Vers un Nouveau Protocole Pour Contrer l'Inversion de Priorité

DOUKHANI AMEL & GHOUALMI.NACIRA
ameldoukhani@yahoo.fr – ghoualmi@yahoo.fr

Université Badji Mokhtar
Faculté des sciences de l'ingénieur
Département d'Informatique, ANNABA, ALGÉRIE
Informatique Industriel

Résumé
Dans un ordonnancement temps réel préemptif, la présence simultanée de priorités fixes et de ressources partagées à accès exclusif peut entraîner un phénomène appelé ***inversion de priorité***. Devant ce problème, nous présentons dans cet article un nouveau protocole d'allocation de ressources nommée *protocole à plafond dynamique* « ***CDP : Ceiling Dynamic Protocol*** » 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, des fois n'aura pas lieu. Le cadre de l'étude porte sur un ordonnancement préemptif de tâches périodiques avec des ressources partagées à plafond de priorité dynamique. Les objectifs de cette proposition sont : contrer l'inversement des priorités, éliminer l'inter-blocage, minimiser le temps de blocage et assurer l'exécution de la tâche la plus prioritaire sans aucune interruption.

Mots-clés : Ordonnancement temps réel, protocoles d'inversion de priorité, Plafond de priorité dynamique, Héritage de priorité, Temps mort.

1. Introduction

Souvent, on distingue des tâches utilisant des ressources partagées. A cet effet 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 priorités survient lorsque l'allocation du CPU est basée sur des priorités, mais que les processus ne sont pas indépendants. 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 (figure1).

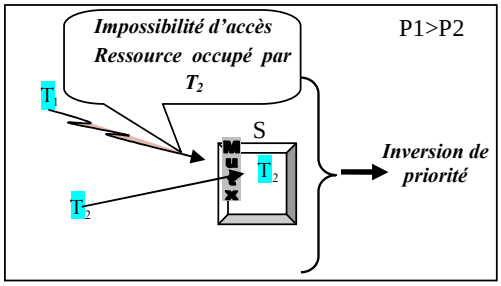

Figure 1 : *illustration de l'inversion de priorité*

Des fois ce problème mène à des dégâts inattendus. Ainsi, pour limiter les problèmes des deux phénomènes pré-cités, des protocoles tels que PIP **[1]**, OCPP et ICPP **[1,5]**, SRP **[2, 3, 4]** ont été introduits.

2. L'inversion de priorité

C'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 **[2]**. Les protocoles d'inversion de priorité ont été définis pour répondre à des problèmes supplémentaires ou non résolus. Ainsi, les risques d'inter-blocage et de chaînes de blocages avec PIP ont été résolus par PCP (Priority Ceiling Protocol : OCPP et ICPP) **[1, 3]**. 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 **[4]**, en partie due au fait que 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. Lorsqu'une tâche préempte une autre, on est sûr que les ressources dont elle a besoin seront disponibles **[3, 4]**.

3. Etude comparatif

Le but du tableau 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. Dans ce tableau 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. Concernant les deux indices ***n*** et ***m*** : selon **[1]** on a :

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.

	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 B_i	$B_i = \min\{ B_i^l, B_i^s \}$ $B_i^l = \sum_{j=i+1}^{n} \max_k \{D_{j,k} : \pi(R_k) \geq P_i \}$ $B_i^s = \sum_{k=1}^{m} \max_{j<i} \{D_{j,k} : \pi(R_k) \geq P_i \}$	$B_i = \max_{j,k} \{D_{j,k}: P_j < P_i$ et $\pi(R_k) \geq P_i \}$	$B_i = \max_{j, k} \{D_{j,k} :$ $P'_j < P'_i$ et $\pi(R_k) \geq P'_i \}$
Inter-blocage	Oui	Non	Non
Chaîne de blocage	Oui	Non	Non
Algorithme d'ordonnancement	RM	RM	RM/EDF
Priorité	Fixe	Fixe	Fixe/dynamique
Implémentation	Dur	Moyen	Facile

Tableau 1 : *Présentation d'une comparaison entre PIP, PCP et SRP*

4. Le protocole proposé pour contrer le phénomène d'inversion de priorité

Dans ce papier nous offrons un nouveau protocole « *protocole à plafond dynamique : **C**eiling **D**ynamic **P**rotocol (**CDP**)* » basé sur la réduction maximale du temps de blocage d'une tâche prioritaire. Le phénomène d'inversion de priorités est minimisé voir éliminé. Le protocole CDP pourrait engendrer un gaspillage du temps CPU (*temps morts*). La notion de temps mort est apparu dans le cas où les tâches à s'exécuté ont des dates d'activations proches dans le temps et leurs sections critiques sont de longue durée d'exécution.

