Les modules sous LINUX

1) Présentation

Dans les cas d'applications classiques de Linux, **le noyau utilise le plus souvent des modules** qui peuvent être dynamiquement chargés et déchargés en fonction des besoins du système. **Ces modules peuvent être des pilotes de périphériques matériels**, comme des cartes d'extension, ou bien liés à un support générique. Au démarrage, un noyau minimal est chargé puis les modules nécessaires sont chargés/déchargés au fur et à mesure des besoins du noyau.
Un pilote de périphérique peut être implémenté directement dans le noyau de LINUX sans être obligé de passer par la notion de module.
Un module se charge dans l'espace noyau de LINUX

Les commandes de base pour manipuler les modules sont les suivantes :

Afficher la liste des modules charges
lsmod

```
root@beagleboard:/home/hello# lsmod
Module                  Size  Used by
gpiokernel1mknod1       3185  0
bufferclass_ti          4768  0
omaplfb                 8733  0
pvrsrvkm              154248  2 bufferclass_ti,omaplfb
sdmak                   3743  0
lpm_omap3530            6605  0
dsplinkk              125793  1 lpm_omap3530
cmemk                  21482  0
8192cu                522320  0
ipv6                  249063  22
minix                  25759  0
rtc_twl                 4451  0
mt9t112                 9246  0
rtc_core               12535  1 rtc_twl
root@beagleboard:/home/hello#
```

Charger un module
insmod nom module.ko

Génère le fichier des dépendances entre les modules
depmod

Charge le module demande et ceux qui sont nécessaires
modprobe

Décharger un module
rmmod nom module

Obtenir des informations sur un module
modinfo nom module

Architecture d'un module hellok.c

Cette architecture est la suivante :

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


MODULE_LICENSE("GPL v2");

static int hello_init(void)
{
        printk(KERN_ALERT "Hello World!\n");
        return 0;
}

static void hello_exit(void)
{
        printk(KERN_ALERT "Goodbye Cruel World!\n");
}

module_init(hello_init); // lancez au chargement du module
module_exit(hello_exit);// lancez au déchargement du module
```

2) Création d'un module sous LINUX

2.1) Création du Makefile et du fichier source

On va utiliser l'utilitaire make pour créer notre environnement de compilation, il nous faut donc déclarer un fichier Makefile définissant notre environnement de compilation.

```
root@beagleboard:/home# mkdir hellok
root@beagleboard:/home# cd hellok
root@beagleboard:/home/hellok# gedit Makefile
```

```
obj-m += hellok.o   # nom du fichier objet à compiler

PWD := $(shell pwd)

all:
        make -C /lib/modules/2.6.32/build M=$(PWD) modules # librairie pour la création

clean:
        make -C /lib/modules/2.6.32/build M=$(PWD) clean
```

Chemin des librairies du noyau pour la compilation

attention au backslah pour la directive clean et all ou le format du fichier n'est pas reconnu
Le chemin /lib/modules/3.1.0/build correspond aux librairies du noyau qui ont été installées lors de la phase de compilation du noyau sur la beagleboard.

Créez le fichier source hellok.c avec un éditeur de texte

```
root@beagleboard:/home/hellok# gedit hellok.c
root@beagleboard:/home/hellok# make
make -C /lib/modules/2.6.32/build M=/home/hellok modules # librairie pour la cr
ation
make[1]: Entering directory `/usr/src/2.6.32'
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory `/usr/src/2.6.32'
root@beagleboard:/home/hellok#
```

```
root@beagleboard:/home/hellok# ls
Makefile        hellok.c   hellok.mod.c  hellok.o
Module.symvers  hellok.ko  hellok.mod.o  modules.order
root@beagleboard:/home/hellok#
```

Pour insérer notre module

insmod hellok.ko pour charger le module

lsmod pour visualiser les modules qui sont chargés

rmmod hellok.ko pour décharger le module

dmesg pour voir les messages du noyau

```
root@beagleboard:/home/hellok# insmod hellok.ko
root@beagleboard:/home/hellok# lsmod
Module                  Size  Used by
hellok                   688  0
gpiokernel1mknod1       3185  0
bufferclass_ti          4768  0
```

Insertion du module

```
root@beagleboard:/home/hellok# dmesg

[ 2506.534423] Hello World!
root@beagleboard:/home/hellok#
```

```
root@beagleboard:/home/hellok# modinfo hellok.ko
filename:       hellok.ko
license:        GPL v2
srcversion:     41FB7FCC5BB669030648F23
depends:
vermagic:       2.6.32 preempt mod_unload modversions ARMv7
root@beagleboard:/home/hellok# rmmod hellok
root@beagleboard:/home/hellok# lsmod
Module                  Size  Used by
gpiokernel1mknod1       3185  0
bufferclass_ti          4768  0
omaplfb                 8733  0
pvrsrvkm              154248  2 bufferclass_ti,omaplfb
sdmak                   3743  0
lpm_omap3530            6605  0
dsplinkk              125793  1 lpm_omap3530
cmemk                  21482  0
8192cu                522320  0
ipv6                  249063  22
minix                  25759  0
rtc_twl                 4451  0
mt9t112                 9246  0
rtc_core               12535  1 rtc_twl
root@beagleboard:/home/hellok#
```

Déchargement du module

En retapant la commande dmesg on obtient :

```
[ 2870.246154] Goodbye Cruel World!
root@beagleboard:/home/hellok#
```

Le module doit être compatible avec le noyau qui tourne sur la beagleboard XM d'où la nécessité d'avoir compilé le noyau et d'utiliser ces librairies par la suite.

Modprobe

Avant d'aborder la partie développement logiciel sur les systèmes d'exploitation temps réel attardons nous quelques minutes sur un exemple concret de système embarqué. Le radar de veille tridimensionnelle LRR

Notion de temps réel

1) Définitions :

Travailler en temps réel sur un système informatique correspond à élaborer, à partir d'informations acquises d'un phénomène extérieur, des informations de commande ou de contrôle dans un temps cohérent avec l'évolution du phénomène contrôlé(***ABRIAL-BOURGNE)***.

Une approche temps réel s'attache avant tout au **déterminisme** des actions, c'est à dire à leur **instant de début et de fin**.
Un système de commande a donc un grand nombre de ***tâches*** à exécuter telles que : *calculer* une action, *écrire* le résultat d'un calcul sur un écran de visualisation, *effectuer* une mesure, *détecter* l'éventuelle présence d'une panne latente, *signaler* des comportements anormaux, ...
Nombre de ces tâches sont indépendantes et peuvent donc être exécutées simultanément. C'est pourquoi on dit qu'un système de commande est naturellement ***parallèle***.

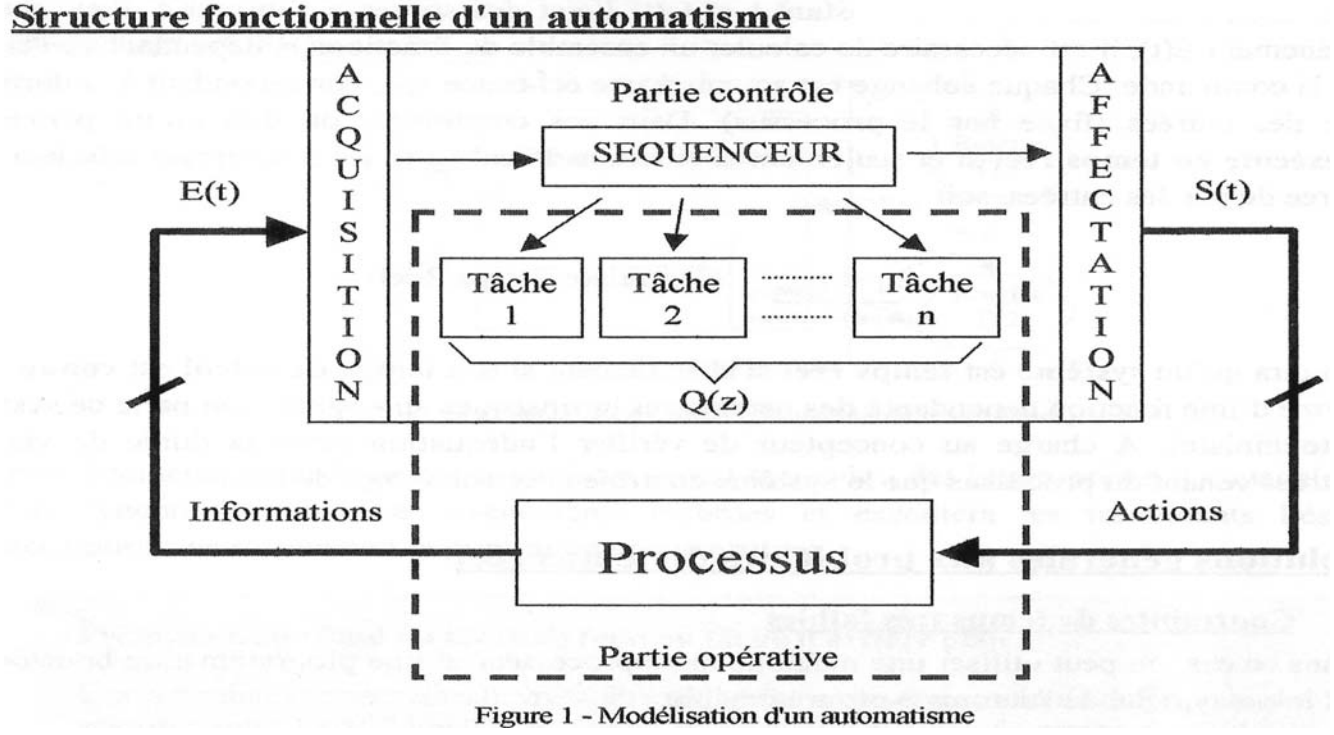


Figure 1 - Modélisation d'un automatisme

Un système temps réel n'est pas forcément un système qui va vite mais un système **qui satisfait à des contraintes temporelles**.
Ces contraintes peuvent être très sévères lorsqu'elles sont inférieures à la milliseconde (pour les systèmes radar) ou très larges lorsqu'elles sont de plusieurs minutes voire de plusieurs heures (pour un processus de type chimique).
En réalité, lorsque les processus à contrôler sont lents, les contraintes de temps sont liées non pas au processus principal mais aux dispositifs auxiliaires tels que les systèmes de mesures, les sécurités, ...

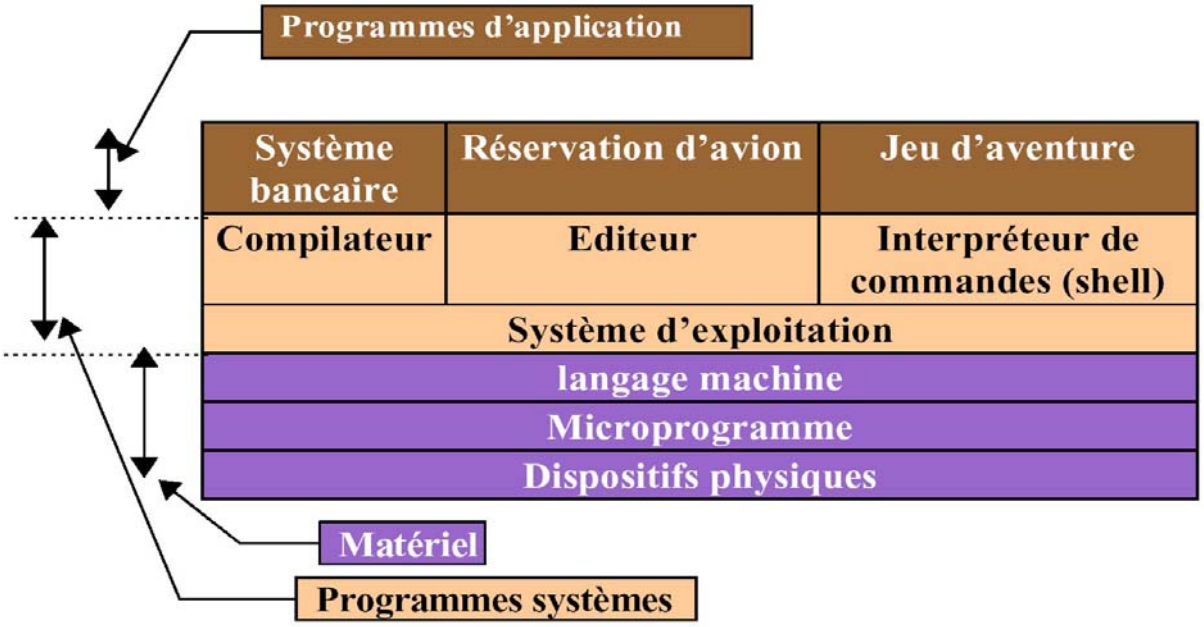
2) Tâches matérielles, tâches logicielles

Un système temps réel comporte en général des tâches soumises à de fortes contraintes de temps, et des tâches pouvant être différées, tout en étant exécutées dans l'intervalle de temps voulu.
Les tâches soumises à de fortes contraintes de temps sont liées à des ***interruptions*** et sont appelées ***tâches matérielles***. Les ***tâches différées*** ou ***tâches logicielles*** sont programmées, soit dans une boucle de scrutation, soit de façon *indépendante* (principe utilisé dans ce cours). Elles sont alors exécutées d'une manière ***pseudo-simultanée (partage des ressources du processeur)***.
Lorsque la programmation est effectuée par une boucle de scrutation (l'occupation du processeur est totale) , on parle d'un système temps réel **monotâche** ; dans le cas contraire, de système temps réel multitâche.

Dans tous les cas, un système temps réel doit gérer l'occupation du processeur en fonction des échéances de temps liées au processus, des ressources disponibles, des synchronisations et des échanges de données.
Dès que l'application devient *complexe*, les systèmes temps réels sont programmés selon le **mode *multitâche*** en raison d'une plus grande facilité de conception.
On fera donc attention à la distinction entre **système multitâche** (de type temps partagé) et **application temps réel** (multitâche, aussi, mais soumis à des contraintes temporelles).
Dans le premier cas, l'application gérera simultanément, de façon concurrente, plusieurs tâches dont l'ensemble traduit la globalité du processus modélisé.
Dans le second cas, l'application réagira en fonction d'événements externes asynchrones dans le respect strict des contraintes de temps imposées par le processus physique modélisé.

3) Le système d'exploitation

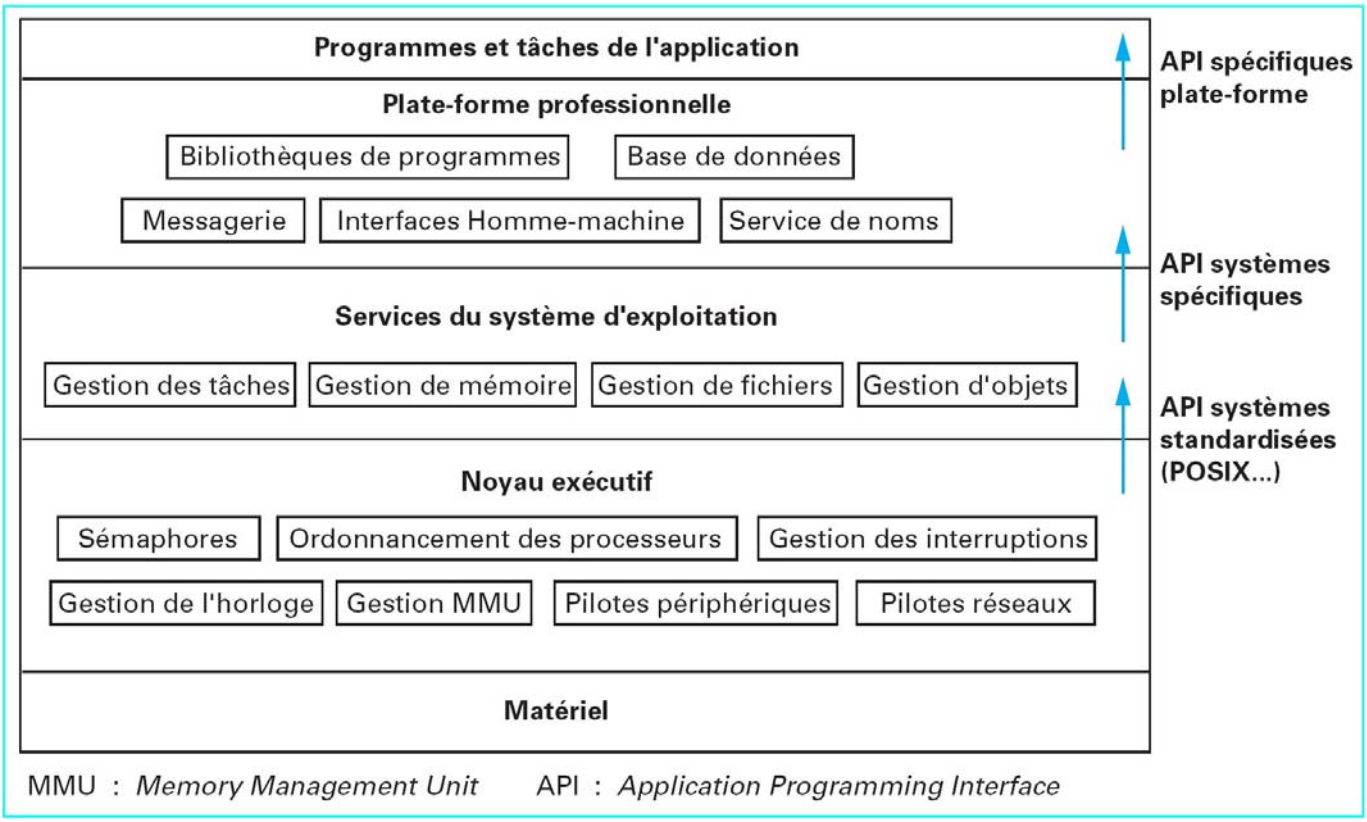
La complexité évidente du matériel implique la réalisation d'une **machine virtuelle** qui gère le matériel : **c'est le système d'exploitation.**



Le rôle principal du système d'exploitation est d'isoler les programmes des détails du matériel. Ils présentent à l'utilisateur une **« machine virtuelle »** ***plus facile à comprendre et à utiliser*** ; qui donne une image du matériel pas toujours fidèle à la réalité

Le BIOS (*Basic Input Output System*, système d'entrées/sorties de base) constitue la couche basse de tous les systèmes d'exploitations sur PC. Il est donc responsable de la gestion du matériel : clavier, écran, disques durs, liaisons séries et parallèles.
Du point de vue de l'utilisation, on peut considérer le BIOS comme **une librairie de fonctions**. Chaque fonction effectue une tâche bien précise, comme par exemple afficher un caractère donné sur l'écran. L'appel de l'une de ces fonctions constitue un ***appel système***.

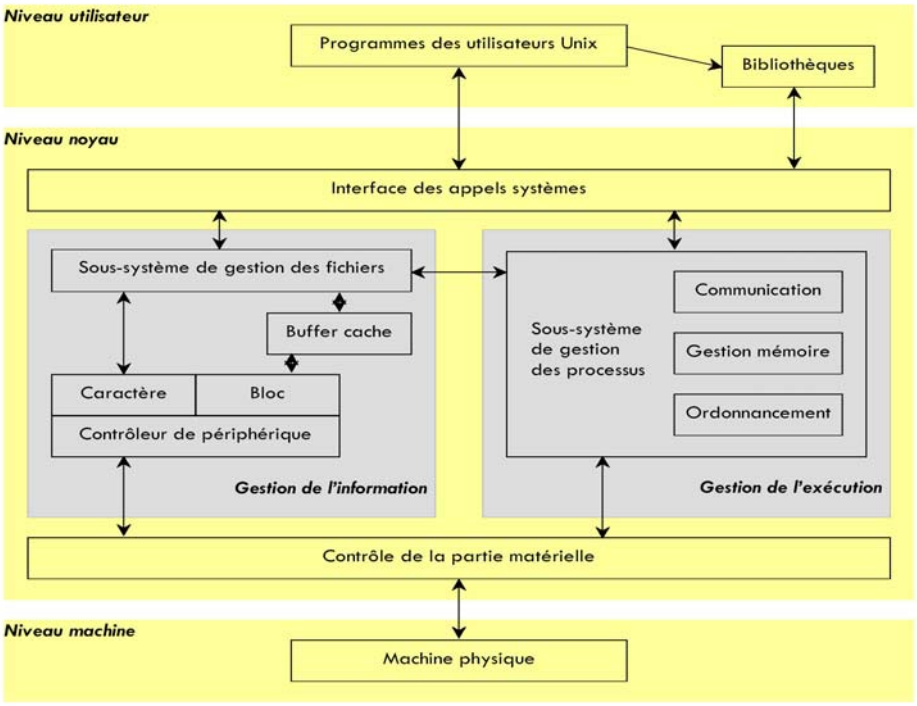
3.1) Les différentes couches d'un système d'exploitation :



3.2) La structure d'un noyau (type UNIX ou LINUX)

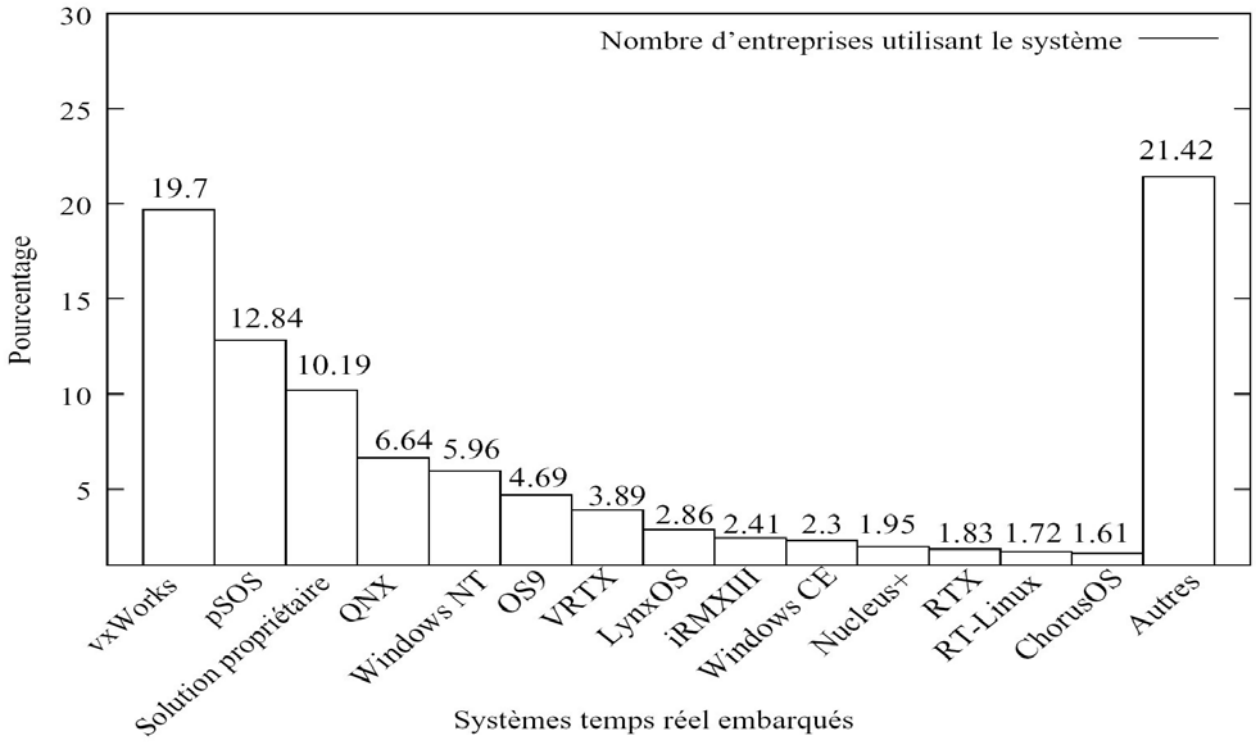
Il est important de différencier **programme** (fichier exécutable) et **processus** (programme en cours d'exécution). Un processus a besoin non seulement d'un programme, mais également des ressources de la machine physique :
- Le processeur sur lequel s'exécute le programme,
- La mémoire dans laquelle il est stocké, ainsi que les variables nécessaires,
- Les périphériques : écran, clavier, souris, imprimante, disques, etc.

Unix peut gérer plusieurs processus simultanément, mais les ressources restent limitées au nombre de processeurs, à la quantité de mémoire disponible, et à certains périphériques communs non partageables (écrans ou imprimante).D'où la nécessité d'optimiser les ressources. Le processeur est alloué par tranche de temps à chaque processus (multiprogrammation).



Les processus correspondent à **l'exécution de tâches** (programmes des utilisateurs, entrées-sorties, ...) par le système.
La commutation des tâches (**multiprogrammation**), c'est-à-dire le passage d'une tâche à une autre, est réalisée par un ordonnanceur (scheduler) au niveau le plus bas du système. Cet ordonnanceur est activé par des interruptions d'horloge, de disque et de terminaux.

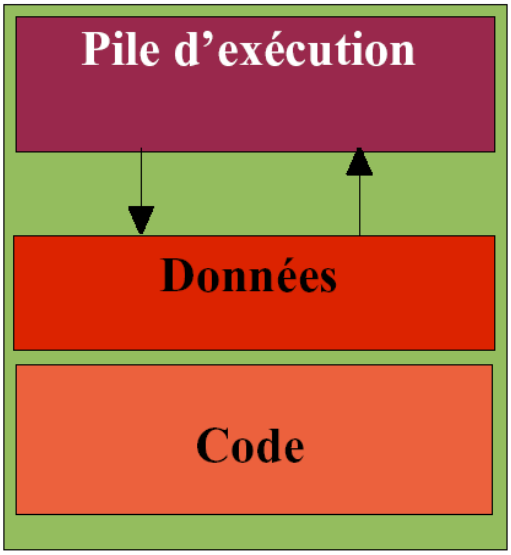
Etat du marché des RTOS



4) Notion de processus:

4.1/ Définitions :

Un programme est composé d'une ou plusieurs séquences d'instructions, agissant sur un ensemble de données. Un programme est essentiellement statique.
Un processus représente l'exécution d'un programme statique. C'est une **entité dynamique** créée à un instant donné et qui disparaît au bout d'un temps fini.
De plus une tâche peut communiquer avec d'autres tâches.
Les processus sont composés d'un espace de travail en mémoire formé de 3 segments :



- **Le code correspond aux instructions, en langage d'assemblage, du programme à exécuter.**
- **La zone de données contient les variables globales ou statiques du programme ainsi que les allocations dynamiques de mémoire.**
- **Enfin, les appels de fonctions, avec leurs paramètres et leurs variables locales, viennent s'empiler sur la pile.**

Le noyau maintient une table pour gérer l'ensemble des processus. Cette table contient la liste de tous les processus avec des informations concernant chaque processus.
L'environnement d'un processus encore appelé *contexte*, est caractérisée par :
- l'état des pointeurs associés à chaque zone mémoire (programme, données, pile).
- des caractéristiques de cette tâche.

La commande ps -aux nous permet de découvrir tous les processus en cours sur LINUX

