UNIVERSITÉ PARIS SUD
IUT D'ORSAY

DUT Informatique
M3101 Système S3
2015 / 2016

Travaux Dirigés n° 2 : Fichiers et tubes

Objectifs : maîtriser les primitives de manipulation de fichiers Unix. Savoir utiliser les tubes (pipes) Unix pour la communication entre les processus.

1 Ouverture et fermeture de fichiers

Les primitives système

```
#include <fcntl.h>
int open(const char *chemin, int drapeaux);
int open(const char *chemin, int drapeaux, mode_t mode);
int creat(const char *chemin, mode_t mode);
```

ouvrent le fichier se trouvant dans le système de fichiers sous le nom `chemin` (qui peut être absolu ou relatif) et renvoient le nouveau descripteur de fichier ou -1 en cas d'échec.

Le paramètre `drapeaux` indique le mode d'accès au fichier. Il doit inclure une de trois valeurs :

— `O_RDONLY` — ouverture du fichier en lecture seule,
— `O_WRONLY` — ouverture du fichier en écriture seule,
— `O_RDWR` — ouverture du fichier en lecture/écriture.

De plus, des attributs additionnels peuvent être ajoutés aux `drapeaux` avec un « ou » binaire. Par exemple, pour ouvrir un fichier `./out.txt` en écriture en le créant s'il n'existe pas, on peut exécuter `open("./out.txt",O_WRONLY|O_CREAT)`. Si on ne souhaite pas garder l'ancien contenu du fichier, on peut le remettre à vide à l'ouverture : `open("./out.txt",O_WRONLY|O_CREAT|O_TRUNC)`. Si, au contraire, on veut préserver le contenu actuel et ajouter des nouvelles données à la fin du fichier, on peut écrire `open("./out.txt",O_WRONLY|O_CREAT|O_APPEND)`. Enfin, si on veut éviter de toucher un fichier existant, on écrit `open("./out.txt",O_WRONLY|O_CREAT|O_EXCL)` pour faire échouer `open()` dans le cas où `./out.txt` existe. Voir la page `man 2 open` pour la liste complète des attributs acceptés par `open()`.

Le paramètre `mode` indique les droits à utiliser si un nouveau fichier est créé. Par exemple, la valeur `0644` (le zéro initial indique une constante octale) attribue au nouveau fichier les droits de lecture et écriture pour le propriétaire et de lecture seule pour les autres utilisateurs. Le propriétaire et groupe du nouveau fichier sont déterminés pas le UID et GID effectifs du processus appelant. **Nota** : Les droits dans le paramètre `mode` sont combinés avec la masque de création de fichiers du processus appelant. Voir la page `man 2 umask` pour les informations détaillées.

Un appel de `creat(fic,mode)` est équivalent à `open(fic,O_WRONLY|O_CREAT|O_TRUNC,mode)`.

L'ensemble de fichiers ouverts, accessibles par les descripteurs correspondants, est hérité par le processus fils après un appel de `fork()`.

La primitive système

```
#include <unistd.h>
int close(int fdesc);
```

ferme et libère le descripteur `fdesc` dans le processus appelant. Toute opération ultérieure sur `fdesc` échouera, avant que ce numéro ne soit pris pour un nouveau descripteur par `open()`, `creat()` ou autre primitive.

2 Lecture et écriture dans un fichier

La primitive système

```
#include <unistd.h>
int read(int fdesc, void *tampon, size_t longueur);
```

lit `longueur` octets (au maximum) dans le fichier ayant `fdesc` pour descripteur et place ces octets en mémoire à partir de l'adresse `tampon`. Le fichier doit être ouvert en mode permettant la lecture, c'est-à-dire `O_RDONLY` ou `O_RDWR`. La primitive renvoie le nombre d'octets lus (qui ne peut pas depasser `longueur`), ou 0 si le fichier est vide, ou enfin -1 en cas d'erreur.

La primitive système

```
#include <unistd.h>
int write(int fdesc, const void *tampon, size_t longueur);
```

écrit dans le fichier ayant `fdesc` pour descripteur les `longueur` octets commençant en mémoire à l'adresse `tampon`. Le fichier doit être ouvert en mode permettant l'écriture, c'est-à-dire `O_WRONLY` ou `O_RDWR`. La primitive renvoie le nombre d'octets écrits ou -1 en cas d'échec.

