CS333 Autumn 2019

Lab Quiz2: Synchronization

Total Marks: 20

Instructions

Before you begin, please check that your top-level home directory has the following files:

- This question paper.
- A tarball `labquiz2_files.tgz`. Untar this file as shown below:
  `tar -zxvf labquiz2_files.tgz`
  You will now find a folder `labquiz2_files` in the home directory, which has files pertaining to the various questions of this lab in the corresponding subfolders. Please refer to these subfolders when reading the problem statement of the corresponding question.
- A tutorial on the `pthreads` library.

Now, create a folder titled `rollnumber_labquiz2` in the same top-level home directory, where you must replace the string `rollnumber` above with your own roll number (e.g., your directory name should look like `12345678_labquiz2`). Create sub-directories for each question, titled `q1`, `q2`, and `q3`, in this directory. After solving the lab questions, submit the following solution files for grading:

- For question 1, submit `q1/master-worker.c`
- For question 2, submit `q2/semaphore.h`, `q2/semaphore.c`, `q2/test_toggle.c` and any other xv6 files you may need to modify
- For question 3, submit `q3/zemaphore.h`, `q3/zemaphore.c`, `q3/test-toggle.c`, `q3/rwlock.h`, `q3/rwlock.c`

Next, tar/gzip your submission directory using the following command:
`tar -zcvf 12345678_labquiz2.tgz 12345678_labquiz2`
to produce a single compressed file of your submission directory. We will pull this submission file from the top-level home directory at the end of the quiz. We will only use this code pulled by us for evaluation.

Please note: Please ensure that you follow the above submission instructions carefully. Make sure that your submission folder contains the correct code that you intended to submit. We will only grade the submissions pulled by us from the lab machines, and no code changes will be permitted after submission.

Exercises

1. **Multiple producers and consumers [5 marks]** In this question, you will implement the master/worker or producer/consumer synchronization pattern that you have seen in an earlier lab, with the change that your code must now support multipe master threads along with multiple worker threads. Much like in the previous lab, the master threads must "produce" numbers by placing them in a shared buffer array, and the worker threads must "consume" the numbers from the buffer and print them to screen.

   You are given a skeleton program `master-worker.c`. This program takes 4 command line arguments: how many numbers to "produce" (M), the maximum size of the buffer in which the produced numbers should be stored (N), the number of worker threads to consume these numbers (C), and the number of master threads to produce numbers (P). The skeleton code spawns P master threads, that produce the specified number of integers from 0 to $M-1$ into a shared buffer array. The main program waits for these threads to join, and then terminates. The skeleton code given to you does not have any logic for synchronization.

   You must modify this skeleton code in the following ways. You must add code to spawn the required number of worker threads, and write the function to be run by these threads. This function will remove/consume items from the shared buffer and print them to screen. Further, you must add logic to correctly synchronize the producer and consumer threads in such a way that every number is produced and consumed exactly once. Further, producers must not try to produce when the buffer is full, and consumers should not consume from an empty buffer. Your solution must only use `pthreads` condition variables for waiting and signaling. (Busy waiting is not allowed.)

   Recall that all programs that use `pthreads` must be linked to the `pthreads` library during compile time. For example, the code given to you can be compiled as shown below.

   ```
   gcc master-worker.c -lpthread
   ```

   If your code is written correctly, every integer from 0 to $M-1$ will be produced exactly once by the master producer thread, and consumed exactly once by the worker consumer threads. We have provided you with a simple testing script that checks this above invariant on the output produced by your program. The script relies on the two print functions that must be invoked by the producer and consumer threads: the master thread must call the function `print_produced` when it produces an integer into the buffer, and the worker threads must call the function `print_consumed` when it removes an integer from the buffer to consume. You must invoke these functions suitably in your solution. Please do not modify these print functions, as their output will be parsed by our grading script.

2. **Semaphores in xv6.** In this question, you will implement the functionality of semaphores in xv6. You will define an array of (NSEM = 10) semaphores in the kernel code, that are accessible across all user programs. User processes in xv6 will be able to refer to these shared semaphores by an index into the array, and perform up/down operations on them to synchronize with each other.

   Untar the original xv6 codebase and our provided patch. Our patch adds three new system calls to the xv6 kernel:

   - `sem_init(index,val)` initializes the semaphore counter at the specified `index` to the value `val`. Note that the index should be within the range [0, NSEM-1].
   - `sem_up(index)` will increment the counter value of the semaphore at the specified index by one, and wakeup any one sleeping process.
   - `sem_down(index)` will decrement the counter value of the semaphore at the specified index by one. If the value is negative, the process blocks and is context switched out, to be woken up by an up operation on the semaphore at a later point.

