Mercredi 17 novembre 2004
Durée : 2 heures
Documents autorisés

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

Corrigé préliminaire : version définitive à apparaître ce fin de semaine.

Examen partiel

Le système des fichiers

Exercice 1. Le programme suivant présente une erreur.

idem.c

```
 1 : #include <stdlib.h>
 2 : #include <sys/stat.h>
 3 : #include <stdio.h>
 4 :
 5 : #define E_ARGS -2
 6 :
 7 : int main(int argc, char *argv[])
 8 : {
 9 :     struct stat info1, info2;
10 :
11 :     if (argc != 3)
12 :         exit(E_ARGS);
13 :
14 :     stat(argv[1], &info1);
15 :     stat(argv[2], &info2);
16 :
17 :     if (info1.st_dev == info2.st_dev && info1.st_ino == info2.st_ino)
18 :     {
19 :         printf("Fichiers identiques\n");
20 :         exit(EXIT_SUCCESS);
21 :     } else
22 :     {
23 :         printf("Fichiers différents\n");
24 :         exit(EXIT_FAILURE);
25 :     }
26 : }
```

En effet, pendant son exécution, il présente un comportement curieux :

```
$ idem fichier1.txt fichier1.txt
Fichiers différents
$ idem fichier2.txt fichier2.txt
Fichiers identiques
```

1. Expliquer la raison du comportement de l'exécutable `idem`.
2. Appliquer au programme les corrections nécessaires.

Solution.

1. Le fichier `fichier1.txt` n'existe pas, le fichier `fichier2.txt` existe :

   ```
   $ ls fichier1.txt fichier2.txt
   ls: fichier1.txt: No such file or directory
   fichier2.txt
   ```

   L'appel à la primitive `stat` ne contrôle pas la valeur de retour de cette primitive. Quand on appelle la primitive avec le chemin `fichier1.txt` on obtient un erreur que l'on prend pas en considération. Le contenu des structures `info1` et `info2` n'est pas touché. Car il s'agit de variables dynamiques (elle ne sont pas déclarées comme `static`) elle ne sont pas initialisées à zéro, et donc leur contenu est hasardeux et différent (avec une probabilité importante).

2. La solution évidente est celle d'ajouter de contrôles sur les valeurs de retour des appels à `stat` :

idemcorr.c

```
 1 : #include <stdlib.h>
 2 : #include <sys/stat.h>
 3 : #include <stdio.h>
 4 :
 5 : #define E_ARGS -2
 6 : #define E_NOFIC -3
 7 :
 8 : int sortie_err(const char *mess, int err)
 9 : {
10 :     perror(mess);
11 :     exit(err);
12 : }
13 :
14 : int main(int argc, char *argv[])
15 : {
16 :     struct stat info1, info2;
17 :
18 :     if (argc != 3)
19 :         sortie_err("nombre args", E_ARGS);
20 :     if (stat(argv[1], &info1) == -1)
21 :         sortie_err(argv[1], E_NOFIC);
22 :     if (stat(argv[2], &info2) == -1)
23 :         sortie_err(argv[2], E_NOFIC);
24 :
25 :     if (info1.st_dev == info2.st_dev && info1.st_ino == info2.st_ino)
26 :     {
27 :         printf("Fichiers identiques\n");
28 :         exit(EXIT_SUCCESS);
29 :     } else
30 :     {
31 :         printf("Fichiers différents\n");
32 :         exit(EXIT_FAILURE);
33 :     }
34 : }
```

```
$ idemcorr fichier1.txt fichier1.txt
fichier1.txt: No such file or directory
```

□

Exercice 2. Écrire un programme C qui affiche sur la sortie standard tous les liens symboliques contenus dans un répertoire passé en paramètre par l'utilisateur. Si l'utilisateur ne passe pas un tel répertoire, le répertoire courant est parcouru.

Solution.

