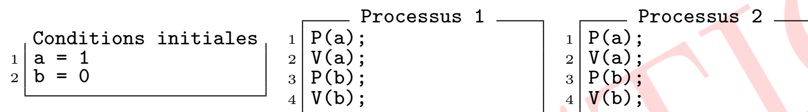
TD3 : Programmation concurrente et synchronisation

RS : Réseaux et Systèmes

Deuxième année

1 Exercices avec les interblocages

★ **Exercice 1:** Considérez le programme suivant, qui utilise deux sémaphores pour obtenir une exclusion mutuelle.



```
Conditions initiales
1 a = 1
2 b = 0
```

```
Processus 1
1 P(a);
2 V(a);
3 P(b);
4 V(b);
```

```
Processus 2
1 P(a);
2 V(a);
3 P(b);
4 V(b);
```

▷ **Question 1:** Montrez que les processus entrent systématiquement en (inter)blocage.

Réponse

L'objectif de la question est d'introduire la notion de diagramme de transition, que l'on a pas vu en cours.
On peut les laisser mariner un moment, puis proposer le tableau ci-contre, puis le compléter ensemble.

	(1 ;0)	(0 ;0)	(1 ;0)	(1 ;-1)	(1 ;0)
V(b)					
	(1 ;-1)	(0 ;-1)	(1 ;-1)	(1 ;-2)	(1 ;-1)
P(b)					
	(1 ;0)	(0 ;0)	(1 ;0)	(1 ;-1)	(1 ;0)
V(a)					
	(0 ;0)	(-1 ;0)	(0 ;0)	(0 ;-1)	(0 ;0)
P(a)					
	(1 ;0)	(0 ;0)	(1 ;0)	(1 ;-1)	(1 ;0)
	P(a)	V(a)	P(b)	V(b)	

Donc, interblocage inévitable car toutes les trajectoires arrivent dans une zone interdite.

Fin réponse

▷ **Question 2:** Proposez une modification des conditions initiales pour éviter ce problème.

Réponse

Mettre b=1 en condition initiale.

Fin réponse

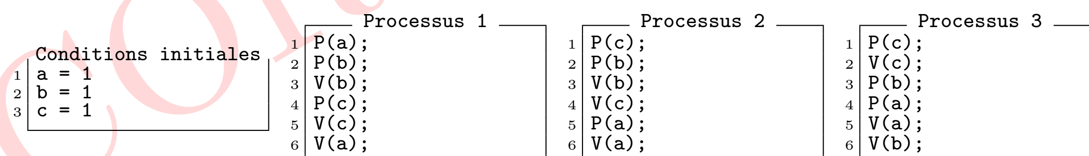
▷ **Question 3:** Montrez graphiquement que votre solution fonctionne.

Réponse

	(1 ;1)	(0 ;1)	(1 ;1)	(1 ;0)	(1 ;1)
V(b)					
	(1 ;0)	(0 ;0)	(1 ;0)	(1 ;-1)	(1 ;0)
P(b)					
	(1 ;1)	(0 ;1)	(1 ;1)	(1 ;0)	(1 ;1)
V(a)					
	(0 ;1)	(-1 ;1)	(0 ;1)	(0 ;0)	(0 ;1)
P(a)					
	(1 ;1)	(0 ;1)	(1 ;1)	(1 ;0)	(1 ;1)
	P(a)	V(a)	P(b)	V(b)	

Fin réponse

★ **Exercice 2:** Considérez le programme suivant, qui se termine parfois par un interblocage.



```
Conditions initiales
1 a = 1
2 b = 1
3 c = 1
```

```
Processus 1
1 P(a);
2 P(b);
3 V(b);
4 P(c);
5 V(c);
6 V(a);
```

```
Processus 2
1 P(c);
2 P(b);
3 V(b);
4 V(c);
5 P(a);
6 V(a);
```

```
Processus 3
1 P(c);
2 V(c);
3 P(b);
4 P(a);
5 V(a);
6 V(b);
```

▷ **Question 1:** Listez les paires de sémaphores que chaque processus cherche à obtenir.

Réponse

L'objectif de la question est de montrer qu'il n'y a pas que les diagrammes de transition dans la vie, et de montrer d'où viennent les interblocages.

