Synchronisation des processus

Introduction

La communication entre processus (en anglais IPC, InterProcess Communication) est un aspect important de la gestion des processus ; elle se traduit généralement en termes de synchronisation. Pour introduire cette notion et les problèmes qui lui sont associés, prenons l'exemple simple d'un spoule d'impression ; quand un processus veut imprimer un fichier, il place le fichier dans un répertoire de spoule accessible à tous les processus (ressource partagée). Un processus spécial appelé démon (daemon) de spoule regarde continuellement le répertoire et quand il trouve quelque chose, il imprime. Le répertoire de spoule est donc un entrepôt dans lequel des processus placent des noms de fichiers et dans lequel le démon de spoule retire un nom de fichier. Pour la gestion du spoule, deux variables sont accessibles à tous les processus : IN qui enregistre la prochaine case libre du répertoire de spoule (considéré ici infini par simplification), OUT qui enregistre la prochaine case où se trouve le fichier à imprimer.

Supposons qu' un processus A lise IN qui contient 7 (ce qui signifie que la valeur 7 est copiée dans une variable locale). Admettons que le processeur soit retiré à A (fin du quantum de temps), un processus B devenant actif. B lit IN et met 8 dans IN après avoir remplit l'élément 7. Quand A redevient actif, il remplit l'élément 7 avec le nom du fichier qu'il veut imprimer écrasant ainsi ce que le processus B avait mis. Quand le processus B redeviendra actif, il ne pourra plus jamais imprimer son fichier.

Cette situation gênante provient du fait que A et B sont dans des situations de **concurrence**. Pour éviter ces situations de concurrence, il faut empêcher que plus d'un processus ne puisse accéder à des **ressources partagées**. Il faut donc réaliser **l'exclusion mutuelle** dans des parties de programme qui concernent l'accès à des ressources partagées. Ces parties sont des **sections critiques**.

Les conditions de base de l'exclusion mutuelle sont au nombre de 4 :

1 - Deux processus ne peuvent simultanément être dans leur section critique.
2 - La vitesse ou le nombre de processeurs ne peuvent avoir d'influence
3 - Un processus en exécution en dehors de sa section critique ne peut bloquer d'autres processus
4 - Un processus ne doit pas attendre indéfiniment à l'entrée de sa section critique

En fait, il y a trois cas à considérer dans l'interaction entre processus :

- les processus n'interagissent pas l'un sur l'autre : il s'agit de simple **compétition**. Les problèmes qui peuvent se poser sont l'exclusion mutuelle, l'interblocage (sur ressources successivement réutilisables) ou la famine (les processus s'exécutent mais ne font rien).
- les processus interagissent indirectement l'un sur l'autre : il s'agit de **coopération par partage**. Les problèmes qui peuvent se poser sont l'exclusion mutuelle, l'interblocage (sur ressources successivement réutilisables) ou la famine.
- les processus interagissent directement l''un sur l'autre : il s'agit de **coopération par communication**. Les problèmes qui se posent sont l'interblocage (sur ressources consommables) ou la famine.

Exclusion mutuelle avec attente active

Passons en revue quelques idées permettant de gérer le problème de l'exclusion mutuelle.

désactivation des interruptions

On peut penser à désactiver les interruptions : une nouvelle interruption n'aura pas de conséquence sur le processus courant. Mais cela est très dangereux :

- cela gêne considérablement le système d'exploitation
- le processus courant peut, pour diverses raisons, ne pas réactiver les interruptions

De plus sur un système à plusieurs processeurs le dispositif ne marche pas. Conclusion : ce n'est pas la bonne solution !

variables "verrous"

Soit une variable globale Lock, correspondant à une ressource donnée et initialisée à 0. Supposons qu'un processus désire entrer dans sa section critique, c'est à dire désire utiliser la ressource associée à Lock :

- si Lock = 0, alors le processus met Lock à 1 et entre dans sa section critique ; après utilisation de la ressource, le processus met Lock à 0 et sort de sa section critique.
- si Lock = 1, le processus attend que Lock soit remise à 0 (attente active)

Mais difficulté : entre le moment où le processus lit Lock et le moment où il écrit dans Lock (lecture et écriture étant deux opérations distinctes successives), un autre processus peut intervenir ! Conclusion : ce n'est pas la bonne solution !

Alternance stricte

On définit une variable partagée entière Turn, initialisée à 0 et globale.

Considérons 2 processus : le processus P0 et le processus P1. Chacun des deux processus ne peut entrer dans sa section critique que si la valeur de Turn est égale à son numéro (0 ou 1).

Supposons que le processus P0 lise Turn et constate que sa valeur est 0 ; il entre dans sa section critique. Si le processus P1 lit à son tour Turn (valeur 0), il doit attendre dans une boucle le passage de Turn à 1 (attente active). Quand le processus P0 sort de sa section critique, il met Turn à 1. Le processus P1 peut alors entrer dans sa section critique; quand il en sortira, il mettra Turn à 0. Cette méthode correspond aux codes suivants pour les processus P0 et P1 :

```
processus P0
début
  Répéter indéfiniment
    TantQue Turn est différent de 0 faire
       // attente active
    FinTantQue
    section critique ;
    Turn = 1;
    section non critique ;
  FinRépéter
fin
```

```
processus P1
début
  Répéter indéfiniment
    TantQue Turn est différent de 1 faire
      // attente active */
    FinTantQue
    section critique ;
    Turn = 0;
    section non critique ;
  FinRépéter
fin
```

Mais imaginons que le processus P1 s'arrête; le processus P0 pourra entrer encore une fois dans sa section critique, mais sera ensuite bloqué (violation de la condition 4). Plus encore, on peut imaginer que le processus P1 boucle indéfiniment dans sa section non critique, le processus P0 finira également par être bloqué (violation de la règle 3). Conclusion : ce n'est pas la bonne solution !

algorithme de Dekker

Dekker a, le premier, trouvé une solution au problème pour deux processus concurrents. La solution est toutefois assez lourde.

