



# Les systèmes d'exploitation temps réel

# TABLE DES MATIERES

**Exemple d'un système d'exploitation temps réel**

1) **Le radar de veille tridimensionnel** **Page N°3**

**Le système d'exploitation temps réel 0S9**

1) **Notion de temps réel** **Page N°9**
2) **Notion de processus et de multitâches** **Page N°13**
3) **Gestion des processus** **Page N°20**
4) **Les techniques d'ordonnancement des processus** **Page N°27**
5) **L'ordonnancement des taches sous OS9** **Page N°31**
6) **communication et synchronisation entre processus** **Page N°34**

**OS9 TDN°1** **Page N°40**

**OS9 TDN°2** **Page N°48**

# Le radar de veille tridimensionnel

# 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***.

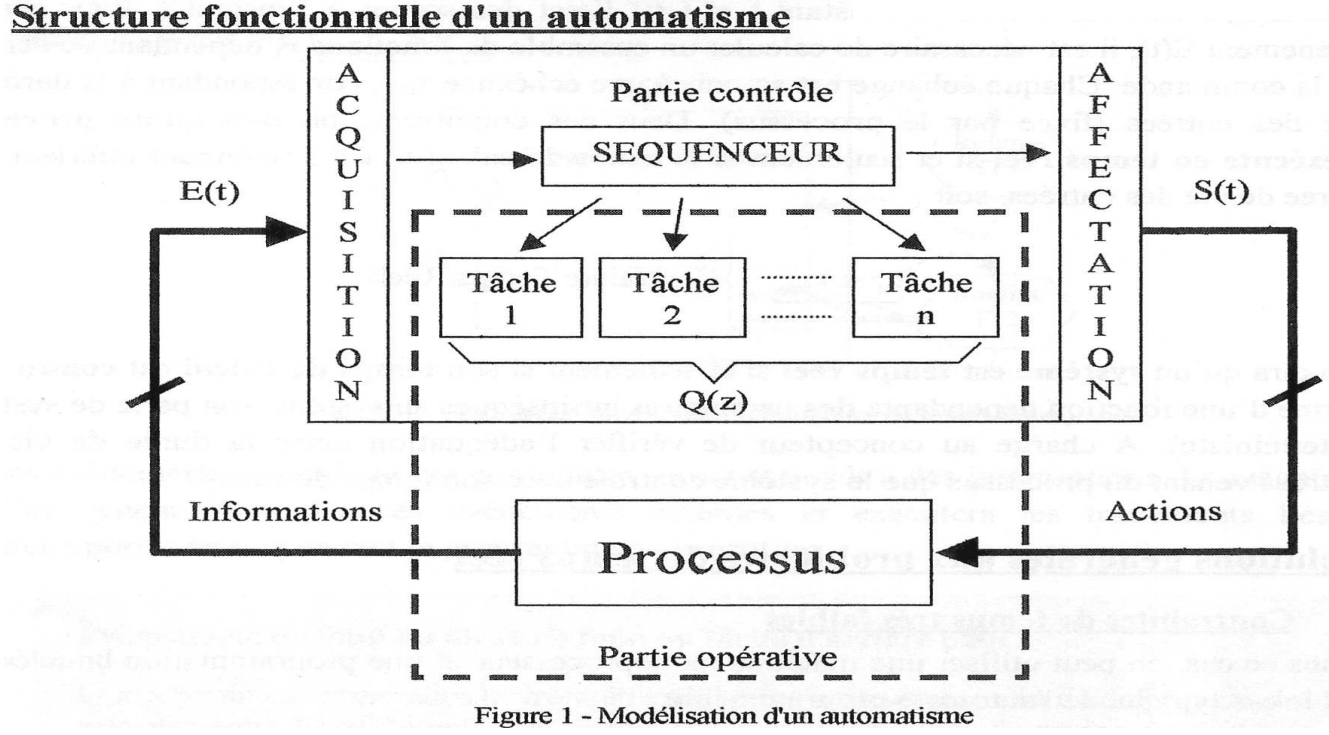


Figure 1 - Modélisation d'un automatisme

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, ... .

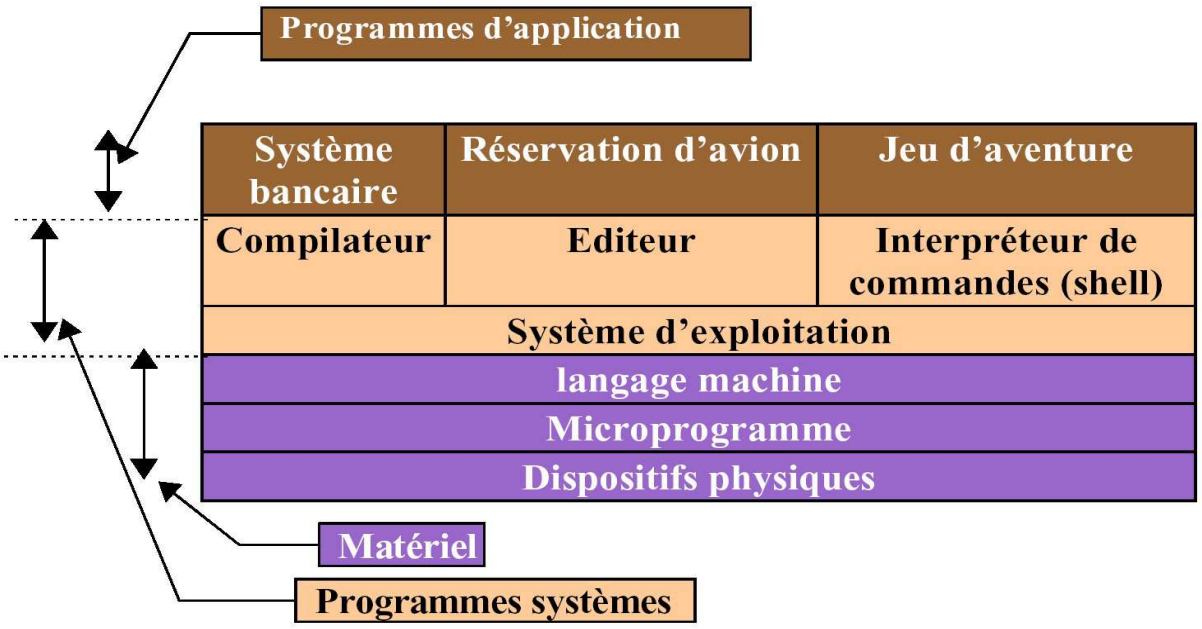
## 2) Tâches matérielles, tâches logicielles

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

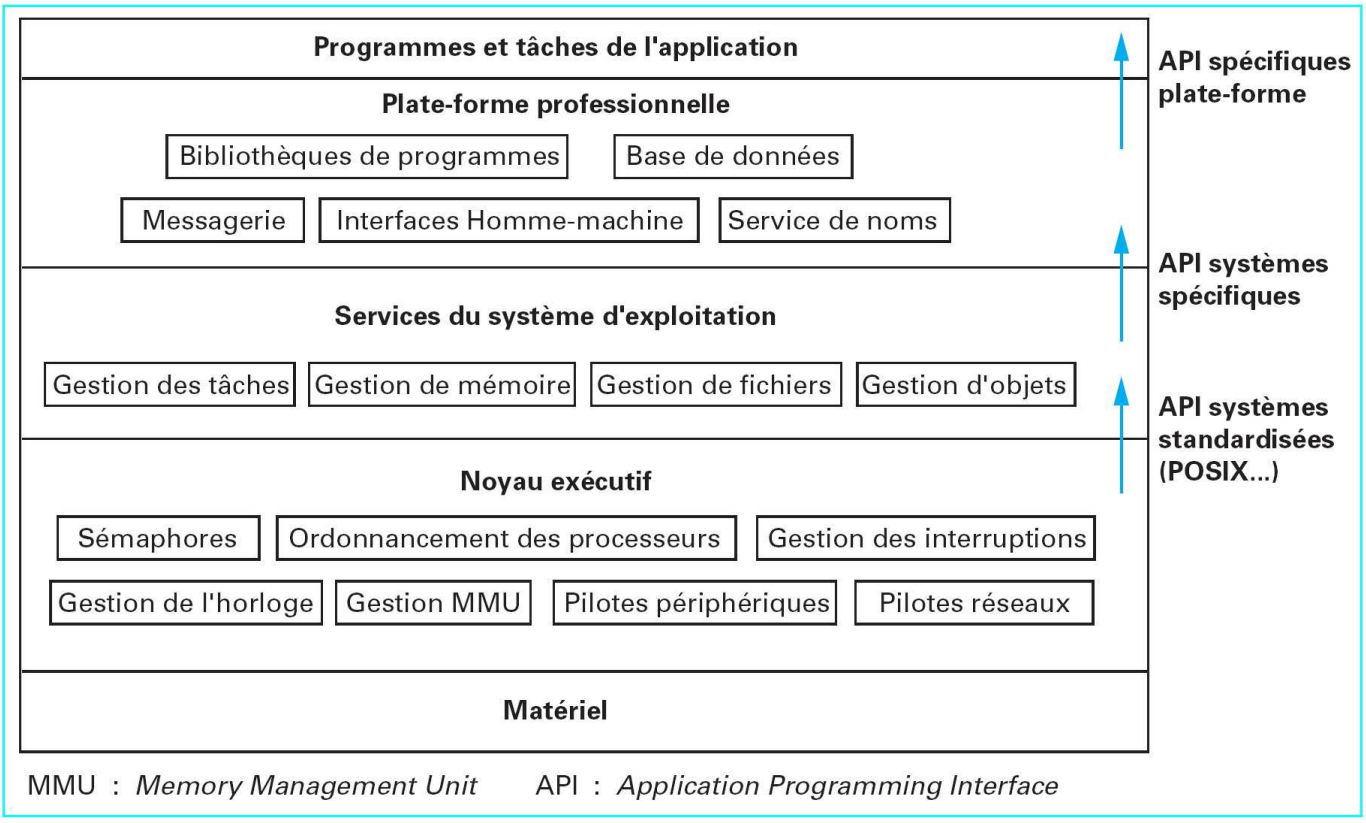
La complexité évidente du matériel implique la réalisation d'une **machine virtuelle** qui gère le matériel : **c'est le système d'exploitation.**



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 :

