CEG4566/CSI4541 – Conception de systèmes temps réel

Chapitre 8 - Communication et synchronisation dans les systèmes temps réel

(Suite)

8.6 Blocages

8.6.1 Introduction

• L'exécution d'un processus nécessite un ensemble de ressources (espace mémoire centrale, espace disque, fichier, périphériques, …) qui lui sont attribuées par le système d'exploitation.

• L'utilisation d'une ressource passe par plusieurs étapes :

- Demande de la ressource ;

- Utilisation de la ressource ;

- Libération de la ressource.

• Lorsqu'un processus demande un accès à une ressource déjà allouée à un autre processus, le système d'exploitation décide de :

- le mettre en attente jusqu'à ce que la ressource demandée devienne disponible ou

- lui retourner un message indiquant que la ressource n'est pas disponible (réessayer plus tard).

• Les ressources sont de deux types :

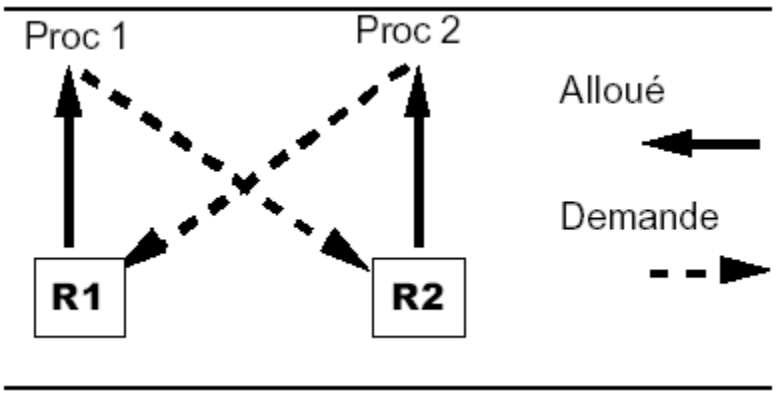
- Ressource **préemptible** : peut être retirée sans risque au processus qui la détient (mémoire).

- Ressource **non préemptible** : ne peut être retirée au processus qui la détient sans provoquer des problèmes (imprimantes).

• On considère ici uniquement les ressources non préemptibles.

• Des problèmes peuvent survenir, lorsque les processus obtiennent des accès exclusifs aux ressources.

Exemple : Un processus Proc1 détient une ressource R1 et attend une autre ressource R2 qui est utilisée par un autre processus Proc2 ; le processus Proc2 détient la ressource R2 et attend la ressource R1. On a une situation d'**interblocage** (Proc1 attend Proc2 et Proc2 attend Proc1). Les deux processus vont attendre indéfiniment.



8.6.2 Qu'est ce qu'un interblocage ?

• Un ensemble de processus est en interblocage si chaque processus attend la libération d'une ressource qui est allouée à un autre processus de l'ensemble.

• Comme tous les processus sont en attente, aucun ne pourra s'exécuter et donc libérer les ressources demandées par les autres. Ils attendront tous indéfiniment.

```
type Sem is ...;
X : Sem := 1; Y : Sem := 1;
```

```
task A;
task body A is
begin
   ...
   Wait(X);
   Wait(Y);
   ...
end A;
```

```
task B;
task body B is
begin
   ...
   Wait(Y);
   Wait(X);
   ...
end B;
```

• Un 'blocage vivant' est quand deux processus s'exécutent mais aucun d'eux ne peut progresser.

```
type Flag is (Up, Down);
Flag1 : Flag := Up;
```

```
task A;
task body A is
begin
   ...
   while Flag1 = Up loop
     null;
   end loop;
   ...
end A;
```

```
task B;
task body B is
begin
   ...
   while Flag1 = Up loop
     null;
   end loop;
   ...
end B;
```

8.6.3 Exemples d'interblocage

8.6.3.1 Accès aux périphériques

Supposons que deux processus A et B veulent imprimer, en utilisant la même imprimante, un fichier stocké sur un DVD ROM. La taille de ce fichier est supérieure à la capacité du disque. Chaque processus a besoin d'un accès exclusif au lecteur DVD et à l'imprimante simultanément.

Analyser la situation d'interblocage.

Réponse : Il y aura interblocage dans les cas suivants :

- le processus A utilise l'imprimante et demande l'accès au lecteur DVD;
- le processus B détient le lecteur DVD et demande l'imprimante.

8.6.3.2 Accès à une base de données

Supposons deux processus A et B qui demandent des accès exclusifs aux enregistrements d'une base de données.

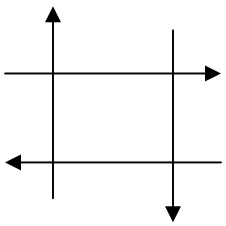
On arrive à une situation d'interblocage si :

- Le processus A a verrouillé l'enregistrement R1 et demande l'accès à l'enregistrement R2 ;
- Le processus B a verrouillé l'enregistrement R2 et demande l'accès à l'enregistrement R1.

