Processus

I - La notion de processus

1) Définition

Un processus est un programme en cours d'exécution. L'exécution d'un processus est séquentielle, elle se fait instruction après instruction (au plus une instruction est exécutée au nom de tel processus).

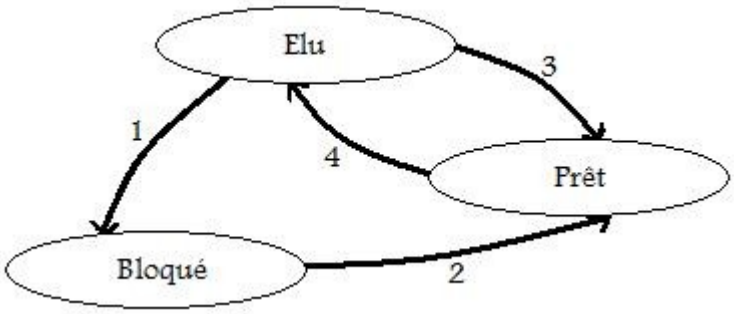
2) États d'un processus

Un processus peut être dans l'un des trois états suivants :

- Élu : les instructions sont en cours d'exécution (utilisent le CPU), on dit aussi actif
- Prêt : toutes les ressources nécessaires sont disponibles, le processus est prêt à utiliser le CPU (dès qu'il aura été choisi), on dit aussi activable
- Bloqué : le processus est en attente d'un événement extérieur (donnée en écriture sur un disque par exemple, saisie clavier, …), on dit aussi non activable.

Pour un processeur donné, un seul processus peut être élu, plusieurs peuvent être prêts ou bloqués.

Les transitions possibles entre les différents états sont les suivantes :



- T_1 : élu => bloqué
- T_2 : bloqué => prêt
- T_3 : élu => prêt
- T_4 : prêt => élu

On peut illustrer ces transitions par un exemple :

Transition T_1 : un processus A est en cours d'exécution, l'instruction actuelle est une requête d'entrée/sortie, le système d'exploitation va passer le processus A à l'état bloqué pour que le CPU ne reste pas inactif en attendant la fin de la requête, il interroge ensuite l'ordonnanceur pour savoir quel processus activer, B par exemple.
Transition T_2 : l'arrivée d'une interruption signalant la fin de l'opération d'entrée/sortie demandée par A va activer un processus de gestion d'interruption qui fera passer l'état du processus A de bloqué à prêt.
Transition T_3 : le processus B était en train de s'exécuter, mais supposons qu'il soit moins prioritaire que A, l'ordonnanceur peut décider de faire passer l'état de B d'élu à prêt (pour élire A).
Transition T_4 : admettons que A soit terminé, l'ordonnanceur fait passer B de prêt à élu.

3) Descripteur de processus

Un processus est connu du système par une structure de données appelée Descripteur de Processus(DdP) ou Process Control Bloc (PCB). Véritable carte d'identité du processus, cette structure mémorise notamment :

- Le numéro du processus
- Sa priorité
- Son état
- Son contexte (copie des registres du processeur...)
- Des pointeurs sur diverses tables (pagination mémoire par ex.)
- Des pointeurs sur le code et les données du processus concerné

Le rôle d'un descripteur est de permettre la manipulation simultanée de plusieurs processus par le système. Ce descripteur est plus ou moins riche suivant les systèmes d'exploitations.

4) Table centrale des processus

La table centrale est une structure utilisée par le S.E. pour mémoriser tous les processus en cours à un instant donné dans le système. Il s'agit en fait de la liste des descripteurs de processus.
L'ordre est celui d'arrivée du processus dans la mémoire.

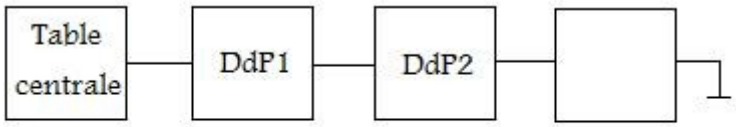


Table centrale des processus

5) Queue d'exploitation

La queue d'exploitation est une structure de données qui rassemble tous les processus qui peuvent s'exécuter à un instant donné. On y trouve donc le processus ELU et tous les processus PRETS. Cette liste est réalisée en chaînant les DdP de ces processus à travers la table centrale.

