

Architecture et Systèmes

Stefan Schwoon

Cours L3, 2016/17, ENS Cachan

Utilisateurs dans Unix

Les systèmes d'exploitation modernes gèrent une multitude d'**utilisateurs** :

“personnes naturelles”

“utilisateurs virtuels” (admin, “daemons”)

voir `/etc/passwd` dans Unix

Objectif : partager des ressources en respectant les contraintes de sécurité, les données privées etc.

Aspects pratiques : mots de passe, dossier ‘home’, ...

On s'intéressera surtout aux relations avec d'autres aspects (processus, fichiers, ...)

Identifiants utilisateur et groupe

Chaque utilisateur est associé à un identifiant numérique (`user id` ou `uid`).

Chaque utilisateur appartient à un groupe *primaire*, et potentiellement à quelques groupes `supplémentaires`.

chaque groupe possède son identifiant numérique (`gid`)

voir `/etc/passwd` pour les groupes primaires

voir `/etc/group` pour les groupes supplémentaires

Les groupes sont déterminés par l'administrateur.

La commande `id` sert à afficher ses identifiants utilisateur et groupe.

Utilisateurs/groupes et processus

Les processus possèdent quelques attributs concernant les utilisateurs :

l'identifiant utilisateur réel (`getuid`)

l'identifiant utilisateur effectif (`geteuid`)

l'identifiants groupe réel et effectif (`getgid`, `getegid`)

une liste d'identifiants groupes supplémentaires (`getgroups`)

Ceux-ci déterminent la plupart des privilèges que possède un processus.

Normalement, ces identifiants sont hérités du processus père lors d'un `fork` (on verra quelques exceptions plus tard).

Utilisateur réel et effectif

Utilisateur réel : l'utilisateur qui a lancé le processus

Utilisateur effectif : l'utilisateur qui détermine les privilèges du processus

Normalement, les deux identifiants sont identiques !

Exceptions : p.ex., `passwd` (pour changer de mot de passe)

L'utilisateur réel est celui qui change son mot de passe.

Or, il n'a pas le droit de modifier le fichier qui contient les mots de passe, seulement root les possède.

L'utilisateur effectif de `passwd` est donc root.

Le mécanisme pour changer d'utilisateur effectif s'appelle `setuid` ; on verra quand on discute les systèmes de fichiers.

Entrées/sorties

Les opérations entrées/sorties transmettent des données entre la mémoire et d'autres processus ou des périphériques.

Dans Unix/Posix, les entrées/sorties se font par des **fichiers**.

Un fichier dans Unix va au-delà d'une collection de données; c'est une structure de données abstraite qui possède au moins une opération de **lecture** et d'**écriture**.

Exemples

Un fichier peut représenter des réalisations différentes, p.ex.:

fichiers sur un disque dur

une zone de mémoire temporaire - buffer (console, pipe);

une structure dans le noyau (`/proc`);

les connections réseau

Les accès se font uniformément par les mêmes opérations, mais selon le type de fichier le noyau fait appel à un **pilote** (*driver*) qui y correspond.

Aspects de E/S

Stockage des données

système de fichiers

droits d'accès

organisation physique d'un disque dur

Gestion au niveau des processus

fonctions pour accéder aux fichiers/créer etc

fonctions pour manipuler les données d'un fichier

Systeme de fichiers

Unix gère un [système de fichiers](#) pour stocker des données au-delà de la vie d'un processus.

Ce système de fichiers est une arborescence :

Les nœuds internes sont les [dossiers](#) (ou [répertoires](#)).

Les feuilles sont des [fichiers](#) (ordinaires ou spéciaux).

Sur certains nœuds on peut greffer un arborescence supplémentaire (p.ex. une partition, une clé USB) ([mount point](#) en anglais).

Voir `mount` pour une liste des systèmes greffés.

Organisation d'un système de fichiers

Les nœuds sont référencés par des **chemins**:

chemin absolu: en commençant par la racine / et suivant les dossiers, p.ex.
`/home/schwoon/toto.txt`

chemin relatif on l'interprète en commençant dans un *dossier actuel*, p.ex.
`schwoon/toto.txt` si on se trouve dans `/home`.

Dans un chemin relatif, `..` veut dire le dossier en-dessus, `.` le dossier actuel.

Ce dossier actuel est un attribut du processus (modifier avec `chdir`).

Le dossier actuel est hérité par les processus fils.

Les chemins absolus et relatifs sont interprétés par toutes les fonctions qui gère les fichiers.

