Communication inter-processus

par Aurélien Lucchi

Date de publication : 12/06/2005

Dernière mise à jour : 26/08/2005

Cet article abordera les différents moyens existant pour faire communiquer des processus entre eux dans le cadre d'un OS. La théorie se mêlera aux exemples pour essayer de faire comprendre au lecteur l'importance de cette communication.

Vous pouvez aussi consulter cet article au format PDF.

Si ce lien ne fonctionne pas chez vous, utilisez celui-ci.

I - Introduction

Le vocabulaire sera introduit au fur et à mesure de son utilisation afin de ne pas perdre trop de lecteurs. Si vous vous perdez tout de même, n'hésitez pas à faire des remarques pour éclaircir les points flous. Ca sera utile pour tout le monde.

La communication entre processus est indispensable dans un système d'exploitation mais cet échange d'information n'est pas sans poser certains problèmes dont voici les 3 principaux :

- La transmission de l'information, par quel moyen la mettre en #uvre ?
- La gestion des sections dites critiques où plusieurs processus veulent accéder en même temps à une ressource. Nous prendrons souvent l'exemple d'un processus Consommateur (le processus lié à l'impression par exemple) qui doit attendre que le processus Producteur (un traitement de texte quelconque) ait produit un message. Pour illustrer cet exemple, je vous conseil de vous reportez à cette page : http://www.ccs.neu.edu/home/kenb/g712/res/example.gif
- La synchronisation de processus. En effet, il est souvent important pour plusieurs processus de s'échanger de l'information, et dans de nombreux cas, il est nécessaire d'établir une synchronisation pour avoir un résultat cohérent.

Les Inter Process Communication (IPC) ont pour but de répondre à toutes ces exigences de manière efficace. Ils sont apparus dans les versions System V d'Unix et sont aujourd'hui présents dans toutes les variantes de ce système (dont Linux bien sûr). Cette présentation s'attachera d'abord à présenter les concepts généraux sans entrer dans les détails d'implémentation des IPC Unix qui seront tout de même partiellement abordés à la fin de cet article.

II - Conditions de concurrence

Les situations de concurrence (race conditions) se rencontrent lorsque plusieurs processus disposent simultanément d'une même ressource (fichier, périphérique, mémoire), alors que chacun d'eux pense en avoir l'usage exclusif. Il faut savoir que les situations de concurrence sont encore aujourd'hui la cause de nombreux bugs dans certains programmes. Ce n'est donc pas un problème à prendre à la légère.

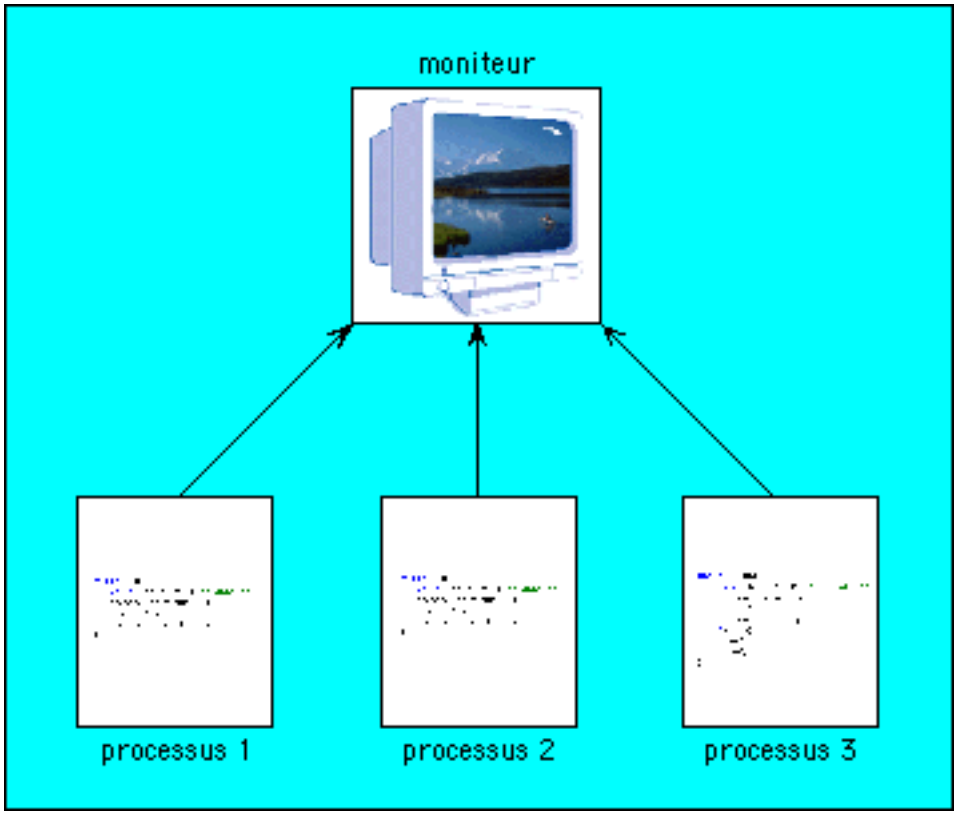
Pour illustrer ces propos, je vais prendre l'exemple bien connu du producteur-consommateur dans lequel un processus produit des messages qu'il dépose dans une file et où un autre processus retire ces messages. Cette situation s'applique pour une imprimante par exemple ou différents processus vont mettre des documents dans le répertoire de spoule de l'imprimante et où le démon d'impression va venir retirer ces messages. Dans ce cas, on utilise généralement 2 variables : " in " qui indique la prochaine entrée libre dans le répertoire de spoule et " out " qui indique le prochain fichier à imprimer.

