Concepts fondamentaux et structure du noyau Linux

Thomas Petazzoni et David Decotigny

Résumé

Cet article a pour but de démystifier la structure et le fonctionnement d'un noyau de système d'exploitation de type Unix, en présentant les éléments essentiels du cœur de Linux.

Introduction

Du point de vue de l'utilisateur, le système Unix repose sur un nombre de notions fondamentales extrêmement limité. C'est d'ailleurs grâce à la définition particulièrement bien réfléchie de celles-ci que réside toute l'élégance et même la puissance de ce système. Historiquement, le système Unix de base ne présente en effet aux applications utilisateur que deux concepts : celui de *processus*, et celui de *fichier*. Linux est un noyau inspiré de cette famille de systèmes.

Nous allons voir quels sont les concepts fondamentaux sous-jacents, que l'on retrouve sur beaucoup de systèmes d'exploitation. Nous indiquerons ensuite sous quelle forme ces grands concepts se retrouvent dans le noyau Linux.

1 Concepts fondamentaux

Un noyau de type Unix offre aux programmes deux types de ressources : les ressources d'exécution (flots d'instructions, tels que `c = f(a + b);`), et les autres par l'intermédiaire d'une abstraction unique : les fichiers. Nous allons commencer par présenter les concepts liés à la gestion de ces deux grandes catégories de ressources. Nous terminerons en indiquant brièvement ce qu'est un programme utilisateur et comment il interagit avec le noyau.

1.1 Ressources d'exécution

Pour exécuter un programme utilisateur, les acteurs centraux sont les instructions et les données manipulées par ces instructions, pris en charge par deux composants fondamentaux : le processeur et la mémoire. Nous présentons ici le cas général des systèmes d'exploitation, sans nous concentrer sur Unix en particulier.

1.1.1 *Threads*

Lorsqu'on veut donner l'illusion d'une exécution en parallèle de plusieurs applications, on doit gérer plusieurs flots d'instructions, sachant que, à chaque instant, seulement un de ces flots sera exécuté sur chaque processeur. Traditionnellement, on appelle de tels flots des *threads*, et on parle de système multitâche. Historiquement, sous Unix, cette notion de *thread* est apparue plus tard que celle de *processus* (voir la section 1.1.3) : il n'y avait pas de distinction claire entre les deux. Dans toute la suite, nous utilisons ce terme de *thread* pour désigner un flot d'instructions, indépendamment de toute bibliothèque dite "de threads" (le programme `int main() { return 0; }` contient un flot d'instructions, et donc un *thread*).

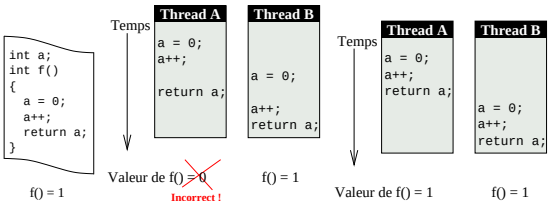
Pour passer d'un *thread* à un autre, un *changement de contexte* est effectué soit à l'initiative du noyau (multitâche préemptif), soit à l'initiative de l'application (multitâche coopératif). Il consiste à *photographier* l'état du premier *thread*, et à restaurer l'état du second *thread* tel qu'il avait précédemment été photographié.

1.1.2 Ordonnancement et synchronisation

La partie du noyau qui s'occupe de gérer ces ressources *threads* est l'*ordonnanceur* : c'est lui qui décide vers quel *thread prêt* effectuer chaque changement de contexte. L'état d'un *thread* (*prêt* ou *bloqué*) dépend des ressources qu'il demande et des ressources disponibles dans le système.

Les sources de blocage peuvent être liées à des ressources matérielles (correspondre à une réalité physique tangible, comme un disque), logiques (zones de mémoire, fichiers, messages réseau, ...), ou purement virtuelles, auquel cas on parle de *synchronisations*. Dans ce dernier cas, un *thread* peut être bloqué en attente qu'un autre *thread* lui signale un événement particulier (il n'y a pas de *vraie* ressource là-dedans). En général, quand on parle de blocage sur une ressource logique ou matérielle, il s'agit en fait d'un blocage sur une synchronisation sous-jacente, qui sert à *protéger* la ressource logique ou matérielle, comme le montre le scénario de la figure 1 : deux *threads* identiques exécutent la même fonction `f()` qui devrait logiquement renvoyer 1 (figure 1 (a)) ; sans synchronisation, leurs exécutions peuvent s'entrelacer de façon incorrecte (figure 1 (b)) ; avec synchronisation, l'objectif serait de se ramener à un scénario sans entrelacement comme celui de la figure 1 (c).

Les synchronisations les plus courantes sont les *mutex* (exclusion mutuelle), les *sémaphores* (protection de ressources constituées d'une quantité connue d'unités, comme des files de messages), les *conditions* (un *thread* se met en sommeil en attendant que la condition soit



(a) Code des *threads*

(b) Entrelacement invalide

(c) Entrelacement correct

FIG. 1 – Entrelacements d'exécutions

validée, ou *signalée* par un autre *thread*), ou encore les *files de messages* (analogie avec un *pipe* Unix).

1.1.3 Espace d'adressage et mémoire virtuelle

Le processeur lit les instructions des *threads* à exécuter en mémoire, et une partie d'entre elles agit sur des données en mémoire (lecture/écriture).

Pour faire du multitâche robuste simplement, on est amené à distinguer deux types d'adresse : les adresses pour accéder aux octets en mémoire physique ("*adresses physiques*"), et les adresses manipulées par le processeur pour ses données et ses instructions ("*adresses effectives*" du processeur relatives à l'"*espace d'adressage*" courant). La traduction `adresses effectives` ⇒ `adresses physiques` est effectuée par un composant particulier, souvent intégré dans le processeur : la MMU (*Memory Management Unit*). Cette traduction est réalisée en fonction de la *configuration* de l'espace d'adressage courant, elle-même stockée en mémoire sous la forme de tables (voir figure 2). Certaines MMU font la traduction dans le sens inverse.

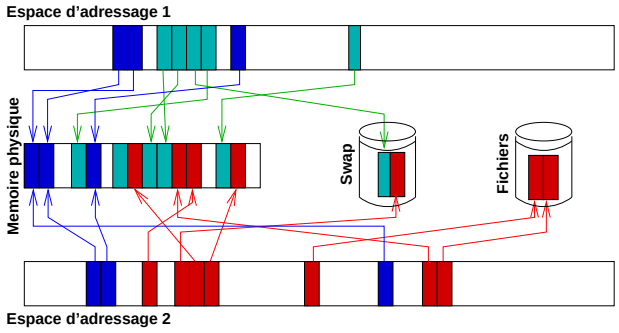


FIG. 2 – Configuration d'espace d'adressage et traduction d'adresse

Ce fonctionnement permet de simplifier le multitâche. D'une part, le chargement est simplifié : les adresses virtuelles de l'application peuvent être fixées indépendamment des adresses physiques, et donc des autres tâches déjà présentes. D'autre part il suffit que chaque application ait une configuration d'espace d'adressage en propre pour garantir le *cloisonnement* des espaces d'adressage, ou la *protection* mémoire des applications. Ce mécanisme n'interdit cependant pas à plusieurs espaces d'adressage de se recouvrir en partie en mémoire physique : il y a alors *partage* possible de zones de mémoire physique.

