Modules

Modules

- Principe inspiré des micro-noyaux (ex: Mach) :
  - Le (micro)noyau ne contient que du code générique
    - synchronization
    - ordonancement
    - Inter-Process Communications (IPC) ...
  - Chaque couche/fonction/pilote de l'OS est écrit(e) de façon indépendante
    - Obligation de définir et utiliser des interfaces/protocoles clairement définis
    - Fort cloisonnement (module MACH = processus)
    - Meilleure gestion des ressources
      - Seuls les modules actifs consomment des ressources

Les Modules de Linux

- Module = fichier objet
  - Implémentation très partielle du concept de micro-noyau
  - Code pouvant être lié dynamiquement à l'exécution
    - Lors du boot (rc scripts)
    - A la demande (noyau configuré avec option CONFIG_KMOD)
  - Mais le code peut aussi, généralement, être lié statiquement au code du noyau (approche monolithique traditionnelle)

Liaisons dynamiques

- Le code d'une librairie n'est pas copié dans l'exécutable à la compilation
- Il reste dans un fichier séparé
- Le *linker* ne fait pratiquement rien à la compilation
  - Il notes les librairies dont un exécutable à besoin
- Le gros du travail est fait au chargement ou à l'exécution
  - Par le *loader* de l'OS
- Le *loader* cherche/charge les bibliothèques dont un programme a besoin
  - Ajoute les éléments nécessaire à l'espace d'adressage du processus

Liaisons dynamiques

- Comportement différent suivant les OS
  - Chargement des bibliothèques au démarrage du processus
  - Chargement des bibliothèques quand le processus en a besoin (*delay loading*)
- Liaisons dynamiques développées pour Multics en 1960
- Utiliser la liaison dynamique implique des relocations
  - Les adresses des sauts ne sont pas connues à la compilation
  - Elles ne sont connues que lorsque le programme et les bibliothèques sont chargées
  - Impossible de pré allouer des espaces
    - Conflits
    - Limitations de l'espace mémoire en 32 bits

Liaisons dynamiques

- On pourrait modifier le programme au chargement
  - Remplace les références aux fonctions extérieures par les adresses mémoire
  - Peu performant
- Utilisation d'une table d'indirection
  - Une table vide est ajoutée au programme à la compilation
  - Toutes les références passent par cette table (*import directory*)
  - Les bibliothèques ont une table similaire pour les symboles qu'elles exportent (*entry points*)
  - Cette table est remplie au chargement par le *loader*
- Plus lent que de la liaison statique

Liaisons dynamiques

- Comment trouver la bibliothèque à l'exécution
  - Dépend des OS
- Spécification du chemin complet dans le programme
  - Très mauvaise idée
- Spécification du nom seulement
  - L'OS doit fournir un mécanisme pour trouver la bibliothèque
- Unix
  - Répertoires bien connus
  - Spécifiés dans /etc/ld.so.conf
  - Variable d'environnement (LD_PRELOAD, LD_LIBRARY_PATH)
- Windows
  - Utilisation du registry pour ActiveX
  - Répertoires bien connus (System32, System…)
  - Ajout de répertoires par *SetDllDirectory()*
  - Repertoire courant et PATH

Modules linux

- Stockés sur le disque sous forme de fichier ELF
- ELF définit 3 types de fichiers qui contiennent
  - Le code
  - Les données
  - Les informations nécessaires pour l'OS ou le *linker/loader*
- *Executable file* : contient les informations pour que l'OS puisse créer une image pour l'exécution et accéder aux données
- *Relocatable file* : indique comment il devra être lié à d'autres fichiers objets pour crée un exécutable ou une bibliothèque dynamique
- *Shared Object file* : contient des informations pour la liaison statique ou dynamique

Modules linux

- Le kernel ne considère que les modules chargés en mémoire par */sbin/insmod*
- Pour chaque module, allocation d'une structure contenant
  - Une structure module
  - Une chaîne **unique** contenant le nom du module (terminée par null)
  - Le code qui implémente les fonctions du module
- Espace mémoire alloué au total
  - Accessible dans le champs *size*

Implémentation des Modules

struct module(linux/module.h)

unsigned long	size_of_struct
module *	**next**
const char *	**name**
unsigned long	**size**
atomic_t	**uc.usecount**
unsigned long	flags
unsigned int	nsyms
unsigned int	ndeps
module_symbol *	syms
module_ref *	**deps**
module_ref *	**refs**
int (*)(void)	**init**
void (*)(void)	**cleanup**
exception_table_entry *	ex_table_start
exception_table_entry *	ex_table_end
module_persist *	persist_start
module_persist *	persist_end
int (*)(void)	**can_unload**
int	runsize
char *	kallsyms_start
char *	kallsyms_end
char *	archdata_start
char *	archdata_end
char *	kernel_data

