PARTIE 1

SYSTEMES TEMPS REEL

Principes généraux

Maîtrise EEA
Maîtrise IUP
UJF Grenoble
SYSTEMES TEMPS REEL

PA DEGRYSE

SOMMAIRE

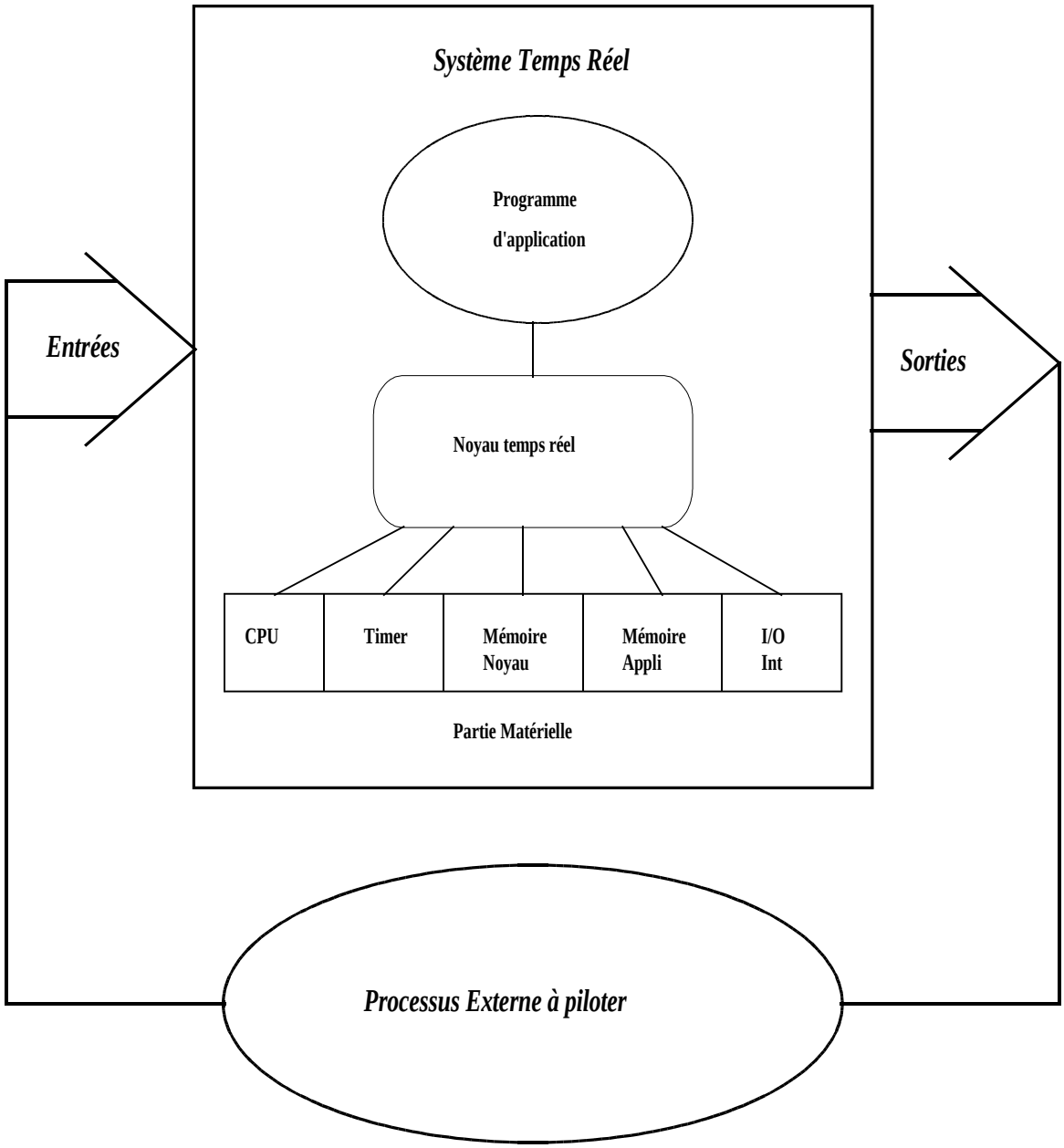
Chapitre 1 :Généralités

A) Introduction : notion de temps Réel

Le programme de ce cours est de comprendre le principe des logiciels pour des applications en temps réel et exploiter un système de programmation multitâche temps réel adapté à un processeur de traitement du signal DSP.

Un système temps réel est un système à la fois logiciel et matériel qui contrôle un environnement par acquisition de données, traitement et renvoi de résultats d'actions afin de respecter des critères de temps de réponse et de fiabilité.

Le dialogue avec l'opérateur fait partie intégrante du système.



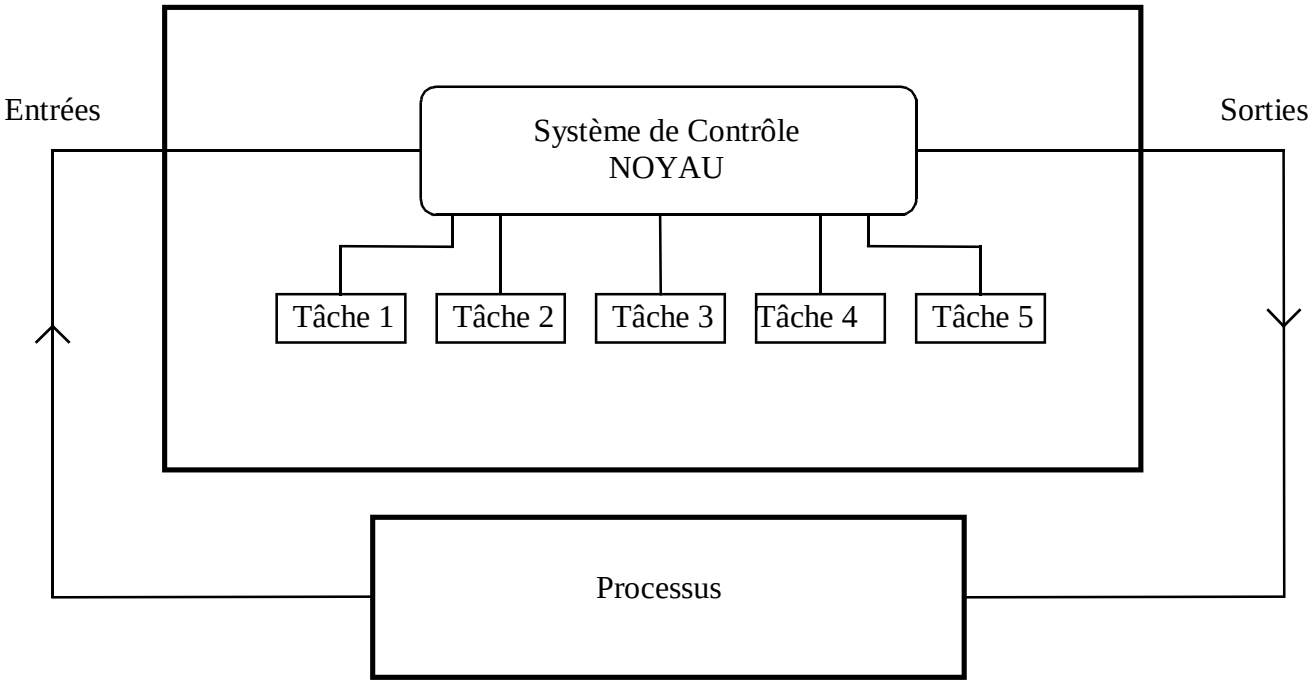
B) Fonctions et contraintes d'un Système Temps Réel

1) Décomposition en tâches ou processus

Un système de commande peut se décomposer en un ensemble de fonctions à réaliser. Suivant le noyau utilisé on parle de «processus», de «tâches » ou de «thread ».

Une partie du contrôle consiste à réaliser le séquencement et l'enchaînement des tâches en fonction d'événements internes ou externes : c'est le rôle du **NOYAU**.

L'analyse fonctionnelle peut être sous forme graphique et doit comporter à la fois la **description logique et temporelle du système.**



2) Constitution d'une tâche ou processus

Une tâche est constituée des éléments suivants :

- Un état : non crée, crée, active, en cours, en sommeil …
- Un programme binaire chargé en mémoire centrale.
- Une zone de mémoire RAM pour les variables.
- Une pile système pour les appels de procédures et les interruptions.
- Des entrées sorties ou «path » : STDIN, STDOUT, STDERR, Pipes …
- Un âge, des tranches de temps, du temps CPU...
- Une tâche mère et des tâches filles.

Chaque système temps réel possède ses propres caractéristiques et doit gérer toutes les tâches présentes. Cela est réalisé par un programme appelé **«NOYAU»**. Cette gestion se fait en utilisant un descripteur de tâches, qui est en fait une zone mémoire réservée au noyau.

C) Contrainte de temps dans un système temps réel

1) Temps de réponse

Un système temps réel est un système qui interagit avec un environnement externe. Cet environnement évolue avec le temps pour réaliser certaines fonctions. Ces fonctions sont déduites du comportement du système de commande.

2) Fonctions d'un système temps réel

Deux fonctions doivent être réalisées dans un système temps réel :

a) *Fonctions logiques :* les sorties adéquates doivent être calculées en fonction des entrées et assurer le bon comportement logique du système externe.

b) *Fonctions temporelles :* Les actions doivent être effectuées au bon moment par rapport à l'évolution des entrées du processus externe.

3) Les contraintes temporelles

a) *Souple :* système dont la performance est dégradée mais sans engendrer des conséquences dramatiques si les contraintes temporelles ne sont pas respectées.

b) *Sévère :* système dont l'incapacité de respecter les contraintes temporelles produit une panne du système.

c) *Ferme :* contrainte sévère mais où une faible probabilité de manquer les limites temporelles peut être tolérée.

4) Echelle de temps et temps de réponse

a) Le rythme d'évolution de l'environnement extérieur détermine l'échelle de temps relative à une contrainte temps réel.

b) Le temps de réponse d'un système en temps réel est la durée entre la présentation des entrées à un système et l'apparition des sorties suite aux traitements effectués sur ces entrées par le système.

c) Le temps de réponse permet d'exprimer la contrainte et est lié à l'échelle de temps d'évolution de l'environnement

5) Sévérité des contraintes et échelle de temps

a) La sévérité et l'échelle de temps d'une contrainte ne sont pas nécessairement liées.

b) On peut combiner des niveaux de sévérité de d'échelles de temps différentes dans un même système.

c) La sévérité des contraintes est généralement une caractéristique prédominante dans un système temps réel.

d) Le temps de réponse est plus souvent lié à la technologie et aux ressources matérielles utilisées.

