**11. Simulate dining philosopher’s problem.AIM:**

**Aim:**Implementation of dining philosophers.

**DESCRIPTION:**

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.

**ALGORITHM:**

**Step 1:** start

**Step 2:** declare preprocessor directive n as 4

declare compltedPhilo = 0,I as integer

**Step 3:** declare structure fork with taken as integer and

declare ForkAvil[n] as structure variable of type structure fork

**Step 4:**  declare structure philosp with left,right as integer and

declare Philostatus[n] as structure variable of type structure philosp

**Step 5:** void goForDinner(int philID)

begin

if(Philostatus[philID].left==10 && Philostatus[philID].right==10)

write"Philosopher %d completed his dinner\n",philID+1

else if(Philostatus[philID].left==1 && Philostatus[philID].right==1)

begin

write"Philosopher %d completed his dinner\n",philID+1

Philostatus[philID].left = Philostatus[philID].right = 10 //remembering that he completed dinner by assigning value 10

int otherFork <- philID-1

if(otherFork== -1)

otherFork <- (n-1)

ForkAvil[philID].taken <- ForkAvil[otherFork].taken <- 0 //releasing forks

write"Philosopher %d released fork %d and fork %d\n",philID+1,philID+1,otherFork+1

compltedPhilo++;

end else if

**Step 6:** else if(Philostatus[philID].left==1 && Philostatus[philID].right==0)

begin

if(philID==(n-1))

begin

if(ForkAvil[philID].taken==0)

begin

ForkAvil[philID].taken <- Philostatus[philID].right <- 1;

write"Fork %d taken by philosopher %d\n",philID+1,philID+1

end if

else

begin

write"Philosopher %d is waiting for fork %d\n",philID+1,philID+1

end else

end if

else

begin

int dupphilID <- philID

philID- <- 1

if(philID== -1)

philID <- (n-1);

if(ForkAvil[philID].taken == 0)

begin

ForkAvil[philID].taken <- Philostatus[dupphilID].right <- 1;

write"Fork %d taken by Philosopher %d\n",philID+1,dupphilID+1

end if

else

begin

write"Philosopher %d is waiting for Fork %d\n",dupphilID+1,philID+1

end else

end else

end else if

**Step 7:** else if(Philostatus[philID].left==0)

begin

if(philID==(n-1))

begin

if(ForkAvil[philID-1].taken==0)

begin

ForkAvil[philID-1].taken <- Philostatus[philID].left <- 1

write"Fork %d taken by philosopher %d\n",philID,philID+1

end if

else

begin

write"Philosopher %d is waiting for fork %d\n",philID+1,philID

end else

end if

else

begin

if(ForkAvil[philID].taken == 0)

begin

ForkAvil[philID].taken <- Philostatus[philID].left <- 1;

write"Fork %d taken by Philosopher %d\n",philID+1,philID+1

end if

else

begin

write"Philosopher %d is waiting for Fork %d\n",philID+1,philID+1

end else

end else

end else if

else

begin

end else

end

**Step 8:** int main()

begin

for(i=0;i<n;i++)

ForkAvil[i].taken=Philostatus[i].left=Philostatus[i].right=0;

while(compltedPhilo<n)

begin

for(i=0;i<n;i++)

goForDinner(i);

write"\nTill now num of philosophers completed dinner are %d\n\n",compltedPhilo

end while

return 0;

end

**Step 9:** stop

**SOURCE CODE:**

#include<stdio.h>

#define n 4

int compltedPhilo = 0,i;

struct fork

{

int taken;

}ForkAvil[n];

struct philosp

{

int left;

int right;

}Philostatus[n];

void goForDinner(int philID)

{

if(Philostatus[philID].left==10 && Philostatus[philID].right==10)

printf("Philosopher %d completed his dinner\n",philID+1);

else if(Philostatus[philID].left==1 && Philostatus[philID].right==1)

{

printf("Philosopher %d completed his dinner\n",philID+1);

Philostatus[philID].left = Philostatus[philID].right = 10; //remembering that he completed dinner by assigning value 10

int otherFork = philID-1;

if(otherFork== -1)

otherFork=(n-1);

ForkAvil[philID].taken = ForkAvil[otherFork].taken = 0; //releasing forks

printf("Philosopher %d released fork %d and fork %d\n",philID+1,philID+1,otherFork+1);

compltedPhilo++;

