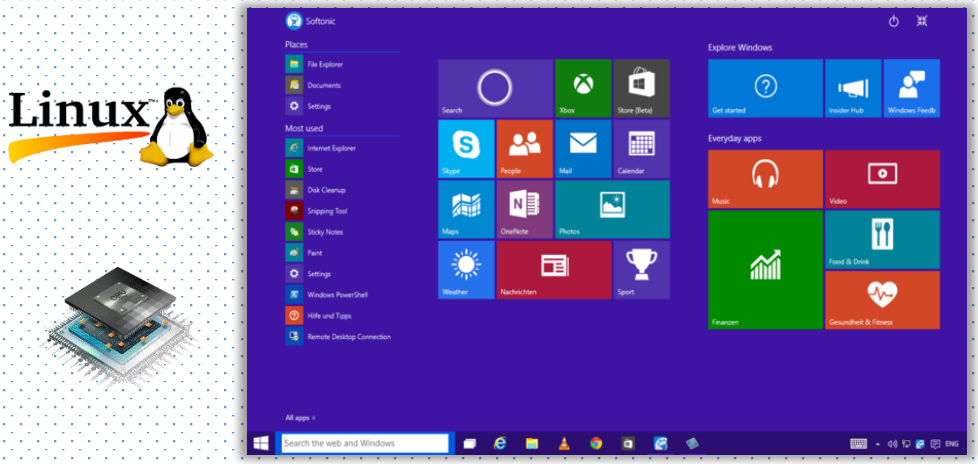
UNIVERSITE VIRTUELLE
DE CÔTE D'IVOIRE



SYSTÈME D'EXPLOITATION

FILIERES

- Développement d'Applications et e-Services (DAS)
- Réseaux et Sécurité Informatique (RSI)
- Multimédia et Arts Numériques (MMX)
- Bases de Données (BD)

AUTEUR

Irié Bi Dizan Paul, Mamadou BAKOUAN

DATE : 19/08/2017

TABLE DES MATIERES

LEÇON 2 : LES PROCESSUS

2.1 Objectifs

À la fin de cette leçon, vous serez capable de :

- ✓ Connaître le rôle et la structure d'un processus
- ✓ Identifier les algorithmes d'ordonnancement des processus par le système d'exploitation
- ✓ Connaitre les politiques employées par le système d'exploitation pour éviter les inter-blocages entre processus.
- ✓ Savoir comment le système d'exploitation s'y prend pour synchroniser les processus sur une ressource critique
- ✓ Identifier les différents modes de communication des processus

2.2 Introduction

Dans un système multitâche, la ressource la plus importante d'une machine est le processeur. Cette ressource est allouée à un et un processus sélectionné parmi un ensemble des processus éligibles. Par conséquent, il convient à bien gérer ce dernier afin de le rendre plus productif. En effet, un système d'exploitation dispose d'un module qui s'occupe de l'allocation du processeur en l'occurrence le Dispatcheur. Ce module est exécuté chaque fois qu'un processus se termine ou se bloque dans le but de réquisitionner le processeur pour un autre processus. En plus de ce Dispatcheur, un système d'exploitation dispose d'un autre module permettant ainsi la mise en ordre des processus qui demandent le processeur.

2.3 Notion de processus

Un processus se définit comme étant un programme en cours d'exécution.

La notion de processus est dynamique : un processus naît lors du chargement d'un programme et meurt à la fin de l'exécution du programme.

A un processus est associé des ressources physiques (mémoire, processeur, entrée/sortie, …) et logiques (données, variables, …). Contrairement à un programme (texte exécutable) qui a une existence statique.

Un programme est une entité composée de séquences d'instructions agissant sur un ensemble de données. C'est du code obtenu par compilation d'un fichier source suivie d'une phase d'édition des liens.

La notion de programme est essentiellement statique car c'est un fichier stocké sur disque.

Des processus distincts peuvent exécuter un même programme (programme réentrant).

Le système d'exploitation manipule deux types de processus :

- **Processus système :** processus lancé par le système (*init* processus père des tous les processus du système)
- **Processus utilisateur :** processus lancée par l'utilisateur (commande utilisateur)

Dès sa création, un processus reçoit les paramètres suivants :

PID : identificateur du processus (numéro unique)

PPID : identificateur du processus père

UID : identificateur de l'utilisateur qui a lancé le processus

GID : identificateur du groupe de l'utilisateur qui a lancé le processus

- Il existe des appels système permettant de créer un processus,
- Un processus peut (père) créer d'autres processus (fils) qui hérite charger son contexte et lancer son exécution (***fork***, ***exec*** sous Unix). les descripteurs de son père. Ce dernier à son tour crée d'autres processus. Un processus a un seul père mais peut avoir plusieurs fils.
- Les processus peuvent se terminer ou ils peuvent être éliminés par d'autres processus (la primitive ***kill***). A la destruction d'un processus, on libère toutes les ressources qu'il avait.
- Dans certains cas, la destruction d'un processus entraîne l'élimination de ces descendants ; cette opération n'affecte pas les processus qui peuvent continuer indépendamment de leur père (processus orphelins).

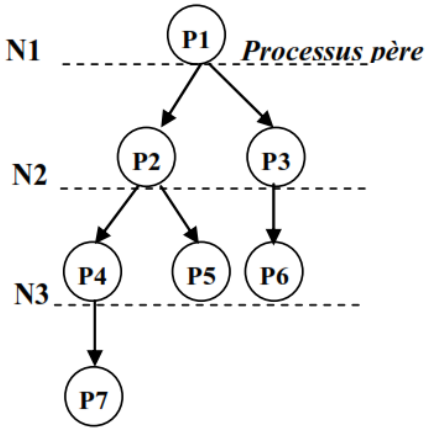


Figure 1 : Hiérarchie des processus

2.4 Les états d'un processus

Dans les systèmes mono programmés, un programme ne quitte pas l'unité centrale avant de terminer son exécution. Pendant cette période, il dispose de toutes les ressources de la machine. Par contre, ce n'est pas le cas dans les systèmes multiprogrammés et temps-partagé, un processus peut se trouver dans l'un des états suivants :

1- **Elu :** (en cours d'exécution) : si le processus est en cours d'exécution

2- **Bloqué** : attente qu'un événement se produit ou bien d'une ressource pour pouvoir continuer
3- **Prêt :** si le processus dispose de toutes les ressources nécessaires à son exécution à l'exception du processeur.

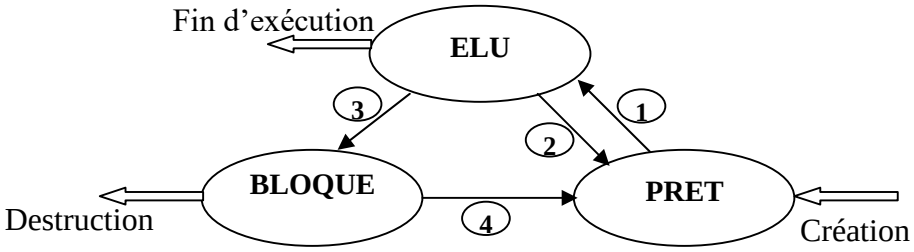


Figure 2 : *Diagramme de transition d'un processus*

🕘 ***Sémantique des Transitions***

(1) : Allocation du processeur au processus sélectionné
(2) : Réquisition du processeur après expiration de la tranche du temps par exemple
(3) : Blocage du processus élu dans l'attente d'un événement (E/S ou autres)
(4) : Réveil du processus bloqué après disponibilité de l'événement bloquant (Fin E/S, etc…) Notons que nous avons omis les états Création et Terminaison de processus puisqu'ils n'interviennent pas dans cette étude.

❖ États particuliers

Selon les systèmes d'exploitation, ces différents états peuvent aussi être possibles :

- ***Zombie*** :

  Si un processus terminé ne peut pas être déchargé de la mémoire, par exemple, si un de ses fils n'est pas terminé, il passe dans un état appelé *zombie*.

