وزارة التعليم العالي و البحث العلمي

BADJI MOKHTAR-ANNABA UNIVERSITY
UNIVERSITÉ BADJI MOKHTAR-ANNABA

جامعة باجي مختار- عنابة

Année : 2009/2010

Faculté des Sciences de l'Ingénieur
Département d'Informatique

MÉMOIRE

Présenté en vue de l'obtention du diplôme de Magistère

L'ordonnancement des threads dans les RTOS : Vers un nouveau protocole pour contrer le phénomène d'inversion de priorité

Filière : Informatique
Option : Informatique Industrielle

Par :
Mme DOUKHANI AMEL

Rapporteur : Mme Nacira Ghoualmi MC A Univ Badji Mokhtar-Annaba

DEVANT LE JURY

Président :	Mme Yamina Tlili	MC A Univ Badji Mokhtar-Annaba
Examinateur :	Mr Rachid Boudour	MC A Univ Badji Mokhtar-Annaba
Examinateur :	Mme Habiba Belleili	MC A Univ Badji Mokhtar-Annaba

Remerciements

Je remercie avant tout Dieu tout puissant de m'avoir guidés, éclairés, données la force, la volonté, le courage et protégés jusqu'à terminer mon modeste travail.

Mes sincères remerciements à mon encadreur;
Mme Ghoualmi Nacira pour son aide, pour ses orientations durant le développement de cette mémoire.

Je remercie également Mme Merrouani Hayette pour son accompagnement à la conférence cosi'2010 à Ouargla

J'adresse un remerciement particulier à mon mari qui m'encouragé beaucoup pour finaliser ce travail

Je n'oublie pas de remercier tous ceux qui ont contribués de loin ou de prés à la réalisation de cette mémoire.

Enfin, je suis redevable à mes parents, à qui les mots ne suffisent guère

A mon mari

A mes chers parents

A mes frères et sœurs

A mon encadreur à qui je souhaite plus de réussite dans sa vie

A mon amie Nadia à qui je souhaite une belle vie avec son marie

Résumé

Les systèmes embarqués sont de plus en plus présents dans notre vie quotidienne. De tels systèmes nécessitent généralement des tâches de priorité fixe ou variable représentant leurs degré d'urgence temporelle, ces tâches se synchronisent les unes aux autres, et collaborent pour réaliser les objectifs attendus du système. Cette collaboration se fait par l'échange de données et/ou par le partage de ressources sous contraintes de précédence et/ou de partage des ressources. Dans notre travail nous nous somme intéressé au phénomène d'*inversion de priorité* dû au partage de ressources dans l'ordonnancement temps réel. Devant ce dernier, nous introduisons un nouveau protocole d'allocation de ressources nommé *protocole à plafond* dynamique « CDP : Ceiling Dynamic Protocol ». CDP est proposé pour des systèmes temps réels, où la survenue du phénomène d'inversion de priorité et le blocage des tâches prioritaires sur une ressource partagée sont rares. L'exécution de la tâche de plus haute priorité est assurée sans aucune interruption.

Mots clés : Ordonnancement temps réel, systèmes embarqué, priorité fixe, ressources partagées, tâches périodiques, protocoles d'inversion de priorité, Plafond de priorité dynamique, Héritage de priorité, Temps mort.

Abstract

The embedded systems are increasingly present in our everyday life. Such systems generally require tasks of fixed or variable priority representing their degree urgently temporal, these tasks synchronize the ones with the others, and collaborate to carry out the awaited objectives of the system. This collaboration is done by the data exchange and/or the sharing of resource with precedence constraints and/or sharing of resource.

In our work we are interested to the phenomenon of inversion of priority due to the sharing of resource in the real time scheduling. In front of this last, we introduce a new protocol of allocation resource named *protocol with dynamic ceiling* "*CDP:Ceiling Dynamic Protocol* ". CDP is proposed for systems real times, where occurred of the phenomenon of inversion of priority and the blocking of the priority tasks on a shared resource are rare. The execution of higher priority task is ensured without any interruption.

Keywords: Real-time scheduling, Embedded system, fixed priority, shared resources, periodic tasks, protocols of inversion of priority, dynamic priority Ceiling, inheritance priority, Idle period.

الملخص

إنّ تداول الأنظمة المشحونة في حياتنا اليومية يتزايد بصورة ملحوظة، حيث أنّ هذه الأخيرة تحتاج لمهمات ذات أولوية ثابتة أو متغيرة تمثل درجة استعجالها الزمني. تشترك هذه المهمات و تتزامن لتحقيق الأهداف المنشودة من النظام و هذا بواسطة تبادل المعلومات و تقسيم الموارد فيما بينها مع مراعاة شروط الأسبقية و تقسيم هذه الموارد. إنّ العمل المنجز في هذه المذكرة يختص بمعالجة ظاهرة انعكاس خاصية أولوية المهمات أثناء تطبيق النظام و ذلك بسبب تقسيم الموارد. أمام هذه الظاهرة قمنا باقتراح قاعدة جديدة خاصة بالأنظمة ذات الوقت الحقيقي لتعيين الموارد لمختلف المهمات. فباستعمال هذه القاعدة يتقلص ظهور مشكلة انعكاس خاصية الأولوية أكثر و كذا توقيف المهمات ذات أولوية كبيرة لفائدة مهمات أخرى أقل أولوية، أمّا المهمة المختصة بالأولوية الأكبر فلا يتم توقيفها حتى انتهائها.

كلمات مفتاحية : ترتيب في الوقت الحقيقي، الأنظمة المشحونة، خاصية الأولوية الثابتة، الموارد المشتركة، المهمات الدورية، قواعد انعكاس خاصية الأولوية، الحد الأعلى لخاصية الأولوية الديناميكية، اشتقاق خاصية الأولوية، الوقت الضائع.

SOMMAIRE

Chapitre II

Ordonnancement temps réel des tâches

Chapitre III

État de l'art sur le phénomène d'inversion de priorité

Chapitre IV

Le protocole CDP proposé

Chapitre V

Conception du système

Chapitre VI

Implémentation et résultats expérimentaux

TABLE DES FIGURES

LISTE DES TABLEAUX

Introduction générale

Les systèmes embarqués sont de plus en plus présents dans notre vie quotidienne et connaissent actuellement un essor considérable. Ces systèmes sont caractérisés par une forte interaction avec les procédés contrôlés dans lesquels sont enfouis et par leurs autonomies de fonctionnement, dont l'alimentation est assurée par des batteries [Akl,09]. En plus, de tels systèmes nécessitent généralement des tâches avec des priorités représentant leurs degré d'urgence temporelle qui se synchronisent les unes aux autres, et qui collaborent pour réaliser les objectifs attendus du système. Cette collaboration se fait par l'échange de données et/ou par le partage de ressources. Cependant, cette collaboration engendre des contraintes de précédence et/ou de partage des ressources. Le problème de l'ordonnancement temps réel des tâches soumises à des contraintes temporelles (échéance) et sous contraintes de disponibilité des ressources requises (ressources partagées) dans les systèmes embarqués, a été largement étudié. Les obstacles récemment rencontré dans cette situation sont l'interblocage, l'inversion de priorité,…etc. Le problème d'inversion de priorité se rencontre dans un grand nombre d'applications industrielle (chaine de production), informatiques (programmation concurrente), aérospatial (mission Mars pathfinder 1997), …etc. Dans ce problème, il s'agit ordonnancer des tâches, sur des ressources à accès exclusives (*sections critique*). Les tâches sont liées entre eux par des relations de partage de ressource, qui expriment qu'une tâche à la fois doit entrer en section critique. Donc, la tâche T_i de plus haute priorité ne peut pas prendre une ressource R_k déjà allouée à une autre tâche T_j tant qu'elle n'est pas libérée.

L'objectif de ce mémoire, est de déterminer une solution qui minimise plus l'apparition de ce type de problème, en respectant à la fois les contraintes de temps et de partage de ressources. En dépit de tous les travaux effectués pour résoudre ce problème, il n'existe pas de solution (protocole) exactes permettant de résoudre parfaitement l'inversion de priorité. Autant d'algorithmes d'ordonnancent ont été utilisés, on peut citer l'algorithme à priorité fixe Rate Monotonique et plusieurs protocoles ont été introduits pour contrer ce problème, nous pouvons citer le protocole PIP (Priority Inheritance Protocol), OCPP (Original Ceiling Priority Protocol), ICPP (Immediate Ceiling Priority Protocol) et également SRP (Stack Resource Policy). Compte-tenu de l'algorithme d'ordonnancement statique Rate Monotonique pour l'évaluation d'une solution, la contribution de notre mémoire consiste dans un premier temps à proposer un nouveau protocole d'allocation de ressource basé essentiellement sur la réduction maximale de nombre de blocage et l'évaluation de la tâche la plus prioritaire (tâche critique) sans interruption. Dans un seconde temps, de présenté une étude comparative de ces protocoles par l'analyse de leurs caractéristiques de fonctionnement, leurs temps d'exécution et leurs difficultés d'implémentation.

Dans le cadre de ce mémoire, nous nous intéressons à un ordonnancement monoprocesseur, préemptif de tâches périodiques avec des ressources partagées de plafond de priorité dynamique. De ce fait nous proposons le protocole CDP, nous présentons à travers un exemple appliqué aux protocoles de l'état de l'art et appliqué au CDP une étude comparative des résultats expérimentaux. Nous terminons notre contribution par la réalisation d'un prototype CDP. Ainsi, ce mémoire est structuré de la manière suivante. Dans le chapitre I " *Aperçu sur les systèmes temps réel* ", nous abordons la notion de système temps réel (STR) et les différentes classes et caractéristiques de ces systèmes. Ainsi nous présentons la notion de tâche temps réel et les modèles de tâche utilisés.

Dans le chapitre II " *ordonnancement temps réel des tâches* ", après une brève définition de l'ordonnancement, de l'ordonnanceur et du contexte d'ordonnancement nous présentons les différentes techniques d'ordonnancement qui garantissent le respect des contraintes temporelles au moyen d'un test d'ordonnançabilité et les contraintes qui doivent être assurées par n'importe quelle stratégie d'ordonnancement temps réel.

Le chapitre III " *état de l'art sur le phénomène d'inversion de priorité* ", est consacré à une présentation détaillée du problème d'inversion de priorité et des différentes techniques de restriction de ce problème dans l'ordonnancement des tâches ainsi que leurs limites. Ensuite, nous abordons une étude comparative selon [Ela,02] des protocoles d'inversion de priorité.

Dans le chapitre IV " *le protocole CDP proposé* ", nous présentons la contribution « *le protocole CDP* », objet de notre investigation, nous détaillons le principe du protocole CDP. Ensuite, nous présentons les règles d'ordonnancements, les caractéristiques, les avantages et les inconvénients qui lui sont associés. Après, pour comparer le CDP avec les anciens protocoles nous montrons ses caractéristiques de fonctionnement. Finalement nous proposons un exemple et nous comparons les résultats d'application obtenus.

Dans le chapitre V " *conception du système* ", après une présentation détaillée de la problématique ainsi que la contribution, nous abordons l'architecture générale de l'application, le modèle de tâche utilisé ainsi que la modélisation de l'exécution des tâches de priorité moyenne et de l'ordonnancement des tâches. Ensuite, nous définissons les différents états-transitions que peut subir une tâche lors de son exécution avec ce protocole.

Le dernier chapitre " *implémentation et résultats expérimentaux* " est dédié aux expérimentations numériques. Après la présentation du prototype du simulateur, nous commentons les résultats obtenus de la simulation.

Nous terminons ce mémoire de Magistère par une conclusion générale sur le sujet abordé et les résultats obtenus, ainsi que les perspectives envisagées sur une suite à donner à ce travail de recherche.

Chapitre I :

Aperçu sur les systèmes temps réel

I.1 Introduction

En informatique industrielle, on parle d'un système temps réel lorsque ce système informatique contrôle (ou pilote) un procédé physique à une vitesse adaptée à l'évolution du procédé contrôlé [Liu,00]. Un système temps réel est toujours décomposé en tâches, chacune ayant des fonctionnalités, des temps d'exécution et des échéances différentes.
Ce qui implique que certaines parties des systèmes doivent fonctionner parfaitement car le non respect de quelques contraintes peut conduire à des dégâts matériels et même humaines. La caractéristique la plus importante d'un système temps réel est les restrictions imposées sur le temps de réponse [Bad,08].

Au cours de ce chapitre, nous abordons la notion de système temps réel (STR) et les différentes classes et caractéristiques de ces systèmes. Ainsi nous présentons la notion de tâche temps réel et les modèles de tâche utilisé.

I.2 Terminologies

- ***Échéance*** : est une contrainte temps à laquelle doit au plus tard se produire un événement [DEM&BON,99].
- ***Prévisibilité*** : capacité à déterminer le temps des opérations à réaliser [DEM&Bon,99].
- ***Déterminisme***: objectif à atteindre pour connaître le temps d'exécution d'un programme [DEM&Bon,99].

- ***Contrainte de temps*** : limite quantifiée sur le temps séparant deux évènements (limite min et/ou max).
- ***Temps réel*** : le temps nécessaire pour résoudre un problème ou accompli un travail.

I.3 Définition

J. A. Stankovic définit un système temps réel dans [Sta,88] comme suit : « *The correctness of the system depends not only on the logical results of computation, but also on the time at which the results are produced* », en français, un système dont le comportement dépend non seulement de l'exactitude du traitement effectué, mais également du temps où les résultats de ces traitements sont produits.

I.4 Le STR et son environnement

Les systèmes temps réel sont des systèmes qui possèdent des contraintes temporelles fortes. Lors de l'arrivée d'un stimulus en provenance de l'environnement externe, le système doit réagir dans un délai donné [Sta,88]. La prise en compte de stimulus différents, induit un parallélisme lors de la conception de tels systèmes selon une logique « événement-condition-action ». La conception aboutit généralement à un logiciel formé d'un ensemble de modules synchronisées, communicants et partageant des ressources critiques [Sta,88].

Le système TR contrôle un système plus large (*le système contrôlé*) [DEM&Bon,99]. La figure I.1 présente le système temps réel avec son environnement :

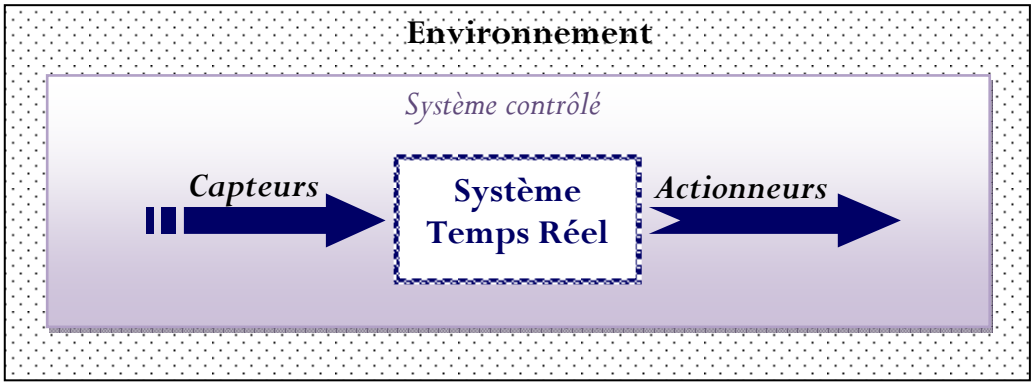

Figure I.1 : Représentation du STR avec son environnement [DEM&Bon,99]

Les *capteurs* scrutent les événements du système contrôlé, fournissent des mesures, … [DEM&Bon, 99].

Les *actionneurs* agissent sur le fonctionnement en fonction des traitements du STR sur le système contrôlé [DEM&Bon, 99].

I.5 Caractéristiques des STR

Les systèmes temps réel et notamment à contraintes dures doivent répondre à trois critères fondamentaux [DEM&Bon,99] :

1) ***L'exactitude logique :*** (exactitude des traitements)

   Les mêmes entrées appliquées au système produisent les mêmes résultats.

2) ***L'exactitude temporelle :***

   Les temps d'exécution de tâches est déterminé (les délais sont connus ou bornés et le retard est considéré comme une erreur qui peut entraîner de graves conséquences).

3) ***La fiabilité :*** (aspect matériel)

   Le système répond à des contraintes de disponibilité (matériel/logiciel).

I.6 Classification des STR

I.6.1 Classement selon l'environnement

I.6.1.1 Systèmes embarqués (enfouis)

Le concept *système embarqué* (*Embedded system* en anglais) désigne un système informatique réactif, autonome dans lequel le processeur/calculateur est intégré (enfoui ou embarqué) dans un système plus large et/ou que le logiciel est entièrement dédié à une application donnée.

Le *système embarqué* est en forte interaction avec l'opérateur et avec le procédé contrôlé dans lequel il est intégré. Cette interaction se fait via un ensemble de périphériques d'entrées/sorties appelés capteurs/actionneurs [Akl, 09] (voir Figure I.2).

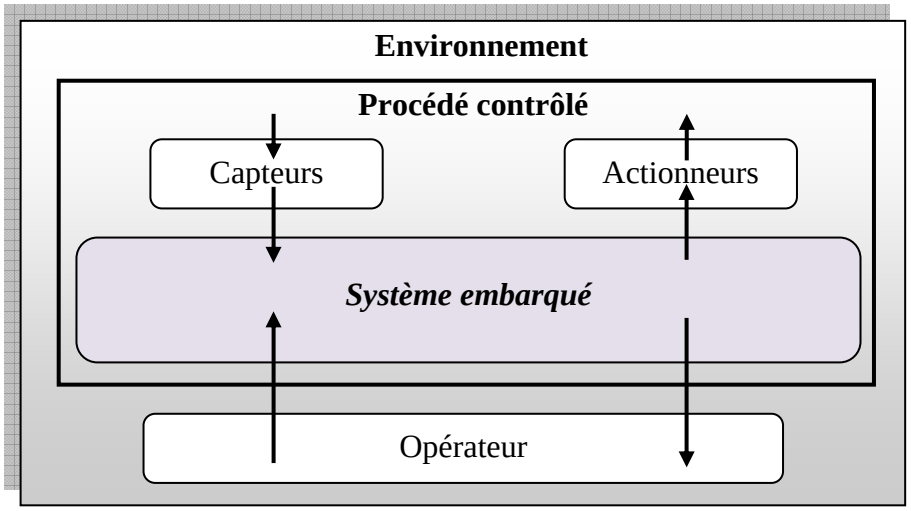

Figure I.2 : Le système embarqué et son environnement [Akl,09]

I.6.1.2 Systèmes répartis (distribués)

Un système réparti est un ensemble de machines autonomes connectées par un réseau, et équipées d'un logiciel dédié à la coordination des activités du système ainsi qu'au partage de ses ressources [Cou,Dol,Kin,94].

I.6.2 Classement selon l'échéance

Les systèmes temps réel sont classifiés par rapport à la tolérance aux échéances [DEM&Bon,99] :

I.6.2.1 STR à contrainte strict ou dur (*Hard Realtime*)

Dans ce cas l'arrivée après échéance d'un évènement attendu ne doit pas se produire c'est-à-dire toutes les échéances doivent être garanties. Parce que le dépassement des échéances cause la défaillance de l'application suivie par celle du système. Par exemple, dans le système de contrôle d'une centrale nucléaire, une défaillance peut avoir des conséquences catastrophiques, telles que la mise en danger de vies humaines et des bouleversements écologiques. Etant données les conséquences du non respect des contraintes temporelles dans les systèmes temps réel strict, il est nécessaire de vérifier avant l'exécution de tels systèmes que les contraintes temporelles seront toujours respectées [Ban,Cuy,Liv,Cab,Rou,Wei,Pua,Dec,02].

Cette vérification est menée par des méthodes d'analyse d'ordonnancement, qui nécessitent la connaissance du pire comportement temporel du système. En général, il s'agit d'un temps d'exécution pire-cas ou WCET (pour *Worst Case Execution Time*) [Dec,03].

I.6.2.2 STR à contrainte souple ou mou (*Soft Realtime*)

Dans ce cas l'arrivée exceptionnelle après échéance d'un évènement attendu ne mettra pas le système en danger c'est-à-dire quelques échéances peuvent ne pas être respectées, de manière occasionnelle. Les exemples typiques d'applications ayant des contraintes temps réel souples sont les applications multimédias. On parle alors de la qualité de service QOS (*Quality Of Service*), c'est à dire le nombre de traitements qui respectent leurs échéances. Dans ce type de système, on cherche à minimiser le nombre de fautes temporelles des tâches [Akl,09].

I.6.2.3 STR à contrainte mixte ou hybride (*Firm Realtime*)

Ce sont des systèmes temps réel à contraintes dur où une faible probabilité de manquer les limites temporelles peut être tolérée. La problématique dans ce contexte est d'éliminer d'une part toute possibilité de fautes temporelles pour les tâches strictes, et de minimiser les fautes temporelles pour les tâches à contraintes souples [Akl,09].

I.7 Domaines d'applications

Les applications temps réel couvrent un très grand domaine dont voici quelque exemple :

- Produits de grande consommation : Cafetière, machines à laver, fours à micro-onde,
- Electronique grand public : Caméras numériques, appareils photo numériques, Multimédia, téléphonie (décodeur vidéo, téléphone portables, consoles de jeu),
- Automobile : Systèmes anti-blocage de freins, contrôle moteur,
- Contrôle de procédés industriels,
- Avionique, spatial, production d'énergie (nucléaire),
- Périphériques informatique : FAX, imprimantes,
- …etc.

I.8 Le jeu de tâches temps réel

I.8.1 Définition d'une tâche temps réel

Les tâches temps réel sont des entités génériques qui ont des contraintes temporelles et des relations de synchronisation et de communication [Cli,92]. Chaque Tâche possède un délai critique, c'est le temps maximal pour s'exécuter depuis sa date de réveil. La date butoir résultante est appelée échéance (voir au dessous *Figure I.3*). Le dépassement d'une échéance est appelée faute temporelle [Bou,98].

La figure I.3 illustre les paramètres temporelle d'une tâche temps réel T_i sur l'axe de temps :

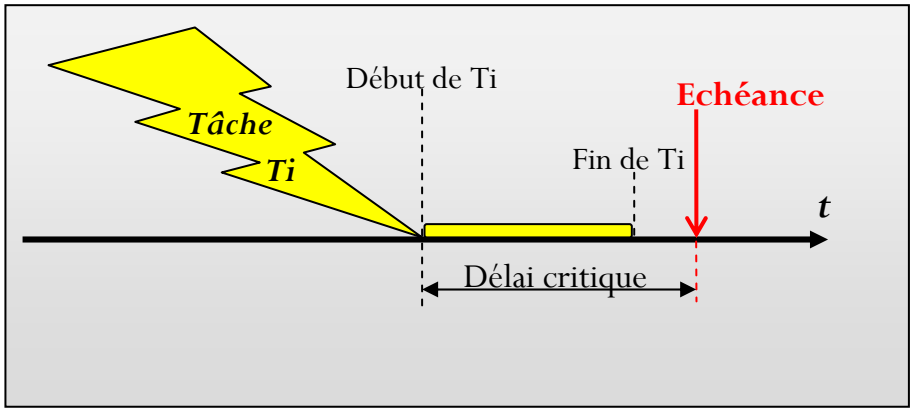

Figure I.3: Représentation des paramètres temporelle de la tâche Ti

Chaque tâche temps réel à un instant d'arrivée dans laquelle elle devient ordonnonçable. La tâche T_i peut commencer son exécution (Début de T) et se termine après l'occupation de Ci de temps processeur. T_i doit se terminer avant l'achèvement du délai critique qui est associé.

I.8.2 Les paramètres typiques d'une tâche T_i

Une tâche temps réel T_i est caractérisée par un ensemble de paramètres :

- **A_i :** Date de réveil ou d'activation de la tâche T_i dans le système (le réveil de la tâche ne signifie pas que la tâche démarre à cet instant).
- **S_i** : Date de démarrage de la tâche T_i.
- **Pe_i** : Période de réveil (si la tâche T_i est périodique).
- **C_i** : Durée de calcul.
- **D_i** : Echéance critique (*deadline*) de la tâche T_i.

Relatif à $\mathbf{Pe_i}$ si la tâche est périodique, à $\mathbf{S_i}$ si la tâche est apériodique.

- $\mathbf{P_i}$: Priorité de la tâche T_i.

- **Fi** : Date de fin d'exécution de la tâche T_i.

- $\mathbf{R_i}$: Temps de réponse; délai entre la date d'activation et sa terminaison. Il est définit comme suit:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{P_{e_j}} \right\rceil * C_j \qquad (I.1)$$

Où : *hp(i)* l'ensemble des tâches de plus forte priorité que T_i

- **UI** : C'est le taux d'utilisation du processeur. Il est définit formellement par :

$$U = \sum_{i=1}^{n} C_i / P_{e_i} \qquad (I.2)$$

Où : n est le nombre de tâches

- **DLi** : le délai de latence d'une tâche T_i, correspond au temps avant le début de l'exécution de la tâche. Dans le cas générale ce délai est égale à :

$$DL_i = S_i - A_i \qquad (I.3)$$

- **Li** : La laxité de la tâche T_i. Indique le temps restant entre la fin d'exécution de la tâche T_i et son échéance relative Di. Il s'agit de la durée dont l'instance d'une tâche T_i peut retarder son exécution sans dépasser son échéance. La laxité **Li** s'exprime par :

$$L_i = D_i - (C_i + DL_i) \qquad (I.4)$$

La figure I.4 représente les différents paramètres typiques d'une tâche temps réel T_i :

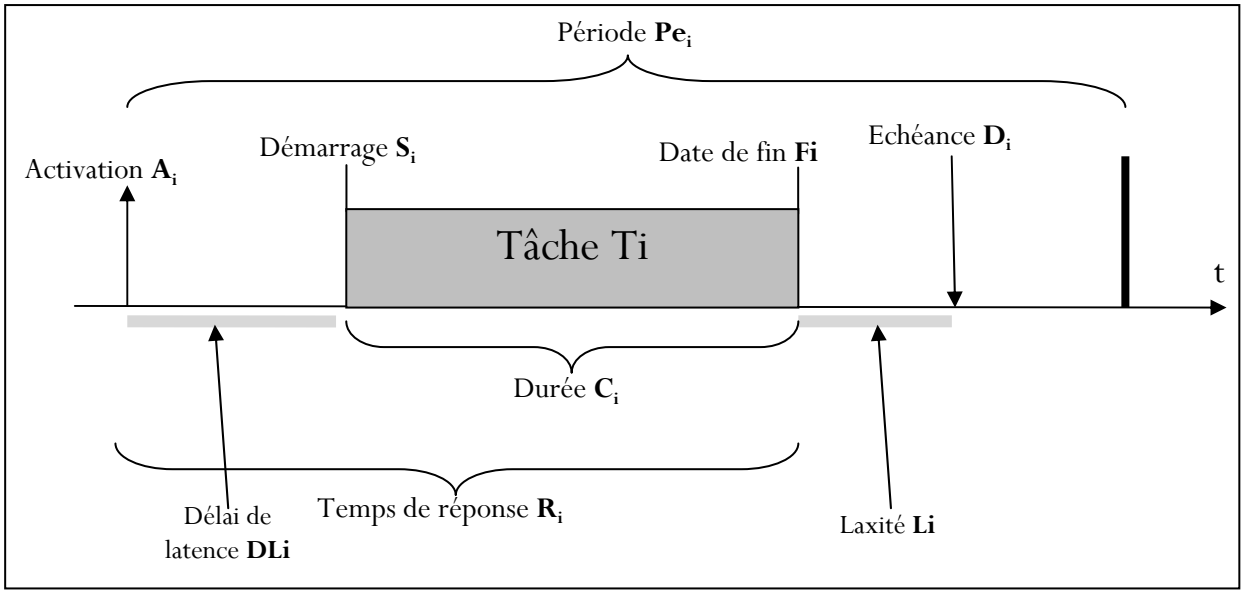

Figure I.4 : Les paramètres typiques d'une tâche T_i

I.8.3 Le cycle de vie et la gestion des tâches temps réel

L'état initial d'une tâche est inexistante. Après sa création, elle passe à l'état passive lorsque toutes les ressources à son bon fonctionnement ont été réquisitionnées. Elle reste dans cet état jusqu'à être réveillée, ce qui peut n'est fait qu'une seule fois pour une tâche apériodique, ou de manière répétée pour une tâche périodique. Après le réveil, elle se trouve dans l'état prête, où elle se retrouve en compétition avec les autres tâches disposées à être exécutées. L'ordonnanceur a alors la charge de choisir les tâches à activer. De l'état prête, le système d'exploitation peut ensuite la faire passer dans l'état élue, état dans lequel la tâche s'exécute [Ker,09][Cot, Del, Kai, Mam,00].

La figure I.5 illustre les différentes étapes de la vie d'une tâche, dans un contexte temps réel.

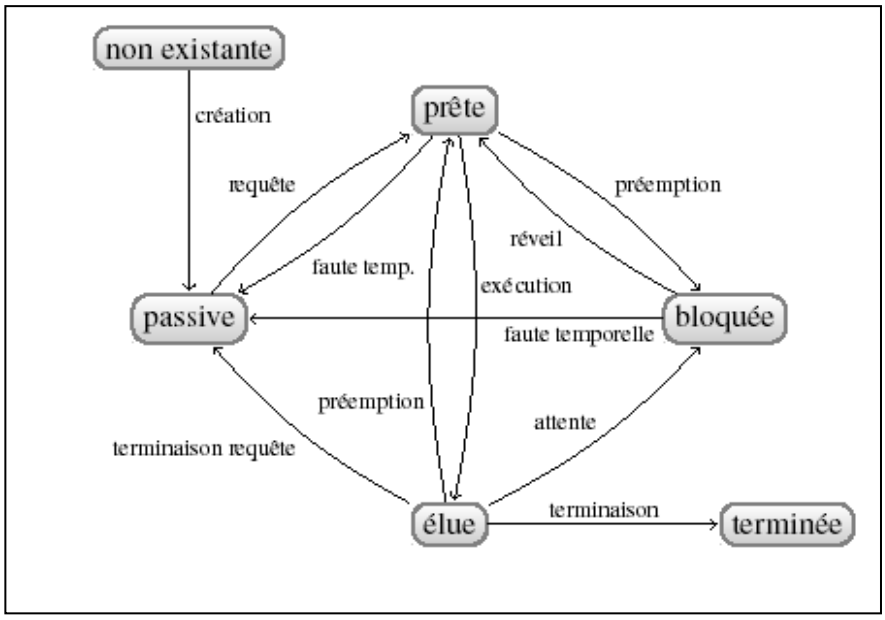

Figure I.5 : Etats et transitions d'une tâche dans un contexte temps réel [Ker, 09] [Cot, Del, Kai, Mam,00]

Ce passage n'est pas du ressort de la tâche, mais bien de l'ordonnanceur, qui s'occupe d'allouer le processeur aux différentes tâches concurrentes, et ce en suivant la politique d'ordonnancement choisie. A tout instant l'ordonnanceur peut replacer la tâche dans l'état prête, pour laisser une autre tâche s'exécuter. Il s'agit de la préemption d'une tâche, qui se fait sans que la tâche préemptée n'en soit consciente. Depuis l'état élue, la tâche peut aussi se retrouver dans l'état bloquée, lors de l'attente d'un événement ou du relâchement d'un mutex[1], par exemple. La tâche ressort de cet état par son réveil suite au relâchement d'un mutex ou au fait qu'un événement sur lequel la tâche attend a été déclenché. Dans ce cas, la tâche passe à l'état prête, prête à continuer son exécution.

Lorsque la tâche s'exécute, elle peut se terminer, et se retrouver dans l'état de terminaison. Elle peut également terminer une occurrence, et passer dans l'état passive, si elle est périodique. Il est intéressant de noter que des fautes temporelles peuvent survenir, la tâche n'ayant pas terminé son traitement à temps. Ceci peut faire passer la tâche de l'état élue ou prête à l'état passive. Elle y restera ensuite jusqu'à une nouvelle requête, pour le cas où elle est périodique [Ker,09] [Cot, Del, Kai, Mam,00].

[1] **Mut**uel **ex**clusion, une primitive de synchronisation utilisée en programmation informatique pour éviter que des ressources partagées d'un système ne soient utilisées en même temps.

I.8.4 Les modèles de tâches

Pour pouvoir ordonnancer un ensemble de tâches, le comportement temporel de chaque tâche doit être *prévisible*, c'est-à-dire connu avant son exécution. Pour cela, nous avons besoin de nous appuyer sur un modèle de tâches qui définit la façon dont une tâche est structurée (ex : présence de ressources partagées, existence de contraintes de précédence internes au tâche, …). Les tâches exécutées par un système temps réel se distinguent entre autres par leurs périodes et leurs durées d'exécution. La *période* d'une tâche est la durée qui s'écoule entre deux requêtes successives d'exécution d'une tâche. La *durée d'exécution* est le temps nécessaire à la tâche pour réaliser le travail qu'elle doit effectuer pendant une période [Bad,08].

- En fonction de leur périodicité, les tâches temps réel sont divisées en trois catégories : périodiques, apériodiques ou sporadiques [Ker,09] [Cot, Del, Kai, Mam,00] :

I.8.4.1 Les tâches périodiques (ou Cycliques)

Les tâches périodiques (ou cycliques) sont des tâches qui sont exécutées à intervalles réguliers. Elles sont entre autre définies par la période avec laquelle elles sont censées s'exécuter, ce qui est géré par l'ordonnanceur.

Une tâche périodique peut par exemple être responsable de l'observation d'un capteur à intervalles réguliers, de la régulation de moteurs, de monitoring, etc.

[Liu&Lay,73] proposent un modèle de tâches périodique dont on décrit les paramètres :

Une tâche temps réel périodique T_i est caractérisée par le quadruple (A_i, C_i, D_i, P_i) ou :

- A_i : est la date d'activation ou de réveil.
- C_i: est la charge du processeur nécessaire pour accomplir l'exécution d'une occurrence de la tâche T_i.
- D_i: est le délai critique alloué à la tâche pour se terminer après chacune de ces activations.
- P_{ei}: est la période d'activation de la tâche T_i, c'est à dire qu'après chaque P_{ei} unités de temps à partir de A_i une occurrence aura lieu.

D'après ces paramètres, on déduit les paramètres suivants:

$$U_i = {C_i}/{P_{e_i}} \qquad (I.5)$$

C'est le taux d'utilisation du processeur pour la tâche T_i.

$$U = \sum_{i=1}^{n} U_i \qquad (I.6)$$

C'est le taux d'utilisation du processeur.

I.8.4.2 Les tâches sporadiques

Elles sont définies comme étant des tâches dont on ne connaît pas à priori de date d'activation, mais dont on connaît l'intervalle minimal, séparant deux requêtes successives. Les tâches sporadiques sont très utilisées dans le contrôle de procédés dynamique rapide. Mais le fait de ne pas connaître les dates de réveil des requêtes pose des problèmes pour prédire le comportement du système [Bad,08].

I.8.4.3 Les tâches apériodiques

Elles sont définies comme étant des tâches dont on ne connait pas l'intervalle de temps minimal séparant deux requêtes successives. Par conséquent, aucun pire cas ne peut être estimé et seules les échéances souples peuvent être respectées [Bur&Foh,91].