6) Expression des contraintes du temps réel

a) Répondre à un événement en un temps donné : le temps de réponse peut être borné de façon absolue ou bien on peut se satisfaire d'un temps de réponse moyen.

b) Effectuer certaines opérations à un moment donné : agenda.

c) Traiter un nombre donné d'événements par unité de temps à un rythme donné.

d) Représentation simultanée de l'évolution logique et temporelle du système.

e) Prédiction et estimation des temps de réponse.

f) Compromis et équilibre entre le matériel et le logiciel.

7) Exemples

a) Contrôle des fonctions d'un véhicule automobile.

b) Guidage d'une fusée.

c) Simulateur de vol.

d) Commutateur de réseau de téléphonie.

e) Contrôle d'une unité de production industrielle (usine chimique, traitement des eaux, centrale nucléaire...)

f) Contrôle des fonctions d'un satellite de télécommunication.

g) Téléphone cellulaire numérique.

Chapitre 2 : Fonctionnement d'un Système temps réel

A) Les interruptions et le DMA

1) Principe des interruptions

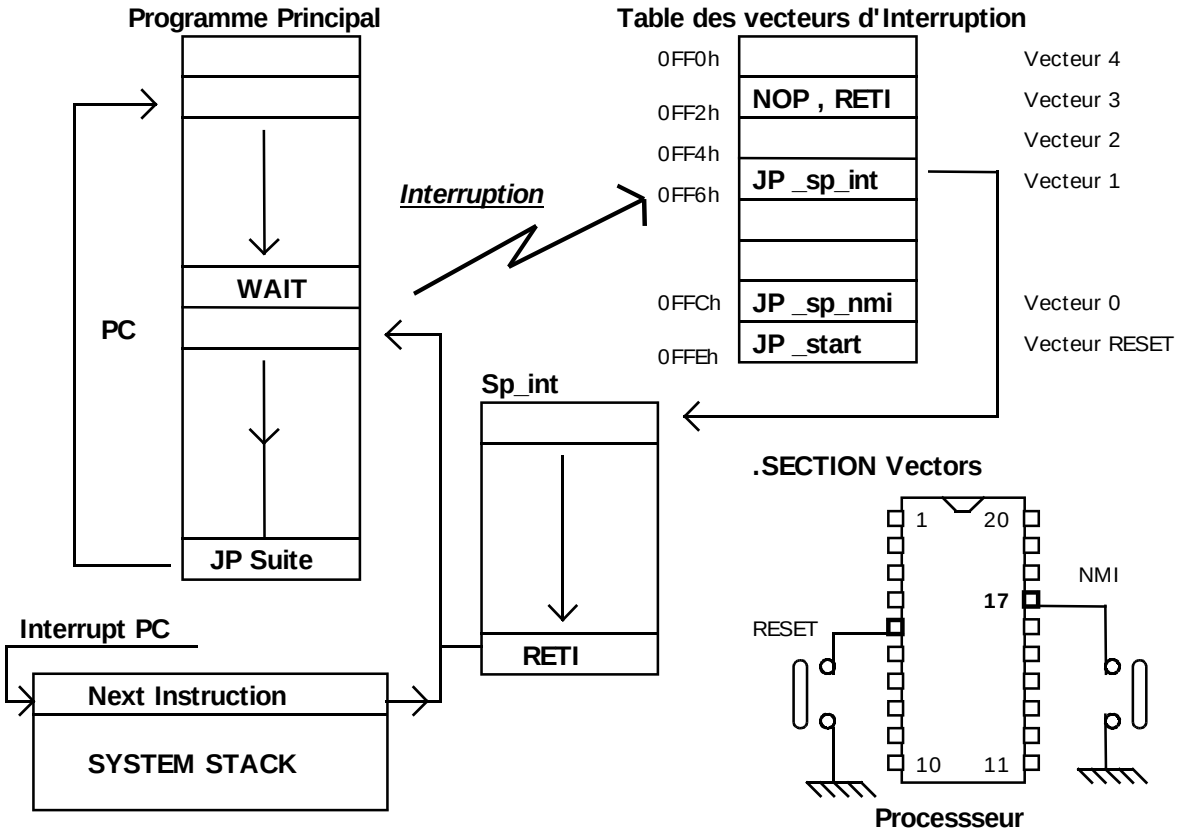
Pour respecter les contraintes du temps réel il faut pouvoir prendre en compte une information dès qu'elle est disponible.

Un système d'interruption est un mécanisme qui permet d'interrompre une partie de programme, dont l'adresse de début est connue du système, pour exécuter une séquence déclenchée par un événement asynchrone.

Le principe de base utilise un tableau de vecteurs qui contient soit toutes les adresses des procédures correspondantes, soit des instructions de saut.

Le contexte du programme principal est sauvegardé en pile système et restitué lors de la fin du programme d'interruption.

Les interruptions sont hiérarchisées en niveau. Le nombre de niveaux est fonction du processeur utilisé



2) Les interruptions matérielles

Une interruption matérielle peut être produite par un composant d'entrées sorties. Il faut alors initialiser les registres de contrôle, les vecteurs interruptions et autoriser le niveau d'interruption pour le processeur.

On parle alors de **«HWI» : Hardware Interrupt**

3) Les interruptions logicielles

Une interruption logicielle fonctionne sur le même principe qu'une interruption matérielle.
La seule différence c'est qu'elle est déclenchée par l'exécution d'une **instruction de type particulière**. On parle alors de **«SWI» : Software Interrupt (Trap).**

4) Les accès mémoire

Les accès à la mémoire centrale peuvent être de plusieurs types. On en distingue deux principaux.

a) Les accès par programme.

Dans un système temps réel on dispose en général de deux types de mémoires :

- La mémoire système utilisée par le noyau.
- La mémoire d'application.

Ces accès mémoires se font de manière habituelle par lecture ou écriture en utilisant en général de pointeurs.

Ces deux zones mémoires doivent être distinctes et indépendantes.

La zone mémoire système doit être protégée : le programme de l'application ne doit pas pouvoir y accéder. Ceci permet de garder l'intégrité du fonctionnement du noyau, garant de la sûreté du système complet.

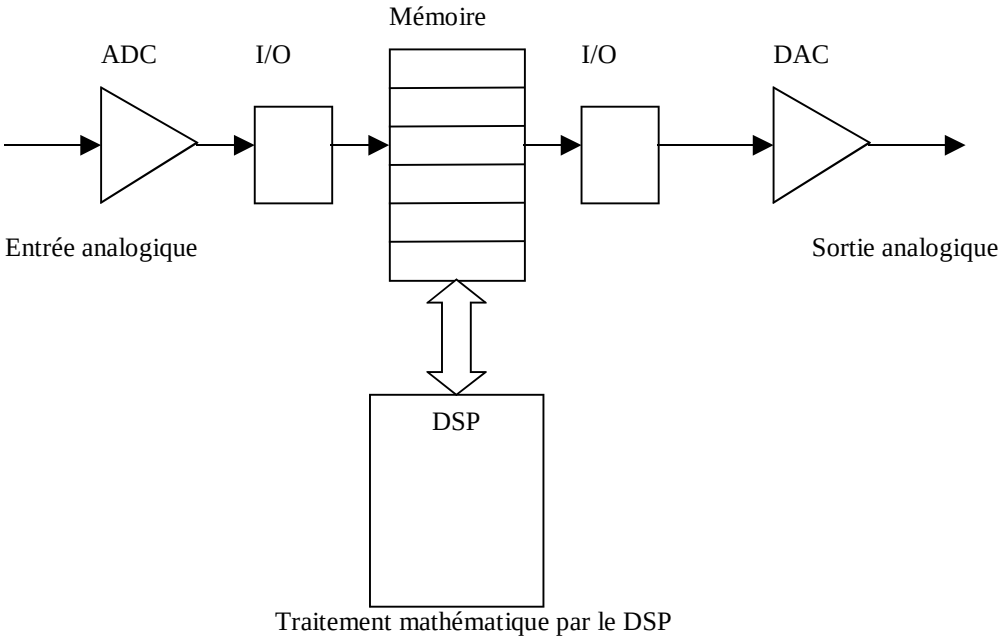
b) Les **A**ccès **D**irect **M**émoire : **DMA**

Un système temps réel doit traiter beaucoup d'informations venant du milieu extérieur. Ces données sont fournies par des composants d'entrées/sorties. Les registres de données de ces circuits périphériques possèdent en général une petite pile FIFO.

La taille de cette pile peut être insuffisante lorsque que le flux de données est trop rapide. Il faut donc disposer d'un mécanisme automatique de transfert en mémoire centrale de ces données. Ce mécanisme doit être indépendant du travail en cours effectué par l'unité centrale.

c) Exemple de circuit **DMA** pour un **DSP**

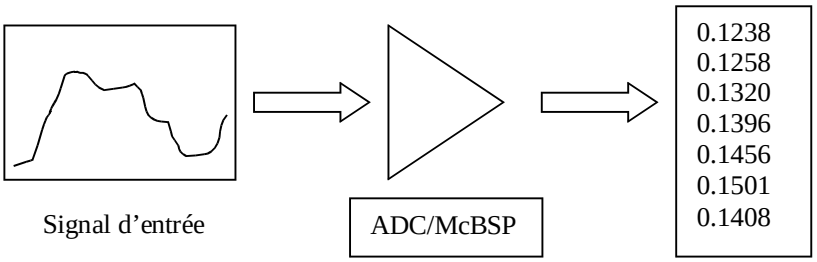
- *Principe d'une implantation de traitement analogique*



Pour un DSP les circuits ADC/DAC sont en général de type "Sigma-Delta". Leurs sorties de données sont sous forme numérique et transmise en mode série.

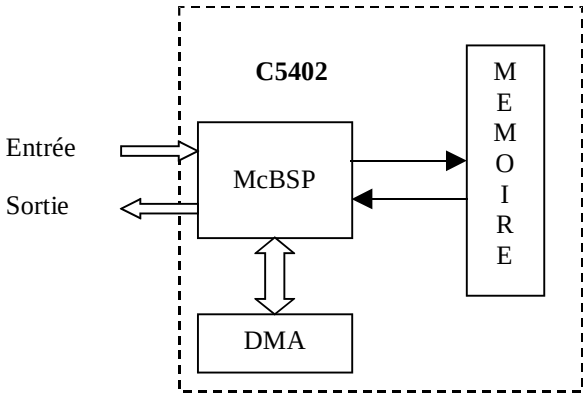
Il faut donc insérer un composant d'interface entre ces composants et le DSP. Sur les DSP de Texas instruments on dispose de ports série multi-canaux haute vitesse, intégrés sur la même puce de silicium.
Ce sont les "Multi-Channel-Bufferised-Serial-Port" : **McBSP**.

Tous les signaux sont donc échantillonnés et se retrouvent sous forme numérique dans la mémoire.



