

Université Paris 7
Licence 3 Informatique – Systèmes
6 janvier 2011

Durée : 3 heures. Livres et photocopies de livres interdits. Ordinateurs interdits.

Notes de cours, TD, manuscrites ou imprimées autorisées.

Le sujet comporte 6 pages.

Votre code doit être écrit de façon lisible, avec des indentations et les accolades appropriées permettant de voir la fin de blocs de code (fin de boucles, etc.).

Exercice 1

Que contient le fichier `fifi` après l'exécution de programme suivant :

```
1 /***** zerbi.c *****/
2 #include <sys/types.h>
3 #include <sys/stat.h>
4 #include <fcntl.h>
5 #include <unistd.h>
6
7 int main(void){
8     int d,c;
9     char x;
10    d=open("fifi", O_CREAT|O_TRUNC|O_RDWR, S_IWUSR|S_IRUSR);
11
12    c=open("fifi", O_RDWR, S_IWUSR|S_IRUSR);
13
14    x='d';
15    write(d,&x,1);
16
17    x='c';
18    write(c,&x,1);
19    return 0;
20 }
```

Exercice 2

Que contient le fichier `fifi` après l'exécution de programme suivant :

```
1 /***** zerbi.c *****/
2 #include <sys/types.h>
3 #include <sys/stat.h>
4 #include <fcntl.h>
5 #include <unistd.h>
```

```

6
7  int main(void){
8      int d,c;
9      char x;
10     d=open("fifi", O_CREAT|O_TRUNC|O_RDWR, S_IWUSR|S_IRUSR);
11
12     c=dup(d);
13
14     x='d';
15     write(d,&x,1);
16
17     x='c';
18     write(c,&x,1);
19     return 0;
20 }

```

Exercice 3

Que contient le fichier fofo après l'exécution de programme suivant :

```

1  /***** aloes.c *****/
2  #include <sys/types.h>
3  #include <unistd.h>
4  #include <stdlib.h>
5  #include <sys/types.h>
6  #include <sys/stat.h>
7  #include <fcntl.h>
8
9
10 int main(int argc, char *argv[]){
11     int i;
12     int d;
13     char *a;
14     char *b;
15
16     i=atoi(argv[1]);
17     d=open(argv[2],O_RDWR);
18     if( d == -1 ) return EXIT_FAILURE;
19
20     a=malloc(i);
21     b=malloc(i);
22
23     read(d,a,i);
24     lseek(d,(off_t) -i, SEEK_END);
25     read(d,b,i);
26     lseek(d,(off_t) 0, SEEK_SET);
27     write(d,b,i);

```

```

28     lseek(d,(off_t) -i, SEEK_END);
29     write(d,a,i);
30     return 0;
31 }
32 /*****/

```

avec

aloes 1 fofo

si fofo contenait initialement cinq caractères : zorro.

Exercice 4

On considère le programme suivant :

```

1  /***** momo.c *****/
2  int main(int argc, char *argv[]){
3      int n,f;
4      pid_t pid;
5      n=atoi(argv[1]);
6      f = open(argv[2],O_WRONLY|O_TRUNC|O_CREAT,S_IRUSR|S_IWUSR);
7      if(f == -1){
8          perror("open");
9          exit(EXIT_FAILURE);
10     }
11     while( n > 1 ){
12         pid=fork();
13         if( pid == -1 ){
14             perror("fork");
15             exit(EXIT_FAILURE);
16         }
17         else if(pid == 0){
18             n--;
19             continue;
20         }
21         else{
22             write(f,&n,sizeof n) ;
23             break;
24         }
25     }
26     close(f);
27     exit(EXIT_SUCCESS);
28 }
29 /*****/

```

On le compile et on le lance

momo 4 f

Question 1: Combien de processus crée cette commande? (Inclure dans le compte le processus initial que nous avons lancé, donc on compte les processus lancés par momo et ses descendants et aussi momo lui-même.)

Question 2: Faire un arbre de processus qui sont engendrés par cette commande, à la racine mettre le processus initial. Pour chaque processus mettre les flèches vers ses fils (son fils). En fait combien de fils aura chaque processus?

Question 3: Sur le dessin que vous avez fait en réponse à la question précédente ajouter pour chaque processus dans l'arbre de processus soit le mot **rien** si ce processus n'écrit rien dans le fichier **f** soit le où les nombres écrits par ce processus dans le fichier **f**.