- ***Swappé*** :

  Lorsqu'un processus est transféré de la mémoire centrale dans la mémoire virtuelle, il est dit « swappé ». Un processus swappé peut être dans un état *endormi* ou *prêt*.

- ***Préempté*** :

L'ordonnanceur a décidé de suspendre l'activité d'un processus. Par exemple, un processus qui consomme trop de temps CPU finira par être préempté. Un ordonnanceur préemptif utilise aussi l'indice de priorité (nice) pour décider le processus qui sera préempté.

- ***Exécution en espace utilisateur*** :

  L'exécution a lieu dans un espace limité : seules certaines instructions sont disponibles.

- ***Exécution en espace noyau*** :

  Par opposition au mode utilisateur, l'exécution du processus n'est pas limitée. Par exemple, un processus dans cet état peut aller lire dans la mémoire d'un autre.

❖ **Mode de comportement des processus**

- séquentiel,
- cyclique,
- périodique,
- etc.

❖ **Terminaison de processus**

- fin correcte du code
- terminaison imposée par le noyau (erreur, dépassement de ressource)
- auto terminaison (suicide) sur exception
- terminaison forcée par un autre processus (« CTRL/C »sous Unix/Linux)
- jamais : processus cyclique permanent ("démon")
- jamais : boucle infinie par erreur

2.5 Cycle d'exécution d'un processus

Un système d'exploitation doit en général traiter plusieurs taches en même temps. Comme il n'a la plupart du temps qu'un seul processeur, il résout ce problème grâce à un pseudo-parallélisme. Il traite une tâche à la fois, s'interrompt et passe à la suivante. La commutation des taches est très rapide, l'ordinateur donne l'illusion d'effectuer un traitement simultané.

L'exécution d'un processus peut être vue comme une séquence de phases. Chaque phase comprend deux cycles : un cycle d'exécution (ou calcul) réalisé par le processeur et un cycle d'entrée sortie assuré par le canal. La dernière phase de tout processus doit comprendre obligatoirement un seul cycle dans lequel sera exécutée la requête informant le système

d'exploitation sur la terminaison du processus. Cet appel permet au système de restituer les ressources utilisées par le processus qui vient de terminer.

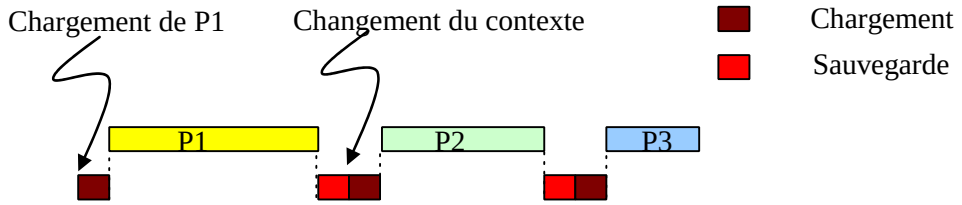


Figure 3 : *Cycle d'exécution d'un processus*

2.6 Le bloc de contrôle d'un processus

Lorsqu'un processus est temporairement suspendu, il faut qu'il puisse retrouver l'état où il se trouvait au moment de suspension, il faut que toutes les informations dont il a besoin soient sauvegardées pendant sa mise en attente.

Notons que le contexte d'un processus dépend du système, mais dans tous les cas c'est un bloc de contrôle de processus (BCP) qui contient toute information nécessaire à la reprise d'un processus interrompu : Etat du processus (prêt, suspendu, ..), quantité de mémoire, temps CPU (utilisé, restant), priorité, compteur d'instructions (indique l'adresse de l'instruction suivante devant être exécutée par le processus), etc.

Les BCP sont rangés dans une table (table des processus) qui se trouve dans l'espace mémoire du système.

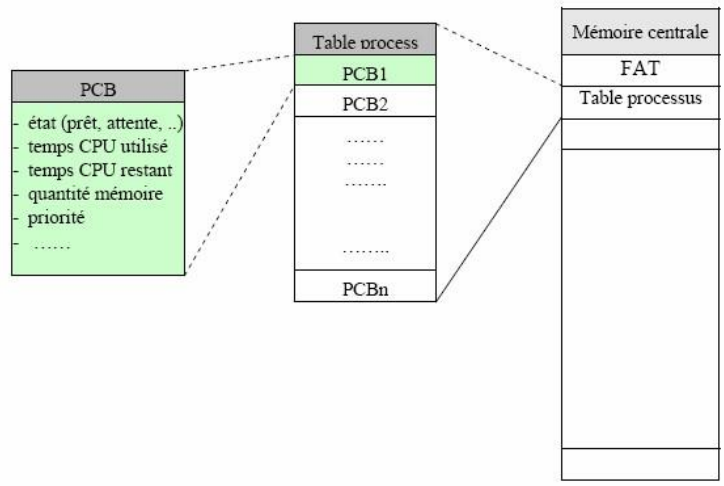


Figure 4 : *Bloc de contrôle de processus (PCB)*

2.7 Ordonnancement des processus

Chaque fois, que le processeur devient inactif, le système d'exploitation doit sélectionner un processus de la file d'attente des processus prêts, et lui passe le contrôle. D'une manière plus concrète, cette tâche est prise en charge par deux routines système en l'occurrence le ***Dispatcheur*** et le ***Scheduleur (Ordonnanceur)***.

2.7.1 Le Dispatcheur

Il s'occupe de l'allocation du processeur à un processus sélectionné par l'Ordonnanceur du processeur. Une fois alloué, le processeur doit réaliser les tâches suivantes :

- ***Commutation de contexte :*** sauvegarder le contexte du processus qui doit relâcher le processeur et charger le contexte de celui qui aura le prochain cycle processeur
- ***Commutation du mode d'exécution :*** basculer du mode Maître (mode d'exécution du dispatcheur) en mode utilisateur (mode d'exécution du processeur utilisateur)
- ***Branchement :*** se brancher au bon emplacement dans le processus utilisateur pour le faire démarrer.

2.7.2 L'Ordonnanceur

Certains systèmes d'exploitation utilisent une technique d'ordonnancement à deux niveaux qui intègre deux types d'Ordonnanceurs :

Ordonnanceur du processeur : c'est un Ordonnanceur court terme opère sur une ensemble du processus présents en mémoire. Il s'occupe de la sélection du processus qui aura le prochain cycle processeur, à partir de la file d'attente des processus prêts.

Ordonnanceur de travail : ou Ordonnanceur long terme, utilisé en cas d'insuffisance de mémoire, son rôle est de sélectionné le sous ensemble de processus stockés sur un disque et qui vont être chargés en mémoire. Ensuite, il retire périodiquement de la mémoire les processus qui sont restés assez longtemps et les remplace par des processus qui sont sur le disque depuis trop de temps.

Nous distinguons plusieurs algorithmes d'ordonnancement, les plus répandus sont :

- Ordonnancement Premier Arrivé Premier Servi
- Ordonnancement du plus court d'abord
- Ordonnancement circulaire : Tourniquet
- Ordonnancement circulaire à plusieurs niveaux
- Ordonnancement avec priorité

2.7.3 Les critères de scheduling

L'ordonnancement est la partie du système d'exploitation qui détermine dans quel ordre les processus prêts à s'exécuter (présents dans la file des prêts) seront élus.

Ses objectifs sont :

- Assurer le plein usage du CPU (agir en sorte qu'il soit le moins souvent possible inactifs);
- Réduire le temps d'attente des utilisateurs.
- Assurer l'équité entre les utilisateurs.

Les divers algorithme de scheduling du processeur possèdent des priorités différentes et peuvent favoriser une classe de processus plutôt qu'une autre. Pour choisir quel algorithme utiliser dans une situation particulière nous devons tenir compte des propriétés des divers algorithmes. Plusieurs critères ont été proposés pour comparer et évoluer les performances des algorithmes de scheduling du processeur. Les critères les plus souvent utilisé sont :

- ***Utilisation du processeur :*** Un bon algorithme de scheduling sera celui qui maintiendra le processeur aussi occupe que possible.

- ***Capacité de traitement :*** C'est la quantité de processus terminé par unité de temps.