Structure module

- L'ensemble des modules chargés est maintenu dans une liste chainée
  - `module * next`
  - Premier élément accessible par la variable `module_list`
  - Le premier module est toujours le même
    - Module fictif de nom *kernel_module*
    - Représente le code du kernel
- Le kernel maintient un compte un compteur d'utilisation
  - `atomic_t uc.usecount`
  - Incrémenté à chaque entrée dans une fonction du module
  - Décrémenté en sortie
  - Un module ne peut être déchargé que lorsque son compteur vaut 0
- `atomic_t`
  - Défini dans `atomic.h`
  - Contient un entier accessible seulement à travers des opérations atomiques
  - Implémenté avec des instructions machines spécifiques
  - `atomic_set, atomic_read, atomic_add`…

Structure module

- Le compteur d'utilisation peut être accédé ou modifié manuellement
  - MOD_INC_USE_COUNT
  - MOD_DEC_USE_COUNT
  - MOD_IN_USE (true si >0)
- Toujours incrémenter le compteur d'utilisation **avant** de faire quoi que ce soit
  - Le kernel peut décharger un module dont le compteur vaut 0 quand il veut!
- Un processus qui utilise un module mal programmé peut être interrompu avant la fin d'une méthode
  - Le compteur n'est pas décrémenté
  - Le module ne peut pas être déchargé
- Valeur accessible à l'exécution dans */proc/modules*

Exemple : Compteur d'utilisation

```
#define MODULE
#include <linux/module.h>

int init_module(void) {... }
void cleanup_module(void) {...}

int start_service(int param) {
      MOD_INC_USE_COUNT;
      start service requested by kernel
 start_error:
    MOD_DEC_USE_COUNT;
 start_ok:
    return request_id;
}

int stop_service(int id) {
      if (id non actif)          {ret_val=0; goto stop_error;}
      if (stop_service(id) ok) {ret_val=id; goto stop_ok;}
  stop_ok:
      MOD_DEC_USE_COUNT;
  stop_error:
      return ret_val;
}
```

Structure module

- Au chargement et au déchargement, le kernel peut appeler des méthodes du module
- Spécifications avec des pointeurs sur fonction
  - `int (*)(void) init` : Pointeur sur fonction sans paramètre retournant un int
    - Retourne 0 quand tout va bien
  - `void (*)(void) cleanup`
- Indication explicite lors de l'écriture du module
  - Inclure `<linux/init.h>`
  - `module_init(ma_fonction);`
  - `module_exit(mon_autre_fonction);`
- Gestion fine de la mémoire
  - Attributs `__init` et `__exit`
  - Supprime la fonction après utilisation
  - Ne peut être utilisé que dans une liaison statique

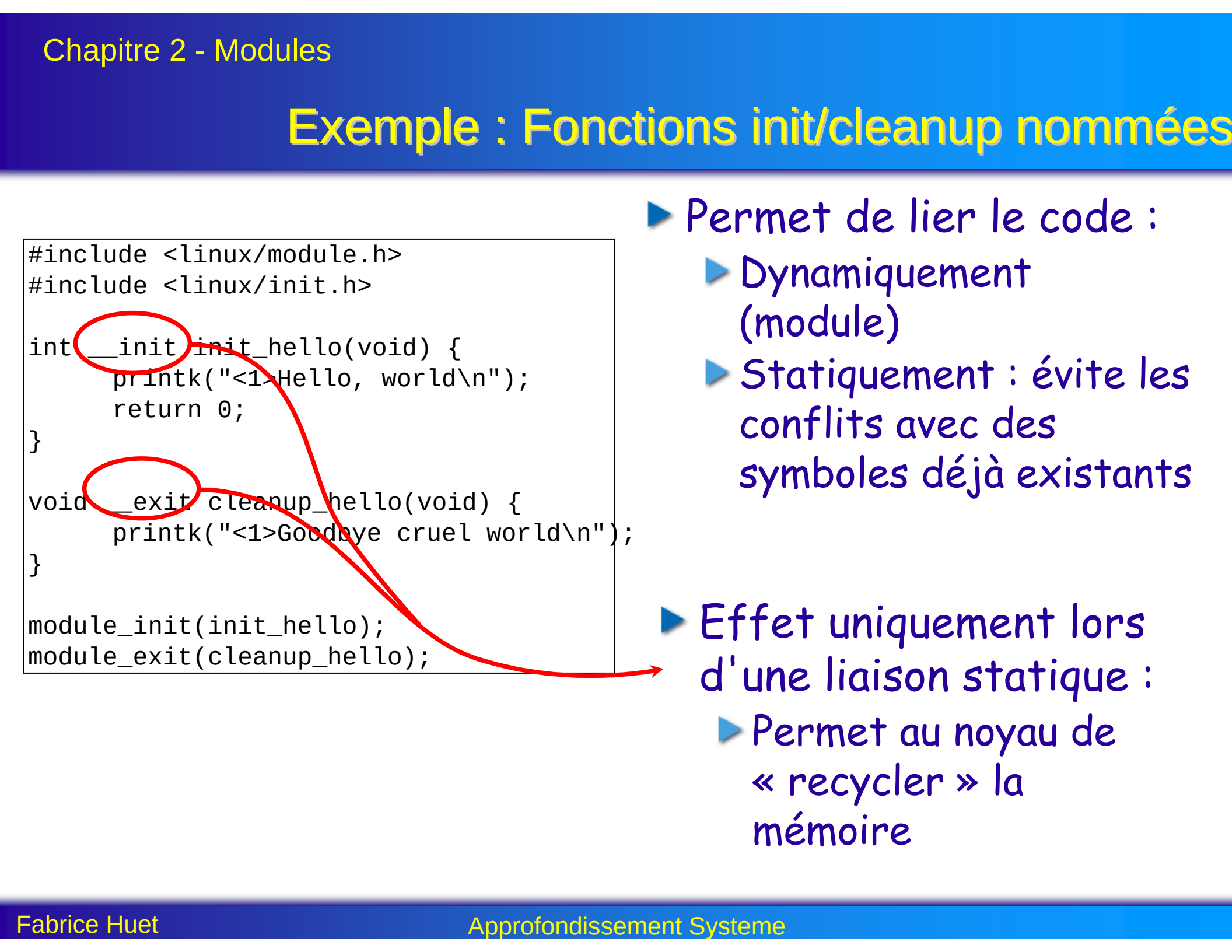
Exemple : Fonctions init/cleanup nommées

