

Linux Embarqué

Séance 2 : Processus, mémoire et modules

Laurent Fiack

Bureau D212 – laurent.fiack@ensea.fr

Menu du jour

Les processus

- Les états des processus
- Création de processus
- Ordonnancement

Mémoire virtuelle

- Mémoire virtuelle paginée
- Accès à la mémoire

Les modules

- Compilation
- Exemple
- Paramètres

Character device driver

Implémentation

Utilisation

read, write, ioctl

Espaces mémoire

procfs

Implémentation

Les entrées/sorties

Port I/O

MMIO

Les interruptions

Implémentation

Timers

Interruptions longues

Les processus

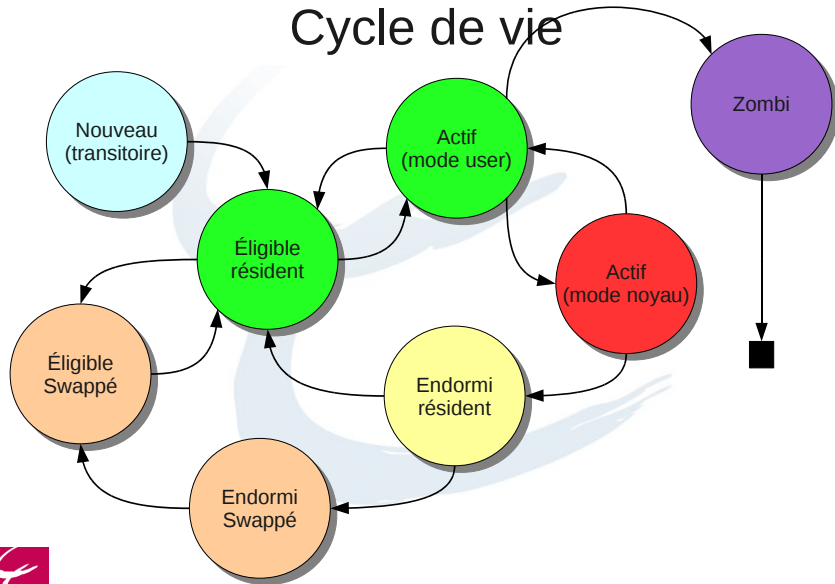
Introduction

- Un processus est lancé par un autre processus (sauf init).
- Les processus sont identifiés par un PID (Process ID)
- Chaque processus possède un parent (sauf init).
- Le PID du parent est le PPID (Parent PID)
- Lister les processus en cours :
 - `ps`
 - `ps -ef`
 - `ps -ef |grep truc`
 - `pstree`
 - `pstree -p`
 - `top`

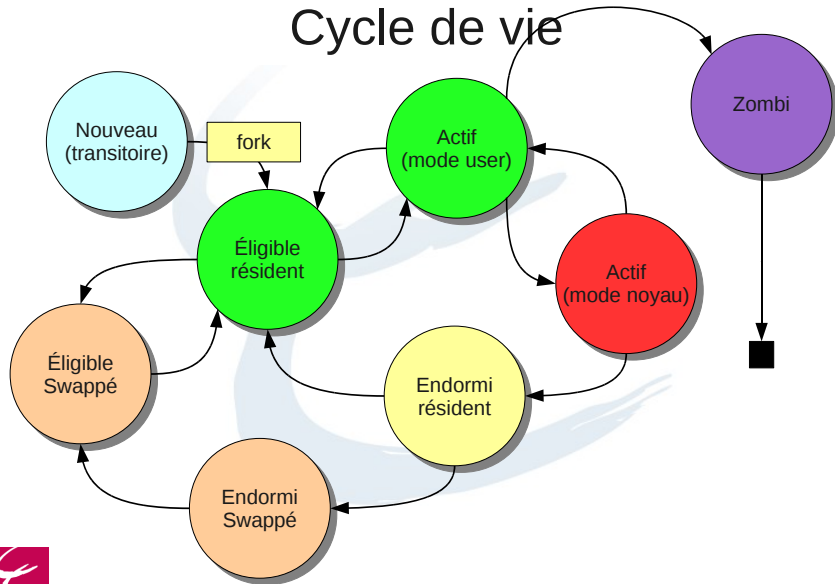
États d'un processus

- `TASK_RUNNING` : en cours d'exécution ou en attente d'être exécuté
- `TASK_INTERRUPTIBLE` : suspendu en attente d'une condition (interruption matérielle, signal, ...)
- `TASK_ZOMBIE` : exécution terminée, mais le processus parent n'a pas réclamé les informations sur son fils
- `TASK_DEAD` : état final, en cours de suppression par le système

