LAB 6

20BCE1837

Aim: Banker Algorithm

Code:

```
#include  <stdio.h>
#include <stdlib.h>
int maxm = 100;

void display(int k[][maxm], int n, int p)
{
	int i, j;
	for (i = 0; i < n; i++)
	{
	for (j = 0; j < p; j++)
	{
	printf("%d\t", k[i][j]);
	}
	printf("\n");
	}
}

void Banker(int allocation[][maxm], int need[][maxm], int max[maxm][maxm], int
resource[1][maxm], int *n, int *m)
{
	int i, j;
	printf("Enter total number of processes: ");
	scanf("%d", n);
	printf("Enter total number of resources: ");
	scanf("%d", m);
	for (i = 0; i < *n; i++)
```

```
        {
        printf("\nProcess %d\n", (i+1));
        for (j = 0; j < *m; j++)
        {
        printf("Allocation for resource %d: ", (j+1));
        scanf("%d", &allocation[i][j]);
        printf("Maximum for resource %d: ", (j+1));
        scanf("%d", &max[i][j]);
        }
        }
        printf("\nAvailable resources:\n");
        for (i = 0; i < *m; i++)
        {
        printf("Resource %d: ", i + 1);
        scanf("%d", &resource[0][i]);
        }

        for (i = 0; i < *n; i++)
        for (j = 0; j < *m; j++)
        need[i][j] = max[i][j] - allocation[i][j];

        printf("\nAllocation Matrix:\n");
        display(allocation, *n, *m);
        printf("\nMaximum Requirement Matrix:\n");
        display(max, *n, *m);
        printf("\nNeed Matrix:\n");
        display(need, *n, *m);
}

int safety(int allocation[][maxm], int need[][maxm], int B[1][maxm], int n, int m,
int a[])
{

        int i, j, k, x = 0, f1 = 0, f2 = 0;
        int F[maxm], resource[1][maxm];
```

```
        for (i=0; i<n; i++)
        F[i] = 0;
        for (i=0; i<m; i++)
        resource[0][i] = B[0][i];

        for (k = 0; k < n; k++)
        {
        for (i = 0; i < n; i++)
        {
        if (F[i] == 0)
        {
                f2 = 0;
                for (j = 0; j < m; j++)
                {
                if (need[i][j] > resource[0][j])
                f2 = 1;
                }
                if (f2 == 0 && F[i] == 0)
                {
                for (j = 0; j < m; j++)
                resource[0][j] += allocation[i][j];
                F[i] = 1;
                f1++;
                a[x++] = i;
                }
        }
        }
        if (f1 == n)
        return 1;
        }
        return 0;
}

void request(int allocation[maxm][maxm], int need[maxm][maxm], int
B[maxm][maxm], int indx, int K)
```

```
{
	int rr[1][maxm];
	int i;
	printf("\nEnter additional request\n");
	for (i = 0; i < K; i++)
	{
	printf("Request for resource %d: ", (i+1));
	scanf("%d", &rr[0][i]);
	}

	for (i = 0; i < K; i++)
	if (rr[0][i] > need[indx][i])
	{
	printf("\nError encountered\n");
	exit(0);
	}

	for (i = 0; i < K; i++)
	if (rr[0][i] > B[0][i])
	{
	printf("\nResources unavailable\n");
	exit(0);
	}

	for (i = 0; i < K; i++)
	{
	B[0][i] -= rr[0][i];
	allocation[indx][i] += rr[0][i];
	need[indx][i] -= rr[0][i];
	}
}

int banker(int allocation[][maxm], int need[][maxm], int resource[1][maxm], int n,
int m)
{
```

```
	int j, i, a[maxm];
	j = safety(allocation, need, resource, n, m, a);
	if (j != 0)
	{
	printf("\nSafe Sequence:\n");
	for (i = 0; i < n; i++)
	printf("P%d ", a[i]);
	printf("\n");
	return 1;
	}
	else
	{
	printf("\n Deadlock has occured.\n");
	return 0;
	}
}

int main()
{
	int All[maxm][maxm], Max[maxm][maxm], Need[maxm][maxm],
resource[1][maxm];
	int n, m, indx, c, r;
	Banker(All, Need, Max, resource, &n, &m);
	r = banker(All, Need, resource, n, m);
	if (r!=0)
	{
	printf("\nEnter\n1: To make an additional request for any of the process\n0:
To exit\n");
	scanf("%d", &c);
	if (c==1)
	{
	printf("\nEnter process number: ");
	scanf("%d", &indx);
	request(All, Need, resource, indx - 1, m);
	r = banker(All, Need, resource, n, m);
```

```
        if (r == 0)
        {
                exit(0);
        }
        }
        }
        return 0;
}
```

