# Ex 4:Title of the Exercise: Mutual Exclusion using semaphore and monitor

Date: –19/09/2022

Aim: –

Multithreading using Mutual exclusion using semaphore and monitor

Procedure: –

Thread synchronization

Thread synchronization is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as a critical section. Processes’ access to critical section is controlled by using synchronization techniques. When one thread starts executing thecritical section (a serialized segment of the program) the other thread should wait until the first thread finishes. If proper synchronization techniques are not applied, it may cause a race condition where the values of variables may be unpredictable and vary depending on the timings of context switches of the processes or threads.

Mutex: --

1. A Mutex is a lock that we set before using a shared resource and release after using it.
2. When the lock is set, no other thread can access the locked region of code.
3. So, we see that even if thread 2 is scheduled while thread 1 was not done accessing the shared resource and the code is locked by thread 1 using mutexes then thread 2 cannot even access that region of code.
4. So, this ensures synchronized access of shared resources in the code.

Code: –

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

pthread\_t tid[2];

int counter;

pthread\_mutex\_t lock;

void\* threadFunction(void\* arg)

{

pthread\_mutex\_lock(&lock);

unsigned long i = 0;

counter += 1;

printf("\n Job %d has started\n", counter);

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

;

printf("\n Job %d has finished\n", counter);

pthread\_mutex\_unlock(&lock);

return NULL;

}

int main(void)

{

int i = 0;

int err;

while(i<2)

{

err = pthread\_create(&(tid[i]), NULL, &doSomeThing, NULL);

if (err != 0)

printf("\ncan't create thread :[%s]", strerror(err));

i++;

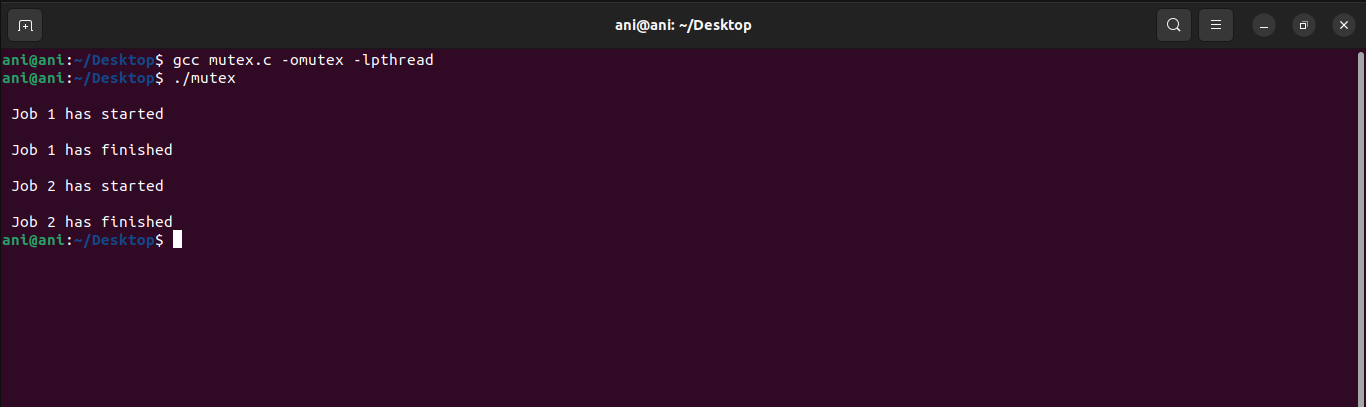
}

pthread\_join(tid[0], NULL);

pthread\_join(tid[1], NULL);

return 0;

}Output: --



Mutual Exclusion using semaphore

A semaphore is an integer variable that allows many processes in a parallel system to manage access to a common resource like a multitasking OS. It is an integer variable (S), and it is initialized with the number of resources in the system. The wait() and signal() methods are the only methods that may modify the semaphore (S) value. When one process modifies the semaphore value, other processes can't modify the semaphore value simultaneously.

