Programmation des systèmes

Devoir surveillé final

19 décembre 2017

Durée : 2h30
Documents (notes de cours, mini-man, TD et TP) autorisés

Vous expliquerez votre code de sorte que votre intention (et pas seulement votre implémentation) soit claire. La gestion des erreurs pourra rester très simple, par exemple en interrompant le programme si nécessaire.

Lorsque vos réponses sont de simples modifications de fonctions précédentes, vous vous contenterez d'indiquer les modifications (en particulier en précisant clairement où ces modifications sont placées).

Vos fonctions utiliseront exclusivement les entrées/sorties de bas niveau (c'est-à-dire `open`, `close`, `read`, `write`, etc.).

Afin de paralléliser la correction, vous rendrez 2 copies, 1 copie par section.

1 Première copie

Vous composerez les deux exercices suivants sur la première copie.

1.1 Petites questions

Voici un extrait de session de test de `./mshell -v` :

```
mshell> xeyes
Added job [1] 9678 xeyes
^Zsigtstp_handler : entering
sigtstp_handler : exiting
sigchld_handler : entering
Stopped job [1] 9678 xeyes
sigchld_handler : exiting
mshell>
```

Q 1. Expliquez pas à pas ce qui se passe dans cette session. En particulier, indiquez dans l'ordre qui envoie un signal à qui.

On a vu en cours un exemple de producteur - consommateur dans un programme multithreadé utilisant mutex et condition. En particulier, le producteur contenait les lignes suivantes :

```
while (nb_occupes == TAILLE_FILE)
    assert(pthread_cond_wait(&cond_producteur, &mutex) == 0);
```

Q 2. Expliquez pourquoi `pthread_cond_wait` est appelé dans une boucle `while` et pas seulement dans un `if`.

1.2 Pseudo-`tree`

L'objectif de cet exercice est de créer une commande ressemblant à `tree`.

```
$ tree .
.
├── fic0
├── rep0
│   ├── fic1
│   └── rep1
│       └── fic2
└── rep2
    └── lnk0 -> ../rep0

4 directories, 3 files
```

Cette commande permet de lister tout le contenu d'une arborescence en « dessinant » un lien entre le nom d'un répertoire et le nom des entrées qu'il contient. Elle compte également le nombre de répertoires et de fichiers rencontrés ; les liens symboliques qui pointent vers un répertoire sont comptabilisés comme répertoires, sinon ils sont comptabilisés comme fichiers.

Pour simplifier ici on utilisera juste l'indentation pour indiquer l'imbrication. On voudra ainsi obtenir par exemple :

```
$ ~/bin/pseudo-tree .
.
  fic0
  rep0
    fic1
    rep1
      fic2
  rep2
    lnk0 -> ../rep0

5 répertoire(s), 3 fichier(s)
```

Notez que l'on a choisi dans cet exemple de comptabiliser le répertoire initial `.`, contrairement à ce que fait `tree` : c'est pour cela qu'il y a une différence de 1 répertoire dans le décompte. Vous pourrez choisir d'implémenter l'une ou l'autre version.

Dans un premier temps, on s'intéresse juste à l'affichage de l'arborescence, c'est-à-dire sans compter les répertoires et fichiers trouvés. Par ailleurs, on traitera les liens symboliques comme des fichiers ordinaires, c'est-à-dire sans indiquer la cible des liens rencontrés.

Q 3. Définissez une fonction `pseudotree` qui prendra *au moins* en argument :

- un chemin (`"."` dans le premier appel de l'exemple ci-dessus),
- un niveau d'imbrication (0 dans le premier appel de l'exemple ci-dessus),

et qui affiche l'arborescence commençant au chemin donné.

Vous pourrez ajouter les arguments supplémentaires qui vous sembleront utiles.

L'algorithme pourrait être plus ou moins le suivant :

- afficher le nom de l'entrée courante,
- si cette entrée est un répertoire, traiter récursivement toutes les entrées de ce répertoire.

Q 4. Modifiez votre fonction `pseudotree` pour afficher la cible de tous les liens symboliques rencontrés.

Vous indiquerez clairement où vous insérez ou modifiez du code de la fonction `pseudotree`, par exemple en plaçant des marques telles que « (1) », « (2) », etc. dans le code de votre réponse à la question précédente.

Q 5. Ajoutez le décompte des fichiers et répertoires rencontrés. Pour les liens symboliques, votre code devra les comptabiliser comme un répertoire si la cible est un répertoire, et comme un fichier sinon.

Vous indiquerez explicitement si vous comptez le répertoire de départ ou pas, c'est-à-dire si votre code compte 4 ou 5 répertoires dans l'exemple.

Q 6. *Bonus* Expliquez brièvement (en quelques phrases) ce qu'il faudrait modifier dans le code pour pouvoir dessiner les liens (├──, └──, etc.)[1].

1. Note culturelle : pour obtenir vraiment le même résultat que `tree`, on utilise en général la fonction `scandir`(3) qui permet au passage de trier les entrées par ordre alphabétique.

2 Deuxième copie

Vous composerez les deux exercices suivants sur la deuxième copie.

2.1 Répéteur

On souhaite écrire une commande `repeteur` qui prend en arguments :

— une commande, que l'on appellera commande principale,
— suivie de n autres commandes, que l'on appellera sous-commandes,

et qui exécute ces commandes de sorte que toutes les sous-commandes reçoivent sur leur entrée standard ce que la commande principale produit sur sa sortie standard. Pour cela `repeteur` devra, comme son nom l'indique, répéter à chaque sous-commande ce que la commande principale aura produit sur sa sortie standard.

