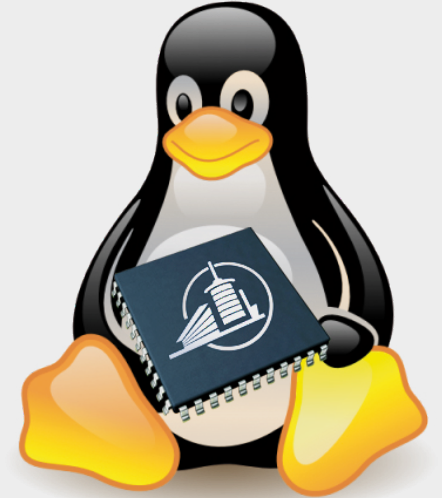


Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

MA-CSEL1 – Construction Systèmes Embarqués sous Linux

# Noyau:
# Modules noyaux

HES-SO//Master TIC/TIN 2020-21

Daniel Gachet – HEIA-FR – Télécommunications

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz

# Contenu

- **Introduction**
  - Modes d'opération (user/kernel space)
  - Mémoire physique et virtuelle
  - Virtual file system
- **Modules**
  - Concept de module
  - Debugging
  - Génération d'un module
  - Installation / désinstallation
  - Paramètres d'un module
  - Allocation dynamique de la mémoire
  - Bibliothèques et fonctions utiles
  - Accès aux entrées/sorties
  - Threads dans le noyau
  - Accès concurrents
  - Mise en sommeil
  - Gestion des interruptions

MASTER OF SCIENCE IN ENGINEERING
MSE

# Introduction

MASTER OF SCIENCE IN ENGINEERING | MSE

# Modes d'opération

Espace utilisateur

Application 1

Application 2

Bibliothèque A

Bibliothèque B

Bibliothèques standards (C, Posix)

Espace noyau

Appels système, services du noyau

Notifications, mise à disposition d'information du système

Noyau Linux

R/W

Drivers

IRQ

Gestion du matériel et des périphériques

Notification d'événements

Hardware

MASTER OF SCIENCE IN ENGINEERING
MSE

# Modes d'opération (II)

**Linux connaît deux modes d'opération:**

- Mode utilisateur (User Mode/Space):
  - *La majorité des programmes utilisateurs travaillent dans ce mode (browser, éditeurs, compilateurs,… )*
  - *Chaque programme/application dispose d'un environnement virtuel et protégé*
  - *Le crash d'une application n'affecte pas les autres*
  - *L'accès direct au matériel n'est pas autorisé, celui-ci doit se faire par l'intermédiaire d'un pilote de périphériques*

- Mode noyau (Kernel Mode/Space):
  - *Le système d'exploitation travaille dans ce mode*
  - *Les pilotes de périphériques sont développés pour fonctionner dans ce mode et pour pouvoir accéder au matériel*
  - *La majorité des piles de protocoles (protocol stacks) sont réalisées pour le noyau, afin d'obtenir de meilleures performances*
  - *Le crash d'un logiciel dans le noyau provoque le crash de tout le système*

# Modes d'opération (III)

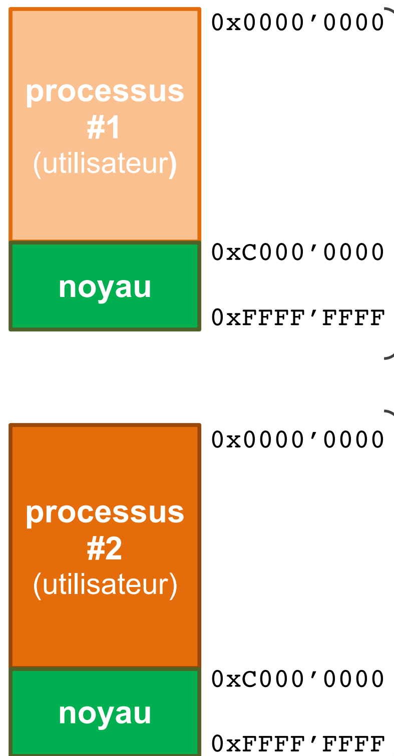
**Interface entre espace utilisateur et espace noyau:**

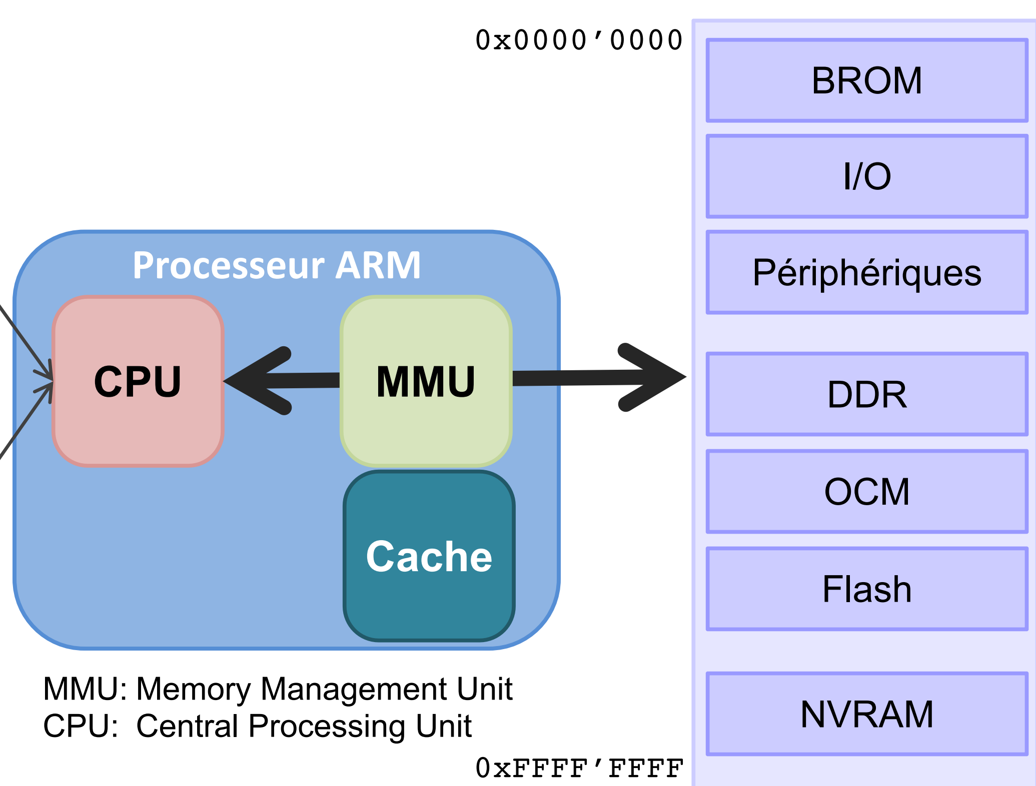
- Appels système (**System calls**)
  - *L'interface entre les 2 espaces est réalisée par l'intermédiaire appels système*
  - *Un appel system est une interruption logicielle, laquelle permet à un processus en espace utilisateur de sortir de son environnement protégé et d'appeler des fonctions du noyau Linux*
  - *Actuellement le noyau Linux offre plus de 400 appels système (opérations sur les fichiers, les périphériques, le réseau, les processus, etc.)*
  - *Cette interface est stable. Seuls de nouveaux services sont proposés par les développeurs du noyau*

- Pseudo système de fichiers (**Pseudo Filesystems**)
  - *Les informations du système et du noyau sont mises à disposition des applications de l'espace utilisateur par le biais d'un pseudo système de fichiers*