Supposons que les deux processus soient repérés par les numéros 1 et 2. Trois variables globales sont utilisées :

- Chouchou qui ne prend que deux valeurs 1 et 2; initialisation à 1;
- P1_veut_entrer et P2_veut_entrer qui ont des variables booléennes; initialisation à FAUX;

Le code des processus P1 et P2, pour l'algorithme de Dekker est le suivant :

```
processus P1
début
  Répéter indéfiniment
    P1_veut_entrer = VRAI ;
    TantQue P2_veut_entrer est VRAI faire
      Si Chouchou = 2 alors
        P1_veut_entrer=FAUX ;
        TantQue Chouchou = 2 faire
          attendre ;
        FinTantQue
        P1_veut_entrer = VRAI ;
      FinSi
    FinTantQue
    section critique ;
    Chouchou = 2 ;
    P1_veut_entrer = FAUX ;
    section non critique ;
  FinRépéter
fin
```

```
processus P2
début
  Répéter indéfiniment
    P2_veut_entrer = VRAI ;
    TantQue P1_veut_entrer est VRAI faire
      Si Chouchou = 1 alors
        P2_veut_entrer=FAUX ;
        TantQue Chouchou = 1 faire
          attendre ;
        FinTantQue
        P2_veut_entrer = VRAI ;
      FinSi
    FinTantQue
    section critique ;
    Chouchou = 1 ;
    P2_veut_entrer = FAUX ;
    section non critique ;
  FinRépéter
fin
```

algorithme de Peterson

Après Dekker, Peterson (1981) a trouvé une solution assez simple au problème de l'exclusion mutuelle. Deux variables globales sont utilisées : Turn, variable entière, et Desir[i], vecteur de valeurs booléennes (on ne considère ici que le cas de deux processus). L'algorithme de Peterson fait intervenir deux procédures : entrer_region() et quitter_region() :

```
entrer_region(numéro_de_processus)
début
  autre = 1 - numéro_de_processus ;
  Desir[numéro_de_processus] = VRAI ;
  Turn = autre ;
  TantQue (Turn = autre et Desir[autre]=VRAI) faire
     attendre ;
  FinTantQue
fin
```

```
quitter_region(numéro_de_processus)
début
  Desir[numéro de processus]=FAUX ;
fin
```

Avant d'entrer dans leur région critique, chaque processus appelle entrer_region() en y passant comme paramètre son numéro de processus, 0 ou 1. Il peut alors entrer ou il doit attendre. A la sortie de la section critique, chaque processus doit invoquer quitter_region().

instruction TAS

Les nouveaux processeurs fournissent des instructions Test And Set qui effectuent une lecture et une écriture successives de manière indivisible (rien ne peut s'intercaler entre la lecture et l'écriture).

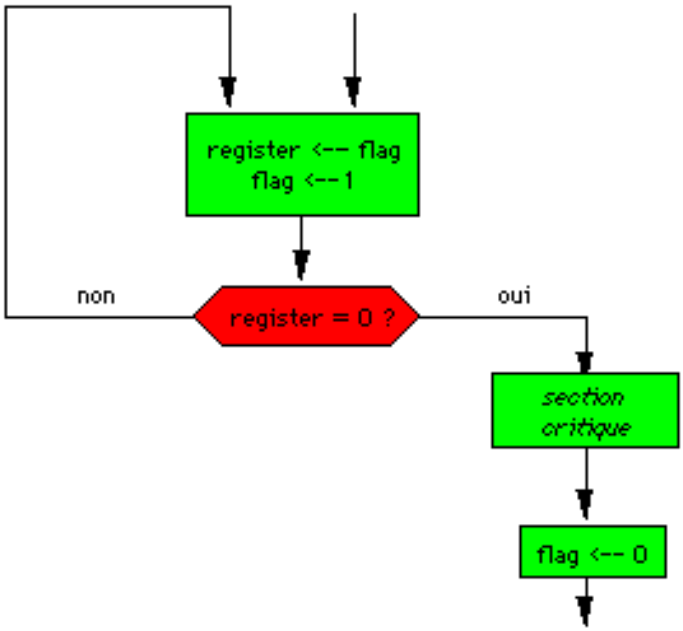
TAS :

- lecture d'une case mémoire et copie dans un registre
- écriture de 1 dans cette case mémoire

Soit une variable globale, flag , initialisée à 0. Comme précédemment, un processus doit invoquer entrer_region pour entrer dans sa section critique et quitter_region pour sortie de sa section critique . Ces deux procédures sont données ci-dessous en langage d'assemblage :

```
entrer_region : TAS register,flag
                CMP register,#0
                JNZ entrer_region
                RTS
```

```
quitter_region : MOVE flag,#0
                 RTS
```



Conclusion générale : les algorithmes de Dekker, de Peterson, l'utilisation de TAS apportent des solutions viables, mais au prix **d'attentes actives** qui consomment du temps CPU !

Un inconvénient important également vient des règles d'ordonnancement. Supposons que l'on ait deux processus H, bloqué et L, actif sur une machine monoprocesseur ; supposons que H possède une priorité plus forte que L et que les règles d'ordonnancement soient telles que si H est prêt, il doit devenir actif. Supposons que L soit dans sa section critique ; à ce moment H passe de l'état bloqué à l'état prêt : il faut que L passe à l'état prêt et H à l'état actif; mais H restera indéfiniment dans une attente active car L n'est pas sorti de sa section critique et L ne reviendra pas actif tant que H le sera !

