**Process Synchronization**

Process Synchronization means sharing system resources by the processes in such a way that, Concurrent access to the shared data is handled thereby minimizing the chances of inconsistent data. Maintaining the data consistency demands mechanism to ensure synchronized execution of cooperating processes.

Process Synchronization was introduced to handle problems that arose while multiple process executions. Below are some of the classical problems depicting flaws of process synchronization in systems where cooperating processes are present.

1. Bounded Buffer (Producer-Consumer) Problem

2. Dining Philosophers Problem

3. The Readers Writers Problem

4. Sleeping Barber Problem

**Producer-Consumer Problem:**

The Producer-Consumer problem is the classic problem that is used for multi-process synchronization I.e. synchronization between more than one process. In producer-consumer problem, there is one Producer which produces something and there is one Consumer which consumed the product produced by the Producer. They both(Producer and Consumer) share the same memory buffer which is of fixed-size

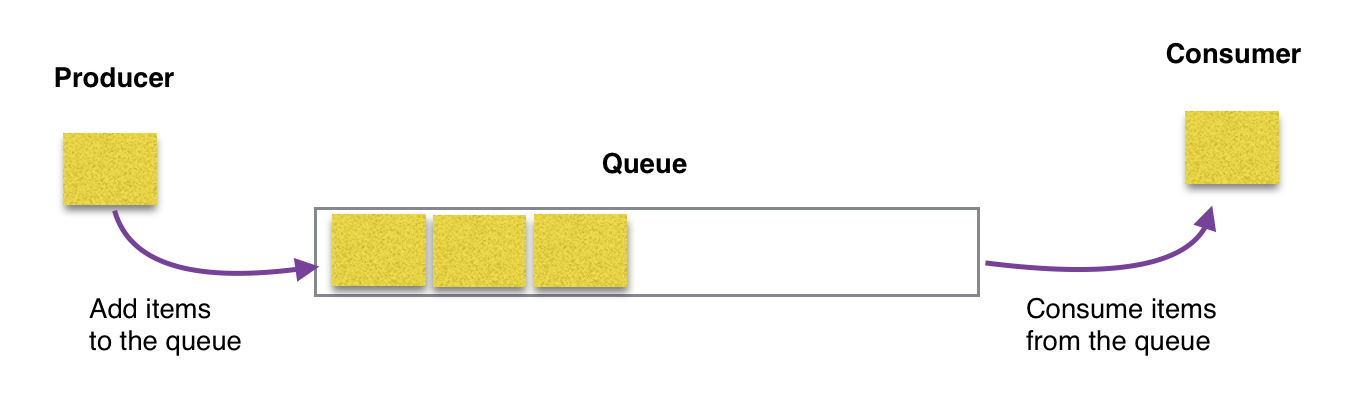
The main job of the Producer is to generate the data, put it into the buffer, and again start generating the data. While the job of the consumer is to consume the data from buffer.

**Problems occurred in Producer-Consumer Problem:**

1. The Producer should produce the data only when the buffer isn’t full. If the buffer is full, Producer shouldn’t be allowed to add any data in the buffer.

2. The Consumer should consume the data only when the buffer isn’t empty. If the buffer is empty, Consumer shouldn’t be allowed to take the data from the buffer.

3. The Producer and the Consumer shouldn’t access the buffer at the same time.



**Semaphores:**

Semaphores are the integer variables that are used to solve the critical section problem by using the two atomic operations, wait and signal that are used for process synchronization.

The definition of wait and signal are as follows:

* **Wait**

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

Wait (S){

While (S<=0);

S--;

T-}

* **Signal:**

The signal operation increments the value of it’s argument S.

Signal (S){

S++;

}

**Solution of Producer-Consumer Problem**:

The above mentioned three problems can be solved by using Semaphore. In producer-consumer problem, we use three semaphore variables:

1. Semaphore S:

This semaphore variable is used to achieve mutual exclusion between processes. By using this variable, either Producer or Consumer will be allowed to use or access the shared buffer at a particular time. This variable is set to 1 initially.

