

**Lab Task:**

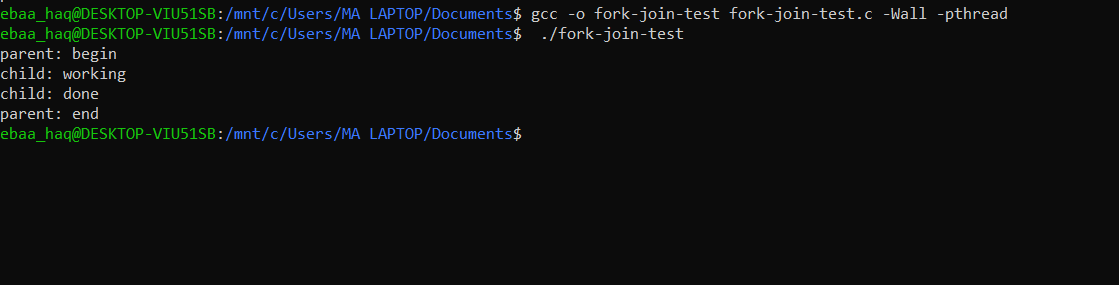
In this homework, we’ll use semaphores to solve some well-known concurrency problems. Many of these are taken from Downey’s excellent “Little Book of Semaphores”, which does a good job of pulling together a number of classic problems as well as introducing a few new variants; interested readers should check out the Little Book for more fun. Each of thefollowing questions provides a code skeleton; your job is to fill in the code to make it work given semaphores. Read the README for details.

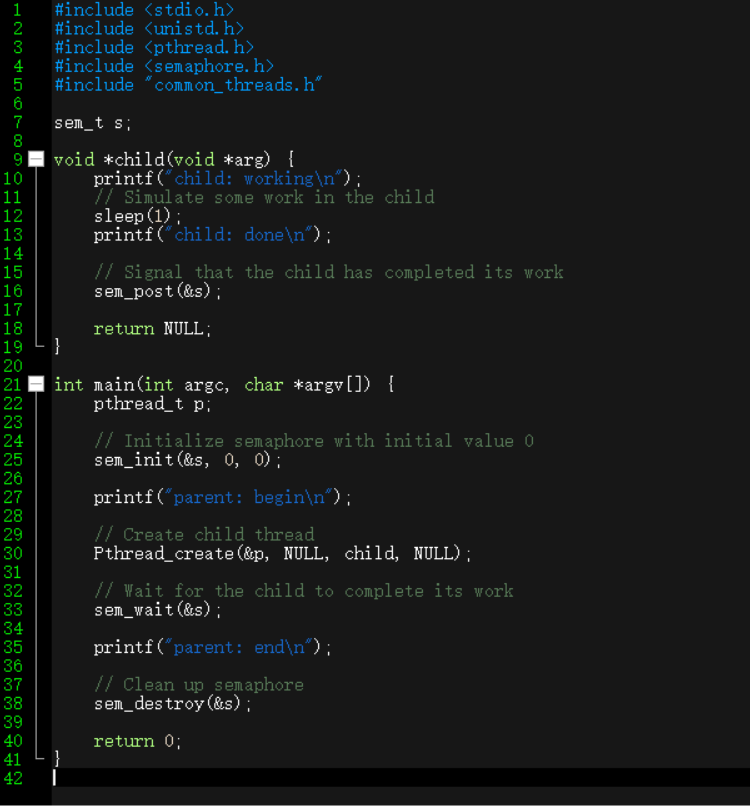
**Tasks:**

1. The first problem is just to implement and test a solution to the fork/join problem, as described in the text. Even though this solution is described in the text, the act of typing it in on your own is worthwhile; even Bach would rewrite Vivaldi, allowing one soon-to-be master to learn from an existing one. See fork-join.c for details. Add the call sleep(1) to the child to ensure it is working.

**Solution:**

Firstly run this command: ./fork-join-test



Solution of fork-join.c is:

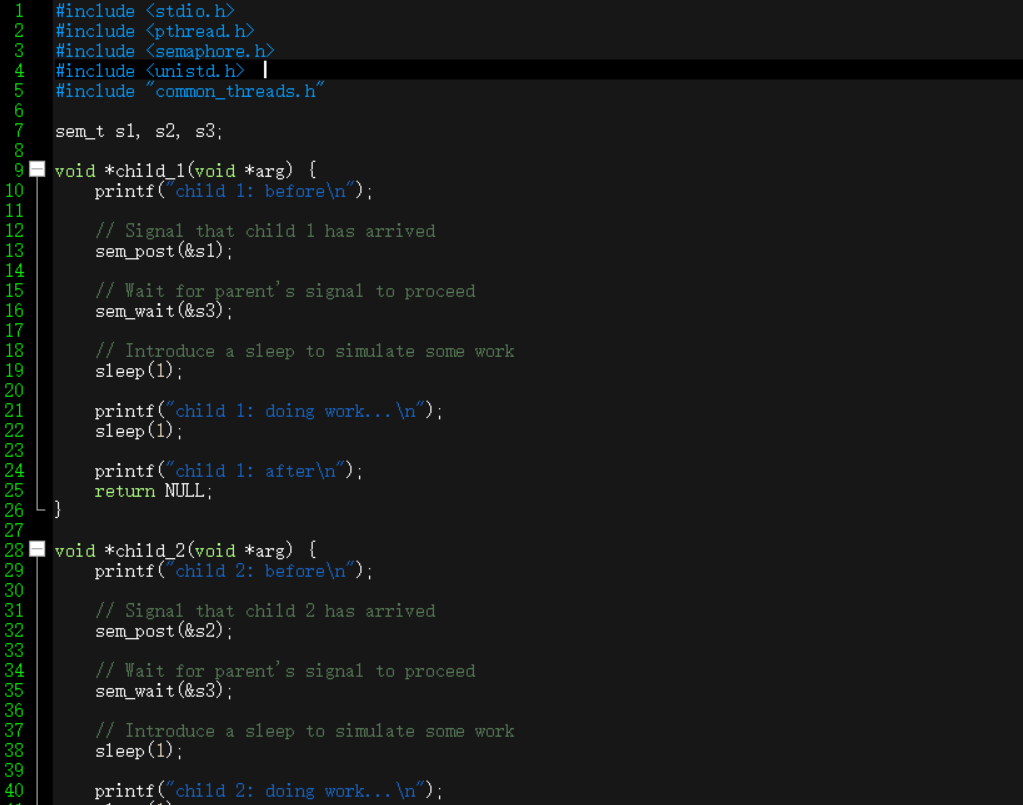
The output to ensure that the program behaves as expected. You should see "child: working," a delay of 1 second, "child: done," and finally "parent: end."

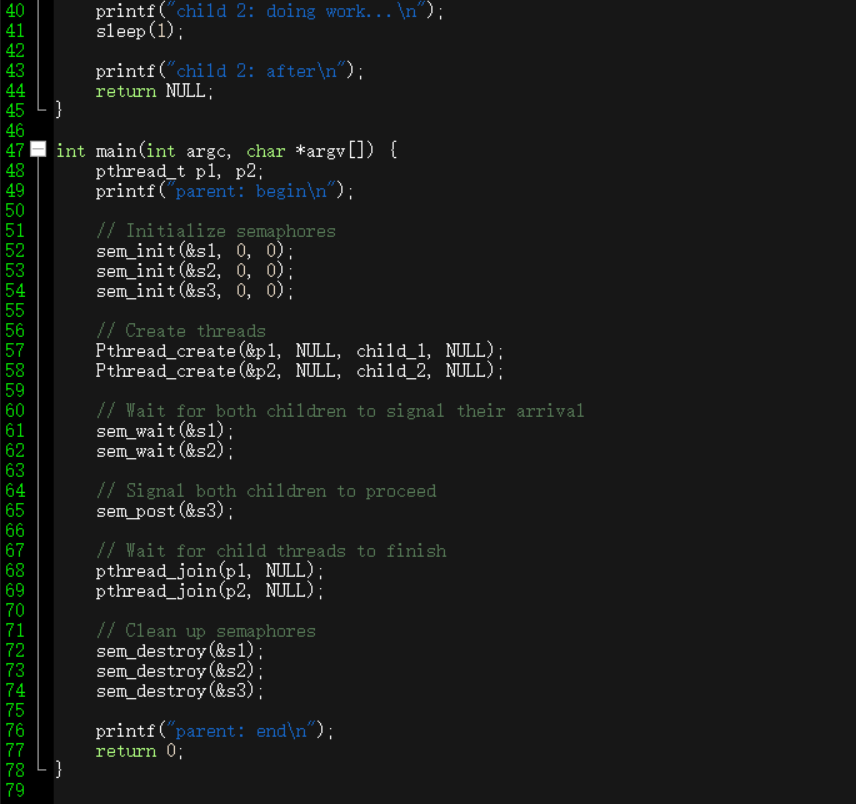
1. Let’s now generalize this a bit by investigating the rendezvous problem. The problem is as follows: you have two threads, each of which are about to enter the rendezvous point in the code. Neither should exit this part of the code before the other enters it. Consider using two semaphores for this task, and see rendezvous.c for details.

