CS431 Operating Systems
Programming Assignment 4 - Thread Synchronization Using Mutex Locks and Semaphores

1 Overview

While multi-threaded applications offer a number of benefits over multi-process designs, they have one important limitation: Synchronizing threads' access to shared data and preventing race conditions is the responsibility of the programmer. With processes, the kernel provides that synchronization. With threads, synchronization must be written into the program. In this programming assignment, you will practice using two synchronization primitives, *semaphores* and *mutex locks*, in a multi-threaded program using the pthread API.

2 Requirements

Refer to the program code given in the file *thread_sync.c*. Although the program does not perform any particularly useful tasks, it demonstrates some important concepts regarding threading. The program involves a large number of concurrent threads that will each write to and read from portions of a file at specific locations within it, effectively serving as a shared buffer to be synchronized. In the main program a total of 20,000 threads are created: 10,000 for writing and 10,000 for reading. Note: if you find your computer too resource limited to manage that many threads, feel free to reduce that number appropriately.

The threads that write to the file execute first, and the main program thread blocks in the meantime. The threads that read from the file execute following all writer threads.

Your job is to modify the two thread functions to implement synchronized reading and writing. Each writer thread will write a portion of the file at a designated offset into it using the `pwrite` system call. The reader threads will each read from the file at a similar file offset using the `pread` system call.

3 Synchronization

This section describes the requirements for the thread functions for the reader and writer threads.

3.1 Writer Threads

This is the function called by each writer thread. Each thread will take the following steps:

1. Place the thread's ID number into the `char` array named `buffer`, that is already declared. The resulting character string **must** be exactly five characters in length. For this, consider using `sprintf` with a format specifier. If you place a number after the % in the format specifier, that indicates the minimum field width. If the resulting string happens to be less than the width, it will be filled with blank spaces. Refer to the sample file contents in Section 3.3.

2. Call wait on the semaphore. Refer to the man page for `sem_wait`.

3. Acquire the mutex lock. Refer to the man page for `pthread_mutex_lock`.

4. Do a `pwrite` of the `char` array `buffer` on the file at the designated file offset for each thread. The designated offset is its thread ID number multiplied by DATA_SIZE. At that location in the file, each thread will write exactly DATA_SIZE, i.e., 5, bytes. Refer to the man page for `pwrite` for parameters and return type.

5. Increment the variable `bytes_produced_consumed` by DATA_SIZE number of bytes.

6. Release the mutex lock. Refer to the man page for `pthread_mutex_lock`.

7. Call signal on the semaphore. Refer to the man page for `sem_post`.

Steps 4 and 5 constitute the critical section of the thread. The mutex lock (and therefore exclusive write access) is needed to write to the file and modify the variable representing the total number of bytes written (produced) to the file shared by all threads. However, the semaphore, initialized to 5, allows up to five threads at a time to contend for the mutex lock. All others will block.

3.2 Reader Threads

This is the function called by each reader thread. Each thread will take the following steps:

1. Call wait on the semaphore. Refer to the man page for `sem_wait`.

2. Acquire the mutex lock. Refer to the man page for `pthread_mutex_lock`.

3. Do a `pread` of the file at the thread's designated file offset, similar to that of the writer threads. Each thread will read exactly DATA_SIZE, i.e., 5, bytes. Refer to the man page for `pread` for parameters and return type.

4. Decrement the value of `bytes_produced_consumed` by DATA_SIZE number of bytes.

5. Release the mutex lock. Refer to the man page for `pthread_mutex_lock`.

6. Call signal on the semaphore. Refer to the man page for `sem_post`.

7. Output the result of the `pread`.

Steps 3 and 4 constitute the critical section of the reader thread. The mutex lock is needed to read from the file and modify the shared variable representing the total number of bytes remaining to be read from the file (consumed), as with the writer threads.

Note that all writer threads execute first, followed by the reader threads. Your program should not have both reader and writer threads executing concurrently.

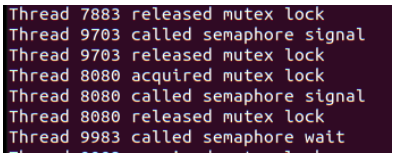

Figure 1: Example Program Console Output

3.3 Program Output

When the program runs, you will see output indicating which thread is running. There will in most cases be no consistent or predictable sequencing of the threads; the order of thread execution is scheduled in a non-deterministic manner. Console output will resemble Figure 1.

Once the program terminates, the contents of the file *sharedfile.txt* should look like this:

```
0000 0001 0002 0003 0004 0005 0006 0007 0008 0009 0010 0011 0012 0013 0014 0015
0016 0017 0018 0019 0020 0021 0022 0023 0024 0025 0026 0027 0028 0029 0030 0031
0032 0033 0034 0035 0036 0037 0038 0039 0040 0041 0042 0043 0044 0045 0046 0047

...

9968 9969 9970 9971 9972 9973 9974 9975 9976 9977 9978 9979 9980 9981 9982 9983
9984 9985 9986 9987 9988 9989 9990 9991 9992 9993 9994 9995 9996 9997 9998 9999
```

4 Building Programs with Pthreads

When building the program, be sure to add the linker option `-lpthread` so that the Pthread library is correctly linked to the executable program, i.e.,

```
gcc -o thread_sync thread_sync.c -lpthread
```

5 Deliverables

Submit the C code source file for your program through the WorldClass assignment dropbox no later than the due date specified in the weekly assignment. **Be sure to add your name to the header block of comments**.

6 Academic Integrity

I recognize that there are only so many ways to write code that meets the requirements of this assignment, and you may consult with other students in this class on the general approach to a solution. Having said that, sharing code, pseudocode, or other implementation details is not permitted, and I expect all work to be your own as evidenced by:

- Program structure, control flow, variable and function names (aside from C library functions, of course), etc. should be manifestly distinct from others' work.
- Your program should be commented in a manner sufficient for me to ascertain that you understand what the code you submit does.