

Système d'exploitation

SYSIR3 - SYSG4

M. Bastreggi (mba)

Haute École Bruxelles Brabant — École Supérieure d'Informatique

Année académique 2020 / 2021

D'après le cours de M.Jaumain

Section ED

Espace Disque

- partitions - DOS - commandes
- GPT
 - ext
 - ext2
 - appels système
- partitions -

partitions DOS

Date des années 80

permet de placer plusieurs F.S. et S.E. sur un disque

- ▶ 4 partitions (primaires)
- ▶ Le **MBR** (Master Boot Record) va contenir un programme d'amorce et la table des partitions
- ▶ Chaque partition contient un Boot Record (BR) : premier secteur avec éventuellement un programme d'amorce.

contenu du MBR

- ▶ Un programme d'amorce (446 bytes)
- ▶ Une table des partitions : 4 descripteurs de 16 bytes
- ▶ 2 derniers bytes de "code magique" = 0x55AA (si code de boot - par le BIOS)

descripteur de partition

- ▶ 1 byte 0x0-non bootable, 0x80-bootable
- ▶ 3 bytes adresse du premier secteur (CHS)
- ▶ 1 byte type de F.S. (06(fat16), 0b(fat32), 0f(étendue), 82(swap), 83(ext,ReiserFS,JFS))
- ▶ 3 bytes adresse du dernier secteur (CHS)
- ▶ 4 bytes : LBA premier secteur
- ▶ 4 bytes : LBA taille en secteurs (après le premier)

devices et linux

"En unix tout est fichier"

- ▶ les devices sont décrits dans le dossier **/dev**
- ▶ les manipuler nécessite des **droits d'administration**

droits d'administration

sans droits sudo il faut connaître le mot de passe de
l'**administrateur**

- ▶ sudo cmd - (password root) - une **commande** admin
- ▶ su - (password root) - une **session** admin (à éviter)

visualiser la table via fdisk

```
#fdisk -l
```

Disque /dev/sda : 465,8 GiB, 500107862016 octets, 976773168 secteurs

Unités : secteur de 1 x 512 = 512 octets

Taille de secteur (logique / physique) : 512 octets / 512 octets

taille d'E/S (minimale / optimale) : 512 octets / 512 octets

Type d'étiquette de disque : dos

Identifiant de disque : 0x00000000

Device	Boot	Start	End	Sectors	Size	Id	Type
/dev/sda1		2048	8386559	8384512	4G	82	Linux swap / Solaris
/dev/sda2	*	8386560	134223871	125837312	60G	83	Linux
/dev/sda3		134223872	553648127	419424256	200G	83	Linux
/dev/sda4		1	1	1	512B	ee	GPT

Un disque avec MBR hybride (DOS-GPT)

contenu du BR

- ▶ 510 bytes de programme de chargement éventuel, celui qui est lu par le loader
- ▶ 2 bytes de signature 55AA

partition étendue

Il y a maximum 4 partitions primaires numérotées de 1 à 4. Une partition **primaire étendue** permet d'étendre cette limitation.

- ▶ C'est une partition primaire de type 0x05, 0x0F
- ▶ Contient une chaîne de partitions logiques
- ▶ Chaque partition logique commence par un secteur EBR (Extended Boot Record)
- ▶ L'EBR permet le chaînage des partitions logiques.

contenu du EBR

- ▶ 446 bytes de programme de chargement éventuel
- ▶ 4 descripteurs de partition dont seuls 2 sont utilisés.
- ▶ 2 bytes de signature 55AA

partition logique : table des partitions

Première entrée de la table des partitions décrit la partition logique même

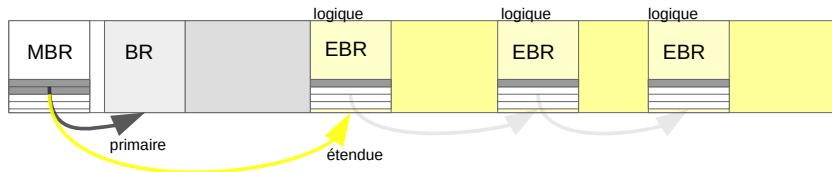
Deuxième entrée de la table des partitions pointe vers la logique suivante :

- ▶ 8 bytes 0
- ▶ 4 bytes = premier secteur de la partition logique suivante ou 0 si c'est la dernière.
- ▶ 4 bytes 0

chaînage

Les EBR des partitions logiques sont chaînés

Le nombre de partitions du disque est désormais limité par la taille du disque ($< 2 \text{ TiB}$).



limitation en taille

- ▶ LBA (Logical Block Address) du dernier secteur (512 bytes), codé sur 4 bytes (32 bits)
- ▶ LBA = un numéro de secteur



un disque avec table de partitions DOS a comme plus grande taille 2TiB

Système d'exploitation

└ Espace Disque

└ partitions - DOS - GPT

└ limitation en taille

- LBA (Logical Block Address) du dernier secteur (512 bytes), codé sur 4 bytes (32 bits)
- LBA = un numéro de secteur
- 💡 un disque avec table de partitions DOS a comme plus grande taille 2TiB

LBA est codé sur 4 bytes donc : Nombre MAX secteurs \times

Taille secteur = taille max disque $2^{32} * 512 \text{ bytes} = 2^{41} \text{ bytes}$

$= 2 * 2^{40} \text{ bytes} = 2 \text{ TiB}$ et la taille des secteurs est 512 bytes

$= 2^9 \text{ bytes}$

partitions GPT GUID Partition Table

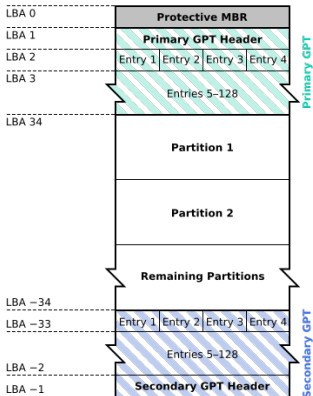
Les LBA sont sur 8 Bytes

Évite les limitations des 2TiB ainsi

- ▶ Le MBR (protective) contient également une table de partitions primaires qui ne fait que cacher le contenu du disque.
- ▶ Une partition primaire est utilisée pour couvrir l'entièreté du disque.
- ▶ Le type est spécifique à GPT (ee) et inconnu aux anciens outils de partitionnement.
- ▶ La table GPT commence au secteur 2 du disque.

partitionnement GPT

GUID Partition Table Scheme

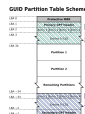


Système d'exploitation

└ Espace Disque

└ partitions - DOS - GPT

└ partitionnement GPT



128 descripteurs de 128 bytes chacun (la table utilise 32 secteurs du disque)

Les partitions commencent au secteur 34.

pour assurer plus de résistance aux pannes, l'entête et les 128 descripteurs de partition sont dupliqués en fin de partition.

Header

Les informations pour s'y retrouver

- ▶ adresse du secondary GPT header
- ▶ premier et dernier secteur utilisable pour les partitions
- ▶ ...

Entry : descripteur d'une partition GPT

- ▶ La table GPT contient 128 descripteurs de partitions.
 - 16 bytes pour le type de F.S.
 - 16 bytes d'identifiant de périphérique
 - LBA sur 8 bytes (LBA début et fin de la partition).
 - 72 bytes de nom.
 - ...

Système d'exploitation

- └ Espace Disque

- └ partitions - DOS - GPT

- └ Entry : descripteur d'une partition GPT

- La table GPT contient 128 descripteurs de partitions.
 - 16 bytes pour le type de F.S.
 - 16 bytes d'identifiant de périphérique
 - LBA sur 8 bytes (LBA début et fin de la partition).
 - 72 bytes de nom.
 - ...

