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

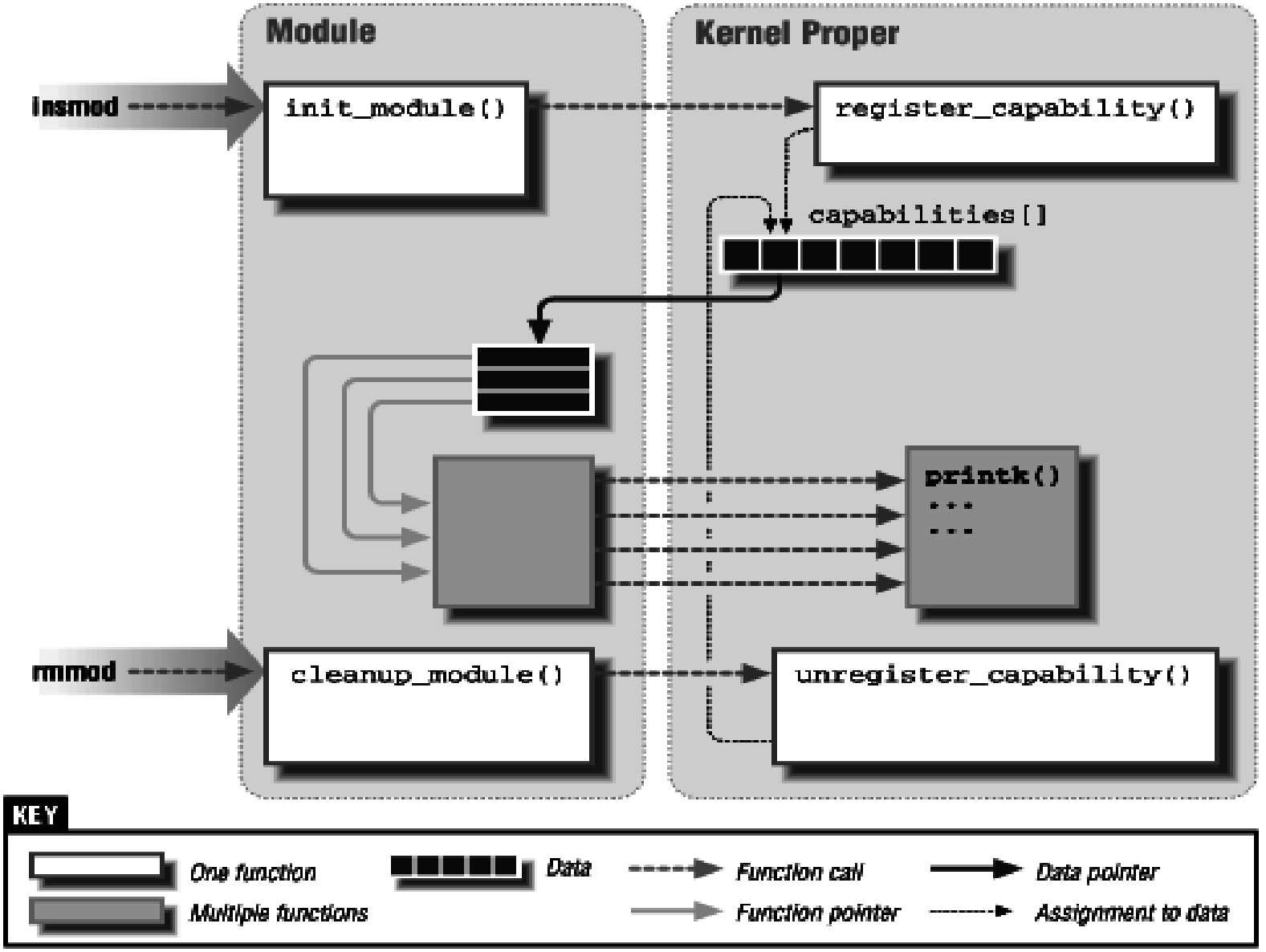
Les Modules de Linux

- Module = fichier objet
  - Implémentation « légère » 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)

Contextes d'Exécution

- Lors du chargement :
  - contexte du processus **`insmod`**
    - A.S. **`create_module()`** puis **`init_module()`**
- Lors du déchargement :
  - contexte du processus **`rmmod`**
- Entre les deux ?
  - Rien de clairement délimité !
  - Comme le reste du noyau
    - Contexte d'un processus lors d'un AS
    - Contexte d'un traitant d'interruption ...