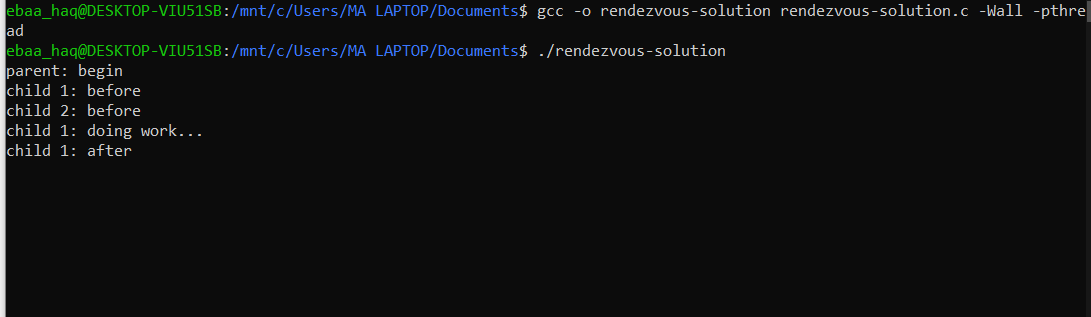
**Solution:**

Solution of rendezvous.c is

* gcc -o rendezvous-solution rendezvous-solution.c -Wall –pthread
* ./rendezvous-solution





****

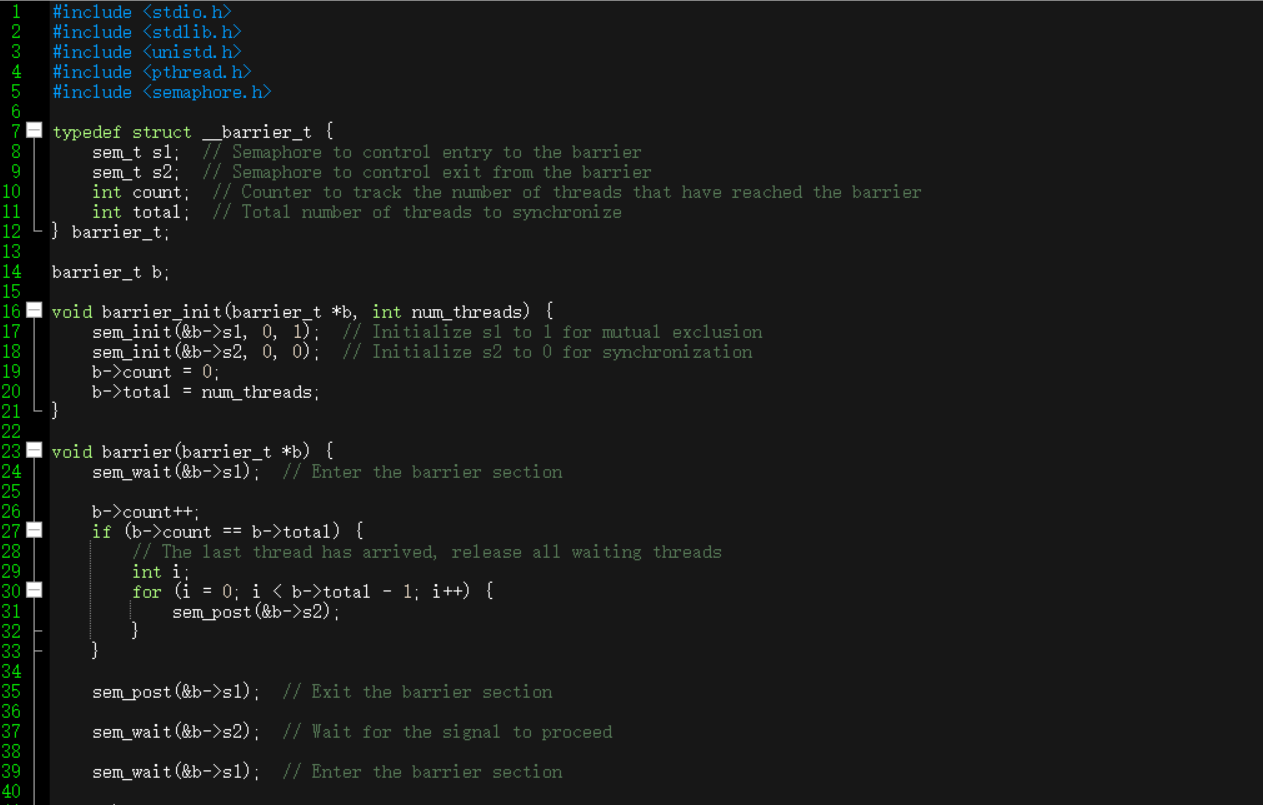
The program is running without any errors, and the output indicates the expected behavior for the rendezvous problem. The output shows that "child 1: before" and "child 2: before" are printed before "child 1: doing work..." and "child 1: after." This indicates that the synchronization between the threads is working as intended.

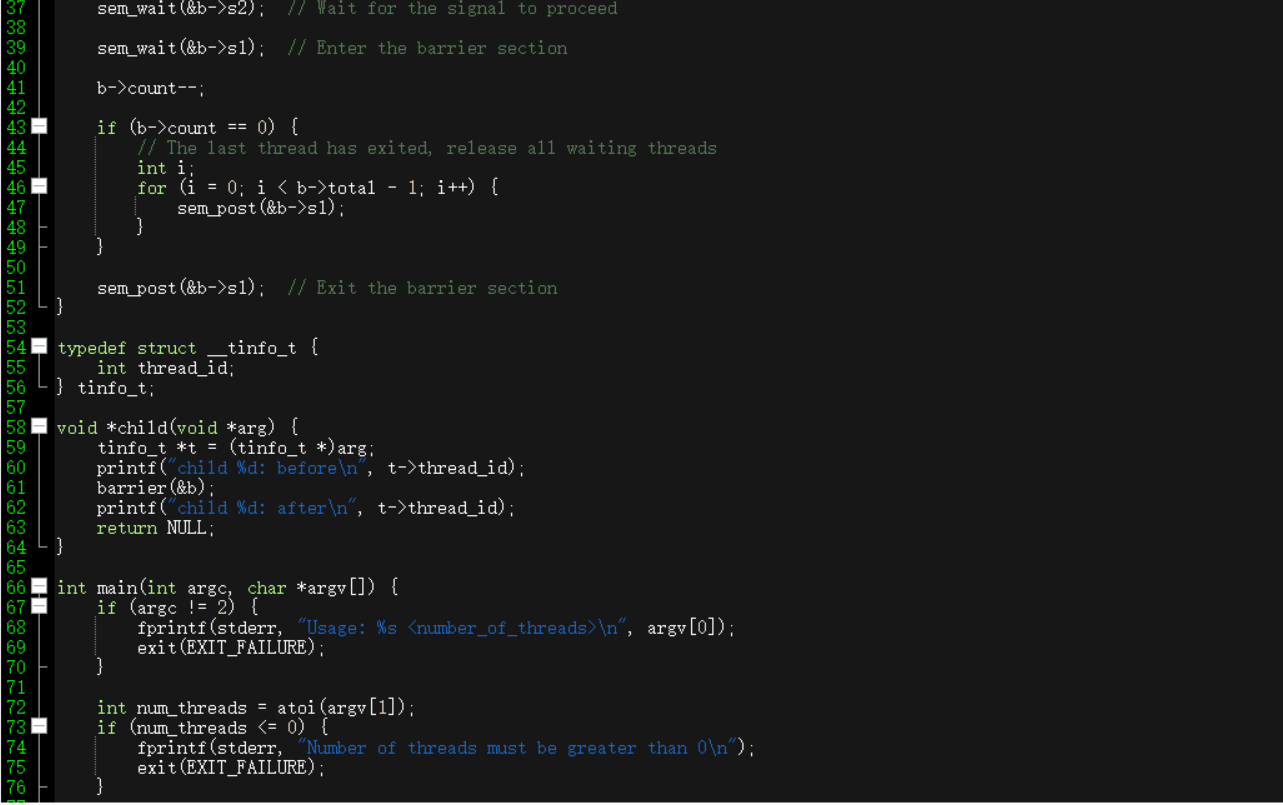
1. Now go one step further by implementing a general solution to barrier synchronization. Assume there are two points in a sequential piece of code, called P1 and P2. Putting a barrier between P1 and P2 guarantees that all threads will execute P1 before any one thread executes P2. Your task: write the code to implement a barrier() function that can be used in this manner. It is safe to assume you know N (the total number of threads in the running program) and that all N threads will try to enter the barrier. Again, you should likely use two semaphores to achieve the solution, and some other integers to count things. See barrier.c for details.

