CONSERVATOIRE NATIONAL DES ARTS ET METIERS

Durée : 2 heures

METHODES DE PROGRAMMATION SYSTEMES

UE NSY103 – NANCY/METZ

CORRECTION DE L'EXAMEN DE RATTRAPAGE de SEPTEMBRE 2012

Année 2011 – 2012, deuxième semestre
Coefficient : 1 (100 % de la note)

Correcteur : Emmanuel DESVIGNE

Documents autorisés : NON

Exercice n°1 : synchronisation *(3 questions pour un total de 7 points)*

Question 1.a) [*1 point*] : Un sémaphore est un objet (ou type abstrait de donnée) contenant une variable entière positive et des mécanismes qui permettent de gérer de façon esthétique les accès concurrent à des ressources par plusieurs processus, en bloquant les processus (sans consommation de CPU) lorsque la ressource n'est pas accessible.

La fonction `Init(S, val)` permet d'initialiser le sémaphore `S` en lui attribuant la valeur `val`.

La fonction `P(S)` : si `S` est strictement supérieur à zéro, `S` est décrémenté. Sinon, le processus appelant est bloqué (pas de consommation de CPU), en attendant qu'un autre processus fasse une opération `V(S)`.

La fonction `V(S)` débloque un des éventuels processus bloqué par un `P(S)` avec `S` nul, et incrémente `S` dans le cas contraire.

À noter que les fonctions `P()` et `V()` sont des fonctions unitaires (il ne peut y avoir préemption durant leur exécution).

Question 1.b) [*2,5 points*] : Nous utiliserons deux sémaphores :

- `s1` indique que des données sont lues et prêtes à être traitées,
- et `s2` indique que des données ont été traitées et un résultat est prêt à être sauvegardé.

Le code de p1, p2, p3 devient :

```
// processus p1
Init(s1, 0)
tant_que(vrai) faire
      lecture_donnees()
      V(s1)
fin_tant_que
```

```
// processus p2
Init(s2, 0)
tant_que(vrai) faire
      P(s1)
      simulation()
      V(s2)
fin_tant_que
```

```
// processus p3

tant_que(vrai) faire
      P(s2)
      sauve_resultats()
fin_tant_que
```

Question 1.c) [*3,5 points*] : Nous utiliserons la même trame de programme que précédemment (avec les sémaphores `s1` et `s2`, qui ont la même utilité), ainsi que deux autres sémaphores :

- `nb_libre_p1_p2` qui indique le nombre d'emplacements libres dans la file d'attente permettant à p1 d'envoyer ses données à p2,
- `et nb_libre_p2_p3` qui indique le nombre d'emplacements libres dans la file d'attente permettant à p2 d'envoyer ses données à p3.

Le code de p1, p2, p3 devient :

```
// processus p1
Init(s1, 0)
Init(nb_libre_p1_p2, 2)
tant_que(vrai) faire
      P(nb_libre_p1_p2)
      lecture_donnees()
      V(s1)
fin_tant_que
```

```
// processus p2
Init(s2, 0)
Init(nb_libre_p2_p3, 2)
tant_que(vrai) faire
      P(nb_libre_p2_p3)
      P(s1)
      simulation()
      V(s2)
      V(nb_libre_p1_p2)
fin_tant_que
```

```
// processus p3


tant_que(vrai) faire
      P(s2)
      sauve_resultats()
      V(nb_libre_p2_p3)
fin_tant_que
```

* * * * * *

Exercice n°2 : les signaux *(4 questions pour un total de 7 points)*

Question 2.a) [*1 point*] : Un signal est une forme (limitée) de communication entre deux processus. C'est un outil permettant à un processus d'envoyer une notification asynchrone à un autre processus, pour lui signaler l'apparition d'un événement. Quand un signal est envoyé à un processus, le système d'exploitation interrompt l'exécution normale de celui-ci. Si ce processus récepteur possède une routine de traitement pour le signal reçu (appelée vecteur de signal ou *signal handler*), il lance son exécution. Dans le cas contraire, il exécute la routine de traitement par défaut de ce signal.

