# OPERATING SYSTEM LAB 7

| Roll No.: K041 | Name: Anish Sudhan Nair |
|---|---|
| Batch No.: A2/K2 | Date: 14/02/2022 |

Aim: To familiarise and understand semaphores

Running code without semaphores

```
[(base) anish@PotatoBook lab % clang no_sema.c -o ns
[(base) anish@PotatoBook lab % ./ns
[aaAABbbcBCcCdDDdEEeeFFGGHHffgghh%
 (base) anish@PotatoBook lab % ./ns
[aaAABbbBCccCdDDdeEEeffFFGGHHgghh%
 (base) anish@PotatoBook lab % ./ns
[aaAABbBCbcCcDDdEEdFFGGHHeeffgghh%
```

Running code with semaphores

```
[(base) anish@PotatoBook lab % clang sema.c -o s
[(base) anish@PotatoBook lab % ./s
[aaAABBCCbbDDccEEFFGGHHddeeffgghh%
 (base) anish@PotatoBook lab % ./s
[aaAABBCCbbccddeeDDEEffFFGGHHgghh%
 (base) anish@PotatoBook lab % ./s
 aaAABBCCbbDDEEccFFGGHHddeeffgghh%
```

CONCLUSION:

In this lab, we were introduced to the concept of semaphores which are required to maintain synchronisation between threads running concurrently. In code 1, we observe that when one process is sleeping, the other is allowed to run thereby causing two different letters to be printed sequentially instead of 2 of the same. In the second code we observe that the sequence is in pairs of two due to the use of semaphores which in spite of the sleeping state assigns wait state to the other process thereby ensuring two processes aren't in the critical state simultaneously. Only after the else block is executed does the other process get the signal state assigned, with the previous one now being assigned wait. This continues until the character string is exhausted. In code 1, since there is no implementation of wait and signal states, as soon as the first process is implemented, the second else block is executed and vice versa leading to undesirable outcomes.