lienssymb.c

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>
 3 : #include <sys/stat.h>
 4 : #include <dirent.h>
 5 : #include <string.h>
 6 :
 7 : #define TAILLE 1024
 8 :
 9 : int main(int argc, char *argv[])
10 : {
11 :     DIR *repertoire;
12 :     struct dirent *entree;
13 :     struct stat fic_info;
14 :     char chemin[TAILLE];
15 :     int longueur;
16 :
17 :     /*
18 :        Remarque :
19 :        strncat ajoute toujours '\0' a la fin de la chaîne
20 :        Pas la même chose avec strncpy
21 :      */
22 :     chemin[0] = 0;
23 :     if (argc > 1)
24 :         strncat(chemin, argv[1], TAILLE - 1);
25 :     else
26 :         strncat(chemin, "./", TAILLE - 1);
27 :
28 :     longueur = strlen(chemin);
29 :     if (chemin[longueur - 1] != '/')
30 :         strncat(chemin, "/", TAILLE - longueur - 1);
31 :
32 :     if ((repertoire = opendir(chemin)) == NULL)
33 :     {
34 :         fprintf(stderr, "%s: ", chemin);
35 :         perror("opendir");
36 :         exit(EXIT_FAILURE);
37 :     }
38 :
39 :     longueur = strlen(chemin);
40 :     while ((entree = readdir(repertoire)) != NULL)
41 :     {
42 :         chemin[longueur] = 0;
43 :         strncat(chemin, entree->d_name, TAILLE - longueur - 1);
44 :         /* Dans la prochaine ligne il faut utiliser */
45 :         /* lstat à la place de stat */
46 :         /* Voir la correction du TD2 */
47 :         /* Pas de pb. si vous avez ecrit stat */
48 :         if (lstat(chemin, &fic_info) == -1)
49 :         {
50 :             fprintf(stderr, "%s: ", chemin);
51 :             perror("lstat");
52 :             continue;
53 :         }
54 :         if (S_ISLNK(fic_info.st_mode))
55 :             printf("%s\n", entree->d_name);
56 :     }
57 :     exit(EXIT_SUCCESS);
58 : }
```

□

La communication par signaux

Exercice 3. Considérer le programme suivant :

signal.c

```
 1 : #include <signal.h>
 2 : #include <stdio.h>
 3 : #include <stdlib.h>
 4 :
 5 : static int n = 0;
 6 :
 7 : void interruption(int signum)
 8 : {
 9 :     signal(signum, interruption);
10 :     switch (signum)
11 :     {
12 :     case SIGINT:
13 :         n += 2;
14 :     case SIGTSTP:
15 :         n--;
16 :         if (n != 0)
17 :             signal(SIGINT, SIG_DFL);
18 :         else
19 :             signal(SIGINT, interruption);
20 :     }
21 :     printf("Valeur de n : %d.\n", n);
22 : }
23 :
24 : int main(void)
25 : {
26 :     signal(SIGINT, interruption);
27 :     signal(SIGTSTP, interruption);
28 :     for (;;);
29 :     exit(EXIT_SUCCESS);
30 : }
```

1. Dire ce qui est affiché à l'écran si, pendant l'exécution du programme en avant-plan, on tape à la console :
   - `CTRL-C`,
   - `CTRL-C, CTRL-C`,
   - `CTRL-Z, CTRL-C`,
   - `CTRL-C, CTRL-Z, CTRL-C`.
2. Réécrire le programme `signal.c` en utilisant l'interface de programmation POSIX pour les signaux.

Solution. Voici le comporemen different su programme : Frappe de `CTRL-C` :

```
$ a.out
Valeur de n : 1.
```

Frappe de `CTRL-C, CTRL-C` :

```
$ a.out
Valeur de n : 1.

$
```

Frappe de `CTRL-Z, CTRL-C` :

```
$ a.out
Valeur de n : -1.

$
```

Frappe de `CTRL-C, CTRL-Z, CTRL-C` :

```
$ a.out
Valeur de n : 1.
Valeur de n : 0.
Valeur de n : 1.
```

Le programme avec l'interface POSIX est le suivant :

sigaction.c

```
 1 : #include <signal.h>
 2 : #include <stdio.h>
 3 : #include <stdlib.h>
 4 :
 5 : static int n = 0;
 6 :
 7 : static struct sigaction action;
 8 :
 9 : void interruption(int signum)
10 : {
11 :     switch (signum)
12 :     {
13 :     case SIGINT:
14 :         n += 2;
15 :     case SIGTSTP:
16 :         n--;
17 :         if (n != 0)
18 :         {
19 :             action.sa_handler = SIG_DFL;
20 :             sigaction(SIGINT, &action, NULL);
21 :         } else
22 :         {
23 :             action.sa_handler = interruption;
24 :             sigaction(SIGINT, &action, NULL);
25 :         }
26 :     }
27 :     printf("Valeur de n : %d.\n", n);
28 : }
29 :
30 : int main(void)
31 : {
32 :     sigemptyset(&action.sa_mask);
33 :     action.sa_handler = interruption;
34 :     sigaction(SIGINT, &action, NULL);
35 :     sigaction(SIGTSTP, &action, NULL);
36 :
37 :     for (;;);
38 :     exit(EXIT_SUCCESS);
39 : }
```

□

Fork et Wait

Exercice 4. Considérer le programme suivant :

forkwait.c

