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 de la séance :

- La durée de la séance de TP devrait (normalement) (à peu près) permettre à tous de *lire attentivement* le texte et de traiter, dans l'ordre, les questions ① à ⑮.
- Les questions suivantes (⑯ à ⑲) ne présentent pas de difficulté particulière, mais abordent des points plus spécifiques ou techniques, qui, tout en étant utiles, vont au delà du strict minimum requis.

1 Prise en main

Pour prendre en main les primitives système de manipulation de fichiers, on se propose de réaliser étape par étape un programme `copier` inspiré de `cp` qui, lancé par `./copier fichier1 fichier2` copie `fichier1` dans `fichier2` en écrasant `fichier2` 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 Ouverture d'un fichier **Poly API UNIX section 3.1.1**

La primitive `open` (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. Elle prend en paramètre :

1. Le chemin vers le fichier à ouvrir
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`

```
int open(const char *pathname, int flags);
```

② Utilisez `open` pour ouvrir en lecture le fichier dont le nom est passé en premier argument du programme (le fichier que l'on va copier). Traitez les cas d'erreurs (fichier inexistant, pas les bons droits d'accès, etc). La fonction `perror` peut générer automatiquement des messages d'erreur. Affichez le numéro du descripteur lorsque l'ouverture réussit.

`open` prend un 3e paramètre lorsque le fichier est à créer (`O_CREAT`) : le mode (permissions réglables avec `chmod`). La doc présente des macros que l'on peut utiliser : par exemple `S_IRUSR|S_IRGRP|S_IROTH` définit des droits de lecture pour l'utilisateur, le groupe, et les autres (`r--r--r--`).

```
int open(const char *pathname, int flags, mode_t mode);
```

③ À la suite, ouvrez en écriture le fichier dont le nom est passé en second argument du programme. L'écraser s'il existe, sinon le créer avec les droits de lecture et écriture pour l'utilisateur. Traitez les cas d'erreur. Affichez le numéro du descripteur lorsque l'ouverture réussit. Vérifiez qu'un nouveau fichier avec les bons droits (`ls -l`) est bien créé. Vérifiez que le contenu d'un fichier qui existe déjà est bien effacé. **Prenez garde à ne pas tester votre programme sur des fichiers utiles, leur contenu serait écrasé.**

1.3 Lecture et écriture dans les fichiers **Poly API UNIX section 3.1.3**

`read` et `write` (dans `unistd.h`, documentation : `man 2 read`, `man 2 write`) permettent de lire et écrire dans un fichier. Le principe que l'on va détailler consiste à lire un fragment de données depuis un fichier, le recopier dans un tampon, puis écrire le contenu de ce tampon dans un autre fichier. En recommençant l'opération l'information sera copiée morceau par morceau d'un fichier vers l'autre.

④ Définissez une constante `BUFSIZE` à 5 pour les besoins de l'exemple. Définissez un tableau de caractères `tampon` de taille `BUFSIZE`. Il s'agira du tampon dans lequel nous ferons transiter les fragments de données.

`read` prend en paramètre un descripteur de fichier, un tampon où recopier les données lues, et enfin un nombre d'octets à recopier (typiquement la taille du tampon que l'on souhaite remplir). Pour le moment, on ignore la valeur de retour. Nous y reviendrons bientôt.

```
ssize_t read(int fd, void *buf, size_t count);
```

⑤ Ajoutez les appels nécessaires pour lire `BUFSIZE` octets du fichier ouvert en lecture et les recopier dans le tampon.

`write` prend en paramètre un descripteur de fichier, un tampon contenant les données à écrire, et enfin le nombre d'octets à écrire depuis ce tampon [1].

```
ssize_t write(int fd, const void *buf, size_t count);
```

1. `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.

⑥ À la suite, ajoutez l'appel nécessaire pour écrire les `BUFSIZE` octets du tampon dans le fichier qui a été ouvert en écriture.

⑦ Créez un fichier texte `source.txt` qui ne contient que le mot "INP-ENSEEIHT". Lancez votre programme `./copier source.txt cible.txt` et observez le résultat dans `cible.txt`

Seuls quelques caractères sont recopiés : tout ne rentrait pas dans le tampon. Un fichier ouvert est comporte un curseur (position courante, *offset*) qui se déplace à chaque opération de lecture ou écriture. En effectuant un nouveau couple de lectures/écritures nous n'allons donc pas reprendre à zéro mais avancer par palier en reprenant à chaque fois là où l'on s'est arrêté.

⑧ Effectuez un deuxième couple de lecture/écriture et testez à nouveau. Constatez que davantage de caractères ont été copiés. Ajoutez un troisième couple de lecture/écriture et testez. Si vous n'avez pas été soigneux, vous devriez constater une anomalie. Sinon, vous avez déjà résolu la question ⑨ :)

`read` renvoie le nombre d'octets qu'elle a pu lire (ou -1 en cas d'erreur). Il n'y a parfois pas assez ou plus assez à lire dans le fichier pour remplir tout le tampon. Dans ce cas il ne faut pas écrire l'intégralité du tampon au risque d'inclure des données parasites.

⑨ Modifiez votre programme afin d'éviter le problème rencontré à l'étape précédente. Testez sur l'exemple.

Lorsqu'il n'y a plus d'octets à lire, autrement dit : lorsque la fin du fichier est atteinte, `read` renvoie 0.

