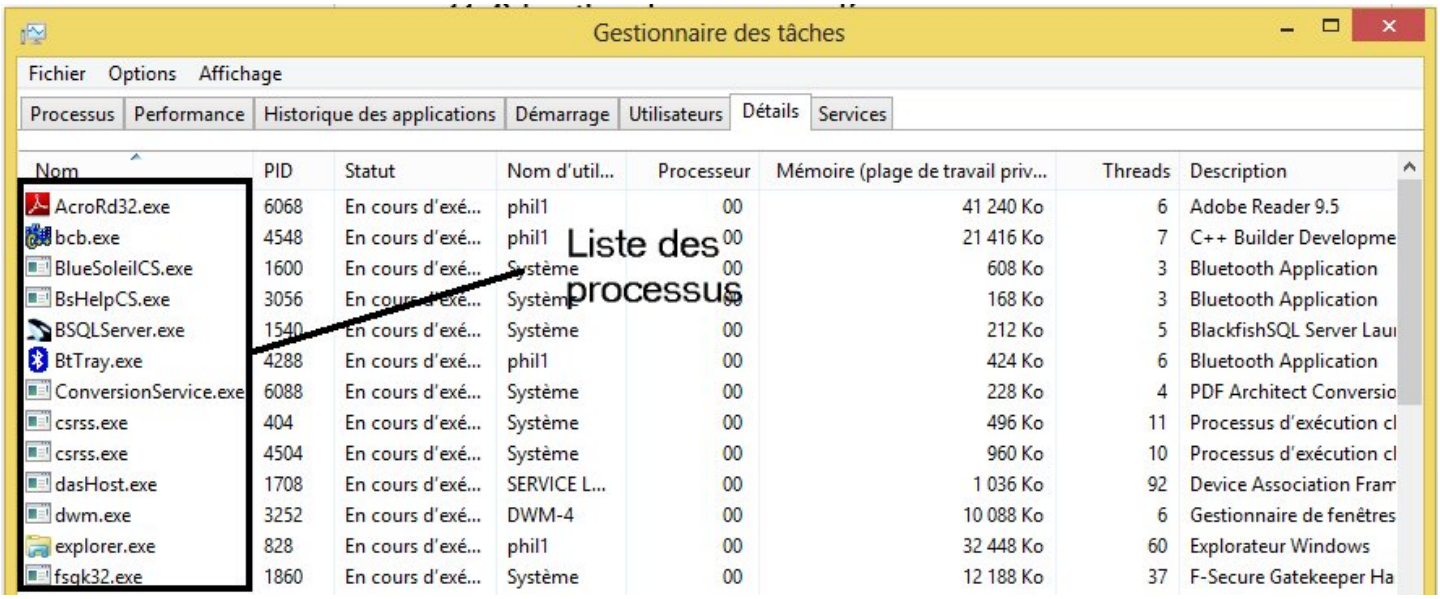
Les Processus

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 :

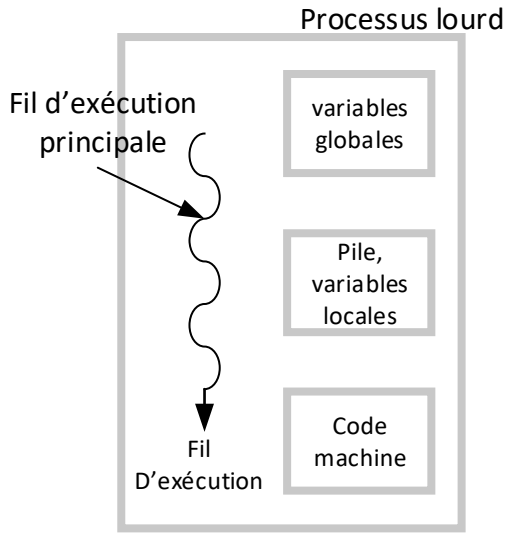
- Leur création nécessite des appels systèmes coûteux en temps,
- Le changement de contexte entre processus est une opération lente, en particulier pour de nombreux transferts en mémoire,
- Le coût des mécanismes de protection associés au processus,
- L'interaction, la synchronisation ou la communication 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 :

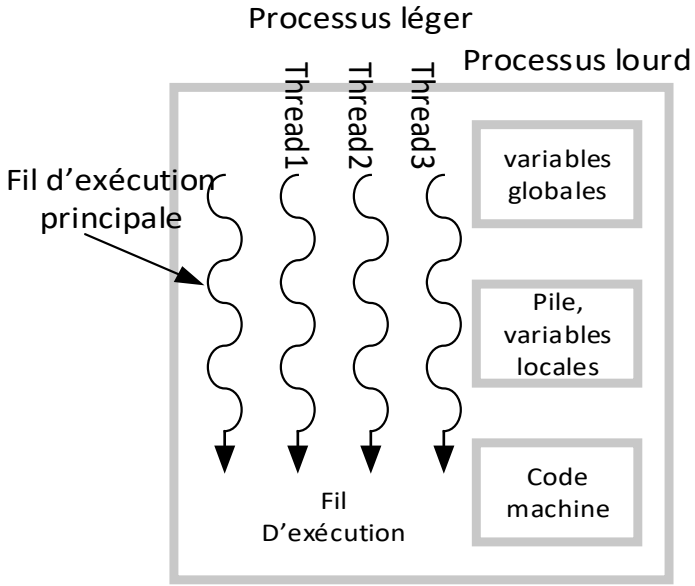


Nom	PID	Statut	Nom d'util...	Processeur	Mémoire (plage de travail priv...	Threads	Description
AcroRd32.exe	6068	En cours d'exé...	phil1	00	41 240 Ko	6	Adobe Reader 9.5
bcb.exe	4548	En cours d'exé...	phil1	00	21 416 Ko	7	C++ Builder Developme
BlueSoleilCS.exe	1600	En cours d'exé...	Système	00	608 Ko	3	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

