
CODE DIGITAL
HARDWARE DESIGN
INTERNET
SCREEN
DATA
SERVICE
GRAPHIC
INPUT
APPLICATION
OS
APP
WIRELESS
USER
NETWORK
DEVELOPMENT
INFORMATIONCONCEPT
COMPUTING
CONCEPTUAL
MEMORY
BACKGROUND
DESKTOP
PROGRAMMING
OPERATING
SYSTEM
TECHNOLOGY
ALLOCATION
PROGRAM
DEVICE
COMPUTER
SOFTWARE
STORAGE
MOBILE
BUSINESS
WEB
COMMUNICATION
SYMBOL
OUTPUT

Jump2Learn
The Online Learning Place

OPERATING SYSTEM - II

CHAPTER 2 : PROCESS SYNCHRONIZATION

AUTHORS

Mr. Yatin Solanki
M.C.A., Ph.D.(Pursuing)

DOLAT USHA INSTITUTE OF APPLIED SCIENCES, VALSAD

Dr. Jaimin Shukla
M.C.A., M. Phil., Ph.D.(Pursuing)

SUTEX BANK COLLEGE OF COMPUTER APPLICATIONS & SCIENCE, AMROLI

Website : www.jump2learn.com | **Email :** info@jump2learn.com | **Instagram :** www.instagram.com/jump2learn
Facebook : www.facebook.com/Jump2Learn | **Whatsapp :** +91-909-999-0960 | **YouTube :** Jump2Learn

PROCESS COORDINATION

BASIC CONCEPTS OF CONCURRENCY:

- A concurrent program specifies two or more sequential programs (a sequential program specifies sequential execution of a list of statements) that may be executed concurrently as parallel processes.
- For example, an airline reservation system that involves processing transactions from many terminals has a natural specifications as a concurrent program in which each terminal is controlled by its own sequential process.
- Even when processes are not executed simultaneously, it is often easier to structure as a collection of cooperating sequential processes rather than as a single sequential program.
- The operating system consists of a collection of such processes which are basically two types:

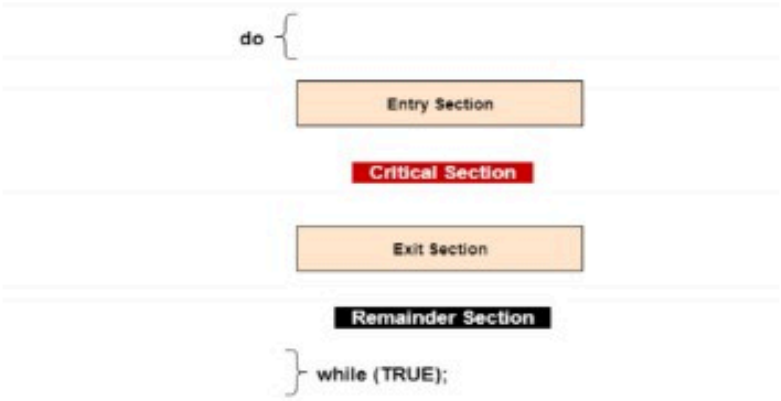
Operating system processes :	Those that execute system code.
User processes :	Those that execute user's code.

- A simple batch operating system can be viewed as 3 processes -**a reader process**, **an executor process** and **a printer process**.
- The reader reads cards from card reader and places card images in an input buffer.
- The executor process reads card images from input buffer and performs the specified computation and store the result in an output buffer.
- The printer process retrieves the data from the output buffer and writes them to a printer Concurrent processing is the basis of operating system which supports multiprogramming.
- The operating system supports concurrent execution of a program without necessarily supporting complex form of memory and file management.
- This form of operation is also known as multitasking.
- Multiprogramming is a more general concept in operating system that supports memory management and file management features, in addition to supporting concurrent execution of programs.

2.1 CRITICAL SECTION PROBLEM

- The critical section is a code segment where the shared variables can be accessed.
- An atomic action is required in a critical section i.e. only one process can execute in its critical section at a time.
- All the other processes have to wait to execute in their critical sections.

