N[LSGJTR‘KMJN[LSGJTR‘KMJN[LSGJTR‘KMJN[LSGJTR‘KMJN[LSGJTR‘KMJN[LSGJTR‘KMJN[LSGJTR‘KMJN[LSGJTR‘KMJN[LSGJTR
XAJBPVRLRTFPXAJBPVRLRTFPXAJBPVRLRTFPXAJBPVRLRTFPXAJBPVRLRTFPXAJBPVRLRTFPXAJBPVRLRTFPXAJBPVRLRTFPXAJBPVRL
J^COPCQE_PSVJ^COPCQE_PSVJ^COPCQE_PSVJ^COPCQE_PSVJ^COPCQE_PSVJ^COPCQE_PSVJ^COPCQE_PSVJ^COPCQE_PSVJ^COPCQE_PSVJ^COPCQE
\RZQRFTIET[U\RZQRFTIET[U\RZQRFTIET[U\RZQRFTIET[U\RZQRFTIET[U\RZQRFTIET[U\RZQRFTIET[U\RZQRFTIET[U\RZQRFTIET[U\RZQRFTI
WSMFBHEEDHYNWSMFBHEEDHYNWSMFBHEEDHYNWSMFBHEEDHYNWSMFBHEEDHYNWSMFBHEEDHYNWSMFBHEEDHYNWSMFBHEEDHYNWSMFBHEEDHYNWSMFBHEE
E‘BLO[VHPWK_E‘BLO[VHPWK_E‘BLO[VHPWK_E‘BLO[VHPWK_E‘BLO[VHPWK_E‘BLO[VHPWK_E‘BLO[VHPWK_E‘BLO[VHPWK_E‘BLO[VHPWK_E‘BLO[VH
C]XNVBVVRMT^C]XNVBVVRMT^C]XNVBVVRMT^C]XNVBVVRMT^C]XNVBVVRMT^C]XNVBVVRMT^C]XNVBVVRMT^C]XNVBVVRMT^C]XNVBVVRMT^C]XNVBVV
RDSDCD\XTHU[RDSDCD\XTHU[RDSDCD\XTHU[RDSDCD\XTHU[RDSDCD\XTHU[RDSDCD\XTHU[RDSDCD\XTHU[RDSDCD\XTHU[RDSDCD\XTHU[RDSDCD\X
LI^CWYZBYCILI^^CWYZBYCILICWYZBYBYCILCWYZBYYBYCICWYZBYYYBYCWYZBYYYYYBYYZBYYYYYYYBYZBYYYYYYYYYZBYYYYYYYYYY
DBE]UXVZOF\DBEE]UXVZOF\DB]UXVZOZOF\D]UXVZOOZOF\]UXVZOOOZOFUXVZOOOOOZOXVZOOOOOOOZXVZOOOOOOOOXVZOOOOOOOOOX
_V_BLQACJYB_V__BLQACJYB_VBLQACJCJYB_BLQACJCCJYBBLQACJJCCJYLQACJJJJCCJQACJJJJJJCCJACJJJJJJJCJACJJJJJJJJCJ
BM]LNRD_HZPBM]]LNRD_HZPBMLNRD_H_HZPBLNRD_H__HZPLNRD_HH__HZNRD_HHHH__HRD_HHHHHH__HD_HHHHHHH_HD_HHHHHHHH_H
HE\JQNSOXHTHE\\JQNSOXHTHEJQNSOXOXHTHJQNSOXOXXHTJQNSOXXOXXHQNSOXXXXOXXNSOXXXXXXOXXNOXXXXXXOXXNOXXXXXXXOXX
Y[W^]]FIOQEY[WW^]]FIOQEY[^]]FIOIOQEY^]]FIOIOOQE^]]FIOOIOOQ]]FIOOOOIOO]FIOOOOOOIOO]IOOOOOOIOO]IOOOOOOOIOO
RT[MTKPJRENRT[[MTKPJRENRTMTKPJRJRENRMTKPJRRREENMTKPJRRRREETKPJRRRRRREKPJRRRRRRRREKPJRRRRRRREKKPJRRRRRRRE
E^MBMCICNRRE^MMBMCICNRRE^BMCICNCNRREBMCICNNNRRRBMCICNNNNRRMCICNNNNNNRCICNNNNNNNNRCICNNNNNNNRCCICNNNNNNNR
ET_^YMCNQBFET__^YMCNQBFET^YMCNQNQBFE^YMCNQQNBFF^YMCNQQQNBFYMCNQQQQQNBMCNQQQQQQQNBMNQQQQQQQNBMNQQQQQQQQNB
ZGNDCNEAGJCZGNNDCNEAGJCZGDCNEAGAGJCZDCNEAGGAJCCDCNEAGGGAJCCNEAGGGGGAJNEAGGGGGGGAJNAGGGGGGGAJNAGGGGGGGAJ
CYGXB\SRJIICYGXB\SRJJIICYB\SRJJJJIICB\SRJJJJJICB\SRJJJJJJI\SRJJJJJJJJSRJJJJJJJJJRJJJJJJJJJRJJJJJJJJJJ
CNJAKKK^MEOCNJAKKK^MMEOCNKKK^MMMMEOCKKK^MMMMMOCKKK^MMMMMMOCK^MMMMMMMCK^MMMMMMMMC^MMMMMMMMC^MMMMMMMMMC
T\CHSYCRQ_ET\CHSYCRQQ_ET\SYCRQQQQ_ETSYCRQQQQQ_ESYCRQQQQQQ_ESYQQQQQQ_ESYQQQQQQQQESYQQQQQQQQQSYQQQQQQQQQQS
JAALRTP_IOOJAALRTP_IIOOJARTP_IIIIOOJRTP_IIIIIOORTP_IIIIIIOORTP_IIIIOORTRTP_IIIIORTRTPP_IIIIRTRTPPP_IIIIR
UE[]TMWOPDMSUE[]TMWOPDMSUE[]TMWOPDMSUE[]TMWOPDMSUE[]TMWOPDMSUE[]TMWOPDMSUE[]TMWOPDMSUE[]TMWOPDMSUE[]TMWO
QN_]QZ]WY\MJQN_]QZ]WY\MJQN_]QZ]WY\MJQN_]QZ]WY\MJQN_]QZ]WY\MJQN_]QZ]WY\MJQN_]QZ]WY\MJQN_]QZ]WY\MJQN_]QZ]W
USBD‘_XKFYRTUSBD‘_XKFYRTUSBD‘_XKFYRTUSBD‘_XKFYRTUSBD‘_XKFYRTUSBD‘_XKFYRTUSBD‘_XKFYRTUSBD‘_XKFYRTUSBD‘_XK
P_NJ_SVITVU‘P_NJ_SVITVU‘P_NJ_SVITVU‘P_NJ_SVITVU‘P_NJ_SVITVU‘P_NJ_SVITVU‘P_NJ_SVITVU‘P_NJ_SVITVU‘P_NJ_SVI
‘BWH^J^RDPPE‘BWH^J^RDPPE‘BWH^J^RDPPE‘BWH^J^RDPPE‘BWH^J^RDPPE‘BWH^J^RDPPE‘BWH^J^RDPPE‘BWH^J^RDPPE‘BWH^J^R
‘UHHVRYG[RQN‘UHHVRYG[RQN‘UHHVRYG[RQN‘UHHVRYG[RQN‘UHHVRYG[RQN‘UHHVRYG[RQN‘UHHVRYG[RQN‘UHHVRYG[RQN‘UHHVRYG
NUR^SINAQO^YNUR^SINAQO^YNUR^SINAQO^YNUR^SINAQO^YNUR^SINAQO^YNUR^SINAQO^YNUR^SINAQO^YNUR^SINAQO^YNUR^SINAQO^YNUR^SINA
[GW‘PXVMVML[[GW‘PXVMVML[[GW‘PXVMVML[[GW‘PXVMVML[[GW‘PXVMVML[[GW‘PXVMVML[[GW‘PXVMVML[[GW‘PXVMVML[[GW‘PXVM

Arnaud REVEL - revel@ensea.fr

Contributions : J.P. Bachmann, M.Maillard, D.Picard

Table des matières

NOTA BENE

Dans certains des listings donnés au sein de ce document, la notation :
`#include <stdio.h> <stdlib.h>` *est utilisée.*

Cependant le `C` *ne connaît pas cette notation qui est simplement utilisée pour des raisons de concision !*

Il faut en fait bien sur comprendre :

```
#include <stdio.h>
#include <stdlib.h>
```

Chapitre 1

Introduction

1.1 Historique

1.1.1 UNIX : l'inspirateur

UNIX est né en 1969 dans les laboratoires de BELL. Ce sont Ken Thompson et Denis Ritchie (qui est aussi l'un des "papas" du C), qui ont créé la première version de ce système d'exploitation. Les principes de base d'UNIX reprenaient ceux du projet MULTICS, qui avait été soutenu conjointement par General Electric, le MIT et les laboratoires BELL. Par la suite, UNIX a connu une histoire quelquefois mouvementée que nous résumons dans le tableau 1.1.

Année	Faits marquants	Caractéristiques
1969	Thompson et Ritchie travaillant aux laboratoires BELL, propriété d'AT&T ⇒ première version d'UNIX	– Système dédié PDP7/9 – Noyau : 16 Ko – Processus : 8 Ko – Fichier : 64 Ko
1969-1970	Réécriture du noyau UNIX en langage C	– Gestion des processus – Multiprogrammation – Gestion des fichiers et volumes – Banalisation des E/S
1971	Kernighan et Ritchie ⇒ langage C	
1975	Premier portage d'UNIX	PDP/11
1975	Thompson vient travailler à l'université de Berkeley	
1977	Création de la version Berkeley Software Distribution (BSD)	
1979	Portages "historiques"	VAX et DEC
1982	Commercialisation par ATT UNIX like (licence ATT) UNIX based (pas de licence)	– communication entre processus – mémoire virtuelle – outils réseaux
1985	Création de MINIX par A.Tanenbaum	
1989	Première BSD libre	
1991	Conception de Linux sur la base de MINIX	architecture 80386
1993	NetBSD et Free BSD	
1999	Mac OS X	

Tab. 1.1 – Rapide historique d'UNIX

1.1.2 Linux

Linux, ou GNU/Linux, est un système d'exploitation compatible POSIX (voir 1.4). GNU/Linux est basé sur le noyau Linux, un système d'exploitation libre créé en 1991 par Linus Torvalds sur un ordinateur compatible PC. Linux ne contient pas de code provenant d'UNIX, mais c'est un système inspiré d'UNIX (et de MINIX, le système d'exploitation d'Andrew Tanenbaum [Tanenbaum, 2003]) et complètement réécrit. Linux lui même n'étant qu'un noyau, il utilise l'ensemble des logiciels du projet GNU pour faire un système d'exploitation complet.

L'annonce de la première version (0.0.1) de son système d'exploitation par Linus sur le groupe de discussion `comp.os.minix` se fit en ces termes :

> « Hello everybody out there using minix - I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. »

Linux est un UNIX-like libre (gratuit et dont les sources sont libres de droits). Par ailleurs, il fonctionne sur de nombreuses machines, et en particulier sur PC, ce qui en fait un redoutable adversaire de Window$. Ce n'est pas le seul UNIX libre (cf. FreeBSD et NetBSD), mais c'est celui le plus populaire puisque sont développement s'est réalisé grâce à l'essor d'Internet. Enfin, un autre avantage indéniable est le support d'un très grand nombre d'architectures (23 à l'heure actuelle).

Mais la grande originalité est surtout le développement de logiciels dits "libres"(souvent développés sous licence GNU — [Wikipedia, 2008b]) spécialement conçus pour fonctionner sous Linux.

Version	Date	Caractéristiques
0.01	17 septembre 1991	diffusion confidentielle
0.02	5 octobre 1991	annonces sur usenet, système quasi inutilisable
0.03	octobre 1991	bash et gcc disponibles en binaire
0.10	décembre 1991	premières contributions externes, internationalisation du clavier
0.11	mi-décembre 1991	pilote pour disquette, SCSI en développement
0.12	5 janvier 1992	mémoire virtuelle, système utilisable, plus de matériel supporté, diffusé en GNU GPL, consoles virtuelles
0.95	7 mars 1992	init/login, X Window est porté, un groupe de discussion existe : alt.os.linux
0.95a	17 mars 1992	Nouveau mainteneur pour les linux root diskette : Jim Winstead
0.96 - 0.99 patch level 15Z		2 ans de développement, pour l'ajout de fonctionnalités et de corrections, les forums comp.os.linux.* sont les plus fréquentés de usenet et sont réorganisés 3 fois, signe que la communauté grandit et est très active.
1.0	mars 1994	Le noyau Linux est stable, pour la production et fournit les services d'un UNIX classique
1.2	mars 1995	Beaucoup plus d'architectures processeur, modules chargeables, ...
2.0	juillet 1996	PowerPC, Multiprocesseur, plus de matériels supportés, gestion du réseau plus complète, apparition de la mascotte Tux
2.2	janvier 1999	FrameBuffer, NTFS, Joliet, IPv6, ...
2.4	janvier 2001	USB, PCMCIA, I2O, NFS 3, ...
2.6	décembre 2003	ALSA, noyau préemptible, NFS 4, ...
2.6.21	avril 2007	Interface de paravirtualisation VMI, Dynticks et Clockevents, ...
2.6.22	juillet 2007	Toute nouvelle couche wifi, allocateur de mémoire SLUB, ordonnanceur d'E/S CFQ, nouveaux pilotes ...
2.6.23	octobre 2007	Nouvel ordonnanceur de tâches CFS, environnement de support des pilotes en espace utilisateur UIO intégré au noyau, SLUB allocateur de mémoire par défaut, ...
2.6.24	janvier 2008	Unification des architectures i386 et $x86_64$, E/S vectorielles, authentification des périphériques USB, ordonnancement de groupe avec CFS, ...
2.6.25	avril 2008	SMACK (alternative à SELinux), gestion du bus CAN, refonte de timerfd, amélioration de la gestion du temps réel...
2.6.26	13 juillet 2008	Intégration du débogueur du noyau kgdb, début de support des réseaux à topologie maillée unifiée, support des écrans Braille, support du PAT pour architecture x86, montage "–bind" en lecture seule, gestion de droits de sécurité par processus (securebits), amélioration de la virtualisation avec KVM...

TAB. 1.2 – Chronologie du noyau Linux d'après Wikipedia

La tendance actuelle...

Linux bénéficie d'une bonne réputation surtout sur le plan de la fiabilité et de la robustesse. Cela le rend attrayant pour de nombreux domaines. Sa gamme d'utilisation est ainsi très variée : en plus de son utilisation sur les stations de travail, on le retrouve plus particulièrement sur les serveurs réseau (stabilité) et sur les cluster de calculs (bonne résistance à la montée en charge). Enfin les nombreux portages de LINUX (Arm, Coldfire, Mips, DSP...), en font un système d'exploitation privilégié dans le domaine de l'embarqué : LINUX et les UNIX-Like (uClinux, QNX, LynxOS, VxWorks...) constituent 70% du marché de l'embarqué. Ce domaine connaît en ce moment une véritable explosion (14 milliards de processeurs pour l'embarqué vendus en 2004 contre seulement 260 millions de processeurs pour

les PC). Cela permet à Linux et aux autres déclinaisons d'UNIX d'être présents de l'équipement automobile à la borne wifi routeur et ainsi de lui assurer encore de beaux jours.

1.2 Principes de base

Ces principes, qui sont ceux d'UNIX, constituent encore la base de Linux :
– définition de la notion de processus (cf. chapitre 3)
– portabilité du code (écriture en C)
– organisation des fichiers en arborescences (cf. chapitre 4).
– banalisation des entrées/sorties (cf. chapitre 4 et 8).

1.3 Rôles du système d'exploitation

Un système d'exploitation se trouve à l'interface entre l'utilisateur et la machine que celui-ci désire utiliser. Sous Linux, le "noyau" est la partie du système d'exploitation (OS[1]) en charge de lien entre le matériel et le logiciel. Les principaux rôles qu'il doit assurer sont :

– Le contrôle de la fiabilité du système
– La gestion du partage de la machine entre différents travaux et/ou utilisateurs (*ordonnancement*)
– Le contrôle des ressources de la machine
– La gestion des entrées/sorties, gestion des périphériques (au moyen de pilotes)
– La conservation et la protection de l'information en cours de traitement
– L'interface avec utilisateur

Cette liste de tâches qui incombent à UNIX nous servira par la suite de trame pour la présentation des mécanismes internes.

1.4 Quelques définitions supplémentaires... (from wikipedia)

POSIX est le nom d'une famille de standards définie depuis 1988 par l'IEEE et formellement désignée IEEE 1003. Ces standards ont émergé d'un projet de standardisation des APIs des logiciels destinés à fonctionner sur des variantes du système d'exploitation UNIX. Le terme POSIX a été suggéré par Richard Stallman en réponse à la demande de l'IEEE d'un nom facilement mémorisable. C'est un acronyme de Portable Operating System Interface, dont le X exprime l'héritage UNIX de l'Interface de programmation.

POSIX spécifie dans 17 documents différents les interfaces utilisateurs et les interfaces logicielles. La ligne de commande standard et l'interface de script est le Korn shell. Les autres commandes, services et utilitaires comprennent awk, echo, ed, et des centaines d'autres. Les services d'entrées/sorties de base (fichiers, terminaux, réseau) doivent être présents (pour les spécifications POSIX sur les attributs de fichiers, voir Spécifications POSIX sur les attributs de fichiers).

POSIX définit aussi une API standard pour les bibliothèques de threading qui est prise en charge par la plupart des systèmes d'exploitation récents. Une suite de tests pour POSIX accompagne le standard. Il est appelé PCTS (POSIX Conformance Test Suite, Suite de tests pour la conformité POSIX).

GNU Le projet GNU[2] est lancé par Richard Stallman en 1984, alors qu'il travaillait au laboratoire

[1] Operating System

[2] Son nom est un acronyme récursif qui signifie en anglais "Gnu's Not Unix" (littéralement, GNU N'est pas UNIX) en référence d'une part à sa similitude ou plutôt son accointance ou ses accointances (atomes crochus, parenté) avec UNIX et d'autre part à sa volonté d'échapper à toute pression des "propriétaires" d'UNIX.

d'intelligence artificielle du MIT, afin de créer un système d'exploitation libre et complet et, d'après ses mots, « ramener l'esprit de coopération qui prévalait dans la communauté informatique dans les jours anciens » (il n'était pas question alors de propriété intellectuelle, et tous les codes sources, distincts, s'échangeaient librement).

On ne peut comprendre réellement ce qu'est le projet GNU si on en néglige ses motivations, relevant de l'éthique et de la philosophie politique. Il vise en effet à ne laisser l'homme devenir ni l'esclave de la machine et de ceux qui auraient l'exclusivité de sa programmation, ni de cartels monopolisant des connaissances en fonction de leurs seuls intérêts. Le projet GNU oeuvre pour une libre diffusion des connaissances, ce qui n'est pas sans implications politiques, éthiques, philosophiques et sociales, ou sociétales.

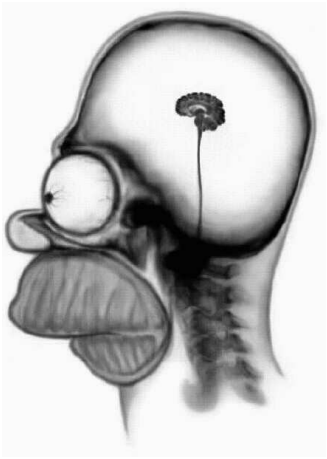
Les programmes disponibles en accord avec le projet GNU sont appelés les paquets GNU ou les programmes GNU. Parmi les composants de base du système, on retrouve : la collection de compilateurs GNU (GCC), les outils binaires GNU (binutils), le shell Bash, la bibliothèque C GNU (glibc), et les outils de base GNU (coreutils).

API Une interface de programmation (Application Programming Interface ou API) permet de définir la manière dont un composant informatique peut communiquer avec un autre. C'est donc une interface de code source fournie par un système informatique ou une bibliothèque logicielle, en vue de répondre à des requêtes pour des services qu'un programme informatique pourrait lui faire. La connaissance des API est indispensable à l'interopérabilité entre les composants logiciels.

ABI En informatique, une application binary interface (ABI, interface binaire-programme), décrit une interface bas niveau entre les applications et le système d'exploitation, entre une application et une bibliothèque, ou bien entre différentes parties d'une application. Une ABI diffère d'une API puisque une API définit une interfaces entre du code source et une bibliothèque, de façon à assurer que le code source compilera sur tout système supportant cette API.

Chapitre 2

Le noyau (Kernel) Linux



2.1 Linux : une architecture hiérarchique

Linux est structuré selon une architecture hiérarchique en couches dont le cœur est le **noyau Linux**[1]

L'intérêt de l'architecture en couches est de permettre la construction hiérachique d'outils spécifiques à partir des fonctionnalités d'un ou plusieurs outils plus simples. Ainsi, **GNU/Linux** est un système ouvert qui permet d'intégrer facilement de nouveaux outils en respectant cependant une interface normalisée.

Le système d'exploitation, du fait de cette structure, permet de plus aux applications d'être moins dépendantes de la machine sur lesquelles elles fonctionnent. Le système d'exploitation masque en effet les particularités de chaque ordinateur, en garantissant les interfaces nécessaires à la compatibilité. Sous Linux, comme sous tous les UNIX, cette interface est princpalement fournie par les appels systèmes.

La version du **13 juillet 2008** est la **2.6.26** (voir [Wikipedia, 2008d] pour des informations sur la politique d'évolution du noyau). Par ailleurs vous trouverez une foule d'informations sur le développement du noyau et les algorithmes utilisés dans le répertoire `/usr/ src/ linux-VERSION/ Documentation/`(voir [Collectif, 2008]).

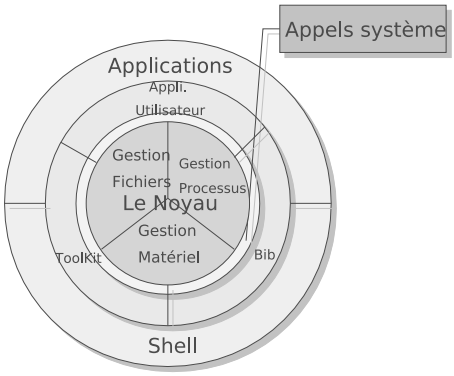


FIG. 2.1 – Architecture hiérarchique d'UNIX/Linux

[1]Onions and UNIX have two things in common ; they both have shell-like structures, and both cause tears to flow from those who look beneath surface[Goodheart and Cox, 1994b].

2.2 Point de vue utilisateur

L'utilisateur peut accéder aux ressources de la machine par l'intermédiaire de différents outils et services mis à sa disposition par le noyau.

L'un des principaux outils d'interface entre l'utilisateur et la machine est l'interpréteur de commandes (bourne-again-shell/bash, C-shell, ...). Par ailleurs, il peut accéder aux ressources de la machine en créant des applications qui font appel au système grâce à des bibliothèques de fonctions. La bibliothèque principale sous Linux est ainsi la glibc (GNU C Library). Le système d'exploitation Linux, et c'est l'une de ses principale forces, est de aussi fourni avec de nombreux logiciels, principalement sous licence GNU :

- assembleur, éditeurs de liens, compilateurs, déboggeurs
- éditeur de texte, de graphiques, logiciels mathématiques
- environnement graphiques
- fonctions réseaux
- ...

Les distributions[2] rassemblent les composants d'un système dans un ensemble cohérent et stable dont l'installation, l'utilisation et la maintenance sont facilitées. Elles comprennent donc le plus souvent un logiciel d'installation et des outils de configuration. Il existe de nombreuses distributions, chacune ayant ses particularités : certaines sont dédiées à un usage spécifique (pare-feu, routeur, grappe de calcul...), d'autres à un matériel spécifique.

Les distributions généralistes les plus connues sont Debian, Gentoo, Mandriva Linux, Red Hat/Fedora, Slackware, Novell SuSE, Ubuntu. Il faut noter que certaines distributions sont commerciales, comme celles de Red Hat, Mandriva (ex-MandrakeSoft) ou de Novell-SuSe, alors que d'autres sont l'ouvrage d'une fondation à but non lucratif comme Gentoo et Debian.

Linux et l'immense partie des logiciels contenus dans une distribution sont libres, mais libre ne veut pas dire gratuit. Lorsque l'on achète une distribution Linux, le prix payé est celui du média, de la documentation incluse et du travail effectué pour assembler les logiciels. Toutefois, pour respecter l'esprit du Libre et se conformer aux exigences des licences utilisées par ces logiciels, les entreprises qui éditent ces distributions les rendent disponibles par téléchargement sans frais.

2.3 Fonctionnalités du noyau

Le noyau est constitué d'un ensemble de gestionnaires qui prennent en charge :

- Le contrôle des processus : assure la création, la synchronisation, la terminaison (cf. chapitre 3), et les communications inter-processus (cf. chapitres 5 et 6).
- L'allocation du temps processeur : assurée par l'ordonnanceur (scheduler — cf. chapitre 9).
- La gestion de fichiers (cf. chapitre 4) : allocation d'espace, contrôle des accès, sauvegarde des données. Implémentation de système de tampon entre le noyau et les périphériques.
- La gestion de la mémoire (cf. chapitre 11) : gestion de l'allocation statique et dynamique de mémoire.
- Les pilotes de périphériques (cf. chapitre 8) : gestion des E/S entre le noyau et les périphériques.

2.3.1 Modules noyau

Le noyau Linux n'est plus un gros bloc de code monolithique comme pouvaient l'être les premiers noyaux UNIX. Il est possible d'y ajouter ou d'y enlever du code dynamiquement par le biais de modules. Cela évite d'avoir un noyau énorme contenant le code pour tous les types de composants,

[2] Définition Wikipedia

même ceux qui ne sont pas utilisés par l'ordinateur de l'utilisateur. Par ailleurs, un autre avantage est qu'il n'est pas nécessaire de recompiler le noyau à chaque fois que l'on désire ajouter une fonctionnalité, ou le support pour du matériel supplémentaire. Enfin, il n'est pas non plus nécessaire de relancer la machine à chaque modification, ce qui est un élément essentiel pour la stabilité d'une machine utilisée en serveur par exemple. Coté programmation, cela rend le code du noyau plus modulaire et plus facile à maintenir. Les modules on été introduits en 1995 pour la sortie du noyau 1.2.

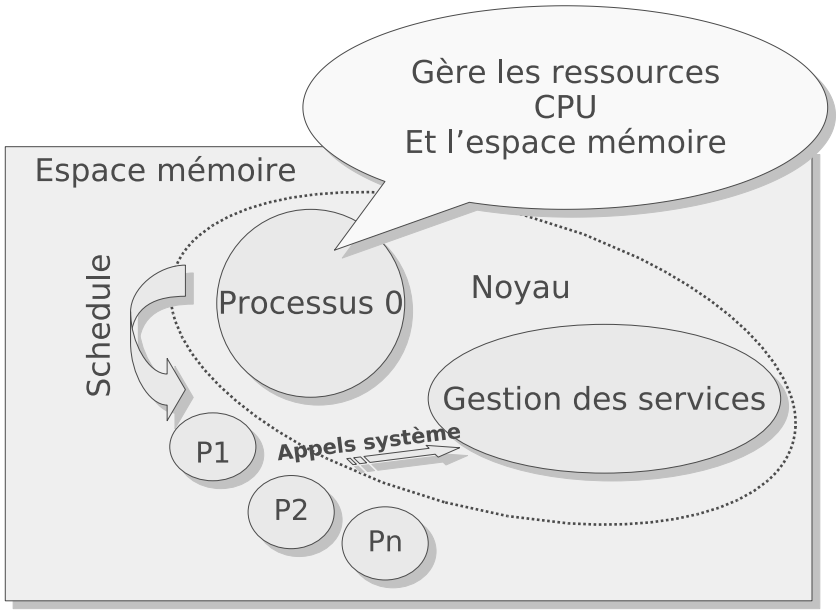
Les modules servent essentiellement à implémenter des pilotes (drivers) pour le matériel. Mais ils servent aussi à implémenter des services dont le code doit être exécuté avec des droits privilégiés, en mode noyau (systèmes de fichiers par exemple). D'un point de vu pratique, le noyau doit inclure les fonctionnalités les plus utilisées et peut laisser dans des modules chargeables tout le reste. Le choix des modules à intégrer peut se faire lors de la compilation du noyau, mais ils peuvent aussi être intégrés dynamiquement après le démarrage du système.

La programmation des modules noyau sera présentée au chapitre 8.

2.4 Contexte "utilisateur" et contexte "noyau"

Dans Linux, la mémoire vive est divisée en deux parties indépendantes (et ce matèriellement depuis les processeurs de la génération de l'Intel 386 avec lequel Linus a développé son système) : l'espace noyau et l'espace utilisateur. Ceci permet plus de sécurité : les applications de l'espace utilisateur ne peuvent ni accidentellement ni intentionnellement accéder à une zone mémoire ne leur appartenant pas.

Le passage du mode utilisateur au mode noyau peut en fait être dû à trois événements particuliers :
- Il y une interruption matèrielle (horloge, périphérique...).
- Il y a eu une anomalie (procédure de déroutement - exception).
- Le processus fait explicitement appel à des services du noyau par un **appel système** (le noyau rend les services en s'exécutant sous l'identité du demandeur).



2.4.1 Gestion de la mémoire

Adresse physique

L'adresse physique détermine l'emplacement d'un élément dans la mémoire vive. Le stockage étant continu dans la mémoire, une adresse physique prend ses valeurs entre 0 et la capacité totale de la RAM installée dans le système.

La commande `cat /proc/iomem` permet d'afficher le mapping en adresses physiques des périphériques dont une machine est équipée (cette sortie dépend du hardware de chaque machine).

```
00000000-0009fbff : System RAM
  00000000-00000000 : Crash kernel
000a0000-000bffff : Video RAM area
000c0000-000cc7ff : Video ROM
000ce800-000cffff : Adapter ROM
000f0000-000fffff : System ROM
00100000-e7e0abff : System RAM        /* Cette machine est équipée de 4 Go */
  00100000-00319256 : Kernel code
  00319257-003c5c63 : Kernel data
e7e0ac00-e7e0cbff : ACPI Non-volatile Storage
e7e0ec00-e7e5cbff : reserved
e7e5cc00-e7e5ebff : ACPI Tables
e7e5ec00-e7ffffff : reserved
e8000000-efffffff : PCI Bus #01
  e8000000-efffffff : 0000:01:00.0
    e8000000-efffffff : vesafb
f0000000-f3ffffff : reserved
f7e00000-f7efffff : PCI Bus #04
  f7ef0000-f7efffff : 0000:04:00.0
    f7ef0000-f7efffff : tg3
f7f00000-f7ffffff : PCI Bus #02
f8000000-feafffff : PCI Bus #01
  f8000000-fbffffff : 0000:01:00.0
    f8000000-fbffffff : nvidia
  fd000000-fdffffff : 0000:01:00.0
  fea00000-fea1ffff : 0000:01:00.0
febfc000-febfffff : 0000:00:1b.0
  febfc000-febfffff : ICH HD audio
fec00000-fed003ff : reserved
fed20000-fed9ffff : reserved
feda0000-fedacfff : pnp 00:0a
fee00000-feefffff : reserved
ff970000-ff9703ff : 0000:00:1f.2
  ff970000-ff9703ff : ahci
ff980800-ff980bff : 0000:00:1d.7
  ff980800-ff980bff : ehci_hcd
ffb00000-ffffffff : reserved
100000000-113ffffff : System RAM
```

Adresse logique

Pour faciliter la vie des programmeurs, le système d'exploitation masque l'accès à la mémoire physique aux programmes qu'ils exécutent. L'adresse logique est l'adresse utilisée au sein d'un programme par le concepteur du système ou des programmes qui sont exécutés dessus.

Vue depuis un programme en mode utilisateur en cours d'exécution la mémoire est ainsi exprimée selon des adresses logiques se traduisant dans un espace virtuel continu appelé **adresses linéaires**.

Cet espace d'adressage possède les propriétés suivantes :
- il commence à l'adresse 0.
- il est continu.
- il est privatif (les autres programmes n'y ont pas d'accès).

Le fait d'avoir un espace d'adressage continu permet de simplifier les opérations sur les adresses (arithmétique des pointeurs). Il permet par exemple de balayer un tableau simplement en incrémentant une adresse. Le fait qu'il soit privatif permet d'assurer qu'aucun autre programme ne peut y accéder (cf. paragraphe 2.4.1 sur la protection mémoire).

L'espace d'adressage virtuel est un espace linéaire de 4 Go (32 bits d'adressage), même si la mémoire physique est de taille très inférieure. Une adresse linéaire est donc comprise entre 0 et `0xffffffff`. C'est dans cet espace continu virtuel que sont juxtaposés les morceaux de codes et de données qui constituent les éléments actifs du système d'exploitation et des applications utilisateur. Une réserve de

1Go de mémoire est affectée au noyau. Ainsi, l'espace `0x00000000` à `0xbfffffff` est dédiée à l'espace utilisateur, alors que la zone `0xc0000000` à `0xffffffff` est dédiée à l'adressage de l'espace noyau ([Corbet et al., 2005] — p 71).

C'est le système de gestion de la **mémoire virtuelle** (avec ou sans l'aide d'un gestion matériel appelé **MMU** — Memory Management Unit) qui permet de faire le lien entre la mémoire physique et les adresses virtuelles. Pour ce faire, Linux utilisent 2 stratégies : la segmentation et la pagination. Grâce à ces 2 mécanismes, il est possible d'avoir une partie de l'espace d'adressage linéaire en mémoire physique, et le reste de l'espace d'adressage linéaire sur une mémoire de masse (généralement un disque dur), ce qui démultiplie d'autant la mémoire adressable par le système d'exploitation (voir chapitre 11).

Principes de protection de la mémoire

Le mode protégé ou superviseur est un des modes d'exploitation des processeurs actuels qui aide matériellement la conception d'un système d'exploitation. Il existe au moins deux modes de fonctionnement différents : un mode réel ou utilisateur, et un mode protégé ou superviseur (exploité par Linux dès la version 0.0.1 — version fonctionnant sous 80386).

Le mode "protégé" des processeurs a été développé pour permettre de limiter les bugs ou les programmes malveillants. Il a par ailleurs aussi été développé pour faciliter la gestion d'un système multi-tâches (voir le chapitre 9 pour les détails sur ce sujet).

En mode protégé, les programmes sont soumis à des règles strictes en termes d'utilisation de la mémoire qui déclenche une exception, c'est à dire une interruption déclenchée directement par le processeur, dès que le programme tente de sortir de l'espace mémoire qui lui est réservé. Il n'est ainsi pas possible de lire des portions de mémoires réservées à d'autres programmes ou au système d'exploitation. Il n'est pas non plus possible de faire des E/S sur certains ports réservés à l'accès au matériel, ce qui pourrait mettre celui-ci dans un état instable et provoquer un dysfonctionnement du système. L'exception générée est en fait interceptée par le système d'exploitation, qui prend les mesures nécessaires. Ainsi, si une des tâches mène à une erreur de protection mémoire, le système d'exploitation peut la fermer, et le système reste fonctionnel.

Les mécanismes de protection doivent ainsi permettre à un système d'exploitation multi-tâches :

- d'éviter qu'un programme rentre dans un état instable
- d'éviter qu'un programme ne provoque une corruption du système et le rende instable.
- d'éviter qu'un programme n'accède au matériel et génère une instabilité du système et/ou de ses périphériques

En fonctionnement "normal", un programme n'a aucune raison de dépasser les limites fixées par le système. Si cependant cela se produit, il s'agit sans doute d'un bug qui met en péril la stabilité du système, ou bien d'un programme mal intentionné qui tenterait délibérément de corrompre le fonctionnement du système.

Les programmes ne peuvent pas accéder à l'espace mémoire dépendant du système. Il est donc impossible qu'un programme écrive de façon anarchique dans une zone où le système stocke des données importantes. A l'inverse, il faut absolument que le système puisse avoir accès à l'espace d'adressage des programmes utilisateur. Prennons un exemple : la fonction de lecture d'un bloc de données à partir d'un fichier doit pouvoir stocker les octets dans une zone de données du programme qui l'a appelée. Le système doit donc avoir accès à l'espace d'adressage du programme utilisateur. Le processeur distingue principallement 2 niveaux de privilège :

- Le mode privilégié ou superviseur. Il est réservé au fonctionnement du noyau. C'est le seul niveau habilité à exécuter certaines instructions critiques, telles que le chargement des informations sur la structuration physique de la mémoire ou la commutation de tâche.

– Le niveau non-privilégié est le niveau de tout les programmes non-systèmes.

Avec un tel découpage, un programme exécuté à un certain niveau de privilège ne peut pas accéder aux données du niveau le plus privilégié. Ceci exclue qu'un virus, qui est un programme non système, donc de niveau de privilège "normal", d'aller se greffer dans un partie du code du système d'exploitation, de privilège élevé.

Un programme ne peut pas non plus appeler directement du code plus privilégié que lui. Si cela était possible sans restriction, un saut incontrollé en mémoire pourrait pointer sur le code du noyau et fasse dérailler le système du fait de données inconsistantes.

Il est bien sur possible d'éxécuter du code noyau à partir de l'espace utilisateur, mais le mécanisme nécéssite un certain nombre de contrôles. Sous Linux, l'exécution de code en mode noyau s'effectue via les appels systèmes.

Chaque programme, quel que soit son niveau de privilège, que ce soit le système, un pilote d'imprimante ou une application, possède son segment[3] de code, son segment de données et son segment de pile.

En pratique, chaque zone mémoire possède un champ qui décrit le niveau de privilège requis pour le code pouvant y accéder (sous 80x86, on parle de DPL qui signifie **Descriptor Privilege Level**). Par exemple, si le niveau de privilège indiqué est 0, alors les zones de code de privilège 1 n'y ont pas accès. Le niveau de privilège courant (parfois appelé CPL, pour **Current Privilege Level**) est placé dans un registre spécial du processeur.

Lorsqu'on tente de charger une adresse pointant sur un segment particulier, le processeur vérifie si le segment auquel on tente d'accéder n'est pas plus privilégié que le segment de code courant.

Si l'on tente de charger un segment plus privilégié que le segment de code contenant cette instruction de chargement ($CPL > DPL$), le processeur déclenche une exception de protection générale. Cette vérification est effectuée même si le code est en espace utilisateur et effectue un appel système. Dans ce cas, le processeur conserve une trace des privilèges du code appelant afin de pouvoir effectuer le test d'accès (On parle de privilège effectif – EPL –**Effective Privilege Level**).

Il est intéréssant de noter que le chargement avec un pointeur nul ne déclenche pas d'exception immédiatement. Par contre, il est impossible d'accéder à la mémoire à travers lui (tout programmeur a évidemment déjà été confronté à ce "bug" !).

Un autre point à remarquer est que les segments de code peuvent être exécutables mais pas lisibles, ce qui garantit une certaine confidentialité au code.

Il est aussi possible de protéger des segments contre l'écriture, si par exemple ils pointent vers des zones de données sensibles mais auxquelles on a besoin d'accéder.

Un dernière vérification est effectuée lors de l'accès aux données : celle de vérification de la limite. En pratique, elle est stockée dans le descripteur de la zone mémoire. Ce contrôle n'intervient que lors des accès à la mémoire. L'offset de l'accès est comparé avec la limite. S'il la dépasse (attention : dans le cas de segments de piles la croissance vers le bas), un exception de protection est déclenchée. Cette limite est là pour 2 raisons :

1. Pour empêcher qu'un programme ne déborde de son segment suite à une erreur et écrase un bout de la mémoire avec ses données (par exemple, s'il accède à un tableau de données, et qu'il y a une erreur lors du calcul de l'index).
2. Pour éviter qu'un programme ne puisse lire ou écrire la totalité de la mémoire, et donc le code du système d'exploitation ou d'autres programmes.

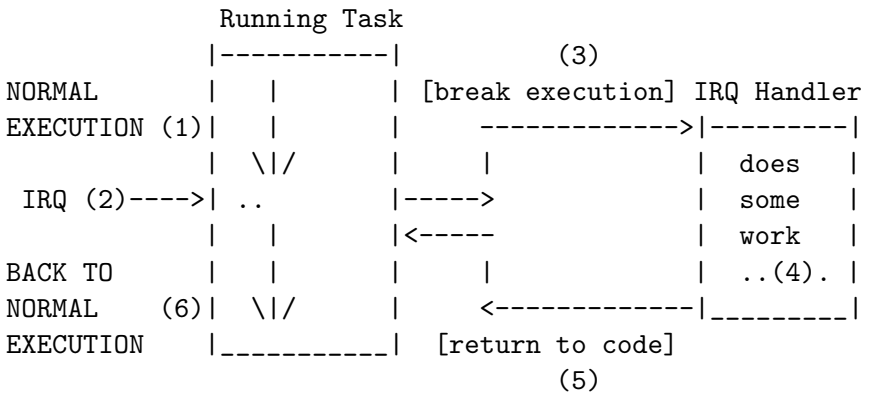
Cette vérification est fondamentale pour le système de protection de la mémoire.

[3] Attention ici, on entend par segment "section de programme" réservée au code/aux données/à la pile/. En effet, un autre acception du terme existe dans le cas du 80x86 et désigne un accès particulier à la mémoire – voir A

2.4.2 Gestion des interruptions

Les **interruptions** sont des signaux matériels qui sont déclenchés par un périphérique matériel quelconque. Elles peuvent donc survenir à n'importe quel moment. On parle alors d'IRQs (IRQ signifie Interrupt ReQuest). La partie logicielle permettant de répondre à ces requêtes émises par le matériel est appelée pilote de périphérique. Une interruption peut aussi être déclenchée par une instruction INT (interruption logicielle). On vera par exemple plus loin que Linux utilise une interruption logicielle pour implémenter les appels systèmes.

Une **exception** est la réponse prédéfinie du processeur à une instruction incorrecte survenant dans l'exécution du code en cours. C'est une interruption générée par le périphérique processeur lui-même. Le processeur déclenche des interruptions pour signaler une erreur dans le code qu'il exécute, comme une violation de protection, ou une instruction non valide. La raison fondamentale de leur existence est la "récupération" des erreurs des programmes par le système d'exploitation. Un OS installe des gestionnaires d'exceptions pour empêcher qu'un programme buggé fasse buggé l'ensemble du système.



User->Kernel Mode Transition caused by IRQ event

Lorsqu'une interruption survient, le contrôle est passé à une routine du noyau chargée de prendre les mesures appropriées. Ce transfert de contrôle peut se faire via une **porte d'interruption** . Les portes permettent le changement du niveau de privilège courant et c'est l'une de leurs raisons d'être. Imaginons par exemple que le contrôleur du disque dur génère une interruption indiquant qu'il a terminé la lecture des données. Le programme en cours d'exécution est interrompu, et le contrôle est transféré à une routine du système, qui est chargée de prendre les mesures ad hoc. Il y a donc un changement du niveau de privilège, puisque le programme interrompu est en mode utilisateur et que le pilote du disque dur est en mode noyau. Le pilote de disque dur va s'exécuter tranquillement, puis, quand il aura fini, il fera un retour d'interruption, qui reviendra au programme interrompu, et donc retournera en mode utilisateur.

On confond parfois l'IRQ et l'interruption qui lui est associée, alors qu'en fait, les IRQs sont des signaux qui transitent par le bus et qui indiquent au processeur qu'un composant matériel a quelque chose à dire. En fait, les IRQs ne transitent pas directement des composants matériels au processeur, mais passent par le(s) contrôleur(s) PIC (Programmable Interrupt Controler). Ce (ou ces) contrôleurs sont chargés :

1. de vérifier qu'il n'y a pas "collision" d'IRQ. Une IRQ est mise en attente si elle survient pendant l'exécution de la routine d'interruption d'une autre IRQ.
2. de générer le numéro d'interruption correspondant à une IRQ précise.

Le(s) PIC(s) peuvent de plus :
– Indiquer si la routine d'interruption d'une IRQs est en cours d'exécution.
– Interdire à certaines IRQs de déclencher des interruptions.
– Changer les numéros d'interruptions déclenchées pour des IRQs données.
A moins qu'elles ne soient ainsi bloquées[4], les IRQs peuvent survenir à n'importe quel moment, selon l'état du matériel. Donc des interruptions peuvent être déclenchées n'importe quand. Il faut donc qu'en permanence, tout soit prêt pour les accueillir.

La commande `cat /proc/interrupts` permet d'afficher les interruptions gérées par un processeur et le compteur qui lui est associé.

```
           CPU0
  0:    1982099   IO-APIC-edge      timer
  1:      15188   IO-APIC-edge      i8042
  9:          4   IO-APIC-fasteoi   acpi
 12:        367   IO-APIC-edge      i8042
 14:     273216   IO-APIC-edge      libata
 15:     107092   IO-APIC-edge      libata
 16:    1051765   IO-APIC-fasteoi   uhci_hcd:usb1, Intel ICH6, eth0, radeon@pci:0000:01:00.0
 17:     101875   IO-APIC-fasteoi   uhci_hcd:usb4, yenta
 18:     266987   IO-APIC-fasteoi   uhci_hcd:usb2, ipw2200
 19:    2071003   IO-APIC-fasteoi   uhci_hcd:usb3
NMI:          0   % NMI : Non Masquable Interrupt
LOC:    3136252   % LOC : Interruption du timer de chaque CPU
ERR:          0   % ERR : erreurs commises sur le bus IO-APIC
MIS:          0   % MIS : Mauvais fonctionnement du mode level
```

Gestion du temps

Parmi les interruptions, celle qui a évidemment le plus de d'importance est le **timer**. Cette interruption est en effet indispensable à l'ordonnancement des tâches, mais elle permet aussi de gérer de nombreux procédures nécessitant des répétitions régulières.

L'un des éléments interessant à connaitre concernant la gestion du temps sous Linux est l'existence de la variable globale `jiffies` qui compte le nombre de d'interruption d'horloge (ticks) depuis le démarrage du système[5]. La constante `HZ`[6] quant à elle définit le nombre de ticks par seconde.

Cette interruption est fournie par un circuit spécial, le PIT 8253 (Programmable Interval Timers) qui est programmé par défaut pour délivrer une interruption toute les 10 ms[7]. Elle transite ensuite via la PIC et correspond au numéro d'intteruption numéro 0 (plus forte priorité), cependant, dans Linux, toutes les interruptions matérielles (dont le timer) sont translatées à partir du numéro d'interruption `20h`.

Dans la version 0.0.1 de Linux, cette interruption est gérée par le controleur d'interruption `timer_interrupt`, associé donc à l'interruption `20h` qui appelle elle-même la fonction `do_timer`.

```
void do_timer(long cpl)
{
  if (cpl)
    current->utime++;
  else
    current->stime++;
  if ((--current->counter)>0) return;
  current->counter=0;
  if (!cpl) return;
  schedule();
}
```

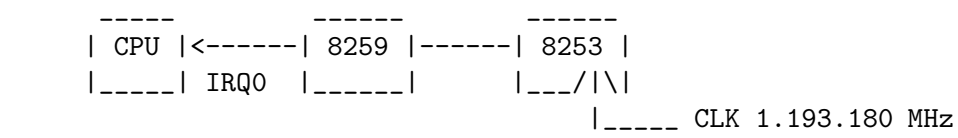
[4]Les interruptions de type NMI, comme leur nom l'indique – Non Masquable Interrupt – ne sont pas masquables. Le processeur ne peut pas ignorer ce signal qui est normalement utilisé pour détecter des erreurs matérielles sévères (mémoire principale défaillante par exemple).

[5]son nom vient de l'expression anglaise "wait a jiffy"

[6]qui est définie dans `include/asm/param.h`

[7]Cependant, il est possible de changer cette valeur en modifiant la valeur de la constante `HZ`

Cette fonction prend comme paramêtre le niveau de privilège actuel (CPL). Elle met à jour les variables temporelles propres au processus courant (temps utilisateur – `utime` si CPL!=0, système – `stime` – sinon) et décrémente l'interval temporel qui lui est alloué par l'ordonnanceur. Enfin, si le système est en mode noyau, il appelle le scheduler (dont nous verrons le détail au chapitre 9).



2.4.3 Implémentation des appels systèmes sous Linux

Les appels système sont des fonctions :
– appelées depuis un programme de l'espace utilisateur
– dont l'exécution s'effectue dans l'espace noyau
– dont le retour se fait dans le programme appelant dans l'espace utilisateur.
Le coût d'un appel système est nettement plus élevé qu'un simple appel de fonction. En effet, un appel de fonction ne suppose que quelques instructions consistant à faire pointer le compteur programme sur une zone mémoire particulière. Le coût d'un appel système, lui, demande beaucoup plus de ressources puisqu'il suppose au moins deux commutations de contextes :

1. Contexte du programme appelant
2. Changement de contexte
3. Contexte du noyau
4. Exécution du service demandé
5. Changement de contexte
6. Contexte du programme appelant.

Cela peut ainsi générer une charge supplémentaire "inutile" du point de vue utilisateur. Il est donc nécéssaire de pouvoir optimiser et minimiser le code exécuté en mode noyau. Par ailleurs, un autre moyen pour limiter le nombre de commutations de contexte est de restreindre le nombre d'appels système, mais que ceux-ci fournissent des services puissants.

Les appels systèmes débouchant sur un mode d'accès à la mémoire privilégié, il est absolument nécessaire de contrôler le passage dans ce mode (pour éviter que l'utilisateur usurpe ces droits), et le retour de ce mode (pour éviter de conserver les droits superviseurs).

Passage en mode noyau

A l'initialisation du système, un ensemble de routines de gestion d'interruptions matérielles et d'exceptions sont mises en place. Par ailleurs, la gestion d'une interruption logicielle unique est mise en place (l'interruption $0x80$ sous 80x86). C'est cette interruption logicielle qui est utilisée par Linux pour implémenter le passage en mode noyau.

Chacun des appels systèmes est numéroté. De surcroît, une table, appelée table des appels systèmes, contient la liste des pointeurs de fonctions vers chacune des fonctions implémentant ces appels. L'exécution d'un appel système particulier s'implémente donc en exécutant le code correspondant à la fonction dont l'index est celui de l'appel système désiré.

Le code de la routine de service répondant à l'interruption logicielle récupère le numéro de l'appel passés dans l'un des registres du processeur (eax sous 80x86) ainsi que le ou les paramètres qui lui

correspondent (passés dans d'autres registres — ebx, ecx et edx sous 80x86). Les registres de pile sont sauvegardés. Les registres pointant sur le code et les données sont mis à pointer sur ceux du noyau. On utilise le numéro de l'appel comme index dans la table des appels systèmes. Au retour de l'appel, un code d'erreur est renvoyé sur le sommet de la pile et le numéro du processus en cours d'exécution dans un registre (eax sous 80x86). On revient de l'interruption, ce qui fait passer le processus à nouveau en mode utilisateur, avec les droits afférents.

Exécution de l'appel depuis le mode utilisateur

Les fonctions d'appels au système sont disponibles dans la bilbliothéque glibc à partir de plusieurs macros spécifiques en fonction du nombre d'arguments.

Sous linux 0.0.1, la macro correspondant à une fonction d'appel sans argument s'écrit :

```
#define _syscall0(type,name) \
type name(void) \
{ \
type __res; \
__asm__ volatile ("int \$0x80" \
  : "=a" (__res) \
  : "0" (__NR_##name)); \
if (__res >= 0) \
  return __res; \
errno = -__res; \
return -1; \
}
```

Cela revient simplement à déclencher l'interruption $0x80$ après avoir passé le numéro de l'appel dans `eax`. Au retour, on inspecte ce même registre pour tester s'il y a eu une erreur que l'on affecte le cas échéant à `errno` (la variable global de gestion des erreurs).

Ainsi l'appel `ftime()` est généré en faisant :

```
_syscall0(int,sys_ftime)
```

__NR_sys_ftime étant égal à 35.

2.5 Quiz

Questions de cours :

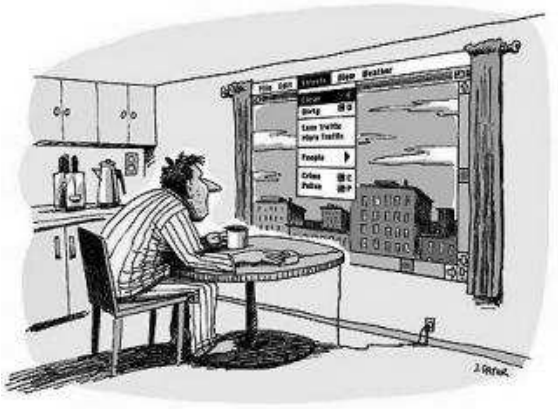
1. Quels sont les fonctionnalités de base d'un système d'exploitation ?
2. Quels sont les spécificités du système d'exploitation Linux ?
3. Qu'est-ce que le Kernel ?
4. Quel est son organisation ?
5. Comment accède-t-on aux ressources du noyau ?

