Guide pratique d'analyse du noyau (version française du *KernelAnalysis HOWTO*)

Roberto Arcomano berto_CHEZ_bertolinux_POINT_com

Version : 0.7.fr.1.0 du 28 juin 2004

Ce document essaye d'expliquer quelques choses au sujet du noyau de Linux, tel que les composants les plus importants, comment ils fonctionnent, etc... Ce guide pratique devrait aider le lecteur à éviter de devoir passer en revue tous les fichiers source du noyau pour rechercher la "bonne fonction", déclaration, et définition, et puis les lier les unes les autres. Vous pouvez trouver la dernière version originale de ce document à http://www.bertolinux.com. Si vous avez des suggestions pour améliorer ce document, veuillez soumettre vos idées en anglais à mon adresse: berto CHEZ bertolinux POINT com. La dernière version française: http://www.traduc.org. Les corrections de la version française sont les bienvenues, veuillez en informer l'auteur sur son site ou sur http://www.traduc.org.

---

1. Introduction

1.1 Version française

La version française de ce guide pratique a été réalisée le 28 juin 2004 par Patrick Trauquesegue.

1.2 Introduction

Ce guide pratique essaye de définir comment les parties du Noyau Linux fonctionnent, ce que sont les principales fonctions et structures de données utilisées, et comment la "roue tourne". Vous pouvez trouver la dernière version de ce document à http://www.bertolinux.com. Si vous avez des suggestions pour améliorer ce document, veuillez soumettre vos idées en anglais à mon adresse: berto CHEZ bertolinux POINT com. Le code utilisé dans ce document se rapporte à la version 2.4.x de noyau de Linux, qui est la dernière version stable du noyau au moment d'écrire ce guide pratique.

1.3 Copyright

1.4 Traductions

Si vous voulez traduire ce document vous êtes libre de le faire. Cependant, vous aurez besoin de ce qui suit:

1. Vérifiez qu'une autre version du document n'existe pas déjà à votre LDP local
2. Maintenez toutes les sections 'Introduction' (y compris 'Introduction', 'copyright', 'Traductions', 'Remerciements')

Avertissement! Vous ne devez pas traduire de fichier TXT ou HTML, vous devez modifier le dossier de LYX (Ndt: ou SGML), de sorte qu'il soit possible de le convertir dans tous autres formats (TXT, HTML, RIFF (RTF?), etc...): pour le faire vous pouvez utiliser l'application "LyX" à téléchargez sur http://www.lyx.org.

Aucun besoin de me demander de traduire! Vous juste devez me faire connaître (si vous voulez) votre traduction.

Merci de votre traduction!

1.5 Remerciements

Merci à Linux Documentation Project pour publier et télécharger rapidement mon document.

Merci à Klaas de Waal pour ses suggestions.

2. Syntaxe utilisée

2.1 Syntaxe de Fonction

Quand nous parlons d'une fonction, nous écrivons:

```
"nom_de_function  [ fichier emplacement . extension ]"
```

Par exemple:

```
"schedule [kernel/sched.c]"
```

nous dit que nous parlons de

"schedule"

function accessible depuis le fichier

[kernel/sched.c]

Note: Nous supposons /usr/src/linux comme racine du dossier.

2.2 Indentation

L'indentation du code source est de 3 caractères blancs.

2.3 Analyse d'InterCallings

Vue d'ensemble

Nous utilisons l'analyse d'"InterCallings" (ICA) pour voir (de manière indentée) comment les fonctions du noyau s'appellent les unes les autres.

Par exemple, la commande de sleep_on est décrite ainsi dans ICA:

```
|sleep_on
|init_waitqueue_entry      --
|__add_wait_queue            |   enqueuing request
   |list_add                 |
      |__list_add          --
   |schedule              ---     waiting for request to be executed
      |__remove_wait_queue --
      |list_del              |   dequeuing request
         |__list_del       --

                          sleep_on ICA
```

L'ICA indenté est suivi par les fonctions d'emplacement:

- sleep_on [kernel/sched.c]
- init_waitqueue_entry [include/linux/wait.h]
- __add_wait_queue
- list_add [include/linux/list.h]
- __list_add
- schedule [kernel/sched.c]
- __remove_wait_queue [include/linux/wait.h]
- list_del [include/linux/list.h]
- __list_del

Note: Nous n'indiquons plus l'emplacement du dossier, s'il est indiqué juste avant.

Détails

Dans un ICA une telle ligne ressemble à ce qui suit

```
function1 -> function2
```

signifie que < function1 > est un pointeur générique vers une autre fonction. Dans ce cas < function1 > pointe vers < function2 >.

Quand nous écrivons:

```
function:
```

ça signifie que < fonction > n'est pas une vraie fonction. C'est une étiquette/un label (typiquement étiquette/label d'assembleur).

Dans beaucoup de sections nous pouvons reporter un code ''C'' ou un ''pseudo-code''. Dans de vrais fichiers source, vous pourriez utiliser code ''assembleur'' ou ''non structuré de ''. Cette différence est à but d'étude.

POUR l'usage d'ICA

Les avantages d'utiliser ICA (Analyse d'InterCallings) sont nombreux:

- Vous obtenez une vue d'ensemble de ce qui se produit quand vous appelez une fonction du noyau.
- L'emplacement des Fonctions est indiqué après la fonction, ainsi ICA pourrait aussi un peu être appelé "référence des fonctions"
- L'analyse d'InterCallings (ICA) est utile dans des mécanismes de sleep/awake (sommeil/éveillé), où nous pouvons voir ce que nous faisons avant le sommeil, l'action propre au sommeil, et ce que nous feront après réveil (après programme).

CONTRE l'usage d'ICA

- Certains des inconvénients d'utiliser ICA sont énumérés ci-dessous: Some of the disadvantages of using ICA are listed below:

Comme tous les modèles théoriques, nous simplifions la réalité évitant beaucoup de détails, comme le vrai code source et les conditions spéciales.

- Additional diagrams should be added to better represent stack conditions, data values, etc.. Des diagrammes additionnels devraient être ajoutés pour mieux représenter les états de pile, valeurs de données, etc...

3. Fondamentaux

3.1 Qu'est-ce que noyau?

Le noyau est le "coeur" de n'importe quel système informatique: c'est le "logiciel" qui permet aux utilisateurs de partager des ressources de informatique.

On peut penser le noyau comme le logiciel principal du SE (Système d'Exploitation ou OS pour Operating System en anglais), qui peut également inclure la gestion de graphiques.

Par exemple, sous Linux (comme d'autres Unix), l'environnement de XWindow n'appartient pas au noyau, parce qu'il contrôle seulement des opérations graphiques (il utilise l' E/S du Mode Utilisateur pour accéder à la carte vidéo).

Par contre, les environnements de Windows (Win9x, WinME, WinNT, Win2K, WinXP, etc...) sont un mélange entre environnement graphique et noyau.

3.2 Quelle est la différence entre Mode Utilisateur et Mode Noyau?

Vue d’ensemble

Il y a quelques années, quand les ordinateurs étaient aussi grands qu’une pièce, les applications utilisateur tournaient avec beaucoup de difficulté et, parfois, faisaient planter l’ordinateur.

Modes Opérationnel

Pour éviter d’avoir des applications qui plantent constamment, les nouveaux SE ont été conçus avec 2 modes opérationnels:

1. Mode Noyau: la machine fonctionne avec une structure de données critique, matériel direct (entrée/sortie ou mémoire tracée), mémoire directe, IRQ, DMA, etc...
2. Mode Utilisateur: les utilisateurs peuvent faire tourner des applications.

```
               |           Applications           /|\
               |         ______________            |
               |         | User Mode  |            |
               |         ______________            |
               |                |                  |
Implementation |        _______ _______            |   Abstraction
    Detail     |        | Kernel Mode |            |
               |        _______________            |
               |                |                  |
               |                |                  |
               |                |                  |
              \|/          Hardware                |
```

Le Mode Noyau "empêche" les applications de Mode Utilisateur d’endommager le système ou ses dispositifs.

Les microprocesseurs modernes implémentent dans le matériel au moins 2 états différents. Par exemple sous Intel, 4 états déterminent le PL (niveau de privilège). Il est possible d’utiliser 0.1.2.3 états, avec 0 utilisé pour le Mode Noyau.

Le SE Unix requiert seulement 2 niveaux de privilège, et nous utiliserons un tel paradigme comme point de référence.

3.3 Commutation du Mode Utilisateur au Mode Noyau

Quand commutons-nous?

Une fois que nous avons compris qu’il y a 2 modes différents, nous devons savoir quand nous passons de l’un à l’autre.

Typiquement, il y a 2 points de commutation:

1. Quand on appelle un Appel Système: après avoir appelé un Appel Système, la Tâche appelle volontairement des des bouts de code résidant en Mode Noyau.
2. Quand une IRQ (ou exception) arrive: après l'IRQ un gestionnaire d'IRQ (ou gestionnaire d'exception) est appelé, alors le contrôle revient à la Tâche qui a été interrompue comme si rien ne s'était passé.

Appels Système

Les Appels Système sont comme des fonctions spéciales qui gèrent les routines du SE qui résident en Mode Noyau.

Un Appel Système peut être appelé quand nous:

- accédons à une unité d'E/S ou à un fichier (comme lecture ou écriture)
- devons accéder à une information privilégiée (comme le pid, en changeant la politique de programmation (changing scheduling policy) ou une autre information)
- devons changer le contexte d'exécution (comme bifurquer ou exécuter une autre application)
- devons exécuter une commande particulière (comme ''chdir'', ''kill", ''brk'', ou ''signal'')

```
                                     |                |
                             ------->| System Call i  | (Accessing Devices)
|                |       |           |  [sys_read()]  |
| ...            |       |           |                |
| system_call(i) |--------           |                |
|   [read()]     |                   |                |
| ...            |                   |                |
| system_call(j) |--------           |                |
|   [get_pid()]  |       |           |                |
| ...            |       ------->| System Call j  | (Accessing kernel data structures)
|                |                   |  [sys_getpid()]|
                                     |                |

   USER MODE                            KERNEL MODE


                        Unix System Calls Working
```

Les Appels Système sont presque la seule interface utilisée par le Mode Utilisateur pour dialoguer avec des ressources de bas niveau (matériel). La seule exception à cette règle est quand un processus utilise l'Appel Système ''ioperm''. Dans ce cas le processus en Mode Utilisateur peut accéder directement à un périphérique (les IRQs ne peuvent pas être utilisées).

NOTE: Toutes les fonctions ''C'' ne sont pas des Appels Système, seulement certaines d'entre-elles.

