opERating systems assignment

Chang, Christopher Hsu

ID: 18821354

Curtin University

Operating Systems – COMP2006

Table of Contents

[Introduction: 3](#_Toc513439189)

[Process SDS: 3](#_Toc513439190)

[Thread SDS: 6](#_Toc513439191)

[Testing: 7](#_Toc513439192)

[Errors: 8](#_Toc513439193)

[Side Note: 8](#_Toc513439194)

[Input Files: 8](#_Toc513439195)

[Expected Results and Actual Results: 8](#_Toc513439196)

[Code: 13](#_Toc513439197)

# **Introduction:**

The following report is for the Operating Systems Assignment for 2018. It will discuss about how mutual exclusion has been achieved for processes and threads in shared memory for the *First Reader’s and Writer’s Problem*. The report will also detail the available test cases I have used to ensure that my program works correctly with the synchronization between different child processes.

# **Process SDS:**

For the process implementation of the First *Reader’s Writers Problem*, mutual exclusion is achieved by having the writer child process wait for all of the readers that are currently executing in their remainder sections (The Reading Part for readers), also by having the reader child processes waiting for a single writer to finish writing and leave its critical section so the readers are allowed to begin reading again since reader’s have priority over writers. This waiting is implemented by using semaphores stored in an array and passing that whole array into shared memory.

This waiting is implemented by using semaphores stored in an array and passing that whole array into shared memory. For the readers the critical section happens when you are trying to update the **“readcount”** shared memory variable. This ensures that as long as you have one reader reading, other readers are allowed to come and read. It isn’t until the last reader finishes reading, the semaphore that protects **“readcount”** will need to be checked if there are no more readers so it is allowed to unlock the lock for writers to begin writing. In the program the semaphore that refers to this is called ***“mutex\_lock”*** which is at index 1 of the semaphore array in the program.

sem\_wait(&semaphores[1]); /\* Ensure mutual exclusion on readcount \*/

(\*(readcount))++;

if( (\*(readcount)) == 1 )

{

/\* rw\_mutex to see if any writing is being performed \*/

sem\_wait(&semaphores[0]);

}

// Unlock so readers can read

sem\_post(&semaphores[1]);

Figure 2.1: Checking if there is at least one reader reading

// Lock the readcount variable

sem\_wait(&semaphores[1]);

(\*(readcount))--;

// When there are no more readers currently reading

if( (\*(readcount)) == 0)

{

// Unlock the rw\_mutex so another reader or writer may use it

sem\_post(&semaphores[0]);

}

sem\_post(&semaphores[1]); // Unlocks the mutex so you can read

Figure 2.2: Checking to see that there are no more readers

Next, there is the read/write semaphore which is used inside the “if” statement of the reader’s function and used at the start and the end of the writer’s function. This is used to do the controlling/business logic to prevent the writers from writing if there is at least one reader about to read the **“data\_buffer”** and to notify the reader’s when the writer has finished executing in its critical section. This semaphore is called **“rw\_mutex”** and it is indexed at element 0 in the array.

if( (\*(readcount)) == 1 )

{

/\* rw\_mutex to see if any writing is being performed \*/

sem\_wait(&semaphores[0]);

}

Figure 2.3: When there is one reader it should wait for the writer to finish writing

// When there are no more readers currently reading

if( (\*(readcount)) == 0)

{

// Unlock the rw\_mutex so another reader or writer may use it

sem\_post(&semaphores[0]);

}

Figure 2.4: When there are no readers reading, release the lock

Next, there is a **“trackerArray”** in shared memory that is used to determine how many readers still have to read a particular element of the array inside the **“data\_buffer”**. This needs to be mutually excluded to prevent a race condition for multiple readers from updating the **“trackerArray”** at the same time because the reading is done in the remainder section of the reader. The updating for the **“trackerArray”** doesn’t have to be mutually excluded for the **“writer()”** function because it happens when the writer is in its critical section. The semaphore that refers to this is called **“array\_track\_sem”** which is indexed at element 2 in the array.