```
 1 : #include <stdlib.h>
 2 : #include <unistd.h>
 3 : #include <stdio.h>
 4 : #include <wait.h>
 5 :
 6 : static int n = 0;
 7 :
 8 : int main(void)
 9 : {
10 :     pid_t pid, pid1, pid2;
11 :
12 :     if ((pid = fork()) == -1)
13 :         exit(EXIT_FAILURE);
14 :
15 :     if (pid != 0)
16 :     {
17 :         n--;
18 :         pid1 = pid;
19 :         pid2 = getpid();
20 :         pid = wait(NULL);
21 :     } else
22 :     {
23 :         n++;
24 :         pid1 = getppid();
25 :         pid2 = getpid();
26 :     }
27 :     printf("%d\t%d\n", pid1, pid2 + n);
28 :     exit(EXIT_SUCCESS);
29 : }
```

1. Détailler (ligne par ligne) le comportement de ce programme.
2. On suppose que le PID du père est 2000, et que le PID du fils est 2001 : dire ce qui est affiché à l'écran.

Solution. Le programme déclare 3 variables de type `pid_t`. Après il fait un appel à la primitive `fork` avec test de la valeur de retour (affectée à la variable `pid`).
Si ce valeur de retour est 0 (on est le fils) on incrémente la valeur de `n` (donc après la ligne 23 elle vaut 1 pour le fils), on demande le PID du pére et son propre PID. Avant sortir, on affiche le PID du pére suivi par son PID incrémentée par 1.
Si la valeur de retour de fork est le PID du fils (on est le père) on décrémente la variable `n` (donc après la ligne 17 elle vaut -1 pour le père), `pid1` contient le PID du fils, et `pid2` contient son pid.
Après la terminaison du fils (primitive `wait`) on imprime à l'écran le PID du fils, et son PID décrémenté par 1.
Voici par exemple ce qui est affiché si le PID du père est 2000 et le PID du fils est 2001 :

```
$a.out
2000 2002
2001 1999
$
```

□

Exercice 5. Considérer l'instruction C suivante :

```
fork() && fork() * fork();
```

On suppose que tout appel à la primitive `fork` ne renvoie pas un code d'erreur.

1. Dire combien de processus cette instruction engendre (on ne compte pas le père).
2. Faire un dessin de l'arbre généalogique du père et des processus engendrés.

Solution. On rappelle que l'opérateur `*` est prioritaire sur l'opérateur `&&` et donc il faut mettre les parenthèses comme il suit :

```
fork() && (fork() * fork());
```

Le numéro de fils crée est 4. On peut vérifier le résultat à l'aide du programme

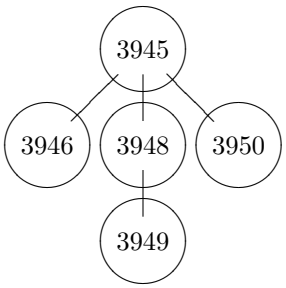
fffork.c

```
 1 : #include <stdio.h>
 2 : #include <unistd.h>
 3 :
 4 : int main(void)
 5 : {
 6 :     fork() && fork() * fork();
 7 :     sleep(2);
 8 :     return 0;
 9 : }
```

et de la commande

```
$ a.out & ps -C a.out -o "%P %p"
 PPID   PID
 3220  3945
 3945  3946
 3945  3948
 3948  3949
 3945  3950
```

L'arbre genealogique est donc :



□

Exercice 6. Écrire un programme qui engendre un processus et se termine. Le processus fils, dès qu'il devient orphelin, affichera à l'écran un message permettant à l'utilisateur de se convaincre que ce processus est orphelin.

Solution.

orphelin.c

```
 1 : #include <unistd.h>
 2 : #include <stdio.h>
 3 : #include <stdlib.h>
 4 :
 5 : int main(void)
 6 : {
 7 :     switch (fork())
 8 :     {
 9 :     case -1:
10 :         exit(EXIT_FAILURE);
11 :     case 0:
12 :         sleep(1);
13 :         printf("Mon père est le processus %d (init).\n", (int) getppid());
14 :     default:
15 :         ;
16 :     }
17 :     exit(EXIT_SUCCESS);
18 : }
```

□

Les tubes anonymes

Exercice 7. Le but du programme suivant est d'écrire la chaîne de caractères `"abcde"` dans une tube, de la lire de cette tube, et enfin de l'afficher à l'écran :

pipe.c