En général, le processus ELU est le premier processus de la queue d'exploitation. Il peut arriver qu'en période transitoire cette condition ne soit pas remplie.

6) Arborescence

Un processus peut créer un ou plusieurs autres processus, appelés processus enfants, ces derniers pouvant eux mêmes créer des processus enfants, on obtient une hiérarchie.

7) La fin d'un processus

Un processus peut se terminer suite aux évènements suivant :

a) Le processus a terminé sa tâche (exit sous Unix)
b) Il s'est produit une erreur (division par zéro, accès à un fichier inexistant par ex.)
c) Il est tué par un autre processus (kill sous Unix)

Sous Unix, dans tous les cas il est tué au final par son père (créateur ou auquel il a été rattaché), dans le a) c'est à la demande du fils, dans le b) c'est à la demande du fils si l'erreur a été prévue, à la demande du système d'exploitation sinon, dans c) c'est à la demande du père (si le père a lui-même un problème par ex.).

II - Ordonnancement

Dans un système multitâche, plusieurs processus sont en cours simultanément, mais le processeur ne peut, à un moment donné, exécuter qu'une seule instruction à la fois. Le processeur travaille en temps partagé (time sharing).

L'ordonnanceur (scheduler) est chargé de déterminer l'ordre d'exécution des processus par le processeur parmi les processus prêts.

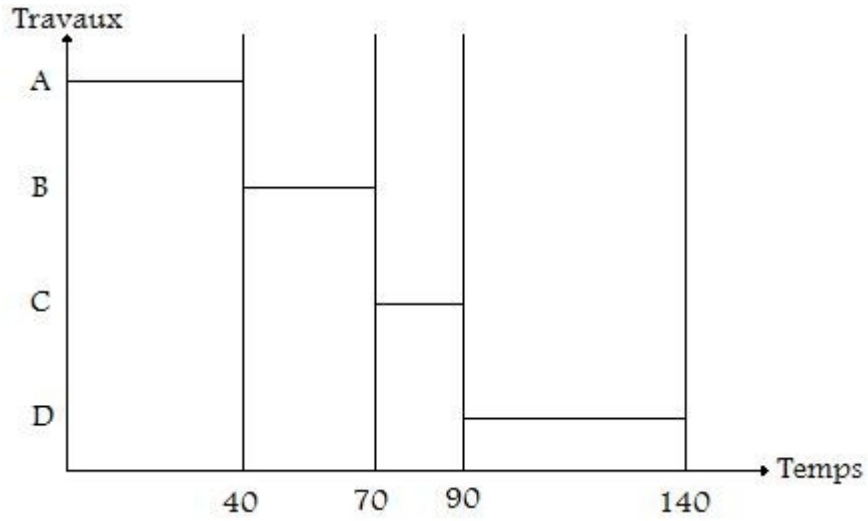
Le dispatcher a pour unique rôle de mettre à jour le processus actif. Il vérifie périodiquement si le premier processus de la queue d'exploitation est marqué « élu ». Si ce n'est pas le cas, il interrompt l'exécution du processus en cours, marque « prêt » dans son DdP, sauve son contexte et le met dans la queue d'exploitation à sa place (suivant les propriétés). Il prend alors le premier processus de la queue d'exploitation, restaure son contexte, le marque « élu » et réactive ce processus (met à jour les registres pour qu'en sortant du dispatcher, ce processus s'exécute). L'exécution du dispatcher n'est pas interruptible.
Algorithme du Dispatcher :

```
LOCK()
Le processus ELU en cours est-il le premier de la queue d'exploitation ?
Si oui alors
        Le laisser ELU
Sinon
        Sauvegarder le contexte du processus en cours dans son DdP
        Marqueur le processus en cours PRET
        Restaurer le contexte du premier processus prêt de la queue d'exploitation
        Marquer ce processus ELU
        Mettre le DdP du processus devenu PRET à sa place dans la queue d'exploitation
Fin si
Réactiver le processus ELU
UNLOCK()
```

Il existe plusieurs méthode d'ordonnancement :

1) Premier entré, premier sorti

La technique FIFO (First In First Out) consiste à prendre les processus dans leur ordre d'arrivée.
Soit par exemple quatre processus A, B, C et D, arrivés dans cet ordre, ne nécessitant pas d'entrées/sorties, avec une durée respective de 40, 30, 20, 50 ms.