- ***Temps de restitution :*** C'est le temps s'écroulant entre la soumission du travail et sa terminaison.

- ***Temps de réponse :*** C'est le temps passer dans la file d'attentez des processus prêts avant la première exécution. Pour calculer le temps de réponse moyen (**TRM**) d'exécution des processus on utilise la formule suivante :

TRM = $\sum_{i=0}^{n} \boldsymbol{TRi/n}$ **,** avec TRi = date fin – date arrivée.

- ***Temps d'attente :*** C'est le temps passé à attendre dans la file d'attente des processus prêts.

Le temps d'attente moyen d'exécution des processus (**TAM**) est calculé comme suit :

TAM = $\sum_{i=0}^{n} \boldsymbol{TAi/n}$ **,** avec TAi = TRi - temps d'exécution.

Les algorithmes d'ordonnancement peuvent être classes en deux grandes catégories :

- **Ordonnanceur non préemptif** : Dans un système à ordonnancement non préemptif ou sans réquisition le système d'exploitation choisi le prochain processus à exécuter et lui alloue le processeur jusqu'à ce qu'il se termine ou qu'il se bloque. Il n'y a pas de réquisition même si le processus s'exécuter pendant des heures.

- **Ordonnanceur préemptif** : dans un schéma d'ordonnanceur préemptif ou avec réquisition le système d'exploitation peut retirer à n'importe quel moment le processeur à un processus même si ce dernier est en cours d'exécution. Au niveau des algorithmes d'ordonnancement préemptif lorsqu'un processus est sélectionné il s'exécute pendant un délai déterminé après ce délai il est remplacé par un autre processus.

Remarque :

Pour représenter schématiquement l'évolution dans le temps des processus, on recourt habituellement à des diagrammes de Gantt.

2.8 Les algorithmes d'ordonnancement

2.8.1 Ordonnancement FCFS (first come first Served)

2.8.1.1 Principe

On l'appelle aussi ordonnancement FIFO (First In, First Out) ; les processus sont rangés dans la file d'attente des processus prêts selon leur ordre d'arriver. Les règles régissant cet ordonnancement sont :

1. Quand un processus est prêt à s'exécuter, il est mis en queue de la file d'attente des processus prêts.
2. Quand le processeur devient libre, il est alloué au processus se trouvant en tête de file d'attente des processus prêts.
3. Le processus élu relâche le processeur s'il se termine ou s'il demande une entrée sortie.

Cet algorithme est classe dans la catégorie des ordonnanceurs non préemptifs ou sans réquisitions. Il n'est pas adapter à un système partagé car dans un tel système chaque utilisateur obtient le processeur à des intervalles réguliers.

2.8.1.2 Exemple d'algorithme FCFS :

Considérons 5 travaux A, B, C, D, E dont le temps d'exécution respectifs et leur arrivage respectifs sont données dans le tableau suivant :

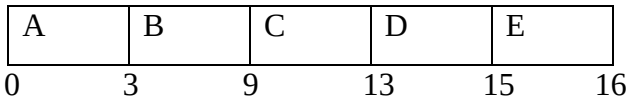
Processus	**Temps d'exécution**	**Temps d'arrivée**
A	3	0
B	6	1
C	4	4
D	2	6
E	1	7

En utilisant un algorithme d'ordonnancement FCFS (Fifo) donnez :

- le diagramme de Gantt,
- le temps de séjour de chaque processus,

- le temps moyen de séjour des processus,
- le temps d'attente de chaque processus,
- le temps moyen d'attente des processus

- **Le diagramme de Gantt des processus**



- **Temps de séjour de chaque processus**

Pour obtenir le temps de séjour, on fait :
TSi = temps fin exécution – temps d'arrivée

processus	Temps de séjour
A	3-0=3
B	9-1=8
C	13-4=9
D	15-6=9
E	16-7=9

- **Temps moyen de séjour des processus**

TMS = $\sum_{i=1}^{n} TSi/n$

TMS = (3+8+9+9+9) / 5= 7.6

- **Temps d'attente de chaque processus**

Pour obtenir le temps d'attente, on fait :
TAi = temps de séjour – temps exécution

processus	Temps d'attente
A	3-3=0

B	8-6=2
C	9-4=5
D	9-2=7
E	99-1=8

- **Temps moyen d'attente des processus**

  TMA = $\sum_{i=1}^{n} TAi/n$

  TMA = 22/5 = 4.4

2.8.2 L'algorithme d'ordonnancement SJF (short job first)

2.8.2.1 Principe

SJF ou *SRTF (Shortest Remaining Time first) (plus court temps d'exécution restant PCTER)* choisit de façon prioritaire les processus ayant le plus court temps d'exécution sans réellement tenir compte de leur date d'arrivée.

Dans cet algorithme les différents processus sont rangés dans la file d'attente des processus prêts selon un ordre croissant de leur temps d'exécution (cycle de calcul). Les règles régissant cet ordonnancement sont :

1. Quand un processus est prêt à s'exécuter, il est inséré dans la file d'attente des processus prêts à sa position approprie.
2. Quand le processeur devient libre, il est assigné au processus se trouvant en tête de la file d'attente des processus prêts (ce processus possède le plus petit cycle processeur.). Si deux processus ont la même longueur de cycle, on applique dans ce cas l'algorithme FCFS.
3. Si le système ne met pas en œuvre la réquisition, le processus élu relâche le processeur s'il se termine ou s'il demande une entrée sortie. Dans le cas contraire, le processus élu perd le processeur également. Quand un processus ayant un cycle d'exécution inférieur au temps processeur restant du processus élu, vient d'entrer dans la file d'attente des prêts. Le processus élu dans ce cas sera mis dans la file d'attente des éligibles, et le processeur est alloué au processus qui vient d'entrer.

❖ **Caractéristiques de l'Ordonnanceur**

- Cet Ordonnanceur peut être **avec ou sans réquisition**
- Implémentation difficile, car il n'existe aucune manière pour connaître le cycle suivant du processeur.

2.8.2.2 Application

Considérons 5 travaux A, B, C, D, E dont le temps d'exécution respectifs et leur arrivage respectifs sont données dans le tableau suivant :

Processus	**Temps d'exécution**	**Temps d'arrivée**
A	3	0
B	6	1
C	4	4
D	2	6
E	1	7

- En utilisant un algorithme **d'ordonnancement SJF en mode non préemptif** donnez :

- le diagramme de Gantt,
- le temps de séjour de chaque processus,
- le temps moyen de séjour des processus,
- le temps d'attente de chaque processus,
- le temps moyen d'attente des processus

- **Le diagramme de Gantt des processus**

A	B	E	D	C
0	3	9	10	12

- **Temps de séjour de chaque processus**

Processus	Temps de séjour
A	3-0=3
B	9-1=8
C	16-4=12
D	12-6=6
E	10-7=3

- **Temps moyen de séjour des processus**

TMS =$\sum_{i=1}^{n} TSi/n$

TMS = (8+3+6+12)/5=29/5 = 5.8

- **Temps d'attente de chaque processus**

Pour obtenir le temps d'attente, on fait :
TAi = temps de séjour – temps exécution

Processus	Temps d'attente
A	3-3=0
B	2
C	12-4=8
D	6-2=4
E	3-1=2

- **Temps moyen d'attente des processus**

TMA = $\sum_{i=1}^{n} TAi/n$

TAM=(0+2+8+4+2)/5= 16/5 = 3.2

En utilisant un algorithme **d'ordonnancement SJF en mode préemptif (avec requisition)** donnez :

- le diagramme de Gantt,
- le temps de séjour de chaque processus,
- le temps moyen de séjour des processus,
- le temps d'attente de chaque processus,
- le temps moyen d'attente des processus

- **Le diagramme de Gantt des processus**

A	B	C	E	D	B
0	3	4	8	9	11	16

- **Temps de séjour de chaque processus**

Processus	Temps de séjour
A	3-0=3
B	16-1=15
C	8-4=4
D	11-6=5
E	9-7=2

- **Temps moyen de séjour des processus**

TMS = $\sum_{i=1}^{n} TSi/n$

TSM= (3+15+4+5+2)/5= 29/5 = 5.8

- **Temps d'attente de chaque processus**

Pour obtenir le temps d'attente, on fait :
TAi = temps de séjour – temps exécution

Processus	Temps d'attente
A	3-3=0
B	15-6 = 9
C	4-4=0
D	5-2=3
E	2-1=1

- **Temps moyen d'attente des processus**

TMA = $\sum_{i=1}^{n} TAi/n$

TAM = (0+9+0+3+1)/5 = 13/5

2.8.3 Algorithme d'ordonnancement basé sur les priorités

2.8.3.1 Principe

Dans cet algorithme les processus sont rangés dans la file d'attente des processus prêts par ordre décroissant de priorité. L'ordonnancement dans ce cas est régi par les règles suivantes :

- Quand un processus est admis par le système il est insérer dans la file d'attente des processus prêts à sa position appropries (selon la valeur de priorité)
- Quand le processeur devient libre il est alloue au processus se trouvant en tête de file d'attente des processus prêts
- Dans un cas de non préemption un processus élu relâche le processeur que s'il se termine ou se bloque.