Enfin, cette approche permet que l'espace d'adressage ait une taille bien supérieure à celle de la mémoire physique installée sur la machine. Il est ainsi possible de spécifier dans les tables de traduction que certaines adresses virtuelles correspondent à une donnée qui se trouve sur disque (dans la zone de *va-et-vient*, ou *swap*, ou dans un fichier), plutôt qu'en mémoire physique (voir figure 2). Ceci confère à la mémoire physique le rôle d'un cache vers l'espace d'adressage. Elle est en effet d'accès plus rapide qu'un disque, mais de taille souvent inférieure. Et lorsque la quantité de mémoire physique disponible devient insuffisante, une partie des informations de cette mémoire est déplacée vers le disque.

L'ensemble de ces mécanismes porte le nom de *gestion de la mémoire virtuelle* (ou *VMM*, pour *Virtual Memory Management*) et repose sur le fonctionnement par *pagination*[1] de la MMU [Tan94, Chapitre 3.3.1].

Une *application* correspond donc à deux données : une configuration de son espace d'adressage, et un ensemble de *threads*. Sous Unix, le terme courant pour désigner ce couple est celui de *processus*.

1.2 Fichiers et espace de nommage

1.2.1 Opérations sur les fichiers

Un fichier est la représentation des ressources du système accessibles à l'utilisateur, qui est à la fois uniforme et simple à manipuler. Parmi ces ressources, on trouve les fichiers "classiques" bien sûr, mais aussi les périphériques matériels, les *sockets* (programmation réseau), etc...

Sous Unix, on accède à un fichier du système par l'intermédiaire d'une interface de programmation unique, principalement formée des fonctions simples de lecture (`read`), d'écriture (`write`), de déplacement de la tête de lecture (`seek`), d'attente d'opérations externes (lecture/écriture par d'autres applications : `select`), et de consultation des informations sur le fichier (taille, date de création : `stat`). Cette interface unique permet de simplifier l'écriture des applications : on peut ainsi utiliser les mêmes fonctions pour accéder à un disque IDE ou SCSI, à une carte son, ou un fichier du type `/etc/passwd`.

Les fichiers d'un système Unix ne supportent pas forcément toutes ces opérations. Ainsi, on distingue les fichiers accessibles en mode *bloc* pour lesquels la fonction `seek` est supportée (on peut se déplacer librement dans tout le fichier), et les autres, dits "mode *caractère*", pour lesquels seules les lectures/écritures séquentielles sont supportées. Par exemple, les fichiers stockés sur

[1] Le fonctionnement par *segmentation*, parfois complémentaire, repose sur les mêmes principes, mais est moins souple et donc peu utilisé.

disque, et le disque lui-même, sont des fichiers accessibles en mode *bloc*. Un *socket* réseau ou une imprimante sont ainsi de type *caractère* seulement.

Inversement, certains fichiers supportent d'autres opérations. Sous Unix, d'autres interfaces étendues ont ainsi été définies, telles que les interfaces pour les *sockets* (`accept`, `bind`, ...), ou celles pour les répertoires (`readdir`, ...). De même, Unix définit des fichiers *spéciaux* qui font le lien direct entre les applications utilisateur et les pilotes de périphériques internes au noyau. Pour ces fichiers spéciaux aussi, une extension à l'interface standard est prévue (fonction `ioctl`), qui permet par exemple de modifier la fréquence d'échantillonnage d'une carte son, ou la résolution d'une carte graphique, etc...

1.2.2 Espace de nommage

Sous Unix, la plupart des fichiers accessibles sont rassemblés dans une arborescence unique globale à tout le système, permettant de les identifier par un nom unique, ou *chemin* (`/home/toto/.bashrc` par exemple) : c'est l'*espace de nommage*. Les fichiers *spéciaux* sont également présents dans cet espace, et sont caractérisés par un moyen d'identifier le pilote associé (le fameux couple de nombres majeur/mineur). Cependant, tous les fichiers n'apparaissent pas nécessairement dans l'espace de nommage, tels les *sockets* réseau.

Sous Unix, l'espace de nommage est unique, mais peut être constitué de plusieurs sous-arborescences, chacune d'entre elles correspondant à un *système de fichiers monté*. En général, un système de fichiers est une organisation particulière des données physiques stockées sur un disque (comme `FAT` ou `ext2`), mais il peut être aussi purement virtuel (comme `/proc`).

1.3 Interaction noyau / applications utilisateur

Dans les systèmes Unix généralistes, il y a cloisonnement entre applications, mais aussi entre les applications et le noyau. À cet effet, celles-ci ne sont pas autorisées à effectuer certaines opérations dites "privilégiées". Tout particulièrement, elles ne sont pas autorisées à modifier leur espace d'adressage pour accéder à l'espace d'adressage d'une autre application. Seul le noyau est autorisé à prendre en charge ces opérations.

Il apparaît par conséquent au moins deux modes d'exécution des instructions qui doivent être supportés par le processeur : le mode privilégié, ou *noyau*, et le mode non privilégié, ou *utilisateur*. Les applications utilisateur fonctionnent en mode *utilisateur* et font appel aux services du noyau au travers d'un changement de mode parfaitement contrôlé : un *appel système*, qui correspond à un appel de fonction avec changement de privilège.

2 Sous-systèmes noyau

Pour ce petit voyage au cœur de Linux, nous sommes partis d'un noyau 2.4.20. Nous resterons dans le cas simple du processeur x86 en uniprocesseur ou en *SMP* (*Symmetric Multi-Processor*).

2.1 Gestion de la mémoire

Le code de gestion de la mémoire se trouve principalement dans les sous-répertoires `mm` et `arch/i386/mm`.

2.1.1 Mémoire physique

La mémoire physique désigne la mémoire physiquement disponible sur l'ordinateur, c'est-à-dire la quantité de mémoire vive, ou *RAM*. Cette mémoire physique est un élément tangible de l'ordinateur ; elle contient physiquement les informations dont le processeur a besoin pour fonctionner, telles que le code du système d'exploitation, le code et les données des programmes, etc...

Pages physiques et allocation. Le système d'exploitation est chargé de gérer cette mémoire physique en permettant son allocation et sa désallocation par *pages* physiques, de taille fixée à 4 Ko sur les plateformes x86. Chacune de ces pages physiques est représentée au sein du système par une structure de type `include/linux/mm.h:struct page`. Cette structure permet notamment de maintenir un compteur de références sur chaque page physique puisqu'une même page physique peut être référencée par plusieurs pages virtuelles ; ce compteur vaut 0 lorsque la page est immédiatement disponible. Le reste de cette structure est directement géré par les sous-systèmes concernés par la page : le système de fichiers ou le système de *swapping*.

