Université Toulouse III – Paul Sabatier
118 route de Narbonne
31062 Toulouse cedex 9

Lab work – n°3
Processus and monitor synchronization

Documentation

The concept of "monitor" can be used to synchronize processes using conditions (in the **multiprocessing** module). See the documentation available in Moodle.

Exercice 1 – Reader-Writers – FIFO version

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.

The behaviors of the processes are as follows:

A Reader	A Writer

Loop on {
…
`start_read()`
Read the shared file;
`end_read()`
…
}
	Loop on {
	…
	`start_write()`
	Modify the shared file;
	`end_write()`
	…
	}

Ensuring monitor synchronization, write the **`start_*`** and **`end_*`** operations so that processes run in order of arrival, and readers arriving before the first waiting writer run in parallel.

To do this, implement an **`ExtendedCondition`** class (using classic **`Condition`**s 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()`**)

Remark : Start from the code skeleton provided in the file *tp2_lectred_base.py*

Remember that in Python a function can take a default value for its arguments:

```
def wait(priority = 1):
```

If **`wait`** is called with no argument then **`priority`** will be 1, otherwise **`priority`** will take the value of the argument.

[Code to upload on moodle]

Exercice 2 – Management of access to voting booths

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.

The constraints 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.

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 defined in the previous exercise.
4. Write an application in which NBE voter processes use the operations of this monitor to synchronize their accesses to existing NBI booths. We will consider that 1/ratio voters have priority (for example by considering that voters for which **`id_voter % ratio == 0`** have priority)

[Code to upload on moodle]

Reminder: If the displays are too fast, it is possible to delay the execution of a process for a few microseconds or nanoseconds using the primitives:

time.sleep(secondes)

See the online manual for their use:

https://docs.python.org/fr/3/library/time.html#time.sleep

We can use a randomly generated value (see the functions random.rand and random.seed) to vary the waiting times from one process to another.

https://docs.python.org/3/library/random.html

But, **be careful**, the timeout is not there to solve the problems of concurrent access to shared variables. In other words: any execution of a parallel application must give a consistent result without timeout!