CORRECTION
CORRECTION

P1 : (a,b) puis (a,c) mais jamais (b,c) ; P2 : (c,b), mais jamais les autres ; P3 : (b,a).

Fin réponse

▷ **Question 2:** On cherche à éviter les interblocages en ordonnant les réservations selon l'ordre $a < b < c$. Discutez les réservations de chaque processus selon ce critère.

Réponse

P1 fait toutes ses réservations dans l'ordre, pas de problème.

P2 viole l'ordre pour (b,c), mais personne d'autre n'a besoin de cette paire en même temps alors c'est bon.

P3 viole l'ordre pour (c,b), mais c'est bon car il ne les verrouille pas en même temps tous les deux. En revanche, il viole aussi l'ordre pour (a,b) et ce coup ci c'est pas bon car P1 a aussi besoin de cette paire.

Fin réponse

▷ **Question 3:** Proposez une modification pour éviter l'interblocage.

Réponse

Si on inverse les P(b) et les P(a) dans P3, c'est bon car tout le monde respecte alors l'ordre (acb) qui n'est pas plus bête qu'un autre ordre. Donc, le fait que ça respecte un ordre donné est une condition suffisante à l'absence de deadlock, mais ce n'est pas une condition nécessaire (car il peut exister un autre ordre qui serait respecté, lui).

Si on s'y sent, on peut *démontrer* au passage que les requêtes ordonnées garantissent l'absence de deadlock. Pour cela, il faut déjà écrire un peu formellement ce que veulent dire interblocage et requêtes ordonnées, puis montrer que l'un empêche l'autre.

Définition : Interblocage. On regarde juste l'interblocage entre deux processus, mais les cas à plus sont pas foncièrement différents.

$\exists P_1, P_2$ processus, $\exists r_1, r_2$ ressources ($P_1 \neq P_2$ et $r_1 \neq r_2$) t.q.

$$(et) \begin{cases} P_1 \text{ possede } r_1 \text{ et } P_1 \text{ demande } r_2 \\ P_2 \text{ possede } r_2 \text{ et } P_2 \text{ demande } r_1 \end{cases}$$

En clair, il existe une chaîne des dépendances (comme on le répète depuis un moment).

Définition : requêtes ordonnées Soit un ordre des ressources O sur l'ensemble des ressources. Les requêtes respectent cet ordre si $\nexists r_1, r_2$ et P t.q. P possède r_2 et demande r_1 tandis que "$r_1 < r_2$" $\in O$.

En clair, si l'ordre demande à ce que r_1 arrive avant r_2, aucun processus n'acquière r_2 avant de demander r_1

Théorème : requêtes ordonnées ⇒ absence de deadlock

démonstration par l'absurde. Supposons qu'il existe un interblocage (alors qu'un ordre O est respecté).

Donc, $\exists P_1, P_2, r_1, r_2$ t.q. (P_1 possede r_1) et (P_1 demande r_2) et (P_2 possede r_2) et (P_2 demande r_1)

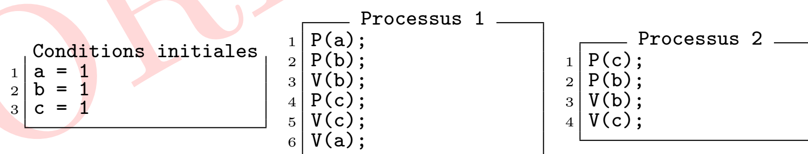
A propos de l'ordre qui est respecté, il contient soit "$r_1 < r_2$" soit "$r_2 < r_1$".

- Dans le premier cas ("$r_1 < r_2$" $\in O$), P_2 a pas le droit d'ordonner ses requêtes comme ça, donc l'ordre est pas respecté, donc absurde, donc boum.
- Dans le second cas ("$r_2 < r_1$" $\in O$), c'est P_1 qui ne respecte pas la règle du jeu, donc absurde.

Dans tous les cas, on atteint l'absurde, donc on a une hypothèse en trop, donc on a démontré ce qu'il fallait.

Fin réponse

★ **Exercice 3:** Considérez le programme suivant.



▷ **Question 1:** Est-ce que les processus peuvent entrer en interblocage ? Pourquoi ?