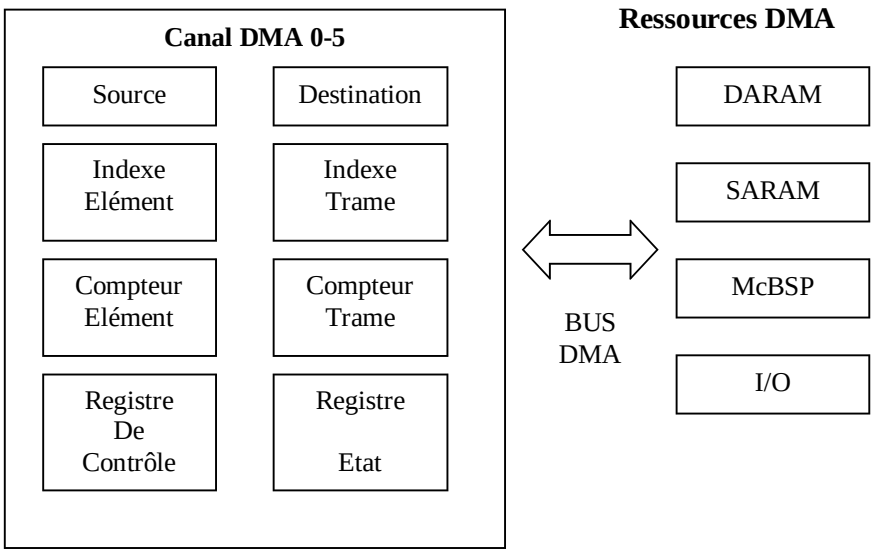
- ✓ Le couple ADC/McBSP transforme le signal d'entrée à deux fois sa fréquence maximale pour respecter le théorème de «Shannon»
- ✓ La sortie de l'ADC est un rapport de sa tension sur celle d'alimentation. Le résultat de la conversion est une fraction.
- ✓ Utiliser des fractions rend efficace le calcul en virgule fixe.

- Accès DMA et McBSP sur un DSP de type C5402.

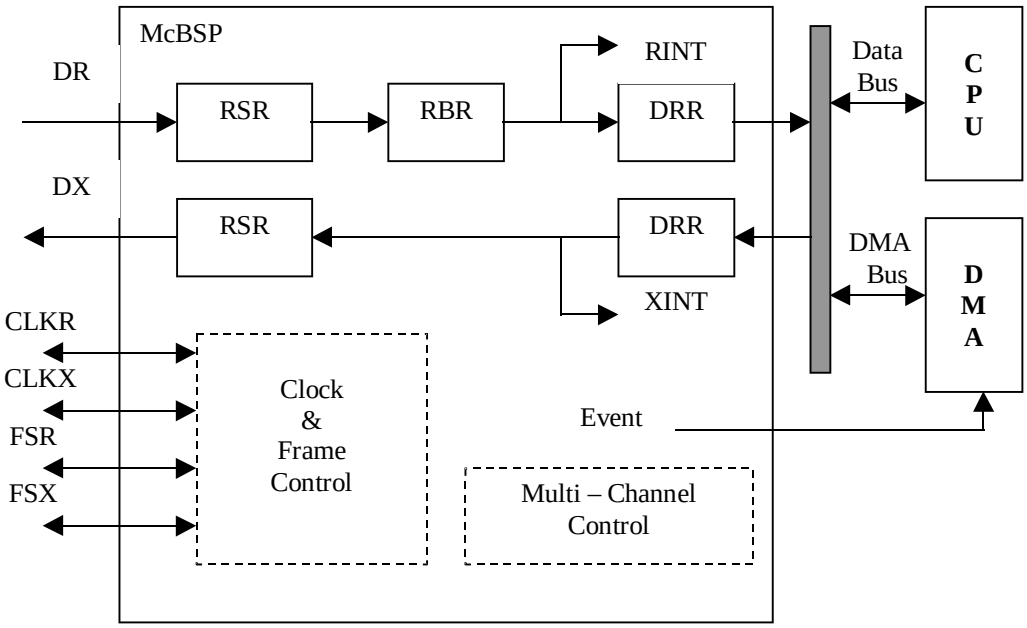


- ✓ Le McBSP communique avec les circuits externes jusqu'à 50 Mbits/seconde.
- ✓ Le circuit DMA contrôle le transfert des données depuis/vers la mémoire à 50 Mbits/seconde. (1 mot pour 4 cycles CPU)
- ✓ Le remplissage et l'extraction des données ne demandent pratiquement pas d'intervention du CPU.
- ✓ La programmation d'une fonction DMA se fait en utilisant une suite de registres spécifiques.
- ✓ Le circuit DMA permet de synchroniser le fonctionnement du CPU sur l'arrivée des données.
- ✓ Il existe des modes indexés pour transférer les données
- ✓ Un circuit C5402 comprend six canaux DMA.
- ✓ Le DMA est prioritaire sur le fonctionnement du CPU.

C5402 DMA



C5402 McBSP



B) Fonctions d'un noyau temps réel

1) Gestion de la mémoire

Dans un noyau temps réel il existe deux types de mémoire.

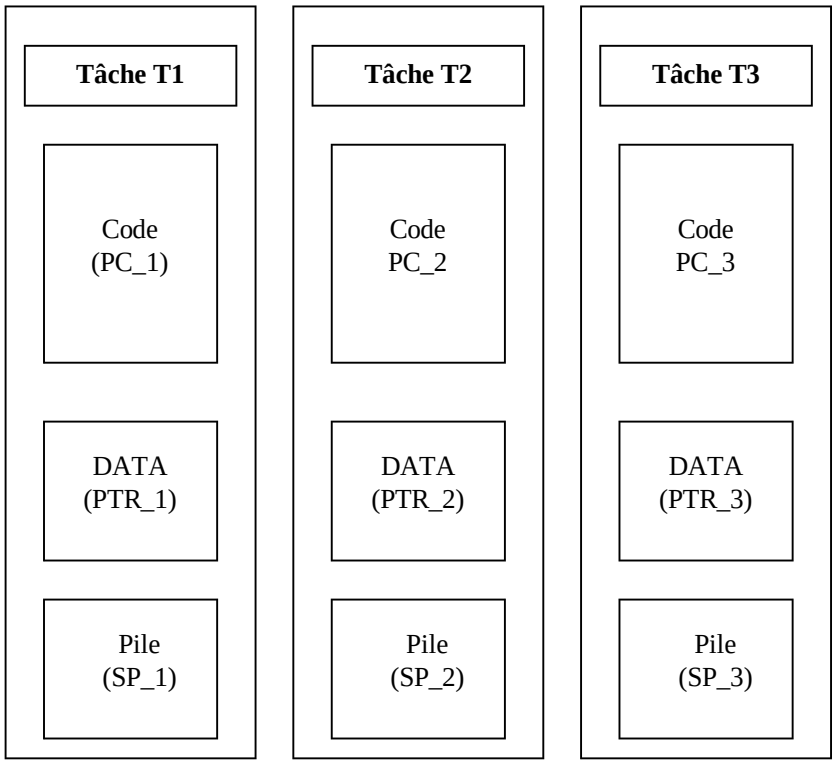
- *La mémoire système.*

  Le noyau a besoin de mémoire pour ses propres variables internes, sa pile et la gestion des interruptions, la gestion du temps et des tâches, etc. ...
  Pour garder l'intégrité du système, cette mémoire ne doit pas être accessible au programme d'application.
  En général cette mémoire n'est accessible que dans un mode dit **superviseur** du CPU et n'est pas accessible pour les tâches.

- *La mémoire d'application.*

  Chaque tâche a besoin comme le noyau de mémoire propre. Une zone mémoire pour le code exécutable pointée par le compteur programme (PC), une zone mémoire pour les données pointée par un pointeur générique (PTR), et une mémoire pour la pile pointée par le pointeur de pile (SP)
  C'est le noyau qui doit lui affecter ces zones mémoires spécifiques à la tâche lors de sa création.

Modèle mémoire comportant trois tâches T1 T2 T3



2) Gestions des tâches

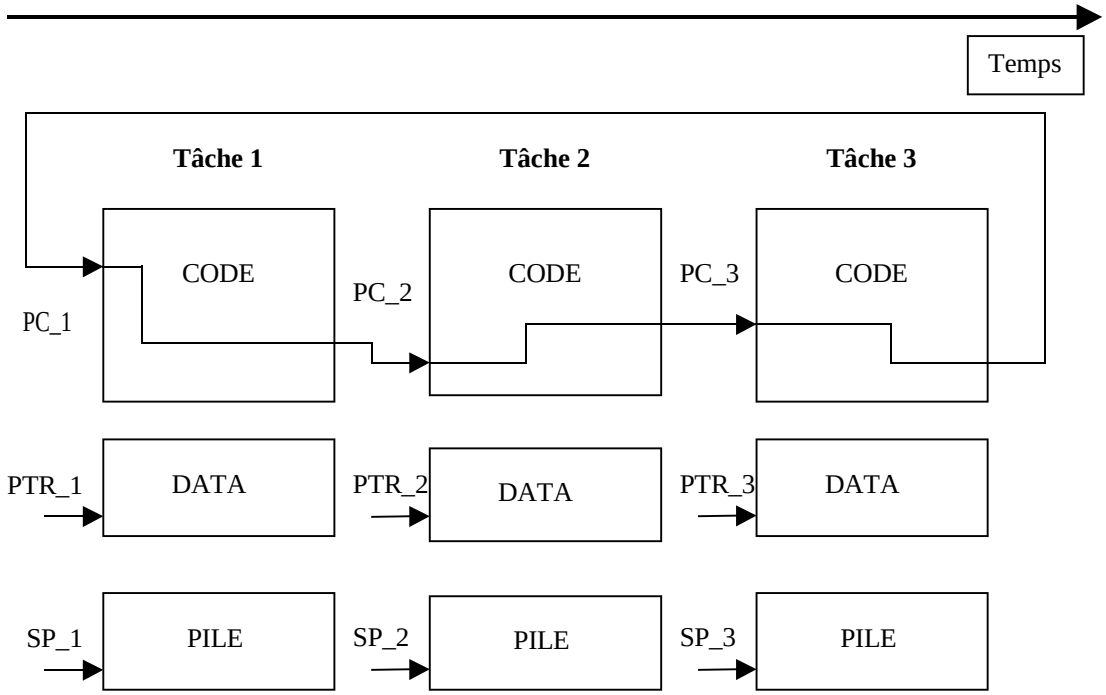
Nous avons vu qu'une tâche peut être décrite par les éléments suivants :

- Un état : non crée, crée, active, en cours, en sommeil …
- Un programme binaire chargé en mémoire centrale.
- Une zone de mémoire RAM pour les variables.
- Une pile système pour les appels de procédures et les interruptions.
- Des entrées sorties ou «path» : STDIN, STDOUT, STDERR, Pipes …

- Un âge, des tranches de temps, du temps CPU...
- Une tâche mère et des tâches filles.

