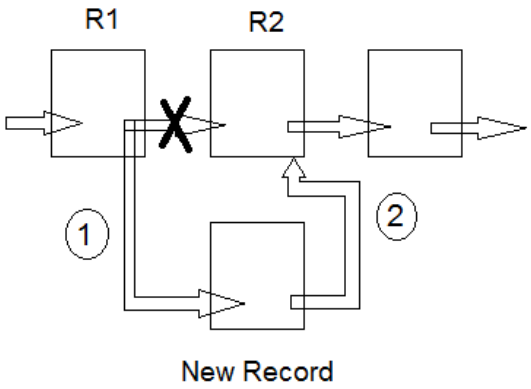
Answers of Problem Set (2):

[Q1] To insert new record between records R1 and R2 for example, two operations are required:

(1) Modify pointer in R1 to point to new record. (2) Let pointer in new record point at R2.



If a thread performs one operation and is pre-empted before doing the other, then other thread tries to traverse the list:

i) If only operation 1 is done then list is broken and error will result ii) If only operation 2 is done, the second thread will read the list without the new item.

[Q2] Two processes may access the shared variable protected by s1 simultaneously, resulting in possible errors. Processes trying to access the shared variable protected by s2 will not be able to operate and remain blocked.

[Q3] As explained in the lecture the solution will operate correctly. It is necessary however to have the same pointer to writing location in buffer by all writing processes to avoid overwriting data. Additional semaphore for mutual exclusion is thus necessary even if a circular buffer is used.

[Q4] We can perform the addition by p threads running in parallel, each adding a different set of (n/p) variables to obtain a partial sum. Main thread then waits until all these threads end, then add the p partial sums to obtain the total sum. Execution time will be reduced by nearly a factor of p.

Semaphores will be needed to ensure that main thread will wait until other threads end. Solution similar to problem P4 below can be used.

[Q5] If a low priority process enters the critical section first and is then pre-empted by a high priority process accessing the same data, the high priority process is blocked waiting for the low-priority process. This undesirable condition is called "priority inversion".

High priority process will be blocked for much longer times if there are many intermediate priority processes that can pre-empt the low priority process forcing the high priority process to wait for them.

To avoid this condition, the priority of the process in the critical section is raised to the priority of process waiting for it, thus intermediate priority processes cannot pre-empt

it. This technique, called "priority inheritance" is especially important in real-time systems.

[P1] a) Since m starts at 3 and is not incremented by any process, then process 1 will run for three iterations of the while loop and is then blocked. Thus "A" will be displayed three times.

b) Semaphore n starts at zero but process 1 performs up(n) three times before blocking. Each up(n) will allow either process 2 or 3 to avoid blocking by down(n) and perform an iteration. If process 2 runs, it performs up(n) again allowing additional iteration by process 2 or 3. Blocking of all process only occurs if process 3 runs three times, and hence "D" will be displayed three times.

c) One possibility is that processes 1 and 3 iterate three times each with process 2 blocked all the time. Thus the minimum number of displayed "B"s is zero.

d) The following events can occur

Process 1 operates, m=2, prints A, n = 1
Process 2 operates, m=2, prints BC, n = 1
Process 2 operates, m=2, prints B and pre-empted before printing C, n = 0
Process 1 operates, m=1, prints A, n = 1
Process 3 operates, m=1, prints D, n = 0
Process 2 operates again, m=1, prints C, n = 1
Process 1 operates, m=0, prints A, n = 2
Process 2 operates, prints BC, n = 2
Process 3 operates, prints D, n = 1
Process 3 operates, prints D, n = 0

Thus, sequence is possible.

[P2] This is another example of the use of semaphores for signalling. The server process waits for a request of service by any number of client processes. The server should remain blocked if no request is made. Thus, s must be initially 0. If a client process needs to issue a request it signals the server by the up(s) operation to be unblocked. If c client processes issue requests, the server will loop for c times performing the service for each and then blocks again.

[P3] We use two semaphore m and n initially equal to 0

Process 1	Process 2
….	….
`up(m);`	`up(n);`
`down(n);`	`down(m);`
`f1;`	`f2;`

Exercise: Extend the above solution to the case of three processes and three functions.

[P4] We use a semaphore m initially equal to zero

Thread 1	Thread 2	….	Thread n	Required Thread
….	……		….	`down(m);`
`up(m);`	`up(m);`		`up(m);`	`down(m);`
				….
				`down(m);  // n times`
				`// continue`

Note that there is no need to use more than one semaphore. Also, a solution forcing the n threads to run in particular order will not be correct.

[P5] We use a semaphore m initially equal to zero

Thread 1

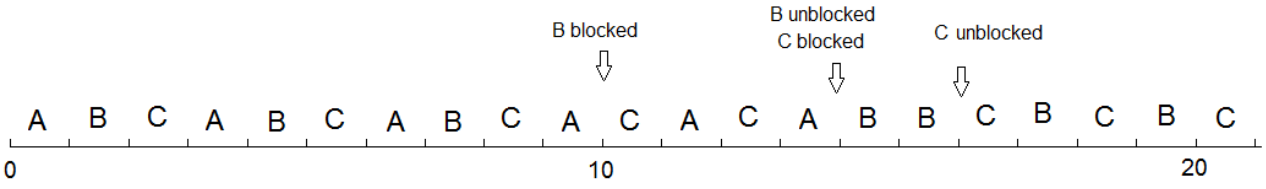
```
….
for (i=0; i<100; i++)
{A[i]= B[i];
 up(m);}
….
```

Thread 2

```
….
for (i=0; i<100; i++)
{down(m);
 C[i]= A[i];}
….
```

Note that if down(m) and up(m) operations are out of the for loops, this will force thread 2 to wait until thread 1 modifies the whole array before making any changes, which results in unnecessary delay.

[P6]



W_{AV}= (8+13+13)/3 = 11.33