La fonction `mm/page_alloc.c:__alloc_pages()` est le cœur de l'allocateur des pages physiques (algorithme de type *buddy system* [PN77]), et fait appel au *thread noyau* `kswapd` (`mm/vmscan.c`) pour *swapper* des pages physiques lorsqu'il n'y a plus assez de pages physiques disponibles [Gor03, Chapitre 12]. Pour les plus petites tailles d'objets à allouer (quelques octets), l'algorithme *slab allocator* [Bon94] (`mm/slab.c`) est implanté au-dessus de cette fonction.

Découpage de la mémoire physique. Pour gérer la mémoire physique, le noyau Linux la découpe en *zones*, représentées par la structure `linux/mmzone.h:zone_t`. Elles sont au nombre de 3 : la zone DMA (environ 16 Mo) pour les transferts directs entre les périphériques et la mémoire (voir la section 2.5.2), la zone en-dessous de 1 Go pour les diverses allocations dynamiques de mémoire, et enfin la zone HIGHMEM pour la mémoire physique au-delà de 1 Go (voir la section 2.1.2).

2.1.2 Mémoire virtuelle

Dans la première partie de cet article, nous avons vu que chaque processus disposait de son propre espace d'adressage. Les adresses de cet espace d'adressage sont codées sur 32 bits sur x86, ce qui correspond à une taille de 4 Go. Il est découpé en pages (fonctionnement par *pagination*). Chacune de ces pages entre ou non en correspondance avec une page physique, en fonction de la configuration des tables de traduction de la MMU.

Traduction d'adresse. Pour rester indépendant de la plate-forme, Linux utilise une pagination à 3 niveaux, comme le présente la figure 3. Ainsi chaque espace d'adressage dispose d'un répertoire de pages global (*PGD*), dont l'adresse est donnée dans un registre (`cr3` sur x86) mis à jour à chaque changement de processus. Chaque entrée de ce répertoire de pages global correspond à un répertoire des pages "du milieu" (*PMD*), dans lequel chaque entrée correspond à une table de pages (*PTE*). Chaque entrée de cette table de pages donne l'adresse de la page physique correspondante.

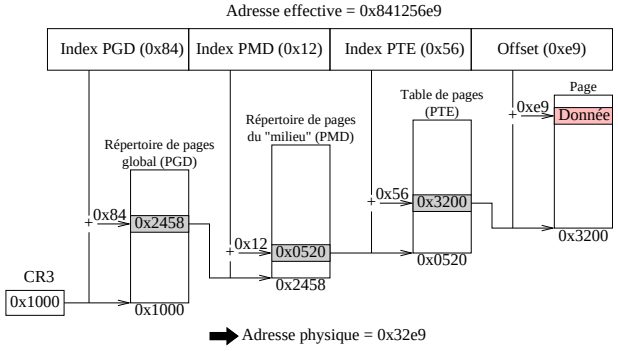


FIG. 3 – Pagination à 3 niveaux

Toutefois, le nombre de niveaux du mécanisme de pagination est dépendant de l'architecture. Par exemple, sur les processeurs x86, la pagination n'est que sur 2 niveaux : répertoire de pages puis tables de pages. Dans ce cas, les répertoires de pages "du milieu" ont une taille de une entrée. Toute la gestion de la mémoire virtuelle de Linux fonctionne comme si la pagination était à 3 niveaux, et la partie dépendante de l'architecture (voir section 2.6) s'occupe de simuler cette pagination à 3 niveaux avec le processeur disponible.

Découpage de l'espace d'adressage. Sous Linux, l'espace d'adressage de chaque processus est découpé en deux parties :

- la première s'étend de 0 à `include/asm/page.h:PAGE_OFFSET`, qui vaut 3 Go sur x86. Cette partie est réservée au code, données du programme et des bibliothèques partagées, ainsi qu'au tas et à la pile. Cette partie diffère donc d'un processus à l'autre, c'est l'*espace utilisateur* de l'espace d'adressage.
- la seconde s'étend de `PAGE_OFFSET` à la fin de la mémoire, soit 4 Go sur x86. Cette partie est réservée au noyau (code et données), et est identique dans tous les espaces d'adressage ; c'est l'*espace noyau* de l'espace d'adressage.

La partie consacrée au noyau est en fait une représentation à l'identique de la mémoire physique : accéder à l'adresse `PAGE_OFFSET + x` en virtuel revient à accéder à l'adresse `x` en physique. On parle d'*identity mapping* de la mémoire physique dans l'espace noyau. Lorsque la quantité de mémoire physique excède la taille de l'espace consacré au noyau (1 Go sur x86), un système appelé HIGHMEM est utilisé pour pouvoir accéder à la mémoire physique située au delà de 1 Go [Gor03, Chapitre 10].

Régions virtuelles. Sous Linux, chaque espace utilisateur est représenté par une structure de type `include/linux/sched.h:mm_struct`. On y trouve notamment la liste des *régions virtuelles*.

Une région virtuelle est une portion contiguë de l'espace utilisateur.

Une première caractéristique d'une région virtuelle est qu'elle peut être sauvegardée sur disque afin de libérer des pages physiques appartenant à la région en cas de besoin. La sauvegarde porte le nom de *backing store* :

- Si ce dernier correspond à un fichier d'un système de fichiers, on parle de *projection* ou *mapping d'un fichier*. Dans ce cas, la région virtuelle reflète le contenu du fichier projeté.
- Sinon on parle de *mapping anonyme*. Dans ce cas, si le système dispose d'un *swap*, le *backing store* de la région est le *swap*.

Une autre caractéristique des régions virtuelles est la manière dont elles se comportent lorsque leur contenu est modifié (écriture en mémoire). Il existe ainsi deux possibilités :

- Si la région est de type `MAP_SHARED` (`man mmap`), toute modification effectuée sur une page de la région est partagée par tous les processus qui ont effectué le même mapping (*mapping* de la même partie du même fichier en `MAP_SHARED`, ou relation père-fils entre deux processus avec `fork()`, voir section 2.2.2). Ce type de mapping permet à deux processus de partager une portion d'espace mémoire.
- Si la région est de type `MAP_PRIVATE`, la première modification effectuée sur une page de la région provoque la copie de la page d'origine. C'est sur cette copie que sont reportées toutes les modifications. Ce mécanisme porte le nom de *Copy On Write*, ou *COW*, et associe la page résultant de la copie à un *mapping* anonyme. Il est par exemple utile lorsque le même programme est lancé deux fois : il suffit de projeter en `MAP_PRIVATE` la partie du fichier exécutable qui contient les données allouées par le compilateur (section `.data` habituellement). Tant qu'aucune modification des

données n'a été effectuée, on est sûr d'économiser de la place en mémoire car la section est partagée par les deux processus. Cependant, grâce au mécanisme *COW*, dès qu'un des deux processus modifie cette région, la modification n'est pas reportée sur l'autre processus.

