mutex);
}while(true);
```

Classic problems of Synchronization

(The Dining-Philosophers problem)



Philosopher either

Thinks

When a philosopher thinks, he does not interact with her 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 pickup 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.

One simple solution is to represent each fork/chopstick with a semaphore.

A philosopher tries to grab a fork/chopstick by executing a wait() operation on that semaphore.

He releases his fork/ chopsticks by executing the signal() operation on the appropriate semaphores.

Thus, the shared data are

semaphore chopsticks[5];

where all the elements of chopsticks are initialized to 1;

The structure of philosopher i

```
do{
wait(chopostick[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 neightbours are eating simultaneously,
It could still create a deadlock.

Suppose that all five philosophers become hungry simultaneously and each grabs their left chopstick.
All elements of chopstick will 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 in 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, where as an even philosopher picks up her right chopstick and then her left chopstick.