PICT, PUNE

Name : Shubham Chemate
Roll No. : 31118
Batch : K-1

Laboratory Practices - I

Title: Process Synchronization

Problem statement:

Write a program to solve classical problems of synchronization using mutex & semaphore.

Software & Hardware Requirements:

Software: ECLIPSE IDE (version 20.03)
WINDOW-10 (64-bit OS)
Java SE 14

Hardware: Intel core-i5 8th gen (6285U) 4-core CPU, 8GB SSD & 512 GB HDD.

Learning objectives:

1) Should get to know about semaphore & mutex
2) Learn how synchronization is done in operating systems.

Learning Outcomes:

1) Student will be able to use of threads, semaphores & mutex while programming
2) student will learn synchronization in operating system.

Theory concepts:

A) Process Synchronization:

- A critical section execution is handled by a semaphore.

PICT, PUNE

- A semaphore is simply a variable that stores an integer value. This integer is accessed by two operations wait() & signal().
- When a process enters the critical section, P(S) is invoked & the semaphore S is set to 1.
- After the process exits the critical section S is re-initialized to 0.

Example shown below:

```
// some code
P(S)
// critical section
// exit from critical section
V(S)
// remaining code
```

- There are two types of semaphores:

1) Binary Semaphore:
   They can be either 0 or 1. They are known as mutex locks, as locks can provide mutual exclusion.
   All the processes can share the same mutex semaphore that is initialized to 1.

2) Counting Semaphore:
   They can have any value and are not restricted over certain domain.
   They can be used to control access to the resource that has a limitation on the number of instances of the resource.

PICT, PUNE

B) Classical problems of synchronization

1) Producer - consumer problem
   i) Solution to this problem involves creating two semaphores "full" & "empty" to keep track of the current number of full & empty buffers respectively.
   ii) Producer produce a product & consumers consumes the product but both use one of the containers each time.

2) Dining philosophers problem.
   i) This problem states that k-philosophers seated around a circular table with one chopstick between each pair of philosophers.
   ii) There is one chopstick between each philosopher so he may eat by any one of them.
   iii) One chopstick may be picked by two philosophers but on philosopher can pick only one chopstick at a time.
   iv) This problem involves the allocation of limited resources to a group of processes in a deadlock-free & starvation free manner.

3) Reader-& writer Problem:
   i) Suppose that a database is to be shared among several concurrent processes.
   ii) Some of these processes may want only

to be read the database whereas others may want only to be read/write the database.

ii) We distinguish between these two processes by referring to the former as readers & the latter one as writers.

iv) Precisely in OS, we call this situation as the readers writers problem.

4) Sleeping barber problem:

i) Barbar shop with one barbar, one barber chair & N chairs to waan in.

ii) When no customers, the barber goes to sleep in barber chair &

iii) When customer comes in, when barber is cutting hair new customers take empty seats to wait @ leave if no vacancy.

Syntax:

1) Wait function

int sem_wait(sem_t *sem)

ii) Signal function:

int sem_post(sem_t *sem)

iii) Semaphore initialization:

sem_init (sem_t *sem, int pshared, unsigned int val);

iv) Destroy Semaphore

sem_destroy(sem_t *sem);

PICT, PUNE

Algorithm:

1) Define number of philosophers
2) Declare semaphores for all chopsticks & initialize it to '1'
3) Declare objects of philosopher class & pass philosophers ID, left chopsticks' ID & right chopstick ID to it.
4) Start the all threads.

Algorithm for philosophers:

1) Define think method.
2) Define method for picking up left chopstick.
3) Define method for picking up right chopstick.
4) Define eat method.
5) While "true" call all above mentioned methods.

Applications: to solve classical problems of synchronization.

Testcases

Separate file with code & output is attached.

Conclusion:

We have sucessfully solved the classical problem of synchronization using mutex & semaphore