Structure interne du noyau Linux 2.4

Adaptation française du *Linux Kernel 2.4 Internals*

Tigran Aivazian <tigran CHEZ veritas POINT com>

Adaptation française: Yaël Gomez

<ygomez CHEZ yosins POINT net>

Relecture de la version française: Claire Boussard

<clboussard CHEZ free POINT fr>

Version 20020807.fr.1.0

13 mai 2003

Historique des versions

Version 20020807.fr.1.0	2003-05-03 Adaptation française.	YG
Version 20020807	2002-08-07 (29 av 6001) Version originale du 7 août 2002.	TA

Une introduction au noyau Linux 2.4. Ce document a été réalisé comme support de cours donnés en interne par l'auteur chez VERITAS Software Ltd. Celui-ci travaille en tant qu'ingénieur senior Noyau Linux au sein de cette société.

Table des matières

1. Introduction

1.1. Dernière version de ce document

La source de la dernière version originale (en anglais) de ce guide peut être téléchargée depuis http://www.moses.uklinux.net/patches/lki.sgml. Il est également possible de la lire en ligne à http://www.moses.uklinux.net/patches/lki.html.

Ce guide en version originale est également diffusé par le Projet de documentation Linux (LDP) [http://www.tldp.org]. Il est disponible via ce projet sous différents formats depuis http://www.tldp.org/guides.html.

L'adaptation française de ce guide a été réalisée dans le cadre du projet traduc.org [http://www.traduc.org]. La dernière version française de ce document est disponible à http://www.traduc.org/docs/guides/lecture/lki/. N'hésitez pas à faire parvenir tout commentaire relatif à cette version à commentaires CHEZ traduc POINT org [mailto:commentaires CHEZ traduc POINT org?subject=lki].

1.2. Droits d'utilisation

1.3. Remerciements

Remerciements à :

- Juan J. Quintela <quintela CHEZ fi POINT udc POINT es>,
- Francis Galiegue <fg CHEZ mandrakesoft POINT com>,

- Hakjun Mun <juniorm CHEZ orgio POINT net>,
- Matt Kraai <kraai CHEZ alumni POINT carnegiemellon POINT edu>,
- Nicholas Dronen <ndronen CHEZ frii POINT com>,
- Samuel S. Chessman <chessman CHEZ tux POINT org>,
- Nadeem Hasan <nhasan CHEZ nadmm POINT com>,
- Michael Svetlik <m.svetlik CHEZ ssi TIRET schaefer TIRET peem POINT com> pour les diverses corrections et suggestions.
- Le chapitre Linux Page Cache a été écrit par : Christoph Hellwig <hch CHEZ caldera POINT de>.
- Le chapitre Mécanismes IPC a été écrit par : Russell Weight <weightr CHEZ us POINT ibm POINT com> et Mingming Cao <mcao CHEZ us POINT ibm POINT com>.

1.4. Adaptation française

L'adaptation française de ce document a été réalisée par Yaël Gomez <ygomez CHEZ yosins POINT net>. La relecture de ce document a été réalisée par Claire Boussard <clboussard CHEZ free POINT fr>. La publication de ce document a été préparée par Jean-Philippe Guérard <jean TIRET philippe POINT guerard CHEZ laposte POINT net>.

2. Amorçage (Booting)

2.1. Construire l'image du noyau Linux

Ce paragraphe décrit les étapes de la compilation d'un noyau Linux et les messages renvoyés à chaque étape. Le processus de construction du noyau dépend de l'architecture, c'est pourquoi je voudrais souligner que l'on ne considérera ici que la compilation d'un noyau Linux/x86.

Quand l'utilisateur tape « make zImage » ou « make bzimage », l'image amorçable du noyau qui en résulte est stockée respectivement en tant que `arch/i386/boot/zImage` ou `arch/i386/boot/bzImage`. Voici comment cette image est construite :

1. Les fichiers sources C et assembleur sont compilés au format objet relogeable (.o) ELF et certains d'entre eux sont regroupés logiquement dans des archives (.a) en utilisant *ar(1)*.
2. En utilisant *ld(1)*, les .o et .a ci-dessus sont liés pour donner `vmlinux` qui est un fichier exécutable ELF 32-bit LSB 80386 non strippé (les symboles n'ont pas été nettoyés), statiquement lié.
3. `System.map` est produit par *nm vmlinux*, les symboles inutiles sont retirés.
4. On entre dans le répertoire `arch/i386/boot`.
5. Le code assembleur du secteur d'amorçage `bootsect.S` est pré-traité avec ou sans *-D__BIG_KERNEL__* pour produire respectivement, selon que la cible est bzImage ou zImage, `bbootsect.s` ou `bootsect.s`.
6. `bbootsect.s` est assemblé et converti en un fichier « binaire brut » (*raw binary*) appelé `bbootsect` (ou `bootsect.s` assemblé et converti en binaire brut donnant `bootsect` pour zImage).
7. Le code Setup `setup.S` (`setup.S` inclut `video.S`) est pré-traité pour donner `bsetup.s` pour bzImage ou `setup.s` pour zImage. De la même façon que pour le code de bootsector, la différence réside en *-D__BIG_KERNEL__*, présent pour bzImage. Le résultat est converti en un

« binaire brut » appelé `bsetup`.

8. On entre dans le répertoire `arch/i386/boot/compressed` et convertit `/usr/src/linux/vmlinux` en $tmppiggy (nom de fichier temporaire) au format binaire brut, en retirant les sections ELF `.note` et `.comment`.

9. *gzip -9 < $tmppiggy > $tmppiggy.gz*

10. On lie $tmppiggy.gz au format ELF relogeable (*ld -r*) `piggy.o`.

11. On compile les fonctions de compression `head.S` et `misc.c` (toujours dans le répertoire `arch/i386/boot/compressed`) en objets ELF `head.o` et `misc.o`.

12. On lie ensemble `head.o`, `misc.o` et `piggy.o` pour obtenir `bvmlinux` (ou `vmlinux` pour zImage, attention à ne pas le confondre avec `/usr/src/linux/vmlinux`!). Notez la différence entre *-Ttext 0x1000* utilisé pour `vmlinux` et *-Ttext 0x100000* pour `bvmlinux`, i.e. pour bzImage compressé le chargeur (loader) est chargé en haut.

13. On convertit `bvmlinux` en un « binaire brut » `bvmlinux.out` en enlevant les sections ELF `.note` et `.comment`.

14. On revient dans le répertoire `arch/i386/boot` et, avec le programme *tools/build*, concatène `bbootsect`, `bsetup` et `compressed/bvmlinux.out` pour obtenir `bzImage` (supprimez le « b » du début pour `zImage`). Ce qui a pour effet d'écrire à la fin du secteur d'amorçage des variables importantes comme `setup_sects` et `root_dev`.

La taille d'un secteur d'amorçage est toujours de 512 octets. La taille de setup doit faire plus de 4 secteurs, mais pas plus de 12k — la règle est :

0x4000 octets >= 512 + setup_sects * 512 + la place pour la pile pendant l'exécution de bootsector/setup

On verra plus tard d'où vient cette restriction.

La taille maximum du bzImage produit à cette étape est d'à peu près 2,5M pour amorcer avec LILO et de 0xFFFF (0xFFFF0 = 1048560 octets) pour amorcer avec une image binaire, c.a.d depuis une disquette ou un cédérom (émulation El-Torito).

Remarquez qu'alors que *tools/build* contrôle la taille du secteur d'amorçage, de l'image du noyau et la limite inférieure de la taille de setup, il ne vérifie pas la taille maximum de setup. Dès lors il est facile de construire un noyau défectueux rien qu'en ajoutant quelques grands espaces (« .space ») à la fin de `setup.S`.

2.2. Démarrage : aperçu

Les détails du processus d'amorçage sont spécifiques à l'architecture, on va donc s'intéresser à l'architecture IBM PC/IA32. A cause de sa conception vieillissante et du besoin de garder une compatibilité ascendante, le microcode (firmware) des PC démarre le système de manière plutôt démodée. Ce processus peut être divisé en six étapes logiques :

1. Le BIOS choisit le périphérique d'amorçage.

2. Le BIOS charge le secteur d'amorçage depuis le périphérique d'amorçage.

3. L'exécution du secteur d'amorçage charge setup, les routines de décompression et l'image compressée du noyau.

4. Le noyau est décompressé en mode protégé.

5. Les initialisations de bas niveau sont réalisées en assembleur.

6. Les initialisations de haut niveau sont réalisées en C.

2.3. Démarrage : BIOS POST

1. L'alimentation démarre le générateur d'horloge et envoie le signal #POWERGOOD sur le bus.
2. La ligne CPU #RESET est positionnée (Le CPU est maintenant en mode réel).
3. %ds=%es=%fs=%gs=%ss=0, %cs=0xFFFF0000,%eip = 0x0000FFF0 (ROM BIOS POST code).
4. Toutes les vérifications POST sont exécutées avec les interruptions désactivées.
5. IVT (Interrupt Vector Table ou table des vecteurs d'interruption) est initialisée à l'adresse 0.
6. La fonction BIOS de chargement du code d'amorçage est invoquée via *int 0x19*, avec %dl contenant le périphérique d'amorçage « numéro du disque ». Elle charge la piste 0, secteur 1 à l'adresse physique 0x7C00 (0x07C0:0000).

2.4. Démarrage : secteur d'amorçage et lancement

Le secteur d'amorçage (bootsector) utilisé pour démarrer Linux peut être soit :

- Le secteur d'amorçage Linux (`arch/i386/boot/bootsect.S`),
- Le secteur d'amorçage de LILO (ou un autre chargeur d'amorçage), ou
- pas de secteur d'amorçage (loadlin, etc.)

On va s'intéresser ici en détail au secteur d'amorçage Linux. Les premières lignes initialisent des macros qui seront utilisées comme valeurs de segment :

```
29 SETUPSECS = 4               /* défaut pour le nombre de secteurs de lancement
30 BOOTSEG   = 0x07C0     /* adresse originelle du secteur d'amorçage */
31 INITSEG   = DEF_INITSEG  /* nous déplaçons l'amorçage ici — hors du chemin *
32 SETUPSEG  = DEF_SETUPSEG /* le lancement démarre ici */
33 SYSSEG    = DEF_SYSSEG   /* le système est chargé à 0x10000 (65536) */
34 SYSSIZE   = DEF_SYSSIZE  /* taille du système~: # en clicks de 16-octets */
```

(Les nombres sur la gauche sont les numéros de lignes du fichier bootsect.S). Les valeurs de `DEF_INITSEG`, `DEF_SETUPSEG`, `DEF_SYSSEG` et `DEF_SYSSIZE` sont tirées de `include/asm/boot.h` :

```
/* Ne changez rien ici, sauf si vous savez vraiment ce que vous faites. */
#define DEF_INITSEG     0x9000
#define DEF_SYSSEG      0x1000
#define DEF_SETUPSEG    0x9020
#define DEF_SYSSIZE     0x7F00
```

Maintenant, regardons le détail du code de `bootsect.S` :

```
54      movw    $BOOTSEG, %ax
55      movw    %ax, %ds
56      movw    $INITSEG, %ax
```

```
57      movw    %ax, %es
58      movw    $256, %cx
59      subw    %si, %si
60      subw    %di, %di
61      cld
62      rep
63      movsw
64      ljmp    $INITSEG, $go

65   # bde - 0xff00 changé en 0x4000 => permet l'accès débogueur > 0x6400(bde).
66   # Ce ne serait pas un soucis si nous avions testé le haut de la mémoire. La
67   # configuration BIOS peut aussi permettre de placer en mémoire haute les ta
68   # des disques wini plutôt que dans le tableau des vecteurs. Il est possible
69   # que l'ancienne pile aie corrompu le tableau des disques.

70 go: movw    $0x4000-12, %di  # 0x4000 est une valeur arbitraire >=
71                               # la longueur du secteur d'amorçage
72                               # + la longueur du setup + laplace pour
73                               # la pile ; 12 est la taille du disk parm.
74      movw    %ax, %ds         # ax et es contiennent déjà INITSEG
75      movw    %ax, %ss
76      movw    %di, %sp         # place la pile à INITSEG:0x4000-12.
```

Les lignes de 53 à 60 déplacent le code du secteur d'amorçage de l'adresse 0x7c00 à 0x9000. Ce qui est fait en :

1. positionnant %ds:%si sur $BOOTSEG:0 (0x7C0:0 = 0x7C00)
2. positionnant %es:%di sur $INITSEG:0 (0x9000:0 = 0x90000)
3. fixant le nombre de mots de 16 bits dans %cx (256 mots = 512 octets = 1 secteur)
4. réinitialisant le drapeau DF (direction) dans EFLAGS pour auto-incrémenter les adresses (cld)
5. avançant et copiant 512 octets (rep movsw)

Si ce code n'utilise pas `rep movsd`, c'est intentionnel (pensez à .code16).

La ligne 64 saute vers le label `go:` dans la nouvelle copie du secteur d'amorçage, soit dans le segment 0x9000. Ceci plus les trois instructions suivantes (lignes 64-76) prépare la pile à $INITSEG:0x4000-0xC, i.e. %ss = $INITSEG (0x9000) et %sp = 0x3FF4 (0x4000-0xC). C'est de là que vient la limite sur la taille de setup mentionnée plus haut (voir Construire l'image du noyau Linux).

Les lignes 77-103 corrigent la table des paramètres du disque afin de permettre la lecture multi-secteurs :

```
77 # La table des paramètres de disque par défaut de nombreux BIOS ne
78 # permet pas la lecture multi-secteurs au-delà du numéro de secteur
79 # maximum spécifié dans le tableau des paramètres par défaut de la
80 # disquette, cela peut signifier 7 secteurs dans certains cas.
81 #
82 # Lire les secteurs un à un est lent et donc hors de question,
83 # nous remédions à cela en créant une table en RAM avec de nouveaux
84 # paramètres (pour le 1er disque). Nous mettons le nb max. de secteurs
85 # à 36 - nous ne rencontrerons pas plus pour un ED 2.88.
86 #
87 # Une valeur trop haute ne nuit pas, une trop basse, si.
88 #
89 # Les segments sont comme suit~: ds = es = ss = cs - INITSEG,
90 # fs = 0, et gs est inutilisé.

91      movw    %cx, %fs        # met fs à 0
```

```
92      movw    $0x78, %bx      # fs:bx est l'adresse du tableau des paramètres
93      pushw   %ds
94      ldsw    %fs:(%bx), %si  # ds:si est la source
95      movb    $6, %cl         # copie 12 octets
96      pushw   %di             # di = 0x4000-12.
97      rep                     # pas besoin de cld -> c'est fait
98      movsw                   # à la ligne 66
99      popw    %di
100     popw    %ds
101     movb    $36, 0x4(%di)   # corrige le compte des secteurs
102     movw    %di, %fs:(%bx)
103     movw    %es, %fs:2(%bx)
```

Le contrôleur de disquette est réinitialisé en utilisant la fonction 0 du service BIOS int 0x13 et les secteurs de setup sont chargés juste après le secteur d'amorçage, i.e. à l'adresse physique 0x90200 ($INITSEG:0x200), en utilisant encore le service BIOS int 0x13, fonction 2 (lire le(s) secteur(s)). Ça se passe aux lignes 107-124 :

```
107 load_setup:
108     xorb %ah, %ah         # reset FDC
109     xorb %dl, %dl
110     int         $0x13
111     xorw %dx, %dx           # lecteur 0, tête 0
112     movb $0x02, %cl         # secteur 2, piste 0
113     movw $0x0200, %bx       # adresse = 512, dans INITSEG
114     movb $0x02, %ah         # service 2, « lire secteur(s) »
115     movb setup_sects, %al   # (suppose que tout est tête 0, piste 0)
116     int $0x13               # lire
117     jnc ok_load_setup       # ok – continuons

118     pushw %ax                 # sort un code d'erreur
119     call print_nl
120     movw %sp, %bp
121     call print_hex
122     popw %ax
123     jmp load_setup

124  ok_load_setup:
```

Si le chargement échoue pour quelque raison que ce soit (la disquette est abîmée ou quelqu'un a retiré la disquette pendant le chargement), on affiche un code d'erreur et on réessaie de lire dans une boucle infinie. Le seul moyen d'en sortir est de réamorcer la machine, à moins que l'une des tentatives réussisse, mais cela a peu de chances d'arriver (si quelque chose merde, ça ne peut qu'empirer).

Si le chargement des secteurs du code de lancement setup_sect réussit, on saute au label `ok_load_setup~:`.

On procède alors au chargement de l'image compressée du noyau à l'adresse physique 0x10000. Ainsi on préserve les zones de données du microcode en mémoire basse (0-64k). Une fois que le noyau est chargé, on saute en $SETUPSEG:0 (`arch/i386/boot/setup.S`). Lorsqu'on n'a plus besoin des données (i.e. plus d'appel au BIOS), elles sont écrasées en déplaçant le noyau complet (compressé) de 0x10000 vers 0x1000 (ce sont des adresses physiques bien sûr). C'est fait par `setup.S` qui met les choses en place pour le mode protégé et saute en 0x1000 qui est le début du noyau compressé, i.e. `arch/386/boot/compressed/{head.S,misc.c}`. Ceci configure la pile puis on appelle `decompress_kernel()` qui décompresse le noyau a l'adresse 0x10000 et on y saute.

Remarquez que les vieux chargeurs d'amorçage (vieilles versions de LILO) ne pouvaient charger que seulement les 4 premiers secteurs de setup, c'est pourquoi il y a un code dans setup qui charge le reste de lui-même si besoin est. D'autre part, le code de setup devait tenir compte des diverses combinaisons de type/version de chargeur par rapport à zImage/bzImage et est donc très complexe.

Examinons la bidouille (kludge) dans le code du secteur d'amorçage qui nous permet de charger un

gros noyau appelé aussi « bzImage ». Les secteurs de setup sont chargés comme d'habitude en 0x90200, mais le noyau est chargé par morceaux de 64k, grâce à une fonction qui appelle le BIOS pour déplacer les données de la mémoire basse vers la mémoire haute. Cette fonction est référencée par `bootsect_kludge` dans `bootsect.S` et est définie en tant que `bootsect_helper` dans `setup.S`. Le label `bootsect_kludge` dans `setup.S` contient la valeur du segment de setup et le décalage du code `bootsect_helper` par rapport à lui de telle façon que bootsector peut utiliser l'instruction `lcall` pour y sauter (saut inter-segment). La raison pour laquelle ce code est dans `setup.S` est simplement qu'il n'y a plus de place dans bootsect.S (ce n'est pas tout à fait exact — il reste approximativement 4 octets et au moins 1 octet de libre dans `bootsect.S` mais c'est évidemment insuffisant). Cette fonction utilise le service BIOS int 0x15 (ax=0x8700) pour déplacer des données vers la mémoire haute et réinitialise toujours %es afin de pointer sur 0x10000. Ainsi on s'assure que le code de `bootsect.S` ne sort pas de la mémoire basse lors de la copie des données depuis le disque.

2.5. Utilisation de LILO comme chargeur d'amorçage (bootloader)

Il y a plusieurs avantages à utiliser un chargeur d'amorçage spécialisé (LILO) par rapport au simple secteur d'amorçage de Linux :

1. La possibilité de choisir entre plusieurs noyaux Linux ou même plusieurs systèmes.
2. La possibilité de passer des paramètres en ligne de commande au noyau (il y a un correctif appelé BCP qui donne cette possibilité à bootsector+setup).
3. La possibilité de charger un noyau bzImage plus grand — jusqu'à 2.5 Mo au lieu de 1 Mo.

Les vieilles versions de LILO (v17 et avant) ne peuvent pas charger les noyaux bzImage. Les nouvelles versions (depuis quelques années) utilisent les mêmes techniques que bootsect+setup de déplacement des données de la mémoire basse vers la haute par le biais de services BIOS. Quelques personnes (Peter Anvin notamment) pensent que le support de zImage devrait être supprimé. La raison principale pour le conserver (d'après Alan Cox), c'est qu'il reste des BIOS « foireux » qui rendent impossible d'amorcer bzImage alors qu'ils chargent bien zImage.

La dernière chose que fait LILO est de sauter vers `setup.S` et les choses se déroulent ensuite comme d'habitude.

2.6. Initialisation de haut niveau

Par « initialisation de haut niveau » on entend tout ce qui n'est pas directement lié au bootstrap, même si certaines parties du code exécuté sont écrites en assembleur, à savoir `arch/i386/kernel/head.S` qui est le début du noyau non compressé. Les étapes suivantes sont exécutées :

1. Initialisation des valeurs de segments (%ds = %es = %fs = %gs = __KERNEL_DS = 0x18).
2. Initialisation de la table des pages.
3. Activation de la pagination en positionnant le bit PG dans %cr0.
4. Initialisation à zéro de la BSS (sur du multiprocesseur, seul le premier processeur fait cela).
5. Copie des premiers 2 ko des paramètres d'amorçage (ligne de commande du noyau).
6. Vérification du type de CPU en utilisant EFLAGS et, si possible, cpuid, capable de détecter un 386 ou plus.
7. Le premier CPU appelle `start_kernel()`, tous les autres appellent `arch/`

`i386/kernel/smpboot.c:initialize_secondary()` si ready=1, qui ne fait que recharger esp/eip et ne retourne pas.

La `init/main.c:start_kernel()` est écrite en C et fait les choses suivantes :

1. Pose un verrou noyau global (c'est nécessaire afin qu'un seul CPU fasse l'initialisation).
2. Effectue les initialisations spécifiques à la plate-forme (analyse de la mémoire, copie une fois encore de la ligne de commande d'amorçage, etc.).
3. Affiche la « bannière » du noyau Linux contenant la version, le compilateur utilisé pour le construire, etc, jusqu'aux messages des tampons noyau. Son contenu est celui de la variable Linux_banner définie dans init/version.c et c'est la même chaîne qui est affichée par *cat /proc/version*.
4. Initialise les trappes (traps).
5. Initialise les IRQ (Interrupt ReQuest).
6. Initialise les données nécessaires à l'ordonnanceur (scheduler).
7. Initialise les données conservant le temps.
8. Initialise le sous-système d'IRQ logiciel (softirq).
9. Analyse les options de la ligne de commande d'amorçage.
10. Initialise la console.
11. Si le support des modules a été compilé dans le noyau, il initialise les capacités de chargement dynamique des modules.
12. Si la ligne de commande contient « profile= », il initialise les tampons nécessaires.
13. `kmem_cache_init()`, initialise la plupart du slab allocator (l'allocateur de tranche ?).
14. Active les interruptions.
15. Évalue la vitesse de ce CPU en BogoMips.
16. Appelle `mem_init()` qui calcule `max_mapnr`, `totalram_pages` et `high_memory` puis affiche la ligne « Memory: ... ».
17. `kmem_cache_sizes_init()`, finit l'initialisation du slab allocator.
18. Initialise les structures de données utilisées par procfs.
19. `fork_init()`, crée l'`uid_cache`, initialise `max_threads` en fonction de la quantité de mémoire disponible et configure `RLIMIT_NPROC` pour que `init_task` soit égal à `max_threads/2`.
20. Crée les divers caches de type slab nécessaires pour le système de fichiers virtuel, la mémoire virtuelle, le cache tampon, etc.
21. Si le support de la communication interprocessus (IPC System V) est compilé dans le noyau, initialise le sous-système IPC. Remarquez que pour shm System V, cela inclut de monter une instance du système de fichiers shmfs en interne (dans le noyau).
22. Si le support des quotas est compilé dans le noyau, crée et initialise un cache slab spécial pour eux.
23. Effectue les « vérifications de bogues (check for bugs) » spécifiques à l'architecture et, à

chaque fois que possible, active les corrections pour les bugs processeur, bus, et cætera. La comparaison de plusieurs architectures révèle que « ia64 n'a pas de bugs » et que « ia32 en a quelques-uns », un bon exemple en est le « bug f00f » qui est testé seulement si le noyau est compilé pour un processeur inférieur au 686 et est alors corrigé.

24. Positionne un drapeau pour indiquer qu'un ordonnancement doit être effectué à la prochaine occasion et crée un thread noyau `init()` qui va « exec » execute_command si on a un paramètre de boot « init= », ou essayer d'exécuter */sbin/init*, */etc/init*, */bin/init*, */bin/sh* dans cet ordre ; si tout échoue, le noyau « panic » et émet la suggestion d'utiliser le paramètre « init= ».

25. Rentre dans une boucle inactive (idle), qui est un fil inactif (idle thread) avec un pid=0.

La chose importante à noter ici c'est que le thread noyau `init()` appelle `do_basic_setup()` qui à son tour appelle `do_initcalls()` qui parcourt la liste des fonctions enregistrées par le biais de `__initcall` ou de la macro `module_init()` et les invoque. Ces fonctions ne dépendent pas les unes des autres ou bien leurs dépendances ont été manuellement fixées par l'ordre de l'édition de liens dans les Makefiles. Ce qui veut dire qu'en fonction de la position des répertoires dans l'arborescence et de la structure des Makefiles, l'ordre dans lequel les fonctions d'initialisation sont appelées peut changer. Quelquefois, c'est important d'en tenir compte car imaginez deux sous-systèmes A et B avec B dépendant d'initialisations faites dans A. Si A est compilé statiquement et que B est un module, alors on est sûr qu'on entrera dans B après que A ait préparé tout l'environnement nécessaire. Si A est un module, alors B en est nécessairement un aussi donc il n'y pas de problème. Mais que se passe-t-il si A et B sont liés statiquement dans le noyau ? L'ordre dans lequel ils sont invoqués dépend du décalage relatif de leurs points d'entrée dans la section ELF `.initcall.init` de l'image noyau. Rogier Wolff a proposé d'introduire une infrastructure à priorité hiérarchique dans laquelle les modules permettraient à l'éditeur de liens (linker) de savoir dans quel ordre (relatif) ils doivent être liés, mais jusqu'ici il n'y pas de correctif disponible qui implémente cela de manière suffisamment élégante pour être acceptable dans le noyau. Néanmoins assurez vous de l'ordre de liage. Si dans l'exemple ci-dessus A et B marchent bien une première fois en étant compilés statiquement, ils marcheront toujours, pourvu qu'ils soient listés séquentiellement dans le même Makefile. S'ils ne marchent pas, changez l'ordre dans lequel leurs fichiers objets sont listés.

Une autre chose qui vaut d'être notée c'est la capacité qu'a Linux d'exécuter un « autre programme init » en passant une ligne de commande « init= » à l'amorçage. C'est utile pour réparer un */sbin/init* abîmé accidentellement ou déboguer les scripts d'initialisation (rc) et `/etc/inittab` à la main, en les exécutant un par un.

2.7. Amorçage multiprocesseur (SMP) sur x86

Sur un système multiprocesseur, le processeur d'amorçage (BP) exécute la séquence normale d'instructions du secteur d'amorçage (bootsector), setup etc. jusqu'à ce qu'on atteigne `start_kernel()`, et ensuite `smp_init()` et plus particulièrement `src/i386/kernel/smpboot.c:smp_boot_cpus()`. Le `smp_boot_cpus()` effectue une boucle pour chaque apicid (jusqu'à `NR_CPUS`) et appelle pour chacun `do_boot_cpu()`. Ce que fait `do_boot_cpu()`, c'est créer (i.e. `fork_by_hand`) une tâche inactive (idle) pour le cpu cible et écrire à des emplacements bien définis par les spécifications Intel MP (0x467/0x469) l'EIP du code « trampoline » contenu dans `trampoline.S`. Ensuite il génère STARTUP IPI sur le cpu cible, ce qui fait que cet AP exécute le code de `trampoline.S`.

Le CPU d'amorçage crée une copie du code de trampoline pour chaque CPU en mémoire basse. Le code AP écrit un nombre magique dans son propre code qui est vérifié par le processeur d'amorçage pour s'assurer que l'AP est en train d'exécuter le code trampoline. La nécessité de mettre le code trampoline en mémoire basse vient des spécifications Intel MP.

Le code trampoline met simplement le registre %bx à 1, passe en mode protégé et saute vers startup_32 qui est l'entrée principale de `arch/i386/kernel/head.S`.

Maintenant que l'AP a commencé l'exécution de `head.S` et découvre que ce n'est pas un processeur d'amorçage, il passe le code qui nettoie la BSS et appelle `initialize_secondary()` qui ne fait qu'appeler la tâche inactive pour ce CPU — rappelez vous que `init_tasks[cpu]` avait déjà

été initialisé par le processeur d'amorçage en exécutant `do_boot_cpu(cpu)`.

Remarquez que init_task peut être partagé mais que chaque tâche inactive doit avoir sa propre TSS. C'est pourquoi `init_tss[NR_CPUS]` est un tableau.

2.8. Libérer les données et le code d'initialisation

Une fois que le système d'exploitation s'est initialisé, la plus grande partie du code et des structures de données ne sont jamais réutilisés. La plupart des systèmes (BSD, FreeBSD etc.) ne peuvent disposer de ces informations, et donc gaspillent la précieuse mémoire physique du noyau. L'excuse qu'ils fournissent (voir le livre McKusick's 4.4BSD) c'est que le code en question est réparti autour de plusieurs sous-systèmes et que ce n'est pas faisable de le libérer : *the relevant code is spread around various subsystems and so it is not feasible to free it*. Linux, bien sûr, ne peut se retrancher derrière de telles excuses car sous Linux « si quelque chose est en principe possible, alors c'est déjà implémenté ou quelqu'un travaille dessus ».

Donc, comme je l'ai dit plus tôt, le noyau Linux ne peut être compilé que comme un binaire ELF, et maintenant nous en avons la raison (ou une des raisons). La raison rattachée à l'élimination des données et du code d'initialisation est que Linux fournit 2 macros à utiliser :

- `__init` — pour le code d'initialisation
- `__initdata` — pour les données

Elles s'évaluent comme des spécificateurs d'attributs gcc (aussi connus comme « gcc magic ») telles que définies dans `include/linux/init.h` :

```
#ifndef MODULE
#define __init        __attribute__ ((__section__ (".text.init")))
#define __initdata    __attribute__ ((__section__ (".data.init")))
#else
#define __init
#define __initdata
#endif
```

Ce qui veut dire que si le code est compilé statiquement dans le noyau (i.e on ne définit pas MODULE), alors celui ci est placé dans une section ELF spéciale `.text.init`, qui est déclarée dans la carte de correspondance de l'éditeur de lien dans `arch/i386/vmlinux.lds`. Sinon (i.e. si c'est un module) les macros sont évaluées à rien.

Ce qui ce passe durant l'amorçage, c'est que le thread noyau « init » (fonction `init/main.c:init()`) appelle une fonction spécifique à l'architecture `free_initmem()` qui libère toutes les pages entre les adresses `__init_begin` et `__init_end`.

Sur un système typique (ma station de travail), le résultat est que 260k de mémoire sont libérés.

Les fonctions enregistrées via `module_init()` sont placées dans `.initcall.init` qui est aussi libéré lors d'une compilation statique. La tendance actuelle sous Linux, quand on crée un sous-système (pas forcement un module), est de fournir dès les premières étapes de la conception les points d'entrée init/exit de telle façon que le sous-système puisse dans le futur être modularisé si besoin. Pipefs en est un exemple, regardez `fs/pipe.c`. Même si un sous système donné ne doit jamais devenir un module, i.e. bdflush (voir `fs/buffer.c`), c'est toujours mieux et plus propre d'utiliser la macro `module_init()` à la place de la fonction d'initialisation, pourvu qu'on n'attache pas d'importance au moment exact où la fonction est appelée.

Il y a deux autres macros qui fonctionnent de manière similaire, appelées `__exit` et `__exitdata`, mais elles sont plus directement liées au support des modules et par conséquent seront expliquées dans un prochain paragraphe.

