LES PROCESSUS

1. NOTION DE PROCESSUS 2

2. MODELES DE REPRESENTATION D'UN SYSTEME DE TACHES 2

2.1. GRAPHES DE PRECEDENCE 2

2.1.1. Spécification dans les langages évolués 3

2.1.2. Langage d'un système de tâches 4

2.2. AUTOMATES FINIS 5

2.3. RESEAUX DE PETRI 6

Exemple d'évolution du Réseau de Pétri: 7

3. SYNCHRONISATION DES PROCESSUS 8

3.1. TEST & SET 8

3.2. PRIMITIVES P ET V ET SEMAPHORES 9

3.3. EXCLUSION MUTUELLE ET SEMAPHORES 12

3.4. MONITEURS ET CONDITIONS 13

4. COMMUNICATION ENTRE PROCESSUS 15

5. GESTION DES PROCESSUS SOUS UNIX 16

5.1. AU NIVEAU DE L'INTERPRETEUR DE COMMANDES 16

5.2. AU NIVEAU DU LANGAGE C 16

5.3. SEMAPHORES ET APPELS SYSTEME EN LANGAGE C 18

1. Notion de processus

Dans un environnement multitâche, un programme est constitué de plusieurs flux d'exécution, appelés processus, pouvant être exécutés en parallèle.

Par définition, on appelle processus est un ensemble d'instructions ou d'actions qui s'exécutent séquentiellement. Un processus est encore appelé tâche séquentielle.

Un processus est caractérisé par un certain nombre d'informations: son nom, son état (actif, activable, bloqué, …), son contexte (compteur ordinal, registres de données, registres d'index, registres de protection mémoire, …) permettant la reprise après interruption du déroulement. Le descripteur d'un processus est appelé vecteur d'état ou mot d'état, il s'agit d'une zone mémoire dans une table système contenant les caractéristiques du processus.

Dans un système d'exploitation **mono tâche**, les différents processus, qu'ils soient Batch ou interactifs, s'exécutent en parallèle et appartiennent à des utilisateurs différents. Pour une utilisation donnée (job Batch ou session interactive) un seul processus s'exécute à la fois. Avec MS-DOS (système **mono utilisateur**), le lancement d'un programme à partir d'un autre programme est pratiquement équivalent à un appel de sous-programme (avec chargement en mémoire centrale de celui-ci). Le programme appelant attend la fin d'exécution du programme appelé avant de continuer.

Dans un système d'exploitation **multitâche**, plusieurs processus peuvent être rattachés au même travail (job Batch ou session interactive). A partir d'un processus initial sont lancés des processus secondaires qui peuvent s'exécuter en parallèle avec le processus initial. Sous Unix, le lancement d'un processus fils à partir d'un processus père peut s'effectuer avec ou sans attente (blocage) du père.

Par la suite, nous utiliserons indifféremment le mot tâche ou le mot processus pour désigner une tâche séquentielle.

2. Modèles de représentation d'un système de tâches

Il existe plusieurs représentations graphiques permettant la description et l'analyse d'un système de tâches: le graphe de précédence, les automates finis et les réseaux de Pétri.

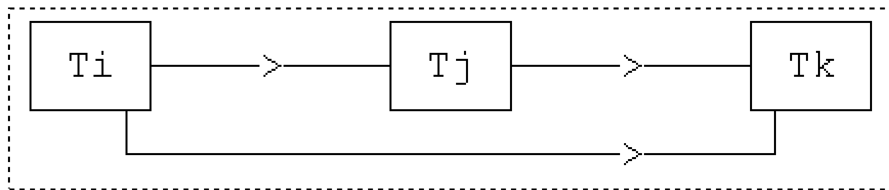
2.1. GRAPHES DE PRECEDENCE

Les graphes de précédence permettent de représenter les relations de précédence d'un système de tâches.

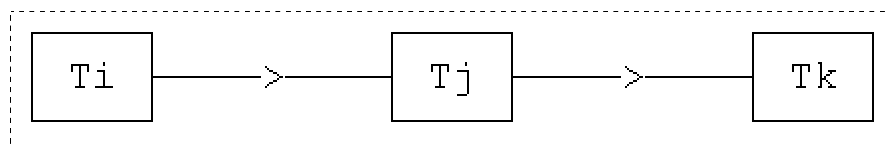
Une **relation de précédence** sur un ensemble E est une relation vérifiant les propriétés suivantes:

1) pour tout T de l'ensemble E, $T < T$ est impossible
2) pour tout (T_1, T_2) de E x E, $T_1 < T_2$ et $T_2 < T_1$ est impossible simultanément
3) la relation $<$ est transitive

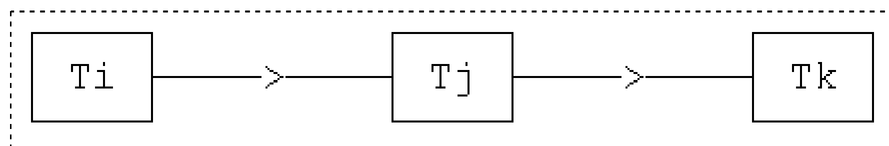
Un **système de tâches** (E, $<$) est un ensemble E de tâches muni d'une relation de précédence $<$. Un système de tâches peut se représenter sous forme de graphe orienté. Les sommets sont les tâches et les arcs représentent les relations de précédence entre les tâches. Un arc de la tâche T_i vers la tâche T_j existe si et seulement si $T_i < T_j$.



La relation de précédence étant transitive, le graphe obtenu est redondant. L'élimination de la redondance permet d'obtenir le graphe minimal appelé **graphe de précédence**:

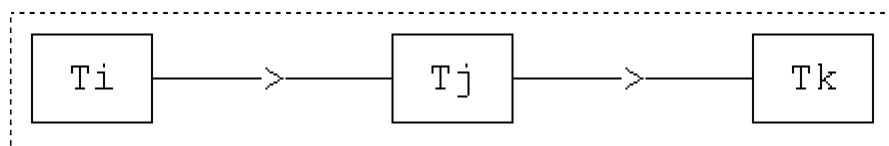


Dans un graphe de précédence, une tâche T_k est appelée **successeur immédiat** de la tâche T_j, s'il existe un arc reliant T_j à T_k. Plus généralement une tâche T_k est un **successeur** de T_i si T_k est un successeur immédiat de T_i ou si T_k est une successeur d'un successeur immédiat de T_i. Une tâche sans successeur est dite **terminale**.



T_k successeur immédiat de T_j et successeur de T_i

Une tâche T_i est appelée **prédécesseur immédiat** de la tâche T_j, s'il existe un arc reliant T_i à T_j. Plus généralement une tâche T_i est un **prédécesseur** de T_k si T_i est un prédécesseur immédiat de T_k ou si T_i est une prédécesseur d'un prédécesseur immédiat de T_k.



