Mercredi 4 janvier 2006
Durée : 3 heures
Documents autorisés

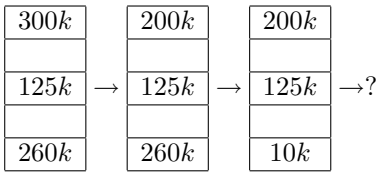
Examen : systèmes d'exploitation
L3, parcours informatique, 2005-2006

Examen

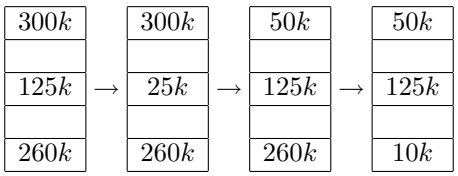
La mémoire, la concurrence

Exercice 1 : allocation contiguë. Étant données des partitions mémoire de 300k, 125k et 260k (dans cet ordre), décrivez, étape par étape, comment chacun des algorithmes First-fit et Best-fit placerait-il des processus de 100k, 250k et 250k (dans cet ordre).

Solution. On dessin une suite de partition mémoire :
Algorithme First-fit :



L'algorithme First-fit ne trouvera pas la place pour ranger le troisième processus de $250k$.
Algorithme Best-fit :



□

Exercice 2 : pagination. On considère un système de pagination avec une table de pages stockée en mémoire (sans niveaux d'indirection, c.à.d. il n'y a pas de répertoire de pages) et de registres associatifs pour bufferiser la correspondance pages/cadres de pages. L'accès en mémoire est fait en 100 nano-secondes, l'accès aux registres associatifs est fait en 10 nano-secondes.
Rappelons que le taux de présence p est la probabilité qu'une correspondance pages/cadres se trouve dans les registres associatifs.

1. Proposez une formule pour calculer le temps effectif d'accès en mémoire en fonction du taux de présence p. Expliquez cette formule.

   Solution. Posons

   $$te(p) = p * (10 + 100) + (1 - p) * (10 + 2 * 100) = 210 - p * 100\,.$$

   Cette formule s'explique comme suit. Le temps effectif $te(p)$ est :
   - avec probabilité p, la correspondance se trouve dans les registres associatifs. Dans ce cas, il faut prendre en compte la somme des temps
     - d'un accès au registres,
     - d'un accès à la mémoire,
   - sinon, avec probabilité $1 - p$, la correspondance n'est pas bufferisée dans les registres associatifs. Dans ce cas, il faut prendre en compte la somme des temps :
     - d'un accès au registres (il faut bien vérifier si la correspondance est dans les registres),
     - d'une consultation de la table des pages, et donc d'un accès à la mémoire,
     - d'un acces à un adresse physique, une fois faite la traduction des adresses logiques vers les adresse physiques.

   □

2. Quel est le temps effectif d'accès en mémoire si le taux de présence est 0,85 ?

Solution.

$$te(0,85) = 210 - 0,85 * 100 = 125\,.$$

□

3. Quel taux de présence est nécessaire si on souhaite obtenir un rapport entre temps effectif d'accès en mémoire et temps d'accès en mémoire plus petit que $1,2$? Est il possible modifier le taux de présence pour réduire ce rapport à un valeur plus petit que $1,1$?

   Solution. On veut

   $$\frac{te(p)}{100} = 2,1 - p < 1,2$$

   ce qui est équivalent à :

   $$0,90 < p$$

   Par contre, la condition

   $$\frac{te(p)}{100} = 2,1 - p < 1,1$$

   donne

   $$1 < p$$

   ce qui n'est pas possible, car la probabilité p est toujours plus petite que 1. □

Exercice 3 : sémaphores.

1. Expliquez ce qu'est un sémaphore.

   Solution. Un sémaphore est une structure de données, composé par une variable entière n qui peut être nulle ou positive, et de deux opérations UP et DOWN.
   - Opération DOWN : signale qu'on veut « prendre possession » de la ressource associée au sémaphore. On décrément la variable n, et si cette variable vaut 0, le processus ayant faite cette opération s'endort. Il sera réveillé de quand la variable entière n sera > 0.
   - Opération UP : signale qu'on « libère » la ressource associée au sémaphore. On incrémente la variable n, et si cette variable vaut 0 avant l'incrémentation, on réveille un processus endormi sur ce sémaphore (en attente que cette la ressource associée au sémaphore se libère).

   Les opération UP et DOWN sont atomiques : par exemple, dans UP, les opérations "tester la variable, l'incrémenter, et si elle étai 0 alors réveiller un processus" ne peuvent pas être entrelacés par des modification de la valeur de la variable par des autres processus. □

2. Proposez un exemple, d'intérêt pour la programmation des systèmes d'exploitation, où l'utilisation des sémaphores est souhaitable.