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

int __init init_hello(void) {
      printk("<1>Hello, world\n");
      return 0;
}

void __exit cleanup_hello(void) {
      printk("<1>Goodbye cruel world\n");
}

module_init(init_hello);
module_exit(cleanup_hello);
```



- Permet de lier le code :
  - Dynamiquement (module)
  - Statiquement : évite les conflits avec des symboles déjà existants
- Effet uniquement lors d'une liaison statique :
  - Permet au noyau de « recycler » la mémoire

Dépendances

- Quand un module (A) utilise des symboles d'un autre module (B), on dit que A est chargé sur B (*loaded on top of*)
- Induit un ordre de chargement des modules
- Les dépendances entre modules sont gérées par le kernel
- Liste des modules utilisés par le module courant dans `module_ref deps`
- Nombre de modules dans `unsigned int ndeps`
- Référence inverse dans `module_ref refs`
  - Liste mise à jour à chaque chargement/déchargement de module
  - Correspond à une incrémentation/décrémentation du compteur d'utilisation

Chargement manuel de modules

- Le chargement manuel d'un module se fait avec */sbin/insmod*
- Il lit le nom du module sur la ligne de commande
- Calcule la taille nécessaire en mémoire pour ce module (code+nom+structure module)
- Fait un appel système à `create_module()` qui exécute `sys_create_module()`
  - Vérifie si l'utilisateur à le droit de charger un module
  - Cherche dans la liste des modules déjà chargés si un module de même nom existe déjà
  - Alloue la mémoire nécessaire
  - Initialise les champs de la structure module
  - Insert le module dans la liste des modules
  - Retourne le début de la zone mémoire du module

Chargement manuel de modules

- Appel `query_module()`avec `QM_MODULES` comme paramètre pour récupérer le nom de tous les modules
- Appel `query_module()` avec `QM_SYMBOL` pour avoir la table des symboles du kernel et des autres modules
- Fais la relocation des symboles du module
- Alloue une zone mémoire utilisateur et la charge avec une copie de la structure module, du nom, et du code du module après relocation.
- Fais un appel système `init_module()` qui exécute `sys_init_module()`
  - Scan la liste des modules et initialise `ndeps` et `deps`
  - Fixe le compteur d'utilisation à 1
  - Exécute la méthode `init`