Imaginons un cas extrême avec 2 processus producteurs qui veulent mettre un fichier dans le répertoire de spoule. Le processus A commence à s'exécuter et lit la valeur de la variable in (in vaut 7 à cet instant) qu'il stocke dans une variable locale à A. Suite à une interruption de l'horloge système, le processeur passe ensuite la main au processus B qui stocke également la valeur de in dans une variable puis insère son fichier et change la valeur de in (in passe à 8). Finalement, le processus A s'exécute de nouveau. Il garde l'ancienne valeur de in (c'est-à-dire 7) et écrase le fichier précédemment inséré par le processus B avant de changer) à son tour la variable in qui vaudra 8 au lieu de 9 puisque 2 fichiers auraient du être insérés. Voici un exemple de conditions de concurrence.

III - Les solutions aux situations de concurrence

III.A - Exclusion mutuelle

L'exclusion mutuelle est une technique permettant de s'assurer que si un processus utilise une ressource, les autres processus ne peuvent pas utiliser cette même ressource. Prenons l'exemple de 3 processus qui veulent chacun écrire un message à l'écran dans le même intervalle de temps. Si chacun des processus écrit son message à l'écran sans faire attention aux autres, vous n'aurez jamais une phrase entière affichée à la suite mais des bouts de phrases, ce qui sera très pénible à lire.



Ceci nous amène à définir une section critique (ou région critique) qui est une zone de code où l'on accède à une ressource partagée (l'écran dans notre exemple). L'objectif est donc d'éviter que plusieurs processus se retrouve dans la même section critique. Ceci est un problème primordial pour la conception d'OS ou d'applications multi-tâches. D'autres contraintes sont à prendre en compte comme nous le verrons plus tard (l'interblocage est un bon exemple) mais intéressons nous d'abord aux solutions existantes pour permettre l'exclusion mutuelle.

III.B - Désactivation des interruptions

Il est possible de désactiver toutes les interruptions lorsqu'un processus entre dans une section critique. Ainsi les interruptions provenant de l'horloge système seront également désactivées. L'horloge système est en fait le maître d'orchestre du microprocesseur et de tous les composants d'un ordinateur. Elle fournit des impulsions à intervalles réguliers qui déterminent l'allocation du processeur à un processus. Si l'on désactive les interruptions, le processus sera le seul à s'exécuter.

Cette méthode n'est pas vraiment conseillée lors du développement d'un logiciel car il est possible d'oublier de

réactiver les interruptions (et là, c'est le drame 9). Elle n'est utilisé qu'à de rares endroits dans un système d'exploitation.

III.C - Le verrou

Il s'agit d'une solution logicielle. La définition du verrou est donnée ci-dessous : Variable partagée, binaire, qui permet à un processus de savoir s'il peut entrer dans sa " section critique ", i.e. sa portion de code où il manipule des ressources partagées. Le verrou permet de créer une exclusion mutuelle. (© Tanenbaum).

En fait, le principe est intéressant mais le verrou seul présente un inconvénient déjà exposé dans le paragraphe sur les conditions de concurrence. Si un processus A lit la valeur du verrou et reprend la main après qu'un processus B est modifié la valeur de verrou, le processus A gardera son ancienne valeur. Il s'agit d'un cas extrême mais on ne peut pas se permettre de l'ignorer lorsque l'on fait un système d'exploitation.