Sleep et Wakeup

Plutôt que des solutions avec attente active qui laissent les processus prêts, il faut rechercher des solutions qui bloquent les processus. Une des plus simples solutions est l'utilisation des appels systèmes SLEEP et WAKEUP :

- SLEEP bloque le processus appelant jusqu'à son réveil par un autre processus
- WAKEUP réveille le processus dont le numéro lui est passé en paramètre.

Pour étudier le fonctionnement de SLEEP et WAKEUP, on prendra l'exemple célèbre des producteurs et des consommateurs, du moins sous une forme simplifiée. Deux processus partagent un buffer commun de taille finie. L'un des processus, le producteur, dépose une information dans le buffer; l'autre processus, le consommateur, retire une information du buffer. Si le buffer est vide, le consommateur attend tant que rien n'arrive dans le buffer; de même, si le buffer est plein, le producteur attend tant que le consommateur n'a pas retiré quelque chose.

En supposant qu'une information constitue une unité, on utilise une variable count pour compter le nombre d'unités qui se trouvent dans le buffer à un moment donné. Soit N le nombre maximal d'unités (capacité du buffer). Le programme du producteur testera count ; si count = N, il attendra. Le consommateur, de même, consultera count ; si count = 0, il attendra.

```
processus producteur :
début
  Répéter indéfiniment
    produire (item) ;
    Si count = N alors
      sleep() ;
    FinSi
    entreposer (item) ;
    count = count +1 ;
    Si count = 1 alors
      wakeup(consommateur) ;
    FinSi
  FinRépéter
fin
```

```
processus consommateur :
début
  Répéter indéfiniment
    Si count = 0 alors
      sleep() ;
    FinSi
    retirer(item) ;
    count = count - 1 ;
    Si count = N-1 alors
      wakeup(producteur) ;
    FinSi
    consommer(item) ;
  FinRépéter
fin
```

Il y a une faille à ce système qui paraît séduisant . Supposons que le buffer soit vide et que le consommateur soit actif : il voit que count = 0, mais à ce moment, pour une raison quelconque, le dispatcher lui enlève le processeur et le met en état prêt. Le producteur est, supposons, maintenant actif; il met une unité dans le buffer et croyant le consommateur endormi, envoie le signal de réveil avec wakeup; mais comme le consommateur ne dort pas, le signal est perdu. Lorsque le consommateur sera de nouveau actif, il s'endormira sous l'action de sleep car il croit toujours que le buffer est vide puisque, pour lui la dernière valeur lue de count est 0. Le producteur va alors continuer à remplir le buffer jusqu'à ce qu'il soit plein et s'endormira à son tour. Au bout du compte, on va avoir deux processus bien endormis pour toujours !

On peut améliorer les choses en ajoutant un "wakeup waiting bit" : quand un signal wakeup est envoyé à un processus qui est déjà réveillé, le bit est positionné à 1. Plus tard, quand le processus doit dormir, le bit est remis à 0, mais le processus ne s'endort pas. Ce système semble marcher mais pour plus de deux processus, cela devient plus difficile.

Sémaphores

Les sémaphores ont été inventés par Dijkstra (1965) pour justement compter le nombre de wakeup envoyés. Une variable de type nouveau, le sémaphore, est définie; elle peut prendre la valeur 0 si aucun wakeup n'a été répertorié et une valeur positive si plusieurs wakeup sont en "attente". Le sémaphore correspond à une ressource convoitée et à une file d'attente de processus attendant la possession de cette ressource.

Sleep et Wakeup ont été généralisées en P et V. On désigne quelquefois P (Proberen) par Down ou Wait et V

(Verhogen) par Up ou Signal. On peut considérer un sémaphore S comme une variable entière, initialisée à une valeur non négative, sur laquelle agissent deux primitives P et V définies comme suit et à laquelle est associée une file d'attente de processus bloqués.

```
P(S)
début
  S = S - 1;
  Si S < 0 alors
     bloquer le processus ;
  FinSi
fin
```

```
V(S)
début
  S = S + 1 ;
  Si S <= 0 alors
     réveiller un processus bloqué de la file d'attente ;
  FinSi
fin
```

Les deux primitives P et S doivent être indivisibles, c'est à dire s'exécuter de bout en bout sans aucune interruption. Un cas particulier de sémaphores est représenté par les sémaphores binaires, qui ne prennent que les valeurs 0 ou 1 et dont la définition est données ci-dessous :

```
PB(S)
début
  Si S=1 alors
     S = 0 ;
  sinon
     bloquer le processus ;
  FinSi
fin
```

```
VB(S)
début
  Si la file d'attente est vide alors
     S = 1 ;
  sinon
     réveiller un processus de la file ;
  FinSi
fin
```

Pour illustrer l'utilisation des sémaphores, appliquons les au problème des producteurs et des consommateurs. 3 sémaphores sont utilisés

- mutex initialisé à 1 ; qui contrôle l'accès à la section critique ;
- vides initialisé à N ; qui compte le nombre de places vide dans le buffer ;
- pleines initialisé à 0 ; qui compte le nombre de places pleines dans le buffer ;

```
processus producteur
début
  Répéter indéfiniment
     produire(item) ;
     P(vides) ;
     P(mutex) ;
     déposer(item) ;
     V(mutex) ;
     V(pleines) ;
  FinRépéter
fin
```

```
processus consommateur
début
  Répéter indéfiniment
     P(pleines) ;
     P(mutex) ;
     retirer(item) ;
     V(mutex) ;
     V(vides) ;
     consommer(item) ;
  FinRépéter
fin
```

Les sémaphores sont utilisés ici pour deux usages :

- l'exclusion mutuelle avec le sémaphore "mutex" ;
- la synchronisation de processus avec les sémaphores "vides" et "pleines".

