Mercredi 5 janvier 2005
Durée : 3 heures
Documents autorisés

Examen : systèmes d'exploitation
L3, parcours informatique, 2004-2005

Corrigé préliminaire et partiel.

Examen

Les valeurs de retour des primitives ne sont pas systématiquement testées dans les programmes de l'énoncé. On supposera donc que les primitives ne renvoient jamais un code d'erreur.

Les processus

Exercice 1.

1. Combien de processus engendre l'évaluation de la commande C

   ```
   fork() && ( fork() || fork() ) ;
   ```

2. Dessiner l'arbre généalogique des processus engendrées par cette ligne.

Solution. Le processus courant (appelons-le le père) engendre dans l'ensemble 3 autres processus. En effet, comme dans une instruction `a && b`, `b` n'est pas évaluée si l'évaluation de `a` donne `0`, de même, dans dans une instruction `a || b`, `b` n'est pas évaluée si l'évaluation de `a` ne donne pas `0`. Donc, dans `fork() && b` seulement le père exécute `b`, et dans `fork() || b` seulement le fils exécute `b`.
Pour s'en convaincre, nous avons écrit le programme suivant :

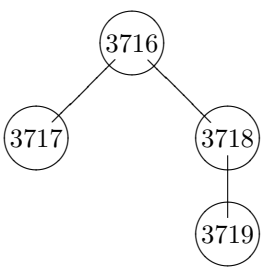
forks.c

```
1 : #include <unistd.h>
2 :
3 : int main(void)
4 : {
5 :   fork() && (fork() || fork());
6 :   sleep(10);
7 :   _exit(0);
8 : }
```

On peut tester donc comme il suit :

```
[lsantoca@localhost janvier050105]$ gcc -Wall -pedantic forks.c -o forks
[lsantoca@localhost janvier050105]$ forks & (sleep 1; ps -o "%P%p%c")
[3] 3716
 PPID   PID COMMAND
 3273  3277 bash
 3277  3716 forks
 3716  3717 forks
 3716  3718 forks
 3718  3719 forks
 3277  3720 bash
 3720  3722 ps
```

On voit donc l'arbre généalogique des processus `forks` :



□

Exercice 2. Considérer le programme suivant :

forkpause.c

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>
 3 : #include <unistd.h>
 4 : #include <signal.h>
 5 :
 6 : void interruption(int signum)
 7 : {
 8 :   if (signum == SIGINT)
 9 :     printf("UN\n");
10 : }
11 :
12 : int main(void)
13 : {
14 :   int pid;
15 :
16 :   signal(SIGINT, &interruption);
17 :   signal(SIGALRM, &interruption);
18 :   pid = fork();
19 :   srand(pid);
20 :   if (!pid)
21 :   {
22 :     sleep(rand() % 2);
23 :     printf("DEUX\n");
24 :     kill(getppid(), SIGINT);
25 :   } else
26 :   {
27 :     alarm(5);
28 :     sleep(rand() % 2);
29 :     printf("TROIS\n");
30 :     pause();
31 :   }
32 :   exit(EXIT_SUCCESS);
33 : }
```

1. Répondre aux questions suivants, en expliquant votre réponse.
   - Que peut se passer si l'on supprime la ligne `30` ?

     Solution. Dans ce cas, le père peut se terminer avant recevoir le signal `SIGINT` par le fils. On pourra donc obtenir les affichages
     `DEUX`
     `TROIS`
     ou
     `TROIS`
     `DEUX`

     □

   - Que peut se passer si l'on supprime la ligne `27` ?

     Solution. Si le signal `SIGINT` est envoyé au père avant que celui exécute la ligne `30`, le père se bloquera en attente d'un signal. Afin de prévenir cette désagréable situation, la ligne `27` prépare un envoi du signal `SIGALRM` après 5 secondes. □

   - Que se passe-t-il si on échange l'ordre des lignes `18-19` ?

     Solution. L'initialisation du générateur de nombres aléatoires est la même pour le fils et le père, car la valeur de `pid` est la même. La temporisation aléatoires du fils et du père (lignes 22 et 28) est donc la même. □

2. Donner les différents affichages pouvant être produits par ce programme.

   Solution. Car l'affichage de `DEUX` précède l'envoi du signal `SIGINT` du fils au père, et ce signal déclenche l'affichage de `UN` par le père, on ne peut pas évidemment avoir un affichage de `UN` précédent un affichage de `DEUX`. On obtient donc les affichages :

   ```
   [lsantoca@localhost janvier050105]$ forkpause
   TROIS
   DEUX
   ```