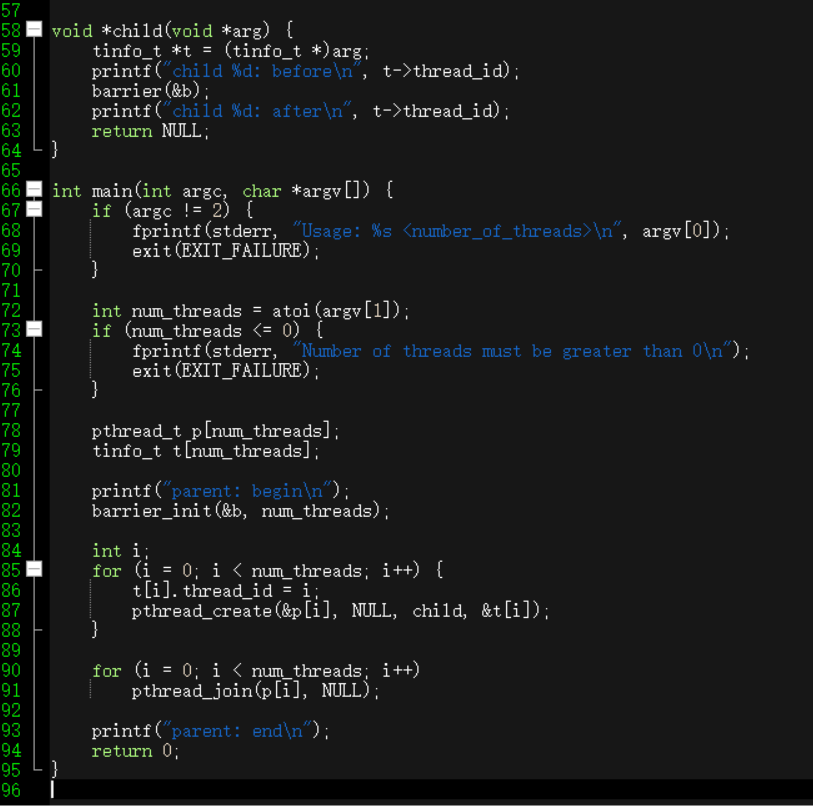
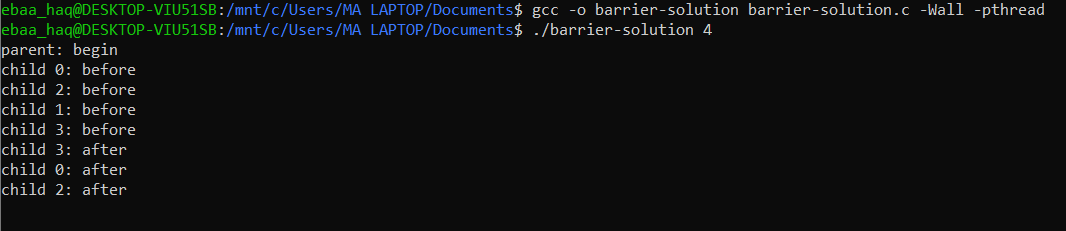
**Solution:**

Solution of barrier.c is:

* gcc -o rendezvous-solution rendezvous-solution.c -Wall –pthread
* ./rendezvous-solution

****

****

****

The output indicates that each child thread prints "before" when it reaches the barrier and "after" when it exits the barrier. The order of "before" and "after" messages is consistent with the barrier synchronization, ensuring that all threads reach the barrier before any one of them proceeds beyond it.

Here's a breakdown of the output:

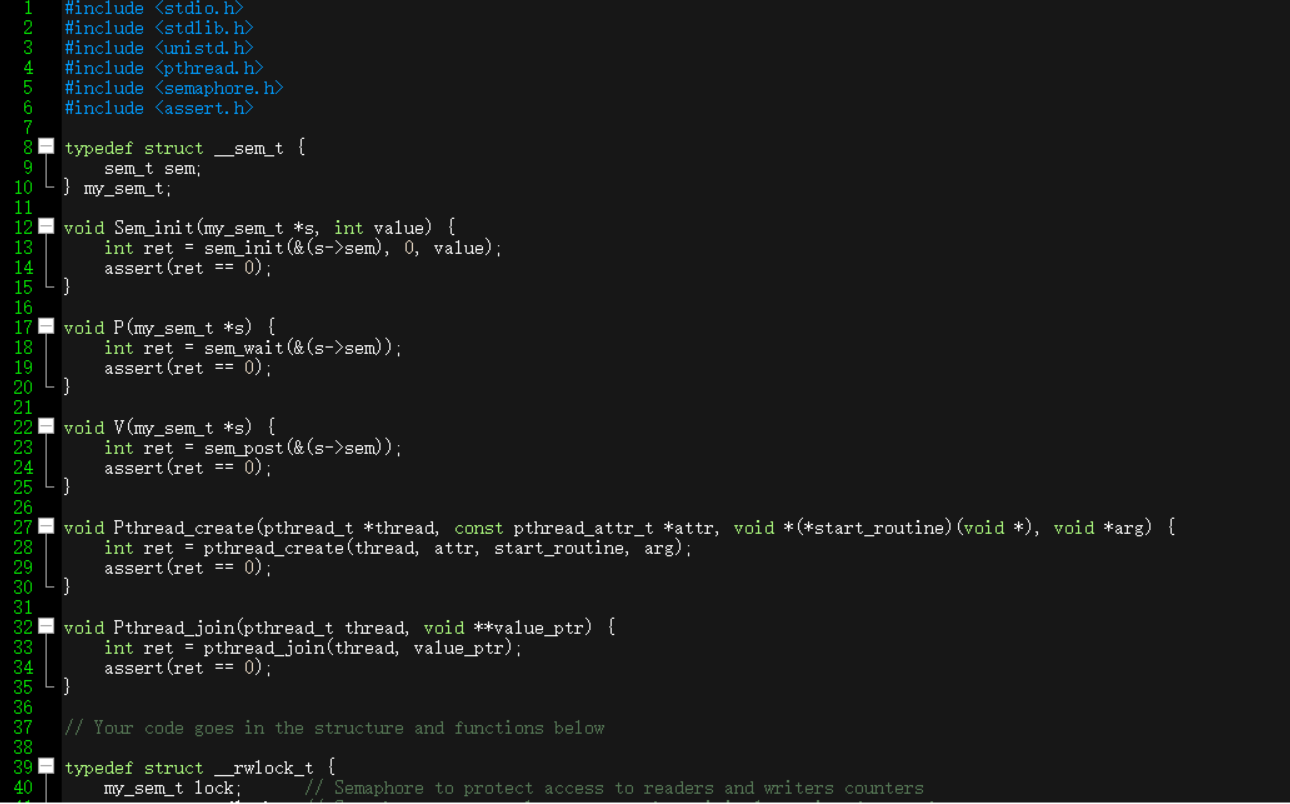
* All child threads (0, 1, 2, 3) reach the barrier.
* The last child thread (3) reaches the barrier, and since all threads have arrived, it releases all waiting threads.
* Child threads 0, 1, and 2 proceed beyond the barrier while Child thread 3 waits for the signal to proceed.
* Child thread 3 receives the signal and proceeds beyond the barrier.

