Examen d'Architecture et Système

15 janvier 2016

Durée : 2 heures.

Remarques : Il y avait 23 points en total, sans compter la question 2c qui était un bonus. Le minimum requis pour avoir une note de 10 était donc de 11,5 points.

1 Sémaphores

On considère le programme suivant qui consiste d'un thread principal et trois threads A, B, C. Le thread principal (dans `main`) lance les trois threads et attend leur fin. Certains endroits dans le programme sont indiqués par des commentaires (voir ci-dessous).

```
void* A(void* data) {
    // A1
    printf("u\n");
    // A2
    printf("v\n");
    // A3
}

void* B(void* data) {
    // B1
    printf("w\n");
    // B2
    printf("x\n");
    // B3
}
```

```
void* C(void* data) {
    // C1
    printf("y\n");
    // C2
    printf("z\n");
    // C3
}

int main() {
   pthread_t ta,tb,tc;
   // MAIN
   pthread_create(&ta,NULL,A,NULL);
   pthread_create(&tb,NULL,B,NULL);
   pthread_create(&tc,NULL,C,NULL);
   pthread_join(ta,NULL);
   pthread_join(tb,NULL);
   pthread_join(tc,NULL);
}
```

Figure 1: Programme avec threads A,B,C.

On considère les résultats possibles du programme, c'est à dire les différentes séquences des lettres `u,v,w,x,y,z`.

(a) Combien de séquences possibles y a-t-il dans le programme ci-dessus ? (inutile de les énumerer tous)

Solution: (2p) Deux façons d'arriver à la bonne réponse:

- Parmi les 6! permutations de `u,v,w,x,y,z`, il faut exclure la moitié où `v` intervient avant `u`, et encore la moitié où `x` intervient avant `w` etc, donc on arrive à $6!/2^3 = 90$.
- On utilise la formule pour "choisir k fois parmi n positions avec répétition": $\binom{n+k-1}{k}$. Commençons avec le mot `uv`; on y insère d'abord les lettres `w,x` sur trois positions (devant `u`, derrière `v` ou entre les deux) ce qui donne $\binom{3+2-1}{2} = 6$ choix. Puis on insère `y,z` dans le mot résultant (avec donc cinq positions possibles) ce qui donne $\binom{5+2+1}{2} = 15$ choix, et $6 \cdot 15 = 90$.

Dans le suivant, on souhaite limiter les séquences à un emsemble précis en introduisant des sémaphores. Vous pouvez utiliser autant de sémaphores que vous voulez, avec un syntaxe simplifié :

- `init(s,n)`: créer sémaphore `s` avec n créneaux ouverts initialement ;
- `wait(s);` attendre un créneau ouvert dans `s`;
- `post(s);` libérer un créneau dans `s`.

Spécifiez les opérations à rajouter pour limiter les résultats aux ensembles suivants, où le point (.) dénote la concaténation, $[uwx]$ l'ensemble des séquences des lettres u, w, x dans n'importe quel ordre, et $L_1 \# L_2$ n'importe quel entrelacement entre les mots dans L_1 et L_2.

(b) $\{uvwxyz\}$

(c) $[uwy].[vxz]$

(d) $(\{uv\}\#\{wx\}).\{yz\}$

Solution: Dans le suivant, `r,s,t` sont des sémaphores déclarées comme variables globales.

(b) (2p) Cette question correspondait exactement à un problème du TP numéro 9. Ayant terminé, `A` libère `B` et celui-ci libère `C`.

```
MAIN: init(s,0); init(t,0);
A3: post(s);
B1: wait(s);
B3: post(t);
C1: wait(t);
```

(c) (2p) Solution symétrique : On prend trois sémaphores, chacun associé avec un thread. Chaque thread, ayant fait sa première action, ouvre deux créneaux dans son sémaphore et attend les deux autres sémaphores.

```
MAIN: init(r,0); init(s,0); init(t,0);
A2: post(r); post(r); wait(s); wait(t);
B2: post(s); post(s); wait(r); wait(u);
C2: post(t); post(t); wait(r); wait(s);
```

Solution asymétrique : Un thread attend tous les autres, puis les libère:

```
MAIN: init(s,0); init(t,0);
A2: wait(s); wait(s); post(t); post(t);
B2: post(s); wait(t);
C2: post(s); wait(t);
```

(d) (2p) `C` doit attendre `A` et `B` qui eux peuvent s'entrelacer n'importe comment.

```
MAIN: init(s,0);
A3: post(s);
B3: post(s);
C1: wait(s); wait(s);
```

Dans les trois cas, donnez les opérations à rajouter dans les endroits `MAIN`, `A1`, etc.