Sous Linux, les régions sont représentées par la structure `include/linux/mm.h:vm_area_struct`. Cette structure contient les bornes de la région virtuelle, ses droits d'accès (lecture/écriture), l'espace d'adressage d'appartenance, un pointeur vers le fichier *mappé* si il y a lieu (champs `vm_file` et `vm_pgoff`), etc... [Gor03, Chapitre 5.4]

Au chargement d'un programme, le processus possède les régions virtuelles correspondant aux *sections* figurant dans le fichier exécutable (code et données, `man objdump`), à son *tas* initial (mémoire allouée dynamiquement, et gérée par les fonctions `malloc()` et `free()`), à sa pile, ou au code et données des bibliothèques de fonctions partagées. En cours d'exécution, il peut agrandir son tas en utilisant l'appel système `brk()`. La pile d'un processus est quant à elle agrandie automatiquement par le noyau en fonction des besoins du processus, jusqu'à une taille limite personnalisable (`man setrlimit`). Les autres régions peuvent être modifiées par la fonction `mm/mremap.c:mremap()`. D'autres régions peuvent être créées par appel à la fonction `mm/mmap.c:mmap()`.

On peut obtenir la liste des régions virtuelles d'un processus en utilisant le système de fichiers `/proc`. Un `cat /proc/[pid]/maps` (`man proc`) donnera la liste des régions du processus `[pid]`. Exemple pour un processus `cat` :

```
$ cat /proc/self/maps
# address         perms offset  dev   inode      pathname
08048000-0804b000 r-xp 00000000 08:02 52439      /bin/cat
0804b000-0804c000 rw-p 00003000 08:02 52439      /bin/cat
0804c000-0804d000 rwxp 00000000 00:00 0
40000000-40011000 r-xp 00000000 08:02 16236      /lib/ld-2.3.1.so
40011000-40012000 rw-p 00011000 08:02 16236      /lib/ld-2.3.1.so
40026000-4012e000 r-xp 00000000 08:02 16241      /lib/libc-2.3.1.so
4012e000-40134000 rw-p 00107000 08:02 16241      /lib/libc-2.3.1.so
40134000-40137000 rw-p 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
```

La première région est le code du programme exécuté, son adresse de début est indiquée dans l'exécutable, et est en général située à l'adresse virtuelle 0x08048000 sur x86 (`info ld`). Cette région est en lecture et exécution seulement. La deuxième région contient les données du programme exécuté, et est en lecture écriture. La région qui suit est le tas.

Ensuite, on trouve le code et les données de toutes les bibliothèques partagées dont le programme a besoin. La première bibliothèque partagée, `ld-2.3.1.so`, est chargée en premier et dans la première région disponible à partir de l'adresse virtuelle 0x40000000 sur x86 (`include/asm/i386/processor.h:TASK_UNMAPPED_BASE`). Cette bibliothèque est le chargeur de bibliothèques dynamiques : elle va charger en mémoire les autres bibliothèques. Les autres régions sont le code et les données des bibliothèques partagées. La dernière région correspond à la pile.

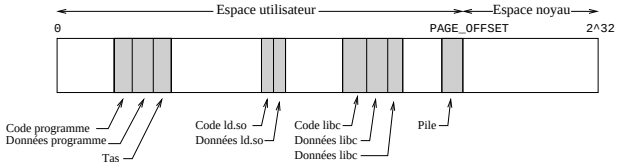


FIG. 4 – Les régions de l'espace d'adressage de `cat`

2.1.3 Mémoire virtuelle et défaut de page

En plus de gérer la création et la destruction des régions virtuelles, de maintenir les tables de correspondance entre adresses virtuelles et physiques, le système de gestion de mémoire virtuelle est responsable de la résolution des défauts de page.

Un défaut de page a lieu lorsque la MMU ne parvient pas à trouver la page physique qui correspond à une adresse virtuelle qui vient d'être accédée. Dans ce cas, une routine du noyau (un *traitant d'exception*) est exécutée pour résoudre cette erreur (fonction `arch/i386/mm/fault.c:do_page_fault()`, puis fonction `mm/memory.c:handle_mm_fault()`, [Gor03, Chapitre 5.6]). Le comportement de cette routine dépend de la page associée à l'adresse virtuelle, ou du type d'accès (lecture/écriture) effectué, comme nous le décrivons ci-dessous. Cette gestion des défauts de page est totalement transparente pour les applications qui tournent dans le système.

Page n'appartenant à aucune région. Le système envoie le signal `SIGSEGV` (le fameux "Segmentation fault") au processus fautif.

Page de type lecture seule accédée en écriture. Si la région est de type `MAP_PRIVATE` en lecture/écriture, le défaut de page est utilisé pour mettre en œuvre le mécanisme de *COW* (voir section 2.1.2). Sinon le système envoie le signal `SIGSEGV` au processus fautif.

Sinon... Une page physique est allouée en mémoire, et le contenu du *backing store* associé (fichier, *swap*) y est copié. Ce mécanisme porte le nom de *demand paging*, et permet de ne rien charger en mémoire physique jusqu'à l'accès effectif aux données.

Un cas particulier se produit lorsque la région est de type *mapping anonyme* ne résultant pas d'un *COW* (tas ou pile par exemple). Dans ce cas, lors du premier accès, le *swap* ne contient encore aucune donnée liée à la page, la page nouvellement allouée est donc remplie de 0.

2.2 Threads, processus et ordonnancement

2.2.1 Threads noyau et threads utilisateur

Il existe sous Linux deux types de *threads* : les threads noyau et les threads utilisateurs.

Les *threads* noyau fonctionnent dans n'importe quel espace d'adressage parce qu'ils résident dans l'espace noyau. Ils sont créés par le noyau en utilisant la fonction

`arch/i386/kernel/process.c:kernel_thread()`, et effectuent, en mode noyau, des traitements internes au noyau. La commande `ps ax` donne la liste des processus, ainsi que la liste des threads noyau, dont le nom est entre crochets. On y trouve par exemple *keventd*, *kswapd*, *ksoftirqd_CPU0*.

Les threads utilisateur ont un espace d'adressage, éventuellement partagé avec d'autres threads utilisateur.

Chaque thread est représenté par une structure de type `include/linux/sched.h:task_struct`. On y trouve notamment l'état du thread (en exécution, prêt, en attente, etc..), l'espace d'adressage d'appartenance, la liste des descripteurs de fichiers ouverts (voir section 2.4), les gestionnaires de signaux (voir section 2.3.2), etc... [Aiv02, Chapitre 2.1].

2.2.2 Création et destruction

La création des threads utilisateur se fait grâce aux appels système `fork()` et `clone()`, qui utilisent tous deux la fonction `kernel/fork.c:do_fork()`.

L'appel système `fork()` crée un nouveau *thread* dans un nouvel espace d'adressage, c'est-à-dire un nouveau processus, en dupliquant le processus appelant. L'espace d'adressage nouvellement créé est une copie de celui de l'appelant, les descripteurs de fichiers ouverts sont également copiés, etc..