Question 4: Pour mettre un peu d'ordre dans les nombres écrits dans le fichier **f** on demande d'ajouter dans le code une (ou plusieurs) instruction de façon à ce que chaque processus attende la terminaison de ses fils (ou de son fils s'il y a un seul) avant d'écrire dans le fichier **f**. Quelle instruction ou quelles instructions faut-il ajouter? Indiquer aussi où.

Question 5: Si on a ajouté les instructions demandées dans la question précédente alors après la terminaison de programme quel sera le contenu exacte de fichier **f**?

Est-ce que le contenu de **f** à la fin de l'exécution est le même si le fichier n'existait pas initialement et si le fichier existait initialement et était non vide?

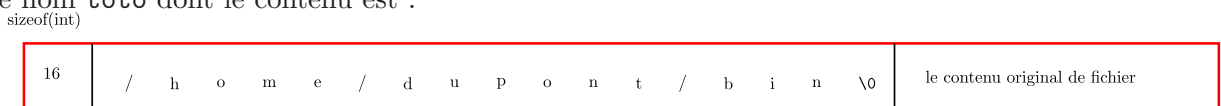
Exercice 5

Le but de cette exercice est d'écrire une commande **poubelle** qui permet de mettre les fichiers réguliers (et uniquement les fichiers réguliers) dans une poubelle. Il y aura une autre commande **recuperer** qui permettra de récupérer et reconstituer les fichiers présents dans la poubelle.

La poubelle c'est le répertoire **.poubelle** situé dans le répertoire d'accueil de l'utilisateur.

Chaque fichier stocké dans la poubelle aura le contenu particulier qui permettra de le reconstituer à l'emplacement original avec la commande **recuperer**. Par exemple si nous avons le fichier **toto** qui est initialement dans le répertoire **/home/dupont/bin**.

Dans ce cas la commande **poubelle** appliquée à **toto** va créer dans la poubelle un fichier de même nom **toto** dont le contenu est :



le chemin absolu

le nombre de caractères dans le chemin absolu

c'est-à-dire :

- le fichier dans la poubelle commence par un entier n (donc sur `sizeof(int)` octets) donnant la longueur de chemin **absolu** vers le fichier (dans notre exemple c'est `strlen('/home/dupont/bin')`, c'est-à-dire 16,
- ensuite c'est le chemin absolu y inclus le caractère nul à la fin (donc $n + 1$ octets),
- et à la fin c'est le contenu de fichier original, octet après octet.

L'écriture de la commande sera divisée en plusieurs questions, chaque question demande d'écrire une fonction particulière. Il n'est pas nécessaire de faire les questions dans l'ordre. Si vous écrivez la réponse pour une question i vous pouvez toujours assumer que vous avez déjà toutes les fonctions

demandées dans les questions précédentes, même si vous ne les avez pas faites. Donc inutile de s'attarder sur une question qui vous pose des problèmes, passez plutôt à la question suivante.

Dans votre code vous pouvez utiliser librement la fonction

```
char *concatenation(char *s,...)
```

(on suppose qu'elle est donnée, on ne demande pas de l'écrire). Cette fonction, avec le nombre variable d'arguments, concatène les chaînes de caractères passées en argument. Par exemple

```
concatenation("/home/dupont/bin", "/", "toto", (char *)NULL)
```

retournera une chaîne de caractères `"/home/dupont/bin/toto"`. Notez que le dernier paramètre de `concatenation` doit être toujours `NUL` (nécessaire pour déterminer le nombre d'arguments).

```
concatenation("toto", (char *)NULL)
```

retournera juste une copie de la chaîne `"toto"`.

De plus vous aurez besoin des fonctions POSIX. Voilà un rappel de quelques de ces fonctions qui peuvent être utiles (il y en a d'autres qui seront utiles aussi, je rappelle juste quelques fonctions qu'on a moins utilisées en cours/TD) :

- `getenv("HOME")` retourne le chemin absolu (sous forme `char *`) vers le répertoire d'accueil de l'utilisateur (chez moi cela retourne `"/home/zielonka"`),
- `char *basename(char *path)` — si `path` est un chemin alors `basename` retourne le dernier élément c'est-à-dire élément après le dernier `/`.

Par exemple si

```
char s[]="/home/dupont/bin/toto"
```

alors `basename(s)` retournera la chaîne `"toto"`.