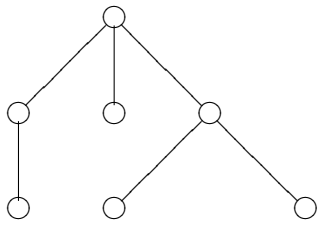
   Solution. Si la variable n vaut ou bien 0, ou bien 1, alors on appel un tel sémaphore *mutex*. Un mutex peut être utilisé pour garantir l'exclusion mutuelle entre processus. Plusieurs structures de données d'un système d'exploitation peuvent être partagés par différents « threads » d'exécutions : pour garantir leur cohérence on les accède en exclusion mutuelle. Par exemple, dans la gestion des bloques en mémoire (voir les caches dans le système de gestion des fichiers de UNIX), la mise à jour des deux listes de bloques libres et occupés est faite en exclusion mutuelle. □

Programmation POSIX

Processus et signaux

Les valeurs de retour des appels système ne sont pas systématiquement testées dans les programmes de l'énoncé : on suppose qu'ils ne renvoient jamais un code d'erreur. Vous pouvez faire de même dans vos solutions.

Exercice 4. Écrire quelque ligne de code C dont l'exécution engendre 6 processus liés au processus ancêtre par l'arbre généalogique suivant :



Solution. On peut se servir des lignes 5 à 8 du programme `processus.c` :

```
1  #include <unistd.h>

3  int main(void)
4  {
5    if(fork())
6      fork() && (fork() || (fork() && fork()));
7    else
8      fork();

10   sleep(2);
11   return 0;
12 }
```

Pour tester le programme :

```
1      [lsantoca@localhost solutions]$ make procs
2      gcc   -Wall -W -pedantic    -c -o processus.o processus.c
3      gcc     processus.o   -o processus
4      ./processus & (ps f -o ppid,pid,cmd | grep processus$)
5      8738  8739          \_ ./processus
6      8739  8740          |   \_ ./processus
7      8740  8742          |   |   \_ ./processus
8      8739  8743          |   \_ ./processus
9      8739  8744          |   \_ ./processus
10     8744  8745          |       \_ ./processus
11     8744  8746          |       \_ ./processus
```

Remarquons que le code

```
1  #include <unistd.h>

3  int main(void)
4  {
5    (fork() || fork())
6      && fork() && (fork() || (fork() && fork()));

8    sleep(2);
9    return 0;
10 }
```

ne marche pas :

```
1      [lsantoca@localhost solutions]$ make procs2
2      ./processus2 & (ps f -o ppid,pid,cmd  | grep processus2$)
3      9013  9014          \_ ./processus2
4      9014  9015          |   \_ ./processus2
5      9015  9016          |   |   \_ ./processus2
6      9015  9017          |   |   \_ ./processus2
7      9015  9018          |   |   \_ ./processus2
8      9018  9019          |   |       \_ ./processus2
9      9018  9020          |   |       \_ ./processus2
10     9014  9021          |   \_ ./processus2
11     9014  9022          |   \_ ./processus2
12     9022  9023          |       \_ ./processus2
13     9022  9024          |       \_ ./processus2
```

□

Exercice 5. Considérez le programme suivant :

```
1  #include <unistd.h>
2  #include <signal.h>
3  #include <stdio.h>

5  struct sigaction action;

7  void handler(int sig)
8  { return ; }

10 int main(void)
11 {
12   sigset_t tous_sauf_sigchld;

14   sigfillset(&tous_sauf_sigchld);
15   sigdelset(&tous_sauf_sigchld,SIGCHLD);

17   action.sa_handler = handler;
18   action.sa_flags = 0;
19   sigemptyset(&action.sa_mask);
20   sigaction(SIGCHLD,&action,NULL);

22   if(fork())
23     sigsuspend(&tous_sauf_sigchld);

25   printf("Fin de [%d]\n",getpid());

27   return 0;
28 }
```

1. On suppose le pid du père est 2000, et que le pid du fils est 2001. Dire ce qui peut s'afficher à l'écran.
2. Expliquer comment on obtient ces affichages. Par exemple, on pourra décrire ligne par ligne ce qui est accompli par le programme, en s'efforçant d'expliquer le fonctionnement et la dynamique des signaux.
3. Pendant l'exécution de ce programme, est-il possible que des signaux soient perdus ou ignorés (justifiez votre réponse) ? Si oui, proposez une amélioration du programme permettant de ne pas ignorer les signaux nécessaires au bon déroulement.