```
root@beagleboard:~# ps -aux
Warning: bad ps syntax, perhaps a bogus '-'? See http://procps.sf.net/faq.html
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  1.7  0.1   1560   564 ?        Ss   11:21   0:01 init [5]
root         2  0.0  0.0      0     0 ?        S    11:21   0:00 [kthreadd]
root         3  0.0  0.0      0     0 ?        S    11:21   0:00 [ksoftirqd/0]
root         4  0.0  0.0      0     0 ?        S    11:21   0:00 [watchdog/0]
root         5  0.0  0.0      0     0 ?        S    11:21   0:00 [events/0]
root         6  0.0  0.0      0     0 ?        S    11:21   0:00 [khelper]
root         7  0.0  0.0      0     0 ?        S    11:21   0:00 [async/mgr]
root         8  0.0  0.0      0     0 ?        S    11:21   0:00 [pm]
root         9  0.0  0.0      0     0 ?        S    11:21   0:00 [sync_supers]
root        10  0.0  0.0      0     0 ?        S    11:21   0:00 [bdi-default]
root        11  0.0  0.0      0     0 ?        S    11:21   0:00 [kblockd/0]
root        12  0.0  0.0      0     0 ?        S    11:21   0:00 [omap2_mcspi]
root        13  0.0  0.0      0     0 ?        S    11:21   0:00 [ksuspend_usbd]
root        14  0.0  0.0      0     0 ?        S    11:21   0:00 [khubd]
root        15  0.0  0.0      0     0 ?        S    11:21   0:00 [kseriod]
```

Ou ps –all

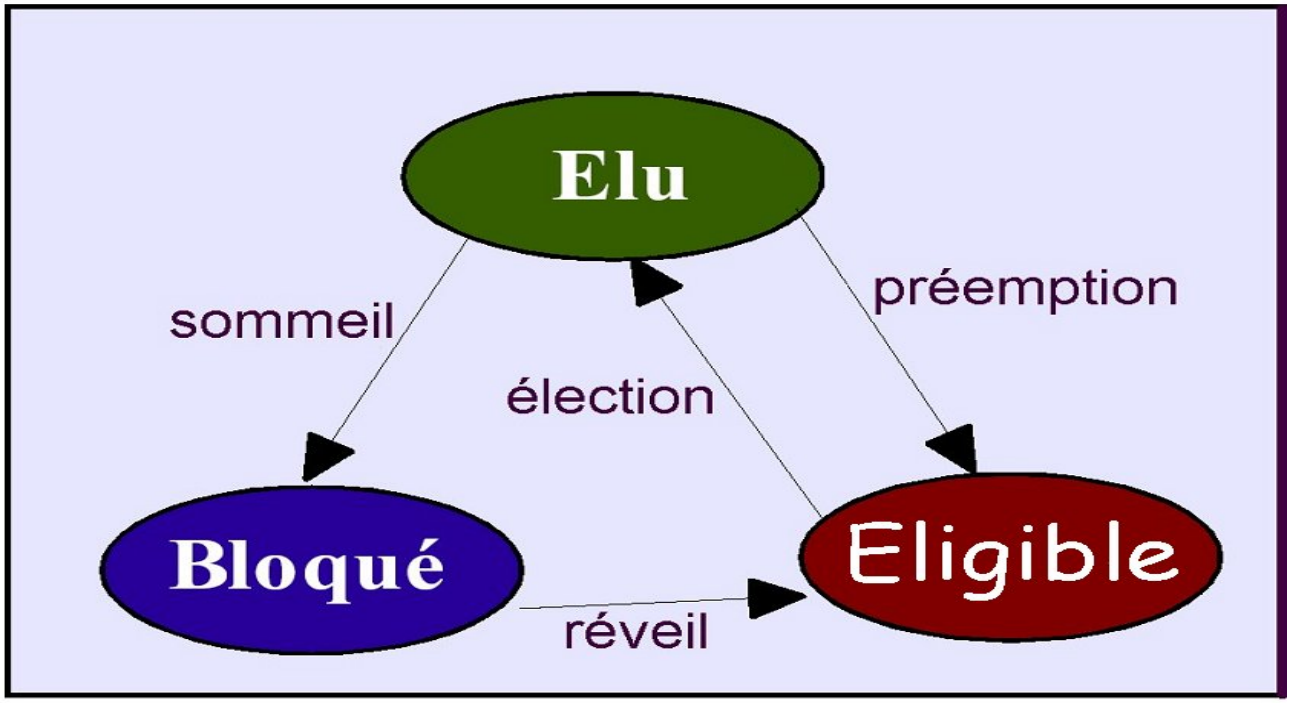
```
root@beagleboard:~# ps -all
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S     0  1037  1035  5  80   0 -  5000 poll_s pts/0    00:00:17 x11vnc
4 R     0  1166  1150  0  80   0 -   854 -      pts/1    00:00:00 ps
root@beagleboard:~#
```

Lorsque le processus ou la tâche cesse d'être active, le système d'exploitation effectue une **sauvegarde du contexte** en rangeant toutes ces données **dans la pile du système**. Lorsque la tâche redevient active, **on restaure le contexte** afin de lui redonner la main.

Les tâches sont classées par ordre de priorité dans une file. Ce classement est effectué par **l'ordonnanceur (scheduler)** et destinée au répartiteur **(dispatcher)** organe chargé d'allouer le processeur aux différentes tâches.

4.2) Etat d'une tâche ou d'un processus :

Une tâche ne peut être que dans l'un des trois états fondamentaux suivants :



- **élu** (en cours d'exécution) c'est à dire en possession d'un processeur.
- **éligible** (ou prête) : la tâche est alors candidate au processeur. Elle possède toutes les ressources nécessaires à son fonctionnement sauf un processeur. Son lancement ne dépend que de sa priorité par rapport aux tâches en cours d'exécution ou aux tâches prêtes.
- **bloqué** (ou suspendue) : la tâche n'est plus en possession d'un processeur. Elle s'est interrompue avant sa fin sur l'apparition d'une condition de blocage (attente d'une ressource indispensable à son exécution future, mise en sommeil volontaire de la tâche, etc...).

Les **transitions d'états** correspondantes sont les suivantes:

- la transition : **éligible** ⟶ **élu** correspond à une **allocation** du processeur.
- la transition : **élu** ⟶ **éligible** correspond à une réquisition du processeur au profit d'une autre tâche. On parle alors de **Préemption**.
- la transition : **élu** ⟶ **bloqué** est généralement due à un appel système impliquant l'attente d'une ressource du système : **Sommeil**.
- la transition: **bloqué** ⟶ **éligible** est appelée **Réveil** . Elle s'effectue lorsque arrive un événement attendu.

Les changements d'états d'une tâche consistent à réaliser une commutation de contexte. C'est à dire à remplacer le contexte d'une tâche par celui d'une autre nouvellement élue en vue de l'exécution de cette dernière par un processeur.

Un processeur est donc responsable de l'exécution d'une ou plusieurs tâches de manière alternative ou quasi simultanée.

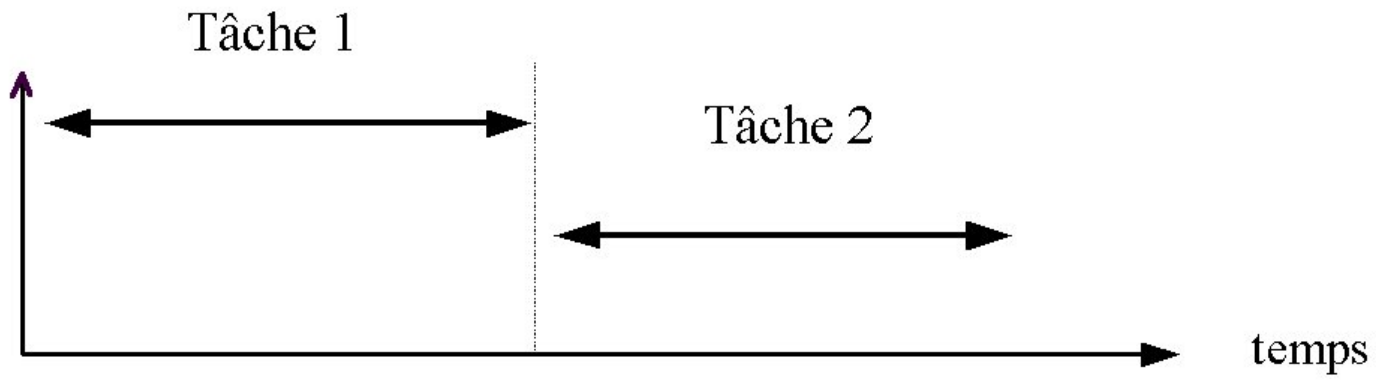
Les noyaux temps réel se divisent en deux catégories appelées *noyaux sans réquisition du processeur* (no preemptive scheduling) et ***noyaux avec réquisition du processeur*** (preemptive scheduling).
Dans un **noyau sans réquisition du processeur**, la tâche est exécutée jusqu'à ce qu'elle fasse **appel à un service** du noyau ou **qu'elle s'arrête** d'elle même. Selon la situation alors présente, le scheduler décide si la tâche doit se poursuivre ou non. On peut donc trouver une configuration avec une tâche prioritaire prête et une moins prioritaire en exécution. Ce noyau est plus simple à écrire.

Dans un ***noyau avec réquisition*** du processeur, une tâche peut à tout instant **perdre le contrôle du processeur** au profit d'une tâche de priorité supérieure. La tâche qui perd le processeur n'a aucune possibilité de le savoir. Ce sont les événements temps réel qui font que
l'ordonnanceur peut prendre cette décision à tout instant. Seuls les noyaux de ce type pourront être considérés comme véritablement temps réel.

préemption: interruption d'une tâche pour une autre plus urgente.

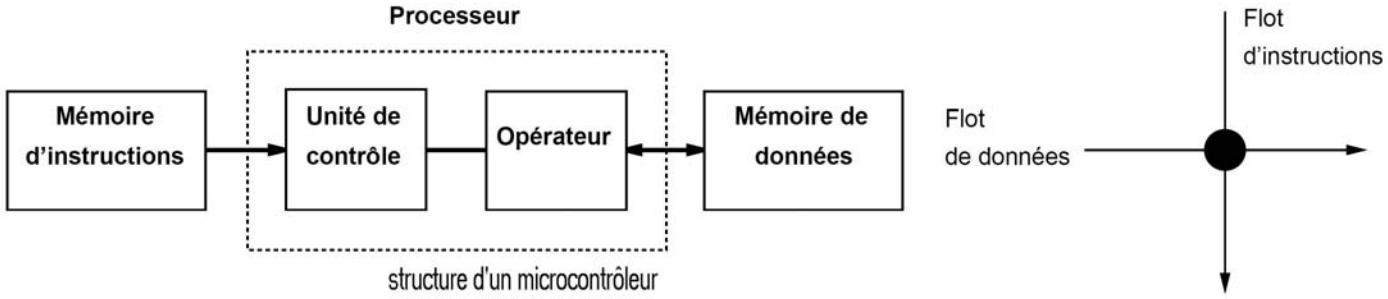
5) Quasi-parallélisme ou réelle simultanéité :

- Système **monotâche/monoprocesseur:** c'est un système ne possédant qu'un seul processeur, et qui permet de n'exécuter qu'une seule tâche à la fois.

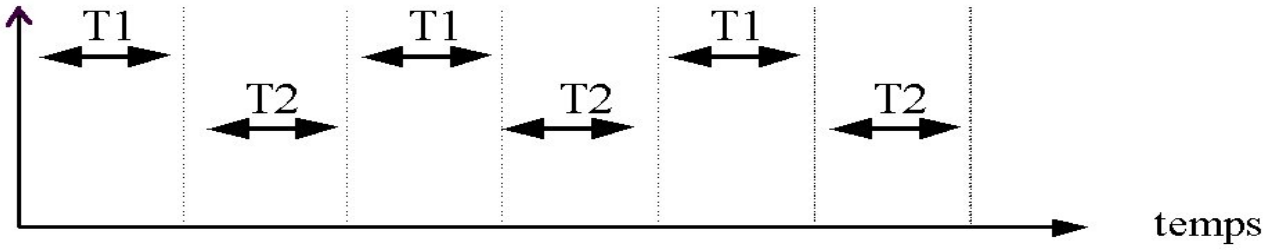


Système monotâche/monoprocesseur

SISD : *Single Instruction stream Single Data stream*

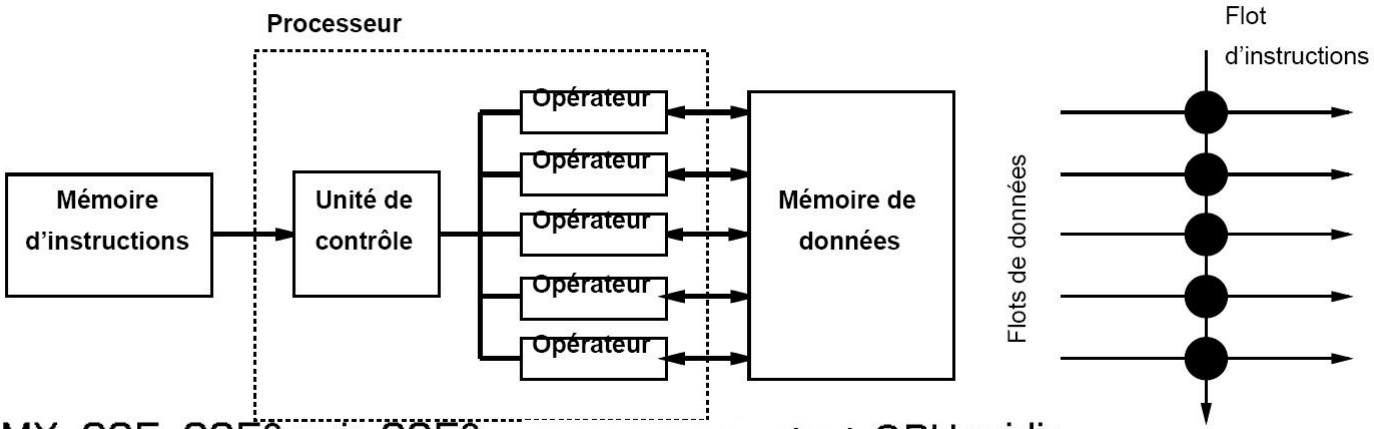


Système **multitâche/monoprocesseur:** C'est un système comportant un seul processeur, mais il est capable, à l'échelle humaine, d'exécuter plusieurs tâches simultanément. A l'échelle du processeur, les processus seront exécutés alternativement. La simultanéité est donc virtuelle, on parle de **quasi-parallélisme** d'exécution.



Système multitâche/monoprocesseur

SIMD SINGLE INSTRUCTION MULTIPLE FLUX DE DONNEES



MMX, SSE, SSE2 puis SSE3, processeur actuel GPU nvidia

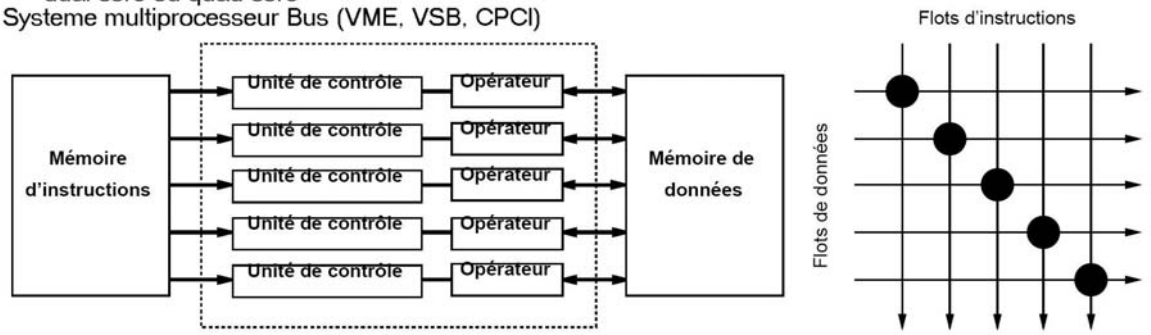
Système **multitâche/multiprocesseur:** C'est un système qui possède plusieurs processeurs. Si le nombre de ses processeurs est au moins égal au nombre de tâches à traiter, le système sera alors capable de fonctionner en réel parallélisme, ou **réelle simultanéité**.

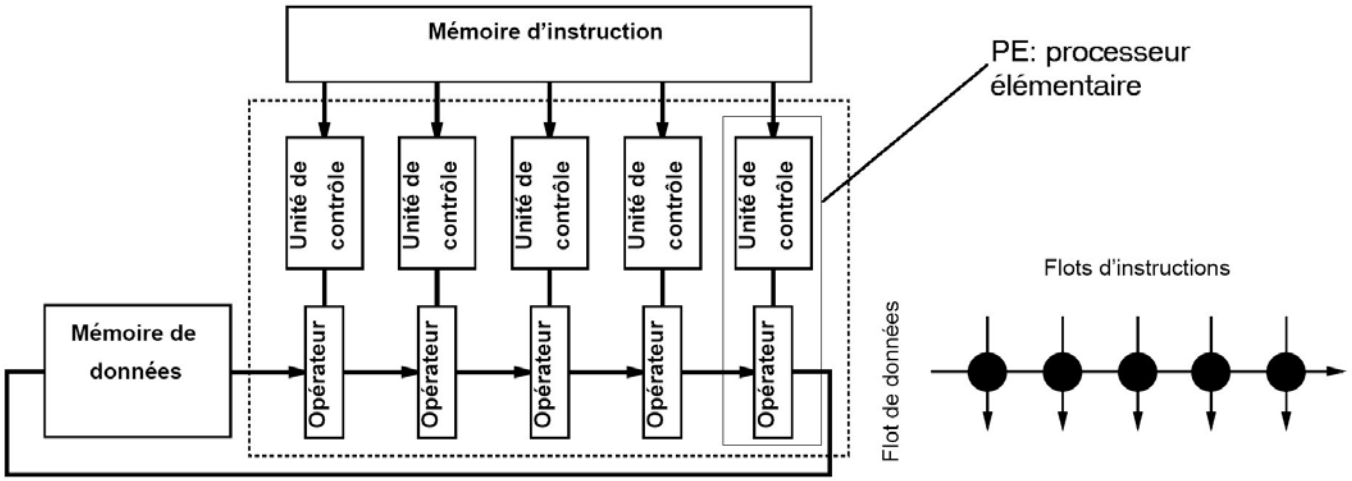


Système multitâche/multiprocesseur

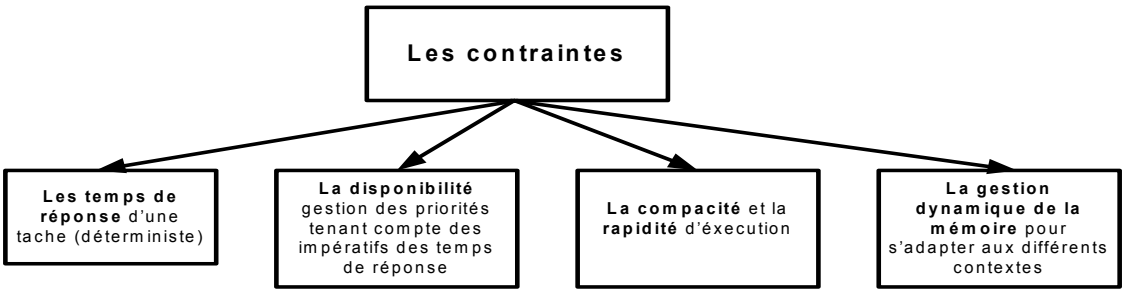
MIMD : *Multiple Instruction stream Multiple Data stream*

dual core ou quad core
Systeme multiprocesseur Bus (VME, VSB, CPCI)





6) Les contraintes d'un système temps réel



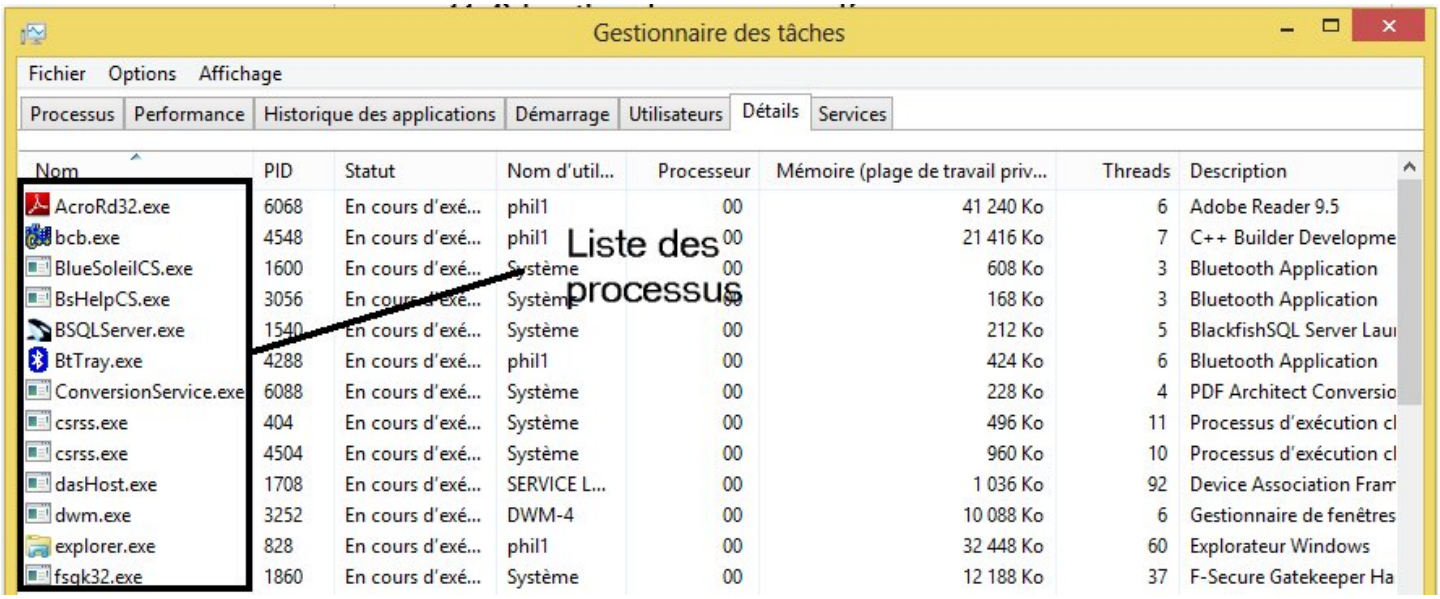
Les Processus

1) gestion des processus lourds et légers

Un processus lourd est un programme en cours d'exécution. Chaque processus lourd implique la gestion d'un espace d'adressage virtuel MMU et de nouvelles copies de toutes les variables et ressources nécessaires à l'exécution (pile, registres, fichiers ouverts, verrous etc…).
Le principal avantage de ces processus est la protection mémoire entre les processus du même système.
Par contre, l'utilisation des processus lourds présente les inconvénients suivants :