Inœuds

Structure de données pour stocker les méta-données d'un fichier permanent, p.ex. sur disque dur

Une partie d'un disque dur est réservée pour stocker ces inœuds.

Les méta-données stockés dans un inœuds contiennent le suivant :
type, propriétaire, groupe, droits d'accès, nombre de pointeurs vers cet inœud, les blocks où les données du fichiers sont stockés, . . . , **mais pas le nom du fichier.**

Relation entre fichiers et inœuds

Un inœud représente une unité de données sur disque; un fichier est une référence vers un inœud avec un nom.

Pour la plupart des fichiers, cette relation est un-à-un (mais pas, p.ex., pour les dossiers).

`ls -i` donne les identifiants du inœud associé avec un fichier ;
`stat` affiche les méta-données d'un inœud.

Organisation d'un dossier

Un **dossier** est un fichier spécial.

Ses données consistent d'une liste de son contenu, avec pour chaque item :

- son nom

- son inœud

Du coup plusieurs entrées peuvent référencer le même inœud avec des noms différents.

Un inœud (et les données stockés avec lui) est libéré lorsqu'on supprime le dernier lien vers lui (fonction `unlink` dans C).

Utilisateurs/groupes et fichiers

Chaque fichier (plus précisément : chaque inode) appartient à un utilisateur et à un groupe.

Les fichiers créés par un processus portent ses identifiants effectifs.

Droits d'accès sont déterminés par les uid et gid associés avec un processus :

9 “ugo” bits: (user,group,others) \times (read,write,execute)

3 autres bits: setuid, setgid, sticky

Commandes: `chmod`, `chown`, `chgrp`, `umask`

Droits d'accès : exemple

Supposons qu'un processus souhaite lire dans un fichier.

Si son id effectif utilisateur égale le propriétaire du fichier, on vérifie le bit (*user,read*).

Si le groupe du fichier est soit égale au groupe effectif du processus, soit dans ses groupes supplémentaires, on vérifie le bit (*group,read*).

Sinon, on vérifie le bit (*others,read*).

Pour écrire : c'est l'analogie avec *write*.

Le bit *execute* est utilisé lorsqu'on utilise une fonction de la famille `exec`.

Droits d'accès sur les dossiers

Les bits rwx ont une signification légèrement différente sur les dossiers :

read: on peut obtenir la liste des fichiers du dossier

write: on peut modifier la liste (créer, renommer, supprimer des fichiers)

execute: on peut obtenir (avec `stat`) les méta-données des fichiers

Setuid et setgid

Les bits setuid/setgid sur les *fichiers*:

Quand un processus exécute le fichier, ses identifiants effectifs deviennent ceux du fichier.

Les identifiants réels restent inchangés.

Le bit setgid sur les *dossiers*:

Les fichiers créés dans ce dossier portent le gid du dossier (et pas du processus qui les a créés).

Gestion des fichiers dans les processus

Chaque processus possède un ensemble de *descripteurs* (qui évolue).

Un descripteur donne accès à un fichier. Quand un processus détient un descripteur pour un fichier, on dit que le fichier est *ouvert* par ce processus.

Fonctions pour obtenir un descripteur (ouvrir un fichier) : `creat`, `open`, `pipe`,
...

`open` ouvre un fichier existant (pour lire ou écrire).

`fork` duplique le processus avec tous ses descripteurs.

`dup` duplique un descripteur au sein d'un processus.

`close` supprime le descripteur dans le processus actuel.

Techniquement, un descripteur est représenté par un simple entier.

Tableau des fichiers ouverts

Le noyau détient un tableau des fichiers ouverts de tous les processus.

Chaque ligne dans ce tableau représente un accès vers un fichier et contient des informations telles que :

inœud référencé, mode d'accès (lire/écrire/les deux), position dans le fichier, ...

Un même fichier peut être référencé par plusieurs lignes dans ce tableau (avec des accès différents).

Plusieurs processus peuvent partager une même référence vers ce tableau.

Une ligne dans ce tableau existe tant qu'il existe une référence vers elle dans un processus quelconque.

Descripteurs standard

Par défaut les trois premiers descripteurs d'un processus sont utilisés ainsi :

0 est l'**entrée standard** (stdin) (p.ex., `getch` ou `scanf` s'en servent)

1 est la **sortie standard** (p.ex., `printf` s'en sert)

2 est la **sortie erreur** (on est censé y mettre des messages d'erreur)

Dans le terminal, l'entree standard est typiquement alimenté par le clavier, et les sorties standard/erreur sont affichés sur l'écran (représentés par des fichiers spéciaux).

