**Assignment 2**

**Question1:**

#include<stdio.h>

int main()

{

int i, limit, total = 0, x, counter = 0, time\_quantum;

int wait\_time = 0, turnaround\_time = 0, arrival\_time[10], burst\_time[10], temp[10];

float average\_wait\_time, average\_turnaround\_time;

printf("\t--Round Robin Algorithm--\n");

printf("\nEnter Total Number of Processes:\t");

scanf("%d", &limit);

x = limit;

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

{

printf("\nEnter Details of Process[%d]\n", i + 1);

printf("Arrival Time:\t");

scanf("%d", &arrival\_time[i]);

printf("Burst Time:\t");

scanf("%d", &burst\_time[i]);

temp[i] = burst\_time[i];

}

printf("\nEnter Time Quantum:\t");

scanf("%d", &time\_quantum);

printf("\nProcess ID\t\tBurst Time\t Turnaround Time\t Waiting Time\n");

for(total = 0, i = 0; x != 0;)

{

if(temp[i] <= time\_quantum && temp[i] > 0)

{

total = total + temp[i];

temp[i] = 0;

counter = 1;

}

else if(temp[i] > 0)

{

temp[i] = temp[i] - time\_quantum;

total = total + time\_quantum;

}

if(temp[i] == 0 && counter == 1)

{

x--;

printf("\nProcess[%d]\t\t%d\t\t %d\t\t\t %d", i + 1, burst\_time[i], total - arrival\_time[i], total - arrival\_time[i] - burst\_time[i]);

wait\_time = wait\_time + total - arrival\_time[i] - burst\_time[i];

turnaround\_time = turnaround\_time + total - arrival\_time[i];

counter = 0;

}

if(i == limit - 1)

{

i = 0;

}

else if(arrival\_time[i + 1] <= total)

{

i++;

}

else

{

i = 0;

}

}

average\_wait\_time = wait\_time \* 1.0 / limit;

average\_turnaround\_time = turnaround\_time \* 1.0 / limit;

printf("\n\nAverage Waiting Time:\t%f", average\_wait\_time);

printf("\nAvg Turnaround Time:\t%f\n", average\_turnaround\_time);

return 0;

}

**Question2:**

#include<stdio.h>

#include<conio.h>

void main()

{

int process,resource,i,j,instanc,k=0,count1=0,count2=0; //count,k variables are taken for counting purpose

printf("\t----Banker's Algorithm----\n");

printf("Enter No. of Process: ");

scanf("%d",&process);

printf("\n"); //Entering No. of Processes

printf("Enter No. of Resources: ");

scanf("%d",&resource);

printf("\n"); //No. of Resources

int avail[resource],max[process][resource],allot[process][resource],need[process][resource],completed[process];

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

completed[i]=0; //Setting Flag for uncompleted Process

printf("Enter No. of Available Instances:\n");

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

{

printf("\t\t");

scanf("%d",&instanc);

avail[i]=instanc; // Storing Available instances

}

printf("Enter Maximum No. of instances of resources that a Process need: ");

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

{

printf("\n\t\tFor P[%d]:\n",i);

for(j=0;j<resource;j++)

{

printf("\t\t");

scanf("%d",&instanc);

max[i][j]=instanc;

}

}

printf("Enter no. of instances already allocated to process of a resource: ");

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

{

printf("\n\t\tFor P[%d]:\n",i);

for(j=0;j<resource;j++)

{

printf("\t\t");

scanf("%d",&instanc);

allot[i][j]=instanc;

need[i][j]=max[i][j]-allot[i][j]; //calculating Need of each process

}

}

printf("Safe Sequence is: ");

while(count1!=process)

{

count2=count1;

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

{

for(j=0;j<resource;j++)

{

if(need[i][j]<=avail[j])

{

k++;

}

}

if(k==resource && completed[i]==0 )

{

printf("P[%d]\t",i);

completed[i]=1;

for(j=0;j<resource;j++)

