ASSIGNMENT #4

Readers-Writers using POSIX API

CSCI 480/514 100 points + 25 extra credits Spring 2023

Check Blackboard for due date. Due by 11:59 PM.

Specifications

In this assignment, you will solve the Reader-Writer problem using the PThreads library. You will get to practice using semaphores in the PThreads library.

There are two parts in this assignment. Part 1 implements the Reader-Writer exactly as we discussed in class. Part 2 is a variant and *serves as extra credit*.

Allocate sufficient time for this assignment. Tackle the problem step by step. For example, first make sure that your threads are all created successfully and let each thread do something trivial, e.g. print out its own id. Then add the semaphores to solve the readers/writers problem.

Preparation

1. Compile and run the provided Pthreads examples in Learning Module 6 on turing/hopper (folder of "PThread code examples").
   Example:
   g++ -o thread_simple thread_simple.cc -lpthread
   Note that you need to include the "-lpthread" when compiling.
   Study these example codes, especially the *thread_simple.cc*, so that you understand their logic. The example will be helpful for the assignment.
2. Study the POSIX Pthreads/semaphore API and the reader-writer problem in the slides/textbook.
3. Read the PreparationforPThreads.pdf, which contain some explanation of an example.

Project Description

PART 1

Your program takes in two command line arguments: number of reader threads, number of writer threads. Your program creates several reader threads and several writer threads based on the arguments. Each thread loops for several iterations for writing or reading.

The readers read a shared string and print the contents to the standard output. The writers write to the same string. The synchronization requirement is the same as explained in class (only one writer can write at any time. Readers can access concurrently, but a writer needs mutual exclusion with other writers and all readers).

Your code needs to initialize the shared string, which is a global variable shared by all threads. You can initialize the shared string in any way. For demonstration, let us say we set it to "All work and no play makes Jack a dull boy."

The main() function will do the initialization to initialize the semaphores (which are also shared by all threads). It also creates the separate reader and writer threads, with number of threads based on command line arguments. Once it has created the threads, the main() function will wait for the threads to finish (using pthread_join()).

A skeleton for main() is:

```
int main(int argc, char *argv[])
{
    /*  Get command line arguments.
        Initialization of semaphores.
        Create reader and writer threads.
        Wait for reader threads to finish.
        Wait for writer threads to finish.
        Cleanup and exit.
    */
}
```

The writer thread will alternate between writing and sleeping for 1 second. In each iteration, it modifies the current contents of the string by chopping the last character of it, until it is an empty string. E.g.: if the current string is "All work", the writer will change it to "All wor". It calls sleep(1) to sleep for 1 second between iterations. A skeleton for writer is:

```
void *writer(void *param)
{
    //some local variables
    //loop until the string is empty:
    {
            //print out a message saying that it is writing
            //writing (chopping the last character of the string)

            //sleep for 1 second
    }
}
```

The above skeleton does *NOT* yet have any synchronization logic. You need to add semaphores at the correct places.

For part 1, the structures of the reader and the writers are the same as described in the lecture and textbook, which uses two semaphores, and a *readcount* to keep track of readers. Make sure you call the related wait and signal/post functions in a correct order.

The reader thread also has a loop structure. It prints out the content of the shared string. In addition, it also prints out the current *readcount* value when the value increments or decrements. When the *readcount* is 0, the reader calls sem_post() on one semaphore (e.g. the *rw_sem* as described below) to signal the writer.

If the string becomes empty, let the readers and writers break out of the loop and exit.

PART 2 (EXTRA CREDIT)

Part 2 is a variant of the classic reader-writer problem. Part 1's specification does not require strict alternating of writers and readers. As a result, some updates of the writers may be missed by readers. And

some contents may be read more than once. Part 2's setup, on the other hand, will result in strict alternating between writers and reader.

In Part 2, we assume that we will have many writers, and one reader.

The one reader reads from the shared string and prints the contents to the standard output. Several writers write to the same shared string. Only one writer can write to the string at any time. And once any writer writes, the reader should display it before other writers change the contents. In some sense, the reader is like a *display server* that displays the contents of the buffer.

Using two semaphores, we can achieve both the mutual exclusion as well as the synchronization between the reader and writers. Here is how it will be done:

The code (main() function) initializes the first semaphore to 1 and the second semaphore to 0. In a loop, the writer performs a *WAIT* operation on the first semaphore before it writes to the shared string. After writing, it performs a *SIGNAL* operation on the second semaphore, to notify the reader about a finished writing. The reader executes the following loop: it performs a *WAIT* operation on the second semaphore, reads contents of the shared string and displays it, then performs a *SIGNAL* operation on the first semaphore, so that one of the waiting writers can proceed.

