# COSC1114 Assignment 1 Report

**1. Identify yourself using student ID, Name, and Github project name/url**

s3873756
Sidhra Fernando-Plant
Github repository: https://github.com/SidhraFernando-Plant/COSC1114-A1
**Instructions to run the solutions can be found in README.md**

**2. Describe any issues and limitations of your implementation**

- Valgrind reports possible leaks for both solutions
- Sleeping barbers problem only completed to CR level, DI and HD parts not implemented
- Not necessarily an issue, but an important note - the solution to Problem A (producer - consumer problem) prints millions of lines of output as it runs for 10 seconds with many threads printing out numbers. I would recommend writing the output to a file to make it a bit more manageable. Because there are so many lines of output already, I put all outputs regarding locking mutex's and changing arrays on one line.
- Another note: please note that multiple customers can enter the waiting room though sometimes it is scheduled in such a way that only one customer is in the waiting room at a time.

**3a)** The program solution implements two algorithms: one from A (producer consumer problem) and one from B (sleeping barber problem)

**4a)**

Note: in both algorithms a value of -1 represents an empty space, and a value greater than -1 represents an filled space or value

**`note: this font style signifies a variable`**

*Producer consumer algorithm*

Global variables:

```
buckets → {-1, -1, -1, -1, -1}
full → false
keepGoing → true
```

Explanation: Buckets is an array of integers which the producer puts values into and the consumer gets values from. It is locked with a mutex, to prevent multiple consumer and producer threads accessing the bucket at once. This could cause the array to be consumed by two threads at once resulting in unpredictably output, or a producer could fill the bucket with a different value (overwrite) while a consumer thread is in the middle of printing. Full is a boolean representing whether all buckets are full, and it is linked to the pthread_cond_wait used to coordinate access to the buckets. keepGoing is used to trigger the end of the program: when it is changed to false, all producer and consumer threads will exit , followed by the main thread.

Main program thread

```
initialise mutex lock for buffer
initialise wait condition
```

```
create 5 threads to carry out the producer function
create 5 threads to carry out the consumer function
sleep for 10 seconds
keepGoing → false
exit program
```

Explanation: Creates the threads to produce and consume and initialises the mutex lock and wait condition used to coordinate access to the buckets variable. keepGoing is changed to false 10 seconds after all threads are created, exiting the program.

Producer function

```
while keepGoing is true
      lock buckets mutex
            while full is true
                  wait on signal for condition wait_here from consumer
            end while
            buffer → new random number between 0 and 1000
            full → true
      unlock buckets mutex
      send signal for condition wait_here to producer
end while
```

Explanation: Fills the buckets. The function is in a while loop hence when **keepGoing** becomes false the thread will stop running. The thread will try and acquire **buckets** - if it is locked it will wait until it is unlocked. If it is unlocked it will acquire the variable and lock it to prevent other threads accessing it at the same time. If the buckets are full, it will wait until the bucket is empty with pthread_cond_wait in a while loop. Once a signal that the bucket is empty is received the buckets are filled with random numbers. **full** is changed to true, and the mutex is unlocked as the thread is finished with the buffer variable. A signal is sent to waiting consumer threads that buckets are now full and ready to be consumed. Changing the global boolean, and waits and signals ensure threads only perform actions when feasible (ie. not consuming an empty bucket), and that the threads always wake up as they always signal to each other (preventing deadlock and starvation).

Consumer function

```
while keepGoing is true
      lock buffer mutex
            while full is false
                  wait on signal for condition wait_here from producer
            end while
            print buffer
            buffer → -1
            full → false
      unlock buffer mutex
      send signal for condition wait_here to consumer
end while
```

Explanation: The algorithm is similar to the producer function. The while loop to exit the thread after 10 seconds is the same, and **buckets** is also immediately locked. Conversely to the producer, while the bucket is empty the thread will wait and will resumed when it receives signal from the producer that the buckets are filled. It will print what is in the buckets and empty them. Finally, **full** is set to false, mutex is unlocked, and signal is sent to waiting producer threads to resume as buckets are empty. The mutex and wait conditions are similarly used in this algorithm to prevent deadlock and starvation.

*Sleeping barbers algorithm*

Global variables:

```
seats → {-1, -1, -1, -1, -1}
customer → false
keepGoing → true
```

Explanation: **seats** is an array of integers representing the waiting room - if a seat is filled, the array value at that position will be the id of the customer sitting in the seat (more on this later). This variable has a mutex lock to ensure that only one thread can access it at a time - if the seating was to be changed by one thread while another was accessing/changing it this would cause unexpected behaviours. The boolean **customer** represents whether there are customers in the waiting room and is linked to a pthread_cond_wait used to send the barber to sleep when the shop is empty and wake him up when there are customers . The boolean **keepGoing** is the same manner as problem A, please refer to the previous explanation regarding this.