8.6.3.3 Circulation routière

Considérons deux routes à double sens qui se croisent.

Analyser ce système.



Remarques :

• La résolution des interblocages constituent un point théorique très étudié. Cette étude n'a cependant pas fourni, pour l'instant, de solution complètement satisfaisante et la plupart des systèmes ne cherche pas à traiter ce phénomène.

• Cependant, si on ne peut pas résoudre le problème dans sa généralité, on devra tenir compte de certaines techniques qui minimisent les risques.

8.6.4 Quand est-ce qu'on a une situation d'interblocage ?

On a une situation d'interblocage si les quatre conditions suivantes sont remplies :

- l'exclusion mutuelle : Une ressource est soit allouée à un seul processus, soit disponible;

- la détention et l'attente : Les processus qui détiennent des ressources peuvent en demander d'autres;

- pas de réquisition : Les ressources allouées à un processus sont libérées uniquement par le processus;

- l'attente circulaire : Il se produit un cycle dans le graphe orienté représentant la détention et l'attente (graphe d'allocation des ressources).

8.6.5 Graphe d'allocation des ressources :

Le graphe d'allocation des ressources est un graphe biparti composé de deux types de nœuds et d'un ensemble d'arcs :

- Les processus qui sont représentés par des cercles;

- Les ressources qui sont représentées par des rectangles. Chaque rectangle contient autant de points qu'il y a d'exemplaires de la ressource représentée;

- Un arc orienté d'une ressource vers un processus signifie que la ressource est allouée au processus.

- Un arc orienté d'un processus vers une ressource signifie que le processus est bloqué en attente de la ressource.

Exemple **:** Ce graphe indique pour chaque processus les ressources qu'il détient ainsi que celles qu'il demande.

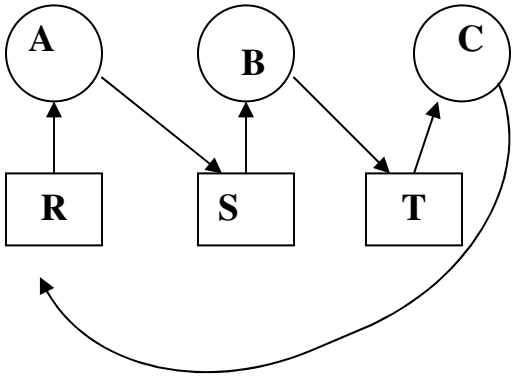
Soient trois processus A, B et C qui utilisent trois ressources R, S et T comme suit :

A	B	C
Demande R	Demande S	Demande T
Demande S	Demande T	Demande R
Libère R	Libère S	Libère T
Libère S	Libère T	Libère R

• Si les processus sont exécutés séquentiellement A suivi de B suivi C, il n'y pas d'interblocage.

• Supposons que l'exécution des processus est gérée par un ordonnanceur circulaire. On atteint une situation d'interblocage, si les instructions sont exécutées dans cet ordre :

1) A demande R
2) B demande S
3) C demande T
4) A demande S
5) B demande T
6) C demande R



8.6.6 Traitement des interblocages

Comme nous l'avons mentionné, les situations d'interblocage peuvent se produire dans un système. Doit-il prendre en compte ce problème ou l'ignorer ?

8.6.6.1 Ignorer complètement les problèmes

On peut ignorer les possibilités d'interblocages.

Cette « stratégie » est celle de la plupart des systèmes d'exploitation courants car le prix à payer pour les éviter est élevé (les restrictions sur les processus).

8.6.6.2 Les détecter et y remédier

On tente de traiter les interblocages, en détectant les processus interbloqués et en les éliminant.

8.6.6.3 Les éviter

En allouant dynamiquement les ressources avec précaution. Le système d'exploitation peut suspendre le processus qui demande une allocation de ressource, s'il constate que l'allocation d'une ressource peut conduire à un interblocage. Il lui attribuera la ressource lorsqu'il n'y aura plus de risque.

8.6.6.4 Les prévenir

En empêchant l'apparition de l'une des quatre conditions de leur existence (*voir paragraphe 7.4*).

8.6.7 La détection et la reprise

• Dans ce cas, le système ne cherche pas à empêcher les interblocages. Il tente de les détecter et d'y remédier.

• Pour détecter les interblocages, il construit dynamiquement le graphe d'allocation des ressources du système qui indique les attributions et les demandes de ressources.

• Dans le cas des ressources à exemplaire unique, il existe un interblocage si le graphe contient au moins un cycle.

• Dans le cas des ressources à exemplaires multiples, il existe un interblocage si le graphe contient au moins un cycle terminal (aucun arc ne permet de le quitter).

• Le système vérifie s'il y a des interblocages :

- A chaque modification du graphe suite à une demande d'une ressource (coûteuse en termes de temps processeur).

- Périodiquement ou lorsque l'utilisation du processeur est inférieure à un certain seuil (la détection peut être tardive).

