Licence d'informatique
Module de Programmation des systèmes

Examen première session 2009

Philippe MARQUET

Juin 2009

Durée : 2 heures.
Documents de cours, TD, et TP autorisés.
Afin d'alléger les codes, limitez-vous aux tests d'erreurs indispensables.

1 Comptons les fichiers comme le shell : `lswc`

On désire écrire un programme `lswc` qui exécute la suite de commandes :

```
% ls -1 | wc -l > nfiles
```

comme le ferait un shell.

Dans cette suite de commandes :
- `ls -1` produit la liste des entrées du répertoire courant, à raison d'une entrée par ligne ;
- `wc -l` affiche le nombre de lignes lues sur son entrée standard.

Exercice 1 (Analyse du fonctionnement du shell)
Décrivez le fonctionnement du shell lors de l'exécution de la suite de commandes

```
% ls -1 | wc -l > nfiles
```

qui explique le résultat de la session suivante :

```
% ls -1 | wc -l
      19
% ls nfiles
ls: nfiles: No such file or directory
% ls -1 | wc -l > nfiles
% cat nfiles
      20
```

à savoir que la valeur 20 et non 19 est produite dans le fichier `nfiles` □

Exercice 2 (`lswc` : enchaînement des commandes)
Décrivez les grandes lignes d'une implémentation et donnez le code d'un programme `lswc` qui exécute la suite de commandes

```
% ls -1 | wc -l > nfiles
```

à la manière du shell. Dans un premier temps, on ne se souciera pas du statut retourné par `lswc`. (On s'interdira bien entendu d'utiliser un appel à la fonction `system()`.) □

Comme le fait le shell, nous désirons que notre implémentation de `lswc` propage aux processus fils les signaux issus de la saisie au clavier des caractères `<intr>` (`control-C`), et `<susp>` (`control-Z`).

Exercice 3 (Propagation des signaux)
Décrivez le principe d'une implémentation de la redirection des signaux aux processus fils et complétez le code de `lswc`. □

Exercice 4 (Reprise de l'exécution)
Une fois les processus fils suspendus, que proposez-vous pour que l'utilisateur puisse indiquer qu'il désire reprendre leur exécution. (Il n'est pas demandé d'implémenter cette proposition.) □

Lors de l'exécution d'une suite de commandes liées par des tubes, le statut retourné par le shell est celui de la dernière commande. Dans le cas ou une des commandes se termine anormalement (par exemple suite à la réception d'un signal), ou dans d'autres cas d'erreur, la suite de commandes retourne un statut d'erreur.

Exercice 5 (Terminaison)
Complétez votre implémentation de `lswc` pour retourner un statut à la manière de ce que fait le shell. □

2 Entrées multiples : `mulent`

L'objet de la commande `mulent` est de lister les entrées sous le répertoire courant qui désignent un nœud du système de fichiers référencés par plusieurs entrées. On qualifie de *multiples* de telles entrées.

Un exemple est fourni par la session suivante :

```
% mkdir d
% touch foo
% touch bar
% ln foo d/gee
% ln -s bar zuu
% ls -1aiR
.:
1222812 .
     23 ..
1222828 bar
1222826 d
1222827 foo
1222831 zuu

./d:
1222826 .
1222812 ..
1222827 gee

% mulent
1222827 2 foo
1222827 2 d/gee
```

La commande `touch` permet de créer un fichier. La commande `ln` permet de créer un lien physique, c'est-à-dire une nouvelle entrée pour un nœud préexistant du système de fichiers ; ce nœud correspondra donc à des entrées multiples. La commande `ln -s` permet de créer un lien symbolique vers une *autre* entrée du système de fichiers. Les options de `ls` utilisées permettent

- d'afficher une entrée par ligne (option `-1`) ;
- de lister toutes les entrées, y compris celle dont le nom débute par `.` (option `-a`) ;
- d'afficher le numéro d'inœud de chacune des entrées (option `-i`) ;
- de traiter le répertoire et, de manière récursive, ses sous répertoires (option `-R`).

La commande `mulent` affiche pour chacune des entrées le numéro d'inœud de l'entrée, le nombre de liens dans le système de fichiers vers cet inœud, le nom de l'entrée. Le nombre de liens correspond au champ `st_nlink` (de type `nlink_t` qui est un entier) de la structure `struct stat`.

Exercice 6 (Prise en compte des répertoires)
On remarque que la commande `mulent` n'affiche pas les entrées qui correspondent à des répertoires et qui pourtant, au sens strict, sont des entrées multiples. Expliquez pourquoi les répertoires sont au sens strict des entrées multiples. □

La spécification de `mulent` indique que lors du parcours d'une arborescence,
- un sous-répertoire n'est considéré (et donc parcouru) que s'il appartient à la même partition (au même périphérique) que le répertoire courant ;
- un lien symbolique n'est suivi que s'il référence un fichier de la même partition que le répertoire courant.