2.9. Traitement de la ligne de commande du noyau

Rappelons nous ce que devient la ligne de commande passée au noyau pendant l'amorçage :

1. LILO (ou BCP) traite la ligne de commande en utilisant les services clavier du BIOS et la stocke dans un endroit bien repéré de la mémoire physique, avec une signature signifiant qu'il y a une ligne de commande valide ici.

2. `arch/i386/kernel/head.S` en copie les 2 premiers k vers la page zéro.

3. `arch/i386/kernel/setup.c:parse_mem_cmdline()` (appelée par `setup_arch()`, elle même appelée par `start_kernel()`) copie 256 octets depuis la page zéro dans `saved_command_line` qui est affiché par `/proc/cmdline`. Cette même routine traite l'option « mem= » si elle est présente et fait les ajustements nécessaires aux paramètres de la VM.

4. Revenons à la ligne de commande traitée par `parse_options()` (appelé par `start_kernel()`) qui traite quelques paramètres « internes au noyau » (actuellement « init= » et l'environnement/arguments pour init) et passe chaque mot à `checksetup()`.

5. `checksetup()` parcourt le code de la section ELF `.setup.init` et invoque chaque fonction, lui passant le mot précédent si celui-ci convient. Remarquez qu'en utilisant une valeur de retour de 0 depuis la fonction enregistrée via `__setup()`, il est possible de passer le même « variable=value » à plus d'une fonction avec « value » invalide pour l'une et valide pour l'autre. Jeff Garzik commente : « les hackers qui font cela s'en mordent les doigts :) ». Pourquoi ? Parce que c'est clairement spécifique à l'ordre d'édition des liens, i.e. un noyau lié dans un sens aura la fonction A appelée avant la fonction B, pour un autre ce sera le contraire, le résultat dépendant de l'ordre.

Alors, comment écrit-on le code qui traite la ligne de commande d'amorçage ? On utilise la macro `__setup()` définie dans `include/linux/init.h` :

```
/*
 * Utilisé pour initialiser les paramètres du noyau avec les valeurs
 * de la ligne de commande
 */
struct kernel_param {
const char *str;
int (*setup_func)(char *);
};

extern struct kernel_param __setup_start, __setup_end;
#ifndef MODULE
#define __setup(str, fn) \
static char __setup_str_##fn[] __initdata = str; \
static struct kernel_param __setup_##fn __initsetup = \
              { __setup_str_##fn, fn }

#else
#define __setup(str,func) /* rien */
#endif
```

Alors, typiquement vous l'utiliserez dans votre code de la façon suivante (pris du code d'un vrai driver, BusLogic HBA `drivers/scsi/BusLogic.c` :

```
static int __init
BusLogic_Setup(char *str)
{
int ints[3];

(void)get_options(str, ARRAY_SIZE(ints), ints);

if (ints[0] != 0) {
```

```
    BusLogic_Error("BusLogic: Obsolete Command Line Entry "
   "Format Ignored\n", NULL);
   return 0;
   }
if (str == NULL || *str == '\0')
   return 0;
   return BusLogic_ParseDriverOptions(str);
   }

__setup("BusLogic=", BusLogic_Setup);
```

Remarquez que `__setup()` ne fait rien pour les modules, donc le code qui veut traiter la ligne de commande d'amorçage et qui peut être soit dans un module, soit lié statiquement doit invoquer sa fonction d'analyse syntaxique manuellement dans la routine d'initialisation du module. Ce qui veut aussi dire qu'il est possible d'écrire du code qui est capable de traiter les paramètres quand il est compilé comme un module et pas quand il est statique ou vice versa.

3. Processus et gestion des interruptions

3.1. Structure de tâche et table des processus

Sous Linux une structure `struct task_struct` est allouée dynamiquement à chaque processus. Le nombre maximum de processus qui peuvent être crées sous Linux est limité par la quantité de mémoire physique présente, et est égal à (voir `kernel/fork.c:fork_init()` :

```
/*
* La valeur par défaut du nombre maximum de threads est fixée à une
* valeur sûre~: la structure des threads peut occuper au maximum la moitié
* de la mémoire.
*/
max_threads = mempages / (THREAD_SIZE/PAGE_SIZE) / 2;
```

ce qui, sur une architecture IA32, veut dire `num_physpages/4`. Par exemple, sur une machine de 512 M, vous pouvez créer 32k de threads. C'est une amélioration considérable par rapport à la limite 4k-epsilon des vieux noyaux (2.2 et avant). De plus, cela peut-être modifié soit pendant l'exécution en utilisant KERN_MAX_THREADS de *sysctl(2)*, soit simplement en utilisant l'interface procfs pour paramétrer le noyau :

```
# cat /proc/sys/kernel/threads-max
32764
# echo 100000 > /proc/sys/kernel/threads-max
# cat /proc/sys/kernel/threads-max
100000
# gdb -q vmlinux /proc/kcore
Core was generated by `BOOT_IMAGE=240ac18 ro root=306 video=matrox:vesa:0x118'.
#0  0x0 in ?? ()
(gdb) p max_threads
$1 = 100000
```

L'ensemble des processus sur un système Linux est représenté par un ensemble de structures `struct task_struct` qui sont liées de deux façons :

1. par une table de hachage, hachée sur le pid, et
2. par une liste circulaire doublement chaînée utilisant les pointeurs `p->next_task` et `p->prev_task`.

La table de hachage est appelée `pidhash[]` et est définie dans `include/linux/sched.h` :

```
/* Hachage sur le PID. (est-ce que ceci ne devrait pas être dynamique~?) */
#define PIDHASH_SZ (4096 >> 2)
extern struct task_struct *pidhash[PIDHASH_SZ];

#define pid_hashfn(x)   ((((x) >> 8) ^ (x)) & (PIDHASH_SZ - 1))
```

Les tâches sont hachées en fonction de leur pid et la fonction précédente est censée distribuer les éléments uniformément dans leur domaine (de `0` à `PID_MAX-1`). La table de hachage est utilisée pour retrouver rapidement une tâche par son pid en utilisant `find_task_pid()` in-line depuis `include/linux/sched.h` :

```
static inline struct task_struct *find_task_by_pid(int pid)
{
struct task_struct *p, **htable = &pidhash[pid_hashfn(pid)];

for(p = *htable; p && p->pid != pid; p = p->pidhash_next);

return p;
}
```

Les tâches de chaque liste de hachage (i.e hachées à la même valeur) sont liées par `p->pidhash_next/pidhash_pprev` qui sont utilisés par `hash_pid()` et `unhash_pid()` pour insérer et retirer un processus donné dans la table de hachage. Ceci est fait sous la protection d'un verrou tournant en lecture/écriture (read/write spinlock) appelé `tasklist_lock` posé pour ÉCRIRE (WRITE).

La double liste chaînée circulaire qui utilise `p->next_task/prev_task` est tenue à jour de façon à ce que l'on puisse parcourir toutes les tâches du système facilement. C'est fait par la macro `for_each_task()` de `include/linux/sched.h` :

```
#define for_each_task(p) \
for (p = &init_task ; (p = p->next_task) != &init_task ; )
```

Les utilisateurs de `for_each_task()` doivent poser un verrou tasklist_lock pour LIRE. Remarquez que `for_each_task()` utilise `init_task` pour marquer le début et la fin de la liste — c'est plus sûr car la tâche inactive (idle — pid 0) ne finit jamais.

Les fonctions qui modifient la table de hachage des processus ou des liens de la table des processus, notamment `fork()`, `exit()` et `ptrace()`, doivent poser `tasklist_lock` pour ÉCRIRE. Ce qui est plus intéressant, c'est que pour écrire il faut aussi désactiver les interruptions sur le CPU local. La raison de cela est loin d'être triviale : la fonction `send_sigio()` parcourt la liste des tâches et donc pose un `tasklist_lock` pour LIRE, et elle est appelée depuis `kill_fasync()` dans un contexte d'interruption. C'est pourquoi ceux qui écrivent doivent désactiver les interruptions alors que ceux qui lisent n'ont pas besoin de le faire.

Maintenant que nous comprenons comment les structures `task_struct` sont liées ensemble, examinons les membres de `task_struct`. Ils correspondent plus ou moins aux membres des structures UNIX « struct proc » et « struct user » combinées ensemble.

Les autres versions d'UNIX séparaient l'information sur l'état des tâches en une partie qui devait être gardée en mémoire tout le temps (appelée « proc struct » qui inclut l'état du processus, les informations d'ordonnancement etc.) et une autre partie qui n'est nécessaire que lorsque le processus tourne (appelée « u area » qui inclut la table des descripteurs de fichiers, les informations de quota disque etc.). La seule raison d'être d'une conception aussi laide est que la mémoire était alors une denrée rare. Les systèmes d'exploitation modernes (bon, seulement Linux pour le moment mais d'autres, comme FreeBSD semblent évoluer dans la même direction) n'ont pas besoin d'une telle séparation, ils maintiennent l'état des processus dans une structure de données du noyau qui réside en mémoire

en permanence.

La structure task_struct est déclarée dans `include/linux/sched.h` et a actuellement une taille de 1680 octets.

Le champ state (état) est déclaré comme :

```
volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */

#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1

#define TASK_UNINTERRUPTIBLE    2
#define TASK_ZOMBIE             4
#define TASK_STOPPED            8
#define TASK_EXCLUSIVE          32
```

Pourquoi est ce que `TASK_EXCLUSIVE` est définie comme 32 et non 16 ? parce que 16 était utilisé par `TASK_SWAPPING` et que j'ai oublié de décaler `TASK_EXCLUSIVE` quand j'ai retiré toutes les références à `TASK_SWAPPING` (quelquefois dans 2.3.x).

La déclaration `volatile` dans `p->state` signifie qu'il peut être modifié de manière asynchrone (depuis un gestionnaire d'interruption) :

1. *TASK_RUNNING* : signifie que la tâche « est censée être » dans la file d'exécution (runqueue). La raison pour laquelle elle peut ne pas être dans cette file, c'est que le marquage d'une tâche comme `TASK_RUNNING` et son placement dans la file n'est pas atomique. Il vous faut maintenir le verrou tournant en lecture-écriture `runqueue_lock` pour lire afin de rechercher dans la file d'exécution . Si vous faite cela vous verrez alors que toutes les tâches dans la file sont dans l'état `TASK_RUNNING`. Néanmoins, la réciproque n'est pas vraie pour les raisons expliquées précédemment. De la même façon, les pilotes peuvent se marquer eux mêmes (ou plutôt le contexte de processus dans lequel ils s'exécutent) comme `TASK_INTERRUPTIBLE` (ou `TASK_UNINTERRUPTIBLE`) et appeler `schedule()`, qui va alors les retirer de la file d'exécution (sauf s'il y a un signal en attente à leur destination, auquel cas, ils restent dans la file).

2. *TASK_INTERRUPTIBLE* : signifie que la tâche est endormie mais qu'elle peut être réveillée par un signal ou par l'expiration d'une alarme.

3. *TASK_UNINTERRUPTIBLE* : idem que `TASK_INTERRUPTIBLE`, sauf qu'elle ne peut pas être réveillée.

4. *TASK_ZOMBIE* : la tâche s'est terminée mais son état n'a pas été collecté par son parent (naturel ou par adoption — pas d'appel `wait()` du parent).

5. *TASK_STOPPED* : La tâche a été arrêtée, soit par un signal de contrôle des travaux (jobs) soit par *ptrace(2)*.

6. *TASK_EXCLUSIVE* : Ce n'est pas un état à part entière mais il peut être combiné par un OU (OR) soit à `TASK_INTERRUPTIBLE` soit à `TASK_UNINTERRUPTIBLE`. Cela signifie que si cette tâche est endormie dans la file d'attente avec beaucoup d'autres, elle sera la seule à être réveillée, sans réveiller les autres tâches en attente, et provoquer un problème de « thundering herd ».

Les drapeaux (flags) de tâches contiennent des informations sur les états des processus, états qui ne sont pas mutuellement exclusifs :

```
unsigned long flags;  /* drapeaux pour chaque processus, cf. plus bas */
/*
 * Drapeaux par processus
```

```
 */
#define PF_ALIGNWARN  0x00000001    /* Averti des pb d'alignement. Pas encore *
                                    /* implanté, pour 486 seulement */
#define PF_STARTING   0x00000002    /* en création */
#define PF_EXITING    0x00000004    /* en train de s'arrêter */
#define PF_FORKNOEXEC 0x00000040    /* s'est scindé, mais n'a pas encore */
                                    /* fait de « exec  » */
#define PF_SUPERPRIV  0x00000100    /* a utilisé les privilèges de */
                                    /* super-utilisateur */
#define PF_DUMPCORE   0x00000200    /* a laissé un « core dump » */
#define PF_SIGNALED   0x00000400    /* tué par un signal */
#define PF_MEMALLOC   0x00000800    /* en train d'allouer de la mémoire */
#define PF_VFORK      0x00001000    /* réveille le parent dans mm_release */
#define PF_USEDFPU    0x00100000    /* à util. le FPU au cours de ce */
                                    /* quantième (SMP) */
```

Les champs `p->has_cpu`, `p->processor`, `p->counter`, `p->priority`, `p->policy` et `p->rt_priority` concernent l'ordonnanceur et seront examinés plus tard.

Les champs `p->mm` et `p->active_mm` pointent respectivement sur l'espace d'adressage des processus décrit par la structure `mm_struct` et vers l'espace d'adressage actif si le processus n'en n'a pas un réel (i.e. pour les threads noyau). Cela aide à minimiser les débordements de TLB lors d'un basculement d'espace d'adressage quand une tâche est sortie de l'ordonnancement. Donc, si on ajoute une tâche dans l'ordonnanceur (qui n'a pas de `p->mm`) alors son `next->active_mm` sera positionné sur le `prev->active_mm` de la tâche qui est sortie, qui sera le même que `prev->mm` si `prev->mm != NULL`. L'espace d'adressage peut être partagé entre les threads si le drapeau (flag) `CLONE_VM` est passé à l'appel système *clone(2)* ou par le biais de l'appel système *vfork(2)*.

Les champs `p->exec_domain` et `p->personality` sont liés à la personnalité de la tâche, i.e. la façon dont certain appels système se comportent pour émuler la « personnalité » de certaines versions éloignées d'UNIX.

Le champs `p->fs` contient les informations sur le système de fichiers, ce qui veut dire sous Linux 3 types d'informations :

1. le dentry de la racine et le point de montage,
2. un dentry de la racine et un point de montage alternatif,
3. le dentry du répertoire de travail courant et le point de montage.

Cette structure inclut aussi un compteur de références car elle peut être partagée entre des tâches clonées quand le drapeau `CLONE_FS` est passé à l'appel système *clone(2)*.

Le champ `p->files` contient la table des descripteurs de fichiers. Elle aussi peut être partagée entre des tâches clonées, pourvu que le drapeau `CLONE_FILES` soit spécifié avec l'appel système *clone(2)*.

le champ `p->sig` contient les gestionnaires (handlers) de signaux et peuvent être partagés entre des tâches par le biais de `CLONE_SIGHAND`.

3.2. Création et terminaison des tâches et des threads noyau

Les livres consacrés aux systèmes d'exploitation définissent les « processus » de différentes façons, depuis « l'instance d'un programme en exécution » jusqu'à « ce qui est produit par les appels système clone(2) ou fork(2) ». Sous Linux il y a trois type de processus :

- le(s) thread(s) idle (inutile/inactif),

- les threads noyau,
- les tâches utilisateur.

Le thread inactif est créé à la compilation pour le premier CPU ; il est ensuite crée « manuellement » pour chaque CPU par le biais de la fonction spécifique à l'architecture `fork_by_hand()` dans `arch/i386/kernel/smpboot.c`, qui utilise l'appel système *fork(2)* appelé à la main (sur certaines architectures). Les tâches inactives partagent une structure init_task mais possèdent une structure TSS privée, dans le tableau `init_tss` de chaque CPU. Les tâches inactives ont toutes un pid = 0 et aucune autre tâche ne peut partager le même pid, i.e. utiliser le drapeau `CLONE_PID` de *clone(2)*.

Les threads noyau sont créés en utilisant la fonction `kernel_thread()` qui invoque l'appel système *clone(2)* en mode noyau. Les threads noyau n'ont pas en général d'espace d'adressage utilisateur, i.e. `p->mm = NULL`, car il font un `exit_mm()` explicite, i.e. via la fonction `daemonize()`. Les threads noyau peuvent toujours accéder directement à l'espace d'adressage du noyau. Il leur est attribué des numéros de pid dans la tranche basse. L'exécution dans l'anneau (ring) 0 du processeur (sur x86, c'est le cas) implique que le thread noyau profite de tous les privilèges d'entrée/sortie et ne peut être préempté par l'ordonnanceur.

Les tâches utilisateurs sont créés avec les appels système *clone(2)* ou *fork(2)*, qui utilisent la fonction *kernel/fork.c:do_fork()*.

Voyons ce qui se passe quand un processus utilisateur fait un appel système *fork(2)*. Bien que *fork(2)* soit dépendant de l'architecture à cause des différentes façons de transmettre la pile et les registres utilisateur, la fonction utilisée réellement `do_fork()` pour ce travail est portable et est placée dans `kernel/fork.c`.

Les actions suivantes sont réalisées :

1. La variable locale `retval` est mise à `-ENOMEM`, car c'est la valeur à laquelle `errno` doit être positionnée si *fork(2)* échoue lors de l'allocation d'une nouvelle structure de tâche.
2. Si `CLONE_PID` est positionné dans `clone_flags`, alors on retourne une erreur (`-EPERM`), à moins que l'appelant soit le thread inactif (pendant l'amorçage seulement). Donc, un thread utilisateur normal ne peut pas passer `CLONE_PID` à *clone(2)* en espérant que ça marche. Pour *fork(2)*, ce n'est pas pertinent, car `clone_flags` est positionné à `SIFCHLD`, ça n'est pertinent que lorsque `do_fork()` est appelé par `sys_clone()` qui passe alors `clone_flags` avec la valeur demandée depuis l'espace utilisateur.
3. `current->vfork_sem` est initialisé (il sera nettoyé plus tard dans l'enfant). C'est utilisé par `sys_vfork()` (l'appel système *vfork(2)* correspond à `clone_flags = CLONE_VFORK|CLONE_VM|SIGCHLD`) pour que le parent dorme jusqu'à ce que l'enfant fasse `mm_release()`, par exemple en résultat de l'`exec()`ution d'un autre programme ou en terminant par *exit(2)*.
4. Une nouvelle structure de tâche est allouée en utilisant la macro spécifique à l'architecture `alloc_task_struct()`. Sur un x86 c'est juste un gfp à la priorité `GFP_KERNEL`. C'est la première raison pour laquelle l'appel système *fork(2)* doit dormir. Si cette allocation échoue, on retourne `-ENOMEM`.
5. Toutes les valeurs de la structure de tâche du processus courant sont copiées dans la nouvelle structure, en utilisant l'assignation de structure `*p = *current`. Peut-être que cela devrait être remplacé par un appel à memcopy ? Plus tard, les champs qui ne doivent pas être hérités par le fils seront positionnés aux valeurs correctes.
6. Un GROS verrou de noyau est posé car autrement le reste du code ne serait pas ré-entrant.
7. Si le parent a des ressources utilisateur (un concept d'UID, Linux est suffisamment flexible pour en faire une question plutôt qu'un fait), alors on vérifie si l'utilisateur dépasse la limite douce `RLIMIT_NPROC` — si c'est le cas, on échoue avec `-EAGAIN`, sinon, on incrémente le

nombre de processus pour l'uid donné `p->user->count`.

8. Si le nombre de tâches du système dépasse la valeur paramétrable fixée par max_threads, on échoue avec `-EAGAIN`.

9. Si le binaire exécuté appartient à un domaine d'exécution modularisé, on incrémente le compteur de références du module.

10. L'enfant est marqué « non terminé » (`p->did_exec = 0`)

11. L'enfant est marqué comme ne pouvant pas être copié en zone d'échange (`p->swappable = 0`)

12. L'enfant est placé dans l'état de sommeil sans interruption (uninterruptible sleep), i.e. `p->state = TASK_UNINTERRUPTIBLE` (À FAIRE : Pourquoi c'est comme ça ? Je pense que ce n'est pas nécessaire — débarrassez vous en, Linus confirme que c'est inutile)

13. Les `p->flags` de l'enfant sont positionnés en fonction de la valeur de clone_flags; pour *fork(2)*, ce sera `p->flags = PF_FORKNOEXEC`.

14. Le pid de l'enfant `p->pid` est fixé en utilisant un algorithme rapide dans `kernel/fork.c:get_pid()` (À FAIRE : le verrou tournant `lastpid_lock` peut être redondant tant que `get_pid()` est toujours appelé sous le gros verrou du noyau depuis `do_fork()`, donc je retire les arguments drapeau de `get_pid()`, le patch à été envoyé à Alan le 20/06/2000).

15. La suite du code dans `do_fork()` initialise le reste de la structure de tâche de l'enfant. Tout à la fin, cette structure est hachée dans la table de hachage `pidhash` et l'enfant est réveillé (À FAIRE : `wake_up_process(p)` positionne `p->state = TASK_RUNNING` et ajoute le processus dans la file d'exécution (runq), néanmoins on n'a probablement pas besoin de fixer `p->state` à `TASK_RUNNING` auparavant dans `do_fork()`). La partie intéressante consiste à mettre `p->exit_signal` à `clone_flags & CSIGNAL`, ce qui pour *fork(2)* signifie seulement `SIGCHLD` et fixer `p->pdeath_signal` à 0. Le `pdeath_signal` est utilisé quand un processus « oublie » son parent originel (en mourant) et peut être fixé/obtenu par le biais des commandes `PR_GET/SET_PDEATHSIG` de l'appel système *prctl(2)* (Vous pouvez dire que la façon dont la valeur de `pdeath_signal` est retournée via un argument pointant dans l'espace utilisateur dans *prctl(2)* est un peu idiote — mea culpa, une fois que Andries Brouwer a eu mis à jour la page man ça a été trop tard pour corriger;)

Donc les tâches sont créées. Il y a plusieurs façons pour une tâche de se terminer :

1. En faisant un appel système *exit(2)* ;

2. En recevant un signal qui la fait mourir par défaut ;

3. En étant forcée de mourir dans certaines conditions ;

4. En appelant *bdflush(2)* avec `func == 1` (c'est spécifique à Linux, pour la compatibilité avec les vieilles distributions qui ont toujours la ligne « update » dans /etc/inittab — de nos jours le travail de mise à jour est fait par le thread noyau kupdate)

Les fonctions implémentant les appels systèmes sous Linux sont préfixées par `sys_`, mais elles ne font en général que vérifier les arguments ou passer des informations de la façon spécifique à l'architecture et le travail effectif est réalisé par les fonctions `do_`. Il en est ainsi avec `sys_exit()` qui appelle `do_exit()` pour faire le travail. Malgré cela, d'autres parties du noyau invoquent parfois `sys_exit()` alors quelles devraient plutôt appeler `do_exit()`.

La fonction `do_exit()` se trouve dans `kernel/exit.c`. Les points remarquables de `do_exit()` sont qu'elle :

- utilise le verrou global du noyau (elle verrouille mais ne déverrouille pas),
- appelle `schedule()` à la fin, qui ne retourne jamais,
- positionne l'état de la tâche à `TASK_ZOMBIE`,
- informe tous les enfants en envoyant `current->pdeath_signal`, s'il n'est pas nul,
- informe le parent en envoyant `current->exit_signal`, qui en général est égal à `SIGCHLD`,
- libère les ressources allouées par le fork, ferme les fichiers ouverts, etc,
- sur les architectures qui utilisent « lazy FPU switching » (ia64, mips, mips64) (À FAIRE : retirer l'argument « flags » pour les sparc, sparc64), fait tout ce qu'il faut selon le matériel pour que le propriétaire du FPU (si celui-ci appartient à la tâche courante) devienne « none » (personne).

3.3. L'ordonnanceur Linux

Le travail de l'ordonnanceur est de répartir l'accès au CPU entre les différents processus. L'ordonnanceur (ou scheduler) est implémenté dans le « fichier main du noyau » `kernel/sched.c`. Le fichier d'en-tête correspondant `include/linux/sched.h` est inclus (explicitement ou non) dans à peu près tous les fichiers sources du noyau.

Les champs de la structure de tâche intéressants pour l'ordonnanceur incluent :

- `p->need_resched` : ce champ est positionné si `schedule()` doit être appelé « à la prochaine occasion ».
- `p->counter` : nombre de tops d'horloge restant dans cette tranche d'ordonnancement, décrémenté par un chronomètre. Quand ce champ devient inférieur ou égal à zéro, il est remis à zéro et `p->need_resched` est positionné. Cette variable est parfois appelée « dynamic priority » d'un processus car cette priorité change par elle-même.
- `p->priority` : priorité statique du processus, elle ne peut être changée que par des appels systèmes répertoriés comme *nice(2)*, *sched_setparam(2)* POSIX.1b ou *setpriority(2)* 4.4BSD/SVR4.
- `p->rt_priority` : priorité temps réel
- `p->policy` : politique d'ordonnancement, spécifie à quelle classe d'ordonnancement la tâche appartient. Les tâches peuvent modifier leur classe d'ordonnancement en utilisant l'appel système *sched_setscheduler(2)*. Les valeurs reconnues sont `SCHED_OTHER` (processus UNIX traditionnel), `SCHED_FIFO` (processus FIFO temps réel POSIX.1b) et `SCHED_RR` (processus temps réel POSIX utilisant l'algorithme round-robin). On peut aussi faire `SCHED_YIELD` OU l'une de ces valeurs pour indiquer que c'est le processus qui décide de libérer le CPU, par exemple en invoquant l'appel système *sched_yield(2)*. Un processus FIFO (premier arrivé, premier servi) temps réel tournera jusqu'à ce que : a) il soit bloqué par une entrée/sortie, b) il libère explicitement le CPU ou c) il soit préempté par un autre processus temps réel avec une valeur de priorité `p->rt_priority` plus importante. Pour `SCHED_RR` c'est la même chose que pour `SCHED_FIFO`, sauf que lorsque la tranche de temps expire il retourne à la fin de la file d'exécution (runqueue).

L'algorithme de d'ordonnancement est simple, malgré l'apparente complexité de la fonction `schedule()`. Cette fonction est complexe car elle implémente les trois algorithmes d'ordonnancement en un seul et aussi à cause de certaines spécificités subtiles de l'architecture multiprocesseur (SMP).

Les gotos apparemment « inutiles » de `schedule()` sont ici dans le but de générer le code le mieux optimisé (pour i386). Remarquez que le code de l'ordonnanceur (comme la plus grande partie du noyau) a été complètement réécrit pour le 2.4, donc la discussion qui suit ne s'applique pas à la

série des 2.2 ou moins.

Regardons cette fonction en détail :

1. Si `current->active_mm == NULL` alors quelque chose est faux. Le processus courant, même un thread noyau (`current->mm == NULL`) doit avoir un `p->active_mm` valide en permanence.

2. S'il y a quelque chose à traiter dans la file de tâches `tq_scheduler`, le faire maintenant. La file de tâches fournit au noyau un mécanisme pour ordonnancer l'exécution ultérieurs de fonctions. On détaillera cela par la suite.

3. Initialiser les variables locales `prev` et `this_cpu` respectivement pour la tâche et le CPU courants.

4. Vérifier si `schedule()` a été appelé par le gestionnaire d'interruption (suite à un bug) et panique si c'est le cas.

5. Libérer le verrou global du noyau.

6. S'il y a un travail à réaliser par IRQ logiciel (softirq), le faire maintenant.

7. Initialiser le pointeur local `struct schedule_data *sched_data` pour le faire pointer sur la zone des données d'ordonnancement par CPU (cacheline-aligné pour éviter le cacheline ping-pong) qui contient la valeur TSC de `last_schedule` et le pointeur vers la dernière structure de tâche ordonnancée (À FAIRE : `sched_data` n'est utilisé que pour le multiprocesseur (SMP) alors pourquoi `init_idle()` l'initialise-t-il aussi sur de l'uniprocesseur ?).

8. Un verrou tournant `runqueue_lock` est posé. Remarquez que nous utilisons `spin_lock_irq()` car dans `schedule()` on est sûr que les interruptions sont activées. Par conséquent, quand on déverrouille `runqueue_lock`, il n'y a qu'à les réactiver au lieu de sauver/restaurer eflags (variante `spin_lock_irqsave/restore`).

9. La machine d'état de la tâche : si la tâche est dans un état `TASK_RUNNING`, on n'y touche pas ; si elle est dans un état `TASK_INTERRUPTIBLE` et qu'un signal est suspendu elle est placée dans l'état `TASK_RUNNING`. Dans tous les autres cas elle est effacée de la file d'exécution.

10. `next` (suivant — le meilleur candidat à l'ordonnancement) prend pour valeur la tâche idle de notre CPU. Néanmoins, la qualité de ce candidat est mise à une valeur très basse (-1000), dans l'espoir de trouver un meilleur candidat.

11. Si la tâche `prev` (courante) est dans un état `TASK_RUNNING`, alors la qualité courante est fixée à sa qualité et cette tâche est marquée comme meilleur candidat à ordonnancer que la tâche idle.

12. Maintenant la queue d'exécution est examinée et les qualités de chaque processus susceptible d'être ordonnancé sur ce CPU sont comparées à la valeur courante ; le processus qui a la qualité la plus haute gagne. Maintenant le concept de « susceptible d'être ordonnancé sur ce CPU » doit être éclairci : sur de l'UP, chaque processus de la queue d'exécution est éligible pour être ordonnancé; sur du multiprocesseur, seuls les processus qui ne sont pas en cours d'exécution sur un autre CPU sont éligibles pour être ordonnancés sur ce CPU. La qualité est calculée au moyen d'une fonction appelée `goodness()`, qui fixe très haut la qualité des processus temps réel (`1000 + p->rt_priority`), une valeur supérieure à 1000 garantit qu'aucun processus `SCHED_OTHER` ne peut gagner ; donc ils ne sont en compétition qu'avec d'autres processus temps réel ayant une plus grande priorité `p->rt_priority`. La fonction goodness retourne 0 si la tranche de temps (process time slice) du processus (`p->counter`) est terminée. Pour des processus non temps réel, la valeur initiale de la qualité est fixée à `p->counter` — de cette façon, le processus a moins de chances d'avoir le CPU s'il l'a déjà eu récemment, i.e. les processus interactifs sont plus favorisés que les mangeurs de CPU. La constante spécifique à l'architecture `PROC_CHANGE_PENALTY` essaie d'implémenter « l'affinité CPU » (i.e. donner l'avantage à un processus du même CPU). Cela donne aussi un léger avantage aux processus ayant mm pointant sur le `active_mm` courant ou aux processus sans espace d'adressage (utilisateur), i.e. les threads noyau.

13. si la valeur courante de la qualité est 0 alors la liste complète des processus (pas seulement ceux de la queue d'exécution!) est examinée et leurs priorités dynamiques sont recalculées par un algorithme simple :

```
recalculate:
        {
                struct task_struct *p;
                spin_unlock_irq(&runqueue_lock);
                read_lock(&tasklist_lock);
                for_each_task(p)
                        p->counter = (p->counter >> 1) + p->priority;
                read_unlock(&tasklist_lock);
                spin_lock_irq(&runqueue_lock);
        }
```

    Remarquez que l'on enlève le verrou `runqueue_lock` avant de recalculer. La raison en est que nous devons traiter l'ensemble des processus, ce qui peut prendre un long moment, durant lequel `schedule()` peut être appelée sur un autre CPU et choisir un processus avec une qualité satisfaisante pour ce CPU, pendant que nous sur ce CPU on est forcé de recalculer. Ok, de l'aveu général c'est un inconvénient parce que pendant qu'on (sur ce CPU) choisit un processus avec la meilleure qualité, `schedule()` s'exécutant sur un autre CPU peut recalculer les priorités dynamiques.