Output:

```
Enter total number of processes: 3
Enter total number of resources: 3

Process 1
Allocation for resource 1: 4
Maximum for resource 1: 1
Allocation for resource 2: 5
Maximum for resource 2: 6
Allocation for resource 3: 8
Maximum for resource 3: 9

Process 2
Allocation for resource 1: 5
Maximum for resource 1: 4
Allocation for resource 2: 1
Maximum for resource 2: 2
Allocation for resource 3: 6
Maximum for resource 3:
5

Process 3
Allocation for resource 1: 8
Maximum for resource 1: 4
Allocation for resource 2: 5
Maximum for resource 2: 1
Allocation for resource 3: 2
Maximum for resource 3: 7
```

```
Available resources:
Resource 1: 6
Resource 2: 5
Resource 3: 1

Allocation Matrix:
4       5       8
5       1       6
8       5       2

Maximum Requirement Matrix:
1       6       9
4       2       5
4       1       7

Need Matrix:
-3      1       1
-1      1       -1
-4      -4      5

Safe Sequence:
P0 P1 P2

Enter
1: To make an additional request for any of the process
0: To exit
0
```

LAB 7

Aim: 1. Peterson Solution

2. Producer consumer problem using Semaphore

Peterson Solution

Code:

```
#include <stdio.h>
#include <pthread.h>

int flag[2];
int turn;

void lock(int i)
{
   flag[i] = 0;
   turn = 1-i;
   while((flag[1-i]) && (turn==(1-i)));
}

void unlock(int i)
{
   flag[i] = 0;
}

void* fn0(void *test){
   lock(0);
   printf("Process 0");
   unlock(0);
}
```

```
void* fn1(void *test){
   lock(1);
   printf("Process 1");
   unlock(1);
}

int main()
{
   pthread_t p1, p2;
   pthread_create(&p1, NULL, fn0, (void*)0);
   pthread_create(&p2, NULL, fn1, (void*)1);
   pthread_join(p1, NULL);
   pthread_join(p2, NULL);
   return 0;
}
```

OUTPUT:

Producer consumer problem using semaphore

Algorithm:

1.

Code:

```
#include <stdio.h>
int main()
{
        int bsize = 10, i = 0, o = 0, pr, cn, ch = 0;
        int bufr[bsize];
        while (ch!=3)
        {
        printf("Enter:\n1 to Produce\n2 to Consume\n3 to Exit\nHere: ");
        scanf("%d", &ch);
        switch (ch)
```

```
{
case 1:
        if ((i + 1) % bsize == o)
        printf("\nBuffer is Full");
        else
        {
        printf("\nEnter the value: ");
        scanf("%d", &pr);
        printf("\n");
        bufr[i] = pr;
        i = (i + 1) % bsize;
        }
        break;
case 2:
        if (i == o)
        printf("\nBuffer is Empty");
        else
        {
        cn = bufr[o];
        printf("\nThe consumed value is %d\n\n", cn);
```

```
                o = (o + 1) % bsize;
                }
                break;
        }
        }
        return 0;
}
```

OUTPUT:

```
Enter:
1 to Produce
2 to Consume
3 to Exit
Here: 1

Enter the value: 5

Enter:
1 to Produce
2 to Consume
3 to Exit
Here: 1

Enter the value: 4

Enter:
1 to Produce
2 to Consume
3 to Exit
Here: 1

Enter the value: 7

Enter:
1 to Produce
2 to Consume
3 to Exit
Here: 2

The consumed value is 5
```

LAB 8

LAB 8

Aim: 1. Peterson Solution

2. Producer Consumer problem with Semaphore

3. Producer Consumer problem without Semaphore

Peterson Solution

Code:

```
#include <stdio.h> int flag[2] = {0}; int turn;

void entryChecker(int pid){ flag[pid] = 1;
turn = pid;
int k = flag[1-pid];
while((k==1) && (turn == pid));
}

void exitSetter(int pid){ flag[pid] = 0;
}

void processDoer(int i){ entryChecker(i);
{
```

```
printf("Inside the critical section of the code\n");
}
exitSetter(i);
}


int main(){ processDoer(1); processDoer(3);
 return 0;
}
```

OUTPUT:

```
Inside the critical section of the code
Inside the critical section of the code
```

Producer Consumer problem with Semaphore Code:

```
#include <stdio.h>
int mutex = 0, empty, full = 0, in=0, out=0, n; int buffer[10];


void wait(int k){ while(k<0){
printf("\nCannot Add Item\n");
}
k--;
}


void signal(int k){ k++;
}
```

```
void producer(){ do{

wait(empty); wait(mutex); printf("Enter an item: "); scanf("%d",
&buffer[in]); in++;

signal(mutex); signal(full);

}while(in<n);

}


void consumer(){ do{

wait(full); wait(mutex);

printf("\nConsumed item: %d", buffer[out]); out++;


signal(mutex); signal(empty);

}while(out<n);

}


int main(){

printf("Enter the size: "); scanf("%d", &n);

empty = n; while(in<n){

producer();

}

while(in!=out){ consumer();
```

```
}

printf("\n"); return 0;

}
```