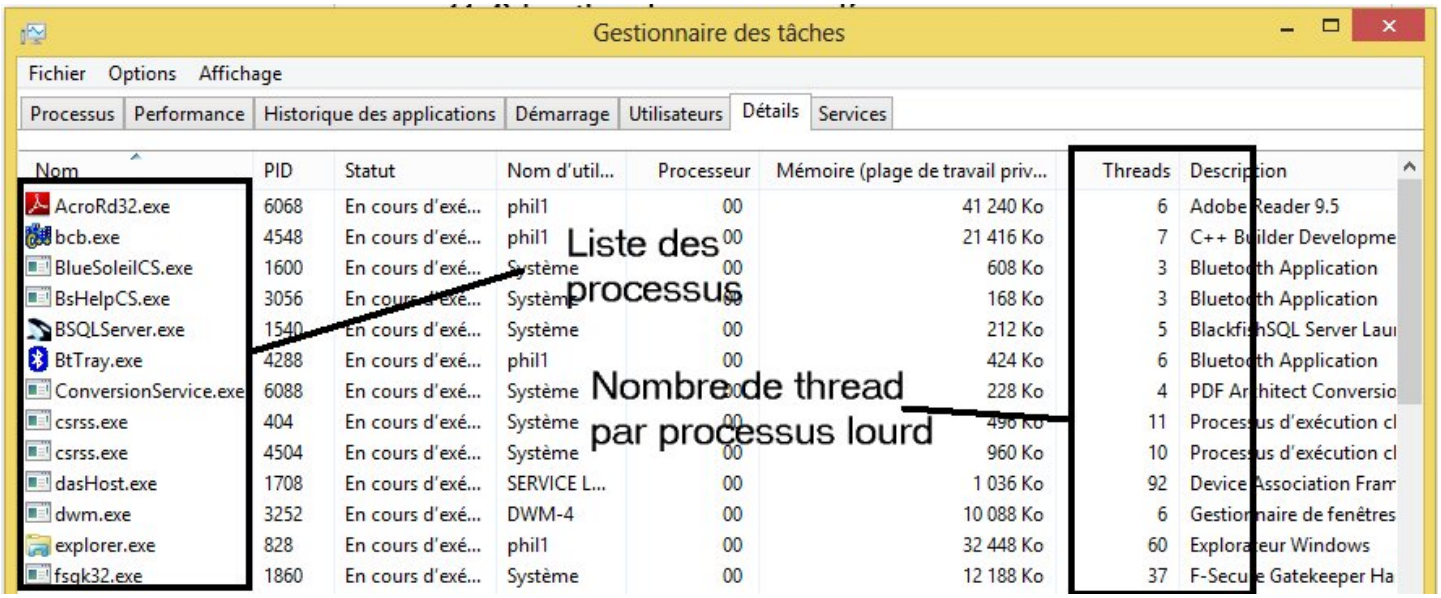
Un processus lourd classique, ne contenant qu'un seul fil d'exécution, est dit **monoprogrammé** :
L'exécution du code du processus est réalisée de manière séquentielle par un fil de contrôle (*thread of control*).
C'est un programme monolithique comprenant un corps principal la fonction main()



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

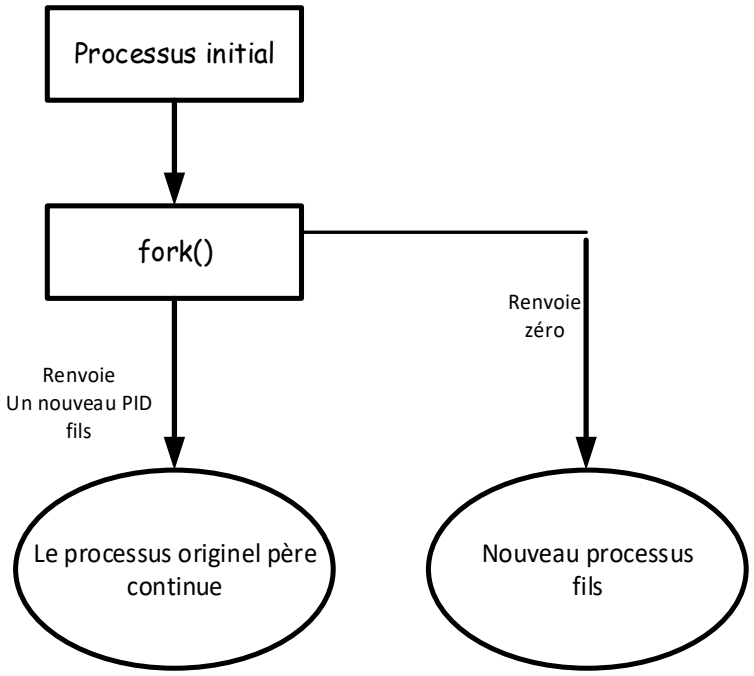
int fork() *pour LINUX (librairie sys_clib.l)*