Compteurs d'événements

Les compteurs d'événements sont une autre façon de gérer le problème des producteurs et des consommateurs. Une variable E d'un nouveau type, le compteur d'événements, est définie. Trois opérations agissent sur E :

- Read (E) lecture de la valeur courante de E
- Advance (E) incrémentation de E
- Await (E, n) attente tant que E < n

Contrairement aux sémaphores, un compteur d'événements ne peut qu'augmenter et jamais diminuer. Pour résoudre le problème des producteurs et des consommateurs, on utilise deux compteurs d'événements :

- in : cumul du nombre d'objets que le producteur a mis dans le buffer
- out : cumul du nombre d'objets retirés du buffer par le consommateur

On doit avoir 0 £ in - out £ N.

- Comportement du producteur : A chaque création d'objet, le producteur vérifie s'il y a de la place dans le buffer à l'aide de la fonction Await.
- Comportement du consommateur : Avant de retirer le $k^{ième}$ objet, le consommateur attend que in atteigne la valeur k.

Les compteurs d'événements sont déclarés de manière globale et correspondent à un type de données particulier assimilable au type entier, event_counter et initialisés à zéro : event_counter in = 0 ; event_counter out = 0.

Les processus producteur et consommateur sont les suivants :

```
processus producteur
début
   sequence=0 ;
   Répéter indéfiniment
      produire(item) ;
      sequence = sequence + 1 ;
      await(out, sequence - N) ;
      déposer(item) ;
      advance(in) ;
   FinRépéter
fin
```

```
processus consommateur
début
   sequence = 0 ;
   Répéter indéfiniment
      sequence = sequence + 1 ;
      await(in, sequence) ;
      retirer(item) ;
      advance(out) ;
      consommer(item) ;
   FinRépéter
fin
```

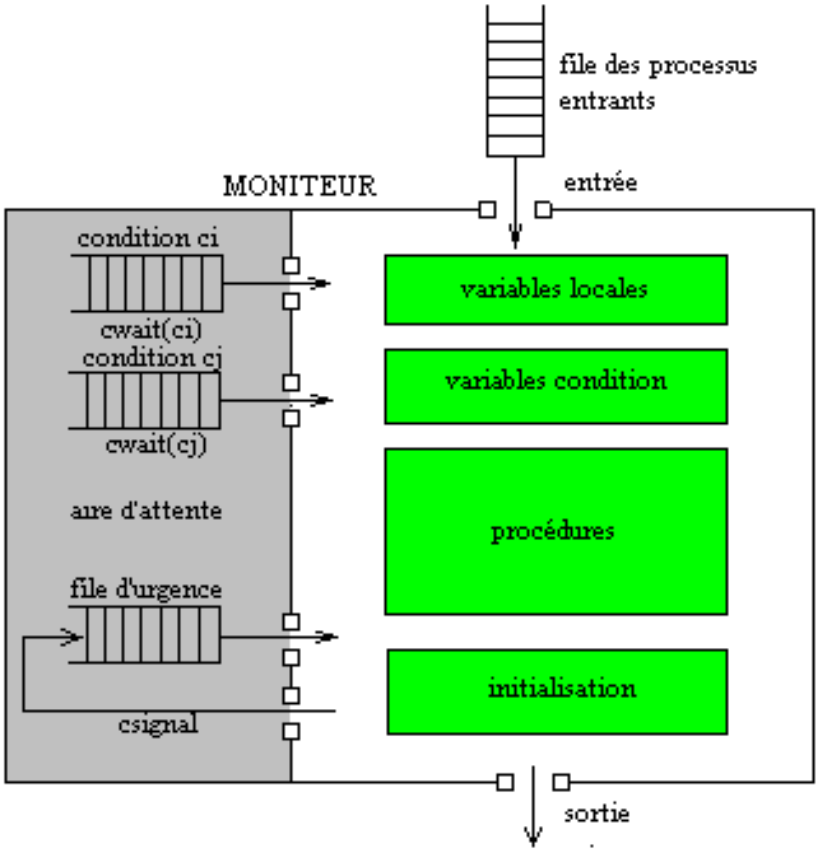
Moniteurs

Les sémaphores ou les compteurs d'événements semblent résoudre le problème de la communication entre processus. Mais c'est au programmeur de les définir et de les utiliser . Il ne doit pas faire d'erreur, sinon .....catastrophe !

Les moniteurs affranchissent le programmeur de ce problème. Ils sont une structure de langage de programmation : c'est le compilateur qui fait le travail . Ils ont été proposés par Hoare (1974) et Brinch Hansen (1973).

Un moniteur est un ensemble comprenant

- des procédures
- des structures de données
- des variables spéciales appelées variables conditions



Cet ensemble constitue un module spécial reconnu par le compilateur du langage. Notons que les langages courants (Pascal, C, ...) ne possèdent pas cette possibilité.

La propriété essentielle d'un moniteur est qu'un seul processus, à un instant donné, peut être actif dans un moniteur. Si un processus est actif dans un moniteur, un autre processus qui voudrait atteindre une procédure du moniteur devra attendre que le premier ait terminé. L'exclusion mutuelle à l'entrée dans le

moniteur est réalisée par le compilateur (avec des sémaphores par exemple), non par le programmeur qui ne s'en préoccupe pas.