- ***Remarque :***

Si le système met en œuvre la réquisition, quand un processus de priorité supérieure à celle du processus élu entre dans l'état prêt ; le processus élu sera mis dans la file d'attente des éligibles à la position approprie, et le processeur est alloué au processus qui vient d'entrer.

❖ ***Caractéristiques de l'Ordonnanceur***

Les principales caractéristiques sont :

- Peut être avec ou sans réquisition

- Un processus de priorité basse risque de ne pas être servi (problème de famine) d'où la nécessité d'ajuster périodiquement les priorités des processus prêts. L'ajustement consiste à incrémenter graduellement la priorité des processus de la file d'attente des éligibles (par exemple à chaque 15 mn on incrémente d'une unité la priorité des processus prêts)

***Remarque** : la priorité adoptée est celle la plus élevée.*

2.8.3.2 Application

Considérons 5 travaux A, B, C, D, E dont le temps d'exécution respectifs et leur arrivage respectifs sont données dans le tableau suivant :

Processus	**Temps d'exécution**	**Temps d'arrivée**	**Priorités**
A	3	0	2
B	6	1	6
C	4	4	3
D	2	6	5
E	1	7	4

En utilisant un algorithme **d'ordonnancement basé sur des priorités en mode non préemptif** donnez :

- le diagramme de Gantt,
- le temps de séjour de chaque processus,
- le temps moyen de séjour des processus,
- le temps d'attente de chaque processus,
- le temps moyen d'attente des processus

- **Le diagramme de Gantt des processus**

A	B	D	C	E

0 3 9 11 15 16

- **Temps de séjour de chaque processus**

Processus	Temps de séjour
A	3-0=3
B	9-1=8
C	15-4=11
D	11-6=5
E	16-7=8

- **Temps moyen de séjour des processus**

TMS $=\sum_{i=1}^{n} TSi/n$

TMS = (3+8+11+5+8)/5=35/5 = 7

- **Temps d'attente de chaque processus**

Pour obtenir le temps d'attente, on fait :
TAi = temps de séjour – temps exécution

Processus	Temps d'attente
A	3-3=0

B	8-6=2
C	11-4=8
D	5-2=4
E	8-1=2

- **Temps moyen d'attente des processus**

  TMA = $\sum_{i=1}^{n} TAi/n$

  TAM= (0+2+8+4+2)/5= 16/5 =3.2

En utilisant un algorithme **d'ordonnancement basé sur des priorités en mode préemptif (avec réquisition)** donnez :

- le diagramme de Gantt,
- le temps de séjour de chaque processus,
- le temps moyen de séjour des processus,
- le temps d'attente de chaque processus,
- le temps moyen d'attente des processus

- **Le diagramme de Gantt des processus**

A	B	D	E	C	A
0	1	7	9	10	14	16

- **Temps de séjour de chaque processus**

Processus	Temps de séjour
A	16-0=16
B	7-1=6
C	14-4=10
D	9-6=3
E	10-7=3

- **Temps moyen de séjour des processus**

TMS = $\sum_{i=1}^{n} TSi/n$

TSM= (16+6+10+3+3)/5= 38/5 = 7.6

- **Temps d'attente de chaque processus**

Pour obtenir le temps d'attente, on fait :
TAi = temps de séjour – temps exécution

Processus	Temps d'attente
A	16-3=0
B	6-6 = 0
C	10-4=6
D	3-2=1
E	3-1=2

- **Temps moyen d'attente des processus**

TMA = $\sum_{i=1}^{n} TAi/n$

TAM = (0+0+6+1+2)/5 = 9/5 = 1.8

2.8.4 Algorithme de tourniquet ou d'ordonnancement circulaire (round robin)

2.8.4.1 Principe

Dans cet algorithme les processus sont rangés dans une file d'attente des éligibles, le processeur est alloué successivement aux différents processus pour une tranche de temps fixe Q appelé Quantum.

Cet Ordonnancement est régit par les règles suivantes :

1. Un processus qui rentre dans l'état éligible est mis en queue de la file d'attente des prêts.
2. Si un processus élu se termine ou se bloque avant de consommer son quantum de temps, le processeur est immédiatement alloué au prochain processus se trouvant en tête de la file d'attente des prêts.
3. Si le processus élu continue de s'exécuter au bout de son quantum, dans ce cas le processus sera interrompu et mis en queue de la file d'attente des prêts et le processeur

est réquisitionné pour être réalloué au prochain processus en tête de cette même file d'attente.

❖ Caractéristiques de l'Ordonnanceur

- Avec réquisition
- Adapté aux systèmes temps partagé.
- La stratégie du tourniquet garantit que tous processus sont servis au bout d'un temps fini. Son avantage est d'éviter la famine. On dit qu'un processus est en ***famine*** lorsqu'il est prêt à être exécuté et se voit refuser l'accès à une ressource (ici le processeur) pendant un temps indéterminé
- L'efficacité de cet ordonnanceur dépend principalement de la valeur du quantum Q:
  - Le choix d'un Q assez petit augmente le nombre de commutation.
  - Le choix d'un Q assez grand augmente le temps de réponse du système

2.8.4.2 Application

Considérons 5 travaux A, B, C, D, E dont le temps d'exécution respectifs et leur arrivage respectifs sont données dans le tableau suivant :

Processus	**Temps d'exécution**	**Temps d'arrivée**	**Priorités**
A	3	0	2
B	6	1	6
C	4	4	3
D	2	6	5
E	1	7	4

En utilisant un algorithme **d'ordonnancement circulaire** donnez :

- le diagramme de Gantt,
- le temps de séjour de chaque processus,
- le temps moyen de séjour des processus,
- le temps d'attente de chaque processus,
- le temps moyen d'attente des processus

- **Le diagramme de Gantt des processus**

A	B	A	C	B	D	E	C	B
0	2	4	5	7	9	11	12	14	16

- **Temps de séjour de chaque processus**

Processus	Temps de séjour
A	5-0=5
B	16-1=15
C	14-4=10
D	11-6=5
E	12-7=8

- **Temps moyen de séjour des processus**

TMS = $\sum_{i=1}^{n} TSi/n$

TMS = (5+15+10+5+8)/5=43/5 = 8.6

- **Temps d'attente de chaque processus**

Pour obtenir le temps d'attente, on fait :
TAi = temps de séjour – temps exécution

Processus	Temps d'attente
A	5-3=2
B	15-6=9
C	10-4=6
D	5-2=4
E	8-1=2

- **Temps moyen d'attente des processus**

TMA = $\sum_{i=1}^{n} TAi/n$

TAM = (2+9+6+4+2)/5= 23/5 =3.2

2.9 Les threads

2.9.1 Définition

Les interfaces graphiques, les systèmes multi-processeurs ainsi que les systèmes répartis ont donné lieu à une révision de la conception usuelle des processus. Cette révision se fonde sur la notion de fils d'exécution (*thread of control*).