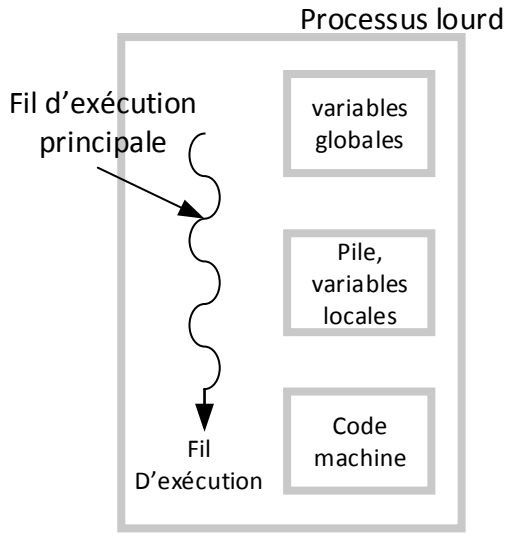
- leur création nécessite des appels systèmes coûteux en temps,
- le changement de contexte entre processus est une opération lente, en particulier pour de nombreux transferts en mémoire,
- le coût des mécanismes de protection associés au processus,
- l'interaction, la synchronisation ou la communication entre processus nécessite l'utilisation de mécanismes de communication spéciaux (tube communicant appelé "*pipe*", *socket*, boîte aux lettres),
- le partage de mémoire entre processus s'effectue par ajout de mécanismes lourds (bibliothèque de partage de mémoire).

La liste des processus lourd est donnée par le gestionnaire des taches :

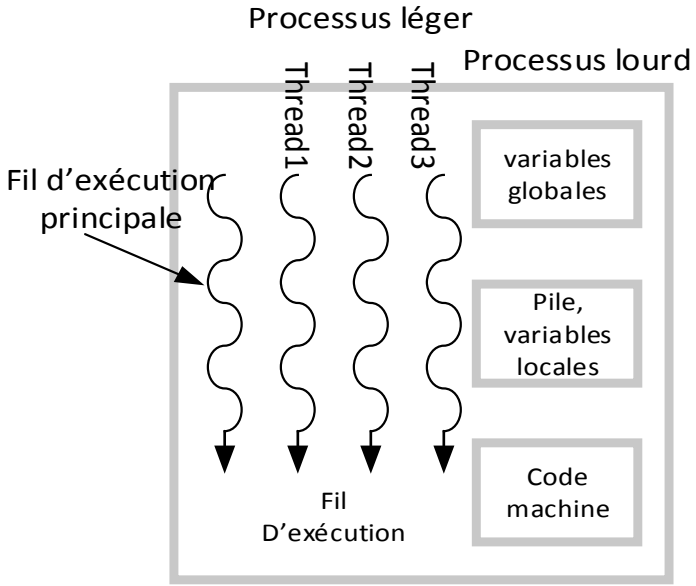


Nom	PID	Statut	Nom d'util...	Processeur	Mémoire (plage de travail priv...	Threads	Description
AcroRd32.exe	6068	En cours d'exé...	phil1	00	41 240 Ko	6	Adobe Reader 9.5
bcb.exe	4548	En cours d'exé...	phil1	00	21 416 Ko	7	C++ Builder Developme
BlueSoleilCS.exe	1600	En cours d'exé...	Système	00	608 Ko	3	Bluetooth Application
BsHelpCS.exe	3056	En cours d'exé...	Système	00	168 Ko	3	Bluetooth Application
BSQLServer.exe	1540	En cours d'exé...	Système	00	212 Ko	5	BlackfishSQL Server Lau
BtTray.exe	4288	En cours d'exé...	phil1	00	424 Ko	6	Bluetooth Application
ConversionService.exe	6088	En cours d'exé...	Système	00	228 Ko	4	PDF Architect Conversio
csrss.exe	404	En cours d'exé...	Système	00	496 Ko	11	Processus d'exécution cl
csrss.exe	4504	En cours d'exé...	Système	00	960 Ko	10	Processus d'exécution cl
dasHost.exe	1708	En cours d'exé...	SERVICE L...	00	1 036 Ko	92	Device Association Fram
dwm.exe	3252	En cours d'exé...	DWM-4	00	10 088 Ko	6	Gestionnaire de fenêtres
explorer.exe	828	En cours d'exé...	phil1	00	32 448 Ko	60	Explorateur Windows
fsgk32.exe	1860	En cours d'exé...	Système	00	12 188 Ko	37	F-Secure Gatekeeper Ha

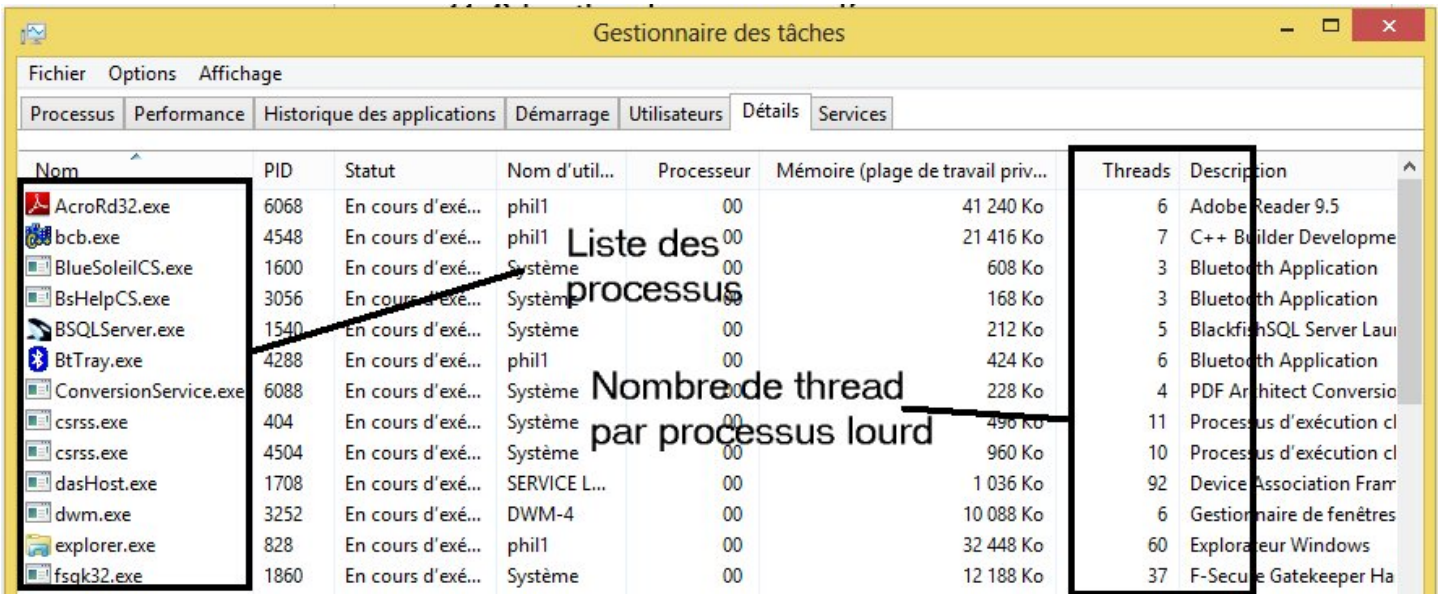
Un processus lourd classique, ne contenant qu'un seul fil d'exécution, est dit **monoprogrammé** :
L'exécution du code du processus est réalisée de manière séquentielle par un fil de contrôle (*thread of control*).
C'est un programme monolithique comprenant un corps principal la fonction main()



Les threads ou processus léger permettent de dérouler plusieurs blocs d'instructions, en PARALLELE, à l'intérieur du même processus. Un thread exécute une fonction. Chaque fonction associée à un thread va être exécutée de façon parallèle et indépendante.



Le système d'exploitation gère donc des centaines de thread :



Nom	PID	Statut	Nom d'util...	Processeur	Mémoire (plage de travail priv...	Threads	Description
AcroRd32.exe	6068	En cours d'exé...	phil1	00	41 240 Ko	6	Adobe Reader 9.5
bcb.exe	4548	En cours d'exé...	phil1	00	21 416 Ko	7	C++ Builder Developme
BlueSoleilCS.exe	1600	En cours d'exé...	Système	00	608 Ko	3	Bluetooth Application
BsHelpCS.exe	3056	En cours d'exé...	Système	00	168 Ko	3	Bluetooth Application
BSQLServer.exe	1540	En cours d'exé...	Système	00	212 Ko	5	BlackfishSQL Server Lau
BtTray.exe	4288	En cours d'exé...	phil1	00	424 Ko	6	Bluetooth Application
ConversionService.exe	6088	En cours d'exé...	Système	00	228 Ko	4	PDF Architect Conversio
csrss.exe	404	En cours d'exé...	Système	00	496 Ko	11	Processus d'exécution cl
csrss.exe	4504	En cours d'exé...	Système	00	960 Ko	10	Processus d'exécution cl
dasHost.exe	1708	En cours d'exé...	SERVICE L...	00	1 036 Ko	92	Device Association Fram
dwm.exe	3252	En cours d'exé...	DWM-4	00	10 088 Ko	6	Gestionnaire de fenêtres
explorer.exe	828	En cours d'exé...	phil1	00	32 448 Ko	60	Explorateur Windows
fsgk32.exe	1860	En cours d'exé...	Système	00	12 188 Ko	37	F-Secure Gatekeeper Ha

Dans un système monoprocesseur l'illusion sera donnée que chaque thread est exécuté de façon parallèle ou concurrente. En réalité chaque thread sera exécuté à tour de rôle mais très rapidement.

Avantages :

- **Multi-tâche moins coûteux** : puisqu'il n'y a pas de changement de mémoire virtuelle, la commutation de contexte (*context switch*) entre deux threads est moins coûteuse que la commutation de contexte de processus lourd
- **Communication entre threads plus rapide et plus efficace** : grâce au partage de certaines ressources entre threads, IPC (*Inter Processus Communication*) inutile pour les threads

- **Inconvénients** :

Programmation utilisant des threads est toutefois plus difficile : obligation de mettre en place des **mécanismes de synchronisation**.

Le système d'exploitation temps réel LINUX sait gérer des processus lourds comme des processus légers

Les fonctions système sont la base même de l'intérêt de l'utilisation d'un système d 'exploitation. Il s'agit d'offrir à l'utilisateur un moyen d'accès aux ressources du système Primitives de Gestion des processus

1) Primitives de Gestion des processus

1.1) Création d'un processus

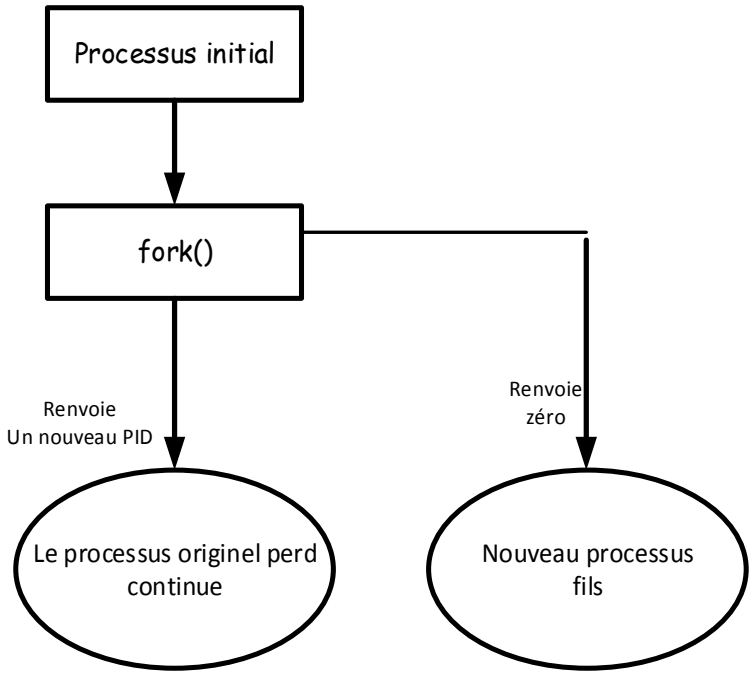
int forkc() *pour LINUX (librairie sys_clib.l)*

- Lorsqu'un programme appelle *fork*, une copie du processus, appelée *processus fils*, est créée. Le processus parent continue d'exécuter le programme à partir de l'endroit où *fork* a été appelé. Le processus fils exécute lui aussi le même programme à partir du même endroit.
  Cette primitive crée un nouveau processus (appelé fils) qui est une copie exacte du processus appelant (processus père)

- La différence est faite par la valeur de retour de fork(), qui est égale à zéro chez le processus fils, et elle est égale au pid du processus fils chez le processus père
- la primitive renvoie -1 en cas d'erreur

Le processus fils hérite du processus père :

- la priorité
- la valeur du masque
- le propriétaire
- les descripteurs des fichiers ouverts
- le pointeur de fichier (offset) pour chaque fichier ouvert
- Le comportement vis à vis des signaux



Il n'est évidemment guère utile d'avoir deux copies du même programme s'exécutant dans le système. Après duplication, le fils va **"changer de programme"** en utilisant la fonction **execv()**. Cette primitive conserve l'identité du processus mais remplace son code exécutable (et ses données) par celui d'un nouveau processus.

1.2) Association d'un programme à un processus (suite)

-Fonction *execv*:
Permet le lancement par le processus appelant d'un nouveau programme préalablement chargé.
int execv*(, char *name, char *proc[]) pour LINUX*
librairie :

name est un pointeur sur une chaine de caractère identifiant le nouveau processus. La chaine de caractère doit correspondre au nom du processus et au chemin correspondant à ce processus dans LINUX.
proc[] est un tableau de pointeur sur chaine de caractère correspondant à la liste des paramètres que reçoit le nouveau processus.

Cette fonction retourne le PID du nouveau processus ou –1 en cas d'échec.

Exemple :

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
int
main (int argc, char * argv[ ])
{
int pid;
char * nv_argv[ ] = { "/home/TD1_A/./exemple_execvp", NULL };
fprintf (stdout, "ok \n Tentative d'exécuter exemple_execv ... ");
pid=execv (nv_argv[0 ], nv_argv);
return (1);
}
```