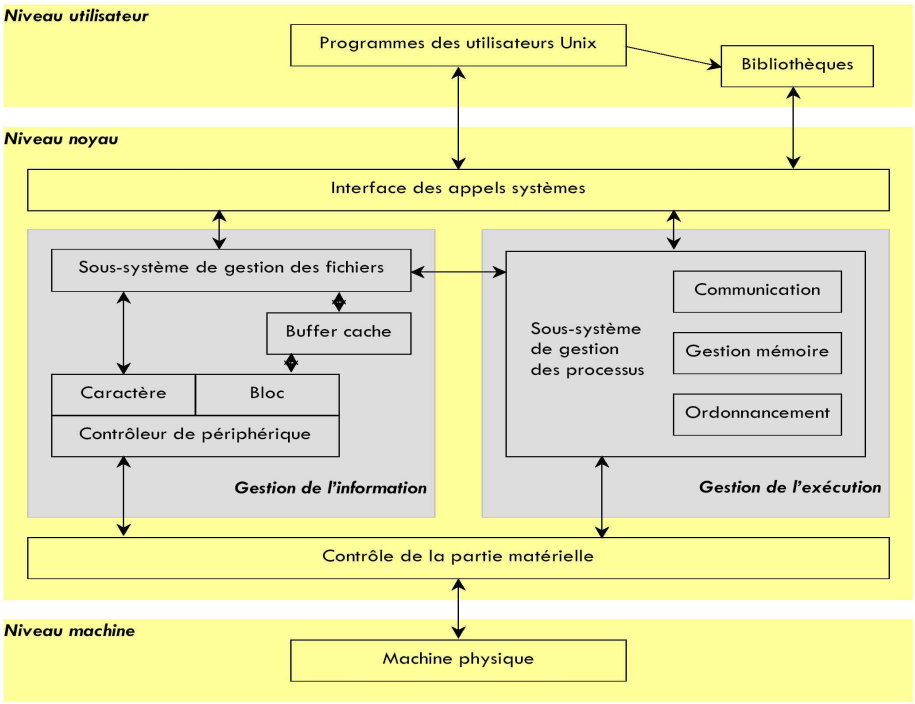



MMU : *Memory Management Unit* API : *Application Programming Interface*

## 3.2) La structure d'un noyau (type UNIX)

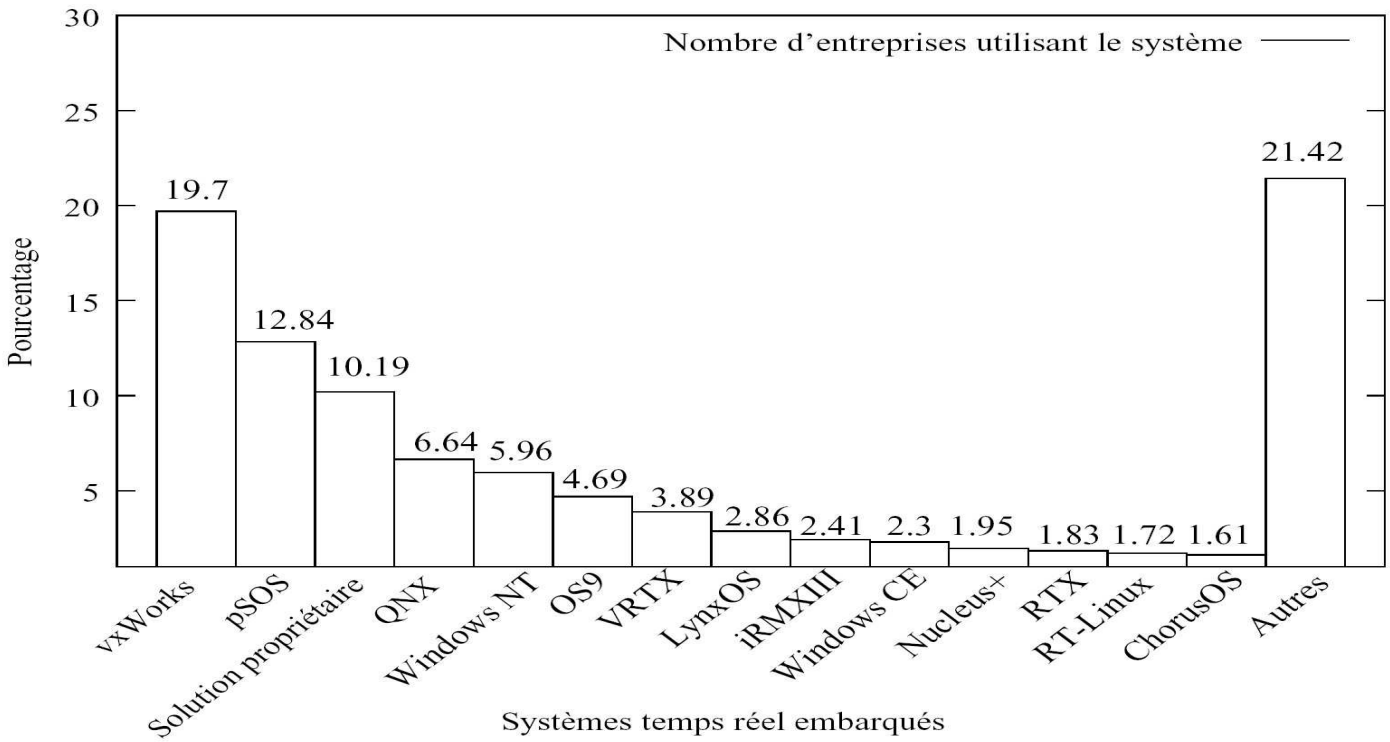
Il est important de différencier **programme** (fichier exécutable) et **processus** (programme en cours d'exécution). Un processus a besoin non seulement d'un programme, mais également des ressources de la machine physique :
- Le processeur sur lequel s'exécute le programme,
- La mémoire dans laquelle 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.

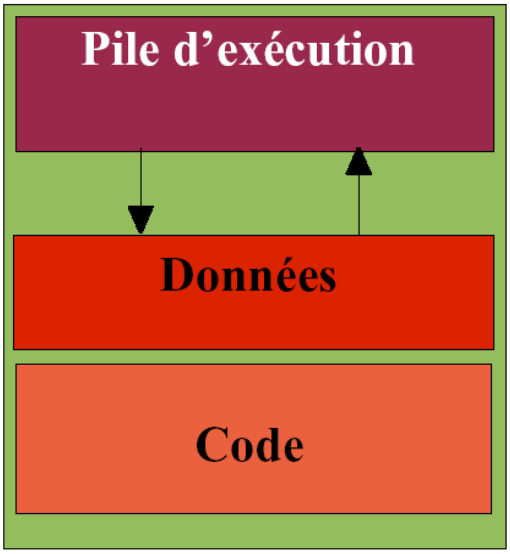
**Etat du marché des RTOS**



## 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.

```
$ procs
Id PId Grp.Usr Prior MemSiz Sig S CPU Time Age Module & I/O
2 1 0.0 128 0.25k 0 w 0.01 ??? sysgo <>>>term
3 2 0.0 128 4.75k 0 w 4.11 01:13 shell <>>>term
4 3 0.0 5 4.00k 0 a 12:42.06 00:14 xhog <>>>term
5 3 0.0 128 8.50k 0 * 0.08 00:00 procs <>>term
6 0 0.0 128 4.00k 0 s 0.02 01:12 tsmon <>>>t1
7 0 0.0 128 4.00k 0 s 0.01 01:12 tsmon <>>>t2
```

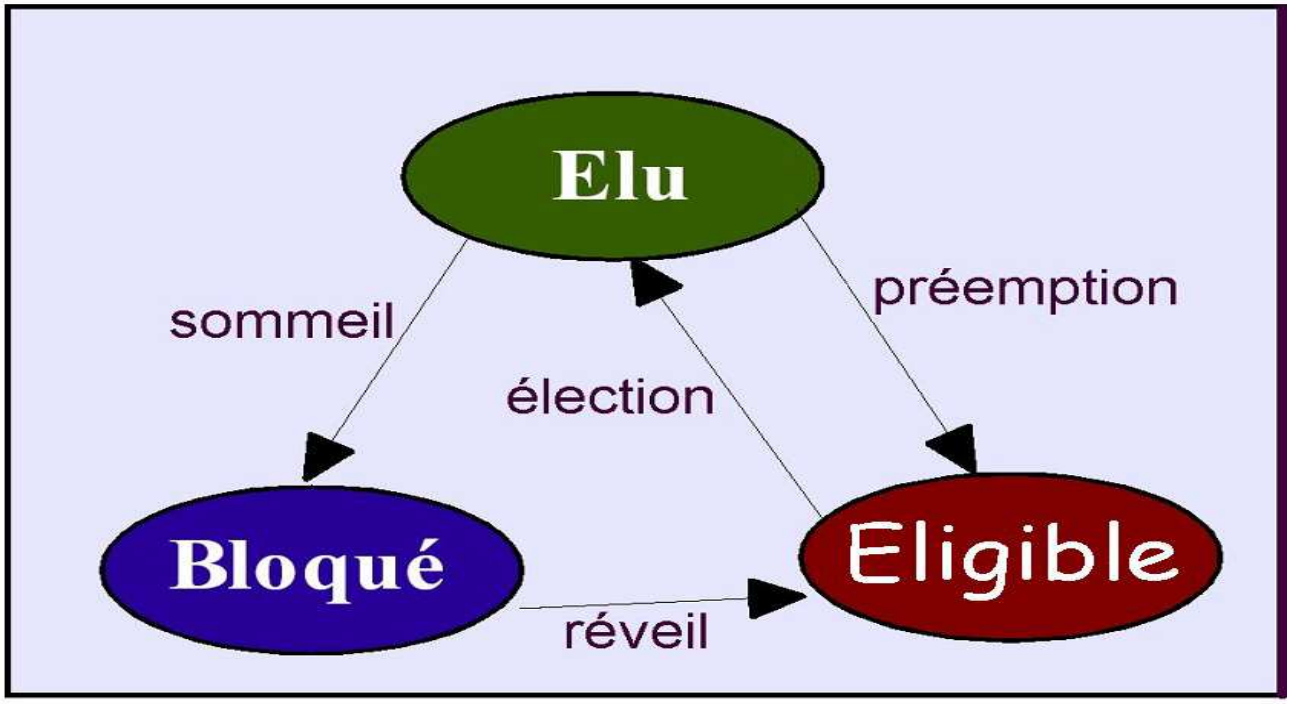
| Name | Description |
|---|---|
| Id | The process ID. |
| Pid | The parent process ID. |
| Grp.usr | The group and user number of the process owner. |
| Prior | The initial priority of the process. |
| MemSiz | The amount of memory the process is using. |
| Sig | The number of any pending signals for the process. |
| S | The process status: |
| w | Waiting |
| s | Sleeping |
| a | Active |
| * | Currently executing |

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.

