We will wait 10 minutes until 10:40 AM for all students to join into the meeting.

We will start the tutorial at **10:40 AM**.

CS 3SD3 - Concurrent Systems Tutorial 6

Mahdee Jodayree

Before we continue.

- During the presentation, Students can ask any slide-related questions.
- Any non-slide-related questions must be asked at the end of the presentation.

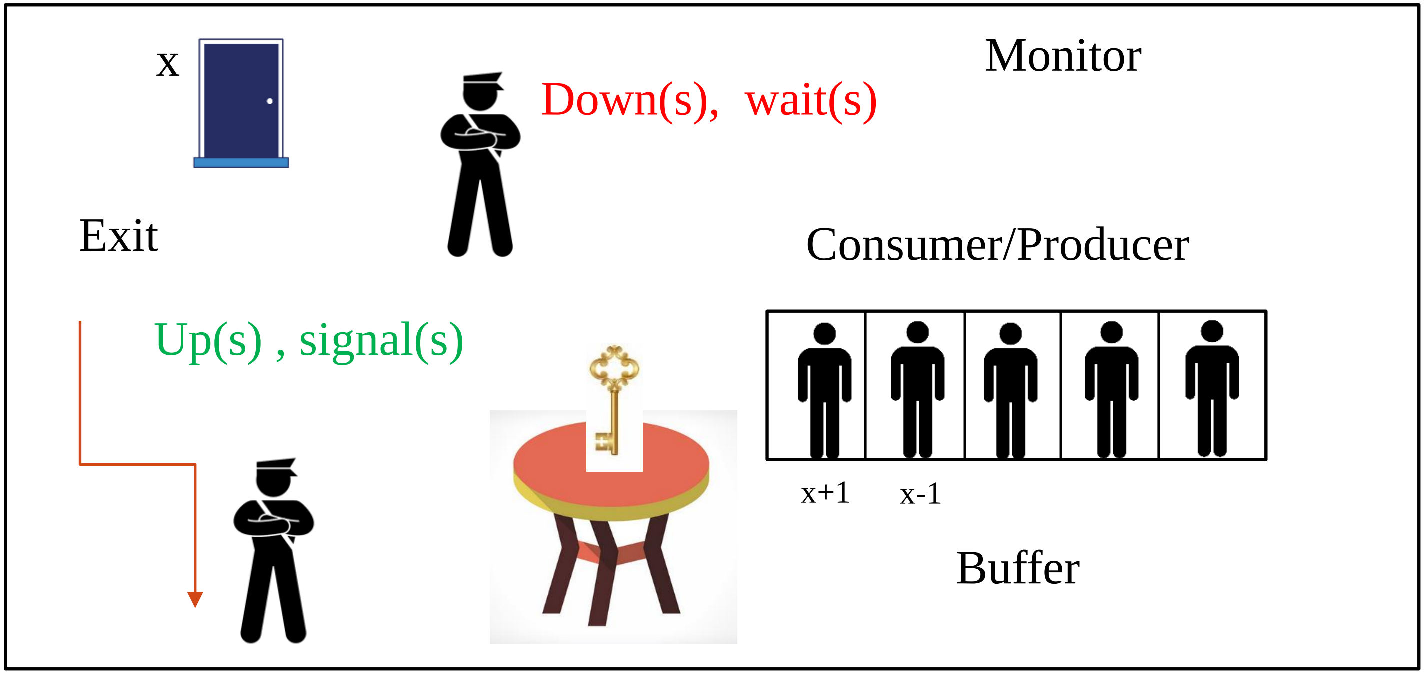
Outline

- Announcements / Reminders
- Buffers.
- Deadlock vs Starvation.
- Safety property.
- Nested Monitor problem.
- Two Examples with solution from the midterm with a solution.

Announcements

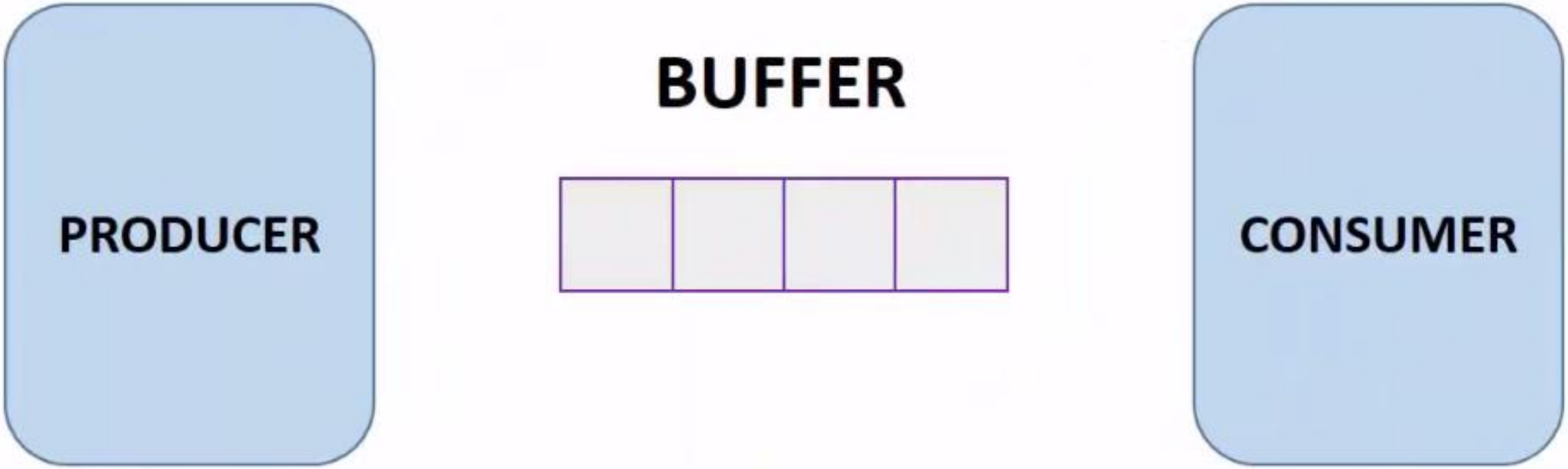
- Assignment 1 is marked.
- Total Question 2 was ten even though its weight is only out of 3.
- The total mark is given for question 12 was 28 even though that question's weight is out of 18.
- This means there were 17 bonus marks for assignment 1. For assignments 2 and 3, there will be no bonus marks.
- For Future reference, your code (i.e. LTSA/FSP, Java, etc.) should NOT contain any errors for future assignments and midterms.
- TA's should be able to copy/paste code and run it without any modifications.
- For assignment #1, many students did not submit the working code.
- **It is not TA's job to troubleshoot/debug their code.**
  - Tony marked Q1, Q2, Q6, Q7. Q2 had 7 bonus marks.
  - Jatin marked Q3 - Q4
  - Mahdee marked Q5 - Q8
  - Weijie Liang marked Q9 - Q12, Q12 had 10 bonus marks
- If you find any discrepancy in your assignment 1's marking, please email the appropriate TA.