L'appel système `clone()` permet de créer un nouveau *thread*, tout en précisant les éléments qu'il partage avec le contexte d'exécution du processus appelant : l'espace d'adressage, la table des descripteurs de fichiers, la table des gestionnaires de signaux, etc... Ainsi, si l'on spécifie qu'il faut partager l'espace d'adressage avec le processus appelant, on obtient un *thread* utilisateur, sinon on obtient un nouveau processus. C'est ainsi que fonctionne la bibliothèque *Linux Threads* pour créer des threads (fonctions `pthread_create()`, `pthread_exit()`, etc..).

Suite à l'appel à `fork()` ou à `clone()`, l'espace d'adressage du fils est identique (partagé ou non) à celui du père. Pourtant, il est souhaitable de pouvoir exécuter des programmes différents dans un système d'exploitation. Les appels système de la famille de `exec()` permettent de réinitialiser l'espace d'adressage courant avec l'image d'un nouveau programme. Le code de cet appel système est la fonction `fs/exec.c:do_execve()`.

La destruction d'un thread se fait en utilisant l'appel système `kernel/exit.c:exit()` [Aiv02, Chapitre 2.2].

2.2.3 Changement de contexte et ordonnancement

Le changement de contexte d'un *thread* `t1` vers un autre `t2` consiste à sauvegarder l'état du processeur, correspondant à l'état du thread `t1`, puis à restaurer l'état du thread `t2` dans le processeur. Cet état rassemble toutes les informations caractérisant le flot d'instructions du thread, c'est-à-dire l'ensemble des registres du processeur. Parmi ces registres se trouve le pointeur de pile, ce qui automatise le changement de pile lors du changement de contexte.

Sous Linux, la sauvegarde et la restauration du contexte autre que la pile est effectuée par les macros `SAVE_ALL` et `RESTORE_ALL` (`arch/i386/entry.S`), tandis que le changement de pile s'effectue au niveau de la macro `include/asm/system.h:switch_to()`.

L'ordonnanceur est chargé de partager le temps processeur entre tous les threads prêts pour l'exécution (non bloqués sur une ressource). Celui de Linux est un ordonnanceur à temps partagé, classique dans la plupart des systèmes généralistes : il garantit une progression équitable de l'exécution des threads en fonction d'une priorité (`man nice`). Il bénéficie de petites extensions dites "temps-réel" sous la forme de files de threads ordonnancées par priorité sans temps partagé (`man sched_setscheduler` avec le paramètre `SCHED_FIFO`).

Le code de l'ordonnanceur se trouve dans `kernel/sched.c`, dont l'algorithme est relativement simple, malgré l'apparente complexité de la fonction `schedule()`. Cette complexité est dûe au fait que trois algorithmes d'ordonnancement sont implémentés dans la même fonction. Le prochain thread est élu en fonction d'une valeur de *goodness*, recalculée pour tous les threads par la fonction `goodness()`. À la fin de la fonction `schedule()` a lieu le changement de contexte vers le nouveau thread [Aiv02, Chapitre 2.3].

2.2.4 Interactions entre applications et noyau

Sur x86, un appel système ne correspond pas à une instruction particulière, mais à l'interruption logicielle (voir section 2.5.2) numéro `0x80` (macro `include/asm/hw_irq.h:SYSCALL_VECTOR`). Le système d'exploitation récupère l'identifiant du service demandé par l'appel système dans le registre `eax` du processeur. La liste des services offerts est donnée dans `include/asm/unistd.h`, et la table des pointeurs vers les fonctions du noyau implantant ces services est définie dans `arch/i386/entry.S:sys_call_table`. Les paramètres fournis à l'appel système sont passés dans les autres registres du processeur.

Pour effectuer le traitement associé à l'appel système demandé, le noyau travaille sur une pile différente de la pile du *thread* utilisateur appelant. Chaque *thread* utilisateur possède donc deux piles : une pile utilisateur extensible située à la fin de son espace d'adressage, et une petite pile noyau non extensible. Le passage d'une pile à l'autre est pris en charge automatiquement par le noyau lors de l'interruption `0x80`.

2.3 Processus : synchronisation et communication

Les processus ne sont pas des entités indépendantes : ils interagissent, soit par synchronisation, soit par communication d'informations.

La synchronisation permet de réglementer les accès concurrents à une ressource partagée, qu'elle soit matérielle ou logique. Sous certaines formes, elles permettent aussi d'échanger de l'information. Nous décrivons ci-dessous les différents types de synchronisation implantés dans Linux, ainsi que les modes de communication inter-processus (*IPC*, pour *Inter-Process Communication*) proposés.

2.3.1 Synchronisation dans le noyau

Opérations atomiques. La façon la plus simple de réglementer les accès concurrents est de les éviter en utilisant des opérations dites *atomiques*, c'est-à-dire des opérations durant lesquelles il n'est pas possible d'être interrompu : l'opération est effectuée d'un seul tenant.

Il existe deux types d'opérations atomiques garanties par le processeur [Aiv02, Chapitre 2.12] :

- Les opérations sur les bits (`include/asm/bitops.h`) permettent de changer la valeur d'un bit dans un bitmap, en effectuant éventuellement un test (fonctions `set_bit()`, `change_bit()`, `test_and_set_bit()`, ...).
- Les opérations sur des variables de type `include/asm/atomic.h:atomic_t` permettent de récupérer, de changer, d'incrémenter, de décrémenter la valeur de manière atomique, ou d'ajouter ou de soustraire une valeur de manière atomique. Un test peut optionnellement être réalisé (fonctions `atomic_set()`, `atomic_sub()`, `atomic_inc_and_test()`, ...).

Toutes les autres primitives de synchronisation reposent sur ces fonctionnalités élémentaires.

Spinlocks. Lorsqu'une opération ne peut pas être réalisée de manière atomique, il faut désactiver les interruptions matérielles (voir la section 2.5.2) de manière à ce que l'opération soit tout de même exécutée d'un seul tenant. Ceci fonctionne parfaitement sur une machine comportant un seul processeur. Toutefois, sur une machine à plusieurs processeurs, désactiver les interruptions ne fonctionne que sur le processeur courant, et du code s'exécutant sur les autres processeurs peut donc accéder à des données sur lesquelles on est en train de travailler.

C'est la raison pour laquelle les *spinlocks* [Aiv02, Chapitre 2.13] ont été introduits (`include/linux/spinlock.h`, `include/asm/spinlock.h`). Sur uniprocesseur, prendre un spinlock (en utilisant la fonction `spin_lock()`) consiste simplement à désactiver les interruptions. En multiprocesseur, un compteur permet de s'assurer de la disponibilité de la ressource. Si celle-ci n'est pas disponible, alors le *thread* attend de manière active (boucle infinie).

L'utilisation des *spinlocks* doit être limitée à protéger du code dont le temps d'exécution est très faible et non bloquant. Il n'est en effet pas acceptable de désactiver les interruptions pendant trop longtemps, ou de consommer du temps à ne rien faire durant une attente active.