- Au niveau de l'interaction entre tâches, les tâches sont divisées en deux catégories : dépendantes et indépendantes [Ker,09] [Cot, Del, Kai, Mam,00].

I.8.4.4 les tâches dépendantes

Il existe deux types de dépendances entre les tâches :

a) Contraintes de précédence

Une tâche doit absolument s'exécuter avant un autre. Lorsqu'une tâche à besoin de résultats produits par une autre tâche ou bien doit attendre qu'une tâche soit arrivée dans un certain état (synchronisation), l'exécution des tâches doit être ordonnée. Il s'agit d'un ordre partiel que l'on peut représenter par un *graphe de précédences*.

b) Partage de ressource

Les tâches se partagent des ressources communes de manière exclusive.

I.8.4.5 les tâches indépendantes

Ce sont des tâches qui ne sont soumises ni à des conditions de précédence, ni des conditions d'exclusion mutuelles. Les tâches indépendantes ne partagent que le processeur. Au niveau des politiques d'ordonnancement, il est clair que les dépendances entre tâches auront un rôle crucial et devront être prises en compte de manière judicieuse [Ker,09] [Cot,Del,Kai,Mam,00].

I.9 Conclusion

Les systèmes temps réel sont constitués d'un système de contrôle, d'un système contrôlé et d'un environnement. Ils se différencient des autres systèmes par la prise en compte de contraintes temporelles dont le respect est aussi important que l'exactitude du résultat, autrement dit le système ne doit pas simplement délivrer des résultats exacts, il doit les délivrer dans des délais imposés. Ces systèmes sont utilisés pour contrôler des processus physiques en suivant leur évolution au rythme de leur déroulement. La notion d'échéance dans ce type de système influe sur l'opération d'ordonnancement qui doit donc assurer que les tâches s'exécutent avant la fin du délai. Le prochain chapitre décrit en détail les politiques d'ordonnancement des tâches dans les systèmes temps réel.

Chapitre II :

Ordonnancement temps réel des tâches

II.1 Introduction

L'ordonnancement des tâches temps réel consiste à organiser dans le temps un ordre d'exécution des tâches, compte tenu de contraintes temporelles (délais, contraintes d'enchaînement) et de contraintes portant sur la disponibilité des ressources requises. Le problème d'ordonnancement dans son contexte général est étudié depuis de nombreuses années dans divers domaines comme la gestion de production, le transport, l'aéronautique, ferroviaire, …etc. De nombreux algorithmes d'ordonnancement ont été développés pour la satisfaction de ces contraintes dans les systèmes temps réel, la plupart considère un ensemble spécifique et homogène de contraintes. Ceux-ci peuvent, par exemple, prendre en compte des tâches périodiques, sporadiques ou apériodiques, acceptant ou non la préemption, dans une architecture mono ou multiprocesseur, mais ils combinent rarement ces contraintes.

Dans ce chapitre, nous présentons les différentes techniques d'ordonnancement qui garantissent le respect des contraintes temporelles au moyen d'un test d'ordonnançabilité et les contraintes qui doivent être assurées par n'importe quelle stratégie d'ordonnancement temps réel.

II.2 Définition

On désigne par ''*ordonnancement*'' le mécanisme (ensemble des règles) qui permet d'allouer le processeur à une tâche à un instant donnée.

L'ordonnancement des tâches temps réel est une opération très importante de fait qu'elle définit l'ordre d'exécution des tâches d'une manière qui prend en compte touts les contraintes de synchronisation entre tâches. Elle doit aussi assurer que toutes les tâches respectent leurs échéances.

L'ordonnanceur (*scheduler* en anglais) est un module particulier du noyau temps réel qui va effectuer le choix de la tâche à exécuter parmi les tâches prêtes [Cot&Bab,94]. Un ordonnanceur est dit *dynamique* s'il choisit la tâche courante pendant l'exécution parmi l'ensemble des tâches prêtes. Autrement, il est dit *statique*, c'est à dire que les décisions d'ordonnancement sont prises avant l'exécution.

II.3 Objectifs

- Dans un système classique, le rôle de l'ordonnancement est de :
  - Maximiser le taux d'occupation du processeur. En théorie, ce taux peut varier entre 0 % et 100 % ; dans la pratique, on peut observer un taux d'occupation variant entre 40 % et 95% selon [Cot,Del,Kai,Mam,04];
  - Minimiser le temps de réponse des tâches.
- Dans un système temps réel, les tâches sont soumises à des contraintes temporelles plus ou moins fortes, c'est-à-dire que leur exécution est assujettie à un délai maximal qui doit être respecté impérativement ou le plus souvent possible [Bad,08]. Le but de l'ordonnancement des tâches temps réel est alors de permettre le respect de ces contraintes temporelles et de prendre en compte les besoins *d'urgence*, *d'importance* et de *réactivité* des applications temps-réel**.**

L'ordonnancement, de plus, doit être *certifiable*, c'est-à-dire qu'il doit être possible de prouver à priori le respect des contraintes temporelles des tâches de l'application [Bad,08].

II.4 Contexte d'ordonnancement

II.4.1 Définition

Les contextes d'étude d'ordonnancement servent à définir au mieux les problèmes d'ordonnancement et les solutions proposées [Gro,99], ces contextes sont formalisés par la notation suivante :

χ= | *architecture* | *réveil*, *charge*, *délai critique*, *période*, *préemption, précédences*, *ressources* | .

Un contexte est une suite d'éléments nommés *termes*. Pour chaque terme du contexte correspond un certain nombre de valeurs possibles.

Le tableau II.1 représente chacun des termes du contexte avec ses valeurs possibles :

Terme	***Valeurs possibles***	***Sémantique***
Architecture	*Pr=1 : m=1*	*Système monoprocesseur centralisé*
	Pr : m =1	*Système multiprocesseur centralisé*
	Pr=1 : m	*Système monoprocesseur avec m mémoires répartis*
	Pr : m	*Système multiprocesseur répartis*
Réveil (Activation)	*Ai=0*	*Tâches simultanées*
	Ai	*Certaines tâches sont différées (à activations quelconques)*
Charge (Durée de calcul)	*Ci=1*	*Tâches à durée unitaire*
	Ci	*Tâches à durée quelconque*
Délai critique	*Di=Pei*	*Tâches à échéance sur requête*
	Di	*Tâches à délai critique quelconque*
Période	*Pei*	*Tâches périodiques*
	Ø	*Tâches non périodiques*
Préemption	*prmpt (ou Ø)*	*Tâches préemptibles*
	noprmpt	*Tâches non préemptibles*
	prmptnoprmpt	*Des tâches ont des portions de code non préemptibles*
Précédence	*Ø*	*Pas de précédences*
	pfn	*Précédences en forme normale*

	préc	*Précédence quelconque*
Ressources	*Ø*	*Pas de ressources critiques*
	res	*Ressources en exclusion mutuelle*
	resRW	*Ressources en lecture écriture*

Tableau II.1 : Termes d'un contexte d'étude d'ordonnancement [Bad,08]

Conceptuellement, chaque tâche est associée à un *processeur virtuel* comprenant son pointeur d'instructions (PC), son pointeur de pile (SP), sa zone de données. Le processeur réel commute de tâche en tâche sous le contrôle (et parfois la volonté) d'un module particulier du noyau temps réel (NTR) : l'*ordonnanceur* (*scheduler*). A chaque commutation, le processeur réel se retrouve chargé avec le contenu du processeur virtuel de la nouvelle tâche. Cette opération s'appelle un *changement de contexte* ou *commutation de contexte*. [Cot,Del, Kai, Mam,00].

II.4.2 Communication

La plupart des applications temps réel nécessitent des communications entre les tâches et imposent des contraintes de précédence entre certaines parties des tâches. L'étude de l'ordonnancement des tâches communicantes consiste à une adaptation des résultats obtenus avec les tâches indépendantes par le découpage des tâches en segments autour des points de communications. La communication entre les tâches est réalisée suivant l'architecture du système en utilisant plusieurs techniques [Bad,08].

II.4.3 Architecture cible

L'architecture du système est un critère important dans le choix de la stratégie d'ordonnancement.

a) Architecture monoprocesseur

C'est le cas le plus simple où toutes les tâches sont exécutées sur le même processeur. La communication entre les tâches est réalisée en général utilisant des variables communes et le bon fonctionnement est géré donc avec des mécanismes de synchronisations.

b) Architecture multiprocesseur

Cette architecture constitue une solution pour de nombreux domaines d'applications qui nécessitent plusieurs unités de traitement. Les architectures multiprocesseur peuvent être classées selon le nombre de processeurs dont le cas de 2 processeurs (biprocesseur) est un cas particulier qui pourrait engendre des solutions particulières. Les processeurs peuvent être homogènes (des processeurs identiques) ou hétérogènes (des différents type de processeurs). La communication dans ce cas est réalisée à base de boite aux lettres qui peuvent être tout simplement une variable partagées dans une mémoire commune [Bad,08]. La figure II.1 représente une architecture multiprocesseur :

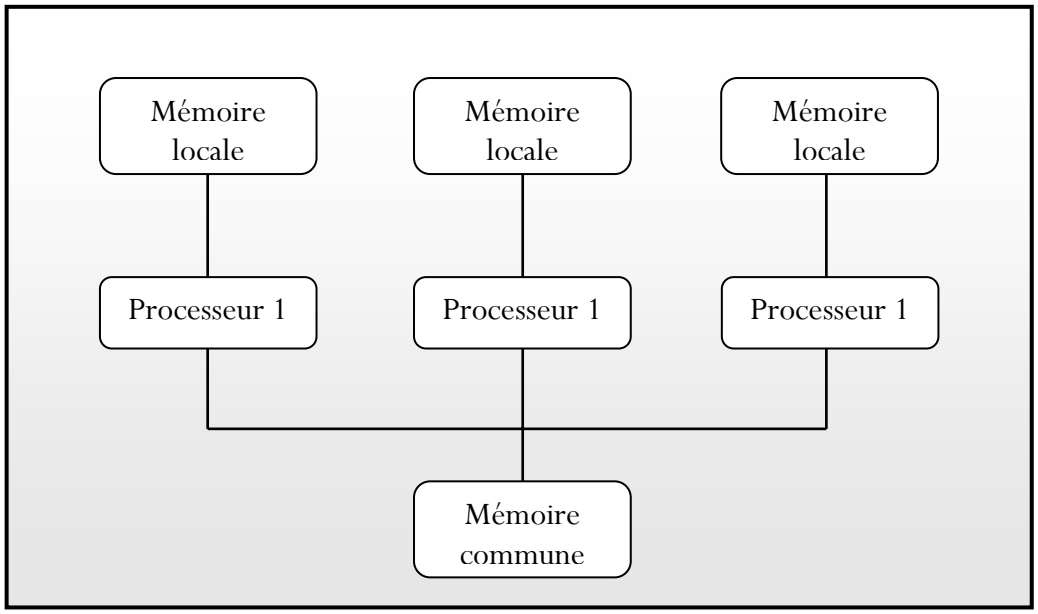

Figure II.1 : Architecture multiprocesseur [Cot,Del,Kai,Mam,04]

c) Architecture distribuée

Ce type d'architecture est devenu une exigence fréquente des utilisateurs, car la répartition des ressources et des traitements informatiques sur un système réparti peut améliorer parfaitement les performances (en terme de temps) spécialement avec la progression des nombres et de puissances des machines en parallèle à la diminution de leurs coûts. Les solutions proposées pour le cas multiprocesseur peuvent être facilement adaptées aux cas distribué mais la gestion des conflits de communication est un peu différente tandis que la seule technique adaptée est la communication avec passage de message [Pau,99].

II.5 Test d'ordonnancement et notion d'ordonnançabilité

Un test d'ordonnancement spécifique est nécessaire pour chaque algorithme d'ordonnancement pour déterminer par un calcul si possible simple et rapide, si tel algorithme peut résoudre un problème d'ordonnancement d'un ensemble des tâches sans avoir besoin d''exécuter l'algorithme.

- Un test d'ordonnancement est dit *exact* lorsqu'il est vérifie pour un problème d'ordonnancement *si et seulement si l'algorithme* d'ordonnancement peut le résoudre.
- Un test d'ordonnancement est dit *nécessaire* lorsqu'il n'est pas vérifie pour un problème d'ordonnancement si l'algorithme d'ordonnancement ne peut pas le résoudre.
- Un test d'ordonnancement est dit *suffisant* lorsqu'il est vérifie pour un problème d'ordonnancement si l'algorithme d'ordonnancement peut le résoudre.

Pour éviter d'avoir à sur-dimensionner un système temps réel, un test d'ordonnancement exact est souhaitable, mais de tels tests n'existent pas forcément, ou alors sont trop coûteux à évaluer. Souvent, ce sont donc des tests suffisants qui sont utilisés.

Un ensemble de tâches est *ordonnançable* lorsque toutes les contraintes temporelles sont satisfaites. Une condition générale nécessaire (mais pas suffisante pour tous les algorithmes) est la suivante :

$$U = \sum \frac{C_i}{Pe_i} \leq \rho \qquad \text{(II. 1)}$$

U : Taux d'utilisation (mono/multi-processeur).

ρ : Nombre de processeur disponible.

II.6 Politiques d'ordonnancement

La problématique de l'ordonnancement dans les systèmes temps réel consiste à définir une politique adéquate d'attribution des ressources (processeur, …) aux différentes tâches de l'application garantissant le respect des contraintes temporelles. En d'autres termes, c'est de définir une politique permettant la construction d'une *séquence*

d'ordonnancement valide[1] de l'ensemble des tâches à exécuter. Cette construction nécessite, d'une part, la modélisation des tâches, et d'autre part, la définition d'un algorithme d'ordonnancement. Ce dernier utilise la connaissance sur les tâches afin de définir la séquence temporelle de leurs exécutions [Akl,09].

II.6.1 Présentation

Les politiques d'ordonnancements peuvent être classées en deux classes : *ordonnancement en ligne* et *ordonnancement hors ligne*.

II.6.1.1 Ordonnancement en ligne

Dans cette classe on a deux types d'ordonnancements *: Ordonnancement de tâches indépendantes* et *Ordonnancement de tâches dépendantes*.

II.6.1.1.1 Ordonnancement de tâches indépendantes

Suivant la politique d'affectation de priorités aux tâches, les algorithmes d'ordonnancement en ligne peuvent être classés en deux types: *algorithmes à priorité fixe* et *algorithmes à priorité variable*. Lorsqu'on utilise un algorithme d'ordonnancement à priorité fixe, les tâches ne peuvent pas changer leurs priorités une fois affectée. Mais dans le cas des algorithmes à priorité variable, elles peuvent changer d'une requête à une autre et même dynamiquement pour la même requête.

II.6.1.1.1.1 Algorithmes à priorité fixe (statique)

Les algorithmes à priorités statiques se basent sur les connaissances a priori de toutes les caractéristiques temporelles des tâches. Ces algorithmes calculent les priorités de chaque tâche initialement et ne varient pas au cours du temps. Nous présentons dans ce qui suit les principaux algorithmes à priorités statiques dans un contexte monoprocesseur.

a) Rate Monotonic (RM)

Cet algorithme d'ordonnancement a été proposé la première fois dans un papier publié par Liu&Layland. Il attribue la priorité la plus forte à la tâche qui possède la plus

[1] Une séquence d'ordonnancement générée par une politique d'ordonnancement est dite valide si toutes les contraintes considérées (temporelles et synchronisations) sont respectées [AKL,09].

petite période. Il est optimal[2] dans le cadre d'un système monoprocesseur de tâches périodiques, synchrones, indépendantes et à échéance sur requête avec un ordonnanceur préemptif. De ce fait, il n'est généralement utilisé que pour ordonnancer des tâches vérifiant ces propriétés [Liu&Lay,73].

❖ ***Caractéristiques*** [Liu&Lay,73]

- RM pénalise les tâches rares mais urgentes (grande périodicité = petite priorité).
- Complexité faible et implémentation facile dans un OS.
- Algorithme optimal dans la classe des algorithmes à priorité fixe.

❖ ***Hypothèses :***

L'algorithme RM impose les hypothèses suivantes :

- Les tâches sont périodiques et dans l'état prêt à chaque début de période.
- L'échéance est à la fin de la période (tâches dites "*à échéance sur requête*" si $D_i = P_{ei}$).
- Le temps de commutation et d'ordonnancement est négligé.
- Les tâches sont indépendantes : il n'existe pas de contraintes de précédence ou d'exclusion mutuelle entre deux tâches.
- La capacité de chaque tâche est connue.
- La tâche peut être préemptée à n'importe quel instant de l'exécution.

❖ ***Condition d'ordonnançabilité***

☞ **Condition suffisante:**

Afin qu'un ensemble de tâches soit ordonnançable pour le *Rate-Monotonic*, il suffit d'appliqué le test d'ordonnancement suivant [Liu&Lay,73] :

$$U = \sum_{i=1}^{n} \frac{C_i}{P_{e_i}} \leq n * \underbrace{\left(2^{\frac{1}{n}} - 1\right)}_{\text{Constante de Liu \& Leyland}} \quad n: Nombre\ de\ tâches \qquad (II.2)$$

[2] Un algorithme est optimal s'il est capable de produire une séquence valide pour toute configuration de tâche ordonnançable [AKL,09].

Ainsi pour $n = 1$ il faut que $U \leq 1$, pour $n=2$ il faut que $U \leq 0{,}82$, Lorsque n est grand ($n \to \infty$) il faut que $U \leq Ln\ (2) = 0{,}69$ [Akl,09], et dans ce dernier cas l'utilisation du processeur doit rester inférieur à 69%.

La figure II.2 représente la variation de la condition suffisante en fonction du nombre de tâches.

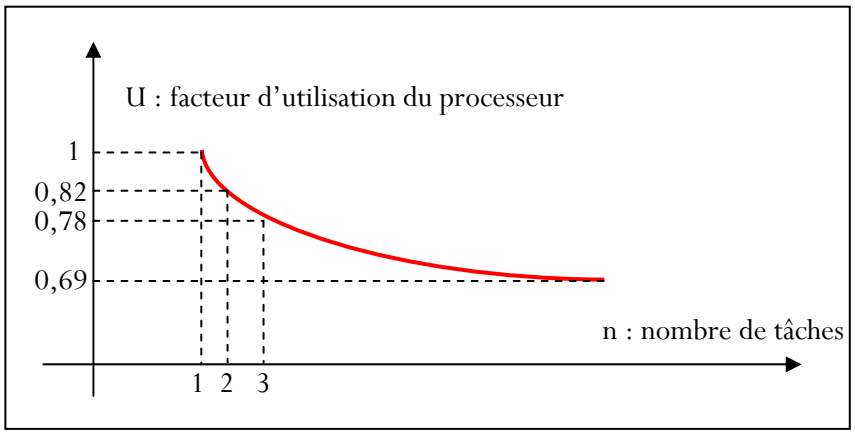

Figure II.2 : Représentation de la condition suffisante d'ordonnancement selon RM [Akl,09]

☞ **Condition nécessaire et suffisante:**

Considérons un ensemble de n tâches périodiques ordonnancé par RM; sachant que le temps de réponse d'une tâche T_i est borné supérieurement par [Jos&Pan,86] :

$$R_i = C_i + \sum_{j \in hp(i)} \left| \frac{R_i}{P_{e_j}} \right| * C_j \qquad \text{(II. 3)}$$

L'ordonnancement de l'ensemble est faisable si et seulement si:

$$\forall\, i\,, 1 \leq i \leq n,\ \ R_i \leq P_{e_i} \qquad \text{(II. 4)}$$

❖ ***L'Algorithme***

Si la(les) condition(s) d'ordonnançabilité sont satisfaites, on calcule les priorités des tâches. La priorité de chaque tâche est affectée statiquement en fonction de la période :

$$P_i = \frac{1}{P_{e_i}}\ (Inverse\ de\ la\ période) \qquad \text{II. 5)}$$

La tâche possédant la plus petite période aura la plus haute priorité. Ainsi, celle possédant la plus longue période aura la priorité la plus basse.

Lors de l'exécution, l'ordonnanceur sélectionne la tâche possédant la plus haute priorité (**HPF** : Highest Priority First).

❖ ***Exemple***

Soit le système temps réel présenté par le tableau II.2 :

Tâche	*Ci*	*Pei*	*Prio (calculée par RM)*
A	3	10	$1/10 = 0.1$
B	4	15	$1/15 = 0.06$
C	2	20	$1/20 = 0.05$

Tableau II.2 : Les valeurs des paramètres des différentes tâches constituant le STR

$$U = \frac{3}{10} + \frac{4}{15} + \frac{2}{20} = 0.666 \leq 3 * \left(2^{\frac{1}{3}} - 1\right) = 0.779$$

✓ Condition d'ordonnançabilité RM satisfaite.

Même lorsque la condition suffisante n'est pas respectée pour une configuration de tâches donnée on ne peut rien dire sur l'ordonnancement de la configuration car la borne supérieure de la charge des tâches a été fixée expérimentalement aux alentours de 88% dans **[Leh, sha&Din,89]**.

❖ ***Temps de réponse d'une tâche***

C'est le délai entre l'activation d'une tâche et sa terminaison. Au pire cas, son temps d'exécution plus son temps d'attente lorsque des tâches plus prioritaires s'exécutent.

Soit *hp(i)* l'ensemble des tâches de plus forte priorité que *i*.

$\lceil x \rceil$ correspond à la partie entière de *x*.

Le temps de réponse est définit comme suit:

$$R_i = C_i + \sum_{j \in hp(i)} \left| \frac{R_i}{P_{e_j}} \right| * C_j \qquad \text{(II. 6)}$$

Et comme cette équation récurrente est difficile à résoudre on utilise la technique de calcul suivante :

On évalue *Ri* de façon itérative avec w_i^n

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left| \frac{w_i^n}{P_{e_j}} \right| * C_j \rho \qquad \text{(II. 7)}$$

On démarre avec $w_i^0 = C_i$

Si $w_i^n > P_{ei}$ alors on s'arrête avec un échec,

Sinon on s'arrête avec réussite lorsque $w_i^{n+1} = w_i^n$

La convergence est assurée tant que :

$$\sum \frac{C_i}{P_{e_i}} \leq 1 \qquad \text{(II. 8)}$$

Avec l'exemple précédant on peut calculer le temps de réponse R_A, R_B, R_C des trois tâches A,B,C comme suit :

R_A : $w_A^0 = 3 \qquad hp(A) = \emptyset \quad \rightarrow \; R_A = 3$

R_B : $w_B^0 = 4$

$w_B^1 = 4 + \left| \frac{4}{10} \right| * 3 = 7$

$w_B^2 = 4 + \left| \frac{7}{10} \right| * 3 = 7 \,, w_B^2 = w_B^1 \qquad \rightarrow \quad R_B = 7$

R_C : $w_C^0 = 2$

$w_C^1 = 2 + \left| \frac{2}{15} \right| * 4 \; + \left| \frac{2}{10} \right| * 3 = 9$

$w_C^2 = 2 + \left| \frac{9}{15} \right| * 4 \; + \left| \frac{9}{10} \right| * 3 = 9 \,, w_C^2 = w_C^1 \qquad \rightarrow \quad R_C = 9$

❖ ***Inconvénients* :**

Le rythme d'arrivée des tâches et leur temps d'exécution doivent être bien définis [DEM99].

b) Deadline Monotonic (DM)

L'algorithme RM qui est optimal dans le contexte où les tâches sont à échéance sur requête ($D_i=P_{ei}$) ne prend pas en compte les tâches dont les échéances sont inférieures aux périodes ($D_i < P_{ei}$). Pour pallier ce manque, Leung et al [Leu&Whi,82] ont proposé un algorithme à priorité statique nommé Deadline Monotonic noté DM. DM est optimal dans le contexte $|1|Ai=0, C_i, D_i, P_{ei}|$ dans la classe des algorithmes à priorités statiques. Cet algorithme n'a vraiment été utilisé que depuis qu'une condition suffisante d'ordonnançabilité a été présentée dans [Aud&Bur,90].

❖ ***Caractéristiques:***

↳ DM sera meilleur pour les tâches dont l'échéance est très inférieure à la période.

❖ ***Hypothèses***

↳ Identiques à celles du *Rate-Monotonic*, mais avec la possibilité d'avoir des échéances inférieures à la période ($Di < Pei$).

❖ ***Condition d'ordonnançabilité***

La condition (suffisante) d'ordonnançabilité devient:

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n * \left(2^{\frac{1}{n}} - 1\right) \quad n: Nombre\ de\ tâches \qquad (II.9)$$

La tâche doit se terminer avant l'échéance *Di*

❖ ***L'Algorithme***

Si la(les) condition(s) d'ordonnançabilité sont satisfaites, on affecte les priorités inversement proportionnelles aux délais critiques des tâches. Ainsi, une tâche T_i de délai critique D_i sera plus prioritaire qu'une tâche T_j de délai critique D_j si $D_i<D_j$.

Lors de l'exécution, l'ordonnanceur sélectionne la tâche avec la plus haute priorité.

❖ **Exemple**

Considérons une configuration constituée de deux tâches indépendantes décrites dans le tableau II.3:

Tâche	*Ai*	*Ci*	*Pei*	*Di*	*prio*
A	0	1	8	1	2
B	0	1	6	2	1

Tableau II.3 : Configuration de tâches périodiques avec affectation de priorités selon DM.

$$U = \frac{1}{8} + \frac{1}{6} = 0.292 \leq 2 * (2^{\frac{1}{2}} - 1) = 0.828 \leq 1$$

Si on vérifie la condition (suffisante) d'ordonnançabilité pour DM :

$$\frac{1}{1} + \frac{1}{2} = 1{,}5 > 2 * (2^{\frac{1}{2}} - 1) = 0.828$$

On remarque que la condition d'ordonnançabilité suffisante n'est pas vérifie. Par conséquent, on ne peut pas déduire que cette configuration de tâches n'est pas ordonnançable. Ceci est dû au fait que le facteur d'utilisation *U* est inférieur à 1 (condition nécessaire d'ordonnancement d'une configuration).

II.6.1.1.1.2 Algorithmes à priorité variable (dynamique)

Pour pallier les problèmes des limites théoriques de charge processeur imposées aux algorithmes d'ordonnancement RM et DM, qui sont optimaux chacun dans un contexte particulier, des algorithmes d'ordonnancement à priorités dynamiques ont été proposés. Dans cette classe d'algorithmes, l'ordre de priorités de tâches peut changer durant l'exécution en assurant chaque fois que la tâche la plus urgente[3] qui sera exécutée le plus tôt.

a) Earliest Deadline First (EDF)

❖ ***Caractéristiques :***

- Priorités dynamiques des tâches ;
- Supporte des tâches périodiques et les tâche apériodiques ;

[3] Qui a la plus proche échéance dans le temps.

- Le travail dont le résultat est nécessaire le plus rapidement est exécuté d'abord ;
- Algorithme optimal dans la classe des algorithmes à priorité dynamique (utilise jusqu'à 100 pourcent de la ressource processeur).

❖ ***Hypothèses***

- Echéances quelconque (Di ≤ Pei), mais connues au réveil.
- L'algorithme consiste à sélectionner la tâche qui a l'échéance la plus proche dans le temps.
- Les priorités sont dynamiquement attribuées en fonction des échéances, au fil du temps.

❖ ***Condition d'ordonnançabilité***

Afin qu'un ensemble de tâches périodiques, avec échéance avant requête $(\forall i : D_i < P_{e_i})$ soit ordonnançable pour EDF, il suffit selon [Leu&Whi,82] que :

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq 1, si\ \exists\ i :\ D_i \leq P_{e_i}\ ;\ \ n: Nombre\ de\ tâches \qquad (II.10)$$

Si l'ensemble de tâches sont périodiques et avec échéance sur requête $(\forall i :\ D_i = P_{e_i})$ la condition devient "*nécessaire* et *suffisante*". Dans ce dernier cas le processeur peut être occupé à 100%.

❖ ***L'Algorithme***

Si la(les) condition(s) d'ordonnançabilité sont satisfaites, on calcule l'échéance de chaque tâche comme suit :

Soit Di(t), l'échéance à l'instant t et Di, le délai critique de la tâche T_i :

✓ Pour une tâche apériodique :

$$D_i(t) = D_i + S_i \qquad (II.11)$$

Où : S_i est la date de démarrage de la tâche T_i

✓ Pour une tâche périodique :

$$D_i(t) = D_i + date\ de\ début\ de\ l'activation\ courante\ \text{à}\ l'instant\ t \qquad (II.12)$$

Lors de l'exécution, l'ordonnanceur sélectionne la tâche avec la plus courte échéance d'abord.

❖ ***Avantages***

- Optimal en ce qui concerne l'utilisation du processeur (utilisation à 100%).
- Aptitude à traiter des tâches apériodiques.

❖ ***Inconvénients***

- Implémentation difficile, il faut réajuster les priorités au fil du temps en testant les différentes échéances.
- Le temps de réponse ne se calcule pas aisément.
- Une échéance manquée provoque une avalanche de retards d'échéances [DEM 99].

b) Least Laxity First (LLF)

L'algorithme Least Laxity First [Liu&Lay,73] noté LLF est basé sur une affectation dynamique de priorités aux instances de tâches au cours de leurs exécutions en fonction de la laxité dynamique L(t) qui représente le temps restant avant la prochaine occurrence de la date de réveil d'une tâche. La tâche à exécuter à un instant *t* est celle dont la laxité dynamique est la plus petite, c'est à dire celle dont l'exécution peut être la moins retardée. D'après la définition de LLF, le calcul des priorités doit être effectué à chaque modification de laxité, donc à chaque exécution d'une unité de temps, ce qui engendre un surcoût processeur conséquent avec la génération d'un nombre important de changements de contexte.

L'algorithme LLF est optimal dans le contexte $|1|Ri, Ci, Di, Pei|$ (tâches indépendantes à échéances inférieures ou égales aux périodes) [Mok,83]. Il n'est toutefois pas très intéressant en monoprocesseur comparé à EDF qui engendre moins de changements de contexte [Pai,06].

❖ ***Inconvénients*** **:**

- Cet algorithme présente l'inconvénient du calcul de priorité à chaque unité de temps, ce qui implique une charge de processeur plus élevée [Bad,08].
- Il faut estimer le temps nécessaire pour chaque travail [DEM,99].

Malgré la flexibilité des algorithmes dynamiques (permettant de prendre en compte l'arrivée imprévue d'événements) et leur optimalité, ces algorithmes consomment beaucoup de temps système (recalcule les priorités des tâches).

II.6.1.1.2 Ordonnancement de tâches dépendantes

Les algorithmes d'ordonnancements vus précédemment reposent sur l'affectation de priorités à des tâches indépendantes. Cependant, les tâches composant une application temps réel collaborent pour réaliser les objectifs attendus du système. Cette collaboration se fait soit par l'échange de données (communication) ou par le partage de ressources. Ceci engendre des contraintes de précédence et de partage de ressources dans le système, lesquelles, il faut prendre en considération dans la vérification de l'ordonnancement du système.

II.6.1.1.2.1 Ordonnancement avec des contraintes de précédences

Au lieu de mettre en œuvre de nouveaux algorithmes d'ordonnancement des tâches liées par des contraintes de précédence avec l'adaptation des conditions d'ordonnancement, la communauté temps réel a pris le parti de transformer l'ensemble de tâches avec des relations de précédence en un ensemble de tâches indépendantes, en intégrant explicitement ou implicitement les relations de précédence dans l'affectation de priorité afin de pouvoir utiliser les algorithmes étudiés précédemment [Akl,09]. Notons que par hypothèse, les tâches liées par des contraintes de précédence ont la même période.

a) RM avec des contraintes précédence

L'algorithme d'ordonnancement RM affecte des priorités aux tâches suivant leurs périodes. En d'autres termes, la tâche de plus courte période se voie assignée une haute priorité. Respectant cette règle, les paramètres temporels des tâches sont modifiés afin de

tenir compte des contraintes de précédences [Ric&Cot,99], c'est-à-dire, obtenir des tâches indépendantes. L'idée de base de ces modifications est qu'une tâche ne peut commencer son exécution avant ses prédécesseurs et qu'elle ne peut préempter ses successeurs.

Soit $\mu = \{T_i\ (R_{i,0},\ C_i,\ D_i,\ P_{ei})\}$ une configuration de tâches en forme normale liées par des contraintes de précédences dans le contexte $|1|R_i,\ C_i,\ D_i,\ P_{ei},\ Pfn\ |$, et soit $T_i \longrightarrow T_j$ un ordre partiel entre les tâches qui signifie que chaque instance de T_i doit être entièrement exécutée avant que celle de T_j ne débute, soit aussi la date de réveil et priorité de chaque tâche.

μ est ordonnançable par RM si et seulement si $\mu* = \{T_i\ (R_{i,0*},\ C_i,\ D_i*,\ P_i)\}$ est ordonnançable dans le contexte $|1|R_i,\ C_i,\ D_i,\ P_i|$ avec :

$$R_{i,0^*} = \max\{R_{i,0}, \max_{T_j \to T_i}\{R_{j,0^*}\}\} \qquad (II.13)$$

En traitant d'abord les tâches sans prédécesseurs et en descendant à chaque pas vers les tâches dont tous les prédécesseurs ont été traités.

$$D_i^* = D_i - (R_{i,0^*} - R_{i,0}) \qquad (II.14)$$

Si deux tâches ont la même période avec $T_i \longrightarrow T_j$ alors $P_i > P_j$.

b) EDF avec des contraintes précédence

Dans le cas d'utilisation de EDF, ***Blazewicz*** a proposé une démarche dans [Bla,76] pour la prise en compte de la précédence ($T_i \longrightarrow T_j$) entre les tâches en forme normale. Ceci se fait on modifiant les paramètres date de réveil et délai critique afin que les contraintes de précédence soient implicitement respectées. Ces modifications sont opérées comme suit :

$$R_{i,0^*} = \max\{R_{i,0}, \max_{T_j \to T_i}\{R_{j,0^*} + C_j\}\} \qquad (II.15)$$

A partir des tâches sans prédécesseurs et en descendant à chaque pas vers les tâches dont tous les prédécesseurs ont été traités.

$$D_i^* = \min\{D_i, \min_{T_i \to T_j}\{D_j^* - C_j\}\} \qquad (II.16)$$

A partir des tâches sans successeurs et en remontant à chaque pas vers les tâches dont tous les successeurs ont été traités.

II.6.1.1.2.2 Ordonnancement des tâches périodiques sous contraintes de ressources

Il est bien rare dans une application temps réel de pouvoir se passer de l'utilisation des ressources critiques (ou ressources partagées), qui peuvent être *matérielles* (processeur, mémoire, réseau, capteurs ou actionneurs), ou *logiques* (sémaphores ou files de messages). Néanmoins, une ressource critique ne peut être utilisée simultanément par plusieurs tâches et que la tâche qui l'utilise doit pouvoir aller au bout de son utilisation même si elle est moins prioritaire. Cependant la mise en concurrence de plusieurs tâches pour l'accès à une ressource engendre des retards d'exécution. De plus, le partage de ressources peut engendrer des problèmes d'*inversion de priorité* ou d'*interblocage* qui se produits comme suit :

- **L'inversion de priorité**

Ce problème se produit lorsqu'une tâche prioritaire demande une ressource détenue par une tâche de moindre priorité (voir *chapitre III*).