Buffer

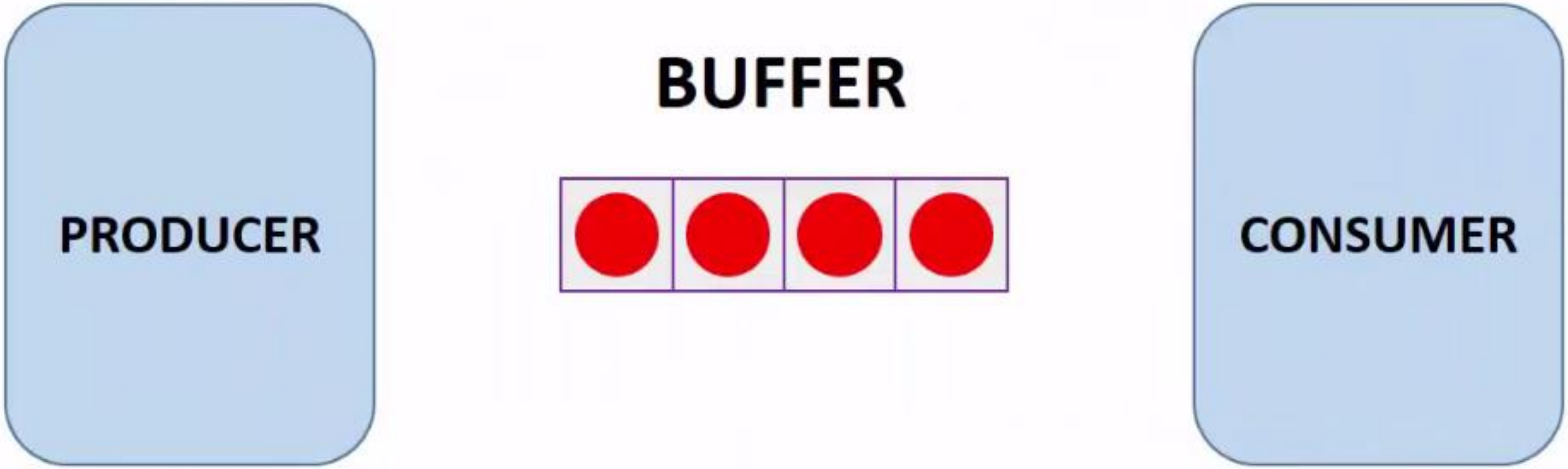


x
Monitor
Down(s), wait(s)
Exit
Consumer/Producer
Up(s) , signal(s)
x+1
x-1
Buffer