Les LBA sont sur 8 bytes (64 bits), GPT permet de gérer des disques de taille $2^{64} \times 512$ bytes)

fdisk : partitionner

manipuler la table des partitions

```
#fdisk /dev/sdb
```

```
...
```

```
d  supprimer une partition
```

```
l  afficher les types de partitions connues
```

```
n  ajouter une nouvelle partition
```

```
o  créer une nouvelle table vide de partitions DOS
```

```
g  créer une nouvelle table vide de partitions GPT
```

```
p  afficher la table de partitions
```

```
...
```

m pour l'aide

mkfs : formater

formater = créer un système de fichiers dans une partition

```
#mkfs.vfat /dev/sdb1  
#mkfs.ext2 /dev/sdb2
```

mount : attacher au système de fichiers

mount attache un nouveau système de fichiers à un répertoire

```
#mount /dev/sda3/ /mnt/data // /mnt/data = point "d'accrochage de la racine"
```

```
#mount // consulte la liste de ce qui est monté  
/dev/sda3 on /mnt/data type ext2 (rw,relatime,noacl)
```

```
...
```

```
#umount /mnt/data // ou unmount /dev/sda3 pour défaire le lien
```


dd : commande de bas niveau

copie les octets d'un fichier sur la sortie standard (octal, hexadécimal, ...)

```
#dd if=/dev/sda bs=512 count=1 | od -tx1
```

- ▶ dd - affiche sur la sortie standard le MBR de /dev/sda
 - if - input file - le fichier lu
 - bs - bloc size = 512 Bytes
 - count - nombre de blocs de taille 512 Bytes
- ▶ od - convertir la sortie de dd (-tx1 : un byte à la fois, hexadécimal)

dd pour visualiser la table des partitions

Obtenir une image en hexadécimal de la table des partitions dans le MBR

```
#dd if=/dev/sda bs=1 skip=446 count=66 | od -tx1
```

```
00000000 00 fe ff ff 82 fe ff ff 00 08 00 00 00 f0 7f 00
00000020 80 fe ff ff 83 fe ff ff 00 f8 7f 00 00 20 80 07
00000040 00 fe ff ff 83 fe ff ff 00 18 00 08 00 e8 ff 18
00000060 00 00 01 00 ee fe ff ff 01 00 00 00 01 00 00 00
0000100 55 aa
0000102
```

Système d'exploitation

└ Espace Disque

└ partitions - commandes

└ dd pour visualiser la table des partitions

Obtenir une image en hexadécimal de la table des partitions dans le MBR

```
#dd if=/dev/sda bs=1 skip=446 count=66 | od -tx1
00000000 00 fe ff 82 fe ff 00 08 00 00 00 00 7f 00
00000020 80 fe ff 83 fe ff 00 85 7f 00 00 20 80 07
00000040 00 fe ff 83 fe ff 00 15 00 08 00 e8 ff 15
00000060 00 00 01 00 ee fe ff 01 00 00 00 01 00 00 00
00000100 55 aa
00000102
```

La table se trouve dans les 64 bytes qui précèdent la marque de boot du MBR.

if=input file, bs=bloc size (1 byte), skip=avancer,
count=nombre de blocs de 1 byte à lire

marque de boot du device 55AA (pour BIOS)

Le premier secteur de la partition 1 : 00 08 00 00 : 2048 en
Little Endian, 08 00 ($1000\ 0000\ 0000 = 2^{11}$)

Marque bootable pour la partition 2 (80)

Types de partition 82(swap), 83(ext2), ee(EFI-GPT)

dd pour visualiser la table des partitions

Pour un MBR GPT

```
#dd if=/dev/sda bs=1 skip=446 count=66 | od -tx1
```

```
00000000 00 00 01 00 ee fe ff ff 01 00 00 00 af 6d 70 74
```

```
66+0 enregistrements lus
```

```
66+0 enregistrements écrits
```

```
00000200 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
*
```

```
66 bytes copied, 0,000115112 s, 573 kB/s
```

```
0000100 55 aa
```

```
0000102
```

Système d'exploitation

└ Espace Disque

└ partitions - commandes

└ dd pour visualiser la table des partitions

Pour un MBR GPT

```
#dd if=/dev/sda bs=1 skip=446 count=66 | od -tx1  
00000000 00 00 01 00 ee fe ff ff 01 00 00 00 af 6d 70 74  
66+0 enregistrements lus  
66+0 enregistrements écrits  
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
+  
66 bytes copied, 0.000115112 s, 573 kB/s  
0000100 55 aa  
0000102
```

Une seule entrée de type ee cache le disque

La vraie table des partitions commence au secteur 2

commande parted

```
# parted -l
```

```
Model: ATA WDC WD5000AAKX-0 (scsi)
```

```
Disk /dev/sda: 500GB
```

```
Sector size ( logical / physical ): 512B/512B
```

```
Partition Table: gpt sync mbr
```

```
Disk Flags:
```

Number	Start	End	Size	File system	Name	Flags
1	1049kB	4294MB	4293MB	linux-swap(v1)	primary	
2	4294MB	68,7GB	64,4GB	ext2	primary	boot
3	68,7GB	283GB	215GB	ext2	primary	
4	283GB	283GB	16,8MB		primary	bios grub
5	283GB	348GB	64,4GB	btrfs	primary	boot, legacy boot
6	348GB	369GB	21,5GB	xfs	primary	

fdisk manipule aussi bien les disques gpt et vous protège des erreurs car l'écriture sur disque n'est pas immédiate.

open, read, lseek pour explorer par un programme c

- Lire le contenu brut d'un device via les appels système open, read, close, avec les droits administrateur.

```
struct maStructure struMBR;  
h=open("/dev/sdb",O_RDONLY); // open du disque sdb !  
read(h,&struMBR,512);        // lecture du MBR  
...  
close(h);
```

Système d'exploitation

└ Espace Disque

└ partitions - commandes

└ open, read, lseek pour explorer par un

programme c

open, read, lseek pour explorer par un
programme c

- Lire le contenu brut d'un device via les appels système
open, read, close, avec les droits administrateur.

```
struct maStructure struMBR;  
h=open("/dev/sdb", O_RDONLY); // open du disque sdb  
read(h, &struMBR, 512);      // lecture du MBR  
close(h);
```

Remarquez l'utilisation du paramètre de sortie (&) struMBR.

Questions ?



Avancement

- partitions - DOS - GPT
- partitions - commandes
- **ext**
- ext2
- appels système

système de fichiers

organiser des données au sein d'une partition nécessite la présence d'un système de fichiers

- ▶ FAT
- ▶ ext, ext2, ext3, ext4
- ▶ ReiserFS
- ▶ NTFS
- ▶ ...

La commande **mkfs** permet de créer un système de fichiers (formater la partition)

Système d'exploitation

└ Espace Disque

└ ext

└ système de fichiers

organiser des données au sein d'une partition nécessite la présence d'un système de fichiers

- FAT
- ext, ext2, ext3, ext4
- ReiserFS
- NTFS
- ...

La commande **mkfs** permet de créer un système de fichiers (formater la partition)

- `mkfs.vfat /dev/sdb1`
- `mkfs.ext2 /dev/sdb2`
- ...

structure ext

Un système de fichiers ext sur un mini-disque (partition) est composé de 4 zones :

- ▶ Boot area.
- ▶ Superbloc.
- ▶ Tableau d'inodes.
- ▶ Tableau de blocs.

Le système de fichiers a au maximum autant de fichiers que d'inodes.

structure interne

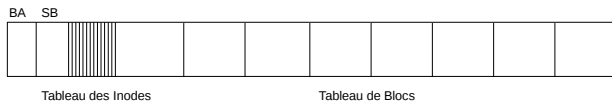


tableau de blocs

- ▶ Un bloc est un ensemble de bytes.
- ▶ Un bloc contient uniquement les données du fichier.
- ▶ La taille du bloc est exprimée en nombre de secteurs($n \times 512$ bytes).
- ▶ La taille et le nombre de blocs sont fixés à la création du Système de Fichiers (formatage)

inode ext

- ▶ Les métadonnées d'un fichier sauf son nom sont dans l'inode
- ▶ Le lien entre un fichier et son inode est fait par les répertoires.
- ▶ Le répertoire contient nom et inode de chaque fichier et sous-dossier qu'il contient.