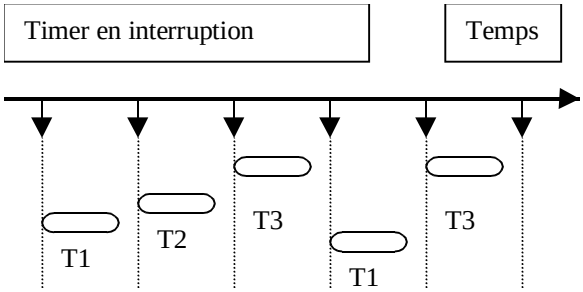
Pour gérer une tâche le noyau crée donc **une structure de données** qui contient tous les éléments ci-dessus. Cette structure de données est placée dans la mémoire système.

Au cours du temps le noyau doit pouvoir lancer la tâche ou l'arrêter. Pour cela il faut être capable de mettre en mémoire l'état des trois pointeurs définis ci-dessus et de les affecter de nouveau à la tâche lors de sa relance.
Cette opération s'appelle **sauver et restaurer le contexte d'une tâche.**



3) Gestion du temps CPU

Pour assurer les contraintes du temps réel, une des fonctions du noyau est de gérer le temps. Le principe de base de cette gestion est de découper le temps en **«tranches de temps».**
Chaque tâche peut occuper une ou plusieurs tranches de temps en fonction d'une priorité définie lors de sa création.
Le découpage du temps se fait en général à l'aide de «Timer» utilisant les interruptions : à chaque interruption le noyau décide quelle tâche doit prendre l'unité centrale et met en attente les autres.
Soit un exemple à trois tâches :



Dans cet exemple, le noyau doit partager le temps d'utilisation de l'unité centrale entre trois tâches T1, T2 et T3.
A chaque interruption il «prend la main», interrompt la tâche en cours d'exécution et il donne le CPU à celle qui est la plus prioritaire.

Cet exemple est représentatif d'un système multitâche mais ne tient pas compte des contraintes temps réel qu'il faut rajouter lors de la conception du système.

4) Gestion des entrées sorties, des interruptions et des DMA

Pour être efficace un noyau temps réel doit gérer les entrées / sorties en interruption et si possible en DMA.
Cela peut se faire par des programmes de lecture et d'écriture sur les composants d'entrées / sorties : **« les drivers en interruption »**.

REMARQUE :

Tous les programmes doivent être :

- ✓ **Relogeables : ils doivent pouvoir être implantés à n'importe quelle adresse mémoire.**
- ✓ **Interruptibles : le noyau doit pouvoir les arrêter à tout moment.**
- ✓ **Réentrants : le noyau doit pouvoir les relancer à partir de la dernière adresse d'arrêt en rechargeant complètement leur contexte**.

Le compilateur «C» de tout noyau possède ces caractéristiques.

Chapitre 3 : Ordonnancement des tâches

A) Principes de bases

1) Les différents états d'une tâche.

La figure ci-dessous représente les différents états d'une tâche et les principales transitions possibles entre ces états.
Le passage d'un état à un autre se fait en utilisant des fonctions du noyau. L'ensemble de ces fonctions s'appelle **les primitives du noyau temps réel**.

- *Etat non crée.*
  Une tâche non crée n'est pas connue du noyau temps réel. Son code binaire exécutable peut être soit sur un disque soit en mémoire morte dans d'un système en ROM.
  Le noyau ne lui a alloué aucun espace mémoire pour ses données et sa pile. La structure de donnée, descripteur de tâche n'existe pas.
  Elle ne peut passer qu'à l'état dormant ou crée.

- *Etat dormant ou crée.*
  Une tâche crée est connue du noyau temps réel : un descripteur existe, deux zones mémoires, l'une pour ses données et l'autre pour sa pile lui ont été réservée.
  Le noyau connaît l'adresse de sa première instruction pour initialiser le Compteur Programme.
  Elle peut revenir à l'état non crée ou passer à l'état prêt.

- *Etat prêt ou actif.*
  Dans cet état une tâche attend simplement que le noyau lui attribue l'unité centrale. Son lancement dépend de sa priorité par rapport aux autres tâches présentes dans le système.
  Lorsqu'elle a la priorité la plus forte le noyau lui donne le processeur et elle commence par sa première instruction.
  De cet état elle ne peut passer que vers les états dormant ou en exécution.

- *Etat en exécution ou en cours.*
  Une tâche en exécution possède le processeur. Elle déroule son code machine tant quelle est prioritaire ou quelle rencontre une condition de blocage.
  A chaque tranche de temps le noyau interrompt la tâche. Il examine si d'autres tâches demandent le processeur. **C'est l'algorithme de préemption du noyau qui détermine quelle sera la prochaine tâche en cours qui aura le CPU**.
  Lorsqu'une tâche s'arrête avant la fin d'une tranche de temps l'algorithme de préemption est en général relancé.

- *Etat bloqué, suspendu ou en sommeil.*
  Une tâche dans cet état n'est plus en possession du processeur elle est interrompue avant sa fin sur une condition de blocage. Cela peut se produire dans les cas suivants :

  - ✓ Par appel à une primitive de mise en sommeil : sleep(durée).
  - ✓ Ressource d'entrée sortie non disponible.
  - ✓ Zone de sémaphore occupé.
  - ✓ Canal de communication vide ou plein.
  - ✓ Etc.

  Pour sortir de l'état bloqué une condition de réveil doit arriver. Ce peut être soit une interruption matérielle, soit une interruption logicielle, soit un signal provenant d'une autre tâche ou tout autre événement.

  En général à la sortie de cet état un algorithme de préemption est relancé et la tâche passe par l'état prêt.

- *File d'attente de tâches.*
  Il peut y avoir plusieurs tâches dans chaque état. L'exécutif temps doit réel doit donc gérer une file d'attente.
  De manière pratique le noyau crée une liste chaînée des descripteurs de tâches dans chaque état.
  Comme les tâches peuvent être interrompues et relancées à tout moment, le noyau doit sauvegarder leur contexte, c'est à dire :

  - ✓ Le compteur programme.
  - ✓ Le pointeur de pile.
  - ✓ Les registres du microprocesseur
  - ✓ Le pointeur vers le descripteur de tâche
  - ✓ Etc.

2) Transitions entre les états d'une tâche.

- *Création d'une tâche*
  Pour créer cette tâche le noyau construit son descripteur, réserve la place pour les variables et la pile, initialise le compteur programme, le pointeur de la zone de données et le pointeur de pile.
  Dans la plupart des noyaux une tâche est connue par son numéro d'identification « ID » lors de sa création.
  Suivant les noyaux la création des tâches peut se faire de deux façons
  - ✓ A la compilation : c'est le cas des systèmes embarqués et nécessite des outils de configuration. La taille du code est réduite. Le noyau « DSP_BIOS » de Texas Instruments fonctionne sur ce principe.
  - ✓ Au moment du « RESET » il existe une tâche mère toujours la même, par exemple « sysgo » sur OS9, qui est systématiquement lancée. Son rôle est alors de créer toutes les autres en utilisant une primitive de création.

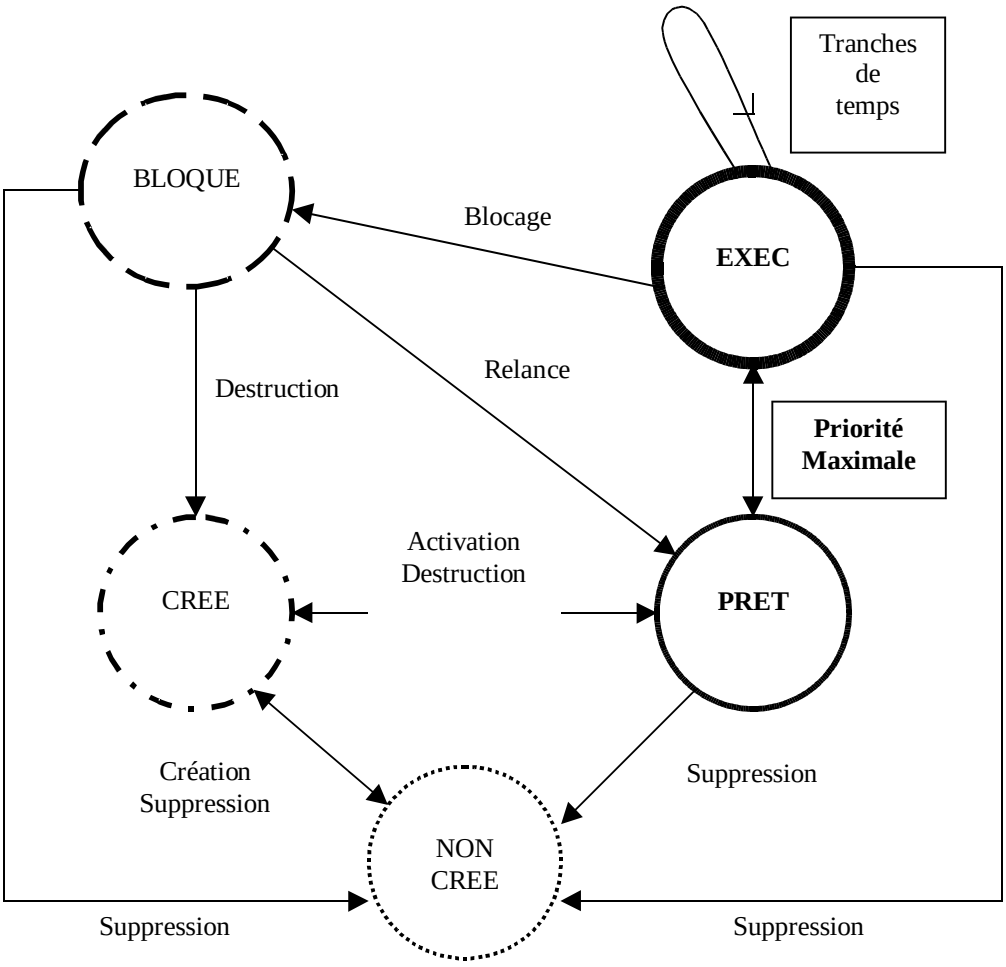
- *Notion de filiation*
  Dans un noyau temps réel la création d'une tâche peut donc se faire aussi par un appel à une primitive système depuis une « tâche mère ». Par exemple sous OS9 cette primitive a pour nom « os_fork »(…).
  Dans les paramètres d'appel de cette fonction on donne en général le nom de la tâche fille, sa priorité, les canaux de communication ouverts (path), les paramètres transmis à la tâche fille, etc.
  Une tâche fille est connue par son numéro d'identification « ID », mais possède aussi une tâche mère connue aussi par son numéro de parent « PID ».
  Un « thread » est une version réduite d'une tâche où le nombre de variables crées est plus faible.