{

avail[j]=avail[j]+allot[i][j];

}

count1++;

}

k=0;

}

if(count1==count2)

{

printf("Stop ..After this.....Deadlock: ");

break;

}

}

getch();

}

**Question 3:**

// File: deadlock.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <pthread.h>

pthread\_mutex\_t mutex1 = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_mutex\_t mutex2 = PTHREAD\_MUTEX\_INITIALIZER;

int flag=0,flag2=0;

// These two functions will run concurrently.

void\* print\_i(void \*ptr) {

pthread\_mutex\_lock(&mutex1);flag=1;

printf("Mutex 1 locked by i\n");

while(flag2==0);printf("!!DeadLock situation!!\nPress CTRL-C to terminate program\n");

pthread\_mutex\_lock(&mutex2);

printf("Process i executing...\n");

pthread\_mutex\_unlock(&mutex2);

pthread\_mutex\_unlock(&mutex1);

}

void\* print\_j(void \*ptr) {

while(flag==0);

pthread\_mutex\_lock(&mutex2);flag2=1;

printf("Mutex 2 locked by j\n");printf("!!DeadLock situation!!\nPress CTRL-C to terminate program\n");

pthread\_mutex\_lock(&mutex1);

printf("Process j executing...\n");

pthread\_mutex\_unlock(&mutex1);

pthread\_mutex\_unlock(&mutex2);

}

int main() {

pthread\_t t1, t2;

printf("\t--DEADLOCK SITUATION BY MUTEX LOCKS--\n");

int iret1 = pthread\_create(&t1, NULL, print\_i, NULL);

int iret2 = pthread\_create(&t2, NULL, print\_j, NULL);

pthread\_join(t1,NULL);

pthread\_join(t2,NULL);

exit(0); //never reached.

}

**Question4:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <semaphore.h>

#include <pthread.h>

#include <unistd.h>

struct Philosopher

{

int number;

int leftForkIndex;

int rightForkIndex;

int eatenTimes;

pthread\_t thread\_id;

};

struct Fork

{

int index;

sem\_t mutex;

};

struct Fork\* forks;

sem\_t global\_mutex;

int NotEatenCount = 0;

int is\_finished()

{

int counter = 0;

sem\_wait(&global\_mutex);

counter = NotEatenCount;

sem\_post(&global\_mutex);

/\* return true, if NotEatenCount = 0 \*/

return counter==0;

//return counter --> causes starvation

}

void\* philosopher\_thread(void \*argument)

{

struct Philosopher\* philosopher = (struct Philosopher\*)argument;

int again = 1;

while(again)

{

/\* think at start \*/

printf("Philosopher %d is Thinking\n", philosopher->number);

/\* think for some time \*/

/\* There is delay from 0.5 to 3.5 second \*/

usleep(500\*(1000 + 100\*(rand() % 60)));

/\* after thinking start to eat \*/

printf("Philosopher %d is trying to eat...\n", philosopher->number);

/\* try to get left fork

\* There is used sem\_trywait, not sem\_wait.

\* it makes possible to resolve deadlocks

\*/

if (sem\_trywait(&forks[philosopher->leftForkIndex].mutex)==0)

{

/\* if philosohers gets left fork successfully \*/

/\* generate random waiting time for right fork \*/

int waiting\_times = 10 + rand() % 50; /\* returns number in [10..59] interval \*/

/\* check waiting time for right fork is ot expired \*/

while(waiting\_times>0)

{

/\* try to get right form

\* There is used sem\_trywait, not sem\_wait.