- **L'interblocage**

Ce problème a lieu lorsque toutes les tâches sont bloquées en attente de ressources détenues par d'autres tâches, elles-aussi bloquées, ce qui engendre une situation d'attente circulaire. Il faut au minimum deux ressources pour donner naissance à un interblocage.
Par exemple, prenons le cas où une tâche T_1 détient la ressource R_0 et demande la ressource R_1 pour continuer son exécution. Si R_1 est détenue par une autre tâche T_0 et que celle-ci demande la ressource R_0 pour se terminer, alors les deux tâches ne peuvent plus être exécutées et le système est définitivement bloqué.

La figure II.3 illustre la situation d'interblocage de deux tâches T_0 et T_1 sur deux ressources critiques R_0 et R_1:

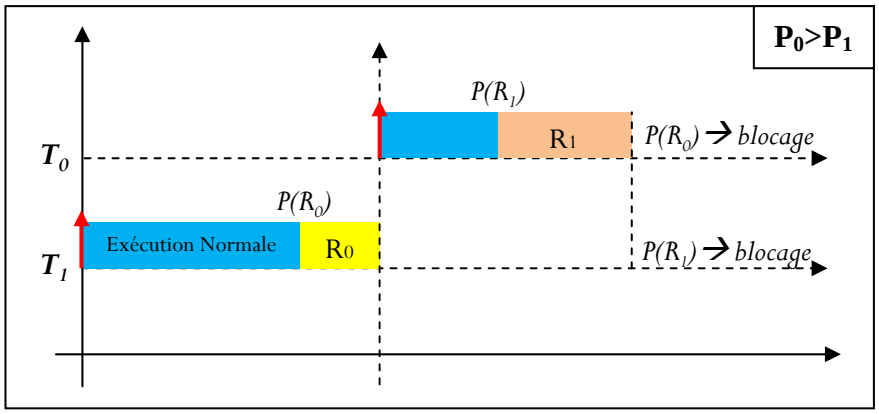

Figure II.3 : Représentation du phénomène d'Interblocage

Afin de pallier à ces deux problèmes, des protocoles de gestion des ressources ont été mis au point pour fonctionner avec les algorithmes d'ordonnancement vus précédemment. On distingue principalement le *PIP* [Sha,Raj&Leh,90] [God,90], le *PCP* [Sha,Raj&Leh,90] [God,90] [Che&Lin,90] et le *SRP* [Bak,91] [Ram,97] [God,90].

A- Le protocole à priorité héritée (PIP)

☙ ***Principe***

Lorsqu'une tâche de faible priorité T_{k+1} bloque, pour l'utilisation d'une ressource, des tâches de plus forte priorité $T_1 \dots T_k$, alors, le travail de la tâche T_{k+1} s'exécute avec une priorité égale à $max\{P_1, P_2, \dots, P_k\}$. Cette technique limite la durée des inversions de priorité mais n'interdit pas les *interblocages*.

☙ ***Test d'ordonnançabilité***

En considérant l'algorithme RM, la prise en compte du partage de ressources entraîne des modifications par rapport au test d'ordonnançabilité de cet algorithme et on ne peut plus trouver de condition nécessaire et suffisante. Ce test peut être étendu de deux façons:

(1) La première méthode consiste à vérifier que chaque tâche peut respecter son échéance même si elle est bloquée par des tâches de priorité plus faible. Dans ce cas, l'ordonnancement d'un ensemble de n tâches périodiques en utilisant PIP, est faisable si :

$$\forall\, i, 1 \le i \le n \;;\; \sum_{j=1}^{i} {C_j}/{P_{ej}} + {Bi}/{P_{ei}} \le i * \left(2^{\frac{1}{i}} - 1\right) \qquad (II.17)$$

Où B_i est la durée maximale de blocage de la tâche T_i par les tâches de priorité inférieure. Le calcul de *Bi* est présenté dans le chapitre III.

(2) La seconde méthode est plus simple à mettre en œuvre, mais plus restrictive. Dans ce cas, l'ordonnancement d'un ensemble de n tâches périodiques en utilisant PIP, est faisable si:

$$\sum_{i=1}^{n} {C_i}/{P_{ei}} + \max_{1\le i\le n}\{{B_i}/{P_{ei}}\} \le n * \left(2^{\frac{1}{n}} - 1\right) \qquad \text{(II.18)}$$

B- Le protocole à priorité plafond (PCP)

☙ ***Principe***

Dans cette technique, une tâche, pour entrer en section critique, doit avoir une priorité strictement supérieure aux "*valeurs plafonds*" des sémaphores verrouillés par les autres tâches. Une "*valeur plafond*" est la "*valeur maximale*" parmi les priorités des tâches qui pourraient utiliser la ressource.

En conséquence, une tâche T_i peut préempter une tâche T_j en section critique si et seulement si sa priorité est supérieure aux "valeurs plafonds" (ce qui veut dire que cette tâche T_i accède à des ressources auxquelles la tâche T_j n'accède pas et que sa priorité est supérieure aux valeurs plafonds des sémaphores verrouillés par la tâche T_j).

☙ ***Test d'ordonnançabilité***

- En considérant l'algorithme RM, les tests présentés pour la technique à priorité héritée peuvent être considérés;
- En considérant l'algorithme EDF, une condition suffisante d'ordonnançabilité est la suivante [Bak,91]:

$$\underset{1\le k\le n}{\forall k}\left(\sum_{i=1}^{n}\frac{C_i}{P_{ei}}\right) + \frac{B_k}{P_{ek}} \le 1 \qquad \text{(II.19)}$$

C- Le protocole d'allocation de la pile (SRP)

☙ ***Principe***

Le principe de ce protocole repose sur une diminution du nombre de préemptions dues au blocage pour le partage de ressources. Pour ce faire, une tâche n'est pas autorisée à démarrer tant que toutes les ressources qui lui sont nécessaires ne sont pas disponibles, ce qui évite les interblocages. Une fois la tâche est démarrée, ses allocations de ressources sont forcément effectuées [Akl,09].

☙ ***Test d'ordonnançabilité***

Baker a dégagé dans [Bak,91] une condition suffisante d'ordonnancement dans le contexte $|1|R_i, C_i, D_i, P_{ei}, res|$ pour EDF avec SRP :

$$\forall i = 1, \dots, n; \sum_{j=1}^{i} \frac{C_j}{D_j} + \frac{B_i}{D_i} \leq 1 \qquad \text{(II. 20)}$$

II.6.1.1.2.3 Ordonnancement des tâches sous contraintes de ressources et de précédences

Afin de traiter le problème d'ordonnancement des tâches périodiques sous contraintes de ressources et de précédences, Spuri et al [Spu&But,94] ont amélioré le protocole SRP avec une démarche qui est résumée en les étapes suivantes :

- Avant que le système entre en fonctionnement, un changement d'échéance relative de chaque tâche s'effectue comme suit :

$$d_j \leftarrow D_i; \ \forall T_i \in P_{ri} \qquad \text{(II. 21)}$$

Où : *Di* est l'échéance relative du processus *Pri* auquel appartiennent les tâches T_j.

- Ensuite cette échéance est modifiée de telle sorte que les contraintes de précédences seront respectées. Parce que si une tâche T_k doit être exécutée après T_j, la tâche T_k aura une échéance plus longue que celle de T_j, et donc une priorité moins que celle de T_j.

$$d_i \leftarrow \min\{(d_j) \cup (d_k - C_j : T_j \rightarrow T_k)\} \qquad \text{(II. 22)}$$

On commence par les tâches qui n'ont pas de successeurs pour modifier leurs échéances.

- Lors de l'exécution, si la requête du processus *Pri* arrive à l'instant t, on effectue le changement de la date de réveil et de l'échéance de la tâche T_j comme suit :

$$R'_j \leftarrow t \;\; ; \;\; d'_j \leftarrow t + d_j \;\; \forall T_j \in P_{ri} \qquad \text{(II. 23)}$$

II.6.1.1.3 Avantages

- ✓ De complexité temporelle faible (le plus souvent linéaire en fonction du nombre de tâches) [Akl, 09],
- ✓ Permet l'arrivée imprévisible de tâches et autorise la création progressive de la séquence d'ordonnancement [Akl, 09].

II.6.1.1.4 Inconvénients

- ✗ L'inconvénient majeur est le surcoût imposé par le calcul des priorités pour l'affectation des tâches aux processeurs par l'ordonnanceur. Cela implique une limitation de complexité dans le choix des algorithmes en ligne [Akl, 09].
- ✗ L'allocation de priorités plus complexe nécessite des services ou routines supplémentaires (comme la gestion des files de tâches actives par exemple) ce qui augmente le temps d'exécution effectif de l'ordonnanceur [Akl, 09].

II.6.1.2 Ordonnancement hors ligne

La technique d'ordonnancement hors ligne a été motivée par l'absence d'algorithmes d'ordonnancement en ligne polynômiaux optimaux dans un contexte plus général, tel que la présence de ressources partagées et/ou de précédences [Gro,99].

Avec l'ordonnancement temps réel hors ligne, la séquence des tâches à exécuter est calculée en hors ligne, et le temps de calcul de cette séquence par un algorithme d'ordonnancement n'est pas un critère capital. Cette approche, nécessite la connaissance à priori de toutes les caractéristiques des tâches du système [Akl,09]. Les approches hors ligne étudiées dans la littérature, sont de complexité au moins exponentielle lorsqu'elles sont exactes, car il s'agit de produire une séquence d'ordonnancement de longueur exponentielle lorsque les tâches considérées sont périodiques.

Ces approches peuvent être classées en deux grandes catégories : *les algorithmes exacts* et *les algorithmes approchées*.

II.6.1.2.1 Les algorithmes d'ordonnancements

A- Algorithmes exacts

Ces algorithmes visent essentiellement à énumérer, souvent de manière implicite, l'ensemble des solutions de l'espace de recherche. Pour améliorer l'énumération des solutions, ces méthodes utilisent des techniques pour détecter le plus tôt possible les échecs (calcul des bornes) et d'heuristiques spécifiques pour orienter les différents choix. Parmi ces techniques d'énumération, on trouve la technique de *séparation et évaluation* (*branch and bound*), ou celle basée sur la programmation linéaire en nombres entiers.

Dans la suite on va citer quelques algorithmes existants pour l'ordonnancement du système temps réel se basent sur la théorie de la recherche dans un graphe :

-DFS (depth first search): Recherche en profondeur d'abord.

Cette technique consiste à parcourir l'arbre de recherche en profondeur d'abord. Soit le graphe de recherche suivant (*figure II.4*) :

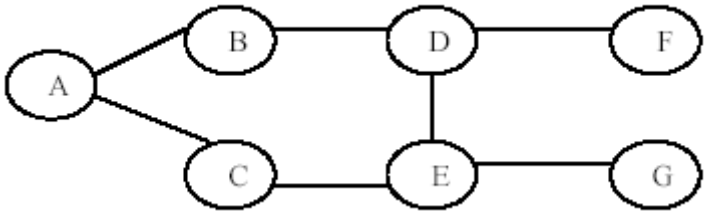

Figure II.4 : Graphe de précédence [Bad,08]

L'ordre de recherche dans ce graphe avec la méthode DFS serait: **ABDFEGC**.

-BFS(Breadth Seaerch first):recherche en largeur d'abord

Cette technique consiste à parcourir l'arbre en largeur. C'est à dire visiter les nœuds de même niveau avant de passer aux successeurs. Avec une telle stratégie l'ordre du parcours du graphe précédent serait : **ABCDEFG**.

-BB(Branch-and-Bound)

Cette technique est plus "intelligente" et elle est la plus utilisée. Elle consiste à éliminer certaines branches de l'arbre lorsqu'elles ne mènent pas à la solution en utilisant des informations heuristiques concernant l'amélioration probable des fils d'un nœud donné.

-Algorithme de xu&parnas

Contrairement à l'algorithme présenté dans [XU&Par,90] dans lequel les tâches ne sont définies que par un seul segment global, et qui ne peut pas être préempté ou chevauché par une autre tâche, pour augmenter les chances de trouver un ordonnancement valide ses auteurs ont décrit un autre algorithme dans [XU&Par,92] qui recherche des séquences d'ordonnancement valides, dans le cas réaliste :

$|1|R_i=0, C_i, D_i, prmpt, prec, res|$ où les ressources peuvent être partagées entres les tâches, avec des contraintes de précédence et de préemption. Pour cet algorithme, chaque tâche est constituée d'un ensemble de sections critiques, et chaque section critique est constituée elle-même d'un ensemble de segments, les sections critiques peuvent être en chevauchement, c.a.d qu'il peut y exister des segments en commun.

B- Algorithmes approchés

Ceux sont des algorithmes sous-optimaux, tels que les méthodes basées sur des heuristiques spécifiques (RM, DM, EDF, LLF,…) ou bien des méta-heuristiques telles que les algorithmes génétiques, le recuit simulé, ou les colonies de fourmis. Ces approches sont pour la plupart basées sur des systèmes de tâches non périodiques de date de réveil non nulle [Gro,99]. On cite parmi les méta-heuristiques appliquées au problème d'ordonnancement des systèmes de tâches temps réel :

-Les algorithmes génétiques

Les algorithmes génétiques sont des heuristiques utilisées pour résoudre les problèmes NP-Complets. Elles se basent sur la génération de nouvelles solutions (individus) à partir d'un ensemble de solutions initiales (population). Cette génération est inspirée des concepts biologiques tels que les chromosomes, croisement et mutation.

Pour résoudre un problème en utilisant les AGs, il suffit de le modéliser convenablement. Cette technique a été implémentée dans le TTA (Time-Triggered Architecture); un système temps réel distribué [Nos,98].

-Recuit simulé

Le recuit simulé est une méthode d'optimisation qui part d'une solution du problème et qui tente de l'améliorer en explorant l'espace de recherche par voisinages. L'algorithme considère les solutions du problème comme des états d'énergie du système et le but est de minimiser cette énergie. A chaque itération, on fait diminuer la ''température'' qui représente le paramètre clé de l'algorithme. L'évaluation de la nouvelle solution obtenue est sujette à une probabilité d'acceptation, de manière à éviter de rester piégé dans un minimum local.

Le recuit simulé a été utilisé dans [Tin,Bur,Wel,92] pour résoudre le problème d'ordonnancement dans un contexte distribué sans préemption. La particularité de ce travail est qu'il propose une nouvelle approche d'accès au bus de communication.

II.6.1.2.2 Avantages [Bad, 08]

- ✓ L'évaluation des tâches est effectué avec ou sans préemption,
- ✓ Technique puissante et fiable.

II.6.1.2.3 Inconvénients [Akl, 09]

- ✗ Le calcul préalable nécessite de connaître parfaitement toutes les tâches présentes dans l'application ainsi que leurs dates d'occurrences, ce qui fige à jamais l'application ;
- ✗ Une autre séquence d'ordonnancement est calculée lors qu'une modification de l'application est opérée (ajout d'une tâche ou la modification de l'une d'entre elles entraînant une modification de ses paramètres temporels),
- ✗ L'approche est inadaptée dans le cas de systèmes de tâches dont les dates d'arrivées ne sont pas forcément connues (tâches non périodiques).

II.6.2 Validation

La validation d'ordonnancement d'une configuration de tâches par un algorithme en ligne s'effectue toujours hors ligne. Et ceci soit à l'aide de modèles analytiques (conditions suffisante et/ou nécessaire) qui se basent sur les critères temporels des tâches, soit par la simulation de la configuration pendant une durée suffisante, appelée période d'étude [Akl,09]. Dans le cas d'un algorithme hors-ligne, la validation consiste en la génération de la séquence d'ordonnancement valide par un algorithme de recherche [Gro,99].

Si un test d'ordonnancement s'appuyant sur une condition suffisante produit un résultat positif, le système de tâches est définitivement ordonnançable. Par contre, si le résultat est négatif, le système peut être ordonnançable ou non [Pai,06]. De la même manière, si un test d'ordonnancement s'appuyant sur une condition nécessaire d'ordonnancement produit un résultat positif, l'ensemble de tâches peut ne pas être ordonnançable, alors que s'il est négatif, le système de tâches est définitivement non ordonnançable [Pai,06]. Si seule une condition suffisante est disponible dans le contexte considéré pour un algorithme donné, et que celle-ci n'est pas vérifiée pour la configuration de tâches, seule une simulation permet de conclure ou non l'ordonnancement de cette configuration.

II.7 Conclusion

Dans ce chapitre nous avons introduit le problème d'ordonnancement des tâches temps réel. D'abord, nous avons défini qu'est ce que une technique d'ordonnancement temps réel et que signifier le contexte d'ordonnancement, en expliquant les différentes contraintes qui doivent être gérées par un algorithme d'ordonnancement. Ensuite, nous avons présenté les différents politiques d'ordonnancement, en ligne, à savoir les méthodes à priorité fixe ou variable ainsi que leurs avantages et inconvénients. Après, nous avons présenté les travaux élaborés pour l'ordonnancement hors ligne ainsi que les avantages et les inconvénients de cette approche. Le chapitre suivant sera consacré en détail à un état de l'art sur les protocoles disponibles pour contrer le phénomène d'inversion de priorités qui est l'objet de notre mémoire.

Chapitre III :
État de l'art sur le phénomène d'inversion de priorité

III.1 Introduction

Le partage de ressources dans un système temps réel peut être à l'origine du phénomène dit d'*inversion de priorité*. Rappelons qu'une *tâche* i est une séquence d'instructions 'programme' que l'on caractérise par une capacité C_i, une priorité P_i et une période Pe_i, tel qu'un ensemble de tâche dont l'exécution doit être synchronisée. La priorité d'une tâche peut être fixée par un des algorithmes d'ordonnancement statiques (ex : Rate Monotonic 'RM' et Deadline Monotonic 'DM') pour des priorités fixes, ou variable par certains algorithmes d'ordonnancement dynamique (ex : Earliest Deadline First 'EDF') pour des priorités dynamiques. Une *ressource* est une structure logicielle pouvant être utilisée par une tâche pour avancer dans son exécution (ex: ensemble de variables, mémoire partagée, fichier etc.). Elle est dite *privée* lorsqu'elle est dédiée à une tâche en particulier, *partagée* si elle est utilisée par plusieurs tâches. Dans ce dernier cas, des mutex (ex : sémaphores) assurent l'accès exclusif [Ela,02].

On définit une *section critique* 'sc' comme étant une partie de code exécutée en exclusion mutuelle. Une tâche est dite bloquée si elle attend une ressource exclusive. Lorsqu'elle la prend, elle entre en section critique. Une ressource exclusive est dite libre si aucune tâche n'est dans une section critique utilisant cette ressource.

Dans ce chapitre nous avons présenté le problème d'inversion de priorité ainsi que les différents protocoles qui sont conduit à empêcher ce problème. Ensuite, nous abordons une étude comparatif selon [Ela,02] de ces protocoles d'inversion de priorité.

III.2 La problématique

Lorsque les tâches partagent des ressources à accès exclusif (tâches dépendantes) mais que l'ordonnancement adopté est à priorité fixe (l'allocation du CPU est basée sur des priorités) alors le problème d'inversion de priorité est probablement produit.

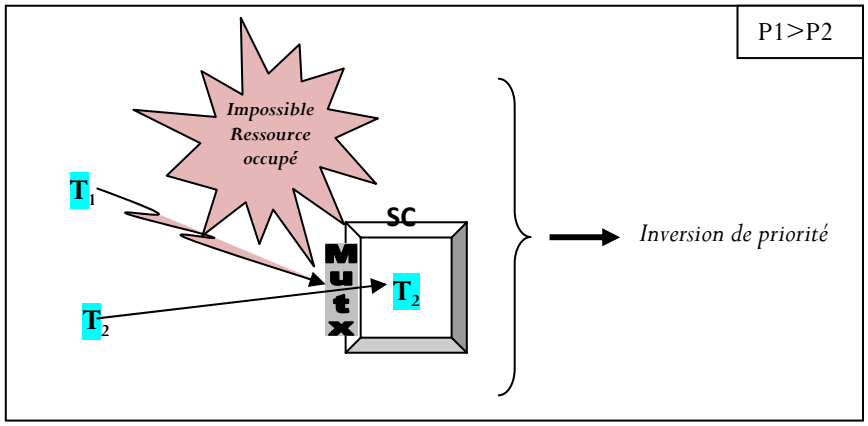

Figure III.1 : Représentation de l'inversion de priorité [Dou&Gho,10]

Lorsqu'une tâche T_1 demande une ressource déjà allouée à une autre tâche T_2, T_1 se met en attente de la ressource même si elle est la plus prioritaire.

La non gestion de l'inversion de priorité peut avoir des effets désastreux. En effet, comme la non gestion de l'inversion de priorité implique qu'une tâche de haute priorité peut ne pas s'exécuter, il est possible qu'une réaction à des situations d'*urgence* ne soit pas prise (par exemple, un ordre d'arrêt d'urgence d'une centrale nucléaire qui serait bloqué par un autre ordre de moindre priorité).

III.3 Définitions

III.3.1 Priorité des tâches

La *priorité* est une caractéristique d'une tâche informatique utilisée par l'ordonnanceur pour déterminer son accès à une ressource [Cot&Bab,94]. Ainsi, chaque tâche se voit attribuer une priorité qui est fonction de son caractère critique. Plus une tâche est importante, plus sa priorité est élevée. Il existe deux types de priorité de tâches :

- ***Priorité statique*** : la priorité de chaque tâche n'évolue pas durant l'exécution. Chaque tâche reçoit une priorité fixe lors de sa création. Toutes les tâches et leurs contraintes temporelles sont connues à la compilation.
- ***Priorité dynamique*** : La priorité de chaque tâche peut évoluer durant l'exécution. Cette caractéristique des RTOS permet d'éviter le problème d'inversion de priorité.

III.3.2 Inversion de priorité

L'*inversion des priorités* est une situation indésirable due au partage de ressource où l'accès d'une tâche à une ressource partagée est empêché par une tâche moins prioritaire, qui possède la ressource [Bou,09]. Ceci se produit lorsque l'ordonnancement des tâches pour l'accès au processeur est explicitement distinct de l'ordonnancement pour l'accès aux ressources. On ne peut alors plus prévoir le temps de blocage de la tâche de plus haute priorité, et les conséquences peuvent être graves.

III.4 Exemple illustratif

Considérons une ressource R_0 partagée entre tâches T_0, T_1 et T_2 telles que $P_0 > P_1 > P_2$.

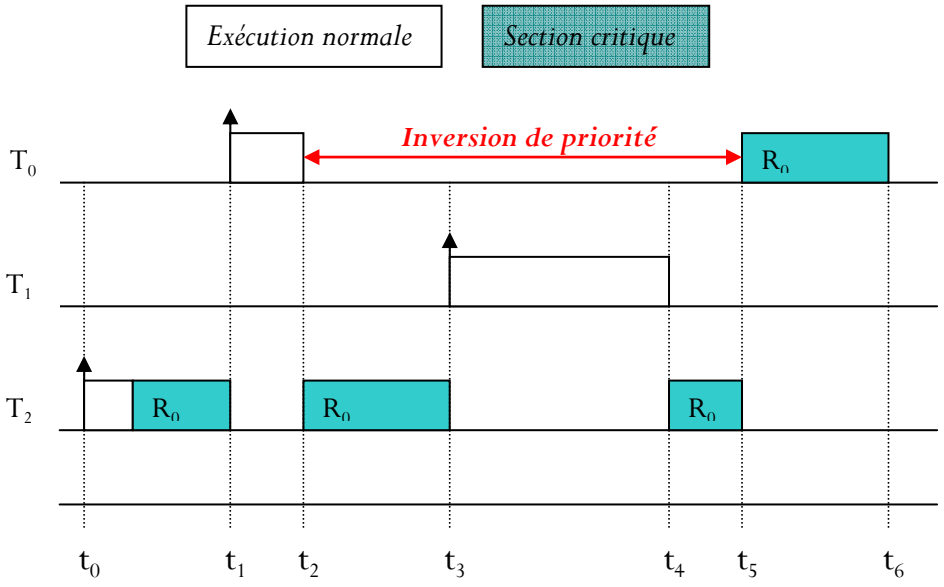

Figure III.2 : Illustration du phénomène d'inversion de priorité [Ela,02]

- A l'instant t_0 : T_2 est activée, peut prendre R_0 et entre en section critique.
- A l'instant t_1 : T_0 préempte T_2 et commence son exécution.

- A l'instant t_2 : T_0 ne peut prendre R_0 car la ressource n'est pas libre. T_2 reprend son exécution.
- A l'instant t_3 : T_1 est activée et préempte T_2.
- A l'instant t_4 : T_1 ne peut prendre R_0 car la ressource n'est pas libre. T_2 reprend son exécution.
- A l'instant t_5 : T_2 libère la ressource R_0, T_0 préempte T_2 et entre en section critique en prenant la ressource. La durée du blocage dû à l'inversion de priorité est égale à t_5-t_2 : elle peut être beaucoup plus importante que la durée de la section critique de T_2 car cette tâche peut être préemptée par d'autres tâches de priorité supérieure.
- A l'instant t_6 : T_0 libère R_0.

III.5 Remèdes au problème

Dans un système avec des ressources partagées, il est impossible d'éliminer toutes les inversions de priorités, mais il est possible d'en limiter les temps d'attente de façon à minimiser et prévoir les temps de blocages. Pour cela, il existe plusieurs approches [Ela,02] :

☞ **Triviale :** rendre une section critique non interruptible [Bou,09].

Priorité(T_0)> Priorité(T_1)>Priorité(T_2)

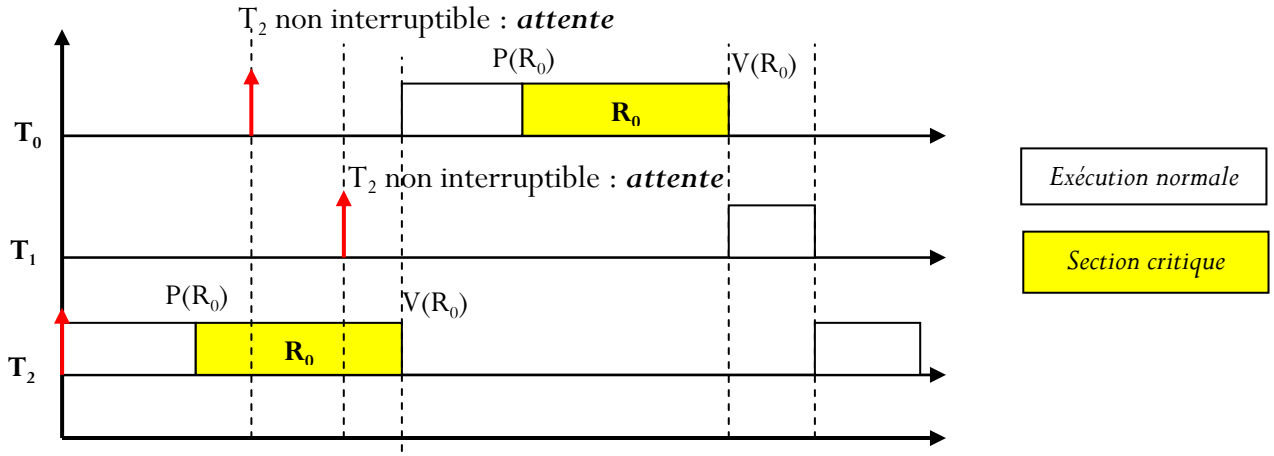

Figure III.3 : Ordonnancement des tâches avec section critique non interruptible [Bou,09]

- Une tâche en section critique ne peut être préemptée par une autre tâche ;
- T1 retardée au plus de la durée de la section critique.

MAIS

- Difficile dans le cas d'une section critique longue ;
- Blocage des processus de haute priorité n'utilisant pas la section critique.

☞ N'autoriser l'accès à des sections critiques qu'à des tâches de même priorité.

☞ On peut empêcher la préemption lors de l'exécution de la section critique d'une tâche mais l'on crée ainsi des blocages inutiles, cette technique n'est donc envisageable que dans le cas où les sections critiques sont de courte durée d'exécution [Ela,02].

☞ On utilise plutôt des protocoles d'accès aux ressources où les conflits sont résolus en réalisant des ajustements sur les priorités des tâches [Ela,02].

On en distingue trois mécanismes principaux (disponibles dans certains RTOS) : *l'héritage des priorités* avec le protocole PIP (Priority Inheritance Protocol), *le plafond des priorités* avec les protocoles : OCPP (Original Ceiling Priority Protocol) et ICPP (Immediate Ceiling Priority Protocol), et l'approche appelée "*stack resource*" avec le protocole SRP (Stack Resource Policy). Après avoir étudié chacun d'eux, nous comparerons leurs différentes caractéristiques pour en déterminer les avantages et les inconvénients [Ela,02].

III.5.1 L'héritage des priorités

Introduit par Sha, Rajkumar et Lehoczky [Bil,87], le protocole PIP empêche des tâches de priorité moyenne de préempter la tâche responsable du blocage d'une tâche de haute priorité et de prolonger ainsi son temps de blocage, la situation remarquée dans l'exemple précédent est donc évitée. Le support d'ordonnancement est RM. Une version modifiée du protocole permet d'utiliser EDF pour avoir des priorités dynamiques (PIP for dynamic priorities) [Cha,01].

III.5.1.1 PIP (**P**riority **I**nheritance **P**rotocol**)**

III.5.1.1.1 Principe

L'idée de base de PIP est d'affecter temporairement la tâche responsable du blocage avec la priorité de la tâche la plus prioritaire [Ela,02]. Formellement, lorsqu'une tâche de faible priorité $\boldsymbol{T_{k+1}}$ bloque, pour l'utilisation d'une ressource, des tâches de plus forte priorité $\boldsymbol{T_1 \ldots T_k}$, alors, le travail de la tâche $\boldsymbol{T_{k+1}}$ s'exécute avec une priorité égale à $\boldsymbol{max\ \{P_1, P_2, \ldots, P_k\}}$. Cette technique limite la durée des inversions de priorité mais n'interdit pas les *interblocages*.

III.5.1.1.2 Règles d'ordonnancements

Chaque tâche T_i possède une priorité fixe (ex : déterminée par RM) ainsi qu'une priorité actuelle $\pi_i(t)$. Lorsqu'une tâche T_i demande une ressource R_k, on a l'une des situations suivantes [God,90]:

> ***Si*** *R_k est libre* ***alors*** *T_i prend la ressource et entre en section critique ;*
> ***Si*** *R_k n'est pas libre* ***alors*** *T_i est bloqué. De plus, la tâche T_j qui bloque T_i hérite de sa priorité actuelle. Cette tâche continue à s'exécuter avec sa nouvelle priorité jusqu'à ce qu'elle libère la ressource.* La tâche *T_j reprend alors sa priorité initiale (c'est-à-dire celle qu'elle avait avant de prendre la ressource).*

L'héritage des priorités étant transitif, il se peut qu'une tâche hérite plusieurs fois si elle bloque plusieurs tâches.

III.5.1.1.3 Exemple

Reprenons l'exemple précédent. L'ordonnancement devient le suivant [Ela,02]:

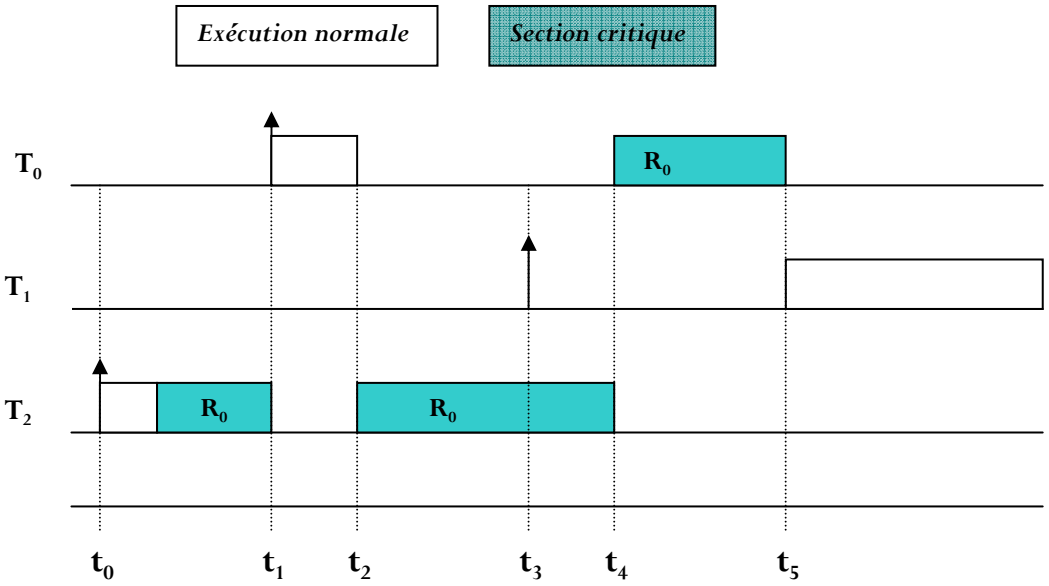

Figure III.4 : Ordonnancement des tâches avec PIP [Ela,02]

- A l'instant t_0 : T_2 est activée, peut prendre R_0 et entre en section critique.
- A l'instant t_1 : T_0 préempte T_2 et commence son exécution.
- A l'instant t_2 : T_0 ne peut prendre R_0 car la ressource n'est pas libre. T_2 reprend son exécution mais avec la priorité de T_0.
- A l'instant t_3 : T_1 est activée mais sa priorité est désormais inférieure à celle de T_2 qui continue son exécution. On voit ici que PIP introduit un nouveau type de blocage: *le blocage d'héritage de priorité* (inheritance blocking).
- A l'instant t_4 : T_2 libère la ressource R_0, reprend sa priorité initiale et est préemptée par T_0. T_0 peut prendre la ressource libre et entre en section critique.
- A l'instant t_5 : T_0 termine son exécution en libérant la ressource R_0, T_1 peut commencer son exécution.

III.5.1.1.4 Caractéristiques

- ☞ Une tâche peut être bloquée plusieurs fois.
- ☞ Estimation pessimiste du temps de blocage.
- ☞ On distingue deux types de blocage: *le blocage direct* (direct blocking) et *le blocage d'héritage de priorité* (priority inheritance blocking) [Ela,02]:

- *Le blocage direct* survient lorsqu'une tâche de haute priorité demande une ressource utilisée par une tâche de plus basse priorité.
- *Le blocage d'héritage de priorité* survient lorsqu'une tâche de priorité moyenne est bloquée par une tâche de plus basse priorité mais qui a hérité d'une priorité plus forte.

Comme on a cité auparavant le protocole PIP n'empêche pas les situations d'interblocage (deadlock), qui peuvent survenir lorsqu'il y a plusieurs ressources partagées [Ela,02] :

Soit l'exemple suivant :

Considérons une tâche T_0 attendant une ressource R_0 puis une autre R_1 avant de libérer R_0. Une autre tâche T_1 effectue l'allocation de ressource inverse, avec $P_0 > P_1$.

