Conception et Réalisation d'un Exécutif Temps Réel

Comment Concevoir et Réaliser un Noyau Temps Réel ?
Mémoire d'un projet de fin d'études.
Université Mouloud MAMMERI de Tizi-Ouzou.
Présenté par M. Toufik SARNI.
Dirigé par Mme Samia Bouzefrane (CNAM – Paris) & M. Lounes Djema (UMMTO)

Le secteur industriel (militaire, aéronautique, télécommunications…) manifeste, une demande de plus en plus accrue de systèmes temps réel complexes et fiables. Ces derniers s'efforcent, à travers des théories mathématiques avancées, de solutions informatiques plus élaborées et, de contraintes temporelles rigoureuses, de subvenir aux besoins industriels.

Poussés par la curiosité de connaître le fonctionnement de tels systèmes et, par leur opportunité, nous essayerons de concevoir et de réaliser un exécutif temps réel simple, en illustrant et en mettant en relief tout au long de ce mémoire qui accompagnera ce travail, toutes les étapes nécessaires.

La philosophie qui sera adoptée, se basera, elle aussi sur la simplicité des mécanismes qui seront mis en œuvre. Ceci afin de permettre, une meilleure compréhension de principes et politiques utilisés d'une part, et d'autre part, offrir un noyau, simple à modéliser, en vue de lui intégrer un certain aspect pédagogique. Justifiant ainsi, le choix de la création au lieu d'une modification ou amélioration d'un système déjà existant.

Ce mémoire présente notre travail en deux chapitres, les deux autres, seront consacrés à des rappels et présentation des systèmes existants.

Le premier chapitre rappelle, les concepts utilisés dans les systèmes temps réel.
Pour cela, il introduit des définitions, et aborde la notion de multitâche et les principes qui en découlent liés au problème de synchronisation et d'ordonnancement temps réel.
Ce chapitre soulève, les principales différences entre les systèmes classiques et les systèmes temps réel.

Le deuxième chapitre complète les concepts présentés au premier chapitre, en décrivant des plates-formes et des systèmes temps réel utilisés dans le milieu industriel, et ce, d'une façon générale. Il aborde les différents éléments les caractérisant, tout en rappelant brièvement leurs diverses évolutions.
Les systèmes comme RTAI (Linux temps réel), VxWorks (Unix temps réel), et les plates-formes CLEOPATRE, BOSSA.

Le troisième chapitre présente, d'une façon exhaustive la conception de notre travail. Il se propose alors de donner une vue détaillée de notre système, et présente entre autres, les différents modules à développer pour notre exécutif temps réel, en se référant durant chaque passe à l'objectif de notre travail.

Le quatrième chapitre concrétise le précédent, en s'accentuant sur la réalisation, et définit nos motivations en matière d'implantations, notamment l'environnement informatique choisi (architecture matérielle, environnement logiciel et de développement). Comme il est tout aussi question dans ce chapitre, de reprendre quelques fragments de code, les analyser et d'éclaircir leurs fonctions.

Chapitre **1**

Les concepts temps réel

Ce chapitre introduit des généralités, définitions et, des notions, sur les systèmes informatiques temps réel qui serviront de base pour aborder les prochains chapitres.
Nous avons jugé utile, d'introduire en premier lieu, un historique, à travers lequel, nous présenterons un bref aperçu, des dates ou époques, qui ont marqué l'évolution des systèmes informatiques temps réel.

1.1. Historique [DP 91]

La première apparition du terme **temps réel** tel que nous l'entendons aujourd'hui, coïncide à peu de choses près avec l'avènement des microprocesseurs et le développement du traitement numérique de l'information, c'est-à-dire bel et bien au début des années 1970. Les "puces" ont entraîné l'intégration de l'ordinateur dans l'environnement industriel : le calculateur industriel était né.
La réalisation d'applications soumises à de fortes contraintes temporelles devenait possible, le délai de réponse exigible variant considérablement en fonction de la dynamique caractéristique de chaque processus et des marges de tolérance des paramètres contrôlés.
La mise en place d'une hiérarchie de systèmes en temps réel au moyen de micro-ordinateurs permet d'éviter les problèmes techniques très complexes que soulèverait la mise en œuvre d'un système global équivalent sur un ordinateur de taille importante.
Il est cependant juste de constater que le développement de ces calculateurs industriels et, d'une manière plus générale, de l'informatique industrielle, ne date que du début des années 1980.
Avec l'avènement de ce que l'on appelle les "nouvelles technologies", l'objectif premier est de réaliser des traitements de plus en plus complexes le plus rapidement possible.
Le terme de **temps réel** n'est certes pas nouveau, il correspond toujours à une catégorie bien précise de traitements critiques dans le temps.

1.2. Qu'est-ce qu'un système temps réel ?

Pour comprendre l'idée des systèmes temps réel, nous avons préféré entreprendre la définition par des exemples [PAL 04], afin de donner quelques milieux concrets, sur lesquels s'ouvrent les systèmes temps réel et d'en comprendre les principes.

Pour cela, considérons le cas d'un avion de ligne, équipé d'un pilotage automatique. L'ordinateur de bord doit, calculer la trajectoire, en fonction des paramètres extérieurs (position de l'avion en hauteur, en latitude et longitude, vitesse…) et doit garantir périodiquement, des résultats fiables, et notamment, un temps de traitement **inférieur de loin à la seconde**.

A présent, considérons la situation suivante dans un aéroport. Un passager se rapprochant d'un distributeur automatique, afin de se procurer un billet d'avion, pour un certain vol Tizi-ouzou, Paris partant dans 5 mn. L'utilisateur fournit en entrée des informations (le départ, la destination, la classe …) à l'ordinateur. Celui-ci, après avoir accepté la requête, lui édite le billet **en quelques secondes**. Ce système, est-il un système temps réel ? .

En effet, dans les deux cas, les systèmes sont des systèmes temps réel, dans la mesure où les traitements des informations doivent, s'effectuer dans des délais bien limités, sous peine de conséquences pouvant être graves.

Ces exemples donnent, une idée intuitive quant à la définition des systèmes temps réel ; dans les prochains paragraphes, nous présenterons d'une façon plus formelle, les généralités et les définitions liées aux systèmes temps réel.

1.3. Présentation générale [SB 98]

Les systèmes temps réel sont, en général, constitués de deux parties: le système contrôlé, processus physique qui évolue avec le temps, appelé aussi *procédé* ou *partie opérative*, et le système contrôlant appelé aussi *partie commande* qui est un système informatique qui interagit avec le premier. Le système contrôlant prend régulièrement connaissance de l'état du procédé par le biais de capteurs, calcule la prochaine action à réaliser sur la base des dernières mesures puis applique une consigne au processus commandé par le biais d'actionneurs (voir figure 1.1).

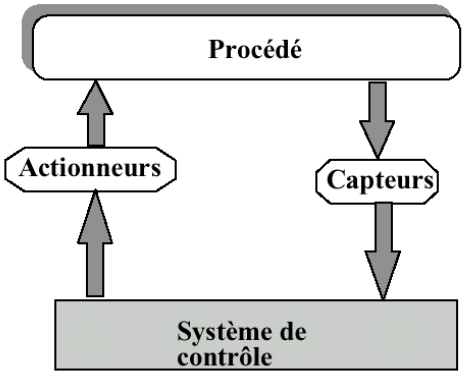


Figure 1.1. *Système temps réel*

Un système est qualifié de temps réel si sa correction ou sa validité ne dépend pas uniquement de la justesse[1] des résultats obtenus mais aussi des instants[2] de production de ces résultats, car les différentes activités d'un tel système doivent nécessairement s'exécuter dans des laps de temps limités appelé *échéance*s.

Le respect des contraintes temporelles est un aspect fondamental et spécifique aux systèmes temps réel, constituant la caractéristique principale qui les distingue des systèmes informatiques classiques (non temps réel).
D'une manière générale, nous distinguons trois types de systèmes temps réel:

- *les systèmes temps réel à contraintes strictes* qui sont des systèmes pour lesquels le non respect d'une échéance ne peut être toléré (exemple: la commande du moteur d'un avion) car cela peut engendrer une catastrophe ;

- *les systèmes temps réel à contraintes relative*s, par opposition, tolèrent les dépassements des échéances ;

- *les systèmes temps réel à contraintes mixtes* qui comprennent des programmes à contraintes strictes et d'autres à contraintes relatives.

Selon l'équipement contrôlé, les contraintes de temps peuvent être de divers ordres, de l'ordre:

- de la milliseconde: système radar,...

- de la seconde: système de visualisation,...

- de quelques minutes : chaîne de fabrication,...

---

1 et 2 : Dans la littérature, le terme **déterminisme** est utilisé pour exprimer ces deux notions.

1.4. Structure d'un système de contrôle [SB 98]

Le système de contrôle consiste en une structure matérielle munie d'un ensemble de logiciels permettant d'agir sur le système contrôlé. La structure matérielle correspond à l'ensemble des ressources physiques (processeurs, cartes d'entrée/sortie, ...) nécessaires au bon fonctionnement du système. L'architecture matérielle peut être:

- *monoprocesseu*r: tous les programmes s'exécutent sur un seul processeur en parallélisme apparent,

- *multiprocesseu*r: les programmes sont répartis sur plusieurs processeurs partageant une mémoire commune,

- *répart*i: les programmes sont répartis sur plusieurs processeurs dépourvus de mémoire commune et d'horloge commune. Ils sont reliés par un médium de communication par lequel ils peuvent communiquer par échange de messages.

La partie logicielle du système de contrôle comprend :

♦ d'une part, un système d'exploitation appelé *exécutif temps réel* chargé de fournir un ensemble de services (voir figure 1.2) que les programmes de l'application peuvent utiliser durant leur exécution. Il doit en particulier :

  * adopter une stratégie pour partager le temps processeur (notamment dans un système multitâches) entre les différentes activités en attente d'exécution en favorisant celles qui ont les délais critiques les plus proches ;

  * gérer l'accès aux ressources partagées ;
  * et offrir des mécanismes de synchronisation et de communication entre les activités du système de contrôle.

♦ d'autre part, afin de garantir un fonctionnement correct du procédé, des programmes sont implantés par l'utilisateur et traduisent les fonctions que doit réaliser le système de contrôle pour la prise en compte de l'état du procédé et sa commande. Ces programmes se présentent sous la forme d'un ensemble d'unités d'exécution appelées **tâches**.

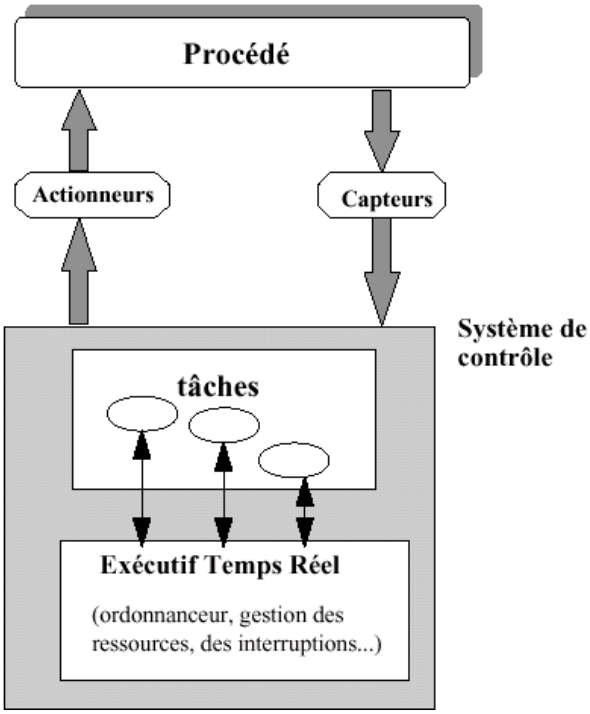


Figure 1.2. *Structure d'un noyau temps réel*

1.5. Exécutif temps réel

Un exécutif temps réel est en définitive, un noyau temps réel, pour lequel un certain nombre de services est ajouté (gestionnaire mémoire, gestionnaire d'entrées/sorties,…) et respectant des principes fondamentales, dont nous avons précédemment présenté les grandes lignes, et nous essayerons d'y apporter des détails dans les prochains paragraphes qui suivront. Cependant, ne nous intéresserons de prés qu'aux concepts qui seront mis en œuvre dans notre travail, et nous n'aborderons pas la gestion mémoire et entrées/sorties, étant donné que les techniques utilisées pour celles-ci, sont semblables à celles des systèmes classiques (non temps réel).

1.5.1. Principes de base du multitâche

Un système informatique est dit, en multitâche, s'il permet alternativement dans le temps, l'exécution de plusieurs *programmes* coexistant en mémoire, tout en assurant leur intégrité.

En vue d'éclaircir davantage le principe de multitâche, quelques définitions essentielles sont présentées dans les sous paragraphes suivants :

1.5.1.1. Notion de processus[1]

C'est une entité dynamique qui naît, qui vit et qui meurt, par opposition à la notion de programme qui est une entité statique qui occupe un espace en mémoire ou sur disque et qui, tant qu'elle n'est pas exécutée, ne produit aucune action [SB 03].
D'autre part, un programme peut être vu comme un ensemble de modules (sous-programmes) et l'exécution d'un module donne naissance à un processus [SB 03].

1.5.1.2. Contexte d'un processus [SB 03]

Un contexte sera représenté dans une structure regroupant les informations essentielles pour l'exécution du processus comme le contenu des registres, sa pile d'exécution son masque d'interruption, son état, son compteur ordinal, ses indicateurs, un pointeur vers sa table des pages, sa clé d'écriture en mémoire, etc.
Les différents états que peut prendre un processus sont les suivants (figure 1.3) :

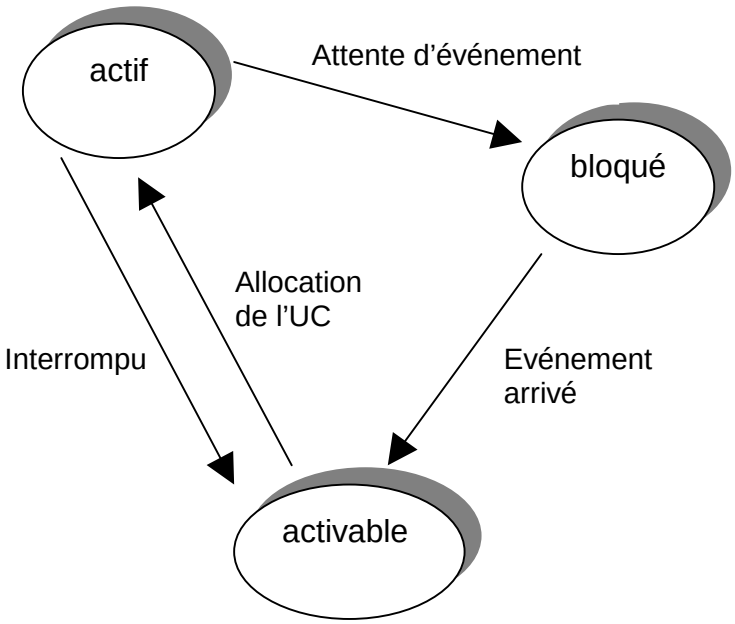


Figure 1.3. Etats des processus.

1. Les notions présentées pour les processus sont, également valable pour les tâches. Cependant, nous aurons tendance à mesure que nous progressions dans ce chapitre, d'utiliser le terme tâche, auquel nous définirons par la suite, les spécificités, pour servir en sens le concept temps réel.

- état **bloqué** : le processus ne dispose pas de toutes les ressources nécessaires pour exécuter l'instruction suivante. Cela est dû au partage des ressources dont le nombre est limité ; avec l'hypothèse que les ressources que nécessite un processus ne lui sont pas toutes allouées à la fois, ce qui est le cas dans la plupart des systèmes ;

- état **actif** : le processus dispose de toutes les ressources nécessaires et il s'exécute ;

- état **activable** ou **prêt** : il dispose de toutes ses ressources, hormis l'unité centrale (CPU), occupée par un autre processus.

Le blocage est soit :

- *blocage matériel* : le blocage est dû à la demande d'une ressource alors que celle-ci n'est pas disponible ; il n'est pas prévu par le programmeur ;

- *blocage intrinsèque* : le blocage est prévu par le programmeur.

La transition de l'état activable à l'état actif se traduit par la modification du vecteur d'état du processus par un processus particulier, le processus du système chargé de l'ordonnancement des tâches appelé *ordonnanceur*, ou *scheduler* en anglais (voir paragraphe 1.5.2.1.).

La transition de l'état actif à l'état bloqué est provoquée par le processus lui-même ; le blocage est intrinsèque ou matériel .

La transition de l'état actif à l'état activable est provoquée par la prise en compte d'une interruption plus prioritaire que le niveau en cours ou bien par l'ordonnanceur, à la fin d'un quantum de temps dans un système temps partagé.

La transition de l'état bloqué à l'état activable est provoquée par un processus du système à la suite de l'arrivée d'un événement attendu par le processus.
Cette dernière transition ne s'accompagne pas automatiquement d'une commutation de contexte tandis que les trois premières sont toujours suivies d'une commutation de contexte. L'arrivée d'un événement est signalé par une interruption. Par conséquent, prendre en compte un événement revient à traiter l'interruption correspondante.

L'ensemble des processus du système est représenté en général par une liste chaînée dont chaque élément est un *bloc de contrôle de processus* et pourrait avoir la structure décrite figure 1.4.

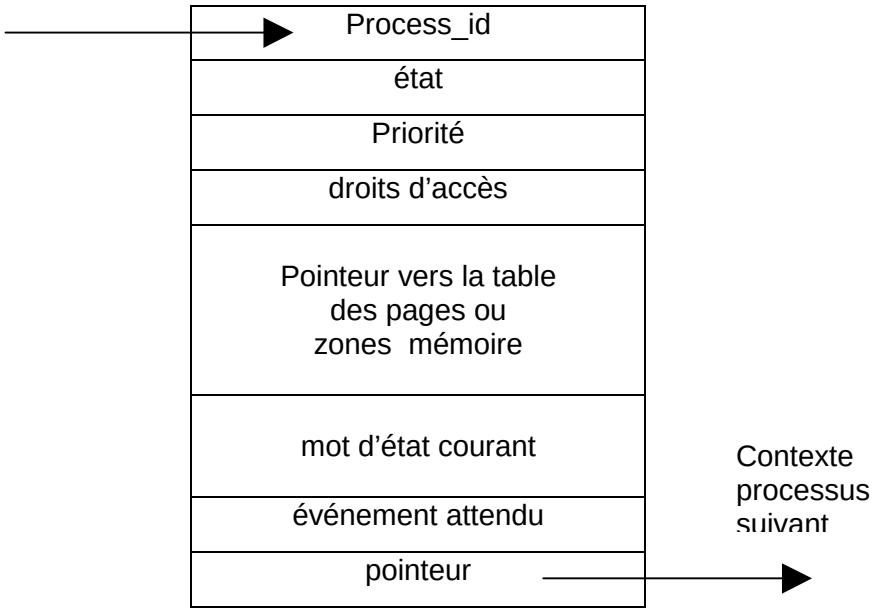


Figure 1.4. Contexte d'un processus.

1.5.1.3. Zones mémoire d'un processus

D'une façon générale, un processus est composé de trois zones mémoire, comme le montre la figure 1.5.

- *Zone programme :* contient le code exécutable du processus ;
- *Zone de données :* les données initialisées, les données non initialisées et les données statiques (externes, globales ou déclarées comme statiques) résident dans cette zone ;
- *Zone de pile :* nous y trouvons généralement des informations temporaires (variables locales, adresses de retour, paramètres …).

L'allocation de ces zones est assurée par l'exécutif, qui dispose d'un gestionnaire mémoire lui permettant lors du lancement d'un processus, d'attribuer à celui-ci, des pointeurs contenant l'adresse du début de chacune des trois zones citées. Ces pointeurs feront partie du contexte du processus lancé.

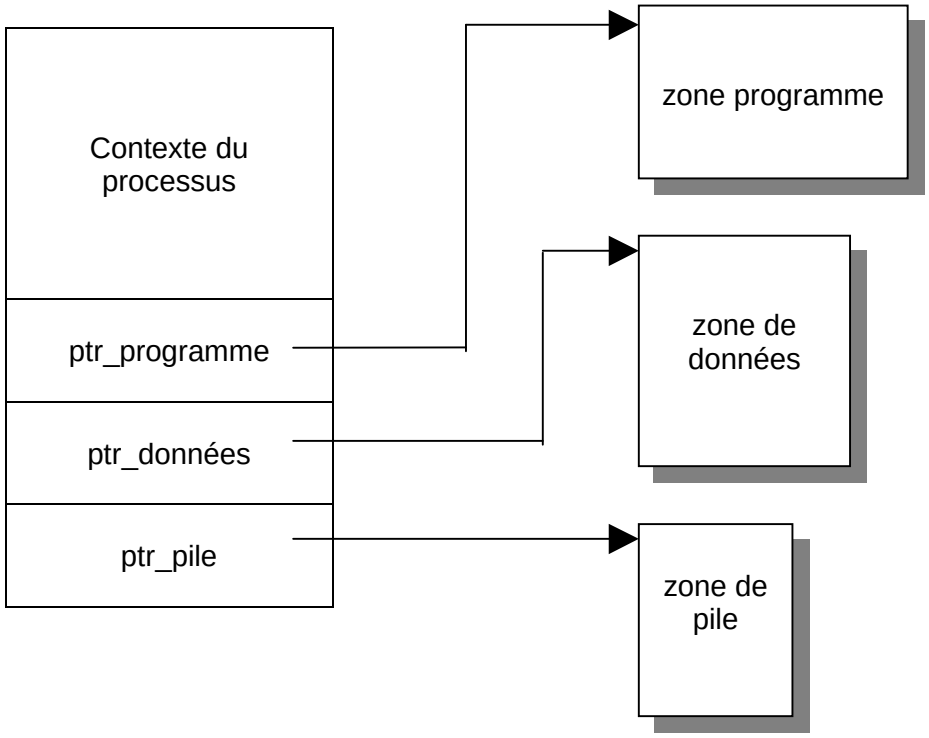


Figure 1.5. zones associées au processus.

1.5.1.4. Les appels au système

L'objectif de tout exécutif temps réel est, d'offrir un environnement dans lequel les processus peuvent, exploiter aisément et efficacement les ressources système.
En effet, il n'est pas commode, dire même fastidieux, de réécrire chaque fois et dans chaque programme les routines complexes et usuelles (exemple : faire une entrée sortie sur disque). Cet amas de code se voit alors réduire à une seule instruction d'appel de fonction.
Pour *faire un appel système*, le mécanisme ne se limite pas généralement à des procédures ou fonctions à implanter en mémoire.
L'exécution de ces fonctions se fait, dans l'espace mémoire de l'exécutif. La quasi-totalité des processeurs actuels disposent, d'instructions et des mécanismes pouvant assurer le bon passage du mode utilisateur (côté programme), au mode système (côté exécutif).
Généralement, une exception est produite lors d'un appel système, provoquant ainsi un déroutement vers un gestionnaire traitant, où sera analysé le type de fonction sollicitée par l'appelant, en tenant compte des paramètres récupérés depuis la pile ou les registres. Ces derniers contiendront au retour les résultats de l'appel système. (Voir figure 1.5).

Pour les exécutifs temps réel, et en particulier ceux des systèmes embarqués, le temps de réponse de tout appel système doit être connu et fourni par le concepteur de l'exécutif ; ce dernier doit entre autres borner ce temps, afin de respecter le déterminisme en temps.
Chaque fonction système (y compris les routines d'interruptions) se voit alors attribuée, un temps maximal de traitement, qui doit aussi être pris en compte par les développeurs d'applications temps réel.

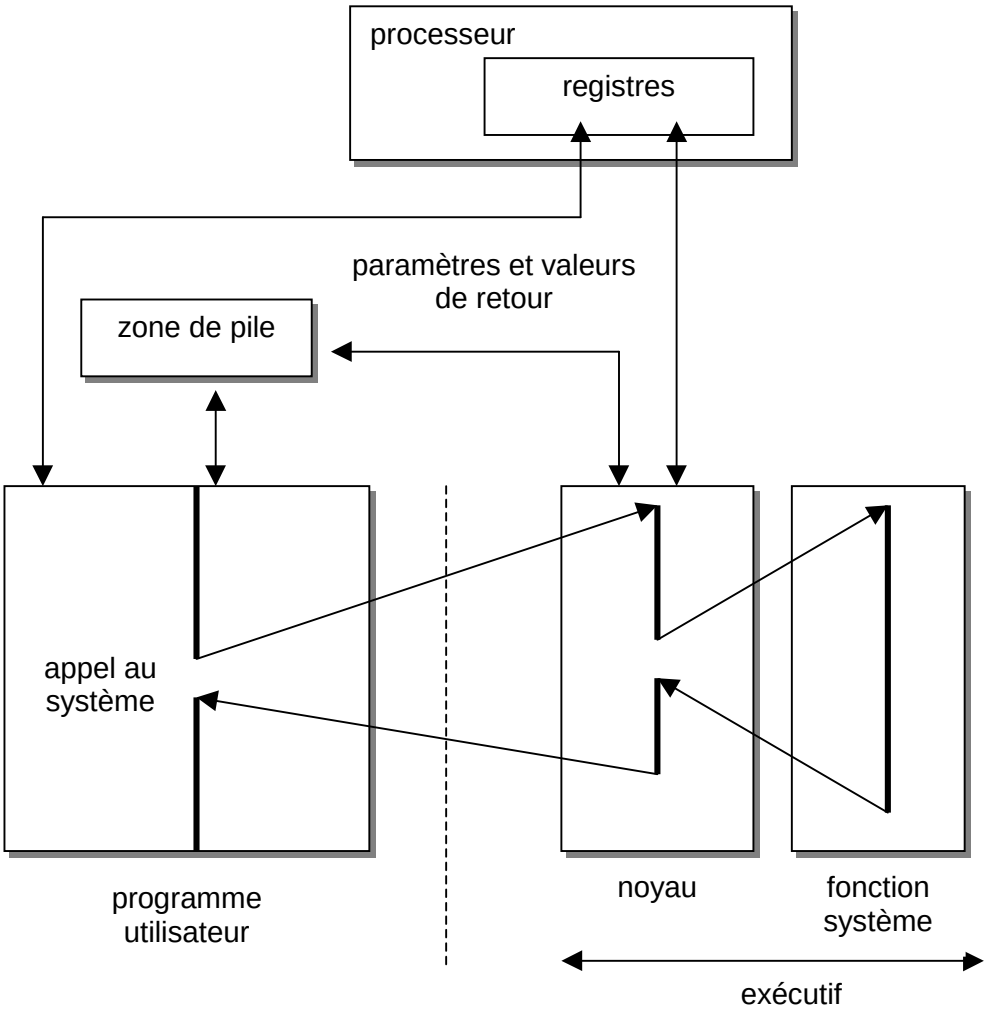


Figure 1.6. déroulement d'un appel système

1.5.1.5. Section critique

Un système multitâche est souvent concerné par un partage de ressources.
Dans la plupart des cas, ces ressources peuvent être utilisées que par un seul processus à la fois et, leur utilisation, ne peut être interrompue.
Tandis que la ressource CPU peut, contrer par elle-même la simultanéité d'accès, les procédures dans lesquelles le code est partageable en mémoire ne peuvent y pallier. Un tel code est dit, *section critique*.
Si deux ou plusieurs processus accèdent à la fois, à la section critique, une erreur fatale voire catastrophique se produira.

Exemple

Nous pouvons illustrer à travers un exemple [PAL 04], le problème lié à la section critique.
Considérons les programmes P1 et P2 définis, en langage C.

```
Void P1(void)
{

Printf ("Je suis P1");

}
```

Processus P1

```
Void P2(void)
{

Printf ("Je suis P2");

}
```

Processus P2

Supposons à présent le scénario d'exécution suivant :

- P1 effectue une entrée sortie, le message « Je suis » est affiché ;
- P1 est interrompu au profit de P2 ;
- P2 effectue une entrée sortie, et le message « Je suis » est affiché ;
- P1 reprend de nouveau la main suite à une interruption ;
- Le message « P1 » est affiché après une entrée sortie ;
- P1 finit l'exécution, et P2 la reprend de nouveau ;
- Le message « P2 » est affiché après une entrés sortie, puis P2 quitte le CPU.

La synthèse du scénario d'exécution de P1 et P2 conduit au résultat : « Je suis Je suis P1 P2 »
L'affichage est alors incorrect.

Le paragraphe suivant propose, une solution pratique au présent problème.

1.5.1.6. Synchronisation par sémaphores binaires

Proposé la première fois par Dijkstra, un sémaphore S est un emplacement mémoire, qui agit comme un verrou, pour protéger les sections critiques.

Le sémaphore peut être manipulé que par deux primitives P(s) et V(s) (indivisibles, ininterruptibles), elles signifient respectivement inhiber et autoriser.
Les primitives peuvent être définies en langage C comme suit : [PAL 04]

```
void P(int S)
{
  while (S ==TRUE);
  S=TRUE;
}
```

```
void V(int S)
{
  S=FALSE;
}
```

La primitive P(s) se bloque, tant que la valeur du sémaphore S est à TRUE.
V(s) positionne le sémaphore S à FALSE, permettant ainsi le déblocage au niveau de P(s).

Nous montrons ci dessous, l'utilisation pratique de ces primitives, par deux processus concurrents.

```
Processus_1
.
.
P(S)
Section critique
V(S)
```

```
Processus_2
.
.
P(S)
Section critique
V(S)
```

Le sémaphore S doit être initialisé à FALSE, pour que le premier processus demandant la section critique, puisse y accéder.

Nous pouvons appliquer le présent principe, pour résoudre le problème auquel nous sommes confrontés, dans le paragraphe précédent.
Il suffit alors d'ajouter les primitives P(S) et V(S) de la manière suivante :

```
Void P1(void)
{
 P(S);
  Printf ("Je suis P1");
 V(S);
}
```

Processus P1

```
Void P2(void)
{
 P(S);
  Printf ("Je suis P2");
 V(S);
}
```

Processus P2

Nous supposons comme déclaré global, le sémaphore S et initialisé à FLASE.
Que ce soit le processus 1 ou le processus 2 qui accède le premier à la section critique, ceci n'influe point l'exécution de l'autre.

En effet, le processus faisant appel à la primitive P(S), exclue l'autre. Ce n'est qu'une fois la primitive V(S) est exécutée que le processus exclu ou bloqué puisse reprendre l'exécution. Ce qui nous donnera toujours un affichage correct.

- Il existe une autre variante de sémaphores utilisée également par les exécutifs. Il s'agit des *sémaphores à compte*, destinés à la protection des ressources réentrantes mais pas forcement critique. Autrement dit, protéger une ressource dont le nombre maximum de processus pouvant y accéder simultanément est fixé, par la valeur du sémaphore.

Remarque :

Contrairement aux systèmes classiques, où on pose généralement la condition de Dijkstra suivante :
« Chaque processus en section critique doit, sortir au bout d'un temps fini. ».
Les exécutifs temps réel doivent quant à eux satisfaire cette condition, mais aussi déterminer ce temps de blocage, et ce, pour encore satisfaire la notion de déterminisme ; les prochains paragraphes mettront en relief les techniques utilisées et à mettre en œuvre dans notre travail.

1.5.2. Ordonnancement des tâches temps réel

1.5.2.1. Notion de tâches temps réel

Dans la littérature, le terme tâche est souvent préféré pour signifier, la périodicité ou la manière cyclique d'exécution, que ce, du processus dont l'aspect est attaché plutôt, au sens de l'unicité de la demande d'exécution.
Cependant, la tâche obéit parfaitement, aux principes du multitâche préalablement énoncés le cas échéant.

Les tâches doivent entre autres respecter des délais critiques imposés par les dynamiques de l'environnement à contrôler. D'un point de vue temporel, les paramètres suivants peuvent caractériser une tâche [SB 98] :

- *la date de réveil* : qui correspond à l'instant à partir duquel la tâche peut commencer son exécution ,
- *la durée d'exécution* : égale à la borne supérieure du temps processeur nécessaire à l'exécution de la tâche,
- *le délai critique* : qui est l'intervalle de temps, à partir de la date de réveil, durant lequel la tâche doit s'exécuter. En dehors de cet intervalle, son exécution devient inutile.
- *la période* : est l'intervalle de temps fixe qui sépare les arrivées successives d'une tâche. Ce paramètre concerne les tâches activées par un signal périodique provenant d'une horloge temps réel interne, c'est le cas par exemple de l'acquisition de données. Ce type de tâches est appelé *tâches périodique*s.

Les tâches dont la date de réveil n'est pas connue avant la mise en exploitation du système sont dites *tâches apériodique*s. En général, les tâches périodiques assurent le déroulement normal des fonctions de commande du procédé alors que les tâches apériodiques traitent les alarmes et les états d'exception [SB 98].

1.5.1.2. Problématique

Au paragraphe 1.5.1.5. (Section critique), nous avons présenté les ressources du système informatique et nous avons cité la ressource processeur, comme étant une ressource qui peut bien par elle-même, contrer le problème de la simultanéité d'accès. Réellement, cette constatation n'est valable, que si une considération du processeur comme système isolé, est faite. En effet, avant l'attribution du processeur à une seule tâche, celle-ci peut être résultat d'une sélection, parmi les tâches coexistant en mémoire, assurée par l'odonnanceur.

Les ordonnanceurs qui effectuent ce travail n'ont qu'une vision très générale des tâches. Ils ne considèrent pas les traitements qu'elles effectuent, mais utilisent, pour effectuer leur sélection, des paramètres définis à l'avance comme la périodicité de la tâche ou son échéance,... [LT 91].

La compétition pour l'accès à une ressource critique est un problème classique et est connu sous le nom d'exclusion mutuelle, pour lequel il existe une variété de solutions plus ou moins complexes. Dans le domaine du temps réel, il s'agit non seulement de garantir l'accès exclusif aux ressources critiques (qui est le rôle de l'exécutif temps réel) mais aussi et surtout de minimiser le temps d'attente pour l'accès à ces ressources. Il existe des protocoles dits *d'allocation de ressources* intégrés aux algorithmes d'ordonnancement et capables de réduire le temps d'attente des tâches bloquées pour l'obtention de ressources afin que celles-ci ne dépassent pas leurs échéances [SB 98].

Sachant que les algorithmes d'ordonnancement s'appliquent aux configurations de tâches périodiques, nous pourrons appréhender les tâches apériodiques avec ces algorithmes si nous les transformons, par exemple, en tâches périodiques particulières appelées *serveur*s [SB 98].

1.5.1.3. Algorithmes d'ordonnancement [SB 98]

Plusieurs critères peuvent définir un algorithme d'ordonnancement :

1.5.1.3.1. Préemptif / non préemptif

Un algorithme d'ordonnancement est :

- *non préemptif*: si toute tâche qui commence son exécution doit être achevée avant qu'une autre tâche obtienne le processeur ;

- *préemptif*: si une tâche peut être interrompue au profit d'une autre et être reprise ultérieurement.

Les algorithmes non préemptifs sont faciles à mettre en œuvre ; de plus, ils ont un faible coût puisque les changements de contexte sont minimisés et il n'y a pas lieu de gérer l'accès concurrent aux ressources critiques. Cependant les algorithmes préemptifs sont plus efficaces puisqu'ils offrent une exécution parallèle des tâches en ayant plus de liberté pour choisir une solution d'ordonnancement.

1.5.1.3.2. Hors ligne / en ligne

Un algorithme d'ordonnancement est dit :

- *hors ligne* : lorsque toutes les tâches ainsi que leurs caractéristiques temporelles sont connues avant le démarrage de l'application. La construction d'une séquence[1] d'exécution des tâches respectant les contraintes temporelles est possible et l'ordonnanceur se réduit à une entité appelé *répartiteur* qui ne fait que lancer les tâches dans l'ordre dicté par la séquence. Il est évident qu'une telle approche ne peut prendre en compte des tâches dont la date d'activation n'est pas connue au départ.

- *en ligne* : lorsque la séquence d'exécution est construite au fur et à mesure uniquement avec les tâches prêtes à l'instant courant ; ce qui permet la prise en compte et l'exécution des tâches apériodiques.

1. Une séquence d'exécution est dite valide si toutes les tâches se sont exécutées dans le respect de leurs contraintes temporelles et de ressources.

1.5.1.3.3. Conduit par la priorité

L'ordonnanceur tel qu'il a été défini précédemment est l'une des fonctions principales de l'exécutif temps réel. Il est chargé d'attribuer le processeur à chaque tâche avant son échéance en utilisant un algorithme d'ordonnancement. On dit qu'un algorithme d'ordonnancement est conduit par la priorité s'il exécute à chaque instant la tâche la plus prioritaire. La priorité est ou bien affectée par le concepteur de l'application en fonction de la criticité des tâches ou encore assignée de manière automatique en fonction d'un paramètre temporel caractérisant les tâches telles que la période ou l'échéance qui reflètent l'urgence et non l'importance de la tâche au sens applicatif.

Une classification de ces algorithmes est basée sur le type de la priorité instaurée entre les tâches, qui peut être *fixe* ou *dynamiqu*e. Un algorithme à priorité fixe affecte, à chaque tâche, une priorité qui reste constante durant l'exécution de cette dernière. Un algorithme à priorité dynamique recalcule la priorité des tâches durant leur exécution.

1.5.1.4. Algorithmes d'ordonnancement à priorités fixes sans partage de ressources

Avant de détailler ce présent paragraphe, nous avons jugé nécessaire, d'introduire une modélisation sur laquelle nous nous appuierons, pour présenter les notions et définitions attachées. Nous mettrons l'accent que sur l'algorithme RM, qui sera évoqué lors des prochains chapitres. Pour le lecteur intéressé par d'autres variantes, il peut consulter les références suivantes : [SB 98, SB03].

1.5.1.4.1. Modélisation d'une tâche

Nous pouvons représenter du point de vue temporel, une tâche périodique T_i par les paramètres suivants [SB 98] (voir figure 1.7.) :

- ***r****i* : la date de première activation,
- ***C i*** : la durée maximale d'exécution,
- ***Di*** : le délai critique maximal que T_i est tenue de respecter,
- ***Pi*** : la période qui est l'intervalle de temps maximal séparant deux instances successives de T_i.

Nous avons nécessairement les relations suivantes: $0 \leq C\,i \leq P\,i \leq D\,i$

Remarque :

Lorsque la période *Pi* est égale au délai critique *Di* la tâche est dite à *échéance sur requêt*e.

Étant réveillée aléatoirement, une tâche apériodique est caractérisée par le triplet

(*ri , Ci ,Di*) où:

- ***ri*** : la date initiale de réveil de la tâche,
- ***C i*** : la durée maximale d'exécution,
- ***D i*** : son délai critique

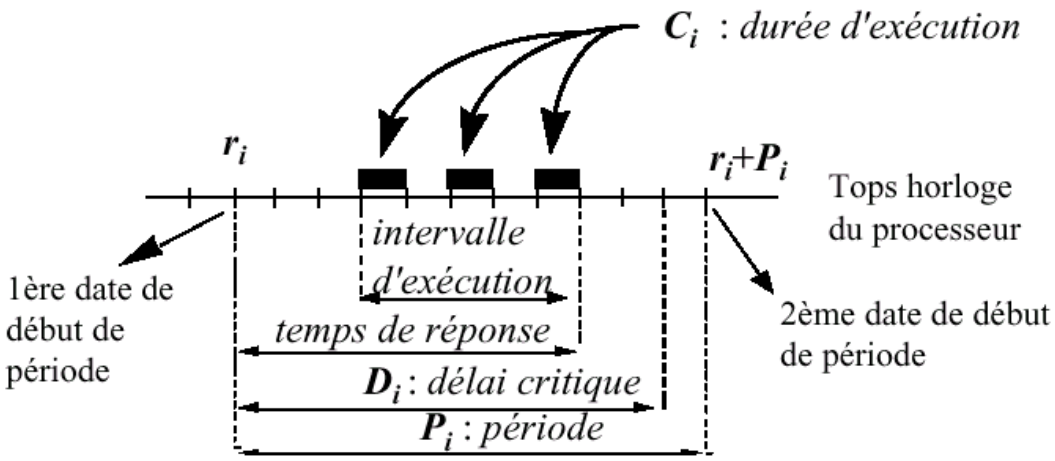


Figure 1.7. Les caractéristiques temporelles d'une tâche périodique [SB 98]

1.5.1.1.2. Condition nécessaire de validation

Pour une configuration donnée, nous définissons le facteur d'utilisation *U* comme étant le pourcentage d'utilisation du processeur par les tâches périodiques: $U = \sum(Ci / Pi)$

Il n'est évidemment pas possible d'utiliser le processeur au-delà de ses capacités c'est-à-dire dépasser un taux d'occupation de 100%: c'est la condition nécessaire pour une configuration de tâches périodiques : $U \leq 1$

1.5.1.1.3. Algorithme Rate Monotonic (RM)

Liu et Layland dans [LL 73] ont proposé, l'algorithme du Rate Monotonic, qui est à la base des théories sur l'ordonnancement des tâches périodiques. Il peut être utilisé, pour ordonnancer des tâches sur un monoprocesseur. L'algorithme repose sur le principe suivant :
La priorité d'une tâche est inversement proportionnelle à sa période. Autrement dit, la grande priorité est associée à la tâche de plus petite période. Et pour le cas des tâches de même période, leur priorité est fixée aléatoirement.
Notons que cet algorithme est, optimal pour les tâches à échéance sur requête.

Théorème (Rate Monotonic) [LL 73] :

Toute configuration de n tâches périodiques à échéance sur requête est fiablement ordonnancée par RM si :

$$\sum_{i=1}^{n} (Ci \ / \ Pi) \leq n\,(2^{1/n} - 1)$$

* La condition est suffisante mais pas nécessaire.
* Lorsque *n* est grand : $n\,(2^{1/n} - 1)$ converge vers *Ln2=0.69*

Exemple 1 [SB 03] :

Soit l'exemple de deux tâches A et B dont les caractéristiques sont données au tableau 1.1. La priorité de A est supérieure à celle de B car la période de A est inférieure à celle de B. Si on déroule la séquence d'exécution, on s'aperçoit que la séquence est valide (Figure 1.8)

(La flèche montante symbolise le réveil de la tâche, et la flèche descendante symbolise la fin de son échéance).

Tâches	Date de réveil	Durée d'exécution	Délai critique	Période
A	0	2	6	6
B	0	3	5	8

Tableau 1.1. Caractéristiques temporelles des tâches

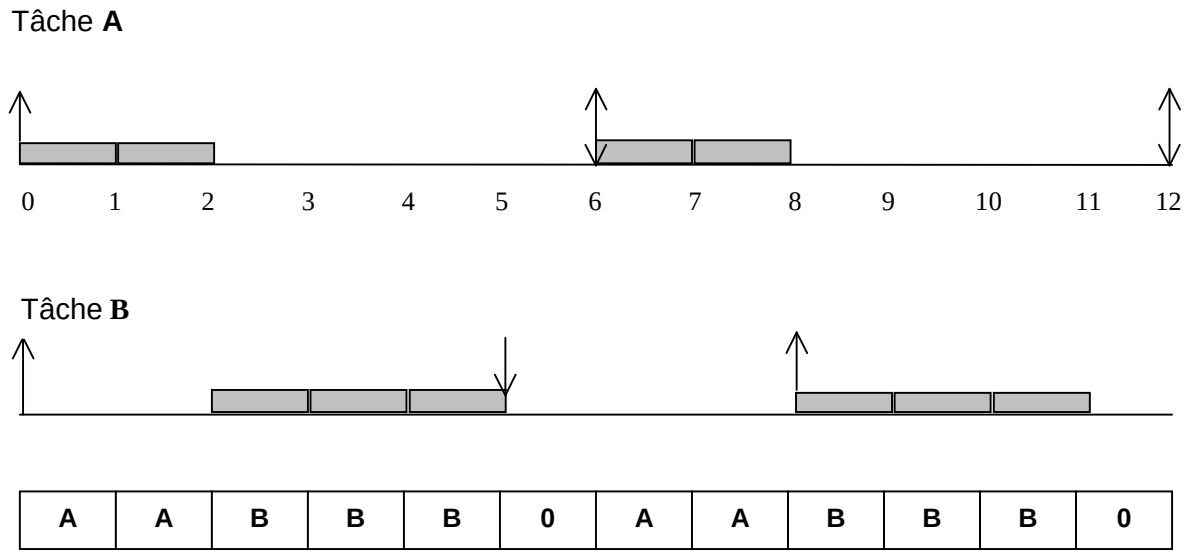


Figure 1.8. Séquence valide obtenue avec RM pour la configuration du tableau 1.1.

Nous pouvons vérifier facilement, le test d'ordonnançabilité. Dans le cas où la condition n'étant pas vérifiée, nous pourrions rien conclure quant à la possibilité d'ordonnancement, car même dans ce cas-ci, l'ordonnancement peut être valide (la condition est suffisante mais pas nécessaire).

Exemple 2 :

Soient les tâches A,B, et C, dont le triplet les caractérisant est respectivement :
(r=0, C=3, P=20) ; (r=0, C=2, P=5) ; (r=0,C=2,P=10).

Suivant l'algorithme RM, priorité(B) > priorité(C) > priorité(A).
En appliquant la condition suffisante d'ordonnançabilité :

$$\sum_{i=1}^{3} (Ci \ / \ Pi) = 3/20 + 2/5 + 2/10 = 0.75 \leq n(2^{1/n} - 1) = 0.77$$

La condition est vérifiée. Le diagramme de Gantt correspondant est donné à la figure 1.9.

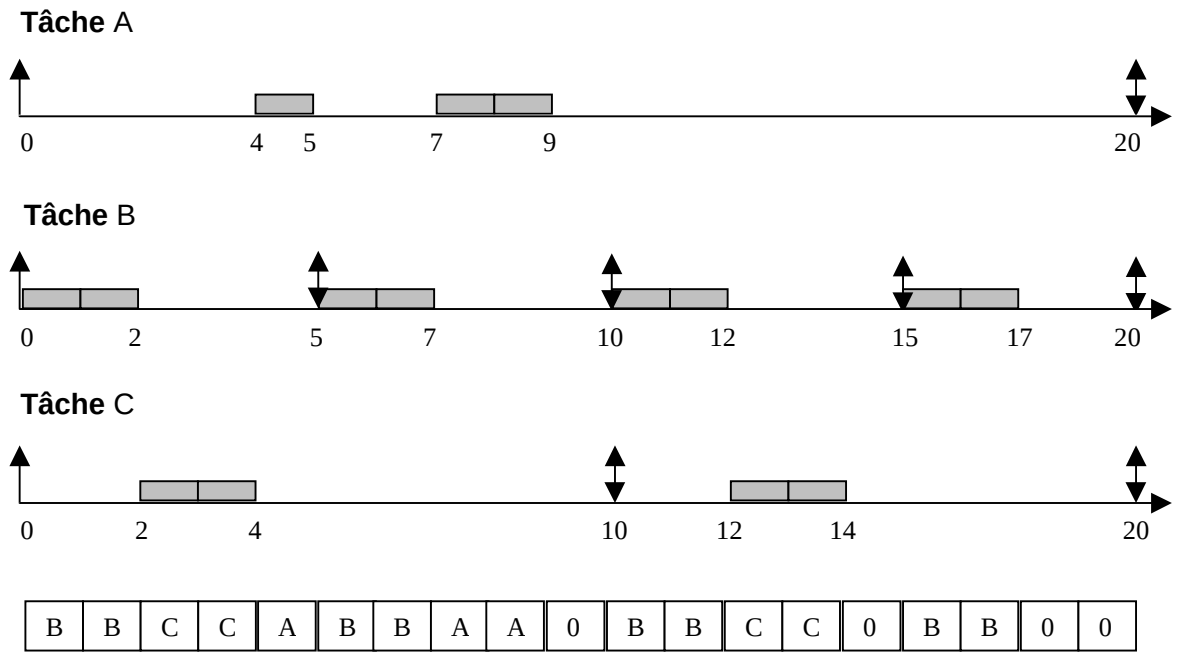


Figure 1.9. séquence d'exécution valide avec RM

1.5.1.1.4. Algorithme Deadline Monotonic (DM)

Il associe la plus forte priorité à la tâche de plus petit délai critique.
La condition suffisante d'ordonnançabilité pour cet algorithme est : $\sum(Ci / Di) < 1$

1.5.1.5. Ordonnancement des tâches apériodiques [SB 98]

Dans le contrôle de procédé, des événements aléatoires en provenance du procédé ou des événements internes sans aucune synchronisation avec le système peuvent survenir à tout instant. Celui-ci doit reconnaître la nature d'un événement et déclencher l'exécution du service lié à cet événement. La prise en compte d'un tel événement se fait grâce à une tâche apériodique. Le problème est alors d'ordonnancer ce type de tâches.

