Implementation & Explanation of Bakery,Dining,Barber Algorithms

Bakery Algorithm(In Java):-

Algorithm:-

On entering the store the customer receives the number.Customer with the lowest number is served.Customers may receive the same number, then the process with the lowest name is served first.Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section .If processes P_i and P_j receive the same number, if $i < j$, then P_i $_i$ served first; else P_j is served first.

$(a, b) < c, d)$ if $a < c$ or if $a = c$ and $b < d$

max $(a_0, \ldots , a_{n-1})$ is a number, k, such that $k \geq a_i$ for $i = 0, \ldots , n-1$

Code:-

```
public class BakeryAlgo extends Thread {

	//Initially,declared 5 threads
	public static final int totalThreads=5;

	//As by the algorthim,holder with smallest ticket can enter first into critical section
	//So assigning a ticket for each thread
	//Used static volatile so that irrespective of any thread or where they are,everyone can access
these arrays
	private static volatile int[] tickets=new int[totalThreads];

	//this array is to check whether it is in critical section or not
	private static volatile boolean[] choosing=new boolean[totalThreads];
	// Variables for the threads.
	public int thread_id; // The id of the current thread.

	//this variable is common for all threads which can enter into critical section
	public static volatile int count = 0;

	public BakeryAlgo (int number)
	{
		//assigning a value for ticket
		thread_id=number;
	}

	//Functions lock,unlock while operating with Critical Section
	public void lock(int number)
	{

		choosing[number] = true;

		// Find the max value and add 1 to get the next available ticket.
		int m = tickets[0];
```

```
		for (int i = 0; i < tickets.length; ++i) {
				if (tickets[i] > m)
					m = tickets[i];
			}
		// Allotting a new ticket value as MAXIMUM + 1
		tickets[number] = m + 1;
		choosing[number] = false;
		// The ENTRY Section starts from here
		for (int other = 0; other < totalThreads; ++other) {

			// Applying the bakery algorithm conditions
			while (choosing[other]) {
			}
			while (tickets[other] != 0 && (tickets[other]
										< tickets[number]
									|| (tickets[other]
											==
tickets[number]
										&& other <
number))) {
			}
		}
	}
	public void unlock(int number)
	{
		tickets[number]=0;
	}

	// Simple test of a global counter.
	public void run() {
		int scale = 2;

		for (int i = 0; i < totalThreads; i++) {

			lock(thread_id);
				// Start of critical section.
				count = count + 1;
				System.out.println("I am " + thread_id + " and count is: " + count);

				// Wait, in order to cause a race condition among the threads.
				try {
					sleep((int) (Math.random() * scale));
				} catch (InterruptedException e) { /* nothing */ }
				// End of critical section.
			unlock(thread_id);

		} // for

	}

	//main function for creating and executing threads
	public static void main(String[] args)
```

```
	{

		//Initializing everything to zero and false
		for (int i = 0; i < totalThreads; ++i) {
			choosing[i]=false;
			tickets[i]=0;
		}

		// Declaring the thread variables
		BakeryAlgo[] threads_create=new BakeryAlgo[totalThreads];

		for (int i = 0; i < totalThreads; ++i) {

			// Creating a new thread with the function
			threads_create[i]=new BakeryAlgo(i);
			//starting thread
			threads_create[i].start();
		}

		for (int i = 0; i < totalThreads; ++i) {

			// Reaping the resources used by
			// all threads once their task is completed !
			try{
			threads_create[i].join();
			}
			catch(InterruptedException e){
				e.printStackTrace();
			}
		}

	}
}
```

Sleeping Barber Problem(In C):-

Algorithm:-

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.

- We use 3 semaphores. Semaphore customers counts waiting customers; semaphore barbers is the number of idle barbers (0 or 1); and mutex is used for mutual exclusion.