Ci-dessous une liste d'Appels Système du noyau 2,4,17 de Linux, de [arch/i386/kernel/entry.S]

```
        .long SYMBOL_NAME(sys_ni_syscall)       /* 0  -  old "setup()" system call*/
        .long SYMBOL_NAME(sys_exit)
        .long SYMBOL_NAME(sys_fork)
        .long SYMBOL_NAME(sys_read)
        .long SYMBOL_NAME(sys_write)
        .long SYMBOL_NAME(sys_open)             /* 5 */
        .long SYMBOL_NAME(sys_close)
        .long SYMBOL_NAME(sys_waitpid)
        .long SYMBOL_NAME(sys_creat)
        .long SYMBOL_NAME(sys_link)
        .long SYMBOL_NAME(sys_unlink)           /* 10 */
        .long SYMBOL_NAME(sys_execve)
        .long SYMBOL_NAME(sys_chdir)
```

```
	.long SYMBOL_NAME(sys_time)
	.long SYMBOL_NAME(sys_mknod)
	.long SYMBOL_NAME(sys_chmod)		/* 15 */
	.long SYMBOL_NAME(sys_lchown16)
	.long SYMBOL_NAME(sys_ni_syscall)				/* old break syscall holder */
	.long SYMBOL_NAME(sys_stat)
	.long SYMBOL_NAME(sys_lseek)
	.long SYMBOL_NAME(sys_getpid)		/* 20 */
	.long SYMBOL_NAME(sys_mount)
	.long SYMBOL_NAME(sys_oldumount)
	.long SYMBOL_NAME(sys_setuid16)
	.long SYMBOL_NAME(sys_getuid16)
	.long SYMBOL_NAME(sys_stime)		/* 25 */
	.long SYMBOL_NAME(sys_ptrace)
	.long SYMBOL_NAME(sys_alarm)
	.long SYMBOL_NAME(sys_fstat)
	.long SYMBOL_NAME(sys_pause)
	.long SYMBOL_NAME(sys_utime)		/* 30 */
	.long SYMBOL_NAME(sys_ni_syscall)				/* old stty syscall holder */
	.long SYMBOL_NAME(sys_ni_syscall)				/* old gtty syscall holder */
	.long SYMBOL_NAME(sys_access)
	.long SYMBOL_NAME(sys_nice)
	.long SYMBOL_NAME(sys_ni_syscall)	/* 35 */		/* old ftime syscall holder */
	.long SYMBOL_NAME(sys_sync)
	.long SYMBOL_NAME(sys_kill)
	.long SYMBOL_NAME(sys_rename)
	.long SYMBOL_NAME(sys_mkdir)
	.long SYMBOL_NAME(sys_rmdir)		/* 40 */
	.long SYMBOL_NAME(sys_dup)
	.long SYMBOL_NAME(sys_pipe)
	.long SYMBOL_NAME(sys_times)
	.long SYMBOL_NAME(sys_ni_syscall)				/* old prof syscall holder */
	.long SYMBOL_NAME(sys_brk)		/* 45 */
	.long SYMBOL_NAME(sys_setgid16)
	.long SYMBOL_NAME(sys_getgid16)
	.long SYMBOL_NAME(sys_signal)
	.long SYMBOL_NAME(sys_geteuid16)
	.long SYMBOL_NAME(sys_getegid16)	/* 50 */
	.long SYMBOL_NAME(sys_acct)
	.long SYMBOL_NAME(sys_umount)					/* recycled never used phys() */
	.long SYMBOL_NAME(sys_ni_syscall)				/* old lock syscall holder */
	.long SYMBOL_NAME(sys_ioctl)
	.long SYMBOL_NAME(sys_fcntl)		/* 55 */
	.long SYMBOL_NAME(sys_ni_syscall)				/* old mpx syscall holder */
	.long SYMBOL_NAME(sys_setpgid)
	.long SYMBOL_NAME(sys_ni_syscall)				/* old ulimit syscall holder */
	.long SYMBOL_NAME(sys_olduname)
	.long SYMBOL_NAME(sys_umask)		/* 60 */
	.long SYMBOL_NAME(sys_chroot)
	.long SYMBOL_NAME(sys_ustat)
	.long SYMBOL_NAME(sys_dup2)
	.long SYMBOL_NAME(sys_getppid)
	.long SYMBOL_NAME(sys_getpgrp)		/* 65 */
	.long SYMBOL_NAME(sys_setsid)
	.long SYMBOL_NAME(sys_sigaction)
	.long SYMBOL_NAME(sys_sgetmask)
	.long SYMBOL_NAME(sys_ssetmask)
	.long SYMBOL_NAME(sys_setreuid16)	/* 70 */
	.long SYMBOL_NAME(sys_setregid16)
	.long SYMBOL_NAME(sys_sigsuspend)
	.long SYMBOL_NAME(sys_sigpending)
	.long SYMBOL_NAME(sys_sethostname)
	.long SYMBOL_NAME(sys_setrlimit)	/* 75 */
	.long SYMBOL_NAME(sys_old_getrlimit)
	.long SYMBOL_NAME(sys_getrusage)
	.long SYMBOL_NAME(sys_gettimeofday)
	.long SYMBOL_NAME(sys_settimeofday)
	.long SYMBOL_NAME(sys_getgroups16)	/* 80 */
	.long SYMBOL_NAME(sys_setgroups16)
	.long SYMBOL_NAME(old_select)
	.long SYMBOL_NAME(sys_symlink)
	.long SYMBOL_NAME(sys_lstat)
	.long SYMBOL_NAME(sys_readlink)		/* 85 */
	.long SYMBOL_NAME(sys_uselib)
	.long SYMBOL_NAME(sys_swapon)
	.long SYMBOL_NAME(sys_reboot)
```

```
.long SYMBOL_NAME(old_readdir)
.long SYMBOL_NAME(old_mmap)            /* 90 */
.long SYMBOL_NAME(sys_munmap)
.long SYMBOL_NAME(sys_truncate)
.long SYMBOL_NAME(sys_ftruncate)
.long SYMBOL_NAME(sys_fchmod)
.long SYMBOL_NAME(sys_fchown16)        /* 95 */
.long SYMBOL_NAME(sys_getpriority)
.long SYMBOL_NAME(sys_setpriority)
.long SYMBOL_NAME(sys_ni_syscall)                                /* old profil syscall holder */
.long SYMBOL_NAME(sys_statfs)
.long SYMBOL_NAME(sys_fstatfs)         /* 100 */
.long SYMBOL_NAME(sys_ioperm)
.long SYMBOL_NAME(sys_socketcall)
.long SYMBOL_NAME(sys_syslog)
.long SYMBOL_NAME(sys_setitimer)
.long SYMBOL_NAME(sys_getitimer)       /* 105 */
.long SYMBOL_NAME(sys_newstat)
.long SYMBOL_NAME(sys_newlstat)
.long SYMBOL_NAME(sys_newfstat)
.long SYMBOL_NAME(sys_uname)
.long SYMBOL_NAME(sys_iopl)            /* 110 */
.long SYMBOL_NAME(sys_vhangup)
.long SYMBOL_NAME(sys_ni_syscall)      /* old "idle" system call */
.long SYMBOL_NAME(sys_vm86old)
.long SYMBOL_NAME(sys_wait4)
.long SYMBOL_NAME(sys_swapoff)         /* 115 */
.long SYMBOL_NAME(sys_sysinfo)
.long SYMBOL_NAME(sys_ipc)
.long SYMBOL_NAME(sys_fsync)
.long SYMBOL_NAME(sys_sigreturn)
.long SYMBOL_NAME(sys_clone)           /* 120 */
.long SYMBOL_NAME(sys_setdomainname)
.long SYMBOL_NAME(sys_newuname)
.long SYMBOL_NAME(sys_modify_ldt)
.long SYMBOL_NAME(sys_adjtimex)
.long SYMBOL_NAME(sys_mprotect)        /* 125 */
.long SYMBOL_NAME(sys_sigprocmask)
.long SYMBOL_NAME(sys_create_module)
.long SYMBOL_NAME(sys_init_module)
.long SYMBOL_NAME(sys_delete_module)
.long SYMBOL_NAME(sys_get_kernel_syms)  /* 130 */
.long SYMBOL_NAME(sys_quotactl)
.long SYMBOL_NAME(sys_getpgid)
.long SYMBOL_NAME(sys_fchdir)
.long SYMBOL_NAME(sys_bdflush)
.long SYMBOL_NAME(sys_sysfs)           /* 135 */
.long SYMBOL_NAME(sys_personality)
.long SYMBOL_NAME(sys_ni_syscall)      /* for afs_syscall */
.long SYMBOL_NAME(sys_setfsuid16)
.long SYMBOL_NAME(sys_setfsgid16)
.long SYMBOL_NAME(sys_llseek)          /* 140 */
.long SYMBOL_NAME(sys_getdents)
.long SYMBOL_NAME(sys_select)
.long SYMBOL_NAME(sys_flock)
.long SYMBOL_NAME(sys_msync)
.long SYMBOL_NAME(sys_readv)           /* 145 */
.long SYMBOL_NAME(sys_writev)
.long SYMBOL_NAME(sys_getsid)
.long SYMBOL_NAME(sys_fdatasync)
.long SYMBOL_NAME(sys_sysctl)
.long SYMBOL_NAME(sys_mlock)           /* 150 */
.long SYMBOL_NAME(sys_munlock)
.long SYMBOL_NAME(sys_mlockall)
.long SYMBOL_NAME(sys_munlockall)
.long SYMBOL_NAME(sys_sched_setparam)
.long SYMBOL_NAME(sys_sched_getparam)   /* 155 */
.long SYMBOL_NAME(sys_sched_setscheduler)
.long SYMBOL_NAME(sys_sched_getscheduler)
.long SYMBOL_NAME(sys_sched_yield)
.long SYMBOL_NAME(sys_sched_get_priority_max)
.long SYMBOL_NAME(sys_sched_get_priority_min)  /* 160 */
.long SYMBOL_NAME(sys_sched_rr_get_interval)
.long SYMBOL_NAME(sys_nanosleep)
.long SYMBOL_NAME(sys_mremap)
.long SYMBOL_NAME(sys_setresuid16)
```

```
.long SYMBOL_NAME(sys_getresuid16)      /* 165 */
.long SYMBOL_NAME(sys_vm86)
.long SYMBOL_NAME(sys_query_module)
.long SYMBOL_NAME(sys_poll)
.long SYMBOL_NAME(sys_nfsservctl)
.long SYMBOL_NAME(sys_setresgid16)      /* 170 */
.long SYMBOL_NAME(sys_getresgid16)
.long SYMBOL_NAME(sys_prctl)
.long SYMBOL_NAME(sys_rt_sigreturn)
.long SYMBOL_NAME(sys_rt_sigaction)
.long SYMBOL_NAME(sys_rt_sigprocmask)   /* 175 */
.long SYMBOL_NAME(sys_rt_sigpending)
.long SYMBOL_NAME(sys_rt_sigtimedwait)
.long SYMBOL_NAME(sys_rt_sigqueueinfo)
.long SYMBOL_NAME(sys_rt_sigsuspend)
.long SYMBOL_NAME(sys_pread)            /* 180 */
.long SYMBOL_NAME(sys_pwrite)
.long SYMBOL_NAME(sys_chown16)
.long SYMBOL_NAME(sys_getcwd)
.long SYMBOL_NAME(sys_capget)
.long SYMBOL_NAME(sys_capset)           /* 185 */
.long SYMBOL_NAME(sys_sigaltstack)
.long SYMBOL_NAME(sys_sendfile)
.long SYMBOL_NAME(sys_ni_syscall)               /* streams1 */
.long SYMBOL_NAME(sys_ni_syscall)               /* streams2 */
.long SYMBOL_NAME(sys_vfork)            /* 190 */
.long SYMBOL_NAME(sys_getrlimit)
.long SYMBOL_NAME(sys_mmap2)
.long SYMBOL_NAME(sys_truncate64)
.long SYMBOL_NAME(sys_ftruncate64)
.long SYMBOL_NAME(sys_stat64)           /* 195 */
.long SYMBOL_NAME(sys_lstat64)
.long SYMBOL_NAME(sys_fstat64)
.long SYMBOL_NAME(sys_lchown)
.long SYMBOL_NAME(sys_getuid)
.long SYMBOL_NAME(sys_getgid)           /* 200 */
.long SYMBOL_NAME(sys_geteuid)
.long SYMBOL_NAME(sys_getegid)
.long SYMBOL_NAME(sys_setreuid)
.long SYMBOL_NAME(sys_setregid)
.long SYMBOL_NAME(sys_getgroups)        /* 205 */
.long SYMBOL_NAME(sys_setgroups)
.long SYMBOL_NAME(sys_fchown)
.long SYMBOL_NAME(sys_setresuid)
.long SYMBOL_NAME(sys_getresuid)
.long SYMBOL_NAME(sys_setresgid)        /* 210 */
.long SYMBOL_NAME(sys_getresgid)
.long SYMBOL_NAME(sys_chown)
.long SYMBOL_NAME(sys_setuid)
.long SYMBOL_NAME(sys_setgid)
.long SYMBOL_NAME(sys_setfsuid)         /* 215 */
.long SYMBOL_NAME(sys_setfsgid)
.long SYMBOL_NAME(sys_pivot_root)
.long SYMBOL_NAME(sys_mincore)
.long SYMBOL_NAME(sys_madvise)
.long SYMBOL_NAME(sys_getdents64)       /* 220 */
.long SYMBOL_NAME(sys_fcntl64)
.long SYMBOL_NAME(sys_ni_syscall)       /* reserved for TUX */
.long SYMBOL_NAME(sys_ni_syscall)       /* Reserved for Security */
.long SYMBOL_NAME(sys_gettid)
.long SYMBOL_NAME(sys_readahead)        /* 225 */
```

Evènement IRQ

Quand arrive une IRQ, la Tâche qui tourne est interrompue pour servir (in order to service) le gestionnaire d'IRQ.

Après que l'IRQ soit gérée, le contrôle retourne exactement au point de l'interruption, comme si rien ne s'était passé.

```
                 Running Task
                |-----------|           (3)
NORMAL          |   |       |  [break execution] IRQ Handler
EXECUTION (1)   |   |       |     ------------->|---------|
                |  \|/      |     |             |  does   |
 IRQ (2)---->   | ..        |----->             |  some   |
                |   |       |<-----             |  work   |
BACK TO         |   |       |     |             |  ..(4). |
NORMAL    (6)   |  \|/      |     <-------------|_________|
EXECUTION       |___________|  [return to code]
                                        (5)
                  USER MODE                      KERNEL MODE

          Transition Mode Utilisateur->Noyau causée par un évènement IRQ
```

Les pas énumérés plus bas se réfèrent à la séquence des évènements dans le diagramme au-dessus/

1. Le processus est exécuté
2. l'IRQ arrive alors que la Tâche tourne.
3. La Tâche est interrompue pour appeler un "gestionnaire d'Interruption".
4. Le code du "gestionnaire d'Interruption" est exécuté.
5. Le contrôle revient à la Tâche en mode utilisateur (comme si rien ne s'était passé)
6. Le processus revient à une exécution normale

Un point spécialement intéressant, l'IRQ Timer, qui se produit à chaque TIMER ms pour gérer:

1. Alarmes
2. Compteurs systèmes and Tâches (utilisé par un programmes pour décider quand arrêter un processus ou pour comptage (for accounting))
3. Le mutiTâche basé sur un mécanisme d'éveil après un temps TIMESLICE.

3.4 Multitache

Mécanisme

Le point central d'un SE moderne est la "Tâche". La Tâche est une application qui tourne en mémoire en partageant toutes ressources (y compris CPU =processeur et mémoire) avec les autres Tâches.

Cette "ressource partagée" est gérée par le "Mécanisme MultiTâche". Le Mécanisme MultiTâche passe d'une Tâche à une autre après un "timeslice" (temps de glissement). Les utilisateurs ont l'"illusion" qu'ils possèdent toutes les ressources. On peut aussi imaginer un scénario simple utilisateur, quand un utilisateur paut avoir l'"illusion" de faire tourner plusieurs Tâches en même temps.

Pour implémenter ce multitâche, la Tâche utilise la variable d'"état" ("the state" variable), qui peut être:

1. READY, prêt pour exécution
2. BLOCKED, en attente d'une ressource

L'état de la Tâche est géré par sa présence dans une liste relative: liste READY et liste BLOCKED.

Changement de Tâche

Le mouvement d'une Tâche à l'autre est appelé ''Task Switching'' (Changement de Tâche). plusieurs ordinateurs ont une instruction matérielle qui réalise automatiquement cette opération. Le Changement de Tâche se produit dans les cas suivants:

1. Après la fin d'un Timeslice: nous devons programmer une Tâche "Prête à exécution" et lui donner accès (schedule a "Ready for execution" task and give it access).
2. Quand une Tâche doit attendre un périphérique: nous devons programmer une nouvelle Tâche et l'y brancher *

* Nous programmons une autre Tâche pour éviter "Busy Form Waiting" (forme occupée en attente), qui se produit quand nous attendons un périphérique au lieu de réaliser un autre travail.

Le Changement de Tâche est géré par l'entité "Schedule" (Programme).

```
Timer     |           |
 IRQ      |           |                           Schedule
  |       |           |                 _________________________
  |----->|   Task 1   |<------------------->|(1)Chooses a Ready Task |
  |       |           |                    |(2)Task Switching        |
  |       |___________|                    |_________________________|
  |       |           |                              /|\
  |       |           |                               |
  |       |           |                               |
  |       |           |                               |
  |       |           |                               |
  |----->|   Task 2   |<-------------------------------|
  |       |           |                               |
  |       |___________|                               |
  .       .     .     .                               .
  .       .     .     .                               .
  .       .     .     .                               .
  |       |           |                               |
  |       |           |                               |
  ------>|   Task N   |<--------------------------------
          |           |
          |___________|

           Changement de Tâche basé sur un glissement de temps (TimeSlice)
```

Un Glissement de temps typique pour linux est environ 10ms.

```
 |           |
 |           | Resource    _____________________________
 |   Task 1  |----------->|(1) Enqueue Resource request |
 |           |   Access   |(2)  Mark Task as blocked    |
 |           |            |(3)  Choose a Ready Task     |
 |___________|            |(4)    Task Switching        |
                          |_____________________________|
                                       |
                                       |
 |           |                         |
 |           |                         |
 |   Task 2  |<-------------------------
```

```
|          |
|          |
|__________|

   Changement de Tâche basé sur l’Attente d’une Ressource
```

3.5 Micronoyau vs SE Monolithique

Vue d’ensemble

Jusqu’à maintenant nous avons vus le SE appelé Monolithique, mais il y a aussi une autre sorte de SE: ’’Micronoyau’’.

Un Micronoyau utilise des Tâches, pas seulement pour les processus en mode utilisateur, mais aussi comme un vrai gestionnaire de noyau, comme Tâche-Disquette, Tâche-DD, Tâche-Réseau etc. Quelques exemples sont Amoeba, et Mach.

POURS et CONTRES le SE Micronoyau

POURS:

- Le SE est plus simple à maintenir parce que chaque Tâche gère un seul type d’opération. Aussi si vous voulez modifier le réseau, vous modifiez Tâche-Réseau (idéalement, si une mise à jour structurelle n’est pas nécessaire).

CONTRES:

- Les performances sont pires que pour un SE Monolithique, parce que vous devez ajouter 2*TASK_SWITCH temps ou temps de changement (le premier pour entrer dans la Tâche spécifique, le second pour en sortir)

Mon opinion personnelle est que, les Micronoyaux sont un bon exemple didactique (comme Minix) mais ils ne sont pas "optimums", aussi ils ne sont pas réellement convenables. Linux utilise quelques Tâches, appelées "Fils Noyau" pour implémenter une petite structure Micronoyau (comme kswapd, qui est utilisé pour récupérer les pages mémoire du stockage de masse). Dans ce cas il n’y a aucun problème de performance parce que le glissement (swapping) est un travail très lent.

