TD 1
Concurrence et synchronisation – INFO3 S5
Sémaphores

Objectifs du TD : Reprendre les éléments théoriques vues en cours sur les sémaphores, et les appliquer à la conception de solutions aux problèmes posés par la concurrence de processus s'exécutant en parallèle

1 L'exclusion mutuelle

Soit 2 processus exécutant les programmes P1 et P2 partageant une variable globale commune RC (ressource critique) :

```
Processus : P1
    Global : RC
    Repeter
        RC := RC +1
    Jusqua Faux
    .....
```

```
Processus : P2
    Global : RC
    Repeter
        RC := RC +2
    Jusqua Faux
    .....
```

1.1 Expliquez pourquoi il est nécessaire de protéger la variable RC. Vous définirez les notions de *ressource critique*, et de *section critique*, et d'*exclusion mutuelle* en les expliquant sur cet exemple.

1.2 Dessinez le *réseau de Pétri* modélisant ce problème et décrivez la configuration indésirable. Au préalable vous reprendrez les éléments du cours correspondant sur les réseaux de Pétri (*Place, Marque, Transition, Pré/Post-condition*)

1.3 Dessinez le *réseau de Pétri* résolvant ce problème à l'aide de sémaphores. Puis vous discuterez cette solution avec trois valeurs initiales de sémaphore distinctes : 0, 1, 2

1.4 Donnez le *pseudo code* résolvant ce problème à l'aide de sémaphores. Puis vous discuterez cette solution en donnant l'évolution de l'état du sémaphore (compteur et liste d'attente) en fonction des opérations atomiques P() et V(). Vous répéterez cette étude avec trois valeurs initiales de sémaphore distinctes : 0, 1, 2

NB : on supposera qu'un sémaphore S est défini et initialisé à 1 par « `Semaphore S :=1` », et manipulable par les opérations « `P(S)` » et « `V(S)` »

2 L'alternance

Soit les 2 processus P1 et P2 suivants :

```
Processus : P1
    Repeter
        Ping()
    Jusqua Faux
    .....
```

```
Processus : P2
    Repeter
        Pong()
    Jusqua Faux
    .....
```

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 Dessinez le *réseau de Pétri* modélisant ce problème et décrivez la/les configuration/s indésirables/s.

2.2 Donnez le réseau de Pétri résolvant ce problème à l'aide de sémaphores. Puis vous discuterez cette solution avec des valeurs initiales de sémaphore distinctes : 0, 1

2.3 Donnez le pseudo code résolvant ce problème à l'aide de sémaphores. Puis vous discuterez cette solution en donnant l'évolution de l'état du sémaphore (compteur et liste d'attente) en fonction des opérations atomiques P() et V() et des valeurs initiales que vous avez choisies.

3 L'étreinte fatale

Soit les 2 processus P1 et P2 suivants :

```
Semaphore : SC := 1
Semaphore : SF := 1
```

```
Processus : P1
    Repeter
        P(SC)
        Crayon()
        P(SF)
        Feuille()
        Ecrire()
        V(SF)
        V(SC)
    Jusqua Faux
    .....
```

```
Processus : P2
    Repeter
        P(SF)
        Feuille()
        P(SC)
        Crayon()
        Ecrire()
        V(SC)
        V(SF)
    Jusqua Faux
    .....
```

On souhaite que les opérations respectent les contraintes suivantes :

- Exclusion mutuelle de `Crayon()`entre P1 et P2 (un seul crayon disponible)
- Exclusion mutuelle de `Feuille()`entre P1 et P2 (une seule fueille disponible)

Pour cela on utilise 2 sémaphores d'exclusion mutuelle SC et SF pour protéger les opérations comme indiqué dans le pseudo-code précédent.

3.1 Dessinez le *réseau de Pétri* modélisant ce problème. Que constatez-vous ?

3.2 Simulez manuellement l'exécution du pseudo-code en donnant les états successifs des sémaphores conduisant à la situation d'étreinte fatale (deadlock).