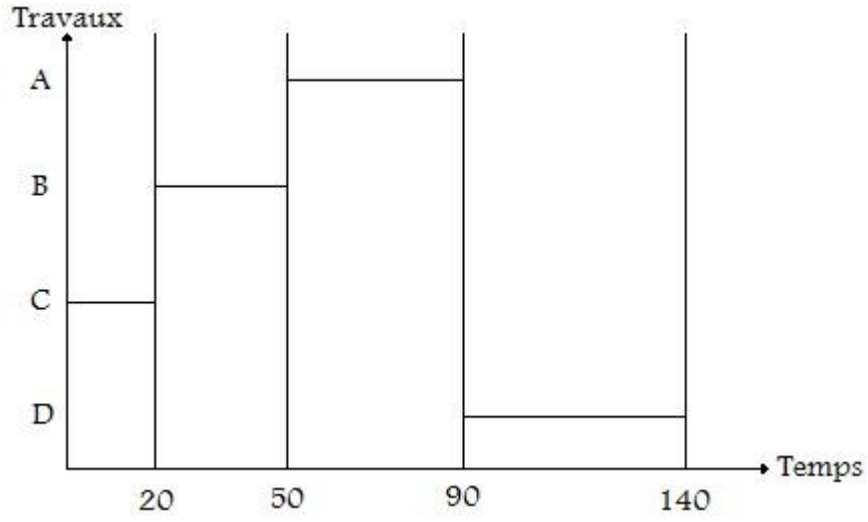


N.B. : Le temps de réponse est le temps qui s'écoule entre le moment où le processus est prêt et le moment où une réponse commence à arriver à l'utilisateur. Pour un processus non interactif, cela correspond au temps réel d'exécution plus le temps passé à attendre les ressources nécessaires, y compris le processeur. Pour un processus interactif, le temps de réponse peut être inférieur, car le processus peut commencer à envoyer des réponses à l'utilisateur avant qu'il soit terminé.

Ici, on obtient : T_A = 40 ms, T_B = 70ms, T_C = 90ms, T_D = 140ms.
T_M = (40+70+90+140)/4 = 85ms

2) Le plus court d'abord

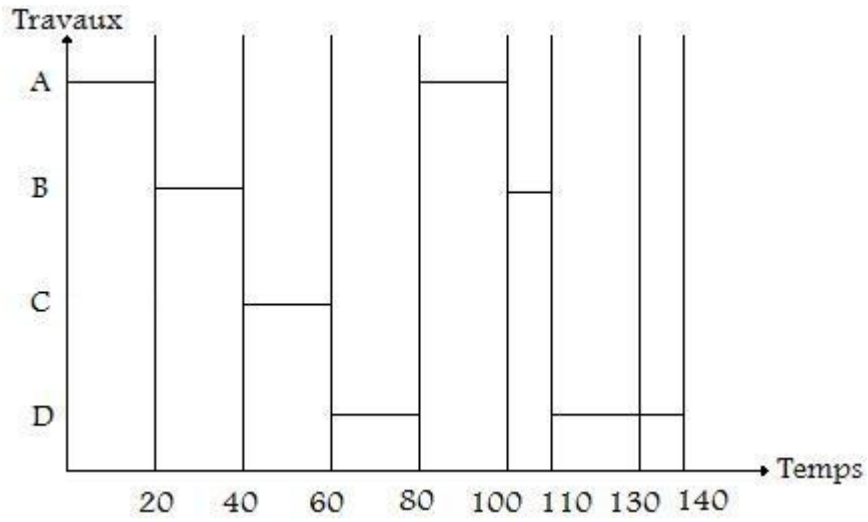
La technique SJF (Shortest Job First) consiste à prendre les processus suivant l'ordre croissant de leur durée d'exécution. Cela réduit le temps moyen de réponse.
On reprend l'exemple précédent.
L'utilisation du processeur se fera de la façon suivante :