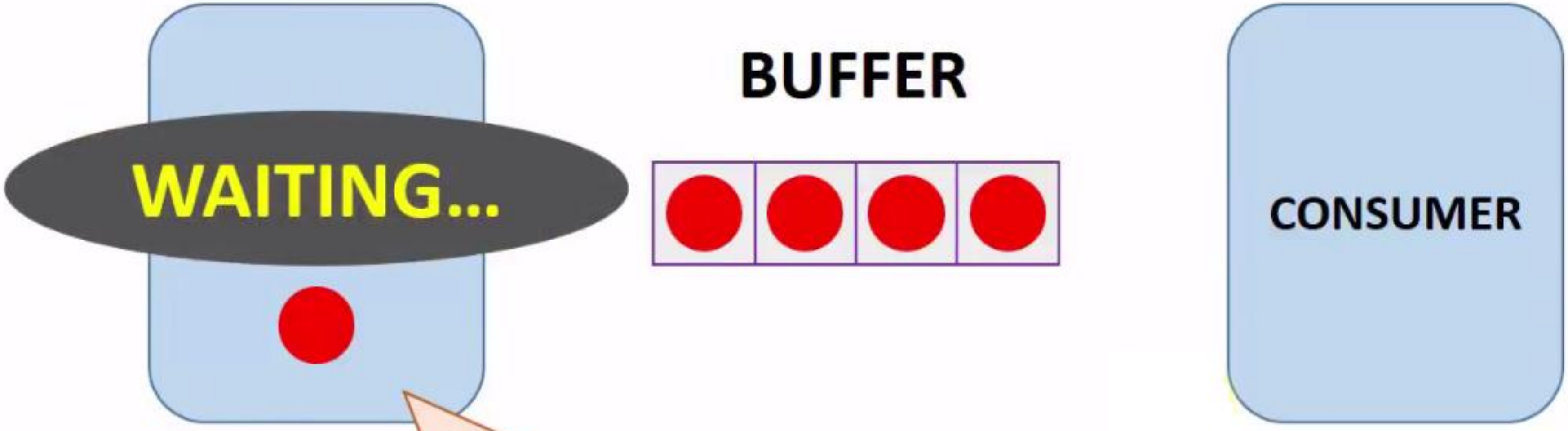
PRODUCER – CONSUMER PROBLEM



PRODUCER – CONSUMER PROBLEM

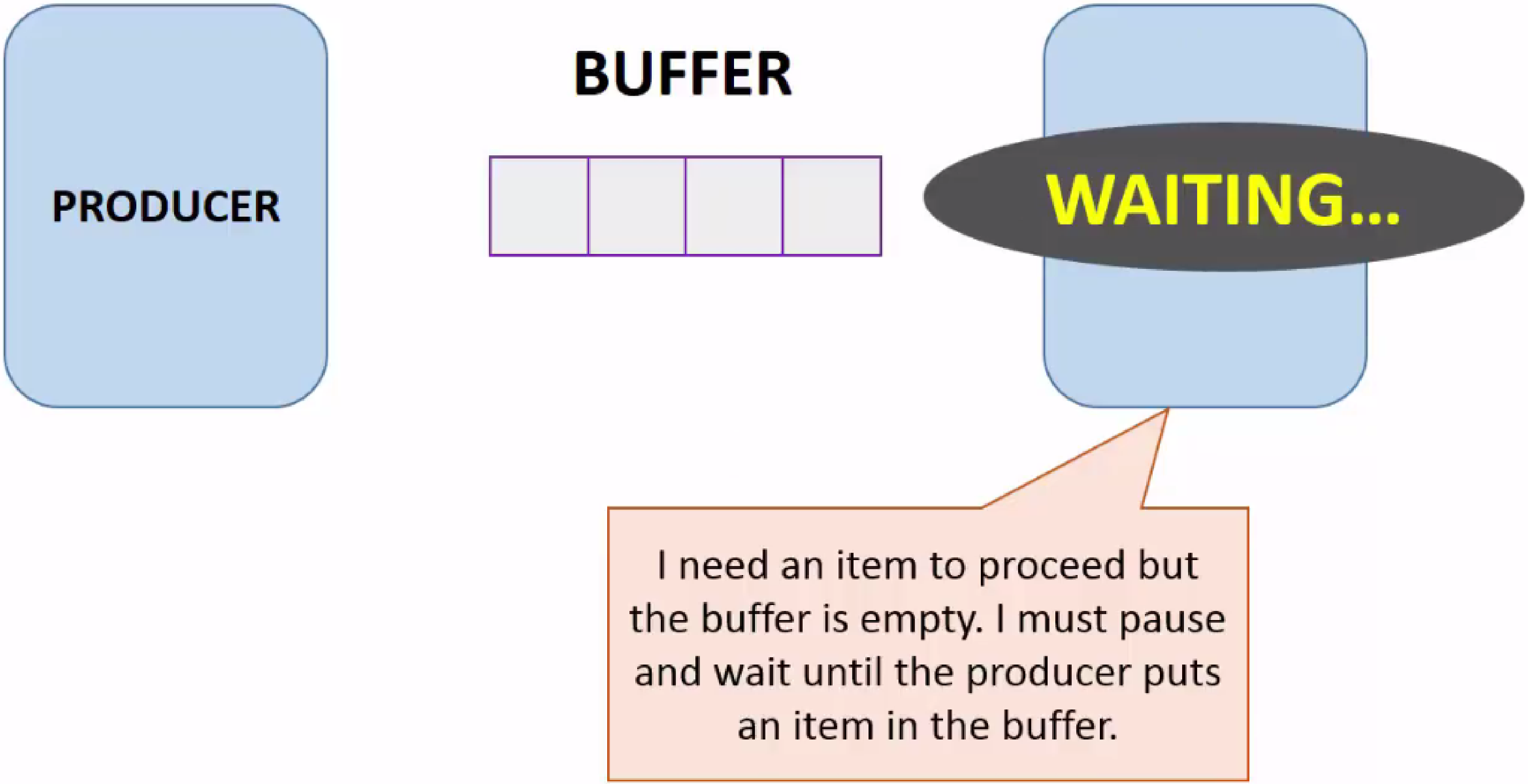


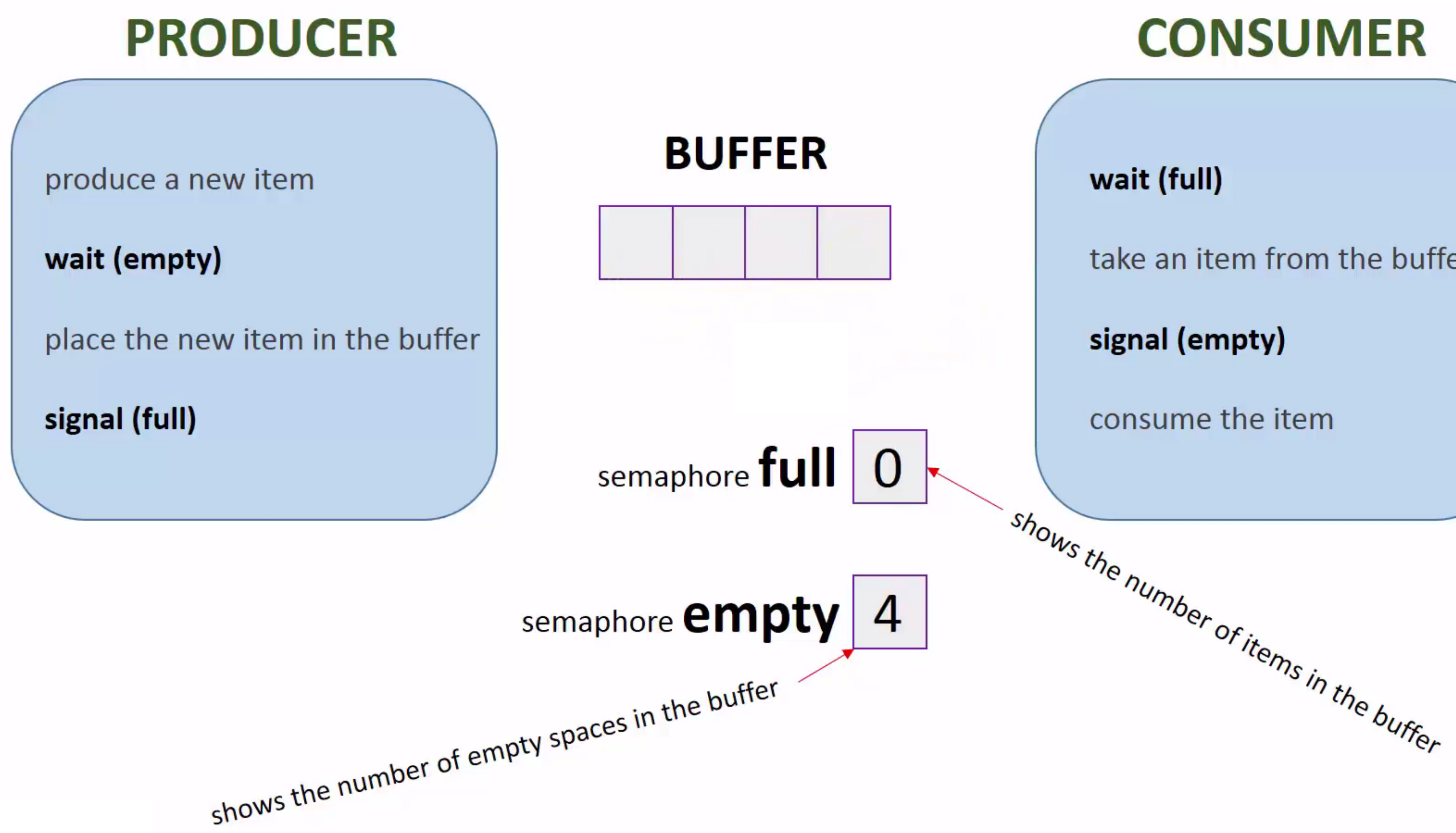
PRODUCER – CONSUMER PROBLEM



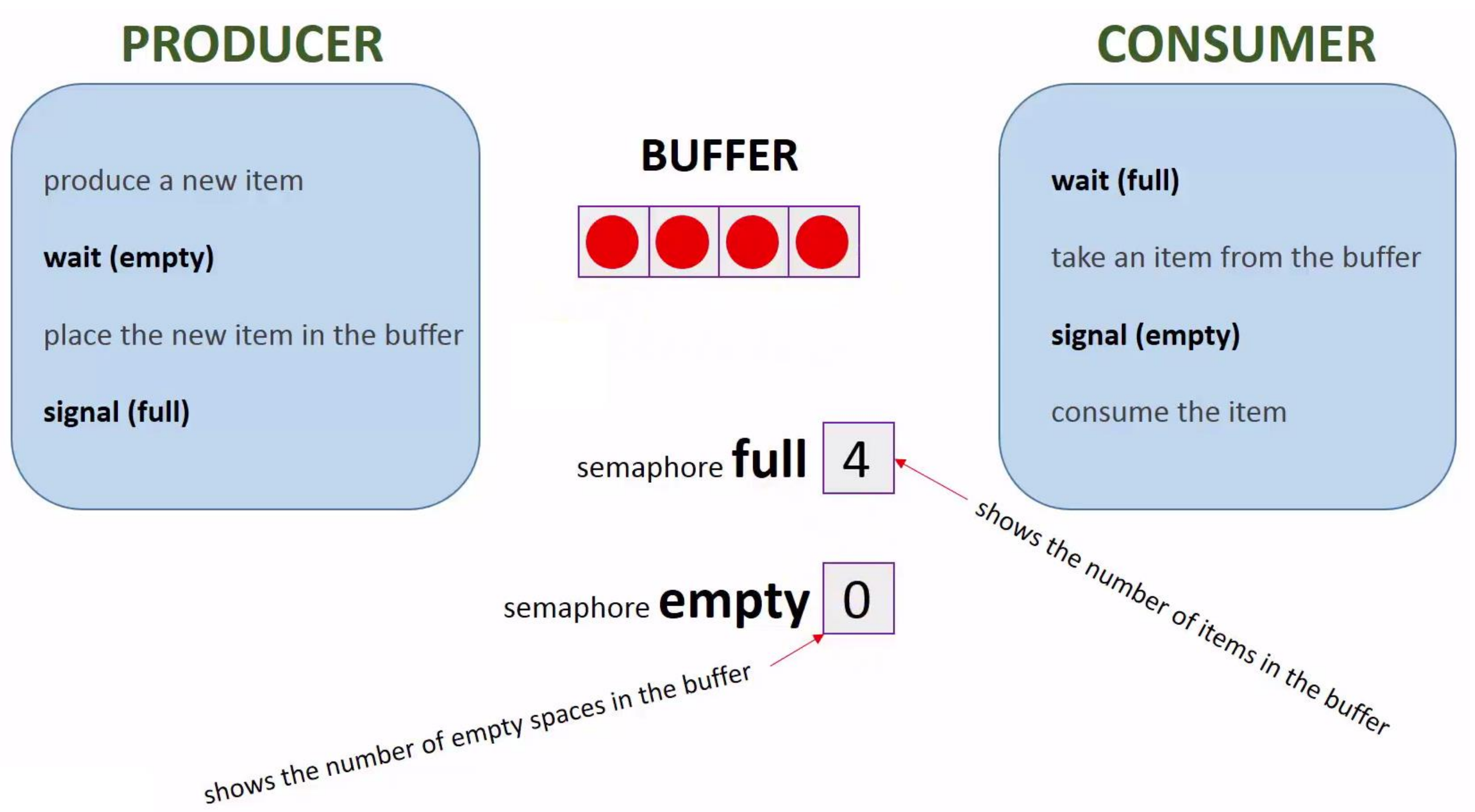
I've made a new item but there is no place in the buffer. I must pause and wait until an empty space appears in the buffer.

PRODUCER – CONSUMER PROBLEM





❖ The producer waits for the semaphore empty
if empty is greater than zero that means there is a place for a new item



The Consumer buffer.

SUMMARY:

- The Producer-Consumer problem says that a producer produces items one by one and places them in a buffer, and a consumer takes items from the buffer one by one. We must arrange both the producer and the consumer in such a way that
  - If the buffer is full, the producer doesn't try to place more items in it but just waits for an empty space to appear.
  - If the buffer is empty, the consumer doesn't try to take an item from it but just waits for a new item to appear.

First we must understand deadlock

- In computer science, deadlock refers to a specific condition when two or more processes are each waiting for another to release a resource, or more than two processes are waiting for resources in a circular chain.
- Example, Two people who are drawing diagrams, with only one pencil and one rule between them. If one person takes the pencil and the other takes the ruler, a deadlock occurs when the person with the pencil needs the ruler and the person with the ruler needs the pencil, before he can give up the ruler. Both requests can’t be satisfied, so a deadlock occurs.

Conditions For Deadlock.

Deadlock can arise if following four conditions hold simultaneously (Necessary Conditions)

1. **Mutual Exclusion:** One or more than one resource are non-sharable (Only one process can use at a time)