Main program thread

```
initialise mutex lock for seats
initialise wait condition
create 1 barber thread
create 1 generateCustomers thread
sleep for 10 seconds
keepGoing → false
exit program
```

Explanation: Creates the barber thread and thread to enter customers into the shop at random millisecond intervals. Creation of customer threads is done in its own thread so that it can keep looping until the signal to stop is sent after 10 seconds. If this looping occurred in the main program then the sleep would not be possible.The mutex lock and wait condition used in the threads is also initialised. After sleeping for 10 seconds while the threads run, **keepGoing** is made false, exiting the program.

generateCustomers function

```
tid → initialise an array of threads of size 300
      for i in range 0 to 300 do
            randomNo → generate a random number between 0 and 700000
            sleep for randomNo milliseconds
            create a thread with tid[i] to execute customer function
```

```
            increment i by 1
      end do
end while
```

Explanation: First, a large array of initialised threads is created - length 300 has been chosen in this case as the program will run for 10 seconds and in this time much less than 100 threads will be generated, so there is ample buffer. Each thread is then created to execute the customer function. Before each new customer thread is created, the generator thread will sleep for a random number of milliseconds so customers enter at random times. The position of the initialised thread in **tid** is passed to the new thread as the customerId.

customer(customerId) function

```
customerNo → customerId parameter
seatAvailable → false
mySeat → -1
lock seats mutex
for i in range 0 to 5 do
      if seats[i] is available and mySeat is -1
            mySeat → i
            seats[i] → customerNo
      end if
end do
if mySeat is not -1
      customer → true
      send signal for condition wait_here to barber
      unlock seats mutex
else
      unlock seats mutex
end if
```

Explanation: **customerNo** is the id of the customer, passed to the function upon thread creation. It is not essential to functionality but is used in print statements to see when a specific customer is getting served (to verify that the barber is serving customers in order). **seatAvailable** is a boolean that represents whether there are any empty seats in the waiting room, and **mySeat** is an integer that represents the customer's seat (the position that they occupy in the seat array). A value of -1 means they do not have a seat.

The algorithm locks the **seats** mutex as it will be reading and modifying this array. It checks for empty seats, and if an empty seat is found it is assigned to the customer by updating the **seats** array to contain the **customerNo**, and updating **mySeat** to the position that the customer is sitting at. The first seat found will be occupied to create a LIFO data structure allowing the barber to service customers in order. If an empty seat was not found, **mySeat** will still be -1 and the thread will unlock the mutex and exit. If a seat was found, **customer** will be made true and a signal will be sent to the barber thread (with pthread_cond_signal) to wake up and serve the customers.

Barber function

```
while keepGoing is true
      lock seats mutex
            while customer is false
                  wait on signal for condition wait_here from customer
            end while
            for i in range 0 to 5 do
                  if seats[i] is occupied
                        randomNo → new random number in range 0-700000
                        sleep for randomNo milliseconds
                        seats[i] → -1
                  end if
            end do
            var customer → false
      unlock seats mutex
      send signal for condition wait_here to customer
end while
```

Explanation: The algorithm begins by locking the seat array, and then waits for a customer to enter by looping while customer is false on a wait condition. When a customer sits in the waiting room, it sends a signal for the barber thread to resume - once the barber thread resumes it will loop through the **seats** array. The barber will service the customer in the first seat, and then move all other customers up to the seat in front of them. He will continue these steps until the first seat is empty. As customers sit in the first seat the find available, the array seats contains customers in order of arrival, so by iterating through the array in this manner the barber is servicing customers in the order they arrived in, creating a LIFO data structure. Further, by shuffling the customers after each haircut, if a customer enters the waiting room straight after someone has had their haircut they will still have to wait behind all the other existing customers as the first seat will be unavailable. After all customers have been served and removed from the Changing the boolean **customers** and use of waits and signals in both threads ensures that the barber is always woken up when there is customers, preventing starvation and deadlock.

