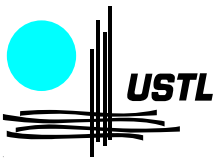

Licence d'informatique
Module de Programmation des systèmes

Examen seconde session 2009

Philippe MARQUET

Septembre 2009

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

1 Compter les include comme le shell : `grpincl`

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

```
% grep #include < main.c | wc -l
```

comme le ferait un shell.

Dans cette suite de commandes :

- la commande `grep` affiche les lignes lues sur son entrée standard qui contiennent la chaîne de caractères `"#include"` ;
- `wc -l` affiche le nombre de lignes lues sur son entrée standard.

L'exécution par le shell repose sur deux processus fils, un pour chacune de ces commandes.

Exercice 1 (`grpincl` : enchaînement des commandes)
Décrivez les grandes lignes d'une implémentation et donnez le code d'un programme `grpincl` qui exécute la suite de commandes

```
% grep #include < main.c | wc -l
```

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

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

Exercice 2 (Propagation des signaux)
Décrivez le principe d'une implémentation de la redirection des signaux aux processus fils et complétez le code de `grpincl`. □

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

2 Liens symboliques courts : `shortlnk`

L'objet de la commande `shortlnk` est de lister les entrées sous le répertoire courant qui sont des *liens symboliques courts* (*short symbolic links*). Un lien symbolique court est un lien symbolique qui référence le nom d'une entrée qui est assez court pour pouvoir être mémorisé dans l'inœud lui-même. Un lien symbolique court n'utilise donc aucun bloc du système de fichiers et la valeur du champ `st_blocks` de la structure `struct stat` est donc nulle.

La spécification de `shortlnk` indique que lors du parcours d'une arborescence, les liens symboliques sont suivis.

Exercice 4 (Implémentation des liens symboliques courts)
Expliquez, en dehors du fait de gagner un bloc, les avantages de l'implémentation choisie des liens symboliques courts sur une implémentation traditionnelle qui utiliserait un bloc pour mémoriser le nom de l'entrée référencée. □

Exercice 5 (Un lien symbolique court)
Proposez une fonction

```
int shortlnk_file(const char *pathanme, const struct stat *st);
```

qui indique si le fichier désigné par les deux paramètres `pathname` (son nom), et `st` (la valeur de type `struct stat` correspondante), est un lien symbolique court, c'est-à-dire :

- est un lien symbolique ;
- l'inœud correspondant à ce lien symbolique n'utilise pas de blocs du système de fichier.

La fonction `shortlnk_file()` retourne un booléen : une valeur `TRUE` si l'entrée est un lien symbolique court, une valeur `FALSE` sinon. □

Pour compléter le développement de la commande `shortlnk`, il reste maintenant à définir une fonction

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

qui

- si l'entrée de nom `pathname` est un lien symbolique court, affiche le nom cette entrée ;
- dans le cas d'un répertoire, vérifie pour chacune de ses entrées, et ce récursivement, s'il s'agit d'un lien symbolique court.

Dans ses grandes lignes, la fonction `main()` de la commande `shortlnk` consisterait en un simple appel à `shortlnk(".")`.

Exercice 6 (Les liens symboliques courts)
Donnez le code de cette fonction `shortlnk()`. □

3 `pthread_join` multiples

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 permettre à de multiples threads d'accéder à cette valeur.

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. Bien entendu, 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 partager la valeur de retour. Trois fonctions d'allocation, de destruction, 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

```
int del_join(struct join_s *j);
```

supprime la structure identifiée par le paramètre.

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 des fonctions équivalentes aux fonctions POSIX `pthread_create()`, `pthread_exit()`, et `pthread_join()`. Deux versions de cette dernière fonction seront fournies : une version ne réalisant que le seul accès à la valeur retournée, et une version réalisant un accès à la valeur retournée et libérant l'ensemble des ressources associées. L'utilisation de nos fonctions de remplacement permet des accès multiples 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

  ```
  int 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.
- La fonction

  ```
  int jn_join_and_clean(
          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`. En plus d'assurer la même fonctionnalité que `jn_join()`, elle supprime les ressources associées à la valeur retournée par le thread, à savoir la structure `struct join_s`.

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

Exercice 8 (Bibliothèque `join`)
Donnez une implémentation des quatre fonctions `jn_create()`, `jn_exit()` `jn_join()`, et `jn_join_and_clean()`. □

Fin du sujet