CST–321 Mutexes and Semaphores

Activity Directions:

In this project, you will demonstrate your understanding of *mutexes* and *semaphores*. Define a simple scenario (in the context of processes and threads in Linux) in which a mutex could be used. Implement your scenario in a C program. Your scenario must be different than any scenario used in any activities.

Define a simple scenario (in the context of processes and threads in Linux) in which a counting semaphore could be used. Your scenario must be different than any scenario completed as part of the Activity 2 Guide. Your counting semaphore must NOT be implemented as a binary semaphore and must have a count greater than one. Implement your scenario in a C program.

For the threads used in both the mutex and the semaphore, you are also not allowed to use the increment operator (++) or decrement operator (--) on the shared resource because these operators are typically thread safe to begin with.

Your screencast must prove without a doubt that a mutex and semaphore are working properly and fixed an actual problem in the code. This should be done by showing the code in a failed state during your screencast.

Example scenarios could include, but are not be limited to, the following:

Semaphore

- Semaphores are typically used to coordinate access to resources, with the semaphore count initialized to the number of free resources.
- A semaphore with a count of 1 behaves much like a mutex and is also known as a binary semaphore.
- When a semaphore is used to control access to a pool of resources, a semaphore tracks only how many resources are free; it does not keep track of which resources are free.
    - wait: decrements the value of semaphore variable by 1. If the new value of the semaphore variable is negative, the process executing wait is blocked (i.e., added to the semaphore's queue). Otherwise, the process continues execution, having used a unit of the resource.
    - signal: increments the value of semaphore variable by 1. After the increment, if the pre-increment value was negative (meaning there are processes waiting for a resource), it transfers a blocked process from the semaphore's waiting queue to the ready queue.
- Practical Scenarios:
    - A system that can only support ten users
    - The producer and consumer problem

- A bar, movie theater, or restrooms that only lets in so many people
- A fixed thread pool that will perform a bunch of work in parallel
- A fixed database connection pool
- A fixed web server request pool
- A parking lot that only holds so many cars

Mutex

- A mutual exclusion object (mutex) is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously. For example:
    - A thread is: each person
    - The mutex is: the door handle
    - The lock is: the person's hand
    - The resource is: the phone
- Practical Scenarios:
    - A bank transaction
    - An ecommerce application tracking inventory of available items (tickets, anything in Amazon)
    - Anything that shares a counter
    - Preventing concurrent access/reading from a file or database data (i.e., preventing dirty reads)
    - A phone booth, a shared mobile phone across family members
    - Inserting data into a common data structure or linked list
    - A one-lane bridge that counts cars

Additional Resources:
https://www.tutorialspoint.com/mutex-vs-semaphore
https://www.geeksforgeeks.org/mutex-vs-semaphore/
https://see.stanford.edu/materials/icsppcs107/23-Concurrency-Examples.pdf

Deliverables:

1. Cover sheet with your name, the name of this assignment, and the date.
2. Detailed description of the scenario, emphasizing the need for a synchronization mechanism.
3. A clear, justified recommendation for the use of mutexes or semaphores in this particular scenario.
4. Analysis of the pros and cons comparing the use of mutexes vs. the use of semaphores.
5. Coding results:
    a. All the source code files. Comment your code, describing the solution and identifying the programmer. Zip up the source code (not the binaries) in a single zip file.

    b. A 3- to 5-minute screencast showing successful execution of the program and explanation of the code, including a demonstration of the application in a failed condition.
6. Package the URL for the screencast and description of the solution into a one document and upload it to the digital classroom.
7. Zip up the source code (not the binaries) in a single zip file and upload it to the digital classroom.