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

Travaux pratiques – n°3
Processus et synchronisation de type « moniteur »

Documentation

Le concept de « moniteur » peut être utilisé pour synchroniser des processus en utilisant des conditions (dans le module **`multiprocessing`**). Voir la documentation à votre disposition sous Moodle.

Exercice 1 – Lecteurs-Rédacteurs – Version FIFO

On considère des activités parallèles (processus) qui simulent des lecteurs et des rédacteurs ayant accès à un fichier commun en lecture ou en écriture. Les lectures peuvent se faire en parallèle mais les écritures ne peuvent se faire qu'en exclusion mutuelle.

Les comportements des processus sont les suivants :

Un lecteur	**Un rédacteur**

Boucler sur {
…
`start_read()`
Lire le fichier partagé;
`end_read()`
…
}
	Boucler sur {
	…
	`start_write()`
	Modifier le fichier partagé;
	`end_write()`
	…
	}

En assurant une synchronisation de type moniteur, écrire les opérations **`start_`*** et **`end_`*** de manière à ce que les processus s'exécutent dans l'ordre d'arrivée, et que les lecteurs arrivant avant le premier rédacteur en attente s'exécutent en parallèle.

Pour cela implémentez une classe **`ExtendedCondition`** (utilisant en interne des **`Condition`**s classiques) offrant les capacités suivantes :

- Possibilité de gérer la priorité forte (i.e. le **`wait(0)`**) en plus de la priorité normale (ie le **`wait(1)`** ou **`wait()`**). On se limite ici au cas de conditions à 2 niveaux de priorité.
- Possibilité de vérifier si la liste d'attente est vide (i.e. le **`.empty()`**)

Remarque : Repartez du squelette de code fourni dans le fichier *tp2_lectred_base.py*

On rappelle qu'en python une fonction peut prendre une valeur par défaut pour ses arguments :

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

Si **`wait`** est appelé sans argument alors **`priority`** vaudra 1, sinon **`priority`** prendra la valeur de l'argument.

[Code à déposer sous Moodle]

Exercice 2 – Gestion d'accès à des isoloirs

On veut simuler le comportement de NBE électeurs partageant les accès à NBI isoloirs (avec NBE très supérieur à NBI). Le comportement d'un électeur consiste à arriver au bureau de vote, à entrer dans un isoloir, à y préparer son enveloppe puis à ressortir de l'isoloir pour la placer dans l'urne.

Les contraintes sont les suivantes :

- Un isoloir ne peut être utilisé que par un électeur à la fois.
- Certains électeurs ont une priorité d'accès à ces NBI isoloirs (par exemple, un accès handicapé). Ceci leur permet, en cas d'affluence, de passer devant les électeurs « non prioritaires ».

1. Proposer une spécification pour un moniteur gérant les accès concurrents de NBE électeurs à ces NBI isoloirs.
2. Proposer les conditions de blocage et de déblocage ainsi que les variables partagées et les conditions associées au moniteur.
3. Implanter ce moniteur à l'aide de la classe **`ExtendedCondition`** définie dans l'exercice précédent.
4. Écrire une application dans laquelle NBE processus électeurs utilisent les opérations de ce moniteur pour synchroniser leurs accès aux NBI isoloirs existant. On considèrera que 1/ratio électeurs sont prioritaires (par exemple en considérant que les electeurs pour lesquels **`id_electeur % ratio == 0`** sont prioritaires)

[Code à déposer sous Moodle]

Rappel : Si les affichages sont trop rapides, il est possible de temporiser l'exécution d'un processus pendant quelques microsecondes ou nanosecondes à l'aide des primitives :

time.sleep(secondes)

Voir le manuel en ligne pour leur utilisation :

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

On peut utiliser une valeur générée aléatoirement (voir les fonctions random.rand et random.seed) pour varier les délais d'attente d'un processus à un autre.

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

Mais, **attention**, la temporisation n'est pas là pour résoudre les problèmes d'accès concurrents à des variables partagée. En d'autres termes : toute exécution d'une application parallèle doit donner un résultat cohérent **sans** temporisation !