3.6 Réseau

Niveaux ISO OSI

Le standard ISO-OSI décrit une architecture réseau avec les niveaux suivants:

1. Niveau physique (exemples: PPP et Ethernet)
2. Niveau lien-donnée (Data-link) (exemples: PPP et Ethernet)
3. Niveau réseau (Network) (exemples: IP, et X.25)
4. Niveau transport (exemples: TCP, UDP)
5. Niveau session (SSL)
6. Niveau présentation (FTP codé binaire-ascii)
7. Niveau application (applications comme Netscape)

Les 2 premiers niveaux lités ci-dessus sont souvent implémentés dans le matériel. Les niveaux suivants sont dans le logiciel (ou firmware (matériel spécifiques?) pour les routeurs).

Plusieurs protocoles sont utilisés par un SE: un d'eux est TCP/IP (le plus important qui réside aux niveaux 3-4).

Que fait le noyau?

Le noyau ne sait rien faire des 2 premiers niveaux de ISO-OSI.

Au RX il:

1. Il gère (handshake) les périphériques de bas niveau (comme carte ethernet ou modem) en recevant des "cadres" (frames) de leur part.
2. Il construit des "paquets" TCP/IP à partir des "cadres" (comme ceux d'Ethernet ou de PPP).
3. Convertit des ''paquets'' dans des ''connexions'' en les passant à la bonne application (grâce au numéro de port) ou
4. Transmet des paquets à la bonne queue

```
(cadres)       (paquets)            (connexions)
frames         packets              sockets
NIC ---------> Kernel ----------> Application
                 (Noyau)
                  |    packets
                  --------------> Forward
                        - RX -       (Transmission)
```

Au stage TX il:

1. Convertit des connexions ou
2. Des données de queues (Queues datas) dans des ''paquets'' TCP/IP
3. Partage des ''packets" en "cadres" (comme ceux d'Ethernet ou de PPP)
4. Envoie des ''cadres'' en utilisant des pilotes HW

```
(connexion)   (paquets)                   (cadres)
sockets       packets                     frames
Application ---------> Kernel ----------> NIC
              packets     /|\
Forward  -------------------
(Transmission)          - TX -
```

3.7 Mémoire virtuelle

Segmentation

La segmentation est la première méthode pour résoudre les problèmes d'allocation mémoire: il vous permet de compiler du code source sans se soucier d'où l'application sera placée dans la mémoire. En fait, cette caractéristique aide les développeurs d'application à développer de façon indépendante du SE et aussi du matériel.

```
|        Stack         |
|          |           |
|         \|/          |
|         Free         |
|         /|\          |      Segment <---> Process
|          |           |
|         Heap         |
| Data uninitialized   |
|  Data initialized    |
|        Code          |
|______________________|

         Segment
```

Nous pouvons dire qu'un segment est l'entité logique d'une application, ou l'image de l'application en mémoire.

Quand on programme, on ne se soucie pas de où nos données sont mises en mémoire, on ne s'intéresse qu'à l'offset à l'intérieur de notre segment (notre application). (the offset inside our segment (our application))

On a l'habitude d'assigner un Segment à chaque processus et vice-versa. Avec Linux ce n'est pas vrai. Linux utilise seulement 4 segments pour chaque Noyau et tous les Processus.

Problème de Segmentation

```
                                         ______________________
                                   ----->|                      |----->
                                   | IN  |     Segment A        | OUT
 _____________________              |     |______________________|
|                     |_____|     |                      |
|      Segment B      |           |     Segment B        |
|                     |_____      |                      |
|_____________________|     |     |______________________|
                            |     |     Segment C        |
                            |     |______________________|
                       ----->|     Segment D        |----->
                         IN  |______________________| OUT

                  Segmentation problem
```

Dans le diagramme au-dessus, nous voulons sortir des processus A et D et entrer dans le processus B. Comme on peut voir il y a assez d'espace pour B, mais nous ne pouvons pas le partager en 2 morceaux, aussi nous NE POUVONS PAS le charger (plus de mémoire).

La raison de ce problème est les purs segments sont des zones continues (parce que ce sont des zones logiques) et ne peuvent pas être partagées.

Pagination

```
 ______________________
|      Page 1          |
|______________________|
|      Page 2          |
```

```
            |______________________|
            |      ..              |      Segment <---> Process
            |______________________|
            |     Page n           |
            |______________________|
            |                      |
            |______________________|
            |                      |
            |______________________|

                  Segment
```

La Pagination partage la mémoire en "n" morceaux, chacun d'eux avec une longueur fixée.

Un processus peut être chargé en une ou plusieurs pages. Quand la mémoire est libérée, toutes les pages sont libérées (voir Problème de Segmentation, avant).

La Pagination est aussi utilisée dans un autre but important, le "Swapping" (glissement). Si une page n'est pas présente dans la mémoire physique, il génère une EXCEPTION, qui poussera le Noyau à chercher une nouvelle page dans la mémoire de stockage. Ce mécanisme permet au SE de charger plus d'applications que celles permises seulement par la mémoire physique.

Problème de Pagination

```
             ______________________
   Page   X |     Process Y        |
            |______________________|
            |                      |
            |       WASTE          |
            |       SPACE          |
            |______________________|

           Problème de Pagination
```

Dans le diagramme qu-dessus, nous pouvons voir ce qui ne va pas dans la politique de pagination: quand un Processus Y se charge dans la Page X, TOUT l'espace mémoire de la Page est alloué, aussi l'espace restant à la fin de la page est perdu.

Segmentation et Pagination

Comment peut-on résoudre les problèmes de segmentation et de pagination? En utilisant les 2 politiques.

```
                                         |      ..              |
                                         |______________________|
                                   ----->|      Page 1          |
                                   |     |______________________|
                                   |     |      ..              |
  ____________________             |     |______________________|
 |                    |            |---->|      Page 2          |
 |      Segment X     |   ---------|     |______________________|
 |                    |            |     |       ..             |
 |____________________|            |     |______________________|
                                   |     |       ..             |
```

```
|      |_____________________|
|---->|       Page 3        |
      |_____________________|
      |         ..          |
```

Le Processus X, identifié par le Segment X, est partagé en 3 morceaux et chacun d’eux est chargé dans une page.

Nous n’avons pas de:

1. Problème de Segmentation: nous allouons par Pages, ainsi nous libérons aussi les Pages, et nous gérons l’espace libre de façon optimisée.
2. Problème de Pagination: seule la dernière page perd de l’espace, mais nous pouvons décider d’utiliser de très petites pages, par exemple 4096 octets de long (perdant au maximum 4096*N_Tâches octets) et gérer une pagination hierarchisée (manage hierarchical paging) (en utilisant 2 ou 3 niveaux de pagination)

```
                         |         |           |         |
                         |         |   Offset2 |  Value  |
                         |         |        /|\|         |
                 Offset1 |         |-----    | |         |
                     /|\ |         |    |    | |         |
                      |  |         |    |   \|/|         |
                      |  |         |    ------>|         |
                     \|/ |         |           |         |
Base Paging Address ---->|         |           |         |
                         | ....... |           | ....... |
                         |         |           |         |

                   Hierarchical Paging
```

4. Démarrage De Linux

Nous démarrons le noyau Linux par du code de C exécuté à partir de l’étiquette asm ’’startup_32:

```
|startup_32:
   |start_kernel
      |lock_kernel
      |trap_init
      |init_IRQ
      |sched_init
      |softirq_init
      |time_init
      |console_init
      |#ifdef CONFIG_MODULES
         |init_modules
      |#endif
      |kmem_cache_init
      |sti
      |calibrate_delay
      |mem_init
      |kmem_cache_sizes_init
      |pgtable_cache_init
      |fork_init
      |proc_caches_init
```

```
|vfs_caches_init
|buffer_init
|page_cache_init
|signals_init
|#ifdef CONFIG_PROC_FS
  |proc_root_init
|#endif
|#if defined(CONFIG_SYSVIPC)
   |ipc_init
|#endif
|check_bugs
|smp_init
|rest_init
   |kernel_thread
   |unlock_kernel
   |cpu_idle
```

- startup_32 [arch/i386/kernel/head.S]
- start_kernel [init/main.c]
- lock_kernel [include/asm/smplock.h]
- trap_init [arch/i386/kernel/traps.c]
- init_IRQ [arch/i386/kernel/i8259.c]
- sched_init [kernel/sched.c]
- softirq_init [kernel/softirq.c]
- time_init [arch/i386/kernel/time.c]
- console_init [drivers/char/tty_io.c]
- init_modules [kernel/module.c]
- kmem_cache_init [mm/slab.c]
- sti [include/asm/system.h]
- calibrate_delay [init/main.c]
- mem_init [arch/i386/mm/init.c]
- kmem_cache_sizes_init [mm/slab.c]
- pgtable_cache_init [arch/i386/mm/init.c]
- fork_init [kernel/fork.c]
- proc_caches_init
- vfs_caches_init [fs/dcache.c]
- buffer_init [fs/buffer.c]
- page_cache_init [mm/filemap.c]
- signals_init [kernel/signal.c]
- proc_root_init [fs/proc/root.c]
- ipc_init [ipc/util.c]
- check_bugs [include/asm/bugs.h]
- smp_init [init/main.c]
- rest_init
- kernel_thread [arch/i386/kernel/process.c]
- unlock_kernel [include/asm/smplock.h]
- cpu_idle [arch/i386/kernel/process.c]

La dernière fonction ''rest_init'' fait ceci:

1. lance le fil ''init'' du noyau
2. appelle unlock_kernel
3. fait tourner la routine de cpu_idle par le noyau, celle sera la boucle à vide s'exécutant quand rien n'est programmé

En fait la procédure start_kernel ne finit jamais. Elle exécutera la routine de cpu_idle sans fin.

Suit la description de ''d'init'', qui est le premier fil du noyau:

```
|init
   |lock_kernel
   |do_basic_setup
      |mtrr_init
      |sysctl_init
      |pci_init
      |sock_init
      |start_context_thread
      |do_init_calls
         |(*call())-> kswapd_init
   |prepare_namespace
   |free_initmem
   |unlock_kernel
   |execve
```

5. Particularités De Linux

5.1 Vue d'ensemble

Linux a quelques particularités qui le distinguent d'autre SE. Ces particularités incluent:

1. Paginations seules
2. Softirq
3. Fils du noyau
4. Modules du noyau
5. Dossier ''Proc''

Eléments De Flexibilité

Les points 4 et 5 donnent une énorme flexibilité aux administrateurs système sur la configuration système du Mode Utilisateur leur permettant de résoudre également des erreurs de noyau critiques ou des problèmes spécifiques sans relancer la machine. Par exemple, si vous devez changer quelque chose sur un gros serveur et vous ne voulez pas faire une réinitialisation, vous pourriez préparer le noyau à dialoguer avec un module, que vous écrirez.

5.2 Paginations seules

Linux n'utilise pas la segmentation pour distinguer les Tâches les unes des autres; il utilise la pagination. (seulement 2 segments sont utilisés pour tous les Tâches, CODE et DATA/stack)

Nous pouvons également dire qu'un défaut de page interTask ne se produit jamais, parce que chaque Tâche utilise un ensemble de Tables de Page différents pour chaque Tâche. Il y a quelques cas où des Tâches différentes pointent vers les mêmes Tables de page, comme les bibliothèques partagées: c'est nécessaire pour réduire l'utilisation de la mémoire; rappelez-vous que les bibliothèques partagées sont seulement du CODE parce que toutes les données sont stockées dans la pile actuelle de la Tâche.

Segments de Linux

Sous le noyau Linux seulement 4 segments existent:

1. Kernel Code [0x10] (Code Noyau)
2. Kernel Data / Stack [0x18] (Données Noyau / Pile)
3. User Code [0x23] (Code Utilisateur)
4. User Data / Stack [0x2b] (Données D'Utilisateur / Pile)

[la syntaxe est ''But [Segment]'']

Sous architecture Intel, les registres de segment utilisés sont:

- CS pour Code Segment
- DS pour Data Segment (Segment Données)
- SS pour Stack Segment (Segment Pile)
- ES pour Alternative Segment (Segment Alternatif) (par exemple utilisé pour faire une copie de mémoire entre 2 segments)

Ainsi, chaque Tâche utilise 0x23 pour le code et le 0x2b pour données/pile

Pagination Linux

Sous Linux 3 niveaux des pages sont utilisés, selon l'architecture. Sous Intel seulement 2 niveaux sont supportés. Linux supporte aussi la Copie sur mécanismes Ecriture (voir Chap.10 pour plus d'information).

Pourquoi les conflits d'adresse interTasks n'existent-ils pas?

La réponse est très très simple: les conflits d'adresse interTask ne peuvent pas exister parce qu'ils sont impossibles. Le traçage linéaire -> physique est fait par "Pagination", ainsi il a juste besoin d'assigner les pages physiques de manière univoque.

Avons-nous besoin de défragmenter la mémoire?

Le numéro d'Assignement de page est un processus dynamique. Nous avons besoin d'une seule page quand une Tâche le demande, ainsi nous la choisissons dans les pages libres de manière ordonnée. Quand nous voulons libérer la page, nous devons seulement l'ajouter à la liste des pages libres.

Et au sujet des pages noyau?

Les pages noyau ont un problème: elles peuvent être assignées de façon dynamique mais nous ne pouvons pas avoir de garantie qu'elles sont dans une zone allouée continue, parce que l'espace linéaire du noyau est équivalent à l'espace physique du noyau.

Pour le Segment Code il n'y a pas de problème. Le code de démarrage est assigné lors du démarrage (aussi nous avons une quantité fixe de mémoire à assigner), et sur les modules nous devons seulement allouer une zone mémoire qui pourrait contenir le code du module.

Le vrai problème est le segment de pile parce que chaque Tâche utilise des pages de pile noyau. Les segments de pile doivent être contigus (selon la définition de pile), ainsi nous devons établir une limite maximum pour la dimension de pile de chaque Tâche. Si nous dépassons cette limite, de mauvaises choses se produisent. Nous recouvrons des structures de données de processus en Mode Noyau.

La structure du noyau nous aide, parce que les fonctions du noyau ne sont jamais:

- recursives
- intercalling plus de N fois.

Une fois que nous connaissons N, et que nous connaissons la moyenne des variables statiques pour toutes les fonctions noyau, nous pouvons estimer une limite de pile.

Si vous voulez tester le problème, vous pouvez créer un module avec une fonction qui s'appelle plusieurs fois. Après un nombre fixé de fois, le module du noyau abandonnera en raison d'une faute du gestionnaire de page d'exception (typiquement écrire à une page lecture-seule).

5.3 Softirq

Quand un IRQ arrive, la commutation de Tâche est reportée à plus tard pour obtenir de meilleures performances. Quelques travaux de Tâche (qui pourrait devoir être fait juste aprés l'IRQ et qui pourraient prendre beaucoup de processeur dans le temps d'interruption, comme accumuler un paquet de TCP/IP) sont mis en queue (alignés) et seront faits au temps programmé (une fois qu'un temps-tranche (time-slice) finit).