On obtient : T_A = 90ms, T_B = 50ms, T_C = 20ms, T_D = 140ms.
T_M = (20+50+90+140)/4 = 75ms.

3) L'ordonnancement circulaire

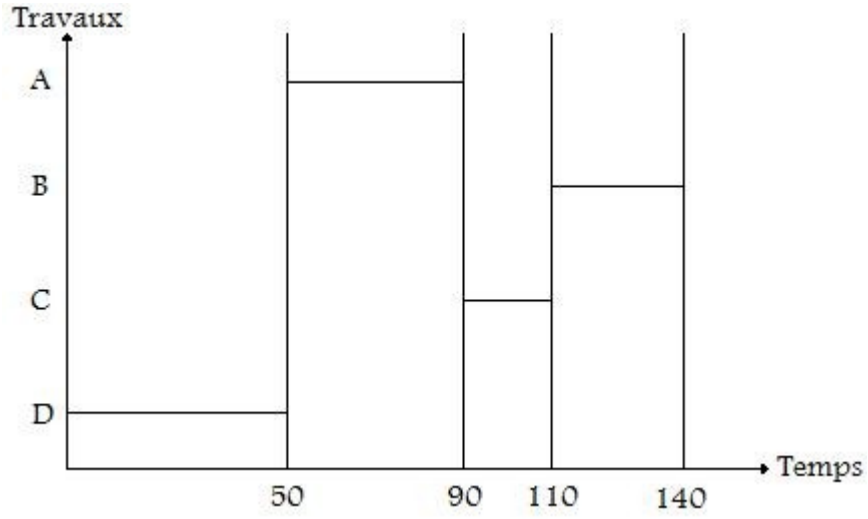
L'ordonnancement circulaire ou tourniquet (round robin) consiste à attribuer à chaque processus un intervalle fixe de temps de processeur, appelé quantum. Le processeur est donné à un autre processus lorsque le temps prévu pour le processus en cours s'est écoulé, lorsque le processus en cours est terminé ou lorsqu'il se bloque (en attendant une saisie clavier par exemple).
On reprend notre exemple avec un quantum de 20 ms.
L'utilisateur du processeur se fera de la façon suivante :



On obtient : T_A = 100ms, T_B = 110ms, T_C = 60ms, T_D = 140ms.
T_M = (100+110+60+140)/4 = 102.5ms

4) L'ordonnancement avec priorité

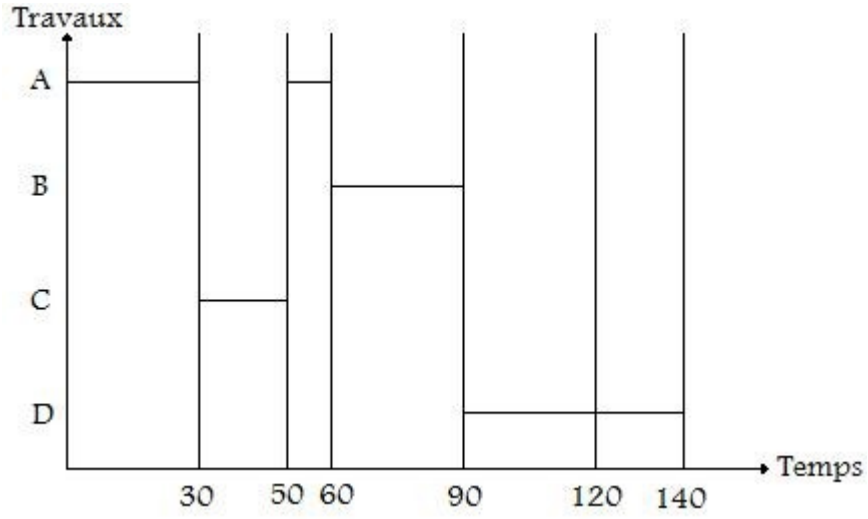
Dans l'ordonnancement avec priorité, chaque processus dispose d'une priorité. L'ordonnanceur choisit d'activer le processus prêt dont la priorité est la plus élevée. Pour éviter qu'un processus de haute priorité ne s'accapare le processeur, il existe des méthodes d'ajustement dynamique de la priorité. Par exemple, après chaque quantum de temps déterminé qui expire, la priorité du processus en cours diminue, jusqu'à ce qu'il y ait un processus prêt plus prioritaire que lui. Le processus courant cède alors sa place dans le processeur.
Supposons que dans l'exemple on ait P(A)=2, P(B)=3, P(C)=2 et P(D)=1 et que 1 corresponde à la priorité la plus élevée.
L'utilisation du processeur se fera de la façon suivante :