T_i prédécesseur immédiat de T_j et prédécesseur de T_k

Si T_x n'est ni successeur ni prédécesseur d'une tâche T_y, alors les tâches T_x et T_y sont dites **indépendantes**. Deux tâches T_x et T_y d'un système de tâches peuvent s'exécuter en parallèle s'il n'y a aucune relation de précédence entre elles.

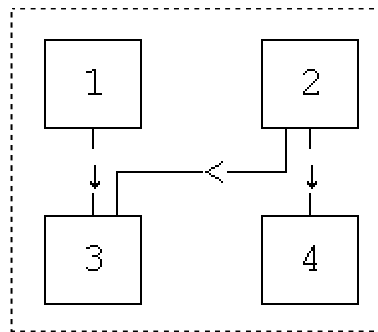
2.1.1. SPECIFICATION DANS LES LANGAGES EVOLUES

Il s'agit de représenter à travers un langage la mise en séquence ou en parallèle des tâches d'un graphe de précédence.

La mise en séquence se note: **SeqBegin** T1; T2; T3 **SeqEnd**
ou encore **{** T1; T2; T3 **}**

La mise en parallèle se note: **ParBegin** T4; T5; T6 **ParEnd**

Exercice: Représenter le graphe de précédence ci-dessous par un programme parallèle utilisant des instructions parbegin/parend et seqbegin/seqend ?



Exercice: Dessiner le graphe de précédence correspondant au calcul de l'expression arithmétique: 2*cos(a) + 4*sin(b) effectué par le programme:

```
{
ParBegin lire(a); lire(b) ParEnd;
ParBegin c:=2*cos(a); d:=4*sin(b) ParEnd;
r:=c+d;
}
```

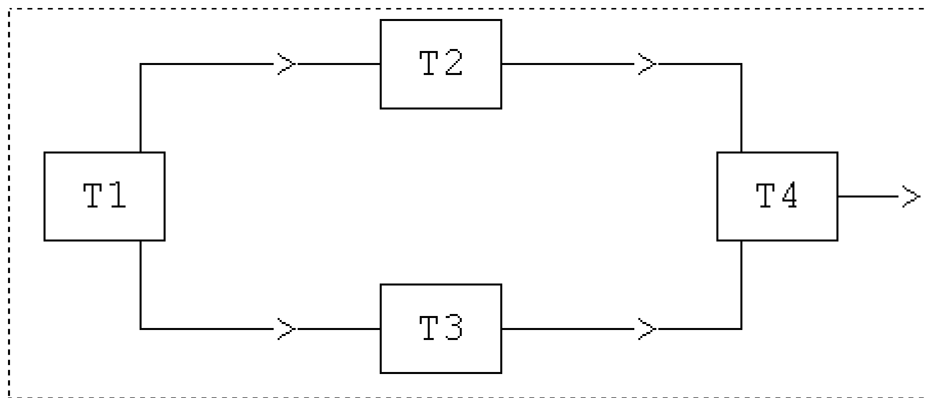
Exercice: Ecrire un programme parallèle faisant intervenir 3 processus pour la multiplication de deux matrices 3x3, la tâche élémentaire étant la multiplication d'une ligne par une colonne.

2.1.2. LANGAGE D'UN SYSTEME DE TACHES

Le langage d'un système de tâches permet d'exprimer à l'aide de mots toutes les réalisations possibles d'exécution d'un système de tâches. Il est défini grâce à un alphabet A={ d1, f1, d2, f2, ..., d_n, f_n } composé d'un ensemble de lettres représentant le début et la fin d'exécution de chaque tâche tel que:

1) chaque lettre apparaît une et une seule fois
2) pour chaque tâche T_k la lettre d_k apparaît avant la lettre f_k
3) si $T_i < T_j$ alors la lettre f_i est avant la lettre d_j

Chaque mot du langage, représente un comportement possible du système. Tous les mots possèdent la même longueur égale au double du nombre de tâches.



Pour le système ci-dessus, l'ensemble de tous les comportements possibles appelé **langage** est:

L(S) = { $d_1f_1d_2f_2d_3f_3d_4f_4$, $d_1f_1d_3f_3d_2f_2d_4f_4$,
$d_1f_1d_2d_3f_2f_3d_4f_4$, $d_1f_1d_3d_2f_3f_2d_4f_4$,
$d_1f_1d_2d_3f_3f_2d_4f_4$, $d_1f_1d_3d_22f_2f_3d_4f_4$ }

Exercice: Deux processus séquentiels...

2.2. AUTOMATES FINIS

Un automate fini est un quadruplet (A, E, e_0, F_t) où:

- A est un ensemble fini d'actions
- E est un ensemble fini d'états
- e_0 est un élément particulier de E appelé état initial
- F_t est une application de E x A dans E appelée fonction de transition.

La fonction de transition permet pour une action donnée de faire passer l'automate vers un nouvel état. Un graphe d'état peut être associé à un automate fini. Les sommets du graphe sont les états et les arcs représentent les différentes actions (tâches) possibles.

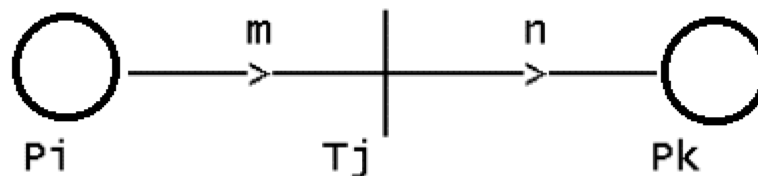
Exercice: Définir les états possibles d'un job en cours d'exécution ainsi que les actions du système d'exploitation.

2.3. RESEAUX DE PETRI

Un réseau de Pétri est un triplet (P, T, F_v) où:

- P est un ensemble fini de places
- T est un ensemble fini de transitions, disjoint de P
- F_v est une fonction de valuation de (P x T) union (T x P) vers l'ensemble des entiers naturels.

La représentation graphique d'un réseau de Pétri utilise des cercles (ou comme ici des carrés) pour les places et des barres pour les transitions. Les flèches relient les cercles aux barres et les barres aux cercles (mais pas les cercles entre eux ni les barres entre elles) et sont étiquetées par l'entier naturel résultat de la fonction de valuation. Un cercle placé avant une transition est appelé place d'entrée de cette transition. Un cercle situé après est appelé place de sortie.



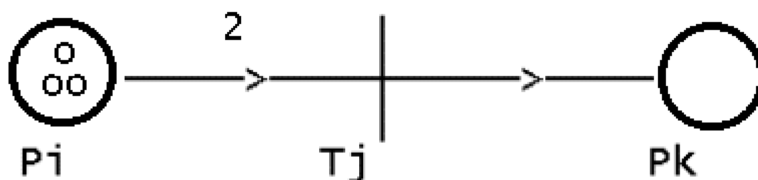
Place d'Entrée Place de Sortie

Avec $F_v(P_i, T_j)=m$ valuation d'entrée et $F_v(T_j, P_k)=n$ valuation de sortie.