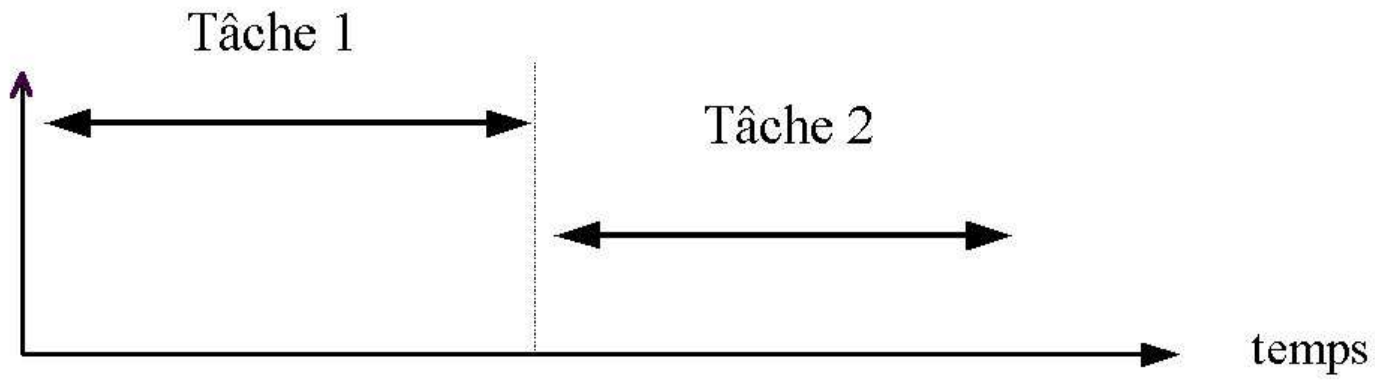
Les noyaux temps réel se divisent en deux catégories appelées *noyaux sans réquisition du processeur* (no preemptive scheduling) et ***noyaux avec réquisition du processeur*** (preemptive scheduling).
Dans un **noyau sans réquisition du processeur**, la tâche est exécutée jusqu'à ce qu'elle fasse **appel 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.

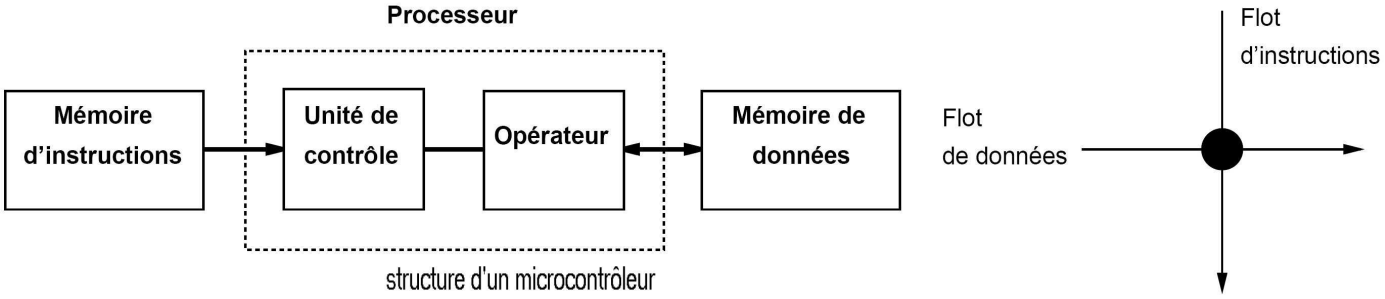
## 5) Quasi-parallélisme ou réelle simultanéité :

- 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.

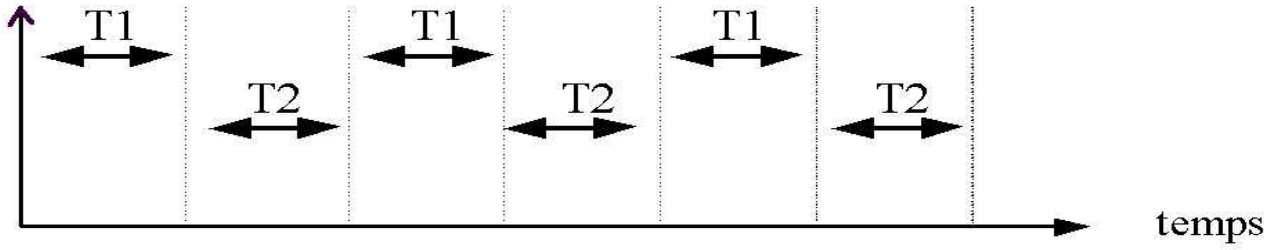



*Système monotâche/monoprocesseur*

## SISD : *Single Instruction stream Single Data stream*

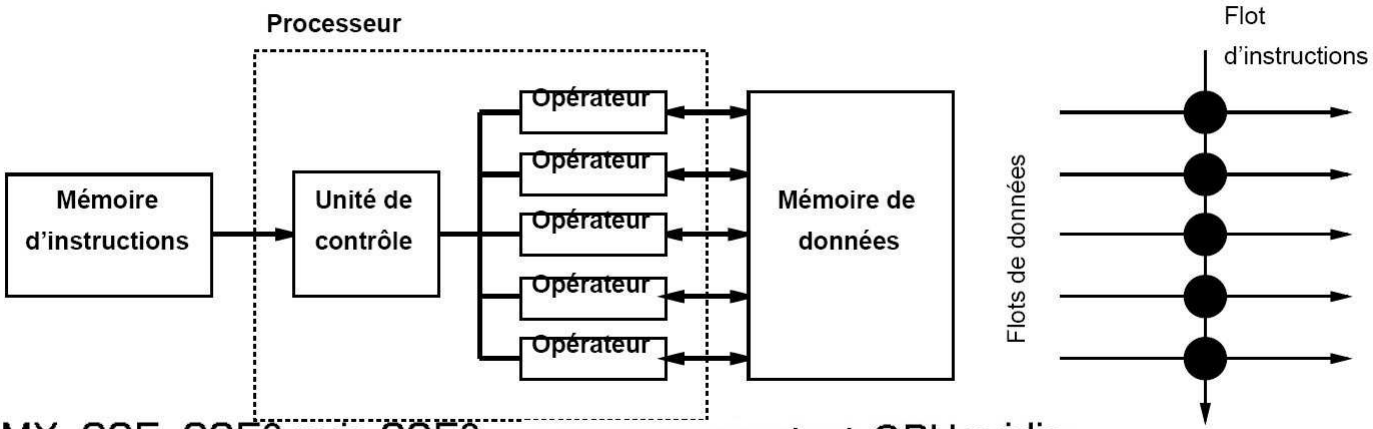



Système **multitâche/monoprocesseur:** C'est un système comportant un seul processeur, mais il est capable, à l'échelle humaine, d'exécuter plusieurs tâches simultanément. A l'échelle du processeur, les processus seront exécutés alternativement. La simultanéité est donc virtuelle, on parle de **quasi-parallélisme**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




*Système multitâche/monoprocesseur*

SIMD SINGLE INSTRUCTION MULTIPLE FLUX DE DONNEES




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

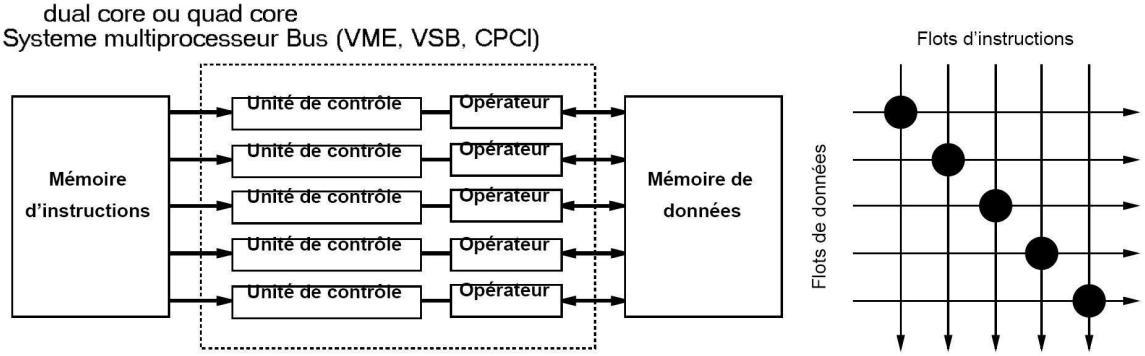
Système **multitâche/multiprocesseur:** C'est un système qui possède plusieurs processeurs. Si le nombre de ses processeurs est au moins égal au nombre de tâches à traiter, le système sera alors capable de fonctionner en réel parallélisme, ou **réelle simultanéité**.

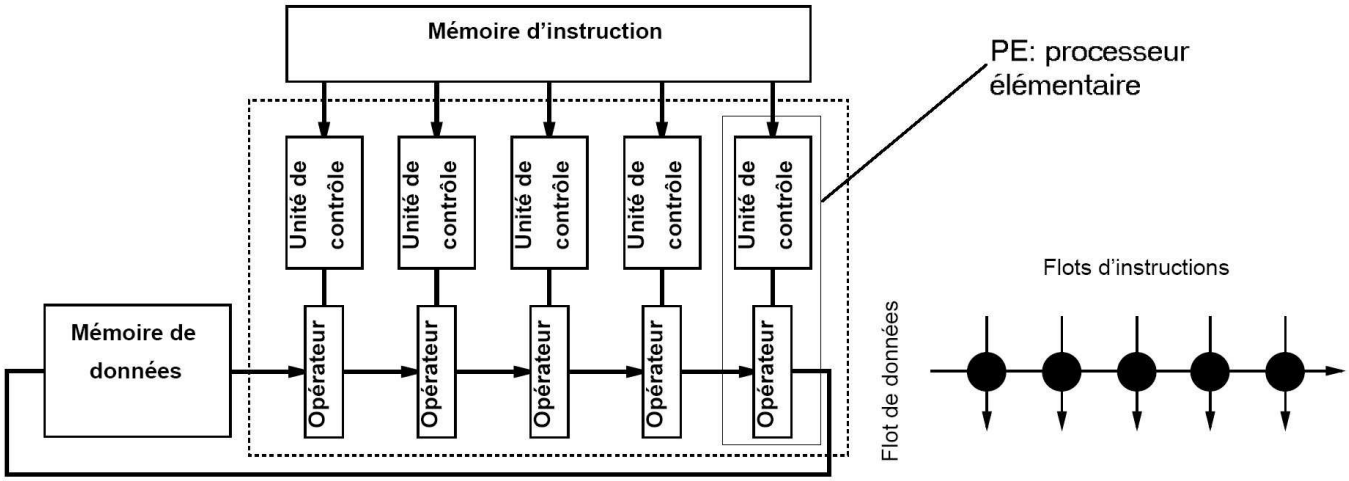



*Système multitâche/multiprocesseur*

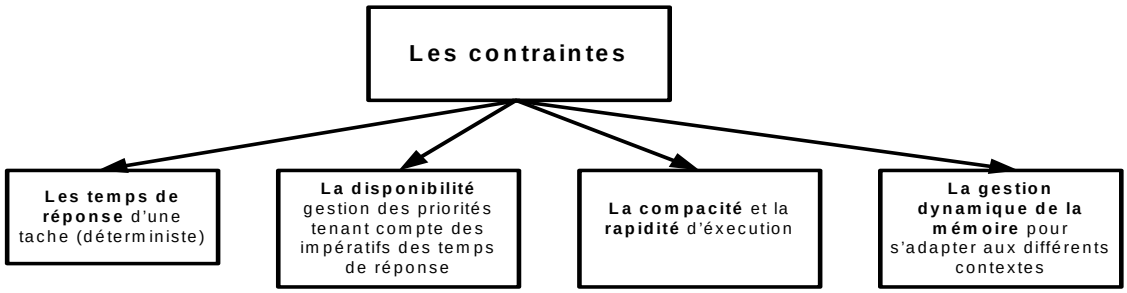
**MIMD :** *Multiple Instruction stream Multiple Data stream*

