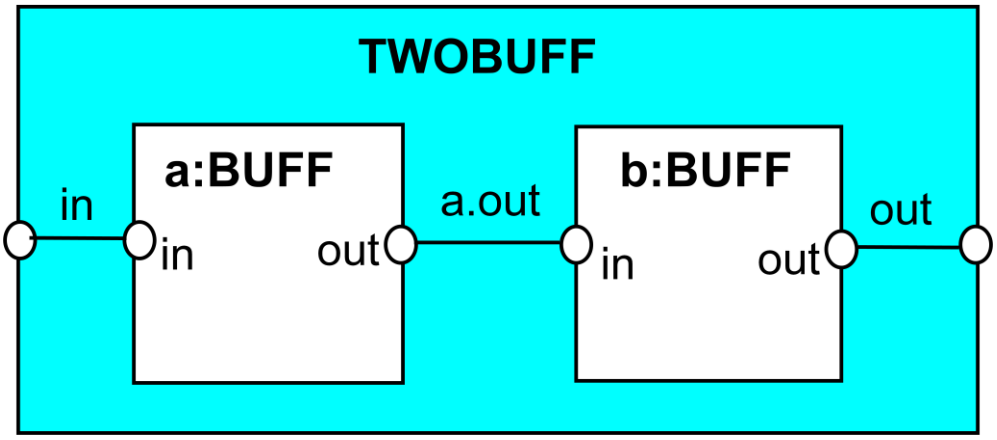
Content: Lecture 5 (shared objects and mutual exclusion)

Sample Exercises:

[Please conduct these as an interactive discussion, rather than an evaluation. Please also make it clear to the students that they are not being evaluated for their performance in these exercises, so that they are not afraid to make mistakes while answering.]

1. Consider a one cell buffer
   BUFF = in(x) -> out(x) -> BUFF



Suppose we want to put the two cells together in the above fashion. What will be the process equation describing the TWOBUFF process.

Answer:

TWOBUFF = (a:BUFF/{in/a.in} || b:BUFF /{a.out/b.in, out/b.out})@{in,out}

Another solution is

TWOBUFF = (BUFF/{x/out} || BUFF/{x/in})@{in,out}

2. Consider the following process equations from lecture slides

RESOURCE = acquire -> release -> RESOURCE.

USER = printer.acquire -> use -> printer.release -> USER.

PRINTER_SHARE = (a:USER || b:USER || {a,b}:PRINTER:RESOURCE).

Describe all behaviors of the composed process.

Answer: The state model compositions are shown below.

3. In class we discussed two methods of lock – to embed the lock inside the object, or to make every "user" of a shared object impose a locking discipline.

Choice 1 :

```
class SynchronizedCounter extends Counter {
…
   synchronized void increment() {
      Counter.increment();
   }
}
```

Choice 2: `synchronized(counter) {counter.increment();}`

Comment on the use of the choices in presence of the recursive locking scheme of Java.

Answer: Choice 2 is less safe, since it requires all user threads of shared objects to remember to impose a locking discipline. However, it is less dependent on the recursive locking scheme since the top level method call is locked. So, even if the increment method is recursive, the code from choice 2 can execute without running into a deadlock.

4. Let x be a shared integer variable. Consider two threads executing the following code

```
while (x< 1){                    while (x >= 1){
        x++;                             x--;
}                                }
```

Initially x is 1. What can you say about the termination of the left hand side loop? What are the possible number of times it can execute? Assume *each line of code* is executed atomically.

Also, comment on the possible termination / non-termination of the program as a whole.

Answer: Executing 0 times is possible, if the left thread executes at the very beginning and the loop is exited.

Executing once is also possible, if the right thread executes one iteration of its loop, exits the loop, and the then the left thread executes as follows

```
                                while(x >= 1)    // x is now 1
                                      x--   // x is now 0
                               while (x >= 1)   // exits
while (x < 1)  // x is now 0
      x++   // x is now 1
while (x < 1) // exits
```

So, in this scenario both loops terminate.

We could construct a scenario where both loops go on forever by alternating one iteration of each loop.

```
                                while  (x >= 1)  // x is now 1
                                      x--    // x is now 0
while (x < 1)  // x is now 0
     x++  // x is now 1
                                while (x >= 1) // x is now 1
                                      x--  //  x is now 0
while (x < 1) // x is now 0
      x++  // x is now 1
…….
```

5. Consider the following multi-threaded program. flag is a boolean variable initialized to false. x is an integer variable initialized to zero. Assume that all condition evaluations and assignments are executed atomically.

```
Thread 1            |  Thread 2
-------------------------------------------
while (!flag){      |  while (x == 0){
   x = 1 - x;}      |           x = x; }
                    |   flag = true;
```

(i) Construct an interleaving where the program does not terminate.
(ii) Construct an interleaving where the program terminates.
(iii) What are the possible values of x when the program terminates? Justify your answer.

(i) If the first thread is not scheduled at all, only the second thread is scheduled and the program does not terminate.

(ii) The following interleaving leads to termination.

```
while (!flag){
x = 1 - x // x == 1
                                  while (x == 0) // exits loop
                                  flag = true
while (!flag) // exits loop
```

(iii) In the above interleaving we have x == 1 when the program terminates.

We can also have x == 0 as follows.

```
while (!flag){
x = 1 - x // x == 1
                                   while (x == 0) // exits loop
while (!flag){
x = 1 - x // x == 0
                                   flag = true;
while (!flag) // exits loop
```