Par convention, et par souci d'alléger les écritures, lorsqu'une valuation est égale à 1, elle n'a pas besoin d'être précisée.

Chaque place peut contenir un certain nombre de jetons. On appelle **marquage** une distribution particulière de jetons. Le marquage initial rend compte de la situation initiale.

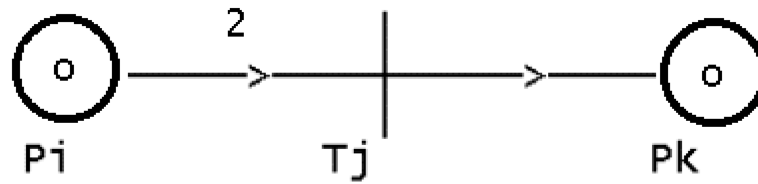
Pour un marquage donné, une transition T est dite **franchissable** si chacune de ses places d'entrée Pi contient un nombre de jetons supérieur ou égal à la valuation $F_v(P_i, T)$. Le franchissement de toutes les transitions franchissables s'effectue en même temps. Chaque place d'entrée Pi des transitions franchissables T_j diminue d'un nombre de jetons égal à la valuation d'entrée $F_v(P_i, T_j)$ et chaque place de sortie P_k augmente d'un nombre de jetons égal à la valuation de sortie $F_v(T_j, P_k)$. Exemple:



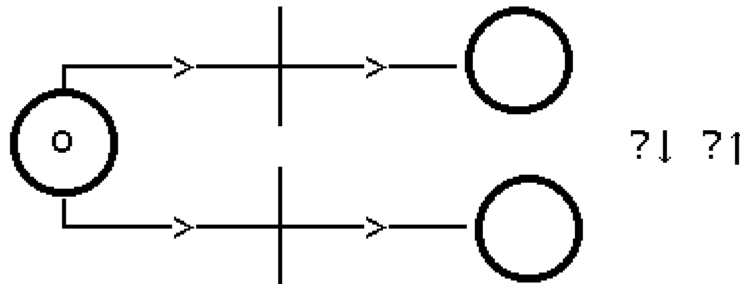
$F_v(P_i, T_j)=m=2$ et $F_v(T_j, P_k)=n=1$

La transition T_j est franchissable. Il y a 3 jetons dans la place d'entrée alors que la transition n'en demande que deux pour être franchie. Après déclenchement du

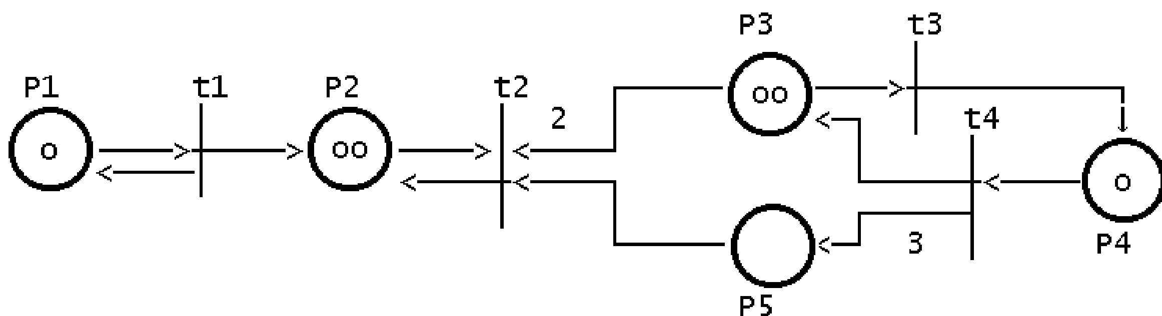
franchissement, il ne restera qu'un jeton dans la place d'entrée (3 au départ - 2 consommés) et il y aura 1 jeton produit dans la place de sortie.



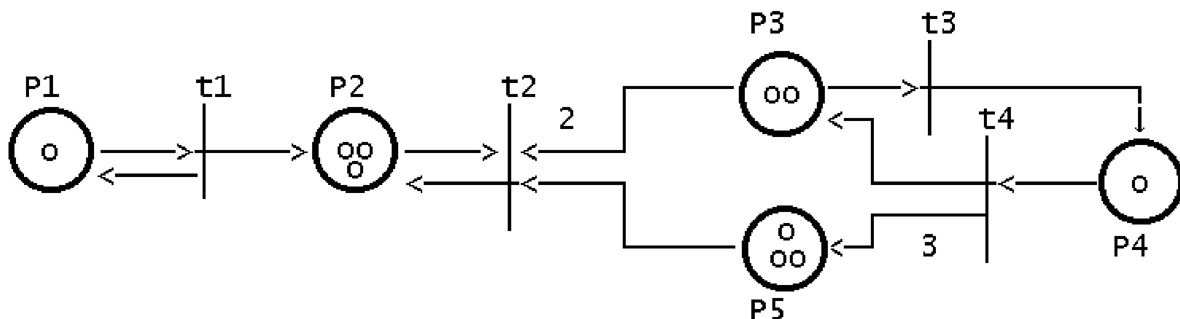
Lorsque qu'un place d'entrée est commune à plusieurs transitions, il se pose le problème du choix du franchissement à réaliser. Il y a un problème d'exclusion mutuelle sur la place d'entrée à résoudre.



EXEMPLE D'EVOLUTION DU RESEAU DE PETRI:

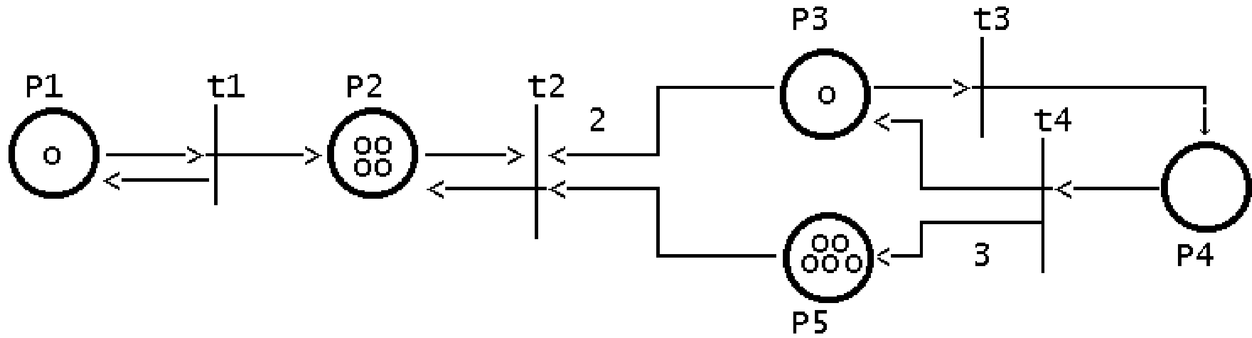


Les transitions franchissables sont t1, t3 et t4. Elles ne partagent pas de place d'entrée commune. Le déclenchement des franchissements donne:

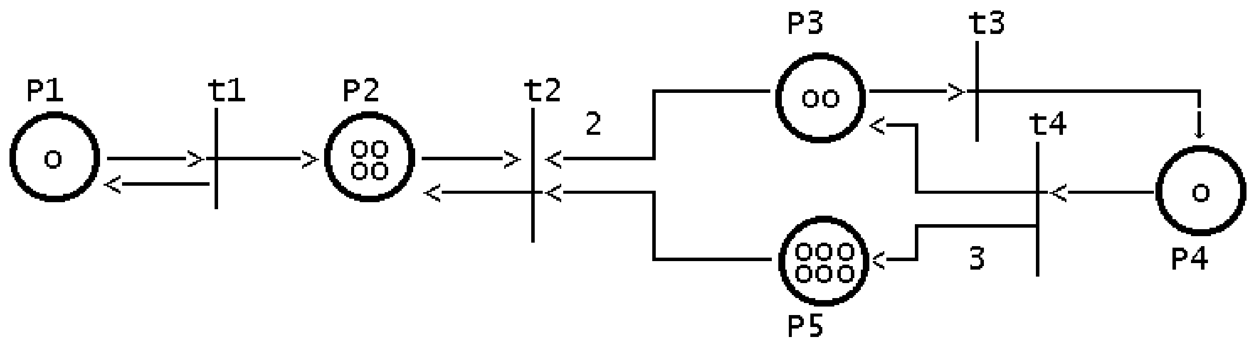


Les transitions franchissables sont t1, t2, t3 et t4. Les transitions t2 et t3 partagent la place d'entrée P3 commune. Il n'y a pas assez de jetons dans cette place pour déclencher à la fois t2 et t3. Il en résulte qu'il peut y avoir deux suites possibles à l'évolution du réseau de Pétri (son comportement n'est donc pas déterministe):

1) SI déclenchement de t1, t2 et t4:



2) SI Déclenchement de t1, t3 et t4:



Les actions (tâches) possibles sont les transitions à franchir. Elles dépendent des jetons (ressources) disponibles dans les places d'entrée. La synchronisation et la communication entre tâches s'effectuent par les jetons produits en sortie.

Exercice: Modéliser l'activité d'un barbier face à plusieurs clients.

Exercice: Reprendre l'exercice ci-dessus avec 3 barbiers.

3. Synchronisation des processus

Un des rôles majeurs d'un système d'exploitation est de veiller au bon enchaînement (ordonnancement) des processus, de gérer les accès concurrents aux différentes ressources.

Les mécanismes classiques de synchronisation sont le Test&Set (TAS), les sémaphores, les verrous d'exclusions mutuelles (*mutex*) avec les variables conditions.

3.1. Test & Set

La plupart des ordinateurs (monoprocesseur) possèdent une instruction matérielle (câblée) permettant de lire et d'écrire le contenu d'un mot de la mémoire. Cette instruction qui s'exécute de manière indivisible est appelée *Test-And-Set*. Son utilisation permet de résoudre les conflits d'accès à une zone critique (mémoire partagée ou section critique). Il est en effet possible de définir une variable de type Verrou (LOCK) qui indique quel processus, maître du verrou, peut être exécuté. Lorsque la variable est égale à 1, il y a un processus qui possède déjà le verrou.

Il n'y a qu'un seul processus qui puisse faire passer le verrou de 0 à 1. Les autres processus le font obligatoirement passer de 1 à 1. Le processus exécutant les instructions ci-dessous met à 1 la variable Nom_Verrou, il teste ensuite si la variable était à 0 juste avant, autrement dit si c'est bien lui qui l'a mise à 1. Il devient alors maître du verrou et doit le libérer quand il a fini.

```
Repeat
        Set_VERROU( Nom_Verrou, 1);
Until Was_Set_VERROU(Nom_Verrou, 0);
Section_Critique;
Set_VERROU( Nom_Verrou, 0);
```

Remarque: La séquence précédente mobilise le processeur dans ce qu'on appelle une attente active. Pour certains, il est alors intéressant de pouvoir tester l'état du verrou avant de rentrer dans la boucle active et de pouvoir suspendre son exécution un certain temps. Le processeur peut alors travailler pour une autre tâche.

3.2. PRIMITIVES P ET V ET SEMAPHORES

L'exclusion mutuelle ou la synchronisation des processus peut être réalisée par les primitives P (Proberen en néerlandais) et V (Verhogen) introduite par Dijkstra en 1965. Ces routines agissent sur des couples formés d'une variable entière S et d'une file d'attente (zone mémoire) appelés sémaphores.

Un sémaphore peut être associé à toute ressource partageable par un nombre fini de processus. En général, à l'initialisation, la variable entière du sémaphore prend pour valeur le nombre maximal de processus qui peuvent se partager la ressource. Chaque demande d'allocation valide de la ressource fera diminuer ce nombre. Lorsque ce dernier devient négatif, les demandes sont enregistrées dans la file d'attente et les processus demandeurs mis en attente.

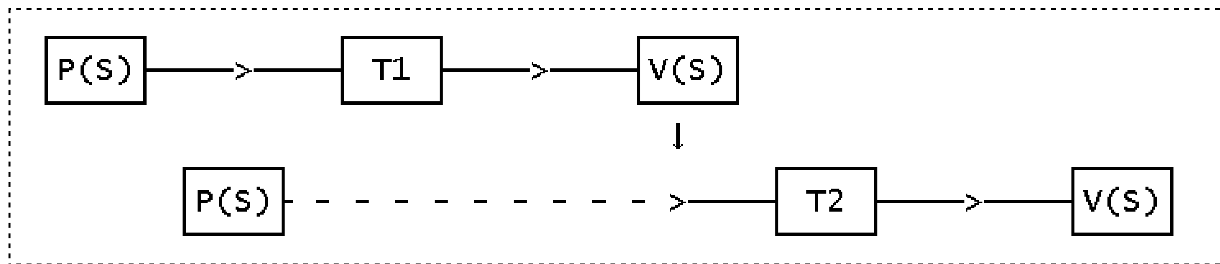
La primitive P(SEM) permet à un processus de demander l'allocation d'une ressource gardée par le sémaphore SEM:

```
SEM:=SEM-1
Si SEM < 0 Alors
        état du processus demandeur = bloqué
        mise en attente dans la file du sémaphore
Fin_Si
```