Semaphore*s* are an abstract entity provided by an operating system (not the hardware). Semaphores:

* are named by a unique semaphore id
* consist of a tuple (id, count, queue), where count is an integer and queue are a list of processes.
  + a non-negative count always means that the queue is empty
  + a count of negative n indicates that the queue contains n waiting processes.
  + a count of positive n indicates that n resources are available and n requests can be granted without delay.
* sem = semcreate(val) -- creates a semaphore with the given initial value
* semdelete(sem) -- delete a semaphore
* wait(sem) -- decrement the semaphore count. if negative, suspend the process and place in queue. (Also referred to as P())
* signal(sem) -- increment the semaphore count, allow the first process in the queue to continue. (Also referred to as V())

Code: --

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

sem\_t mutex;

void\* thread(void\* arg)

{

//wait

sem\_wait(&mutex);

printf("\nEntered..\n");

//critical section

sleep(4);

//signal

printf("\nJust Exiting...\n");

sem\_post(&mutex);

}

int main()

{

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

pthread\_t t1,t2;

pthread\_create(&t1,NULL,thread,NULL);

sleep(2);

pthread\_create(&t2,NULL,thread,NULL);

pthread\_join(t1,NULL);

printf("The Thread ID is of thread 1 is %ld\n",t1);

pthread\_join(t2,NULL);

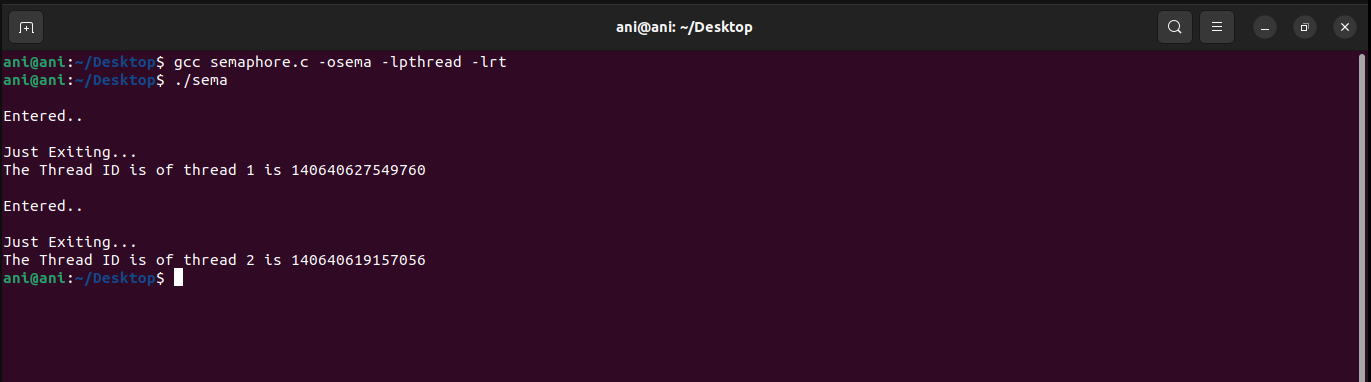
printf("The Thread ID is of thread 2 is %ld\n",t2);

sem\_destroy(&mutex);

return 0;

}

Output: --



Mutual Exclusion using Monitor

It is a synchronization technique that enables threads to mutual exclusion and the wait() for a given condition to become true. It is an abstract data type. It has a shared variable and a collection of procedures executing on the shared variable. A process may not directly access the shared data variables, and procedures are required to allow several processes to access the shared data variables simultaneously.

At any particular time, only one process may be active in a monitor. Other processes that require access to the shared variables must queue and are only granted access after the previous process releases the shared variables.

The monitor is made up of four primary parts:

* Initialization: The code for initialization is included in the package, and we just need it once when creating the monitors.
* Private Data: It is a feature of the monitor in an operating system to make the data private. It holds all of the monitor's secret data, which includes private functions that may only be utilized within the monitor. As a result, private fields and functions are not visible outside of the monitor.
* Monitor Procedure: Procedures or functions that can be invoked from outside of the monitor are known as monitor procedures.
* Monitor Entry Queue: Another important component of the monitor is the Monitor Entry Queue. It contains all of the threads, which are commonly referred to as procedures only.