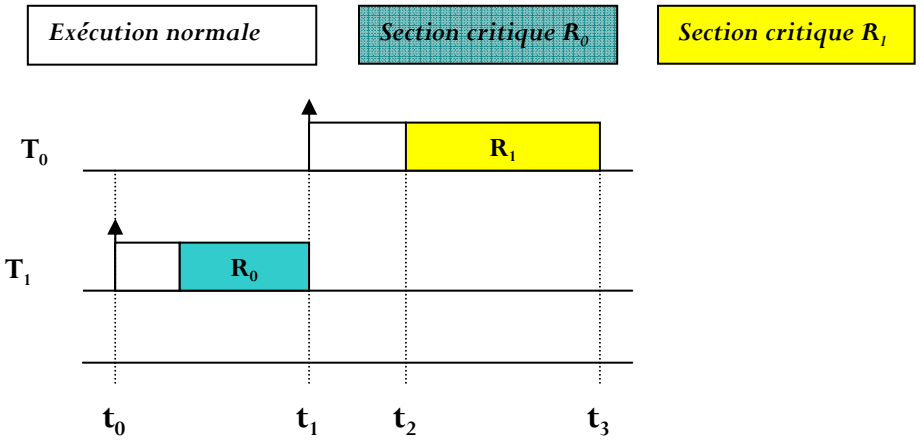

Figure III.5 : Situation d'inter-blocage avec PIP [Ela,02]

- A l'instant t_0 : T_1 est activée, peut prendre R_0 et entre en section critique.
- A l'instant t_1 : T_0 est activée et préempte T_1.
- A l'instant t_2 : T_0 peut prendre R_1 car cette ressource est libre.
- A l'instant t_3 : T_0 ne peut pas prendre R_0 car cette ressource est utilisée par T_1. De même T_1 ne peut pas prendre R_1 et on a une situation d'interblocage.

De plus, le protocole PIP peut entraîner le phénomène de *chaîne de blocage*, qui survient lorsque la tâche la plus prioritaire est bloquée une fois par toutes les autres

tâches. Considérons trois tâches T_0, T_1, T_2 avec $P_0 > P_1 > P_2$, ainsi que 2 ressources R_0 et R_1. T_0 utilise R_0 puis R_1, T_1 utilise R_1 et T_2 utilise R_0. L'ordonnancement suivant illustre le phénomène de chaîne de blocage [Ela,02] :

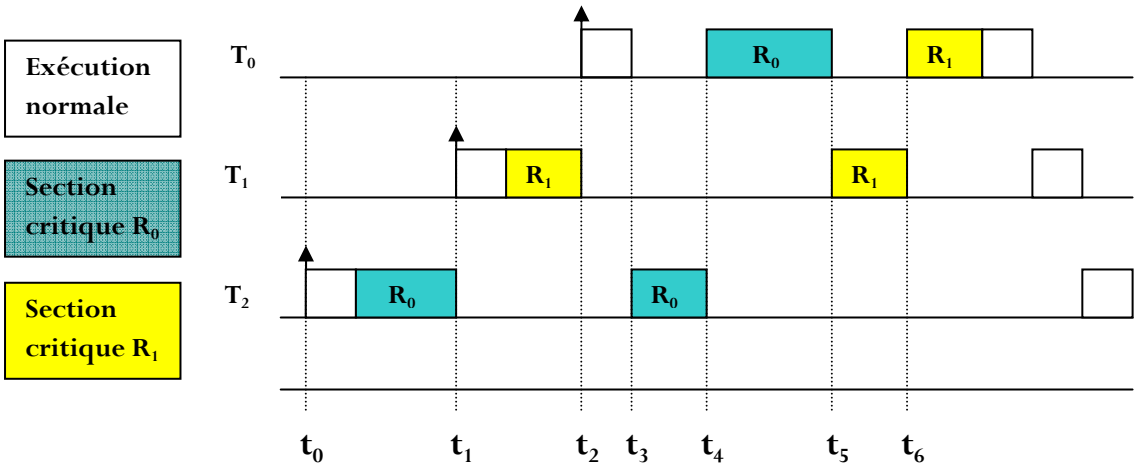

Figure III.6 : Situation de chaine de blocage avec PIP [Ela,02]

- A l'instant t_0 : T_2 est activée, peut prendre R_0 et entre en section critique.
- A l'instant t_1 : T_1 est activée et préempte T_2 car $P_1 > P_2$, ensuite T_1 peut prendre R_1 et entre en section critique.
- A l'instant t_2 : T_0 est activée et préempte T_1 car $P_0 > P_1$.
- A l'instant t_3 : T_0 ne peut pas prendre R_0 car cette ressource est utilisée par T_2 qui reprend son exécution avec la priorité de T_0.
- A l'instant t_4 : T_2 libère R_0 et reprend sa priorité initiale, T_0 préempte T_2 et entre en section critique en prenant la ressource R_0.
- A l'instant t_5 : T_0 libère R_0 mais ne peut prendre R_1 car cette ressource est utilisée par T_1 ; T_1 reprend son exécution avec la priorité de T_0.
- A l'instant t_6 : T_1 libère R_1 et reprend sa priorité initiale, T_0 préempte T_1 et entre en section critique en prenant la ressource R_1.

La tâche T_0, de plus forte priorité, a donc été bloquée une fois par toutes les autres tâches: au temps t_3 elle est bloquée par T_2 et au temps t_5 elle est bloquée par T_1.

Pour déterminer le temps de blocage B_i, on considère les deux résultats suivants [Sha,Raj&Leh,90] :

1) s'il y a n tâches de basse priorité pouvant bloquer une tâche T_i alors T_i peut être bloquée pour une durée maximale de n sections critiques;
2) s'il y a m ressources distinctes qui peuvent bloquer une tâche T_i alors T_i peut être bloquée pour une durée maximale de m sections critiques.

Le temps de blocage est alors égal à la durée de min {n, m} sections critiques.

- Pour chaque tâche avec une priorité inférieure à P_i on cherche l'ensemble des sections critiques pouvant bloquer T_i, on fait la somme des durées maximales de la plus longue section critique de chacun de ces ensembles,
- Pour chaque ressource, on cherche l'ensemble des sections critiques S_k qui l'utilisent et peuvent bloquer T_i, on fait la somme des durées maximales de la plus longue section critique de chacun de ces ensembles.

Le temps de blocage est égal au minimum des deux durées trouvées.

De façon plus formelle, on ordonne les indices des tâches selon leur priorité, de la plus forte à la plus faible. En notant $\pi(R_k)$ la priorité maximum des tâches accédant à R_k et $D_{j,k}$ la durée de la section critique associée, on a: [Ela,02]

$$\boldsymbol{B_i^T = \sum_{j=i+1}^{n} \max_{k} \{D_{j,k} : \pi(R_k) \geq P_i\}} \qquad \textbf{(III. 1)}$$

$$\boldsymbol{B_i^s = \sum_{k=1}^{m} \max_{j<i} \{D_{j,k} : \pi(R_k) \geq P_i\}} \qquad \textbf{(III. 2)}$$

$$\boldsymbol{B_i = \min\{B_i^T, B_i^s\}} \qquad \textbf{(III. 3)}$$

Une autre méthode pour calculer le temps de blocage Bi est illustré comme suit [INF,09] :

$$B_i = \sum_{j=1}^{k} usage(R_j, i) CS(R_j) \qquad \text{(III. 4)}$$

Où :

– *k* c'est le nombre de ressource.

– *usage* est une fonction 0/1 :

Si la ressource $\boldsymbol{R_j}$ est utilisée par au moins une tâche de priorité inférieure à *i*, et au moins une tâche de

priorité supérieure ou égale à *i*, $\boldsymbol{usage(R_j, i) = 1}$

Sinon 0

– $\boldsymbol{CS(R_j)}$ le temps maximum requis par une tâche de priorité inférieure pour passer à travers la section critique $\boldsymbol{R_j}$.

III.5.1.1.5 Avantages

- Limitation du temps de blocage (le rend de durée bornée).

III.5.1.1.6 Inconvénients

- Ce protocole peut mener à un Interblocage.
- N'élimine pas complètement l'inversion de priorité.
- plus de possibilité de chaînes de blocage. Beaucoup d'interactions entre tâches entraîneront une diminution du nombre de priorités effectives et un regroupement des tâches sur quelques priorités hautes.

III.5.2 Protocoles à plafond de priorité (PCP : *Priority Ceiling Protocols*)

Définis par Sha, Rajkumar et Lehoczky, OCPP et ICPP sont des améliorations de PIP permettant d'éviter les situations d'interblocage ainsi que les chaînes de blocage. Le support d'ordonnancement est RM. Une version modifiée du protocole permet d'utiliser EDF pour avoir des priorités dynamiques (DCPP: Dynamic Ceiling Priority Protocol) [Che&Lin,90].

On définit *le plafond de priorité* (priority ceiling) d'une ressource R_k comme étant la plus haute priorité des tâches accédant à R_k, et on la note $\pi(R_k)$. *Le plafond de priorité courant* du système (current priority ceiling), noté π', est égal au maximum des plafonds de priorité des ressources utilisées, ou Ω si aucune ressource n'est utilisée (Ω est une priorité plus basse que n'importe qu'elle priorité) [Ela,02].

III.5.2.1 OCPP (**O**riginal **C**eiling **P**riority **P**rotocol)

III.5.2.1.1 Principe

L'idée de base de OCPP est d'étendre le PIP avec des règles pour prendre les sémaphores, ces règles interdisant une tâche d'entrer en section critique s'il y a des sémaphores bloquées susceptibles de bloquer cette tâche. Chaque tâche a *une priorité statique* assignée au départ et *une priorité dynamique* qui est le maximum entre sa priorité statique et toute priorité dont elle hériterait en bloquant une tâche plus prioritaire [Ela,02].

Ce protocole suppose que chaque tâche possède une priorité fixe et que les ressources utilisées sont connues avant le début de l'exécution [Ela,02].

III.5.2.1.2 Règles d'ordonnancements

Chaque ressource R_k (SC) à laquelle on associe un mutex a une valeur statique, nommée priorité plafond (*ceiling*) de R_k :

$$\boldsymbol{\pi(R_k) = max_{T_i\ use(R_k)}\ P_i} \qquad \textbf{(III. 5)}$$

La priorité plafond courante à un instant t :.

$$\boldsymbol{\pi' = max_{R_k\ inUse(t)}\ \pi(R_k)} \qquad \textbf{(III. 6)}$$

Lorsqu'une tâche T_i demande une ressource R_k à un instant t, on considère les situations suivantes [God,90]:

a) R_k est utilisée par une autre tâche. Dans ce cas la requête de T_i est rejetée et T_i est bloquée.

b) R_K est libre. Deux cas :

- **Si** la priorité dynamique de T_i est strictement supérieure au plafond de priorité courant[1] π', alors T_i peut prendre R_k.

$$\boldsymbol{P_i Dyn(T_i, t) > \pi'(t)} \qquad \textbf{(III. 7)}$$

- **Si** $P_i \leq \pi'$, T_i ne peut prendre R_k que si T_i a pris la ressource R dont le plafond de priorité est égal à π'. Dans le cas contraire, la requête de T_i est rejetée et T_i se retrouve bloquée.

$$\boldsymbol{\pi(R, t) = \pi'(t)} \qquad \textbf{(III. 8)}$$

Lorsque T_i se bloque, la tâche T_j qui bloque T_i hérite de la priorité actuelle P_i de T_i. Elle est alors exécutée avec cette priorité jusqu'à ce qu'elle libère toutes les ressources dont le plafond de priorité est supérieur à P_i. Ensuite, sa priorité redevient celle qu'elle était [Ela,02].

III.5.2.1.3 Exemple

Considérons 3 tâches T_0, T_1 et T_2, avec $P_0 > P_1 > P_2$, ainsi que trois ressources partagées R_0, R_1 et R_2 telles que [Cha,01]: T_0 utilise R_0 puis R_1, T_1 utilise R_2 et T_2 utilise R_2 puis R_1.

On a donc:

- $\pi(R_0) = P_0$
- $\pi(R_1) = P_0$
- $\pi(R_2) = P_1$

[1] *Le plafond courant* c'est le plafond de tout les ressources utilisés à l'instant t (excepté ceux déjà utilisés par T_i).

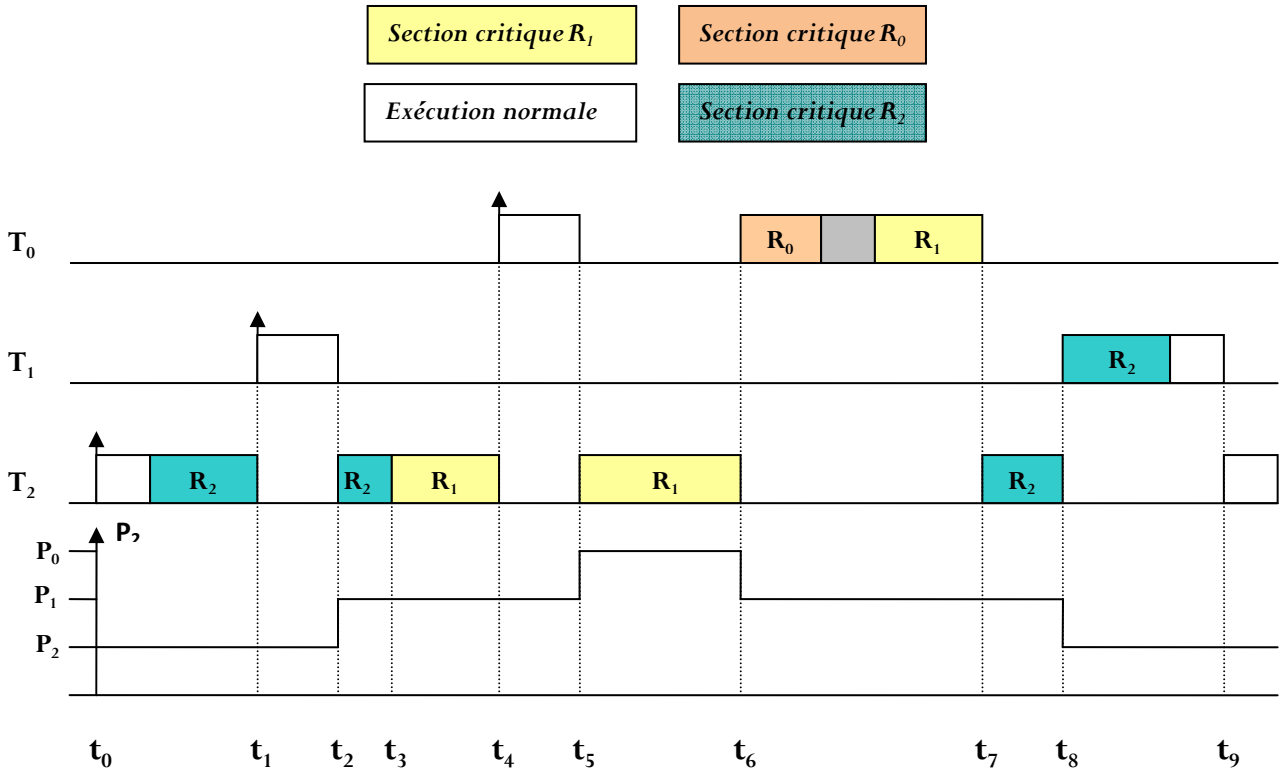

Figure III.7 : Ordonnancement des tâches avec OCPP [Ela,02]

- A l'instant t_0 : T_2 est activé, peut prendre R_2 car $P_2 > \pi' = \Omega$.
- A l'instant t_1 : T_1 préempte T_2
- A l'instant t_2 : T_1 ne peut prendre R_2 car P_1 n'est pas $> \pi' = \pi(R_2)$. T_2 hérite de la priorité de T_1 et reprend son exécution.
- A l'instant t_3 : T_2 peut prendre R_1 car aucune ressource n'est utilisée.
- A l'instant t_4 : T_0 préempte T_2.
- A l'instant t_5 : T_0 ne peut prendre R_0 car P_0 n'est pas $> \pi' = \pi(R_1)$, T_2 hérite de la priorité de T_0 et reprend son exécution.
- A l'instant t_6 : T_2 libère R_1 et reprend sa priorité $P_2 = P_1$. T_2 est alors préemptée par T_0 et T_0 prend R_0 car on a $P_0 > \pi' = \pi(R_2)$.
- A l'instant t_7 : T_0 termine son exécution, T_2 reprend la sienne et prend la ressource R_2.
- A l'instant t_8 : T_2 libère R_2 et reprend sa priorité initiale P_2: T_1 préempte T_2 et prend R_2 car on a $P_1 > \pi' = \Omega$.
- A l'instant t_9 : T_1 termine son exécution, T_2 peut reprendre et terminer la sienne.

III.5.2.1.4 Caractéristiques

On distingue désormais trois cas de blocage : comme pour PIP on a *le blocage direct* (direct blocking) et *le blocage d'héritage de priorité* (priority inheritance blocking), on a en plus les situations de *blocage de plafond de priorité* (priority ceiling blocking) comme on peut le voir au temps t_5 dans l'exemple précédent. Avec OCPP, le temps de blocage est réduit à la durée de la plus longue section critique parmi celles qui peuvent bloquer une tâche, car une tâche ne peut être bloquée que pour la durée d'une section critique. La difficulté est d'identifier l'ensemble des sections critiques susceptibles de bloquer une tâche [Ela,02].

Pour une section critique exécutée par une tâche T_j utilisant une ressource R_k, on montre qu'elle peut bloquer une tâche T_i si et seulement si $P_j < P_i$ et $\pi(R_k) \geq P_i$ (ce blocage est évident dans l'exemple précédant à l'instant t_5) [Ela,02].

Le temps de blocage est donc : B_i est égale à la durée de la plus grande section critique parmi celles des tâches de priorité $\leq P_i$ et dont la ressource relative R_i est telle que $\pi(R_i) \geq P_i$. De manière plus formelle, en notant $D_{j,k}$ la durée de la section critique de la tâche T_j utilisant R_k, on a [Ela,02]:

$$B_i = \max_{j,k}\{D_{j,k} : P_j < P_i \; et \; \pi(R_k) \geq P_i\} \qquad \text{(III. 9)}$$

III.5.2.1.5 Avantages

- Une tâche peut être bloquée une seule fois.
- Les interblocages sont évités et l'exclusion mutuelle est assurée par le protocole même.
- La propriété de transitivité de l'héritage des priorités interdit en effet d'avoir un cycle de blocages [Cha,01].
- Les chaînes de blocage sont également éliminées car une tâche ne peut être bloquée que par au plus une tâche de priorité inférieure [Sha,Raj&Leh,90].

III.5.2.2 ICPP (Immediate Ceiling Priority Protocol)

III.5.2.2.1 Principe

Le principe de ICPP est identique à celui de OCPP, sauf qu'ici *la priorité dynamique* de chaque tâche est égale au maximum entre sa priorité statique et celles des ressources qu'elle utilise (actuellement), c'est-à-dire que sa priorité augmente dès qu'elle prend une ressource [And,99].

Le ICPP est plus simple à implémenter et demande moins de changements de contexte qu'OCPP.

III.5.2.2.2 Exemple

Avec l'exemple précédent, on peut obtenir l'ordonnancement suivant [Ela,02] :

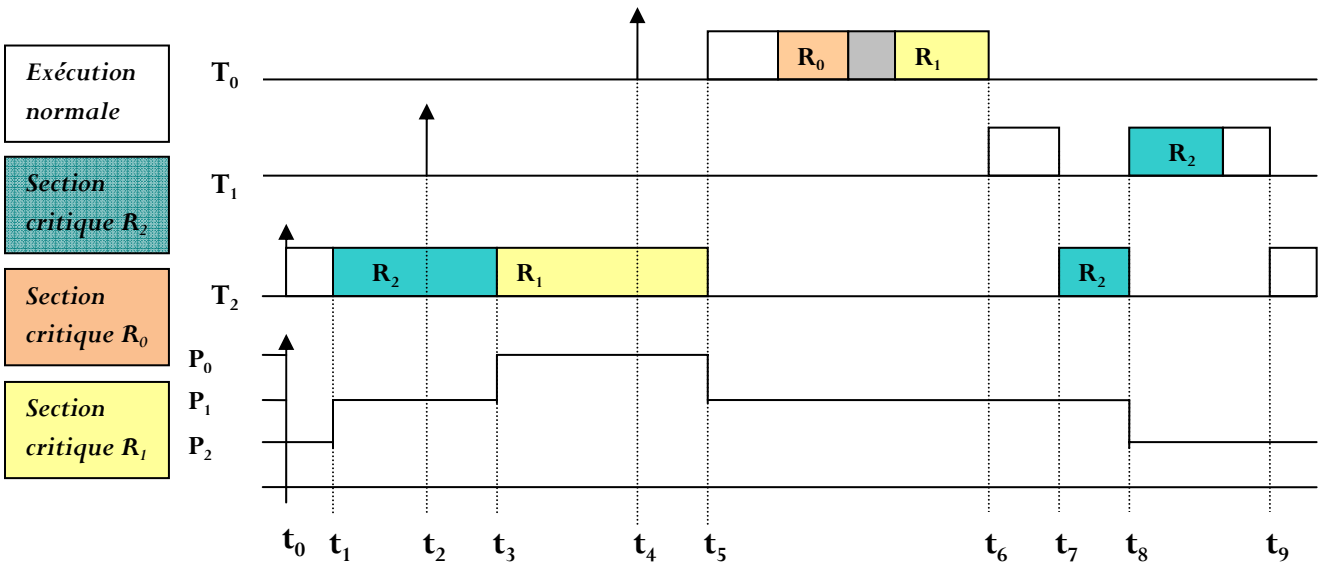

Figure III.8 : Ordonnancement des tâches avec ICPP [Ela,02]

- A l'instant t_0 : T_2 est activé et commence son exécution.
- A l'instant t_1 : T_2 peut prendre R_2 car $P_2 > \pi' = \Omega$. T_2 entre en section critique avec $P_2 = \pi(R_2) = P_1$.
- A l'instant t_2 : T_1 est activé, T_2 peut continuer à s'exécuter car $P_2 = P_1$.
- A l'instant t_3 : T_2 peut prendre R_1 car $P_2 \leq \pi'$ et P_2 a pris la ressource R_2 ($\pi(R_2) = \pi'$). P_2 continue son exécution avec $P_2 = \max \{\pi(R_1), \pi(R_2)\} = \pi(R_1) = P_0$

- A l'instant t_4 : T_0 est activé, T_2 peut continuer à s'exécuter car $P_2 = P_0$.
- A l'instant t_5 : T_2 libère R_1 et retrouve sa priorité $P_2 = P_1$. T_2 est alors préemptée par T_0 qui commence son exécution.
- A l'instant t_6 : T_0 termine son exécution, T_1 ou T_2 peuvent reprendre la leur puisqu'elles ont la même priorité, ici par exemple T_1 reprend son exécution.
- A l'instant t_7 : T_1 ne peut prendre R_2 car cette ressource est prise (par T_2), T_2 reprend donc son exécution.
- A l'instant t_8 : T_2 libère R_2 et reprend sa priorité initiale P_2: T_1 préempte T_2 et prend R_2 car on a $P_1 > \pi' = \Omega$.
- A l'instant t_9 : T_1 termine son exécution, T_2 peut reprendre et terminer la sienne.

III.5.2.2.3 Caractéristiques

Par rapport à OCPP, ce protocole modifie plus souvent les priorités puisque celles-ci changent même s'il n'y a pas de blocage. Mais il entraîne moins de commutations de contextes.

Les même cas de blocage de OCPP sont existe avec ICPP. Le temps de blocage est donc : B_i est égale à la durée de la plus grande section critique parmi celles des tâches de priorité $\leq P_i$ et dont la ressource relative R_i est telle que $\pi(R_i) \geq P_i$. De manière formelle on peut garder les formules précédentes de OCPP ou d'une autre manière :

$$\boldsymbol{B_i = \max_{j=1}^{k} usage(R_j, i) CS(R_j)} \qquad \textbf{(III.10)}$$

Où:

k c'est le nombre de ressource.

– *usage* est une fonction 0/1 :

Si la ressource $\boldsymbol{R_j}$ est utilisée par au moins une tâche de priorité inférieure à *i*, et au moins une tâche de priorité supérieure ou égale à *i*, $\boldsymbol{usage(R_j, i) = 1}$

Sinon 0

– $\boldsymbol{CS(R_j)}$ le temps maximum requis par une tâche de priorité inférieure pour passer à travers la section critique$\boldsymbol{R_j}$.

III.5.2.2.4 Avantages

- ICPP évite complètement les problèmes d'interblocage (*deadlock).*
- OCPP et ICPP ont une durée de blocage maximale plus faible que PIP.

III.5.2.2.5 Inconvénients

- OCPP et ICPP n'éliminent pas l'inversion de priorité.

III.5.3 Stack Resource

L'important nombre de commutations de contextes induits par PCP a conduit à définir un nouveau protocole basé sur *les piles de ressources* [Bak,91].

Le protocole SRP permet de limiter le nombre de changements de contextes entres les tâches en ne permettant la préemption d'une tâche par une autre que si on est sûr que les ressources utilisées par cette dernière seront disponibles [Sha,Raj&Leh,90].

III.5.3.1 SRP (Stack Resource Policy)

III.5.3.1.1 Principe

L'idée de base de SRP est que les tâches sont bloquées au démarrage de leur exécution si on sait que les ressources qu'elles utilisent ne seront pas disponibles [Cha,01].

SRP introduit les notions de *niveau de préemption* (preemption level) et de *plafond de préemption* (preemption ceiling) :

III.5.3.1.2 Règle d'ordonnancements

- Chaque tâche T_i possède un niveau de préemption fixe P'_i, et une tâche T_i ne peut préempter une tâche T_j que si $P'_i > P'_j$. Contrairement à PIP et PCP, SRP peut être

utilisé aussi bien avec RM qu'avec EDF [Ram,97]. Dans ce dernier cas, *les niveaux de préemption doivent être ordonnés inversement par rapport aux dates d'échéance* D_i.

- Chaque ressource R_k possède un plafond de préemption $\pi(R_k)$ qui est le maximum des niveaux de préemption des tâches pouvant bloquer sur R_k [Ela,02].

$$\boldsymbol{\pi(R_k) = max_{T_i\ BlocOn(R_k)}\ P'_i} \qquad \textbf{(III. 11)}$$

$\pi(R_k)$: P' max des tâches qui l'utilisent à cet instant [Ram,97].

- Le système possède un plafond de préemption π' qui est le maximum des $\pi(R_k)$ des ressources en cours d'utilisation.

$$\boldsymbol{\pi' = max\ \pi(R_k)} \qquad \textbf{(III. 12)}$$

Les tâches prêtes à s'exécuter sont placées dans une pile par ordre décroissant de leur priorité. On effectue un test de préemption sur le sommet de la pile: pour qu'une tâche T_i puisse préempter T_j, il faut que sa priorité soit la plus forte parmi les tâches prêtes à s'exécuter, mais il faut également que son niveau de préemption soit supérieur au plafond de préemption du système [Ela,02].

Pour les deux tâches T_i et T_j de priorité respective P_i et P_j [Ram,97]

Si *les priorités des tâches sont dynamiques* ***alors***

$$\boldsymbol{si\ P_i > P_j\ et\ A_i > A_j\ alors\ P'_i > P'_j}$$

Si les priorités sont statiques alors les niveaux de préemptions P' sont pris égales aux priorités fixes P (c'est-à-dire $\boldsymbol{P'_i = P_i\ et\ P'_j = P_j}$)

Quand une tâche T_i demande une ressource R_k elle ne l'obtient que si son $\boldsymbol{P'_i > \pi'}$ ou si elle détient la ressource qui fixe le plafond courant.

Formellement le test de préemption à réalisé pour que T_i puisse préempte T_j est le suivant [Ela,02] :

$$\begin{cases} \boldsymbol{P'_i > P'_j} \\ \boldsymbol{P'_i > \pi'} \end{cases} \qquad \text{(III. 13)}$$

Si ce n'est pas le cas, on garde la tâche dans la pile et on effectue de nouveau le test à chaque fois que π' change de valeur.

III.5.3.1.3 Exemple

Considérons 3 tâches T_0, T_1 et T_2, avec $P_0 > P_1 > P_2$, ainsi que deux ressources partagées R_0 et R_1 telles que T_0 utilise R_0 puis R_1, T_1 utilise R_1 puis R_0 et T_2 utilise R_0

On a donc:

$\pi(R_0) = P'_0$

$\pi(R_1) = P'_0$

$P'_0 < P'_1 < P'_2$

On peut obtenir l'ordonnancement suivant [God,90] :

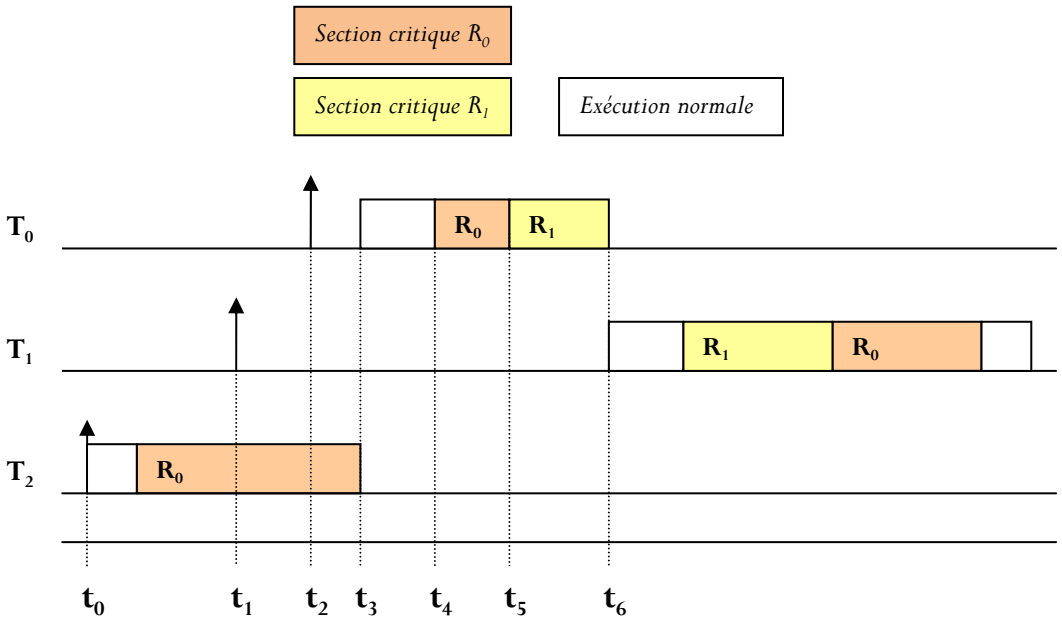

Figure III.9 : Ordonnancement des tâches avec SRP [Ela,02]

- A l'instant t_0 : T_2 est activé, commence son exécution et peut prendre R_0, le plafond de préemption du système devient $\pi' = \pi(R_0) = P'_0$.
- A l'instant t_1 : T_1 est activé, mais $P'_1 < P'_2$ donc T_1 ne peut préempter T_2.
- A l'instant t_2 : T_0 est activé, mais T_2 peut continuer à s'exécuter car $P'_2 > P'_0$.
- A l'instant t_3 : T_2 libère R_0 et termine son exécution. T_0 et T_1 sont les tâches prêtes à s'exécuter. Comme $P_0 > P_1$ alors P_0 commence son exécution.

- A l'instant t_4 : T_0 prend R_0, on a alors $\pi' = \pi(R_0) = P'_0$. Le test de préemption sur T_1 n'est pas satisfaisant puisque $P_1 < P_0$, donc T_0 continue son exécution.
- A l'instant t_5 : T_0 libère R_0 et prend R_1, on a alors $\pi' = \pi(R_1) = P'_0$. Le test de préemption sur T_1 n'est pas satisfaisant puisque $P_1 < P_0$, donc T_0 continue son exécution.
- A l'instant t_6 : T_0 termine son exécution, T_1 peut alors s'exécuter.

III.5.3.1.4 Caractéristiques

Grâce au test de préemption, on est sûr que lorsqu'une tâche commence son exécution, elle ne sera pas bloquée sur une ressource. Les situations d'interblocage sont donc impossibles, et il en est de même pour les chaînes de blocage [Ela,02].

Le calcul du temps de blocage est le même que pour PCP. Pour une tâche T_i il est égal à la durée de la plus longue section critique parmi les tâches pouvant la bloquer. La seule différence est que les tâches peuvent être bloquées uniquement au moment de la préemption, ce qui supprime les changements de contextes supplémentaires induits par PCP lors des blocages [God,90].

$$B_i = \max_{j,k}\{D_{j,k} : P'_j < P'_i \; et \; \pi(R_k) \geq P'_i\} \qquad \textbf{(III. 14)}$$

Comme indiqué précédemment, le protocole SRP s'utilise aussi bien avec un ordonnanceur de type fixe (RM) que dynamique (EDF). Le critère d'ordonnançabilité avec RM est le même que pour PIP et PCP. Pour EDF, le système de tâches doit vérifier l'inéquation suivante [Bak,91] :

$$\boldsymbol{pour\ tout\ i \leq n, \sum_{k=1}^{i} (C_k/T_k) + B_i/T_i \leq 1} \qquad \textbf{(III. 15)}$$

III.5.3.1.5 Avantages

- La facilité d'implémentation en partie due au fait que, les blocages se font au niveau de la préemption et non au niveau de l'accès aux ressources.

- La Réduction est maximale dans le nombre de changements de contextes entres les tâches

III.5.3.1.6 Inconvénients

- L'inconvénient de cette méthode est qu'une tâche qui n'utilise aucune ressource partagée peut se retrouver bloquée par le test de préemption.

III.6 Etude comparatif

Le but du tableau III.1 est de comparer les différents protocoles permettant de limiter les effets de l'inversion de priorités, par l'analyse de leurs caractéristiques de fonctionnement et leur difficulté d'implémentation.

	PIP	**PCP**	**SRP**
Blocages	Accès aux ressources	Accès aux ressources	Préemption
Types de blocage	Direct, héritage de priorité	Direct, héritage de priorité, plafond de priorité	Direct, Plafond de préemption
Nombre de blocages	Min{n, m}	1	1
Temps de blocage	$\begin{cases} B_i = \min\{ B_i^l , B_i^s \} \\ B_i^l = \sum_{j=i+1}^{n} \max_k \{D_{j,k} : \pi(R_k) \geq P_i \} \\ B_i^s = \sum_{k=1}^{m} \max_{j<i} \{D_{j,k} : \pi(R_k) \geq P_i \} \end{cases}$	$B_i = \max_{j,k} \{D_{j,k}: P_j < P_i$ et $\pi(R_k) \geq P_i \}$	$B_i = \max_{j,k} \{D_{j,k} : P'_j < P'_i$ et $\pi(R_k) \geq P'_i \}$
Interblocage	Oui	Non	Non
Chaîne de blocage	Oui	Non	Non
Algorithme	RM	RM	RM/EDF
Priorité	Fixe	fixe	Fixe/dynamique
Implémentation	Dure	Moyenne	Facile

Tableau III.1 : Comparaison entre OCPP, ICPP et SRP [Ela,02]

III.7 Exemple concret

Dans le but à concrétiser le phénomène d'inversion de priorité nous abordons l'exemple suivant de la mission Mars Pathfinder (4 Juillet 1997) qui a été pris de la référence [INF,09].