  - Fixe le compteur d'utilisation à 0
- Libère la mémoire utilisateur et termine

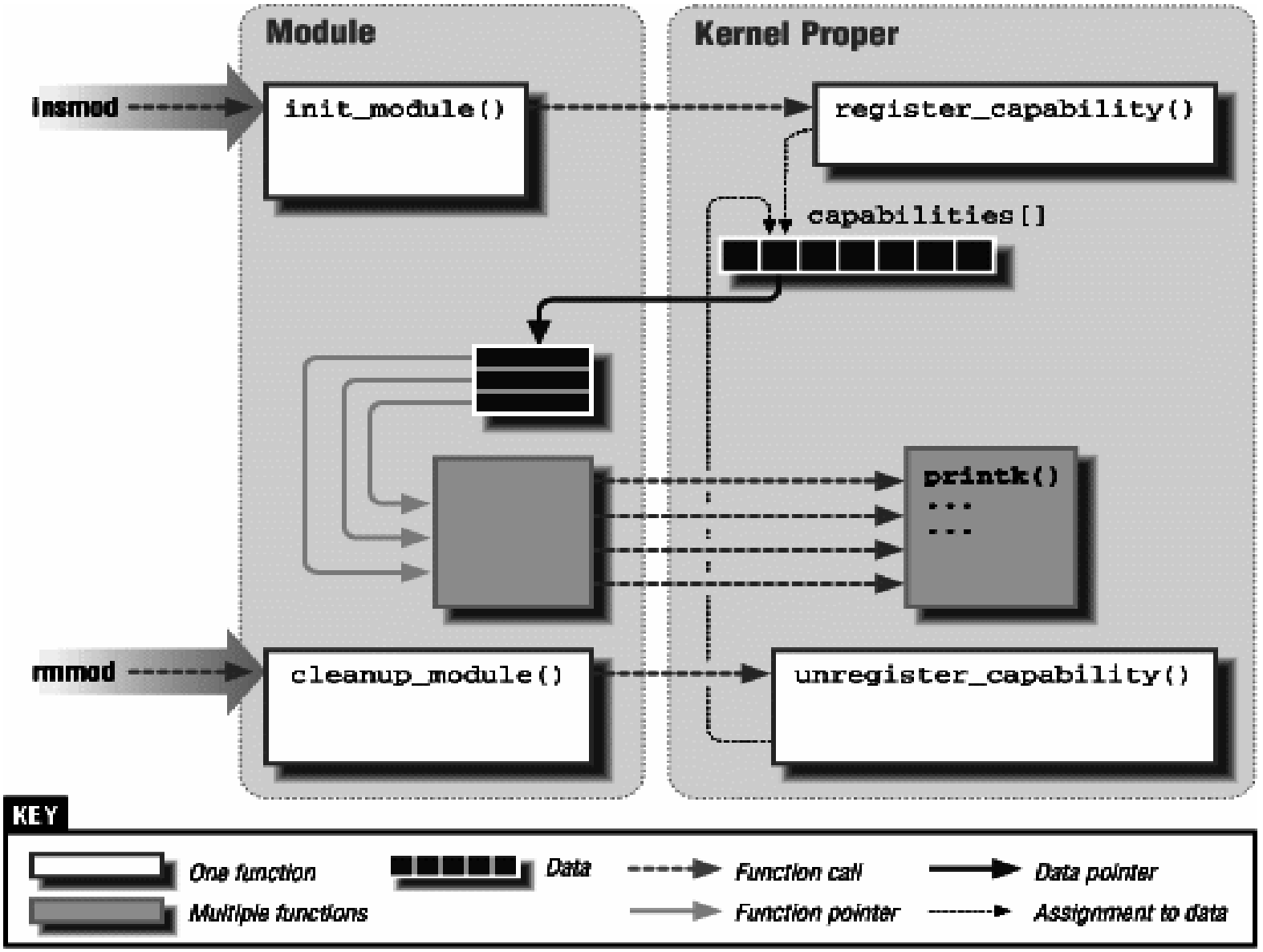
Déchargement manuel de modules

- Le déchargement utilise la commande `/sbin/rmmod`
- Lit le nom du module sur la ligne de commande
- Appel `query_module()`avec `QM_MODULES` comme paramètre pour récupérer le nom de tous les modules
- Appel `query_module()`avec `QM_REFS` plusieurs fois pour obtenir toutes les dépendances
  - Si un module dépendant est trouvé, fin
- Fais un appel système `delete_module` qui exécute `sys_delete_module()`
  - Vérifie les droits de l'utilisateur
  - Invoque `find_module()` pour trouver le module
  - Vérifie que `refs` est null ET `uc.usecount` vaut 0, sinon termine
  - Invoque la méthode `cleanup` si définie
  - Regarde `deps` et supprime le module des `refs`
  - Libère la mémoire et retourne 0

Chargement/déchargement automatique de modules

- Un module peut être chargé quand les fonctionnalités qu'il fournit sont demandées
  - Kernel compilé avec l'option **CONFIG_KMOD**
- Exemple : système de fichier « exotique »
- Utilisation par le kernel de */sbin/modprobe*
- Fonctionnement similaire a *insmod*
  - Mais gestion des dépendances
- En pratique, modprobe appelle insmod
- Les dépendances sont spécifiées par */sbin/depmod*
  - Cherche tous les modules (dans /lib/modules)
  - Écrit la liste des dépendances dans *modules.dep*
- Le déchargement est effectué par */sbin/modprobe -r*
- On peut demander dans le code d'un module le chargement d'un autre module
  - `request_module()`

Interactions entre Modules et Noyau (Rubini)



Écriture de modules

- Un module ne doit pas faire appels aux bibliothèques de base
  - Pas de *stdio.h….*
- Tout ce qui concerne le kernel est déclaré dans les .h dans *include/linux* et *include/asm*
- Certaines déclarations du kernel sont protégées contre une utilisation involontaire
  - Encadrées par un `#ifdef __KERNEL__`
  - Il faut donc définir ce symbole
- Pour utiliser *module.h* il faut définir le symbole `MODULE`
  - Provoque la création d'un symbole `__module_kernel_version`
  - Si plusieurs fichiers pour un module
    - Éviter les importations multiples
    - Ou définir `__NO_VERSION__` avant d'inclure *module.h*
- Pour les multiprocesseur, définir `__SMP__`

Exemple de Module minimal (Rubini)

hello.c

