LAB-5

Prahalad VIjaykumar

2003128

**PART B**

1)The first question goes through the concept of semaphore . Played around with different values in the modulo spectrum and changed the buffer size .

The key functions here are:

Semget : which is used to create a number of semaphores whose parameters are a key,number of semaphore and permission access.

Here a semaphore of size 2 is created on stores a value of 5 and other stores value 0

Semctl : once a semaphore is created it cant be used directly since it has to be initialized here the initialization is done using this function .

Semop performs the increment and decrement of the value based on the number provided to the function if positive then increment and the other end decrements

P function or the wait function which decrements the value and then waits until the signal arrives from the V function which then increments the value of the semaphore.

Semaphore code snippet:

(Code taken from Geeksforgeeks)

struct Semaphore {

int value;

// q contains all Process Control Blocks(PCBs)

// corresponding to processes got blocked

// while performing down operation.

Queue<process> q;

} P(Semaphore s)

{

s.value = s.value - 1;

if (s.value < 0) {

// add process to queue

// here p is a process which is currently executing

q.push(p);

block();

}

else

return;

}

V(Semaphore s)

{

s.value = s.value + 1;

if (s.value <= 0) {

// remove process p from queue

Process p = q.pop();

wakeup(p);

}

else

return;

}

Here semaphore ops are perform on a child and a parent process

Here the child process will wait until a signal is received from the parent process , buf\_used is stored in index 0 . Since the value is 0 it will wait until the value is incremented by 1 by the parent process if the value is greater than 0 it will decrement and move ahead with the functionality given to it .

Notice the beginning scenario when the signal is sent once after that the child process had to wait for a second because the parent process was put to sleep and only if it is active can it provide the signal to the buf\_used index.

The same applies to the buffer\_space store in index 1 which has an initial value of 5 whose signal providence is performed by process 0 but noticed that the issue which was existed for the child process (pausing for the parent process to provide signal) wasn’t repeated in the parent process this is because there is a high initial value of 5 even if the child process is asleep the parent process still will have a value greater than 0 since it started with a larger value.

But when came to the 8 buffer writing that is when the issue arises for the parent process this is because the value of the buffer\_space would have become less than 0 hence waiting for the child process to provide the signal for the same but it is sleeping for 1 sec due to the modulo condition x%3==1 . The Output received:

The writer process begins.

Writer's report: item 0 put in buffer

The reader process begins.

Reader's report: item 0 == 0

Writer's report: item 1 put in buffer

Writer's report: item 2 put in buffer

Writer's report: item 3 put in buffer

Writer's report: item 4 put in buffer

Reader's report: item 1 == 1

Writer's report: item 5 put in buffer

Writer's report: item 6 put in buffer

Reader's report: item 2 == 4

Reader's report: item 3 == 9

Reader's report: item 4 == 16

Writer's report: item 7 put in buffer

Writer's report: item 8 put in buffer

Writer's report: item 9 put in buffer

Reader's report: item 5 == 25

Reader's report: item 6 == 36

Reader's report: item 7 == 49

Reader's report: item 8 == 64

Reader's report: item 9 == 81

Reader done.

Writer done.

Semaphore cleanup complete.

2) here the changes to be done are just add all the templates used for adding semaphores then set the semaphore value to 1 so the changes to be made will only happen once . and there are two semaphores, one for buffer space and another for buffer use.

Child process waits for the buf\_used process once it arrives, copies the value to a local memory and prints it and sends a signal for buffer space .

While the parent process waits for buffer\_space at initial it doesn't wait since the value starts with one and then it sends signal to buffer\_used the result obtained :

The parent process begins.

Parent's report: current index = 0

The child process begins.

Parent's report: current index = 1

Child's report: current value = 0

Child's report: current value = 1

Parent's report: current index = 2

Child's report: current value = 1

Parent's report: current index = 3

Child's report: current value = 4

Parent's report: current index = 4

Child's report: current value = 9

Parent's report: current index = 5

Child's report: current value = 16

Parent's report: current index = 6

Child's report: current value = 25

Parent's report: current index = 7

Child's report: current value = 36

Parent's report: current index = 8

Child's report: current value = 64

Parent's report: current index = 9

Child's report: current value = 81

The child is done

The parent is done

3)This code is purely based on mutex lock which can be represented by a binary semaphore. Here initially the code starts off with sem\_init,sem\_create,P and V which are defined and are the same set of code used in semaphore which is done in (1). Here this can also be done with predefined functions like sem\_init etc. these functions come under POSIX Semaphore.

Mutex Lock involves creating a Semaphore with the count of value of semaphore as 1 so it can have only 2 values which are 0 and 1 .sem\_init is used to create a 3 semaphore which handles buffer\_space,buffer\_used and the mutex respectively.

Firstly the writer checks if the value in buffer space is greater than 0 if yes then move ahead to the mutex lock where the value can either be 0 or 1 if it is 1 then makes the update to the buffer matrix add sends a signal that the write has done hence freeing up space for the next writer or reader.