The loop structure of the reader and writers are similar as Part 1. If the string becomes empty, let the readers and writers break out of the loop and exit.

While Part 2 allows the alternating behavior of the reader and writers, it has a new issue: since there are more writers than readers and they are strictly alternating, there will be writers waiting on a first semaphore at the end due to the alternating nature. To solve this, we can let the reader signal all the waiting writers before it exits by calling sem_post() multiple times. Note that we know the number of waiting writers is *the number of total writers minus the number of total readers*.

(You can also think about what if there are more readers than writers. In other words, what if we lift the assumption of having only one reader. Not required for the assignment.)

Pthread semaphores

For both Part 1 and Part 2, you will need two semaphores for thread synchronization/mutual exclusion. Their initial values are different.

Use Part 1 semaphores as an example, assume you have two semaphores as below (You can name them differently).

```
sem_t  rw_sem; //used by both readers and writers
sem_t cs_sem;  //used for protecting critical sections of readers
```

You can use the following to initialize both to 1 (assuming unnamed semaphore):

```
sem_init(&rw_sem, 0, 1);
sem_init(&cs_sym, 0, 1);
```

You must check the return value in your program to know if the operation was successful.

PThread semaphore is not supported on all platforms since it is just a specification (e.g. Mac OS X does not provide implementation of PThread unnamed semaphore). Although it is supported on turing,

checking return values is an important practice in general for such operations. You can use perror() or strerror() to get the details of the error if the operation fails. Check the related manual for usage.

Cleanup

You need to clean up the resources in main() by destroying the semaphore which can be done by calling sem_destroy().

When printing to the standard output, use printf(), or prebuild the entire string before printing, to avoid confusing interleaving of outputs among concurrent threads. You can optionally use fflush(stdout) after the call of printf() to flush the output buffer.

Output

Your program will be run in a way like this:

Part 1:
$ ./z1234567_project4 10 3

Part 2:
$ ./z1234567_project4_p2 1 10

You output will look something like below:

Sample output from the reader (Other printouts such as readcount, if applicable, or printouts from the writers are omitted):

All work and no play makes Jack a dull boy.
All work and no play makes Jack a dull boy
All work and no play makes Jack a dull bo
All work and no play makes Jack a dull b
All work and no play makes Jack a dull
All work and no play makes Jack a dull
All work and no play makes Jack a dul
All work and no play makes Jack a du
All work and no play makes Jack a d
All work and no play makes Jack a
All work and no play makes Jack a
All work and no play makes Jack
All work and no play makes Jack
All work and no play makes Jac
All work and no play makes Ja
All work and no play makes J
All work and no play makes
All work and no play makes
All work and no play make
All work and no play mak
All work and no play ma
All work and no play m
All work and no play
All work and no play
All work and no pla
All work and no pl
All work and no
All work and no
All work and n
All work and

All work and
All work an
All work a
All work
All work
All wor
All wo
All w
All
All
Al
A

For Part 1, if there is one reader and one writer, the output from the reader can look similar as above but the output does not need to be exactly the same since the reader thread and the writer thread does not have to be strictly alternating.

You will see different outputs with different numbers of readers and writers. Try 10 readers, 3 writers (Or 50 readers, 10 writers.) You can see some interesting behavior – for example, with more readers, you may see that the readers will keep reading for a while with the reader count being incremented before the writers get to write. Some possible outputs are provided, but your output may vary since the order of printout is based on CPU scheduling. Note that the read count is in the critical section, so its value should always be consistent and should never be negative. It should not suddenly increase or decrease more than one.

For Part 2 extra credit, the output from the reader should look exactly as sample output.

Requirements

The programs should 1) work according to the specifications; 2) be comprehensible and well commented; 3) check error conditions and have proper error handling.

Submission

Submission requirement is the similar as previous assignments with the following exception:

In your Makefile, if you do not do part 2, you only need one executable output "your-zid_project4". On the other hand, if you do both parts, then you need to make sure your compilation produces TWO executable files called "your-zid_project4" and "your-zid_project4_p2". For a student with z1234567 as their zid, the executable would be *z1234567_project4 and z1234567_project4_p2*.

Note that the directory must be called: "z1234567_project4_dir", all in lowercase.

Notify your TA that you've done the extra credit part by adding a line of text when submitting your assignment in Blackboard ("I did extra credit.") so that s/he will do the corresponding testing and grading.