Interactions entre Modules et Noyau (Rubini)



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#
```

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

Liste simplement chainée (tête = module_list)

Compteur d'utilisations : permet d'éviter que le module ne disparaisse en cours de route.
Change avec macros :
- MOD_INC_USE_COUNT
- MOD_DEC_USE_COUNT

Fonction d'init (cad pas necéssairement nommée init_module ...)

Fonction de terminaison (pas forcément nommée cleanup_module ...)

Fonction supplémentaire (facultative) pour décider (bloquer) le déchargement du module.

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;
}
```

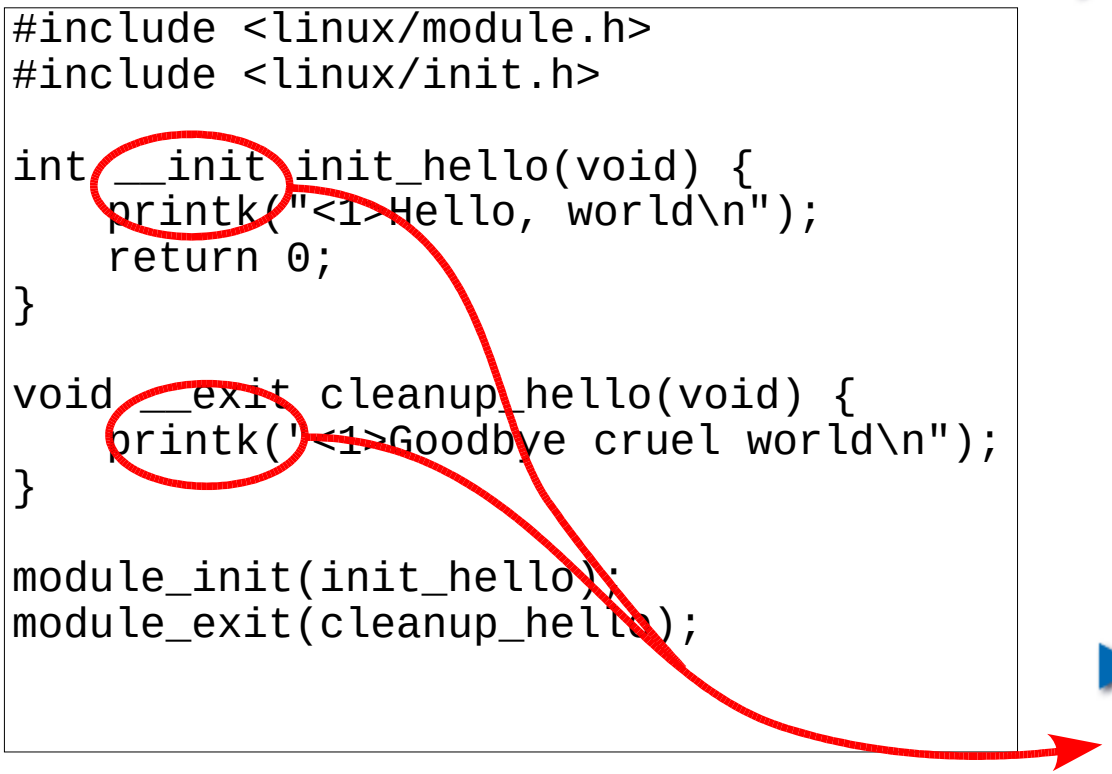
Exemple : Fonctions init/cleanup (re)nommées

```
#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

Symboles Externes et Dépendances entre Modules

- Philosophie = ne pas tout réécrire à chaque fois !
  - Le code d'un module peut-être utile à d'autres modules
    - Ce module « exporte » ses services (publie les symboles correspondants)
    - 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);`
- Module « fournisseur » compilé avec **`-DEXPORT_SYMTAB`**
  - Mais il peut choisir de ne rien fournir ... (!)
    - Macro **`EXPORT_NO_SYMBOLS`**

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'@)

Liaison Dynamique

- Module ayant besoin des services d'un autre module
  - Lors du chargement du module client, insmod
    - recherche les symboles importés par le nouveau module
    - recherche les symboles exportés par d'autres modules
    - Assure la liaison dynamique (ou échoue)
      - Construction des listes de symboles exportés et importés par le nouveau module
    - Met à jour les compteurs de reférence : **`MOD_INC_USE_COUNT`**
  - Lors du déchargement, **`rmmod`** fait l'inverse :
    - Reconstruction des listes de symboles exportés pour les modules restants
    - Mise à jour des compteurs de références

Dépendances entre 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

- Nombre de symboles exportés