```
UN
[lsantoca@localhost janvier050105]$ forkpause
DEUX
UN
TROIS
[lsantoca@localhost janvier050105]$ forkpause
DEUX
TROIS
UN
```

bien que le dernier soit assez improbable. □

La communication par signaux

signaux.c

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>
 3 : #include <unistd.h>
 4 : #include <signal.h>
 5 :
 6 : sigset_t ens_vide, ens_tstp;
 7 : struct sigaction action;
 8 :
 9 : void handler(int sig)
10 : {
11 :   switch (sig)
12 :   {
13 :   case SIGQUIT:
14 :     sigprocmask(SIG_SETMASK, &ens_vide, NULL);
15 :     sigprocmask(SIG_SETMASK, &ens_tstp, NULL);
16 :     sigaction(SIGINT, &action, NULL);
17 :     break;
18 :   case SIGINT:
19 :     printf("A+\n");
20 :     exit(0);
21 :     break;
22 :   case SIGTSTP:
23 :     printf("Bonjour\n");
24 :     break;
25 :   default:
26 :     break;
27 :   }
28 :   return;
29 : }
30 :
31 : int main(void)
32 : {
33 :   action.sa_handler = handler;
34 :   sigemptyset(&action.sa_mask);
35 :
36 :   sigemptyset(&ens_vide);
37 :   sigemptyset(&ens_tstp);
38 :   sigaddset(&ens_tstp, SIGTSTP);
39 :   sigprocmask(SIG_SETMASK, &ens_tstp, NULL);
40 :
41 :   sigaction(SIGTSTP, &action, NULL);
42 :   sigaction(SIGQUIT, &action, NULL);
43 :
44 :   for (;;);
45 :
46 :   exit(0);
47 : }
```

On rappelle que la frappe de `CTRL-C` envoie le signal `SIGINT` au processus qui se trouve en avant-plan. De même pour `CTRL-\` et `SIGQUIT` (on suppose que l'on se trouve pas sur une machine Sun) et `CTRL-Z` et `SIGTSTP`.

Exercice 3. Commenter, ligne par ligne, le programme `signaux.c`, en donnant assez d'explications.

Exercice 4. Dire ce qui est affiché à l'écran si – pendant l'exécution en avant-plan du programme ci dessus – l'on tape :

1. `CTRL-Z`, `CTRL-C`,

   Solution.

   ```
   [lsantoca@localhost janvier050105]$ signaux

   [lsantoca@localhost janvier050105]$
   ```

   □

2. `CTRL-\`, `CTRL-C`,

Solution.

```
[lsantoca@localhost janvier050105]$ signaux
A+
[lsantoca@localhost janvier050105]$
```

□

3. CTRL-Z, CTRL-\, CTRL-C,

   Solution.

   ```
   [lsantoca@localhost janvier050105]$ signaux
   Bonjour
   A+
   [lsantoca@localhost janvier050105]$
   ```

   □

4. CTRL-\, CTRL-Z, CTRL-C,

   Solution.

   ```
   [lsantoca@localhost janvier050105]$ signaux
   A+
   [lsantoca@localhost janvier050105]$
   ```

   □

5. CTRL-\, CTRL-Z, CTRL-\, CTRL-C,

   Solution.