- 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 };
printf ("ok \n Tentative d'exécuter exemple_execv ... ");
fork() ;
//pid=execv (nv_argv[0 ], nv_argv);
pid=execv { "/home/TD1_A/./exemple_execvp",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

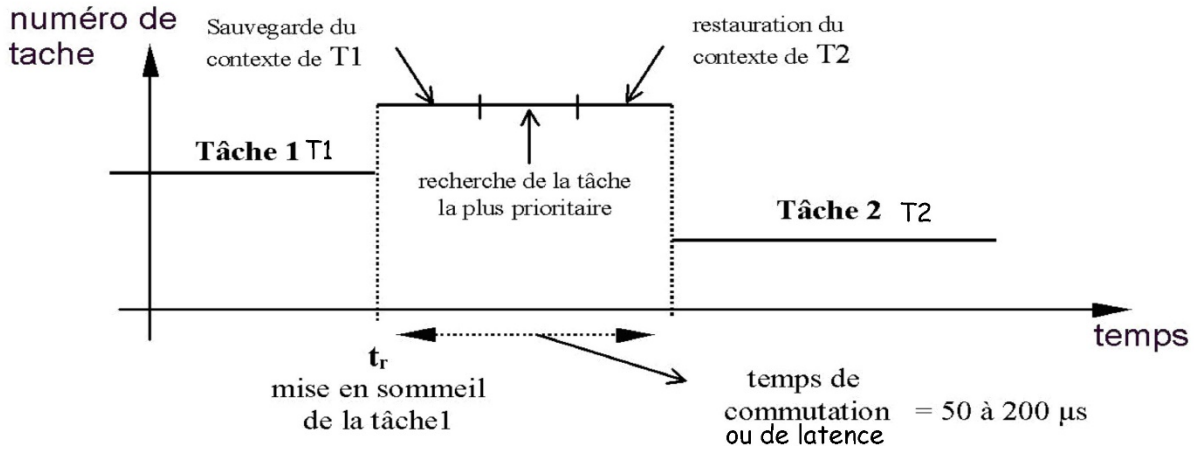
2 - Passage de l'état éligible à l'état élu

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

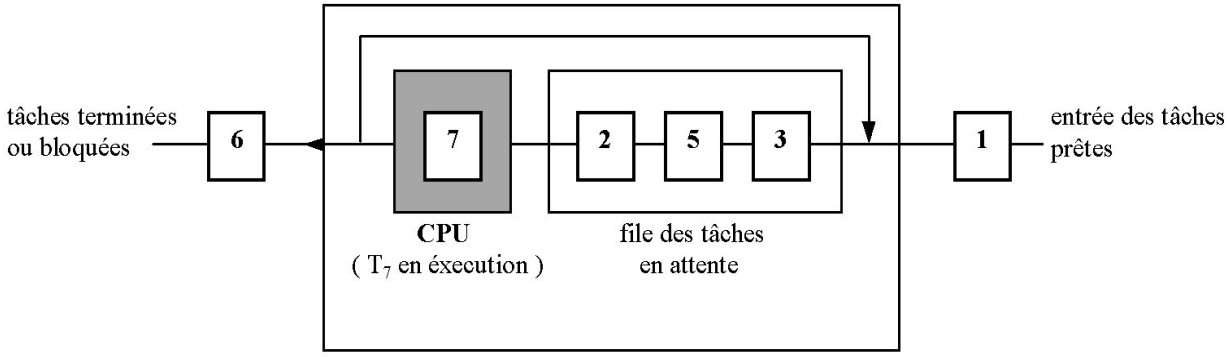
- Ordonnancement circulaire
- Ordonnancement par priorité
- Ordonnancement par priorité avec file multiple

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

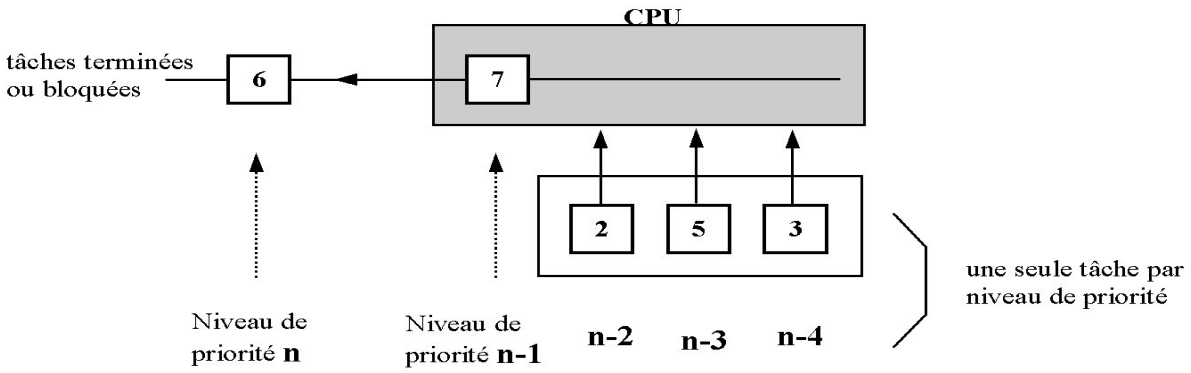
3.2) Ordonnancement par priorité

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

Principe :