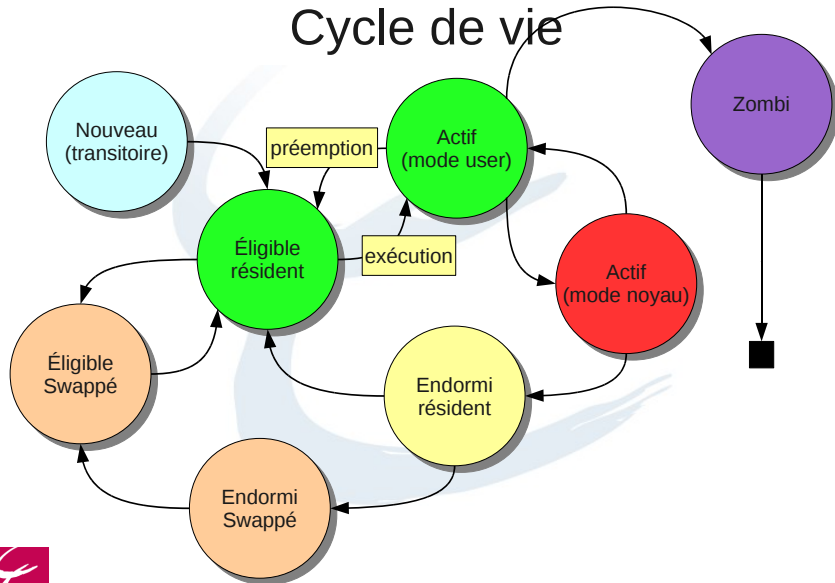
Cycle de vie



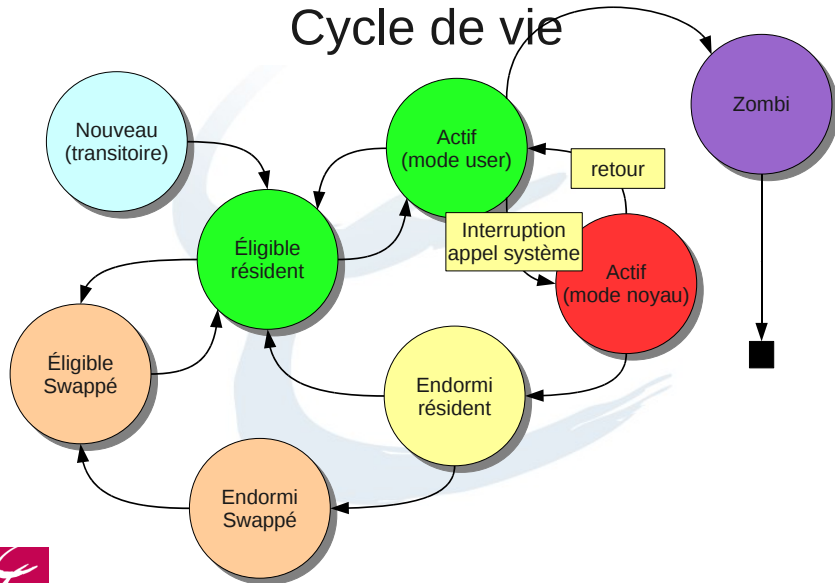
Cycle de vie



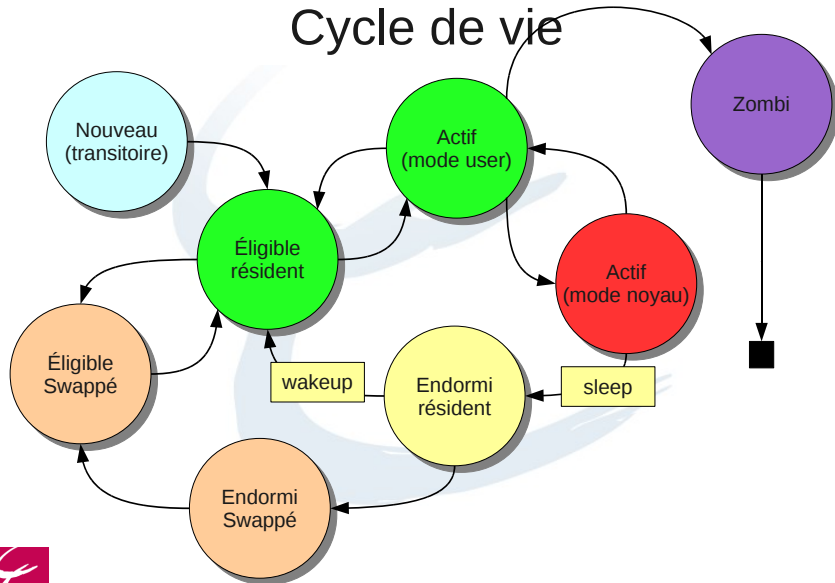
Cycle de vie



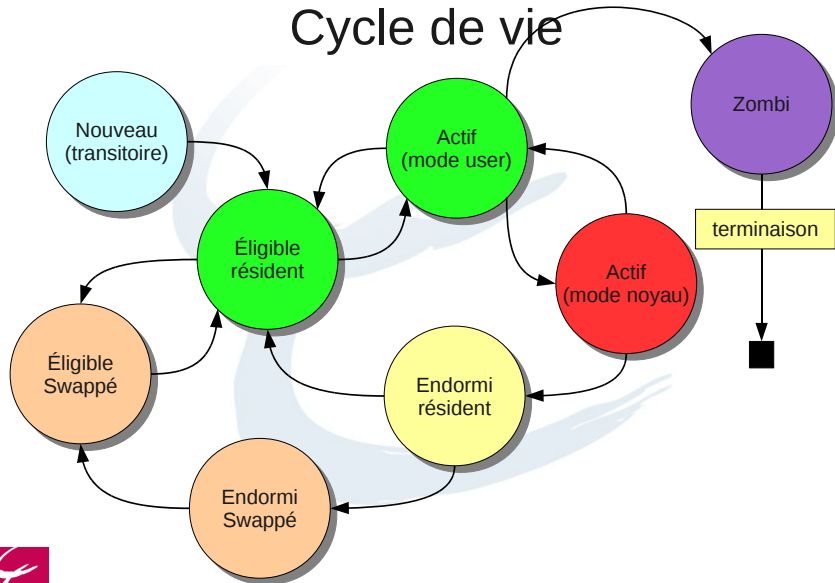
Cycle de vie



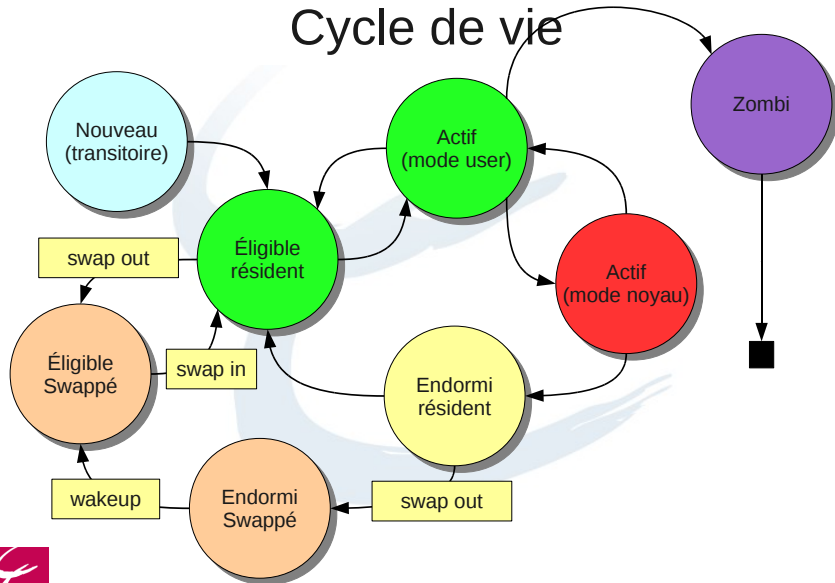
Cycle de vie



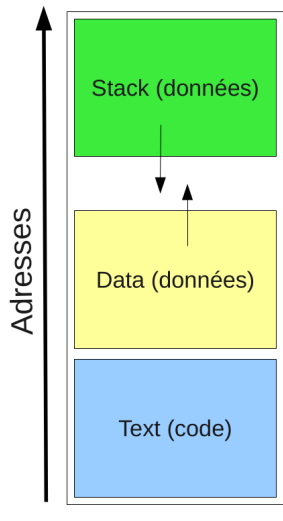
Cycle de vie



Cycle de vie



Création de processus : fork()



```
int main (int argc, char ** argv) {  
    int pid, status;  
  
    pid = fork();  
  
    if (pid != 0) {  
        printf("Je suis le père %d et mon fils est %d\n",  
               getpid(), pid);  
        wait(&status);  
    }  
    else {  
        printf("Je suis le fils %d et mon père est %d\n",  
               getpid(), getppid());  
    }  
  
    return 0;  
}
```

