TP 2
Concurrence et synchronisation – INFO3 S5
Moniteurs

Objectifs du TP : Reprendre les éléments théoriques vus en cours sur les moniteurs, puis en TD, et les appliquer à la conception de solutions en langage Python aux problèmes posés par la concurrence de processus s'exécutant en parallèle

1 Un Sémaphore de comptage défini par moniteur

Un sémaphore de comptage est défini par les éléments suivants :

- Un compteur entier
- Une liste d'attente fifo
- 2 opérations atomiques `P()` et `V()` (en exclusion mutuelle)

1.1 Lisez le programme source Python contenu dans de fichier `semaphore moniteur.py`, qui reprend en Python l'*exercice 1 du TD 2*, et exécutez-le. Que constatez-vous ?

1.2 Complétez le code Python de ce programme en traduisant le *pseudo code* implémentant un sémaphore de comptage à l'aide de moniteur traité dans l'*exercice 1 du TD 2*.

1.3 Puis vous testerez votre solution sur les 2 exemples proposés (EM et alternance) avec différentes valeurs initiales de sémaphore (ex :0, 1, 2).

NB : on utilisera les mutex/verrous `threading.Lock` (`Lock()`, `acquire()`, `release()`), et les conditions `threading.Condition` (`Condition(mutex)`, `wait()`, `notify()`) du module `threading`

NB : Le langage de programmation Python ne dispose que de pseudo-moniteurs où l'ex*clusion mutuelle (EM)* doit être gérée manuellement avec un sémaphore binaire (mutex) dans chaque *point d'entrée*.

2 L'alternance

On souhaite que les 2 opérations `Ping()` et `Pong()` respectent les contraintes suivantes :

- Exclusion mutuelle
- On débute par `Ping()`
- `Ping()`et `Pong()` doivent alterner strictement (`Ping-Pong-Ping-Pong-Ping-Pong-`…..)

2.1 Lisez le programme source Python contenu dans de fichier `alternance-moniteur.py`, qui reprend en Python l'*exercice 2 du TD 2*. Exécutez-le. Que constatez-vous ?

2.2 A partir de la solution en *pseudo-code* de l'*exercice 2 du TD 2*, *implémentez en Python* la solution à base de *moniteur* permettant de reproduire ce comportement d'alternance.

2.3 Testez votre solution sur différentes simulations, en changeant le contenu de la liste `NomsThreads` et la variable entière `NbCoups`.

2.4 Dessinez le *chronogramme* de l'évolution des 2 processus en repérant l'ordre des opérations `wait()` et `notify()` sur les variables condition.

2.5 Dans cette solution, `Ping` doit nécessairement débuter l'alternance. Proposez une modification permettant au premier processus quel qu'il soit (`Ping` ou `Pong`) de débuter l'alternance.

3 Le problème des Producteurs Consommateurs

Le `Tampon t` est une ressource commune aux processus, qui est munie d'un ensemble d'opérations de manipulation `Deposer`(),`Retirer`()….

On souhaite que les opérations `Deposer`() et/ou `Retirer`()r sur le tampon respectent les contraintes suivantes :

- C1 : Exclusion mutuelle des opérations `Deposer`() et/ou `Retirer`() → mutex
- C2 :Attente des consommateurs si le tampon est vide (`EstVide`()) → condition
- C3 :Attente des producteurs si le tampon est plein (`EstPlein`()) → condition

3.1 Lisez le programme source Python contenu dans de fichier `producteur_consommateur_moniteur.py`, qui reprend en Python l'*exercice 2 du TD 2*. Exécutez-le. Que constatez-vous ?.

3.2 Complétez le code Python afin d'implémenter votre solution par moniteur, en reprenant les solutions en *pseudo-code* vue dans l'*exercice 2 du TD 2*, afin de respecter les 2 contraintes C1 et C2.

3.3 Testez votre solution sur différentes configurations en modifiant les variables `tailleMax`, `nomsThreads` et `nbCoups`. Vérifiez que les *assertions logiques* contrôlant les opérations `wait()` et `notify()` sont correctes, que les configurations indésirables sont bien évitées, et qu'il n'y a pas d'interblocage.

3.4 Dessinez le *chronogramme* de l'évolution de chaque processus en repérant l'ordre des opérations `wait()` et `notify()` sur les variables condition.

3.5 Répéter les questions 3.2 et 3.3 pour résoudre les 3 contraintes C1 et C2 et C3

4 Une usine d'assemblage d'aéroplanes

7 processus `Carlingue` `Aile` `Moteur` `Roue` `Carlingue1Ailes2` `Carlingue1Ailes2Roues3` `Aeroplane` del'usine d'assemblage d'aéroplanes fonctionnent de manière autonome et en parallèle.

Cependant différentes contraintes de synchronisation doivent être respectées

- La chaine de montage `Carlingue1Ailes2` ne peut fonctionner que si au moins 2 ailes et une carlingue ont été produites par `Carlingue Aile`
- La chaine de montage `Carlingue1Ailes2Roues3` peut fonctionner que si au moins 3 roues et un assemblage `Carlingue1Ailes2` ont été produits par `Roue Carlingue1Ailes2`
- La chaine de montage `Aeroplane` peut fonctionner que si au moins 2 moteurs et un assemblage `Carlingue1Ailes2Roues3` ont été produits par `Moteur Carlingue1Ailes2Roues3`

4.1 Lisez le programme source Python contenu dans de fichier `aeroplane-moniteur-ProdCons.py`, qui reprend en Python l'*exercice 3 du TD 2*. Exécutez-le. Que constatez-vous ?.

4.2 Implémentez en Python la solution vue en TD utilisant des tampons Producteurs Consommateurs. Testez-la sur différentes situations, en vérifiant le bon ordonnancement des opérations d'assemblage. Pour cela vous pouvez modifier les variables : `nbAvionsPrevus` et `tailleMaxTamponsChaines.`

4.3 Implémentez en Python dans le fichier `aeroplane-moniteur.py` la solution à base de moniteur vue en TD.

4.4 Testez votre solution sur différentes configurations. Dessinez le *chronogramme* de l'évolution de chaque processus en repérant l'ordre des opérations `wait()` et `notify()` sur les variables condition. Vérifiez que l'ordre est conforme à vos prévisions.