III.D - Alternance stricte

Les processus vont exécuter tour à tour leur propre section critique. L'alternance doit être respectée peu importe le temps d'exécution de chaque processus. L'exemple donné ci-dessous concerne 2 processus (a) et (b). Lorsque le processus (a) quitte sa section critique, il positionne turn à pour permettre au processus (b) d'entrer dans sa section critique.

```
while (TRUE) {
while (tour != 0) /* attente */
section_critique() ;
tour = 1 ;
section_noncritique() ;
}
(a)

while (TRUE) {
while (tour != 1) /* attente */
section_critique() ;
tour = 0 ;
section_noncritique() ;
}
(b)
```

Si cette solution évite les situations de concurrence, un processus peut se retrouver bloqué (en attente d'entrer dans sa section critique) alors que l'autre n'est pas non plus dans sa section critique. La consommation en temps CPU peut alors augmenter de manière importante.

III.E - Solution de Peterson

Sachez que, pour l'anecdote, ce fut d'abord T. Dekker (1965) qui proposa une solution logicielle utilisant plusieurs variables de verrouillage. Bien que valable, cette solution était en fait assez complexe à mettre en #uvre. En 1981 Peterson proposa un algorithme plus simple pour des résultats identiques (algorithme extrait du livre de Tanenbaum cité en référence) :

```
# define  FALSE   0
# define  TRUE    1
# define   N   2 /* nombre de processus */
int turn ;          /* A qui le tour ? */
int interested [N] ; /* tous à 0 au départ */

enter_region (int process)
{
        int other; /* nombre des autres processus */
        other =1 - process ; /* l'opposé du processus */
```

```
        interested[process] = TRUE ; /* montre que le process est intéressé */
        turn = process ; /* définit indicateur */
        while (turn == process AND interested[other] == TRUE ) ;
}

leave_region (int process)
{
        interested[process] = FALSE ; /* permet de quitter la section critique */
}
```

Cette méthode exige de connaître le nombre maximum de processus intéressés. Elle exige également l'assignation d'un identifiant à chaque processus.

III.F - Instruction TSL (test and set lock)

Cette instruction est une opération câblée du processeur. Par définition, c'est donc une opération atomique qui ne peut pas être interrompue par aucun évènement. Elle s'utilise sous la forme :

TSL RX, LOCK

Elle lit le contenu du mot mémoire lock qu'elle place dans le registre TX, puis stocke une valeur différente de 0 à l'adresse mémoire lock.

Exemple d'utilisation de TSL

```
int lock = 0;
while (1) {
   while (!TSL(lock)]); /* on attend */
   /**** section critique ****/
   lock = 1; /* on sort de la section critique */
   /**** section non critique ****/
}
```

IV - Sommeil et activation

Il faut savoir que dans la plupart des systèmes d'exploitation, chaque tâche possède une priorité (plus la tâche est importante et plus sa priorité est forte habituellement). Ceci est très utile pour l'ordonnancement des tâches mais c'est aussi problématique vis-à-vis de ce que l'on appelle l'inversion de priorité qui survient lorsqu'une tâche est suspendue dans l'attente d'une ressource contrôlée par une tâche moins prioritaire.

Prenons un exemple avec 2 processus : B et H, B possédant une priorité plus faible que H. Les règles d'ordonnancement font que H s'exécute à chaque fois qu'il est dans l'état prêt (pour simplifier, disons que l'état prêt correspond à l'un des 3 états fondamentaux d'un processus : élu, prêt et endormi).

A un instant donné, B est dans sa section critique alors que H est dans l'état prêt. H prend alors le processeur et entre en attente active sur une ressource devant être libéré par B. Hors, B ne prendra jamais la main car H est occupé et il est plus prioritaire. Nous avons encore une fois à faire avec une boucle infinie qui va consommer tout le temps CPU.

C'est à cet instant qu'entre en jeu les primitives SLEEP et WAKEUP qui permettent de réveiller et d'endormir un processus (SLEEP fait donc passer le processus à l'état endormi.