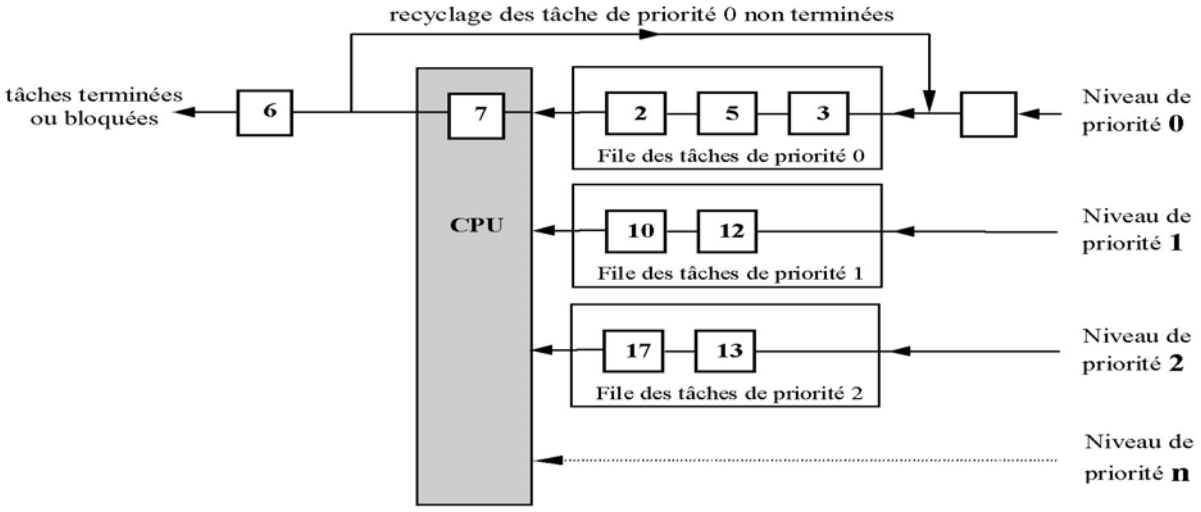
Chaque tâche reçoit à sa création un niveau de priorité.
L'ordonnanceur lance la tâche la plus prioritaire. Elle conserve le processeur tant qu'elle n'est pas terminée ou bloquée.

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.

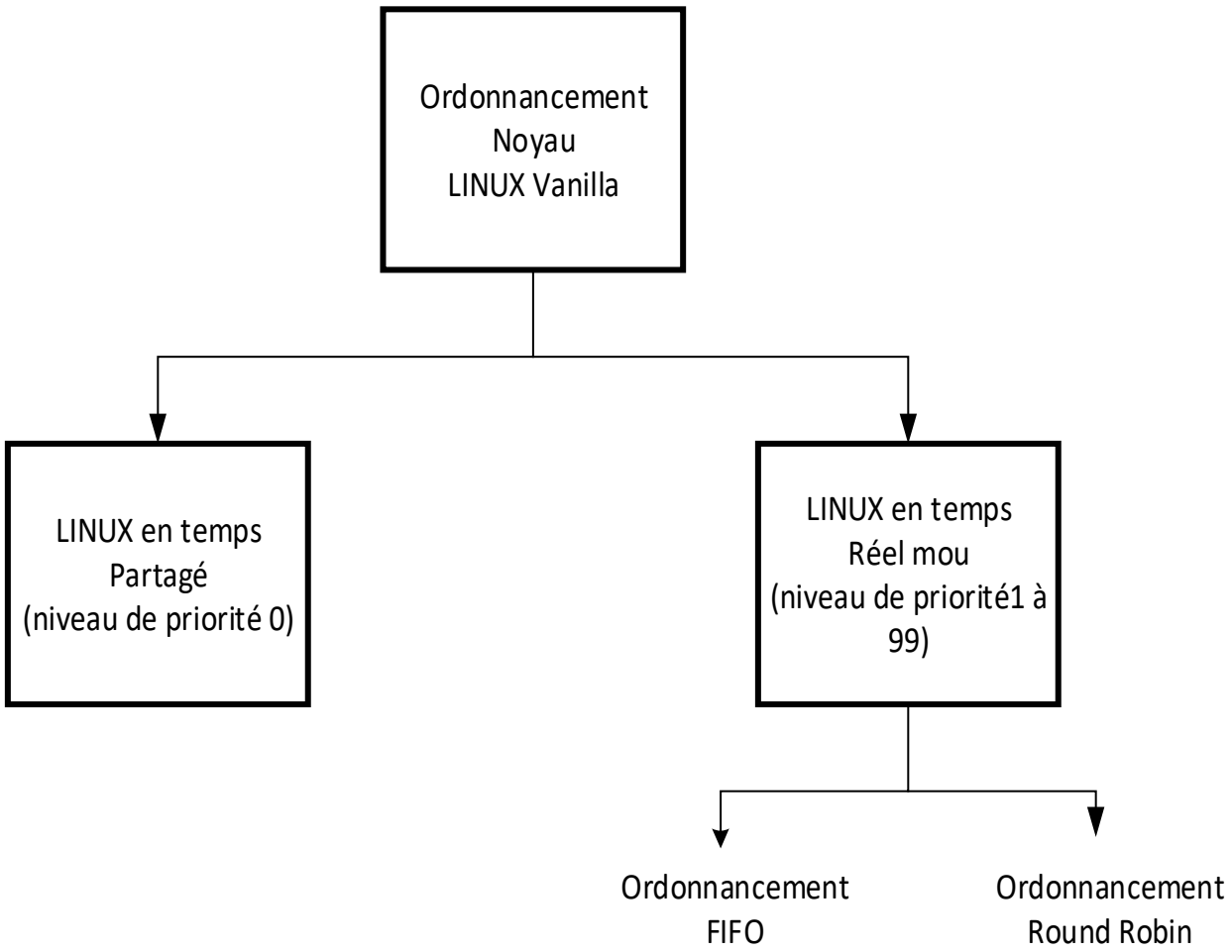
Ex :
Temps de commutation : 0.5 ms (temps nécessaire au dispatcher et ordonnanceur)
- Q = 50 ms

