Chapitre 1

Compiler le noyau Linux

Sommaire

Ce chapitre présente les principaux éléments permettant de modifier le noyau Linux. Nous essaierons dans ce chapitre de nous abstraire de toutes les contraintes liées à l'objectif (la modification du noyau), pour nous concentrer sur les outils permettant de mettre en place une telle modification. De ce fait, les exemples de code proposés ici pourront paraître particulièrement futiles, mais il est bien important de ne pas perdre de vue le but recherché, qui est de comprendre *comment* modifier le noyau.

Nous allons dans un premier temps découvrir comment compiler le noyau, cette opération est probablement familière à nombre d'entre vous, mais il est important qu'elle soit bien maîtrisée par chacun avant de passer à la suite.

Nous nous intéresserons alors à la notion de module, nous verrons comment utiliser les modules sous Linux, comment les charger ou les décharger dynamiquement dans ou depuis un noyau en cours d'exécution.

Nous étudierons alors dans la section suivante les principes de base permettant d'écrire un module, cette technique est en effet à favoriser chaque fois que possible dans le développement de code noyau puisqu'elle simplifie et accélère notablement la tâche du développeur.

Nous consacrerons enfin la dernière section au débogage, activité aussi peu prisée que malheureusemnt nécessaire et particulièrement délicate lorsqu'il s'agit de code du noyau.

1.1 Compiler le noyau Linux $Revision : 1.5$

Nous allons découvrir ensemble dans cette section comment obtenir les sources du noyau Linux, comment y appliquer des *patches*, et comment le compiler. Il est en effet nécessaire de maîtriser ces étapes avant de pouvoir envisager sereinement de se lancer dans la modification ou l'ajout de code source !

1.1.1 Obtenir le noyau Linux

Le noyau Linux (ainsi, d'ailleurs, que l'essentiel des autres logiciels permettant de composer le système Linux) est entièrement libre, et peut donc être obtenu aisément à un coût généralement faible, car limité aux frais de mise à disposition (par exemple l'édition d'un cédérom).

Vous pouvez ainsi obtenir Linux en achetant une revue informatique offrant un cédérom, ou encore directement une des nombreuses distributions disponibles sur le marché. De telles distributions coûtent généralement quelques centaines de francs et contiennent traditionnellement le noyau Linux ainsi que de nombreuses applications gratuites ; toutes ces applications sont pré-compilées et accompagnées d'outils permettant d'installer de façon cohérente un système complet parfaitement opérationnel.

Certaines distributions peuvent également intégrer des applications payantes, qui justifient généralement à elles seules les différences de prix.

Le choix d'une distribution, s'il n'est pas motivé par la présence de telle ou telle application précise, semble souvent relever davantage de critères subjectifs que d'une véritable analyse objective. Nous ne nous engagerons pas ici dans un débat si sensible.

Si l'obtention, via le réseau, d'une distribution complète reste encore pénible pour le particulier, celle du noyau Linux seul est tout à fait envisageable. Ceci est d'autant plus vrai qu'il n'est pas nécessairement utile de transférer l'intégralité de chaque nouvelle version.

Vous pourrez donc trouver les fichiers sources de chaque version du noyau (et en particulier ceux de la dernière version) sur différents sites internet. Le site suivant

`http ://www.kernel.org`

collectionne en particulier les différentes versions du noyau ; un mirroir français est accessible à `http ://www.fr.kernel.org`. Le site `http ://www.lip6.fr` est également une source intéressante.

1.1.2 Les différentes versions du noyau Linux

Chaque version du noyau Linux est identifiée par un numéro composé de trois entiers notés de la façon suivante :

`X.Y.Z`

X et Y constituent le *numéro de version* ; un changement de valeur de X dénote de profondes évolutions dans le noyau alors qu'un changement de valeur de Y traduit quand à lui des modifications moins significatives. Les valeurs impaires de Y désignent des versions de développement du noyau, c'est-à-dire des versions parfois instables et susceptibles d'évoluer de façon imprévisible ; elles sont donc à déconseiller à l'utilisateur final. Les valeurs paires désignent au contraire des versions stables et qui n'évolueront plus (à par l'application d'éventuelles corrections).

Z est appelé *numéro de parution* ; ses changements dénotent des évolutions mineures (généralement des corrections d'erreurs) lorsque Y est pair, et de nouvelles versions intermédiaires (éventuellement accompagnées de nombreuses évolutions) lorsque Y est impair.

1.1.3 Compiler le noyau Linux

Les sources du noyau Linux sont composées de plusieurs milliers de fichiers répartis sur de nombreux répertoires et occupant quelques dizaines de méga-octets. Aussi chaque nouvelle version est-elle généralement fournie dans un fichier d'archive compressé. Il est donc nécessaire d'en extraire les fichiers dans un répertoire choisi, traditionnellement `/usr/src`.

Ainsi si vous avez obtenu le fichier `linux-2.4.4.tar.bz2`, qui contient les sources de Linux 2.4.4, et si vous l'avez déposé dans le répertoire `/archives/sources`, vous utiliserez par exemple les commandes suivantes

```
# cd /usr/src
# bunzip2 < /archives/sources/linux-2.4.4.tar.bz2 | tar xf -
# cd linux
# make mrproper
#
```

Vous obtenez ainsi une arborescence contenant les sources du noyau et expurgée des fichiers inutiles car pouvant être générés (c'est ce à quoi sert la dernière commande).

Notons qu'il est parfois intéressant de faire cohabiter les sources de différentes versions du noyau sur un même système ; il peut être utile, pour cela, de placer chaque arborescence dans un répertoire différent (`linux-x.y.z` par exemple), et de créer un lien symbolique `linux` pointant vers la version en cours d'utilisation. Attention dans un tel cas, lors de l'extraction des fichiers d'une nouvelle version, à ne pas écraser ceux en place.

Il est également important, lors de la première installation des sources du noyau, de s'assurer que les sous-répertoires `linux` et `asm` du répertoire `/usr/include`

soient bien des liens vers les sous-répertoires de mêmes noms du répertoire `/usr-/src/linux/include`. Ces deux sous-répertoires ont en effet pût être créés, lors de l'installation du système, comme de vrais répertoires, autonomes. Dans ce cas, lors de l'installation des sources d'un nouveau noyau, il risque de ne plus y avoir de cohérence entre les fichiers inclus et le noyau. Ceci ne pose aucun problème pour la compilation de ce dernier, mais peut en poser lors de la compilation d'applications utilisant certaines fonctionnalités de bas niveau.

Configuration

Une fois installé, le noyau doit être configuré avant d'être compilé. Le but de cette configuration est de choisir les caractéristiques fondamentales du noyau, celles liées à l'architecture sur laquelle il doit s'exécuter (type et nombre des processeurs, nature des bus, ...). La configuraion permet également de spécifier les éléments qui doivent être intégrés directement dans le noyau, ceux qui doivent être compilés sous forme de module, et ceux qui ne doivent pas être compilés, car inutiles.

La configuration s'effectue de façon interactive grâce à des scripts intégrés dans l'arborescence du noyau. L'interface la plus élémentaire est obtenue par la commande suivante

```
# make config
rm -f include/asm
( cd include; ln -sf asm-i386 asm)
/bin/sh scripts/Configure arch/i386/config.in
#
# Using defaults found in .config
#
*
* Code maturity level options
*
Prompt for development and/or incomplete code/drivers
(CONFIG_EXPERIMENTAL) [Y/n/?]
```

L'utilisateur doit alors répondre à une longue série de questions. Ceci peut rapidement se révéler fastidieux, mais fort heureusement, les valeurs par défaut sont souvent satisfaisantes, en particulier grâce au fait qu'elles sont sauvegardées d'une compilation à l'autre (dans le fichier `.config`).