sem\_wait(&semaphores[2]);

// Once read, decrement the number of readers left to read this value

trackerArray[j%BUFFER\_SIZE] = trackerArray[j%BUFFER\_SIZE] - 1;

sem\_post(&semaphores[2]);

Figure 2.5: Ensures mutual exclusion of the trackerArray

Finally, there is a semaphore that protects the write out to file function. This lock used as a precaution to the write out of file function so multiple writers or readers cannot access the **“writeOutFile”** function simultaneously because a writer/reader might finish it’s writing early while another one is trying to write out to file. The semaphore that referes to this is called **“write\_out\_sem”** and is indexed at element 3 in the array.

// Lock Writing out to file

sem\_wait(&semaphores[3]);

writeOutFile(getpid(), datacount, "reader-%d has finished reading %d pieces

of data from the data\_buffer\n");

sem\_post(&semaphores[3]);

Figure 2.6: Write out to file semaphore

The semaphores required, ***“trackerArray”*** and the ***“data\_buffer”*** were implemented using shared memory. The following POSIX shared memory functions were used to create shared memory, and their respective functions to close the shared memory:

shm\_open()

ftruncate()

mmap()

sem\_destroy()

sem\_unlink()

close()

munmap()

# **Thread SDS:**

In order to achieve mutual exclusion for a multi-threaded solution to the ***“Readers-Writers Problem”***, it uses all the above implementation of semaphores except as mutexes and they are refered as:

pthread\_mutex\_lock(&mutex);

pthread\_mutex\_unlock(&mutex);

Figure 3.1: For binary mutexes they use lock and unlock instead of

wait and post

There are also an extra conditional mutex that is used to signal writers. In the reader() function, there is a while loop that checks if the variable **“letWriters”** is positive because it wants a reader to grab the conditioned mutex and waits for the writer’s to finish writing so a reader can read. Essentially, when a writer arrives into the while loop, it releases the mutex and waits for the condition to be signalled which is done in the writer’s function as soon as the writer has finished reading to let all readers know that they are allowed to begin reading the data buffer. In this case a signal is not used, but instead a broadcast is because the program wants to be able to release the lock for all the readers.

// Reader grabs the mutex lock

pthread\_mutex\_lock(&mutex);

// When a reader wants to read come in

while(sharedMem->letWriters)

{

// Conditional Wait

pthread\_cond\_wait(&cond, &mutex); // Readers Busy Waiting

}

pthread\_mutex\_unlock(&mutex);

Figure 3.2: Conditioned wait in reader’s function

pthread\_mutex\_lock(&mutex);

sharedMem->letWriters--;

sharedMem->writing--;

pthread\_cond\_broadcast(&cond);

pthread\_mutex\_unlock(&mutex);

Figure 3.3: The end of the writers() function. When a writer is about to finish writing it decrements it’s count bringing it back to zero and broadcasts it back to the readers() function to allow readers to start reading.

For all of the threads, the shared resources are stored as global variables to ensure that the reader and writer functions can access them (for locking/unlocking) and incrementing/decrementing shared counters. In order to allocate memory for all the buffers malloc() was used and free() was used to clean up memory, this was used to ensure valgrind didn’t give any memory leaks. These functions were used in order to create and clean up the pthreads:

pthread\_mutex\_init()

pthread\_create()

pthread\_join()

pthread\_mutex\_destroy()

# **Testing:**

In order to test each implementations of the programs, multiple input files were used. In order to cover all test cases different input files and input arguments were used. The cases I tested for were:

* Standard Test Case
  + FILE\_SIZE = 100
  + BUFFER\_SIZE = 20

And for each of these cases I test for these command line arguments, The command line arguments are given in the format of (R W t1 t2) all separated by spaces:

* Standard Values given from Assignment Specification
  + 5 2 1 1
* Readers Greater than Writers (Same values as Standard Values)
  + 5 2 1 1
* Zero Sleep Times
  + 1 5 0 0
* Readers Less than Writers
  + 3 5 1 1
* No Readers or Writers:
  + 1 0 3 2
  + 0 1 3 2
* EXTREME (**Note: This test case is too much to fit in the report so please refer to the sim\_out file in the electronic submission under TestFiles directory**)
  + 1000 1000 0 0

All of the testing was performed at Curtin University in Building 314 on Level 2 in room 219 on the lab machines. Sepcifically machine a-06.

