Le corrigé

Question 1 (2 pts) : Généralités

Répondez aux questions suivantes (les réponses doivent être justifiées, concises et claires) :

1) **[1 pt]** Donnez le nombre de processus créés par le bout de code suivant (supposez que l'appel à « *fork* » ne retourne pas d'erreur) :

```
int nb=0;        while(fork() && nb <2) nb=nb+1;
```

Le nombre de processus créés est 3.
Le processus principal PP crée un fils F1 qui va quitter la boucle.
Comme nb=0 <2, PP incrémente nb (nb=1) et passe à l'itération suivante.
PP crée un fils F2 qui va quitter la boucle.
Comme nb=1 <2, PP incrémente nb (nb=2) et passe à l'itération suivante.
PP crée un fils F3 qui va quitter la boucle.
Comme nb=2 <2, PP va quitter la boucle.

2) **[1 pt]** Pour obtenir un verrou, votre coéquipier décide d'utiliser l'instruction « *TSL* (Test and Set Lock) » et vous propose de remplacer « *while(TSL(verrou) !=0);* » par « *while(TSL(verrou) !=0) { while(verrou); }* ». Que pensez-vous de sa proposition ?

La proposition de votre coéquipier permet de réduire le nombre d'appels à TSL. Après le 1ère appel, TSL sera appelée uniquement si le verrou est à 0. L'intérêt de sa proposition est de remplacer durant l'attente active des exécutions à répétition d'une instruction atomique, complexe et coûteuse (qui nécessite un verrouillage du bus mémoire pour une lecture suivie d'une écriture de la mémoire) par une simple lecture d'une valeur de la mémoire (ou d'un cache).

Question 2 (3 pts) : Interblocage

Un système comporte *8* ressources d'un même type *R* partagées par *4* tâches *T1, T2, T3* et *T4*. Chaque tâche peut demander, une à une, au maximum *3* ressources qu'elles libèrent, au bout d'un temps fini, à la fin de son exécution.

1) **[1 pt]** Montrez qu'il peut y avoir interblocage dans un tel système.

2) **[1 pt]** Montrez qu'il ne peut y avoir d'interblocage dans un tel système si on ajoute une ressource de type R.

3) **[1 pt]** Supposez que *n* et *m* sont respectivement les nombres de tâches et de ressources de type *R* et que chaque tâche *Ti* a besoin au total de *ri* ressources. Donnez en fonction de ces paramètres une condition suffisante qui garantit l'absence d'interblocage.

1) Oui, car les 4 conditions nécessaires et suffisantes sont satisfaites dans le cas où chacune des tâches détient 2 ressources et demande une autre (2 x4 =8). Toutes les tâches sont bloquées en attente d'une ressource détenue par une autre.

2) Si le nombre de ressources disponibles est 0 alors il y a au moins une tâche qui détient toutes les ressources dont elle a besoin (2*3+3=9). Cette tâche peut s'exécuter jusqu'à la fin et libérer les ressources qu'elle détient et permettre ainsi aux autres de poursuivre leurs exécutions (3*3=9).

3) Il suffit que (r1+r2+….+rn – n) < m car si chaque tâche Ti détient (ri-1) ressources, (r1+r2+…+rn –n) ressources sont utilisées. Il faudrait 1 de plus pour permettre à une tâche de s'exécuter jusqu'à la fin. Les ressources récupérées à la fin d'exécution de cette tâche vont permettre à une autre de s'exécuter jusqu'à la fin, et ainsi de suite.

Question 3 (2 pts) : Windows

Expliquez comment implémenter sous Windows, en utilisant l'API win32, le traitement réalisé par la commande Linux « p1.exe | p2.exe où p1.exe et p2.exe sont deux exécutables. Ne donnez pas de code. Indiquez, par contre, toutes les étapes à suivre sous forme de commentaires clairs et précis.

Il suffit de : (0.25 pt par étape)

1. **créer un pipe anonyme.**
2. **rendre héritable le handle d'écriture dans le pipe.**
3. **créer un processus fils en indiquant que a) les objets héritables vont être dupliqués au fils, b) la sortie standard (Startupinfo) est le handle d'écriture du pipe et c) son exécutable est p1.exe.**
4. **fermer le handle d'écriture.**
5. **Rendre héritable le handle de lecture dans le pipe.**
6. **créer un processus fils en indiquant que a) les objets héritables vont être dupliqués au fils, b) l'entrée standard (Startupinfo) est le handle de lecture du pipe et c) son exécutable est p2.exe.**
7. **fermer le handle de lecture du pipe.**
8. **Attendre la fin des deux fils.**

Question 4 (5 pts) : Synchronisation

1) **[2.5 pts]** Considérez le problème des lecteurs / rédacteurs et la première solution utilisant les sémaphores (vue en classe).

```
Semaphore mutex=1, redact =1;
int Nbl =0 ;
```

```
Redacteur () {
  while(1) {  P(redact);
              Ecriture();
              V(redact);
          }
}
```

```
Lecteur () {
   while(1) {    P(mutex) ;
                 Nbl++ ;
                 if (Nbl==1) P(redact);
                 V(mutex);
                 Lecture();
                 P(mutex);
                 Nbl--;
                 if(Nbl==0) V(redact);
                 V(mutex);
          }
}
```