Lors d'une mise à jour du noyau par le biais d'un patch (voir un peu plus loin), il est important de relancer la configuration afin de prendre en compte les éventuels nouveaux paramètres. La commande

```
# make oldconfig
```

permet de ne s'intéresser qu'à ces nouveaux paramètres, les autres conservant leur valeur sauvée dans le fichier `.config` lors de la dernière configuration.

Notons que ce type de configuration, basée sur des scripts, nécessite le shell `bash`.

Il est cependant possible de profiter d'interfaces plus évoluées pour configurer le noyau. La première est obtenue par la commande suivante :

```
# make menuconfig
```

Elle fonctionne entièrement en mode texte et permet, par le biais de menus et boutons radio, de parcourir les options d'une façon moins linéaire que la configuration de base. Ceci permet, lors d'une nouvelle configuration, d'accéder rapidement aux éléments que l'on souhaite changer.

Elle permet, de plus, de sauvegarder la configuration dans un fichier différent du classique `.config`, ce qui peut se révéler pratique pour faire cohabiter au sein de la même arborescence des configurations différentes.

Une dernière possibilité est offerte par la commande suivante :

make xconfig

Elle offre les mêmes possibilités que la précédente, mais fonctionne en mode graphique (sous X windows, donc).

Comme vous l'aurez remarqué, il est *a priori* inutile de modifier le fichier `Makefile` du répertoire `/usr/src/linux`, mais il peut être intéressant de noter que ce fichier définit notamment une variable `EXTRAVERSION` qui peut être utilisée pour différencier deux compilations d'une même version du noyau. L'utilisation des modules rend généralement une telle pratique inutile, mais elle peut se révéler intéressante lors de l'application de patches non officiels.

Il suffit donc de donner une valeur à cette variable dans le fichier `Makefile`, par exemple :

```
EXTRAVERSION=-test
```

On obtiendra alors, lors de l'exécution de la commande `uname` sur le noyau ainsi compilé :

```
# uname -a
Linux albert 2.4.3-test #1 Mon Avr 30 13 :16 :08 CEST 2001 i586
unknown
#
```

Pour les versions du noyau antérieures à la 2.6, lorsque la configuration est terminée, les dépendances entre les fichiers doivent être générées de la façon suivante :

make dep

À partir du 2.6, cette commande n'existe plus car il n'est plus nécessaire de générer explicitement de fichier de dépendances.

Compilation et installation

La compilation du noyau est alors réalisée de la façon suivante :

make vmlinux

Avant de pouvoir redémarrer le système sous la nouvelle version du noyau ainsi compilée, il est nécessaire de compiler les modules si leur utilisation a été activée lors de la configuration de la compilation du noyau.

Cela peut être réaliser par la commande suivante :

make modules

Notons que la commande

make all

ou tout simplement

make

réalise directement les deux opérations précédentes.

Les fichiers nécessaires au démarrage du système peuvent alors être installés dans le répertoire voulu et le chargeur utilisé (généralement LILO ou GRUB) en être informé. Ces opérations sont dépendantes de la distribution installée sur le système. La commande suivante permet d'automatiser cette installation :

make install

Attention cependant à s'assurer que ce que réalise cette commande est cohérent avec les choix faits pour la configuration du démarrage du système. Sur une distri-

bution correctement installée, le binaire invoqué (`installkernel`) doit a priori réaliser les bonnes opérations.

Enfin, sachez qu'à partir de Linux 2.6, la commande suivante vous donne l'ensemble des cibles potentielles de `make`.

```
# make help
```

Outils nécessaires

La compilation et la configuration dynamique d'une version du noyau nécessite certains outils logiciels. Ils sont généralement disponibles sur toute distribution, au moins dans sa version "développeur" ou équivalent.

Il faut cependant prendre garde aux versions de ces outils, en particulier lorsque l'on décide de compiler une version récente du noyau ; en effet ces outils évoluent au rythme des versions du noyau.

Le fichier `Documentation/Changes` donne la liste des principaux outils ainsi que les versions recommandées pour la version du noyau considéré.

1.1.4 Appliquer des patchs

Plutôt qu'installer systématiquement chaque nouvelle version, il peut être intéressant d'appliquer des *patchs*, en particulier pour l'utilisateur qui souhaite se tenir à jour des dernières évolutions d'une version de développement du noyau.

Chaque nouvelle parution d'une version est généralement disponible sous forme de fichier de patchs (en plus de l'archive dont nous avons parlé). Un fichier de patch contient en fait les différences entre une version de parution et la suivante. La commande `patch` permet alors de construire la nouvelle version de parution à partir du fichier de patchs et de la version de parution déjà installée.

Application manuelle

Imaginons que la version `2.6.0-test6` soit installée dans le répertoire `/usr/src/linux` et que les patches soient dans le répertoire `/archives` sous forme compressée par `bzip2`, alors vous utiliserez la commande suivante :

```
# cd /usr/src/linux
# bunzip2 < /archives/patch-2.6.0-test7.bz2 — patch -p1
```

La commande `bunzip2` sera éventuellement à remplacer par `gunzip` ou autre selon le format de compression éventuelle du patch. Attention, jusqu'à Linux 2.4, l'option `-p1` de la commande `patch` est à remplacer par `-p0`.

Automatisation

Un script, nommé `patch-kernel`, se trouve dans l'arborescence des fichiers sources du noyau. Il se charge d'identifier la version du noyau actuellement installée et de trouver dans le répertoire courant les fichiers de patchs applicables. Il applique alors la commande `patch` afin de mettre à jour les fichiers sources.

Ainsi, si le répertoire `/archives/patchs` contient les fichiers de patchs du noyau, on utilisera les commandes suivantes :

```
# cd /archives/patchs
# /usr/src/linux/scripts/patch-kernel
#
```

L'option `help` de cette commande permet d'en connaitre les différents paramètres.

1.2 Programmation dans le noyau

L'écriture de code devant être intégré dans le noyau n'est pas fondamentatement différente de celle d'un autre code. Elle présente pourtant certaines spécificités qu'il est bon de ne jamais oublier.

Une des premières différences est essentiellement superficielle, et bien qu'étant la plus visible et parfois la plus prenante, elle reste la plus simple à gérer. Il s'agit bien sûr de l'absence de la librairie de programmation classique, la célèbre `libc` qui implante toutes les fonctions de base auquel le programmeur ne fait même plus attention tellement il les utilise.

1.2.1 Les fonctions élémentaires

L'affichage

Le premier programme qu'écrit un programmeur lorsqu'il découvre un nouvel environnement consiste généralement en l'affichage d'un message de type "*Hello world !*". En ce qui concerne la programmation noyau, la fonction `printf()` n'est pas disponible, elle doit être remplacée par la fonction `printk()` qui présente la même interface :

```
int printk(const char * fmt, ...);
```

Cette fonction est définie dans `include/linux/kernel.h` et codée dans `kernel/printk.c`.

En plus des paramètres classiques de `printf()`, cette fonction accepte un premier paramètre supplémentaire optionnel permettant d'aiguiller l'affichage du message. Nous en reparlerons lorsque nous aborderons les techniqus de débogage en 1.5.2.

Un message affiché de façon classique, sans l'utilisation de ce premier paramètre optionnel, sera envoyé dans les "logs " du système et pourra par exemple être visualisé par le biais de la commande `dmesg`.

1.2.2 La gestion de la mémoire

Les fonctions`malloc()` et `free()` de la librairie C ne sont donc pas disponibles en mode noyau, elles non plus.