Question 2.b) [*1 point*] : Le programme arme un « vecteur de signal » pour que le système d'exploitation exécute la fonction `sg()` à la réception d'un signal utilisateur SIGUSR1. Puis, il entre dans une boucle infinie (note du correcteur : ce qui n'est évidemment pas acceptable en terme de consommation de CPU).

À la réception du signal SIGUSR1, la boucle infinie est interrompue pour exécuter la fonction `sg()`, qui affiche le texte « *Signal SIGUSR1 reçu* ». Le vecteur de signal `sg()` est de nouveau armé pour le signal SIGUSR1 (ainsi, le message « *Signal SIGUSR1 reçu* » sera toujours affiché à chaque réception du signal SIGUSR1, quel que soit le nombre de fois où ce dernier est reçu). Puis le programme reprendra sa boucle infinie.

Question 2.c) [*2,5 points*] : Le programme que nous aurions à écrire aurait à faire appel à la primitive : `kill(p, SIGUSR1)` avec `p` le numéro de PID du programme de l'énoncée. Or, comment connaître ce numéro de PID ?

La solution triviale consiste à mettre la ligne suivante au début de la fonction `main()` du programme de l'énoncé :

```
printf("PID du processus :%ld\n", getpid());
```

Ce PID serait ainsi passé manuellement en paramètre au programme que nous aurions à écrie.

Une autre solution (utilisée classiquement par les *daemons* UNIX) consiste à ce que le programme de l'énoncé écrive son PID dans un fichier (classiquement, beaucoup de *daemons* `xxx` qui tournent en tâche de fond sous UNIX écrivent leur numéro de PID dans un fichier « `/var/run/xxx.pid` »).

Enfin, toutes les solutions (probablement plus complexes) de communications inter-processus vues en NSY 103 conviennent, et seront acceptées :

- utilisation d'un pipe nommé pour transmettre le numéro de PID (pipe nommé uniquement, l'utilisation de pipes anonymes imposant un lien de parenté entre les deux processus),
- utilisation d'un serveur réseau (socket TCP ou UDP),
- utilisation de MSQ,
- écriture du numéro de PID dans une zone de mémoire partagée,
- etc.

Question 2.d) [*2,5 points*] :

```
#include <stdio.h.h>
#include <signal.h>


/* Variable globale accessible depuis toute fonction */
int     cptr_usr1=0;


void   sg(int sig)
{
    cptr_usr1++;
    signal(sig, sg);
}
```

```
void   sg2(int sig)
{
    printf("Nombre de signaux SIGUSR1 reçus :%d\n", cptr_usr1);
    signal(sig, sg2);
}

void   main()
{
    signal(SIGUSR1, sg);
    signal(SIGUSR2, sg2);
    while(1);
}
```

Dans les cas extrêmes, deux processus pourraient envoyer, dans un même cycle de l'ordonnanceur du système d'exploitation, un même signal SIGUSR1 à notre programme. Le compteur « `cptr_usr1` » ne serait incrémenté qu'une seule fois, bien que le signal ait été envoyé deux fois par deux processus différents.

La solution consisterait à ne pas utiliser les signaux utilisateurs classiques SIGUSR1 et SIGUSR2 (qui ne s'empilent pas), mais les signaux temps réels, qui ont la propriété de s'empiler (le vecteur associé à un signal temps réel est exécuté autant de fois que le signal a été reçu, même si ces envois ont eu lieu dans le même tour de l'ordonnanceur).