# **Errors:**

There were no known errors with running **“Simple Data Sharing”** program between both processes and threads. No memory leaks have been resulted from running these programs as they were all tested with valgrind.

# **Side Note:**

***Though I do want to point out something with the threads. If you were to run any command line parameters but the t2 time is zero. eg. 1 2 3 0, the program would seem appear like it is in deadlock or running an infinite loop but I can assure the program still terminates successfully it just takes longer to end because due to the writer’s wait time being zero the writer is constantly skipping the if() statement inside the writer() function causing an infinite loop until reader() function to finish reading. But once ther program terminates it writes the correct output to the sim\_out file.***

With the above implication, it is just my theory on what I think is happening in the program. A theory that might fix this is to implement a bounded buffer which I attempted and successfully failed so my Boolean conditions covered for this.

# **Input Files:**

All of the test files were written as one line separated with spaces with integers ranging from 1 to the file size for simplicity for reading. The name of the input file is called shared\_data.txt. Refer to the README for more.

../../../../Desktop/Assignment/TestFiles/TestFile.png

# **Expected Results and Actual Results:**

**Note:** I call a function called printReaderArray() that prints all of the elements a certain reader has read at the end of the while loop for each process and thread and that shows what elements that each reader has read from the data\_buffer. In the code this will be removed.

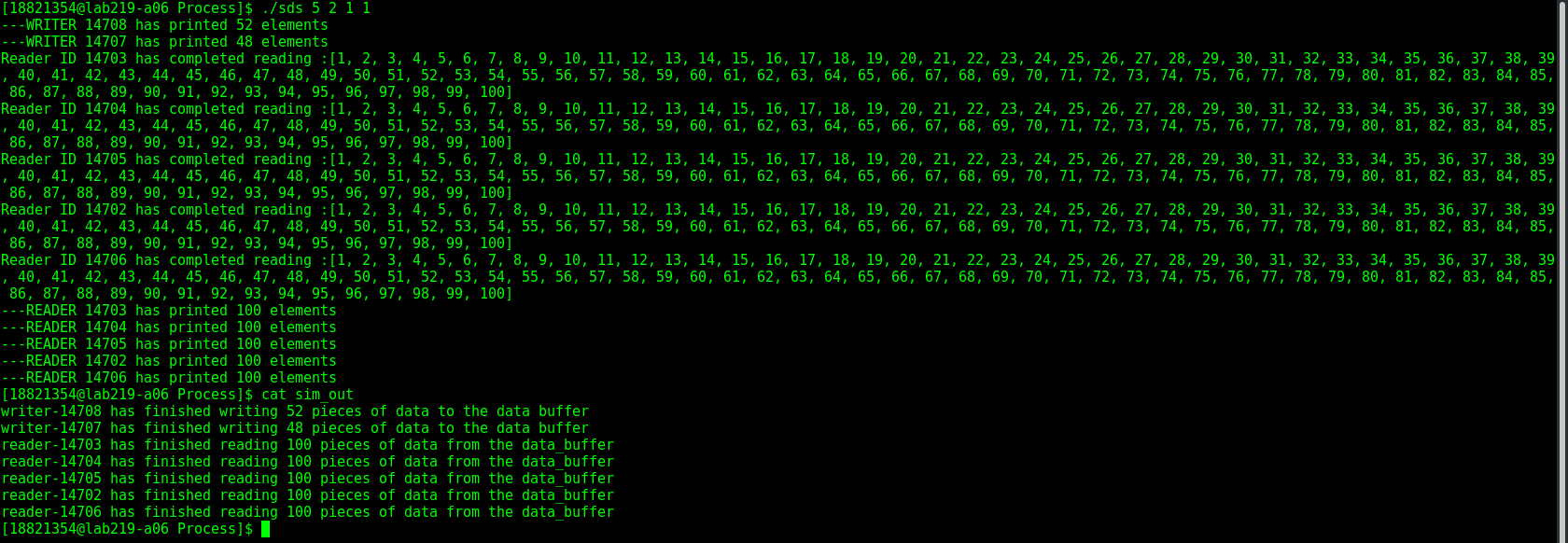
All expected output will be exactly the same as the actual output except they might appear in a different order because of the sleep and wait times in context with the printf() statements. The actual outputs below will show each reader reading the contents of the data\_buffer and then a cat output of the sim\_out file.