2. **Hold and Wait:** A process is holding at least one resource and waiting for resources.
3. **No Pre-emption:** A resource cannot be taken from a process unless the process releases the resource.
4. **Circular Wait**: A set of processes are waiting for each other in circular form.

Deadlock vs. Starvation

Deadlock

Deadlock occurs in a system when **all its constituent processes** are blocked.

Example: Considering the original dining philosophers problem:

```
PHIL = (sitdown -> right.get -> left.get -> eat ->
        left.put -> right.put -> arise -> PHIL).
```

When all philosophers take the right fork and wait for the left one. Because they all have only 1 option to do (taking the left fork), all will wait forever. That is, the whole system cannot do anything else.

Deadlock vs. Starvation

Starvation

Starvation is a concurrent-programming situation in which **an action** is never executed.

Example: Considering the "not so hungry" dining philosophers problem:

```
PHIL = THINK,
THINK = (think -> right.get -> (left.get -> EAT
                        | giveup -> right.put -> THINK),
EAT = (eat -> left.put -> right.put -> THINK).
```

- There is a trace where all philosophers repeatedly take the right fork and give up. That is, the whole system is not blocked but the eat actions of all philosopher processes are never executed.
- There is a trace where a single philosopher p^* repeatedly takes the right fork and gives up. That is, the eat action of philosopher p^* is never executed.

Property

Property

A property is an attribute of a program that is true for every possible execution of that program.

Example: Considering the deadlock-free property and the model of the original dining philosophers problem. Although there exists many sequences does not have deadlock, it does not hold deadlock-free property.