Sémaphores. Les *sémaphores* (`include/asm/semaphore.h`) [Aiv02, Chapitre 2.14] sont des primitives de synchronisation. Contrairement aux *spinlocks*, pendant qu'on possède un sémaphore, on peut réaliser des opérations bloquantes. Ceci serait en effet dangereux si on utilisait des *spinlocks* : étant donné que les interruptions sont désactivées, on risquerait de ne pas être réveillé suite au blocage. Avec les sémaphores, un thread qui demande une ressource non disponible est placé dans un état d'attente (donc non éligible par l'ordonnanceur), et donc susceptible d'être réveillé lorsque la ressource deviendra disponible (suite à une interruption par exemple).

2.3.2 Synchronisation et communication au niveau utilisateur

Le noyau propose une série d'appels système et de paradigmes pour permettre aux processus de se synchroniser et de communiquer. Ces fonctionnalités sont implantées au-dessus de celles bâties dans le noyau. Certaines sont disponibles sur la plupart des Unix (signaux, fichiers), d'autres sont propres à l'API dite *IPC System V*, désormais largement répandue.

Sémaphores. Les appels système `semget()`, `semctl()` et `semop()` permettent de créer et d'utiliser des sémaphores au niveau utilisateur. Leur implémentation se situe dans `ipc/sem.c`.

Files de messages. Les files de messages [Aiv02, Chapitre 5.2] sont à la fois un mécanisme de synchronisation et de communication : elles permettent aux processus de se synchroniser sur l'attente d'un évènement, et la survenue de l'évènement est accompagnée d'informations personnalisables (le contenu du message).

Les files de messages s'utilisent via les appels systèmes `msgget()`, `msgctl()`, `msgsnd()`, `msgrcv()`, ... Ces appels système sont implantés dans `ipc/msg.c`.

Fichiers. Les fichiers sont un moyen de communication évident (voir la section 2.4), notamment les fichiers type *pipe* (`man pipe`) ou *fifo* (`man mkfifo`).

Ils sont aussi un moyen de synchronisation proche des sémaphores, grâce aux appels système `fcntl()` (flags `F_GETLK` et `F_SETLK`) et `flock()`.

Signaux. Les signaux sont un mécanisme de communication basique permettant de déclencher l'exécution de routines depuis d'autres processus. Ces routines sont appelées les *gestionnaires de signaux*. Dans le noyau, ces gestionnaires de signaux sont définis dans la structure `include/linux/sched.h:struct signal_struct`. Pour chaque processus, ils sont rassemblés dans le champ `sig` de la structure `include/linux/sched.h:struct task_struct`.

Le nombre de signaux et leur sémantique sont parfaitement limités (`include/asm/signal.h`) : seulement deux signaux, `SIGUSR1` et `SIGUSR2` sont disponibles pour l'utilisateur. Le système définit pour chaque signal un gestionnaire par défaut, qu'il est possible de changer pour la plupart des signaux en utilisant les appels système `signal()` ou `sigaction()`, dont le code est dans `kernel/signal.c`. Le signal `SIGSEGV` introduit en section 2.1.3 est l'un de ces signaux.

Mémoire partagée. La mémoire partagée [Aiv02, Chapitre 5.3] permet de disposer d'une région virtuelle de mémoire qui est partagée entre plusieurs espaces d'adressage, et donc entre plusieurs processus. C'est un moyen très efficace de communiquer des informations entre processus, mais ce n'est pas un moyen de synchronisation. L'utilisation d'une région de mémoire partagée s'accompagne donc souvent de l'utilisation d'un sémaphore pour protéger l'accès à cette ressource partagée.

Les appels système permettant d'utiliser la mémoire partagée sont `mmap()` (hautement portable) et `shmat()/shmget()/shmctl()` (API *IPC System V*). Ils sont respectivement implémentés dans `mm/mmap.c` et `ipc/shm.c`.

2.4 Fichiers

Le support des différents systèmes de fichiers (ext2, FAT, ...) repose sur la même architecture de base : le *VFS*, ou *Virtual File System*. Il définit les abstractions, les structures, et la liste des opérations communes à tous les systèmes de fichiers, et assure *1)* la cohérence de l'espace de nommage global (points de montage), *2)* l'interaction uniforme avec les applications utilisateur (interface de programmation unique), et *3)* l'intégration du sous-système de fichiers avec les autres sous-systèmes, notamment la gestion de la mémoire virtuelle (*mapping* de fichiers). Nous nous limitons ici à présenter le VFS, les détails sur les systèmes de fichiers sont abordés dans l'article "*ext2/ext3 : voyage au centre des fichiers*".

Le code du *VFS* se trouve principalement dans les fichiers source du répertoire `fs`, et le code des différents systèmes de fichiers se trouve dans les sous-répertoires correspondants.

2.4.1 Structures fondamentales

La figure 5 présente un aperçu de l'architecture du VFS, et des structures importantes en jeu.

Pour une application utilisateur, un fichier ouvert correspond à un entier. Pour tous les appels système de manipulation d'un fichier, le noyau utilise cet entier comme un index dans le tableau `fd_array` de la structure noyau `include/linux/sched.h:files_struct` propre à chaque processus (champ `files` de `include/linux/sched.h:task_struct`). L'entrée correspondante fait référence à un objet de type `include/linux/fs.h:struct file`.

Cette structure rassemble des informations propres à cette ouverture du fichier par ce processus (curseur de lecture/écriture, mode d'accès). Elle rassemble également les références vers les données communes à toutes les ouvertures du fichier qui ont été faites dans l'ensemble du système.

Parmi ces références, le pointeur `f_op` pointe vers la liste des opérations qui peuvent être effectuées sur le fichier (`read`, `write`, ...). C'est la structure `include/linux/fs.h:struct file_operations` qui les rassemble, sous la forme d'une série de pointeurs vers des fonctions. Ces fonctions sont implantées par le système de fichiers prenant en charge ce fichier.

Le pointeur `f_vfsmnt` de la structure `file` pointe justement vers une structure de type `include/linux/mount.h:struct vfsmount`, et fait ainsi référence au système de fichiers prenant en charge le fichier. Elle est unique pour chaque montage d'un système de fichiers. D'une part, elle permet d'insérer la sous-arborescence de fichiers liée à ce système de fichiers dans l'arborescence globale. D'autre part, elle fait référence à une structure de type `include/linux/fs.h:struct super_block` (pointeur `mnt_sb`). Ce *superblock* est unique pour chaque périphérique bloc monté, et rassemble un pointeur (champ `s_op`) vers les opérations nécessaires pour gérer les *inodes* (décrits plus bas) du système de fichiers (structure `include/linux/fs.h:super_operations`). Le système de fichiers doit ainsi implanter les opérations de création, suppression, etc... des *inodes*.

Le pointeur `f_dentry` de la structure `file` pointe sur une structure de type `include/linux/dcache.h:struct dentry`. Cette structure est utilisée par le noyau pour construire un arbre en mémoire représentant partiellement l'arborescence (champs `d_parent` et `d_subdirs`) des fichiers du système. Elle joue le rôle de cache pour la recherche de fichiers lors de leur ouverture, et possède donc un nom (champ `d_iname` : c'est par exemple "`toto`" pour la *dentry* correspondant à `/home/toto`). Elle possède également un pointeur vers l'*inode* du fichier associé sur disque. Plusieurs `dentry` peuvent correspondre au même `inode`, ce qui signifie qu'un même fichier peut être présent sous différents noms dans différents répertoires : c'est la notion de *hard link*.