Dans les noyaux récents (2.4.x) les mécanismes de softirq sont passés à un kernel_thread (fil de noyau): ''ksoftirqd_CPUn''. n représente le nombre processeur exécutant le kernel_thread (dans système monoprocessor ''ksoftirqd_CPU0'' utilise PID 3).

Préparation De Softirq

Autoriser Softirq

Le ''cpu_raise_softirq'' est une routine qui réveille le fil du noyau ''ksoftirqd_CPU0'', pour le laisser contrôler le travail mis en file d'attente.

```
|cpu_raise_softirq
   |__cpu_raise_softirq
   |wakeup_softirqd
      |wake_up_process
```

- cpu_raise_softirq [kernel/softirq.c]
- __cpu_raise_softirq [include/linux/interrupt.h]
- wakeup_softirq [kernel/softirq.c]
- wake_up_process [kernel/sched.c]

la routine ''__de cpu_raise_softirq__'' placera le bon octet dans le vecteur décrivant le softirq en suspens.

''wakeup_softirq'' utilise le ''wakeup_process'' pour réveiller le fil du noyau ''ksoftirqd_CPU0''.

Exécution De Softirq

A FAIRE: décrire les structures de données impliquées dans le mécanisme de softirq.

Quand le fil noyau ''ksoftirqd_CPU0'' a été réveillé, il exécute les travaux de la file.

Le code de ''ksoftirqd_CPU0'' est (boucle principale sans fin):

```
for (;;) {
   if (!softirq_pending(cpu))
      schedule();
      __set_current_state(TASK_RUNNING);
   while (softirq_pending(cpu)) {
      do_softirq();
      if (current->need_resched)
         schedule
   }
   __set_current_state(TASK_INTERRUPTIBLE)
}
```

- ksoftirqd [kernel/softirq.c]

5.4 Fils Du Kernel

Quoique Linux soit un OS monolithique, quelques ''fils du noyau'' existent pour faire le ménage.

Ces Tâches n'utilisent pas la mémoire Utilisateur; elles partagent la mémoire NOYAU. Elles opèrent également avec le privilège le plus élevé (ANNEAU 0 sur une architecture i386) comme tout autre bout de code Mode Noyau.

Des fils du noyau sont créés par la fonction ''kernel_thread [arch/i386/kernel/process]'', qui appelle un appel système ''clone'' [arch/i386/kernel/process.c] de l'assembleur (qui est ''une fourche'' comme l'appel système):

```
int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
{
        long retval, d0;

        __asm__ __volatile__(
                "movl %%esp,%%esi\n\t"
                "int $0x80\n\t"         /* Linux/i386 system call */
                "cmpl %%esp,%%esi\n\t"  /* child or parent? */
                "je 1f\n\t"             /* parent - jump */
                /* Load the argument into eax, and push it.  That way, it does
                 * not matter whether the called function is compiled with
                 * -mregparm or not.  */
                "movl %4,%%eax\n\t"
                "pushl %%eax\n\t"
                "call *%5\n\t"          /* call fn */
                "movl %3,%0\n\t"        /* exit */
                "int $0x80\n"
                "1:\t"
```

```
                :"=&a" (retval), "=&S" (d0)
                :"0" (__NR_clone), "i" (__NR_exit),
                 "r" (arg), "r" (fn),
                 "b" (flags | CLONE_VM)
                : "memory");
        return retval;
}
```

Une fois appelée, nous avons une nouvelle Tâche (habituellement avec un nombre très bas de PID, comme 2.3, etc...) attendant une ressource très lente, comme un événement échange ou usb. Une ressource très lente est utilisée parce qu'autrement nous aurions une commutation de Tâche coûteuse (overhead).

Au-dessous une liste de la plupart des fils du noyau courants (avec ''ps x''):

```
PID      COMMAND
 1        init
 2        keventd
 3        kswapd
 4        kreclaimd
 5        bdflush
 6        kupdated
 7        kacpid
67        khubd
```

Le fil noyau ''init'' est le premier processus créé, au démarrage. Il appellera toutes autres Tâches Mode Utilisateur (à partir du fichier /etc/inittab) comme les démons console, les démons tty et les démons réseau (''rc'' scripts).

Exemple de fils noyau: kswapd [mm/vmscan.c].

''kswapd'' est créé par ''clone() [arch/i386/kernel/process.c]''

Routines d'initialisation:

```
|do_initcalls
   |kswapd_init
      |kernel_thread
         |syscall fork (in assembler)
```

do_initcalls [init/main.c]

kswapd_init [mm/vmscan.c]

kernel_thread [arch/i386/kernel/process.c]

5.5 Modules Du Kernel

Vue d'ensemble

Les modules du noyau Linux sont des bouts de code (exemples: fs, net, et pilotes hw) fonctionnant en Mode Noyau que vous pouvez ajouter à chaud.

Le coeur de Linux ne peut pas être modularisé: programmation et gestion d'interruption ou noyau du réseau, etc...

Sous "/lib/modules/KERNEL_VERSION/" vous pouvez trouver tous les modules installés sur votre système.

Chargement et déchargement de module

Pour charger un module, tapez ce qui suit:

```
insmod MODULE_NAME parameters
```

```
exemple: insmod ne io=0x300 irq=9
```

NOTE: Vous pouvez utiliser modprobe à la place d'insmod si vous voulez que le noyau recherche automatiquement certains paramètres (par exemple en utilisant le pilote de PCI, ou si vous avez indiqué un paramètre dans le fichier /etc/conf.modules).

Pour décharger un module, tapez ce qui suit:

```
 rmmod MODULE_NAME
```

Définition de module

Un module contient toujours:

1. fonction "init_module", exécutée à la commande d'insmod (ou modprobe)
2. fonction "cleanup_module", exécutée à la commande de rmmod

Si ces fonctions ne sont pas dans le module, vous devez ajouter 2 macros pour indiquer quelles fonctions agiront en tant qu'init et sortie du module:

1. module_init(FUNCTION_NAME)
2. module_exit(FUNCTION_NAME)

NOTE: un module peut "voir" une variable du noyau seulement si elle a été exportée (avec la macro EXPORT_symbol).

Un truc utile pour ajouter de la flexibilité à votre noyau

```
// kernel sources side
void (*foo_function_pointer)(void *);

if (foo_function_pointer)
  (foo_function_pointer)(parameter);




// module side
extern void (*foo_function_pointer)(void *);

void my_function(void *parameter) {
  //My code
}
```

```
int init_module() {
  foo_function_pointer = &my_function;
}

int cleanup_module() {
  foo_function_pointer = NULL;
}
```

Cette simple astuce vous permet d'avoir une flexibilité très élevée dans le noyau, parce que ne ferez exécuter la routine "ma_fonction" que quand vous chargerez le module. Cette routine fera tout ce que vous voulez faire: par exemple le module ''rshaper'', qui commande le trafic d'entrée de bande passante du réseau, fonctionne dans ce cas de figure.

Notez que le mécanisme entier de module est possible grâce à quelques variables globales exportées vers les modules, telles qu'une liste entête (vous permettant d'étendre la liste autant que vous voulez). Les exemples typiques sont fs, pilotes génériques (caractère, bloc, réseau, téléphonie). Vous devez préparer le noyau à accepter votre nouveau module; dans certains cas vous devez créer une infrastructure (comme celle de la téléphonie, créée récemment) pour être aussi standard que possible.

5.6 Le dossier Proc

Proc fs (système de fichiers) est situé dans le dossier /proc, qui est un dossier spécial vous permettant de dialoguer directement avec le noyau.