IV.A - Problème du producteur-consommateur

Reprenons le problème du producteur consommateur introduit dans la première partie de cet article en y ajoutant les primitives sleep et wakeup. Si le tampon est plein, le producteur passera à l'état endormi alors que le si tampon est vide, le consommateur passera également à l'état endormi.

```
#define N 100 /* nombre d'emplacements dans le tampon */
int count = 0 ; /* nombre d'éléments dans le tampon */

void producer ()
{
  while (TRUE) {
    produce_item();
    if (count == N)
      sleep () ; /* si tampon plein, entre en sommeil */
    put_item(); /* produit l'élément */
    count = count + 1;
    if (count == 1)
      wakeup(consumer) ; /* si tampon vide avant incrémentation, on réveil le consommateur */
  }
}

void consumer () {
  while (TRUE) {
    if (count == 0)
      sleep() ; /* si tampon vide, entre en sommeil */
```

```
      remove_item();
      count = count - 1;
      if (count == N-1)
        wakeup (producer); /* si tampon plein avant décrémentation, on réveil le producteur */
      consume_item(); /* imprime l'élément */
    }
  }
```

Il résulte 2 problèmes du code ci-dessus (nous avons déjà parlé du premier problème) :

- L'accès à la variable count n'étant pas protégé, ceci peut entraîner des incohérences dans les valeurs de la variable (suite à un ordonnancement un peu particulier de l'ordonnanceur déjà abordé auparavant).

- Signal perdu : Un signal wakeup envoyé à un processus qui ne dort pas est perdu. Cela arrive si le consommateur vient de lire le décompte pour constater qu'il est à 0 mais l'ordonnanceur décide alors d'exécuter le producteur alors que le consommateur n'est pas dans l'état endormi (mais dans l'état prêt à être exécuté).

V - Les sémaphores

Les sémaphores ont été inventés en 1965 par Dijkstra (mathématicien et informaticien néerlandais) et peuvent être définis comme des compteurs permettant de gérer les accès à des ressources communes. Le premier rôle des sémaphores est la synchronisation et l'exclusion mutuelle entre processus.

Un sémaphore est en fait une variable à valeurs entières non négatives et accessible au travers de 2 opérations essentielles :

- Opération P (Proberen = tester). Si la ressource n'est pas disponible (variable R égalé à 0) alors le processus sera mis à l'état endormi.

- Opération V (Verhogen = incrémenter). On incrémente la variable d'une unité.

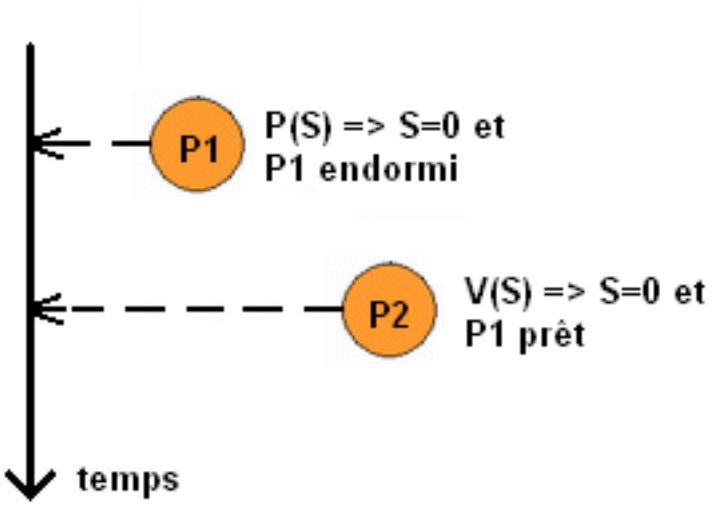
L'implémentation de ces 2 méthodes repose sur :

- l'atomicité des opérations (la suite d'opérations est non interruptible, sauf en cas de cas extrême comme la panne de courant :).

- l'existence d'un mécanisme permettant de mémoriser les demandes non satisfaites avec les opérations P pour réveiller le processus en attente le moment venu (lorsque qu'un autre processus aura incrémenter la variable sur laquelle notre processus endormi était en attente).

V.A - Exemple 1

Dans l'exemple donné ci-dessous avec 2 processus P1 et P2, nous définissons deux sémaphores, Sa et Sb, l'un signalant que P1 a fini ce qu'il devait faire et que c'est maintenant au tour de P2 à s'exécuter.




```
var Sa: semaphore := 1
var Sb: semaphore := 0

process P1
Begin
  P( Sa )
  // Code de P1
```

```
    V( b )
  End

  process P2
  Begin
    P( Sb )
    // Code de P2
    V( a )
  End
```

Dans l'exemple précédent, vous avez dû remarquer qu'on utilisait des sémaphores binaires, c'est-à-dire n'ayant que deux états :

- 0 verrouillé (ou occupé),

- 1 déverrouillé (ou libre).