C'est cet objet de type `include/linux/fs.h:struct inode` qui ras-

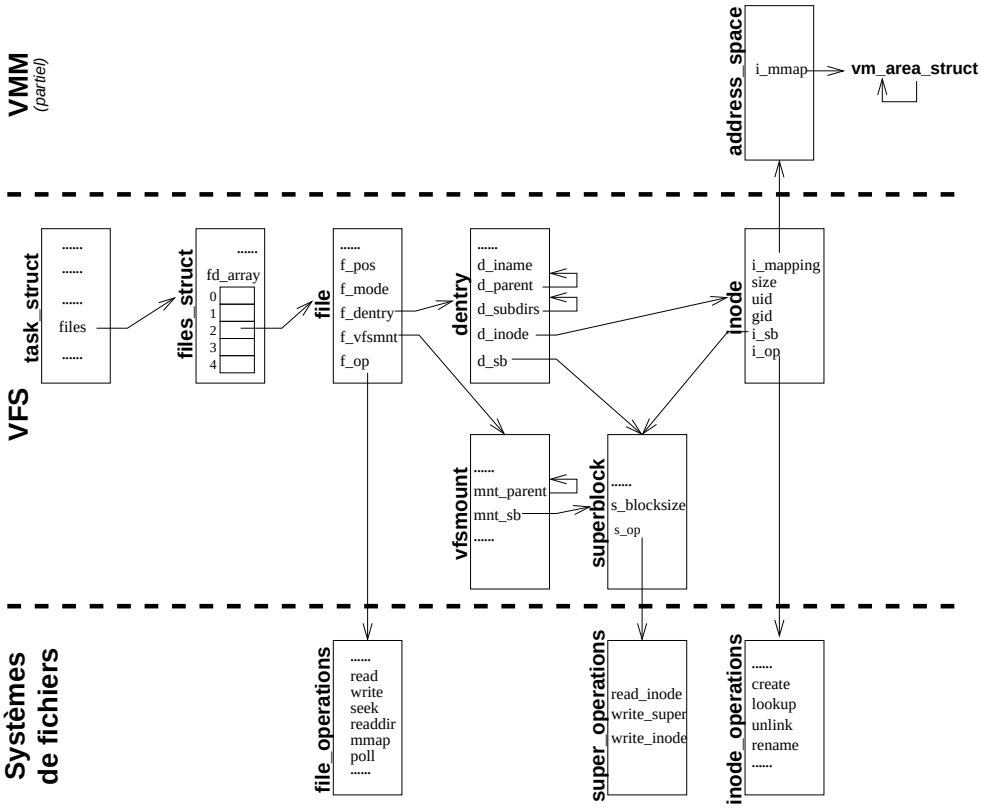


FIG. 5 – Architecture de VFS

semble les informations sur le fichier (taille, propriétaire, droits d'accès, compteur de références pour gérer les liens et la suppression des fichiers...), et un pointeur vers les opérations, sous la forme de pointeurs vers des fonctions (champ `include/linux/fs.h:struct inode_operations`), qui permettent de les modifier (`unlink()`, `mkdir()`, `truncate()`, ...). Ces opérations sont implantées par le système de fichiers auquel appartient le fichier. Il existe exactement un objet de ce type par fichier dans le système, associé à un numéro d'*inode*. La plupart de ses informations est en général stockée sur disque. La structure `inode` contient également un pointeur (champ `i_mapping`) vers une structure de type `include/linux/fs.h:struct address_space`, qui rassemble la liste des régions virtuelles (voir section 2.1.2) qui projettent le fichier en mémoire. Toutefois, un *inode* ne correspond pas nécessairement à un fichier dans l'espace de nommage : il existe des *inodes anonymes*, comme les *sockets*.

2.4.2 Ouverture d'un fichier

Lorsqu'une application utilisateur fait l'appel système `open()` [Bro01, Chapitre 2], le VFS effectue la recherche du fichier composante par composante (`home` puis `toto`, puis `.bashrc` pour le fichier `/home/toto/.bashrc`). Pour cela, il utilise l'arbre des `dentry` dans la fonction `fs/namei.c:link_path_walk()`. Au fur et à mesure de la recherche, si une *dentry* n'est pas déjà en cache, `fs/namei.c:real_lookup()` est invoquée pour construire cette nouvelle `dentry`. Pour cela, l'*inode* correspondant à la composante est construit en utilisant la fonction `lookup()` de `inode_operations` à partir de l'*inode* de la dernière composante trouvée.

Une fois le `dentry` et son `inode` associé récupérés, une structure `file` est allouée et un emplacement disponible dans le tableau `fd_array` du processus appelant y fait référence. L'identifiant du fichier renvoyé à l'utilisateur est l'index dans ce tableau : c'est le *descripteur du fichier*.

Cette procédure ne s'applique pas à tous les descripteurs de fichiers. Ainsi, un descripteur de fichier correspondant à une *socket* ne peut être retourné que par l'appel système `socket()`.

2.4.3 Manipulation d'un fichier

Lors de la manipulation d'un fichier par les appels système `read()` ou `write()` (`fs/read_write.c`) par exemple, le noyau récupère la structure `file` associée au descripteur de fichier passé en paramètre, puis appelle la fonction correspondante de `file_operations` [Bro01, Chapitre 3].

2.5 Pilotes de périphérique

Un pilote de périphérique permet de représenter une ressource matérielle (disque dur, carte son, carte réseau, ...) sous la forme d'une ressource logique manipulable

par l'utilisateur (fichier, interface réseau). Certains pilotes sont directement accessibles à l'utilisateur sous la forme de ces ressources logiques. D'autres ne le sont pas : ils exportent des interfaces utilisées en interne au noyau par les autres pilotes.

Le code des pilotes de périphériques se trouve principalement dans les sous-répertoires du répertoire `drivers`. La plupart de ces pilotes se présente sous la forme de modules chargeables dynamiquement (voir l'article sur le sujet dans ce numéro).

On se limitera à exposer les interactions ayant lieu avec le matériel : on ne rentrera pas dans le détail des multiples pilotes de périphériques. Pour plus d'informations sur la programmation de pilotes de périphériques sous Linux, voir [RC01], et pour plus d'informations sur le matériel, voir [Mes99].

2.5.1 Types de pilotes et sous-systèmes

L'interface exportée par les pilotes de périphériques est de trois types : caractère, bloc, ou réseau. Les deux premiers types se matérialisent sous la forme d'un fichier accessible en mode caractère ou mode bloc (voir section 1.2.1). L'interface réseau exportée par les périphériques réseau est interne au noyau, l'utilisateur y accède au travers de *socket*s et de la pile de protocoles réseau sous-jacente.