Un **thread** ou **fil (d'exécution)** ou **tâche** (terme et définition normalisés par ISO/CEI 2382-7:2000 ; autres appellations connues : **processus léger**, **fil d'instruction**, **processus allégé**, **exétron**, voire **unité d'exécution** ou **unité de traitement**) est similaire à un processus car tous deux représentent l'exécution d'un ensemble d'instructions du langage machine d'un processeur. Du point de vue de l'utilisateur, ces exécutions semblent se dérouler en parallèle. Toutefois, là où chaque processus possède sa propre mémoire virtuelle, les *threads* d'un même processus se partagent sa mémoire virtuelle. Par contre, tous les *threads* possèdent leur propre pile d'appel.

Un processus classique est composé d'un espace d'adressage et d'un seul fil de commande. L'idée est d'associer plusieurs fils d'exécution à un espace d'adressage et à un processus. Les fils d'exécution sont donc des processus dégradés (sans espace d'adressage propre) à l'intérieur d'un processus ou d'une application.

2.9.2 Pourquoi des files d'exécution ou threads

Certaines applications se dupliquent entièrement au cours du traitement. C'est notamment le cas pour des systèmes client-serveur, où le serveur exécute un `fork`, pour traiter chacun de ses clients. Cette duplication est souvent très coûteuse. Avec des fils d'exécution, on peut arriver au même résultat sans gaspillage d'espace, en ne créant qu'un fil de contrôle pour un nouveau client, et en conservant le même espace d'adressage, de code et de données. Les fils d'exécution sont, par ailleurs, très bien adaptés au parallélisme. Ils peuvent s'exécuter simultanément sur des machines multiprocesseurs.

2.9.3 De quoi sont constitués les fils d'exécution

Éléments d'un fil de commande	Éléments d'un processus ordinaire
Compteur de programme Pile Registres Fils de commande fils Statut	Espace d'adressage Variables globales Table des fichiers ouverts Processus fils Compteurs Sémaphores

Le contexte comparé d'un fil de commande et d'un processus.

Les éléments d'un processus sont communs à tous les fils d'exécution de ce processus.

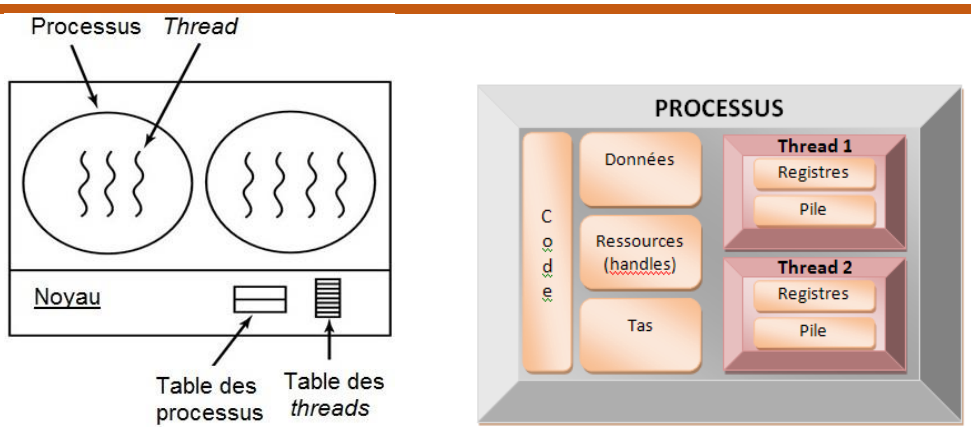


Figure 5 : Représentation des threads

2.9.4 Utilisation

Les *threads* sont classiquement utilisés avec l'interface graphique (*Graphical user interface*) d'un programme, pour des attentes asynchrones dans les télécommunications ou encore pour des programmes de calcul intensif (comme l'encodage d'une vidéo, les simulations mathématiques, etc.).

En effet, dans le cas d'une interface graphique, les interactions de l'utilisateur avec le processus, par l'intermédiaire des périphériques d'entrée, sont gérées par un *thread*, technique similaire à celle utilisée pour les attentes asynchrones, tandis que les calculs lourds (en termes de temps de calcul) sont gérés par un ou plusieurs autres *threads*. Cette technique de conception de logiciel est avantageuse dans ce cas, car l'utilisateur peut continuer d'interagir avec le programme même lorsque celui-ci est en train d'exécuter une tâche. Une application pratique se retrouve dans les traitements de texte où la correction orthographique est exécutée tout en permettant à l'utilisateur de continuer à entrer son texte. L'utilisation des *threads* permet donc de rendre l'utilisation d'une application plus fluide, car il n'y a plus de blocage durant les phases de traitements intenses.

Dans le cas d'un programme de calcul intensif, l'utilisation de plusieurs *threads* permet de paralléliser le traitement, ce qui, sur les machines multiprocesseur, permet de l'effectuer bien plus rapidement.

2.9.5 Threads et multitâche

Les *threads* se distinguent du multi-processus plus classique par le fait que deux processus sont typiquement totalement indépendants et isolés les uns des autres, et ne peuvent interagir qu'à travers une API fournie par le système, tel que IPC. D'un autre côté, les *threads* partagent une information sur l'état du processus, des zones de mémoires, ainsi que d'autres ressources : en fait, à part leur pile d'appel, des mécanismes comme le *Thread Local Storage* et quelques rares exceptions spécifiques à chaque implémentation, les *threads* partagent tout. Puisqu'il n'y a pas de changement de mémoire virtuelle, la commutation de contexte (*context switch*) entre deux

threads est moins coûteuse que la commutation de contexte entre deux processus. On peut y voir un avantage de la programmation utilisant des *threads* multiples.

2.9.6 Avantages et inconvénients

L'avantage essentiel des fils d'exécution (threads) est de permettre le parallélisme de traitement en même temps que les appels système bloquants. Ceci impose une coordination particulière des fils d'exécution d'un processus. Dans une utilisation optimale, chaque fil de commande dispose d'un processeur. A. Tanenbaum, dans *Modern Operating Systems*, distingue trois modèles d'organisations possibles : le modèle répartiteur/ouvrier (*dispatcher/worker*), l'équipe et le pipeline.

Dans certains cas, les programmes utilisant des *threads* sont plus rapides que des programmes architecturés plus classiquement, en particulier sur les machines comportant plusieurs processeurs. Hormis le problème du coût de la commutation de contexte, le principal surcoût dû à l'utilisation de processus multiples provient de la communication entre processus séparés. En effet, le partage de ressources entre *threads* permet une communication plus efficace entre les différents *threads* d'un processus qu'entre deux processus distincts. Là où deux processus séparés doivent utiliser un mécanisme fourni par le système pour communiquer, les *threads* partagent une partie de l'état du processus, notamment sa mémoire. Dans le cas de données en lecture seule, il n'y a même pas besoin du moindre mécanisme de synchronisation pour que les *threads* utilisent les mêmes données.

La programmation utilisant des *threads* est toutefois plus rigoureuse que la programmation séquentielle, et l'accès à certaines ressources partagées doit être restreint par le programme lui-même, pour éviter que l'état d'un processus ne devienne temporairement incohérent, tandis qu'un autre *thread* va avoir besoin de consulter cette portion de l'état du processus. Il est donc obligatoire de mettre en place des mécanismes de synchronisation (à l'aide de sémaphores, par exemple), tout en conservant à l'esprit que l'utilisation de la synchronisation peut aboutir à des situations d'interblocage si elle est mal utilisée.