Appel système fork()

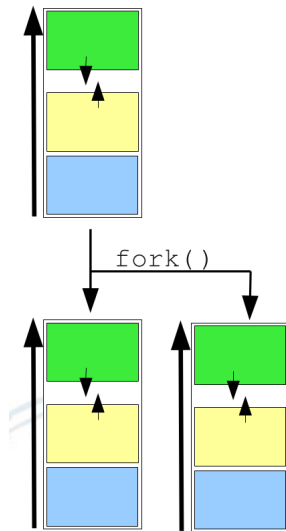
Copie le processus courant

■ Sont hérités :

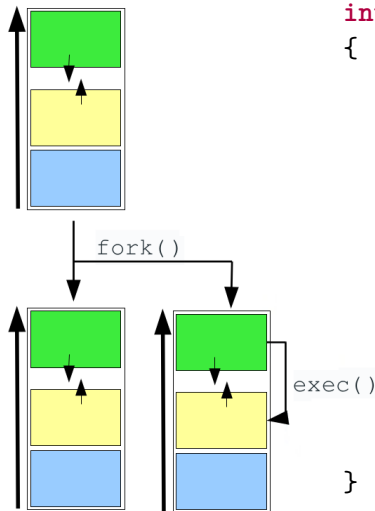
- les variables initialisées
- l'environnement
- les fichiers ouverts
- le traitement des signaux
- le GID (Group Identifier)

■ Ne sont pas hérités :

- Le PID (un nouveau est attribué)
- Le PPID (Parent PID)



Remplacement de code : exec()



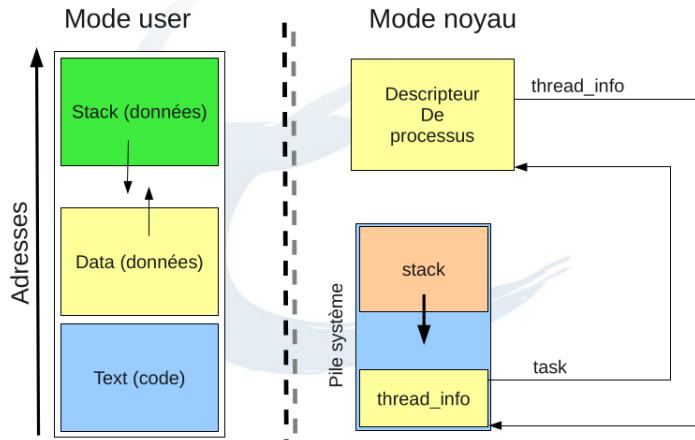
```
int main(int argc, char **argv)
{
    int pid, status;
    pid = fork();
    if(pid != 0) {
        wait(&status);
    }
    else {
        printf("je suis le fils %d\n", getpid());
        execlp("/usr/bin/ls", "ls", NULL);
        printf("ceci ne sera jamais affiché");
    }
    return 0;
}
```

Structures associées

À chaque processus est associé :

- Un contexte utilisateur : le processus tel que vu par l'utilisateur
- Un contexte Système : le processus tel que vu par le noyau, décomposé en deux parties
 - Un descripteur de processus
 - Une pile système

Structures associées



Pile système

- Sert à stocker le contexte matériel du processus
 - Ensemble des données contenues dans les registres nécessaires à l'exécution du processus
 - Les processus ne peuvent pas partager les registres, lorsqu'un changement de processus survient, le contexte matériel est sauvegardé dans la pile système.
 - Lorsque le processus est de nouveau actif, le contexte matériel est restauré depuis la pile
 - Le descripteur de processus contient un pointeur vers la pile système

Ordonnancement

- Le système est multi-tâches : plusieurs processus s'exécutent en même temps
- La simultanéité est simulé par un partage du temps de calcul
 - On stoppe l'exécution d'un processus pour en exécuter un autre
 - Le temps est découpé en quantas
 - Les quantas sont attribués aux processus en cours d'exécution

Ordonnancement

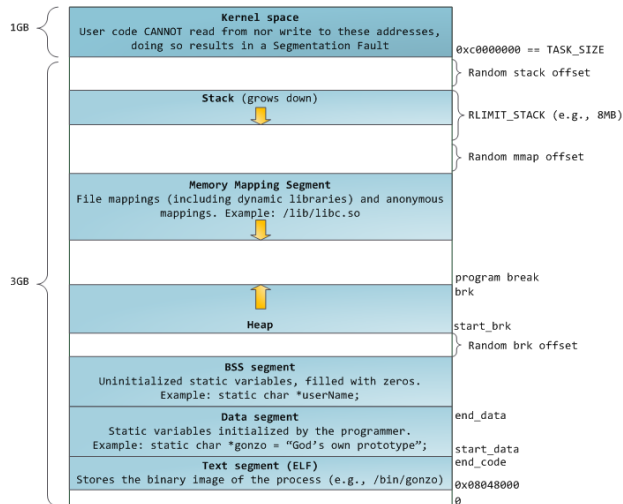
- Le passage d'un processus à un autre est appelé commutation de processus
 - Dans le cas du multitâches coopératif, la commutation est initiée par les programmes eux-mêmes
 - Dans le multitâches préemptif (cas de Linux), des évènements externes force la commutation
- La commutation est déclenchée par l'interruption d'un timer, générée par l'horloge, généralement toutes les 10ms
 - Cette interruption appelle la fonction `schedule()`
- Certains processus sont plus prioritaires que d'autres lors de l'ordonnancement
 - En fonction de la valeur de priorité dans la `task_struct`

Commutation de processus

- Un processus est associé à un contexte matériel particulier, nécessaire au bon déroulement de son exécution
- Lors d'une commutation de processus, le noyau doit sauvegarder le contexte du processus sortant
- Puis il doit restaurer le contexte du processus entrant

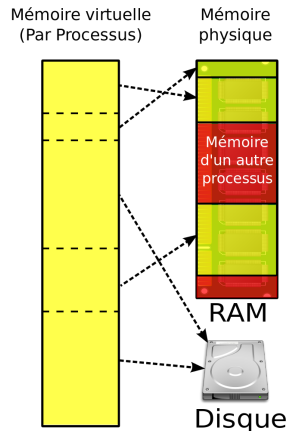
Mémoire virtuelle

Espace d'adressage d'un processus



Mémoire virtuelle : Le pourquoi du comment ?