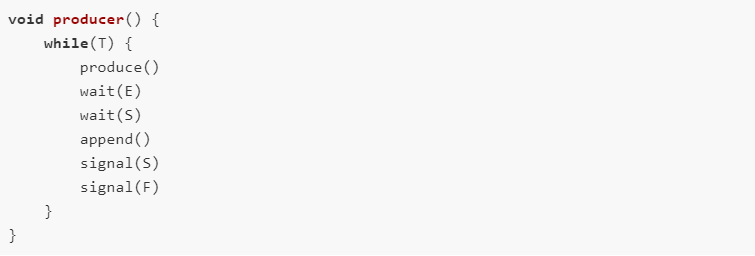
2. Semaphore E:

This semaphore variable is used to define the empty space in the buffer. Initially, it is set to the whole space of the buffer i.e “n” because the buffer is initially empty.

3. Semaphore F:

This semaphore variable is used to define the space that is filled by the producer. Initially, it is set to “0” because there is no space filled by the producer initially.

The following is the pseudo-code for the producer:

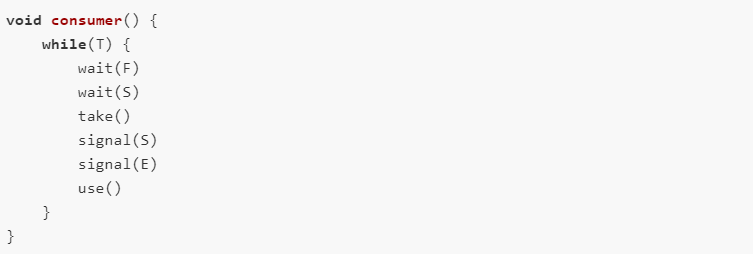


The above code can be summarized as:

* while() is used to produce data, again and again, if it wishes to produce, again and again.
* produce() function is called to produce data by the producer.
* wait(E) will reduce the value of the semaphore variable "E" by one i.e. when the producer produces something then there is a decrease in the value of the empty space in the buffer. If the buffer is full i.e. the vale of the semaphore variable "E" is "0", then the program will stop its execution and no production will be done.
* wait(S) is used to set the semaphore variable "S" to "0" so that no other process can enter into the critical section.
* appends() function is used to append the newly produced data in the buffer.
* signal(s) is used to set the semaphore variable "S" to "1" so that other processes can come into the critical section now because the production is done and the append operation is also done.
* signal(F) is used to increase the semaphore variable "F" by one because after adding the data into the buffer, one space is filled in the buffer and the variable "F" must be updated.

This is how we solve the produce part of the producer-consumer problem.

Now, let's see the consumer solution. The following is the code for the consumer:



The above code can be summarized as:

* while() is used to consume data, again and again, if it wishes to consume, again and again.
* wait(F) is used to decrease the semaphore variable "F" by one because if some data is consumed by the consumer then the variable "F" must be decreased by one.
* wait(S) is used to set the semaphore variable "S" to "0" so that no other process can enter into the critical section.
* take() function is used to take the data from the buffer by the consumer.
* signal(S) is used to set the semaphore variable "S" to "1" so that other processes can come into the critical section now because the consumption is done and the take operation is also done.
* signal(E) is used to increase the semaphore variable "E" by one because after taking the data from the buffer, one space is freed from the buffer and the variable "E" must be increased.
* use() is a function that is used to use the data taken from the buffer by the process So, this is how we can solve the producer-consumer problem.

**Solution of Producer-Consumer Problem using Monitor**:

monitor ProducerConsumer

condition full, empty;

int count;

procedure enter();

{

if (count == N) wait(full); // if buffer is full, block

put\_item(widget); // put item in buffer

count = count + 1; // increment count of full slots

if (count == 1) signal(empty); // if buffer was empty, wake consumer

}

procedure remove();

{

if (count == 0) wait(empty); // if buffer is empty, block

remove\_item(widget); // remove item from buffer

count = count - 1; // decrement count of full slots

if (count == N-1) signal(full); // if buffer was full, wake producer

}

count = 0;

end monitor;

Producer();

{

while (TRUE)

{

make\_item(widget); // make a new item

ProducerConsumer.enter; // call enter function in monitor

}

}

Consumer();

{

while (TRUE)

{

ProducerConsumer.remove; // call remove function in monitor

consume\_item; // consume an item

}}

**Dinning Philosopher Problem:**

The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pickup the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.

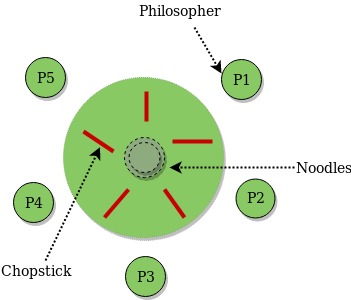


Fig: Dinning Philosopher Problem

The dining philosophers problem is another classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes.