Pour gérer les blocages de processus lorsque ceux-ci ne peuvent continuer, deux opérations sont utilisées : cwait(C) et csignal(C) où C est une variable condition (qui correspond à une file d'attente)

- cwait (C) : le processus est placé dans la file d'attente C : blocage
- csignal (C) : un processus est retiré de la file d'attente C : déblocage

Illustrons le principe des moniteurs avec le problème des producteurs et des consommateurs.

```
processus producteur
début
  Répéter indéfiniment
     produire ;
     ProdCons.deposer ;
  FinRépéter
fin
```

```
processus consommateur :
début
   Répéter indéfiniment
      ProdCons.retirer ;
      consommer ;
   FinRépéter
fin
```

```
monitor ProdCons
début
 condition : pleines, vides ;
 entier : compteur ;

 procédure deposer
 début
  Si compteur = N alors
      cwait(pleines) ;
      //buffer plein
      deposer_un_objet ;
      compteur = compteur+ 1 ;
      Si compteur = 1 alors
        signal(vides) ;
        //réveil
      FinSi
  FinSi
 fin


 procédure retirer
 début
   Si compteur = 0 alors
     cwait(vides) ;
     //buffer vide
     retirer_un_objet ;
     compteur = compteur - 1 ;
     Si compteur = N-1 alors
        signal(pleines);
        //réveil
     FinSi
   FinSi
 fin


 compteur = 0 ;
 //[initialisation]
fin
```

Au départ , la variable compteur est initialisée à 0. Supposons que le processus consommateur veuille consommer ; il entre dans le moniteur avec ProdCons.retirer (possible car aucun processus n'y est présent). Comme compteur est à 0, cwait (vides) le bloque et il est mis dans la file d'attente "vides" ; il a donc quitté le moniteur. Si maintenant le processus producteur intervient, il entre dans le moniteur avec ProdCons.deposer (possible car le processus consommateur n'y est plus) ; il dépose un objet et compteur

passe à 1 ; signal (vides) débloque alors le processus qui était dans la file d'attente "vides" ; toutefois celui-ci ne peut entrer aussitôt dans le moniteur car le processus producteur y est toujours ; quand celui-ci sort, le processus consommateur fraîchement débloqué entre dans le moniteur et consomme ; compteur revient à 0.

Messages

Les solutions proposées jusqu'ici au problème de la communication de processus : sémaphores, compteurs d'événement, moniteurs supposent que les processus communiquent par l'intermédiaire d'une mémoire partagée. Ce n'est plus le cas dans un système réparti composé de plusieurs machines en réseau. Il faut alors utiliser la méthode de l'échange de messages

Deux primitives sont utilisées :

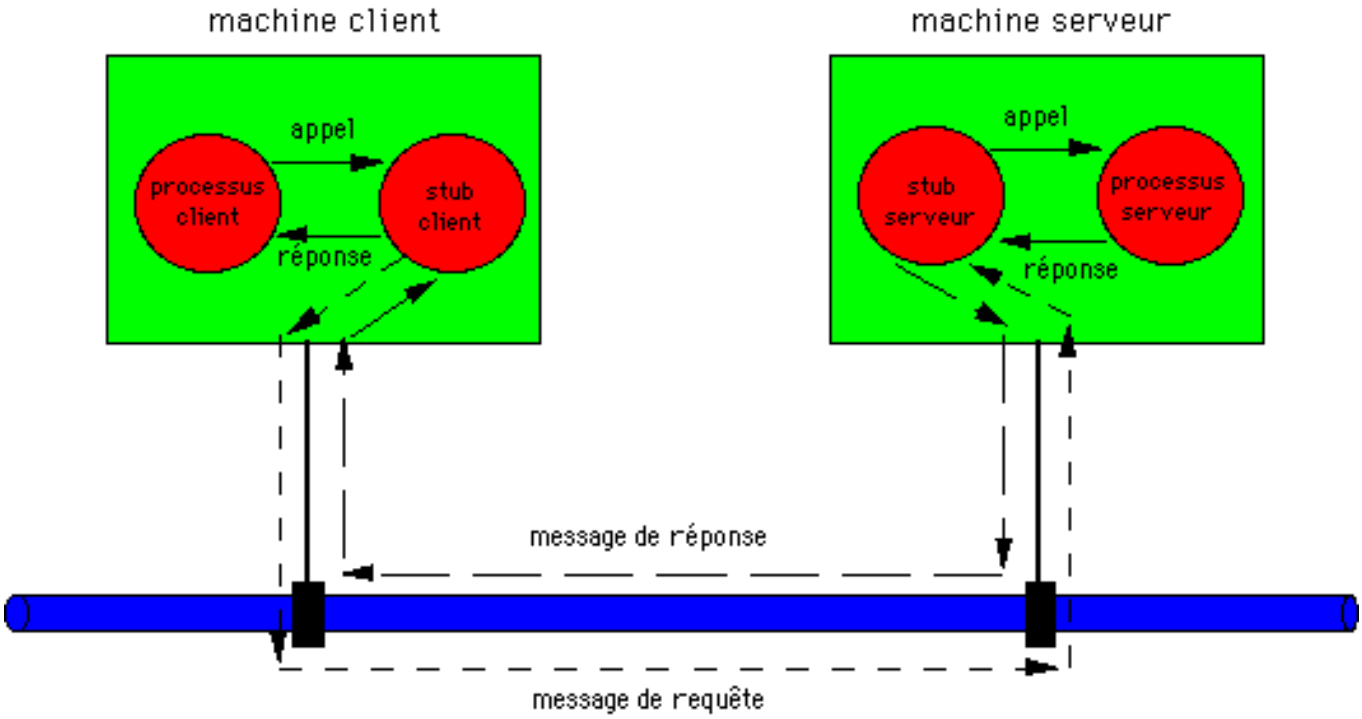
- send (destination, message) envoi d'un message vers un processus destination
- receive (source, message) réception d'un message provenant d'un processus source.