```
#define MODULE
#include <linux/module.h>

int init_module(void) {
      printk("<1>Hello, world\n");
      return 0;
}
void cleanup_module(void) {
      printk("<1>Goodbye cruel world\n");
}
```

```
root# gcc -c hello.c
root# insmod ./hello.o
Hello, world
root#
```

```
root# lsmod
Module                  Size  Used by    Tainted: PF
hello                    748   0  (unused)
input                   5632   0  (autoclean)
ohci1394               26880   0  (unused)
ieee1394               60288   0  [ohci1394]
...
```

```
root# rmmod hello
GoodBye cruel world
root#
```

Dépendance entre modules

- Pour être utilisable par un autre module, il faut exporter les symboles
- Par défaut, tous les symboles sont exportés
- Pour ne rien exporter
  - `EXPORT_NO_SYMBOL;`
- Activer le mécanisme d'exportation
  - `EXPORT_SYMTAB`
  - À définir avant d'inclure module.h
- Utilisation de macros d'export dans le code
  - `EXPORT_SYMBOL(nom_du_symbole);`
  - `EXPORT_SYMBOL_NOVERS(nom_du_symbole);`
  - `EXPORT_SYMBOL_GPL(nom_du_symbole);`
- Les symboles sont placés dans la *module symbole table* du kernel

Configuration de modules

- 3 autres macro permettent d'ajouter de la documentation dans le code objet
- MODULE_AUTHOR (name)
  - Nom de l'auteur
- MODULE_DESCRIPTION (desc)
  - Description du module
- MODULE_SUPPORTED_DEVICE (dev)
  - Description des périphériques supportés par ce module
  - Devrait être utilisé pour le chargement automatique
- MODULE_LICENCE(type)
  - Permet de "teinter" un module :
    - `"Proprietary"`
    - `"GPL"`
    - `"GPL v2"`
    - `"GPL and addintionnal rights"`
    - `"Dual BSD/GPL"`
    - `"Dual MPL/GPL"`

Exemple de Makefile

```
include /usr/src/linux-2.4/.config

CFLAGS = -D__KERNEL__ -DMODULE -DLINUX -Wall -Wstrict-prototypes -fomit-frame-pointer \
            -pipe -I/lib/modules/$(shell uname -r)/build/include -DEXPORT_SYMTAB
CC= gcc

MODS_DIR=/lib/modules/$(shell uname -r)/LABS
MODS= hello.o world.o

all: $(MODS)

$(MODS): $(MODS:.o=.c)

$(MODS): generic_mod.h

$(MODS_DIR):
        mkdir $@

install: $(MODS_DIR) $(MODS)
        cp $(MODS) $(MODS_DIR)
        depmod -a

uninstall: $(MODS_DIR)
        rm -rf $(MODS_DIR)
        depmod -a
```

Utilisation des Symboles Statiques du Noyau ?

- En principe, seuls les symboles exportés
  - Déclarés **"extern"**
  - Les symboles qui ont été prévus pour ça
    - Constitue l'API offerte par le noyau aux modules
- Que faire si on a besoin d'un symbole statique
  - Solution raisonnable 1 : s'en passer !
  - Solution raisonnable 2 : modifier le noyau
    - Ajouter **"extern"** et recompiler le noyau ...
  - Solution sale (mais efficace :-) : tricher
    - Rechercher @ dans la table des symboles
      - /proc/ksyms
      - /boot/System.mapXXX

Exemple d'Utilisation de Symbole Statique

- Hyp: on souhaite utiliser la variable **module_list**
  - Pb : symbole non exporté par le noyau
  - Solution sale :
    - Recherche symbole dans System.map

```
vmdebian:~/TP2# grep module_list /boot/System.map-2.4.27tpasy
c01180d3 T get_module_list
c023edc0 D module_list
```

    - Déclaration du symbole dans le code du module