Questions ouvertes :

1. Quels sont les raisons du succès de Linux ?
2. Connaissez-vous d'autres systèmes d'exploitations ?
3. Quels sont les Unix existants ?
4. Pourquoi les sytèmes actuels préfèrent le multi-threading au multi-process ?
5. Unix est-il temps-réel ? Pourquoi ?

Chapitre 3

Gestion des processus



L'une des tâches principales du noyau est de gérer les ressources de la machine en partageant le temps CPU entre les différents programmes lancés sur cette machine. Par ailleurs, le noyau doit aussi gérer la création et la terminaison de ces programmes.

3.1 Fonctionnement d'un processus

3.1.1 Définition d'un processus

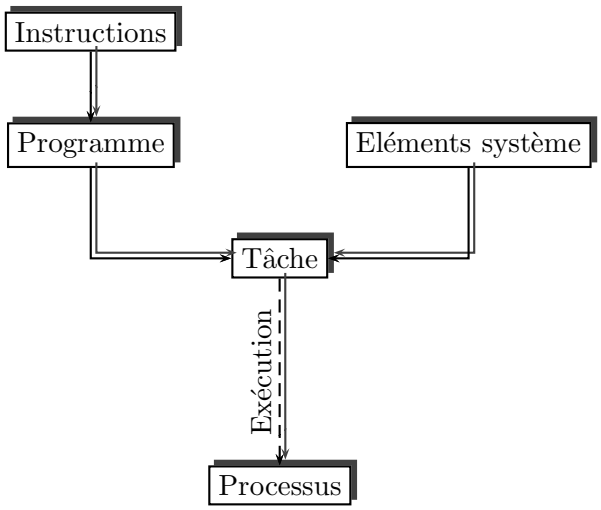


FIG. 3.1 – Notion de processus

Pour définir la notion de processus, nous approcherons le concept de manière graduelle :

1. Un programme est un ensemble d'instructions
2. Une tâche est un programme pourvu de la structure système nécessaire à son exécution.
3. Un processus est une tâche en cours d'exécution.
4. Une tâche est *statique* alors que le processus est *dynamique*.
5. Le **vecteur d'état** d'un processus est constitué de l'ensemble des variables et des procédures utilisables par ce processus. Ce vecteur d'état est sauvegardé lors de la commutation vers un autre processus (voir détails dans le chapitre 9).

3.1.2 Structures associées

A chaque processus est associé un ensemble de contextes qui sont :
– Le contexte Utilisateur (vue du processus par l'utilisateur)
– Le contexte Système (vue du processus par le noyau)
Dans chacun des ces contextes, un ensemble de structures particulières permet d'assurer une cohérence et pertinence du point de vue recherché.

Du point de vue du contexte utilisateur, la mémoire est fonctionnellement décomposée en 3 segments :
– Le segment de code (ou text).
– Le segment de données.
– Le segment de pile (en anglais stack).

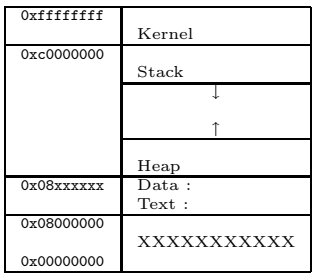
Le segment de données est lui-même décomposé en 3 sous-régions :
– La zone des variables statiques initialisées (déclarées dans le code par `static int a=0;`).
– La zone des variables statiques non-initialisées, ou reservation (en anglais BSS = Block Start Symbol). Code équivalent `static int b;`
– La zone où les variables sont allouées dynamiquement (`int *c=(int *)malloc(sizeof(int));`). Ce qu'on appelle le tas (heap en anglais). Le tas croît dans le sens des adresses positives lorsque des allocations de mémoire sont faites.

Indications des zones mémoire avec GCC

Le compilateur GNU GCC fournit un certains nombre de symboles qui identifient les emplacements des différentes zones mémoires.
La commande ***nm*** *permet de visualiser ces symboles :*
__bss_start *: indique le début de la zone BSS*
__data_start *: indique le début de la zone de données*
_edata *: indique la fin de la zone de données*
_end *: représente la fin de la mémoire accessible par l'utilisateur.*

La zone de pile est l'endroit dans lequel on stocke :
– Les variables locales d'un programme
– Les paramètres de fonctions
– L'adresse de retour des fonctions
La pile croît dans le sens des adresses négatives lorsque l'on empile des données.

Le schéma ci-desssous montre l'espace d'adressage d'un programme. La zone Kernel est la zone mémoire où se trouve le noyau du système d'exploitation. Comme on l'a vu, le programme n'y a pas accès. La zone XXXXXXX est réservée et non utilisable.



Pour plus de détails voir annexe B.

La commande `cat /proc/self/maps` permet d'afficher l'espace d'adressage du programme en cours (self), c'est à dire du cat :

```
08048000-0804c000 r-xp 00000000 08:05 32593      /bin/cat      /* Zone de code */
0804c000-0804d000 rwxp 00004000 08:05 32593      /bin/cat      /* Zone de données */
0804d000-0806e000 rwxp 0804d000 00:00 0          [heap]        /* Tas */
40000000-40019000 r-xp 00000000 08:05 179180     /lib/ld-2.6.1.so
40019000-4001a000 r-xp 00018000 08:05 179180     /lib/ld-2.6.1.so
4001a000-4001b000 rwxp 00019000 08:05 179180     /lib/ld-2.6.1.so
4001b000-4001c000 rwxp 4001b000 00:00 0
4001c000-4001d000 r-xp 00000000 08:05 310258     /usr/share/locale/fr_FR.UTF-8/LC_IDENTIFICATION
4001d000-40024000 r-xs 00000000 08:05 293550     /usr/lib/gconv/gconv-modules.cache
40024000-40025000 r-xp 00000000 08:05 310333     /usr/share/locale/fr_FR.UTF-8/LC_MEASUREMENT
40025000-40026000 r-xp 00000000 08:05 310289     /usr/share/locale/fr_FR.UTF-8/LC_TELEPHONE
40026000-40027000 r-xp 00000000 08:05 310291     /usr/share/locale/fr_FR.UTF-8/LC_ADDRESS
40027000-40028000 r-xp 00000000 08:05 310331     /usr/share/locale/fr_FR.UTF-8/LC_NAME
40028000-40029000 r-xp 00000000 08:05 310329     /usr/share/locale/fr_FR.UTF-8/LC_PAPER
40029000-4002a000 r-xp 00000000 08:05 310313     /usr/share/locale/fr_FR.UTF-8/LC_MESSAGES/SYS_LC_MESSAGES
4002a000-4002b000 r-xp 00000000 08:05 310290     /usr/share/locale/fr_FR.UTF-8/LC_MONETARY
4002b000-4002c000 r-xp 00000000 08:05 310288     /usr/share/locale/fr_FR.UTF-8/LC_TIME
4002c000-4002d000 r-xp 00000000 08:05 310173     /usr/share/locale/en/LC_NUMERIC
40032000-4016c000 r-xp 00000000 08:05 184194     /lib/i686/libc-2.6.1.so
4016c000-4016d000 r-xp 00139000 08:05 184194     /lib/i686/libc-2.6.1.so
4016d000-4016f000 rwxp 0013a000 08:05 184194     /lib/i686/libc-2.6.1.so
4016f000-40173000 rwxp 4016f000 00:00 0
40173000-40257000 r-xp 00000000 08:05 309653     /usr/share/locale/UTF-8/LC_COLLATE
40257000-4028d000 r-xp 00000000 08:05 309525     /usr/share/locale/ISO-8859-15/LC_CTYPE
bff28000-bff3e000 rw-p bff28000 00:00 0          [stack]       /* Zone de pile */
ffffe000-fffff000 r-xp 00000000 00:00 0          [vdso]
```

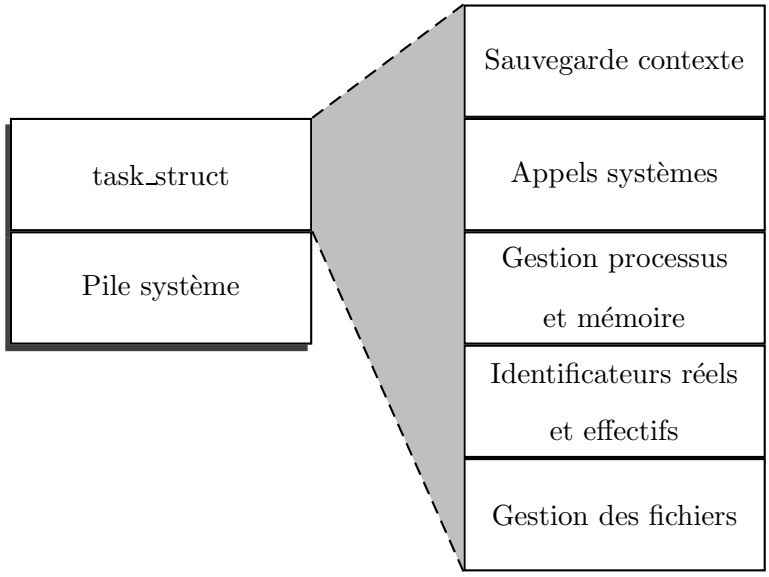


FIG. 3.2 – Contexte système

Du point de vue du système le processus est manipulé via 2 structures :

- La `task_struct` (sorte de carte d'identité du processus)
- La pile système associée à ce processus

La `task_struct` est de loin la structure qui évolue le plus au fil et à mesure des évolutions du système. D'une structure de 29 lignes de long à l'origine (noyau 0.0.1), nous sommes passés maintenant à 261 lignes sous le kernel 2.6.22.18 que j'utilise pour rédiger ces lignes !

Dans les versions antèrieures au noyau 2.4, un tableau à 4090 entrée était réservé pour contenir les descripteurs de chacun des processus s'exécutant sur le système (table `struct task_struct * task[NB_TASKS]`).

Afin d'éviter cette limite statique au nombre de processus, cette structure a été remplacée depuis le noyau 2.4 par une liste doublement chaînée. Par ailleurs, il est aussi possible d'accèder aux processus en utilisant une table de hachage avec pour clef de hachage le PID du processus.

En conséquence, le nombre maximum de processus qu'un système Linux peut gérer dépend maintenant uniquement de la mémoire disponible sur le système.

Détails d'implémentation

Le nombre maximum de thread est donnée par la formule (voir la fonction `void __init fork_init(unsigned long mempages)` *du fichier* `kernel/fork.c`*) :*

$$max\_threads = \frac{mempages}{8 * \frac{THREAD\_SIZE}{PAGE\_SIZE}} \tag{3.1}$$

Avec $PAGE\_SIZE = 4096$ *(*`include/ asm/ page.h`*) et* $THREAD\_SIZE = 8192$ *(*`include/ asm/ thread_info.h`*). La fonction* `void __init fork_init(unsigned long mempages)` *est appelée dans* `start_kernel(void)` *par* `fork_init(num_physpages)` *(voir* `init/ main.c`*).* `num_physpages`*, représente le nombre maximum de pages physiques accessibles et est calculée par* `void __init find_max_pfn(void)` *(voir* `arch/ i386/ kernel/ e820.c`*).*

La valeur de `threads-max` peut être visualisé et modifiée dans `/procfs` :

```
#cat /proc/sys/kernel/threads-max
16381
#echo 12345 > /proc/sys/kernel/threads-max
```

3.1.3 Vie et mort d'un processus

Les processus se créent par "mitose" à partir d'un processus initial. Soit un processus A implémentant un programme P1. Ce processus possède une zone de code, une zone de données, ainsi qu'une zone de pile. Par ailleurs, une task_structA lui est associée. Pour créer un nouveau processus, il suffit que le processus A fasse appel à l'appel système `fork()`. Dans ce cas, le système recopie le code, les données et la pile du process A, alloue une nouvelle place dans la table des processus à laquelle correspond le numéro du nouveau processus B créé. Par ailleurs, la task_struct est presque entièrement conservée à l'exception des données propres au nouveau processus (N^o du processus). De fait, les deux processus A et B possèdent des espaces d'adressages physiques différents (même si l'adressage linéaire n'a peut-être pas changé)[1].

Le processus qui a lancé la commande fork est appelé *processus père*, le processus créé est le *processus fils*. Après appel du fork, le processus père peut récupérer le numéro de PID de son fils et le fils, peut récupérer le numéro de PID de son père par la commande `getppid()`. Grâce à la valeur retournée par le fork (numéro du PID du fils dans le père, 0 dans le fils), il est possible de savoir quel processus est actif et par conséquent, d'exécuter le code propre au processus. Cependant, cela demande de dupliquer une partie de code (code du fils dans le père et vice-versa). Il est alors possible de remplacer les segments de code, de données et de pile du fils par un autre programme grâce à la commande `exec`.

En fonctionnement normal, lorsqu'un processus lance un processus fils, il continue de s'exécuter, puis se met à attendre la mort de ce processus fils grâce à la commande `wait(&status)` ou `waitpid(pid, &status, options)`, pour attendre le processus *pid* en particulier. Il existe alors deux modes de fonctionnement :

– Le processus fils se termine correctement en mettant un terme à son exécution par la commande `exit(n)`[2]. Le fils envoie un signal **SIGCLD** à son père pour le réveiller (le détail du traitement des

[1] En réalité, tant que l'un des processus n'effectue pas en accès en écriture dans la zone de données ou de pile, l'adresse physique elle-même est conservée – c'est le mécanisme du "copy on write" – voir aussi 11.0.2

[2] n étant un entier quelconque pouvant servir au père pour savoir dans quel état est mort le fils

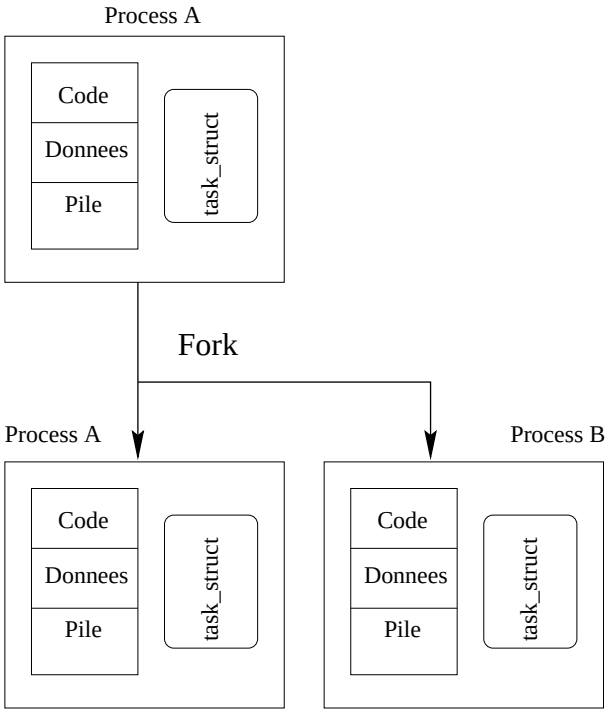


FIG. 3.3 – fork() d'un processus

signaux sera vu au chapitre 5). Les 8 bits de poids fort de la valeur retournée dans status permettent de récupérer l'entier envoyé en paramètre dans le exit.

– Le processus est interrompu par un signal qui lui est envoyé et qu'il ne peut intercepter. Le fils meurt et envoie un signal **SIGCLD** à son père pour le réveiller. Les 7 bits de poids faible de la valeur retournée dans status permettent de récupérer le numéro du signal qui a mis fin à l'exécution du processus fils. Le 8^{eme} bit indique s'il y a eu génération d'une image mémoire ou non (`core`).

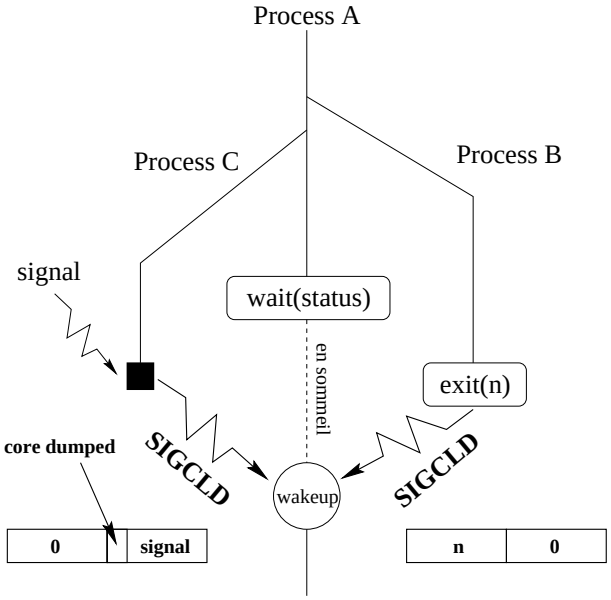


FIG. 3.4 – Fonctionnement des appels systèmes `wait` et `exit`

Pour résumer, les différents états d'un processus sont les suivants :

– En cours de création : état inconsistant au moment de l'appel système fork uniquement.
– Actif : détient les ressources du CPU.
– Endormi : En attente d'événement (I/O par exemple, ou wait). Si celui-ci survient, le processus s'active.
– Activable : le processus est éligible dans la liste des processus qui peuvent être préempté par le scheduler.
– Zombie : Exécution terminée ; Les ressources sont restituées (mémoire, disque, etc...). Il doit transmettre son vecteur d'état avant de disparaître. Il occupe toujours une place dans la table des processus.

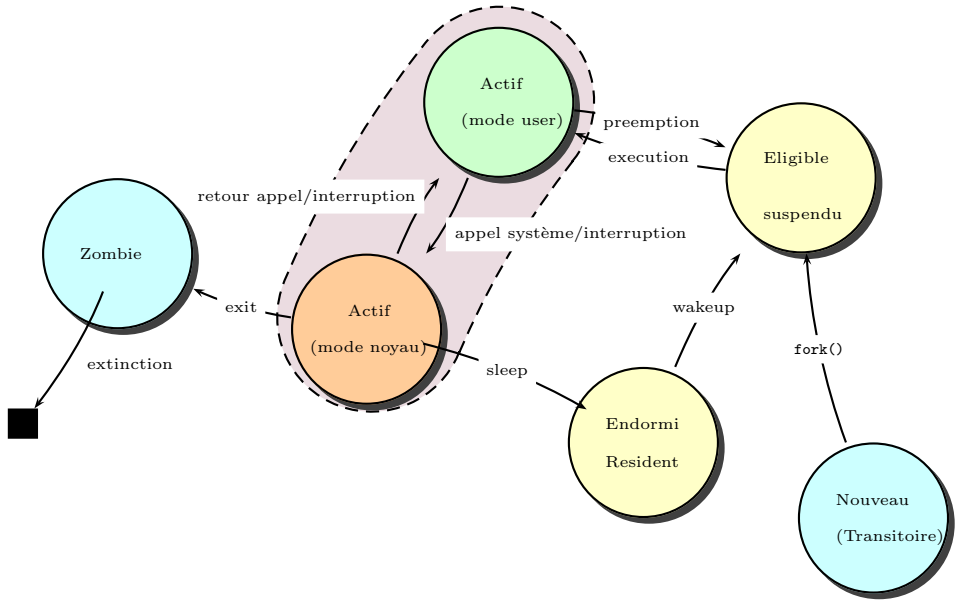


FIG. 3.5 – Etats des processus

3.1.4 Initialisation du système

Nous avons décrit le moyen de créer un processus à partir d'un processus déjà existant. Mais que ce passe-t-il au démarrage du système (*boot*) ?

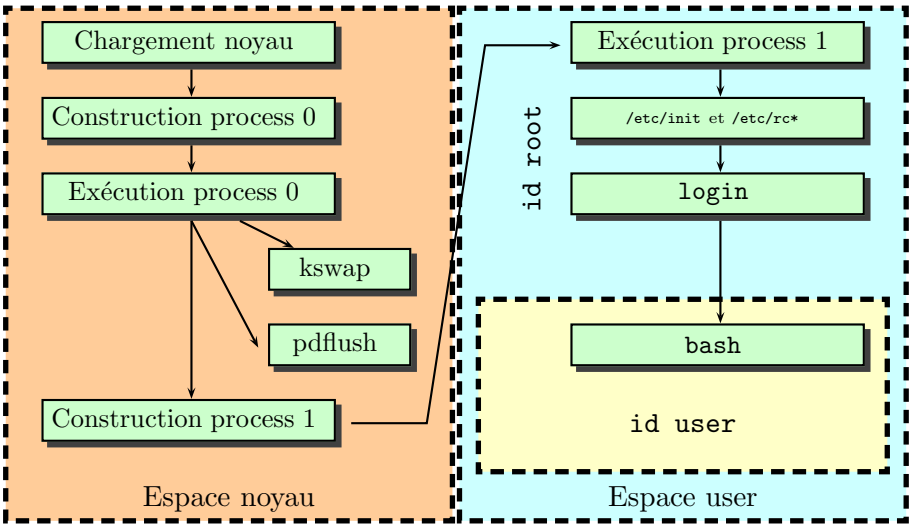
En fait, le système commence à aller chercher des informations sur les premiers blocs du disque de manière à constituer le noyau d'UNIX. Ce noyau est formé, à l'origine, du processus 0 (processus swapper ou idle).

Ce processus 0 n'a ni segment de texte, ni segment de données, ni pile utilisateur. Son code est en zone système et comporte sa task_struct. Il dispose d'une entrée dans la table des processus et d'une pile système.

La séquence de démarrage du système se déroule comme suit :

– Chargement des structures de données du noyau via le boot loader
– Création de l'environnement du processus 0
– /* Processus 0 */
  – mise en place de la segmentation et de la pagination
  – initialisation de la gestion du temps
  – initialisation de la gestion des consoles
  – initialisation des interruptions est exceptions
  – initialisation de l'ordonnanceur

  - initialisation du cache
  - initialisation du disque dur
  - Appel fork pour la création du processus 1
  - boucle sans fin
- /* Processus 1 */
  - Passage en mode utilisateur
  - ...
  - Exécution de /etc/init



3.2 Programmation des processus

Cette section a pour but de donner les instructions C qui permettent de gérer les processus.

3.2.1 Identificateurs d'un processus

Tous les processus possèdent un numéro d'identification unique appelé **PID** (*Process IDentifier*). Le PID du père d'un processus est le **PPID** (*Parent Process IDentifier*). En C, ces identificateurs sont récupérés grace aux fonctions :

```
pid_t getpid(void);
pid_t getppid(void);
```

```
/* Exemple d'utilisation des commandes GETPID et GETPPID */

#include <stdio.h>
#include <stdlib.h>

/* Programme principal */

int main(int argc, char *argv[]) {
  printf("Bonjour ! Je suis %d et mon papa est %d\n",getpid(),getppid
      ());
}
```

3.2.2 Création d'un processus fils

La création d'un processus s'effectue par l'intermédiaire de la fonction :

```
pid_t fork(void);
```

Qui crée un processus dont le PID est renvoyé (-1 si le fork échoue). Un exemple est donné ci-dessous :

```
/* Utilisation de la commande FORK */

#include <stdio.h>
#include <stdlib.h>

/* Programme principal */

int main(int    argc, char *argv[])
{
  int    pidfils;

  if (pidfils=fork())
  {
    printf("%d : : %d, je suis ton père !!!\n",getpid(),pidfils);
    sleep(10);
  }
  else
  {
    printf("%d : j'ai faim et mon papa est %d\n",getpid(),getppid());
    sleep(10);
  }
}
```

A la fin de ce code, le père et le fils ne se distinguent que par leur PID, ce qui permet de sauter dans l'une ou l'autre partie de l'alternative.

3.2.3 Terminaison d'un fils

Pour terminer un processus, il suffit d'utiliser la fonction :

```
void exit(int status);
```

Le status est renvoyé au père (voir plus loin).

3.2.4 Attente d'un fils

Le père peut se mettre en attente jusqu'à la terminaison de son fils. Cette requète est effectuée grâce à la fonction :

```
pid_t wait(int *status)
```

Qui renvoie le numéro du processus terminé et qui met à jour la valeur de status en fonction du mode de terminaison du fils :

- Si le fils s'est terminé normalement par un *exit(n)* : l'octet de poids faible de status vaut 0 et l'octet de poids fort vaut n.
- Si le fils a été tué par le signal numéro k : l'octet de poids fort est nul. Les 7 bits de poids faible correspondent à la valeur de k ; le huitième bit est à 1 si une image mémoire a été générée (*core dumped*).

Un exemple de fonctionnement du wait est donné ci-dessous :

```
/* Utilisation de la commande FORK */

#include <stdio.h>
#include <stdlib.h>

/* Programme principal */
int main(int   argc, char *argv[])
{
  int   pidfils;
  int   fpid;
  int   status;

  switch (pidfils=fork())
  {
    case 0:/*code du fils*/
      sleep(10);
      exit(2);
    break;

    case -1:
      perror("Le fork a echoue");
    exit(1);
    break;

    default : /*code du pere*/
      printf("Je suis %d et j'attends la fin de mon process fils %d\n", \
              getpid(),pidfils);
      fpid=wait(&status);

      /* Si l'octet faible est different de 0 alors le
                          process fils a rencontre un probleme */
      if (0xff & status)
        {
          printf("Probleme ! Mon fils a renvoye %d dans l'octet de poids
              faible\n", (0xff & status));
        }
      else
      {
         printf("Mon fils %d s'est termine par un exit %d\n",fpid,\
                 (0xff & (status>>8)));
      }

   }
}
```

La consultation de status peut aussi s'effectuer à l'aide des macros :

- WIFEXITED(status) : terminaison normale (WEXITSTATUS(status) permet d'obtenir le code de retour)

– WIFSIGNALED(status) : terminaison par un signal (WTERMSIG(status) permet alors d'obtenir le numéro du signal)
– WIFSTOPPED(status) : terminaison par un signal (WSTOPSIG(status) permet alors d'obtenir le numéro du signal)
– WIFCOREDUMP(status) : Fichier core généré

Le code précédant peut ainsi se réécrire :

3.2.5 Remplacement du code du fils

Nous avons vu qu'après un fork, les codes du père et du fils sont identiques et qu'ils ne diffèrent que par leur PID. Si l'on veut faire effectuer 2 tâches distinctes par les 2 processus, il faut donc prévoir chacun des codes correspondant qui serait présent dans les 2 processus mais qu'un seul utilisera à chaque fois. C'est une perte de ressources et, par ailleurs, cela implique de coder les codes des 2 programmes.

La fonction *exec* permet de pallier ce problème en remplaçant le code du processus fils par un code spécifique. En fait, il s'agit même du seul moyen d'exécuter de nouveau programmes avec UNIX : Un programme voulant en lancer un autre (comme un shell par exemple) *fork* puis remplace le code de son fils par le code du programme à exécuter.

Il existe différentes formes d'appels de la fonction *exec* selon l'environnement ou les paramètres que l'on souhaite faire parvenir au programme.

execl nombre de paramètres figé. Le programme est dans le repertoire courant
```
int execl( const char *path, const char *arg,...) ;
```
execlp même chose mais le **$PATH** est intégré pour la recherche du code
execle même chose mais la variable d'environnement **envp** est donnée au programme
```
int execle( const char *path, const char *arg , ..., char *const envp[]) ;
```
execv nombre de paramètres libre
```
int execv( const char *path, char *const argv[]) ;
```

`execvp` et `execve` sont les équivallents de `execp` et `exece` pour `execv`.

Ci-dessous nous donnons un exemple d'utilisation d'`execl`.

Le père lance la commande *ls* puis reprend la main. Notez que le nom de la commande est donné 2 fois ! (1 fois pour indiquer où se trouve le code, une seconde fois en tant que `argv[0]` – voir (*)).

```
    /* Utilisation de la commande EXECL */
    /* lancement de la commande ls */

    #include <stdio.h> <stdlib.h>

    /* Programme principal */

    int main(int    argc, char *argv[])
    {
      int    pidfils,fpid;
      int    pidpere=getpid();
      int    status;

      if (pidfils=fork())
         {
          fpid=wait(&status);
          printf("Le process fils %d est termine ",fpid);
          if (0xff & status)
            {
              printf("et il y a eu un probleme !\n");
            }
          else
            {
              printf("et tout s'est bien passe\n");
            }
        }
      else
        {
          printf("Je suis le processus %d et je vais faire un ls\n",getpid
              ());
(*)       execl("/usr/bin/ls","ls",0);
          printf("Croyez-vous que ce commentaire sera ecrit ?");
        }
    }
```

Dans le second exemple, la fonction `execv` est utilisée afin de montrer comment s'effectue le passage des paramètres (noter le caractère nul pour signifier que la liste des paramètres est terminée).

```
/* Utilisation de la commande EXECVP */
/* lancement de la commande ls */

#include <stdio.h> <stdlib.h>

/* Programme principal */

int main(int    argc, char *argv[])
{
  int    pidfils,fpid;
  int    pidpere=getpid();
  int    status;
  char   *par[5];

  par[0]="ls";
  par[1]="-a";
  par[2]="-p";
  par[3]=(char *)0;

  if (pidfils=fork())
    {
      fpid=wait(&status);
      printf("Le process fils %d est termine ",fpid);
      if (0xff & status)
        {
          printf("et il y a eu un probleme !\n");
        }
      else
        {
          printf("et tout s'est bien passe\n");
        }
    }
  else
    {
      printf("Je suis le processus %d et je vais faire un ls -a -p\n",
          getpid());
      execvp(par[0],par);
      printf("Croyez-vous que ce commentaire sera ecrit ?");
    }
}
```

3.2.6 Modification des règles d'ordonnancement

3.3 Quiz

Questions de cours :

1. Qu'appele-t-on *processus* ?
2. Sous Unix, comment est organisé un processus ?
3. Quels sont les structures appartenant à un processus qui restent toujours en mémoire ?
4. Dans quel cas un processus passe dans l'état *zombie* ?
5. Dans quel cas un processus passe dans l'état *sleep* ?
6. Dans quel cas un processus est swappé out ?
7. Dans quel cas un processus est swappé in ?
8. Quel est le principe de fonctionnement du multi-tâche sous Unix ?
9. Quelle est la fonction associée ?
10. Après l'utilisation de cette fonction, quels sont les différences entre le père et le fils ?
11. Quand peut se terminer un processus ?
12. Qu'appele-t-on le **PID** ? Le **PPID** ?
13. Quel est le processus aïeul de tous les processus ?
14. Que se passe-t-il lorsqu'un fils meurt ?
15. Que se passe-t-il lorsqu'un père meurt ?
16. Quelle est la fonction permettant de mettre en place le code spécifique d'un fils ?
17. Après l'utilisation de cette fonction, quels sont alors les différences entre le père et le fils ?
18. Est-ce que la pile et le tas peuvent entrer en collision ? Que se passe-t-il alors ?
19. Quelle sont les moyens de passer des informations à un fils ?

3.4 Exercices

1. Ecrire un programme permettant de lancer n fils séquentiellement
2. Ecrire un programme qui lance n fils en parallèle
3. Que peut on reprocher au programme suivant :
```
int main()
{
   for( ; ;)
      fork() ;
}
```
4. Ecrivez un programme permettant de lancer la compilation d'un programme C à partir d'un menu, puis son linkage et enfin son execution tout en gardant le contrôle au niveau de l'interpréteur de commande.
5. Ecrire un programme permettant d'afficher une image en fond d'écran (utiliser *xv*) avec un rafraîchissement donné (les images sont choisies dans une liste fournie dans la ligne de commande).

Chapitre 4

Gestion de fichiers et systèmes de fichiers

Outre la gestion des processus, le noyau s'occupe de la gestion des fichiers. On pourrait a priori penser que cette tâche est simple et plutôt secondaire pour le système. Il n'en est rien. En effet, UNIX propose une interface générique pour la gestion des entrées/sorties de manière à intégrer dans un même concept *les fichiers ordinaires*, les *les flux réseau* et les *les périphériques* (imprimantes, bandes...).

4.1 Définitions

- Les fichiers ordinaires : c'est une suite d'octets quelconque qui n'a pas d'organisation *a priori*. Un fichier exécutable a une organisation sur le disque (`a.out`, ELF...) qui permet le chargement de son image en mémoire lors de son exécution.
- Les répertoires : ce sont des fichiers (!) qui assurent une correspondance logique entre un nom et sa position sur le disque physique. Le répertoire racine (root) est le fichier répertoire de base qui pointe sur l'ensemble des autres fichiers répertoires, définissant ainsi un arbre dont les feuilles les plus profondes sont des fichiers ordinaires ou des périphériques.
- Les liens : il en existe 2 types ; *les liens matériels* qui représentent les différents accès possibles au fichier physique (si le nombre de liens matériels passe à 0 le fichier est supprimé) ; *les liens symboliques* qui définissent des *chemins d'accès* synonymes à un fichier.
- Les fichiers spéciaux : les fichiers spéciaux sont situés dans les répertoires `/dev` et correspondent à des périphériques (ce concept sera approfondi au chapitre 8). Pour autoriser la manipulation de périphériques fonctionnant de manières très différentes, 2 types de fichiers spéciaux sont définis :
  - le mode c : mode caractère ; mode de fonctionnement par défaut

  - le mode b : mode bloc ; mode de fonctionnement des périphériques utilisant un cache
- les pipes et pipes nommés (voir section 4.4)
- les sockets (s) : canal standardisé pour la communication entre 2 processus distants. Cette partie nécéssitant des connaissances réseau, nous ne la détaillerons pas dans ce document.

4.2 Organisation des fichiers dans un système de fichiers

Les fichiers sont des entités appartenant à une arborescence construite à partir de différents disques physiques, en employant 2 stratégies :
- Division d'un disque physique local en plusieurs disques logiques (systèmes de fichiers) grâce au partitionnement et aggrégation au système de fichier. Le descripteur des différents systèmes de fichiers est chargé grâce à la commande *mount*.
- *Montage* des systèmes de fichiers distants par le réseau grâce à NFS (*Networked File System*). Cette partie nécéssitant des connaissances réseau, nous ne la détaillerons pas dans ce document.

4.2.1 Disque physique

Un disque dur est une mémoire de masse magnétique constituée de plateaux rigides en rotation (en aluminium généralement, mais il en existe aussi en verre ou en céramique). Les faces de ces plateaux sont recouvertes d'une couche magnétique, sur laquelle sont stockées des données binaire grâce à une tête de lecture/écriture qui modifie le champ magnétique local pour écrire sur la surface du disque. A l'origine, l'état du support magnétique est dans un état indeterminé, et il est donc nécessaire de le **formatter**, c'est-à-dire, de préparer le support magnétique en y inscrivant un système de fichiers, de façon à ce qu'il soit reconnu par le système d'exploitation. Le formatage de bas niveau, en particulier, s'occupe de rendre la surface du disque conforme à ce qu'attend le contrôleur matériel du disque, tandis que le formatage de haut niveau concerne les informations logicielles propres au système d'exploitation. Le formatage de bas niveau divise la surface d'un disque en pistes concentriques et secteurs associés à ces pistes. Le formatage de bas niveau inscrit des propriétés telles que les numéros des secteurs, nécessaires au matériel et au contrôleur de disque. De plus, il est possible de recourir à plusieurs formats de bas niveau sur le même périphérique. Par exemple, il peut être utile d'augmenter la taille des secteurs pour réduire l'espace entre les enregistrements.

Même s'il ne s'agit pas d'entités purement physiques, car non visibles par les controleurs de disques, il est intéressant de définir ici la notion de **bloc**. Un bloc, est une unité élementaire de données sur un disque. En général, un bloc contient 512 ou 1024 octets (qui peuvent en réalité correspondre à 2 ou plusieurs secteurs physiques sur le disque).

4.2.2 Le système de fichier Ext2

S'appuyant sur le formatage de bas niveau, le formatage de haut niveau, consiste essentiellement à créer un certain nombre d'unités fonctionnelles (signifiantes pour les système d'exploitation) constituées d'assemblages de blocs du disques. Il existe de nombreux types de systèmes de fichiers (Ext2, Ext3, raiserFS, MINIX, FAT, NTFS...) mais nous ne détaillerons ici que le type Ext2, qui est celui le plus utilisé actuellement (dans sa version journalisée – Ext3 – surtout). Une lecture de [Poirier, , Corbet et al., 2005] ainsi que [Card,] (en français et de l'auteur lui-même) est vivement conseillée pour un approfondissement.

Sur Ext2, on distingue 2 unités fonctionnelles :
- Le **bloc de démarrage** (Boot bloc) = bloc 0 du disque : utilisé au démarrage du système. Il n'est donc pas directement utilisé par Linux.

– n **groupes de blocs** : qui contiennent les données et leurs méta-données. Le nombre de groupes dépend de 2 facteurs, la taille de la partition et la taille d'un bloc, sachant que dans un groupe il ne peut y avoir que $8 \times b$ blocs (avec b la taille d'un bloc en octets).

Chaque groupe de blocs contient lui-même 6 sous-unités fonctionnelles :

– Le **Super bloc** = bloc 0 du groupe : il contient la description de l'organisation du système de fichiers. Ces données ne sont pas propres à un groupe de blocs en particulier mais à l'ensemble du système de fichier. Pour des raisons de robustesse, le superbloc est recopié dans chacun des groupes.
– Les **descripteurs de groupe** : il contiennent les descriptions de l'organisation des groupes de blocs. Pour des raisons de robustesse encore, tous les descripteurs de groupe sont recopiés.
– La carte binaire de l'occupation des données
– La carte binaire de l'occupation des inodes
– La **table des inodes** (index-node) : contient la liste des inodes.
– Viennent ensuite les blocs de données à proprement parler.

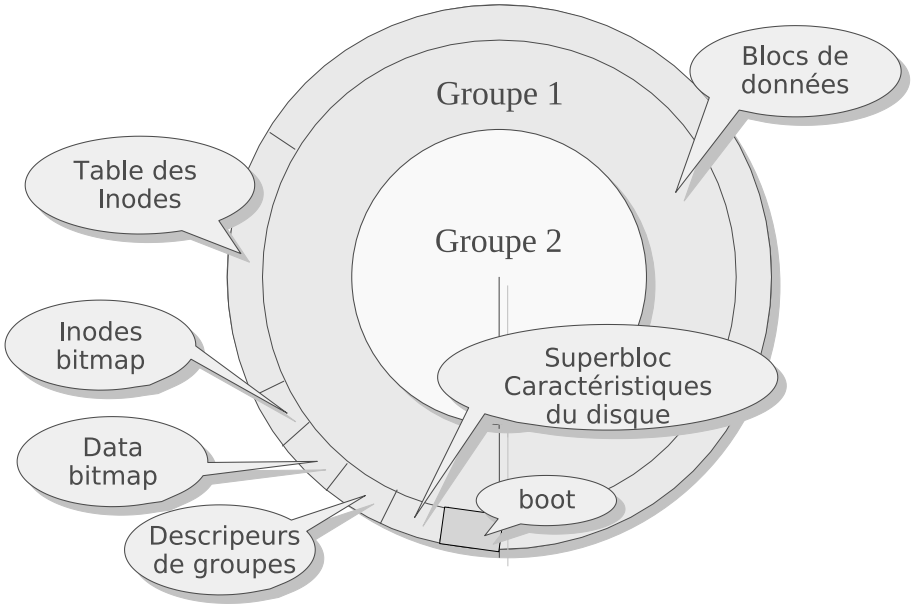


FIG. 4.1 – Organisation d'un disque au format Ext2.

La notion d'inode étant essentielle à la compréhension du fonctionnement du système de fichier, nous allons commencer par sa description puis reviendrons aux autres structures par la suite.

La table des inodes

L'inode est le descripteur d'un fichier. C'est une structure de 128 octets qui spécifie, entre autres :

– Les attributs
  – accès
  – type de fichier
  – propriétaire
  – groupe
– Les 15 adresses (32 bits) physiques de blocs de données[1]. Le i-number est l'index dans la table des inodes de l'inode du fichier.

Par défaut la première entrée de la tables des inodes est inutilisée et la seconde contient le descripteur du répertoire racine.

[1] 12 adresses immédiates+adresse de la table de première indirection+adresse de la table de seconde indirection+adresse de la table de troisième indirection

On peut accéder à un fichier :
– par accès direct
– par indirection simple
– par indirection double
– par indirection triple
Avec des blocs de 4Ko, on peut accéder au total à 16 Go pour des blocs de 1024 octets.

Le superbloc

Le superbloc assure la gestion des inodes et des blocs libres. Il contient des données génériques sur le système de fichier telles que la taille des blocs, leur nombre, le nombre de blocs ou d'inodes libres, le nombre de blocs par groupe, et d'autres informations propres au disque telles que la date de la dernière opération de montage ou de vérification de l'intégrité.

Les descripteurs de groupes

Un descripteur de groupe contient :
– Le numéro de bloc de la table binaire de l'occupation des données
– Le numéro de bloc de a table binaire de l'occupation des inodes
– Le numéro du premier bloc de la table des inodes
– Le nombre de blocs libres dans le groupe
– Le nombre d'inodes libres dans le groupe
– Le nombre de répertoires dans le groupe

La table binaire de l'occupation des données

La table binaire de l'occupation des données donne une représentation de l'occupation des données dans le groupe sous la forme d'une carte binaire : si le bloc est occupé, le bit est à 1 alors qu'il est à 0 autrement. Ce type de représentation permet un accès rapide aux blocs libres et à l'occupation du disque.

La table binaire de l'occupation des inodes

Cette table a un fonctionnement identique à la précédente mais concerne l'occupation des inodes dans la table des inodes. Cela permet de repérer rapidement les inodes pouvant être utilisées ou ré-utilisées.

4.2.3 Le système de fichier Ext3

Lors des accès en écriture sur le disque, le système d'exploitation n'écrit pas directement les données sur le disque du fait de sa lenteur mais les gardes en mémoires et les écrits par grand blocs à intervales réguliers (cf. chapitre 11). Si un crash du système survient avant cette syncrhonisation des données, le système de fichier perd son intégrité. Lors du redémarrage du système une procédure de recouvrement de l'intégrité peut être éffectuée (commande `fsck`) mais requiert beaucoup de temps car il est nécéssaire d'explorer tous les fichiers et répertoires du disque.

Le système de fichier Ext3 est une évolution du système Ext2 qui a pour principale amélioration la prise en compte d'un fichier journal. L'intérêt principal d'une journalisation est de permettre d'éviter la phase de récupération de l'intégrite du système. Pour ce faire, avant toute écriture sur le disque, le système écrits d'abord les informations dans le journal, puis sur le disque. Si tout se passe bien, les données du journal sont alors invalidées. Lors d'un crash, si celui-ci survient après l'écriture du journal

mais avant l'écriture sur le disque, le système peut recopier les données du journal dans le système de fichier au prochain redémarrage.

Evidemment, un tel mécanisme est beaucoup plus robuste mais consomme 2 fois plus de temps et n'est que rarement utilisé tel quel. En réalité, plutôt qu'écrire dans le journal les données du fichier, le système écrit plutôt les méta-données (modification du superbloc, groupes de blocs, inodes, bitmap...). Lors du crash, le système peut ainsi identifier facilement les sources d'intégrité.

Sous Linux, la stratégie suivie pour l'écriture du journal (souvent repéré `.journal` à la racine du système de fichiers) est définie par l'administrateur. 3 stratégies sont utilisables :

Journal : A la fois les données et les méta-données sont ecrites dans le journal. C'est une stratégie très lente.

Ordonné : Seules les méta-données sont écrites dans le journal. Cependant, par dans la phase d'écriture réelle, les données sont écrites sur le disque avant les méta-données. En particulier, l'écriture des méta-données du journal et des données sont groupées dans la même transaction. Ainsi, en cas de crash, ces informations peuvent être récupérées depuis le journal. Ce comportement est celui qui est implémenté par défaut dans Ext3.

writeback : Seules les méta-données sont écrites dans le journal et les données sont écrites après les méta-données. C'est le mode le plus rapide.

4.2.4 Ext4

Ext4 est la version suivante du système de fichiers Ext3. Outre le fait qu'il puisse gérer les volumes d'une taille allant jusqu'à 1 024 péta-octets, la fonctionnalité majeure de ext4 est la prise en charge d'extent, qui permet l'allocation d'une zone réservée pour un fichier. Un extent est en effet une zone de stockage contiguë réservée pour un fichier sur le système de fichiers d'un ordinateur. Lorsqu'on commence à écrire sur un fichier, un extent entier est alloué. Lorsqu'on écrit à nouveau sur ce fichier, éventuellement après avoir réalisé d'autres opérations d'écriture, les données sont ajoutées là où l'écriture précédente s'était arrêtée. Cela réduit ou élimine la fragmentation des fichiers.

L'option extent est active par défaut depuis le noyau Linux 2.6.23 ; avant cela, elle devait être explicitement indiquée lors du montage de la partition (par exemple : `mount /dev/sda1 /mnt/point -t ext4dev -o extents`)

Le système de fichiers Ext4 est compatible avec Ext3, ce qui signifie qu'il peut être monté comme une partition Ext3 (en utilisant le type de système de fichiers “Ext3” lors du montage). L'inverse est également possible, le montage d'un système de fichiers Ext3 comme un Ext4 (en utilisant le type de système de fichiers ”ext4dev”). Cependant, si la partition Ext4 utilise les extent, la rétro-compatibilité est perdue, et avec elle la possibilité de monter le système de fichiers en tant qu'Ext3.

4.2.5 Quelques commandes utiles...

La commande `df` permet d'afficher les périphériques logiques montés sur l'arborescence Linux. Voici son fonctionnement sur une partition principale “/” qui fait 8Go et possède des blocs de 4Ko

```
#df -lh  /* -l : disques locaux ; -h : lisible par un humain */
Sys. de fich.          Tail. Occ. Disp. %Occ. Monté sur
/dev/sda5              7.7G  5.2G  2.2G  70% /

#df -B 4096 /* -B : blocs de 4096 octets */
Sys. de fich.        4K-blocs       Occupé Disponible Capacité Monté sur
/dev/sda5              2016068    1338542    575113  70% /

#df -il   /* -i : information sur les inodes */
Sys. de fich.          Inodes   IUtil.   ILib. %IUti. Monté sur
/dev/sda5            1026144  209148  816996   21% /
```

La commande `df` permet d'afficher des données génériques sur un fichier ou un système de fichier (option `-f`). Dans le cas de notre système précédent, on retrouve bien les informations sur le nombre

total d'inodes (1026144) et sur les inodes disponibles (816996). A partir du nombre de blocs disponibles (575113), on peut aussi calculer le nombre d'octets libres sur le disque (2.2Go).

```
#stat -f .
  File: "."
    ID: 8a8610207648e242 Namelen: 255     Type: ext2/ext3
Block size: 4096       Fundamental block size: 4096
Blocks: Total: 2016068     Free: 677526      Available: 575113
Inodes: Total: 1026144     Free: 816996
```

La commande `dumpe2fs`, quant à elle, permet d'afficher les données complètes du superbloc et de chacun des groupes du système de fichier. De nouveau, on retrouve les informations sur le nombre d'inodes libres (Inode count : 1026144). Les informations de blocs ne sont cependant pas cohérentes... c'est compter sans le fichier de journalisation de 128Mo ! Par ailleurs, comme on peut le voir, 5% des blocs sont reserves pour root (`Reserved block count`).

```
#dumpe2fs /dev/sda5
Filesystem volume name:   /
Last mounted on:          <not available>
Filesystem UUID:          b06cced0-550c-43e8-907c-485a17ee0b9e
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
Filesystem features:      has_journal resize_inode dir_index filetype needs_recovery sparse_super large_file
Filesystem flags:         signed directory hash
Default mount options:    (none)
Filesystem state:         clean
Errors behavior:          Continue
Filesystem OS type:       Linux
Inode count:              1026144
Block count:              2048279
Reserved block count:     102413
Free blocks:              677632
Free inodes:              817093
First block:              0
Block size:               4096
Fragment size:            4096
Reserved GDT blocks:      500
Blocks per group:         32768
Fragments per group:      32768
Inodes per group:         16288
Inode blocks per group:   509
Filesystem created:       Sun Feb 24 20:15:31 2008
Last mount time:          Wed Jul 30 14:49:39 2008
Last write time:          Wed Jul 30 14:49:39 2008
Mount count:              150
Maximum mount count:      -1
Last checked:             Sun Feb 24 19:34:50 2008
Check interval:           0 (<none>)
Reserved blocks uid:      0 (user root)
Reserved blocks gid:      0 (group root)
First inode:              11
Inode size:               128
Journal inode:            8
First orphan inode:       212478
Default directory hash:   tea
Directory Hash Seed:      350a8f17-0fcc-4cc3-b45d-09f106668e10
Journal backup:           inode blocks
Taille du journal:        128M

Groupe 0 : (Blocs 0-32767)
  superbloc Primaire à 0, Descripteurs de groupes à 1-1
  Blocs réservés GDT à 2-501
  Bitmap de blocs à 502 (+502), Bitmap d'i-noeuds à 503 (+503)
  Table d'i-noeuds à 504-1012 (+504)
  0 blocs libres, 16277 i-noeuds libres, 2 répertoires
  Blocs libres :
  I-noeuds libres : 12-16288
Groupe 1 : (Blocs 32768-65535)
  superbloc Secours à 32768, Descripteurs de groupes à 32769-32769
  Blocs réservés GDT à 32770-33269
  Bitmap de blocs à 33270 (+502), Bitmap d'i-noeuds à 33271 (+503)
  Table d'i-noeuds à 33272-33780 (+504)
  30573 blocs libres, 16267 i-noeuds libres, 0 répertoires
  Blocs libres : 34834-34982, 34995-40959, 40961-47206, 47238-49151, 49183-51199, 51224-53247, 53251-61445,
      61473-65535
  I-noeuds libres : 16291, 16304-16392, 16394-16446, 16453-32576
...
Groupe 62 : (Blocs 2031616-2048278)
  Bitmap de blocs à 2031616 (+0), Bitmap d'i-noeuds à 2031617 (+1)
  Table d'i-noeuds à 2031618-2032126 (+2)
  16143 blocs libres, 16278 i-noeuds libres, 0 répertoires
  Blocs libres : 2032127-2039848, 2039858-2048278
  I-noeuds libres : 1009861-1009862, 1009865-1009894, 1009899-1026144
```

Accès aux fichiers et répertoires

Une fois donnée la description du système de fichier, nous devons détailler comment s'effectue la recherche du contenu des données d'un fichier donné ainsi que les méta-informations qui lui sont associées. En fait, le principe d'accès aux fichiers et sa plasticité réside principalement dans l'utilisation des inodes. En effet, un fichier (quelque soit son type) est représenté par un numéro d'index unique dans la table des inodes (figure 4.2) qui le définit de manière univoque.

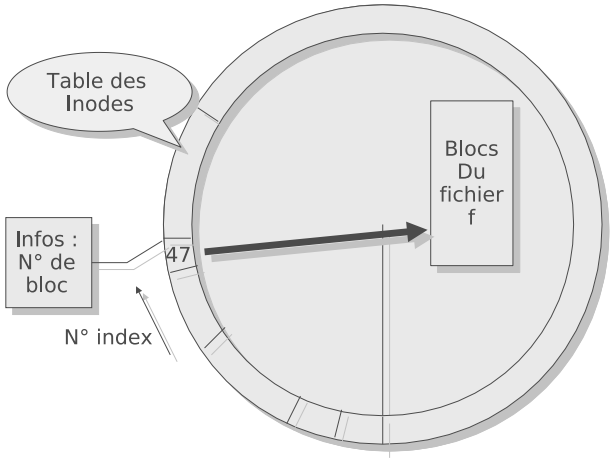


FIG. 4.2 – Lien entre un inode et la matérialisation d'un fichier.

En effet, les méta-données contenues dans l'inode (type, droits d'accès...) permettent de définir les attibuts du fichier et les données d'accès aux blocs de fichiers (adresses d'accès direct et indirect) permettent de spécifier entièrement les qualités du fichiers ainsi que sa localisation physique (figure 4.3).

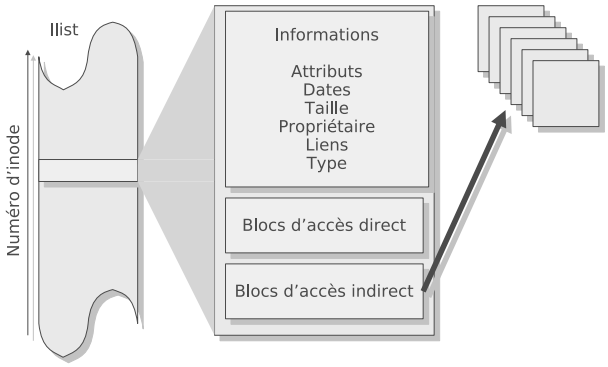


FIG. 4.3 – Détails de la méthode d'accès à la matérialisation d'un fichier.

Gestion des répertoires

Sous Ext2, les répertoires sont des fichiers de taille variable (multiple de 4 pour des raisons de rapidité) dont les entrées sont structurées en chaînes de caractères correspondant à une référence pour chaque fichier ou sous-répertoire. A cette chaîne de caractère est associée un entier correspondant au numéro d'inode du fichier référencé par ce nom (voir figure 4.4).