# Mémoire physique et mémoire virtuelle

- Tous les processus possèdent leur propre espace d'adressage virtuel
- Chaque processus fonctionne comme s'il disposait de la totalité de la mémoire




MASTER OF SCIENCE IN ENGINEERING | MSE

# Organisation de la mémoire virtuelle (machine 32 bits)

- **Répartition standard de la mémoire**
  - 1GB pour le noyau Linux (kernel space)
  - 3GB pour chaque processus (user space)
- **Noyau Linux**
  - Code et données identiques pour tous
  - Accès aux I/O et aux périphériques
  - Si plus de 1GB est nécessaire pour le noyau
    - *Changer le mode 1GB/3GB → 2GB/2GB → 3GB/1GB*
    - *Activer le support «HIGHMEM»*
    - *Changer pour une architecture 64bits*
- **Processus**
  - Code et données du processus (programme, pile, …) différents d'un processus à l'autre
  - Tout n'est pas alloué au lancement du processus, allocations dynamiques selon les besoins

# Organisation de la mémoire virtuelle (machine 64 bits)

Référence : https://www.kernel.org/doc/Documentation/arm64/memory.txt



So Kernel + User Spaces add for 256 TiB which is a tiny part of the 16 777 216 TiB addressable over 64 bit!

https://commons.wikimedia.org/wiki/File:Linux_Virtual_Memory_Layout_64bit.svg

MASTER OF SCIENCE IN ENGINEERING MSE

# Système de fichiers virtuels



MASTER OF SCIENCE IN ENGINEERING
MSE

# Système de fichiers virtuels (II)

- **Linux met à disposition des applications fonctionnant dans l'espace utilisateur les données du matériel et les informations du noyau et du système par l'intermédiaire d'un système de fichiers virtuels.**
- **Ce système de fichiers virtuels permet aussi bien d'accéder à des informations physiques (p. ex. stockées sur un disque dur ou en mémoire flash/ram), qu'à des données d'interfaces sérielles, qu'à des données du réseau, qu'à des informations du système stockées dans des fichiers virtuels créés par le noyau lui-même.**
- **A part les systèmes de fichiers permettant l'accès au matériel, trois systèmes de fichiers virtuels sont très importants**
  - **`procfs`**: informations sur le système d'exploitation (monté sous **`/proc`**)
  - **`sysfs`**: informations sur le matériel et les périphériques (monté sous **`/sys`**)
  - **`debugfs`**: informations utiles au développement (à monter généralement soi-même: `mount -t debugfs none /sys/kernel/debug`)

MASTER OF SCIENCE IN ENGINEERING
MSE

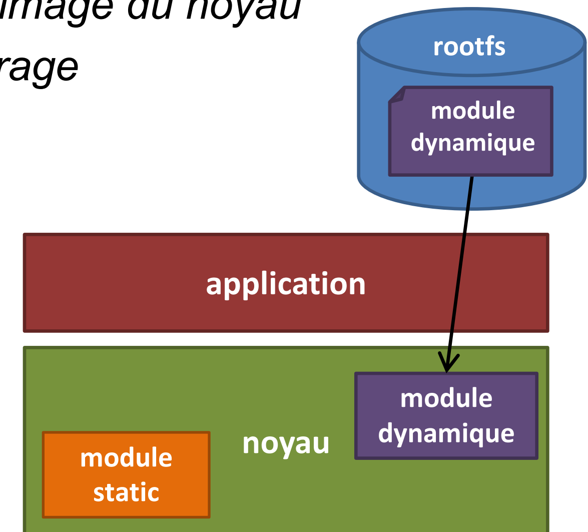
# Concept de module

MASTER OF SCIENCE IN ENGINEERING
MSE

# Introduction

- **La fonctionnalité du noyau Linux peut en tout temps être adaptée, modifiée ou étendue**
  - Des applications nécessitant un accès au matériel ou devant traiter une très grande quantité d'information en provenance d'interfaces réseau (piles de protocoles) seront assez naturellement conçues pour fonctionner dans l'espace noyau en collaboration avec les threads du noyau Linux. Dans ce cas on parle de modules noyau ou d'objets noyau *(kernel objects)*.

- **Linux propose deux techniques pour lier ces modules avec le noyau**
  - Statique
    - *Le module est compilé et linké statiquement avec l'image du noyau*
    - *Le module est chargé avec le noyau lors du démarrage*
  - Dynamique
    - *Le module est compilé séparément du noyau, mais en utilisant la version courante du noyau (current running kernel)*
    - *Le module est chargé par des applications utilisateur en fonction des besoins*
    - *Ces modules/objets noyau ont l'extension *.ko*




MASTER OF SCIENCE IN ENGINEERING
MSE

# Avantages des modules noyaux

- **Il existe de nombreux avantages à développer des modules noyaux**
  - Développement plus aisé et plus rapide
    - *Pas nécessaire de redémarrer le noyau Linux après une modification du module, les modules peuvent être chargés, testés, déchargés, régénérés*
  - Maîtrise de la taille du noyau Linux
    - *En ne chargeant que les modules utiles, on peut plus facilement garder la taille de l'image du noyau Linux au minimum*
  - Chargement de pilotes de périphériques que si le système les reconnaît
    - *Si le système ne connaît pas quels périphériques sont attachés au système avant son démarrage ou si les périphériques ne sont systématiquement utilisés par les applications, alors ceux-ci ne seront pas chargés dans le noyau Linux*
  - Diminution du temps de démarrage des cibles
    - *Aucun temps n'est perdu à initialiser des périphériques ou des fonctions du noyau si elles ne sont pas utilisées*
  - Pas de modification des sources de la distribution
    - *Ils ne nécessitent pas (en principe) de modification des sources du noyau Linux et de son dépôt (git repository) car les modules peuvent être développés à l'extérieur de l'arborescence du noyau Linux*

# Limitations des modules noyaux

- **Le développement de module sous Linux souffre de quelques limitations:**
  - Le module doit impérativement être développé en C. C++ ou d'autres langages de programmation ne sont pas supportés (voir http://vger.kernel.org/lkml/#s15-3).
  - Le module doit impérativement être développé sous licence en général GPL (GNU Public Licence)
  - La bibliothèque standard C n'est pas disponible dans le noyau. D'autres méthodes doivent être utilisées. Les fichiers propres au noyau sont placés dans le répertoire `<linux/>`
  - L'usage des nombres en virgule flottante n'est pas supporté si le µP ne dispose pas d'un coprocesseur mathématique
  - Le débogage des modules est nettement plus complexe
- **Attention**
  - Une fois chargé dans le noyau, le module a plein contrôle sur l'ensemble du système avec tous les privilèges
  - Une erreur de programmation dans un module peut causer le crash de tout le système

MASTER OF SCIENCE IN ENGINEERING | MSE

# Squelette d’un module

```c
// skeleton.c
#include <linux/module.h>      // needed by all modules
#include <linux/init.h>        // needed for macros
#include <linux/kernel.h>      // needed for debugging

static int __init skeleton_init(void)
{
    pr_info ("Linux module skeleton loaded\n");
    return 0;
}

static void __exit skeleton_exit(void)
{
    pr_info ("Linux module skeleton unloaded\n");
}

module_init (skeleton_init);
module_exit (skeleton_exit);

MODULE_AUTHOR ("Daniel Gachet <daniel.gachet@hefr.ch>");
MODULE_DESCRIPTION ("Module skeleton");
MODULE_LICENSE ("GPL");
```

MASTER OF SCIENCE IN ENGINEERING | MSE

# Explications du module

- **Initialisation**
  - La fonction «**skeleton_init**» est appelée lorsque le module est chargé dans le noyau. Elle retourne un code d'erreur (0 pour succès, une valeur négative, p. ex. - EACCES, en cas d'erreur). Liste des codes d'erreurs:

    *#include <linux/errno.h>*

    Elle est éliminée après l'initialisation du module (macro `__init`).
  - La macro «**module_init**» permet de déclarer le nom de la fonction d'initialisation.