- Traduction à la volée d'adresses virtuelles (logicielles) en adresses physiques (matérielles = RAM)
- La mémoire virtuelle permet :
 - D'utiliser de la mémoire de masse comme extension de la mémoire vive ;
 - D'augmenter le taux de multiprogrammation ;
 - De mettre en place des mécanismes de protection de la mémoire ;
 - De partager la mémoire entre processus.



Mémoire virtuelle

Mémoire virtuelle paginée

Principe de fonctionnement (1/3)

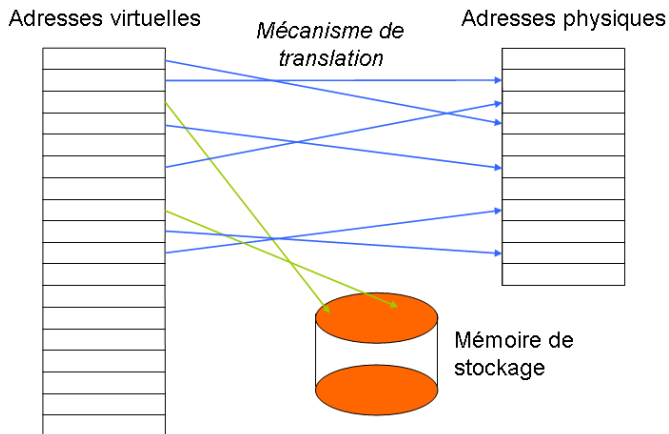
- Adresses manipulées par les programmes sont virtuelles, indiquent la position d'un mot dans la mémoire virtuelle.
- Mémoire virtuelle est formée de zones de même taille, appelées pages.
 - Adresse virtuelle = couple (numéro de page, déplacement dans la page).
 - Taille des pages = puissance de deux → déterminer sans calcul le déplacement
 - eg. 10 bits de poids faible de l'adresse virtuelle pour des pages de 1 024 mots, 22 bits pour le numéro de page.
- Mémoire physique également composée de zones de même taille, appelées cadres (*frames*),
 - un cadre contient une page : taille d'un cadre = taille d'une page.
 - Taille de l'ensemble des cadres dans la RAM utilisés par un processus est appelé *Resident set size*.

Principe de fonctionnement (2/3)

- Mécanisme de traduction (*translation*, ou génération d'adresse) convertit les adresses virtuelles en adresses physiques,
 - Consulte une table des pages (*page table*) pour connaître le numéro du cadre qui contient la page recherchée.
 - L'adresse physique obtenue est le couple (numéro de cadre, déplacement).
- Il peut y avoir plus de pages que de cadres :
 - les pages qui ne sont pas en mémoire sont stockées sur un autre support (disque), elles seront ramenées dans un cadre quand on en aura besoin.

Principe de fonctionnement (3/3)

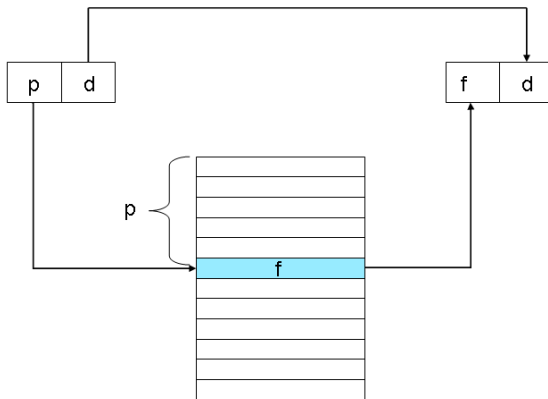
- On traduit des adresses virtuelles en adresses physiques, et certaines informations peuvent être temporairement placées sur un support de stockage.



Traduction (1/2)

- Table des pages indexée par le numéro de page.
- Chaque ligne est appelée **entrée dans la table des pages** (PTE pour *pages table entry*)
 - Contient le numéro de cadre.
- Table des pages peut être située n'importe où en mémoire,
 - Un registre spécial (PTBR pour Page Table Base Register) conserve son adresse.
- En pratique, le mécanisme de traduction est géré par la MMU (*memory management unit*)
 - Contient une partie de la table des pages, stockée dans un cache
 - Évite d'avoir à consulter la table des pages (en RAM) pour chaque accès mémoire.

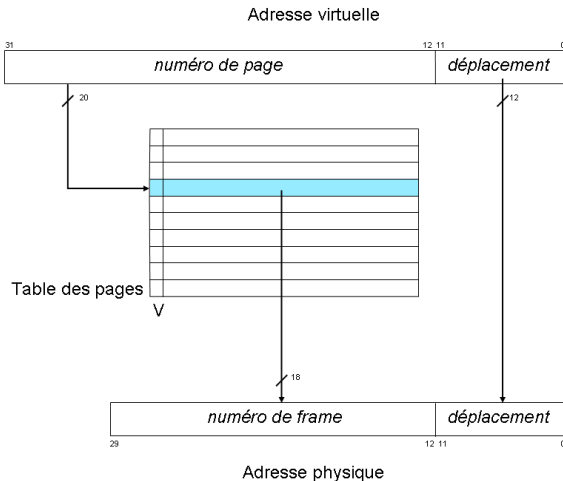
Traduction (2/2)



- Traduction de l'adresse virtuelle (page, déplacement) en adresse physique (frame, déplacement)