At any instant, a philosopher is either eating or thinking. When a philosopher wants to eat, he uses two chopstick- one from the left and one from the right. When a philosopher wants to think, he keeps down both chopsticks at the original place.

**Solution of Dining Philosopher Problem using Semaphore:**

It is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, then he has to stop at some time. A philosopher is an endless time of thinking and eating.

An array of semaphore, stick[5], for each of the chopsticks. The code for each philosopher is:

wait(stick[i]);

/\*

mod is used because if i=5, next

chopstick is 1 (dining table is circular)

\*/

wait(stick[(i+1)%5]);

/\* eat \*/

signal(stick[i]);

signal(stick[(i+1)%5]);

/\* think \*/}

When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.

But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever. The possible solutions for this are:

 A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.

 Allow only four philosophers to sit at the table . That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

Solution of Dining Philosopher Problem using Monitor:

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:

 THINKING – When philosopher doesn’t want to gain access to either fork.

 HUNGRY – When philosopher wants to enter the critical section.

 SEATING – When philosopher has got both the forks, i.e., he has entered the section.

Philosopher i can set the variable state[i] = EATING only if her two neighbors are not eating

(state[(i+4) % 5] != EATING) and (state[(i+1) % 5] != EATING).

**//Dining-Philosophers Solution Using Monitors**

monitor DP

{

status state[5];

condition self[5];

// Pickup chopsticks

Pickup(int i)

{

// indicate that I’m hungry

state[i] = hungry;

// set state to eating in test()

// only if my left and right neighbors are not eating

test(i);

// if unable to eat, wait to be signaled

if (state[i] != eating)

self[i].wait;

}

// Put down chopsticks

Putdown(int i)

{

// indicate that I’m thinking

state[i] = thinking;

// if right neighbor R=(i+1)%5 is hungry and

// both of R’s neighbors are not eating,

// set R’s state to eating and wake it up by

// signaling R’s CV

test((i + 1) % 5);

test((i + 4) % 5);

}

test(int i)

{

if (state[(i + 1) % 5] != eating

&& state[(i + 4) % 5] != eating

&& state[i] == hungry) {

// indicate that I’m eating

state[i] = eating;

// signal() has no effect during Pickup(),

// but is important to wake up waiting

// hungry philosophers during Putdown()

self[i].signal();

}

}

init()

{

// Execution of Pickup(), Putdown() and test()

// are all mutually exclusive,

// i.e. only one at a time can be executing

for

i = 0 to 4

// Verify that this monitor-based solution is

// deadlock free and mutually exclusive in that

// no 2 neighbors can eat simultaneously

state[i] = thinking;

}

} // end of monitor

**Reader-Writer Problem**:

Reader-Writer problem is another example of classic synchronization .The readers-writers problem relates to an object such as a file that is shared between multiple processes. Some of these processes are readers i.e. they only want to read the data from the object and some of the processes are writers i.e. they want to write into the object.

Any number of readers can read from the shared resource simultaneously, but only one writer can write to the shared resource. When a writer is writing data to the resource, no other process can access the resource. A writer cannot write to the resource if there are non zero number of readers accessing the resource at that time.

What is the problem?

Basically, we perform two operations on a file i.e. read and write. All the processes can perform these two operations. But the problem that arises here is that:

* If a process is writing something on a file and another process also starts writing on the same file at the same time, then the system will go into the inconsistent state. Only one process should be allowed to change the value of the data present in the file at a particular instant of time.
* Another problem is that if a process is reading the file and another process is writing on the same file at the same time, then this may lead to dirty-read because the process writing on the file will change the value of the file, but the process reading that file will read the old value present in the file. So, this should be avoided

**Solution of Reader-Writer Problem using Semaphore:**

Functions used are:

 Wait(): decrements the semaphore value.

 Signal(): increments the semaphore value.