Dans le cas de receive, il y a blocage si aucun message n'est reçu.

Quatre propriétés caractérisent les messages :

- acquittement ; sur un réseau, les messages peuvent se perdre. Pour être sûr qu'un message arrive à destination, le récepteur envoie un message d'acquittement. Quand un processus a envoyé un message, il attend pendant un temps défini le retour de l'acquittement. S'il ne le reçoit pas, il ré-émet le message.
- numérotation ; un message porte un numéro qui l'identifie sans ambiguïté. Ainsi, dans le cas où un message est bien reçu par un processus, mais où l'acquittement s'est perdu, le processus va recevoir à nouveau le même message. Comme ces deux messages portent le même numéro, on peut ignorer le second.
- adressage ; les processus source ou destination doivent être nommés sans ambiguïté. Dans un système réparti simple, un processus sera désigné par nom_processus@nom_machine ; dans le cas où le système est plus important et géré par plusieurs administrateurs, les machines sont réparties en domaines et un processus aura la désignation nom_processus@nom_machine.nom_domaine. Ainsi deux machines appartenant à deux domaines différents peuvent avoir le même nom.
- authentification ; pour assurer le confidentialité des messages, on est conduit dans certains cas à coder le message en utilisant une clé connue seulement des utilisateurs autorisés.

Le échanges de message sont utilisés dans la méthode de l' "appel de procédure à distance". Soit deux machines quelconques reliées par réseau et supposons qu'un processus "client" sur une des machines désire, par exemple lire un fichier par l'intermédiaire d'un processus "serveur" situé sur l'autre machine.



La technique de l'appel de procédure à distance fait intervenir des processus spéciaux appelés "stubs" (en français talon ou souche) pour chacune des machines :

- le processus client appelle localement le stub client
- le stub client envoie un message au stub serveur
- le stub serveur appelle le processus serveur
- le processus serveur fournit des données au stub serveur
- le stub serveur envoie ces données au stub client par l'intermédiaire d'un message
- le stub client fournit les données au processus client

L'intérêt de la méthode est que les envois de messages sont masqués aux utilisateurs qui ne perçoivent que des appels locaux de procédure. Toutefois, il y a des difficultés à surmonter :

- passage de paramètres d'une machine à l'autre : représentations internes des données différentes
- panne du serveur : que reçoit le client ?

3 possibilités en cas de panne du serveur :

- "at least once" : l'appel sera retransmis au moins une fois
- "at most once" : aucun appel ne sera transmis plus d'une fois
- "maybe" : rien n'est garanti

Appliquons l'échange de messages au problème des producteurs et des consommateurs. On considère que l'on a N messages de tailles identiques gérés par le système d'exploitation (utilisation d'une mémoire tampon).

Le producteur produit et remplit un message vide. Le consommateur vide un message plein et consomme. Si le consommateur ne trouve pas de message plein, il attend. Si le producteur ne trouve pas de message vide, il attend.

```
processus producteur
début
  Répéter indéfiniment
      produire(item) ;
      receive(consommateur, m) ;
      remplir_message(m, item) ;
      send(consommateur, m) ;
  FinRépéter
fin
```

```
processus consommateur
début
   Pour i= 0 à N faire
      send(producteur, m) ;
   FinPour
   Répéter indéfiniment
      receive(producteur, m) ;
      vider_message(m, item) ;
      consommer(item) ;
      send(producteur, m) ;
   FinRépéter
fin
```

Equivalence des méthodes

En définitive, les méthodes précédentes résolvent le problème de la communication entre processus . Il y en a bien sûr d'autres : séquenceurs (Reed et Kanodia - 1979), path expressions (Campbell et Habermann - 1974), sérialiseurs (Atkinson et Hewitt - 1979).

On peut montrer que ces méthode sont équivalentes sémantiquement. En particulier, on peut montrer que

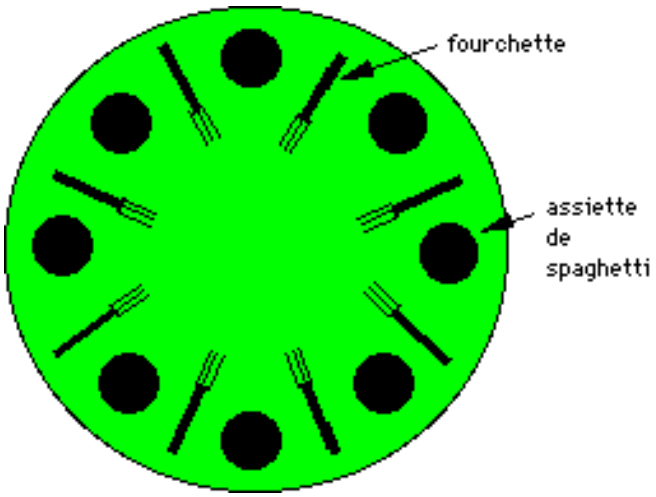
- à partir des sémaphores, on peut construire des moniteurs et pratiquer l'échange de messages
- à partir des moniteurs, on peut réaliser des sémaphores et pratiquer l'échange de messages
- à partir des messages, on peut réaliser des sémaphores et des moniteurs.

Nous renvoyons le lecteur aux ouvrages spécialisés pour les démonstrations correspondantes.

