Synchronization Applications -B.Ajay
21011102020

Aim:

To write a program to simulate

1.Producer Consumer Problem

2.Dining Philosopher's Problem

Using semaphores

Algorithm:

1.**Producer Consumer Problem**:

1.Declare a Producer ,consumer function that reads/write to the common buffer depending on BUFFER_LEN

2.Initialize buffer sem_t empty,full,buf indices,and item produced as global variable

3.In the produce function we produce first and then we wait and lock the mutex to modify the global buffer and its index

4.In the consumer function we wait if the buffer is empty and apply lock on mutex as we consume the item and modify index

5.Then we apply unlock on both functions

6.In the main function create producer,consumer variables as pthread variables

And initialize them using pthread_create ();

7.Once all Job has been executed use pthread_join to finish off remaining task by thread and free mutex memory

2.**Dining Philosopher's Problem**

1. The idea behind this problem is for a person to eat he needs 2 chopsticks which can only be availed only when the neighbour is thinking .This problem has to be solved without any of the philosopher starve/ create any deadlock .

2.In the main function we initialize chopstick mutexes and initialize philosopher threads and once the function finishes execution we use pthread_join,pthread_mutex_destroy

3.We define another function eatPhil that prints philosopher k is thinking thernm apply lock on left and right chopstick and eat then unlock and release both.Then we print who finished eating .

Description:

1.pthread_create - create a new thread

Syntax:

```
#include <pthread.h>
 int pthread_create(pthread_t *restrict thread,const pthread_attr_t *restrict attr,
                              void *(*start_routine)(void *),void *restrict arg);
```

Description:

The pthread_create() function starts a new thread in the calling process. The new thread starts execution by invoking start_routine(); arg is passed as the sole argument of start_routine().

2.pthread_join:

Syntax:

pthread_join - join with a terminated thread

```
#include <pthread.h>
 int pthread_join(pthread_t thread, void **retval);
```

Description:

The pthread_join() function waits for the thread specified by thread to terminate. If that thread has already terminated, then pthread_join() returns immediately. The thread specified by thread must be joinable.

3.semaphore.h:

```
#include <semaphore.h>
```

Description:

The <semaphore.h> header defines the sem_t type, used in performing semaphore operations. The semaphore may be implemented using a file descriptor, in which case applications are able to open up at least a total of OPEN_MAX files and semaphores.

4.sem_init():

Syntax:

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Description:

The sem_init() function is used to initialise the unnamed semaphore referred to by sem. The value of the initialised semaphore is value. Following a successful call to sem_init(), the semaphore may be used in subsequent calls to sem_wait(), sem_trywait(), sem_post(), and sem_destroy(). This semaphore remains usable until the semaphore is destroyed.

5.sem_post:

Syntax:

```
int sem_post(sem_t *sem);
```

Description

The sem_post() function unlocks the semaphore referenced by sem by performing a semaphore unlock operation on that semaphore.

6.sem_wait(sem t*sem)

Syntax:

```
int sem_wait(sem_t *sem)
```

Description:

The sem_wait() function locks the semaphore referenced by sem by performing a semaphore lock operation on that semaphore. If the semaphore value is currently zero, then the calling thread will not return from the call to sem_wait() until it either locks the semaphore or the call is interrupted by a signal. The sem_trywait() function locks the semaphore referenced by sem only if the semaphore is currently not locked; that is, if the semaphore value is currently positive. Otherwise, it does not lock the semaphore.

Code:

Procons.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
int item=0;
#define buflen 1
int buf[buflen];
int in=0,out=0;
sem_t empty,full;
pthread_mutex_t mutex;
void produce(void *param){
	do{
	item++;
	sem_wait(&empty);
	pthread_mutex_lock(&mutex);
	buf[in]=item;

	printf("Producer Produced : %d\n",buf[in]);
	in=(in++)%buflen;
	pthread_mutex_unlock(&mutex);
	sem_post(&full);
//	sleep(1);
	}while(1);



}

void consume(void *param){
	do{
	sem_wait(&full);
	pthread_mutex_lock(&mutex);

	printf("Consumer Consumed :%d\n",buf[out]);
	out=(out++)%buflen;
	pthread_mutex_unlock(&mutex);
	sem_post(&empty);
//	sleep(1);
	}while(1);