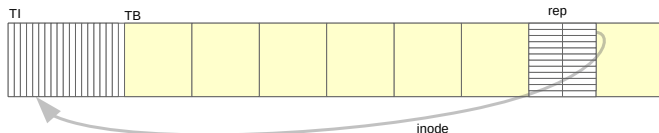


tableau d'inodes

Un inode est un ensemble structuré. L'inode contient uniquement les méta-données du fichier.

- ▶ Chaque inode a un numéro associé : son indice dans le tableau d'inode.
- ▶ L'inode contient toutes les informations concernant son fichier sauf les données.

contenu d'inode

- ▶ numéro du propriétaire (uid,gid).
- ▶ droits d'accès (rwxrwxrwx).
- ▶ type (normal, spécial, répertoire, lien soft, ...).
- ▶ dates (dernier accès, dernière modif des données, de l'inode, effacement).
- ▶ nombre de blocs.
- ▶ nombre de liens.
- ▶ taille du fichier.
- ▶ liste continue des blocs de ce fichier (10 pointeurs +3)

Système d'exploitation

└ Espace Disque

└ ext

└ contenu d'inode

contenu d'inode

- numéro du propriétaire (`uid,gid`).
- droits d'accès (`rw-rw-rw-`).
- type (normal, spécial, répertoire, lien soft, ...).
- dates (dernier accès, dernière modif des données, de l'inode, effacement).
- nombre de blocs.
- nombre de liens.
- taille du fichier.
- liste continue des blocs de ce fichier (10 pointeurs +3)

Pourquoi la liste CONTINUE des blocs ?

- La liste continue des blocs suit la numérotation des blocs au sein du fichier
- CONTINUE - permet de représenter les trous par des pointeurs à 0
- En ext, les fichiers creux n'occupent pas inutilement l'espace disque

liste des blocs

13 numéros de blocs (indices du tableau de blocs sur 4 bytes)

- ▶ Les 10 premiers sont les numéros des 10 premiers blocs de données du fichier.
- ▶ Le 11ème est le bloc d'indirection simple
- ▶ Le 12ème est le bloc d'indirection double
- ▶ Le 13ème est le bloc d'indirection triple

Système d'exploitation

└ Espace Disque

└ ext

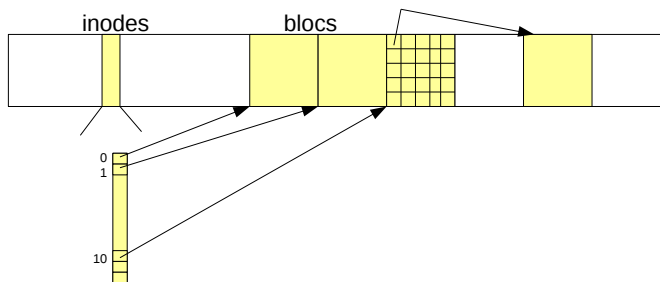
└ liste des blocs

13 numéros de blocs (indices du tableau de blocs sur 4 bytes)

- Les 10 premiers sont les numéros des 10 premiers blocs de données du fichier.
- Le 11ème est le bloc d'indirection simple
- Le 12ème est le bloc d'indirection double
- Le 13ème est le bloc d'indirection triple

la commande administrateur **debugfs** permet de voir le détail de l'utilisation des blocs par un fichier dans une Système de Fichiers ext

structure interne



Combien de blocs peut-on avoir pour un fichier si les blocs ont une taille de 2 secteurs ?

liste des blocs : quelques exemples

- ▶ Dessinez comment est mémorisé un fichier de 10 nombres ? de 2000 nombres ? de 200000 nombres ?
- ▶ Quelle est la taille d'un fichier contenant les 26 lettres de l'alphabet ? Quelle est son occupation sur le disque ?
- ▶ Quelle est la taille maximale d'un fichier si les blocs sont de 4 secteurs et les numéros de blocs sur 4 bytes ?
- ▶ Comment est représenté un fichier de 160000 bytes avec une taille de bloc de 512 bytes ?

liste des blocs

- ▶ /home est un minidisque sur un serveur linux, les blocs ont une taille de 8 secteurs.
- ▶ Combien de blocs peut-on avoir pour un fichier ?

Système d'exploitation

└ Espace Disque

└ ext

└ liste des blocs

- /home est un minidisque sur un serveur linux, les blocs ont une taille de 8 secteurs.
- Combien de blocs peut-on avoir pour un fichier ?

un bloc = 8 secteurs = 4096 bytes = 1024 pointeurs (4096 bytes / 4 bytes)

nb blocs = $10 + 1024 + (1024 * 1024) + (1024 * 1024 * 1024)$

fichiers creux

il est possible de créer un fichier creux en utilisant l'appel système lseek (se déplacer dans un fichier suivi de l'appel système write (écrire plus loin que la fin par exemple)

- ▶ Les 'positions' jamais écrites à l'intérieur d'un fichier sont lues par l'appel système read comme des 0 binaires.
- ▶ Un bloc vide n'est pas occupé sur le disque.
- ▶ Le numéro/pointeur d'un tel bloc est mis à 0 dans la liste des blocs dans l'inode.
- ▶ L'appel système read, renvoie artificiellement des 0 pour un tel bloc.

La commande **du -h** fichier , permet de vérifier la différence entre la taille logique et physique du fichier

répertoires

- ▶ Un répertoire est un fichier structuré et contient des enregistrements.
- ▶ Un enregistrement pour chaque fichier du répertoire (inode, nom)
- ▶ La racine (sommet de l'arborescence) à l'inode 2.
- ▶ L'inode 1 rassemble les blocs défectueux

répertoires

- ▶ Dessinez comment est mémorisé le répertoire root contenant les répertoires `.`, `..`, `home`, et `bin`.
- ▶ Dessinez comment est mémorisé le répertoire `/home/g12345` contenant `pgr1.asm` et `pgr2.asm`.

super bloc

Contient :

- ▶ début et fin des tableaux d'inodes et de blocs.
- ▶ nombre d'inodes et de blocs sur ce F.S.
- ▶ taille d'un bloc.
- ▶ si le F.S. a été correctement démonté.
- ▶ emplacement des blocs libres (sous forme de liste)

gestion des blocs libres

- ▶ Le super bloc contient un numéro de bloc de chaînage des blocs libres.
- ▶ Le dernier numéro de ce bloc est le numéro d'un bloc contenant à son tour des numéros de blocs libres.

lien hardware

Un lien hardware est un nouveau nom pour le même fichier.

- ▶ Dans le répertoire, le nom du fichier et son numéro d'inode.
- ▶ Dans le répertoire, le lien hard est un nom différent avec le même numéro d'inode.
- ▶ Dans l'inode, le compteur de liens est incrémenté.
- ▶ A l'effacement d'un nom, le compteur de liens est diminué de 1. S'il est à 0, le S.E. peut libérer les blocs du fichier.

Système d'exploitation

└ Espace Disque

└ ext

└ lien hardware

lien hardware

Un lien hardware est un nouveau nom pour le même fichier.

- Dans le répertoire, le nom du fichier et son numéro d'inode.
- Dans le répertoire, le lien hard est un nom différent avec le même numéro d'inode.
- Dans l'inode, le compteur de liens est incrémenté.
- A l'effacement d'un nom, le compteur de liens est diminué de 1. S'il est à 0, le S.E. peut libérer les blocs du fichier.

En réalité une autre condition est nécessaire à l'effacement : le fichier ne doit pas être ouvert par un process.