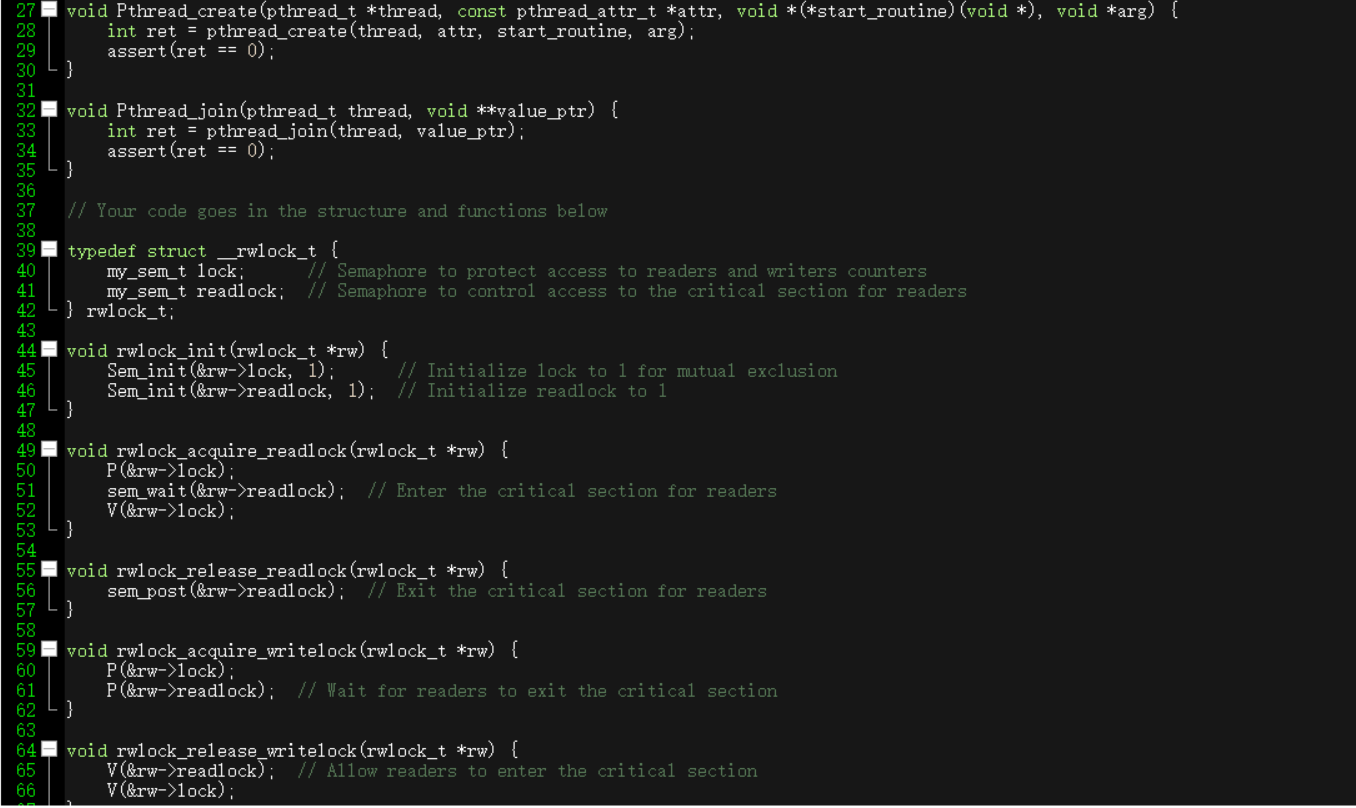
1. Now let’s solve the reader-writer problem, also as described in the text. In this first take, don’t worry about starvation. See the code in reader-writer.c for details. Add sleep() calls to your code to demonstrate it works as you expect. Can you show the existence of the starvation problem?

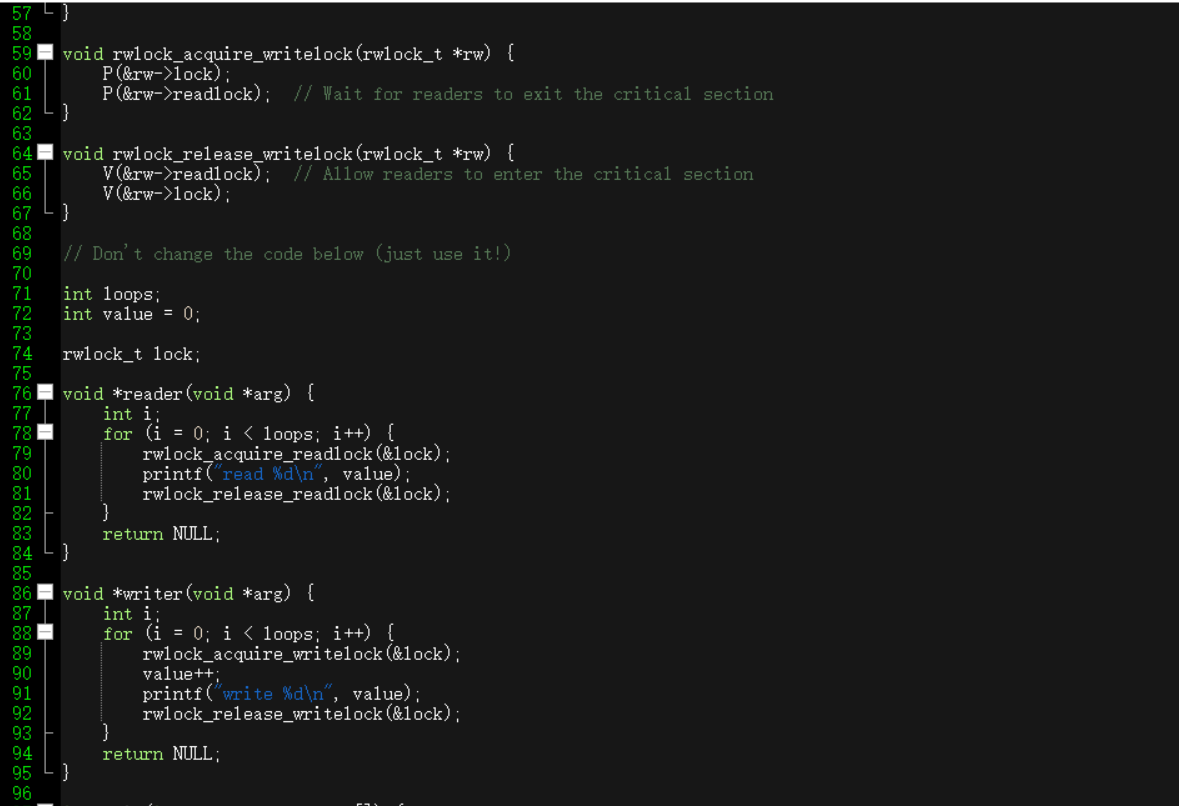
**Solution:**

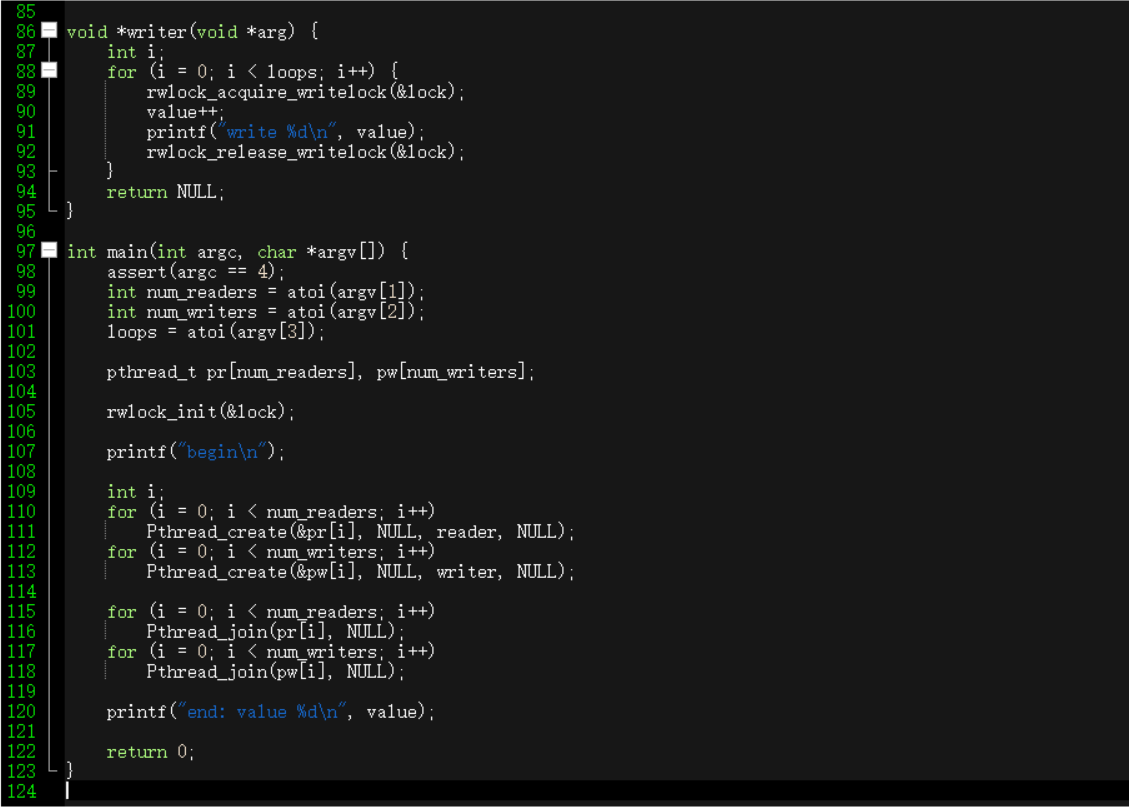
Solution of reader-writer.c is:

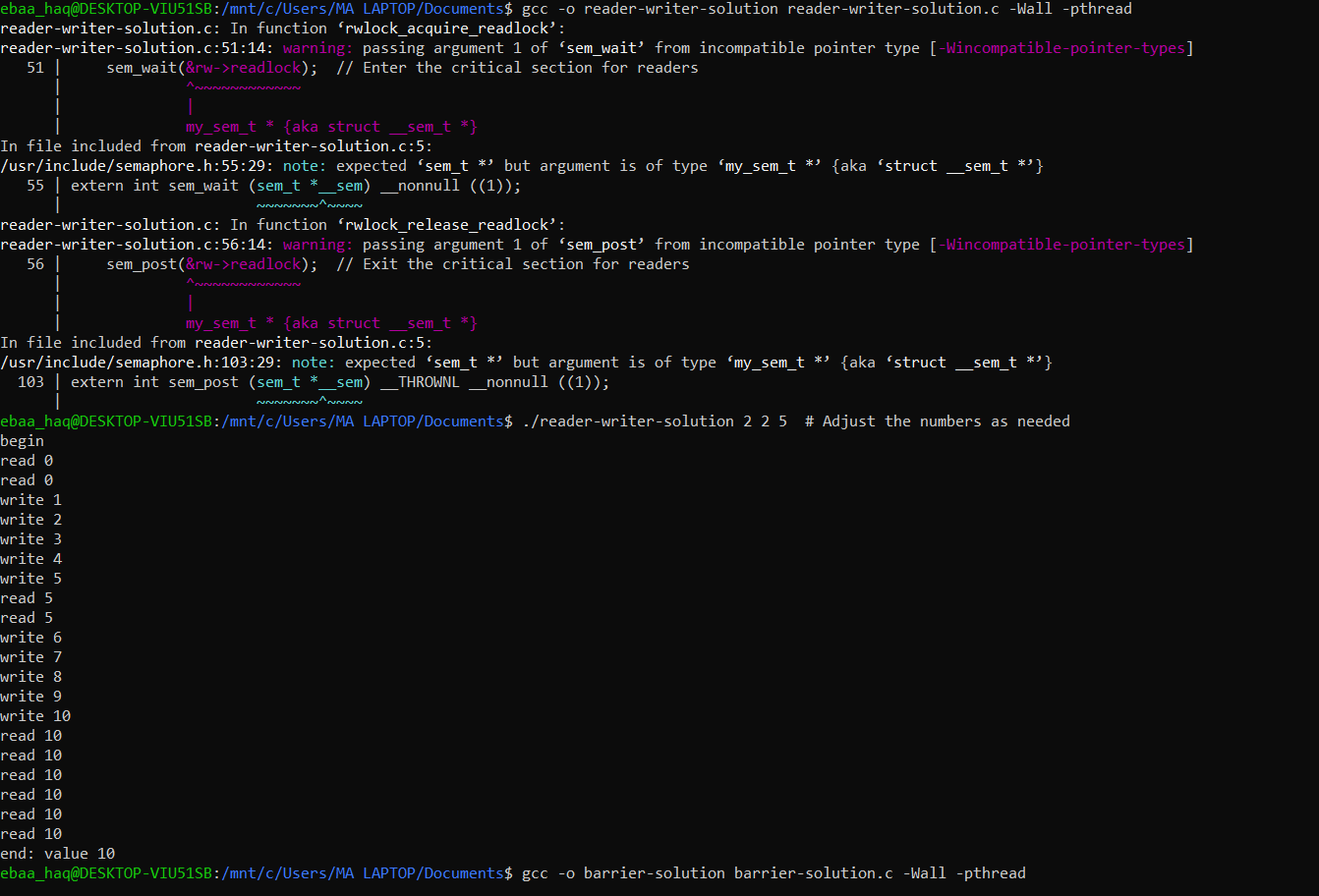
* gcc -o reader-writer-solution reader-writer-solution.c -Wall -pthread
* ./reader-writer-solution 225 # Adjust the numbers as needed

****

****

****

****

****

The output of your program demonstrates the reader-writer problem. Here's a brief explanation:

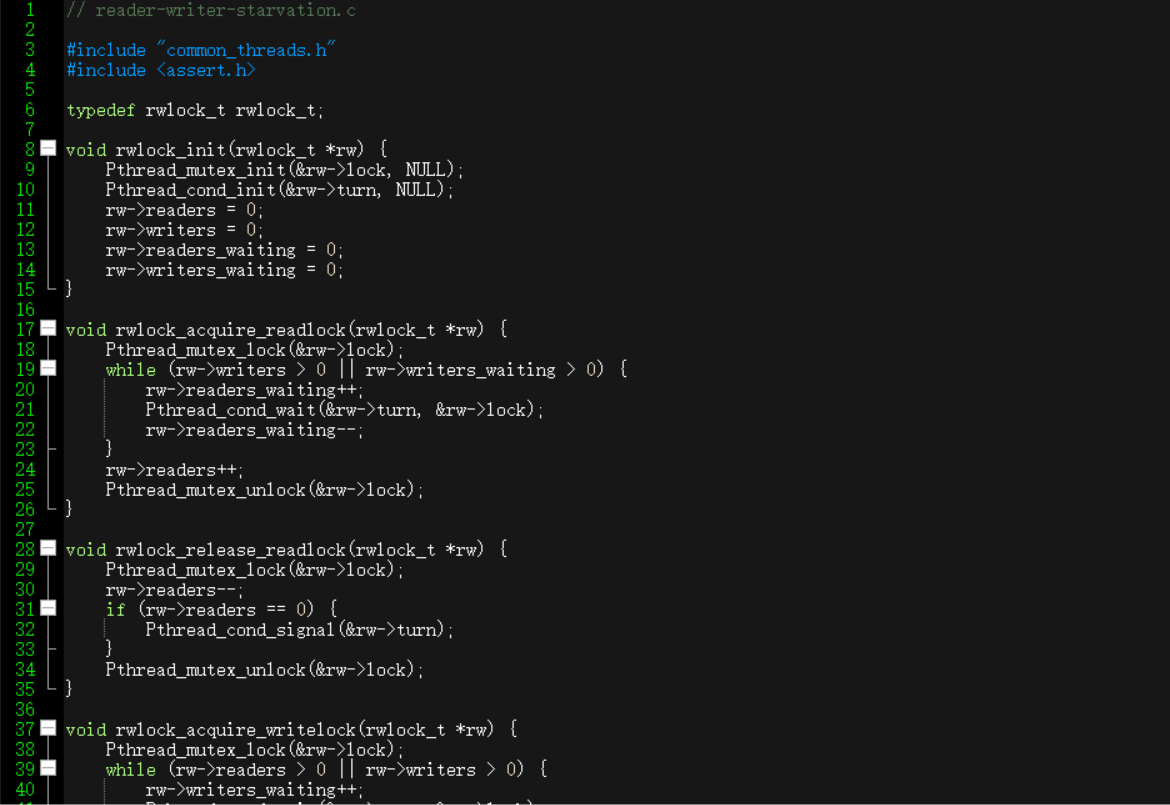
* The program starts with begin and creates the specified number of reader and writer threads.
* Readers and writers access a shared variable value while using a reader-writer lock to control access.
* The interleaved output shows the operations performed by readers (read) and writers (write).
* Readers can access the critical section simultaneously, demonstrating multiple readers allowed at the same time.
* Writers access the critical section exclusively, ensuring that only one writer can modify the shared variable at a time.
* The program ends with end: value 10, indicating the final value of the shared variable after all threads have completed.