On obtient : T_A = 90ms, T_B = 140ms, T_C = 110ms, T_D = 50ms.
T_M = (90+140+110+50)/4 = 97.5ms.

5) Tourniquer avec priorité

Dans la méthode du tourniquet avec priorité, les processus attendent dans plusieurs files selon leur priorité. La file de la plus grande priorité est servie d'abord. A l'intérieur d'une file, l'ordonnancement se fait selon la méthode du tourniquet. Supposons que dans l'exemple on ait p(A)=1, P(B)=2, P(C)=1 et P(D)=2, que 1 corresponde à la priorité la plus élevée et que le quantum de temps soit de 30 ms.
L'utilisation du processeur se fera de la façon suivante :



On obtient : T_A = 30ms, T_B = 90ms, T_C = 50ms, T_D = 140ms.
T_M = (30+90+50+140)/4 = 85ms.

III - Communication inter-processus

1) Définition

Certains processus participant à une tâche complexe doivent communiquer entre eux et synchroniser leurs actions, c'est ce que l'on appelle la communication inter-processus.

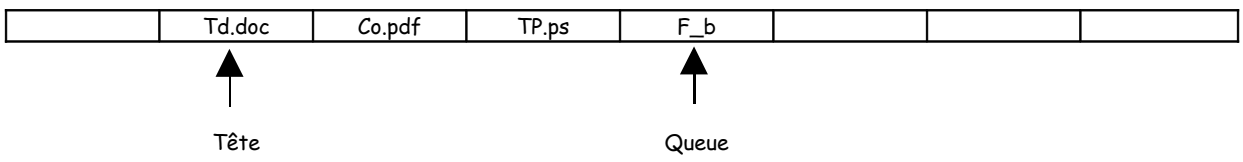
Il existe deux méthodes classiques de communication inter-processus : les variables partagées (shared variables) et l'échange de messages (message passing).

2) Variables partagées

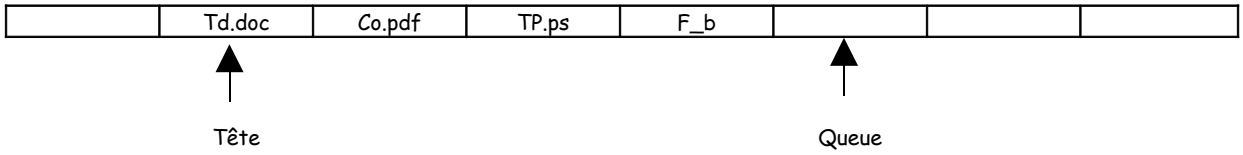
Exemple : un processus A est dédié à la gestion d'une imprimante, un processus B désire imprimer un document. B doit communiquer à A le nom et l'emplacement du fichier qu'il veut imprimer. Supposons que plusieurs processus veulent imprimer leurs résultats sur une imprimante commune. Les noms des fichiers à imprimer sont placés dans un tableau spécial fonctionnant comme une file : la variable *tête* qui indique le premier fichier dans la file, et la variable *queue* qui indique l'endroit d'insertion d'un nouveau fichier dans la file sont des variables partagées.
Supposons que le processus B veut imprimer un fichier, nommé *F_b*.