Par exemple, si le descripeur d'un fichier ouvert est `fd` et qu'on veut y lire un nombre flottant et le placer dans la variable `v`, on utilise l'instruction `read(fd,&v,sizeof(float))`. Si on veut écrire la valeur de la variable `v` dans `fd`, on écrit `write(fd,&v,sizeof(float))`.

Les primitives `read()` et `write()` avancent la position courante (dans la description de fichier ouvert référencée par le descripteur) du nombre d'octets lus ou écrits. La primitive système

```
#include <unistd.h>
off_t lseek(int fdesc, off_t deplacement, size_t origine);
```

permet de modifier la position directement. Le paramètre `deplacement` indique le nombre d'octets à sauter, et peut être positif, négatif ou nul. Le paramètre `origine` indique le point de départ pour le déplacement et peut avoir une des trois valeurs :

— `SEEK_SET` — le déplacement est fait par rapport au début du fichier,
— `SEEK_CUR` — le déplacement est fait par rapport à la position actuelle,
— `SEEK_END` — le déplacement est fait par rapport à la fin du fichier.

La primitive renvoie la nouvelle valeur de position ou -1 en cas d'erreur.

Par exemple, si on veut que la prochaine opération de lecture dans `fd` renvoie des données à partir du cinquième octet, on écrit `lseek(fd,4,SEEK_SET)`, car le premier octet se trouve à la position 0. Si on veut que la prochaine écriture écrase les cinq derniers octets dans `fd`, on écrit `lseek(fd,-5,SEEK_END)`.

Les fichiers spéciaux, tels que tubes, sockets ou terminaux, représentent des canaux de communication plutôt que des blocs de données. La notion de position n'a pas de sens pour ces fichiers, et la primitive `lseek()` renverra -1.

3 Tubes

Les tubes (*pipes* en anglais) sont des canaux de communication unidirectionnels manipulés à l'aide de descripteurs de fichiers. Tout processus ayant accès à ces descripteurs peut lire ou écrire dans un tube : seuls les processus descendant du créateur d'un tube, et le créateur lui-même pourront l'utiliser [1].

1. Dans les Unix modernes il est possible de passer des descripteurs de fichier entre les processus à l'aide de sockets.

Un tube se comporte comme une file *fifo* (*first in, first out*) : les premières données entrées dans le tube seront les premières à être lues. Toute lecture est destructive : si un processus lit une donnée dans le tube, celle-ci n'y sera plus présente, même pour les autres processus.

La primitive système

```
#include <unistd.h>
int pipe(int tube[2]);
```

crée un tube et renvoie 0 en cas de succès et -1 en cas d'échec. Après appel :

— `tube[0]` est le descripteur du tube créé *en lecture*,

— `tube[1]` est le descripteur du tube créé *en écriture*.

Les appels `close(tube[0])` et `close(tube[1])` ferment le tube en lecture ou en écriture, respectivement, pour le processus appelant. Nous allons appeler un processus « lecteur », s'il détient le descripteur de lecture `tube[0]` non-fermé. Nous allons appeler un processus « écrivain », s'il détient le descripteur d'écriture `tube[1]` non-fermé.

On utilise la primitive `write(tube[1],tampon,longueur)` pour écrire dans le tube. Sous Linux, la capacité d'un tube (c'est-à-dire le nombre d'octets qui peuvent être écrits dans le tube pendant que aucun lecteur ne le lit de son côté) est 65536 octets. Si une opération d'écriture surpasse la capacité du tube, la primitive `write()` *se bloque* en attente qu'un lecteur décharge le tube. La primitive renvoie le nombre d'octets écrits ou -1 en cas d'échec. S'il n'y a plus de lecteurs sur le tube — c'est-à-dire qu'aucun processus ne possède le descripteur `tube[0]` — le processus qui appelle `write()` est tué par le signal `SIGPIPE`. Si ce signal est ignoré, bloqué ou capté, `write()` renvoie -1.

La primitive `read(tube[0],tampon,longueur)` est utilisée pour lire dans le tube. Elle renvoie le nombre d'octets lus, ou 0 si le tube est vide *et* qu'il n'y a plus d'écrivains sur le tube (ou -1 en cas d'erreur). S'il reste un écrivain et le tube est vide, le processus qui appelle `read()` *se bloque* en attente de données.

