Fichiers

Objectifs

- Comprendre la notion de descripteur de fichier
- Saisir le principe des lectures et écritures successives avec déplacement du curseur (*offset*)
- Implanter un algorithme de recopie de données via un tampon
- Comprendre et gérer l'aspect bloquant (ou non) des opérations de lecture et écriture
- Maîtriser les primitives système `open`, `read`, `write`, `lseek`, et `close`
- Comprendre la duplication des descripteurs lors de la création de processus et le partage des offsets

Ressources

Pour ce TP, comme pour les suivants, vous pourrez vous appuyer sur

- Le polycopié intitulé « Systèmes d'exploitation : Unix », qui fournit une référence généralement suffisante sur la sémantique et la syntaxe d'appel des différentes primitives de l'API Unix. Chaque section du sujet de TP indique la (ou les) section(s) du polycopié correspondant au contenu présenté.
- Les pages du manuel en ligne (commande `man`), et plus particulièrement les sections 2 et 3.

Déroulement et objectifs pour la séance
La durée de la séance de TP devrait (normalement) (à peu près) permettre à tous de traiter, dans l'ordre, les questions ① à ⑤.

Rendu (17/4) : code source répondant aux questions ② et ⑤. (Voir la page de dépôt Moodle)

0 Avant de commencer...

La réalisation du TP demande une connaissance de base de la syntaxe d'appel et de la sémantique des primitives de l'API fichiers Unix. Ces notions sont présentées de manière progressive dans le tutoriel proposé en **préparation** du TP (*Attention* : ce tutoriel ne remplace en aucun cas le TP ; il devrait être suivi en dehors de la séance de TP, dans le cas où vous estimeriez utile d'avoir une présentation « en douceur » de l'API).

Le QCM accompagnant ce TP va vous permettre de vous situer par rapport à cette connaissance de base. Comptez une petite dizaine de minutes. Si votre score est inférieur à 75/100, vous auriez sans doute (eu) intérêt à jeter un coup d'œil au tutoriel...

1 Prise en main

Pour prendre en main les primitives système de manipulation de fichiers, on se propose de réaliser un programme `copier` inspiré de `cp`. Ce programme, lancé par

```
./copier fichier_source fichier_destination
```

copie le contenu de `fichier_source` dans `fichier_destination` en écrasant le contenu de `fichier_destination` ou en le créant au besoin.

1.1 Descripteurs de fichiers

Un processus dispose d'une table permettant de désigner les fichiers qu'il a ouverts. Les éléments de cette table sont identifiés par leur indice (entier), appelé descripteur de fichier (*file descriptor*). Les descripteurs seront utilisés par le processus pour identifier le fichier sur lequel portent les opérations de lecture ou d'écriture demandées. Lorsqu'un processus ouvre un fichier, un descripteur est ajouté à cette table. Au départ, un processus a trois descripteurs de fichiers dans sa table : 0 (entrée standard), 1 (sortie standard), et 2 (sortie d'erreur).

Dans le répertoire spécial `/proc`, des informations sur chaque processus en cours d'éxécution sont accessibles dans un sous-répertoire avec pour nom son PID. À l'intérieur, le sous-répertoire `fd` donne un aperçu de sa table des descripteurs sous forme de liens vers les fichiers ouverts (ex. dans `/proc/1234/fd` pour le processus 1234).

① Lancez un `less` sur un fichier quelconque afin d'ouvrir ce fichier et afficher son contenu. Sur un autre terminal, identifier le PID du processus associé à ce "less" à l'aide de `ps` ou `pgrep` puis se rendre dans `/proc/le_pid/fd` et faire un `ls -l`. Constatez que les descripteurs de l'entrée standard, sortie standard, et sortie d'erreur, sont rattachés au terminal duquel a été lancé la commande [a] et que le fichier ouvert par `less` possède un descripteur dans la table.

a. Vous pouvez connaître/vérifier l'identité d'un terminal en lançant la commande `tty` dans ce terminal

Vous pourrez utiliser cette technique lors de vos tests pour avoir un état des lieux des descripteurs et fichiers qu'un processus manipule.