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 habituelle pour des applicatifs génériques ne disposant pas d'un besoin absolu 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.

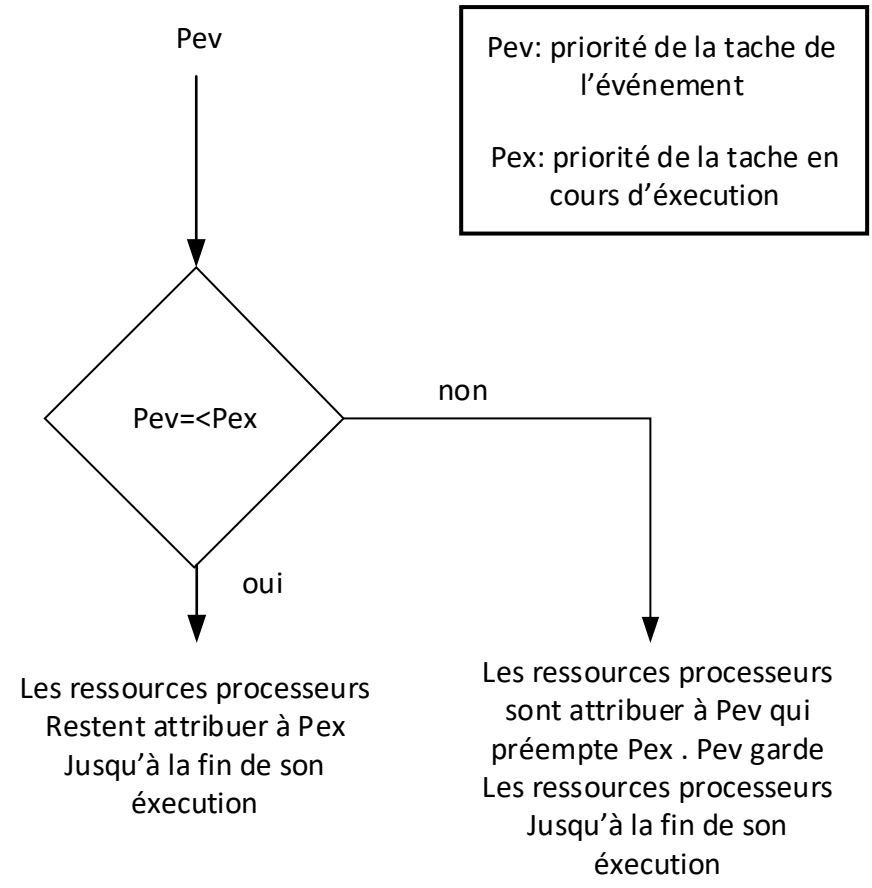
Voici par exemple le lancement du processus top avec la priorité minimale:

$ nice -n 19 ./top

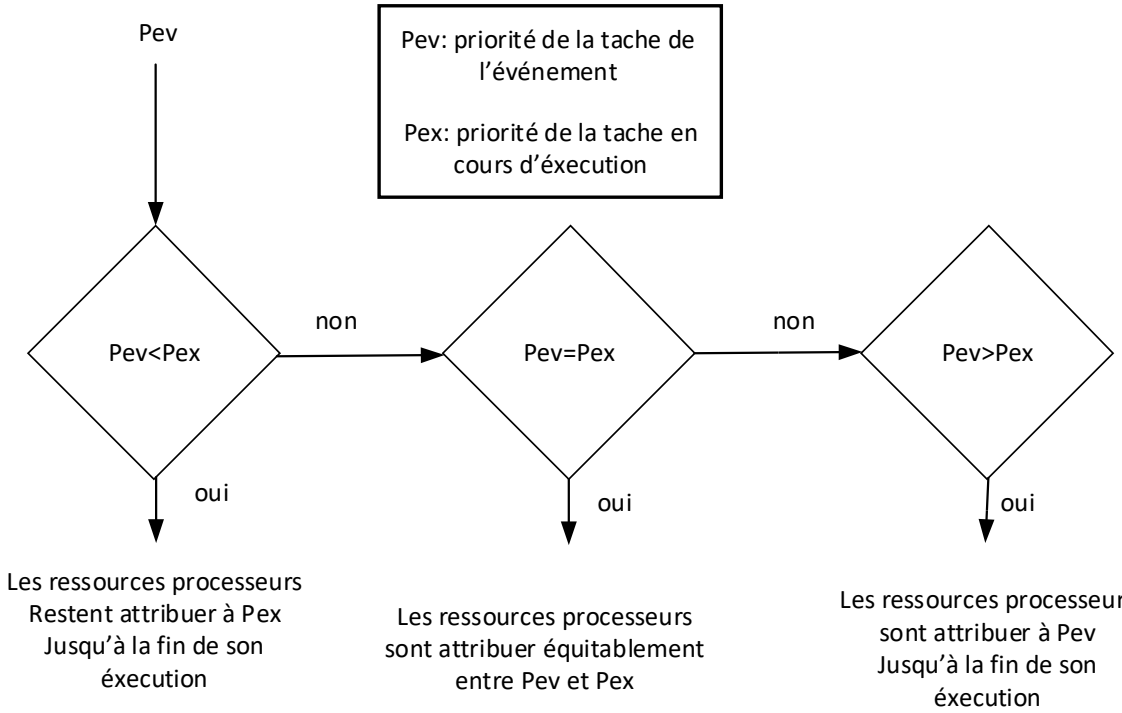
Les taches du temps réel partagé seront préemptées par des taches noyaux ou utilisateur plus prioritaires (taches dites temps réel).