Une première solution consiste à ordonnancer les tâches apériodiques, en second plan, au sein de la configuration des tâches périodiques. On commence par ordonnancer l'ensemble des tâches périodiques afin de déduire les temps creux où seront insérées les tâches apériodiques.

Une autre solution consiste à ordonnancer les tâches apériodiques sur le même plan que les tâches périodiques. Une tâche périodique appelée *serveur* est dédiée à l'ordonnancement des tâches apériodiques.

Les caractéristiques temporelles du serveur sont celles d'une tâche classique:

- *rs* : la date de réveil,
- *Cs* : la durée maximale d'exécution réservée à l'exécution des traitements non périodiques,
- *P*s: la période évaluée de sorte à prendre en compte l'ensemble des requêtes non périodiques.

calculées en fonction des caractéristiques des tâches apériodiques à servir.

1.5.1.5.1. Serveur à scrutation [SB 98]

Si aucune requête non périodique n'a été mémorisée avant la date d'activation du serveur, celui-ci aura *un temps d'exécution nul* pendant la période courante (voir figure 1.10).

Si, par contre, une requête a été mémorisée, le serveur utilisera son temps d'exécution *Cs* à sa prochaine activation pour exécuter le traitement associé. Si le traitement n'est pas terminé sur une période, il sera poursuivi pendant la (les) période(s) suivante(s). Le temps de réponse (temps écoulé entre la date de réveil et la date de fin d'exécution) des tâches apériodiques peut être important surtout si elles surviennent juste après la fin d'exécution du serveur. Au niveau de l'analyse de l'ordonnançabilité, ce serveur est considéré comme une tâche périodique.

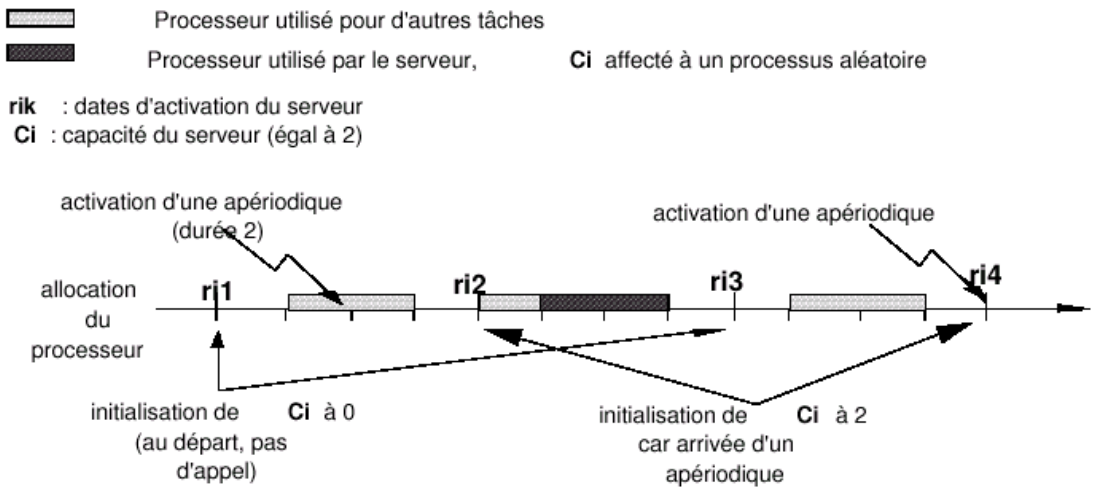


Figure 1.10 Serveur à scrutation

1.5.1.5.2. Serveur ajournable [SB 98] :

Pour remédier à l'incompatibilité des rythmes d'arrivée, ce serveur maintient son temps d'exécution même si aucune requête non périodique n'est survenue (voir figure 1.11). Dans le cas où une requête survient, le serveur exécute le traitement associé dans la mesure où son temps n'est pas épuisé ; il n'attend pas la prochaine activation pour la servir.

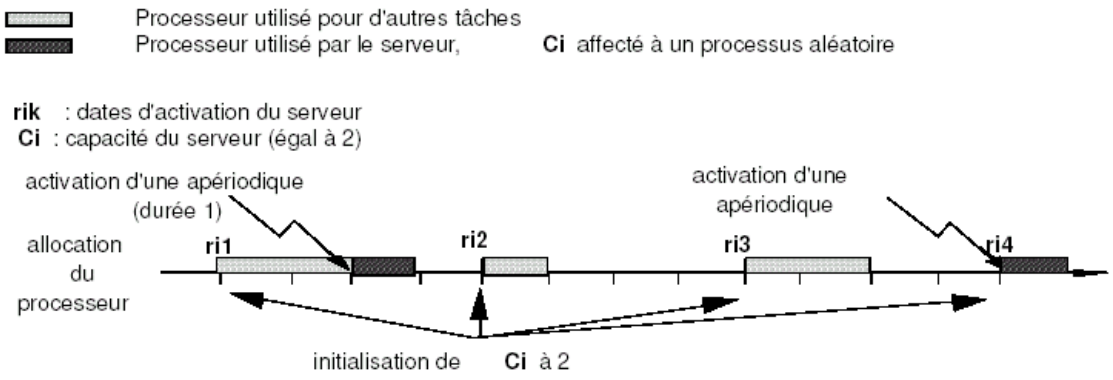


Figure 1.11 Serveur ajournable

1.5.1.6. Algorithmes à priorités variables sans partage de ressources :

Comme préalablement énoncé, pour ce type d'algorithmes, les tâches ont une priorité, qui évolue durant la vie de l'application. Nous citerons les deux principaux existants.

1.5.1.6.1. Algorithme Earliest Deadline (ED)

Les tâches sont classées par ordre croissant de leur échéance. Il est optimal, pour un système sans partage de ressources.
La condition suffisante d'ordonnançabilité pour les tâches périodiques et apériodiques indépendantes est la suivante : $\sum(Ci / Di) \leq 1$

Il existe, une condition nécessaire et suffisante d'ordonnançabilité, mais qui concerne que les tâches périodiques à échéance sur requête. Celles-ci doivent alors vérifier l'inégalité suivante : $U \leq 1$ (U représente le facteur d'utilisation du processeur).

1.5.1.6.2. Algorithme Least Laxity (LL)

Dans cet algorithme, la plus forte priorité est associée dynamiquement, à la tâche dans laquelle le temps, séparant la fin d'exécution et l'échéance, est le plus petit.

La condition nécessaire et suffisante d'ordonnançabilité pour les tâches périodiques à échéance sur requête est : $U \leq 1$ (U représente le facteur d'utilisation du processeur).

Remarque

Dans [Hen 75], il a été démontré, que ED utilise mieux le temps CPU, parmi tous les algorithmes cités précédemment (à priorités statiques et dynamiques).

1.5.1.7. Problèmes liés à l'exclusion

Comme nous l'avons souligné, l'exécutif temps réel doit garantir l'exclusion mutuelle, entre tâches, lors de l'accès à une ressource. Nous avons abordé une technique (sémaphores binaires) permettant la résolution pratique du problème lié à l'exclusion. Cependant, s'il y a exclusion mutuelle, l'approche préemptive basée sur les priorités conduit à des situations *d'interblocage* et *d'inversion de priorités*. Ces deux derniers sont illustrés, à travers des exemples, dans les sous paragraphes suivants :

1.5.1.7.1. Interblocage

Exemple :

Soit l'exemple de deux tâches périodiques A et B, se partageant deux ressources R1 et R2. Avec priorité(A) > priorité(B)

Admettant à présent les situations suivantes (voir figure 1.12.) :

1- La tâche B se réveille la première et demande la ressource R1.
2- R1 étant libre, la tâche B l'obtient.
3- La tâche A se réveille, préempte B et, demande la ressource R2.
4- R2 étant libre, la tâche A l'obtient.
5- La tâche A est dans la section critique de R2, demande la ressource R1.
6- La tâche A se bloque, et attend la libération de R1.
7- La tâche B reprend la main ; détenant R1, elle demande R2.
8- La tâche B se bloque, et attend la libération de R2.
9- Le système se bloque définitivement (étreinte fatale ou deadlock).

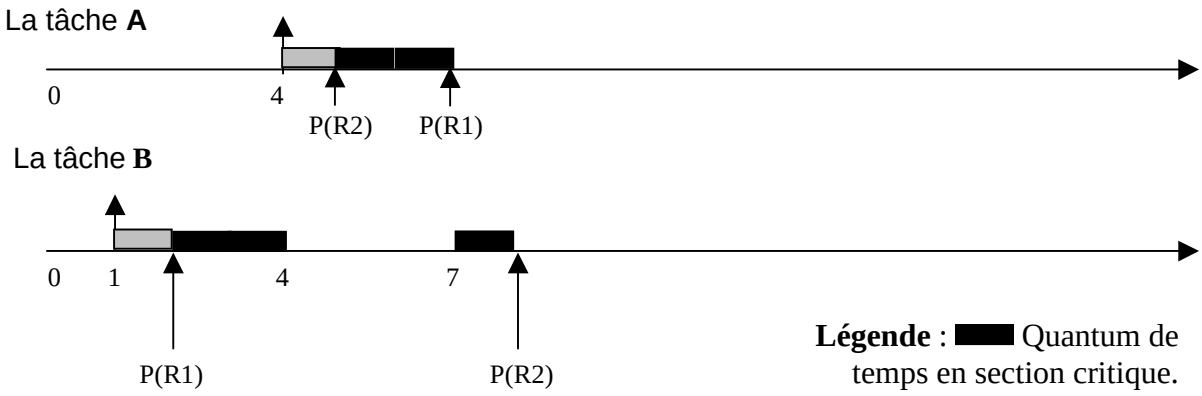


Figure 1.12. Situation d'interblocage.

1.5.1.1.2. Inversion de priorités

Exemple :
Considérons trois tâches périodiques A,B et C avec priorité(A) > priorité(B) > priorité(C), et se partagent une seule ressource critique S.

Les notations à adopter sur le diagramme de Gantt :
- La demande d'une ressource : (par exemple, par la tâche A) A/
- Attribution de la ressource à une tâche : (par exemple, à A) A
- Libération de la ressource : (obligatoirement par la tâche en section critique) /*

Admettant la séquence d'exécution de ces trois tâches, donnée par la figure 1.13.

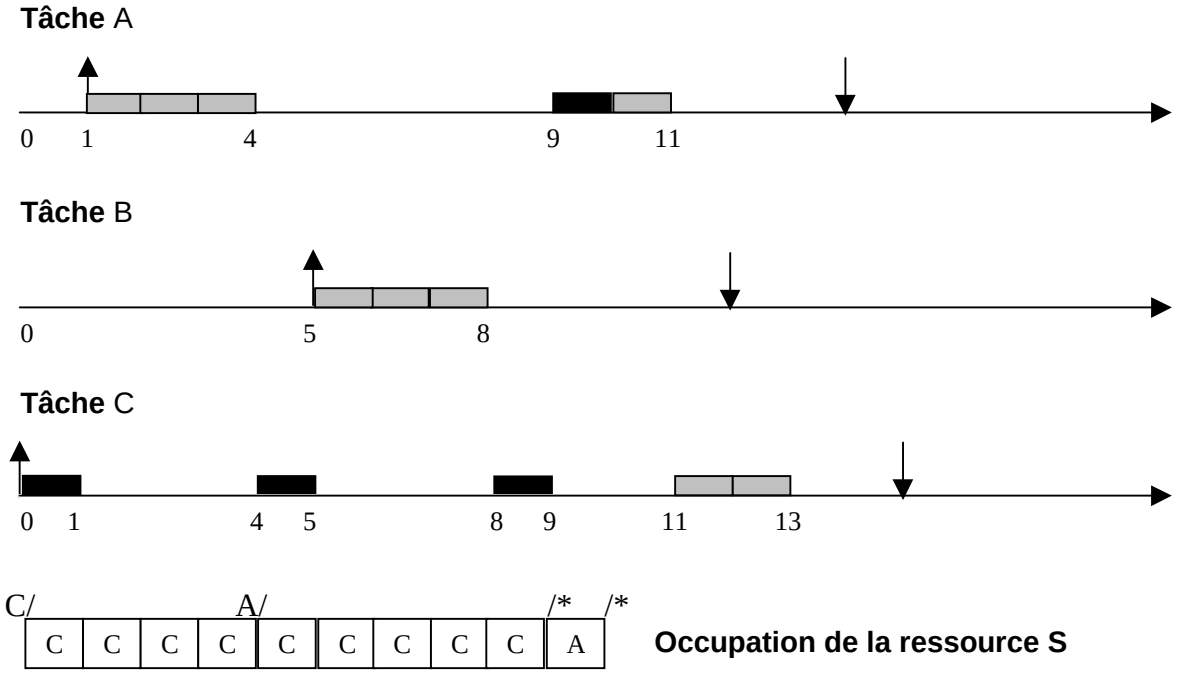


Figure 1.13 Exemple d'inversion de priorités

- La tâche C demande la ressource S à l'instant 0 et l'obtient.
- La tâche C est préemptée par la tâche A. (car priorité(A) > priorité(C))
- La tâche A demande à l'instant 4 la ressource S (détenue par la tâche C) et se bloque.
- La tâche C reprend la main, et elle est toujours en section critique.
- La tâche B préempte la tâche C (car priorité(B) > priorité(C)).
- La tâche C reprend l'exécution à l'instant 8 et libère la ressource S à l'instant 9.
- La tâche A obtient la ressource S à l'instant 9.

La tâche A possède la plus haute priorité dans le système considéré, pourtant, elle a été bloquée par des tâches moins prioritaires.

Ce phénomène auquel nous assistons est appelé *inversion de priorités*.

1.5.1.8. Protocoles d'allocation de ressources

Les solutions pour prévenir la situation d'interblocage et d'inversion de priorités existent, nous présenterons les deux protocoles qui seront mis en œuvre dans notre travail. Pour le lecteur intéressé par autres que ceux présentés, il peut consulter [SB 98].

1.5.1.8.1. Protocole à priorité héritée (PPH)

Le principe [SRL 90] étant de rehausser la priorité de la tâche obtenant la ressource critique, à celle de la tâche qu'elle bloque en attente de la même ressource.
Autrement dit, c'est attribuer à la tâche T entrante en section critique, une priorité égale au Max {Pr(T_1), Pr(T_2), Pr(T_3),…, Pr(T_n)}, sous condition que les tâches :
T_1 , T_2 , T_3 , ... T_n aient une priorité (Pr) supérieure à celle de T et attendent toutes la ressource détenue par T. (la priorité de T étant rétablie, au moment de la libération de la ressource qu'elle a occupée).
De cette manière, aucune tâche de priorité inférieure ne pourra, préempter la tâche T durant son exécution en section critique.

Exemple :

Nous appliquons le protocole à priorité héritée, à l'exemple donné précédemment dans le paragraphe 1.5.2.6.2. et nous aurons alors la figure 1.14.

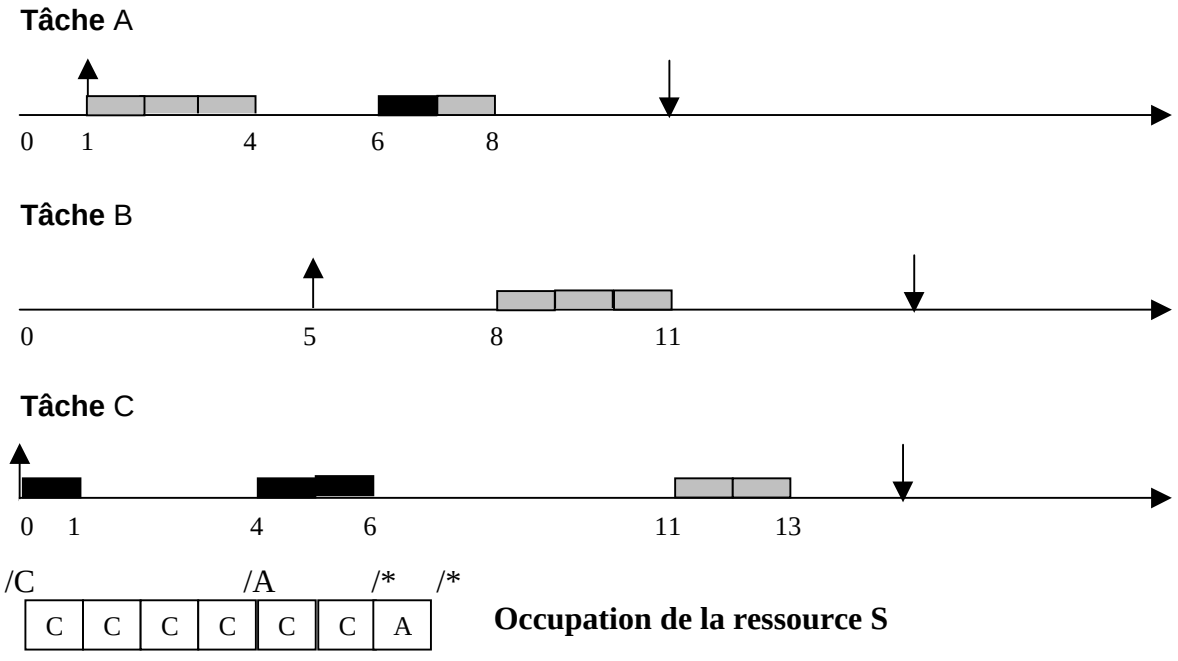


Figure 1.14 résolution d'inversion de priorité par PPH.

Nous constatons qu'à l'instant 4 (instant où la tâche A demande la ressource S), la tâche C hérite de la priorité de la tâche A.
C'est ainsi que la tâche B, n'a pu préempter la tâche C à l'instant 5 (Comme ce fut le cas dans la figure 1.13.). La tâche A entre alors en section critique à l'instant 6 (après libération de la ressource par la tâche C).

Aussi, nous pouvons remarquer le fait que la tâche C reprend sa priorité à l'instant 6 (justifié par le fait que la tâche C reprend son exécution qu'à partir du moment où la tâche B se termine. Autrement dit, à l'instant 6 nous avons de nouveau : Pr(B) > Pr(C)).

1.5.1.1.2. Protocole à priorité plafond (PCP)

En anglais *Priority Ceiling Protocol*. Ce protocole à été proposé [SRL 90], pour prévenir la situation d'interblocage que nous avons vu au paragraphe 1.5.2.6.1.
Il définit une nouvelle notion : *priorité plafond* (PP) d'une ressource, qui est une valeur maximum des priorités des tâches qui l'utilisent, en posant la condition suivante :
Une tâche T ne peut entrer en section critique, que si la priorité de la ressource R qu'elle demande, est strictement supérieure à la priorité plafond des ressources en cours d'utilisation (hormis celles de T). En section critique, la tâche devra hériter de la priorité de la tâche la plus prioritaire qu'elle bloque.
Ceci alors sous-entend que le protocole à priorité plafond, résout également le problème d'inversion de priorité.

Exemple :

Nous pouvons appliquer le protocole à priorité plafond, pour résoudre le problème d'inversion de priorité donné au paragraphe 1.5.2.6.2 ; vu le cas similaire présenté en appliquant l'héritage de priorité (figure 1.14), nous nous intéresserons alors qu'au problème d'interblocage du paragraphe 1.5.2.6.1.

Calculons la priorité plafond pour chacune des ressources R1 et R2 :
Pour appliquer le protocole à priorité plafond, ceci suppose que nous connaissions *à priori*, les ressources susceptibles d'être utilisées ; cela nous permettra alors de pouvoir calculer les priorités plafond de chacune des ressources dans le système considéré.

En se référant alors à l'exemple d'interblocage donné (paragraphe 1.5.2.7.1), nous pouvons écrire :

PP(R1)= Max {Pr(A), Pr(B)} = Pr(A).
PP(R2)= Max {Pr(A), Pr(B)} = Pr(A).

Le scénario d'exécution des deux tâches suivant le PCP serait alors le suivant
(voir figure 1.15) :

- La tâche B se réveil à l'instant 1 et s'exécute.
- La tâche B demande la ressource R1 (à t=2) qui est libre, et l'obtient (P(R1)).
- La tâche A se réveil à l'instant 4, préempte B (car Pr(A) > Pr(B)).
- La tâche A demande à l'instant 5 la ressource R2 qui est libre, mais ne l'obtient pas. (PP(R2) = Pr(A) n'est pas > PP(R1) ; elle ne satisfait pas la règle énoncée par le PCP).
- Puisque A se bloque, la tâche B reprend l'exécution et obtient R2 à l'instant 5.
- La tâche B à l'instant 5 hérite de la priorité de la tâche A.
- La tâche B demande R2 à l'instant 6 et l'obtient.
- La tâche B libère R1 et R2 à l'instant 8, et sa priorité est alors restaurée.
- La tâche A obtient la ressource R2 pour laquelle elle a été bloquée, puis R1 (à t=10).
- La tâche A libère la ressource R1 et R2 à l'instant 12.

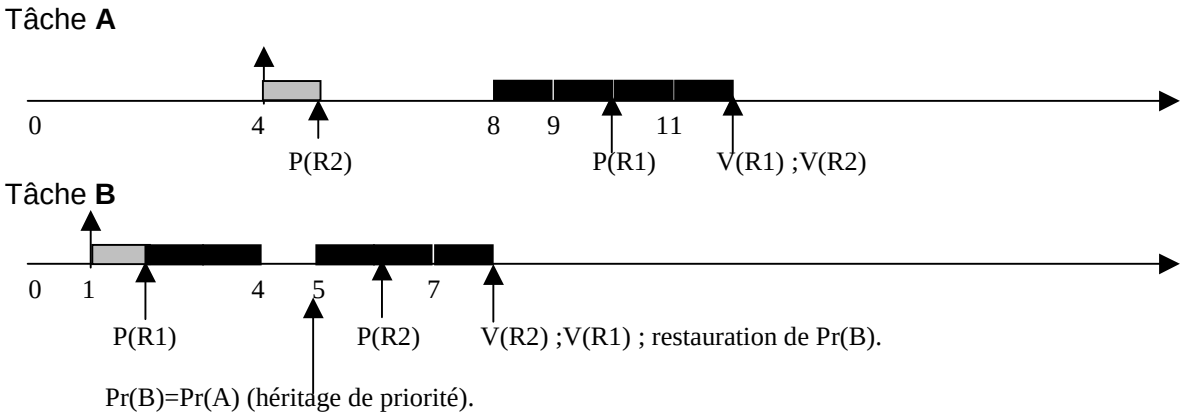


Figure 1.15. résolution de l'interblocage, avec le PCP.

Remarque :

Bien que le protocole à priorité héritée soit capable d'éviter l'inversion de priorité, il ne prévoit cependant pas l'interblocage. Il a été abordé que pour illustrer le phénomène d'inversion de priorité.
Nous allons discuter, de l'implémentation du protocole à priorité plafond dans les prochains chapitres, pour cela, nous allons nous focaliser dans ce qui suit, uniquement sur le PCP.

1.5.1.1.3. Facteur de blocage

Les solutions que nous avons vus au paragraphe précédent, permettent de borner le temps d'attente des tâches sur l'accès aux ressources. Ces bornes B, peuvent ensuite être ajoutées au temps d'exécution des tâches et ainsi être intégrées dans les tests d'acceptation des configurations.

Pour le protocole à priorité plafond, le temps maximal de blocage B est donné par l'algorithme suivant [TH 95] :

Début
Soit une tâche *T*:
1. Considérer les tâches moins prioritaires que *T*
2. Sélectionner celles qui possèdent des ressources communes avec *T*
3. Considérer uniquement les ressources dont la priorité plafond est ≥ priorité (*T*), soit *s* ce nombre
4. $B = \max_{i=1,s}$ (durée des sections critiques pour l'utilisation de la ressource c_i)

Fin.

Si *T* est la tâche la moins prioritaire, alors le temps de blocage est nul car il n'y a pas de tâche moins prioritaire qui peut la bloquer.

Exemple :

Soit l'exemple de cinq tâches, dont les caractéristiques sont données au tableau 1.2, se partageant les ressources c_1 et c_2 dont le temps des sections critiques correspondant est respectivement t_1 et t_2 (exprimé en µs).

Tâches	Durée d'exécution	Période	Priorité	t_1	t_2
A1	10	60	3	4	0
A2	8	200	1	4	0
A3	10	100	2	2	2
A4	9	60	3	0	3
A5	5	200	1	0	3

Tableau 1.2. Caractéristiques des tâches.

En utilisant le protocole à priorité plafond, calculons B_3 pour la tâche A_3.
Pour cela, supposons que les priorités des tâches respectent les règles de l'algorithme RM.
Calculons les priorités plafond des ressources c_1 et c_2.

$PP(c_1)$ = max priorités { A_1, A_2, A_3 }= 3.
$PP(c_2)$ = max priorités { A_4, A_5, A_6 }= 3.

Appliquons à présent, l'algorithme de calcul de la durée maximale de blocage.

⊗ Trouvons les tâches moins prioritaires que A_3 : A_2 et A_5.

⊗ Considérons parmi ces tâches, celles qui partagent des ressources avec A_3 : A_2 partage c_1 et A_5 partage c_2.

⊗ Prenons uniquement les ressources dont la priorité plafond est supérieure ou égale à la priorité de A_3 : c_1 et c_2 car $PP(c_1)=3$, $PP(c_2)=3$ et $Priorité(A_3)=2$.

⊗ $B_3 = \max (t_1(A_2) , t_2(A_5)) = \max (4 , 3) = 4$.

1.5.1.1.4. Algorithme RM avec partage de ressources

En tenant compte du temps maximal de blocage B, le test de faisabilité RM préalablement énoncé devient alors comme suit :

$$\sum_{k=1}^{i} (C_k / P_k) + (B_i / P_i) \leq i\,(2^{1/i} - 1) \quad \text{pour} \quad \forall\, i \leq n$$

Conclusion

Le premier chapitre nous a permis d'introduire d'une façon générale, les principes utilisés dans les exécutifs temps réel monoprocesseur, en se focalisant sur ceux qui nous servirons de discussion dans les prochains chapitres.

D'une façon assez exhaustive, ce chapitre donne des généralités d'un système temps réel, en mettant en évidence la notion temps et l'importance capitale à travers laquelle, nous pouvons voir la spécificité d'un tel système.

Nous avons abordé par la suite la notion d'exécutif temps réel, tout en rappelant quelques notions de base du multitâche (processus, contexte, les appels au système, la section critique), et nous avons intégré la notion *temps* au processus, ce qui nous a amené à définir la notion de *tâche*. Cette dernière, qu'elle soit périodique (de priorité fixe ou dynamique) ou apériodique, nous avons présenté dans chacun des cas, les techniques souvent utilisées et à utiliser dans notre travail, pour pouvoir les ordonnancer.

Pour présenter l'ordonnancement des tâches, nous les avons séparé en catégories, suivant le fait que l'ordonnancement se fait sur des tâches dépendantes ou indépendantes, au sens du partage de ressources. Pour ce dernier, nous avons passé en revue, les protocoles permettant l'évitement des problèmes liés à l'exclusion mutuelle, tout en donnant la possibilité de pouvoir calculer ou borner, la durée de blocage d'une tâche, en attente d'une ressource.

Le prochain chapitre va illustrer à travers des exemples concrets, quelques exécutifs temps réel présents et rencontrés dans le milieu industriel, en essayant de compléter et de mettre en évidence les techniques vues dans ce chapitre.

Bibliographie

[DP 91] : Alain DORSEUIL et Pascal PILLOT- *Le Temps Réel En Milieu Industriel* (DUNOD 1991).

[Hen 75] : R. Henn, *« Deterministic Modelle fur die Prozessorzuteilung in einer harten Realzeit-Umgebung* », Phd Thesis, Technical Univ. Munich, 1975.

[LL 73] : C. L. Liu, J. W. Layland - *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*- Journal of the ACM, Vol. 20, n°1, pp. 46-61, 1973.

[LT 91] : Lourent TOUTAIN - Thèse de doctorat à l'université du Havre en 1991.
Thème : SMASON : Un simulateur pour les systèmes répartis et temps-réel.

[PAL 04] : Phillip A.Laplante – *Real Time Systems Design And Analysis*. Third Edition. (A JOHN WILLEY & SONS 2004).

[SB 03] : Samia Bouzefrane - *Les Systèmes d'Exploitation Unix, Linux, et Windows XP* (DUNOD 2003).

[SB 98] : Samia Bouzefrane- Thèse de doctorat à l'université de Poirier en 1998.
Thème : *Étude temporelle des Applications Temps Réel Distribuées à Contraintes Strictes basée sur une Analyse d'Ordonnançabilité.*

[SRL 90]: L. Sha, R. Rajkumar, J. Lehoczky, *« Priority inheritance protocols: an approach to real-time synchronisation* », IEEE Transaction Computers, Vol. 39 n°9, pp. 1175-1185, 1990.

[TH 95] : K. Tindell, H. Hansson, "*Real-Time Systems and Fixed Priority Scheduling*", Report of Uppsala Univ., oct. 1995.

Chapitre **2**

Exemples de systèmes et plates-formes temps réel

Nous voulons à travers ce chapitre, compléter les concepts temps réel présentés au premier chapitre d'une part, et d'autre part, faciliter l'abord des prochains chapitres.
Les deux premiers exécutifs que nous aurons à présenter, couvriront respectivement d'une façon assez exhaustive les environnements Linux et Unix temps réel ; quant aux plates-formes temps réel, elles montreront un cas pratique d'un système embarqué, et le développement de nouvelles politiques d'ordonnancement notamment temps réel.

Nous avons alors choisi, les systèmes suivants :

- **Real-Time Application Interface (RTAI)** : variante de Linux temps réel, développé à Dipartimento di Ingeniera Aerospaziale, Politecnico di Milano du Prof. *Paolo Mantegazza*.

- **VxWorks :** variante de Unix temps réel, développé par Wind River Systems.

Ainsi que les plates-formes suivantes :

- **Composants Logiciels sur Etagères Ouverts Pour les Applications Temps- Réel Embarquées (CLEOPATRE) :** Projet lancé par plusieurs organismes français, en vue de fournir entre autres un RTOS[1] libre.

- **BOSSA :** plate-forme ajoutée au noyau Linux pour le développement d'ordonnanceurs de processus. (IUT de Nantes).

1. Real Time Operating System.

2.1. Real-Time Application Interface (RTAI)

2.1.1. Historique

Le projet RTAI a pour origine le département d'ingénierie aérospatiale (DIAPM) de l'Ecole polytechnique de Milan (Politecnico di Milano).
Pour des besoins internes, le professeur Paolo Montegazza du DIAPM entreprit de développer un produit inspiré de RTLinux[1] mais intégrant quelques améliorations et corrections, concernant en particulier les modes temps réel et, la gestion des nombres flottants. Contrairement à RTLinux, RTAI n'ai pas développé pour créer un produit commercial mais surtout afin d'éviter, des coûts de licences, sur des RTOS propriétaires comme QNX, alors utilisé au DIAPM.
Du fait de cette optique de développement interne, l'architecture actuelle de la distribution RTAI apparaît comme complexe au niveau de la définition de l'API (syntaxe et nombre d'appels). RTAI est rapidement apparu aux yeux des responsables du projet RTLinux comme un redoutable concurrent, sachant que RTLinux avait lui une vocation commerciale.

2.1.2. Linux comme système temps réel

Linux n'est pas nativement un système temps réel. Le noyau Linux est en effet conçu dans le but d'en faire un système généraliste donc basé sur la notion de temps partagé et non de temps réel.
La communauté Linux étant très active, plusieurs solutions techniques sont cependant disponibles pour améliorer le comportement du noyau afin qu'il soit compatible avec les contraintes d'un système temps réel. Concrètement, les solutions techniques disponibles sont divisées en deux familles :

1. Les patchs dits "préemptifs" permettant d'améliorer le comportement du noyau Linux en réduisant les temps de latence[2] de ce dernier. Ces modifications ne transforment pas Linux en noyau temps réel à contraintes "strictes" mais permettent d'obtenir des résultats satisfaisants dans le cas de contraintes temps réel "relatives". Cette technologie est disponible auprès de différents projets open source et la notion de noyau préemptif est intégrée dans le noyau de développement 2.5.

2. Le noyau temps réel auxiliaire. Les promoteurs de cette technologie considèrent que le noyau Linux ne sera jamais véritablement temps réel et ajoute donc à ce noyau un véritable ordonnanceur temps réel à priorités fixes. Ce noyau auxiliaire traite directement les tâches temps réel et délègue les autres tâches au noyau Linux, considéré comme la tâche de fond de plus faible priorité. Cette technique permet de mettre en place des systèmes temps réel "strictes". Cette technologie est utilisée par RTLinux et RTAI.

---

1. variante de Linux temps réel, développé à New Mexico Institute of Technologie par Mickaël Barabanov sous la direction du Prof. *Voctor Yodaiken.*
2. temps durant lequel le processeur est occupé à exécuter un certain nombre d'opérations, précédant l'exécution de la routine d'interruption ou la fonction système.

2.1.3. Présentation générale

Sous RT-Linux et RTAI, les interruptions sont gérées d'abord par le noyau temps réel et ne sont passées au noyau de Linux uniquement lorsqu'il n'y a pas de tâches temps réel (voir figure 2.1.).

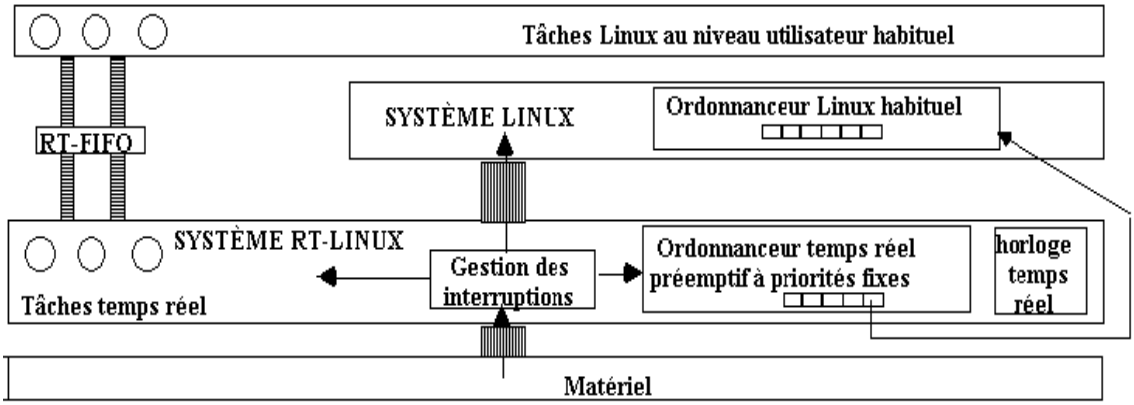


Figure 2.1. Architecture de RT-Linux / RTAI.

Dans RTAI, ceci est réalisé grâce à une couche RTHAL (*RT Hardware Abstraction Layer*).
Lors du lancement du RTAI avec RTHAL, les suivantes actions sont réalisées [RPG] :

- Génération d'un duplicata de la table des descripteurs d'interruption (IDT), et du gestionnaire d'interruptions. Ainsi, la nouvelle IDT devient la table valide du système, quand RTAI est actif.

- Redirection des interruptions du RTHAL, des fonctions d'interruption *enable/disable* et les fonctions de *flags save/restore*, aux fonctions RTAI équivalentes. Celles-ci sont référencées et regroupées dans une seule structure du RTHAL.

- Changement du gestionnaire de fonctions dans la nouvelle *idt_table* à celle du dispatcher RTAI ; ce dernier prend alors le contrôle du système et de ses interruptions.

- Réserver les Timers (8254 et APIC) et les services pour le domaine temps réel.

Sous RT-Linux et RTAI, le noyau temps réel, ses composants ainsi que les applications temps réel s'exécutent tous dans l'espace d'adressage du noyau Linux comme des modules du noyau qui sont chargeables et déchargeables dynamiquement.

Avantages :

- le temps de commutation des contextes des tâches est minimisé.
- modularité du système.

Inconvénient : un bug dans une tâche temps réel peut causer le crash du système.

2.1.4. Les services RTAI

RTAI peut être décomposé en plusieurs services élémentaires, comme les schedulers, FIFOs, et mémoire partagée (*shared memory*), et, d'un ensemble de services "avancés" comme POSIX[1] et l'allocation dynamique de la mémoire.
Ces services sont fournis à travers des modules noyau, qui peuvent être chargé ou déchargé dynamiquement avec les commandes standards de Linux : *insmod* et *rmmod*.
Le module *rtai* est requis, chaque fois qu'un service temps réel est sollicité ; quant aux autres modules ils sont nécessaires seulement lorsque le service temps réel associé est désiré [RPG].
Par exemple, si on veut seulement installer les gestionnaires d'interruption, on aura alors qu'à charger le module *rtai*. Par contre, si on désire communiquer avec un processus Linux standard, il faudrait à ce moment-là charger le module *rtai_fifos*, mais aussi le module *rtai*.
Notons qu'il y a un ordre à respecter lors du chargement des modules. Il faut toujours charger le module *rtai* en premier avant tout autre (car l'initialisation des autres modules dépendent des fonctions fournis par le module *rtai*) [RPG].

Les services élémentaires fournissent un environnement temps réel, avec un ordonnancement préemptif à priorité fixe, ces quatre modules sont les suivants :

- **rtai** : framework de base, gestion des interruptions et un support *Timer*..
- **rtai_sched** : ordonnancement basé sur la priorité, temps réel et préemptif, choisi en fonction de la configuration matérielle.
- **rtai_fifos** : communication par FIFO et synchronisation par sémaphores.
- **rtai_shm** : partage de mémoire.

1. POSIX=Portable Operating System Iterface for Computer Environments (standard IEEE 1003.1b), fondation qui a pour objectif d'homogénéiser les différents systèmes Unix commercialisés.

Les services “avancés” sont fournis dans les modules suivants :

- **lxrt** : utilise les appels système de RTAI à partir de l’espace utilisateur
- **rtai_pthread.o** : Pthreads, POSIX 1003.1c
- **rtai_pqueues.o** : Pqueues, POSIX 1003.1b (support files de messages)
- **rt_mem_mgr** : gestion dynamique de la mémoire pour le temps réel.

2.1.4.1. Les schedulers

La distribution RTAI fournit trois ordonnanceurs, tous temps réel, préemptifs et basés sur la priorité fixe.

a- UP scheduler : pour Uni Processor (localisé dans le répertoire *upscheduler*).

- Utilisé par les plates-formes monoprocesseur.
- Supporte les timers 8254 (fréquence maximale : 1 193 180 Hz).
- Supporte le mode one-shot ou le mode périodique (pas les deux à la fois).

b- SMP scheduler : pour Symmetric Multi Processor (répertoire *smpscheduler*).

- Dédié aux machines multi-processeurs.
- Supporte les Timers 8254 et APIC.
- Supporte le mode one-shot ou le mode périodique (pas les deux à la fois).
- La fonction : `rt_set_runnable_on_cpus` affecte une tâche à un ensemble de CPUs.
- La fonction : `rt_set_runnable_on_cpuid` affecte une tâche à un processeur précis.

c- MUP scheduler : pour Multi-Uniprocessor (dans le répertoire *mupscheduler*)

- Dédié uniquement aux machines multi-processeurs.
- Supporte le mode one-shot **et** le mode périodique.

2.1.4.2. Inter-Process Communications (IPCs) :

Le terme en anglais *Inter-Process Communications* (IPCs), décrit les différents mécanismes de synchronisation et de communication entre processus.
Linux fournit le standard Système V IPC pour le partage de mémoire, FIFOs, sémaphores, tubes (Pipes),… qui peuvent être utilisés par les processus utilisateurs de Linux, pour le transfert et le partage de données.

Bien que ces mécanismes ne soient pas disponibles pour les tâches temps réel, RTAI fournit un ensemble de mécanismes IPCs, qui peuvent être utilisés pour transférer et/ou partager des données entre tâches et processus, dans les domaines temps réel et l'espace utilisateur Linux.

Ces mécanismes regroupent dans le RTAI natif [RPG]:

- FIFOs temps réel : (First In First Out) est un buffer en lecture/écriture, utilisé pour le transfert de données asynchrones entre les tâches temps réel et les processus Linux, ce buffer est ouvert en écriture par un processus et en lecture par un autre.

- Mémoire partagée : fournit un moyen de transfert de données d'une traite entre le domaine temps réel et l'espace utilisateur Linux, à travers lequel, une portion de mémoire physique est réservée pour la s'entre partager.

- Mailboxes : fournit une capacité de transfert de données de tailles définis par l'utilisateur, entre Linux et RTAI. Il est supposé qu'un format de protocole de messages soit imposé au niveau de l'application.

- Sémaphores : sont utilisés pour la synchronisation entre tâches ou partage de ressources communes.

- RPCs (*Remote Procedure Calls*) : similaire au cas de QNX. Les RPCs transferts un entier non signé ou un pointeur à la tâche(s) destinataire(s).

Pour les IPCs fournis par le module POSIX, on citera [RPG]:

- Mutexes : la variable mutex agit comme un verrou d'exclusion mutuelle, pour permettre aux *threads*, le contrôle d'accès aux données. Ceci garantit qu'une seule thread à la fois peut agir sur la donnée protégée.

- Variables conditionnelles : une variable condition offre un moyen pour nommer un événement. Ce dernier peut être vu comme simple conteur avec une valeur limite à atteindre, une valeur particulière, ou même une variable de type flag (set/clear) ; ou plus encore, il peut être vu comme une coïncidence, impliquant de multiples événements. La thread est fortement concernée par ces événements, car la rencontre d'un certain d'eux, permet à la thread de modifier son exécution courante vers une phase ou un traitement particulier, suivant la nature de l'événement. La librairie *Pthread* fournie avec POSIX permet, justement d'une part à la thread d'exprimer son intérêt à un événement particulier, et d'autre part, de signaler aux threads qu'un événement est rencontré.

- La file de messages : POSIX offre une forme d'abstraction générale de communication entre tâches temps réel. La queue de messages, offre la possibilité aux messages de différentes tailles d'être insérés ou placés en elle, tout en gérant leur rangement suivant leur priorité.

2.1.4.2.1. FIFOs temps réel :

Pour une FIFO native à RTAI, les suivantes caractéristiques sont données :

- Ressemble au mécanisme de tube (pipe).
- Peut servir à la communication entre tâches temps réel, entre processus ou encore entre tâches temps réel et processus Linux.
- Peut se créer dans l'espace utilisateur ou l'espace noyau.
- Peut être créée en lui associant un nom.
- Sa taille peut être modifiée après la création de la FIFO.
- Supporte plusieurs lecteurs/écrivains.
- Supporte les sémaphores pour la synchronisation.
- Correspond aux supports /dev/rtf0..63 dans le système de fichiers.

Implémentation :

Voici à présent, quelques principaux prototypes de fonctions, qui servent à manipuler la FIFO (en incluant le fichier *rtai_fifos.h* dans le module d'application) :

- Création d'une FIFO :

```
int fd = rtf_open_sized(const char *dev, perm, size);
```

- Pour détruire une FIFO :

```
int rtf_destroy (unsigned int fifo);
```

- Changement de taille d'une FIFO dans l'espace utilisateur :

```
rtf_resize(int fd, int new_size)
```

- A partir de Linux, la création se fait :

```
$mknod /dev/rtfX c Y X
```

où:

X est le numéro inférieur du périphérique (allant de 0 à 63).
c impilque que la FIFO est un périphérique de type caractère.
Y est le numéro supérieur.

Exemple :

```
$mknod /dev/rtf1 c 63 1
```
(exécutée à partir d'un shell Linux).
Puis, dans le programme C utilisateur, on peut ouvrir la FIFO et la manipuler :

```
…
fifo_id = Open("/dev/rtf1", O_READONLY);
…
```

2.1.4.2.2. Le partage de mémoire [RPG]:

- Les données écrites ne sont pas mises en queue.
- Ceci implique un accès direct et non séquentiel aux structures de données de tailles différentes.
- La mémoire partagée peut être écrite ou lue par n'importe quel nombre de processus Linux ou de tâches temps réel.

- Le blocage pour synchronisation n'est pas directement supporté ; un compteur devrait alors être associé à chaque donnée pour s'assurer de sa bonne mise à jour.
- L'exclusion mutuelle n'est pas garantie entre l'environnement temps réel, et ce, de Linux.
- L'interruption des lecteurs/écrivains n'est pas détectée, il est alors recommandé d'adopter un mécanisme de protection comme Mutexes ou similaire.

Implémentation :

Créer le périphérique de mémoire partagée. Ce dernier reste dans le système, et nous n'aurons pas à le recréer, lors d'une prochaine utilisation. Il est implémenté sous forme d'un véritable périphérique. Mais pour une utilisation particulière d'adresse mappée on peut utiliser la fonction :

```
adr = rtai_malloc_adr( start_ address, name, size) ;
```

Pour ce faire, on suit les étapes suivantes :

- Compiler et charger le module *rtai_shm*

```
$cd <rtai>/shmem
$make
$insmod rtai_shm
```

- De même, compiler et charger le module d'application.
- La création du périphérique est prise en compte dans le fichier *Make* (le périphérique est /dev/rtai_shm).

- Pour pouvoir utiliser la mémoire partagée à partir de l'espace utilisateur :

```
adr = rtai_malloc(name, size);
```

où :

name : identifiant du bloc (il est de type long non signé).
size : est la taille en octets du bloc de mémoire partagée (4096 est la valeur par défaut).
adr : est le pointeur renvoyé vers la base du bloc créé.

- La libération se fait par le biais de la fonction : `rtai_free(name, adr);`

- Pour pouvoir utiliser la mémoire partagée dans l'espace noyau :

```
adr = rtai_kmalloc(name, size);
```

- Pour la libération : `rtai_kfree(name);`

2.1.4.2.3. Mailboxes (Boite aux lettres) [RPG]

Le mailbox est flexible, et possède les caractéristiques suivantes :

- Il peut être explicitement installé, pour prendre en charge des messages de tailles personnalisées.
- Plusieurs émetteurs/récepteurs peuvent se connecter à la fois au mailbox, où l'ordre dans lequel les messages sont pris dépend, de la priorité des récepteurs.
- Lorsqu'un long message devrait être envoyé, le service mailbox offre des fonctions qui permettent au processus d'envoyer seulement une partie du message en lui retournant le nombre d'octets non envoyés du message ou bien, continuer à envoyer le message jusqu'à ce que celui-ci soit accepté.

Les mailboxes peuvent remplacer les FIFOs, cependant le nombre d'appels à la fonction *memcpy* va doubler, et rendra inefficace la communication, bien que ça soit acceptable pour les messages courts.