Le système Mars Pathfinder considère plusieurs processus, un processus1 de haute priorité, un processus 2 de basse priorité et d'autres processus de priorités moyennes ainsi que une zone de mémoire partagée (appelée *information bus*) contenant des informations partagées entre ces processus. Sachant que :

• ***Processus 1*** modifie régulièrement ces informations (accès en exclusion mutuelle).
Si cette tâche ne s'exécute pas pendant une certaine durée (révélateur d'un problème), tout le système est réinitialisé (Reset).

• ***Processus 2*** s'exécute peu fréquemment, récolte des informations météo, et écrivant ces information dans *information bus* (accès en exclusion mutuelle).

• ***Processus 3*** (dit de communication), débloqué sur occurrence d'une interruption.

Le Système ordonnancé par RM, sur VxWorks provoque le problème d'*inversion de priorité* qui survient à chaque fois que l'interruption qui débloque le processus 3 a lieu pendant que le processus 2 a l'accès exclusif sur l'information bus (en section critique).

III.8 Conclusion

Les différents protocoles illustrés précédemment permettent de limiter les effets des inversions de priorité dans l'ordonnancement de tâches périodiques sur un processeur. Ces protocoles ont été définis pour répondre à des problèmes supplémentaires ou non résolus. Ainsi, les risques d'interblocage et de chaînes de blocages avec PIP ont été résolus par PCP [Ela,02]. De même, le nombre important de changements de contexte entres les tâches avec PCP a conduit à définir le protocole SRP. Ce dernier, fonctionnant aussi bien avec un ordonnanceur à priorité fixe que dynamique, se distingue de plus par sa facilité d'implémentation [Bak,91], en partie due au fait que les blocages se font au niveau de la préemption et non au niveau de l'accès aux ressources. Le chapitre suivant sera consacré à présenté le nouveau protocole d'allocation de ressource proposé (CDP) pour contrer le phénomène d'inversion de priorité.

Chapitre IV :
Le protocole CDP proposé

IV.1 Introduction

Souvent, nous distinguons des tâches utilisant des ressources partagées. A cette répercussion l'accès simultané à ces ressources doit être contrôlé afin de garder un état cohérent. Le partage de ressources en mutuelle exclusion peut engendrer des *inter-blocages* ou des *inversions de priorités*. La situation d'inversion de priorité survient lorsque l'attribution du CPU est basée sur des priorités, mais que les processus ne sont pas indépendants[1]. Alors, lorsqu'une tâche T_1 demande une ressource déjà allouée à une autre tâche T_2, T_1 se met en attente de la ressource même si elle est prioritaire. Des fois ce problème mène à des dégâts inattendus. Ainsi, pour limiter les problèmes des deux phénomènes pré-cités, des protocoles tels que PIP [Sha,Raj&Leh,90] [Ela,02] [God,90], OCPP, ICPP [Sha,Raj&Leh,90] [Ela,02] [God,90] [Che&Lin,90] et SRP [Ela,02] [Bak,91] [Ram,97] [God,90] ont été introduites. Devant ce problème, nous essayons dans ce chapitre de présenter un nouveau protocole d'allocation de ressources nommé *protocole à plafond dynamique* « ***CDP : C**eiling **D**ynamic **P**rotocol* » pour systèmes temps réels où la survenue du phénomène d'inversion de priorité et le blocage des tâches prioritaires sur une ressource partagée à accès exclusive sont rares. Le cadre de l'étude porte sur un ordonnancement monoprocesseur, préemptif de tâches périodiques avec des ressources partagées de plafond de priorité dynamique.

[1] Les processus partagent des ressources en communs.

Au cours de ce chapitre, nous détaillons l'idée du protocole CDP ainsi que les règles d'ordonnancements, les caractéristiques, les avantages et les inconvénients qui lui sont associés. Ensuite, pour comparer le CDP avec les anciens protocoles nous présentons ses caractéristiques de fonctionnement à travers une étude comparative complémentaire. Puis, nous proposons un exemple et nous comparons les résultats d'application obtenus.

IV.2 L'objectif

Dans notre proposition nous essayons de fournir un nouveau protocole d'allocation de ressource pour systèmes temps réels où la survenue du phénomène d'*inversion de priorités* et le *blocage* des tâches prioritaires sur une ressource partagée à accès exclusive sont très exceptionnels. Dans certain cas l'*inversion de priorités* sera évitée complètement, le temps de blocage était négligeable et le nombre de commutation de contexte (***Nbr_comut***) sera trop réduit, tend souvent vers à:

$$\boldsymbol{Nbr\_comut = (nbr\ de\ tâches \times 2) - 2} \qquad \textbf{(IV. 1)}$$

IV.3 Notations

Le tableau IV.1 présente les notations utilisées dans ce chapitre :

Notations	*Significations*
T_i	la tâche T_i
T_a	Tâche en attente (dans l'état bloquée ou interrompue)
R_k	la ressource R_k
P_i	la priorité statique de T_i
π'	La priorité plafond courante (la plus haute priorité à l' instant t)
$Nb_{T_{user(R_k)}}$	Le nombre de tâches sollicitant la section critique R_k
$PDM(T_i)$	L'instant d'arrivé de la tâche T_i
D_{R_k}	Le nombre de cycles consécutifs requis à l'utilisation de la ressource R_k
D_{TM}	La durée du temps mort

Tableau IV.1 : Notations utilisées

IV.4 L'outil de gestion du temps

On a désigné le diagramme de *Gantt* comme une technique pour montrer l'exécution de différentes tâches de notre modèle. Ce type de diagramme a été mis au point par un américain Henry Gantt. On représente au sein de deux axes, en ligne les différentes tâches et en colonne les unités de temps exprimées par cycle. La durée d'exécution d'une tâche est matérialisée au sein du diagramme par un ou plusieurs rectangles selon le nombre de cycles requise à l'exécution de cette tâche [Sen,07].

L'objectif principal de diagramme de Gantt est de déterminer la date de réalisation de chaque tâche ainsi que celle de l'application.

IV.5 Le protocole CDP

Dans cette section nous détaillons le principe de base et les règles de fonctionnement du protocole CDP. Ainsi, on dit que :

- Une tâche est *préemptée* si elle est interrompue par une tâche de priorité statique plus élevée.
- Une tâche T_i est *bloquée* si elle se met en attente d'une ressource R_k qui sera utilisé ultérieurement par une tâche prioritaire, sachant que durant le temps de ce blocage le processeur ou bien ne fait rien (*temps mort* forcé) ou il exécute une tâche de priorité statique plus basse que T_i qui bénéficiant d'une priorité dynamique plus élevé (π': *la plus haute priorité des tâches à l' instant t*).

IV.5.1 Le principe du CDP [Dou,Gho,10]

L'idée de base de CDP est de rendre dynamique le plafond de priorité d'une ressource R_k interdisant une tâche T_i d'entrer en section critique R_k si celle-ci sera sollicitée ultérieurement par une autre tâche de priorité supérieure. Dans le cas où T_i peut libérer la section critique désirée avant qu'une tâche prioritaire n'atteigne sa date d'activation, elle peut la prendre. Chaque tâche n'a au départ qu'une seul priorité fixe assignée au départ l'algorithme d'ordonnancement statiques RM, un instant d'arrivé ainsi qu'une séquence des unités à exécuté. Des fois, une priorité temporaire est attribuée à une tâche de priorité basse

pour éviter le gaspillage du temps CPU (*exploitation du temps mort*) et au même temps empêcher le phénomène d'*inversion de priorité*.

Chaque ressource R_k a une valeur dynamique qui est la priorité plafond, notée $\pi(R_k)$ et nommée *priorité plafond* de R_k égale à la plus haute priorité des tâches nécessitant R_k pour accomplir leurs exécutions ; c'est-à-dire que $\pi(R_k)$ change (diminue) dès que une tâche prioritaire courante quitte définitivement la ressource R_k . De manière plus formelle :

$$\pi(R_k) = \begin{cases} \max_{T_i \text{ nécessite } (R_k)} P_i \\ \\ \varphi \end{cases} \qquad (IV.2)$$

Où : $\max_{T_i \text{ nécessite } (R_k)} P_i$ est la valeur maximale des priorités P_i des tâches nécessitent R_k pour accomplir leurs exécutions et φ : est une priorité plus basse que la priorité de la tâche la moins prioritaire des tâches qui sont utilisées R_k. Si toutes les tâches nécessitent R_k sont terminées leurs exécutions de dans alors $\pi(R_k) = \varphi$.

Ce protocole suppose que chaque tâche possède une priorité fixe et que les ressources utilisées, les PDM, les séquences d'exécutions ainsi que les durées dont les ressources sont utilisées sont connues avant le début de l'exécution.

IV.5.2 Les règles d'ordonnancement proposés pour CDP

- Si deux tâches ou plus veulent commencer (ou continuer) leur exécution à un instant t alors c'est la tâche prioritaire qui prend la main pour s'exécuter.
- Une tâche T_i de priorité statique P_i peut préempter l'exécution de la tâche courante T_j de priorité statique P_j et commence son exécution si et seulement si $P_i > P_j$.
- Lorsqu'une tâche T_i de priorité statique P_i veut prendre une ressource R_k à un instant t, on a l'une des situations suivantes [Dou,Gho,10] :
  - Si $Nb_{T_{user}(R_k)} = 1$ (R_k *ressource privée*) alors T_i peut prendre la ressource R_k et entre en section critique.

↳ Si $(\boldsymbol{Nb_{T_{user(R_k)}} > 1})$ et $(\boldsymbol{P_i = \pi(R_k)})$ alors T_i peut prendre la ressource R_k.

↳ Si $(\boldsymbol{Nb_{T_{user(R_k)}} > 1})$ et $(\boldsymbol{P_i < \pi(R_k)})$ alors on a trois cas possibles :

(a) Si l'exécution de T_i dans $\boldsymbol{R_k}$ se termine avant ou avec le minimum des instants d'arrivés (PDM[2]) de toutes les tâches prioritaires à Ti alors T_i peut prendre la ressource R_k et entre en section critique [Dou,Gho,10]. De manière formelle:

$$\boldsymbol{P_j > P_i;\ D_{R_k} \leq \left[min\left(PDM(T_j)\right) - t\right]} \qquad \textbf{(IV.3)}$$

(b) S'il y a des tâches $\boldsymbol{T_a}$ de priorités inférieures ou égales à $\boldsymbol{P_i}$ *qu*i sont [Dou,Gho,10] :

- interrompues par $\boldsymbol{T_i}$ ou par des tâches de priorité inférieures à $\boldsymbol{P_i}$;
- bloquée sur une ressource $\boldsymbol{R_k}$;

alors on commence le test d'allocation par la tâche la plus prioritaire entre eux :

☞ Si $\boldsymbol{T_a.etat = prête}$ alors $\boldsymbol{T_a}$ commence son exécution.

☞ Si $\boldsymbol{T_a.etat = préemptée}$ alors $\boldsymbol{T_a}$ reprend son exécution.

☞ Si $\boldsymbol{T_a.etat = bloquée\ sur\ R_{k'}}$ (k' peut être égal à k) alors si $\boldsymbol{D_{R_{k'}} \leq [min\ (PDM(T_j)) - t]}$ alors la tâche $\boldsymbol{T_a}$ rehausse sa priorité à celle de $\boldsymbol{\pi'}$ (héritage de la priorité $\boldsymbol{\pi'}$). Elle est alors exécutée avec la priorité hérité jusqu'à une tâche de priorité supérieure à $\boldsymbol{\pi'}$ atteigne sa date d'activation, dans ce cas sa priorité redevient celle qu'elle était [Dou,Gho,10].

(c) La non prise de R_k par T_i provoque un temps mort qui dure :

[Le minimum des PDM des tâches n'ont pas encore déclenchées leurs exécution - l'instant dans T_i veut prendre R_k]. De manière formelle la durée de temps mort $\boldsymbol{D_{TM}}$ est donnée comme suit [Dou,Gho,10] :

$$\boldsymbol{P_j > P_i;\ D_{TM} = \left[min\left(PDM(T_j)\right) - t\right]/cycles} \qquad \textbf{(IV.4)}$$

Tel que : $\boldsymbol{T_j}$ n'ont pas encore atteint leur date d'activation

[2] **P**ériode de **D**épart **M**inimale qu'est égale à l'instant d'arrivé de la tâche.

IV.5.3- Exemples d'applications

A. Exemple 1

Considérons trois (03) tâches périodiques triées en ordre de priorité tel que T_0 est la tâche la plus prioritaire et T_2 la moins prioritaire, ainsi que deux ressources partagées R_1 et R_2 et une ressource privées R_0. T_0 utilise R_0 puis R_1, T_1 utilise R_2 et T_2 utilise R_2 puis R_1.

Les priorités, les instants d'arrivés ainsi que les séquences d'exécutions des tâches sont présentés dans le tableau IV.2 :

Tâches	*Priorités*	*PDM*	*Séquences d'exécution*
T_0	20	3	$ER_0ER_1R_1$
T_1	15	1	ER_2R_2E
T_2	10	0	ER_2R_1E

Tableau IV.2 : Les tâches avec leurs Pi, PDM et séquences

Dans un état initial on a :

- $\pi(R_0) = P_0$
- $\pi(R_1) = P_0$
- $\pi(R_2) = P_1$

La colonne *PDM* indique que T_0 ne doit pas démarrer avant l'instant 3 (mais elle pourrait démarrer après) alors que T_1 ne doit pas démarrer avant l'instant 1, etc [INF,09].

La colonne séquences d'exécutions indique que la tâcheT_0 doit commencer son exécution par une cycle d'exécution normal (*E*) dans aucune section critique puis entrer dans la section critique R_0 pour une cycle d'exécution, ensuite entrer dans une autre cycle d'exécution normal et finalement prend la ressource R_1 pendant deux cycles d'exécutions. Donc au total on a pour T_0 ($ER_0ER_1R_1$). Même principe pour T_1 et T_2 [INF,09].

La figure IV.1 montre l'ordre d'exécution des 3 tâches:

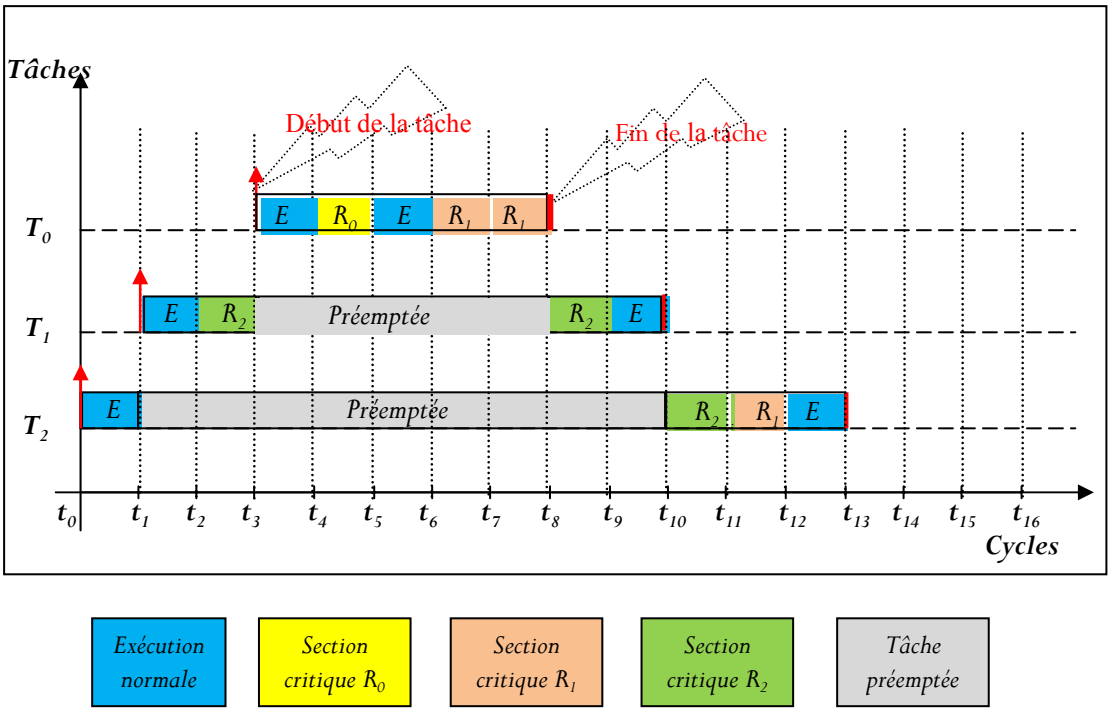

Figure IV.1 : L'ordonnancement des tâches avec CDP

A l'instant t_0 : T_2 est activée et commence son exécution.

A l'instant t_1 : T_1 préempte T_2 et commence a s'exécuté car $P_1>P_2$.

A l'instant t_2 : T_1 peut prendre la ressource R_2 car $(\boldsymbol{Nb_{T_{user(R_2)}} > 1)et(P_1 = \pi(R_2))}$, T_1 entre en section critique.

A l'instant t_3 : T_0 est réveillée et préempte T_1 car $P_0>P_1$, T_0 commence son exécution.

A l'instant t_4 : T_0 continue à s'exécuté et prendre la ressource R_0 et entre en section critique parce que $\boldsymbol{(Nb_{T_{user(R_0)}} = 1)et(P_0 = \pi(R_0))}$.

A l'instant t_5 : T_0 libère R_0 définitivement $(\boldsymbol{\pi(R_0) = \varphi})$ et continue son exécution dans aucune section critique.

A l'instant t_6 : T_0 peut prendre R_1 pendant deux cycles car $\boldsymbol{(Nb_{T_{user(R_1)}} > 1)et(P_0 = \pi(R_1))}$.

A l'instant t_8 : T_0 termine son exécution ($\pi(R_1)= P_2$) et T_1 reprend la sienne.

A l'instant t_{10} : T_1 termine son exécution ($\pi(R_2)= P_2$), T_2 reprend la sienne et prend la ressource R_2 car $P_2=\pi(R_2)$.

A l'instant t_{11} : T_2 quitte R_2 définitivement ($\pi(R_2)= \boldsymbol{\varphi}$) et prendre R_1 et entre en section critique car $P_2=\pi(R_1)$.

A l'instant t_{12} : T_2 quitte R_1 ($\pi(R_1)= \boldsymbol{\varphi}$) et continue son exécution dans aucune section critique.

A l'instant t_{13} : T_2 termine son exécution.

B. Exemple 2 (cas de blocage sur R_k)

Soient trois (03) tâches périodiques, T_0 est la tâche la plus prioritaire et T_2 la moins prioritaire. Ainsi que deux ressources partagées R_1 et R_2 et une ressource privées R_0. T_0 utilise R_0 puis R_1, T_1 utilise R_2 puis R_1 et T_2 utilise R_2 puis R_1.

Le tableau IV.3 récapitule l'énoncé de l'exemple :

Tâches	*Priorités*	*PDM*	*Séquences d'exécution*
T_0	*20*	*3*	$ER_0ER_1R_1$
T_1	*15*	*0*	$ER_2R_1R_1E$
T_2	*10*	*1*	ER_2R_1E

Tableau IV.3 : Les tâches avec leurs Pi, PDM et séquences

Dans un état initial on a :

- $\pi(R_0) = P_0$
- $\pi(R_1) = P_0$
- $\pi(R_2) = P_1$

La figure IV.2 montre l'ordre d'exécution des 3 tâches :

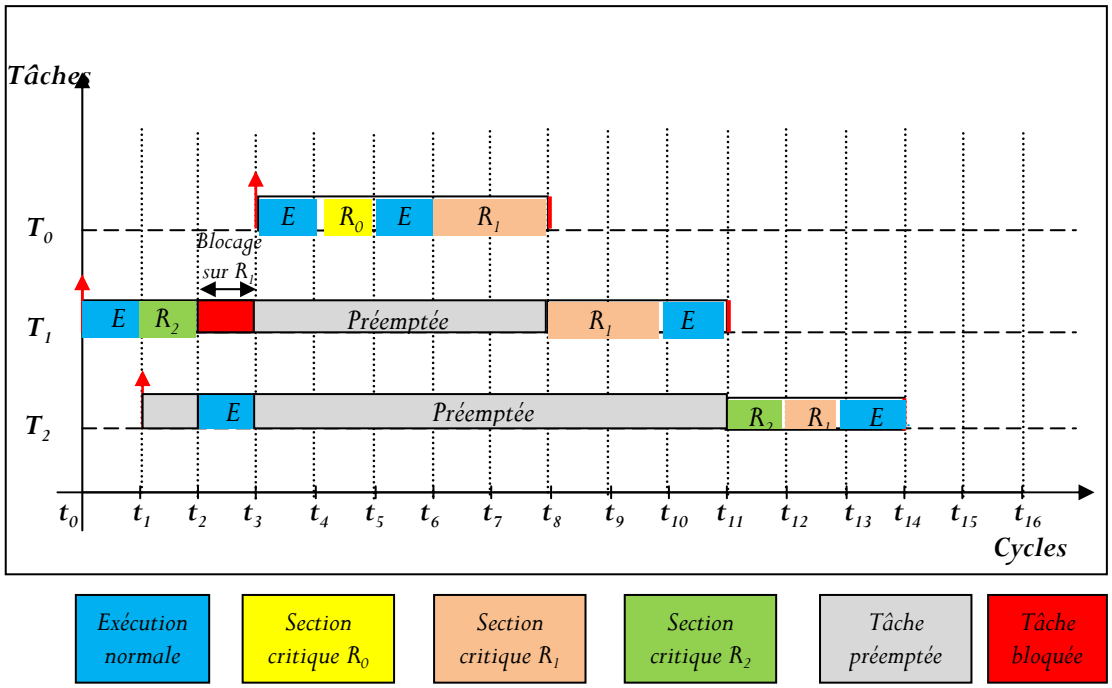

Figure IV.2 : L'ordonnancement des 3 tâches avec CDP

A l'instant t_0 : T_1 est activée et commence son exécution.

A l'instant t_1 : T_2 est activée, T_1 peut prendre la ressource R_2 car :

$$(\boldsymbol{Nb_{T_{user(R_2)}} > 1})\boldsymbol{et}(\boldsymbol{P_1 = \pi(R_2)}).$$

A l'instant t_2 : T_1 quitte R_2 ($\pi(R_2)= P_2$) mais elle ne peut pas prendre la ressource R_1 car

$$(\boldsymbol{Nb_{T_{user(R_1)}} > 1})\ \boldsymbol{et}\ \boldsymbol{\left(P_1 < \pi(R_1)\right)}$$

$$\boldsymbol{et(\ D_{R_1} = 2\ périodes > [PDM(T_0) - t_2] = t_3 - t_2 =\ 1\ période}),$$

selon la règle d'ordonnancement (b) on a $T_2.état$ = *prête* alors T_2 hérite de $\boldsymbol{\pi'}$, $P_2 = \boldsymbol{\pi'} = P_1$ et commence son exécution.

A l'instant t_3 : T_0 est réveillée et préempte T_2 car $P_0 > P_1$, T_0 commence son exécution et T_2 reprend sa priorité initiale P_2.

A l'instant t_4 : T_0 continue à s'exécuté et prendre la ressource R_0 et entre en section critique parce que $\boldsymbol{(Nb_{T_{user(R_0)}} = 1)et\left(P_0 = \pi(R_0)\right)}$.

A l'instant t_5 : T_0 quitte R_0 ($\pi(R_0)= \boldsymbol{\varphi}$) et continue son exécution dans aucune section critique.

A l'instant t_6 : T_0 peut prendre R_1 pendant deux cycles consécutives car

$$(\boldsymbol{Nb_{T_{user(R_1)}} > 1)et\big(P_0 = \pi(R_1)\big)}.$$

A l'instant t_8 : T_0 libère R_1 définitivement ($\pi(R_1)$= P1) et termine son exécution. T_1 reprend la sienne et prendre la ressource R_1 pendant deux cycles consécutifs car $P_1=\pi(R_1)$

A l'instant t_{10} : T_1 quitte R_1 ($\pi(R_1)= P_2$) et continue son exécution dans aucune section critique durant un cycle.

A l'instant t_{11} : T_0 termine son exécution et T_2 peut reprendre la sienne. T_2 prend R_2 car $P_2=\pi(R_2)$

A l'instant t_{12}: T_2 quitte R_2 ($\pi(R_2)= \boldsymbol{\varphi}$) et entre dans la section critique R_1 car $P_2=\pi(R_1)$.

A l'instant t_{13} : T_2 quitte R_1 ($\pi(R_1)= \boldsymbol{\varphi}$) et continue son exécution dans aucune section critique.

A l'instant t_{14}: T_2 termine son exécution.

C. Exemple 3 (cas de gaspillage de temps CPU)

Soient trois (03) tâches périodiques, T0 est la tâche la plus prioritaire et T2 la moins prioritaire. Ainsi que deux ressources partagées R0 et R1. T0 utilise R0 puis R1, T1 utilise uniquement R1 et T2 utilise R0 puis R1.

Le tableau IV.4 récapitule l'énoncé de l'exemple :

Tâches	*Priorités*	*PDM*	*Séquences d'exécution*
T_0	*20*	*5*	ER_0R_1E
T_1	*15*	*2*	$E\ R_1R_1R_1E$
T_2	*10*	*0*	$ER_0R_0R_1R_1E$

Tableau IV.4 : Les tâches avec leurs Pi, PDM et séquences

Dans un état initial on a :

- $\pi(R_0) = P_0$
- $\pi(R_1) = P_0$

La figure IV.3 montre l'ordre d'exécution des 3 tâches :

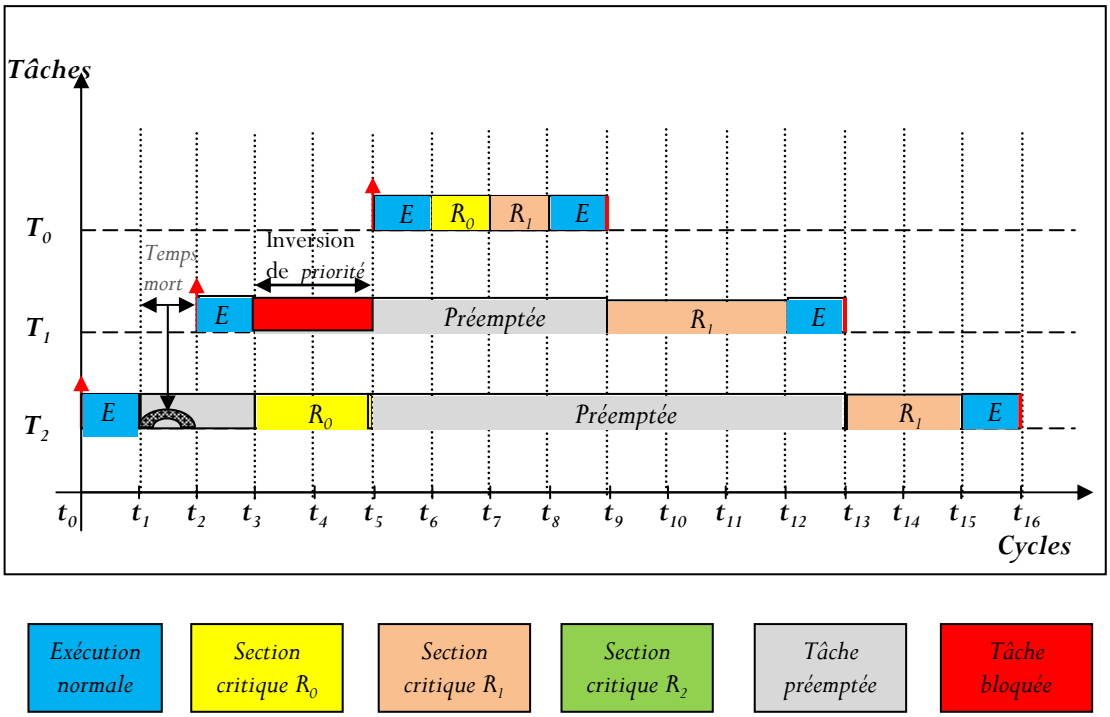

Figure IV.3 : L'ordonnancement des 3 tâches avec CDP

A l'instant t_0 : T_2 est activée et commence son exécution.

A l'instant t_1 : T_2 ne peut pas prendre la ressource R_0 car $(\boldsymbol{Nb_{T_{user(R_0)}} > 1})$

$\boldsymbol{et(P_2 < \pi(R_0))et(\ D_{R_0} = 2\ cycles > [PDM(T_1) - t_1] = \ 1\ cycle)}$,

selon la règle d'ordonnancement (c) le processeur reste inactif pendant un temps mort qui dure $\boldsymbol{D_{TM} = [PDM(T_1) - t_1] = t_2 - t_1 = 1\ cycle}$.

A l'instant t_2 : T_1 est activée et commence son exécution.

A l'instant t_3 : T_1 ne peut pas prendre R_1 car $\boldsymbol{(Nb_{T_{user(R_1)}} > 1)et}$

$\boldsymbol{(P_1 < \pi(R_1))et(D_{R_1} = 3\ cycles > [PDM(T_0) - t_3] = t_5 - t_3 = 2\ cycles)}$, selon la règle d'ordonnancement (b) on a $T_2.état$=*bloquée* alors T_2 hérite de $\boldsymbol{\pi'}$, $P_2 = \boldsymbol{\pi'} = P_1$ et reprend son exécution, elle peut prendre la ressource R_0 car

$\boldsymbol{D_{R_0} = 2\ cycles \leq [PDM(T_0) - t_3] = \ 2\ cycles}$.

A l'instant t_5 : T_2 libère R_0, T_0 est réveillée et préempte T_2 car $P_0 > P_1$, T_0 commence son exécution et T_2 reprend sa priorité initiale P_2.

A l'instant t_6 : T_0 peut prendre la ressource R_0 car $P_0 = \pi(R_0)$.

A l'instant t_7 : T_0 quitte R_0 ($\pi(R_0)= \varphi$) et prend la ressource R_1 car $P_0=\pi(R_1)$.

A l'instant t_8 : T_0 libère R_1 ($\pi(R_1)= P_1$) et continue son exécution dans aucune section critique.

A l'instant t_9 : T_0 termine son exécution et T_1 peut reprendre la sienne, elle prend R_1 pendant 3 cycles car $P_1=\pi(R_1)$.

A l'instant t_{12} : T_1 libère R_1 ($\pi(R_1)= P_2$) et continue son exécution dans aucune section critique.

A l'instant t_{13} : T_1 termine son exécution, T_2 reprend la sienne et prend la ressource R_1 car $P_2=\pi(R_1)$.

A l'instant t_{15} : T_2 quitte R_1 ($\pi(R_1)= \varphi$) et continue son exécution dans aucune section critique.

A l'instant t_{16} : T_2 termine son exécution.

IV.5.4 Les caractéristiques du CDP

- Le *CDP* empêche les situations d'interblocage (*deadlock*), qui peuvent survenir lorsqu'il y a des ressources partagées. Il provoque un cas de *blocage sur la ressource* R_k lorsque la durée de la section critique de la tâche courante est assez élevée et peut dépasser la PDM d'une tâche prioritaire (c'est le cas dans l'exemple 2 à l'instant t_2 et dans l'exemple 3 à l'instant t_3) [Dou,Gho,10].
- Le *CDP* exécute la tâche la plus prioritaire sans aucune interruption [Dou,Gho,10].
- Le temps de blocage c'est la durée pendant laquelle une tâche de priorité moyenne T_m ne peut plus progresser à cause de la condition (*la durée d'exécution de Tm dans* R_k*>l'instant d'arrivé d'une tâche prioritaire*), une tâche de moindre priorité bénéficiant de cette situation de blocage et reprend son exécution avec une priorité plus élevé (hérité de π'). On peut calculer le temps de blocage B_m d'une tâche T_m à l'instant t sur une ressource R_k comme suit : À chaque fois que le protocole d'ordonnancement (b) ou (c) est exécutée on compte la durée de cette exécution, le temps de blocage de T_m est la somme de ces durés [Dou,Gho,10]. De manière plus formelle :

$$\boldsymbol{P_j > P_m : \ B_m = B_m + \left[min\left(PDM(T_j)\right) - t \right]} \qquad \textbf{(IV. 5)}$$

Le tableau IV.5 montre les différentes caractéristiques de fonctionnements associées au protocole CDP proposé.

	CDP
Blocages	*Accès aux ressources*
Types de blocage	*Section critique longue*
Nombre de blocage de la tâche prioritaire	*0*
Nombre de blocage des tâches moyennes	*Variable (en fonction des PDM et des durées de sections critiques)*
Temps de blocage de Tm	$\mathbf{P_j > P_m}$: $\boldsymbol{B_m = B_m + [\min\left(PDM(T_j)\right) - t]}$
Interblocage	*Non*
Chaîne de blocage	*Non*
Gaspillage de temps CPU	*Oui / Non*
Algorithme d'ordonnancement	*RM*
Priorité	*Fixe*
Temps d'exécution de la tâche prioritaire	*Nombre de cycles requis à l'exécution*
Implémentation	*Moyen*

Tableau IV.5 : Les caractéristiques attribuées au protocole CDP [Dou,Gho,10]

IV.5.5 Avantages [Dou,Gho,10]

- ✓ CDP ne mène jamais à un interblocage ;
- ✓ CDP assure l'exécution de la tâche la plus prioritaire sans aucune interruption ;
- ✓ Bien adapté lorsque les PDM des tâches sont trop proches et les sections critiques sont de courte duré d'exécution;
- ✓ Temps de blocage très réduit ;
- ✓ Minimise le nombre de commutation de contexte.

IV.5.6 Désavantages [Dou,Gho,10]

- ✗ Le gaspillage de temps CPU (*temps mort*) implique un temps d'exécution total trop élevé.
- ✗ Les sections critiques de longue durée d'exécution provoquent souvent des cas de blocage.

IV.6 Étude comparative

Nous proposons dans cette section un exemple qui sera appliqué avec les protocoles d'inversion de priorité cités dans l'état de l'art et dans la contribution. Nous présentons par la suite une étude comparative des résultats obtenus.

L'idée de l'exemple a été prise de la référence [INF,09]. Considérons quatre (04) tâches T_0, T_1, T_2 et T_3, avec $P_0 > P_1 > P_2 > P_3$, ainsi que deux ressources partagées R_0, R_1 telles que : T_0 utilise R_0 pendant un cycle puis R_1 pendant un cycle, T_1 utilise R_1 pendant deux cycles successives et T_3 utilise R_0 pendant trois cycles de suites.

