Operating Systems (coe628)

Lab 6

Due Week of March 4, 2024

Description

In this lab, you will learn how to synchronize the actions of multiple threads. Synchronization is a way to ensure correct flow of execution between two or more threads working with shared data. We cover two types of synchronization: ***locking*** and ***waiting.***

Locking is used to prevent race condition between two or more threads to access shared data. It is used to ensure that only one thread that can access shared data at a time (to prevent race conditions).

Waiting is used to enforce the correct sequence of execution. In this lab, we will use **mutexes** in order to handle these two type of synchronization.

Objectives

The main objective of this lab is to learn how to synchronize threads in a POSIX, compliant operating system using C. (i.e. Identical source code should work in Linux, Solaris, Windows with cygwin, Mac OSX, etc.)

A *mutex*, which stands for mutual exclusion, is the most basic form of synchronization. A *mutex* is used to protect a critical region, to ensure that only one thread at a time executes the code within the region. Since only one thread at a time can lock a given *mutex*, this guarantees that only one thread at a time can be executing the instructions within the critical region. Although we talk of a critical region being protected by a *mutex*, what is really protected is the data being manipulated within the critical region. That is, a *mutex* is normally used to protect data that is being shared between multiple threads. Mutexes are for locking and cannot be used for waiting. Condition variables are used in combination with *mutex* in situations when waiting is needed.

What you have to do

Part 1: Please follow those Steps below to perform (Part 1)

1. Download the code lab6.c. In the code, each of the placeholders should be replaced with one or more C instructions in order to complete the program. The required libraries are included but you may need to include more libraries if you follow a different approach. In this program, the goal is to have five threads each of which generates 2000 random numbers and adds them to the shared variable *sum*. The generator threads have been implemented in the generator unction. Read this function and make sure you understand what it is doing.
2. It is probably convenient to replace the random number added with the constant 1 and to reduce the number of loops to 20 instead of 2000. Thus each thread should increment 20 times; with 5 threads the total should be 5*20 =100.

3. Replace placeholder **A** with the code for creating five generator threads and variables for keeping them as you learned in Lab 5.
4. Replace placeholder **B** with the code for making sure the all five threads have been finished before the main function finishes.
5. Now your program should work and at the end of its execution the sum of generated value is stored in the *sum* variable. In order to verify if the program is working correctly, every generator function also sums its generated values and prints it when it finished generating numbers.
6. Run the program and check if the program has performed correctly. Most probably, the sum of separate generator classes is not equal to total sum. This is because the access to shared variable has not been synchronized. The region of code working with sum variable is a critical section and only one thread should be able to execute it at a time. *Mutex* can be used to ensure exclusive access to critical section of the code working with *sum*.
7. A *mutex* is a variable of type pthread_mutex_t. Astatic *mutex* can be initialized by a constant PTHREAD_MUTEX_INITIALIZER. A static *mutex* works most of the time. Replace placeholder **C** with static declaration of a *mutex* and initialize it.
8. Lock function (*pthread_mutex_lock* function*)* should be called on the *mutex* just before the region of the code that is considered critical section. If a thread tries to lock a mutex that is already locked by some other thread*, pthread_mutex_lock* blocks until the *mutex* is unlocked. Place the appropriate call to *mutex* lock in the correct location on the code.
9. After the critical section is finished the thread should unlock the mutex (by calling *pthread_mutex_unlock* function) in order to allow other threads to enter the critical section. Place appropriate call to mutex unlock in the correct location of the code. Make sure all generators can run interleavably but in a safe manner. Now your program should run correctly, run the program again and verify its correctness.
10. The *print_function* prints the value of the *sum* variable. This function is called after all generators have finished. (This ends part 1.)

Part 2: Please follow those Steps below to perform (Part 2)

11. We will try to create a new thread instead of calling it as a function from the main thread. Remove *print_function* call from the main function and replace placeholder **D** with the code for creating a thread that runs the print thread and a variable for accessing it.
12. Replace placeholder **E** with the code for making sure the print thread has finished before the main function finishes.
13. Now the printing function will run as a thread too, but we need to make sure that printing the value of *sum* is executed after all random numbers have been generated. In other words, the print function should wait until all generators have finished generating numbers. Therefore, a waiting mechanism is needed to ensure this synchronization. Condition variables can be used for this purpose. *pthread_cond wait* and *pthread_cond_signal* are two main functions of condition variables. The *pthread_cond wait* puts a thread into sleep until *a* pthread_cond_signal call is made on the same variable. Replace placeholder **F** with declaration and initialization of a condition variable, which is a variable of type *pthread_cond_t*. A condition variable can be initialized by assigning *PTHREAD_COND_INITIALIZER* to it.
14. The program uses *finished producers* to store the number of generator threads that have finished their work. Replace placeholder **G** with the correct instructions to put the thread into sleep if all

generators have not finished working yet. Notice that it is possible that the wrong signal may be fired on a condition variable and hence it's highly recommended to use 'while' instead of 'if 'for checking a condition.

15. Replace placeholder **H** with needed code so the producer thread fires a signal on the condition variable if all generators have finished working.

*Run you program in order to make sure it works correctly.

Submit your lab

On a departmental lab computer, do the following

a. Zip the submission folder:
   zip -r coe628_lab6.zip coe628_lab6
b. Submit the folder:
   submit coe628 lab6 coe628_lab6.zip