- **Nettoyage**
  - La fonction «**skeleton_exit**» est appelée lorsque le module est désinstallé. Si le module est compilé statiquement avec le noyau, la fonction est écartée (macro `__exit`).
  - La macro «**module_exit**» permet de déclarer le nom de la fonction d'initialisation.

- **Métadonnées**
  - Les macros «**MODULE_AUTHOR**», «**MODULE_DESCRIPTION**» et «**MODULE_LICENSE**» permettent de déclarer des informations sur le module et le type de licence utilisé (généralement GPL).

# Debugging

MASTER OF SCIENCE
IN ENGINEERING
MSE

## Debugging d’un module

- **Le débogage des pilotes sous Linux pour des systèmes embarqués est relativement malaisé.**
  - L’utilisation de debugger, tel que kgdb, peut naturellement est utilisé, mais généralement nécessite une infrastructure supplémentaire (p. ex. probes JTAG).
  - La manière la plus simple est d’insérer à l’intérieur du code des **printf.** Dans le noyau, cette technique peut être réalisée avec la méthode **printk**, p. ex. :

```
printk (KERN_DEBUG "Here I am: %s:%s\n", __FILE__, __LINE__);
```

  - Ces messages sont stockés dans un tampon circulaire (cela évite de consommer trop de mémoire si le nombre de messages explose)
  - Ces messages peuvent également être affichés sur la console après avoir passé un filtre dont le niveau est spécifié par le paramètre du noyau «**loglevel**» ou par **/proc/sys/kernel/printk** (voir Documentation/sysctl/kernel.txt)
  - La commande **dmesg** permet de lire les messages, p. ex.

```
$ dmesg | tail –n10
```

MASTER OF SCIENCE IN ENGINEERING
MSE

# Debugging d’un module (II)

- **Il existe 8 niveaux de sévérité, du plus haut (0) au plus bas (7):**

| | |
|---|---|
| **<0> KERN_EMERG** | Used for emergency messages, usually those that precede a crash. |
| **<1> KERN_ALERT** | A situation requiring immediate action. |
| **<2> KERN_CRIT** | Critical conditions, often related to serious hardware or software failures. |
| **<3> KERN_ERR** | Used to report error conditions; device drivers often use KERN_ERR to report hardware difficulties. |
| **<4> KERN_WARNING** | Warnings about problematic situations that do not, in themselves, create serious problems with the system. |
| **<5> KERN_NOTICE** | Situations that are normal, but still worthy of note. A number of security-related conditions are reported at this level. |
| **<6> KERN_INFO** | Informational messages. Many drivers print information about the hardware they find at startup time at this level. |
| **<7> KERN_DEBUG** | Used for debugging messages. |

- **L’usage de la fonction `printk` n’est plus recommandé lors de développement de nouveaux modules. On lui préférera les fonctions spécialisées «pr_xxx»**
  - `pr_emerg(), pr_alert(), pr_crit(), pr_err(), pr_warning(), pr_notice(), pr_info()`
  - `pr_debug()` (le module doit être compilé avec le flag `-DDEBUG`, pour plus de détails https://www.kernel.org/doc/local/pr_debug.txt)

MASTER OF SCIENCE IN ENGINEERING | MSE

# Génération d’un module

# Génération d'un module

- **Linux propose deux solutions**
  - «out of tree»
    - *Cette solution consiste à placer le code source du module à l'extérieur de l'arborescence des sources du noyau Linux*
    - *Avantage: gestion des modifications très simple et indépendantes de la génération du noyau*
    - *Désavantage: n'est pas intégré dans la configuration et génération du noyau et ne pourra pas être linké statiquement avec le noyau*
  - «inside tree»
    - *Cette solution consiste à intégrer le code source du module avec les sources du noyau Linux*
    - *Le module pourra ainsi être linké statiquement avec le noyau*
    - *Il est naturellement également possible de le charger dynamiquement*

## Génération «out of tree» (I)

- Les modules Linux doivent être générés par l'intermédiaire d'un Makefile, p.ex.

```
# Part executed when called from kernel build system
ifneq ($(KERNELRELEASE),)
obj-m += mymodule.o               ## name of the generated module
mymodule-objs := skeleton.o       ## list of objets needed by that module
CFLAGS_skeleton := -DDEBUG        ## to enable printing of debugging messages

# Part executed when called from standard make in the module source directory
else
CPU := arm64
KDIR := /home/lmi/workspace/nano/buildroot/output/build/linux-5.8.6/
TOOLS := /home/lmi/workspace/nano/buildroot/output/host/usr/bin/aarch64-linux-gnu-
PWD := $(shell pwd)

all:
   $(MAKE) -C $(KDIR) M=$(PWD) ARCH=$(CPU) CROSS_COMPILE=$(TOOLS) modules

clean:
   $(MAKE) -C $(KDIR) M=$(PWD) clean
endif
```

- La commande «`make`» génère le module pour la machine cible.
- Le module généré est «`mymodule.ko`».
  «`.ko`» indique qu'il s'agit d'un *« kernel object »* pouvant être chargé dans le noyau

MASTER OF SCIENCE IN ENGINEERING
MSE

# Génération «out of tree» (II)

- **La compilation du module s'effectue en trois phases**
  1. Le Makefile du module est appelé avec la commande **make** ou **make all**
  2. Lors de ce premier appel, la variable **KERNELRELEASE** n'est pas définie. Le make va par conséquent appeler le Makefile du noyau spécifié par la variable **KDIR**.
     On doit  impérativement spécifier
     - L'architecture du processeur avec la variable **ARCH=$(CPU)** et
     - Le compilateur à utiliser avec la variable **CROSS_COMPILE=$(TOOLS)**
  3. Le Makefile du noyau a la logique pour la génération de modules et, grâce à la variable **M**, il peut obtenir la liste des sources à compiler en réinterprétant le Makefile du module pour obtenir le nom du module à générer avec la définition **obj-m**. ainsi que la liste des objets nécessaire pour construire le module avec la variable **mymodule-objs**

- **Attention**
  Un module compilé pour une version X du noyau ne pourra pas être chargé sur une version Y, «invalid module format»

# Génération «inside tree»

- **Pour être généré avec le noyau, le code source du module doit impérativement être déposé dans l'arborescence des sources du noyau Linux.**
  - Ajouter les sources dans le répertoire approprié du noyau p. ex. dans `./drivers/misc`
    - Remarque: si le module est suffisamment petit, quelques milliers de lignes de code, on ne créera qu'un seul fichier. Si celui-ci est réellement très grand, on pourra alors le partager en plusieurs fichiers et le déposer dans son propre répertoire
  - Modifier le fichier de configuration `Kconfig` pour y ajouter le nouveau module

```
config MISC_MY_MODULE
  tristate "Miscellaneous Module Skeleton"
  help
      Module skeleton for education purpose
```

  - Modifier le Makefile selon l'entrée de Kconfig

```
obj-$(CONFIG_MISC_MY_MODULE) += skeleton.o
```

  - Reconfigurer le noyau pour générer le module `(make linux-menuconfig)`
  - Régénérer le noyau `(make)`
  - Voir `./Documenation/kbuild` pour plus de détails et des exemples plus complets

# Installation / Désinstallation

MASTER OF SCIENCE
IN ENGINEERING
MSE

# Installation et désinstallation d'un module

- **L'installation et la désinstallation de module ne peuvent être réalisées qu'en ayant les droits « root ».**
- **Linux propose plusieurs outils pour la gestion des modules noyaux**
  - La commande `modinfo <module_name>.ko` nous renseigne sur le module: paramètres, licence, description, dépendances, …
  - Pour installer un module dans le noyau, peut utiliser la commande `insmod`, p. ex.