Property

Safety property

A safety property asserts that nothing bad happens during the execution.

- The most important safety properties for concurrent programs are **mutual exclusion** and **deadlock-free**.
- How to state deadlock-free property?

Deadlock-free property

There is no case such that all constituent processes of the system are blocked.

- How to state mutual exclusion property?

Mutual exclusion property

There is no case such that more than one process of the system can access a shared resource at a time.

Property

Liveness property

A liveness property asserts that something good eventually happens.

- The most important liveness property for concurrent programs is the **starvation-free**, requests for shared resources are eventually granted.
- How to state starvation-free property?

Starvation-free property

When a process requests for a shared resource, it is eventually granted.

Property

Q: Can deadlock-free be a liveness property? If it can, how to state it in liveness fashion?

A: Yes, it can.

Deadlock-free property (Liveness fashion)

When a process is blocked, it is eventually released.

Q: What are different between deadlock-free and starvation-free in liveness fashion?

A: They are still different.

- Deadlock-free property simply requires that the block process will eventually released.
- Starvation-free property does not only require the release but also the grant of shared resource to the requesting process. In other words, **if a system is starvation-free, it is deadlock-free.**

Property

Progress property

A progress property asserts that whatever state a system is in, it is always the case that a specified action will eventually be executed.

- Progress properties are the subset of liveness properties.
- How to state starvation-free property in progress fashion?

Starvation-free property (Progress fashion)

Action “A” will eventually be executed, where “A” is an action can only be executed after obtaining shared resources.

Q: Can deadlock-free property be stated in Progress fashion?
A: Yes, it can.

Deadlock-free property (Progress fashion)

Action “A” will eventually be executed, where “A” is a desired action.

Modelling properties in FSP

- Safety properties are modelled by **deterministic processes** which will be concurrently-composed with the system.
- Progress properties are modelled by sets of actions and at least one of these actions will be executed infinitely often.

Buffer and safety property examples.

Do not forget in LTSA tool examples, Under the Chapter 5, there are several examples related to buffer and safety property.

Nested Monitor problem

- Is when a Consumer tries to get a value, but the buffer is empty.
  - It blocks the releases the lock on the semaphore full

then

- Producer tries to put a character into the buffer, but also blocks.

This is called nested monitor problem

Nested Monitors and Semaphores

- Suppose that, in place of using the *count* variable and condition synchronization directly, we instead use two semaphores *full* and *empty* to reflect the state of the buffer.

```
const Max = 5
range Int = 0..Max

SEMAPHORE ...as before...

BUFFER =  (put -> empty.down ->full.up ->BUFFER
          |get -> full.down ->empty.up ->BUFFER
          ).

PRODUCER = (put -> PRODUCER).
CONSUMER = (get -> CONSUMER).

||BOUNDEDBUFFER = (PRODUCER|| BUFFER || CONSUMER
                  ||empty:SEMAPHORE(5)
                  ||full:SEMAPHORE(0)
                  )@{put,get}.
```

Does this behave as desired?

- **It deadlocks after the trace *get*!!!**
- Why? *CONSUMER* tries to get a character, but the buffer is empty. It blocks and releases the lock on the semaphore *full*. PRODUCER tries to put a character into the buffer, but also blocks.
- It is called *nested monitor problem*.

The confusing part about this slide was that

- Producer
  - Empty.down is a lock
  - Put
  - Full.up means the buffer is full
- Consumer
  - Full.down is a lock
  - Get
  - empty.up means the buffer is now empty.

Nested Monitors and Semaphores

- Suppose that, in place of using the *count* variable and condition synchronization directly, we instead use two semaphores *full* and *empty* to reflect the state of the buffer.

```
const Max = 5
range Int = 0..Max

SEMAPHORE ...as before...

BUFFER =  (put -> empty.down ->full.up ->BUFFER
          |get -> full.down ->empty.up ->BUFFER
          ).

PRODUCER = (put -> PRODUCER).
CONSUMER = (get -> CONSUMER).

||BOUNDEDBUFFER = (PRODUCER|| BUFFER || CONSUMER
                  ||empty:SEMAPHORE(5)
                  ||full:SEMAPHORE(0)
                  )@{put,get}.
```

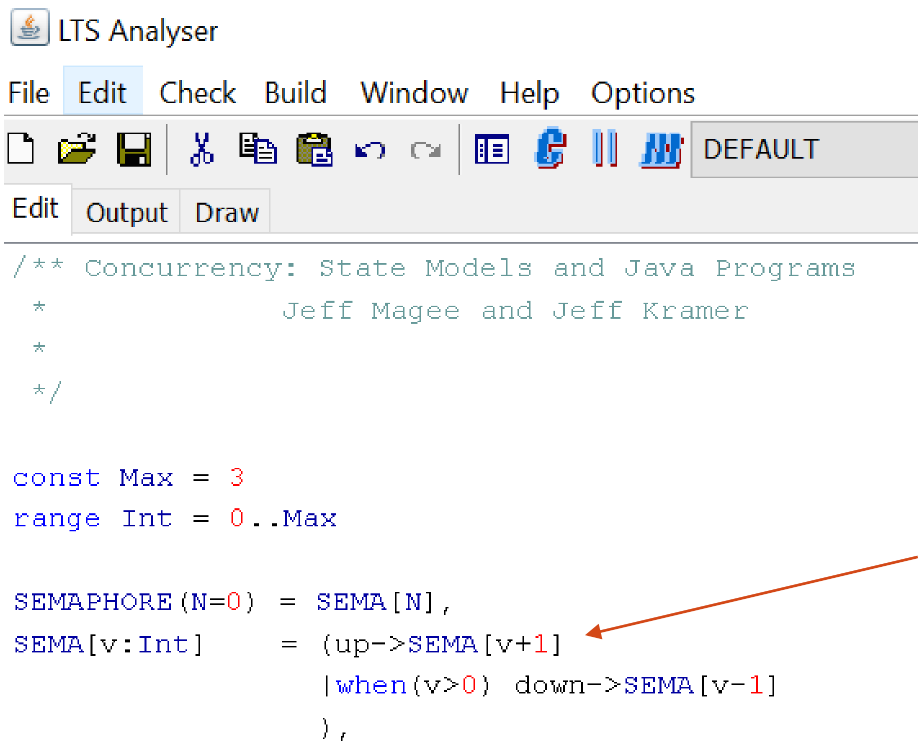
Lock

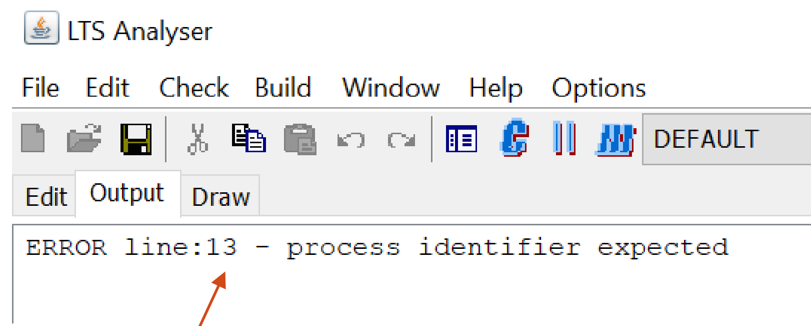
Lock

Does this behave as desired?