Exemple

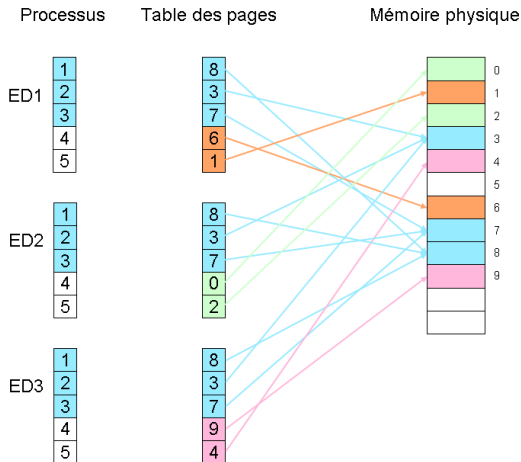
- Processeur génère des adresses virtuelles sur 32 bits
 - Accès à 4 Gio de mémoire.
- La taille de la page est de 4 Kio.
- Déplacement occupe les 12 bits de poids faible,
- Le champ numéro de page occupe les 20 bits de poids fort.



Exemple

- Champ spécial appartenant à chaque PTE.
- Pour simplifier, sur un bit : le bit de validité.
- Si 0 \rightarrow numéro de cadre est invalide.
- Il faut donc se doter d'une technique permettant de mettre à jour cette PTE pour la rendre valide.
- Trois cas peuvent se produire :
 - L'entrée est valide : elle se substitue au numéro de page pour former l'adresse physique.
 - L'entrée dans la table des pages est invalide.
 - Il faut trouver un cadre libre en mémoire vive
 - Et mettre son numéro dans cette entrée de la table des pages.
 - L'entrée dans la table des pages est valide mais correspond à une adresse sur la mémoire de masse où se trouve le contenu du cadre.
 - Un mécanisme devra ramener ces données pour les placer en mémoire vive.

Partage de mémoire



- Trois processus en cours d'exécution
- Trois instances sont situées aux mêmes adresses virtuelles (1, 2, 3, 4, 5).
- Deux zones mémoire distinctes : code et données.
- Il suffit de garder les mêmes entrées dans la table des pages pour que les trois instances se partagent la zone de code.
- Les entrées correspondantes aux pages de données sont, elles, distinctes.

Mémoire virtuelle
Accès à la mémoire

Allocation mémoire

- Le noyau s'occupe de gérer la mémoire
 - Lorsqu'un processus a besoin de plus de mémoire, il fait une demande au noyau
 - La demande s'effectue via un appel de la libc :
 - `malloc(size)` allouer size
 - `calloc(n,size)` n éléments de taille size initialisés à 0
 - `realloc(ptr, size)` modifier la taille d'une zone allouée avec malloc ou calloc
 - `free(ptr)` libérer une zone
 - `brk(addr)` faire croître le tas jusqu'à addr
 - `sbrk(incr)` augmenter le tas de incr

Écrire dans une adresse physique

- Accès à un périphérique mappé en mémoire
- Appel à la primitive `mmap()`

```
void *mmap(void *addr, size_t length, int prot, int flags,  
           int fd, off_t offset);
```

- Exemple

```
uint32_t * p;  
int fd = open("/dev/mem", O_RDWR);  
p = (uint32_t*)mmap(NULL, 4, PROT_WRITE|PROT_READ, MAP_SHARED,  
                    fd, 0xFF203000);  
  
*p = (1<<8);
```

Les modules

Introduction

- Le noyau Linux n'est plus un bloc de code monolithique
- Il est possible d'ajouter ou d'enlever du code dynamiquement
 - Évite d'avoir un noyau énorme
 - Pas nécessaire de recompiler le noyau à chaque fois que l'on désire ajouter une fonctionnalité
 - Pas nécessaire de relancer la machine a chaque modification
- Les modules servent à implémenter des **pilotes** (drivers)
- Ils servent aussi à implémenter des services dont le code doit être exécuté avec des **droits privilégiés**

Sources du noyau

- Pour compiler un module, il faut les sources (généralement dans `/usr/src/linux-<version>`)

Version du noyau :

```
uname -r
```

Installer les sources :

```
sudo apt update
```

```
sudo apt install linux-headers-$(uname -r)
```

```
sudo apt install build-essential checkinstall
```

Sources du noyau

- Pour compiler un module, il faut les sources (généralement dans `/usr/src/linux-<version>`)

```
.
|- 3rdparty          |- ipc
|- Documentation     |- kernel
|- arch              |- lib
|- block             |- mm
|- crypto            |- net
|- drivers           |- scripts
|- fs                |- security
|- include           |- sound
|- init              |- usr
```


Compilation

■ Utilisation d'un Makefile

```
obj-m:=le_module.o
```

```
KERNEL_SOURCE=/lib/modules/$(shell uname -r)/build
```

```
all:
```

```
    make -C $(KERNEL_SOURCE) M=$(PWD) modules
```

```
clean:
```

```
    make -C $(KERNEL_SOURCE) M=$(PWD) clean
```

```
install:
```

```
    make -C $(KERNEL_SOURCE) M=$(PWD) install
```

Exemple de module

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

```
MODULE_AUTHOR("Quelqu'un");
MODULE_DESCRIPTION("Exemple de module");
MODULE_SUPPORTED_DEVICE("Tous");
MODULE_LICENSE("GPL");
```

```
static int __init le_module_init(void) {
    printk(KERN_INFO "Hello world!\n");
    return 0;
}
```

```
static void __exit le_module_exit(void) {
    printk(KERN_ALERT "Bye bye...\n");
}
```

```
module_init(le_module_init);
module_exit(le_module_exit);
```

Gestion des modules

- Commandes en espace utilisateur :
 - insmod : insérer le module sans vérification des dépendances
 - modprobe : charger le module avec vérification des dépendances
 - rmmod : décharger le module
 - lsmod : lister les modules chargés
 - modinfo : afficher des infos sur le module
 - dmesg : afficher les messages émis par les modules

Paramètres d'un module

- Différentes macros pour déclarer les arguments à passer au module :

```
module_param(var, type, droits)
```

```
module_param_array(var, type, addr, droits)
```

```
module_param_string(nom_dans_modinfo, var, taille, droits)
```

- Utilisation dans le code du module :

```
int param = 3;
```

```
module_param(param, int, 0);
```