    ```
    $ insmod mymodule.ko
    ```

  - Si l'on rencontre des problèmes ou des erreurs lors de l'installation d'un module, la commande `dmesg` permet souvent d'obtenir plus de renseignements sur la cause de l'erreur.
  - Pour obtenir la liste des modules déjà installés dans le noyau, il suffit simple de le lire le contenu du fichier «`/proc/modules`» (`cat /proc/modules`) ou d'utiliser la commande `lsmod`
  - Pour désinstaller un module du noyau, il suffit d'utiliser la commande `rmmod`. Il est important de noter que cette opération n'est autorisée que si le module n'est plus utilisé, p. ex.

    ```
    $ rmmod mymodule
    ```

# Installation et désinstallation d'un module (II)

- **Si un module dépend d'autres modules, il est impératif de les avoir préalablement chargés dans le noyau Linux. Cette tâche étant relativement fastidieuse, la commande `modprobe` offre une alternative très intéressante aux commandes précédentes.**
  - Pour installer un module: `$ modprobe <module_name>`
  - Pour désinstaller un module: `$ modprobe -r <module_name>`
  - Pour charger le module souhaité, `modprobe` interprète le fichier `modules.dep` situé dans le répertoire `/lib/modules/<kernel_version>/`
  - Ce fichier est généré par les Makefile du noyau. Pour inclure son propre module dans ce fichier, il suffit de compléter le Makefile du module avec l'instruction suivante, p. ex.

```
MODPATH := $(HOME)/workspace/buildroot/output/target # production mode
install:
	$(MAKE) -C $(KDIR) M=$(PWD) INSTALL_MOD_PATH=$(MODPATH) modules_install
```

    - La variable `INSTALL_MOD_PATH` indique le chemin du répertoire où est placé le root file system
    - En mode de développement sous NFS:

```
MODPATH := /home/lmi/workspace/nano/rootfs
$ sudo make install
```

MASTER OF SCIENCE IN ENGINEERING | MSE

# Paramètres d’un module

MASTER OF SCIENCE IN ENGINEERING
MSE

# Paramètres d'un module

- Linux permet de passer des paramètres à un module lors de son chargement.
- Le code ci-dessous montre les adaptations à apporter au squelette du module pour supporter deux paramètres

```
/* skeleton.c */
#include <linux/module.h>      /* needed by all modules */
#include <linux/init.h>        /* needed for macros */
#include <linux/kernel.h>      /* needed for debugging */
#include <linux/moduleparam.h>   /* needed for module parameters */

static char* text= "dummy help";
module_param(text, charp, 0);

static int elements= 1;
module_param(elements, int, 0);

static int __init skeleton_init(void)
{
    pr_info ("Linux module skeleton loaded\n");
    pr_info ("text: %s\nelements: %d\n", text, elements);
    return 0;
}

static void __exit skeleton_exit(void)
{
    pr_info ("Linux module skeleton unloaded\n");
}

module_init (skeleton_init);
module_exit (skeleton_exit);

MODULE_AUTHOR ("Daniel Gachet <daniel.gachet@hefr.ch>");
MODULE_DESCRIPTION ("Module skeleton");
MODULE_LICENSE ("GPL");
```

MASTER OF SCIENCE | IN ENGINEERING
MSE

# Paramètres d'un module (II)

- **La macro `module_param` permet de définir des paramètres dans un module. Cette macro est disponible depuis l'interface «`linux/moduleparam.h`»**

  1er argument : indique le nom du paramètre et de la variable dans le module

  2ème argument : indique le type de paramètre (`byte, short, ushort, int, uint, long, ulong, charp ou bool`).
  Ce paramètre est contrôlé lors de la compilation.

  3ème argument : spécifie les droits d'accès au fichier contenant les paramètres (`/sys/module/<module_name>/parameters/<param>` ).
  La valeur 0 indique que ce fichier n'existe pas.

- **Le passage de paramètres s'effectue lors du chargement du module**
  - Avec `insmod:`

    ```
    $ insmod mymodule.ko elements=-1 'text="bonjour le monde"'
    ```
  - Avec `modprobe`, il suffit d'inclure les paramètres du module dans le fichier `/etc/modprobe.conf`, p. ex.

    ```
    options mymodule elements=5 text="salut les copains..."
    ```
  - Si le module est linké statiquement avec le noyau, on peut passer les paramètres dans la ligne de commande du noyau lors de son lancement, p. ex.

    ```
    mymodule.elements=10
    ```

MASTER OF SCIENCE IN ENGINEERING
MSE

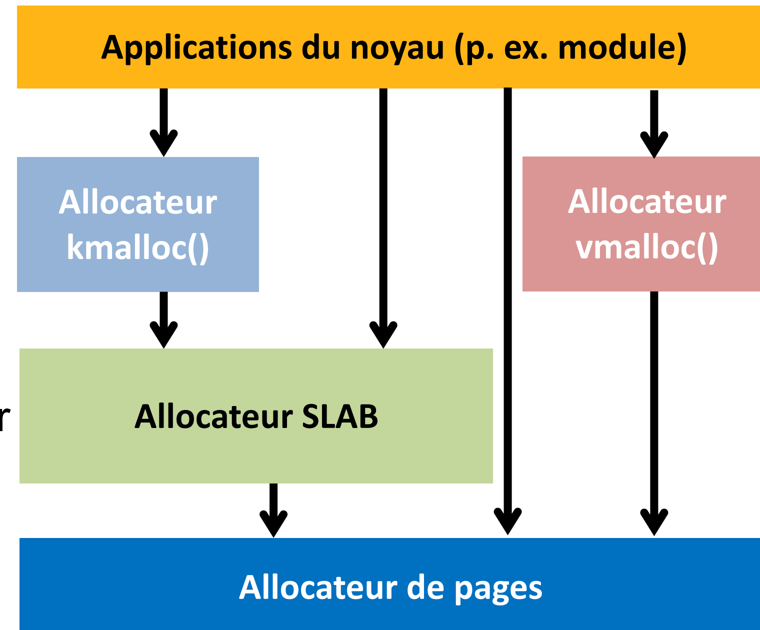
# Allocation dynamique de la mémoire

MASTER OF SCIENCE IN ENGINEERING
MSE

# Gestion de la mémoire dynamique par le noyau Linux

- **Il est courant que les modules noyaux doivent faire appel à des allocations dynamiques de la mémoire pour la création d'objets ou de structures de données.**

- **Linux dispose d'une infrastructure très puissante pour couvrir ces besoins. Cette infrastructure est répartie sur trois niveaux.**
  - Les allocateurs `kmalloc` et `vmalloc` offrant des services de base aux applications fonctionnant dans le noyau. De ces deux allocateurs, on préférera généralement utiliser `kmalloc` pour ses performances.
  - Les allocateurs SLAB permettent de créer de caches, lesquels contiennent des objets de même taille (voir `/proc/slabinfo`).
    - *Il existe différentes implémentations de ces allocateurs et peuvent être choisies lors de la configuration du noyau*
  - Les allocateurs de pages permettent d'allouer de grandes zones mémoires, bien que dépendant de l'architecture, usuellement des blocs de 4KiB.