Implémentation :

Les services du mailbox sont offerts par RTAI scheduler. Pour pouvoir les utiliser, il faudrait inclure le fichier *rtai_sched.h* lors de l'écriture du module d'application.
Le mailbox est créé et initialisé, par le biais de la fonction :

```
result = rt_mbx_init(MBX *mbx, size) ;
```

où :

mbx : pointeur vers une la structure de données mailbox (déclarée dans *rtai_sched.h*).
size : taille du mailbox à utiliser, choisie en fonction de la taille des messages à envoyer.

result : devrait être nul, sinon il y a erreur de création du mailbox.

- Pour supprimer le mailbox :

```
result = rt_mbx_delete(MBX *mbx) ;
```

- Pour envoyer un message d'une façon inconditionnelle (retourner une fois le message en entier a été envoyé ou bien, une erreur est survenue) :

```
int rt_mbx_send(MBX *mbx, void *msg, int msg_size) ;
```

- Pour recevoir un message d'une manière inconditionnelle :

```
int rt_mbx_receive(MBX *mbx, void *msg, int msg_size) ;
```

2.1.4.2.4. Messages RTAI et RPCs [RPG]

RTAI offre une facilité de communication de messages inter-tâches, par laquelle, une valeur de 32-bit peut être passée entre les tâches temps réel.
Remote Procedure Calls (RPCs) fait de même, mais les tâches sont dans ce cas couplées en attente d'une réponse du receveur.

Implémentation :

Les messages RTAI ainsi que RPCs, sont implantés au niveau du fichier *rtai_sched.c*, il faudrait alors inclure dans le module d'application le fichier r*tai_sched.h* à chaque référence aux fonctions associées.

- pour faire un appel distant inconditionnel :

```
rt_rpc(RT_TASK *task, unsigned int msg, unsigned int *reply);
```

La fonction *rt_rpc* envoie un message à la tâche *task*, et se met en attente d'une réponse *reply*. La tâche appelante est alors bloquée et mise en file d'attente.
La tâche *task* récupère le message par le biais de la fonction *rt_receive* (faisant partie de la liste des messges RTAI), réalise le traitement le cas échéant, puis se termine en léguant en quelque sorte le contrôle à la fonction appelante, en invoquant la fonction : *rt_return(…).*
La fonction appelante est réveillée, et retrouve la réponse pointée par *reply.*

2.1.5. Les modules sous Linux

Un module noyau Linux est une portion de code qui, lorsqu'elle est chargée, elle devient partie intégrante du noyau Linux ayant accès au même espace mémoire.

Une tâche temps réel s'exécutant comme un module noyau consiste en trois parties :

- La fonction `init_module`()
- Code spécifique de la tâche (API RTAI ou POSIX)
- La fonction `cleanup_module`()

`Init_module() :`

- un module noyau doit toujours contenir une fonction `init_module`

- `init_module`() est invoqué par la commande `insmod`

- `init_module`() contient des allocations de ressources, la déclaration et le démarrage de tâches.

`Cleanup_module() :`

- Cette entrée est invoquée lorsque le module est supprimé

- `cleanup_module`() est invoqué par la commande `rmmod`

- c'est l'endroit idéal pour la libération des ressources, l'arrêt et la suppression des tâches

2.1.6. Exemple

Nous allons donner un exemple, à travers lequel, nous illustrons la manière de procéder pour créer un module d'application temps réel avec RTAI (en langage C), sa compilation, son chargement, tout en détaillant son code.

```
// Fichier tache.c
#include <linux/module.h>
#include <asm/io.h>

#include <rtai.h>
#include <rtai_sched.h>

#define TICK_PERIOD 10000000
#define STACK_SIZE 2000
#define LOOPS 1000000000
#define NTASKS 2

static RT_TASK tache[NTASKS];

static RTIME tick_period;

static void Thread(int t)
{
     unsigned int loops = LOOPS;
     while(loops--) {
     rt_printk("TASK %d %d\n", t, tache[t].priority);
          rt_busy_sleep(500000);
          rt_task_wait_period();
     }

}
```

```
int init_module(void)
{
     RTIME now;
     int i;

     for (i = 0; i < NTASKS; i++) {
          rt_task_init(&tache[i], Thread, i, STACK_SIZE,
               NTASKS - i - 1, 0, 0);
     }
     tick_period =
     start_rt_timer(nano2count(TICK_PERIOD));
     now = rt_get_time() + NTASKS*tick_period;
     for (i = 0; i < NTASKS; i++) {
          rt_task_make_periodic(&tache[NTASKS - i - 1],
          now, NTASKS*tick_period);
     }
     return 0;
}

void cleanup_module(void)
{
     int i;

     stop_rt_timer();
     for (i = 0; i < NTASKS; i++) {
          rt_task_delete(&tache[i]);
     }

}
```

Commençons par analyser le code :

Les fichiers d'entête, servent à compiler le fichier *tache.c*, que nous avons écrit.
`#include <linux/module.h>` est nécessaire pour pouvoir créer un module noyau.
`#include <asm/io.h>` contient des définitions d'IN/OUT du x86 d'Intel.

`#include <rtai.h>` et `#include <rtai_sched.h>` contiennent des définitions ou les prototypes de fonctions RTAI temps réel à utiliser.

`static RT_TASK tache[NTASKS] ;` déclare un tableau static de deux tâches.
RT_TASK est de type : struct tache * (définit dans *rtai.h*).
`static RTIME tick_period ;` RTIME est de type : long long

Traitons en premier lieu la fonction, `init_module(void)`, qui sera d'ailleurs la première à être exécutée (la commande : *insmod* va l'invoquer en premier, c'est en quelque sorte la fonction `main()` habituelle).
Elle commence alors par créer deux tâches en invoquant la fonction : `rt_task_init`, dont le prototype est le suivant :

```
int rt_task_init( RT_TASK *task, // contexte de la tâche
                  void (*rt_thread)(int), //la thread
                  int data, //paramètres à la thread
                  int stack_size, //la taille de la zone de pile
                  int priority,
                  int uses_fpu,
                  void(*signal)(void) );
```

et l'adresse de la fonction `thread` écrite plus haut est donnée, comme paramètre.

Puisque nous n'avons pas défini ou déclaré le mode *one_shot*, le timer 8254 sera alors utilisé. La fonction : `start_rt_timer`, active le timer, en lui passant comme paramètre le nombre de top d'horloge (en anglais *tick*), signifiant la période. Ceci après avoir converti ces tops, en une valeur, relative à la cadence de l'horloge considérée (dans ce cas, il s'agit du timer 8254) par le biais de la fonction : `nano2count`.
La fonction `rt_get_time()` renvoie, le temps absolu en unité interne, écoulé depuis l'activation du timer.
La fonction `rt_task_make_periodic` initialise les tâches précédemment créées, en les déclarant comme périodiques, et dont la date de réveil fixée à l'instant courant.

En invoquant `rt_printk`, la fonction `thread` affiche un message, la valeur du paramètre *t* passé en argument par la fonction `rt_task_init` lors de la création de la tâche, et, la priorité de la tâche.
Une fois ceci fait, la tâche se bloque pendant 500µs, en sollicitant pour cela, la fonction : `rt_busy_sleep(int nanosecs)`, cette dernière bloque la tâche pour une période donnée en paramètre, et ne lègue pas pendent ce temps là, le contrôle au scheduler (la tâche n'est pas préemptée). Après quoi, la tâche attend une période (10 ms), mais peut être dans ce cas préemptée.
Ce scénario est répété par les deux thread en concurrence, tant que le compteur `LOOP` ne s'est pas épuisé.

La compilation :

Pour compiler le module, nous pouvons soit, directement procéder à la compilation, ce qui s'avère fastidieux si une erreur de frappe survienne, ou que nous aurons à recompiler à nouveau le module.
Pour cela, il est préférable de créer un fichier *Makefile* dans lequel nous écrirons, les commandes une seule fois.

Un exemple du fichier *Makefil*e est le suivant :

```
all: tache.o

CC=gcc
RTAISRC=/root/Downloads/kernel/rtai-24.1.13
KERNELSRC = /lib/modules/$(shell uname -r)/build


CFLAGS=  -I$(RTAISRC)/include -I. -D__KERNEL__
-I$(KERNELSRC)/include
 -Wall -Wstrict-prototypes -Wno-trigraphs -O2
-fno-strict-aliasing
-fno-common -fomit-frame-pointer -pipe
-mpreferred-stack-boundary=2
-march=i686 -DMODULE

tache.o: tache.c
     gcc  $(CFLAGS) -c -o $@ $<

clean:
     rm -f tache.o
```

Une fois ce fichier créé, il suffit alors d'exécuter la commande : *make*
Ceci va compiler le source nommé *tache.c* et délivrera en sortie le module *tache.o*

Le chargement :

Avant de charger le module *tache.o* créé, nous devons au préalable, charger les modules RTAI (*rtai.o* et *rtai_sched.o*), pour cela on exécute les commandes suivantes :

```
$cd ./<path rtai>
$make
$insmod rtai
$cd ./<path rtai>/<path rtaisched>
$make
$insmod rtai_sched
$cd ./<path exemple>
$make
$insmod tache
```

Le déchargement :

Pour décharger le module *tache.o* il suffit d'exécuter :
`$rmmod tache`

Ceci va exécuter la fonction : `cleanup_module()` écrite dans le fichier : *tache.c*
Si nous l'analysons, nous constaterions qu'elle fait appel à la fonction : `stop_rt_timer()`, qui stop le timer, puis deux appels à la fonction `rt_task_delete` sont effectués, pour détruire les deux tâches créées.

2.2. VxWorks

2.2.1. Historique

VxWorks est aujourd'hui l'exécutif temps réel le plus utilisé dans l'industrie. Il est développé par la société Wind River[1] qui a également racheté récemment les droits du noyau temps réel PSOS, un peu ancien mais largement utilisé.
VxWorks est fiable, à faible empreinte mémoire, totalement configurable et porté sur un nombre important de processeurs (PowerPC, 68K, CPU32, ColdFire, MCORE, 80x86, Pentium, i960, ARM, StrongARM, MIPS, SH, SPARC, NECV8xx, M32 R/D, RAD6000, ST 20, TriCore).
Un point fort de VxWorks a été le support réseau (sockets, NFS, RPC...) disponible dès 1989 aux développeurs, bien avant tous ses concurrents. VxWorks est également conforme à POSIX 1003.1b.

2.2.2. Présentation générale [DP 91]

L'approche proposée par l'exécutif VxWorks de Wind River Systems, est née de deux constations :

- la première est qu'Unix, riche d'utilitaires, de possibilités réseaux et de convivialité de développement n'est pas en mesure d'offrir des outils adaptés au temps réel et en conséquence de répondre à des performances temps réel ;
- la deuxième, est que l'architecture fortement optimisée des systèmes d'exploitation temps réel (tel OS9, par exemple) ne facilite pas forcement leur utilisation en tant que plate-forme de développement.

VxWorks est donc au sens strict un exécutif temps réel, spécialement conçu pour le développement en environnement Unix. La figure 2.2. présente l'architecture globale de cet exécutif, avec les fonctionnalités spécifiquement temps réel et les modules d'extension propres à l'environnement Unix.

Tout comme beaucoup d'autres exécutifs de ce type, Unix est vu tel un ensemble logiciel de haut niveau pour le développement des applications, exécutées et mises au point sous VxWorks. Le noyau de ce dernier offre des primitives de base de gestion de processus et de sémaphores sur lesquelles sont construites les fonctions système accessible par l'intermédiaire du système d'exploitation.

1. http://www.windriver.com

<table>
<tr><td>Loging</td><td colspan="2">Entrées/Sorties Formatées (Unix)</td><td colspan="2">Désassembleur Symbolique</td><td>Chargeur (Unix)</td><td>Remote login</td><td>NFS</td><td>RPC</td><td>Sockets</td><td colspan="2">Debugger symbolique</td></tr>
<tr><td>Système de fichiers</td><td>Système d’E/S</td><td colspan="2">Sémaphores</td><td colspan="2">Support coprocesseur</td><td>Table de Symboles système</td><td>Gestion mémoire</td><td>Gestion réseau</td><td>Gestion du multitâche</td><td>Gestion des ITs</td></tr>
<tr><td colspan="11">Libraires</td></tr>
<tr><td colspan="11">NOYAU TEMPS REEL</td></tr>
</table>

Figure 2.2. Architecture globale de l’exécutif VxWorks

2.2.3. caractéristiques temps réel [DP 91]

L’ordonnancement des tâches est basée sur le principe de priorité (256 niveaux possibles) avec préemption. Pour des tâches de priorités identiques, c’est la technique du *round-robin* qui est mise en œuvre. Le temps de commutation de contexte est indépendant du nombre de tâches installées dans le système ; à titre indicatif, sur un 68020 à 25 Mhz, il est de 17µs

La synchronisation est assurée par signaux, les fonctionnalités état basée sur les primitives de gestion de sémaphores. VxWorks supporte trois types de sémaphores :

- les sémaphores booléens,
- les sémaphores à compte,
- les sémaphores à priorité.

Ces derniers permettent de contourner l’effet subversif des priorités relatives aux tâches. En effet, une tâche de faible priorité sera susceptible d’hériter momentanément de la priorité de la tâche de plus forte priorité en attente sur le sémaphore. Les trois types de sémaphores acceptent des attentes avec *time-out* et autorisent le chaînage des processus dans les files de type FIFO ou à priorité. Du point de vue de la communication, l’exécutif intègre des outils bâtis à partir des primitives de gestion de sémaphores. On dispose de :

- files de messages, de type FIFO, de longueur variable et accessibles à partir de tout processus utilisateur ou système, y compris une routine de service d’interruption ;

- tubes de communication (*pipe*), faisant office d’interface entre un processus et une file de message avec l’avantage d’utiliser des services d’entrées/sorties indépendants du matériel (fonctions de gestion de périphériques virtuels).

Lors de l'arrivée d'une interruption, une routine ISR (*Interrupt Service Routine*) est invoquée, l'exécutif n'intervenant pratiquement pas entre l'arrivée du phénomène d'interruption et l'exécution de la routine ISR. Les routines ISR peuvent si besoin est, envoyer des signaux, manipuler des sémaphores, des files de messages ou de pipes.

Par ailleurs, VxWorks dispose de fonctionnalités autorisant la manipulation des vecteurs d'interruption directement en langage de haut niveau tel que le langage C ainsi que l'association d'une routine écrite en C à une source d'interruption matérielle.

A propos du masquage des interruptions et de la préemption, VxWorks minimise les temps de fonctionnement en "mode système", mode dans lequel généralement tout exécutif est non préemptible (et au pire, non interruptible) pour assurer l'intégrité de manipulation des ressources du noyau (files internes, pointeurs, etc.) de telle sorte que le temps de latence en interruption se réduit à 8µs.

Le fonctionnement en mode "système" ne concerne que les manipulations vitales de gestion du multitâche et les opérations de base de synchronisation.
Tout ce qui concerne les fonctionnalités de communication, de gestion mémoire, de gestion multiprocesseur (par réseaux) est essentiellement exécuté en mode utilisateur (c'est à dire dans le contexte du processus concerné), donc préemptible.

Le tableau 2.1. résume les performances temporelles de VxWorks pour un processeur 68020 à 25Mhz. Outre les caractéristiques précitées, VxWorks intègre une agence de gestion du temps incluant des *watchdog*, des *timer* de délai sur processus, des possibilités de *time-out* sur opérations du noyau, etc.
De même, l'exécutif dispose d'une agence de gestion mémoire supportant les fonctions d'allocation compatibles C ANSI et Unix.

Commutation de contexte	17µs
Commutation de tâches	35µs
Opérations sur sémaphores (P et V)	8µs
Latence à la préemption	11µs
Latence en interruption	8µs

Tableau 2.1. Performance de VxWorks

2.2.4. L'interface avec Unix [DP 91]

L'une des caractéristiques de VxWorks est de s'intégrer pleinement dans un environnement de développement de type Unix, et de faire partie de cette nouvelle génération d'exécutifs basés sur ce quasi standard de développement.
Qui dit environnement Unix, sous-entend généralement une machine cible, une machine hôte et un réseau , donc le support de toute une série d'utilitaires standard de communication par réseau. La figure 2.3. présente les différents utilitaires de communication réseaux supportés par VxWorks.

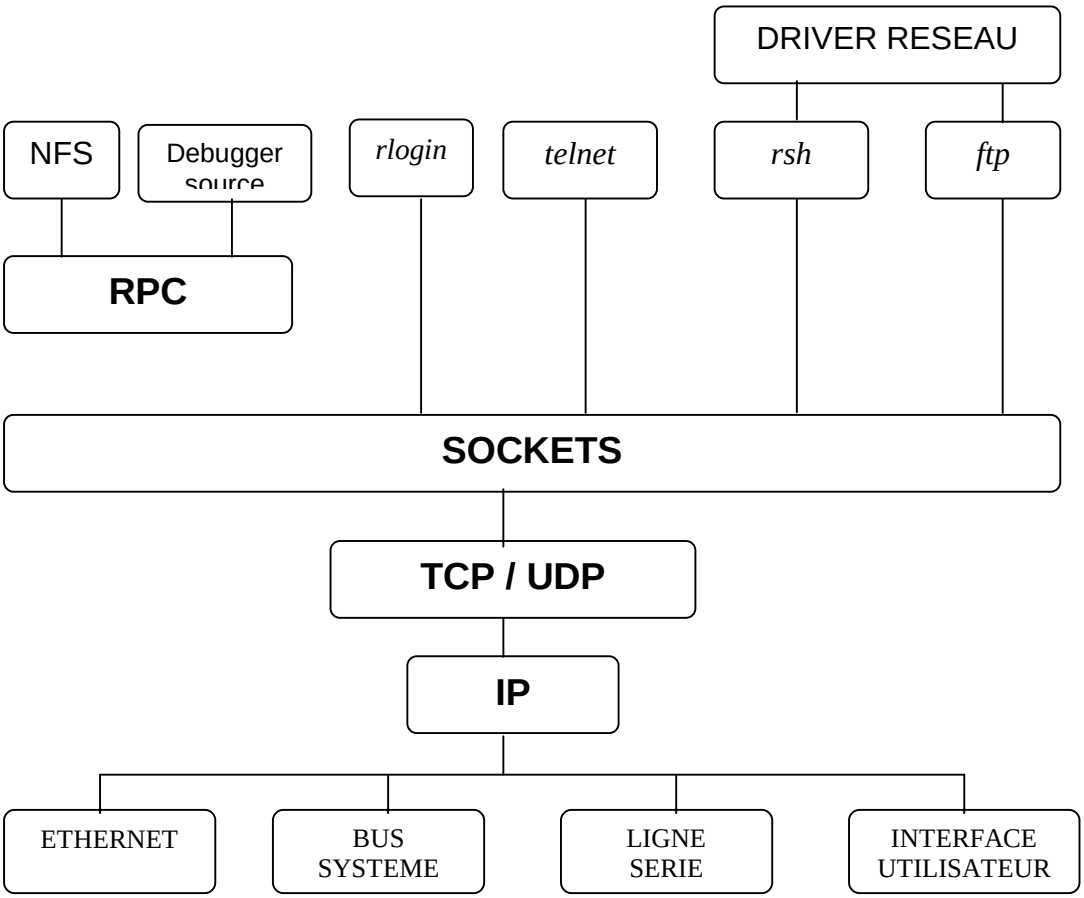


Figure 2.3. L'environnement de communication de VxWorks

On distingue les librairies de communication (*sockets*) compatibles Unix BSD 4.3, les protocoles de transmission TCP, IP, UDP, le protocole de session RPC, le protocole de transfert de fichiers NFS et les process applicatifs telnet, ftp, rlogin et rsh.

Les *sockets* sont des primitives de communication à la base du développement d'utilitaires réseau ou d'application. VxWorks supporte des connexions en mode datagramme (par l'intermédiaire de UDP/IP) et en mode connecté (grâce à TCP/IP), ce dernier mode autorisant un contrôle de bout en bout du flot de données transmis. Le protocole RPC (*Remote Procedure Call*) assure l'appel par les tâches VxWorks ou les processus Unix de routines exécutées indifféremment par la machine cible (VxWorks) ou hôte (Unix). Enfin, signalons brièvement les rôles respectifs des divers process supportés :

- rsh : *remote shell*, permet aux processus applicatifs sous VxWorks d'accéder par réseau aux commandes Unix ;
-ftp : *file transfert protocol*, autorise l'accès à partir du système VxWorks aux fichiers du système Unix et inversement ;
- telnet assure la connexion au shell VxWorks à partir de la machine Unix, de même que rlogin assure aux utilisateurs de la cible (VxWorks) une connexion au shell de la machine hôte (Unix).

2.2.5. Environnement de compilation et d'exécution [SETR]

2.2.5.1. Chargeur

L'objectif du chargeur dans l'environnement VxWorks est de pouvoir lancer l'exécutif et/ou les applications.
Lors du démarrage de VxWorks (système cible), le processus d'amorce ou de *boot*, suit les opérations suivantes :

- Lancement du chargeur à la mise sous tension (EEPROM). Chargeur et hôte (Unix) communiquent par lien série.
- Modification éventuelle des paramètres de *boot*.
- Configuration sur la cible (VxWorks) d'une interface Ethernet.
- Chargement par ftp ou rsh de l'image.
- Démarrage de l'exécutif et des tâches VxWorks.
- Montage NFS du répertoire utilisateur.

Les paramètres de *boot* peuvent être les suivants :

- Adresse IP de l'hôte (station Unix) et de la cible (VxWorks).
- Localisation de l'image constituant l'exécutif.
- Nom de l'utilisateur.
- Protocole à utiliser pour le chargement.

Les paramètres généralement modifiés par l'utilisateur sont, le nom de l'utilisateur, le fichier image et le protocole de transfert.

Nous donnons dans ce qui suit quelques commandes du chargeur :

- CRTL+X : redémarrage de la carte.
- @ : chargement et lancement de l'exécutif.
- c : modification des paramètres de chargement.
- p : affichage des paramètres de chargement.
- ? : aide en ligne.

Exemple :

1. A partir de l'hôte (exemple : Sun), on se connecte à la cible (VxWorks sur processeur Motorola 68030) par une liaison série.

2. Affichage et modification éventuelle des paramètres à l'invite de boot :

```
[VxWorks Boot] : p
boot device           : ei
processor number      : 0
host name             : SUNKabylie
file name :           : /exemple/VxWorks/bootimage
inet on ethernet (e)  : 192.12.11.12
host inet   (h)       : 192.12.11.1
user   (u)            : toufik
ftp password (pw)   (blank = use rsh) :
flags   (f)           : 0x0
target name   (tn)    : VxTiziOuzou
```

3. Chargement de l'exécutif :

```
[VxWorks Boot] : @
 …
 ->
```

2.2.5.2. Compilation croisée

Nous pouvons compiler les fichiers d'application de VxWorks sur la station Sun de l'exemple précédent. Pour cela, il suffit d'avoir un compilateur croisé (supportant également le jeu d'instruction de Motorola 68030 exemple : cc68k). Bien évidemment, le binaire produit, ne s'exécutera pas sur la station Sun mais destiné à l'environnement VxWorks.
De plus, il n y aura pas d'édition de liens, celle-ci se fera dynamiquement, lors du chargement du module objet dans la machine cible (sous VxWorks).

Exemple :

Soit l'exemple de programme trivial (bonjour.c) :

```
#include <vxWorks.h>
#include <stdio.h>
#include <taskLib.h>

mafonction()
{
  int id=taskIdSelf();
        printf("Bonjour le monde, nom tache :%s.
           id tache =%x\n",taskName(id),id) ;

}
```

N.B : il n y a pas de fonction main ().

La compilation sur l'hôte (pas d'édition de liens) :

```
SUNKabylie$cc68k –g –c bonjour.c –DCPU=MC68030
```

2.2.5.3. Le shell VxWorks

Puisque l'abstraction faite dans les exécutifs temps réel, est de considérer toute unité d'exécution comme étant une tâche, le shell est alors vu comme telle, et constitue l'interface entre l'exécutif et l'utilisateur.
Les services offerts par un shell VxWorks :

- Chargement de code et de données. Edition des liens.
- Mise au point (niveau assembleur).
- Consultation et modification des ressources VxWorks (ex : tâches).
- Gestion de symbole des modules.
- Services classiques offerts par un shell : variables, redirection d'E/S, appel de fonction, etc.

Quelques commandes du shell VxWorks :

- *ls, ll, cd, pwd*. Commandes de parcourt du répertoire utilisateur.
- *ld*. Chargement d'un module et édition des liens.
- *ModuleShow*. Affiche la liste des modules.
- *sp, td, ts, tr, repeat*. Opérations sur les tâches : création, suppression, etc.
- *i, ti, checkStack, pc, printErrno*. Information sur les tâches.
- *help*. Aide en ligne.

Exemple :

Ainsi, pour charger le module créé dans l'exemple précédent, il suffit de se placer dans le répertoire où se trouve le module.

```
->cd "nom_répertoire"
```

Puis de charger le module :

```
->ld < bonjour.o
```

Et on pourra exécuter le module de deux manières :

- soit l'exécuter par appel de fonction.
- Soit par la création d'une tâche.

1ère possibilité :

```
->mafonction()
Bonjour le monde, nom tache : cshell. id tache =0x3a8460
```

2ème possibilité :

```
->sp mafonction
task spawned:  id = 0x3a5bc0, name = t1
Bonjour le monde, nom tache : t1. id tache =0x3a8460
```

2.2.6. API VxWorks

VxWorks est très utilisé dans les systèmes embarqués. A cet effet, il dispose d'un seul espace d'adressage. Ainsi, les appels au système ne sont que des appels de fonctions, mais ceci n'empêche pas le fait que VxWorks soit muni d'une API complète, et facilite aux concepteurs d'applications, le développement rapide de leurs programmes, en leur évitant l'embrouillement et la prise en compte des spécificités du code noyau. Autrement dit, l'utilisateur a comme impression qu'il développe dans un espace propre à lui, sans trop tenir compte des variables noyau (le fait contraire est vu dans Linux temps réel).

On se propose de présenter en bref, les principaux services que fournit l'exécutif VxWorks. Les services comme : gestion des tâches, gestion du temps, les sémaphores, les interruptions, la mémoire et la communication par messages.

2.2.6.1. Gestion des tâches

Pour manipuler l'entité tâche dans un programme C, il faudrait inclure le fichier entête contenant les définitions et les prototypes de fonctions de gestion des tâches, ceci par le biais du fichier:

```
#include <tasklib.h>
```

- création d'une tâche :

```
TASK_ID tId = taskSpawn(char *nom,
                        unsigned char priorit_e,
                        unsigned flags,
                        unsigned taille_pile,
                        VOID_FUNCPTR fonction,
                        void arg1,...);
```

- suspension /reprise d'une tâche :

```
STATUS taskSuspend(TASK ID tId);
STATUS taskResume(TASK ID tId);
```

- verrouillage et déverrouillage d'une tâche :

```
STATUS taskLock();
STATUS taskUnLock();
```

- destruction d'une tâche :

```
STATUS taskDelete(TASK ID tId);
```

2.2.6.2. Gestion du temps

Une multitude de fonctions est offerte pour l'utilisateur, en vue de subvenir aux besoins temps réel dans les applications à concevoir, nous en citerons que quelques-unes d'elles.

- Pour bloquer une tâche pendant une certain nombre de top d'horloge :

```
STATUS TaskDelay(int nb_tick) ;
```

- Pour positionner/consulter le nombre de top d'horloge, depuis l'initialisation :

```
STATUS Tickset(ULONG tick) ;
ULONG Tickget(void) ;
```

- Pour modifier ou consulter la fréquence d'horloge du système :

```
STATUS SysClkRateSet(ULONG frq) ;
ULONG SysClkRateGet(void) ;
```

- Pour créer/activer (exécuter une fonction à un chien de garde) :

```
int wd=wdCreate() ;
wdStart(int wd,int delai,(FUNCPTR)function,param_fonction) ;
```

2.2.6.3. Les sémaphores

```
#include <semLib.h>
```

- Pour créer un sémaphore booléen / à compte / mutex, initialisé à une valeur *val* :

```
SEM_ID semBCreate(unsigned options, unsigned val);
SEM_ID semCCreate(unsigned options, unsigned val);
SEM_ID semMCreate(unsigned val);
```

Telle que : `options =` {`SEM_Q_PRIORITY, SEM_Q_FIFO`}

- Pour la prise / libération d'un sémaphore :

```
STATUS semTake(SEM_ID semId, int timeout);
STATUS semGive(SEM_ID semId);
```

- Pour la destruction d'un sémaphore :

```
STATUS semDelete(SEM_ID semId);
```

2.2.6.4. Manipulation des interruptions

VxWorks gère jusqu'à 256 niveaux d'interruption, et permet leur manipulation en donnant la possibilité aux programmeurs d'applications, d'associer une routine d'interruption à une fonction C, qu'ils programment.

```
#include <intLib.h>
```

- Pour connecter une routine à une interruption :

```
STATUS intConnect(int vecNum,VOID FUNCPTR funcPtr);
```

- Pour positionner le masque d'une interruption :

```
STATUS intLevelSet(int level);
```

- Pour bloquer / débloquer les interruptions :

```
STATUS intLock();
STATUS intUnLock();
```

2.2.6.5. Gestion de la mémoire

Rappelons le, toutes les tâches possèdent un seul espace d'adressage, les suivantes fonctions travaillent alors toutes dans le même espace d'adressage (du noyau).

```
#include <memLib.h>
```

- Pour allouer / libérer un bloc de mémoire :

```
void *ptr = malloc(unsigned taille);
void free(void *ptr);
```

- Pour afficher la mémoire disponible :

```
void memShow(BOOL flag);
```

2.2.6.6. Communication par messages

```
#include <msgQLib.h>
```

- Pour créer une file de messages :

```
MS_Q_ID msgQCreate(int maxMsg, int msgLen,int options);
```

- Pour envoyer / recevoir un message :

```
STATUS msgQSend(MSG Q ID msgQ, void *message, int longueur);

STATUS msqRReceive(MSG Q ID msgQ, void *buffer,int maxlen,
                   int timeout);
```

2.3. CLEOPATRE

C'est un atelier de développement et de vérification d'une bibliothèque de Composants Logiciels sur Etagères Ouverts Pour les Applications Temps-Réel Embarquées.

Projet développé en France, et impliquant plusieurs organismes français (Institut de Recherche en Informatique de Nantes, Université de Paris Nord, Laboratoire de Robotique de Paris rattaché au CNRS, ROBOSOFT – SA, Institut Universitaire de Technologie de Nantes, et le CEA : Service Robotique et Systèmes intégrés).

2.3.1. Objectifs [CLEO]

Les produits industriels intègrent de plus en plus de logiciels. Ces logiciels, dits enfouis, peuvent influer très fortement sur la compétitivité du produit final de par les réductions de coût de développement et leurs innovations. Leur qualité principale réside dans leur aptitude à supporter, à la demande, une évolution rapide, facile, peu onéreuse, et ce, tout en garantissant une qualité de service prédéfinie. Le développement reste, en effet, un aspect souvent mal maîtrisé, notamment par méconnaissance des concepts temps-réel.

Ce projet apporte des solutions au développement d'applications temps-réel embarquées par la mise à disposition après vérification, de composants logiciels libres et ouverts s'appuyant sur Linux.
Linux Temps-Réel constitue un standard qu'il est opportun de faire évoluer pour s'adapter aux exigences des nouveaux systèmes embarqués. Cette évolution passe par le transfert des compétences universitaires en matière d'informatique temps-réel (ordonnancement, tolérance aux fautes, gestion de ressources,…) et l'apport de réponses aux besoins d'applications industrielles de plus en plus complexes.

2.3.2. Organisation du projet [CLEO]

Le projet CLEOPATRE, de type exploratoire est organisé en sept sous-projets :
Les cinq premiers sont la spécification et le développement des composants appartenant à quatre étagères (Ordonnancement, Tolérance aux fautes, Mécanismes de synchronisation, Serveurs de tâches apériodiques). (Voir figure 2.4.) .
Dans le sixième sous-projet, on expérimentera, sur une application réelle de robotique mobile, l'adéquation de Linux temps-réel muni de son noyau configurable et de sa librairie d'utilitaires, vis à vis des exigences des développeurs.
Le dernier sous-projet se chargera de la documentation et de la dissémination via Internet notamment par la création d'un site. Il effectuera une étude de marché, établira un plan de commercialisation dans laquelle il pourra ultérieurement assurer le support technique auprès des utilisateurs finaux

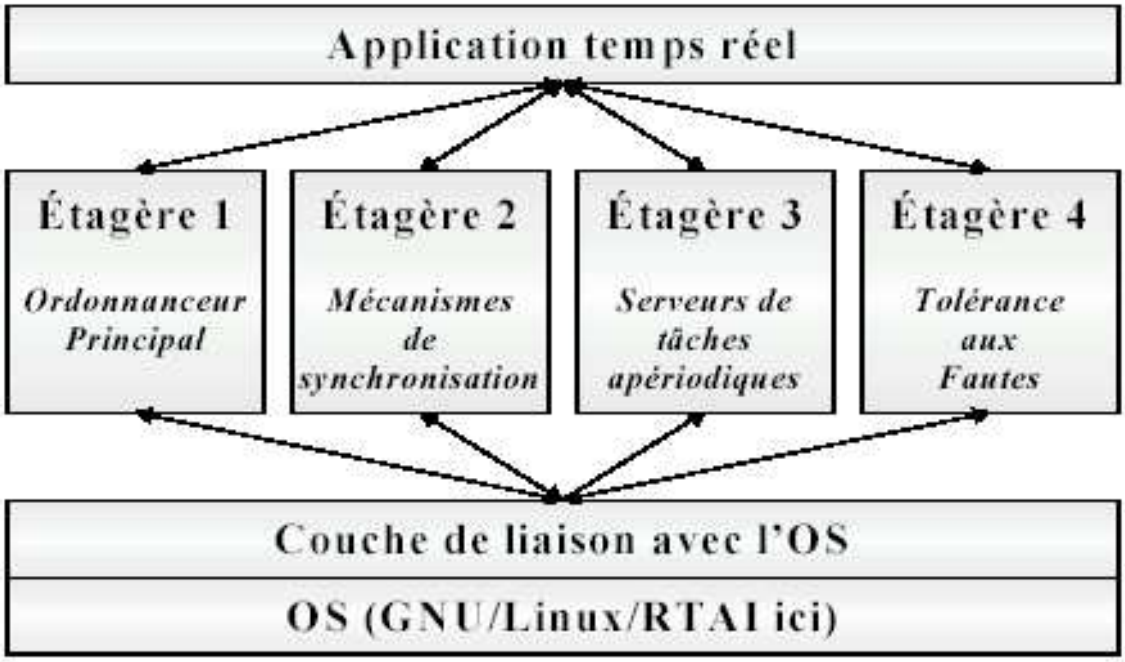


Figure 2.4. Les quatre premières étagères de CLEOPATRE

2.3.3. Démonstrateur [CLEO]

Un robot mobile autonome comme démonstrateur permet de concevoir une multitude de scénarios pour tester et certifier ces composants tant du point de vue de la sûreté de fonctionnement (fiabilité, sécurité) que du respect des contraintes temporelles. L'intégration de composants dédiés au traitement temps-réel et conformes au standard Linux dans une plate-forme constitue une avancée certaine. Celle-ci est destinée non seulement à tester les composants et leur interopérabilité mais aussi démontrer la compétitivité et l'adaptativité de Linux temps-réel en vue de son industrialisation.

2.3.4. Perspectives de marché [CLEO]

CLEOPATRE souhaite, fournir une gamme de composants suffisamment étendue pour toucher un panel le plus large possible de secteurs d'applications, allant des applications les plus critiques, en général à caractère fortement cyclique (industrie militaire, avionique) aux applications les moins critiques (multimédia, vidéo à la demande) en passant par des applications où cohabitent des contraintes à degré de criticité échelonnés (robotique mobile avec télé-opération via Internet). Les perspectives actuelles de ce marché sont donc grandes, aussi bien dans les secteurs de haute technologie qu'au niveau des produits grand public. Ne pas oublier l'intérêt pédagogique de ces logiciels dans l'enseignement universitaire.

2.4. BOSSA[1]

Développé à l 'Ecole des Mines de Nantes [2], BOSSA est une plate-forme ajoutée au noyau Linux pour le développement d'ordonnanceurs de processus.

2.4.1. Objectifs

L'objectif de BOSSA est de simplifier la conception d'ordonnanceurs de sorte qu'un programmeur d'application puisse développer des politiques d'ordonnancement spécifiques sans être expert des systèmes d'exploitation.

Pour ce faire, BOSSA offre les avantages suivants :

- Simplifier l'implémentation d'un scheduler : BOSSA inclut un domaine de langage dédié (DSL : *Domain-Specific Language*), offrant un niveau d'abstraction plus élevé qui permet l'évolution et l'implémentation de nouvelles politiques d'ordonnancement. Un compilateur dédié vérifie le code DSL de BOSSA pour une compatibilité avec l'OS cible, et le translate en code C.

- Intégration simplifiée du scheduler : Une politique d'ordonnancement BOSSA est implantée comme un module qui reçoit les changements d'état des processus sous la forme d'événements pour utiliser cette information et prendre des décisions d'ordonnancement.

- Sécurité : La sécurité est garantie tant au niveau de l'implémentation de la nouvelle politique qu'au niveau langage DSL, étant donné que ce dernier ne génère pas de pointeurs ni de boucles infinies.

2.4.2. Les composants BOSSA

Pratiquement, le noyau BOSSA comprend trois parties :

- Un noyau standard, dans lequel les politiques d'ordonnancement sont remplacées par des événements pris en compte par BOSSA.
- La politique d'ordonnancement écrite par le programmeur contient, des routines gérant les événements possibles de BOSSA. La politique est soit écrite en C directement, ou bien en DSL, et translaté dans ce cas en C, par un compilateur dédié.
- Un OS indépendant, qui gère l'interaction entre le reste du noyau et la politique d'ordonnancement.

1.BOSSA est soutenu en partie par France Telecom, Micro$oft, et le Conseil de Recherche Danois.
2. http://www.emn.fr/x-info/bossa

2.4.3. Du noyau Linux à BOSSA [RTS 05]

Préparer un noyau pour qu'il soit utilisable avec BOSSA nécessite l'insertion des événements cités plus haut à des points particuliers du noyau, appelés *points d'ordonnancement*. L'évolution du noyau Linux pour supporter BOSSA est assez complexe, pour diverses raisons.

D'abord, BOSSA doit être utilisable avec n'importe quelle version de Linux. Une solution basée sur les patches serait insuffisante car les numéros de ligne correspondant aux points d'ordonnancement ainsi que le code entourant ces points diffèrent d'une version de Linux à une autre.
De plus, certains changements nécessaires pour supporter BOSSA dépendent des propriétés de flux de contrôle. Sans compter le fait que toute modification effectuée à la main sur plusieurs fichiers source (le code source de Linux est de l'ordre de 100Mo) peut générer des erreurs.
C'est alors que le principe de réécriture a été utilisé pour implémenter une fonctionnalité qui contient une collection de fragments de code ainsi qu'une description formelle des points d'ordonnancement.
Cette fonctionnalité correspond à un ensemble de règles de réécriture pour décrire les conditions à vérifier en vue d'insérer des événements spécifiques.
Voici un exemple de règle de réécriture :

```
n:(call try_to_wake_up))=>
Rewrite(n,bossa_unblock_process(args))
```

Cette règle sera appliquée à chaque fois qu'un appel à la fonction `try_to_wake_up` est trouvé. Cet appel sera référencé par n.
L'utilisation de `Rewrite` indique que l'appel `try_to_wake_up` est remplacé par un appel à `bossa_unblock_process`. La fonction `wake_up_process` suivante illustre l'effet de l'application de la règle décrite ci-dessus :

```
wake_up_process(struct task_struct *p)
{
 #ifdef CONFIG_BOSSA
 return bossa_unblock_process(WAKE_UP_PROCESS, p,0);
 #else
 return try_to_wake_up(p,0);
 #endif
}
```

Le noyau est réécrit à l'aide d'une quarantaine de règles logiques d'une assez grande complexité implémentées en Ocaml et en PERL par l'intermédiaire de CIL (C Intermediate Language).

2.4.4. BOSSA : un hiérarchie d'ordonnanceur [RTS 05]

Un ordonnanceur est une application complexe puisqu'il est question de comprendre les conventions utilisées dans l'implémentation d'un système d'exploitation. Il est souvent nécessaire de bien connaître le noyau pour rédiger une nouvelle politique car celle-ci est souvent complètement indissociable du reste du noyau. Idéalement, pour pouvoir implanter de nouvelles politiques d'ordonnancement, l'ordonnanceur et le reste du noyau doivent être complètement dissociables mais parfaitement interfacés.

BOSSA propose un niveau d'abstraction spécifique au domaine de l'ordonnancement. Au lieu de faire directement appel à des fonctions de l'ordonnanceur (typiquement `schedule()`), les drivers font appel à un système d'événements.
En effet, la réécriture sur le noyau Linux consiste à analyser le code par l'intermédiaire de règles logiques afin de détecter les points d'ordonnancement (par exemple élection d'un nouveau processus, changement d'état d'un processus, etc.). Chaque point d'ordonnancement est remplacé par le lancement d'un événement qui dépend des conditions d'appel. Les événements sont interprétés par le RTS (Run - Time System) de BOSSA (voir figure 2.5.) qui invoque la fonction appropriée définie par l'ordonnanceur.

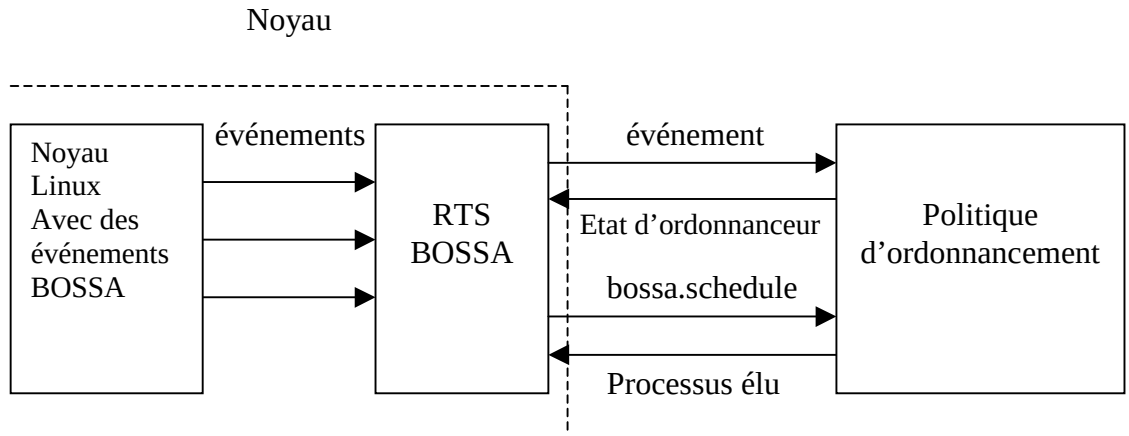


Figure 2.5. Architecture de BOSSA.

Pour régler les problèmes de cohabitation de programmes temps réel et non temps réel, BOSSA introduit la notion de hiérarchie d'ordonnanceurs. Un ordonnanceur de processus est un ordonnanceur classique qui gère les processus en vue de leur attribuer le processeur.

Un ordonnanceur virtuel est un ordonnanceur qui contrôle d'autres ordonnanceurs. Ainsi, on peut créer un ordonnanceur virtuel avec des ordonnanceurs fils auxquels il peut donner la main selon des critères bien définis (par exemple, priorité) ou bien selon une proportion (par exemple on donnera la main une fois sur trois à l'ordonnanceur fils numéro 1 contre deux fois sur trois à l'ordonnanceur fils numéro 2).

L'ordonnanceur du système aura donc une forme hiérarchique ou arborescente avec comme feuilles des ordonnanceurs de processus et comme nœuds des ordonnanceurs virtuels. Lorsque le RTS de BOSSA envoie un événement de type `bossa.schedule`, celui-ci est destiné au premier ordonnanceur dans la hiérarchie.

Du point de vue de la programmation, un ordonnanceur virtuel va avoir pour tâche principale de propager l'événement reçu vers l'ordonnanceur fils approprié et de mettre à jour l'état de l'ordonnanceur fils. Ainsi, les événements descendent la hiérarchie jusqu'à atteindre un ordonnanceur de processus qui va, en traitant l'événement, agir directement sur les processus qu'il gère.

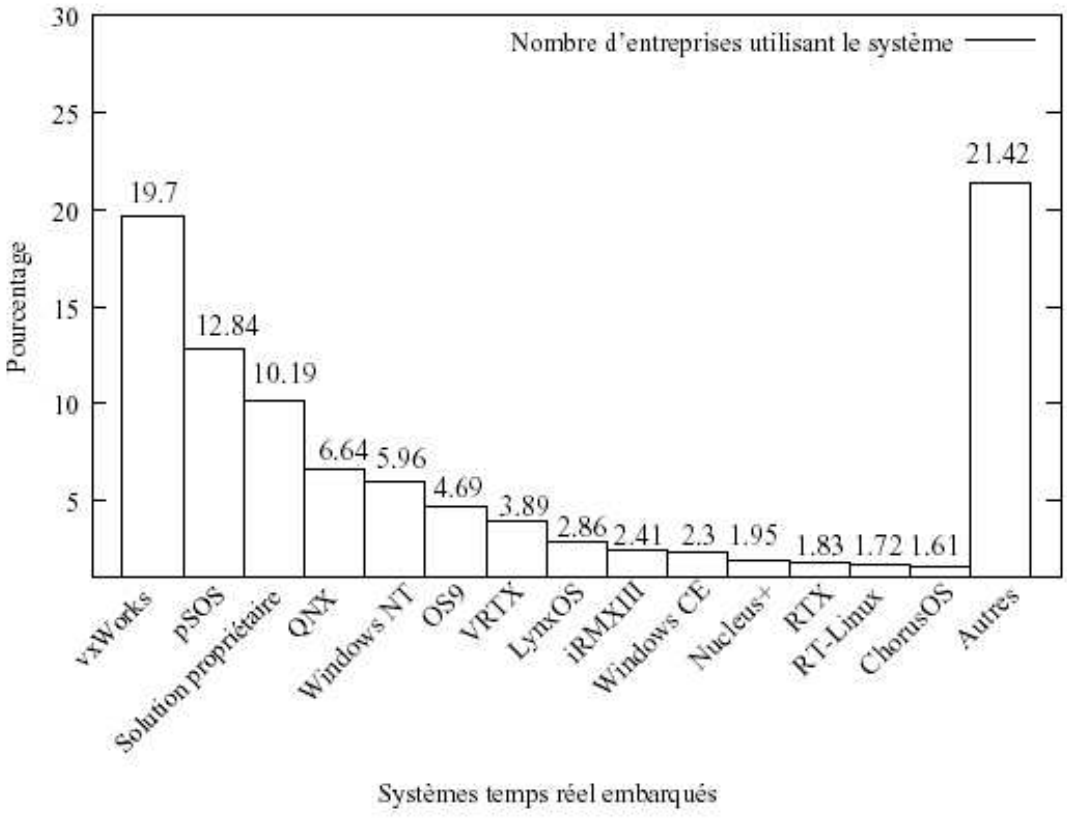
Conclusion

Petit soit-il, ce chapitre nous a permis de faire un tour d'horizon, sur les différents systèmes temps réel existants.
Nous avons insisté durant ce chapitre, sur les deux systèmes temps réel RTAI et VxWorks, afin de voir les différents mécanismes de communication entre tâches d'une part, et que nous n'avons pas d'ailleurs abordé lors de notre présentation des concepts temps réel au chapitre précédent ; d'autre part, nous avons présenté à travers ces systèmes l'API que doit offrir un exécutif temps réel, en vue de donner une idée sur celle que nous aurons à présenter dans les prochains chapitres.

