Système T.D

Exercice 1 (Exclusion mutuelle par variables partagées)

Si vous souhaitez dessiner vos graphes sur PC utilisez Graphviz ou cet editeur online https ://edotor.net/. Exemple de graphe :

```
graph {
tor nodes, place the cursor left to a node name
{
node[label="pid1, t=1"] a
}
a -- b
a -- c;
a -- d;
}
```

On dispose de deux processus (P_0 et P_1). L'objectif de l'exercice est d'établir un algorithme utilisant des variables partagées par les processus pour réaliser l'exclusion mutuelle des sections critiques en respectant les spécifications données en cours. On ne fait aucune hypothèse sur l'atomicité des manipulations des variables.

Question 0.1 Rappeler pourquoi une simple attente active sur une variable `tour` qui vaudrait vrai ou faux selon qu'un processus est ou non en section critique, immédiatement suivie d'une mise à jour de `tour` ne convient pas.

Question 0.2 On propose dans un premier algorithme, la solution suivante. La variable `tour` prend les valeurs 0 ou 1. L'algorithme (1) de P_i est le suivant :

```
 tantque
vrai faire
  actions avant sc
  tantque tour <> i faire rien
  section critique i
  tour <- 1-i
  actions apres sc
fintantque
```

(noter que `1-i` $= 0$ si `i` $= 1$ et 1 si `i` $= 0$). La phase d'initialisation du système est réduite à : `tour <- 0`. Construire le système de transitions étiquetées de cette solution. Montrer que l'exclusion mutuelle est bien réalisée mais que la condition de progression n'est pas satisfaite.

Question 0.3 Pour remédier à cet inconvénient, on emploie au lieu de `tour`, deux variables booléennes `D0` et `D1` : `Di` est vraie ssi P_i demande à passer en section critique. L'algorithme (2) de P_i est alors :

```
tantque vrai faire
  actions avant sc
  Di <- vrai
  tantque D(1-i) faire rien
  section critique i
  Di <- faux
  actions après sc
fintantque
```

(`D(1-i)` désigne `D1` si `i` = 0 et `D0` si `i` = 1). Les variables `Di` sont initialisées à faux.

Construire le système de transitions étiquetées de cette solution. Montrer que la progression est maintenant assurée. Montrer que le système possède un état d'interblocage.

Question 0.4 La dernière solution est due à G.I. Peterson (1981). Elle vérifie les propriétés attendues d'une solution d'exclusion mutuelle. On utilise une variable `tour` *et* les deux variables `Di`.

```
 tantque vrai faire ---
Algorithme de Peterson ---
  actions avant sc
  Di <- vrai
  tour <- 1-i
  tantque ( D(1-i) et (tour = 1-i) ) faire rien
  section critique i
  Di <- faux
  actions après sc
fintantque
```

les `Di` à faux. Construire encore le système de transitions étiquetées du système. Montrer que :

1. l'exclusion mutuelle est garantie ;
2. il n'y a pas d'interblocage ;
3. l'attente bornée est vérifiée : combien de passages en section critique de $P_{(1-i)}$, P_i doit-il attendre au plus lorsqu'il a demandé à passer en section critique ?

Exercice 2 (Fork-1)
Écrire un programme C (pour système UNIX) qui crée 4 fils $F1, F2, F3$ et $F4$. Chaque fils aura le même code (fonction `void fils(...)`) et affichera son numéro de fils ainsi que son `pid`. Le père affichera un simple message à chaque création de fils puis un dernier message avant de se terminer.