3.3 Proposez un *réseau de Pétri* résolvant ce problème à l'aide de sémaphores tout en conservant les 2 sémaphores SF et SC tels qu'ils sont utilisés dans le pseudo-code.

3.4 En dériver le pseudo-code de votre solution.

4 Le problème des Producteurs Consommateurs

Soit les 2 processus Producteur et Consommateur suivants :

```
Global :
        Tampon : t
```

```
Processus : Producteur
    Repeter
        Deposer (t, p)
    Jusqua Faux
```

```
Processus : Consommateur
    Repeter
        p := Retirer (t)
    Jusqua Faux
```

```
Processus : Principal
    InitTampon (t, TailleMax);
    ParDebut
        Producteur;
        Consommateur;
    ParFin
```

```
Type Tampon :
InitTampon (Tampon: t , Entier: tailleMax)
NbProduits (Tampon: t) → Entier
TailleMax (Tampon: t) → Entier
EstVide (Tampon: t) → Booleen
EstPlein (Tampon: t) → Booleen
Deposer (Tampon: t, Produit: p)
Retirer (Tampon: t) → Produit
```

Le tampon est une ressource commune aux 2 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`()
- C2 :Attente des consommateurs si le tampon est vide (`EstVide`())
- C3 :Attente des producteurs si le tampon est plein (`EstPlein`())

4.1 Dessinez le *réseau de Pétri* modélisant ce problème sans synchronisation. Que constatez-vous ?

4.2 Proposez un réseau de Pétri résolvant la contrainte C1 à l'aide de sémaphores. Vérifier qu'aucune configuration interdite n'est atteinte, et qu'aucun interblocage ne se produit.

4.3 En dériver le pseudo-code de votre solution et vérifiez par une simulation manuelle que l'évolution des sémaphores est conforme à votre réseau de Pétri, en particulier que les configurations indésirables sont bien évitées et qu'il n'y a pas d'interblocage.

4.4 Répéter les questions 4.2 et 4.3 pour résoudre les 2 contraintes C1 et C2, puis les 3 :C1 et C2 et C3

5 Chaine d'assemblage d'un aéroplane

Soit les 8 programmes/processus suivants composant une chaîne d'assemblage d'un aéroplane :

```
Semaphore: A compléter
```

```
Processus : Carlingue
    repeter
        Une carlingue est
achevée
```

```
Processus : Carlingue1Ailes2
    repeter
        Un assemblage 1 carlingue avec 2 ailes est
acheve
```

```
Processus : Aile
    repeter
        Une aile est achevée
```

```
Processus : Carlingue1Ailes2Roues3
    Repeter
        Un assemblage 1 carlingue et 2 ailes avec 3
roue est acheve
```

```
Processus : Moteur
    repeter
        Une aile est achevée
```

```
Processus : Aeroplane
    Repeter
        Un aéroplaneest acheve
```

```
Processus : Roue
    repeter
        Une aile est achevée
```

```
Processus : Principal
    ParDebut
        Carlingue1Ailes2
        Carlingue1Ailes2Roues3
        Aeroplane
        Carlingue
        Aile
        Moteur
        Roue
    ParFin
```

Les 7 processus contrôlant la chaine d'assemblage 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
- La chaine de montage `Carlingue1Ailes2Roues3` peut fonctionner que si au moins 3 roues et un assemblage `Carlingue1Ailes2` ont été produits
- La chaine de montage `Aeroplane` peut fonctionner que si au moins 2 moteurs et un assemblage `Carlingue1Ailes2Roues3` ont été produits

5.1 Dessinez le *réseau de Pétri* modélisant ce problème sans synchronisation. Que constatez-vous ?

5.2 Proposez un réseau de Pétri résolvant ce problème à l'aide de sémaphores. Vérifier qu'aucune configuration interdite n'est atteinte, et qu'aucun interblocage ne se produit.

5.3 En dériver le pseudo-code de votre solution et vérifiez par une simulation manuelle que l'évolution des sémaphores est conforme à votre réseau de Pétri, en particulier que les configuration indésirables ne sont bien évitées et qu'il n'y a pas d'interblocage.