While the child process performs the operation the parent process calls fork next to make their childs which will be the reader. Here waits for the signal from buf\_used to check this update is done by the writer. Once the value is greater than 0 it goes into the mutex lock, reads the value and sends the signal which frees up the space for operation of the next writer or reader. Then sends a signal for a free buffer\_space.

After reading and writing are done all are cleaned up!!!

Sample output:

The writer process 1 begins.

The writer process 3 begins.

The writer process 2 begins.

The writer process 4 begins.

The writer process 5 begins.

The reader process 1 begins.

Reader 1: item 0 == 100

The writer process 6 begins.

Reader 1: item 1 == 101

Reader 1: item 2 == 102

Reader 1: item 3 == 103

Reader 1: item 4 == 104

The reader process 2 begins.

Reader 2: item 0 == 105

All child processes spawned by parent

Parent waiting for children to finish

Reader 2: item 1 == 300

Reader 2: item 2 == 200

Reader 2: item 3 == 400

The reader process 3 begins.

Reader 3: item 0 == 500

Reader 3: item 1 == 106

Reader 3: item 2 == 600

The reader process 4 begins.

The reader process 5 begins.

Reader 4: item 0 == 301

Reader 5: item 0 == 201

Reader 4: item 1 == 401

Reader 1: item 5 == 501

Reader 1: item 6 == 107

Reader 1: item 7 == 601

Reader 1: item 8 == 302

Reader 1: item 9 == 202

Reader 2: item 4 == 402

Reader 2: item 5 == 502

Reader 2: item 6 == 108

Reader 2: item 7 == 602

Reader 2: item 8 == 303

Writer 1 done.

Reader 3: item 3 == 203

Reader 5: item 1 == 403

Reader 3: item 4 == 503

Reader 4: item 2 == 109

Reader 5: item 2 == 603

Reader 3: item 5 == 304

Reader 4: item 3 == 204

Reader 5: item 3 == 404

Reader 3: item 6 == 504

Reader 4: item 4 == 604

Reader 5: item 4 == 305

Reader 3: item 7 == 205

Reader 4: item 5 == 405

Reader 5: item 5 == 505

Reader 4: item 6 == 605

Reader 1: item 10 == 306

Reader 1: item 11 == 206

Reader 1 done.

Reader 2: item 9 == 406

Reader 2: item 10 == 506

Reader 2: item 11 == 606

Reader 2 done.

Reader 3: item 8 == 307

Reader 5: item 6 == 207

Reader 4: item 7 == 407

Reader 3: item 9 == 507

Reader 5: item 7 == 607

Reader 4: item 8 == 208

Reader 3: item 10 == 308

Reader 5: item 8 == 408

Writer 2 done.

Reader 4: item 9 == 508

Reader 3: item 11 == 608

Reader 3 done.

Writer 3 done.

Writer 4 done.

Reader 5: item 9 == 209

Writer 5 done.

Writer 6 done.

Reader 4: item 10 == 309

Reader 5: item 10 == 409

Reader 4: item 11 == 509

Reader 5: item 11 == 609

Reader 5 done.

Reader 4 done.

Semaphore cleanup complete.

Mutex :

wait (mutex);

…..