- **It deadlocks after the trace *get*!!!**
- Why? *CONSUMER* tries to get a character, but the buffer is empty. It blocks and releases the lock on the semaphore *full*. PRODUCER tries to put a character into the buffer, but also blocks.
- It is called *nested monitor problem*.

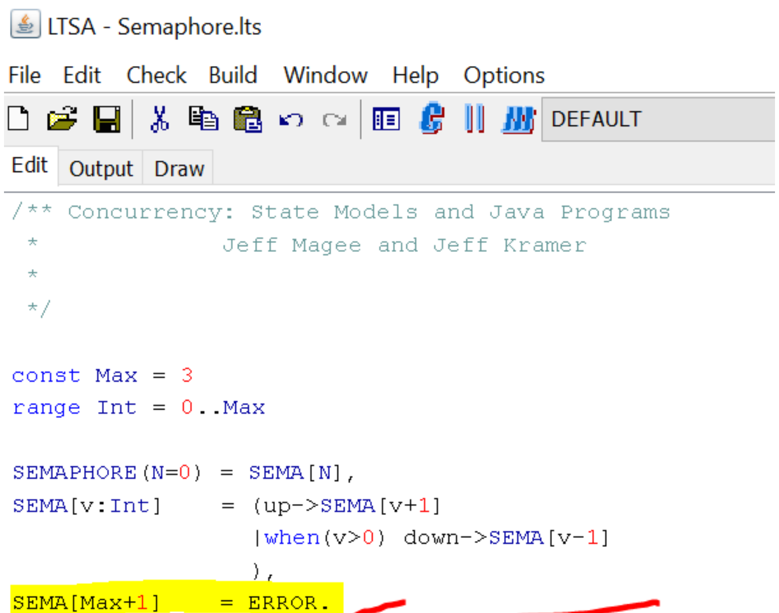
Safety Property Example

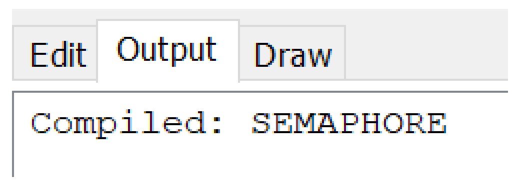


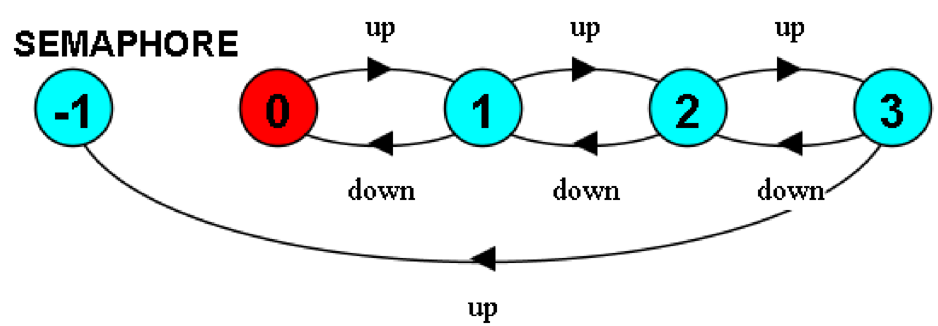


What if Semaphore exceeds the Max value?

Safety Property Example







ERROR with
Capital letters

From lecture notes

LTSA - BoundedBuffer_nestedSema.lts

File Edit Check Build Window Help Options

BOUNDEDBUFFER

Edit Output Draw

```
const Max = 5
range Int = 0..Max

SEMAPHORE(I=0) = SEMA[I],
SEMA[v:Int]    = (up->SEMA[v+1]
                 |when(v>0) down->SEMA[v-1]
        ).

BUFFER =  (put -> empty.down ->full.up ->BUFFER
          |get -> full.down ->empty.up ->BUFFER
          ).

PRODUCER = (put -> PRODUCER).
CONSUMER = (get -> CONSUMER).

||BOUNDEDBUFFER = (PRODUCER|| BUFFER || CONSUMER
                 ||empty:SEMAPHORE(5)
                 ||full:SEMAPHORE(0))
          @{put,get}.
```

What if Semaphore is 5 and we try to increment it to 6? This code shows an error message, however many codes would not run until you add a safety property.

```
Compiled: PRODUCER
Compiled: BUFFER
Compiled: CONSUMER
Warning - SEMA.6 defined to be ERROR
Compiled: SEMAPHORE(5)
Warning - SEMA.6 defined to be ERROR
Compiled: SEMAPHORE(0)
```

LTSA - BoundedBuffer_nestedSema.lts

File Edit Check Build Window Help Options

BOUNDEDBUFFER

Edit Output Draw

```
const Max = 5
range Int = 0..Max

SEMAPHORE(I=0) = SEMA[I],
SEMA[v:Int]    = (up->SEMA[v+1]
                 |when(v>0) down->SEMA[v-1]
           ),
SEMA[6] = ERROR.
BUFFER =  (put -> empty.down ->full.up ->BUFFER
          |get -> full.down ->empty.up ->BUFFER
          ).

PRODUCER = (put -> PRODUCER).
CONSUMER = (get -> CONSUMER).

||BOUNDEDBUFFER = (PRODUCER|| BUFFER || CONSUMER
                  ||empty:SEMAPHORE(5)
                  ||full:SEMAPHORE(0))
          @{put,get}.
```

Edit Output Draw

```
Compiled: PRODUCER
Compiled: BUFFER
Compiled: CONSUMER
Compiled: SEMAPHORE(5)
Compiled: SEMAPHORE(0)
```

Do not Forget to change the dot to a comma, because we need to add a safety property.
We need to add a dot after the safety property.
The word ERROR must be in capital letters.