On veut limiter à *Lmax* le nombre maximal de lecteurs qui accèdent simultanément à la base de données. Lorsque ce nombre est atteint, tout lecteur qui veut accéder à la base de données est mis en attente jusqu'à ce qu'une « entrée se libère ».

a) **[1 pt]** Complétez/modifiez le pseudo-code précédent pour prendre en compte cette nouvelle contrainte.

b) **[1.5 pt]** Cette nouvelle contrainte permet-elle d'éliminer le problème de famine des rédacteurs ? Justifiez votre réponse. Si vous répondez non, indiquez comment modifier/compléter le code donné en a) pour éliminer le problème de famine des rédacteurs. Les temps de lecture et écriture sont supposés finis.

a) **Semaphores SL = Lmax ; // pour limiter à Lmax le nombre de lecteurs**

```
Lecteur () {
   while(1) {   P(SL) ;  P(mutex) ;
                 Nbl++ ;
                 if (Nbl==1) P(redact);
                V(mutex);
                Lecture();
               P(mutex);
               Nbl--;
              if(Nbl==0) V(redact);
              V(mutex); V(SL) ;
          }
}
```

b)

```
Semaphore mutex=1, redact =1;
Semaphore tour=1,SL=Lmax ;
int Nbl =0 ;
```

```
Redacteur () {
  while(1) {  P(tour) ; P(redact);
               Ecriture(); V(tour) ;
              V(redact);
          }
}
```

```
Lecteur () {
   while(1) {    P(tour) ;P(SL) ; P(mutex) ;
                  Nbl++ ;
                  if (Nbl==1) P(redact);
                 V(mutex); V(tour) ;
                Lecture();
               P(mutex);
               Nbl--;
              if(Nbl==0) V(redact);
              V(mutex); V(SL) ;
          }
}
```

2) **[2.5 pts]** Considérez le moniteur ProducteurConsommateur suivant (cas d'un producteur et d'un consommateur) :

```
Moniteur ProducteurConsommateur
{     const N=100 ;
      int tampon[N], compteur =0, ic=0, ip=0 ;
      boolc nplein, nvide ; // variables de condition
```

```
    void produire (int objet)
     {      if (compteur==N) wait(nplein) ;
            tampon[ip] = objet ;
            ip = (ip+1)%N ; compteur++ ;
            if (compteur==1) signal(nvide) ;
     }
     int consommer ( )
     {      int objet ;
            if (compteur ==0) wait(nvide) ;
            objet = tampon[ic] ;
            ic = (ic+1)%N ; compteur -- ;
            if(compteur==N-1) signal(nplein) ;
            return objet ;
     }
}
```

a) **[1 pt]** Est-il possible de réduire à *1* le nombre de variables de condition dans le cas d'un tampon de longueur *1 (N=1)* ? Justifiez votre réponse. Si oui, donnez le code modifié.

b) **[1.5 pt]** Expliquez comment adapter cette solution aux cas de plusieurs producteurs et plusieurs consommateurs. Est-il possible, dans ce cas, de réduire à *1* le nombre de variables de condition pour *N=1* ? Justifiez votre réponse. Si oui, donnez le code modifié.

a) Oui car, dans ce cas, on a une alternance entre une production et une consommation : 1 production ; 1 consommation ; 1 production ….. Une seule variable de condition suffit pour assurer cette alternance.

```
Moniteur ProducteurConsommateur
{   const N=1 ;
    int tampon[N], compteur =0, ic=0, ip=0 ;
    boolc tour ; // variables de condition
    void produire (int objet)
    {       if (compteur==N) wait(tour) ; // attendre la fin d'une consommation
            tampon[ip] = objet ;
            ip = (ip+1)%N ; compteur++ ;
            if (compteur==1) signal(tour) ; // signaler la fin d'une production
    }
    int consommer ( )
    {       int objet ;
            if (compteur ==0) wait(tour) ; // attendre la fin d'une production
            objet = tampon[ic] ;
            ic = (ic+1)%N ; compteur -- ;
            if(compteur==N-1) signal(tour) ; //signaler la fin d'une consommation
            return objet ;
    }
}
```

b) Oui car, dans ce cas aussi, on a une alternance entre une production et une consommation : 1 production ; 1 consommation ; 1 production ….. Une seule variable de condition suffit pour

assurer cette alternance. Par contre, comme il y a plusieurs producteurs et plusieurs consommateurs, il faut vérifier à la sortie de l'attente « wait(tour) » si la condition de sortie de cette attente est toujours valide.

```
Moniteur ProducteurConsommateur
  {   const N=1 ;
      int tampon[N], compteur =0, ic=0, ip=0 ;
      boolc tour ; // variables de condition
     void produire (int objet)
      {       while (compteur==N) wait(tour) ;
              tampon[ip] = objet ;
              ip = (ip+1)%N ; compteur++ ;
              if (compteur==1) signal(tour) ;
      }
      int consommer ( )
      {       int objet ;
              while (compteur ==0) wait(tour) ;
              objet = tampon[ic] ;
              ic = (ic+1)%N ; compteur -- ;
              if(compteur==N-1) signal(tour) ;
              return objet ;
      }
  }
```

Question 5 (3 pts) : Gestion de la mémoire

On considère un système de pagination à *3* niveaux dans lequel les adresses (virtuelles et physiques) sont codées sur *48* bits. La taille d'une page est de *4 KiO*. La taille de chaque table de pages, peu importe son niveau, est égale à *32 KiO*. Chaque entrée d'une table de pages est composée de *8* octets.