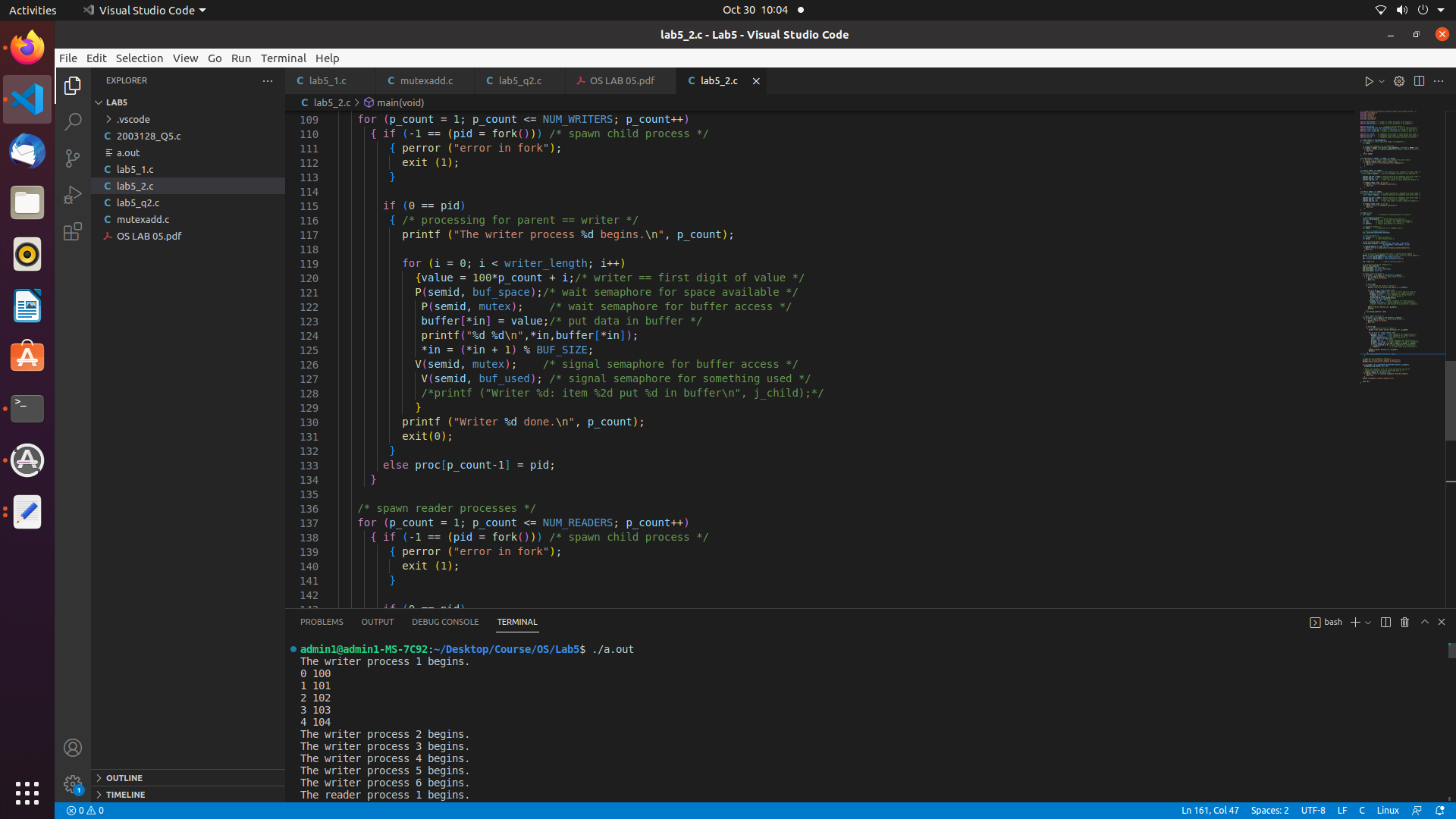
Critical Section

…..

signal (mutex);

This hence avoids multiple threads accessing the same address location simultaneously hence smooth sailing of all the operation.

4)Mutex will have only two values 0 and 1 so if multiple threads are concurrently trying to access an address then mutex allows one thread to access that location and locks it so the other threads have to wait until the previous thread has completed the task and once it is done it sends a signal that the mutex is free hence giving access to other thread. Removing this will make occurrence of multiple updates simultaneously hence will miss out on few data. The issue was clearly spotted when multiple threads were accessing the same index hence performing multiple operations on the same index than waiting. This can be observed by adding a print statement in the writer section.