2 Input/output

On considère le programme dans Figure 2 où un pipe est crée entre père et fils. Le père est censé envoyer un lettre `a` par seconde au fils qui les imprime. On rappelle que `p[0]` est en mode lecture, `p[1]` en mode écriture et que `1` représente la sortie standard.

```
1: int main () {
2:   int p[2];
3:   pipe(p);
4:   close(p[0]);

5:   if (fork()) {
6:     while (1) {
7:       sleep(1);
8:       write(p[1],"a",1);
9:     }
10:  } else {
11:    char c = 'a';
12:    while (1) {
13:      read(p[0],&c,1);
14:      write(1,&c,1);
15:    }
16:  }
17: }
```

Figure 2: Programme avec pipe.

Lorsqu'on lance le programme, on voit que l'écran se remplit toute de suite avec des `a` et que Ctrl+C ne suffit pas pour arrêter le programme. Expliquez ce comportement.

(a) Étant donné la description, le père est-il mort ou vivant ? Et le fils ?

Solution: (3p) Ce programme avait été demontré pendant le cours. Les observations nous permettent les conclusions suivantes :

- Seul le fils écrit sur la sortie standard (à priori, l'écran), du coup le fils doit être vivant.
- Ctrl+C fait que le shell envoie SIGINT au groupe des processus "du premier plan". Sauf utilisation de `setpgid`, un fils fait partie du même groupe que son père. Du coup, si Ctrl+C n'arrive pas à tuer le fils, c'est parce que le père n'est plus en premier plan car il est mort (voir 2c).

Trois points maximum: 0,5 points pour avoir correctement determiné le statut d'un processus, et 1 point supplémentaire pour avoir donné la bonne raison.

(b) Pourquoi l'écran se remplit-il à une telle vitesse ?

Solution: (1p) La vitesse d'affichage nous indique que l'appel de `read` en ligne 13 n'est pas bloquant. En général, ça peut arriver en raison de "fin de ficher" ou d'une erreur. Il n'y a pas "fin de fichier" car le fils lui-même a `p[1]` ouvert. La raison est donc forcement une erreur. Et ceci pour une raison bête : le descripteur `p[0]` a été fermé dans la ligne 4, donc avant le `fork`. On essaye donc à lire dans un fichier fermé.

(c) Question bonus : Si un processus est mort, expliquez pourquoi.

Solution: (2p) Le père est mort car il reçoit un signal (non rattrapé) SIGPIPE qui intervient lorsqu'un processus écrit dans un pipe dont tous les débouchés ont été fermé (ce qui est le cas en raison de la ligne 4). Deux réponses ont évoqué ce fait.

(d) Qu'est-ce qu'on peut changer pour que le programme se comporte comme prévu ?

Solution: (1p) Il suffit d'enlever la ligne 4 (ou l'échanger avec la ligne 5).

Tester la valeur renvoyée par `read` n'arrive pas à établir le comportement prévu, mais en général, c'est le bon réflexe, et j'ai donné un demi-point pour cela.

3 Signaux

On considère le programme dans la Figure 3 avec, encore une fois, un processus père et un processus fils. Père et fils sont censés jouer au "ping-pong", c'est à dire alterner en affichant `pere` et `fils`, et d'assurer l'ordre alternant par l'envoi des signaux. On rappele que l'appel système `pause` attend la livraison d'un signal.

```
1: void handler () { }
2: void parent (int cpid) {
3:   signal(SIGUSR1,handler);
4:   while (1) {
5:     pause();
6:     kill(cpid,SIGUSR1);
7:     printf("pere\n");
8:   }
9: }
```

```
10: void child () {
11:   int ppid = getppid();
12:   signal(SIGUSR1,handler);
13:   while(1) {
14:     kill(ppid,SIGUSR1);
15:     printf("fils\n");
16:     pause();
17:   }
18: }

19: int main () {
20:   int cpid = fork();
21:   if (cpid) parent(cpid);
22:   else child();
23: }
```

Figure 3: Père et fils communiquant par signaux.