14. A partir de ce moment on est certain que `next` pointe sur la tâche à ordonnancer, donc on initialise `next->has_cpu` à 1 et `next->processor` à `this_cpu`. Le `runqueue_lock` peut être levé.

15. Si on rebascule sur la même tâche (`next == prev`) alors on peut simplement réacquérir le verrou global du noyau et retourner, sans avoir à traiter tout ce qui concerne le matériel (registres, pile etc.) ni ce qui est lié à la VM (changer le répertoire de page, recalculer `active_mm` etc.).

16. La macro `switch_to()` est spécifique à l'architecture. Sur un i386, cela concerne a) la gestion du FPU, b) la gestion de la LDT, c) le rechargement des registres de segment, d) la gestion de la TSS et e) le rechargement des registres de déboguage.

3.4. Implémentation des listes chaînées Linux

Avant d'examiner l'implémentation des files d'attentes, nous devons nous familiariser avec l'implémentation standard des doubles listes chaînées de Linux. Les files d'attentes (comme beaucoup d'autres choses dans Linux) en font une utilisation importante et elles sont appelées dans le jargon « implémentation list.h » car le fichier le plus significatif est `include/linux/list.h`.

La structure de données fondamentale ici est `struct list_head` :

```
struct list_head {
    struct list_head *next, *prev;
    };

#define LIST_HEAD_INIT(name) { &(name), &(name) }

#define LIST_HEAD(name) \
        struct list_head name = LIST_HEAD_INIT(name)

#define INIT_LIST_HEAD(ptr) do { \
        (ptr)->next = (ptr); (ptr)->prev = (ptr); \
} while (0)

#define list_entry(ptr, type, member) \
        ((type *)((char *)(ptr)-(unsigned long)(&((type *)0)->member)))
```

```
#define list_for_each(pos, head) \
        for (pos = (head)->next; pos != (head); pos = pos->next)
```

Les trois premières macros sont utilisées pour initialiser une liste vide en faisant pointer `next` et `prev` sur elle même. Les restrictions syntaxiques du C rendent évidentes les conditions de leur utilisation — par exemple, `LIST_HEAD_INIT()` peut être utilisée pour l'initialisation des éléments de la structure lors de sa déclaration, la seconde peut être utilisée pour l'initialisation d'une variable statique et la troisième peut être utilisée dans une fonction.

La macro `list_entry()` donne accès aux éléments individuels de la liste, par exemple (dans `fs/file_table.c:fs_may_remount_ro()`):

```
struct super_block {
   ...
   struct list_head s_files;
   ...
} *sb = &some_super_block;

struct file {
   ...
   struct list_head f_list;
   ...
} *file;

struct list_head *p;

for (p = sb->s_files.next; p != &sb->s_files; p = p->next) {
     struct file *file = list_entry(p, struct file, f_list);
     do something to 'file'
}
```

Un bon exemple de l'utilisation de la macro `list_for_each()` se trouve dans l'ordonnanceur quand on parcourt la queue d'exécution en cherchant le processus de meilleure qualité :

```
static LIST_HEAD(runqueue_head);
struct list_head *tmp;
struct task_struct *p;

list_for_each(tmp, &runqueue_head) {
    p = list_entry(tmp, struct task_struct, run_list);
    if (can_schedule(p)) {
        int weight = goodness(p, this_cpu, prev->active_mm);
        if (weight > c)
            c = weight, next = p;
    }
}
```

Ici, `p->run_list` est déclaré comme `struct list_head run_list` dans la structure `task_struct` et sert d'ancrage à la liste. Le retrait ou l'ajout d'un élément à la liste (au début ou à la fin de la liste) est fait par les macros `list_del()/list_add()/list_add_tail()`. Les exemples ci-dessous ajoutent et retirent des tâches à la file d'exécution :

```
static inline void del_from_runqueue(struct task_struct * p)
{
        nr_running--;
        list_del(&p->run_list);
        p->run_list.next = NULL;
}

static inline void add_to_runqueue(struct task_struct * p)
{
```

```
        list_add(&p->run_list, &runqueue_head);
        nr_running++;
}

static inline void move_last_runqueue(struct task_struct * p)
{
        list_del(&p->run_list);
        list_add_tail(&p->run_list, &runqueue_head);
}

static inline void move_first_runqueue(struct task_struct * p)
{
        list_del(&p->run_list);
        list_add(&p->run_list, &runqueue_head);
}
```

3.5. Les files d'attente (Wait Queues)

Quand un processus demande au noyau de faire quelque chose qui n'est pas possible pour l'instant mais qui le sera plus tard, le processus est endormi et il est réveillé à un moment où la requête a plus de chances d'être satisfaite. L'un des mécanismes utilisés pour réaliser cela est appelé « file d'attente (wait queue) ».

L'implémentation Linux autorise le réveil sémantique en utilisant le drapeau `TASK_EXCLUSIVE`. Avec les files d'attentes, vous pouvez utiliser soit des files bien connues et leurs fonctions `sleep_on / sleep_on_timeout / interruptible_sleep_on / interruptible_sleep_on_timeout`, ou définir votre propre file d'attente et utiliser `add/remove_wait_queue` pour ajouter et supprimer des tâches vous même et `wake_up/wake_up_interruptible` pour les réveiller lorsque c'est nécessaire.

Un exemple du premier usage des files d'attentes est l'interaction entre l'allocateur de pages (dans `mm/page_alloc.c:__alloc_pages()`) et le démon noyau `kswapd` (dans `mm/vmscan.c:kswap()`), par le biais de la file d'attente `kswapd_wait`, déclarée dans `mm/vmscan.c`; le démon `kswapd` dort dans cette file, et il est réveillé à chaque fois que l'allocateur de pages a besoin de libérer des pages.

Un exemple d'utilisation d'une file d'attente autonome est l'interaction entre un processus utilisateur demandant des données via l'appel système *read(2)* et le noyau tournant dans un contexte d'interruption pour fournir les données. Le gestionnaire d'interruption peut ressembler à (`drivers/char/rtc_interrupt()` simplifié) :

```
static DECLARE_WAIT_QUEUE_HEAD(rtc_wait);

void rtc_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        spin_lock(&rtc_lock);
        rtc_irq_data = CMOS_READ(RTC_INTR_FLAGS);
        spin_unlock(&rtc_lock);
        wake_up_interruptible(&rtc_wait);
}
```

Alors, le gestionnaire d'interruption obtient les données en lisant un port d'entrée/sortie spécifique au périphérique (la macro`CMOS_READ()` réalise quelques instructions `outb/inb`) puis réveille tous ceux qui dorment dans la file d'attente `rtc_wait`.

Maintenant, l'appel système *read(2)* peut être implémenté comme :

```
ssize_t rtc_read(struct file file, char *buf, size_t count, loff_t *ppos)
{
        DECLARE_WAITQUEUE(wait, current);
        unsigned long data;
```

```
        ssize_t retval;

        add_wait_queue(&rtc_wait, &wait);
        current->state = TASK_INTERRUPTIBLE;
        do {
                spin_lock_irq(&rtc_lock);
                data = rtc_irq_data;
                rtc_irq_data = 0;
                spin_unlock_irq(&rtc_lock);

                if (data != 0)
                        break;

                if (file->f_flags & O_NONBLOCK) {
                        retval = -EAGAIN;
                        goto out;

                }
                if (signal_pending(current)) {
                        retval = -ERESTARTSYS;
                        goto out;
                }
                schedule();
        } while(1);
        retval = put_user(data, (unsigned long *)buf);
        if (!retval)
                retval = sizeof(unsigned long);

out:
        current->state = TASK_RUNNING;
        remove_wait_queue(&rtc_wait, &wait);
        return retval;
}
```

Ce qui se passe dans `rtc_read()` est :

1. On déclare un élément de file d'attente pointant sur le contexte du processus courant.
2. On ajoute cet élément à la file d'attente `rtc_wait`.
3. On marque le contexte courant comme `TASK_INTERRUPTIBLE` ce qui signifie qu'il ne sera pas réordonnancé à la fin de son prochain temps de sommeil.
4. On vérifie s'il y a des données disponibles ; s'il y en a, on s'interrompt, on copie les données dans le tampon utilisateur, on se marque `TASK_RUNNING`, on se retire nous même de la file d'attente et on retourne.
5. S'il n'y a pas encore de données, on regarde si l'utilisateur a spécifié une entrée/sortie non bloquante et si oui on échoue avec `EAGAIN` (qui est la même chose que `EWOULDBLOCK`)
6. Nous regardons également s'il y a un signal suspendu et si tel est le cas on demande aux « couches supérieures » de relancer l'appel système si nécessaire. Par « si nécessaire » je pense aux détails de la disposition du signal tels que spécifiés lors de l'appel système *sigaction(2)*.
7. Alors on « bascule hors de la tâche (switch out) », i.e. on s'endort, jusqu'à ce qu'on soit réveillé par la routine d'interruption. Si on ne s'est pas marqué nous même comme `TASK_INTERRUPTIBLE`, alors l'ordonnanceur peut nous ordonnancer avant que les données soient disponibles, ce qui cause des traitements inutiles.

Il vaut la peine de remarquer que l'utilisation des files d'attente rend plus facile d'implémenter l'appel système *poll(2)* :

```
static unsigned int rtc_poll(struct file *file, poll_table *wait)
{
        unsigned long l;

        poll_wait(file, &rtc_wait, wait);

        spin_lock_irq(&rtc_lock);
        l = rtc_irq_data;
        spin_unlock_irq(&rtc_lock);


        if (l != 0)
                return POLLIN | POLLRDNORM;
        return 0;
}
```

Tout le travail est fait par la fonction indépendante du matériel `poll_wait()` qui fait les manipulations nécessaires sur la file d'attente; il suffit de la faire pointer sur la file d'attente qui est réveillée par notre gestionnaire d'interruption spécifique au périphérique.

3.6. Les chronomètres du noyau (timers)

Maintenant portons notre attention sur les chronomètres du noyau. Ils sont utilisés pour reporter l'exécution d'une fonction particulière (appelée « timer handler ») à un instant donné dans le futur. La principale structure de données est `struct timer_list` déclarée dans `include/linux/timer.h` :

```
struct timer_list {
        struct list_head list;
        unsigned long expires;
        unsigned long data;
        void (*function)(unsigned long);
        volatile int running;
};
```

Le champ `list` sert à s'ancrer à la liste, en étant protégé par le verrou tournant `timer_list_lock`. Le champ `expires` est la valeur de `jiffies` qui provoque l'invocation de `function` avec le paramètre `data`. Le champ `running` est utilisé sur les multiprocesseurs pour tester si le « timer handler » est exécuté par un autre CPU.

Les fonctions `add_timer()` et `del_timer()` ajoutent et enlèvent un chronomètre donné à la liste. Quand un chronomètre expire, il est retiré automatiquement. Avant d'être utilisé, un chronomètre DOIT être initialisé par le biais de la fonction `init_timer()`. Et avant qu'il soit ajouté, les champs `function` et `expires` doivent être positionnés.

3.7. Bouts de listes (Bottom Halves)

Il est parfois raisonnable de séparer l'ensemble de ce qui doit être fait par le gestionnaire d'interruption en travail immédiat (i.e. acquitter l'interruption, mettre à jour les statistiques, et cætera) et en travail qui peut être remis à plus tard, pendant que les interruptions seront actives (i.e. faire des pré-traitements sur les données, réveiller les processus pour ces données, et cætera).

Les Bottom halves sont un vieux mécanisme pour différer l'exécution de tâches du noyau et elles existent depuis Linux 1.x. Dans Linux 2.0, un nouveau mécanisme a été ajouté, appelé « task queues », qui sera le sujet du prochain paragraphe.

Les Bottom halves sont sérialisées par le verrou tournant `global_bh_lock`, ce qui veut dire qu'il ne peut y avoir qu'un seul bottom half en exécution sur n'importe quel CPU à un instant donné. Cependant quand on essaie d'exécuter le gestionnaire, si `global_bh_lock` n'est pas disponible, le bottom half est marqué (i.e. ordonnancé) pour l'exécution — donc le traitement peut continuer, au contraire d'une boucle tournant sur `global_bh_lock`.

Il ne peut y avoir au total que 32 bottom halves enregistrés. Les fonctions nécessaires à la manipulation des bottom halves sont comme suit (toutes exportées dans les modules) :

- `void init_bh(int nr, void (*routine)(void))` : installe le gestionnaire de bottom half pointé par l'argument `routine` à l'emplacement `nr`. Les emplacements sont énumérés dans `include/linux/interrupt.h` sous la forme `XXXX_BH`, i.e. `TIMER_BH` ou `TQUEUE_BH`. Typiquement, une fonction d'initialisation d'un sous-système (`init_module()` pour les modules) installe les bottom half requis en utilisant cette fonction.
- `void remove_bh(int nr)` : fait le contraire de `init_bh()`, i.e. elle désinstalle le bottom half installé à l'emplacement `nr`. Il n'y pas de vérification d'erreur effectuée ici, alors, par exemple `remove_bh(32)` va faire paniquer (panic/Oops) le système. Typiquement, une routine de nettoyage d'un sous-système (`cleanup_module()` pour modules) utilise cette fonction pour libérer son emplacement qui pourra être réutilisé plus tard par un autre sous-système. (À FAIRE : ne serait il pas bien d'avoir un `/proc/bottom_halves` listant tous les bottom halves enregistrés sur le système ? Ce qui signifie que `global_bh_lock` doit être mis en lecture/écriture, évidemment)
- `void mark_bh(int nr)` : marque le bottom half à l'emplacement `nr` pour exécution. Typiquement, une routine d'interruption marquera son bottom half (d'où le nom!) pour exécution à un « moment plus sûr ».

Les Bottom halves sont des mini-tâches (tasklets) verrouillées globalement, donc la question « quand les routines bottom half sont-elles exécutées ? », se ramène à « quand les mini-tâches sont-elles exécutées ? ». Et la réponse est, à deux endroits : a) à chaque `schedule()` et b) à chaque chemin de retour d'interruption/appel système dans `entry.S` (À FAIRE : par conséquent, le cas `schedule()` est vraiment embêtant — c'est comme ajouter une autre interruption très très lente, pourquoi ne pas se débarrasser du label `handle_softirq` de `schedule()` ?).

3.8. Les files de tâches (Task Queues)

Les files de tâches peuvent être vues comme l'extension dynamique des vieux bottom halves. En fait, dans le code source elles sont parfois référencées comme « nouveaux » bottom halves. Plus précisément, les vieux bottom halves dont on a discuté dans les section précédentes ont les limitations suivantes :

1. Il y en a seulement un certain nombre (32).
2. Chaque bottom half peut être associé à un seul gestionnaire.
3. Les bottom halves sont dévorées par le verrou tournant mis pour qu'elles ne puissent pas bloquer.

Par contre, avec les files de tâches, un nombre arbitraire de fonctions peuvent être chaînées et exécutées les unes après les autres ultérieurement. On crée une nouvelle file de tâches en utilisant la macro `DECLARE_TASK_QUEUE()` et on y ajoute une tâche au moyen de la fonction `queue_task()`. La file de tâches peut alors être traitée en utilisant `run_task_queue()`. Au lieu de créer votre propre file de tâches (que vous devrez gérer manuellement), vous pouvez utiliser les files de tâches prédéfinies de Linux qui sont traitées à des moments bien précis :

1. *tq_timer* : la file de tâche des chronomètres, s'exécute à chaque interruption de chronomètre et quand on libère un périphérique tty (fermeture ou libération d'un périphérique terminal à demi ouvert). Tant que le gestionnaire de chronomètre s'exécute dans un contexte d'interruption, la tâche `tq_timer` s'exécute aussi dans un contexte d'interruption et donc ne peut pas bloquer.
2. *tq_scheduler* : la file de tâche de l'ordonnanceur, exécutée par l'ordonnanceur (et aussi quand on ferme un périphérique tty, comme `tq_timer`). Comme l'ordonnanceur a été exécuté dans le contexte du processus étant réordonnancé, les tâches de `tq_scheduler` peuvent faire ce

quelles veulent, i.e. bloquer, utiliser les données du contexte processus (mais pourquoi voudraient-elles le faire), et cætera.

3. *tq_immediate* : c'est un vrai bottom half `IMMEDIATE_BH`, donc les pilotes peuvent `queue_task(task, &tq_immediate)` (mettre une tâche dans la file) puis `mark_bh(IMMEDIATE_BH)` (la marquer) pour être exécutée dans un contexte d'interruption.

4. *tq_disk* : utilisé par les accès bas niveau aux périphériques bloc (et RAID) pour lancer les requêtes effectives. Cette file de tâches est exportée vers les modules mais ne doit être utilisée que dans les buts précis pour lesquels elle a été conçue.

A moins que le pilote utilise sa propre liste de tâches, il n'a pas besoin d'appeler `run_tasks_queues()` pour traiter la file, sauf dans les circonstances expliquées ci-dessous.

La raison pour laquelle les files de tâches `tq_timer/tq_scheduler` ne sont pas traitées seulement aux endroits habituels mais aussi ailleurs (la fermeture d'un périphérique tty en est un exemple) devient claire si on se rappelle que le pilote peut ordonnancer les tâches dans la file, et que ces tâches n'ont un sens que pendant qu'une instance particulière du périphérique reste valide — généralement jusqu'à ce que l'application le ferme. Donc, le pilote peut avoir besoin d'appeler `run_task_queue()` pour évacuer les tâches (et n'importe quoi d'autre) qu'il a mises dans la file, parce que les autoriser à s'exécuter plus tard n'aurait aucun sens — i.e. les structures de données utiles auraient été libérées/réutilisées par une instance différente. C'est la raison pour laquelle on voit `run_task_queue()` s'appliquer à `tq_timer` et `tq_scheduler` à d'autres endroits qu'une interruption de chronomètre et que `schedule()` respectivement.

3.9. Mini-tâches (Tasklets)

Pas encore, dans une révision future.

3.10. IRQ logicielles (softirq)

Pas encore, dans une révision future.

3.11. Comment les appels système sont-ils implémentés sur une architecture i386 ?

Il y a deux mécanismes sous Linux pour les implémenter :

- Les portes d'appel (call gates) lcall7/lcall27;
- l'interruption logicielle int 0x80.

Les programmes Linux natifs utilisent int 0x80 alors que les binaires d'autres UNIX (Solaris, UnixWare 7, et cætera) utilisent le mécanisme lcall7. Le nom historique « lcall7 » est inexact car il couvre aussi lcall27 (i.e. Solaris/x86), tandis que la fonction de prise en charge est appelée lcall7_func.

Quand le système démarre, la fonction `arch/i386/kernel/traps.c:trap_init()` est appelée pour configurer l'IDT de façon à ce que le vecteur 0x80 (de type 15, dpl 3) pointe à l'adresse d'entrée de system_call dans `arch/i386/kernel/entry.S`.

Quand une application dans l'espace utilisateur fait un appel système, les arguments sont passés via le registre et l'application exécute l'instruction « int 0x80 ». Ce qui provoque une trappe (trap) dans le mode noyau et le processeur saute au point d'entrée system_call dans `entry.S`. Voici ce qu'il fait :

1. il sauvegarde les registres ;

2. il positionne %ds et %es à KERNEL_DS, pour que toutes les références aux données (et autres segments) soient faites dans l'espace d'adressage du noyau ;

3. si la valeur de %eax est supérieure à `NR_syscalls` (actuellement 256), il échoue avec l'erreur `ENOSYS` ;

4. si la tâche est tracée (`tsk->ptrace & PF_TRACESYS`), il fait un traitement spécial, ceci pour supporter des programmes comme strace (analogue à *truss(1)* dans SVR4) ou les débogueurs ;

5. il appelle `sys_call_table+4*(syscall_number de %eax)`. Cette table est initialisée dans le même fichier (`arch/i386/kernel/entry.S`) pour pointer individuellement sur les gestionnaire d'appels systèmes, lesquels sous Linux sont (généralement) préfixés par `sys_`, i.e. `sys_open`, `sys_exit`, et cætera. Ces gestionnaires d'appels systèmes C vont trouver leurs arguments dans la pile où `SAVE_ALL` les a stockés.

6. il entre dans le « chemin de retour de l'appel système ». C'est un label séparé car il n'est pas utilisé seulement par int 0x80 mais aussi par lcall7, lcall27. Il est chargé des mini-tâches (incluant les bottom halves), de vérifier si un `schedule()` est nécessaire (`tsk->need_resched != 0`), de vérifier s'il y a des signaux suspendus et dans ce cas de les traiter.

Linux supporte jusqu'à 6 arguments pour les appels système. Ils sont passés dans %ebx, %ecx, %edx, %esi, %edi (et %ebp utilisé temporairement, voir `_syscall6()` dans `asm-i386/unistd.h`). Le numéro de l'appel système est passé via %eax.

3.12. Opérations atomiques

Il y a deux types d'opérations atomiques : bitmaps et `atomic_t`. Les bitmaps sont très pratiques pour maintenir le concept d'unités « allouées » ou « libres » pour de grands ensembles de données où chaque unité est identifiée par un nombre, par exemple les inodes libres ou les blocs libres. Ils sont aussi largement utilisés pour des verrouillages simples, par exemple pour fournir un accès exclusif à un périphérique ouvert. Un exemple de ceci peut être trouvé dans `arch/i386/kernel/microcode.c` :

```
/*
 *  Nb de bits dans microcode_status. (31 bits de plus pour le futur)
 */
#define MICROCODE_IS_OPEN  0  /* positionné lorsque le périphérique */
                              /* est utilisé */

static unsigned long microcode_status;
```

Il n'est pas nécessaire d'initialiser `micro-code_status` à 0 car la BSS est systématiquement remise à zéro sous Linux.

```
/*
 * Nous n'autorisons ici qu'un seul utilisateur à la fois pour ouvrir/fermer.
 */
static int microcode_open(struct inode *inode, struct file *file)
{
        if (!capable(CAP_SYS_RAWIO))
                return -EPERM;

        /* one at a time, please */
        if (test_and_set_bit(MICROCODE_IS_OPEN, &microcode_status))
                return -EBUSY;

        MOD_INC_USE_COUNT;
        return 0;
}
```

Les opérations sur les bitmaps sont :

- *void set_bit(int nr, volatile void *addr)* : fixe le bit `nr` dans le bitmap pointé par `addr`.
- `void clear_bit(int nr, volatile void *addr)` : met à 0 le bit `nr` dans le bitmap pointé par `addr`.
- `void change_bit(int nr, volatile void *addr)` : change le bit `nr` (s'il est positionné (1), on le met à 0, s'il ne l'est pas, on le positionne) dans le bitmap pointé par `addr`.
- `int test_and_set_bit(int nr, volatile void *addr)` : positionne le bit `nr` atomiquement et retourne l'ancienne valeur.
- `int test_and_clear_bit(int nr, volatile void *addr)` : met à 0 le bit `nr` atomiquement et retourne l'ancienne valeur.
- `int test_and_change_bit(int nr, volatile void *addr)` : change le bit `nr` atomiquement et retourne l'ancienne valeur.

Ces opérations utilisent la macro `LOCK_PREFIX`, qui s'évalue au préfixe de l'instruction de verrouillage du bus (lock) sur les noyaux multiprocesseur et à rien sur de l'uniprocesseur. Ce qui garantit l'atomicité de l'accès dans les environnements multiprocesseurs.

Parfois les manipulations de bits ne sont pas pratiques, on aimerait mieux utiliser des opérations arithmétiques — addition, soustraction, incrémentation, décrémentation. Un cas typique en est les compteurs de références (i.e. pour les inodes). Cette facilité est fournie par le type de donnée `atomic_t` et les opérations suivantes :

- *atomic_read(&v)* : lit la valeur de la variable `atomic_t v`,
- *atomic_set(&v, i)* : fixe la valeur de la variable `atomic_t v` à l'entier `i`,
- *void atomic_add(int i, volatile atomic_t *v)* : ajoute l'entier `i` à la valeur de la variable atomique pointée par `v`,
- *void atomic_sub(int i, volatile atomic_t *v)* : soustrait l'entier `i` de la valeur de la variable atomique pointée par `v`,
- *int atomic_sub_and_test(int i, volatile atomic_t *v)* : soustrait l'entier `i` de la valeur de la variable atomique pointée par `v` ; retourne 1 si la nouvelle valeur est 0, retourne 0 autrement.
- *void atomic_inc(volatile atomic_t *v)* : incrémente la valeur de 1.
- *void atomic_dec(volatile atomic_t *v)* : décrémente la valeur de 1.
- *int atomic_dec_and_test(volatile atomic_t *v)* : décrémente la valeur ; retourne 1 si la nouvelle valeur est 0, retourne 0 sinon.
- *int atomic_inc_and_test(volatile atomic_t *v)* : incrémente la valeur ; retourne 1 si la nouvelle valeur est 0, retourne 0 sinon.
- *int atomic_add_negative(int i, volatile atomic_t *v)* : ajoute la valeur de `i` à `v` et retourne 1 si le résultat est négatif. Retourne 0 si le résultat est supérieur ou égal à 0. Cette opération est utilisée pour implémenter les sémaphores.

3.13. Verrous tournants, verrous tournants en lecture/écriture et verrous tournants gros lecteurs

Depuis les premiers jours du support Linux (début des années 90s), les développeurs ont été

confrontés au classique problème de l'accès à des données partagées entre différents types de contextes (processus utilisateur vs interruption) et les différentes instances d'un même contexte sur plusieurs CPU.

Le support multiprocesseur et à rien sur de l'UP. Ce qui garantit l'atomicité a été ajouté à Linux 1.3.42 le 15 Novembre 1995 (le patch original avait été fait en octobre de la même année).

Si une région de code critique peut être exécutée soit dans un contexte de processus soit dans un contexte d'interruption, alors la façon de le protéger sur de l'UP est d'utiliser les instructions `cli/sti` :

```
unsigned long flags;

save_flags(flags);
cli();
/* critical code */
restore_flags(flags);
```

Tandis que cela marche très bien sur de l'UP, ce n'est bien évidement d'aucune utilité sur du multiprocesseur et à rien sur de l'UP. Ce qui garantit l'atomicité car la même séquence de code peut être exécutée simultanément sur un autre processeur, et pendant que `cli()` fournit une protection contre des accès concurrents avec un contexte d'interruption sur chaque CPU individuellement, elle ne fournit pas de protection du tout contre la concurrence entre contextes sur des CPU différents. C'est là que les verrous tournants sont utiles.

Ils y a trois types de verrous tournants (spinlocks) : basique (vanilla), lecture-écriture (read-write) et gros-lecteur (big-reader). Les verrous tournants en lecture-écriture doivent être utilisés quand il y a une tendance naturelle à avoir « beaucoup de lecteurs et peu d'écrivains ». Un exemple de ceci est l'accès à la liste des systèmes de fichiers enregistrés (voir `fs/super.c`). Cette liste est gardée par `file_systems_lock`, un verrou tournant en lecture-écriture parce qu'on a besoin d'un accès exclusif quand on enregistre/dés-enregistre un système de fichiers, mais n'importe quel processus peut lire le fichier `/proc/filesystems` ou utiliser l'appel système *sysfs(2)* pour forcer un examen systématique en lecture seule de la liste des systèmes de fichiers. C'est là que l'utilisation des verrous tournant en lecture-écriture prend tout son sens. Avec un verrou tournant en lecture-écriture, on peut avoir plusieurs lecteurs en même temps mais un seul écrivain et il ne peut pas y avoir de lecteurs quand il y a un écrivain. En passant, se serait bien si les nouveaux lecteurs n'obtenaient pas de verrous pendant qu'un écrivain essaie d'en poser un, i.e. si Linux pouvait gérer correctement le problème de la frustration éventuelle d'un écrivain par plusieurs lecteurs. Il faudrait pour cela que les lecteurs soient bloqués pendant qu'un écrivain essaie de poser un verrou. Ce n'est pas le cas actuellement et il n'est pas évident que ça le devienne — l'argument opposé est — les lecteurs posent généralement un verrou pour un temps très court alors devraient-ils réellement être privés lorsque l'écrivain pose un verrou pour une période potentiellement longue ?

Les verrous tournant « gros-lecteur » sont une forme de verrou tournant en lecture-écriture très optimisé pour des accès en lecture très légers, avec une pénalisation des écritures. Il y a un nombre limité de verrous gros-lecteur — actuellement il en existe seulement deux, dont un n'est utilisé que sur le sparc64 (global irq) et l'autre n'est utilisé que pour le réseau. Dans tous les autres cas où les modèles d'accès ne collent pas à l'un de ces deux scénarios, on doit utiliser des verrous tournant basiques. Vous ne pouvez pas bloquer pendant que n'importe quelle sorte de verrou tournant est maintenu.

Les verrous tournants sont fournis en trois saveurs : simple (plain), `_irq()` et `_bh()`.

1. `spin_lock()/spin_unlock()` simple : si vous savez que les interruptions sont toujours inactivées ou si vous n'êtes pas en concurrence avec des contextes d'interruption (i.e. depuis un gestionnaire d'interruption), alors vous pouvez les utiliser. Ils ne touchent pas à l'état d'interruption du CPU courant.

2. `spin_lock_irq()/spin_unlock_irq()` : si vous savez que les interruptions sont toujours activées, alors vous pouvez utiliser cette version, qui simplement désactive (au verrouillage) et réactive (au déverrouillage) les interruptions sur le CPU courant. Par exemple, `rtc_read()` utilise `spin_lock_irq(&rtc_lock)` (les interruptions sont toujours acti-

vées dans `read()`) alors que `rtc_interrupt()` utilise `spin_lock(&rtc_lock)` (les interruptions sont toujours désactivées dans le gestionnaire d'interruption). Remarquez que `rtc_read()` utilise `spin_lock_irq()` et pas le plus générique `spin_lock_irqsave()` parce que les interruptions sont toujours activées lorsqu'on entre dans n'importe quel appel système, .

3. `spin_lock_irqsave()/spin_unlock_irqrestore()` : la forme la plus robuste, à utiliser quand l'état d'interruption n'est pas connu, mais seulement si les interruptions ont de l'importance, i.e. il n'y a pas de raison de l'utiliser si nos gestionnaires d'interruption n'exécutent pas de code critique.

La raison pour laquelle vous ne pouvez pas utiliser de `spin_lock()` simples si vous êtes en concurrence avec des gestionnaires d'interruption, c'est que si vous en posez un et qu'une interruption survienne sur le même processeur, il attendra indéfiniment la levée du verrou : le propriétaire du verrou ayant été interrompu, ne continuera pas tant que le gestionnaire d'interruption n'aura pas retourné.