Les priorités, les instants d'arrivé ainsi que les séquences d'exécutions des tâches sont présentés dans le tableau IV.6 [Dou,Gho,10]:

Tâches	*Priorités*	*PDM*	*Séquences d'exécution*
T_0	25	4	ER_0R_1E
T_1	15	2	ER_1R_1E
T_2	10	2	EE
T_3	4	0	$ER_0R_0R_0R_0E$

Tableau IV.6 : Les tâches avec leurs Pi, PDM et séquences [Dou,Gho,10]

IV.6.1 L'exemple avec les priorités des tâches assignées :

La figure IV.4 montre l'ordre d'exécution des 4 tâches avec leurs priorités assignées :

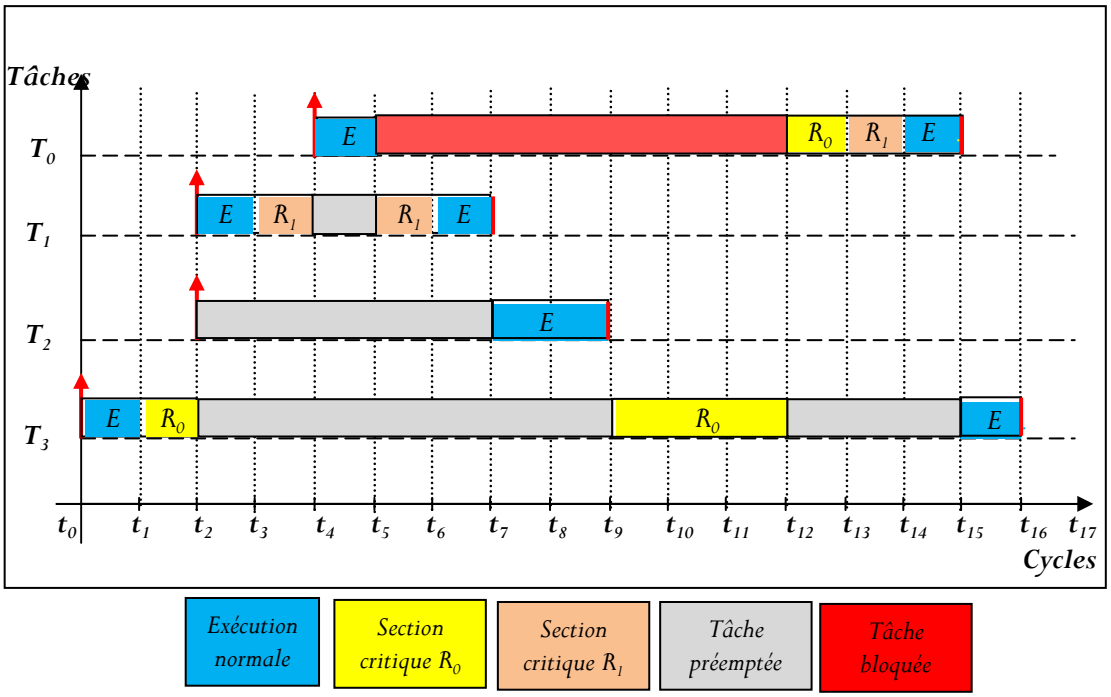

Figure IV.4 : L'ordonnancement des 4 tâches avec leurs priorités assignées [Dou,Gho,10]

A l'instant t_0 : T_3 est activée et commence son exécution.

A l'instant t_1 : T_3 peut prendre R_0 et entre en section critique.

A l'instant t_2 : T_1 préempte T_3 et commence son exécution car $P_1 > P_3$.

A l'instant t_3 : T_1 peut prendre R_1 et entre en section critique.

A l'instant t_4 : T_0 est réveillé et préempte T_1 car $P_0 > P_1$, T_0 commence son exécution.

A l'instant t_5 : T_0 ne peut prendre R_0 car la ressource n'est pas libre. T_1 reprend son exécution dans la section critique R_1 et T_0 se retrouve bloquée.

A l'instant t_6 : T_1 libère R_1 et continue son exécution dans aucune section critique durant un cycle.

A l'instant t_7 : T_1 termine son exécution. T2 s'exécute pendant deux cycles dans aucune section critique.

A l'instant t_9 : c'est la tâche T3 qui continue son exécution dans R_0 durant 3 cycles car T_0 se retrouve bloqué sur cette ressource.

A l'instant t_{12} : T3 libère la ressource R_0, T_0 préempte T_3 et entre en section critique en prenant la ressource R_0.

A l'instant t_{13} : T_0 libère la ressource R_0, prend la ressourcé R1 et entre en section critique pendant un cycle. La durée du blocage dû à l'inversion de priorité est égale à t_{12}-t_5.

A l'instant t_{14} : T_0 libère R_1 et continue son exécution dans aucune section critique durant un cycle.

A l'instant t_{15} : T_0 termine son exécution. T3 s'exécute pendant un cycle dans aucune section critique.

A l'instant t_{16} : T_3 termine son exécution.

Conclusion : ***temps d'exécution de*** $T_0 = t_{15} - t_4 = 11$ ***cycles***

On se serait attendu à ce que T_0 puisse faire sa séquence d'exécution en 4 cycles (ER_0R_1E), car c'est la tâche la plus prioritaire, mais 11 cycles ont été requis. Pire encore, T_1 qui est moins prioritaire que T_0 a terminé plus rapidement sa séquence d'exécution, d'où le nom *inversion de priorité*.

Il existe autant de mécanismes pour éviter (minimiser) cette situation d'inversion de priorité telle que PIP, ICPP, OCPP, SRP :

IV.6.2 L'exemple avec PIP (Priority Inheritance Protocol)

La figure IV.5 montre l'ordre d'exécution des 4 tâches avec le protocole d'ordonnancement PIP:

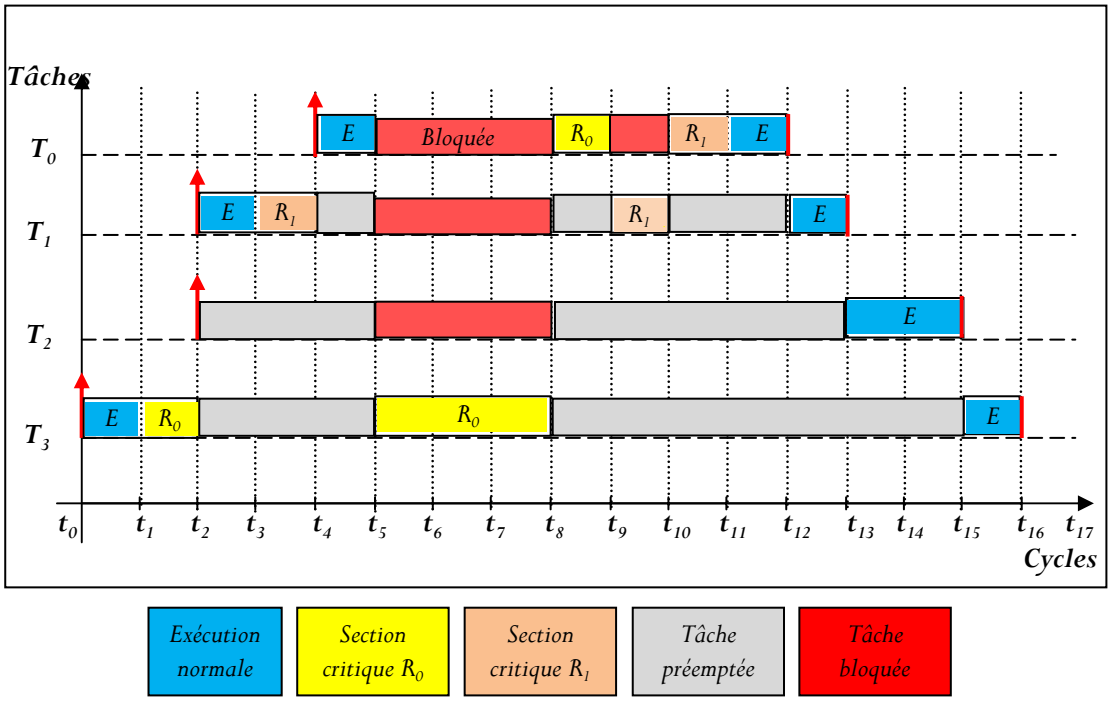

Figure IV.5 : L'ordonnancement des 4 tâches avec PIP [Dou,Gho,10]

A l'instant t_0 *:* T_3 est activée et commence son exécution.

A l'instant t_1 *:* T_3 peut prendre R_0 car elle est libre et entre en section critique.

A l'instant t_2 *:* T_1 préempte T_3 et commence son exécution car $P_1 > P_3$.

A l'instant t_3 *:* T_1 peut prendre R_1 car elle est libre et entre en section critique.

A l'instant t_4 *:* T_0 est réveillée et préempte T_1 car $P_0 > P_1$, T_0 commence son exécution.

A l'instant t_5 *:* T_0 ne peut prendre R_0 car la ressource n'est pas libre. T_3 reprend son exécution mais avec la priorité P_0 de T_0($\boldsymbol{P_3 = P_0}$) durant 3 cycles.

A l'instant t_8 *:* T_3 libère la ressource R_0, reprend sa priorité initiale P_3 et est préemptée par T_0. T_0 peut prendre la ressource libre R_0 et entre en section critique.

A l'instant t_9 *:* T_0 ne peut prendre R_1 car la ressource n'est pas libre. T_1 reprend son exécution mais avec la priorité P_0 de T_0($\boldsymbol{P_1 = P_0}$) durant un cycle.

A l'instant t_{10} *:* T_1 libère la ressource R_1, reprend sa priorité initiale P_1 et est préemptée par T_0. T_0 peut prendre la ressource libre R_1 et entre en section critique.

A l'instant t_{11} *:* T_0 libère R_1 et continue son exécution dans aucune section critique durant un cycle.

A l'instant t_{12} *:* T_0 termine son exécution. T1 s'exécute pendant un cycle dans aucune section critique.

A l'instant t_{13} *:* T_1 termine son exécution. T2 s'exécute pendant deux cycles dans aucune section critique.

A l'instant t_{15} *:* T_2 termine son exécution. T_3 reprend la sienne pendant un cycle dans aucune section critique.

A l'instant t_{16} *:* T_3 termine son exécution.

Conclusion : ***temps d'exécution de*** $T_0 = t_{12} - t_4 = 8$ ***cycles.***

IV.6.3 L'exemple avec OCPP (Original Ceiling Priority Protocol)

Dans les deux sections IV.6.3 et IV.6.4 nous rappelons que le *plafond de priorité* courant du système (current priority ceiling), noté π', est égal au maximum des plafonds de priorité des ressources utilisées, ou Ω si aucune ressource n'est utilisée (Ω est une priorité plus basse que n'importe qu'elle priorité) [Ela,02].

$$\pi(R_0) = P_0$$

$$\pi(R_1) = P_1$$

La figure IV.6 montre l'ordre d'exécution des 4 tâches avec le protocole d'ordonnancement OCPP:

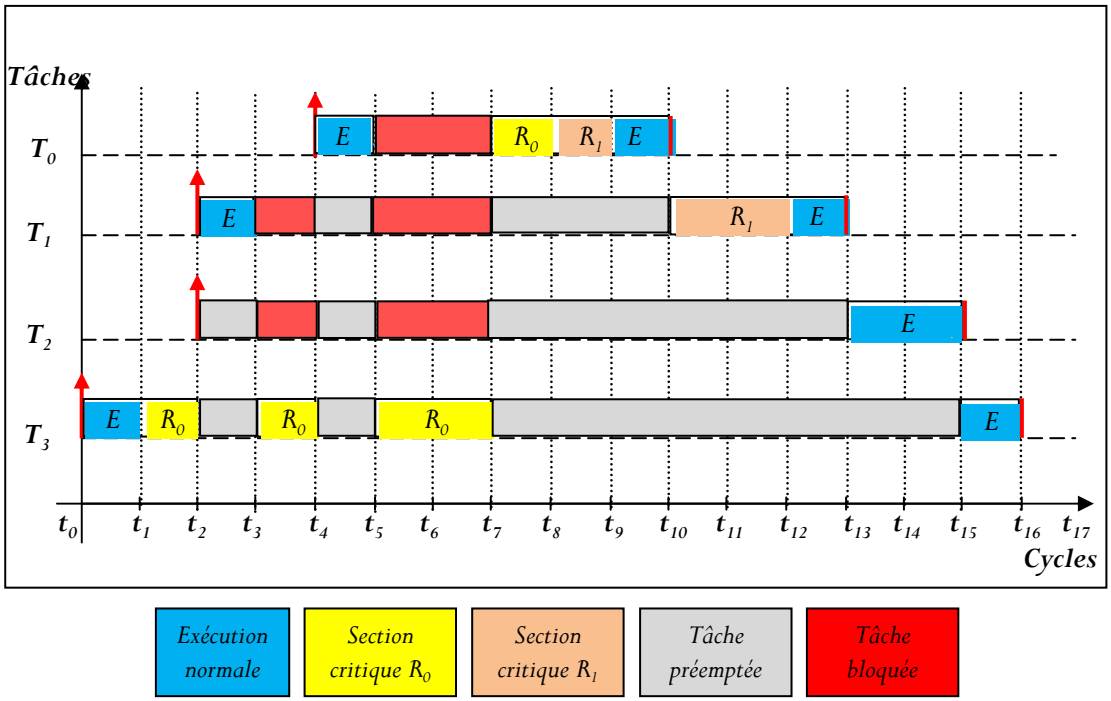

Figure IV.6 : L'ordonnancement des 4 tâches avec OCPP [Dou,Gho,10]

A l'instant t_0 : T_3 est activée et commence son exécution.

A l'instant t_1 : T_3 peut prendre R_0 car $P_3 > \pi' = \Omega$ et entre en section critique.

A l'instant t_2 : T_1 est réveillée, préempte T_3 et commence son exécution car $P_1 > P_3$.

A l'instant t_3 : T_1 ne peut prendre R_1 car P_1 n'est pas $> \pi' = \pi(R_0)$, T_3 hérite de la priorité P_1 de T_1 et reprend son exécution.

A l'instant t_4 : T_0 est réveillée et préempte T_3 car $P_0 > P_3$, T_0 commence son exécution.

A l'instant t_5 : T_0 ne peut prendre R_0 et se retrouve bloquée car P_0 n'est pas $> \pi' = \pi(R_0)$, T_3 reprend son exécution avec la priorité P_0 durant deux cycles consécutifs.

A l'instant t_7 : T_3 libère R_0 et reprend sa priorité P_3. T_3 est alors préemptée par T_0 et T_0 prend R_0 car $P_0 > \pi' = \Omega$.

A l'instant t_8 : T_0 libère R_0 et peut prendre R_1 car $P_0 > \pi' = \Omega$.

A l'instant t_9 : T_0 libère R_1 et s'exécute pendant un cycle dans aucune section critique.

A l'instant t_{10} : T_0 termine son exécution. T_1 reprend la sienne et peut prendre R_1 pendant deux cycles consécutifs car $P_1 > \pi' = \Omega$.

A l'instant t_{12} : T_1 libère R_1 et s'exécute pendant un cycle dans aucune section critique.

A l'instant t_{13} : T_1 termine son exécution. T2 peut commencer la sienne, elle s'exécute pendant deux cycles dans aucune section critique.

A l'instant t_{15} : T_2 termine son exécution. T_3 reprend la sienne pendant un cycle dans aucune section critique.

A l'instant t_{16} : T_3 termine son exécution.

Conclusion : *temps d'exécution de* $T_0 = t_{10} - t_4 = 6$ *cycles.*

IV.6.4 L'exemple avec ICPP (Immediate Ceiling Priority Protocol)

$$\pi(R_0) = P_0$$

$$\pi(R_1) = P_1$$

La figure IV.7 montre l'ordre d'exécution des 4 tâches avec le protocole d'ordonnancement ICPP:

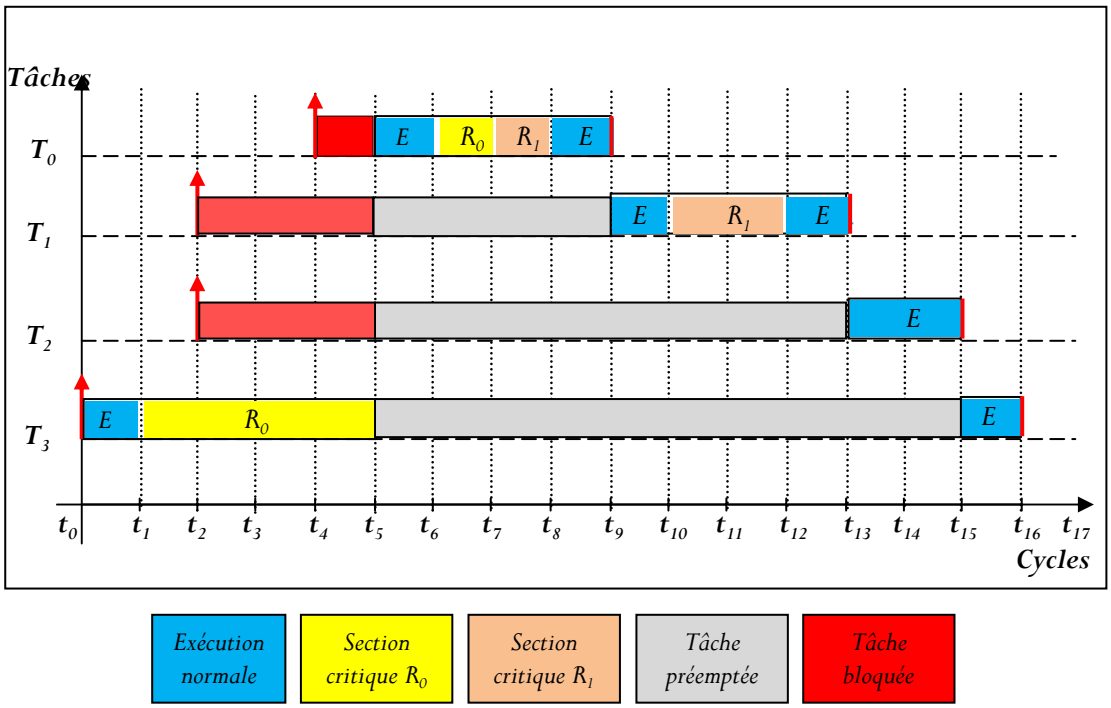

Figure IV.7 : L'ordonnancement des 4 tâches avec ICPP [Dou,Gho,10]

A l'instant t_0 : T_3 est activée et commence son exécution.

A l'instant t_1 : T_3 peut prendre R_0 car $P_3 > \pi' = \Omega$ et entre en section critique avec $P_3 = \pi(R_0) = P_0$.

A l'instant t_2 : T_1 est réveillée, T_3 peut continuer à s'exécuter car $P_3 = P_0 > P_1$ de T_1.

A l'instant t_4 : T_0 est réveillée mais T_3 peut continuer à s'exécuter car $P_3 = P_0$.

A l'instant t_5 : T_3 libère R_0 et reprend sa priorité initiale P_3. T_3 est alors préemptée par T_0 qui commence son exécution.

A l'instant t_6 : T_0 prend R_0 car $P_0 > \pi' = \Omega$.

A l'instant t_7 : T_0 libère R_0 et peut prendre R_1 car $P_0 > \pi' = \Omega$.

A l'instant t_8 : T_0 libère R_1 et s'exécute pendant un cycle dans aucune section critique.

A l'instant t_9 : T_0 termine son exécution. T_1 commence son exécution normale.

A l'instant t_{10}: T_1 peut prendre R_1 pendant deux cycles consécutifs car $P_1 > \pi' = \Omega$.

A l'instant t_{12} : T_1 libère R_1 et s'exécute pendant un cycle dans aucune section critique.

A l'instant t_{13} : T_1 termine son exécution. T2 peut commencer la sienne, elle s'exécute pendant deux cycles dans aucune section critique.

A l'instant t_{15} : T_2 termine son exécution. T_3 reprend la sienne pendant un cycle dans aucune section critique.

A l'instant t_{16} : T_3 termine son exécution.

Conclusion : ***temps d'exécution de*** $T_0 = t_9 - t_4 = 5$ ***cycles***

IV.6.5 L'exemple avec SRP (Stack Resource Policy)

$$\pi(R_0) = P'_0$$

$$\pi(R_1) = P'_1$$

La figure IV.8 montre l'ordre d'exécution des 4 tâches avec le protocole d'ordonnancement SRP:

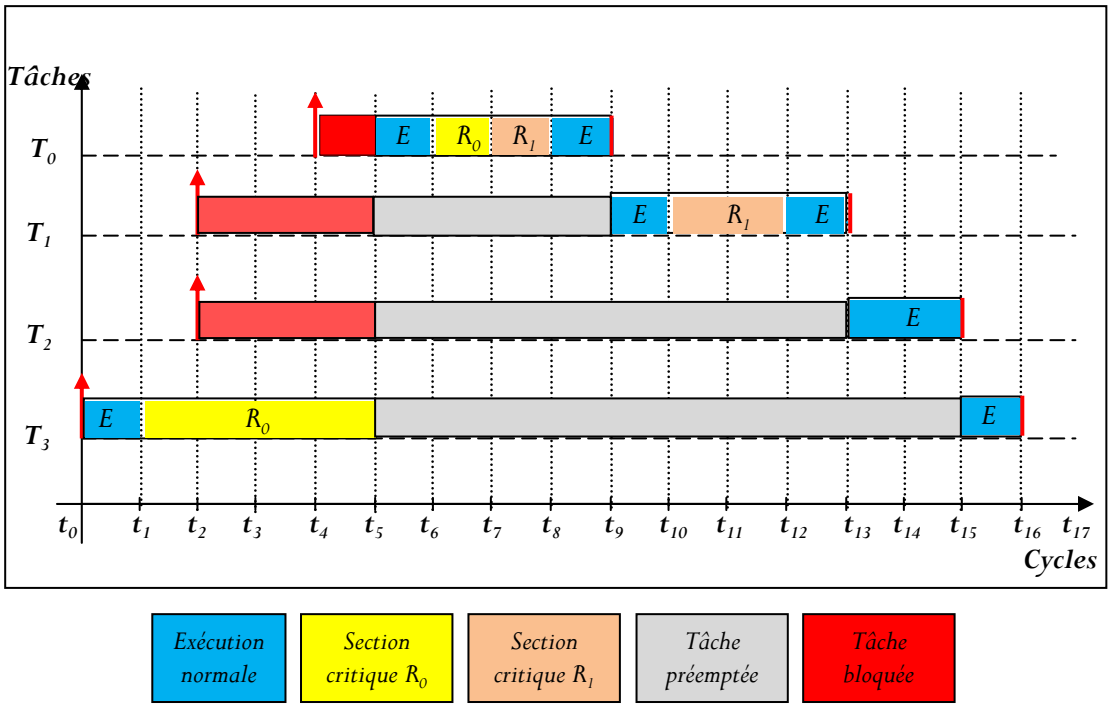

Figure IV.8 : L'ordonnancement des 4 tâches avec SRP [Dou,Gho,10]

A l'instant t_0 : T_3 est activée et commence son exécution.

A l'instant t_1 : T_3 peut prendre R_0 et entre en section critique, le plafond de préemption du système devient $\pi' = \pi(R_0) = P'_0$.

A l'instant t_2 : T_1 et T_2 sont activées, mais ($P'_1 < P'_3$) et ($P'_2 < P'_3$) donc ni T_1 ni T_2 ne peut préempter T_3.

A l'instant t_4 : T_0 est réveillée, mais T_3 peut continuer à s'exécuter car $P'_3 > P'_0$.

A l'instant t_5 : T_3 libère R_0; T_0, T_1 et T_2 sont les tâches prêtes à s'exécuter. Comme $P_0 > P_1 > P_2$ alors T_0 commence son exécution.

A l'instant t_6 : T_0 prend R_0, on a alors $\pi' = \pi(R_0) = P'_0$. Le test de préemption sur T_1 et T_2 n'est pas satisfaisant puisque ($P_1 < P_0$) et ($P_2 < P_0$), donc T_0 continue son exécution.

A l'instant t_7 : T_0 libère R_0 et prend R_1, on a alors $\pi' = \pi(R_1) = P'_1$. Le test de préemption sur T_1 et T_2 n'est pas satisfaisant puisque ($P_1 < P_0$) et ($P_2 < P_0$), donc T_0 continue son exécution.

A l'instant t_8 : T_0 libère R_1 et continue son exécution dans aucune section critique durant une cycle.

A l'instant t_9 : T_0 termine son exécution, T_1 peut alors commencer la sienne.

A l'instant t_{10} : T_1 prend R_1 durant deux cycles de suites, on a alors $\pi' = \pi(R_1) = P'_1$. Le test de préemption sur T_2 n'est pas satisfaisant puisque $P_2 < P_1$, donc T_1 continue son exécution.

A l'instant t_{12}: T_1 libère R_1 et continue son exécution dans aucune section critique durant un cycle.

A l'instant t_{13} : T_1 termine son exécution. T2 peut commencer la sienne, elle s'exécute pendant deux cycles dans aucune section critique.

A l'instant t_{15} : T_2 termine son exécution, T_3 reprend la sienne durant un cycle dans aucune section critique.

A l'instant t_{16} : T_3 termine son exécution.

***Conclusion :** temps d'exécution de $T_0 = t_9 - t_4 = 5$ cycles.*

IV.6.6 L'exemple avec CDP (Ceiling Dynamic Protocol)

Dans un état initial on a :

- $\pi(R_0) = P_0$
- $\pi(R_1) = P_0$

La figure IV.9 montre l'ordre d'exécution des 4 tâches avec le protocole d'ordonnancement CDP:

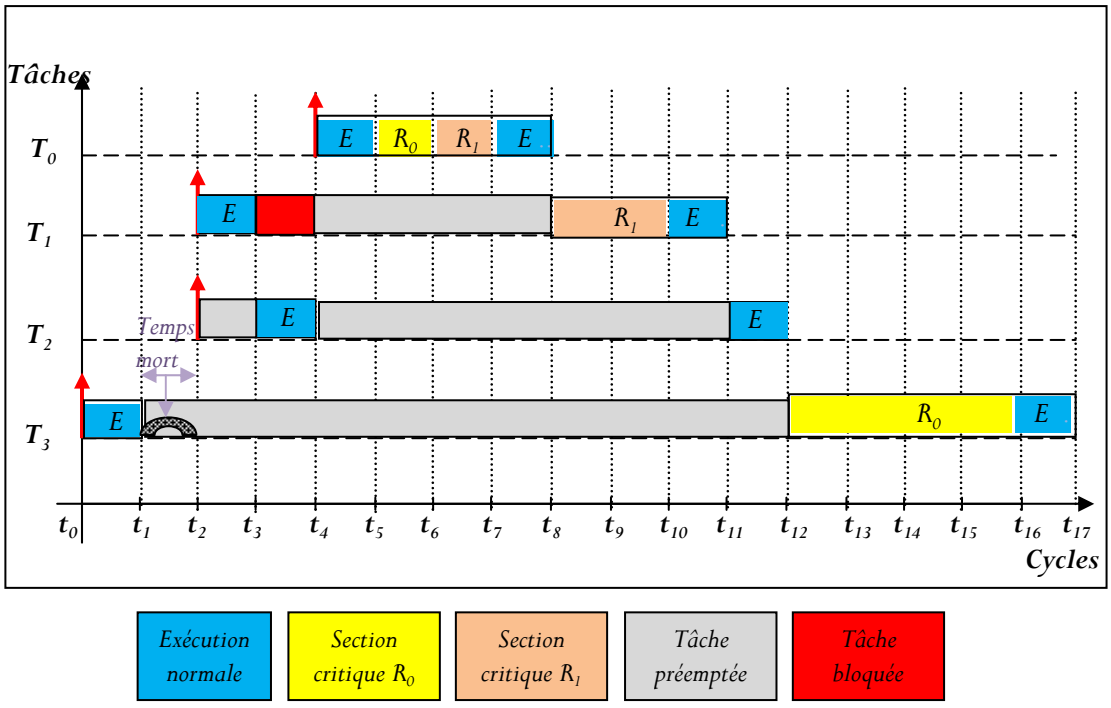

Figure IV.9 : L'ordonnancement des 4 tâches avec CDP [Dou,Gho,10]

A l'instant t_0 *:* T_3 est activée et commence son exécution.

A l'instant t_1 *:* T_3 ne peut pas prendre la ressource R_0 car $(\boldsymbol{Nb_{T_{user(R_0)}} > 1})$

$\boldsymbol{et(P_3 < \pi(R_0))et(D_{R_0} = 4\,périodes > [PDM(T_2 ou\, T_3) - t_1] = 1\,cycle)}$, selon la règle d'ordonnancement (c) le processeur reste inactif pendant un temps mort qui dure $\boldsymbol{D_{TM} = [PDM(T_2) - t_1] = t_2 - t_1 = 1\,cycle}$.

A l'instant t_2 *:* T_1 est activée et commence son exécution et T_2 reste en attente car $P_1 > P_2$.

A l'instant t_3 *:* T_1 ne peut pas prendre la ressource R_1 car : $\left(\boldsymbol{Nb_{T_{user(R_1)}} > 1}\right)$

$\boldsymbol{et(P_1 < \pi(R_1))et(D_{R_1} = 2\,cycles > [PDM(T_0) - t_3] = t_4 - t_3 = 1\,cycle)}$, selon la règle d'ordonnancement (b) on a $T_2.état$=*prête* alors T_2 hérite de $\boldsymbol{\pi'}$, $P_2 = \boldsymbol{\pi'} = P_1$ et commence son exécution.

A l'instant t_4 *:* T_0 est réveillée et préempte T_2 car $P_0 > P_1$, T_0 commence son exécution et T_2 reprend sa priorité initiale P_2.

A l'instant t_5 *:* T_0 continue à s'exécuté et prendre la ressource R_0 et entre en section critique parce que $\boldsymbol{(Nb_{T_{user(R_0)}} > 1)et(P_0 = \pi(R_0))}$.

A l'instant t_6 *:* T_0 quitte R_0 ($\pi(R_0) = \boldsymbol{P_3}$) et prend R_1 pendant un cycle car $P_0 = \pi(R_1)$.

A l'instant t_7 *:* T_0 quitte R_1 ($\pi(R_1) = \boldsymbol{P_1}$) et continue son exécution durant un cycle dans aucune section critique.

A l'instant t_8 *:* T_0 termine son exécution. T_1 reprend la sienne et prend la ressource R_1 pendant deux cycles successifs car $P_1 = \pi(R_1)$.

A l'instant t_{10} *:* T_1 quitte R_1 ($\pi(R_1) = \boldsymbol{\varphi}$) et continue son exécution dans aucune section critique durant un cycle.

A l'instant t_{11} *:* T_1 termine son exécution. T_2 reprend la sienne dans aucune section critique durant un cycle car $\boldsymbol{P_2 > P_3}$

A l'instant t_{12} *:* T_2 termine son exécution et T_3 peut reprendre la sienne et prend R_0 durant quatre cycles car $P_3 = \pi(R_0)$.

A l'instant t_{16} : T_3 quitte R_0 ($\pi(R_0)= \varphi$) et continue son exécution dans aucune section critique.

A l'instant t_{17} : T_3 termine son exécution.

Conclusion : temps d'exécution de $T_0 = t_8 - t_4 = 4$ ***cycles.***

On constate que la tâche la plus prioritaire T_0 *est exécutée dans 4 cycles comme indique leur séquence d'exécution. Donc,* T_0 *n'était pas bloqué et termine son exécution dans les normes.*

IV.7 Comparaison des résultats

Le but du tableau IV.7 est de comparer les résultats de l'application des quatre protocoles d'inversion de priorités sur l'exemple proposé.

	PIP	***PCP***	***SRP***	***CDP***
Types de blocage	*héritage de priorité*	*héritage de priorité plafond de priorité*	*Plafond de préemption*	*Section critique longue*
Nombre de blocages de la tâche prioritaire	*2*	*1*	*1*	*0*
Nombre de blocages des tâches moyennes	*1*	*1*	*1*	*1*
Gaspillage de temps CPU	*Non*	*Non*	*Non*	*Oui*
Temps de blocage de la tâche prioritaire	*4 cycles*	*OCPP : 2 cycles ICPP : 1 cycle*	*1 cycle*	*0 cycle*
Interblocage	*Non*	*Non*	*Non*	*Non*
Chaîne de blocage	*Oui*	*Non*	*Non*	*Non*
Priorité	*Fixe*	*Fixe*	*Fixe/dynamique*	*Fixe*
Temps d'exécution de la tâche prioritaire	*8 cycles*	*OCPP : 6 cycles ICPP : 5 cycles*	*5 cycles*	*4 cycles*
Temps total d'exécution	*16 cycles*	*16 cycles*	*16 cycles*	*17 cycles*

Tableau IV.7 : Étude comparative des résultats obtenus par l'exemple proposé [Dou,Gho,10]

On constate que dans l'ordonnancement avec le protocole CDP la tâche la plus prioritaire T_0 est exécutée dans 4 cycles comme indique leur séquence d'exécution. Donc, T_0 n'était pas bloquée et termine son exécution dans les normes. Mais, malgré ce privilège le temps d'exécution total est élevé à cause de gaspillage de temps dû à l'instant t_1 (voir figure IV.9).

IV.8 Conclusion

Dans ce chapitre nous avons introduit un nouveau protocole ***CDP*** « ***C****eiling* ***D****ynamic* ***P****rotocol* » d'allocation de ressources dans l'ordonnancement temps réel pour empêcher le phénomène d'inversion de priorité. D'abord, nous avons présenté en détail notre protocole CDP. Ensuite, nous proposons un exemple où on a appliqué les quatre protocoles d'inversion de priorité puis nous comparons les résultats obtenus en fonction de ses caractéristiques de fonctionnement. Les résultats obtenus par le CDP sont satisfaisant par rapport au paramètre temps d'exécution et le nombre de blocage de la tâche prioritaire ainsi que le respect de l'ordre d'exécution selon l'algorithme RM, par contre l'optimalité reste toujours une solution approchée. Le chapitre suivant sera consacré à la modélisation et la réalisation du prototypa du protocole CDP.

Chapitre V :
Conception du système

V.1 Introduction

Après une présentation détaillée de la problématique (*phénomène d'inversion de priorité*) ainsi que la solution que nous avons proposé (*le protocole CDP*) pour empêcher ce problème, nous abordons au cours de ce chapitre l'architecture générale de l'application, le modèle de tâche utilisé ainsi que la modélisation de l'exécution des tâches de priorité moyenne et de l'ordonnancement des tâches. Ensuite, nous définissons les différents états-transitions qui peuvent subir une tâche lors de son exécution avec ce protocole.

V.2 La modélisation du système

L'idée de base est qu'en disposant d'un ensemble de tâches temps réels à s'exécuter, ces tâches partagent des ressources en communs. Nous avons utilisé un ordonnanceur temps réel préemptif ainsi que le protocole CDP pour allouer les ressources aux tâches et empêcher le phénomène d'inversion de priorité.