Processus B :
‹charger le fichier *F_b* dans [queue]› (instructionI1)
‹queue = queue+1› (instruction I2)

On obtient :
Après I1 :



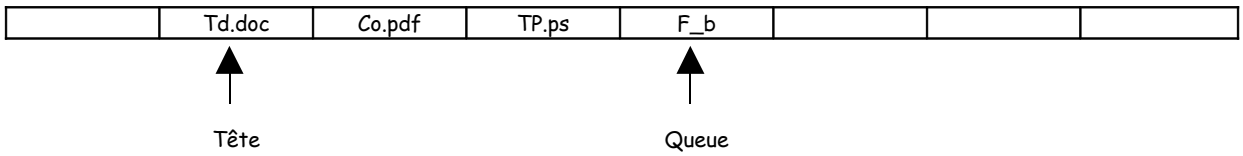
Après I2 :



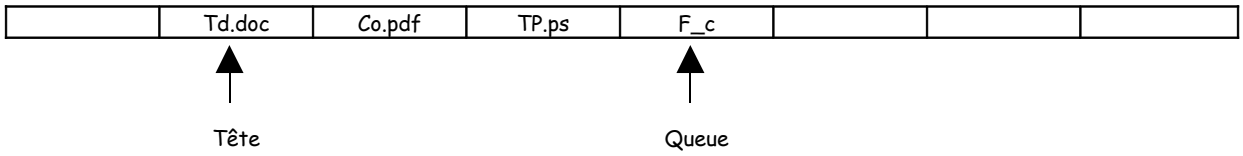
Exemple de conflit d'accès : Deux processus B et C veulent placer au même moment leurs fichiers (*F_b* et *F_c*) dans la file d'impression. Supposons que l'ordonnancement s'exécute alternativement : I1b, I1c, I2b, I2c.

Processus B :
<charger le fichier F_b dans [queue] > (instruction I1b)
<queue = queue+1> (instruction I2b)
Processus C :
<charger le fichier F_c dans [queue] > (instruction I1c)
<queue = queue+1> (instruction I2c)

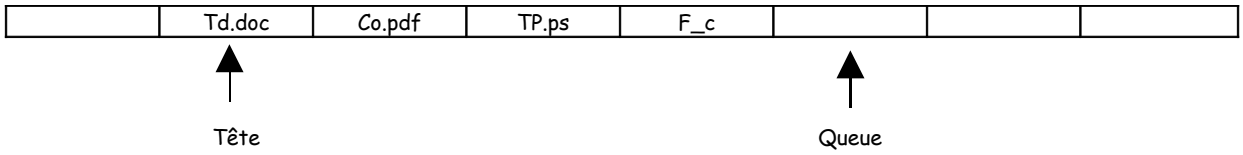
On obtient :
Après I1b :



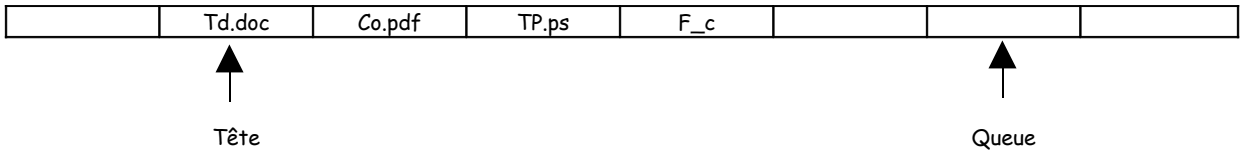
Après I1c :



Après I2b :



Après I2c :



Problème : Le fichier *F_b* ne sera jamais imprimé, car il a été écrasé par *F_c*.
Source du problème : la variable partagée *queue* est lue et modifiée par deux processus à la fois.