   The scaffolding to implement these system calls has already been done or you, and you will need to only implement the system call logic in the files `semaphore.h` and `semaphore.c`. You must define the semaphore structure, and the array of NSEM semaphores, in the header file. You must also implement the functions that operate on the semaphores in the C file. Your implementation must directly invoke (or reimplement with minor changes, if required, the logic of) the `sleep` and `wakeup` functions of the xv6 kernel. If you are writing modified sleep/wakeup functions in xv6, you may modify the files `proc.c`, `proc.h`, and `defs.h`.

   (a) **[3 marks]** Implement the logic of the three system calls specified above. Then, run the test program `test_sem` given to you to verify the correctness of your implementation. The test program spawns two children. When you run the test program before implementing semaphores, you will find that the children print to screen before the parent. However, once the semaphore logic has been correctly implemented by you in `semaphore.c` and `semaphore.h`, you will find that the parent prints to screen before the children, by virtue of the semaphore system calls. Do not modify this test program in any way, but simply use it to test your semaphore implementation.

   (b) **[2 marks]** Now, consider another test program `test_toggle` provided to you in the patch. This program spawns two children, and all three processes (one parent and 2 children) print several statements to screen. We would like you to add semaphore-based synchronization to this program such that the print statements of the processes are interleaved with each other in the order child1, child2, parent, child1, child2, parent, ... and so on. You must only add code corresponding to the semaphore related system calls to the file `test_toggle.c`, and you must not change anything else in this program.

3. **Semaphores using `pthreads` condition variables and mutexes.** In this lab, you will implement the synchronization functionality of semaphores using `pthreads` mutexes and condition variables. Let's call these new userspace semaphores that you implement as `zemaphores`, to avoid confusing them with semaphores provided by the Linux kernel. You must define your zemaphore structure in the file `zemaphore.h`, and implement the functions `zem_init`, `zem_up` and `zem_down` that operate on this structure in the file `zemaphore.c`. The semantics of these zemaphore functions are similar to those of the semaphores you have studied in class.

   You will now run several test programs using your zemaphore implementation described above. The script `build.sh` can compile these test programs for you.

   (a) **[3 marks]** Once you implement your zemaphores, you can use the program `test-zem.c` to test your implementation. This program spawns two threads, and all three threads (the main thread and the two new threads) share a zemaphore. Before you implement the zemaphore logic, the new threads will print to screen before the main thread. However, after you implement the zemaphore correctly, the main thread will print first, owing to the synchronization enabled by the zemaphore. You must not modify this test program in any way, but only use it to test your zemaphore implementation.

   (b) **[2 marks]** You are given a simple program with three threads in `test-toggle.c`. In its current form, the three threads will all print to screen in any order. Modify this program in such a way that the print statements of the threads are interleaved in the order thread0, thread1, thread2, thread0, thread1, thread2, ... and so on. You must only use your zemaphores to achieve this synchronization between the threads. You must not directly use the mutexes and condition variables of the `pthreads` library in the file `test-toggle.c`.

   (c) **[5 marks]** You will now use zemaphores to implement reader-writer locks. A reader-writer lock provides separate functions to lock and unlock for reader and writer threads. Multiple reader threads can hold the lock concurrently. However, as always, no writer should hold the lock concurrently with another reader or writer. In this question, you must implement reader-writer locks with reader preference. That is, requests for a read lock made by threads, while other reader threads already hold the lock, should be allowed, even if allowing such readers may prolong the waiting time of writers.

   You must define your reader-writer lock structure in `rwlock.h`, using one or more zemaphores, and any other regular variables (e.g., integers) as required. You must implement the reader/writer lock/unlock functions in `rwlock.c`, using only the init/up/down operations on your zemaphores for synchronization. You must not access the zemaphore counter directly or perform any other operation on the zemaphore other than the init/up/down operations. You must not use any other synchronization primitives (e.g., `pthreads` condition variables or mutexes).

   We have provided a test program `test-rwlock.cpp` to test your reader-writer lock implementation. This program takes two arguments: R and W. It then spawns R reader threads, followed by W writer threads, and then R reader threads again. Each of these threads try to acquire a common reader-writer lock, and hold it for a certain duration. The program monitors the times when the threads acquire and release the locks to verify the correctness of your implementation. You can test your reader-writer lock implementation by running this program with different values of R and W (say, in the range 0–20). You need not submit this test program.