NOTES TP

• Une interruption est un événement asynchrone destiné au processeur, les causes d'une interruption peuvent être matérielles (horloge de la machine, périphérique d'E/S) ou logicielles (appel système)

• Un appel système est une interface entre un programme utilisateur et le noyau du système d'exploitation. Il s'agit d'une manière standardisée pour les programmes utilisateur d'interagir avec les fonctionnalités fournies par le noyau du SE

• Lorsqu'un programme utilisateur souhaite effectuer une opération qui nécessite des privilèges ou un accès direct au matériel, il fait un appel système

• Ces opérations peuvent inclure la lecture ou l'écriture dans des fichiers, la création de processus, la gestion de la mémoire, l'interaction avec des périphériques, ...

Exemple :

Lorsqu'un programme a besoin de lire ou écrire sur le disque, il fait un appel système pour demander au noyau d'effectuer cette tâche en son nom, l'appel système agit comme une passerelle entre le mode utilisateur et le mode noyau, la transition est généralement déclenchée par une interruption

• il existe 2 modes d'exécution : utilisateur & noyau (privilégié)

• Mode utilisateur :

Les programmes en cours d'exécution dans ce mode n'ont qu'un accès limité aux ressources matérielles et aux fonctionnalités du système. Ils ne peuvent pas effectuer des opérations critiques qui pourraient compromettre la stabilité du système ou violer la sécurité, les programmes en mode utilisateur peuvent faire des appels système pour demander des services au noyau du système d'exploitation

• Mode noyau :

Dans ce mode en a un accès complet aux ressources matérielles, le SE peut alors répondre aux interruptions matérielles, effectuer des opérations critiques et répondre aux appels système émis par les programmes en mode utilisateur

Mode noyau (privilégié) → Appels système

Mode utilisateur → Fonctions des bibliothèques C (printf() …)

Remarque : le mode noyau est dangereux, par exemple si un programme se plante durant son exécution alors il va affecter tout le système d'exploitation ce qui va mettre l'ordinateur hors service, le mode utilisateur par contre est plus sûre car même si un programme se plante le système va continuer de marcher correctement car il est loin du noyau

Manipulation de Fichiers

Table de D.F (descripteur de fichier) :

0 ← stdin

1 ← stdout

2 ← stderr

3 ← vide

Quelques appels systèmes :

1) open : utilisé par un programme pour ouvrir un fichier spécifié sur le système de fichiers

Signature : int open(const char *path, int flags)

- path : C'est le chemin du fichier à ouvrir

- flags : C'est un ensemble de drapeaux qui spécifient le mode d'ouverture du fichier (lecture, écriture, création, etc.)

- mode : Il s'agit d'un argument optionnel utilisé lors de la création d'un fichier pour spécifier les droits d'accès du fichier

Valeur retournée : retourne un descripteur de fichier, qui est un entier non négatif servant à identifier de manière unique le fichier ouvert au sein du processus, retourne **-1** en cas d'erreur

Remarque : s’il n’y’a pas d’errer l'appel système open retourne au moins 3 choses : stdin, stdout et stderr

2) access : utilisé pour déterminer si un processus a des droits d'accès spécifiques à un fichier ou à un répertoire, il permet à un programme de vérifier les permissions associées à un fichier ou à un chemin de répertoire, en fournissant le chemin du fichier et le type d'accès demandé en tant que paramètres

Signature : int access(const char *path, int mode)

- path : C'est le chemin vers le fichier ou le répertoire dont on veut vérifier les permissions

- mode : C'est un ensemble de bits qui spécifie le type d'accès que le programme souhaite vérifier, comme la lecture, l'écriture ou l'exécution

Valeur retournée : retourne 0 si l'accès spécifié est autorisé et -1 en cas d'échec

3) read() : utilisée pour lire des données depuis un descripteur de fichier :

Signature : int read(int fd, void *buffer, size_t count)

- int fd : descripteur de fichier (file descriptor)

- void *buffer : Pointeur vers le tampon de réception, c'est l'endroit où les données lues seront stockées

- size_t count : nombre d'octets à lire

Valeur retournée : renvoie le nombre d'octets lus avec succès, et -1 en cas d'erreur

4) write() : utilisée pour écrire des données dans un descripteur de fichier

- int fd : descripteur de fichier, L'identifiant du fichier ouvert

- const void *buffer : pointeur vers le tampon de données à écrire

- size_t count : nombre d'octets à écrire

Valeur retournée : retourne le nombre d'octets écrits, qui peut être inférieur au nombre demandé en cas d'erreur

Rôle des drapeaux :