**Processes:**

**Test Case 1: Readers Greater Than Writers**

Input : 5 2 1 1

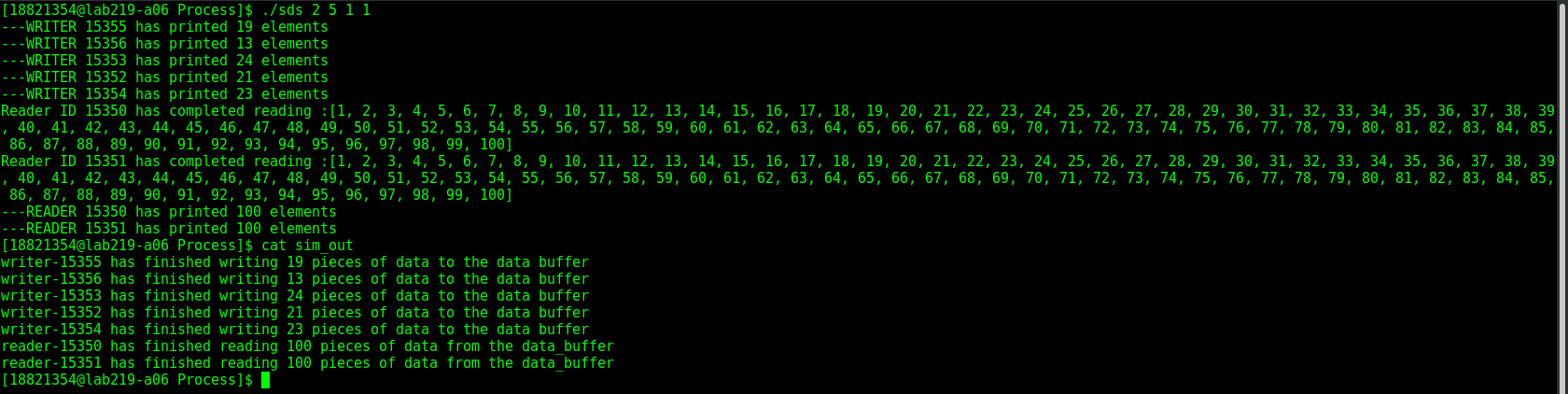
Output:



**Test Case 2: Readers Less Than Writers**

Input : 2 5 1 1

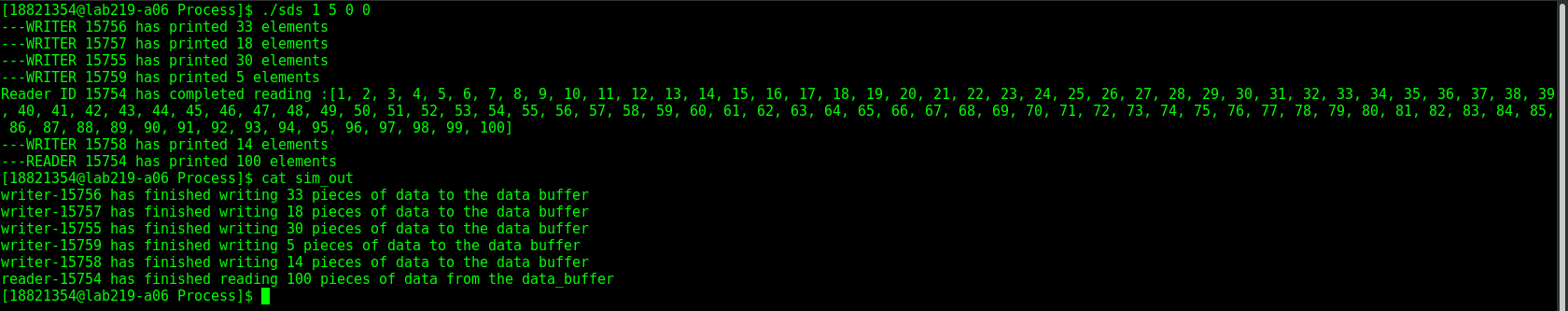
Output:



**Test Case 3: Zero Sleep Times**

Input : 1 5 0 0

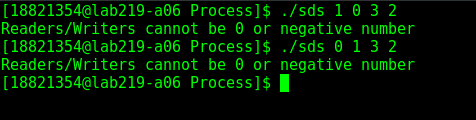
Output:



**Test Case 4: No Readers or Writers**

Input : 1 0 3 2, 0 1 3 2

Output:

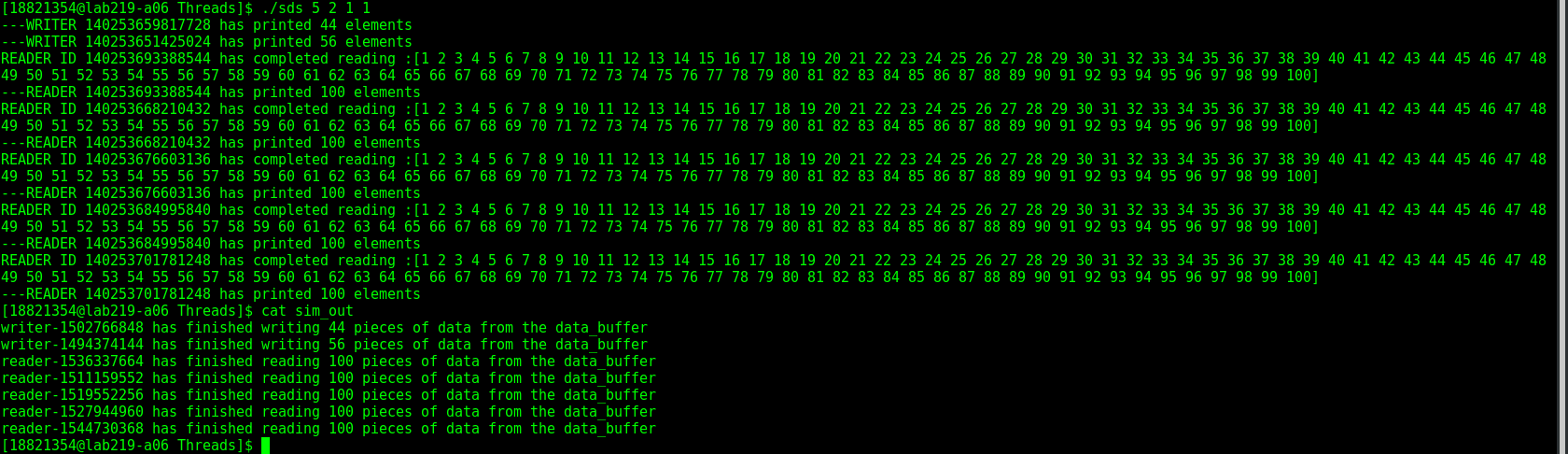


**Threads:**

**Test Case 1: Readers Greater Than Writers**

Input : 5 2 1 1

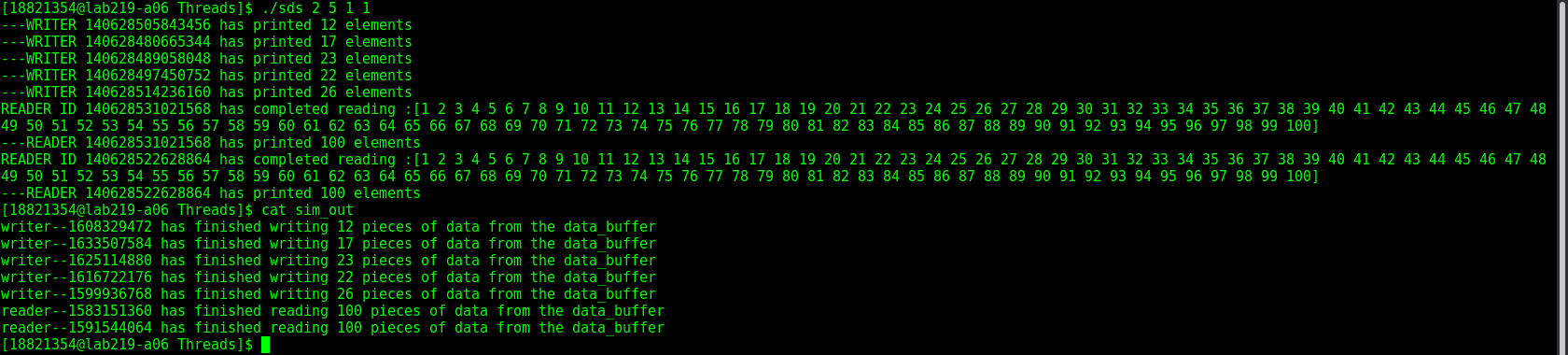
Output:



**Test Case 2: Readers Less Than Writers**

Input : 2 5 1 1