- Chargement :

```
modprobe le_module param=2
```

Exemple de paramètre

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

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

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

```
MODULE_AUTHOR("Quelqu'un d'autre");
```

```
MODULE_DESCRIPTION("Exemple de module");
```

```
MODULE_SUPPORTED_DEVICE("Presque tous");
```

```
MODULE_LICENSE("GPL");
```

```
static int param;
```

```
module_param(param, int, 0);
```

```
MODULE_PARM_DESC(param, "Un paramètre de ce module");
```

```
static int __init le_module_init(void) {
```

```
    printk(KERN_INFO "Hello world!\n");
```

```
    printk(KERN_DEBUG "le paramètre est=%d\n", param);
```

```
    return 0;
```

```
}
```

```
static void __exit le_module_exit(void) {  
    printk(KERN_ALERT "Bye bye...\n");  
}
```

```
module_init(le_module_init);
```

```
module_exit(le_module_exit);
```

Character device driver

Character device driver

- Permet l'accès à un périphérique depuis le mode user.
- S'oppose au block ou network driver. Différence interface noyau.
- Device driver identifié dans le noyau par un numéro appelé "majeur"
- Enregistrement du driver au moment de l'initialisation du module
- Ne pas oublier de décharger le module

Character device driver

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

```
int register_chrdev(unsigned char major, const char *name, struct file_operations *fops);
```

```
int unregister_chrdev(unsigned int major, const char *name);
```

Renvoient 0 ou >0 si tout se passe bien.

`register_chrdev`

- **major** : numéro majeur du driver, 0 indique que l'on souhaite une affectation dynamique.
- **name** : nom du périphérique qui apparaîtra dans `/proc/devices`.
- **fops** : pointeur vers une structure qui contient des pointeurs de fonction. Ils définissent les fonctions appelées lors des appels système (`open`, `read...`) du côté utilisateur.

`unregister_chrdev`

- **major** : numéro majeur du driver, le même qu'utilisé dans `register_chrdev`.
- **name** : nom du périphérique utilisé dans `register_chrdev`.

Opérations sur le fichier

- Il n'est pas obligatoire de définir toutes les opérations
 - `open()` : pour initialiser les ressources liées au périphériques.
 - `release()` : pour libérer ces mêmes ressources.
 - `read()/write()` : permet d'échanger des données avec le périphérique.

```
struct file_operations fops = {  
    .owner = THIS_MODULE,  
    .read = my_read_function,  
    .write = my_write_function,  
    .open = my_open_function,  
    .release = my_release_function /* correspond a close */  
};
```

- Certaines méthodes non implémentées sont remplacées par des méthodes par défaut.
- Les autres méthodes non implémentées retournent `-EINVAL`.

Implémentation des appels systèmes

```
static ssize_t my_read_function(struct file *file, char *buf, size_t count, loff_t *ppos) {
    printk(KERN_DEBUG "read()\n");
    return 0;          // (mais c'est une connerie)
}

static ssize_t my_write_function(struct file *file, const char *buf, size_t count,
                                loff_t *ppos) {
    printk(KERN_DEBUG "write()\n");
    return 0;          // (c'est stupide ici aussi)
}

static int my_open_function(struct inode *inode, struct file *file) {
    printk(KERN_DEBUG "open()\n");
    return 0;
}

static int my_release_function(struct inode *inode, struct file *file) {
    printk(KERN_DEBUG "close()\n");
    return 0;
}
```

Entrée périphérique

- Entrée périphérique : fichier pour interagir avec un module
- Création :
`mknod /dev/nom_du_peripherique c majeure mineure`
- Périphérique de type char (c) ou block (b)
- majeure : numéro définissant une classe de périphériques
 - disques durs, usb, pci...
- mineure : instance particulière pour ce type de composant
 - les constructeurs peuvent être différents
 - le pilote est le même

Utilisation du driver

- Créer son fichier spécial : `mknod /dev/le_driver c 254 0`
- Compiler et charger le module
- Depuis le terminal : `echo "bonjour" > /dev/_le_driver`
- Test avec `dmesg`
- Depuis un programme

Depuis un programme

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

int main(void) {
    int file = open("/dev/le_driver", O_RDWR);

    if(file < 0) {
        perror("open");
        exit(errno);
    }

    write(file, "hello", 6);
    close(file);

    return 0;
}
```

Exemple de fonction read()

```
#define BUF_LEN 64
static char *buffer;

static ssize_t my_read_function(struct file *file, char *buf, size_t count, loff_t *ppos) {
    int lus = 0;

    printk(KERN_DEBUG "read: demande lecture de %d octets\n", count);
    /* Check for overflow */
    if (count <= BUF_LEN - (int)*ppos)
        lus = count;
    else lus = BUF_LEN - (int)*ppos;
    if(lus)
        copy_to_user(buf, (int *)buffer + (int)*ppos, lus);
    *ppos += lus;
    printk(KERN_DEBUG "read: %d octets reellement lus\n", lus);
    printk(KERN_DEBUG "read: position=%d\n", (size_t)*ppos);
    return lus;
}
```

Exemple de write()

```
static ssize_t my_write_function(struct file *file, char *buf, size_t count, loff_t *ppos) {
    int ecrits = 0, i = 0;

    printk(KERN_DEBUG "write: demande ecriture de %d octets\n", count);
    if (count <= BUF_LEN - (int)*ppos) /* Check for overflow */
        ecrits = count;
    else ecrits = BUF_LEN - (int)*ppos;

    if(ecrits)
        copy_from_user((int *)buffer + (int)*ppos, buf, ecrits);
    *ppos += ecrits;
    printk(KERN_DEBUG "write: %d octets reellement ecrits\n", ecrits);
    printk(KERN_DEBUG "write: position=%d\n", (int)*ppos);
    printk(KERN_DEBUG "write: contenu du buffer\n");
    for(i=0;i<BUF_LEN;i++)
        printk(KERN_DEBUG " %d", buffer[i]);
    printk(KERN_DEBUG "\n");
    return ecrits;
}
```

ioctl

- ioctl utilisé pour configurer le périphérique
- Commandes codées sur un entier
- Peuvent avoir un argument
- Entier ou pointeur

- Côté module :

```
#include <linux/fs.h>.  
long ioctl(struct file *file, unsigned int cmd,  
           unsigned long arg);
```

- Côté utilisateur :