Cette particularité est exploitée pour les fichiers temporaires, pour être sûrs de leur suppression même si il y a un crash on fait suivre le open par un unlink du fichier. Il sera dès lors supprimé au close ou à la fin du process.

ext : faiblesses

- ▶ 14 caractères de nom de fichier (ou de répertoire).
- ▶ Inodes et blocs éparpillés sur disque : déplacement fréquent de la tête
- ▶ Liste des blocs libres -> fragmentation.
- ▶ Le superbloc unique avec des informations vitales pour le F.S.

Avancement

- partitions - DOS - GPT
- partitions - commandes
- ext
- **ext2**
- appels système

performances

Lire le secteur suivant du fichier, c'est

- ▶ positionner la tête sur le bon cylindre,
- ▶ attendre que la rotation du disque arrive sur le bon secteur
- ▶ activer la bonne tête de lecture

Pour des raisons de performance, il y a intérêt à grouper dans un (ou plusieurs contigus) cylindre, les informations inode, blocs correspondants, inodes libres et blocs libres.

améliorations en ext2

- ▶ amélioration des performances
- ▶ amélioration de la sécurité
- ▶ extensions

Système d'exploitation

└ Espace Disque

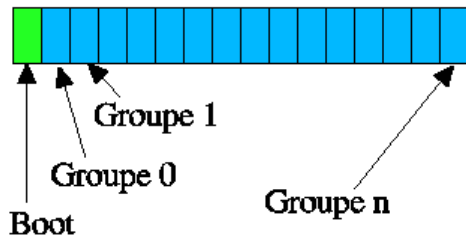
└ ext2

└ améliorations en ext2

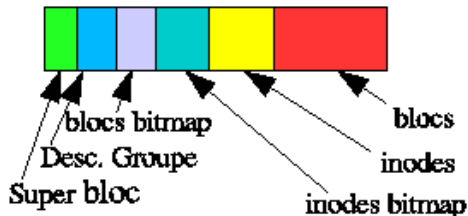
- amélioration des performances
- amélioration de la sécurité
- extensions

- performances - groupes et taille des blocs 4KiB
- sécurité - copies du SB
- extensions - noms longs, liens soft

performance groupes



Et, pour chacun de ces groupes,

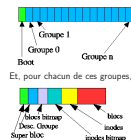


Système d'exploitation

└ Espace Disque

└ ext2

└ performance groupes



le descripteur de groupe contient les descripteurs pour les tables suivantes : emplacement des bitmaps, nombre de blocs et inodes libres pour le groupe, nombre de répertoires.

nombre de répertoires car ext2 essaye de les répartir sur tout le disque

stratégie groupes

Grouper l'information ne suffit pas. Il faut, si possible adopter les règles de gestion :

- ▶ tous les inodes d'un répertoire sont dans un même groupe.
- ▶ l'inode d'un nouveau répertoire est dans un autre groupe que celui du parent.
- ▶ les blocs et les inodes correspondants sont dans un même groupe
- ▶ les blocs d'un même inode sont alloués séquentiellement
- ▶ cette stratégie reste efficace si le disque n'est pas trop rempli (90 %)

stratégie allocation de blocs

l'allocation d'un nouveau bloc pour un fichier se fait en plusieurs étapes :

- ▶ recherche d'un bloc immédiatement voisin (même piste, à ± 16 blocs de 'distance'.
- ▶ sinon, réservation de 8 blocs libres dans le même groupe (un byte de la bitmap)
- ▶ sinon, recherche d'un bloc isolé du même groupe
- ▶ sinon, réservation de 8 blocs libres dans un autre groupe

La libération des blocs réservés non utilisés se fait à la fermeture du fichier.

Système d'exploitation

└ Espace Disque

└ ext2

└ stratégie allocation de blocs

stratégie allocation de blocs

l'allocation d'un nouveau bloc pour un fichier se fait en plusieurs étapes :

- recherche d'un bloc immédiatement voisin (même piste, à +/- 16 blocs de 'distance'.
- sinon, réservation de 8 blocs libres dans le même groupe (un byte de la bitmap)
- sinon, recherche d'un bloc isolé du même groupe
- sinon, réservation de 8 blocs libres dans un autre groupe

La libération des blocs réservés non utilisés se fait à la fermeture du fichier.

comment retrouve-t-on ces blocs ?

remarquez l'intérêt d'avoir une bitmap au lieu d'une liste de blocs libres

performance taille des blocs

- ▶ On améliore encore les performances par une taille de bloc de 4K (taille des pages mémoire)
- ▶ Les blocs de 4K sont donc composés de 8 secteurs.
- ▶ (*) Porter la taille des blocs de 1K à 4K provoque une perte de place de l'ordre de 50% par fragmentation interne.

sécurité superbloc

- ▶ Informations sur la structure du système de fichier.
- ▶ Pour améliorer la sécurité on en fait plusieurs copies
- ▶ On évite ainsi qu'il soit toujours sur le même plateau ou lu par la même tête.

sécurité superbloc

- ▶ Nombre de blocs et d'inodes du disque
- ▶ Taille d'un bloc
- ▶ Nombre de blocs et d'inodes par groupe
- ▶ Mount count et Max mount count
- ▶ Time last check et Max check interval ...

Système d'exploitation

└ Espace Disque

└ ext2

└ sécurité superbloc

- Nombre de blocs et d'inodes du disque
- Taille d'un bloc
- Nombre de blocs et d'inodes par groupe
- Mount count et Max mount count
- Time last check et Max check interval ...

Mount count et Time last check permettent de forcer un fsck après Max mount count montages ou Max check interval jours.

descripteur de groupe

- ▶ emplacement **bitmaps**
- ▶ nombre de blocs et inodes libres
- ▶ **nombre de répertoires** dans le groupe

extensions inodes

Le contenu des inodes a été complété :

- ▶ 12 adresses de bloc
- ▶ 3 adresses de bloc d'indirection simple, double et triple
- ▶ ACL (droits plus fins)

extensions noms longs

- ▶ Noms longs (ne sont plus limités à 14 caractères).
- ▶ La structure d'un répertoire change.
- ▶ On résout ce problème de compatibilité descendante avec les fonctions opendir, readdir...

répertoires ext2

Chaque entrée de répertoire a 3 champs de taille fixe et un de taille variable

- ▶ N° d'inode (4 bytes)
- ▶ Longueur de l'entrée (2 bytes) valeur étendue à un multiple de 4
- ▶ Longueur du nom de fichier (2 bytes)
- ▶ Nom du fichier (x bytes - taille variable)

le tout est étendu à un multiple de 4 bytes.

exemple - répertoires

- ▶ N° d'inode (4 bytes)
- ▶ Longueur de l'entrée (2 bytes)
- ▶ Longueur du nom de fichier (2 bytes)
- ▶ Nom du fichier (x bytes)

Voici le répertoire / contenant **..**, **..**, **UnFichier**, **user** :

```
00000002 000C 0001 .*** 00000002 000C 0002 ..**  
00000124 0014 0009 UnFichier*** 00000456 000C 0004 user
```

Système d'exploitation

└ Espace Disque

└ ext2

└ exemple - répertoires

- N° d'inode (4 bytes)
- Longueur de l'entrée (2 bytes)
- Longueur du nom de fichier (2 bytes)
- Nom du fichier (x bytes)

Voici le répertoire / contenant ., .., UnFichier, user :

```
00000002 000C 0001 .+++ 00000002 000C 0002 ..++
00000124 0014 0009 UnFichier+++ 00000456 000C 0004 user
```

Une entrée de répertoire ne peut franchir une frontière de bloc

exemple - répertoires

- ▶ N° d'inode (4 bytes)
- ▶ Longueur de l'entrée (2 bytes)
- ▶ Longueur du nom de fichier (2 bytes)
- ▶ Nom du fichier (x bytes)

Si on efface UnFichier ?