1.2 Implémentation du programme `copier`

1.2.1 Travail demandé

② Le programme `copier` à réaliser devra
- définir un tableau de `BUFSIZE` caractères qui servira de tampon mémoire ;
- ouvrir les fichiers source et destination ;
- lire les données du fichier source fragment par fragment, chaque fragment lu étant stocké dans le tampon mémoire, avant d'être (immédiatement) écrit dans le fichier destination ;
- fermer les fichiers source et destination, une fois le dernier fragment transféré.

Attentes et indications

1. Il est demandé en outre que l'implémentation se conforme aux **bonnes pratiques en matière de traitement des erreurs** suite aux appels système, c'est-à-dire :
   - utiliser une valeur de terminaison non nulle en cas d'erreur (seulement) ;
   - différentier les valeurs de terminaison en fonction des causes d'erreur.

   Pour cela, vous pourrez définir une fonction de traitement des erreurs qui sera appelée avec/après chaque appel de primitive système. Cette fonction devra simplement, en cas d'erreur, afficher un messages d'erreur avec `perror` et terminer le processus.
   Vous pourrez tester cette fonction en provoquant volontairement des erreurs.
2. Il est vraisemblable que le dernier fragment lu sera de taille inférieure à `BUFSIZE`. Il faut donc veiller à ne pas écrire dans le fichier destination plus d'octets que d'octets effectivement lus.
3. La taille du tampon sera définie par une constante `BUFSIZE`. La valeur de cette constante sera choisie suffisamment petite pour que le fichier source puisse être recopier fragment par fragment, et non en une seule fois. On pourra par exemple choisir 32 comme valeur. Notez cependant qu'en pratique, on utilise une taille de tampon beaucoup plus importante (p. ex. 4096 octets) [1].

La section suivante résume les principaux points techniques qui peuvent vous être utiles à la réalisation du programme `copier.`

1. Les échanges avec les périphériques comme le disque se font par blocs. Il est donc plus efficace que la taille du tampon soit (un multiple de) la taille d'un bloc. 4096 octets est une taille de bloc assez fréquente.

1.2.2 Documentation

Cette section n'est pas destinée à être lue du début à la fin : vous pouvez vous contenter de vous arrêter sur les points pour lesquels vous pensez avoir besoin de précisions.

Ouverture d'un fichier

Poly API UNIX section 3.1.1

*int open(const char *chemin, int options);*
*int open(const char *chemin, int options, mode);*

La primitive `open` (définie dans `fcntl.h`, voir `man 2 open`) ouvre un fichier au niveau du système. Au niveau du processus un descripteur est ajouté dans la table des descripteurs. `open` retourne la valeur de ce nouveau descripteur (un entier) ou -1 en cas d'erreur.

`open` prend en paramètre :

1. Le chemin vers le fichier à ouvrir (chaîne de caractères)
2. Des options sur la manière d'accéder au fichier, par exemple `O_RDONLY` pour la lecture seule. Consultez la signification des autres options dans le polycopié ou le manuel, notamment `O_WRONLY`, `O_RDWR`, `O_CREAT`, `O_TRUNC`, et `O_APPEND`. Les options peuvent être combinées avec `|` (ou bit à bit))
3. lorsque le fichier doit être créé (utilisation de l'option `O_CREAT`), un mode (droits d'accès comme pour `chmod`) doit être précisé. Le `man` présente des macros utiles pour définir le mode : par exemple `S_IRUSR|S_IRGRP|S_IROTH` définit des droits de lecture pour l'utilisateur, le groupe, et les autres (`r--r--r--`).

Lecture et écriture dans les fichiers

Poly API UNIX section 3.1.3

*ssize_t read(int desc, void *tampon, size_t nb);*
*ssize_t write(int desc, const void *tampon, size_t nb);*

`read` et `write` (définies dans `unistd.h`, voir `man 2 read`, et `man 2 write`) permettent de lire et écrire dans un fichier. `read`/`write` retourne le nombre d'octets effectivement lus ou écrits [2], ou -1 en cas d'erreur.