```
/* /boot/System.map nous donne l'adresse du symbole 'module_list'
 * Comme module_list est un pointeur sur struct module, ce que nous
 * récupérons est de type struct module **
 */
struct module **module_list_ptr = (struct module **)0xc023edc0;
```

    - Pourquoi sale au fait ?
      - Il faut vérifier (et modifier) le code du module à chaque recompilation du noyau (les symboles changent d'@)

Configuration de modules

- Certains modules ont besoin d'informations à l'exécution
  - Adresse hardware pour les I/O
  - Paramètres pour modifier le comportement (DMA…)
- 2 façons de les obtenir
  - Données par l'utilisateur
  - Auto détection
- Les paramètres sont passés au chargement à *insmod* ou *modprobe*
  - insmod monModule toto=25 titi="truc"
- Possibilité de les mettre dans un fichier de configuration */etc/modules.conf*
  - `add options hello sparam="Salut !"`

Configuration de modules

- Un module doit annoncer la liste des paramètres qu'il accepte
- Macro MODULE_PARM de *module.h*
  - Nom du paramètre
  - String indiquant son type
- A placer hors de toute fonction, en général à la fin du fichier
- 5 types
  - b: un byte
  - h : un short (2 bytes)
  - i : un integer
  - l : un long
  - s : une string

Configuration de modules

- Macro MODULE_PARM_DESC de *module.h*
  - Permet d'ajouter une description de l'option
- Description insérée dans le fichier objet
- Visualisable par des outils comme *objdump*
- Les paramètres des modules doivent avoir une valeur par défaut
  - Dans le code du module, comparer la valeur actuelle avec la valeur par défaut
  - Si différente, alors passage explicite par *insmod/modprobe*
  - Sinon, faire autodétection

```
int base_port = 0x300;
MODULE_PARM (base_port, “i”);
MODULE_PARM_DESC (base_port, “The base I/O port (default 0x300)”);
```

Trace de modules

- Technique de base pour débugger
  - Tracer l'exécution
- Pour des programmes utilisateurs
  - Utilisation de printf
- Mais inaccessible depuis l'espace kernel
- Un équivalent existe
  - printk
- Les messages sont classés par niveau (*loglevel*)
- Les niveaux sont accessibles par des macros
  - Définis dans */linux/kernel.h*
  - Strings ("<[0-7]>"), concaténées au message à la compilation
  - Plus le niveau est petit plus la priorité est élevée
- Si aucun niveau de précisé, DEFAULT_MESSAGE_LOGLEVEL dans *kernel/printk.c*

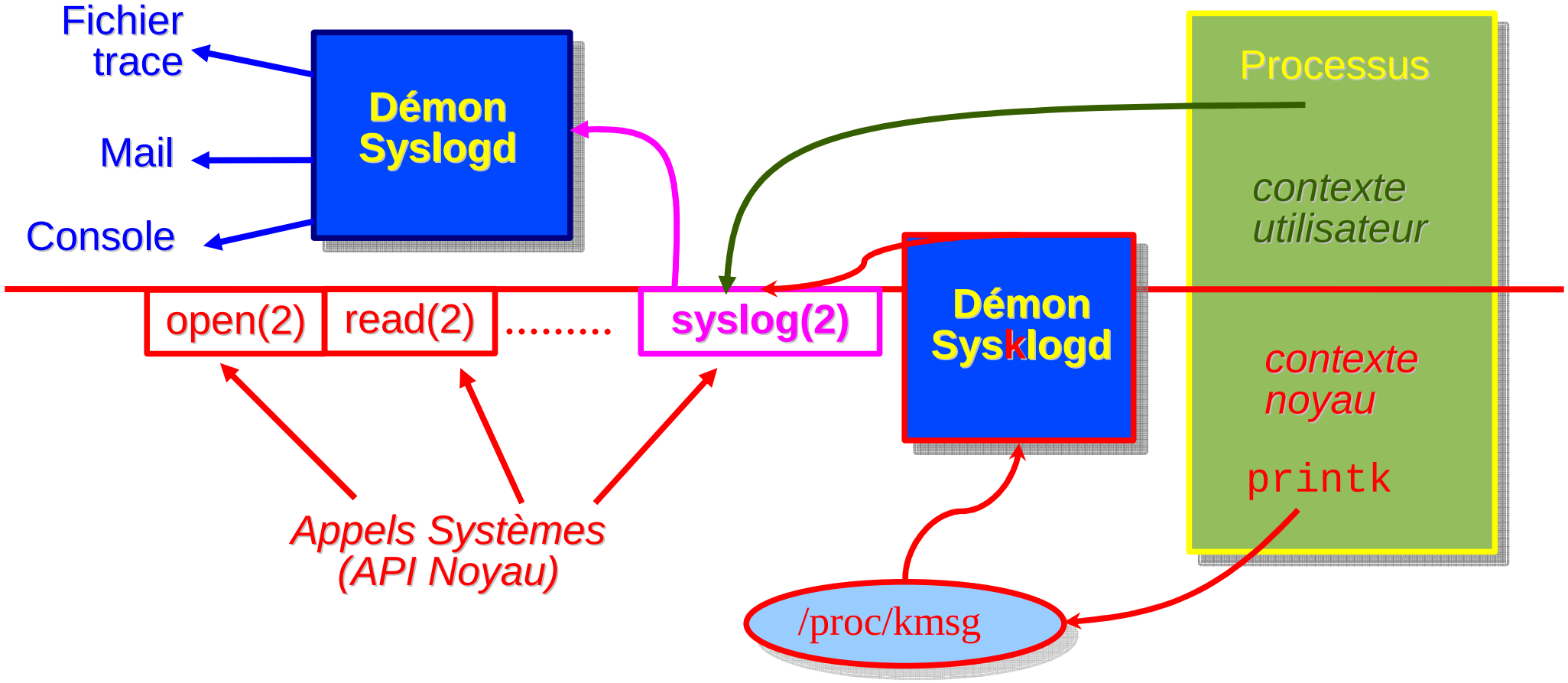
Niveaux de log

- KERN_EMERG
  - Messages urgents, juste avant un plantage
- KERN_ALERT
  - Situation requérant une action immédiate
- KERN_CRIT
  - Situation critique, habituellement une panne hardware/software
- KERN_ERR
  - Erreurs, problèmes hardware
- KERN_WARNING
  - Situations problématique ne présentant pas un gros problème pour le système
- KERN_NOTICE
  - Situation normale
- KERN_INFO
  - Information, par exemple nom du hardware
- KERN_DEBUG
  - Debug

Intéraction klogd syslogd

- Les messages dont la priorité est inférieure à *console_log_level* sont affichés sur la console
- La variable *console_log_level* est initialisée à DEFAULT_CONSOLE_LOGLEVEL
  - Option -c de *klogd*
  - Sur les kernels recents */proc/sys/kernel/printk*
- Les messages sont envoyés au démon *klogd*
- Si *klogd* ne tourne pas
  - Messages lisibles dans */proc/kmsg*
- Si *klogd* et *syslogd* tournent, traitement des messages suivant la configuration de syslog

Intéraction klogd syslogd



Klogd, OOPS et Symboles...

- Message de panique du noyau quand une adresse non valide est manipulée
- Un OOPS contient
  - Une description textuelle
  - Le numéro du OOPS
  - Le numéro du CPU (0 pour mono-processeur)
  - les registres CPU
  - La pile d'appels
  - Les instructions que le CPU executait
- Généré par *printk* (*arch/*/kernel/traps.c*)
- Ne contient souvent aucun symbole, juste des adresses…
- Si klogd tourne, il peut transformer les adresses en symboles
  - Beaucoup plus lisibles
  - À condition qu'il ai accès aux derniers symboles
- La liste des fonctions change au cours du temps
  - Chargement/Déchargement de modules
  - Solution = SIGUSR1 : demande au démon klogd de relire les symboles
  - Ou lancer klogd en mode paranoïaque (-p)

Exemple de OOPS