Pour voir la retombée du système RTAI, nous avons préféré la plate-forme CLEOPATRE, qui prend effectivement RTAI comme support de base pour produire un haut niveau d'abstraction. Ceci donne alors l'idée de ce que peut être la mise en pratique d'un exécutif temps réel.

BOSSA est une plate-forme temps réel qui consiste quant à elle en une aide à l'écriture de nouvelles politiques d'ordonnancements. Nous avons jugé utile de la présenter, car l'avantage, de Bossa, est de pouvoir utiliser des ordonnaceurs temps réel, tout en ayant un système d'exploitation généraliste, qui offre plus de fonctionnalités qu'un système temps réel.

A titre indicatif, nous donnons dans ce qui suit, l'état du marché concernant l'utilisation des systèmes temps réel embarqués [SETR].



Bibliographie

[CLEO] : http://www.cleopatre.org

[DP 91] : Alain DORSEUIL et Pascal PILLOT- *Le Temps Réel En Milieu Industriel* (DUNOD 1991).

[RPG] : RTAI Programming Guide 1.0 (http://www.rtai.org).

[RTS 05] : BOSSA et le Concert Virtuel Réparti, intégration et paramètrage souple d'une politique d'ordonnancement spécifique pour une application multimédia distribuée
J. Cordry, N. Bouillot, and S. Bouzefrane.
13th International Conference on Real-Time Systems, Paris, France, April 2005.

[SETR] : Systèmes Embarqués Temps Réel : VxWorks
Université de Brest, IUP GMI, Singhoff Frank C-208.

Chapitre **3**

Conception de l'exécutif temps réel

Ce chapitre va présenter d'une façon détaillée la conception de notre exécutif temps réel. Nous allons rappeler puis élargir l'objectif de notre système, ce qui nous conduira à parler du cadre de travail en se référant aux objectifs fixés.
Nous allons aborder durant ce chapitre, les raisons qui nous ont conduit, à opter pour les méthodes et les politiques à adopter, en se basant bien sûr, sur les concepts que nous avons vus durant les précédents chapitres. Nous allons expliciter et expliquer les différents modules à développer, en les combinant aux stratégies que nous aurons à appliquer, et ce, d'une façon assez formelle.

Loin de l'architecture matérielle, ce chapitre va essayer de séparer au mieux l'abstraction physique sur laquelle notre exécutif s'y implantera, et l'abstraction qui englobera les politiques à adopter au sein même de notre exécutif temps réel.
Ce chapitre se veut être une spécification et une solution au but visé à travers une question posée indirectement et qui se répétera incessamment tout long de ce chapitre, qui est :
quoi faire ? ; il laisse cependant le soin au chapitre suivant, d'y apporter une réponse à la question du : *comment le faire ?*.

Nous n'allons pas adopter une méthode spécifique, pour concevoir notre système. Nous avons pensé au départ à utiliser UML temps réel, mais celle-ci semble plutôt donner des solutions et des formalismes sous le paradigme objet, et d'autant plus qu'elle est toujours au stade de la recherche. Les autres méthodes, comme SART n'est pas adaptée à ce genre de conception, puisqu'elle ne spécifie que les applications temps réel.

3.1. Objectif

Nous voulons concevoir un exécutif temps réel, simple dans ses mécanismes de base, pouvant mettre en évidence des principes et des politiques connus dans la littérature des systèmes d'exploitation, et pouvant ainsi intégrer un certain aspect pédagogique.

Partant de cette philosophie, cet exécutif sera à usage général, et s'adressera aussi bien aux développeurs d'applications temps réel, qu'aux étudiants et enseignants voulant promouvoir et/ou mettre en relief les concepts régissant ce système.

Nous devons penser à concevoir un environnement, offrant aux développeurs d'applications temps réel, une interface de programmation (API), avec un ensemble de services, à la fois restreint et indispensable.

3.2. Cadre du travail

Afin de rejoindre l'objectif de notre travail, nous allons considérer des principes et des politiques bâties autour d'une architecture monoprocesseur.

En vertu de ce qui a été présenté dans les précédents chapitres, nous allons principalement assurer la gestion des tâches périodiques et apériodiques en tenant compte de la faisabilité du système mais aussi de la synchronisation entre tâches.
Même si la spécification nous impose une certaine "précision" du système temps réel à savoir, son interaction avec le procédé, nous ne pourrons la formuler *à priori*, étant donné que l'objectif est de créer un exécutif à usage général.

La portabilité ne sera pas prise en compte, dans la mesure où, le domaine temps réel se limite suivant l'objectif visé, à implémenter des politiques et de voir les principes mis en œuvre, ce qui porte peu à les exporter vers d'autres architectures, et d'autant plus que ce domaine, impose généralement la prise en compte des caractéristiques de la machine sur laquelle il est implanté.

L'interopérabilité qui est liée à la question de coopération avec d'autres systèmes, ne sera prise en compte à notre niveau, et devrait être fixée par les applications temps réel, le cas échéant.

La modularité est un aspect fondamental à la conception de notre système, il permet d'une part de subvenir aux besoins fixés en terme de simplicité des mécanismes, et d'autre part, faciliter le développement et la maintenance de notre système.

Nous supposons alors connues les notions d'architecture de base d'un système informatique, et les techniques appliquées à celui-ci, ainsi que les notions présentées aux précédents chapitres.

3.3. Architecture générale du système

Partant de la notion du système, qui doit par définition avoir des entrées et des sorties, et en se référant à l'objectif fixé, l'idée générale de conception peut se résumer à créer un système qui a pour entrée les demandes de l'utilisateur et pour sortie, les traitements et les résultats effectués, suite aux demandes d'entrée.
Etant donné que le système est un exécutif temps réel, et sera implanté sur un système informatique, la première idée venant à l'esprit, étant de définir le matériel comme une ressource à exploiter, sur laquelle notre système s'y "reposera". L'utilisateur quant à lui, sera placé au niveau d'abstraction immédiatement supérieur à celui du système considéré. Autrement dit, adopter l'approche en couches.

L'utilisateur sera placé dans l'espace de l'exécutif, séparé néanmoins par une interface (API) à travers laquelle se fera la communication ou l'interaction. Cette philosophie est souvent utilisée dans les exécutifs temps réel, afin de garantir la modularité du système et *à fortiori*, minimiser le temps de commutations de contexte, cependant l'inconvénient est qu'un *bug* au niveau d'une tâche temps réel, entraînera le crash du système. Nous avons donc fait un compromis.

Nous allons expliciter les différentes parties qui composent notre exécutif, en adoptant une approche modulaire, et feront l'objet des sous paragraphes prochains.
La figure 3.1 illustre, l'architecture générale de notre système. Notons que l'interaction entre le matériel et l'exécutif sera vue plus en détail, dans le prochain chapitre.

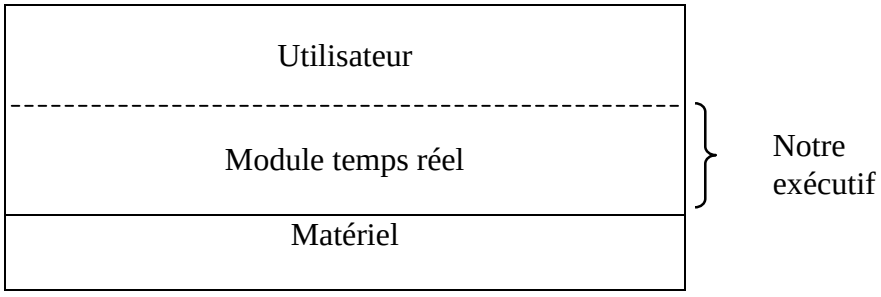


Figure 3.1. Architecture générale de l'exécutif temps réel.

3.4. Module temps réel

Avant de détailler le présent module, il est nécessaire de donner les grandes lignes auxquelles il fera référence.
Afin d'obéir à l'objectif du système à créer, qui consiste entre autres à fournir un environnement de développement d'applications temps réel, nous allons appliquer les concepts vus au premier chapitre.

Pour ce faire, notre exécutif doit alors offrir la possibilité de gérer des tâches périodiques à contraintes strictes ; pour cela, nous les associons à la politique d'ordonnancement RM (*Rate Monotonic*). Ce choix est justifié par la simplicité de son implémentation d'une part, et d'autre part, il s'avère comme étant l'algorithme optimal pour des tâches à échéance sur requête (voir théorème RM).

Par ailleurs, notre exécutif, doit également prendre en charge la gestion des tâches apériodiques. Ceci, en les gérant avec une tâche périodique particulière dite *serveur*. Pour ce dernier, nous utiliserons plus exactement le type ajournable, et ce, pour remédier à l'incompatibilité des rythmes d'arrivée, vue dans le cas d'un serveur à scrutation. Afin de simplifier l'ordonnancement des tâches apériodiques, celles-ci vont être gérées par la tâche périodique *serveur*, en FIFO.

La synchronisation entre tâches est essentielle à tout exécutif. Pour cela, nous allons faire munir notre système d'une gestion de sémaphores binaires. En effet, les sémaphores à comptes demandent nettement plus de complexité à les mettre en œuvre ; ce qui contredit notre objectif fixé en terme de simplicité des mécanismes, et d'autant plus que nous aurons à protéger que des sections critiques, et la réentrance du code ne sera que critique.

Comme énoncé dans la problématique de l'ordonnancement temps réel au premier chapitre, la synchronisation par sémaphores, doit être associée à un protocole de réservation de ressources, en vue de borner le temps d'attente aux ressources. Par conséquent, nous optons pour le protocole à priorité plafond, pour son évitement de l'interblocage ou l'étreinte fatale, mais aussi pour son évitement au phénomène d'inversion de priorités.

Intuitivement, nous pouvons décomposer le module temps réel, en deux sous modules :

- module des tâches ;
- module d'ordonnancement.

Ces modules sont issus, certes, d'une décomposition intuitive, mais à plus forte raison, d'une décomposition systémique, car, en considérant la partie temps réel comme un système, il en découle une décomposition en sous systèmes, qui doivent répartir ou prendre en charge les entrées et les sorties du premier module les englobant ; or, nous pouvons prendre en charge le flux d'entrée (du côté utilisateur) par le premier module (tâches), et après quoi, celui-ci agira sur le second (ordonnancement), pour produire ou provoquer par le biais du matériel, le flux de sortie. (voir figure 3.2.).

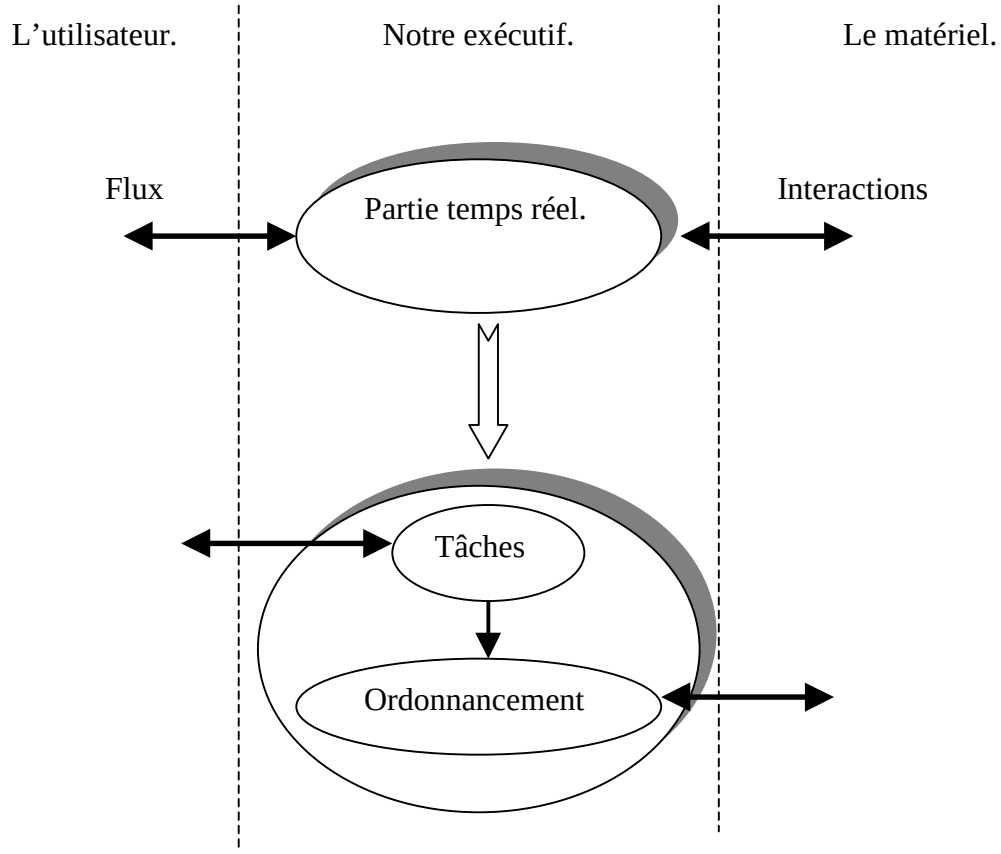


Figure 3.2. Décomposition du module temps réel.

3.4.1. Module des tâches

Ce module est censé interagir avec l'abstraction supérieure (utilisateur), en offrant à ce dernier la possibilité de manipuler l'entité *tâche*.
Trois classes de tâches sont à considérer, en vue de couvrir l'ensemble du module des tâches, à savoir, la classe des tâches périodiques, la classe serveur et la classe des tâches apériodiques (voir figure 3.3.). Chacune de ces classes est rattachée à une interface à travers laquelle elle peut être manipulée par la couche immédiatement supérieure (utilisateur).

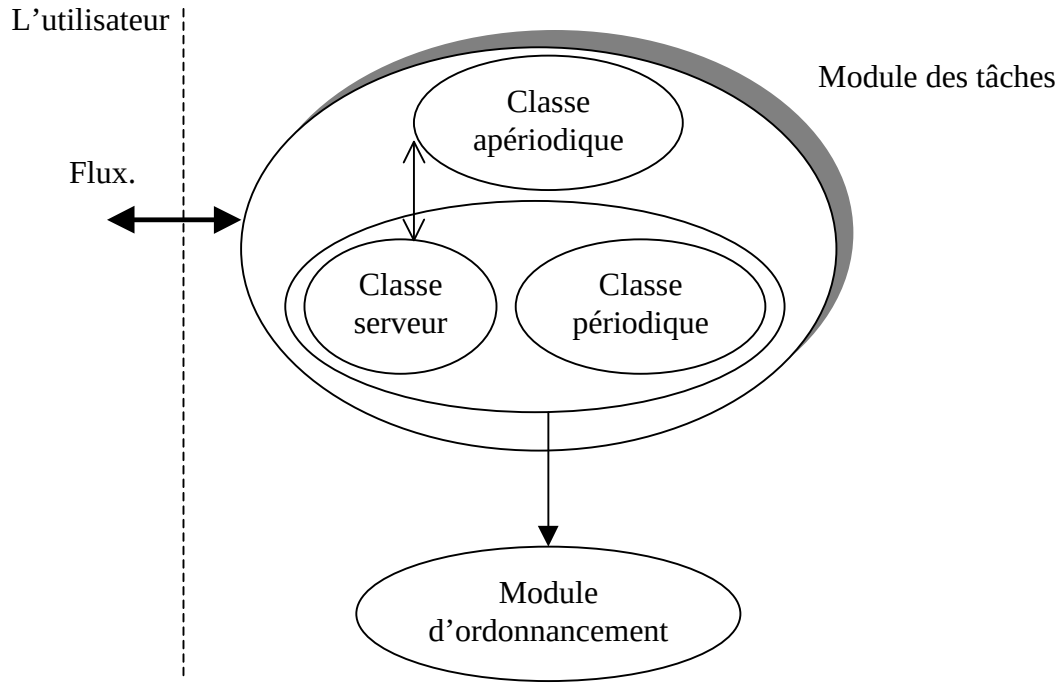


Figure 3.3. Les classes régissant le module des tâches.

Par ailleurs, pour qu'elle puisse agir sur le module d'ordonnancement, seules deux classes d'entre elles à savoir, la classe des tâches périodiques et serveur, seront manipulées par le module d'ordonnancement. En effet, seule la classe serveur, est responsable de l'ordonnancement des tâches apériodiques.

Pour qu'elles puissent être ordonnancées, les tâches périodiques, après leur avoir attribué une priorité[1](selon RM), seront mises dans une file d'attente, accessible et manipulée par le module d'ordonnancement. Il y va de même pour la classe serveur, mais la différence réside dans le fait que celle-ci devrait contenir une file propre à elle pour sa propre gestion des tâches apériodiques, autrement elle est considérée au même titre que la classe périodique du côté module d'ordonnancement.

Nous devons aborder la synchronisation dans la classe des tâches périodiques, étant donné le fait que le protocole à priorité plafond soit mis en œuvre. En effet, ce protocole suppose *à priori* la connaissance de chaque ressource susceptible à être utilisée, mais aussi du temps d'exécution de chaque section critique associée, en vue de borner le temps d'attente. Par conséquent, le module des tâches devra associer pour chacune des tâches de la classe périodique, une file de sémaphores qui vont être utilisés par cette tâche le cas échéant, et ce, avant même leur mise à disposition au module d'ordonnancement.

Le module des tâches doit être en mesure de satisfaire la faisabilité du système, autrement dit, il devra, de par les caractéristiques temporelles de la classe périodique et celle du serveur, déterminer la faisabilité du système en vérifiant plus particulièrement la condition d'ordonnancement RM vue au premier chapitre et ceci bien sûr, après avoir déterminé les priorités plafond de chaque ressource à utiliser, ainsi que leur facteur de blocage le cas échéant. Etant donné que la condition du théorème de faisabilité ou d'ordonnancement est suffisante mais pas nécessaire, il ne faut donc pas disant "stigmatiser" le fait d'une non-faisabilité du système, vu qu'il peut y avoir des configurations valides ne satisfaisant pas cette condition. Par conséquent, le module des tâches tâchera, de rendre cette condition comme étant optionnelle au système.

Le flux d'entrée et de sortie liant la couche utilisateur et le module des tâches, se résume essentiellement à la manipulation des tâches comme préalablement énoncé. L'utilisateur pourra à travers une interface (API), le liant au module des tâches, créer et supprimer des tâches dans les différentes classes, de pouvoir créer et supprimer des sémaphores et de les associer aux tâches correspondantes, mais aussi, avoir la possibilité de gérer le temps (horloge temps réel), dont il est généralement le rôle de chaque exécutif temps réel. Nous allons nous contenter pour un premier temps, de cette discussion sur les services offerts à l'utilisateur, car nous les verrons plus en détail, dans le prochain chapitre.

1. Dans notre système, les tâches de même période vont avoir une priorité égale.

3.4.1.1. Entité tâche

Dans notre système, elle représente l'unité de base d'exécution pour une quelconque manipulation, aussi bien par l'utilisateur que par le module d'ordonnancement.
Pour cela, elle introduit à travers son contexte, des informations nécessaires pour satisfaire de part et d'autre les côtés qu'elle serve. Ce contexte est vu en partie dans le premier chapitre, quant aux informations relatives au matériel, celles-ci seront traitées en détail dans le prochain chapitre.

Un bon nombre d'informations sur la tâche doit être, inclus pour prendre en charge les concepts temps réel. Pour cela, nous devons globalement faire munir le contexte de la tâche d'informations sur le matériel (contexte de la CPU), et d'ordonnancement. Pour les deux derniers, ceux-ci doivent être initialisés lors de la création de la tâche correspondante.

Nous n'allons pas dans ce cas inclure des informations relatives aux zones mémoires, dans la mesure où les tâches temps réel seront développées dans l'espace d'adressage mémoire du noyau et en outre, le code exécutable des applications ainsi que leurs données seront confondus. Ceci sous-entend, qu'il n y aura pas de réservation de mémoire d'une façon dynamique au sein du noyau, autrement dit, les applications seront considérées écrites et figées une fois pour toutes sans possibilité d'allocation dynamique de mémoire.
Quant à la zone de pile, celle-ci est réservée d'une manière statique, lors de la création de la tâche, et ne sera référencée ou intégrée que dans le contexte processeur.

Pour qu'elle puisse être utilisable par le module d'ordonnancement, nous allons faire ajouter au contexte de la tâche temps réel, les informations permettant un ordonnancement en RM. Pour introduire la notion de temps aux tâches, la date de réveil de la tâche, sa durée d'exécution, et sa période sont ajoutées conformément à la politique RM d'ordonnancement, ainsi que deux compteurs, pour la sauvegarde du temps d'exécution restant et la prochaine date de réveil. Et nous devons par ailleurs, avoir des informations sur le type ou la classe de tâche (périodique, apériodique, serveur), pour qu'elle soit suivant sa nature, être manipulée par le module d'ordonnancement.
Aussi, nous devons rajouter des informations sur les ressources susceptibles à être utilisées par la tâche, prétendant par cela, le facteur de blocage de la tâche, la file des sémaphores à utiliser et éventuellement le sémaphore pour lequel la tâche pourrait être en attente.
Etant donné que nous aurons à gérer les tâches à travers des files d'attente, elles devront alors bien évidemment avoir des informations comme le prédécesseur et le successeur dans la file.

Un ensemble d'états caractérise la tâche, des informations comme PRET, SUSPENDU, SEM, ATTENTE, qui désignent respectivement, la tâche prête, tâche suspendue, opère en section critique, et attente d'une ressource(sémaphore).

Chaque tâche possède son degré de priorité, mis à jour à chaque création d'une nouvelle tâche, disposant également d'une sauvegarde de la priorité initiale lors d'un éventuel héritage de priorité en section critique. (voir figure 3.4).

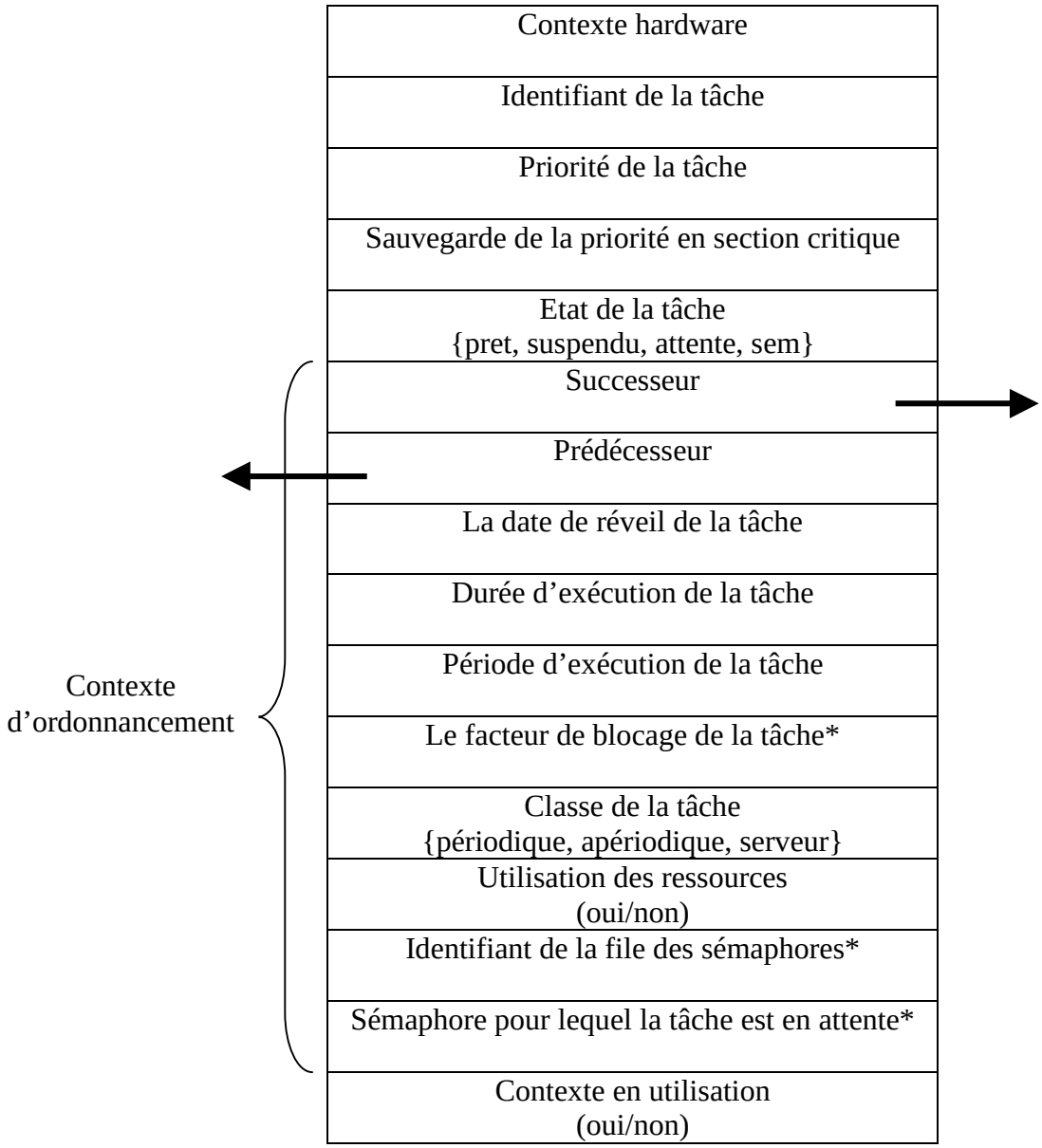


Figure 3.4. Contexte de la tâche temps réel de notre système.

* l'information est valide, s'il y a utilisation des ressources par cette tâche.

3.4.1.2. Entité sémaphore

Chaque ressource du système sera associée à un sémaphore. Comme nous l'avons énoncé, nous allons utiliser **des sémaphores binaires**. Les sémaphores seront présents dans le système s'il y a au moins une tâche susceptible de les utiliser, cette dernière devrait à ce moment-là spécifier *la file des sémaphores associés aux tâches*, préalablement créée.
Chaque tâche utilisant des ressources critiques, dispose de sa propre file de sémaphores.

Un sémaphore dans notre système, disposera d'autant d'entrées de prédécesseur et de successeur que de nombre de tâches pouvant exister dans le système. D'une part, ceci est dû au fait qu'un sémaphore peut être présent dans plusieurs files de sémaphores associés aux tâches, et d'autre part, il nous faudrait imposer cette manière de faire, en vue de pouvoir déterminer éventuellement les ressources communes entre les tâches qui seront utilisées dans le protocole à priorité plafond. Pour simplifier les choses, nous allons définir une bijection entre l'identifiant de la tâche et le numéro du successeur/prédécesseur correspondant dans le sémaphore.

Ainsi, si on suppose que la tâche *n*, voulant utiliser les sémaphores : S_i, S_{i+1}, …S_j (pour j>i) alors le $successeur_n$ de S_i serait égal à S_{i+1} . Le raisonnement est analogue jusqu'à S_j . (Voir figure 3.5).

Par ailleurs, le sémaphore dispose de deux autres entrées successeur/prédécesseur, mais cette fois-ci pour gérer une autre file de sémaphores lors de l'exécution des tâches. Cette file est **unique**, du fait que les sémaphores à gérer sont binaires. Cette file est mise en œuvre pour indiquer d'une part les sémaphores qui sont en cours d'utilisation, et d'autre part, permettre le recensement des sémaphores, en vue de la mise en pratique du protocole à priorité plafond.

Chaque sémaphore dispose de sa propre identité, ainsi que celle de la tâche qui le détient le cas échéant. Il dispose également d'une valeur indiquant si oui ou non le sémaphore est libre. Puisque les sémaphores sont associés au protocole à priorité plafond (PCP), nous devons alors faire munir le sémaphore conformément au PCP d'une priorité plafond.

Partant du principe qu'une section critique peut avoir plusieurs temps d'exécution (suivant le nombre d'itération ou de boucle, les conditions sur les différentes parties du code à exécuter…), nous allons rajouter un tableau de valeurs, contenant les durées d'exécution des sections critiques associées aux tâches. Ceci bien sûr sous la supposition qu'une tâche n'aura à utiliser la même section critique de différentes manières, autrement dit, pour des raisons de simplicité, durant toute l'exécution d'une tâche, si celle-ci utilise plusieurs fois une référence à une section critique, le code de cette dernière devrait être le même chaque fois que la demande par cette tâche soit faite.

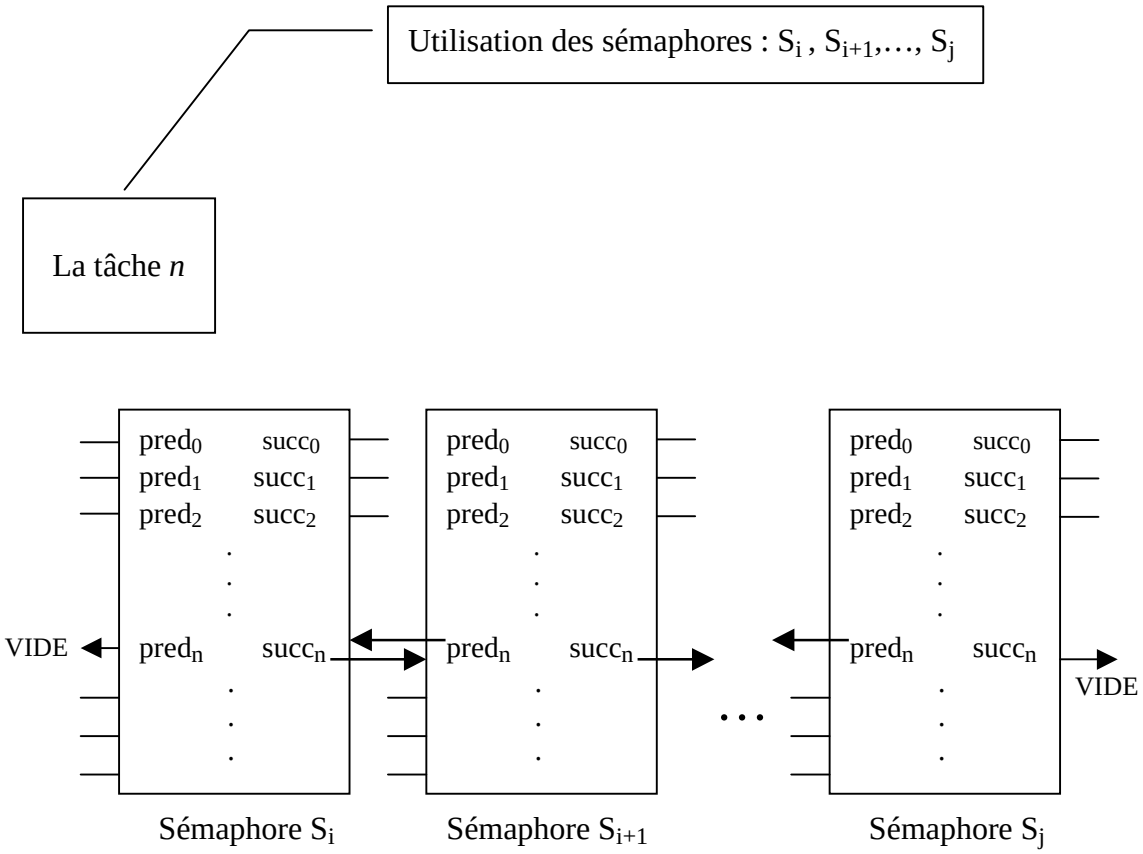


Figure 3.5. Représentation de la n^ième file des sémaphores correspondante à la tâche *n*.

3.4.1.3. Files d'attente

Nous avons évoqué lors des précédents points, l'utilisation des files d'attente, aussi bien pour les tâches (périodiques et serveur) que pour les sémaphores.
Pour mettre en place une file d'attente, nous allons la faire doter d'un entête contenant les informations comme la tête et la queue de la file, ainsi que le nombre d'éléments présents. Les éléments de chaque file devront avoir des informations comme prédécesseur, et successeur, afin d'avoir un double chaînage pour une manipulation rapide de la file.
Ce principe sera utilisé pour toutes les files d'attente du système, y compris celle des sémaphores associés aux tâches (vue au précédent paragraphe).
La figure 3.6. illustre le principe des files d'attente à utiliser.

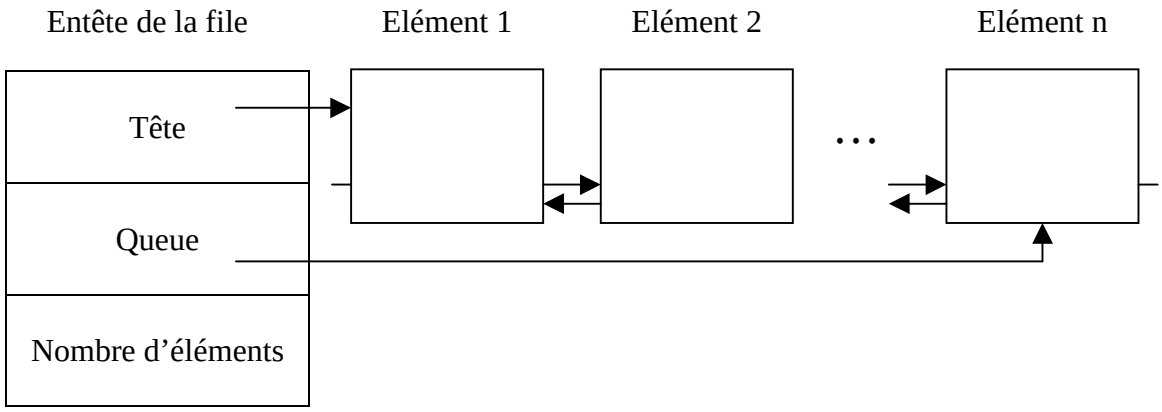


Figure 3.6. Principe de file d'attente de notre exécutif.

3.4.1.4. Pile des tâches apériodiques

L'arrivée des tâches apériodiques, comme son nom l'indique, se fait d'une manière imprévisible. Pour pouvoir les gérer, nous allons leur consacrer une pile, contenant les identifiants des tâches apériodiques arrivées.
Comme nous l'avons préalablement énoncé, la gestion des tâches apériodiques est assurée par la tâche serveur. Celui-ci pour des raisons de simplicité, les ordonnance en FIFO.
Le serveur manipule cette pile, en faisant l'extraction de l'identifiant le plus ancien arrivé. A vrai dire, nous avons abusé du terme pile, dans la mesure où les opérations sur une pile se résument à dépiler/empiler, ce qui correspond à la politique LIFO. La pile que nous aurons à utiliser pour les tâches apériodiques, sera similaire dans le principe à une pile ordinaire, mais la seule différence est que nous allons dépiler toujours depuis le bas de cette pile. Bien évidemment, ceci retombe directement sur le principe d'une file, mais c'est justement pour éviter toute confusion avec les autres files d'attente du système, que nous avons choisi de la baptiser autrement.
La figure 3.7. illustre ce principe.

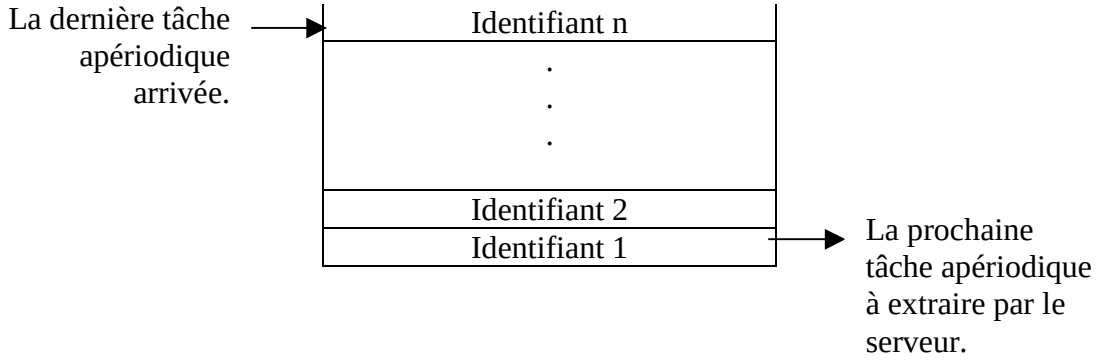


Figure 3.7. Pile des identifiants des tâches apériodiques.

3.4.1.5. Résumé

Pour récapituler, nous allons donner pour chacun, des éléments, que nous avons préalablement défini, les files auxquelles il appartient et/ou manipule.

- Les tâches périodiques et la tâche serveur (s'il y a gestion des apériodiques), vont être toujours associées à *une file d'attente des tâches périodiques prêtes*.
- Les tâches périodiques (hormis le serveur) susceptibles d'utiliser des ressources (sémaphores), vont chacune avoir *une file des sémaphores associés aux tâches*.
- Les sémaphores vont être associés à *une file de gestion de sémaphores*.
- Les tâches apériodiques vont être gérées par le serveur(ajournable), et se trouveront dans *la pile des tâches apériodiques*.

(Voir figure 3.8.).

Pour la dynamique régissant ces éléments, celle-ci est prise en charge par le module d'ordonnancement, qui fera l'objet du prochain paragraphe.

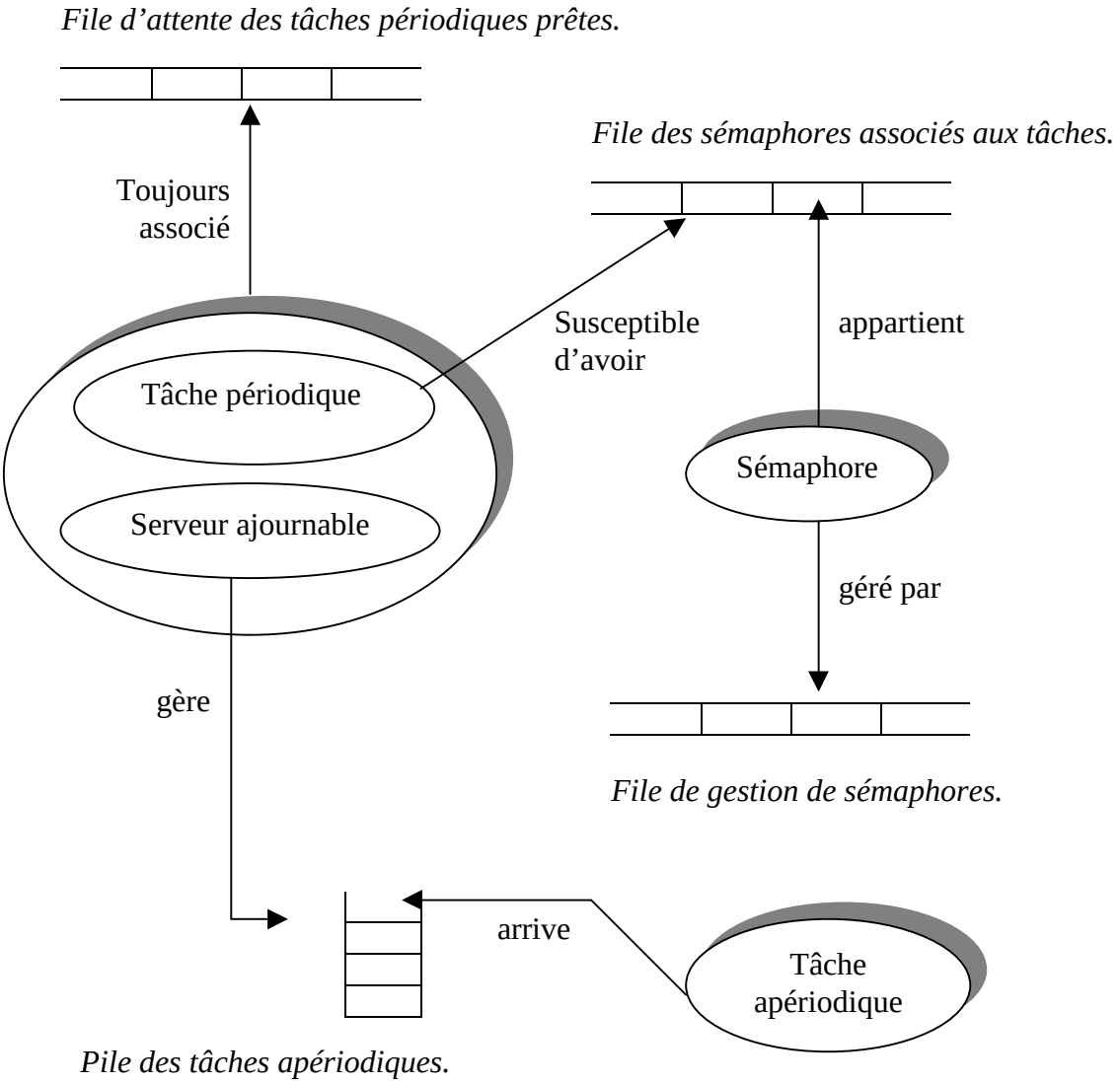


Figure 3.8. Entités du module des tâches.

3.4.2. Module d'ordonnancement

Ce module peut être considéré comme étant la partie "action" de la partie temps réel, et manipulera essentiellement les entités définies dans le module des tâches, conformément aux politiques et stratégies préalablement énoncées.
L'ordonnancement intervient alors qu'une fois les différentes entités du module des tâches sont mises en place.

Pour gérer la classe des tâches périodiques, nous allons dériver du module d'ordonnancement, une partie qui s'occupera de la recherche à partir de la file des tâches périodiques prêtes, la tâche dont la prochaine date de réveil est minimum avec la plus grande priorité. En d'autres termes, une partie exécutive de la politique RM. La tâche éventuellement trouvée par cette partie, sera considérée comme étant la tâche élue du système.

En parlant de classe de tâches, nous devons inclure les apériodiques. Nous avons envisagé d'ordonnancer les apériodiques par la tâche périodique *serveur*. Ceci revient à dire, que ce n'est qu'une fois la tâche serveur est élue, que celle-ci recherchera les tâches apériodiques à élire, car il y va de la responsabilité de la tâche périodique *serveur* d'ordonnancer les tâches apériodiques.

Nous avons vu dans l'ordonnancement RM, qu'il peut y avoir des creux temporels entre les tâches, autrement dit, la période durant laquelle aucune tâche n'est prête à être élue. Nous allons considérer une tâche particulière *nop_task* (pour : *no opreration task*) que l'ordonnanceur sollicitera, dès qu'il n y a aucune tâche périodique à élire.

Par ailleurs, pour pouvoir gérer les tâches périodiques conformément à la politique RM, nous devons faire "palpiter" l'ordonnanceur à des intervalles réguliers, assuré par l'horloge temps réel. Un compteur *absolu* va être dédié pour prendre en charge la notion de temps, et de pouvoir ainsi déduire les dates de réveil des tâches…

En outre, il faudrait faire munir le module d'ordonnancement d'une partie qui interagira avec le matériel afin de pouvoir faire charger la tâche élue. Cette partie va être vue plus en détail dans le prochain chapitre, étant donné qu'elle prend en charge la spécificité matérielle.

La figure 3.10 illustre la décomposition du module d'ordonnancement.

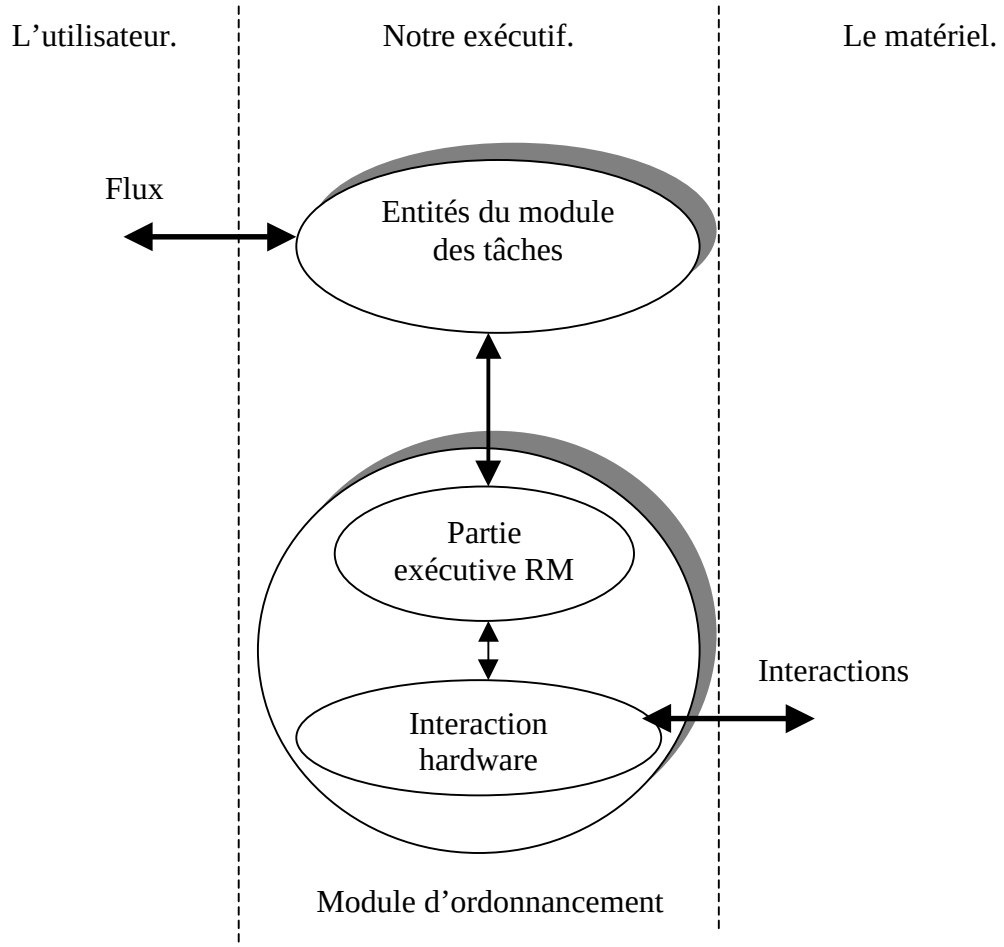


Figure 3.9. Décomposition du module d'ordonnancement.

3.4.2.1. Ordonnancement des périodiques

La partie exécutive RM dans la figure 3.9. est responsable de la gestion des tâches périodiques. A chaque top d'horloge émis par le matériel considéré, cette partie recherchera dans la file des tâches prêtes, la tâche la plus prioritaire suivant RM, léguant cette tâche à la partie *interaction hardware* le cas échéant ; le cas contraire, c'est la tâche *nop_task* qui sera élue ; ces actions se répéterons "indéfiniment" et à chaque top d'horloge. Nous pouvons donc dire que l'algorithme d'ordonnancement ainsi utilisé, est un algorithme *inline*.

Puisque nous avons abordé la synchronisation entre tâches périodiques, *la partie exécutive RM* devrait alors prendre en charge la gestion des tâches manipulant les sémaphores. Nous avons cité auparavant, que les tâches en attente d'une ressource critique (sémaphore) ont un attribut ou état que nous avons défini comme : ATTENTE. *La partie exécutive RM*, est responsable de rendre à nouveau prête (état : PRET) la tâche en attente d'une ressource libérée. Autrement dit, dès qu'il y a libération d'une ressource attendue par une tâche périodique, cette dernière va rentrer en concurrence avec les autres tâches, et ce, en modifiant son état ATTENTE en état PRET.

Pour être élue, la tâche périodique doit avoir un état parmi {PRET : prête, SEM : opère en section critique}, et avoir la plus grande priorité (attribuée lors de sa création suivant la politique RM), mais aussi avoir une date de réveil supérieure ou égale au *compteur absolu* associé à l'horloge temps réel préalablement défini. A noter, que la date de réveil de la tâche à élire sera mise à jour en ajoutant chaque fois la période de cette tâche. C'est ainsi que la tâche périodique (quelconque ou serveur) ayant pour période P une prochaine date de réveil $R=R+P$.
La capacité de la tâche périodique élue, est réduite d'une unité de temps à chaque top d'horloge, et ce n'est qu'après épuisement de cette capacité, que la date de réveil R est de nouveau déterminée.

