Système d'exploitation
Le noyau Linux

Christophe Garion

ISAE/DMIA - SUPAERO/IN
10 avenue Édouard Belin
31055 Toulouse Cedex 4

Plan

Système d'exploitation

Un système d'exploitation : pourquoi faire ?

- une abstraction des capacités matérielles des machines : notion de machine virtuelle ;
- un gestionnaire de ressources : CPU, mémoire, périphérique ;
- en plus : gestion des systèmes de fichiers, du réseau, de la sécurité etc.

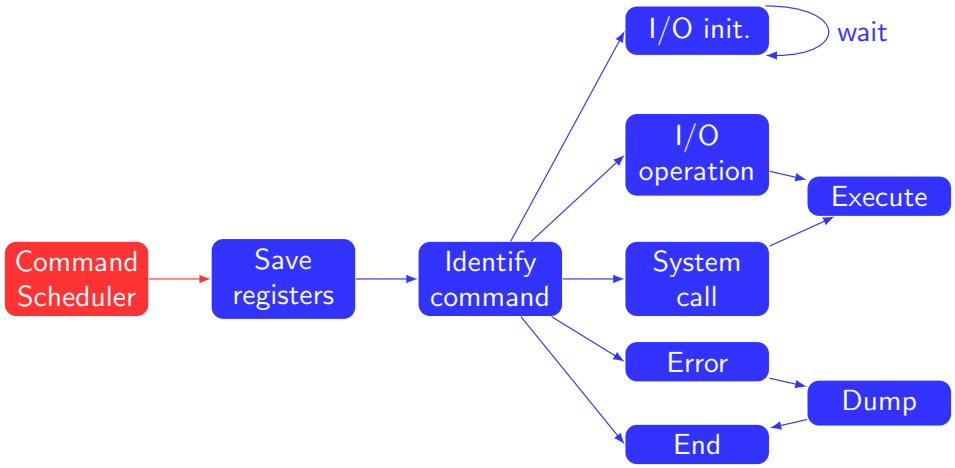
Partition de la mémoire vive :

- espace noyau
- espace utilisateur

Commandes :

- interruptions
- exceptions
- « services »

Le noyau de base : une machine à état



Fonctions majeures d'un noyau

- gestion des processus
  - création, suppression, suspension, reprise, intercommunication (IPC)
- gestion de la mémoire
  - allocation et libération de la mémoire physique
  - accorder des capacités mémoires aux processus
- gestion des I/O
  - pilotage des périphériques
  - standardisation des interfaces via les API
- gestion des mémoires de stockage
  - allouer et libérer les blocs mémoire
- gestion de l'abstraction du stockage
  - essentiellement : le système de fichier (gestion, droits d'accès etc.)
- services réseaux
  - implantation de protocoles (TCP/IP, PPP etc)
  - services (serveur web etc.)
- interface *user space*
  - échange d'information contrôles/commandes
  - commandes complexes (CLI)
  - quotas etc.

Pourquoi différents types de noyaux ?

Le noyau a accès :

- aux ressources matérielles
- à des fonctionnements privilégiés du processeur
  - accès aux registres
  - modification de la gestion des interruptions
  - exemples : *supervisor mode* sur MC86xx, modes protégés sur Intel etc.

Différences entre les différentes architectures :

- qu'exécute-t-on en *kernel mode* ?
- fonctionnalités supplémentaires : *kernel space*, *user space* ?
- monolithique ou pas ?

Noyau monolithique

Noyau monolithique : un seul fichier exécutable en *kernel space* fournissant une interface virtuelle.

Code intégré :

- modules difficiles à concevoir, developper, tester
- bug → crash du système
- très efficace (pas de latence due à la communication inter-processus)
- portabilité « mauvaise »
- exploitation spécialisée (embarqué par exemple)
- encombrement mémoire

Quelques exemples : anciens BSD, Linux avant 1.2, OS 360.

Micro-noyau

Le noyau est toujours un fichier unique, mais seul le code du noyau est exécuté en *kernel space*.

Mécanisme de délégation de droits pour les composants en *user space* pour des traitements spécifiques.

Architecture orientée **services**.

Approche

On ne fait pas confiance au code (sécurité, certification), on le met en *user space*.