\* it makes possible to resolve deadlocks

\*/

if (sem\_trywait(&forks[philosopher->rightForkIndex].mutex)==0)

{

/\* philisopehrs gets 2 forks!\*/

printf("Philosopher %d is Eating\n", philosopher->number);

/\* check this philisopers eaten before at least once\*/

if (!philosopher->eatenTimes)

{

/\* if didn't eat,

\* decrement "Not Eaten Philosophers Count

\*/

sem\_wait(&global\_mutex);

NotEatenCount--;

sem\_post(&global\_mutex);

}

/\* increments eaten time for this philosopehers \*/

philosopher->eatenTimes++;

/\* eat for some time \*/

/\* There is delay from 0.5 to 3.5 second \*/

usleep(500\*(1000 + 100\*(rand() % 60)));

/\* put left fork on table \*/

sem\_post(&forks[philosopher->rightForkIndex].mutex);

/\* if it's here, it means waiting\_times was greater than 0

\* Therefhore make waiting\_times negative in order to mark

\* this philosopers eaten succesfuuly at this time

\*/

waiting\_times =- waiting\_times;

} else {

/\* Cannot get right fork

\* decrements timer for waiting right fork

\*/

waiting\_times--;

/\* delay for 0.1 sec\*/

usleep(100000);

/\* waiting\_times has [10..59] value

\* Therefore philosopher waits from 1 to 6 seconds for right fork

\*/

}

}

/\* if waiting\_times is 0, it means philosopers cannot get right fork and

\* waiting time is expired - he will release left fork, despite he is hungry \*/

if (waiting\_times==0)

{

printf("Philosopher %d cannot take second fork...\n", philosopher->number);

}

/\* put left fork on table \*/

sem\_post(&forks[philosopher->leftForkIndex].mutex);

} else {

printf("Philosopher %d cannot eat at this moment...\n", philosopher->number);

}

/\* Checking for if all philosophers done eating \*/

again = !is\_finished();

}

}

int main(int argc, char\* argv[])

{

struct Philosopher\* philosophers;

int i, count = 5;

/\* check command line arguments \*/

if (argc>=2)

/\* gets number of philosophers from command line\*/

count = atoi(argv[1]);

srand((unsigned int)time(NULL));

/\* if arguments is invalid \*/

if (count<2 || count>1000)

/\* replace with 5 \*/

count = 5;

/\* create array of structures for philosophers \*/

philosophers = (struct Philosopher\*) malloc(sizeof(struct Philosopher) \* count);

/\* create array of structures for forks \*/

forks = (struct Fork\*)malloc(sizeof(struct Fork) \* count);

/\* create global mutex in order to determinate all philophers eaten at least 1 time \*/

sem\_init(&global\_mutex,0,1);

/\* at start, no philosophers eaten \*/

NotEatenCount = count;

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

{

/\* initialzied mutex of each fork \*/

sem\_init(&forks[i].mutex,0,1);

/\* Each philosopher not yet eaten \*/

philosophers[i].eatenTimes = 0;

/\* Set number of philosophers (used for output) \*/

philosophers[i].number = i + 1;

/\* Set index of left fork \*/

philosophers[i].leftForkIndex = i;

/\* Set index of rigth fork

\* There indices will be used later

\*/

if (i+1==count)

philosophers[i].rightForkIndex = 0;

else

philosophers[i].rightForkIndex = i+1;

}

/\* run philosophers thread \*/

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

pthread\_create(&philosophers[i].thread\_id, NULL, philosopher\_thread, &philosophers[i]);

/\* check is finished \*/

while(!is\_finished())

usleep(100);

/\* finisgh all threads \*/

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

pthread\_join(philosophers[i].thread\_id, NULL);

/\* prints statistic \*/

printf("\nStatistics:\n");

for(i=0;i<count;i++){

printf("Philosopher %d eaten for %d times\n", philosophers[i].number, philosophers[i].eatenTimes);

}

/\* free all dynamically allocated memory \*/

free(forks);

free(philosophers);

return 0;

}

**Question5:**

/\* To compile me for Unix, type: gcc -o filename filename.c -lpthread \*/

#include <pthread.h>

#include <stdio.h>

#define BSIZE 4

#define NUMITEMS 30

typedef struct {

char buf[BSIZE];

int occupied;

int nextin, nextout;

pthread\_mutex\_t mutex;

pthread\_cond\_t more;

pthread\_cond\_t less;