- Nombre de modules « fournisseurs »
- Table des symboles exportés
- Liste des modules utilisés (« fournisseurs »)
- Liste des modules « clients »

Chargement de Modules à la Demande

- Chargement (auto.) pour résoudre une dépendance
  - Détectée lors du chargement d'un nouveau module
  - Noyau configuré avec **`CONFIG_KMOD`**
  - Le noyau fait appel à un processus externe : **`modprobe`**
    - Recherche les « fournisseurs »
      - à partir d'une **liste de dépendance** (**`/lib/modules/.../*.dep`**)
      - La liste est construite lors du boot par la commande **`depmod`**
    - **`modprobe`** charge récursivement les modules
      - Applique les directives de `/etc/modules.conf`
- Chargement à la demande explicite d'un autre module
  - fonction **`request_module()`**

Déchargement de Modules Chargés à la Demande

- Non prise en charge par le noyau
- Exécution manuelle : **`modprobe -r`**
  - Décharge tous les modules
    - dont le compteur de reférences est à 0
    - et pour lesquels **`can_unload()`** retourne 1
  - Peut-être automatisée à l'aide cron

Dépendances Liées à la Version du Noyau

- La liste des symboles exportés par le noyau varie d'une version à l'autre
  - Recompiler les modules pour chaque version
    - Installation indépendante (`/lib/modules/<uname -r>/`)
    - Inclusion de `<linux/version.h>`
      - Inclus automatiquement par `<linux/module.h>`
      - Macros `UTS_RELEASE`, `LINUX_VERSION_CODE`, `KERNEL_VERSION(major,minor,release)`
- Modules compilés à partir de multiples sources
  - Le linker (ld -r) se plaint de définitions multiples
    - <linux/version.h> ne doit être compilé qu'une seule fois
    - Solution : Déclarer la macro `__NO_VERSION__` avant d'inclure `<linux/module.h>`

printk et klogd

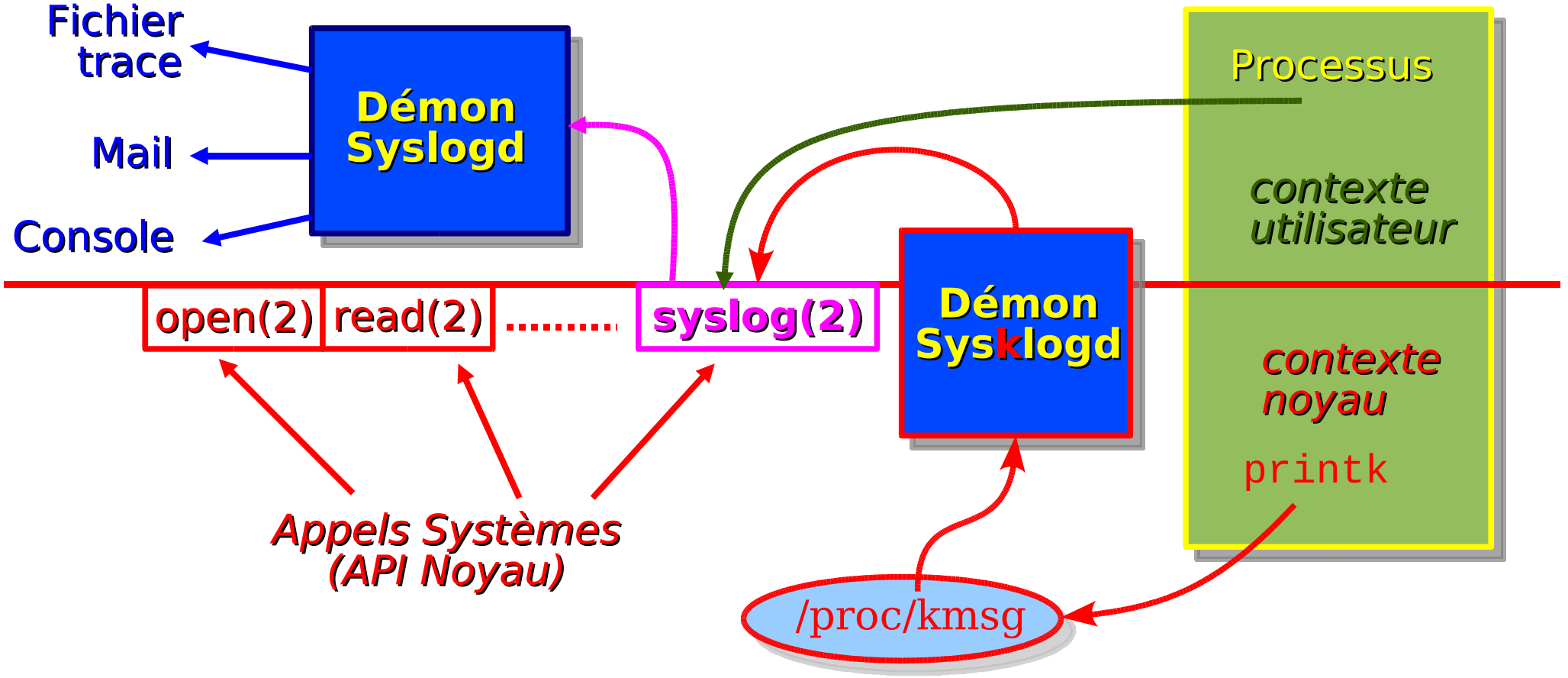
- printk permet de générer des traces
  - Fonctionnement copié sur celui de syslog(3)
    - Les traces sont récupérées (eventuellement) par le démon klogd
    - Les traces ont un niveau de priorité (équivalent aux priorités de syslog)

<linux/kernel.h> :