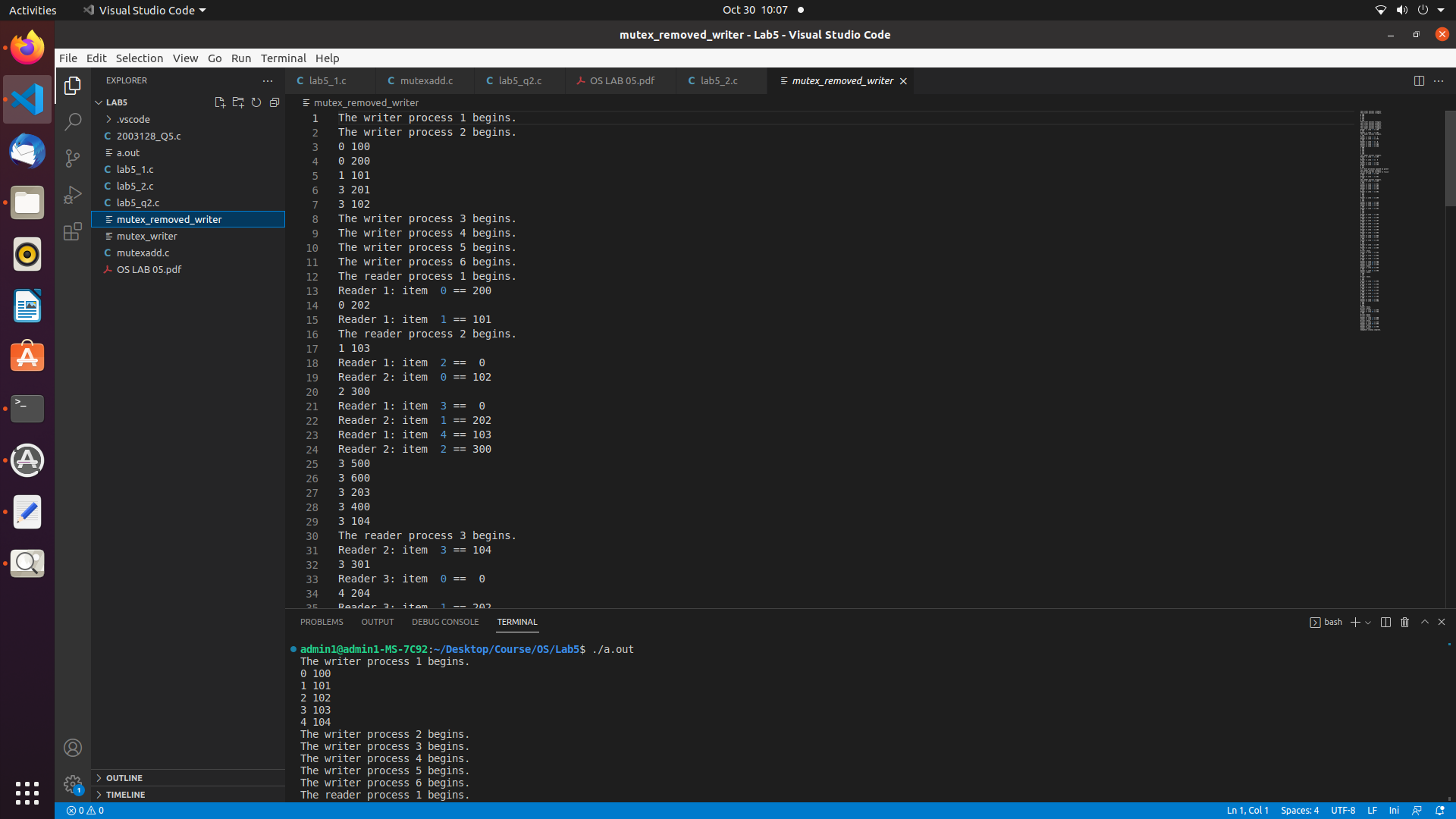


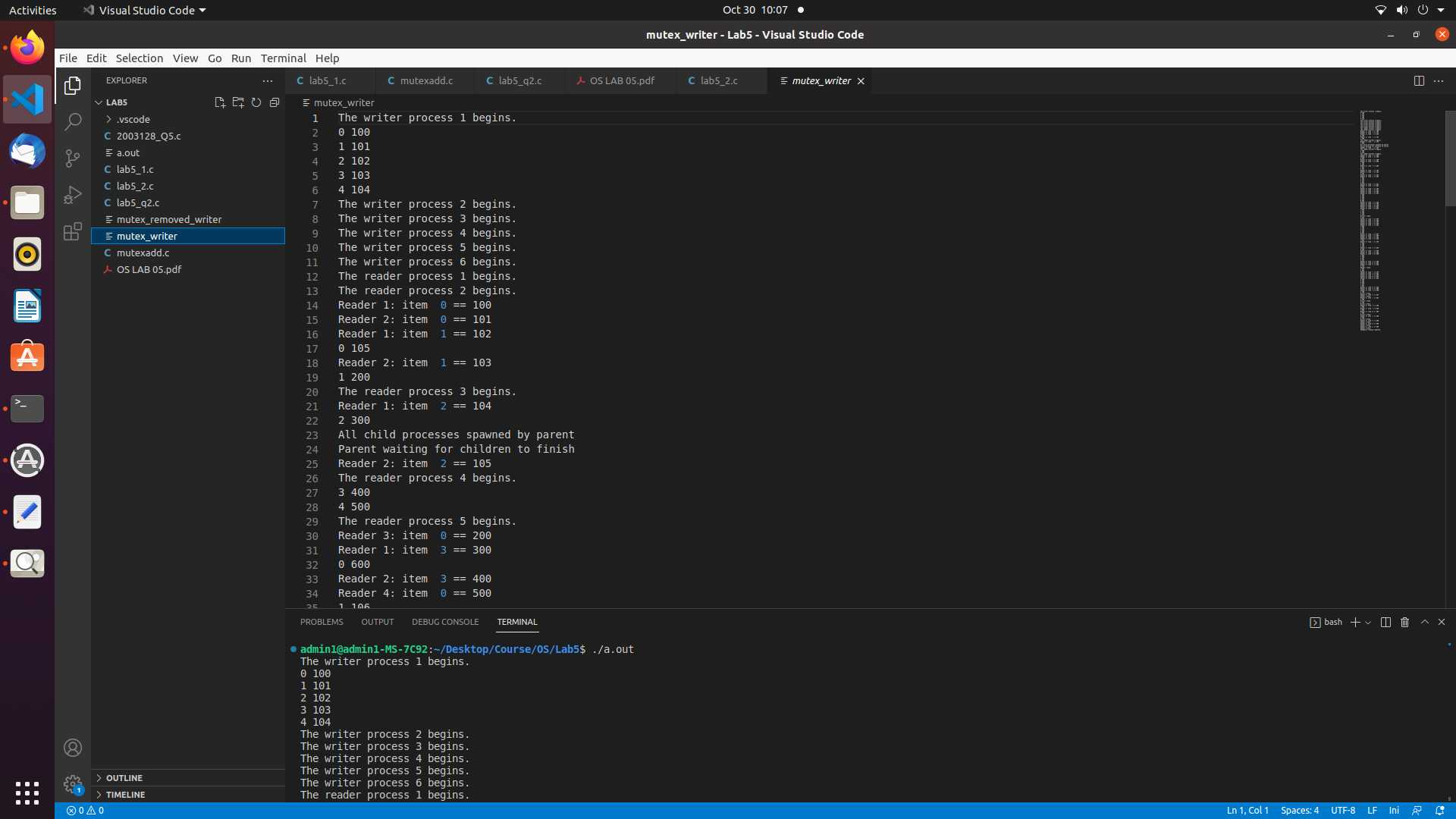
With mutex lock:It can be clearly observed that there are updates one after the other , each index is updated each time the thread is trying to access the buffer .