Dans la suite du papier nous rappelons que la tâche i est notée $\mathbf{T_i}$, la tâche en attente est notée $\mathbf{T_a}$, la ressource k est notée $\mathbf{R_k}$, la priorité de la tâche i est notée $\mathbf{P_i}$, La priorité plafond courante (la plus haute priorité des tâches à l' instant t) est notée π', le nombre de tâche qui sollicite la section critique R_k est notée $Nb_{T_{user}(R_k)}$ et la l'instant d'arrivé de la tâche i est notée **PDM(Ti)**, le nombre de périodes successives requises à l'utilisation de la ressource R_k est notée D_{R_k}.

4.1-Principe du CDP

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 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 :

$$\pi(R_k) = \begin{cases} max_{T_i \text{ nécessite } (R_k)} P_i \\ \\ \varphi \end{cases} \qquad (1)$$

Où : P_i : Priorités des tâches nécessitent R_k pour accomplir leurs exécutions. φ : est une priorité plus basse que la priorité de la tâche la moins prioritaire des tâches qui ont utilisé R_k

Remarque : 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. Les ressources utilisées, les périodes de départ minimal, les séquences d'exécutions ainsi que les durées dont les ressources sont utilisées sont connues avant le début de l'exécution.

4.2- Protocoles d'ordonnancements proposés

- 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 :
  - Si $Nb_{T_{user}(R_k)} = 1$ alors T_i peut prendre la ressource R_k et entre en section critique.
  - Si $(Nb_{T_{user}(R_k)} \geq 1) et (P_i = \pi(R_k))$ alors T_i peut prendre la ressource R_k.
  - Si $(Nb_{T_{user}(R_k)} \geq 1) et (P_i < \pi(R_k))$ alors on a trois cas possibles :
    - **(a)** Si l'exécution de T_i dans R_k se termine avant ou avec les périodes de départ minimale de toutes les tâches prioritaires à Ti alors T_i peut prendre la ressource R_k et entre en section critique. De manière Formelle :

$$P_j > P_i : D_{R_k} \leq \left[min\left(PDM(T_j)\right) - t \right] \qquad (2)$$

(b) s'il y a des tâches T_g de priorités inférieures ou égales à P_i qui sont :

- interrompues par T_i ou par des tâches de priorité inférieures à P_i ;
- bloquées sur une ressource R_k ;

Alors on commence le test d'allocation pour la tâche la plus prioritaire entre eux :

- Si $T_a.etat = prête$ alors T_a peut commencer son exécution.
- Si $T_a.etat = préemptée$ alors T_a peut reprendre son exécution.
- Si $T_a.etat = bloquée\ sur\ R_{k'}$ (k' peut être égal à k) alors Si $D_{R_{k'}} \leq [min(PDM(T_j)) - t]$ alors la tâche T_a rehausse sa priorité à celle de π'(héritage de la priorité π'). Elle est alors exécutée avec la priorité hérité jusqu'à ce que une tâche de priorité supérieure à π' atteigne sa date d'activation, dans ce cas sa priorité redevient celle qu'elle était.

(c) La non prise de R_k par T_i provoque un temps mort qui dure :
[Le minimum des périodes de départ minimales des tâches qui n'ont pas encore déclenché leur exécution - l'instant où T_i veut prendre R_k]. De manière plus formelle :

$$P_j > P_i : D_{TM} = \left[min\left(PDM(T_j)\right) - t\right] \quad (3)$$

Tel que : T_j n'ont pas encore atteint leurs dates d'activation.