Les fonctions suivantes, définies dans `linux/slab.h`, permettent cependant de les remplacer dans de nombreux cas "élémentaires" :

```
void * kmalloc(size_t nb_bytes, int priority);
void kfree(const void * adress);
```

L'utilisation de ces deux fonctions est sans grande surprise pour le programmeur C, à l'exception d'un petit "détail" : le second paramètre de la fonction d'allocation mémoire.

La gestion de la mémoire au sein d'un système d'exploitation moderne tel que Linux est cependant bien plus complexe et ne peut être décrite entièrement ici. Elle fera donc l'objet d'un chapitre spécifique.

1.2.3 Manipulation de blocs mémoire

Le fichier `linux/string.h` définit différentes fonctions de manipulation de zones mémoire. Une version par défaut de ces fonctions est implantée dans le fichier `lib/string.c`, mais une implantation optimisée pour l'architecture spécifique du système peut être définie dans le répertoire dédié.

Nous ne décrirons pas ici ces fonctions, nous nous contenterons de les énumérer. Elles ont en effet la même interface et fournissent le même service que leurs équivalent

de la librairic C. Vous trouverez donc toutes les informations utiles en consultant les pages de manuel de ces dernières.

La première liste que voici contient des fonctions permettant la manipulation (copie, comparaison, ...) de chaines de caractères :

```
char * strpbrk(const char *,const char *);
char * strtok(char *,const char *);
char * strsep(char **,const char *);
__kernel_size_t strspn(const char *,const char *);
char * strcpy(char *,const char *);
char * strncpy(char *,const char *, __kernel_size_t);
char * strcat(char *, const char *);
char * strncat(char *, const char *, __kernel_size_t);
int strcmp(const char *,const char *);
int strncmp(const char *,const char *,__kernel_size_t);
int strnicmp(const char *, const char *, __kernel_size_t);
char * strchr(const char *,int);
char * strrchr(const char *,int);
char * strstr(const char *,const char *);
__kernel_size_t strlen(const char *);
__kernel_size_t strnlen(const char *,__kernel_size_t);
```

Les fonctions suivantes permettent quand à elles d'initialiser, comparer, copier, ... des zones de mémoire quelconque ; elles aussi ont des équivalents dans la librairie C.

```
void * memset(void *,int,__kernel_size_t);
void * memcpy(void *,const void *,__kernel_size_t);
void * memmove(void *,const void *,__kernel_size_t);
void * memscan(void *,int,__kernel_size_t);
int memcmp(const void *,const void *,__kernel_size_t);
void * memchr(const void *,int,__kernel_size_t);
```

1.2.4 La synchronisation

L'espace d'adressage du noyau est unique et partagé par l'ensemble des processus (lorsqu'ils s'exécutent en mode noyau, bien entendu). Toutes les ressources peuvent donc être directement partagées entre de nombreux flux d'exécution sans autre forme de procés. Une telle facilité de partage présente cependant un inconvénient : la nécessité de synchroniser les accès à de telles ressources.

Les outils permettant de réaliser une telle synchronisation au cœur du noyau Linux sont trop nombreux et variés pour les décrire exhaustivement ici, et le chapitre 4 leur est consacré.

1.2.5 Les listes doublement chaînées

Le fichier `linux/list.h` définit et implante (sous forme d'un ensemble de fonctions *inline*) une structure de liste doublement chaînée.

Le programmeur rigoureux que vous êtes probablement risque d'être quelque peu choqué par la structure définissant un maillon de la liste (et, tant qu'à faire, toute la liste !). Celle-ci ne contient en effet aucun champ permettant de référencer l'élément contenu dans la liste !

C'est en fait au contraire la structure de données définissant les objets devant être contenus dans la liste qui doit intégrer un champ pointant sur le maillon qui le rattache à la chaîne ! Le monde à l'envers en quelque sorte ...

Qui plus est, il est absolument nécessaire que ce champ soit le premier de la structure, afin d'avoir la même adresse. C'est en effet uniquement cette égalité des adresses entre la structure des données et le maillon de la liste chaînée qui nous permet de faire le lien entre eux.

Déclaration et initialisation

Le type d'une liste doublement chaînée est déclaré de la façon suivante :

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

La déclaration d'une liste `ma_liste` est cependant généralement réalisée de la façon suivante :

```
LIST_HEAD(name);
```

Cette macro permet en effet d'initialiser correctement les champs de la structure définissant la chaîne.

Insertion

Les deux fonctions suivantes permettent d'insérer l'élément `new` dans la liste définie par `head` :

```
void list_add(struct list_head *new, struct list_head *head);
void list_add_tail(struct list_head *new, struct list_head *head);
```

1.2.6 Règles d'écriture

La programmation au sein du noyau Linux a ses règles, y compris en ce qui concerne la forme qu'elle doit prendre. Bien sûr, ces règles n'ont rien d'impératif, et il est tout à fait possible, techniquement parlant, de les ignorer. Il est cependant préférable de les respecter au maximum, afin d'assurer la cohérence du code, même au niveau de l'écriture. Cette cohérence en facilite en effet grandement la lecture et la compréhension.

Je ne décrirai pas ces règles ici, elles sont consignées dans le fichier `Documentation/CodingStyle` des sources du noyau. Je vous invite à les consulter rapidement.

1.2.7 Copie de données

Les fonctions présentées dans la section 1.2.3 permettent de déplacer, copier, comparer des zones de mémoires situées dans l'espace d'adressage du noyau.

Il est parfois également nécessaire de copier des données vers ou depuis l'espace utilisateur, par exemple dans le code implantant un appel système. Il faut alors être particulièrement attentif à la légitimité de tous les accès réalisés, car la moindre erreur peut mettre en péril le système.

Lors de l'invocation d'un appel système, le processus utilisateur peut en effet, volontairement ou non, passer aux système des données non valides ; il appartient alors à ce dernier de se protéger contre un tel comportement.

Un premier type de vérification consiste à s'assurer de la validité des données elles-mêmes ; un telle vérification ne présente guère d'originalité, puisqu'il s'agit simplement de s'assurer que la valeur des paramètres passés à une fonction est bien conforme aux exigences de cette fonction. Ce type de vérification s'applique aux données passées par valeur. Linux ne propose aucun mécanisme pour aider une telle vérification puisqu'il s'agit là simplement de mettre en place des stratégies efficaces de programmation défensive.

Le second type de vérification s'applique au contraire aux paramètres passés par adresses[1] pour lesquels il est important de s'assurer que l'utilisation de l'adresse

[1] Même si en C le passage de paramètres par adresse n'existe pas, c'est comme cela que nous appellerons le passage d'un paramètre de type pointeur.

en question est bien légitime pour le processus appelant. Supposons par exemple qu'un processus puisse passer une adresse de buffer quelconque à l'appel système `write()`, il pourrait alors sauvergarder dans un fichier le contenu de la mémoire d'un autre processus, ce qui pourrait éventuellement lui permettre d'obtenir des informations confidentielles.

Le noyau Linux fournit différentes fonctions et macros permettant de mettre en place de telles vérifications afin d'éviter de corrompre l'intégrité du noyau ; elles sont définies dans `asm/uaccess.h`.

Contrôle des pointeurs utilisateur La fonction suivante permet de s'assurer de la validité d'une zone mémoire définie par un pointeur :

```
int access_ok(int type, void * addr,int size);
```

Le paramètre `addr` désigne, naturellement, l'adresse de la zone à vérifier, et `size` sa taille. Le paramètre `type` peut prendre les valeurs `VERIFY_READ` ou `VERIFY_WRITE` selon que l'on souhaite vérifier un accès en lecture ou un accès en lecture/écriture.

Notons que cette fonction n'assure pas qu'un transfert de données peut être réalisé sans erreur, mais simplement que la zone est bien dans l'espace d'adressage du processus.

