Program to simulate Race Condition

Leave a Comment / Programs / By Baljit Singh Saini

A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place is called a ***race condition***. In this post we will write a Program using threads to simulate race condition.

To understand the below program, as a recommendation kindly go through the concept of how race condition occurs by visiting this link

/* Program to show the race condition.
Program to create two threads: one to increment the value of a shared variable and second to decrement the value of shared variable. Both the threads are executed, so the final value of shared variable should be same as its initial value. But due to race condition it would not be same. */

```
#include<pthread.h>
#include<stdio.h>
#include<unistd.h>
void *fun1();
void *fun2();
int shared=1; //shared variable
int main()
 {
 pthread_t thread1, thread2;
 pthread_create(&thread1, NULL, fun1, NULL);
 pthread_create(&thread2, NULL, fun2, NULL);
 pthread_join(thread1, NULL);
 pthread_join(thread2,NULL);
 printf("Final value of shared is %d\n",shared); //prints the last updated value of shared variable
 }



void *fun1()
{
    int x;
    x=shared;//thread one reads value of shared variable
    printf("Thread1 reads the value of shared variable as %d\n",x);
    x++;  //thread one increments its value
    printf("Local updation by Thread1: %d\n",x);
```

```
    printf("Value of shared variable updated by Thread1 is: %d\n",shared);
}



void *fun2()
 {
     int y;
     y=shared;//thread two reads value of shared variable
     printf("Thread2 reads the value as %d\n",y);
     y--;  //thread two increments its value
     printf("Local updation by Thread2: %d\n",y);
     sleep(1); //thread two is preempted by thread 1
     shared=y; //thread one updates the value of shared variable
     printf("Value of shared variable updated by Thread2 is: %d\n",shared);
 }
```

Note: the final value of shared variable should have been 1 but it will be either 2 or 0 depending upon which thread executes first. This happened because the two processes were not synchronized. When one thread was modifying the value of shared variable the other thread must not have read its value for modification. This can be achieved using locks or semaphores.

Output:

```
baljit@baljit:~/cse325$ gcc racecondition.c -lpthread
baljit@baljit:~/cse325$ ./a.out
Thread1 reads the value of shared variable as 1
Local updation by Thread1: 2
Thread2 reads the value as 1
Local updation by Thread2: 0
Value of shared variable updated by Thread2 is: 0
Value of shared variable updated by Thread1 is: 2
Final value of shared is 2
```

Race condition

How it Works?

Thread1 reads the value of shared variable as 1 and then increments it to 2. Now, before it could update the shared variable, Thread1 is preempted (using sleep()) by Thread2 which reads the unstable value of shared variable as 1. Thread2 then decrements it to 0. Then, both the threads prints the updated value of shared variable. Since, the final updation is done by Thread1, so the final value comes out to be 2

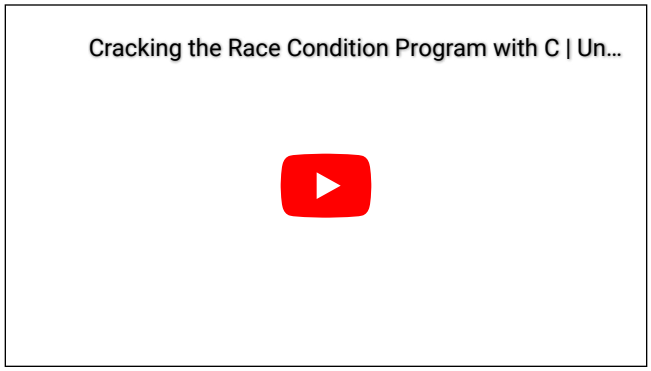
Viva Questions on Program to simulate Race Condition

Q1. What is race condition?

Q2. Which section in the program can be called the critical section?

Q3. What should be the correct value of the *shared* variable in the above program and why?

Video Link



Relevant Programs

- Program for Thread Creation in Linux
- Process Synchronization using mutex locks
- Process Synchronization using semaphores

← Previous Post

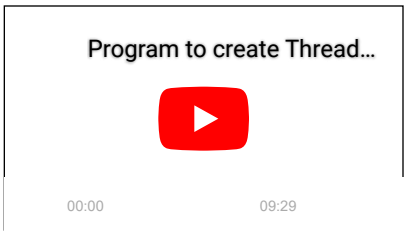
Next Post →

Leave a Comment

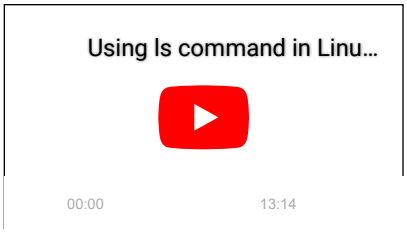
You must be logged in to post a comment.

Search ...

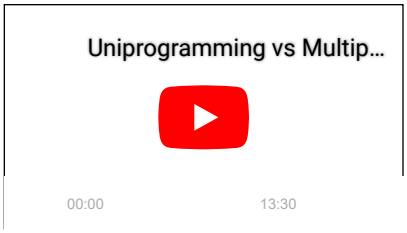
OS Lab



Linux Essentials Series



OS Theory



Categories