8.6.8 Méthode pratique de détection des interblocages

Soient *n* le nombre de processus (P1,P2,…,Pn) et *m* le nombre de types de ressources (E1,E2,..,Em) qui existent dans un système. Cette méthode utilise deux matrices et deux vecteurs :

- Matrice C des allocations courantes d'ordre (n x m). L'élément C[i,j] désigne le nombre de ressources de type Ej détenues par le processus Pi.

- Matrice R des demandes de ressources d'ordre (*n* x *m*). L'élément R[i,j] est le nombre de ressources de type Ej qui manquent au processus Pi pour pouvoir poursuivre son exécution.

- Vecteur A d'ordre *m*. L'élément A[j] est le nombre de ressources de type Ej disponibles (non attribuées).

- Vecteur E d'ordre *m*. L'élément E[j] est le nombre total de ressources de type Ej existantes dans le système

L'algorithme de détection :

1) Rechercher un processus Pi non marqué dont la rangée i de R est inférieure à A ;

2) Si ce processus existe, ajouter la rangée *i* de C à A, marquer le processus et revenir à l'étape 1 ;

3) Si ce processus n'existe pas, les processus non marqués sont en interblocage. L'algorithme se termine.

Exemple :

Considérons un système où nous avons trois processus en cours et qui dispose de 4 types de ressources : 4 dérouleurs de bandes, 2 tables traçantes, 3 imprimantes et un lecteur de CD ROM. Les ressources détenues et demandées par les processus sont indiquées par les matrices C et R.

E=(4 2 3 1) A=(2 1 0 0)

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

$$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

C : ressources attribuées ;

R : ressources demandées non encore obtenues ;

A : ressources disponibles ;

E : toutes les ressources du système

• Seul le processus P3 peut obtenir les ressources dont il a besoin. Il s'exécute jusqu'à la fin puis libère ses ressources, ce qui donne : A = (2 2 2 0).

• Ensuite P2 peut obtenir les ressources dont il a besoin. Il s'exécute jusqu'à la fin et libère ses ressources, ce qui donne : A = (4 2 2 1).

• Enfin, P1 peut obtenir les ressources dont il a besoin. Il n'y a pas d'interblocage.

8.6.9 La reprise des interblocages

Lorsque le système détecte un interblocage, il doit le supprimer, ce qui se traduit généralement par la réalisation de l'une des opérations suivantes :

- Retirer temporairement une ressource à un processus pour l'attribuer à un autre.
- Restaurer un état antérieur (retour arrière) et éviter de retomber dans la même situation.
- Supprimer un ou plusieurs processus.

8.6.10 L'évitement des interblocages

• Dans ce cas, lorsqu'un processus demande une ressource, le système doit déterminer si l'attribution de la ressource est **sûre**. Si c'est le cas, il lui attribue la ressource. Sinon, la ressource n'est pas accordée.

• Un état est **sûr** si tous les processus peuvent terminer leur exécution (il existe une séquence d'allocations de ressources qui permet à tous les processus de se terminer).

• Un état est **non sûr** si on ne peut garantir que les processus pourraient terminer leurs exécutions.

Comment déterminer si un état est sûr ou non sûr ?

• Dijkstra (1965) a proposé un algorithme d'ordonnancement, (*algorithme du banquier*) qui permet de répondre à cette question.

• Cet algorithme, utilise les mêmes informations que celles de l'algorithme de détection précédent (A,E,C,R).

Un état est caractérisé par ces quatre tableaux:

E=(4 2 3 1) A=(2 1 0 0)

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix} \qquad R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

C : ressources attribuées ;

R : ressources demandées non encore obtenues ;

A : ressources disponibles ;

E : toutes les ressources du système

Algorithme du banquier :

• Consiste à examiner chaque nouvelle requête pour voir si elle conduit à un état sûr. Si c'est le cas, la ressource est allouée, sinon la requête est mise en attente.

• L'algorithme qui détermine si un état est sûr :

1) trouver un processus Pi non marqué dont la rangée i de R est inférieure à A ;

2) Si un tel processus n'existe pas alors l'état est non sûr (il y a interblocage). L'algorithme se termine.

3) Sinon, ajouter la rangée i de C à A, marquer le processus ;

4) Si tous les processus sont marqués alors l'état est sûr et l'algorithme se termine, sinon aller à l'étape 1.

• Cet algorithme permet bien d'éviter les interblocages mais il est peu utile en pratique car on ne connaît pas toujours à l'avance les besoins en ressources des processus.

8.6.11 La prévention des interblocages

Pour prévenir les interblocages, on doit éliminer une des quatre conditions nécessaires à leur apparition.

• Pour éviter l'exclusion mutuelle, il est parfois possible de sérialiser les requêtes portant sur une ressource.

Par exemple, pour les imprimantes, les processus « spoulent » leurs travaux dans un répertoire spécialisé et un démon d'impression les traitera, en série, l'un après l'autre.

• En ce qui concerne la deuxième condition, elle pourrait être évitée si les processus demandaient leurs ressources à l'avance. Ceci est en fait très difficile à réaliser dans la pratique car l'allocation est, en général, dynamique. Empêcher cette condition serait donc particulièrement coûteuse.