Lecture/écriture d'une variable La copie d'une variable unique depuis ou vers l'espace utilisateur peut être réalisé à l'aide de l'une des fonctions suivantes :

```
put_user(x,ptr);
__put_user(x,ptr);
get_user(x,ptr);
__get_user(x,ptr);
```

Nous ne décrivons pas ces macros comme des fonctions, car il est impossible de définir leur prototype de façon réaliste. `x` est la variable, et `ptr` est l'adresse dans l'espace utilisateur d'une variable du même type que `x`.

Les versions préfixées par __ réalisent moins de vérifications (elles ne réalisent pas d'appel à `access_ok`).

Ces fonctions renvoient 0 en cas de succès, `-EFAULT` en cas d'erreur.

```
unsigned long copy_from_user(void *to, const void __user *from, unsigned long n);
unsigned long copy_to_user(void __user *to, const void *from, unsigned long n);
```

Il existe, ici aussi, une version préfixée par __ de chacune de ces fonctions, qui réalise la même action, mais sans appel préalable à `access_ok()`.

Imaginons alors une fonction invoquée par un appel système et chargée de transférer des données vers l'espace utilisateur, ces données provenant d'un périphérique, d'un buffer réseau ou de tout autre emplacement sous la responsabilité du noyau. Le code suivant, apparemment correcte en mode utilisateur ne l'est plus en mode noyau :

```
int lireDonnees(char * bufferUtilisateur, int tailleBU)
{
   tailleBU = min(tailleBU, tailleBS);
   memcopy(bufferUtilisateur, bufferSysteme, tailleBufferU);

   return tailleBU;
}
```

Il faut alors réécrire cette fonction de la façon suivante, par exemple :

```
int lireDonnees(char * bufferUtilisateur, int tailleBU)
{
   tailleBU = min(tailleBU, tailleBS);
```

```
    if (!access_ok(VERIFY_WRITE, bufferUtilisateur, tailleBU)) {
       return -EINVAL;
    }

    copy_to_user(bufferUtilisateur, bufferSysteme, tailleBufferU);

    return tailleBU;
}
```

Naturellement, si le buffer doit être à nouveau utilisé avant de revenir en mode utilisateur, il est inutile de refaire le test d'accessibilité.

1.3 Les modules $Revision$: 1.5

Les modules permettent à Linux d'offrir à l'administrateur une grande souplesse dans la configuration du noyau. Le but est en effet d'ajouter ou de supprimer dynamiquement des parties de code dans un noyau en cours d'exécution. Les avantages d'une telle possibilité sont multiples :

- L'utilisation de la mémoire par le noyau est optimisée, puisqu'un module ne sera présent au cœur du noyau que s'il est vraiment nécessaire, il n'occupera pas de place en mémoire s'il est inutile.
- La stabilité et la sécurité du noyau sont accrues.
- L'architecture du noyau est nécessairement moins monolithique, les interfaces entre les différents sous-systèmes sont donc mieux définies.
- Différentes parties du noyau peuvent être compilées indépendamment les unes des autres ; ceci permet par exemple à un constructeur de fournir les pilotes de ses périphériques sous forme compilée s'il ne souhaite pas en diffuser les sources.

Nous allons voir dans cette section comment compiler et utiliser les modules sous Linux.

1.3.1 Compilation et installation

Lors de la configuration de la compilation du noyau, l'utilisation des modules est validée par l'option "*Enable loadable module support*" qui positionne la variable `CONFIG_MODULES`. C'est généralement une bonne idée d'activer cette option.

Il est parfois inutile d'utiliser les modules, par exemple sur une machine ne disposant que de peu de périphériques différents et faisant un usage régulier de ces périphériques. Ainsi le noyau d'un routeur sera compilé "statiquement", c'est-à-dire sans l'utilisation de modules.

La compilation des modules dont le code source fait partie de l'arborescence du noyau sont compilés en exécutant la commande suivante dans le répertoire `/usr/src/linux` :

```
# make modules
```

Les modules sont ensuite installés en exécutant

```
# make modules_install
```

Cette commande place alors les modules compilés dans une arborescence située dans le répertoire `/lib/modules/`*`version`* où *`version`* représente le numéro de version du noyau (tel que donné par la commande `uname -r`).

1.3.2 Insertion et suppression

Les commandes `insmod` et `modprobe` permettent d'insérer un nouveau module dans le noyau en cours d'exécution, la commande `rmmod` permet ensuite de le supprimer (s'il n'est plus utilisé), la commande `lsmod` permet enfin de lister les modules actuellement chargés dans le noyau. Bien entendu, seul l'administrateur doit pouvoir utiliser les commandes `insmod`, `modprobe` et `rmmod`.

Ces commandes font partie d'un paquetage nommé *modutils*, qui peut être obtenu de la même façon que le noyau. Il est important de veiller à la cohérence entre la version du noyau et la version des modutils. Le fichier `Documentation/Changes` des sources du noyau précise la version recommandée.

Une fois ce paquetage correctement compilé et installé, un module peut être chargé de la façon suivante :

```
# insmod 3c59x
Using /lib/modules/2.2.17-21mdk/net/3c59x.o
#
```

Ici, le module contenant le pilote d'une carte réseau est chargé sur un système Linux 2.2.17.

La commande `modprobe` permet quant à elle de prendre en compte les dépendances entre les modules. Elle se charge pour cela d'installer dans le noyau en cours d'exécution non seulement le module souhaité, mais également tous ceux dont il dépend. Ainsi le chargement du module `lp` donne par exemple le résultat suivant :

```
# modprobe lp
# lsmod
Module                  Size  Used by
parport_probe           3536   0  (autoclean)
parport_pc              7568   1  (autoclean)
lp                      5552   0  (unused)
parport                 7744   1  [parport_probe parport_pc lp]
#
```

Nous pouvons observer ici que quatre modules ont été insérés automatiquement dans le noyau.

Les dépendances entre modules sont décrites dans un fichier `modules.dep` situé dans le répertoire contenant les modules (c'est-à-dire `/lib/modules/<version>`)

Notons enfin qu'il est possible de passer des paramètres à un module lors de son insertion dans le noyau, par exemple de la façon suivante :

```
# insmod ne io=0x300
```

Le nom et la signification des paramètres dépendent bien sûr du module, il est donc impossible de les lister ici.

1.3.3 Dépendances entre modules

Les dépendances entre modules sont gérées par le biais d'un fichier nommé `modules.dep` et situé dans le répertoire contenant la version compilée des modules ; ce fichier est au format `Makefile`, et il est généré par la commande `depmod`. Cette commande doit donc être utilisée (avec l'option `-a`) lors de la compilation

d'un nouveau noyau ou de l'ajout d'un module dans l'arborescence ; ceci est réalisé automatiquement lors de l'utilisation de la commande `make modules_install`.

1.3.4 Configuration

Le chargement des modules peut être configuré par le fichier `/etc/modules.conf`. Il serait hors de propos de donner ici une description exhaustive du format de ce fichier, nous vous conseillons pour cela d'en consulter la page de manuel. Décrivons tout de même brièvement les principaux éléments de ce fichier.

Il s'agit, comme souvent sous Linux, d'un fichier texte orienté ligne, dans lequel un commentaire est introduit par un caractère `#` et se poursuit jusqu'à la fin de la ligne.