2) Autres fonctions systèmes

Identification des processus

#include
int pid = getpid ()
Cette primitive fournit respectivement le numéro du processus appelant

#include

int ppid = getppid()

Cette primitive fournit respectivement le numéro du processus Père du processus appelant

Mise en sommeil d'un processus

#include

int sleep(int n)
Si n est à 0 le processus dort indéfiniment
Si n>0 temps en ms d'endormissement

Si un signal réveille un processus, la fonction du processus qui traite ce signal s'exécute, l'exécution du processus continue par la suite après l'instruction sleep

Terminaison d'un processus

Un processus se termine lorsqu'il n'a plus d'instructions ou lorsqu'il exécute la fonction :
#include
void exit(int statut)
L'argument status est un entier qui permet d'indiquer au shell (ou au père de façon générale) qu'une erreur s'est produite. On le laisse à zéro pour indiquer une fin normale.

Attendre la fin d'un processus fils

#include

int wait ();

L'appel wait() permet à un processus d'attendre la fin de l'un de ses fils. Si le processus n'a pas de fils ou si une erreur se produit, wait() retourne -1. Sinon, wait() bloque jusqu'à la fin de l'un des fils, et elle retourne son PID.

Librairie :

L'ordonnancement des tâches sous LINUX

Dans un système d'exploitation il y a un certain nombre de processus ou de taches qui doivent se partager des ressources processeurs communes. Le rôle de d'ordonnanceur est de choisir le ou les processus ou tache(s) qui doivent accaparer les ressources processeurs

1) L'ordonnanceur (scheduler)

C'est le module du noyau multitâche chargé de gérer un (ou plusieurs) processeur(s) pour un ensemble de tâches.

Cette gestion consiste à :

- attribuer le processeur libre ou libéré à une tâche prête à commencer ou à continuer son exécution.
- reprendre le processeur affecté à une tâche, soit à sa demande explicite (attente d'un signal par ex.), soit de façon inconditionnelle (préemption).

2) Commutation des tâches

L'ordonnancement des tâches est du ressort de l'ordonnanceur, le changement de contexte des tâches est, l'affaire du dispatcher (répartiteur).

1 - Sauvegarde du contexte

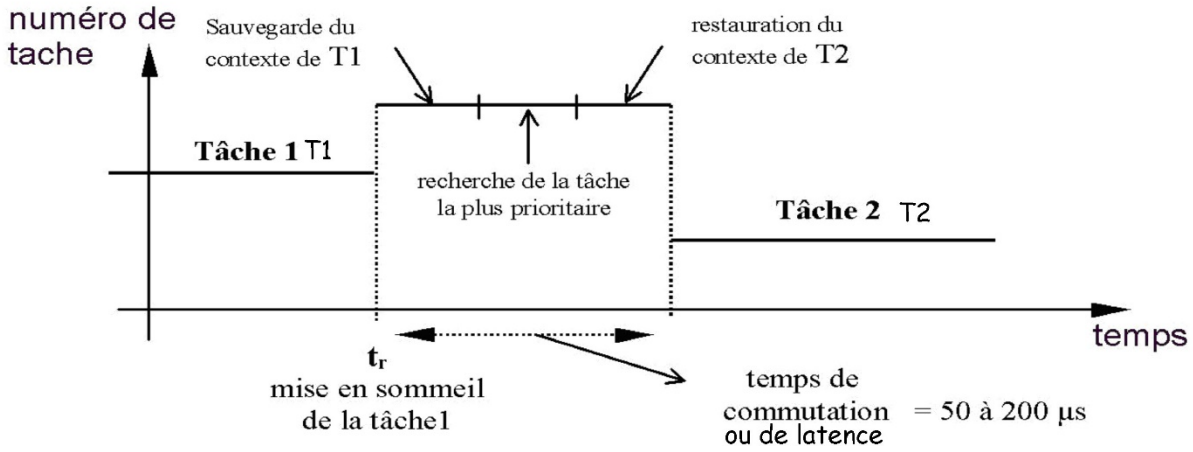
2 - Passage de l'état éligible à l'état élu

3 - Restauration du contexte et exécution de la tâche (restauration de l'état des registres)

Une commutation de tâches peut être provoquée :

- sur demande de la tâche en exécution (attente d'une condition ou fin de tâche)
- sur décision de l'ordonnanceur (tâche en cours moins prioritaire qu'une autre tâche)
- en réponse à un phénomène externe (IT)

Ex : Commutation des tâches lors d'une mise en sommeil



3) Les différentes techniques d'ordonnancement

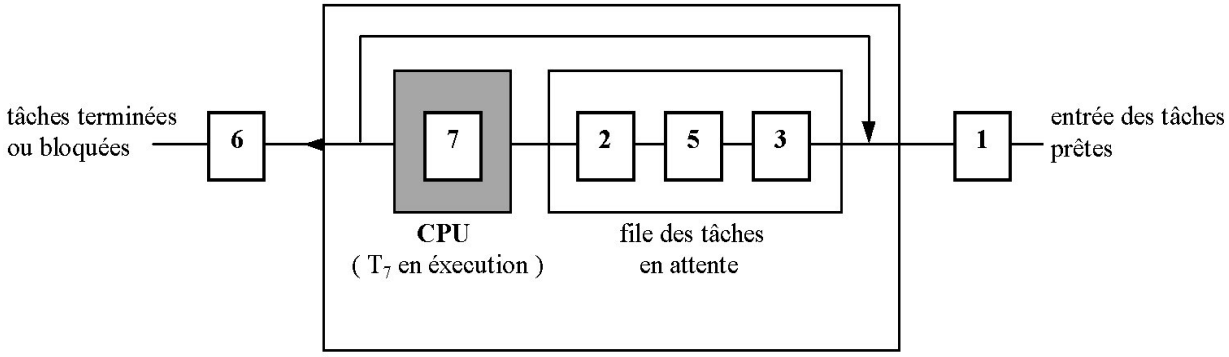
- Ordonnancement circulaire
- Ordonnancement par priorité
- Ordonnancement par priorité avec file multiple

3.1) Ordonnancement circulaire

Son algorithme est le plus ancien, le plus simple et le plus fiable. On l'appelle aussi le **tourniquet**.
Ce type d'ordonnanceur n'est pas utilisable dans les noyaux temps réel car **toutes les tâches ont ici la même priorité.**

Principe :

Chaque tâche **possède une quantité de temps en quantum** pendant lequel elle est autorisée à occuper le processeur. Lorsque son quantum est écoulé, le processeur est alloué à la tâche suivante dans la file.
Si la tâche se termine avant la fin du quantum ou si elle se bloque, le temps restant est alloué à une autre tâche.



Les 4 tâches (7 2 5 3) sont en exécution apparente,
mais seule la « 7 » est en réel possession du processeur

Inconvénients :

Si le système comporte de nombreuses tâches se bloquant très rarement, la puissance du processeur est alors altérée par le nombre de tâches présentes dans la file.
- Dans un noyau avec réquisition du processeur (préemptif), une tâche peut à tout instant perdre le processeur au bénéfice d'une tâche de priorité supérieure

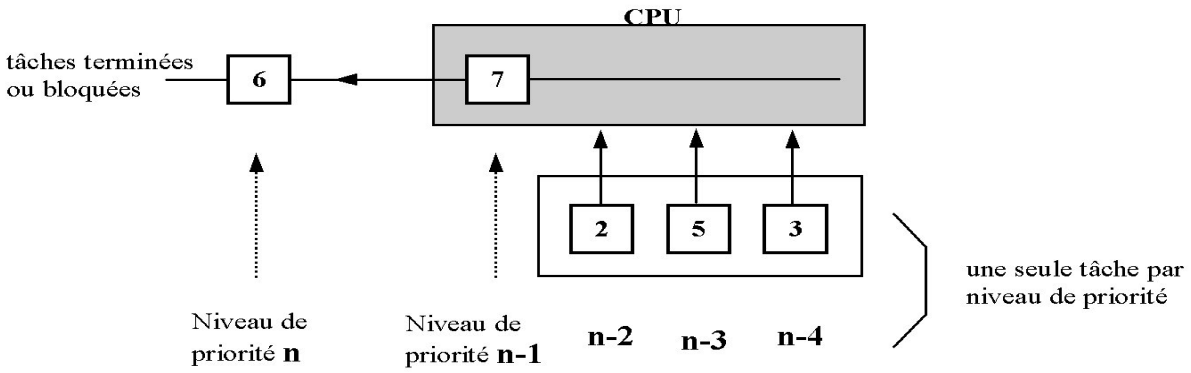
3.2) Ordonnancement par priorité

L'ordonnancement par file circulaire suppose que les tâches ont la même priorité. Ceci est rarement le cas avec un processus industriel.

Principe :

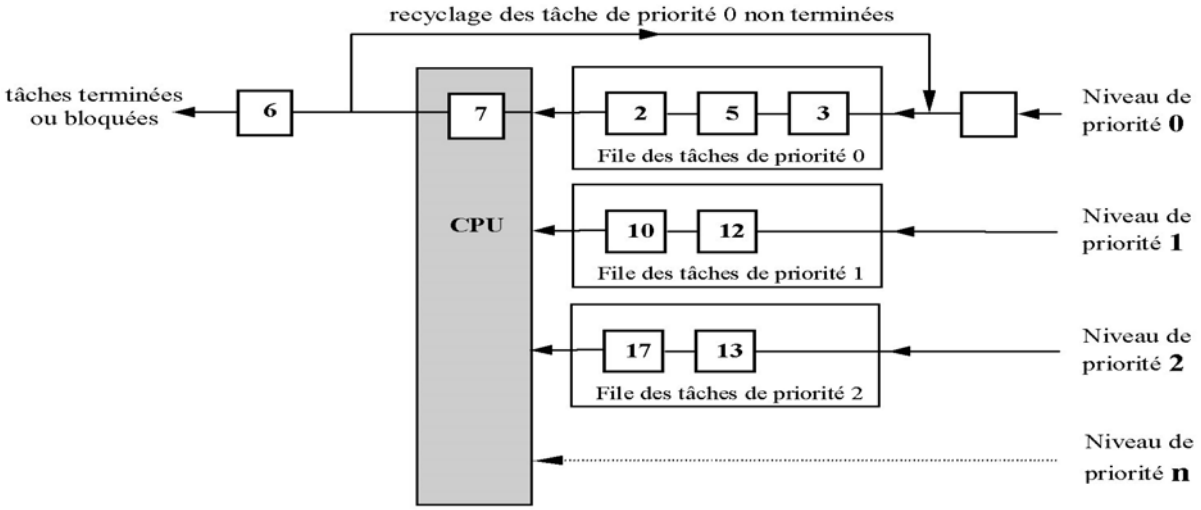
Chaque tâche reçoit à sa création un niveau de priorité.
L'ordonnanceur lance la tâche la plus prioritaire. Elle conserve le processeur tant qu'elle n'est pas terminée ou bloquée.

Il n'y a pas comme avec l'ordonnancement circulaire, plusieurs tâches en exécution apparente.



3.3) Ordonnancement par priorité avec files multiples (MTR86)

C'est une combinaison de l'ordonnancement circulaire et de l'ordonnancement par priorité.
Principe : ici plusieurs tâches peuvent coexister à un même niveau de priorité.



4) Technique du temps partagé et quantum de temps

Dans un noyau utilisant la technique du temps partagé, chaque processus est interrompu toutes les x millisecondes par le système (IT généré par horloge temps réel).
A chaque processus est alloué un quantum de temps Q encore appelé tranche de temps ou time slice. Le quantum alloué est un multiple de l'unité élémentaire de temps du système (le tick).

4.1) Principe du round-robin :

Si une tâche, pendant le quantum de temps qui lui est alloué, est bloquée, cette tâche sera insérée en queue de file d'attente du processeur et ne récupérera un quantum de temps que lorsque tous les autres processus en attente auront utilisé le leur (tous les processus n'utilisent pas forcément la totalité de leur quantum).

Toute tâche encore en activité à la fin de son quantum, est privée du processeur et insérée en queue de file des processus éligibles.

4.2) Choix du quantum de temps :

C'est une constante initialisée au démarrage du système (fichier de configuration).

Critère de choix :

- un quantum trop petit entraîne de fréquentes commutations de tâches. Le processeur risque alors d'être trop accaparé par le dispatcher (responsable de la commutation de contexte).