`read` et `write` prennent en paramètre

1. le descripteur de fichier `desc` fourni par l'appel à `open` ;
2. l'adresse d'un tampon recueillant/contenant les données lues/écrites ;
3. le nombre `nb` d'octets à copier dans/depuis le tampon [3] (généralement la taille du tampon).

Il faut enfin noter que les accès au fichier se font à partir d'une position courante (curseur ou *offset*), et que chaque accès fait progresser ce curseur du nombre d'octets lus/écrits.

Fermeture d'un fichier

int close(desc)

`close` (voir `man 2 close`) permet de fermer un fichier après utilisation. `close` prend en paramètre le descripteur de fichier `desc` fourni par l'appel à `open`, et retourne 0 (ou -1 en cas d'erreur).

2. `read` ou `write` peuvent lire ou écrire moins d'octets que demandé, par exemple dans le cas où un signal est reçu en cours d'accès (`read` et `write` peuvent être bloquantes), ou encore, pour la lecture si la fin du fichier est atteinte, auquel cas la valeur de retour est zéro.
3. `size_t` est un type d'entiers qui désigne une taille de donnée. `ssize_t` désigne également une taille ou la valeur -1 pour signifier une erreur.

2 Création de processus et accès concurrents **Poly section 3.1.8**

Lors de la création d'un fils avec `fork`, l'espace mémoire du processus est dupliqué et cela inclut la table des descripteurs de fichiers.

③ Mettez en évidence cette duplication en ouvrant un nouveau fichier `temp.txt` en écriture puis créez un nouveau processus. Le père (resp. fils) écrira 10 fois (une par seconde) "PERE\n" (resp. "FILS\n") dans le fichier. Testez le programme et constatez le résultat. On pourra utiliser la fonction `dprintf` (qui fonctionne comme `printf` en précisant en premier argument un descripteur de fichier) pour simplifier cette manipulation. Au besoin, on pourra utiliser les macros `STDIN_FILENO` et `STDOUT_FILENO` qui correspondent à 0 et 1, les descripteurs de l'entrée et de la sortie standard.

④ Reprenez exactement le même scénario en effectuant cette fois une ouverture du fichier dans chaque processus au lieu d'une seule avant le `fork` et constatez les différences.

Dans le dernier cas, des écritures d'un processus ont écrasé celles de l'autre. La position courante sur le fichier (qui se déplace pendant les lectures et écritures) n'est plus partagée : il y a deux ouvertures différentes sur le fichier au niveau du système.

`lseek` (voir `man 2 lseek`) permet de manipuler la position courante sur le fichier et prend en paramètres :

1. Le descripteur de fichier sur lequel effectuer l'opération.
2. Le nombre d'octets par rapport à une position de référence.
3. La position de référence en question parmi lesquelles `SEEK_SET` le début du fichier (positionnement absolu), `SEEK_CUR` la position courante, et `SEEK_END` la fin du fichier.

```
off_t lseek(int fd, off_t offset, int whence);
```

⑤ Écrivez un programme `scruter` dont la procédure `main(-)` lance successivement deux processus. Le premier processus ouvre un fichier `temp` et y écrit les entiers de 1 à 30, un par seconde, en revenant au début du fichier tous les 10 entiers. Le second processus (frère du premier) affiche régulièrement (par exemple toutes les 5 secondes) le contenu du fichier `temp` sur la sortie standard (en affichant un entier par ligne). Prévoir le contenu final du fichier et comparer avec les observations.

Pour cet exercice, on écrira/lira les entiers directement (sous leur représentation interne) **sans passer par une représentation sous forme de chaîne de caractères** conservée dans un tampon de caractères. Autrement dit :
les entiers seront écrits/lus directement dans le fichier `temp` par `write`/`read` et non par des fonctions de la famille `printf`/`scanf`. Par contre, l'affichage sur le terminal pourra se faire au moyen de `printf`.

Compléments (*optionnels*, à voir à votre gré, *après* la séance de TP)

Quelques exercices sont disponibles en annexe, qui abordent des points plus spécifiques ou techniques, qui ne sont pas indispensables, mais peuvent être utiles à connaître : E/S non bloquantes, opérations sur les répertoires, opérations sur les i-nœuds.