   ```
   [lsantoca@localhost janvier050105]$ signaux
   Bonjour
   A+
   [lsantoca@localhost janvier050105]$
   ```

   □

6. CTRL-Z, CTRL-Z, CTRL-\, CTRL-C,

   Solution.

   ```
   [lsantoca@localhost janvier050105]$ signaux
   Bonjour
   A+
   [lsantoca@localhost janvier050105]$
   ```

   □

La communication par tubes

Considérer le programme suivant :

tubes.c

```
 1 : #include <stdlib.h>
 2 : #include <unistd.h>
 3 : #include <fcntl.h>
 4 : #include <sys/stat.h>
 5 :
 6 : char *nom_pf = "tube_pf";
 7 : char tampon[100];
 8 : int no_lu;
 9 :
10 : void pere(d_lecture)
11 : {
12 :   int d_ecriture;
13 :
14 :   while ((no_lu = read(d_lecture, tampon, sizeof(tampon))) > 0)
15 :   {
16 :     d_ecriture = open(nom_pf, O_WRONLY);
17 :     write(d_ecriture, tampon, no_lu);
18 :     close(d_ecriture);
19 :   }
20 :   exit(0);
21 : }
22 :
23 : void fils(d_ecriture)
24 : {
25 :   char *message = "abcdefghilmno\n";
26 :   int d_lecture = open(nom_pf, O_RDONLY);
27 :
28 :   do
29 :   {
30 :     if (*message == 0)
31 :       break;
32 :     write(d_ecriture, message++, sizeof(char));
33 :     no_lu = read(d_lecture, tampon, sizeof(tampon));
34 :     write(1, tampon, no_lu);
35 :   }
36 :   while (no_lu > 0);
37 :
38 :   exit(0);
39 : }
40 :
41 : int main(void)
42 : {
43 :   int fp[2];
44 :
45 :   unlink(nom_pf);
46 :   mkfifo(nom_pf, 0600);
47 :   pipe(fp);
48 :
49 :   if (fork())
50 :     pere(fp[0]);
51 :   else
52 :     fils(fp[1]);
53 :
54 :   exit(0);
55 : }
```

.

Exercice 5. Donner un aperçu sur le programme, en expliquant de façon succincte ce qui devrait faire.

Solution. Le but du programme (une fois corrigés les erreurs qui suivent), est d'afficher le message `abcdefghilmno` à l'écran. Le message est écrit par le fils, caractère par caractère, dans une tube anonyme. Le père renvoie le même message (de que des données sont disponibles dans le tube anonyme) au fils à travers un tube nommé. Le fils lit le message (de que des données sont disponibles) du tube nommé est l'affiche à l'écran. □

Exercice 6. Dans le programme on y trouve des erreurs de traitement des tubes : une première erreur regarde le

traitement des tubes nommées, une deuxième erreur regarde le traitement des tubes anonymes.
Trouver ces deux erreurs, pour chacune erreur expliquer la raison pour lequel il s'agit d'une erreur et le corriger.

Solution. Erreur traitement des tubes anonymes : le père ne ferme pas son descripteur sur l'entrée du tube et donc le nombre d'écrivains ne tombera jamais à 0. Le père se retrouvera bloqué à la ligne 14 si le tube anonyme sera vide : il s'agit d'un auto-bloquage plutôt que d'un inter-bloquage.
Comme correction, on peut ajouter

```
close(fp[1]);
```

avant la ligne 50.
Erreur traitement des tubes nommées : observons que le fils est bloqué à la ligne 22 quand il ouvre le tube nommé. Pour se débloquer il faut que quelque autre processus (le père) ouvre le même tube en écriture.
D'ailleurs le père, avant ouvrir ce même tube en écriture (ligne 16) se trouve en attente de données par le fils sur la tube anonyme. On a donc inter-bloquage entre fils et père.
Une simple solution et de déplacer la ligne 16 avant le boucle `do ... while`, et d'effacer la ligne 18 (le descripteur `d_ecriture` du père sera fermé par `exit`). □