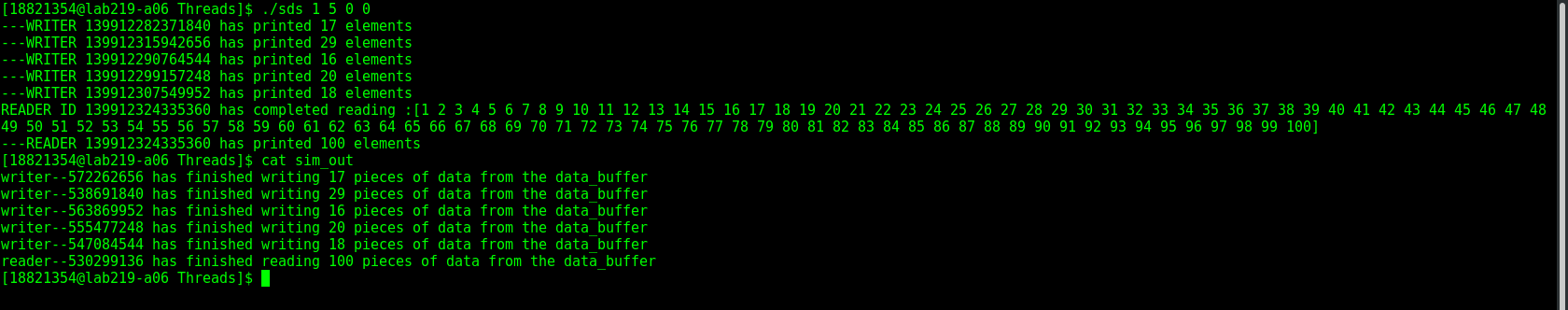
Output:



**Test Case 3: Zero Sleep Times**

Input : 1 5 0 0

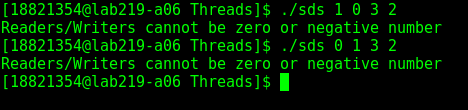
Output: ***NOTE: THIS TEST CASE TOOK ABOUT 20MINS TO RUN***

****

**Test Case 4: No Readers or Writers**

Input : 1 0 3 2, 0 1 3 2

Output:

****

# **Code:**

All of the code will be be attached to the hard copy of this report. If you are looking at the soft copy please refer to the actual .c and .h files