Chaque ligne peut être notamment d'un des types suivants

`alias nom_alias résultat` qui définit `nom_alias` comme un alias pour `résultat`, ce qui permet par exemple de définir un nom plus générique comme `sound` pour un module réel comme `opl3sa2` ;

`path[TAG]=chemin` permet de définir le répertoire (identifié par le paramètre `chemin`) contenant les modules d'un type (indentifié par `TAG`) ;

`keep` permet de conserver la configuration actuelle de sorte qu'une définition suivante de `path` ait pour effet d'ajouter le chemin et non de le remplacer ;

`pre-install module commande` permet de définir une commande à exécuter avant le chargement d'un module ;

`post-install module commande` permet de définir une commande à exécuter après le chargement d'un module ;

`pre-remove module commande` permet de définir une commande à exécuter avant le déchargement d'un module ;

`post-remove module commande` permet de définir une commande à exécuter après le déchargement d'un module ;

`option module options` permet de définir des options devant être passées à un module lors de son chargement.

Cette liste n'est pas complète, mais elle donne un aperçu de ce qu'il est possible de configurer avec ce fichier. Ajoutons simplement qu'il est possible de définir des variables, d'utiliser des conditions (`if ...`) ainsi que le résultat de l'exécution d'une commande.

Voici par exemple comment nous pouvons ajouter le sous-répertoire `devel` du répertoire des modules dans la liste des répertoires contenant les modules divers :

```
# Nous placerons les modules en cours de développement ici :
keep
path[misc]=/lib/modules/`uname -r`/devel
```

Voici comment une carte son peut être déclarée au près des outils du paquetage *modutils* :

```
# Configuration de la carte son
alias sound opl3sa2
alias midi mpu401
options opl3sa2 io=0x370 mss_io=0x530 mpu_io=0x330 irq=5 dma=0 dma2=7
```

1.3.5 Chargement à la demande

Une fonctionnalité particulièrement intéressante de la gestion des modules sous Linux est la possibilité d'automatiser le chargement et le déchargement des modules à la demande. Cela signifie que l'administrateur n'a plus besoin d'utiliser explicitement les commandes `insmod` et `rmmod`, car elles seront automatiquement invoquées chaque fois que nécessaire.

Pour activer cette fonctionnalité, il faut valider l'utilisation de `kmod` lors de la compilation du noyau. C'est en effet cet utilitaire (exécuté sous forme de thread noyau) qui s'occupe de charger les modules lorsque nécessaire ; il invoque pour cela la commande `modprode` dont le chemin est définit dans `/proc/sys/kernel/modprobe`.

Il suffit alors qu'un processus cherche à utiliser une partie du noyau gérée par un module pour que celui-ci soit chargé. Ainsi une requète d'impression pourra avoir pour effet de charger le module `lp` (et éventuellement ceux dont il dépend) afin que le démon d'impression puisse accéder au matériel correspondant. De même l'utilisation de la commande `ifconfig` déclenchera le chargement du module de gestion de la carte réseau avant de pouvoir la configurer.

*

1.4 Écriture d'un module $Revision$: 1.6

La construction d'un module peut se révéler particulièrement utile lors de l'écriture de code devant être intégré au noyau. Un module permet en effet d'accélerer le cycle modifications-compilation-test par la possibilité qu'il offre d'insérer le code dans un noyau en cours d'exécution sans même le redémarrer. L'écriture d'un module présente également l'intérèt de forcer le programmeur à avoir une certaine rigueur dans l'écriture de son code, et en particulier dans la définition de l'interface avec les autres éléments du noyau.

Nous allons étudier dans cette section comment écrire un module, sans contraintes particulières sur ce qu'il doit réaliser ; le but étant de pouvoir réutiliser la majorité de ce code dans les différents exemples que nous donnerons dans le reste de ce livre.

1.4.1 Quelle version du noyau ?

Si vous souhaitez écrire du code qui puisse être compilé pour différentes version du noyau Linux, vous devrez prendre garde à vous adapter aux petites spécificités de chaque version grâce à l'utilisation des macros suivantes, définies dans `linux/version.h`.

La macro *KERNEL_VERSION*(*a*,*b*,*c*) converti un numéro de version, exprimé sous la forme de trois entiers correspondant aux trois champs de la numérotation classique de version de Linux, en un entier unique.

La macro *LINUX_VERSION_CODE* donne la représentation sous forme d'un entier unique du numéro de version des sources du noyau.

Il est donc possible d'utiliser ces macros pour compiler telle ou telle portion de code en fonction du noyau cible :

```
#if LINUX_VERSION_CODE < KERNEL_VERSION(2, 4, 0)
#   error Nécessite au moins la version 2.4 du noyau
#elif LINUX_VERSION_CODE < KERNEL_VERSION(2, 6, 0)
  // Définitions spécifiques à la version 2.4
#else
  // Définitions spécifiques à la version 2.6
#endif
```

1.4.2 Initialisation

La première chose que doit réaliser un module est son identification auprès du noyau. Cela est réalisé simplement par la définition d'une fonction d'initialisation au cœur du module.

Cette fonction doit avoir l'interface suivante :

```
int initialisation_de_mon_module(void)
{
    /* Phase d'initialisation */
    ...

    /* Terminaison sans échec */
    return 0;
}
```

Elle doit ensuite être définie comme fonction d'initialisation de la façon suivante

```
module_init(initialisation_de_mon_module);
```

La fonction `module_init` est définie dans le fichier `init.h`, qui doit donc être inclus par le module.

Notons qu'avant Linux 2.4, la fonction d'initialisation d'un module devait nécéssairement être nommée `init_module` et n'avait pas à être déclarée par la fonction `module_init`.

Le code d'initialisation figurant dans la fonction d'initialisation est bien sûr extrèmement dépendant du type du module. Ainsi un module intégrant un pilote de périphérique se chargera d'initialiser le matériel, alors qu'un module définissant un nouvel appel système devra modifier la table des appels système. Nous verrons des exemples spécifiques dans les prochains chapitres.

1.4.3 Terminaison d'un module

Tout comme la phase d'initialisation, la phase de terminaison d'un module est très liée aux objectifs du module, et elle est réalisée de la même façon par une fonction unique, dont le prototype est le suivant :

```
void cleanup_module(void)
{
    /* Libération des éventuelles ressources */
    ...
}
```

Comme vous pouvez le constater, l'interface est extrèmement simple ; toute la difficulté viendra de ce que doit réaliser cette fonction.

De la même façon que pour la fonction d'initialisation, le nom sera défini par un appel à la fonction `module_exit` à partir de Linux 2.4.

1.4.4 Passage de paramètres

Il est possible, lors de l'insertion d'un module dans le noyau, de passer des paramètres à ce module, comme nous l'avons vu dans la section 1.3.2.

Linux 2.4

Lors de la réalisation d'un module, quelques macros (définies dans `linux/module.h`) permettent de déclarer les paramètres qu'il accepte :

`MODULE_PARM(nom, type)` permet de définir le nom et le type d'un paramètre. Le type est une chaîne de caractères de la forme ``[min[-max]]b,h,i,l,s'', où `min` et `max` peuvent définir les valeurs extrèmes et où `b,h,i,l` ou `s` désigne un type octet, entier court, entier, entier long ou chaîne de caractères, respectivement.

`MODULE_PARM_DESC(nomParam, description)` permet d'associer une description textuelle (une chaîne de caractères) à un paramètre.

Ces informations sont elles aussi visualisables par la commande `modinfo`.

Linux 2.6

Sous Linux 2.6, la gestion des paramètres d'un module est un plus évoluée. Elle est mise en œuvre par un ensemble de macros définies dans le fichier `linux/moduleparam.h`.

la première de ces macros, et la plus importante, permet de définir un paramètre et de lui associer un type :

```
module_param(name, type, perm);
```

Les paramètres ont ici la signification suivante :

name est, bien sûr, le nom du paramètre ;

type donne le type du paramètre, les valeurs possibles sont *byte*, **short**, *ushort*, **int**, *uint*, **long**, *ulong*, *charp*, *bool* et *invbool* ;

perm doit être, pour l'instant, laissé à 0 ; le but est de définir les permissions caractérisant l'intégration de ce paramètre dans le système de fichiers *sysfs*.