Linux utilise le dossier ''proc'' pour supporter les communications directes du noyau: c'est nécessaire dans plusieurs de cas, par exemple quand vous voulez voir les structures de données de processus principaux ou autoriser le dispositif ''proxy-arp'' pour une interface et pas pour d'autres, vous voulez changer le nombre maximum de fils, ou si vous voulez corriger certains états de bus, comme ISA ou PCI, pour savoir quelles cartes sont installées et quelles adresses E/S et IRQs leur sont assignées.

```
|-- bus
|   |-- pci
|   |   |-- 00
|   |   |   |-- 00.0
|   |   |   |-- 01.0
|   |   |   |-- 07.0
|   |   |   |-- 07.1
|   |   |   |-- 07.2
|   |   |   |-- 07.3
|   |   |   |-- 07.4
|   |   |   |-- 07.5
|   |   |   |-- 09.0
|   |   |   |-- 0a.0
|   |   |   `-- 0f.0
|   |   |-- 01
|   |   |   `-- 00.0
|   |   `-- devices
|   `-- usb
|-- cmdline
|-- cpuinfo
|-- devices
|-- dma
|-- dri
|   `-- 0
|       |-- bufs
|       |-- clients
```

```
|       |-- mem
|       |-- name
|       |-- queues
|       |-- vm
|       `-- vma
|-- driver
|-- execdomains
|-- filesystems
|-- fs
|-- ide
|   |-- drivers
|   |-- hda -> ide0/hda
|   |-- hdc -> ide1/hdc
|   |-- ide0
|   |   |-- channel
|   |   |-- config
|   |   |-- hda
|   |   |   |-- cache
|   |   |   |-- capacity
|   |   |   |-- driver
|   |   |   |-- geometry
|   |   |   |-- identify
|   |   |   |-- media
|   |   |   |-- model
|   |   |   |-- settings
|   |   |   |-- smart_thresholds
|   |   |   `-- smart_values
|   |   |-- mate
|   |   `-- model
|   |-- ide1
|   |   |-- channel
|   |   |-- config
|   |   |-- hdc
|   |   |   |-- capacity
|   |   |   |-- driver
|   |   |   |-- identify
|   |   |   |-- media
|   |   |   |-- model
|   |   |   `-- settings
|   |   |-- mate
|   |   `-- model
|   `-- via
|-- interrupts
|-- iomem
|-- ioports
|-- irq
|   |-- 0
|   |-- 1
|   |-- 10
|   |-- 11
|   |-- 12
|   |-- 13
|   |-- 14
|   |-- 15
|   |-- 2
|   |-- 3
|   |-- 4
|   |-- 5
|   |-- 6
|   |-- 7
|   |-- 8
|   |-- 9
```

```
|   `-- prof_cpu_mask
|-- kcore
|-- kmsg
|-- ksyms
|-- loadavg
|-- locks
|-- meminfo
|-- misc
|-- modules
|-- mounts
|-- mtrr
|-- net
|   |-- arp
|   |-- dev
|   |-- dev_mcast
|   |-- ip_fwchains
|   |-- ip_fwnames
|   |-- ip_masquerade
|   |-- netlink
|   |-- netstat
|   |-- packet
|   |-- psched
|   |-- raw
|   |-- route
|   |-- rt_acct
|   |-- rt_cache
|   |-- rt_cache_stat
|   |-- snmp
|   |-- sockstat
|   |-- softnet_stat
|   |-- tcp
|   |-- udp
|   |-- unix
|   `-- wireless
|-- partitions
|-- pci
|-- scsi
|   |-- ide-scsi
|   |   `-- 0
|   `-- scsi
|-- self -> 2069
|-- slabinfo
|-- stat
|-- swaps
|-- sys
|   |-- abi
|   |   |-- defhandler_coff
|   |   |-- defhandler_elf
|   |   |-- defhandler_lcall7
|   |   |-- defhandler_libcso
|   |   |-- fake_utsname
|   |   `-- trace
|   |-- debug
|   |-- dev
|   |   |-- cdrom
|   |   |   |-- autoclose
|   |   |   |-- autoeject
|   |   |   |-- check_media
|   |   |   |-- debug
|   |   |   |-- info
|   |   |   `-- lock
|   |   `-- parport
```

```
|   |       |-- default
|   |       |   |-- spintime
|   |       |   `-- timeslice
|   |       `-- parport0
|   |           |-- autoprobe
|   |           |-- autoprobe0
|   |           |-- autoprobe1
|   |           |-- autoprobe2
|   |           |-- autoprobe3
|   |           |-- base-addr
|   |           |-- devices
|   |           |   |-- active
|   |           |   `-- lp
|   |           |       `-- timeslice
|   |           |-- dma
|   |           |-- irq
|   |           |-- modes
|   |           `-- spintime
|   |-- fs
|   |   |-- binfmt_misc
|   |   |-- dentry-state
|   |   |-- dir-notify-enable
|   |   |-- dquot-nr
|   |   |-- file-max
|   |   |-- file-nr
|   |   |-- inode-nr
|   |   |-- inode-state
|   |   |-- jbd-debug
|   |   |-- lease-break-time
|   |   |-- leases-enable
|   |   |-- overflowgid
|   |   `-- overflowuid
|   |-- kernel
|   |   |-- acct
|   |   |-- cad_pid
|   |   |-- cap-bound
|   |   |-- core_uses_pid
|   |   |-- ctrl-alt-del
|   |   |-- domainname
|   |   |-- hostname
|   |   |-- modprobe
|   |   |-- msgmax
|   |   |-- msgmnb
|   |   |-- msgmni
|   |   |-- osrelease
|   |   |-- ostype
|   |   |-- overflowgid
|   |   |-- overflowuid
|   |   |-- panic
|   |   |-- printk
|   |   |-- random
|   |   |   |-- boot_id
|   |   |   |-- entropy_avail
|   |   |   |-- poolsize
|   |   |   |-- read_wakeup_threshold
|   |   |   |-- uuid
|   |   |   `-- write_wakeup_threshold
|   |   |-- rtsig-max
|   |   |-- rtsig-nr
|   |   |-- sem
|   |   |-- shmall
|   |   |-- shmmax
```

```
|   |   |-- shmmni
|   |   |-- sysrq
|   |   |-- tainted
|   |   |-- threads-max
|   |   `-- version
|   |-- net
|   |   |-- 802
|   |   |-- core
|   |   |   |-- hot_list_length
|   |   |   |-- lo_cong
|   |   |   |-- message_burst
|   |   |   |-- message_cost
|   |   |   |-- mod_cong
|   |   |   |-- netdev_max_backlog
|   |   |   |-- no_cong
|   |   |   |-- no_cong_thresh
|   |   |   |-- optmem_max
|   |   |   |-- rmem_default
|   |   |   |-- rmem_max
|   |   |   |-- wmem_default
|   |   |   `-- wmem_max
|   |   |-- ethernet
|   |   |-- ipv4
|   |   |   |-- conf
|   |   |   |   |-- all
|   |   |   |   |   |-- accept_redirects
|   |   |   |   |   |-- accept_source_route
|   |   |   |   |   |-- arp_filter
|   |   |   |   |   |-- bootp_relay
|   |   |   |   |   |-- forwarding
|   |   |   |   |   |-- log_martians
|   |   |   |   |   |-- mc_forwarding
|   |   |   |   |   |-- proxy_arp
|   |   |   |   |   |-- rp_filter
|   |   |   |   |   |-- secure_redirects
|   |   |   |   |   |-- send_redirects
|   |   |   |   |   |-- shared_media
|   |   |   |   |   `-- tag
|   |   |   |   |-- default
|   |   |   |   |   |-- accept_redirects
|   |   |   |   |   |-- accept_source_route
|   |   |   |   |   |-- arp_filter
|   |   |   |   |   |-- bootp_relay
|   |   |   |   |   |-- forwarding
|   |   |   |   |   |-- log_martians
|   |   |   |   |   |-- mc_forwarding
|   |   |   |   |   |-- proxy_arp
|   |   |   |   |   |-- rp_filter
|   |   |   |   |   |-- secure_redirects
|   |   |   |   |   |-- send_redirects
|   |   |   |   |   |-- shared_media
|   |   |   |   |   `-- tag
|   |   |   |   |-- eth0
|   |   |   |   |   |-- accept_redirects
|   |   |   |   |   |-- accept_source_route
|   |   |   |   |   |-- arp_filter
|   |   |   |   |   |-- bootp_relay
|   |   |   |   |   |-- forwarding
|   |   |   |   |   |-- log_martians
|   |   |   |   |   |-- mc_forwarding
|   |   |   |   |   |-- proxy_arp
|   |   |   |   |   |-- rp_filter
```

```
|   |   |   |   |   |-- secure_redirects
|   |   |   |   |   |-- send_redirects
|   |   |   |   |   |-- shared_media
|   |   |   |   |   `-- tag
|   |   |   |   |-- eth1
|   |   |   |   |   |-- accept_redirects
|   |   |   |   |   |-- accept_source_route
|   |   |   |   |   |-- arp_filter
|   |   |   |   |   |-- bootp_relay
|   |   |   |   |   |-- forwarding
|   |   |   |   |   |-- log_martians
|   |   |   |   |   |-- mc_forwarding
|   |   |   |   |   |-- proxy_arp
|   |   |   |   |   |-- rp_filter
|   |   |   |   |   |-- secure_redirects
|   |   |   |   |   |-- send_redirects
|   |   |   |   |   |-- shared_media
|   |   |   |   |   `-- tag
|   |   |   |   `-- lo
|   |   |   |       |-- accept_redirects
|   |   |   |       |-- accept_source_route
|   |   |   |       |-- arp_filter
|   |   |   |       |-- bootp_relay
|   |   |   |       |-- forwarding
|   |   |   |       |-- log_martians
|   |   |   |       |-- mc_forwarding
|   |   |   |       |-- proxy_arp
|   |   |   |       |-- rp_filter
|   |   |   |       |-- secure_redirects
|   |   |   |       |-- send_redirects
|   |   |   |       |-- shared_media
|   |   |   |       `-- tag
|   |   |   |-- icmp_echo_ignore_all
|   |   |   |-- icmp_echo_ignore_broadcasts
|   |   |   |-- icmp_ignore_bogus_error_responses
|   |   |   |-- icmp_ratelimit
|   |   |   |-- icmp_ratemask
|   |   |   |-- inet_peer_gc_maxtime
|   |   |   |-- inet_peer_gc_mintime
|   |   |   |-- inet_peer_maxttl
|   |   |   |-- inet_peer_minttl
|   |   |   |-- inet_peer_threshold
|   |   |   |-- ip_autoconfig
|   |   |   |-- ip_conntrack_max
|   |   |   |-- ip_default_ttl
|   |   |   |-- ip_dynaddr
|   |   |   |-- ip_forward
|   |   |   |-- ip_local_port_range
|   |   |   |-- ip_no_pmtu_disc
|   |   |   |-- ip_nonlocal_bind
|   |   |   |-- ipfrag_high_thresh
|   |   |   |-- ipfrag_low_thresh
|   |   |   |-- ipfrag_time
|   |   |   |-- neigh
|   |   |   |   |-- default
|   |   |   |   |   |-- anycast_delay
|   |   |   |   |   |-- app_solicit
|   |   |   |   |   |-- base_reachable_time
|   |   |   |   |   |-- delay_first_probe_time
|   |   |   |   |   |-- gc_interval
|   |   |   |   |   |-- gc_stale_time
|   |   |   |   |   |-- gc_thresh1
```

```
|   |   |   |   |   |-- gc_thresh2
|   |   |   |   |   |-- gc_thresh3
|   |   |   |   |   |-- locktime
|   |   |   |   |   |-- mcast_solicit
|   |   |   |   |   |-- proxy_delay
|   |   |   |   |   |-- proxy_qlen
|   |   |   |   |   |-- retrans_time
|   |   |   |   |   |-- ucast_solicit
|   |   |   |   |   `-- unres_qlen
|   |   |   |   |-- eth0
|   |   |   |   |   |-- anycast_delay
|   |   |   |   |   |-- app_solicit
|   |   |   |   |   |-- base_reachable_time
|   |   |   |   |   |-- delay_first_probe_time
|   |   |   |   |   |-- gc_stale_time
|   |   |   |   |   |-- locktime
|   |   |   |   |   |-- mcast_solicit
|   |   |   |   |   |-- proxy_delay
|   |   |   |   |   |-- proxy_qlen
|   |   |   |   |   |-- retrans_time
|   |   |   |   |   |-- ucast_solicit
|   |   |   |   |   `-- unres_qlen
|   |   |   |   |-- eth1
|   |   |   |   |   |-- anycast_delay
|   |   |   |   |   |-- app_solicit
|   |   |   |   |   |-- base_reachable_time
|   |   |   |   |   |-- delay_first_probe_time
|   |   |   |   |   |-- gc_stale_time
|   |   |   |   |   |-- locktime
|   |   |   |   |   |-- mcast_solicit
|   |   |   |   |   |-- proxy_delay
|   |   |   |   |   |-- proxy_qlen
|   |   |   |   |   |-- retrans_time
|   |   |   |   |   |-- ucast_solicit
|   |   |   |   |   `-- unres_qlen
|   |   |   |   `-- lo
|   |   |   |       |-- anycast_delay
|   |   |   |       |-- app_solicit
|   |   |   |       |-- base_reachable_time
|   |   |   |       |-- delay_first_probe_time
|   |   |   |       |-- gc_stale_time
|   |   |   |       |-- locktime
|   |   |   |       |-- mcast_solicit
|   |   |   |       |-- proxy_delay
|   |   |   |       |-- proxy_qlen
|   |   |   |       |-- retrans_time
|   |   |   |       |-- ucast_solicit
|   |   |   |       `-- unres_qlen
|   |   |   |-- route
|   |   |   |   |-- error_burst
|   |   |   |   |-- error_cost
|   |   |   |   |-- flush
|   |   |   |   |-- gc_elasticity
|   |   |   |   |-- gc_interval
|   |   |   |   |-- gc_min_interval
|   |   |   |   |-- gc_thresh
|   |   |   |   |-- gc_timeout
|   |   |   |   |-- max_delay
|   |   |   |   |-- max_size
|   |   |   |   |-- min_adv_mss
|   |   |   |   |-- min_delay
|   |   |   |   |-- min_pmtu
```

```
|   |   |   |   |-- mtu_expires
|   |   |   |   |-- redirect_load
|   |   |   |   |-- redirect_number
|   |   |   |   `-- redirect_silence
|   |   |   |-- tcp_abort_on_overflow
|   |   |   |-- tcp_adv_win_scale
|   |   |   |-- tcp_app_win
|   |   |   |-- tcp_dsack
|   |   |   |-- tcp_ecn
|   |   |   |-- tcp_fack
|   |   |   |-- tcp_fin_timeout
|   |   |   |-- tcp_keepalive_intvl
|   |   |   |-- tcp_keepalive_probes
|   |   |   |-- tcp_keepalive_time
|   |   |   |-- tcp_max_orphans
|   |   |   |-- tcp_max_syn_backlog
|   |   |   |-- tcp_max_tw_buckets
|   |   |   |-- tcp_mem
|   |   |   |-- tcp_orphan_retries
|   |   |   |-- tcp_reordering
|   |   |   |-- tcp_retrans_collapse
|   |   |   |-- tcp_retries1
|   |   |   |-- tcp_retries2
|   |   |   |-- tcp_rfc1337
|   |   |   |-- tcp_rmem
|   |   |   |-- tcp_sack
|   |   |   |-- tcp_stdurg
|   |   |   |-- tcp_syn_retries
|   |   |   |-- tcp_synack_retries
|   |   |   |-- tcp_syncookies
|   |   |   |-- tcp_timestamps
|   |   |   |-- tcp_tw_recycle
|   |   |   |-- tcp_window_scaling
|   |   |   `-- tcp_wmem
|   |   `-- unix
|   |       `-- max_dgram_qlen
|   |-- proc
|   `-- vm
|       |-- bdflush
|       |-- kswapd
|       |-- max-readahead
|       |-- min-readahead
|       |-- overcommit_memory
|       |-- page-cluster
|       `-- pagetable_cache
|-- sysvipc
|   |-- msg
|   |-- sem
|   `-- shm
|-- tty
|   |-- driver
|   |   `-- serial
|   |-- drivers
|   |-- ldisc
|   `-- ldiscs
|-- uptime
`-- version
```

Dans ce dossier il y a aussi les Tâches qui utilisent le PID comme nom de fichier (vous avez accès à toutes les informations sur les Tâches, comme le chemin du fichier binaire, la mémoire utilisée, etc...).

Ce qui est intéressant c'est que non seulement vous pouvez voir les valeurs du noyau (par exemple, voir l'information sur toute Tâche ou sur les options actives du réseau de votre pile TCP/IP) mais vous pouvez aussi en modifier certaines, typiquement ceux qui sont dans le dossier /proc/sys:

```
/proc/sys/
          acpi
          dev
          debug
          fs
          proc
          net
          vm
          kernel
```

/proc/sys/kernel

Suivent des valeurs du noyau très importantes et bien connues, prêtes à être modifiées:

```
overflowgid
overflowuid
random
threads-max // Nombre maximum de fils, typiquement 16384
sysrq // hachage noyau: vous pouvez voir les valeurs du registre istant (istant) et plus
sem
msgmnb
msgmni
msgmax
shmmni
shmall
shmmax
rtsig-max
rtsig-nr
modprobe // emplacement fichier modprobe
printk
ctrl-alt-del
cap-bound
panic
domainname // nom de domaine de votre poste Linux
hostname // nom d'hôte de votre poste Linux
version // informations sur la date de compilation de votre noyau
osrelease // version du noyau (i.e. 2.4.5)
ostype // Linux!
```

/proc/sys/net

Il peut être considéré comme le sous-dossier le plus utile de proc. Il vous parmet de changer des paramètres très importants de la configuration réseau de votre noyau.

```
core
ipv4
ipv6
unix
ethernet
802
```

/proc/sys/net/core

La liste suivante est la configuration générale du réseau, comme netdev_max_backlog" (typiquement 300), la longueur de tous les paquets réseau. Cette valeur peut limiter la bande passante du réseau quand il reçoit des paquets. Linux doit attendre pour programmer le temps de vider les tampons (du aux mécanismes de la moitié d'en bas), envirn 1000/HZ ms (Listed below are general net settings, like "netdev_max_backlog" (typically 300), the length of all your network packets. This value can limit your network bandwidth when receiving packets, Linux has to wait up to scheduling time to flush buffers (due to bottom half mechanism), about 1000/HZ ms)

```
  300    *        100             =     30 000
packets     HZ(Timeslice freq)         packets/s

30 000   *       1000             =      30 M
packets     average (Bytes/packet)   throughput Bytes/s
```

Si vous voulez un plus gros débit, vous pouvez augmenter netdev_max_backlog,en tapant:

```
echo 4000 > /proc/sys/net/core/netdev_max_backlog
```

Note: Attention à certaines valeurs HZ: sous certaines architectures (comme alpha ou arm-tbox) c'est 1000, aussi vous pouvez avoir un débit moyen de 300 Mo/s.

/proc/sys/net/ipv4

"ip_forward", active ou désactive la réexpédition ou transmission ip (ip forwarding) dans votre poste Linux. C'est une configuration générique pour tous les périphériques, que vous pouvez spécifier pour chaque périphérique que vous choisissez.

/proc/sys/net/ipv4/conf/interface

Je pense que c'est l'entrée de /proc la plus utile, parce qu'elle permet de modifier la configuration réseau pour supporter les réseaux sans fils (voir Wireless-HOWTO pour plus d'informations, ou le guide pratique officiel Wireless-HOWTO).

Quelques exemples où vous pouvez utiliser ces réglages:

- "forwarding", pour activer la réexpédition ip pour votre interface
- "proxy_arp", pour activer les éléments arp interposés (proxy arp feature). Pour en savoir plus voir Proxy arp HOWTO sur Linux Documentation Project et Wireless-HOWTO ou le guide pratique officiel, pour les Proxy arp qui utilisent les réseaux sans fils.
- "send_redirects" pour éviter que l'interface envoie ICMP_REDIRECT (voir comme avant Wireless-HOWTO ou le guide pratique officiel pour en savoir plus).

6. Traitement MultiTâche Linux

6.1 Vue d’ensemble

Cette section va analyser les structures de données -- le mécanisme utilisé pour contrôler l’environnement de traitement multitâche sous Linux.

États de Tâche

Une Tâche Linux peut avoir (can be) un des états suivants (selon [include/linux.h]):

1. TASK_RUNNING, signifie qu’elle est dans "la liste prête" ("Ready List")
2. TASK_INTERRUPTIBLE, Tâche attendant un signal ou une ressource (sommeil)
3. TASK_UNINTERRUPTIBLE, Tâche attendant une ressource (sommeil), elle est dans la même "file d’attente d’attente"
4. TASK_ZOMBIE, enfant de Tâche sans père
5. TASK_STOPPED, Tâche à corriger (task being debugged)

Interaction Graphique

```
      _______________     CPU Disponible     _______________
     |               |  ----------------->  |               |
     | TASK_RUNNING  |                      | Real Running  |
     |_______________|  <-----------------  |_______________|
                             CPU Occupée
           |   /|\
 Attend     |     | Ressource
Ressource   |     | Disponible
          \|/    |
   ______________________
  |                      |
  | TASK_INTERRUPTIBLE / |
  | TASK-UNINTERRUPTIBLE |
  |______________________|

                    Flot Multitâche Principal
```

6.2 Timeslice (glissement de temps)

Programmation PIT 8253

Toutes les 10ms (selon la valeur de HZ) un IRQ0 se produit, qui nous aide en environnement multitâche. Ce signal vient de PIC 8259 (en arch 386+) qui est relié à PIT 8253 avec une horloge de 1,19318 mégahertz.

```
    _____         ______         ______
   | CPU |<------| 8259 |------| 8253 |
   |_____| IRQ0  |______|      |___/|\|
                                    |_____ CLK 1.193.180 MHz

// From include/asm/param.h
#ifndef HZ
#define HZ 100
#endif

// From include/asm/timex.h
```

```
#define CLOCK_TICK_RATE 1193180 /* Underlying HZ */

// From include/linux/timex.h
#define LATCH ((CLOCK_TICK_RATE + HZ/2) / HZ) /* For divider */

// From arch/i386/kernel/i8259.c
outb_p(0x34,0x43); /* binary, mode 2, LSB/MSB, ch 0 */
outb_p(LATCH & 0xff , 0x40); /* LSB */
outb(LATCH >> 8 , 0x40); /* MSB */
```

Ainsi nous programmons 8253 (PIT, Programmable Interval Timer ou Compteur à intervalle programmable) avec LATCH (VERROU) = (1193180/hz) = 11931,8 quand HZ=100 (défaut). LATCH indique le facteur divisant la fréquence.