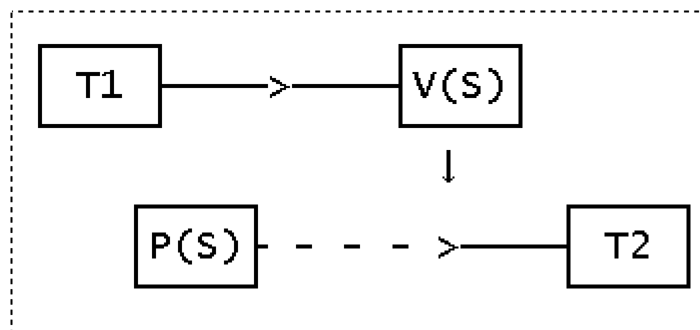
La primitive V(SEM) permet de libérer une ressource en cours d'utilisation et gardée par le sémaphore SEM:

```
SEM:=SEM+1;
Si SEM <=0 Alors il y a des processus à débloquer
        état du premier processus en attente = ACTIVABLE
Fin_Si
```

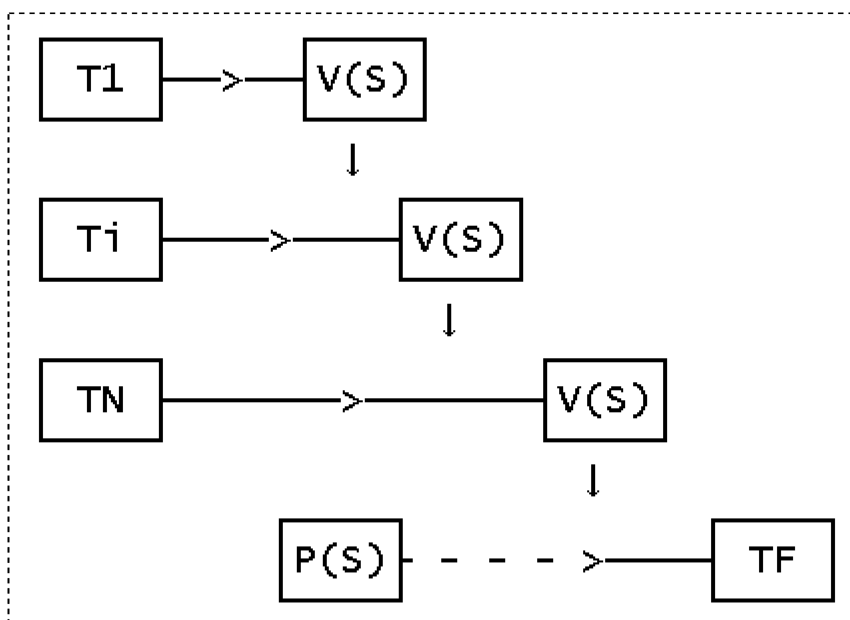
Lorsqu'un sémaphore est utilisé pour garantir l'exclusion mutuelle de processus demandeur (accès unique à une ressource), alors le processus ayant alloué la ressource doit obligatoirement la libérer ensuite. Cette libération permet au premier processus en attente d'être réactivé.



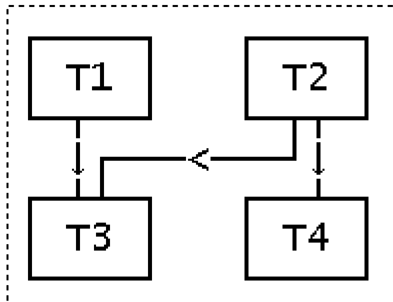
L'utilisation d'un sémaphore initialisé à 0 peut permettre la synchronisation de 2 processus. Le premier processus possède implicitement le droit d'utiliser. Il effectue sa tâche. Le second est en attente du sémaphore qui sera "libéré" par le premier à la fin de son exécution.



Plus généralement l'utilisation d'un sémaphore initialisé à une valeur négative égale à (N-1) peut permettre l'attente de la complétion de N processus. Chaque processus en fin de tâche "libère" (ajoute un droit) au sémaphore. Lorsque la valeur du sémaphore devient positive (à la Nième libération), le processus en attente du sémaphore est réactivé.



Le système de tâches, exprimé ci-dessous par un graphe de précédence peut être programmé à l'aide des structures ParBegin / ParEnd et SeqBegin / SeqEnd en utilisant 3 sémaphores. Chaque tâche signale à ses successeurs qu'ils peuvent démarrer. Il y a ici autant de sémaphores qu'il y a d'arc de précédence. La tâche T3 est en attente de la complétion T1 et T2. La tâche T4 est en attente de la complétion de T2.

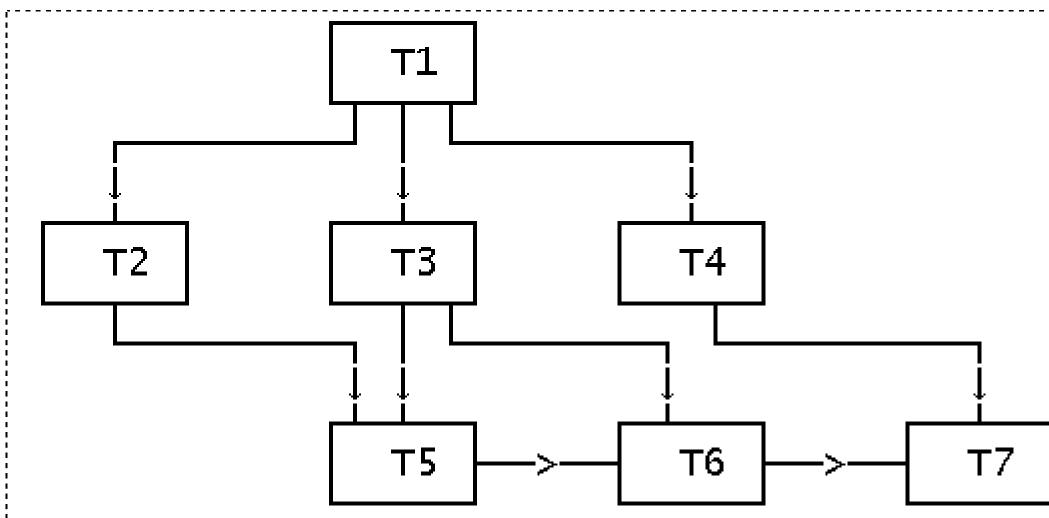



```
Init(SA, 0); Init(SB, 0); Init(SC, 0);
  ParBegin
  SeqBegin T1; V(SA)  SeqEnd;
  SeqBegin T2; V(SB); V(SC)  SeqEnd;
  SeqBegin    P(SA); P(SB); T3  SeqEnd;
  SeqBegin          P(SC); T4  SeqEnd
ParEnd
```