4.3-Caractéristiques du CDP

- Le CDP empêche les situations d'inter-blocage, qui peuvent survenir lorsqu'il y a plusieurs ressources partagées. Il peut provoquer le 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 l'instant d'arrivé d'une tâche prioritaire.
- Le CDP accompli l'exécution de la tâche la plus prioritaire sans aucune interruption.
- 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 de T_m sur 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. De manière plus formelle :

$$P_j > P_m : B_m = B_m + [min\left(PDM(T_j)\right) - t] \quad (4)$$

4.4-Étude Comparative Complémentaire

Le tableau 2 montre les différentes particularités de fonctionnements associées au protocole CDP.

	CDP (*Ceiling Dynamique Protocol*)
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 de durées de section critique)
Temps de blocage de Tm	$P_j > P_m : B_m = B_m + [min(PDM(T_j)) - t]$
Inter-blocage	Non
Chaîne de blocage	Non
Algorithme d'ordonnancement	En perspective
Priorité	Fixe
Temps d'exécution de la tâche prioritaire	Nombre de périodes requises à l'exécution
Implémentation	En perspective

Tableau 2 : *les caractéristiques attribuées au protocole CDP*

4.5- Avantages

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

4.6- Inconvénients

- × 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.

5- Exemple Proposé

L'idée de l'exemple a été prise de la référence **[1,3]**. On a sélectionné quatre tâches avec leurs priorités, périodes de départ minimal ainsi que leurs séquences d'exécution. Le tableau 3 propose cet exemple :

Tâches	*Priorités*	*Périodes de départ minimal*	*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	$ER_0R_0R_0E$

Tableau 3 : *les 4 tâches avec leurs Pi, PDM et séquences d'exécution associées*

Dans les figures qui suivent les différentes couleurs signifient :
Exécution normale, Tâche bloquée, che préemptée,
Section critique R_0, ection critique R_1.

5.1- L'exemple avec les priorités assignées aux tâches :

La figure 2 montre l'ordre d'exécution des 4 tâches avec leurs priorités assignées:

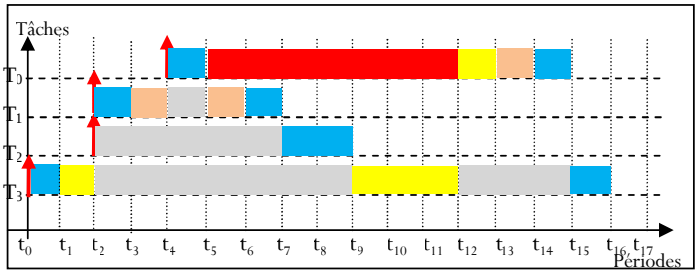

Figure 2 : l'ordonnancement des 4 tâches avec leurs priorités assignées

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 une période. A l'instant t_7 :T_1 termine son exécution. T_2 s'exécute pendant deux périodes dans aucune section critique. A l'instant t_9 : c'est la tâche T_3 qui continue son exécution dans R_0 durant 3 périodes car T_0 se retrouve bloquée sur cette ressource. A l'instant

t_{12} : T_3 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 R_1 et entre en section critique pendant une période. 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 une période. A l'instant t_{15} : T_0 termine son exécution. T_3 s'exécute pendant une période dans aucune section critique. A l'instant t_{16} : T_3 termine son exécution.

Conclusion :
Temps d'exécution de $T_0 = t_{15} - t_4 = 11$ périodes

On se serait attendu à ce que T_0 puisse faire sa séquence d'exécution en 4 périodes (ER_0R_1E), car c'est la tâche la plus prioritaire, mais 11 périodes ont été requises. 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.

