

---

# Problems on Process Coordination - 1

## Group Members

Rahul Padhy (2022201003)  
Arun Das (2022201021)  
Praddyumn Shukla (2022201001)  
Vaibhav Saxena (202220)

## Supervisor

Nikhil

---

# Problem 1: The Editors Reporters Problem

- Given the number of editors  $n$  and the number of articles  $m$ , we have to assign the articles to the editors.
- Each editor will randomly pick an article and then accept or reject it.
- Each editor requires exactly one second to read an article. While reading no other editor can read the same article.
- If an editor is reading an article, other editors should not wait on the same article.
- If an editor accepts an article, no other editor can read it.
- Program ends when there is no article that can be accepted by any editor. Some articles may not be accepted by anyone.

# Solution

- Create a semaphore for each article.
- Each editor is run as a different thread.
- Whenever an editor tries to access an article, first lock the semaphore for the article.
- While the article is locked other editors can't access the article.
- After reading the article, the semaphore is unlocked.
- A list is maintained keeping track of open articles and the articles accepted by the editors.

# Preventing Waiting on an Article

- Before accessing the article, first check if the semaphore is already locked.
- This is done using the `sem_getvalue` function.
- If the article is already locked, pick a different random article.

```
35     int num = available[rand()%available.size()];  
36     int *buffer = (int *)malloc(sizeof(int));  
37     sem_getvalue(&semaphores[num], buffer);  
38     if(*buffer == 0)  
39         continue;
```

# Results

- Logs are printed to show the actions of the editors.
- The output contains the number of articles picked up by each editor and the indices of the articles picked.
- There are no race conditions or deadlock as semaphores are used and only one editor is allowed to read the article at a time.
- No starvation because a new random article is picked whenever an article is locked.

```
arun@arun-M1-NoteBook-Ultra:~/AOS_Project$ ./a.out 3 6
Editor 1 picked up article 3
Editor 2 picked up article 4
Editor 0 picked up article 0
Editor 0 ACCEPTED article 0
Editor 1 REJECTED article 3
Editor 1 picked up article 5
Editor 2 ACCEPTED article 4
Editor 2 picked up article 3
Editor 0 picked up article 1
Editor 1 ACCEPTED article 5
Editor 1 picked up article 2
Editor 2 ACCEPTED article 3
Editor 0 REJECTED article 1
Editor 2 picked up article 1
Editor 0 has finished its job
Editor 1 ACCEPTED article 2
Editor 2 ACCEPTED article 1
Editor 1 has finished its job
Editor 2 has finished its job
1
1
2
3 6
3
2 4 5
```

## Problem 2: The Dish Washing Problem

- There are  $n$  taps and  $n-1$  scrubbers (one between each consecutive taps).
- There are  $m$  students. A student has to wash three utensils in order - his plate, glass and spoon.
- A student should acquire a free tap first and will continue to occupy the same spot until he has washed all utensils. If a person is waiting for a tap, he continues to wait.
- Once a person has found a tap he will look for a free scrubber next to his tap. He must coordinate with his neighbors to finish washing his utensils as fast as possible.
- A plate takes 4 seconds to scrub and 5 seconds to wash.
- A glass takes 3 seconds to scrub and 3 seconds to wash.
- A spoon takes 2 seconds to scrub and 1 second to wash.
- While a person is washing his utensils, his neighbor can scrub his utensils.

# Solution

- Create semaphores for each tap and semaphores for each scrubber.
- Each student is a different thread and is randomly assigned a tap at the start.
- A student randomly picks up a scrubber from the left or right side and locks the semaphore.
- After picking up the scrubber sleep is used to simulate the scrubbing.
- After scrubbing unlock the semaphore and start washing.
- While the student is washing a neighbour a neighbour can pick up the unlocked scrubber.
- All three utensils are washed this way.
- A student who waits on a tap keeps waiting until his turn.

