Unix : concepts et outils avancés
TD no 4 : système de fichiers

Olivier Aumage
LIP, bureau 302
email : Olivier.Aumage@ens-lyon.fr

18 octobre 2001

1 Entrées/sorties

- Écrivez une version en **C** de la commande `cat` en utilisant uniquement les fonctions standards d'entrées/sorties de la bibliothèque du **C** `fread` et `fwrite`, et en lisant et écrivant des blocs d'un caractère.
- Écrivez une seconde version de `cat` en utilisant uniquement les appels système d'entrées/sorties `read` et `write`.
- En utilisant la commande interne `time` du shell `bash`, comparez les performances des deux versions pour une lecture/écriture d'un fichier raisonnablement volumineux (i.e. de plus de 1 Mo). Exemple :

```
$ time ./my_cat < /lib/libc.so.6 > tst

real    0m8.658s
user    0m2.920s
sys     0m5.740s
$
```

- Interprétez les résultats obtenus. Quelles sont les commandes de plus bas niveau : `read/write` ou `fread/fwrite` ?
- Essayez le programme suivant :

```
#include <stdio.h>
#include <unistd.h>

int
main()
{
  fwrite("Hello ", sizeof(char), 6, stdout);
  write(STDOUT_FILENO, "World !", 7);

  return 0;
}
```

L'observation est-elle en accord avec votre interprétation des résultats précédents ?

2 Tampons

Le *"fichier"* `stdout` utilise par défaut une zone de mémoire tampon pour grouper les requêtes d'écriture et améliorer les performances. Les commandes `setbuf` et `setvbuf` permettent de changer le tampon (pour en mettre un de plus grande taille par exemple) ou de sélectionner un mode d'utilisation différent. En particulier, `setbuf(stdout, NULL)` supprime le *tamponnage* de la sortie standard.

- Utilisez `setbuf` pour refaire un comparatif *équitable* des paires `read/write` et `fread/fwrite;`
- Faites quelques tests avec des accès par blocs de plus d'un caractère.

3 Déplacement dans un fichier

La fonction `lseek` permet d'obtenir ou de changer la position du *pointeur courant* d'un descripteur de fichier. Ce pointeur courant indique le point d'application de la prochaine opération d'entrée ou de sortie sur le descripteur concerné.

- Écrivez une version du programme `tail` qui affiche les `n` dernières lignes d'un fichier. Le programme devra accepter 0, 1 ou 2 paramètres. Les appels suivants doivent tous être valides :
  ```
  $ my_tail -10 fichier.txt
  $ my_tail fichier2.txt
  $ my_tail < fichier3.txt
  $ my_tail -15 < fichier4.txt
  ```
  Le nombre de lignes à afficher par défaut est 10. Vous aurez besoin d'utiliser la forme *étendue* de la fonction `main` :
  ```
  int main(int argc, char *argv[])
  ```
  Le paramètre `argc` indique le nombre d'arguments passés en ligne de commande. Le tableau `argv` contient les chaînes de caractères correspondant à ces arguments. L'argument `0` existe toujours et correspond à la partie *commande* de la ligne de commande (i.e. le chemin et le nom du fichier spécifié pour désigner la commande). Il en résulte que `argc` vaut toujours au-moins `1`.
  En outre, vous aurez probablement besoin de la fonction `atoi` pour convertir une chaîne en nombre entier.
- Quel est le résultat de `ls /usr/bin|my_tail` ? Correspond-il au résultat attendu ? Dans la négative, proposez une hypothèse sur la cause de l'échec (et testez la commande `tail` originale).

4 Duplication de descripteurs

La fonction `dup` duplique (comme son nom l'indique) un descripteur de fichier. Le descripteur créé contrôle le même fichier que le descripteur original. Les points communs entre les deux descripteurs sont répertoriés dans le `man`. Le numéro du descripteur créé est le plus petit disponible.

- Utilisez la commande `dup` dans votre version du programme `tail` pour utiliser uniquement le descripteur `STDIN_FILENO` dans la boucle de lecture indépendament de la spécification (ou non) d'un fichier en entrée.

5 Exercices supplémentaires

- Écrivez un programme `gen` qui génère une chaîne de caractères (le choix du caractère est libre) de longueur spécifiée par l'utilisateur sur la sortie standard. Réalisez les tests suivants (sur une machine Solaris de l'école !) :
  ```
  $ time gen 10000000 > ./tst
  $ time gen 10000000 > /tmp/tst
  $ time gen 10000000 > /var/tmp/tst
  $
  ```
  N'oubliez pas d'effacer ces fichiers après les tests et proposez une hypothèse expliquant la hiérarchie observée.

6 Pour la route ...

Le répertoire `/home/oaumage/Unix` contient un programme `surprise` et un programme `filtre` compilés pour Solaris/x86. Les fichiers *source* et la version intermédiaire en assembleur de `surprise` sont également disponibles. **Note :** ce programme est volontairement écrit dans un très mauvais style et regroupe une quantité non négligeable de choses à ne pas faire ! ! !

Essayez les commandes suivantes dans l'ordre :

- `./surprise`
- `./surprise|./surprise`
- `cat /dev/null|./surprise|./surprise`
- `cat /dev/null|./surprise|./surprise|./filtre`
- `cat /dev/null|./surprise|./surprise >/tmp/tst` puis `cat /tmp/tst`
- `cat /dev/null|./surprise|./surprise >/tmp/tst` puis `cat /tmp/tst|./filtre`

Questions :

- Comment fonctionne le programme ? Quelques pistes de lecture :
  - la fonction `f` est-elle appelée ? Si oui, comment ?
  - quel est le nombre d'itérations de la boucle sur `j` ?
  - quel est le rôle des variables `a`, `b` et `c` ?
  - quel est le rôle de chacun des éléments de la ligne de commande ?
- Un tel programme est-il portable ? Si non, quelles sont les suppositions faites ?
- Que pensez-vous du niveau de sécurité d'un tel programme ?
- Qu'en déduisez-vous sur l'usage des fonctions d'entrées ?
- Pouvez-vous proposer un bilan des choses à ne pas faire, et néanmoins présentes dans ce programme ?
- Proposez une version simple et *propre* de ce programme telle que `./surprise` affiche directement le résultat escompté.

— ❃ —