} buffer\_t;

buffer\_t buffer;

void \* producer(void \*);

void \* consumer(void \*);

#define NUM\_THREADS 2

pthread\_t tid[NUM\_THREADS]; /\* array of thread IDs \*/

main( int argc, char \*argv[] )

{

int i;

pthread\_cond\_init(&(buffer.more), NULL);

pthread\_cond\_init(&(buffer.less), NULL);

pthread\_create(&tid[1], NULL, consumer, NULL);

pthread\_create(&tid[0], NULL, producer, NULL);

for ( i = 0; i < NUM\_THREADS; i++)

pthread\_join(tid[i], NULL);

printf("\nmain() reporting that all %d threads have terminated\n", i);

} /\* main \*/

void \* producer(void \* parm)

{

char item[NUMITEMS]="IT'S A SMALL WORLD, AFTER ALL.";

int i;

printf("producer started.\n");

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

{ /\* produce an item, one character from item[] \*/

if (item[i] == '\0') break; /\* Quit if at end of string. \*/

pthread\_mutex\_lock(&(buffer.mutex));

if (buffer.occupied >= BSIZE) printf("producer waiting.\n");

while (buffer.occupied >= BSIZE)

pthread\_cond\_wait(&(buffer.less), &(buffer.mutex) );

printf("producer executing.\n");

buffer.buf[buffer.nextin++] = item[i];

buffer.nextin %= BSIZE;

buffer.occupied++;

/\* now: either buffer.occupied < BSIZE and buffer.nextin is the index

of the next empty slot in the buffer, or

buffer.occupied == BSIZE and buffer.nextin is the index of the

next (occupied) slot that will be emptied by a consumer

(such as buffer.nextin == buffer.nextout) \*/

pthread\_cond\_signal(&(buffer.more));

pthread\_mutex\_unlock(&(buffer.mutex));

}

printf("producer exiting.\n");

pthread\_exit(0);

}

void \* consumer(void \* parm)

{

char item;

int i;

printf("consumer started.\n");

for(i=0;i<NUMITEMS;i++){

pthread\_mutex\_lock(&(buffer.mutex) );

if (buffer.occupied <= 0) printf("consumer waiting.\n");

while(buffer.occupied <= 0)

pthread\_cond\_wait(&(buffer.more), &(buffer.mutex) );

printf("consumer executing.\n");

item = buffer.buf[buffer.nextout++];

printf("%c\n",item);

buffer.nextout %= BSIZE;

buffer.occupied--;

/\* now: either buffer.occupied > 0 and buffer.nextout is the index

of the next occupied slot in the buffer, or

buffer.occupied == 0 and buffer.nextout is the index of the next

(empty) slot that will be filled by a producer (such as

buffer.nextout == buffer.nextin) \*/

pthread\_cond\_signal(&(buffer.less));

pthread\_mutex\_unlock(&(buffer.mutex));

}

printf("consumer exiting.\n");

pthread\_exit(0);

}

**Question6:**

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#include <time.h>

#include <semaphore.h>

#include <unistd.h>

#define BSIZE 4

#define NUMITEMS 50

#define NUM\_THREADS 30

typedef struct {

char buf[BSIZE];

int occupied;

int nextin, nextout;

pthread\_mutex\_t mutex;

pthread\_cond\_t more;

pthread\_cond\_t less;