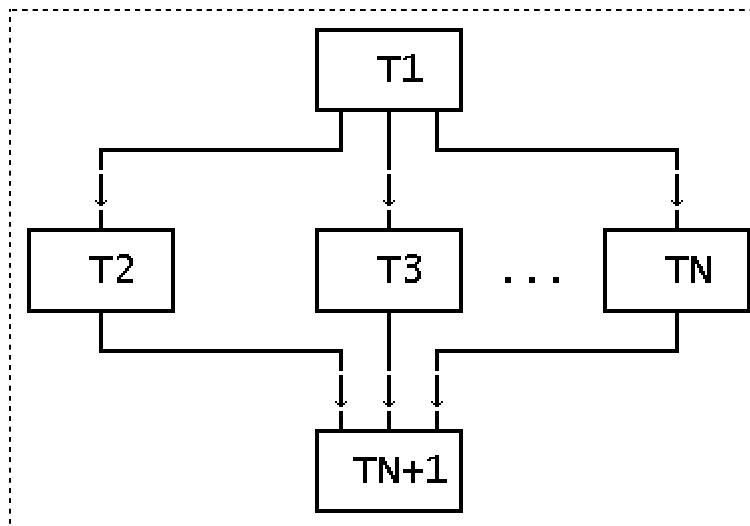
Il est possible de n'utiliser qu'un seul sémaphore. Pour cela on applique la devise : « diviser pour régner » : dans un premier temps, on supprime une relation de précédence (par exemple celle entre T2 et T3) et on programme le graphe obtenu sans sémaphore ; puis on rajoute le sémaphore pour tenir compte de la synchronisation entre **T2** et **T3**..

```
Init(S, 0);
ParBegin
  SeqBegin T2; V(S); T4 ;  SeqEnd;
  SeqBegin T1; P(S);  T3;  SeqEnd;
ParEnd
```

Exercice: Analyser le graphe ci-dessous. Utiliser les structures ParBegin/ParEnd et SeqBegin/SeqEnd pour exprimer la réalisation de ce système de tâches en utilisant un nombre minimal de sémaphores.



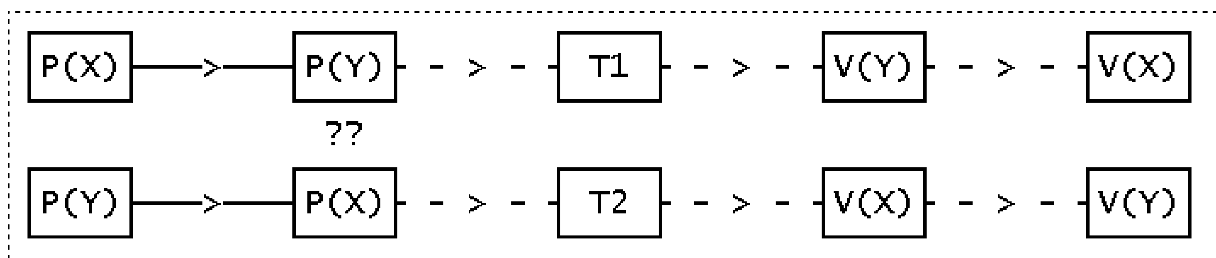
Exercice: Exprimer le graphe de précédence suivant à l'aide la structure ParBegin/ParEnd, en utilisant N+1 chaînes de tâches s'exécutant en parallèle et avec seulement 2 sémaphores.



Les sémaphores sont utilisables à travers les langages de programmation. Leur définition peut varier dans langage à un autre, il n'en reste pas moins que la philosophie des sémaphores reste la même globalement. Il est à noter qu'avec certains langages et certaines bibliothèques de fonctions associées, les sémaphores ne peuvent pas gérés plus d'un droit d'utilisation (initialisation de sémaphore à une valeur supérieure à 1).

3.3. EXCLUSION MUTUELLE ET SEMAPHORES

Tous les problèmes de conflit d'accès ne sont pas résolus par l'utilisation de sémaphores. Soient deux sémaphores X et Y deux initialisés à 1, l'exemple suivant conduit à un blocage:



Lorsque deux programmes doivent faire l'acquisition de deux sémaphores pour posséder le droit d'exécuter une action exclusive (utilisation d'une ressource non partageable ou exécution d'une section critique), la manière de programmer l'un ou l'autre programme peut conduire à une situation de blocage. Le problème à résoudre est d'interdire l'accès simultané par plusieurs processus à une ressource non partageable quelle que soit la manière de programmer.

En 1965, Dijkstra a posé et résolu un problème de synchronisation connu sous le nom du problème des philosophes: Cinq philosophes se partagent une table ronde. Autour de la table se trouvent cinq chaises, chacune appartenant à un philosophe. Sur la table sont disposées cinq baguettes, cinq assiettes et un plat de riz qui est toujours plein. Chaque philosophe passe son temps à penser ou à manger. Lorsqu'un philosophe a faim, il s'assoit à table et essaie de prendre les 2 baguettes situées de part et d'autre de son bol.

Si l'un au moins des 2 philosophes voisins a déjà pris une baguette, l'opération est impossible. Après avoir mangé, le philosophe repose ses baguettes et se remet à penser.

Exercice: Ecrire la procédure Philosophe(i) en utilisant 5 sémaphores, un par baguette. On supposera que la baguette de gauche est toujours prise avant la droite et que le même ordre est respecté pour les reposer. Montrer alors qu'il existe un enchaînement d'actions menant à un blocage.

Exercice: Modifier la procédure en utilisant un sémaphore spécifique ne permettant qu'à 4 philosophes de s'asseoir à table et montrer qu'un blocage n'est plus possible.

Exercice: En remarquant que 2 philosophes seulement peuvent manger simultanément, analyser s'il est plus efficace en terme de parallélisme de permettre à 2, 3 ou 4 philosophes exactement de s'asseoir à table.

3.4. MONITEURS ET CONDITIONS

Les **moniteurs**, introduits par Brinch-Hansen, Dijkstra et Hoare, permettent de rassembler dans une même structure toutes les sections critiques associées à une ressource (ou à un ensemble de ressources). Cette structure contient d'une part des variables internes et les variables partageables (ou ressources) et d'autre part les instructions qui les manipulent: procédures internes ou fonctions assurant l'accès aux ressources (sections critiques). L'exclusion mutuelle des processeurs demandeurs est réglé par le fait qu'un seul processus à la fois peut utiliser le moniteur à un instant donné.

Grâce aux moniteurs, la gestion des sections critiques n'est plus à la charge du programmeur mais est directement prise en compte par le langage de programmation (Concurrent Pascal, Simone, Portal ou Modula).

L'utilisation de **conditions** à l'intérieur d'un moniteur permet d'augmenter les possibilités de synchronisation et d'assurer la succession d'allocation des ressources par les processus. Les conditions sont des entités gérées par le langage et le système d'exploitation. Elles permettent la gestion d'une file d'attente manipulée par 2 opérateurs: **wait** et **signal**:

> wait : bloque dans la file d'attente le processus exécutif et lève l'exclusion sur le moniteur.
>
> signal : relance le plus ancien processus bloqué sur la condition C et suspend le processus exécutif jusqu'à ce que le processus relancé s'arrête.