MASTER OF SCIENCE IN ENGINEERING
MSE

# Allocateur kmalloc

- **`kmalloc` est l'allocateur de mémoire d'usage général pour les modules du noyau Linux**
  - Il permet d'allouer des objets jusqu'à 128 MiB
  - Pour les petites tailles, il s'appuie sur les caches génériques SLAB, nommés `kmalloc-XXX` ou `size-xxx` (voir `/proc/slabinfo`)
  - Pour de plus grandes tailles, il utilise sur l'allocateur de pages
  - **`kmalloc`** garantit que la zone allouée est physiquement contigüe

- **Signification des drapeaux (flags)**
  - **`GFP_KERNEL`** : allocation mémoire standard
  - **`GFP_ATOMIC`**: permet d'allouer de la mémoire dans du code ne pouvant être interrompu (interrupt handler ou sections critiques)
  - **`GFP_DMA`**: permet d'allouer de la mémoire pouvant être utilisée pour des transferts DMA, on préférera cependant les services de l'interface `<linux/dma-mapping.h>`, p.ex. `dma_alloc_coherent()`
  - Plus de détails dans `<linux/gfp.h>`

- **`kmalloc` doit être utilisé comme l'allocateur principal à moins qu'il existe une bonne raison d'utiliser un autre**

MASTER OF SCIENCE IN ENGINEERING

MSE

# Allocateur kmalloc (II)

- **Fonctions pour l'allocation de mémoire dans `<linux/slab.h>`**

  `void* kmalloc (size_t size, gfp_t flags)` : alloue `size` octets et retourne le pointeur sur le bloc de données (adresse virtuelle)

  `void* kzalloc (size_t size, gfp_t flags)` : alloue et initialise à zéro le bloc

  `void* kcalloc (size_t n, size_t size, gfp_t flags)` : alloue de la mémoire pour `n` éléments de taille `size` et initialise à zéro le contenu

  `void* krealloc (const void* p, size_t new_size, gfp_t flags)` : change la taille du bloc pointé par `p` pour la taille `new_size`. Le bloc est réalloué et son contenu copié.

- **Libération de mémoire**

  `void kfree (const void* objp)` : libère la mémoire allouée

- **Exemple**

```
char* buffer = kzalloc (1000, GFP_KERNEL);
if (buffer == NULL) /* treat error… */;
// ...
kfree (buffer);
buffer = 0;
```

MASTER OF SCIENCE IN ENGINEERING

MSE

# Allocateur devm_kmalloc (III)

- **Pour la réalisation de pilotes de périphériques, il existe également des services avec une libération automatique des blocs lorsque le module ou le périphérique est extrait.**

```
void* devm_kmalloc (struct device* dev, size_t size, gfp_t flags);

void* devm_kzalloc (struct device* dev, size_t size, gfp_t flags);

void* devm_kcalloc (struct device* dev, size_t n, size_t size, gfp_t flags);
```

- **Pour une libération immédiate du bloc mémoire**

```
void devm_kfree (struct device* dev, const void* objp);
```

MASTER OF SCIENCE IN ENGINEERING | MSE

# Bibliothèques et fonctions utiles

MASTER OF SCIENCE
IN ENGINEERING
MSE

# Bibliothèques

- **A l'intérieur du noyau Linux on ne dispose pas des bibliothèques standard C. Par contre, le noyau fournit toute une série de bibliothèques et de fonctions fort utile…**
- **String dans `<linux/string.h>`**
  - Relié à la mémoire: `memset, memcpy, memmove, memscan, memcmp, memchr`
  - Relié aux strings: `strcpy, strcat, strcmp, strchr, strrchr, strlen` et d'autres variantes
  - Allocation et copie de blocs mémoire: `kmemdump`
  - Allocation et copie de strings: `kstrdup, strrndup`
- **Conversion de string dans `<linux/kernel.h>`**
  - Conversion de strings en entiers: `kstrtoul, kstrtol, kstrto*`
  - Fonctions sur les strings: `sprintf, sscanf`

MASTER OF SCIENCE IN ENGINEERING
MSE

# Bibliothèques (II)

- **Listes chaînées `<linux/list.h>`** (simple doubly linked list)
  - Très pratique et utilisées à des milliers d'endroits dans le noyau
  - Ajouter un membre `struct list_head` dans la structure des éléments participant à la liste chaînée. On le nommera généralement `node`.
  - Créer une liste. Si la liste est globale on utilisera la macro `LIST_HEAD`, si la liste fait partie d'une structure on définira un élément `struct list_head` et on l'initialisera avec la macro `INIT_LIST_HEAD`
  - Pour manipuler les éléments de la liste chaînée, on pourra utiliser
    - Ajouter des éléments: `list_add (), list_add_tail ()`
    - Supprimer, déplacer ou remplacer des éléments: `list_del (), list_move (), list_move_tail (), list_replace ()`
    - Tester la liste: `list_empty ()`
    - Itérer sur la liste: `list_for_each_*()` de la famille des macros
  - Il existe également des variantes sûres «safe» de ce méthodes

MASTER OF SCIENCE IN ENGINEERING
MSE

# Exemple de liste chaînée

```c
// definition of a list element with struct list_head as member
struct element {
    // some members
    struct list_head node;
};

// definition of the global list
static LIST_HEAD (my_list);

// allocate on element and add it at the tail of the list
void alloc_ele () {
    struct element* ele
    ele = kzalloc(sizeof(*ele), GFP_KERNEL); // create a new element
    if (ele != NULL)
      list_add_tail(&ele->node, &my_list); // add element at the end of the list
}

// process all elements of the list
void process_all() {
    struct element* ele;
    list_for_each_entry(ele, &my_list, node) {// iterate over the whole list
      // do something with ele
    }
}
```

MASTER OF SCIENCE IN ENGINEERING
MSE

# Accès aux entrées/sorties

MASTER OF SCIENCE IN ENGINEERING
MSE

# Types d'entrées/sorties

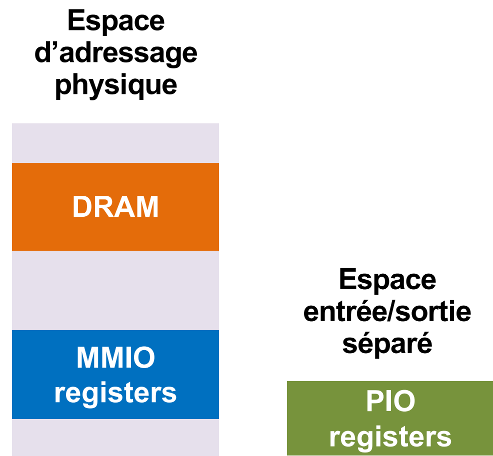
- **Selon les processeurs, on distingue deux types d'entrées/sorties**
  - **MMIO** - memory-mapped I/O (le plus courant)
    - Une seule zone d'adresse pour la mémoire et les périphériques d'entrées/sorties
    - Instructions usuelles pour accéder les périphériques d'entrées/sorties
    - `cat /proc/iomem` permet d'obtenir les zones d'entrées/sorties
    - La méthode `request_mem_region` permet d'informer le noyau sur une zone

```
struct resource* request_mem_region (unsigned long start,
                                     unsigned long len,
                                     char* name);
```