Without the mutex lock:It can be observed that index 3 is updated the value in the same index multiple times which is not supposed to happen.

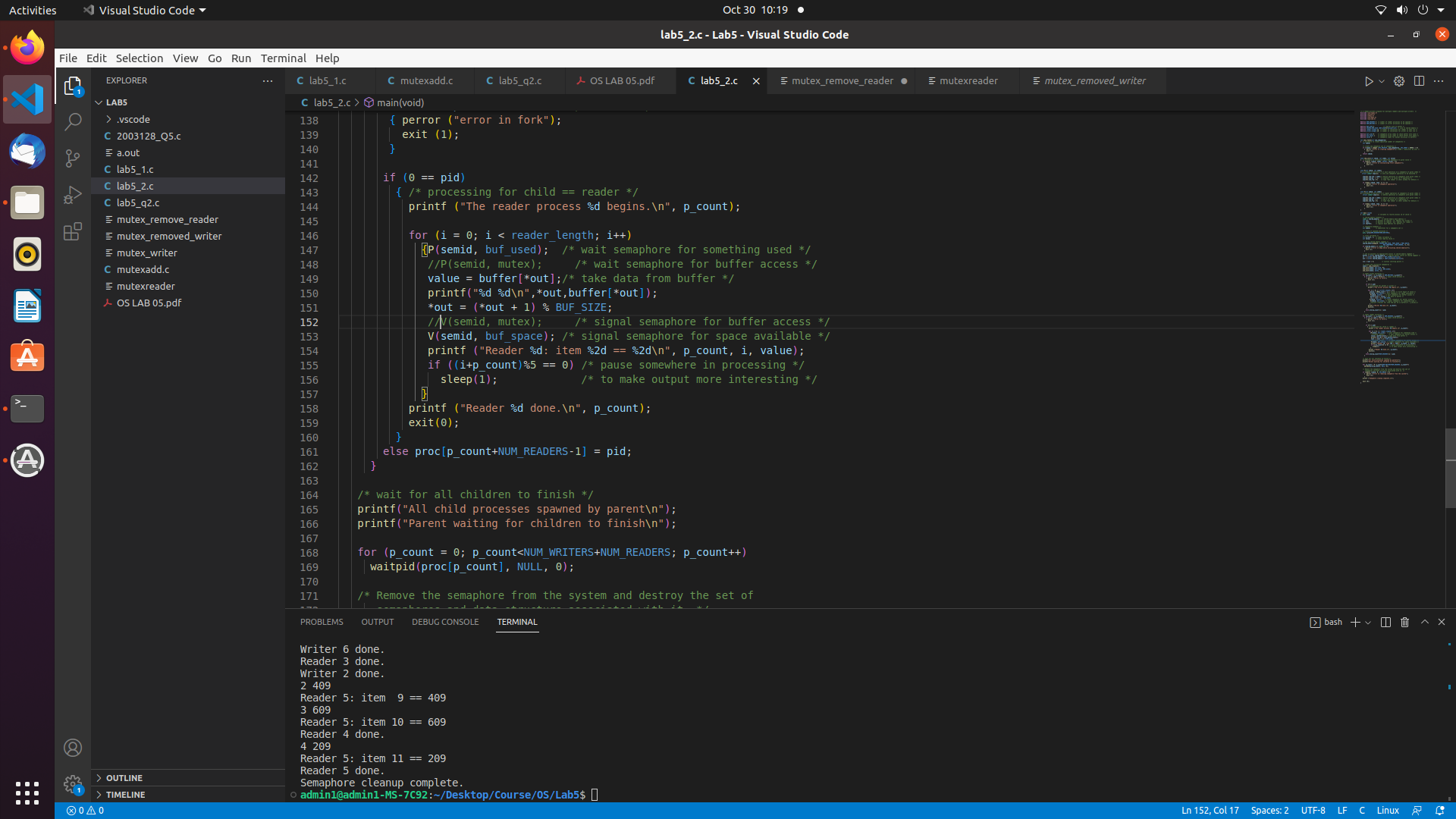
**Mutex writer relation:**

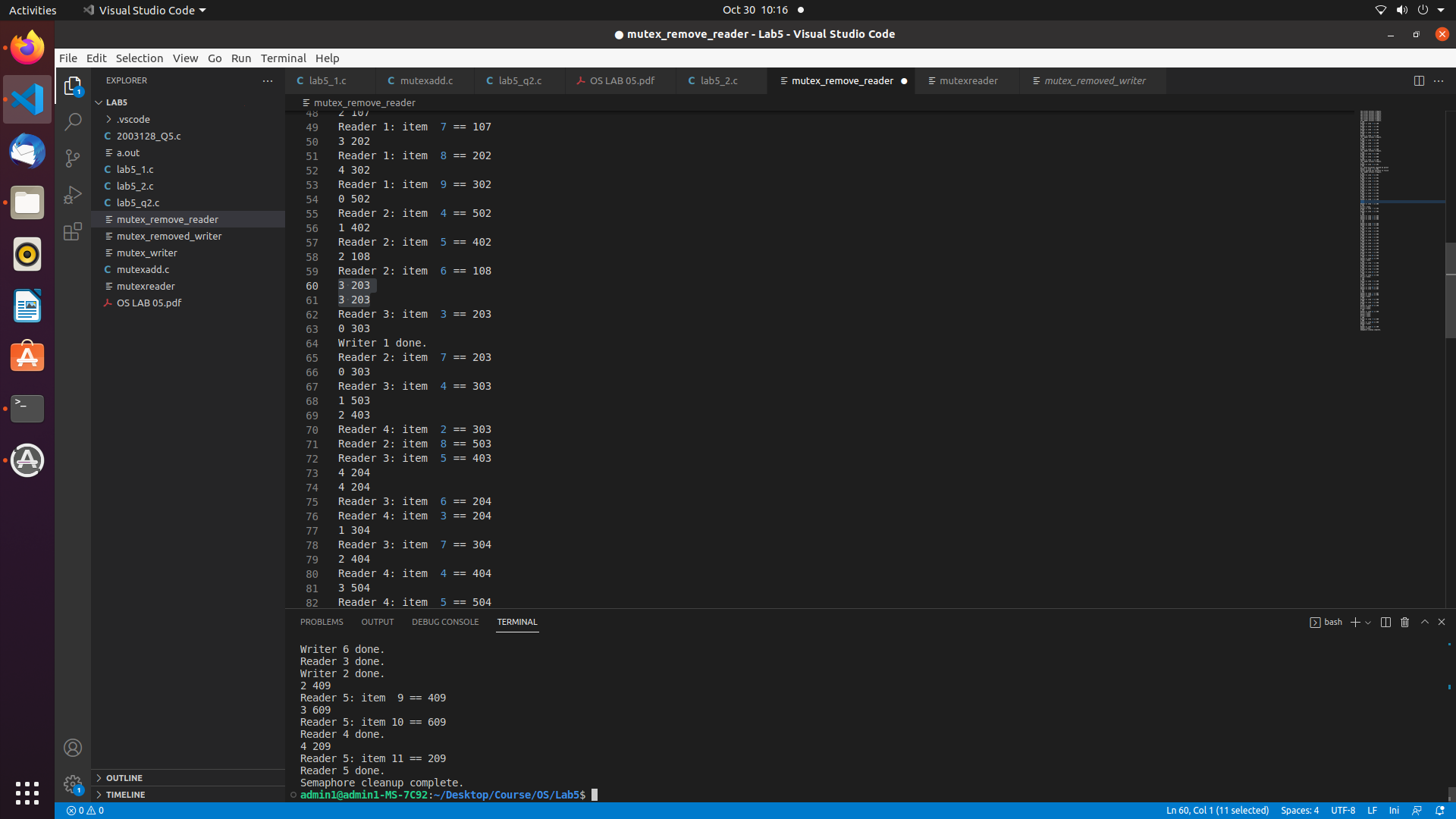
Without Mutex Lock:multiple index is updated simultaneously