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



## 6) Les contraintes d'un système temps réel



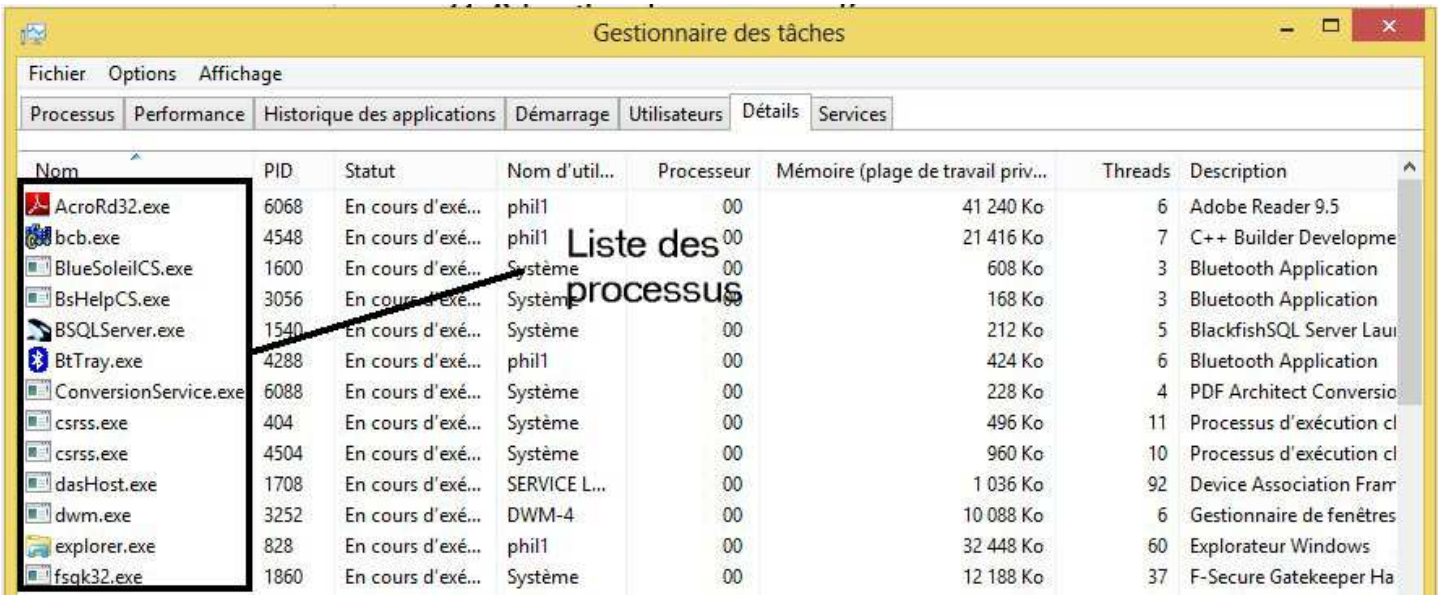
# Les Processus

## 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 :

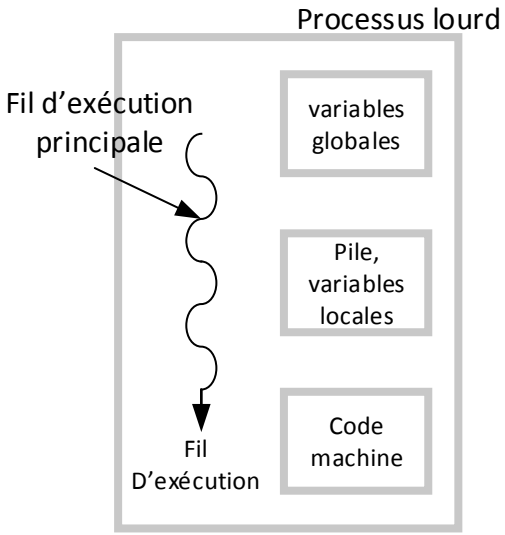
- leur création nécessite des appels systèmes coûteux en temps,
- le changement de contexte entre processus est une opération lente, en particulier pour de nombreux transferts en mémoire,
- le coût des mécanismes de protection associés au processus,
- l'interaction, la synchronisation ou la communication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

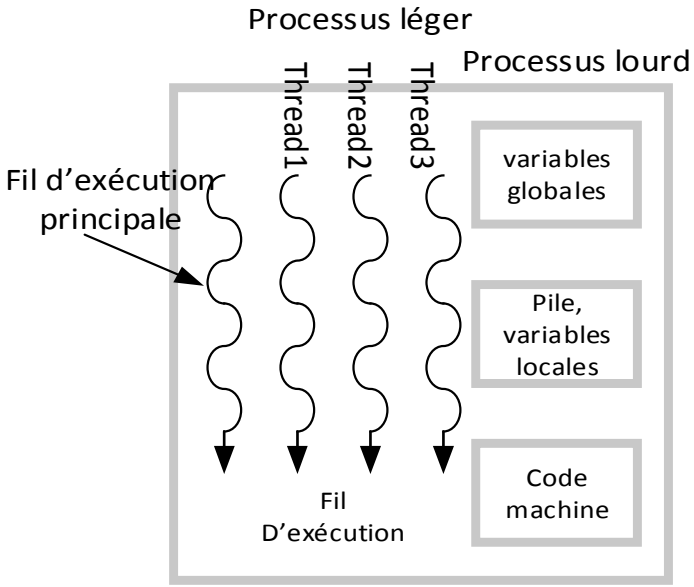



| Nom | PID | Statut | Nom d'util... | Processeur | Mémoire (plage de travail priv... | Threads | Description |
|---|---|---|---|---|---|---|---|
| AcroRd32.exe | 6068 | En cours d'exé... | phil1 | 00 | 41 240 Ko | 6 | Adobe Reader 9.5 |
| bcb.exe | 4548 | En cours d'exé... | phil1 | 00 | 21 416 Ko | 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 |

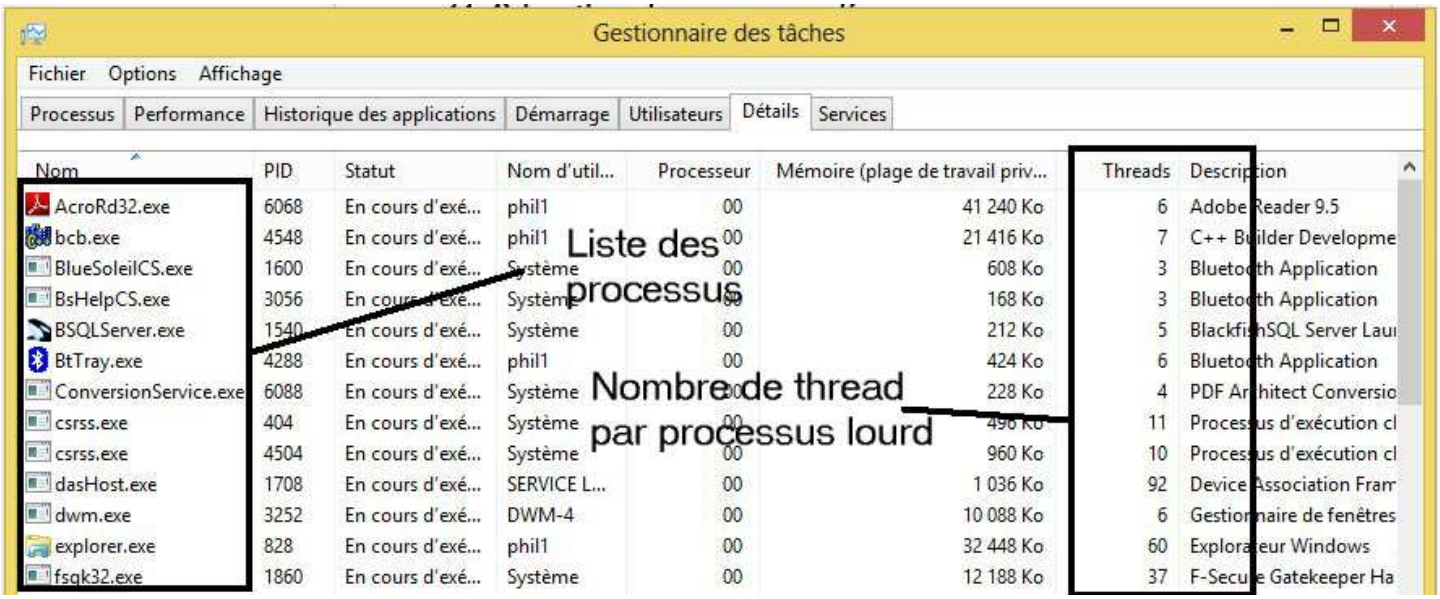
Un processus lourd classique, ne contenant qu'un seul fil d'exécution, est dit **monoprogrammé** :
L'exécution du code du processus est réalisée de manière séquentielle par un fil de contrôle (*thread of control*).
C'est un programme monolithique comprenant un corps principal la fonction main()




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 :




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 OS9 ne sait gérer que des processus lourds**

***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

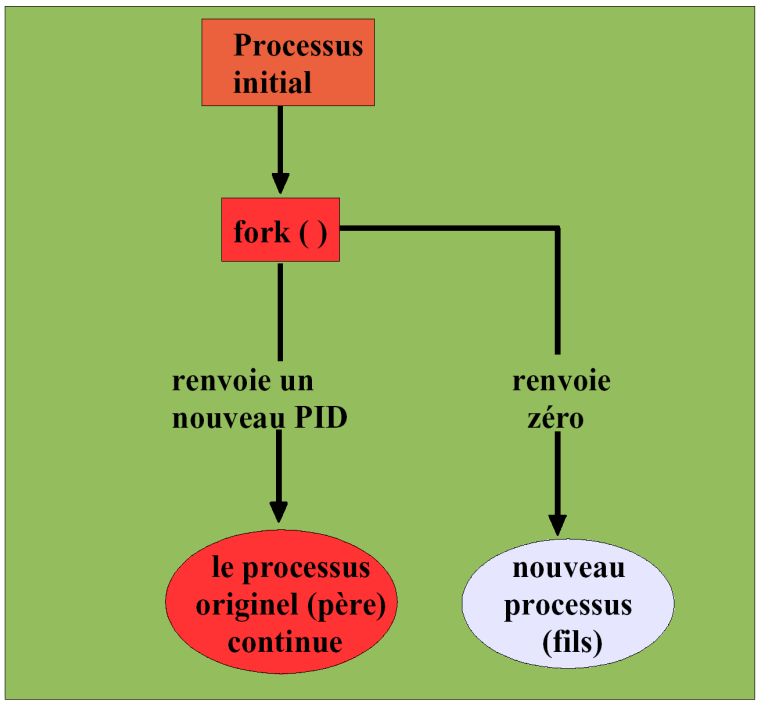
### 1.1) Création d'un processus

***int os9forkc()*** *pour OS9 (librairie sys_clib.l)*

- 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 os9forkc(), 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 évidement 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 primitive **os9exec()**. Cette primitive conserve l'identité du processus mais remplace son code exécutable (et ses données) par celui d'une nouvelle commande.