- ▶ On récupère l'entrée supprimée en agrandissant la taille de l'entrée précédente.
- ▶ Cette place pourra être récupérée si une nouvelle entrée peut y prendre place.

exemple - répertoires

Avant d'effacer un fichier :

```
00000002 000C 0001 .*** 00000002 000C 0002 ..** \\
00000124 0014 0009 UnFichier*** 00000456 000C 0004 user\\
```

Si on efface UnFichier, on obtient :

```
00000002 000C 0001 .*** 00000002 0020 0002 ..** (22 étoiles)** 00000456 000C 0
```

exemple - répertoires

Fonctions de lecture de répertoires (opendir, readdir, ...).

```
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <stdlib.h>
int main()
{ struct dirent *dirp; DIR *dp;
  chdir("/home/mba");
  dp=opendir(".");
  while ((dirp=readdir(dp))!=NULL)
    printf ("inode: %8d, taille: %6d, longueur: %6d: %s\n",
            dirp->d_ino, dirp->d_off, dirp->d_reclen, dirp->d_name);
  closedir (dp);
  exit (0);
}
```

Système d'exploitation

└ Espace Disque

└ ext2

└ exemple - répertoires

Fonctions de lecture de répertoires (opendir, readdir, ...).

```
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <stdio.h>

int main()
{ struct dirent *dirp; DIR *dir;
  char ("*/home/nelia");
  opendir("*");
  while ((dirp=readdir(dir))!=NULL)
    printf ("%s\t",dirp->d_name);
    dirp->d_ino,dirp->d_off,dirp->d_reclen,dirp->d_name);
  closedir (dir);
  exit (0);
}
```

Attention, Les données renvoyées par readdir() sont écrasées lors de l'appel suivant à readdir() sur le même flux répertoire. Les noms des fichiers sont terminés par 0.

extension liens soft

- ▶ Un répertoire contient le nom du fichier et son numéro d'inode.
- ▶ Dans le répertoire, le lien soft est un autre fichier il a son propre inode.
- ▶ Dans l'inode de ce lien soft, on définit le type comme un lien soft.
- ▶ Dans les données de ce lien soft, on inscrit le chemin spécifié dans la commande qui a créé le lien.

Plusieurs liens soft vers le même fichier, auront un contenu différent en fonction de l'endroit d'où a été établi le lien.

Que fait la commande `ln -s ../../cible ../rep/source ?`

Système d'exploitation

└ Espace Disque

└ ext2

└ extension liens soft

extension liens soft

- Un répertoire contient le nom du fichier et son numéro d'inode.
- Dans le répertoire, le lien soft est un autre fichier il a son propre inode.
- Dans l'inode de ce lien soft, on définit le type comme un lien soft.
- Dans les données de ce lien soft, on inscrit le chemin spécifié dans la commande qui a créé le lien.

Plusieurs liens soft vers le même fichier, auront un contenu différent en fonction de l'endroit d'où a été établi le lien.
Que fait la commande `ln -s ../cible ../rep/source` ?

quid de `ln -s ../cible ../rep/source`

créé dans `../rep` un fichier lien nommé `source`

`source` à une taille de 11 caractères et contient le texte

`"../cible"`

Système d'exploitation

└ Espace Disque

└ ext2

└ extension liens soft

extension liens soft

- Un répertoire contient le nom du fichier et son numéro d'inode.
- Dans le répertoire, le lien soft est un autre fichier il a son propre inode.
- Dans l'inode de ce lien soft, on définit le type comme un lien soft.
- Dans les données de ce lien soft, on inscrit le chemin spécifié dans la commande qui a créé le lien.

Plusieurs liens soft vers le même fichier, auront un contenu différent en fonction de l'endroit d'où a été établi le lien.
Que fait la commande `ln -s ../cible ../rep/source` ?

pour des chemins assez courts aucun bloc n'est réservé au fichier lien, le chemin est alors stocké dans la zone réservée aux pointeurs de bloc.

liens soft ext2

- ▶ Dessinez un F.S. ne contenant que le fichier '/home/origine' contenant abc.
- ▶ Ajouter le fichier /soft qui est un lien soft sur /home/origine.
- ▶ Ouvrez successivement les deux fichiers. Est-ce le même fichier ?
- ▶ Expliquer ce qui se passe quand on efface ces fichiers.

liens

- ▶ Peut-on faire un lien hard sur un fichier qui n'existe pas ?
- ▶ Peut-on faire un lien soft sur un fichier qui n'existe pas ?
- ▶ Que penser des backup d'un F.S. comprenant des liens ?
- ▶ Peut-on faire un lien hard sur un fichier qui se trouve sur un autre minidisque ?
- ▶ Un utilisateur peut-il récupérer la place d'un de ses fichier qui a été lié ?
- ▶ Citer des avantages et des inconvénients des deux types de liens.

liens

- ▶ Dessinez une arborescence de fichiers avec des liens.
- ▶ Que faut-il faire quand on efface un fichier ?

intégrité

- ▶ Il existe des outils qui permettent de tester et rétablir l'intégrité d'un système de fichiers.
- ▶ Le programme **fsck** est un de ces outils et tourne sous linux.
- ▶ Le test d'intégrité teste différents aspects nous en exposons deux dans la suite.

intégrité - blocs

Les blocs sont soit des blocs d'un fichier, soit des blocs libres. Lors d'un dysfonctionnement (coupure de courant, p.e.), des blocs peuvent être mal attribués. (bloc requis mais pas encore retiré des libres, ...): l'état du disque est incohérent

- ▶ Montrer qu'un état incohérent peut avoir des conséquences graves sur la sécurité d'un système.
- ▶ Comment savoir qu'un système a été victime d'une panne de courant ?
- ▶ Montrer que l'information 'bloc utilisé/inutilisé' apparaît deux fois. Il y a redondance de l'information.
- ▶ Que faut-il faire pour rétablir l'intégrité d'un F.S. ?

check

- ▶ Constituer une table, deux nombres par bloc.
- ▶ Le premier nombre correspond au nombre de fois qu'un bloc est référence par la liste des blocs de tous les inodes.
- ▶ Le deuxième nombre correspond au nombre de fois qu'un bloc est référencé par la liste des blocs libres.
- ▶ Citez les différents cas qui peuvent se produire.
- ▶ Dans ces différents cas, proposez une solution.

corrections

- ▶ 1-N : Les blocs attribués ne sont plus vides (1-0).
- ▶ 0-0 : Les blocs ni vides, ni attribués sont vides (0-1).
- ▶ N-0 : Les blocs attribués plusieurs fois sont recopiés (1-0, 1-0, ...N occurrences).
- ▶ 0-N : Les blocs N fois vides ne sont qu'une fois vide (0-1).
- ▶ Les cas 0-0 ne risque-t-il pas de détruire de l'information ? Proposer une solution.

intégrité - liens

Le nombre de liens représente le nombre de fois qu'un fichier est nommé dans l'arborescence. Il est aussi donné dans l'inode de ce fichier. Lors d'un dysfonctionnement (coupure de courant, p.e.), cette information peut être corrompue : l'état du disque est incohérent

- ▶ Montrer qu'un état incohérent peut avoir des conséquences graves sur la sécurité d'un système.
- ▶ Que faut-il faire pour rétablir l'intégrité d'un F.S. ?

check

- ▶ Constituer une table, deux nombres par inode.
- ▶ Le premier nombre correspond au nombre de fois qu'un inode est référencé en parcourant le F.S.
- ▶ Le deuxième nombre correspond au nombre de liens inscrit dans l'inode.
- ▶ Citez les différents cas qui peuvent se produire.
- ▶ Dans ces différents cas, proposez une solution.

corrections

- ▶ $N-L$ où $N > L$: Le nombre de liens est mis à N .
- ▶ $N-L$ où $N < L$: Le nombre de liens est mis à N .
- ▶ Que risque risque-t-il de se produire si on laisse $N > L$?
- ▶ Que risque risque-t-il de se produire si on laisse $N < L$?