Une opération *signal* n'a aucun effet si la file d'attente de la condition est vide. Toute opération *wait* provoque une mise systématique en file d'attente.

Les deux seules opérations sur la condition étant les opérations *wait* et *signal*, il n'est pas possible de connaître l'état de la "condition". C'est pourquoi une condition (entité système) est toujours utilisée en conjonction avec une variable booléenne (variable utilisateur gérée en accès exclusif).

Exemple:

```
Single Ressource: Monitor
  busy    : Boolean;
  nonbusy : Condition

  Procedure Acquire
  Begin
     If  busy  then nonbusy.WAIT;
     busy:=true
  End;

  Procedure Release
  Begin
     busy:=false;
     nonbusy.SIGNAL
  End;

Begin  (* initialisation *)
   busy:=false;
End Single Ressource
```

Remarque: On rencontre aussi les notations wait(C) et signal(C) dans certains langages de programmation.

Dans l'exemple précédent, la condition *nonbusy* est utilisée avec la variable booléenne *busy* interne au moniteur. Cette dernière variable est une ressource non partageable, cela veut dire en fait qu'un seul processus peut lire (tester) ou mettre à jour cette variable.

Supposons qu'un processus a appelé la fonction *release*. Ce processus possède l'accès exclusif au moniteur lorsqu'il est en train d'exécuter les instructions relatives à cette fonction. Ces instructions sont ainsi exécutées de manière indivisible. Si un autre processus appelle la fonction *acquire*, il ne peut exécuter les instructions relatives à cette fonction tant que le premier processus "possédant" le moniteur ne le libère pas. Ce n'est que lorsque la fonction *signal* est exécutée que le moniteur est libéré et que le processus appelant la fonction *acquire* peut prendre le moniteur et exécuter cette fonction.

Lorsque plusieurs processus appellent la fonction *acquire*, un seul peut exécuter les instructions relatives à cette fonction. Si la ressource protégée par la variable booléenne *busy* est libre (*busy* égale à false), alors le premier processus entrant dans le moniteur n'appelle pas la fonction *wait*, il prend la ressource en mettant à jour la variable booléenne *busy* avec la valeur true, puis quitte le moniteur. Un des autres processus en attente prend alors le moniteur pour exécuter les instructions de la fonction *acquire*. La variable *busy* étant égale à true, la fonction *wait* est appelée, ce qui a pour effet de suspendre le processus exécutif en libérant à nouveau le moniteur. Un autre processus peut exécuter à

son tour les instructions de la fonction *acquire*. Si la ressource n'a été pas relâchée, alors le processus sera à son tour suspendu et mis en attente.

Si maintenant, le processus possédant la ressource gardée par la variable *busy* relâche cette ressource en appelant la fonction *release*, alors la variable *busy* est mise à jour avec la valeur false et la fonction *signal* est appelée. Ces deux instructions s'exécutent de manière indivisible car l'accès au moniteur est exclusif. Ceci garantit le fait qu'aucun autre processus puisse tester la variable avant que le signal soit envoyé.

Il est important de noter ici que l'opération *signal* ne débloque qu'un seul processus parmi ceux qui sont en attente sur la condition. En général c'est le plus vieux qui sera réactivé. Il est néanmoins possible d'ordonner la file d'attente en donnant une priorité lors de l'appel du wait.

Exemple:

avec C.wait(P) où P est la priorité donnée à la condition C

C.signal relancera le processus de plus grande priorité.

Cette structure statique des moniteurs est bien adaptée au cas des systèmes multiprocesseurs à mémoire partagée.

4. Communication entre processus

Lorsque deux processus sont indépendants, ils ne partagent des ressources communes. Chacun d'eux exécute une tâche différente.

Lorsque l'on désire augmenter la rapidité d'une application, il est intéressant de la décomposer en tâches indépendantes.

Techniques utilisées :

- Les Signaux
- Les Fichiers
- Les PIPES
- Les Boîtes aux lettres
- Les Sockets
- Mémoire partagée
- Communication par messages

5. Gestion des processus sous Unix

On ne considère ici que les processus classiques (et non les THREADS).

5.1. AU NIVEAU DE L'INTERPRETEUR DE COMMANDES

La commande ***ps*** permet de lister les processus existants.

La commande ***kill*** permet de terminer un processus.

Lorsqu'un utilisateur veut lancer une tâche en arrière plan (background) il ajoute le caractère ***&*** à la fin de la ligne commande. Le processus Unix lancé devient indépendant du processus père et s'exécute en parallèle.

Un processus lancé en direct (avant-plan ou foreground) peut être interrompu à l'aide des touches ***CTRL+Z*** puis repris avec la commande ***fg*** ou mis en arrière-plan avec la commande ***bg***.

La commande ***jobs*** permet de lister les processus s'exécutant en arrière plan.

Exercice:

- Lancer la commande *sleep 1000* en arrière-plan (*&*)
- Visualiser les processus en arrière-plan (background) avec la commande *jobs*.
- Visualiser les processus en cours avec la commande *ps*.
- Ramener la commande sleep en avant-plan (foreground) avec la commande *fg*.
- La remettre en arrière-plan (*CTRL+z*), vérifier avec la commande *jobs*.
- Lancer la commande *sleep 500* en arrière-plan.
- Ramener la commande *sleep 1000* en avant-plan et l'interrompre par *CTRL+c*.
- Tuer le process *sleep 500* avec la commande *kill*.

5.2. AU NIVEAU DU LANGAGE C

La fonction *system* du langage C permet d'exécuter à l'aide d'interpréteur (shell) une commande donnée sous forme de chaîne de caractères. Le sous-processus ignore les signaux SIGINT et SIGKILL.

La fonction *fork* permet de dupliquer un processus par clonage. Les deux processus sont identiques sauf en ce qui concerne leur numéro d'identification (PID) et leur résidence en mémoire.

Les fonctions *shmget* et *shmat* permettent le partage de variables globales stockées dans une zone mémoire spécifique.

La fonction *vfork* (utilisation obsolète): crée un nouveau processus avec une gestion dynamique de la mémoire...

Les fonctions de type *exec*: exécute avec ou sans retour à l'appelant (chaînage).

La fonction *waitpid* met le processus appelant en attente d'un signal de fin d'exécution.

Remarque: La variable *errno* du langage C (nécessite la directive *include "errno.h"*) est mise à jour avec une valeur non nulle lorsque une erreur survient lors d'un appel système (system call) ou d'une fonction d'une bibliothèque (library). Cette variable n'est pas mise à une valeur nulle lorsque tout ce passe bien (on récupère la valeur d'avant...)