## 1.2) Association d'un module à un processus (suite)

**-Primitives os9exec:**

Permet le lancement par le processus appelant d'un nouveau programme.

***int os9exec****( os9forkc, char *modname, char **argv, char **envp,0 , priority, 3) pour OS9*

librairie : sys_clib.l

**modname** est un pointeur sur une chaine de caractère identifiant le nouveau processus. La chaine de caractère doit correspondre au nom du module préalablement chargé dans la mémoire RAM d'OS9.

**argv** est un pointeur sur un pointeur qui pointe sur la chaine de caractère correspondant à la liste des paramétres que reçoit le nouveau processus.

**Envp** est un pointeur sur un pointeur qui pointe sur la chaine de caractère correspondant à la liste des variables d'environnement.

**Priority** entier court qui correspond à la priorité du nouveau processus.

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

**exemple**

```
extern int os9forkc();
extern char **_environ;
char *argblk[] = {
    "rename", /* nom du module en mémoire que l'on va associer au nouveau processus */
    "-x",      /* liste de 4 paramètre que l'on va faire passer au nouveau processus */
    "oldname",
    "newname",
    0,
};

main ()
{
   int pid;
   if ((pid = os9exec(os9forkc,argblk[0],argblk,_environ,0,128,3)) > 0)
        wait(0); /* attendre la fin du processus fils qui est rename pour sortir */
   else printf ("can't fork %s",argblk[0]);
}
```

## 2) Autres fonctions systèmes

### Identification des processus

***#include <process.h>***

***int pid = getpid ()***

Cette primitive fournit respectivement le numéros du processus appelant et celui du processus Père

Librairie : sys_clib.l

***#include <UNIX/os9def.h >***

***int ppid = getppid()***

Cette primitive fournit respectivement le numéro du processus Père
Librairie : sys_clib.l

**Mise en sommeil d'un processus**

***#include <signal.h>***

***Int tsleep(int n) pour OS9***
Si n est à 0 le processus dort indéfiniment
Si n=1 le processus dort durant la période time slice ou tick
Si n>1 et le MSB de n=0 la valeur n donne le nombre de tick pendant lequel le processus dort
Si n>1 et le MSB de n=1 le processus dort pendant la période (n/256) secondes
Le processus retourne le nombre de tick ou de temps qu'il lui reste à dormir

Librairie : sys_clib.l

Si un signal réveille un processus, la fonction du processus qui traite ce signal s'execute, l'execution du processus continue par la suite après l'instruction tsleep

**Terminaison d'un processus**

Un processus se termine lorsqu'il n'a plus d'instructions ou lorsqu'il exécute la fonction :
***#include <stdlib.h>***
***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.

Librairie clib.l

**Attendre la fin d'un processus fils**

***#include <process.h>***

***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 : sys_clib.l

**Prépositionnement d'une alarme : alm_set()**

Déclaration
#include alarm.h
***int alm_set( int sigcode, int time) ;***

Cette fonction envoie au processus père un signal du type sigcode tous les X temps (ou X est =time).

1. Si time est >1 et le MSB de time est à 0, la valeur time donne la période (time.ticks) ou une alarme est généré (envoie du signal sigcode périodiquement).

Si time est >1 et le MSB de time est =1, la valeur time donne la période (time/256) secondes ou une alarme est générée (envoie du signal sigcode périodiquement).
Le processus retourne le nombre de ticks ou de temps qu'il lui reste à dormir. Retourne –1 si une erreur a lieu.

Librairie : sys_clib.l

# L'ordonnancement des tâches

## 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

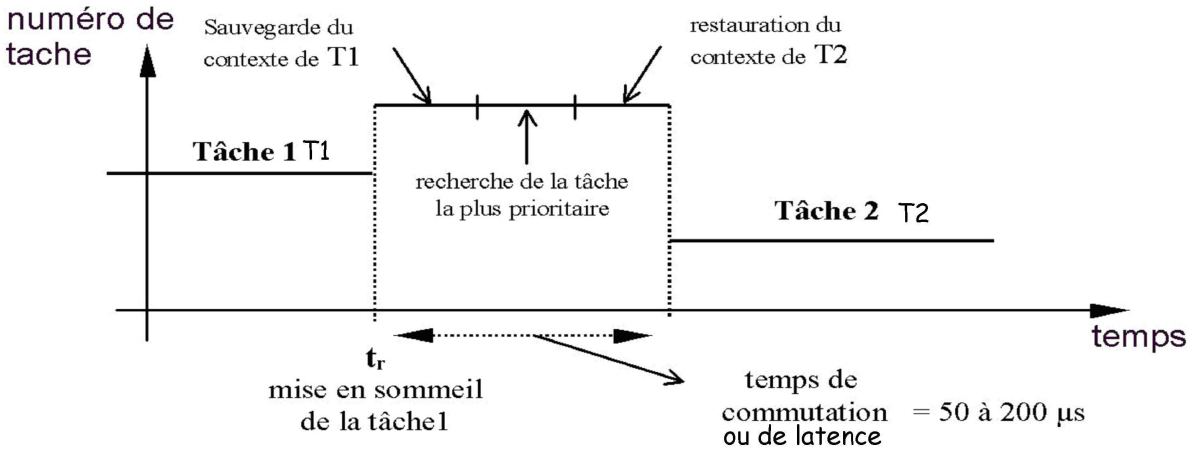
2 - Passage de l'état éligible à l'état élu

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

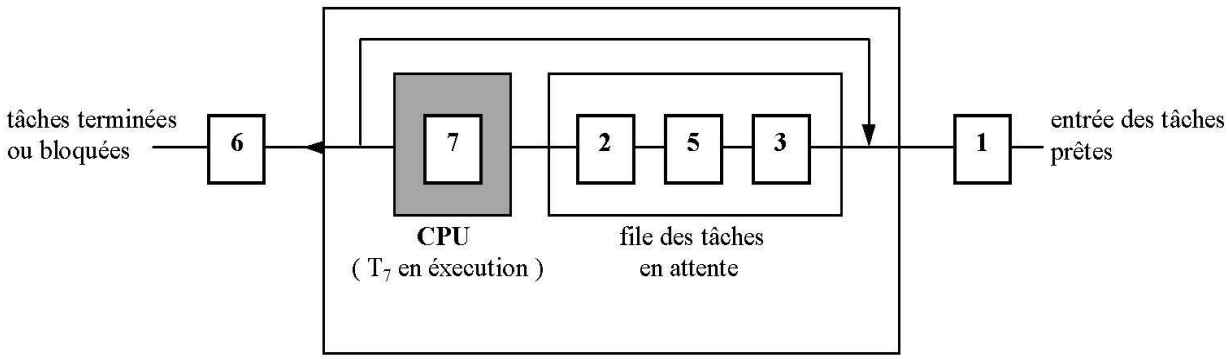
- Ordonnancement circulaire
- Ordonnancement par priorité
- Ordonnancement par priorité avec file multiple

### 3.1/ Ordonnancement circulaire

Son algorithme est le plus ancien, le plus simple et le plus fiable. On l'appel 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

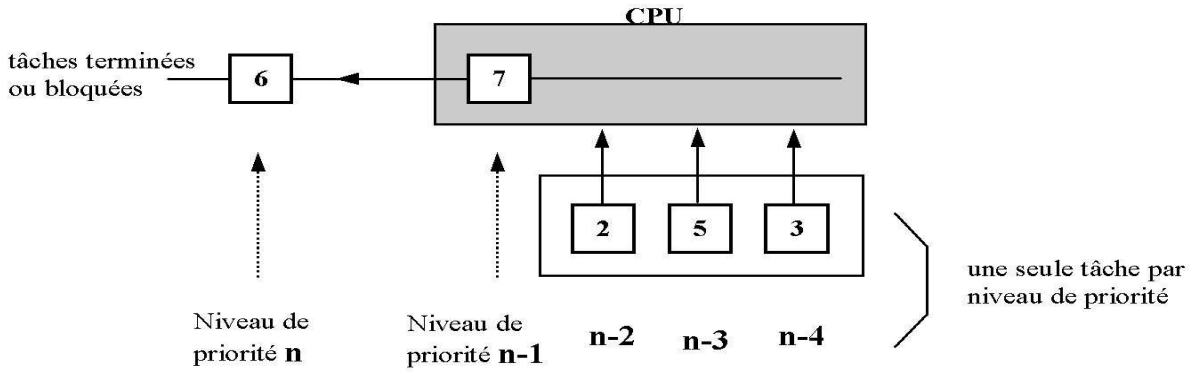
L'ordonnancement par file circulaire suppose que les tâches ont la même priorité. Ceci est rarement le cas avec un processus industriel.

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.