Durant l'exécution d'une tâche périodique, cette dernière est susceptible d'être péemptée au profit d'une tâche plus prioritaire, il s'agit alors d'un ordonnancement préemptif basé sur la priorité, vu au premier chapitre. Dans notre cas, la préemption est due au fait qu'à chaque top d'horloge, il y a élection de la tâche périodique la plus prioritaire prête (avec état PRET ou SEM).

Comme nous l'avons déjà évoqué, l'ordonnancement des tâches périodiques demandant une ressource, est combiné avec un protocole de gestion de ressources (PCP) afin de borner le temps d'attente d'une ressource, par une tâche périodique(hormis serveur).

Le protocole à priorité plafond (PCP) va être combiné avec l'algorithme d'ordonnancement, et devra garantir la libération d'une ressource au bout d'un temps fini et défini ; et devra par ailleurs parvenir aux situations d'interblocage (étreinte fatale), mais aussi au phénomène d'inversion de priorités.
Nous allons faire munir *La partie exécutive RM*, de protocole à priorité plafond.
Pour ce faire, nous pensons activer le mécanisme, dès qu'il y a demande ou libération de ressources.
Lors d'une demande de ressource (sémaphore) **libre**, à ce moment-là, il est à recenser les ressources en cours d'utilisation, et en se référant à la priorité plafond de la ressource sollicitée, décider conformément au PCP (vu au premier chapitre), quant à l'attribution de la ressource libre, à la tâche demandante.
Ce protocole doit en outre, procéder à l'héritage de priorité, dans le cas bien sûr , où la tâche est préemptée par une autre tâche plus prioritaire et de plus, elle est en demande de la même ressource (sémaphore) déjà détenue par cette tâche préemptée.

Lors de la libération de la ressource, *la partie exécutive RM* munie du PCP, va restaurer la priorité, s'il y a eu héritage de priorité, et libère la ressource.

Dans le chapitre suivant, nous allons parler plus en détail de cette partie, et nous verrons la manière de l'implémenter.

3.4.2.2. Ordonnancement des apériodiques

Pour gérer les tâches apériodiques, nous devons les associer à une tâche périodique dite *serveur* qui se chargera durant sa capacité d'exécution, de faire exécuter l'une des tâches apériodique arrivée, suivant une politique propre au *serveur*.
Au sein du serveur, nous préférons gérer les tâches apériodiques suivant la politique FIFO, étant donnée sa simplicité à la mettre en œuvre.
Le serveur est une tâche périodique unique dans le système, nous allons opter pour le serveur ajournable, pour remédier à l'incompatibilité des rythmes d'arrivée vue dans un serveur à scrutation. Autrement dit, le serveur que nous pensons utiliser, devra maintenir sa capacité d'exécution, et ce, même si aucune tâche apériodique n'est présente dans le système, de cette manière, dès qu'il y a, arrivée d'une tâche apériodique, celle-ci est servie, sans pour autant attendre la prochaine date de réveil du serveur. Ceci renforce en quelque sorte la contrainte des tâches apériodiques.

En absence des tâches apériodiques, le serveur est mis à l'état SUSPENDU, par *la partie exécutive RM*, et dès qu'une tâche apériodique arrive, le serveur reprend l'état PRET, et pourra servir immédiatement la tâche apériodique en lui fournissant toute sa capacité. Cette dernière est exploitée par l'apériodique arrivée, mais elle devrait poursuivre son exécution lors de la prochaine échéance du serveur, dans le cas où la capacité du serveur s'avérerait insuffisante devant la capacité de la tâche apériodique.
Dans le cas où la tâche apériodique s'achève avant l'épuisement de la capacité du serveur, se dernier se mettrait immédiatement à l'état SUSPENDU, sauf s'il y a davantage de tâches apériodiques à servir.

Comme nous l'avons déjà mentionné, les tâches apériodiques arrivent dans *la pile apériodiques*, et à chaque top d'horloge et en présence du *serveur*, il y a vérification de cette pile. De cette manière, nous pouvons envisager d'appliquer la stratégie énoncée auparavant, qui consiste entre autres à servir la tâche apériodique dès son arrivée par le serveur ajouranble. *La partie exécutive RM* fera donc, l'extraction(depuis le bas) de l'identifiant à partir de cette pile consacrée aux apériodiques, dans le cas bien sûr, où le serveur serait en phase de repos (aucune apériodique n'est en phase de traitement).
Dans le cas contraire, *la partie exécutive RM* devra "attendre" la terminaison de la tâche apériodique en cours de traitement pour pouvoir faire l'extraction de l'apériodique suivante. Ceci nous renseigne sur le fait que les tâches apériodiques sont traitées d'une façon séquentielle par le serveur, suivant leur arrivée, et qu'il n y aura bien évidemment pas de préemption entre les apériodiques.

Nous pensons que le signal d'arrivée des apériodiques, devra être lié à un événement matériel, par le biais des interruptions, en vue de prendre en charge cet événement. Autrement dit, les interruptions matérielles vont nous servir, d'une part, de distinguer la nature de l'événement apériodique arrivé, et d'autre part, de pouvoir traiter l'événement dès son arrivée.
Pour cela, nous imposons le fait de connaître la durée de chacune des interruptions, afin de garantir le déterminisme du système en temps. Cette durée fera une caractéristique technique du système, et doit être prise en compte par les développeurs d'applications temps réel, pour notre exécutif.
Nous allons voir plus en détail la manière d'implémenter cette technique de mesure de temps d'interruptions, dans le prochain chapitre.

3.4.2.3. Résumé

Pour récapituler, le module d'ordonnancement intervient sur l'entité tâche, en changeant ses états dans l'ensemble {PRET, SEM, ATTENTE, SUSPENDU}, en vue d'élire une tâche périodique conformément à la politique utilisée.
Pour les tâches périodiques hormis le serveur, les états sont illustrés dans la figure 3.10.

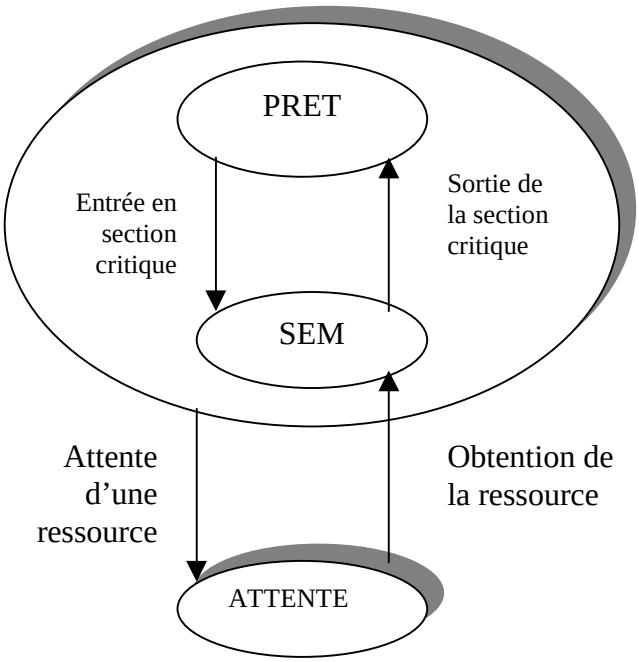


Figure 3.10. Etat d'une tâche périodique (hormis serveur).

La tâche périodique serveur prend uniquement deux états, puisqu'elle n'utilise pas de ressources (voir figure 3.11.).
Nous avons par ailleurs vu, que les apériodiques sont prises en compte par la tâche serveur, qui est la seule responsable de leur ordonnancement. La figure 3.12. résume d'une façon générale le fonctionnement.

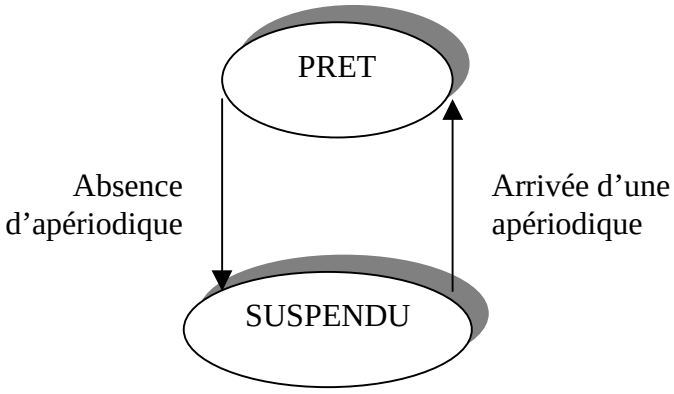


Figure 3.11. Etats du serveur ajournable.

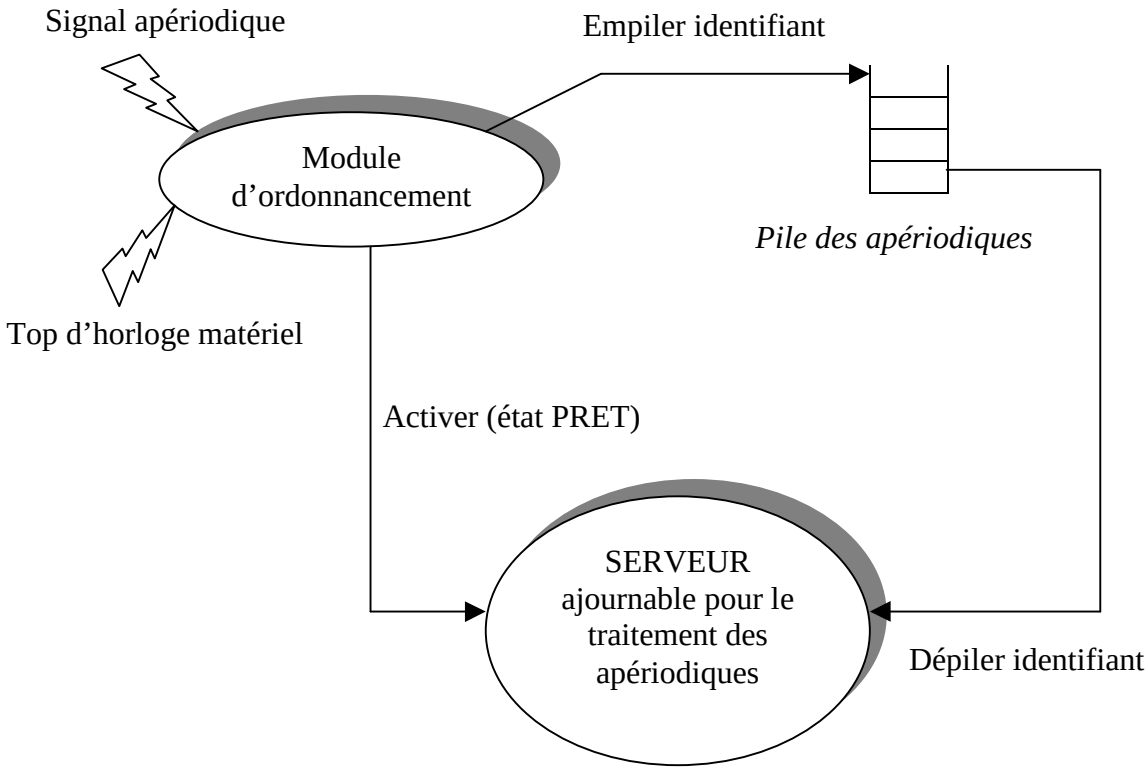


Figure 3.12. Traitement de l'apériodique.

Conclusion

En voulant concevoir notre exécutif temps réel, ce chapitre nous a permis de formuler les spécifications en se référant aux objectifs fixés.

Partant de ces derniers, nous avons mis en évidence les couches de notre système, à savoir le matériel, l'exécutif et l'utilisateur (non final). Nous nous sommes limités à la couche exécutif (partie temps réel), en tenant compte des interactions qu'il peut y avoir avec les autres couches.

Nous avons alors spécifié les stratégies et les politiques qui devront être mis en œuvre, pour atteindre l'objectif de notre travail. Nous avons adopté une approche modulaire, ce qui nous a conduit à subdiviser pour un premier temps le module temps réel en deux sous-modules, à savoir, le module des tâches et le module d'ordonnancement.
Pour ces derniers, et en vertu de ce qui a été présenté aux précédents chapitres, nous avons parlé des tâches temps réel, de leur ordonnancement, mais aussi de leur synchronisation par des sémaphores, tout en les englobant dans leur environnement modulaire.

Pour chaque module, nous avons considéré les différentes entités qu'il manipule, et nous avons discuté les grandes lignes concernant la mise en place de ces entités conformément aux stratégies et politiques choisies.

Avec ce formalisme assez exhaustif, nous sommes en mesure d'aborder la réalisation, qui mettra en pratique tout ce qui a été présenté dans ce chapitre, en détaillant davantage les modules, et la manière de les implanter.

Chapitre 4

Réalisation de l'exécutif temps réel

A travers ce chapitre, nous allons donner une solution pratique à la conception que nous avons présentée auparavant. Les modules, alors conçus, seront implantés d'une manière à préserver la sémantique ou la logique dont ils sont porteurs, tout en les dotant d'une caractéristique leur permettant de s'adapter à un environnement matériel, que nous allons devoir choisir.

Une partie sera alors consacrée à la manière, de faire intégrer des aspects à ces modules, tant au niveau matériel que logiciel. Ce dernier se focalisera essentiellement sur le codage, et servira de formalisme pour présenter chaque fois, les différents composants logiciel sous la forme du langage de programmation postulé.
La spécificité matérielle sera décrite avec cette même façon de faire, cependant, nous nous focaliserons sur les interactions bas niveau pour décrire les caractéristiques matérielles que les modules devront prendre en charge.

Les politiques et principes, régissant les modules conçus, seront considérés connus, pour n'en tenir compte que de leur réalisation. Ils seront présentés dans l'ordre du fonctionnement de l'exécutif, en d'autres termes, décrire la logique des modules, dans l'ordre chronologique de leur traitement par le système informatique considéré.

4.1. Choix de l'architecture matérielle

Durant toute la conception de notre exécutif temps réel, nous avons supposé les politiques et stratégies bâties autour d'une architecture monoprocesseur. Le choix est porté sur ce dernier dans ce présent chapitre et devra, rejoindre l'idée de base et les objectifs fixés.
Si l'exécutif est conçu pour une utilisation spécifique, nous choisirions à ce moment-là une architecture aussi particulière, voire dédiée. C'est ainsi que des mico-contrôleurs, et des processeurs enfouis sont destinés le plus souvent aux systèmes embarqués, comme les distributeurs automatiques de billets, les robots autonomes…
Dans notre cas, l'exécutif sera comme préalablement énoncé, à usage général et nous devons, choisir une architecture hardware sur laquelle nous pouvons garantir le fonctionnement des modules, du temps réel, conçus.
Pour notre exécutif, nous allons choisir une architecture basée sur le x86 d'Intel, pour sa prise en compte des conditions ou critères des modules à implanter, mais aussi et surtout pour sa forte disponibilité sur le marché et son large utilisation par le grand public.
Nous jugeons essentiel de présenter cette architecture à mesure que nous progressions dans ce chapitre, afin d'éclaircir davantage la prise en charge des interactions matérielles se trouvant au bas niveau de notre système. Cependant, nous supposons connu l'architecture du 8086 d'Intel, son jeu d'instruction, mais également de l'architecture de l'IBM PC, et nous nous intéresserons, qu'aux spécificités matérielles du PENTIUM (586 d'Intel) accessibles par programmation.

4.2. Langages et environnement de programmation

Pour l'étape de codage, nous pensons à utiliser deux types de langage, suivant la nature de la partie à programmer. En effet, Il est fastidieux voire impossible, de programmer l'exécutif tout entier en assembleur. Ni même encore en langage haut niveau, étant donné que nous aurons à programmer des routines tenant compte des spécificités matérielles.
Nous avons choisi le langage assemblage Masm (de Microsoft) pour écrire les fragments nécessitant un accès bas niveau, mais aussi pour l'optimisation du code.
Nous n'avons pas conçu notre système sous le paradigme orienté objet ; nous comptons plutôt adopter une approche procédurale pour réaliser notre exécutif.
Le langage haut niveau sera donc le langage C (compilateur de Borland), et servira à implanter les fonctions nécessaires, pour la mise en pratique des modules que nous avons préalablement définis, ainsi que les utilitaires nécessaires aussi bien au niveau exécutif qu'au niveau environnement de programmation. Pour ce dernier, il s'agira de l'environnement Win32, étant donnés les logiciels de programmation à utiliser.
Bien évidemment, l'environnement de programmation ainsi que les langages n'influent point, quant à la manière dont l'exécutif sera mis en œuvre, ce n'est donc qu'une question de convivialité du côté programmeur de l'exécutif temps réel.
Nous supposons, ces langages connus, ainsi que leur manipulation dans l'environnement dans lequel ils opèrent.

4.3. Parties de l'exécutif

Pour rejoindre l'idée de notre conception, et en se basant sur l'architecture matérielle choisie, nous allons procéder à la réalisation des différentes parties de notre exécutif.
Pour ce faire, nous allons suivre l'idée de conception, en implémentant les différents modules mais cette fois-ci dans le sens inverse, en terme d'ordre d'appartenance aux niveaux d'abstraction régissant notre système. Autrement dit, nous allons réaliser en premier lieu, les parties proches du matériel, pour pouvoir ensuite arriver aux couches supérieures de notre système. C'est souvent l'approche adoptée lors de la réalisation en couches, dans les noyaux de systèmes d'exploitation.

En terme d'implantation de l'exécutif, la philosophie à adopter consiste à considérer des parties concaténées (noyau monolithique) sur disque, lesquelles seront chargées à la demande et une fois pour toutes. Nous allons faire charger en mémoire l'exécutif en parties, ceci afin de permettre la prise en considération des états de la machine, d'une part, et d'autre part, pour permettre aisément une éventuelle extension ou développement au sein de ces parties.

C'est alors que vient l'idée de réaliser *la partie démarrage* de notre exécutif. Cette partie sera bien sûr, le plus bas niveau de notre système, et fournira principalement après une initialisation de la machine, un environnement assez riche en matière de services système élémentaires, en vue de développer les couches supérieures. Ces dernières porteront principalement en contexte, sur les modules que nous avons mis en évidence dans le chapitre conception, s'ajoute bien sûr, l'interface utilisateur (API). Ces couches seront regroupées dans une seule partie, que nous allons définir comme étant *la partie opérationnelle*. (voir figure 4.1.).

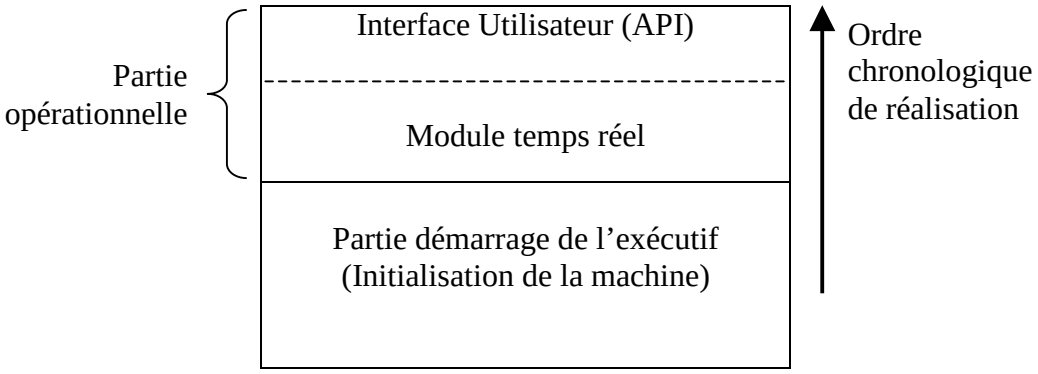


Figure 4.1. Les deux grandes parties à réaliser pour notre exécutif.

4.4. Réalisation de la partie démarrage

Cette partie a pour objectif d'initialiser la machine sur laquelle tournera notre exécutif et de fournir un ensemble assez riche de fonctions, en vue de les utiliser lors de la programmation en haut niveau. Cette partie, et suivant l'architecture du PENTIUM ainsi que de l'IBM PC, sera subdivisée, en trois sous parties, à savoir le boot, la partie 16-bit (mode réel du PENTIUM) et la partie 32-bit (mode protégé du PENTIUM).

Il faut savoir que pour programmer un noyau, la signification part d'elle-même. Partant de "rien", nous devons programmer des routines élémentaires qui nous faciliteront la manipulation des données en mémoire, des chaînes de caractères mais aussi la manipulation des E/S de périphériques. En d'autres termes, reprogrammer les fonctions (librairies) que nous incluons habituellement et d'une façon évidente dans un langage haut niveau, lors de la programmation d'applications.

4.4.1. Le boot

Il faut savoir que lors de la mise sous tension de la machine IBM PC, le processeur exécute les premières instructions (en mode réel ou 16-bit) à partir d'une mémoire non volatile (ROM, FLASH…). La routine à laquelle appartiennent ces premières instructions exécutées en premier lieu est dite *bootstrup*. Cette dernière est une partie d'un programme appelé BIOS, qui s'occupe de la configuration et de l'initialisation des périphériques de base présents sur la machine (carte graphique, disques, clavier…).

Après quoi, le BIOS suivant sa configuration (sauvée au CMOS) charge en mémoire (0000 :7C00h) le premier secteur du disque (secteur physique 0) spécifié dans la configuration. Le BIOS vérifie la signature du secteur (0x55AA) si elle est présente à la fin de ce secteur. Si tel est le cas, il lègue le contrôle au programme ainsi chargé depuis le secteur 0. Ce programme est appelé *zone d'amorce* ou *boot*.

Pour programmer le *boot*, celui-ci devrait être écrit en assembleur, étant donné que nous devons prendre en charge la taille de ce programme. En effet, la taille d'un secteur physique est de 512 octets, et nous serons obligés d'en tenir compte à ne pas dépasser cette taille, et d'inscrire à la fin de ce secteur, la signature dont nous avons parlé plus haut.

La mission du *boot* dans notre système sera de charger d'autres secteurs en mémoire et de leur léguer par la suite la main. Autrement dit, charger en mémoire et à partir du disque de démarrage, une partie de notre noyau (partie 16-bit). Nous allons en quelque sorte refaire ce que le BIOS a fait lors du chargement du secteur de *boot*, mais dans notre cas, nous allons devoir charger un nombre plus important de secteurs (18 secteurs successifs), dans un endroit différent en mémoire (par convention à 2000h :0100h) et depuis une adresse physique différente sur disque (secteur physique 1 succédant le *boot*).

Pour ce faire, nous allons utiliser les routines BIOS, pour pouvoir lire des secteurs physiques à partir du disque de démarrage (ah=2 int 13h), de rebooter dans le cas où les secteurs à lire s'avéreraient défectueux (int 19h), d'effacer l'écran et d'afficher un message en cas de réussite/erreur(ah=8 et ah=0eh int 10h), et d'attendre dans le cas d'une erreur, la confirmation de l'utilisateur (touche clavier), pour rebooter (AX=0 int 16h) après avoir émis un bip sonore d'une durée de 2 secondes (respectivement, envoyer une valeur paire au port 61h et (ah=86h int 15h)).

Notons que pour utiliser l'interruption 13h (lire des secteurs physiques) dans notre programme de boot en assembleur, il faudrait spécifier l'identité du lecteur de démarrage (valeur mise dans le registre DX du processeur lors du chargement du secteur de *boot*). Notre programme de *boot* sauvera cette valeur pour une éventuelle utilisation (par convention à 1000h :0000).
De plus, il est à spécifier le numéro du cylindre disque (registre CH=0), le numéro de la tête du disque(DH=0), l'identité du lecteur (DL=0 disquette), le numéro du secteur (CL=2), le nombre de secteurs à lire (AL=18), spécifier la fonction d'écriture (AH=2) mais aussi l'adresse mémoire dans laquelle, les secteurs vont être chargés (ES:BX=2000h :0100h par convention). Pour le lecteur intéressé par d'autres routines BIOS, il peut consulter la référence [Ms dos].

Si aucune erreur n'est survenue, lors du chargement de la partie 16-bit (18 secteurs physiques chargés), le *boot* fera après cela, un saut vers cette dernière (2000h :0100h), pour lui léguer le contrôle.

4.4.2. Partie 16-bit (mode réel)

Comme nous l'avons vu, la partie *boot* est conditionnée par sa taille (512 octets). Nous devons alors programmer une autre partie, qui assurera d'une part, l'initialisation de la machine, en fournissant avant ceci, un nombre de services systèmes (librairie) nécessaires, en vue de les utiliser dans un langage haut niveau, et "d'accélérer" ainsi le développement de notre exécutif temps réel, et d'autre part, faire charger la partie 32-bit (mode protégé).
Rappelons-le, qu'il appartient au secteur de *boot* de charger la partie 16-bit, et ce, à partir du secteur physique 1, succédant immédiatement le *boot*.
Pour initialiser la machine, nous devons au préalable et comme nous l'avons déjà souligné, disposer de fonctions (utilitaires) qui nous permettra un contrôle aisé de données comme les chaînes de caractères… , et de pouvoir manipuler dans un environnement haut niveau des fonctions comme l'affichage…
Pour cela, nous devons définir un ensemble de fonctions, à la fois restreint et indispensable et qui fera l'objet du sous paragraphe suivant. Puis, nous allons aborder la manière d'initialiser la machine, moyennant notamment ces fonctions utilitaires que nous avons implantées.

4.4.2.1. Fonctions utilitaires

4.4.2.1.1. Manipulation de chaînes de caractères

Pour manipuler les chaînes de caractères, nous allons réécrire les deux fonctions :

```
const char far *strcpy(char far dest, char far src) ;
char far *strcat(char far dest, char far src) ;
```

Ces deux fonctions ressemblent à celles définies dans la librairie standard du C (string.h), qui consistent respectivement à la copie d'une chaîne de caractères, et de la concaténation.

4.4.2.1.2. Manipulation de données en mémoire

Pour déplacer des données en mémoire nous allons réécrire la fonction :

```
char far *memncpy(char far *src, char far *dest, int nbr) ;
```

Copie un bloc de mémoire de taille *nbr* de l'adresse *src* vers *dest*.

4.4.2.1.3. Conversion Décimale - ASCII

Pour pouvoir convertir un nombre (non signé) en une chaîne de caractères (ASCII) en vue d'être affichée :

```
Const char far * decasc(unsigned long i) ;
```

4.4.2.1.4. Conversion Hexadécimale - ASCII

Pour pouvoir convertir une chaîne de caractères (ASCII) en valeurs hexadécimales (ASCII) nous allons avoir recours à la fonction :

```
const char far * hexasc(const char far * ptr, int cpt) ;
```

Elle nous va faire donc renvoyer une chaîne de caractères, dont le contenu sera en ASCII, et la sémantique en hexadécimale. Notons que le nombre de caractère à convertir est défini dans *cpt*.

4.4.2.1.5. Réécriture des fonctions *vspintf, printf*

Pour afficher une chaîne de caractères sur la sortie standard, nous allons définir trois fonctions :

```
const char far * vprint(char far *chaine, char far *prm) ;
void print(char far *str,...) ;
void printk(char far *chaine, unsigned char couleur) ;
```

- La fonction *vprint* ainsi définie, permet de scanner la chaîne de caractères *chaine*, et de recenser les endroits où se présente les paramètres, puis remplacer ces derniers par les valeurs récupérées, depuis le contenu de l'adresse de pile *prm* correspondant au type d'argument ainsi scanné. Elle ressemble un peu à la fonction définie dans la bibliothèque standard C (*stdio.h*) sous le nom *vsprintf*. Cependant notre fonction est beaucoup plus simple et nous donnons ci-après les paramètres ou arguments reconnus.

%d : valeur en décimal non signée.
%x : valeur en hexadécimal avec 4-digit.
%X : valeur en hexadécimal avec 8-digit.
%p ou %P : adresse mémoire sur 32-bit.
%s : chaîne de caractères.
%c : caractère.
%% : caractère '%'.

Rappelons que la convention du langage C, fait que les paramètres passés pour une fonction quelconque soient toujours empilés avant l'appel de fonction, dans le sens inverse au sens que nous avons défini. Par exemple, en faisant, appel à la fonction *fct(arg1,arg2,arg3) ;* l'argument *arg3* sera le premier empilé, puis suivi de *arg2*, et enfin *arg1*, avant bien sûr l'appel à *fct.*
Il est important de rappeler cette convention, dans la mesure où notre fonction *vprint*, va dépiler les valeurs passées en argument(trouvées suivant l'ordre et à partir de l'adresse *prm*), et de les remplacer à l'intérieur de la chaîne scannée *chaine*, aux endroits où se présentent les formats présentés précédemment.

- La fonction *print*, préalablement définie, aura pour mission, d'afficher sur la sortie standard (écran), la chaîne de caractères *str*, et ce, en tenant compte des éventuels paramètres pouvant être passés en argument. Elle ressemble à la fonction définie dans la bibliothèque standard C (*stdio.h*) sous le nom de *printf.*
Notre fonction *print* utilisera la fonction *vprint*, en lui passant le premier paramètre *str*, mais également l'adresse des éventuels paramètres. Cette adresse serait égale à l'adresse du pointeur *str* (c-à-d son adresse dans la pile) à laquelle on rajoute 4, car la taille d'une adresse mémoire de type FAR (segment : offset), prend 4-octet ou caractères. Donc le paramètre *prm* se voit alors attribué l'adresse du premier paramètre de la fonction *print* éventuel (correspondant au trois points de l'argument *print*).
La fonction *print* devra bien évidemment appeler la routine d'affichage que nous allons voir dans le prochain point.

- La fonction *printk* écrite en assembleur, est disant le bas niveau de la fonction *print*, car elle est responsable d'appeler la routine BIOS d'affichage (int 10h), en vue d'afficher la chaîne de caractères *chaine* avec une couleur ou attribut *couleur*.
En premier lieu, elle détecte le mode dans lequel se trouve l'affichage (graphique ou texte), si c'est le mode texte, elle procède à l'affichage caractère par caractère de la chaîne *chaine*, en vérifiant à chaque fois s'il s'agit d'une tabulation ou un saut de ligne, car pour les deux cas, un traitement disant spécifique est entamé. Les caractères sont affichés grâce au service (int 10h) BIOS (ah=9) mais aussi avec (ah=0eh) (voir [Ms dos] pour plus de détails).
Si c'est le mode graphique, nous allons utiliser à ce moment-là que la fonction (ah=0eh), en vérifiant là aussi la présence de tabulation et le saut de ligne.
Notons que le saut de ligne et la tabulation sont conformément aux caractères définis en langage C (c-à-d respectivement '\n' et '\t').

4.4.2.1.6. Réécriture de *Putchr*

Nous allons également rajouter une fonction écrite en assembleur :
`void putchr(unsigned char caractere, unsigned char couleur, int colonnes) ;`
qui permet, d'afficher un caractère *caractere* avec une couleur ou attribut *couleur*, en le ressassant dans un nombre de colonnes *colonnes.*

(Les fonctions ainsi définies sont dans le source .\utils\16\
leurs prototypes seront dans le fichier .\include\kernel.h
les couleurs ou les attributs sont définis dans le fichier .\include\couleur.h)

4.4.2.2. Initialisation machine avec *DTCT16*

Nous pensons implanter une fonction *DTCT16* appelée depuis la fonction *main* , et qui aura pour rôle de détecter le processeur présent dans le système (sa fréquence d'horloge …), la quantité de mémoire vive (RAM) disponible et les disques (lecteurs disques durs et disquesttes) présents dans le système en utilisant les routines BIOS . Nous devons au préalable initialiser les segments ainsi que la zone de pile, pour pouvoir ensuite appeler la fonction *main* (se trouvant dans .\16\kr16.cpp). Nous allons donc écrire dans un fichier en assembleur (se trouvant dans .\16\start16.asm) effectuant cette initialisation, et qui sera concaténé lors de l'édition des liens au précédent (fonction *main*), et sera considéré alors comme le *startup* de la fonction *main* du langage C.

Les informations ainsi collectées seront sauvegardées, dans une structure *superstruct* (adresse par convention 1000h :0101h), puis affichées sur l'écran.

Elle mettra aussi en place (par convention à 6000h :0000h) le premier fragment d'un gestionnaire que nous avons baptisé *basculeur*, qui assurera le retour au mode protégé (32-bit), lors d'une sortie du mode 32-bit vers le mode 16-bit.

Elle fera depuis le disque de démarrage, une copie temporaire de la partie 32-bit vers l'adresse 9000h :0000 (par convention).

Puis, elle active la ligne A20, en vue d'accéder aux 64Ko de la mémoire haute. Suite à cela, elle pourra déplacer la copie temporaire 32-bit du noyau se trouvant à 9000h :0000, vers l'adresse physique définitive 0x100000.
Une fois ceci fait, elle fera l'initialisation de la table des segments (GDT) (par convention à 4000h :0000), en vue de passer en mode protégé. Ce dernier sera activé par la suite, pour léguer ensuite le contrôle à la partie 32-bit chargée.

Nous allons détailler davantage les sous routines auxquelles, la fonction *DTCT16* fait référence durant son exécution, dans les prochains sous paragraphes qui suivront.

4.4.2.2.1. Détection de la CPU

Pour pouvoir détecter la CPU (identité et fréquence d'horloge), nous allons développer une fonction en assembleur, dont le prototype en C est :

```
const char  far * init_cpu(char nbr);
```

Cette fonction va retourner suivant le paramètre *nbr*, l'identité du constructeur de la CPU, son numéro d'identification, ou sa fréquence d'horloge. Respectivement, les valeurs de *nbr* seront 0,1, ou 2.

- L'identité du constructeur est obtenue grâce à l'instruction du PENTIUM : `cpuid`, en forçant avant cela, le registre EAX à zéro. La chaîne de caractères retournée contenant l'identité du constructeur serait la concaténation des valeurs des registres EBX,EDX,ECX.
- En mettant la valeur 1 dans EAX, l'instruction `cpuid` renvoie dans EAX le numéro d'identification de la CPU.

Pour détecter la fréquence d'horloge de la CPU, nous allons avoir recours à l'instruction du PENTIUM : `rdtsc`, qui fournit le nombre de cycles passés, et ce, depuis la mise sous tension de la CPU.
Nous devons en premier lieu, inhiber toutes les interruptions en ne laissant que celle du timer. Pour ce faire, nous allons reprogrammer le PIC 8259A (maître et esclave) en masquant les interruptions arrivant sur les lignes IRQ1…IRQ15 et ne laisser que IRQ0 (la programmation du PIC sera vue plus en détail dans les prochains paragraphes).
En vue de mesurer la fréquence d'horloge de la CPU, nous allons effectuer deux prélèvements des cycles passés grâce à `rdtsc`. En effectuant la première mesure, nous allons avoir le nombre de cycles retournés par `rdtsc`, dans les registres EDX : poids forts EAX :poids faibles. Après quoi, nous allons sauver temporairement ces valeurs dans EBX et ECX respectivement.

Nous allons séparer le deuxième prélèvement du premier, avec l'instruction `hlt`. Cette dernière consiste à bloquer la CPU, tant qu'il n y a pas d'interruptions. Or nous avons inhibé toutes les interruptions, sauf celle du timer, ce qui revient à dire que, la CPU se bloquera durant un top d'horloge du timer. Ce dernier est programmé par le BIOS, de façon à fournir environ 18,206 tops/seconde. En faisant le deuxième prélèvement, nous allons lui retrancher la première valeur mesurée (sauvée dans EBX, ECX), et ce, pour avoir le nombre de cycles passé depuis le premier prélèvement, autrement dit, le nombre de cycles passés depuis un top d'horloge. Cette différence sera divisée alors par (1/18,206), pour avoir la fréquence en Hz puis sur 10^6 pour avoir la fréquence en Mhz.

4.4.2.2.2. Mémoire étendue et conventionnelle

Pour pouvoir détecter la quantité de mémoire disponible, nous allons implanter la fonction écrite en assembleur dont le prototype en C est :

```
unsigned long init_ram(short int prm);
```

Cette fonction nous renvoie suivant la valeur de *prm*, la quantité de mémoire étendue (RAM) ou la quantité de mémoire conventionnelle. Respectivement les valeurs 0 ou 1 du paramètre *prm*.

- Pour avoir la quantité de mémoire étendue présente dans le système, nous allons la lire à partir du CMOS, à l'emplacement 31h(poids forts) et 30h(poids faible de la quantité de mémoire). Sachant que pour lire/écrire une valeur se trouvant à l'emplacement quelconque *y* du CMOS, il suffit d'envoyer la valeur *y* au port 70h, puis de lire/écrire le contenu depuis/dans le port 71h.
- Idem pour la mémoire conventionnelle, il suffit de la lire à 16h(poids forts) et 15h (poids faibles de la mémoire conventionnelle).

4.4.2.2.3. Détection des disques

Nous devons recenser les disques attachés à notre système, en vue de pouvoir y accéder éventuellement. Pour ce faire, nous allons utiliser la routine BIOS (int 13h).
Pour recenser les unités de disquettes disponibles, nous devons mettre dans le registre DL, la valeur zéro indiquant ce type d'unité, la valeur de la fonction quant à elle, sera de 8 placée bien sûr dans AH. Pour d'amples informations, le lecteur peut consulter [Ms dos].
Lors de cette détection, nous allons principalement sauver dans la structure, les informations comme le nombre de cylindres, de secteurs, et de têtes.
Idem pour les disques durs (avec DL=80h).
(La détection des disques, se trouve dans le fichier .\include\disq16.h)

4.4.2.2.4. Sauvegarde d'informations hardware

Nous devons sauvegarder les informations collectées sur le matériel (CPU, mémoire …), dans un emplacement mémoire que nous allons devoir définir comme réservé ; comme nous l'avons déjà énoncé et par convention, nous utiliserons pour cette sauvegarde une zone mémoire à 1000h : 0001h ou en adresse physique 0x10001. Sachant que 1000h :0000h a été réservée par le *boot*, pour sauvegarder l'identité du lecteur de démarrage.
Pour cela, une structure en C *superstruct* sera déclarée à l'adresse 1000h :0001h, en vue de sauvegarder les informations que nous venons de citer.
(La structure est déclarée dans le fichier .\include\dtct16.h)

4.4.2.2.5. Mise en place du *basculeur 16-bit*

Dans le PENTIUM, nous avons principalement deux modes, le mode réel et le mode protégé, citons également le mode virtuel. Pour le premier mode (mode 16-bit), il s'agit d'un fonctionnement en mode 8086, tandis qu'au deuxième (mode 32-bit), c'est un mode compatible avec le 80386.
Les deux modes sont complètement distincts. Les routines BIOS opèrent en 16-bit (mode réel), nous ne pouvons alors les utiliser en étant dans le mode 32-bit. Ce qui revient à dire, qu'il va falloir réécrire tous les drivers de périphériques standards pour le mode 32-bit. Nous nous sommes dit alors la chose suivante : Pourquoi ne pas faire commuter le processeur du mode 32-bit vers le mode 16-bit, chaque fois qu'il y a accès au périphérique (exemple : E/S disque …) ?. Ceci "freinera" considérablement le système, mais c'est un compromis que nous allons devoir faire, et ce, pour des raisons de simplicité.
Pour ce faire, nous avons pensé à mettre en place un serveur que nous avons baptisé *basculeur*, à qui nous pouvons depuis le mode 32-bit, solliciter pour une entrée/sortie (doit être bien sûr, prise en charge par le *basculeur*). Le *basculeur* doit alors sauver le contexte en cours, mais aussi les paramètres et l'identité de l'entrée/sortie. Une fois ceci fait, il passe en mode réel, et exécute la fonction adéquate dans la partie 16-bit, cette dernière étant basée sur les routines BIOS. Le *basculeur* doit retourner en mode protégé et restaurer le contexte, tout en mettant à disposition de l'appelant, les résultats de l'entrée/sortie effectuée.

Toute cette "gymnastique" doit être prise en charge de part et d'autre (c-à-d depuis le mode réel et le mode protégé). Dans ce paragraphe nous allons décrire, uniquement le *basculeur16-bit*.
Nous allons écrire en assembleur le *basculeur 16-bit*, puis le copier à l'adresse mémoire, 6000 :0000h (par convention). Il faut savoir que le *basculeur 16-bit* ne sera sollicité que par sa partie complémentaire (*basculeur 32-bit*). Nous pouvons alors être sûr, que les premières instructions à effectuer par le *basculeur 16-bit*, se dérouleront en mode protégé.

Le basculeur 16-bit devra alors en premier lieu, faire désactiver la pagination (éventuellement activée), et ce, en mettant le bit-31 (PG) du registre CR0 à zéro. Après quoi, les registres de segmentation (DS,ES,SS) sont chargés à 20h, qui correspond au segment de données 16-bit dans la GDT (voir le paragraphe sur l'initialisation de le GDT), puis force le bit-0 du registre CR0 à 0 pour passer en mode réel. Une instruction de saut vers 6000h :$+4 doit être faite, afin de mettre à jour la file des instructions pour le mode 16-bit d'une part, et d'autre part, pour charger le registre CS avec une valeur de segment valide en mode 16-bit (segment :offset). (voir le paragraphe sur le passage au mode 32-bit).
La CPU est désormais en mode réel, il faudrait alors recharger les segments de données pour pointer sur le segment du *basculeur 16-bit*. Autrement dit, (DS,ES,SS) vont être forcés à 6000h.
Il doit aussi initialiser le registre SP pour la pile (0FFF0h), et de redéfinir la table des interruptions (IDT), avec comme adresse de base 0 et de taille 3FFFh (255 vecteurs d'interruption en mode réel de taille 4-octect), et ce, moyennant l'instruction `lidt` (voir le paragraphe initialisation de la table d'interruptions).

Le basculeur 16-bit, doit prélever l'identité de la fonction, depuis la zone des paramètres, à l'adresse 9000h :0050h (par convention), sachant que les premiers 4Fh octets du segment 9000h sont réservés par convention pour la sauvegarde du contexte en mode 32-bit.
Suivant la valeur ainsi trouvée, il exécutera la fonction correspondante, et nous donnons dans ce qui suit, les valeurs associées pour identifier les fonctions :

printk id=1
putchr id=2
write id=3
read id=4

Notons aussi que les paramètres de ces fonctions, sont à retirer depuis l'adresse 9000h :100h (par convention), et le buffer ou la zone de données (comme par exemple les chaînes de caractères à afficher …) est à l'adresse 9000h:0200h (par convention).
Une fois la fonction, sollicitée, est exécutée, le basculeur 16-bit doit retourner en mode protégé puis léguer le contrôle au *basculeur 32-bit* qui est à l'adresse physique 0x63000 (par convention), et ce, après avoir bien sûr, chargé les valeurs de retour de la fonction ainsi exécutée le cas échéant. (voir paragraphe : passage en mode protégé, pour comprendre la manière d'y procéder).

Remarque

Lors de l'écriture de ce fragment en assembleur, nous avons tenu compte du fait, que l'adresse physique de base de ce programme est à 0x60000, et ceci, pour pouvoir intervenir sur le contenu des emplacements dans le programme, dans le cas d'un adressage indirect.
(*Le basculeur 16-bit* se trouve dans le fichier source .\16\bascul.asm)

4.4.2.2.6. Chargement de la partie 32-bit

Comme nous l'avons déjà souligné, les parties de l'exécutif seront concaténées sur disque. Puisque nous avons chargé jusqu'à présent 1 secteur physique de *boot*, et 18 secteurs pour la partie 16-bit. Nous aurons donc notre partie 32-bit au 19^{ième} secteur physique sur disque. Ce qui est équivalent à dire que sur une disquette 3pouces1/4, ayant 18 secteurs/tête et 2têtes/cylindre, nous allons avoir l'emplacement disque de la partie 32-bit comme suit :

Cylindre =0, Tête=1, secteur=1.

Il suffit après quoi, de faire charger ces paramètres dans les registres du processeur, comme c'était le cas d'ailleurs lors du chargement de la partie 16-bit, et de lire 18 secteurs vers l'adresse temporaire 9000h :0000 (en faisant appel à la routine BIOS int 13h) , puis d'incrémenter l'adresse de 18*512 octets, ainsi que le paramètre Tête de 1 pour lire davantage 18 autres secteurs (étant donné que la partie 32-bit sera plus importante que la partie 16-bit en terme de taille).
Après activation de la ligne A20 (voir prochain paragraphe), nous allons copier cette partie temporaire ainsi chargée, vers une adresse physique définitive, se trouvant à 0x100000 (par convention).

4.4.2.2.7. Activation de la ligne A20

Il faut savoir que le PENTIUM en mode réel (mode de démarrage par défaut), se comporte comme un processeur 8086. Il ne peut alors dans ce mode adresser plus de 1 Mo. Autrement dit, l'espace d'adressage en mode réel est dans l'intervalle suivant :
[0000h :0000 … FFFFh :FFFFh]. Nous pouvons néanmoins activer la ligne A20 du bus pour que le processeur puisse accéder aux 64Ko de mémoire immédiatement supérieure à l'adresse FFFFh :FFFFh. C'est ainsi que l'adresse FFFFh:0010h correspondante bien sûr à l'adresse physique 0x100000 (car 0x100000=FFFF*10h+10h) peut être accédée depuis le mode réel du PENTIUM.
Pour ce faire, nous allons devoir lire la valeur du port 92h, puis faire positionner à 1, l'avant dernier bit des poids faibles de la valeur lue, et de l'écrire dans le port 92h.

4.4.2.2.8. Initialisation de la table des segments (GDT)

En vue de passer en mode protégé et d'exécuter la partie chargée (partie 32-bit), nous sommes obligés d'initialiser la table des segments (GDT : *Global Descriptor Table*) du PENTIUM.
Rappelons que la GDT, est une table implantée en mémoire physique, dont l'adresse est sauvegardée dans le registre GDTR (accessible uniquement par les instructions LGDT ou SGDT), contenant la description de chaque segment qui doit être éventuellement chargée dans un registre de segmentation (CS, DS, ES, SS, FS, GS). Le descripteur de chaque segment fait 8 octets. C'est alors qu'en ajoutant chaque fois la valeur 8 à l'index, que nous pouvons indexer ou parcourir l'ensemble des descripteurs segment présents dans cette table. Le processeur ajoute bien sûr, en vue de localiser un segment dans la table, l'adresse de base se trouvant dans GDTR, autrement dit l'adresse mémoire physique de la GDT.
Notre table va résider par convention à l'adresse 4000h :0000 (0x40000 en adresse physique) et doit être remplie d'informations sur les segments à utiliser. Chaque descripteur de segment est défini comme suit [Intel]:

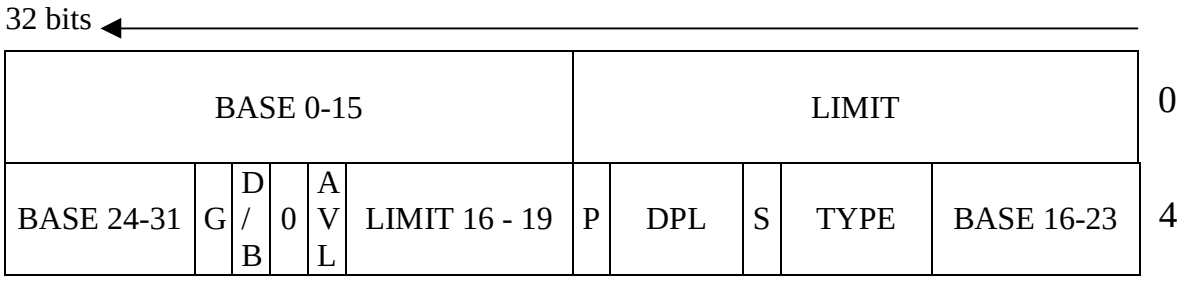


Figure 4.2. Descripteur segment du PENTIUM.

La **base**, sur 32-bit, est l'adresse linéaire où débute le segment en mémoire.