La complexité des programmes utilisant des *threads* est aussi nettement plus grande que celle des programmes déférant **séquentiellement** le travail à faire à plusieurs processus plus simples (la complexité est similaire dans le cas de plusieurs processus travaillant en parallèle). Cette complexité accrue, lorsqu'elle est mal gérée lors de la phase de conception ou de mise en œuvre d'un programme, peut conduire à de multiples problèmes tels que :

- Interblocages apparemment aléatoires :
  - Si un algorithme est mal conçu et permet un interblocage, le changement de vitesse du processeur, par l'utilisation incomplète d'un quantum de temps alloué ou au contraire de la nécessité d'utiliser un quantum supplémentaire, peut provoquer la situation d'interblocage,
  - De même, des périphériques plus ou moins rapides que ceux de la machine ayant servi à développer/tester le programme induisent des synchronisations temporellement différentes, et donc peuvent provoquer un interblocage non-détecté auparavant,

- Enfin, le changement de charge processeur de la machine (un programme lourd tournant en parallèle, par exemple) peut là aussi déclencher plus facilement des interblocages qui n'avaient pas été vus auparavant, avec l'illusion d'un phénomène aléatoire ;

- Complexification inutile de certains algorithmes, qui ne bénéficient pas de la parallélisation du code mais qui sont pénalisés par des synchronisations d'accès aux ressources partagées :
  - Sur-parallélisation du code alors qu'un traitement séquentiel serait plus adapté, la perte de temps liée à la commutation de contexte n'étant pas compensée par la parallélisation,
  - Introduction de variables globales, forçant l'utilisation de synchronisation à outrance et donc des commutations de contexte inutiles ;
- Sur-utilisation des mécanismes de synchronisation inadaptés là où ils ne sont pas nécessaires, induisant un surcoût de temps processeur inutile :
  - La mise en œuvre d'un pipeline FIFO entre deux *threads* seulement peut être effectuée facilement sans aucun mutex,
  - Utilisation d'une section critique au lieu d'un mutex lecteurs/rédacteur, par exemple lorsque beaucoup de *threads* lisent une donnée tandis que peu écrivent dedans ou qu'une donnée est plus souvent lue qu'écrite.

2.9.7 Support des threads

2.9.7.1 Système d'exploitation

Les systèmes d'exploitation mettent généralement en œuvre les *threads*, souvent appelés *threads système* ou *threads natifs* (par opposition aux *threads* liés à un langage de programmation donné). Ils sont utilisés au travers d'une API propre au système d'exploitation, par exemple Windows API ou les Threads POSIX. En général, cette API n'est pas orientée objet, et peut être relativement complexe à bien mettre en œuvre car composée uniquement de fonctions primitives, ce qui demande souvent à avoir quelques notions sur le fonctionnement de l'ordonnanceur.

L'avantage des *threads* natifs est qu'ils permettent des performances maximales, car leur API permet de minimiser le temps passé dans le noyau et d'éliminer les couches logicielles inutiles. Leur principal inconvénient est, de par la nature primitive de l'API, une plus grande complexité de mise en œuvre.

2.9.7.2 Langages de programmation

Certains langages de programmation, tels que Smalltalk et certaines implémentations de Java, intègrent un support pour les *threads* implémentés dans l'espace utilisateur (*green threads (en)*), indépendamment des capacités du système d'exploitation hôte.

La plupart des langages (Java sur la plupart des systèmes d'exploitation, C# .NET, C++, Ruby…) utilisent des extensions du langage ou des bibliothèques pour utiliser directement les services de multithreading du système d'exploitation, mais de façon portable. Enfin, des

langages comme Haskell utilisent un système hybride à mi-chemin entre les deux approches. À noter que, pour des raisons de performances en fonction des besoins, la plupart des langages permettent d'utiliser au choix des *threads natifs* ou des *green threads* (notamment via l'utilisation de fibres). D'autres langages, comme Ada (langage), implémentent également un multitâche indépendant du système d'exploitation sans pour autant utiliser réellement le concept de *thread*.

Le C++, depuis la nouvelle norme du C++ nommée C++11, possède aussi une bibliothèque de gestion des *threads* (issue de Boost) : le modèle de classe est `std::thread`. Celui-ci est simple d'utilisation, et permet de bien créer et exécuter ses *threads*. Auparavant, chaque framework devait implémenter sa propre surcouche de gestion des *threads*, en général câblée directement sur les *threads* natifs du système.

2.9.7.3 Paradigmes de développement

- **Réentrance**

En programmation parallèle, le principe de base est d'assurer la réentrance des entités logicielles utilisées par les *threads*, soit par conception (fonctions pures), soit par synchronisation (notamment par l'utilisation d'un mutex encadrant l'appel à la fonction).

- **Programmation procédurale**

En programmation procédurale, on utilise le plus souvent directement les *threads* natifs du système d'exploitation. Leur utilisation est alors directement dépendante de l'API du système, avec ses avantages et inconvénients. Notamment, les fonctions utilisées par les *threads* doivent être réentrantes ou protégées par mutex.

- **Programmation orientée objet**

En programmation orientée objet, l'utilisation des *threads* se fait en général par héritage depuis une classe mère (ou modèle de classe) générique, possédant une méthode virtuelle pure qui contiendra le code à exécuter en parallèle. Il faut et il suffit alors d'écrire la classe dérivée implémentant ce que l'on veut paralléliser, de l'instancier et d'appeler une méthode particulière (souvent nommée *Run* ou équivalent) pour démarrer le *thread*. Des méthodes d'arrêt ou d'attente de fin de tâche sont également présentes, simplifiant fortement la création de *threads* simples. Toutefois, des opérations plus complexes (barrière sur un nombre important de *threads*, réglages précis de priorité, etc.) peuvent être moins faciles qu'avec l'utilisation des *threads* natifs.

On parle de *classe réentrante* lorsque des instances distinctes d'une telle classe peuvent être chacune utilisées par des *threads* sans effet de bord, ce qui signifie en général une absence d'élément global ou statique dans l'implémentation de la classe. On parle aussi de *classe thread-safe* lorsque plusieurs *threads* peuvent utiliser une seule et même instance de cette classe sans engendrer de problèmes de concurrence. Une classe *thread-safe* est forcément réentrante, mais la réciproque est fausse.

- **Programmation fonctionnelle**

Par nature, comme tout élément en programmation fonctionnelle est réentrant - et souvent *thread-safe* - à quelques rares exceptions près, la mise en œuvre des *threads* est fortement simplifiée par ce paradigme. En général, la parallélisation du code ne dépend que des contrats et des séquences d'appel : $f(g(x))$ requiert bien entendu de calculer $y=g(x)$ avant $f(y)$, ce qui empêche de paralléliser les deux fonctions. Mais ces contraintes ne sont pas spécifiques à la programmation fonctionnelle, elles sont inhérentes à l'algorithme implémenté.

2.9.8 Patrons de conception (*Design Patterns*) classiques avec les *threads*

Beaucoup de *patterns* peuvent bénéficier des *threads*, comme *Decorator*, *Factory Method*, *Command* ou encore *Proxy*. De manière générale, toute passation d'information à une entité logicielle d'un *pattern* peut faire l'objet de l'utilisation de *threads*.

2.9.8.1 Le modèle maître-esclave et client-serveur

On L'appelle aussi modèle **répartiteur/ouvrier**. Dans ce modèle, les demandes de requêtes à un serveur arrivent dans une boîte aux lettres. Le fil de commande répartiteur a pour fonction de lire ces requêtes et de réveiller un fil de commande ouvrier. Le fil de commande ouvrier gère la requête et partage avec les autres fils d'exécution, la mémoire cache du serveur.

Cette architecture permet un gain important car elle évite au serveur de se bloquer lors de chaque entrée-sortie.

- Un *thread* « maître » centralise le travail à faire et le distribue aux *threads* « esclaves », collectant ensuite leurs résultats.
- Un pool de *threads* (le *pool* étant statique ou dynamique) permet d'interroger plusieurs serveurs sans bloquer le client.
- Le Pair à pair (*Peer-to-peer*) est un modèle similaire au maître-esclave, mais où le *thread* maître n'est pas figé (chaque *thread* est le maître de tous les autres, tout en étant esclave lui-même).