Par exemple,

```
$ repeteur ls "wc -c" "wc -w" "wc -l"
```

a pour résultat de passer le résultat de la commande principale `ls` aux trois sous-commandes `wc -c`, `wc -l`, `wc -w`. Le résultat sera l'affichage du résultat des sous-commandes, sans qu'on puisse définir à l'avance l'ordre d'affichage. On pourra par exemple obtenir :

```
407
35
36
```

où 407 est le nombre de caractères (affiché par `wc -c`), 35 le nombre de lignes (affiché par `wc -l`) et 36 le nombres de mots (affiché par `wc -w`).

L'algorithme peut être décrit à gros traits ainsi :

— je crée un tube pour récupérer ce que la commande principale va produire sur sa sortie standard et je lance la commande principale en redirigeant sa sortie standard dans ce tube,
— je crée un tube par sous-commande (soit n tubes) pour leur transmettre ce que la commande principale a produit et je lance chaque sous-commande en redirigeant son entrée standard depuis le tube créé pour elle,
— tant que la commande principale produit des choses sur sa sortie standard, je la répète à toutes les sous-commandes,
— quand la commande principale a terminé de produire sa sortie standard, j'attends que toutes les commandes se terminent.

Q 7. Expliquer ce qui se passerait s'il n'y avait qu'un seul tube, connectant directement la sortie standard de la commande principale aux entrées standards de toutes les sous-commandes. En particulier, expliquer en quoi le comportement ne serait pas celui espéré.

Vous supposerez que les trois variables `imain`, `istart` et `iend` ont été définies telles que :

— la commande principale est `argv[imain]`,
— les sous-commandes sont `argv[istart]` jusqu'à `argv[iend]`,

où `argv` est le tableau des arguments de la ligne de commande de `repeteur`.

Vous utiliserez la fonction `makeargv` vue en TP pour écrire la solution.

```
char **makeargv(const char *s);
```

Étant donnée une chaîne (comme `"wc -c"`), cette fonction retourne un tableau de chaînes respectant la spécification d'un `argv`.

On accordera presque tous les points à une solution correcte supposant que ni la commande principale ni les sous-commandes n'ont d'arguments, donc sans `makeargv`.

Q 8. Donner le code qui permet de lancer la commande principale (et les déclarations de variables nécessaires). On donnera bien sûr aussi le code de gestion du tube.

Q 9. Donner le code qui permet de lancer les n sous-commandes (et les déclarations de variables nécessaires). On donnera bien sûr aussi le code de gestion des tubes.

Q 10. Donner le code manquant pour terminer correctement l'écriture de ce programme.

Q 11. Que se passe-t-il si une des sous-commandes échoue, ou plus généralement si elle se termine avant d'avoir lu toute son entrée standard ?

Pour remédier à ce problème, on décide de traiter le signal `SIGPIPE`. On suppose la fonction suivante pour ce traitant :

```
void sigpipe_handler (int sig) {
    /* Fonction qui ne fait rien ! */
}
```

Q 12. Expliquer en quoi cela suffit à remédier au problème évoqué plus haut.

Q 13. Donner le code nécessaire pour mettre en place ce traitant de signal.

On ajoute à la fonction `sigpipe_handler` un affichage indiquant le passage dans la fonction. Voici le résultat obtenu :

```
Passage dans sigpipe_handler
Passage dans sigpipe_handler
Passage dans sigpipe_handler
Passage dans sigpipe_handler
Passage dans sigpipe_handler
...
Passage dans sigpipe_handler
Passage dans sigpipe_handler
Passage dans sigpipe_handler
Passage dans sigpipe_handler
Passage dans sigpipe_handler
```

Q 14. Quand sont déclenchés les signaux SIGPIPE ?

Q 15. Proposer une solution simple (en quelques phrases, il n'est pas nécessaire de donner du code) permettant de gérer proprement les erreurs sur les tubes et de n'avoir qu'un passage dans `sigpipe_handler` par commande qui échoue.

2.2 Producteurs – consommateur

On considère dans cet exercice un programme multi-thread composé de `N` threads producteurs et un thread consommateur.

Chaque thread producteur produit un nombre entier et le dépose dans une structure de données avec la fonction `depose` :

```
void depose(struct mbox *m, int d, int index);
```

La structure `mbox` est définie ainsi :

```
struct mbox {
     int data[N];
     /* structures de synchronisation */
};
```

Le paramètre `d` représente le nombre à déposer. Le paramètre `index` représente l'index du thread (compris entre `0` et `N-1`). Le thread d'index `i` écrit son nombre `d` dans la case `data[i]`.

Le consommateur appelle la fonction `moyenne` :

```
double moyenne(struct mbox *m);
```

La fonction `moyenne` attend que tous les threads producteurs aient déposé leur nombre ; puis elle calcule la moyenne des `N` nombres et elle la retourne comme résultat.

Q 16. Ajouter à la structure `mbox` les variables nécessaires pour réaliser la synchronisation entre producteurs et consommateurs. En particulier, la fonction `moyenne` est bloquante si les producteurs n'ont pas encore tous déposé leur nombre.

Indice : vous pouvez utiliser soit des sémaphores, soit des `pthread_cond_t` avec des `pthread_mutex_t` (mais ne mélangez pas les deux).

Q 17. Écrire le code de la fonction `mbox_init` pour initialiser les valeurs de la structure.

Q 18. Écrire le code des fonctions `depose` et `moyenne`.