}
```

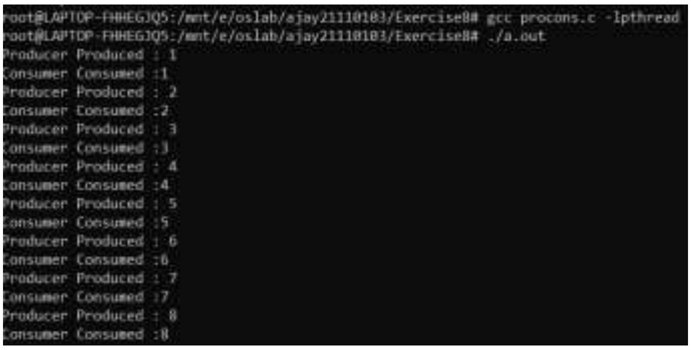
```
int main(){
        pthread_t producer,consumer;
        sem_init (&empty,0,buflen);
        sem_init(&full,0,0);
        pthread_mutex_init(&mutex,NULL);
        pthread_create(&producer,NULL,(void*)produce,NULL);
        pthread_create(&consumer,NULL,(void*)consume,NULL);



        pthread_join(producer,NULL);
        pthread_join(consumer,NULL);
        pthread_mutex_destroy(&mutex);
        sem_destroy(&empty);
        sem_destroy(&full);

}
```

Output:



```
root@LAPTOP-FHHEGJQ5:/mnt/e/oslab/ajay21110103/Exercise8# gcc procons.c -lpthread
root@LAPTOP-FHHEGJQ5:/mnt/e/oslab/ajay21110103/Exercise8# ./a.out
Producer Produced : 1
Consumer Consumed :1
Producer Produced : 2
Consumer Consumed :2
Producer Produced : 3
Consumer Consumed :3
Producer Produced : 4
Consumer Consumed :4
Producer Produced : 5
Consumer Consumed :5
Producer Produced : 6
Consumer Consumed :6
Producer Produced : 7
Consumer Consumed :7
Producer Produced : 8
Consumer Consumed :8
```

DiningPhilosopher.c:

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#define no_philosopher 5
#define no_chopstick 5
```

```
pthread_t philosopher[no_philosopher];
pthread_mutex_t chopstick[no_chopstick];
void eatPhil(int k){
    printf("Philosopher %d ->Thinking\n",k );
    pthread_mutex_lock(&chopstick[k]);
    pthread_mutex_lock(&chopstick[(k+1)%no_philosopher]);
    printf("Philosopher %d -> Eating\n",k);
    sleep(1);
    pthread_mutex_unlock(&chopstick[k]);
    pthread_mutex_unlock(&chopstick[(k+1)%no_philosopher]);
    printf("Philosopher %d -> Ate\n",k);
}




int  main()
{

    for(int i=1;i<=no_chopstick;i++){
        pthread_mutex_init(&chopstick[i],NULL);
    }
    for(int i=1;i<=no_philosopher;i++){
        pthread_create(&philosopher[i],NULL,(void*)eatPhil,(int* )i);
    }
    for(int i=1;i<=no_philosopher;i++){
        pthread_join(philosopher[i],NULL);

    }
    for(int i=1;i<=no_chopstick;i++){
```

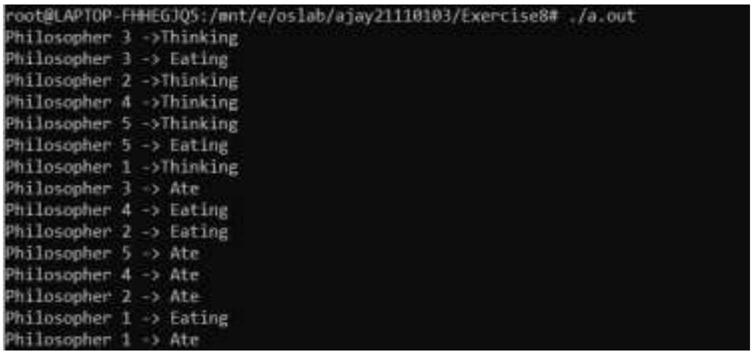
```
                pthread_mutex_destroy(&chopstick[i]);

        }


        return 0;

}
```

Output:



```
root@LAPTOP-FHHEGJQ5:/mnt/e/oslab/ajay21110103/Exercise8# ./a.out
Philosopher 3 ->Thinking
Philosopher 3 -> Eating
Philosopher 2 ->Thinking
Philosopher 4 ->Thinking
Philosopher 5 ->Thinking
Philosopher 5 -> Eating
Philosopher 1 ->Thinking
Philosopher 3 -> Ate
Philosopher 4 -> Eating
Philosopher 2 -> Eating
Philosopher 5 -> Ate
Philosopher 4 -> Ate
Philosopher 2 -> Ate
Philosopher 1 -> Eating
Philosopher 1 -> Ate
```

Result:

Thus the above programs were simulated in C using semaphores and pthreads