L'accès à un fichier est donc aisée si l'on dispose des informations contenues dans un répertoire (figure 4.5).

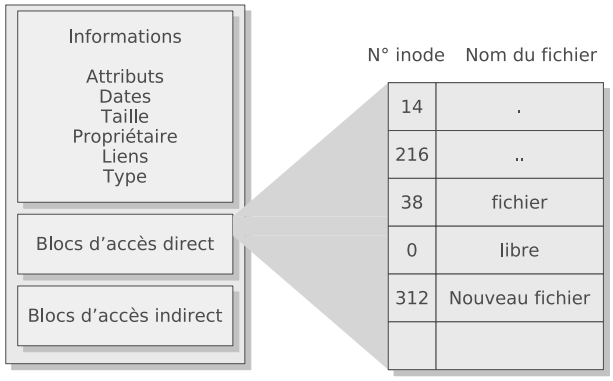


FIG. 4.4 – Description du contenu d'un répertoire.

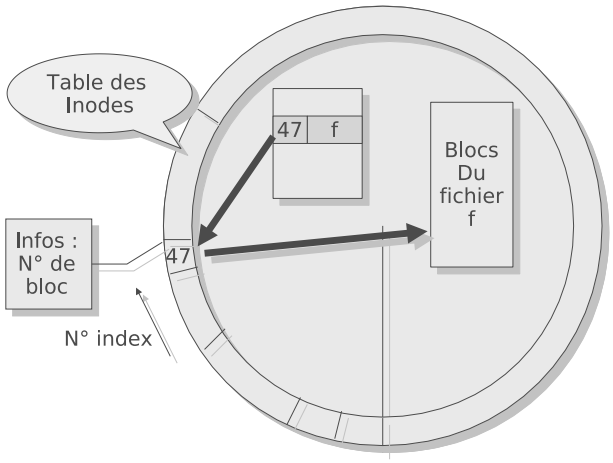


FIG. 4.5 – Accès à un fichier depuis le contenu d'un répertoire.

Pour accèder aux informations contenues dans ce répertoire, 2 méthodes existent :
– Par accès par référence absolue
– Par accès par référence relative

Accès par référence absolue Dans ce cas, la référence est donnée depuis la racine du système de fichier. Or, comme on l'a dit plus tôt, la position de l'inode de la racine du système de fichier se trouve toujours en deuxième position dans la table des inodes. Il est ainsi facile de proche en proche de déterminer les informations relatives aux diverses répertoires qui composent le chemin d'accès au fichier (4.6).

Accès par référence relative Dans le cas d'un référencement, l'accès se fait par l'intermédiaire du répertoire courant. En effet, le système d'exploitation conserve en mémoire le numéro d'inode du répertoire courant. Par ailleurs, chaque répertoire contient 2 entrées spéciales `.` et `..` qui référencent réspectivement, le répertoire courant et le répertoire parent. En utilisant un chaînage semblable à celui évoqué plus haut, on peut ainsi aisément accéder au fichier.

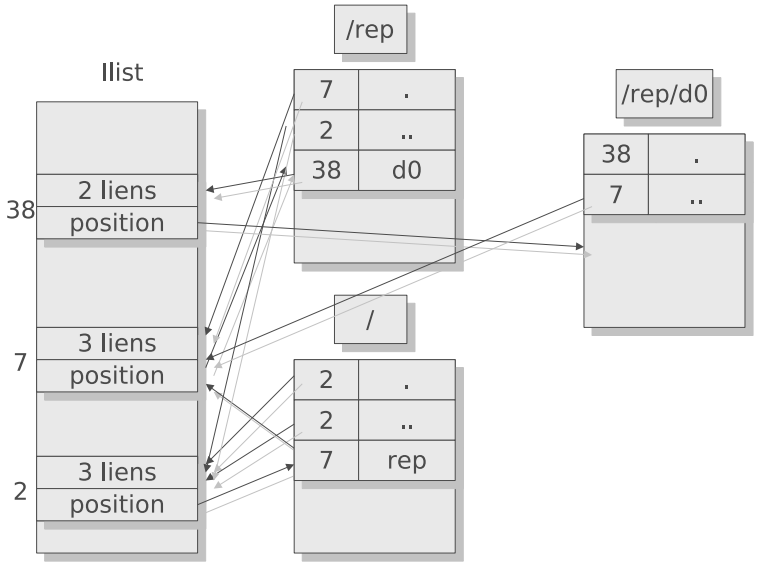


FIG. 4.6 – Chaînage des répertoire pour accès à un fichier en référencement absolu.

Cas de gros fichiers

L'accès aux informations contenues par un fichier s'effectue en accèdant aux adresses des blocs de ce fichier. Ces adresses sont, soit directement les 12 adresses 32 bits contenues dans une inode, soit sont elles-mêmes référencées via les adresses d'indirection. Il existe 3 types d'indirection :

1. L'indirection simple : les adresses des blocs du fichier sont contenues dans un bloc référencé par l'adresse d'indirection simple contenue dans l'inode.
2. L'indirection double : les adresses des blocs du fichier sont contenues dans un ensemble de blocs référencés par un bloc dont l'adresse est donnée par la référence d'indirection double contenue dans l'inode.
3. L'indirection triple : les adresses des blocs du fichier sont contenues dans un ensemble de blocs référencés par un ensemble de blocs, eux-mêmes référencés par un bloc dont l'adresse est donnée par la référence d'indirection triple contenue dans l'inode (ouf !).

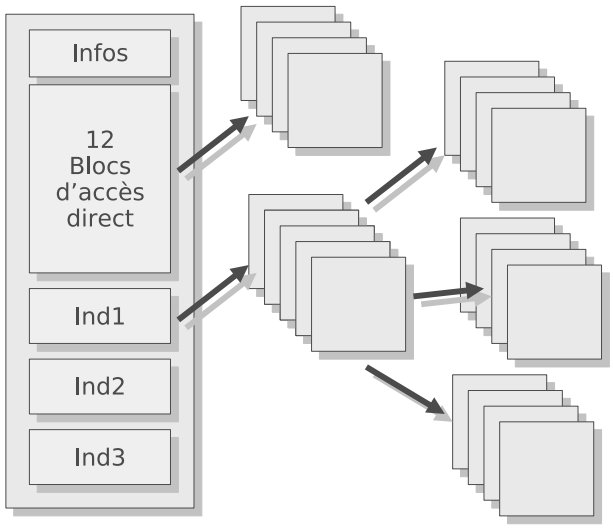


FIG. 4.7 – Méthode d'accès à de gros fichiers.

Liens vers des fichiers

Il existe 2 types de liens vers des fichiers, les liens matériels et des liens symboliques.

Liens matériels Le fonctionnement des liens matériels repose entièrement sur la notion d'inode. En effet, étant donné ce mécanisme, rien n'empêche qu'un nom contenu dans un répertoire ne référence la même inode. Il est ainsi possible de définir plusieurs chemins vers ce fichier en créant de nouveaux liens vers un même inode.

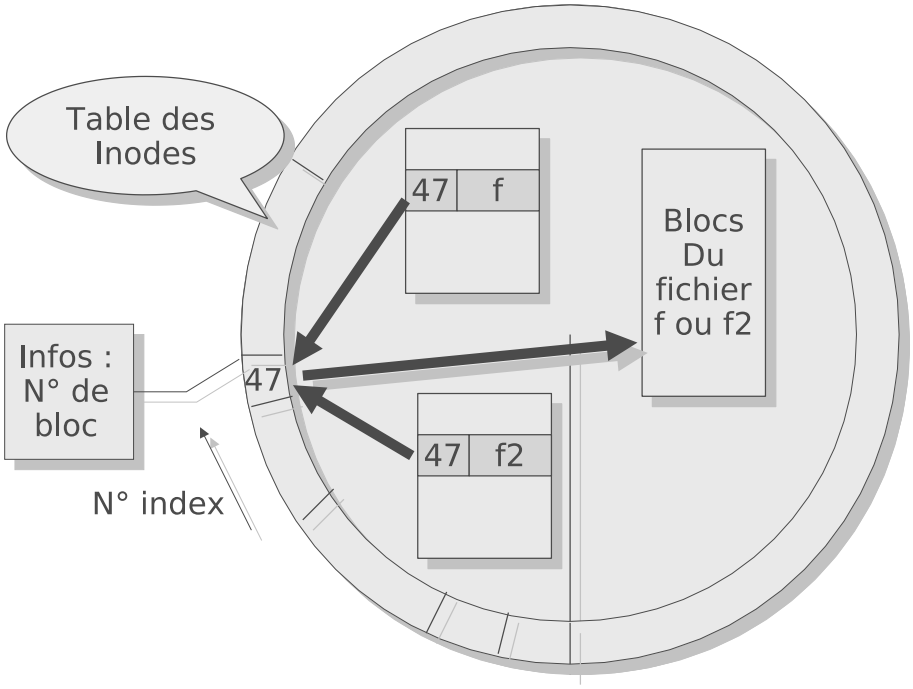


FIG. 4.8 – Fonctionnement des liens physiques.

ATTENTION TOUTEFOIS

Les liens fonctionnant à partir de l'utilisation du mécanisme des inodes, il ne peuvent être utilisés qu'au sein d'un même système de fichier. Par exemple, il n'est pas possible depuis un disque local de faire un lien vers un disque monté par le réseau. Il faut alors ce tourner vers les liens symboliques.

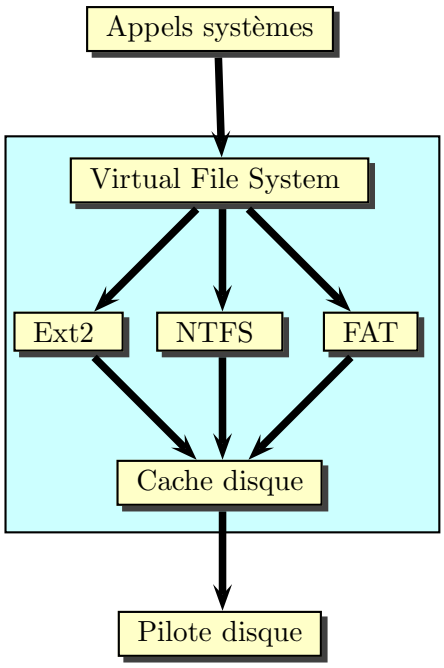
REMARQUE

Un fichier n'existe que par sa matérilisation sous la forme d'une référence dans une inode. Si un lien matériel est effectué, le nombre de références vers ce fichier augmente. Ce fichier est considéré comme "effacé" uniquement si l'ensemble des références vers l'inode le désignant a été supprimé.

Liens symboliques A la différence des liens matériels, les liens symboliques sont gérés de manière spécifiques par le système. En effet, dans le cas du lien symbolique, une entrée dans un répertoire définit le nouvel inode de ce lien lequel contient un chemin d'accès symbolique vers un fichier existant et non un lien vers ce fichier.

4.2.6 Le Virtual File System

En pratique, Linux met à la disposition de l'utilisateur une couche d'abstraction (système de fichiers virtuel ou VFS) des systèmes de fichiers qui permet de rendre transparents leur utilisation malgrè leurs différences d'implémentation (Ext3, FAT32, XFS...). C'est aussi vrai pour les montages réseau (NFS, SMB...) ou pour le paramètrage système (`procfs`, `devfs` et `sysfs`).



Pour ce faire, linux propose une interface d'appels systèmes commune pour les différentes opérations sur les fichiers. Le noyau s'occupe ensuite de faire la redirection de l'appel système vers une routine spécifique au système de fichier ciblé pour l'opération demandée. Afin d'unifier ces appels systèmes malgrè les différences d'implémentation, le noyau propose un modèle de fichier commun très proche des systèmes de fichier natifs UNIX (et donc proches d'Ext2/3). Ce modèle fait appel à 4 objets placés en mémoire lorsque le système de fichier est monté : l'objet superblock, l'objet fichier, l'objet dentry et l'objet inode

Un système de fichier est une collection d'inodes comportant une inode particulière associée à la racine. On accède aux autres inodes en partant de cette racine et en chaînant de proche en proche les inodes via le chemin d'accès vers le fichier.

Un système de fichier possède un ensemble de caractéristiques qui s'appliquent de manière uniforme à l'ensemble des inodes du système de fichier (comme par exemple un drapeau READ-ONLY ou la taille d'un bloc). Chaque système de fichier est représenté par une strutcture `super_block`.

Il existe une forte corrélation entre le numéro du périphérique et le superbloc. Chaque système de fichier doit apparaître avec un numéro de périphérique unique (celui sur lequel réside ce système). Certains périphériques sont spécifiés comme ne nécessitant pas de périphérique réel (tels que NFS ou `/proc`). Dans ce cas, un périphérique anonyme, dont le numéro majeur est 0 est automatiquement assigné.

Chaque type de système de fichier connu par Linux est représenté par la structure `file_system_type`. Cette structure contient simplement une méthode (`read_super`) qui instancie un superbloc représentant le système de fichier en question.

L'objet superblock

Cette structure contient des informations globales portant sur le système de fichiers monté telles que le nombre et la taille des blocs, le type de système de fichiers, l'identifiant du périphérique sur lequel il est monté, etc. Elle contient également un champ référençant les pointeurs sur les fonctions du superblock.

La structure `super_block` est définie dans `<linux/fs.h>`.

L'objet inode

Cette structure représente un objet quelconque du système de fichiers. Cela peut être un fichier, un répertoire, un lien symbolique... Chaque Inode a un numéro unique (inode number) qui permet d'identifier l'objet correspondant. Certains objets ont un inode mais pas de structure file (liens symboliques) et vice-versa. Dans un système de fichier de type UNIX, l'inode contient essentiellement les informations contenues dans les véritables inodes du système de fichiers, c'est-à-dire taille, attribut de l'inode, nombre de liens...

La structure `inode` est définie dans `<linux/fs.h>`.

L'objet fichier

La structure file représente tout objet sur quoi on peut lire/écrire (ce qui est très courant sous Linux :)). Elle contient aussi les informations sur les fichiers ouverts par un processus. Elle fournit aussi l'implémentation des opérations disponibles sur le fichier.

La structure `file` est définie dans `<linux/fs.h>`.

L'objet dentry

La structure `dentry` (Directory ENTRY) contient les informations liant l'entrée logique correspondant à un répertoire et le fichier réel correspondant (chaque système de fichier enregistrant ces informations à sa manière). VFS gère en effet l'intégralité des chemins d'accès aux fichier. C'est lui qui convertit ces chemins en entrées dans un cache (dcache) avant de passer la main au système réel sous-jacent. Chaque `dentry` référence un parent qui doit déjà exister dans le dcache. Les `dentry` référencent aussi les informations de montage des systèmes de fichier. A chaque `dentry` correspond par ailleurs une inode.

Ainsi, quand le système recherche le répertoire `/tmp/test pathname`, le noyau créé une `dentry` pour `/`, une autre pour `tmp` et une troisième pour `test`.

Il est à noter que les objets `dentry` n'ont pas de correspondants sur le disque.

La structure `dentry` est définie dans `<linux/dcache.h>`.

```
struct dentry {
        atomic_t d_count;
        unsigned int d_flags;           /* protected by d_lock */
        spinlock_t d_lock;              /* per dentry lock */
        struct inode *d_inode;          /* Where the name belongs to - NULL is
                                         * negative */
        /*
         * The next three fields are touched by __d_lookup.  Place them here
         * so they all fit in a cache line.
         */
        struct hlist_node d_hash;       /* lookup hash list */
        struct dentry *d_parent;        /* parent directory */
        struct qstr d_name;

        struct list_head d_lru;         /* LRU list */
        /*
         * d_child and d_rcu can share memory
         */
        union {
                struct list_head d_child;       /* child of parent list */
                struct rcu_head d_rcu;
        } d_u;
        struct list_head d_subdirs;     /* our children */
        struct list_head d_alias;       /* inode alias list */
        unsigned long d_time;           /* used by d_revalidate */
        struct dentry_operations *d_op;
        struct super_block *d_sb;       /* The root of the dentry tree */
        void *d_fsdata;                 /* fs-specific data */
#ifdef CONFIG_PROFILING
        struct dcookie_struct *d_cookie; /* cookie, if any */
#endif
        int d_mounted;
        unsigned char d_iname[DNAME_INLINE_LEN_MIN];    /* small names */
};
```

Accès aux fichiers

L'accès aux fichiers est effectué par VFS grâce aux inodes. Quand le chemin d'un fichier à manipuler est transmis à VFS par un programme utilisateur, VFS parcourt alors le disque, élément par élément ; c'est-à-dire les fichiers et les répertoires. Le parcours s'arrête quand VFS trouve l'inode du dernier élément à manipuler. A ce moment, VFS peut alors manipuler les données du programme utilisateur. La force de VFS tient au fait qu'il sait, en fonction de la partition qu'il parcourt, dialoguer avec un système de fichiers FAT, NTFS, ext2, ext3, ReiserFS ou XFS.

4.2.7 /procfs

procfs (process file system – système de fichiers processus) est un pseudo-système de fichiers (pseudo car dynamiquement généré au démarrage) utilisé pour accéder aux informations du noyau sur les processus. Le système de fichiers est souvent monté sur le répertoire `/proc`.

Puisque `/proc` n'est pas une arborescence réelle, il ne consomme aucun espace disque mais seulement une quantité limitée de mémoire vive. Cela aboutit à un paradoxe apparent : un fichier non vide a une taille affichée de 0 (avec ls).

Voir [Daudel, 2005] et [Cornavin, 2005] pour plus de détails.

4.2.8 /sysfs

Sysfs est un système de fichiers virtuel introduit par le noyau Linux 2.6. Sysfs permet d'exporter depuis l'espace noyau vers l'espace utilisateur des informations sur les périphériques du système et leurs pilotes, et est également utilisé pour configurer certaines fonctionnalités du noyau. Sysfs a été conçu pour exporter les informations présentes dans l'arbre des périphériques qui ainsi n'encombrerait plus procfs.

Pour chaque objet ajouté dans l'arbre des modèles de pilote (pilotes, périphériques, classes de périphériques), un répertoire est créé dans sysfs. La relation parent/enfant est représentée sous la forme de sous-répertoires dans `/sys/devices/` (représentant la couche physique). Le sous-répertoire `/sys/bus/` est peuplé de liens symboliques, représentant la manière dont chaque périphérique appartient aux différents bus. `/sys/class/` montre les périphériques regroupés en classes, comme les périphériques réseau par exemple, pendant que `/sys/block/` contient les périphériques de type bloc.

Pour les périphériques et leurs pilotes, des attributs peuvent être créés. Ce sont de simples fichiers, la seule contrainte est qu'ils ne peuvent contenir chacun qu'une seule valeur et/ou n'autoriser le renseignement que d'une valeur (au contraire de certains fichiers de procfs, qui nécessitent d'être longuement parcourus). Ces fichier sont placés dans le sous-répertoire du pilote correspondant au périphérique. L'utilisation de groupes d'attributs est possible en créant un sous-répertoire peuplé d'attributs.

La commande `sysctl` est utilisée pour modifier les paramètres du noyau en cours d'exécution. Les paramètres utilisables sont ceux listés dans le répertoire /proc/sys. `sysctl` peut être aussi bien utilisé pour lire que pour écrire des paramètres.

Voir [Daudel, 2005] pour plus de détails.

4.2.9 devfs

Devfs est un système de fichiers présent sur de nombreux systèmes d'exploitation assimilés à des Unix tels que FreeBSD et Linux (bien que toutes les distributions ne l'utilisent pas).

Sur tous systèmes Unix, de nombreux périphériques d'entrée-sortie (tels que les disques, les imprimantes, les terminaux virtuels etc) sont traités comme des fichiers spéciaux. Cependant, maintenir ces fichiers dans un système de fichier classique peut devenir plutôt complexe : c'est particulièrement le cas pour certains périphériques pouvant être "montés à chaud" (c'est-à-dire montés alors que le système est opérationnel) tels que les périphériques USB par exemple. Devfs simplifie ce problème en contrôlant automatiquement la gestion de la création, de la suppression et des permissions de ces fichiers.

L'utilisation de Devfs n'est plus à ce jour conseillée pour les noyaux Linux version 2.6.x. Il a été remplacé par udev. udev est un gestionnaire de périphériques remplaçant Devfs sur les noyaux Linux de la série 2.6. Sa fonction principale est de gérer les périphériques dans le répertoire `/dev`. udev s'exécute en mode utilisateur et dialogue avec hotplug qui lui s'exécute en mode noyau.

4.3 Les fichiers

4.3.1 Cycle de vie d'un fichier

Un fichier suit le cycle de vie définit figure 4.9.

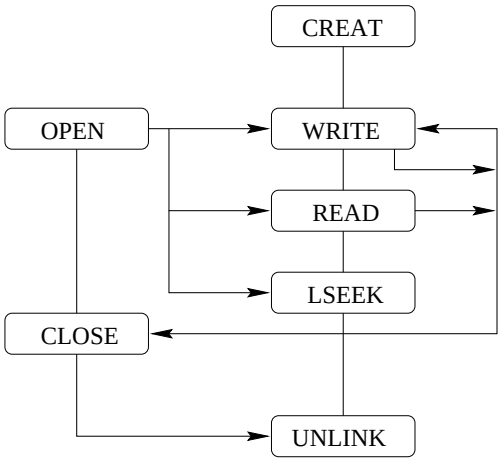


FIG. 4.9 – Cycle de vie d'un fichier

Création et/ou ouverture

La création d'un fichier s'effectue en fait en 2 étapes : le système cherche d'abord dans le répertoire où l'on veut créer le fichier, une occurrence du nom du fichier. Si le nom n'est pas déjà utilisé, il crée une entrée dans le répertoire correspondant au nom du fichier, crée un nouvel inode et effectue le lien entre cet inode et ce nom de fichier.

Lorsque le fichier est déjà créé, la procédure d'ouverture consiste simplement à renvoyer le numéro du déscripteur du fichier dans la tableau `u_ofile` des fichiers ouverts (définie dans la task_struct du processus). A l'origine 3 fichiers *standards* sont préalablement ouverts :
– Le fichier STDIN : entrée standard (clavier) associée au descripteur 0
– Le fichier STOUT : sortie standard (écran) associée au descripteur 1
– Le fichier STDERR : sortie d'affichage des messages d'erreurs (par défaut l'écran) associée au descripteur 2
Par ailleurs, le tableau `file[]` contient les informations sur tous les fichiers ouverts à un instant donné. Une entrée y est utilisée à chaque définition des paramètres d'un fichier (type, mode, etc...).

Chaque entrée de cette table contient :
– l'indicateur de mode open
– le type du fichier
– le compteur du nombre d'ouvertures du fichier (f_count)
– le pointeur f_offset (pointeur de déplacement dans le fichier utilisé à la fois en lecture et écriture)
– le numéro d'inode associé au fichier

```
int open(char *path, int mode, int droits)
```

modes :
– O_RDONLY
– O_WRONLY
– O_RDWR

Fermeture

```
int close(int fd)
```

Ferme les fichiers ouverts d'un processus actif par suppression des connexions entre les fichiers ouverts et leur descripteur.

Suppression

`int unlink(char *path)`

Supprime un lien matériel d'un fichier. La destruction n'est effective que lorsque tous les noms le désignant sont détruits.

Lecture et écriture

`int read(int fd, char *buf, unsigned byte)`

`int write(int fd, char *buf, unsigned byte)`

`buf` : données à transférer
`nbyte` : nombre d'octets à transférer.
`nlus` : ou necrits : renvoyés par les appels.

4.3.2 Effet du fork sur l'ouverture d'un fichier

Les fichiers ouverts par le père sont conservés en ouverture par le fils. Par conséquent, on se retrouve avec 1 seul descripteur partagé par 2 processus et pointant sur le même élément de la table des fichiers.

4.3.3 Ouverture multiple d'un même fichier

Il est tout à fait possible pour un processus d'ouvrir 2 fois un même fichier, par exemple en ouvrant le fichier en lecture et en écriture. Dans ce cas, le processus possède 2 descripteurs différents (auxquels sont associés des compteurs de position distincts et des modes d'accès différents), mais sollicitant le même inode (incrémentant donc le compteur de référence associé à cet inode).

4.3.4 Utilisation d'un descripteur existant

La fonction dup, duplique le descripteur associé à un fichier. Il faut d'abord fermer le descripteur disponible car dup retourne la plus petite valeur de descripteur disponible.

dup2 permet de demander explicitement la duplication d'un descripteur de valeur connue.

Ces commandes sont utilisées pour les redirections. Ainsi, lorsqu'on veut rediriger l'entrée standard vers un fichier, il suffit de fermer le canal d'entrée et de demander à dupliquer le descripteur de l'entrée standard (descripteur 0) pour l'associer au descripteur du fichier.

4.4 Les pipes

4.4.1 Fonctionnement

Les pipes sont des canaux de communication unidirectionnels entre un producteur et un consommateur.

Un tube est un inode disponible d'un système de fichiers. Le principe du fonctionnement du pipe provient du fait que l'appel système pipe retourne 2 descripteurs de fichiers vers le même inode :l'un est utilisé en écriture, l'autre en lecture. Les pointeurs de position sont gérés dans la table des inodes et non dans la table des fichiers. Quand un processus écrit dans le pipe, le compteur de position d'écriture est incrémenté. De même, quand un processus vient lire dans le pipe, le compteur de position de lecture

est incrémenté. Quand le compteur de lecture à rejoint le compteur d'écriture, les 2 compteurs sont remis à 0.

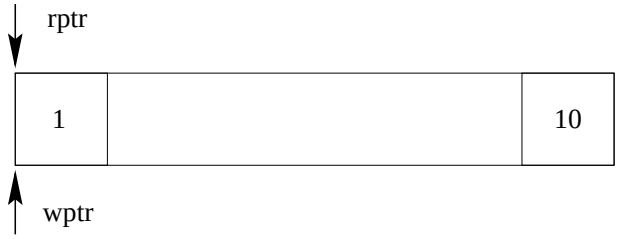


FIG. 4.10 – Un pipe est géré dans les premiers blocs d'accès directs de l'inode (donc dans le cache si 10 blocs disques sont libres)

S'il n'y a rien à lire, le processus consommateur est bloqué. Si le pipe est plein, le processus producteur est bloqué (ce qui peut conduire à un fractionnement des données si d'autres processus écrivent dans le pipe). Le positionnement du flag O_NDELAY, permet d'éviter ce comportement (l'écriture renvoie le nombre d'octets effectivement écrits).

La gestion d'un pipe se fait sur les blocs d'accès direct de l'inode qui lui est affecté (la taille est donc limitée à 10 blocs). Comme ces blocs sont gérés dans le cache, les pipes sont donc gérés directement en mémoire et non sur le disque.

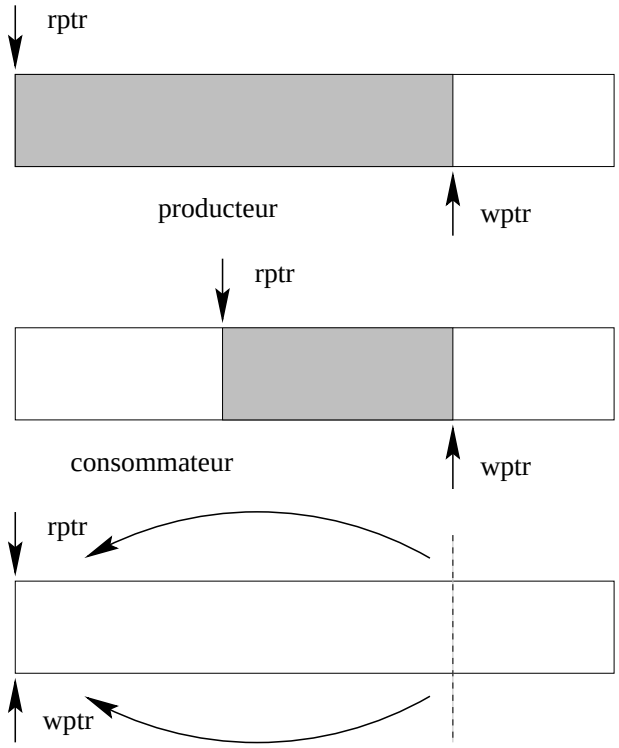


FIG. 4.11 – Le process consommateur est bloqué s'il n'y a rien à lire. Le processus producteur est bloqué si le pipe est plein. Les pointeurs sont remis à 0 s'ils sont égaux.

Pour pouvoir utiliser un pipe, il est nécessaire de posséder le descripteur associé à ce pipe. Cela n'est donc possible qu'au sein d'une filiation de processus, ou en utilisant des processus de communication IPC.

4.4.2 Programmation

L'initialisation d'un pipe se fait par le biais de la fonction :

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

Cette fonction créé un pipe avec 2 descripteurs (`filedes[0]` en entrée, `filedes[1] en sortie`). Elle retourne -1 en cas d'erreur (appel système), 0 sinon. On peut ensuite envoyer (`write()`) ou recevoir (`read()`) des informations sur le pipe. Le seul problème est que l'on ne sait pas `a priori` la taille des informations qui transitent sur le pipe.

On utilise le controleur de flux avec l'option `FIONREAD`. Cela donne un commande de type `ioctl(filedes,FIONREAD,&n)`. Avec `filedes` le descripteur du fichier et `n` l'entier devant recevoir la longueur du pipe.

Un exemple utilisant ce procédé est donné ci-dessous :

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

int main()
{
   int status,pipedes[2];
   char buff[100];
   int len;
   char *msg="Salut Fred !";
   struct stat pipestat;

   if (pipe(pipedes))      exit(1);
   if (fork())
   {  /* Code du pere */
      write(pipedes[1],msg,strlen(msg));
      wait(&status);
      return;
   }
   else
   {   /* Code du fils */
       while(len==0) /* Attend que le pipe soit rempli */
       {
           ioctl(pipedes[0],FIONREAD,&len)
       }
       read(pipedes[0],buf,len);
       printf("Mon pere adore l'art de decaler les sons : %s\n",buf);
       return;
   }

}
```

Si plusieurs processus sont susceptibles d'utiliser un même pipe, il est nécessaire de synchroniser les accès au pipe par signaux (voir plus loin), ou ouvrir 2 pipes utilisés chacun pour un dialogue spécifique (fils vers père et père vers fils par exemple).

Dans l'exemple ci-dessous, le père crée un fils et lui envoie un message, puis le fils accuse réception du message. Les descripteurs inutiles sont fermés.

```
#include <stdio.h>

int pipdes1[2],pipdes2[2];

void fils()
{
  char buff[21];

  /* Fermeture du pipe 1 en ecriture et 0 en lecture */
  close(pipdes1[1]);
  close(pipdes2[0]);

  read(pipdes1[0],buff,21);
  printf("fils : %s\n",buff);
  write(pipdes2[1],"Message recu",13);
  return(0);
}

int main()
{
  int status;
  char bufp[13];

  if (pipe(pipdes1))     exit(1);
  if (pipe(pipdes2))     exit(1);

  if (fork())
    {
      /* Fermeture du pipe 0 en ecriture et 1 en lecture */
      close(pipdes1[0]);
      close(pipdes2[1]);
      write(pipdes1[1],"Salut les process...",21);
      read(pipdes2[0],bufp,13);
      printf("Pere: %s\n",bufp);
      wait(&status);
      exit(0);
    }
  else
    {
      fils();
    }
}
```

Dans les exemples ci-dessus, le fils récupère le descripteur grâce à la conservation des informations de fichiers ouverts par le fork. Par contre, si le fils est mis en place par un exec, les descripteurs de fichiers (et donc de pipe) sont oubliés. Il est donc nécessaire de transmettre le descripteur du pipe au fils.

L'exemple ci-dessous montre comment passer le numéro du pipe en argument.

```
/* Programme du pere */
#include <stdio.h> <sys/types.h> <sys/stat.h>

int main()
{
  int pipdes[2],status,len;
  char msg[80],buf[11];
  struct stat pipestat;

  if (pipe(pipdes))      exit(1);
  /* Conversion du descripteur en chaine de caracteres */
  sprintf(buf,"%d",pipdes[1]);

  if (fork())
    {
      close(pipdes[1]);
      do
        {
          if (fstat(pipdes[0],&pipestat))          exit(2);
          len=(int)pipestat.st_size;
        }
      while(len==0);

      if (read(pipdes[0],msg,len)!=len) exit(3);
      msg[len]=0;
      printf("%s\n",msg);
      wait(&status);
      exit(0);
    }
  else
    {
      execl("pipe_fils1","pipe_fils1",buf,0);
    }
}

/* Programme du fils : pipe_fils1 */

#include <stdio.h> <stdlib.h>

int     main(int argc, char ** argv)
{
  int idpip;
  char *msg="Salut papa, je suis ton fils";

  /* Conversion du premier argument en numero de descripteur */
  idpip=(int)atoi(argv[1]);
  if (write(idpip,msg,strlen(msg))!=strlen(msg))          exit(1);
  exit(0);
}
```

Utilisation d'un descripteur existant Pour éviter de passer le descripteur, il est possible d'exploiter les descripteurs par défaut déjà existants : **STDIN** (0), **STDOUT** (1) et **STDERR** (2). Les processus doivent bien sûr au préalable se mettre d'accord sur la convention d'utilisation de chacun de ces descripteurs.

Pour utiliser un descripteur donné, il faut qu'il soit disponible. On peut alors associer le descripteur du pipe que l'on veut créer au descripteur standard grâce à la fonction `dup()` qui duplique le descripteur passé en paramètre et renvoie la plus petite valeur de descripteur non utilisé :

```
int dup(int oldfd);
```

Pour obtenir un descripteur donné, il faut donc le fermer et utiliser la fonction `dup()` jusqu'à ce que le descripteur voulu soit obtenu... cela peut ^être fastidieux!

La solution consiste à utiliser la fonction `dup2()` qui demande explicitement la duplication d'un descripteur donné :

```
int dup2(int oldfd, int newfd);
```

Dans l'exemple ci-dessous, on fait communiquer 2 processus par un pipe en utilisant le descripteur 0.

Le père lit dans le pipe dans lequel le fils envoie des informations.

```
/* Programme du pere */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

int main()
{
  int pipdes[2],status,len;
  char msg[80];
  struct stat pipestat;

  if (pipe(pipdes))      exit(1);
  if (fork())
    {
      close(pipdes[1]);
      if (dup2(pipdes[0],0)==-1)         exit(2);

      do
        {
          if (fstat(0,&pipestat))        exit(3);
          len=(int)pipestat.st_size;
        }
      while(len==0);

      if (read(0,msg,len)!=len) exit(4);
      msg[len]=0;
      printf("%s\n",msg);
      wait(&status);
      exit(0);
    }
  else
    {
      close(pipdes[0]);
      if (dup2(pipdes[1],1)==-1)         exit(2);
      execl("pipe_fils2","pipe_fils2",0);
    }
}

/* Programme du fils : pipe_fils2 */
#include <stdio.h>

int     main(int argc, char ** argv)
{

  char *msg="Salut papa, je suis ton fils";

  if (write(1,msg,strlen(msg))!=strlen(msg))     exit(1);

  exit(0);
}
```

Exercice : Utiliser la commande `dup()` pour connecter 2 commandes UNIX (*ls* et *sort*) par un pipe. La première commande écrit dans le pipe en utilisant la sortie standard tandis que la deuxième lit le pipe par l'entrée standard. Le processus père crée le pipe puis le processus qui mettra en place la première commande. Par la suite il crée le deuxième processus.

Le processus fils 1 ferme le descripteur 1 pour le rendre disponible, y associe l'entrée du pipe (pour y écrire via STDOUT) puis ferme les descripteurs initiaux du pipe. Il met en place la première commande. Le second processus, ferme le descripteur 0, associe le pipe (pour lire via STDIN), ferme les descripteurs initiaux et met en place la deuxième commande.

Il est *indispensable* de fermer les descripteurs initiaux du pipe pour que celui-ci disparaisse normalement à la mort des processus sinon il y a blocage du deuxième processus (et donc du père) en attente sur la fermeture du pipe.

Le programme réalise l'équivalent de la ligne de commande `ls | sort`. C'est ce principe qui est utilisé par le shell pour effectuer les redirections et la connexion des commandes par pipe. Vous pouvez utiliser ce principe pour créer votre propre shell.

4.5 Les pipes nommés

Un pipe nommé est un tube référencé dans un système de fichiers. Il fonctionne de la même manière que le pipe classique, mais puisqu'il est référencé par un nom, il est possible d'écrire dans ce pipe pour n'importe quel processus.

La commande utilisée pour créer un pipe nommé est :

```
int mkfifo(const char *pathname, mode_t mode);
```

`pathname` est le chemin d'accès au fichier ; `mode` est le mode d'accès au fichier.

Voici comment créer, par exemple, un pipe accessible au propriétaire uniquement.

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

int main()
{
    int mode=0600;

    if (mkfifo("pipe0.###",mode,0)) exit(1);

    exit(0);
}
```

4.6 Le verrouillage

Le verrouillage des fichiers permet de régler les problèmes d'accès concurrents à une ressource critique. Le partage d'un fichier entre plusieurs processus peut en effet entraîner une incohérence du fonctionnement. La solution consiste à verrouiller l'accès au fichier lorsqu'un processus désire y accéder de manière à bloquer les autres processus et à libérer l'accès au fichier dès que possible.

Il existe différents types de verrous :
- Verrou externe : Les options O_CREAT et O_EXCL de la primitive open permettent de gérer les fichiers comme des sémaphores binaires. La forme utilisée est la suivante : L'instruction ne s'exécute que si le verrou est inexistant.
- Verrou consultatif : il n'a pas d'influence sur les primitives E/S.

– Verrou impératif : il peut modifier le fonctionnement d'une primitive E/S (ex : en rendant l'écriture bloquante ou en l'empêchant). Garantit le meilleur niveau de sécurité.
– Verrou partagé et verrou exclusif : un verrou partagé en lecture garantit à plusieurs lecteurs l'accès à un fichier en empêchant la pose d'un verrou exclusif en écriture.

Par défaut un verrou est consultatif et devient impératif après modification du bit sgid.

Exemple de verrou externe :

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

int     main(int argc, char **argv)
{
  char * verrou="my_verrou";
  int fd;


  close(fd);


  if (fork())
    {
      if(fd=open(verrou, O_CREAT | O_EXCL, 0))
        {
          printf("Pere : fd=%d\n",fd);
        }
      else
        printf("Pere : fd=%d\n",fd);
      sleep(10);

      close(fd);
      wait(0);
    }
  else
    {
      if(fd=open(verrou, O_CREAT | O_EXCL, 0))
        {
          printf("Fils : fd=%d\n",fd);
        }
      else
        printf("Fils : fd=%d\n",fd);


      sleep(10);

      close(fd);
    }
}
```

4.6.1 Verrouillage avec la fonction fcntl

```
int fcntl(int fd, int op, struct flock *verrou)
```

fd : descripteur du fichier
verrou : l'adresse du verrou
– F_RDLCK : verrou partagé en lecture
– F_WRLCK : verrou exclusif en écriture
– F_UNLCK : absence de verrou
op : l'une des opérations
– F_GETLK : accès aux caractéristiques d'un verrou existant
– F_SETLK : définition ou modification d'un verrou en mode non bloquant.
– F_ETLKW : définition ou modification d'un verrou en mode bloquant.

4.6.2 Verrouillage par lock

```
int lockf(int fd, int op, long taille)
```

taille : portée du verrouillage à partir de l'enregistrement courant.
– $taille > 0$: verrouillage vers la fin du fichier
– $taille < 0$: verrouillage vers le début du fichier
– $taille = 0$: verrouillage jusqu'à la fin du fichier
op :
– F_LOCK : verrouillage exclusif en mode bloquant
– F_TLOCK : verrouillage exclusif en mode non bloquant
– F_ULOCK : déverrouillage
– F_TEST : test d'existence d'un verrou

4.7 Quiz

1. Quels sont les différents types de fichiers ?
2. Qu'est-ce qu'un *inode* ?
3. Comment accède-t-on à un fichier ?
4. Pourquoi la taille maximum que peut atteindre un fichier sur un disque dont les blocs font 512 octets est 1 Go ? (une adresse prend 4 octets)
5. Quel est le mode de fonctionnement du lien matériel ?
6. Quels sont les effets d'un fork sur un fichier ?
7. Quels sont les effets d'un exec sur un fichier ?
8. Comment et où sont gérés les pipes ?
9. Comment 2 processus distincts peuvent-ils échanger des informations via un pipe (expliquer les différents cas possible : fork, exec, 2 processus sans lien de parenté...)

4.8 Exercices

1. Soient N process *identiques* accédant en parallèle à un fichier contenant un nombre à incrémenter. Ecrire le programme de ces process et utiliser un verrou externe pour synchroniser l'accès. Observez le résultat sans le verrou.
2. Ecrire un programme permettant le dialogue entre un processus père et son fils via le même pipe. Les échanges sont synchronisés par des commandes spécifiques envoyées en fin de pipe.
3. Ecrire un programme permettant à un processus d'interroger un "serveur de calcul" en écoute sur une entrée terminale et le programme permettant à ce dernier de répondre sur le terminal d'où venait l'appel.

Chapitre 5

Les signaux

Un signal est généré par le noyau lorsqu'une exception apparaît au cours de l'exécution du processus de manière à provoquer le traitement particulier correspondant ou **détruire** le processus dont le fonctionnement est de toute façon altéré (comportement par défaut). Cependant, les signaux peuvent aussi être générés par les processus eux-mêmes de manière à communiquer entre eux. En effet, du fait de l'autonomie des processus, ces derniers peuvent accéder de manière concurrente à une ressource donnée. S'ils veulent travailler de concert, un système de signalisation doit être mis en place.

ATTENTION

Il ne faut pas confondre Interruption et Signal. En effet, une interruption est immédiatement traitée par le processeur puis le noyau dès la détection d'une interruption par le ou les gestionnaires d'interruption hardware de la machine. Les signaux, quant à eux, sont uniquement visibles par les processus quand ceux-ci disposent de la ressource processeur (les signaux ne sont pas "temps réel").

5.1 Principaux signaux

La liste des signaux utilisés sous Linux est fournie dans le fichier header `/usr/include/bits/signum.h`.

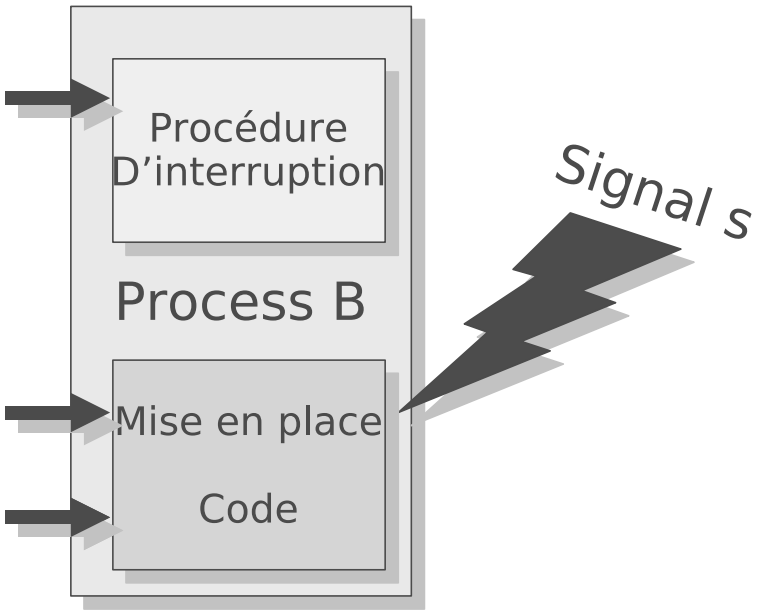
```
#define SIGHUP          1       /* Hangup (POSIX).  */
#define SIGINT          2       /* Interrupt (ANSI).  */
#define SIGQUIT         3       /* Quit (POSIX).  */
#define SIGILL          4       /* Illegal instruction (ANSI).  */
#define SIGTRAP         5       /* Trace trap (POSIX).  */
#define SIGABRT         6       /* Abort (ANSI).  */
#define SIGIOT          6       /* IOT trap (4.2 BSD).  */
#define SIGBUS          7       /* BUS error (4.2 BSD).  */
#define SIGFPE          8       /* Floating-point exception (ANSI).  */
#define SIGKILL         9       /* Kill, unblockable (POSIX).  */
#define SIGUSR1         10      /* User-defined signal 1 (POSIX).  */
#define SIGSEGV         11      /* Segmentation violation (ANSI).  */
#define SIGUSR2         12      /* User-defined signal 2 (POSIX).  */
#define SIGPIPE         13      /* Broken pipe (POSIX).  */
#define SIGALRM         14      /* Alarm clock (POSIX).  */
#define SIGTERM         15      /* Termination (ANSI).  */
#define SIGSTKFLT       16      /* Stack fault.  */
#define SIGCLD          SIGCHLD /* Same as SIGCHLD (System V).  */
#define SIGCHLD         17      /* Child status has changed (POSIX).  */
#define SIGCONT         18      /* Continue (POSIX).  */
#define SIGSTOP         19      /* Stop, unblockable (POSIX).  */
#define SIGTSTP         20      /* Keyboard stop (POSIX).  */
#define SIGTTIN         21      /* Background read from tty (POSIX).  */
#define SIGTTOU         22      /* Background write to tty (POSIX).  */
#define SIGURG          23      /* Urgent condition on socket (4.2 BSD).  */
#define SIGXCPU         24      /* CPU limit exceeded (4.2 BSD).  */
#define SIGXFSZ         25      /* File size limit exceeded (4.2 BSD).  */
#define SIGVTALRM       26      /* Virtual alarm clock (4.2 BSD).  */
#define SIGPROF         27      /* Profiling alarm clock (4.2 BSD).  */
#define SIGWINCH        28      /* Window size change (4.3 BSD, Sun).  */
#define SIGPOLL         SIGIO   /* Pollable event occurred (System V).  */
#define SIGIO           29      /* I/O now possible (4.2 BSD).  */
#define SIGPWR          30      /* Power failure restart (System V).  */
#define SIGSYS          31      /* Bad system call.  */
```

5.2 Lancer un signal

L'envoi d'un signal se fait par la commande `kill`.

Prototype : `int kill(int pid, int sig)`
- Si `sig=0`, `kill` vérifie l'existence du processus `pid` sans envoyer le signal.
- Sinon :
  - Si `pid>0`, envoie le signal `sig` au processus `pid`.
  - Si `pid=0`, envoie le signal à tous les processus ayant le même numéro de groupe que le processus émetteur.
  - Si `pid=-1`, envoie le signal à tous les processus ayant le même numéro d'utilisateur que le processus émetteur.
  - Si `pid<-1`, envoie le signal à tous les processus dont le numéro de groupe est `pid`.

5.3 Mise en place des routines d'exception



5.3.1 La fonction signal

Un processus peut intercepter des signaux de manière à traiter un événement particulier. La mise en place du traitement du signal se fait en utilisant la fonction `signal`.

Prototype : `void *signal(int sig, void (*fonction()))`

- Si `fonction=SIG_DFL` : L'action effectuée est celle faite par défaut.
- Si `fonction=SIG_IGN` : Le signal est ignoré.
- Si `fonction` est une adresse : Le signal provoque l'exécution de la fonction lors de la réception du signal.
- Lors de la réception d'un signal, le système remet en place l'action par défaut avant d'exécuter la fonction. Celle-ci doit donc prévoir de réarmer le signal.
- L'arrivée d'un signal pendant une E/S peut provoquer une erreur E/S. Il faut alors réitérer la requête.
- Les traitements de signaux définis par un processus sont transmis aux processus fils mais un exec remet en place l'action par défaut.

– `signal` retourne l'adresse de la fonction associée au signal avant changement ou la valeur -1 en cas d'erreur.

5.3.2 Autres fonctions de traitement des signaux

– `sigset` : équivalent à `signal` mais masque le signal dans la fonction.
– `sighold` : ignore le signal et l'ajoute au masque.
– `sigrelse` : retire le signal du masque.
– `sigignore` : ignore le signal.
– `sigpause` : retire le signal du masque et attend un signal.

Dans le cas d'une gestion portable des signaux, on utilisera avantageusement :
– `sigaction` : ajout de routines de traitement des signaux.
– `struct sigaction` : strcture définissant les opérations a effectuer en fonction des différents signaux.
– `sigfillset, sigemptyset` : manipulation de la structure sigaction.
– `sigprocmask` : Blocage, déblocage de signaux.
– ...

> Si le père met en place le traitement `SIG_IGN` sur le signal `SIGCLD` et fait un `wait`, le père sera bloqué jusqu'à la mort de tous ses fils. `wait` retourne alors -1. Si le père ne fait pas de `wait`, les fils ne passeront pas dans l'état zombie.

5.3.3 Fonctions pause et alarm

La fonction `pause`, met le processus appelant en attente d'un signal. `pause` retourne -1 avec errno=EINTR si le signal est capturé par le processus.

La fonction `alarm(int n)` active le signal `SIGALARM` au bout de n secondes. Si n=0, il y a annulation des demandes précédentes. C'est un mécanisme simpliste de gestion de timer.

`alarm` retourne le temps qui restait jusqu'à l'activation de la précédente demande. Si une demande est faite avant que la précédente ne soit satisfaite, la précédente est perdue.

5.3.4 Exemples de programmation des signaux

Dans l'exemple ci-dessous, le programme intercepte le signal **SIGINT** généré par la touche **DEL**.