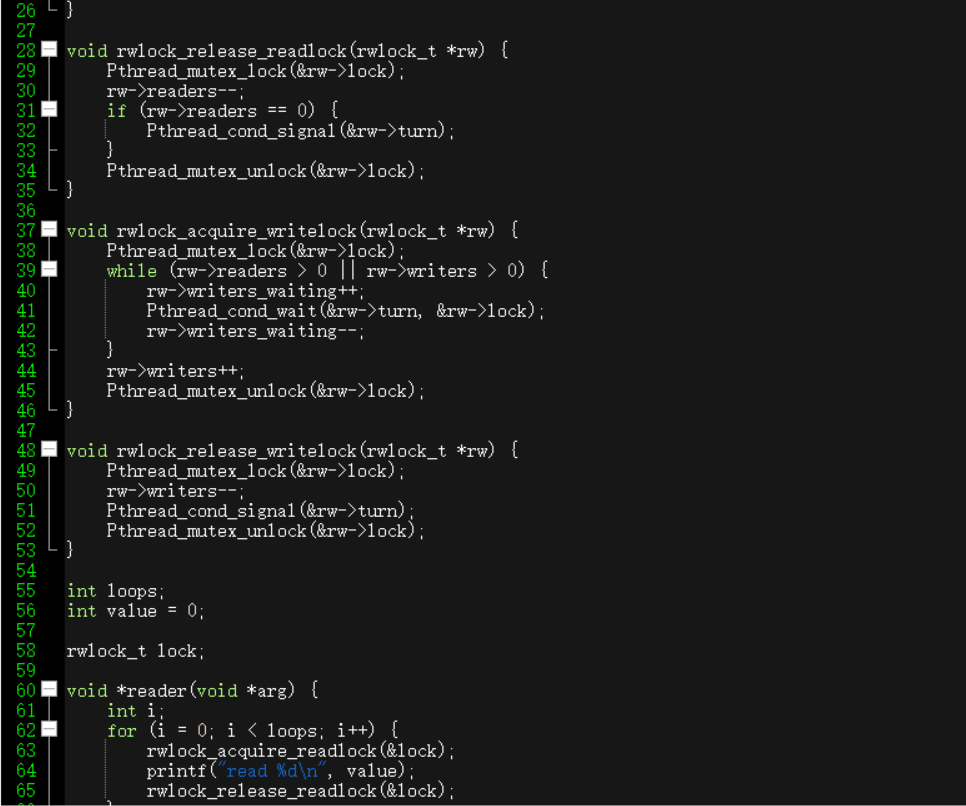
1. Let’s look at the reader-writer problem again, but this time, worry about starvation. How can you ensure that all readers and writers eventually make progress? See reader-writer-nostarve.c for details.

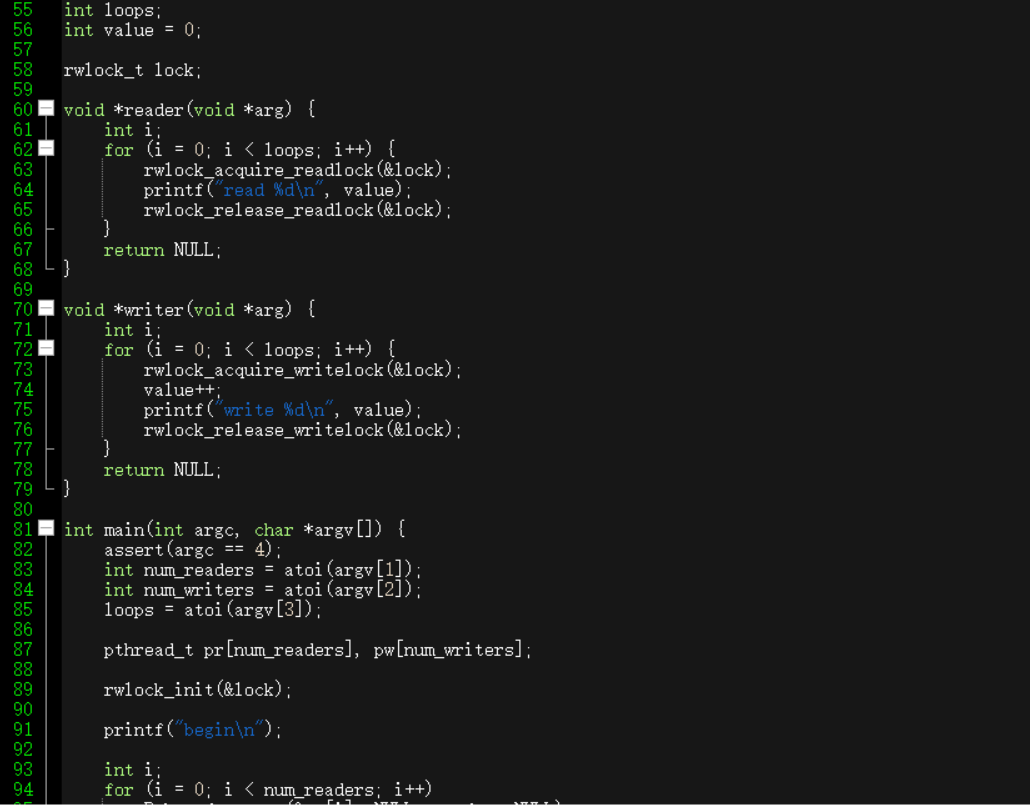
**Solution:**

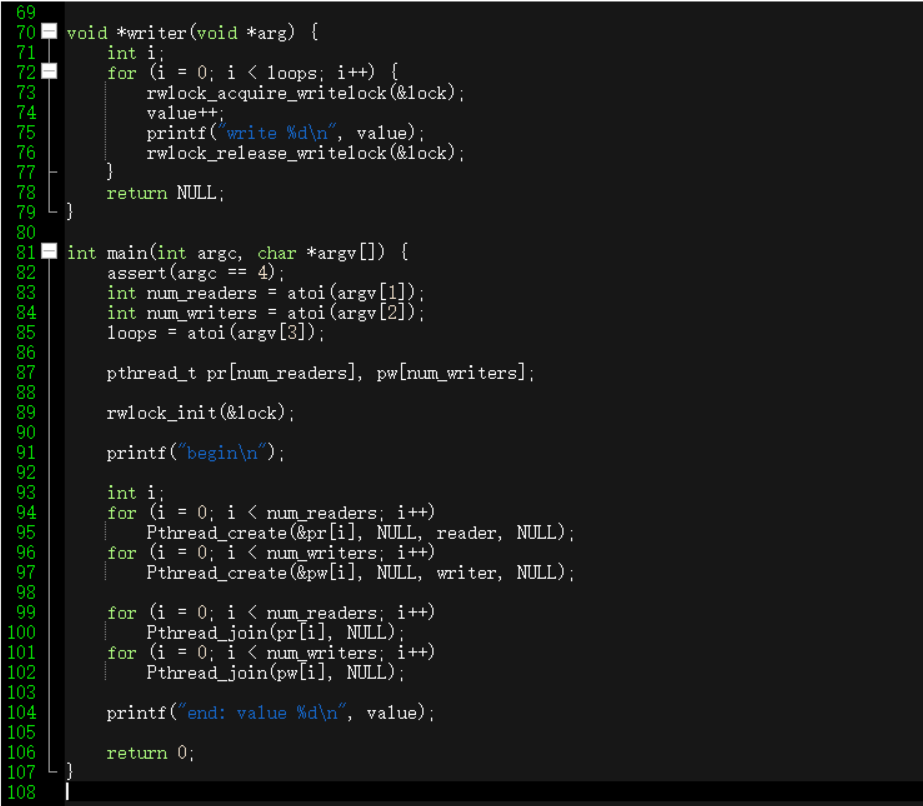
Solution for Starvation is:

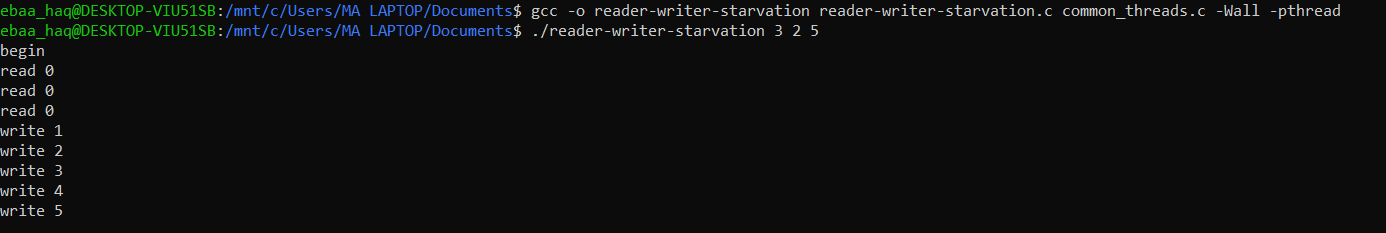
* gcc -o reader-writer-starvation reader-writer-starvation.ccommon\_threads.c -Wall –pthread
* ./reader-writer-starvation 325
* gcc -o reader-writer-nostarve reader-writer-nostarve.ccommon\_threads.c -Wall -pthread
* ./reader-writer-nostarve 3 2 5