Exercice: Expliquer le fonctionnement des programmes ps_exec.c et ps_fork.c suivants:

```
/*
 *  ps_exec.c
 *  ~~~~~~~~~
 */

#include <stdio.h>

main(argc, argv)
  int  argc; char **argv;
{
  printf("\nPS_EXEC: debut execution\n");
  printf("\nPS_EXEC: pid          = %d\n", getpid());
  printf("\nPS_EXEC: nb arguments = %d\n", argc);
  printf("\nPS_EXEC: argv[ 0 ]    = %s\n", argv[ 0 ]);
  printf("\nPS_EXEC: argv[ 1 ]    = %s\n", argv[ 1 ]);
  exit(0);
}  /* ---  fin  ps_exec.c  --- */


/*
 *  ps_fork.c
 *  ~~~~~~~~~
 */

#include <stdio.h>

main()
{
  int  pidp, pidf, pidk, pidw;

  pidp=getpid();
  printf("\nPS_FORK: debut execution : pid = %d\n", pidp);

/* creation du processus fils */
```

```
  pidk = fork();
  if ( pidk == -1) {
    printf("\nPS_FORK: creation de processus impossible\n"); exit(1);
  }

  printf("\nP+F: code execute par les 2 ! mais : getpid() = %d\n", getpid());

  if (pidk == 0) {  /* code execute par le process fils */
    pidf=getpid();
    printf("\nFILS: valeur retournee par le fork = %d\n", pidk);
    printf(  "FILS: pid(avant exec) = %d\n", pidf);
    sleep(5); execl("ps_exec", "liste des arguments", 0);
..../* tester erreur execl */
    printf(  "FILS: pid(apres exec) = %d\n", pidf);
    pidw=0;
  } else {  /* code execute par le process pere */
    pidf=0;
    printf("\nPERE: valeur retournee par le fork = %d\n", pidk);
    pidw=wait(0);
    printf("\nPERE: pid du fils se terminant = %d\n", pidw);
  }

  printf("\nP+F: fin du processus : %d\n", getpid());
  exit(0);
}  /* ---  fin  ps_fork.c  --- */
```

5.3. SEMAPHORES ET APPELS SYSTEME EN LANGAGE C

Les trois fonctions de base (system calls) sont **semget**, **semctl** et **semop**.

■ La fonction **semget** permet de récupérer l'identificateur du système de l'ensemble des *nsems* sémaphores associés au numéro utilisateur donné par *key*. La variable *semflg* permet de donner les droits d'utilisation de l'ensemble de sémaphores.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

int semget(key, nsems, semflg)

```
int key_t key;
int nsems, semflg;
```

Pour créer un sémaphore associé au nombre 3 par exemple, il suffit de créer et récupérer l'identificateur associé à l'ensemble réduit à un seul élément:

```
key=3;
nsems=1;
if ( ( semid=semget( key, nsems, 00620 | IPC_CREAT) ) == -1 ) {
  perror("creation semaphore impossible"); exit(1);
}
```

Les droits d'accès sont ici Lecture (00440) pour l'utilisateur et le groupe et Modification (00200) par l'utilisateur seulement.

■ La fonction **semctl** permet d'effectuer une opération *cmd* dite de contrôle sur le sémaphore *semnum* de l'ensemble précédemment crée et identifié par la variable *semid*. La variable *arg* est un récipient permettant l'échange d'information entre l'utilisateur et le système.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

int semctl(semid, semnum, cmd, arg)

```
int semid, semnum, cmd;
union semun {
val;
struct semid_ds *buf;
ushort *array;
} arg;
```

Le numéro *semnum* de sémaphore de l'ensemble associé à *semid* (et donc à *key*) varie de 0 à *nsems-1*, la valeur *nsems* ayant été déclarée à la création du groupe de sémaphores.

Si on note (key, semnum) le sémaphore semnum du groupe associé à key, alors pour initialiser à zéro la valeur du sémaphore (key, 0) on fera:

```
semnum=0;  arg.val=0;
if ( semctl(semid, semnum, SETVAL, arg) == -1 ) {
   perror("Pb. initialisation semaphore"); exit(1);
}
```

De même pour récupérer la valeur courante du sémaphore (key, 0), on fera:

```
semnum=0;
if ( ( val=semctl(semid, semnum, GETVAL, NULL) ) == -1 ) {
   perror("Pb. recuperation valeur semaphore"); exit(1);
}
printf(("Valeur du semaphore = %d\n", val);
```

L'opération de contrôle **IPC_SET** permet la mise à jour des permissions d'accès d'un sémaphore: *sem_perm.uid*, *sem_perm.gid* et *sem_perm.mode*. Elle n'est réalisable que par un process dont le pid est égale au pid du super utilisateur, au pid de création *sem_perm.cuid* ou à la valeur actuelle *sem_perm.uid* du sémaphore.

L'opération de contrôle **IPC_RMID** permet de supprimer l'identificateur système *semid* des tables du système. Tous les sémaphores du groupe cessent d'exister. Cette opération est réalisable dans les mêmes conditions que précédemment.

L'opération de contrôle **IPC_STAT** permet de récupérer dans la variable *arg* toutes les caractéristiques de tous les sémaphores associé à *semid*.

D'autres opérations de contrôle permettent de récupérer la valeur du *sempid* (GETPID) donnant le numéro d'identification du process ayant crée le sémaphore , la valeur *semncnt* (GETNCNT) et la valeur *semzcnt* (GETZCNT).

■ La fonction **semop** permet d'utiliser le sémaphore *sops[sem].sem_num* d'un groupe identifié par *semid*. Les services demandées dépendent des valeurs précisées dans les champs de la structure de données de type *sembuf*. Les différents services possibles sont appelées opérations normales par opposition aux opérations dites de contrôle.

```
#include <sys/types.h>
#include <sys/ipc.h>
```

```
#include <sys/sem.h>
int semop( semid, sops, nsops )
int semid;
struct sembuf *sops;
size_t nsops;
```

Pour demander un droit au sémaphore *semid*, on fera:

```
struct sembuf bops;
bops.sem_num=0;
bops.sem_op=-1;

if ( semop(semid, &bops, 1) == -1 ) {
  if (errno == EINTR) {
    printf("  Deblocage par signal\n");
    ...
  } else {
    printf("  Erreur decrementation semaphore\n");
    ...
  }
}
```

De même la libération d'un droit est réalisée par:

```
bops.sem_num=0;
bops.sem_op =1; /* > 0 : redonne un droit */
bops.sem_flg=0;

if ( semop(semid, &bops, 1) == -1 ) {
  perror("Pb. incrementation semaphore\n"); exit(2);
}
```

Exercice: Expliquer le fonctionnement des programmes sema1.c et sema2.c.

--ooOOoo--