```
Unable to handle kernel NULL pointer dereference at virtual address 00000004 80289912
*pde = 591be001
Oops: 0002 deadman multipath md bonding1 bonding cpqci cpqhealth cpqrom e100 lpfcdd
CPU: 1
EIP: 0010:[<80289912>] Not tainted
EFLAGS: 00010202
eax: 00000000 ebx: f8f37138 ecx: f8b36d98 edx: 00000000
esi: 00619380 edi: f504ec00 ebp: 00000900 esp: d91bdcf8
ds: 0018 es: 0018 ss: 0018
Process spew (pid: 1230, stackpage=d91bd000)
Stack: f504ec00 f504ed70 f587f370 00bb8000 00000080 00000001 00000050 00000000
       80286341 d91bdd56 d91bdd58 00610180 f504ec00 00003a01 daa9c320 00609250
       00bb8000 00000020 00000200 f587f200 f5a2e000 00610180 00000600 09001000
Call Trace: [<80286341>] [<80286427>] [<80219fdc>] [<8021a051>] [<8013b78c>]
            [<8013b94b>] [<80139b70>] [<8012fd22>] [<80130020>] [<8013008c>]
            [<80130b1292ff>] [<8012919c>] [<801d8425>] [<801d34d9>] [<80137717>]
            [<80106f27>]
Code: 89 42 04 89 10 c7 01 00 00 00 00 c7 41 04 00 00 00 00 8b 03
```

Klogd, OOPS et Symboles...

- Analyse post-mortem possible avec *ksymoops*
- Utile si klogd a planté
- Nécessite les informations suivantes
  - System.map du noyau
  - Liste des modules (par défaut utilise */proc/modules*)
  - Liste des symboles du kernel (*/proc/ksyms*)
  - Une copie de l'image du noyau
- Rend les adresses lisibles par des humains
  - EIP; c88cd018 <[kbdb]__memp_fget+158/b54>

SysRQ

- Combinaison de touches qui force le kernel a répondre
  - Sauf si il est complètement bloqué
- Nécessite l'option de compilation CONFIG_MAGIC_SYSRQ :
  - Touches alt-SysRQ-action (sysRQ=printscreen) sur x86
- Actions:
  - 'r' : sort le clavier du mode raw
  - 'k' : tue tous les programmes sur la console en cours
  - 'b' : reboot immédiat
  - 's' : synchronisation des disques
  - 'u' : remonte les disques en *read-only*
  - 'p' : dump les registres CPU
  - 't' : liste des taches
  - 'e' : envoie SIGTERM à tous les processus sauf *init*
  - 'i' : envoie SIGKILL à tous les processus sauf init
  - 'l' : envoie SIGKILL à tous les processus
- Doit être activé à l'exécution
  - `echo 1 > /proc/sys/kernel/sysrq`

Linux Kernel Linked List

Linked List

- Le langage C n'a pas une bonne collection de structure de données
- Les listes chainées sont pourtant très utilisées
  - Les recoder à chaque fois?
  - Le noyau fournit une implémentation circulaire dans */include/linux/list.h*
- Caractéristiques
  - Pas de limitation sur les types
    - Elles peuvent contenir n'importe quoi
  - Portables
  - Faciles à utiliser
    - Indépendant des types
  - Code lisible

Linked List

- Implémentation classique des listes chaînées
  - L'élément est contenu dans le maillon de la chaîne