```
#include <signal.h>

void spint(int sig)
{
    printf("Recu signal d'interruption %d\n",sig);
}

int main()
{
    signal(SIGINT,spint);
    pause();
    exit(0);
}
```

Dans l'exemple suivant, le processus crée un fils, lui envoie un signal et attend sa fin. Le fils met en place le traitement du signal et se met en attente. *Le père fait un sleep pour garantir la mise en place du fils et du traitement du signal.*

```
#include <stdio.h>

void spsig(int sig)
{ printf("Signal %d recu\n",sig); }
int main()
{
   int idfils, status;

   if (idfils=fork())   {   /* pere */
      sleep(5);
      kill(idfils,SIGUSR1);
      wait(&status);
      exit(0);
   }
   else   {   /* fils */
      signal(SIGUSR1,spig);
      pause();
      exit(1);
   }
}
```

5.4 Les signaux temps réel

Le nombre de signaux classiques libres pour l'utilisateur étant très limité (SIGUSR1 et SIGUSR2), la norme POSIX.1b a imposé la présence de 8 signaux (au minimum) dits temps réel. Linux, pour sa part, supporte 32 signaux temps-réel numérotés de 32 (SIGRTMIN[1]) à 63 (SIGRTMAX).

ATTENTION

Le qualificatif "temps-réel" est ici trompeur puisqu'il n'impose a priori aucune contrainte de temps au niveau de l'ordonnanceur de tâches. Ainsi, le scheduler délivrera le signal à un processus uniquement quand celui-ci sera ordonnancé.

Contrairement aux signaux standards, les signaux temps-réel n'ont pas de signification prédéfinie : l'ensemble complet de ces signaux peut être utilisée à des fins spécifiques à l'application[2].

Comme pour la plupart des signaux, l'action par défaut pour un signal temps-réel non capturé est de terminer le processus récepteur.

Les signaux temps-réel se distinguent par contre de leurs homologues classiques par des propriétés spécifiques :

1. Plusieurs instances d'un signal temps-réel peuvent être empilées (c'est-à-dire mémorisées puis transmis). Au contraire, si plusieurs instances d'un signal standard arrivent alors qu'il est bloqué, une seule instance sera mémorisée.

[1] Les applications doivent toujours se référer aux signaux temps-réel en utilisant la notation SIGRTMIN+n, car la plage des numéros des signaux varie suivant les Unix

[2] Notez quand même que l'implémentation LinuxThreads utilise les trois premiers signaux temps-réel

2. Un signal temps-réel peut être envoyé via la commande `kill()`, comme pour un signal classique. Cependant, le signal peut aussi être envoyé en utilisant `sigqueue()`, et être alors accompagné d'une valeur (un entier ou un pointeur). Si le processus récepteur positionne un gestionnaire en utilisant l'attribut `SA_SIGINFO` de l'appel `sigaction()` il peut alors accéder à la valeur transmise dans le champ `si_value` de la structure `siginfo_t` passée en second argument au gestionnaire. De plus, les champs `si_pid` et `si_uid` de cette structure fournissent le PID et l'UID réel du processus émetteur.
3. Si différents signaux temps-réel sont envoyés au processus, ils sont délivrés en commençant par le signal de numéro le moins élevé (le signal de plus fort numéro est celui de priorité la plus faible). Par contre, les divers signaux temps-réel du même type sont délivrés dans l'ordre où ils ont été émis.

Si des signaux standards et des signaux temps-réel sont simultanément en attente pour un processus, Posix ne précise pas d'ordre de délivrance. Linux, pour sa part donne lui la priorité aux signaux temps-réel.

D'après POSIX, une implémentation doit permettre l'empilement d'au moins _POSIX_SIGQUEUE_MAX (32) signaux temps-réel pour un processus. Linux, lui, jusqu'au noyau 2.6.7 inclus, impose une limite pour l'ensemble des signaux empilés sur le système pour tous les processus. Cette limite peut être consultée, et modifiée via le fichier `/proc/sys/kernel/rtsig-max`. Le fichier `/proc/sys/kernel/rtsig-nr`, pour sa part, indique combien de signaux temps-réel sont actuellement empilés. Dans Linux 2.6.8, ces interfaces /proc ont été remplacées par la limite de ressources `RLIMIT_SIGPENDING`, qui spécifie une limite par utilisateur pour les signaux empilés[3].

5.5 Quiz

1. Quels sont les 2 grands genres de signaux générés par UNIX ?
2. Quels sont, pour l'utilisateur, les possibilités de générer de nouveaux signaux ?
3. Est-il possible d'ignorer tous les signaux ?
4. Quelle serait la commande permettant de tuer l'ensemble des processus d'un utilisateur ?

5.6 Exercices

1. Ecrire un programme capable de récupérer une erreur de calcul. Tester sur un exemple.
2. Ecrire un programme permettant de lancer n processus en parallèle et de les synchroniser pour qu'ils effectuent *ensemble* un traitement particulier ?
3. Ecrire un programme permettant de réaliser un traitement donné tous les n secondes.

[3] voir `setrlimit()` pour plus de détails

Chapitre 6

Gestion de la communication inter-processus

6.1 Problématique

Les processus sont des entités entièrement indépendantes ayant chacune leur propre cycle de vie. Si les échanges d'informations au sein d'une même filiation sont relativement aisés, ils sont plus problématiques lorsqu'il s'agit de processus issus de filiations différentes. En effet, au sein d'une même filiation de processus, les informations peuvent circuler de différentes manières :
- Par le biais de la connaissance des pid et ppid et des valeurs de retour des exit
- Par les signaux
- Par les initialisations faites dans un processus père et héritées par les processus fils (il faut noter cependant qu'après le fork, aucune modification des données n'est plus possible).
- Par passage de paramètres lors d'un exec (la encore, aucune modification ultérieure n'est possible)
- Par ouverture de pipes

Par contre, dès qu'il s'agit de communication entre processus issus de filiation distinctes, il est beaucoup plus difficile à un processus d'adresser un autre processus puisqu'il ne connaît pas a priori son numéro PID.

Au regard de ce qui a été vu jusqu'à présent, on peut cependant proposer des méthodes qui consistent à communiquer grâce à des canaux définis a priori. Par exemple, des processus peuvent communiquer via des fichiers partagés sur lesquels ils se synchronisent ou des pipes nommés, ou encore, des canaux de communications existants (STDIN, STDOUT, STDERR).

Cependant, les ressources utilisées pour mettre en place ces procédés sont lourdes à gérer et ne sont pas facile à mettre en oeuvre. C'est pourquoi ont été mis en place des mécanismes de communication entre processus.

6.2 Les Inter Process Communication (IPC)

Il existe 3 types de mécanismes de communication inter-processus. Ces 3 IPC sont :
- Les queues de messages

  - Sortes de boites aux lettres où les processus peuvent déposer ou récupérer des messages.
- La mémoire commune
  - allocation d'une zone de mémoire partageable entre processus. Il n'y a pas de transfert d'informations. Le segment est directement adressable par le processus.
- Les sémaphores
  - Ensembles de compteurs mis à jour par des processus pour gérer les accès à des ressources communes.

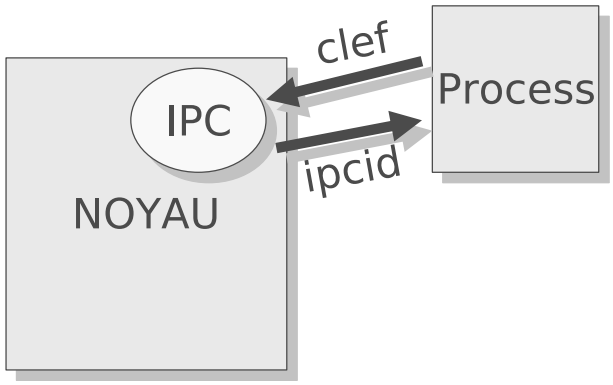
En fait, les sémaphores ne sont pas à proprement parler des mécanismes de communication au sens large, mais des indicateurs comptabilisant les accès à des ressources partagées, et, en particulier, aux autres types d'IPC.

En terme de performances croissantes, on peut classer les moyens de communications entre processus de la manière suivante :
- Fichiers
- Pipes
- Queues de messages
- Sémaphores
- mémoire commune

pour ce qui est de la rapidité d'accès des communications entre processus.

6.2.1 Principes généraux sur l'implémentation des IPC

Les IPC fonctionnent sur un principe consistant à offrir une clef publique permettant d'accéder à une ressource IPC donnée spécifiée par son ID. Le mode d'accès à la ressource est définie par le processus qui l'a créé mais peut être modifié au cours du temps. En fait, la gestion des IPC se fait par un mécanisme du noyau complètement indépendant, ce qui permet d'accéder et de modifier de manière externe les caractéristiques de l'IPC.



6.2.2 Queues de message

Quand plusieurs processus veulent communiquer, la gestion de pipes peut devenir extrêmement complexe. C'est pour pallier ce problème que le mécanisme de queues de messages a été mis au point.

Les queues de messages sont des sortes de boites aux lettres où un processus peut venir déposer ou retirer des messages s'il en possède la clef. Chaque message peut être associé à un type, ce qui permet à un processus de ne sélectionner que les messages qui l'intéressent.

Tous les processus voulant utiliser une queue de messages doivent partager la même clef. Par ailleurs, chaque processus se voit attribuer un numéro auquel correspondra un type de message et qui servira à identifier le destinataire.

Une queue de messages est définie par
– un identifiant `msqid`
– une structure `msqid_ds` contenant les informations associées à la queue.

Implémentation

La fonction `int msgget(key_t key,int msgflg )` permet de créer une queue de messages associée à la clef `key_t key`. Cette fonction renvoie l'identificateur de la queue de messages. Et `msgflg` est un indicateur permettant de spécifier les droits d'accès à la queue ainsi que le mode de création (`IPC_CREAT` – pour demander la création – `IPC_EXCL` – pour empêcher l'utilisation par un autre process).

– si `key` = `IPC_CREAT` = 0, il y a création d'une nouvelle queue avec un `msqid` différent à chaque appel. il n'y a pas de clef associée à la queue.
– si `key` $\neq 0$
  – si la queue n'existe pas et si `IPC_CREAT` est positionné, elle est créée
  – si la queue existe, la fonction retourne son `msqid` ou une erreur si `IPC_CREAT` et `IPC_EXCL` sont positionnés.

L'exemple suivant crée une queue de message associée à la clef $21_{(h)}$ ou récupère son identifiant s'il existe.

```
#include <sys/types.h> <sys/ipc.h> <sys/msg.h>

int main()
{
    ket_t key=0x21;

    int msqid,opperm=0777;

    if ((msqid=msgget(key,opperm | IPC_CREAT))==-1)
    {
        perror("mssget");
        exit(1);
    }

    printf("clef : %x    msqid : %d\n",key,msqid);
    exit(0);
}
```

La fonction `int msgctl(int msqid,int cmd, struct msqid_ds *buf)` permet la modification ou la récupération des informations de contrôle de la queue de message (numéro d'utilisateur et de groupe utilisant la queue, permissions et longueur maximum) :

– si `cmd` vaut `IPC_STAT`, la fonction récupère les informations associées à la queue de messages et les place dans la structure `buf`.
– si `cmd` vaut `IPC_SET`, la fonction permet de modifier les informations associées à la queue de messages.
– si `cmd` vaut `IPC_RMID`, la fonction permet de supprimer la queue de messages.

Dans l'exemple suivant, le programme permet d'afficher les paramètres d'une queue de messages dont l'identifiant est passé en paramètre.

```
#include <sys/types.h> <sys/ipc.h> <sys/msg.h> <stdio.h> <stdlib.h>

int main(int arc, char * argv[])
{
   extern int errno;
   int msqid;
   struct msqid_ds   buf;

   if (argc!=2)   perror("Nb d'arguments. incorrect");
   else           msqid=atoi(argv[1]);

   if (msgctl(msqid,IPC_CREAT,&buf)==-1)
   {
      perror("msgctl");
      exit(1);
   }
   printf("User ID : %d\n",buf->msg_perm.uid);
   printf("Groupe ID : %d\n",buf->msg_perm.gid);
   printf("Permissions : %d\n",buf->msg_perm.mode);
   printf("Longueur : %d\n",buf->msg_qbytes);
   printf("Nb de msg. : %d\n",buf->msg_qnum);
   exit(0);
}
```

La fonction `int msgsnd(int msqid, struct msgbuf *msgp, int msgsz, int msgflg)` envoie `msgsz` octets du message contenu dans la structure pointée par `msgp` dans la queue dont l'identifiant est `msqid`. `msgflg` précise le comportement de la fonction en cas d'impossibilité d'envoi :
– si `msgflg` vaut 0, le processus est bloqué jusqu'à ce que l'écriture soit possible
– si `msgflg` vaut `IPC_NOWAIT`, il y a retour sans attente avec la valeur -1
Le programme suivant envoie un message constant et de type donné en paramètre dans la queue de messages (spécifiée en paramètre également).

> Nb : Notez la déclaration d'une structure `msgb` différente de la structure pré-déclarée `msgbuff` due à une spécification de longueur du membre `msgbuf.mtext[1]` non convenable (vérifier dans `/usr/include/sys/msg.h`)

```
#include <sys/types.h> <sys/ipc.h> <sys/msg.h>
#include <stdio.h> <stdlib.h> <string.h>

int main(int arc, char * argv[])
{
   int msqid;
   struct msgb {
                  long mtype;
                  char mtext[80];
               }  msgp;
   char *msg="Nul n'est jamais assez fort pour ce calcul";

   if (argc!=3)   perror("Nb d'arguments. incorrect");
   else
   {
      msqid=atoi(argv[1]);
      msgp.mtype=atoi(argv[2]);
   }
   strcpy(msgp.mtext,msg);

   if(msgsnd(msqid,&msgp,strlen(msg),IPC_NOWAIT)==-1)
   {
      perror("msgsnd");
      exit(1);
   }
   exit(0);
}
```

La fonction `int msgrcv( int msqid, struct msgbuf *msgp, int msgsz, long msgtyp, int msgflg )` est l'équivalent de la fonction `msgsnd()` pour la réception. Elle demande la récupération de `msgsz` octets d'un message de type `msgtyp` dans la queue de messages identifiée par `msgid`. Le message est stocké dans la structure pointée par `msgp` et retourne le nombre d'octets lus, si l'appel est réussi, ou -1 sinon.

– si `msgtyp` = 0, le premier message dans la queue est lu
– si `msgtyp` > 0, le premier message dont le type correspond est renvoyé
– si `msgtyp` < 0, le premier message dont le type est le plus petit inférieur ou égal à $|msgtype|$ est renvoyé.

`msgflg` précise le comportement de la fonction en cas d'impossibilité de lecture du message :

– si `MSG_NOERROR` est positionné, le message est retourné tronqué à la valeur demandée ; sinon la fonction retourne une erreur. *NB : le reste du message est perdu*
– si `IPC_NOWAIT` est positionné, la fonction retourne une erreur dans le cas où il n'existe pas de message du type désiré. Sinon, le processus est mis en attente d'un message de ce type.

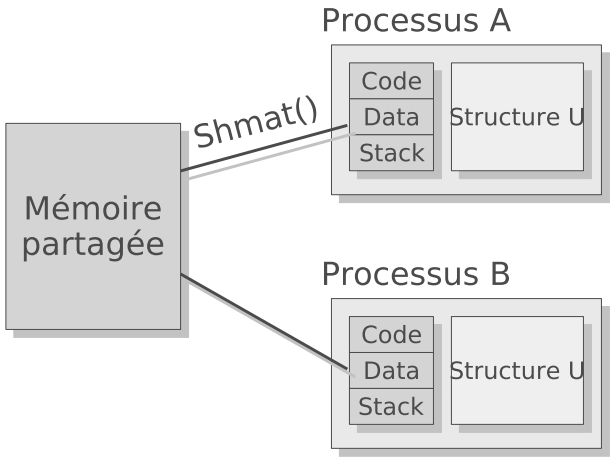
Le programme suivant demande la lecture d'un message donné. Les caractéristiques du message sont passées en paramètres.

```
#include <sys/types.h> <sys/ipc.h> <sys/msg.h>
#include <stdio.h> <stdlib.h> <string.h>

int main(int arc, char * argv[])
{
   int lg,msqid,rtn,flags;
   long mtype;
   struct msgb {
                    long mtype;
                    char mtext[80];
                }   msgp;

   if (argc!=4)   perror("Nb d'arguments incorrect");
   else
   {
      msqid=atoi(argv[1]);
      mtype=atoi(argv[2]);
      lg=atoi(argv[3]);
   }
   flags = IPC_NOWAIT | MSG_ERROR;

   if((rtn=msgrcv(msqid,&msgp,lg,mtype,flags))==-1)
   {
      perror("msgrcv");
      exit(1);
   }
   msgp.metxt[rtn]=0; /* fin de chaine */
   printf("%d caracteres : %s\n",rtn,msgp.mtext);
   exit(0);
}
```

6.2.3 Mémoire partagée

La mémoire partagée permet d'attacher un segment de mémoire à un ou plusieurs processus. Après initialisation, la mémoire s'utilise avec les fonctions C courantes. C'est un énorme avantage puisqu'il n'est pas nécessaire de disposer de fonctions spécifiques comme c'est le cas pour les queues de messages, et, de plus, la communication est immédiate.

L'attachement du segment se déroule en 2 temps :
– La création du segment partagé (`shmget`).
– L'attachement proprement dit (`shmat`).



Implémentation

Les fonctions utilisées pour contrôler la mémoire partagée sont très semblables à celles utilisées pour la gestion des queues de messages.

La fonction `int shmget(key_t key, int size, int shmflg);` retourne l'identifiant du segment associé à la clef `key` ou -1 en cas d'erreur. Le flag `shmflg` sert à déterminer les droits d'accès au segment et le mode de création (`IPC_CREAT` et `IPC_EXCL`). Dans le cas où le flag `IPC_CREAT` est mis, ou que `IPC_PRIVATE` est positionné, un nouveau segment de taille `size` est créé.

L'exemple suivant montre l'emploi de `shmget()` pour créer un segment mémoire partagée de 100 octets associé à la clef 1 et accessible par tous les utilisateurs.

```
#include <sys/types.h> <sys/ipc.h> <sys/shm.h>
#include <stdio.h> <stdlib.h> <string.h>

int main(int arc, char * argv[])
{
    int shmid;
    if ((shmid=shmget((key_t)1,100,IPC\_CREAT+0666))==-1)
    {
        perror("shmget");
        exit(1);
    }
    printf("%s%d\n","L'identifiant memoire est ",shmid);
}
```

La fonction `int shmctl(int shmid, int cmd, struct shmid_ds *buf);` est utiliser pour manipuler la structure de données associée au segment mémoire.

Si `cmd` =
- `IPC_STAT` : la fonction recopie le contenu de la structure `shmid_ds` dans la structure pointée par `buf`
- `IPC_SET` : la fonction remplace les éléments utilisateurs, groupe et droits d'accès du membre `shm_perm` de la structure `shmid_ds` par ceux contenus dans la structure pointée par `buf`
- `IPC_RMID` : la fonction détruit le segment mémoire et sa structure de données (ne pas détruire le segment tant qu'un processus y est attaché)
- `SHM_LOCK` : la fonction verrouille le segment mémoire (le processus doit avoir les droits super-utilisateur)
- `SHM_UNLOCK` : la fonction déverrouille le segment (même restriction que pour `SHM_UNLOCK`)

La fonction `char *shmat( int shmid, char *shmaddr, int shmflg)` attache le segment mémoire identifié par `shmid` au segment de données du processus. Ce mécanisme s'effectue différemment suivant que `shmflg` prend la valeur :
- 0 : le segment est attaché à l'adresse déterminée par le système
- $\neq 0$ et `SHM_RND` n'est pas positionné : le segment est attaché à l'adresse spécifiée par `shmaddr` (qui doit être un multiple de `SHMLBA` (frontière de page)
- $\neq 0$ et `SHM_RND` est positionné : le segment est attaché à l'adresse donnée par l'expression (`shmaddr`-(`shmaddr` modulo `SHMLBA`)). Elle est cadrée par le système au plus proche multiple de `SHMLBA`
- si `SHM_RDONLY` est positionné : le segment est attaché en lecture seulement

La fonction `int shmdt ( char *shmaddr)` détache le segment mémoire situé à l'adresse `shmaddr` (adresse fournie par `shmat()`.

L'exemple suivant montre comment 2 processus peuvent utiliser la mémoire commune pour se transmettre des messages. Les 2 processus partagent le segment mémoire associé à la clef 1 et utilisent

le premier octet comme indicateur de disponibilité. La longueur du message est limitée par la taille du segment.

Pour utiliser cet exemple, il faut au préalable lancer le programme utilisant `shmget()` présenté plus haut. Lancer ensuite `shm1` puis `shm2`.

Le processus associé à `shm1` va placer un message dans le segment mémoire et attendre qu'il soit lu. Le processus associé à `shm2` attend qu'un message soit disponible, le récupère, remet à jour l'indicateur, puis affiche le message.

```c
/* shm1.c */
#include <sys/types.h> <sys/ipc.h> <sys/shm.h>

int main(int arc, char * argv[])
{
   int shmid;
   char *buf;
   char *msg="Ceci est un message transmis par la mémoire commune";

   if ((shmid=shmget((key_t)1,0,0))==-1) {
      perror("shmget");
      exit(1); }
   if ((buf=shmat(shmid,0,0))== (char *)-1) {
      perror("shmat");
      exit(1); }
   *buf=0;                 /* segment libre */
   strcpy(buf+1,msg);      /* transmission du message */
   *buf=1;                 /* message disponible */
   while(*buf);            /* attend que le segment soit libere */
   shmdt(buf);             /* detachement */
}
```

```c
/* shm2.c */
#include <sys/types.h> <sys/ipc.h> <sys/shm.h>

int main(int arc, char * argv[])
{
   int shmid;
   char *buf;
   char msg[100];

   if ((shmid=shmget((key_t)1,0,0))==-1) {
      perror("shmget");
      exit(1); }
   if ((buf=shmat(shmid,0,0))== (char *)-1) {
      perror("shmat");
      exit(1); }
   while(*buf!=1);         /* attend que le message soit disponible */
   strcpy(msg,buf+1);      /* lecture message */
   *buf=0;                 /* segment libre */
   printf("%s\n",msg);     /* affichage du message */
   shmdt(buf);             /* detachement */
}
```

6.2.4 Sémaphores

Certaines ressources ne peuvent pas être utilisées en parallèle, soit pour des raisons physiques (imprimante, fichier, pipe...) soit pour des raisons de préservation de l'intégrité des données. Nous illustrons ce cas sur l'exemple suivant :

Exemple d'accès à une ressource critique Exemple : opération bancaire. Une opération bancaire, est une opération délicate puisque la moindre perturbation dans la transaction peut entraîner des erreurs dans le calcul des sommes en jeu. Nous nous proposons de modéliser une transaction bancaire par les 3 opérations suivantes :

- LIRE X (qui lit le solde du compte considéré)
- ADD C (qui ajoute un crédit C à ce solde)
- ECR X (qui sauvegarde ce solde)

Considérons maintenant 2 processus en concurrence, P1 et P2. On peut alors imaginer les 2 séquences possibles suivantes :

	P1	P2			
Opération	LIRE X	LIRE X	ou	LIRE X	-
	ADD 1	ADD 1		ADD 1	-
	ECR X	-		ECR X	-
	-	ECR X		-	LIRE X
	-	-		-	ADD 1
	-	-		-	ECR X
Résultat	X =	X + 1		X =	X + 2

Comme on le voit, 2 modes opératoires conduisent à des configurations différentes.

Ce type de ressources est appelé "ressource critique". Pour réglementer les conflits d'accès à ces ressources ont été mis en place des indicateurs, appelés sémaphores, qui empêchent l'accès simultané de plusieurs processus à une ressource partagée.

Le principe est de décrémenter un compteur lors de l'accès à la ressource et de l'incrémenter lors de sa libération.

Le sémaphore ne peut devenir négatif. Un processus qui demande à décrémenter un tel sémaphore est mis en attente.

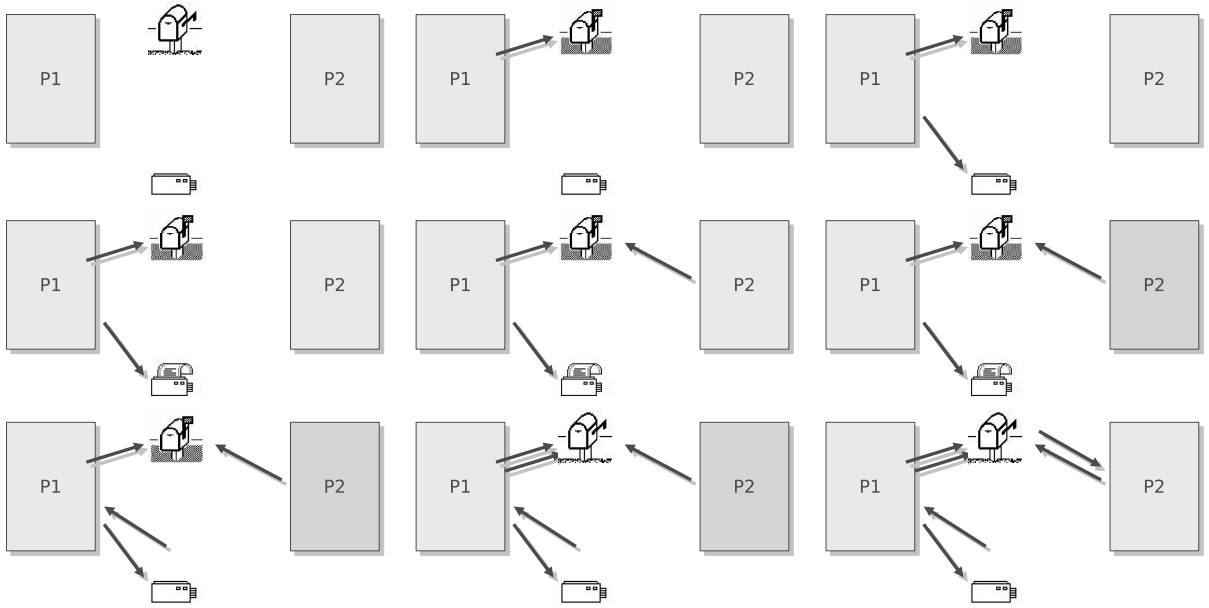
6.2.5 Implémentation

Sous UNIX, on accède à un *ensemble* de sémaphores dont chaque élément peut prendre n'importe quelle valeur et être modifié par -1 ou +1. On utilise souvent un ensemble de 1 sémaphore (pour raisons de complexité).

ATTENTION

Le sémaphore est un ***indicateur*** *de disponibilité. Rien n'empêche d'accéder à la ressource en outrepassant l'avertissement donné par le sémaphore.*

- Un processus crée un ensemble de sémaphores à l'aide de la fonction `semget()`. A cet ensemble de sémaphores est associé un identifiant `semid` et une structure de données `semid_ds` contenant les informations relatives à l'ensemble de sémaphores (nombre de sémaphores, créateur, droits d'accès, heure de modification...)



– L'ensemble contient n sémaphores (numérotés à partir de 0). A chaque sémaphore est associée une structure définissant la valeur courante du sémaphore et les processus en attente sur ce sémaphore.
– La manipulation des sémaphores s'effectue grâce à la fonction `semop()` qui utilise n structures `sembuf` (n est le nombre de sémaphores à manipuler en une opération). Chaque structure définit l'opération à réaliser sur un sémaphore donné.

Considérons le cas d'un sémaphore de l'ensemble qui protège la ressource R. Pour que R soit utilisable, il faut commencer par initialiser le sémaphore à sa valeur maximale. L'initialisation des sémaphores fait partie des opérations effectuées par `semctl()`, celle-ci travaillant directement sur les structures associées à l'ensemble de sémaphores.

– Si un processus désire acquérir la ressource R, il va demander à décrémenter le sémaphore d'une valeur donnée P. Si la valeur courante du sémaphore `semval`$\geq P$, alors `semval` est décrémentée de P et on considère que la ressource est acquise.
– Si `semval`$< P$, alors le sémaphore n'est pas décrémenté et le processus est bloqué en attendant que la valeur du sémaphore augmente suffisamment (libération par un autre processus).
– Si un processus libère la ressource, il retournera la valeur demandée précédemment au sémaphore, qui en sera augmentée d'autant.

Chaque processus manipulant un sémaphore possède une copie `semadj` du nombre de points pris au sémaphore. Cette copie sert à remettre à jour le sémaphore si le processus est tué sans avoir libéré la ressource.

La fonction `int semget(key_t key, int nsems, int semflg)` crée un ensemble de sémaphores associé à la clef `key` ou -1 en cas d'erreur. `semflg` définit les droits d'accès[1] ainsi que le mode de création (`IPC_CREAT` ou `IPC_EXCL`). Un nouvel ensemble de sémaphores est crée si `key` vaut `IPC_PRIVATE` ou si `IPC_CREAT` est positionné et que l'ensemble n'existe pas déjà. `semget()` retourne une erreur si un ensemble est déjà associé à une clef et que `IPC_CREAT` et `IPC_EXCL` sont positionnés.

La fonction `int semctl(int semid, int semnum, int cmd, union semun arg)` permet le contrôle des données associées à l'ensemble de sémaphores identifié par `semid`. Le paramètre `cmd` permet de spécifier l'opération effectuée. Si cette dernière concerne un sémaphore particulier, `semnum` représente le numéro du sémaphore visé. `arg` est une structure à champ variable permettant de manipuler le type d'objet souhaité, selon l'opération `cmd` effectuée.

[1] Pour les sémaphores, les autorisations sont représentées par `r` (**READ**) et `a` (**ALTER**)

La variable `arg` est définie comme ayant la structure `semun` :

```
union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
} arg;
```

Les valeurs (et donc les opérations) pouvant être prises par `cmd` sont :
- `GETVAL` : retourne la valeur `semval` du sémaphore correspondant.
- `SETVAL` : initialise la valeur de `semval` du sémaphore avec la valeur de `arg.val` et remet à 0 la valeur `semadj` du sémaphore dans le processus
- `GETPID` : retourne le PID du processus ayant manipulé le sémaphore pour la dernière fois (`sempid`).
- `GETNCNT` : retourne le nombre de processus en attente d'une valeur suffisante du sémaphore (`smcnt`)
- `GETZCNT` : retourne le nombre de processus en attente du passage à 0 du sémaphore (`semzcnt`)
- `GETALL` : place dans le tableau pointé par `arg.array` les valeurs `semval` des sémaphores
- `SETALL` : initialise les valeurs des sémaphores avec celles contenues dans le tableau pointé par `arg.array` et remet à jour les valeurs `semadj` des sémaphores correspondants dans les processus
- `IPC_STAT` : copie le contenu de la structure associée à l'ensemble de sémaphores dans la structure pointée par `arg.buff`
- `IPC_SET` : initialise la structure associée à l'ensemble de sémaphores avec les informations utilisateur, groupe et permissions contenues dans la structure pointée par `arg.buf`
- `IPC_RMID` : détruit l'ensemble de sémaphores et les structures associées à `semid`

L'exemple suivant montre comment créer un ensemble de 1 sémaphore (!) associé à la cle 10. La valeur du sémaphore est initialisée à 1 :

```
#include <sys/types.h> <sys/ipc.h> <sys/sem.h>

int main(int arc, char * argv[])
{
    int semid;
    union semun {
        int val;
        struct semid_ds *buf;
        ushort array[1];
    }  arg;

    if ((semid=semget((key_t)10,1,IPC_CREAT+0777))==-1) {
        perror("semget");
        exit(1); }
    printf("%s%d","Id semaphore recupere=",semid);
    argv.val=1;
    if (semctl(semid,0,SETVAL,arg)==-1)    {
        perror("semctl");
        exit(1);
    }
    printf("Semaphore initialise");
}
```

La fonction `int semop(int semid, struct sembuf *sops, unsigned nsops)` permet de réaliser un ensemble d'opérations sur `nsops` sémaphores de l'ensemble identifié par `semid`. `semop` est une opération dite "atomique" dans le sens où aucune interruption n'est possible durant la réalisation des

opérations spécifiées dans `sops[]`. Pour chaque sémaphore concerné par l'appel `semop`, l'opération à exécuter `semop` et le numéro de sémaphore correspondant `sem_num` sont précisés dans la structure `sembuf`. L'ensemble des `nsops` structures représentant un tableau pointé par `sops`.

Selon les valeurs de `sem_op` et `sem_flg`, le fonctionnement est le suivant :
- Si `sem_op`< 0 : le processus essaie de diminuer la valeur du sémaphore (acquisition) :
  - si la valeur `semval` du sémaphore est assez grande, `semval` est effectivement diminué de $|sem\_op|$. Si, de plus, le bit `SEM_UNDO` de `sem_flg` est positionné[2], `sem_op` est ajouté à la valeur `semadj` du processus pour le sémaphore correspondant (cela permet de savoir le nombre de "ponts" possédés par le sémaphore).
  - si la valeur du sémaphore est insuffisante pour réaliser l'opération et si le bit `IPC_NOWAIT` de `sem_flg` est positionné, il y a retour immédiat de la fonction
  - si la valeur du sémaphore est insuffisante pour réaliser l'opération et si si le bit `IPC_NOWAIT` de `sem_flg` n'est pas positionné, le processus est suspendu et la valeur de `semncnt` est incrémentée de 1. L'attente se termine lorsque la valeur du sémaphore devient suffisante ou lorsque l'ensemble de sémaphores est retiré du système, ou encore, lorsque le processus reçoit un signal devant être capturé.
- Si `sem_op`> 0 : on ajoute sa valeur à la valeur courante du sémaphore spécifié. Si le bit `SEM_UNDO` de `sem_flg` est positionné, on diminue d'autant la valeur `semadj` du processus pour le sémaphore spécifié[2]
- Si `sem_op`$= 0$: le processus attend que le sémaphore soit à 0 :
  - si le bit `IPC_NOWAIT` est positionné, la fonction renvoie 0 dans le cas où le sémaphore est à 0 et -1 sinon
  - si `semval`$\neq 0$ et que `IPC_NOWAIT` n'est pas positionné, le processus est suspendu et `semzcnt` est incrémenté de 1. L'attente se termine quand `semval` devient nul ou si l'ensemble de sémaphores est détruit, ou encore, si le processus reçoit un signal devant être capturé

[2]L'utilisation du flag `SEM_UNDO` est importante car, si un processus ayant diminué la valeur d'un sémaphore est tué ou se termine, la valeur du sémaphore ne sera restituée que si ce bit était positionné. Dans le cas contraire, les processus en attente du sémaphore peuvent rester bloqués.

L'exemple ci-dessous montre comment utiliser un sémaphore pour contrôler l'accès exclusif à une ressource. Le processus commence par récupérer l'identificateur de l'ensemble puis essaie de diminuer la valeur du sémaphore. S'il obtient satisfaction, il s'endort pendant 10 secondes et restitue sa valeur au sémaphore. Il indique à chaque fois son numéro de processus. La fonction `semcall` prend en charge l'opération à effectuer sur le sémaphore.

```
#include <sys/types.h> <sys/ipc.h> <sys/sem.h>

int semid;

void semcall(int op)
{
   struct sembuf sb;

   sb.sem_num=0;
   sb.sem_op=op;
   sb.sem_flg=SEM_UNDO;
   if (semop(semid,&sb,1)==-1) {
      perror("semop");
      exit(1);
   }
}

int main(int arc, char * argv[])
{
   int pid=getpid();;

   if ((semid=semget((key_t)10,1,0))==-1) {
      perror("semget");
      exit(1); }

   printf("Semaphore %d recupere\n",semid);
   printf("Demande d'acces par %d\n",pid);
   semcall(-1);
   printf("Acces accorde %d\n",pid);
   sleep(10);
   semcall(1);
   printf("Acces libere par %d\n",pid);
}
```

6.2.6 Contrôle des ressources IPC

Les ressources IPC étant contrôlées par le noyau, et non par les processus eux-même, on peut y accéder et les modifier grâce aux commandes UNIX `ipcs` et `ipcrm`.

`ipcs` affiche le nombre de queues de message et de sémaphores créés ainsi que les segments de mémoire.

`icprm` permet de détruire des ressources.

6.3 Quiz

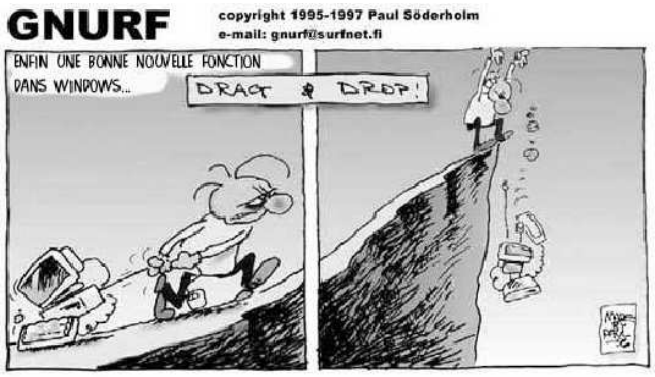
1. Quels sont les moyens de communications entre processus ?
2. Les IPC sont ils gérés par le noyau ?
3. Parmi les IPC, quels sont ceux qui permettent *réellement* de transférer des données .
4. Quels sont les principes de fonctionnement communs à tous les IPC ?
5. Comment est définie la structure associée à une queue de messages ?
6. Quelle est la différence dans l'adressage d'un segment de mémoire commune ?
7. Un sémaphore déclenche-t-il un comportement impératif ?
8. Est-il possible de décrémenter un sémaphore de plus de 1 unité ? Est-il possible de décrémenter un sémaphore si sa valeur est inférieure à 0 ?

6.4 Exercices

Voir TP en annexe

Chapitre 7

Le multithreading



7.1 Introduction

La mise en place de la gestion de processus, comme nous l'avons vu au chapitre 3, consomme beaucoup de ressources système puisqu'à chaque création de processus, il est nécessaire de dupliquer l'ensemble des segments de code, de données et de pile, ainsi que la task_struct. Par ailleurs, lorsqu'on désire paralléliser un algorithme, il est intéressant de pouvoir réutiliser simplement les segments de code et de données dans plusieurs tâches. Enfin, le concept de processus n'est pas forcément bien adapté aux machines multi-processeurs et aux environnement graphiques. C'est pour répondre à ces demandes qu'a été introduit le concept du thread.

7.2 Les threads

7.2.1 Définitions

Un thread est un ensemble d'instructions (on appel cela aussi un "fil" d'instructions) qui sont exécutées dans le contexte d'un processus. Ce sont des entités d'exécution indépendantes appartenant cependant au processus lanceur (voir figure 7.1).

Chaque thread possède son propre pointeur de programme (PC) et sa propre zone de pile afin de pouvoir conserver leur contexte lors de l'ordonnancement. Par contre, les threads partagent le code du processus ainsi que le segment mémoire.

La mise en oeuvre des threads prend tout son intérêt dans un contexte multiprocesseurs (pas nécessairement processeurs de calculs : les threads permettent aussi de tirer partie des co-processeurs). De ce fait, il est possible de faire plusieurs entrées/sortie en parallèle réel (non simulé) et de gérer des évènements simultanés. Ce qui est tout à fait satisfaisant du point de vue de la programmation événementielle (interfaces graphiques).

L'appel d'un thread est similaire à celui d'une fonction mais il n'y a pas d'attente d'exécution de code (pas de retour d'un thread). Le thread est mis en place et s'exécute en parallèle avec le code principal du processus.

7.2.2 Propriétés

De cette définition découle un ensemble les propriétés intéressantes suivantes :

– Comme ils appartiennent à 1 processus, les threads sont invisibles à l'extérieur de celui-ci par les autres processus.
– A l'inverse, il est possible de dialoguer très facilement entre les différents threads attachés à un même processus.
– Plusieurs threads correspondant à une même zone de code peuvent être lancés simultanément ce qui permet de donner l'apparence d'un traitement en parallèle et ainsi d'accélérer les traitements.

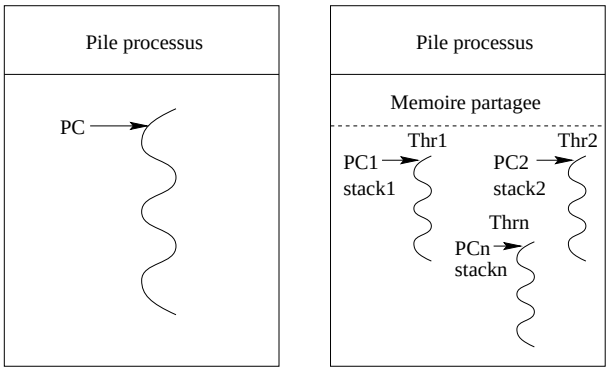


FIG. 7.1 – Différences entre un processus unique et un processus possédant un ensemble de threads

7.3 Mise en oeuvre des threads sous Linux

Pour passer à l'implémentation des threads, il est nécessaire de se poser 2 genre de questions :

1. Est-il nécessaire de modifier le noyau pour rendre accessible la notion de thread ?
2. Quelle est la granularité d'unn thread et quel lien faire avec la notion de processus ?

De la première question découlent 3 type d'implémentations des threads :
– Au niveau utilisateur (user-level). A ce moment la, la gestion des threads est entièrement faite dans l'espace utilisateur.
– Au niveau noyau (kernel-level). Dans ce cas, les threads sont directement gérés par le noyau.
– Implémentation hybride : les threads sont gérés à la fois en mode noyau et en mode utilisateur pour tirer bénéfices des 2.
Nous détaillons ci-dessous les avantages/inconvénients des 2 approches mais il est surtout conseillé de lire le papier d'U.Drepper – développeur principal de la glib – et I.Molnar – développeur noyau et en particulier du sceduler – sur le sujet [Drepper and Molnar, 2002].

7.3.1 Implémentation en mode utilisateur

C'est ce type d'implémentation qui est historiquement apparue en premier. L'idée était en effet de rendre accessible la notion de thread sans pour autant modifier l'architecture du noyau. Ainsi, la gestion des threads (l'exécution, la synchronisation , etc...) est réalisée par l'intermédiaire d'une bibliothèque utilisateur spécifique (la thread library), fonctionnant en mode utilisateur (pas de commutation de contexte). L'intérêt principal est de rendre complètement transparente la gestion des threads pour le noyau. Il n'y a ainsi pas d'appels système effectué pour leur gestion et la commutation entre threads est donc plus rapide.

En pratique, le noyau continue à ne voir que des processus. C'est la bibliothèque thread qui se charge de faire exécuter le code des threads sur des processus particuliers appelés "LightWeight Process (LWP)".

Ces processus sont qualifiés de “légers” dans le sens où ils n'implémentent pas toutes les structures des processus et en particulier, n'ont pas de segment de code (c'est celui du processus principal qu'ils utilisent). Un LWP mis en place par un thread appartenant à un processus donné exploite ainsi l'espace d'adressage et les ressources du process initial avec le contexte d'exécution du thread.

– Le LWP exécute le code du thread, les appels systèmes et les demandes de page. Le LWP se substitue au processus initial pour prendre le contrôle du thread pendant le passage en mode noyau.
– Le noyau, quant à lui, gère le passage des signaux aux différents LWP, l'ordonnancement des LWP et l'affectation des LWP aux différents processus (binding[1] possible).
– La thread library gère l'affectation des threads aux LWP par multiplexage (réservation possible d'un LWP par thread binding), la création de LWP supplémentaires (gestion de nouveaux signaux) et l'interprétation des signaux.

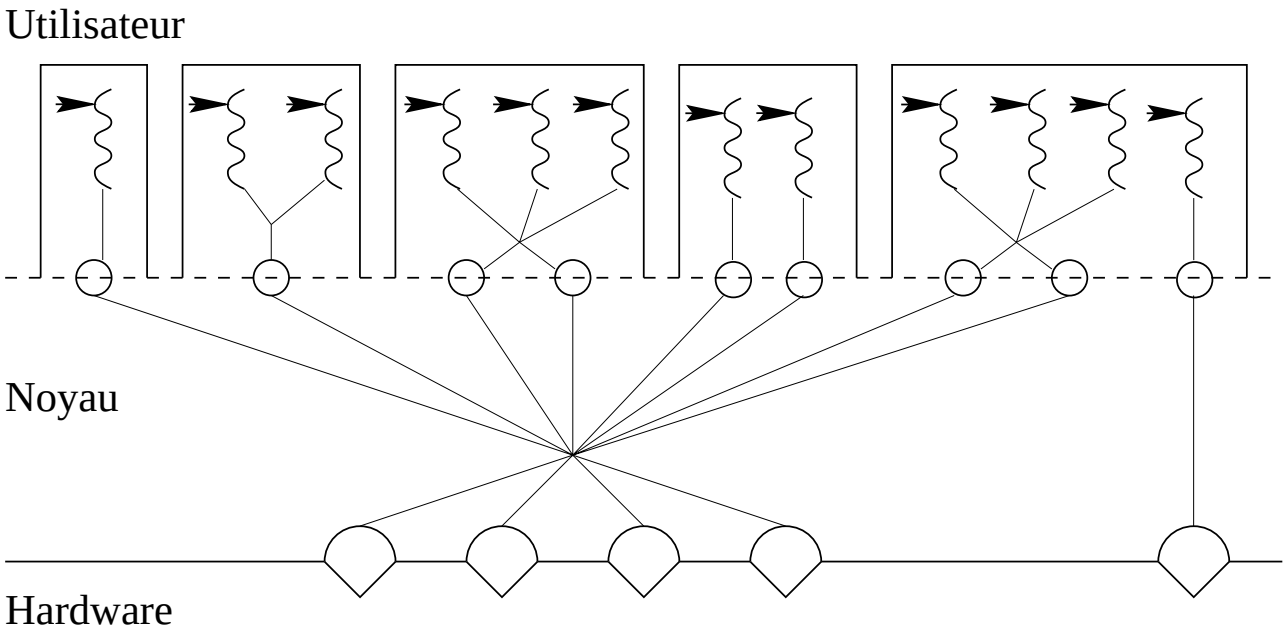


FIG. 7.2 – Différents types de gestion de threads

Implémentation 1 :1 versus M :N

Lorsqu'on pense à implémenter les threads à partir des processus, on s'aperçoit qu'on à un degré de parallélisme supplémentaire. En effet, il est possible soit d'implémenter un LWP par thread (c'est ce qu'on appele l'implémentation 1 :1), soit de considérer que plusieurs thread peuvent tourner sur un même LWP (implémentation M :N).

On peut ainsi avoir les différentes configurations de threads montrées figure 7.2 :
– Un thread associé à un LWP (équivalent à un processus)
– 2 threads associés à un LWP (commutation de thread)
– 3 threads associés à 2 LWP (commutation selon les besoins entre les 3 threads)
– 2 threads associés à 2 LWP différents (binding — pour éviter de changer le contexte d'un LWP on associe un thread à un LWP unique)
– 1 thread associé à un LWP lui-même associé à un processeur (on assure ainsi qu'un thread gros consommateur de calcul a accès en permanence à un processeur)

L'intérêt d'une telle approche est qu'un double niveau d'ordonnancement peut être effectué. En effet, plusieurs threads pouvant être ordonnancé sur un même LWP, il peut être possible d'utiliser le temps

[1] affectation d'un LWP à un processus

alloué à ce processus de manière plus efficace si un processus se met par exemple en attente avant la fin du temps d'ordonnancement du LWP. Cependant, il faut prendre garde à une bonne entente entre le scheduler noyau et le scheduler des threads associé au LWP pour ne pas dégrader les permorfances. Cependant, l'infrastructure nécéssaire à cette entente peut s'avérer lourde...

Un autre intérêt de l'utilisation d'implémentation M :N est en termes de simplification de la gestion des signaux. En effet, dans une implémentation 1 :1, le noyau doit gérer le masque de signaux, qui est propre au processus principal contenant les thread, en le répartissant sur l'ensemble de ces threads. Si le nombre de thread devient grand, alors le temps de traitement des signaux devient prohibitif... Dans une implémentation M :N, par contre, le nombre de LWP rester relativement faible et le traitement des signaux peut s'effectuer en espace utilisateur. La gestion des signaux est cependant un peu particulière. En effet, la gestion des signaux est faite pour la totalité du processus (1 seul handler/masque de signaux). Ainsi, chaque LWP qui prend en compte un thread appartenant à un processus a un masque de signal correspondant à celui du processus de manière à pouvoir masquer tous les signaux que le processus peut masquer. Par contre, chaque thread possède un masque de signal spécifique qui est nécessairement un sous-ensemble de celui du LWP. En définitive, c'est le LWP et la thread library qui filtrent les signaux devant atteindre les threads. Lorsque le noyau émet un signal vers le thread, il l'envoie en fait au LWP gérant ce thread. La threads library interprète alors la distribution du signal en fonction des masques spécifiques de thread. Si le signal est asynchrone, il est délivré au premier thread réceptif. A l'inverse, si le signal est synchrone, il est délivré au thread ayant créé le problème.

Un inconvénient majeurs de l'implémentation M :N en mode utilisateur est que lorsqu'un appel système bloquent le LWP, tous les threads qui lui sont rattachés sont bloqués (même ceux qui ne nécéssitaient pas la ressource bloquante). Un autre inconvénient provient du fait que le noyau ne voyant que les LWP, sa politique de gestion des priorités peut se faire au détriment des threads. En effet, si on imagine 3 processus de même priorité avec l'un d'entre eux contenant 10 threads. Chacun des threads n'aura qu'un trentième du temps alors qu'il devrait en avoir un douzième dans un cas de politique d'ordonnancement égalitaire. Le cas est encore plus criant dans un contexte multi-processeurs. En effet, le noyau ne voyant qu'un LWP, il ne peut pas répartir les threads sur les différentes unités de calcul dont il dispose.

7.3.2 Implémentation en mode noyau

L'inconvénient principal est d'avoir à modifier le noyau pour la prise en compte des Threads. C'est pourquoi ce genre d'implémentation n'est arrivée que plus tardivement. Le grand avantage cependant est de standardiser la notion de tâche exécutable, ce qui régularise leur gestion au niveau du noyau. On notera qu'en réalité l'implémentation des threads ne s'effectue pas entièrement en mode noyau mais plutôt de manière **hybride** puisqu'il est toujours nécéssaire de disposer d'une bibliothèque de fonctions au niveau utilisateur pour pouvoir faire référence aux threads noyau.

7.3.3 Bibliothèques existantes

Il existe plusieurs bibliothèques permettant de manipuler des threads sous LINUX dont l'implémentation correspond à des choix de modèles parmi ceux présentés plus haut. Cependant, la plupart répondent à l'interface de programmation définie par la norme POSIX.1C.

Threads POSIX

La manpage française sur les pthreads (comprenez POSIX Threads – `http ://manpagesfr.free.fr/man/man7/pthreads.7.html`), nous renvoie ce qui suit.

POSIX.1 spécifie un ensemble d'interfaces (fonctions, fichiers d'entête) pour la programmation par

thread, plus connu sous le nom de threads POSIX ou Pthreads. Un seul processus peut contenir plusieurs threads, chacun d'eux exécutant le même programme. Ces threads partagent la même mémoire globale (segments de données et de tas) mais chaque thread possède sa propre pile (variables automatiques).

POSIX.1 réclame également que les threads partagent un ensemble d'autres attributs (c'est-à-dire que ces attributs sont à l'échelle du processus plutôt que spécifiques à chaque thread) :
– ID du processus (PID)
– ID du processus parent (PPID)
– ID du groupe du processus (PGID) et ID de session
– terminal de contrôle
– ID utilisateur (UID) et ID de groupe (GID)
– descripteurs de fichier ouverts
– verrous d'enregistrement (voir fcntl(2))
– dispositions de signaux
– masque de création de fichier (umask(2))
– répertoire courant (chdir(2)) et répertoire racine (chroot(2))
– temporisateurs d'intervalle (setitimer(2)) et temporisateurs POSIX (timer_create(3))
– valeur de "courtoisie" (setpriority(2))
– limites des ressources (setrlimit(2))
– mesures de la consommation de temps CPU (times(2)) et de ressources (getrusage(2))

En plus de la pile, POSIX.1 spécifie divers autres attributs propres à chaque thread, incluant :

– ID de thread (TID) (le type de donnée pthread_t)
– masque de signaux (pthread_sigmask(3))
– la variable errno
– pile de signal alternative (sigaltstack(2))
– politique et priorité d'ordonnacement temps réel (sched_setscheduler(2) et sched_setparam(2))

Les fonctionnalités suivantes, spécifiques à Linux, sont également par thread :

– capacités (voir capabilities(7))
– affinité CPU (sched_setaffinity(2))

Compilation sous Linux Sous Linux, les programmes qui utilisent l'API Pthreads devraient être compilés en utilisant la commande :

```
$cc -pthread
```

Implémentations Linux des threads POSIX Sous Linux, deux implémentations de threading sont fournies par la glibc :

LinuxThreads Il s'agit de l'implémentation Pthreads originale. Depuis la glibc 2.4, cette implémentation n'est plus prise en charge.

NPTL (Native POSIX Threads Library) Il s'agit de l'implémentation Pthreads moderne. En comparaison avec LinuxThreads, NPTL fournit une conformité plus proche des spécifications POSIX.1 et une meilleure performance lorsque l'on crée un grand nombre de threads. NPTL est disponible depuis la glibc 2.3.2 et nécessite des fonctionnalités qui sont présentes dans le noyau Linux 2.6.

Toutes les deux sont appelées implémentations 1 :1, signifiant que chaque thread correspond à une entité d'ordonnacement noyau.

Les deux implémentations utilisent l'appel système Linux clone(2).

Dans la bibliothèque NPTL, les primitives de synchronisation de thread (mutexes, attachement de thread, etc.) sont implémentées en utilisant l'appel système Linux futex(2). LinuxThreads Les fonctionnalités notables de cette implémentation sont les suivantes :

- En plus du thread principal (initial) et des threads que le programme crée avec pthread_create(3), l'implémentation crée un thread "gestionnaire". Ce thread gère la création et la terminaison des threads. (Des problèmes peuvent survenir si ce thread est tué par inadvertance.)
- Les signaux sont utilisés de manière interne par l'implémentation. Sous Linux 2.2 et suivants, les trois premiers signaux temps réel sont utilisés. Sur les noyaux plus anciens, SIGUSR1 et SIGUSR2 sont utilisés. Les applications doivent éviter d'utiliser les signaux qui le sont par l'implémentation.
- Les threads ne partagent pas les PID. (En effet, les threads LinuxThreads sont implémentés comme des processus qui partagent plus d'informations qu'à l'habitude, mais qui ne partagent pas un PID commun.) Les threads LinuxThreads (y compris le thread gestionnaire) sont visibles en tant que processus séparés lorsqu'on utilise ps(1).

L'implémentation LinuxThreads dévie des spécifications POSIX.1 de plusieurs façons, incluant :

- Les appels à getpid(2) renvoient une valeur différente dans chaque thread.
- Les appels à getppid(2) dans les threads autres que le thread principal renvoient le PID du thread gestionnaire ; à la place, getppid(2) dans ces threads devrait retourner la même valeur que getppid(2) dans le thread principal.
- Lorsqu'un thread crée un nouveau processus fils avec fork(2), chaque thread devrait être capable d'attendre (wait(2)) le fils. Toutefois, l'implémentation ne permet qu'au thread qui a créé le fils de l'attendre.
- Lorsqu'un thread appelle execve(2), tous les autres threads sont terminés (comme requis par POSIX.1). Toutefois, le processus résultant a le même PID que le thread qui a appelé execve(2) : il devrait avoir le même PID que le thread principal.
- Les threads ne partagent pas les UID et GID. Cela peut provoquer des complications avec les programmes Set-UID et provoquer des échecs dans les fonctions Pthreads si une application modifie ses références avec seteuid(2) ou une fonction similaire.
- Les threads ne partagent pas un PGID et un ID de session commun.
- Les threads ne partagent pas les verrous d'enregistrement créés avec fcntl(2).
- Les informations renvoyées par times(2) et getrusage(2) sont par thread et non à l'échelle du processus.
- Les threads ne partagent pas les valeurs "undo" des sémaphores (voir semop(2)).
- Les threads ne partagent pas les temporisateurs d'intervalles.
- Les threads ne partagent pas une valeur de "courtoisie" commune.
- POSIX.1 distingue les notions de signaux qui s'adressent au processus dans son ensemble et des signaux qui s'adressent aux threads de manière individuelle. Suivant POSIX.1, un signal s'adressant à un processus (envoyé par kill(2), par exemple) devrait être géré par un seul thread, choisi arbitrairement dans le processus. LinuxThreads ne supporte pas la notion de signaux s'adressant à un processus : les signaux peuvent seulement être envoyés à des threads spécifiques.
- Tous les threads ont des configurations distinctes pour les piles utilisées spécifiquement lors du traitement des signaux. Toutefois, la configuration d'un thread nouvellement créé est copiée sur la configuration de son créateur, aussi partagent-ils la même pile spécifique pour les signaux. (Il serait préférable qu'un nouveau thread démarre sans pile de traitement de signaux, car si deux threads gèrent des signaux en utilisant la même pile, on assistera probablement à des plantages imprévisibles.)

NPTL Avec NPTL, tous les threads d'un processus sont placés dans le même groupe de threads ; tous les membres d'un groupe de threads partagent le même PID. NPTL n'utilise pas de thread

gestionnaire. NPTL utilise de manière interne les deux premiers signaux temps réel ; ces signaux ne doivent donc pas être utilisés dans l'application.

NPTL a aussi au moins une non conformité avec POSIX.1 :

– Les threads ne partagent pas une valeur de courtoisie commune.

Certaines non conformités NPTL peuvent apparaîtrent sur d'anciens noyaux :

– Les informations renvoyées par times(2) et getrusage(2) sont spécifiques aux threads plutôt que d'être à l'échelle du système (corrigé dans le noyau 2.6.9).
– Les threads ne partagent pas les limites de ressources (corrigé dans le noyau 2.6.10).
– Les threads ne partagent pas les temporisateurs d'intervalles (corrigé dans le noyau 2.6.12).
– Seul le thread principal a le droit de démarrer une nouvelle session avec setsid(2) (corrigé dans le noyau 2.6.16).
– Seul le thread principal a le droit de faire d'un processus un leader dans un groupe de processus avec setpgid(2) (corrigé dans le noyau 2.6.16).
– Tous les threads ont des configurations distinctes pour les piles utilisées spécifiquement lors du traitement des signaux. Toutefois, la configuration d'un thread nouvellement créé est copiée sur la configuration de son créateur, aussi partagent-ils la même pile spécifique pour les signaux (corrigé dans le noyau 2.6.16).

Veuillez noter les points suivants concernant l'implémentation NPTL :
– Si la limite de ressource logicielle de taille de pile (voir la description de RLIMIT_STACK dans setrlimit(2)) est définie à une valeur autre que unlimited, cette valeur définit la taille par défaut de la pile pour les nouveaux threads. Pour être effective, cette limite doit être définie avant que le programme ne soit exécuté, peut-être en utilisant la commande shell intégrée ulimit -s (limit stacksize dans le shell C).

Déterminer l'implémentation de threading Depuis la glibc 2.3.2, la commande getconf permet de déterminer l'implémentation de threading du système, par exemple :

```
bash$ getconf GNU_LIBPTHREAD_VERSION
NPTL 2.6.1
```

ou directment en exécutant (!) la bibliothèque C :

```
bash$ /lib/libc.so.6
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 4.2.2 20070909 (prerelease) (4.2.2-0.RC.1
   mdv2008.0).
Compiled on a Linux 2.6.22 system on 2007-11-21.
Available extensions:
        crypt add-on version 2.1 by Michael Glad and others
        GNU Libidn by Simon Josefsson
        Native POSIX Threads Library by Ulrich Drepper et al
        BIND-8.2.3-T5B
For bug reporting instructions, please see:
<http://www.gnu.org/software/libc/bugs.html>.
```

7.3.4 Création d'un thread

La mise en place d'un thread se fait par la commande :

`int pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr, void *(*start_routine)(void*), void *arg)`

– thread : identificateur du thread. Invisible à l'extérieur du processus.
– attr = attributs du thread
– start_routine = adresse de la procédure associée sous forme d'un pointeur sur une fonction
– arg = paramètres passés à la procédure.

Ex : création de 10 threads

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define NTHREADS 10

void *athread (void *arg)
{
  printf ("thread ID: %2d, arg: %2d\n", pthread_self (), (int) arg);
  return (0);
}

int main (int argc, char *argv[])
{
  int i, n;
  pthread_t list[NTHREADS];

  for (i = 0; i < NTHREADS; ++i)
    {
      n=pthread_create (&list[i], 0, athread, (void *)(i*i));

      if (n)
  {
    fprintf (stderr, "thr_create: %s\n", strerror (n));
    exit (1);
  }
    }

  for (i = 0; i < NTHREADS; ++i)
    {
      if (n=pthread_join (list[i], NULL))
  {
    fprintf (stderr, "thr_create: %s\n", strerror (n));
    exit (1);
  }
    }
  return (0);
}
```

7.3.5 Terminaison d'un thread

Pour terminer un thread, il faut demander sa terminaison en utilisant la fonction `void pthread_exit(void *status);`.