****

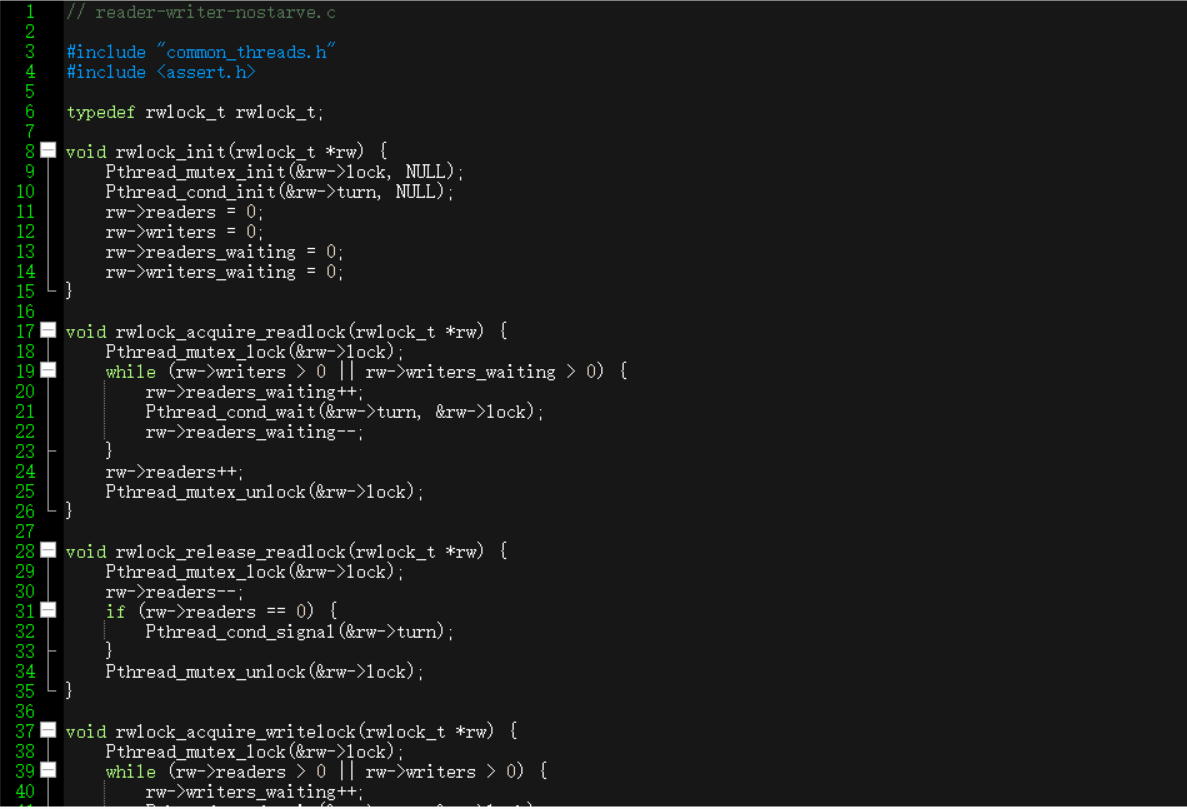
****

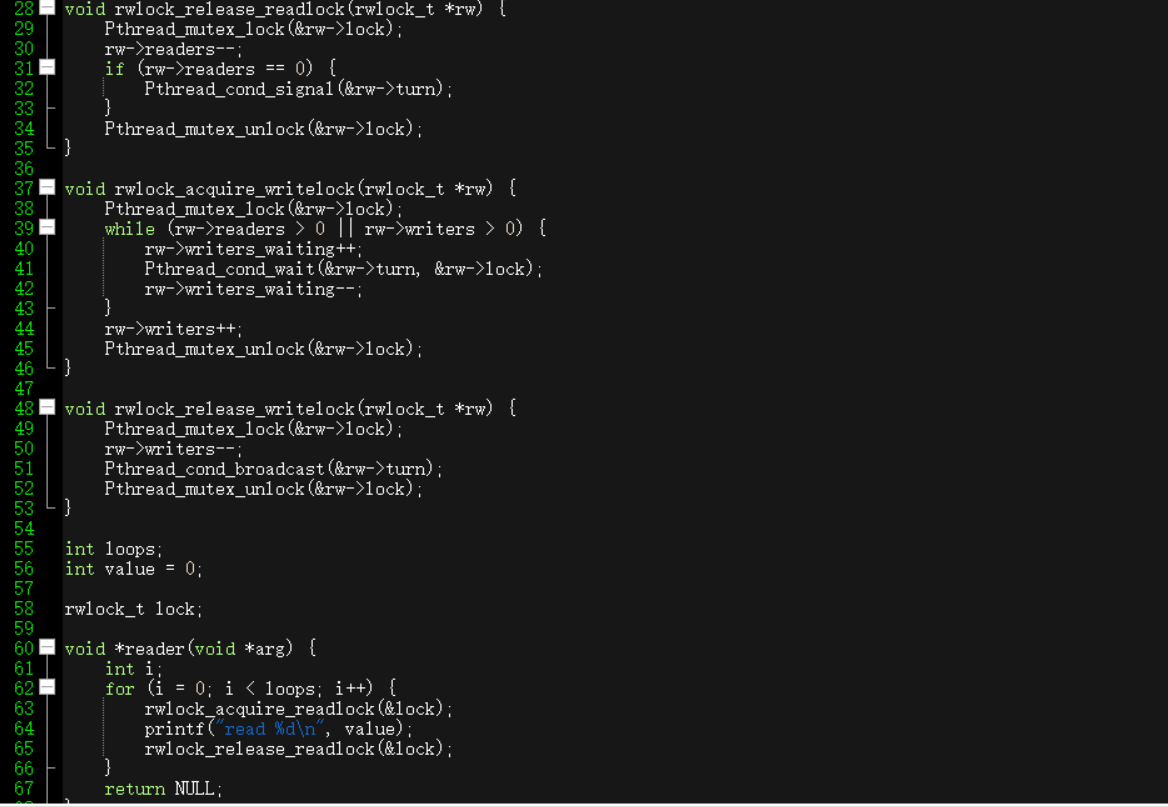
****

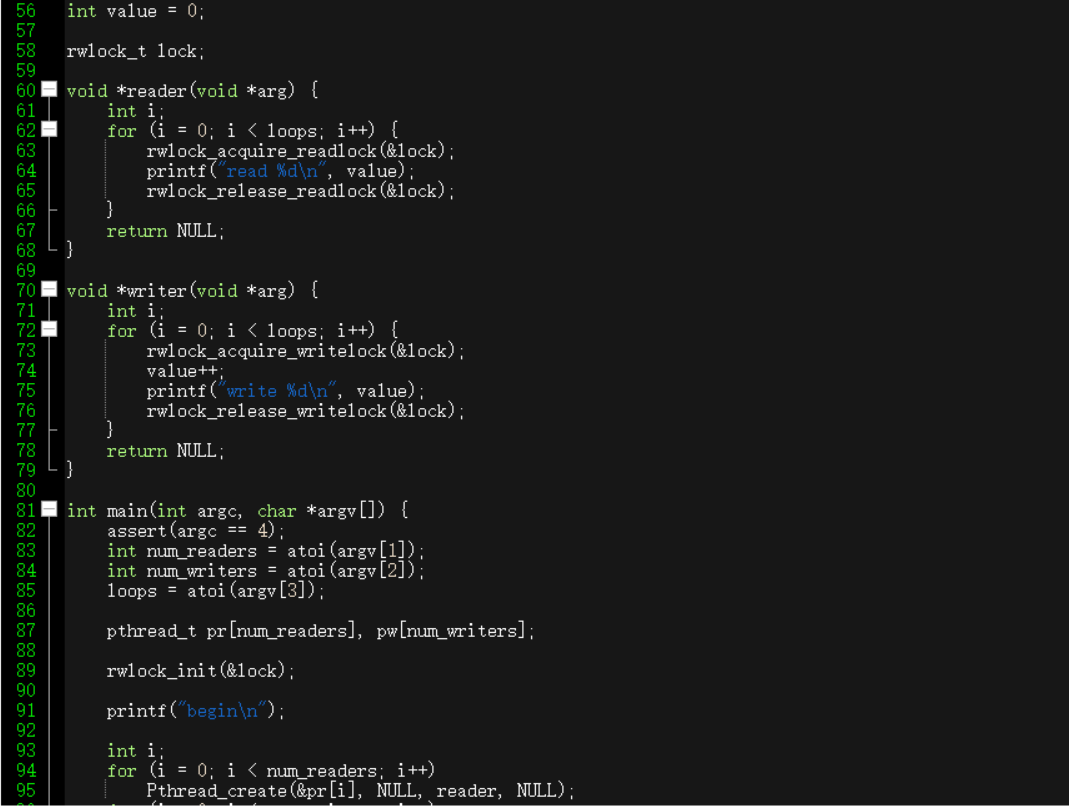
****It appears that the reader-writer-starvation program executed with 3 readers, 2 writers, and 5 loops. Here's a brief explanation of the output:

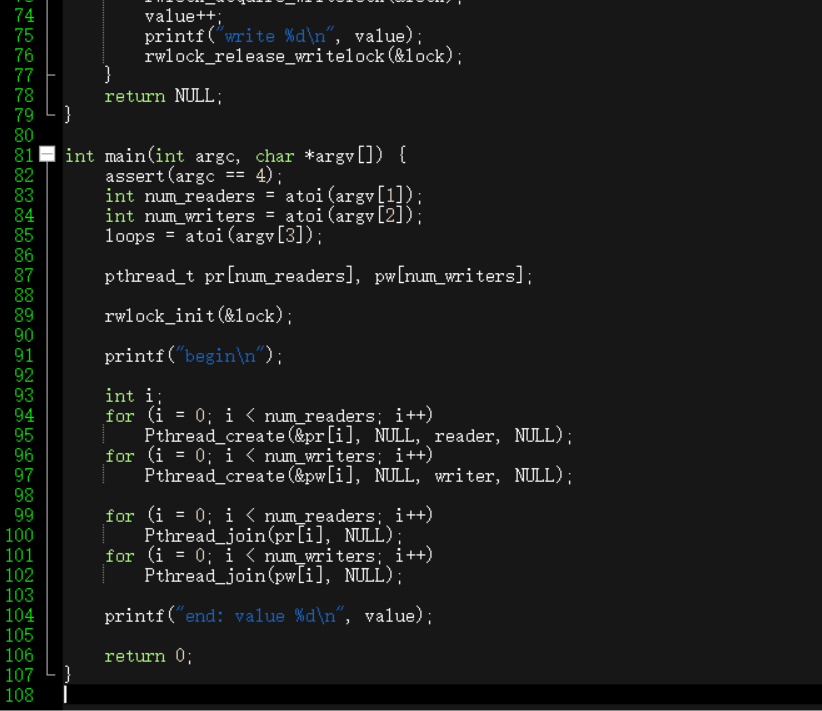
1. Read Operations:
   * Initially, all read operations start with the value 0.
   * The readers are repeatedly trying to acquire the read lock, but it seems that they are unable to make progress, as evident from the repeated "read 0" outputs.
2. Write Operations:
   * The writers increment the shared value, and you see the sequence of increasing values.
3. End Result:
   * The program seems to be stuck in a state where readers are unable to acquire the read lock, leading to a potential starvation issue.

**Solution for reader-writer-nostarve is:**

****

****

****

****

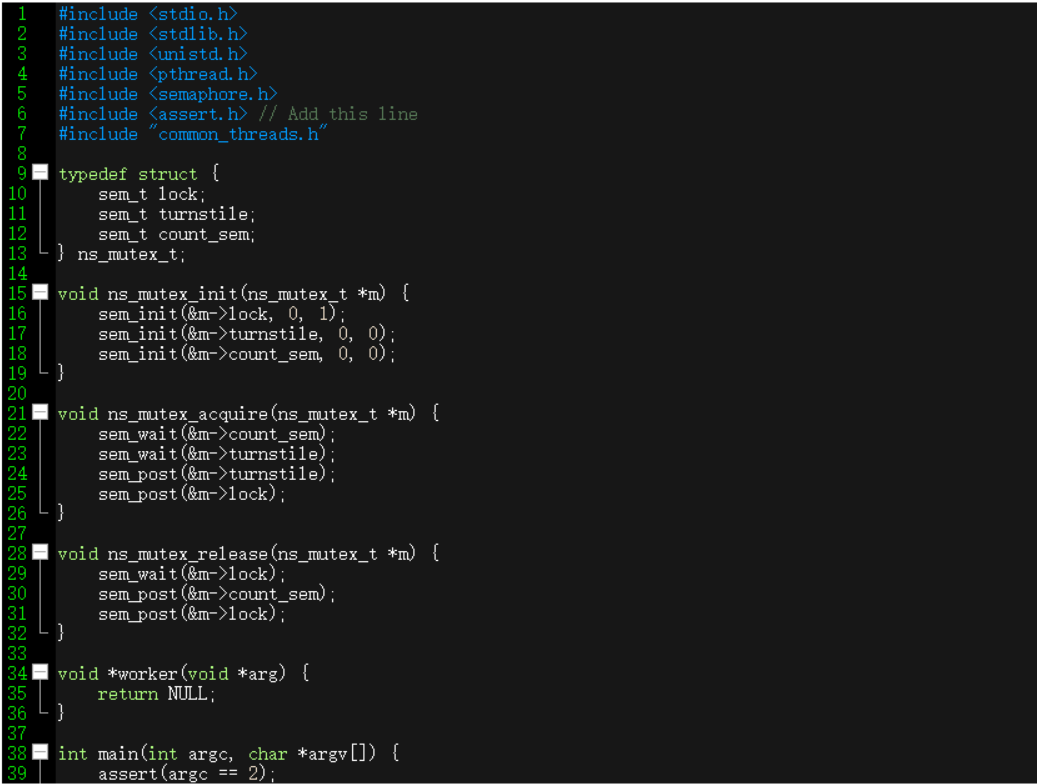
****

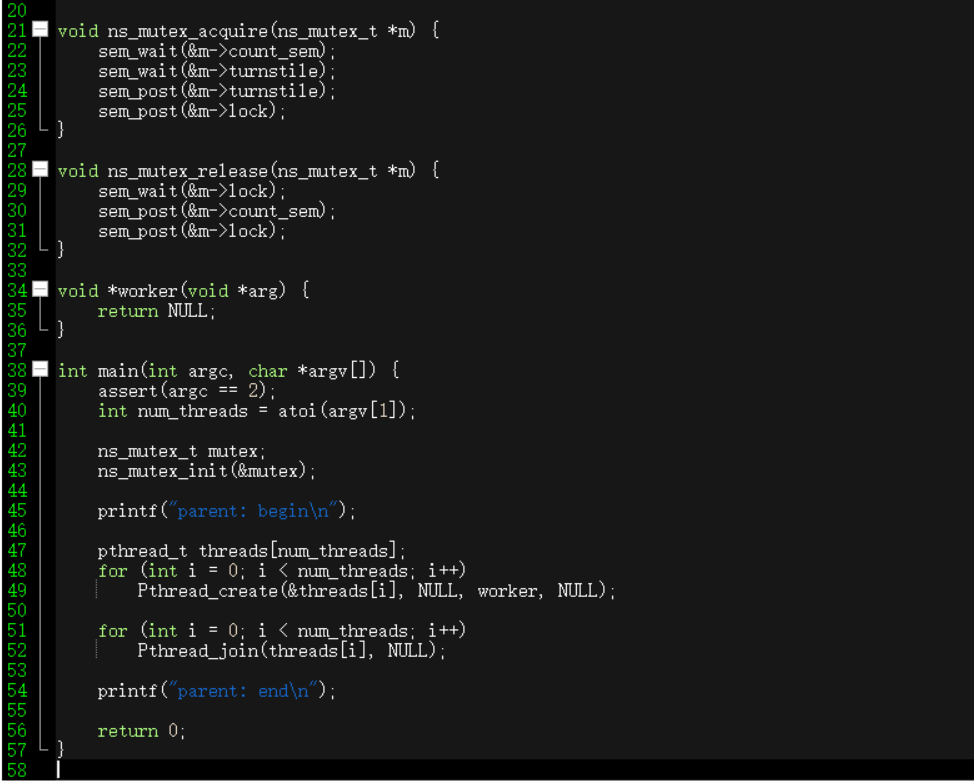
It looks like the reader-writer-nostarve program has executed with 3 readers, 2 writers, and 5 loops. Here's a brief explanation of the output:

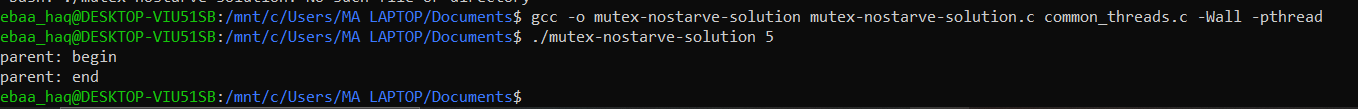
1. Read Operations:
   * Initially, all read operations start with the value 0.
   * As writers increment the value, readers observe the updated value.
2. Write Operations:
   * The writers increment the shared value, and you see the sequence of increasing values.
   * The write operations happen without being blocked by readers, indicating a lack of starvation.
3. End Result:
   * The program concludes by printing the final value of 10 after all threads have completed their operations.
4. Use semaphores to build a no-starve mutex, in which any thread that tries to acquire the mutex will eventually obtain it. See the code in mutex-nostarve.c for more information?

**Solution:**

* gcc gcc -o mutex-nostarve-solution mutex-nostarve-solution.c common\_threads.c -Wall -pthread
* ./mutex-nostarve-solution 5

****

****

****

The output shows, "parent: begin" and "parent: end," indicates that the program ran successfully. The exact output may vary depending on the content of the program and the input parameters provided.