Code:-

```
	#include<stdio.h>
#include<pthread.h>
#include<unistd.h>
#include<semaphore.h>
#include<time.h>
#include<stdlib.h>

//lets define max number of customes allowed as 20

int maxCustomer=20;

int everythingOver=0;

//first four variables for semaphores
sem_t bChair; //barber chair
sem_t bRoom; //to enter into waiting room
sem_t bSleep; //barber to sleep until customer arrives
sem_t bSeat; //barber to release seat only after cutting is over

//random wait function

void randwait(int secs) {
	int len;

	// Generate a random number…
	len = (int) ((1 * secs) + 1);
	sleep(len);
}
```

```
}

//one function for barber working,another for customer functioning
void *customer(void *);
void *barber(void *junk) {
        // While there are still customers to be serviced…
        // Our barber is omnicient and can tell if there are
        // customers still on the way to his shop.
        while (!everythingOver) {

                // Sleep until someone arrives and wakes you..
                printf("The barber is sleeping\n\n");
                sem_wait(&bSleep);

                // Skip this stuff at the end…
                if (!everythingOver) {

                        // Take a random amount of time to cut the
                        // customer’s hair.
                        printf("The barber is cutting hair\n\n");
                        randwait(2);
                        printf("The barber has finished cutting hair.\n\n");

                        // Release the customer when done cutting…
                        sem_post(&bSeat);
                }
                else {
                        printf("The barber is going home for the day.\n\n");
                }
        }
}
int main()
{
   //take input from user for number of customes and chairs
```

```
int nCustomers,nChairs;
printf("Enter number of customers and chais:\n");
scanf("%d",&nCustomers);
scanf("%d",&nChairs);

//if number of customers exceeds max then dont allow them
if(nCustomers>maxCustomer)
{
   printf("Only %d customers are allowed , rest can leave\n",maxCustomer);
}

//arrange an value for every customer
int Customers[maxCustomer];
for(int i=1;i<=maxCustomer;i++)
{
   Customers[i]=i;
}
/*
//checking whether its working right
for(int i=1;i<=maxCustomer;i++)
{
   printf("%d",Customers[i]);
}
*/

//Now , create a thread for barber and n customers;
pthread_t tBarber;
pthread_t tCustomer[maxCustomer];

//initialize semaphores
sem_init(&bChair,0,1);
sem_init(&bSeat,0,0);
```

```
    sem_init(&bSleep,0,0);
    sem_init(&bRoom,0,nChairs);

    pthread_create(&tBarber,NULL,barber,NULL);
    for(int j=1;j<=maxCustomer;j++)
    {
        pthread_create(&tCustomer[j],NULL,customer,(void*)&Customers[j]);
    }

    //join threads after waiting them for finish
    for(int k=1;k<=maxCustomer;k++)
    {
        pthread_join(tCustomer[k],NULL);
        sleep(1);
    }

    int everythingOver=1;
    //after everything gets done,wake barber so he will wake up
    sem_post(&bSleep);
    pthread_join(tBarber,NULL);
}

void *customer(void *input)
{
            int number= *(int *)input;
 printf("Customer %d going to enter waiting room\n\n",number);
 sem_wait(&bRoom);

//until the chair becomes free,wait for it
 sem_wait(&bChair);

 printf("Customer %d entered waiting room\n\n",number);
```

```
sem_post(&bRoom);

//then wait for barber chair

//when you acquired chair,wake up the barber as he is sleeping or if he is sleeping
printf("Barber has got customer %d so woke up\n\n",number);
sem_post(&bSleep); //which means you woke up barber

//wait till barber cuts ur hair
sem_wait(&bSeat);

//when done with your cutting,leave the chair for next customer
sem_post(&bChair);

printf("Customer %d is done with his cutting and left\n\n",number);

}
```

Dining Philosopher(In C):-

Algorithm:-

Five Philosophers share a common circular table surrounded by five chairs.Five single chopsticks are available.Whenever a philosopher wants to eat, he tries to pick up two chopsticks that are closest to him/her.A philosopher can not pick the chopstick in the hand of neighbor.After finishing, the philosopher puts back the chopsticks and starts thinking.

A solution of the Dining Philosophers Problem is to use a semaphore to represent a chopstick. A chopstick can be picked up by executing a wait operation on the semaphore and released by executing a signal semaphore.

In the above structure, first wait operation is performed on chopstick[i] and chopstick[(i+1) % 5]. This means that the philosopher i has picked up the chopsticks on his sides. Then the eating function is performed.

After that, signal operation is performed on chopstick[i] and chopstick[(i+1) % 5]. This means that the philosopher i has eaten and put down the chopsticks on his sides. Then the philosopher goes back to thinking.