Réponse

Le processus 1 cherche à obtenir les paires (a,b) puis (a,c) tandis que le processus 2 cherche à obtenir (c,b). Comme les ensembles sont disjoints, cela ne pose pas de problème. On peut faire un diagramme

CORRECTION
CORRECTION

pour s'en convaincre, si on s'ennuie (mais il est plus efficace de remarquer qu'on respecte alors l'ordre (acb)).

Fin réponse

2 Exercices avec les conditions de compétition

★ **Exercice 4:** Dans un lavomatique, on cherche une solution pour permettre de répartir les machines à laver équitablement entre les clients. Considérez le programme suivant. Pour obtenir une machine, chaque client doit utiliser la fonction alloue(). Après usage de la machine, il doit utiliser libère().

Conditions initiales
```
1 #define NMACHINES 5
2
3 Semaphore nlibre = sem(5)
4 int dispo[NMACHINES] = (1,1...1)
```

client
```
1  alloue() {
2    int i;
3    P(nlibre)
4    for (i=0; i < NMACHINES; i++)
5      if (dispo[i] != 0) {
6        dispo[i] = 0;
7        return i;
8      }
9  }
10
11 libère(int machine) {
12   dispo[machine] = 1
13   V(nlibre)
14 }
```

▷ **Question 1:** Ce programme présente une condition de compétition. Laquelle ? Pourquoi ?

Réponse

Il arrive que ce programme attribue la même machine à deux clients, si on se fait interrompre (et que le contrôle passe à quelqu'un d'autre) entre les lignes "if (dispo[i] != 0)" et "dispo[i] = 0"

Pour expliquer, le plus simple est de faire une bande dessinée comme ca :

— Image 1 : P1 s'exécute
  P1 : teste la ligne `if (dispo[5])`
  P2 est gelé
— Image 2 : entre les deux, un context switch s'est produit. P2 s'exécute
  P1 est gelé
  P2 teste la ligne `if (dispo[5])`

— Image 3 : P2 continue
  P1 est gelé
  P2 réalise `dispo[5]:=FALSE` et `return 5`
— Image 4 : P1 est de nouveau ordonnancé
  P1 réalise `dispo[5]:=FALSE` et `return 5`. Il a déjà vérifié que 5 était dispo, alors il ne verra pas que P2 l'a déjà pris. Donc, P1 et P2 utilisent tous les deux la même machine, ce qui viole un invariant.

Fin réponse

▷ **Question 2:** Comment corriger le problème ?

Réponse

Ajouter un verrou pour exclusion mutuelle comprenant ces deux lignes. Soit sur le `for` entier, soit sur chaque itération.

On peut proposer les deux opérations, et parler des performances respectives des solutions. Si je verrouille de plus grosses sections de mon code, je baisse le degré de parallélisme, mais l'opération de verrouillage n'est pas gratuite. Je peux donc perdre plus de temps à verrouiller/déverrouille que ce que je gagne en parallélisant.

Fin réponse

3 Problèmes de synchronisation

★ **Exercice 5: Problème du barbier.** La boutique du barbier est composée d'une salle d'attente contenant n chaises et du salon où se trouve la chaise du barbier. Lorsque le barbier a fini de raser un client, il fait

CORRECTION
CORRECTION

entrer le client suivant dans le salon. Si la salle d'attente est vide, le barbier s'y installe pour dormir. Si un client trouve le barbier endormi, il le réveille. Si non, il s'installe dans la salle d'attente s'il reste de la place (et rentre chez lui sinon).

▷ **Question 1:** Écrivez le code du client et du barbier sans vous préoccuper de synchronisation, mais simplement des opérations que veulent réaliser les processus. En particulier, ignorez pour l'instant que le barbier dort parfois.

Réponse

INITIALISATION

```
int places = NB_CHAISES
```

BARBIER

```
boucle infinie:
  si (place<NB_CHAISES)
    places ++
    coupe_cheveux
```

CLIENT

```
si (place>0)
  place --
  se_faire_couper_les_cheveux
sinon
  rentrer_chez_soi_les_cheveux_longs
```

Fin réponse

▷ **Question 2:** Il s'agit maintenant d'ajouter les synchronisations nécessaires au programme écrit à la question précédente. Le premier problème à résoudre est une condition de compétition entre les clients lorsqu'ils rentrent dans la salle d'attente. Corrigez ce problème.

Réponse

Faut protéger le début du client (le test $nbplace > 0$ et le décrément) par un mutex.