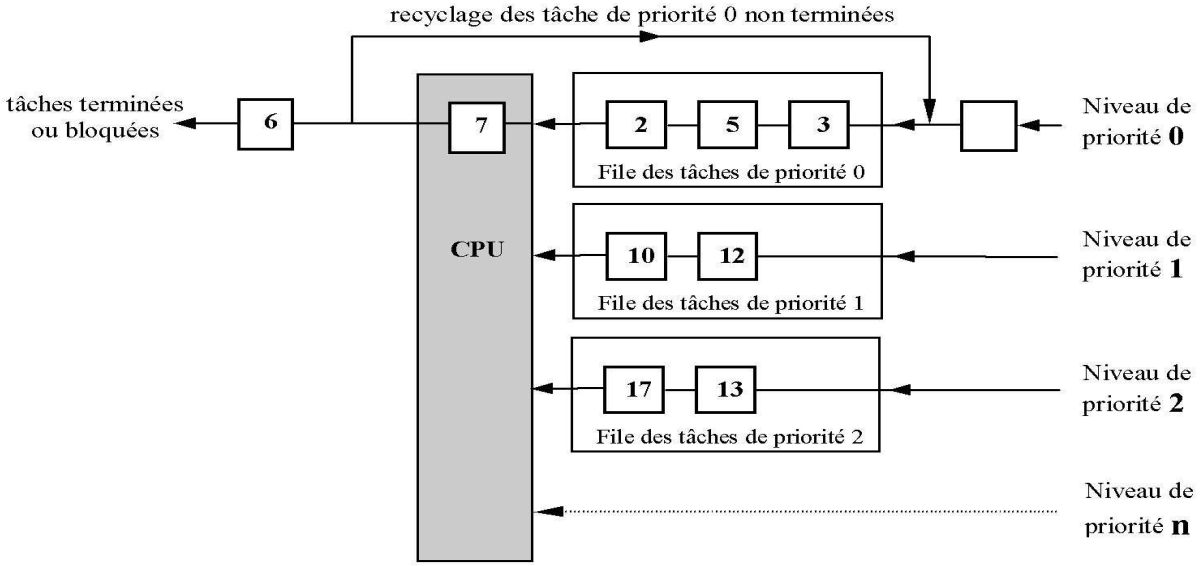
Il n'y a pas comme avec l'ordonnancement circulaire, plusieurs tâches en exécution apparente.




## 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 des tâches occupe donc 20% du temps du processeur ( overhead important ).

# L'ordonnancement des taches
# Sous OS9

## 1) Présentation et stratégie d'ordonnancement

Le système d'exploitation OS9 gère cela grâce à un ordonnanceur. Cet ordonnanceur (*scheduler*) est un programme exécutable chargé de répartir le temps processeur entre les différentes tâches concurrentes (qui souhaite s'exécuter). Le *scheduler* d'OS9 gère le temps de façon autoritaire (il est le chef) et utilise plusieurs méthodes pour le faire :

L'ordonnancement par priorité

L'ordonnancement préemptif

L'ordonnancement *Round Robin* (chacun son tour)

Pour savoir quelle méthode appliquer, OS9 définit deux types de tâches :

Les taches de faible priorité

Les taches de haute priorité

La priorité d'une tâche fixe le niveau d'urgence de son exécution (*short*).
Le *scheduler* est exécuté de façon cyclique. Le temps qui sépare deux exécutions est une tranche de temps (*time slice*). Le *time slice* est exprimé en *ticks* (unité de temps égale à 1/256 secondes ≈ 4 ms).

## 2) Les différentes techniques d'ordonnancement sous OS9

### 2.1) Ordonnancement par priorité

Dans le cas des tâches de faible priorité, le *scheduler* utilise l'ordonnancement par priorité qui repose sur un algorithme qui utilise une file des tâches concurrentes organisée par ordre de priorité croissante (file d'attente) :

```
DEBUT 'ordonnancement par priorité'
AGE = AGE + 1
SI priorité de la tâche en cours <= (priorité de la première tâche de la file+1)
    Remettre la tâche en cours dans la file d'attente (priorité initiale)
    Autoriser la première tâche de la file à s'exécuter (priorité initiale)
    Sinon
    Priorité des taches en fil d'attente= Priorité des taches en fil d'attente+1
FINSI
FIN 'ordonnancement par priorité'
```

Dans ce type d'ordonnancement, la priorité initiale d'une tâche fixe la périodicité avec laquelle cette tâche va être exécutée. Ainsi, si l'on applique l'algorithme précédent à une tâche P1 ayant une priorité initiale de 9 et à une tâche P2 ayant une priorité initiale de 10, on s'aperçoit que P2 est exécutée deux fois plus souvent que P1 ( ▸ indique la tâche exécutée) :

| Tâche/ *Age* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | **Σ*timeslice*** |
|---|---|---|---|---|---|---|---|---|---|---|
| P1 | 126 | 127 | >126 | 126 | 127 | >126 | 126 | 127 | >126 | **3** |
| P2 | >128 | >128 | 128 | >128 | >128 | 128 | >128 | >128 | 128 | **6** |

Cette méthode permet de garantir que chaque tâche sera exécutée même si sa priorité initiale est faible.

Pour fixer les idées, le temps de commutation est de 55 µs auquel on ajoute 1,5 µs par processus présent dans la file d'attente pour un système à base de processeur 68020 cadencé à 20MHz.

Ce principe de file de tâches pose pourtant quelques problèmes.
Tout d'abord, ce mécanisme est antagoniste au temps réel, en effet, le temps de réponse à un événement dépend de la place qu'occupe dans la file d'attente le processus qui l'a déclenché. Ce qui amène à se poser la question suivante : Comment gérer les événements asynchrones, tels que les interruptions ?

## 2.2) Ordonnancement *Round Robin*

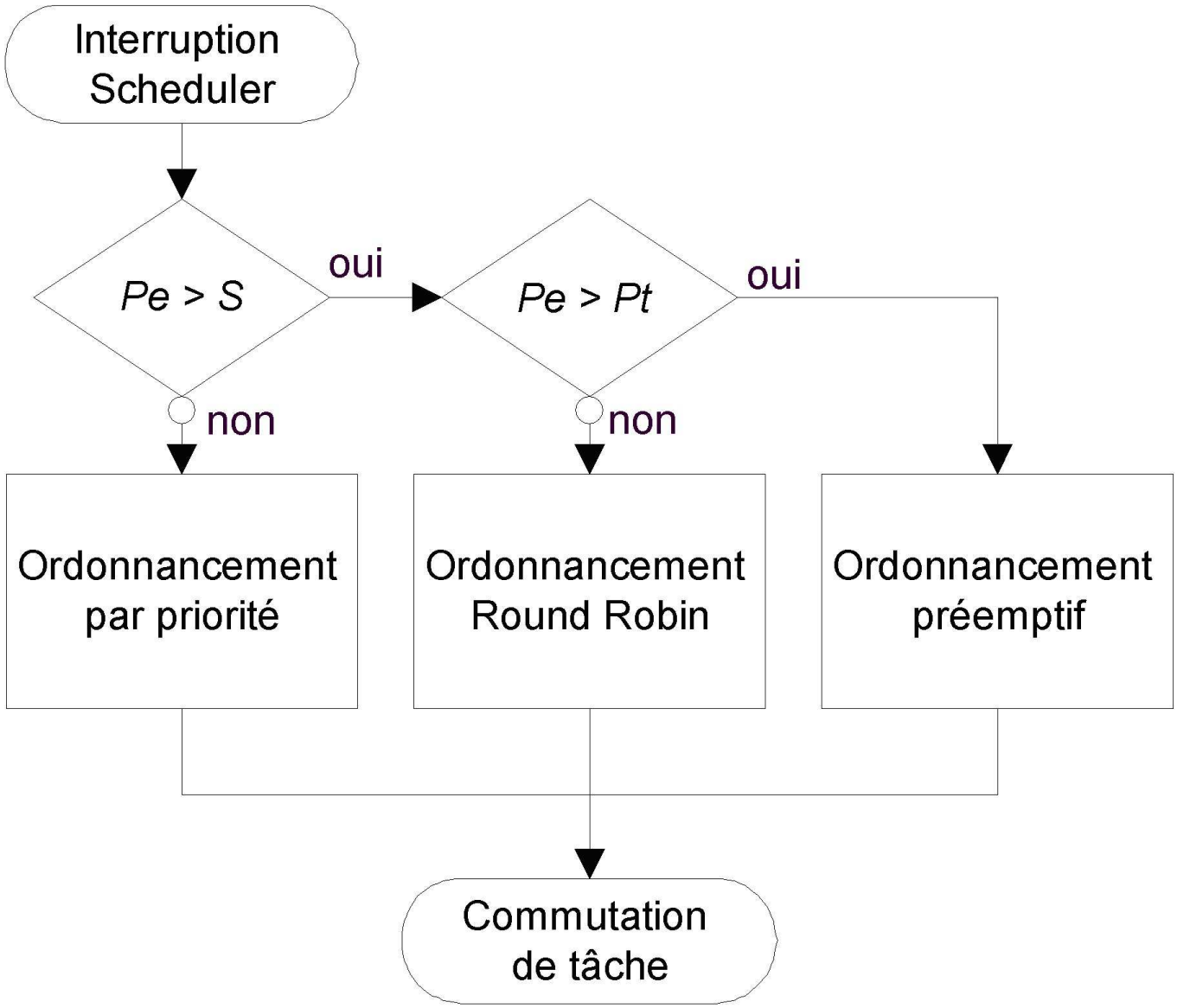
L'ordonnancement Round Robin est utilisé pour partager le temps entre tâches de même priorité. Dans ce type d'ordonnancement, chaque tâche se voit attribuer tour à tour une tranche de temps.
Il permet de garantir une répartition équitable du temps entre tâches de même priorité.

## 2.3) Ordonnancement préemptif

Les deux méthodes précédentes ne peuvent pas être appliquées lorsque la tâche en cours revêt un caractère critique. En effet, certaines tâches ne peuvent pas être interrompues. C'est le cas des tâches systèmes (comme le *scheduler* par exemple) et des tâches utilisateurs (visible grâce à la commande *procs*) dont la priorité dépasse un certain seuil appelé seuil de préemption (D_MaxAge).
Cette méthode permet de garantir qu'une tâche critique ne pourra pas être interrompue par une tâche de faible priorité.
En fait, OS9 effectue cette opération en empêchant que la priorité des tâches « normales » dépasse le seuil de préemption. Ainsi, toute tâche qui demande à s'exécuter avec une priorité initiale supérieure au seuil de préemption ne sera pas interrompue.
Si plusieurs tâches ont une priorité qui dépasse le seuil de préemption, OS9 effectue alors un ordonnancement Round Robin entre ces tâches (et uniquement entres celles-ci).