Une partition (ou disque logique, périphérique, *device*) est représentée par une valeur entière de type `dev_t` telle celle utilisée dans les structures `struct stat`.

En ce qui concerne notre implémentation, nous supposerons que la partition qui correspond au répertoire courant est identifiée par la valeur de la variable

```
static dev_t current_dev;
```

Exercice 7 (Partition unique)
Expliquez ce qui justifie de ne pas explorer les fichiers en dehors de la partition courante. □

Exercice 8 (Une entrée multiple)
Proposez une fonction

```
int mulent_file(const char *pathanme, int *nlink, int *inumber);
```

qui indique par une valeur de retour booléenne si l'entrée `pathanme` est multiple. Dans ce cas, la valeur qui est à l'adresse `nlink` est renseignée par le nombre de liens associés à l'entrée, et la valeur qui est à l'adresse `inumber` est renseignée par le numéro d'inœud correspondant à cette entrée. □

Exercice 9 (À la recherche des entrées multiples)
Proposez une fonction

```
void mulent(const char *pathname);
```

qui
- dans le cas d'un répertoire devant être traité, itère sur chacune de ses entrées par des appels à `mulent()` et termine ;
- dans le cas d'un lien symbolique devant être traité, réalise un appel à `mulent()` sur le fichier référencé par le lien ;
- vérifie si l'entrée est multiple et affiche les informations nécessaires si c'est la cas.

Un appel à `mulent(".")` forme, en sus d'une initialisation de `current_dev`, le gros de la fonction pincipale la commande `mulent`. □

3 Rémanence de `pthread_join`

La spécification POSIX de la fonction `pthread_join()` d'attente de terminaison d'un thread précise que la valeur ne peut être récupérée qu'une unique fois par un unique thread. Nous proposons ici de rendre cette valeur rémanente : elle pourra être accédée de multiples fois, par autant de threads que désiré.

Le principe est de mémoriser la valeur de type `void *` retournée par le thread au sein d'une structure et de faire des accès à cette structure pour accéder à cette valeur. De plus, un accès à cette structure alors que le thread n'a pas encore terminé devra être bloquant en attendant la terminaison.

Un type sera défini pour une telle structure :

```
struct join_s ;
```

Une telle structure sera mémorisée pour chacun des threads dont on désire garder la valeur de retour rémanente. Deux fonctions d'allocation et de recherche de telles structures sont fournies (vous n'avez pas à les écrire). La fonction

```
struct join_s *new_join(pthread_t tid);
```

retourne un pointeur sur une nouvelle structure ; elle sera associée à la terminaison du thread `tid`.

La fonction

```
struct join_s *get_join_by_tid(pthread_t tid);
```

retourne pour sa part une référence sur la structure `struct join_s` associée au thread `tid`.

Notre objectif est de fournir trois fonctions équivalentes aux fonctions POSIX `pthread_create()`, `pthread_exit()`, et `pthread_join()`. L'utilisation de nos fonctions de remplacement permet des accès rémanents à la valeur retournée par un thread.

- La fonction

  ```
  int jn_create(
          pthread_t *tid,
          const pthread_attr_t *attr,
          void *(*threadfunction)(void*),
          void *arg);
  ```

  sera appelée pour la création d'un thread. En plus de réaliser la création du thread par `pthread_create()`, elle se devra d'initialiser une structure `struct join_s` pour le thread.
- La fonction

  ```
  void jn_exit(void *value_ptr);
  ```

  sera utilisée pour terminer un thread. Avant d'appeler `pthread_exit()`, cette fonction devra
  - mettre à jour la structure `struct join_s` qui lui est associée ;
  - débloquer d'éventuels threads en attente de sa terminaison.
- La fonction

  ```
  int jn_join(
          pthread_t tid,
          void **value_ptr);
  ```

  sera utilisée en remplacement de `pthread_join()` par un thread voulant accéder à la valeur retournée par le thread `tid`. Cette fonction devra
  - si le thread a terminé, retourner la valeur mémorisée dans la structure `struct join_s` ;
  - sinon, si le thread n'a pas terminé, se mettre en attente de cette terminaison.
- Lors de la terminaison par `jn_exit()`, il est donc nécessaire de réveiller les éventuels threads en attente de cette terminaison.

Une structure `struct join_s` se compose d'un indicateur de terminaison du thread, de la valeur de type `void *` retournée par le thread, d'un mécanisme permettant aux threads en attente de la terminaison du thread de se bloquer. Ces différentes valeurs peuvent être potentiellement accédées simultanément par plusieurs threads. On ajoutera donc un mécanisme assurant l'exclusion mutuel des accès.

Exercice 10 (Structure `struct join_s`)
Donnez une définition de la structure `struct join_s`. □

Exercice 11 (Bibliothèque `join`)
Donnez une implémentation des trois fonctions `jn_create()`, `jn_exit()` et `jn_join()`. □

Fin du sujet