**Student Name :** Badal Sherpa

**Student ID :** 11813239

**Email Address :** sangay5chan@gmail.com

**GitHub Link :** <https://github.com/BadalSherpa/Operating-system>

**Project No. 15**

**Code :**

#include<pthread.h>

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

void \*fun1();

void \*fun2();

int resource=1; //shared resource

void \*fun1()

{

int x;

x=resource;

x++; // this thread one increments its value

sleep(1); //this thread one is preempted by thread 2

resource=x; //thread one updates the value of resource variable

}

void \*fun2()

{

int y;

y=resource;//this thread two reads value of shared variable

y--;//this thread decreaments its Value

sleep(1);//this thread two is preempted by thread 1

resource=y; //thread one updates the value of resource 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 resource is %d\n",resource);

}

A screenshot of a cell phone

Description automatically generated

**Description:**

The situation where two or more processes are reading or writing some shared data or resource & the final results depends on who runs precisely when are called race conditions. This race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data. It is a situation that may occur inside a critical section. This happens when the result of multiple thread execution in critical section differs according to the order in which the threads execute.Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction. Also, proper thread synchronization using locks or atomic variables can prevent race conditions. For example : When two or more concurrently running threads/processes access a shared data item or resources and final results depends on the order of execution, we have race condition. Suppose we have two threads A and B as shown below.

Thread A Thread B

Count++ Count--

Thread A increase the share variable count by 1 and Thread B decrease the count by 1. If the current value of Count is 10, both execution orders yield 10 because Count is increased by 1 followed by decreased by 1 for the former case, and Count is decreased by 1 followed by increased by 1 for the latter. However, if Count is not protected by mutual exclusion, we might get difference results

**Algorithm:**

Consider-

* Two processes P1 and P2 are executing concurrently.
* Both the processes share a common variable named “resource” having initial value = 1.
* Process P1 tries to increment the value of count.
* Process P2 tries to decrement the value of count.

Count=1

P1

{

----------------

Count++

-----------------

}

P2

{

----------------

Count--

-----------------

}

Now, when these processes execute concurrently without synchronization, different results may be produced.

**Description (purpose of use):**

The number of line of code of my program is 32 LOC .In this program the first function fun1 which is a increment function contains 8 lines of code and the time complexity for it is O(n),also for the second function fun2 which is a decrement function contains 8 lines of code with O(n) time complexity, and the main function contains 9 line of code with time complexity O(n). Therefore the overall time complexity of the program is O(n)+O(n)+O(n)= O(n)

**Code snippet:**

Thread 1 :  
void \*fun1()

{

int x;

x=resource;

x++; // this thread one increments its value

sleep(1);

resource=x; //thread one updates the value of resource variable

}

Thread 2 :

void \*fun2()

{

int y;

y=resource;//this thread two reads value of shared variable

y--;//this thread decreaments its Value

sleep(1);//

resource=y; //thread one updates the value of resource variable

}  
In Thread 1, after entering the if condition, there might be a chance Thread 2 has modified the value of n. Thus there is ambiguity in the output.  
Also we must realize that each "x++" or "y--" is three instructions internally : Load, Increment/Decrement, Store.  
There might be a case as where a thread might load x, while it is being incremented in another thread but not yet stored back.

**Description:**

Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction. The critical section in a code segment where the shared variables can be accessed. Atomic action is required in a critical section i.e. only one process can execute in its critical section at a time. All the other processes have to wait to execute in their critical sections.

The critical section problem needs a solution to synchronise the different processes. The solution to the critical section problem must satisfy the following conditions:

* **Mutual Exclusion**:Mutual exclusion implies that only one process can be inside the critical section at any time. If any other processes require the critical section, they must wait until it is free.
* **Progress**:Progress means that if a process is not using the critical section, then it should not stop any other process from accessing it. In other words, any process can enter a critical section if it is free.
* **Bounded Waiting** :Bounded waiting means that each process must have a limited waiting time. Itt should not wait endlessly to access the critical section.

**Description:**

Addition program to support the race condition using semaphore variable.

This program create two threads: one to increment the value of a shared variable and second to decrement the value of shared variable. Both the threads make use of semaphore variable so that only one of the threads is executing in its critical section :  
  
#include<pthread.h>  
#include<stdio.h>  
#include<semaphore.h>  
void \*fun1();  
void \*fun2();  
int resource=5; /  
sem\_t s; //semaphore variable  
int main()  
{  
sem\_init(&s,0,1);

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",resource); //prints the last updated value of resource variable  
}  
  
void \*fun1()  
{  
    int x;  
    sem\_wait(&s); //executes wait operation on s  
    x=resource;//thread one reads value of shared variable  
    x++;  //thread one increments its value  
    sleep(1);  //thread one is preempted by thread 2  
    resource=x; //thread one updates the value of shared variable  
    sem\_post(&s); //executes signal operation on s  
}  
  
void \*fun2()  
{  
    int y;  
    sem\_wait(&s);  
    y=resource;  
    y--;    
    sleep(1);    
    resource=y;   
    sem\_post(&s);  
}  
A screen shot of a computer

Description automatically generated  
 the final value of shared resource variable will be 1. When any one of the threads execute the wait operation the value of "s" becomes zero and hence the other thread (even if it preempts the running thread) is not able to successfully execute the wait operation on "s" thus not able to read the inconsistent value of shared variable. Thus only one of the thread is running in its critical section at any given time

**Description:**

**Test cases:-**

resource=1

P1

{

----------------

x++

-----------------

}

P2

{

----------------

y--

-----------------

}

### ****Case-01:****

The execution order of the instructions may be-

P1(1), P1(2), P1(3), P2(1), P2(2), P2(3)

In this case,

Final value of count of resource = 1

### ****Case-02:****

The execution order of the instructions may be-

P2(1), P2(2), P2(3), P1(1), P1(2), P1(3)

In this case,

Final value count of resource = 1

### ****Case-03:****

The execution order of the instructions may be-

P1(1), P2(1), P2(2), P2(3), P1(2), P1(3)

In this case,

Final value of count of resource = 2

### ****Case-04:****

The execution order of the instructions may be-

P2(1), P1(1), P1(2), P1(3), P2(2), P2(3)

In this case,

Final value count of resource = 0

### ****Case-05:****

The execution order of the instructions may be-

P1(1), P1(2), P2(1), P2(2), P1(3), P2(3)

In this case,

Final value count of resource = 0

It is clear from here that inconsistent results may be produced if multiple processes execute concurrently without any synchronization.

 When the resource shared is less than available resource it returns -1:

A close up of text on a white background

Description automatically generated

**GitHub Link :**

[**https://github.com/BadalSherpa/Operating-system**](https://github.com/BadalSherpa/Operating-system)