Questions ?



question

- ▶ `handle = open(Nom, Mode)`
- ▶ `n = read(handle, adresse, nombre)`
- ▶ `n = write(handle, adresse, nombre)`
- ▶ `pos = lseek(handle, position)`
- ▶ `close(handle)`
- ▶ Décrire ce que fait chacun de ces appels système.

Questions ?



Avancement

- partitions - DOS - GPT
- partitions - commandes
- ext
- ext2
- appels système

appels système et librairie c

- ▶ Dans un système unix il existe une fonction de la librairie standard par Appel Système, elle porte souvent le même nom.
- ▶ Cette fonction masque le basculement de mode (INT, SYSCALL)
- ▶ Les arguments sont passés dans les registres
- ▶ Le retour d'un appel système en cas d'erreur est une valeur négative
- ▶ La variable globale **errno** contient, dans ce cas, le code d'erreur

errno

<**errno.h**> définit la **variable globale** entière **errno** renseignée par les Appels Système

La valeur de errno n'est significative que lorsque l'appel système a échoué (généralement en renvoyant -1)

errno

Comment utiliser errno ?

```
#include <stdio.h>
void perror (const char* monmsg ); // affiche "monsg : message d'erreur"

#include <errno.h>
printf ("%d\n" , errno); // pas beaucoup d'intérêt

#include <strings.h>
char * strerror (int errnum); // retourne message d'erreur
```

Attention, errno n'est pas positionné d'office -> il faut toujours tester le retour de l'appel système.

structures utilisées pour les fichiers

- ① sur disque
 - tableau d'inodes, tableau de blocs, répertoires
- ② en RAM ? Dans la mémoire de l'OS :
 - TDFO Table des Descripteurs de Fichiers Ouverts (n° inode, offset dans le fichier)
 - Globale au système
 - On y mémorise l'inode et la position de lecture/écriture pour un fichier ouvert
 - table de descripteurs de fichiers
 - contient des pointeurs vers la TDFO
 - une par processus dans la table des processus.
 - parfois appelée "table des handle"

open

- ▶ crée une nouvelle entrée dans la TDFO globale
- ▶ ajoute dans la table des handle du process, l'adresse (handle) de l'entrée créée
- ▶ retourne l'indice de ce nouveau handle (le premier libre : 3 (0,1,2 - stdin, stdout, stderr ...))

Système d'exploitation

└ Espace Disque

└─ appels système

└─ open

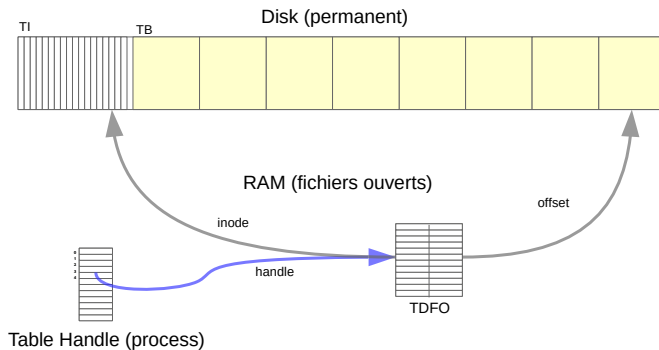
- crée une nouvelle entrée dans la TDFO globale
- ajoute dans la table des handle du process, l'adresse (handle) de l'entrée créée
- retourne l'indice de ce nouveau handle (le premier libre : 3 (0,1,2 - stdin, stdout, stderr ...))

Attention à la nomenclature qui peut induire en erreur. Les pages de manuel appellent "descripteur de fichier" l'indice (fd)

.

open

```
f=open(filename, flags , mode)
```



open

```
h=open(filename, flags)  
h=open(filename, flags, mode)
```

- ▶ filename - chemin
- ▶ flags - masque de bits pour le mode d'ouverture (O_RDONLY, O_WRONLY, O_RDWR, ...)
- ▶ mode - masque de bits pour les droits (exemple 0755)
- ▶ h - descripteur, l'indice de la table des handle

close

```
e=close(h)
```

Supprime le handle dans la table

L'entrée dans la TDFO ne sera pas toujours supprimée

2020-09-13

Système d'exploitation

- └ Espace Disque

- └ appels système

- └ close

close

`*close(h)`

Supprime le handle dans la table

L'entrée dans la TDFO ne sera pas toujours supprimée

Cela arrive dans le cas où un autre pointeur identique à celui-ci existe.

dup, fork

open - flags pour l'écriture

```
O_CREAT // nouveau fichier si n'existe pas,  
O_EXCL // avec O_CREAT, erreur si le fichier préexiste ,  
O_TRUNC // sera tronqué à une longueur nulle ,  
          // si le fichier existe , est un fichier régulier ,  
          // est ouvert en écriture ,  
O_APPEND // positionnement en fin avant chaque écriture
```

Système d'exploitation

- └ Espace Disque

- └ appels système

- └ open - flags pour l'écriture

```
O_CREAT // nouveau fichier si n'existe pas,
O_EXCL // avec O_CREAT, erreur si le fichier préexiste,
O_TRUNC // sera tronqué à une longueur nulle,
        // si le fichier existe, est un fichier régulier,
        // est ouvert en écriture,
O_APPEND // positionnement en fin avant chaque écriture
```

O_APPEND : Initialement, et avant chaque write(2), la tête de lecture/écriture est placée à la fin du fichier

Deux process ouvrent un fichier **sans** O_APPEND et y font plusieurs écritures

Deux process ouvrent un fichier **avec** O_APPEND et y font plusieurs écritures

Différence de comportement ? À tester ... P1 & P2

open - flags pour l'écriture

Quelques combinaisons des flags du mode d'ouverture

O_WRONLY		O_CREAT		// nouveau fichier si n'existe pas,
O_WRONLY		O_CREAT		O_TRUNC // idem mais troncature si existe
O_WRONLY		O_APPEND		// positionnement avant chaque écriture

open - mode

Le troisième paramètre définit les droits pour un nouveau fichier

Ce paramètre doit être fourni lorsque `O_CREAT` est spécifié dans `flags`; dans les autres cas il est ignoré.

```
mode = mode & ~umask  
mode = 0666 & ~022 = 0644 (rw-r--r--)
```

- ▶ Tient compte de la valeur de `umask` (022 par défaut)
- ▶ Ne s'applique qu'aux accès ultérieurs au fichier
- ▶ `mode = 0666` donne 0644

Système d'exploitation

└ Espace Disque

└─ appels système

└─ open - mode

open - mode

Le troisième paramètre définit les droits pour un nouveau fichier

Ce paramètre doit être fourni lorsque O_CREAT est spécifié dans flags ; dans les autres cas il est ignoré.

```
mode = mode & ~umask  
mode = 0666 & ~022 = 0644 (rw-r--r--)
```

- Tient compte de la valeur de umask (022 par défaut)
- Ne s'applique qu'aux accès ultérieurs au fichier
- mode = 0666 donne 0644

Comment savoir si le fichier doit être créé ?