(a) À quoi servent les appels `signal` dans les lignes 3 et 12 ? Pourquoi ne pas les enlever ?

Solution: Les appels de `signal` changent la disposition du signal SIGUSR1 qui, par défaut, est de terminer le processus. L'objectif n'est pas seulement d'éviter que le processus sera tué, mais aussi de permettre `pause` à être interrompu par la livraison d'un tel signal (il ne suffit donc pas d'ignorer le signal).

Par contre, il est faux de dire que l'envoi de SIGUSR1 résulterait dans une erreur en l'absence des lignes 3 et 12. Tout signal possède une disposition à tout moment, il n'est jamais "non-défini".

(b) Que sont les comportement possibles du programme (par rapport à la description donnée ci-dessus) ?

Solution: (3p) Autre que le comportement prévu, il y a principalement deux possibilités:

- Le fils peut tuer le père s'il démarre et envoi son premier signal avant que le père n'ait installé son gestionnaire de signal. (1p)
- Le programm peut s'arrêter parce que père et fils sont tous les deux dans un appel de `pause` sans qu'il y ait un signal à livrer. Ceci peut arriver au premier tour (le fils envoit son signal quand le père est entre les appels de `signal` et `pause`). Ou bien cela arrive si l'un des processus arrive à afficher son nom deux fois de suite. P.ex. le scénario suivant peut intervenir :

  (a) Le père est en `pause`, le fils lui envoie un signal, affiche `fils` puis fait `pause`.

(b) Le signal est livré au père qui se reveille et envoie un signal au fils avant d'être interrompu par l'ordonnanceur.

(c) Le signal est livré au fils qui se reveille, envoie un signal au père, affiche `fils` une deuxième fois et s'endort.

(d) Le père reprend la main, le signal est livré, il affiche `père` puis s'endort.

Deux points pour expliquer que les processus peuvent être bloqués, et un pour ne mentionner que le double affichage.

Il n'était pas nécéssaire d'analyser les situations inhabituels (`fork` échoue, envoi de signaux depuis l'extérieur), mais les réponses pour ces cas ont été valorisées. La réentrance, par contre, peut seulement être un problème à l'intérieur d'un seul processus (donc si on utilisait des threads).

(c) Quels méchanismes existent pour assurer le bon comportement du programme, et comment faut-il les appliquer (il suffit de décrire les principes, il n'est pas demandé de décrire les appels système en détail).

Solution: (2p) Le problème de "ping-pong" était le sujet d'un TP. Une solution discutée à l'époque était l'utilisation de `sigsuspend` qui agit comme pause mais sous un masque de blocage donné. On bloque la livraison de tous les signaux en dehors des appels `sigsuspend` et on permet la livraison de SIGUSR1 pendant ces appels. Voir la solution exemplaire sur la page web pour les détails.

D'autres solutions qui étaient acceptés sont la mémoire partagée ou les sémaphores. Pour utiliser ces derniers il faut soit utiliser des threads au lieu du `fork`, soit utiliser des sémaphores globaux (voir `sem_open`.

4 Système de fichiers

(a) Chaque inœud possède un champ qui compte les liens durs vers lui. À quoi sert ce compteur ?

Solution: (1p) Le compteur sert simplement à savoir si un inœud est encore requis. L'appel système `unlink` (utilisé par `rm`) décroit ce compteur et libère l'inœud si son compteur est zéro.

(b) Dans un dossier `toto` on possède un fichier `a.txt` et un sous-dossier `tata`. Combien de liens durs y a-t-il sur l'inœud de `toto` ?

Solution: (1p) Il y a trois liens dur : Un pour `toto` dans le dossier supérieur, un pour `.` dans `toto` et un pour `..` dans `tata`.

(c) Dans la ligne de commande, lorsqu'on lance une commande (tel que `cat`, `ls`, etc), le shell cherche un programme exécutable dans les dossiers indiquées par la variable `$PATH`. Par défaut cette variable ne contient pas le dossier actuel (`.`). Pourquoi ?

Solution: (1p) Principalement, c'est pour des raisons de sécurité. Cela empêche le shell d'accidentellement exêcuter un programme (potentiellement malveillant) dans le dossier actuel qui aurait le même nom qu'une commande qu'on utilise fréquemment (comme `ls`) ou un nom similaire.