- un quantum trop grand risque d'allouer le processeur beaucoup plus longtemps que ne le nécessitent les tâches.

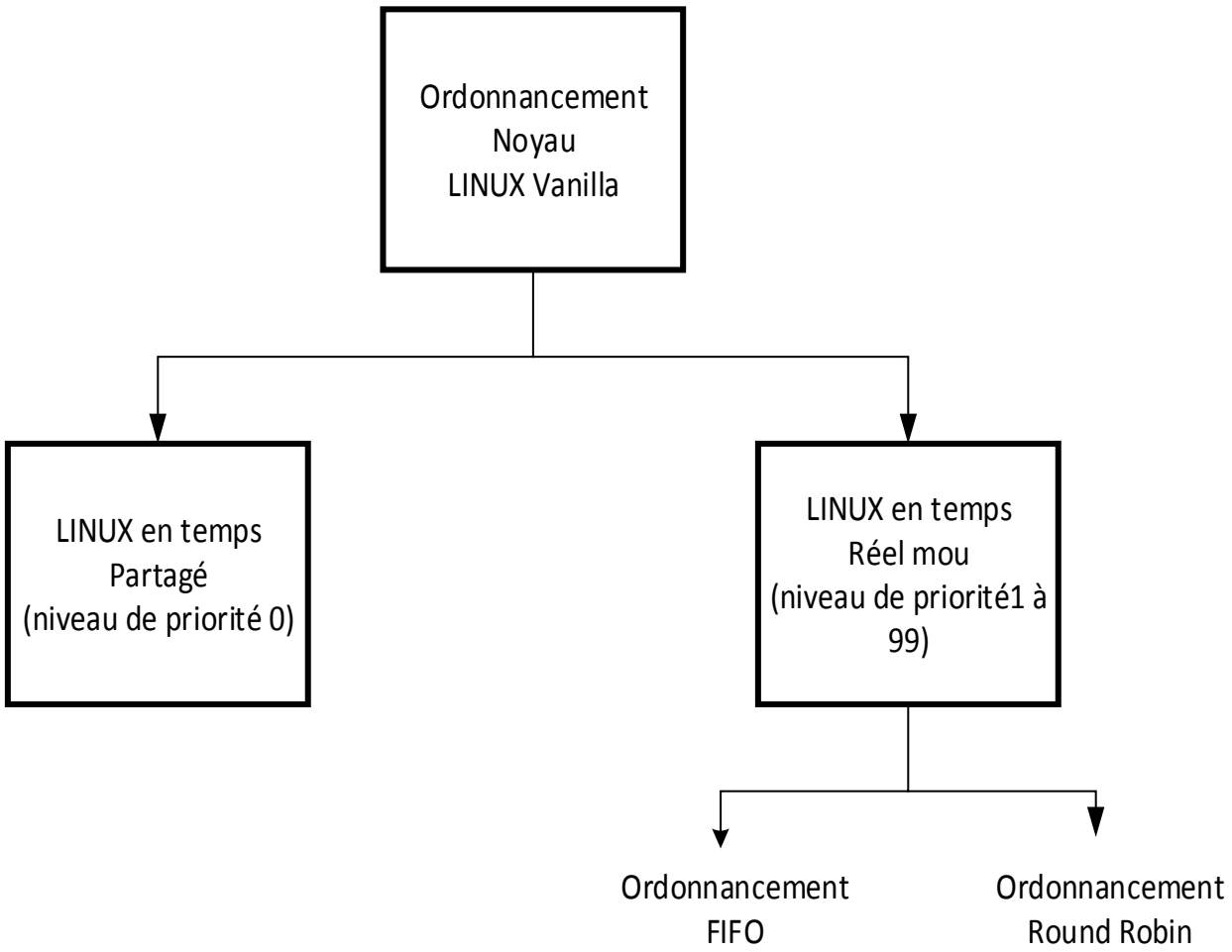
Ex :
Temps de commutation : 0.5 ms (temps nécessaire au dispatcher et ordonnanceur)
- Q = 50 ms

Soient 5 tâches. Chacune d'entre elles attendra 200 ms avant de récupérer un nouveau quantum de temps. Le temps de commutation est ici négligeable.
- Q = 1 ms

Le temps d'attente de chaque tâche sera de 4 ms. Le temps de commutation destâches occupe donc 20% du temps du processeur (overhead important).

5) L'ordonnancement sous LINUX

Au niveau des ressources processeurs LINUX propose trois types de stratégies d'ordonnancement afin d'attribuer les ressources processeur aux processus ou aux threads.



Une stratégie en temps partagé qui reste habituel pour des applicatifs génériques ne disposant pas d'un besoin absolue des ressources processeurs
Une stratégie dite temps réel mou réservé aux threads et processus du noyau mais aussi à certains applicatifs du domaine utilisateur. Les temps de latence sont de l'ordre de la ms.
Une stratégie dite temps réel dur réservé aux threads et processus du noyau mais aussi à certains applicatifs du domaine utilisateur. Les temps de latence sont de l'ordre de la microseconde. Pour le domaine utilisateur le noyau doit être patchés (LINUX-RT, Xenomai….) afin de mener les threads ou les processus du domaine utilisateur vers le temps réel dur.

5.1) La stratégie d'ordonnancement du temps réel partagé

L'idée principale de cette stratégie est de répartir équitablement à tous les processus et threads du domaine utilisateur les ressources processeurs. Chaque tache ou processus doit avoir une période allouée par le processeur pour s'exécuter. Cette tranche de temps allouée est fixe mais certaines taches ou processus ont besoin de plus de temps, une priorité plus importante d'accéder aux ressources du processeur. Ils existent un moyen pour un processus ou une tache dites à temps partagés de laisser plus ou moins passer son tour aux autres, c'est le niveau de courtoisie, un chiffre qui varie de -19 à +19. Une valeur de -19 donnée à un processus le rendra non courtois et il prendra toujours son temps donné et alloué par le processeur sans laisser sa place. Une valeur de 19 fera qu'un processus sera toujours courtois et laissera toujours passer

son tour pour des processus de courtoisie plus élevées. La commande LINUX nice permet de régler ce niveau de courtoisie.

Pour utiliser nice, il suffit de faire précéder la commander à lancer par *nice* suivit de l'option *-n* suivit de la priorité désirée.

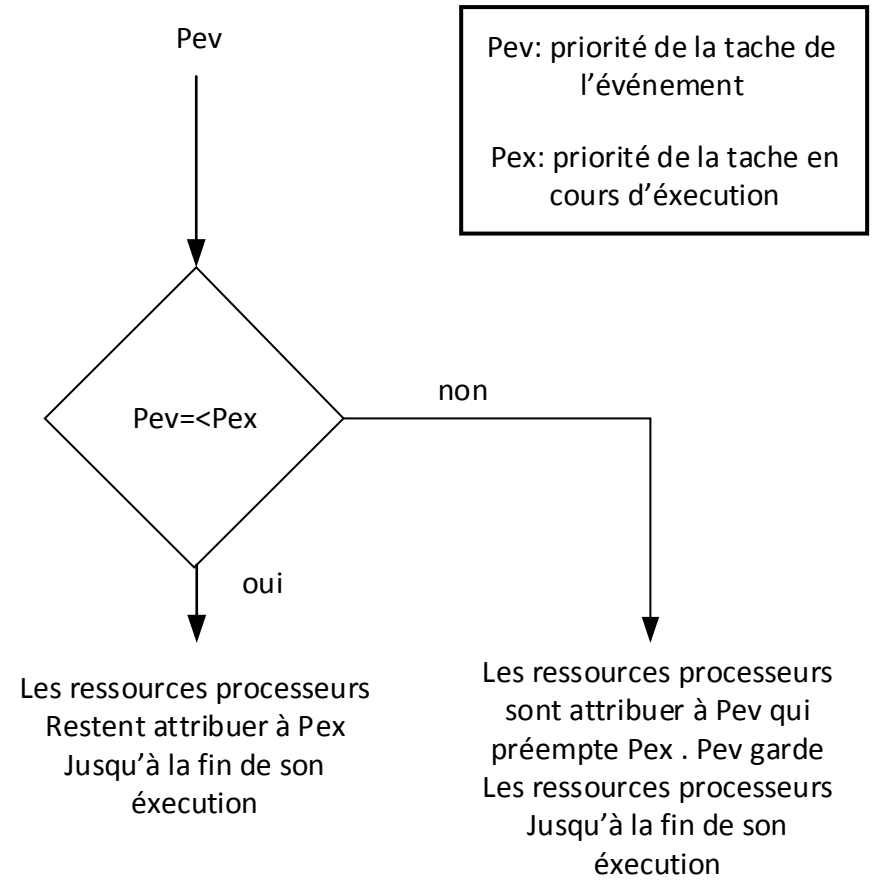
Voici par exemple le lancement du processus top avec la priorité minimale:

$ nice -n 19 ./top

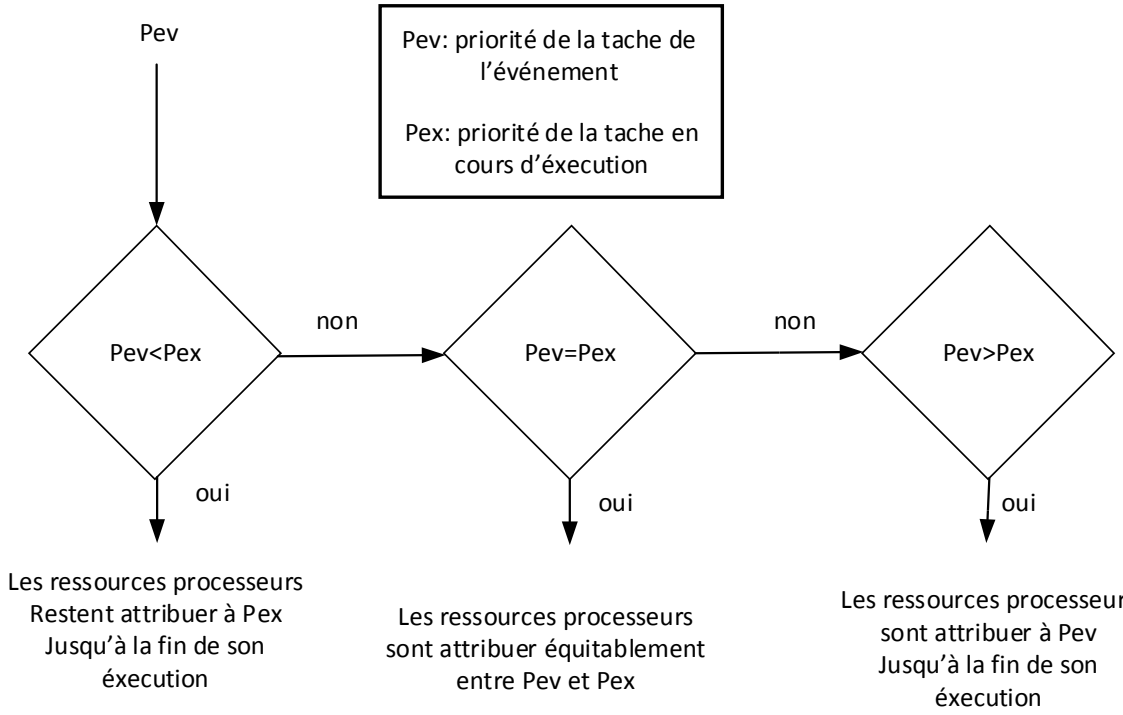
Les taches du temps réel partagé seront préemptés par des taches noyaux ou utilisateur plus prioritaires (taches dites temps réel).

5.2) La stratégie d'ordonnancement dite temps réel mou

Les taches dites temps réel mou ont un niveau de priorité qui varie de 1 à 99 (99 étant le niveau le + prioritaire). Une règle domine pour les taches dites temps réel: la préemption dans le cas où un processus ou une tache plus prioritaire arrive.
Deux algorithmes sont utilisés pour l'ordonnancement des taches temps réels
Le FIFO Ou à priorité égale un processus prendra totalement les ressources processeurs et ne relachera celle-ci que lorsqu'il aura terminé ou lors de sa mise en sommeil.



Le round robin ou chaque processus de priorité identique se partage pendant un temps fixe les ressources processeurs à tour de rôle (temps partagé).



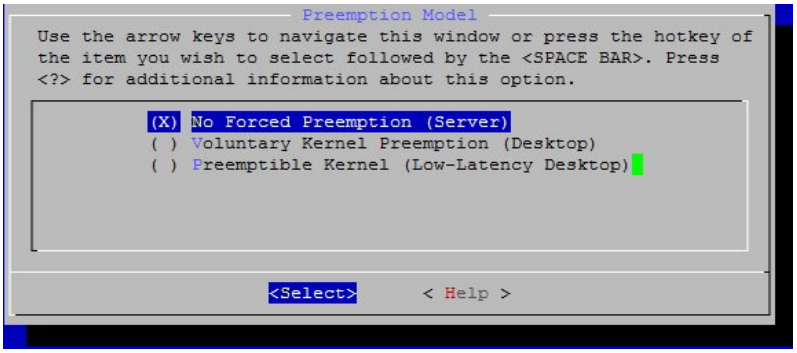
Un processus ou une tache utilisateur de priorité 99 peut être préempté par un processus ou une tache du domaine du noyau. On a donc peu de déterminisme même si le processus est le plus prioritaire.
La commande chrt permet de modifier la priorité d'un processus ou une tache du domaine utilisateur.

```
chrt -r 25 ./led  // algorithme de round robin avec priorité de 25 sur le processus led
chrt -f 26 ./led  // algorithme fifo  avec priorité de 26 sur le processus led.
```

Il est possible de modifier dans le code source la priorité du processus ou d'une tache.
L'exécution en grand nombre des taches temps réel mou vont affamer en ressource processeur les taches en temps partagé de niveau 0. Afin d'éviter la famine totale en ressource processeur des taches dites à temps partagés, 5% des ressources processeur leurs sont encore alloués. Les 95% restants sont laissés aux taches dites temps réels mou.

5.3) Ordonnancement dit temps réel dur

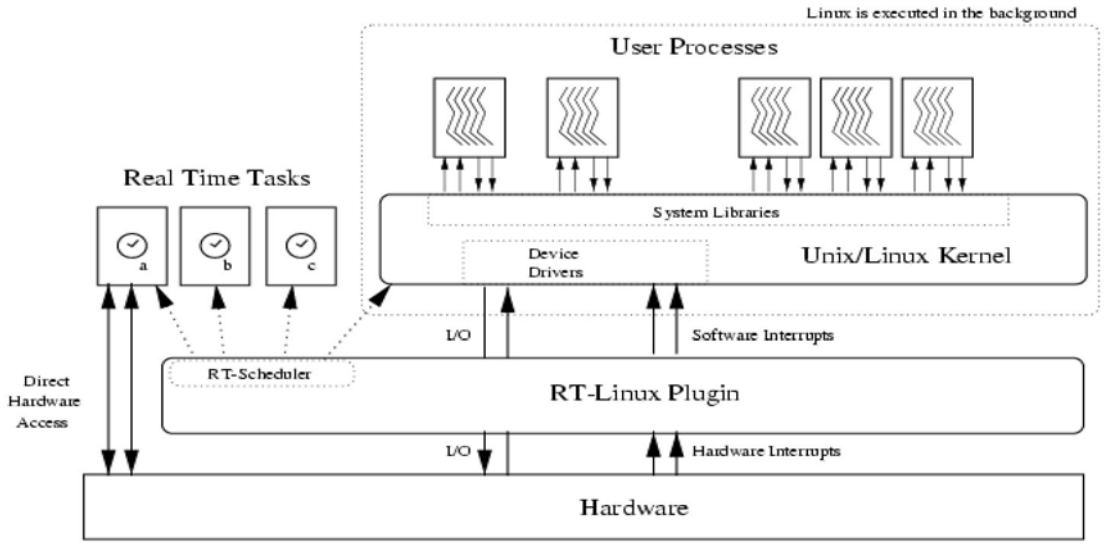
Un noyau LINUX dispose d'un ordonnancement préemptif (Le noyau est capable d'arrêter une tache en cours pour une tache plus prioritaire). La limite se situe dans le fait qu'un processus ou une tache du noyau ne peut être préempté par une tache utilisateur. Depuis le noyau 2.6 lors de la phase de compilation du noyau il est possible de rendre le noyau préemptible en cochant le paramètre preemptible kernel.
La configuration est accessible dans le menu Processor types and features/ Preemption Model de la configuration du noyau.



Un tel noyau permet d'obtenir des performances satisfaisantes dans le cas d'applications de temps réel mou. L'option **no forced preemption** considère les taches du noyau comme non preemtible. . L'option **Voluntary Kernel Preemption** insère des points de *descheduling* supplémentaires. Concrètement, les appels système vers l'espace noyau relancent volontairement le *scheduler* dans leur déroulement. **Dans l'option Preemptible kernel**, tout le noyau (hors sections critiques, handler d'interruptions) devient préemptible, on est passé dans une gestion du temps réel mou (La latence maximale est alors de l'ordre de la milliseconde). Le noyau et ses processus ou taches associés pourront être préemptés par une tâche urgente mais la préemption n'est pas totale et certaines taches ou processus du noyau ne pourront être préemptés (le gestionnaire d'interruptions par exemple).

Dans le cas de temps réel dur (temps de latence de quelques microsecondes), il est nécessaire d'utiliser des extensions. Concrètement, cela correspond à un noyau temps réel auxiliaire chargé dans l'espace du noyau Linux sous forme de modules.