Fin réponse

▷ **Question 3:** Assurez-vous ensuite que le barbier ne commence pas à couper les cheveux tant que le client n'est pas prêt.

Réponse

Faut monter un rendez-vous unilatéral : le barbier avance pas au delà du point de synchro tant qu'il n'a pas reçu l'autorisation du client. C'est la sémaphore privée `client` dans la correction ci-dessous.

Fin réponse

▷ **Question 4:** Enfin, assurez-vous ensuite que le client ne s'assoit pas sur le siège tant que le barbier n'est pas prêt.

Réponse

Faut monter un autre rendez-vous unilatéral dans l'autre sens. C'est la sémaphore privée `barbier` dans la correction ci-dessous. On peut remarquer qu'avec deux rendez-vous unilatéraux, on se retrouve avec un rendez-vous normal. On vient de reconstruire le roméo/juliette du cours.

Fin réponse

Réponse

INITIALISATION

```
clients = sem(0) /* clients dans la salle d'attente */
barbier = sem(0) /* barbiers prêts à couper */
exclusion = sem(1) /* pour l'exclusion mutuelle */
int places = NB_CHAISES
```

CORRECTION

```
BARBIER                        CLIENT
boucle infinie:                P(exclusion)
  P(clients)                   si (place>0)
  P(exclusion)                   place --
  places ++                      V(clients)
  V(barbier)                     V(exclusion)
  V(exclusion)                   P(barbier)
  coupe_cheveux                  se_faire_couper_les_cheveux
                               sinon
                                 V(exclusion)
                                 rentrer_chez_soi_les_cheveux_longs
```

Question à poser à l'oral ensuite : Quel(s) schéma(s) de synchronisation vu en cours est étendu ici ?

Réponse : C'est un producteur/consomateur avec plusieurs producteurs.

Autre question subsidiaire : Dans quelle mesure puis-je déplacer V(exclusion) ?

Réponse : On peut toujours permuter deux V dans un code sans changer la sémantique, mais si jamais on place ce V après le P(barbier), ça veut dire que les autres clients peuvent pas tester s'il reste des places dans la salle d'attente (et y entrer) avant que ce client se soit fait couper les cheveux. Donc, la file d'attente se fait sur la sémaphore `exclusion`, ie la salle est vide et la queue est sur le trottoir.

Fin réponse

★ **Exercice 6: Problème des philosophes.** Cinq philosophes, réunis pour philosopher, ont au moment du repas un problème pratique à résoudre. En effet, le repas est composé de spaghetti qui, selon la coutume de ces philosophes, se mangent avec deux fourchettes. Or, la table n'est dressée qu'avec une seule fourchette par couvert. Les philosophes décident d'adopter le rituel suivant :

- Chaque philosophe prend une place à table.
- Chaque philosophe qui mange utilise la fourchette à sa droite et celle à sa gauche (pas celle d'en face).
- À tout instant, chaque philosophe est dans l'un des états suivants :
  - il mange avec deux fourchettes ;
  - il a faim, et attend la fourchette de droite, celle de gauche ou les deux ;
  - il pense, et n'utilise pas de fourchette.
- Initialement, tous les philosophes pensent.
- Un philosophe qui mange s'arrête en un temps borné.

▷ **Question 1:** Proposez une solution à ce problème. Vous pourrez utiliser l'une des méthodes vues plus haut pour éviter l'interblocage.

Réponse

Si on fait un bête "P(droite), P(gauche)", on court à l'interblocage, car on ne réserve pas dans l'ordre. Le piège est que "gauche" de l'un est "droite" de l'autre.

La méthode la plus simple pour éviter l'interblocage, c'est de faire comme on a fait plus tôt dans le TD : on ordonne les ressources et on fait les réservations dans l'ordre. Ici, on numérote les fourchettes et chaque philosophe prend d'abord celle de plus petit numéro à coté de lui. Ca casse le cycle de dépendance qui débouche sur l'interblocage (car l'une des flèches de réservation et en sens inverse des autres). Ça revient à dire que le philosophe numéro 0 est gaucher, soit dit au passage, d'où la question suivante :

Fin réponse

▷ **Question 2:** On ajoute maintenant l'hypothèse suivante au problème :

- Les philosophes sont droitiers : ils ne prennent pas de fourchette à gauche sans avoir d'abord celle de droite.