}

else if(Philostatus[philID].left==1 && Philostatus[philID].right==0)

{

if(philID==(n-1))

{

if(ForkAvil[philID].taken==0)

{

ForkAvil[philID].taken = Philostatus[philID].right = 1;

printf("Fork %d taken by philosopher %d\n",philID+1,philID+1);

}

else

{

printf("Philosopher %d is waiting for fork %d\n",philID+1,philID+1);

}

}

else

{

int dupphilID = philID;

philID-=1;

if(philID== -1)

philID=(n-1);

if(ForkAvil[philID].taken == 0)

{

ForkAvil[philID].taken = Philostatus[dupphilID].right = 1;

printf("Fork %d taken by Philosopher %d\n",philID+1,dupphilID+1);

}

else

{

printf("Philosopher %d is waiting for Fork %d\n",dupphilID+1,philID+1);

}

}

}

else if(Philostatus[philID].left==0)

{

if(philID==(n-1))

{

if(ForkAvil[philID-1].taken==0)

{

ForkAvil[philID-1].taken = Philostatus[philID].left = 1;

printf("Fork %d taken by philosopher %d\n",philID,philID+1);

}

else

{

printf("Philosopher %d is waiting for fork %d\n",philID+1,philID);

}

}

else

{

if(ForkAvil[philID].taken = = 0)

{

ForkAvil[philID].taken = Philostatus[philID].left = 1;

printf("Fork %d taken by Philosopher %d\n",philID+1,philID+1);

}

else

{

printf("Philosopher %d is waiting for Fork %d\n",philID+1,philID+1);

}

}

}

else

{

}

}

int main()

{

for(i=0;i<n;i++)

ForkAvil[i].taken=Philostatus[i].left=Philostatus[i].right=0;

while(compltedPhilo<n)

{

for(i=0;i<n;i++)

goForDinner(i);

printf("\nTill now num of philosophers completed dinner are %d\n\n",compltedPhilo);

}

return 0;

}

**OUTPUT:**

Fork 1 taken by Philosopher 1

Fork 2 taken by Philosopher 2

Fork 3 taken by Philosopher 3

Philosopher 4 is waiting for fork 3

Till now num of philosophers completed dinner are 0

Fork 4 taken by Philosopher 1

Philosopher 2 is waiting for Fork 1

Philosopher 3 is waiting for Fork 2

Philosopher 4 is waiting for fork 3

Till now num of philosophers completed dinner are 0

Philosopher 1 completed his dinner

Philosopher 1 released fork 1 and fork 4

Fork 1 taken by Philosopher 2

Philosopher 3 is waiting for Fork 2

Philosopher 4 is waiting for fork 3

Till now num of philosophers completed dinner are 1

Philosopher 1 completed his dinner

Philosopher 2 completed his dinner

Philosopher 2 released fork 2 and fork 1

Fork 2 taken by Philosopher 3

Philosopher 4 is waiting for fork 3

Till now num of philosophers completed dinner are 2

Philosopher 1 completed his dinner

Philosopher 2 completed his dinner

Philosopher 3 completed his dinner

Philosopher 3 released fork 3 and fork 2

Fork 3 taken by philosopher 4

Till now num of philosophers completed dinner are 3

Philosopher 1 completed his dinner

Philosopher 2 completed his dinner

Philosopher 3 completed his dinner

Fork 4 taken by philosopher 4

Till now num of philosophers completed dinner are 3

Philosopher 1 completed his dinner

Philosopher 2 completed his dinner

Philosopher 3 completed his dinner

Philosopher 4 completed his dinner

Philosopher 4 released fork 4 and fork 3

Till now num of philosophers completed dinner are 4\*/

**12. Simulate producer-consumer problem using threads.AIM:**

Implementation of producer and consumer problem.

**DESCRIPTION:**

Producer can produce an item and consumer can consume an item. If the buffer is empty producer can produce an item and consumer can’t consume an item if the buffer is empty. If the buffer is full then consumer can consume an item and producer can’t produce an item if the buffer is full. And while consumer process is executing producer can’t produce an item and while producer process is executing consumer can’t consume an item.