    - La méthode `release_mem_region` permet de libérer une zone

```
void release_mem_region (unsigned long start,  unsigned long len);
```

  - **PIO** - port I/O (spécialement sur les machines Intel x86)
    - Différentes zones d'adresse pour la mémoire et les périphériques d'entrées/sorties
    - Instructions spécialisées pour accéder les périphériques d'entrées/sorties
    - `cat /proc/ioports` permet d'obtenir les zones d'entrées/sorties




MASTER OF SCIENCE IN ENGINEERING
MSE

# Accès aux entrées/sorties en zone mémoire

- **Dans le noyau Linux, l'accès aux entrées/sorties placées en mémoire physique n'est pas possible directement. Il faut préalablement créer une zone mémoire virtuelle pour y mapper les entrées/sorties.**
  - L'interface `<linux/io.h>` fournit les services nécessaires
  - La méthode `ioremap` permet de mapper dans la mémoire virtuelle du noyau les entrées/sorties souhaitées

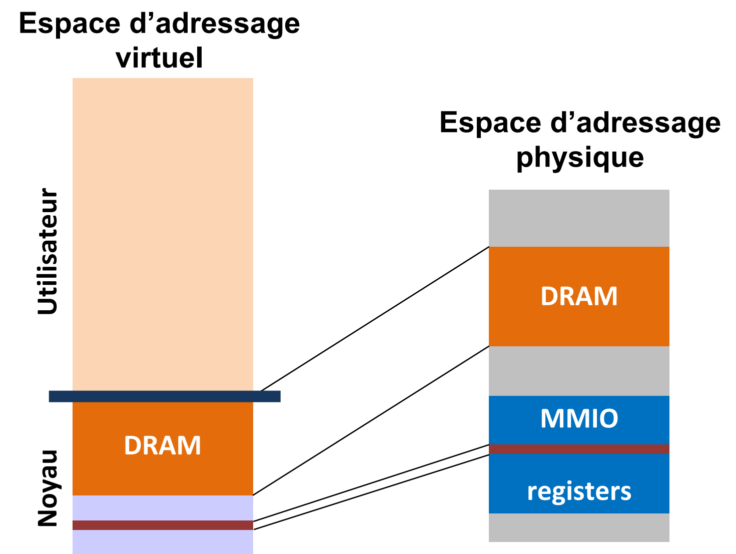
    ```
    void* ioremap (unsigned long phys_addr, unsigned long size);
    ```

  - La méthode `iounmap` permet de libérer cette zone mémoire

    ```
    void iounmap (unsigned long address);
    ```

- **Attention**

  il faut impérativement contrôler que `ioremap` retourne une valeur **non `NULL`** !



# Accès aux entrées/sorties en zone mémoire (II)

- **Il n'est pas recommandé d'utiliser l'adresse retournée par la fonction « `ioremap` » comme pointeur sur les registres.**
- **Il est plutôt conseillé d'utiliser les macros d'accès suivantes:**

```
type read[b, w, l, q] (void* addr);

void write[b, w, l, q] (type value, void* addr);
```

**ou**

```
type ioread[8, 16, 32, 64] (const volatile void* addr);

void iowrite[8, 16, 32, 64] (type value, volatile void* addr);
```

**ou pour des accès répétés**

```
void ioread[8, 16, 32, 64]_rep
    (const volatile void* addr, void* buffer, unsigned count);

void iowrite[8, 16, 32, 64]_rep
    (volatile void* addr, const void* buffer, unsigned count);
```

MASTER OF SCIENCE IN ENGINEERING
MSE

# Accès aux entrées/sorties en zone mémoire (III)

- **Il existe également une interface gérée**

```
void* devm_ioremap(
        struct device* dev,
        resource_size_t offset,
        resource_size_t size);


void devm_iounmap(
        struct device* dev,
        void* addr);


void* devm_ioremap_resource(
        struct device* dev,
        struct resource* res);
```

Ce service prend en charge la réservation et le mapping de la ressource physique

MASTER OF SCIENCE IN ENGINEERING
MSE

# Threads dans le noyau

MASTER OF SCIENCE IN ENGINEERING | MSE

# Création de threads dans le noyau

- **Certaines applications nécessitent l'usage de plusieurs threads. L'interface `<linux/kthread.h>` propose des services simplifiés pour leur création et leur destruction.**
  - La macro **`kthread_run`** permet de créer simplement un thread

    ```
    struct task_struct* kthread_run
        (int (*threadfn)(void *data), data, namefmt,...);
    ```

    - `threadfn` *est la fonction implémentant le corps du thread*
    - `data` *est un pointeur sur des données passées au thread*
    - `namefmt` *est le nom du thread passé sous la forme d'un* `printf`
  - Pour stopper un thread, il suffit d'utiliser la fonction **`kthread_stop`**

    ```
    int kthread_stop (struct task_struct *k);
    ```

  - La fonction **`kthread_should_stop`** offre un service au thread pour tester périodiquement s'il doit s'arrêter.

    ```
    int kthread_should_stop (void);
    ```

# Création de threads dans le noyau (II)

- **Le corps d'un thread créé avec l'interface kthread.h prend la forme suivante**

```
int thread (void* data)
{
    while (!kthread_should_stop()) {
        /* do something... */
    }
    return 0;
}
```

- **La commande `ps` permet de lister tous les threads et processus**
- **La commande `cat /proc/<pid>/stat` permet d'afficher des informations sur l'état du processus**
  - On trouve facilement sur Internet de petits programmes offrant une représentation plus conviviale, p. ex. http://www.brokestream.com/procstat.html

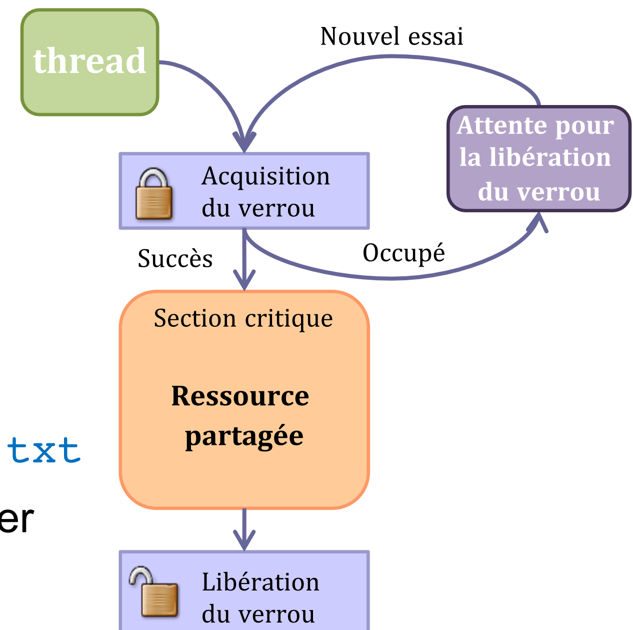
MASTER OF SCIENCE IN ENGINEERING

MSE

# Accès concurrents

# Ressources partagées et sections critiques

- **Tout système temps-réel est confronté un jour ou l'autre aux problèmes d'accès concurrents sur des ressources partagées entre plusieurs threads. Il en est de même sous Linux et plus particulièrement dans son noyau.**
- **L'idéal est d'éviter au maximum toute variable globale ou partagée.**
- **Si ce n'est pas possible, Linux propose 3 mécanismes principaux pour protéger ces sections critiques**
  - Les mutexes
  - Les spinlocks
  - Les accès atomiques
- **Avec l'utilisation de mutexes ou de spinlocks, le danger de créer des deadlocks est latent.**
  - Linux propose des outils de validation. Plus de détails sous `Documentation/lockdep-design.txt`
  - Dans la mesure de possible on peut essayer d'utiliser des algorithmes libre de loquets, tel que RCU (Read Copy Update). Plus de détails sous [http://en.wikipedia.org/wiki/Read-copy-update](http://en.wikipedia.org/wiki/Read-copy-update)