Problèmes classiques

Le repas des philosophes

Ultra-classique. N philosophes sont autour d'une table ronde; chacun d'eux a devant lui une assiette de spaghettis qu'il doit manger à l'aide de deux fourchettes; mais chaque assiette est séparée de la voisine par une seule fourchette. Il y a donc N fourchettes ce qui signifie que deux philosophes voisins ne peuvent manger simultanément.



Le problème des philosophes modélise une système n'ayant qu'un nombre limité de ressources. Chaque philosophe se conduit de la manière suivante : il essaie d'obtenir les deux fourchettes ; s'il réussit, il mange un certain temps, puis pose les deux fourchettes et se met à penser (Heureusement, les philosophes ne mangent pas tout le temps, ils pensent aussi !). Après quoi il tentera à nouveau de manger. En assimilant chaque philosophe à un processus, tentons de donner une solution satisfaisante à ce problème. On examinera plusieurs solutions plus ou moins bonnes.

solution 1 :

```
processus philosophe( i)
début
  Répéter indéfiniment
      penser();
      prendre_fourchette(i);
      //fourchette de gauche
      prendre_fourchette((i+1) modulo N);
      //fourchette de droite
      manger();
      poser_fourchette(i);
      poser_fourchette((i+1)modulo N);
  FinRépéter
fin
```

La fonction prendre_fourchette(n) bloque le processus s'il n'est pas possible de prendre la fourchette n. Cette solution ne marche pas : si simultanément tous les philosophes prennent la fourchette gauche, ils ne pourront plus prendre la fourchette droite et seront donc tous bloqués !

solution 2 : modification de solution 1 . Après avoir pris la fourchette de gauche, le philosophe vérifie si la fourchette de droite est libre ; si elle ne l'est pas, le philosophe pose la fourchette de gauche et reprendra tout le scénario un peu plus tard.

Cette solution ne marche pas non plus : il est possible que tous les philosophes fassent exactement la même chose simultanément : ils prennent la fourchette de gauche, vérifient qu'ils ne peuvent prendre la fourchette de droite, posent la fourchette de gauche, attendent un temps donné et recommencent indéfiniment la même séquence d'actions (inutiles). On appelle ce cas une "famine" (starvation) et pour cause

!.

solution 3 : comme la solution 2, mais avec un délai aléatoire (différent pour chaque philosophe) entre la prise des deux fourchettes.

Cette solution marche avec une grande probabilité, mais le risque précédent est toujours présent car on ne peut se fier au hasard.

solution 4 : emploi d'un sémaphore binaire mutex initialisé à 1

```
processus philosophe( i)
début
  Répéter indéfiniment{
     penser();
     P(mutex);
     prendre_fourchette(i);
     //fourchette de gauche
     prendre_fourchette((i+1) modulo N);
     //fourchette de droite]
     manger();
     poser_fourchette(i);
     poser_fourchette((i+1)modulo N);
     V(mutex);
  FinRépéter
fin
```

Cette solution est correcte mais peu pratique puisqu'à un moment donné, un seul philosophe peut manger.

solution 5 (la bonne) : On considère que le philosophe est dans un des trois états suivants : penser, manger, vouloir manger (c'est à dire attendant les fourchettes). On affecte un sémaphore à chaque philosophe : s[i].

```
sémaphore mutex = 1;
sémaphore s[N] ; // un sémaphore par philosophe
PENSE = 0 ; FAIM = 1 ; MANGE = 2 ;

processus philosophe( i)
début
  Répéter indéfiniment
     penser();
     prendre_fourchettes(i);
     manger();
     poser_fourchettes(i);
     prendre_fourchettes (i)
     P(mutex);
     état[i] = FAIM ;
     test(DROITE) ;
     V(mutex);
  FinRépéter
fin

procédure test(i)
début
```

```
    Si état[i]=FAIM et état[GAUCHE] différent de MANGE
    et état[DROITE] différent de MANGE alors
       état[i] = MANGE ;
       V(s[i]) ;
    FinSi
  fin
```

Lecteurs et Ecrivains

Ce problème modélise les accès à une base de données. Les écrivains écrivent des données que les lecteurs lisent. Le problème à régler est celui où simultanément plusieurs processus tentent de lire ou d'écrire. Plusieurs processus peuvent lire simultanément, mais si un processus est en train d'écrire, aucun autre processus ne doit être autorisé à atteindre les données (en lecture ou en écriture).

Une solution est donnée ci-dessous ; elle utilise deux sémaphores mutex et bd et un compteur rc de lecteurs . Cette solution favorise les lecteurs par rapport aux écrivains.

```
mutex =1 ;
  bd = 1 ;
  rc = 0 ;
```

```
processus lecteur
début
  Répéter indéfiniment
    P(mutex) ;
    rc = rc +1;
    Si rc = 1 alors
       P(bd) ;
       V(mutex) ;
       lire_données() ;
       P(mutex) ;
       rc = rc -1;
       Si rc = 0 alors
         V(bd) ;
         V(mutex) ;
         utiliser_données() ;
       FinSi
FinSi
FinRépéter
fin
```

```
processus écrivain :
début
  Répéter indéfiniment
      fabriquer_données() ;
      P(bd) ;
      ecrire_données() ;
      V(bd) ;
  FinRépéter
fin
```

Le barbier endormi

Une boutique de barbier contient un fauteuil, n chaises d'attente et .... un barbier. Si aucun client n'est présent, le barbier s'assied dans le fauteuil et s'endort. Quand un client arrive, il réveille le barbier. Si d'autres clients arrivent pendant que le barbier travaille, soit ils s'asseyent sur les chaises d'attente, soit ils quittent la boutique si il n'y a pas assez de chaises d'attente.