L'utilisation la plus courante des verrous tournant est pour accéder à des structures de données partagées entre les contextes des processus utilisateur et des gestionnaires d'interruption :

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;

my_ioctl()
{
        spin_lock_irq(&my_lock);
        /* section critique */
        spin_unlock_irq(&my_lock);
}

my_irq_handler()
{
        spin_lock(&lock);
        /* section critique */
        spin_unlock(&lock);
}
```

Il y a deux choses à remarquer dans cet exemple :

1. Le contexte de processus, représenté ici par une méthode typique de pilote — `ioctl()` (les arguments et les valeurs de retour ont été omis pour simplifier), doit utiliser `spin_lock_irq()` parce qu'il sait que les interruptions sont toujours activées pendant l'exécution de la méthode `ioctl()` du périphérique.

2. Le contexte d'interruption, représenté ici par `my_irq_handler()` (on omet encore les arguments par souci de clarté) peut utiliser la forme simple `spin_lock()` parce que les interruptions sont désactivées à l'intérieur du gestionnaire d'interruption.

3.14. Les sémaphores et les sémaphores en lecture/écriture

Parfois, pendant l'accès à une structure de données partagées, il faut effectuer des opérations qui peuvent bloquer, par exemple la copie de données dans l'espace utilisateur. La primitive de verrouillage disponible pour de tels scénarios sous Linux est appelée sémaphore. Il y a deux types de sémaphores : basique et lecture/écriture. En fonction de la valeur initiale du sémaphore, on peut obtenir soit une exclusion mutuelle (valeur initiale de 1) soit un type d'accès plus sophistiqué.

Les sémaphores en lecture-écriture diffèrent des sémaphores basiques de la même façon que les verrous tournant en lecture-écriture diffèrent des verrous tournant basiques : on peut avoir plusieurs lec-

teurs au même instant mais seulement un écrivain et il ne peut y avoir de lecteurs pendant qu'il y a des écrivains — i.e. l'écrivain bloque tous les lecteurs et les nouveaux lecteurs sont bloqués tant qu'un écrivain attend.

De plus, les sémaphores basiques peuvent être interrompus — juste en utilisant les opérations `down/up_interruptible()` au lieu des simples `down()/up()` et en vérifiant la valeur retourné par `down_interruptible()` : ce ne sera pas zéro si l'opération est interrompue.

Utiliser les sémaphores pour l'exclusion mutuelle est la solution idéale dans le cas où une section de code critique peut appeler par référence des fonction inconnues enregistrées par d'autres sous-systèmes/modules, i.e. l'appelant ne peut pas savoir à priori si la fonction bloque ou non.

Un exemple simple de l'utilisation d'un sémaphore dans `kernel/sys.c`, est l'implémentation des appels systèmes *gethostname(2)/sethostname(2)*.

```
asmlinkage long sys_sethostname(char *name, int len)
{
        int errno;

        if (!capable(CAP_SYS_ADMIN))
                return -EPERM;
        if (len < 0 || len > __NEW_UTS_LEN)
                return -EINVAL;
        down_write(&uts_sem);
        errno = -EFAULT;
        if (!copy_from_user(system_utsname.nodename, name, len)) {
                system_utsname.nodename[len] = 0;
                errno = 0;
        }
        up_write(&uts_sem);
        return errno;
}

asmlinkage long sys_gethostname(char *name, int len)
{
        int i, errno;

        if (len < 0)
                return -EINVAL;
        down_read(&uts_sem);
        i = 1 + strlen(system_utsname.nodename);
        if (i > len)
                i = len;
        errno = 0;
        if (copy_to_user(name, system_utsname.nodename, i))
                errno = -EFAULT;
        up_read(&uts_sem);
        return errno;
}
```

Les points à remarquer dans cet exemple sont :

1. Les fonctions peuvent bloquer pendant la copie de donnée depuis/vers l'espace utilisateur dans `copy_from_user()/copy_to_user()`. Donc elles ne peuvent utiliser aucune forme de verrou tournant ici.

2. Le type de sémaphore choisi est lecture-écriture plutôt que basique parce qu'il peut y avoir beaucoup de requêtes *gethostname(2)* concurrentes qui n'ont pas besoin d'être mutuellement exclusives.

Bien que l'implémentation des sémaphores et des sémaphores lecture-écriture de Linux soit très sophistiquée, on peut imaginer des scénarios qui ne sont pas encore implémentés, par exemple il n'y a pas de concept de sémaphores lecture-écriture interruptibles. C'est évidemment parce qu'il n'y a pas

de situation du monde réel qui nécessite ces implémentations de primitives exotiques.

3.15. Le support noyau des modules chargeables

Linux est un système d'exploitation monolithique et malgré toute la publicité moderne à propos des « avantages » offerts par les systèmes d'exploitation basés sur des micro-noyaux, la vérité reste (dixit Linus Torvalds lui même) :

```
… considérer la transmission de messages comme l'opération de
base d'un système d'exploitation n'est qu'un exercice d'informatique
masturbatoire. Peut-être que cela vous procure du plaisir, mais vous
n'obtenez rien de FAIT dans la réalité.
```

```
(… message passing as the fundamental operation of the OS is just
an exercise in computer science masturbation. It may feel good, but you
don't actually get anything DONE.)
```

Par conséquent, Linux est et restera toujours basé sur une conception monolithique, ce qui signifie que tous les sous-systèmes s'exécutent dans le même mode privilégié et partagent le même espace d'adressage ; la communication entre eux s'accomplit par le biais d'appels aux fonctions C habituelles.

Cependant, bien que répartir les fonctionnalités du noyau dans des « processus » séparés comme cela se fait dans les micro-noyaux soit absolument une mauvaise idée, découper en modules noyau chargeables dynamiquement à la demande peut être souhaitable dans certaines circonstances (i.e. sur les machines avec peu de mémoire ou pour les noyaux d'installation qui autrement contiendraient des pilotes de périphériques ISA auto-testés qui sont mutuellement exclusifs). La décision d'inclure le support pour les modules chargeables est prise à la compilation et est déterminée par l'option `CONFIG_MODULES`. Le support pour le chargement automatique des modules via le mécanisme `request_module()` est une option de compilation séparée (`CONFIG_KMOD`).

Les fonctionnalités suivantes peuvent être implémentées en modules chargeables sous Linux :

1. Pilotes de périphériques caractère et bloc, incluant des pilotes de périphérique divers.
2. Les disciplines des lignes de terminal.
3. Les fichiers virtuels (réguliers) dans `/proc` et dans devfs (i.e. `/dev/cpu/microcode` vs `/dev/misc/microcode`).
4. Les formats des fichiers binaires (i.e. ELF, aout, et cætera).
5. Les domaines d'exécution (i.e. Linux, UnixWare7, Solaris, et cætera).
6. Les systèmes de fichiers.
7. Les communications inter processus System V.

Il n'y a que peu de choses qui ne peuvent pas être implémentées comme des modules sous Linux (probablement parce que les rendre modulaires n'aurait aucun sens) :

1. les algorithmes d'ordonnancement,
2. les politiques de la VM (mémoire virtuelle),
3. le cache tampon (Buffer cache), le cache de page (page cache) et les autres caches.

Linux fournit plusieurs appels systèmes pour aider au chargement des modules :

1. `caddr_t create_module(const char *name, size_t size)` : alloue `size` octets en utilisant `vmalloc()` et place une structure de module au début de ceux ci. Ce nouveau module est ensuite lié à la liste commençant par module_list. Seul un processus avec `CAP_SYS_MODULE` peut invoquer cet appel système, les autres se verront retourner `EPERM`.
2. `long init_module(const char *name, struct module *image)` : charge et reloge l'image du module, ce qui provoque l'appel de la fonction d'initialisation du module. Seul un processus avec `CAP_SYS_MODULE` peut invoquer cet appel système, les autres se verront retourner `EPERM`.
3. `long delete_module(const char *name)` : essaie de décharger le module. Si `name == NULL`, essaie de décharger tous les modules inutilisés.
4. `long query_module(const char *name, int which, void *buf, size_t bufsize, size_t *ret)` : renvoie les informations concernant un module (ou concernant tous les modules).

L'interface de commande disponible pour les utilisateurs consiste en :

- *insmod* : insère un module seul.
- *modprobe* : insère un module en incluant tous les modules dont il dépend.
- *rmmod* : retire un module.
- *modinfo* : affiche des informations concernant le module, i.e. auteur, description, paramètres du module, et cætera.

On peut charger manuellement un module en utilisant *insmod* ou *modprobe*, mais le module peut aussi être chargé automatiquement par le noyau quand une fonction particulière est requise. L'interface du noyau prévue pour cela est une fonction appelée `request_module(name)` qui est exportée vers les modules, ainsi les modules peuvent eux-mêmes charger d'autres modules. `request_module(name)` crée en interne un thread noyau qui « execs »-ute dans l'espace utilisateur la commande *modprobe -s -k nom_du_module*, en utilisant l'interface noyau standard `exec_usermodehelper()` (qui est aussi exportée vers les modules). La fonction renvoie 0 en cas de succès, mais il n'est généralement pas utile de tester le code de retour de `request_module()`. Il vaut mieux utiliser ce modèle :

```
if (check_some_feature() == NULL)
    request_module(module);
if (check_some_feature() == NULL)
    return -ENODEV;
```

Par exemple, `fs/block_dev.c:get_blkfops()` suit ce modèle pour charger le module `block-major-N` quand un essai est fait pour ouvrir le périphérique bloc de majeur `N`. évidemment, il n'y a pas de module appelé `block-major-N` (Les développeurs Linux choisissent des noms sensés pour leurs modules), ce nom est associé au bon module en utilisant le fichier `/etc/modules.conf`. Cependant, pour les plus connus des nombres majeur (et pour d'autre modules) les commandes *modprobe/insmod* savent quels modules réels charger sans avoir besoin d'un alias explicite dans `/etc/modules.conf`.

Un bon exemple de chargement d'un module est dans l'appel système *mount(2)*. L'appel système *mount(2)* prend un type de système de fichier comme argument sous la forme d'une chaîne de caractères que `fs/super.c:do_mount()` passe ensuite à `fs/super.c:get_fs_type()` :

```
static struct file_system_type *get_fs_type(const char *name)
{
        struct file_system_type *fs;

        read_lock(&file_systems_lock);
        fs = *(find_filesystem(name));
        if (fs && !try_inc_mod_count(fs->owner))
                fs = NULL;
        read_unlock(&file_systems_lock);
        if (!fs && (request_module(name) == 0)) {
                read_lock(&file_systems_lock);
                fs = *(find_filesystem(name));
                if (fs && !try_inc_mod_count(fs->owner))
                        fs = NULL;
                read_unlock(&file_systems_lock);
        }
        return fs;
}
```

Une ou deux choses à remarquer dans cette fonction :

1. D'abord on essaie de trouver le système de fichier dont le nom est donné parmi ceux déjà enregistrés. C'est fait sous la protection de `file_systems_lock` posé pour lire (nous ne sommes pas en train de modifier la liste de systèmes de fichier enregistrés).

2. Si un tel système de fichier est trouvé, alors on essaie d'obtenir une nouvelle référence sur lui en incrémentant le compteur maintenu par le module. Ce qui retourne toujours 1 pour les modules liés statiquement ou pour les modules qui ne sont pas enlevés du noyau. Si `try_inc_mod_count()` retourne 0, alors on considère que c'est une erreur — i.e. si le module est là mais en train d'être retiré, c'est pareil que s'il n'était pas là du tout.

3. Nous enlevons le `file_systems_lock` parce que l'opération suivante (`request_module()`) sera une opération bloquante, donc on ne peut pas maintenir un verrou tournant dessus. Maintenant, dans ce cas particulier, il aurait fallu enlever le `file_systems_lock` de toutes façons, même en étant sûr que `request_module()` ne soit pas bloquant et que le chargement du module s'exécute atomiquement dans le même contexte. C'est parce que la fonction d'initialisation du module va tenter d'appeler `register_filesystem()`, qui va poser le même verrou tournant en lecture-écriture `file_systems_lock` pour écrire.

4. Si l'essai de chargement est réussi, alors on pose le verrou tournant `file_systems_lock` et on essaie de localiser le système de fichiers nouvellement enregistré dans la liste. Remarquez que c'est un peu faux parce qu'il est en principe possible pour un bug dans la commande modprobe de causer un coredump après avoir réussi à charger le module demandé, dans ce cas `request_module()` va échouer même si le nouveau système de fichiers est enregistré, et `get_fs_type()` ne le trouvera pas.

5. Si le système de fichier est trouvé et que nous sommes capables d'obtenir une référence sur lui, on la retourne. Autrement on renvoie NULL.

Quand un module est chargé dans le noyau, il peut référencer tous les symboles exportés comme publics par le noyau en utilisant la macro `EXPORT_SYMBOL()` ou par les autres modules actuellement chargés. Si le module utilise des symboles d'un autre module, il est marqué comme dépendant de ce module durant le recalcul des dépendances, effectué à l'amorçage par la commande *depmod -a* (par exemple après l'installation d'un nouveau noyau).

Habituellement, l'ensemble des modules doivent correspondre à la version des interfaces noyau qu'ils utilisent, ce qui sous Linux signifie simplement « la version du noyau » car il n'y a pas en général de mécanisme de gestion de version de l'interface noyau. Cependant il existe une fonctionnalité limitée appelée « module versioning » ou `CONFIG_MODVERSIONS` qui permet d'éviter de recompiler les modules quand on change de noyau. Ce qui ce passe ici c'est que la table des symboles noyau est traitée différemment pour les accès internes et pour l'accès depuis les modules. Les élé-

ments de la partie publique (i.e. exportée) de la table des symboles sont construits en faisant des sommes de contrôle 32bit des déclarations C. Donc, pour résoudre un symbole utilisé par un module pendant le chargement, le chargeur doit faire correspondre la représentation complète du symbole y compris la somme de contrôle ; il refusera de charger le module si les symboles différent. Ce qui se produit seulement quand à la fois le noyau et le module sont compilés avec le « module versioning » activé. Si l'un d'entre eux utilise le nom originel du symbole, le chargeur essaie simplement de faire correspondre la version déclarée par le module et celle exportée par le noyau et refuse de charger le module si elles différent.

4. Système de fichiers virtuel (Virtual Filesystem : VFS)

4.1. Le cache inode et les interactions avec le Dcache

Dans le but de supporter de multiples types de systèmes de fichiers, Linux contient un niveau d'interface noyau spécial appelé VFS (Virtual Filesystem Switch ou commutateur de systèmes de fichiers virtuels), similaire à l'interface vnode/vfs trouvé dans les dérivés de SVR4 (il provient à l'origine des implémentations BSD et Sun).

Le cache inode Linux est implémenté par un seul fichier, `fs/inode.c`, qui consiste en 977 lignes de code. Il est intéressant de noter qu'il n'y a pas eu tant de changements ces 5-7 dernières années : on peut encore reconnaître un peu de code en comparant la dernière version avec, disons, la 1.3.42.

Voici la structure du cache inode de Linux :

1. Une table de hachage globale, `inode_hashtable`, où chaque inode est haché sur la valeur du pointeur du super-bloc et le numéro de l'inode codé sur 32 bits. Les inodes sans super-bloc (`inode->i_sb == NULL`) sont ajoutés à une liste doublement chaînée ancrée sur `anon_hash_chain`. Comme exemple d'inodes anonymes, on a les sockets créées par `net/socket.c:sock_alloc()`, en appelant `fs/inode.c:get_empty_inode()`.

2. Une liste globale du type utilisé (`inode_in_use`), qui contient les inodes valides avec `i_count>0` et `i_nlink>0`. Les inodes nouvellement alloués par `get_empty_inode()` et `get_new_inode()` sont ajoutés à la liste `inode_in_use`.

3. Une liste globale du type inutilisé (`inode_unused`), qui contient les inodes valides avec `i_count = 0`.

4. Une liste par super-bloc du type modifié (dirty) (`sb->s_dirty`) qui contient les inodes valides avec `i_count>0`, `i_nlink>0` et `i_state & I_DIRTY`. Quand l'inode est marqué modifié, il est ajouté à la liste `sb->s_dirty` s'il est aussi haché. Maintenir une liste d'inodes modifiés par super-bloc permet de synchroniser les inodes rapidement.

5. Un cache inode propre — un cache SLAB appelé `inode_cachep`. Quand les objets inode sont alloués et libérés, ils sont pris et remis dans le cache SLAB.

Les listes de types sont ancrées sur `inode->i_list`, la table de hachage sur `inode->i_hash`. Chaque inode peut être dans une table de hachage et dans une seule liste de type (utilisé, inutilisé, modifié).

Toutes ces listes sont protégées par un unique verrou tournant : `inode_lock`.

Le sous-système de cache inode est initialisé quand la fonction `inode_init()` est appelée depuis `init/main.c:start_kernel()`. La fonction est marquée `__init`, ce qui veut dire que son code est supprimé plus tard. On lui passe un seul argument — le nombre de pages physiques du système. C'est ainsi que le cache inode peut se configurer lui même en fonction de la quantité de mémoire disponible, i.e. créer une plus grande table de hachage s'il y a assez de mémoire.

Il n'y a qu'une seule information statistique à propos du cache inode, le nombre d'inodes inutilisés,

stocké dans `inodes_stat.nr_unused` et accessible aux programmes utilisateurs par les fichiers `/proc/sys/fs/inode-nr` et `/proc/sys/fs/inode-state`.

On peut examiner une de ces liste avec *gdb* s'exécutant sur un noyau en activité :

```
(gdb) printf "%d\n", (unsigned long)(&((struct inode *)0)->i_list)
8
(gdb) p inode_unused
$34 = 0xdfa992a8
(gdb) p (struct list_head)inode_unused
$35 = {next = 0xdfa992a8, prev = 0xdfcdd5a8}
(gdb) p ((struct list_head)inode_unused).prev
$36 = (struct list_head *) 0xdfcdd5a8
(gdb) p (((struct list_head)inode_unused).prev)->prev
$37 = (struct list_head *) 0xdfb5a2e8
(gdb) set $i = (struct inode *)0xdfb5a2e0
(gdb) p $i->i_ino
$38 = 0x3bec7
(gdb) p $i->i_count
$39 = {counter = 0x0}
```

Remarquez que la valeur 8 a été déduite de l'adresse 0xdfb5a2e8 pour obtenir l'adresse de `struct inode` (0xdfb5a2e0) d'après la définition de la macro `list_entry()` dans `include/linux/list.h`.

Pour comprendre comment le cache inode fonctionne, nous allons suivre la vie de l'inode d'un fichier régulier sur un système de fichier ext2 quand il est ouvert et fermé :

```
fd = open("file", O_RDONLY);
close(fd);
```

L'appel système *open(2)* est implémenté dans la fonction `fs/open.c:sys_open` et le travail effectif est réalisé par la fonction `fs/open.c:filp_open()`, qui est divisée en deux parties :

1. `open_namei()` : remplit la structure nameidata contenant les structures dentry et vfsmount.
2. `dentry_open()` : étant donnés dentry and vfsmount, cette fonction alloue une nouvelle `struct file` et les lie ensemble ; elle invoque aussi la méthode `f_op->open()` spécifique au système de fichiers, qui a été positionnée dans `inode->i_fop` lors de la lecture de l'inode dans `open_namei()` (qui fournit l'inode via `dentry->d_inode`).

La fonction `open_namei()` interagit avec le cache dentry via `path_walk()`, qui à son tour appelle `real_lookup()`, qui invoque la méthode `inode_operations->lookup()` spécifique au système de fichiers. Le rôle de cette méthode est de trouver l'entrée dans le répertoire parent qui correspond au nom et ensuite faire `iget(sb, ino)` pour avoir l'inode correspondant — ce qui nous amène dans le cache inode. Quand l'inode est lu, dentry est instancié grâce à `d_add(dentry, inode)`. Pendant que nous y sommes, remarquez que pour les systèmes de fichiers de style UNIX qui ont adopté le concept du numéro d'inode sur disque, il revient à la méthode de recherche de gérer l'ordre des octets (endianness) spécifique au format du CPU, i.e. si le numéro de l'inode d'entrée du répertoire en binaire (fs-specific) est au format 32 bits petit boutien (little-endian) on peut faire :

```
unsigned long ino = le32_to_cpu(de->inode);
inode = iget(sb, ino);
d_add(dentry, inode);
```

Ainsi, quand on ouvre un fichier, on utilise `iget(sb, ino)` qui en réalité est `iget4(sb, ino, NULL, NULL)`, et on :

1. Essaie de trouver un inode correspondant au super-bloc et un numéro d'inode dans la table de hachage sous la protection de `inode_lock`. Si l'inode est trouvé, son compteur de références (`i_count`) est incrémenté; s'il était à 0 avant l'incrémentation et que l'inode n'est pas utilisé, il est retiré de tous les types de liste (`inode->i_list`) dont il fait actuellement partie (il doit être dans la liste `inode_unused`, bien sûr) et inséré dans la liste de type `inode_in_use`; finalement, `inodes_stat.nr_unused` est décrémenté.

2. Si l'inode est actuellement verrouillé, on attend qu'il soit déverrouillé pour que `iget4()` soit sûr de renvoyer un inode non verrouillé.

3. Si l'inode n'a pas été trouvé dans la table de hachage, c'est que nous rencontrons cet inode pour la première fois, donc on appelle `get_new_inode()`, en lui passant le pointeur sur l'endroit de la table de hachage où l'inode doit être inséré.

4. `get_new_inode()` alloue un nouvel inode depuis le cache SLAB `inode_cachep` mais cette opération peut bloquer (allocation `GFP_KERNEL`), alors elle doit enlever le verrou tournant `inode_lock` qui protège la table de hachage. Dès qu'elle a enlevé le verrou, elle peut réessayer de chercher l'inode dans la table de hachage ; s'il est trouvé cette fois ci, elle le renvoie (après incrémentation de la référence par `__iget`) et détruit celui qui a été alloué entre temps. Si elle ne le trouve toujours pas dans la table de hachage, on utilisera le nouvel inode que l'on vient juste d'allouer ; celui-ci est initialisé aux valeurs voulues et la méthode `sb->s_op->read_inode()` spécifique au système de fichiers est invoquée pour renseigner le reste de l'inode. Ce qui nous ramène du cache inode au code du système de fichier — rappelez vous que nous venions du cache inode lorsque la méthode `lookup()` spécifique au système de fichiers a invoqué `iget()`. Pendant que la méthode `s_op->read_inode()` lit l'inode sur le disque, l'inode est verrouillé (`i_state = I_LOCK`); il est déverrouillé après le retour de la méthode `read_inode()` et tous les processus qui l'attendaient sont alors réveillés.

Maintenant, regardons ce qui ce passe quand on ferme le descripteur de ce fichier. L'appel système *close(2)* est implémenté dans la fonction `fs/open.c:sys_close()`, qui appelle `do_close(fd, 1)` qui annule (remplace par NULL) le descripteur du fichier dans la table des descripteur des fichiers processus et invoque la fonction `filp_close()` qui effectue la plus grande partie du travail. Les choses intéressantes se passent dans `fput()`, qui vérifie si ce descripteur était la dernière référence au fichier, et si c'est le cas appelle `fs/file_table.c:_fput()` qui appelle `__fput()`, là ou les interactions avec dcache ont lieu (donc avec le cache inode aussi — rappelez vous que dcache est le maître du cache inode!). `fs/dcache.c:dput()` fait un `dentry_iput()` qui nous ramène au cache inode via `iput(inode)`, alors essayons de comprendre `fs/inode.c:iput(inode)` :

1. Si le paramètre passé est NULL, nous ne faisons rien du tout et nous retournons.

2. S'il y a une méthode `sb->s_op->put_inode()` spécifique au système de fichiers, elle est invoquée immédiatement sans verrou tournant (ainsi elle peut bloquer).

3. Le verrou tournant `inode_lock` est posé et `i_count` est décrémenté. Si ce n'était PAS la dernière référence à cet inode, alors on vérifie simplement qu'il n'y a pas trop de références à lui auquel cas `i_count` pourrait boucler autour des 32 bits qui lui sont alloués, et si oui on affiche un avertissement et retourne. Remarquez que l'on appelle `printk()` pendant que le verrou tournant `inode_lock` est posé — c'est bien car `printk()` ne peut jamais bloquer, de plus elle peut être appelée dans absolument tous les contextes (même dans les gestionnaires d'interruption!).

4. Si c'était la dernière référence active, alors il y a encore un peu de travail à faire.

Le travail effectué par `iput()` sur la dernière référence de l'inode est relativement complexe, alors on lui consacrera une liste propre :

1. Si `i_nlink == 0` (i.e. le fichier a été délié (unlink) pendant qu'il était ouvert) alors l'inode est retiré de la table de hachage et de la liste de son type ; s'il reste des pages de données dans

le cache de page pour cet inode, elles sont enlevées au moyen de `truncate_all_inode_pages(&inode->i_data)`. Puis la méthode spécifique au système de fichiers `s_op->delete_inode()` est invoquée, qui, comme son nom l'indique, détruit la copie de l'inode sur le disque. S'il n'y a pas de méthode `s_op->delete_inode()` enregistrée dans le système de fichiers (i.e. mémoire (ramfs)) on appelle `clear_inode(inode)`, qui invoque `s_op->clear_inode()` si elle est enregistrée et si l'inode correspond à un périphérique bloc, le compteur de références de ce périphérique est enlevé par `bdput(inode->i_bdev)`.

2. Si `i_nlink != 0`, alors on cherche s'il y a d'autres inodes dans la même valeur de hachage et s'il n'y en a pas, c'est que l'inode n'est pas utilisé, on l'efface de la liste de son type et on l'ajoute à la liste `inode_unused`, en incrémentant `inodes_stat.nr_unused`. Si on trouve des inodes pour la même valeur de hachage, on les efface de la liste de leur type et on les ajoute à la liste `inode_unused`. Si c'était un inode anonyme (NetApp .snapshot), on l'efface de la liste de son type et on le nettoie/détruit complètement.

4.2. Enregistrement/dés-enregistrement de systèmes de fichiers

Le noyau Linux fournit un mécanisme permettant de créer de nouveaux systèmes de fichiers avec un minimum d'effort, ceci pour des raisons historiques :

1. Dans un monde où les gens utilisent des systèmes d'exploitation non-Linux pour protéger leur investissement dans des logiciels traditionnels, Linux devait permettre l'interopérabilité en supportant une multitude de systèmes de fichiers différents — la plupart n'ayant pas d'intérêt par eux même mais seulement pour la compatibilité avec des systèmes d'exploitation non-Linux.

2. Pour écrire des systèmes de fichiers, les gens devaient disposer d'une interface très simple pour qu'ils puissent faire de la rétro-ingénierie sur les systèmes de fichiers propriétaires en créant des versions en lecture seule de ceux-ci. En conséquence, le VFS Linux (système de fichiers virtuel) rend vraiment facile l'implémentation de systèmes de fichiers en lecture seule ; 95% du travail réside alors dans l'ajout du support complet en écriture. Pour prendre un exemple concret, j'ai écrit un système de fichiers BFS en lecture seule pour Linux en à peu près 10 heures, mais il m'a fallu plusieurs semaines pour le terminer et avoir un support complet en écriture (et même aujourd'hui certains puristes se plaignent qu'il n'est pas complet car « il ne supporte pas la compactification »).

3. L'interface VFS est exportée, donc tous les systèmes de fichiers Linux peuvent être implémentés en tant que modules.

Considérons les étapes nécessaire à l'implémentation d'un système de fichiers sous Linux.

Le code nécessaire peut soit être un module chargeable dynamiquement, soit être statiquement lié au noyau, ce qui est fait de façon très transparente sous Linux. Il suffit de créer et initialiser une structure `struct file_system_type` et de l'enregistrer auprès du VFS en utilisant la fonction `register_filesystem()` comme dans l'exemple de `fs/bfs/inode.c` :

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


static struct super_block *bfs_read_super(struct super_block *, void *, int);

static DECLARE_FSTYPE_DEV(bfs_fs_type, "bfs", bfs_read_super);

static int __init init_bfs_fs(void)
{
        return register_filesystem(&bfs_fs_type);
}
```

```
static void __exit exit_bfs_fs(void)
{
        unregister_filesystem(&bfs_fs_type);
}

module_init(init_bfs_fs)
module_exit(exit_bfs_fs)
```

Les macros `module_init()/module_exit()` assurent que, lorsque BFS est compilé comme un module, les fonctions `init_bfs_fs()` et `exit_bfs_fs()` deviennent respectivement `init_module()` et `cleanup_module()` ; si BFS est lié statiquement dans le noyau, le code `exit_bfs_fs()` disparaît puisqu'il est inutile.

`struct file_system_type` est déclaré dans `include/linux/fs.h` :

```
struct file_system_type {
        const char *name;
        int fs_flags;
        struct super_block *(*read_super) (struct super_block *, void *, int);
        struct module *owner;
        struct vfsmount *kern_mnt; /* Pour monter le noyau, si c'est un FS_SING
        struct file_system_type * next;
};
```

Les champs eux-mêmes sont décrits ci-dessous :

- *name* : nom lisible par un être humain, apparaît dans le fichier `/proc/filesystems` et est utilisé comme clef pour trouver un système de fichiers par son nom ; ce même nom est utilisé pour le type de système de fichiers dans *mount(2)*, et il doit être unique : il ne peut y avoir (évidemment) qu'un seul système de fichiers pour un nom donné. Pour les modules, name pointe sur l'espace d'adressage du module et n'est pas copié : cela veut dire que *cat /proc/filesystems* peut planter (oops) si le module a été déchargé mais que le système de fichier est toujours enregistré.

- *fs_flags* : l'un ou plusieurs (combinés par un OU) des drapeaux : `FS_REQUIRES_DEV` pour les systèmes de fichiers qui ne peuvent être montés que sur un périphérique bloc, `FS_SINGLE` pour les systèmes de fichiers qui ne peuvent avoir qu'un super-bloc, `FS_NOMOUNT` pour les systèmes de fichiers qui ne peuvent pas être montés depuis l'espace utilisateur par l'appel système *mount(2)* : ils peuvent néanmoins être montés en interne en utilisant l'interface `kern_mount()`, i.e. pipefs (système de fichiers tube).

- *read_super* : un pointeur sur la fonction qui lit le super-bloc durant l'opération de montage. Cette fonction est requise : si elle n'est pas fournie, l'opération de montage (soit dans l'espace utilisateur soit à l'intérieur du noyau) finira toujours par un échec sauf dans le cas `FS_SINGLE` où il « Oupsera » dans `get_sb_single()`, en essayant de déréférencer un pointeur NULL dans `fs_type->kern_mnt->mnt_sb` avec (`fs_type->kern_mnt = NULL`).

- *owner* (propriétaire) : pointeur sur le module qui implémente le système de fichiers. Si le système de fichiers est lié statiquement dans le noyau, il vaut NULL. Vous n'avez pas besoin de le fixer manuellement car la macro `THIS_MODULE` le fait bien, automatiquement.

- *kern_mnt* : pour les systèmes de fichiers `FS_SINGLE`. Positionné par `kern_mount()` (À FAIRE : `kern_mount()` devrait refuser de monter des systèmes de fichiers si `FS_SINGLE` n'est pas positionné).

- *next* : liage dans une liste chaînée simple ancrée sur `file_systems` (cf `fs/super.c`). Cette liste est protégée par le verrou tournant en lecture-écriture `file_systems_lock` et les fonctions `register/unregister_filesystem()` la modifient en liant et déliant les entrées de la liste.

Le travail de la fonction `read_super()` est de remplir les champs du super-bloc, allouer l'inode racine (root) et initialiser toutes les informations privées du système de fichiers associées à cette instance montée de système de fichiers. Donc, typiquement le `read_super()` :

1. Lira le super-bloc sur le périphérique spécifié via l'argument `sb->s_dev`, en utilisant la fonction de cache tampon `bread()`. S'il prévoit de lire immédiatement d'autres blocs de métadonnées consécutifs, alors cela vaut la peine d'utiliser `breada()` pour ordonnancer la lecture asynchrone de quelques blocs de plus.

2. Vérifiera que le super-bloc contient un nombre magique valide et que l'ensemble « ait l'air » sain.

3. Initialisera `sb->s_op` de façon à ce qu'il pointe sur la structure `struct super_block_operations`. Cette structure contient les fonctions spécifiques au système de fichiers implémentant des opérations comme « read inode », « delete inode », et cætera.

4. Allouera l'inode racine et le dentry racine en utilisant `d_alloc_root()`.

5. Si le système de fichier est monté en lecture seule alors mettra `sb->s_dirt` à 1 et marquera le tampon contenant le super-bloc modifié (À FAIRE : pourquoi est ce que l'on fait ça ? Je l'ai fait dans BFS pour faire comme MINIX…)

4.3. Gestion des descripteurs de fichier

Sous Linux il y a plusieurs niveaux d'indirection entre le descripteur de fichier utilisateur et la structure inode du noyau. Quand un processus fait un appel système *open(2)*, le noyau retourne un petit entier non négatif qui peut être utilisé pour les opérations d'entrée/sortie suivantes sur ce fichier. Cet entier est un index dans un tableau de pointeurs sur `struct file`. Chaque structure de fichier pointe sur un dentry via `file->f_dentry`. Et chaque dentry pointe sur un inode via `dentry->d_inode`.