Without the safety property

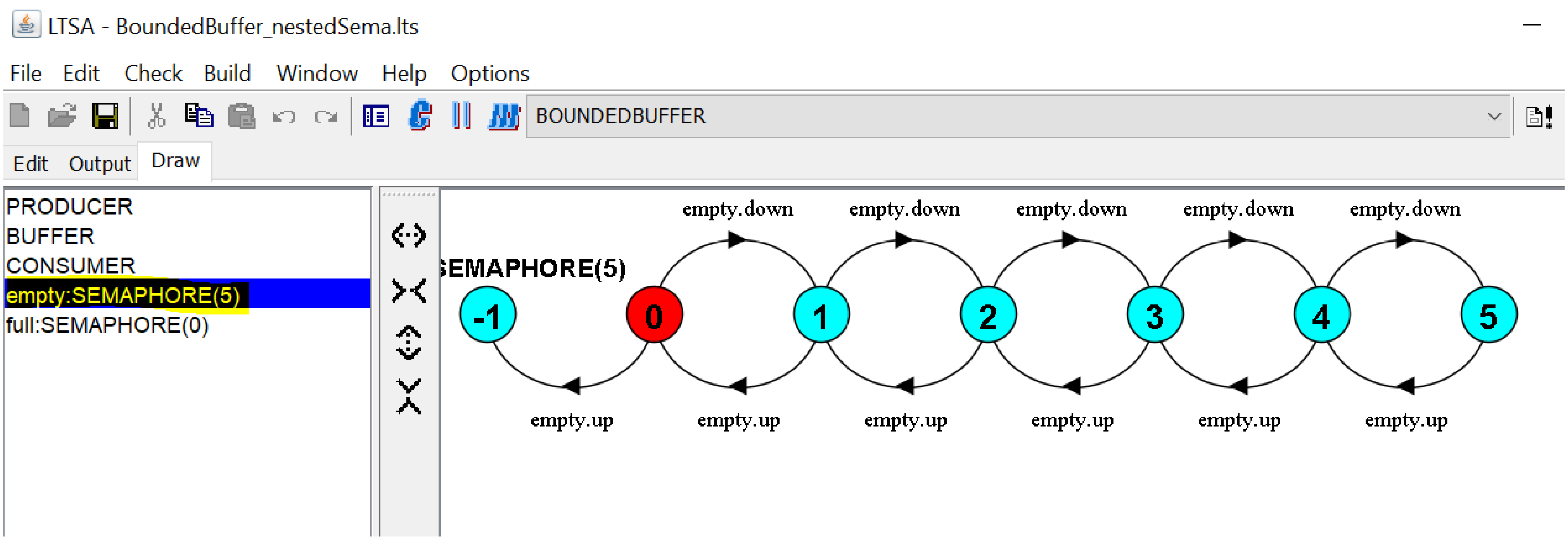
The output would have an error messages and in some cases it would not run.

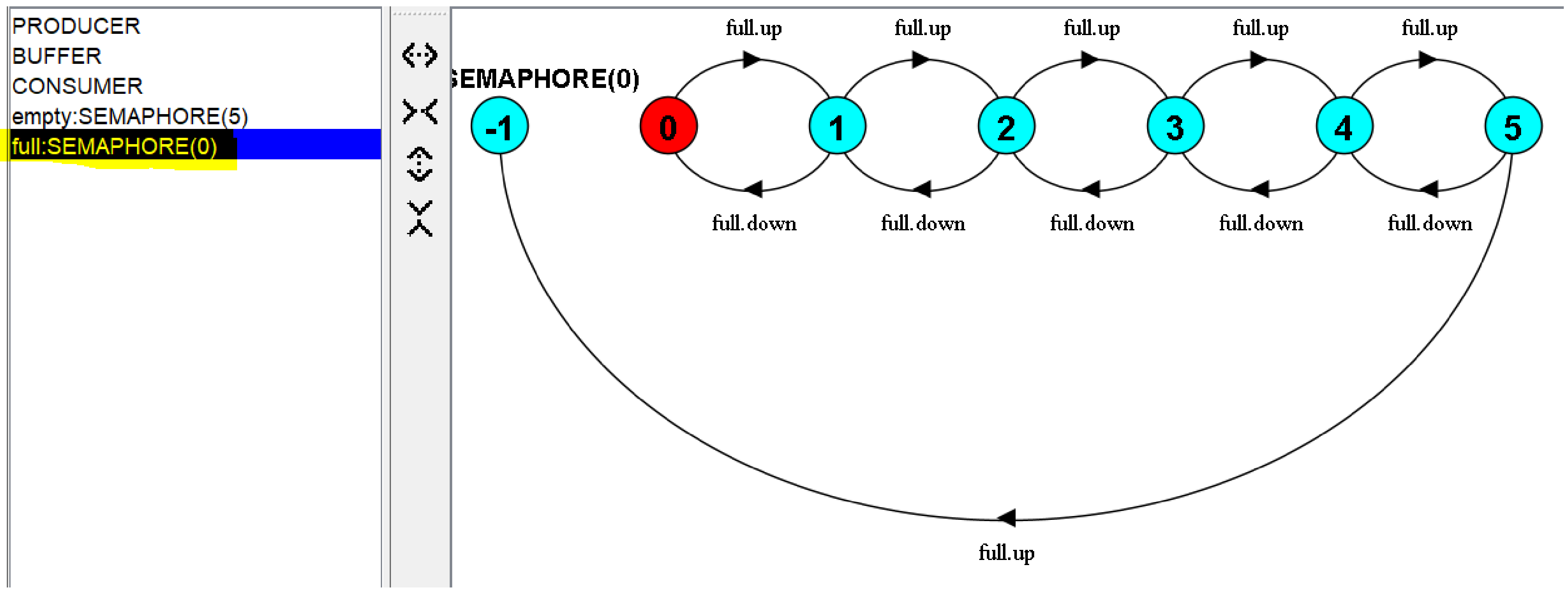
Edit Output Draw

```
Compiled: PRODUCER
Compiled: BUFFER
Compiled: CONSUMER
Warning - SEMA.6 defined to be ERROR
Compiled: SEMAPHORE(5)
Warning - SEMA.6 defined to be ERROR
Compiled: SEMAPHORE(0)
```

After adding the safety property.

This example runs without the safety property but in many cases , You must add safety property or else it would not run.




PRODUCER
BUFFER
CONSUMER
empty:SEMAPHORE(5)
full:SEMAPHORE(0)
SEMAPHORE(0)
-1
0
1
2
3
4
5
full.up
full.up
full.up
full.up
full.up
full.down
full.down
full.down
full.down
full.down
full.up

Fixed nested (on the right side).