LATCH = 11931,8 donne à 8253 (en sortie) une fréquence de 1193180/11931,8 = 100 Hz, ainsi la période = 10ms

Ainsi Timeslice = 1/hz.

Avec chaque Timeslice nous interrompons temporairement l'exécution du processus courant (sans commutation de Tâche), et nous faisons du ménage, après quoi nous retournerons de nouveau à notre processus précédent.

Linux Timer (Compteur) IRQ ICA

```
Linux Timer IRQ
IRQ 0 [Timer]
 |
\|/
|IRQ0x00_interrupt        //   wrapper IRQ handler
   |SAVE_ALL              ---
      |do_IRQ                |   wrapper routines
         |handle_IRQ_event  ---
            |handler() -> timer_interrupt  // registered IRQ 0 handler
               |do_timer_interrupt
                  |do_timer
                     |jiffies++;
                     |update_process_times
                     |if (--counter <= 0) { // if time slice ended then
                        |counter = 0;        //   reset counter
                        |need_resched = 1;   //   prepare to reschedule
                     |}
         |do_softirq
         |while (need_resched) { // if necessary
            |schedule             //   reschedule
            |handle_softirq
         |}
   |RESTORE_ALL
```

Des fonctions peuvent être trouvées sous:

- IRQ0x00_interrupt, SAVE_ALL [include/asm/hw_irq.h]
- do_IRQ, handle_IRQ_event [arch/i386/kernel/irq.c]
- timer_interrupt, do_timer_interrupt [arch/i386/kernel/time.c]
- do_timer, update_process_times [kernel/timer.c]
- do_softirq [kernel/soft_irq.c]
- RESTORE_ALL, while loop [arch/i386/kernel/entry.S]

Notes:

1. La fonction "IRQ0x00_interrupt" (comme d'autres IRQ0xXY_interrupt) est directement dirigée par IDT (Interrupt Descriptor Table ou Tableau de descripteur d'interruption, semblable au Real Mode Interrupt Vector Table, voir chap 11), ainsi CHAQUE interruption arrivant au processeur est contrôlée par la routine "IRQ0x#NR_interrupt", où #NR est le numéro d'interruption. Nous nous référons à elle en tant que "gestionnaire d'irq d'emballage" ("wrapper irq handler").
2. des routines d'emballage sont exécutées, comme "do_IRQ","handle_IRQ_event" [arch/i386/kernel/irq.c].
3. Après ceci, la main est passée à la routine IRQ officielle (pointée par "handler()"), précédemment enregistré avec "request_irq" [arch/i386/kernel/irq.c], dans ce cas "timer_interrupt" [arch/i386/kernel/time.c].
4. la routine "timer_interrupt" [arch/i386/kernel/time.c] est exécutée, et quand elle se termine,
5. le contrôle revient à des routines assembleur [arch/i386/kernel/entry.S].

Description:

Pour gérer le Multitâche, Linux (comme chaque autre Unix) utilise un ''compteur'' variable pour suivre combien de CPU (processeur) a été utilisée par la Tâche. Ainsi, à chaque IRQ 0, le compteur est décrémenté (point 4) et, quand il atteint 0, nous devons commuter la Tâche de gérer le temps partagé (point 4 la variable "need_resched" est mise à 1, puis, au point 5 les routines assembleur contrôlent "need_resched" et appellent, si besoin, "le programme" [kernel/sched.c]).

6.3 Programmateur

Le programmateur est le bout de code qui choisit quelle Tâche doit être exécutée à un moment donné (chooses what Task has to be executed at a given time).

A chaque fois que vous devez changer la Tâche courante, choisissez un candidat. Ci-dessous il y a la fonction ''programme [kernel/sched.c]''.

```
|schedule
   |do_softirq // manages post-IRQ work
   |for each task
      |calculate counter
   |prepare_to__switch // does anything
   |switch_mm // change Memory context (change CR3 value)
   |switch_to (assembler)
      |SAVE ESP
      |RESTORE future_ESP
      |SAVE EIP
      |push future_EIP *** push parameter as we did a call
         |jmp __switch_to (it does some TSS work)
         |__switch_to()
          ..
         |ret *** ret from call using future_EIP in place of call address
      new_task
```

6.4 Moitié inférieure, files d'attente de Tâche et Tasklets

Vue d’ensemble

En Unix classique, quand un IRQ se produit (par un périphérique), Unix fait la "commutation de Tâche" pour interroger la Tâche qui a demandé le périphérique.

Pour améliorer les performances, Linux peut remettre le travail non urgent à plus tard, pour mieux gérer la grande vitesse des évènements.

Ce dispositif est géré depuis le noyau 1.x par "la moitié inférieure" (BH pour Bottom Half). Le gestionnaire d’irq "marque" une moitié inférieure, pour être exécuté plus tard, le temps de programmé (scheduling time).

Dans les derniers noyaus il y a une "queue de tâche" qui est plus dynamique que BH et il y a aussi une petite tâche ("tasklet") pour gérer les environnements multiprocesseur.

Le schéma BH est:

1. Déclaration
2. Marque
3. Exécution

Déclaration

```
#define DECLARE_TASK_QUEUE(q) LIST_HEAD(q)
#define LIST_HEAD(name) \
   struct list_head name = LIST_HEAD_INIT(name)
struct list_head {
   struct list_head *next, *prev;
};
#define LIST_HEAD_INIT(name) { &(name), &(name) }

      ''DECLARE_TASK_QUEUE'' [include/linux/tqueue.h, include/linux/list.h]
```

La macro "DECLARE_TASK_QUEUE(q)" est utilisée pour déclarer une structure appelée file d’attente de tâche qui gère "q" (managing task queue).

Marque

Voici le schéma Ica pour la fonction "mark_bh" [include/linux/interrupt.h]:

```
|mark_bh(NUMBER)
   |tasklet_hi_schedule(bh_task_vec + NUMBER)
      |insert into tasklet_hi_vec
         |__cpu_raise_softirq(HI_SOFTIRQ)
            |soft_active |= (1 << HI_SOFTIRQ)

                   ''mark_bh''[include/linux/interrupt.h]
```

Par exemple, quand un gestionnaire d’IRQ veut "remettre" du travail, il ferait "mark_bh(NOMBRE)", où NOMBRE est un BH déclaré (voir la section précédente).

Exécution

Nous pouvons voir ceci appelé par la fonction "do_IRQ" [arch/i386/kernel/irq.c]:

```
|do_softirq
   |h->action(h)-> softirq_vec[TASKLET_SOFTIRQ]->action -> tasklet_action
      |tasklet_vec[0].list->func
```

"h->action(h);" est la fonction qui a été précédemment alignée.

6.5 Routines de très bas niveau

set_intr_gate

set_trap_gate

set_task_gate (non utilisé).

(*interrupt)[NR_IRQS](void) = { IRQ0x00_interrupt, IRQ0x01_interrupt, ..}

NR_IRQS = 224 [kernel 2.4.2]

6.6 Commutation de Tâche

Quand la commutation de Tâche se passe-t-elle?

Maintenant nous allons voir comment le noyau de Linux permute d'une Tâche à l'autre.

La permutation de Tâche est nécessaire dans beaucoup de cas, comme le suivant:

- quand TimeSlice finit, nous devons donner accès à d'autres Tâches
- quand une Tâche décide d'accéder à une ressource, elle dort en l'attendant, ainsi nous devons choisir une autre tâche
- quand une Tâche attend une tube, nous devons donner accès à une autre Tâche, qui écrirait dans la tube

Commutation de Tâche

```
                           TASK SWITCHING TRICK
#define switch_to(prev,next,last) do {                                 \
        asm volatile("pushl %%esi\n\t"                                 \
                     "pushl %%edi\n\t"                                 \
                     "pushl %%ebp\n\t"                                 \
                     "movl %%esp,%0\n\t"        /* save ESP */         \
                     "movl %3,%%esp\n\t"        /* restore ESP */      \
                     "movl $1f,%1\n\t"          /* save EIP */         \
                     "pushl %4\n\t"             /* restore EIP */      \
                     "jmp __switch_to\n"                               \
                     "1:\t"                                            \
                     "popl %%ebp\n\t"                                  \
                     "popl %%edi\n\t"                                  \
                     "popl %%esi\n\t"                                  \
                     :"=m" (prev->thread.esp),"=m" (prev->thread.eip),  \
```

```
                          "=b" (last)                                  \
                         :"m" (next->thread.esp),"m" (next->thread.eip),    \
                          "a" (prev), "d" (next),                      \
                          "b" (prev));                                 \
} while (0)
```

L’astuce c’est:

1. ’’pushl %4’’ qui met future_EIP dans la pile
2. ’’jmp __switch_to’’ qui exécute la fonction’’__switch_to’’, mais au contraire de ’’call’’ nous reviendrons à la valeur suivante (to valued pushed in point 1) du poiint 1 (donc une nouvelle tâche!)

```
      U S E R   M O D E                 K E R N E L     M O D E

 |          |     |          |        |          |      |          |
 |          |     |          | Timer  |          |      |          |
 |          |     |  Normal  |  IRQ   |          |      |          |
 |          |     |   Exec   |------>|Timer_Int.|      |          |
 |          |     |     |    |        | ..       |      |          |
 |          |     |    \|/   |        |schedule()|      | Task1 Ret|
 |          |     |          |        |_switch_to|<--   |  Address |
 |__________|     |__________|        |          |  |   |          |
                                      |          |  |S  |          |
Task1 Data/Stack   Task1 Code         |          |  |w  |          |
                                      |          | T|i  |          |
                                      |          | a|t  |          |
 |          |     |          |        |          | s|c  |          |
 |          |     |          | Timer  |          | k|h  |          |
 |          |     |  Normal  |  IRQ   |          |  |i  |          |
 |          |     |   Exec   |------>|Timer_Int.|  |n  |          |
 |          |     |     |    |        | ..       |  |g  |          |
 |          |     |    \|/   |        |schedule()|  |   | Task2 Ret|
 |          |     |          |        |_switch_to|<--   |  Address |
 |__________|     |__________|        |__________|      |__________|

Task2 Data/Stack   Task2 Code        Kernel Code  Kernel Data/Stack
```

6.7 Fourche

Vue d’ensemble

La fourche est utilisée pour créer une autre Tâche. Nous commençons par une Tâche Parent, et nous copions plusieurs structures de données vers l’Enfant de la Tâche.

```
                               |         |
                               | ..      |
         Task Parent           |         |
         |         |           |         |
         |  fork   |---------->|  CREATE |
         |         |          /|   NEW   |
         |_________|         / |   TASK  |
                            /  |         |
             ---           /   |         |
             ---          /    | ..      |
                         /     |         |
         Task Child     /
```

```
|         |   /
|  fork   |<-/
|         |
|_________|

              Fork SysCall
```

Ce qui n'est pas copié

La nouvelle Tâche juste créée (''Enfant de Tâche '') est presque égale au parent (''Parent de Tâche''), il y a seulement quelques différences:

1. évidemment le PID
2. l'enfant ''fork()'' renverra 0, alors que le parent ''fork() ''renverra le PID de la Tâche Enfant, pour les distinguer en Mode Utilisateur
3. Toutes les pages de données de l'enfant sont marquées ''LECTURE + EXÉCUTION'', aucune "ÉCRITURE'' (tandis que le parent a droit d'ÉCRITURE sur ses propres pages) ainsi, quand une demande d'écriture se produit, une exception de ''Faute de page'' est générée qui créera une nouvelle page indépendante: ce mécanisme s'appelle ''Copy on Write'' (copie sur écriture) (voir Chap.10).

Fourche ICA

```
|sys_fork
   |do_fork
      |alloc_task_struct
         |__get_free_pages
       |p->state = TASK_UNINTERRUPTIBLE
       |copy_flags
       |p->pid = get_pid
       |copy_files
       |copy_fs
       |copy_sighand
       |copy_mm // should manage CopyOnWrite (I part)
          |allocate_mm
          |mm_init
             |pgd_alloc -> get_pgd_fast
                |get_pgd_slow
          |dup_mmap
             |copy_page_range
                |ptep_set_wrprotect
                   |clear_bit // set page to read-only
          |copy_segments // For LDT
       |copy_thread
          |childregs->eax = 0
          |p->thread.esp = childregs // child fork returns 0
          |p->thread.eip = ret_from_fork // child starts from fork exit
       |retval = p->pid // parent fork returns child pid
       |SET_LINKS // insertion of task into the list pointers
       |nr_threads++ // Global variable
       |wake_up_process(p) // Now we can wake up just created child
       |return retval

              fork ICA
```

- sys_fork [arch/i386/kernel/process.c]
- do_fork [kernel/fork.c]
- alloc_task_struct [include/asm/processor.c]
- __get_free_pages [mm/page_alloc.c]
- get_pid [kernel/fork.c]
- copy_files
- copy_fs
- copy_sighand
- copy_mm
- allocate_mm
- mm_init
- pgd_alloc -> get_pgd_fast [include/asm/pgalloc.h]
- get_pgd_slow
- dup_mmap [kernel/fork.c]
- copy_page_range [mm/memory.c]
- ptep_set_wrprotect [include/asm/pgtable.h]
- clear_bit [include/asm/bitops.h]
- copy_segments [arch/i386/kernel/process.c]
- copy_thread
- SET_LINKS [include/linux/sched.h]
- wake_up_process [kernel/sched.c]

Copy on Write (Copie sur Ecriture)

Pour implémenter la Copie sur Ecriture pour Linux:

1. Marquez toutes les pages copiées en lecture-seule, entraînant une Faute de Page quand une Tâche essaye d'y écrire.
2. Le gestionnaire de Faute de page cré une nouvelle page.

```
| Page
| Fault
| Exception
|
|
-----------> |do_page_fault
                |handle_mm_fault
                   |handle_pte_fault
                      |do_wp_page
                         |alloc_page      // Alloue une nouvelle page
                         |break_cow
                            |copy_cow_page // Copie l'ancienne page vers une nouvelle
                            |establish_pte // reconfigure les pointeurs de la Table de Page
                               |set_pte

             Page Fault ICA
```

- do_page_fault [arch/i386/mm/fault.c]
- handle_mm_fault [mm/memory.c]
- handle_pte_fault
- do_wp_page
- alloc_page [include/linux/mm.h]

- break_cow [mm/memory.c]
- copy_cow_page
- establish_pte
- set_pte [include/asm/pgtable-3level.h]

7. Gestion de Mémoire de Linux

7.1 Vue d’ensemble

Linux utilise segmentation + pagination, ce qui simplifie la notation.