Chaque tâche contient un champ `tsk->files` qui est un pointeur sur `struct files_struct` défini dans `include/linux/sched.h` :

```
/*
 * Ouvrir la structure du tableau des fichiers
 */
struct files_struct {
        atomic_t count;
        rwlock_t file_lock;
        int max_fds;
        int max_fdset;
        int next_fd;
        struct file ** fd;      /* tableau fd courant */
        fd_set *close_on_exec;
        fd_set *open_fds;
        fd_set close_on_exec_init;
        fd_set open_fds_init;
        struct file * fd_array[NR_OPEN_DEFAULT];
};
```

`file->count` est un compteur de références, incrémenté par `get_file()` (appelé habituellement par `fget()`) et décrémenté par `fput()` et par `put_filp()`. La différence entre `fput()` et `put_filp()`, c'est que `fput()` fait un travail supplémentaire, nécessaire habituellement pour les fichiers réguliers, comme libérer les verrous flocks ou le dentry, etc, tandis que `put_filp()` ne fait que manipuler les structures de la table de fichier, i.e. décrémente le compteur, retire le fichier de `anon_list` et l'ajoute à `free_list`, sous le contrôle du verrou tournant `files_lock`.

`tsk->files` peut être partagé entre parent et enfant si le thread enfant a été créé en utilisant

l'appel système `clone()` avec `CLONE_FILES` mis dans l'argument drapeaux. On peut voir cela dans `kernel/fork.c:copy_files()` (appelé par `do_fork()`) qui ne fait qu'incrémenter `file->count` si `CLONE_FILES` est mis au lieu de copier la table des descripteurs de fichiers selon l'habitude consacrée par la tradition du classique *fork(2)* UNIX.

Quand un fichier est ouvert, la structure de fichier allouée pour lui est installée à la position `current->files->fd[fd]` et un bit `fd` est mis dans le bitmap `current->files->open_fds`. Tout ceci est réalisé sous le contrôle du verrou tournant en lecture-écriture `current->files->file_lock`. Quand le descripteur est fermé, le bit `fd` est nettoyé dans `current->files->open_fds` et `current->files->next_fd` est rendu égal à `fd`, indication qui aidera à trouver le premier descripteur libre la prochaine fois que ce processus voudra ouvrir un fichier.

4.4. Gestion de la structure des fichiers

La structure de fichier est déclarée dans `include/linux/fs.h` :

```
struct fown_struct {
        int pid;                /* le pid ou -pgrp auquel SIGIO doit être envoy
        uid_t uid, euid;        /* uid/euid du processus définissant le proprié
        int signum;             /* le signal posix.1b rt à  envoyer sur l'IO */
};

struct file {
        struct list_head        f_list;
        struct dentry           *f_dentry;
        struct vfsmount         *f_vfsmnt;
        struct file_operations  *f_op;
        atomic_t                f_count;
        unsigned int            f_flags;
        mode_t                  f_mode;
        loff_t                  f_pos;
        unsigned long           f_reada, f_ramax, f_raend, f_ralen, f_rawin;
        struct fown_struct      f_owner;
        unsigned int            f_uid, f_gid;
        int                     f_error;

        unsigned long           f_version;

        /* nécessaire pour les pilotes de tty, et peut-être pour d'autres */
        void                    *private_data;
};
```

Regardons les divers champs de `struct file` :

1. *f_list* : ce champ lie la structure de fichiers à une (et une seule) de ces listes : a) `sb->s_files`, liste de tous les fichiers ouverts sur le système de fichiers, si l'inode correspondant n'est pas anonyme, alors `dentry_open()` (appelé par `filp_open()`) lie le fichier à la liste ; b) `fs/file_table.c:free_list`, contenant les structures de fichiers inutilisées ; c) `fs/file_table.c:anon_list`, quand une nouvelle structure de fichier est créée par `get_empty_filp()`, elle est placée dans cette liste. Toutes ces listes sont protégées par le verrou tournant `files_lock`.

2. *f_dentry* : le dentry correspondant à ce fichier. Le dentry est créé au moment de la vérification nameidata par `open_namei()` (ou plutôt `path_walk()` qu'elle appelle) mais l'actuel champs `file->f_dentry` est fixé par `dentry_open()` à la valeur du dentry trouvé.

3. *f_vfsmnt* : le pointeur sur la structure `vfsmount` du système de fichiers contenant le fichier. C'est fixé par `dentry_open()` mais provient d'une partie du résultat de la recherche de nameidata par `open_namei()` (ou plutôt `path_init()` qu'elle appelle).

4. *f_op* : le pointeur sur `file_operations` qui contient diverses méthodes qui peuvent être invoquées sur le fichier. C'est copié depuis `inode->i_fop` qui a été placé ici par la méthode

spécifique au système de fichiers `s_op->read_inode()` durant la recherche de nameidata. On regardera les méthodes `file_operations` en détail plus loin dans ce chapitre.

5. *f_count* : compteur de références manipulé par `get_file/put_filp/fput`.

6. *f_flags* : Les drapeaux `O_XXX` de l'appel système *open(2)* copiés ici (avec de légères modifications par `filp_open()`) par `dentry_open()` après nettoyage de `O_CREAT`, `O_EXCL`, `O_NOCTTY`, `O_TRUNC` — il n'y a pas de raison de les stocker en permanence puisqu'ils ne peuvent pas être modifiés par des appels `F_SETFL` (ou demandés par `F_GETFL`) de *fcntl(2)*.

7. *f_mode* : une combinaison des drapeaux de l'espace utilisateur et du mode, calculée par `dentry_open()`. Le but de cette manipulation est de stocker les accès en lecture et en écriture dans des bits séparés qu'on puisse tester facilement comme dans `(f_mode & FMODE_WRITE)` et `(f_mode & FMODE_READ)`.

8. *f_pos* : la position courante dans le fichier pour la prochaine lecture ou écriture dans le fichier. Sous i386 c'est de type `long long`, i.e. une variable 64bit.

9. *f_reada, f_ramax, f_raend, f_ralen, f_rawin* : pour supporter la lecture anticipée (readahead) — trop complexe pour être discuté par des mortels ;)

10. *f_owner* : propriétaire des entrées/sorties sur le fichier qui recevra les notifications d'entrées/sorties asynchrones via le mécanisme `SIGIO` (voir `fs/fcntl.c:kill_fasync()`).

11. *f_uid, f_gid* — reçoit l'id (identificateur) de l'utilisateur et du groupe du processus qui a ouvert le fichier au moment où la structure de fichier a été créée dans `get_empty_filp()`. Si le fichier est une socket, c'est utilisé par netfilter ipv4.

12. *f_error* : utilisé par le client NFS pour retourner les erreurs d'écritures. C'est fixé dans `fs/nfs/file.c` et vérifié dans `mm/filemap.c:generic_file_write()`.

13. *f_version* — mécanisme de gestion de version pour l'invalidation des caches, incrémentée (en utilisant la variable globale `event`) à chaque changement de `f_pos`.

14. *private_data* : données privées du fichier qui peuvent être utilisées par le système de fichiers (i.e. coda y stocke les données d'identification) ou par les pilotes de périphériques. Les pilotes de périphériques (en présence de devfs) peuvent utiliser ce champ pour différencier plusieurs instances au lieu d'utiliser le classique nombre mineur encodé dans `file->f_dentry->d_inode->i_rdev`.

Maintenant regardons la structure `file_operations` qui contient les méthodes pouvant être invoquées sur les fichiers. Rappelons nous que c'est une copie de `inode->i_fop` évalué par la méthode `s_op->read_inode()`. Elle est déclarée dans `include/linux/fs.h` :

```
struct file_operations {
  struct module *owner;
  loff_t (*llseek) (struct file *, loff_t, int);
  ssize_t (*read) (struct file *, char *, size_t, loff_t *);
  ssize_t (*write) (struct file *, const char *, size_t, 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);
  int (*mmap) (struct file *, struct vm_area_struct *);
  int (*open) (struct inode *, struct file *);
  int (*flush) (struct file *);
  int (*release) (struct inode *, struct file *);
  int (*fsync) (struct file *, struct dentry *, int datasync);
  int (*fasync) (int, struct file *, int);
  int (*lock) (struct file *, int, struct file_lock *);
  ssize_t (*readv) (struct file *, const struct iovec *,
                    unsigned long, loff_t *);
  ssize_t (*writev) (struct file *, const struct iovec *,
```

```
                        unsigned long, loff_t *);
};
```

1. *owner* : un pointeur sur le module qui possède le sous-système en question. Seuls les pilotes ont besoin de le fixer à `THIS_MODULE`, heureusement les systèmes de fichiers peuvent l'ignorer car leurs compteurs de modules sont contrôlés au moment de mount/umount alors que les pilotes ont besoin de les contrôler au moment des open/release.

2. *llseek* : implémente l'appel système *lseek(2)*. Il est habituellement omis et `fs/read_write.c:default_llseek()` est utilisé, qui fait ce qu'il faut (À FAIRE : forcer tous ceux qui le positionnent à NULL actuellement à utiliser default_llseek — de cette façon nous économisons un `if()` dans `llseek()`)

3. *read* : implémente l'appel système `read(2)`. Les systèmes de fichiers peuvent utiliser `mm/filemap.c:generic_file_read()` pour les fichiers réguliers et `fs/read_write.c:generic_read_dir()` (qui renvoie simplement `-EISDIR`) pour les répertoires.

4. *write* : implémente l'appel système *write(2)*. Les systèmes de fichiers peuvent utiliser `mm/filemap.c:generic_file_write()` pour les fichiers réguliers et l'ignorer pour les répertoires.

5. *readdir* : utilisé par le système de fichiers. Ignoré pour les fichiers réguliers, il implémente les appels système *readdir(2)* et *getdents(2)* pour les répertoires.

6. *poll* : implémente les appels système *poll(2)* et *select(2)*.

7. *ioctl* : implémente les contrôles d'entrée/sortie des pilotes ou des systèmes de fichiers spécifiques. Remarquez que les ioctls comme `FIBMAP`, `FIGETBSZ`, `FIONREAD` sur les fichiers génériques sont implémentés aux niveaux supérieurs donc ils ne lisent jamais la méthode `f_op->ioctl()`.

8. *mmap* : implémente l'appel système *mmap(2)*. Les systèmes de fichiers peuvent utiliser *generic_file_mmap* ici pour les fichiers réguliers et l'ignorer pour les répertoires.

9. *open* : appelé au moment de *open(2)* par `dentry_open()`. Les systèmes de fichiers l'utilisent rarement, par exemple coda essaie de cacher localement le fichier au moment de l'ouverture.

10. *flush* : appelé à chaque *close(2)* de ce fichier, pas nécessairement le dernier (voir la méthode `release()` plus bas). Le seul système de fichier qui l'utilise est le client NFS, pour vider toutes les pages modifiées. Remarquez qu'il peut retourner une erreur qui sera repassée dans l'espace utilisateur qui a fait l'appel système *close(2)*.

11. *release* : appelé au dernier *close(2)* sur ce fichier, i.e. quand `file->f_count` atteint 0. Quoiqu'il soit défini comme retournant un entier, la valeur de retour est ignorée par le VFS (voir `fs/file_table.c:__fput()`).

12. *fsync* : correspond directement aux appels systèmes *fsync(2)/fdatasync(2)*, le dernier argument spécifiant si c'est fsync ou fdatasync. Aucun travail pratiquement n'est fait par le VFS à ce niveau, sauf faire correspondre le descripteur fichier à une structure de fichier (`file = fget(fd)`) et monter/descendre le sémaphore `inode->i_sem`. Le système de fichiers ext2 ignore actuellement le dernier argument et fait exactement la même chose pour *fsync(2)* que pour *fdatasync(2)*.

13. *fasync* : cette méthode est appelée quand `file->f_flags & FASYNC` change.

14. *lock* : la portion spécifique au système de fichier du mécanisme POSIX *fcntl(2)* de verrouillage de région de fichier. Le seul bug ici est qu'il est appelé après la partie dépendant du système de fichiers (`posix_lock_file()`), et que s'il réussit alors que le verrou POSIX standard a échoué, il ne sera jamais déverrouillé au niveau dépendant du système de fichiers.

15. *readv* : implémente l'appel système *readv(2)*.

16. *writev* : implémente l'appel système *writev(2)*.

4.5. Super-bloc et gestion des points de montage

Sous Linux, les informations à propos des systèmes de fichiers montés sont gardées dans deux structures séparées — `super_block` et `vfsmount`. La raison en est que Linux autorise le montage du même système de fichiers (périphérique bloc) sur plusieurs points de montage, ce qui signifie que le même `super_block` peut correspondre à des structures `vfsmount` multiples.

Regardons d'abord `struct super_block`, déclarée dans `include/linux/fs.h` :

```
struct super_block {
    struct list_head        s_list;      /* À laisser au début */
    kdev_t                  s_dev;
    unsigned long           s_blocksize;
    unsigned char           s_blocksize_bits;
    unsigned char           s_lock;
    unsigned char           s_dirt;
    struct file_system_type *s_type;
    struct super_operations *s_op;
    struct dquot_operations *dq_op;
    unsigned long           s_flags;
    unsigned long           s_magic;
    struct dentry           *s_root;
    wait_queue_head_t       s_wait;
    struct list_head        s_dirty;     /* inodes modifiés */
    struct list_head        s_files;

    struct block_device     *s_bdev;
    struct list_head        s_mounts;    /* vfsmount(s) de celui-ci */
    struct quota_mount_options s_dquot;  /* options spécifiques aux quotas */

    union {
            struct minix_sb_info    minix_sb;
            struct ext2_sb_info     ext2_sb;
            ... all filesystems that need sb-private info ...
            void                    *generic_sbp;
    } u;
   /*
    * Le champ suivant n'est *que* pour VFS. Aucun système de fichiers
    * n'a affaire à  lui, même pour jeter un œil. Vous aurez été
    * prévenus.
    */
    struct semaphore s_vfs_rename_sem;    /* Usine à  gaz */

   /* Le champ suivant est utilisé par knfsd pour convertir un handle de
    * fichier (basé sur le numéro d'inode) en un dentry. Lorsqu'il dessine
    * un chemin dans l'arbre du dcache à  partir du bas, il peut y avoir
    * provisoirement un sous-chemin de dentrys qui n'est pas connecté à
    * l'arbre principal. Ce sémaphore assure qu'il n'y ait jamais qu'un seul
    * chemin libéré ainsi par système de fichiers. Noter qu'avoir des
    * fichiers (ou n'importe quoi d'autre, en dehors d'un répertoire)
    * non connectés est autorisé, mais pas d'avoir des répertoires non
    * connectés…
    */
    struct semaphore s_nfsd_free_path_sem;
};
```

Les différents champs de la structure `super_block` sont :

1. *s_list* : une liste doublement chaînée de tous les super-blocs actifs ; remarquez que je ne dis pas

« de tous les systèmes de fichiers montés » car sous Linux on peut avoir plusieurs instances d'un système de fichiers montées correspondant à un seul super-bloc.

2. *s_dev* : pour les systèmes de fichiers qui requièrent qu'un bloc soit monté, i.e. pour les systèmes de fichiers `FS_REQUIRES_DEV`, c'est le `i_dev` du périphérique bloc. Pour les autres (appelés systèmes de fichiers anonymes) c'est un entier `MKDEV(UNNAMED_MAJOR, i)` où `i` est le premier bit non positionné dans le tableau `unnamed_dev_in_use`, de 1 à 255 inclus. Voir `fs/super.c:get_unnamed_dev()/put_unnamed_dev()`. Il a été suggéré plusieurs fois que les systèmes de fichiers anonymes n'utilisent pas le champ `s_dev`.

3. *s_blocksize, s_blocksize_bits* : taille de bloc (blocksize) et log2(blocksize).

4. *s_lock* : indique si le super-bloc est actuellement verrouillé par `lock_super()/unlock_super()`.

5. *s_dirt* : mis quand le super-bloc est modifié, et enlevé à chaque fois qu'il est copié sur le disque.

6. *s_type* : pointeur sur `struct file_system_type` du système de fichiers correspondant. La méthode `read_super()` du système de fichiers n'a pas besoin de le positionner puisque le VFS `fs/super.c:read_super()` le fait pour vous si `read_super()` spécifique au système de fichiers réussit et le réinitialise à NULL si elle échoue.

7. *s_op* : pointeur sur la structure `super_operations` qui contient les méthodes spécifiques au fs pour lire/écrire les inodes, et cætera. Il revient à la méthode `read_super()` du système de fichiers d'initialiser correctement `s_op`.

8. *dq_op* : les opérations de quota disque.

9. *s_flags* : les drapeaux du super-bloc.

10. *s_magic* : le nombre magique du système de fichiers. Utilisé par les systèmes de fichiers minix pour en différencier les multiples versions.

11. *s_root* : dentry de la racine du système de fichier. Il revient à `read_super()` de lire l'inode racine sur le disque et de le passer à `d_alloc_root()` pour allouer le dentry et l'instancier. Certains systèmes de fichiers écrivent « root » (racine) autrement que « / » et utilisent alors la fonction plus générique `d_alloc()` pour attacher le dentry à un nom, par exemple pipefs se monte lui-même sur « pipe: » comme sa propre racine au lieu de « / ».

12. *s_wait* : file d'attente des processus attendant que le super-bloc soit déverrouillé.

13. *s_dirty* : une liste de tous les inodes modifiés (dirty). Rappelez vous que si un inode est modifié (`inode->i_state & I_DIRTY`), il est dans la liste des inodes modifiés spécifique au super-bloc lié via `inode->i_list`.

14. *s_files* : une liste de tous les fichiers ouverts sur ce super-bloc. Utile pour décider si un système de fichiers peut être remonté en lecture seule, voir `fs/file_table.c:fs_may_remount_ro()` qui explore la liste `sb->s_files` et refuse le remontage s'il y a des fichiers ouverts en écriture (`file->f_mode & FMODE_WRITE`) ou des fichiers avec un effacement (unlink) suspendu (`inode->i_nlink == 0`).

15. *s_bdev* : pour `FS_REQUIRES_DEV`, il pointe sur une structure de périphérique bloc décrivant le périphérique sur lequel le système de fichiers est monté.

16. *s_mounts* : une liste de toutes les structures `vfsmount`, une pour chaque instance montée de ce super-bloc.

17. *s_dquot* : encore pour les quotas disque.

Les opérations du super-bloc sont décrites dans la structure `super_operations` déclarée dans `include/linux/fs.h` :

```
struct super_operations {
        void (*read_inode) (struct inode *);
        void (*write_inode) (struct inode *, int);

        void (*put_inode) (struct inode *);
        void (*delete_inode) (struct inode *);
        void (*put_super) (struct super_block *);
        void (*write_super) (struct super_block *);
        int (*statfs) (struct super_block *, struct statfs *);
        int (*remount_fs) (struct super_block *, int *, char *);
        void (*clear_inode) (struct inode *);
        void (*umount_begin) (struct super_block *);
};
```

1. *read_inode* : lit l'inode sur le système de fichiers. Est appelée seulement depuis `fs/inode.c:get_new_inode()` depuis `iget4()` (et donc `iget()`). Si un système de fichiers veut utiliser `iget()`, alors `read_inode()` doit être implémentée — autrement `get_new_inode()` va paniquer. Pendant que l'inode est lu, il est verrouillé (`inode->i_state = I_LOCK`). Quand la fonction retourne, tous les processus en attente de `inode->i_wait` sont réveillés. Le rôle de la méthode `read_inode()` du système de fichier est de localiser le bloc du disque qui contient l'inode à lire, elle utilise la fonction du cache tampon `bread()` pour le lire et initialiser les différents champs de la structure de l'inode, par exemple les `inode->i_op` et `inode->i_fop` pour qu'au niveau du VFS on sache quelles opérations peuvent être effectuées sur l'inode ou le fichier correspondant. Les systèmes de fichiers qui n'implémentent pas `read_inode()` sont ramfs et pipefs. Par exemple, ramfs a sa propre fonction `ramfs_get_inode()` de génération d'inodes et toutes les opérations sur les inodes l'appellent quand il faut.

2. *write_inode* : écrit l'inode sur le disque. Similaire à `read_inode()` dans ce qu'elle a besoin de localiser le bloc disque pertinent et d'interagir avec le cache tampon en appelant `mark_buffer_dirty(bh)`. Cette méthode est appelée sur les inodes modifiés (ceux marqués modifiés avec `mark_inode_dirty()`) quand les inodes ont besoin d'être synchronisés individuellement ou lors de la synchronisation du système de fichiers complet.

3. *put_inode* : appelée à chaque fois que le compteur de références est décrémenté.

4. *delete_inode* : appelé dès qu'à la fois `inode->i_count` et `inode->i_nlink` atteignent 0. Le système de fichiers efface la copie sur le disque de l'inode et appelle `clear_inode()` sur l'inode du VFS pour « l'exécuter sans autre forme de procès ».

5. *put_super* : appelé dans les dernières étapes de l'appel système *umount(2)* pour notifier au système de fichiers que toutes les variables privées qu'il détient concernant cette instance doivent être libérées. Typiquement il voudra `brelse()` le verrou contenant le super-bloc et `kfree()` tous les bitmaps alloués pour les blocs libres, inodes, et cætera.

6. *write_super* : appelé quand il faut écrire le super-bloc sur le disque. Il devra trouver le bloc contenant le super-bloc (habituellement conservé dans la zone `sb-private`) et `mark_buffer_dirty(bh)` . Il doit aussi nettoyer le drapeau `sb->s_dirt`.

7. *statfs* : implémente les appels systèmes *fstatfs(2)/statfs(2)*. Remarquez que le pointeur sur `struct statfs` passé en argument est un pointeur noyau, pas un pointeur utilisateur donc nous n'avons besoin de faire aucune entrée/sortie depuis/vers l'espace utilisateur. Si `statfs(2)` n'est pas implémentée alors elle échouera avec `ENOSYS`.

8. *remount_fs* : appelé à chaque fois qu'il faut remonter le système de fichiers.

9. *clear_inode* : appelée depuis `clear_inode()` au niveau VFS. Les systèmes de fichiers qui attachent des données privées à la structure d'inode (via le champ `generic_ip`) doivent les libérer ici.

10. *umount_begin* : appelé durant un démontage forcé pour que le système de fichiers soit prévenu

d'avance et fasse de son mieux pour éviter de rester occupé. Utilisé uniquement par NFS actuellement. Ceci n'a rien à voir avec l'idée d'un support générique du démontage forcé au niveau VFS.

Alors regardons ce qu'il se passe quand nous montons un système de fichiers présent sur un disque (`FS_REQUIRES_DEV`). L'implémentation de l'appel système *mount(2)* est dans `fs/super.c:sys_mount()` qui n'est qu'un emballage qui copie les options, le type de système de fichiers et le nom du périphérique pour la fonction `do_mount()` qui fait réellement le travail :

1. Le pilote du système de fichiers est chargé si besoin est et le compteur de références du module est incrémenté. Remarquez que pendant l'opération de montage, le compteur de références du module du système de fichiers est incrémenté deux fois — une fois par `do_mount()` lors de l'appel à `get_fs_type()` et une fois par `get_sb_dev()` lors de l'appel à `get_filesystem()` si `read_super()` a réussi. Le premier incrément est là pour éviter que le module soit déchargé pendant que nous sommes dans la méthode `read_super()` et le second pour indiquer que le module est utilisé par l'instance montée considérée. Évidemment, `do_mount()` décrémente le compteur avant de retourner, donc finalement le compteur n'augmente que de 1 après chaque montage.

2. Puisque dans notre cas `fs_type->fs_flags & FS_REQUIRES_DEV` est vrai, le super-bloc est initialisé par un appel à `get_sb_bdev()` qui obtient la référence des périphériques de bloc et interagit avec la méthode `read_super()` du système de fichiers pour remplir le super-bloc. Si tout se passe bien, la structure `super_block` est initialisée et nous avons une référence de plus au module du système de fichiers et une référence au périphérique de bloc sous-jacent.

3. Une nouvelle structure `vfsmount` est allouée et liée à la liste `sb->s_mounts` et à la liste globale `vfsmntlist`. Le champ `mnt_instances` de `vfsmount` permet de trouver toutes les instances montées sur notre super-bloc. Le champ `mnt_list` permet de trouver toutes les instances pour tous les super-blocs du système. Le champ `mnt_sb` pointe sur le super-bloc et `mnt_root` obtient une nouvelle référence sur le dentry `sb->s_root`.

4.6. Exemple de système de fichiers virtuel : pipefs

Comme exemple simple de système de fichiers qui ne requiert pas un périphérique bloc pour être monté, considérons pipefs dans `fs/pipe.c`. Le préambule de ce fichier va droit au but et ne nécessite guère d'explications :

```
static DECLARE_FSTYPE(pipe_fs_type, "pipefs", pipefs_read_super,
        FS_NOMOUNT|FS_SINGLE);

static int __init init_pipe_fs(void)
{
        int err = register_filesystem(&pipe_fs_type);
        if (!err) {
                pipe_mnt = kern_mount(&pipe_fs_type);
                err = PTR_ERR(pipe_mnt);
                if (!IS_ERR(pipe_mnt))
                        err = 0;
        }
        return err;
}

static void __exit exit_pipe_fs(void)
{
        unregister_filesystem(&pipe_fs_type);
        kern_umount(pipe_mnt);
}

module_init(init_pipe_fs)
```

```
module_exit(exit_pipe_fs)
```

Le système de fichiers est du type FS_NOMOUNT|FS_SINGLE, ce qui signifie qu'il ne peut pas être monté depuis l'espace utilisateur et qu'il ne peut avoir qu'un super-bloc dans tout le système. Dire que le fichier est de type FS_SINGLE signifie aussi qu'il doit être monté via kern_mount() une fois qu'il a réussi à s'enregistrer via register_filesystem(), et c'est exactement ce qui se passe dans init_pipe_fs(). Le seul bug de cette fonction est que si kern_mount() échoue (i.e. parce que kmalloc() échoue dans l'allocation add_vfsmnt()), alors le système de fichiers reste enregistré tandis que l'initialisation du module a échoué. Ce qui fera « planter » *cat /proc/filesystems*. (je viens juste d'envoyer un patch à Linus mentionnant cela, bien que ce ne soit pas un vrai bug aujourd'hui car pipefs ne peut pas être compilé comme un module, la fonction devrait être réécrite en gardant à l'esprit que dans le futur il pourrait devenir un module).

Le résultat de register_filesystem() est que pipe_fs_type est lié à la liste file_systems si bien qu'on peut lire /proc/filesystems et y trouver l'entrée « pipefs » avec un drapeau « nodev » indiquant que FS_REQUIRES_DEV n'est pas mis. Il faudrait vraiment améliorer le fichier /proc/filesystems pour qu'il supporte tous les nouveaux drapeaux FS_ (et j'ai écrit un patch pour le faire) mais on ne peut pas le faire parce que cela planterait toutes les applications utilisateur qui l'utilisent. Bien que les interfaces du noyau Linux changent toutes les cinq minutes (toujours en mieux), quand on en vient à la compatibilité dans l'espace utilisateur, Linux est un système d'exploitation très conservateur qui permet à beaucoup d'applications d'être utilisées longtemps sans recompilation.

Le résultat de kern_mount() est que :

1. Un nouveau numéro de périphérique non nommé (anonyme) est alloué en positionnant un bit dans le bitmap unnamed_dev_in_use; s'il n'y a plus de bit disponible, alors kern_mount() échoue avec EMFILE.

2. Une nouvelle structure de super-bloc est allouée par le biais de get_empty_super(). La fonction get_empty_super() parcourt la liste des super-blocs ancrée sur super_block et cherche une entrée vide, i.e. s->s_dev == 0. Si on ne trouve pas de super-bloc vide, on en alloue un nouveau en utilisant kmalloc() à la priorité GFP_USER. Le nombre maximum de super-blocs dans tout le système est vérifié dans get_empty_super(), donc s'il commence à y avoir des échecs, on peut ajuster le réglage /proc/sys/fs/super-max.

3. Une méthode spécifique au système de fichiers pipe_fs_type->read_super(), i.e. pipefs_read_super(), est invoquée qui alloue l'inode racine et le dentry racine sb->s_root, et affecte la valeur &pipefs_ops à sb->s_op.

4. Alors kern_mount() appelle add_vfsmnt(NULL, sb->s_root, "none"), qui alloue une nouvelle structure vfsmount et la lie à vfsmntlist et sb->s_mounts.

5. Cette nouvelle structure vfsmount est affectée à pipe_fs_type->kern_mnt qui est retourné. La raison pour laquelle la valeur retournée par kern_mount() est une structure vfsmount est que même les systèmes de fichiers FS_SINGLE peuvent être montés plusieurs fois et qu'alors leurs mnt->mnt_sb pointeraient sur la même variable qu'il serait idiot de retourner à chaque appel à kern_mount().

Maintenant que le système de fichiers est enregistré et monté dans le noyau, nous pouvons l'utiliser. Le point d'entrée du système de fichiers pipefs est l'appel système *pipe(2)*, implémenté dans la fonction sys_pipe() dépendante de l'architecture mais le travail effectif est fait par une fonction portable fs/pipe.c:do_pipe(). Regardons do_pipe(). Les interactions avec pipefs se produisent quand do_pipe() appelle get_pipe_inode() pour allouer un nouvel inode pipefs. Pour cet inode, inode->i_sb prend la valeur du super-bloc de pipefs pipe_mnt->mnt_sb, les opérations fichier i_fop sont mises à rdwr_pipe_fops et le nombre de lecteurs et d'écrivains (contenu dans inode->i_pipe) est fixé à 1. La raison pour maintenir un champ inode i_pipe séparé au lieu de le laisser dans l'union fs-private est que les tubes (pipes) et les FIFO partagent le même code et les FIFO peuvent exister sur d'autre systèmes de fichier qui utilisent d'autres chemins d'accès dans la même union, ce qui est du très mauvais C et ne fonctionne que par pur hasard.

Alors, oui, les noyaux 2.2.x fonctionnent par chance et s'arrêteront dès que vous réarrangerez tant soit peu les champs dans l'inode.

Chaque appel système *pipe(2)* incrémente un compteur de références de l'instance `pipe_mnt`.

Sous Linux, les pipes ne sont pas symétrique (pipes bidirectionnels ou STREAM), i.e. les deux côtés du fichier ont des opérations `file->f_op` différentes — `read_pipe_fops` et `write_pipe_fops` respectivement. Écrire sur le côté lecture retourne `EBADF` de même que lire sur le côté écriture.

4.7. Exemple de système de fichiers sur disque : BFS

Comme exemple simple d'un système de fichiers Linux sur disque, considérons BFS. Le préambule du module BFS est dans `fs/bfs/inode.c` :

```
static DECLARE_FSTYPE_DEV(bfs_fs_type, "bfs", bfs_read_super);

static int __init init_bfs_fs(void)
{
        return register_filesystem(&bfs_fs_type);
}

static void __exit exit_bfs_fs(void)
{
        unregister_filesystem(&bfs_fs_type);
}

module_init(init_bfs_fs)
module_exit(exit_bfs_fs)
```

Une macro spéciale pour déclarer le fstype `DECLARE_FSTYPE_DEV()` est utilisée pour mettre le `fs_type->flags` à `FS_REQUIRES_DEV` ce qui signifie que BFS requiert un vrai périphérique bloc pour être monté.

La fonction d'initialisation du module enregistre le système de fichiers auprès du VFS et la fonction de nettoyage (présente seulement quand BFS est configuré en tant que module) le dés-enregistre.