4 Exercices

Exercice 1 : Connais-toi toi-même. Considérons le programme `exo1.c` (les entêtes sont omises) :

```
int main(int argc, char ** argv) {
  int fd;
  if (argc <= 1) { write(2, "missing argument\n", 17); return 1; }
  fd = open(argv[1], O_WRONLY);
  if (fd == -1) { perror("open() error"); return 1; }
  write(fd, argv[1], strlen(argv[1]));
  close(fd);
  return 0;
}
```

Supposons que l'utilisateur actuel est `bob` et que la commande `ls -l` produit l'affichage suivant :

```
-rw-r--r-- 1 bob users 1457 Sep 12 22:40 exo1.c
-r-xr-xr-x 1 bob users 7360 Sep 12 22:41 exo1
```

Quel sera le résultat de la commande `./exo1 exo1.c` ? Détaillez les affichages, les modifications des fichiers existants et les nouveaux fichiers créés avec leur contenu.

Quel sera le résultat de la commande `./exo1 exo1` ?

Quel sera le résultat de la commande `./exo1 exo1.h` ?

Quel sera le résultat de la commande `./exo1 exo1.h`, si on remplace `open(argv[1], O_WRONLY)` par `open(argv[1], O_WRONLY | O_CREAT, 0400)` ?

Quel sera le résultat de la commande `./exo1 exo1.c`, si on remplace `open(argv[1], O_WRONLY)` par `open(argv[1], O_WRONLY | O_CREAT | O_EXCL, 0600)` ?

Exercice 2 : On avance. Considérons le programme `exo2.c` :

```
int main (int argc, char ** argv) {
  int fd, nbLus;
  char c;
  fd = (argc > 1) ? open(argv[1], O_RDONLY) : 0;
  if (fd == -1) { perror("open() error"); return 1; }
  while (1) {
    nbLus = read(fd, &c, 1);
    if (nbLus == 0) break;
    if (c >= 'a' && c < 'z') c++;
    write (1, &c, 1);
  }
  return 0;
}
```

Quel sera le résultat de la commande `./exo2 exo2.c` ?

Quel sera le résultat de la commande `./exo2` ?

Exercice 3 : Miroir, miroir joli. Le programme suivant doit inverser le contenu du fichier reçu en paramètre sur la ligne de commande :

```
int main(int argc, char ** argv) {
  int fd;
  off_t pos, size;
  char carDeb, carFin;
  if (argc <= 1) { write(2, "missing argument\n", 17); return 1; }
  fd = open(argv[1], O_RDWR);
  if (fd == -1) { perror("open() error"); return 1; }

  size = lseek(fd,                        );  // déterminer la taille du fichier

  for (pos = 0; pos < size / 2; pos++) {

    lseek(fd,                             );  // se déplacer vers le début

    read(fd, &carDeb, 1);                        // lire un caractère au début

    lseek(fd,                             );  // se déplacer vers la fin

    read(fd, &carFin, 1);                        // lire un caractère à la fin

    lseek(fd,                             );  // reculer d'un octet en arrière

    write(fd, &carDeb, 1);                       // écrire un caractère à la fin

    lseek(fd,                             );  // se déplacer vers le début

    write(fd, &carFin, 1);                       // écrire un caractère au début
  }
  close(fd);
  return 0;
}
```

Complétez les appels de `lseek()` dans le code source du programme.

Quel sera le résultat du programme si on remplace `size / 2` par `size` dans la boucle **`for`** ?

Exercice 4 : Je dirai tout à papa. Une application est composée d'un processus père et d'un fils. Le fils réalise le travail suivant :

```
lire un réel x au clavier
tant_que x <> 0 faire
| si x < 0 alors
| | message d'avertissement
| sinon
| | transmettre f(x) au père
| demander un réel x au clavier
```

La fonction `f()` est définie dans le code du fils. Le père reçoit les valeurs `y` (= `f(x)`). Lorsque le

dernier `y` est reçu, il calcule la moyenne des `y` et l'affiche à l'écran. Les valeurs sont transmises par un tube.

On suppose que $\forall x \in \mathbb{R}, f(x) > 0$. Comment le père peut-il détecter la fin de la suite des valeurs reçues ? Écrire les algorithmes et les programmes C du père et du fils dans cette hypothèse.

On ne fait aucune hypothèse sur `f()`. Proposer une solution pour la détection de fin d'arrivée des valeurs reçues par le père et en donner les algorithmes puis les programmes C correspondants.