* * * * * *

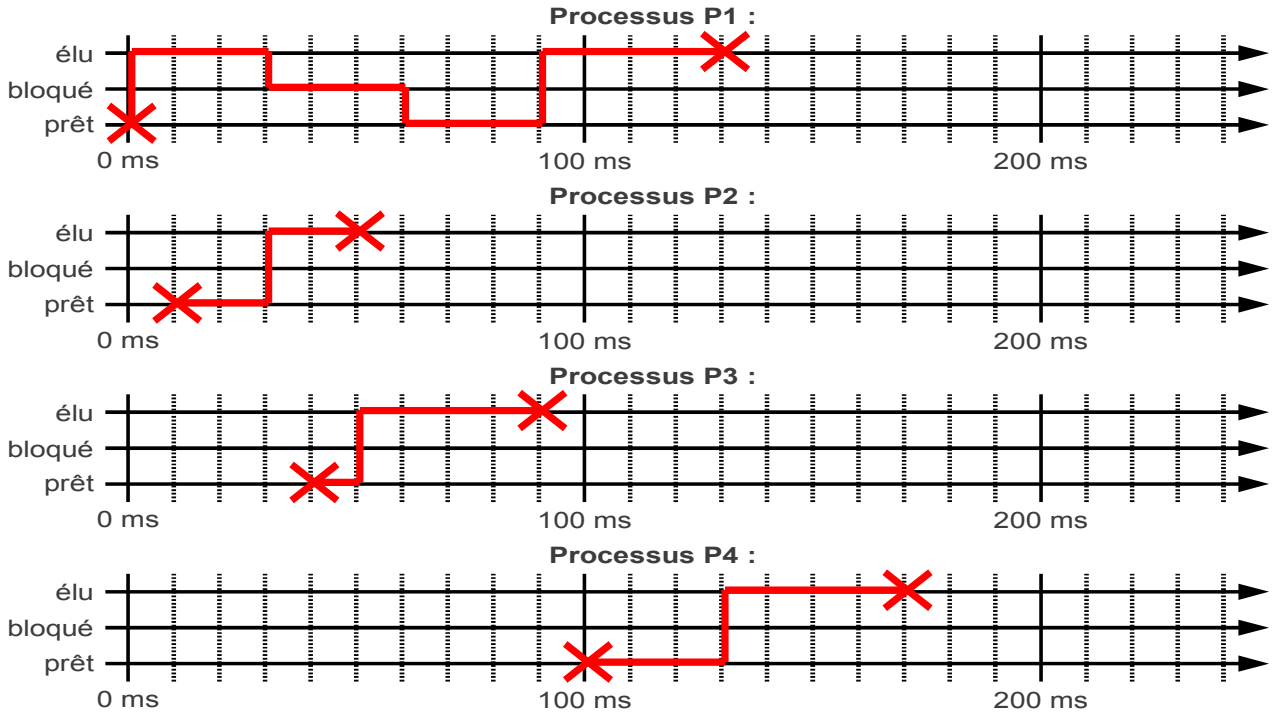
Exercice n°3 : ordonnancement *(4 questions pour un total de 6 points)*

Question 3.a) [*1 point*] : L'ordonnanceur (ou *scheduler* en anglais) est le code du système d'exploitation qui réalise le travail d'élection (c'est à dire qui choisit le processus – ou le thread si cette notion est connue du système d'exploitation – qui sera exécuté, lorsque plusieurs processus – ou thread – sont en compétition pour obtenir le CPU – état prêt –). Dans un système d'exploitation préemptif, il arme une horloge pour partager équitablement le CPU (afin d'éviter les famines). Enfin, l'ordonnanceur effectue le travail de « commutation de contexte ».

Note du correcteur : ½ point si le candidat explique le travail d'élection, ½ point supplémentaire s'il aborde la notion de thread, et 1 point pour le travail de commutation de contexte.

Question 3.b) [*1 point*] : Un système d'exploitation temps réel utilise un ordonnanceur et fournit des primitives qui, si elles sont bien utilisées, permettent d'assurer mathématiquement qu'un processus possède le CPU durant au minimum un temps donné, et ne puisse rester plus longtemps qu'un temps donné sans avoir de CPU. Ainsi, les systèmes d'exploitation temps réels n'ont pas pour but initial d'être performants et rapides, mais ils assurent aux processus d'avoir les ressources minimum nécessaires (en terme de CPU) pour réaliser leur tâche.

Question 3.c) [*2 points*] :



Question 3.d) [2 points] : (note au correcteur : 2 solutions possibles à la fin, P1/P4... ou P4/P1...)