Un micro-kernel RT-LINUX temps réel est inséré entre le hardware et Linux. Linux est une tâche de priorité restreinte. Le micro-kernel capture les interruptions pour ses routines avant Linux qui reçoit alors des interruptions virtuelles (Virtualisation des interruptions)



Les avantages :

- Le micro-noyau garantie des temps de commutation de contexte très courts performances (< 20µs)
- On bénéficie de tout l'environnement Linux (Xwindows. réseau) qui peut communiquer avec le micro-noyau via des IPC spécifiques

Les inconvénients :

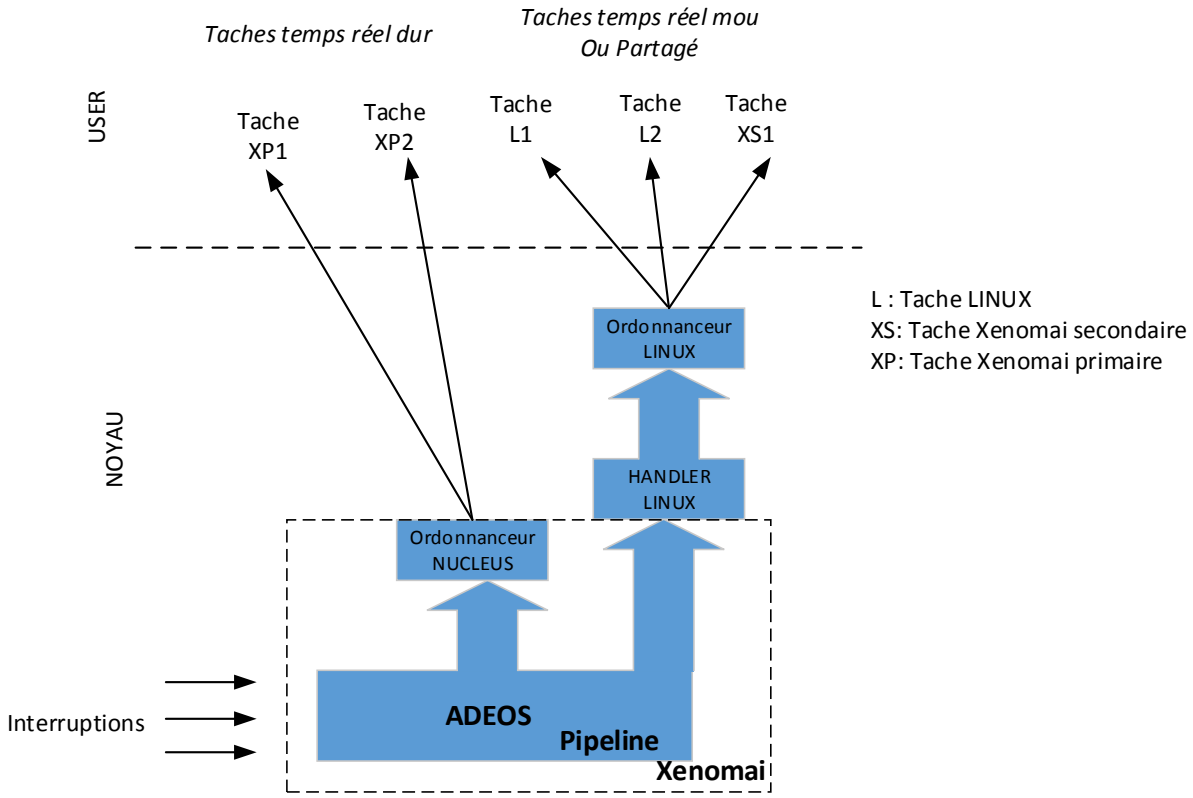
- programmation mode kernel –
- pas de protection de l'intégrité du système
- API limitée et simple comparée à des systèmes propriétaires,
- debug parfois complexe.

5.4) Xenomai

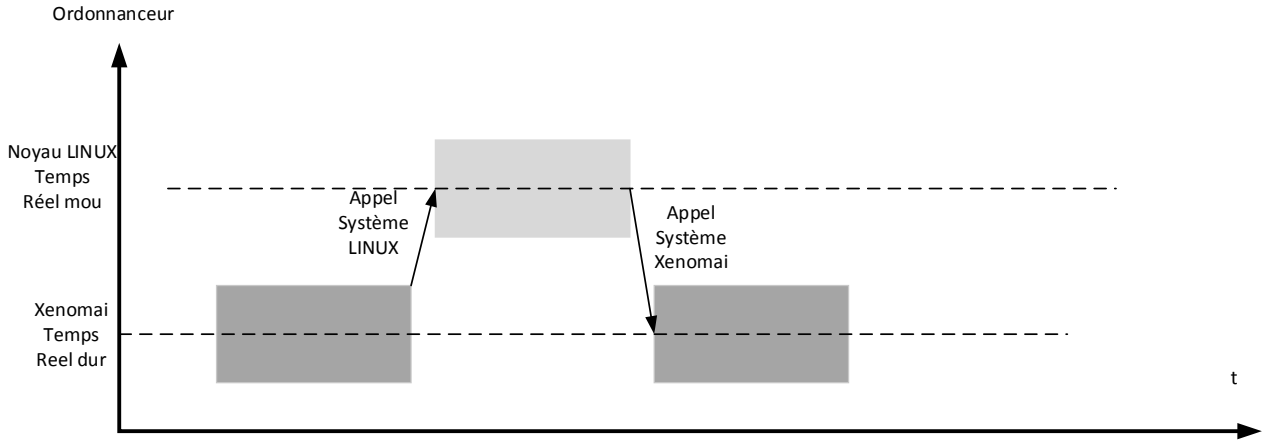
Le composant Xenomai 2, dérivé du projet RTAI permet de développer des programmes temps réel dur dans l'espace utilisateur (et non dans l'espace noyau), ce qui est un avantage notoire.
Xenomai est un système d'exploitation temps réel qui prend la main sur LINUX en considérant LINUX comme une simple tache.
La préemption du noyau est écarté car une tache xenomai sera toujours prioritaire sur une tache LINUX même une tache LINUX exécuté dans l'espace noyau avec des sections critiques.
Les interruptions seront gérés par un nanokernel de Xenomai appelé ADEOS. Celui-ci va intercepté les événements liés aux interruptions pour les dispatcher vers l'OS xenomai ou LINUX.
ADEOS est donc une couche logicielle de Xenomai qui s'interface entre le matériel et les systèmes d'exploitation Xenomai ou LINUX.

5.1) Le lien entre Xenomai et LINUX

Chaque OS dispose de son propre ordonnanceur mais qu'en est-il lorsqu'une tache Xenomai a besoin d'une ressource système par l'intermédiaire d'un appel système à LINUX.
Une tâche créée *via* l'API Xenomai pourra s'exécuter sous Linux, ce mécanisme totalement transparent est appelé *shadow threading*.
On dit qu'une tache ordonnancée par xenomai est dite en mode primaire.
Une tache xenomai ordonnancée par LINUX est dite en mode secondaire.



Le passage d'un mode à l'autre est fonction des ressources accédées et donc des appels système Xenomai ou Linux. Une tache xenomai passera en mode secondaire lors d'un appel système LINUX et repassera en mode primaire lors d'un appel système Xenomai.



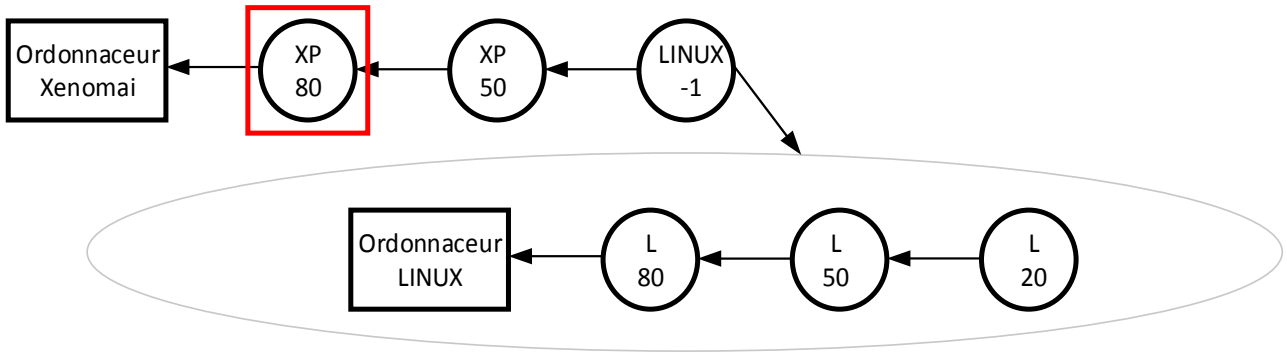
Une tâche en mode secondaire est du temps réel mou et son temps d'exécution est borné (Interrupt Shield). Pour les tâches à contrainte temps réel dur, il faut absolument rester en mode primaire

5.2) L'ordonnancement

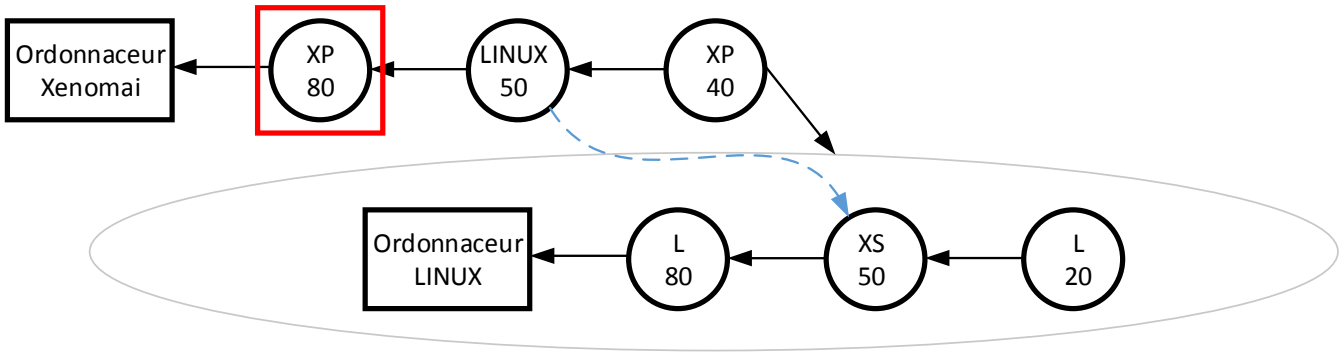
Le noyau Linux est une tâche de fond dont la priorité **évolue dynamiquement** en fonction de la tâche Xenomai de plus haute priorité éligible en mode secondaire. Dans ce cas précis ***une tâche Linux peut préempter une tâche Xenomai.***

La priorité de la tâche Linux dans l'ordonnanceur Xenomai est déterminée de la manière suivante :

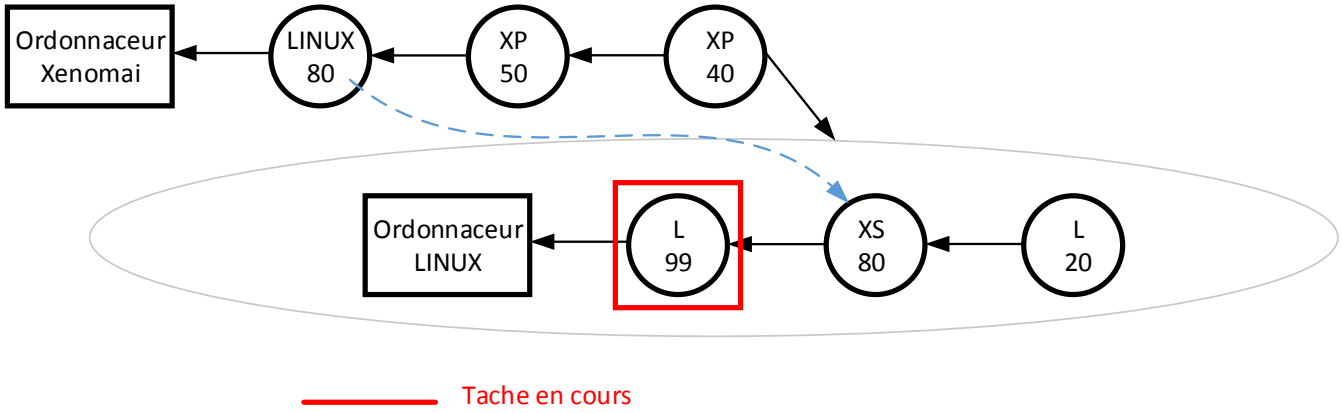
• -1 s'il n'y a aucune tâche en secondary mode (idle Xenomai) LINUX reste une tache de fond la moins prioritaire ;



• la priorité maximale des tâches en secondary mode ou éligible en mode secondaire pour éviter les inversions de priorité.



Les tâches Xenomai en mode secondaire sont en concurrence avec les autres tâches Linux de la classe SCHED_FIFO/RR.
Les priorités ne changent pas d'un scheduler à l'autre, la correspondance est donc immédiate.
Les tâches Xenomai ont donc 99 niveaux de priorité possibles, exactement comme la classe SCHED_FIFO/RR du temps réel mou de LINUX.
Nucleus est le nom du noyau interne Xenomai

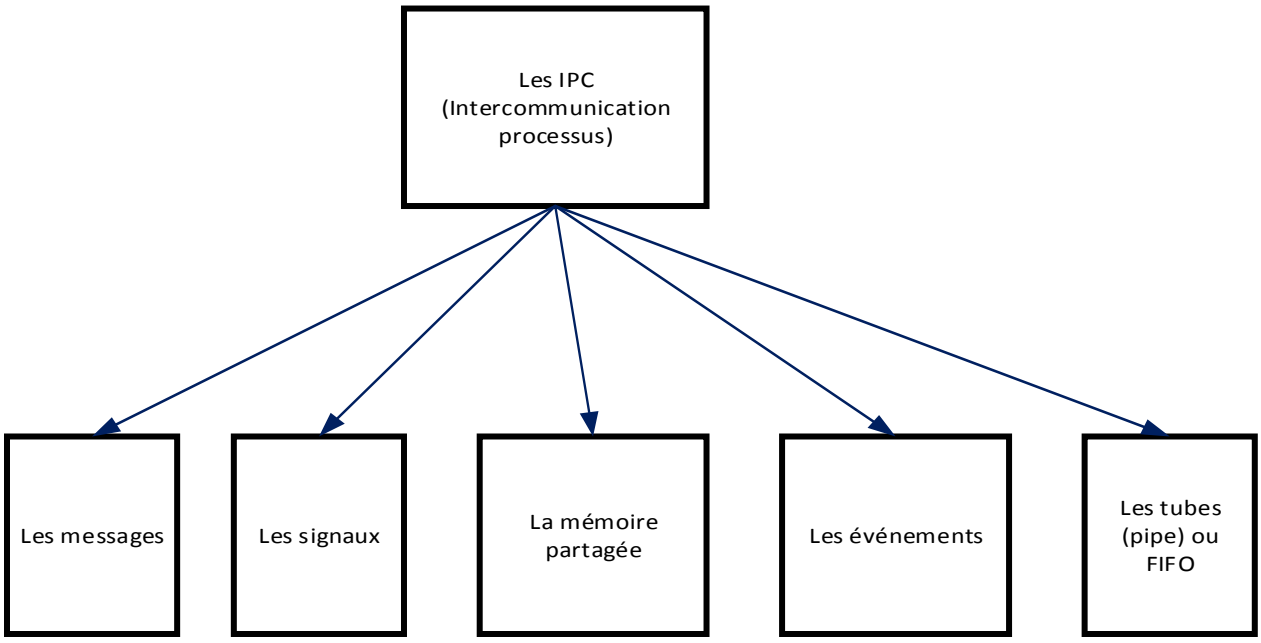


Une tâche Linux peut préempter une tâche Xenomai.

Communication et synchronisation entre processus

1) Communication entre processus

Les tâches ou les processus ont besoin **d'échanger des données** pour coopérer durant l'exécution d'une application.
Les processus communiquent entre eux sous différentes formes de communication interprocessus (IPC).



Un processus a besoin de ***ressources*** pour s'exécuter : mémoire, processeur, mais aussi entrées/sorties (fichiers, communications avec d'autres processus s'exécutant ou non sur la même machine).
Une ressource peut être locale à un processus; dans ce cas, il est seul à l'utiliser et il n'y a pas de problèmes de synchronisation.
Mais dans la plupart des cas, **les ressources sont partagées**. Le nombre de points d'accès d'une ressource est le nombre de processus qui peuvent simultanément l'utiliser. Par exemple, une imprimante ou un processeur sont des ressources à un seul point d'accès. Pour chaque ressource partagée, il faut mettre en place une politique de synchronisation.

Communication et synchronisation entre processus

1. 1) présentation

Les tâches ou les processus ont besoin **d'échanger des données** pour coopérer durant l'exécution d'une application.
Un processus a besoin de ***ressources*** pour s'exécuter : mémoire, processeur, mais aussi entrées/sorties (fichiers, communications avec d'autres processus s'exécutant ou non sur la même machine).
Une ressource peut être locale à un processus; dans ce cas, il est seul à l'utiliser et il n'y a pas de problèmes de synchronisation. Mais dans la plupart des cas, **les ressources sont partagées**.
Le nombre de points d'accès d'une ressource est le nombre de processus qui peuvent simultanément l'utiliser. Par exemple, une imprimante ou un processeur sont des ressources à un seul point d'accès. Pour chaque ressource partagée, il faut mettre en place une politique de synchronisation.

2) synchronisation entre processus

2.1) *généralités*

Il existe 2 types de ressources :

- les **ressources matérielles** (mémoires, registres, organe périphérique, etc.)
- les **ressources logicielles** (fichiers, sous-programmes, buffers de données, etc.)

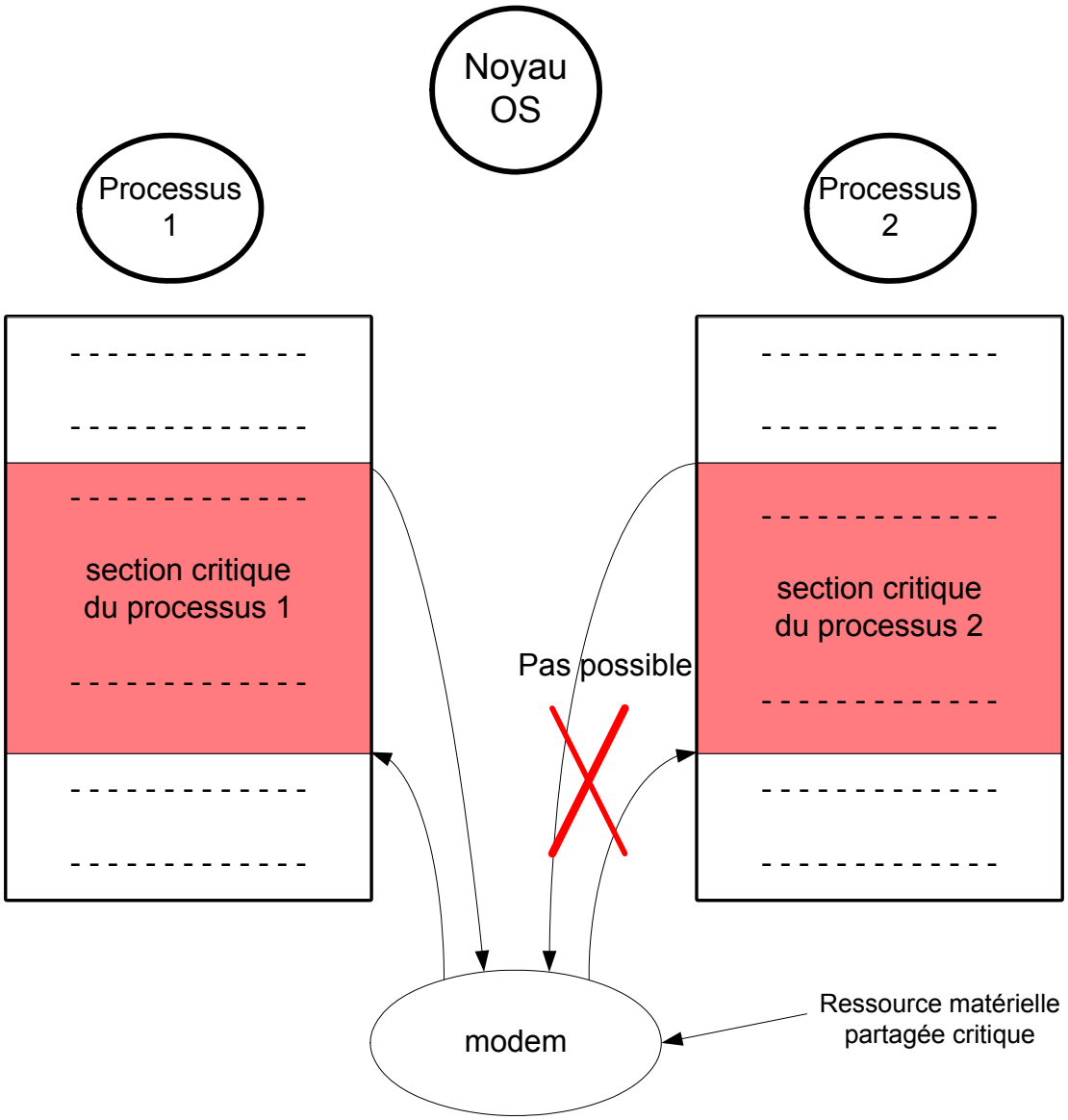
Parmi ces deux types de ressources, il faut distinguer les ressources :

- **locales** ,c'est à dire utilisées par une seule tâche.
- **communes**, c'est à dire **partageables** par plusieurs tâches.