Proposez une nouvelle solution tenant compte de cette nouvelle hypothèse. Vous serez amené à utiliser l'autre méthode d'évitement des interblocages vue en cours.

Réponse

Bon, l'ordonnancement des requêtes est maintenant interdit, il faut faire autre chose. D'après le (merveilleux) cours, une autre possibilité est de faire des réservations globales : ne pas bloquer une ressource si on est pas certain d'obtenir les autres dont on a besoin. C'est ce qu'on va faire.

CORRECTION

CORRECTION

Ici, il faut éviter de prendre une fourchette si l'on n'est pas sûr de pouvoir manger. On introduit une variable d'état pour chaque philosophe c[i] qui vaut 0 si le philosophe pense, 1 si il veut manger et 2 si il mange. Le tableau c doit être partagé par tous les processus de façon à ce que chacun puisse tester si ses voisins sont déjà en train de manger avant d'essayer de s'y mettre.

On utilise un mutex pour protéger le tableau et éviter les compétitions. On regarde si on peut prendre les deux : si oui, on le dit dans le tableau (pour que les autres n'essaient pas) ; si non, on s'endort sur une sémaphore privée pour que nos voisins nous réveillent quand ils ont fini.

Voilà pour le principe. Pour l'écrire, on fait une fonction `test` qui réveille un philosophe qui a faim (entièrement protégé par mutex) :

```
 test(k):
 si c[k] = 1 et c[k+1] != 2 et c[k-1]!=2
 alors
   c[k]:=2
   V(sempriv[k]) /* on libere le philosophe de son attente */
```

Du coup la procédure générale pour le philosophe i est la suivante :

```
 penser()
P(mutex); /* pour proteger le tableau c */
c[i]:= 1; /* i a faim puisqu'il a fini de penser */
test(i); /* il regarde si il peut manger; si oui, V(sempriv[i]) est fait */
V(mutex); /* pour que les voisins puissent rendre les fourchettes s'ils les avaient */
P(sempriv[i]); /* Si le V est fait, ça continue et il mange.
               Sinon il attend qu'un voisin le reveille en libérant les fourchettes */
manger() /* miam, miam */
P(mutex); /* on va dire a tout le monde que l'on a fini de manger */
c[i]:= 0; /* on pose les couverts */
test(i-1); /* on regarde si les voisins peuvent manger maintenant que l'on */
test(i+1); /* a posé les couverts et on fait leur V(sempriv[i]) si oui */
V(mutex);
```

Et on boucle.

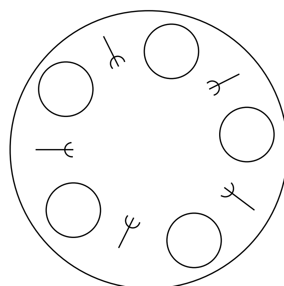
Certains disent qu'il peut y avoir une famine en cas de coalition car à 5 philosophes, les philosophes peuvent s'arranger pour qu'un des philosophes soit toujours bloqué par un de ces voisins (2 est bloqué si 3 et 1 mangent alternativement avec une phase commune à chaque fois). Mais je trouve ça bête vu que l'algorithme est basé sur le bon vouloir des philosophes à suivre l'algo donné. Autrement dit, on utilise des sémaphores et pas des moniteurs alors les philosophes mal lunés peuvent faire vraiment n'imp.

En revanche, on peut présenter une 4ieme possibilité, dite du «maître d'hotel» : je ne prend pas la fourchette si c'est la dernière sur la table. C'est beau, ça fait le parallèle avec la dernière possibilité donnée dans le cours qui consiste à modifier l'algo. Parait que «L'algorithme général est connu sous le nom de "l'algorithme du banquier".», dixit Riveill. Pour la rédaction propre de la chose, on verra plus tard. Il faut une semaphore initialisée à 4 (alors qu'il y a 5 fourchettes, on est d'accord) prise avant de meme commencer à chercher à manger, et relachée après le repas.

Ce qui nous donne en résumé :

- Première solution : truc bête avec interblocage
- Deuxième solution : requêtes ordonnées
- Troisième solution : requêtes globales
- Quatrième solution : Modification de l'algorithme (connaissance globale).

Fin réponse



CORRECTION CORRECTION