Un sémaphore binaire est également qualifié de mutex (mutal exclusion) qui permet l'exclusion mutuelle et la synchronisation entre threads.

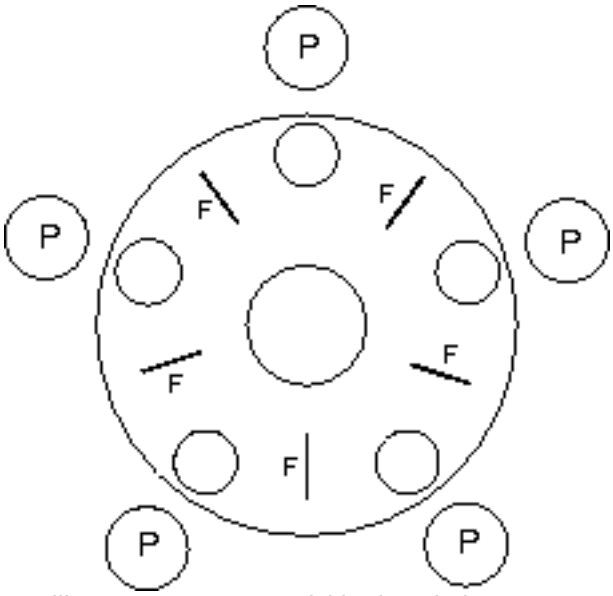
V.B - Exemple 2

L'exemple ci-dessous reprend encore une fois le modèle du producteur-consommateur que vous connaîtrez sûrement dans les moindres détails à la fin de cet article :-) .

```
  #define N 100 /* nombre d'emplacements dans le tampon */

  semaphore mutex = 1; /* contrôle l'accès à la section critique */
  semaphore production = N; /* nombre de places vides */
  semaphore consommation = 0; /* nombre de places occupées */

  void consumer () {
  {
    while (TRUE)
    {
      P(consommation); /* une place pleine en moins */
      P(mutex); /* section critique */
      retirer_case(); /* retire un élément dans le tampon */
      V(mutex); /* fin section critique */
      V(production); /* une place vide en plus */
    }
  }

  void producer ()
  {
    while (TRUE)
    {
      P(production); /* une place vide en moins */
      P(mutex); /* section critique */
      remplir_case();/* place un élément dans le tampon */
      V(mutex); /* fin section critique */
      V(consommation); /* une place pleine en plus */
    }
  }
```

VI - Quelques problèmes classiques

VI.A - Dîner des philosophes



Ce problème a été imaginé par E. Dijkstra en 1965. En voici la description :

Une table circulaire a été dressée à l'intention de cinq philosophes avec cinq chaises, cinq assiettes et cinq fourchettes. Chaque philosophe peut prendre une chaise et manger s'il dispose de deux fourchettes de chaque côté de son assiette. Après avoir mangé, il pose les fourchettes à leur place et quitte la table. Il peut faire ce cycle d'opérations tant qu'il le veut. Comment faire pour que les philosophes mangent autant de fois qu'ils le veulent ? De fait, un seul philosophe peut manger à la fois. Si deux philosophes, persistent à vouloir manger en même temps, aucun ne peut le faire.

Dans un premier temps, voici ci-dessous une mauvaise solution :

```
# define N  5 /* nombre de philosophes */

philosophe(int i) /* i : numéro du philosophe */
{
  while (TRUE)
  {
     penser( ) ;
     prendre_fourchette(i) ; /* prend la fourchette gauche */
     prendre_fourchette( (i+1)%N) ; /* prend la fourchette droite */
     manger( ) ;
     poser_fourchette( i ) ; /* repose la fourchette gauche */
     poser_fourchette((i+1)%N) ; /* repose la fourchette droite */
  }
}
```

Supposez que les cinq philosophes prennent leur fourchette en même temps. Aucun d'eux ne pourra prendre sa fourchette droite, ce qui conduira à un interblocage.

