

1. General outline of the assigned work

This assignment is to improve understanding of the bounded buffer problem AKA producer-consumer problem which is about concurrent access to a shared resource. This shared resource in this assignment is a circular buffer where memory is contiguous. As the memory is produced and then soon consumed, it does not need to be reshuffled, we simply move the pointers to form an end-to-end connected buffer.

A bounded buffer lets multiple producers and multiple consumers share a single buffer. Producers write data to the buffer and consumers read data from the buffer. A buffer of size n can store n buffer items and without proper synchronisation of produce and consume operations, this program will not work.

In this assignment, we perform the following operations in order:-

1. Read an item from a file
2. Write the item to buffer - Produce
3. Read item from buffer - Consume
4. Write item to another file

These operations must be synchronised among the thread(s) using locks or semaphores. We have two types of threads; `IN` threads and `OUT` threads. In my solution I have also added an extra worker thread that facilitates the copying. The main thread creates/initialises a circular buffer, creates all `IN` and `OUT` threads. Then, the main thread waits for all these threads to finish their work. We use the `sleep`/`nanosleep` function to put the threads to sleep to ensure synchronisation and avoid racing conditions.

The program maintains a log of all the operations ever performed, each log will have the details of the operations like what was the operation, which thread performed it, what was buffer index, offset, etc. The logging needs to be done in order and precisely, there should not be false logs of operations that didn't occur or logs of operations not in order of how they actually occurred, etc.

Each item buffer consists the following:-

1. Character byte
2. Offset position
3. State of item (empty item, etc)

The following is what each IN thread does:-

```
While (true)
```

```
    nanosleep()

    #critical section - READ
    mutex_lock
        Read Item (from I/p file)
        save operation details to Log File
    mutex_unlock

    nanosleep()

    #critical section - PRODUCE
    semaphore_wait
    mutex_lock
        Produce request
        save operation details to Log File
        update buffer index
    mutex_unlock
    semaphore_post
```

The following is what each OUT thread does:-

```
While (true)
```

```
    nanosleep()

    #critical section - CONSUME
    semaphore_wait
    mutex_lock
        Consume request
        mutex_lock
            save operation details to Log File
        mutex_unlock
        update buffer index
    mutex_unlock
    semaphore_post

    nanosleep()

    #critical section - WRITE
    mutex_lock
        Write Item (to output file)
        mutex_lock
            save operation details to Log File
        mutex_unlock
    mutex_unlock
```

These while loops keep running for the thread functions unless the threads are joined in the main function. All updates to the buffer state must be done in a critical section. Producers must block if the buffer is full and consumers must block if the buffer is empty. The program finishes when all producers (in threads) have reached EOF and then are joined and consumers go to sleep after buffer is empty and no more items are produced. Values like number of IN/OUT threads to be created, path to file that needs to be copied names of log and output files are all received as command line arguments.

2. Assigned work/components done successfully

1. Copies the input file exactly with no discrepancies.
2. Logs all the operation details completely; thread type, threadID/number, offset, integer value of the byte read and buffer index for produce/consume operations.
3. No deadlocks.
4. No racing conditions.
5. Proper synchronisation.
6. Proper use of mutex locks.
7. Operations performed and logged in proper order.

3. Assigned work/components not done successfully

Based on the number of IN threads to be created (nIN), I get extra lines in my log file. So for example, if the ideal number of operation lines in the log file is supposed to be 6088 then if I create 10 IN threads I get 6098 (6088 + 10) lines, if I create 100, I get 6188 and so on. Upon using grep on log file I noticed that my IN thread was reading from the files nIN number of times extra. Below is the result of the number of read, write, consume and produce operations when nIN = 1.