Exercice 5 : Entreprise familiale. Un travail à réaliser comporte une partie initiale globale (notée `TI`), deux travaux `T1` et `T2` chacun décomposé en 3 parties (`T1a`, `T1b`, `T1c` et `T2a`, `T2b`, `T2c`) et une partie finale `TF`. Les travaux `T1` et `T2` peuvent être réalisés simultanément avec les réserves suivantes, où « début » et « fin » indiquent les instants de début et de fin d'un travail donné :

— début(`T2a`) $\geq$ fin(`T1a`),
— début(`T1c`) $\geq$ fin(`T2b`).

En supposant qu'on travaille sur une machine mono-processeur, donner deux exemples d'exécution où les travaux $\texttt{T1}_i$ et $\texttt{T2}_j$ ne sont pas exécutés dans le même ordre.

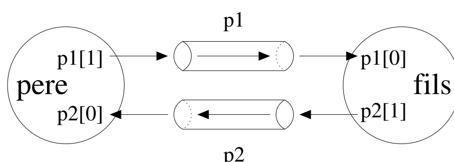
Implémenter ce travail à réaliser à l'aide de deux processus, un père et un fils, le fils réalisant le travail `T2` et le père le travail `T1`, la synchronisation étant réalisée grâce à des tubes (on utilisera la propriété de lecture bloquante des tubes).

5 Entraînement : exercice corrigé

Communication bidirectionnelle. Donner l'organisation d'une application de transmission bidirectionnelle d'informations entre un processus père et un fils via des tubes : le père envoie 5 entiers au fils qui les affiche et renvoie ces entiers multipliés par 2. Le père affiche ces doubles. Écrire ensuite le programme C correspondant.

Correction. Dans ce problème, on crée deux tubes `p1` et `p2` pour faire communiquer les processus :

— le père a accès en écriture sur `p1` et en lecture sur `p2`,
— le fils a accès en écriture sur `p2` et en lecture sur `p1`.



Un programme possible est le suivant. Pour des raisons de place, les tests d'échec des diverses primitives ne sont pas réalisés, ils sont bien sûr à ajouter dans un travail sérieux.

```
#include <stdio.h>
#include <unistd.h>
#define  NB_ENTIERS 5

void fils(int p1[2], int p2[2]) {
  int i, nombre, nb_lus;

  close(p1[1]);    /* Fermeture du tube p1 en ecriture pour le fils */
  close(p2[0]);    /* Fermeture du tube p2 en lecture pour le fils */
  nb_lus = read(p1[0],&nombre,sizeof(int));   /* Lecture nombre sur p1 */
  while (nb_lus == sizeof(int)) {
    nombre = nombre*2;                        /* Calcul du double */
    printf("Le double est : %d\n",nombre);
    write(p2[1],&nombre,sizeof(int));         /* Ecriture double sur p2 */
    nb_lus = read(p1[0],&nombre,sizeof(int)); /* Lecture nombre sur p2 */
  }
  close(p1[0]);    /* Fermeture du tube p1 en ecriture pour le fils */
  close(p2[1]);    /* Fermeture du tube p2 en lecture pour le fils */
  exit(0);
}

int main() {
  int i, nombre,
      p1[2],       /* Descripteurs du tube p1 */
      p2[2];       /* Descripteurs du tube p2 */

  pipe(p1);        /* Creation du tube p1 */
  pipe(p2);        /* Creation du tube p2 */

  if (fork() == 0) /* Creation du fils */
    fils(p1,p2);   /* Code du fils */
  else {
    close(p1[0]);  /* Fermeture du tube p1 en lecture pour le pere */
    close(p2[1]);  /* Fermeture du tube p2 en ecriture pour le pere */
    for (i = 0; i < NB_ENTIERS; i++) {
      printf("Entrez un entier\n");     /* Demande d'un entier */
      scanf(" %d",&nombre);             /* Saisie d'un entier */
      write(p1[1],&nombre,sizeof(int)); /* Ecriture de l'entier sur p1 */
    }
    close(p1[1]);  /* Fermeture du tube p1 en ecriture pour le pere */

    printf("Les doubles sont :\n");
    for (i = 0; i < NB_ENTIERS; i++) {
      read(p2[0],&nombre,sizeof(int));  /* Lecture d'un entier sur p2 */
      printf("%d ",nombre);             /* Affichage de l'entier */
    }
    printf("\n");  /* Important pour forcer l'affichage */
    close(p2[0]);  /* Fermeture du tube p2 en lecture pour le pere */
  }
  return 0;
}
```