Operating Systems (Code: COMP2006)

CURTIN UNIVERSITY
Computing Discipline
School of Electrical Engineering, Computing, and Mathematical Sciences

Simple Data Sharing
Due date: 4pm, 7 May, 2018

You are asked to write a program in C in Linux environment to simulate the operations of a set of readers and a set of writers to share a given set of data, i.e., implementing the first readers – writers problem.

1. Consider a file named **shared_data** that is to be shared by a set of ***r*** **readers**. However, we assume that the readers cannot read the data *directly* from the file. Instead, each reader can only read the content of the **shared_data** from a **shared memory** area called **data_buffer**. For simplicity, assume the **shared_data** contains only d integers, i.e., a sequence (1, 2, 3, …, d), and the **data_buffer** is an array of b integers, for $b \leq d$. To test your program, you can set the constants $d = 100$ and $b = 20$.

2. Consider a set of ***w*** **writers** each of which can get only **one** integer at a time from the **shared_data**, and write it into the **data_buffer**. Further, each integer should be retrieved from the **shared_data** and written to the **data_buffer** only by one writer at a time. Thus, we expect the **data_buffer** to contain a sequence of b unique integers at maximum.

3. Access to the **data_buffer** follows the **readers / writers** problem with priority given to the readers, i.e., the first readers – writers problem. More specifically, multiple readers can read the buffer at the same time, but a writer can write to the buffer only when there is no other writer writing and no reader reading at the time. As discussed in the textbook / lecture, a reader is not allowed to access the buffer when a writer is accessing it, i.e., it must wait until the writer has finished writing.

4. Create ***r*** **tasks**, i.e., **reader-1**, **reader-2**, …, **reader-*r***, each of which runs a **reader()** function. Each **reader-*i*** keeps a counter c_i (initialized with 0) that gives the number of data it has read. For each **reader-*i***, the routine **reader()** is to read data from the **data_buffer** and increment its counter c_i by one. In the simulation, the function calls **sleep (t_1)** after reading and updating the counter. Each reader task terminates after it has finished reading all shared data, *not* when the **data_buffer** is empty. You can you're your own protocol such that the readers know that data in **shared_data** have all been written to the **data_buffer**. Before terminating, **reader-*i*** writes a message to a shared file named **sim_out**, i.e.,

   reader-pid has finished reading c_i pieces of data from the data_buffer.

   where **pid** is the pid of the terminating task. **Note** that a correct simulation produces each $c_i = d$.

5. Create ***w* tasks**, i.e., **writer-1**, **writer-2**, …, **writer-*w***, each of which runs a **writer()** function. Each **writer-*j*** keeps a counter c_j (initialized with 0) that gives the number of data it has written. For each **writer-*j***, the routine **writer()** is to get data from the **shared_data**, write the data to the **data_buffer,** increment its counter c_j by one. In the simulation, the function calls **sleep (t_2)** after writing and updating the counter. Each writer task terminates after the data in **shared_data** have all been written to the **data_buffer**, *not* when the **data_buffer** is full. You can make your own protocol for the writers such that each of them knows when to terminate. Before terminating, **writer-*j*** writes a message to the **sim_out**, i.e.,

   writer-pid has finished writing c_j pieces of data to the data_buffer

   where **pid** is the pid of the terminating task. **Note** that a correct simulation produces $\sum_{j=1}^{w} c_j = d$ among all w writers.

6. To test for the correctness of your program, you should run the program as follows:

   sds r w, t1 t2

   To test your program, you can use r = 5, w = 2, t1 = 1, t2 = 1, or any other possible values.

Implementation using Pthread (40%)

1. Create one thread for each of the ***r* tasks**, **reader-1**, **reader-2**, …, **reader-*r***, and one thread for each of the ***w* tasks**, **writer-1**, **writer-2**, …, **writer-*w***.

2. Use pthread mutual exclusion functions, i.e., pthread_mutex_lock(), pthread_mutex_unlock(), pthread_cond_wait(), pthread_cond_signal() to solve the critical section problems in the project.

3. Use pthread_cond_wait( ) and pthread_cond_signal( ) for each waiting reader and writer.

4. You have to describe/discuss in detail each of the variables, including its data structure, the threads that access them, and how mutual exclusion is achieved.

5. Remember to clean up all resources created in your program. The parent thread can handle this job before terminating itself.

Implementation using Process (40%)

1. Create one process for each of the ***r* tasks**, **reader-1**, **reader-2**, …, **reader-*r***, and one process for each of the ***w* tasks**, **writer-1**, **writer-2**, …, **writer-*w***.

2. Use POSIX semaphores to solve the critical section problems in the project.

3. Since processes do not share memory, the parent process needs to create at least one shared memory area, i.e., the **data_buffer**.

4. Read Chapter 3 of the textbook (Operating System Concepts by Silberschatz, et al.) to learn how to create shared memory, and Chapter 5 on how to use POSIX semaphores. Make sure that the parent process waits for the termination of its child processes and remove the shared memory, and semaphores.

5. You have to describe/discuss in detail each of the variables, including its data structure, the processes that access them, and how mutual exclusion is achieved.

6. Remember to clean up all resources created in your program, i.e., shared memory, etc. The parent process can handle this job before terminating itself.

Instruction for submission

1. Assignment submission is **compulsory**. Students will be penalized by a deduction of ten percent per calendar day for a late submission. **An assessment more than seven calendar days overdue will not be marked and will receive a mark of 0**.

2. You must (i) submit a hard copy of your assignment report to the Computing Department's office, (ii) submit the soft copy of the report to the unit Blackboard (**in one zip file**), and (iii) put your program files i.e., read_write.c, makefile, and other files, e.g., test input, in your home directory, under a directory named **OS/assignment**.

3. Your assignment report should include:

   - A signed cover page that includes the words "Operating Systems Assignment", your name in the form: family, other names, and a declaration stating the originality of the submitted work, that it is your own work, etc. Your name should be as recorded in the student database.

   - Software solution of your assignment that includes (i) all source code for the programs with proper in-line and header documentation. Use proper indentation so that your code can be easily read. Make sure that you use meaningful variable names, and delete all unnecessary comments that you created while debugging your program; and (ii) readme file that, among others, explains how to compile your program and how to run the program.

   - Detailed discussion on how any mutual exclusion is achieved and what processes / threads access the shared resources.

   - Description of any cases for which your program is not working correctly or how you test your program that make you believe it works perfectly.

   - Sample inputs and outputs from your running programs.

   Your report will be assessed (worth 20% of the overall assignment mark).

4. Due dates and other arrangements may only be altered with the consent of the majority of the students enrolled in the unit and with the consent of the lecturer.

5. Demo requirements:

- You may be required to demonstrate your program and/or sit a quiz during workshop sessions (to be announced).

- For demo, you MUST keep the source code of your programs in your home directory, and the source code MUST be that submitted. The programs should run on any machine in the department labs.

Failure to meet these requirements may result in the assignment not being marked