5.2- L'exemple Avec PIP (Priority Inheritance Protocol) :

La figure 3 suivante montre l'ordre d'exécution des 4 tâches selon le protocole PIP :

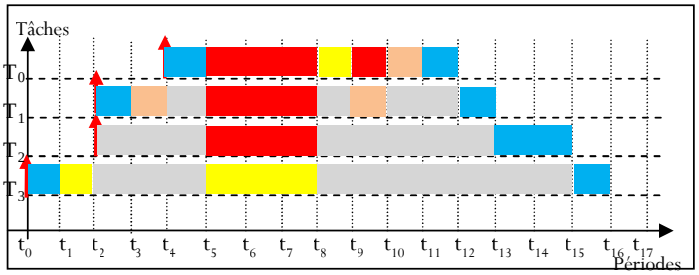

Figure 3 : l'ordonnancement des 4 tâches avec PIP

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 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_3 reprend son exécution mais avec la priorité P_0 de T_0 ($P_3 = P_0$) durant 3 périodes. 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 ($P_1 = P_0$) durant une période. 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 une période. A l'instant t_{12} : T_0 termine son exécution. T1 s'exécute pendant une période dans aucune section critique. A l'instant t_{13} : T_1 termine son exécution. T2 s'exécute pendant deux périodes dans aucune section critique. A l'instant t_{15} : T_2 termine son

exécution. T_3 reprend la sienne pendant une période dans aucune section critique. A l'instant t_{16} : T_3 termine son exécution.

Conclusion :
Temps d'exécution de $T_0 = t_{12} - t_4 = 8$ périodes

5.3- L'exemple avec OCPP (Original Ceiling Priority Protocol) :

Sachant que *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).

La figure 4 suivante montre l'ordre d'exécution des 4 tâches selon le protocole OCPP:

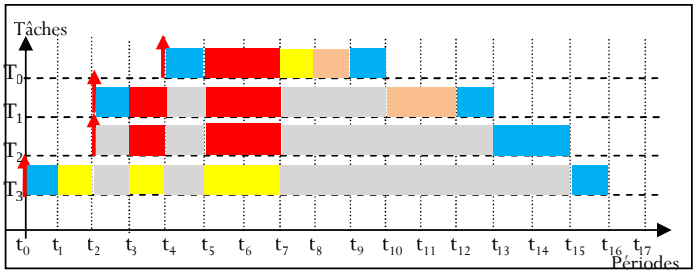

Figure 4 : l'ordonnancement des 4 tâches avec OCPP

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 $P_3 > \pi'$ et $\pi' = \Omega$ 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 supérieur à π' et $\pi' = \pi(R_0)$, 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 supérieure à π' et $\pi' = \pi(R_0)$, T_3 reprend son exécution avec la priorité P_0 durant deux périodes consécutives. 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 $P_0 > \pi'$ et $\pi' = \Omega$. A l'instant t_8 : T_0 libère R_0 et peut prendre R_1 car $P_0 > \pi'$ et $\pi' = \Omega$. A l'instant t_9 : T_0 libère R_1 et s'exécute pendant une période 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 périodes consécutives car $P_1 > \pi'$ et $\pi' = \Omega$. A l'instant t_{12} : T_1 libère R_1 et s'exécute pendant une période dans aucune section critique. A l'instant t_{13} : T_1 termine son exécution. T2 s'exécute pendant deux périodes dans aucune section critique. A l'instant t_{15} : T_2 termine son exécution. T_3 reprend la sienne pendant une période dans aucune section critique. A l'instant t_{16} : T_3 termine son exécution.

Conclusion :
Temps d'exécution de $T_0 = t_{10} - t_4 = 6$ périodes

5.4- L'exemple avec ICPP (Immediate Ceiling Priority protocol) :

La figure 5 suivante montre l'ordre d'exécution des 4 tâches selon le protocole ICPP :