```
int ioctl(int fd, int cmd, char *argp);
```


Exemple

```
long my_ioctl(struct file *f, unsigned int cmd, unsigned long arg) {
    int retval = 0;

    switch(cmd) {
        case ... : ... break;
        case ... : ... break;
        default : retval = -EINVAL; break;
    }
    return retval;
}
```

- Ne pas oublier de le rajouter au file_operations

```
struct file_operations fops = {
    ...
    .unlocked_ioctl = my_ioctl_function;
};
```

Allocation mémoire

- Similaire au mode utilisateur
- `kmalloc()` et `kfree()`
- Un argument supplémentaire pour `kmalloc()`
 - `GFP_KERNEL` : allocation normale de la mémoire du noyau ;
 - `GFP_USER` : allocation mémoire pour le compte utilisateur (faible priorité) ;
 - `GFP_ATOMIC` : alloue la mémoire à partir du gestionnaire d'interruptions.

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

```
buffer = kmalloc(BUF_LEN, GFP_KERNEL);  
if(buffer == NULL) {  
    printk(KERN_WARNING "kmalloc error\n");  
    return -ENOMEM;  
}  
kfree(buffer);  
buffer = NULL;
```

Espaces mémoire

- Allocation mémoire du buffer au chargement du module.

```
buffer = kmalloc(BUF_LEN, GFP_KERNEL);
```

- `copy_from_user` et `copy_to_user` pour transférer un buffer depuis/vers l'espace utilisateur.

```
copy_from_user(unsigned long dest, unsigned long src, unsigned long len);
```

```
copy_to_user(unsigned long dest, unsigned long src, unsigned long len);
```

procfs

procfs

- procfs (process file system) est un pseudo-système de fichiers monté sur /proc utilisé pour accéder aux informations du noyau en cours d'exécution
- Ne correspond à aucun fichier sur block devices
- Fichiers accessibles en lecture (information, debugging), et en écriture (modification des paramètres du noyau)
- Infos sur les processus (/proc/[0-9]+)/
 - cmdline : arguments passés au programme
 - cwd : repertoire de travail du process
 - environ : variables d'environnement
 - fd/ : repertoire contenant les fichiers ouverts
 - exe : lien symbolique vers l'exécutable
 - maps : memory map du process
 - mem : mémoire virtuelle du process

Sans oublier

- Composants matériels, réseau...etc
 - `/proc/cpuinfo` : informations sur le CPU
 - `/proc/meminfo` : information sur la mémoire physique
 - `/proc/vmstats` : information sur la mémoire virtuelle
 - `/proc/mounts` : information sur les mounts
 - `/proc/filesystems` : information sur les systèmes de fichiers actifs
 - `/proc/uptime` : uptime du système
 - `/proc/cmdline` : command line du noyau
- Debugging

Création d'un fichier

```
static inline struct proc_dir_entry *proc_create(  
    const char *name, umode_t mode,  
    struct proc_dir_entry *parent,  
    const struct file_operations *proc_fops);
```

- name : nom du fichier à créer
- mode : droit d'accès
- parent : entrée parente dans le procfs (NULL si à la racine)
- file_operations : une structure contenant les pointeurs des fonctions utilisables

Création d'une arborescence

Support des chemins complexes (/proc/un/chemin/complexe)

```
struct proc_dir_entry *proc_mkdir(const char *name,  
    struct proc_dir_entry *parent);
```

- name : nom du répertoire
- parent : entrée parente dans l'arborescence

Suppression

```
void remove_proc_entry(const char* name,  
                        struct proc_dir_entry* parent);
```

- name : nom du fichier
- parent : entrée parente dans l'arborescence

Accès aux fichiers

```
struct file_operations proc_fops;  
proc_fops->read = votre_fonction_read;  
proc_fops->write = votre_fonction_write;
```

- Prototype de la fonction read()

```
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
```

- Prototype de la fonction write()

```
ssize_t (*write) (struct file *, const char __user *,  
                 size_t, loff_t *);
```

Exemple de fonction read()

```
ssize_t fops_read(struct file *file, char __user * buffer,
                  size_t count, loff_t * ppos) {
    int errno=0;
    int copy;

    if (count > TAILLE) count=TAILLE;
    if ((copy=copy_to_user(buffer, message, strlen(message)+1)))
        errno = -EFAULT;
    printk(KERN_INFO "message read, %d, %p\n", copy, buffer);
    return count-copy;
}
```

Fonction write()

```
int write_func(struct file* file, const char __user * buffer,  
              size_t count, loff_t *offset);
```

- file : décrit le fichier à écrire
- buffer : zone mémoire où écrire les données
- count : max octets à écrire dans buffer
- offset : position dans le fichier

Exemple de fonction write()

```
ssize_t fops_write(struct file * file, const char __user * buffer,
                  size_t count, loff_t * ppos) {
    int len = count;

    if (len > TAILLE) len = TAILLE;

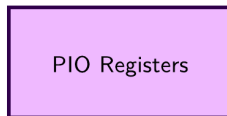
    printk(KERN_INFO "Recieving new messag\n");
    if (copy_from_user(message, buffer, count)) {
        return -EFAULT;
    }
    message[count] = '\0';
    size_of_message = strlen(message);
    printk(KERN_INFO "New message : %s\n", message);
    return count;
}
```

Les entrées/sorties

Memory-mapped I/O vs Port I/O



Physical Memory
address space, accessed with
normal load/store instructions



Separate I/O address space,
accessed with specific instructions

Port I/O

- Chaque périphérique est connecté à un bus d'I/O et dispose de ces propres adresses
- On appelle ces adresses les "ports d'E/S" (I/O ports)
- Le noyau permet de lire et écrire sur le bus d'I/O avec des routines spécifiques
- On peut réserver des ports afin qu'un module ait l'exclusivité de leur utilisation

Utilisation des Port I/O

- Récupérer un octet depuis le port défini par l'adresse `read_addr`
`int inb(int read_addr)`
- Envoie l'octet `value` sur le port défini par l'adresse `write_addr`
`int inb(int write_addr, char value)`
- Une routine par type (b pour byte, w pour mot de 16 bits, l pour mot de 32 bits)

Exemple