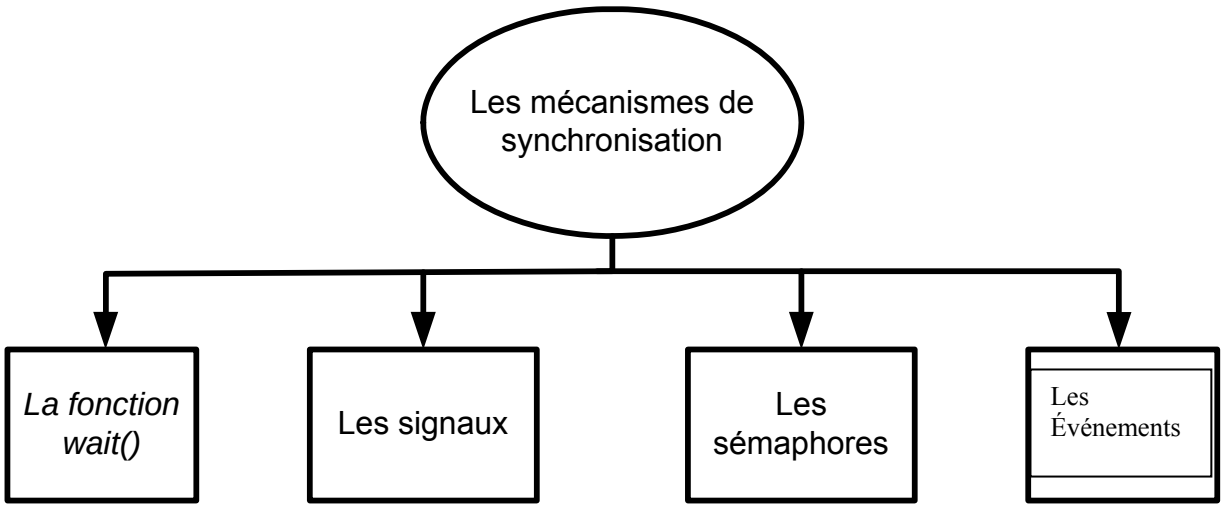
Les ressources partageables sont accessibles par plusieurs tâches :

- Si une seule tâche à la fois peut accéder à la ressource, alors la ressource est **critique** (ou **non réentrante**). Ex: une imprimante.
- Si ***n*** tâches peuvent y accéder à la fois, alors elle est à ***n* entrées**. C'est une ressource partageable **réentrante**.

Lorsque la ressource est **partageable et non réentrante** (ressource critique), il faut faire appel au principe d'**exclusion mutuelle**.
Elle empêche l'accès à la donnée si celle-ci est utilisée par une autre tâche.
La partie du programme conduisant à un conflit d'accès à une ressource est appelée ***section critique***.



Les différentes méthodes de synchronisation entre processus



2.3) Les signaux

Un signal est une petite quantité d'information (de la taille d'un entier court) envoyée entre deux processus. Le signal est la solution la plus simple pour faire communiquer plusieurs processus.
Ils permettent aussi d'avertir simplement un processus qu'un événement est arrivé (Les signaux sont envoyés par le noyau pour indiquer l'arrivée d'un événement).
Il existe environ 32 signaux prédéfinis, ayant chacun une signification. Il n'est pas possible de définir de nouveaux signaux. Les signaux sont envoyés en particulier lorsque le processus tente une opération illégale comme une division par zéro ou un accès à une zone mémoire ne lui appartenant pas.

Manipulation des signaux en langage C

On peut envoyer un signal à un autre processus en appelant la primitive

```
#include <signal.h>
int kill( int pid, int signum )
```

envoie le signal de numéro signum au processus de PID pid.

signum	Nom	Signification
1	HUP	fin de session
8	FPE	exception calcul virgule flottante
9	KILL	fin du processus (non modifiable)
10	USR1	définissable par l'utilisateur
11	SEGV	référence mémoire invalide
12	USR2	définissable par l'utilisateur
17	CHLD	terminaison d'un fils
18	CONT	reprise d'un processus stoppé
19	STOP	stoppe le processus
29	WINCH	redimensionnement de fenêtre

Les signaux entre 0 et 32 sont réservés par le système. L'utilisateur peut donc utiliser tout les autres entre 33 et 255.
- Cette primitive permet à un processus d'envoyer un signal à un autre processus.
pid est le n° du processus visé, signum est le n° du signal employé.
- la valeur de retour est négative en cas d'erreur
- Si signum=0 alors aucun signal n'est émis
→ permet de savoir si le n° pid correspond à un processus

Captation d'un signal

Un processus A doit informer le noyau qu'il est sensible aux différents signaux émis. Pour cela on utilise une structure sigaction. Cette structure définit précisément le comportement à adopter en cas de réception du signal considéré par un processus.

La structure sigaction est composée des champs suivants :

type	nom	particularité
sighandler_t	sa_handler	pointeur sur le gestionnaire du signal,
sigset_t	sa_mask	Type sigset_t, ensemble de signaux qui correspond à la liste des signaux qui sont bloqués pendant l'exécution du gestionnaire
int	sa_flag	le troisième membre sa_flags contient un OU binaire entre différentes constantes permettant de configurer le comportement du gestionnaire de signal par défaut sa_flag=0 comportement normal

La fonction sigaction permet de définir le comportement du processus en fonction des différents signaux. Elle prend trois arguments et renvoie un entier valant 0 si elle réussit, et - 1 si elle échoue. Le premier argument est le numéro du signal, c'est un pointeur sur la structure sigaction. Cette structure définit précisément le comportement à adopter en cas de réception du signal considéré. Le troisième argument sera positionné à la valeur NULL

```
#include <signal.h>
int sigaction (int numero, struct sigaction *nouvelle, NULL) ;
```

Validation des signaux

sigfillset( ) permet de valider tous les signaux connus sur le système

```
#include <signal.h>
int sigfillset(sigset_t *ensemble) ;
```

Mise en place du gestionnaire de signal

Lorsqu'un processus intercepte un signal on lance le gestionnaire de signal qui correspond à la fonction suivante. Le membre sa_handler du type sighandler_t doit pointer sur le gestionnaire de signal.

```
#include <signal.h>
void gestionnaire_signal(int numero, struct siginfo *info, void * inutile) ;
```

Le premier paramètre indique le numéro du signal intercepté. La structure siginfo indique certaines propriétés du signal émis (non utile).

Exemple :

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
Void gestionnaire (int numero)
{
switch (numero) {
        case SIGQUIT :
        printf ("\n SIGQUIT reçu \n");
        break;
        case SIGINT :
        printf ("\n SIGINT reçu \n");
        break;
        }
}
int main (void)
{
struct sigaction action;
action . sa_handler = gestionnaire;
sigfillset(& action.sa_mask);
action . sa_flags = 0;
if (sigaction (SIGQUIT, & action, NULL) != 0) { // action pour le signal SIGQUIT
        printf ("Erreur %d \n", errno);
        exit (0);
        }
while (1)
        {
        sleep(0);
        }

        return 0;
}
```

Les sémaphores

Définition :

Un sémaphore est l'association d'un **compteur "c"** (à valeurs entières) avec une **file d'attente "f"**.

Le compteur peut prendre une valeur soit positive, nulle ou négative.

La file d'attente est de type premier entré premier sorti (FIFO).

Règles de fonctionnement :

Le sémaphore ne peut être modifié que par 2 primitives P(s) et V(s) :

- P(s) décrémente le sémaphore et suspend la tâche appelante s'il est négatif.

- V(s) incrémente le sémaphore et relance une tâche s'il est inférieur ou égal à zéro.

```
P(s):
    DEBUT
    |   c(s) <- c(s) - 1
    |   SI  c(s) < 0
    |   |   ALORS
    |   |     Bloquer la tâche ayant effectué la requête
    |   |     Mettre cette tâche de la file d'attente f(s)
    |   FINSI
    FIN
```

```
V(s) :
    DEBUT
    |   c(s) <- c(s) + 1
    |   SI  c(s) <= 0
    |   |   ALORS
    |   |     Relancer la tâche en attente dans f(s)
    |   FINSI
    FIN
```

Il suffit d'initialiser le sémaphore à 1 et d'encadrer chaque région critique par les deux opérations **P(s)** (*Prendre* la ressource) et **V(s)** (libérer la ressource).

Différents cas de figure :

- Si c(s) > 0 (=1) : aucune tâche n'est en section critique
- Si c(s) < 1 (ou n) : une (ou n) tâche est en section critique:

  ⇒ si c(s) < 0 alors abs(f(s)) = nombre de tâche dans la file d'attente

si c(s) = 0 alors la file d'attente est vide

Les sémaphores sous LINUX

1) Les différentes fonctions

Création d'un sémaphore

```
#include <events.h>
sem_t * sem_open(char * name,0,0644,0);
Char * name  : chaine de caractère correspondant au nom du sémaphore
retourne 0 si une erreur à lieu -1 est retourné
```

Prendre le sémaphore

```
#include <events.h>
int sem_wait (sem_t * semaphore);
retourne 0 si une erreur à lieu -1 est retourné .
```

Libérer les sémaphores

```
#include <events.h>
int sem_post (sem_t * semaphore);
retourne 0 si une erreur à lieu -1 est retourné .
```

fermer le sémaphore

```
#include <events.h>
int sem_close(sem_t * semaphore);
retourne 0 si une erreur à lieu -1 est retourné .
```

3) Exemple

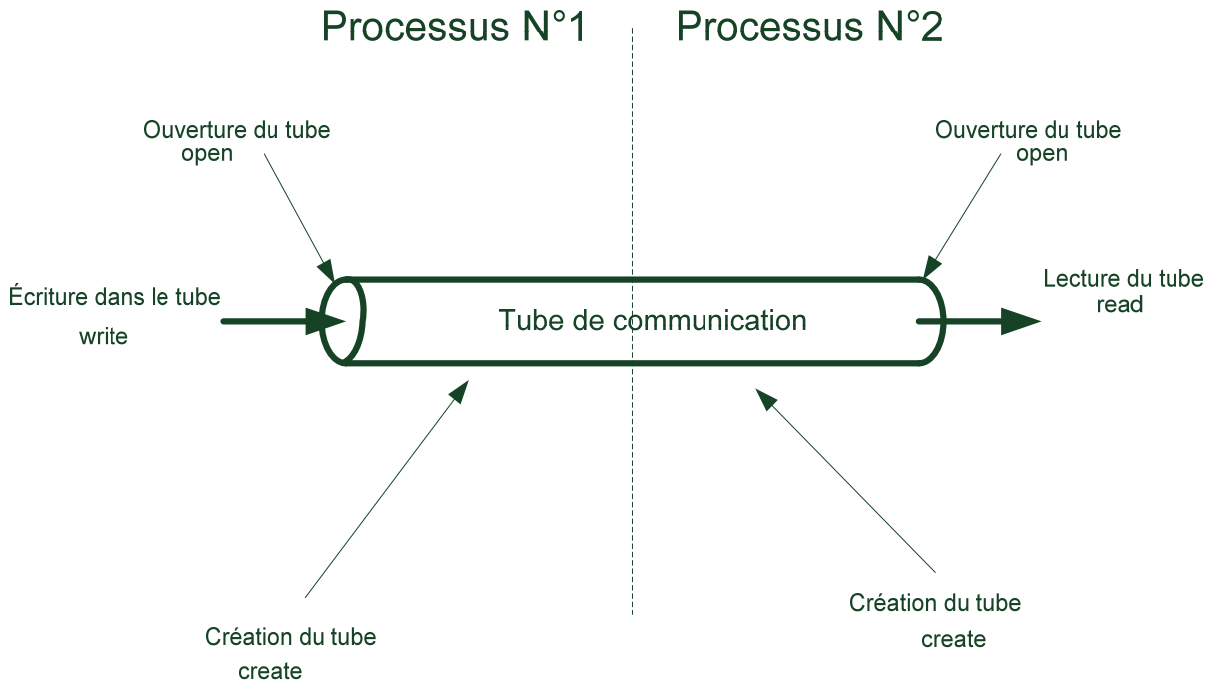
```
char chbusy[]={"sh"};
sem_t *sema1;

void main (void)
{
--------------
--------------
sema1= sem_open(chbusy,0,0644,0);// création du sémaphore
----------------------------
--------------
--------------
sem_wait(sema1);// prendre la sémaphore
 Zone critique1
sem_post(sema1);// libérer le sémaphore
sem_close(sema1);// fermer le semaphore

 }
```

Les tubes

Un tuyau est un canal de communication unidirectionnel et temporaire établi entre un premier processus qui écrit dans le tuyau et un deuxième processus qui lit les données du tuyau.
Les tubes (pipes) constituent le mécanisme fondamental de communication entre processus sous UNIX. Ce sont des files d'octets. Il est conseillé de les utiliser de façon unidirectionnelle. Si l'on veut une communication bidirectionnelle entre deux processus, il faut créer deux tubes, chacun étant utilisé dans un sens. Un tube est implémenté de façon proche d'un fichier.



Les tubes non nommés ne possèdent pas de nom. Il n'existe que le temps de l'exécution du processus qui le crée. Plusieurs processus peuvent lire ou écrire dans un même tube, sans qu'on puisse différencier l'origine des informations en sortie. La capacité d'un tube est limitée (PIPE_BUF octets). Les processus communiquant par un tube non nommé doivent avoir **un lien de parenté** (par exemple descendance d'un ancêtre commun ayant créé ce tube : problème classique de l'héritage des descripteurs de fichiers par un fils).
On crée un tube par un appel à la fonction pipe, déclarée dans unistd.h :
int pipe(int descripteur[2]);
La fonction renvoie 0 si elle réussit, et elle crée alors un nouveau tube. La fonction pipe remplit le tableau descripteur passé en paramètre, avec :
• descripteur[0] désigne la sortie du tube (dans laquelle on peut lire des données) ;
• descripteur[1] désigne l'entrée du tube (dans laquelle on peut écrire des données) ;
Le principe est qu'un processus va écrire dans descripteur[1] et qu'un autre processus va lire les mêmes données dans descripteur[0].

Les fonctions :

```
#include <unistd.h>
int pipe( int fildes[2] );
```

fildes[2] : un tableau ou la fonction peut stocker les descripteurs de fichier pour les terminaisons du tube .

Création d'un tube non nommé

Cette fonction retourne 0 en cas de succès

Par défaut le flag O_NONBLOCK est inactif, tu peux le prépositionné ce flag en utilisant la fonction *fcntl()*

```
#include <unistd.h>
ssize_t read( int fildes, void* buf, size_t nbyte );
```

fildes : Le descripteur de fichier que tu veux lire
buf : Un pointeur sur un buffer ou la fonction peut stocker les données qui sont lues.
Nbyte : Le nombre d'octets que tu veux lire

Cette fonction permet de lire un flux d'octets à partir d'un fichier
Elle retourne le nombre d'octets lus

- Si aucun processus a un tube ouvert pour l'écriture la fonction read retourne 0
- Si un processus a un tube ouvert et un attribut non bloquant O_NONBLOCK prépositionné la fonction read retourne -1 si aucune donné n'a été envoyé dans le tube
- Si un processus a un tube ouvert et un attribut non bloquant O_NONBLOCK non prépositionné la fonction read est bloqué jusqu'à ce que des données soient envoyées dans le tube.

```
#include <unistd.h>
ssize_t write( int fildes, const void* buf, size_t nbytes );
```

fildes : Le descripteur de fichier pour l'écriture
buf : Un pointeur sur un buffer ou la fonction stocke les données pour l'écriture
Nbyte : Le nombre d'octets que tu veux écrire

Cette fonction permet d'écrire un flux d'octets à partir d'un fichier

Si le tube ouvert a prépositionné le flag O_NONBLOCK une requête d'écriture peut causer le blocage du processus tant que les données ne sont pas lues.
Si le tube ouvert a effacé le flag O_NONBLOCK une requête d'écriture ne provoque pas le blocage du processus tant que les données ne sont pas lues.
La fonction retourne le nombre d'octets écrit ou – 1 en cas de problème

Exemple :

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/wait.h>
#include <math.h>
#include <stdlib.h>

int main(void)
{
pid_t pid_fils;
int tube[2];
double x, valeurW, valeurR;
puts("Création d'un tube");
if (pipe(tube) != 0) /* pipe */
{
fprintf(stderr, "Erreur dans pipe\n");
exit(1);
}
        switch(pid_fils = fork()) /* fork */
        {
        case -1 :       perror("Erreur dans fork\n");
                        exit(errno);
        case 0 : /* processus fils */
                        close(tube[1]);
                        read(tube[0], &valeurR, sizeof(double));
                        printf("Le fils (%d) a lu : %.2f\n", getpid(), valeurR);
                        break;
        default : /* processus père */
                        printf("Fermeture sortie dans le père (pid = %d)\n", getpid());
                        close(tube[0]);
                        puts("\nEntrez x :");
                        scanf("%lf", &x);
                        valeurW =sin(x);//Right-click on the project: Properties | Linker |Extra Libs | Add. Then
                //type 'm' in the Name box.
                        //valeurW =(x);
                        write(tube[1], &valeurW, sizeof(double));
                        wait(NULL);
                        break;
        }
return 0;
}
```

Un tube nommé possède un nom dans le système. S'il n'est pas détruit, il persiste dans le système après la fin du processus qui l'a créé. Tout processus possédant les autorisations appropriées peut l'ouvrir en lecture ou en écriture. Un tube nommé conserve toutes les propriétés d'un tube : taille limitée, lecture destructive, lecture/écriture réalisées en mode FIFO. Le tube nommé passe par un fichier sur le disque. L'intérêt est que **les deux processus n'ont pas besoin d'avoir un lien de parenté**. Pour créer un tube nommé, on utilise la fonction mkfifo de la bibliothèque sys/stat.h.

Les fonctions :

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo( const char* path, mode_t mode );
```

Cette fonction crée un tube du type nommé

path : Le nom du chemin ou tu veux utiliser le fichier spécial type FIFO

mode : La permission que tu souhaites donner pour accéder à ce fichier spécial type FIFO

Retourne 0 en cas de succès ou -1 en cas d'echec

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open( const char * path, int oflag );
```

path : Le nom du chemin du fichier que tu veux ouvrir.
oflag : Flag qui spécifie l'état et le mode d'accès au fichier.