Code:-

```
	#include<stdio.h>
#include<pthread.h>
#include<unistd.h>
#include<semaphore.h>
#include<time.h>
#include<stdlib.h>


//4 arrays:- semaphore array(sArray),state array(prof_state),input passing array(prof),thread for each prof array(pthread)

//we have five professors
#define n 5

//we have three states for each professor THINKING,EATING,HUNGRY
//So lets define 0,1,2 as eating,hungry,thinking

int prof_state[n];

//lets define a mutex such that no two professors can eat at the same time

sem_t mutex;
sem_t sArray[n]; //for each philosopher , this array is used to control the behaviour

//function for random time
void randwait(int secs) {
	int len;

	// Generate a random number…
	len = (int) ((1 * secs) + 1);
	sleep(len);
}
//total 4 functions :- one for taking forks,one for putting forks,one for testing whether forks free, one for functioning

void* function(void* num);
void test_neighbours(int);
void fork_take(int);
void fork_free(int);


int main()
```

```
{
   int prof[5];
   for(int i=0;i<5;i++)
   {
      prof[i]=i;
   }

   //create a thread for each philosopher
   pthread_t pthread[n];

   sem_init(&mutex,0,1);
   //initialize every semaphore
   for(int i=0;i<5;i++)
   {
      sem_init(&sArray[i],0,0);
   }

   //initialize thread for each professor and then wait for them to complete
   for(int i=0;i<5;i++)
   {
      pthread_create(&pthread[i],NULL,function,&prof[i]);

      printf("Professor %d is thinking right now\n\n",i+1);
   }
   //join threads after completion

   for(int i=0;i<5;i++)
   {
      pthread_join(pthread[i],NULL);
   }
}
void* function(void* num)
{

         while(1)
   {
                  int* input = num;

                  randwait(1);

      //take fork if free
                  fork_take(*input);

                  //randwait(0);

      //after ur completion put out forks for next use
                  fork_free(*input);
         }
}

void fork_take(int inp1)
{
```

```
    //dont enter any other when one person is waiting
    //so we use mutex here
    sem_wait(&mutex);

    //here prof came to take fork which means he is hungry,so change state
    prof_state[inp1]=1;
    int p=inp1+1;
    printf("Professor %d is hungry right now \n \n",p);

    //now check if forks are free
    test_neighbours(inp1);

    //then release the mutex when your work is over
    sem_post(&mutex);

    // wait until the particular prof gets his work done
        sem_wait(&sArray[inp1]);

        randwait(1);
}
void fork_free(int inp2)
{
      //dont enter any other when one person is waiting
    //so we use mutex here
    sem_wait(&mutex);

    //here prof came to put fork which means he is thinking,so change state
    prof_state[inp2]=2;
    int p=inp2+1;

    int l=(inp2+4)%5;
    int r=(inp2+1)%5;
    //int ll=l+1;
    //int rr=r+1;
    printf("Professor %d is putting forks %d , %d right now.. \n\n",p,l+1,r+1);
    printf("Professor %d is thinking right now\n\n",p);
    sem_post(&mutex);

    //now check if forks are free
    test_neighbours(l);
    test_neighbours(r);

    //then release the mutex when your work is over


}
void test_neighbours(int inp3)
{
    int l=(inp3+4)%5;
    int r=(inp3+1)%5;
    //if prof is hungry(0), and neighbours are not eating then proceed
    if(prof_state[inp3]==1 && prof_state[l]!=0 && prof_state[r]!= 0)
```

```
    {
        //you can eat now :) so change the state
        prof_state[inp3]=0;

        randwait(2);

        int ll=l+1;
        int rr=r+1;

        printf("Professor %d have taken forks %d,%d \n\n",inp3+1,ll,rr);
        printf("Professor %d is eating right now..\n",inp3+1);
        sem_post(&sArray[inp3]);

    }
}
```

Notion of Monitors:-

High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

In Concurrent Programming (also known as parallel programming), a **monitor** is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become false. Monitors also have a mechanism for signaling other threads that their condition has been met. A monitor consists of a mutex(lock object and **condition variables.**A condition variable essentially is a container of threads that are waiting for a certain condition

Monitors provide a structured concurrent programming primitive, which is used by processes to ensure exclusive access to resources, and for synchronizing and communicating among users.

A shared data resource can be protected by placing in the monitor.

```
monitor monitor−name
{
        shared variable declarations
        procedure body P1 (… ) {
            . . .
        }
        procedure body P2 (… ) {
            . . .
        }
        procedure body Pn (… ) {
            . . .
        }
        {
```

```
            initialization code
        }
}
```