La figure V.1 présente l'architecture générale du système:

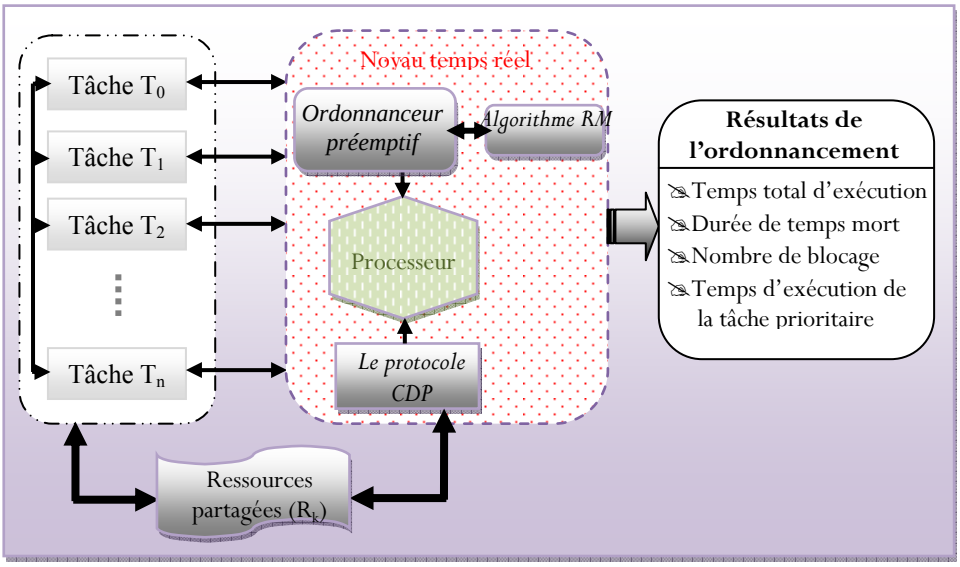

Figure V.1 : Architecture générale du système

Pour pouvoir implémenter le protocole CDP nous avons adopté une architecture générale du système monoprocesseur-multitâche.

D'abord, Le *noyau temps réel* est la partie fondamentale du système. Il gère les ressources, permet aux différents tâches de communiquer entre eux, assure l'ordonnancement et le changement de contexte et essentiellement garantir les temps d'exécution des tâches. Il intègre : un ordonnanceur préemptif, un algorithme RM, un processeur et le protocole CDP.

- L'algorithme *Rate Monotonic (RM)* est un algorithme facile à implanter et optimal dans la classe des algorithmes à priorité fixe. La priorité est l'inverse de la périodicité ($P_i = \frac{1}{P_{e_i}}$) et la tâche ayant la période la plus petite, dans un jeu de tâches, est la plus prioritaire.
- L'ordonnanceur adopte le principe de cet algorithme « *HPF : Highest Priority First* » pour sélectionner la tâche possédant la plus haute priorité. Une condition nécessaire et suffisante d'ordonnançabilité ($\forall\, i\,, 1 \leq i \leq n,\ R_i \leq P_{e_i}$) est appliqué sur l'ensemble des tâches pour reconnaître si tel ensemble est ordonnançable ou non par RM.
- Si la condition d'ordonnançabilité est satisfaite alors le processeur exécute la tâche sélectionnée par l'ordonnanceur et utilise l'ensemble des règles d'ordonnancement

prédéfinies par le CDP pour allouer aux tâches les ressources sollicitées à leurs exécutions.

Finalement, le système diffuse les résultats obtenus inclus le type de blocage s'il y on a, la durée de gaspillage de temps CPU, le temps de blocage des tâches moyenne, le temps d'exécution de chaque tâche ainsi que le temps total d'exécution.

V.2.1 Organigramme du module CDP

La figure V.2 détaille le fonctionnement du protocole CDP proposé via un organigramme :

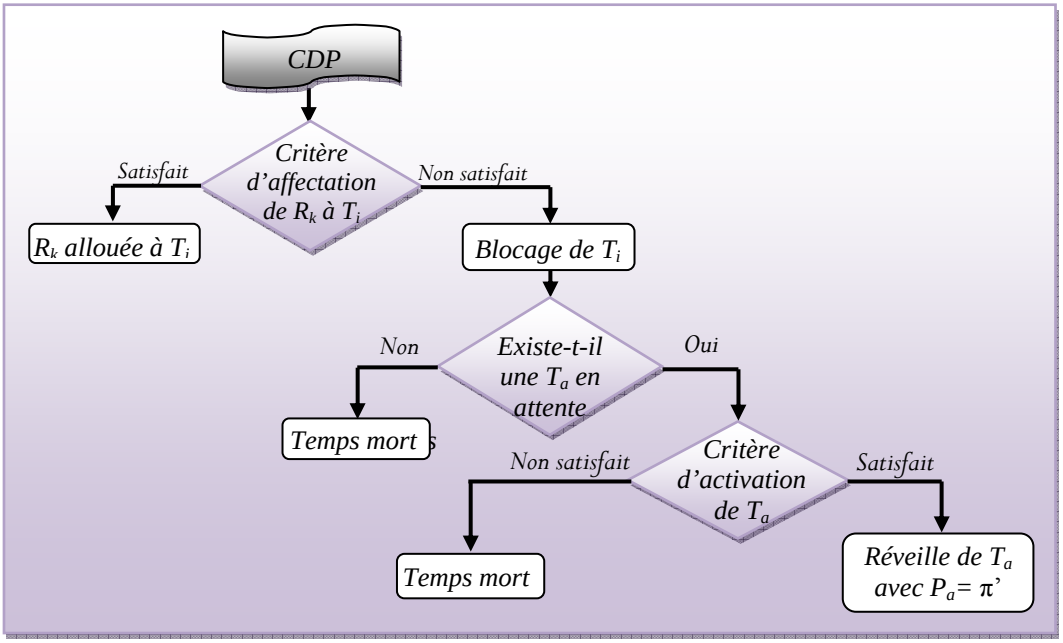

Figure V.2 : L'organigramme du CDP

Une fois la tâche T_i demande une ressource R_k durant son exécution, le protocole d'allocation de ressource CDP valide un critère d'allocation, si celui-ci est satisfait alors la ressource R_k sera allouée à la tâche T_i sinon elle sera bloquée.

Si la tâche Ti a été bloquée, le protocole CDP cherche s'il y a une tâche dans l'état prête, préemptée ou bloquée qui peut être déclenchée son exécution sinon un gaspillage de temps CPU (temps mort) forcé sera produit par la non prise de la ressource R_k.

V.2.2 L'algorithme du CDP

Algorithme *CDP ;*
{Pour allouer R_k *à* T_i*}*
Debut
$\pi' = \varphi$
$nb = nbr(R_k)$
Pour *k allant de 1 à nb* ***faire*** $\pi(R_k) = \boldsymbol{max}_{T_i nécessite(R_k)}\ \boldsymbol{P_i}$
Si $Nb_{T_{user(R_k)}} = 1$ ***alors*** $T_i.état := elue;$
Si $(\boldsymbol{Nb_{T_{user(R_k)}}} > 1)$ *et* $(\boldsymbol{P_i} = \boldsymbol{\pi(R_k)})$ ***alors*** $T_i.état := elue;$
Si $(\boldsymbol{Nb_{T_{user(R_k)}}} > 1)$ *et* $(\boldsymbol{P_i} < \pi(\boldsymbol{R_k}))$ ***alors***
Pour $P_j > P_i$ ***Si*** $D_{R_k} \leq \left[min\left(PDM(T_j)\right) - t\right]$ ***alors*** $T_i.état := elue;$
Sinon
Debut
$T_i.etat = bloquée$;
Avec $P_a = \pi'$
Si $T_a.etat = prête$ ***alors*** T_a *commence son exécution*
Si $T_a.etat = préemptée$ ***alors*** T_a *reprend son exécution*
Si $T_a.etat = bloquée\ sur\ R_{k'}$ *(k' peut être égal à k)* ***alors*** T_a *peut prendre* R_k
si $D_{R_{k'}} \leq [min\ (PDM(T_j)) - t]$
Fin
Si *aucune tâche n'est exécutée* ***alors*** *gaspillage de temps CPU qui dure :*
$D_{TM} = \left[min\left(PDM(T_j)\right) - t\right]/cycles$
Fin.

V.2.3 Le modèle de tâches utilisé

Dans notre travail nous avons choisi un modèle de tâche périodique qui représente les tâches activées à des intervalles réguliers. Ce modèle s'appuie sur un modèle introduit initialement par Liu et al dans [Liu&Lay,73].

Soit une tâche périodique *Ti* modélisée par les paramètres suivants :

$$T_i < PDM_i, C_i, CR_i, P_{ei}, Seq_i >$$

Où :

PDM_i : L'instant d'arrivé ou la date d'activation au plus tôt de la tâche *Ti*.

C_i : la charge processeur. Elle est égale à la période $C_i = P_{ei}$, c'est-à-dire que le processeur sera chargé d'exécuté la tâche durant touts les cycles de la période.

CR_i : Le nombre de cycles requis à l'exécution de la tâche T_i ($CR_i \leq P_{ei}$).

P_{ei} : La période de la tâche *Ti* qui est la durée pendant laquelle la tâche doit achever son exécution ($P_{ei} = D_i$). La *durée d'exécution* est le temps nécessaire à la tâche pour réaliser le travail qu'elle doit effectuer pendant une période.

Seq_i : La séquence à exécuté par la tâche T_i (portions de codes).

La figure V.3 représente l'exécution de la tâche *Ti* pour quatre occurrences dans un diagramme de Gantt. La zone d'exécution de la tâche se situe donc entre les dates (*Ai* et *Fi*). L'échéance d'une tâche est égale à sa période (D*i* = *Pei),* alors la tâche est à échéance sur requête. Nous pouvons noter que la tâche *Ti* est préemptée trois fois lors de son exécution et son exécution s'effectue alors en quatre fois. Cependant, lors de la quatrième occurrence, la tâche s'exécute de façon complète jusqu'à sa terminaison.

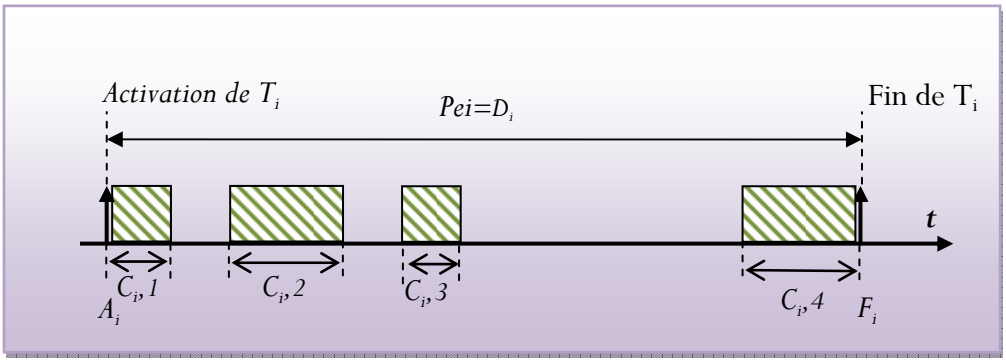

Figure V.3 : Représentation de l'exécution d'une tâche T_i avec CDP

- Le temps d'exécution de la tâche T_i ($TempsExe_i$) est égale au nombre de cycles dans laquelle le processeur est chargé à l'exécution de la tâche T_i. De manière formelle :

$$TempsExe_i = \sum C_{i,k} \qquad (V.1)$$

Où : k : c'est le nombre d'occurrence pendant laquelle le processeur est chargé d'exécutée la tâche T_i.

- Le temps de blocage B_i de la tâche T_i est le nombre de cycles durant laquelle la tâche Ti a été bloquée sur les ressources partagées au profit des autres tâches de moindre priorité. De manière formelle :

$$P_j > P_i : B_i = B_i + [\min_{j \in hp(i)} \left(PDM(T_j)\right) - t] \qquad (V.2)$$

Où $hp(i)$ est l'ensemble des tâches de plus forte priorité que T_i.

- Le temps de réponse R_i de la tâche T_i est égale au temps d'exécution de la tâche T_i plus le temps d'attente PB_i durant laquelle la tâche T_i a été préemptée ou bloquée par une autre tâche. De manière formelle :

$$R_i = TempsExe_i + PBi \qquad (V.3)$$

Et :

$$PBi = B_i + le\ temps\ de\ préemption\ de\ T_i \qquad (V.4)$$

On peut aussi calculer R_i par l'écart entre l'instant de terminaison et l'instant d'arrivé de la tâche T_i. De manière formelle :

$$R_i = F_i - PDM_i \qquad (V.5)$$

V.3 Le partage des ressources entre les tâches

Les ressources partagées sont dites critiques si elles ne peuvent être utilisées simultanément que par une seule tâche. Une tâche T_i de durée d'exécution CRi ($CR_i \leq R_i$) qui accède à des ressources critiques R1, R2,…, Rn possède dans son code des sections, appelées sections critiques, dans lesquelles elle accède à ces ressources. Pour assurer l'accès en exclusion mutuelle à ces ressources par les différentes tâches, le protocole CDP est utilisé pour une allocation sans conflit des ressources aux tâches.

La figure V.4 représente l'exécution d'une tâche T_i qui utilise la ressource R_k après un délai α dès son activation.

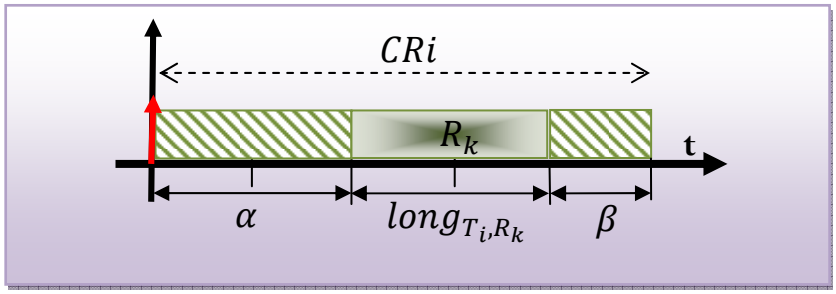

Figure V.4 : Représentation temporelle d'une tâche T_i utilisant une ressource R_k

Sachant que :

- α représente le temps d'exécution avant la prise de la ressource R_k ;

- $long_{T_i,R_k}$ est la durée de la prise de la ressource R_k par la tâche T_i.

- β est le temps après la prise de la ressource R_k.

Le nombre de cycles requis à l'exécution de la tâche T_i est :

$$CR_i = \alpha + long_{T_i,R_k} + \beta \qquad (V.6)$$

V.4 L'ordonnancement des tâches avec CDP

La figure V.5 illustre simplement l'ordonnancement des 3 tâches périodiques (T_0, T_1 et T_2) triées en ordre de priorité et utilisant entre eux deux ressources partagées. T_0 est la tâche la plus prioritaire et T_2 la moins prioritaire. T_0 utilise R_0 puis R_1, T_1 utilise R_2 et T_2 utilise R_2 puis R_1. La figure montre comment sont ordonnancées les tâches mais ne précise pas comment partagent les ressources communes. Nous avons utilisé dans ce schéma les notations suivantes : **L** : Lancer l'exécution. **S** : Suspendre l'exécution.

C : *Continuer l'exécution* **T** : *Terminer l'exécution*.

Les numéros de 1 à 4 représentent les différentes transitions possibles.

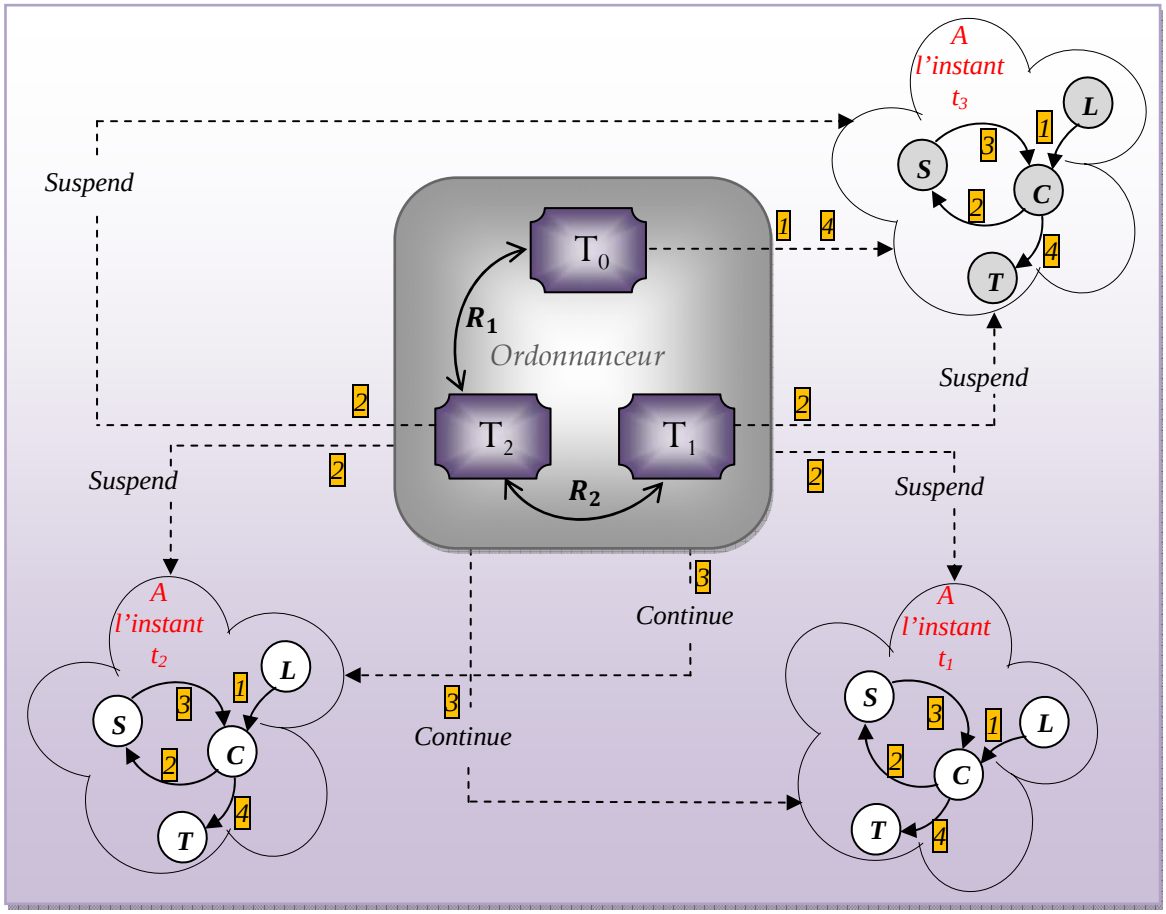

Figure V.5 : L'ordonnancement des tâches

L'ordonnanceur sélectionne la tâche à lancer et puis peut continuer ou suspendre une tâche grâce aux signaux « *continue* » et « *suspend* » représentés par des flèches en pointillés. Lorsqu'une tâche est suspendue à instant t, l'ordonnanceur sélectionne la tâche suivante à continuer. La tâche prioritaire continue son exécution sans interruption jusqu'à sa terminaison (c'est le cas à l'instant t_3 dans la figure V.5).

V.5 Les états-transitions d'une tâche T_i avec CDP

La figure V.6 montre les différents états-transitions possibles d'une tâche Ti lors de son exécution dans un environnement temps réel qui renferme le protocole CDP proposé.

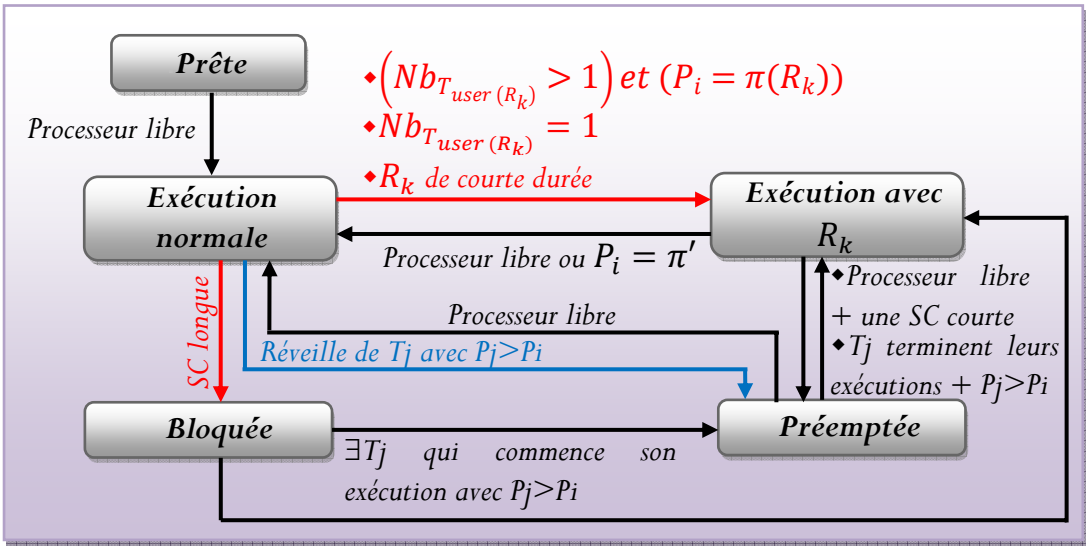

Figure V.6 : Etats et transitions d'une tâche Ti avec le CDP

Après le réveil, la tâche Ti se trouve dans l'état prête, où elle se retrouve en compétition avec les autres tâches disposées à être exécutées excepté la tâche de plus haute priorité qui sera passé après son activation de l'état élue à l'état terminée sans passée par les deux états ''*préemptée ou bloquée*''. L'ordonnanceur a alors la charge de choisir les tâches à activer. De l'état prête, le système d'exploitation peut ensuite la faire passer dans l'état élue.

-Depuis l'état élue (Exécution normale) : la tâche T_i peut se retrouver

- Dans l'état bloquée, lors de l'attente d'une ressource partagée à accès exclusive.
- Dans l'état préemptée, lors du réveille d'une autre tâche prioritaire.

- Dans l'état élue (Exécution avec R_k), lors de la satisfaction de l'un des règles d'ordonnancement associées au CDP pour l'allocation de R_k.

-Depuis l'état élue (Exécution avec R_k) : la tâche T_i peut se retrouver

- Dans l'état préemptée, lors du réveille d'une tâche prioritaire qui ne besoin pas ce ressource durant son exécution.
- Dans l'état élue (Exécution normal), lorsque T_i est la tâche prioritaire courante ou le processeur est libre.

-Depuis l'état préemptée : la tâche T_i peut se retrouver

- Dans l'état élue (Exécution avec R_k), lors de la satisfaction de l'un des règles d'ordonnancement associées au CDP pour l'allocation de R_k.
- Dans l'état élue (Exécution normal), lorsque toutes les tâches prioritaire à T_i sont terminées son exécution ou le processeur est libre.

-Depuis l'état bloquée : la tâche T_i peut se retrouver

- Dans l'état préemptée, lors du réveille d'une autre tâche prioritaire.
- Dans l'état élue (Exécution avec R_k), lors de la satisfaction de l'un des règles d'ordonnancement associées au CDP pour l'allocation de R_k.

V.6 Conclusion

Dans ce chapitre nous avons présenté la conception et spécification de notre système ainsi que la modélisation de la tâche, de l'ordonnancement des tâches et les états-transitions de chaque tâche. Le chapitre suivant sera consacré à la réalisation du prototype et la présentation des résultats obtenus.

Chapitre VI :

Implémentation et résultats expérimentaux

VI.1 Introduction

Avec une approche totalement différente de l'approche analytique, la simulation peut permettre de valider les résultats obtenus par l'une et l'autre des méthodes [Tou,91]. Après la modélisation présentée dans le chapitre précédant, nous allons maintenant détailler le simulateur que nous avons développé pour ordonnancer un ensemble de tâches temps réels toute en empêchant le phénomène d'inversion de priorité à l'aide du protocole CDP.

VI.2 L'outil de développement

Dans la réalisation de notre système, nous avons choisis l'outil de développement *Delphi6*. Delphi6 est un environnement de programmation visuel orienté objet, il incarne la suite logique de la famille turbo pascal avec ses nombreuses versions. On a utilisé cet outil car il apporte de nombreux avantages tel que : contient des composants qui permettent de créer des interfaces d'application, dispose d'une bibliothèque qui permet l'utilisation directe des fonctions, des procédures ainsi que les composants visuels réutilisable sans écrire le code source, dispose d'un compilateur rapide et permet des importations des objets des autres applications Windows.

VI.3 Description du simulateur CDP

Le simulateur CDP est écrit en Delphi6 et a été testé sur Windows trust 3.0 de Microsoft Windows XP. Professionnel Version 2002, Service pack 3, v.5512.

Le simulateur couvre trois parties essentielles : *le paramétrage des tâches, l'ordonnancement des tâches* (le test d'ordonnançabilité plus l'algorithme d'ordonnancement RM inclut le protocole CDP) et *l'affichage des résultats numériques de l'ordonnancement*.

La figure VI.1 représente l'interface du simulateur CDP :

Figure VI.1 : Interface du simulateur CDP

VI.3.1 La création des tâches

Le premier paramètre de chaque tâche est '*Name*' qui lui est associé, le second paramètre est la période d'exécution '*Pei*', le troisième paramètre '*CRi*' indique le nombre de cycles requis à l'exécution, le quatrième paramètre '*PDM*' indique l'instant d'arrivée de la tâche, le cinquième paramètre '*sequence*' représente le morceau de code associe à la tâche et le dernier paramètre '*type*' indique si la tâche sollicite des ressources partagées ou non.

La figure VI.2 ci-dessous est une copie d'écran de la fenêtre de paramétrage des tâches :

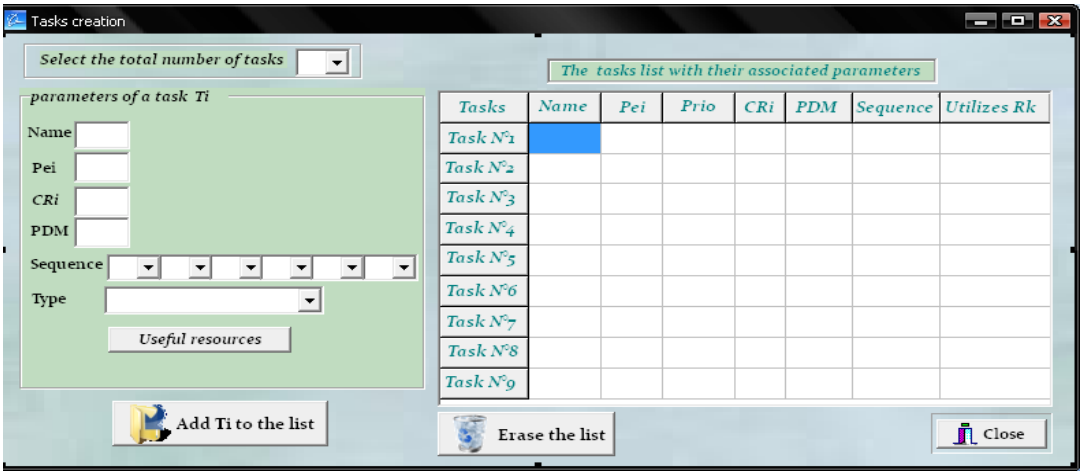

Figure VI.2 : Fenêtre de paramétrage de la tâche T_i

D'abord, nous engageons notre application par la saisie du nombre total de tâches à exécutées. Ensuite, nous saisissons les valeurs des différents paramètres dans les champs qui convient. Puis nous appuyons sur le bouton « *Useful ressources* », la figure VI.3 est affichée pour désigner les ressources sollicitées par la tâche T_i :

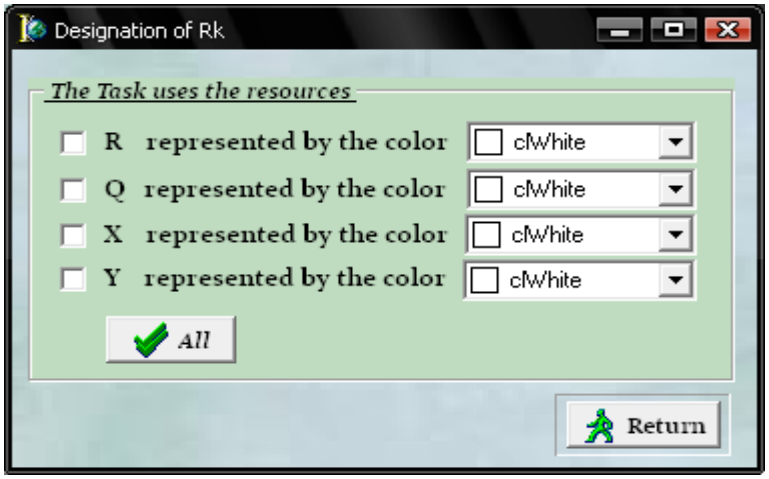

Figure VI.3 : Fenêtre des ressources disponibles

Après avoir sélectionné les ressources sollicitées on clique sur le bouton « *Return* » pour revenir à la fenêtre de paramétrage des taches. Et puis, on clique sur le bouton « *Add T_i to the list* » pour ajouter la tâche ainsi crée dans la liste des tâches à ordonnancer. Et ainsi de suite jusqu'à créer toutes les tâches. Finalement, nous appuyons sur le bouton « *quitter* » pour revenir à la page d'accueil.

Dans l'onglet '***Scheduler***' de la page d'accueil nous sélectionnons la commande '***Ordonnancability test***' pour vérifier si l'ensemble des tâches ainsi créées soient ordonnançable par l'algorithme statique Rate Monotonique ou non (l'algorithme RM affecte les priorités fixes aux différentes tâches). Ainsi, si le test d'ordonnançabilité nécessaire est satisfaisant alors le message de confirmation sur l'ordonnançabilité des tâches sera affiché (Figure VI.4).

Figure VI.4 : Message de satisfaction de l'ordonnançabilité

Nous appuyons sur le bouton '***OK***', la liste des tâches triées par ordre de priorités est affichée.

Dans le même onglet on sélectionne la deuxième commande '***Scheduling launch*** ' pour lancer l'ordonnancement des tâches.

Finalement, le simulateur permet de dégager les résultats suivants :

- ✓ Le nombre de blocage des tâches.
- ✓ Le temps total de blocage.
- ✓ La durée de gaspillage de temps CPU.
- ✓ Les temps de réponse de chaque tâche.
- ✓ Le temps total d'exécution des tâches.

VI.4 Les structures de données

VI.4.1 La structure de la tâche

Chaque tâche est structurée sous forme d'enregistrement (record) qui comporte au totale huit (08) champs. On peut décrire la structure de la tâche comme suit :

```
Type TTache = record
        nom:string;
        pei:integer;
        prio:real;
        CRi:integer;
        PDM:integer;
        Seq,res, etat:string[6];
        end;
```

VI.4.2 La structure de l'ensemble des tâches à ordonnancer

L'ensemble de tâches qui constituent le système est structuré sous forme d'un tableau de taille égale aux nombre totale des tâches à s'ordonnancées. On peut décrire cette structure comme suit :

```
Liste=array[1..nbr_taches]of TTache;
```

Nous avons également utilisé un variable globale de type liste de tâches pour définir le module CDP :

```
Var list:Liste;
```

VI.5 Le programme du CDP

VI.5.1 Les composants du CDP

Pour implémenter le protocole CDP nous définissons les quatre fonctions suivantes :

```
function long_Rk(const i:integer;r:string;lis:liste;posi:integer):integer;
var j,nbrT:integer;
begin
 long_Rk:=0;  nbrT:=0;
 if use_Rk(i,r)=true then
 begin
  j:=posi;
  while lis[i].seq[j]=r do begin
                             long_Rk:=nbrT+1;
                             j:=j+1;
                            nbrT:=nbrT+1;
                           end;
 end;
end;
```

```
function use_Rk (const i:integer;r:string):boolean;
var j:integer;
begin
use_Rk:=false;
j:=1;
while(j<=list[i].CRi)and(list[i].seq[j]<>r)do j:=j+1;
if j<=list[i].CRi then use_Rk:=true;
end;
```

```
function nbrT_userRk(const r:string):integer;
var j,nbrT:integer;
begin
nbrT_userRk:=0;
nbrT:=0;
j:=1;
while j<=nbr_tach do
if list[j].res<>'no Rk' then
                 begin
                  if use_Rk(j,r)=true then
                   begin
                   nbrT_userRk:=nbrT+1;
                   nbrT:=nbrT+1;
                   j:=j+1;
                   end
                  else j:=j+1;
                 end
                 else j:=j+1;
end;
```

```
function P_Rk(const r:string;lis:liste):real;
var j:integer;
begin
P_Rk:=0;
for j:=nbr_tach downto 1 do
if use_Rk(j,r)=true then  P_Rk:=lis[j].prio;
end;
```

VI.5.2 Le module CDP

```
procedure CDP(const tache:integer;ressource:string);
var j,jj,k:integer;task:TTache;
 nbT_useR:integer; P_R:real;
begin
affect:=false;
tach_reveille:=0;
nbT_useR:=nbrT_userRk(ressource);
P_R:=P_Rk(ressource,list);
p:=0;
if nbT_useR=1 then begin
            affect:=true;
            tach_reveille:=tache;
            end;
if (nbT_useR>1)and(list[tache].prio=P_R) then  begin
                                                affect:=true;
                                                tach_reveille:=tache;
                                                end;
if (nbT_useR>1)and(list[tache].prio<P_R) then
begin
for j:=1 to nbr_tach do pdm[j]:=10000;
for j:=1 to tache-1 do pdm[j]:=tab_pdm[j];
pdm_min:=MinIntValue(pdm);
if long_Rk(tache,ressource,list,s[tache])<=(pdm_min-temps) then
 begin
 affect:=true;
 tach_reveille:=tache;
 end
 else
 begin
  affect:=false;
  lis_pre:=list_pret;
  lis_blo:=list_bloque;
  lis_att:=list_att;
while(affect=false)and((vide(lis_pre)=false)or(vide(lis_blo)=false)or(vide(lis_att)=false))
do
```

```
begin
      task:=tt;
      for j:=1 to nbr_tach do
      for jj:=1 to nbr_tach do
       if (lis_pre[j].prio>lis_blo[jj].prio)and(lis_pre[j].prio>task.prio)then
        begin
        task:=lis_pre[j];
        tach_reveille:=j;
        end
       else if lis_blo[jj].prio>task.prio then
           begin
           task:=lis_blo[jj];
           tach_reveille:=jj;
           end;
      if vide(lis_att)=false then
      begin
      j:=1;
       if (task.prio<>0)then
       begin
       while (j<=nbr_tach)and(lis_att[j].prio<task.prio)do j:=j+1;
       if (j<=nbr_tach)then begin
                              task:=lis_att[j];
                              tach_reveille:=j;
                             end
       end
      else
          begin
          task:=lis_att[j];
          j:=j+1;
          while j<=nbr_tach do
          if lis_att[j].prio>task.prio then begin
                                              task:=lis_att[j];
                                              tach_reveille:=j;
                                              end
           else j:=j+1;
          end;
      end;
 {.....}
      if task.etat='pret'then
      begin
       if list[tach_reveille].seq[s[tach_reveille]]<>'E'then
       begin
       {----------}
       for j:=1 to nbr_tach do pdm[j]:=10000;
       for j:=1 to tach_reveille-1 do pdm[j]:=tab_pdm[j];
       pdm_min:=MinIntValue(pdm);
```

```
   iflong_Rk(tach_reveille,list[tach_reveille].seq[s[tach_reveille]],lis_pre,s[tach_reveille])
            <=(pdm_min-temps)then
        begin
         affect:=true;
         nb_blocage:=nb_blocage+1;
         temp_blocage:=temp_blocage+(pdm_min-temps);
         p:=lis_pre[tach_reveille].prio;
        {affectation de p_prim----------------------------------}
         k:=1;
         while(k<=nbr_tach)and(tab_pdm[k]<>10000)do k:=k+1;
         if k<=nbr_tach then  p_prim:=ch[k];
        {p_prim-------------------------------------------------}

         lis_pre[tach_reveille].prio:=p_prim;
         lis_pre[tach_reveille]:=tt;
         lis_att[tach_reveille]:=tt;
         lis_blo[tach_reveille]:=tt;
       end
      end
      else
       begin
        affect:=true;
        nb_blocage:=nb_blocage+1;
        temp_blocage:=temp_blocage+(pdm_min-temps);
        lis_pre[j]:=tt;
        lis_blo[j]:=tt;
        lis_att[j]:=tt;
       end;
     end;
{.....................................................}
      if task.etat='preempt' then
     begin
      if list[tach_reveille].seq[s[tach_reveille]]<>'E' then
      begin
      {------------------------}
      for j:=1 to nbr_tach do pdm[j]:=10000;
      for j:=1 to tach_reveille-1 do pdm[j]:=tab_pdm[j];
      pdm_min:=MinIntValue(pdm);

     iflong_Rk(tach_reveille,list[tach_reveille].seq[s[tach_reveille]],lis_att,s[tach_reveille])
     <=(pdm_min-temps)then
        begin
         affect:=true;
         nb_blocage:=nb_blocage+1;
         temp_blocage:=temp_blocage+(pdm_min-temps);
         p:=lis_att[tach_reveille].prio;
        {affectation de p_prim----------------------------------}
```

```
          k:=1;
          while(k<=nbr_tach)and(tab_pdm[k]<>10000)do k:=k+1;
          if k<=nbr_tach then  p_prim:=ch[k];
          {p_prim----------------------------------------------------}
          lis_att[tach_reveille].prio:=p_prim;
          lis_att[tach_reveille]:=tt;
          lis_pre[tach_reveille]:=tt;
          lis_blo[tach_reveille]:=tt;
        end;
       end
       else
        begin
        affect:=true;
        nb_blocage:=nb_blocage+1;
        temp_blocage:=temp_blocage+(pdm_min-temps);
        lis_pre[j]:=tt;
        lis_blo[j]:=tt;
        lis_att[j]:=tt;
        end;
      end;
 {.....................................................}
      if task.etat='bloquee' then
      begin
       for j:=1 to nbr_tach do pdm[j]:=10000;
       for j:=1 to tach_reveille-1 do pdm[j]:=tab_pdm[j];
       pdm_min:=MinIntValue(pdm);

    iflong_Rk(tach_reveille,list[tach_reveille].seq[s[tach_reveille]],lis_blo,s[tach_reveille])
    <=(pdm_min-temps)then
        begin
         affect:=true;
         nb_blocage:=nb_blocage+1;
         temp_blocage:=temp_blocage+(pdm_min-temps);
         tach_reveille:=j;
         p:=lis_blo[tach_reveille].prio;
        {affectation de p_prim----------------------------------}
         k:=1;
         while(k<=nbr_tach)and(tab_pdm[k]<>10000)do k:=k+1;
         if k<=nbr_tach then  p_prim:=ch[k];
        {p_prim----------------------------------------------------}
         lis_blo[tach_reveille].prio:=p_prim;
         lis_blo[tach_reveille]:=tt;
         lis_pre[tach_reveille]:=tt;
         lis_att[tach_reveille]:=tt;
        end;
       end;
lis_blo[tach_reveille]:=tt;
```

```
lis_pre[tach_reveille]:=tt;
lis_att[tach_reveille]:=tt;
end;
{while affect=false.....}
{le blocage de la tache qui n'a pas prendre une ressource}
      list_bloque[tache]:=list[tache];
      list_bloque[tache].etat:='bloquee';
      list_pret[tache]:=tt;
      list_elu[tache]:=tt;
      list_att[tache]:=tt;

if affect=true then
begin
task.etat:='elue';
list_elu[tach_reveille]:=task;
if task.etat='pret' then begin
                    list_pret[tach_reveille]:=tt;
                    if p<>o then list_elu[tach_reveille].prio:=p_prim;
                    end;
if task.etat='preempt' then begin
                      list_att[tach_reveille]:=tt;
                      if p<>o then list_elu[tach_reveille].prio:=p_prim;
                      end;
if task.etat='bloquee' then begin
                      list_elu[tach_reveille].prio:=p_prim;
                      list_bloque[tach_reveille]:=tt;
                      end;
end;
 end;
end;
end;
```

VI.6 Exemple expérimenté par le simulateur CDP

Nous appliquons l'exemple présenté par le tableau VI.1 pour expérimenter notre simulateur CDP et montrer les résultats numériques obtenus.