Il est également possible de rendre public le paramètre sous un nom différent de la variable locale qui le contient, grâce à la macro suivante :

```
module_param_named(name, value, type, perm);
```

Les paramètres de cette macro sont les mêmes que ceux de *module_param*, et *value* est le nom de la variable locale.

1.4.5 Informations associées à un module

Quelques macros permettent d'ajouter certaines informations au module, elles sont définies dans `linux/module.h`, ce sont :

`MODULE_ALIAS(nom)` permet de définir un *alias* qui peut être utilisé pur nommer le module ;

`MODULE_AUTHOR(nom)` permet de définir le nom de l'auteur du module ;

`MODULE_LICENSE(license)` définit la license selon laquelle ce module est fourni ;

`MODULE_DESCRIPTION(description)` permet de donner une brève description du module ;

`MODULE_SUPPORTED_DEVICE(nomPériphérique)` permet de spécifier le nom symbolique du ou des périphériques pris en charge par le module (s'il s'agit d'un pilote). Cette dernière macro n'est pas encore réellement implantée.

Leur utilisation n'est en rien obligatoire lors de l'écriture d'un module, mais peut s'avérer fort utile. Ces informations sont visualisables par la commande `modinfo`.

Une de ces macros est cependant assez importante, il s'agit de la définition de license. Les différentes valeurs pouvant être passées en paramètre sont les suivantes :

`"GPL"` si le module est fourni selon la license GPL (v2 ou suivante) ;

`"GPL"` si le module est fourni selon la license GPL v2 ;

`"GPL and additional rights"` si le module est fourni selon une license accordant davantage de droits que la license GPL v2 ;

`"Dual BSD/GPL"` si le module est fourni selon une license double BSD / GPL ;

`"Dual MPL/GPL"` si le module est fourni selon une license double MPL / GPL ;

`"Proprietary"` si le module est fourni selon une license propriétaire.

Le but de la définition d'une telle license n'est pas d'empêcher ou de conditionner le fonctionnnement d'un module en fonction de la license qu'il arbore, mais de permettre à chacun de ne se soucier des rapports de bogue que s'ils le concernent réellement.

1.4.6 Un exemple

Le but de cette section est simplement de donner un exemple complet de module, et ce sans contrainte particulière sur ce que fait ce module. Ainsi le module que nous allons écrire ici ne présente aucun intéret dans son utilisation. Nous vous invitons, en revanche, à vous inspirer de ce code pour l'écriture de vos propres modules.

Version pour Linux 2.4

Voici donc le code complet d'un module élémentaire, qui se contente d'afficher un message lorsqu'il est chargé, et un autre message lorsqu'il est supprimé :

```
/*
 *    Écriture d'un module permettant d'afficher un message dans les
 * logs.
 *    Version pour Linux 2.4
 *                                        (C) Emmanuel Chaput 2000-2003
 */

/* Ce code ne peut être compilé que sous forme de module */
#define MODULE

#include <linux/version.h>

#if (LINUX_VERSION_CODE < KERNEL_VERSION(2, 4, 0)) || (LINUX_VERSION_CODE >=  KERNEL_VERSION(2, 6, 0))
#   error "Ce module nécessite Linux 2.4"
#endif

#ifdef MODULE
/* En cas d'utilisation d'informations de version */
#   ifdef MODVERSIONS
#      include <linux/modversions.h>
#   endif
#   include <linux/module.h>
#endif

#include <linux/kernel.h>    /* printk() */
#include <linux/init.h>      /* module_init() */


/* Quelques définitions à titre d'info */
MODULE_AUTHOR("Emmanuel Chaput (chaput@enseeiht.fr)");
MODULE_DESCRIPTION("Un module sans intéret ...");
MODULE_LICENSE("GPL");
```

```
/* Définition des paramètres */
static int niveauLog = 0;
MODULE_PARM(niveauLog, "0-7i");
MODULE_PARM_DESC(niveauLog, "Niveau d'affichage des messages");

/*
 * Initialisation du module
 */
static int init_echo(void)
{
    printk("<%d>Module echo en place.\n", niveauLog);

    /* Tout s'est bien passé */
    return 0;
}

static void exit_echo(void)
{
    printk("<%d>Module echo supprimé.\n", niveauLog);
}

module_init(init_echo);
module_exit(exit_echo);

/*
 * $Log: echo-2.4.c,v $
 * Revision 1.1  2003/10/31 09:40:08  manu
 * Début de mise à jour du chapitre compilation pour la 2.6
 *
 */
```

Comme vous pouvez le constater, les deux fonctions sont ici réduites à leur plus simple expression.

Compilation Voici le fichier `Makefile` permettant de compiler ce module élémentaire sous la version 2.4 du système :

```
# Les fichiers include du noyau, à changer si l'installation ne
# correspond pas au noyau cible
INCLUDE_NOYAU = /usr/src/linux/include

# Les paramètres du compilateur
CFLAGS += -Wall -I$(INCLUDE_NOYAU) -I.

# Les paramètres pour la compilations d'éléments du noyau
KCFLAGS = $(CFLAGS) -DMODVERSIONS -D__KERNEL__

# Lecture de la version du noyau
KERNEL_VERSION = $(shell awk -F\" '/UTS_RELEASE/ {print $$2}' $(INCLUDE_NOYAU)/linux/version.h)

MODULES = echo-2.4
OBJETS  = $(MODULES).o
SOURCES = $(TARGET).c

all: $(OBJETS)

echo-2.4.o : echo-2.4.c
        $(CC) $(KCFLAGS) -c echo-2.4.c

install:
        mkdir -p /lib/modules/$(KERNEL_VERSION)/misc /lib/modules/misc
        install -c $(OBJETS) /lib/modules/$(KERNEL_VERSION)/misc
        install -c $(OBJETS) /lib/modules/misc

clean:
```

```
rm -f *.o *~ core
```

Tout cela mérite quelques explications.

Dans le fichier source, la définition de la macro __KERNEL__ est nécessaire pour profiter des définitions valables en mode noyau et pas de celles valables en mode utilisateur. La définition de la macro MODULE permet de spécifier que l'on est en train d'écrire un module. Lors de l'écriture d'un élément qui peut être indifféremment lié statiquement avec le noyau ou chargé dynamiquement comme module, on pourra se baser sur cette macro pour inclure de façon conditionnelle les définitions de fonctions et les appels de macros spécifiques aux modules.

L'inclusion du fichier `linux/modversions.h` est ensuite conditionnée par la définition de la macro MODVERSIONS. Cette macro doit être positionnée pour permettre au module d'être chargé sur un noyau de version différente de celui pour lequel il a été initialement compilé. Le fichier ainsi inclus donne la définition des symboles exportés par le noyau et ses modules. Si nous ne procédons pas à cette inclusion ici, lors du chargement du module, nous obtiendrions le message suivant :

```
# insmod echo.o
echo.o : unresolved symbol printk
```

En effet, si nous cherchons le symbole *printk* dans le noyau en cours d'exécution, nous trouvons par exemple :

```
# grep printk /proc/ksyms
c0114a38 printk_R1b7d4074
```

L'inclusion de `linux/modversions.h` aura donc pour effet de remplacer les appels à *printk* par des appels à *printk_R1b7d4074*, de sorte que le chargement du module se passe bien.

Une fois compilé, nous pouvons donc charger notre module et constater l'affichage des messages d'initialisation et de terminaison. Lançons-le par exemple avec un niveau d'affichage de 1 (alerte système) :

```
# insmod echo.o niveauLog=1
# tail -1 /var/log/kernel/alerts
Jun 4 16 :35 :12 albert kernel : Module echo en place.
```

Ici, le service d'affichage des messages système a été configuré de façon à ce que les messages d'alerte soient enregistrés dans le fichier `/var/log/kernel/alerts`.