F_OK : vérifie l'exitance d'un fichier

R_OK : vérifie si le fichier est en mode lecture

W_OK : vérifie si le fichier est en mode écriture

X_OK : vérifie si le fichier est exécutable

O_RDONLY : ouvrir le fichier en lecture seule

O_WRONLY : ouvrir le fichier en écriture seule

O_RDWR : ouvrir le fichier en lecture et écriture

O_APPEND : ouvrir le fichier en mettre le curseur à la fin

O_CREAT : créer le fichier s'il n'existe pas

O_TRUNC : Tronquer (vider) le fichier à zéro s'il existe

Man 1 → commandes utilisateur

Man 2 → Appels système

Man 3 → fonctions des bibliothèques C

exit(1) → EXIT_SUCCESS

exit(-1) → EXIT_FAILURE

Quelques notes :

• le UMASK est plus prioritaire que les programmes .c donc même si on change les droits d'un fichier ses droits vont rester les mêmes que ceux définis par le UMASK

• Lorsqu'un appel système échoue, il modifie la valeur de la variable errno pour indiquer la nature de l'erreur

• mode_t est un type de données utilisé pour représenter les modes de fichiers, c'est-à-dire les autorisations associées à un fichier, li est souvent défini comme un type entier non signé

Stat : permet de récupérer des informations d'un fichier passé en paramètres

Signature : int stat(char *path, struct stat *p)

- le path commence à partir du dossier courant où le fichier exécutable est stocké

- il faut inclure les 2 bibliothèques sys/stat.h et sys/types.horloge

Valeur retournée : retourne un entier (0 en cas de succès) et rempli la structure P

```
typedef struct {
    int st_dev ;     \\ PID du fichier contenant le fichier
    int st_ino ;     \\ numéro de l'inode
    int st_mode ;  \\ les previliges du fichier
    int st_mlink ;   \\ nombre de liens symboliques
    int st_size ;     \\ PID du fichier contenant le fichier
    int st_uid ;      \\ user id of owner
    int st_gid ;      \\ group id of owner
    int st_atime ;   \\ date du dernier accès
    int st_mtime ;  \\ date de la dernière modification de statuts
}                                                    ou d'état
```

• La commande ls -l utilise l'appel système stat, après avoir récupérer le st_uid, elle utilise la fonction getpwuid de la bibliothèque <pwd.h> qui le prend en paramètres et créer une structure et retourne un pointeur vers cette structure

Signature : getpwuid(int st_uid)

typedef struct {

pw_name ;

pw_password ; \\ mdp de la session utilisateur

pw_uid ;

pw_gid ;

pw_dir ;

} // Ces données sont dans le etc (faire cat /etc/passwd)

Au lieu de faire : password *p = getpwuid(StatusBuffer.st_uid) ;

p -> pw_name ;

On fait : getpwuid(StatusBuffer.st_uid) -> pw_name ;

opendir(3) readdir(3) closedir(3) : le (3) veut dire qu'ils se trouvent dans le Man 3

DIR * = dirp = opendir(char *Path)

struct client *P = readdir(DIR* dirp)

printf("%s\n", P->d_name)

closedir(dirp)

En utilisant #include <dirent.h> :

Dirent {

char d_name[256]

int d_type

... d'autres variables qui ne nous intéressent pas et qui sont à éviter

}

Getopt : cette fonction parcourt la ligne de commande et retourne -1 à la fin

Exemple : int opt = getopt(argc, argv, "fr")

Si on met : après le f "f:r" cela veut dire qu'il faut mettre des arguments après le f

inode : c'est un structure qui contient les informations de chaque fichier, c'est l'équivalent du PCB pour les fichiers

Partitionnement et Montage

• Un disque dur c'est un matériel physique utilisé pour stocker les données dans un ordinateur, il peut être divisé en plusieurs partions logiques et chaque partition peut avoir son propre système de fichier

Sur Windows :

- Systèmes de fichier : NTFS - FAT

- Nommage des disques physiques : par des lettres comme C:, D: … cette technique de nommage ne permet pas de révélé est ce que C: et D: sont 2 disques physiques, ou 2 partitions du même disque

Sur linux :

- Systèmes de fichier : ext2 - ext3 - ext4

- Nommage des disques physiques : sda, sdb, sds, ect, Un chiffre est ajouté au nom de disque pour représenter une partition, par exp: sdal sda2 …

Sur linux on a 2 types de partitions : primaire & étendue

Partition primaire : c'est des partitions physiques, on peut avoir 4 partitions primaires au maximum

Partition étendue : c'est des participions logiques, et on peut avoir n'importe qu'elle nombre de partitions

Pour utiliser une partition sur linux, il faut la monter :

1- connecter le nouveau disque

2- Démonter le disque (dans le cas de flash disque par exemple), pour pouvoir effectuer des opérations sur ses partitions, avec la commande :

umount NomPartition

3- Partitionner le nouveau disque : créer des partitions sur ce disque avec la commande :

fdisk NomDisque

4- Formater les partitions crées, en leur associant un système de fichier avec la commande :

mkfs -t NomSystem. NomPartition.

Une fois formater on peut vérifier le type de système Installer avec la commande :

fsck -N Nompartition

5- Monter la partition sur des répertoires de l'arborescence avec la commande :

mount NomPartion NomDossier Montage