Tâches	*Périodes*	*PRi*	*PDM*	*Séquences*
T_0	*40*	*4*	*4*	*ERQE*
T_1	*50*	*4*	*2*	*EQQE*
T_2	*60*	*2*	*2*	*EE*
T_3	*70*	*6*	*0*	*ERRRRE*

Tableau VI.1 : les paramètres des tâches à s'exécutées

Dans un état initial nous avons :

- $\pi(R) = P_0$
- $\pi(Q) = P_0$

Nous commençons par la création des tâches à travers la saisie de ses paramètres dans la fenêtre de paramétrage des tâches suivante (Figure VI.5).

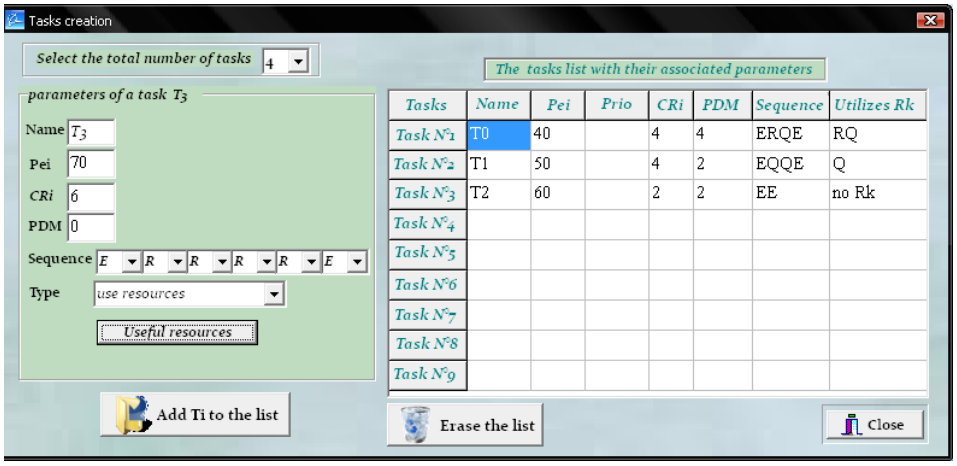

Figure VI.5 : Fenêtre de paramétrage des tâches

Après l'ordonnancement des tâches avec CDP, le simulateur dégage la fenêtre VI.6 des résultats obtenus :

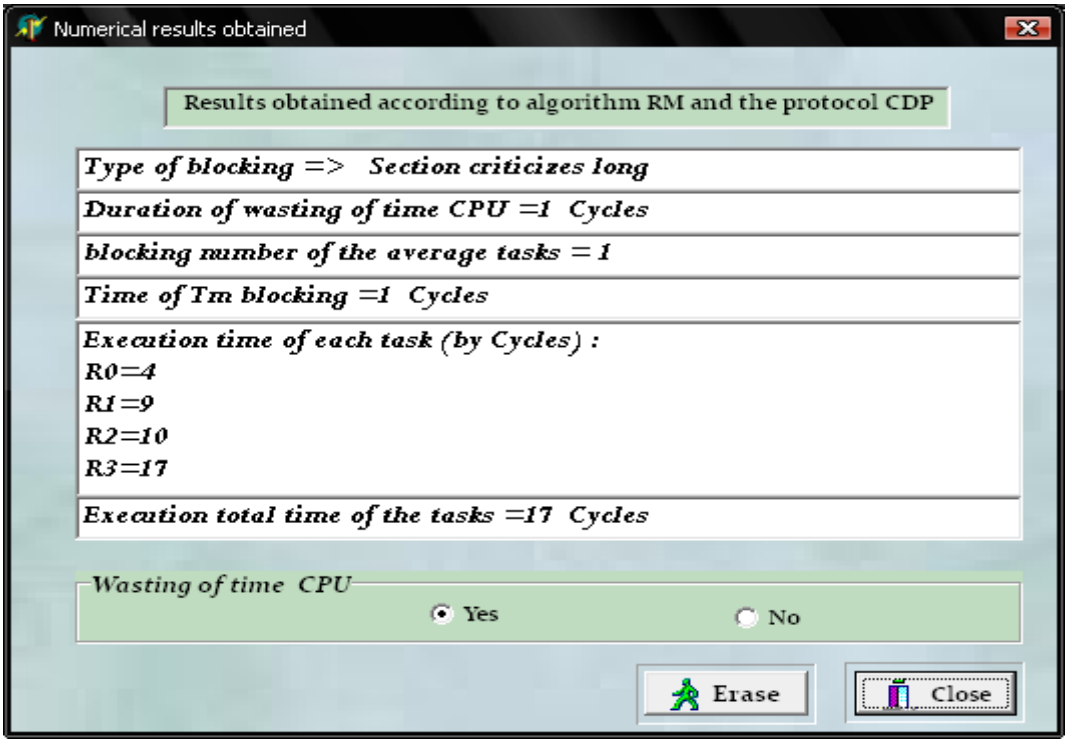

Figure VI.6 : Fenêtre des résultats diffusés par le simulateur CDP

La figure VI.7 illustre via un diagramme de Gantt l'ordre d'exécution des quatre tâches par le simulateur CDP et le temps CPU gaspillé causé par une section critique longue (à l'instant t_1, T_3 veut prendre la ressource R durant quatre cycles successifs).

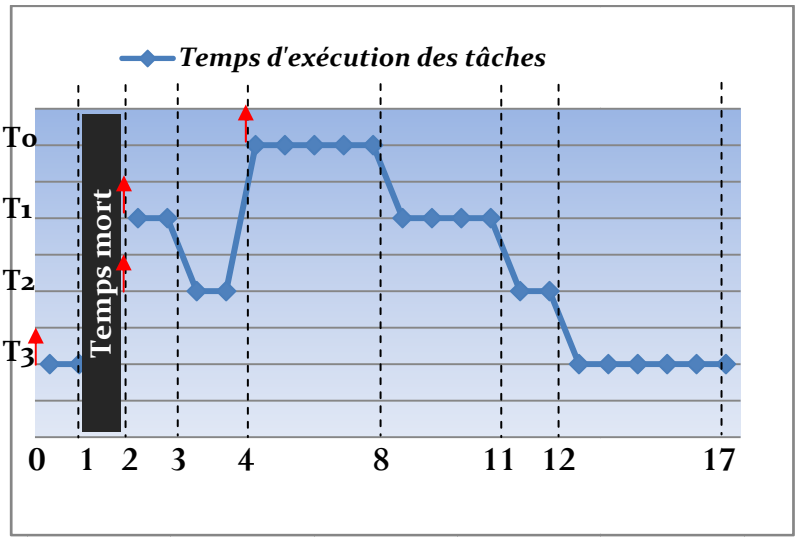

Figure VI.7 : Diagramme d'exécution des tâches selon CDP

VI.7 Comparaison des résultats expérimentaux

Dans cette section nous comparons les résultats obtenus par le simulateur de CDP avec celles obtenus par les protocoles PIP, OCPP, ICPP et SRP cités dans le chapitre III.

Les figures VI.8, VI.9, VI.10, VI.11 représentent le déroulement des quatre tâches selon CDP avec chacun des protocoles d'inversion de priorité :

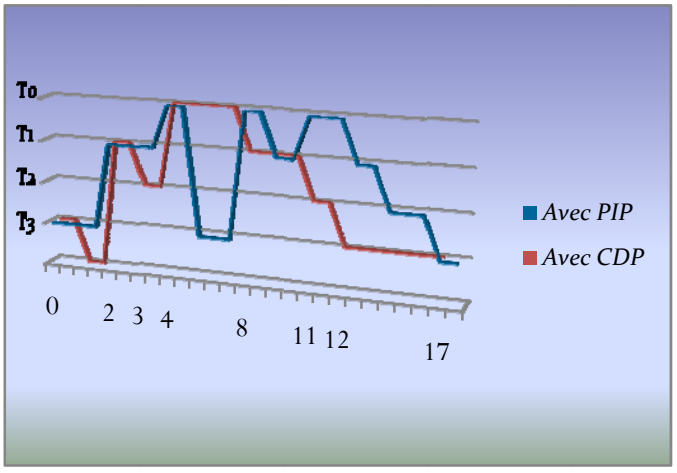

Figure VI.8 : Diagrammes d'exécution des tâches selon PIP et CDP

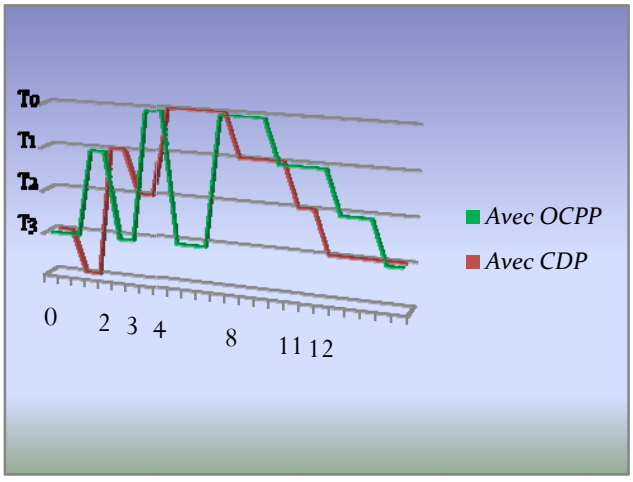

Figure VI.9 : Diagrammes d'exécution des tâches selon OCPP et CDP

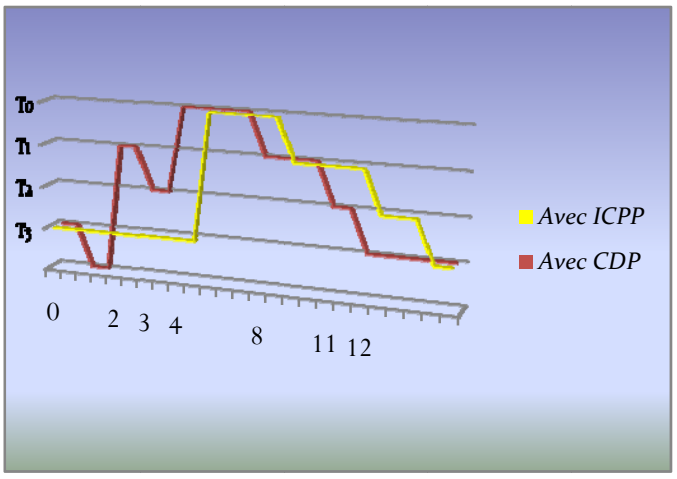

Figure VI.10 : Diagrammes d'exécution des tâches selon ICPP et CDP

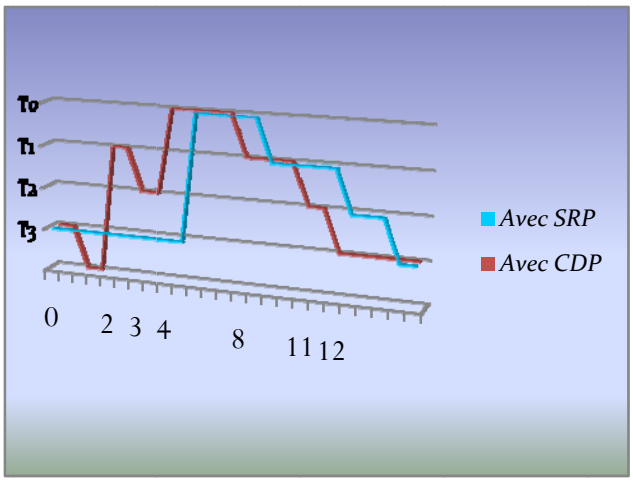

Figure VI.11 : Diagrammes d'exécution des tâches selon SRP et CDP

La figure VI.12 représente le temps de blocage de la tâche prioritaire (par cycle) dû au partage de ressources avec les autres tâches moins prioritaire selon les cinq protocoles d'inversion de priorité.

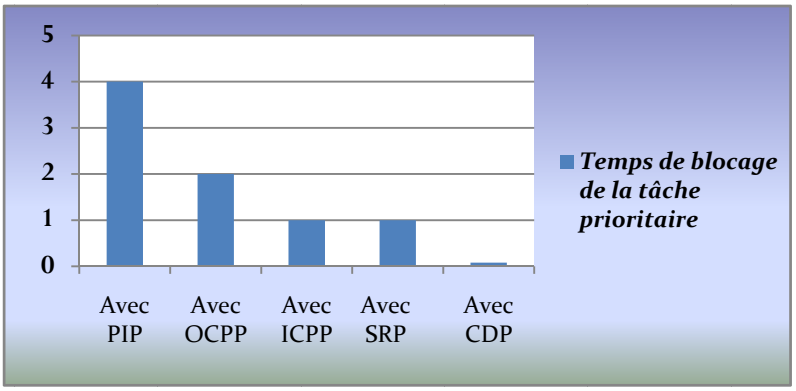

Figure VI.12 : Représentation du temps de blocage de la tâche prioritaire

La figure VI.13 représente le temps d'exécution de la tâche prioritaire (par cycle) avec les cinq protocoles d'inversion de priorité.

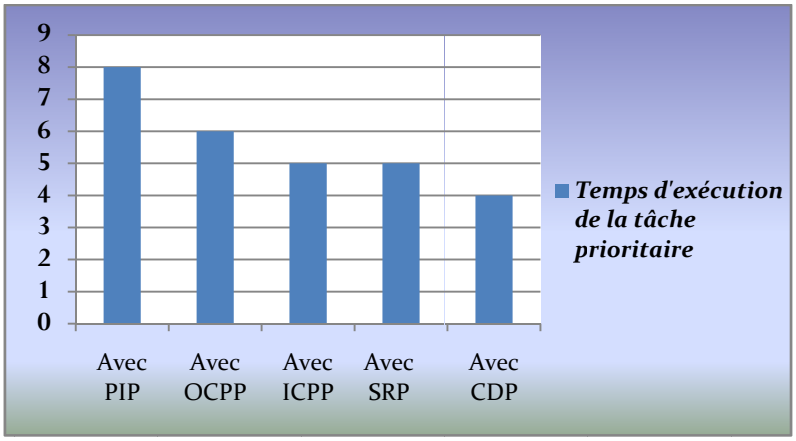

Figure VI.13 : Représentation du temps d'exécution de la tâche prioritaire

La figure VI.14 représente le nombre de changement de contexte effectué par les tâches lors de l'ordonnancent avec les cinq protocoles d'inversion de priorité.

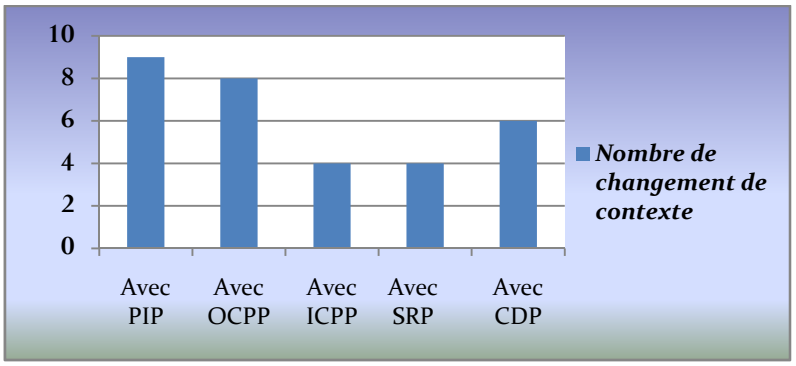

Figure VI.14 : Représentation du nombre de changement de contexte

La figure VI.15 représente la durée d'inversion de priorité (par cycle) provenu durant l'exécution des tâches par les cinq protocoles d'inversion de priorité.

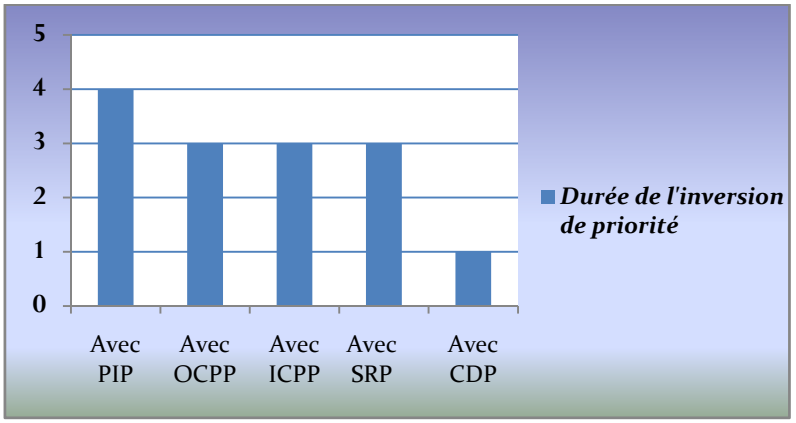

Figure VI.15 : Représentation de la durée de l'inversion de priorité

La figure VI.16 représente le temps d'exécution total des tâches (par cycle) avec les cinq protocoles d'inversion de priorité.

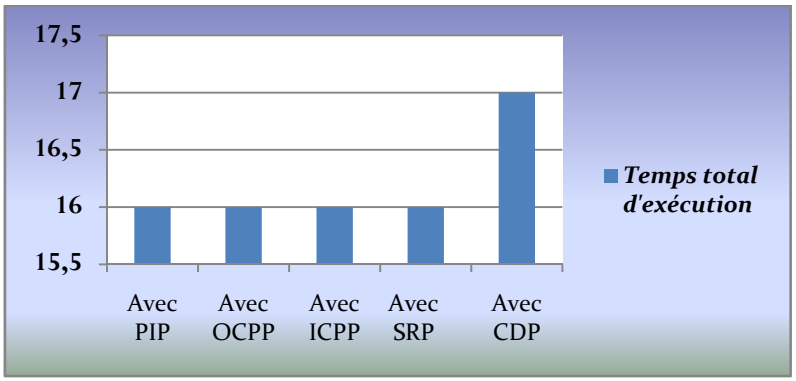

Figure VI.16 : Représentation du temps total d'exécution

Conclusion générale et perspectives

L'ordonnancement dans les systèmes temps réel embarqué consiste à définir la séquence temporelle admissible des tâches à exécuter. Ces tâches collaborent via des ressources partagées entre elles. Un protocole adéquat d'attribution de ces ressources aux différentes tâches du système embarqué ainsi qu'une politique d'ordonnancement sont adoptés pour garantir le respect des contraintes temporelles et de partage de ressource. Un probable problème sensible d'inversion de priorité provient du fait que les applications temps réel notamment à contrainte strict nécessitent des ressources partagées à accès exclusives adoptant un algorithme à priorité fixe pour la gestion des tâches.

Dans ce mémoire de Magistère, nous avons étudié le problème d'inversion de priorité qui se pose dans le domaine de l'ordonnancement monoprocesseur des systèmes temps réel.

Dans une première partie, nous avons abordé les différentes techniques de résolution du problème d'inversion de priorité. Ainsi, les risques d'inter-blocage et de chaînes de blocages avec PIP ont été résolus par PCP. De même, le nombre important de changements de contexte entres les tâches avec PCP a conduit à définir le protocole SRP. Ce dernier, fonctionnant aussi bien avec un ordonnanceur à priorité fixe que dynamique et les blocages se font au niveau de la préemption et non au niveau de l'accès aux ressources. L'inconvénient de cette méthode est qu'une tâche qui n'utilise aucune ressource partagée peut se retrouver bloquée par le test de préemption.

Dans une seconde partie, Pour gérer l'accès aux ressources critiques, éviter l'inversion de priorité et prévenir l'inter blocage nous avons proposé un nouveau

protocole d'allocation de ressources 'le protocole CDP'. Nous avons commencé par détailler le principe de base du protocole. Puis, nous avons défini l'ensemble des règles d'ordonnancement associées et comment calculé le temps de blocage des tâche de priorité moyenne s'il y on a.

Nous avons conçu et implémenté le protocole CDP, puis réalisé des essais pour constater les résultats à chaque changement de paramétrage de tâches et comparées ces résultats par celles des autres protocoles précédemment introduits. Les résultats obtenus sont satisfaisant par rapport au paramètre temps d'exécution et le nombre de blocage de la tâche prioritaire ainsi que le respect de l'ordre d'exécution selon l'algorithme RM, par contre l'optimalité reste toujours une solution approchée. Le protocole CDP proposé prouve qu'il :

- Minimise essentiellement la durée d'inversion de priorité.
- Est bien adapté sur les applications dont la tâche de plus haute priorité est critique.
- Réduire le nombre de blocage des tâches moyenne.
- Exclut complètement le blocage de la tâche prioritaire.

Mais, le gaspillage du temps CPU dû au CDP et au partage de ressources implique un temps d'exécution total trop élevé et les sections critiques de longue durée d'exécution provoquent souvent des cas de blocage.

En perspective nous envisageons d'appliquer le CDP dans un domaine réel, pour la suite de ce travail, d'empêcher le phénomène d'inversion de priorité tout en intégrant de nouvelle règles d'ordonnancement afin d'améliorer le paramètre temps total d'exécution des tâches et de se rapprocher à l'optimum, réduire le temps mort et traiter le cas où on a deux tâches possédant la priorité la plus haute.

Bibliographie

[Akl,09] Abbas Akli, « *Ordonnancement de tâches dans un système embarqué*», Thèse de magistère de l'ESI (École Nationale Supérieure d'Informatique) -Alger, 25 février 2009.

[And,99] C. André, « *Systèmes temps-réel, Université de Nice Sophia Antipolis* », 1999.

[Aud&Bur,90] N. Audsley and A. Burns, « *Real-Time Systems Scheduling* »; YCS 134, Departement of Computer Science, University of York, 1990.

[Bad,08] Lyes Badis, « *Ordonnancement hors-ligne des tâches temps réel à contraintes strictes à l'aide des colonies de fourmis en environnement multiprocesseur* », Thèse de magistère de l'Institut National d'Informatique-Alger, 21 Juillet 2008.

[Bak,91] T.P. Baker, « *Stack-Based Scheduling of Realtime Processes* », the Journal of Real-Time Systems, 3, pp. 67-99, 1991.

[Bil,87] W. E. Biles, « *Introduction to Simulation* », Proceedings of the Winter Simulation Conference, 1987.

[Bla,76] J. Blazewicz, «*Scheduling dependent tasks with different arrival times to meet deadlines*», in modelling and Performance Evaluation of Computer Systems, North Holland, Amsterdam, p. 57–65, 1976.

[Ban,Cuy,Liv,Cab,Rou,Wei,Pua,Dec,02] M. Banâtre, F. Cuyollaa, E. Livache, G. Cabillic, J. Routeau F. Weis, I. Puaut & D. Decotigny, *« Informatique diffuse et systèmes embarqués »,* Rennes, rapport d'activité, INRIA thème 1B 2002.

[Bou,98] Samia Bouzefrane, « *Étude temporelle des Applications Temps Réel Distribuées à Contraintes Strictes basée sur une Analyse d'Ordonnançabilité*», Thèse de Doctorat, LISI, ENSMA, 1998.

[Bou,09] Samia Bouzefrane, « *Ordonnancement centralisé de tâches temps réel* », CEDRIC – CNAM. http://cedric.cnam.fr/~bouzefra, accès 2009.

[Bur&Foh,91] A.Burns and G.Fohler, « *Incorporating flexibility into offline scheduling for hard real-time systems*», Technical Report RR-03-91, School of Computer Science,Technische Universität Wien, 1991.

[Cha,01] Chalmers « Parallel and Distributed Real-Time Systems », 2001.

[Che&Lin,90] M.I. Chen & K.J. Lin, « *Dynamic Priority Ceilings* », a Concurrency Control Protocol for Real-Time Systems, *The Journal of Real-Time Systems, 2*, pp 325-346, 1990.

[Cli,92] CliffordW. Mercer, « *An Introduction to Real-Time Operating Systems:Scheduling Theory* », School of Computer Science; Carnegie Mellon University; Pittsburgh, Pennsylvania 15213. November 13, 1992.

[Cot&Bab,94] F. Cottet and J.P. Babau, « *Off-Line Temporal Analysis of Hard Real-Time Applications* », second IEEE Workshop on Real-time Applications, Washington DC 1994.

[Cot,Del,Kai,Mam,00] Francis COTTET, Joëlle DELACROIX, Claude KAISER et Zoubir MAMMERI, « *Ordonnancement temps réel* ». HERMES Science Publications, 2000.

[Cot,Del,Kai,Mam,04] Francis COTTET; Joëlle DELACROIX; Claude KAISER; Zoubir MAMMERI; « *Ordonnancement temps réel, Ordonnancement centralisé* », Techniques de l'Ingénieur, traité Informatique industrielle, S8055, 2004.

[Cou,Dol,Kin,94] G. Coulouris, J. Dollimore, and T. Kindberg, « *Distributed Systems-Concepts and Design, 2nd Ed* », Addison-Wesley Publishers Ltd, 1994.

[Dec,03] D. Decotigny, « *Une infrastructure de simulation modulaire pour l'évaluation de performances de systèmes temps-réel »,* thèse de Doctorat de l'université de Rennes 1, 2003.

[Dem&Bon,99] I. Demeure and C. Bonnet, « *Introduction aux systèmes temps réel* », Collection pédagogique de télécommunications, Hermès, septembre 1999.

[Dou&Gho,10] Amel Doukhani et Nacira Ghoualmi, « *Vers un nouveau protocole pour contrer le phénomène d'inversion de priorité* », COSI'2010, 7ème colloque sur l'optimisation et les systèmes d'information (Université Kasdi Merbah, faculté STSM)-Ouargla-Algérie. pp 270-283, 19 Avril 2010.

[Ela,02] Patrick Elard, « *LES PROTOCOLES D'INVERSION DE PRIORITE* » Université de Bretagne Occidentale, 2002.
http://beru.univ-brest.fr/~singhoff/ter/rapports/elard2002.doc, accès 2008.

[God,90] S. Goddard « *Resource Sharing* », CSCE, Real-Time Systems, 1990.

[Gro,99] E. Grolleau, « *Ordonnancement temps réel hors-ligne optimal à l'aide de réseaux de Pétri en environnement monoprocesseur et multiprocesseur*» thèse de Doctorat, université de Poitiers, 1999.

[INF,09] INF4600 Systèmes temps réel, chap 4 : Notions avancées sur l'ordonnancement (partie 2). Ecole polytechnique de Montréal
http://www.cours.polymtl.ca, accès 2009.

[Jos&Pan,86] M. Joseph and P. Pandaya, « *Finding Response Times in a Real-Time System* »; the Computer Journal, 29(5), pp. 390-395, 1986.

[Ker,09] Omar KERMIA, « *Ordonnancement temps réel multiprocesseur de tâches non-préemptives avec contraintes de prédédence, de périodicité stricte et de latence* ». PhD thesis, Université Paris XI, UFR scientifique d'Orsay, 2009.

[Laa,05] Yacine Laalaoui, « *ordonnancement temps réel hors ligne a l'aide des colonie de fourmis en environnement monoprocesseur* », Thèse de magistère de l'INI, 6 Juillet 2005.

[Leh,sha&Din,89] J. Lehoczky, L. Sha and Y. Ding, «*The rate monotonic scheduling algorithm : exact characterization and average case behaviour* », in Proc.10th IEEE Real-Time Systems Symposium, Santa Monica, CA, p. 166-171, 1989.

[Leu&Whi,82] J. Leung & J.W. Whitehead, « *On the complexity of fixed-priority scheduling of periodic real-time tasks»,* Performance Evaluation, 2(4), 1982.

[Leu&Mer,80] J. Leung and M. Merril, « *A Note on Preemptive Scheduling of Periodic Real-Time Tasks* »; Information Processing Letters, 11(3), pp 115-118, November 1980.

[Liu,00] Jane W. S. Liu, « *Real-time Systems* », Prentice Hall (ISBN 0-13-099651-3), 2000.

[Liu&Lay,73] C.L.Liu and J.W.Layland, « *Schudeling algorithme for multiprogramming in a hard real time environnement* », Journal of The Association for Computing Machinery (ACM), v20, n1, pp 46-61,1973.

[Mok,83] A.K. Mok, « *Fundamental design problems for the hard real-time environments* », PhD. MIT, 1983.

[Nos,98] Roman Nossal, « *An evolutionary approach to multiprocessor schedulingof dependent tasks* », First workshopon biology inspired solutions to parallel processing problems, Orlando, Florida,IPDPS workshop 1998.

[Pai,06] S. Pailler, « *Analyse hors ligne d'ordonnancement d'applications temps réel comportant des taches conditionnelles et sporadiques* » thèse de Doctorat de l'université Poitiers, 2006.

[Pau,99] Laurent Pautet, « *Conception et realisation de composants logiciels pour les applications distribués temps reel* », these de l'Ecole Nationale Supérieure des Télécommunications, 25 janvier 1999.

[Ram,97] S. Ramamurthy, « *A lock-free approach to object sharing in real time systems* », University of North Carolina, USA, 1997.

[Ric&Cot,99] P. Richard & F. Cottet, « *Ordonnancement temps réel monoprocesseur avec contraintes de précédence simple et généralisée* », Rapport de Recherche LISI n°9902, 1999.

[Sen,07] M. Oscar Baez Senties, « *Méthodologie d'aide à la décision multicritère pour l'ordonnancement d'ateliers discontinus* », Thèse de Doctorat, l'Institut National Polytechnique de TOULOUSE, 2007.

[Sha,Raj&Leh,90] L. Sha, R. Rajkumar and J. Lehoczky, « *Priority Inheritance Protocols: An Approach to Real-Time Synchronization* »; IEEE Tr. on Computers, 39(9), September 1990.

[Spu&But,94] M. Spuri and G.C. Buttazzo, « *Efficient aperiodic service under earliest deadline scheduling* », Proc. of the IEEE Real-Time Systems Symposium, San Juan, Portorico, 1994.

[Sta,88] J. A. Stankovic, « *Misconceptions about real-time computing: a serious problem for the next-generation systems* », IEEE Computer., V21 N10, Oct.pp 10-19, 1988.

[Tin,Bur,Wel,92] K. Tindell, A. Burns, A.J. Wellings, « *Allocating hard- real-time tasks (an np hard problem made easy)* », Journal of Real- Time Systems, 4, (2), pp 145-165, 1992.

[Tou,91] Laurent TOUTAIN, « *SAMSON : Un simulateur pour systèmes répartis et temps-réel* », Thèse de Doctorat de l'université du HAVRE, 25 Novembre 1991.

[XU&Par,90] J. Xu and D. L. Parnas, « *Scheduling processes with release times, deadlines, precedence, and exclusion relations* ». IEEE Trans. on Software Engineering 16: 360-369. IEEE Computer Society Press, pp. 140-149, 1993.

[XU&Par,92] J.XU& D.Parnas, « *pre-run-time scheduling of process with exclusion relations on nested or overlapping critical sections* »,Phoneix conference on comuters and Communications, PP.6.4.7.1-6.4.7.9,apr 1992.