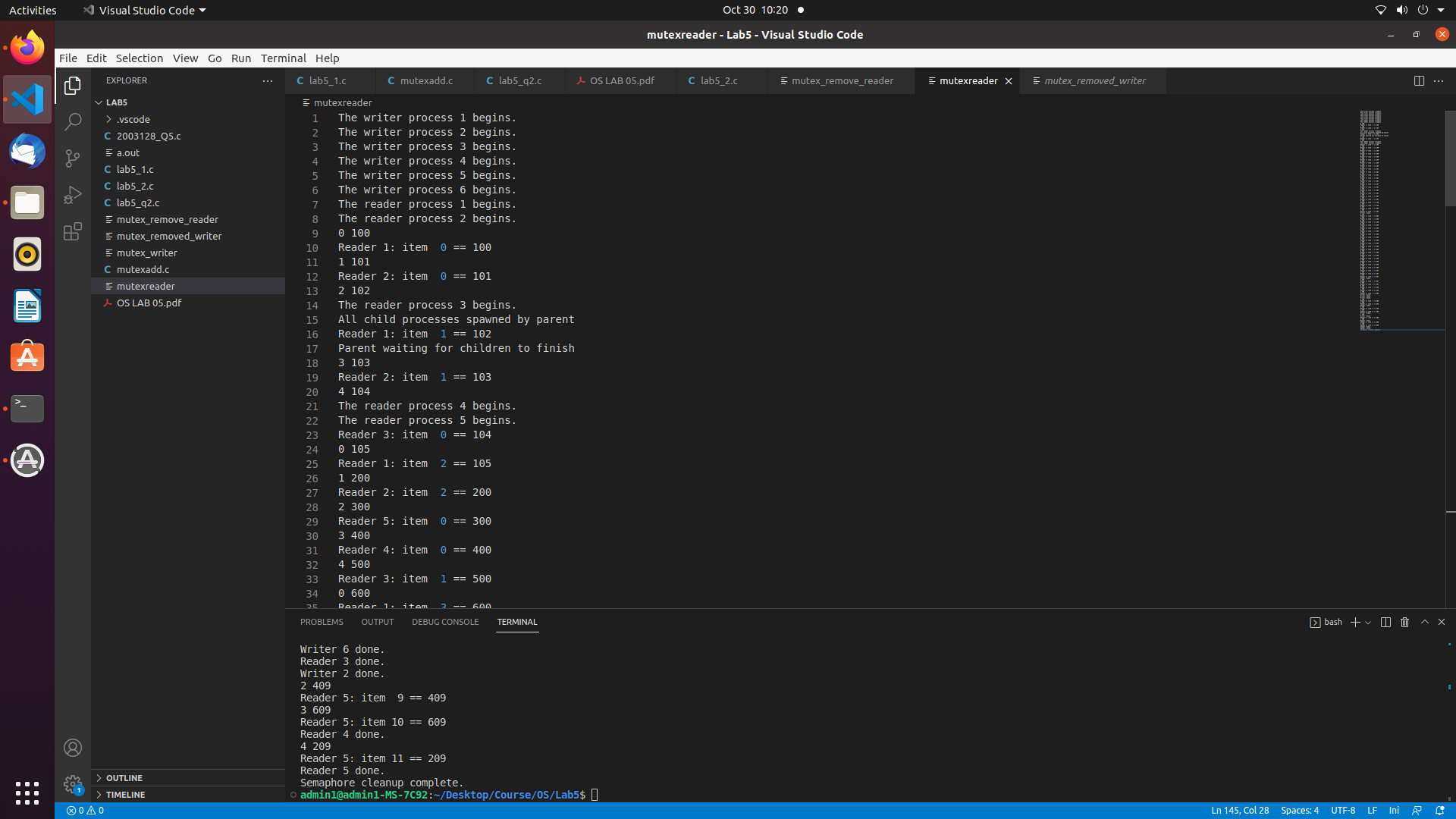
With Mutex Lock:can observe 0,1,2,3,4 are updated once at a time and happens so on.

**Mutex Reader Relation:**

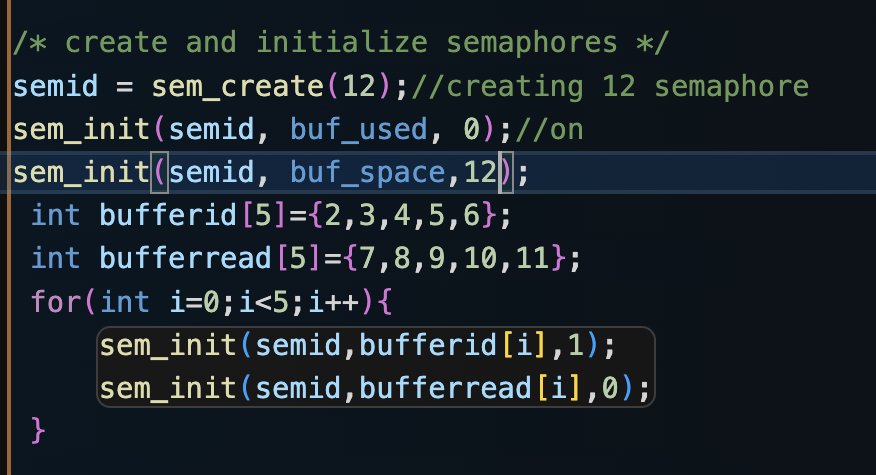
****

Mutex without reader will make the thread read the same index more than once.