Micro-noyau

Caractéristiques :

- indépendance de la plate-forme
- encombrement faible
- cloisonnement des composants : robuste, fiable, sécurisé
- HAL : *Hardware Abstraction Layer*
- le reste de l'OS utilise le HAL
- mais problèmes de performance (modèle deux-tiers pour accéder au matériel)

Quelques exemples : Mach, Windows, Mac OS X

Noyau hybride ou modulaire

Evolution des noyaux monolithiques :

- un seul fichier avec le code s'exécutant en *kernel space*
- autres fichiers pouvant être liés dynamiquement ou statiquement au noyau

Pourquoi ?

Encombrement mémoire (tous les pilotes sont dans le même exécutable dans un noyau monolithique)

Dans une architecture hybride, le noyau contient :

- le strict nécessaire pour démarrer
- le strict nécessaire pour s'initialiser
- peut charger des modules si besoin est (*Loadable Kernel Module*)
  - on charge un module quand on en a besoin
  - on le décharge ensuite

Noyau hybride ou modulaire

Avantages et inconvénients :

- les modules sont en *kernel space*
- temps CPU nécessaire pour charger/décharger les modules
- sécurité

Un exemple : GNU/Linux

Exo-kernels ou code-morphing

Combinaison des micro noyaux et des approches hybrides.

Principe

Redescendre les applications au niveau matériel (contraire du HAL).

Caractéristiques :

- noyau lié intimement au matériel (on ne « peut » pas les dissocier)
- moins de couches d'abstraction
- maximisation des performances
- le développeur prend des décisions sur la « gestion » du matériel
- fiabilité
- performance : appel de fonctions

Développements au MIT (*Parallel and Distributed Operating Systems group*) par exemple.

Historique de Linux

Quelques dates clés :

- 1969 : Unix, laboratoire Bell AT&T (Ritchie et Thompson)
- 1984 : MIT AI Lab, projet GNU avec le noyau Hurd
- 1987 : Vrije Universiteit Amsterdam, Minix (A. Tanenbaum)
- 1991 : Helsinki, Linus Torvald
- 1994 : version 1.0
  - structure générale du noyau
  - fonctionnalités principales
- 1996 : version 2.0
  - support d'autres architectures, SMP
  - périphériques : son, multimédia etc.
  - réseai
- 2001 : version 2.4
  - gestion de la mémoire
  - périphériques : USB, IEEE 1394 etc.
- 2004 : version 2.6
  - ordonnanceur, mémoire virtuelle
  - périphériques
- coût estimé (UE sur Debian) : 882 Meuros

Quelques rappels sur le fonctionnement d'un OS Unix

Deux ressources offertes :

- flots d'instructions
- les fichiers

Pour les ressources d'exécution :

- *threads* : exécution sur le processeur
- changement de contexte lors du changement de *thread*
  - noyau (préemptif)
  - application (coopératif)
- ordonnancement et synchronisation
- espace d'adressage pour chaque *thread* (gestion par la MMU) : adresses physiques et effectives

Définition (processus)

Ensemble de threads + espace d'adressage

Rappels sur le fonctionnement d'un OS Unix

Fichiers :

- fichiers
- périphériques
- sockets

Interface de programmation unique (disques, carte son, socket réseau etc.) :

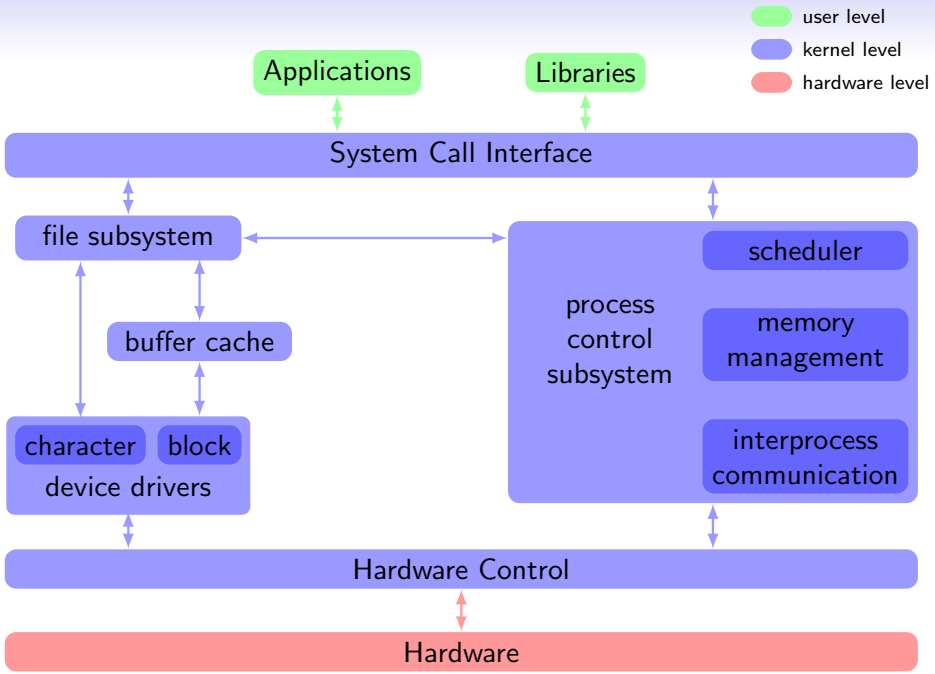
- `read`
- `write`
- `seek`
- `select`
- `stat`