Cette fonction retourne le descripteur de fichier

```
#include <stdio.h>

FILE* fdopen( int filedes, const char* mode );
```

Filedes : Le descripteur de fichier que tu veux associer avec un flux binaire
mode : le mode d'accès au fichier

Cette fonction associe un flux de fichier à un descripteur de fichier

Cette fonction retourne un descripteur de flux de fichier en cas de succes – 1 en cas d'échec

```
#include <unistd.h>

int unlink( const char * path );
```

path : le nom du chemin que tu veux effacer

Cette fonction efface un lien à un fichier.

```
/* processus création du tube */

#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>

int main()
{
int fd;
FILE *fp;
char *nomfich="/usr/pipe"; /* nom du fichier attention au nom sous QNX ?*/
if(mkfifo(nomfich, O_NOCTTY | O_NDELAY | O_WRONLY | O_CREAT | O_TRUNC | S_IRUSR |
S_IWUSR | S_IRGRP | S_IWGRP) != 0) /* création du fichier */
{
perror("Problème de création du noeud de tube");
exit(1);
}
fd = open(nomfich, O_WRONLY); /* ouverture en écriture */
fp=fdopen(fd, "w"); /* ouverture du flot */
fprintf(fp, "coucou\n"); /* écriture dans le flot */
unlink(nomfich); /* fermeture du tube */
return 0;
}
```

/* processus lecture du tube */

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>

int main()
{
int fd;
FILE *fp;
char *nomfich="/tmp/test.txt", chaine[50];
fd = open(nomfich, O_RDONLY); /* ouverture du tube */
fp=fdopen(fd, "r"); /* ouverture du flot */
fscanf(fp, "%s", chaine); /* lecture dans le flot */
puts(chaine); /* affichage */
unlink(nomfich); /* fermeture du flot */
return 0;
}
```

LINUX TD N°1: gestion des processus

1) Objectifs

Etablir un programme faisant appels aux fonctions système de LINUX :

- Duplication d'un processus
- Création d'un processus en définissant sa priorité
- Extraction des principales caractéristiques de ce processus

2) Prérequis

- lecture préalable du fascicule les outils de développement
- cours sur le temps réel avec l'ordonnancement, sur le système d'exploitation temps réel LINUX et sur les appels systèmes sous LINUX

3) Présentation

Supposons que La gestion d'un programme est réalisée logiciellement grâce à 4 processus, lourds. Ces processus vont s'exécuter **parallèlement**. En réalité chaque tache va s'exécuter à tour de rôle, ce qui donne à l'utilisateur l'impression d'un déroulement en parallèle. On ne souhaite pas pour l'instant fournir le code de chaque processus mais **comprendre les mécanismes** qui nous permettront de créer ces 4 processus.

4) Questionnaires

utilisez le logiciel putty et x11vnc pour accéder au shell de la beagleboard-XM (interpréteur de commande)

4.1) Créer sous l'éditeur gedit un programme en langage C permettant d'afficher à l'écran je suis le processus N°1 mon PID est mon PPID est . il sera nommé process1.
Ce programme sera sans fin et doit donc posséder une boucle infinie.
Compiler et testez le programme.
Vérifiez que le processus est actif en mémoire grâce à la commande ps –a et donnez la commande qui permet d'éliminer ce processus
Le programme doit être lancé en mode multitâche

4.2) Créez un programme quasi-identique au premier et qui permet d'afficher à l'écran je suis le processus N°2 mon PID est mon PPID est . Il sera nommé process2.

4.3) Complétez le troisième programme page suivante qui permet de lancer les 2 processus créés précédemment. On aurait pu créer les 4 processus mais cela alourdit inutilement ce programme.
Un processus associé au premier programme process1
Un processus associé au deuxième programme process2
On utilisera la fonction fork() et execv() pour lancer les 2 processus
Ce programme permettra aussi de tuer les 2 processus précédents
Au départ le programme proposera le menu suivant :
tapez 1 pour lancer le processus N°1
tapez 2 pour lancer le processus N°2
tapez 3 pour tuer le processus N°1
tapez 4 pour tuer le processus N°2
tapez 5 pour sortir.
Prévoir les messages d'erreurs dans le programme, si le choix de l'opérateur est incohérent.
Lancez ce programme et vérifiez si tout fonctionne bien.

```
#include <fcntl.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <linux/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <sys/mman.h>
#include <ctype.h>
#include <sys/ioctl.h>
#include <stropts.h>
#include <math.h>
int errno=0;
int fid1=0,fid2=0;
char choix;
//char chbusy[]={"ch"};
char * proc1[]={
     "_____________________________",
     0
     };
char * proc2[]={
     "_____________________________",
     "3",
     0
     };
int lance1();
int lance2();
int tue1();
int tue2();
int arret();

int lance2()
{
     int prio=0x80;
     char priorite[4]="128";

     printf("\n lancement du processus ");
     //prio=atoi(pid2);
     fid2=____________;// dupliquer le processus père
     if (fid2==0)
     {
     execv (__________________);// charger le processus fils
     }
     else if (fid2<0){
          printf("impossible de lancer le processus \n");
          return -1;
     }
else return fid2;
}
```

```
int lance1()
{
      int prio=0x80;
      char priorite[4]="128";

      printf("\n lancement du processus \n");

      fid1=_____________;// dupliquer le processus père

      if (fid1==0)
      {
      execv (_____________);// charger le processus fils
      }
      else if (fid1<0){
            printf("impossible de lancer le processus \n");
            return -1;
      }
else return fid1;
}

int tue2()
{     char nno;
      if(_____________________<0) // tuer le processus fils signal 22
      return -1;
      puts(" la tache N2 est tue\n");
      fid2=0;
      return 0;

}

int tue1()
{     char nno;
      if(_____________________0)// tuer le processus fils signal 22
      return -1;
      puts("la tache N1 est tue\n");
      fid1=0;
      return 0;
}

int main(int argc, char *argv[])
{
char choix;
while (1)
{
printf("****************************************\n");
printf("****************************************\n");
printf (" je suis dans le menu l'action desiree \n");
printf (" tapez 1 pour lancer le processus N1 \n");
printf (" tapez 2 pour lancer le processus N2 \n");
printf (" tapez 3 pour tuer le processus N1 \n");
printf (" tapez 4 pour tuer le processus N2\n");
printf (" tapez 5 pour sortir \n");
printf("****************************************\n");
printf("****************************************\n");
scanf ("%c",_________________);
printf (" le choix est %c \n", choix);
```

```
switch (choix)
          {
          case '1' : if(fid1!=0)
                     printf("tache 1 deja en cours\n");
                     else
                     {
                     if((errno=_____________________==-1)
                     printf("impossibilite de lancer la tache\n");
                     else
                     usleep(300000);
                     fid1=errno;
                          }
                          break;
           case '2' : if(fid2!=0)
                     printf("tache 2 deja en cours\n");
                     else
                     {
                     if((_____________________)==-1)
                     printf("impossibilite de lancer la tache\n");
                     else
                     fid2=errno;
                          }
                       break;

           case '3' :     if(fid1==0) printf("tache 1 deja tue\n");
                          else {if ((errno=tue1())!=0)
                          printf("impossibilite de tuer la tache\n");}
                          break;
          case '4':       if(fid2==0) printf("tache 2 deja tue\n");
                          else {if((errno=tue2())!=0)
                          printf("impossibilite de tuer la tache\n"); }
                          break;
          case '5' :      {printf("sortie\n");
                          exit(0);
                          }
}
 }
return 0;
 }
```

TD2 LINUX : Les sémaphores

Présentation

On suppose que deux processus lourds se partage une mémoire de données partagé afin de s'échanger des informations en lecture ou en écriture.
De façon simple le premier processus remplit une mémoire de 26 octets l'alphabet. Le deuxième processus remplit cette mémoire avec que des 1.

mem.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <semaphore.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define SHMSZ 27


int main()
{
char ch;
int shmid,i,fid1;
key_t key;
char * proc1[]={
     "/home/mem1/./mem1",
     0
     };
char *shm,*s;
//name the shared memory segment
key = 1000;
printf("processus pere \n");
//créé le segment de mémoire partagé avec la clé
shmid = shmget(key,SHMSZ,IPC_CREAT|0666);
if(shmid<0)
{
perror("failure in shmget");
exit(-1);
}
//Attache ce segment avec la mémoire virtuelle
shm = shmat(shmid,NULL,0);
//démarre l'écriture dans la mémoire
fid1=fork() ;

     if (fid1==0)
     {
     execv (proc1[0], proc1) ;
     }
     else if (fid1<0){
          printf(« impossible de lancer le processus \n ») ;
```

```
          return -1 ;
     }
usleep(40000) ;
s = shm ;
for(ch='A' ;ch<='Z' ;ch++)
{
*s++ = ch ;
usleep(10000) ;
}
sleep(1) ;
s = shm ;
for(i=0 ;i<26 ;i++)
printf(« %c « ,*s++) ;
shmctl(shmid, IPC_RMID, 0) ;
exit(0) ;
}
```

mem1.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <semaphore.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define SHMSZ 27
char SEM_NAME[]= "vik";
int main()
{
char ch,i;
int shmid;
key_t key;
char *shm,*s;
//nom du segment de mémoire partagée
key = 1000;
printf("processus fils \n");
//créé le segment de mémoire partagée avec la clé
shmid = shmget(key,SHMSZ,0666);
if(shmid<0)
{
printf("reader:failure in shmget \n");
exit(-1);
}
//attache ce segment à la mémoire virtuelle
shm = shmat(shmid,NULL,0);
//démarre l'écriture
s = shm;
for(i=0;i<26;i++)
{
*s++ = '1';
usleep(5000);
}
shmctl(shmid, IPC_RMID, 0);
exit(0);}
```

Testez sur une console terminal ces deux programmes. Le programme père mem sera lancé et créera le processus fils mem1.

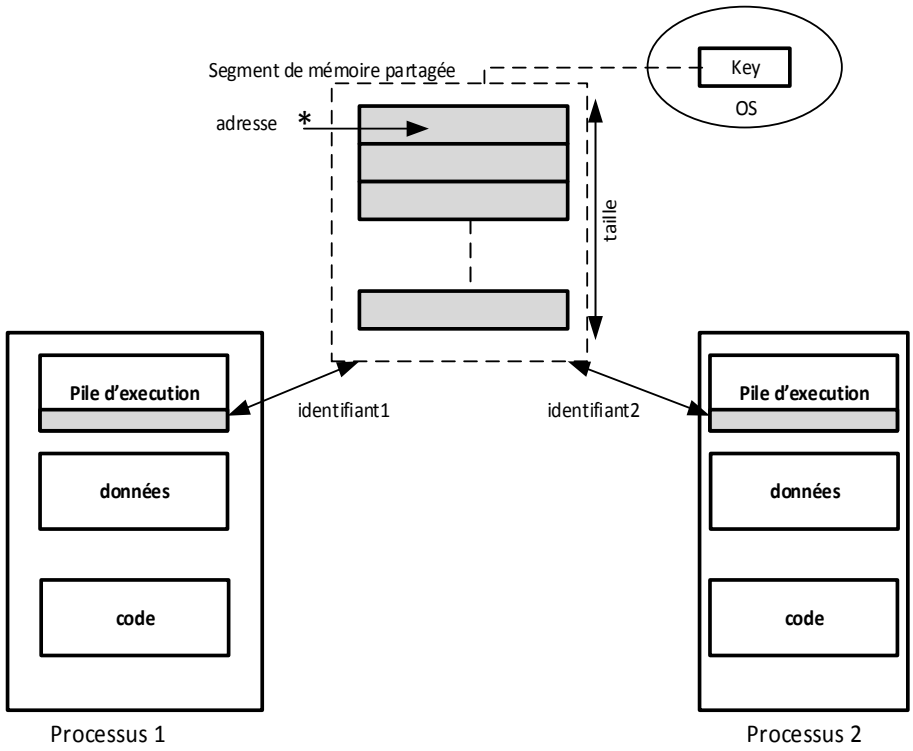
a) Qu'observez-vous et pourquoi
b) Encadrez les zones critiques du programme mem.c et mem1.c

On peut comprendre facilement le danger de ne pas synchroniser les accès en lecture ou en écriture des deux programmes dans une zone de mémoire partagée. Afin de synchroniser les actions de ces deux processus dans cette zone de mémoire partagée on souhaite utiliser le mécanisme des sémaphores.

c) Proposez les modifications logicielles afin d'intégrer les sémaphores dans le processus mem.c et mem1.c

Ressource : Mise en place d'une zone de mémoire partagée

Un segment de mémoire partagé permet à des processus de communiquer ensemble en s'envoyant des informations dans une zone mémoire dit segment de mémoire partagé.



Lien ou création d'un segment de mémoire partagée

int shmget (key_t key, int taille, int attributs);

Le premier argument est la clé transmise pour rechercher ou créer un bloc de mémoire partagée.
Le second argument de cette routine est la taille du segment désiré, en octets.
Le troisième argument correspond aux autorisations d'accès pour accéder à la zone de mémoire partagée. Dans l'appel de fonction on utiliser la constante IPC_CREAT|0666 pour créer et avoir les pleins droits, 0666 pour y accéder avec les pleins droits sans création.
Cette fonction retourne l'identifiant du segment partagé (-1 si erreur)

Une fois obtenu l'identifiant d'un segment partagé, on doit l'attacher dans l'espace mémoire du processus à l'aide de la fonction shmat( )

Attachement d'un segment dans l'espace d'adressage du processus

L'appel-système shmat( ) permet d'attacher le segment dans l'espace d'adressage du processus.

char * shmat (int identifiant, char * adresse, 0);

Premier argument l'identifiant du segment partagé
second argument c'est l'adresse désirée pour l'attachement. Si cette adresse est nulle, le noyau recherche un emplacement libre dans l'espace d'adressage du processus.
La fonction retourne l'adresse du segment partagée

Suppression d'un segment de mémoire partagée

l'appel-système shmctl( ) permet de paramétrer ou de supprimer un segment partagé

int shmctl (int identifiant, int commande, 0);

Premier paramètre l'identifiant du segment partagé
Le second paramètre détermine la commande que doit effectuer la fonction shmctl. Le passage lors de l'appel de fonction de La constante IPC_RMID permet de détruire le segment partagée.

Exemple d'utilisation :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <semaphore.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define SHMSZ 100 // taille de la mémoire
char SEM_NAME[]= "vik";
int main()
{
char ch,i=0;
int shmid;
key_t key;
char *shm,*s;
char tab[]="bonjour"
//nom du segment de mémoire partagée
key = 800;
//crée le segment de mémoire partagé avec la clé
shmid = shmget(key,SHMSZ,IPC_CREAT|0666);
 //attacher ce segment à une mémoire virtuelle
shm = shmat(shmid,NULL,0);
```

```
s = shm;

do
{
*s++=tab[i++] ;

}while(tab[i] !=0x00) ;

shmctl(shmid, IPC_RMID, 0);// détruire le segment partagé

exit(0);
}
```

LINUX TD N°3 : Les signaux

1) Objectifs

Etablir un programme sensible à un signal envoyé par une commande shell ou un programme

2) Prérequis

- cours sur le temps réel avec l'ordonnancement, sur le système d'exploitation temps réel LINUX et sur les appels systèmes sous LINUX

3) Présentation

Pour que le système multitâche ou plutôt multiprocessus de LINUX puisse fonctionner, il faut que chaque processus puisse communiquer entre eux. Le moyen le plus simple pour cela est d'utiliser le signal. Par exemple lorsque le processus gestion de l'alarme détecte une anomalie il doit indiquer au processus visualisation d'activer les avertisseurs sonores ou visuelles.

4) Préliminaires

On a créé un programme nommée sendsig permettant d'envoyer un signal vers un processus.
Ce programme est donné il s'agira lors du TD d'utiliser le programme sendsig. Une fois exécuté
Le programme attend en ligne de commande les directives suivantes :
sendsig numéro du PID du processus numéro du signal.

5) questionnaires

5.1) Créer un programme en langage C visu qui permet d'effectuer les actions suivantes lorsqu'il reçoit un signal :
si ce signal a comme numéro 60 écrire à l'écran alarme active.
si ce signal a comme numéro 61 écrire à l'écran alarme désactivé.
si ce signal a comme numéro 62 écrire tuer le processus.
Compilez ce programme.
Lancez ce programme en mode multitâche et vérifiez son bon fonctionnement grâce à la commande sendsig. Ce programme sera nommé visualisation.

5.2) Le programme hydro.c qui suit propose à l'utilisateur le menu suivant :

Tapez 1 pour activer l'alarme
Tapez 2 pour désactiver l'alarme
Tapez 3 pour tuer le processus visu
Tapez 4 pour tuer le processus père
Bien entendu suivant les choix précédents on doit respectivement lancer les actions suivantes :

- Envoyer le signal 60 vers le processus visu
- Envoyer le signal 61 vers le processus visu
- Envoyer le signal 62 vers le processus visu

Le processus père hydro lancera la processus fils visu
Complétez le programme suivant afin qu'il corresponde au cahier des charges précédents :
Compilez et testez le bon fonctionnement du processus hydro.

HYDRO.C

```
#include <fcntl.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <linux/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <sys/mman.h>
#include <ctype.h>
#include <sys/ioctl.h>
#include <stropts.h>
#include <math.h>


char * proc1[]={
     "_____________________",
     "128"
     };


void main (void)
{
int fid1;
char choix;
setbuf(stdin,NULL);
setbuf(stdout,NULL);
fid1=fork();

     if (fid1==0)
     {
     execv (proc1[0], proc1);
     }
     else if (fid1<0){
          printf("impossible de lancer le processus \n");
          return -1;
```

```
     }
/*d=atoi(proc1[1]);*/
while(1)
{
printf (" tapez 1 pour activez l'alarme \n");
printf (" tapez 2 pour desactivez l'alarme \n");
printf (" tapez 3 pour tuer le processus visu \n");
printf (" tapez 4 pour tuer le processus pere \n");
scanf("%c",&choix);
switch (choix){
case '1': {_____________________;break;}
case '2': {_____________________;break;}
case '3': {_____________________;break;}
case '4':exit(0);
default: printf("mauchais choix saisi\n");}
}
}
```

visu.c

```
#include <fcntl.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <linux/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <sys/mman.h>
#include <ctype.h>
#include <sys/ioctl.h>
#include <stropts.h>
#include <math.h>



void handler_sigrtmax(int num, siginfo_t * info, void * unused)
{
static int compteur = 0;
_____________________________________

_____________________________________

printf("la valeur du signal est %d\n",num);
}
```

```
int main(int argc, char *argv[])
{
     int sig,i;
unsigned char devid=0x00;
unsigned char donnee[14];

struct sigaction action;
sigfillset(& action.sa_mask);
action.sa_sigaction = _____________________;// gestionnaire de signaux
action.sa_flags = 0;
for (i = 1; i < 512; i++)
sigaction(_____________________);
printf ("Fils : PID=%d, pere: PPID=%d\n",getpid( ) , getppid( ));


     while (1)
     {
     sleep(0);
     }

     return 0;
}
```