Chapitre 5 : Fichiers

Section 5.1 : Gestion des utilisateurs

Un système multi-utilisateurs doit:

- être capable d'identifier et/ou authentifier les utilisateurs du système
- être capable d'exécuter des processus de différents utilisateurs simultanéments (et savoir qui est responsable de quoi)
- L'OS doit fournir des mécanismes simples pour contrôler l'accès aux différentes ressources
- L'Os doit être capable d'allouer certaines ressources à un utilisateur particulier à un moment donné.

UNIX associe à chaque username un nombre entier positif. La table est situé dans /etc/passwd :

- username
- password (old version of UNIX)
- userid
- identifiant du groupe principal auquel il appartient
- nom et prénom de l'utilisateur
- Répertoire de démarrage de l'utilisateur
- shell de l'utilisateur

5.1.1 Les 3 types d'utilisateurs UNIX

1. **Root** : L'administrateur du système (ID 0)
    - Droit de réaliser toutes les opérations sur le système
        - Création de nouveaux utilisateurs
        - Reformater les disques
        - arrêter le système ou un processus
        - accès à tous les fichiers
2. **Utilisateurs normaux**
    - accès réduits à leur environnement
3. **utilisateurs Daemon**
    - utilisateurs crées pour lancer un programme en tâche de fond

On associe à chaque processus un identifiant utilisateur sotcké dans la table des

processus. On peut :

- le récupérer avec getuid(2)
- le modifier avec setuid(2) (root)

> login appartient à root qui setuid puis l'user s'auth et exécute execve pour lancer le shell de l'utilisateur

Groupes d'utilisateurs

- groupe principal
  - dans passwd
- groupe secondaire
  - dans etc/group

Section 5.2 : Système de fichiers

Le système d'exploitation doit faire face à différents dispositifs de stockage en fournissant la même interface. En pratique, la plus part des dispositifs dont sont composées de zones de stockage elles mêmes divisées en secteurs (générale) (blocs de centaines miliers d'octets)

Interface de bas niveau

```
int err=devise_read(addr_t addr, sector_t* buf)
int err=devise_write(addr_t addr, sector_t* buf)
```

Unix et les fichiers

UNIX utilise une abstraction de plus haut niveau avec son OS pour interragir avec les fichiers qui est appelé **système de fichiers** ou **file system**

Sous linux, les fichiers sont des suites ordonnées d'octets. Un nom leur est associé et les programmes l'utilisent pour accéder aux fichiers. rien ne garantit que l'on écrit sur des secteurs consécutifs.

inodes : lien entre les différents secteurs qui composent un fichier. il a une taille fixe.

```
    struct simple_inodes{
        uint16 mode;        // drapeaux et permissions
        uid_t uid;          // priopriétaire du fichier
        gid_t gid;          // groupe id
```

```
        uint32 size;            // taille fichier bytes
        uint32 atime;       // temps dernière acces
        uint32 mtime;       // temps dernière modif
        uint32 ctime;       // temps dernier changement d'état
        uint16 nlinks;      // nombres de liens vers ce fichier
        uint32 zone[10];        // liste ordonnées des secteurs
    }
```

Répertoire

Sous unix, le nom du fichier n'est pas directement associé au fichier. il est stocké dans un répertoire.

Un **Répertoire** est un abstraction qui permets de regrouper ensemble plusieurs fichiers et/ou répertoires.

Le répertoire est un fichier au format spécial, il contient le nom des fichier et leur types. Sous unix, les fichiers sont organisées sous la forme d'un arbre. (la racine est /.) on peut rapidement intégrer un système de fichiers via la commade mount.

david://home/student est un serveur

chaque répertoire contient 2 répertoires spéciaux :

- **.** : Le répertoire courant
- **..** : Le répertoire parent

Les métadonnées associées aux permissions:

- r : autorisation de lecture (4)
- w : autorisations d'écriture et de modifications (2)
- x : autorisation d'exécution (1)

Ces permissions se trouvent sous la forme : Owner-Group-Others. on peut les modifier via **chmod**. Pour qu'execve puisse se lancer, il est nécessare de posséder les bits r et x correspondant (on additionne les nombres pour obtenir les permission.

Chemin de fichiers

- **absolu** : nom complet depuis la racine
- **relatif** : relatif depuis la position dans les répertoires (.. parent)

readdir : lecture d'un répertoire (Attention pas de multithread)

Link et Unlink

Lien vers un fichier, un seul fichier accessibe depuis différents endroits
ln : rendre un fichier accessible depuis un autre nom dans le même répertoire (même inode)
rm : utilise un unlink

5.2.1 Utilisation des fichiers

Lecture et écriture dans un fichier

- Appels systèmes :
  - open
  - read
  - write
  - close
- Fonctions: (stdio)
  - fopen
  - fread
  - fwrite
  - fclose

Lors de l'exécution de open(2), l'OS vérifie si le processus à les accès pour le fichier puis retourne le descripteur de fichier(un int) :

- **-1** : erreur
- **0** : Entrée std
- **1** : Sortie std
- **2** : Sortie d'erreurs std

Si un processus accède à un fichier et que les permissions changent, il ne sera pas impacté.

Attention : en UNIX, il est important de ne pas trop ouvrir de fichiers en même temps et de bien les fermer une fois terminé.