} buffer\_t;

buffer\_t buffer;

void \* producer(void \*);

void \* consumer(void \*);

void \* Round\_Robin(void\* ptr);

void \* Bankers\_Algo(void\* ptr);

void \* Mutex\_locks(void\* ptr);

void \* dining\_phil(void\* ptr);

void \* bded\_buff(void\* ptr);

void printprograms();

pthread\_t tid[NUM\_THREADS]; /\* array of thread IDs \*/

pthread\_mutex\_t mutex1 = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_mutex\_t mutex2 = PTHREAD\_MUTEX\_INITIALIZER;

int flag=0,flag2=0;

void\* print\_i(void \*ptr) {

pthread\_mutex\_lock(&mutex1);flag=1;

printf("Mutex 1 locked by i\n");

while(flag2==0);printf("!!DeadLock situation!!\nPress CTRL-C to terminate program\n");

pthread\_mutex\_lock(&mutex2);

printf("Process i executing...\n");

pthread\_mutex\_unlock(&mutex2);

pthread\_mutex\_unlock(&mutex1);

}

void\* print\_j(void \*ptr) {

while(flag==0);

pthread\_mutex\_lock(&mutex2);flag2=1;

printf("Mutex 2 locked by j\n");printf("!!DeadLock situation!!\nPress CTRL-C to terminate program\n");

pthread\_mutex\_lock(&mutex1);

printf("Process j executing...\n");

pthread\_mutex\_unlock(&mutex1);

pthread\_mutex\_unlock(&mutex2);

}

struct Philosopher

{

int number;

int leftForkIndex;

int rightForkIndex;

int eatenTimes;

pthread\_t thread\_id;

};

struct Fork

{

int index;

sem\_t mutex;

};

struct Fork\* forks;

sem\_t global\_mutex;

int NotEatenCount = 0;

int is\_finished()

{

int counter = 0;

sem\_wait(&global\_mutex);

counter = NotEatenCount;

sem\_post(&global\_mutex);

/\* return true, if NotEatenCount = 0 \*/

return counter==0;

//return counter --> causes starvation

}

void\* philosopher\_thread(void \*argument)

{

struct Philosopher\* philosopher = (struct Philosopher\*)argument;

int again = 1;

while(again)

{

/\* think at start \*/

printf("Philosopher %d is Thinking\n", philosopher->number);

/\* think for some time \*/

/\* There is delay from 0.5 to 3.5 second \*/

usleep(500\*(1000 + 100\*(rand() % 60)));

/\* after thinking start to eat \*/

printf("Philosopher %d is trying to eat...\n", philosopher->number);

/\* try to get left fork

\* There is used sem\_trywait, not sem\_wait.

\* it makes possible to resolve deadlocks

\*/

if (sem\_trywait(&forks[philosopher->leftForkIndex].mutex)==0)

{

/\* if philosohers gets left fork successfully \*/

/\* generate random waiting time for right fork \*/

int waiting\_times = 10 + rand() % 50; /\* returns number in [10..59] interval \*/

/\* check waiting time for right fork is ot expired \*/

while(waiting\_times>0)

{

/\* try to get right form

\* There is used sem\_trywait, not sem\_wait.

\* it makes possible to resolve deadlocks

\*/

if (sem\_trywait(&forks[philosopher->rightForkIndex].mutex)==0)

{

/\* philisopehrs gets 2 forks!\*/

printf("Philosopher %d is Eating\n", philosopher->number);

/\* check this philisopers eaten before at least once\*/

if (!philosopher->eatenTimes)

{

/\* if didn't eat,

\* decrement "Not Eaten Philosophers Count

\*/

sem\_wait(&global\_mutex);

NotEatenCount--;

sem\_post(&global\_mutex);

}

/\* increments eaten time for this philosopehers \*/

philosopher->eatenTimes++;

/\* eat for some time \*/

/\* There is delay from 0.5 to 3.5 second \*/

usleep(500\*(1000 + 100\*(rand() % 60)));

/\* put left fork on table \*/

sem\_post(&forks[philosopher->rightForkIndex].mutex);

/\* if it's here, it means waiting\_times was greater than 0

\* Therefhore make waiting\_times negative in order to mark

\* this philosopers eaten succesfuuly at this time

\*/

waiting\_times =- waiting\_times;