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).



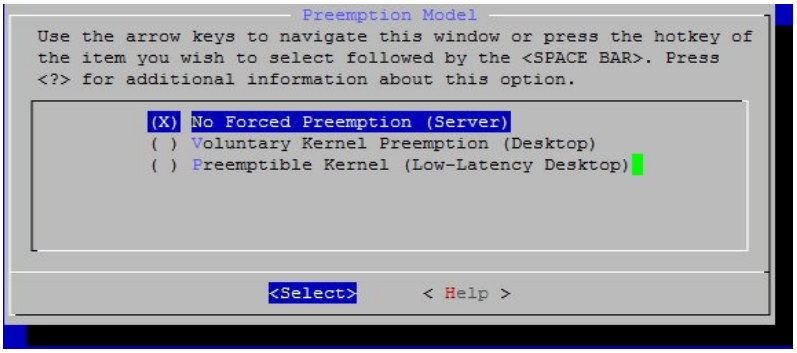
Un processus ou une tache utilisateur de priorité 99 peut être préempté par un processus ou une tache du domaine du noyau. On a donc peu de déterminisme même si le processus est le plus prioritaire.
La commande chrt permet de modifier la priorité d'un processus ou une tache du domaine utilisateur.

```
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

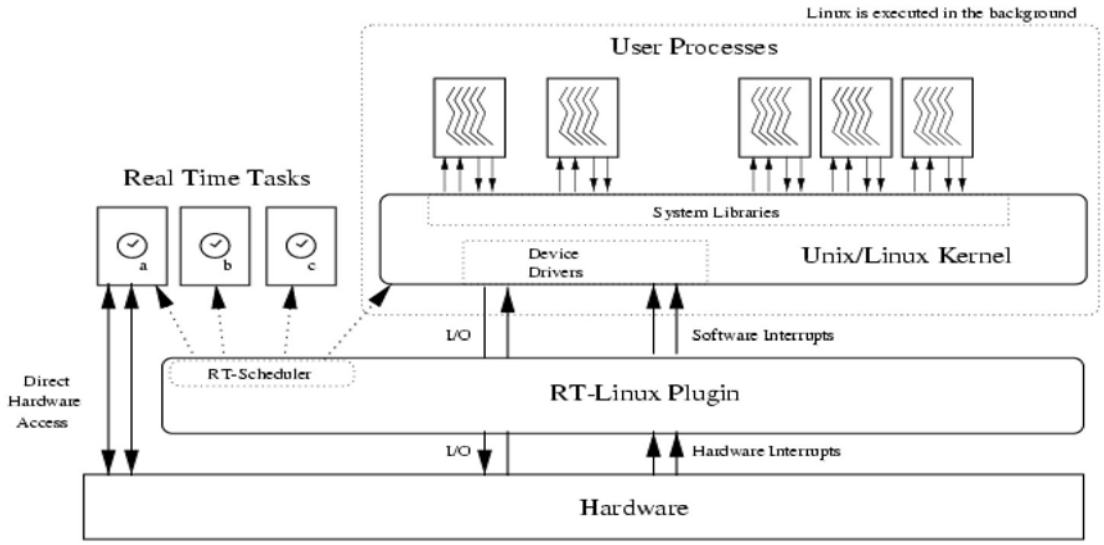
Un noyau LINUX dispose d'un ordonnancement préemptif (Le noyau est capable d'arrêter une tache en cours pour une tache plus prioritaire). La limite se situe dans le fait qu'un processus ou une tache du noyau ne peut être préempté par une tache utilisateur. Depuis le noyau 2.6 lors de la phase de compilation 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 :