MASTER OF SCIENCE IN ENGINEERING
MSE

# Les mutexes

- **L'interface `<linux/mutex.h>` propose les services reliés aux mutuxes**
  - Initialisation statique d'une mutex
    ```
    DEFINE_MUTEX (name);
    ```
  - Initialisation dynamique d'une mutex
    ```
    void mutex_init (struct mutex* lock);
    ```
  - Accès à la ressource critique. Si l'accès n'est pas autorisé, le thread sera bloqué. Attention: ce service ne pas être interrompu, empêchant la destruction du thread.
    ```
    void mutex_lock (stuct mutex* lock);
    ```
  - Accès à la ressource critique, mais interruptible par le signal fatal (SIGKILL). Une valeur non zéro est retourné si le verrou n'a pas été tenu.
    ```
    int mutex_lock_killable (stuct mutex* lock);
    ```
  - Accès à la ressource critique, mais interruptible par tous les signaux
    ```
    int mutex_lock_interruptible (stuct mutex* lock);
    ```
  - Accès à la ressource, mais sans attente (non zéro si pas disponible)
    ```
    int mutex_trylock (struct mutex* lock);
    ```
  - Libération de la ressource critique
    ```
    void mutex_unlock (struct mutex* lock);
    ```

# Les spinlocks

- **Les spinlocks permettent de protéger des sections critiques pour des parties de code ne pouvant pas être mis en mode sommeil** *(interrupt handlers)*.
  - L'utilisation de spinlocks demande une extrême précaution
  - Les spinlocks déclenchent le mécanisme de préemption du noyau
  - Les spinlocks restent en attente actives jusqu'à ce que l'accès soit libre
- **L'interface `<linux/spinlock.h>` propose les services reliés aux spinlocks**
  - Initialisation statique d'un spinlock

    ```
    DEFINE_SPINLOCK (name);
    ```
  - Initialisation dynamique d'un spinlock

    ```
    void spin_lock_init (spinlock_t* lock);
    ```
  - Pour un verrouillage dans le contexte de threads (interruptions autorisées)

    ```
    void spin_[un]lock (spinlock_t* lock);
    ```
  - Pour un verrouillage entre threads et interruptions (interruptions déclenchées)

    ```
    void spin_lock_irqsave/_unlock_irqrestore
          (spinlock_t *lock, unsigned long flags);
    ```
  - Pour un verrouillage entre threads et interruptions software (interruptions matérielles autorisées)

    ```
    void spin_[un]lock_bh (spinlock_t* lock);
    ```

MASTER OF SCIENCE IN ENGINEERING
MSE

# Les variables atomiques

- **Les variables atomiques peuvent être d'une grande utilité si la ressource partagée est une valeur entière. Il important de noter que l'opération `n++` n'est pas atomique sur tous les processeurs (p. ex. ARM).**
- **L'interface `<linux/atomic.h>` propose les services reliés aux variables atomiques**
  - Le type `atomic_t` représente un nombre entier signé (minimum 24 bits)
  - Opérations pour lire et écrire un compteur
    ```
    atomic_set (atomic_t* v, int i);
    int atomic_read (atomic_t *v);
    ```
  - Opérations sans valeurs de retour
    ```
    void atomic_[inc/dec] (atomic_t* v);
    void atomic_[add/sub] (int i, atomic_t* v);
    ```
  - Opérations retournant la nouvelle valeur
    ```
    int atomic_[inc/dec] _and_return (atomic_t* v);
    int atomic_[add/sub]_and_return (int i, atomic_t* v);
    ```
  - Il existe encore d'autres opérations…

MASTER OF SCIENCE IN ENGINEERING
MSE

# Les opérations atomiques sur les bits

- **L'interface `<linux/bitops.h>` propose des opérations très efficaces pour manipuler des bits.**
- **Sur la plupart des plateformes, ils s'appliquent sur des type `unsigned long`.**
  - Opérations pour poser, effacer et changer a bit donné

```
void set_bit (int nr, unsigned long * addr);
void clear_bit (int nr, unsigned long * addr);
void change_bit (int nr, unsigned long * addr);
```

  - Opération pour tester un bit

```
int test_bit (int nr, unsigned long *addr);
```

  - Opérations pour tester et modifier (retourne la valeur avant modification)

```
int test_and_set_bit (int nr, unsigned long *addr);
int test_and_clear_bit (int nr, unsigned long *addr);
int test_and_change_bit (int nr, unsigned long *addr);
```

# Mise en sommeil

MASTER OF SCIENCE IN ENGINEERING | MSE

## Mise en sommeil

- **Dans les applications temps-réel il existe une multitude de raisons pour mettre en sommeil un processus ou un thread.**
- **La plus courante est l'attente de données d'un périphérique d'entrées/sorties.**



MASTER OF SCIENCE IN ENGINEERING
MSE

# Les waitqueues

- **Les waitqueues permettent de mettre en sommeil un processus ou un thread et de le réveiller lors de l'arrivée de l'événement.**

- **L'interface `<linux/wait.h>` propose les services reliés aux waitqueues**
  - Initialisation statique d'une waitqueue

    ```
    DECLARE_WAIT_QUEUE_HEAD(queue);
    ```
  - Initialisation dynamique d'une waitqueue

    ```
    void init_waitqueue_head (wait_queue_head_t* queue);
    ```

- **Quelques macros pour mettre en sommeil un thread**
  - Opération pour mettre en sommeil le thread jusqu'à la queue soit notifiée et que la condition C soit vraie. Attention: ce service ne peut pas être interrompu, empêchant la destruction du processus en espace utilisateur.

    ```
    wait_event (queue, condition);
    ```
  - Idem, mais peut être interrompu par le signal «SIGKILL».
    Si le thread a été interrompu, la valeur -ERESTARTSYS est retournée.

    ```
    int wait_event_killable (queue, condition);
    ```
  - Idem, mais peut être interrompu par n'importe quel signal.
    Si le thread a été interrompu, la valeur -ERESTARTSYS est retournée.