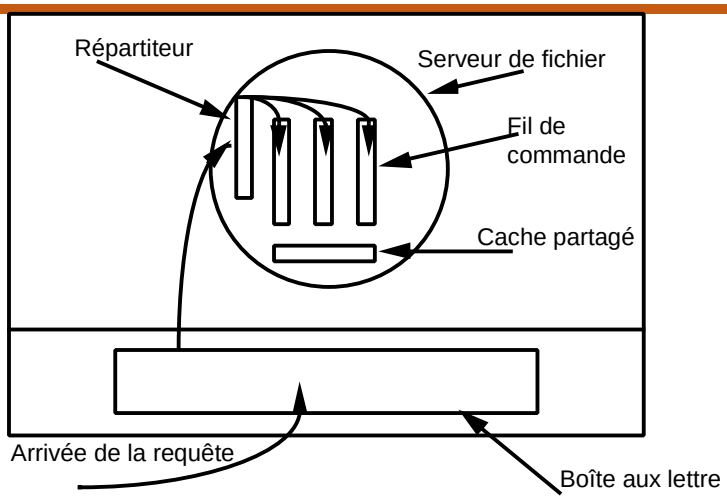


***Figure 6** Le répartiteur*

2.9.8.2 L'équipe

Dans le modèle de l'équipe, chaque fil de commande traite une requête. Il n'y a pas de répartiteur. À la place, on met en implante une file de travaux en attente.

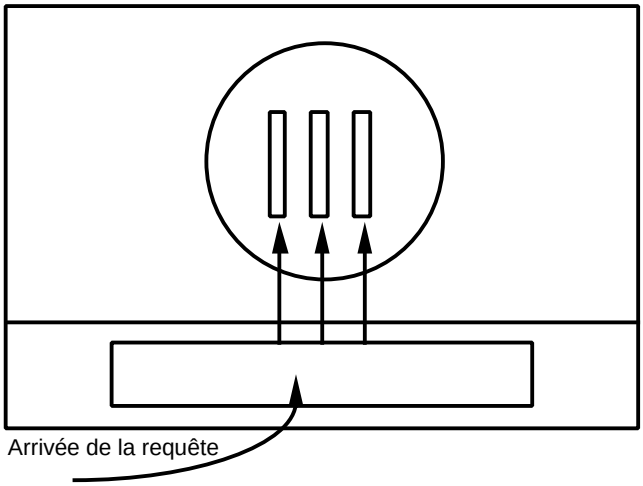


Figure 7 : L'équipe

2.9.8.3 Le pipeline

Dans le modèle du pipeline, chaque fil de commande traite une partie de la requête. Plusieurs architectures matérielles implantent ce type d'organisation. Ce modèle n'est cependant pas approprié pour toutes les applications.

- La tâche est découpée en diverses étapes successives et/ou opérations unitaires. Chaque *thread* réalise une des étapes et passe son résultat au suivant.
- Plusieurs pipelines peuvent coexister afin d'accélérer encore le traitement.

- Des traitements comme la compilation, le streaming, le traitement multimédia (décodage de vidéos par exemple) bénéficient fortement d'une implémentation utilisant des *threads*.

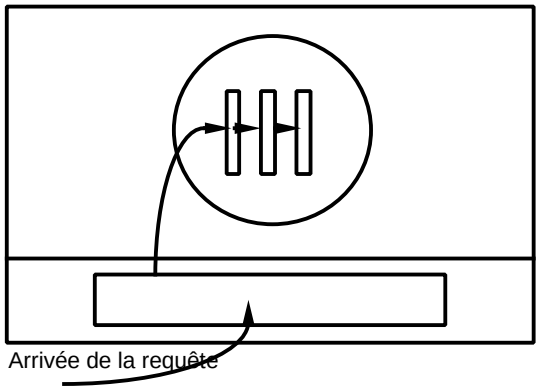


Figure 8 :Le pipeline

❖ **Remarques**

Il ne faut pas confondre la technologie *Hyper-Threading* de certains processeurs Intel avec les *threads*. Cette technologie permet en effet aussi bien l'exécution simultanée de processus distincts que de *threads*. Toute machine comportant des processeurs multiples (SMP) ou des processeurs intégrant l'Hyper-Threading permet aussi bien l'exécution plus rapide de programmes utilisant des *threads* que de multiples processus.

2.10 Interblocage

2.10.1 Introduction

Dans un système multiprocessus, l'ordonnanceur alloue le processeur à chaque processus selon un algorithme d'ordonnancement : la politique choisie conditionne l'ordre d'exécution des processus et très souvent, les exécutions des processus s'entrelacent les unes avec les autres. Chaque processus dispose d'un espace d'adressage propre et indépendant, protégé par rapport aux autres processus. Malgré tout, les processus peuvent avoir besoin de communiquer entre eux pour échanger des données par exemple : ils ne sont donc pas totalement indépendants et effectuent des accès concurrents aux ressources logicielles ou matérielles.

Un **interblocage** (ou **étreinte fatale**, ***deadlock*** en anglais) est un phénomène qui peut survenir en programmation concurrente. L'interblocage se produit lorsque deux processus concurrents s'attendent mutuellement. Les processus bloqués dans cet état le sont définitivement, il s'agit

donc d'une situation catastrophique. Les mécanismes conduisant aux phénomènes d'interblocage ont été étudiés principalement par Edward Coffman, Jr.

2.10.2 Définition ressource

Une ressource désigne toute entité dont a besoin un processus pour s'exécuter. La ressource peut être matérielle comme le processeur ou en périphérique ou elle peut être logicielle comme une variable. Une ressource est caractérisée :

- Par un état : elle est libre ou occupée
- Par son nombre de points d'accès, c'est-à-dire le nombre de processus pouvant l'utiliser en même temps.

Dans des conditions normales de fonctionnement, un processus ne peut utiliser une ressource qu'en suivant la séquence suivante :

Requête → Utilisation → Libération

- *La requête* : le processus fait une demande pour utiliser la ressource. Si cette demande ne peut pas être satisfaite immédiatement, parce que la ressource n'est pas disponible, le processus demandeur se met en état attente jusqu'à ce que la ressource devienne libre

- *Utilisation* : Le processus peut exploiter la ressource

- *Libération* : Le processus libère la ressource qui devient disponible pour les autres processus éventuellement en attente.

2.10.3 Définitions des situations d'interblocage, de famine et de coalition

Un ensemble de n processus est dit en situation d'interblocage lorsque l'ensemble de ces n processus attend chacun une ressource déjà possédée par un autre processus de l'ensemble. Autrement dit, Un ensemble de processus est dans une situation d'interblocage si chaque processus de l'ensemble attend un événement qui ne peut être produit que par un autre processus de l'ensemble.

Dans une telle situation aucun processus ne peut poursuivre son exécution. L'attente des processus est infinie.

❖ **Exemple1**

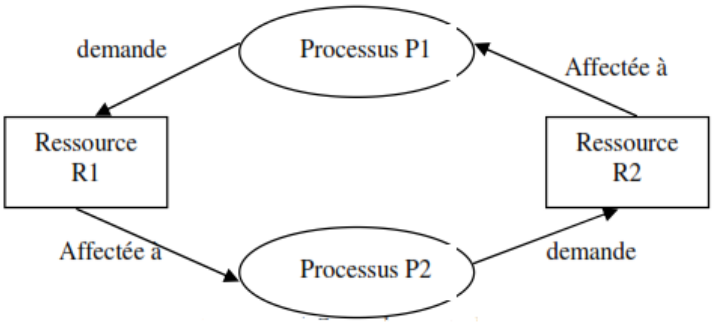
Un exemple concret d'interblocage peut se produire lorsque deux processus légers (*thread* en anglais) essayent d'acquérir deux mutex dans un ordre différent. Par exemple avec deux mutex (*M1* et *M2*), deux processus légers (*P1* et *P2*) et la séquence d'opération suivante :

1. *P1* acquiert *M1*.
2. *P2* acquiert *M2*.
3. *P1* attend pour acquérir *M2* (qui est détenu par *P2*).
4. *P2* attend pour acquérir *M1* (qui est détenu par *P1*).

Dans cette situation, les deux processus légers sont définitivement bloqués.

❖ **Exemple2**

Un système possède une instance unique de chacun des deux types de ressources R1 et R2. Un processus P1 détient l'instance de la ressource R1 et un autre processus P2 détient l'instance de la ressource R2. Pour suivre son exécution, P1 a besoin de l'instance de la ressource R2, et inversement P2 à besoin de l'instance de la ressource R1. Une telle situation est une situation d'interblocage.