Cette fonction force la fin d'un thread et passe la valeur status qui sera récupérée par `int pthread_join(pthread_t thread, void **status);`

Un `void pthread_exit(void *value_ptr);` est fait implicitement à la fin de start_routine.

La fonction `pthread_detach()` est comparable à la fonction `void pthread_exit(void *status);` à la différence que le thread est placé dans un mode "détaché".

Les thread non-détachés (défaut) passent à l'état zombie quand ils se terminent tant que pthread_join n'a pas récupéré leur status.

C'est pourquoi il est nécessaire d'attendre la fin d'un thread par : `int pthread_join(pthread_t thread, void **status`

– thread_id : numéro du thread ou NULL. Numéro du thread attendu ou le premier disponible.
– status : valeur passée par thread_exit.

7.3.6 Fonctions liées à la gestion des threads

Comme les threads sont des entités autonomes, il est nécessaire de pouvoir les identifier et les contrôler séparément.

La fonction, `pthread_t pthread_self(void);` renvoie le numéro du thread (équivalent getpid()).

La fonction `int pthread_equal(pthread_t t1, pthread_t t2);`, quant à elle, permet de vérifier si les threads `t1` et `t2` sont identiques.

La fonction `pthread_cancel()` permet de demander la suppression d'un thread qui l'aura permis au préalable via `pthread_setcancelstate()` (autorisé/interdit) et `pthread_setcanceltype()` (synchrone/retardé).

Il existe aussi des fonctions permettant de controler individuellement l'odonnancement des threads.

`pthread_getconcurrency()` et `pthread_setconcurrency()`, pour leur part, permettent de contrôler le nombre de thread tournant en parallèle en modifiant la valeur de "concurrence". Les fonction `pthread_getschedparam()` et `pthread_setschedparam()` permettent de régler plus précisement la statégie d'ordonnancement partage du temps CPU entre thread en spécifiant leur prorité individuelle.

7.3.7 Attributs des threads

```
pthread_attr_init() : Initializes an attribute set for use in the
   pthread_create() call.
pthread_attr_destroy() : Destroys the content of an attribute set.
pthread_attr_getdetachstate(), pthread_attr_getguardsize(),
pthread_attr_getinheritsched(), pthread_attr_getprocessor_np(),
pthread_attr_getschedparam(), pthread_attr_getschedpolicy(),
pthread_attr_getscope(), pthread_attr_getstackaddr(),
pthread_attr_getstacksize(), pthread_attr_setdetachstate(),
pthread_attr_setguardsize(), pthread_attr_setinheritsched(),
pthread_attr_setprocessor_np(), pthread_attr_setschedparam(),
pthread_attr_setschedpolicy(), pthread_attr_setscope(),
pthread_attr_setstackaddr(), pthread_attr_setstacksize()
```

7.3.8 Synchronisation de threads

Comme les threads sont autonomes, et comme ils partagent des zones de données communes, il est nécessaire de les synchroniser les uns sur les autres. Cette synchronisation peut se faire :
- par le biais de mutex : sortes de verrous manipulés par l'intermédiaire de variables (équivalents de sémaphores à un état).
- par l'utilisation de variables conditionnelles
- par des sémaphores
- par des mécanismes de type plusieurs lecteurs/un seul écrivain

Mutex

Les mutex sont des variables utilisables par plusieurs threads ou processus et utilisées pour verrouiller des variables en mémoire partagée (mutex).

La pose d'un mutex se fait par la commande :
- `int pthread_mutex_lock(pthread_mutex_t *mutex);` : pose un verrou ou se bloque si le verrou est déjà posé.

– `int pthread_mutex_unlock(pthread_mutex_t *mutex) ;` : ôte le verrou.
– `int pthread_mutex_unlock(pthread_mutex_t *mutex) ;` : essaye de poser le verrou, sort en erreur sinon.

– Avantage :
  – Demande peu de ressources et de temps

– Inconvénient :
  – le thread en attente peut attendre indéfiniment (starvation — famine)

Ex : accès concurrentiel aux champs d'une base de données. Le champ `idnumber` est protégé car plusieurs threads peuvent y accéder en même temps.

```
#include <pthread.h> <stdlib.h> <string.h>

typedef struct _employee
{
  pthread_mutex_t lock;
  char *firstname;
  char *lastname;
  struct _employee *manager;
  int idnumber;
} EMPLOYEE;

extern int IDnumber;
extern pthread_mutex_t IDlock;

EMPLOYEE * EmployeeNew (char *first, char *last)
{
  EMPLOYEE *ep;

  if (!(ep = (EMPLOYEE *) malloc (sizeof (*ep))))
    return (NULL);

  ep->firstname = strdup (first);
  ep->lastname = strdup (last);
  pthread_mutex_lock (&IDlock);
  ep->idnumber = IDnumber++;
  pthread_mutex_unlock (&IDlock);
  ep->manager = NULL;
  return (ep);
}
```

Variable condition

Les variables conditions sont utilisées pour bloquer un thread jusqu'à ce qu'une condition soit vérifiée. Un mutex est associé à cette condition. Il est pris par le thread qui trouve la condition vérifiée.

– `int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr) ;` : initialise la variable condition.
– `int pthread_cond_destroy(pthread_cond_t *cond) ;` : détruit la condition.
– `int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex) ;` : bloque le thread en attente de la condition. Prend le mutex si la condition est vérifiée.
– `int pthread_cond_signal(pthread_cond_t *cond) ;` : libère un thread bloqué en attente de la condition.
– `int pthread_cond_broadcast(pthread_cond_t *cond) ;` : libère tous les threads en attente sur la condition.
– `int pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex, const struct timespec *restrict abstime);` : identique à `pthread_cond_wait(cond, mutex)` mais pas au delà du moment spécifié par la variable abstime (time-out).

Ex : Un thread producteur reçoit des informations d'un socket et doit les envoyer à la queue d'une liste chaînée. Pendant ce temps, un processus consomme les données produites. On associe un mutex et une variable condition à la liste. Quand le thread producteur veut accéder à la liste, il demande le mutex associé, met l'information à la queue, signale la disponibilité de l'information et libère le mutex. A l'inverse, le thread consommateur demande le mutex, puis se met en attente de la condition

associée au mutex. Il libère alors le mutex et attend que la condition soit satisfaite (c'est la fonction `cond_wait()` qui permet de faire les 2 actions en même temps). Lorsqu'il est réveillé, il récupère le mutex, lit les informations, et libère finalement le mutex.

```
#include <pthread.h>
typedef struct _node
{
  struct _node *next;
} NODE;
static NODE list;
static pthread_mutex_t list_lk;
static pthread_cond_t list_cv;
static NODE end_message;

#define BUFSIZE 512

void * consumer (void *arg)
{
  for (;;)
    {
      NODE *elemp;
      pthread_mutex_lock (&list_lk);
      while (empty_list ())
        pthread_cond_wait (&list_cv, &list_lk);
      elemp = (NODE *)dequeue ();
      pthread_mutex_unlock (&list_lk);
      if (process (elemp) == 0)
        break;
    }
  return (0);
}

void * producer (void *arg)
{
  int fd = (int) arg;
  int n;
  char buf[BUFSIZE];

  /* Recoit un message d'un socket */
  while ((n = recv (fd, buf, BUFSIZE, 0)) != -1)
    {
      NODE *listp;
      if (n == 0)
        listp = &end_message;
      else if (!(listp = (NODE *)build_list (buf, n)))
        continue;
      pthread_mutex_lock (&list_lk);
      enqueue (listp);
      pthread_cond_signal (&list_cv);
      pthread_mutex_unlock (&list_lk);
      if (n == 0)
        break;
    }
  close (fd);
  return (0);
}
```

Sémaphore

Les sémaphores s'utilisent exactement comme pour les processus. Nous ne nous étendrons donc pas sur le sujet. Nous présentons uniquement les fonctions associées à la gestion des sémaphores.

- `int sem_init(sem_t *sem, int pshared, unsigned int valeur);`
- `int sem_wait(sem_t * sem);` : décrémente la valeur
- `int sem_trywait(sem_t * sem);` : retourne 0 si valeur>0 ou sort en erreur
- `int sem_post(sem_t * sem);` : incrémente la valeur
- `int sem_getvalue(sem_t * sem, int * sval);`
- `int sem_destroy(sem_t * sem);`

Multiple readers/one writer

Ce mécanisme, comme son nom l'indique, permet de gérer l'accès à une ressource ayant plusieurs lecteurs mais un seul écrivain.

Les fonctions associées sont les suivantes :
- `int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr);` : initialise la variable condition.

– `int pthread_rwlock_destroy(pthread_rwlock_t *rwlock) ;` : détruit la variable condition.
– `int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock) ;` : tente d'acquérir un read-write lock :
  – si rwlock est acquis en read, plusieurs threads peuvent l'acquérir.
  – si rwlock est acquis en write, le thread est bloqué.
– `int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock) ;` : même chose mais sort en erreur plutôt que de se bloquer.
– `int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock) ;` : acquiert un write lock si libre, se bloque si read ou write.
– `int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock) ;` : même chose mais sort en erreur plutôt que de se bloquer.
– `int pthread_rwlock_unlock(pthread_rwlock_t *rwlock) ;` : libère le verrou.

Ex : Dans l'exemple ci-dessous, des demandes sont faites pour trouver, ajouter, ou supprimer une connexion à un host. Bien sûr, plusieurs demandes sont possibles, s'il s'agit de trouver l'host `pthread_rwlock_rdlock()`, mais une seule demande d'ajout ou de suppression doit exister `pthread_rwlock_wrlock()`. Dans tous les cas, on libère le r/w lock à la fin.

```
#include <pthread.h>
#include <errno.h>

#define FIND 0x01
#define DELETE 0x02
#define ADD 0x03
#define MODIFY 0x04

#define MAXHOSTNAME 256

struct hostpair
{
  char name[MAXHOSTNAME];
  unsigned long ipaddr;
  pthread_mutex_t lk;
} he;

struct query
{
  int cmd;
  int status;
  struct hostpair he;
};

extern struct hostpair *lookup (const struct hostpair *);
extern int find (struct hostpair *);
extern int delete (struct hostpair *);
extern int add (struct hostpair *);
pthread_rwlock_t host_rw;

void HostOp (struct query *hop)
{
  struct hostpair *hp;

  switch (hop->cmd)
    {
    case FIND:
      pthread_rwlock_rdlock (&host_rw);
      hop->status = find (&hop->he);
      break;
    case DELETE:
      pthread_rwlock_wrlock (&host_rw);
      hop->status = delete (&hop->he);
      break;
    case ADD:
      pthread_rwlock_wrlock (&host_rw);
      hop->status = add (&hop->he);
      break;
    default:
      hop->status = EINVAL; /* Invalid argument */
      break;
    }

  if (hop->status != EINVAL)
    pthread_rwlock_unlock (&host_rw);
}
```

7.3.9 Les threads en mode noyau

Le concept de thread, jusqu'alors envisagé du coté utilisateur est aussi utilisable en mode noyau (il faut pour cela travailler dans cet espace, ce que nous pouvons faire lors de développement de modules noyau – voir chapitre 8).

L'interface de programmation est très simples :
- Un appel à la macro `struct task_struct *kthread_run(threadfn, data, namefmt, ...);` déclare le thread `namefmt`, l'associe à la fonction `threadfn`, lui passe l'argument `data` et le réveille. Elle renvoit l'identifiant du thread créé.
- A l'inverse, un appel à `int kthread_stop(struct task_struct *k);` permet d'interrompre le thread défini par l'identifiant `k`.

Il est intéressant de noter que dans un contexte multi-processeurs, un thread noyau `k` peut être associé à un processeur particulier `cpu`, via la fonction `void kthread_bind(struct task_struct *k, unsigned int cpu);`.

Ci-dessous un exemple de module développé par Pierre Ficheux pour illustrer le concept de `kthread`.

```
#include <linux/module.h>
#include <linux/delay.h>

static int kthreadA_id, kthreadB_id;
static DECLARE_COMPLETION(on_exit);

MODULE_DESCRIPTION("kthread1");
MODULE_AUTHOR("Pierre Ficheux, Open Wide");
MODULE_LICENSE("GPL");

static int kthread_func (void *data)
{
  int nrun = 0;
  char *s = (char*)data;

  daemonize(s);
  printk (KERN_INFO "%s starting\n", s);

  allow_signal (SIGTERM);

  while( 1 ) {
    ssleep (1);
    nrun++;
    printk (KERN_INFO "%s running %d\n", s, nrun);

    if (signal_pending (current) /* || nrun == 20*/)
      break;
  }

  printk (KERN_INFO "%s exiting\n", s);
  complete_and_exit (&on_exit, 0);
}

static int __init kthread1_init(void)
{
  if (!(kthreadA_id = kernel_thread (kthread_func, "kthreadA", CLONE_KERNEL)))
    return -EIO;

  if (!(kthreadB_id = kernel_thread (kthread_func, "kthreadB", CLONE_KERNEL))) {
    kill_proc (kthreadA_id, SIGTERM, 1);
    wait_for_completion (&on_exit);
    return -EIO;
  }

  return 0;
}

static void __exit kthread1_exit(void)
{
  kill_proc (kthreadA_id, SIGTERM, 1);
  kill_proc (kthreadB_id, SIGTERM, 1);
  wait_for_completion (&on_exit);
  wait_for_completion (&on_exit);
}

module_init (kthread1_init);
module_exit (kthread1_exit);
```

Quelques threads noyaux

kacpid S'occupe des évenements en relation avec la gestion énergétique (Advanced Configuration and Power Interface)

kswapd C'est le thread noyau chargé de la gestion du swap (voir section 11.1).

pdflush C'est le thread noyau en charge de la gestion du cache. Il vide les tampons sales sur le disque pour récupèree de la mémoire vive (voir section 11.2).

kblockd Exécute les fonctions de la queue `kblockd_workqueue` dont le but est d'activer pèriodiquement les pilotes de périphériques bloc

ksoftirqd Exécute les tasklets.

7.4 Quiz

1. Quelles sont les différences entre un processus et un thread ?
2. Qu'est-ce qu'un Light Weight Process ?
3. Est-il possible d'associer un thread à LWP ? Et à un processeur ?
4. Peut-on modifier l'attribution des LWP ?
5. Par rapport aux IPC, quels sont les nouveaux modes de communication entre threads ?
6. Donnez les commandes équivalentes à : fork(), exit(), getpid().
7. Quels sont les intérêts de la programmation multithreadée ?

7.5 Exercices

1. Créer n threads fonctionnant en parallèle sur un même calcul. Comparer avec une implantation avec des processus.
2. Dans l'exercice précédent, on se propose de faire effectuer le calcul en même temps en déclenchant le calcul par un signal. Même chose en utilisant une variable condition.

Chapitre 8

Les modules du noyau

Jusqu'à présent, bien que nous eussions utilisés les ressources systèmes du noyau, tous les programmes que nous avons présenté ont été programmés dans l'espace utilisateur. L'utilisation des fonctionnalités du noyau s'est effectuée grâce aux appels systèmes.

Dans ce chapitre, nous allons franchir la barrière des appels systèmes en programmant directement des fonctionnalités internes du noyau[1]. Évidemment, dans ce cadre, il faut prendre des précautions pour ne pas ruiner le fonctionnement du système d'exploitation : en mode noyau on peut tout faire... surtout des bêtises ! :)

Par ailleurs, il va falloir perdre un certain nombre d'habitudes. En particulier, pour développer, le programmeur s'appuie énormément sur la libC qui fait l'interface entre l'espace utilisateur et l'espace noyau. Mais quant est-il quand on est déjà dans l'espace noyau ?

8.1 Les équivalents noyau des fonctions usuelles

En programmation noyau, des choses aussi simples qu'afficher un message à l'écran ou réserver de la mémoire doivent être reconsidérées. En effet, point de `printf()` ni de `malloc()`. Les copies mémoires elles-mêmes (affectations entre variables, `memcpy()`, `strcpy()`), ne sont plus si faciles.

Il existe heureusement des équivalents à ces fonctions. Il faut cependant se poser des questions sur leur mode d'utilisation.

8.1.1 printk()

La fonction `printk(KERN_XXX "format", args...)` affiche un message sur la console du mode noyau. Normalement, le message est écrit sur la console physique de l'ordinateur, mais ce comportement peut être modifié (fonction register_console).

La chaîne générée peut débuter par un numéro qui définit la priorité du message. Les macros suivantes, spécifiant les priorités, sont définies dans l'en-tête ¡linux/kernel.h¿ :

KERN_EMERG Système inutilisable
KERN_ALERT L'administrateur doit réagir immédiatement
KERN_CRIT Situation critique
KERN_ERR Erreur du noyau

[1] Les exemples présentés ici sont plus ou moins fortement inspirés de [Ficheux, 2006a, Ficheux, 2006b, Ficheux, 2007, Ficheux, 2008, Ke-Fong, 2008]

KERN_WARNING Avertissement
KERN_NOTICE Informatif
KERN_INFO Informatif
KERN_DEBUG Message de déboguage

La différence majeure avec la fonction `printf()` est que les formats float et double ne sont pas supportés.

Un autre aspect essentiel pour l'utilisation en mode noyau est que printk est protégée des interruptions et peut donc être utilisée dans les gestionnaires d'interruptions et les portions de code critique, ce qui en fait un bon mode de debuggage (voire le seul selon Linus itself).

8.1.2 kmalloc() et kfree()

La fonction `void * kmalloc (size_t taille, int priority) ;` permet d'allouer une zone physiquement contiguë de mémoire dans l'espace noyau. Le paramètre taille est le nombre d'octets à allouer. Le paramètre priority précise l'importance et le type d'allocation souhaitée. Quelques unes des valeurs possibles sont GFP_KERNEL, GFP_DMA, GFP_ATOMIC, GFP_BUFFER, et GFP_NFS.

`kmalloc` ne peut allouer plus de 128 ko de mémoire. Par ailleurs, la taille allouée est en puissance de 2.

Pour allouer plus de mémoire, il est nécessaire de faire appel aux fonctions d'allocation en mémoire virtuelle.

Pour libérer la mémoire allouer par `kmalloc`, il faut faire appel à la fonction `void kfree (void * ptr) ;`.

8.1.3 copy_from_user et copy_to_user

Il faut faire très attention lorsque l'on manipule, depuis l'espace noyau, des données qui proviennent de l'espace utilisateur. En effet, ces données ne sont pas directement manipulables car à tout moment le process qui les détient peut être préempté. Le noyau met donc à disposition du développeur 2 fonctions qui permettent d'effectuer les copies de l'espace utilisateur à l'espace noyau (et vice versa) de manière sure.

- `unsigned long copy_from_user (void *to, const void user *from, unsigned long n) ;` : copie `n` octets de l'espace `from` utilisateur dans l'espace `to` noyau.
- `unsigned long copy_to_user (void user *to, const void *from, unsigned long n) ;` : à l'inverse cette fonction copie `n` octets de l'espace `from` noyau dans l'espace `to` utilisateur.

8.2 Compilation d'un module

De la même manière qu'il faut "oublier" certaines habitudes de programmation pour programmer un module noyau, il faut aussi prendre des "pincettes" pour compiler un tel module. Cependant, la compilation d'un module est relativement simple, surtout si le module est développé au sein d'un unique fichier source (ce qui peut aller à l'encontre de bonnes habitudes de programmation).

Nous donnons ci-dessous un squelette de Makefile permettant de compiler un module `mon_module_a_moi`.

```
obj-m := mon_mondule_a_moi.o
module-objs := mon_mondule_a_moi.o
KERNEL_SOURCE = /usr/src/linux-$(shell uname -r)

all:
    make -C $(KERNEL_SOURCE) M=$(PWD) modules
clean:
    make -C $(KERNEL_SOURCE) M=$(PWD) clean
install:
    make -C $(KERNEL_SOURCE) M=$(PWD) modules_install
$
```

L'entrée `all` permet à proprement parler de compiler le module. Ces modules, depuis la version 2.6 du noyau `mon_module_a_moi.ko` (ko pour Kernel Object).

L'entrée `clean` permet d'effacer les modules compilés.

L'entrée `install` permet d'installer le module dans les répertoires usuels où sont installés tous les modules (généralement /lib/modules/‘uname -r‘).

8.3 Chargement/Déchargement d'un module dans le noyau

Une fois le module compilé, il faut demander au noyau de charger se module afin de l'exécuter.

Il existe 2 moyens de le faire :
insmod nommodule.ko : permet d'installer le module sans vérifier les dépendances. Peut donc échouer si le module en nécessite.
modprobe nommodule : installe le module ainsi que toutes ses dépendances.
Pour retirer le module du noyau, c'est la fonction **rmmod monmodule** qu'il faut utiliser.

On signalera aussi l'existence de la commande `lsmod` qui permet de lister les modules actuellement installés sur le noyau et la commande `modinfo` qui permet d'obtenir des informations spécifiques sur un module :

```
$modinfo ext3
filename:       /lib/modules/2.6.22.18-desktop-1mdv/kernel/fs/ext3/ext3.
    ko.gz
license:        GPL
description:    Second Extended Filesystem with journaling extensions
author:         Remy Card, Stephen Tweedie, Andrew Morton, Andreas Dilger
    , Theodore Ts'o and others
depends:        jbd
vermagic:       2.6.22.18-desktop-1mdv SMP mod_unload 686
$
```

Il est possible de passer des paramètres lors du chargement d'un module. Toutefois ces paramètres sont réduis à 3 types seulement :
- types simples parmi (short, ushort, int, uint, long, ulong, charp – char pointer, bool – boolean, invbool – boolean inversé).
- chaînes de caractères
- tableaux de types simples

8.4 Programmation d'un module

8.4.1 Squelette minimal

Comme on l'a vu au paragraphe précédent, il y a 2 grandes étapes dans la vie d'un module dans l'espace noyau : son chargement et son déchargement. Le squelette d'un module noyau doit donc au minimum gérer ces 2 étapes :

```
#include <linux/module.h>
#include <linux/kernel.h> /* Pour KERN_INFO */

MODULE_DESCRIPTION("Hello World module");
MODULE_AUTHOR("Pierre Ficheux, Open Wide");
MODULE_LICENSE("GPL");

static int __init hello_init(void)
{
  printk(KERN_INFO "Hello World\n");
  return 0;
}

static void __exit hello_exit(void)
{
  printk(KERN_INFO "Goodbye, cruel world!\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

On trouve donc 2 fonctions, l'une traitant le chargement (`hello_init()`), l'autre le déchargement (`hello_exit()`). Il faut ensuite les déclarer au niveau du noyau (c'est le rôle des macros `module_init(hello_init)` et `module_exit(hello_exit))`.

Les macros utilisées au début du code source ne sont pas obligatoires mais peuvent aider au débuggage du module et à son utilisation par un tiers.

La ligne `MODULE_DESCRIPTION`, `MODULE_AUTHOR` et `MODULE_LICENSE` définissent les informations qui seront renvoyées par `modinfo` pour ce module.

NOTE : Théoriquement, il ne devrait pas y avoir d'autre licence possible sur le noyau que GPL [Wikipedia, 2008b]. Linus est très clair la dessus :

> ...
> You are not allowed to create and distribute a derived work unless it is GPLd.
> ...
> It is very clear : a kernel module is a derived work of the kernel by default. End of story.
> You can then try to prove (through development history etc) that there would be major reasons why it's not really derived. But your argument seems to be that nothing is derived, which is clearly totally false, as you yourself admit when you replace "kernel" with "Harry Potter".
> Linus (LKML - 04/12/2003)

Il existe cependant des modules propriétaires diffusés sous forme binaire et ne répondant pas à la licence GPL. Malheureusement, ils sont compilés pour UNE version du noyau particulière, ce qui est

très contraignant pour la portabilité.

Il est donc intéressant pour quiconque veut pouvoir utiliser Linux sans être trop contraint par la licence (par exemple, pour vendre son logiciel) de programmer un minimum de chose en mode noyau et de reporter le reste en espace utilisateur, pour lequel existe une licence plus modérée (LGPL - [Wikipedia, 2008c]).

Si vous êtes intéressés par ces notions de licences, je vous recommande la lecture de l'étude réalisée par IDC sur l'initiative du Secrétariat d'Etat à l'industrie [sur l'initiative du Secrétariat d'Etat à l'industrie, 2008].

8.4.2 Passage de paramètres à un module

La programmation du passage de paramètres au noyau est réalisable en utilisant plusieurs macros dédiées à cet effet.

- module_param(nom_var,type, droits)
- module_param_array(nom_var, type, addr, droits) (addr est l'adresse d'une variable qui contient le nombre d'éléments initialisés.)
- module_param_string(nom_apparant_dans_modinfo, nom_var, taille_de_la_chaine, droits)

Les droits est la valeur définissant les droits d'accès au fichier virtuel correspondant à cette variable et défini dans `/sys/module/nom_module/parameters/nom_parametre` (pseudo-système de fichier `sysfs` – [Wikipedia, 2008e]).

Une description du paramètre, qui sera affichée par `modinfo`, peut être fournie via la macro `MODULE_PARM_DESC(var,"Description")`.

Voici un exemple du même auteur :

```
#include <linux/module.h>
#include <linux/kernel.h> /* Pour KERN_INFO */

MODULE_DESCRIPTION("Hello world module");
MODULE_AUTHOR("Pierre Ficheux, Open Wide");
MODULE_LICENSE("GPL");

#define MAX_STRING      10
#define MAX_TAB         16

/* Paramètres: entier, chaîne, tableau */
static int entier;
static char chaine[MAX_STRING];
static int tableau[MAX_TAB];

module_param(entier, int, 0644);
module_param_array(tableau, int, NULL, 0644);
module_param_string(chaine, chaine, sizeof(chaine), 0644);

MODULE_PARM_DESC(entier,"Un entier");
MODULE_PARM_DESC(chaine,"Une chaîne de caractères");
MODULE_PARM_DESC(tableau,"Un tableau d'entiers séparés par des virgules");

static int __init hello_init(void)
{
  register int i;

  printk(KERN_INFO "entier= %d chaine= %s\n", entier, chaine);

  for (i = 0  ; i < MAX_STRING ; i++)
    printk(KERN_INFO "tableau[%d] = %d\n", i,  tableau[i]);

  return 0;
}

static void __exit hello_exit(void)
{
  printk(KERN_INFO "Goodbye, cruel world!\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

8.4.3 Communiquer avec le module

Via procfs

`procfs` (process file system) est un pseudo-système de fichiers monté sur `/proc` (pseudo car dynamiquement généré au démarrage) utilisé pour accéder aux informations du noyau en cours d'exécution. Puisque `/proc` n'est pas une arborescence réelle, il ne consomme pas d'espace disque mais de la mémoire vive. `procfs` est disponible sur Linux mais aussi Solaris, BSD, IBM AIX et QNX.

Un module noyau peut créer des fichiers et des répertoires dans `procfs` pour récupérer ou envoyer des informations à l'utilisateur.

- `struct proc_dir_entry* create_proc_entry(const char* name, mode_t mode, struct proc_dir_entry* parent) ;` : création d'un fichier régulier dans `/proc`. Cette fonction renvoi une structure de type `proc_dir_entry` qui peut être modifiée pour spécifier les fonctions de traitement des lecture/écriture dans le fichier.
- `struct proc_dir_entry* proc_mkdir(const char* name, struct proc_dir_entry* parent) ;` : création d'un répertoire dans `/proc`.
- `void remove_proc_entry(const char* name, struct proc_dir_entry* parent) ;` : suppression d'un fichier ou répertoire.

La structure `proc_dir_entry` spécifie 2 pointeurs vers des fonctions de traitement en lecture/écriture de/vers le fichier `/proc` correspondant :

- `entry->read_proc = read_proc_function;` : la fonction `read_proc_function` est de type `int read_func(char* page, char** start, off_t off, int count, int* eof, void* data) ;` .
- `entry->write_proc = write_proc_function;` : la fonction `write_proc_function` est de type `int write_func(struct file* file, const char* buffer, unsigned long count, void* data) ;`

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>

MODULE_DESCRIPTION("Hello procfs ... le module");
MODULE_AUTHOR("Arnaud revel");
MODULE_LICENSE("GPL");

char message[100];

/* Appelée sur accès à /proc/hello */
static int read_func(char* page, char** start, off_t off, int count, int* eof, void* data)
{
  int len = 0;
  len += sprintf(page + len, "%s\n", message);
  return len;
}
static int write_func(struct file* file, const char* buffer, unsigned long count, void* data)
{
  int len=count;

  if(copy_from_user(message, buffer, count)) {
    return -EFAULT;
  }
  message[count]=0;

  printk(KERN_INFO "Modification message : %s\n",message);

  return len;
}

static int __init hello_init(void)
{
  struct proc_dir_entry *proc_entry;

  printk(KERN_INFO "Debut procfs\n");

  sprintf(message, "%s\n", "coucou");

  /* Entrée /proc */
  proc_entry = create_proc_entry("hello", 0, NULL);
  if (proc_entry) {
    proc_entry->read_proc = read_func;
    proc_entry->write_proc = write_func;
  } else {
    printk(KERN_ERR "Can't create proc entry for square\n");
    return -EAGAIN;
  }

  return 0;
}

static void __exit hello_exit(void)
{
  printk(KERN_INFO "Fin procfs\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Via une entrée périphérique

La manière la plus commune cependant d'interagir avec les modules est de lui associer un périphérique, dans la grande tradition Unixienne selon laquelle "tout est fichier". La gestion des fichiers s'effectuant via la table des inodes, c'est ce mécanisme même qui est utilisé aussi pour les périphériques.

En mode super-utilisateur, il est alors nécessaire d'appeler la commande suivante :

```
# mknod /dev/nom_du_peripherique c majeure mineure
```

`mknod c` définit un périphérique en mode caractères (c'est-à-dire à accès direct) et `mknod b` un périphérique à accès bufferisé (tels que les disques).

La majeure correspond à un numéro définissant une classe de périphériques (disques durs, usb, pci...) et donc le module de gestion qui lui est associé (appelé le pilote de périphérique). La mineure, quant à elle, représente une instance particulière pour ce type de composant (les constructeurs peuvent être différents – par contre, le pilote est le même). Ces numéros sont spécifiquement réservés et dédiés par le noyau (voir `/usr/src/linux/Documentation/devices.txt` dans le code source du noyau pour plus d'informations). Il est à noter cependant que les numéros **60 à 63, 120 à 127 et 240 à 254**

sont réservés pour une utilisation locale ou expérimentale. Si la majeure est à 0, le noyau alloue un numéro de majeure dynamiquement.

Le listing ci-dessous correspond à un extrait du fichier `devices.txt` pour les majeures 3 et 4.

```
  3 block  First MFM, RLL and IDE hard disk/CD-ROM interface
      0 = /dev/hda     Master: whole disk (or CD-ROM)
     64 = /dev/hdb     Slave: whole disk (or CD-ROM)

    For partitions, add to the whole disk device number:
      0 = /dev/hd?     Whole disk
      1 = /dev/hd?1    First partition
      2 = /dev/hd?2    Second partition
        ...
     63 = /dev/hd?63   63rd partition

    For Linux/i386, partitions 1-4 are the primary
    partitions, and 5 and above are logical partitions.
    Other versions of Linux use partitioning schemes
    appropriate to their respective architectures.

  4 char  TTY devices
      0 = /dev/tty0    Current virtual console
      1 = /dev/tty1    First virtual console
        ...
     63 = /dev/tty63   63rd virtual console
     64 = /dev/ttyS0   First UART serial port
        ...
    255 = /dev/ttyS191  192nd UART serial port

    UART serial ports refer to 8250/16450/16550 series devices.
```

ATTENTION

Si vous utilisez un numéro de majeur déjà réservé, il y a un risque évident de désorganisation du noyau !

Programmer un pilote de périphérique consiste principalement à définir l'implémentation des opérations de fichier associées à ce périphérique. La liste totale des opérations possibles est définie par la structure `file_operations` :

```
struct file_operations {
  struct module *owner;
  loff_t (*llseek) (struct file *, loff_t , int);
  ssize_t (*read) (struct file *, char __user *, size_t , loff_t *);
  ssize_t (*write) (struct file *, const char __user *, size_t , loff_t *);
  ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long,
     loff_t);
  ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long,
     loff_t);
  int (*readdir) (struct file *, void *, filldir_t);
  unsigned int (*poll) (struct file *, struct poll_table_struct *);
  int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
  long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
  long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
  int (*mmap) (struct file *, struct vm_area_struct *);
  int (*open) (struct inode *, struct file *);
  int (*flush) (struct file *, fl_owner_t id);
  int (*release) (struct inode *, struct file *);
  int (*fsync) (struct file *, struct dentry *, int datasync);
  int (*aio_fsync) (struct kiocb *, int datasync);
  int (*fasync) (int, struct file *, int);
  int (*lock) (struct file *, int, struct file_lock *);
  ssize_t (*sendfile) (struct file *, loff_t *, size_t , read_actor_t , void *);
  ssize_t (*sendpage) (struct file *, struct page *, int, size_t , loff_t *, int)
     ;
  unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long
     , unsigned long, unsigned long);
  int (*check_flags)(int);
  int (*dir_notify)(struct file *filp , unsigned long arg);
  int (*flock) (struct file *, int, struct file_lock *);
  ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *,
     size_t , unsigned int);
  ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
     size_t , unsigned int);
};
```

Cependant, il n'est évidemment pas obligatoire de définir toutes les opérations. Les opérations les plus courantes à implémenter sont :

- `open` : pour initialiser les ressources liées au périphériques.
- `release` : pour libérer ces mêmes ressources.
- `read/write` : permet d'échanger des données avec le périphérique.

Il s'agit alors de mettre à jour la structure de la manière suivante :

```
static struct file_operations mon_driver_fops = {
  .owner =  THIS_MODULE,
  .read =    mon_driver_read ,
  .write =   mon_driver_write ,
  .open =    mon_driver_open ,
  .release =  mon_driver_release ,
};
```

La commande `register_chrdev(major, "mon_driver", &mon_driver_fops);` permet d'enregistrer le driver tandis que `unregister_chrdev(major, "mon_driver")` permet de libérer le driver.

Ci-joint un exemple développé par Stelian Pop et Pierre Ficheux :

```
/*
 * Includes
 */
#include <linux/kernel.h> /* printk() */
#include <linux/module.h> /* modules */
#include <linux/fs.h>             /* file_operations */

MODULE_DESCRIPTION("mydriver1");
MODULE_AUTHOR("Stelian Pop/Pierre Ficheux, Open Wide");
MODULE_LICENSE("GPL");

/*
 * Arguments
 */
static int major = 0; /* Major number */

module_param(major, int, 0644);
MODULE_PARM_DESC(major, "Static major number (none = dynamic)");

/*
 * File operations
 */
static ssize_t mydriver1_read(struct file *file, char *buf, size_t count, loff_t *ppos)
{
  printk(KERN_INFO "mydriver1: read()\n");

  return count;
}

static ssize_t mydriver1_write(struct file *file, const char *buf, size_t count, loff_t *ppos)
{
  printk(KERN_INFO "mydriver1: write()\n");

  return count;
}

static int mydriver1_open(struct inode *inode, struct file *file)
{
  printk(KERN_INFO "mydriver1: open()\n");

  return 0;
}

static int mydriver1_release(struct inode *inode, struct file *file)
{
  printk(KERN_INFO "mydriver1: release()\n");

  return 0;
}

static struct file_operations mydriver1_fops = {
  .owner =  THIS_MODULE,
  .read =   mydriver1_read,
  .write =  mydriver1_write,
  .open =   mydriver1_open,
  .release =  mydriver1_release,
};

/*
 * Init and Exit
 */
static int __init mydriver1_init(void)
{
  int ret;

  ret = register_chrdev(major, "mydriver1", &mydriver1_fops);
  if (ret < 0) {
    printk(KERN_WARNING "mydriver1: unable to get a major\n");

    return ret;
  }

  if (major == 0)
    major = ret; /* dynamic value */

  printk(KERN_INFO "mydriver1: successfully loaded with major %d\n", major);

  return 0;
}

static void __exit mydriver1_exit(void)
{
  if (unregister_chrdev(major, "mydriver1") < 0) {
    printk(KERN_WARNING "mydriver1: error while unregistering\n");
    return;
  }

  printk(KERN_INFO "mydriver1: successfully unloaded\n");
}

/*
 * Module entry points
 */
module_init(mydriver1_init);
module_exit(mydriver1_exit);
```

8.4.4 Outils pour la programmation de pilotes de périphériques

Gestion des ports d'entrées/sorties (d'après [Wikipedia, 2008a])

Jusqu'à présent, nous avons envisagé l'écriture d'un module noyau en faisant une totale abstraction du matériel. Pourtant, le coeur de la programmation en mode noyau consiste à écrire ou adapter des pilotes de périphériques existant.

Dans cette partie, nous allons voir quelques outils logiciels mis à la disposition du développeur et permettant de simplifier l'accès aux périphériques.

Les périphériques d'un ordinateur sont reliés au reste du système par des ports d'entrées et/ou de sortie.

Un port d'entrée est essentiellement composé de tampons trois états qui font apparaître, au moment voulu, les niveaux logiques du périphérique d'entrée (choisi par le bus d'adresse) sur le bus de données.

Un port de sortie est essentiellement composé de bascules logiques reliées au bus de données dans lesquelles le processeur vient écrire. Les sorties des bascules contrôlent les périphériques, généralement via un étage de puissance.

On distingue principalement trois façons de gérer les entrées/sorties :

Entrées/sorties programmées Pendant l'exécution de son programme principal, le microprocesseur va périodiquement lire l'état des périphériques d'entrée et modifie, si nécessaire, l'état des ports de sortie. C'est la technique la plus simple.

Interruptions Cette technique est utilisée lorsque le processeur doit réagir rapidement à un changement d'état d'un port d'entrée. Le périphérique prévient le processeur par une ligne d'interruption prévue à cet effet. Le processeur interrompt la tâche en cours, saute dans le sous-programme destiné à gérer la demande spécifique qui lui est adressée ; à la fin du sous-programme, le processeur reprend l'exécution du programme principal là où il l'avait laissée.

Accès direct à la mémoire Cette technique, connue souvent par ses initiales DMA (Dynamic Memory access), est utilisée lorsque l'on doit procéder à un transfert rapide d'un grand nombre de données entre, par exemple, un lecteur de CD et un disque dur. Plutôt que de transférer les octets d'abord vers un registre du processeur, puis seulement vers le disque dur, les octets sont transférés directement d'un périphérique à l'autre sans passer par les registres du processeur. Le transfert des données est organisé par le contrôleur d'interruptions, qui prend la place du processeur pendant le transfert et gère les bus d'adresses et de contrôle.

Ports d'entrée/sortie

Un processeur définit un certain nombre d'adresses de ports d'entrées/sorties qui permettent d'échanger des informations avec ces périphériques via des commandes bas-niveau :

- `int inb(int read_addr) ;` : récupère un octet depuis le port définit par l'adresse `read_addr`
- `void outb(int write_addr, char value) ;` : envoie l'octect `value` sur le port définit par l'adresse `write_addr`.

La liste des ports disponibles sur une machine est accessible via la commande :

```
$ cat /proc/ioports
0000-001f : dma1
0020-0021 : pic1
0040-0043 : timer0
0050-0053 : timer1
0060-006f : keyboard
0080-008f : dma page reg
00a0-00a1 : pic2
00c0-00df : dma2
00f0-00ff : fpu
0170-0177 : 0000:00:1f.2
  0170-0177 : libata
01f0-01f7 : 0000:00:1f.2
  01f0-01f7 : libata
0376-0376 : 0000:00:1f.2
  0376-0376 : libata
03c0-03df : vesafb
03f6-03f6 : 0000:00:1f.2
  03f6-03f6 : libata
03f8-03ff : serial
04d0-04d1 : pnp 00:02
0900-090f : pnp 00:08
0910-091f : pnp 00:08
0920-092f : pnp 00:08
0930-093b : pnp 00:0c
093c-093f : pnp 00:08
0940-097f : pnp 00:08
0cf8-0cff : PCI conf1
1000-107f : 0000:00:1f.0
  1000-1005 : pnp 00:02
    1000-1003 : ACPI PM1a_EVT_BLK
    1004-1005 : ACPI PM1a_CNT_BLK
  1006-1007 : pnp 00:03
  1008-100f : pnp 00:02
    1008-100b : ACPI PM_TMR
  1020-1020 : ACPI PM2_CNT_BLK
  1028-102f : ACPI GPE0_BLK
  1060-107f : pnp 00:03
    1060-107f : iTCO_wdt
1080-10bf : 0000:00:1f.0
  1080-10bf : pnp 00:03
10c0-10df : pnp 00:03
10e0-10e5 : ACPI CPU throttle
2000-2fff : PCI Bus #03
  2000-20ff : PCI CardBus #04
  2400-24ff : PCI CardBus #04
bf20-bf3f : 0000:00:1d.3
  bf20-bf3f : uhci_hcd
bf40-bf5f : 0000:00:1d.2
  bf40-bf5f : uhci_hcd
bf60-bf7f : 0000:00:1d.1
  bf60-bf7f : uhci_hcd
bf80-bf9f : 0000:00:1d.0
  bf80-bf9f : uhci_hcd
bfa0-bfaf : 0000:00:1f.2
  bfa0-bfaf : libata
d000-dfff : PCI Bus #01
  de00-deff : 0000:01:00.0
ec40-ec7f : 0000:00:1e.2
  ec40-ec7f : Intel ICH6
ed00-edff : 0000:00:1e.2
  ed00-edff : Intel ICH6
f400-f4fe : pnp 00:03
$
```

Le port série se trouve ainsi par exemple à l'adresse 0x3f8.

Gestion des interruptions

Lorsqu'un périphérique désire l'attention du microprocesseur parce qu'il a des informations à lui communiquer, il lève des interruptions qui seront ou non prises en comptes (voir la section 2.4.2 pour une présentation générale sur la gestion des interruptions par le système d'exploitation).

La fonction `int request_irq(unsigned int irq, void (*gest)(int, void *, struct pt_regs *), unsigned long drap_interrupt, const char *devname, void *dev_id);` permet d'associer un numéro d'interruption (`irq`) à une fonction `gest` qui traite cette interruption pour le compte du driver du périphérique `devname` repéré par son identifiant `dev_id` (seulement utile dans le cas où il y a partage de l'interruption – sinon, on répète la fonction d'interruption).

ATTENTION

Même si la mise en place d'une fonction de gestion d'interruption ressemble beaucoup à celle d'un traitement de signal, les mécanismes en jeu sont très différents !
En effet, un signal, même s'il peut avoir une origine matérielle, est un évènement "logiciel" généré par le noyau. Il n'y a en particulier aucune nécessité de traitement rapide ni de priorité entre les signaux. Il est délivré au processus ciblé uniquement lorsque celui-ci est ordonnancé et pas avant.
Le traitement d'une interruption, par contre, préempte tout processus utilisateur au moment où il est reçu.

ATTENTION 2

Bien que l'arrivée d'une interruption force le passage en mode noyau vers la routine de traitement de l'interruption en question, le noyau Linux n'est pas un noyau temps réel. En effet, un processus en mode noyau n'est pas préemptible (ou du moins pas tout le temps dans les dernières versions du noyau) par un autre processus ce qui en fait un système non-déterministe (nous reviendrons sur les concepts Temps Réel au chapitre 10).

Le drapeau `drap_interrupt` sert à spécifier le type de gestionnaire utilisé :
SA_INTERRUPT indique que le gestionnaire d'interruptions est un gestionnaire rapide (dans ce cas, les interruptions sont masquées durant le traitement de l'interruption).
SA_SHIRQ indique que le gestionnaire supporte le partage de l'interruption avec d'autres gestionnaires.
– Il existe d'autres valeurs du drapeau mais elles n'ont pas d'utilité ici.
La fonction de gestion de l'interruption doit retourner `IRQ_HANDLED` si l'interruption a été traitée correctement. Sinon, elle doit retourner `IRQ_NONE`.

La libération d'une interruption s'effectue via l'appel à `void free_irq(unsigned int irq, void *dev_id) ;`.

Création de Timer

L'API noyau fournit la possibilité de créer des timer logiciels programmé grâce à la fonction `void fastcall init_timer(struct timer_list * timer) ;`. Où `timer_list` est une structure définissant en particulier un champ `function` qui donne la fonction appelée lorsque le temps définit par le timer est dépassé et un champ `data` qui définit un paramètre à passer à la fonction sous la forme d'un entier long. L'activation effective du timer se fait grâce à la fonction `int mod_timer(struct timer_list * timer, unsigned long expires) ;` qui donne le moment en nombre de ticks **depuis le démarrage du système** où doit être déclenché le timer. Ainsi, si l'on veut déclencher le timer dans 100 ticks, `expires` vaut `jiffies+100`. Par aiileurs, il faut réarmer le timer dans la fonction de gestion si l'on veut un timer pèriodique.

L'exemple ci-dessous montre la programmation d'un timer type :

```
#include <linux/timer.h>
#define INTERVALLE 100

static struct timer_list timer;

static void montimer(unsigned long data)
{
    ...
    /* Il faut réarmer le timer si l'on veut un appel pèriodique */
    mod_timer(&timer, jiffies+100);
    ...
}

static int __init module_init(void)
{
    ...
    init_timer(&timer);
    timer.function = montimer;
    timer.data = 0;
    mod_timer(&timer, jiffies + INTERVALLE);
}
```

Traitement top-half/bottom-half

Lorsqu'il nécessaire de répondre rapidement à une interruption mais que le traitement de l'interruption peut être long, le traitement est en fait découpé en 2 parties :
- La partie top-half : acquite rapidement cette interruption au niveau matériel et introduit le traitement à effectuer au niveau d'une tasklet (faible latence) ou dans une file de traitement (workqueue).
- La partie bottom-half : c'est la gestion différée (asynchrone) de l'interruption.

Gestion des tasklets

Les tasklets sont des appels qui sont effectués uniquement dans un contexte d'interruption. Créer une tasklet revient à faire une demande au noyau pour exécuter une tâche atomique de manière différée, en fonction de sa disponibilité (mais jamais au dela d'un tick).

Pratiquement, une tasklet est représentée par une structure très semblable à la structure timer (elle contient un champ fonction `func` et un champ donné `data`), qu'il faut initialiser.

La macro `DECLARE_TASKLET(name, func, data);`, permet de déclarer la tasklet `name`, de l'associer à la fonction `func` en lui passant les données `data`.

La demande d'exécution de la tasklet à proprement parler s'effectue via la fonction `tasklet_schedule(&name);`. Il existe aussi une demande d'exécution haute priorité (tasklet_hi_schedule(&name);) à n'utiliser que dans le cadre de pilote faible latence tels que les buffer audio.

Une tasklet peut être inihibée `DECLARE_TASKLET_DISABLED()`, `tasklet_disable()`, `tasklet_disable_nosync()` réactivée par `tasklet_enable()` ou supprimée `tasklet_kill()`.

```
#include <linux/interrupt.h>

void tasklet_function(unsigned long);

char tasklet_data[64];

DECLARE_TASKLET(test_tasklet, tasklet_function, (unsigned long) &
    tasklet_data);

void tasklet_function(unsigned long data)
{
  struct timeval now;
  do_gettimeofday(&now);
  printk("%s at %ld,%ld\n", (char *) data, now.tv_sec, now.tv_usec);
}

int init_module(void) {
  sprintf(tasklet_data,"%s\n", "Linux tasklet called in init_module");
  tasklet_schedule(&test_tasklet);
}
```

Gestion des files de travail

Les files de travail (workqueues) sont, comme les tasklets, des moyens de faire exécuter des tâches au noyau. Il y a cependant des différences de taille :
- Les tasklets sont des entités exécutées de manière atomique dans le temps d'une interruption alors que les workqueues fonctionnent dans le contexte d'un processus noyau ce qui leur assure plus de souplesse quant à la possibilité d'être interrompues/reprises.
- Les workqueues peuvent s'exécuter de manière différée.

Comme pour les tasklets, les workqueues se déclarent en associant une fonction à exécuter à un nom de workqueue, grâce à la macro `DECLARE_WORK(nom,fonction,donnees)`. La création de la file à proprement parler se fait par `create_workqueue(name)` qui renvoit un pointeur sur une structure `struct workqueue_struct`.