• La troisième condition n'est pas raisonnablement traitable pour la plupart des ressources sans dégrader profondément le fonctionnement du système. On peut cependant l'envisager pour certaines ressources dont le contexte peut être sauvegardé et restauré.

• Enfin, on peut résoudre le problème de l'attente circulaire en numérotant les ressources et en n'autorisant leur demande, par un processus, que lorsqu'elles correspondent à des numéros croissants ou en accordant aux processus une seule ressource à la fois (s'il a besoin d'une autre ressource, il doit libérer la première). Dans ce cas, si un processus détient une ressource i et attend une autre ressource j on a forcément $i < j$. Le processus qui détient la ressource j ne peut pas se mettre en attente de la ressource i (pas de risque d'interblocage).

8.7 Communication par message

8.7.1 Introduction

Les tâches temps-réel ont besoin d'échanger des données afin :

- d'accomplir leurs objectifs.
- coopérer durant l'exécution d'une application.

Problèmes de la programmation mono-tâche :

- Les formes de communication utilisées en programmation mono-tâche (paramètres d'appels de procédure, valeur de retour de fonction) ne sont pas adaptées au modèle d'exécution pseudo-simultanée des tâches (pseudo-parallèle).
- Aucune raison pour qu'une tâche réceptrice soit prête à prendre le message au moment où il a été envoyé par la tâche émettrice.
- Le débit instantané des messages produits par l'émetteur peut excéder la capacité d'absorption du récepteur et il faut alors pouvoir, soit agir sur le rythme d'émission, soit stocker temporairement les messages dans une mémoire tampon.

Solutions :

• La seule forme de communication possible reposera donc sur une zone commune de données ou sur des messages.

• Ces données peuvent prendre la forme d'un message; celui-ci est créé par le noyau temps-réel, délivré à une ou plusieurs tâches et effacé.

• Le noyau offre un service de communication (ou de gestion de messages).

Note 1: L'ordre d'arrivée et de délivrance des messages est important.

Note 2 : On associera la notion de synchronisation entre tâches à celle de communication. En effet, pour se synchroniser, les tâches seront obligées de communiquer entre elles.

8.7.2 Communication par messages

8.7.2.1 Principes

• Les principaux mécanismes de communication reposent sur la notion de file (ou queue) de messages.

• Pour éviter les problèmes liés à l'exécution en parallèle des tâches et répondre aux exigences que posent la synchronisation et la notion d'exclusion mutuelle, le mécanisme de communication le plus simple est celui de la boîte aux lettres (*mailbox*).

8.7.2.2 Boîte aux lettres

• Un message est déposé par la tâche émettrice dans un endroit réservé case-mémoire de 32 bits ou de file de messages (*queue*) avec plusieurs cases-mémoires (32 bits par message) à cet effet. La tâche réceptrice consultera alors la boîte aux lettres pour voir si un message est disponible.

• Les échanges s'effectuent de façon indirecte, le message est placé en mémoire par la tâche émettrice et il est récupéré dans cette case par la tâche réceptrice.

• Une boîte aux lettres est un mécanisme de communication déclaré par l'utilisateur mais géré par le noyau.

• Les primitives que l'on rencontre généralement dans les noyaux temps-réel sont de deux natures:

- ***poster (boîte, message)***
- ***message = retirer (boîte)***

• La première permet de poster le message donné à la boîte aux lettres précisée.

• La seconde fournit en retour un pointeur (message) contenu dans la boîte aux lettres citée.

• Les opérations réalisées par ces deux primitives sont non interruptibles de façon à éviter que plusieurs tâches ne retirent ou ne déposent simultanément un message.

Note : Dans les programmes C (concurrent) les boites aux lettres (*mbox*) sont définies comme des pointeurs (char *mbox ; 32 bits ; variable globale). Un message est une variable également de la taille d'un pointeur. Avec ce principe, l'utilisateur déclare le contenu de la boîte aux lettres, il a donc la possibilité de l'initialiser et &mbox sera l'adresse de cette boîte.

8.7.2.3 Poster un message

• Les primitives qui permettent de gérer les boîtes aux lettres ont des particularités qui dépendent de la conception du noyau.

• Le postage d'un message est une fonctionnalité qui dépend du noyau. D'une manière générale, la tâche poste (quel que soit l'état de la boîte) et, éventuellement, poursuit son code.

• Sous certains noyaux, la tâche productrice du message peut être suspendue si la boîte aux lettres est pleine ; ceci permet de réaliser un mécanisme de synchronisation (Producteur-consommateur).

8.7.2.4 Attendre un message

• Lorsqu'une tâche attend un message, on peut également observer plusieurs comportements selon les noyaux. Le plus fréquent est le suivant :

Une tâche “ récepteur ” veut récupérer un éventuel message dans une boîte aux lettres précise ; elle utilise la primitive équivalente à retirer. Si la boite aux lettres est vide, la tâche est suspendue, soit pour une durée précisée (timeout ≠ 0), soit jusqu'à ce qu'un message soit présent dans la boite aux lettres (timeout = 0). Dans ce deuxième cas, la tâche restera suspendue tant que la boite aux lettres restera vide.