**ALGORITHM:**

**Step 1:** start

**Step 2:** declare necessary variables

Global variables: BUFFERSIZE <- 10 (Use preprocessor directives to declare)

pthread\_t tidp[20],tidc[20]

pthread\_mutex\_t mutex

semm\_t full,empty

int counter,buffer[BUFFERSIZE]

void \*producer(void \*param)

void \*consumer(void \*param)

void initialize()

int read()

void write(int item)

Local variables: n1,n2,I of int type

**Step 3:** call initialize()

**Step 4:** read number of producers(n1)

**Step 5:** read number of consumers(n2)

**Step 6:** for I <- 0 repeat to n1

Call pthread\_create (&tidp[i],NULL,producer,NULL)

**Step 7:** for I <- 0 repeat to n2

Call pthread\_create (&tidc[i],NULL,consumer,NULL)

**Step 8:** for I <- 0 repeat to n1

Call pthread\_join(tidp[i],NULL)

**Step 9:** for I <- 0 repeat to n2

Call pthread\_join(tidc[i],NULL)

**Step 10:** sleep(5)

**Step 11:** exit(0)

Void initialize()

**Step 1:** call pthread\_mutex\_init(&mutex,NULL)

**Step 2:** call sem\_init(&full,1,0)

**Step 3:** call sem\_init(&empty,1,BUFFERSIZE)

**Step 4:** counter <- 0

Void write(int item)

**Step 1:** buffer[counter++]=item

Int read()

**Step 1:** return(buffer[--counter])

Void \*producer(void \*param)

**Step 1:** declare waittime,item,I of int type

**Step 2:** item = rand()%5

**Step 3:** waittime = rand()%5

**Step 4:** sem\_wait(&empty)

**Step 5:** pthread\_mutex\_lock(&mutex)

**Step 6:** write “producer has produce an item”,item

**Step 7:** call write(item)

**Step 8:** pthread\_mutex\_unlock(&mutex)

**Step 9:** call sem\_post(&full)

Void \*consumer(void \*param)

**Step 1:** declare waittime,item,I of int type

**Step 2:** waittime = rand()%5

**Step 3:** sem\_wait(&full)

**Step 4:** pthread\_mutex\_lock(&mutex)

**Step 5:** item = call read(item) function

**Step 6:** write “consumer has consumed an item”,item

**Step 7:** pthread\_mutex\_unlock(&mutex) call write(item)

**Step 8:** call sem\_post(&empty)

**SOURCE CODE:**

#include <stdio.h>

#include <stdlib.h>

// Initialize a mutex to 1

int mutex = 1;

// Number of full slots as 0

int full = 0;

// Number of empty slots as size

// of buffer

int empty = 10, x = 0;

// Function to produce an item and

// add it to the buffer

void producer()

{

// Decrease mutex value by 1

--mutex;

// Increase the number of full

// slots by 1

++full;

// Decrease the number of empty

// slots by 1

--empty;

// Item produced

x++;

printf("\nProducer produces"

"item %d",

x);

// Increase mutex value by 1

++mutex;

}

// Function to consume an item and

// remove it from buffer

void consumer()

{

// Decrease mutex value by 1

--mutex;

// Decrease the number of full

// slots by 1

--full;

// Increase the number of empty

// slots by 1

++empty;

printf("\nConsumer consumes "

"item %d",

x);

x--;

// Increase mutex value by 1

++mutex;

}

// Driver Code

int main()

{

int n, i;

printf("\n1. Press 1 for Producer"

"\n2. Press 2 for Consumer"

"\n3. Press 3 for Exit");

// Using '#pragma omp parallel for'

// can give wrong value due to

// synchronization issues.

// 'critical' specifies that code is

// executed by only one thread at a

// time i.e., only one thread enters

// the critical section at a given time

#pragma omp critical

for (i = 1; i > 0; i++) {

printf("\nEnter your choice:");

scanf("%d", &n);

// Switch Cases

switch (n) {

case 1:

// If mutex is 1 and empty

// is non-zero, then it is

// possible to produce

if ((mutex == 1)

&& (empty != 0)) {

producer();

}

// Otherwise, print buffer

// is full

else {

printf("Buffer is full!");

}

break;

case 2:

// If mutex is 1 and full

// is non-zero, then it is

// possible to consume

if ((mutex == 1)

&& (full != 0)) {

consumer();

}

// Otherwise, print Buffer

// is empty

else {

printf("Buffer is empty!");

}

break;

// Exit Condition

case 3:

exit(0);

break;

}

}

}

o/p: $ gcc pc.c –o pc

$pc enter