O_CREAT et le fichier n'existe pas

La décision est prise à l'exécution

En absence du paramètre mode, RDX = valeur aléatoire

-> droits aléatoires !!

exemple - open oubli du mode

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>
main(){
    int h;
    unlink ("creatOK"); // suppression du fichier
    unlink ("creatNOTOK");
    h=open("creatOK",O_WRONLY | O_CREAT | O_TRUNC ,0666);
    close(h);
    h=open("creatNOTOK",O_WRONLY | O_CREAT | O_TRUNC);
    close(h);
}
```

```
>ls -l
-r-s--s--T 1 user0 users 0 29 janv. 12:17 creatNOTOK
-rw-r--r-- 1 user0 users 0 29 janv. 12:17 creatOK
```

exemple - oubli de O_TRUNC

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>
main(){
    int h;
    h=open("fichier.dat",O_WRONLY | O_CREAT | O_TRUNC ,0666);
    write(h,"abcdefghijklmnopqrstuvxyz\n",27);
    close(h);

    h=open("fichier.dat",O_WRONLY | O_CREAT ,0666); // sans O_TRUNC
    write(h,"bonjour!\n",10);
    close(h);
}
```

```
> cat fichier.dat
bonjour ! klmnopqrstuvxyz
```

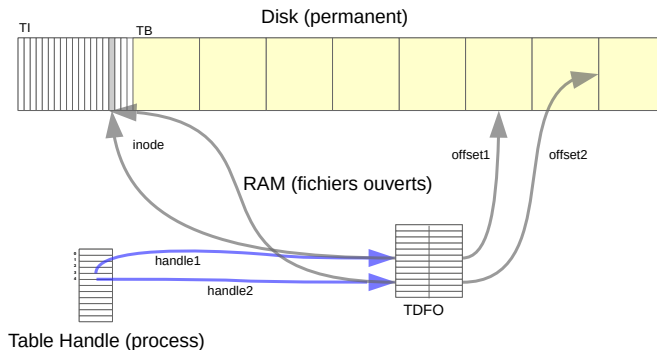
double open



Deux open successifs pour même fichier et même process, donnent deux indices et deux handles différents dans la TDFO (deux offsets qui évolueront indépendamment)

double open

Avec deux open du même fichier :



défi

Soient deux exécutions **simultanées** de ce code

Quelle taille et contenu aura le fichier nouveau.dat ?

Qu'en est il si on remplace O_TRUNC par O_APPEND ?

```
main (int argc, char * argv []){
    int    f, /* handle */
           i;
    if (argc < 2){          printf ("usage: _\%s_<lettre>\n", argv[0]);
                           exit (1);
    }
    unlink ("nouveau.dat"); // suppression du fichier
    f=open("nouveau.dat",O_WRONLY | O_CREAT | O_TRUNC,0666);
    for (i=1; i<=10; i++){
        write(f,argv [1],1);
    }
    close(f);
}
```

Système d'exploitation

- └ Espace Disque
 - └ appels système
 - └ défi

défi

Soient deux exécutions **simultanées** de ce code
 Quelle taille et contenu aura le fichier nouveau.dat ?
 Qu'en est il si on remplace O_TRUNC par O_APPEND ?

```
main(int argc, char *argv){
  int f, /* handle */
      i;
  if (argc < 2){
    printf ("usage: ./%s <lettre>\n", argv[0]);
    exit(1);
  }
  unlink ("nouveau.dat"); // suppression du fichier
  fopen("nouveau.dat", O_WRONLY | O_CREAT | O_TRUNC, 0666);
  for (i=0; i<=10; i++)
    write(f, argv[i], 1);
  close(f);
}
```

comment faire ce test ?

il n'y a pas de partage de l'entrée de la TDFO

chaque process écrit depuis la position 0 10 caractères

les écritures se superposent - dépend de l'ordre

d'ordonnancement

wc -c nouveau.dat

10 nouveau.dat

et si on retire les deux flags ?

stat, fstat et contenu d'inode

```
e=stat(filename, struct stat *strstat )  
e=fstat(h, struct stat *strstat )
```

- ▶ la structure stat contient une partie du contenu d'un inode
- ▶ man 2 stat pour une série de macros POSIX :
S_ISREG(mode), S_ISDIR(mode), ...

Système d'exploitation

└ Espace Disque

└ appels système

└ stat, fstat et contenu d'inode

```
#include <sys/stat.h>
struct stat;
```

- la structure stat contient une partie du contenu d'un inode
- man 2 stat pour une série de macros POSIX :
S_ISREG(mode), S_ISDIR(mode), ...

lstat - pour un fichier lien

structure stat

Quelques champs :

ino_t	st_ino;	<i>/* Numéro i-noeud</i>	<i>*/</i>
mode_t	st_mode;	<i>/* Droits, type de fichier</i>	<i>*/</i>
nlink_t	st_nlink;	<i>/* Nb liens matériels</i>	<i>*/</i>
uid_t	st_uid;	<i>/* UID propriétaire</i>	<i>*/</i>
gid_t	st_gid;	<i>/* GID propriétaire</i>	<i>*/</i>
off_t	st_size;	<i>/* Taille totale en octets</i>	<i>*/</i>
blksize_t	st_blksize;	<i>/* Taille de bloc pour E/S</i>	<i>*/</i>
blkcnt_t	st_blocks;	<i>/* Nombre de blocs alloués</i>	<i>*/</i>
time_t	st_atime;	<i>/* Heure dernier accès</i>	<i>*/</i>
...			

exemple - stat

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
main()
{ int h; struct stat inode;
  h=open("alphabet.dat",O_WRONLY | O_CREAT | O_TRUNC,0666);
  write(h,"abcdefghijklmnopqrstuvwxy",26);
  close(h);
  stat("alphabet.dat",&inode);
  printf("st_ino_\n",inode.st_ino);
  printf("st_mode_\n",inode.st_mode);
  printf("st_uid_\n",inode.st_uid);
  printf("st_size_\n",inode.st_size);
  printf("st_blocs_\n",inode.st_blocks);
  printf("st_atime_\n",inode.st_atime);
  exit(0);
}
```

Sticky bit, SGID, SUID

Masque de bits du **mode** du fichier

- ▶ STICKY BIT - (01000) - pour un répertoire : suppression de fichiers autorisée au propriétaire du fichier ou du répertoire (casier)
- ▶ SGID - (02000) - s'exécute avec les droits de son groupe peu importe quel utilisateur l'exécute
- ▶ SUID - (04000) - s'exécute avec les droits de son propriétaire peu importe quel utilisateur l'exécute

ruid, euid

Qui suis-je ? Quels droits ai-je sur les fichiers ?

- ▶ real uid/gid - qui nous sommes et notre groupe
- ▶ effective uid/gid - détermine nos droits d'accès aux fichiers
- ▶ normalement ces deux valeurs sont identiques
- ▶ elle peuvent différer lorsqu'on exécute un programme dont les droits SUID ou GUID sont positionnés (passwd)

Système d'exploitation

└ Espace Disque

└ appels système

└ ruid, euid

Qui suis-je ? Quels droits ai-je sur les fichiers ?

- real uid/gid - qui nous sommes et notre groupe
- effective uid/gid - détermine nos droits d'accès aux fichiers
- normalement ces deux valeurs sont identiques
- elle peuvent différer lorsqu'on exécute un programme dont les droits SUID ou GUID sont positionnés (passwd)

L'utilisation de programmes SUID root est à éviter un maximum.

De nombreuses failles de sécurité existent.

Pour chercher les fichiers qui ont une permission SUID :

>find / -perm -04000

les appels système getuid et geteuid permettent d'obtenir ces deux valeurs

fonctions - répertoires

opendir(3), readdir(3), closedir(3)

```
DIR* opendir(chemin)
struct dirent * readdir(DIR * dir) // NULL à la fin
int closedir(DIR * dir)
```

struct dirent contient notamment

- ▶ le numéro d'inode
- ▶ le nom du fichier

... writedir ?

fonctions - répertoires

```
DIR *dp=opendir(nomdir)
struct dirent * dirp=readdir(dp)
closedir (dp)
```

Que fait donc un ls ?

Système d'exploitation

- └ Espace Disque
 - └ appels système
 - └ fonctions - répertoires

```
DIR *diropenendir(char*)  
struct dirent * dirreadendir(DIR*)  
closedir(DIR*)
```

Que fait donc un ls ?

Pour un ls récursif, songez à construire le chemin. par concaténation de chaînes.

exemple - readdir et stat

```
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <stdlib.h>
int main()
{ struct dirent *dirp; DIR *dp; struct stat inode;
  chdir("/home/jcj");
  dp=opendir(".");
  while ((dirp=readdir(dp))!=NULL)
  { printf ("%8d_ %s",dirp->d_ino,dirp->d_name);
    stat (dirp->d_name,&inode);
    if (S_ISDIR(inode.st_mode))
      printf ("est un repertoire\n");
    else printf ("\n");
  }
  closedir (dp);
  exit (0);
}
```

read