+ primitives spécifiques pour les sockets, répertoires, périphériques etc.

Appel système

Appel aux services du noyau par une application : appels système.

Vue fonctionnelle de l'arch. du noyau Linux



Gestion de la mémoire : mémoire physique

On se place dans le cadre d'utilisation d'un processeur x86 avec le noyau 2.6.27.1

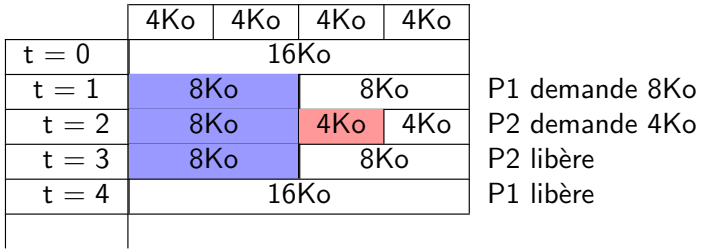
Mémoire physique : la RAM avec des pages physiques de taille de 4Ko sur x86.

Chaque page est représentée par une structure avec compteur de référence :

```
struct page {
        unsigned long flags;            /* Atomic flags, some possibly
                                         * updated asynchronously */
        atomic_t _count;                /* Usage count, see below. */
        union {
                atomic_t _mapcount;     /* Count of ptes mapped in mms,
...
```


Mémoire physique : allocation

Algorithme d'allocation de pages physiques : *buddy allocator*
Principe simple : on découpe la mémoire en puissances de 2.



D. Knuth.
The Art of Computer Programming, volume 1 : Fundamental Algorithms.
Addison-Wesley, third edition, 1997.

Si plus de place : *swap*

Mémoire physique : allocation

Problème du *buddy allocator* : fragmentation.

Pour de plus petites allocations, on utilise le *slab allocator*.

Principes :

- pour des petits objets usuels (descripteurs de fichiers etc.), l'initialisation prend plus de temps que l'allocation/désallocation
- on alloue donc l'espace une fois pour toute pour ces objets avec une initialisation

Principe

Il s'agit en fait d'un cache...

M. Gorman.
Understanding the Linux Virtual Memory Manager.
Prentice Hall, 2004.
Available on http:
//www.csn.ul.ie/~mel/projects/vm/guide/pdf/understand.pdf.

M. Tim Jones.
Anatomy of the linux slab allocator.
http://www.ibm.com/developerworks/linux/library/

Mémoire physique : découpage

Trois zones dans la mémoire :

```
enum zone_type {
        ZONE_DMA,
        ZONE_DMA32,
        ZONE_NORMAL,
        ZONE_HIGHMEM,
        ZONE_MOVABLE,
        __MAX_NR_ZONES
};
```

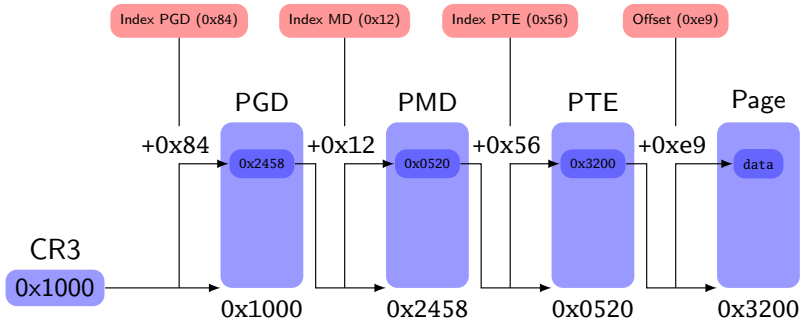
- zone DMA pour l'accès direct aux périphériques
- zone normale (1 Go)
- zone « Highmem » (cf. plus loin)