    ```
    int wait_event_interruptible (queue, condition);
    ```

# Les waitqueues (II)

- Opération pour mettre en sommeil le thread jusqu'à la queue soit notifiée et que la condition C soit vraie ou bien que le temps soit écoulé.
  La valeur 0 est retournée si le temps est écoulé. (timeout en *jiffies, 1-10ms*)

```
int wait_event_timeout (queue, condition, timeout);
```

- Idem, mais peut être interrompu par n'importe quel signal.
  La valeur 0 est retournée si le temps est écoulé, si le thread a été interrompu, la valeur -ERESTARSYS est retournée, sinon une valeur positive.

```
int wait_event_interruptible_timeout
     (queue, condition, timeout);
```

- **Quelques macros pour réveiller le thread**
  - Opération pour réveiller tous les processus dans la queue

```
wake_up(&queue);
```

  - Opération pour réveiller que les processus ininterruptibles de la queue.

```
wake_up_interruptible (&queue);
```

- **D'autres opérations sont disponibles dans l'interface.**

# Gestion des interruptions

MASTER OF SCIENCE IN ENGINEERING

MSE

# Traitement des interruptions sous Linux

- La figure ci-dessous montre schématiquement le traitement des interruptions par le noyau Linux pour les processeurs ARM



MASTER OF SCIENCE IN ENGINEERING
MSE

# Installation des routines de traitement des interruptions

- L'interface `<linux/interrupt.h>` propose deux fonctions pour l'enregistrement et l'effacement des routines de traitement d'interruptions *(interrupt handlers)*

```
int request_irq (unsigned int irq,
                 irq_handler_t handler,
                 unsigned long flags,
                 const char *dev_name,
                 void *dev_id);

void free_irq (unsigned int irq, void *dev_id);
```

```
irq = vecteur d'interruption (numéro)
handler = routine de traitement des interruptions
flags = fanions de gestion des interruptions
 IRQF_DISABLED – garde irqs déclenché lors de l'appel de la routine de traitement
 IRQF_SAMPLE_RANDOM – irq est utilisée pour la génération de nombres aléatoires
 IRQF_SHARED – permet de partager l'irq avec plusieurs périphériques
 IRQF_TIMER – fanion pour marquer cette interruption comme timer
 IRQF_TRIGGER_<xx> – fanion pour sélectionner le trigger (xx: FALLING, RISING, …)
dev_name = nom du périphérique d'interruption
dev_id = paramètre spécifique à l'application(doit impérativement être non nul
         si l'interruption est partagée IRQF_SHARED)
```

# Routine de traitement des interruptions (interrupt handler)

- **La routine de traitement des interruptions prend la forme suivante**

```
irqreturn_t short_interrupt(int irq, void *dev_id)
{
    /* do something...*/
    return IRQ_HANDLED;
}
```

**`irq = vecteur ayant levé l'interruption`**

**`dev_id = paramètre spécifique à l'application fournie lors de l'enregistrement de la routine de traitement`**

- **Si la routine a été en mesure de traiter l'interruption, `IRQ_HANDLED` sera retourné, dans le cas contraire, la valeur de retour devra être `IRQ_NONE`**
- **La valeur de retour est utilisée par le noyau pour éliminer les interruptions parasites** *(spurious interrupts)*

MASTER OF SCIENCE IN ENGINEERING
MSE

# Etat des Interruptions

- La commande « `cat /proc/interrupts` » fournit des informations très intéressantes sur l'état des interruptions avec le nom associé au vecteur d'interruption.

```
# cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
  1:          0          0          0          0     GICv2  25 Level     vgic
  3:        359        386        482        457     GICv2  30 Level     arch_timer
  4:          0          0          0          0     GICv2  27 Level     kvm guest timer
  8:      38452          0          0          0     GICv2  92 Level     sunxi-mmc
  9:        461          0          0          0     GICv2  93 Level     sunxi-mmc
 10:         43          0          0          0     GICv2  94 Level     sunxi-mmc
 11:          0          0          0          0     GICv2 103 Level     musb-hdrc.1.auto
 12:          0          0          0          0     GICv2 104 Level     ehci_hcd:usb1
 13:          0          0          0          0     GICv2 105 Level     ohci_hcd:usb3
 14:          0          0          0          0     GICv2 110 Level     ehci_hcd:usb2
 15:          0          0          0          0     GICv2 111 Level     ohci_hcd:usb4
 19:         21          0          0          0     GICv2  63 Level     1c25000.ths
 22:          0          0          0          0     GICv2 114 Level     eth0
 25:         84          0          0          0     GICv2  32 Level     ttyS0
 26:          0          0          0          0     GICv2  72 Level     1f00000.rtc
 29:          0          0          0          0     GICv2 152 Level     arm-pmu
 30:          0          0          0          0     GICv2 153 Level     arm-pmu
 31:          0          0          0          0     GICv2 154 Level     arm-pmu
 32:          0          0          0          0     GICv2 155 Level     arm-pmu
103:          0          0          0          0  sunxi_pio_edge  38 Edge 1c0f000.mmc cd
IPI0:       713       1050      35946        629       Rescheduling interrupts
IPI1:        15        492        484        484       Function call interrupts
IPI2:         0          0          0          0       CPU stop interrupts
IPI3:         0          0          0          0       CPU stop (for crash dump) interrupts
IPI4:         0          0          0          0       Timer broadcast interrupts
IPI5:         0          0          0          0       IRQ work interrupts
IPI6:         0          0          0          0       CPU wake-up interrupts
Err:          0
```

MASTER OF SCIENCE IN ENGINEERING | MSE

# Tâches usuelles d’une routine de traitement d’interruptions

- **Les tâches typiques d’une routine de traitement**
  - Quittancer la notification de l’événement sur le périphérique ayant levé l’interruption, afin d’éviter que les interruptions ne soient plus générées ou que l’interruption soit levée continuellement *(deadlock)*
  - Lire ou écrire les données du ou vers le périphérique
  - Réveiller l’éventuel thread en attente de l’événement, afin qu’il puisse terminer le traitement de l’information et son opération, typiquement avec une waitqueue

```
wake_up_interruptible (&queue);
```

MASTER OF SCIENCE IN ENGINEERING
MSE

# Contraintes et limitations

- **La programmation des routines de traitement des interruptions doit toujours être réalisée avec beaucoup de soin. Il en va de même dans le noyau Linux.**
  - Le temps de traitement dans une interruption doit être limité au minimum. Si le traitement est lourd et complexe, il peut être délégué à une softirq, une workqueue ou depuis la version 2.6.30 à un thread
  - Il n'est pas possible de transférer des données avec des applications de l'espace utilisateur
  - Il n'est pas possible d'effectuer des opérations nécessitant des mises en sommeil
- **L'utilisation de fonctions pour autoriser et/ou bloquer les interruptions (`disable_irq/enable_irq`) doit être évitée au maximum**

MASTER OF SCIENCE IN ENGINEERING

MSE

# Traitement des interruptions par thread

- **Linux propose depuis la version 2.6.30 un mécanisme permettant de traiter les interruptions dans une routine appelée par un thread. Ce mécanisme permet d'avoir des traitements longs non bloquants.**

```
int request_threaded_irq (unsigned int irq,
                          irq_handler_t handler,
                          irq_handler_t thread_fn,
                          unsigned long flags,
                          const char * dev_name,
                          void * dev_id);
```

```
irq       = vecteur d'interruption (numéro)
handler   = routine appelée lorsque l'interruption est levée. Pour passer la main
            au thread, cette routine retournera la valeur IRQ_WAKE_THREAD
thread_fn = routine de traitement appelée par un thread.
flags     = fanions de gestion des interruptions
dev_name  = nom du périphérique d'interruption
dev_id    = paramètre spécifique à l'application(doit impérativement être non nul)
```

MASTER OF SCIENCE | IN ENGINEERING
MSE

# Interface gérée

- Il existe également une interface gérée

```
devm_request_irq()
devm_free_irq()
devm_request_threaded_irq()
```

MASTER OF SCIENCE
IN ENGINEERING
MSE