Code: --

import threading

import time

class testAndSet():

#lock variable used in test and set is defined here

Lock = 0

def test(self,\*args):

if self.Lock == 0:

# The critical section goes here...

self.criticalsection(args[0])

else:

#while other process is executing current process is waiting

while self.Lock == 1:

print(f"Process {args[0]} waiting")

def criticalsection(self,i):

self.Lock = 1

print(f'Process {i} Entered Critical Section.\nPerform operation on shared resource')

#exit section

self.Lock = 0

print(f'Process {i} exited Critical Section')

def main(self):

t1 = threading.Thread(target = self.test, args = (0,))

t1.start()

t2 = threading.Thread(target = self.test, args = (1,))

t2.start()

t3 = threading.Thread(target = self.test, args = (2,))

t3.start()

t4 = threading.Thread(target = self.test, args = (3,))

t4.start()

t5 = threading.Thread(target = self.test, args = (4,))

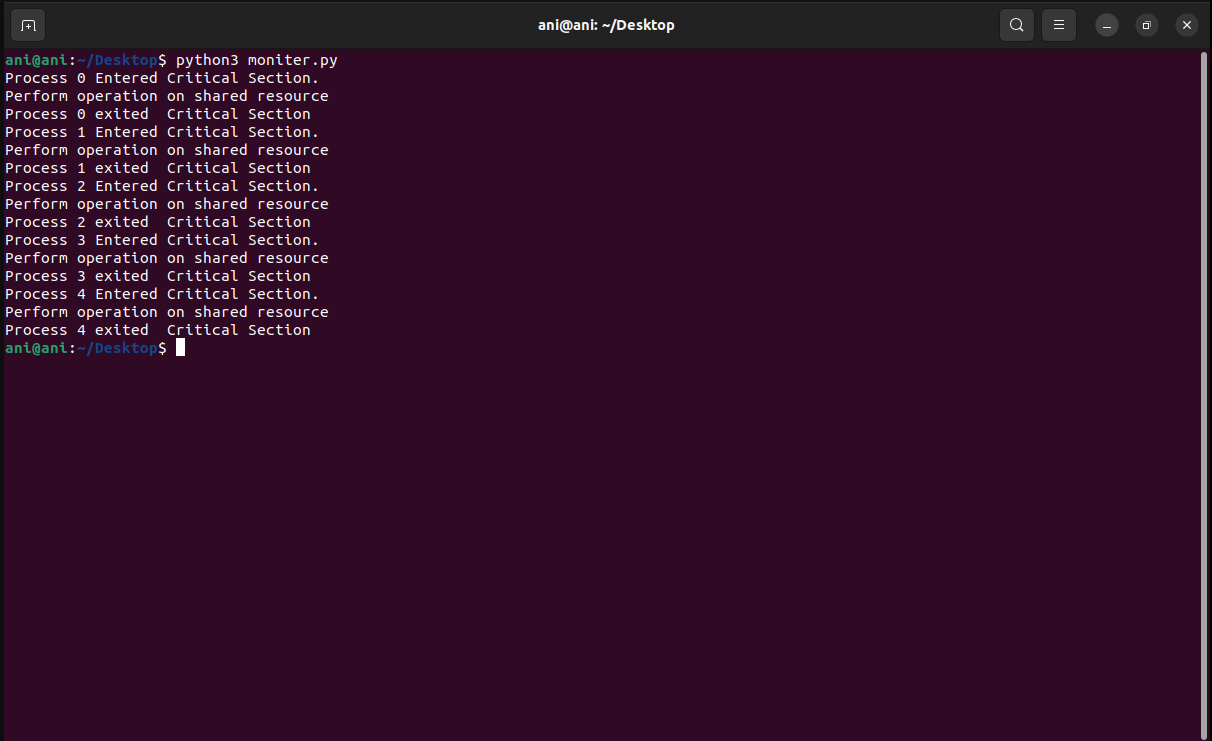
t5.start()

if \_\_name\_\_=="\_\_main\_\_":

t = testAndSet()

t.main()

Output:--



Result: -

Successfully executed Mutual Exclusion using semaphore and monitor