Lors de l'ouverture d'un fichier, l'OS maintient en plus de l'inode , un offset pointer.

un **offset pointer** est la position actuelle de la tête de lecture/écriture au sein d'un fichier.

On peut lire et écrire dans un fichier avec :

```
    #include <unistd.h>
    ssize_t read(int fd, void *buf, size_t, size_t count);
    //descripteur de fichiers, pointeur vers la zone à écrire,
```

```
    ssize_t write(int fd, const void *buf, size_t, size_t count
    //descripteur de fichiers, pointeur vers la zone à lire, qu
```

Lorsqu'un fichier est écrit sur un ordinateur, envoyé par internet à un autre ordinateur on peut avoir les problèmes suivants :

- problèmes de compatibilité
  - pas les mêmes types de données
  - Représentation 32 ou 64 bits
- Représentation intra processeur
  - big endion :
  - little endion :

> **lseek** est utilisé pour déplacer l'*offset pointer*

Fichiers Temporaires

via la commande **mkstemp**(3)

```
    char template[]="/tmp/sinf1252procXXXXXX";
    int fd = mkstemp(template);
    unlink(template);
    //acces
    if (close(fd)==-1)
        //error
```

Duplication de description de fichiers

dup : arg : descipteur et retourne le plus petit descripteur libre (même offset pointer) et même mode d'accès fichiers. dup2

5.2.2 Les pipes

Une **pipe** est un flux de bytes unidirectionnel qui relie 2 processus qui ont un ancètre commun. En pratique, les pipes sont utilisées notament par le shell. Lorsqu'un pipe (‘|’) est utilisé en shell. Il relie la sortie std de la première avec l'entrée std de la seconde.

Lorsque 2 processus ont un ancètre en commun, il estpossible de les relier via un pipe. Sinon, il est possible d'obtenir un résultat similaire avec fifo. On peut créer un fifo avec mkfifo(idem que open).

Section 5.3 Signaux

Les **signaux** sont les mécanismes de communication le plus primitif sous unix. Sous forme d'interruptions logiciel

Il en existe 2 types :

- Signaux synchrones : directement causé par l'exécution d'un processus
- Signaux asunchrones : pas directement causé par une instruction d'un processus

Il existe différents signaux avec différents effets. on peut contourner les signaux avec des handler. Chaque signal est identifié par un entier positif reconnu par le programme. Les principaux signaux sont :

- **SIGALARM** : Lorsqu'une alarme survient ou appel à settimer prends fin
- **SIGBUS** : Erreur au niveau matériel
- **SIGSEGV** : erreur mémoire (accès hors de la zone mémoire)
- **SIGFPE** : erreur mathématique (0/0)
- **SIGTERM** : commande kill (peut être traitée)
- **SIGKILL** : idem SIGTERM mais pas traitable et surpasse tout pour kill
- **SIGINIT** : ctrl-c termine normalement le programme
- **SIGUSR1** et **SIGUSR2** : peut être utilisé par le processus sans coord particulière
- **SIGCHLD** : indique qu'un processus fils à terminé (généralement ignoré)
- **SIGGHUP** : au début indiquait la perte de liaison avec terminal, mais maintenant est utilisé dans les serveurs pour recharger les fichiers de config

> SigKILL ne libère pas proprement la mémoire et les ressources

5.3.1 Envoi de signaux

Commande **kill(pid, sig)**:

- pid > 0 : signal au processus identifié
- pid == 0 : tous les processus du même groupe de processus
- pid == -1 : tous les processus pour lesquels il a les permissions
- pid < -1 : tous les processus du groupe abs //>

5.3.2 Traitement des signaux

- signals (n°signal, fonction handler)
- sigaction () pas étudié dans ce cours (skip)

5.3.4 Temporiser

création de timeout pour des fonctions (ex :appels systèmes bloquants)

Section 5.4 : Sémaphores nommées

(skip) bref : sémaphores avec un nom qui gère la coordination entre les processus

Section 5.5 : Partage de fichiers

- Moyen de communication principal entre les procesus
- persistance des données
- lenteur
    - réglé par l'utilisation d'un buffer (taille change en fonction de la charge du système)
        - vidé régulierrement par L'OS

5.5.1 Plusieurs processus, un seul fichier

Lorsqu'un processus ouvre un fichier, le noyau d'os lui associe le premier decripteur libre de la table. Ce descripteur pointe vers un *open file object*

un **Open File Object** : structure maintenue par le noyau qui contient différentes informations qui sont nécéssaires pour faire les différentes manipulations et opérations sur les fichiers. il comprend :

- Mode
- L'offset pointer (tête de lecture et d'écriture d'un fichier)
- référence vers système de fichiers (généralement inode)

Coordination

pread et pwrite prennent comme argument l'offset pointer et garantissent que les read et write seront atomiques.

Dans l'exmemple d'un database, on à besoin de verrous (lock) (= mutex). Il existe deux techniques :

- **Mendatory lock** : les processus placent des locks et l'os vérifie qu'il n'y a pas de violation de locks

- **Advisory lock** : les processus vérifient eux-mêmes les locks (+ commun , + simple à implémenter, meilleures performances)

utilisation de **flock** (skip)