- *Activation d'une tâche*
  L'activation d'une tâche est une demande d'exécution. Elle peut être
  - ✓ Immédiate : passage de l'état dormant à l'état prêt. Le lancement effectif est réalisé par l'ordonnanceur en fonction de sa priorité en fonction de toutes les autres taches du système.
  - ✓ Différée : même chose que ci-dessus mais avec un délai d'attente.
  - ✓ Cyclique : même chose mais de manière périodique

- *Blocage d'une tâche*
  La tâche quitte la file des tâches actives. Cela se produit soit par appel à une primitive de mise en sommeil, soit lorsque les données sur lesquelles elle doit travailler ne sont pas disponibles, périphérique non prêt, zone de communication pleine en écriture ou vide en lecture, sémaphore indisponible etc.

- *Relance d'une tâche*
  Lorsque que la condition de blocage a disparue le noyau relance la tâche. En général il a reçu un signal ou une interruption indiquant que les données dont a besoin la tâche sont disponibles. Elle repasse alors dans la file des tâches actives.

- *Fin et suppression d'une tâche*
  Lorsque qu'une tâche se termine, instruction « return() en « C », elle est détruite. Cela peut aussi se produire par l'appel à une primitive système.
  Le noyau remet alors à disposition des autres tâches les zones mémoires libérées.



3) Systèmes avec ou sans préemption du processeur.

Il existe deux types de systèmes temps réels avec ou sans préemption (réquisition) du processeur.

a) Systèmes sans préemption.

Dans ce type de système une tâche s'exécute jusqu'au moment où elle fait appel à une primitive du noyau. A cet instant le noyau peut décider si elle doit continuer ou non en fonction des priorités des autres tâches.
Une application utilisant ce type de noyau doit être bien conçue, car une tâche qui se bloque peut provoquer le blocage de l'ensemble des autres tâches.
Il est cependant facile à réaliser

b) Systèmes avec préemption

Dans ce type de noyau une tâche peut perdre le CPU à tout instant. Pour réaliser ce mécanisme on utilise un « timer » qui découpe le temps en tranches de temps par interruption. A chaque interruption le noyau donne le CPU à la tâche la plus prioritaire suivant un algorithme précis.

B) Préemption, réquisition, ordonnancement des tâches

1) Structure d'un Descripteur de tâche

a) Les éléments constituant un descripteur de tâche.

Pour pouvoir lancer une tâche le noyau a besoin de connaître :

- ✓ L'adresse de son point d'entrée courant (PC)
- ✓ Son état.
- ✓ Sa priorité.
- ✓ L'adresse du pointeur de pile (SP).
- ✓ L'adresse de la zone de données.
- ✓ La valeur des registres du processeur lors de son précédent arrêt.

Descripteur de tâche

Pointeur Vers le début De la tâche	Status	Priorité	Pointeur De Pile	Pointeur Vers les Données	Pointeur Vers les Registres

b) Exemple d'un descripteur

Pointeur de fonction comme pointeur de tâche.

```
typedef  void      (*TACHE_ADR)(void)
```

Structure de données pour les registres du processeur.

```
typedef struct {
        Reg1 ;
        Reg2 ;
        …
}REGISTRES ;
```

Une structure de données peut alors simplement représenter un descripteur de tâche :

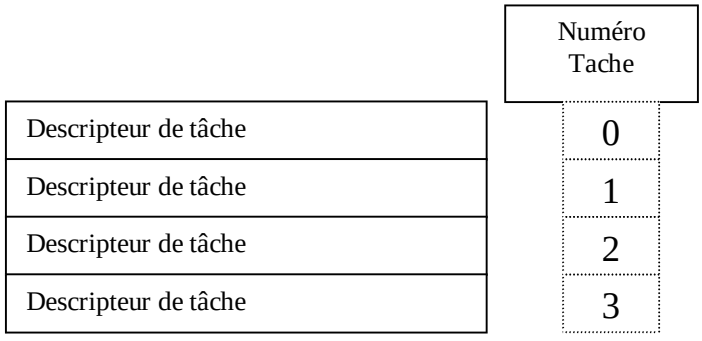
```
typedef struct {
        TACHE_ADR    tache_adr ;
        Char         etat ;
        Char         priorité ;
        Ptr          *pile ;
        Ptr          *données ;
        REGISTRES    registres ;
}TACHE;
```

c) Toutes les tâches d'un système.

Pour décrire toutes les tâches d'un système on peut utiliser soit

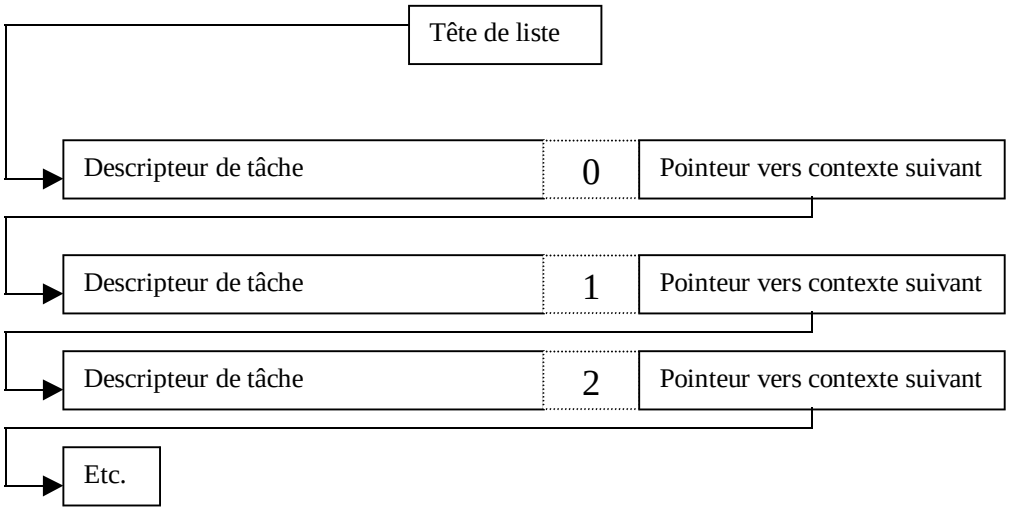
- ✓ Un tableau de taille fixe : le nombre de tâche maximal est fixé à l'avance par la taille du tableau, la gestion des tâches consiste simplement à gérer l'indice dans le tableau.

Tableau de tâches



- ✓ Une liste chaînée : c'est la taille mémoire disponible qui limite le nombre de tâches. La gestion des tâches est un peu plus complexe.

Liste chaînée des tâches



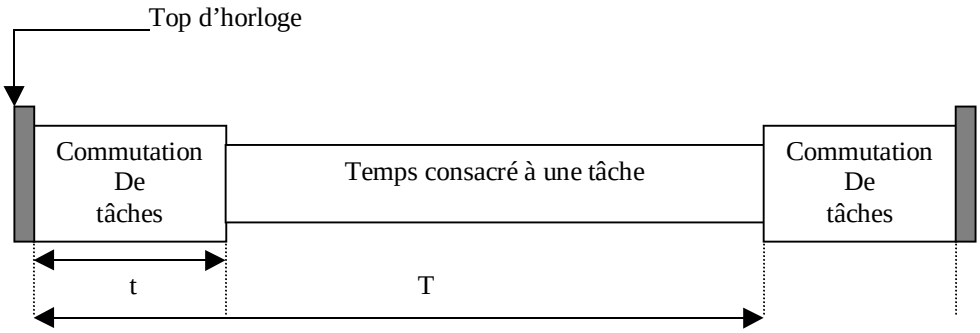
2) Qualités d'un ordonnancement.

L'ordonnancement est le mécanisme par lequel tout système multitâches choisit la tâche qui doit être en possession de l'unité centrale.
Dans les systèmes avec disque de type UNIX, Windows ou autre ce mécanisme peut être complexe et doit être transparent pour l'utilisateur. Tout cela est coûteux en temps processeur et ne peut être utilisé dans des applications temps réel. Il existe cependant des extensions temps réel de ces systèmes.
Dans une application temps réel il faut prendre en compte le contraintes de temps dès la conception. Les noyaux doivent hiérarchiser les tâches mais rester souple pour s'adapter à toute application.

a) Efficacité

Le processeur doit consacrer le maximum de temps à l'application et le minimum à la commutation des tâches.

Tranche de temps et utilisation du processeur



La variable « T » représente la tranche de temps et la variable « t » le temps de commutation.
On peut définir l'efficacité comme le rapport :

$$E = \frac{T - t}{T}$$

Plus « E » est voisin de « 1 » pour le système est efficace. Dans un système « T » est fixe et « t » varie. Plus le nombre de tâches à commuter augmente, plus l'efficacité diminue.
Avec un processeur classique « T » est voisin de 1 ms et il faut éviter que l'efficacité soit inférieure à 90%.

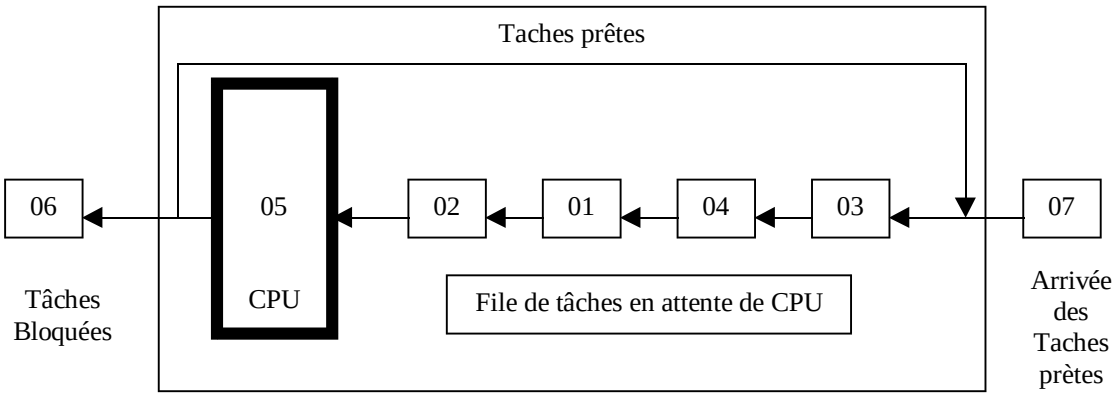
b) Temps de réponse, traitement des interruptions
Le temps de réponse est le temps mis par le noyau pour répondre à une sollicitation externe. Pour un noyau de tranche de temps de 1ms il faut un temps de réponse de l'ordre de 1 µs.
Le temps de réponse aux interruptions est le critère le plus important d'un noyau temps réel. Toute entrée/sortie matérielle est gérée soit par interruption, soit par DMA. Ce temps doit être connu car il fixe le délai à partir une tâche peut être lancée par le matériel.
Pendant la commutation de tâche le noyau est en mode « superviseur » et toutes les interruptions doivent être mémorisées pour être traitées par la suite. Il peut en être de même lors de l'exécution des primitives du système. Cela peut retarder la pris en compte des interruptions.

c) Impartialité.
L'ordonnancement doit effecteur un partage équitable du processeur.

d) Débit
Suivant les critères temps réel de l'application, le noyau doit donner du temps CPU à toutes les tâches qui le demande en fonction de leur priorité.