Ces trois catégories de pilotes se retrouvent respectivement dans les sous-répertoires `char`, `block` et `net`. On trouve dans la catégorie caractère les pilotes de dispositifs de pointage et de saisie, les ports d'imprimante, les ports séries, les consoles, etc.. Dans la catégorie bloc, on trouve les pilotes de disques durs, de CD-ROM, disquettes, etc...

Les pilotes de périphériques qui ne sont pas accessibles directement par l'utilisateur sont utilisés par les autres pilotes de périphériques évoqués ci-dessus. Il se chargent de gérer les sous-systèmes liés à la gestion du bus USB, des cartes SCSI, du bus PCI, des cartes Firewire, du bus I2C, etc...

2.5.2 Interaction avec le matériel

Pour échanger des données avec les périphériques matériels, les pilotes de périphériques utilisent trois modes d'accès [Mes99] : les ports d'entrée/sortie, la projection des périphériques dans l'espace des adresses physiques, et le *DMA* (pour *Direct Memory Access*).

Le premier est particulier aux processeurs de la famille x86, et correspond à l'équivalent de petites zones dans lesquelles le pilote peut dialoguer avec le périphérique mot à mot (octets, mots de 16 bits, mots de 32 bits) à l'aide d'instructions particulières (voir les macros `include/asm/io.h:OUT` et `IN` qui définissent les macros `outb()`, `inb()`, ...).

Dans le deuxième mode, le périphérique prend en charge tous les accès à une partie de l'espace des adresses physiques : il est *mappé* dans l'espace physique. Pour dialoguer avec lui, le pilote doit simplement lire et écrire dans cette portion de la mémoire physique. Sous Linux, ceci est réalisé grâce à l'*identity mapping* de la mémoire physique dans l'espace noyau (voir la section 2.1.2).

Le troisième mode permet de transférer des données entre un périphérique et la mémoire physique sans utiliser le processeur. On permet au périphérique de prendre en charge ce transfert et de signaler la terminaison de ce transfert par une interruption matérielle. Les zones éligibles pour un transfert *DMA* (`include/asm-i386/dma.h`) sont soumises à des contraintes de position dans la mémoire physique et de taille (avec le bus ISA, des zones de 64 Ko contiguës situées en dessous des 16 Mo). Sur les architectures plus modernes (PCI ou AGP par exemple), ces contraintes sont levées.

Les interruptions sont des évènements qui interrompent le flot d'exécution courant des instructions : une routine spécifique à chaque interruption et prédéfinie par le système est alors exécutée. Les interruptions proviennent de plusieurs sources :

- le processeur pour signaler des évènements internes : division par zéro, instruction invalide, défaut de page, ... On parle d'*exceptions*.
- le matériel pour signaler la terminaison d'un transfert (*DMA*) ou un évènement requérant un traitement (frappe d'une touche au clavier, réception d'un paquet réseau, ...). Il s'agit des fameuses *IRQ*s (`include/asm-i386/irq.h`).
- le logiciel pour exécuter un appel système.

Le traitement des interruptions doit être le plus court possible parce que durant l'exécution de la routine, les interruptions sont désactivées, ce qui diminue la réactivité du système. On déporte donc les traitements les plus longs grâce à divers mécanismes implantés par le noyau Linux : les *tasklets* et les *softirqs*.

2.5.3 Représentation dans l'espace de nommage

Les fichiers spéciaux qui permettent d'interagir avec les périphériques sont traditionnellement regroupés dans le répertoire `/dev`. Il existe un système de fichiers virtuel chargé de ne faire apparaître dans ce répertoire que les périphériques effectivement accessibles (*devfs*).

D'autre part, il est possible de consulter l'état des pilotes et leur configuration dans le système de fichiers particulier de type *procfs* traditionnellement monté sur `/proc`. Ce système permet également de modifier quelques uns de ces paramètres, mais le système de fichiers *sysfs* [Cor03] en cours de développement généralisera cette possibilité à tous les pilotes.

2.6 Code dépendant de l'architecture

Le noyau Linux compile sur 17 architectures : le code dépendant de l'architecture a donc été séparé du reste du noyau. Ce code se trouve dans le répertoire `arch/i386` et concerne essentiellement :

- L'initialisation du système (phase de boot), `boot/setup.S`.

– La gestion des interruptions. Le code de bas niveau est dans `kernel/entry.S`, et le code lié aux exceptions se situe dans `kernel/traps.c`.
– La gestion des contrôleurs d'interruptions matérielles, `kernel/irq.c`, `kernel/i8259.c` et `kernel/apic.c`
– L'initialisation du fonctionnement en multi-processeurs (`kernel/smp.c` et `kernel/smpboot.c`).
– Le code de bas niveau pour les primitives de synchronisation (`kernel/semaphore.c`, `kernel/signal.c`).
– La gestion de la pagination : répertoire `kernel/mm`, avec notamment `fault.c` qui contient la routine de bas niveau de gestion des défauts de page.
– Le code bas niveau pour la gestion du bus PCI.

Conclusion

Nous avons présenté les concepts fondamentaux aux systèmes d'exploitation en général, et les éléments essentiels du noyau Linux en particulier.

Cette présentation omet volontairement certains éléments, tels que la structure et l'interaction avec le sous-système réseau, ou encore les interfaces internes au noyau qui structurent l'assemblage des pilotes de périphériques. Pour en savoir plus, nous vous encourageons à étudier le noyau Linux par vous-mêmes, à lire les documentations s'y référant, ou à travailler sur votre propre noyau...

Thomas Petazzoni et David Decotigny
Projet KOS (`http://kos.enix.org`)
`Thomas.Petazzoni@enix.org` et `d2@enix.org`

Références

[Aiv02] Tigran Aivazian. *Linux Kernel 2.4 Internals*. GNU, `http://www.mc.man.ac.uk/LDP/LDP/lki/lki.html`, 2002.

[Bon94] Jeff Bonwick. The slab allocator : An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98, 1994.

[Bro01] Andries Brouwer. *A small trail through the Linux kernel*. `http://www.win.tue.nl/~aeb/linux/vfs/trail.html`, 2001.

[Cor03] Jonathan Corbet. *Porting device drivers to the 2.5 kernel*. Eklektix, `http://lwn.net/Articles/driver-porting/`, 2003.

[Gor03] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. GNU, `http://www.csn.ul.ie/~mel/projects/vm/`, 2003.

[Mes99] Hans-Peter Messmer. *The Indispensable PC Hardware Book : Your Hardware Questions Answered (3rd Edition)*. Number ISBN 0201403994. Addison-Wesley, 1999.

[PN77] James L. Peterson and Theodore A. Norman. Buddy systems. *Communications of the ACM*, 20(6) :421–431, 1977.

[RC01] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers*. Number ISSN 0596000081. O'Reilly, `http://www.xml.com/ldd/chapter/book/index.html`, 2001.

[Tan94] Andrew Tanenbaum. *Systémes d'exploitation*. Number ISBN 2100045547. InterEditions / Pentice Hall, 1994.

[Vah95] Uresh Vahalia. *UNIX Internals : The New Frontiers*. Number ISSN 0131019082. Prentice Hall, 1995.