```

/* default is the first printer port on PC's */
static unsigned long base = 0x378;
/* Version-specific methods for the fops structure. */
ssize_t short_write(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos) {
    int retval = count;
    unsigned char *kbuf = kmalloc(count, GFP_KERNEL), *ptr;

    if (!kbuf) return -ENOMEM;
    if (copy_from_user(kbuf, buf, count)) return -EFAULT;
    ptr = kbuf;
    while (count--) {
        outb(*(ptr++), base);
        wmb();          // write memory barrier
    }

    return retval;
}

struct file_operations short_fops = {
    .owner = THIS_MODULE,
    .write = short_write
};

```

Memory-Mapped I/O

- Les périphériques modernes contiennent un petit espace de mémoire adressable.
 - Des registres de contrôle (similaires aux ports d'E/S)
 - Du stockage de données (textures video, paquets réseau, ...)
 - etc...
- Le noyau permet de mapper cet espace dans l'espace mémoire
- On appelle ce mapping la mémoire d'E/S (I/O memory)

Utilisation de la MMIO

■ Mapping :

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

■ Démapping :

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

■ Test de disponibilité :

```
int check_mem_region(unsigned long start, unsigned long len);
```

■ Remapping :

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

- Le pointeur obtenu pointe vers la mémoire du périphérique
- On ne doit pas utiliser ce pointeur comme n'importe quel pointeur, à cause de problèmes d'optimisation matériel (reordering des instructions)

Communication avec la mémoire d'I/O

■ Lire :

```
unsigned int ioread8(void *addr);  
unsigned int ioread16(void *addr);  
unsigned int ioread32(void *addr);
```

■ Écrire :

```
void iowrite8(u8 value, void *addr);  
void iowrite16(u16 value, void *addr);  
void iowrite32(u32 value, void *addr);
```

■ En boucle :

```
void ioread8_rep(void *addr, void *buf, unsigned long count);  
...
```

Les Interruptions

Interruptions

- Permet d'associer un numéro d'interruption (`irq`) à une fonction handler qui traite cette interruption pour le compte du driver du périphérique `devname` repéré par son identifiant `dev_id`
- La fonction de gestion de l'interruption retourne :
 - `IRQ_HANDLED` si l'interruption a été traitée correctement
 - `IRQ_NONE`

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

```
int request_irq(unsigned int irq,  
               void (*handler) (int, void , struct pt_regs *),  
               unsigned long flags /* SA_INTERRUPT ou SA_SHIRQ */,  
               const char *device, void *dev_id);
```

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

Les timers

- Création d'un timer logiciel programmé

```
void timer_setup(struct timer_list *timer,  
                void (*callback)(struct timer_list *), unsigned int flags)
```

- Active le timer et donne le moment en nombre de ticks depuis le démarrage du système où doit être déclenché le timer

```
int mod_timer(struct timer_list * timer, unsigned long expires)
```

- Si l'on veut déclencher le timer dans 100 ticks, expires vaut jiffies+100
- Il faut réarmer le timer dans la fonction de gestion si l'on veut un timer périodique
- On peut convertir les jiffies en millisecondes avec msec_to_jiffies()

Exemple

```
#include <linux/timer.h>
#define INTERVALLE 100

static struct timer_list timer;

static void montimer(struct timer_list *t) {
    ...
    /* Il faut réarmer le timer si l'on veut un appel périodique */
    mod_timer(&timer, jiffies + INTERVALLE);
    ...
}

static int __init module_init(void) {
    ...
    timer_setup(&timer, montimer, 0);
    mod_timer(&timer, jiffies + INTERVALLE);
}
```

Gestion des interruptions longues

- Parfois le traitement à effectuer lors d'une interruption peut être très long
- Le noyau doit rester le moins longtemps possible dans une routine d'interruption (blocage du système)
- Traitement en deux temps :
 - Partie top-half : acquittement rapide de l'interruption et ajout du traitement à effectuer dans une liste de tasklet (faible latence) ou une file de traitements (workqueue)
 - Partie bottom-half : gestion différée (asynchrone) du traitement de l'interruption par la tasklet ou la workqueue

Tasklets

- Les tasklets sont des appels effectués uniquement dans un contexte d'interruption
- Créer une tasklet revient à faire une demande au noyau pour exécuter une tâche atomique de manière différée
 - En fonction de sa disponibilité
 - Jamais au delà d'un tick

Tasklets

- Déclarer la *tasklet name*, de l'associer à la fonction *func* en lui passant les données *data*

```
DECLARE_TASKLET(name, func, data)
```

- Demande l'exécution de la tasklet

```
tasklet_schedule(&name)
```

- Demande d'exécution haute priorité à n'utiliser que dans le cadre de pilote faible latence tels que les buffer audio.

```
tasklet_hi_schedule(&name)
```

- Inhiber une tasklet

```
DECLARE_TASKLET_DISABLED(), tasklet_disable(), tasklet_disable_nosync()
```

- Permet de la réactiver

```
tasklet_enable()
```

- Permet de la supprimer

```
tasklet_kill()
```

Exemple

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

void tasklet_function(unsigned long);

char tasklet_data[64];

DECLARE_TASKLET(test_tasklet, tasklet_function, (unsigned long) &tasklet_data);

void tasklet_function(unsigned long data) {
    struct timeval now;
    do_gettimeofday(&now);
    printk("%s at %ld,%ld\n", (char *) data, now.tv_sec, now.tv_usec);
}

int init_module(void) {
    sprintf(tasklet_data,"%s\n", "Linux tasklet called in init_module");
    tasklet_schedule(&test_tasklet);
}
```

Les workqueues

- Fonctionnent dans le contexte d'un processus noyau
 - Plus de souplesse quant à la possibilité d'être interrompues/reprises
- Peuvent s'exécuter de manière différée
- Création :
`create_workqueue(name)`
 - Créé la file et renvoie un pointeur sur une structure `struct workqueue_struct`
- Destruction :
`void destroy_workqueue(struct workqueue_struct *queue)`
 - Permet de détruire une file

Utilisation des workqueues

- Déclare la workqueue en associant une fonction à exécuter à un nom de workqueue
`DECLARE_WORK(nom, fonction, donnees)`
- Met la tâche dans la file pour une exécution immédiate
`int queue_work(struct workqueue_struct *wq, struct work_struct *work)`
- Met la tâche dans la file pour une exécution différée
`int queue_delayed_work(struct workqueue_struct *wq,
 struct work_struct *work, unsigned long delay)`
- Supprimer une tâche
`int cancel_delayed_work(struct work_struct *work)`
- Vider une file de travail
`void flush_workqueue(struct workqueue_struct *queue)`