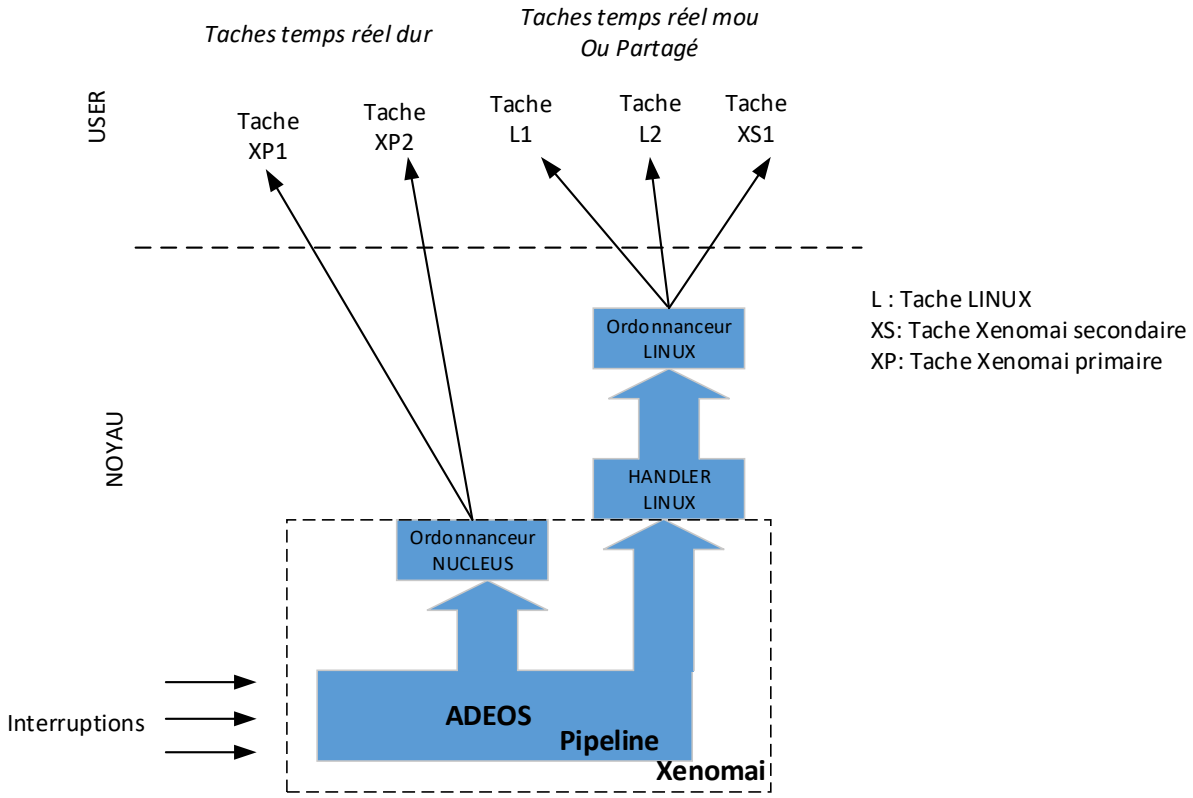
- Programmation mode kernel –
- Pas de protection de l'intégrité du système
- API limitée et simple comparée à des systèmes propriétaires,
- Debugparfois complexe.

5.4) Xenomai

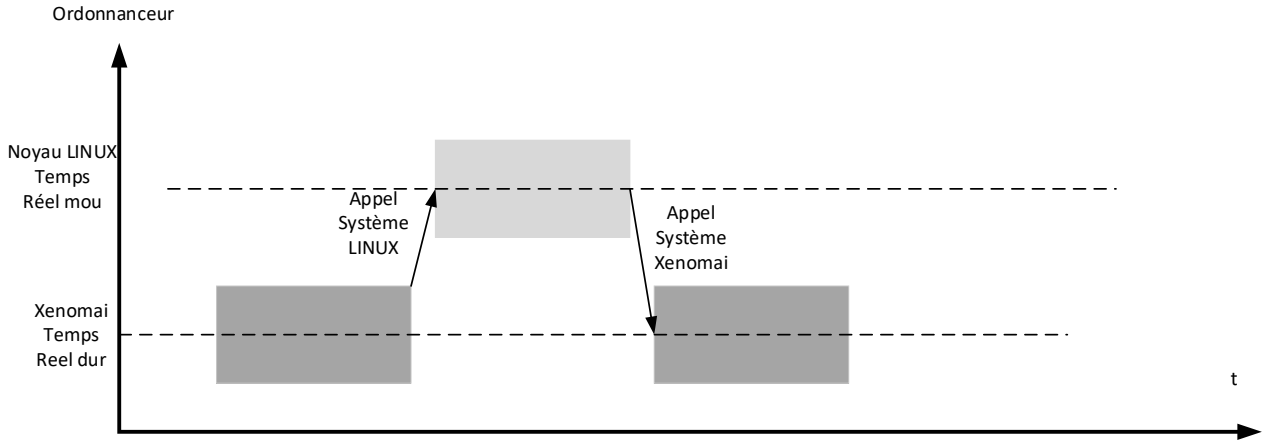
Le composant Xenomai 2, dérivé du projet RTAI permet de développer des programmes temps réel dur dans l'espace utilisateur (et non dans l'espace noyau), ce qui est un avantage notoire. Xenomai est un système d'exploitation temps réel qui prend la main sur LINUX en considérant LINUX comme une simple tache.
La préemption du noyau est écartée 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 gérées par un nanokernel de Xenomai appelé ADEOS. Celui-ci va intercepter 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.



