**1.**

**ANS.**

The algorithm is still ineffective. The reason is that it still continues to break MUTUAL EXCLUSION.

Explanation:

Consider two processes P0 and P1.

* On entering the error “Enter\_Critical()” process P1 achieves turn =1;
* P0 makes “flag[0]=TRUE”;
* P0 executes the condition “ while (flag [1] == TRUE && turn == 0 )”. The condition is true, so P0 enters into Critical section.
* On entering the "Enter. Critical()" process PO achieves turn=0;
* P1 makes “flag[1]=TRUE";
* P1 executes the condition while (flag [0] == TRUE && turn
* Condition is true, so P1 enters into Critical Section.
* Both the processes PO and P1 are in Critical Section.

**2.**

**ANS.**

* When the two processes are running on a shared-memory multiprocessor and the turn variable is used, the busy waiting technique works.
* Two processes employ the turn variable to control access to a critical region.
* The difference in a multiprocessor is that the two processes are running on different CPUs.
* The only stipulation is that turn be stored as a shared memory variable.
* However, the issue is that waiting wastes CPU time.

**3.**

**ANS.**

* When a computer is being built, it is frequently first mimicked using a program that executes one instruction at a time, sequentially.
* As a result, there is no chance of a race condition.
* When two processes execute crucial code at the same time, a race condition occurs.
* Due to race conditions, there may be redundancy and inconsistency in the output.
* However, if each program is executed sequentially, no two programs will be in the critical section at the same time.
* In sequential execution, race conditions can never occur.

**4.**

**ANS.**

A queue implemented using LIFO stack can cause starvation because:

* A process which is pushed last to the queue is always removed first by the scheduler.
* For example, in case of Shortest Job First scheduling, if short duration processes keep on coming, then long processes may be held off indefinitely and will never be removed from the queue. This will lead to starvation.

**5.**

**ANS.**

Among mutexes , empty and full**,  mutex is used for mutual exclusion**. it provides mutual exclusion.

mutual exclusion defines that if a process p executes in its critical section then ,no other process should be executing in their critical section. Mutex is used to ensure this mutual exclusion.

example : producer-consumer problem

**6.**

**ANS.**

 There are two processes in the bounded buffer problem:

* producer process
* consumer process

Problem : how they would get to know when to produce and when to consume ?

Solution : using counting semaphores . Namely ,Full & empty **.**

Impose the following conditions:

* No producer process is allowed to produce a product if the buffer is full. because there was no room in the buffer
* If the buffer is empty, no consumer can consume (i.e., no product can be taken out of the buffer).

But here when statements are reversed then if the buffer is full then the producer process will be allowed to execute and produce products and wait will be applied on the consumer process and the consumer process will be blocked for that period of time as there is nothing to consume ,in reality .

hence full =0;

and empty = buffer\_size[]

**7.**

**ANS.**

Semaphore Customers = 0;  
Semaphore Barber = 0;  
Mutex Seat = 1; int FreeSeats = N;   
  
Barber() {  
while(true) {  
sem\_wait(Customers);  
  
sem\_wait(Seats);  
  
FreeSeats++;  
    
sem\_post(Barber);  
    
sem\_post(Seat);  
  
cut\_hair();  
}  
}  
  
Customer() {  
while(true) {  
  
sem\_wait(Seat);  
if(FreeSeats > 0) {  
    
FreeSeats--;  
    
sem\_post(Customers);  
    
sem\_post(Seat);  
    
sem\_wait(Barber);  
// customer is having hair cut  
get\_haircut();  
} else {  
  
sem\_post(Seat);  
}  
}  
}

**8.**

**ANS.**

define variables:  
semaphore: artist=1

count variable: viewer =0   
semaphore: s=1   
do  
{  
wait(artist); // to ensure only one artist can enter.  
      
signal(artist);  
    
} while(true);  
do  
{  
wait(s);.  
    
viewer++;  
  
if(viewer==1)  
wait(artist);  
    
signal(s);  
      
wait(s);  
    
viewer--;   
    
if(viewer==0)  
signal(artist);  
    
signal(s);  
    
} while(true);

**9.**

**ANS.**

import java.util.concurrent.\*;

public class SleepingBarber extends Thread {

public static Semaphore customers = new Semaphore(0);

public static Semaphore barber = new Semaphore(0);

public static Semaphore accessSeats = new Semaphore(1);

public static final int CHAIRS = 5;

public static int numberOfFreeSeats = CHAIRS;

class Customer extends Thread {

int iD;

boolean notCut = true;

public Customer(int i) {

iD = i;

}

public void run() {

while (notCut) {

try {

accessSeats.acquire();

if (numberOfFreeSeats > 0) {

System.out.println("Customer " + this.iD + " just sat down.");

numberOfFreeSeats--;

customers.release();

accessSeats.release();

try {

barber.acquire();

notCut = false;

this.get\_haircut(); // cutting...

} catch (InterruptedException ex) {

}

} else { // there are no free seats

System.out

.println("There are no free seats. Customer " + this.iD + " has left the barbershop.");

accessSeats.release();

notCut = false;

}

} catch (InterruptedException ex) {

}

}

}

public void get\_haircut() {

System.out.println("Customer " + this.iD + " is getting his hair cut");

try {

sleep(5050);

} catch (InterruptedException ex) {

}

}

}

class Barber extends Thread {

public Barber() {

}

public void run() {

while (true) {

try {

customers.acquire();

accessSeats.release();

numberOfFreeSeats++;

barber.release();

accessSeats.release();

this.cutHair();}

catch (InterruptedException ex) {

}

}

}

public void cutHair() {

System.out.println("The barber is cutting hair");

try {

sleep(5000);

} catch (InterruptedException ex) {

}

}

}

public static void main(String args[]) {

SleepingBarber barberShop = new SleepingBarber();

barberShop.start(); // Let the simulation begin

}

public void run() {

Barber Bob= new Barber();

bob.start();

for (int i = 1; i < 16; i++)

{

Customer aCustomer = new Customer(i);

aCustomer.start();

try {

sleep(2000);

} catch (InterruptedException ex) {

}

;

}

}

}