1.4.7 Compter les utilisations

Il est important de connaitre, à tout moment, le nombre des "utilisateurs" d'un module, afin de ne pas risquer de supprimer du noyau un module en cours d'utilisation.

Pour cela, chaque fois qu'une portion de code nécessite l'utilisation d'un module, elle doit être encadrée par un appel aux fonctions suivantes, définies (et codées *inline*) dans le fichier `linux/module.h`.

```
int try_module_get(struct module *module);
```

et

```
void module_put(struct module *module)
```

La première de ces fonctions renvoie 0 en cas d'échec. En cas de succès, elle assure que le module est vérouillé jusqu'à ce qu'il soit relâché explicitement par un appel à `module_put`.

Dans la version 2.4, le décompte des utilisations d'un module était géré différemment. Dans cette version du noyau, le fichier `linux/module.h` définit les macros suivantes :

`MOD_INC_USE_COUNT` doit être utilisée au début de chaque nouvelle utilisation des fonctionnalités offertes par un module ;

`MOD_DEC_USE_COUNT` doit être invoquée à la fin de toute utilisation des fonctionnalités offertes par un module ;

`MOD_IN_USE` permet de déterminer si le module courant est en cours d'utilisation ou non.

Naturellement, il est important que les deux premières macros soient utilisées de façon parfaitement symétrique, sous peine de vérouiller un module dans le noyau.

1.5 Débogage *Revision* : 1.4

La recherche de bogues dans le noyau peut se révéler bien plus délicate que dans une application traditionnelle. Une erreur grave qui ferait classiquement "planter" une application, éventuellement en générant un fichier `core` permettant une analyse a posteriori, risque ici de geler complètement le système, sans même laisser au programmeur l'occasion d'enregistrer la moindre information lui permettant de traquer le bogue ...

Nous allons, à titre d'exemple, introduire une erreur dans notre module `cpc`. Modifions simplement ce dernier de la façon suivante :

```
/*
 *      L'implémentation de l'appel système.
 */
int sys_cpc(int pid, int how, uid_t *uid, gid_t gid)
```

puis :

```
   /* Modification des champs du processus */
   if (how & CPC_UID)
      p->uid = *uid;
```

autrement dit, pour la routine de service, le paramètre `uid` est l'adresse de la nouvelle valeur de l'UID.

Nous obtenons alors le comportement suivant :

```
# cpc 0 0 6230
Erreur de segmentation
```

Voyons maintenant comment retrouver l'origine du bogue.

1.5.1 Les "kernel oops"

En cas d'erreur grave survenue lors de l'exécution d'un processus, le noyau génère un "kernel oops", ou "houps du noyau". Il s'agit en fait d'un message décrivant l'état du processeur lorsque le problème est survenu.

```
Unable to handle kernel NULL pointer dereference at virtual address 00000000
current->tss.cr3 = 01cd6000, %cr3 = 01cd6000
*pde = 00000000
```

```
Oops: 0000
CPU:    0
EIP:    0010:[<c588d0e9>]
EFLAGS: 00010202
eax: c1612000   ebx: 00000000   ecx: 00000003   edx: c3e74000
esi: 00000000   edi: 400f4b93   ebp: c1613fbc   esp: c1613fbc
ds: 0018   es: 0018   ss: 0018
Process cpo (pid: 6234, process nr: 90, stackpage=c1613000)
Stack: bffff9bc c010a0d4 00001856 00000003 00000000 00000000 400f4b93 bffff9bc
       000000ff 0000002b 0000002b 000000ff 080484f8 00000023 00000282 bffff9b4
       0000002b
Call Trace: [<c010a0d4>]
Code: 66 8b 06 66 89 82 02 01 00 00 f6 c1 02 74 07 66 89 9a 0a 01
```

Nous pouvons observer sur la pile les paramètres passés à notre appel système :
- le numéro du processus à modifier : 1856 (soit 6230 en décimal) ;
- la valeur du flag `how` : 3 (modification de l'UID et du GID) ;
- la valeur de l'UID : 0 ;
- la valeur du GID : 0.

Nous pouvons alors également utiliser la commande `ksymoops`.

```
>>EIP; c588a0e9 <[cposyscall].text.start+89/f0>   <=====
Trace; c010a0d4 <dump_thread+130c/260c>
Code;  c588a0e9 <[cposyscall].text.start+89/f0>
00000000 <_EIP>:
Code;  c588a0e9 <[cposyscall].text.start+89/f0>   <=====
   0:   66 8b 06                   mov    (%esi),%ax   <=====
Code;  c588a0ec <[cposyscall].text.start+8c/f0>
   3:   66 89 82 02 01 00 00       mov    %ax,0x102(%edx)
Code;  c588a0f3 <[cposyscall].text.start+93/f0>
   a:   f6 c1 02                   test   $0x2,%cl
Code;  c588a0f6 <[cposyscall].text.start+96/f0>
   d:   74 07                      je     16 <_EIP+0x16> c588a0ff <[cposyscall].text.start+9f/
Code;  c588a0f8 <[cposyscall].text.start+98/f0>
   f:   66 89 9a 0a 01 00 00       mov    %bx,0x10a(%edx)
```

Nous avons alors ici à notre disposition le code assembleur au voisinage de l'erreur. Cette information peut permettre de comprendre la source du problème, ceci réclame cependant un minimum de compétences en assembleur et de connaissances des structures de données manipulées.

1.5.2 L'aide du noyau

Comme pour du code classique, il ne faut jamais négliger, lors de la phase de débogage de code noyau, les outils les plus élémentaires. Le système nous offre en particulier la possibilité d'afficher des informations par la simple fonction `printk`. Cela permet en particulier de suivre l'évolution des données et le flux d'exécution afin de mieux cerner le bogue.

L'une des principales différences entre la fonction noyau *`printk()`* et son "homologue" utilisateur *`printf()`* est que les messages affichés par *`printk()`* sont aiguillés par le démon `syslogd` (on peut obtenir un traitement équivalent de messages émanents d'un processus en mode utilisateur par l'utilisation de la fonction *`syslog()`* à la place de la fonction *`printf()`*).

Ce service définit un certain nombre de niveaux d'importance à accorder aux messages, ces niveaux sont déclarés dans le fichier `linux/kernel.h`, ils sont au nombre de huit :

***`KERN_EMERG`* (valeur 0)** pour les messages décrivant un évènement rendant le système inutilisable ;

***KERN_ALERT* (valeur 1)** pour les messages signalant un évènement grave nécessitant une action immédiate ;

***KERN_CRIT* (valeur 2)** pour les messages traduisant une situation critique ;

***KERN_ERR* (valeur 3)** pour les messages informant d'une erreur ;

***KERN_WARNING* (valeur 4)** pour les messages d'avertissement ;

***KERN_NOTICE* (valeur 5)** pour les messages informant d'une situation normale mais significative ;

***KERN_INFO* (valeur 6)** pour les messages strictement d'information ;

***KERN_DEBUG* (valeur 7)** pour les message de chasse au bogue.

Le démon `syslogd` peut donc être configuré pour que les messages produits par le noyau soient enregistrés de façon cohérente afin de faciliter le débogage ; cette configuration se fait via le fichier `/etc/syslog.conf`.