## 3) Gestion de l'ordonnancement sous OS9

Pour gérer les événements asynchrones prioritaires, les exceptions, OS9 utilise le mécanisme de préemption. C'est à dire que le système définit un seuil de priorité au-dessus duquel la tâche devient préemptive. En désignant *Pe*, comme la priorité de l'événement, *S* le seuil de préemption et *Pt* la priorité du processus en cours, on a l'algorithme suivant :

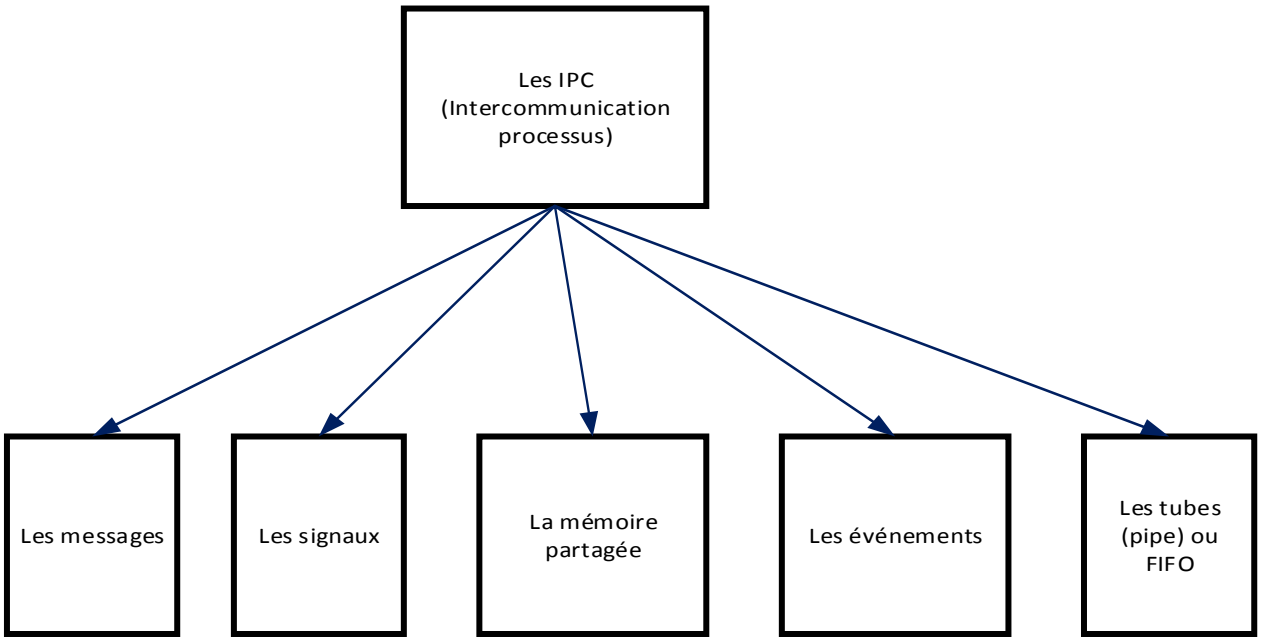



L'ordonnancement Round Robin permet de répartir le temps processeur entre les processus préemptifs. Chacun de ces processus se voit allouer le même temps à tour de rôle.
Pour fixer les idées, le temps de préemption est de 11 µs pour un système à base de processeur 68020 cadencé à 20 MHz.

# 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.

## Les signaux

Un signal est une petite quantité d'information (de la taille d'un int) 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.

1. 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 )
```

qui 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 255 sont réservés par le système. L'utilisateur peut donc utiliser tout les autres entre 256 et $2^{16}$.
- 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
Librairie : sys_clib.l

## Capture d'un signal

Un processus A doit informer le noyau qu'il est sensible aux différents signaux émis. Pour cela il utilise la fonction suivante

```
#include <signal.h>
int intercept(void(*icpthand)(int)) ;
```

ou icpthand est un pointeur sur une fonction qui renvoit un entier.
Si un signal est émis le noyau execute la fonction pointée par icpthand

**Exemple :**

```
Main()
{…….
intercept(fct1) ; /* fct1 adresse de la fonction */
}
int fct1(int sig) /* sig numéro du signal envoyé */
{…….
}
```

Library sys_clib.l

**Masquage des signaux**

```
#include <signal.h>
int sigmask(int level)
```

- Si un signal sensible au processus est envoyé et que level=0 l'ordonnanceur donne la main à ce processus en executant la fonction pointée par la routine intercept.
- si le level est différent de 0 le signal est placé dans une file d'attente de signaux attendant d'être traités. La file d'attente des signaux est traitée lorsque level repasse à 0.

Dans le cas ou le processus est endormi par la fonction sleep et que ce processus vient d'être réveillé par un signal on execute :
- la fonction pointée par la routine intercept
- puis l'instruction qui suit dans le processus la fonction sleep().

Library sys_clib.l

## 3) synchronisation entre processus

### 3.1) 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. )

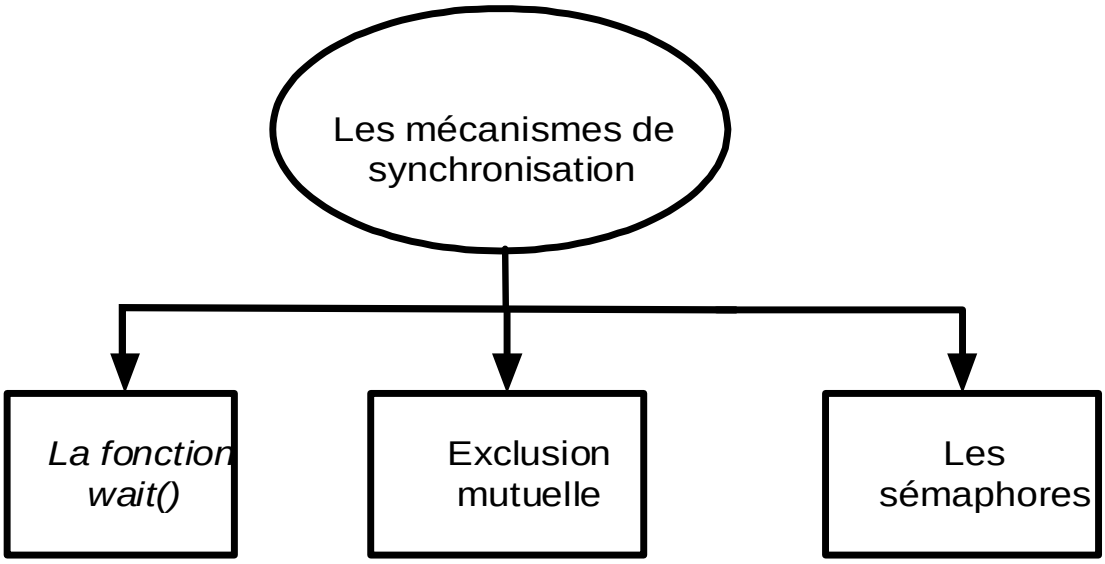
Parmi ces deux types de ressources, il faut distinguer les ressources :

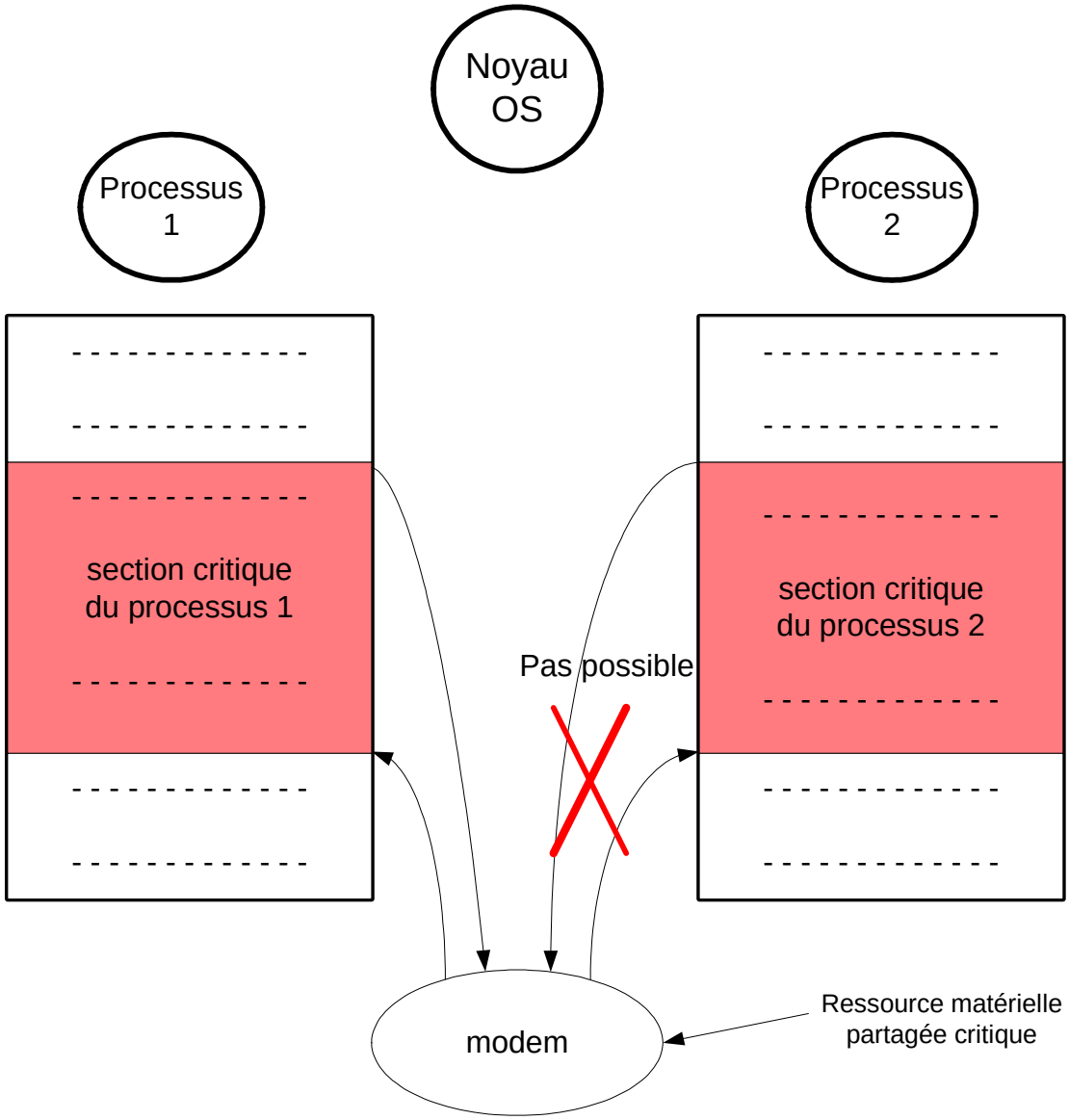
- **locales** ,c'est à dire utilisées par une seule tâche.
- **communes**, c'est à dire **partageables** par plusieurs tâches.

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***.






## 3.2) 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.

# OS9 TD N°1: le ballon hydrophore

## 1) Objectifs

Etablir un programme faisant appels aux fonctions système d'OS9 :

- 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 sous OS9
- cours sur le temps réel avec l'ordonnancement, sur le système d'exploitation temps réel OS9 et sur les appels systèmes sous OS9

## 3) Présentation

La gestion du système ballon hydrophore est réalisée logiciellement grâce à 4 processus. Ces processus vont s'éxécuter **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 telnet pour accéder au shell du système OS9 (interpréteur de commande)

**4.1)** Créer sous l'éditeur Hawk 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 charger le module dans la mémoire de hawk.
n.b n'oubliez pas d'associer les librairies pour le linker pour cela cliquez droit sur le nom du components →properties →link →o-code libraries choisir les librairies adécquates liés à chaque fonction utilisée dans le code source :
C:\Mwos\OS9\68020\LIB\unix.l;
C:\Mwos\OS9\68020\LIB\sys_clib.l;
C:\Mwos\OS9\68020\LIB\sclib.l;
C:\Mwos\OS9\68020\LIB\clib.l
Pour avoir un affichage correct on rajoutera les lignes suivantes ;
setbuf(stdin,NULL);
setbuf(stdout,NULL);
avec le fichier d'en tête suivant :

#include <stdio.h>

**4.2)** Lancez le programme grâce à l'utilitaire telnet connecté à la cible OS9

vérifiez que le processus est actif en mémoire grâce à la commande procs et donnez la commande qui permet d'éliminer ce processus
le programme doit être lancée en mode multitâche

**4.3)** Modifiez ce programme en éliminant la boucle sans fin compiler, charger , executer.
Ce programme est il présent dans la liste des processus actifs, commentez ce résultat.
La fonction exit (0) permet de détruire le processus en cours bien plus proprement.

**4.4)** Créer un premier programme qui permet d'afficher bonjour je suis le processus N°1
mon PID est mon PPID est . Ce programme sera chargé dans la mémoire de la cible, il sera nommé process1 (le processus sera actif indéfiniment)
créer un deuxième programme qui permet d'afficher bonjour je suis le processus N°2
mon PID est mon PPID est .
ce programme sera chargé dans la mémoire de la cible, il sera nommé process2
le processus ne sera pas actif indéfiniment.
Charger les 2 modules de ces 2 programmes dans la mémoire de hawk.

**4.5)** Complétez le troisième programme qui permet de lancer les 2 processus créés précédemment. On aurait pu créer les 4 processus (voir ballon hydrophore) mais cela alourdit inutilement le programme.
un processus associé au premier programme process1
un processus associé au deuxième programme process2
On utilisera la fonction os9exec() pour lancer les 2 processus
ce programme permettra aussi de tuer les 2 processus précédents
au départ le programme propose 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.

**4.6)** Pourquoi les caractères se mélangent ils lorsqu'on voit apparaître le texte sur Telnet ?