LTSA - BoundedBuffer_nestedSema.lts

File Edit Check Build Window Help Options

BOUNDEDBUFFER

Edit Output Draw

```
*/
const Max = 5
range Int = 0..Max

SEMAPHORE(I=0) = SEMA[I],
SEMA[v:Int]    = (up->SEMA[v+1]
                 |when(v>0) down->SEMA[v-1]
          ).

BUFFER =  (put -> empty.down ->full.up ->BUFFER
          |get -> full.down ->empty.up ->BUFFER
          ).

PRODUCER = (put -> PRODUCER).
CONSUMER = (get -> CONSUMER).

||BOUNDEDBUFFER = (PRODUCER|| BUFFER || CONSUMER
                  ||empty:SEMAPHORE(5)
                  ||full:SEMAPHORE(0))
          @{put,get}.
```

LTSA - BoundedBuffer_fixedSema.lts

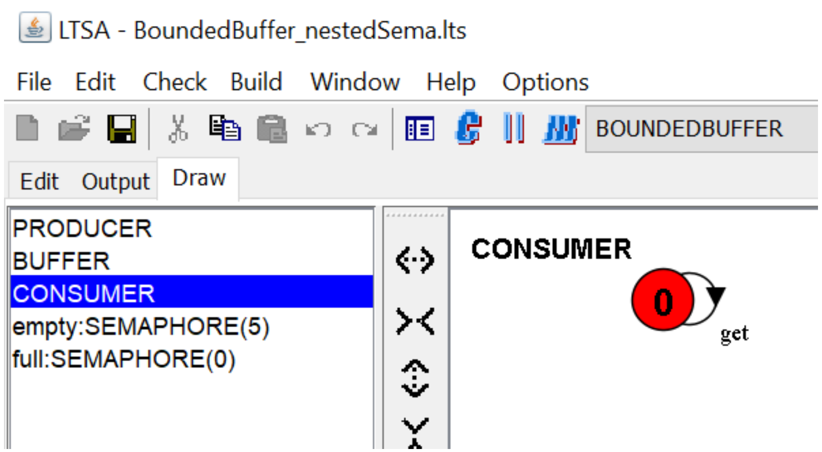
File Edit Check Build Window Help Options

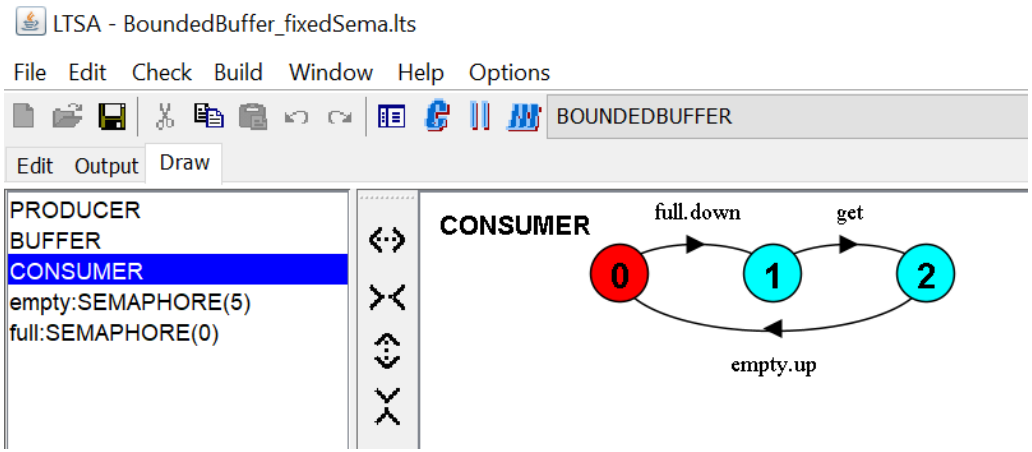
BOUNDEDBUFFER

Edit Output Draw

```
*/
const Max = 5
range Int = 0..Max

SEMAPHORE(I=0) = SEMA[I],
SEMA[v:Int]    = (up->SEMA[v+1]
                 |when(v>0) down->SEMA[v-1]).

BUFFER =  (put -> BUFFER
          |get -> BUFFER
          ).

PRODUCER =(empty.down->put->full.up->PRODUCER).
CONSUMER =(full.down->get->empty.up->CONSUMER).

||BOUNDEDBUFFER = (PRODUCER|| BUFFER || CONSUMER
                  ||empty:SEMAPHORE(5)
                  ||full:SEMAPHORE(0))
          @{put,get}.
```

This improves the Producer and Consumer.





Sample question.

5.[10] Consider two philosophers system. Each philosopher either think, or eats cookies or drinks cola. However there is only one cookie dispenser and only one cola distributor (they are separate machines), so only one philosopher can get cookies or cola at a time, the other must wait.

a.[5] Model this system using FSP.

b.[5[Model this system using any kind of Petri nets.

Solutions (not unique, syntax might differ):

a. We have processes: PHIL, COOKIE_DISP and COLA_DIST

PHIL = (think → PHIL | get_cookie → eat_cookie → PHIL | get_cola → drink_cola → PHIL)
COOKIE_DISP = (give_cookie → COOKIE_DIS)
COLA_DIST = (give_cola → COLA_DIST)

||PHIL_SYST = (a: PHIL || b: PHIL || {a,b}:: COOKIE_DISP || {a,b}:: COLA_DIST)
/{a.get_cookie/a.give_cookie, b.get_cookie/b.give_cookie,
a.get_cola/a.give_cola, b.get_cola/b.give_cola}

Sample question.

6.[14] Two workers W1 and W2 working separately need two different tools, say *drill* and *clamp* to do some work (say precise drill). In order to do the job, each worker needs get both tools. However, due to the nature of work, W1 needs to get drill first and clamp second, while W2 needs to get clamp first and drill second. We obviously want to avoid a situation when W1 grabs drill and waits for clamp, while W2 grabs clamp and waits for drill.

a.[7] Model the situation described above carefully avoiding deadlock with FSP.

b.[7] Model the situation described above carefully avoiding deadlock with Petri nets (any kind)

Solutions:

a. A possible solution (W1 is 'more important' worker):

TOOL = (get → put → TOOL)

W1 = (get_drill → get_clamp → job → release_drill → release_clamp → W1)

W2 = (get_clamp → GET_DRIL),
GET_DRILL = (get_drill → job → release_clamp → release_drill → W2
| release_clamp → W2)

||TWO_WORKERS = (w1:W1 || w2: W2 || drill:TOOL || clamp:TOOL)
/ { {w1,w2}.get_drill/drill.get, {w1,w2}.release_drill/drill.put,
{w1,w2}.get_clamp/clamp.get, {w1,w2}.release_clamp/clamp.put }

Any Questions?