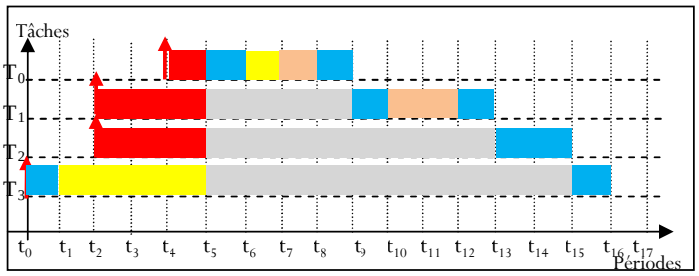

Figure 5 : l'ordonnancement des 4 tâches avec ICPP

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 $P_3>\pi$'et $\pi'= \Omega$ et entre en section critique avec $P_3=\pi(R_0)$ et $\pi(R_0)=P_0$. A l'instant t_2 : T_1 est réveillée, T_3 peut continuer à s'exécuter car $P_3=P_0$ et $P_0>P_1$ 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 $P_0 > \pi$' et $\pi' = \Omega$. A l'instant t_7 : T_0 libère R_0 et peut prendre R_1 car $P_0 > \pi$' et $\pi' = \Omega$. A l'instant t_8 : T_0 libère R_1 et s'exécute pendant une période 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 périodes consécutives car $P_1>\pi$' et $\pi'= \Omega$. A l'instant t_{12} : T_1 libère R_1 et s'exécute pendant une période dans aucune section critique. A l'instant t_{13} : T_1 termine son exécution. T_2 s'exécute pendant deux périodes dans aucune section critique. A l'instant t_{15} : T_2 termine son exécution. T_3 reprend la sienne pendant une période dans aucune section critique. A l'instant t_{16} : T_3 termine son exécution.

Conclusion *:*

Temps d'exécution de $T_0 = t_9 - t_4 = 5$ périodes

5.5- L'exemple avec SRP (Stack Resource Policy) :

La figure 6 suivante montre l'ordre d'exécution des 4 tâches selon le protocole SRP :

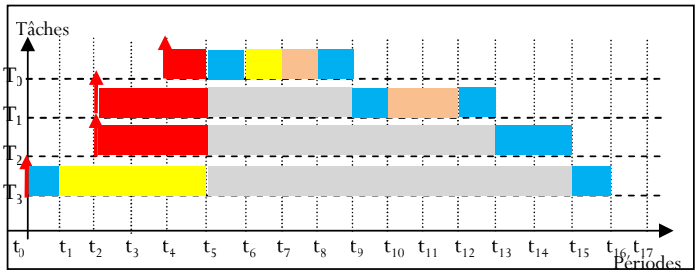

Figure 6 : l'ordonnancement des 4 tâches avec SRP

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 ($\pi'=\pi(R_0)$) et ($\pi(R_0)=P'_0$). 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 période. A l'instant t_9 : T_0 termine son exécution, T_1 peut alors s'exécuter. A l'instant t_{10} : T_1 prend R_1 durant deux périodes de suites, on a alors ($\pi'=\pi(R_1)$) et ($\pi(R_1)=P'_1$). 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 une période. A l'instant t_{13} : T_1 termine son exécution. T_2 s'exécute pendant deux périodes dans aucune section critique. A l'instant t_{15} : T_2 termine son exécution, T_3 reprend la sienne durant une période dans aucune section critique. A l'instant t_{16} : T_3 termine son exécution.

Conclusion *:*
Temps d'exécution de $T_0 = t_9 - t_4 = 5$ périodes

5.6- L'exemple avec CDP (Ceiling Dynamic Protocol):
La figure 7 suivante montre l'ordre d'exécution des 4 tâches selon le protocole CDP :