On peut changer ces descripteurs avec `dup` (utilisé par le shell lors des redirections).

Créer un descripteur

`open`: ouvrir un fichier existant. Exemples :

`open("myfile", O_RDONLY)` : ouvrir fichier en lecture (alternatives:
`O_WRONLY, O_RDWR`)

`open("myfile", O_WRONLY | O_CREAT)` : ouvrir pour écrire, créer le fichier
s'il n'existe pas

`open("myfile", O_WRONLY | O_CREAT | O_TRUNC, 0666)` : Comme
avant, mais détruire ancien contenu s'il en existe ; en plus, spécifier droits
d'accès.

`creat`: raccourci pour `open` avec `O_WRONLY, O_CREAT` et `O_TRUNC`

Lecture et écriture

`read/write(fd, p, n)` : lire/écrire n octets à partir de l'adresse p dans fichier fd

`read` et `write` renvoient les nombres d'octets qui ont été réellement lus/écrits (ce nombre peut être inférieur à n). Un renvoi de -1 veut dire erreur.

Il est conseillé de vérifier les valeurs renvoyées en cas d'erreur.

`read` un renvoi de 0 veut dire "fin de fichier".

`read` bloque si aucune donnée est disponible actuellement, mais d'autres processus pourraient en créer dans le futur.

Manipuler des descripteurs

`dup` and `dup2`: recopier un descripteur vers un autre

`g = dup(f)` : créer un nouveau descripteur `g` avec le même comportement que `f`

`dup2(f, g)` : recopier `f` vers `g`, même si `g` est déjà utilisé

`pipe`: créer un conduit unidirectionnel (tube)

```
int p[2]; pipe(p);
```

Les données écrites dans `p[1]` apparaissent dans `p[0]`.

Les tubes

Un tube permet à deux processus d'échanger des données.

La lecture sur un tube soit renvoie toute de suite les données qui sont dedans, soit elle bloque jusqu'à ce que des données arrivent (ou que tous accès en écriture ont été fermés).

Écrire sur un tube qui n'a plus d'accès en lecture donne un signal `SIGPIPE`.

Utilisation dans le shell : `cmd1 | cmd2`

Le shell crée un tube, plus fork deux fois, ferme ses accès au tube, et attend les deux fils.

Le premier fils ferme l'accès en lecture, redirige la sortie standard vers le descripteur écriture, et fait un exec sur `cmd1`.

Le deuxième fils ferme l'accès en écriture, redirige l'entrée standard vers le descripteur lecture, puis fait exec sur `cmd2`.

Modifier position de lecture/écriture

`lseek` modifie la position dans un fichier où la prochaine opération de lecture/écriture sera effectuée.

Pas disponible sur tous les types de fichier (p.ex. pas sur les tubes).

Syntaxe: `lseek (f, p, m)`, où `m` est l'une des valeurs suivantes :

`SEEK_SET`: se positionner à l'octet numéro `p` (premier octet à 0)

`SEEK_CUR`: avancer position par `p` octets

`SEEK_END`: position relative à la fin du fichier (`p` peut être négatif)

Renvoie la position obtenue (peut être utilisé pour déterminer la taille du fichier).

I/O: Descripteurs et streams

Sous C il existe deux familles de fonctions pour les entrées/sorties :

`open`, `write`, `read`, ...

Appels système défini par POSIX

travaillent sur les *descripteurs* (0, 1, 2, ...)

`fopen`, `printf`, `scanf`, ...

Fonctions utilisateurs défini par le standard ANSI-C

travaillent sur des *stream* (`stdin`, `stdout`, `stderr`, ...)

plusieurs modes de comportement en sortie : sans tamponnage,
tamponnage par bloc, tamponnage par ligne

Ne pas mélanger les appels des deux familles !

Streams en sortie avec tamponnage

Un stream est associé avec un descripteur.

Dans un stream en mode tamponné, les opérations d'écriture ne sont pas directement transmises au descripteur mais on stocke les données dans une zone tampon.

Transmission entre zone tampon et descripteur :

- sur appel de `fflush` ;

- sur fermeture du stream ;

- en mode tamponnage par ligne : sur apparition d'une nouvelle ligne (`\n`) ;

- en mode tamponnage par bloc : quand le tampon est rempli.

Opérations

Le mode de tamponnage peut être modifié par `setvbuf`.

Il existe quelques raccourcis, voir la page man.

Important lorsqu'on :

- fait passer des données à un autre processus (par un pipe)

- `printf` sans nouvelle ligne

`fdopen` crée un stream au-dessus d'un descripteur.