A diagram that demonstrates the critical section is as follows:



2.1 CRITICAL SECTION PROBLEM

- In the above diagram, the entry section handles the entry into the critical section.
- It acquires the resources needed for execution by the process.
- The exit section handles the exit from the critical section. It releases the resources and also informs the other processes that the critical section is free.
- The critical section problem needs a solution to synchronize the different processes. The solution to the critical section problem must satisfy the following conditions:

1. **MUTUAL EXCLUSION:**

Mutual exclusion implies that only one process can be inside the critical section at any time. If any other processes require the critical section, they must wait until it is free.

2. **PROGRESS:**

Progress means that if a process is not using the critical section, then it should not stop any other process from accessing it. In other words, any process can enter a critical section if it is free.

3. **BOUNDED WAITING:**

Bounded waiting means that each process must have a limited waiting time. Itt should not wait endlessly to access the critical section.

2.2 PRODUCER/ CONSUMER PROBLEM

- The producer–consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-**process** synchronization problem.
- The problem describes two processes, the producer and the consumer, who share a common, **fixed**-size buffer used as a queue.
- The producer consumer problem is a synchronization problem. There is a fixed size buffer and the producer produces items and enters them into the buffer. The consumer removes the items from the buffer and consumes them.

- A producer should not produce items into the buffer when the consumer is consuming an item from the buffer and vice versa. So the buffer should only be accessed by the producer or consumer at a time.
- The producer consumer problem can be resolved using semaphores. The codes for the producer and consumer process are given as follows:

PRODUCER PROCESS:

The code that defines the producer process is given below:

```
do{
.
. PRODUCE ITEM
.
wait(empty);
wait(mutex);
.
. PUT ITEM IN BUFFER
.
signal(mutex);
signal(full);

}while(1);
```


- In the above code, **mutex, empty** and **full** are semaphores. Here mutex is initialized to 1, empty is initialized to n (maximum size of the buffer) and full is initialized to 0.
- The **mutex** semaphore ensures mutual exclusion. The **empty** and **full** semaphores count the number of empty and full spaces in the buffer.
- After the item is produced, wait operation is carried out on empty. This indicates that the empty space in the buffer has decreased by 1. Then wait operation is carried out on mutex so that consumer process cannot interfere.
- After the item is put in the buffer, signal operation is carried out on mutex and full. The former indicates that consumer process can now act and the latter shows that the buffer is full by 1.

CONSUMER PROCESS:

The code that defines the consumer process is given below:

```
do{

    wait(full);
    wait(mutex);
..
. REMOVE ITEM FROM BUFFER
.
    signal(mutex);
```

```
    signal(empty);
.
. CONSUME ITEM
.
}while(1);
```

- The wait operation is carried out on **full**. This indicates that items in the buffer have decreased by 1. Then wait operation is carried out on **mutex** so that producer process cannot interfere.
- Then the item is removed from buffer. After that, signal operation is carried out on mutex and empty. The former indicates that consumer process can now act and the latter shows that the empty space in the buffer has increased by 1.

2.3 SEMAPHORES

- Semaphore is a simply a variable. This variable is used to solve critical section problem and to achieve process synchronization in the multi processing environment.
- Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations **wait** and **signal** that are used for process synchronization.

The definitions of wait and signal are as follows:

1. **WAIT**

   The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

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

2. **SIGNAL**

   The signal operation increments the value of its argument S.

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

TYPES OF SEMAPHORES:

There are two main types of semaphores i.e. counting semaphores and binary semaphores:

1. **COUNTING SEMAPHORES**

- These are integer value semaphores and have an unrestricted value domain.
- These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources.
- If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.

2. **BINARY SEMAPHORES**

- The binary semaphores are like counting semaphores but their value is restricted to 0 and 1.
- The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0.
- It is sometimes easier to implement binary semaphores than counting semaphores.

ADVANTAGES OF SEMAPHORES:

- Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
- There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
- Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

DISADVANTAGES OF SEMAPHORES:

- Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.
- Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.
- Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

2.4 MONITORS

- Monitors were developed in the 1970s to make it easier to avoid race conditions and implement mutual exclusion.
- A monitor is a collection of procedures, variables, and data structures grouped together in a single module or package.
- Processes can call the monitor procedures but cannot access the internal data structures.