   Solution. Explication : À la ligne 23, le père bloque tous les signaux (ces signaux seront pris en compte au retour de l'appel système `sigsuspend`) sauf `SIGCHLD`, et il s'endort. Il sera réveillé par le premier signal lui délivré, c.-à-d. `SIGCHLD` car les autres sont bloqués.
   On a deux possibles affichages. L'affichage souhaité est :

```
1      [lsantoca@localhost solutions] synchro
2      Fin de [2001]
3      Fin de [2000]
4      [lsantoca@localhost solutions]
```

Avec cet affichage on suppose que le fils se termine après que le père se met en attente du signal `SIGCHILD`. Le signal `SIGCHILD`, envoyé par le fils au père à la terminaison du fils (le fils a donc déjà affiché son PID), réveille et débloque le père qui affiche son PID.

L'autre possible affichage est :

```
1      [lsantoca@localhost solutions] synchro
2      Fin de [2001]
```

Ici, par contre, le fils se termine et envoie le signal `SIGCHILD` avant que le père se met en attente de ce signal. Le signal `SIGCHILD` est traité par la routine `handler`, mais le père restera bloqué.

Dans ce deuxième cas, si le but de l'envoie du signal `SIGCHLD` est de réveiller le père, alors on peut bien dire que ce signal n'accompli pas son rôle, et donc il est ignoré (car traité trop tôt).

Pour éviter que le père se bloque à cause d'une livraison précoce su signal `SIGCHLD`, on peut retarder la livraison du signal `SIGCHLD` jusqu'à ce que le père appelle `sigsuspend`. On accompli ce but en le masquant (ou bloquant, ce qui est la même chose). Le signal `SIGCHLD` sera alors débloqué par le père pendant l'exécution de `sigsuspend`.

Nous accomplissons cette tache par le programme suivant, dont les commentaires nous expliquent aussi le fonctionnement du programme.

```
1  #include <unistd.h>
2  #include <signal.h>
3  #include <stdio.h>

5  struct sigaction action;

7  void handler(int sig)
8  { return ; }

10 int main(void)
11 {
12   sigset_t tous_moins_sigchld, seulement_sigchld;

14   /* On crée un ensemble de signaux,
15      contenant tous les signaux sauf SIGCHLD */
16   sigfillset(&tous_moins_sigchld);
17   sigdelset(&tous_moins_sigchld,SIGCHLD);

19   /* On crée un ensemble de signaux,
20      contenant seulement SIGCHLD */
21   sigemptyset(&seulement_sigchld);
22   sigaddset(&seulement_sigchld,SIGCHLD);

24   /* On interceptera le signal SIGCHLD par handler :
25    on fait cela en 2 temps :
26    - initialisation de la structure action,
27    - appel système sigaction
28    Le tous est équivalent à
29    signal(SIGCHLD,handler)
30   */
31   action.sa_handler = handler;
32   action.sa_flags = 0;
33   sigemptyset(&action.sa_mask);
34   sigaction(SIGCHLD,&action,NULL);

36   /* Avant le fork et
```

```
37       tant que père et fils ne sont pas prêts
38       on bloque SIGCHLD */
39    sigprocmask(SIG_BLOCK,&seulement_sigchld,NULL);

41    if(fork())
42      /*
43         le père se mets dans l'état endormi,
44         en attente de SIGCHLD :
45         - SIGCHLD n'est plus masqué,
46         - tous les autres signaux sont masqué.
47      */
48      sigsuspend(&tous_moins_sigchld);

50    /*
51       Le père se réveille seulement à cause
52       de la réception de SIGCHLD
53       (traité par la procédure handler de type "rien faire")
54       Ce signal il est lui envoyé par le fils à sa terminaison
55    */

57    /* Affichage finale, pour chaque processus :
58       si le signal SIGCHLD n'a pas été ignoré par le père,
59       on aura deux affichages en total
60    */
61    printf("Fin de [%d]\n",getpid());

63    return 0;
64  }
```