```
struct ma_liste {
        void * monElement;
        struct ma_liste *suivant;
        struct ma_liste *precedent;
}
```

- L'implémentation des listes du kernel fait l'inverse
  - Les maillons sont contenus dans les éléments

```
struct ma_liste_kernel {
        struct list_head list;
        struct maStruct monElement;
}
```

  - C'est la structure `list_head` qui maintient la liste

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

Linked List

- Les listes sont déclarées par des macros
- INIT_LIST_HEAD
  - Prend un pointeur sur la liste dans une structure
  - Initialise la liste
- LIST_HEAD(HEADNAME)
  - HEADNAME : nom de la structure
  - Déclare et initialise la structure
- list_add, list_add_tail, list_del…

Linked List

- Dans une implémentation classique
  - On manipule des pointeurs sur la liste
  - On accède aux éléments de la liste en déréférencent
- Dans les listes kernel
  - Comment retrouver l'élément à partir du pointeur sur liste?
  - Utilisation d'une macro `list_entry()`
  - 3 paramètres
    - Pointeur vers le maillon courant
    - Le type de la structure
    - Le nom du maillon dans cette structure

Linked List – Exemple (d'après Linux Kernel Linked List Explained)

- On définit la structure suivante

```
struct kool_list {
    int to;
    struct list_head list;
    int from;
  }
```

- Et on instancie une liste

```
struct kool_list my_list;
INIT_LIST_HEAD(&mylist.list);
```

- On ajoute un nouvel élément

```
struct kool_list *tmp;
tmp = (struct kool_list *)malloc(sizeof(struct
                                  kool_list));
list_add(&(tmp->list),&(mylist.list));
```

Linked List – Exemple (d'après Linux Kernel Linked List Explained)

- Parcourir la liste
  - Utilisation de la macro list_for_each

```
struct list_head *pos;
struct kool_list *tmp;
list_for_each(pos,&mylist.list)
    tmp = list_entry(pos,struct kool_list, list);
}
```

Fonctionnement de list_entry

- Comment peut-on obtenir un pointeur sur la structure englobante?
  - On a un `struct list_head`
-
```
#define list_entry(ptr, type, member)
        ((type *)((char *)(ptr)-(unsigned long)
                    (&((type *)0)->member)
                  )
         )
```
- Exemple pour `kool_list`

```
((struct kool_list *)((char *)(pos) - (unsigned long)
                      (&((struct kool_list *)0)->list)))
```

- Principe
  - On a un pointeur vers list_head dans notre structure
  - On a donc l'adresse absolue de cet élément
  - Il faut calculer la position du début de la structure par rapport à ce pointeur

Fonctionnement de list_entry

- Comment trouver cet offset?
  - On part de l'adresse 0
  - On cast cette adresse comme la structure englobante
  - On prend l'adresse de l'élément qui nous intéresse
  - On a donc l'offset de cet élément dans notre structure de données
- Exemple
  - Une structure foo_bar contenant un élément boo
  - On veut connaître l'offset de cet élément
  - `(unsigned long) (&((struct foo_bar *)0)->boo)`