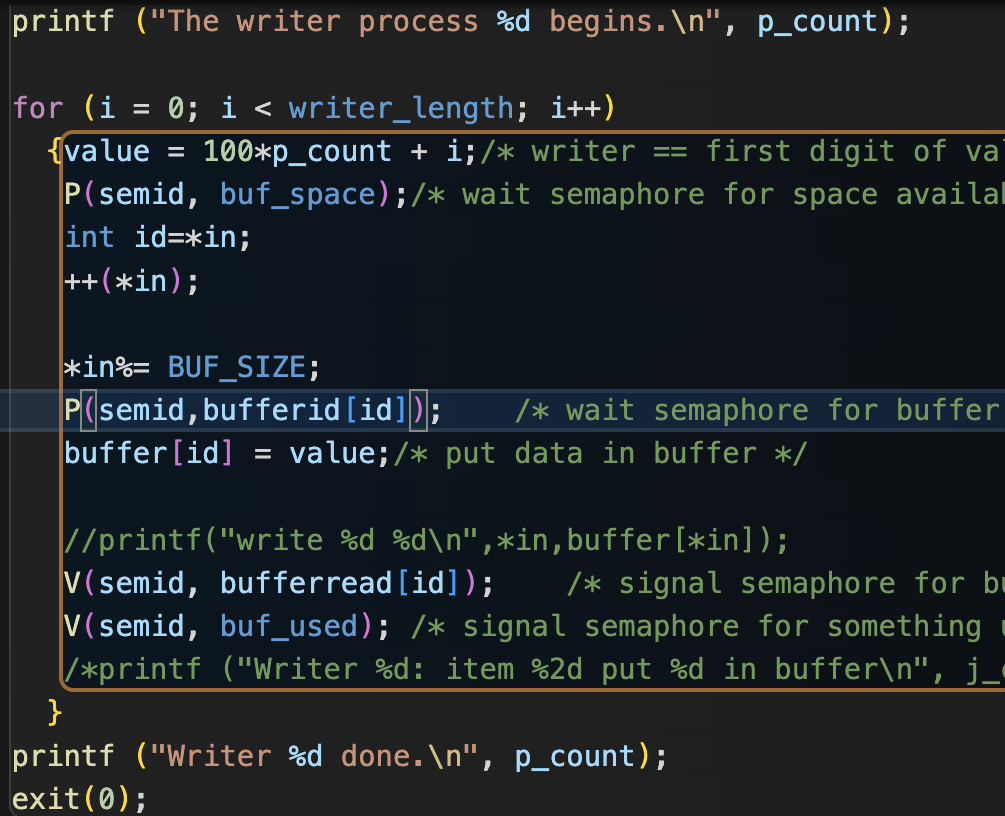
Mutex with reader is just like mutex with writer where there is only one index access at a time.



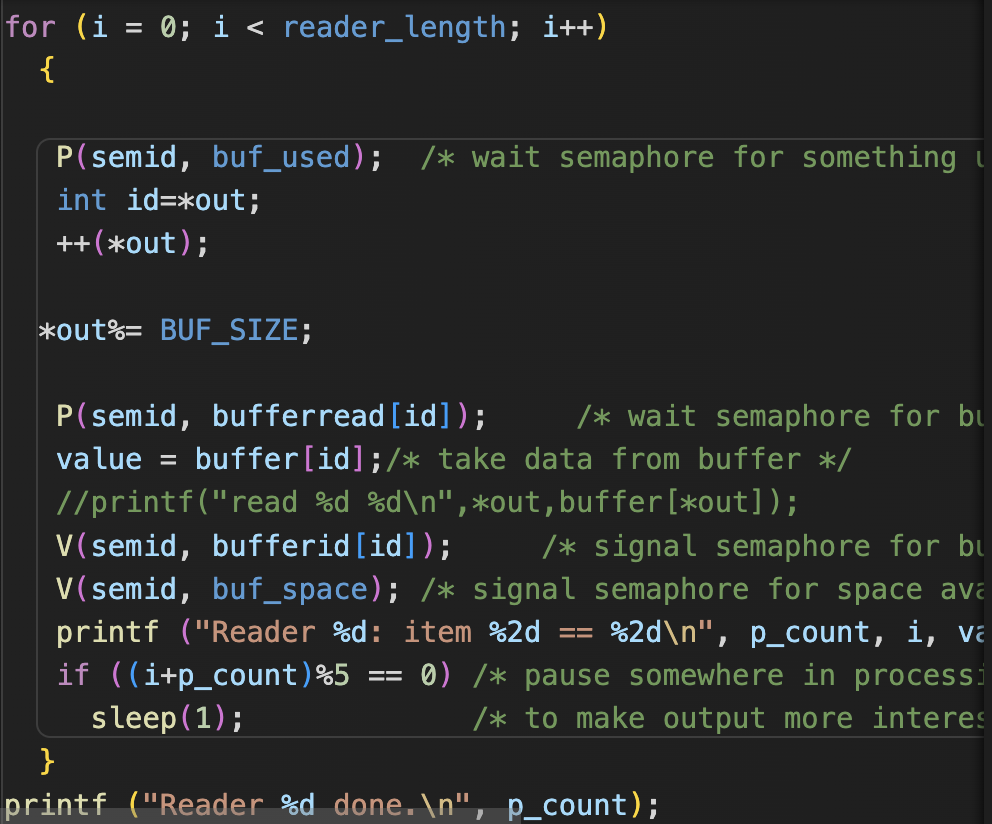
5) Add 10 extra semaphores, there are 2 for each of the 5 array indexes of the buffer.



For the writer, the process waits for the buffer[id] where id represents the index of the block. after performing all these computations the signal is sent to bufferread[id] which is for the reader to execute.



For the Reader process, the signal is received from the writer from bufferread[i] reads the value and then after computation. The signal is sent to the writer so that the process can get active and update the value of the array again and this keeps repeating .



This above approach is separately working for each buffer separately irrespective of the value in the buffer hence this helps in organizing the threads .