La **limite**, sur 16-bit, définit la longueur du segment. Si le bit **G** est à 0, la limite est exprimée en octets, sinon, elle est exprimée en nombre de pages de 4Ko.

Le **type** définit le type de segment, comme code, données ou pile.

Le bit **S** est mis a 1 pour un descripteur de segment et a 0 pour un descripteur système (un genre particulier de descripteur que nous verrons par la suite).

Le **DPL** indique le niveau de privilège du segment (0-3). Le niveau 0 correspond au mode super-utilisateur. Le niveau 3 est souvent utilisé pour le mode utilisateur.

Le bit **P** est utilisé pour déterminer si le segment est présent en mémoire physique. Il est à 1 si c'est le cas.

Le bit **D/B** précise la taille des instructions et des données manipulées (16-bit ou 32-bit). Il est mis à 1 pour 32-bit.

Le bit **AVL** est librement disponible.

Dans la table GDT du PENTIUM, le premier descripteur (descripteur 0) est réservé par Intel. Nous devons alors débuter au prochain, autrement dit à l'emplacement 8 (taille d'un descripteur de segment).
Nous envisageons une table avec un descripteur de code (index 8) de privilège 0 et adressant tout l'espace d'adressage (4Go). Pour le segment de données (index 0x10), nous ferons de même, sauf qu'une différence est à noter bien sûr au niveau du champ type. La pile utilisera également ce segment de données.
Le troisième segment que nous aurons à déclarer (index 0x18), concerne le segment de code en mode 16-bit, nécessaire pour prendre la main en passant du mode 32-bit au mode 16-bit. Un segment de données (index 0x20) sera associé au précédent, et prendra en charge l'accès aux données lors d'un basculement du mode 32-bit au mode 16-bit.
Puis, nous devons prendre en charge les segments pour le mode utilisateur. Ceci pour une éventuelle utilisation d'espaces distincts entre le noyau et l'utilisateur. Pour cela, nous allons déclarer un segment de code utilisateur (index 0x28) avec le privilège 3 (le plus faible privilège) et son segment de données/pile associé (index 0x30) également avec le plus faible privilège.

Une fois les segments déclarés, nous allons copier cette table à l'emplacement spécifié (0x40000), puis de charger avec l'instruction `lgdt`, l'adresse de cette table ainsi que sa taille lue depuis un emplacement mémoire, autrement dit, cette instruction doit avoir un adressage indirect, dont le contenu à atteindre sera l'adresse de la GDT et sa taille.

(L'initialisation de la GDT se trouve dans le fichier source : .\16\init_gdt.asm)

4.4.2.2.9. Passage en mode protégé

Nous pouvons à présent et après avoir chargé la partie 32-bit à l'adresse mémoire physique 0x100000, et initialisé la table GDT, pouvoir passer en mode protégé.
Pour ce faire, il suffit de positionner le bit-0 du registre CR0 à 1. Il va falloir avant ceci, inhiber les interruptions, étant donné que nous n'avons pas initialisé au préalable une table de vecteurs d'interruption (IDT).
Après cela, nous devons vider la file des instructions (présentes dans le cache), et ce, en faisant un saut inconditionnel, vers l'instruction suivante.
Nous devons ensuite charger les descripteurs segment de données (DS, ES, SS, FS, GS) avec la valeur index correspondante dans la table GDT. Pour notre cas elle serait de 0x10.

Nous sommes à présent en mode protégé, nous devons faire un saut (FAR) vers la partie 32-bit chargée. Puisque le langage assembleur MASM ne supporte pas ce genre d'instruction (saut vers CS :EIP), nous devons avoir son code instruction. Pour ce dernier, il s'agit de précéder le tout d'une valeur 66h, afin de dire au processeur qu'il devra prendre le format 32-bit de l'instruction, suivi de l'instruction de branchement vers l'adresse de la partie 32-bit chargée (CS=08 :EIP=100000h) dont le code est :
0eah, 00100000h,08.
Bien sûr, la valeur 8, correspond à l'index 8 (code segment) dans la table GDT.
(La fonction qui permet le passage en mode protégé est écrite en assembleur, se trouvant dans le fichier source : .\16\mod_prtg.asm).

4.4.3. Partie 32-bit (mode protégé)

Cette partie sera disant de base, pour le reste des parties qui seront réalisées. En effet, nous allons se baser sur cette partie pour implémenter les stratégies et politiques de notre exécutif. C'est en partie pour cette raison, que nous allons devoir considérer les parties à venir, à savoir, les modules temps réel et l'interface utilisateur, comme étant rattachées à la partie 32-bit *à fortiori* en terme de compilation. Autrement dit, nous allons considérer la partie 32-bit, comme la partie "mère" de toutes les parties à venir, étant donné que l'état du processeur, dans lequel s'opéreront ces modules ainsi que leur compilation se fera en mode 32-bit. Notons que les fonctions utilitaires définies dans la partie 16-bit, seront utilisées et recompilées pour ce présent mode.

Un peu comme la partie 16-bit, nous devons établir quelques initialisations, avant le lancement de la fonction *main* de la partie 32-bit. Nous allons alors écrire en assembleur un programme qui prendra le premier contrôle dès attribution par la partie 16-bit (lors du passage en mode protégé). Ce programme fera l'initialisation de la pile à l'adresse physique 6FFFFCh (par convention) en forçant ESP à cette valeur, puis invoquera la fonction *main* de la partie 32-bit.

(Le *startup* pour la partie 32-bit est localisé dans le ficher source : .\32\start32.asm.)
(La fonction *main* quant à elle, sera placée dans le fichier source: .\32\kr32.cpp.)
(Les fonctions utilitaires pour le 32-bit sont dans le source : .\utils\32).

Une fois la fonction *main* est appelée, celle-ci à son tour fera un appel pour mettre en place la deuxième partie du *basculeur*(*basculeur 32-bit*). Ce dernier est bien sûr complémentaire à celui déjà présenté (*basculeur 16-bit*).

Pour prendre en charge les interruptions matérielles et logicielles, une routine est appelée pour mettre en place les vecteurs d'interruption et les routines correspondantes, dans une table dédiée en mémoire appelée IDT dans le PENTIUM.

Puis une fonction est appelée pour programmer le PIC 8259A, en vue de prendre en charge les lignes d'interruptions IRQ suivant les niveaux définis dans la IDT.

Des fonctions d'initialisation temps réel sont appelées, pour après quoi, faire appel à l'application temps réel de l'utilisateur (*application_tr*).

Nous allons détailler dans les sous paragraphes suivants, les différentes routines citées ci-dessus, cependant, vu son importance, le module temps réel sera considéré comme paragraphe indépendant.

4.4.3.1. Mise en place du *basculeur 32-bit*

Comme nous l'avons déjà souligné, cette partie est en définitive, complémentaire à celle présentée dans le paragraphe portant sur *le basculeur 16-bit*.
Cette partie écrite également en assembleur fera, la copie du programme *basculeur 32-bit* à l'adresse mémoire physique 0x63000 (par convention). A remarquer que cette partie se logera juste au-dessus de la partie *basculeur 16-bit*, mais sans pour autant avoir un quelconque chevauchement. Lors d'une éventuelle extension, il faudrait penser à ce que les deux parties soient distinctes et de veiller à garder leur intégrité en mémoire.
Ajoutons le fait que les instructions, du *basculeur 16-bit* sont différentes de celles du *basculeur 32-bit*, étant donné que nous ferons une compilation différente et d'autant plus que le mode du processeur dans chacun des cas est différent (mode réel et mode protégé).

La routine basculeur 32-bit, commence par enregistrer le contexte processeur par le biais de la fonction *sauver_context*, à savoir, les registres généraux respectivement (EAX, EBX, ECX, EDX, DS, ES, SS, EDI, ESI, EBP, ESP, CS, EIP) mais aussi, l'adresse et la taille des tables IDT et GDT (avec les instructions respectivement `sidt` et `sgdt`) et éventuellement le répertoire des pages CR3. Le contexte est comme nous l'avons déjà vu, sera enregistré à partir de l'adresse physique 0x90000 (par convention).
Notons que pour le registre EIP (conteur ordinal), celui-ci n'est pas accessible par instruction, et nous allons seulement enregistrer un emplacement connu du programme que nous avons appelé *retour*, et à partir de-là, le *basculeur 32-bit* poursuivra son exécution lors du retour depuis *le basculeur 16-bit*.

Une fois le contexte enregistré, nous devons faire un saut (FAR) vers *le basculeur 16-bit*. Ce dernier comme nous l'avons supposé, se logera à l'adresse mémoire physique 0x60000. Etant donné que l'assembleur MASM, ne supporte pas les sauts longs, nous devons encore du coup, écrire le code de l'instruction correspondante (jmp far 18h :60000h). Bien sûr 18h est l'index du descripteur de code 16-bit, déclaré dans la GDT.
Le code instruction du saut, doit être précédé par 67h étant donné que nous sommes sur le point de passer d'un segment de code 32-bit vers un segment de code 16-bit, sinon une erreur de protection générale se produira et la machine va redémarrer puisqu'il n y a pas encore de prise en compte d'interruptions à ce niveau-là.
Au retour (point *retour* cité précédemment), nous allons bien évidemment faire l'inverse de ce qui a été fait, notamment en restaurant le contexte de la CPU depuis l'adresse physique 0x90000.
Nous ajoutons le fait que le passage de paramètres éventuels pour les fonctions à exécuter en mode réel, que nous avons vu dans le *basculeur 16-bit*, ne s'effectue pas à ce niveau (basculeur 32-bit), mais il appartient à la fonction (appelante), de prendre en charge l'empilement ou l'enregistrement de ses paramètres, avant l'appel au *basculeur 32-bit*. C'est pour cela que nous avons songé d'ailleurs, d'écrire les fonctions d'E/S prise en charge par le basculeur 16-bit, sous forme de macros, en langage C, afin de procéder à l'enregistrement de paramètres, de l'identifiant de la fonction et l'appel au *basculeur 32-bit*, chaque fois qu'il y a appel de fonctions d'E/S prise en charge depuis le mode protégé.

(*Le basculeur 32-bit* est dans le fichier source suivant : .\32\bascul.asm)

4.4.3.2. Initialisation de la table des vecteurs d'interruption (IDT)

Cette table permet la prise en charge des interruptions, en contenant des descripteurs (voir figure 4.3) ayant les adresses mémoire physique des routines d'interruptions, pour chaque interruption prise en charge. Cette table est consultée par le processeur à chaque interruption. Son adresse en mémoire physique et sa taille sont initialisées avec l'instruction `lidt`, et chargées dans le registre IDTR du processeur. Nous allons devoir mettre en place cette table (IDT), en vue de prendre en charge les interruptions.

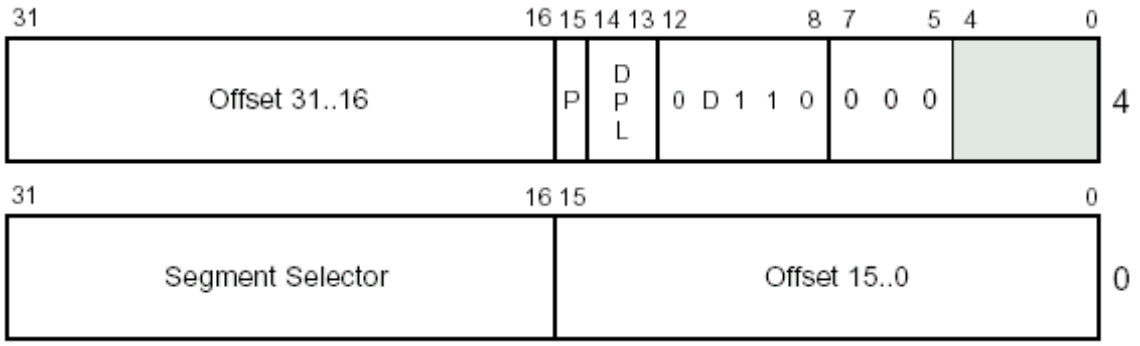


Figure 4.3. Descripteur d'interruptions[Intel].

Pour ce faire, nous allons écrire un ensemble de fonctions, qui permettent la mise en place de la IDT, et les routines d'interruption correspondantes.
init_idt est une fonction appelée depuis la fonction principale *main*, qui permet justement la mise en place de la IDT, et ce, en appelant à son tour, la fonction :

```
int set_idt(void* offset, short int selecteur, short int type,short int numero_descripteur) ;
```

Cette fonction permet de remplir la IDT, en ayant comme paramètres, les informations du descripteur d'interruption à mettre en place.

- Le paramètre **offset** permet de spécifier l'adresse physique de la routine d'interruption à traiter.
- Le paramètre **selecteur** permet de spécifier le segment de code avec lequel se fera le traitement de la routine d'interruption (la contenu de CS lors du traitement de l'interruption).
- Le paramètre **type** permet de spécifier le type du descripteur (voir figure 4.3).
- Le paramètre **numero_descripteur** permet de spécifier le numéro de l'interruption, en vue de placer le descripteur d'interruption, à l'index correspondant à la IDT. Sachant que la IDT peut être parcourue, en ajoutant chaque fois 8-octet à son index (taille du descripteur).

Nous allons prendre en charge les 17 NMI (*Non Maskable Interrupt*), dont le numéro d'interruption varie de 0 à 17 (convention Intel), les autres jusqu'à 39h sont réservées par Intel. Nous devons également prendre en charge les interruptions provenant des périphériques (IRQ), notre convention est d'associer le numéro 50h à l'interruption provenant de l'IRQ0 jusqu'à 57h (IRQ7). Puis 60h pour IRQ8 jusqu'à 67h (IRQ15).

Il faudrait alors, appeler au sein de la fonction *init_idt* la fonction *set_idt* 17 fois (correspondent aux NMIs), ajoutant 16 fois (correspondent aux IRQs). Ce qui fait 33 appels de la fonction *set_idt*. Cette dernière bien sûr est appelée, avec des paramètres différents, étant donnés les différents numéros des interruptions et les adresses des routines d'interruption correspondantes.
Notons que l'adresse où se logera la IDT est par convention à l'adresse physique 0x50000. Cette adresse sera prise comme adresse de base par la fonction *set_idt*, pour pouvoir indexer la table et de mettre en place les descripteurs d'interruption.

Nous devons ensuite, indiquer au processeur l'emplacement et la taille de la IDT ainsi créée, et ce, en les chargeant avec l'instruction `lidt`, avec le même format de paramètres que ceux de la GDT précédemment vue. Nous devons alors lui spécifier avec un adressage indirect, l'adresse mémoire physique de la IDT (0x50000 par convention) ainsi que sa taille (256 niveaux * 8 octects des descripteurs = 2048 octects).

(Les fonctions *init_idt* et *set_idt* sont dans le fichier source suivant : .\32\init_idt.cpp).

4.4.3.2.1. Les routines d'interruption

Pour une routine d'interruption dans notre système, nous allons la prendre en charge avec deux fonctions. Une de ces fonctions sera écrite en assembleur, en vue de faire des prétraitements avant le traitement d'interruption. L'autre fonction sera écrite en C, pour traiter l'interruption proprement dite.
Nous devons alors programmer les 33 fonctions en assembleur et en C, étant donné le nombre d'interruptions à prendre en charge dans la IDT.
Cependant, et pour des raisons de simplicité, nous n'allons pas écrire le code qui traitera les NMIs, nous nous contenterons d'afficher un message pour signaler l'arrivée de l'interruption, et éventuellement faire une boucle sur elle-même, dans le cas d'une erreur fatale (Erreur de protection générale …).
Ce raisonnement est aussi valable pour les routines prenant en charge les interruptions matérielles. Nous nous contenterons là aussi, uniquement des périphériques à prendre en charge (le clavier(IRQ1), PIC esclave(IRQ2), l'horloge temps réel (IRQ8)). Pour les autres IRQs, leur routine ne disposeront d'aucun traitement.

Ce sont les routines écrites en assembleur, qui après leur réalisation du prétraitement d'interruption, vont appeler les fonctions écrites en langage C.
Donc, lors de la mise en place de la IDT, nous devons spécifier l'adresse des routines écrites en assembleur, et non pas celle écrite en C.

(Les routines écrites en assembleur sont dans le fichier source : .\32\it.asm.
Les fonctions traitant les interruptions sont dans le fichier suivant : .\32\ints.cpp).

4.4.3.3. Programmation du PIC 8259A

Pour que nous puissions réaliser physiquement, ce que nous avons supposé lors de la mise en place de la IDT, à savoir, la prise en charge des interruptions de périphériques (IRQ), nous devons programmer le PIC, de telle sorte à ce qu'il délivre les numéros de demande d'interruptions compris entre [50h - 57h] correspondant à [IRQ0-IRQ7] pour le maître, et [60h - 67h] correspondant à [IRQ8 –IRQ 15] pour l'esclave.
Nous allons donc faire une redirection des IRQ, pour qu'elles ne puissent pas entrer en conflit avec les interruptions NMI du PENTIUM réservées par Intel.
Par exemple, et avec cette configuration, le PIC fournira la valeur 51h au processeur à chaque interruption venant du clavier (IRQ1), rappelons que cette valeur, va servir au processeur comme index dans la table des vecteurs d'interruption (IDT) pour récupérer le vecteur d'interruption à partir de cet emplacement et d'exécuter par la suite la routine correspondante (driver clavier dans ce cas).

Pour ce faire, nous allons en premier lieu initialiser le PIC maître et esclave, et ce, en mettant respectivement dans les ports 20h et A0h la valeur 11h. Cette valeur va servir en partie à indiquer au PIC que nous allons l'activer (ICW1) grâce à la valeur 1 du poids fort, et d'utiliser le ICW4 indiqué par la valeur 1 du poids faible.

Puis envoyer les valeurs à retourner pour IRQ0 et IRQ8 respectivement au maître et esclave, et comme nous l'avons énoncé, nous allons utiliser les valeurs 50h et 60h. Il faudrait alors envoyer ces valeurs respectivement au maître et esclave (ICW2), sur les ports 21h et A1h.

Il faudrait par la suite initialiser le ICW3, en envoyant la valeur 4 au maître et la valeur 2 à l'esclave respectivement au port 21h et A1h.
Pour la valeur $4 = (000100)_2$ qui est mise dans ICW 3 de l'esclave, la position du 1 dans le mot, va indiquer la broche à laquelle l'esclave est attaché, et dans ce cas, elle correspond à la broche numéro 2, autrement dit IRQ2. Idem pour la valeur 2 mise dans le ICW3 de l'esclave.

Nous avons spécifié dans la configuration (ICW1) que nous allons utiliser le ICW4. Pour ce dernier, nous allons envoyer la valeur 1 au port 21h et A1h respectivement du maître et esclave. Cette valeur va indiquer aux PICs, qu'il s'agit d'un mode normal 8086. (voir documentation du PIC 8259A pour plus de détails).

La fonction :
```
void init_pic() ;
```

sera chargée de faire cette initialisation. Après que cette dernière soit faite, les deux paramètres passés en argument à la fonction :

```
void masquage_it(unsigned char ICW2_Master, unsigned char
ICW2_Esclave) ;
```

vont servir à masquer/démasquer les interruptions, suivant le fait que le bit à 1 sert de masquage et 0 pour démasquer l'interruption correspondante à la position du bit dans le mot. (Les paramètres sont à envoyer respectivement vers le port 21h et A1h).

Comme nous l'avons déjà abordé, nous allons uniquement pour des raisons de simplicité, prendre en charge le clavier(IRQ1), le PIC esclave(IRQ2), et l'horloge temps réel (IRQ8).
Ce qui revient à dire, envoyer les valeurs suivantes : 0xF9 et 0xFE respectivement au maître et l'esclave.

(Les deux fonction ci-dessus, se trouvent dans le fichier source : .\32\init_pic.cpp).

4.5. Module temps réel

Une fois l'initialisation de la machine est faite, la fonction principale *main* fait appel à la fonction *application_tr*, dans laquelle se trouve le code utilisateur (module d'applications temps réel). Comme nous l'avons énoncé lors de la conception, l'utilisateur va pouvoir lancer des tâches temps réel, dans l'espace d'adressage du noyau. Ceci sous-entend, que l'application temps réel de l'utilisateur va faire partie du noyau. Ce qui revient à dire aussi, que l'utilisateur devra recompiler le noyau, chaque fois qu'il y a modification dans le module d'application temps réel (fonction *application_tr*).
Du point de vue utilisateur, son application sera développée au sein de la fonction *application_tr*. Mais du point de vue noyau, le code sera considéré comme une poursuite d'exécution ou même un simple appel de fonction à *application_tr*.

Pour pouvoir développer ses applications temps réel, l'utilisateur pourra manipuler des fonctions pour créer/supprimer des tâches et des sémaphores, implantées au niveau module des tâches. Il dispose également d'un ensemble de fonctions utilitaires pour les tâches, et de gestion du temps.
L'utilisateur n'a pas, à se soucier, quant à l'ordonnancement des tâches temps réel. Celui-ci doit lui être totalement transparent, et d'autant plus que les fonctions d'ordonnancement sont implantées de sorte, à ne servir que la politique d'ordonnancement, et n'offrent aucun service à l'utilisateur.

Nous allons aborder dans les sous paragraphes suivants, les différentes fonctions permettant la manipulation des tâches (côté utilisateur et ordonnancement), des sémaphores, mais aussi de fonctions pour la manipulation du temps (extension de l'API).

(la fonction *application_tr* doit être écrite par l'utilisateur dans le fichier source : .\treel\appli_tr.cpp).

4.5.1. Création d'une tâche

Pour créer une tâche temps réel périodique/apériodique, l'utilisateur doit appeler la fonction :

```
int creer_tache(THREAD thread, DATE ri, TIME ci, TIME pi,
unsigned int num_file_sem) ;
```

Qui permet d'initialiser une tâche temps réel périodique, dont le code ou la routine d'exécution associée est indiqué par le paramètre *thread*, qui n'est rien d'autre qu'une adresse vers une fonction à exécuter. A noter que *THREAD* est de type (*void* *).
La première date de réveil r_i de cette tâche, est spécifiée dans le deuxième paramètre (par rapport à l'horloge logique absolue), dont le type *DATE* est *double* (64-bit).
Ci et Pi indiquent respectivement la capacité d'exécution de la tâche et la période de la tâche (en unités d'horloge logique absolue) et TIME est de type *unsigned long* (32-bit).
num_file_sem indique le numéro de la file des sémaphores associés à cette tâche, ce numéro est spécifié par l'utilisateur lors de la création de la file (voir *add_semaphore*).

Pour créer une tâche apériodique, il suffit de mettre la période de la tâche à 0. Ce qui revient à dire que le paramètre Pi=0. A ce moment-là, le paramètre ri (date de réveil) devient inutile, dans la mesure où la tâche apériodique survient comme son nom l'indique d'une manière complètement aléatoire.

Lors de l'appel à la fonction *creer_tache*, celle-ci recherche d'abord un contexte libre en appelant la fonction *get_tache_libre*. Cette dernière parcourt, le tableau global de structures de contexte des tâches (la taille du tableau est de 256 par convention, définie par *max_tache*), en vérifiant à l'intérieur de chaque structure, le champ nommé *magic*, s'il contient la signature ou la valeur 0xBABA. Si tel n'est pas le cas, elle retourne le numéro de la structure libre (non signée), sinon un nombre négatif (-1) est renvoyé.
Le numéro positif renvoyé par *get_tache_libre*, va servir d'identifiant à la tâche à créer. Le cas contraire, la fonction *creer_tache* va aussi renvoyer la valeur (-1) indiquant à l'utilisateur, l'impossibilité de créer la tâche.
S'il y a un espace libre pour la tâche, la fonction *creer_tache* va tester la valeur du paramètre Pi (période) en vue de déterminer le type de la tâche (périodique/apériodique).

• S'il s'agit d'une tâche périodique, il faudrait tester si cette tâche est susceptible d'utiliser des ressources (sémaphores), et ce, en vérifiant la file des sémaphores associée aux tâches dont le numéro est donné en paramètre (*num_file_sem*). Si la file n'est pas vide, il faudrait alors sauver cette valeur dans le contexte de la tâche, et plus précisément dans le champ : *sched.ressource.file* en indiquant aussi que la tâche utilise des ressources, en mettant une signature 0xFAFA dans le champ *sched.ressource.magic*.
Puis remplir les autres champs, *id* (identifiant de la tâche), *sched.pi* (la période), *etat* (mis à l'état PRET), *sched.type* (mis à PERIODIQUE), mais également la date de réveil *sched.ri*, et le compteur associé *sched.cpt_ri* (sur lequel, le calcul de la prochaine date de réveil est effectué).

Pour initialiser la zone de pile (2048 octets par convention pour chaque tâche), un espace est réservé d'une façon statique à l'emplacement mémoire physique 0x5FFFFE (par convention), et cette adresse sera considérée comme **le bas de la pile** pour la première tâche. C'est ainsi que pour réserver un espace de pile pour la tâche *i*, il suffit de retrancher la valeur ((i+1)*2048) à l'adresse 0x5FFFFE, pour retrouver le bas de la pile de la tâche *i*.
A noter que la première zone (à partie de 0x5FFFFE) est réservée pour une future utilisation d'où la valeur du (i+1) dans la formule précédente. La valeur retrouvée sera sauvée dans le contexte de la tâche (contexte hardware) et plus précisément dans le champ : *tss.esp* et de faire charger à cet emplacement mémoire, l'adresse de retour une fois terminée qui, sera l'adresse de la fonction : *nop_thread*(boucle sur elle-même), dans la mesure où, les tâches sont censées exécuter les routines d'une façon "indéfinie" dans le temps, et d'autant plus que nous n'aurons pas à charger d'autres modules une fois les tâches terminées.

Le champ *temp_pr*, qui sauve la priorité, dans le cas d'un héritage de priorité au sein d'une section critique, est initialisé à 0 (indique qu'aucune priorité n'est sauvée).
Le champ *tss.eflags* (relatif au hardware) sera initialisé à la valeur 0x200, pour faire positionner le bit IF de la CPU (*Interrupt Flag*) à 1 et les autres à 0.

Une fois ceci fait, il faudrait calculer la priorité de la tâche parmi celles présentes dans le système, et ce, conformément à la politique RM. C'est ainsi que la fonction *set_priorite* est appelée avec les paramètres nécessaires, à savoir, l'identifiant de la tâche, et la période de la tâche (rappelons que la priorité suivant RM, est plus grande pour les tâches dont la période est plus petite). La fonction *set_priorite*, repose sur un algorithme d'insertion, c'est à dire, que si nous avons les tâches T_1, T_2 avec des priorités P_1, P_2, tel que $P_1<P_2$, et que nous aurons à insérer une tâche T_3, dont la priorité s'avère intermédiaire au précédentes, nous devons à ce moment-là, donner la priorité de T_2 à T_3, puis faire incrémenter celle de T_2. Le raisonnement est analogue pour *n* tâches.
Comme nous l'avons souligné dans la conception, les tâches avec une même priorité se verront attribuer une même priorité.

La fonction *get_min_ri* est ensuite appelée par *creer_tache*, et ce, pour déterminer l'identifiant de la tâche, avec la plus petite date de réveil, et qui sera mise dans la variable globale *id_min_ri*, en vue d'être utilisée par l'ordonnanceur.

La fonction *add_file_pret*, qui permet d'insérer une tâche dans une file, sera appelée avec les paramètres suivant : *&tache[i]* et *&file_pret_periodque* indiquant respectivement, l'adresse de la tâche et l'adresse de la file des tâches périodiques prêtes.
Après ceci, le compteur global des tâches périodiques est incrémenté de 1.

• S'il s'agit par contre d'une tâche apériodique (période nulle), il faudrait à ce moment-là, juste remplir le champ *id* (identifiant), *sched.type* (mettre à la valeur APERIODIQUE), d'appeler la fonction *add_file_pret* mais avec cette fois-ci l'adresse de la file des tâches apériodiques appelée *file_serveur*. Puis d'incrémenter le nombre de tâches apériodique (variable globale *nbr_tache_aperiodique*).

Une fois l'un des deux grands points faits (périodique/apériodique), il faudrait continuer à charger le contexte de la tâche, dont les informations cette fois-ci concernent aussi bien les tâches périodiques que les apériodiques. C'est alors que la fonction *creer_tache*, charge le contexte de la tâche (qu'elle soit périodique/apériodique) d'informations suivantes :
Le champ *sched.ci* (la capacité d'exécution passé en argument), *sched.cpt_ci* (indique la capacité courante de la tâche, c'est un compteur d'ailleurs auquel, la valeur de la capacité, sera décrémentée à chaque top d'horloge).
Le champ *tss.eip*, va être chargé de la valeur *thread* (passé en paramètre), indiquant l'adresse du code ou de la routine à exécuter par la tâche.
Enfin la signature du contexte (comme quoi, le contexte est pris ou réservé par une tâche), en mettant la valeur 0xBABA dans le champ *magic*. Et la valeur de l'identifiant de la tâche est retournée à l'utilisateur.

(la fonction *creer_tache* est dans le fichier source : .\treel\tache.cpp)

4.5.2. Création d'un serveur pour les apériodiques

S'il y a création d'une tâche apériodique, celle-ci devrait être alors associée à la tâche périodique serveur en vue d'être ordonnancée.
La fonction permettant de créer un serveur est :

```
int creer_serveur(DATE ri, TIME ci, TIME pi) ;
```

Les paramètres indiquent respectivement la date de réveil du serveur, sa capacité et sa période (Idem aux paramètres *creer_tache*).

Les valeurs de retour sont :
Positive : serveur créé, et la valeur indique son identifiant.
-1 : contexte de tâche plein (erreur de création).
-2 : un serveur est déjà créé (erreur de création).

Etant donné que la tâche serveur est en définitive une tâche périodique, la fonction *creer_serveur* fait alors entre autres, appel à la fonction : *creer_tache* que nous avons définie dans le paragraphe précédent.

La fonction *creer_serveur* test s'il y a présence d'un serveur dans le système, en inspectant la valeur de la variable globale : *etat_serveur*, et renvoie une erreur (-2) si elle n'est pas nulle. Sinon elle fait appel à la fonction :

```
creer_tache ((THREAD)&serveur_tache, ri, ci, pi, 255) ;
```

serveur_tache est la fonction associée au serveur que nous allons voir en détail dans les prochains paragraphes.
Les paramètres *ri, ci, pi,* sont ceux passés en argument à la fonction *creer_serveur*.
Pour éviter l'utilisation des ressources par le serveur, nous allons mettre comme numéro de file de sémaphore, la valeur 255 qui est supérieure au nombre de files associées aux tâches déclarées (32), ce qui entraînera le fait que la fonction *creer_tache* va refuser cette valeur et ne pas accepter l'utilisation de ressources pour cette tâche.
Une fois ceci fait, il faudrait tester la valeur retournée par la fonction *creer_tache* ainsi appelée. Si aucune erreur n'est survenue (tâche créée), à ce moment-là, il faut mettre la valeur *etat_serveur* à 1, pour dire qu'il y a création d'un serveur, et d'enregistrer après ceci, l'identifiant dans *id_serveur* en vue d'être repéré par l'ordonnanceur. Puis redéfinir le type de la tâche du champ *sched.type* à SERVEUR après qu'il aurait été fixé par la fonction *creer_tache* à la valeur PERIODIQUE.
Il faudrait également, remettre le serveur à l'état SUSPENDU, dans le cas où la variable globale contenant le signal apériodique (*signal_aperiodique*) serait à 0. Pour enfin retourner à l'utilisateur l'identifiant du serveur.

(voir le fichier source: .\treel\tache.cpp).

4.5.3. Test de faisabilité

Une fois toutes les tâches créées, il faudrait tester leur faisabilité en vérifiant pour cela, la condition d'ordonnancement RM vue au premier chapitre. Comme nous l'avons dit lors de la conception, il ne faudrait pas que cette condition soit stricte, autrement dit, le test doit être fait, mais n'influe pas s'il ne satisfait pas la condition RM.
La fonction de ce test est :

```
int faisabilite( ) ;
```

cette fonction teste en premier lieu, la file des tâches périodique prêtes, pour s'assurer qu'il y a au moins une tâche qui est préalablement créée. Le cas échéant, vérifier s'il y a utilisation de ressources, et ce, en vérifiant la variable globale *utilisation_ressources.* Si c'est le cas, il faudrait calculer alors les priorités plafond en faisant appel à la fonction *set_priorite_plafond*. Cette dernière, doit attribuer à chaque ressource(sémaphore créé) la priorité plafond = max{Priorités(Ti) pour i=1,n} telle que Ti est susceptible d'utiliser la ressource concernée, sue après avoir parcouru *la file des sémaphores associée à une tâche*.

Une fois les priorités plafond fixées, pour chaque ressource (sémaphore), la condition de faisabilité sera faite en faisant appel à la fonction *condition_RM*, et tout en parcourant *la file des tâches périodiques prêtes*. Deux cas sont alors envisageables.

D'une part, la tâche est susceptible d'utiliser des ressources (en présence de la signature 0xFAFA dans le champ *sched.ressource.magic* du contexte de la tâche), à ce moment-là, la fonction *condition_RM* sera appelée avec les paramètres *Ci, Pi, Bi, i,* indiquant respectivement la capacité d'exécution, la période de la tâche, le facteur de blocage de la tâche, et l'identifiant de la tâche. Suivant ces paramètres, la fonction *condition_RM* va pouvoir déterminer en utilisant le quotient cumulé ∑(Ci/Pi) sauvé dans la variable réelle globale *facteur_U*, si $\sum (C_k / P_k) + (B_i / P_i) \leq i\,(2^{1/i} - 1)$, et pour éviter d'écrire des fonctions qui prennent en charge la puissance d'un nombre réel, nous avons pensé à utiliser plutôt un tableau de réels, déclaré global (*tableau_RM*) contenant les valeurs du terme : $i\,(2^{1/i} - 1)$, qu'il suffit d'accéder avec l'indice i.
Si la condition n'est pas vérifiée *condition_RM* retourne –1, sinon elle retourne 0.

D'autre part, si la tâche n'utilise pas de ressources, il faudrait appeler dans ce cas, la fonction *condition_RM* avec les paramètres suivants *Ci, Pi, 0, i,* idem aux premiers, sauf que cette fois-ci, le facteur de blocage Bi est bien sûr nul.

Lorsque nous avons évoqué le facteur de blocage Bi, nous n'avons pas expliqué la façon, de le déterminer. En réalité, s'il y a utilisation de ressources, un appel est fait à la fonction *determiner_Bi(int id)* avec *id* l'identifiant de la tâche. La fonction *determiner_Bi* est l'implémentation de l'algorithme de K. Tindell, et H. Hansson proposé au premier chapitre, pour la détermination du facteur de blocage. Elle retourne donc le facteur Bi, qui est le temps maximum de blocage d'une tâche par les moins prioritaires qu'elle, utilisant la même ressource. Notons que si le test de faisabilité est réussi dans le cas d'utilisation de ressources, le facteur Bi est ajouté au contexte de la tâche dans le champ : *sched.Bi*
(voir le source .\treel\tache.cpp).

4.5.4. Création d'un sémaphore

Pour que l'utilisateur puisse utiliser un sémaphore, il faudrait d'abord le créer par la fonction :

```
SEMAPHORE creer_semaphore() ;
```

Cette fonction retourne l'adresse mémoire physique de la structure du sémaphore créé, s'il y a erreur elle retourne un pointeur nul (NULL).

Un peu comme la fonction *creer_tache*, la fonction *creer_semaphore* doit en premier lieu vérifier s'il y a une structure de sémaphore libre, en appelant la fonction *get_sem_libre*, qui parcourt l'ensemble du tableau de structures sémaphores, en vérifiant chaque fois la signature 0xBABA. Si aucune signature trouvée dans le champ *magic*, la fonction *get_sem_libre* retourne immédiatement le numéro de la structure du sémaphore, qui va servir d'identifiant à ce sémaphore durant toute son utilisation. Si aucune structure libre trouvée, une valeur négative est renvoyée, et à ce moment-là, la fonction *creer_semaphore* retourne un pointeur nul (NULL).

Une fois ceci fait, il faudrait initialiser les éléments de la structure du sémaphore, en mettant dans le champ *val* (valeur du sémaphore) la valeur 1, dans le champ *id* l'identifiant du sémaphore, dans le champ *priorite* (priorité plafond) la valeur 0, puis marquer la structure comme étant prise, en mettant dans le champ *magic* la valeur 0xBABA et de retourner enfin l'adresse de cette structure (SEMAPHORE).

4.5.4.1. Ajouter un sémaphore dans une file

Pour pouvoir utiliser un sémaphore créé, et comme nous l'avons précisé dans la conception, il faudrait d'abord l'ajouter dans une file de sémaphores associés à une tâche. Pour cela, il faudrait appeler la fonction :

```
int add_semaphore(unsigned int num_file, SEMAPHORE sem,
unsigned long temps_SC) ;
```

l'entier *num_file*, est un numéro qui doit être donné par l'utilisateur, et représente le numéro de la file dans laquelle sera inséré le sémaphore *sem* ; cette file est appelée lors de la conception : *file des sémaphores associée à une tâche*. Les valeurs permises vont de 0 à 31 (par convention), et représentent le nombre ou le tableau de files déclarées comme globales.

Le paramètre sémaphore spécifie, le sémaphore créé par la fonction *creer_semaphore*. Quant au troisième paramètre, il spécifie le temps (en unité d'horloge logique absolue) de la section critique associée au sémaphore *sem.*

Il faudrait intervenir notamment dans cette fonction, sur les champs de la structure du sémaphore *sem,* qui sont les suivants :
time[file_sem], npred[file_sem], nsucc[file_sem], qui représentent respectivement :

- le temps de la section critique associée au sémaphore *sem,*(nous avons supposé lors de la conception qu'un sémaphore n'aura par simplicité qu'un seul temps associé à une section critique pour une même tâche) ;
- le prédécesseur dans la file numéro *file_sem ;*
- le successeur dans la file numéro *file_sem ;*

4.5.4.2. La primitive P(s) et le protocole à priorité plafond

Pour que l'utilisateur puisse utiliser une ressource (sémaphore) au sein du code associé à une tâche périodique, nous allons réécrire la primitive P(s) de Dijkstra en vue de prendre en charge le protocole à priorité plafond.
En faisant appel à la fonction :

```
void P(SEMAPHORE s) ;
```

celle-ci, devra en premier lieu vérifier si la valeur du sémaphore *s*, est à 0 (si la ressource est prise).
Si c'est le cas, elle doit non seulement attendre la remise à 1 de la valeur du sémaphore, mais aussi procéder éventuellement à l'héritage de priorité. Pour ce dernier, la primitive vérifie d'abord, si la priorité de la tâche en cours (dont l'identifiant est dans la variable globale *courant_id*) faisant appel à cette primitive, est supérieure à la priorité de la tâche qui détient le sémaphore *s* (l'identifiant de la tâche qui détient *s*, se trouve dans le champ *id_tache* du sémaphore *s*). La primitive à ce moment-là doit procéder, à l'héritage de priorité, et ce, en sauvant la priorité de la tâche qui détient le sémaphore *s*, dans le champ *temp_pr* de cette tâche, puis forcer sa priorité à celle qui demande *s*, dont la priorité est supérieure. Une fois ceci fait, la primitive doit mettre l'état de la tâche en ATTENTE, et sauver l'adresse du sémaphore *s* pour qui, elle est attente, et ce, dans le champ du contexte de la tâche, *sched.ressource.attente*, puis boucler sur elle-même.

Dans le cas où le sémaphore serait libre (valeur du sémaphore à 1) lors de l'appel à la primitive *P(s),* elle ne doit pas détenir directement le sémaphore *s*, avant de vérifier la condition posée par le protocole à priorité plafond.
La condition comme nous l'avons déjà vue dans le premier chapitre, consiste à vérifier si la priorité plafond du sémaphore *s* est strictement supérieure à toutes les priorités plafond, des sémaphores en cours d'utilisation par les autres tâches. Pour cela, la primitive doit parcourir *la file des sémaphores* (excluant ceux possédés par la tâche demandant *s*) puis comparer la priorité du sémaphore *s*, à celle du sémaphore trouvé, dans *la file des sémaphores*. Si elle n'est pas supérieure (priorité) pour l'un des sémaphores trouvés dans la file, il faudrait alors bloquer la tâche qui demande la ressource (le sémaphore *s*). Avant ceci, et en vue de faire un héritage de priorité, la primitive doit vérifier la priorité de la tâche en cours demandant le sémaphore *s*, si elle est supérieure à celle de la tâche détenant le sémaphore trouvé dans *la file des sémaphores*, et dont la priorité plafond est supérieure à celle du sémaphore *s*. Procéder le cas échéant à l'héritage de priorité, comme vu plus haut.

Il faudrait mettre après quoi, la tâche demandant *s*, en attente, en forçant comme déjà vu, son état à ATTENTE, et mettre aussi dans le champ *sched.ressource.attente*, du contexte de cette tâche, l'adresse du sémaphore *s*.
Une fois ceci fait, il faudrait boucler, tant que le sémaphore pour qui l'attente est faite, n'est pas encore libéré (sa valeur est 0). Une fois le sémaphore libéré, le scheduler est responsable de mettre la tâche à l'état PRET. La primitive est à ce moment-là réveillée, et doit refaire depuis le début la vérification de le la valeur du sémaphore, c'est-à-dire, depuis la toute première étape où il est question de vérifier si la valeur du sémaphore est à 0 (voir plus haut).

Maintenant après avoir passé toutes ces vérifications, le sémaphore *s* est en mesure d'être détenu par la tâche le demandant. La primitive doit à ce moment-là, mettre à 0 la valeur du sémaphore (pour indiquer que le sémaphore *s* est pris), puis de mettre dans le champ *id_tache* de la structure du sémaphore *s*, l'identifiant de cette tâche, de mettre également l'état de cette tâche à l'état SEM pour indiquer que la tâche va opérer en section critique, et enfin ajouter le sémaphore *s* dans *la file des sémaphores*, en faisant appel à la fonction *add_sem_file(s,&file_sem_courant)*, tel que &*file_sem_courant*, correspond à l'adresse de *la file des sémaphores*.

4.5.4.3. La primitive V(s) et le protocole à priorité plafond

la fonction :

```
void V(SEMAPHORE s) ;
```

est aussi réécrite en vue de prendre en charge le protocole à priorité plafond, et de suivre également la logique de la primitive *P(s)*, pour libérer le sémaphore *s* détenu par le biais de celle-ci.

En premier lieu, la primitive *V(s)* doit vérifier s'il y a eu héritage de priorité, en vérifiant pour cela, le champ *temp_pr* du contexte de la tâche détenant le sémaphore *s*, s'il n'est pas nul. Dans ce cas, il faudrait restaurer la priorité, en forçant la priorité de la tâche, à celle sauvée dans le champ *temp_pr*, puis de mettre ce dernier à 0.
La primitive doit appeler par la suite, la fonction *rem_file_sem(s,&file_sem_courant)*, pour retirer le sémaphore *s* de *la file des sémaphores*.
Il faudrait également rendre l'état de la tâche libérant *s* à l'état PRET (après avoir été à l'état SEM). Et enfin, mettre à 1 la valeur du sémaphore *s*.

(les primitives P et V sont dans le fichier source : .\treel\schedule.cpp).

4.5.5. Ordonnancement des tâches temps réel

A travers ce paragraphe, nous allons discuter de la façon d'ordonnancer les tâches périodiques. En partant de l'activation de l'horloge par l'utilisateur, suivi de l'administration d'interruption d'horloge temps réel, puis la sélection de la tâche périodique et le chargement de son contexte, et enfin la commutation de contexte et le retour de l'interruption ; nous allons nous accentuer sur toutes ces étapes, tout en abordant également l'ordonnancement des tâches apériodiques.

4.5.5.1. Activation de l'horloge temps réel - RTC

Sur les machines basées sur une architecture Intel, la RTC est constituée du processeur MC146818 de Motorola et d'une RAM de 64 octets alimentés par une pile ; ainsi l'horloge temps réel continue à fonctionner même lorsque l'alimentation secteur est coupée. Les demandes d'interruption générées par la RTC arrivent sur l'entrée IRQ8 du deuxième contrôleur d'interruption PIC 8259A. [SB 03]
La RTC est accessible par les ports d'E/S d'adresse 70h pour le registre d'adresse et 71h pour le registre de donnée. La RAM contient des registres d'état et de contrôle adressables individuellement ; ce sont les registres A, B, C et D. [SB 03]
D'autres cellules conservent l'heure courante sous la forme : seconde, minute, heure, jour, mois, année. La RTC peut être programmée pour qu'elle délivre des impulsions d'horloge périodiques arrivant sur l'IRQ8, et par défaut, il y a 1024 interruptions d'horloge / seconde.

Nous allons programmer la RTC, pour qu'elle nous délivre des interruptions périodiques, en vue de les utiliser pour ordonnancer les tâches.
Pour ce faire, nous allons écrire la fonction :

```
void activer_horloge();
```

qui doit être appelée par l'utilisateur une fois toutes les tâches créées. Cette fonction initialise la RTC, en envoyant la valeur 0Bh au port 70h, pour sélectionner le registre B, puis d'envoyer par la suite, la valeur lue depuis le port 71h (la donnée) en mettant son bit 5 (interruption périodique) à 1, autrement dit, armer l'interruption périodique.

4.5.5.2. L'interruption d'horloge temps réel

A chaque top d'horloge temps réel, une interruption 60h (IRQ8) se produit. Le processeur va indexer dans la table IDT, pour extraire l'adresse de la routine d'interruption, qui dans notre cas est l'adresse de la routine *int60* écrite en assembleur.
Cette routine va exécuter en premier lieu l'instruction `pushad` qui empile respectivement les registres suivants : EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, puis sauvegarde la valeur du registre ESP dans la variable *_esp* déclarée globale pour une future utilisation.
La routine *int60* appelle par la suite la fonction *call_int60* écrite en C.

La fonction *call_int60* commence par incrémenter le compteur absolu *cpt_absolu* (déclaré global), qui représente le temps absolu du système ou d'une autre manière, l'horloge logique comme nous l'avons déjà nommé dans les paragraphes précédents. Cette horloge logique est incrémentée comme nous venons de le dire, à chaque appel de fonction *call_int60*, autrement dit, 1024 incrémentation par seconde.
Suivant la variable globale *etat_IT* (non 0, le scheduler est actif, et 0 inactif) la fonction *call_int60* va faire appel à la fonction *schedule_tr* qui s'occupe de l'ordonnancement des tâches temps réel.

(La routine *int60* est dans le fichier : .\32\it.asm ; *call_int60* se trouve dans : .\32\ints.cpp).

4.5.5.3. Ordonnancement des tâches périodiques

Comme nous l'avons cité précédemment, la fonction :

```
void schedule_tr();
```

s'occupe de l'ordonnancement des tâches périodiques, en adoptant pour cela, la politique Rate Monotonic.
La fonction prend en charge, trois cas pour pouvoir séparer les tâches et les actions associées à entreprendre, en inspectant dans chacun des cas la variable globale *courant_id*, qui rappelons le, contient l'identifiant de la tâche en cours initialisé à la valeur -1.

