Exercises of Parallelism - Monitors

# Exercise 1 – Producer/Consumer Model (seances 1 and 2)

We consider the Producer-Consumer model in which two families of processes access a shared buffer to deposit (Producer processes) or withdraw (Consumer processes) messages. The buffer is managed in a circular way. Withdrawals are done in the order of the deposits.

We propose to write a Monitor to synchronize the access to the common buffer according to the different variants stated below.

This monitor is specified as follows:

| Monitor Prod\_Cons:  put(. . . ) -- used by a Producer process to deposit a message  get(. . . ) -- used by a Consumer process to withdraw a message |
| --- |

## Variant 1 – N-cells buffer, messages of a single type

This is the basic variant, in which the buffer capacity is up to N messages and the policy applied is the one described above. Only a single type of messages is considered.

## Variant 2 – Two types of messages, alternated deposits

Now, the messages considered may be of two types (for example, black/white or recto/verso...).

A Producer deposits messages of a certain type. Messages are put in an alternating way. Withdrawals are done according to the same policy as above.

## Variant 3 – Two types of messages, choice of a message type when consumed

In this variant, Producers are still of two types but the deposits are not necessarily alternate. Consumers are also of two types and a Consumer specifies the type of message it wants to withdraw. However, withdrawals are still done in the order of the deposits.

## Variant 4 – Producers always produce two times (bonus)

In this variant, there is no more type, but when a producer produces, the next produced element must come from it. For this variant, only focus on steps 1. 2. and 3., i.e. do not give the code. In case of conflict, there is a priority for consumer.

## Questions

For each variant:

1. Specify the monitor.
2. Specify the blocking and unblocking conditions of a Producer process and a Consumer process.
3. Deduce the state variables and the “condition” variables used in the monitor.
4. Give the “code” (algorithm) of the monitor.

# Exercise 2 – Reader-Writer Model (seance 3)

The reader-writer model schematizes a situation encountered in the management of shareable files or in the access to databases.

Two families of processes access this information. Readers only want to consult the information. Writers want to modify this information. The readers can therefore access the information in parallel, whereas writers must access it in mutual exclusion.

The behaviors of these processes are therefore as follows:

| Reader | Writer |
| --- | --- |
| Loop on  ...  Ask to read  Read  Warn reading is finished  ... | Loop on  ...  Ask to write  Write  Warn writing is finished  ... |

We propose to write a monitor managing the access to the common resource according to the policy defined previously and according to the following variants.

The specification of the monitor is as follows:

| Monitor Reader\_Writer:  startRead()  endRead()  startWrite()  endWrite() |
| --- |

## Variant 1

When no readers are reading, readers and writers have the same priority. On the other hand, as soon as a reader is reading, all other readers requesting it can also read, since readings can be done in parallel, regardless of the number of editors waiting.

When a writer is writing, no other client can access the resource (neither reader nor writer).

When the writer is done writing, it tries to activate a writer over the readers.

## Variant 2 – Priority of writers over readers

We want to give priority to writers so that the information available is not obsolete for readers.

When a writer requests access to the resource, it should therefore get it as soon as possible. Of course, it is not possible to interrupt ongoing reading or writing. Likewise, it has no right to go before other writers in the condition (who arrived before it and have not yet been accepted and thus waiting in the condition). On the other hand, it has priority over the waiting readers.

## Variant 3 - More equitable access management

To what erroneous situations can variants 1 and 2 lead?

What solution can be proposed to correct such situations?

In this variant, a writer who finishes writing must give priority access to all the readers waiting at that moment, and not to the next writer as specified in variant 1. On the other hand, readers who arrive later (after this writing request) must respect the rule of priority of writers over readers as expressed in variant 2.

## Questions :

For each variant :

1. Specify the monitor.
2. Specify the blocking and unblocking conditions of a Reader process and a Writer process.
3. Deduce the state variables and the “condition” variables used in the monitor.
4. Give the “code” (algorithm) of the monitor.

# Bonus Exercise

Propose a way to improve the Producer/Consumer system so that we keep the properties (start consuming in order of arrival, i.e. the oldest one) but with the additional property to be able to produce or consume multiple elements at the same time. For this variant, propose steps 1. 2. and 3. of the methodology.

# Exercise 3 – Factory example (seance 4), exam-like exercise

## Order processing

On the occasion of the Christmas holidays, your favorite store has decided to make available to its faithful customers a certain number (NB\_COUNTERS) of special counters to take care of their orders. The principle of operation of these counters is as follows:

* A customer waits for a free counter, places his order there, and then waits for it to be ready.
* The processing of an order consists of a number (NB\_STEPS) of successive steps and specialized employees are assigned to each of these steps. Step i of the order processing can only start when the employee in charge of the previous step (i-1) has finished his task.
* When the order has been processed, the customer leaves the counter, fully satisfied with

this new service.

For example, in practice, the store provides three counters for its customers and the processing of an order consists of four steps: (1) listing the ordered items, (2) picking up the ordered items, (3) wrapping them with gift wrap, and (4) charging the customer.

## Problem

We consider two types of processes: Customer and Employee.

* A Customer process places an order at a free counter and waits for this order to be processed before leaving this counter. At most NB\_COUNTERS orders can be processed in parallel.
* An Employee process, specialized in step i, takes charge of the first pending order of this step, completes its part of the work before returning the processed order to its original counter. He can then take over a new order. When the last step has been executed on an order, the Customer process that formulated this order can resume its execution.

It is assumed that the function :

* apply\_step (nb\_step: int, counter: int, counters: list<order\_t>)

which can be used by an Employee process in order to accomplish its part of the work (i.e. the *nb\_step* step) on a given order placed on the counter *counter* and thus make a modification to it.

Finally we have the function init\_couter(nb\_counters) that initialize the counter (a list of order\_t)

We propose to synchronize, using a monitor named OrderManagement, Customer processes and Employee processes so that orders are processed as efficiently as possible.

Specification of this monitor is as follows :

| Monitor OrderManagement:  # Used by client  start\_command (order: order\_t) -> int  end\_command(counter: int)  # Used by employees  start\_step (step:int) -> int # returns the counter position  end\_step (counter: int) |
| --- |

The behavior of the Customer and Employee processes is as follows:

| Process Client:  # Go to the store  c = start\_command()  # walk around  end\_command(c)  # Go home satisfied | Processus Employee (my\_step:int):  while (True):  counter = startStep(my\_step)  apply\_step(counter)  end\_step (counter) |
| --- | --- |

## Questions :

1. Specify the blocking and waking conditions of a Client process and a Employee process.
2. Deduce the state variables and the “condition” variables used in the monitor.
3. Give the “code” (algorithm) of the monitor.
   * val in tab : returns true if val is in tab
   * tab.index(val) : returns the position of val in tab (or crashes if it does not exist)