```
#define KERN_EMERG      "<0>"   /* system is unusable                   */
#define KERN_ALERT      "<1>"   /* action must be taken immediately     */
#define KERN_CRIT       "<2>"   /* critical conditions                  */
#define KERN_ERR        "<3>"   /* error conditions                     */
#define KERN_WARNING    "<4>"   /* warning conditions                   */
#define KERN_NOTICE     "<5>"   /* normal but significant condition     */
#define KERN_INFO       "<6>"   /* informational                        */
#define KERN_DEBUG      "<7>"   /* debug-level messages                 */
```

    - klogd retransmet les traces vers syslog (avec le niveau de priorité correspondant)

Schéma de Fonctionnement



Klogd, OOPS et Symboles...

- OOPS = message de panique du noyau
  - Peut-être provoqué manuellement (CONFIG_MAGIC_SYS_REQ) : <altgr>+<scoll_lock>
  - Affiche le contenu des registres et de la pile du processus actif
  - + autres fonctions :
    - echo 1 > /proc/sys/kernel/sysrq
    - <alt>+<sysreq>+<b|o|s|p|t|m|h>
- Pb : le contenu de la pile donne les adresses des fonctions
  - La liste des fonctions change au cours du temps
    - Chargement/Déchargement de modules
  - Solution = SIGUSR1 : demande au démon klogd de relire les symboles

Utilisation Excessive de printk ?

- Un des outils de déboguage les plus efficaces ...
  - ... mais chaque utilisation fait grossir le code objet !
  - Nettoyage du code final fastidieux
  - Apparition "spontanée" de bugs toujours possible
    - Version nettoyée peu bavarde
- Solution : compilation conditionnelle des printk
  - Exemple : utilisation de macros ciblées

```
#ifdef DEBUG_MMBR_OPS
#include <linux/sched.h>
#define DPRINTK(format,arg...) PRINTK(KERN_DEBUG "%5d %s %s:" format,\
                                      current->pid,__FILE__, \
                                      __FUNCTION__,##arg)
#else
#define DPRINTK(format,arg...) do {} while (0)
#endif
```

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
```

Configuration des Modules

- Les modules peuvent être configurés à l'aide de paramètres
  - Déclaration dans le code du module à l'aide de macros :
    - `MODULE_PARM(symbole, "type")`
    - `MODULE_PARM_DESC(symbole,"Description textuelle")`
  - 5 types de données possibles pour "type"
    - "b" : byte
    - "h" : short (2 octets)
    - "i" : integer
    - "l" : long
    - "s" : string
  - Possibilité de déclarer des tableaux
    - Ex: `"3-5b"` signifie entre 3 et 5 entiers attendus

Affectation des Valeurs aux Paramètres des Modules

- Par ordre de priorité décroissant :
  1. Directement en ligne de commande
     - `insmod/modprobe ./hello.o param=24`
  2. Déclaration dans `/etc/modules.conf`
     - `add options hello sparam="Salut !"`
  3. Valeur par défaut dans le code du module

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

int param = 18
char *sparam = "Hi, folks !"

MODULE_PARM(param,"i");
MODULE_PARM_DESC(param, "set param value (default = 18)");
MODULE_PARM(sparam,"s");
MODULE_PARM_DESC(param, "set sparam value (default = "Hi, folks !)");
```

Quelques Macros Utiles...

- `MODULE_AUTHOR("Nom De L'Auteur");`
- `MODULE_DESCRIPTION("Supports ... ");`
- `MODULE_LICENCE(type)`
  - Permet de "teinter" un module :
    - `"Proprietary"`
    - `"GPL"`
    - `"GPL v2"`
    - `"GPL and addintionnal rights"`
    - `"Dual BSD/GPL"`
    - `"Dual MPL/GPL"`