Une fois le système de fichiers enregistré, on peut procéder au montage, ce qui invoquera la méthode `fs_type->read_super()` implémentée dans `fs/bfs/inode.c:bfs_read_super()`. Elle fait ce qui suit :

1. `set_blocksize(s->s_dev, BFS_BSIZE)` : comme nous sommes sur le point d'interagir avec le périphérique bloc via le cache tampon, on doit initialiser quelques trucs, à savoir fixer la taille de bloc et aussi informer le VFS via les champs `s->s_blocksize` et `s->s_blocksize_bits`.

2. `bh = bread(dev, 0, BFS_BSIZE)` : on lit le bloc 0 du périphérique passé via `s->s_dev`. Ce bloc est le super-bloc du système de fichiers.

3. Le super-bloc est validé par son nombre `BFS_MAGIC`, et, s'il est valide, est stocké dans le champ sb-private `s->su_sbh` (qui est en réalité `s->u.bfs_sb.si_sbh`).

4. Alors on alloue le bitmap d'inodes en utilisant `kmalloc(GFP_KERNEL)` et on remet tous les bits à 0 sauf les deux premiers que l'on met à 1 pour indiquer qu'il ne faut jamais allouer les inodes 0 et 1. Le 2 est l'inode racine et le bit correspondant sera mis à 1 quelques lignes plus tard de toutes façons — le système de fichiers doit avoir un inode racine valide au moment du montage !

5. Puis on initialise `s->s_op`, donc on pourra maintenant invoquer le cache inode via `iget()`, ce qui revient à invoquer `s_op->read_inode()`. On trouve le bloc contenant l'inode spécifié (par `inode->i_ino` et `inode->i_dev`) et on le lit. Si on n'arrive pas à avoir l'inode racine, on libère le bitmap d'inodes ainsi que le tampon du super-bloc dans le tampon de cache

et on renvoie NULL. Si l'inode racine était OK en lecture, on alloue un dentry de nom `"/"` (puisqu'il devient racine) et on l'instancie avec cet inode.

6. Maintenant on parcourt tous les inodes du système de fichiers et on les lit dans le but de positionner les bits correspondants dans notre bitmap d'inodes interne et aussi de calculer d'autres paramètres internes comme le décalage du dernier inode et les blocs de début/fin du dernier fichier. Chaque inode qu'on lit est renvoyé dans le cache inode via `iput()` — on ne conserve pas de référence sur lui plus longtemps que nécessaire.

7. Si le système de fichiers n'a pas été monté en lecture seule (read_only), on marque le tampon du super-bloc modifié (dirty) et on met le drapeau `s->s_dirt` (À FAIRE : Pourquoi faire ça ? A l'origine, je l'ai fait parce que `minix_read_super()` le fait mais ni minix ni BFS ne semblent modifier le super-bloc dans `read_super()`).

8. Tout est bon, donc on retourne le super-bloc initialisé à l'appelant au niveau du VFS, i.e. `fs/super.c:read_super()`.

Quand la fonction `read_super()` est retournée avec succès, le VFS obtient une référence sur le module du système de fichiers via l'appel à `get_filesystem(fs_type)` dans `fs/super.c:get_sb_bdev()` et une référence au périphérique bloc.

Maintenant, examinons ce qu'il se passe quand on fait une entrée/sortie sur le système de fichiers. Nous avons déjà examiné comment les inodes sont lus quand `iget()` est appelé et comment ils sont relâchés sur un `iput()`. La lecture des inodes configure, parmi d'autres choses, `inode->i_op` et `inode->i_fop` ; l'ouverture d'un fichier propage `inode->i_fop` vers `file->f_op`.

Parcourons le code de l'appel système *link(2)*. L'implémentation de l'appel système est dans `fs/namei.c:sys_link()` :

1. Les noms de l'espace utilisateurs sont copiés dans l'espace noyau par le biais de la fonction `getname()` qui effectue les vérifications d'erreurs.

2. Ces noms sont convertis au format nameidata en utilisant l'interaction `path_init()/path_walk()` avec le dcache. Le résultat est stocké dans les structures `old_nd` et `nd`.

3. Si `old_nd.mnt != nd.mnt` alors le lien inter-périphériques (cross-device link) `EXDEV` est retourné — on ne peut pas lier entre systèmes de fichiers, ce qui sous Linux se traduit par — on ne peut pas lier entre les instances montées d'un système de fichiers (ou, en particulier entre systèmes de fichiers).

4. Un nouveau dentry correspondant à `nd` est créé par `lookup_create()`.

5. Une fonction générique `vfs_link()` est appelée qui vérifie si on peut créer une nouvelle entrée dans le répertoire puis invoque la méthode `dir->i_op->link()` qui nous ramène à la fonction `fs/bfs/dir.c:bfs_link()` spécifique au système de fichiers.

6. Dans `bfs_link()`, on teste si c'est un répertoire qu'on essaye de lier et si oui, on refuse avec l'erreur `EPERM`. C'est le même comportement que le standard (ext2).

7. On essaie d'ajouter la nouvelle entrée de répertoire au répertoire spécifié en appelant la fonction `bfs_add_entry()` qui parcourt toutes les entrées en cherchant une place inutilisée (`de->ino == 0`) et, quand elle en trouve une, écrit la paire nom/inode dans le bloc correspondant et le marque modifié (avec une priorité non-super-bloc).

8. Si on a réussi à ajouter l'entrée de répertoire, il n'y plus de risque que l'opération échoue donc on incrémente `inode->i_nlink`, met à jour `inode->i_ctime` et marque l'inode modifiée tout en créant la nouvelle instance de dentry avec l'inode.

D'autres opérations sur les inodes comme `unlink()/rename()`, etc, fonctionnent de la même

façon, cela ne vaut pas la peine de les expliquer toutes en détails.

4.8. Domaines d'exécution et formats binaires

Linux supporte le chargement des binaires des applications utilisateur depuis les disques. Plus intéressant, les binaires peuvent être stockés sous différents formats et la réponse du système d'exploitation aux programmes via les appels systèmes peut dévier de la norme (la norme étant le comportement de Linux) si nécessaire, afin d'émuler le comportement d'appels systèmes d'autre versions (Solaris, UnixWare, etc). C'est à cela que servent les domaines d'exécution et les formats binaires.

Chaque tâche Linux a une personnalité stockée dans sa `task_struct` (`p->personality`). Les personnalités existantes à l'heure actuelle (soit dans le noyau officiel ou par l'ajout d'un patch) incluent le support pour FreeBSD, Solaris, UnixWare, OpenServer et beaucoup d'autres systèmes d'exploitation populaires. La valeur de `current->personality` se décompose en deux partie :

1. les trois octets hauts — émulation de bug : `STICKY_TIMEOUTS`, `WHOLE_SECONDS`, et cætera.
2. l'octet du bas — personnalité propre, un nombre unique.

En changeant la personnalité, on peut changer la façon dont le système d'exploitation traite certains appels système, par exemple l'ajout de `STICKY_TIMEOUT` à `current->personality` fait que l'appel système *select(2)* préserve la valeur du dernier argument (timeout) au lieu de stocker le temps d'activité. Quelques programmes bogués comptent sur des systèmes d'exploitation bogués (pas Linux) et donc Linux fournit un moyen d'émuler les bugs dans les cas où le code source n'est pas disponible et donc que les bugs ne peuvent pas être corrigés.

Le domaine d'exécution est un ensemble de personnalités contiguës implémentées par un seul module. Habituellement, il y a un seul domaine d'exécution qui implémente une seule personnalité, mais quelquefois il est possible d'implémenter des personnalités « proches » dans un seul module sans trop de conditions à remplir.

Les domaines d'exécution sont implémentés dans `kernel/exec_domain.c` et ont été complètement récrits pour les noyaux 2.4, par rapport aux 2.2.x. La liste des domaines d'exécution couramment supportés par le noyau, avec l'ensemble des personnalités qu'ils supportent, est disponible dans le fichier `/proc/execdomains`. Les domaines d'exécution, à l'exception de `PER_LINUX`, peuvent être implémentés comme des modules chargeables dynamiquement.

L'interface utilisateur consiste en l'appel système *personality(2)*, qui fixe la personnalité actuelle du processus ou renvoie la valeur de `current->personality` quand l'argument personnalité a la valeur impossible 0xffffffff. Évidemment, le comportement de cet appel système lui-même ne dépend pas de la personnalité.

L'interface noyau pour l'enregistrement des domaines d'exécution est constitué de deux fonctions :

- `int register_exec_domain(struct exec_domain *)` : enregistre le domaine d'exécution en le liant à la liste chaînée simple `exec_domains` sous la protection en écriture du verrou tournant `exec_domains_lock`. Renvoie 0 pour un succès, différent de zéro pour un échec.
- `int unregister_exec_domain(struct exec_domain *)` : dés-enregistre le domaine d'exécution en le déliant de la liste `exec_domains`, en utilisant encore le verrou tournant `exec_domains_lock` en mode écriture. Renvoie 0 en cas de succès.

La raison pour laquelle `exec_domains_lock` est en lecture-écriture est que seules les requêtes d'enregistrement et de dés-enregistrement modifient la liste, tandis que faire *cat /proc/filesystems* appelle `fs/exec_domain.c:get_exec_domain_list()`, qui n'a besoin que d'un accès en lecture à la liste. L'enregistrement d'un nouveau domaine d'exécution définit un « lcall7 handler » et une table de conversion des numéros de signaux. Actuellement, le patch ABI étend ce concept de

domaine d'exécution pour inclure des informations supplémentaires (comme les options de socket, les types de socket, les familles d'adresses, les tables d'errno (symboles d'erreurs)).

Les formats binaires sont implémentés de manière similaire, i.e. une liste chaînée simple de formats est définie dans `fs/exec.c` et est protégée par un verrou `binfmt_lock` en lecture-écriture. Comme pour `exec_domains_lock`, le `binfmt_lock` en lecture est posé dans la plupart des cas sauf pour les enregistrements/dés-enregistrements de format binaire. L'enregistrement d'un nouveau format étend l'appel système *execve(2)* par de nouvelles fonctions `load_binary()/load_shlib()` ainsi que `core_dump()`. La méthode `load_shlib()` n'est utilisée que par le vieil appel système *uselib(2)* pendant que la méthode `load_binary()` est appelée par `search_binary_handler()` depuis `do_execve()` qui implémente l'appel système *execve(2)*.

La personnalité du processus est déterminée au chargement du format binaire par la méthode `load_binary()` correspondante en utilisant quelques heuristiques. Par exemple, pour reconnaître les binaires UnixWare7, on marque d'abord le binaire en utilisant l'utilitaire *elfmark(1)*, qui fixe `e_flags` de l'entête ELF à la valeur magique 0x314B4455 qui a été détectée au moment du chargement ELF et la personnalité courante `current->personality` est mise à PER_UW7. Si cette heuristique échoue, alors on en utilise une plus générique, telle que considérer que si l'emplacement de l'interpréteur ELF est `/usr/lib/ld.so.1` ou `/usr/lib/libc.so.1`, ceci indique que le binaire est un SVR4 et mettre alors la personnalité à PER_SVR4. On peut écrire un petit programme utilitaire qui utilise les capacité du *ptrace(2)* de Linux pour exécuter le code pas à pas et ainsi forcer un programme à s'exécuter dans n'importe quelle personnalité.

Une fois que la personnalité est connue (et par conséquent `current->exec_domain`), les appels système sont pris en charge comme suit. Admettons que le processus fasse un appel système par le biais de l'instruction de porte lcall7. Cela transfert le contrôle à `ENTRY(lcall7)` de `arch/i386/kernel/entry.S` comme cela a été préparé dans `arch/i386/kernel/traps.c:trap_init()`. Après avoir converti la disposition de la pile de façon appropriée, `entry.S:lcall7` obtient un pointeur sur `exec_domain` depuis `current` puis le décalage (offset) du gestionnaire lcall7 dans `exec_domain` (qui est fixé en dur à 4 dans le code assembleur, si bien que vous ne pouvez pas décaler le champs `handler` dans la déclaration C de la structure `struct exec_domain`) et va là bas. Donc, en C, ça ressemble à ceci :

```
static void UW7_lcall7(int segment, struct pt_regs * regs)
{
       abi_dispatch(regs, &uw7_funcs[regs->eax & 0xff], 1);
}
```

où `abi_dispatch()` est une enveloppe autour de la table des pointeurs de fonction qui implémente les appels système `uw7_funcs` de la personnalité actuelle.

5. Le cache de pages Linux

Dans ce chapitre nous décrivons le cache de pages (pagecache) de Linux 2.4. Le cache de pages est — comme son nom le suggère — un cache des pages physiques. Dans le monde UNIX, le concept de cache de pages est devenu populaire avec l'introduction de UNIX SVR4, où il a remplacé le cache tampon (buffer cache) pour les opérations d'entrées/sorties (I/O) des données.

Alors que le cache de pages SVR4 n'est utilisé que pour cacher les données des systèmes de fichiers et donc utilise la structure vnode et un offset dans le fichier comme paramètres de hachage, le cache de pages de Linux est conçu pour être plus générique, et donc utilise une structure address_space (espace d'adressage — expliquée plus bas) comme premier paramètre. Parce que le cache de pages Linux est fortement couplé à la notion d'espace d'adressage, vous aurez besoin au moins d'une compréhension de base des espaces d'adressage pour appréhender la façon dont fonctionne le cache de pages. Un espace d'adressage est une espèce d'unité de gestion de mémoire logicielle (MMU) qui relie toutes les pages d'un objet (par exemple un inode) à une autre zone (typiquement les blocs physiques d'un disque). La structure address_space est définie dans `include/linux/fs.h` comme :

```
        struct address_space {
```

```
                struct list_head        clean_pages;
                struct list_head        dirty_pages;
                struct list_head        locked_pages;
                unsigned long           nrpages;
                struct address_space_operations *a_ops;
                struct inode            *host;
                struct vm_area_struct *i_mmap;
                struct vm_area_struct *i_mmap_shared;
                spinlock_t          i_shared_lock;

        };
```

Pour comprendre la façon dont l'espace d'adressage fonctionne, il suffit de regarder quelques un de ces champs : `clean_pages`, `dirty_pages` et `locked_pages` sont des listes doublement chaînées de toutes les pages vierges, modifiées, et verrouillées qui appartiennent à cet espace d'adressage, `nrpages` est le nombre de pages dans cet `address_space`, `a_ops` définit les méthodes de cet objet et `host` est un pointeur vers l'inode auquel appartient l'espace d'adressage — il peut aussi être NULL par exemple dans le cas de l'espace d'adressage du gestionnaire de mémoire virtuelle (swapper). L'utilisation de `clean_pages`, `dirty_pages`, `locked_pages` et `nrpages` est évidente, donc regardons d'un œil plus attentif la structure `address_space_operations`, définie dans le même en-tête :

```
        struct address_space_operations {
                int (*writepage)(struct page *);
                int (*readpage)(struct file *, struct page *);
                int (*sync_page)(struct page *);
                int (*prepare_write)(struct file *, struct page *, unsigned, un
                int (*commit_write)(struct file *, struct page *, unsigned, uns
                int (*bmap)(struct address_space *, long);
        };
```

Pour une vue basique des principes des espaces d'adressage (et du cache de pages) il faut regarder ->`writepage` et ->`readpage`, mais en pratique il faut aussi regarder ->`prepare_write` et ->`commit_write`.

Vous pouvez probablement deviner ce que font les méthodes de address_space_operations grâce à leur nom ; néanmoins, elles nécessitent quelques explications. Leur utilisation au cours d'une entrée/sortie de données du système de fichier, ce qui est et de loin la façon la plus fréquente de passer par le cache de pages, fournit un bon moyen de les comprendre. Contrairement aux autres systèmes d'exploitation de type UNIX, Linux possède des opérations génériques sur les fichiers (un sous ensemble des opération vnode SYSV) pour les entrées/sorties de données au travers du cache de pages. Cela veut dire que les données ne vont pas directement interagir avec le système de fichiers lors d'un read/write/mmap (lire/écrire/projeter en mémoire), mais seront lues/écrites dans le cache de pages à chaque fois que ce sera possible. Le cache de pages doit obtenir les données du système de fichiers réel à bas niveau lorsque l'utilisateur veut lire une page qui n'est pas encore en mémoire, ou écrire des données sur le disque quand la mémoire libre diminue.

Pour lire, les méthodes génériques vont d'abord essayer de trouver la page qui correspond au tuplet inode/index voulu.

```
hash = page_hash(inode->i_mapping, index);
```

Ensuite, on teste si la page existe vraiment.

```
hash = page_hash(inode->i_mapping, index);
page = __find_page_nolock(inode->i_mapping, index, *hash);
```

Si elle n'existe pas, on alloue un nouvelle page, et on l'ajoute au hachage du cache de pages.

```
page = page_cache_alloc();
__add_to_page_cache(page, mapping, index, hash);
```

Après que la page ait été hachée on utilise l'opération ->`readpage` d'address_space pour remplir la page avec les données. (le fichier est une instance ouverte de l'inode).

```
error = mapping->a_ops->readpage(file, page);
```

Finalement nous pouvons copier les données dans l'espace utilisateur.

Pour écrire dans le système de fichier il y a deux manières : une pour les projections en mémoire modifiables (mmap), et une pour la famille des appels système `write(2)`. Le cas mmap est très simple, donc nous le traiterons en premier. Quand un utilisateur modifie une projection (mapping), le sous-système VM (mémoire virtuelle) marque la page modifiée.

```
SetPageDirty(page);
```

Le thread noyau bdflush qui essaie de libérer les pages, soit en arrière plan soit parce que la mémoire libre risque de manquer, va essayer d'appeler ->`writepage` sur les pages qui sont explicitement marquées modifiées. La méthode ->`writepage` doit maintenant écrire le contenu des pages sur le disque et libérer la page.

La deuxième manière est _beaucoup_ plus compliquée. Pour chaque page dans laquelle l'utilisateur écrit, nous faisons en gros ce qui suit : (pour le code complet voir `mm/filemap.c:generic_file_write()`).

```
page = __grab_cache_page(mapping, index, &cached_page);
mapping->a_ops->prepare_write(file, page, offset, offset+bytes);
copy_from_user(kaddr+offset, buf, bytes);
mapping->a_ops->commit_write(file, page, offset, offset+bytes);
```

D'abord nous essayons de trouver la page hachée ou d'en allouer une nouvelle, ensuite nous appelons la méthode ->`prepare_write` d'address_space, nous copions le tampon utilisateur dans la zone mémoire du noyau et finalement nous appelons la méthode ->`commit_write`. Comme vous l'avez probablement constaté ->`prepare_write` et ->`commit_write` sont fondamentalement différentes de ->`readpage` et de ->`writepage`, parce qu'elles ne sont pas appelées seulement quand une entrée/sortie physique est nécessaire mais à chaque fois que l'utilisateur modifie le fichier. Il y a deux façons (ou plus ?) de gérer cela, la première utilise le cache tampon (buffer cache) de Linux pour différer l'entrée/sortie physique, en remplissant un pointeur `page->buffers` avec buffer_heads, ce qui sera utilisé dans try_to_free_buffers (`fs/buffers.c`) pour provoquer une entrée/sortie dès que la mémoire manquera, et c'est très largement utilisé dans le noyau actuel. L'autre façon marque juste la page comme modifiée et compte sur ->`writepage` pour faire le reste du travail. Du fait de l'absence d'un bitmap de validité dans la structure page, cela ne fonctionne pas avec un système de fichiers qui a une granularité plus petite que `PAGE_SIZE`.

6. Mécanismes de communication inter-processus (IPC)

Ce chapitre décrit les mécanismes de sémaphore, la mémoire partagée et les files de messages IPC tels qu'ils sont implémentés dans le noyau Linux 2.4. Il est organisé en quatre parties. Les trois premières parties couvrent les interfaces et les fonctions supportées respectivement par les Section 6.1, « Sémaphores », les Section 6.2, « Les files de messages », et la Section 6.3, « La mémoire partagée ». La Section 6.4, « Les primitives des IPC Linux » partie décrit un ensemble de fonctions et de structures de données communes aux trois mécanismes.

6.1. Sémaphores

Les fonctions décrites dans cette partie implémentent les mécanismes de sémaphore au niveau utilisateur. Remarquez que cette implémentation repose sur l'utilisation des sémaphores et des verrous tournants du noyau. Pour éviter toute confusion, le terme « sémaphore noyau » sera utilisé en référence aux sémaphores du noyau. Toutes les autres utilisations du mot « sémaphore » feront référence aux sémaphores du niveau utilisateur.

6.1.1. Interfaces d'appels système des sémaphores

6.1.1.1. sys_semget()

L'appel complet de sys_semget() est protégé par Section 6.4.2.2, « struct ipc_ids », un sémaphore noyau global.

Dans le cas où un nouvel ensemble de sémaphores doit être créé, la fonction Section 6.1.3.1, « newary() » est appelée pour créer et initialiser le nouvel ensemble de sémaphores. L'identificateur du nouvel ensemble est retourné à l'appelant.

Dans le cas où une valeur de clef est fournie pour un ensemble de sémaphores, Section 6.4.1.7, « ipc_findkey() » est invoquée pour rechercher l'index de tableau correspondant au descripteur du sémaphore. Les paramètres et permissions de l'appelant sont vérifiés avant de retourner l'identificateur de l'ensemble de sémaphores.

6.1.1.2. sys_semctl()

Pour les commandes Section 6.1.3.4.1, « IPC_INFO et SEM_INFO », Section 6.1.3.4.1, « IPC_INFO et SEM_INFO », et Section 6.1.3.4.2, « SEM_STAT », Section 6.1.3.4, « semctl_nolock() » est appelée pour exécuter les fonctions nécessaires.

Pour les commandes Section 6.1.3.5.1, « GETALL », Section 6.1.3.5.4, « GETVAL », Section 6.1.3.5.5, « GETPID », Section 6.1.3.5.6, « GETNCNT », Section 6.1.3.5.7, « GETZCNT », Section 6.1.3.5.3, « IPC_STAT », Section 6.1.3.5.8, « SETVAL », et Section 6.1.3.5.2, « SETALL », Section 6.1.3.5, « semctl_main() » est appelée pour exécuter les fonctions nécessaires.

Pour les commandes Section 6.1.3.3.1, « IPC_RMID » et Section 6.1.3.3.2, « IPC_SET », Section 6.1.3.3, « semctl_down() » est appelé pour exécuter les fonctions nécessaires. D'un bout à l'autre de ces opérations, le verrou noyau global Section 6.4.2.2, « struct ipc_ids » est maintenu.

6.1.1.3. sys_semop()

Après avoir validé les paramètres d'appel, les données des opérations du sémaphore sont copiées depuis l'espace utilisateur vers un tampon temporaire. Si un petit tampon temporaire est suffisant, un tampon de pile est utilisé, sinon, un grand tampon est alloué. Après avoir copié les données des opérations du sémaphore, le verrou tournant global de sémaphore est verrouillé, et l'identificateur de l'ensemble de sémaphores spécifique à l'utilisateur est validé. Les permissions d'accès pour l'ensemble de sémaphores sont également validées.

On analyse syntaxiquement (parse) toutes les opérations de sémaphore spécifiées par l'utilisateur. Pendant ce processus, on tient le compte de toutes les opérations dont le drapeau SEM_UNDO est mis. Un drapeau `decrease` est mis si une des opérations soustrait quelque chose à la valeur du sémaphore, et un drapeau `alter` est mis si une des valeurs des sémaphores est modifiée (i.e. augmentée ou diminuée). Le nombre des sémaphores à modifier est validé.

Si SEM_UNDO a été imposé à une des opérations de sémaphores, alors on recherche dans la liste undo (défaire) de la tâche courante une structure undo associée à cet ensemble de sémaphores. Pendant la recherche, si on trouve une valeur de -1 pour l'identificateur d'un ensemble de sémaphores de l'une des structures undo, alors Section 6.1.3.11, « freeundos() » est appelé pour libérer la structure undo et la retirer de la liste. Si aucune structure undo n'est trouvée pour cet ensemble de sémaphores alors Section 6.1.3.12, « alloc_undo() » est appelé pour en allouer et en initialiser une.

La fonction Section 6.1.3.9, « try_atomic_semop() » est appelée avec le paramètre `do_undo` égal à

0 pour exécuter la séquence d'opérations. La valeur de retour indique que les opérations réussissent, échouent ou n'ont pas été exécutées parce qu'elles avaient besoin de bloquer. Chacun de ces cas est décrit plus bas :

6.1.1.3.1. Opérations de sémaphores non-bloquantes

La fonction Section 6.1.3.9, « try_atomic_semop() » retourne zéro pour indiquer que toutes les opérations de la séquence ont réussi. Dans ce cas, Section 6.1.3.8, « update_queue() » est appelée pour parcourir la file des opérations de sémaphores suspendues pour l'ensemble de sémaphores et réveiller toutes les tâches qui n'ont plus besoin de bloquer. Dans ce cas, cela termine l'exécution de l'appel système sys_semop().

6.1.1.3.2. Opérations de sémaphores qui échouent

Si Section 6.1.3.9, « try_atomic_semop() » retourne une valeur négative, c'est qu'une condition d'échec a été rencontrée. Dans ce cas, aucune des opérations n'a été exécutée. Cela se produit soit quand une opération de sémaphore risque de produire une valeur de sémaphore invalide soit quand une opération marquée IPC_NOWAIT est incapable de se terminer. La condition de l'erreur est alors retournée à l'appelant de sys_semop().

Avant que sys_semop() retourne, un appel est fait à Section 6.1.3.8, « update_queue() » pour parcourir la file des opérations de sémaphores suspendues pour l'ensemble de sémaphores et réveiller toutes les tâches endormies qui n'ont plus besoin de bloquer.

6.1.1.3.3. Opérations de sémaphore bloquantes

La fonction Section 6.1.3.9, « try_atomic_semop() » retourne 1 pour indiquer que la séquence des opérations de sémaphore n'a pas été exécutée car l'un des sémaphores aurait bloqué. Dans ce cas, un nouvel élément Section 6.1.2.5, « struct sem_queue » contenant les opérations de ce sémaphore est initialisé. Si une de ces opérations doit altérer l'état du sémaphore, le nouvel élément est ajouté à la fin de la file. Sinon, le nouvel élément est ajouté en tête de la file.

L'élément `semsleeping` de la tâche courante est positionné pour indiquer que cette tâche est endormie sur cet élément Section 6.1.2.5, « struct sem_queue ». La tâche courante est marquée TASK_INTERRUPTIBLE, et l'élément `sleeper` de Section 6.1.2.5, « struct sem_queue » est positionné pour identifier cette tâche comme le dormeur. Le verrou tournant global de sémaphore est ensuite déverrouillé, et schedule() est appelé pour endormir la tâche courante.

Quand elle est réveillée, la tâche reverrouille le verrou tournant global de sémaphore, détermine pourquoi elle a été réveillée, et comment elle doit répondre. Les cas suivants sont traités :

- Si le sémaphore a été retiré, l'appel système échoue avec EIDRM.
- Si l'élément `status` de la structure Section 6.1.2.5, « struct sem_queue » a été mis à 1, c'est que la tâche a été réveillée pour réessayer l'exécution des opérations du sémaphore. Un autre appel à Section 6.1.3.9, « try_atomic_semop() » est fait pour exécuter la séquence d'opérations du sémaphore. Si try_atomic_sweep() renvoie 1, alors la tâche doit encore bloquer comme décrit ci-dessus. Sinon, 0 est retourné en cas de succès, ou le code d'erreur approprié en cas d'échec. Avant que sys_semop() retourne, current->semsleeping est nettoyé, et Section 6.1.2.5, « struct sem_queue » est retiré de la file. Si une des opérations de sémaphore était alors en train d'altérer le sémaphore (augmentation ou diminution), Section 6.1.3.8, « update_queue() » est appelé pour parcourir la file des opérations de sémaphore suspendues pour l'ensemble de sémaphores et toutes les tâches endormies qui ne doivent plus bloquer sont réveillées.
- Si l'élément `status` de la structure Section 6.1.2.5, « struct sem_queue » n'est PAS mis à 1, et que l'élément Section 6.1.2.5, « struct sem_queue » n'a pas été enlevé de la file, c'est que la tâche a été réveillée par une interruption. Dans ce cas, l'appel système échoue avec EINTR. Avant de retourner, current->semsleeping est nettoyé, et Section 6.1.2.5, « struct sem_queue » est retiré de la file. De plus, Section 6.1.3.8, « update_queue() » est appelé si l'une des opérations a altéré des opérations.
- Si l'élément `status` de la structure Section 6.1.2.5, « struct sem_queue » n'est PAS mis à 1, et que l'élément Section 6.1.2.5, « struct sem_queue » n'a pas été retiré de la file, alors les opéra-

tions du sémaphore ont déjà été exécutées par Section 6.1.3.8, « update_queue() ». Le `status` de la file, nul en cas de succès ou négatif en cas d'échec, devient la valeur de retour de l'appel système.

6.1.2. Structures spécifiques au support des sémaphores

Les structures suivantes sont spécifiques au support des sémaphores :

6.1.2.1. struct sem_array

```
/* Une structure de données sem_array pour chaque ensemble de sémaphores
   du système. */
struct sem_array {
    struct kern_ipc_perm sem_perm; /* permissions – voir ipc.h */
    time_t sem_otime; /* instant du dernier semop */
    time_t sem_ctime; /* instant du dernier changement */
    struct sem *sem_base; /* pointe sur le premier sémaphore du tableau */
    struct sem_queue *sem_pending; /* opérations suspendues à  exécuter */
    struct sem_queue **sem_pending_last; /* dernière operation suspendue */
    struct sem_undo *undo; /* les undos demandés pour ce tableau * /
    unsigned long sem_nsems; /* aucun sémaphore dans le tableau */
};
```

6.1.2.2. struct sem

```
/* Une structure de sémaphore pour chaque sémaphore du système. */
struct sem {
    int     semval;     /* valeur courante */
    int     sempid;     /* pid de la dernière opération */
};
```

6.1.2.3. struct seminfo

```
struct  seminfo {
    int semmap;
    int semmni;
    int semmns;
    int semmnu;
    int semmsl;
    int semopm;
    int semume;
    int semusz;
    int semvmx;
    int semaem;
};
```

6.1.2.4. struct semid64_ds

```
struct semid64_ds {
    struct ipc64_perm sem_perm;  /* permissions – voir ipc.h */
    __kernel_time_t sem_otime;   /* instant du dernier semop */
    unsigned long   __unused1;
    __kernel_time_t sem_ctime;   /* instant de la dernière modification */
    unsigned long   __unused2;
    unsigned long   sem_nsems;   /* aucun sémaphore dans le tableau */
    unsigned long   __unused3;
    unsigned long   __unused4;
};
```

6.1.2.5. struct sem_queue

```
/* Une . */
struct sem_queue {
  struct sem_queue *  next;    /* entrée suivante dans la file */
  struct sem_queue ** prev;    /* entrée précédente dans la file, */
                               /* *(q->prev) == q */
  struct task_struct* sleeper; /* ce processus */
  struct sem_undo *   undo;    /* structure d'undo  */
  int                 pid;     /* identificateur du processus demandeur */
  int                 status;  /* statut d'exécution de l'opération */
  struct sem_array *  sma;     /* tableau de sémaphores pour les operations */
  int                 id;      /* identificateur interne du sémaphore */
  struct sembuf *     sops;    /* tableau des opérations suspendues */
  int                 nsops;   /* nombre d'opérations */
  int                 alter;   /* l'opération va modifier le sémaphore */
};
```

6.1.2.6. struct sembuf

```
/* les appels systèmes semop prennent un tableau de ceux-ci. */
struct sembuf {
    unsigned short  sem_num;        /* index du sémaphore dans le tableau */
    short           sem_op;         /* operation sur le sémaphore */
    short           sem_flg;        /* options */
};
```

6.1.2.7. struct sem_undo

```
/* Chaque tâche a sa liste d'opérations à  annuler. Les annulations sont
 * faites automatiquement quand le processus se termine.
 */
struct sem_undo {
  struct sem_undo *  proc_next;  /* entrée suivante pour ce processus */
  struct sem_undo *  id_next;    /* entrée suivante pour cet ensemble */
                                 /* de sémaphores */
  int                semid;      /* identificateur de l'ensemble de sémaphores
  short *            semadj;     /* tableau d'adaptations, une par sémaphore */
};
```

6.1.3. Les fonctions du support des sémaphores

Les fonctions suivantes sont utilisées spécifiquement pour le support des sémaphores :

6.1.3.1. newary()

newary() utilise la fonction Section 6.4.1.1, « ipc_alloc() » pour allouer la mémoire requise pour le nouvel ensemble de sémaphores. Elle alloue assez de mémoire pour le descripteur de l'ensemble de sémaphores et pour chacun des sémaphores de l'ensemble. La mémoire allouée est remise à 0, et l'adresse du premier élément du descripteur de l'ensemble de sémaphores est passé à Section 6.4.1.2, « ipc_addid() ». Section 6.4.1.2, « ipc_addid() » réserve une entrée dans le tableau pour le nouveau descripteur et initialise (Section 6.4.2.1, « struct kern_ipc_perm ») les données pour l'ensemble. La variable globale `used_sems` reçoit le nombre de sémaphores du nouvel ensemble et ainsi l'initialisation des données (Section 6.4.2.1, « struct kern_ipc_perm ») du nouvel ensemble est terminée. D'autres initialisations concernant cet ensemble sont listées ci-dessous :

- L'élément `sem_base` pour l'ensemble est initialisé à l'adresse suivant immédiatement la por-

tion (Section 6.1.2.1, « struct sem_array ») des données nouvellement allouées, ce qui correspond au premier sémaphore de l'ensemble.