Gestion de la mémoire : mémoire virtuelle

Chaque processus a un espace d'adressage, adresses sur 32 bits sur x86
➥ 4 Go découpés en pages

Indépendant de l'architecture : système de pagination à trois niveaux

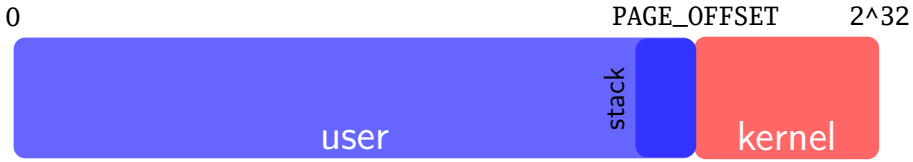
Adresse effective = 0x841256e9



Mémoire virtuelle : découpage

Espace d'adressage découpé en deux parties :

- de 0 à 3 Go (`PAGE_OFFSET` sous x86 !) : reservée au code, données du programme, bibliothèques partagées, tas, pile (*user space*)
- de 3 Go à 4Go : reservée au noyau, c'est la même pour tous les espaces d'adressage
  - représentation directe avec la mémoire physique
  - utilisation de Highmem pour la mémoire physique au delà de 1 Go (mais perte de performances...)



Régions virtuelles

Notion de région virtuelle : portion contiguë de l'*user space*

- en cas de manque de mémoire physique on peut la sauvegarder
  - dans un fichier (projection d'un fichier)
  - mapping anonyme (swap)
- modification en écriture :
  - region `MAP_SHARED` : partagée par d'autres processus
  - region `MAP_PRIVATE` : modification après copie (COW)

Régions virtuelles d'une application

Au démarrage d'une application, le processus posséde les régions virtuelles :

- du code exécutable
- des données du programme
- du tas
- de sa pile
- du code et des données des bibliothèques utilisées

Défauts de page

Problèmes de défauts de page :

- on ne trouve pas la page physique correspondant à une adresse virtuelle
- page n'appartenant à aucune région : SIGSEGV
- page en lecture seule : SIGSEGV

Exemple :

```
int main() {
    int *not_allocated = (int *) 100;
    int test = *not_allocated;
    return 0;
}
```


Processus sous Linux

Définition (processus)

Ensemble de threads + espace d'adressage

Caractéristiques :

- numéro d'indentifiant sur 16 bits (pid_t dans <sys/types.h>, getpid(), getppid())
- processus parent
- commande ps dans le terminal
- tuer un processus : kill -KILL pid (terminal), kill (child_pid, SIGTERM) (prog.)
- attention aux processus zombies : ressources non libérées (à la charge du parent !)

Création et destruction de processus

Trois appels système pour la création :

- fork : créé un nouveau processus dans un nouvel espace d'adressage (copie de l'original), mêmes descripteurs de fichiers ;

  Attention : l'exécution reprend au fork pour le père et le fils !
- clone : créé un nouveau processus en précisant quels sont les éléments partagés : espace d'adressage, descripteurs de fichiers etc.
- exec : pour réinitialiser l'espace d'adressage courant.
  - execvp et execlp : programme dans le PATH
  - execv, execvp et execve : arguments sous forme d'un tableau de pointeurs vers des chaînes (fini par NULL)
  - execve et execle : environnement en plus

Destruction via exit.

Création avec fork

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

int main ()
{
  pid_t child_pid;
  printf ("ID de processus du programme principal : %d\n",(int)getpid ());
  child_pid = fork ();
  if (child_pid != 0) {
    printf ("je suis le processus parent, ID : %d\n", (int) getpid ());
    printf ("identifiant du processus fils : %d\n", (int) child_pid);
  }
  else
    printf ("je suis le processus fils, ID : %d\n", (int) getpid ());
  return 0;
}
```

Utilisation de exec

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

int main ()
{
  pid_t child_pid;
  child_pid = fork ();
  if (child_pid != 0) {
      return child_pid;
  }
  else {
      execvp ("ls", NULL);
      /* On ne sort de la fonction execvp uniquement si une erreur survient. */
     fprintf (stderr, "une erreur est survenue au sein de execvp\n");
     abort ();
  }
}
```