• Il est à noter que plusieurs tâches peuvent attendre devant la même boîte aux lettres. Dans ce cas, c'est la plus prioritaire qui recevra le message. Si elles ont la même priorité, la plus ancienne (dans l'attente) sera la réceptrice.

• Si aucun message n'est présent après la durée d'attente précisée (timeout ≠ 0), la tâche reprend son exécution et un message d'erreur est émis.

• Si un message est présent, la fonction retourne le message. Une fois le message retiré, la boîte aux lettres est vide.

• Lorsqu'une tâche attend devant une boîte aux lettres et qu'un message y arrive, elle le retire directement et se retrouve donc prête à l'exécution ; si elle est plus prioritaire que la tâche qui a posté le message, cette dernière se retrouve prête à l'exécution, le processeur étant attribué à la tâche consommatrice.

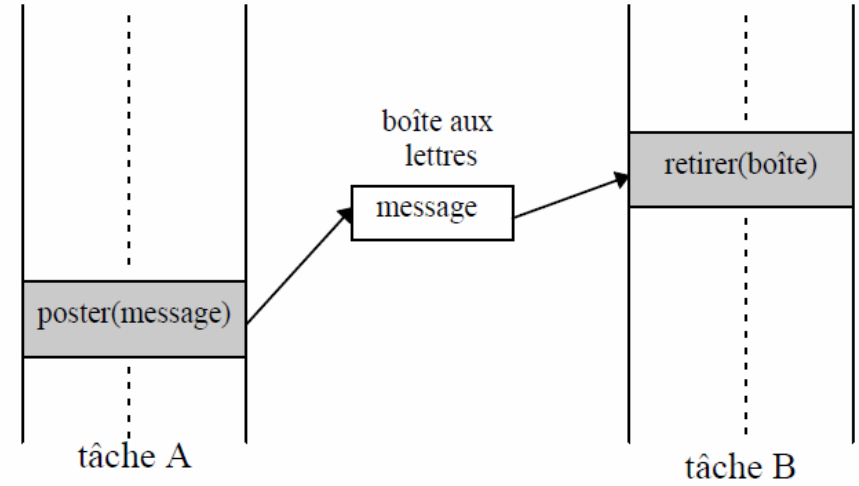
8.7.2.5 Utilisations des boîtes aux lettres

• Selon la manière dont elle est déclarée, une boîte aux lettres peut avoir des rôles différents.

• Elle peut servir au *transfert de données*, à la *synchronisation* ou à l'*exclusion mutuelle*.

• Dans le premier cas, la boite aux lettres est initialisée à zéro (elle est vide). La transmission de la donnée se fera selon le schéma de la figure suivante.

Dans le cas de la figure suivante, même si la tâche B est plus prioritaire que la tâche A, il faudra qu'elle attende (pas de timeout) que la tâche A ait posté son message pour continuer son exécution (dès qu'elle reçoit son message, car plus prioritaire). La boite aux lettres étant vide au départ, elle était immédiatement disponible pour la tâche A. Si elle avait été initialisée à une valeur non nulle, la tâche B aurait pu retirer un message avant que la tâche A ne poste le sien.



Il est parfois nécessaire d'initialiser une boite aux lettres ce qui sert à synchroniser le passage des informations.

Exemple *(d'après Marc Rivaletto)*

Prenons un cas précis pour évoquer cet aspect de l'utilisation des boites aux lettres.