Three variables are used: mutex, wrt, readcnt to implement solution:

 semaphore mutex, wrt: semaphore mutex is used to ensure mutual exclusion when readcnt is updated i.e. when any reader enters or exit from the critical section and semaphore wrt is used by both readers and writers

 int readcnt: readcnt tells the number of processes performing read in the critical section, initially 0

Writer Process:

do {

// writer requests for critical section

wait(wrt);

// performs the write

// leaves the critical section

signal(wrt);

} while(true);

Here Writer requests the entry to critical section. If allowed, I.e. wait() gives a true value, it enters the critical section and performs the write. If not allowed, it keeps on waiting. Then it leaves the critical section

**Reader Process:**

do {

// Reader wants to enter the critical section

wait(mutex);

// The number of readers has now increased by 1

readcnt++;

// there is at least one reader in the critical section

// this ensure no writer can enter if there is even one reader

// thus we give preference to readers here

if (readcnt==1)

wait(wrt);

// other readers can enter while this current reader is inside

// the critical section

signal(mutex);

// current reader performs reading here

wait(mutex); // a reader wants to leave

readcnt--;

// that is, no reader is left in the critical section,

if (readcnt == 0)

signal(wrt); // writers can enter

signal(mutex); // reader leaves

} while(true);

Reader requests the entry to critical section. If allowed, it increments the count of number of the readers inside the critical section, where at the same time, it restricts the entry of any writer into the critical section if any reader is inside. It then, signals mutex as any other reader is allowed to enter while others are already reading.

After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore wrt as now, writer can enter the critical section. If not allowed, it keeps on waiting.

**Solution of Reader-Writer Problem using Monitor:**

monitor ReadersWriters

condition OKtoWrite, OKtoRead;

int ReaderCount = 0;

Boolean busy = false;

procedure StartRead()

{

if (busy) // if database is not free, block

OKtoRead.wait;

ReaderCount++; // increment reader ReaderCount

OKtoRead.signal();

}

procedure EndRead()

{

ReaderCount-- ; // decrement reader ReaderCount

if ( ReaderCount == 0 )

OKtoWrite.signal();

}

procedure StartWrite()

{

if ( busy || ReaderCount != 0 )

OKtoWrite.wait();

busy = true;

}

procedure EndWrite()

{

busy = false;

If (OKtoRead.Queue)

OKtoRead.signal();

else

OKtoWrite.signal();

}

Reader()

{

while (TRUE) // loop forever

{

ReadersWriters.StartRead();

readDatabase(); // call readDatabase function in monitor

ReadersWriters.EndRead();

}

}

Writer()

{

while (TRUE) // loop forever

{

make\_data(&info); // create data to write

ReaderWriters.StartWrite();

writeDatabase(); // call writeDatabase function in monitor

ReadersWriters.EndWrite();

}

}

**Sleeping Barber Problem:**

The barber shop has one barber, one barber chair, and N chairs for waiting customer, if any, to sit in. If there is no customer at present, barber sits down in the barber chair and falls asleep.

When the customer arrives, he has to wake up the barber. If additional customers arrive when the barber is cutting the customer’s hair, they either sit down(if there is an empty chair) or leave the shop(if all chairs are full).

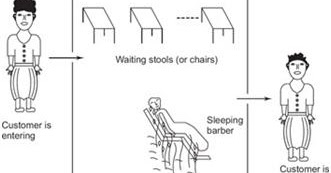


Fig :Sleeping Barber Problem

**Solution of Sleeping Barber using Semaphore:**

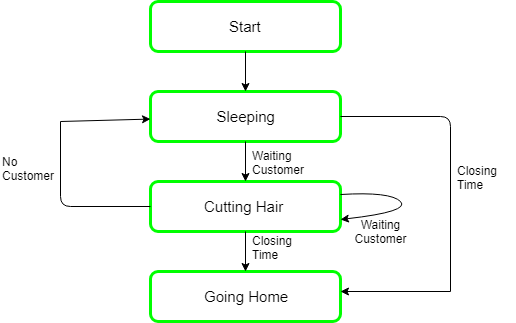
The solution to this problem includes three semaphores.

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.



**Solution of Sleeping Barber Problem using Semaphore:**

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

}

}

}

**Solution of Sleeping Barber using Monitor:**

monitor sleepingbarber;

int waiting;

const int CHAIRS;

condn barber,customer;

void get\_haircut( ){

if (waiting < CHAIRS) {

waiting++;

cSiginal(customer);

cWait(barber);

//get hair cut

waiting--;

}

}

void cut\_hair( ){

while (waiting == 0) cWait(customer)

cSignal(barber);

//cut hair

}

void customer( ) {

gethaircut( );

}

void barber( ) {

while true{cut hair( );}

}

main( )

parbegins(barber,customer)