Ordonnancement

Changement de contexte de t_1 vers t_2 :

- sauvegarde de l'état du processeur pour t_1
- restauration de l'état du processeur pour t_2
- en particulier, pointeur de pile restauré

Ordonnanceur du noyau 2.4 (en $O(n)$) :

- liste globale des tâches (même dans un SMP)
- calcul d'un score de *goodness*
  - combien de clics d'horloge le *thread* a laissé par rapport à ce que l'on lui avait donné
  - affinité vers un CPU (système multi-processeurs)
  - utilisation de `nice`
  - tâche temps réel

Ordonnancement

Ordonnanceur du noyau 2.6 (en $O(1)$!) :

- créé par Ingo Molnar
- les raisons : performances sur les JVM (beaucoup de *threads*) et les systèmes multi-processeurs
- une liste par processeur (pour les SMP)
- préemption possible : l'ordonnanceur peut « décaler » une tâche de priorité moins importante
- prioritarisation de tâches dynamique (exemple de *thread* attendant sur des I/O)
- *load-balancing* sur les SMP

Synchronisation dans le noyau

Opérations atomiques (pas d'interruption) :

- opérations sur les bits : changement de valeur, test
- des opérations sur des variables de certains types (entiers par exemple) : incrément, décrément, addition, soustraction, test

Spinlocks : permet de désactiver les interruptions matérielles, fonctionnent sur du SMP

➥ pour du code avec temps d'exécution très faible

Sémaphores : primitives de synchronisation

- on peut réaliser des opérations bloquantes
- thread ou processus demandant une ressource non disponible placé en attente

  ➥ il peut être réveillé si elle devient disponible

Synchronisation et communication au niveau utilisateur

Ensemble d'appels système (API IPC System V) :

- `semget`, `semctl`, `semop`
- file de message : `msgget`, `msgctl`, `msgsnd`, `msgrcv`
- fichier (avec *pipe* ou FIFO)
- signaux `SIGUSR1` et `SIGUSR2`
- mémoire partagée avec `mmap`, `shmap`, `shmget` et `shmctl`

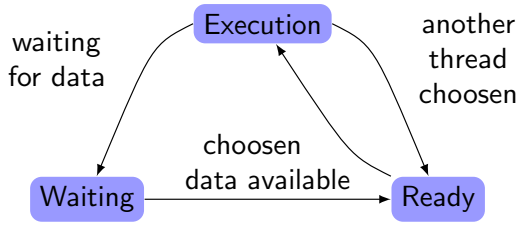
Threads

On peut avoir besoin à l'intérieur d'un même processus d'exécuter plusieurs fils d'exécution (pour partager un espace d'adressage par exemple).

Définition (thread)

Un thread est un fil d'exécution. Il possède :

- *un compteur ordinal*
- *des registres pour les variables en cours*
- *une pile*
- *un état*



Threads noyau

Thread noyau : fonctionne dans n'importe quel espace d'adressage

- réside en espace noyau **seulement**
- créés par le noyau
- traitement interne
- `keventd`, `kswapd`,...

Création :

```
int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
```


Threads utilisateur

Thread utilisateur représenté par une structure :

- état du thread (en exécution, en attente, prêt, ...)
- espace d'adressage d'appartenance
- descripteurs de fichiers ouverts
- gestionnaires de signaux etc.

API de threading standard POSIX.

Création avec `pthread_create` :

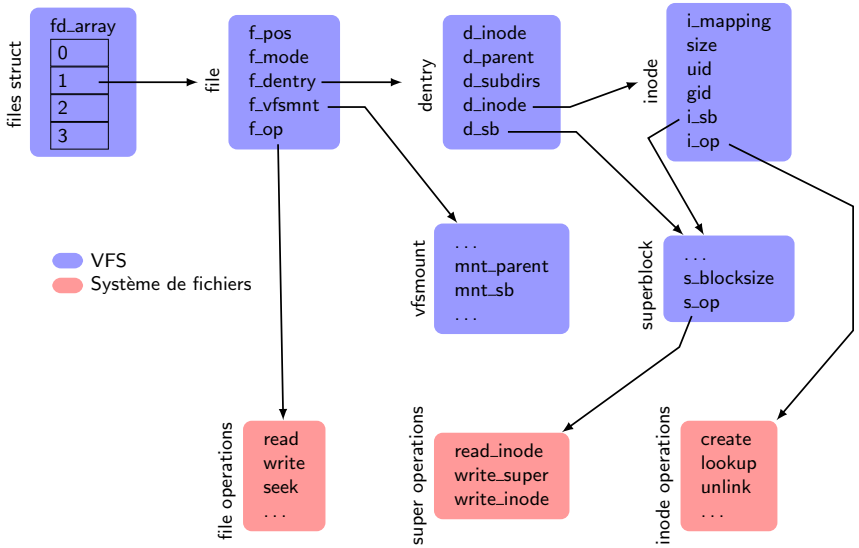
1. identifiant de thread : `*pthread_t`
2. attributs de thread : `*pthread_attr_t` (en particulier pour l'annulation des threads)
3. fonction du thread : `void* (*) (void*)`
4. argument de la fonction : `void*`

