**Q.1.** Consider the too-much-milk problem which may occur when 2 persons are sharing the same house.

|  |  |  |
| --- | --- | --- |
|  | **PersonA** | **Person B** |
| 3:00  3:05  3:10  3:15  3:20  3:25  3:30 | Look in fridge. Out of milk.  Leave for market.  Arrive at market.  Leave market.  Arrive home, put milk away. | Look in fridge. Out of milk.  Leave for market.  Arrive at market.  Leave market.  Arrive home. Too much milk |

In order to avoid too-much-milk problem, they should be synchronized using semaphores. Assume that the shared data are defined as follows:

Shared data:

Semaphore **OKToBuyMilk** = 1;

Int **NoMilk**;

**NoMilk** is true when there is no milk in the fridge. **OKToBuyMilk** is used to achieve mutual exclusion between process A and B. Before buying milk by calling the function “BuyMilk”, each process should check whther the value of OkToBuyMilk is 1. Otherwise, they should wait. An appropriate function call should be used for this purpose which firstly verifies that OhToBuyMilk = 1 and then reduces its value and hence block other processes. After this is done, the process is in its critical section where the value of NoMilk is firstly checked. If there is no milk, then BuyMilk is called. After completion, another appropriate function should be called for ending the critical section.

Taking into account this information, please provide the pseudocode for processes A and B.

**Q.2.** Consider the functions FunctionFirst() and functionSecond() defined as follows:

Shared data:

Semaphore s1 = 1, s2 = 1, d = 1;

Int c1 = 0, c2 = 0;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Void functionFirst(){  wait(s1);  c1 = c1 + 1;  if(c1 == 1) wait(d);  signal(s1);   |  | | --- | | Critical section code |     wait(s1);  c1 = c1 – 1;  if(c1 == 0) signal(d);  signal(s1);   |  | | --- | | Remainder section code |   } | Void functionSecond(){  wait(s2);  c2 = c3 + 1;  if(c2 == 1) wait(d);  signal(s2);   |  | | --- | | Critical section code |     wait(s2);  c2 = c2 – 1;  if(c2 == 0) signal(d);  signal(s2);   |  | | --- | | Remainder section code |   } |

Two different functions involving critical section codes are given above. Assume that there are N processes running in our system which may call functionFirst() or functionSecond() including the semaphore operations **wait()** and **signal().**

1. How many processes may have functionFirst() calls running in their critical sections? What would be the corresponding d value at that time?
2. While one functionFirst() is in its critical section, how many functionSecond() calls can be running in its critical section? What would be the corresponding d value at that time?

**Assignment**

**Q.3.** Consider a bank where there are N workers and assume that any number of **customers** may exit at a time. A worker waits for a customer and then provides service. If no worker is free, then a customer should wait. After completing service to a customer, the worker process should update **wrkr.** Similarly, each customer should update **cstmr** as soon as it arrives. Using two semaphore variables **wrkr** and **cstmr**, synchronize workers and customer processes. Do not forget to initialize semaphore variables appropriately.

Semaphore cstmr = …………………., wrkr = ………………..;

|  |  |
| --- | --- |
| **Worker process:**  do{  …………….  Provide service;  …………….  }while (true); | **Customer process:**  do{  …………….  …………….  get service;  }while (true); |
|  |  |