3) Échanges de messages

Les mécanismes de communication par variables partagées conviennent bien à des processus qui tournent sur le même processeur ou sur plusieurs processeurs possédant une mémoire commune. Pour des systèmes distribués, i.e. constitués de plusieurs processeurs ayant chacun leur propre mémoire et reliés par un réseau local, la communication inter-processus peut se faire par un échange de message :

- Un processus peut envoyer un message par réseau à un destinateur précis : send(destination, message)
- Un autre processus peut « consommer » un message par : receive(source, message)

Les problèmes qui se posent sont les suivants :

- Perte de message lors du passage par réseau => nécessité d'acquittement
- Perte d'acquittement => nécessité de retransmission d'un message
- Reconnaissance d'un message envoyé 2 fois (à cause de perte d'acquittement)
- Authentification des processus envoyant et recevant le message (protection)

4) Le problème des philosophes

C'est un problème classique très connu dans le domaine de la communication inter-processus :
5 philosophes (un philosophe représente un processus) sont assis autour d'une table. Sur la table se trouvent 5 assiettes remplies de spaghettis et 5 fourchettes (les fourchettes sont des ressources partagées) ; pour manger les spaghettis il faut 2 fourchettes (les spaghettis sont très glissants). Si un philosophe n'a pas faim, il pense (exécute un programme). S'il a faim il essaye d'attraper 2 fourchettes voisines (accéder à 2 ressources) et de manger.
Deux risques se présentent :
Inter Blocage (deadlock, étreinte fatale) : Les 5 philosophes attrapent au même moment leur fourchette gauche et la gardent jusqu'à pouvoir attraper aussi leur fourchette droite, les 5 processus sont indéfiniment bloqués.
Famine (starvation) : les 5 philosophes attrapent au même moment leur fourchette gauche, voient que leur fourchette droite n'est pas disponible, reposent leur fourchette gauche, et retente à nouveau d'attraper les 2 fourchettes. S'ils répètent ces tentatives toujours tous au même moment, aucun n'arrive à manger alors qu'il y a assez de spaghettis pour tous !

5) La notion de sémaphore

La plupart du temps, les processus effectuent des opérations qui ne conduisent pas à des situations de conflit. Seules certaines parties bien spécifiques du code font appel à des ressources partagées, on appelle ces parties sections spécifiques.

Les conflits d'accès concurrents aux variables partagées ont plusieurs solutions possibles (*sémaphores*, *moniteurs*,...). L'idée commune de ces solutions est l'exécution mutuelle ; une variable partagée ne doit pas être lue ni modifiée par plus d'un processus à la fois. Si d'autres processus veulent y accéder, ils doivent attendre que le premier processus ait fini.

Prenons le cas du sémaphore :

Un sémaphore est une variable entière positive (un compteur prenant les valeurs 0, 1, 2, …) qui ne peut être modifiée que par les primitives *Wait* et *Signal*. L'opérateur *Signal* incrémente le sémaphore S de 1. L'opérateur *Wait* décrémente le sémaphore S de 1, à condition que S ne devienne jamais négatif. Si S=0, l'opération *Wait(S)* doit attendre que S devienne positif (i.e. qu'une opération *Signal(S)* soit effectuée par un autre processus).

Exemple d'application :
On considère deux processus A et B, on veut que l'instruction B2 soit exécutée avant l'instruction A2. On initialise un sémaphore S à 0, puis :

Processus A	Processus B
Instruction A1	Instruction B1
Wait(S)	Instruction B2
Instruction A2	Signal(S)
	Instruction B3

ANNEXE : Exemple de la recette de Tanenbaum.

Pour bien montrer la différence entre programme et processus, nous reprendrons une image de Tanenbaum. Supposez qu'un informaticien décide de faire un gâteau pour l'anniversaire de sa fille. Pour être certain de le réussir, il ouvre devant lui son livre de recettes de cuisine à la page voulue. La recette lui précise les denrées dont il a besoin, ainsi que les instruments nécessaires. A ce stade, nous pouvons comparer la recette sur le livre de cuisine au programme, et l'informaticien au processeur. Les denrées sont les données d'entrée, le gâteau est la donnée de sortie, et les instruments sont les ressources nécessaires. Le processus est l'activité dynamique qui transforme les denrées en gâteau.