♦ Le premier cas est traité si la variable, *courant_id* est à –1, autrement dit, s'il s'agit de la première exécution de la tâche. Dans ce cas, la fonction *schedule_tr* charge le contexte de la tâche *nop_task* (tâche périodique qui boucle sur elle-même) d'informations comme, l'adresse de la routine dans le champ *tss.eip*, le registre eflags dans le champ *tss.eflags*, et puis la zone de pile, à l'adresse de pile que nous avons réservé (voir création de tâche), qui par convention est à l'emplacement 0x5FFFFC, et cette dernière sera mise dans le champ *tss.esp* du contexte de la tâche *nop_task*. Cette dernière rappelons-le, s'exécute uniquement lorsqu'il n y a aucune tâche périodique à exécuter durant un ou plusieurs tops d'horloge, en d'autres termes, elle est sollicitée pour "combler les vides ou creux" temporels pouvant exister entre tâches périodiques.
Une fois ceci fait, la fonction *schedule_tr* teste si la tâche dont la date de réveil est minimale (tâche identifiée par la variable globale *id_min_ri* initialisée lors de la création des tâches), est inférieure ou égale à l'horloge logique (*cpt_absolu*), autrement dit, savoir si

la tâche est prête à être réveillée. Le cas échéant, elle doit vérifier si cette tâche s'agit d'une tâche périodique quelconque ou bien de la tâche périodique *serveur*. Pour ce dernier, nous y reviendrons dans le paragraphe abordant l'ordonnancement des apériodiques. Quant aux autres tâches périodiques, la fonction *schedule_tr* va à ce moment-là, sauver l'identifiant de cette tâche périodique (dont la date de réveil est minimale) dans la variable globale *courant_id*, et de réaliser par la suite le chargement de contexte de cette tâche, en appelant dans ce cas, la fonction *premiere_commutation*, et comme paramètre l'adresse du contexte de la tâche avec l'identifiant *courant_id*. A noter que le chargement et la commutation de contexte seront vus dans les prochains paragraphes.

♦ Le deuxième cas est traité, si la variable *courant_id* est à –2, ce qui veut dire que l'interruption d'horloge est apparue au moment de l'exécution de la tâche *nop_task*. Il faudrait alors chercher la tâche la plus prioritaire dont la date de réveil est minimum, dans *la file des tâches périodiques prêtes*, et ce, en faisant appel à la fonction *get_next_tache*, qui retourne NULL (pointeur nul) s'il n y a aucune tâche périodique prête dans la file au moment *cpt_absolu*, ou bien retourne l'adresse du contexte de la tâche élue.
Si la valeur retournée par *get_next_tache*, n'est pas nulle, il faudrait alors, appeler la fonction de chargement de contexte : *commuter*, qui sauvegarde le contexte de la tâche passé en premier paramètre, et charger le contexte de la tâche passé en deuxième paramètre, ce qui retombe à faire passer dans ce cas, les paramètres suivants : *&nop_tr* et *&p*, qui correspondent respectivement à l'adresse du contexte de la tâche *nop_task*, et celle de la tâche trouvée dans *la file des tâches périodiques prêtes*.

♦ Le troisième cas est le cas par défaut, c'est-à-dire, si l'identifiant contenu dans la varaible *courant_id* est une valeur quelconque, autrement dit, l'interruption est survenue au sein d'une tâche périodique quelconque (y compris la tâche serveur).
Si la tâche s'avère être la tâche serveur, à ce moment-là, un traitement spécifique lui sera consacré et nous l'aborderons au paragraphe traitant l'ordonnancement des apériodiques.
Il faudrait par ailleurs, et après avoir décrémenté la capacité d'exécution (*cpt_ci*), tester si la capacité de la tâche n'est pas épuisée, ce qui revient à dire, tester la variable *sched.cpt_ci*, du contexte de la tâche identifiée par *courant_id*, si elle est nulle. Dans ce cas, il faudrait recalculer la date du prochain réveil de cette tâche, et ce, en forçant le champ *sched.cpt_ri* à prendre la valeur : (*sched.cpt_ri* (ancienne valeur) + *sched.pi* (la période de la tâche)) puis de recharger *sched.cpt_ci* par *sched.ci*.
Une fois ceci fait, il va falloir faire appel à la fonction *get_next_tache*. Si cette dernière renvoie un pointeur non nul, il faudrait à ce moment-là, charger le contexte de la tâche correspondant à ce pointeur et de sauver le contexte courant, (par le biais de la fonction *commuter*) et par la suite, mettre à jour la variable *courant_id* à celle de l'identifiant de la tâche ainsi chargée.
Si par contre, la fonction *get_next_tache* renvoie un pointeur nul, il faudrait charger plutôt le contexte de la tâche *nop_task* (par le biais bien sûr, de la fonction *commuter*), et de forcer la variable *courant_id* à prendre la valeur –2.

La figure 4.4. résume ces trois cas, sous un automate.

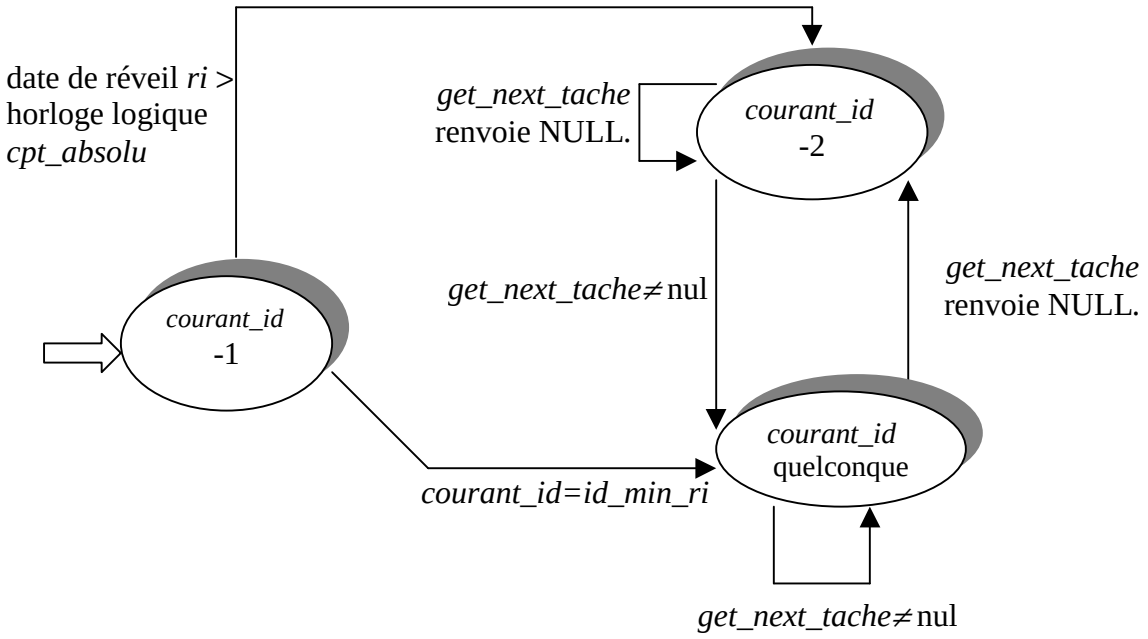


Figure 4.4. Automate d'états du scheduler.

4.5.5.3.1. Election d'une tâche périodique

Comme nous l'avons déjà vu, l'élection d'une tâche périodique, se fait par le biais de la fonction *get_next_tache*, qui parcourt *la file des tâches périodiques prêtes.* Dans cette dernière, si une tâche périodique est dans un état ATTENTE (attente d'une ressource), il faudrait vérifier à ce moment-là, si la ressource pour laquelle cette tâche est en attente serait libre, et de rendre ainsi l'état de cette tâche à l'état PRET, en vue de la faire rentrer en concurrence avec les autres tâches périodiques.
Pour sélectionner une tâche parmi d'autres, conformément à la politique Rate Monotonic, il va falloir vérifier si la prochaine date de réveil *cpt_ri* de chacune des tâches est inférieure ou égale à l'horloge logique (*cpt_absolu*), mais aussi, tester si cette tâche est dans l'un des états suivants {PRET,SEM}, puis de sauver temporairement la tâche jusqu'à ce qu'une autre tâche plus prioritaire soit trouvée, et prendra alors sa place. Cette façon de faire sera appliquée durant tout le parcourt de *la file des tâches périodiques prêtes*, pour renvoyer à la fin l'adresse de la tâche périodique qui est sauvée temporairement.
Si par contre, aucune tâche n'est trouvée ou encore ne satisfait pas les conditions citées, un pointeur NULL (valeur 0) sera retourné.

4.5.5.4. Ordonnancement des apériodiques

Les tâches apériodiques sont ordonnancées par la tâche périodique serveur. Ce dernier étant avant tout une tâche périodique, qu'il faudrait tenir compte durant l'ordonnancement, ou encore durant les trois cas ou états qui composent notre fonction *schedule_tr*.

♦ Dans le premier cas (*courant_id* est à –1), il s'agit de la première exécution du scheduler, et si après test de l'identifiant contenu dans *id_min_ri*, s'avère correspondre à la tâche serveur, il faudrait alors faire appel à la fonction *get_aperiodique* en vue d'extraire en FIFO depuis *la pile des apériodiques*, l'identifiant de la tâche apériodique, et de le placer dans une variable globale *aperiodique_id*, qui sera accessible par la tâche serveur, dont le code d'exécution est écrit au sein de la fonction : *void serveur_tache*( ) ;
Bien sûr, il va falloir aussi par la suite, faire le premier chargement du contexte avec celle du serveur en appelant la fonction *premiere_commutation*, comme c'était le cas d'ailleurs pour les tâches périodiques.

♦ Le cas où *courant_id* est à –2, c'est-à-dire, une interruption survenue au sein de la tâche *nop_task*, il faudrait à ce moment-là faire juste un appel à la fonction *get_aperiodique*, pour extraire éventuellement l'identifiant d'une tâche apériodique depuis *la pile des apériodiques* et de le placer dans la variable globale *aperiodique_id*, puis de poursuivre les étapes vues dans l'ordonnancement périodiques correspondantes à ce cas.

♦ Dans le cas par défaut (*courant_id* quelconque), il faudrait en premier lieu faire appel à la fonction *get_aperiodique*, puis de tester par la suite si l'identifiant *courant_id* correspond à ce du serveur, tout en vérifiant la disponibilité d'une apériodique (*aperiodique_id* non nul).
Si tel est le cas, décrémenter le compteur de la capacité de l'apériodique *sched.cpt_ci*, et de le vérifier par la suite s'il est épuisé. Dans ce dernier cas, il est à recharger la capacité *sched.cpt_ci* par celle de *sched.ci*, et de mettre à 1 les variables globales suivantes : *aperiodique_suivante* et *reinitialiser_serveur* qui indiquent respectivement s'il y a une apériodique en phase de traitement, autrement dit, l'autorisation d'extraire davantage les éventuelles tâches apériodiques par *get_aperiodique*, et pour l'autre variable, indiquer à la fonction *commuter* qu'il s'agit d'un chargement de contexte particulier réservé au serveur.
Une fois ceci fait, au sein de ces conditions, il va falloir également, tester la variable globale *signal_aperiodique*, qui indique l'état de *la pile des apériodiques*. Si cette dernière contient un élément (identifiant d'une apériodique), ce qui revient à dire aussi que la variable *signal_aperiodique* est à 1, il faudrait à ce moment-là, faire appel à la fonction *get_aperiodique* étant donné que l'ancienne tâche apériodique est épuisée en capacité et qu'il y a davantage de tâches apériodiques à traiter. Mais si par contre *la pile des apériodiques* est vide, il faudrait alors, suspendre le serveur en mettant son état à SUSPENDU.
Une fois ces actions réalisées, l'ordonnancement des périodiques correspondant à ce cas, se poursuit alors à ce niveau.

4.5.5.4.1. Serveur des aperiodiques

Comme nous l'avons cité précédemment, le code exécutable ou la fonction associée à la tâche périodique serveur est *serveur_tache*. Cette dernière, vérifie s'il y a une apériodique disponible au traitement, en vérifiant la variable *aperiodique_id,* mais aussi, s'assurer que l'état du serveur est dans l'état PRET.
Dans ce cas, elle extrait l'adresse de la fonction (se trouvant dans le champ *tss.eip* du contexte de la tâche), associée à l'apériodique dont l'identifiant est *aperiodique_id,* et fait appel à cette fonction. C'est pour cela, que nous disions que les tâches apériodiques dans notre système, vont s'exécuter dans le contexte du serveur. Les threads ou les fonctions associées aux apériodiques vont avoir alors la même zone de pile que celle du serveur, autrement dit, il s'agit d'un simple appel de fonction à l'apériodique lancé depuis la fonction associée au serveur (*serveur_tache*).
Après le retour depuis la fonction associée à l'apériodique appelée, le serveur est mis à l'état SUSPENDU.
(la fonction *serveur_tache* est dans le fichier source : .\32\schedule.cpp).

4.5.5.4.2. Election d'une apériodique

L'élection d'une apériodique ne se fait pas de la manière dont elle se fait celle des tâches périodiques. Ceci est dû principalement à la simplicité mis en œuvre pour ordonnancer les apériodiques. En effet, la fonction *get_aperiodique* qui s'occupe de l'élection ou à vrai dire de l'extraction de l'apériodique depuis *la pile des apériodiques*, est établie suivant FIFO.
La fonction *get_aperiodique* intervient directement sur *la pile des apériodiques*, elle est conditionnée néanmoins par l'état de la variable globale *aperiodique_suivante,* qui indique s'il y a une apériodique en phase de traitement, puisque nous avons supposé un traitement séquentiel des apériodiques, et doivent s'exécuter alors une après l'autre.
Dans le cas où il n y aurait pas d'apériodique en phase de traitement par le serveur (*aperiodique_suivante* est à 1), la fonction *get_aperiodique* devra par la suite, tester s'il y a toujours un signal apériodique autrement dit, tester la variable globale *signal_aperiodique* si elle est à 1. Si c'est vérifié, deux cas sont alors à prendre en charge :

- Le premier cas, le sommet de *la pile des apériodiques* est différent de son bas, en d'autres termes, *la pile des apériodiques* contient un élément, il faudrait alors envisager d'extraire du bas de *la pile des apériodiques* (*bottom*) l'élément et décrémenter *bottom,* puis mettre l'élément extrait dans la variable globale *aperiodique_id,* et de rendre l'état du serveur à l'état PRET.

- Le deuxième cas est en définitive, une suite au premier, qui consiste à tester si le sommet de *la pile des apériodiques* est devenu égal au bas de celle-ci. Dans ce cas, il va falloir mettre la variable globale *signal_aperiodique* à 0, pour dire que *la pile des apériodiques* est vide.

A noter que pour les deux cas, la variable globale *aperiodique_suivante* sera mise à 0 pour signifier que l'extraction est effectuée, et ne pas exécuter prochainement la fonction *get_aperiodique* tant que la présente apériodique extraite, ne s'est pas encore épuisée en capacité lors de son traitement par le serveur des apériodiques.

(La fonction *get_aperiodique* est dans le fichier source : .\32\schedule.cpp)

4.5.5.4.3. Signal apériodique

Nous avons abordé dans les sous paragraphes précédents, que l'état de *la pile des apériodiques* est représenté par la variable globale *signal_aperiodique*. Nous avons supposé lors de la conception, que les différents signaux des apériodiques fournis par les capteurs externes au système, seront associés aux différents niveaux d'interruption dans le système. Il appartient alors à l'utilisateur, de mettre une fonction imbriquée dans la routine d'interruption correspondante au signal de l'apériodique ou encore, au capteur externe associé. Nous avons implanté cette routine en langage C, qu'il suffit à l'utilisateur d'insérer dans l'une des routines d'interruption (haut niveau) associée au signal apériodique.
La fonction :

```
void signaler_aperiodique(int id) ;
```

permet d'empiler dans *la pile des apériodiques*, l'identifiant de la tâche apériodique *id*, en manipulant la variable globale *top_aperiodique* initialisée à l'adresse du dernier élément de la structure *pile_aperiodique*.
La fonction *signaler_aperiodique* permet également, après cet empilement, de mettre à 1 la variable globale *signal_aperiodique* pour dire que la pile des apériodiques contient un élément.

(la fonction signaler_aperiodique est dans le fichier source: .\32\schedule.cpp
la déclaration de la structure de pile des apériodiques est dans : .\include\proc.h).

4.5.5.5. Chargement de contexte d'une tâche périodique

Nous avons abordé dans le paragraphe 4.5.5.3. (l'interruption d'horloge temps réel) qu'une routine *int60* est exécutée, et elle sauvegarde suite à cela, le contexte en exécutant l'instruction `pushad`. Il faut savoir qu'à ce moment-là, le traitement se fait au sein d'une routine d'interruption, et que le contexte respectivement ci-après est empilé par le processeur lors de l'interruption d'horloge temps réel : EFLAGS, CS, EIP.
Ce qui revient à dire, qu'après avoir exécuté `pushad`, nous aurons tout le contexte nécessaire à l'exécution de la tâche, il suffit après quoi, de sauver le sommet de la pile contenant ce contexte, pour pouvoir le manipuler, lors du chargement de contexte. Pour cela, nous déclarons une variable globale *_esp*, qui sauvegarde le sommet de la pile après empilement du contexte de la tâche interrompue (voir figure 4.5).

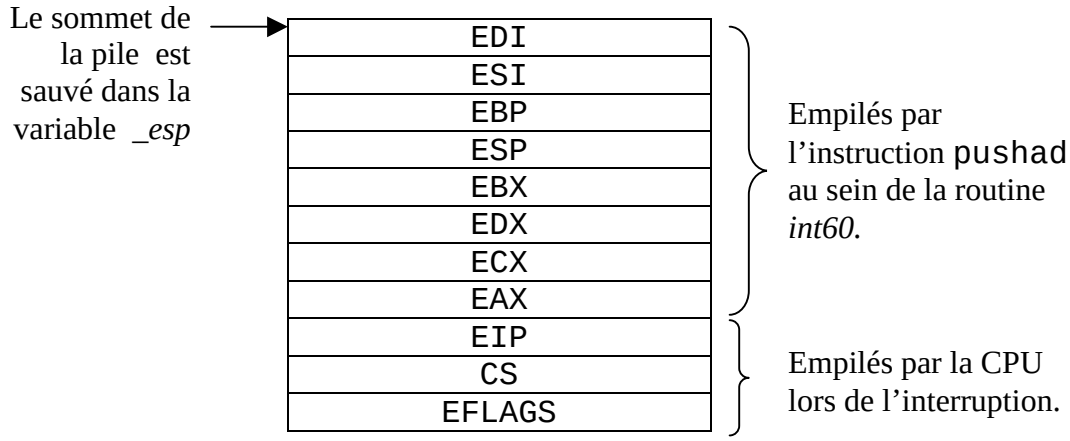


Figure 4.5. Sauvegarde du contexte lors de l'interruption d'horloge temps réel.

4.5.5.5.1. Première commutation de contexte

Nous avons abordé dans le paragraphe 4.5.5.4. (Ordonnancement des tâches périodiques) ainsi que dans le paragraphe 4.5.5.5 (Ordonnancement des apériodiques), que nous devons faire appel à la fonction *premiere_commutation*, s'il s'agit de la première exécution du scheduler, autrement dit, la première interruption d'horloge temps réel.
La fonction :

```
void premiere_commutation(struct tache * succ) ;
```

permet de sauver le contexte suivant : EFLAGS, CS, EIP dans cet ordre et à l'adresse de la zone de pile défini dans le champ *tss.esp* du contexte de la tâche *succ*. Ceci est dû au fait que lors de la première interruption, la zone de pile est celle du noyau que nous avons définie dans le *startup* (entête de la fonction main) par convention à 0x6FFFFC, or, la zone de pile consacrée aux tâches périodiques débute à 0x5FFFFC, et varie suivant l'identifiant de la tâche périodique (voir création d'une tâche périodique). Et d'autant plus qu'il n y aura pas de sauvegarde du contexte courant, étant donné que ce dernier s'agit de ce, du noyau. Une fois EFLAGS, CS, EIP sont empilés dans l'espace de pile consacré à la tâche *succ*, il faudrait par la suite redéfinir la variable globale *_esp* à l'adresse de cette nouvelle zone de pile consacrée à la tâche *succ*, et ce, en la décalant en amant par rapport aux trois informations sauvées (EFLAGS, CS, EIP) de 8 positions, pour établir la configuration donnée dans la figure 4.5.
la figure 4.6 illustre un tel chargement.

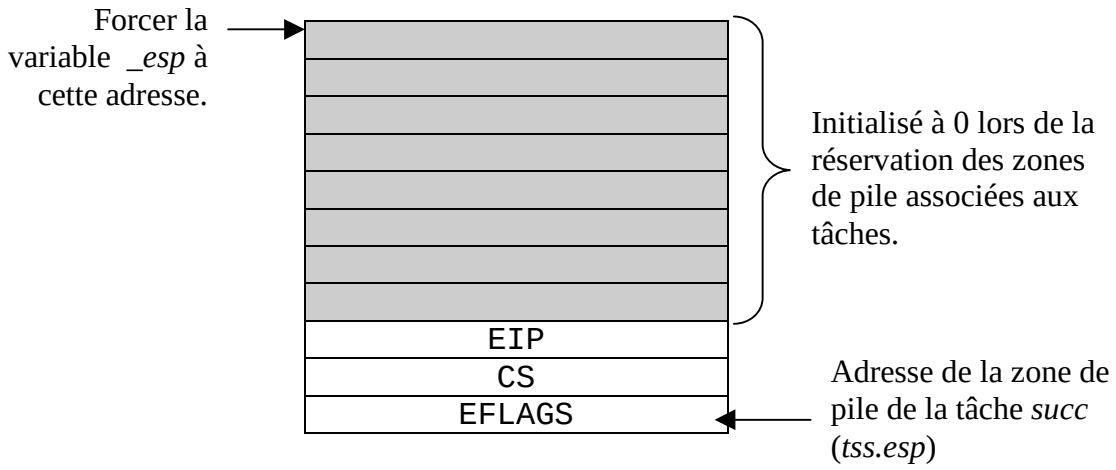


Figure 4.6. Remplissage de la zone de pile par la fonction *premiere_commutation*.

La fonction *premiere_commutation* est toujours appelée depuis la fonction d'ordonnancement *schedule_tr*. Celle-ci est appelée depuis la fonction *call_int60,* qui, à son tour est appelée par la routine *int60*. Lors du retour à cette dernière, il faut dépiler le contexte que nous avons empilé au départ (instruction `pushad`), et ce, avec l'instruction `popad`. Mais avant cela, remarquons que la zone de pile dans laquelle nous avons sauvé le contexte de la tâche est totalement différente de celle qui est en cours. C'est en partie pour cette raison que nous avons d'ailleurs déclaré la variable globale *_esp*, pour l'utiliser à forcer à ce niveau le pointeur de pile ESP de la CPU. ESP pointera désormais vers l'adresse contenue dans *_esp*, autrement dit, la pile de travail de la CPU est celle que nous avons définie dans la figure 4.6.

Dans notre cas (première commutation), en tenant compte de la configuration de la pile présentée dans la figure 4.6. l'instruction `popad` va dépiler dans les registres correspondant à la CPU, toutes les cases grisées (voir figure 4.6), ce qui équivalent à dire, que tous les registres seront initialisés à 0.

Il y aura après quoi, l'instruction `iretd`, qui s'agit bien sûr d'une instruction de retour d'une interruption. En l'exécutant, la CPU va dépiler les informations (EIP, CS, EFLAGS) de la pile, or, la zone de pile est bel et bien celle de la tâche que nous avons voulu charger par la fonction *premiere_commutation*.

4.5.5.5.2. Commutation de contextes

Un peu similaire au sous paragraphe précédent, traitant la première commutation, dans ce cas présent, il faudrait charger le contexte de la nouvelle tâche élue passé en paramètre à la fonction *commuter*, mais aussi, sauver le contexte de la tâche en cours passé également en paramètre.

La fonction :

```
void commuter(struct tache *pred, struct tache * succ) ;
```

sauvegarde le contexte de la tâche en cours *pred*, et charge le contexte de la tâche *succ*. Pour ce faire, la fonction *commuter* devra en premier lieu se positionner à l'adresse mémoire contenue dans la variable globale *_esp* (sommet de la pile du contexte en cours), puis à partir de là, lire les informations empilées (voir figure 4.5.) et les sauvegarder dans les champs *tss* de la structure du contexte de la tâche *pred*.
Une fois ceci fait, il va falloir charger le contexte de la tâche *succ* dans sa zone de pile, or, la zone de pile de la tâche *succ* est sauvée dans le champ *tss.esp*. Ce dernier va servir de pointeur de base, pour remplir les informations des champs du *tss* de la tâche *succ*. Une fois les informations du contexte de la tâche *succ* sont chargées dans sa zone de pile, il faudrait après quoi, redéfinir la variable globale *_esp*, pour qu'elle contienne la nouvelle adresse de pile, autrement dit, l'adresse du sommet de la pile de la tâche *succ* contenant le contexte ainsi chargé.

Remarque

Nous avons cité dans le paragraphe 4.5.5.5. (Ordonnancement des apériodiques), qu'une variable globale *reinitialiser_serveur* est mise à 1 pour indiquer à la fonction *commuter*, que le traitement sera particulier (tâche périodique serveur). En effet, la fonction *commuter* teste chaque fois cette variable, et si elle est à 1, alors quelques informations du contexte du serveur seront réinitialisée. Ces informations sont : *tss.eip, tss.esp, tss.eflags.* qui seront forcées respectivement aux valeurs suivantes : *&serveur_tache, _esp_serveur, 0x200*.
Qui signifient respectivement, l'adresse de la fonction ou de la routine du serveur, l'adresse de pile donnée lors de la création du serveur (réinitialisation), et la valeur EFLAGS, autorisant les interruptions.
Après cela, il va falloir remettre à 0 la variable globale *reinitialiser_serveur*.
Cette réinitilisation est nécessaire lorsque le serveur finit l'exécution d'une tâche apériodique. Rappelons que cette dernière intervient dans l'espace serveur, et qu'il faudrait alors réinitialiser à la fin du traitement de l'apériodique.

4.5.5.6. Retour de l'interruption d'horloge temps réel

Le retour depuis la routine d'interruption temps réel, se fait certes par l'instruction `iretd`, mais avant ceci, un bon nombre d'opérations doit être effectué au sein de la routine *int60*.
Tenant compte du mécanisme d'interruption du PIC 8259A, il faudrait après chaque traitement de l'interruption, envoyer à ce contrôleur le signal EOI (*End of Interrupt*), qui indique au PIC, que l'interruption est traitée, et il peut désormais servir les interruptions moins prioritaires.
Dans notre cas, l'interruption d'horloge temps réel est sur l'IRQ8, autrement dit, sur la broche 0 du PIC esclave. Ce qui veut dire encore, qu'il va falloir envoyer le signal EOI, aussi bien au PIC maître qu'à l'esclave.
Pour pouvoir envoyer EOI au maître, il faut envoyer la valeur 0x20 au port 0x20, et pour le cas de l'esclave, la valeur 0x20 au port 0xA0.

Par ailleurs, l'horloge temps réel délivre un signal périodique, mais pour qu'elle puisse le faire périodiquement, nous devons lire la valeur du registre C du CMOS en d'autres termes, envoyer la valeur 0xC au port 0x71, puis lire depuis 0x71. Cette façon de faire est imposée par le constructeur, du fait que plusieurs interruptions peuvent surgir depuis l'IRQ8 (signal d'alarme, interruption périodique …), ainsi en lisant les bits *status* du registre C, nous serons en mesure de savoir, l'identité du signal d'interruption émis, par l'horloge temps réel. Mais puisque nous n'allons pas utiliser d'autres signaux d'horloge temps réel, nous nous contenterons juste de lire le registre C du CMOS, sans pour autant vérifier l'identité de l'interruption. Une éventuelle utilisation des autres signaux de la RTC, impose à ce moment-là, la prise en compte de l'identité de l'interruption et les traitements adéquats.

Ce n'est qu'après tous ces traitements, que l'instruction `iretd` est finalement exécutée.

4.6. Interface utilisateur temps réel (API)

En plus des fonctions citées dans les paragraphes précédents, nous devons enrichir l'ensemble de ces fonctions (API) pour d'une part, offrir la possibilité à l'utilisateur de manipuler le temps (horloge logique considérée), et d'autre part, moyennant ces fonctions, l'utilisateur sera en mesure de déterminer le temps pris par chaque routine du système, en vue de respecter le déterminisme en temps. C'est ainsi que l'utilisateur pourra savoir, le temps de latence du scheduler, des interruptions …
Nous avons jugé utile de rappeler avant cela, la liste de fonctions offertes à l'utilisateur, que nous avons définie lors de la description de notre système.

4.6.1. Manipulation des tâches

```
#include <kernel.h>
#include <proc.h>
```

- `int creer_tache(THREAD thread, DATE ri, TIME ci, TIME pi,u16 num_file_sem) ;`

Permet de créer une tâche périodique/apériodique (voir paragraphe 4.5.1).

- `int creer_serveur(DATE ri, TIME ci, TIME pi) ;`

Permet de créer un serveur de tâches apériodiques (voir paragraphe 4.5.2.).

- `int faisabilite() ;`

Etablie le test de faisabilité RM, pour l'ensemble des tâches présentes dans le système (voir paragraphe 4.5.3.) .

- `int supprimer_tache (int id_tache) ;`

Permet de supprimer une tâche qu'elle soit, périodique, apériodique, ou serveur. Les tâches sont alors retirées de la file d'attente associée, et leur contexte est libéré (*magic=0*).
Retourne 0 en cas de réussite –1 sinon.

- `void suspendre_tache(u16 id);`

Permet de suspendre une tâche périodique (y compris serveur), dont l'identifiant est *id.*

- `void suspendre_tache_courante();`

Permet de suspendre une tâche périodique courante (en cours d'exécution).

- `void suspendre_tache_delai(TIME t) ;`

Suspendre la tâche en cours, durant un temps *t*, exprimé en µs. TIME est de type *long*.

- `void reprendre_tache(u16 id);`

Reactive une tâche périodique (état PRET) dont l'identifiant est *id.*

- `int tache_courante() ;`

Renvoie l'identifiant de la tâche courante.

- `etat etat_tache(unsigned int id) ;`

Renvoie l'état de la tâche *id*. (Ensemble d'états {PRET, SUSPENDU, SEM, ATTENTE}).

- `void signaler_aperiodique(unsigned int id) ;`

Voir paragraphe 4.5.5.4.3.

4.6.2. Manipulation des sémaphores

```
#include <kernel.h>
#include <proc.h>
```

- `SEMAPHORE creer_semaphore();`

Permet de créer un sémaphore.
Retourne l'adresse du sémaphore créé ou un pointeur NULL en cas d'erreur.

- `int supprimer_semaphore(SEMAPHORE sem);`

Permet de rendre la structure du sémaphore *sem,* libre pour une éventuelle allocation, cependant le sémaphore *sem* n'est pas retiré de la file des sémaphores, s'il y est.

- `int add_semaphore(int num_file,SEMAPHORE sem,long time) ;`

Rajoute le sémaphore *sem,* avec un temps d'exécution en section critique *time*, dans la file numéro *num_file* des sémaphores associés à une tâche périodique (voir paragraphe 4.5.4.1)

- `void init_file_sem();`

Initialise à 0 toutes les files des sémaphores, pour éviter tout éventuel conflit.

- `void P(SEMAPHORE s);`

Primitive de Dijkstra, pour obtenir le sémaphore *s*.

- `void V(SEMAPHORE s) ;`

Primitive de Dijkstra, pour libérer le sémaphore *s*.

4.6.3. Manipulation du temps

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

- `DATE get_absolute_time() ;`

Renvoie la valeur du compteur absolu (horloge logique) de type *double* (64-bit), qui est incrémentée à chaque interruption d'horloge temps réel (par défaut, 1024 IT /seconde).

- `DATE time2absolute(char hh, char mm, char ss);`

Convertit le format hh : mm : ss, en valeur du compteur absolu (horloge logique).
Exemple : time2absolute(31,02,11) retourne ((31*3600+2*60+11)*1024).

- `DATE date_sys2absolute(char hh, char mm, char ss) ;`

Convertit la date (l'heure) courante du système sous la forme hh : mm : ss en valeur du compteur absolu. Si la date spécifiée en paramètre est écoulée, la fonction retourne 0.

- `char lire_heure() ;`

Renvoie l'heure (hh) actuelle du système.

- `char lire_minute() ;`

Renvoie les minutes (mm) du système.

- `char lire_seconde() ;`

Renvoie les secondes (ss) du système.

- `void activer_horloge();`

Voir paragraphe 4.5.5.1.

4.6.3.1. Déterminer le temps d'une routine

Pour que l'utilisateur puisse satisfaire le déterminisme en temps, toutes les fonctions ou routines écrites dans ses applications temps réel, doivent alors être déterminées en temps d'exécution, c'est-à-dire, l'utilisateur doit être en mesure de savoir le temps pris par chaque routine qu'il a écrit. Pour les routines système (temps de latence du scheduler ou d'interruption…) celles-ci seront données à titre indicatif, mesurées sur un processeur donné, mais que l'utilisateur devra retester si l'exécutif est implanté dans un environnement matériel (CPU) différent, étant donné que le temps d'exécution d'une routine est étroitement lié à la fréquence d'horloge de la CPU sur laquelle le code s'exécute.

Pour mesurer le temps d'exécution d'une fonction, celle-ci devra être placée entre deux fonctions de mesure. La première fonction (*prologue*) de mesure sera appelée *_p_time*, et la deuxième (*épilogue*) *_v_time*.
Leurs prototypes en C sont :

```
void _p_time();
long _v_time(long frequence_cpu);
```

La fonction *_p_time* va faire appel à la fonction `rdtsc` du PENTIUM, pour lire le nombre de cycles processeur écoulés depuis la mise sous tension de la CPU, puis stocke cette valeur (64-bit) dans deux variables globales (32-bit) *_rdtsc1* et *_rdtsc2* respectivement poids faibles, et poids forts.

La fonction *_v_time* va faire également appel à la fonction `rdtsc`, pour effectuer une autre mesure. A cette dernière sera retranché les valeurs sauvées dans les variables globales *_rdtsc1* et *_rdtsc2* pour n'avoir que les cycles processeur écoulés depuis la première mesure. Il va falloir par la suite diviser la valeur obtenue (cycles processeur écoulés) sur la fréquence d'horloge de la CPU (en MHz) pour la convertir en temps (µs) et de la renvoyer.

Bien évidemment la fonction à mesurer sera placée entre les deux citées ci-dessus, et le temps écoulé entre le prologue et l'épilogue, ne sera rien d'autre que le temps de la fonction que nous voulons mesurer.

4.6.4. Routine d'affichage temps réel

Comme nous l'avons vu, le temps de chaque routine de notre système, doit être su, en vue de pouvoir garantir le déterminisme en temps. Il y va alors de même pour la routine d'affichage. Pour cette dernière, l'utilisateur doit la prendre en charge comme étant une tâche périodique (lui fixant une certaine période ou priorité). A notre niveau (noyau), nous allons écrire que la fonction qui va assurer l'E/S (affichage), Il est donc à la charge de l'utilisateur, d'implémenter les mécanismes nécessaires notamment d'envoi de messages à afficher par cette fonction.

```
#include <kernel.h>

void print(char * chaine,…) ;
```

Utilisée brut, cette fonction ne permet que de déboguer le noyau, elle n'assure cependant pas le déterminisme en temps, étant donné qu'elle sollicite le *basculeur* (sortie 16-bit) pour effectuer l'E/S par le biais des routines BIOS.

Pour y pallier, nous avons écrit une autre fonction :

```
print_tr(char * chaine, …) ;
```

qui permet d'afficher les caractères sur l'écran, en les projetant directement sur la mémoire physique vidéo 0xB8000. Ainsi nous pouvons garantir la préemption (car la routine est exécutée en mode 32-bit) et l'exclusion mutuelle, mais le temps d'exécution reste dépendant de la taille de la chaîne de caractères à afficher. En lui consacrant une tâche périodique, l'utilisateur pourrait alors éviter au sein des autres tâches effectuant l'appel d'E/S d'affichage, le temps d'attente dû à l'exclusion mutuelle et au temps d'exécution de la routine d'affichage.

Comme nous l'avons cité, la routine *print_tr*, permet d'afficher la chaîne de caractères *chaine*, en projetant ses caractères un par un dans la mémoire vidéo du mode texte (mapée en mémoire à 0xB8000). Pour pouvoir afficher un caractère, il suffit de placer celui-ci, à une adresse paire, décalée par rapport à l'adresse de base 0xB8000, dont cette dernière, représente l'onglet haut gauche de l'écran. Quant aux adresses impaires, elles représentent la couleur ou l'attribut du caractère dont l'adresse de ce dernier précède immédiatement celle de sa couleur. Pour pouvoir agir sur ces cases mémoire, nous avons défini, deux variables globales *cord_x* et *cord_y*, qui permettent de sauver l'adresse courante du curseur et qui varient respectivement de [0 – 79](colonnes) et de [0 –23](lignes).
Ainsi, il est possible de tenir compte des sauts de ligne, et ce, en incrémentant juste la variable *cord_y* et en mettant à 0 la variable *cord_x*.
Pour *scroler* l'écran, il suffit de copier les lignes d'au-dessous vers les lignes d'au- dessus. Autrement dit, copier chaque case mémoire se trouvant à l'adresse mémoire 0xB8000+X vers l'adresse 0xB8000+X - (80*2), étant donné que nous disposons de 80 colonnes et chacune de ces colonnes, se décompose en caractère + attribut.
L'exclusion mutuelle est assurée par la variable globale *sem_affich*, initialisée à 1. Cette variable est un verrou, pour empêcher d'utiliser simultanément l'affichage.

4.7. Performances obtenues

Nous allons donner à titre indicatif, les performances obtenues avec notre exécutif, sur une machine tournant sur un processeur AMD de 800Mhz.
- En présence de 20 tâches périodiques (y compris le serveur), et 10 tâches apériodiques, nous avons eu un temps d'ordonnancement (int 60h) atteignant les 18µs.
- Pour signaler l'arrivée d'une apériodique par le biais d'une interruption : 6µs.
(Avec ces conditions, la précision ou le déterminisme en temps serait alors : ±(18+6)µs).
- Les primitives sur les sémaphores P et V, ont des pics de 4µs en présence de 20 tâches périodiques utilisant toutes des sémaphores.
A noter que pour mesurer le temps de traitement du scheduler, il faudrait placer l'instruction en assembleur : `int 60h` entre les deux fonctions de mesure, et d'ôter l'instruction : `mov esp,_esp` se trouvant dans le fichier IT.asm (dans la routine int60), ceci pour ne pas permettre une commutation de contexte mais aussi inhiber les ITs avec l'instruction en assembleur `cli`.
Pour les temps d'exécution des autres fonctions (API), nous avons jugé inutile de les présenter, dans la mesure où l'utilisateur devrait effectuer lui-même, les mesures, en présence d'une horloge processeur différente de celle donnée.

Conclusion

Au terme de ce chapitre, nous avons préféré dresser des tableaux récapitulatifs, sur ce qui a été présenté dans la réalisation. Le premier tableau 4.1. montre la configuration du noyau sur disque, en indiquant la position ou emplacement des différentes parties composant le noyau, exprimé en secteurs physiques.
Quant au tableau 4.2, il indique les adresses fixes du noyau en mémoire, elles représentent d'ailleurs celles que nous avons citées dans ce chapitre, comme étant des adresses prises par convention. Le choix des valeurs de ces adresses dépend certes, de l'architecture matérielle, mais aussi de la prise en compte des éventuelles extensions.

N° du secteur physique.	**Contenu du secteur.**
Secteur 0	Le Boot
Secteur 1	Partie 16-bit du noyau
Secteur 18	Partie 32-bit du noyau

Tableau 4.1. Représentation du noyau sur disque.

Adresse mémoire Physique.	**Contenu de l'adresse**
0x6FFFFC (sens décroissant)	Sommet de la pile du noyau en mode 32-bit.
0x5FFFFC (sens décroissant)	Début d'espace des piles des tâches périodiques (2048 octets pour chaque tâche).
0x100000	**Partie 32-bit du noyau.**
0x090200	Buffer pour les fonctions en mode 16-bit. Chargé depuis le mode 32-bit.
0x090100-0x0901FF	Zone de paramètres pour fonctions en mode 16-bit. Chargé depuis le mode 32-bit.
0x090000-0x0900FF	Contexte de la CPU lors du passage 32-bit au 16-bit. Chargé depuis le mode 32-bit.
0x070000-0x080000	Libre.
0x063000 0x060000	*Basculeur 32-bit.* *Basculeur 16-bit.*
0x050000	IDT : table des Interruptions.
0x040000	GDT : table des segments
0x030000	Libre.
0x020100	**Partie 16-bit du noyau.**
0x01FFFE (Sens décroissant) 0x010001 0x010000	Sommet de la pile du noyau 16-bit. Structure contenant le matériel détecté. ID du lecteur de boot (1octet).
0x000000	Vecteurs d'interruption en mode réel.

Tableau 4.2. Représentation du noyau et ses données en mémoire physique.

Bibliographie

[Intel] : http://www.intel.com

[Ms dos] : *MS-DOS Avancé*. Deuxième édition.
(P.S.I 1989), version française : ISBN :2-7124-0618-4

[SB 03] : Samia Bouzefrane - *Les Systèmes d'Exploitation Unix, Linux, et Windows XP*
(DUNOD 2003).

Ce modeste travail nous a permis, de passer en revue les différentes techniques utilisées dans les systèmes temps réel. Il nous a donné l'idée sur les degrés de difficultés, au moment où nous nous apprêtions à mettre en œuvre les principes, les notions et politiques régissant la théorie du temps réel, à travers l'exécutif que nous avons conçu et réalisé.
Cette "aventure" nous a enrichis en matière de programmation, en développant une certaine maîtrise aux langages qui auparavant, étaient obscures, notamment le langage assembleur. A ce dernier, une connaissance de l'architecture matérielle est plus que nécessaire, pour mettre en place les éléments de base de notre noyau.
La réalité, qu'impose cette façon de faire, nous "rééduque", pour faire preuve d'abstraction devant chaque situation ou état du problème auquel nous sommes confrontés.

Partant alors de ce principe, nous avons établi le modèle en couche de notre noyau, et nous avons distingué l'utilisateur de l'exécutif, et ce dernier du matériel. Nous nous sommes basés sur l'exploitation du matériel, au profit de l'utilisateur(non final).

Au sein de l'exécutif, nous avons pris en charge les tâches temps réel (périodiques, apériodiques associées au serveur). Les tâches périodiques, peuvent utiliser des ressources critiques ou se synchroniser par des sémaphores binaires. Cette gestion de ressources est associée au protocole à priorité plafond, en vue de borner le temps d'attente à ces ressources, et prévenir notamment les situations d'interblocage et d'inversion de priorités.
L'ordonnancement des tâches périodiques est basé sur la priorité fixe de l'algorithme Rate Monotonic, à contraintes strictes. Les tâches apériodiques sont quant à elles, ordonnancées avec la tâche périodique serveur en FIFO.

Nous avons découvert durant ce travail, les différents systèmes temps réel existants sur le marché. Nous avons préféré quelques systèmes, suivant leur utilisation, mais aussi et surtout pour compléter les notions ou concepts de base temps réel que nous avons abordé. Ceci donne en partie une idée sur ce que peut être un système temps réel, dans un environnement industriel ou mobile, et dans notre cas, il pourrait servir à concrétiser les perspectives et donner une idée pratique quant au développement futur de notre système.

Perspectives

Notre exécutif temps réel est simple dans ses mécanismes de base, ceci afin d'obéir aux exigences ou aux buts fixés. Par conséquent, des perspectives pouvant être formulées, en vue de donner une idée sur ce que peut être un développement futur de notre système.

Un exécutif ou un système d'exploitation doit être développé par une équipe de programmeurs chevronnés. Ceci nous ramène à dire, que les différentes parties de notre exécutif doivent être maintenues séparément.

La partie 16-bit (mode réel) peut servir les étudiants/enseignants intéressés, au développement des applications pour le 8086 d'Intel, notamment en offrant des routines basées sur les services BIOS du système, à savoir, la prise en compte de périphériques d'E/S (Disques…), et une interface de commandes, et de former ainsi un OS simple temps partagé/temps réel.

La partie 32-bit est beaucoup plus prometteuse que la précédente, étant donné qu'elle prend en charge la technologie Pentium d'Intel, et d'autant plus qu'elle est le support de notre exécutif. Plusieurs visions en découlent, pour n'en citer que les pertinentes d'entre elles, nous pouvons penser à séparer l'espace utilisateur de l'espace noyau, pour en créer deux espaces complètement distincts, en se basant pour cela sur les nouvelles technologies d'Intel x86, qui réduisent de plus en plus le temps de commutation de contexte.
Pour les politiques d'ordonnancement, nous nous sommes limités dans notre travail, au Rate Monotonic à priorité fixe ; cependant, l'implémentation d'un algorithme d'ordonnancement à priorité dynamique serait envisageable, et il suffit à ce moment-là, de faire une extension de la fonction *get_next_tache* dans notre module d'ordonnancement.
Notre exécutif souffre du manque de communication entre tâches, puisque seule une communication par variable globale est permise. Il est alors comme perspective de développer un moyen de communication plus élaboré tel que le RPC, ou le mécanisme de boîtes aux lettres. S'ajoutant à cela, la transformation des sémaphores binaires de notre système, en sémaphores à compte.
Bien évidemment, le volume noyau serait plus important dans ces cas, et il faudrait du coup, effectuer des transformations au niveau de la fonction de chargement du noyau (partie 16-bit), de plus, une fonction de chargement de la partie 32-bit, doit être mise en place en mode protégé, afin de pouvoir assurer le chargement du reste de cette partie (partie 32-bit), puisque uniquement 64Ko immédiatement supérieures au premier méga octet (0x100000), sont accessibles (par la ligne A20) depuis le mode réel.

Par ailleurs, nous pouvons songer à transformer la partie temps réel, en temps partagé, en laissant telles qu'elles, les différentes structures de données déjà mises en place, dans la mesure où le temps partagé peut être perçu comme du temps réel dépourvu de la notion du temps, notre exécutif est d'ailleurs réalisé de sorte à supporter cette transformation. Il faudrait néanmoins pour ce faire, procéder au développement des modules comme la gestion mémoire paginée, la gestion des processus (création…), et d'ordonnancement, en impliquant notamment la gestion du multitâche hardware.

L'exécutif est conçu et réalisé entre autres à des fins pédagogiques, il appartient alors aux futurs développeurs de ce noyau, de couvrir davantage de principes des systèmes d'exploitation, et de promouvoir ceux déjà existants dans cet exécutif, pour servir cette honorable cause.

Annexes

A. Compilation de l'exécutif

A.1. Organisation des fichiers sources et utilitaires

Noyau
- ***noyau.bin :*** le noyau temps réel compilé.
- ***Compiler_exemple_PCP.bat :*** Lancer la compilation avec l'exemple PCP.
- ***Compiler_exemple_Periodiques.bat*** : La compilation avec l'exemple des périodiques.
- ***Compiler_exemple_aperiodique.bat :*** La compilation avec exemple d'apériodique.
- ***Maktreel :*** le fichier du Make.

Boot
- ***Boot.asm :*** contient le boot.
- ***Boot.inc :*** contient des macros à utiliser dans *boot.asm*
- ***Bios.inc :*** contient des macros BIOS à utiliser dans *boot.asm*
- ***Mak :*** fichier du Make.

16 Partie 16-bit du noyau.
- ***Bascul.asm :*** partie 16-bit du *basculeur*
- ***Init_cpu.asm :*** détection de la CPU (identité, fréquence…).
- ***Init_ram.asm :*** détection de la mémoire vive (étendue et conventionnelle).
- ***Init_gdt.asm :*** Initialisation de la table des segments (GDT).
- ***Mod_prtg.asm :*** Passage en mode protégé
- ***Printk.asm :*** fonction d'affichage
- ***Putchr.asm :*** fonction *putchr( )*
- ***Start16.asm :*** entête de la fonction principale : *main( )*
- ***kr16.cpp :*** fonction principale *main( )*
- ***dtct16.cpp :*** Initialisation de la machine.
- ***wrt_read.cpp :*** lecture et écriture de secteurs physiques
- ***mak :*** fichier du Make

32 Partie 32-bit du noyau.
- ***Bascul.asm :*** partie 32-bit du *basculeur*.
- ***IT.asm :*** routines d'interruption (bas niveau).
- ***Start32.asm :*** entête de la fonction *main ( )*
- ***Kr32.cpp :*** fonction principale *main ( )*
- ***Init_IDT.cpp :*** intialisation de la table des interruptions (IDT)
- ***Init_PIC.cpp :*** initialisation du PIC 8259A.
- ***Ints.cpp :*** routines d'interruption (haut niveau).
- ***_w_r.cpp :*** lecture et écriture de secteurs physiques.
- ***id_fonct :*** identificateurs de fonctions pour le basculeur.
- ***maktreel :*** le fichier du Make.