3) Ordonnancement circulaire.

C'est un algorithme simple qui fait une permutation circulaire des tâches.



e) Avantages
- ✓ Simple à mettre en œuvre
- ✓ Toutes les tâches on au moins une fois le CPU.

f) Inconvénients
- ✓ La priorité est simplement l'ordre d'arrivée des tâches.
- ✓ Les tâches ne peuvent pas garder le CPU entre deux tranches de temps.

4) Ordonnancement par priorité.

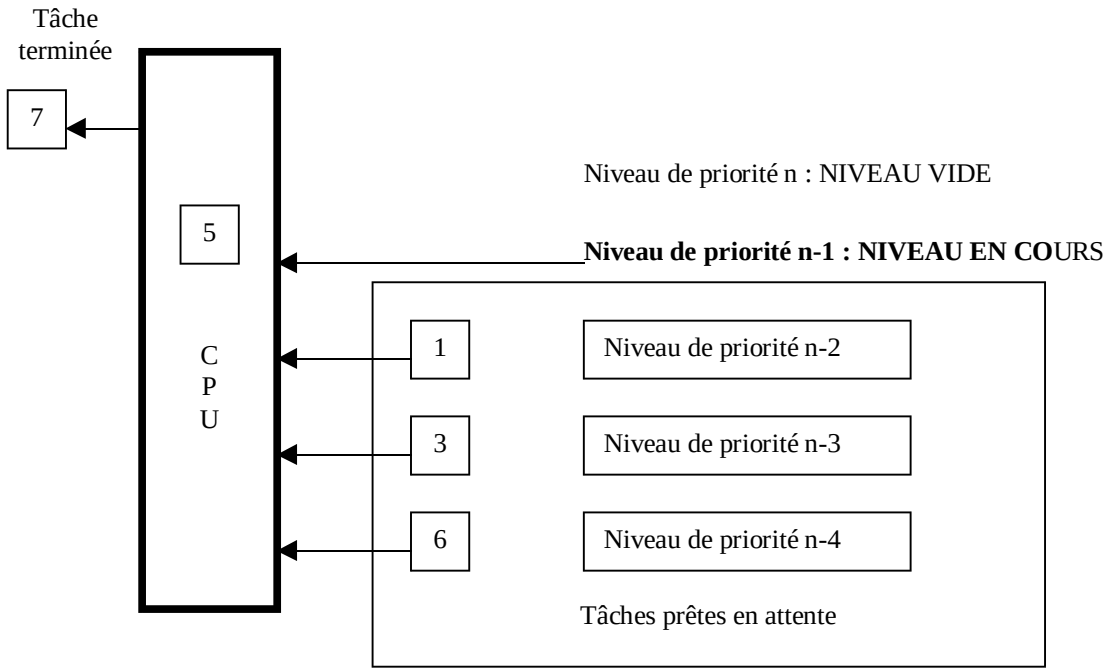
Dans l'exemple précédent toutes les tâches avaient en fait la même priorité. Cela n'est pas souvent acceptable dans les systèmes temps réel.

Dans l'ordonnancement par priorité chaque tâche a une priorité reçue lors de sa création. La tâche la plus prioritaire dispose du CPU.

Si une interruption intervient, elle peut lancer une nouvelle tâche et le noyau doit de nouveau exécuter un algorithme de préemption. Si la tâche courante est moins prioritaire que celle lancée par l'interruption, elle doit laisser sa place même si sa tranche de temps n'est pas finie.

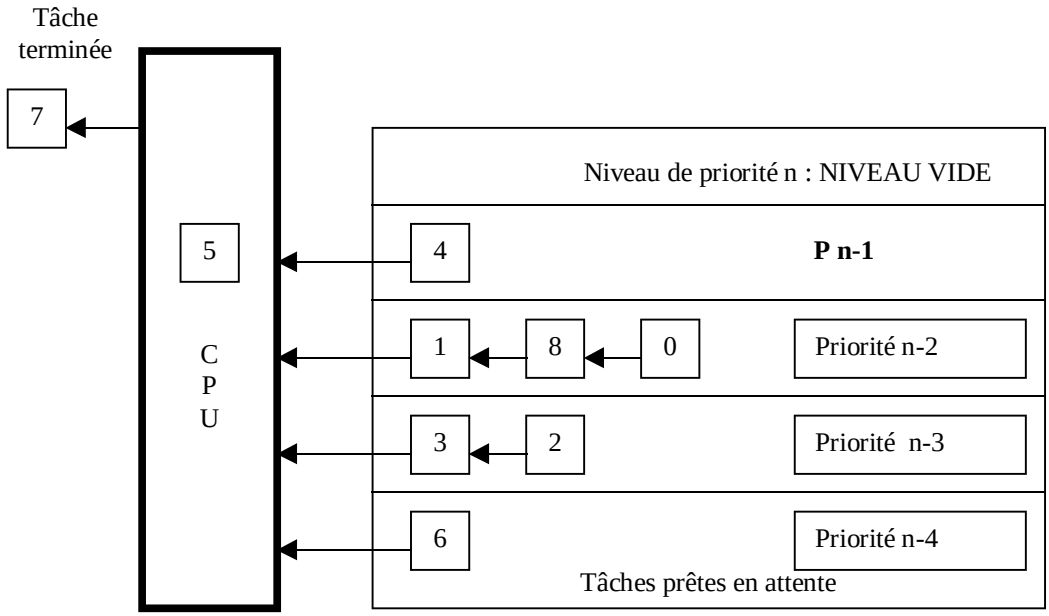
Si une tâche se termine avant la fin d'une tranche de temps le noyau exécute un algorithme de préemption.

L'emploi de ces noyaux est intéressant pour des applications avec beaucoup d'entrées sorties et peu de calculs.



5) Ordonnancement par priorité et files multiples.

Ce principe est une combinaison des deux modes précédents : on dispose d'une file circulaire par niveau de priorité.
Suivant le nombre de tâches par niveau il est facile de contrôler la charge du CPU.



S'il n'y a qu'un seul niveau avec plusieurs tâches on retrouve l'algorithme ciculaire.
S'il n'y a qu'une seule tâche par niveau on retrouve l'algorithme par niveau.
Dans l'exemple ci-dessus on a une combinaison des deux systèmes.
Cette méthode est la plus utilisée dans les noyaux temps réels embarqués (WxWorks, Psos, etc.)

6) Ordonnancement par priorité, files multiples et vieillissement.

Dans un algorithme circulaire c'est l'ordre d'arrivée des tâches dans la file d'attente des tâches prêtes qui donne la priorité pour un niveau.
Dans un algorithme par vieillissement chaque tâche possède une priorité initiale en général codée de [1 à 256].
A chaque tranche de temps toutes les tâches qui n'avaient pas le CPU voient leurs priorités augmenter de une unité : elles changent de niveau.

Le noyau donne alors le CPU à la tâche qui est la plus vieille dans le plus haut niveau.
Quand une tâche a obtenu le CPU elle reprend sa priorité initiale.

Exemple de trois tâches T1 niveau 128, T2 niveau 130 et T3 niveau 131. On suppose que les tâches sont arrivées à l'instant initial dans l'ordre T1, T2 puis T3.

Dans le tableau ci-dessous est indique en caractères gras la tâche qui dispose du CPU à chaque tranche de temps :

Temps	0	1	2	3	4	5	6	7	8	9	10
T1	128	129	130	**131**	128	129	130	131	**132**	128	129
T2	130	**131**	130	131	**132**	130	**131**	130	131	**132**	130
T3	**131**	131	**132**	131	132	**133**	131	**132**	131	132	**133**

On définit en plus deux priorités modifiables dynamiquement par des tâches de type superviseur :

- ✓ MAX_AGE : un âge à partir duquel les tâches ne vieillissent plus.
- ✓ MIN_PRIOR : un âge en dessous duquel les tâches ne peuvent plus avoir le CPU.

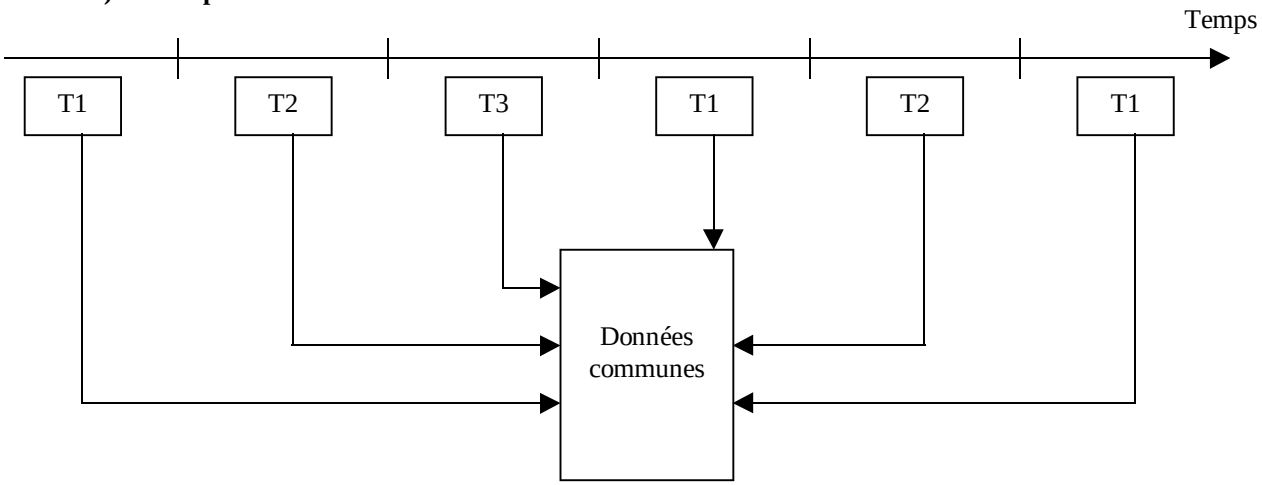
Cette méthode est utilisée dans les noyaux temps réels à vocation mixte : noyau temps réel et système d'exploitation de type UNIX, OS9, QNX etc.

Chapitre 4 : Communication inter tâches

A) Le partage de ressources mémoire ou de périphériques.