OUTPUT:

```
Enter the size: 5
Enter an item: 4
Enter an item: 1
Enter an item: 6
Enter an item: 2
Enter an item: 1

Consumed item: 4
Consumed item: 1
Consumed item: 6
Consumed item: 2
Consumed item: 1
```

Producer Consumer problem without Semaphore

Code:

```
#include <stdio.h>


int count = 0, in=0, out=0, n; const int bsize = 5;

int buffer[5];


void producer(){ while(1){

while(count == bsize); printf("Enter value in buffer: "); scanf("%d", &buffer[in]);

in++;

in = in%bsize; count++;

}
```

```
}

void consumer(){ while(1){
while(count == 0);
printf("Consumed value %d", buffer[out]); out++;
out = out%bsize; count--;
}
}

int main(){
printf("Enter the size: "); scanf("%d", &n); while(in<n){
producer();
}

while(in!=out){ consumer();
}
printf("\n"); return 0;
}
```

OUTPUT:

```
Enter the size: 5
Enter value in buffer: 2
Enter value in buffer: 4
Enter value in buffer: 6
Enter value in buffer: 1
Enter value in buffer: 2
```

LAB 9

LAB 9

Aim: 1. Dining Philosophers Problem

2. Reader Writer Problem

Dining Philosophers Problem

Code:

```
#include <stdio.h>
#define n 5
int cp = 0, i;

struct fork
{
        int taken;
} forkTaken[n];

struct philosp
{
        int l;
        int r;
} pstatus[n];

void goForDinner(int pID)
{
        if (pstatus[pID].l == 10 && pstatus[pID].r == 10){
        printf("Philosopher %d completed his dinner\n", pID + 1);
        }

        else if (pstatus[pID].l == 1 && pstatus[pID].r == 1){
        printf("Philosopher %d completed his dinner\n", pID + 1);
        pstatus[pID].l = pstatus[pID].r = 10;
```

```
        int otherFork = pID - 1;
        if (otherFork == -1){
        otherFork = (n - 1);
        }

        forkTaken[pID].taken = forkTaken[otherFork].taken = 0;
        printf("Philosopher %d released fork %d and fork %d\n", pID + 1, pID + 1,
otherFork + 1);
        cp++;
        }
        else if (pstatus[pID].l == 1 && pstatus[pID].r == 0){
        if (pID == (n - 1)){
        if (forkTaken[pID].taken == 0){
                forkTaken[pID].taken = pstatus[pID].r = 1;
                printf("Fork %d taken by philosopher %d\n", pID + 1, pID + 1);
        }
        else{
                printf("Philosopher %d is waiting for fork %d\n", pID + 1, pID + 1);
        }
        }
        else{

        int dpID = pID;
        pID -= 1;

        if (pID == -1)
                pID = (n - 1);

        if (forkTaken[pID].taken == 0)
        {
                forkTaken[pID].taken = pstatus[dpID].r = 1;
                printf("Fork %d taken by Philosopher %d\n", pID + 1, dpID + 1);
        }
        else
        {
                printf("Philosopher %d is waiting for fork %d\n", dpID + 1, pID + 1);
```

```
	}
	}
	}
	else if (pstatus[pID].l == 0){
	if (pID == (n - 1)){
	if (forkTaken[pID - 1].taken == 0){
		forkTaken[pID - 1].taken = pstatus[pID].l = 1;
		printf("Fork %d taken by Philosopher %d\n", pID, pID + 1);
	}
	else{
		printf("Philosopher %d is waiting for fork %d\n", pID + 1, pID);
	}
	}
	else{

	if (forkTaken[pID].taken == 0){
		forkTaken[pID].taken = pstatus[pID].l = 1;
		printf("Fork %d taken by Philosopher %d\n", pID + 1, pID + 1);
	}
	else{
		printf("Philosopher %d is waiting for fork %d\n", pID + 1, pID + 1);
	}
	}
	}
}


int main()
{
	for (i = 0; i < n; i++){
	forkTaken[i].taken = pstatus[i].l = pstatus[i].r = 0;
	}

	while (cp < n){
	for (i = 0; i < n; i++){
	goForDinner(i);
	}
```

```c
        printf("\nNumber of philosophers who completed dinner: %d\n", cp);
        }
        return 0;
}
```

OUTPUT:

```
Fork 1 taken by Philosopher 1
Fork 2 taken by Philosopher 2
Fork 3 taken by Philosopher 3
Fork 4 taken by Philosopher 4
Philosopher 5 is waiting for fork 4

Number of philosophers who completed dinner: 0
Fork 5 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
Philosopher 5 is waiting for fork 4

Number of philosophers who completed dinner: 0
Philosopher 1 completed his dinner
Philosopher 1 released fork 1 and fork 5
Fork 1 taken by Philosopher 2
Philosopher 3 is waiting for fork 2
Philosopher 4 is waiting for fork 3
Philosopher 5 is waiting for fork 4

Number of philosophers who completed dinner: 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
Philosopher 5 is waiting for fork 4

Number of philosophers who completed dinner: 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
Philosopher 5 is waiting for fork 4
```