Operating Systems
Lab Assignment 3: Synchronization & Deadlock
Session: July-Dec. 2020; B.Tech. (I.T.) & B.I. Section-C
Assigned On: 01 Oct, 2020
Deadline for Submission: 10 Oct., 2020

NOTE: Although one assignment is assigned to one group, for proper learning of the course, it is important to learn the solutions of all the assignments (not only this assignment but all the previous assignments also). So, I suggest to all the students to collect and understand the solutions to all assignments from the corresponding groups, after the assignments are evaluated and feedback are received to the corresponding groups. Although in the viva for this particular assignment you have to defend for only the assignment assigned to your group only, towards the end of the semester, we may conduct grand vivas where questions from any assignment any be asked to any student irrespective of his/her group .

1. Implement a solution to the bounded-buffer producer-consumer problem using only two shared variables '**in**' and '**out**' (apart from the shared buffer itself), which indicate the positions in the buffer array where / from where the next item to be inserted / retrieved by the producer and the consumer process respectively. Note that in this solution, the buffer can contain maximum (BUFFERSIZE-1) items. In your solution, do not use busy waiting based iterative loops (as shown in book), rather, use the **pause()** and **kill()** system calls to block and unblock a process as demonstrated in my example programs / videos. Note that, in this solution, although **in** and **out** are shared variables, none of them are updated by multiple processes, rather, **in** is updated only by the producer process and **out** is updated only by the consumer process. Hence, we do not need to synchronize the access of these two shared variables.

2. Implement a solution to the bounded-buffer producer-consumer problem using only two shared variables '**in**' and '**out**' (apart from the shared buffer itself), which indicate the positions in the buffer array where / from where the next item to be inserted / retrieved by the producer and the consumer process respectively. Note that in this solution, the buffer can contain maximum (BUFFERSIZE-1) items. In your solution, do not use busy waiting based iterative loops (as shown in book), rather, use the **pause()** and **kill()** system calls to block and unblock a process as demonstrated in my example programs / videos. Note that, in this solution, although **in** and **out** are shared variables, none of them are updated by multiple processes, rather, **in** is updated only by the producer process and **out** is updated only by the consumer process. Hence, we do not need to synchronize the access of these two shared variables.
Now, implement the following on the top of the above solution:-

Producer process, on each iteration, takes details of a student (name, enroll no, section, age etc.) from the user and puts this data as an item into the shared buffer. The consumer process on the other hand, on each of its iteration, retrieves a data item from the buffer and saves it in a CSV file on the disk, where the fields (name, enroll no. etc.) of a data-item are stored in comma-separated format (as usually stored in CSV).

3. Implement the Peterson's two-process synchronization solution. Now, apply the solution on the race condition problem instance I have created in my sample program files "producer3.c" and "consumer3.c",

to demonstrate that your solution ensures that now the two processes will update the shared variable 'data->x' in a synchronized way and hence, would always output the correct result.

4. Implement a solution to the bounded-buffer producer-consumer problem which, in addition to the **in** and **out** shared variables as indicated in Q. No.1, also can use a shared **counter** variable whose value will indicate the no. of items in the shared buffer. Note that, this solution, unlike the solution of Q. No. 1, should be able to allow the shared buffer to contain maximum BUFFERSIZE items. However, unlike the **in** and **out** variables, the **counter** variable would be updated by both of the processes, and hence, creates a race condition. Implement the Peterson's two-process synchronization solution to ensure synchronized access to the **counter** variable by the two processes.

5. Implement the Bakery algorithm for 3 processes. Demonstrate by creating a sample example, similar to that shown in my "producer3.c" and "consumer3.c" example programs, the race condition situation among the three processes, when no synchronization solution is applied. Now, apply your bakery algo. solution and show that the race condition problem is resolved (by showing that it now always produces the correct results).

6. Implement blocking semaphore using the **pause()** and **kill()** system calls (to block and unblock a process, as demonstrated in my example programs / videos). Now, use your solution to ensure mutually exclusive access to a shared variable by two cooperating processes. Demonstrate the correctness of your solution by applying it to solve the race condition problem that I have created in my sample programs "producer3.c" and "consumer3.c".

7. Implement blocking semaphore using the **pause()** and **kill()** system calls (to block and unblock a process, as demonstrated in my example programs / videos). Now, use your solution to provide a solution to the bounded buffer producer-consumer problem. However, unlike in Q. No. 1 and 4, where the shared buffer is viewed as an 'array', you have to implement the shared buffer as a shared 'stack' only.

8. Implement blocking semaphore using the **pause()** and **kill()** system calls (to block and unblock a process, as demonstrated in my example programs / videos). Now, use your solution to provide a solution to the Readers-Writers problem, where each reader process reads the content of a file, whereas a writer process writes some texts input from the user into the same file (appends at the end of the file on each write).

9. Implement the Banker's algorithm. The algorithm should take the following inputs from the user:-

***n*:** total number of processes in the system

***m*:** number of resource types in the system

***Available*:** a vector of length *m* indicates the number of available resources of each type.

***Max*:** an *n* x *m* matrix defines the maximum demand of each process. If Max[*i,j*] = *k*, then process P*i* may request at most *k* instances of resource type R*j*.

***Allocation*:** an *n* x *m* matrix defines the number of resources of each type currently allocated to each process.

The program outputs whether the system currently is in safe state or not. If it is in safe state then output the corresponding safe sequence also.

After showing the previous output (if the system is currently in safe state), the program again takes the following input from the user:-

***i*:** some value between 1 and *n*

***Request*:** a vector of length *m* indicates the request-vector for process P*i*. If Request[*j*] = *k*, the process P*i* wants *k* instances of resource type R*j*.

The program decides whether the request can be granted immediately or not depending on whether or not granting the request leaves the system in an unsafe state.

10. Implement the deadlock detection algorithm (which is a variation of the Banker's Algo.) which takes the following inputs from the user:-

***n*:** total number of processes in the system

***m*:** number of resource types in the system

***Available*:** a vector of length *m* indicates the number of available resources of each type.

***Allocation*:** an *n* x *m* matrix defines the number of resources of each type currently allocated to each process.

***Request*:** an *n* x *m* matrix indicates the current request of each process. If Request[*i*,*j*] = *k*, then process P*i* is requesting *k* more instances of resource type R*j*.

The program outputs whether the system currently is in a deadlock state or not. If not, then the program also displays the sequence in which the processes can be executed.

Assignments of Questions to Groups:-

Group-ID	Q. No.		Group-ID	Q. No.
1	1		10	10
2	2		11	2
3	3		12	3
4	4		13	4
5	5		14	5
6	6		15	6
7	7		16	7
8	8		17	8
9	9		18	9
			19	10

What to Submit:-

1) The codes
2) PPT with audio-descriptions OR, a video presentation
3) VIVA (to be done by the TAs)

Where & How to Submit:-

Instructions will be given by the TA in-charge Mr. Fahiem