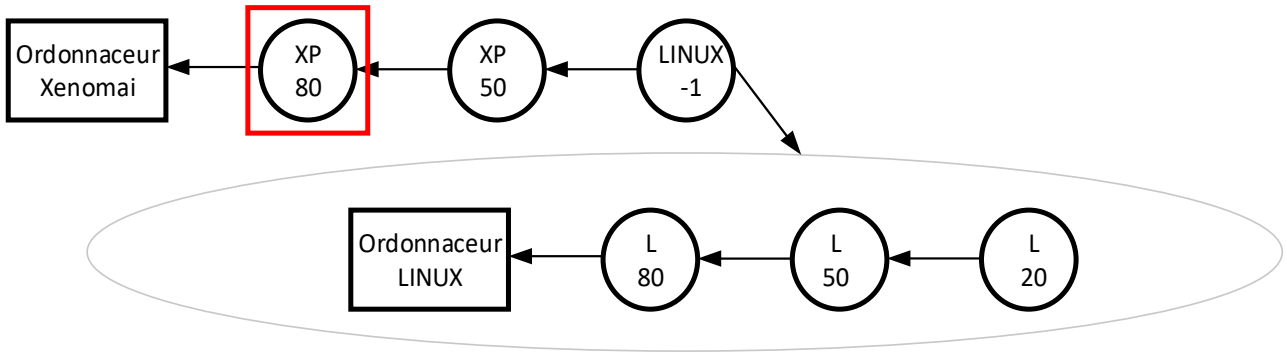
Une tâche en mode secondaire est du temps réel mou et son temps d'exécution est borné (Interrupt Shield). Pour les tâches à contrainte temps réel dur, il faut absolument rester en mode primaire

5.2) L'ordonnancement

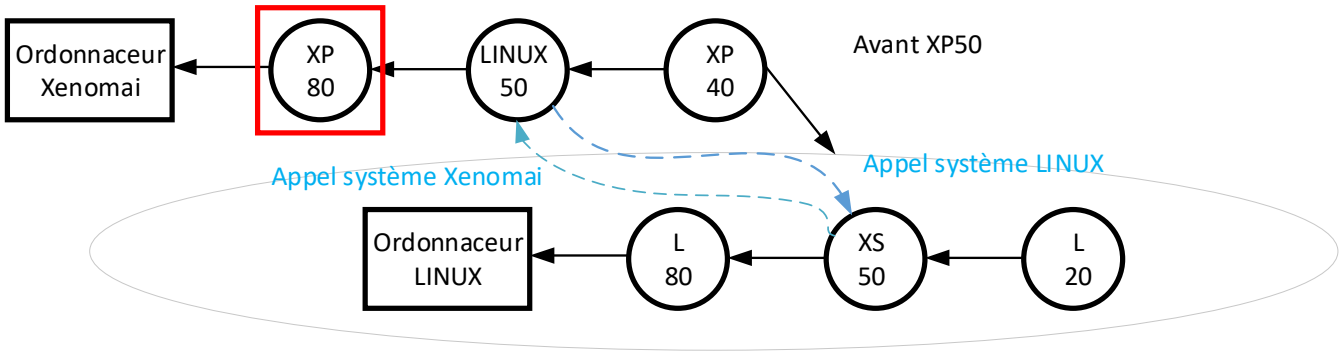
Le noyau Linux est une tâche de fond dont la priorité **évolue dynamiquement** en fonction de la tâche Xenomai de plus haute priorité éligible en mode secondaire. Dans ce cas précis ***une tâche Linux peut préempter une tâche Xenomai.***

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

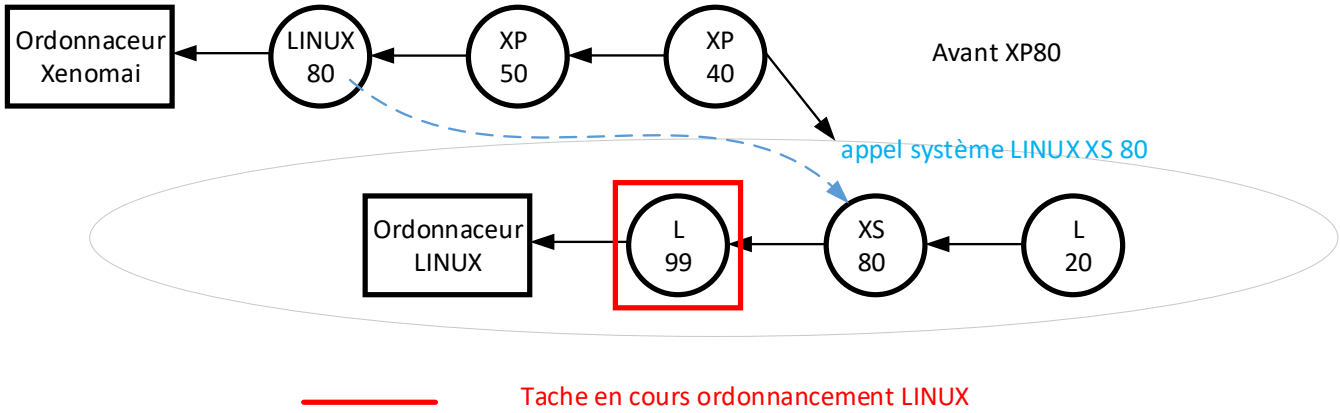
• -1 s'il n'y a aucune tâche en secondary mode (idle Xenomai) LINUX reste une tache de fond la moins prioritaire ;



• 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.