- Monitors have an important property that makes them useful for achieving mutual exclusion: only one process can be active in a monitor at any instant.
- The compiler usually enforces mutual exclusion.
- Typically when a process calls monitor procedure, the first few instruction of procedure will check to see if any other process is currently active in monitor. If so, calling process will be suspended until the other process left the monitor.
- If no other process is using monitor, the calling process may enter.

```
Monitor Producer Consumer
        Condition Full,empty;
        Integer count;
        Procedure insert(item:integer)
        Begin
        If count =N then wait(full);
        Insert_item(item);
        Count := count+1;
        If count =1 then signal(empty)
        End;
        Function remove : integer;
        Begin
        If count=0 then wait (empty)
        Remove = remove_item;
        Count= := count-1 ;
        If count = N – 1 then signal(full)
        End;
        Count :=0;
End monitor;

Procedure producer;
Begin
        While true  do
        Begin
                Item= produce_item;
                ProducerConsumer.insert(item)
        End
End;
```

```
Procedure consumer;
Begin
    While true do
    Begin
    Item= producerConsumer.remove;
    Consume_item(item)
    End;
End;
```

- The solution uses condition variables, along with two operations on them, wait and signal. When a monitor procedure discovers that it cannot continue (e.g., the producer finds the buffer full), it does a wait on some condition variable, full.
    - This action causes the calling process to block. It also allows another process that had been previously prohibited from entering the monitor to enter now.
    - This other process, for example, the consumer, can wake up its sleeping partner by doing a signal on the condition variable that its partner is waiting on.
    - If a signal is done on a condition variable on which several processes are waiting, only one of them, determined by the system scheduler, is revived.
    -

2.5 INTER PROCESS COMMUNICATION

BASIC CONCEPTS OF INTERPROCESS COMMUNICATION AND SYNCHRONIZATION:

- In order to cooperate, concurrently executing processes must communicate and synchronize.
- Interprocess communication is based on the use of shared variables (variables that can be referenced by more than one process) or message passing.
- Synchronization is often necessary when processes communicate.
- Processes are executed with unpredictable speeds. Yet to communicate one process must perform some action such as setting the value of a variable or sending a message that the other detects.
- This only works if the events perform an action or detect an action are constrained to happen in that order.
- Thus one can view synchronization as a set of constraints on the ordering of events.
- The programmer employs a synchronization mechanism to delay execution of a process in order to satisfy such constraints.

2.6 CLASSIAL IPC PROBLEMS

2.6.1 THE DINING PHILOSOPHER

Consider 5 philosophers to spend their lives in thinking & eating. A philosopher shares common circular table surrounded by 5 chairs each occupies by one philosopher. In the center of the table there is a bowl of rice and the table is laid with 6 chopsticks as shown in below figure. When a philosopher thinks she does not interact with her colleagues. From time to time a philosopher gets hungry and tries to pickup two chopsticks that are closest to her. A philosopher may pickup one chopstick or two chopsticks at a time but she cannot pickup a chopstick that is already in hand of the neighbour. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she finished eating, she puts down both of her chopsticks and starts thinking again. This problem is considered as classic synchronization problem. According to this problem each chopstick is represented by a semaphore. A philosopher grabs the chopsticks by executing the wait operation on that semaphore. She releases the chopsticks by executing the signal operation on the appropriate semaphore. The structure of dining philosopher is as follows:

```
do{
Wait ( chopstick [i]);
Wait (chopstick [(i+1)%5]);
. . . . . .
Eat
. . . . . .
Signal (chopstick [i]);
Signal (chopstick [(i+1)%5]);
. . . . . .
Think
. . . . . .
} While (1);
```


2.6.2 THE SLEEPING BARBER PROBLEM

Problem : The analogy is based upon a hypothetical barber shop with one barber. There is a barber shop which has one barber, one barber chair, and n chairs for waiting for customers if there are any to sit on the chair.

- If there is no customer, then the barber sleeps in his own chair.
- When a customer arrives, he has to wake up the barber.
- If there are many customers and the barber is cutting a customer's hair, then the remaining customers either wait if there are empty chairs in the waiting room or they leave if no chairs are empty.