Segments

Linux utilise seulement 4 segments:

- 2 segments (code and data/stack ou code et donnée/pile) pour l’ESPACE NOYAU de [0xC000 0000] (3 Go) à [0xFFFF FFFF] (4 Go)
- 2 segments (code et donnée/pile) pour l’ESPACE UTILISATEUR de [0] (0 Go) à [0xBFFF FFFF] (3 Go)

```
                                    __
   4 Go--->|                |     |
           |     Noyau      |     |   Espace Noyau (Code + Donnée/Pile)
           |                |   __|
   3 Go--->|----------------|   __
           |                |     |
           |                |     |
   2 Go--->|                |     |
           |     Tâche      |     |   Espace Utilisateur (Code + Donnée/Pile)
           |                |     |
   1 Go--->|                |     |
           |                |     |
           |________________|   __|
 0x00000000
          Adresses Linéaires Noyau/Utilisateur
```

7.2 Implémentation i386 spécifique

A nouveau, Linux implémente la Pagination en utilisant 3 Niveaux de Pagination, mais dans l’architecture i386 seulement 2 d’entre elles sont vraiment utilisées:

```
   ------------------------------------------------------------------
     A   D   R   E   S   S   E      L   I   N   E   A   I   R   E
   ------------------------------------------------------------------
        \___/                 \___/                     \_____/

     PD offset              PF offset                 Frame offset
     [10 bits]              [10 bits]                 [12 bits]
           |                      |                           |
           |                      |     -----------           |
           |                      |     |  Value  |----------|---------
           |     |          |     |     |---------|   /|\    |        |
           |     |          |     |     |         |    |     |        |
```

```
          |     |         |     |     |         |    | Frame offset  |
          |     |         |     |     |         |   \|/              |
          |     |         |     |     |---------|<------             |
          |     |         |     |     |         |      |             |
          |     |         |     |     |         |      | x 4096      |
          |     |         |  PF offset|_________|-------             |
          |     |         |       /|\ |         |                    |
      PD offset |_________|-----   |  |         |          __________|
            /|\ |         |    |   |  |         |          |
             |  |         |    |  \|/ |         |         \|/
 _____       |  |         |    ------>|_________|   PHYSICAL ADDRESS
|     |     \|/ |         |    x 4096 |         |
| CR3 |--------->|         |           |         |
|_____|         | ....... |           | ....... |
                |         |           |         |

               Dossier Page           Fichier Page

                       Pagination Linux i386
```

7.3 Traçage Mémoire

Linux gère le Contrôle d'Accès seulement avec des paginations, ainsi les différentes Tâches auront les mêmes adresses de segment, mais un CR3 différent (registre utilisé pour stocker l'Adresse de Page du Dossier), pointant vers une Page différente.

En Mode Utilisateur une Tâche ne peut pas dépasser la limite de 3 Go (0 x C0 00 00 00), ainsi seulement les 768 premières entrées de répertoire de page sont significative (768*4mb = 3Go).

Quand une Tâche passe en Mode Noyau (par Appel System ou par IRQ) d'autres entrées de répertoire de 256 pages deviennent importantes, et elles pointent vers les mêmes fichiers de page que toutes autres Tâches (qui sont les même que le noyau).

Notez que l'Espace Linéaire du Noyau (et seulement du noyau) est égal à l'Espace Physique du Noyau, ainsi:

```
          ________________ _____
         |Other KernelData|___  |  |                |
         |----------------|   |  |__|                |
         |     Kernel     |\  |____|   Real Other   |
3 GB --->|----------------| \      |   Kernel Data  |
         |                |\ \     |                |
         |              __|_\_\____|__   Real       |
         |      Tasks     |  \ \   |     Tasks      |
         |              __|___\_\__|__   Space      |
         |                |    \ \ |                |
         |                |     \ \|----------------|
         |                |      \ |Real KernelSpace|
         |________________|       \|________________|

         Adresses Logiques          Adresses Physiques
```

L'Espace Linéaire du Noyau correspond à l'Espace Physique du Noyau transféré 3 Go plus bas (translated 3GB down) (en fait les tables de page sont quelque chose comme { "00000000", "00000001" }, ainsi elles n'effectuent aucune virtualisation, elles reportent seulement des adresses physiques qu'elles prennent des linéaires).

Notez que vous n'aurez pas un "conflit d'adresses" entre les espaces Noyau et Utilisateur parce que nous pouvons gérer des adresses physiques avec les Tableaux de Page.

7.4 Attribution de mémoire de bas niveau

Initialisation du Démarrage

Nous commençons à partir du kmem_cache_init (lancé par start_kernel [init/main.c] au démarrage).

```
|kmem_cache_init
   |kmem_cache_estimate
```

kmem_cache_init [mm/slab.c]

kmem_cache_estimate

Maintenant nous continuons avec mem_init (également lancé par start_kernel[init/main.c])

```
|mem_init
   |free_all_bootmem
      |free_all_bootmem_core
```

mem_init [arch/i386/mm/init.c]

free_all_bootmem [mm/bootmem.c]

free_all_bootmem_core

Allocation du temps d'exécution

Sous Linux, quand nous voulons allouer de la mémoire, par exemple pendant le mécanisme "copy_on_write" (voir Chap.10), nous appelons:

```
|copy_mm
   |allocate_mm = kmem_cache_alloc
      |__kmem_cache_alloc
         |kmem_cache_alloc_one
            |alloc_new_slab
               |kmem_cache_grow
                  |kmem_getpages
                     |__get_free_pages
                        |alloc_pages
                           |alloc_pages_pgdat
                              |__alloc_pages
                                 |rmqueue
                                 |reclaim_pages
```

Les fonctions peuvent être trouvées ci-dessous:

- copy_mm [kernel/fork.c]
- allocate_mm [kernel/fork.c]
- kmem_cache_alloc [mm/slab.c]
- __kmem_cache_alloc
- kmem_cache_alloc_one
- alloc_new_slab
- kmem_cache_grow
- kmem_getpages
- __get_free_pages [mm/page_alloc.c]
- alloc_pages [mm/numa.c]
- alloc_pages_pgdat
- __alloc_pages [mm/page_alloc.c]
- rm_queue
- reclaim_pages [mm/vmscan.c]

TODO: Comprendre les Zones

7.5 Échange (Swap ou glissement)

Vue d'ensemble

L'échange est géré par le service kswapd (fil du noyau).

kswapd

Comme d'autres fils du noyau, le kswapd a une boucle principale qui attend de se réveiller.

```
|kswapd
   |// initialization routines
   |for (;;) { // Main loop
      |do_try_to_free_pages
      |recalculate_vm_stats
      |refill_inactive_scan
      |run_task_queue
      |interruptible_sleep_on_timeout // on dort en attendant une nouvelle demande d'échange
   |}
```

- kswapd [mm/vmscan.c]
- do_try_to_free_pages
- recalculate_vm_stats [mm/swap.c]
- refill_inactive_scan [mm/vmswap.c]
- run_task_queue [kernel/softirq.c]
- interruptible_sleep_on_timeout [kernel/sched.c]

Quand avons-nous besoin d’échanger?

L’échange est nécessaire quand nous devons accéder à une page qui n’est pas en mémoire physique.

Linux utilise le fil du noyau ’’kswapd’’ pour ce faire. Quand la Tâche reçoit une faute d’exception de page nous faisons ce qui suit:

```
 | Faute d’Execption de Page
 | causée par toutes ces conditions:
 |   a-) Page Utilisateur
 |   b-) Accès lecture ou écriture
 |   c-) Page non présente
 |
 |
 -----------> |do_page_fault
                 |handle_mm_fault
                    |pte_alloc
                       |pte_alloc_one
                          |__get_free_page = __get_free_pages
                             |alloc_pages
                                |alloc_pages_pgdat
                                   |__alloc_pages
                                      |wakeup_kswapd // Nous réveillons le fil de noyau kswapd

                  Page Fault ICA
```

- do_page_fault [arch/i386/mm/fault.c]
- handle_mm_fault [mm/memory.c]
- pte_alloc
- pte_alloc_one [include/asm/pgalloc.h]
- __get_free_page [include/linux/mm.h]
- __get_free_pages [mm/page_alloc.c]
- alloc_pages [mm/numa.c]
- alloc_pages_pgdat
- __alloc_pages
- wakeup_kswapd [mm/vmscan.c]

8. Réseau Linux Linux Networking

8.1 Comment le réseau Linux est-il géré?

Il existe un pilote de périphérique pour chaque type de NIC. De l’intérieur, Linux appellera TOUJOURS une routine standard de haut niveau: "netif_rx [net/core/dev.c]", qui contrôlera à quel protocole de niveau 3 le cadre appartient, et il appellera la bonne fonction de niveau 3 (ainsi nous utiliserons un pointeur vers la fonction pour déterminer laquelle est la bonne). (There exists a device driver for each kind of NIC. Inside it, Linux will ALWAYS call a standard high level routing: "netif_rx [net/core/dev.c]", which will controls what 3 level protocol the frame belong to, and it will call the right 3 level function (so we’ll use a pointer to the function to determine which is right).

8.2 Exemple TCP

Nous allons voir maintenant un exemple de ce qui se produit quand nous envoyons un paquet TCP à Linux, à partir de l'appel ''netif_rx [net/core/dev.c] ''.

Gestion d'interruption: "netif_rx"

```
|netif_rx
   |__skb_queue_tail
      |qlen++
      |* simple pointer insertion *
   |cpu_raise_softirq
      |softirq_active(cpu) |= (1 << NET_RX_SOFTIRQ) // met l'octet NET_RX_SOFTIRQ dans le vecteur BH
```

Fonctions:

- __skb_queue_tail [include/linux/skbuff.h]
- cpu_raise_softirq [kernel/softirq.c]

Gestion de Post Interruption: "net_rx_action"

Une fois que l'interaction d'IRQ est finie, nous devons suivre la prochaine partie de la vie du cadre et examiner ce que fait NET_RX_SOFTIRQ.

Nous appellerons ensuite ''net_rx_action [net/core/dev.c]'' selon "net_dev_init [net/core/dev.c]".

```
|net_rx_action
   |skb = __skb_dequeue (the exact opposite of __skb_queue_tail)
   |for (ptype = first_protocol; ptype < max_protocol; ptype++) // Determine
      |if (skb->protocol == ptype)                               // what is the network protocol
         |ptype->func -> ip_rcv // according to ''struct ip_packet_type [net/ipv4/ip_output.c]''

    **** MAINTENANT NOUS SAVONS QUEL PAQUET EST IP ****
         |ip_rcv
            |NF_HOOK (ip_rcv_finish)
               |ip_route_input // cherche dans la table de routage pour déterminer la fonction à appeler
                  |skb->dst->input -> ip_local_deliver // selon la précédente vérification de la table de routage, la destination est la machine locale
                     |ip_defrag // rassemble les fragments IP
                        |NF_HOOK (ip_local_deliver_finish)
                           |ipprot->handler -> tcp_v4_rcv // selon ''tcp_protocol [include/net/protocol.c]''

     **** MAINTENANT NOUS SAVONS QUEL PAQUET EST TCP ****
                           |tcp_v4_rcv
                              |sk = __tcp_v4_lookup
                              |tcp_v4_do_rcv
                                 |switch(sk->state)

     *** Le paquet peut être envoyé à la tâche qui utilise une connexion (socket) relative ***
                                 |case TCP_ESTABLISHED:
                                    |tcp_rcv_established
                                       |__skb_queue_tail // enfile le packet vers la connexion
                                       |sk->data_ready -> sock_def_readable
                                          |wake_up_interruptible


     *** Le packet est toujours pris en main par le 3-way TCP handshake ***
                                 |case TCP_LISTEN:
                                    |tcp_v4_hnd_req
                                       |tcp_v4_search_req
                                       |tcp_check_req
                                          |syn_recv_sock -> tcp_v4_syn_recv_sock
                                       |__tcp_v4_lookup_established
                                 |tcp_rcv_state_process

                    *** 3-Way TCP Handshake ***
                                    |switch(sk->state)
                                    |case TCP_LISTEN: // We received SYN
                                       |conn_request -> tcp_v4_conn_request
                                          |tcp_v4_send_synack // Send SYN + ACK
                                             |tcp_v4_synq_add // set SYN state
                                    |case TCP_SYN_SENT: // we received SYN + ACK
                                       |tcp_rcv_synsent_state_process
                                          tcp_set_state(TCP_ESTABLISHED)
                                             |tcp_send_ack
                                                |tcp_transmit_skb
                                                   |queue_xmit -> ip_queue_xmit
                                                      |ip_queue_xmit2
                                                         |skb->dst->output
                                    |case TCP_SYN_RECV: // We received ACK
                                       |if (ACK)
                                          |tcp_set_state(TCP_ESTABLISHED)
```

Fonctions ci-dessous:

- net_rx_action [net/core/dev.c]
- __skb_dequeue [include/linux/skbuff.h]
- ip_rcv [net/ipv4/ip_input.c]
- NF_HOOK -> nf_hook_slow [net/core/netfilter.c]
- ip_rcv_finish [net/ipv4/ip_input.c]
- ip_route_input [net/ipv4/route.c]
- ip_local_deliver [net/ipv4/ip_input.c]
- ip_defrag [net/ipv4/ip_fragment.c]
- ip_local_deliver_finish [net/ipv4/ip_input.c]
- tcp_v4_rcv [net/ipv4/tcp_ipv4.c]
- __tcp_v4_lookup
- tcp_v4_do_rcv
- tcp_rcv_established [net/ipv4/tcp_input.c]
- __skb_queue_tail [include/linux/skbuff.h]
- sock_def_readable [net/core/sock.c]
- wake_up_interruptible [include/linux/sched.h]
- tcp_v4_hnd_req [net/ipv4/tcp_ipv4.c]
- tcp_v4_search_req
- tcp_check_req
- tcp_v4_syn_recv_sock
- __tcp_v4_lookup_established
- tcp_rcv_state_process [net/ipv4/tcp_input.c]
- tcp_v4_conn_request [net/ipv4/tcp_ipv4.c]
- tcp_v4_send_synack
- tcp_v4_synq_add
- tcp_rcv_synsent_state_process [net/ipv4/tcp_input.c]
- tcp_set_state [include/net/tcp.h]
- tcp_send_ack [net/ipv4/tcp_output.c]

Description:

- D'abord nous déterminons le type de protocole (IP, puis TCP)
- NF_hook (fonction) est une routine d'emballage (wrapper) qui gère d'abord le filtre réseau (par exemple mur de feu), puis lui appelle la ''fonction''.
- Après que nous contrôlions 3-way TCP Handshake qui se compose de:

```
SERVEUR (ECOUTE)                                  CLIENT (CONNECTING)
                           SYN
                   <--------------------


                        SYN + ACK
                   -------------------->


                           ACK
                   <--------------------

                    3-Way TCP handshake
```

- À la fin que nous devons seulement lancer "tcp_rcv_established [net/ipv4/tcp_input.c]" qui donne le paquet à la connexion utilisateur et le réveille.

9. Système de fichier Linux

TODO

10. Trucs utiles

10.1 Pile et tas (Stack and Heap)

Vue d'ensemble

Ici nous regardons comment la "pile" et le "tas" sont alloués en mémoire

Allocation mémoire

```
FF..        |                 | <-- bas de la pile
       /|\  |                 |   |
 valeurs|   |                 |   |   pile
 élevées|   |                 |  \|/  grandit
            |                 |
XX..        |                 | <-- haut de la pile [Pointeur de pile]
            |                 |
            |                 |
            |                 |
00..        |_________________| <-- fin de pile [Segment de pile]

                   Pile
```

Les valeurs d'adresse de mémoire commencent à 00. (qui est aussi là où le Segment de Pile commence) et ils vont jusqu'à la valeur FF..