La solution qui décrit le fonctionnement correct du système utilise 3 sémaphores : clients (qui, en fait, compte le nombre de clients qui attendent), barbiers (qui compte le nombre de barbiers qui attendent le client : 0 ou 1), et mutex pour l'exclusion mutuelle. La solution utilise aussi une variable poireau qui compte le nombre de clients qui attendent (copie de clients); la raison de cette variable est qu'il n'est pas possible de lire la valeur d'un sémaphore.

```
clients = 0 ;
barbiers = 0 ;
mutex = 1 ;
poireau = 0 ;
```

```
processus barbier
début
   Répéter indéfiniment
      P(clients) ;
      //si pas de clients dormir
      P(mutex) ;
      poireau = poireau - 1 ;
      V(barbiers) ;
      V(mutex) ;
      raser() ;
   FinRépéter
fin
```

```
processus client
début
  P(mutex);
  Si poireau < CHAISES alors
      //rester
      poireau = poireau + 1 ;
      V(clients) ;
      V(mutex) ;
      P(barbiers) ;
      se_faire_raser() ;
  sinon V(mutex);
      //partir
  FinSi
fin
```

Synchronisation des processus

Exercices

Exercice 1

Le programme synchro permet de simuler les différents stratagèmes explicités dans le cours. Téléchargez-le et essayez ses différentes possibilités.

NB : le simulateur comprend, outre le programme de simulation, un fichier Valeur.txt qu'il faut placer dans le même répertoire que synchro.exe.

Exercice 2

Considérons les processus concurrents suivants (avec i=0 ou 1) qui utilisent un vecteur de booléens : blocked[i] et un entier turn. A l'initialisation, on a

blocked[0]=FAUX ; blocked[1]=FAUX ; turn = 0.

```
processus P(i)
début
  Répéter indéfiniment
     blocked[i]=VRAI ;
     TantQue turn <> i faire
          TantQue blocked[1-i] faire ;
          FinTantQue
          turn = i ;
     FinTantQue
     section critique ;
     blocked[i]=FAUX ;
     section non critique ;
  FinRépéter
fin
```

Montrer que cette solution au problème de l'exclusion mutuelle est incorrecte.

Exercice 3

On considère la définition suivante des sémaphores :

```
P(S)
début
  Si S > 0 alors
    S = S - 1 ;
  sinon
     bloquer le processus ;
  FinSi
fin
```

```
V(S)
début
  Si il existe un processus dans la file d'attente alors
     réveiller le processus ;
  sinon
     S = S + 1 ;
  FinSi
fin
```

Comparer cette définition à celle donnée dans le cours.

Exercice 4

On peut définir les sémaphores à partir des sémaphores binaires. Le procédé est explicité ci-dessus, mais avec une erreur.

```
P(S)
début
  PB(mutex) ;
  S = S - 1 ;
  Si S < 0 alors
    VB(mutex) ;
    PB(delai) ;
  sinon
    VB(mutex) ;
  FinSi
fin
```

```
V(S)
début
  PB(mutex) ;
  S = S + 1 ;
  Si S <= 0 alors
    VB(delai) ;
  FinSi
  VB(mutex) ;
fin
```

où PB et VB sont les primitives qui agissent sur les sémaphores binaires mutex (initialisé à 1) et delai (initialisé à 0).

Corriger l'erreur et montrer que cette définition est acceptable.

Exercice 5

Un restaurant du type "restauration rapide" emploie 4 types d'employés :

- les preneurs d'ordre qui reçoivent les désirs des clients
- les cuisiniers qui préparent les plats demandés
- les conditionneurs qui emballent (avec force carton) la nourriture
- les caissiers qui donnent aux clients leur commande et encaissent le montant.

En considérant chaque employé comme un processus, étudier la synchronisation de tous ces processus.

Synchronisation des processus

Solution des exercices

Solution de l'exercice 2

Supposons que le processus P(0) se "plante" dans sa section critique ; blocked[0] reste à VRAI. Dans le processus P(1), la boucle "TantQue blocked[0] faire ;" fait alors attendre indéfiniment ce processus.

Solution de l'exercice 4

Correction de l'erreur :

```
P(S)                    V(S)
début                   début
  PB(mutex) ;             PB(mutex) ;
  S=S-1 ;                 S = S+1 ;
  Si S<0 alors            Si S<=0 alors
     PB(delai) ;             VB(delai) ;
  FinSi                   FinSi
  VB(mutex) ;             VB(mutex) ;
fin                     fin
```

Les primitives PB(mutex) et VB(mutex) qui encadrent le corps des processus P(S) et V(S) assurent une exclusion mutuelle de ces corps ce qui est nécessaire car les primitives ne peuvent être interrompues.

Solution de l'exercice 5

On peut identifier les processus Client, Preneur, Cuisinier, Conditionneur et Caissier qui doivent s'exécuter en synchronisation. Par exemple, un cuisinier ne reçoit de commande que si un preneur d'ordre lui en donne une. En définissant 10 sémaphores de synchronisation S1 à S10 initialisés à 1, on a la situation suivante :

Client	Preneur	Cuisinier	Conditionneur	Caissier
début	début	début	début	début
P(S1)	P(S2)	P(S4)	P(S5)	P(S8)
demande	prend	reçoit	conditionne	reçoit
V(S2)	V(S1)	V(S3)	V(S6)	V(S7)
P(S3)	P(S3)	P(S6)	P(S7)	P(S10)
paie	transmet	prépare	passe	donne et encaisse
V(S10)	V(S4)	V(S5)	V(S8)	V(S9)
fin	fin	fin	fin	fin