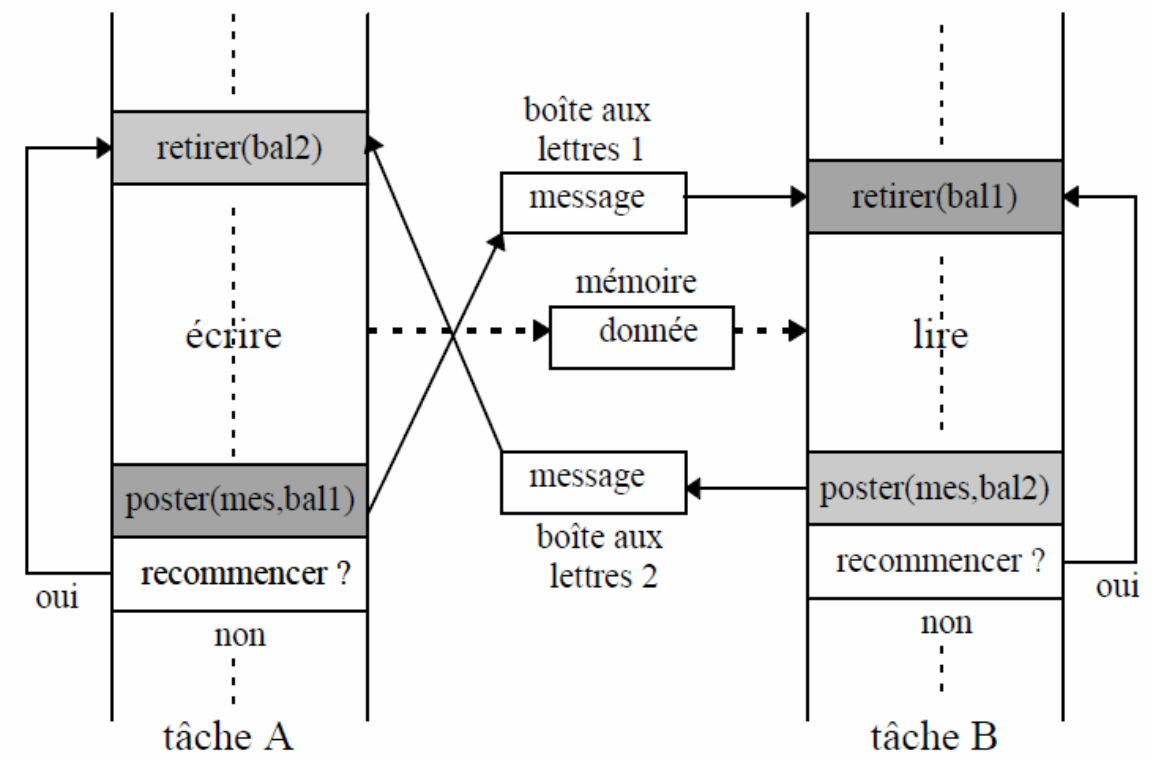
Soit une tâche A qui produit des données qu'elle range en mémoire. Elle ne peut les faire passer qu'une par une à la tâche B qui utilisera ces données, et une donnée peut être écrasée par la suivante si elle n'est pas utilisée par la tâche B. Le problème sera résolu en prenant deux boites aux lettres :

BAL1 et BAL2. Elles serviront à la synchronisation du processus. La valeur des messages postés dans BAL1 et BAL2 n'a pas d'importance, seule leur présence est indispensable.

La *figure suivante* explique clairement cette méthode. On remarquera que la zone mémoire qui sert au transfert de la donnée, peut être aussi une boîte aux lettres ce qui ne modifiera en aucun cas le fonctionnement de cette méthode. Dans ce cas, on se souviendra qu'une donnée lue est automatiquement retirée de la boite aux lettres.

Pour que la tâche A puisse écrire la première donnée, il faut qu'elle ait d'abord retiré un message de la BAL2. Il sera donc nécessaire d'initialiser la BAL2 pleine à la création et de laisser, au début, BAL1 vide.

Cette méthode permet à la tâche B d'éviter de lire deux fois de suite (ou plus) la même valeur de la donnée. La BAL1 sert à dire qu'une donnée est disponible, la BAL2 qu'une donnée vient d'être lue (la lecture ne vide pas la case mémoire).



Ces boîtes aux lettres, utilisées dans ces rôles bien particuliers, " remplacent " des drapeaux événements ou des sémaphores car elles sont liées, comme dans l'exemple, à celle de synchronisation.

De telles méthodes de synchronisation peuvent présenter un inconvénient : il est en effet possible d'interbloquer deux tâches lors de l'utilisation simultanée de plusieurs ressources.

Pour éviter ce problème de synchronisation lors du transfert de multiples données, on peut utiliser la notion de files de messages. Ce sont des mémoires tampons dynamiques de taille fixée. Elles sont créées et gérées par le noyau temps réel grâce à des primitives.

8.7.2.6 Files de messages

• Une file de messages peut être assimilée à un regroupement de plusieurs boîtes aux lettres.

• La principale (et fondamentale) différence avec la structure précédente vient du fait que celle-ci est créée dynamiquement par l'application.

• Les appels-système qui gèrent les files de messages sont les suivants :

- *création d'une file, gérée par priorité*
- *création d'une file, gérée par priorité ou FIFO*
- *poster un message à la fin de la file*
- *poster un message au début de la file*
- *attendre un message d'une file*
- *accepter un message d'une file*
- *informer sur le nombre de messages présents*

• Les messages (*msg*) peuvent être postés dans la file tant que celle-ci n'est pas pleine en précisant uniquement l'identificateur (*id*) de la file choisie. Les messages dans une file sont stockés soit dans l'ordre FIFO soit dans l'ordre LIFO.

• Au niveau de l'attente, celle-ci peut être soit par priorité soit par ancienneté (FIFO), ceci est précisé à la création de la file.

• Lorsqu'une tâche retire un message, elle retire le premier de la file. Si une file est vide, la tâche est suspendue, soit tant qu'aucun message n'arrive, soit pendant une durée fixée (timeout) lors de la demande de retrait de messages.

• Les commutations de tâches qui peuvent intervenir lors de l'utilisation de files de messages sont du même type que celles rencontrées lors de l'utilisation de boîtes aux lettres. Elles dépendent principalement des priorités affectées aux différentes tâches.