XX.. est la valeur réelle du pointeur de pile.

La pile est utilisée par les fonctions pour:

1. les variables globales
2. les variables locales
3. l'adresse de retour

Par exemple, pour une fonction classique:

```
 |int foo_function (parameter_1, parameter_2, ..., parameter_n) {
    |variable_1 declaration;
    |variable_2 declaration;
      ..
    |variable_n declaration;

    |// Body function
    |dynamic variable_1 declaration;
    |dynamic variable_2 declaration;
     ..
    |dynamic variable_n declaration;

    |// Le code est à l'intérieur du Segment Code, pas le segment Donnée/Pile!

    |return (ret-type) value; // souvent c'est à l'intérieur d'un registre, pour l'i386 le registre eax est utilisé.
```

```
 |}
nous avons

          |                       |
          | 1. parameter_1 pushed | \
    P     | 2. parameter_2 pushed |  | Avant
    I     | ...................   |  | l'appel
    L     | n. parameter_n pushed | /
    E     | ** Return address **  | -- Appel
          | 1. local variable_1   | \
          | 2. local variable_2   |  | Après
          | .................     |  | l'appel
          | n. local variable_n   | /
          |                       |
         ...                     ...   Pile
         ...                     ...   libre
          |                       |
    T     | n. dynamic variable_n | \
    A     | ...................   |  | Alloué par
    S     | 2. dynamic variable_2 |  | malloc & kmalloc
          | 1. dynamic variable_1 | /
          |_______________________|

           Usage typique de la pile

Note: l'ordre des variables peut être différent selon l'architecture du matériel.
```

10.2 Application vs Processus

Définition basse

Nous devons distinguer 2 concepts:

- Application: c'est le code utile que nous voulons exécuter
- Processus: c'est l'IMAGE en mémoire de l'application (elle dépend de la stratégie mémoire utilisée, de la segmentation et/ou de la Pagination).

Souvent le Processus s'appelle également Tâche ou Fil.

10.3 Bloquages (Locks)

Vue d'ensemble

2 genre de serrures:

1. intraCPU
2. interCPU

10.4 Copy_on_write

Copy_on_write est un mécanisme utilisé pour réduire l'utilisation de mémoire. Il remet l'allocation mémoire à plus tard de mémoire jusqu'à ce que la mémoire soit vraiment nécessaire.

Par exemple, quand une Tâche exécute l'Appel Système "fork()" (pour créer une autre Tâche), nous utilisons toujours les mêmes pages de mémoire que le parent, en mode lecture seule. Quand une Tâche ÉCRIT dans la page, elle cause une exception et la page est copiée et marquée "rw" (lecture, écriture).

```
1-) la page X est partagée entre la Tâche parent et la Tâche enfant
 Tâche Parent
 |          | Accès RO    ______
 |          |---------->|Page X|
```

```
 |_________|           |______|
                          /|\
                           |
 Têche Enfant              |
 |         | Accès RO      |
 |         |----------------
 |_________|


2-) Write request
 Task Parent
 |         | Accès RO   ______
 |         |---------->|Page X|    Trying to write
 |_________|           |______|
                          /|\
                           |
 Task Child                |
 |         | Accès RO      |
 |         |----------------
 |_________|


3-) Final Configuration: Either Task Parent and Task Child have an independent copy of the Page, X and Y
 Task Parent
 |         | Accès RW   ______
 |         |---------->|Page X|
 |_________|           |______|


 Task Child
 |         | Accès RW   ______
 |         |---------->|Page Y|
 |_________|           |______|
```

11. Détails spécifiques 80386

11.1 Procédure de démarrage

```
bbootsect.s [arch/i386/boot]
setup.S (+video.S)
head.S (+misc.c) [arch/i386/boot/compressed]
start_kernel [init/main.c]
```

11.2 Descripteurs 80386 (et plus)

Vue d’ensemble

Les descripteurs sont des structure de données utilisées par le microprocesseur Intel i386+ pour rendre la mémoire virtuelle.

Types de descripteurs

- GDT (Table Globale de Descripteur)
- LDT (Table Locale de Descripteur)
- IDT (Table de Descripteur d’Interruption)

12. IRQ

12.1 Vue d'ensemble

IRQ est un signal asyncrone envoyé au microprocesseur pour annoncer qu'une requête est complétée

12.2 Schéma d'interaction

```
                                 |<-->  IRQ(0) [Timer]
                                 |<-->  IRQ(1) [Device 1]
                                 | ..
                                 |<-->  IRQ(n) [Device n]
_________________________________|
 /|\       /|\           /|\
  |         |             |
 \|/       \|/           \|/

Tâche(1)  Tâche(2) ..   Tâche(N)


          IRQ - Schéma d'Interaction de Tâches
```

Que se produit?

Un SE typique utilise beaucoup de signaux IRQ pour interrompre l'exécution normale de processus et fait le ménage. Ainsi:

1. IRQ (i) se produit et Tâche(j) est interrompue
2. IRQ(i)_handler est exécuté
3. le contrôle revient à la Tâche(j) interrompue

Sous Linux, quand un IRQ survient, d'abord la routine d'emballage IRQ (appelée "interrupt0x??") est appellée, puis l'IRQ(i)_handler "officiel" est exécuté. Ceci permet certains droits comme la préemption de timeslice.

13. Fonctions utilitaires

13.1 list_entry [include/linux/list.h]

Definition:

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

Signification:

La macro "list_entry" est utilisée pour retrouver un pointeur de struct parent, en utilisant seulement un pointeur de struct interne.

Exemple:

```
struct __wait_queue {
   unsigned int flags;
   struct task_struct * task;
   struct list_head task_list;
};
struct list_head {
   struct list_head *next, *prev;
};

// et avec la définition type:
typedef struct __wait_queue wait_queue_t;

// nous aurons
wait_queue_t *out list_entry(tmp, wait_queue_t, task_list);

// où tmp pointe vers list_head
```

Ainsi, dans ce cas, au moyen du pointeur *tmp [list_head] nous retrouvons le pointeur *out [wait_queue_t].

```
 ____________ <---- *out [nous calculons que]
|flags       |             /|\
|task *-->   |              |
|task_list   |<----    list_entry
|  prev * -->|    |         |
|  next * -->|    |         |
|____________|    ----- *tmp [nous avons ça]
```

13.2 Sommeil

Code de sommeil

Fichiers:

- kernel/sched.c
- include/linux/sched.h
- include/linux/wait.h
- include/linux/list.h

Fonctions:

- interruptible_sleep_on
- interruptible_sleep_on_timeout
- sleep_on
- sleep_on_timeout

Fonctions appelées:

- init_waitqueue_entry
- __add_wait_queue
- list_add
- __list_add

- __remove_wait_queue

Analyse D'InterCallings:

```
|sleep_on
   |init_waitqueue_entry  --
   |__add_wait_queue        |   enqueuing request to resource list
      |list_add              |
         |__list_add        --
   |schedule              ---     waiting for request to be executed
      |__remove_wait_queue --
      |list_del              |   dequeuing request from resource list
         |__list_del        --
```

Description:

Sous Linux chaque ressource (idéalement un objet partagé par plusieurs utilisateurs et plusieurs processus), a une file pour gérer TOUTES les Tâches qui la demandant.

Cette file s'appelle l"file d'attente" et elle consiste en plusieurs éléments que nous appellerons l"élément de file d'attente":

```
***   structure de file d'attente [include/linux/wait.h]  ***


struct __wait_queue {
   unsigned int flags;
   struct task_struct * task;
   struct list_head task_list;
}
struct list_head {
   struct list_head *next, *prev;
};
```

Fonctionnement de graphique:

```
        ***  élément de file d'attente  ***

                              /|\
                               |
       <--[préc *, drapeau, tâchesk *, suiv *]-->




                 ***  liste de file d'attente ***

          /|\           /|\           /|\                /|\
           |             |             |                  |
--> <--[tâch1]--> <--[tâch2]--> <--[tâche3]--> .... <--[tâcheN]--> <--
|                                                                  |
|__________________________________________________________________|



              ***   tête de file d'attente ***

       tâche1 <--[préc *, blocage, suiv *]--> tâcheN
```

"tête de file d'attente d'attente" pointe vers premier (avec suiv *) et dernier (avec préc *) éléments de la "liste de file d'attente".

Quand un nouvel élément doit être ajouté, "__add_wait_queue" [include/linux/wait.h] est appellé, après quoi la routine générique "list_add" [include/linux/wait.h], est exécutée:

```
***   function list_add [include/linux/list.h]  ***

// classic double link list insert
static __inline__ void __list_add (struct list_head * new,  \
                                   struct list_head * prev, \
                                   struct list_head * next) {
   next->prev = new;
   new->next = next;
   new->prev = prev;
   prev->next = new;
}
```

Pour compléter la description, nous voyons aussi la fonction "__list_del" [include/linux/list.h] appelée par "list_del" [include/linux/list.h] dans "remove_wait_queue" [include/linux/wait.h]:

```
***   fonction list_del [include/linux/list.h]  ***


// classic double link list delete
static __inline__ void __list_del (struct list_head * prev, struct list_head * next) {
   next->prev = prev;
   prev->next = next;
}
```

Consideration sur la pile

Une liste type (ou la file) est habituellement gérée en l'allouant dans le Tas (voir le Chap.10 pour la définition de Tas et Pile au sujet et voir où les variables sont assignées). Autrement ici, nous allouons statiquement les données File d'Attente dans une variable locale (Pile), puis la fonction est interrompue par le programme, à la fin, (retournant du programme (returning from scheduling)) effacera la variable locale.

```
  new task <----|          task1 <------|          task2 <------|
                |                       |                       |
                |                       |                       |
|..........|    |        |..........|   |        |..........|   |
|wait.flags|    |        |wait.flags|   |        |wait.flags|   |
|wait.task_|____|        |wait.task_|___|        |wait.task_|___|
|wait.prev |-->          |wait.prev |-->         |wait.prev |-->
|wait.next |-->          |wait.next |-->         |wait.next |-->
|..        |             |..        |            |..        |
|schedule()|             |schedule()|            |schedule()|
|..........|             |..........|            |..........|
|__________|             |__________|            |__________|

   Stack                    Stack                   Stack
```

14. Variables statiques

14.1 Vue d'ensemble

Linux est écrit en ''langue C'', et toutes les applications ont:

1. Des variables locales
2. Des modules variables (dans le fichier source et relatif seulement à ce module)
3. Des variables Globales/Statiques présentent dans seulement 1 copie (la même pour tous les modules)

Quand une variable statique est modifiée par un module, tous autres modules voient la nouvelle valeur.

Les variables statiques sous Linux sont très importantes, parce qu'elles sont la seule façon d'ajouter un nouveau support au noyau: elles sont typiquement des pointeurs en tête d'une liste d'éléments enregistrés, qui peuvent être:

- ajouté
- supprimé
- peut-être modifié

```
                              _______      _______      _______
Variable globale  -------> |Item(1)| -> |Item(2)| -> |Item(3)|  ..
                             |_______|    |_______|    |_______|
```

14.2 Variables principales

Current (Courant)

```
                              ________________
Current ----------------> | Actual process |
                             |________________|
```

Current pointe vers la structure ''task_struct'' , qui contient toutes les données sur un processus:

- pid, name, state, counter, policy of scheduling (pid, nom, état, compteur, politique de programmation)
- les pointeurs vers plusieurs structures de données comme: fichiers, vfs, autres processus, signaux...

Current n'est pas une vraie variable, c'est

```
static inline struct task_struct * get_current(void) {
   struct task_struct *current;
   __asm__("andl %%esp,%0; ":"=r" (current) : "0" (~8191UL));
   return current;
}
#define current get_current()
```

Les lignes au-dessus prennent juste la valeur du registre ''esp'' (pointeur de pile) et le rendent disponible comme une variable, de laquelle nous pouvons pointer vers la structure task_struct.

De l'élément ''curent'' nous pouvons accéder directement à n'importe quel autre processus (prêt, arrêté ou dans tout autre état) de la structure de données noyau, par exemple en changeant l'ETAT (comme un pilote E/S le ferait), le PID, la présence dans la liste prêt ou la liste bloqué, etc...

Systèmes de fichiers enregistrés

```
                          ______      _______      ______
file_systems  ------> | ext2 | -> | msdos | -> | ntfs |
 [fs/super.c]         |______|    |_______|    |______|
```

Quand vous utilisez une commande comme ''modprobe un_fs'' vous ajoutez une nouvelle entrée à la liste des systèmes de fichiers, dès que vous l'enlevez (avec ''rmmod'') il est supprimé.

Systèmes de fichiers montés

```
                         ______      _______      ______
mount_hash_table  ---->|   /  | -> | /usr  | -> | /var |
[fs/namespace.c]       |______|    |_______|    |______|
```

Quand vous utilisez la commande ''mount'' pour ajouter un fs, la nouvelle entrée est insérée dans la liste, jusqu'à ce que la commande ''umount'' supprime l'entrée.

Type de Paquet Réseau Enregistré

```
                         ______      _______      ______
     ptype_all  ------>|  ip  | -> |  x25  | -> | ipv6 |
[net/core/dev.c]       |______|    |_______|    |______|
```

Par exemple, si vous ajoutez le support IPv6 (en chargeant le module relatif) une nouvelle entrée est ajoutée à la liste.

Protocole de Réseau Internet Enregistré

```
                           ______      _______      _______
inet_protocol_base ----->| icmp | -> |  tcp  | -> |  udp  |
[net/ipv4/protocol.c]    |______|    |_______|    |_______|
```

Aussi d'autres types de paquet ont plusieurs protocoles internes dans chaque liste (comme IPv6).

```
                           ______      _______      _______
inet6_protos ----------->|icmpv6| -> | tcpv6 | -> | udpv6 |
[net/ipv6/protocol.c]    |______|    |_______|    |_______|
```

Périphérique Réseau Enregistré

```
                           ______      _______      _______
dev_base --------------->|  lo  | -> |  eth0 | -> |  ppp0 |
[drivers/core/Space.c]   |______|    |_______|    |_______|
```

Périphérique Caractère (Char) Enregistré

```
                           ______      _______      ________
chrdevs ---------------->|  lp  | -> | keyb  | -> | serial |
[fs/devices.c]           |______|    |_______|    |________|
```

’’chrdevs’’ n’est pas un pointeur vers une vraie liste, mais c’est un vecteur standard.

Périphérique Bloc Enregistré

```
                            ______      ______      ________
bdev_hashtable --------->|  fd  | -> |  hd  | -> |  scsi  |
[fs/block_dev.c]          |______|    |______|    |________|
```

’’bdev_hashtable’’ est un vecteur de hachage.

15. Glossaire

16. Liens

Site de téléchargemet du Noyau Linux Officiel et de patches

Grosse documentation sur le Noyau Linux

Liste de diffusion Officielle du Noyau

Guides du Project de Documentation Linux