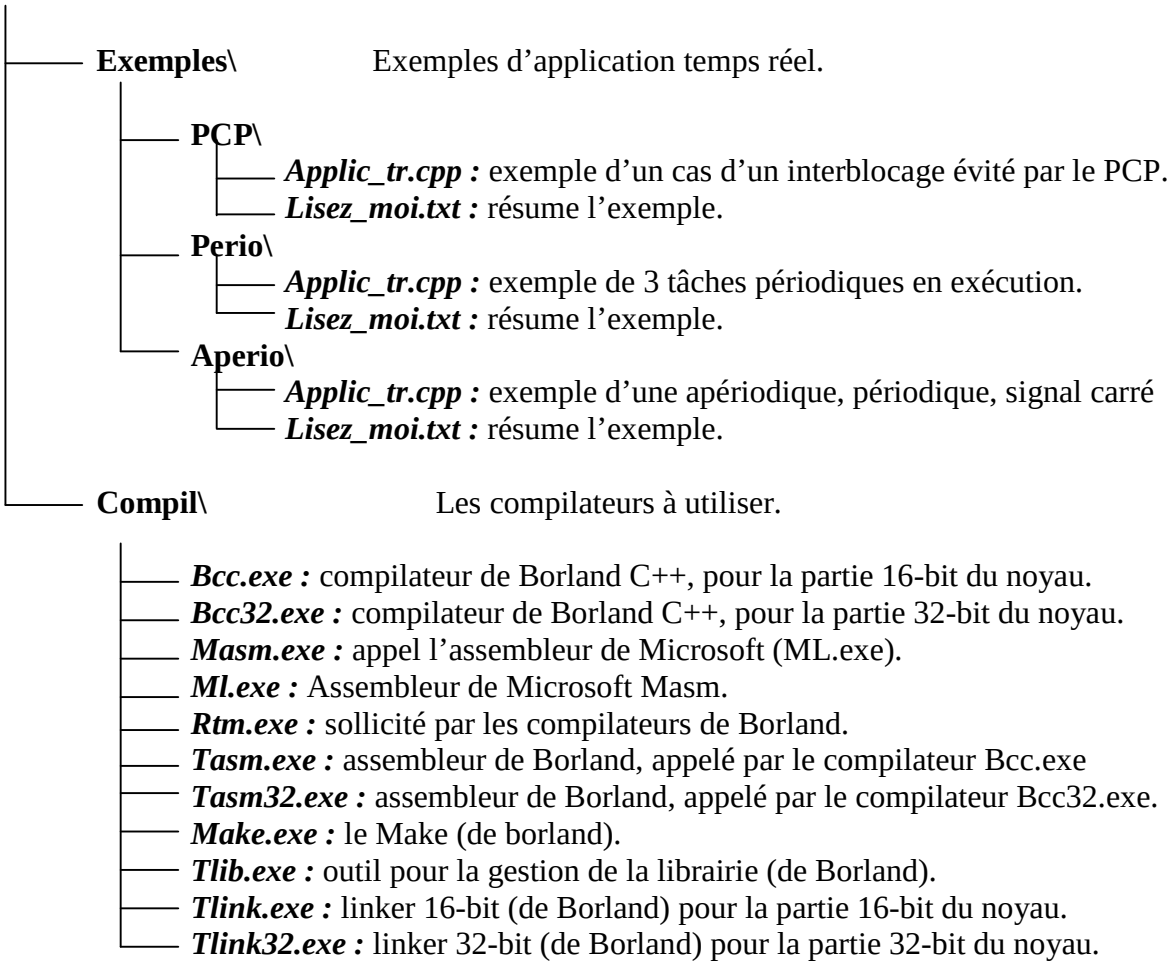
- **Treel**
  - ***tache.cpp :*** contient des fonctions pour créer/supprimer tâches/sémaphores.
  - ***Schedule.cpp :*** contient la fonction d'ordonnancement et l'API.
  - ***Rdtsc.asm :*** mesure du temps des fonctions et routine d'attente de délai.
  - ***Applic_tr.cpp :*** Contient l'application temps réel (au sein de *application_tr( )*).
  - ***Maktreel :*** fichier make de ce répertoire.
  - **Debug**
    - ***tache.cpp :*** Version déboguée de celle présentée ci-dessus.
    - ***Schedule.cpp :*** Version déboguée du scheduler.

- **Include** Les fichiers Include
  - ***Kernel.h :*** prototypes de toutes les fonctions du noyau (16-bit et 32-bit).
  - ***Kbd.h :*** la map du clavier français.
  - ***Proc.h :***déclaration de structures des tâches/sémaphores/files.
  - ***Idt.h :*** les prototypes des routines d'interruption.
  - ***Dtct16.h :*** superstructure et déclarations pour la fonction *dtct16( )*
  - ***Disq16.h :*** détection des disques et chargement de la partie 16-bit.
  - ***Define.h :*** des fonctions pour inhiber ou autoriser les interruptions.
  - ***Couleur.h :*** contient les attributs de couleur pour l'affichage

- **Utils**
  - ***Print.cpp*** *:* la fonction d'affichage temps réel.
  - ***String.cpp*** *:* les utilitaires de chaînes de caractères.
  - ***Mem.cpp*** *:* manipulation de zones mémoire.
  - ***Decasc.cpp*** *:* conversion décimal - ASCII.
  - ***Hexasc.cpp*** *:* conversion hexadécimal – ASCII.

- **Lib**
  - ***Kernel16.lib :*** Librairie de la partie 16-bit
  - ***Kernel_tr.lib :*** Librairie de la partie 32-bit et de Treel.

- **Outils**
  - ***Cat.exe :*** pour concaténer le boot avec la partie 16-bit du noyau.
  - ***Cat32.exe :*** pour concatener (boot+16-bit) avec la partie 32-bit du noyau.
  - ***Flink.exe :*** pour avoir le binaire du fichier exécutable (32-bit) sans l'entête.
  - ***Chargeur.exe :*** installer le noyau sur une disquette.
  - **Boch**
    - ***VGABIOS-elpin-2.40 :*** ROM VGA pour la machine virtuelle Bochs.
    - ***BIOS-bochs-latest :*** ROM BIOS pour la machine virtuelle Bochs.
    - ***Boch.conf :*** fichier de configuration de la machine virtuelle Bochs.
    - ***DBUG.exe :*** Machine virtuelle Bochs.
    - ***Bochsout.txt :*** fichier de sortie pour la machine virtuelle Bochs.



A.2. Les compilateurs

Comme nous l'avons souligné dans le chapitre réalisation, nous avons utilisé l'assembleur MASM (ML) version 6.11 (de Microsoft) pour les routines nécessitant des accès bas niveau. Pour le langage haut niveau (langage C), nous avons choisi, le compilateur BCC version 4.52 (de Borland) pour la partie 16-bit, et BCC32 version 4.52 (de Borland) pour la partie 32-bit. BCC et BCC32 sont respectivement accompagnés des éditeurs de liens (de Borland) TLINK et TLINK32.
BCC et BCC32 font appel respectivement aux : TASM et TASM32 (de Borland), qui sont des assembleurs, par défaut, de ces compilateurs. Puisque nous effectuons la compilation *via* assembleur, nous aurons besoins alors de ces assembleurs.
Etant donné le grand nombre de fichiers sources à compiler, et à maintenir, il va falloir adopter une méthode élaborée pour la compilation, qui consiste à utiliser un MAKE (de Borland).
Les compilateurs sont à trouver dans : Noyau\compil\

A.3. Utilitaires développés

Pour pouvoir manipuler les différentes parties du noyau (boot, 16-bit, 32-bit, treel), nous avons créé des utilitaires nous permettant d'une part, de concaténer (Noyau\Outils\cat.exe) le boot avec la partie 16-bit du noyau, autrement dit, avoir un seul fichier qui contient le boot (512 octets), suivi juste après de la partie 16-bit. D'autre part, nous allons rajouter la partie 32-bit du noyau, au fichier ainsi obtenu, mais cette fois-ci la partie 32-bit sera "éloignée" de la fin du premier fichier avec un décalage spécifié comme paramètre à l'utilitaire : Noyau\Outils\Cat32.exe. A vrai dire, ce décalage, est une réservation d'espace pour la partie 16-bit, pour une éventuelle extension.
Contrairement à Tlink.exe (de Borland), qui permet de créer un fichier binaire (sans entête de l'exécutable), Tlink32.exe (de Borland) quant à lui, ne dispose pas d'option pour avoir un binaire "pur" (sans entête de l'exécutable). La partie 32-bit et après l'avoir *linké*, contiendra un entête faisant 1536 octets (600h). Il va falloir ôter cet entête, en créant un utilitaire (Noyau\Outils\flink.exe) à qui suffit, de fournir en paramètre le fichier exécutable et la taille de l'entête, pour avoir un nouveau fichier spécifié également en paramètre, et qui ne contiendra que le code exécutable. C'est d'ailleurs ce fichier binaire qui sera utilisé et rajouté avec cat32.exe cité plus haut. Nous allons détailler ces utilitaires dans le prochain paragraphe.

A.2. La compilation

A.2.1. Le boot

Pour compiler le boot (boot.asm), il faudrait utiliser le MASM.exe, qui va générer un fichier OBJ, puis le *linker* avec Tlink.exe /t, pour avoir un fichier binaire.

A.2.2. La partie 16-bit et utils

Compiler tous les fichiers .asm (avec masm.exe) et .cpp (avec bcc.exe), et les *linker* avec Tlink.exe /t pour avoir un fichier binaire (l'adresse de base par défaut est 100h, qui coïncide avec l'adresse de la partie 16-bit en mémoire qui par convention est à 2000 :100h).
A noter qu'il faudrait définir une variable d'environnement __C16__ pour le compilateur Bcc.exe avec l'option **–D__C16__** afin d'indiquer au préprocesseur qu'il s'agit d'une compilation de la partie 16-bit, et notamment pouvoir différencier les prototypes de fonctions déclarées dans le fichier kernel.h (suivant le fait que la fonction appartient à la partie 16-bit ou la partie 32-bit). En effet, nous avons déclaré dans le même fichier (kernel.h) aussi bien les fonctions 16-bit que celles du 32-bit. De plus, il se peut qu'une même fonction puisse être déclarée dans les deux parties, par exemple les fonctions utilitaires (hexasc…). En déclarant alors la variable d'environnement __C16__ le préprocesseur va ignorer ou exclure suivant le cas, les déclarations de l'une des parties.
Il est à déclarer également l'emplacement par défaut des fichiers inclus (Include), et dans notre cas, ils se trouvent tous à l'emplacement : Noyau\Include

A.2.3. La partie 32-bit, Treel et utils

Compiler tous les fichiers .asm (avec masm.exe), .cpp (avec bcc32.exe), et les *linker* avec Tlink32.exe, en indiquant l'adresse de base à 0x100000 (adresse où se logera la partie 32-bit par convention), et ce, avec l'option **–B :0xF0000** (1000h est ajouté automatiquement à cette valeur). Un peu similaire à la partie 16-bit, il faudrait là aussi déclarer la variable d'environnement avec l'option **-D__C32__** pour que le préprocesseur puisse exclure les déclarations ou prototypes de la partie 16-bit se trouvant dans Noyau\Include\kernel.h. Déclarer également l'emplacement des fichiers inclus : Noyau\Include
En sortie, nous allons avoir un fichier exécutable (contenant un entête).

A.2.4. Concaténer le Boot et la partie 16-bit

Une fois que nous aurions compilé le boot et la partie 16-bit, et comme nous l'avons souligné, il va falloir ensuite lier ou concaténer ces deux parties pour avoir un seul fichier. Pour ce faire, nous allons utiliser l'utilitaire cat.exe que nous avons développé en C, qui consiste à se positionner à la fin du premier fichier, et écrire à partir de-là , le contenu du deuxième fichier. En sortie, le premier fichier (boot) sera modifié, et sa taille serait la taille du boot (512 octets) + la taille de partie 16-bit.
La syntaxe pour lancer Noyau\outils\cat.exe est : **cat.exe [fichier1] [fichier2]**

A.2.5. Extraire le binaire de l'exécutable 32-bit

Une fois la compilation de la partie 32-bit (partie 32-bit, Treel, Utils) est faite, il faudrait à ce moment-là extraire le binaire(sans entête) du fichier exécutable produit lors de la compilation (Etant donnée la non disponibilité de l'option sous Tlink32 (de Borland), pour produire un fichier binaire exécutable sans entête).
Pour ce faire, nous avons alors développé en C, un utilitaire : Noyau\utils\flink.exe qui consiste à se positionner (en octets) sur un emplacement dans le fichier spécifié en paramètre, puis de commencer la lecture à partir de cet emplacement, et d'inscrire en même temps dans un autre fichier spécifié également en paramètre, les données lues du premier fichier. Autrement dit, sachant la taille de l'entête d'un exécutable compilé en 32-bit (qui est de **1536 octets** convention Win32), nous pourrons l'ôter et n'avoir ainsi que le code binaire ou exécutable, dans un autre fichier.
La syntaxe de Noyau\outils\flink.exe est alors la suivante :
Flink.exe [fichier1] [fichier2] [décalage]

A.2.6. Concaténer le (boot+16-bit) avec le binaire 32-bit

Après avoir extrait le binaire de l'exécutable 32-bit, il va falloir le concaténer avec le fichier (boot+16-bit), ce dernier obtenu avec l'utilitaire cat.exe
Il faudrait néanmoins laisser un espace libre entre la fin de la partie 16-bit, et le binaire 32-bit, et ce, pour une éventuelle extension de la partie 16-bit. Notre convention est de faire fixer le binaire 32-bit, au $19^{ième}$ secteur physique sur disque (ou bien le $18^{ième}$ après le secteur 0), autrement dit, la partie 16-bit aura ainsi une taille pouvant aller jusqu'à 17 secteurs physiques (l'autre secteur est pris par le boot en première position). Rappelons que la taille d'un secteur physique est de 512 octets.

Pour ce faire, nous avons développé un utilitaire en C : Noyau\outils\cat32.exe, qui consiste à faire concaténer le binaire 32-bit spécifié en argument, avec le premier fichier (boot+16-bit) spécifié comme premier paramètre, avec un décalage du deuxième par rapport au premier exprimé en octets (d'après notre convention, le décalage serait de 9216 octets, qui est équivaut à 18 secteurs (incluant le 0) de 512 octets).
Il suffit alors de déterminer la taille du premier fichier (boot+16-bit), de se positionner à sa fin, et de remplir ou rajouter à partir de-là des "vides" (0xFF), jusqu'à atteindre la taille de 9216 octets spécifiés en argument, puis commencer à cet emplacement-là, l'écriture du fichier binaire 32-bit qui est donné en deuxième paramètre.

En sortie nous allons avoir le premier fichier (boot+ 16-bit) modifié, et contiendra également le boot+16-bit, un espace libre (de taille : 9216 - taille(boot+16-bit)), mais aussi la partie 32-bit.
La syntaxe de Noyau\outils\cat32.exe est alors la suivante :
Cat32.exe [fichier1] [fichier2] [taille à atteindre pour le fichier1]

- Bien évidemment, le fichier résultant suite à cette dernière opération ne serait rien d'autre que notre noyau, qui va se trouver dans la racine (Noyau\noyau.bin).

A.3. Le Make

Etant donné le nombre important d'opérations ou de commandes pour effectuer la compilation, il faudrait songer à établir une méthode élaborée, qui va nous faciliter la tâche d'une part, et d'autre part, éviter les erreurs qui peuvent en résulter.
Pour ce faire, nous avons choisi d'utiliser le Make de Borland. Au sein de chacune des parties à compiler, et comme l'indique l'arborescence des fichiers du noyau, nous avons écrit un fichier de commandes pour le make, afin de réaliser la compilation d'une manière automatique. Nous supposons les commandes connues, et nous donnons ci-après que le fonctionnement général ou la logique avec laquelle le Make opère.

- Le lancement du Make se fait à partir de la racine, via des fichiers BAT. Ces derniers correspondent aux exemples présents dans le répertoire Noyau\exemples\… suivant son type, le BAT va faire copier le source de l'exemple, depuis le répertoire correspondant, vers le répertoire Noyau\Treel. En remplaçant éventuellement ce, existant. Puis lance la commande .\Compil\make.exe –f –DTEST makTreel (pour lancer la compilation). Nous allons voir l'utilité de la variable d'environnement TEST lorsque nous allons aborder l'outils Bochs.

- Le fichier Maktreel se trouvant dans la racine va faire, à son tour des appels à travers des commandes du Make, pour le lancement de la compilation des parties suivantes : boot, 16, et 32 mais aussi et finalement la Concaténation, pour produire le noyau suivant la logique décrite dans le paragraphe précédent.

- Chacune des actions lancées va effectuer la compilation de la partie correspondante, et également l'édition des liens.

- La dernière action est la Concaténation, dans cette partie, le boot est concaténé avec la partie 16-bit. Le fichier produit va être concaténé avec la partie 32-bit, pour enfin avoir le noyau.bin copié dans la racine (Noyau\).

A.4. Installer le noyau sur disquette

Pour pouvoir installer le noyau (Noyau\Noyau.bin) obtenu après compilation sur une disquette, nous avons développé un utilitaire en C (compilé avec bcc.exe), qui va lire chaque fois 512 octets à partir du fichier spécifié en paramètre, et de les écrire dans des secteurs physiques à partir du secteur 0 de la disquette. Ceci en utilisant les routines BIOS de la librairie du C (16-bit) de Borland.
La syntaxe de l'utilitaire Noyau\outils\chargeur.exe est la suivante :
Chargeur.exe [fichier]

Pour éviter l'écriture de cette commande chaque fois que nous aurons à charger le noyau sur disquette, nous avons mis en place un fichier dans la racine (Noyau\), portant le nom : INSTALLER_SUR_DISQUTTE.BAT, qui fera appel au fichier chargeur.exe avec comme paramètre : noyau\noyau.bin (noyau compilé).

B. Exemples d'applications temps réel

En vue de mettre en évidence les services du noyau, nous avons implémenté quelques exemples qui feront l'objet des sous-paragraphes prochains.
Ces exemples sont loin d'être des applications temps réel proprement dites, car, celles-ci sont souvent difficiles à mettre en œuvre, puisqu'elles utilisent des capteurs qui nécessitent une prise en charge du côté noyau (drivers de périphériques), ce qui sort de l'objectif de notre travail. Nous allons prendre en charge néanmoins quelques périphériques d'E/S standards à savoir, l'affichage, le clavier, et le port parallèle (LPT).

B.1. Exemple avec PCP

Le premier exemple sera celui vu au premier chapitre (paragraphe 1.5.1.7.1. page 25) traitant l'interblocage. L'exemple va démontrer que la situation d'étreinte fatale est évitée par le protocole à priorité plafond (PCP) (c-à-d que nous allons avoir en sortie, la solution donnée dans la figure 1.15. page 29 du premier chapitre).
Pour cela, il faudrait déboguer les primitives P(s) et V(s), en affichant des informations et leur état de traitement chaque fois que l'appel est fait à ces primitives ; des informations comme : la tâche demandante du sémaphore, l'identité du sémaphore, l'héritage de priorité, et le blocage suite à une non vérification de conditions du PCP.
Il faudrait alors utiliser la version déboguée du scheduler dans : Noyau\Treel\Debug\schedule.cpp . Le lancement du fichier BAT (exemple PCP) dans la racine (Noyau\) permet de compiler automatiquement la version *debug* du scheduler en incluant le présent exemple.

Il va falloir après cela, implanter les fonctions nécessaires, et ce, dans le fichier réservé aux applications temps réel (*applic_tr.cpp*) qui sera copié lors de la compilation vers l'emplacement : Noyau\Treel\ pour être compilé.
Comme nous l'avons déjà souligné dans la partie réalisation, une fois l'initialisation de la machine est faite, un appel de fonction est effectué à *application_tr( )* en vue de lancer l'application temps réel. Elle sera considérée en quelque sorte, comme la fonction *main( ).*
Au sein de la fonction *application_tr( ),* nous avons écrit le code initialisant les deux tâches (thread1 et thread2) conformément à l'exemple du premier chapitre (interblocage -page 25) mais avant cela, et comme les deux threads sont susceptibles d'utiliser des sémaphores, il a fallu créer les sémaphores (*creer_semaphore( ))* et de **les ajouter** dans une file qui sera associée aux thread1 et thread2.
La première file, dont nous avons nous même donné le numéro 0, sera celle qui sera associée à la thread1, en vue d'utiliser respectivement les sémaphores R2 et R1. Pour cela deux appels de fonction à *add_semaphore_file* (l'ordre des appels n'est pas tenu en compte) sont nécessaires, pour déclarer avec le numéro de file 0, les sémaphores (R1 et R2), mais aussi le temps de la section critique associée pour chacun des sémaphores à utiliser, et dans notre cas, il serait de 2 unités de temps pour R1 et de 4 unités de temps pour R2. Les unités de temps sont les quantums du système , et il y a 1024tops/seconde.
Idem pour la deuxième file, dont le numéro que nous lui avons attribué est 1, et le temps d'exécution des sections critiques est de 5ut pour R1 et 2ut pour R2.

Une fois les sémaphores déclarés et insérés dans *la file des sémaphores associés aux tâches* ce n'est qu'après cela que nous pouvons procéder à la création des tâches (thread1 et thread2).

Et pour ce faire, il suffit d'appeler la fonction *creer_tache* avec les paramètres comme l'adresse de la fonction ou de la thread (thread1 ou thread2), la date de réveil (1ut pour thread1 et 4ut pour thread2), la capacité de la tâche ou sa durée d'exécution (5 pour la thread1 et 6 pour la thread2), la période de la tâche(100 pour thread1 et 101 pour thread2, ceci pour juste faire en sorte que la thread1 soit plus prioritaire que thread2), et enfin le numéro de la file des sémaphores à utiliser (comme nous l'avons déjà suggéré, le numéro de file 0 pour la thread1 et le numéro de file 1 pour la thread2).

Il faudrait tester à chaque appel de fonction *creer_tache* le numéro renvoyé par celle-ci, s'il est négatif alors il s'agit d'une erreur sinon il représente l'identifiant de la tâche.

Pour tester la faisabilité ou l'ordonnançabilité de ce système, il va falloir lancer la fonction *faisabilite( )*, qui, dans notre cas, va déterminer les facteurs de blocage de chacune des tâches, en les calculant avec l'algorithme de K.Tindell et H.Hansson donné au premier chapitre (Facteur de blocage – page 30). Une fois les facteurs de blocage calculés, la fonction *faisabilite( )* va vérifier la condition de faisabilité RM en tenant compte des facteurs de blocage calculés (voir premier chapitre – page 31). La fonction renvoie un nombre négatif en cas d'une non satisfaction de cette formule. Rappelons que cette condition est nécessaire mais pas suffisante.

Nous devons après cela, lancer l'ordonnancement de ces tâches, en faisant appel à la fonction *activer_horloge( )*.

Pour le corps de thread1 et thread2, nous avons tout simplement imité le comportement de l'exemple d'interblocage. pour respecter les délais d'attente notamment entre les primitives sur les sémaphores, nous avons utilisé la fonction *attendre_tache_delai* dont le délai passé en paramètre est exprimé en µs.

Le source de cet exemple est le suivant :

```
#include <kernel.h>
#include <proc.h>
SEMAPHORE R1;
SEMAPHORE R2;
/*********************************************************************/
thread1()
 {
  suspendre_tache_delai(1000);
  P(R2);
  suspendre_tache_delai(2000);     //attendre 2000µs = 2ms
  P(R1);
  suspendre_tache_delai(2000);
  V(R1);
  V(R2);
 }
/*********************************************************************/
thread2()
 {
  suspendre_tache_delai(1000);
  P(R1);
  suspendre_tache_delai(4000);     //attendre 4000µs
  P(R2);
  suspendre_tache_delai(2000);
  V(R2);
  V(R1);
 }
/*********************************************************************/
void application_tr()          //le noyau appelle cette fonction
 {
  int i,j,k;
   R1=creer_semaphore();
   R2=creer_semaphore();

   add_semaphore(0,R1,2);  //(la file 0, sémaphore R1, temps de la SC:2ut)
   add_semaphore(0,R2,4);

   add_semaphore(1,R1,5);  //(la file 1, sémaphore R1, temps de la SC:5ut)
   add_semaphore(1,R2,2);

i=creer_tache((THREAD)&thread1,4,5,100,0/*file 0 des sémaphores */);
if (i<0) print_rt("Erreur -%d: Création de Tache0\n",-i);
 else print_rt("Tâche %d périodique. Ressources à utiliser R2 et R1\n",i);

i=creer_tache((THREAD)&thread2,1,6,101,1);
if (j<0) print_rt("Erreur -%d: Création de Tache1\n",-j);
 else print_rt("Tâche %d périodique. Ressources à utiliser R1 et R2\n",j);

k=faisabilite();           //faisabilité RM avec facteur de blocage
if (k<0) print_rt("Faisabilité erreur: -%d\n",-k);
    else print_rt("Faisabilité ... OK\n");

print_rt("priorite de tache[%d]: %d Facteur de blocage
Bi=%d\n",i,tache[i].priorite,tache[i].sched.Bi) ;
print_rt("priorite de tache[%d]: %d Facteur de blocage
Bi=%d\n",j,tache[j].priorite,tache[j].sched.Bi) ;

 activer_horloge();   // activer l'horloge temps réel
 while(1);
}
```

B.2. Exemple de tâches périodiques

C'est un exemple disant trivial mais important, qui permet de voir l'exécution de trois tâches périodiques en exécution. Dans cet exemple encore, nous avons implémenté, ce, donné dans le premier chapitre (Exemple2 – page 20).
Les corps des threads associées aux tâches périodiques à créer, ont des boucles infinies, mais nous voulons nous accentuer sur la manière dont le scheduler fonctionne. Pour cela, nous allons utiliser la version déboguée du scheduler et de tâche se trouvant respectivement dans : Noyau\Treel\Debug\schedule.cpp et Noyau\Treel\Debug\tache.cpp. Il suffit de lancer le fichier BAT (exemple périodiques) à partir de la racine (Noyau\) pour qu'il compile automatiquement la version déboguée, en incluant le présent exemple.
Le code source (Noyau\exemples\perio\applic_tr.cpp) de cet exemple est le suivant :

```
#include <kernel.h>
#include <proc.h>

thread1()
 {
  while(1);
 }
thread2()
 {
  while(1);
 }
thread3()
 {
  while(1);
 }
/*********************************************************************/
void application_tr()      //fonction appelée par le noyau
{
  int i,j,k,l;

  i=creer_tache((THREAD)&thread1,0,3,20,0);
  if (i<0) print_rt("Erreur -%d: Cr,ation de Thread1\n",-i);
    else print_rt("* Tâche %d p,riodique. Priorit,:
%d\n",i,tache[i].priorite);

  j=creer_tache((THREAD)&thread2,0,2,5,0);
  if (i<0) print_rt("Erreur -%d: Cr,ation de Thread2\n",-i);
    else print_rt("* Tâche %d périodique. Priorité:
%d\n",j,tache[j].priorite);

  k=creer_tache((THREAD)&thread3,0,2,10,0);
  if (k<0) print_rt("Erreur -%d: Cr,ation de Thread3\n",-k);
    else print_rt("* Tâche %d périodique. Priorité:
%d\n",k,tache[k].priorite);

  l=faisabilite();     //faisabilité RM sans partage de ressources
   if (l<0) print_rt("Faisabilité, erreur: -%d\n",-l);
       else print_rt("Faisabilité, ... OK\n");

  activer_horloge();
  while(1);
}
```

B.3. Exemple de tâche apériodique et périodiques

Cet exemple va nous permettre d'une part, de comprendre la manière de créer une tâche temps réel apériodique, prise en charge par une tâche périodique particulière dite *serveur*.
D'autre part, mettre l'accent sur la différence entre notre exécutif temps réel, et les systèmes classiques (non temps réel).

♦ La tâche apériodique que nous allons créer va prendre en charge le périphérique clavier, puisque les signaux provenant de celui-ci sont apériodiques ou surviennent d'une manière aléatoire dans le temps. Son rôle est alors de lire le caractère tapé depuis le clavier, et de le mettre dans une zone tampon.

♦ Une tâche périodique sera créée pour récupérer les caractères déposés dans la zone tampon par la tâche apériodique, et les afficher. Si le tampon est vide elle se mettra en attente.

♦ Une autre tâche périodique va s'exécuter en parallèle aux tâches précédantes. Son rôle sera de générer un signal carré sur le port LPT (port parallèle) de période double à celle de la tâche qui le génère.

A vrai dire, cet exemple est une expérience qui va essayer de mettre en évidence l'importance de notre système temps réel pour certaines applications critiques, en le comparant à un autre exemple similaire mais implémenté sur un système classique (Linux et Windows98 de Microsoft).
Le but de l'expérience est alors de voir le comportement du signal carré généré par la tâche périodique sur le port LPT dans le cas de notre système, et de le comparer à celui généré dans un système classique (Linux et Windows98 de Microsoft).

Pour ce faire, il faudrait disposer d'un oscilloscope (l'idéal qu'il soit numérique), et d'un câble connecté à la deuxième broche du port parallèle, qui correspond au bit D0 de données.
Nous allons décrire à présent le corps des tâches que nous aurons à implémenter sur notre exécutif.

B.3.1. Tâche apériodique

Cette tâche comme nous l'avons déjà évoqué, va s'occuper de mettre les caractères tapés depuis le clavier, dans une zone tampon.
Pour lire un caractère depuis le clavier, dans le cas d'un IBM PC, il faudrait d'abord tester s'il est possible de la faire, ceci indiqué par le bit 1 du port 64h. Si ce bit est à 0, alors le caractère peut être lu à partir du port 60h.
La tâche apériodique *thread_clavier* va tester cette condition, puis le cas échéant lire le caractère. Si la valeur de ce dernier est inférieure à 85 (il s'agit d'un appui d'une touche et non pas une relâche, car pour ces deux états, une interruption clavier est provoquée), et que le buffer ou le tampon clavier n'est pas débordé, alors il va falloir avant d'insérer le caractère dans le buffer, le convertir suivant une table map (*kbdmap[ ]*) qui va transformer le code du caractère en valeur correspondante au *clavier français*.

B.3.1.1. Signal apériodique

Comme nous l'avons abordé dans le chapitre réalisation, la fonction : `signaler_aperiodique(unsigned short int id_tache) ;`

Est responsable de signaler ou de mettre dans la pile des apériodiques, l'identifiant *id_tache* de la tâche apériodique correspondante. Nous avons dit que cette fonction doit être appelée au sein de la routine d'interruption correspondante à la tâche apériodique.
Nous devons alors ajouter l'appel de fonction *signaler_aperiodique* dans la routine d'interruption associée au clavier (*void call_int51 ( )* dans le fichier Noyau\32\ints.cpp), cependant, la fonction *signaler_aperiodique* nécessite un paramètre qui représente l'identifiant de l'apériodique. Or, nous ne disposons pas de cette valeur ; la solution est de déclarer une variable globale : *id_aperiodique_clavier*, et de l'utiliser dans l'appel de fonction *signaler_aperiodique* se trouvant comme nous l'avons dit au sein de la routine d'interruption clavier.
Le fait de créer la tâche apériodique au sein de la fonction *application_tr ( )*, va nous délivrer son identifiant avec lequel nous allons forcer la variable globale ainsi déclarée (*id_aperiodique_clavier*).

B.3.2. Tâche périodique d'affichage

Cette tâche dont la thread est : *thread_affichage* va extraire les caractères déposés éventuellement par la tâche apériodique, pour les afficher.
Elle doit d'abord tester s'il y a un caractère dans le tampon clavier, autrement dit, tester si la valeur du pointeur bas de la file du tampon *bot_buffer_clavier* n'est pas égale à la valeur du pointeur haut de la file du tampon *top_buffer_clavier*, pour ensuite afficher le caractère se trouvant à l'emplacement *bot_buffer_clavier* par le biais de la fonction *print_rt*
Autrement, elle doit attendre, et ce, tant que les pointeurs sont égaux.

B.3.3. Tâche périodique du signal carré

Cette tâche dont le corps est *thread_signal*, va mettre tantôt à 1 et tantôt à 0, les bits de données du port LPT. Sur l'IBM PC, le port de données du port parallèle est 378h.
Une valeur *sig* initialisée à 0, va être inversée et envoyée chaque fois que la tâche est élue, sur le port LPT avec la fonction d'E/S *outpb*, que nous avons écrit dans le fichier : Noyau\Utils\32\Utils.cpp
Pour ne pas avoir une modification du signal ainsi envoyé durant la durée d'exécution de la tâche, il va falloir maintenir le niveau du signal durant la capacité de la tâche, en utilisant la fonction *attendre_tache_delai* pour ce faire. De cette manière, nous allons avoir un signal carré d'une période égale au double de celle de la tâche périodique du signal carré. Autrement dit, le signal est inversé une seule fois lors de l'élection de la tâche, puis celle-ci se bloque (pendant sa capacité d'exécution), jusqu'à la prochaine élection, pour inverser une autre fois le signal et ainsi de suite.

B.3.4. Création des tâches

La création des tâches, le test de faisabilité, ainsi que l'activation de l'horloge temps réel doit se faire au sein de la fonction *application_tr( )*, qui, rappelons-le, est appelée par notre noyau temps réel.

La tâche apériodique est créée par la fonction *creer_tache*, avec comme période 0, et une date de réveil qui ne sera pas prise en charge.
Pour pouvoir ordonnancer l'apériodique ainsi créée, il faudrait créer la tâche *serveur* avec la fonction *creer_serveur* dont les paramètres sont : la date de réveil, la capacité d'exécution et la période.
Les deux autres tâche périodiques (affichage et signal carré) seront créées de la même manière que celle vue dans les précédents exemples.

B.3.5. Résultats obtenus sur machine réelle

En posant la valeur 2ut comme période de la tâche périodique du signal carré, celle du serveur à 10ut, et de l'affichage à 15ut ; nous avons pu effectivement obtenir un signal parfaitement carré (sur un oscilloscope analogique) d'une période de 4ms.
En enfonçant des touches sur le clavier, celles-ci sont bien affichées, sans pour autant perturber le signal carré. Ceci s'explique par le fait que la tâche produisant le signal carré est la plus prioritaire dans le système, et alors aucune tâche ne la préempte.
Notons aussi le fait de "jouer" sur la période de la tâche périodique d'affichage, agit bien évidemment sur le temps de réponse d'affichage. La capacité de la fonction d'affichage quant à elle, contrôle ou influence sur le nombre de caractères à extraire et à afficher d'une traite.

Maintenant si nous rendons la tâche périodique du signal carré beaucoup plus importante que celles des autres tâches (affichage et serveur), nous observerions sur l'oscilloscope, des perturbations ou une gigue (*jitter*) sur le signal carré généré. Ceci veut dire que le signal soit, il est maintenu à 1 ou à 0, et ce, pendant que l'autre tâche (affichage ou serveur) qui a préemptée celle qui délivre le signal carré, est en exécution.
Cependant nous sommes certains d'après la vérification du test de faisabilité RM, que les prolongements des bits 1 / 0 dont nous avons parlé, ne peuvent dépasser dans le temps leur période (délai critique), ou en d'autre terme l'échéance. Celle-ci est d'ailleurs une caractéristique majeure d'un système temps réel.

En effectuant le test sur des systèmes d'exploitation classiques (Linux, Windows98 de Microsoft), nous constatons que le système génère bien au repos un signal carré avec la période spécifiée, mais dès que nous "stressons" le système (touches claviers, souris …), d'importantes perturbations sont vues sur l'oscilloscope. La gigue sur le signal carré, dépasse de loin sa valeur critique ou sa période.
Il est clair que ces systèmes ne sont pas alors des systèmes temps réel, et ne peuvent être utilisés pour contrôler des procédés à contraintes temporelles.

```
#include <kernel.h>
#include <proc.h>

#define LPT 0x378

extern u8 kbdmap[];     //contient la map du clavier

char buffer_clavier[256];
char *bot_buffer_clavier;
char *top_buffer_clavier;

extern u8 id_aperiodique_clavier;

/*****************************************************************/

thread_clavier()
{
  u8 i=0;
  while((inpb(0x64) & 0x01) == 0);
  i=inpb(0x60);
  i--;

    if ((i<=85) && (top_buffer_clavier <= &buffer_clavier[256]))
         *(top_buffer_clavier++)=kbdmap[i*4];  /*insertion du caractère
lu depuis la map clavier, dans le buffer clavier*/

    else if (i<=85)
        {
         print_rt("Buffer clavier plein …:%p\n",top_buffer_clavier);
        }
}
/*****************************************************************/

thread_affichage()
{
  char c;
  //extraction d'un caractère du tampon clavier

  while(1)
  {
      if  (bot_buffer_clavier!=top_buffer_clavier)
           {
             goto label;
           }
      else
           bot_buffer_clavier=top_buffer_clavier=&buffer_clavier[0];

    while(bot_buffer_clavier==top_buffer_clavier); /*attente d'une
insertion d'un caractère*/
    label:
      print_rt("%c",*bot_buffer_clavier++);
  }

}
```

```
/*********************************************************************/
thread_signal()
 {
  u8 sig=0;
   while(1)
   {
    outpb(LPT,sig=~sig);
    suspendre_tache_delai(1000);    /* mentient le niveau du signal
jusqu'a l'epuisement de la capacite (1ms)*/
   }
 }

/*********************************************************************/
void application_tr()
{
int i;

  i=creer_tache((THREAD)&thread_clavier,0,1,0,0);
if (i<0) print_rt("Erreur -%d: Création de l'apériodique clavier.\n",-i);
else {
      id_aperiodique_clavier=i; /* sera utilisé dans la routine
d'interruption pour signaler l'arrivée de l'IT (voir call_int51 dans
.\32\ints.cpp).*/

     print_rt("* Tâche [%d]: Apériodique associée au clavier
(Ci=%dut)\n",i,tache[i].sched.ci);
     }
  i=creer_tache((THREAD)&thread_affichage,0,1,15,0);
if (i<0)
print_rt("Erreur -%d: Création de la périodique affichage.\n",-i);
else print_rt("* Tâche [%d]: Périodique pour l'affichage (Ci=%dut
Pi=%dut)\n",i,tache[i].sched.ci,tache[i].sched.pi);

   i=creer_tache((THREAD)&thread_signal,0,1,2,0); /*pour un signal carré
sur LPT, la période est de 2ut<=> 1.953ms*/
if (i<0)
print_rt("Erreur -%d: Cr,ation de la p,riodique pour le signal carr,
(LPT).\n",-i);
else
print_rt("* Tƒche [%d]: P,riodique pour le signal carr, (LPT). (Ci=%dut
Pi=%dut)\n",i,tache[i].sched.ci,tache[i].sched.pi);

   i=creer_serveur(0,1,10);
if (i<0) print_rt("Erreur -%d: Création du serveur des apériodiques.\n",-
i);
else
print_rt("* Tƒche [%d]: Serveur d'ap,riodiques (Ci=%dut
Pi=%dut)\n",i,tache[i].sched.ci,tache[i].sched.pi);

   i=faisabilite();         // test de faisabilit, RM (ordonnançabilité)
   if (i<0) print_rt("Faisabilité erreur: -%d\n",-i);
            else  print_rt("Faisabilité du système... OK.\n");

   activer_horloge();
while(1);
 }
```

C. Emulateur du x86 d'Intel (Bochs)

Programmer un système d'exploitation n'est pas une mince tâche à faire. Des bugs peuvent survenir au cours d'exécution, sans pour autant avoir la possibilité de voir la cause du crash ou de l'anomalie. D'autant plus qu'il est pénible de relancer la machine après avoir recompilé et chargé sur disque le noyau du système d'exploitation, chaque fois que nous aurons à rectifier ou essayer une nouvelle version de ce noyau.
Heureusement l'outil Bochs existe, et tourne sur plusieurs plates-formes (Linux, Win32…), et permet d'émuler une machine IBM PC basée sur un processeur Intel 386/486.
Avec Bochs, nous pouvons désassembler le code du noyau, facilitant ainsi la maintenance, la détection des bugs, mais aussi gagner un temps considérable lors du développement.
Bochs est alors une machine virtuelle ou en d'autres termes un logiciel écrit en C++ disposant de toutes les fonctionnalités que celles présentes dans une machine réelle. C'est ainsi que nous pouvons paramètrer dans le fichier de configuration de Bochs, la quantité de mémoire RAM, les MIPS CPU,…
Nous avons testé avec le logiciel Bochs, plusieurs systèmes d'exploitation connus (Linux, Windows 98 …), l'exécution est assez lente néanmoins à cause du nombre important de calculs effectués par ce logiciel.

Deux interfaces sont présentes sous Bochs :
- Interface de commandes ;
- Interface graphique (GUI) pour visualiser les résultats (ressemble à l'écran d'une machine réelle).

C.1. Interface de commandes de Bochs

Cette interface est semblable à un shell, à travers laquelle nous pouvons lancer la simulation du noyau, faire des breakpoints, désassembler la CPU virtuelle…
Nous allons donner ci-après quelques commandes nécessaires pour lancer l'exécution et manipuler la CPU. Pour les autres commandes, taper la commande : **help**

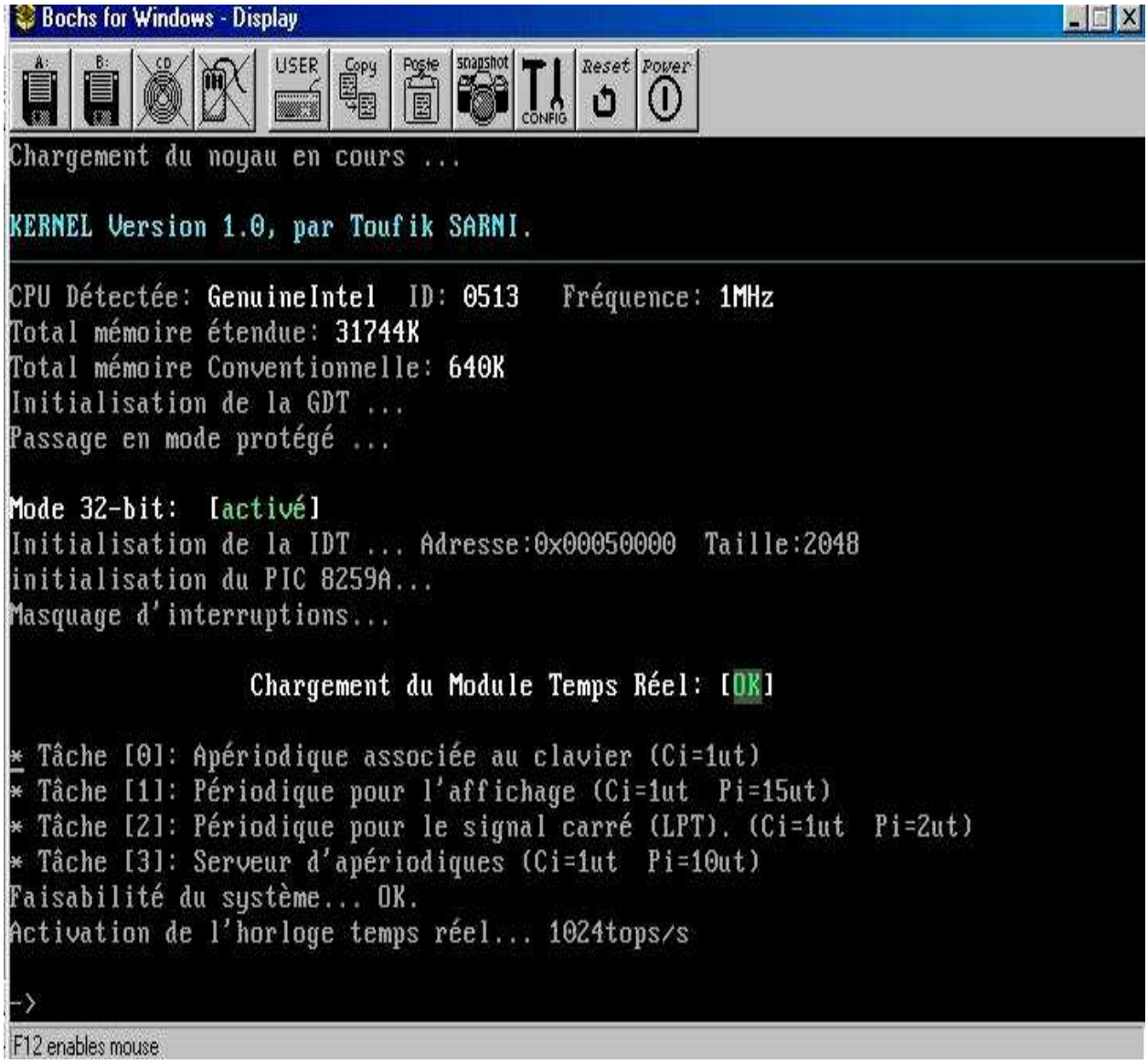
c : permet de lancer la simulation.
lb adresse_physique : permet de faire un breakpoint à l'adresse physique indiquée.
CTRL+C : permet d'interrompre la simulation lancée et retourne au débogueur.
xp adresse_physique : permet d'afficher le contenu de l'adresse physique.
print-stack : permet d'afficher la pile du processeur pointé par ESP/SP.
info reg : permet d'afficher (Dump) les registres généraux de la CPU.
info cpu : permet d'afficher tous les registres de la CPU.

C.2. Interface graphique (GUI) de Bochs

Cette interface comme nous l'avons souligné, représente l'écran simulé par Bochs. C'est ainsi que nous pouvons voir les résultats du traitement effectué par Bochs.
Il existe des commandes graphiques sur cette interface, permettant l'arrêt de Bochs, le redémarrage, une capture de l'écran Bochs dans un fichier …
Il faut noter que lors du lancement de la simulation, et si le programme simulé autorise la capture de caractères à partir du clavier, alors il est possible après avoir rendu la fenêtre de cette interface active, de taper les caractères à partir du clavier de la machine réelle.

Exemple

Voici un exemple de l'interface graphique (GUI) de Bochs lancée avec notre noyau temps réel, avec l'exemple d'apériodique donné à l'annexe B.



C.3. Les fichiers associés à Bochs

Dans l'annexe A (arborescence des fichiers), nous avons présenté dans le répertoire Boch, les fichiers qu'il contient. Ces fichiers sont une configuration du logiciel Bochs, de son BIOS pour le démarrage, de sa ROM VGA, ainsi qu'un fichier de sortie.
Tous ces fichiers doivent être spécifiés à Bochs, par le biais du fichier de configuration, qui, dans notre cas est *boch.conf*. Ce dernier contient toute la configuration de la machine, de l'emplacement du fichier image (noyau à lancer) … et doit être spécifié en paramètre avant le lancement de la simulation.

C.4. Lancer Bochs après compilation du noyau

Nous avons évoqué dans l'annexe A (Le make), qu'il y a une variable d'environnement du Make, nommée TEST, qui, suivant sa présence, décider sur le lancement de la simulation de notre noyau compilé avec l'outil Bochs.
Si la variable TEST est définie, le noyau compilé (Noyau\noyau.bin) sera lancé sur la machine Bochs, et ce, après avoir bien sûr configuré le fichier *boch.conf* pour supporter ou prendre en charge ce noyau.