CS 451 – Operating Systems

Lab 06 Assignment

Due: 11:59 PM on Wednesday, 1 September 2020

You are to submit your files, to the Lab 6 folder on Blackboard by the due date and time.

Objective

- Learn how to synchronize threads in a POSIX compliant operating system using C.
- Implement conditional variables when waiting is required

Description

In this lab, you will learn how to synchronize the actions of multiple threads. Synchronization is a means 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 types of synchronization. **Condition variables** are used in combination with mutex in situations when waiting is needed.

Condition Variables:

Overview

Condition variables provide yet another way for threads to synchronize. While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data. Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met. This can be very resource consuming since the thread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling.

A condition variable is always used in conjunction with a mutex lock. The typical sequence for using condition variables:

- Create and initialize a condition variable
- Create and initialize an associated mutex
- Define a predicate variable (variable whose condition must be checked)

Creating / Destroying Condition Variables:

`pthread_cond_init ( pthread_cond_t condition, pthread_condattr_t attr)`

`pthread_cond_destroy ( pthread_cond_t condition)`

`pthread_condattr_init ( pthread_condattr_t attr )`

`pthread_condattr_destroy ( pthread_condattr_t attr )`

`pthread_cond_init( )` creates and initializes a new condition variable object. The ID of the created condition variable is returned to the calling thread through the condition parameter. Condition variables must be of type `pthread_cond_t` **.**

The optional **`attr`** object is used to set condition variable attributes. There is only one attribute defined for condition variables: process-shared, which allows the condition variable to be seen by threads in other processes. The attribute object, if used, must be of type **`pthread_condattr_t`** (may be specified as NULL to accept defaults).

Waiting / Destroying Condition Variables:

`pthread_cond_wait ( pthread_cond_t condition, pthread_mutex_t mutex )`

`pthread_cond_signal ( pthread_cond_t condition )`

`pthread_cond_broadcast ( pthread_cond_t condition )`

`pthread_cond_wait( )` blocks the calling thread until the specified condition is signalled. This routine should be called while mutex is locked, and it will automatically release the mutex while it waits.

The `pthread_cond_signal( )` routine is used to signal (or wake up) another thread which is waiting on the condition variable. It should be called after mutex is locked.

The `pthread_cond_broadcast( )` routine should be used instead of `pthread_cond_signal( )` if more than one thread is in a blocking wait state.

It is a logical error to call `pthread_cond_signal( )` before calling `pthread_cond_wait( )`.

What is mutex?

A **mutex**, which stands for *mutual exclusion*, is the most basic form of synchronization. A mutex is used to protect a critical region, to make certain 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.

Task 1

1. Download the code **`lab__6.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 200 random numbers and adds them to the shared variable sum. The generator threads have been implemented in **`generator_function()`**. Read this function and make sure you understand what it is doing.

2. Replace placeholder A with the code for creating five generator threads and variables for keeping them.

3. Replace placeholder B with the code for making sure the all five threads have been finished before the main function finishes.

4. 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. Run the program and check if the program has performed correctly also include your program output in the README.

5. 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.

6. A mutex is a variable of type `pthread_mutex_t`. A static mutex can be initialized by a constant `PTHREAD_MUTEX_INITIALIZER`. Replace placeholder C with static declaration of a mutex and initialize it.

7. 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.

8. 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.

9. The `print_function` prints the value of the sum variable. This function is called after all generators have finished. (This ends task 1.)

Task 2 - Conditional Variables

1. We will try to create a new thread instead of calling it as a function from the main thread.
2. 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.
3. Replace placeholder E with the code for making sure the print thread has finished before the main function finishes.
4. 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.
5. 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.
6. Replace placeholder H with needed code so the producer thread fires a signal on the condition variable if all generators have finished working.

Note: Run you program in order to make sure it works correctly. Include the output of the program in the README.

What to turn in

- Submit the **‘lab__6.c’** file
- A makefile (do not forget to link pthread)
- A README file (include outputs (at the end) for task 1 and task 2)