**Programms in C:**

**23. Compute the addition of array elements and calculate average using shared memory. Average will be calculated using barrier. Array is declared as shared resource.**

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

void main()

{ int i, n;

int \*a, \*b, \*c;

clrscr();

printf("How many Elements in each array...\n");

scanf("%d", &n);

a = (int \*)malloc(n \* sizeof(int));

b = (int \*)malloc(n \* sizeof(int));

c = (int \*)malloc(n \* sizeof(int));

printf("Enter Elements of First List\n");

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

{

scanf("%d", a + i);

}

printf("Enter Elements of Second List\n");

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

{

scanf("%d", b + i);

}

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

{

\*(c + i) = \*(a + i) + \*(b + i);

}

printf("Resultant List is\n");

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

{

printf("%d\n", \*(c + i));

}

getch();

}

**24. Producer-Consumer problem:**

1. **Implement producer, consumer problem using bounded buffer where array is shared. When producer will put the element in the array, consumer will wait until producer will release the array and same operation is applicable for consumer. Implement mutual exclusion using semaphore. When array is full, producer will wait until consumer will consume at least one element. When buffer is empty, consumer will wait until producer will produce at least one element.**

1. **Implement producer, consumer problem using un-bounded buffer where array is shared. When producer will put the element in the array, consumer will wait until producer will release the array and same operation is applicable for consumer. Implement mutual conclusion using semaphore. When buffer is empty, consumer will wait until producer will produce at least one element.**

#include<stdio.h>

#include<stdlib.h>

int mutex=1,full=0,empty=3,x=0;

void producer();

void consumer();

int wait(int);

int signal(int);

int main()

{

int n;

clrscr();

printf("\n 1.Producer \n 2.Consumer \n 3.Exit");

while(1)

{

printf("\n Enter your choice:");

scanf("%d",&n);

switch(n)

{

case 1:

if((mutex==1)&&(empty!=0))

producer();

else

printf("Buffer is full\n");

break;

case 2:

if((mutex==1)&&(full!=0))

consumer();

else

printf("Buffer is empty\n");

break;

case 3:

exit(0);

break;

}

}

}

int wait(int s)

{

return (--s);

}

int signal(int s)

{

return(++s);

}

void producer()

{

mutex=wait(mutex);

full=signal(full);

empty=wait(empty);

x++;

printf("\n Producer produces the item %d:",x);

mutex=signal(mutex);

}

void consumer()

{

mutex=wait(mutex);

full=wait(full);

empty=signal(empty);

printf("\n Consumer consumes item %d:",x);

x--;

mutex=signal(mutex);

}

**25. Reader-Writer problem:**

**More than one Reader can read from a file, but, when one writer will write, no other writer and readers can operate on the file. When one writer will write, whole file is locked by that writer. Implement these features using mutual exclusion.**

Code :

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<semaphore.h>

int data=0,rdcnt=0; sem\_t mutex,writeblock;

void \* reader(void \* no)

{

printf("\n\tReader %d is executing ",(int)no); sem\_wait(&mutex);

printf("\n\tWait to mutex by %d reader",(int)no);

rdcnt++; if(rdcnt==1)

{

sem\_wait(&writeblock);

printf("\n\tWait to writerblock by %d reader",(int)no);

}

//sleep(2);

printf("\n\t\*\*\*Reader %d read data = %d ",(int)no,data); if(rdcnt==1)

{

sem\_post(&writeblock);

printf("\n\tSignal to writerblock by %d reader",(int)no);

}

sem\_post(&mutex);

printf("\n\tSignal to mutex by %d reader\n",(int)no);

}

void \* writer(void \* no)

{

printf("\n\tWriter %d is executing ",(int)no);

sem\_wait(&writeblock);

printf("\n\tWait to writerblock by %d writer",(int)no);

//sleep(2); data+=5;

printf("\n\t\*\*\*Writer %d write data = %d ",(int)no,data); sem\_post(&writeblock);

printf("\n\tSignal to writer by %d writer\n",(int)no);

}

int main()

{

int no,i,ir=0,iw=0,ch; sem\_init(&mutex,0,1); sem\_init(&writeblock,0,1);

printf("\nEnter no of readers and writers to create : "); scanf("%d",&no);

pthread\_t r[no],w[no];

/\*for(i=0;i<no;i++)

{

pthread\_create(&r[i],NULL,reader,(void \*)i); pthread\_create(&w[i],NULL,writer,(void \*)i); }\*/

/\*for(i=0;i<no;i++)

{

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

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

}\*/ do

{

printf("\n\t1.Reader\n\t2.Writer\n\t3.terminate\n\tYour choice : "); scanf("%d",&ch);

switch(ch)

{

case 1: pthread\_create(&r[ir],NULL,reader,(void \*)ir);

pthread\_join(r[ir++],NULL); break;

case 2: pthread\_create(&w[iw],NULL,writer,(void \*)iw);

pthread\_join(w[iw++],NULL);

break;

}

}while(ch!=3);

sem\_destroy(&mutex);

sem\_destroy(&writeblock);

}

**26. Dinning philosophers’ problem”**

1. **There are five philosophers sitting in rounded dining table and five fork are used for eating spaghetti .When one philosopher is wishing to eat, he should check if both forks are free or not, otherwise he should wait for getting two forks.**
2. **There is a room. Within room, only four philosophers are only allowed. After entering, they will seat on dining table. If forks are available, they will finish their dinner and then come out from room. Entrance in room and getting fork are controlled by semaphore.**

#include<stdio.h>

#include<semaphore.h>

#include<pthread.h>

#define N 5

#define THINKING 0

#define HUNGRY 1

#define EATING 2

#define LEFT (ph\_num+4)%N

#define RIGHT (ph\_num+1)%N

sem\_t mutex;

sem\_t S[N];

void \* philospher(void \*num);

void take\_fork(int);

void put\_fork(int);

void test(int);

int state[N];

int phil\_num[N]={0,1,2,3,4};

int main()

{

int i;

pthread\_t thread\_id[N];

sem\_init(&mutex,0,1);

clrscr();

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

sem\_init(&S[i],0,0);

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

{

pthread\_create(&thread\_id[i],NULL,philospher,&phil\_num[i]);

printf("Philosopher %d is thinking\n",i+1);

}

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

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

}

void \*philospher(void \*num)

{

while(1)

{

int \*i = num;

sleep(1);

take\_fork(\*i);

sleep(0);

put\_fork(\*i);

}

}

void take\_fork(int ph\_num)

{

sem\_wait(&mutex);

state[ph\_num] = HUNGRY;

printf("Philosopher %d is Hungry\n",ph\_num+1);

test(ph\_num);

sem\_post(&mutex);

sem\_wait(&S[ph\_num]);

sleep(1);

}

void test(int ph\_num)

{

if (state[ph\_num] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)

{

state[ph\_num] = EATING;

sleep(2);

printf("Philosopher %d takes fork %d and %d\n",ph\_num+1,LEFT+1,ph\_num+1); printf("Philosopher %d is Eating\n",ph\_num+1); sem\_post(&S[ph\_num]);

}

}

void put\_fork(int ph\_num)

{

sem\_wait(&mutex);

state[ph\_num] = THINKING;

printf("Philosopher %d putting fork %d and %d down\n",ph\_num+1,LEFT+1,ph\_num+1); printf("Philosopher %d is thinking\n",ph\_num+1); test(LEFT); test(RIGHT); sem\_post(&mutex);

}