Présentation d'une solution au problème des philosophes

Voici donc une solution à ce problème qui évitera les inter-blocages et autorisera un parallélisme maximal pour un

nombre arbitraire de philosophes. On utilise un tableau contenant l'état du philosophe (mange, pense, veut prendre les fourchettes). Un philosophe donné ne peut manger que si aucun de ses voisins ne mange.

```
# define N  5 /* nombre de philosophes */

# define GAUCHE  (i-1)%N
# define DROITE  (i+1)%N
# define PENSE  0
# define FAIM  1
typedef int sémaphore ;
int état [N] ;
sémaphore mutex = 1 ;
sémaphore s[N] ;

philosophe(int i) /* i : numéro du philosophe */
{
  while( TRUE)
  {
     penser ( ) ;
     prendre-fourchette( i ) ; /* prendre 2 fourchettes ou bloque */
     manger( ) ;
     poser-fourchette( i ) ; /* repose 2 fourchettes */
  }
}

prendre-fourchette(int i)
{
  down (mutex) ; /* entre en section critique */
  état [i] = FAIM ;
  test(i) ; :* tente de prendre les 2 fourchettes */
  up (mutex) ; /* quitte la section critique */
  down (s[i]) ; /* bloque s'il n'a pas pu prendre les fourchettes */
}

poser-fourchette(int i)
{
  down (mutex) ; /* entre en section critique */
  état [i] = PENSE ; /* philosophe a fini de manger */
  test(GAUCHE) ; /* regarde si voisin de gauche peut manger */
  test(DROITE) ; /* regarde si voisin de droite peut manger */
  up (mutex) ; /* quitte la section critique */
}

test(int i) ;
{
  if (état [i] ==FAIM AND état [GAUCHE] != MANGE AND état [DROITE] != MANGE )
 {
   état [i] =MANGE ;
   up( s[i] ) ;
 }
}
```

VII - Implémentation des IPC sous UNIX

Les IPC étant apparus avec les versions System V d'Unix, il me semble important d'aborder l'implémentation des IPC sous UNIX-Like.

Pour répondre aux éxigences de la communication interprocessus qui ont été abordés dans la première partie de cet article, 3 catégories d'IPC ont été implémentées :

- Les sémaphores : Compteurs permettant de gérer les accès à des ressources communes. Pour rappel, ils permettent la synchronisation et l'exclusion mutuelle entre processus via 2 opérations P et V sauf que, sous Linux, il existe une opération Z qui permet également de remettre à 0 un sémaphore.

- Les files de messages

Utilise le mécanisme de boite aux lettres afin d'assurer la communication d'informations entre les processus.

- La mémoire partagée : il s'agit d'une zone de mémoire partageable entre plusieurs processus.

Sous Linux, le système assure la gestion des IPC avec une table spécifique à l'objet (il y a donc une table pour chaque type d'IPC) et non pas par des fichiers, ce qui fait que les IPC sont un peu l'exception sous le monde UNIX dans lequel habituellement, "tout n'est que fichier".

VII.A - Gestion des clefs

L'identification des IPC se fait grâce à des clefs qui sont gérées par le système d'exploitation (il se charge de la création, de la récupération et de la suppression). Sous linux, chaque type d'objets possède son propre lot de clés (la même clé peut être utilisé par un sémaphore et par une file de messages par exemple).

Ainsi, avant de pouvoir faire quoi que ce soit avec un IPC, il faut disposer de la clef utilisée lors de sa création. Remarquez que pour construire la clef, le mécanisme de base d'UNIX utilise l'inode d'un fichier (l'inode de chaque fichier étant unique).

VII.B - Les files de messages

Il s'agit d'un implémentation du concept de boite aux lettres sous UNIX (BSD gère cette fonction avec un mécanisme de socket locale).Contrairement au canal de communication sous UNXI, la file de message ne présente pas de notion de flot continu d'octets mais plutôt une notion de paquets ou une lecture extrait exactement le produit d'une écriture.

Il est ainsi possible d'envoyer depuis un processus un message destiné à un autre processus (voir à plusieurs). Autre concept important, il est possible de réaliser une extraction sélective selon le type du message.

VII.C - La mémoire partagée

Cette zone de mémoire partageable entre plusieurs processus permet d'échanger rapidement des données entre processus : aucun transfert entre processus n'est nécessaire puisque lorsqu'un processus change le contenu de la mémoire, l'autre processus n'aura qu'à lire cette même mémoire.

VIII - Références

- Systèmes d'exploitation 2ième édition de Andrew Tanenbaum
- Cours sur les sémaphores
- Threads et sémaphores en java