**4b) Describe in great detail, 2 real-world industrial/business scenarios, where each algorithm applies. Describe how your variables should translate to such an environment.**

*Producer consumer algorithm*

This algorithm applies in a restaurant environment, where food items are continuously made and served to customers - I will refer to the situation of a burger restaurant. The **buckets** array would be the 'pass' (a designated area such as a bench where finished burgers are place). The 5 producer threads would be the kitchen staff making the burgers, and the 5 consumer threads would be the wait staff serving the burgers. Ordering is abstracted out of this particular scenario, assume that there is a constant influx of orders. The kitchen staff (producers) would check if there are already burgers on the pass. If the pass is full with burgers, they will wait until the pass is empty as otherwise they will not have anywhere to put their food - this is the pthread_cond_wait within the while loop. If there is space, they will prepare a burger and put it on the pass, and ring a bell to request service (this is the pthread_cond_signal). Once the burger

is made, they will check the pass again and make another burger or wait (this is equivalent to the while loop). The wait staff (consumers) will also check if there is a burger on the pass before doing anything. If no burger is on the pass, they will wait for a burger to be made and placed there as otherwise there is nothing for them to serve (pthread_cond_wait). If there are burgers on the pass, they will take one and serve it, and then return to the pass and verbally cue the kitchen staff to proceed with more orders (pthread_cond_signal). The while loop would be equivalent to them then coming back and checking/waiting/serving again. They will continue to do this while the restaurant remains open, and once the time reaches their closing time they will stop these processes - this is equivalent to the boolean **`keepGoing`** being changed to false.

Another situation where this algorithm would apply would be the system of garbage disposal (in a space like a shopping centre where the bin often fills up and many access itt). In this case, the array of buckets would be all of the bins in the space, producer threads would be customers in the shopping centre, and consumer threads would be the cleaning staff responsible for emptying the bin. When a customer has rubbish to dispose of they will look for a bin that is not full. If all bins are full, they dispose of the rubbish so they will wait for a bin to be emptied (pthread_cond_wait in while loop). If the bin is not full, they will place their rubbish in it. Producers will continue to produce rubbish and place it in the bins until they are full. Once all of the bins are full, customers will not have anywhere to put their rubbish so they will ask cleaning staff to empty the bins so that they can dispose of their rubbish (alternatively, if a cleaner sees a bin full they will empty it, so there could be a verbal or visual cue equivalent to the pthread_cond_signal). A cleaner will check bins to see if they are full or not, and if no bins are full they will wait until one is (pthread_cond_wait in while loop). Once a bin is full, the cleaner will empty it (consuming it), providing a visual signal to customers that they can now dispose of their rubbish there. Again, this will continue until the shooting centre closes, which would be equivalent to **`keepGoing`** becoming false.

*Sleeping barbers algorithm*

A service for taking photos can apply this algorithm (particularly taking school or staff photos where many individuals need to get their photo taken). The barber thread is the photographer, and customer threads are those wanting to get their photo taken. Let us say photos are being taken in a medium size room with a designated queueing area. Following the algorithm, someone who wants to get their photo taken would enter the room and check the queueing space. If it was full, they would leave as there would be no space for them to line up. This aligns with the for loop in the algorithm that checks all seats to see if they are empty - with seats equivalent to spots in the queue. If the customer finds a spot in the queue they will occupy it, equivalent to a customer id being assigned to a position in the seats array. The customer arrival will send a visual cue to the photographer that there are people who need their photo taken (like the pthread_cond_signal). The photographer will have a view of the queueing space, and if he can see that there is no-one there he will wait until some people come and line up, following the pthread_cond_wait in the while loop. Once he receives the visual signal that there are people waiting for a photo, he will begin taking people's photos and sending them on their way once he has completed their photo. He will always take the photo of the person at the front of the queue, and as he takes people's photos people will gradually move up to the front of the queue - this matches the way the barber thread shuffles around the seats array in the algorithm.

Another situation where this algorithm could be used is a drive through covid testing clinic. The barber thread is the person conducting the covid tests, and the customer threads are people

wanting to get a test. Following the algorithm, if someone wants to get a test done they will drive up to the clinic. If the clinic is a capacity they will leave as there will be no room for their car to join the queue, but if it is not full they will drive in and join the queue. While there are no cars in the queue, the person conducting the tests will wait until some cars join the queue (following the pthread_cond_wait in while loop). Once a car has joined the queue, the tester will begin testing the queue, starting at the first car. Once the tester has tested someone, they will drive off and the car that was second in the queue will move up to the first position, matching how the barber shifts the array in the algorithm. The tester will keep testing people until there is no one in the queue - after this they will wait again until more people join the queue and keep testing when people arrive (following the while(keepGoing) loop). The tester will continue this cycle of testing and waiting, and cars will keep arriving and getting tested until the closing time of the testing clinic - the clinic closing represents **`keepGoing`** becoming false.