- La file `sem_pending` est initialisée et laissée vide .

Toutes les opérations suivant l'appel à Section 6.4.1.2, « ipc_addid() » sont exécutées sous le verrou tournant global des sémaphores. Après déverrouillage de ce verrou, newary() appelle Section 6.4.1.4, « ipc_buildid() » (via sem_buildid()). Cette fonction utilise l'index du descripteur de l'ensemble de sémaphores pour créer un identificateur unique, qui est alors retourné à l'appelant de newary().

6.1.3.2. freeary()

freeary() est appelée par Section 6.1.3.3, « semctl_down() » pour exécuter les fonctions listées ci-dessous. Elle est appelée avec le verrou tournant global de sémaphores verrouillé et elle retourne avec ce verrou déverrouillé.

- La fonction Section 6.4.1.3, « ipc_rmid() » est appelée (par l'intermédiaire de l'enveloppe sem_rmid()) pour effacer l'identificateur de l'ensemble de sémaphores et pour récupérer un pointeur sur l'ensemble de sémaphores.
- La liste d'opérations à annuler de l'ensemble de sémaphores est invalidée.
- Tous les processus suspendus sont réveillés et amenés à échouer avec EIDRM.
- Le nombre de sémaphores utilisés est diminué du nombre de sémaphores de l'ensemble retiré.
- La mémoire associée à l'ensemble de sémaphores est libérée.

6.1.3.3. semctl_down()

semctl_down() fournit les opérations Section 6.1.3.3.1, « IPC_RMID » et Section 6.1.3.3.2, « IPC_SET » de l'appel système semctl(). L'identificateur de l'ensemble de sémaphores et les permissions d'accès sont vérifiés avant chacune de ces opérations, et dans chaque cas, le verrou tournant global de sémaphore est maintenu tout au long de l'opération.

6.1.3.3.1. IPC_RMID

Les opérations IPC_RMID appellent Section 6.1.3.2, « freeary() » pour retirer l'ensemble de sémaphores.

6.1.3.3.2. IPC_SET

Les opérations IPC_SET mettent à jour les éléments `uid`, `gid`, `mode`, et `ctime` de l'ensemble de sémaphores.

6.1.3.4. semctl_nolock()

semctl_nolock() est appelée par Section 6.1.1.2, « sys_semctl() » pour exécuter les fonctions IPC_INFO, SEM_INFO et SEM_STAT.

6.1.3.4.1. IPC_INFO et SEM_INFO

IPC_INFO et SEM_INFO provoquent l'initialisation et le chargement d'un tampon temporaire Section 6.1.2.3, « struct seminfo », sans changer les données statistiques du sémaphore. Alors, tout en maintenant le verrou noyau global de sémaphore `sem_ids.sem`, les éléments `semusz` et `semaem` de la structure Section 6.1.2.3, « struct seminfo » sont mis à jour suivant la commande donnée (IPC_INFO ou SEM_INFO). La valeur de retour de l'appel système est l'identificateur maximum des ensembles de sémaphores.

6.1.3.4.2. SEM_STAT

SEM_STAT provoque l'initialisation d'un tampon temporaire Section 6.1.2.4, « struct semid64_ds ». Le verrou tournant global de sémaphore est maintenu pendant la copie des valeurs `sem_otime`, `sem_ctime`, et `sem_nsems` dans le tampon. Ces données sont ensuite copiées dans l'espace utilisateur.

6.1.3.5. semctl_main()

semctl_main() est appelée par Section 6.1.1.2, « sys_semctl() » pour exécuter un grand nombre des fonctions supportées, comme cela est décrit dans les paragraphes ci-dessous. Avant d'exécuter l'une des opérations suivantes, semctl_main() pose le verrou tournant global de sémaphore et valide l'identificateur de l'ensemble de sémaphores et les permissions. Le verrou tournant est relâché avant de retourner.

6.1.3.5.1. GETALL

L'opération GETALL charge les valeurs du sémaphore courant dans un tampon noyau temporaire et les copie depuis l'espace utilisateur. Une petit tampon de pile est utilisé si le sémaphore est petit. Sinon, le verrou tournant est temporairement enlevé pour allouer un tampon plus grand. Le verrou est maintenu pendant la copie des valeurs de sémaphore dans le tampon temporaire.

6.1.3.5.2. SETALL

L'opération SETALL copie les valeurs des sémaphores depuis l'espace utilisateur dans le tampon temporaire, et ensuite dans l'ensemble de sémaphores. Le verrou tournant est enlevé pendant la copie des valeurs depuis l'espace utilisateur dans le tampon temporaire, et pendant la vérification de la vraisemblance des valeurs. Si l'ensemble de sémaphores est petit, alors le tampon de pile est utilisé, autrement un tampon plus grand est alloué. Le verrou tournant est reposé et maintenu pendant que les opérations suivantes sont réalisées sur l'ensemble de sémaphores :

- Les valeurs des sémaphores sont copiées dans l'ensemble de sémaphores.
- Les ajustements des sémaphores de la file d'annulation relative à l'ensemble de sémaphores sont nettoyés.
- La valeur `sem_ctime` pour l'ensemble de sémaphores est fixée.
- La fonction Section 6.1.3.8, « update_queue() » est appelée pour parcourir la file des semops suspendues et pour rechercher toutes les tâches qui peuvent être terminées par l'opération SETALL. Toutes les tâches suspendues qui ne sont plus bloquées sont réveillées.

6.1.3.5.3. IPC_STAT

Dans l'opération IPC_STAT, les valeurs `sem_otime`, `sem_ctime`, et `sem_nsems` sont copiées dans le tampon de pile. Les données sont ensuite copiées dans l'espace utilisateur avant d'enlever le verrou tournant.

6.1.3.5.4. GETVAL

Pour GETVAL, s'il n'y a pas d'erreur, la valeur de retour de l'appel système est égale à la valeur du sémaphore spécifié.

6.1.3.5.5. GETPID

Pour GETPID, s'il n'y a pas d'erreur, la valeur de retour de l'appel système est égale au `pid` associé à la dernière opération sur le sémaphore.

6.1.3.5.6. GETNCNT

Pour GETNCNT, s'il n'y a pas d'erreur, la valeur de retour de l'appel système est égale au nombre de

processus attendant que le sémaphore devienne négatif. Ce nombre est calculé par la fonction Section 6.1.3.6, « count_semncnt() ».

6.1.3.5.7. GETZCNT

Pour GETZCNT, s'il n'y a pas d'erreur, la valeur de retour de l'appel système est égale au nombre de processus attendant que le sémaphore soit nul. Ce nombre est calculé par la fonction Section 6.1.3.7, « count_semzcnt() ».

6.1.3.5.8. SETVAL

Après avoir validé la nouvelle valeur du sémaphore, les fonctions suivantes sont exécutées :

- Les ajustements de ce sémaphore sont recherchés dans la file d'annulation . Les ajustements trouvés sont remis à zéro.
- La valeur du sémaphore est fixée à la valeur donnée.
- La valeur `sem_ctime` est mise à jour.
- La fonction Section 6.1.3.8, « update_queue() » est appelée pour parcourir la file des semops suspendues et pour chercher les tâches qui peuvent se terminer par l'opération Section 6.1.3.5.2, « SETALL ». Toutes les tâches qui ne sont plus bloquées sont réveillées.

6.1.3.6. count_semncnt()

count_semncnt() compte le nombre de tâches attendant que la valeur du sémaphore devienne négative.

6.1.3.7. count_semzcnt()

count_semzcnt() compte le nombre de tâches attendant que la valeur du sémaphore devienne nulle.

6.1.3.8. update_queue()

update_queue() parcourt la file des semops suspendues d'un ensemble de sémaphores et appelle Section 6.1.3.9, « try_atomic_semop() » pour déterminer quelles séquences d'opérations de sémaphore peuvent réussir. Si l'état de l'élément de la file indique que les tâches bloquées ont déjà été réveillées, alors on saute cet élément. Pour les autres éléments de la file, le drapeau `q-alter` est passé comme paramètre d'annulation à Section 6.1.3.9, « try_atomic_semop() », indiquant que toutes les opérations de modification doivent être annulées avant de retourner.

Si on prévoit que la séquence d'opérations va bloquer, alors update_queue() retourne sans faire aucun changement.

Une séquence d'opérations peut échouer si une des opérations de sémaphore provoque une valeur de sémaphore invalide, ou qu'une opération marquée IPC_NOWAIT ne peut pas se finir. Dans un tel cas, les tâches qui sont bloquées sur la séquence d'opérations du sémaphore sont réveillées, et l'état de la file est positionné au code d'erreur approprié. L'élément est retiré de la file .

Si la séquence d'opérations ne modifie rien, alors elles ont du passer la valeur zéro comme paramètre d'annulation à Section 6.1.3.9, « try_atomic_semop() ». Si ces opérations ont réussi, elles sont considérées terminées et retirées de la file. La tâche bloquée est réveillée, et le `status` de l'élément de la file est positionné pour indiquer le succès.

Si la séquence d'opérations doit modifier la valeur du sémaphore, mais peut réussir, les tâche endormies qui n'ont plus besoin d'être bloquées sont réveillées. L'état de la file est mis à 1 pour indiquer que les tâches bloquées ont été réveillées. Les opérations n'ont pas été exécutées, donc l'élément n'est pas retiré de la file. Les opérations de sémaphore doivent être exécutées par une tâche réveillée.

6.1.3.9. try_atomic_semop()

try_atomic_semop() est appelé par Section 6.1.1.3, « sys_semop() » et Section 6.1.3.8, « update_queue() » pour déterminer si la séquence des opérations du sémaphore va réussir. Il le détermine en essayant d'exécuter toutes les opérations.

Si une opération bloquante est rencontrée, le processus est arrêté et toutes les opérations annulées. -EAGAIN est renvoyé si IPC_NOWAIT est mis. Autrement 1 est renvoyé pour indiquer que la séquence d'opérations est bloquée.

Si une valeur de sémaphore est ajustée au-delà des limites du système, alors toutes les opérations sont annulées, et -ERANGE est renvoyée.

Si toutes les opérations de la séquence réussissent, et que le paramètre `do_undo` n'est pas nul, toutes les opérations sont annulées, et 0 est renvoyé. Si le paramètre `do_undo` est nul, toutes les opérations ont réussi et sont maintenues de force, et le champ `sem_otime` du sémaphore est mis à jour.

6.1.3.10. sem_revalidate()

sem_revalidate() est appelée quand le verrou tournant global de sémaphore a été temporairement levé et que l'on a besoin de reverrouiller. Elle est appelée par Section 6.1.3.5, « semctl_main() » et Section 6.1.3.12, « alloc_undo() ». Elle valide l'identificateur du sémaphore et les permissions et en cas de succès, retourne avec le verrou tournant global de sémaphore verrouillé.

6.1.3.11. freeundos()

freeundos() parcourt la liste d'annulations du processus à la recherche de la structure d'annulation désirée. Si elle est trouvée, cette structure est retirée de la liste et libérée. Un pointeur sur la structure suivante dans la liste d'annulations du processus est retourné.

6.1.3.12. alloc_undo()

alloc_undo() doit être appelée avec le verrou tournant global de sémaphore verrouillé. En cas d'erreur, il retourne avec le verrou déverrouillé.

Le verrou tournant global de sémaphores est déverrouillé, et kmalloc() est appelée pour allouer suffisamment de mémoire pour la structure Section 6.1.2.7, « struct sem_undo », et pour un tableau de valeurs d'ajustement, une par sémaphore de l'ensemble. En cas de succès, le verrou tournant global est remis par l'appel à Section 6.1.3.10, « sem_revalidate() ».

La nouvelle structure semundo est initialisée, et l'adresse de cette structure est placée à l'adresse fournie par l'appelant. La nouvelle structure d'annulation est placée en tête de la liste d'annulations de la tâche courante.

6.1.3.13. sem_exit()

sem_exit() est appelée par do_exit(), et est responsable de l'exécution de tous les ajustements undo pour la tâche qui se termine.

Si le processus courant a bloqué sur un sémaphore, alors il est retiré de la liste Section 6.1.2.5, « struct sem_queue » pendant que le verrou tournant global de sémaphore est maintenu.

La liste d'annulations pour la tâche courante est ensuite parcourue, et les opérations suivantes sont réalisées en maintenant et relâchant le verrou tournant global de sémaphore pour chacun des éléments de la liste. Les opération suivantes sont exécutées pour chacun des éléments d'annulation :

- La structure d'annulation et l'identificateur de l'ensemble de sémaphores sont validés.
- La liste d'annulations de l'ensemble de sémaphores correspondant est parcourue pour trouver une référence à cette structure d'annulation et pour la retirer de la liste.
- Les modifications indiquées dans la structure d'annulation sont effectuées sur l'ensemble de sémaphores.

- Le paramètre `sem_otime` de l'ensemble de sémaphores est mis à jour.
- Section 6.1.3.8, « update_queue() » est appelée pour parcourir la file des semops suspendues et pour réveiller toutes les tâches qui n'ont plus à être bloquées par l'exécution des opérations d'annulation.
- La structure d'annulation est libérée.

Quand le traitement de la liste est terminé, la valeur current->semundo est nettoyée.

6.2. Les files de messages

6.2.1. L'interface d'appel système des messages

6.2.1.1. sys_msgget()

L'appel à sys_msgget() est entièrement protégé par un sémaphore global de file de messages (Section 6.4.2.2, « struct ipc_ids »).

S'il faut créer une file de messages, la fonction Section 6.2.3.1, « newque() » est appelée pour créer et initialiser cette nouvelle file de messages, le nouvel identificateur de file est renvoyé à l'appelant.

Si une valeur de clef est fournie pour une file de messages existante, on appelle Section 6.4.1.7, « ipc_findkey() » pour retrouver l'index correspondant dans le tableau global des descripteurs de file de messages (msg_ids.entries). Les paramètres et permissions de l'appelant sont vérifiés avant de renvoyer l'identificateur de la file de messages. L'opération de recherche et de vérification est réalisée pendant que le verrou tournant global de file de messages est maintenu (msg_ids.ary).

6.2.1.2. sys_msgctl()

Les paramètres passés à sys_msgctl() sont : un identificateur de file de messages (`msqid`), l'opération (`cmd`), et un pointeur sur un tampon dans l'espace utilisateur du type Section 6.2.2.7, « struct msqid_ds » (`buf`). Cette fonction met à notre disposition six opérations : IPC_INFO, MSG_INFO, IPC_STAT, MSG_STAT, IPC_SET et IPC_RMID. L'identificateur de file de messages et les paramètres des opérations sont validés, puis l'opération (cmd) est exécutée comme suit :

6.2.1.2.1. IPC_INFO (or MSG_INFO)

L'information de la file de messages globale est copiée dans l'espace utilisateur.

6.2.1.2.2. IPC_STAT (or MSG_STAT)

Un tampon temporaire de type Section 6.2.2.6, « struct msqid64_ds » est initialisé et le verrou tournant global de file de messages est posé. Après vérification des permissions d'accès du processus appelant, l'information de la file de messages associée à l'identificateur de file de messages est chargée dans le tampon temporaire, le verrou tournant global de file de messages est déverrouillé, et le contenu du tampon temporaire est copié dans l'espace utilisateur par Section 6.2.3.13, « copy_msqid_to_user() ».

6.2.1.2.3. IPC_SET

Les données utilisateur sont copiées via Section 6.2.3.13, « copy_msqid_to_user() ». Le sémaphore global de file de messages et le verrou tournant sont récupérés et relâchés à la fin. Après que l'identificateur de la file de messages et que les permissions du processus courant ont été validés, l'information de la file de message est mise à jour avec les données fournies par l'utilisateur. Plus tard, Section 6.2.3.6, « expunge_all() » et Section 6.2.3.3, « ss_wakeup() » sont appelés pour réveiller tous les processus endormis dans les files d'attente des récepteurs et émetteurs de la file de messages. Ceci parce que, certains récepteurs peuvent maintenant être exclus par des permissions d'accès plus strictes et certains émetteurs devenir capables d'envoyer un message grâce à l'augmentation de la taille de la file d'attente.

6.2.1.2.4. IPC_RMID

Le sémaphore global de file de messages est obtenu et le verrou tournant global de file de messages verrouillé. Après validation de l'identificateur de la file de messages et des permissions d'accès de la tâche courante, Section 6.2.3.2, « freeque() » est appelée pour libérer les ressources relatives à l'identificateur de file de messages. Le sémaphore global de file de messages et le verrou tournant global de file de messages sont libérés.

6.2.1.3. sys_msgsnd()

sys_msgsnd() reçoit comme paramètres un identificateur de file de messages (`msqid`), un pointeur sur un tampon de type Section 6.2.2.2, « struct msg_msg » (`msgp`), la taille du message envoyé (`msgsz`), et un drapeau (`msgflg`) indiquant soit d'attendre (wait) soit de ne pas attendre (not wait). Il y a deux files d'attente de tâches et une file d'attente de messages associées à l'identificateur de file de messages. S'il y a une tâche dans la file d'attente de réception qui attend ce message, le message est délivré directement au récepteur, et le récepteur est réveillé. Autrement, s'il y a assez de place dans la file d'attente, le message est stocké dans cette file. En dernier recours, la tâche émettrice se met elle-même dans la file d'attente d'émission. Examinons de façon plus approfondie les opérations exécutées par sys_msgsnd() :

1. Elle valide l'adresse du tampon et le type de message, puis invoque Section 6.2.3.7, « load_msg() » pour charger le contenu du message utilisateur dans un objet temporaire `msg` de type Section 6.2.2.2, « struct msg_msg ». Les champs type de message et taille de message de `msg` sont aussi initialisés.

2. Elle verrouille le verrou tournant global de file de messages et obtient le descripteur de file de messages associé à l'identificateur de la file de messages. Si une telle file de messages n'existe pas, retourne EINVAL.

3. Invoque Section 6.4.1.5, « ipc_checkid() » (via msg_checkid()) pour vérifier que l'identificateur de file de messages est valide et appelle Section 6.4.1.8, « ipcperms() » pour vérifier les permissions du processus appelant.

4. Regarde la taille du message pour voir s'il reste assez de place dans la file d'attente pour le message. Si non, les actions suivantes sont exécutées :

   a. Si IPC_NOWAIT est spécifié dans `msgflg`, le verrou tournant global de file de messages est déverrouillé, les ressources mémoire pour ce message sont libérées, et EAGAIN est renvoyé.

   b. Elle invoque Section 6.2.3.4, « ss_add() » pour retirer la tâche courante de la file d'attente d'émission, déverrouille aussi le verrou tournant global de file de messages et invoque schedule() pour endormir la tâche courante.

   c. Une fois réveillée, obtient à nouveau le verrou tournant global et vérifie que l'identificateur de la file de messages est toujours valide. S'il n'est plus valide, ERMID est renvoyé.

   d. Invoque Section 6.2.3.5, « ss_del() » pour retirer la tâche émettrice de la file d'attente d'émission. S'il y a un signal suspendu pour la tâche, sys_msgsnd() déverrouille le verrou tournant global, invoque Section 6.2.3.9, « free_msg() » pour libérer le tampon du message, et renvoie EINTR. Autrement, la fonction retourne en arrière pour vérifier encore s'il y a assez de place dans la file d'attente des messages.

5. Invoque Section 6.2.3.12, « pipelined_send() » pour essayer d'envoyer directement le message au récepteur en attente.

6. S'il n'y a pas de récepteur attendant ce message, met `msg` dans la file d'attente des messages (msq->q_messages). Met à jour les champs `q_cbytes` et `q_qnum` pour le descripteur de file

de messages, ainsi que les variables globales `msg_bytes` et `msg_hdrs`, qui indiquent le nombre total d'octets utilisés pour les messages et le nombre total de messages dans tout le système.

7. Si le message a été envoyé ou enlevé de la file avec succès, met à jour les champs `q_lspid` et `q_stime` du descripteur de file de messages et relâche le verrou tournant global de file de messages.

6.2.1.4. sys_msgrcv()

La fonction sys_msgrcv() reçoit en paramètres un identificateur de file de messages (`msqid`), un pointeur sur un tampon du type Section 6.2.2.2, « struct msg_msg » (`msgp`), la taille de message désirée (`msgsz`), le type de message (`msgtyp`), et le drapeau (`msgflg`). Elle cherche dans la file d'attente de messages associée à l'identificateur de file de messages le premier message de la file qui correspond au type désiré, le copie dans le tampon utilisateur donné. Si on ne trouve pas de message de ce type, la tâche faisant la requête est mise dans la file d'attente de réception jusqu'à ce que le message désiré soit disponible. Examinons de façon plus approfondie les opérations effectuées par sys_msgrcv() :

1. D'abord, elle invoque Section 6.2.3.10, « convert_mode() » pour déduire le mode de recherche de `msgtyp`. Ensuite sys_msgrcv() verrouille le verrou tournant global de file de messages et obtient le descripteur de file de messages associé à l'identificateur de file de messages. Si celle file de messages n'existe pas, renvoie EINVAL.

2. Vérifie si la tâche courante a les permissions voulues pour accéder à la file de messages.

3. Appelle Section 6.2.3.11, « testmsg() » sur chaque message de la file d'attente en commençant par le premier pour vérifier si le type du message correspond au type demandé. sys_msgrcv() continue sa recherche jusqu'à ce qu'elle trouve un message du type voulu ou jusqu'à la fin de la liste. Si le mode de recherche est SEARCH_LESSEQUAL, c'est le premier message de la file dont le type est inférieur au égal à `msgtyp` qui est cherché.

4. Si le message est trouvé, sys_msgrcv() exécute les sous-étapes suivantes :

   a. Si la taille du message est plus grande que celle attendue et si `msgflg` indique qu'aucune erreur n'est autorisée, on déverrouille le verrou tournant global de file de messages et on renvoie E2BIG.

   b. On retire le message de la file d'attente et on met à jour les statistiques de la file de messages.

   c. On réveille toutes les tâches endormies dans la file d'attente d'émission. Après le retrait du message de la file à l'étape précédente, l'un des émetteurs peut continuer. On va à la dernière étape".

5. Si aucun message correspondant aux critères des récepteurs n'est trouvé dans la file d'attente de messages, `msgflg` est vérifié. Si IPC_NOWAIT est mis, le verrou tournant global de file de messages est levé et ENOMSG est retourné. Autrement, le récepteur est mis dans la file d'attente de réception comme suit :

   a. Une structure de données Section 6.2.2.5, « struct msg_receiver » `msr` est allouée et est ajoutée en tête de la file d'attente.

   b. Le champ `r_tsk` de `msr` prend pour valeur la tâche courante.

   c. Les champs `r_msgtype` et `r_mode` sont initialisés avec respectivement le type de message et le mode voulus.

d. Si `msgflg` indique MSG_NOERROR, alors le champ r_maxsize de `msr` prend pour valeur `msgsz`, autrement INT_MAX.

e. Le champs `r_msg` est initialisé pour indiquer qu'aucun message n'a encore été reçu.

f. Une fois l'initialisation terminée, l'état de la tâche réceptrice est mis à TASK_INTERRUPTIBLE, le verrou tournant global de file de messages est enlevé, et schedule() est invoquée.

6. Une fois le récepteur réveillé, le champ `r_msg` de `msr` est vérifié. Ce champ est utilisé pour stocker le message envoyé à travers le tube ou en cas d'erreur, pour stocker le statut d'erreur. Si le champ `r_msg` contient le message voulu, alors on va à la dernière étape". Sinon, le verrou tournant global de file de messages est une nouvelle fois posé.

7. Une fois le verrou tournant obtenu, le champ `r_msg` est re-vérifié pour voir si le message a été reçu pendant qu'on attendait le verrou. Si le message a été reçu, on va à la dernière étape".

8. Si le champ `r_msg` est resté inchangé, c'est que la tâche a été est réveillée pour une nouvelle tentative. Dans ce cas, `msr` est enlevé de la file. S'il y a un signal suspendu pour la tâche, on enlève le verrou tournant global de file de messages et on renvoie EINTR. Sinon, la fonction doit revenir en arrière et réessayer.

9. Si le champ `r_msg` montre qu'une erreur s'est produite pendant le sommeil, le verrou tournant global de file de messages est levé et l'erreur est renvoyée.

10. Après avoir vérifié l'adresse du tampon utilisateur `msp`, le type de message est chargé dans le champ `mtype` de `msp`, et Section 6.2.3.8, « store_msg() » est invoquée pour copier le contenu du message dans le champ `mtext` de `msp`. Finalement la mémoire occupée par ce message est libérée par la fonction Section 6.2.3.9, « free_msg() ».

6.2.2. Structures spécifiques aux messages

Les structures de données pour les files de messages sont définies dans msg.c.

6.2.2.1. struct msg_queue

```
/* une structure msq_queue pour chaque file présente dans le système */
struct msg_queue {
        struct kern_ipc_perm q_perm;
        time_t q_stime;                 /* instant du dernier msgsnd */
        time_t q_rtime;                 /* instant du dernier msgrcv */
        time_t q_ctime;                 /* instant du dernier changement */
        unsigned long q_cbytes;         /* nb actuel d'octets dans la file */
        unsigned long q_qnum;           /* nb de messages dans la file */
        unsigned long q_qbytes;         /* nb maxi d'octets dans la file */
        pid_t q_lspid;                  /* pid du dernier msgsnd */
        pid_t q_lrpid;                  /* pid de la dernière réception */

        struct list_head q_messages;
        struct list_head q_receivers;
        struct list_head q_senders;
};
```

6.2.2.2. struct msg_msg

```
/* une structure msg_msg pour chaque message */
struct msg_msg {
        struct list_head m_list;
```

```
        long  m_type;
        int m_ts;           /* taille du texte du message */
        struct msg_msgseg* next;
        /* le message proprement dit suit immédiatement */
};
```

6.2.2.3. struct msg_msgseg

```
/* un segment de message pour chaque message */
struct msg_msgseg {
        struct msg_msgseg* next;
        /* la partie suivante du message suit immédiatement */
};
```

6.2.2.4. struct msg_sender

```
/* un msg_sender pour chaque émetteur endormi */
struct msg_sender {
        struct list_head list;
        struct task_struct* tsk;
};
```

6.2.2.5. struct msg_receiver

```
/* une structure msg_receiver pour chaque émetteur endormi */
struct msg_receiver {
        struct list_head r_list;
        struct task_struct* r_tsk;

        int r_mode;
        long r_msgtype;
        long r_maxsize;

        struct msg_msg* volatile r_msg;
};
```

6.2.2.6. struct msqid64_ds

```
struct msqid64_ds {
        struct ipc64_perm msg_perm;
        __kernel_time_t msg_stime;      /* instant du dernier msgsnd */
        unsigned long   __unused1;
        __kernel_time_t msg_rtime;      /* instant du dernier msgrcv */
        unsigned long   __unused2;
        __kernel_time_t msg_ctime;      /* instant du dernier changement */
        unsigned long   __unused3;
        unsigned long  msg_cbytes;      /* nb actuel d'octets dans la file */
        unsigned long  msg_qnum;        /* nb de messages dans la file */
        unsigned long  msg_qbytes;      /* nb max d'octets dans la file */
        __kernel_pid_t msg_lspid;       /* pid du dernier msgsnd */
        __kernel_pid_t msg_lrpid;       /* pid de la dernière réception */
        unsigned long  __unused4;
        unsigned long  __unused5;
};
```

6.2.2.7. struct msqid_ds

```
 struct msqid_ds {
        struct ipc_perm msg_perm;
        struct msg *msg_first;          /* premier message de la file, inutilis
        struct msg *msg_last;           /* dernier message de la file, inutilis
        __kernel_time_t msg_stime;      /* instant du dernier msgsnd  */
        __kernel_time_t msg_rtime;      /* instant du dernier msgrcv */
        __kernel_time_t msg_ctime;      /* instant du dernier changement */
        unsigned long  msg_lcbytes;     /* recycler les champs pour 32 bits */
        unsigned long  msg_lqbytes;     /* idem */
        unsigned short msg_cbytes;      /* nb actuel d'octets dans la file */
        unsigned short msg_qnum;        /* nb de messages dans la file */
        unsigned short msg_qbytes;      /* nb max d'octets dans la file */
        __kernel_ipc_pid_t msg_lspid;   /* pid du dernier msgsnd */
        __kernel_ipc_pid_t msg_lrpid;   /* pid de la dernière réception */
};
```

6.2.2.8. msg_setbuf

```
struct msq_setbuf {
        unsigned long   qbytes;
        uid_t           uid;
        gid_t           gid;
        mode_t          mode;
};
```

6.2.3. Fonctions de support des messages

6.2.3.1. newque()

newque() alloue la mémoire pour un nouveau descripteur de file de messages (Section 6.2.2.1, « struct msg_queue ») puis appelle Section 6.4.1.2, « ipc_addid() », qui réserve une entrée dans le tableau des files de messages pour le nouveau descripteur. Le descripteur de message est initialisé comme suit :

- La structure Section 6.4.2.1, « struct kern_ipc_perm » est initialisée.
- Les champs `q_stime` et `q_rtime` du descripteur de message sont initialisés à 0. Le champ `q_ctime` est mis à CURRENT_TIME.
- Le nombre maximum d'octets alloués pour cette file de messages (`q_qbytes`) est mis à MSGMNB, et le nombre d'octets utilisés actuellement par la file (`q_cbytes`) est initialisé à 0.
- La file d'attente de messages (`q_messages`), la file d'attente de réception (`q_receivers`), la file d'attente d'émission (`q_senders`) sont initialisées vides toutes les trois.

Pour toutes les opérations suivant l'appel de Section 6.4.1.2, « ipc_addid() », le verrou tournant global de file de messages est maintenu. Une fois le verrou tournant déverrouillé, newque() appelle msg_buildid(), qui est en fait Section 6.4.1.4, « ipc_buildid() ». Section 6.4.1.4, « ipc_buildid() » utilise l'index du descripteur de la file de messages pour créer un identificateur de file de messages unique qui est renvoyé à l'appelant de newque().