La soumission de la tâche à la file peut ensuite s'éffectuer de 2 manières :
- via l'appel à `int queue_work(struct workqueue_struct *wq, struct work_struct *work)` : la tâche est mise dans la file immédiatement.
- via `int queue_delayed_work(struct workqueue_struct *wq, struct work_struct *work, unsigned long delay)` : la tâche est mise dans la file mais ne s'exécutera pas avant que `delay` ticks ne soient passés.

Il est possible par la suite de supprimer une tâche (`int cancel_delayed_work(struct work_struct *work);`, de vider une file de travail (`void flush_workqueue(struct workqueue_struct *queue)` ou de détruire une file (`void destroy_workqueue(struct workqueue_struct *queue)`).

Ci-dessous un exemple tiré du *Linux Kernel Module Programming Guide* ([Salzman, 2008]). Ce module effectue l'appel d'une tâche de comptage à intervalles réguliers :

```
 *  sched.c - schedule a function to be called on every timer interrupt.
 *
 *  Copyright (C) 2001 by Peter Jay Salzman
 */

/* The necessary header files */
/* Standard in kernel modules */
#include <linux/kernel.h> /* We're doing kernel work */
#include <linux/module.h> /* Specifically, a module */
#include <linux/proc_fs.h>  /* Necessary because we use the proc fs */
#include <linux/workqueue.h>  /* We scheduale tasks here */
#include <linux/sched.h>  /* We need to put ourselves to sleep and wake up later */
#include <linux/init.h>   /* For __init and __exit */
#include <linux/interrupt.h>  /* For irqreturn_t */

struct proc_dir_entry *Our_Proc_File;
#define PROC_ENTRY_FILENAME "sched"
#define MY_WORK_QUEUE_NAME "WQsched.c"

/* The number of times the timer interrupt has been called so far */
static int TimerIntrpt = 0;
static void intrpt_routine(void *);
static int die = 0;    /* set this to 1 for shutdown */

/* The work queue structure for this task, from workqueue.h */
static struct workqueue_struct *my_workqueue;
static struct work_struct Task;
static DECLARE_WORK(Task, intrpt_routine, NULL);

/*
 * This function will be called on every timer interrupt. Notice the void*
 * pointer - task functions can be used for more than one purpose, each time
 * getting a different parameter.
 */
static void intrpt_routine(void *irrelevant)
{
  /* Increment the counter */
  TimerIntrpt++;

  /*
   * If cleanup wants us to die
   */
  if (die == 0)     queue_delayed_work(my_workqueue, &Task, 100);
}
/*
 * Put data into the proc fs file.
 */
ssize_t procfile_read(char *buffer, char **buffer_location, off_t offset,
                      int buffer_length, int *eof, void *data)
{
  int len;     /* The number of bytes actually used */

  /* It's static so it will still be in memory  when we leave this function */
  static char my_buffer[80];

  /*
   * We give all of our information in one go, so if anybody asks us
   * if we have more information the answer should always be no.
   */
  if (offset > 0) return 0;

  /* Fill the buffer and get its length */
  len = sprintf(my_buffer, "Timer called %d times so far\n", TimerIntrpt);

  /* Tell the function which called us where the buffer is */
  *buffer_location = my_buffer;

  /* Return the length */
  return len;
}
/* Initialize the module - register the proc file */
int __init init_module()
{
  /* Create our /proc file */
  Our_Proc_File = create_proc_entry(PROC_ENTRY_FILENAME, 0644, NULL);

  if (Our_Proc_File == NULL) {
    remove_proc_entry(PROC_ENTRY_FILENAME, &proc_root);
    printk(KERN_ALERT "Error: Could not initialize /proc/%s\n",
           PROC_ENTRY_FILENAME);
    return -ENOMEM;
  }

  Our_Proc_File->read_proc = procfile_read;
  Our_Proc_File->owner = THIS_MODULE;
  Our_Proc_File->mode = S_IFREG | S_IRUGO;
  Our_Proc_File->uid = 0;
  Our_Proc_File->gid = 0;
  Our_Proc_File->size = 80;
```

```
    /*
     * Put the task in the work_timer task queue, so it will be executed at
     * next timer interrupt
     */
    my_workqueue = create_workqueue(MY_WORK_QUEUE_NAME);
    queue_delayed_work(my_workqueue, &Task, 100);


    printk(KERN_INFO "/proc/%s_created\n", PROC_ENTRY_FILENAME);

    return 0;
}
/* Cleanup */
void __exit cleanup_module()
{
    /* Unregister our /proc file */
    remove_proc_entry(PROC_ENTRY_FILENAME, &proc_root);
    printk(KERN_INFO "/proc/%s_removed\n", PROC_ENTRY_FILENAME);

    die = 1;    /* keep intrp_routine from queueing itself */
    cancel_delayed_work(&Task); /* no "new ones" */
    flush_workqueue(my_workqueue);  /* wait till all "old ones" finished */
    destroy_workqueue(my_workqueue);

    /*
     * Sleep until intrpt_routine is called one last time. This is
     * necessary, because otherwise we'll deallocate the memory holding
     * intrpt_routine and Task while work_timer still references them.
     * Notice that here we don't allow signals to interrupt us.
     *
     * Since WaitQ is now not NULL, this automatically tells the interrupt
     * routine it's time to die.
     */
}
/*
 * some work_queue related functions
 * are just available to GPL licensed Modules
 */
MODULE_LICENSE("GPL");
```

Gestion des accès concurrents

Les accès concurrents aux ressources sont des situations naturellement courantes dans le cas de développements de périphériques. En effet, le périphérique matériel constitue justement la ressource partagée entre les différents processus du système d'exploitation. Il est cependant primordial de bien gérer les concurrences sur ces ressources.

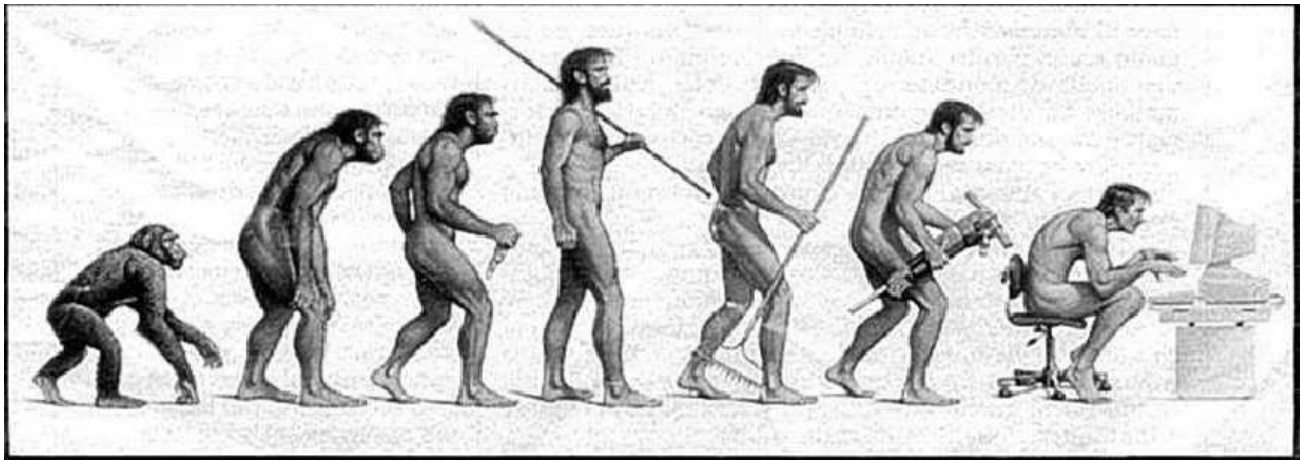
La déclaration d'un sémaphore se fait via la macro `DECLARE_MUTEX(sem)`, l'acquisition via `mutex_lock(&sem)` et la libération par `mutex_unlock(&sem)`.

A NOTER

Il est à noter qu'une ancienne interface de programmation est toujours utilisée. Elle utilisait `down(&sem)` et `up(&sem)` pour l'acquisition/libération.

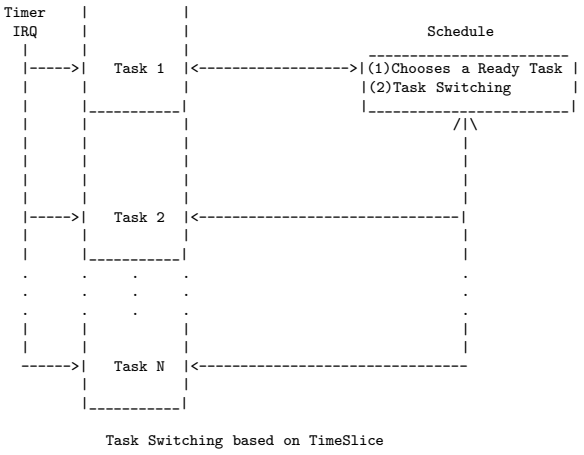
Chapitre 9

L'ordonnancement



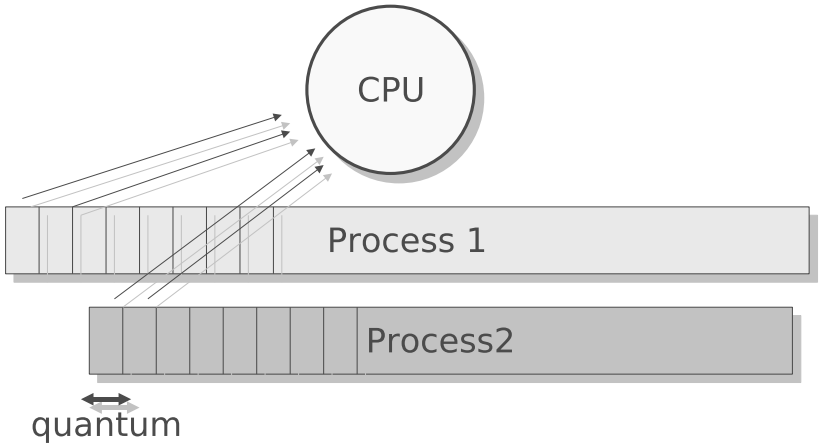
9.1 Principe du temps partagé

Un système d'exploitation est dit "multi-tâche", s'il permet à plusieurs tâches de fonctionner avec un semblant de "parallèlisme" sur une même machine. En effet, la plupart des ordinateurs ne possédant qu'un unique processeur, le parallélisme n'est que simulé. Même dans le contexte d'ordinateurs multi-processeur, le degré de parallélisme logique (nombre de tâches tournant en parallèle) dépasse en général largement le parallélisme réel de la machine (nombre de processeurs – sur un PC courant de 1 à 4).



Le parallélisme est simulé en utilisant le principe du **temps partagé** . L'idée est de partager le temps en tranches (appelés **quanta**) qui sont affectées (de manière plus ou moins parcimonieuse

selon les stratégie d'ordonnancement utilisées) aux différents processus tournant sur le système. Si les tranches sont de petite taille et que le temps de retour à une tâche donnée est court, l'utilisateur a l'impression que les tâches tournent en même temps. Concernant le passage d'une tâche à une autre, appelé, **commutation entre tâches** , celles-ci peuvent être initiées par les programmes eux-mêmes (multitâche coopératif – les tâches ont l'initiative de la commutation) ou par le système d'exploitation lors d'événements externes (multitâche préemptif – le noyau interrompt une tâche pour passer la main à une autre). Le système d'exploitation Linux fonctionne selon le principe du temps partagé et utilise les mécanismes de préemption. Toutefois, le noyau lui-même ne pouvant être préempté (voir à ce titre le chapitre 10, Linux n'est pas un système temps réel (même si le noyau 2.6 a fait des efforts dans ce sens).



Pour implémenter le temps partagé, il est nécessaire que le système d'exploitation puisse interrompre un processus à intervales réguliers (un quantum) pour donner la main à un autre selon une politique d'équité. Toutefois, on peut considérer que certains processus sont plus prioritaires que d'autres et il est alors nécessaire de biaiser les calculs de répartitions pour tenir compte de ce paramètre. Chaque processus dispose donc d'un paramètre de **priorité** qui définit son importance.

Dans la suite de ce chapitre, nous allons présenter certains des algorithmes d'ordonnancement qui ont été ou sont encore utilisés par Linux. Avant cela, nous devons introduire la notion d'état d'une tâche et la notion de commutation de tâche.

9.2 Les différents états d'une tâche

Au cours de sa "vie", un processus va rencontrer différentes situations qui le feront passer dans un certain nombre d'états définis par le système d'exploitation.

A sa création, un processus est généré dans un état indertéminé (-1) puis il passe ensuite dans un ou plusieurs des états ci-dessous (défini dans `include/linux/sched.h`) lorsqu'il est pris en charge par l'ordonnanceur de tâches[1] :

```
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define TASK_STOPPED            4
#define TASK_TRACED             8
#define EXIT_ZOMBIE             16
#define EXIT_DEAD               32
```

[1] Attention la liste de ces états a été modifiée depuis la noyau 2.6

TASK_RUNNING : le processus est soit en la file d'attente d'exécution soit en exécution.
TASK_INTERRUPTIBLE : c'est un des deux état d'attente. Le processus peut être réveillé n'importe quand par le scheduler, ou bien se réveiller tout seul à la fin d'un timer.
TASK_UNINTERRUPTIBLE : c'est le deuxième état d'attente. Le processus ne se réveillera que dans un seul cas : à la fin de son timer. Cet état est rarement utilisé mais peut être utile pour éviter qu'un processus ne se réveille et ne corrompe des données partagées.
TASK_STOPPED : le processus est stoppé , généralement par un signal système.
TASK_TRACED : le processus a été arrêté par le debugger.
EXIT_ZOMBIE : c'est un état particulier. Le processus a terminé son exécution mais le pére n'a pas exécuté l'appel "wait()" sur son état. Ces processus deviennent alors des fils adoptifs du processus init qui exécute un "wait()" périodiquement pour les éliminer (voir chapitre 3).
EXIT_DEAD : Etat final : le processus est en train d'être retiré car le parent a effectué un "wait()". Le passage de l'état EXIT_ZOMBIE à EXIT_DEAD évite les appels concurrents de plusieurs thread qui effectuent le même "wait()" sur le même processus.

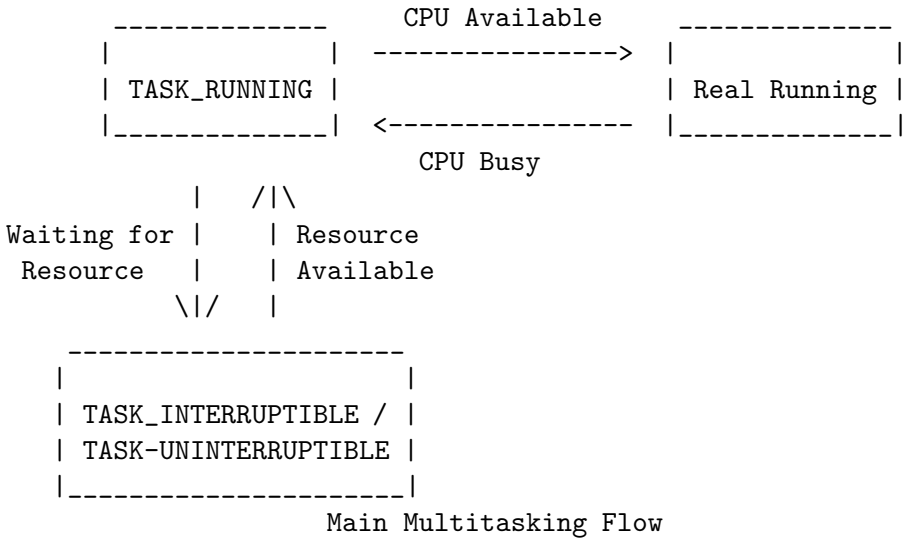


FIG. 9.1 – Figure tirée de [Arcomano, 2003]

9.3 Commutation de tâche

La commutation de tâche ou commutation de contexte consiste à sauvegarder l'état d'un processus ou d'un thread et à restaurer l'état d'un autre processus/thread de façon à ce partager les ressources d'un seul processeur.

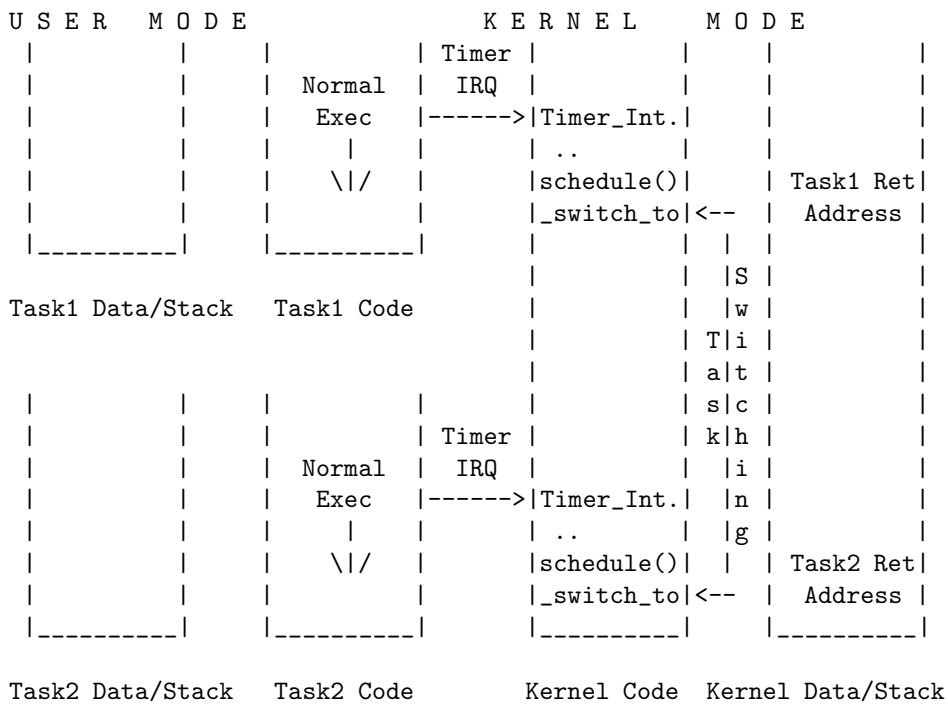
Une commutation de contexte peut être plus ou moins coûteuse en temps processeur suivant le système d'exploitation et l'architecture matérielle utilisés.

Le contexte sauvegardé doit au minimum inclure une portion notable de l'état du processeur (registres généraux, registres d'états, etc.) ainsi que, pour certains systèmes, les données nécessaires au système d'exploitation pour gérer ce processus.

La commutation de contexte invoque au moins trois étapes. Par exemple, en présumant que l'on veut commuter l'utilisation du processeur par le processus P1 vers le processus P2 :

– Sauvegarder le contexte du processus P1 quelque part en mémoire (usuellement sur la pile de P1).
– Retrouver le contexte de P2 en mémoire (usuellement sur la pile de P2).
– Restaurer le contexte de P2 dans le processeur, la dernière étape de la restauration consistant à

reprendre l'exécution de P2 à son point de dernière exécution.



Certains processeurs peuvent sauvegarder et restaurer le contexte du processeur en interne, évitant ainsi d'avoir à sauvegarder ce contexte en mémoire vive. C'est le cas du processeur Intel 80386, avec lequel le premier Linux a été programmé, et de ces successeurs.

```
#define switch_to(prev,next,last) do {                                  \
        asm volatile("pushl %%esi\n\t"                                  \
                     "pushl %%edi\n\t"                                  \
                     "pushl %%ebp\n\t"                                  \
                     "movl %%esp,%0\n\t"        /* save ESP */          \
                     "movl %3,%%esp\n\t"        /* restore ESP */       \
                     "movl $$1f,%1\n\t"         /* save EIP */          \
                     "pushl %4\n\t"             /* restore EIP */       \
                     "jmp ___switch_to\n"                               \
                     "1:\t"                                             \
                     "popl %%ebp\n\t"                                   \
                     "popl %%edi\n\t"                                   \
                     "popl %%esi\n\t"                                   \
                     :"=m" (prev->thread.esp),"=m" (prev->thread.eip),  \
                      "=b" (last)                                       \
                     :"m" (next->thread.esp),"m" (next->thread.eip),    \
                      "a" (prev), "d" (next),                           \
                      "b" (prev));                                      \
 } while (0)
```

Le cas du 80386 (d'après [Vieillefond, 1992])

Le processeur 80386 a été conçu pour pouvoir effectuer automatiquement la commutation de tâches en effectuant les opérations nécessaires vues ci-dessus. Pour ce faire, il dispose :
- *d'un registre de tâche TR*
- *d'un segment d'état de tâche TSS*
- *d'un descripteur d'état de tâche*
- *d'un guichet tâche*

LE SEGMENT D'ETAT DE TACHE : TSS

L'état d'une tâche dépend de l'état des registres. Pour chaque tâche, on définit un segment d'état de tâche unique : TSS (Task State Segment), qui fait partie des objets de type segment système et qui décrit le contexte d'une tâche à un instant donné. Ce segment est accessible par un descripteur situé en GDT, qui lui-même est repéré par un sélecteur situé dans le registre TR (Task Register) interne à l'iAPX 386.
L'octet d'accès d'un descripteur de TSS (32 bits) comporte un champs DPL qui contrôle l'accès au TSS. Comme pour un segment données, il faut que : $EPL \leq DPL$.
Le registre TR, qui identifie la tâche en cours, est composé de deux parties : le sélecteur et une partie non visible du programme. Le sélecteur pointe sur le descripteur de TSS situé en GDT. Il y a alors recopie des caractéristiques du descripteur dans la partie cachée de TR : la base et la limite du TSS en cours. Ainsi, comme les registres segment, ce registre offre au processeur la possibilité de prendre connaissance des éléments du TSS en cours sans qu'il ait à rechercher ce TSS en mémoire. Le segment TSS est alors identifié, il contient 26 doubles mots qu'on peut scinder en 5 parties.

Sélecteur de LDT
Registres généraux
Pointeurs de pile associés à chaque niveau de privilège
Back-link

- *(Back link) représente le lien avec la tâche quittée et permet éventuellement le retour à cette tâche. C'est dans ce mot qu'est localisé le sélecteur du TSS quitté (utile dans le cas d'un système multi-tâches coopératif).*
- *Les pointeurs de pile sont les valeurs initiales des pointeurs de pile respectifs à chaque niveau de privilège : SS0 :ESP0, SS1 :ESP1, SS2 :ESP2.*
- *Le registre CR3 qui contient l'adresse de base du répertoire de pages.*
- *Les registres généraux définissent l'état de la tâche activée. Ils regroupent les registres de travail, les registres de sélecteur, le pointeur de la pile courante SS :ESP, le compteur programme en cours : CS :EIP.*
- *LDT contient le sélecteur de la LDT relative à la tâche activée.*

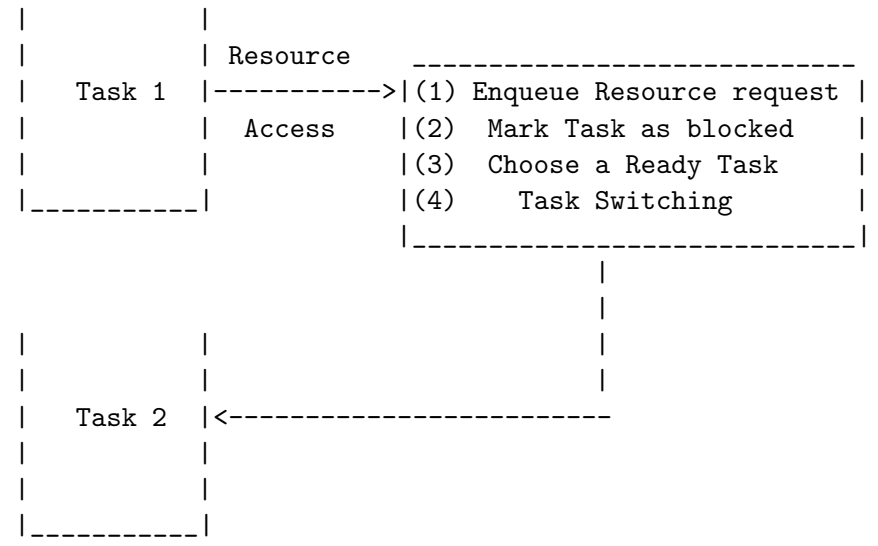
LA COMMUTATION DE TACHES

La commutation de tâches séquence automatiquement les événements suivants :
- *Sauvegarde de tous les registres dans le TSS en cours.*
- *Chargement de TR avec le sélecteur, qui permet d'accéder au descripteur du nouveau TSS.*
- *Chargement des registres du 80386 à partir des valeurs mémorisées dans le nouveau TSS.*
- *Puis la tâche commence là où elle a été arrêtée précédemment.*

S'il y a chaînage des tâches, le bit NT du registre des indicateurs d'état EFlag indique, au moment du retour, que l'on doit effectuer un retour avec commutation de tâches. L'identité de la tâche de retour est déterminée par la lecture du champ (back link) qui mémorise le sélecteur du descripteur du TSS relatif à la tâche de retour.

9.4 Le scheduler

Comme on l'a vu au chapitre 2.4.2, le gestionnaire d'interruption du timer fait appel au scheduler toutes les 10 ms. Ce dernier est chargé de sélectionner le processus le plus adapté parmi la liste des processus. Par ailleurs, il est fait aussi appel au scheduler quand un processus est mis en attente suite à l'absence de disponibilité d'une ressource ou quand une interruption a été traitée par le noyau.



Task Switching based on Waiting for a Resource

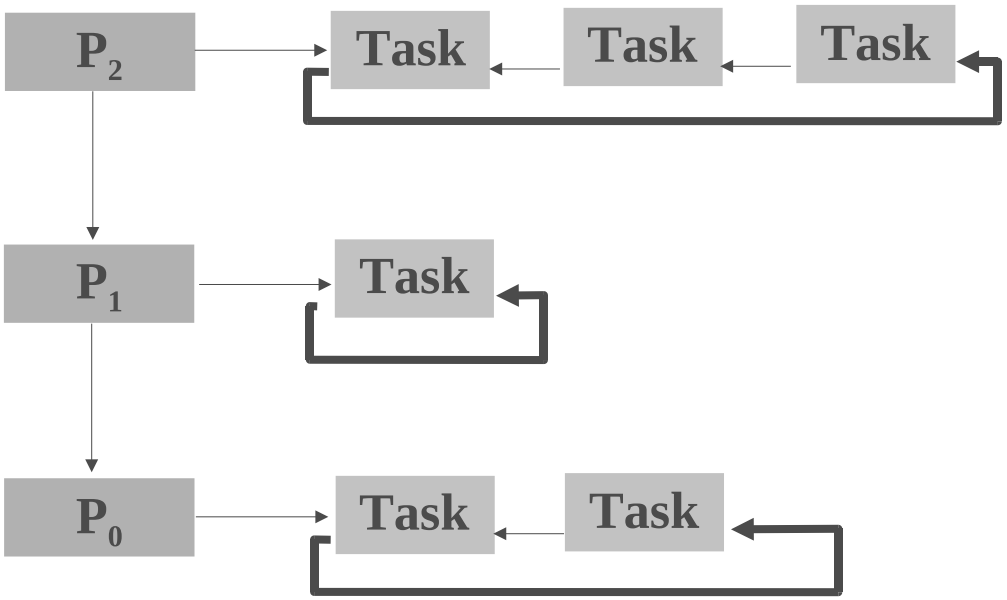
L'algorithme d'ordonnancement d'un système Linux doit répondre à plusieurs objectifs contradictoires :
– le temps de réponse des processus doit être le plus faible possible
– le fonctionnement de tous le processus doit être assuré
– certains processus sont plus importants que d'autres (!)
– il faut interdire qu'un ou plusieurs processus s'accaparent le processeur *ad libitum*...
L'ensemble des régles qui déterminent quand et comment choisir un nouveau processus à exécuter est appelée **politique d'ordonnancement** (scheduling policy).

Ces politiques d'ordonnancement ont énormément variée au cours du temps pour répondre aux besoins du moment : la première distribution officielle de Linux (version 1.0) ne supportait que le processeur i386 et les machines mono-processeur. On a vu ensuite apparaître des contraintes concernant l'utilisation d'architectures différentes (noyau 1.2), des machines multiprocesseurs (noyau 2.0 – Introduction du concept de Symetric Multi Processing, SMP), du temps réel et des threads (noyau 2.2). Depuis le noyau 2.4, les améliorations tendent surtout à améliorer les performances, pour tendre en particulier vers le temps réel : le noyau 2.4 a surtout optimisé la gestion de la mémoire. Dans le noyau 2.6 par contre, l'accent a été mis sur une utilisation quasi temps-réel : en effet, le noyau a été rendu quasi préemptif en introduisant des points de préemption (hooks) dans son exécution. De surcroit, l'ordonnanceur a été entièrement réecrit afin de l'optimiser encore en termes de montée en charge et répartition d'ordonnancement sur multi-processeurs. Ainsi, la complexité de l'algorithme d'ordonnancement est en $O(1)$, c'est à dire en temps borné quelque soit le nombre de processus (il est par contre en $O(n)$ si n est le nombre de processeurs).

Nous commencerons pas analyser le code du premier scheduler de Linux (celui du 0.0.1) afin de comprendre les principes généraux du scheduling puis nous développerons le scheduler en $O(1)$ utilisé du noyau noyau noyau 2.5.2 au noyau 2.6.22. Nous finirons enfin par le dernier scheduler développé mis en oeuvre à partir du noyau 2.6.23 en octobre 2007.

9.4.1 Le noyau 0.0.1

La structure `task_struct` du noyau 0.0.1 définit 2 champs nécéssaires à leur ordonnancement :
`long priority` : ce champ définit la priorité statique du processus
`long counter` : ce champ représente la priorité dynamique du processus (pratiquement, c'est le nombre de ticks d'horloges qui lui reste à consommer dans la "pèriode d'ordonnancement" – voir ci-dessous)
Le principe général de l'ordonannceur du noyau 0.0.1 est de découper le temps en "pèriodes d'ordonnancement"[2]. A chaque début de pèriode, l'ensemble des processus est initialisé avec sa priorité statique (`priority` – initialisé à 15 comme indiqué dans le fichier `include/linux/sched.h`).



A chaque tick d'horloge, comme on la vu dans la gestion sur la gestion du temps (2.4.2), la fonction `schedule()` est appelée. Cette fonction parcourt, simplement la table des processus (en commençant par la fin) et sélectionne le processus dont la priorité dynamique est la plus forte (le premier qui vient dans le cas d'ex-aequo). On rappelle aussi que la fonction de gestion du temps décrémente le champ `counter` du processus courant, c'est à dire sa priorité dynamique.

[2] c'est ma traduction de "epochs" en anglais. Je trouve personnellement que "époques" comme on le traduit parfois, ne couvre pas vraiment le champs sémantique d'"epoch"...

Le code de la fonction `schedule` est donné a titre d'illustration :

```
/*
 *  'schedule()' is the scheduler function. This is GOOD CODE! There
 * probably won't be any reason to change this, as it should work well
 * in all circumstances (ie gives IO-bound processes good response etc).
 * The one thing you might take a look at is the signal-handler code here.
 *
 *   NOTE!!  Task 0 is the 'idle' task, which gets called when no other
 * tasks can run. It can not be killed, and it cannot sleep. The 'state'
 * information in task[0] is never used.
 */
void schedule(void)
{
  int i,next,c;
  struct task_struct ** p;

/* check alarm, wake up any interruptible tasks that have got a signal */

  for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
    if (*p) {
      if ((*p)->alarm && (*p)->alarm < jiffies) {
          (*p)->signal |= (1<<(SIGALRM-1));
          (*p)->alarm = 0;
        }
      if ((*p)->signal && (*p)->state==TASK_INTERRUPTIBLE)
        (*p)->state=TASK_RUNNING;
    }

/* this is the scheduler proper: */

  while (1) {
    c = -1;
    next = 0;
    i = NR_TASKS;
    p = &task[NR_TASKS];
    while (--i) {
      if (!*--p)
        continue;
      if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
        c = (*p)->counter, next = i;
    }
    if (c) break;
    for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
      if (*p)
        (*p)->counter = ((*p)->counter >> 1) +
            (*p)->priority;
  }
  switch_to(next);
}
```

Il est à noter que la priorité statique du processus peut être modifiée par la commande `nice`. En fait, la valeur de `nice` vient simplement s'ajouter à la valeur de la priorité) comme le montre le code de la fonction en question :

```
int sys_nice(long increment)
{
  if (current->priority-increment>0)
    current->priority -= increment;
  return 0;
}
```

9.4.2 Scheduler en O(1) - d'après [Bovet and Cesati, 2005]

A partir du noyau expérimental 2.5.2 en janvier 2002 et jusqu'au noyau 2.6.22 en octobre 2007, l'ordonnanceur utilisé dans le noyau était l'ordonnanceur en $O(1)$ développé par Ingo Molnar. Cet ordonnanceur est évidemment bien plus complexe que celui du 0.0.1. Cependant, il obéit au même principe général : un quantum de temps est donné à chaque processus et ce quantum est décrémenté à chaque fois que ce processus détient la ressource processeur. Toutefois la typologie des processus conditionne la manière dont les quanta sont alloués : les **processus temps réels** sont ainsi traités différements des **processus conventionnels**. Par ailleurs, au sein même des processus conventionnels, on distingue les **processus interactifs** (qui interagissent avec l'utilisateur comme un interpréteur de commande ou une application graphique qui doivent donc avoir un bon temps de réponse) des **processus tournant en tâche de fond** ("batch process" – processus de calcul ou serveurs). Cette classification interactif/batch s'effectue en fonction de l'historique du processus et en particulier de son temps d'endormissement moyen (un processus qui dort souvent est favorisé par le scheduler. En effet, un tel processus est souvent un processus qui accède à des ressources bloquantes ou qui ne se comporte pas de façon "égoïste" en consommant entièrement la tranche de temps qui lui est allouée).

Par ailleurs, le scheduler en $O(1)$ prend en charge les contextes multi-processeurs. Pour ce faire, chaque processeur est doté de sa propre file d'ordonnancement qu'il ordonnance selon l'algorithme général. Pour maintenir l'équilibre des charges entre les processeurs le noyau effectue un rééquilibrage toutes les 200ms.

Processus conventionnels

Priorité statique La priorité statique par défaut d'un processus est 120 mais la fonction `nice` permet de la faire varier entre 100 (la plus haute) et 139 (la plus basse). Cependant, seul l'administrateur peut atteindre les valeurs entre 100 et 119. Un utilisateur standard ne peut qu'augmenter la priorité statique d'un processeur.

Quantum Plus un processus est prioritaire, plus son quantum est élevé. Sa valeur est donnée par la formule suivante :

$$Quantum_{ms} = \begin{cases} (140 - priorite\ statique) \times 20 & si\ priorite\ statique < 120 \\ (140 - priorite\ statique) \times 5 & si\ priorite\ statique \geq 120 \end{cases} \quad (9.1)$$

Priorité dynamique C'est cette priorité qui est utilisée par le scheduler pour calculer l'éligibilité des processus. Comme pour la priorité statique, sa valeur s'échelonne entre 101 et 140. Elle est reliée à la priorité statique via la formule suivante :

$$priorite\ dynamique = \max(100, \min(priorite\ statique - bonus + 5, 139)) \quad (9.2)$$

Le **bonus** dépend du temps moyen passé à dormir.

Temps moyen	Bonus
$\in [0ms, 100ms]$	0
$\in [100ms, 200ms]$	1
$\in [200ms, 300ms]$	2
$\in [300ms, 400ms]$	3
$\in [400ms, 500ms]$	4
$\in [500ms, 600ms]$	5
$\in [600ms, 700ms]$	6
$\in [700ms, 800ms]$	7
$\in [800ms, 900ms]$	8
$\in [900ms, 1000ms]$	9
$1sec$	10

Interactif ou batch ? Un processus est considéré comme **interactif** si $bonus \geq \frac{priorite\_statique}{4} - 23$, sinon il est considéré comme **batch**.

Processus temps réel

Priorité temps-réel A chaque processus temps-réel est associée une priorité temps-réel qui s'échelonne entre 0 (plus forte priorité) et 99 (plus faible priorité). Cette priorité peut être modifiée par les fonctions `sched_setscheduler()` et `sched_param()`.

Tout processus temps-réel d'une priorité donnée s'exécute forcément avant un processus temps-réel de priorité infèrieure (mais dont la valeur de priorité est supèrieure). En réalité un processus temps-réel est toujours considéré comme actif et n'est préempté que si :
– Le processus est préempté par un processus de plus forte priorité
– Le processus est endormi suite à un évenement bloquant
– Le processus est arrêté (TASK_STOPPPED ou TASK_TRACED) ou tué
– Le processus renonce délibérément au CPU grâce à la commande `sched_yield()`
– Le processus a atteint la fin de son quantum
Dans le dernier cas, le comportement dépend de l'algorithme temps-réel utilisé pour ordonnancer le processus. Il en existe 2 :
SCHED_FIFO : C'est le principe "First In-First Out" qui est utilisé. Donc, même si d'autres processus temps-réel de même priorité sont dans la file et si aucun processus de priorité supèrieure, le processus reste actif autant de temps qu'il le désire.
SCHED_RR : C'est le principe "Round Robin" qui est utilisé. Lorsque le processus a consommé son quantum il est mis en queue de sa file.

L'algorithme d'ordonnancement

L'ordonnancement en $O(1)$ (c'est à dire en temps constant quelque soit le nombre de processus en fonctionnement) a été obtenu en créeant autant de files d'attente que de priorités possibles, à savoir, les 100 priorités temps réel plus les priorités des processus ordinaires qui peuvent varier de 100 à 139. Il y a ainsi un ensemble 140 files de priorités (appelé ensemble des files d'attente des processus actifs) associé à chaque processeur ainsi qu'une carte binaire indiquant les files vides. Quand les processus changent de priorité, ils sont simplement déplacés d'une file à une autre. Grâce à la carte binaire il est aisé de séléctionner le processus de plus forte priorité.

Quand les processus ont entièrement consommé leur tranche de temps, ils sont mis dans un ensemble de 140 files de priorité correspondant aux processus expirés. Quand toutes les files actives sont expirés il y a simplement une permutation entre l'ensemble actif et l'ensemble expiré.

Interaction avec le scheduler au niveau utilisateur

Pour interagir avec le scheduler, Linux dispose d'un certain nombre d'appels système mis à la disposition de l'utilisateur.

Appel système	Description
`nice( )`	change la priorité d'un processus [3]
`getpriority( )`	renvoit la priorité max d'un groupe de processus
`setpriority( )`	définit la priorité d'un groupe de processus
`sched_getscheduler( )`	renvoit la politique d'ordonnancement d'un processus
`sched_setscheduler( )`	définit la politique d'ordonnancement et la priorité d'un processus[4]
`sched_getparam( )`	renvoit la priorité d'un processus
`sched_setparam( )`	définit la priorité d'un processus
`sched_yield( )`	abandon volontaire du processeur sans blocage
`sched_get_priority_min( )`	renvoit la priorité min. pour une politique
`sched_get_priority_max( )`	renvoi la priorité max. pour une politique
`sched_rr_get_interval( )`	renvoi le quantum de temps pour une politique Round Robin (tourniquet)

9.4.3 Le "Completely Fair Scheduler" (D'après [Collectif, 2008, Lacombe, 2007b, Kumar, 2008] et le code source évidemment !).

Le Completely Fair Scheduler (ordonnanceur complètement équitable), ou CFS est l'ordonnanceur de tâches écrit par Ingo Molnár (fortement inspiré cependant par le travail de Con Kolivas – "Rotating Staircase Deadline"[Lacombe, 2007a]) qui a fait son apparition à partir de la version 2.6.23 (octobre 2007) du noyau.

Cet ordonnanceur a été écrit suite au constat selon lequel l'ordonnanceur en $O(1)$ supportait difficilement la monté en charge, en particulier dans le contexte multi-processeur. Plusieurs pistes peuvent être données pour expliquer ces difficultés :

– Comme on l'a vu, l'ancien ordonnanceur utilise le temps moyen d'endormissement pour gratifier les processus en considérant qu'ils sont plus "fair-play". Cependant, des processus qui s'endorment souvent peuvent être des processus qui accèdent beaucoup aux ressources et qui peuvent ainsi être à l'origine de réordonnancement fréquents.
– L'évaluation de l'interactivité des processus, au travers du temps d'endormissement est faussée quand le système est chargé. En effet, les statistiques d'endormissement ne peuvent être fiables que s'il y a suffisamment d'échantillons pour les calculer. S'il y a beaucoup de processus à ordonnancer en un temps borné (ce qui est le cas du scheduler $O(1)$) le nombre d'échantillons du temps moyen est divisé d'autant. Ainsi les processus interactifs peuvent temporairement être considérés comme gros consommateurs de CPU (aux moments précis des interactions) ce qui peut augmenter leur temps de latence.
– Certaines attaques du scheduler étaient facilement réalisables.

[4] La plage de priorité est : [-20,19] pour root et [0,19] pour un utilisateur. Plus le chiffre est faible, plus la priorité est élevée.

[4] les trois valeurs possibles sont :
SCHED_FIFO First In First Out
SCHED_RR Round Robin
SCHED_OTHER type non définit
SCHED_BATCH exécution de processus en mode "batch"

Fonctionnement

Contrairement à l'ordonnanceur en $O(1)$, CFS n'utilise pas des files de processus mais un arbre équilibré rouge-noir. L'arbre trie les processus selon une valeur représentative du manque de ces processus en temps d'allocation du processeur, par rapport au temps qu'aurait alloué un processeur dit multitâche idéal, sur lequel tous les processus s'exécuteraient en même temps et à la même vitesse. Ainsi, à chaque intervention de l'ordonnanceur, il "suffit" à ce dernier de choisir le processus le plus en manque de temps d'exécution pour tendre au mieux vers le comportement du processeur multitâche idéal (ce choix s'effectue en $O(1)$ – dû au temps de recherche du minimum dans un arbre équilibré rouge-noir. Cependant, le temps d'insertion lui est en $O(\log n)$). De plus, l'ordonnanceur utilise une granularité temporelle à la nanoseconde, rendant redondante la notion de quantum de temps. Cette connaissance précise signifie également qu'aucune heuristique (basée sur des statistiques, donc pouvant commettre des erreurs) n'est requise pour déterminer l'interactivité d'un processus.

Un processeur multitâche idéal est un processeur qui peut exécuter en même temps (ce qui est bien sur impossible) plusieurs processus en leur donnant chacun une portion équitable de la puissance du processeur. Ainsi, dans le cas où un seul processus est ordonnancé, 100% du processeur lui est alloué. Dans le cas de 2, 50% serait alloué à chacun (en parallèle !). Sur un processeur réel, une tâche seulement peut être exécutée. En conséquence, pendant ce temps, les autres processus attendent le CPU, ce qui les désavantage et rend le processus qui détient le processeur moins "fair-play".

Le Completely Fair Scheduler trace le déséquilibre de "fair-play" au niveau de chaque processus. Quand un processus attend d'avoir le CPU, l'ordonnanceur calcule le temps qu'il aurait dû avoir s'il avait été sur un processeur idéal. Ce temps est calculé en divisant le temps d'attente (exprimé en nanosecondes) par le nombre total de processus en attente. La valeur résultante est le temps CPU auquel le processus a droit. C'est cette valeur qui est utilisée pour trier les processus en vue de l'ordonnancement et qui détermine le quantum de temps alloué au processus avant d'être préempté. La priorité statique (qui peut être modifiée par `nice()`), agit comme un coefficient de pondération qui favorise ou pénalise le processus en amplifiant/réduisant l'écart de temps entre le temps idéal et le temps réellement utilisé. Le coefficient de pondération est tabulé de telle manière que le passage d'un niveau l de `nice` donné au niveau suivant $l+1$ (par exemple de 0 à 1) fait perdre 10% du temps CPU par rapport à un processus resté au niveau l. Pour éviter une commutation de contexte trop importante (et fasse que le temps système devienne prépondérant sur le temps alloué aux processus) il est nécessaire d'assurer qu'un temps CPU minimal est accordé à un processus préempté. Pour ce faire, le niveau de granularité minimal (en nanosecondes) du temps alloué au processus est donné par `/proc/sys/kernel/sched_granularity_ns`. Cependant, en augmentant la granularité on augmente aussi le temps de latence d'une tâche qui est donné par $latence = \frac{n^2 \cdot granularite}{n-1}$ (avec n le nombre de processus à ordonnancer).

L'ordonnanceur modulaire

En plus de ce nouvel ordonnanceur "équitable", la version 2.6.23 du noyau a apporté une gestion plus modulaire de l'ordonnancement. Ainsi a été créée la notion de classe d'ordonnancement (`struct sched_class`) associé à des files d'attentes spécifiques gérées par des modules différents. Le scheduler principal (`kernel/sched.c`) subsiste mais délègue la gestion spécifique aux modules impliqués. Les processus temps réels (liés aux ordonnancements de type `SCHED_FIFO` et `SCHED_RR`) sont gérés par `kernel/sched_rt.c`. Cet ordonnanceur est identique (à un détail près, la suppression de la liste des processus expirés, qui n'est plus nécessaire) à celui présenté pour l'ordonnanceur en $O(1)$. Le "Completely Fair Scheduler" pour sa part, est associé à la gestion de type `SCHED_OTHER`, `SCHED_BATCH` et au module `kernel/sched_fair.c`. Un dernier module (`sched_idletask.c`) se charge lui de la gestion de la tâche "idle" sur chaque processeur.

Notion de groupes d'ordonnancement

A partir du noyau 2.6.24, la notion d'équité du scheduler a été élargie à l'équité entre utilisateurs. Soient par exemple 2 utilisateurs A et B faisant tourner respectivement 2 et 48 tâches sur une même machine. La notion de groupe d'ordonnancement permet d'être équitable avec A et B au niveau utilisateur (et non plus au niveau processus). Ainsi, dans ce cas, chaque utilisateur bénéficie pour lui tout seul de 50% du temps CPU : ces 50% sont partagés entre ses 2 processus pour A et entre ses 48 processus pour B. D'un point de vue implémentation, elle est directe puisqu'il s'agit simplement de chaîner les ordonnanceurs CFS : un ordonannceur est associé à la liste des groupes puis, au sein d'un groupe, un nouvel ordonnanceur est associé pour répartir equitablement le temps CPU entre les tâches.

9.5 Exercice

Soient P1, P2 et P3, 3 processus ayant respectivement comme valeur nice : -2 (processus super-utilisateur), 0 et 2

1. Expliquer l'ordonnancement pendant les 20 premiers ticks
2. Au bout de combien de ticks le processus P3 sera-t-il élu ?
3. Le processus P1 est swappé out. Que devient l'ordonancement ?
4. Le processus P1 est swappé in et le processus P2 est bloqué en attente d'un fichier. Que devient l'ordonancement ?

Chapitre 10

Linux et le temps réel

10.1 Définition (d'après Wikipédia)

Les systèmes informatiques temps réel se différencient des autres systèmes informatiques par la prise en compte de contraintes temporelles dont le respect est **aussi** important que l'exactitude du résultat, autrement dit le système ne doit pas simplement délivrer des résultats exacts, il doit les délivrer dans des délais imposés.

Pour garantir le respect de limites ou contraintes temporelles, il est nécessaire que :
– les différents services et algorithmes utilisés s'exécutent en temps borné. Un système d'exploitation temps réel doit ainsi être conçu de manière à ce que les services qu'il propose (accès hardware, etc.) répondent en un temps borné ;
– les différents enchaînements possibles des traitements garantissent que chacun de ceux-ci ne dépassent pas leurs limites temporelles. Ceci est vérifié à l'aide du test d'acceptabilité[1].

On distingue le temps réel strict ou dur (de l'anglais hard real-time) et le temps réel souple ou mou (soft real-time) suivant l'importance accordée aux contraintes temporelles. Le temps réel strict ne tolère aucun dépassement de ces contraintes, ce qui est souvent le cas lorsque de tels dépassements peuvent conduire à des situations critiques, voire catastrophiques : pilote automatique d'avion, système de surveillance de centrale nucléaire, etc. À l'inverse le temps réel souple s'accommode de dépassements des contraintes temporelles dans certaines limites au-delà desquelles le système devient inutilisable : visioconférence, jeux en réseau, etc.

10.2 Le noyau "classique" et le temps-réel

10.2.1 Avant le noyau 2.6

Avant la verson 2.6, le noyau n'est pas du tout préemptible et donc toute notion de temps-réel, même mou, est exclue. En effet, comme nous l'avons indiquée plus tôt, bien que l'arrivée d'une interruption force le passage en mode noyau vers la routine de traitement de l'interruption en question, un processus en mode noyau n'est pas préemptible par un autre processus. De ce fait, il est n'est pas déterministe. Imaginons par exemple qu'un processus A déclenche une interruption. Dans ce cas, il commute en mode noyau pour la traiter. Si, pendant ce temps, une deuxième interruption, gérée par un autre processus B plus prioritaire est déclenchée, il faudra attendre la fin du traitement de la première interruption pour pouvoir traiter la seconde. Un autre cas de figure est le suivant : considérons un processus qui a

[1] c'est-à-dire une preuve que les limites temporelles ne seront jamais dépassées quelle que soit la situation. L'existence de cette preuve dépend de l'ordonnanceur utilisé et des caractéristiques des tâches du système. Pour un système souple, on pourra se contenter de mesures statistiques.

consommé l'intégralité du quantum de temps qui lui était attribué mais a déclenché une interruption avant la fin du quantum. Dans ce cas, il faudra attendre la fin du traitement de l'interruption pour réordonnancer les processus.

10.2.2 A partir du noyau 2.6 (d'après [Lefranc, 2004])

A partir du noyau 2.6 le noyau a été est rendu préemptible en ajoutant des points de préemption ("hooks") au coeur du noyau. Ces points de préemption sont des instructions qui permettent au noyau de s'interrompre afin d'exécuter un processus de plus haute priorité. Les décisions d'ordonnancement peuvent avoir lieu dans les quatre circonstances suivantes :
- lorsque le processus passe de l'état d'exécution à l'état d'attente (à cause d'une requête d'entrée/sortie par exemple) ;
- lorsque le processus passe de l'état d'exécution à l'état prêt (si une interruption est levée par exemple) ;
- lorsque le processus passe de l'état d'attente à l'état prêt (à la fin d'une requête d'entrée/sortie par exemple) ;
- lorsque le processus se termine.

Lorsque l'ordonnancement n'a lieu que dans le premier et dernier cas, on dit que le schéma d'ordonnancement est non préemptif. Dans les autre cas, on dit que le schéma est préemptif.

Le noyau a donc été doté de ces points de préemption. Cependant, certaines parties critiques du noyau sont restées non-préemptives afin de garantir que la robustesse du noyau (contexte multi-processeurs, utilisation du co-processeur arithmétique).

L'ajout de la possibilité de préemption du noyau et l'existence de classes d'ordonnancement de type "temps-réel" (donc ne fonctionnant pas selon le principe de "time-sharin") permet de diminuer de façon significative les temps de latence des applications interactives et offre de vraies caractéristiques temps-réel au noyau 2.6. Cependant, comme le noyau n'est pas encore complètement préemptif, les performances atteintes sont celle d'un temps réel mou au sens où on l'a défini plus haut : en moyenne les temps de réponse sont réspectés mais on ne peut assurer précisément dans tous les cas.

10.3 Les noyaux Linux Temps-réel (d'après [Ferre, 2000])

Etant donnée la notoriété grandissante de Linux, liée principalement aux qualités intrinsèque des logiciels libres et ouverts (robustesse du code et documentation fournie et multi-lingues apportés par la communauté, support multi-architectures, modularité...), mais du fait de ses faiblesses sur les aspects temps réels, certaines solution tâchant d'adapter Linux au temps réel "dur" ont été développées. Nous citerons principalement RTLinux dont nous détaillerons les fonctionnalités mais on peut citer aussi RTAI (un fork de RTLinux utilisé surtout en milieu universitaire), ADEOS, ART Linux, KURT,...

L'idée de RTLinuxest de faire cohabiter au sein d'un même système d'exploitation un micro-noyau temps réel et le noyau Linux "classique". Ainsi, l'utilisateur peur continuer à utiliser les services de haut niveau, tout fournissant un comportement déterministe avec un temps de latence est faible pour les tâches qui le nécessitent.

Ainsi, le coeur de RTLinux est un micro-noyau qui exécute le véritable noyau Linux comme sa tâche de plus faible priorité. RTLinux implémente en fait une sorte de machine virtuelle pour que le noyau Linux standard soit complètement préemptif. Dans cet environnement, toutes les interruptions sont initialement prises en compte par le noyau temps réel et sont passées à Linux seulement s'il n'y a pas de tâche temps réel à exécuter.