Supposez que, sur ces entrefaites, le fils de l'informaticien vient se plaindre qu'il a été piqué par une abeille. L'informaticien interrompt son travail, enregistre l'endroit où il en est, extrait de sa bibliothèque le livre de première urgence, et localise la description des soins à apporter dans ce cas. Lorsque ces soins sont donnés et que son fils est calmé, l'informaticien retourne à sa cuisine pour poursuivre l'exécution de sa recette.

On constate qu'il y a en fait deux processus distincts avec deux programmes différents. Le processeur a partagé son temps entre les deux processus, mais ces deux processus sont indépendants.

Supposez maintenant que la fille de l'informaticien vient lui annoncer de nouveaux invités. Devant le nombre, l'informaticien décide de faire un deuxième gâteau d'anniversaire identique au premier. Il va devoir partager son temps entre deux processus distincts qui correspondent néanmoins à la même recette, c'est à dire au même programme. Il est évidemment très important que le processeur considère qu'il y a deux processus indépendants, qui ont leur propre état, même si le programme est le même.

Création de processus

Plusieurs méthodes :

- création statique : création à l'initialisation d'un nombre fixe de processus banalisés (pb : occupent place mémoire même si non utilisés).
- création dynamique : il existe une opération de création (et de destruction) de processus qui définit la suite d'instructions, l'état initial des données, des variables du nouveau processus et des registres du processeur.

Une opération de création de processus se présente sous la forme suivante :
id := créer_processus (programme, contexte), où les paramètres définissent l'état initial. La valeur retournée éventuellement par cette opérations permet d'identifier le processus qui a été ainsi créé.

La plupart des systèmes qui permettent la création dynamique de processus considèrent que la relation processus créateur – processus créé est importante, d'où une structure d'arbre : le processus créé devient fils du processus créateur. Si un processus est détruit, on peut parcourir sa descendance pour la détruire au préalable. Il arrive qu'un processus père soit détruit avant que l'un de ses fils ne soit terminé, le fils peut alors être rattaché par exemple au processus qui a initialisé le travail (login).

L'exemple d'Unix
Dans le système Unix, la création dynamique de processus se fait par la fonction fork (). Cette fonction crée un processus qui est une exacte copie (programme, données, variables, registres processeur) du processus qui demande la création. La seule distinction entre le processus créateur et le processus créé, est la valeur retournée par la fonction fork () :

- le processus créateur reçoit le numéro du processus créé
- le processus créé reçoit la valeur 0

Exemple :

```
id_fils := fork ();   { création du fils}
si id_fils = 0 alors {il s'agit du processus fils}
sinon                {il s'agit du processus père}
```

La fonction fork () crée le processus fils, ensuite le processus père continue, tandis que le processus fils démarre avec une copie de la partie du programme du père qui suit l'appel de la fonction. Le processus fils à ses propres zones de données et de variables (identiques à celles du père au départ). Par la suite, les deux processus sont indépendants et n'ont pas de données communes à l'exception des objets externes.

Si le processus père se termine avant le processus fils, le fils est rattaché au processus racine de façon à conserver la structure d'arbre. Le processus père peut attendre la terminaison d'un de ses fils par la fonction (en langage C) : id_fils := wait (&status); qui retourne dans id_fils le numéro d'un processus qui est terminé, en mettant dans la variable *status* un code indiquant la façon dont ce processus s'est terminé (on peut mettre NULL à la place de &status si on n'a pas besoin de ce code).

Par ailleurs le système Unix fournit une fonction exec () qui permet à un processus de changer le programme en cours d'exécution. Cette fonction n'affecte évidemment que le processus demandeur. Elle remplace le code des instructions, supprime les données, variables et registres qui étaient relatifs à l'ancien programme et en les initialise pour le nouveau. La combinaison de la fonction fork () et de cette fonction exec () pour le processus fils permet de réaliser la fonction créer_processus mentionnée au début.