```
#include <stdio.h>
#include <process.h>
#include <errno.h>
#include <UNIX/os9def.h>
#include <modes.h>
#include <events.h>
extern int os9forkc();
extern **_environ;
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 fid,prio=0x80;
        char priorite[4]="128";

        printf("\n lancement du processus ");
        prio=atoi(_____);
        if((fid2=_____
_____)
                return fid2;
        else
                printf("impossible de lancer le processus");
        return -1;
}
```

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

        printf("\n lancement du processus ");
        prio=_____;
        if((_____
_____)
                return fid1;
        else
                printf("impossible de lancer le processus");
        return -1;
}

int tue2()
{       char nno;
        if(_____)
        return -1;
        puts(" la tache N°2 est tué");
        fid2=0;
        return 0;

}

int tue1()
{       char nno;
        if(kill(_____)
        return -1;
        puts(" _____");
        fid1=0;
        return 0;
}

void main (void)
{
char choix;
while (1)
{
printf (" je suis dans le menu l'action désirée \n");
printf (" tapez 1 pour lancer le processus N°1 \n");
printf (" tapez 2 pour lancer le processus N°2 \n");
```

```
printf (" tapez 3 pour tuer le processus N°1 \n");
printf (" tapez 4 pour tuer le processus N°2\n");
printf (" tapez 5 pour sortir \n");
scanf ("%c",&_____________________________);
printf (" le choix est %c \n", _____________________________);
switch (choix)
            {
            case '1' : if(fid1!=0)
                                    printf("tache 1 deja en cours");
                              else
                              {

      if((errno=_____________________________________________)
                                    printf("impossibilite de lancer la tache");
                                    else
                                    fid1=errno;
                              }
                              break;
             case '2' : if(_____________)
                                    printf("tache 2 deja en cours");
                              else
                              {
                                    if((_____________________________________)
                                    printf("impossibilite de lancer la tache");
                                    else
                                    fid2=errno;
                              }
                        break;

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

# Suite TD1 OS9 : Les sémaphores

## 1) Présentation

OS9 ne gère pas directement les sémaphores mais les événements. Une série de fonctions en C permettent la gestion des événements.
On peut notamment par un choix judicieux des paramètres d'entrées des fonctions obtenir le même comportement que pour les sémaphores
les fonctions qui gèrent les événements sont les suivantes :

## 2) Les différentes fonctions

#include <events.h>
**int _ev_creat (int ev_value, int wait_inc, int signal_inc, char *ev_name);**
librairie sys_clib.l
_ev_creat() créé un événement.
ev_value est la valeur initiale pour le compteur d'événement .
**wait_inc** est la valeur d'incrément d'attente. Il est additionné à la valeur du compteur d'événement quand la fonction _ev_wait() est exécuté .
**signal_inc** est une valeur d'incrément de signal. Il est additionné à la valeur du compteur d'événement quand la fonction _ev_signal() est exécuté.
ev_name est un pointeur sur le nom de l'événement.
Un numéro d'identification d'événement est créé si l'événement est créé avec succès
Si une erreur a lieu -1 et retourné .

#include <events.h>
**int _ev_link (char *ev_name);**
librairie sys_clib.l
_ev_link() crée un lien à un événement existant. Un numéro d'identification d'événement est retourné si l'événement est lié avec succès.
ev_name est un pointeur sur le nom de l'événement.
Si une erreur à lieu -1 est retourné .

#include <events.h>
**int _ev_wait (int ev_id, int ev_min, int ev_max);**
librairie sys_clib.l
_ev_wait() attends qu'un événement a lieu. La valeur du compteur d'événement est comparée à la gamme spécifiée par ev_min et ev_max.
Si la valeur du compteur d'événement n'est pas dans la gamme spécifiée, le processus se met en attente jusqu'à ce qu'un autre process se trouve dans cette gamme.
Si la valeur du compteur d'événement est dans la gamme spécifiée, la valeur d'incrément d'attente (**wait_inc**) est additionnée à la valeur du compteur d'événement.
La valeur du compteur d'événement est retournée par la fonction.
ev_id spécifie le numéro d'identification de l'événement.
ev_min spécifie la valeur de la gamme minimale pour le compteur d'événement .
ev_max spécifie la valeur de la gamme maximale pour le compteur d'événement .
Si une erreur a lieu -1 et retourné .

#include <events.h>
**int _ev_signal ( int ev_id, int allflag);**
librairie sys_clib.l
ev_signal() indique qu'un événement a eu lieu
La variable du compteur d'événement est additionnée avec la valeur du signal d'incrément **signal_inc**
ev_id spécifie le numéro d'identification de l'événement.
allflag spécifie le process a activé. Seulement les 16 bits de poids faibles sont utilisés
Si allflag est à zéro le premier process en attente de l'événement est activé.
Si allflag est 0x8000, tous les process en attente pour l'événement qui ont une valeur dans la gamme spécifié sont réveillés et deviennent éligible.
ev_signal() retourne 0 si une erreur à lieu -1 est retourné .

## 3) Exemple

```
char chbusy[]={"ch"};

void main (void)
{
--------------
--------------
ev_id=_ev_creat(1,-1,1,chbusy);
----------------------------
---------------
---------------
_ev_wait(ev_id,1,1);
Zone critique1
_ev_signal(ev_id,0);
}
```

## 4) Questions

4.1) Comment s'appelle le nom de l'événement ou le sémaphore ?
4.2) Pourquoi peut on assimiler la fonction _ev_wait(ev_id,1,1) comme une tentative de prise de sémaphore ?
4.3) Pourquoi peut on assimiler la fonction _ev_signal(ev_id,0) comme une libération de sémaphore
4.4) Utilisez le sémaphore précédent dans toutes les zones critiques du TDN°1, testez votre programme et conclure.

# OS9 TD N°2 : Les signaux

## 1. 1) Objectifs

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

## 1. 2) Prérequis

- lecture préalable du fascicule les outils de développement sous OS9
- cours sur le temps réel avec l'ordonnancement, sur le système d'exploitation temps réel OS9 et sur les appels systèmes sous OS9

## 3) Présentation

Pour que le système multitâches ou plutôt multiprocessus du ballon hydrophore 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

Il n'existe pas sous OS9 la commande UNIX kill permettant d'envoyer un signal vers un processus. La commande kill sous OS9 a un effet très limité qui ne permet que de tuer un processus.
Afin de trouver une commande équivalente on a créé un programme nommée sendsig permettant d'envoyer un signal vers un processus.
Cette fonction est donnée il s'agira lors du TD de compiler le programme sendsig.c et de charger son module sous OS9 (rajoutez dans les unités la fonction help.c -→ add unit pour la compilation). Une fois chargé le module la commande est la suivante :
**Sendsig** numéro du PID du processus numéro du signal.

## 5) questionnaires

utilisez le logiciel telnet pour accéder au shell du système OS9 (interpréteur de commande)

**5.1)** Compilez et chargez dans la mémoire OS9 le module sendsig

**5.2)** 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 300 écrire à l'écran alarme active.
si ce signal a comme numéro 301 écrire à l'écran alarme désactivé.

si ce signal a comme numéro 302 écrire tuer le processus.
Compilez ce programme et chargez le dans la mémoire d'OS9
Lancez ce programme en mode multitâche et vérifiez son bon fonctionnement grâce à la commande sendsig et procs. Ce programme sera nommé visualisation.

**5.3)** 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 300 vers le processus visu
- Envoyer le signal 301 vers le processus visu
- Envoyer le signal 302 vers le processus visu

Complétez le programme suivant afin qu'il corresponde au cahiers des charges précédents :
Compilez et testez le bon fonctionnement du processus hydro.

## HYDRO.C

```
#include <stdio.h>
#include <process.h>
#include <errno.h>
#include <signal.h>
#include <UNIX/os9def.h>

int tue(int num);
int lance1();
int lance2();
void intfct(int sig);
int arret();

extern int os9forkc();
extern **_environ;
int pid1=0,d;

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

```
void main (void)
{
int choix;
setbuf(stdin,NULL);
setbuf(stdout,NULL);
if((pid1=_____)
        printf("création du  processus visu \n");
    else
        printf("impossible de lancer le processus\n");
/*d=atoi(proc1[1]);*/
while(1)
{
printf (" tapez 1 pour activez l'alarme \n");
printf (" tapez 2 pour désactivez l'alarme \n");
printf (" tapez 3 pour tuer le processus visu \n");
printf (" tapez 4 pour tuer le processus pere \n");
scanf("%d",&choix);
switch (choix){
case 1: {kill(_____);break;}
case 2: {kill(_____);break;}
case 3: {kill(_____);break;}
case 4:_____;
default: printf("_____\n");}
}
}
```