2.2 THE SLEEPING BARBER PROBLEM

SOLUTION :

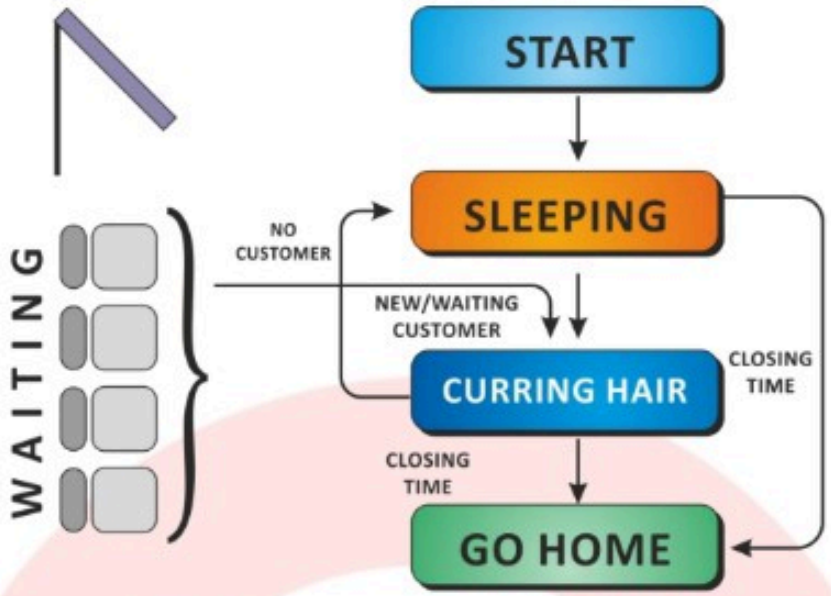
The solution to this problem includes three semaphores. First is for the customer which counts the number of customers present in the waiting room (customer in the barber chair is not included because he is not waiting). Second, the barber 0 or 1 is used to tell whether the barber is idle or is working, And the third mutex is used to provide the mutual exclusion which is required for the process to execute. In the solution, the customer has the record of the number of customers waiting in the waiting room if the number of customers is equal to the number of chairs in the waiting room then the upcoming customer leaves the barbershop.

When the barber shows up in the morning, he executes the procedure barber, causing him to block on the semaphore customers because it is initially 0. Then the barber goes to sleep until the first customer comes up.

When a customer arrives, he executes customer procedure the customer acquires the mutex for entering the critical region, if another customer enters thereafter, the second one will not be able to anything until the first one has released the mutex. The customer then checks the chairs in the waiting room if waiting customers are less then the number of chairs then he sits otherwise he leaves and releases the mutex.

If the chair is available then customer sits in the waiting room and increments the variable waiting value and also increases the customer's semaphore this wakes up the barber if he is sleeping.

At this point, customer and barber are both awake and the barber is ready to give that person a haircut. When the haircut is over, the customer exits the procedure and if there are no customers in waiting room barber sleeps.



2.3 SOLUTION TO SLEEPING BARBER PROBLEM

Algorithm for Sleeping Barber problem:

```
Semaphore Customers = 0;
Semaphore Barber = 0;
Mutex Seats = 1;
int FreeSeats = N;
 Barber {
    while(true) {
        /* waits for a customer (sleeps). */
        down(Customers);
         /* mutex to protect the number of available seats.*/
        down(Seats);
         /* a chair gets free.*/
        FreeSeats++;
        /* bring customer for haircut.*/
        up(Barber);
         /* release the mutex on the chair.*/
        up(Seats);
        /* barber is cutting hair.*/
    }
}
```

```
Customer {
   while(true) {
      /* protects seats so only 1 customer tries to sit
        in a chair if that's the case.*/
      down(Seats); //This line should not be here.
      if(FreeSeats > 0) {
          /* sitting down.*/
          FreeSeats--;
          /* notify the barber. */
          up(Customers);
          /* release the lock */
          up(Seats);
          /* wait in the waiting room if barber is busy. */
          down(Barber);
          // customer is having hair cut
      } else {
          /* release the lock */
          up(Seats);
          // customer leaves
      }
   }
}
```