Exercice 3 (Fork : once upon a time...)
Il était une fois un programmeur débutant en programmation système sous UNIX. Ayant découvert la possibilité de créer plusieurs processus il voulût s'en servir pour réaliser le travail suivant qu'il avait jusqu'à présent programmé de manière séquentielle : stocker dans une variable entière `v`, la valeur $f(x) + g(x)$ pour un x choisi, où f et g sont deux fonctions à argument entier renvoyant chacune 0 ou 1 mais dont les calculs, indépendants l'un de l'autre, sont très longs. Il écrivit donc le programme P_1 suivant en C :

```
1 /* inclusions nécessaires ...  */
2 int f(int x) { /* code de f ... */ }
3 int g(int x) { /* code de g ... */ }
4
```

```
 5 main()
 6       {
 7       int n,v,x;
 8
 9       v = 0;
10       x = 3; /* essai de calcul pour x=3 */
11       if ((n=fork()) != 0) v = v + f(x);
12       else v = v + g(x);
13       printf("valeur de v: %d\n",v);
14       } /* main */
```

P_1 est compilé sans erreur mais son exécution surprend le programmeur (on supposera dans la suite que $f(3) = 1$ et $g(3) = 0$).

Question 0.5 Expliquez pourquoi en donnant d'abord deux exemples d'exécutions possibles *distinctes* puis en indiquant les différentes erreurs commises.

Question 0.6 Notre programmeur rencontre alors un collègue chevronné qui écrit une version correcte du programme. Donnez l'algorithme d'une solution au problème en détaillant les actions du père et du fils et en résolvant le problème du retour de résultat du fils. Écrivez ensuite le programme C correspondant.

Question 0.7 Quelque temps plus tard, notre programmeur transmet son programme à un ami travaillant sur un autre site, vantant l'accroissement de performances de cette nouvelle version. Hélas, il est très déçu lorsque son ami lui indique que la nouvelle version est *plus lente* que la version séquentielle ! Dépité, notre héros retourne consulter son collègue expérimenté qui lui explique la raison de tout cela. Saurez-vous donner cette explication ?

Exercice 4
Qu'affiche le programme suivant :

```
 main() { int i;
 for (i=1;i<8;i++)
    {
    printf("\% d",i);
    switch (fork())
        {
        case 0 : printf("! \n");exit(1);
        default : sleep(1);
        }
    }
printf("fin\n"); }
```

Exercice 5 (Tube simple)
Une application est composée d'un processus père et d'un fils. Le fils réalise le travail suivant :

```
lire un réel x au clavier tantque x <> 0 faire
  si x < 0 alors message d'avertissement
  sinon transmettre f(x) au père
```

```
  demander un réel x au clavier
fintantque
```

La fonction f est définie dans le code du fils. Le père reçoit les valeurs y ($= f(x)$). Lorsque le dernier y est reçu, il calcule la moyenne des y et l'affiche à l'écran. Les valeurs sont transmises par un tube : préciser comment.

Question 0.8 On suppose que $\forall x \in \mathbb{R},\ f(x) > 0$. Comment le père peut-il détecter la fin de la suite des valeurs reçues ? Écrire les algorithmes et les programmes C du père et du fils dans cette hypothèse.

Exercice 6 (Sémaphores)
Rappeler la signification des opérations élémentaires P et V sur les sémaphores. On considère un système sur lequel s'exécutent trois processus $P1, P2$ et $P3$. $P1$ exécute successivement les parties de code A et B, et $P2$ les parties C et D. $P3$ ne doit commencer à réellement exécuter son code que lorsque $P1$ a terminé la partie A de son code. B et C sont en exclusion mutuelle.

Question 0.9 Construire le graphe de précédence de ce système de processus.

Question 0.10 Résoudre le problème entre $P3$ et $P1$ avec un sémaphore.

Question 0.11 Même chose pour l'exclusion mutuelle entre $P1$ et $P2$. Peut-on utiliser le même sémaphore ?

Question 0.12 Implanter ce système en C ou Java.

Exercice 7 (Gestion Processus)
On souhaite programmer un système dans lequel le processus Père lance 3 fils. Chaque fils réalise un travail. Le processus Père attend en principe la fin des travaux des fils et reçoit un compte rendu de ceux-ci du type réussi ou échoué. En cas d'échec, les fils renvoient une valeur (1–10) qui doit être affichée par le Père. Si Fils2 retourne une erreur, le Père arrête immediatement tous les fils non terminés, affiche un message et termine son exécution.

Question 0.13 Expliquez comment gérer le retour d'information au processus Père ;

Question 0.14 Donnez le code C correspondant.