TPs Parallelism : Synchronization using monitors

Upload all your codes on Moodle

Start by looking into **ping\_pong.py** then **even\_odd.py** and finally **tennis.py** (they include the 4 steps of the methodology)

# Exercise 1 - Producer/Consumer

Add example when it lock and lock due to conditidion.

## Part 1 - Version without synchronization

1. Retrieve the code named m1\_prodCons\_base.py provided in Moodle.
2. Run the application for different parameter values.

Examples of execution:

**python3 prod\_cons\_base.py 3 3 1**

**python3 prod\_cons\_base.py 2 2 2**

**python3 prod\_cons\_base.py 4 4 4**

and see the problems with unshared variables.

1. Modify this program to replace the constants **nb\_times\_prod** and **nb\_times\_cons** (number of deposits made by a producer and number of withdrawals made by a consumer) with two additional parameters given at runtime.
2. Modify this program to use shared variables between processes. Note the access conflicts on the shared variables.

*(it should not work, just base for next stop)*

*the last question about modify should change the way to say.*

## Part 2 - Producer/Consumer Model - Basic version

From the provided code, **add the monitor synchronization** - using mutual exclusion locks and conditions - needed to implement the basic producer-consumer version (seen in class and TD) in which withdrawals occur in the order of deposits. As in the tennis court example, build a monitor in a python class.

* Run the application for different configurations to ensure the validity of the results obtained.

## Part 3 - Producer/Consumer Model - Alternate Deposits

Modify your code (**keeping** the previous version) to implement the version where the producers alternately drop their messages in the buffer (see V2 of the TD). For the tests, use an even number of producers.

## Part 4 - Producer/Consumer Model - On Demand Withdrawals

Modify your code (**keeping** the previous version) to implement the version where consumers request to withdraw a message of a certain type and their withdrawals must be done in the order of the deposits (see V3 of the TD). We will no longer force the alternation of deposits.

## Part 5 - To go further: Producer/Consumer Model - Dual Deposits

Modify your code (**keeping** the previous version) to implement a version where at each deposit, a producer deposits two messages in an (**unbreakable** and) **consecutive** way in the buffer (so, the **producer()** function will have to be adapted). The removal of messages will be done in the same way as in the basic version, i.e. in the order of the deposits (final exam of 2017).

# Exercise 2 - Reader-Writer

We consider parallel activities (processes) that simulate readers and writers having read or write access to a common file. Reads can be done in parallel but writes can only be done in mutual exclusion.

Ensuring monitor synchronization, write the **start\_\*** and **end\_\*** operations so that a writer gives priority to another writer and has priority over pending read requests (V2 of the TD).

The code skeleton is provided in the file **lect\_red\_base.py**

* Writer processes write in a file five time their own number, printing each time they write
* Reader processes read it the same file the whole content, printing each read, then the aggregated read data

# Exercise 3 - Management of a small road

Parallel activities (processes) are used to simulate the behavior of vehicles traveling in a certain direction. We consider a portion of a single lane on which, in order to avoid collisions, only vehicles going in the same direction can circulate.

The behavior of each vehicle is as follows:

|  |
| --- |
| vehicle(direction):  loop n\_time :  Drive normally on the large road in the direction direction  enter\_road(direction)  Drive on the small road in the direction direction  exit\_road() |

Write the **Road** monitor to synchronize the accesses to the single lane so as to allow an unlimited number of vehicles to travel in the single lane as long as they go in the same direction.

Test these operations by writing an application in which **N1** vehicles going in one direction and **N2** vehicles going in the other direction coexist (**N1** and **N2**, **n\_time** are the parameters of the application).

Examples of possible tests:

**N1** = 2, **N2** = 5, **n\_times** = 2

**N1** = 0, **N2** = 3, **n\_times** = 2

**N1** = 4, **N2** = 0, **n\_times** = 3

The code skeleton is provided in the file **vu\_base.py**

**will remove it.**

# Exercise 4 - To go further

1. Implement the Reader/Writer of Exercise 2 with the last version seen in TD: Writer give reader priority when finishing and Reader are blocked if a writer is waiting (but they are not prevented to start when another reader start)

# 

# Exercice 5 - Management of access to voting booths

The goal of this exercise is to implement better conditions, including priority and the empty() method.

We want to simulate the behavior of **NBV** voters sharing access to **NBP** polling booths (with **NBV** much higher than **NBP**). The behavior of a voter consists in arriving at the polling station, entering a polling booth, preparing his envelope and coming out of the booth to place it in the ballot box.

To do this, implement an **ExtendedCondition** class (using classic **Conditions** internally) that offers the following capabilities:

• Ability to handle high priority (i.e. **wait(0)**) in addition to normal priority (i.e. **wait(1)** or **wait()**). We limit ourselves here to the case of conditions with 2 priority levels.

• Possibility to check if the waiting list is empty (i.e. **.empty()**)

Start from the code skeleton provided in the file **extended\_condition.py**

The constraints for the voting booth are as follows:

* A voting booth can only be used by one voter at a time.
* Some voters have priority access to these voting booths (for example, disabled access). This allows them, in the event of a surge, to move ahead of "non-priority" voters.

Questions:

1. Propose a specification for a monitor that manages competing accesses of **NBV** voters to these **NBP** booths.
2. Propose blocking and unblocking conditions as well as shared variables and conditions associated with the monitor.
3. Implement this monitor using the **ExtendedCondition** class previously defined.
4. Write an application in which **NBV** voter processes use the operations of this monitor to synchronize their accesses to existing **NBP** booths. We will consider that 1/ratio voters have priority (for example by considering that voters for which **id\_voter % ratio == 0** have priority). Use the skeleton of the previous exercises as a starting point. The arguments must be