Pour minimiser les changements de contexte avec le noyau Linux, les concepteurs ont émulé le contrôleur d'interruptions matériel. Ainsi, quand Linux masque une interruption, celle-ci est quand

même prise en compte par RTLinux et aiguillée vers une file d'attente si elle n'est pas utile au niveau du micro-noyau. Lorsque Linux rétablit la possibilité de traiter les interruptions, celles arrivées entre temps sont disponibles et peuvent être traitées par la routine de gestion d'interruptions de Linux (handler d'IT Linux). Les codes assembleurs de masquage d'interruption et de retour d'interruption sont remplacés dans le noyau Linux par des macro-instructions équivalentes (S_CLI, S_STI et S_IRET). Toutes les interruptions matérielles sont attrapées par l'émulateur qui distribue des interruptions logicielles (soft interrupts).

Les tâches temps réel suivent les spécifications de la norme POSIX.1c décrivant les threads. Elles peuvent êtres des applications utilisateur, un service fourni par l'environnement RTLinux ou un pilote de périphérique. Chargées en tant que module noyau, elles s'exécutent en suivant les ordres dictés par le scheduler. Le temps CPU est fourni à la tâche de plus haute priorité (scheduler de base RTLinux).

Ces "kernel threads" disposent des droits entiers sur la machine et peuvent accéder à toutes les ressources. Ils occupent tous le même espace d'adressage (celui du noyau !) et se chargent dynamiquement. Il résulte de ces trois points qu'il est délicat de développer un tel module puisque la moindre erreur de programmation est fatale au système. En revanche, cette architecture permet d'améliorer sensiblement les performances en éliminant les temps de changement de niveau de protection et en raccourcissant le temps de changement de contexte. Toutes les ressources d'une tâche RTLinux sont allouées statiquement (exemples : mémoire, fifo).

Un règle importante cependant sous RTLinux est que si un service est intrinsèquement non temps réel, il doit être fourni par Linux et non par les modules RTLinux. Ceci s'applique naturellement aux pilotes de périphériques (device drivers) qui, s'ils n'ont aucune contrainte temps réel à tenir, restent inchangés. Ils doivent tout de même être recompilés dans l'environnement RTLinux.

Chapitre 11

La gestion de la mémoire

La mémoire centrale est divisée en 2 parties :
– L'espace noyau.
– L'espace utilisateur qui est allouée aux processus actifs.
La taille du noyau ne varie pas ou peu au cours du temps (les seules variations proviennent du chargement/déchargement des modules principalement) mais celle allouée aux processus est en perpetuel changement puisque les processus peuvent être créés/détruits à tout moment.

L'idée de la pagination et de la segmentation est d'une part de pouvoir utiliser un espace mémoire supérieur à la taille de la mémoire physique et de rendre indépendants l'espace d'adressage virtuel et l'espace d'adressage physique.

11.0.1 Segmentation

Le but de la segmentation est d'offrir à l'utilisateur une vue de la mémoire correspondant à des régions, dédiés à une utilisation particulière comme par exemple : le code d'un programme, les données, la pile, un ensemble de sous-programmes, des modules, un tableau, etc. Chaque objet logique est alors désigné par un segment particulier.

Ainsi, la segmentation indexe la mémoire en segments alloués uniquement sur besoin, dont les adresses sont stockés dans une table de descripteurs de segment. Cette table contient des données propres aux segments, comme :
– un pointeur sur l'adresse linéaire du début du segment.
– un pointeur sur l'inode du fichier dont le contenu est initialement chargé dans la région.
– le type : code, partagée, données privées ou pile
– la taille
– le statut (verrouillée, demandée, chargement en cours, valide)
– un compteur de référence des processus l'utilisant
Les adresses logiques sont alors découpées en deux parties : une partie qui sert à **sélectionner** le segment dans la table et une partie qui indique le **décalage** dans ce segment. Pour obtenir une adresse linéaire à partir d'une adresse logique, il suffit de regarder dans la table des descripteurs de segment à l'index contenu dans la partie de selection, d'extraire l'adresse linéaire du segment contenue dans le descripteur et d'y ajouter le décalage de la seconde partie de l'adresse logique.

Segments d'un processus Un processus dispose par défaut de trois segments : un segment dans lequel se trouve le code (souvent appelé *text*), un segment contenant les données (*data*) et la pile (*stack*). D'autres segments peuvent être ajoutés comme les bibliothèques partagées, les fichiers ouverts, etc...

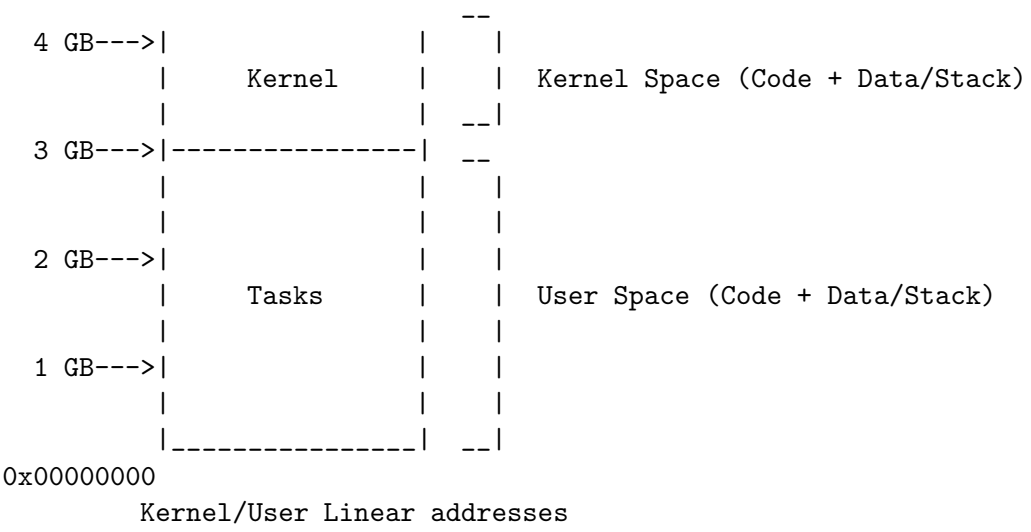
MMU Généralement, les processeurs actuels disposent d'une unité matérielle pour effectuer cette traduction (*segmentation unit* dans le *MMU*), mais ceci reste cependant plutôt propre à la famille *Intelx*86. La mise en oeuvre de la segmentation est donc ainsi dépendante de l'architecture sur la quelle s'éxécute le noyau.

Sous Linux, la segmentation est désactivée par défaut (elle s'active uniquement si nécéssaire), les processus ont donc directement accès à l'espace d'*adressage linéaire*.

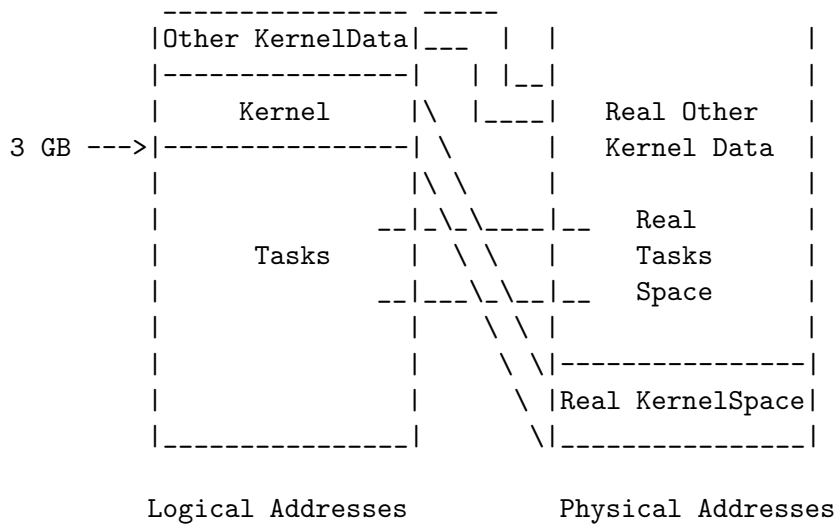
Le noyau Linux comporte ainsi uniquement 4 segments :

1. Le segment de Code Noyau [0x10]
2. Le segment de Données/Pile Noyau [0x18]
3. Le segment de Code Utilisateur [0x23] (utilisé par tous les processus utilisateurs)
4. Le segment de Données/Pile Utilisateur [0x2b] (utilisé par tous les processus utilisateurs)

L'espace noyau va de [0xC000 0000] (3 GB) à [0xFFFF FFFF] (4 GB). Tandis que l'espace utilisateur va de [0] (0 GB) to [0xBFFF FFFF] (3 GB).



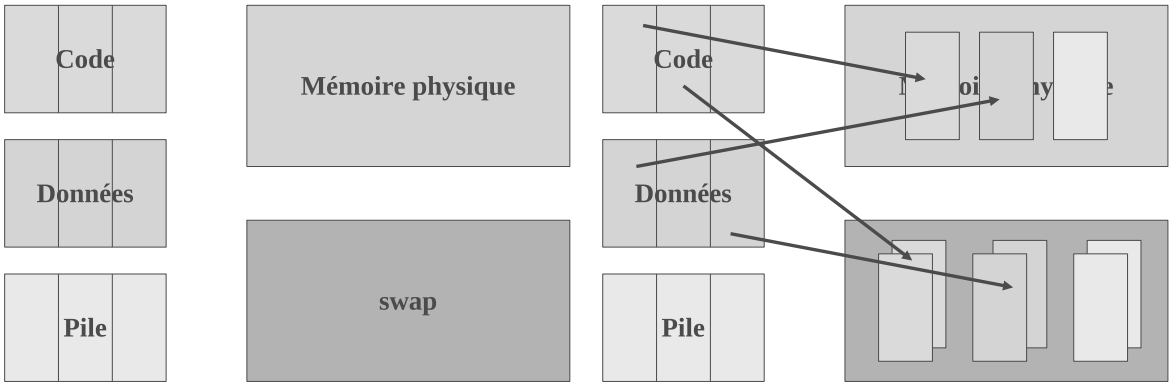
En mode noyau, les adresses linéaires sont identiques aux adresses physiques au détail près qu'elles sont simplement translatées de 3 Go. Il n'y a donc en fait dans ce cas pas de mémoire virtuelle puisque l'adresse linéaire s'interpréte directement en adresse physique.



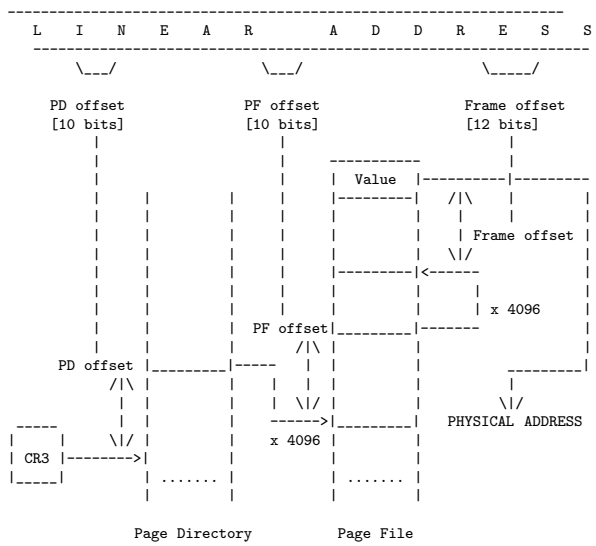
Dans les architecture utilisant la famille $Intelx86$ les segments sont adréssés par des registres spécifiques :
– CS pour le segment de Code
– DS pour le segment de Données
– SS pour la Pile
– ES pour d'autres segments (utilisé, par exemple, pour faire une copie entre 2 segments)

11.0.2 Pagination

Le mécanisme de pagination consiste lui à découper la mémoire en un certain nombre de *pages* de taille modérée. La mémoire physique est également décomposée en zones de même taille, appelées "cadres de pages" (frames en anglais), dans lesquelles prennent place les pages. Un dictionnaire de traduction (translation, ou génération d'adresse) assure la conversion des adresses virtuelles en adresses physiques, en consultant une "table des pages" (page table en anglais) pour connaitre le numéro du cadre (donc l'adresse physique) qui contient la page recherchée. Il peut bien surx y avoir plus de pages que de cadres... et c'est d'ailleurs là tout l'intérêt de ce mécanisme ! (il faudra cependant gérer les conflits...) En ce qui concerne les pages qui ne sont pas en mémoire, elles sont stockées sur le disque, et seront ramenées dans un cadre en fonction des besoins. Ainsi, le principe de la pagination est de ne garder qu'un faible nombre de pages en mémoire physique alors que le reste est sur le disque.



L'espace d'adressage linéaire correspond donc à un re-mapping de l'espace d'adressage physique. Classiquement, une adresse linéaire est divisée en trois champs : un **répertoire**, une **table** et un **décalage**. Il s'agit donc d'une architecture à deux niveau.



Sur les architectures $x86$, les pages font 4Ko, et les adresses linéaire de 32 bits sont décomposées en 10

bits de répertoire, 10 bits de table et 12 bits d'offset. Sur les architectures 64 bits, il y a généralement entre 3 et 4 niveaux de pagination.

Pour traduire une adresse linéaire en adresse physique, il suffit de regarder dans le répertoire des pages à l'entrée indiquée par la partie répertoire de l'adresse linéaire. Plusieurs cas peuvent se présenter :

1. L'entrée est valide : elle se substitue au numéro de page pour former l'adresse physique.
2. L'entrée dans la table des pages est invalide. Dans ce cas il faut trouver un cadre libre en mémoire physique et mettre son numéro dans cette entrée de la table des pages. Puisque les processus peuvent augmenter dynamiquement la mémoire qu'ils adressent (avec *malloc* ou *brk*, il faut éviter de fragmenter la mémoire en allouant un maximum de pages contigües. Il existe pour cela des algorithmes très poussés sur la recherche de pages libres.
3. L'entrée dans la table des pages est valide mais le processus ne détient pas les droits suffisants en lecture ou écriture.
4. L'entrée dans la table des pages est valide mais correspond à une adresse sur la mémoire de masse où se trouve le contenu du cadre. Un mécanisme devra ramener ces données pour les placer en mémoire physique.

Dans les deux derniers cas, une interruption appelée défaut de page (ou parfois faute de page, traduction de l'anglais page fault) est générée par le matériel et donne la main au système d'exploitation. Celui-ci a la charge de trouver un cadre disponible en mémoire centrale afin de l'allouer au processus responsable de ce défaut de page, et éventuellement de recharger le contenu du cadre par le contenu sauvé sur la mémoire de masse (couramment le disque dur, sur une zone appelée zone d'échange ou swap).

Il se peut qu'il n'y ait plus aucun cadre libre en mémoire centrale : celle-ci est occupée à 100 %. Dans ce cas un algorithme de pagination a la responsabilité de choisir une page "victime". Cette page se verra soit immédiatement réaffectée au processus demandeur, soit elle sera d'abord sauvegardée sur disque dur, et l'entrée de la table des pages qui la référence sera mise à jour. On notera que la page victime peut très bien appartenir au processus qui manque de place.

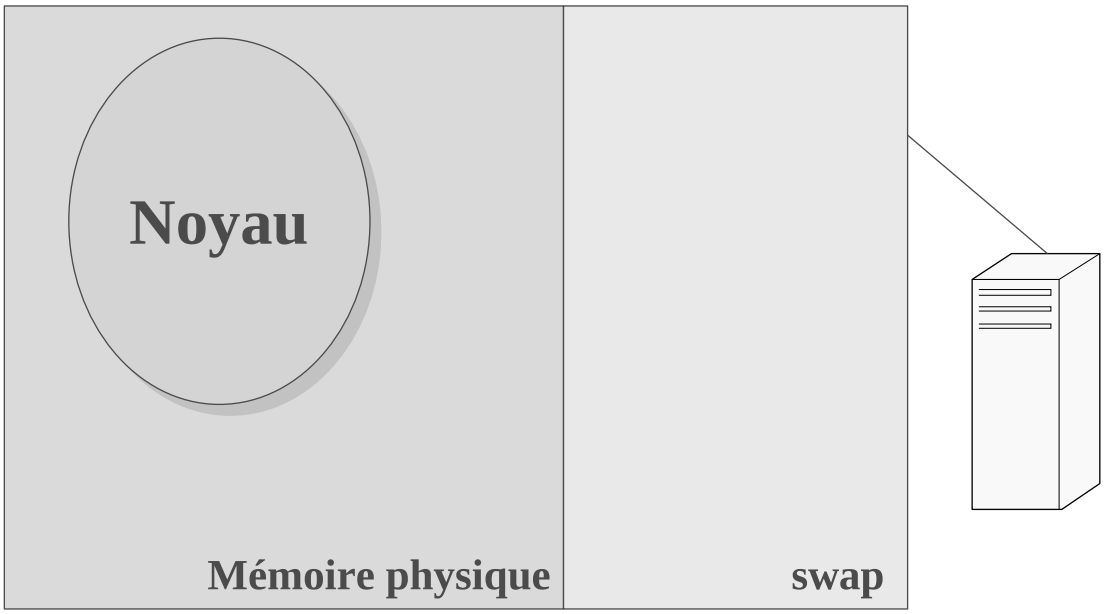
MMU Du point de vue matériel, le *MMU* est souvent capable de faire la traduction d'une adresse linéaire en adresse physique (circuit *page unit*). Pour cela, il suffit de charger dans un registre l'adresse du répertoire des pages. Chaque processus sauvegarde l'adresse de son réprertoire de page dans son descripteur et lors de la commutation de processus, le noyau charge ce registre avec la bonne valeur, mettant ainsi le processus dans le bon espace d'adressage. Pour chaque nouveau processus, il faut donc créer un nouveau répertoire de pages qui lui sera associé. L'allocation des tables de pages peut se faire dynamiquement (sauf la première, bien entendu).

copy on write Lors de la création d'un nouveau processus (avec *fork()*), il faudrait en théorie dupliquer toutes les zones de mémoire. Cependant, comme *fork()* est très souvent suivi de *exec()* qui va remplacer toute la mémoire, les implémentations modernes ne font une copie que lors de l'écriture (*copy on write*) pour s'éviter des écritures inutiles. Typiquement, les adressage linéaire de deux processus après *fork* sont les mêmes, jusqu'à ce qu'ils écrivent des données.

11.1 Swapping

Dans la section précédente, nous avons vu que le mécanisme de pagination permettait de considérer que les zones de mémoires accessibles par le système pouvaient non seulement se trouver en mémoire physique mais aussi sur le disque. Le mécanisme permettant de transférer des zones de la mémoire centrale dans une zone du disque est appelée **swap in**, celui permettant de la récupérer par la suite

swap out. La zone de swap est gérée par groupement de blocs et sans cache et ne doit pas être structurée en système de fichiers.



Les systèmes Unix anciens ont utilisé comme méthode de swap la technique dite du "va-et-vient". Ce mécanisme est relativement basique, en effet, dans ce cas, les zones nécéssaires au fonctionnement d'un processus sont entièrement chargées par le système. Lors d'un manque de mémoire, certains processus sont entièrement transférés sur disque par un processus spécifique appelé swapper (en fait le processus 0), selon une stratégie qui prend en compte l'état du processus (bloqué ou non), sa priorité et le temps qu'il a passé en mémoire. C'est le même processus qui s'occupe de les rechargés quand ils sont disponibles.

La pagination à la demande a ensuite été introduite dans les systèmes Unix à partir de BSD 4.3 et System V : l'idée est que seules les pages nécessaires à l'exécution d'un processus soient appelées en mémoire physique. C'est le daemon voleur de pages qui est chargé d'effectuer ce travail : périodiquement il vérifis voir si le nombre de pages libres ne dépasse pas un seuil min. Dans ce cas, il transfère des pages sur le disque. Il existe de nombreuses stratégies pour le choix des pages à transférer[1]

Linux est un système purement paginé. Dans ce sens il n'effectue par réellement de swap mais stocke dans la zone de swap les pages mémoires inutilisées. Ces opérations sont réalisées à la demande du noyau par le daemon kswapd lors de l'occurence de fautes de pages. Ce dernier se charge de maintenir un certain nombre de pages libres en mémoire. Il vérifie périodiquement ou après une forte allocation de mémoire, l'espace disponible. Si cet espace devient insuffisant, il libère certaines pages en prenant soin de les recopier sur disque au préalable. Le choix de ces pages est fait en fonction de l'algorithme de l'horloge.

[1]
- FIFO - First In First Out : ordre chronologique de chargement
- LRU - Least Recently Used : ordre chronologique d'utilisation
- Algorithme de l'horloge
- LFU - Least Frequently Used
- Random : au hasard
- MIN : algorithme optimal

11.2 La gestion du cache disque

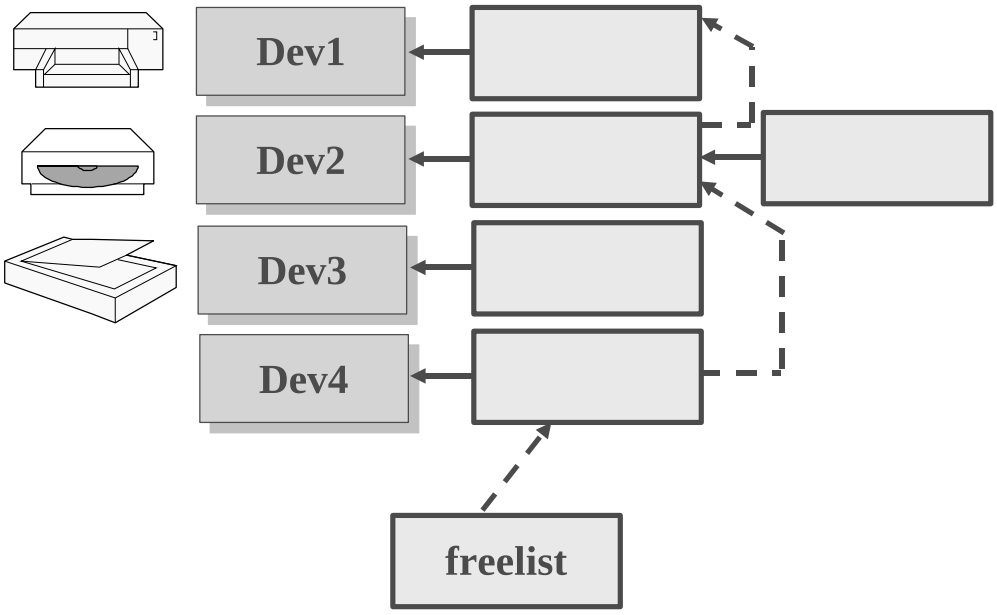
Le cache (ou antémémoire comme préconise de l'appeler l'AFNOR[2]) est une région de la mémoire utilisée pour le accélérer les accès aux fichiers. Cette zone permet d'accéder directement et rapidement au contenu de ces fichiers sans avoir à relire/réécrire les informations sur le disque dont le temps d'accès est lent. Ce mécanisme accélère les accès disque mais pose des problèmes de cohérence entre le contenu du cache et celui des fichiers.

11.2.1 Principe

Les accès au disque se font par blocs. La fréquence des accès disques est diminuée si certains blocs sont conservés en mémoire plutôt qu'être écrits directement sur le disque. En pratique, un bloc n'est effectivement écrit sur le disque que si le cache est saturé. N'écrire les blocs qu'en cas de saturation du cache peut cependant être risqué si un incident matériel survient entre temps ("plantage", coupure de courant...). On remédie à ce problème en exécutant un commande qui recopie les blocs sur le disque à intervales réguliers (depuis la version 2.6 du noyau c'est le thread noyau `pdflush` qui s'en charge).

Coté lectures, l'amélioration des performances se fait par anticipation des lectures : en lisant par blocs, on augmente la probabilité que l'information soit présente en mémoire plutôt que sur le disque.

La gestion du cache a énormément changé au cours du temps. Pour des raisons de simplicités, nous donnerons ici la première implémentation effectuée par Linux Torvalds dans la version du noyau 0.0.1.



11.2.2 Implémentation des caches dans Linux 0.0.1

Dans la version 0.0.1 du noyau, la gestion du cache est effectuée en conservant en mémoire (en zone système) la liste des blocs auxquels le système à accès (`struct buffer_head * start_buffer = &end` – `end` fin de la mémoire accessible par l'utilisateur avec GCC). Elle n'est évidemment pas swappable afin d'assurer l'intégrité des données qui y sont stockées.

Un tampon (buffer – structure `struct buffer_head`) est une zone de mémoire auquel un bloc physique est associé. A chaque disque accessible par le système est alors affectée une liste doublement chaînée contenant les tampons qui lui sont associés. Le cache est ainsi en fait constitué de l'ensemble de tous

[2]Association Française de NORmalisation. Représentant français à l'ISO.

ces tampons, quelque soit les disques auxquels ils sont affectés. La taille du cache est définie à la génération du noyau dans `include/linux/config.h` (`BUFFER_END 0x200000`).

Chacune des listes chaînées affectées à chacun des périphériques est accessible directement via une table de hachage `struct buffer_head *hashtable[NR_HASH]`. Par ailleurs est aussi conservé un pointeur sur la liste doublement chaînée des tampons libres `free_list`.

Un tampon est ainsi, soit dans la listes des tampons libres, soit dans la liste hash-codée des tampons associés à un périphérique (ou les 2). Ces listes sont circulaires et doublement chaînées. L'entête des tampons contient les informations permettant la recherche du bloc dont le tampon est la copie (`voir include/linux/fs.h`).

```
struct buffer_head {
        char * b_data;                  /* pointer to data block (1024
           bytes) */
        unsigned short b_dev;           /* device (0 = free) */
        unsigned short b_blocknr;       /* block number */
        unsigned char b_uptodate;
        unsigned char b_dirt;           /* 0-clean,1-dirty */
        unsigned char b_count;          /* users using this block */
        unsigned char b_lock;           /* 0 - ok, 1 -locked */
        struct task_struct * b_wait;
        struct buffer_head * b_prev;
        struct buffer_head * b_next;
        struct buffer_head * b_prev_free;
        struct buffer_head * b_next_free;
};
```

`b_data` : pointeur en mémoire du tampon représentant le bloc
`b_dev` : numéro du périphérique
`b_blocknr` : numéro du bloc sur le périphérique
`b_uptodate` : indique si le tampon est valide
`b_dirt` : indique si le tampon n'a pas été utilisé par le système (clean) ou doit être réécrit sur le disque (dirty)
`b_count` : est le compteur d'utilisations du tampon
`b_lock` : verrou du tampon
`b_wait` : pointeur sur les processus voulant accéder à ce tampon.
`b_prev` et `b_next` : pointeur de chaînage dans la `hashtable`.
`b_prev_free` et `b_next_free` : pointeur de chaînage dans la `free_list`.

Un tampon peut-être :
– disponible et non attribué à un pilote, chaîné dans la `free_list`.
– utilisé par un pilote et chaîné dans une table de hachage de la `hastable`.
– disponible et attribué à un pilote sans être actif, chaîné simultanément à la `free_list` et à une table de hachage de la `hastable`.

11.2.3 Algorithmes

Lecture

Il y a 4 cas intéressants :
– Le tampon est dans la liste de hachage de la `hashtable`
  1- le tampon est dans la `free_list`

2- le tampon est déjà alloué
– Le tampon n'est pas dans la liste de hachage de la `hashtable`
3- un tampon de la `free_list`
4- la `free_list` est vide

1. Le chaînage de la liste de hachage de la `hashtable` n'est pas modifié et le tampon est supprimé de la `free_list`
2. attente de la libération du tampon
3. Le premier tampon de la `free_list` est transféré dans la liste de hachage de la `hashtable`. Les informations ne sont pas présentes dans le tampon et il faut les charger.
4. Le processus demandeur s'endort jusqu'à la restitution d'un tampon.

écriture

Pour écrire un bloc, il faut écrire les tampons qui le constituent, puis transférer ces tampons en début de `free_list`. En mode synchrone, il faut écrire le bloc puis restituer le tampon. Il faut alors vérifier que *ce* tampon n'est pas demandé par un processus puis si *un* tampon n'est pas demandé. On peut alors remettre le tampon dans la `free_list`.

Pour une explication exhaustive de la gestion mémoire sous Linux, reportez vous à [Bovet and Cesati, 2005] ou au PDF suivant [Nayani et al., 2002].

11.3 Quelques commandes pour visualiser l'utilisation mémoire

La commande la plus simple pour afficher les informations relatives à l'occupation mémoire est `free` :

```
#free
             total       used       free     shared    buffers     cached
Mem:       1035704     980544      55160          0      55692     553608
-/+ buffers/cache:     371244     664460
Swap:      1044184          0    1044184
```

Nous trouvons ainsi la mémoire physique totale, la mémoire utilisée (`used`), la mémoire libre (`free`), la mémoire partagée (`shared`), la mémoire utilisée dans les buffers (`buffers` – les buffers sont des zones mémoires utilisées pour stocker temporairement des méta-données systèmes tels que les informations inodes), le cache mémoire (`cached`), la taille totale du swap et le swap utilisé.

La commande `vmstat` vmstat fournit des informations à propos des processus, de la mémoire, de la pagination, des entrées-sorties, des interruptions et de la répartition du temps CPU.

```
#vmstat
procs -----------memory---------- ---swap-- -----io---- -system-- ----cpu----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa
 2  0      0  50692  56484 553920    0    0    51    42  414 1508 16  4 79  1
```

```
 Procs
 r: Nombre de processus en compétition pour le temps CPU.
 b: Nombre de processus dormants.
 w: Nombre de processus transférés dans le swap mais qui ne seraient exécutés
 sinon. Ce champs est calculable, mais Linux ne fait jamais de swap intégral
 aussi désespéré.
 Memory

 swpd: Quantité de mémoire virtuelle utilisée (ko) [Ndt: mémoire disponible -
 mémoire utilisé + swap utilisé ]
 free: Quantité de mémoire [Ndt: physique !] libre (ko).
 buff: Quantité de mémoire utilisée comme tampons d'E/S (ko).


 Swap

 si: Quantité de mémoire paginé lue depuis un disque en ko/s.
 so: Quantité de mémoire paginé transférée sur disque en ko/s.


 IO

 bi: Blocs écrits par seconde sur des périphériques orientés bloc.
 bo: Blocs lus par seconde sur des périphériques orientés bloc.


 System

 in: Nombre d'interruptions par seconde, y compris l'horloge.
 cs: Nombre de changement de contextes (context switches) par seconde (appels
 systèmes + commutations de tâches).

 Il s'agit de la répartition du temps CPU en pourcentages.

 us: user time : temps consommé par les processus
 sy: system time : temps passé dans le noyau
 id: idle time : temps CPU inutilisé
```

Le fichier `/proc/meminfo` donne des informations très détaillées sur la mémoire.

```
#cat /proc/meminfo
MemTotal:      1035704 kB
MemFree:         48272 kB
Buffers:         56804 kB
Cached:         554068 kB
SwapCached:          0 kB
Active:         556540 kB
Inactive:       353304 kB
HighTotal:      130912 kB
HighFree:          256 kB
LowTotal:       904792 kB
LowFree:         48016 kB
SwapTotal:     1044184 kB
SwapFree:      1044184 kB
Dirty:             152 kB
Writeback:           0 kB
AnonPages:      298976 kB
Mapped:          76980 kB
Slab:            55448 kB
SReclaimable:    45128 kB
SUnreclaim:      10320 kB
PageTables:       2500 kB
NFS_Unstable:        0 kB
Bounce:              0 kB
CommitLimit:   1562036 kB
Committed_AS:   537952 kB
VmallocTotal:   114680 kB
VmallocUsed:     17892 kB
VmallocChunk:    96468 kB
HugePages_Total:     0
HugePages_Free:      0
HugePages_Rsvd:      0
Hugepagesize:     4096 kB
```

Annexe A

Éléments de présentation des microprocesseurs de la famille 80x86

A.1 Préambule

Le code source du noyau 0.0.1, qui sert à illustrer le détail des implémentations choisies dans Linux pour mettre en place les éléments du système d'exploitation, est basé sur le processeur 80386. C'est pour cette raison que nous le détaillons ici. Cependant, la plupart des microprocesseurs actuels possèdent des fonctionnalités équivalentes à celui-ci. En particulier, un mode protégé qui permet de séparer les espaces noyau et utilisateurs.

A.2 Les registres du 80x86

Le microprocesseur Intel 80386 est un processeur 32 bits apparu en 1985. Il succède au 286, au 186, et au 8086. Il hérite de certaines de leurs caractéristiques et garde une certaine compatibilité avec eux.

Le 80386 dispose de 16 registres :
- 4 registres généralistes 32 bits utilisés pour contenir des données ou des pointeurs : EAX, EBX, ECX, EDX
- 4 registres 32 bits de pointeurs et d'index : ESI, EDI, EBP, ESP, EIP
- 6 registres 16 bits de segments utilisés pour pointer sur les différents segments (données, code, etc.) : CS, DS, DS, ES, FS, GS
- les registres de contrôle et de statut qui donnent des informations sur l'exécution du programme en cours : EFLAGS, CR0 (Machine Status Word, Task Switched), CR1, CR2, CR3, CR4 (PCE Flag), CR8, DR0, DR1, DR2, DR3, DR6, DR7, GDTR (Global Descriptor Table Register), IDTR (Interrupt Descriptor Table Register), LDTR (Local Descriptor Table Register), MSRs (Model Specific Registers), MXCSR, TSS (Relationship To Task register), Task register, etc.

A.2.1 Registres généralistes (32 bits)

- Le registre généraliste d'accumulation EAX qui peut contenir les données des utilisateurs (ce registre joue le rôle d'opérande implicite dans de nombreuses opérations : mul, div, etc. Dans ce cas le résultat de l'opération est stocké dans ce même registre).
- Le registre ebx (base index) est utilisé comme pointeur sur les données. Usuellement, ce registre contient un offset relatif au segment de données permettant de repérer une information de ce segment.

– Le registre ECX utilisé comme compteur, utilisé pour les boucles et les opérations sur les chaînes de caractères.
– Le registre EDX utilisé pour les entrées/sorties, pour contenir des offsets lors des opérations d'affichage ou de saisie.

A.2.2 Registres pointeurs et index (32 bits)

– Le registre EIP (Instruction Pointer) contient l'offset de la prochaine instruction à exécuter. Il est modifié automatiquement à chaque exécution et peut être manipulé par des instruction de déroutement du type jmp, call, ret, etc. On ne peut pas accéder directement à ce registre.
– Le registre EDI (Destination Index) et le registre ESI (Source Index), qui sont utilisés lors des opérations sur les manipulations, copies... de suites d'octets.

A.2.3 Registres de segment (16 bits)

– Le registre CS (Code Segment) contient le numéro du segment mémoire dans lequel sont stockées les instructions assembleur du code à exécuter. On ne peut pas accéder directement à ce registre.
– Le registre SS (Stack Segment) contient le numéro du segment mémoire dans lequel est stockée la pile. On peut accéder directement à ce registre ce qui permet d'utiliser plusieurs piles.
– Le registre ESP (Stack Pointer) contient le déplacement pour atteindre le sommet de la pile. La partie basse 16 bits de ce registre peut être utilisée comme le registre SP.
– Le registre EBP (Frame Base Pointer) contient un déplacement correspondant à une position dans la pile. Ce registre sert à pointer sur une donnée dans la pile. La partie basse 16 bits de ce registre peut être utilisée comme le registre BP.
– Les registres DS, ES, FS, GS (Data Segment) Ces registres 16 bits contiennent les numéros de segment mémoire dans lesquels sont stockées des données. Le registre DS est utilisé pour le segment de données du code ; les autres permettent de référencer d'autres types de données. On ne peut pas accéder directement à ce registre.

A.2.4 Registres d'états et de contrôle

– Le registre d'état EFLAG : Les bits 1, 3, 5, 15 et les bits de 22 à 31 sont réservés par le microprocesseur pour son fonctionnement interne. Les 21 autres bits forment des indicateurs binaires qui sont modifiés en fonction des calculs internes effectués par le processeur (drapeau de dépassement, drapeau zéro, drapeau signe...).
– Les registres de contrôle CR0 à CR4 déterminent le mode de fonctionnement du processeur et les caractéristiques de la tâche en cours d'exécution.
– Les registres de gestion mémoire, GDTR, IDTR, task register, et LDTR spécifient la localisation des structures de données utilisées en mode protégé.
– Les registres de débuggage DR0 à DR7 contrôlent et permettent de surveiller les opérations de débuggage du processeur.

A.2.5 Mode protégé du 80x86 (d'après [Bouillaguet,])

Le mode protégé de la série processeurs 80x86 existe depuis le 286.

Adressage mémoire en mode protégé

En mode réel, une adresse logique, c'est à dire telle qu'elle est vue par le programmeur, est formée d'un offset et d'un segment. Néanmoins, l'ensemble Offset + Segment n'est pas une adresse physique.

Une adresse logique n'est pas directement exploitable par le processeur : la mémoire vive est adressée par des adresses physiques. Ce sont des adresses physiques et non pas logiques qui sont envoyées par le processeur sur son bus d'adressage. Les programmes manipulent des adresses logiques, et le processeur les convertit en adresses physiques pour pouvoir les exploiter. En mode réel, la méthode de calcul de l'adresse physique par le processeur est relativement simple : $Adresse = (Segment * 16) + Offset$. Elle est composée du registre de segment décalé de 4 bits vers la gauche à laquelle on additionne l'adresse d'offset. De cela, il résulte plusieurs choses :

- Une adresse physique peut correspondre à plusieurs adresses logiques.
- Par exemple : 2736h : 0050h et 2700h : 3B0h correspondent à la même adresse physique.
- Une adresse logique ayant un offset égal à 0 commencera forcement à une adresse physique multiple de 16.

En mode protégé, une adresse logique est composée d'un sélecteur et d'un offset, et non plus d'une adresse de segment et d'un offset. L'offset a toujours la même signification qu'en mode réel, sauf qu'il est exprimé sur plus de 16 bits (24 sur 286 et 32 à partir du 386, bientôt 64 sur de nouveaux processeurs...). A partir du i386 et jusqu'à aujourd'hui, cela limite la taille des segments à 4 Go. Le sélecteur est lui par contre toujours noté sur 16 bits. Cependant, il ne référence plus une portion particulière de la mémoire, mais il indique le numéro d'un descripteur de segment dans une table. Un descripteur de segment est une structure de données de 8 octets qui décrit physiquement un segment : c'est la que sont stockés l'adresse physique de son "début" (l'adresse physique qui correspond à l'offset 0 de ce segment), sa taille et ses attributs. C'est au système d'exploitation qu'il appartient de remplir correctement les descripteurs de segments. Puisqu'en mode protégé on peut indiquer l'adresse physique du début d'un segment, appelée aussi adresse de base du segment, un sélecteur peut faire référence à un segment se trouvant à n'importe quel endroit de la mémoire. Tous les descripteurs sont réunis dans une seule et même zone de la mémoire, appelée Table des Descripteurs Globaux (en abrégé : GDT). Elle peut-être située n'importe où dans la mémoire vive, tant que le processeur sait où elle se trouve... Pour calculer une adresse physique à partir d'une adresse logique, on additionne l'adresse de base du segment, et l'offset : $Adresse = Descripteur[selecteur].Base + Offset$. Il ne faut pas confondre sélecteur et segment : le segment est la portion de mémoire décrite au processeur par un descripteur de segment, et le sélecteur est en quelque sorte le "numéro" du descripteur dans la table des descripteurs globaux. Concrètement, les sélecteurs sont chargés dans les registres de segment tout comme le sont les adresses de segment en mode réel. Les descripteurs de segment La structure des descripteurs contient les informations suivantes : Certaines valeurs sont, pour des raisons historiques éclatées en plusieurs morceaux non contigus. Pour représenter ceci, les "morceaux" d'une même valeur sont de la même couleur, et les nombres entre crochet indiquent quels bits de la valeur sont stockés dans le champ. Par exemple, Limite [0..15] indique que les bits 0 à 15 (donc en fait le premier Word) de la valeur Limite sont stockés à cet endroit.

- La Limite qui est l'offset maximal qui peut être utilisé avec le segment.
- La Base qui représente l'adresse Physique du segment, c'est à dire l'emplacement physique de l'offset 0 de ce segment.
- Le champ G qui indique la Granularité de la limite. Si il est mis à 1, alors la limite doit être interprétée en tant que Pages de 4Ko chacune. Si il est à 0, alors la limite est exprimée en octets.
- Le champ P définit la présence du segment en mémoire physique. Si le segment entier a été écrit sur le disque et vidé de la mémoire vive (cf. chapitre sur la mémoire virtuelle...), une interruption est déclenchée automatiquement lorsqu'il est accédé, pour permettre au système d'exploitation de le recharger en mémoire. C'est le seul mécanisme de mémoire virtuelle disponible sur le 286. Son principe est simple : on "swappe" des segment entiers sur le disque, et on les recharge selon les besoins. L'inconvénient apparaît très vite dès que la taille des segments augmente, ou quand seulement certaines parties du segment sont utiles.
- DPL est un champ de 2 bits qui correspond au niveau de privilège du descripteur :
- Un bit spécial est mis à 1 si le descripteur référence un segment de code ou de données, par opposition

aux segments spéciaux, notamment les portes ou les descripteurs de tâche pour lesquels ce bit est mis à 0.
- Le bit A est mis à 1 lorsqu'un sélecteur pointant sur ce descripteur est chargé dans un registre de segment. Cela permet de savoir si un segment à été accédé.
- Le Type est un champ de 3 bits. L'interprétation qu'on peux avoir des deux derniers bits dépend du 1er.
  - ST est le Type de Segment. C'est lui qui conditionne le reste. Si il vaut 1, alors c'est un segment de code, sinon c'est un segment de données. en effets, les paramètres des segments de code et de données ne sont pas les mêmes. La différence entre les segments de code et de données est que seuls les segments de code sont exécutables. Il est impossible d'exécuter des instructions à l'intérieur d'un segment de données.
  - Pour les champs propres aux segments de données :
    - Le bit E modifie la façon dont est interprété le champ Limite. En effet, si il est à 1, la limite sera un offset minimal. Cette faculté n'est vraiment utile que dans le cas d'un segment de pile (chaque PUSH diminue SP, donc l'offset dans le segment de pile), où un offset trop bas "déborderait" sur d'autres portions de la mémoire.
    - Le bit B n'est interprété que si E = 1. En effet, dans ce cas là, le champ limite indique l'offset minimal. L'offset maximal, quant à lui est de 64Ko si $B = 0$, 4Go sinon.
    - Le champ W indique la possibilité de modifier le contenu du segment de quelque manière que ce soit. Si un programme tente t'écrire dans un segment protégé contre l'écriture, une exception sera déclenchée.
  - Les champs propres aux segments de code sont :
    - Le bit D indique la taille par défaut des opérandes dans le segment : soit la taille d'une opérande est de 8 bits, soit c'est celle par défaut du segment courant.
    - C indique que le segment est "conforming".
    - R Indique qu'il est possible de lire le contenu du segment, par quelque moyen que ce soit. Si ce bit est mis à 0, il est impossible de lire le contenu du segment, mais il est possible de l'exécuter. Ceci sert donc garantir la confidentialité au code.

La table des descripteurs globaux peut être placée n'importe où dans la mémoire, tant que le processeur connaît son adresse physique (en effet, une adresse logique ne lui suffirait pas, vu que c'est grâce à elle qu'il calcule les adresses physiques). Dans le cas d'un petit programme indépendant on la place généralement dans le segment de données. Pour un système d'exploitation, par contre, il est situé à un endroit précis de la mémoire connu défini par le système lui-même (cf. implémentation Linux). Le premier descripteur (index 0) est obligatoirement un descripteur nul (descripteur dont tous les champs sont à 0). Son chargement dans un registre de segment ne causera pas d'erreur, mais dès l'accès à ce segment, une violation de protection surviendra, et une exception sera déclenchée. Les descripteurs doivent êtres placés de façon contiguë dans la mémoire. On peut considérer cet suite de descripteurs comme un tableau d'enregistrement représentant chacun un descripteur. Il peut y en avoir 8192 au maximum (la raison de cette limite se trouve dans le format des sélecteurs). Pour indiquer au processeur l'emplacement en mémoire de la GDT, on se sert d'une instruction spécifique : LGDT (Load Global Descriptor Table). Elle s'utilise de la seule et unique manière suivante : LGDT FWORD PTR GDT. Où FWORD PTR signifie que l'opérande qui le suit est une référence mémoire de six octets de long. GDT est en effet une variable de 6 octets composée de la façon suivante :

- Les 2 premiers octets contiennent la longueur (en octet) de cette table.
- Les 4 derniers octets contiennent l'adresse Physique de cette table en RAM.

La GDT cohabite avec une autre table, la Table des Descripteurs Locaux (LDT). Sa structure est absolument identique à celle de la GDT, à ceci près qu'il peut y en avoir plusieurs simultanément dans le système, mais dans des tâches distinctes. Le format des sélecteurs chargés dans les registres de segment est le suivant :

- L'index est le numéro du descripteur dans la table. Un index de 2 correspond au 2ème descripteur

dans la GDT, et il est donc placé à l'offset 16 (en octet) dans cette table.
– TI Indique à quelle table se rapporte l'index. 0 signifie la table des descripteurs globaux, 1 la table des descripteurs locaux de la tâche courante.
– RPL est un champ de 2 bits, qui identifie le niveau de privilège du demandeur.

Commutation en mode protégé

Pour parvenir au mode protégé, il est nécessaire de manipuler le registre CR0 du processeur. Si bit son bit de poids faible est à 1, le processeur est alors en mode protégé, sinon, il est en mode réel. Si on lit ce registre on pourra savoir en quel mode se trouve le processeur. Par contre, si on l'écrit, on fixe le mode dans lequel se trouve le processeur. Pour commuter en mode protégé, il suffit donc de faire :

```
MOV EAX,CR0
OR AL, 1
MOV CR0, EAX
```

Dès l'exécution de ces 3 instructions, le processeur fonctionne en mode protégé. Par ailleurs, à ce moment précis, CS contient une adresse de SEGMENT (car on est encore en mode réel). Or il devrait contenir le SÉLECTEUR du segment de code. Pour remédier ce problème on utilise une petite astuce, qui se code sous cette forme :

```
DB 0EAh DW \$+4, Selecteur_Code
```

Ce code est en fait un JMP FAR 16 bits (écrit en hexadécimal) vers l'instruction suivante (l'instruction JMP FAR se code en effet de la façon suivante : 0EAh — WORD PTR Offset du saut — WORD PTR Segment du Saut). Le registre IP est chargé avec l'offset de l'instruction immédiatement contiguë au saut, et CS avec le sélecteur du segment de code. Pourquoi est-ce que ça n'a pas planté dès la commutation en mode protégé ? Comment le processeur peut-il lire le saut sensé rétablir la situation, puisque CS contient déjà une valeur aberrante ? En fait, l'unité de décodage d'instruction du processeur dispose d'un petit cache (de l'ordre de 16 ou 32 octets...) dans lequel il stocke les instructions à l'avance. Au moment de la commutation, le saut est déjà dans ce buffer (parfois appelé prefetch queue), et donc n'a pas besoin d'être lu à CS :IP. Mais il faut rétablir l'ordre normal après la commutation. Il faut bien sûr cependant que la table des descripteurs globaux ait été chargée de façon adéquate préalablement à la commutation. Pour revenir au mode réel, il suffit d'effectuer le parcours inverse :

```
MOV EAX, CR0
AND AL, NOT 1
MOV CR0, EAX
DB 0EAh DW \$+4, Mon_Code
```

Ici, Mon_Code est le nom du segment de code. L'assembleur met ici la valeur du segment dans lequel est chargé le programme. Il faut noter que le problème du descripteur est le même que celui du segment : CS doit avoir un contenu approprié au mode dans lequel le processeur se trouve.

Protection

Dans tout système multi-tâche les programmes ont des droits limités. Le noyau de l'OS est le maître absolu et est chargé de "maintenir l'ordre" dans le système. Les principaux buts des mécanismes de protection sont :

– Éviter qu'un programme ne provoque un plantage.
– Éviter qu'un programme ne pousse le code du système à planter.

– Éviter qu'un programme exécute une action sur le matériel qui puisse engendrer un plantage.

Pour cela il faut que les programmes soient dans l'incapacité d'effectuer une action "néfaste". Ils sont à cet effet légèrement bridés. S'ils dépassent les limites que le système leur a fixé, ils sont terminés sans ménagement. Un programme fonctionnant correctement n'a aucune raison de dépasser ces limites. Si cependant cela se produit, il s'agit sans doute d'un bug qui met en péril la stabilité du système, ou bien d'un programme mal intentionné qui tenterait délibérément de faire planter le système. Les zones de mémoire sont protégées : les programmes ne peuvent pas accéder aux aires du système. Il est donc impossible qu'un programme écrive de façon anarchique dans une zone où le système stocke des données importantes. Cependant, il faut que le système puisse avoir accès à l'aire des programmes, ne serait-ce que pour qu'il remplisse son rôle. Un exemple : la fonction de lecture d'un bloc de données à partir d'un fichier doit stocker les octets dans une zone de données du programme qui l'a appelée. Le système doit donc avoir accès à l'aire des programmes. Le processeur distingue 4 niveaux de privilège :

– Le mode le plus privilégié est le mode 0. Il est réservé au coeur du système : gestion du multi-tâche, de la mémoire virtuelle, chargement des exécutables en mémoire... C'est le seul niveau habilité à exécuter certaines instructions critiques, telles que le chargement de tables des descripteurs.
– Le mode 1 et 2 sont réservés aux pilotes de périphériques ainsi qu'à certaines bibliothèques du système. Ils ne sont pas utilisés sous Linux.
– Le niveau 3 est le niveau de tout les programmes non-systèmes. C'est le moins privilégié.

