Synchronization/race condition/Critical section problem

**Program 1:**   
/\* Program to show the race condition.   
Program create two threads: one to increment the value of a shared variable and second to decrement thev 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>  
void \*fun1();  
void \*fun2();  
int shared=1; //shared variable  
int main()  
{  
pthread 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  
    x++;  //thread one increments its value  
    sleep(1);  //thread one is preempted by thread 2  
    shared=x; //thread one updates the value of shared variable  
}  
  
void \*fun2()  
{  
    int y;  
    y=shared;//thread two reads value of shared variable  
    y--;  //thread two increments its value  
    sleep(1); //thread two is preempted by thread 1  
    shared=y; //thread one updates the value of shared variable  
}  
  
/\* 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 \*/  
    
  
**Program 2:**  
/\* Program for process synchronization using locks   
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 locks so that only one of the threads is executing in its critical section \*/  
  
#include<pthread.h>  
#include<stdio.h>  
void \*fun1();  
void \*fun2();  
int shared=1; //shared variable  
pthread\_mutex\_t l; //mutex lock  
int main()  
{  
pthread\_mutex\_init(&l, NULL); //initializing mutex locks  
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;  
    pthread\_mutex\_lock(&l); //thread one acquires the lock. Now thread 2 will not be able to acquire the  
                                            //lock until it is unlocked by thread 1  
    x=shared;//thread one reads value of shared variable  
    x++;  //thread one increments its value  
    sleep(1);  //thread one is preempted by thread 2  
    shared=x; //thread one updates the value of shared variable  
    pthread\_mutex\_unlock(&l);  
}  
  
void \*fun2()  
{  
    int y;  
    pthread\_mutex\_lock(&l);  
    y=shared;  
    y--;    
    sleep(1);    
    shared=y;   
    pthread\_mutex\_unlock(&l);  
}  
  
/\* the final value of shared variable will be 1. When any one of the threads acquires the lock and is making changes to shared variable the other thread (even if it preempts the running thread) is not able to acquire the lock and 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 \*/  
  
**Program 3:**  
  
/\* Program for process synchronization using semaphores  
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 shared=1; //shared variable  
sem\_t s; //semaphore variable  
int main()  
{  
sem\_init(&s,0,1); //initialize semaphore variable - 1st argument is address of variable, 2nd is number of  
                           //processes sharing semaphore, 3rd argument is the initial value of semaphore variable  
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;  
    sem\_wait(&s); //executes wait operation on s  
    x=shared;//thread one reads value of shared variable  
    x++;  //thread one increments its value  
    sleep(1);  //thread one is preempted by thread 2  
    shared=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=shared;  
    y--;    
    sleep(1);    
    shared=y;   
    sem\_post(&s);

//The *sem\_post()* function unlocks the semaphore referenced by *sem* by performing a semaphore unlock operation on that semaphore.  
}  
  
/\* the final value of shared 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 \*/