Name: Daivakshi Vaidya **Student Number:** 217761016

EECS 3221 - OS Fundamentals Assignment 2

---

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:-