# Continually Checking the Left and Right Scrubber

- While waiting for an empty scrubber a student shouldn't wait on the scrubber on only a particular side.
- Similar to previous question use `sem_getvalue` to check if the scrubber on one side is locked. If it is checked the other side and continue while an empty scrubber is not found.

```
36     if(tap_number != n)
37         sem_getvalue(&scrotch[tap_number], buffer);
38     if(*buffer == 0){
39         if(tap_number != 0)
40             sem_getvalue(&scrotch[tap_number-1], buffer);
41         if(*buffer != 0){
42             sem_wait(&scrotch[tap_number-1]);
43             acquired_scrotch_number = tap_number-1;
44             break;
45         }
46     }
```



# Results

- The actions of each student and their finishing times are logged.
- No starvation occurs as all students are randomly assigned a tap at the start and scrubbers are checked continually on both sides.
- There are no deadlocks or race conditions as semaphores are used to acquire a scrubber.

```
arun@arun-Mi-NoteBook-Ultra: ~/AOS_Project$ ./a.out 2 4
Student 2 goes to tap 1
Student 3 goes to tap 0
Student 0 goes to tap 0
Student 1 goes to tap 0
Student 2 picked up scrotch 0
Student 2 completed scrubbing his Plate
Student 3 picked up scrotch 0
Student 3 completed scrubbing his Plate
Student 2 washed his Plate
Student 2 picked up scrotch 0
Student 2 completed scrubbing his Glass
Student 3 washed his Plate
Student 3 picked up scrotch 0
Student 2 washed his Glass
Student 2 picked up scrotch 0
Student 3 completed scrubbing his Glass
Student 2 completed scrubbing his Spoon
Student 3 washed his Glass
Student 3 picked up scrotch 0
Student 2 washed his Spoon

----- STUDENT 2 FINISHED AT TIME 19 -----

Student 3 completed scrubbing his Spoon
Student 3 washed his Spoon

----- STUDENT 3 FINISHED AT TIME 22 -----

Student 0 picked up scrotch 0
Student 0 completed scrubbing his Plate
Student 0 washed his Plate
Student 0 picked up scrotch 0
Student 0 completed scrubbing his Glass
Student 0 washed his Glass
Student 0 picked up scrotch 0
Student 0 completed scrubbing his Spoon
Student 0 washed his Spoon

----- STUDENT 0 FINISHED AT TIME 40 -----

Student 1 picked up scrotch 0
Student 1 completed scrubbing his Plate
Student 1 washed his Plate
Student 1 picked up scrotch 0
Student 1 completed scrubbing his Glass
Student 1 washed his Glass
Student 1 picked up scrotch 0
Student 1 completed scrubbing his Spoon
Student 1 washed his Spoon

----- STUDENT 1 FINISHED AT TIME 58 -----

arun@arun-Mi-NoteBook-Ultra: ~/AOS_Project$
```

## Problem 3 : Concurrent Merge Sort

- Given a number  $n$  and  $n$  numbers, sort the numbers using Merge Sort.
- Recursively make two child processes, one for the left half, one for the right half. If the number of elements in the array for a process is less than 5, perform a selection sort.
- The parent of the two children then merges the result and returns back to the parent and so on.
- Compare the performance of the merge sort with a normal merge sort implementation and make a report.
- You must use the `shmget`, `shmat` functions as taught in the tutorial.
- (Bonus) Use threads in place of processes for Concurrent Merge Sort. Add the performance comparison to the above report.

# Solution

- Firstly, a generic version normal merge sort is implemented, i.e., without any threads or processes.
  - If a partition size is observed to be less than 5, then Selection Sort is applied for that.
- Its observed that this version of merge sort works well enough for input sizes upto  $2 * 10^8$ .

```
AOS_Project on [ ] main [!?]
● > g++ Q3_Normal_Merge_Sort.cpp

AOS_Project on [ ] main [!?]
● > ./a.out
20000000
Time elapsed : 5.76259 seconds.

AOS_Project on [ ] main [!?] took 8s
● > ./a.out
100000000
Time elapsed : 29.1887 seconds.

AOS_Project on [ ] main [!?] took 34s
● > ./a.out
200000000
Time elapsed : 59.4702 seconds.

AOS_Project on [ ] main [!?] took 1m3s
○ > [ ]
```

## Solution (continued...)

- Now, a concurrent version of merge sort is implemented.
- The **shmget()** system call allocates a System V shared memory segment and returns the identifier of the memory segment.
- The **shmat()** system call attaches to the shared memory segment specified by the shmget() call above and returns the address of the shared memory segment.
- At each merge\_sort step, wherein left and right partitions are made, 2 separate processes are spawned to work upon the corresponding partitions.
  - The restriction being that if a certain partition contains less than 5 elements, then Selection Sort is applied to that partition and for this, no new process is spawned.
-

# Solution (continued...)