1) **[1 pt]** Quelle est la taille maximale, en nombre de pages, de l'espace virtuel d'un processus, supporté par un tel système ?

2) **[2 pts]** Donnez le format d'une adresse virtuelle. Expliquez au moyen d'un schéma simple comment convertir une adresse virtuelle en une adresse physique.

1) $2^{48}/ 2^{12} = 2^{36}$ = 64 Gi pages (1 pt)

2) (1 pt) L'adresse virtuelle est composée de 4 champs (3 niveaux de tables de pages + déplacement dans la page): 12 bits + 12 bits + 12 bits + 12 bits. Le nombre d'entrées dans chaque table de pages est : $32 \times 2^{10} / 8 = 2^{12}$. Le nombre de bits pour le déplacement dans la page est donné par la taille d'une page ($4Ki = 2^{12}$).

(1 pt) Ex : 0x BA9 876 543 210

Les 12 bits de poids fort (BA9) vont permettre de sélectionner une entrée dans la table du 1^{er} niveau. Cette entrée pointe vers le début d'une table du $2^{ième}$ niveau. Les 12 bits suivants de l'adresse (876) vont permettre de sélectionner une entrée dans cette table du $2^{ième}$ niveau. Cette

entrée pointe, à son tour, vers le début d'une table du 3ième niveau. Les 12 bits suivants de l'adresse (543) vont permettre de sélectionner une entrée dans cette table du 3ième niveau. Cette entrée contient toutes les informations concernant la page référencée. Si le bit de présence est en mémoire, l'adresse virtuelle est obtenue en remplaçant les 36 bits de poids fort (numéro de page BAE876543) par le numéro de cadre. Sinon, un défaut de page est généré pour charger la page en mémoire et mettre à jour la table de pages.

Question 6 (5 pts) : Ordonnancement

1) **[2 pts]** Un système monoprocesseur gère l'exécution de *4* processus *P1, P2, P3* et *P4* décrits dans le Tableau *1* :

Processus	*Priorité*	*Temps d'exécution*
P1	*20*	*3 unités de CPU, 3 unités d'E/S, 2 unités de CPU,*
P2	*14*	*2 unités de CPU, 1 unité d'E/S, 1 unité de CPU,*
P3	*15*	*4 unités de CPU*
P4	*10*	*2 unités de CPU*

Tableau *1*

Supposez que :

- le système est doté d'un seul périphérique d'E/S et les requêtes d'E/S sont traitées selon la politique *PAPS* (Premier Arrivé, Premier Servi).
- l'ordonnancement est préemptif à priorités statiques,
- le temps de commutation est égal à *0*,
- tous les processus arrivent dans le système à l'instant *0*,
- La priorité *0* est la plus basse.

Donnez le diagramme de Gantt montrant l'ordonnancement des processus.

P1P1P1P3P3P3P1P1P3P2P2P4P2P4.

2) **[3 pts]** Considérez un système composé de *n* tâches périodiques de type *A* et *m* tâches périodiques de type *B*. Les tâches de type *A* ont une période de *10* et une durée d'exécution de *1*. Les tâches de *B* ont une période de *8* et une durée d'exécution de *2*. Ces tâches sont indépendantes et leurs échéances sont leurs périodes.

   a) **[1 pt]** Supposez que pour ordonnancer ces tâches, le système utilise l'algorithme *EDF*. Donnez le nombre maximal de tâches à admettre dans le système sans compromettre leur ordonnançabilité.

   b) **[1 pt]** Supposez que la politique d'ordonnancement utilisée est *RMA*. Donnez en fonction de *m* et *n* une condition suffisante d'ordonnançabilité de toutes les tâches.

   c) **[1 pt]** Supposez que *n=4, m=3* et que les tâches sont ordonnancées *EDF*. Donnez la plus petite période à affecter aux tâches de type *A* sans compromettre l'ordonnançabilité de toutes les tâches.

a) n (1/10) + m(2/8) <= 1 => 2n+5m = 20 au maximum 8 tâches pour n=7 et m=1, si m>0.
Sinon, on pourrait atteindre 10 tâches de type A avec 0 tâches de type B.
b) n (1/10) + m(2/8) <= (n+m) ($2^{1/(n+m)}$ – 1)

c) 4 (1/p) + 3(2/8) <= 1 => 4 (1/p) <= 1/4 => p=16