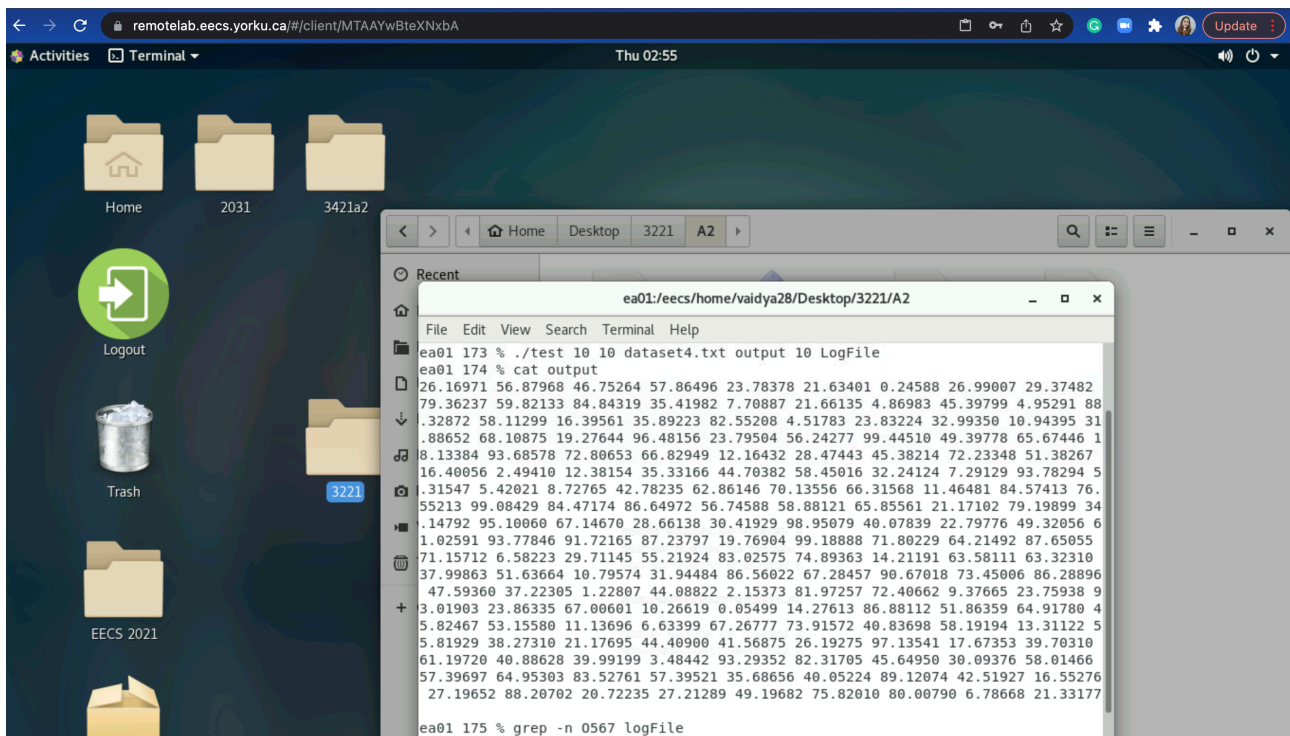
```
Devi@Daivakshi-MacBook-Pro A2 % cc -Wall -o cpy copy.c -lpthread
&& ./cpy 1 10 dataset4.txt output 10 LogFile && wc -l logFile
Creating threads ...
Joining threads ...
Process Complete
    6089 logFile
Devi@Daivakshi-MacBook-Pro A2 %
```

```
Devi@Daivakshi-MacBook-Pro A2 % grep -n read logFile | wc -l
    1523
Devi@Daivakshi-MacBook-Pro A2 % grep -n prod logFile | wc -l
    1522
Devi@Daivakshi-MacBook-Pro A2 % grep -n cons logFile | wc -l
    1522
Devi@Daivakshi-MacBook-Pro A2 % grep -n writ logFile | wc -l
    1522
Devi@Daivakshi-MacBook-Pro A2 %
```

I am unable to resolve this bug and get the exact number of lines. My code logic is precise however when implemented practically, it is causing this minor bug that might interfere with the evaluation of my assignment.

4. Assigned work/components not done successfully

Program run on EECS servers:-



```
remotelab.eecs.yorku.ca/#/client/MTAAYwBteXNxbA
Thu 02:55

Activities Terminal

Home 2031 3421a2

Logout

Trash

EECS 2021

ea01:/eecs/home/vaidya28/Desktop/3221/A2
File Edit View Search Terminal Help
ea01 173 % ./test 10 10 dataset4.txt output 10 LogFile
ea01 174 % cat output
26.16971 56.87968 46.75264 57.86496 23.78378 21.63401 0.24588 26.99007 29.37482
79.36237 59.82133 84.84319 35.41982 7.70887 21.66135 4.86983 45.39799 4.95291 88
.32872 58.11299 16.39561 35.89223 82.55208 4.51783 23.83224 32.99350 10.94395 31
.88652 68.10875 19.27644 96.48156 23.79504 56.24277 99.44510 49.39778 65.67446 1
8.13384 93.68578 72.80653 66.82949 12.16432 28.47443 45.38214 72.23348 51.38267
16.40056 2.49410 12.38154 35.33166 44.70382 58.45016 32.24124 7.29129 93.78294 5
.31547 5.42021 8.72765 42.78235 62.86146 70.13556 66.31568 11.46481 84.57413 76.
55213 99.08429 84.47174 86.64972 56.74588 58.88121 65.85561 21.17102 79.19899 34
.14792 95.10060 67.14670 28.66138 30.41929 98.95079 40.07839 22.79776 49.32056 6
1.02591 93.77846 91.72165 87.23797 19.76904 99.18888 71.80229 64.21492 87.65055
71.15712 6.58223 29.71145 55.21924 83.02575 74.89363 14.21191 63.58111 63.32310
37.99863 51.63664 10.79574 31.94484 86.56022 67.28457 90.67018 73.45006 86.28896
47.59360 37.22305 1.22807 44.08822 2.15373 81.97257 72.40662 9.37665 23.75938 9
3.01903 23.86335 67.00601 10.26619 0.05499 14.27613 86.88112 51.86359 64.91780 4
5.82467 53.15580 11.13696 6.63399 67.26777 73.91572 40.83698 58.19194 13.31122 5
5.81929 38.27310 21.17695 44.40900 41.56875 26.19275 97.13541 17.67353 39.70310
61.19720 40.88628 39.99199 3.48442 93.29352 82.31705 45.64950 30.09376 58.01466
57.39697 64.95303 83.52761 57.39521 35.68656 40.05224 89.12074 42.51927 16.55276
27.19652 88.20702 20.72235 27.21289 49.19682 75.82010 80.00790 6.78668 21.33177
ea01 175 % grep -n 0567 logFile
```