Threads utilisateur : exemple de création

```
#include <pthread.h>
#include <stdio.h>
/* Affiche des x sur stderr.  Parametre inutilise. Ne finit jamais. */
void* print_xs (void* unused)
{
  while (1)
    fputc ('x', stderr);
  return NULL;
}

int main ()
{
  pthread_t thread_id;
  /* Cree un nouveau thread. Le nouveau thread executera la fonction
     print_xs. */
  pthread_create (&thread_id, NULL, &print_xs, NULL);
  /* Affiche des o en continue sur stderr. */
  while (1)
    fputc ('o', stderr);
  return 0;
}
```


Architecture de base : le VFS



Architecture de base des fichiers : le VFS

VFS (*Virtual File System*) :

- abstractions, opérations communes à tous les systèmes de fichiers
- cohérence de l'espace de nommage (points de montage)
- API unique pour l'espace utilisateur
- intégration avec les autres sous-systèmes

```
int inf, outf;
char buf[512];

inf = open("/media/usbstick/test.txt");
outf = open("/tmp/try.txt");

do {
  i = read(inf, buf, sizeof(buf));
  write(outf, buf, i);
}

close(outf);
close(inf);
```


VFS : quelques définitions

4 structures importantes :

- `superblock` : information sur un système monté
- `inode` : information sur un fichier
- `file` : **interaction** entre un fichier ouvert et un processus
- `dentry` : information sur les entrées de répertoires

ext3fs : un exemple sur disque dur

Définition (bloc)

Bloc logique regroupant 2, 4 ou 8 blocs physiques de 512 octets.

Définition (inode)

Informations sur un fichier codées sur 128 octets.

Chaque partition est découpée en blocs identiques regroupés en **groupes de blocs** que l'on indexe :

- super-bloc (identique dans tous les groupes)
- descripteurs de groupe (identique dans tous les groupes)
- table *bitmap* d'allocation des blocs du groupe
- table *bitmap* d'allocation des inodes du groupe
- table des inodes du groupe
- blocs de données

ext3fs : le super-bloc

Il contient l'ensemble des données caractérisant la structure et l'état du système de fichiers.
Par exemple (avec `debugfs`) :

```
debugfs 1.40-WIP (02-Oct-2006)
debugfs :  open /dev/hda8
statsdebugfs :  stats
...
Filesystem OS type :       Linux
Inode count :              1909440
Block count :              3813760
Reserved block count :     190688
Free blocks :              635901
Free inodes :              1844300
First block :              0
Block size :               4096
Fragment size :            4096
Blocks per group :         32768
Fragments per group :      32768
Inodes per group :         16320
Inode blocks per group :   510
Last mount time :          Tue Nov  7 08 :46 :32 2006
Last write time :          Tue Nov  7 19 :54 :24 2006
Mount count :              23
Maximum mount count :      30
Last checked :             Fri Jun 16 07 :26 :33 2006
```


ext3fs : table d'inodes

Tout fichier est associé à un inode unique.
Chaque inode a un numéro unique (1 à 10 réservés)

Exemple avec `ls -ila` :

```
total 76K
      2 drwxrwsr-x   6 root  staff     4,0K 2006-01-14 22 :50 ./
      2 drwxr-xr-x  23 root  root      4,0K 2006-09-02 17 :29 ../
 718180 drwxr-xr-x   2 ftp   nogroup   4,0K 2006-01-14 22 :50 ftp/
     11 drwxr-xr-x   2 root  root       48K 2004-10-14 11 :36 lost+found/
 538565 -rw-r-----   1 tof   staff      207 2005-03-23 09 :12 semantic.cache
 277441 drwxrwx---   3 spamd spamd     4,0K 2005-01-07 16 :03 spamd/
1468801 drwx--x--- 169 tof   www-users 8,0K 2006-11-08 01 :05 tof/
```