6.2.3.2. freeque()

Quand une file de messages doit être retirée, la fonction freeque() est appelée. Cette fonction suppose que le verrou tournant global de file de messages a déjà été verrouillé par la fonction appelante. Elle libère les ressources noyau associées à cette file de messages. D'abord, elle appelle Section 6.4.1.3, « ipc_rmid() » (via msg_rmid()) pour retirer le descripteur de file de messages du tableau global des descripteurs de file de messages. Ensuite elle appelle Section 6.2.3.6, « expunge_all() » pour réveiller les récepteurs et Section 6.2.3.3, « ss_wakeup() » pour réveiller les

émetteurs endormis dans cette file de messages. Plus tard le verrou tournant global de file de messages est relâché. Tous les messages stockées dans la file de messages sont libérés et la mémoire occupée par le descripteur de file de messages est libérée.

6.2.3.3. ss_wakeup()

ss_wakeup() réveille toutes les tâches attendant dans une file d'attente d'émission de messages donnée. Cette fonction est appelée par Section 6.2.3.2, « freeque() », ensuite tous les émetteurs de la file sont retirés.

6.2.3.4. ss_add()

ss_add() reçoit en paramètres un descripteur de file de messages et la structure de données d'un récepteur de message. Le champ `tsk` de la structure de données du récepteur de message prend pour valeur le processus courant, l'état du processus courant devient TASK_INTERRUPTIBLE, ensuite elle insère la structure de donnée du récepteur de message en tête de la file d'attente d'émission de la file de messages donnée.

6.2.3.5. ss_del()

Si la structure de données considérée de l'émetteur de message (`mss`) est toujours associée à une file d'attente d'émission, alors ss_del() retire `mss` de la file.

6.2.3.6. expunge_all()

expunge_all() reçoit en paramètres un descripteur de file de messages (`msq`) et la valeur d'un entier (`res`) indiquant la raison du réveil des récepteurs. Pour chaque récepteur endormi associé à `msq`, le champ `r_msg` prend pour valeur cette raison (`res`), et la tâche associée est réveillée. Cette fonction est appelée quand un message est retiré ou quand une opération de contrôle des messages est effectuée.

6.2.3.7. load_msg()

Quand un processus envoie un message, la fonction Section 6.2.1.3, « sys_msgsnd() » invoque d'abord la fonction load_msg() qui charge le message de l'espace utilisateur vers l'espace noyau. Le message est représenté dans la mémoire du noyau comme une liste chaînée de blocs. Associée au premier, la structure Section 6.2.2.2, « struct msg_msg » décrit l'ensemble du message. Le bloc de données associé à la structure msg_msg a une taille limitée à DATA_MSG_LEN. L'allocation du bloc de données et de la structure s'effectue dans un bloc de données mémoire contigu dont la taille peut aller jusqu'à une page mémoire. Si le message complet ne tient pas dans le premier bloc de données, des blocs additionnels sont alloués et sont organisés en liste chaînée simple. Ces blocs additionnels ont une taille limitée à DATA_SEG_LEN, et chacun comprend une structure Section 6.2.2.3, « struct msg_msgseg » associée. La structure msg_msgseg et le bloc de données associé sont alloués dans un bloc de données mémoire contigu dont la taille peut aller jusqu'à une page mémoire. Cette fonction renvoie l'adresse de la nouvelle structure Section 6.2.2.2, « struct msg_msg » en cas de succès.

6.2.3.8. store_msg()

La fonction store_msg() est appelée par Section 6.2.1.4, « sys_msgrcv() » pour reconstituer un message reçu dans le tampon de l'espace utilisateur fourni par l'appelant. Les données décrites par la structure Section 6.2.2.2, « struct msg_msg » et toutes les structures Section 6.2.2.3, « struct msg_msgseg » sont copiées à la suite dans le tampon de l'espace utilisateur.

6.2.3.9. free_msg()

La fonction free_msg() libère la mémoire occupée par une structure de données Section 6.2.2.2, « struct msg_msg » d'un message, et les segments du message.

6.2.3.10. convert_mode()

convert_mode() est appelée par Section 6.2.1.4, « sys_msgrcv() ». Elle reçoit en paramètres

l'adresse du type de message spécifié (`msgtyp`) et un drapeau (`msgflg`). Elle renvoie le mode de recherche déterminé d'après la valeur de `msgtyp` et `msgflg`. Si `msgtyp` est nul, alors SEARCH_ANY est renvoyé. Si `msgtyp` est négatif, alors `msgtyp` est remplacé par sa valeur absolue et SEARCH_LESSEQUAL est renvoyé. Si MSG_EXCEPT est spécifié dans `msgflg`, alors SEARCH_NOTEQUAL est renvoyé. Sinon SEARCH_EQUAL est retourné.

6.2.3.11. testmsg()

La fonction testmsg() vérifie si le message correspond aux critères spécifiés par le récepteur. Elle renvoie 1 si une des conditions suivantes est vraie :

- Le mode de recherche indique de rechercher tous les messages (SEARCH_ANY).
- Le mode de recherche est SEARCH_LESSEQUAL et le type de message est inférieur ou égal au type désiré.
- Le mode de recherche est SEARCH_EQUAL et le type de message est le même que celui demandé.
- Le mode de recherche est SEARCH_NOTEQUAL et le type de message n'est pas celui désiré.

6.2.3.12. pipelined_send()

pipelined_send() autorise un processus à envoyer directement un message à un récepteur en attente plutôt que de l'insérer dans la file d'attente de messages associée. La fonction Section 6.2.3.11, « testmsg() » est invoquée pour trouver le premier récepteur qui attend le message donné. S'il est trouvé, ce récepteur est retiré de la file d'attente de réception, et la tâche réceptrice associée est réveillée. On stocke le message dans le champ `r_msg` du récepteur, on renvoie 1. Si aucun récepteur n'attend le message, on renvoie 0.

En cherchant un récepteur, on peut trouver des candidats récepteurs d'une taille trop petite pour le message donné. Ces récepteurs sont retirés de la file, et sont réveillés avec un statut d'erreur de E2BIG, qui est stocké dans le champ `r_msg`. La recherche continue ensuite jusqu'à ce qu'on trouve un récepteur valide ou jusqu'à la fin de la file.

6.2.3.13. copy_msqid_to_user()

copy_msqid_to_user() copie le contenu d'un tampon noyau dans le tampon utilisateur. Elle reçoit en paramètres un tampon utilisateur, un tampon noyau du type Section 6.2.2.6, « struct msqid64_ds », et un drapeau de version indiquant si c'est la nouvelle version des IPC ou l'ancienne. Si le drapeau de version est égal à IPC_64, copy_to_user() est invoquée pour copier directement depuis le tampon noyau vers le tampon utilisateur . Sinon, un tampon temporaire de type struct msqid_ds est initialisé, et les données noyau sont transférées vers ce tampon temporaire. Plus tard copy_to_user() sera appelée pour copier le contenu du tampon temporaire dans le tampon utilisateur.

6.2.3.14. copy_msqid_from_user()

La fonction copy_msqid_from_user() reçoit en paramètres un tampon message noyau du type struct msq_setbuf, un tampon utilisateur et un drapeau de version indiquant si c'est la nouvelle version des IPC ou l'ancienne. Dans le cas de la nouvelle version, copy_from_user() est appelée pour copier le contenu du tampon utilisateur dans un tampon utilisateur temporaire de type Section 6.2.2.6, « struct msqid64_ds ». Puis les champs `qbytes`, `uid`, `gid`, et `mode` du tampon noyau sont renseignés avec les valeurs correspondantes du tampon temporaire. Dans le cas de la vieille version des IPC, un tampon temporaire de type struct Section 6.2.2.7, « struct msqid_ds » est utilisé à la place.

6.3. La mémoire partagée

6.3.1. Interfaces d'appels système de la mémoire partagée

6.3.1.1. sys_shmget()

L'appel à sys_shmget() est complètement protégé par un sémaphore global de mémoire partagée.

Si l'on a besoin d'un nouveau segment de mémoire partagée, on appelle la fonction Section 6.3.3.1, « newseg() » pour créer et initialiser ce segment. L'identificateur du nouveau segment est retourné à l'appelant.

Dans le cas où une valeur de clef est fournie pour un segment de mémoire partagée existant, on cherche l'index correspondant dans le tableau des descripteurs de mémoire partagée, et on vérifie les paramètres et les permissions de l'appelant avant de retourner l'identificateur du segment de mémoire partagée. L'opération de recherche et de vérification est réalisée avec le verrou tournant global de mémoire partagée maintenu.

6.3.1.2. sys_shmctl()

6.3.1.2.1. IPC_INFO

Un tampon temporaire Section 6.3.2.1, « struct shminfo64 » est chargé avec tous les paramètres de mémoire partagée du système et est copié dans l'espace utilisateur pour que l'application appelante puisse y accéder.

6.3.1.2.2. SHM_INFO

Le sémaphore global de mémoire partagée et le verrou tournant global de mémoire partagée sont maintenus pendant que l'on recherche les informations statistiques pour la mémoire partagée à l'échelle du système. La fonction Section 6.3.3.2, « shm_get_stat() » est appelée pour calculer d'une part le nombre de pages de mémoire partagée qui résident en mémoire et d'autre part le nombre de pages de mémoire partagée qui ont été transférées dans la mémoire virtuelle (swap). Il y a d'autres statistiques comme le nombre total de pages de mémoire partagée et le nombre de segments de mémoire partagée en cours d'utilisation. Les nombres `swap_attempts` et `swap_successes` sont constants et nuls (codés en dur). Ces statistiques sont stockées dans un tampon temporaire Section 6.3.2.2, « struct shm_info » et copiées dans l'espace utilisateur pour l'application appelante.

6.3.1.2.3. SHM_STAT, IPC_STAT

Pour SHM_STAT et IPC_STATA, un tampon temporaire de type Section 6.3.2.4, « struct shmid64_ds » est initialisé, et le verrou tournant global de mémoire partagée est verrouillé.

Dans le cas de SHM_STAT, Le paramètre identificateur du segment de mémoire partagée attendu est un numéro d'index (i.e. il vaut de 0 à n où n est le nombre d'identificateurs de mémoire partagée dans le système). Après avoir validé l'index, Section 6.4.1.4, « ipc_buildid() » est appelée (via shm_buildid()) pour convertir l'index en identificateur de mémoire partagée. Dans le cas de SHM_STAT, c'est l'identificateur de mémoire partagée qui sera la valeur retournée. Remarquez que c'est une caractéristique non documentée, mais maintenue du programme ipcs(8).

Dans le cas IPC_STAT, l'identificateur du segment de mémoire partagée attendu doit avoir été généré par un appel à Section 6.3.1.1, « sys_shmget() ». L'identificateur est validé avant l'exécution. Dans le cas de IPC_STAT, c'est 0 qui sera la valeur de retour.

Pour SHM_STAT comme pour IPC_STAT, les permissions d'accès de l'appelant sont vérifiées. Les statistiques voulues sont chargées dans un tampon temporaire et ensuite copiées vers l'application appelante.

6.3.1.2.4. SHM_LOCK, SHM_UNLOCK

Après validation des permissions d'accès, le verrou tournant global de mémoire partagée est verrouillé, et l'identificateur de segment de mémoire partagée est validé. La fonction Section 6.3.3.3, « shmem_lock() » est appelée pour SHM_LOCK comme pour SHM_UNLOCK. Les paramètres de Section 6.3.3.3, « shmem_lock() » identifient la fonction à exécuter.

6.3.1.2.5. IPC_RMID

Durant IPC_RMID, le sémaphore global de mémoire partagée et le verrou tournant global de mémoire partagée sont maintenus tout du long. L'identificateur de mémoire partagée est validé, et en-

suite, s'il n'y pas d'attachements, Section 6.3.3.4, « shm_destroy() » est appelée pour détruire le segment de mémoire partagée. Sinon, le drapeau SHM_DEST est mis pour le marquer « à détruire », et le drapeau IPC_PRIVATE pour empêcher qu'un autre processus puisse référencer l'identificateur de mémoire partagée.

6.3.1.2.6. IPC_SET

Après validation de l'identificateur du segment de mémoire partagée et des permissions utilisateur, les drapeaux `uid`, `gid`, et `mode` du segment de mémoire partagée sont mis à jour avec les données utilisateur. Le champ `shm_ctime` peut aussi être mis à jour. Ces changement sont réalisés pendant que le sémaphore global de mémoire partagée et le verrou tournant global de mémoire partagée sont maintenus.

6.3.1.3. sys_shmat()

sys_shmat() prend comme paramètres un identificateur de segment de mémoire partagée, une adresse à laquelle le segment doit être attaché (`shmaddr`), et des drapeaux qui seront décrits ci-dessous.

Si `shmaddr` est différent de zéro, et que le drapeau SHM_RND est spécifié, alors `shmaddr` est arrondi pour devenir un multiple de SHMLBA. Si `shmaddr` n'est pas un multiple de SHMLBA et que SHM_RND n'est pas spécifié, EINVAL est renvoyé.

Les permissions d'accès de l'appelant sont validées et le champ `shm_nattch` du segment de mémoire partagée est incrémenté. Remarquez que cet incrémentation garantit que le compteur de liens soit non nul et empêche la destruction du segment de mémoire partagée durant le processus d'attachement au segment. Ces opérations sont exécutées sous la protection du verrou tournant global de mémoire partagée.

La fonction do_mmap() est appelée pour créer une correspondance entre la mémoire virtuelle et les pages du segment de mémoire partagée. C'est fait en maintenant le sémaphore `mmap_sem` de la tâche courante. Le drapeau MAP_SHARED est passé à do_mmap(). Si une adresse est fournie par l'appelant, le drapeau MAP_FIXED est aussi passé à do_mmap(). Sinon, do_mmap() choisira une adresse virtuelle pour le correspondant du segment de mémoire partagée.

Remarquez que Section 6.3.3.5, « shm_inc() » sera invoquée à l'intérieur de la fonction do_mmap() via la structure `shm_file_operations`. Cette fonction est appelée pour fixer le PID et le temps courant, et pour incrémenter le nombre d'attachements à ce segment de mémoire partagée.

Après l'appel à do_mmap(), le sémaphore global de mémoire partagée et le verrou tournant global de mémoire partagée sont obtenus tous les deux. Le compteur d'attachements est ensuite décrémenté. Le solde des changements pour ce compteur est de 1 après l'appel à shmat(), à cause de l'appel à Section 6.3.3.5, « shm_inc() ». Si, après avoir décrémenté le compteur d'attachements, celui-ci devient nul, et si le segment est marqué « à détruire » (SHM_DEST), alors Section 6.3.3.4, « shm_destroy() » est appelée pour libérer les ressources du segment de mémoire partagée.

Finalement, l'adresse virtuelle à laquelle la mémoire partagée est associée est renvoyée à l'appelant à l'adresse spécifiée par l'utilisateur. Si un code d'erreur a été retourné par do_mmap(), ce code est passé comme valeur de retour de l'appel système.

6.3.1.4. sys_shmdt()

Le sémaphore global de mémoire partagée est maintenu pendant l'exécution de sys_shmdt(). Dans la `mm_struct` du processus courant on cherche la `vm_area_struct` associée à l'adresse de la mémoire partagée. Quand elle est trouvée, do_munmap() est appelée pour supprimer la correspondance avec l'adresse virtuelle pour le segment de mémoire partagée.

Remarquez que do_munmap() fait un rappel à Section 6.3.3.6, « shm_close() », qui exécute les fonctions de comptabilité de la mémoire partagée, et libère les ressources du segment de mémoire partagée s'il n'y plus d'autre attachement. sys_shmdt() retourne toujours 0.

6.3.2. Les structures de support de la mémoire partagée

6.3.2.1. struct shminfo64

```
struct shminfo64 {
        unsigned long   shmmax;
        unsigned long   shmmin;
        unsigned long   shmmni;
        unsigned long   shmseg;
        unsigned long   shmall;
        unsigned long   __unused1;
        unsigned long   __unused2;
        unsigned long   __unused3;
        unsigned long   __unused4;
};
```

6.3.2.2. struct shm_info

```
struct shm_info {
        int used_ids;
        unsigned long shm_tot;  /* shm total alloué */
        unsigned long shm_rss;  /* shm total résident */
        unsigned long shm_swp;  /* shm total copié en mémoire virtuelle (swap)*/
        unsigned long swap_attempts;
        unsigned long swap_successes;
};
```

6.3.2.3. struct shmid_kernel

```
struct shmid_kernel /* données privées du noyau */
{
        struct kern_ipc_perm    shm_perm;
        struct file *           shm_file;
        int                     id;
        unsigned long           shm_nattch;
        unsigned long           shm_segsz;
        time_t                  shm_atim;
        time_t                  shm_dtim;
        time_t                  shm_ctim;
        pid_t                   shm_cprid;
        pid_t                   shm_lprid;
};
```

6.3.2.4. struct shmid64_ds

```
struct shmid64_ds {
    struct ipc64_perm   shm_perm;       /* permissions pour les opérations */
    size_t              shm_segsz;      /* taille du segment (en octets) */
    __kernel_time_t     shm_atime;      /* instant du dernier attachement */
    unsigned long       __unused1;
    __kernel_time_t     shm_dtime;      /* instant du dernier détachement */
    unsigned long       __unused2;
    __kernel_time_t     shm_ctime;      /* instant du dernier chgt */
    unsigned long       __unused3;
    __kernel_pid_t      shm_cpid;       /* pid du créateur */
    __kernel_pid_t      shm_lpid;       /* pid de la dernière opération */
    unsigned long       shm_nattch;     /* nb d'attachements actuel */
    unsigned long       __unused4;
    unsigned long       __unused5;
};
```

6.3.2.5. struct shmem_inode_info

```
struct shmem_inode_info {
    spinlock_t       lock;
    unsigned long    max_index;
    swp_entry_t      i_direct[SHMEM_NR_DIRECT]; /* pour les premiers blocs */
    swp_entry_t      **i_indirect;  /* double indirection pour les blocs */
    unsigned long    swapped;
    int              locked;        /* dans la mémoire */
    struct list_head list;
};
```

6.3.3. Les fonctions du support de la mémoire partagée

6.3.3.1. newseg()

La fonction newseg() est appelée quand il faut créer un nouveau segment de mémoire partagée. Elle agit sur 3 paramètres du nouveau segment : la clef, le drapeau et la taille. Ayant vérifié que la taille du segment de mémoire partagée créé est entre SHMMIN et SHMMAX et que le nombre total de segments de mémoire partagée ne dépasse pas SHMALL, elle alloue un nouveau descripteur de segment de mémoire partagée. La fonction Section 6.3.3.7, « shmem_file_setup() » est invoquée plus tard pour créer un fichier non-lié de type tmpfs. Le pointeur de fichier renvoyé est sauvegardé dans le champ `shm_file` du descripteur de segment de mémoire partagée associé. La taille du fichier est fixée égale à la taille du segment. Le nouveau descripteur de segment de mémoire partagée est initialisé et inséré dans le tableau IPC global des descripteurs de mémoire partagée. L'identificateur du segment de mémoire partagée est créé par shm_buildid() (via Section 6.4.1.4, « ipc_buildid() »). Cet identificateur de segment est sauvegardé dans le champ `id` du descripteur de segment de mémoire partagée, ainsi que dans le champ `i_ino` de l'inode associé. De plus, l'adresse des opérations de mémoire partagée définie dans la structure `shm_file_operation` est stockée dans le fichier associé. La valeur de la variable globale `shm_tot`, qui indique le nombre total de segments de mémoire partagée du système, est aussi augmentée pour refléter ce changement. En cas de succès, l'identificateur du segment est renvoyé à l'application appelante.

6.3.3.2. shm_get_stat()

shm_get_stat() fait le tour des structures de mémoire partagée, et calcule le nombre total de pages utilisées par la mémoire partagée et le nombre total de pages de mémoire partagée qui ont été copiées dans la mémoire virtuelle sur le disque. Il y a une structure de fichier et une structure d'inode pour chaque segment de mémoire partagée. Comme les données sont obtenues via l'inode, le verrou tournant de chaque structure d'inode accédée est verrouillé et déverrouillé successivement.

6.3.3.3. shmem_lock()

shmem_lock() reçoit en paramètres un pointeur sur le descripteur du segment de mémoire partagée et un drapeau indiquant s'il est verrouillé ou déverrouillé. L'état de verrouillage du segment de mémoire partagée est stocké dans l'inode associé. Cet état est comparé avec l'état désiré ; shmem_lock() retourne simplement s'ils correspondent.

Tout en maintenant le sémaphore associé à l'inode, l'inode est mis dans l'état verrouillé. Ce qui suit est réalisé pour chaque page dans le segment de mémoire partagée :

- find_lock_page() est appelée pour verrouiller la page (en positionnant PG_locked) et pour incrémenter le compteur de références de la page. L'incrémentation du compteur de références assure que le segment de mémoire partagée reste verrouillé tout au long de l'opération.
- Si l'état désiré est verrouillé, PG_locked est nettoyé, mais le compteur de références reste incrémenté.
- Si l'état désiré est déverrouillé, le compteur de références est décrémenté deux fois, une pour la référence courante, et une pour la référence existante qui a obligé la page à rester verrouillée en

mémoire. Alors PG_locked est nettoyé.

6.3.3.4. shm_destroy()

Pendant shm_destroy(), le nombre total de pages de mémoire partagée est ajusté pour prendre en compte le retrait du segment de mémoire partagée. Section 6.4.1.3, « ipc_rmid() » est appelée (via shm_rmid()) pour retirer l'identificateur de mémoire partagée. Section 6.3.3.3, « shmem_lock() » est appelée pour déverrouiller les pages de mémoire partagée, ramenant à zéro le compteur de références de chaque page. fput() est appelée pour décrémenter le compteur d'utilisations `f_count` de l'objet fichier associé, et si nécessaire, pour libérer les ressources de l'objet fichier. kfree() est appelée pour libérer le descripteur du segment de mémoire partagée.

6.3.3.5. shm_inc()

shm_inc() fixe le PID, le temps actuel, et incrémente le nombre d'attachements pour le segment de mémoire partagée donné. Ces opérations sont réalisées avec le verrou tournant global de mémoire partagée mis.

6.3.3.6. shm_close()

shm_close() met à jour les champs `shm_lprid` et `shm_dtim` et décrémente le nombre de segments de mémoire partagée attachés. S'il n'y a plus d'attachement au segment de mémoire partagée, alors Section 6.3.3.4, « shm_destroy() » est appelée pour libérer les ressources du segment de mémoire partagée. Ces opérations sont réalisées à la fois sous le sémaphore global de mémoire partagée et sous le verrou tournant global de mémoire partagée.

6.3.3.7. shmem_file_setup()

La fonction shmem_file_setup() met en place un fichier non-lié dans le système de fichiers tmpfs, de nom et de taille donnés. Si les ressources mémoire sont suffisantes pour ce fichier, elle crée un nouveau dentry sous le point de montage racine de tmpfs, et alloue un nouveau descripteur de fichier et un nouvel objet inode de type tmpfs. Ensuite, elle associe le nouvel objet dentry avec le nouvel objet inode en appelant d_instantiate() et sauve l'adresse de l'objet dentry dans le descripteur de fichier. Le champ `i_size` de l'objet inode est mis à la taille du fichier et le champ `i_nlink` est mis à zéro pour marquer l'inode comme non-lié. De plus, shmem_file_setup() stocke l'adresse de la structure d'opérations `shmem_file_operations` dans le champ `f_op`, et initialise les champs `f_mode` et `f_vfsmnt` du descripteur de fichier. La fonction shmem_truncate() est appelée pour terminer l'initialisation de l'objet inode. En cas de succès, shmem_file_setup() renvoie le descripteur du nouveau fichier

6.4. Les primitives des IPC Linux

6.4.1. Les primitives génériques des IPC Linux utilisées avec les sémaphores, les messages et la mémoire partagée

Les mécanismes de sémaphores, de messages et de mémoire partagée de Linux sont construits sur un ensemble de primitives communes. Ces primitives sont décrites dans la section ci-dessous.

6.4.1.1. ipc_alloc()

Si la mémoire à allouer est plus grande que PAGE_SIZE, vmalloc() est utilisée pour allouer la mémoire. Sinon, c'est kmalloc() qui est appelée avec GFP_KERNEL.

6.4.1.2. ipc_addid()

Quand un nouvel ensemble de sémaphores, une file de messages, ou un segment de mémoire partagée est ajouté, ipc_addid() appelle d'abord Section 6.4.1.6, « grow_ary() » pour s'assurer que la taille du tableau de descripteurs correspondant est suffisante en regard des possibilités maximum du système. Le tableau de descripteurs est parcouru pour trouver le premier élément inutilisé. Si un élé-

ment inutilisé est trouvé, le compteur des descripteurs utilisés est incrémenté. La structure Section 6.4.2.1, « struct kern_ipc_perm » pour le nouveau descripteur de ressource est initialisée, et l'index du tableau pour le nouveau descripteur est renvoyé. Si ipc_addid() réussit, elle retourne avec le verrou tournant global verrouillé pour l'IPC donnée.

6.4.1.3. ipc_rmid()

ipc_rmid() retire un descripteur d'IPC du tableau global des descripteurs du type IPC, met à jour le compteur des identificateurs qui sont en cours d'utilisation, ajuste l'identificateur maximum dans le tableau de descripteurs correspondant si nécessaire. Un pointeur sur le descripteur d'IPC associé à l'identificateur d'IPC donné est renvoyé.

6.4.1.4. ipc_buildid()

ipc_buildid() crée un identificateur unique associé à chaque descripteur d'un type d'IPC donné. Cet identificateur est créé au moment où le nouvel élément IPC est ajouté (i.e. un nouveau segment de mémoire partagée ou un nouvel ensemble de sémaphores). Les identificateurs IPC sont facilement convertis en index du tableau de descripteurs correspondant. Pour chaque type d'IPC, on maintient un numéro de séquence qui est incrémenté à chaque fois qu'un descripteur est ajouté. Un identificateur est créé en multipliant le numéro de séquence par SEQ_MULTIPLIER et en ajoutant le produit à l'index du tableau de descripteurs. Le numéro de séquence utilisé pour créer un identificateur d'IPC particulier est stocké dans le descripteur correspondant. L'existence du numéro de séquence rend possible la détection des identificateurs d'IPC dépassés.

6.4.1.5. ipc_checkid()

ipc_checkid() divise l'identificateur IPC donné par SEQ_MULTIPLIER et compare le quotient avec la valeur sauvegardée seq du descripteur correspondant. Si elles sont égales, l'identificateur IPC est considéré comme valide et on renvoie 1. Autrement, on renvoie 0.

6.4.1.6. grow_ary()

grow_ary() offre la possibilité que le nombre maximum (paramétrable) d'identificateurs pour un type d'IPC donné soit changé dynamiquement. Elle force la limite maximum actuelle a rester inférieure ou égale à la limite permanente du système (IPCMNI) et la diminue si nécessaire. Elle s'assure aussi que le tableau des descripteurs est assez grand. Si la taille du tableau existant est assez grande, la limite maximum courante est renvoyée. Autrement, un nouveau tableau plus grand est alloué, l'ancien tableau est copié dans le nouveau, et l'ancien tableau est libéré. Le verrou tournant global correspondant est maintenu pendant la mise à jour du tableau de descripteurs du type d'IPC donné.

6.4.1.7. ipc_findkey()

ipc_findkey() parcourt le tableau de descripteurs de l'objet spécifié Section 6.4.2.2, « struct ipc_ids », et cherche la clef spécifiée. Une fois trouvée, l'index du descripteur correspondant est renvoyé. Si la clef n'est pas trouvée, alors -1 est retourné.

6.4.1.8. ipcperms()

ipcperms() vérifie les permissions utilisateur, groupe, et autres pour l'accès aux ressources IPC. Elle retourne 0 si la permission est donnée et -1 sinon.

6.4.1.9. ipc_lock()

ipc_lock() prend un identificateur d'IPC comme l'un de ses paramètres. Elle verrouille le verrou tournant global pour le type donné d'IPC, et renvoie un pointeur sur le descripteur correspondant à l'identificateur IPC spécifié.

6.4.1.10. ipc_unlock()

ipc_unlock() relâche le verrou tournant pour le type d'IPC spécifié.

6.4.1.11. ipc_lockall()

ipc_lockall() verrouille le verrou tournant global pour le mécanisme d'IPC donné (i.e. mémoire partagée, sémaphores, et messages).

6.4.1.12. ipc_unlockall()

ipc_unlockall() déverrouille le verrou tournant global pour le mécanisme d'IPC donné (i.e. mémoire partagée, sémaphores, et messages).

6.4.1.13. ipc_get()

ipc_get() prend pour paramètres un pointeur sur un type particulier d'IPC (i.e. mémoire partagée, sémaphores, ou files de messages) et un identificateur de descripteur, et elle renvoie un pointeur sur le descripteur d'IPC correspondant. Remarquez que bien que les descripteurs de chaque type d'IPC soient de types différents, la structure commune Section 6.4.2.1, « struct kern_ipc_perm » est intégrée comme première entité dans tous les cas. La fonction ipc_get() renvoie ce type de donnée commun. Le modèle attendu consiste en un appel à ipc_get() à travers une fonction enveloppe (i.e. shm_get()) qui force le type de donnée au type de donnée correct du descripteur.

6.4.1.14. ipc_parse_version()

ipc_parse_version() retire le drapeau IPC_64 de la commande s'il est présent et renvoie soit IPC_64 soit IPC_OLD.

6.4.2. Les structures des IPC génériques utilisées avec les sémaphores, les messages, et la mémoire partagée

Les mécanismes de sémaphores, de messages, et de mémoire partagée utilisent tous les structures communes suivantes :

6.4.2.1. struct kern_ipc_perm

Chacun des descripteurs d'IPC possède un objet de ce type comme premier élément. Il permet l'accès à tous les descripteurs depuis toutes les fonctions IPC génériques en utilisant un pointeur de ce type de donnée.

```
/* utilisé par les structures de données internes au noyau */
struct kern_ipc_perm {
    key_t key;
    uid_t uid;
    gid_t gid;
    uid_t cuid;
    gid_t cgid;
    mode_t mode;
    unsigned long seq;
};
```

6.4.2.2. struct ipc_ids

La structure ipc_ids décrit les données communes aux sémaphores, files de messages et à la mémoire partagée. Il y a trois instances globales de cette structure de données --`semid_ds`, `msgid_ds` et `shmid_ds`-- pour les sémaphores, les messages et la mémoire partagée respectivement. Pour chaque instance, le sémaphore `sem` est utilisé pour protéger l'accès à la structure. Le champ `entries` pointe sur un tableau de descripteurs d'IPC, et le verrou tournant `ary` protège l'accès à ce tableau. Le champ `seq` est un numéro de séquence global qui sera incrémenté quand une nouvelle ressource IPC sera créée.

```
struct ipc_ids {
```

```
    int size;
    int in_use;
    int max_id;
    unsigned short seq;
    unsigned short seq_max;
    struct semaphore sem;
    spinlock_t ary;
    struct ipc_id* entries;
};
```

6.4.2.3. struct ipc_id

Il existe un tableau de structures ipc_id dans chaque instance de la structure Section 6.4.2.2, « struct ipc_ids ». Ce tableau est alloué dynamiquement et peut être remplacé par un tableau plus grand Section 6.4.1.6, « grow_ary() » si nécessaire. Le tableau est quelquefois référencé comme tableau de descripteurs, tant que le type de donnée Section 6.4.2.1, « struct kern_ipc_perm » est utilisé comme descripteur de donnée commun par les fonctions IPC génériques.

```
struct ipc_id {
    struct kern_ipc_perm* p;
};
```