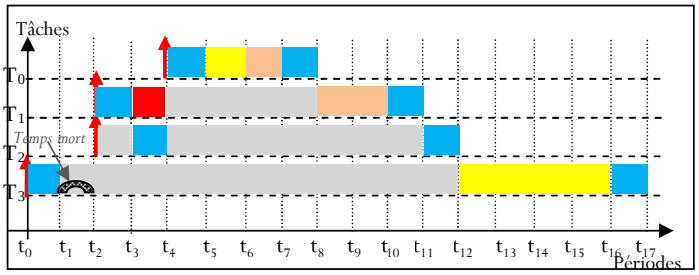

Figure 7 : l'ordonnancement des 4 tâches avec CDP

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 : $(Nb_{T_{user}(R_0)} > 1)$ et $P_3 < \pi(R_0)$ et $(D_{R_0} = 4 \text{ périodes}) > ([PDM(T_2 \text{ ou } T_3) - t_1] = [2 - 1] = 1 \text{ période})$, selon la règle d'ordonnancement (c) le processeur reste inactif pendant un temps mort qui dure $D_{TM} = [PDM(T_2) - t_1] = t_2 - t_1 = 1 \text{ période}$. 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 : $(Nb_{T_{user}(R_1)} > 1)$ et $P_1 < \pi(R_1)$ et $(D_{R_1} = 2 \text{ périodes}) > ([PDM(T_0) - t_3] = t_4 - t_3 = 1 \text{ période})$, selon la règle d'ordonnancement (b) on a *T_2.état=prête* alors T_2 hérite de π', $P_2 = \pi' = P_1$ et commence son exécution. A l'instant t_4 : T_0 est réveillé 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'exécuter et prend la ressource R_0 et entre en section critique parce que $P_0 = \pi(R_0)$. A l'instant t_6 : T_0 quitte R_0 ($\pi(R_0) = P_3$) et prend R_1 pendant deux périodes car $P_0 = \pi(R_1)$. A l'instant t_7 : T_0 quitte R_1 ($\pi(R_1) = P_1$) et continue son exécution durant une période 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 périodes successives car $P_1 = \pi(R_1)$. A l'instant t_{10} : T_1 quitte R_1 ($\pi(R_1) = \varphi$) et continue son exécution dans aucune section critique durant une période. A l'instant t_{11} : T_2 reprend son exécution dans aucune section critique durant une période car $P_2 > P_3$. A l'instant t_{12} : T_2 termine son exécution et T_3 peut reprendre la sienne et prend R_0 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 $T_0 = t_8 - t_1 = 4$ périodes.

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

6. Comparaison des résultats

Le but du tableau 4 est de comparer les résultats de l'application des quatre protocoles d'inversion de priorités sur l'exemple proposé.

	PIP	**PCP**	**SRP**	**CDP (proposé)**
Types de blocage	héritage de priorité	héritage de priorité plafond de priorité	Plafond de préemption	Section critique longue
Nombre de blocage de la tâche prioritaire	2	1	1	0
Nombre de blocage des tâches moyennes	1	1	1	1
Gaspillage de temps CPU	Non	Non	Non	Oui
Temps de blocage de la tâche prioritaire	4 périodes	OCPP : 2 périodes ICPP : 1 période	1 période	0 période
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 périodes	OCPP : 6 périodes ICPP : 5 périodes	5 périodes	4 périodes
Temps total d'exécution	16 périodes	16 périodes	16 périodes	17 périodes

Tableau 4 : *présentation d'une étude comparative des résultats obtenus par l'exemple proposé*

7. Conclusions & Perspectives

Les différentes techniques de résolution du problème d'inversion de priorité (PIP, PCP, SRP) atteignent certain seuil de réussite dans la réduction de ce problème. De même les risques de la situation d'inter-blocage et des chaines de blocage ont été éliminés notamment par le SRP. Mais, malgré tout ça le phénomène d'inversion de priorité reste un cauchemar pour quelques systèmes temps réel et particulièrement les systèmes temps réel à contrainte dure. Le protocole d'ordonnancement proposé « CDP» apporte plusieurs améliorations par rapport aux anciens protocoles. Il minimise essentiellement la production du phénomène d'inversion de priorité, lorsque la tâche la plus prioritaire commence son exécution, elle ne sera jamais bloquée sur une ressource ou interrompue par une tâche de moindre priorité. Ce protocole peut fournir des résultats intéressants, notamment sur des tâches de sections critiques courtes. Mais, le gaspillage du temps CPU dû à ce protocole 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 de contrer le phénomène d'inversion de priorité en évitant d'exploiter les temps mort.

RÉFÉRENCES

[1] L. Sha, R. Rajkumar, J. P. Lehoczhy, « *Priority inheritance protocols : an approach to real-time synchronization* », IEEE Transactions on Computers, Vol. n°39, n°9, sept, 1990.

[2] Samia Bouzefrane, « *Étude temporelle des Applications Temps Réel Distribuées à Contraintes Strictes basée sur une Analyse d'Ordonnançabilité* », Thèse de Doctorat, LISI, ENSMA, 1998.

[3] S. Ramamurthy, « *A lock-free approach to object sharing in real time systems* », University of North Carolina, USA, 1997.

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

[5] 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).