Le shell et les scripts

Exercice 7. Considérer le script `pluff.sh`.

pluff.sh

```
 1 : #!/bin/bash
 2 :
 3 : if [ "$1" -eq  0 ]
 4 :     then
 5 :     echo $1
 6 : else
 7 :     ARG=‘expr $1 - 1‘
 8 :     $0 $ARG $2 | $2
 9 : fi
```

Écrire un programme en langage C équivalent au script `pluff.sh`. On utilisera les primitives `pipe`, `dup`, `execlp`.

Solution.

pluff.c

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>
 3 : #include <unistd.h>
 4 :
 5 : int main(int argc, char *argv[])
 6 : {
 7 :   int arg = atoi(argv[1]);
 8 :
 9 :   if (arg == 0)
10 :     printf("%s\n", argv[1]);
11 :   else
12 :   {
13 :     int tube[2];
14 :     char tampon[100];
15 :
16 :     pipe(tube);
17 :     if (fork())
18 :     {
19 :       close(STDIN_FILENO);
20 :       dup(tube[0]);
21 :       close(tube[0]);
22 :       close(tube[1]);
23 :       execlp(argv[2], argv[2], NULL);
24 :     } else
25 :     {
26 :       close(STDOUT_FILENO);
27 :       dup(tube[1]);
28 :       close(tube[0]);
29 :       close(tube[1]);
30 :       sprintf(tampon, "%d", --arg);
31 :       execlp(argv[0], argv[0], tampon, argv[2], NULL);
32 :     }
33 :     exit(EXIT_FAILURE);
34 :   }
35 :   exit(EXIT_SUCCESS);
36 : }
```

□

Exercice 8. Considérer le programme `succ.c` :

succ.c

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>
 3 :
 4 : int main(void)
 5 : {
 6 :   int i;
 7 :
 8 :   scanf("%d", &i);
 9 :   fprintf(stderr, "%d\n", 1);
10 :   printf("%d\n", ++i);
11 :   exit(0);
12 : }
```

On supposera qu'un exécutable `succ` (correspondant au programme `succ.c`) et le fichier `pluff.sh` se trouvent dans un des répertoires de la variable d'environnement `PATH`.

1. Dire ce qui est affiché à l'écran par la commande

   ```
   $ pluff.sh 3 succ
   ```

   Solution.

   ```
   [lsantoca@localhost janvier050105]$ pluff.sh 3 succ
   1
   1
   1
   3
   ```

   □

2. Combien de processus nommés `succ` sont engendrés par la même commande ? Combien de processus nommés `succ` sont engendrés par la commande

   ```
   $ pluff.sh n succ
   ```

   où n est un nombre entier positif.

   Solution. 3 et n processus `succ`, respectivement. □

3. Peut-on remplacer les lignes 9-10 du programme `succ.c` par la ligne

   ```
   printf("%d\n%d\n",1,++i);
                              ?
   ```

   Expliquer votre réponse.

   Solution. Il n'est pas possible de faire ce remplacement, si par exemple on veut avoir le même résultat avec la commande `pluff.sh 3 succ`. En effet, le `stderr` n'est pas rédirigé quand dans le shell on utilise les tubes `|`. □

Un peu de programmation

Exercice 9. Donner le schéma d'un programme serveur. Le serveur écoute sur une tube nommé, et, chaque fois qu'il reçoit une requête de connexion par un processus client, il crée un fils qui s'occupe de traiter la connexion à l'aide d'une nouvelle couple de tubes nommées (la communication est bidirectionnelle) dédiée à cette connexion. Le serveur ne s'arrête jamais.
On peut faire l'hypothèse qu'un client demande une connexion en écrivant son PID dans le tube du serveur. On détaillera seulement la gestion des tubes par le fils, et non le traitement de la connexion.