Chaque inode :

- comprend des informations sur le fichier (type, propriétaire etc.)
- pointeurs vers les blocs de données (contenu des fichiers)

Allocation : si possible dans le même groupe de blocs (fragmentation).

ext3fs : un exemple d'inode

```
[root@pctof]/home # ls -ilap tof/svn.txt
1469055 -rw-r----- 1 tof staff 8,2K 2006-10-26 15 :50 tof/svn.txt

[root@pctof]/home # debugfs
debugfs 1.40-WIP (02-Oct-2006)
debugfs :  open /dev/hda8
debugfs :  stat <1469055>
Inode : 1469055   Type : regular    Mode :  0640   Flags : 0x0   Generation : 2073991(
User :  1000   Group :    50   Size : 8396
File ACL : 0    Directory ACL : 0
Links : 1   Blockcount : 24
Fragment :  Address : 0    Number : 0    Size : 0
ctime : 0x4540bd15 -- Thu Oct 26 15 :50 :13 2006
atime : 0x4540bd15 -- Thu Oct 26 15 :50 :13 2006
mtime : 0x4540bd15 -- Thu Oct 26 15 :50 :13 2006
BLOCKS :
(0) :2978551, (1) :3020812, (2) :3021043
TOTAL : 3
```

Pilotes de périphériques

Trois types :

- caractère (port imprimante, souris, console)
- bloc (disques durs, CD Rom)
- réseau

Échanges de données :

- ports d'I/O (sur x86) : zone de dialogue avec le périphérique
- projection dans l'espace des adresses physiques
- DMA (pas d'utilisation du processeur)

Interruptions :

- processeur (division par zéro etc.) : exception
- matériel (fin traitement DMA, entrée clavier etc) : IRQ
- appel système

Où trouver tout cela dans les sources ?

`kernel` : *System call interface subsystem* (*process scheduler*, *timing*, appels système)

- planification (`sched.c`) : primitives `schedule`, `sleep on`, `wakeup`
- gestion des I/O (`resource.c`)
- *timing* (`timer.c`, `itimer.c`) : `time`, ...
- `fork` (`fork.c`)
- chargement dynamique des modules (`ksyms.c` et `module.c`)
- messages de *warning* ou d'erreur (`panic.c`), erreurs I/O (`printk.c`)
- fin d'appel (`exit.c`)
- signaux (`signal.c`) : `signal`, `sigaction`, ...

`mm` : *memory management subsystem* (gestion du *swap*, de la table des pages, segmentation)

Où trouver tout cela dans les sources ?

`ipc` : *inter-process communication subsystem*

- *shared memory* (`shm.c`) : disposer de blocs de mémoire communs entre processus
- sémaphores (`sem.c`)
- passage de message (`msg.c`)
- droits et permission (`util.c`)

`fs` : *file subsystem* (systèmes de fichiers, gestion du cache)

- cache (`buffer.c`, `dcache.c`, `device.c`, `block_dev.c`)
- format binaires exécutables
- appels systèmes open (`open.c`), `close` (`close.c`), lire/écrire (`read_write.c`), monter (`super.c`)
- gestion des inodes

Où trouver tout cela dans les sources ?

`net` : *networking subsystem* (protocoles etc.)

- IPv4, IPv6, AppleTalk (`protocols.c`)
- pare-feu
- sockets (`socket.c`)

`drivers` : *device drivers subsystem*

- le plus gros
- regroupés par sous-systèmes

`arch` : parties spécifiques aux différentes architectures

`Documentation` : la documentation

Remarque

Ne pas oublier `man` pour la documentation !

Bibliographie

Kernel trap.
http://www.kerneltrap.org.

The linux kernel archives.
http://www.kernel.org.

D. P. Bovet and M. Cesati.
Understanding the Linux kernel.
O'Reilly Media, 3rd edition, 2005.

M. Gorman.
Understanding the Linux Virtual Memory Manager.
Prentice Hall, 2004.
Available on http://www.csn.ul.ie/~mel/projects/vm/guide/pdf/understand.pdf.

A. Tanenbaum.
Systèmes d'exploitation.
Dunod, 2nd edition, 2003.
In French.