```
n=read(fd,&buff_in,count)
```

- ▶ fd indice dans la table des handle
- ▶ buff_in adresse en RAM pour accueil des bytes lus
- ▶ count nombre de bytes à lire
- ▶ n nombre de bytes réellement lus

quelle est l'utilité de n ?

Système d'exploitation

└ Espace Disque

└ appels système

└ read

```
int read(fd, &buff, n, count)
```

- fd indice dans la table des handle
- buff_in adresse en RAM pour accueil des bytes lus
- count nombre de bytes à lire
- n nombre de bytes réellement lus

quelle est l'utilité de n ?

Cas où $n < \text{count}$: "fin de fichier" ou lecture depuis terminal
L'offset dans la TDFO est mis à jour

write, sync

```
n=write(h,&buff_out,count)
```

L'écriture est asynchrone (cache)

```
sync()
```

- ▶ sync vide le tampon du cache sur le disque
- ▶ synchronise inodes - tampon - disque
- ▶ à utiliser à bon escient

Système d'exploitation

└ Espace Disque

└ appels système

└ write, sync

write, sync

```
int write(&n,&buff_out,count)
```

L'écriture est asynchrone (cache)

```
sync()
```

- sync vide le tampon du cache sur le disque
- synchronise inodes - tampon - disque
- à utiliser à bon escient

abuser de la synchronisation reviendrait à perdre l'avantage du système de mémoire cache

application : une commande filtre

Voici comment on écrirait **cat** : la plus simple des **commandes filtre** :

```
int main()
{
    char c;
    while ((read (0,&c,1)) > 0)
        write (1,&c,1);
}
```

Système d'exploitation

- └ Espace Disque

- └ appels système

- └ application : une commande filtre

Voici comment on écrirait **cat** : la plus simple des commandes filtre :

```
int main()
{
    char c;
    while ((read (0,&c,1)) > 0)
        write (1,&c,1);
}
```

La notion de commande filtre est importante dans les systèmes unix. Ces commandes très simples (KISS) contribuent à construire des outils d'administration bien utiles.

lseek

```
p=lseek(h,déplacement, origine )
```

- ▶ Modifie l'offset dans la TDFO
- ▶ origine = SEEK_SET, SEEK_CUR, SEEK_END
- ▶ p=lseek(h,10,SEEK_SET) // 10 bytes après le début
- ▶ p=lseek(h,0,SEEK_CUR) //intérêt ?

Permet de déplacer l'offset au delà de la fin du fichier !

Comment ajouter des données à la fin d'un fichier ?

exemple - lseek

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>
main(){
    int h;
    char buff [5];
    h=open("alphabet.dat",O_WRONLY | O_CREAT /*/ O_TRUNC*/ ,0666);
    write(h,"abcdefghijklmnopqrstuvwxy",26);
    close(h);
    h=open("alphabet.dat",O_RDONLY);
    lseek(h, -15, SEEK_END);
    read(h, buff, 3); // lmn
    write(1, buff, 3); // lmn -> stdout
    read(h, buff, 3); // opq
    write(1, buff, 3); // opq -> stdout
    close(h);
}
```

exemple - lseek

```
main()
{ int h,p;  char buff [5];
  h=open("alphabet.dat",O_WRONLY | O_CREAT | O_TRUNC,0666);
  write(h,"abcdefghijklmnopqrstuvwxy",26);
  close(h);
  h=open("alphabet.dat",O_RDWR);
  lseek(h,3,SEEK_SET); read(h,buff,5); // defgh
  p=lseek(h,0,SEEK_CUR); // p=8
  write(h,"01234",5); // replace ijklm
  lseek(h,-6,SEEK_END); read(h,buff,5); // uvwxy
  write(1,buff,5); // uvwxy
  close(h);
  printf("\n%d\n",p); // 8
  exit(0);
}
```

```
>cat alphabet.dat
abcdefghijklmnopqrstuvwxy
```

application : créer un fichier creux

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>
main()
{ int h,p;  char buff [5];
  h=open("creux.dat",O_WRONLY | O_CREAT | O_TRUNC,0666);
  write(h,"abcdefghijklmnopqrstuvwxy",26);
  lseek(h,10000,SEEK_CUR);
  write(h,"0123456789",10);
  close(h);
  exit(0);
}
```

Les commandes ls et du (disk usage) nous permettent d'examiner le fichier créé.

application : créer un fichier creux

```
> ls -l creux.dat  
-rw-r--r-- 1 user0 users 10036 29 janv. 13:49 creux.dat  
> du -h creux.dat  
8,0K   creux.dat
```

Le fichier creux.dat a une taille de 10036 et occupe réellement deux blocs sur le disque (ici les blocs font 4Kib)

dup, dup2

```
f2=dup(f1)  
f2=dup2(f1, f)
```

- ▶ duplication d'un handle
- ▶ les handle aux indices f1 et f sont identiques
- ▶ partage du descripteur (et donc de l'offset dans le fichier)

dup et dup2 sont utilisés par le shell pour la programmation des redirections et des pipes

Système d'exploitation

└ Espace Disque

└ appels système

└ dup, dup2

```
f2=dup(f1)  
f2=dup2(f1, f)
```

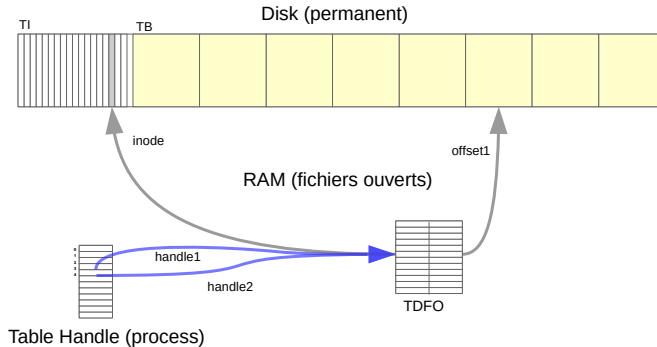
- duplication d'un handle
- les handle aux indices f1 et f sont identiques
- partage du descripteur (et donc de l'offset dans le fichier)

dup et dup2 sont utilisés par le shell pour la programmation des redirections et des pipes

f est préalablement fermé si nécessaire

après un dup

```
f1=open (f, flags , mode);  
f2 = dup(f1)
```







exemple - redirection

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
main()
{ int h;
  char buff [5];
  h=open("alphabet.dat",O_WRONLY | O_CREAT,0666);
  write(h,"abcdefghijklmnopqrstuvwxyz",26);
  close(h);
  h=open("alphabet.dat",O_RDWR);
  dup2(h,1);
  close(h);
  write(1,"ABCDE",5);
  exit(0);
}
```

```
> cat alphabet.dat
ABCDEFghijklmnopqrstuvwxyz
```


Questions ?



-  Modern Operating Systems Fourth edition - Andrew Tanenbaum, Herbert Bos - Pearson Education
-  Advanced Programming in the UNIX Environment Third Edition - W.Richard Stevens, Stephen A. Rago - Addison Wesley (2014)
-  Programmation Système en C sous Linux 2ième édition - Christophe Blaess - Eyrolles (2005)
-  Intel 64 and IA-32 Architectures Software Developer's Manual - December (2011) (pour toutes les images du chapitre mémoire)

remerciements

merci à P.Bettens et M.Codutti pour la mise en page

Crédits

Ces slides sont le support pour la présentation orale des activités d'apprentissage **SYSIR3** et **SYSG4** à HE2B-ÉSI

Crédits

La distribution opensuse
du système d'exploitation **GNU Linux**.

LaTeX/Beamer comme système d'édition.

GNU make, rubber, pdfnup, ... pour les petites tâches.

Images et icônes

deviantart, flickr, The Noun Project 