Systems Software/Programming – Lab Manual

Lab 10 – POSIX Thread Management and Synchronization

Write C program for each of the problem below. After executing each of the C program, you will need to capture the output in text file format.
C program file will be named as StudentID_Lab10_x.c
Text file with output captured will be names as StudentID_Lab10_x.txt

Note: All of these problems will require thread synchronization mechanisms. These assignments will teach how to divide the computation work among different threads so that it can be done with improved run time.

Information used for thread synchronization problems:
Assume that you are given a task to maintain a memory page table management. In our case page table is nothing but an array ptable[5][3] which keeps 5 entries, where 1st column has process id (generate random number from 1-10), 2nd column has page number (generate random number from 50-1000, assuming first 50 pages are used by OS) and the 3rd column has how frequently these pages have been accessed (initialize to 1 – because we assume that the page is loaded due to the first read request) . We will assume that each process id will store only one page number ptable which means there a single entry for each unique process id.

Problem 1: Write C program StudentID_Lab10_1.c to do the following. This problem will teach you how to use POSIX MUTEX locks
We may have multiple thread requesting for reading and updating the entries in ptable. In this problem we assume that:

- Only one thread is allowed to update the ptable entry
    - Writer will randomly generate process id (between 1-10), page number (between 50-1000) and frequency (initialize to 1).
    - It will check if ptable is full (all 5 entries filled).
        - If any of the entries is available, it will write new entry into empty spot.

        - If all 5 entries are full writer will check for the frequency count of all 5 entries and will overwrite the entry which has the least count (also known as Least Frequently Used - LFU Algorithm)
- Only one thread is allowed to read the ptable entry
    - Reader will randomly generate process id (between 1-10) and will increment the frequency count for that process id entry in ptable

Problem 2: Write C program StudentID_Lab10_2.c to do the following. This problem will teach you how to use POSIX READ-WRITE locks
We modify problem 1 by stating additional requirement that:

- More than one threads are allowed to read the ptable entry
    - Reader will randomly generate process id (between 1-10).
    - Only one thread will increment the frequency count for that process id entry in ptable by acquiring a lock on mutex

Problem 3: Write C program StudentID_Lab10_3.c to do the following. This problem will teach you how to use SEMAPHORES
We modify the problem 2 further by stating that:

- More than one threads are allowed to update the ptable as long as it is updating a different entry
    - Writer will randomly generate process id (between 1-10) and (50-1000)
    - It will check if there is any ptable is full (all 5 entries filled). If any entry if empty writer will write this new entry
    - If all 5 entries are full writer will check for the frequency count of all 5 entries
    - It will acquire a lock for an entry and will overwrite the entry which has the least count (also known as Least Frequently Used - LFU Algorithm) then release the lock

Optional Assignment:
You are given a task to write a code to print the file content onto a printer. (You can assume that in our case printer is non-buffered output to STDOUT). There are

3 threads used in the process: thread 1 is reading the file and putting the content into filebuf char array, thread2 copies the content from filebuf to printbuf and thread 3 takes content from printbuf print it (display it). You can assume that filebuf and printbuf char arrays are of the same size so you can copy the whole content from filebuf to printbuf and empty the filebuf. You can also assume that at a time only one process is trying to print the file

Ensure the following:

- Thread 1 must wait if filebuf is full. If filebuf is empty Thread 1 start reading the file and puts the data in filebuf
- Thread 2 must wait if filebuf is empty or printbuf is full. Once it sees that filebuf is full and printbuf is empty, Thread 2 will copy the content from filebuf to printbuf
- Thread 3 must wait if printbuf is empty. It will print/display the content once printbuf is full