} else {

/\* Cannot get right fork

\* decrements timer for waiting right fork

\*/

waiting\_times--;

/\* delay for 0.1 sec\*/

usleep(100000);

/\* waiting\_times has [10..59] value

\* Therefore philosopher waits from 1 to 6 seconds for right fork

\*/

}

}

/\* if waiting\_times is 0, it means philosopers cannot get right fork and

\* waiting time is expired - he will release left fork, despite he is hungry \*/

if (waiting\_times==0)

{

printf("Philosopher %d cannot take second fork...\n", philosopher->number);

}

/\* put left fork on table \*/

sem\_post(&forks[philosopher->leftForkIndex].mutex);

} else {

printf("Philosopher %d cannot eat at this moment...\n", philosopher->number);

}

/\* Checking for if all philosophers done eating \*/

again = !is\_finished();

}

}

///////////////////////MAIN FUNCITON OF THE PROOGRAM//////////////////////

int main(){

int t;

printf("\t--Multithreaded Program to simultaneously execute N processes in threads--\n");

printf("Enter no of programs to run (max 5): ");

scanf("%d",&t);

printprograms();

printf("Enter the program numbers to be added to thread (seperated by spaces): ");

int A[t];

int i;

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

scanf("%d",&A[i]);

for(i=0;i<t;i++){

pthread\_t thread\_id;

switch(A[i]){

case(1):{

pthread\_create(&thread\_id,NULL,Round\_Robin,NULL);

break;

}

case (2):{

pthread\_create(&thread\_id,NULL,Bankers\_Algo,NULL);

break;

}

case (3):{

pthread\_create(&thread\_id,NULL,Mutex\_locks,NULL);

break;

}

case (4):{

pthread\_create(&thread\_id,NULL,dining\_phil,NULL);

break;

}

case (5):{

pthread\_create(&thread\_id,NULL,bded\_buff,NULL);

break;

}

}

pthread\_join(thread\_id,NULL);

}

return 0;

}

void printprograms(){

printf("\n\n\tPrograms Index :\n");

printf("1. Round Robin CPU Scheduling Algorithm\n");

printf("2. Banker's Algorithm\n");

printf("3. Mutex Locks leading to Deadlock\n");

printf("4. Dining Philosphers Problem using Monitors\n");

printf("5. Bounded Buffer problem using Semaphores\n\n\n\n");

}

void\* Round\_Robin(void\* ptr){

int i, limit, total = 0, x, counter = 0, time\_quantum;

int wait\_time = 0, turnaround\_time = 0, arrival\_time[10], burst\_time[10], temp[10];

float average\_wait\_time, average\_turnaround\_time;

printf("\t--Round Robin Algorithm--\n");

printf("\nEnter Total Number of Processes:\t");

scanf("%d", &limit);

x = limit;

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

{

printf("\nEnter Details of Process[%d]\n", i + 1);

printf("Arrival Time:\t");

scanf("%d", &arrival\_time[i]);

printf("Burst Time:\t");

scanf("%d", &burst\_time[i]);

temp[i] = burst\_time[i];

}

printf("\nEnter Time Quantum:\t");

scanf("%d", &time\_quantum);

printf("\nProcess ID\t\tBurst Time\t Turnaround Time\t Waiting Time\n");

for(total = 0, i = 0; x != 0;)

{

if(temp[i] <= time\_quantum && temp[i] > 0)

{

total = total + temp[i];

temp[i] = 0;

counter = 1;

}

else if(temp[i] > 0)

{

temp[i] = temp[i] - time\_quantum;

total = total + time\_quantum;

}

if(temp[i] == 0 && counter == 1)

{

x--;

printf("\nProcess[%d]\t\t%d\t\t %d\t\t\t %d", i + 1, burst\_time[i], total - arrival\_time[i], total - arrival\_time[i] - burst\_time[i]);

wait\_time = wait\_time + total - arrival\_time[i] - burst\_time[i];

turnaround\_time = turnaround\_time + total - arrival\_time[i];

counter = 0;