- `char *dirname(char *path)` — si `path` est un chemin alors `dirname` retournera ce chemin sans le dernier élément (et sans `/` à la fin). Par exemple si

```
char s[]="/home/dupont/bin/toto"
```

alors `dirname(s)` retournera la chaîne `"/home/dupont/bin"`.

Donc en concaténant le résultat de `dirname`, le caractère `/` et le résultat de `basename` on retrouve le chemin initial.

- `char *getcwd()` retourne le chemin absolu vers répertoire courant (le répertoire de travail) d'un processus.

Question 1: On suppose que l'argument `reference` de la fonction

```
char *absolue(char *reference)
```

est un chemin absolu ou relatif correctement formé.

Si `reference` est un chemin absolu (commence par `/`) alors la fonction retourne juste une copie de `reference`.

Par contre si `reference` est un chemin relatif (ne commence pas par `/`) alors la fonction `absolue` retourne le chemin absolu correspondant. On suppose que ce chemin est juste la concaténation de répertoire de travail (les références relatives sont relatives vis à vis ce répertoire) et de la référence relative `reference` elle-même (les deux séparés par le caractère `/`). On demande d'écrire la fonction `absolue`.

Question 2: Écrire la fonction

```
char *creer_poubelle(void)
```

Cette fonction vérifie si le répertoire d'accueil de l'utilisateur contient déjà la poubelle. Si la poubelle n'existe pas alors la fonction va la créer (créer le répertoire `.poubelle` dans le répertoire d'accueil). La fonction retourne le chemin absolu vers le répertoire poubelle (sans `/` à la fin).

Question 3: Écrire la fonction

```
int move_fichier(char *reference)
```

qui prend comme argument une référence `reference` absolue ou relative vers un fichier régulier (on suppose que argument fourni est toujours valide, on ne demande pas de faire des vérifications de validité de la référence). La fonction fait la copie de ce fichier dans la poubelle dans le format qui a été décrit au début de cet exercice (c'est-à-dire en stockant dans la copie la longueur de chemin absolu et le chemin absolu vers le fichier copié). Et ensuite, une fois le fichier est dans la poubelle, `move_fichier` supprime le fichier donné par la référence (ou plus exactement elle supprime le lien dur `reference`). La fonction retourne 0 si elle ne rencontre pas de problèmes et `-1` sinon.

Question 4: Écrire la fonction

```
int move_tout(char *reference)
```

Cette fonction met dans la poubelle tous les fichiers réguliers se trouvant dans l'arborescence donnée par `reference` (dans le répertoire `reference`, dans tous les sous-répertoires etc.)

L'algorithme à implémenter est le suivant :

- si `reference` est un fichier régulier alors appliquer à ce fichier la fonction `move_fichier` de la question précédente,
- si `reference` est une référence vers un lien symbolique alors la fonction supprime seulement ce lien (on supprime les liens symboliques avec la même fonction qui sert à supprimer les liens durs),
- si `reference` est une référence vers un répertoire alors la fonction parcourt ce répertoire et pour chaque élément `entry` de ce répertoire (à l'exception de `.` et `..`) la fonction applique récursivement `move_tout` à fichier `"reference/entry"` A la fin de parcours de répertoire `reference` la fonction supprime le répertoire `reference`.

On suppose que tous les fichiers sont soit des fichiers réguliers, soit de répertoires soit de liens symboliques. Bien sûr c'est à vous de vérifier quel le type de fichier `reference` que la fonction reçoit.

Question 5: Écrire la fonction

```
int recuperer(char *nom)
```

qui restaure le fichier `nom`. On suppose que `nom` c'est un nom sans `/`. Si la poubelle contient le fichier `nom` alors la fonction restaure le chemin complet vers ce fichier et ensuite elle restaure le fichier lui même. Après la restauration le fichier `nom` sera supprimé de la poubelle. Par exemple pour le fichier `toto` décrit au début de l'exercice la fonction tout d'abord restaure si c'est nécessaire tous les répertoires sur le chemin `/home/dupond/bin` et ensuite restaure le fichier `toto` dans ce répertoire sous son format initial (sans préfixe ajouté au moment où on mettait `toto` à la poubelle). La fonction retourne 0 si elle s'exécute correctement et `-1` en cas de problèmes.

On supposera que vous avez à votre disposition une fonction

```
int refaire_chemin(char *path)
```

qui restaure tous les répertoires sur le chemin `path`.