• Les files de messages permettent donc de transférer beaucoup de données entre tâches sans se préoccuper de savoir si on va écraser le message ou si la tâche consommatrice ne va pas utiliser deux fois le même message.

Note : On peut utiliser les files de messages pour gérer des ressources par exclusion mutuelle.

8.7.2.7 Communication par zone commune de données

• La façon la plus simple et la plus rapide de transmettre des données est de ne pas en transmettre.

• Il suffit de disposer d'une zone de mémoire commune dans laquelle seront disposées les données en libre service.

• L'allocation de cette zone sera réalisée à l'aide de variables globales ou de requêtes spécifiques.

• Dans un système mono-tâche, l'accès aux zones communes de données est séquentiel. L'ordre dans lequel il s'effectue est imposé par la structure du programme.

• Dans un système multitâche, l'accès aux zones communes de données est fonction de l'ordonnancement des tâches.

• Le problème de communication par zone commune de données est surtout lié à la synchronisation entre les tâches qui communiquent. En effet les méthodes d'écriture et de lecture de données sont assez évidentes lorsqu'on travaille avec des pointeurs. Il suffit de lire une donnée seulement après que celle-ci ait été écrite et de ne pas écrire une nouvelle donnée avant que l'ancienne n'ait été lue. Comme précédemment, ces problèmes ont été facilement résolus en utilisant une boîte aux lettres.

8.7.2.8 Les rendez-vous

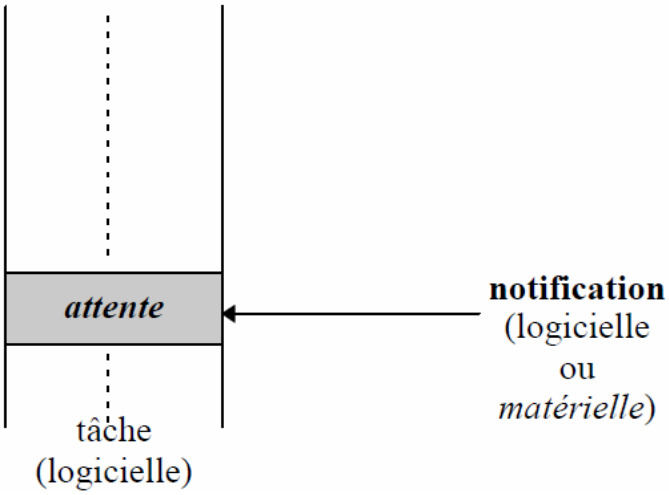
a) Rendez-vous unilatéral

• On utilise le terme de ***rendez-vous*** lorsqu'un processus attend une notification extérieure, la reçoit et continue son exécution.

• Un rendez-vous sous-entend que la tâche en attente n'espère pas une information, la notification étant uniquement un signal qui l'autorise à poursuivre son exécution.

• Un rendez-vous unilatéral est un point où une tâche se synchronise avec quelque chose d'externe, sans que l'agent extérieur ait besoin de se synchroniser avec la tâche. L'agent extérieur peut être une autre tâche ou une interruption. Le code en attente doit être celui d'une tâche et non celui d'une routine d'interruption car cette dernière ne peut rester en attente pendant un temps indéterminé.

• La figure suivante illustre ce simple rendez-vous. Les appels servant à la notification sont ceux autorisés pour les tâches matérielles.



b) Rendez-vous bilatéral

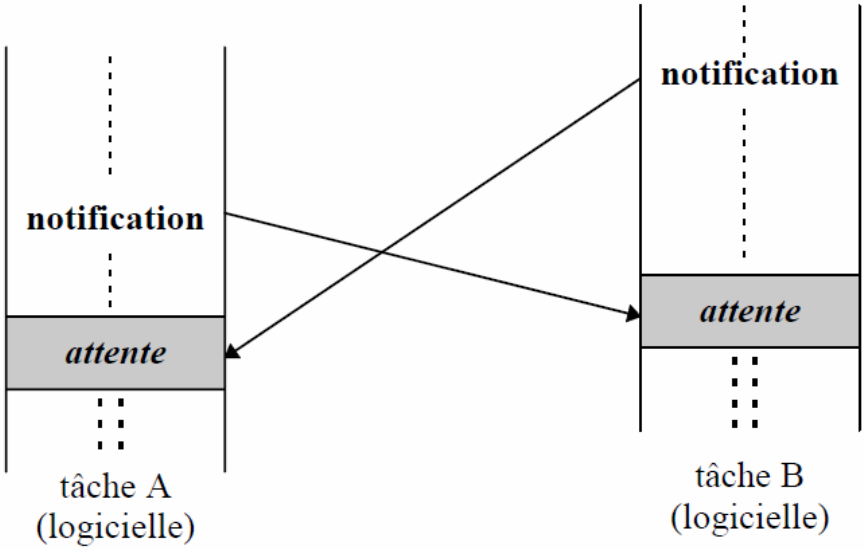
Un rendez-vous bilatéral peut être considéré comme un double rendez-vous unilatéral où les deux parties doivent se synchroniser avant de continuer.