Les tâches doivent coopérer en échangeant des données. Ces données peuvent être de type simple ou complexe.
En programmation monotâche les échanges de données se font par le passage de paramètres entre les procédures ou l'utilisation de variables globales. Il n'y aucune ambiguïté sur les valeurs passée, car à un instant donné une seule section de programme a accès à ces données.
Dans un système multitâches une tâche peut accéder à des données et être interrompue par le noyau. Une autre tâche peut alors accéder à ces mêmes données. **On parle alors d'accès concurrents.** Des tâches peuvent donc lire ou écrire des données communes et le résultat dépend de l'ordonnancement des tâches.

1) Exemple.



Supposons dans cet exemple que T1 fournit des données à T2, que T2 les transforme et que T3 les utilise.
On voit sur ce shéma que T1 n'a pas fini décrire ses données avant que T2 ne les transforme. Il en est de même pour T3 qui vient les exploiter avant que T1 et T2 aient fini leur travail.
Il faut donc disposer de mécanismes qui s'assurent que tout le travail se fasse dans l'ordre pour qu'il n'y ait aucune ambiguïté sur la valeur des données en mémoire exploitées par les tâches.

2) Section critique

Pour résoudre ce problème il faut donc interdire l'accès simultané à des données partagées par plusieurs tâches.
Cela revient à mettre en place des mécanismes d'**exclusion mutuelle**.
Une **section critique** est une partie de programme qui ne peut pas être interrompue par une autre tâche. La durée de cette partie doit être la plus courte possible. Les règles qui régissent les sections critiques sont les suivantes :

- ✓ A un instant donné une seule tâche doit occuper une section critique
- ✓ Aucune hypothèse ne doit être faite sur le nombre et la durée des tâches en exécution.
- ✓ Toute tâche suspendue en dehors d'une section critique ne doit pas en bloquer une autre.
- ✓ Une tâche doit attendre un temps fini devant une section critique.

a) Exclusion mutuelle par masquage des interruptions.

C'est la méthode la plus simple : le masquage des interruption bloque l'ordonnanceur et les tâches matérielles.
Le système est alors en mode monotâche : la seule tâche active est celle qui accède à la section critique.

- ✓ Avantage :
  Système très simple à mettre en œuvre

- ✓ Inconvénients :
  Le système ne répond plus aux entrées sorties par interruption.
  Le masquage des interruptions doit être fait par le noyau et non les tâches.
  Le temps de réponse peut augmenter et ne plus respecter les critères du temps réel.
  Le système peut se bloquer complètement.

CONCLUSION :
Il faut réserver ce principe au noyau pour mettre à jour ses propres variables globales.

b) Exclusion mutuelle par variable verrou et attente active

Une variable de type entier peut servir de verrou : si verrou vaut 0 la ressource est libre, si verrou vaut 1 la ressource est occupée.
L'algorithme peut être le suivant :

```
1     Entier          verrou = 0
2     TANT QUE (verrou == 1)
3           Attendre
4     verrou = 1
5     /* début de section critique */
6           ------
7           ------
8     /* fin de section critique */
9     verrou = 0
10    -----------
```

Cette solution est relativement simple à mettre en œuvre. Cependant la tâche qui est en attente sur le verrou consomme du temps CPU inutilement.
Si une interruption ou un tranche de temps arrive entre le début et la fin des lignes 234 une autre tâche peut prendre la ressource.
Pour éviter ce problème chaque microprocesseur possède une instruction assembleur qui réalise les lignes **234** de manière indivisible et non-Interruptibles, par exemple TAS (Test And Set) pour un 68000 de Motorola.
Si plusieurs objets sont à protéger il est possible d'utiliser plusieurs verrous.

c) Exclusion mutuelle par variable verrou et mise en sommeil de tâche.

Il est possible d'améliorer le programme ci-dessus part l'utilisation de deux primitives « dort » et « réveille ».
Le programme devient alors :

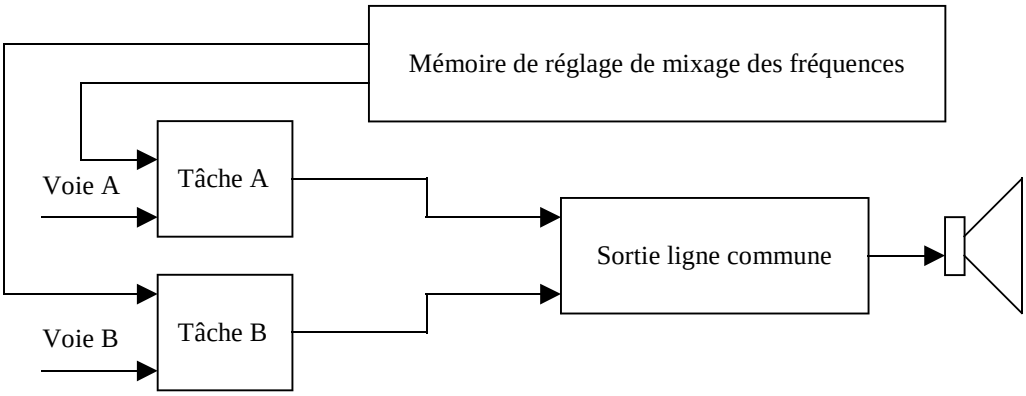
```
1     Entier          verrou = 0
2     SI (verrou == 1)
3           dort()
4     verrou = 1
5     /* début de section critique */
6           ------
7           ------
8     /* fin de section critique */
9     verrou = 0
10    réveille(tâches en sommeil sur verrou)
11    -----------
```

La primitive **« void dort(void) »** place la tâche dans la file d'attente des tâches en sommeil.
La primitive **« réveille(tâches en sommeil sur verrou) »** envoie un signal de type interruption logicielle à toutes les tâches en sommeil sur le verrou.

3) Interblocage mutuel

a) Exemple

Soit une table de mixage numérique. Deux tâches doivent lire un signal sonore chacune sur une voie d'entrée ligne et sortir un son mélangé sur une seule sortie ligne en utilisant le même réglage de répartition des fréquences de mixage stocké en mémoire centrale.
Du fait de la structure du système une seule tâche ne peut accéder à la mémoire de mixage et à la ligne de sortie à un instant donné.



- ✓ La tâche A accède au contenu de la table de mixage des fréquences et **se la réserve**, puis tente d'accéder à la ligne de sortie **utilisée par la tâche B** -> *elle se bloque*.
- ✓ La tâche B cherche à lire le contenu de la table de mixage. Comme elle est **utilisée par A** *elle se bloque aussi*.

Les deux tâches sont alors bloquées. Cette situation peut donc se produire quand au moins deux tâches doivent accéder simultanément à deux ressources communes : On parle d'*étreinte fatale.*

b) Solutions

- ✓ La seule solution pour éviter un Interblocage est de libérer toutes les ressources utilisées avant d'en accaparer une autre.
- ✓ La seconde solution est limiter dans le temps le blocage des tâches en attente de libération des ressources partagées.

Dans le cas d'applications plus complexes ce problème n'est pas facile à détecter ! ! !

4) Les sémaphores

Dans les exemples précédents la variable verrou est crée par une tâche et doit être transmise aux autres tâches qui l'utilisent lors de leurs créations.
Les sémaphores sont des mécanismes analogues d'exclusion mutuelle, mais gérés par le noyau. Ce sont des variables globales accessibles par toutes les tâches.
En général une seule tâche doit un créer un sémaphore en indiquant au noyau son nom, elle reçoit en retour un numéro d'identification.

a) Les primitives du noyau.

Pour gérer les sémaphores tout noyau possède les primitives suivantes :

- ✓ Entier CREER_SEM(nom) : création d'un sémaphore connu par son nom.
- ✓ Entier LIEN_SEM(nom) : se lier à un sémaphore pour obtenir son numéro.
- ✓ Booléen ATT_SEM(num) : attendre la libération d'un sémaphore.
- ✓ Booléen LIB_SEM(num) : libère le sémaphore.
- ✓ Booléen DEL_SEM(num) : destruction de la variable sémaphore.

Ces fonctions sont indivisibles et peuvent avoir des variantes suivant le type de sémaphore crée.

b) Les sémaphores booléens : «mutex » (mutuelle exclusion)
Un sémaphore booléen est une variable globale du noyau qui ne peut prendre que deux valeurs «0» ou «1».
Quand le sémaphore vaut «0» il est libre, quand il vaut «1» il est occupé par une tâche.
La fonction LIEN_SEM teste sa valeur et met la tâche en sommeil si le sémaphore est occupé. Elle est réveillée par un signal logiciel quand une autre tâche le libère en exécutant la primitive LIB_SEM.
D'où l'algorithme :

Tâche de création
1 CREER_SEM
2 ……

Tâche d'utilisation
1 LIEN_SEM
2 ……
3 ATT_SEM
4 …..(mise en sommeil de la tâche par le noyau si le sémaphore est occupé)

1	**Début Section Critique**
5	**……...**
2	**Fin Section Critique**

6 LIB_SEM (libération du sémaphore et envoie d'un signal aux autres tâches)
7 ……

c) Les sémaphores à compte.
Le principe des sémaphores à compte est identique à celui des sémaphores booléens. La variable sémaphore et cette fois ci un entier qui peut prendre une valeur négative, nulle ou positive.
- ✓ La fonction CREE_SEM crée le sémaphore avec une valeur initiale positive.
- ✓ La fonction ATT_SEM décrémente le sémaphore de «-1» et met la tâche en sommeil s'il est négatif.
- ✓ La fonction LIB_SEM incrémente le sémaphore de «+1» et envoie un signal à toutes les tâches bloquées sur le sémaphore.

CONCLUSION : Ce type de sémaphore permet à plusieurs tâches de partager la même ressource. Par exemple si la valeur initiale est «3», trois tâches peuvent partager une ressource est la quatrième est bloquée.

d) Les sémaphores privés.
Un sémaphore est privé quand une seule tâche peut exécuter la primitive «ATT_SEM». Les autres tâches ne peuvent exécuter que la fonction LIB_SEM.
Ce mécanisme permet de synchroniser les tâches. (voir chapitre suivant)

B) Communication par message

1) Principes et primitives du noyau

a) Principes
Les méthodes ci-dessus nécessitent de gérer les échanges entre tâches de manière détaillée. Pour communiquer des données entre tâche on utilise le modèle **Producteur/Consommateur.**
Il existe dans tout noyau des primitives de plus haut niveau qui facilitent ces échanges.
Un message est une zone mémoire commune à plusieurs tâches ou l'on peut déposer des informations soit sous forme binaire soit sous forme ASCII.

b) Primitives
On dispose des primitives :
- ✓ CREER_MESSAGE(nom_mess)
- ✓ LIRE_MESSAGE(expéditeur , nom_mess)
- ✓ ECRIRE_MESSAGE(destinataire , nom_mess)
- ✓ DETRUIRE_MESSAGE(nom_mess)

Le message peut être connu soit par son nom soit par un numéro d'identification.