}

if(i == limit - 1)

{

i = 0;

}

else if(arrival\_time[i + 1] <= total)

{

i++;

}

else

{

i = 0;

}

}

average\_wait\_time = wait\_time \* 1.0 / limit;

average\_turnaround\_time = turnaround\_time \* 1.0 / limit;

printf("\n\nAverage Waiting Time:\t%f", average\_wait\_time);

printf("\nAvg Turnaround Time:\t%f\n", average\_turnaround\_time);

printf("End of Round Robin CPU Scheduling Algorithm\n");

}

void\* Bankers\_Algo(void\* ptr){

int process,resource,i,j,instanc,k=0,count1=0,count2=0; //count,k variables are taken for counting purpose

printf("\t----Banker's Algorithm----\n");

printf("Enter No. of Process: ");

scanf("%d",&process);

printf("\n"); //Entering No. of Processes

printf("Enter No. of Resources: ");

scanf("%d",&resource);

printf("\n"); //No. of Resources

int avail[resource],max[process][resource],allot[process][resource],need[process][resource],completed[process];

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

completed[i]=0; //Setting Flag for uncompleted Process

printf("Enter No. of Available Instances:\n");

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

{

printf("\t\t");

scanf("%d",&instanc);

avail[i]=instanc; // Storing Available instances

}

printf("Enter Maximum No. of instances of resources that a Process need: ");

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

{

printf("\n\t\tFor P[%d]:\n",i);

for(j=0;j<resource;j++)

{

printf("\t\t");

scanf("%d",&instanc);

max[i][j]=instanc;

}

}

printf("Enter no. of instances already allocated to process of a resource: ");

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

{

printf("\n\t\tFor P[%d]:\n",i);

for(j=0;j<resource;j++)

{

printf("\t\t");

scanf("%d",&instanc);

allot[i][j]=instanc;

need[i][j]=max[i][j]-allot[i][j]; //calculating Need of each process

}

}

printf("Safe Sequence is: ");

while(count1!=process)

{

count2=count1;

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

{

for(j=0;j<resource;j++)

{

if(need[i][j]<=avail[j])

{

k++;

}

}

if(k==resource && completed[i]==0 )

{

printf("P[%d]\t",i);

completed[i]=1;

for(j=0;j<resource;j++)

{

avail[j]=avail[j]+allot[i][j];

}

count1++;

}

k=0;

}

if(count1==count2)

{

printf("Stop ..After this.....Deadlock: ");

break;

}

}

printf("End of Banker's ALgorithm\n");

}

void\* Mutex\_locks(void\* ptr){

pthread\_t t1, t2;

printf("\t--DEADLOCK SITUATION BY MUTEX LOCKS--\n");

int iret1 = pthread\_create(&t1, NULL, print\_i, NULL);

int iret2 = pthread\_create(&t2, NULL, print\_j, NULL);

pthread\_join(t1,NULL);

pthread\_join(t2,NULL);

printf("Program Never Terminates.\n");

}

void\* dining\_phil(void\* ptr){

struct Philosopher\* philosophers;

int i, count = 5;

printf("Enter the number of philosophers: ");

scanf("%d",&count);

srand((unsigned int)time(NULL));

/\* if arguments is invalid \*/

if (count<2 || count>1000)

/\* replace with 5 \*/

count = 5;

/\* create array of structures for philosophers \*/

philosophers = (struct Philosopher\*) malloc(sizeof(struct Philosopher) \* count);

/\* create array of structures for forks \*/

forks = (struct Fork\*)malloc(sizeof(struct Fork) \* count);

/\* create global mutex in order to determinate all philophers eaten at least 1 time \*/

sem\_init(&global\_mutex,0,1);

/\* at start, no philosophers eaten \*/

NotEatenCount = count;