Les messages émis par le noyau sont associés à la *facility* `kern`, ainsi si l'on souhaite enregistrer dans le fichire `/var/log/kernel/alerts` les messages d'alerte du noyau, on ajoutera la ligne suivante dans le fichier `/etc/syslog.conf` :

```
kern.alert      /var/log/kernel/alerts
```

Le niveau d'un message affiché avec *printk()* peut être spcifié en accollant la macro correspondante (*KERN_DEBUG* et ses amis) devant le format d'affichage. Ainsi

```
printk(KERN_ALERT"Erreur fatale dans le module toto ...\n");
```

envoie à `syslogd` le message correspondant avec le niveau d'affichage *KERN_ALERT*. Un comportement similaire peut être obtenu en plaçant la chaîne de caractères correspondante en début du format d'affichage (les huit macros contiennent respectivement les chaînes `"<0>"` à `<"7">`).

1.5.3 Les débogueurs

Il est malheureusement bien plus délicat d'utiliser un débogueur pour chercher des erreurs de programmation au sein d'un noyau qu'au sein d'un programme classique.

Il est cependant possible d'utiliser un débogueur pour observer notemment les valeurs de certaines variables sur le système en cours d'exécution. Pour cela, le système fournit le fichier `/proc/kcore` qui peut être vu comme un fichier `core` classique par un débogueur.

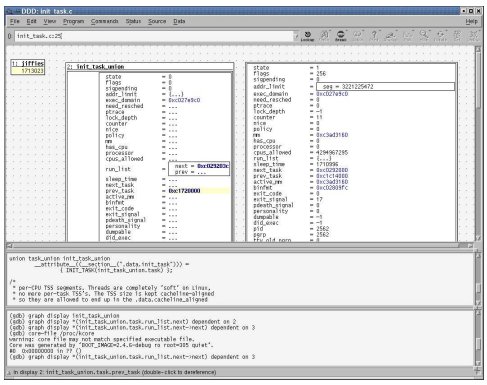

FIG. 1.1 – Débogage avec *gdb* et *ddd*

Ainsi, la figure 1.1 montre comment il est possible d'utiliser le débogueur `gdb` (ici *via* l'interface graphique `ddd`) pour observer un noyau en cours d'exécution.

Nous pouvons y observer une partie de la liste des tâches en cours d'exécution et l'état de celles-ci. Notons qu'il est nécessaire de forcer le débogeur à relire le fichier `core` afin d'observer l'évolution du système.

Un tel débogage est relativement limité, puisqu'il ne permet pas d'agir directement sur le noyau, ni d'observer les modules, du fait de leur aspect dynamique. Il présente cependant le gros avantage de permettre d'observer directement un système en cours d'exécution.

Afin de profiter d'un maximum des avantages d'un débogueur, il est conseillé d'effectuer une compilation du noyau avec l'option de débogage ; pour cela, on ajoutera l'option `-g` à la variable `CFLAGS` du fichier `arch/i386/Makefile` (ou son équivalent pour les autres architectures). On utilisera alors le fichier `vmlinux` ainsi généré comme fichier binaire pour le débogage.

1.5.4 Le débogueur kgdb

Le débogueur *kgdb* est développé spécifiquement pour le débogage d'un noyau Linux en cours d'exécution sur une machine physique réelle. Il nécessite en fait l'utilisation de deux machines : la machine de test, sur laquelle le noyau à déboguer est en cours d'exécution, et la machine de développement, sur laquelle le noyau est compilé, et sur laquelle le débogueur s'exécute. Les deux machines sont reliées entre elles par un cable "null-modem".

En fait, kgdb est fourni sous la forme d'un patch, qui doit être appliqué aux sources du noyau avant la recompliation de ce dernier. C'est ensuite le débogueur gdb qui est utilisé. Le site web `http ://kgdb.sourceforge.net/` donne toutes les informations nécessaires à l'utilisation de kgdb.

1.5.5 Les émulateurs

L'utilisation d'un émulateur peut aider au débogage d'un noyau par la possibilité qu'il offre de contrôler l'exécution du système qu'il héberge. On pourra ainsi placer des points d'arret, exécuter du code pas à pas, consulter l'état des registres, de la mémoire, et tout cela sans perturber l'exécution de ce système ni sans en être tributaire.

L'émulateur *bochs*, par exemple, est tout à fait capable d'exécuter le système Linux et permet de débuger les programmes qu'il héberge.

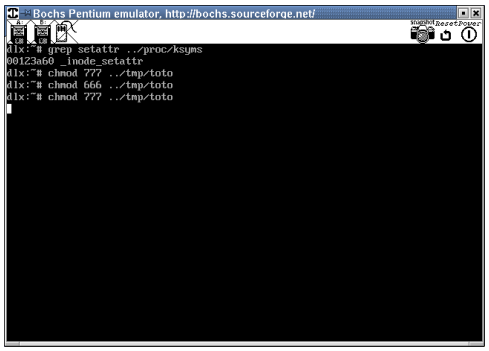

FIG. 1.2 – L'émulateur *bochs* peut exécuter Linux

La figure 1.2 montre par exemple l'émulateur *bochs* en train d'exécuter Linux alors que la figure **??** montre l'interface BFE qui permet de dialoguer plus confortablement avec *bochs* à des fins de débogage. L'utilisateur a consulté l'adresse de la fonction `_inode_setattr()` dans le fichier `/proc/ksyms` du système hébergé par

le débogueur. Il a alors placé un point d'arrèt à cette adresse. Lors de l'invocation de la commande `chmod`, qui utilise cette fonction, le système a donc été stoppé et l'utilisateur a eu tout le loisir de consulter l'état de la pile, des registres, de la mémoire *etc.*

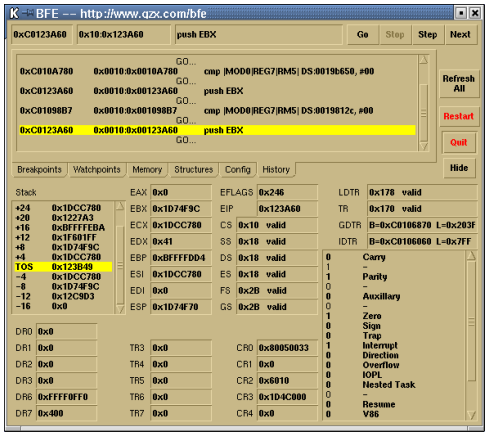

FIG. 1.3 – Débogage avec *bochs* et *bfe*

Il est ensuite tout à fait possible de faire évoluer le système pas à pas, de placer des points d'arret, ou encore des points d'observation (*watchpoint*), ...

1.5.6 Linux en mode utilisateur

Aussi étrange que cela puisse paraître dans un premier temps, il est possible de compiler le noyau Linux en mode utilisateur, c'est-à-dire de sorte qu'il s'exécute comme un processus classique au sein d'un système "réel".

Pour cela, la compilation du noyau sera réalisée de la façon suivante :

```
# make ARCH=um
```

Le système *User Mode Linux* est décrit dans les fichiers du répertoire `Documentation/uml`.

1.6 Le système initrd

Le système initrd permet à un noyau Linux de charger dans un disque virtuel en mémoire vive un système de fichiers qui sera utilisé dans un premier temps comme système de fichiers racine. Le but est que ce système de fichiers contienne tous les éléments nécessaires au chargement complet du noyau, et en particulier des modules.

Ce système de fichiers contient donc un nombre très limités de fichiers, essentiellement des modules résultant de la compilation des éléments du noyau qui n'ont put être intégrés dans ce dernier. On y intègrera par exemple le pilote du périphérique contenant le système de fichiers qui doit finalement constituer la racine de l'arborescence du système. Il est en effet bien sûr nécessaire que le noyau intègre ce pilote pour pouvoir démarrer le système, mais il ne pourra pas le charger sous forme de module puisque le système de fichiers ne peut pas être monté sans lui. Si ce pilote ne peut être compilé que sous forme de module, pour des raisons de taille du noyau, l'utilisation de initrd est alors nécessaire.