Exemple : *(d'après Marc Rivaletto)*

Considérons deux tâches interdépendantes A et B, avec chacune une conjoncture critique au-delà de laquelle une tâche ne peut continuer si l'autre n'a pas atteint sa propre conjoncture.

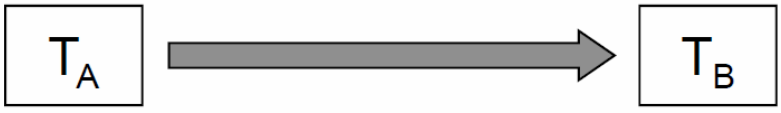
Si la tâche A y arrive en premier, elle avertit la tâche B et attend que B fasse pareil pour continuer son code.

Pour les mêmes raisons que précédemment, une routine d'interruption ne peut participer à un rendez-vous sous peine de bloquer toute l'application.



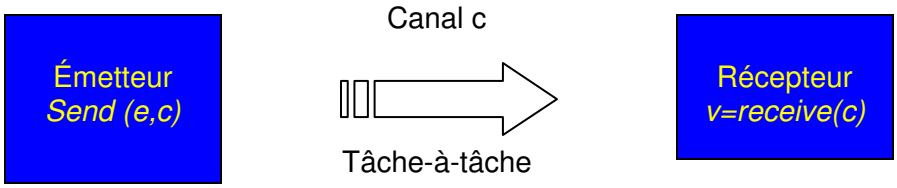
8.7.3 Notion de communication synchrone et asynchrone

8.7.3.1 Communication synchrone



Si la communication se fait directement de tâche à tâche, on parle de communication synchrone; les tâches s'attendent mutuellement.

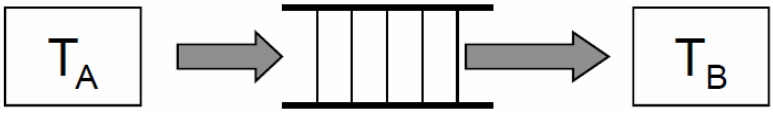
C'est le cas du ***rendez-vous***.



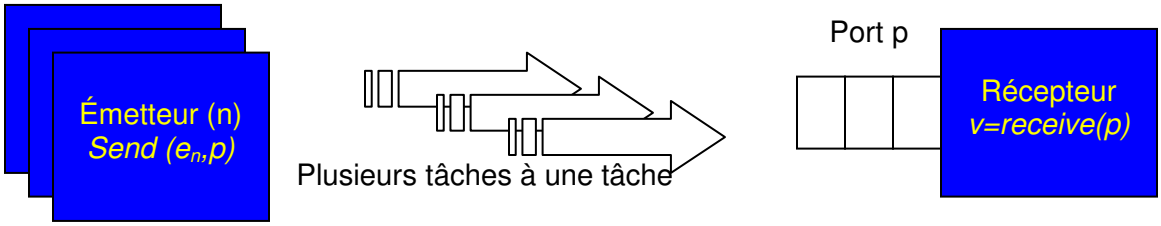
Send (e,c) : Envoyer la valeur de l'expression *e* à travers le canal *c*.

v=receive(c) : Recevoir une valeur dans une variable locale *v* à partir du canal *c*.

8.7.4 Communication asynchrone



- Dans la majorité des cas, la communication est asynchrone: la communication nécessite la présence d'une mémoire tampon.
- Cette mémoire tampon est la *boîte au lettres*, ou *file de messages*, ou *queue de messages*.
- Cette file est généralement bornée: un nombre maximum de messages peut y prendre place
- Le modèle de communication asynchrone nécessite les éléments suivants:
  - Un message
  - Caractéristiques: taille, structure prédéfinie (date, taille effective, etc.)
  - Une file de messages
  - Caractéristiques: nom, taille (totale), nombre de messages maximum
  - Une file d'attente
  - Les tâches en attente d'un message sont placées dans une file d'attente. L'ordre dans lequel les tâches recevront le message dépendra de la politique de la file d'attente.
    - Politique FIFO: la première tâche en attente sera la première servie.
    - Politique "par priorité": la tâche la plus prioritaire sera desservie.
    - L'envoi d'un message s'accompagne d'un mode de réveil des tâches en attente.
    - Mode *normal*: une seule tâche en attente est informée de la présence d'un message.
    - Mode *broadcast*: toutes les tâches en attente d'un message sont informées de la présence d'un message.



Send (e,p) : Envoyer la valeur de l'expression *e* au port *p*. Le message est mis en file d'attente.

v=receive(p) : Recevoir une valeur dans une variable locale *v* à partir du port *p*. Le processus appelant l'opération *receive* sera bloqué s'il n'y a aucun message en attente dans la file d'attente du port *p*.

Exemple : File de message (d'après cours Daniel Rossier – heig-vd)

Un message peut être diffusé à toutes les tâches en attente (le mode de réveil est alors le mode *broadcast*).