⑩ Généralisez avec une boucle la recopie avec lecture et écriture successives afin que le programme fonctionne sur des tailles arbitraires de fichiers et de tampons. Le programme `copier` est maintenant opérationnel !

En pratique on utilisera maintenant une taille de tampon beaucoup importante (p. ex. 4096 octets) [2].

Comme `open` et la plupart des primitives système, `read`, `write`, et `close` peuvent échouer (par exemple une opération sur un descripteur fermé, une écriture sur un descripteur en lecture seule, etc.) Il est important de détecter ces erreurs sur le moment afin d'éviter qu'elles soient passées sous silence et indirectement à l'origine de problèmes difficiles à diagnostiquer ensuite.

⑪ Généralisez un traitement des erreurs simple (par exemple en définissant une fonction réalisant l'affichage d'un message d'erreur avec `perror` et la fin du processus) effectué après chaque utilisation des primitives.

Notez que les bonnes pratiques en la matière sont

- d'utiliser une valeur de terminaison non nulle en cas d'erreur (seulement) ;
- de différentier les valeurs de terminaison en fonction des causes d'erreur.

Provoquez volontairement une erreur pour tester.

1.4 Fermeture d'un descripteur

`close` (voir `man 2 close`) permet de fermer un descripteur de fichier après utilisation. Il pourra ensuite être réutilisé. Elle prend en paramètre le descripteur à fermer.

⑫ Fermez les descripteurs des deux fichiers ouverts dans notre programme après utilisation. Pensez au cas d'erreur d'un descripteur déjà fermé (arrive souvent lors de malencontreux copier-coller).

`read` et `write` sont bloquantes. Pour certains fichiers (comme un terminal ou les tubes) `read` ou `write` peuvent se mettre en attente de données, et ainsi bloquer l'exécution du programme appelant. Une fermeture d'un descripteur de fichier en écriture (resp. en lecture) avec `close` peut éviter de causer un blocage non désiré d'un lecteur, en lui indiquant que la fin du fichier est atteinte, et qu'il n'y a plus d'écriture potentielle (resp. d'un écrivain en lui indiquant qu'il n'y a plus de lecteurs pour le fichier dans lequel il demande à écrire). Ce point sera détaillé dans le TP suivant.

2. 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.

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.

⑬ 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.

⑭ 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);
```

⑮ Écrivez un programme `scruter` qui ouvre un nouveau fichier `temp` et y écrit les entiers de 1 à 30, un par seconde, en revenant au début du fichier tous les 10 entiers. Dans un second processus (ni fils ni père du premier), afficher régulièrement le contenu du fichier sur la sortie standard (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) s**ans 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

Programme `cat`

⓰ S'inspirer du programme `copier` pour créer une version simplifiée de la commande `cat` qui affiche sur la sortie standard successivement le contenu des fichiers dont les noms sont passés en arguments. En l'absence d'arguments c'est l'entrée standard qui est utilisée.

Lecture non bloquante

Poly API UNIX sections 3.3.3 et 3.3.4

La lecture et l'écriture sont bloquantes. Les options `O_NONBLOCK` et `O_NDELAY` rendent (si possible) un descripteur de fichier non bloquant (aucun effet par exemple sur les fichiers ordinaires). On peut spécifier ces options en même temps que les autres, à l'ouverture du fichier (`open`). La primitive `fcntl` permet de modifier les options d'un descripteur de fichier déjà ouvert, on peut, entre autres, l'utiliser pour rendre bloquant ou non bloquant l'accès à un fichier après qu'il ait été ouvert.

La section 3.3.4 du poly API UNIX présente un exemple de lecture non bloquante. Elle inclut également, pour les besoins de l'exemple, une portion de code qui configure le terminal afin qu'il n'attende pas un retour de ligne pour fournir des données en entrée standard au processus.

⓱ Expérimentez l'exemple fourni. Comparez avec une lecture bloquante (prédire le comportement).

Répertoires

Poly API UNIX section 3.1.7

`mkdir` et `rmdir` (`fcntl.h` et `sys/stat.h`, voir `man 2 mkdir` et `man 2 rmdir`) permettent de créer ou supprimer des répertoires vides et prennent en paramètre un chemin (et des droits d'accès pour `mkdir`). `unlink` (voir `man 2 unlink`) prend un chemin vers un fichier et supprime le lien physique associé (elle coupe la branche associée dans l'arborescence). Si aucun lien physique ne pointe sur un fichier (il n'est plus dans aucun répertoire), il est considéré supprimé. En pratique la plupart des fichiers ne sont liés qu'à un répertoire et `unlink` est souvent synonyme de suppression.

⓲ Concevez un programme qui créée un répertoire temporaire de travail `temp` dans lequel des processus fils vont venir placer des fichiers de travail. Après la terminaison des processus fils, le processus père viendra vider puis supprimer ce répertoire.

i-nœuds

Les fichiers sont représentés dans le système par des *i-nœuds* : une structure de données qui contient des informations sur le fichier en question (droits d'accès, propriétaire, date de modification, taille, etc). À partir d'un chemin vers un fichier, la primitive `stat` (voir `man 2 stat`) permet d'accéder à cette structure.

⓳ Étudiez le fonctionnement de la primitive `stat` et écrivez un programme qui affiche des informations détaillées sur le fichier dont le nom est passé en argument.