Operating Systems Principles Assignment 1 report

*Aditya Kerhalkar [s3844647]*

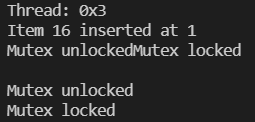
*GitHub project URL:* [*https://github.com/s3844647/osp\_assignment\_1*](https://github.com/s3844647/osp_assignment_1)

*Please note, while I have tried to finish everything, I’m not sure about the extent and accuracy of my knowledge.*

# Implementation details and issues

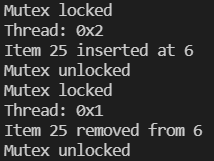
The solutions I have aimed to implement are of the producer-consumer problem (A) and the sleeping barbers problem (D). These implementations were based on my understanding of course topics and some online sources which are cited within the code itself.

The producer-consumer solution was implemented exclusively using two condition variables (enabling threads to sleep) and one mutex (restricting access). It displays information about items and their buckets, as well as which thread is performing a specific action. However, it doesn’t seem as if the concurrency has worked well as you get some inconsistencies. For instance in the output:



Under normal circumstances, the locking and unlocking of mutexes should be on different lines and sequential. This may indicate that two threads are trying to perform the same action at the same time (race condition?).

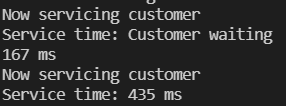
Comparatively normal behaviour looks like this:



Graceful exit seems to work appropriately. When the program is done running it displays a message, sleeps for ten seconds and then exits.

The sleeping barbers solution was implemented slightly differently. I ended up using semaphores as I couldn’t figure out how to get it to work using condition variable. In the end the solution uses two semaphores and one mutex.

When I have run this program, at most two seats seem to be occupied, evidenced by the output ‘2 seats remaining’. In addition, similarly to the producer-consumer problem, there is some irregular output which may indicate race conditions.



It’s possible this is an issue with the output rather than with the program itself.

# General algorithm details

## Producer-consumer problem

This is a common concurrency problem. Essentially, there are two main processes, the producer and the consumer, as well as a ‘buffer’ (represented by an array of integers and referred to as ‘buckets’ in this solution). The producer generates data and aims to add it to the buffer, and the consumer removes it (in this case, prints it to standard output).

The main issue here is how to ensure that data is added and removed in order, including a producer adding to a full array or a consumer removing from an empty one. This is solved by using condition handling and mutex.

When the method is run, it starts with initialising the mutex and creating threads: five producers and five consumers. The producer process generates a random number between 1 and 100. It then locks the mutex so that only one thread at a time can access the following code, avoiding race conditions. Both the producer and the consumer use condition variables to ensure that the thread sleeps until it is activated for the first time, and then signalled the second time (using the integer variables ‘prodDone’ and ‘conDone’ to store their respective states).

The producer adds its randomly generated number to the array (initially at position 0). It then increments the inserting position, signals other threads and unlocks the mutex. The consumer essentially does the same but receives an item from the array and prints it out. There are two integer variables ‘in’ and ‘out’ which control positions.

Finally, the main method waits for the threads to be finished.

## Sleeping barbers problem

This problem also centres around a distinction between two types of procedure. There is one ‘barber’ and multiple ‘customers’. My implementation uses semaphores and a mutex. When the method is run it initialises the semaphores and creates a barber thread and a thread for making customers. The ‘makeCustomer’ thread then makes multiple customer threads: as many as it can generate in 10 seconds, with a random interval of 100-499 milliseconds between each.

When each customer method is made (‘entering’), it locks the mutex and takes up a spot in the ‘waiting room’ (decrementing number of available seats). The mutex is so that multiple threads cannot all access the waiting room at once. It then notifies the barber process and, if it is busy with another customer, will wait for the barber (using sem\_post and sem\_wait respectively). If there are no available seats the customer will leave.

The barber method, meanwhile, initially sets the semaphore ‘custReady’ to wait, indicating that it is waiting for a customer to service. When custReady signals, it increments the integer ‘availableSeats’ (removing from the waiting room), signals that the barber is ready, and then ‘services’ the customer. This is simulated by sleeping for between 100 and 499 milliseconds. Once all of the threads have run the program gracefully exits.

## Applications of problems

In general, the producer-consumer problem is relevant to issues of multiple processes having access to one resource. A notable example is printing. Many computers may have access to only one printer, and it is important to ensure that all printing jobs are completed sequentially with no disruption to each other. It can be modelled by a problem in which there are multiple ‘producers’ (computers generating data to be printed) and one ‘consumer’ (the printer). The process can ‘lock’ to ensure that only one computer at a time can access the printer resource.

In the case of the sleeping barbers problem (or concurrency problems in general), one possible application is task scheduling. The ‘barber’ might be substituted with the operating system and each ‘customer’ is a task which must be scheduled. This must be done in a way so that the OS runs each task in a specific order without deadlocks or race conditions. (???)

# References

[1] ‘pthread.h’, *The Single UNIX Specification*, 1997. Available at: <<https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html>> Accessed 12 September 2021.

[2] D. Yan, ‘Producer-Consumer Problem Using Condition Variable in C++’, *GitConnected*, 2020. Available at: <<https://levelup.gitconnected.com/producer-consumer-problem-using-condition-variable-in-c-6c4d96efcbbc>> Accessed 12 September 2021.

[3] ‘Producer Consumer Problem in C’, *GeeksforGeeks*, 2021. Available at: <<https://www.geeksforgeeks.org/producer-consumer-problem-in-c/>> Accessed 12 September 2021.

[4] ‘In which area of Computer Science is the "Producer Consumer Problem" applied or implemented ?’, *ResearchGate*, 2014. Available at: <<https://www.researchgate.net/post/In-which-area-of-Computer-Science-is-the-Producer-Consumer-Problem-applied-or-implemented>> Accessed 12 September 2021.

[5] M. De Beijer, ‘Database Concurrency Conflicts in the Real World’, *Code Magazine*, 2006. Available at <<https://www.codemag.com/article/0607081/Database-Concurrency-Conflicts-in-the-Real-World>> Accessed 12 September 2021.

References used in writing code can be found within the code files.