2) Communication par boite à lettre

Une boite à lettre est une zone mémoire pouvant contenir plusieurs messages. C'est une structure de données crée dynamiquement par le noyau à chaque appel à la fonction CREE_MESS.
Si une tâche vient lire une boite à lettre vide elle est mise en sommeil. Elle est réveillée quand un message qui la concerne à été déposé dans la boite à lettre.
Si une tâche vient écrire dans une boite à lettre pleine elle est mise en sommeil. Elle est réveillée quand la boite à lettre a été libérée par une opération de lecture.

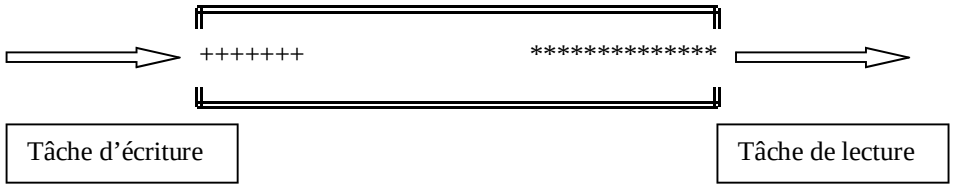
Message 1	Producteur	Consommateur	Etat	Contenu
Message 2	Producteur	Consommateur	Etat	Contenu
Message 3	Producteur	Consommateur	Etat	Contenu
Message 4	Producteur	Consommateur	Etat	Contenu

3) Communication par tube : « pipe »

Un « PIPE» est une zone mémoire de type FIFO de taille fixe. En langage « C » on le gère comme un canal de communication identique au PATH « STDIN », « STDOUT » et « STDERR ».
Pour cela le programmeur a besoin primitives suppléméntaires :

- ✓ PIPE_OPEN(/nom_canal,mode) ouverture du canal en lecture,écriture, modification, ajout.
- ✓ PIPE_WRITE(/nom_canal) écriture dans le canal dont le numéro est donné par la fonction PIPE_OPEN.
- ✓ PIPE_READ(/nom_canal) lecture dans le canal.
- ✓ PIPE_CLOSE(/nom_canal) fermeture du canal.

Si une tâche vient lire un PIPE vide, elle est mise en sommeil. Elle est réveillée quand une donnée été déposée dans le PIPE.
Si une tâche vient écrire dans un PIPE plein, elle est mise en sommeil. Elle est réveillée quand une place dans le PIPE est disponible.



Il est possible d'utiliser les fonctions de la bibliothèque d'entrées sorties standard du langage « C » pour utiliser un PIPE. A savoir :

- ✓ **open(), fopen()**
- ✓ **read(), fread()**
- ✓ **write, fwrite**
- ✓ **close(), fclose()**

Chapitre 5 :Mécanismes de synchronisation

A) Définitions

1) Rôles des fonctions de synchronisation.

Dans un système temps réel l'intérêt de décomposer en tâches une application et de permettre la mise en place plus aisée de fonctions.
Ces fonctions peuvent être indépendantes les unes des autres mais la plupart du temps elles doivent collaborer et s'échanger des données. C'est le cas par exemple entre des tâches de type matériel et des tâches de type logiciel.
L'analyse du problème doit comporter deux approches :

- ✓ Une approche de fonctions logiques à réaliser.
- ✓ Une approche temporelle à respecter.

Les mécanismes de synchronisation permettent de fixer l'organisation de ces deux approches en respectant le cahier des charges de l'application.

2) Types de mécanismes mis en œuvre.

La synchronisation des tâches nécessite la présence des mécanismes suivants.

- ✓ Activation d'une tâche avec passage d'information.
- ✓ Auto blocage ou blocage d'une autre tache.
- ✓ Envoi de signaux de réveil à une ou plusieurs tâches.
- ✓ Installation d'une routine de réception de signal (interruption logicielle).

L'envoi de signaux peut être mémorisé ou non. Dans le premier cas si la tâche est active le signal sera exploité lors du retour de la tâche en sommeil. Dans le second cas le signal est perdu.

B) Synchronisation directe

Une synchronisation est directe quand le noyau possède les primitives d'envoi de signaux en précisant le destinataire.

1) Fonction d'arrêt

La primitive **bloque(numero_tache , code)** envoi un signal d'arrêt à une tâche en mode prêt. Elle sauvegarde le contexte de la tâche. Suivant le noyau utilisé le code peut avoir plusieurs signification :

- ✓ Mise de la tâche en mode bloqué.
- ✓ Mise de la tâche en mode crée.
- ✓ Mise de la tâche en mode non crée.
- ✓ Etc.

Toutes ces actions sont dépendantes des différents états possibles accordés par le séquenceur aux tâches.

2) Fonction mise en sommeil

La fonction **dort(temps)** permet à une tâche de se mettre en sommeil soit permanent soit pour une durée déterminée.
En général un temps de valeur nulle correspond à une mise en sommeil de durée infinie.

3) Fonction de réveil

La fonction **signal(numero_tache , code)** permet d'envoyer un signal de réveil à une tâche qui est endormie.

4) Installation d'une routine d'interception de signal.

Pour recevoir un signal il faut installer une procédure d'interruption logicielle. Pour cela on utilise une fonction du type **interception(sp_signal)** qui place la routine **sp_signal** dans la table des fonctions d'interruption.

Toute routine d'interruption doit être **très courte** car elle est exécutée directement par le noyau et n'est pas elle-même interruptible.

On trouvera ci dessous la forme que peut prendre l'algorithme d'une tâche qui utilise un tel mécanisme :

```
1       /* définition d'une routine d'interception de signal */
2           intercept       sp_signal(code)
3             DEBUT
4                   SELON(code)
1                       code_1 : action_1
2                       code_2 : action_2
3                       etc.
8             FIN
```

Les **"actions_x"** sont en général un chargement de valeur dans une variable globale : cela permet d'avoir une routine d'interception de signal qui soit la plus courte possible.

```
N       /* programme principal */
N+1         prog_princ()
N+2         DEBUT
N+3                 interception(sp_signal)
N+4         ……..
N+x                 REPETER
N+x+1               ……
N+y                         dort(temps)
N+y+1               ……..
N+y+2       FIN
```

- ✓ A la ligne N+3 le programme principal installe la routine d'interception de signal
- ✓ A la ligne N+y la tâche se place elle-même en sommeil.
- ✓ Lors de la réception du signal de valeur (code), le noyau lance la procédure « sp_signal». En fin de cette procédure on retourne en séquence à la ligne N+y+1.
- ✓ Il suffit alors de tester la variable globale mise à jour pour connaître le signal reçu.

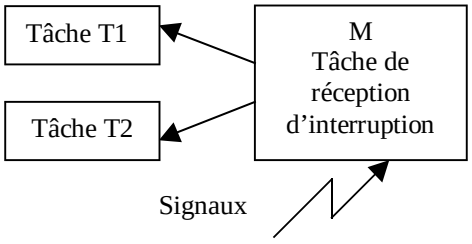
C) Synchronisation indirecte

Dans une synchronisation indirecte on n'utilise plus directement le nom ou le numéro de la tâche, mais des variables globales du noyau comme les **sémaphores,** les **événements** ou les variables de **rendez-vous**.

1) Synchronisation par sémaphore

Un sémaphore booléen ou à compte permet simplement de synchroniser deux tâches à partir d'une troisième.

Soit deux tâches logicielles T1 et T2 qui doivent être synchronisée par une interruption matérielle.



Chaque tâche peut créer un sémaphore privé initialisé à zéro, soit P1=0 pour T1 et P2=0 pour T2. Elles se mettent en sommeil chacune sur son sémaphore P1=-1 et P2=-1.

Lorsqu'une interruption matérielle arrive, la tache matérielle M envoie un signal aux deux sémaphore P1 et P2 qui repassent alors à la valeur 0. Cela permet de réveiller les deux tâches T1 et T2. Si les signaux sont mémorisés il peut y avoir de multiples réveils des tâche à chaque interruption matérielle reçue.

2) Synchronisation par variable d'événement

Une variable d'événement est un booléen qui peut valoir soit « 0 » soit « 1 ».
Le mécanisme est analogue à celui décrit ci-dessus.

Dans certains noyaux les événements peuvent être des variables codées sur 16 ou 32 bits. Les primitives peuvent tenir compte de plusieurs bits en utilisant les techniques de masquage.

Exemple une variable d'événement peut correspondre à la combinaison suivante :

xxx1 0xxx 11xx xxxx

Il est aussi possible de combiner plusieurs variables d'événement.

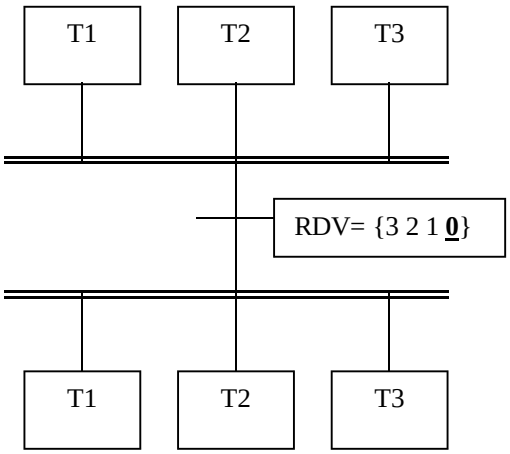
3) Synchronisation par rendez-vous

Dans les exemples précédents une interruption matérielle ou une tâche avait pour rôle des réveiller les autres tâches.

Si aucune tâche n'a ce rôle précis il n'est plus possible d'utiliser les méthodes précédentes car il est impossible de savoir à l'avance l'ordre d'arrivée au point de synchronisation.

La méthode la plus générale consiste alors à créer une variable globale initialisée au nombre de tâches qui doivent de synchroniser.

Lorsque qu'une tâche arrive au rendez-vous elle décrémente de une unité la variable correspondante. Lorsque cette variable vaut **«0»** cela veut dire que tout le monde est arrivé : on peut alors réveiller toutes les tâches.



CONCLUSIONS

- **Les sytèmes de contrôle informatiques utilisent la plupart du temps le principe du multitâche.**
- **Chaque système est construit autour d'un NOYAU et de ses primitives.**
- **Le mécanisme de base de gestion est celui des INTERRUPTIONS, soit matérielles, soit logicielles.**
- **Le noyau peut être temps réel ou non. Les systèmes d'exploitation de type UNIX ou autre ne sont pas temps réel, mais possèdent des extensions temps réel.**
- **Le développement d'une application temps réel nécessite une ANALYSE précise des contraintes logiques et temporelles du processus**.