L'intérêt de cette structure est qu'un programme exécuté à un certain niveau de privilège ne peut pas accéder aux données des niveaux plus privilégiés. Ceci exclue donc pour un virus, qui est un programme non système, donc de niveau de privilège 3, d'aller se greffer dans un partie du code du système d'exploitation, de privilège 0. Un programme ne peut pas non plus appeler directement du code plus privilégié que lui [1]. Cela est cependant possible, mais avec un certain contrôle. C'est le cas par exemple sous Linux pour les appels systèmes. Les privilèges sont gérés au niveau des segments. Chaque Descripteur contient un champ appelé DPL qui signifie Descriptor Privilege Level, soit "Niveau de Privilège du Descripteur". Cette valeur indique le niveau de privilège minimum nécessaire pour accéder au segment. Par exemple, si le niveau de privilège indiqué est 1, alors les segments de code de privilège 2 et 3 n'y auront pas accès. Le niveau de privilège courant (parfois appelé CPL, pour Current Privilege Level) est placé dans les 2 bits de poids faible du registre CS. Dans les sélecteurs placés dans CS ce champ s'appelle RPL. Dans le segment de code et aussi dans SS (le segment de pile), ce champ s'appelle le CPL. La présence de ces deux segments est obligatoire au fonctionnement de n'importe quelle séquence d'instruction. Chaque programme de n'importe que niveau de privilège, que ce soit le système, un pilote d'imprimante ou une application, possède son segment de code et son segment de pile. Puisque ces deux registres de segments pointent obligatoirement vers des segment du niveau de privilège du code actuellement exécuté, on considère que ce niveau de privilège est le niveau de privilège courant. Lorsqu'on tente de charger un sélecteur dans un registre de segment, le processeur vérifie si le segment auquel on tente d'accéder n'est pas plus privilégié que le segment de code courant. Pour faire cette comparaison, le processeur compare CPL au DPL du segment concerné. Si l'on tente de charger dans un registre un segment plus privilégié que le segment de code contenant cette instruction de chargement (CPL ¿ DPL), le processeur déclenchera une exception de protection générale. Une autre vérification entre aussi en compte : c'est le test du niveau de privilège du demandeur. Cela complexifie beaucoup le mécanisme de protection, mais le rend pratiquement infaillible. Le niveau de privilège du demandeur est en fait le RPL du sélecteur que l'on cherche à charger dans un registre de segment. C'est le niveau de privilège du segment de code qui a créé le sélecteur. Par exemple, un programme en langage évolué crée un pointeur vers un de ses segments de données. Tôt ou tard, il faudra charger ce pointeur, l'offset dans un registre d'adressage, le sélecteur dans un registre de segment. Le RPL du pointeur que l'on cherchera à charger correspondra au CPL du segment de code ayant créé le pointeur. Le processeur va le comparer au DPL du segment qu'on tente de charger (qui est placé dans

[1] Effectivement, si cela était possible sans restriction, il serait possible qu'un saut chaotique atterrisse au milieu du code de l'OS, avec des valeurs sans signification dans les registres, et fasse planter le tout.

le descripteur). Si il s'avère que le RPL est supérieur au DPL - donc en clair si un programme a créé un sélecteur pointant sur un segment trop privilégié pour lui - le processeur déclenchera une une exception de protection générale. Le test du DPL n'est-t-il pas suffisant. En effet, si une routine de niveau de privilège 2, crée un sélecteur pointant sur un segment de niveau de privilège 1. Si elle transmet ce sélecteur à une routine de niveau 0, la stabilité du système serait en danger, car la routine de niveau 0 a le droit, elle, d'écrire sur le segment de niveau 1, auquel la routine de niveau 2 n'avait pourtant pas accès. Avec le RPL, cette situation est interdite. En effet, le RPL du segment dont il est question serait de 2 - il a été créé par une routine de CPL 2. Dans la routine de niveau 0, le chargement de ce sélecteur dans un registre de segment provoquera une exception de protection générale, car il est interdit de charger un sélecteur dont le RPL est supérieur au DPL. Lorsqu'une erreur de protection a été générée, il suffit de terminer la tâche ayant causé l'erreur, même si elle a eu lieu dans le code du système, et tout retourne dans l'ordre... En résumé, si le créateur du sélecteur n'avait pas accès au segment qu'il référence, toute tentative de chargement de ce segment provoquera une erreur quel que soit le CPL courant. En fait, le CPL perd des privilège pour refléter le RPL lors des comparaisons. Le processeur compare le plus faible de CPL et RPL au DPL, pour évaluer si l'accès au segment est autorisé ou non. Ce mécanisme permet de repérer avec certitude le niveau de privilège original d'un segment transmis d'une procédure à une autre. Si CPL est moins privilégié que RPL, alors c'est une procédure plus privilégiée qui a créée un sélecteur et qui l'a passé à la procédure actuelle. Par exemple, ici, un pilote - par exemple - de niveau de privilège 2 créée un buffer, et renvoie un pointeur dessus à un programme de niveau 3. Dans le programme, Le RPL du segment en question sera 2, alors que le CPL sera 3. Il faudra que le DPL du segment concerné soit de 3, sinon le programme ne pourra pas le charger dans un registre de segment. L'inverse peut aussi avoir lieu. Une procédure de niveau 3 créée un descripteur et le passe à une routine plus privilégiée, de niveau 2. Dans ce cas, le RPL sera de 3 (rappelons que le RPL indique le niveau de privilège du segment qui a créé le sélecteur), tandis que le CPL sera de 2. Dans un cas comme dans l'autre, on peut être sûr que le plus faible de CPL ou de RPL a été en contact direct avec le segment. C'est ce mécanisme que le processeur applique : lors de la tentative de chargement d'un sélecteur dans un registre de segment, il compare le plus faible de CPL ou de RPL au DPL du descripteur du segment. Il reste un dernier cas à évoquer : celui des segments de code dociles. Un segment de code docile est un segment dont le niveau de privilège s'adapte à celui qui lui a transféré le contrôle. Par exemple, une routine chargée d'effectuer des calculs trigonométriques. Elle doit pouvoir être appelée par les programmes de niveau 3, mais aussi éventuellement par le système. On peut donc la placer dans un segment docile. Si une routine du système, de niveau 1 par exemple, fait un saut vers ce segment, il sera exécuté au niveau de privilège 1. Si par contre il est appelé par un programme de niveau 3, il sera exécuté au niveau 3, indépendamment de son DPL. Une fois effectuée la vérification des privilèges, le processeur vérifie que le segment que l'on charge dans un registre lui correspond. En effet, il paraît peu vraisemblable de charger SS avec un segment de code. Cela arrive rarement, et ne peut être dû qu'à un bug. Le processeur est donc immunisé contre ce genre de problèmes. Il se base sur les informations fournies dans le descripteurs. Voici la liste des types autorisés pour chaque registre de segment :

	DS,ES,FS et GS	SS	CS
Lecture seule	*		
Lecture/Écriture	*	*	
Exécution seule			*
Exécution/lecture	*		*

Tout chargement qui ne satisfait pas ces règles déclenche l'habituelle exception de protection générale. Le chargement avec un sélecteur nul ne déclenche pas d'exception immédiatement, mais par contre, il est impossible d'accéder à la mémoire à travers lui. Il est intéressant de remarquer que les segments de code peuvent être exécutables mais pas lisibles, ce qui garanti une certaine confidentialité au code. Il est possible de protéger des segments contre l'écriture, si par exemple ils pointent vers des zones de données sensibles mais auxquelles on a besoin d'accéder. Il reste une dernière vérification à effectuer lors de l'accès aux données : la vérification de la limite. Elle est stockée dans le descripteur. Ce

contrôle n'intervient que lors des accès à la mémoire. L'offset de l'accès est comparé avec la limite. S'il la dépasse, c'est à dire si il est plus grand (sauf dans le cas de segments de piles à croissance vers le bas), l'habituelle exception de protection est déclenchée. Cette limite est là pour 2 raisons :

1. Pour empêcher qu'un programme déborde de son segment suite à une erreur et écrase un bout de la mémoire avec ses données, si par exemple, il accède à un tableau de données, et qu'il y a une erreur lors du calcul de l'index.
2. Pour éviter qu'un programme ne puisse lire ou écrire la totalité de la mémoire, et donc le code du système d'exploitation ou d'autres programmes.

Cette vérification est la plus simple, mais c'est peut-être la plus fondamentale du système de protection.

Gestion des interruptions

Un transfert de contrôle est :
– Soit c'est l'effet d'une instruction (CALL, JMP, INT, RET, IRET) qui modifie volontairement les registres CS et IP qui déterminent où sera lue la prochaine instruction à exécuter.
– Soit c'est une action du processeur sous l'action d'une interruption.
Une interruption peut survenir de trois façons différentes :

– Elle est déclenchée par le un périphérique matériel quelconque. On parle alors d'IRQs (IRQ signifie Interrupt ReQuest).
– Si ce périphérique est le processeur lui-même, on parle d'exceptions. Le processeur déclenche des interruptions pour signaler une erreur dans le code qu'il exécute, comme une violation de protection, ou une instruction non valide.
– Elle peut aussi être déclenchée par une instruction INT (interruption logicielle).

Lorsqu'un interruption est appelée par le matériel, le contrôle doit être transmis à une routine ad hoc. Mais il faut connaître l'adresse de cette routine en mémoire, pour pouvoir l'appeler. Cette adresse dépend du numéro de l'interruption, et est stockée dans un tableau en mémoire : la Table des Descripteurs d'Interruption (abrégé en IDT). C'est une liste de descripteurs, tout comme la GDT, à ceci prêt qu'elle ne contient pas des descripteurs de segment, mais des portes d'interruption : c'est un descripteur, mais qui indique une adresse précise en mémoire. Par exemple, en mode protégé, l'instruction `INT 21H` va sauter à l'adresse lue dans la 33ème porte de l'IDT (car 21h = 33 en décimal). L'IDT est en fait l'équivalent, en mode protégé, de la table des vecteurs d'interruptions du mode réel, située à l'adresse 0000h : 0000h. Un descripteur de porte d'interruption est une structure de 8 octets, comparable à un descripteur de segment :

– L'Offset est l'offset où sera transféré le contrôle. C'est la valeur qui sera chargée dans le registre EIP.
– Le Selecteur indique le descripteur du segment où le contrôle sera transféré (chargé dans CS).
– P renseigne sur la présence de ce segment en mémoire (1= il y est, 0 = il n'y est pas).
– DPL indique le niveau de privilège de la porte. C'est lui qui définit quel sera le niveau de privilège maximal (3 = le moins privilégié) qui aura le droit d'accéder à cette porte.
– le bit compris entre le DPL et le Type indique si le présent descripteur est un descripteur de segment (1 = c'est un descripteur de segment, 0 = c'est un descripteur d'autre chose).
– Le Type correspond au type de porte. Il existe des portes d'appel, d'interruption, de trappe et de tâche, avec des versions 16 et 32 bits.

En mode protégé, le principe est le même que pour la table des descripteurs : il y a une instruction spéciale pour initialiser l'IDT.

```
LIDT FWORD PTR IDT_buff
```

Où IDT_buff est une variable de 6 octets, exactement comme pour la GDT. Il existe quatre types de portes :

1. Les portes d'appel.
2. Les portes d'interruption.
3. Les portes de trappe.
4. Les portes de tâche.

Lorsqu'une interruption survient (une IRQ, par exemple), le contrôle est passé à une routine chargée de prendre les mesures appropriées. Ce transfert de contrôle peut se faire via une porte d'interruption ou via une porte de trappe (et même par une porte de tâche). Il y a cependant une petite nuance : à travers une porte d'interruption, le Flag est sauvegardé sur la pile, en même temps que l'adresse de retour (il se passe la même chose en mode réel). Mais dans le vrai Flag, le drapeau IF est mis à 0 - c'est exactement ce que fait l'instruction CLI (IF = 0 : interruptions matérielles désactivées, IF = 1 : interruptions matérielles activées). Le fait qu'IF soit à 0 n'empêche cependant ni les exceptions de survenir, ni l'instruction INT de fonctionner correctement. Le fait de mettre IF à 0 pendant les routines d'interruption a pour conséquence d'empêcher que la présente routine d'interruption (qui s'occupe peut-être de la lecture de données à partir du disque dur.) ne soit interrompue fâcheusement par une autre IRQ quelconque. Lorsque le IRET survient, il recharge le flag avec la copie qui était sur la pile, et donc restaure le drapeau IF dans son état antérieur. Avec une porte de trappe, le flag est quand même sauvegardé sur la pile, mais l'état du drapeau IF reste inchangé, ce qui a pour conséquence qu'une routine d'interruption appelée via une porte de trappe pourra être interrompue par une autre interruption matérielle (l'IRQ Timer est appelée au moins 18,2 fois par seconde, donc il n'est pas du tout improbable qu'elle puisse survenir pendant une autre routine d'interruption). Évidement, le PIC - Programmable Interrupt Controler - est là pour éviter un tel mic-mac. Voici maintenant la liste des valeurs que peur prendre le champ Type en fonction du type de porte :

Type				Description
0	0	0	0	Réservé
0	1	0	0	Porte d'appel 16 bits
0	1	1	0	Porte d'interruption 16 bits
0	1	1	1	Porte de trappe 16 bits
1	1	0	0	Porte d'appel 32 bits
1	1	1	0	Porte d'interruption 32 bits
1	1	1	1	Porte de trappe 32 bits

Il existe des portes d'interruptions 16 bits et d'autres 32 bits. En effet, l'instruction INT sauvegarde automatiquement l'adresse de retour sur la pile. Toutes les portes d'interruption placent sur la pile le sélecteur du segment en cours d'exécution au moment de l'interruption. Cependant, les portes 16 bits sauvegardent un offset sur 16 bits, tandis que les portes 32 bits sauvegardent un offset sur 32 bits. Une porte d'interruption 16 bits appelée dans un segment de code 32 bits peut faire planter le système : si l'offset dans le segment de code 32 bits est supérieur à 65535, il sera tronqué à son mot de poids faible lorsqu'il sera sauvegardé sur la pile par un porte 16 bits. Lorsque l'instruction IRET surviendra, elle dépilera un offset incorrect, avec toutes les conséquences que cela peut avoir. Une interruption ayant une porte 32 bits appelée dans un segment 16 bits ne posera aucun problème, mais comme on l'a vu, l'inverse n'est pas vrai. C'est une des raisons qui tend à éviter le mélange de code 16 et 32 bits. Les routines d'interruptions doivent être accédées par des portes 32 bits : ainsi, si une interruption est déclenchée, une adresse de retour 32 bits sera sauvée. Les portes permettent le changement du niveau de privilège courant et c'est l'une de leurs raisons d'être. Imaginons par exemple que le contrôleur du disque dur génère une interruption indiquant qu'il a terminé la lecture des données. Le programme en cours d'exécution est interrompu, et le contrôle est transféré à une routine du système, qui est chargée de prendre les mesures ad hoc. Il y a donc un changement du niveau de privilège, puisque le

programme interrompu est de niveau 3 et que le pilote du disque dur est de niveau 0, 1 ou 2. Le pilote de disque dur va s'exécuter tranquillement, puis, quand il aura fini, il fera un IRET, qui reviendra au programme interrompu, et donc retournera au niveau de privilège 3. Le transfert de contrôle déclenche une erreur de protection générale si le DPL du segment appelé est numériquement supérieur, donc moins privilégié que le CPL du programme interrompu. D'autre part, s'il y a un changement de niveau de privilège, il y a aussi une commutation de pile. Imaginons que la pile soit presque pleine au moment où notre IRQ arrive. Le pilote de disque dur, s'il écrit beaucoup sur la pile, pourra causer une faute de pile, même s'il est très bien programmé. Pour éviter cela, le processeur charge à partir du segment d'état de tâche (ou TSS) de la tâche courante, les "coordonnées" (SS et ESP) d'un autre segment de pile, pour la durée de l'interruption. Au IRET, les paramètres de l'ancien segment de pile (qui étaient stockés avec l'adresse de retour sur la pile du nouveau) sont rechargés. On confond parfois l'IRQ et l'interruption qui lui est associée, alors qu'en fait, les IRQs sont des signaux qui transitent par le bus et qui indiquent au processeur qu'un composant matériel a quelque chose à dire. En fait, les IRQs ne transitent pas directement des composants matériels au processeur, mais passent par le contrôleur PIC (Programmable Interrupt Controler). Ce (ou ces) contrôleurs sont chargés, non pas de vérifier la validité de votre billet, mais :

1. de vérifier qu'il n'y a pas "collision" d'IRQ. Une IRQ est mise en attente si elle survient pendant l'exécution de la routine d'interruption d'une autre IRQ.
2. de générer le numéro d'interruption correspondant à une IRQ précise. En effet, bien que les IRQ soient numérotés de 0 à 15, elles correspondent à des numéros d'interruption complètement différents.

Le(s) PIC(s) peuvent de plus :

– Indiquer si la routine d'interruption d'une IRQs est en cours d'exécution.
– Interdire à certaines IRQs de déclencher des interruptions.
– Changer les numéros d'interruptions déclenchées pour des IRQs données.

En effet, par défaut les IRQs sont assignées aux numéros d'interruption suivants :
– Les IRQs de 0 à 7 correspondent respectivement aux interruptions 8 à 0Fh (elles sont gérées par le premier PIC)
– Les IRQs de 8 à 15 correspondent respectivement aux interruptions 70h à 78h (elles sont gérées par le second PIC)

Lorsque l'IRQ 0 surviendra, si elle est autorisée, elle déclenchera l'interruption 8 (ce phénomène a d'ailleurs lieu 18,2 fois par seconde puisque l'IRQ 0 est l'interruption TIMER). De même pour l'IRQ 13 et l'interruption 75h. Le processeur n'accordera aucune suite aux IRQ si le bit IF (comme Interrupt enable Flag) du Flag est à 0. Ce bit est contrôlé par les instruction CLI (qui le met à 0) et STI (qui le met à 1). Ces instructions sont indispensables pour éviter qu'une interruption survienne pendant une phase délicate, par exemple la commutation en mode protégé. A moins qu'elles ne soient ainsi bloquées, les IRQs peuvent survenir à n'importe quel moment, selon l'état du matériel. Donc des interruptions peuvent être déclenchées n'importe quand. Il faut donc qu'en permanence, tout soit prêt pour les accueillir : il faut une IDT convenablement construite, avec des portes adaptées aux interruptions susceptibles d'être déclenchées, et il faut aussi qu'à tout instant la pile soit valide puisqu'une interruption y stocke l'adresse de retour. En mode protégé, valide signifie : qui a un descripteur correct, qui a un sélecteur correct, qui n'est pas en train de déborder et qui est au bon niveau de privilège. Dans le cas contraire, et si tout est convenablement prévu, une commutation de tâche est déclenchée, vers une tâche (du système d'exploitation) ayant un segment de pile valable, et chargée de tout remettre en l'ordre. Si une telle commutation de tâche n'est pas prévue le système plante. Une exception est la réponse normale, documentée et prévisible du processeur à une instruction incorrecte. La raison fondamentale de leur existence est la "récupération" des erreurs des programmes par le système d'exploitation. Tout bon OS en mode protégé installe ses gestionnaires d'exception pour empêcher qu'un programme buggé fasse planter l'ensemble du système. Les exceptions sont au

nombre de 17, une par type d'erreur, et représentent les interruptions de 0 à 11h. Une exception est déclenchée lorsque le cas d'erreur qu'elle représente survient. Pour certaines d'entre elles, le processeur place sur la pile un code d'erreur, après le flag et l'adresse de retour. Ce code indique de façon plus précise la raison de l'erreur. N° Nom Description Code erreur Adresse de retour 0 Division par 0 appelée par DIV ou IDIV si on tente une division par 0 non. EAX et EDX contiennent toujours la valeur qu'ils avaient avant la division l'instruction fautive 1 Trace Appelée automatiquement après chaque instruction si le bit TF du Flag est à 1. Sert pour le debbugage non l'instruction suivante (pour continuer l'execution de manière transparente) 2 Interruption non masquable (NMI) Ceci n'est pas une exception à proprement parler, mais un signal indiquant une défaillance grave du matériel (mémoire defectueuse, ou autres bétises du genre...) non sur l'instruction suivante 3 Point d'arrêt Ce n'est pas vraiment non plus une exception, mais elle est déclenchée par l'instruction de point d'arrêt INT 3. Utilisé pour le debbugage non l'instruction suivante (pour continuer l'execution de manière transparente) 4 Overflow Déchenchée si le bit OF du flag est à 1 pendant l'exécution de l'instruction INTO. Voir la documantation de cette instruction non l'instruction suivante 5 BOUND hors limite Déclenchée par l'instruction BOUND en cas de dépassement de la limite. Voir la documentation de cette instruction non l'instruction suivante 6 Instruction invalide Le processeur ne parvient pas à identifier la prochaine instruction non l'instruction fautive 7 FPU (coprocesseur mathématique) non disponible Declenchée par une instruction déstinée au FPU. Deux cas sont possibles : * Le bit EM (EMulation de FPU) du registre CR0 est à 1. L'OS doit simuler la présence d'un FPU. * Les bits MP (Moniteur de Coprocesseur) et TS (Task Switched) de CR0 sont à 1. L'OS doit gérer le partage du FPU dans un environnement multi-tâche. Non sur L'instruction fautive (pour la simuler dans le premier cas, ou la relancer après avoir fait le nécessaire dans le second cas) 8 Double faute Une exception a elle-même causé une exception. Par exemple, l'exception de pile invalide a causé une exceptionde pile invalide. Je pense que cette interruption doit être une porte de tâche, pour fournir un contexte d'exécution valide oui sur l'instruction fautive 9 Debordement de segment du FPU ? Cette interruption n'est pas générés par les processeurs plus récents que le i386. ? ? 0Ah TSS (Task State Segment) invalide Les circonstances pouvant entrainer ce cas de figure seront étudiées dans la page sur les commutations de tâche. oui l'instruction fautive 0Bh Segment non présent déclenché si on charge dans un registre de segment un selécteur dont le bit P du descripteur est à 0 oui l'instruction fautive 0Ch Faute de pile Erreur liée au segment de pile : dépassement de la limite ou commutation de tâche avec un sgement de pile non-présent oui l'instruction fautive 0Dh Erreur de protection générale Quasiment toutes les violations de protections oui l'instruction fautive 0Eh Faute de page voir page sur la mémoire vitruelle oui (format spécial) l'instruction fautive 0Fh Réservé 10h Erreur de FPU LE coprocesseur rencontre une erreur de traitement numérique non sur l'instruction fautive 11H Erreur d'alignement (à partir du i486) Les bit AM (Alignment Mask) de CR0 et AC(Alignment Check) du Flag sont à 1, le CPL est 3, et on a tenté de faire un accès à la mémoire non aligné (par exemple lire un Word à une adresse impaire ou un Double-Word à une adresse qui n'est pas un multiple de 4) oui (toujours 0) l'instruction fautive Quelques nouvelles exceptions ont été ajoutées sur les Pentium, Pentium II et Pentium III, mais elle ne sont pas d'une importance vitale, et la documentation Intel est très exhaustive (voir sur la partie developer de leur site). Par contre, elle doivent être prévues, il faut donc que leurs entrés respectives dans l'IDT pointent vers une routine générale chargée de prendre des mesures... Certaines exceptions placent par ailleurs un code erreur sur la pile. Ce code a la structure suivante est un Double-Word, dont les 16 bits de poids fort sont inutilisés. Les 3 bits de poids faibles sont des champs et ont une importance dans l'interpretationde l'index. Réservé Index TI IDT EXT Le champ Index est le numéro du descripteur ayant "causé" l'erreur. Cependant, est-il dans la GDT, dans la LDT, dans l'IDT... * si TI = 0, alors l'index pointe dans la GDT, sinon, il pointe dans la LDT. * si IDT = 0, alors la table est déterminée par le bit TI. Sinon, il pointe sur un descripteur de porte présent dans l'IDT. * si EXT = 0, alors l'exception est causée par le programme interrompu, sinon elle a été causée par un évenement exterieur (par exemple, une IRQ a déclenché une interurption dont la porte était invalide... Ce n'est pas de la "faute" du programme en cous d'execution). Ce code d'erreur sert à identifier le segment ayant causé l'erreur, pour envisager de la réparer. Par exemple, si on tente d'adresser un

segment au-delà de sa limite, une erreur de protection générale sera déclenchée, et le code erreur fera référence au segment en question. Un problème se pose : les IRQ 0 à 8 correspondent respectivement aux interruptions 8 à 0Fh, par défaut en mode réel... Or en mode protégé, les interruptions déclenchés par les IRQs tombent sur les mêmes numéros que ceux des exceptions : Timer (IRQ 0) déclenchera une double faute, et une frappe sur le clavier (IRQ 1) un débordement de segment de coprocesseur.

Pour éviter cela, deux solutions sont possibles :

1. A l'aide du PIC, changer les numéros d'interruption associés aux IRQs, de façons à ce qu'elles ne perturbent pas les exceptions.
2. Dans le gestionnaire d'exceptions, on peut tester grâce au PIC si, par hasard, une IRQ n'est pas active en ce moment. Dans ce cas, on appelle la procédure d'interruption.

A.3 Résumé assembleur 80x86

- AND destination,masque : Applique un "et" à destination par masque
- CALL adresse : Appelle une procédure qui est à l'adresse adresse
- CMP a, b : Compare les deux variables a et b. Toujours suivit d'un saut conditionnel.
- CMPS[B/D/W] : Compare l'octet/le mot/le double-mot DS :ESI à ES :EDI.
- IN destination,port : Lit une valeur 8 bits sur le port port (16 bits) et la stocke dans destination. Le seul registre autorisé pour port est DX.
- IRET valeur : Quitte une interruption.
- JMP offset : Saute à l'adresse offset.
- J[cas] offset : Saute à l'adresse offset si la condition [cas] est exacte. [cas] est une condition relative aux drapeaux.
  En non-signé
  - JA : est supérieur ($a > b$), si CF=ZF=0.
  - JAE ou JNB ou JNC : est supérieur ou égal ($a => b$), si CF=0.
  - JB ou JC : est inférieur ($a < b$), si CF=1.
  - JBE : est inférieur ou égal ($a <= b$), si CF=ZF=1.

  En signé
  - JG : est supérieur ($a > b$), si SF=ZF=0.
  - JGE : est supérieur ou égal ($a => b$), si SF=OF.
  - JL : est inférieur ($a < b$), si $SF \neq OF$.
  - JLE : est inférieur ou égal ($a <= b$), si $SF \neq OF$ et ZF=1.
  - JE ou JZ : est égal ($a = b$), si ZF = 1.
  - JNE ou JNZ : est différent ($a <> b$), si ZF = 0.
- LEA destination,source : Ecrit l'adresse de source dans destination.
- LDS destination,adresse : Copie l'adresse adresse en 32 bits dans le registre DS, son segment, et dans destination (16 bits), son offset.
- LES destination,adresse : Copie l'adresse adresse en 32 bits dans le registre ES, son segment, et dans destination (16 bits), son offset.
- LFS destination,adresse : Copie l'adresse adresse en 32 bits dans le registre FS, son segment, et dans destination (16 bits), son offset.
- LGS destination,adresse : Copie l'adresse adresse en 32 bits dans le registre GS, son segment, et dans destination (16 bits), son offset.
- LSS destination,adresse : Copie l'adresse adresse en 32 bits dans le registre SS, son segment, et dans destination (16 bits), son offset.
- LODS[B/D/W] : Copie l'octet/le mot/le double-mot ES :EDI dans AL/AX/EAX (instruction inverse de STOS[B/W/D]).
- MOV dst,src : Copie la valeur src dans dst.

- MOVS[B/D/W] : Copie l'octet/le mot/le double-mot DS :ESI dans ES :EDI.
- MOVZX dst,src : Etend à 32 bits le nombre contenu dans src (8 bits) et transfert le résultat dans dst (16 ou 32 bits).
- MUL source : Multiplie la destination explicite par source, les deux nombres sont considérés comme non signés.
- NOT destination : Inverse les bits de destination.
- OR destination, masque : Applique un "OU logique" à destination par masque.
- OUT source,port : Ecrit la valeur source (8 bits) sur le port port (16 bits). Le seul registre autorisé pour port est DX.
- PUSH valeur : Met une [valeur] dans la pile.
- POP registre : Sort une valeur de la pile et la stocke dans un [registre].
- REP instruction : Répète l'instruction [instuction] ECX fois.
- RET valeur : Quitte la procédure en cours.
- SHL registre,valeur : Décalage binaire du registre de valeur vers la gauche (L = Left), les bits apparaissant à droite sont complétés par des zéros.
- SHR registre,valeur : Décalage binaire du registre de valeur vers la droite (R = Right), les bits apparaissant à gauche sont complétés par des zéros.
- STOS[B/D/W] : Copie AL/AX/EAX dans l'octet/le mot/le double-mot ES :EDI (inverse de LODS[B/W/D]).
- SCAS[B/D/W] : Compare AL/AX/EAX à l'octet/le mot/le double-mot ES :EDI (permet de rechercher une valeur dans une chaine de caractères).
- TEST source,masque : Teste si les bits masque de source sont posés ou non, et modifie ZF en conséquence (ZF posé si les bits de source sont posés, sinon ZF=0). L'instruction permet de tester un bit particulier de source.
- XOR destination,masque : Applique un "ou exclusif" à destination par masque.

A.4 Description sommaire de l'assembleur en ligne du C

Une instruction assembleur du C se construit de la manière suivante :

```
asm ( assembler template
    : output operands                   /* optional */
    : input operands                    /* optional */
    : list of clobbered registers       /* optional */
    );
```

- "assembler template" représente les instructions d'assemblage à exécuter.
- "output operands" indique les sorties modifiées par le code d'assemblage
- "input operands" indique les entrées ayant un impact sur le code d'assemblage
- "list of clobbered registers" indique les registres pouvant avoir été modifiés par le code d'assemblage.

Exemple :

```
int a=10, b;
asm ("movl %1, %%eax;
        movl %%eax, %0;"
    :"=r"(b)        /* output */
    :"r"(a)         /* input */
    :"%eax"         /* clobbered register */
    );
```

Le code d'assemblage déplace le contenu de %1 dans EAX (`movl`). Puis déplace le contenu de EAX dans %0. %0 correspond à la variable de sortie ("=") b qui est associée à un registre quelconque ("r" – cette directive est appelée un "contrainte"). %1, quant à lui correspond à un registre quelconque qui contiendra la valeur de a. Le quatrième argument indique que le registre EAX est modifié par le code d'assemblage.

Les autres contraintes possibles sont :
– a,b,c,d : respectivement eax,ebx,ecx ou edx
– S : esi
– D : edi
– m : adresse dans une zone de mémoire
– i : un entier
– g : un registre, une zone mémoire ou un entier
– r : un registre
– q : aex,bex,cex ou dex
– f : un registre flottant
Il existe par ailleurs 2 modificateurs de contraintes :
– = : opérande en écriture (comme vu dans l'exemple)
– & : opérande modifiée préccocément lors de l'exécution du code d'assemblage

A.5 Directives GNU Assembleur (GAS)

– .align : alignement en mémoire de données (même si leurs types sont différents)
– .ascii "chaîne" : permet de rentrer une "chaîne" de caractères (sans le NULL)
– .asciz "chaîne" : identique à .ascii mais le caractère NULL (fin de chaîne est inséré à la fin)
– .bss : début de la zone de réservation (bss)
– .byte un,plusieurs,plusieurs : permet de déclarer un ou plusieurs octets et de les initialiser
– .data : début de la zone de donnée
– .end : fin du code source
– .equ symbole,expression : initialise la constante défini par le symbole à la valeur défini par expression
– .extern symbole : spécifie que le symbole est défini dans un autre module
– .fill nombre, taille, valeur : remplit nombre copies d'une certaine taille ayant une certaine valeur
– .global symbole : le symbol est global (visible dans tous les modules)
– .hword un,plusieurs,plusieurs : permet de déclarer un ou plusieurs mots de 16 bits et de les initialiser
– .include fichier : insert le contenu du fichier spécifié dans le fichier source
– .set symbole,expression : initialise la constante défini par le symbole à la valeur défini par expression
– .skip expression : reserve expression octets non initialisés
– .text : début de la zone de texte
– .word un,plusieurs,plusieurs : permet de déclarer un ou plusieurs mots de 32 bits et de les initialiser

Annexe B

Segmentation de la mémoire - Process Address Space

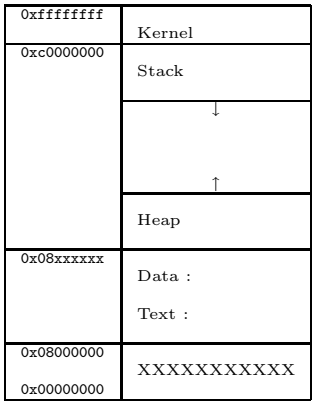
David PICARD picard@ensea.fr

Ce document est un exemple d'évolution de la mémoire vu par le programme au cours de son exécution. Pour faciliter la vie des programmeurs, les systèmes d'exploitation modernes masquent l'accès à la mémoire physique aux programmes qu'ils exécutent. Chaque programme évolue dans un espace d'adressage virtuel possédant des propriétés remarquable. Le lien entre cet espace d'adressage et la mémoire physique est totalement pris en charge par le système d'exploitation. Cet espace d'adressage virtuel est appelé "process address space" en anglais. L'exemple présenté dans ce document est valable pour linux sur une architecture x86 32 bits. Le fonctionnement de la mémoire est très fortement dépendant du système d'exploitation et de l'architecture de la machine (les tailles et positions des différentes zones peuvent varier), néanmoins, le principe général est globalement le même.

B.1 Structure de la mémoire

Généralement, l'espace d'adressage du processus possède les propriétés suivantes :
- il commence à l'adresse 0.
- il est continu.
- il est privatif (les autres programmes n'y ont pas d'accès).

Le fait d'avoir un espace d'adressage continu et privatif permet de faire des opérations sur les adresses (arithmétique des pointeurs) de manière très simple, comme par exemple balayer un tableau simplement en incrémentant une adresse. la mémoire est décomposée en segments de manière à séparer diverses zone comme suit :



la zone Kernel est la zone mémoire où se trouve le noyau du système d'exploitation, le programme n'y a pas accès. Le code du programme se trouve dans la zone Text, et les variables déclarées dans le programme sont dans la zone Data. La zone Heap (tas) est la zone dans laquelle est réservée la mémoire lors d'une allocation (`malloc()`). Elle croît dans le sens des adresses positives lorsque des allocations de mémoire sont faites. La zone stack (pile) est l'endroit dans lequel on stocke des copies des variables lorsque l'on appelle une nouvelle fonction. Elle croît dans le sens des adresses négatives lorsque l'on empile des données. La zone XXXXXXX est réservée et non utilisable.

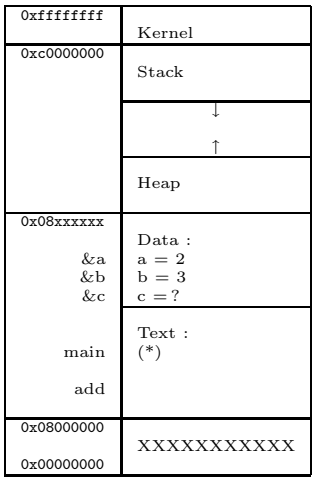
Concrètement, l'adressage utilisable par le programme va de `0x08000000` à `0Xc0000000`.

B.2 Exemple de programme.

Voyons un exemple de code :

```
#include <stdlib.h>

int main()
{
        int a = 2;
        int b = 3;
        int * c;

        c = malloc(sizeof(int));

        add(a, b, c);

        free(c);

        return c;
}

void add(int arg1, int arg2, int * arg3)
{
        *arg3 = arg1 + arg2;
        return;
}
```

Et une représentation de la mémoire qui va avec :

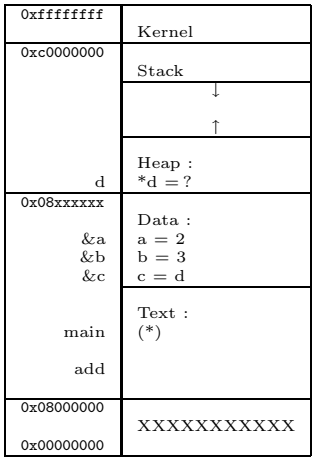


le marqueur (*) représentera dans la suite du document le bloc d'instructions auquel se trouve le programme au moment où on représente la mémoire.

La zone Text contient le code du programme divisé en deux parties : les fonctions `main()` et `add()`. Dans le bloc de mémoire contenant les instructions de `main()`, les instructions sont disposées de manière continues, c'est à dire que pour avancer dans l'exécution du programme, il suffit d'incrémenter l'adresse de l'instruction courante. Chaque fonction est ainsi disposée dans un bloc d'instructions continues. Le programme est découpé dans les blocs d'instructions continues des fonctions qu'il contient. La zone Data quant à elle contient a, b et c avec leur valeur par défaut. c, n'ayant pas été initialisé, peut contenir n'importe quoi (*ie.* une adresse non-valide).

B.3 Allocation de mémoire.

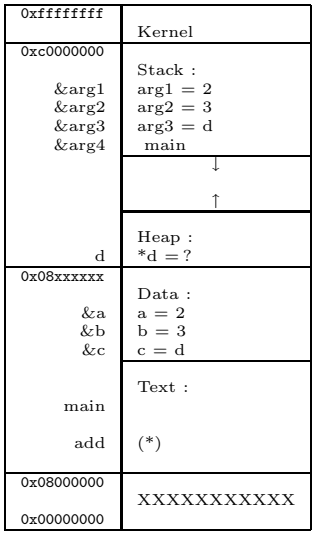
Lors de son éxécution, le programme va appeler la fonction `malloc()`, qui va allouer un espace en mémoire de la taille d'un int. L'adresse de cet espace est stocké dans c. Après l'appel à `malloc()` la mémoire ressemble à ce qui suit :



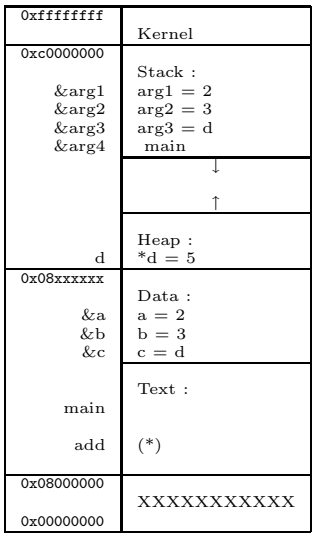
*d a été réservé dans le tas, à l'adresse d. La taille du tas a donc augmenté de `sizeof(int)`, et on peut ranger des données de la taille d'un `int` à l'adresse d. c pointe maintenant vers une zone de mémoire utilisable, par contre, cette zone (*d) n'a pas été initialisée et peut contenir n'importe quoi.

B.4 Appel de fonction.

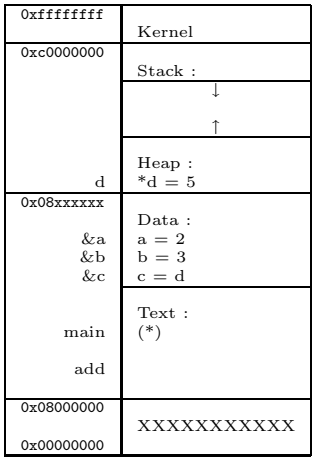
Le programme appelle ensuite la fonction `add()`. Pour cela, on copie en pile (Stack) l'addresse à laquelle il faudra revenir après l'éxécution de la fonction (c'est à dire l'adresse de l'instruction qui se trouve juste après l'instruction qui opère le saut vers `add()`), ainsi que les arguments passés, ce qui organise la mémoire comme suit :



Les instructions qui sont éxécutées sont celle qui se trouve dans le bloc de mémoire à l'adresse add. `add()` fait son addition en travaillant sur les copies trouvées dans la pile. Elle remplit donc *arg3 = *d avec le résultat de arg1 + arg2.

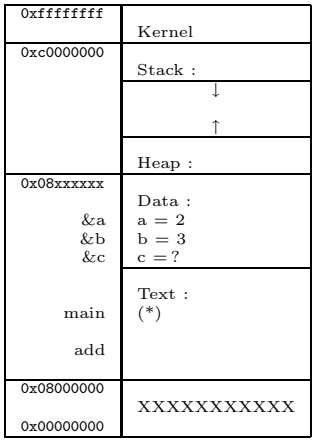


Puis on retourne dans le main à l'adresse que l'on avait stocké en pile. La pile est vidée de ce que l'on avait mis dedans :



B.5 Libération de mémoire.

Enfin, le programme appelle `free()`, ce qui libère le tas (heap) de l'espace alloué en d :



On notera que c contient toujours l'ancienne adresse d. Cependant, cette adresse n'est plus utilisable, car elle ne pointe pas vers un espace alloué.

Annexe C

Travaux pratiques

C.1 Processus, signaux et fichiers

1. Créer un processus fils qui s'endort pendant 10 secondes et rend la main à son père.
   - Quel est le status retourné quand tout se passe normalement ?
   - Quel est le status retourné quand vous tuez le fils prématurément par le biais d'un signal de votre choix ?
   - Observer ce qui se passe lorsque le père ne se met pas en attente sur le fils après sa mort ?
   - Que se passe-t-il lorsque c'est le père qui est tué prématurément ?
2. Générer par une boucle n processus issus du même père. Mettre le père en attente de tous les fils. Afficher les status des fils au fur et à mesure de leur disparition. Les fils font un `sleep(2*n+1)` avec des valeurs de n différentes.
3. Générer un nombre de processus illimité. Existe-t-il une limite ?
4. Créer un programme générant un fils et demandez lui de devenir leader en utilisant `setpgrp()`. Créer un fils à partir de ce processus. Tuez en une fois les processus fils.
5. Créer un programme capable de lancer différentes commandes unix, par menu ou directement en donnant la chaîne de caractères correspondante.
6. Un père et son fils veulent écrire dans le même fichier.
   - Ecrire un programme permettant l'exclusion mutuelle par la pose d'un verrou externe.
   - Modifier le programme pour utiliser un verrou posé par la commande lockf (mode bloquant ou non-bloquant).

C.2 Communication par pipe entre deux processus

C.2.1 Principe

Un processus met en place un fils et le pipe qui servira de couloir de communication entre eux puis envoie à son fils, par l'intermédiaire du pipe, les caractères lus sur `stdin`. Quand le fichier `stdin` est fermé (End Of File – `CTRL-D` en mode console), le père envoie le signal `SIGUSR1` au fils pour lui indiquer la fin du traitement et attend la mort de son fils.

C.2.2 Rôle du fils

Il doit lire les caractères présents dans le pipe et les convertir en majuscules avant de les afficher sur `stdout`. Il mettra en place le traitement sur le signal `SIGUSR1` qui lui indiquera que le père a fini d'envoyer des caractères et devra consommer tous les caractères présents dans le pipe puis envoyer un acquittement sur `stdout` avant de se terminer.

C.2.3 Notes

- Le programme de traitement du fils sera mis en place par un exec.
- Le programme doit pouvoir fonctionner avec un redirection de `stdin`
  - `prog_pere < fichier`
- Pour convertir les caractères en majuscules, utiliser `toupper()`. Inclure <`ctype.h`>.
- Pour lire des chaînes sur `stdin`, vous pouvez utiliser la fonction `fgets(char *s, int size, FILE *stream)` qui retourne la chaîne lue depuis le fichier `stream` dans `s` (jusqu'au `LF` non compris ou jusqu'à la lecture de `size` caractères) ou retourne `0` sur la rencontre de fin de fichier (`CTRL-D`).

C.3 Attention

Les appels `read` et `write` sur un pipe sont bloquant par défaut et se terminent si le pipe est fermé à l'autre extrémité. Il retourne le nombre de caractères, respectivement, lus ou écrits sinon.

Penser à tester les cas d'erreurs (valeur `-1` retournée par les appels systèmes).

C.4 Mise en place d'un mécanisme client-serveur avec communication par segment mémoire partagé protégé par sémaphores

Dans le cadre du TP, seul le processus client est à réaliser.

C.4.1 Principe

Le processus client fournit au serveur des tableaux de valeurs aléatoires dont il calcule la moyenne. Le processus client va faire un certain nombre de requêtes auprès du serveur et entrer en concurrence avec d'autres processus pour accéder à la ressource commune, ici, le segment mémoire.

Pour gérer les conflits d'accès, 3 sémaphores sont mis en place :
- `seg_dispo` : protège l'accès au segment mémoire. L'acquisition de ce sémaphore indique que l'on peut utiliser le segment.
- `seg_init` : L'acquisition de ce sémaphore par le client indique au serveur que le segment est initialisé.
- `res_ok` : L'acquisition de ce sémaphore par le serveur indique au client que le résultat est prêt.

Afin de tester la cohérence du fonctionnement, le client calculera sa propre moyenne pour la comparer avec celle fournie par le serveur. De plus, le client transmettra son numéro de tty et un numéro de requête lors de chaque demande et vérifiera lorsqu'il récupérera le résultat qu'il correspond bien à la requête envoyée.

C.4.2 Dialogue

Le dialogue entre les 2 processus est le suivant :

client		serveur	
0	récupère les identifiant du segment, des sémaphores et initialise le générateur.	0	Crée le segment mémoire. Crée et initialise les sémaphores.
1	demande à acquérir seg_dispo	1	attend que seg_init passe à 0
2	initialise segment et acquiert `seg_init`	2	calcule résultat et acquiert `res_ok`
3	attend que res_ok passe à 0	3	essaie d'acquérir `seg_init` (indique résultat lu)
4	lit résultat et libère `seg_init`	4	libère res_ok puis `seg_init`
5	libère `seg_dispo`	5	boucle sur 1
6	affiche résultats		
7	boucle sur 1		

Un certain nombres de procédures réunies dans la librairie `libseg.a` sont disponibles pour réaliser le programme :

`init_rand()` : initialise le générateur
`long getrand()` : retourne un entier long signé
`int ntty()` : retourne un entier correspondant au numéro du tty

Les routines suivantes bloquent le processus en attendant d'aboutir. semid désigne l'identifiant de l'ensemble de sémaphores :

`wait_sem(semid, sem)` : attend que le sémaphore sem passe à 0
`acq_sem(semid,sem)` : tente d'acquérir le sémaphore sem
`lib_sem(semid,sem)` : libère le sémaphore sem

La structure du segment sera définie ainsi :

```
struct shmseg
{
        int tty;                /* Numero de tty */
        int req;                /* numero de requete */
        long tab[maxval];       /* Tableau de valeurs */
        long result;            /* resultat */
}
```

Cette structure est définie sous le type SEGMENT dans le fichier segdef.h. Ce fichier réunit tous les includes nécessaires ainsi que la définition des constantes suivantes :

`cle`	(`key_t`)3	clé d'accès commune au segment et aux sémaphores
`seg_dispo`	0	sémaphore accès segment
`seg_init`	1	sémaphore segment initialisé
`res_ok`	2	sémaphore résultat prêt
`maxval`	100	nombre de valeurs à calculer
`segsize`		taille du type SEGMENT

L'application sera conçue à partir de 4 fichiers :

- `libseg.a` : bibliothèque pré-compilée
- `segdef.h` : fichier de déclaration à inclure
- `xxxxx.c` : votre programme client

Le fichier client devra déclarer une structure associée à la manipulation des sémaphores : `struct sembuf sop;`

L'organisation du programme sera la suivante :

1. *Initialisations (faire une fonction séparée)*
   - *récupérer les identifiants segment et sémaphore avec détection d'erreurs*
   - *attacher le segment avec détection d'erreurs*
   - *initialiser le générateur*
2. *Boucle contrôlée par le compteur de requête*
   - *demander à acquérir* ***seg_disop***
   - *initialiser le segment et calculer le résultat local*
   - *demander à acquérir* ***seg_init***
   - *attendre* ***res_ok***
   - *libérer résultat et* ***seg_init***
   - *libérer* ***seg_dispo***
   - *afficher résultats comparés client et serveur*
3. *détacher segment mémoire et exit*

C.4.3 Travail à effectuer

1. Créer le makefile de votre application
2. Réaliser le client selon le modèle ci-dessus
3. écrire vos propres routines de manipulation des sémaphores et remplacer le module semgest.o par le votre.

C.5 Mise en place d'un mécanisme client-serveur utilisant des threads

Le but est d'implanter sous forme de threads
le fonctionnement client/serveur mis en place au TP2

C.5.1 Principe

Vous devez gérez le dialogue entre un ou plusieurs threads producteurs de nombres aléatoires (qui écrivent dans un segment de mémoire partagée) et UN consommateur qui calcule la moyenne de ces nombres.

Réfléchissez bien aux mutex et variables conditions dont vous avez besoin pour assurer la validité des données !

C.5.2 Comparaison

Comparez les temps de calculs avec ceux obtenus dans l'implantation avec IPC.

C.6 Mise en place d'une messagerie

C.6.1 Principe

Le but du TP est de créer un gestionnaire de messagerie utilisant les queues de messages.

Le rôle du gestionnaire est double : il doit d'une part se mettre en attente des messages qui lui sont adressés (utilisation du numéro de tty) et d'autre part, pouvoir envoyer des messages à d'autres terminaux.

C.6.2 Implantation

Vous redéfinirez la structure `msgp` de la manière suivante :

```
struct msgb {
        long mtype;
        long tty;
        char mtext[80];
        } msgp;
```

Où `mtype` est le numéro de tty du destinataire, `tty` est celui de l'émetteur et `mtext` est le message envoyé.

A priori, le gestionnaire est en attente de message. Lorsque l'utilisateur veut envoyer un message il fait un `CTRL-C` (correspond à l'envoi du signal `SIGINT`). Le gestionnaire devra capter ce signal et mettre en place l'envoi du message.

Annexe D

Bibliographie

Bibliographie

[Arcomano, 2003] Arcomano, R. (2003). Kernelanalysis-howto. http ://tldp.org/ HOWTO/ KernelAnalysis-HOWTO.html.

[Blaess, 2000] Blaess, C. (2000). *Programmation Système En C Sous Linux - Signaux, Processus, Threads, Ipc Et Sockets*. Eyrolles.

[Bouillaguet,] Bouillaguet. Le mode protégé. http ://charles.moostik.net/ pmode/.

[Bovet and Cesati, 2005] Bovet, D. P. and Cesati, M. (2005). *Understanding the Linux Kernel*. O'Reilly, 3 edition.

[Card,] Card, R. Le système de fichiers ext2. http ://www.april.org/ actions/ confs/ 19980624/ ext2fs.ag.ps.

[Cegielski, 2004] Cegielski, P. (2004). *Conception des systèmes d'exploitation : Le cas Linux*. Collection Noire. Eyrolles.

[Collectif, 2008] Collectif (2008). Documentation du noyau linux. file :///usr/src/linux-VERSION/Documentation.

[Corbet et al., 2005] Corbet, J., Rubini, A., and Kroah-Hartman, G. (2005). *Linux Device Drivers*. O'Reilly, 3 edition.

[Cornavin, 2005] Cornavin, A. N. D. J. (2005). Explorer procfs. Gazette Linux n°115 - http ://ftp.traduc.org/doc-vf/gazette-linux/html/2005/115/lg115-E.html. Article publié sous Open Publication License.

[Daudel, 2005] Daudel, O. (2005). */proc et /sys*. O'Reilly.

[D.Picard, 2006] D.Picard (2006). Segmentation de la mémoire - process address space. http ://www-etis.ensea.fr/ Members/ dpicard/ info_vm.

[Drepper and Molnar, 2002] Drepper, U. and Molnar, I. (2002). The native posix thread library for linux.

[Ferre, 2000] Ferre, N. (2000). Les linux temps réel. http ://nferre.free.fr/ emlnx/ rapport/ node9.html.

[Ficheux, 2006a] Ficheux, P. (2006a). Programmation noyau sous linux, partie 1 : Api des modules linux. *GNU Linux Magazine France*, (88) :60–66.

[Ficheux, 2006b] Ficheux, P. (2006b). Programmation noyau sous linux, pilotes en mode caractère. *GNU Linux Magazine France*, (89) :76–83.

[Ficheux, 2007] Ficheux, P. (2007). Programmation noyau sous linux, partie 3 : techniques avancées. *GNU Linux Magazine France*, (93) :92–98.

[Ficheux, 2008] Ficheux, P. (2008). Programmation noyau sous linux, partie 4 : les pilotes usb. *GNU Linux Magazine France*, (101) :90–98.

[Goodheart and Cox, 1994a] Goodheart, B. and Cox, J. (1994a). *The magic garden explained*. Prentice Hall, Australia.

[Goodheart and Cox, 1994b] Goodheart, B. and Cox, J. (1994b). *The Magic Garden Explained : The Internals of UNIX System V Release 4*. Prentice Hall, Upper Saddle River, NJ.

[Intel,] Intel. Intel 64 and ia-32 architectures software developer's manuals. http ://www.intel.com/ products/ processor/ manuals/.

[Ke-Fong, 2008] Ke-Fong, L. (2008). Programmation de modules noyau linux. *Linux Developer's Journal*, 5(1) :60–67.

[Kumar, 2008] Kumar, A. (2008). Multiprocessing with the completely fair scheduler. http ://www.ibm.com/ developerworks/ linux/ library/ l-cfs/ index.html. Existe en PDF.

[Lacombe, 2007a] Lacombe, E. (2007a). La refonte de l'ordonnanceur de tâches. *GNU Linux Magazine France*, (95) :14–16.

[Lacombe, 2007b] Lacombe, E. (2007b). Un nouvel ordonnanceur de tâches pour linux. *GNU Linux Magazine France*, (97) :4–11.

[Lefranc, 2004] Lefranc, S. (2004). Les améliorations du noyau linux 2.6. http ://www.chear.defense.gouv.fr/ fr/ think_tank/ archives/ rstd/ 64/ rstd64p77.pdf.

[Nayani et al., 2002] Nayani, A., Gorman, M., and de Castro, R. S. (2002). Memory management in linux desktop companion to the linux source code. http ://www.ecsl.cs.sunysb.edu/ elibrary/ linux/ mm/ mm.pdf.

[Northrup, 1997] Northrup, C. (1997). *Programmer avec les threads UNIX*. International Thompson Publishing France, Paris.

[Philipp, 1991] Philipp, J. (1991). *UNIX : Les mécanismes internes*. Presse de l'école nationale des ponts et chaussées, Paris.

[Philipp, 1994] Philipp, J. (1994). *L'administration sous UNIX*. Presse de l'école nationale des ponts et chaussées, Paris.

[Poirier,] Poirier, D. The second extended file system : Internal layout. http ://www.nongnu.org/ ext2-doc/ ext2.html.

[Pélissier, 1996] Pélissier, C. (1996). *UNIX : Utilisation Administration système et réseau*. Traité des Nouvelles Technologies : Série Informatique. HERMES, seconde edition.

[Salzman, 2008] Salzman, P. J. (2008). The linux kernel module programming guide. http ://tldp.org/ LDP/ lkmpg/ 2.6/ html/.

[Sandeep,] Sandeep, S. Gcc inline assembly "how to". http ://www.ibiblio.org/ gferg/ ldp/ GCC-Inline-Assembly-HOWTO.html.

[sur l'initiative du Secrétariat d'Etat à l'industrie, 2008] sur l'initiative du Secrétariat d'Etat à l'industrie, I. (2008). Les différents modèles économiques du logiciel libre. http ://www.telecom.gouv.fr/ fonds_documentaire/ rapports/ logicielslibres.pdf.

[Tanenbaum, 2003] Tanenbaum, A. (2003). *Systèmes d'exploitation*. Informatique. Pearson Education.

[Vieillefond, 1992] Vieillefond, C. (1992). *Mise en oeuvre du 80386*. Sybex.

[Wikipedia, 2008a] Wikipedia (2008a). Entrées-sorties. http ://fr.wikipedia.org/ wiki/ Entrées-sorties.

[Wikipedia, 2008b] Wikipedia (2008b). Licence publique générale gnu. http ://fr.wikipedia.org/ wiki/ Licence_publique_générale_GNU.

[Wikipedia, 2008c] Wikipedia (2008c). Licence publique générale limitée gnu. http ://fr.wikipedia.org/ wiki/ Licence_publique_générale_limitée_GNU.

[Wikipedia, 2008d] Wikipedia (2008d). Noyau linux. http ://fr.wikipedia.org/ wiki/ Noyau_Linux.

[Wikipedia, 2008e] Wikipedia (2008e). Sysfs. http ://fr.wikipedia.org/ wiki/ Sysfs.

Annexe E

Index

Index