Yash Jain yjain1@ufl.edu UFID: 6912-1648

DOS Lab01

(a) Analyze what can go wrong considering that the two processes are sharing access to the buffer, and are running at possibly different speed (priorities), for varying bursts of CPU processing times. Then, based on your analysis, consider all possible problems or faulty situations that may arise and impact the Producer, the Consumer, or both.

Ans:

1. The producer can produce overwrite data at a location even before consumer consumed it, thus there is data loss or dirty read can happen by a consumer. (data overwriting, multiple updates at the same location before past data was consumed) Thus, leading the data in the buffer to be inconsistent.
2. Atomicity and consistency of data may not be preserved. A producer may be producing a data 'x' at a location b[2] for the second time, there might be a case when the producer is context switched when it was performing this task. In such a scenario, the producer cannot update the new data until it gains back the control of execution.
3. In the above case, Race Condition may arise, if the shared variable is getting updated simultaneously by two processes. The data value updated by the last process stays.
4. If consumer has higher priority, then it will execute for more time. It will try to consume values in the buffer which are not filled as producer didn't produce enough data. This is an error as consumer cannot consume at a location unless producer has produced before at that location.
5. The consumer cannot know whether the producer has produced at a particular location unless there is some kind of synchronization between them. (lack of synchronization) The consumer might just keep checking or might just keep waiting if it is not signalled by the producer. (scenario of busy waiting)
6. There might a case where producer tries to produce when the buffer is already full or other case where consumer tries to consume when the buffer is empty.
7. The process might enter into a deadlock as producer might keep producing and when the buffer is full it will have to stop given the condition it will just keep waiting for consumer to consume who if is of lower priority may not get a chance to execute, thus producer keeps on waiting.

(b) You are asked to implement a mutex in Xinu, using only Xinu semaphores (the semaphore's count must always be either 0 or 1 to be a proper mutex). A functional mutex consists of two methods: acquire and release. A mutex is acquired to start a critical section; this is when no other processes can run code that also requires that mutex – they will wait instead. Once the critical section ends, the mutex is released, and other processes can continue if they were waiting to acquire the mutex themselves.

Ans: Implement wait and signal functions of Xinu in Mutex_acquire and Mutex_release. i.e wrapping up wait and signal in these above functions.

(c). Using the mutex functionality you just created, implement the producer-consumer problem correctly (meaning, no problems or faulty situations). You are to use the provided *main.c* to define and allocate a shared circular buffer, and then create both the producer and the consumer processes. Note that more than one mutex may be required. You should program both processes with "simulation code" to actually produce or consume. Both processes should output to the console what they are producing or consuming.

Ans: main_mutex.c

(d) Use Xinu semaphores properly (counting up and down higher than 1) to implement the producer-consumer problem correctly and effectively. You are to use a copy of your main() form part (c), replacing the producer and consumer functions. Note that a mutex may still be required along with the proper use of semaphores. Be sure that your implementation isn't faulty to critical case scenarios (locking and the like). You should program both processes with "simulation code" to actually produce or consume as in part (c). Both processes should output to the console what they are producing or consuming.

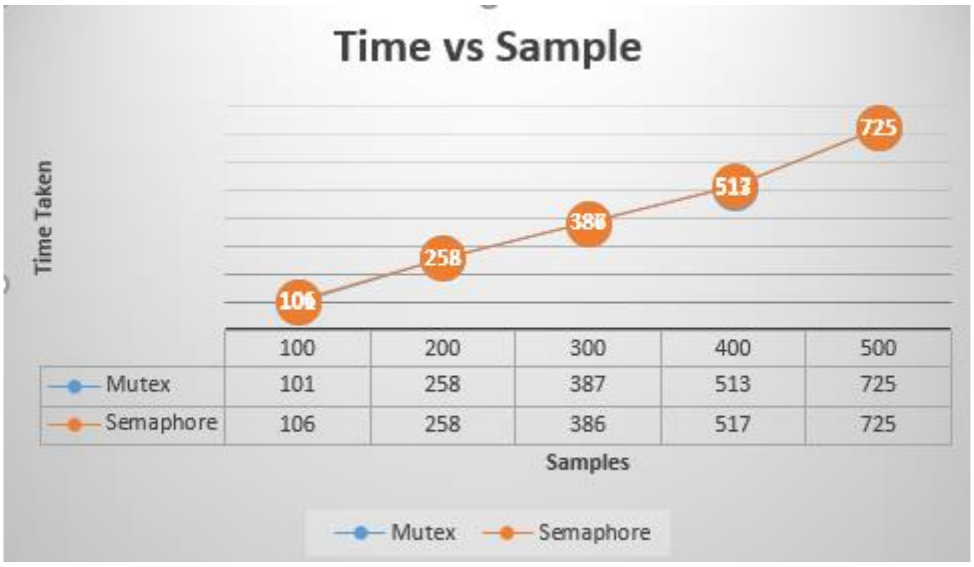
Ans: main_semaphore.c

Here two options are there to solve the problem. One using 2 semaphores and other is using two semaphores and one mutex. This given problem works fine for two semaphores but it may not work properly if there are more than one processes for producer and consumer. i.e. then the updates to the pointer where the producer should produce next or where the consumer should consume next should be updated in a critical section else inconsistencies may arise.

(e) Consider the differences between the two implementations. How efficiently are the processes able to cooperate before one is required to wait?

Ans: The first implementation which is using only mutex, either of the processes once making any kind of changes in the buffer acquires the lock. Hence when producer is producing consumer has to wait irrespective if there are other locations in the buffer which the consumer can consume.
Whereas in the semaphore implementation where we using counting semaphore such as empty and full to keep a track of the empty and full locations inside the buffer. Thus, when producer produces it acquires lock over the empty semaphore and signals the full semaphore and vice-versa for the consumer. Thus, we keep the track of the empty and full slots in the buffer. Also we maintain a pointer to the position of the producer and the consumer.
Thus, by this the consumer should not wait until and unless the buffer is completely empty or the producer should not wait until the buffer is completely full.

(f) You may have noticed that there is a timing function within *main.c*. You are asked to compile a graph plotting the amount of samples consumed against the amount of time taken for part (c) and (d), with 100, 200, 300, 400, and 500 samples, to compare the performance of the mutex version and the semaphore version. By uncommenting the *time_and_end()* function, at the end of execution, the amount of time taken for each amount of samples can be seen. Consider the performance of each method; why might one be faster than the other?
Ans:



	100	200	300	400	500
Mutex	101	258	387	513	725
Semaphore	106	258	386	517	725

Graph data values overlap.

In the above case we find that the time taken for both the cases is somewhat similar. It was a case where we had only two processes one producer and other the consumer. There are not multiple instances of these processes. Also the priority of each process is the same so CPU gives equal time to both the processes. And execute in round robin fashion.
Thus, we cannot find any significant difference in the times taken for each of the samples in both the methods. Also the data was also lesser and the code was also very small that it might a case where one process can be completely executed once in a single time slice.

We can also see the output of the processes, producer produces one data a location the consumer process is the next one which is getting executed and it consumes what the producer has produced. It can be seen in the output of the code that the processes alternate hence, the time taken in this case is somewhat similar.