```
 1 : #include <stdlib.h>
 2 : #include <unistd.h>
 3 : #include <stdio.h>
 4 :
 5 : static int tube[2];
 6 :
 7 : void ecrire(void)
 8 : {
 9 :     char *message = "abcde";
10 :
11 :     while (*message != 0)
12 :         write(tube[1], message++, 1);
13 :     exit(EXIT_SUCCESS);
14 : }
15 :
16 : void lire(void)
17 : {
18 :     int no_lu;
19 :     char tampon[100], *curr = tampon;
20 :
21 :     while ((no_lu = read(tube[0], curr, 2)) > 0
22 :            && curr < (tampon + sizeof(tampon) - 1))
23 :         curr += no_lu;
24 :     printf("%s", tampon);
25 :     exit(EXIT_SUCCESS);
26 : }
27 :
28 : int main(void)
29 : {
30 :     if (pipe(tube) == -1)
31 :         exit(EXIT_FAILURE);
32 :     switch (fork())
33 :     {
34 :     case -1:
35 :         exit(EXIT_FAILURE);
36 :     case 0:
37 :         ecrire();
38 :     default:
39 :         lire();
40 :     }
41 :     exit(EXIT_FAILURE);
42 : }
```

Dans ce programme on y trouve deux erreurs : le premier est une erreur du traitement des objets système. L'autre est plus proprement une erreur du langage C. Trouvez-les, en bien expliquant de quel type d'erreur il s'agit, et proposez des corrections au programme.

Solution. Un premier erreur est que le père ne ferme pas son descripteur sur l'entre de la tube. Le nombre d'écrivains sur le tube ne tombera jamais à zéro, avec le résultat que un essai de lecture (par le père) sera bloquante.
Le deuxième erreur est qu'on écrit dans le tube les caractères `'a','b','c','d','e'`, mais l'on oublie le terminateur de chaînes `'\0'`. Une fois corrigé le premier problème l'affichage à l'écran sera hasardeux (par exemple : `abced,@ø-@Ððÿ¿,@ø-@àòÿ¿`).
Le programme corrigé est le suivant :

pipecorr.c

```
 1 : #include <stdlib.h>
 2 : #include <unistd.h>
 3 : #include <stdio.h>
 4 :
 5 : static int tube[2];
 6 :
 7 : void ecrire(void)
 8 : {
 9 :     char *message = "abced";
10 :
11 :     close(tube[0]);             /* Cette ligne n'est pas strictement nécessaire */
12 :     do
13 :         write(tube[1], message, 1);
14 :     while (*message++ != 0);    /* On écrit dans le tube le terminateur de chaînes aussi */
15 :     exit(EXIT_SUCCESS);
16 : }
17 :
18 : void lire(void)
19 : {
20 :     int no_lu;
21 :     char tampon[100], *curr = tampon;
22 :
23 :     close(tube[1]);             /* Sans cette ligne le père se bloque */
24 :     while ((no_lu = read(tube[0], curr, 2)) > 0
25 :            && curr < (tampon + sizeof(tampon) - 1))
26 :         curr += no_lu;
27 :     printf("%s", tampon);
28 :     exit(EXIT_SUCCESS);
29 : }
30 :
31 : int main(void)
32 : {
33 :     if (pipe(tube) == -1)
34 :         exit(EXIT_FAILURE);
35 :     switch (fork())
36 :     {
37 :     case -1:
38 :         exit(EXIT_FAILURE);
39 :     case 0:
40 :         ecrire();
41 :     default:
42 :         lire();
43 :     }
44 :     exit(EXIT_FAILURE);
45 : }
```

□

Exercice 8. Écrire un programme qui crée une tube et puis engendre un processus fils. Le fils lit des caractères de l'entrée standard et transforme les minuscules en majuscules, le résultat étant écrit dans le tube. Le père lit du tube, efface les espace répétées, et imprime ces caractères à l'écran (sortie standard).

Solution.

filtres.c

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>
 3 : #include <unistd.h>
 4 : #include <ctype.h>
 5 :
 6 : void fils(w_desc)
 7 : {
 8 :     int c;
 9 :
10 :     while ((c = getchar()) != EOF)
11 :     {
12 :         if (isprint(c) || c == '\n')
13 :         {
14 :             c = toupper(c);
15 :             write(w_desc, &c, sizeof c);
16 :         }
17 :     }
18 :     exit(EXIT_SUCCESS);
19 : }
20 :
21 : void pere(r_desc)
22 : {
23 :     int dernier = 'a', c;
24 :
25 :     while (read(r_desc, &c, sizeof c) > 0)
26 :     {
27 :         if (c != ' ' || dernier != ' ')
28 :             printf("%c", c);
29 :         dernier = c;
30 :     }
31 :     exit(EXIT_SUCCESS);
32 : }
33 :
34 : int main(void)
35 : {
36 :     int tube[2];
37 :
38 :     if (pipe(tube) == -1)
39 :         exit(EXIT_FAILURE);
40 :
41 :     switch (fork())
42 :     {
43 :     case -1:
44 :         exit(EXIT_FAILURE);
45 :     case 0:
46 :         close(tube[0]);
47 :         fils(tube[1]);
48 :     default:
49 :         close(tube[1]);
50 :         pere(tube[0]);
51 :     }
52 :     exit(EXIT_FAILURE);
53 : }
```

□