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

{

/\* initialzied mutex of each fork \*/

sem\_init(&forks[i].mutex,0,1);

/\* Each philosopher not yet eaten \*/

philosophers[i].eatenTimes = 0;

/\* Set number of philosophers (used for output) \*/

philosophers[i].number = i + 1;

/\* Set index of left fork \*/

philosophers[i].leftForkIndex = i;

/\* Set index of rigth fork

\* There indices will be used later

\*/

if (i+1==count)

philosophers[i].rightForkIndex = 0;

else

philosophers[i].rightForkIndex = i+1;

}

/\* run philosophers thread \*/

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

pthread\_create(&philosophers[i].thread\_id, NULL, philosopher\_thread, &philosophers[i]);

/\* check is finished \*/

while(!is\_finished())

usleep(100);

/\* finisgh all threads \*/

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

pthread\_join(philosophers[i].thread\_id, NULL);

/\* prints statistic \*/

printf("\nStatistics:\n");

for(i=0;i<count;i++){

printf("Philosopher %d eaten for %d times\n", philosophers[i].number, philosophers[i].eatenTimes);

}

/\* free all dynamically allocated memory \*/

free(forks);

free(philosophers);

}

void\* bded\_buff(void\* ptr){

int i;

pthread\_cond\_init(&(buffer.more), NULL);

pthread\_cond\_init(&(buffer.less), NULL);

pthread\_create(&tid[1], NULL, consumer, NULL);

pthread\_create(&tid[0], NULL, producer, NULL);

for ( i = 0; i < NUM\_THREADS; i++)

pthread\_join(tid[i], NULL);

printf("\nmain() reporting that all %d threads have terminated\n", i);

}

void \* producer(void \* parm)

{

char item[NUMITEMS]="IT'S A SMALL WORLD, AFTER ALL.";

int i;

printf("producer started.\n");

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

{ /\* produce an item, one character from item[] \*/

if (item[i] == '\0') break; /\* Quit if at end of string. \*/

pthread\_mutex\_lock(&(buffer.mutex));

if (buffer.occupied >= BSIZE) printf("producer waiting.\n");

while (buffer.occupied >= BSIZE)

pthread\_cond\_wait(&(buffer.less), &(buffer.mutex) );

printf("producer executing.\n");

buffer.buf[buffer.nextin++] = item[i];

buffer.nextin %= BSIZE;

buffer.occupied++;

/\* now: either buffer.occupied < BSIZE and buffer.nextin is the index

of the next empty slot in the buffer, or

buffer.occupied == BSIZE and buffer.nextin is the index of the

next (occupied) slot that will be emptied by a consumer

(such as buffer.nextin == buffer.nextout) \*/

pthread\_cond\_signal(&(buffer.more));

pthread\_mutex\_unlock(&(buffer.mutex));

}

printf("producer exiting.\n");

pthread\_exit(0);

}

void \* consumer(void \* parm)

{

char item;

int i;

printf("consumer started.\n");

for(i=0;i<NUMITEMS;i++){

pthread\_mutex\_lock(&(buffer.mutex) );

if (buffer.occupied <= 0) printf("consumer waiting.\n");

while(buffer.occupied <= 0)

pthread\_cond\_wait(&(buffer.more), &(buffer.mutex) );

printf("consumer executing.\n");

item = buffer.buf[buffer.nextout++];

printf("%c\n",item);

buffer.nextout %= BSIZE;

buffer.occupied--;

/\* now: either buffer.occupied > 0 and buffer.nextout is the index

of the next occupied slot in the buffer, or

buffer.occupied == 0 and buffer.nextout is the index of the next

(empty) slot that will be filled by a producer (such as

buffer.nextout == buffer.nextin) \*/

pthread\_cond\_signal(&(buffer.less));

pthread\_mutex\_unlock(&(buffer.mutex));

}

printf("consumer exiting.\n");

pthread\_exit(0);

}