□

Exercice 6. Le programme suivant

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>

5  int main(void)
6  {
7    printf("Bonjour");
8    execlp("ls","ls",NULL);
9    printf("Exécution de ls accomplie.\n");
10   exit(EXIT_SUCCESS);
11 }
```

ne s'exécute pas comme prévu : quelque chaîne de caractères n'est pas affichée.

1. Dire ce qui est affiché à l'écran.

   Solution. Seulement le contenu du répertoire courant est affiché. □

2. En expliquer la raison.

   Solution.
   - La chaîne `Bonjour` ne s'affiche pas : en effet la fonction `printf` copie cette chaîne dans un tampon dans l'espace d'adressage du processus, et ce tampon n'est pas vidé. L'appel système suivant `execlp` recouvrera l'espace d'adressage du processus avec le code et les données de la commande `ls`. En faisant ceci, elle écraserai cette chaîne, qui sera donc jamais affichée.
   - La chaîne `Execution de ls accomplie` ne s'affiche pas. Encore, l'appel système suivant `execlp` recouvrera l'espace d'adressage du processus avec le code et les données de la commande `ls`. On ne revient pas de cet appel (pourvu que l'appel système n'a pas échoue).

□

3. Réécrire ce programme, de façon que toutes les chaînes de caractères et le contenu du répertoire courant s'affichent dans l'ordre suggéré par le programme. On se servira de l'appel système `exec` pour exécuter la commande `ls`.

Solution.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <wait.h>

6  int main(void)
7  {
8    printf("Bonjour");
9    fflush(stdout);
10   if(!fork())
11     execlp("ls","ls",NULL); /* On assume que cela et fork n'échouent pas */
12   else
13     {
14       wait(NULL);
15       printf("Exécution de ls accomplie.\n");
16     }
17   exit(EXIT_SUCCESS);
18 }
```

□

La communication par tubes

Exercice 7. On se propose d'implanter la ligne shell `ls -l | wc -l` (qui compte le nombre d'entrées dans le répertoire courant) par le programme `lslwcl.c` suivant :

```
1  #include <unistd.h>
2  #include <stdlib.h>

4  int main(void)
5  {
6    int p[2];

8    pipe(p);

10   if(fork())
11     {
12       close(STDIN_FILENO);
13       dup(p[0]);
14       close(p[0]);

16       execlp("wc","wc","-l",NULL);
17      }
18   else
19     {
20       close(STDOUT_FILENO);
21       dup(p[1]);
22       close(p[1]);

24       execlp("ls","ls","-l",NULL);
25      }
26   exit(EXIT_FAILURE);
```

```
27 }
```

Ce programme contient un erreur de traitement des tubes.

1. Que se passe-t'il à l'exécution ?

   Solution. Le père, en exécutant la commande `wc`, sera bloqué en attente de données sur le tube. □

2. Expliquez la raison du comportment inattendu du programme.

   Solution. La cause est que le père a oublié de fermer son descripteur à l'entrée du tube. Le père est donc écrivain sur ce tube, et le nombre d'écrivains ne tombera jamais à zéro. Cette condition étant nécessaire afin de signaler la fin de donnés, le père restera en attente de la fin des données du tube. □

3. Corrigez le programme par conséquent.

   Solution.

```
 1 #include <unistd.h>
 2 #include <stdlib.h>

 4 int main(void)
 5 {
 6   int p[2];

 8   pipe(p);

10   if(fork())
11     {
12       close(p[1]);
13       /* si non le nombre d'écrivain
14          ne tombera jamais à 0,
15          et le programme se bloque */

17       close(STDIN_FILENO);
18       dup(p[0]);
19       close(p[0]);

21       execlp("wc","wc","-l",NULL);
22      }
23   else
24     {
25       close(p[0]);
26       /* moins important,
27          bon style de programmation */

29       close(STDOUT_FILENO);
30       dup(p[1]);
31       close(p[1]);

33       execlp("ls","ls","-l",NULL);
34      }

36   exit(EXIT_FAILURE);
37 }
```

□

4. Réécrire le programme `lslwcl.c` en utilisant un tube nommée `lslwcl.tube` au lieu d'un tube anonyme.

   Solution.

```
 1 #include <unistd.h>
 2 #include <stdlib.h>
 3 #include <fcntl.h>
 4 #include <sys/stat.h>

 6 #define TUBE "lslwc.tube"

 8 int main(void)
 9 {
10   int p[2];

12   /* On ne fait pas ici de contrôles sur les valeur de retour
13      des appels système :-( */

15   unlink(TUBE); mkfifo(TUBE,0600);

17   if(fork())
18     {
19       p[0] = open(TUBE,O_RDONLY);

21       close(STDIN_FILENO);
22       dup(p[0]);
23       close(p[0]);

25       execlp("wc","wc","-l",NULL);
26      }
27   else
28     {
29       p[1] = open(TUBE,O_WRONLY);

31       close(STDOUT_FILENO);
32       dup(p[1]);
33       close(p[1]);

35       unlink(TUBE);
36       execlp("ls","ls","-l",NULL);
37      }

39   exit(EXIT_FAILURE);
40 }
```

□