❖ **Définition : Coalition et famine**

On parle de coalition de n processus contre p autres processus lorsqu' un ensemble de n processus monopolisent des ressources au détriment des p autres processus. On dit également que les p processus qui ne peuvent pas s'exécuter faute de ressources sont en situation de famine.

On dit qu'un processus est dans une situation de famine (starvation) s'il attend indéfiniment une ressource (qui est éventuellement occupée par d'autre processus).

Notons que l'interblocage implique nécessairement une famine, mais le contraire n'est pas toujours vrai.

2.10.4 Conditions nécessaires à l'obtention d'un interblocage

Les quatre conditions listées ci-dessus doivent être simultanément vérifiées pour qu'un interblocage puisse se produire :

- Exclusion mutuelle : une ressource au moins doit se trouver dans un mode non partageable
- Occupation et attente : un processus au moins occupant une ressource attend d'acquérir des ressources supplémentaires détenues par d'autres processus. Les processus demandent les ressources au fur et à mesure de leur exécution.
- Pas de réquisition : les ressources sont libérées sur seule volonté des processus les détenant
- Attente circulaire : il existe un cycle d'attente entre au moins deux processus. Les processus impliqués dans ce cycle sont en interblocage. Autrement dit, il doit avoir un ensemble de processus {P0, P1, …,Pn} en attente tels que : P0 est en attente pour une ressource détenue par P1, P1 est en attente d'une ressource détenue par P2, …, Pn–1 est en attente d'une ressource détenue par Pn, et Pn est en attente d'une ressource détenue par P0 (P0 attend P1 attend P2 … Pn attend P0).

La figure ci-dessous donne un exemple d'attente circulaire entre deux processus P1 et P2.

- Les deux processus P1 et P2 utilisent les trois mêmes ressources : un lecteur de bandes magnétiques, un disque dur et une imprimante.
- Le processus P1 commence par demander la bande magnétique puis l'imprimante et enfin le disque avant d'effectuer son traitement.
- Le processus P2 commence par demander le disque puis l'imprimante et enfin la bande magnétique avant de commencer son traitement.
- Le processus P1 s'est exécuté et a obtenu la bande magnétique ainsi que l'imprimante.
- Le processus P2 lui a obtenu le disque et il demande maintenant à obtenir l'imprimante. L'imprimante a déjà été allouée au processus P1, donc le processus P2 est bloqué.
- Le processus P1 ne peut pas poursuivre son exécution car il est en attente du disque qui a déjà été alloué au processus P2. On a donc une attente circulaire entre P1 et P2 : en effet le processus P2 attend l'imprimante détenue par le processus P1 tandis que le processus P1 attend le disque détenu par le processus P2.

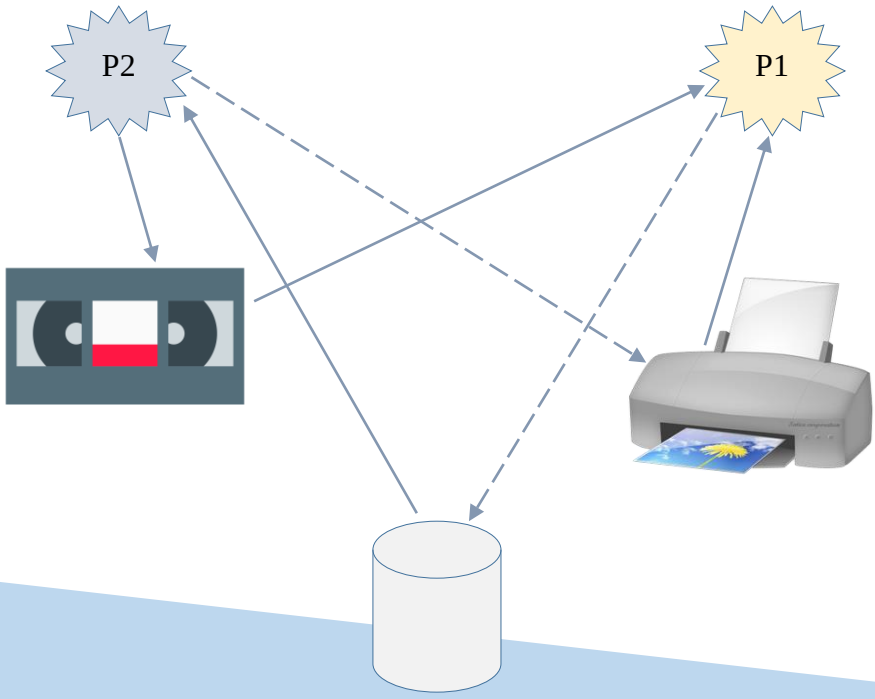


Figure 9 : Exemple d'attente circulaire

2.10.5 Graphe d'allocation de ressources

Les inter-blocages peuvent être représentés (modélisés) en utilisant **un graphe orienté** appelé graphe d'allocation de ressources.

On peut décrire l'état d'allocation des ressources d'un système en utilisant un graphe. Ce graphe est composé de N nœuds et de A arcs.

L'ensemble des nœuds est partitionné en deux types :

- P = {P1, P2, …, Pm} : l'ensemble de tous les processus
- R = {R1, R2, …, Rn} l'ensemble de tous les types de ressources du système

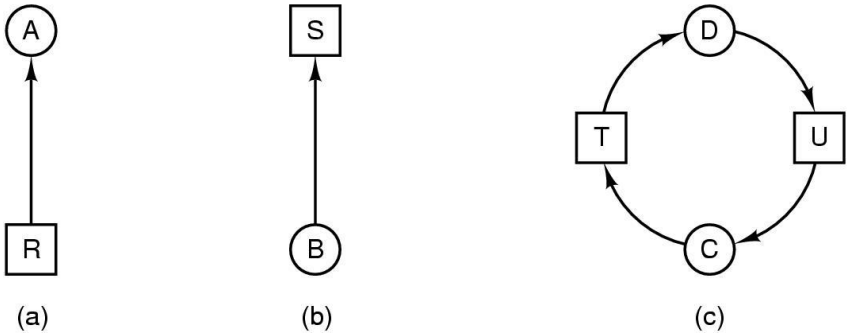
- Un arc allant du processus Pi vers un type de ressource Rj est noté Pi→Rj ; il signifie que le processus Pi a demandé une instance du type de ressource Rj.

- Un arc du type de ressourcé Rj vers un processus Pi est noté Rj→Pi ; il signifie qu'une instance du type de ressource Rj a été alloué au processus Pi.

- Un arc Pi→Rj est appelé arc de requête. Un arc Rj→Pi est appelé arc d'affectation.

- Graphiquement, on représente chaque processus Pi par un cercle et chaque type de ressource Rj comme un rectangle. Puisque chaque type de ressource Rj peut posséder plus d'une instance, on représente chaque instance comme un point dans le rectangle.

- Un arc de requête désigne seulement le rectangle Rj, tandis que l'arc d'affectation doit aussi désigner un des points dans le rectangle.



(a) (b) (c)

(a) Le processus A détient la ressource R.

(b) Le processus B demande la ressource R.

(c) Le processus C attend la ressource T, qui est détenue par le processus D. Le processus D attend la ressource U, qui est détenue par le processus C → Interblocage.

- Quand un processus Pi demande une instance du type de ressource Rj, un arc de requête est inséré dans le graphe d'allocation des ressources.
- Quand cette requête peut être satisfaite, l'arc de requête est instantanément transformé en un arc d'affectation. Quand plus tard, le processus libère la ressource l'arc d'affection est supprimée.

Il peut être démontré que :

- Si le graphe d'allocation ne contient pas de cycles alors il n'y a pas d'inter-blocage
- Si le graphe contient un cycle alors :
  - Si il y a une seule instance par type de ressources, alors inter-blocage
  - S'il y a plusieurs instances par type de ressources, alors il y a une possibilité d'inter-blocage

Exemple : L'état d'allocation d'un système est décrit par les ensembles suivants :

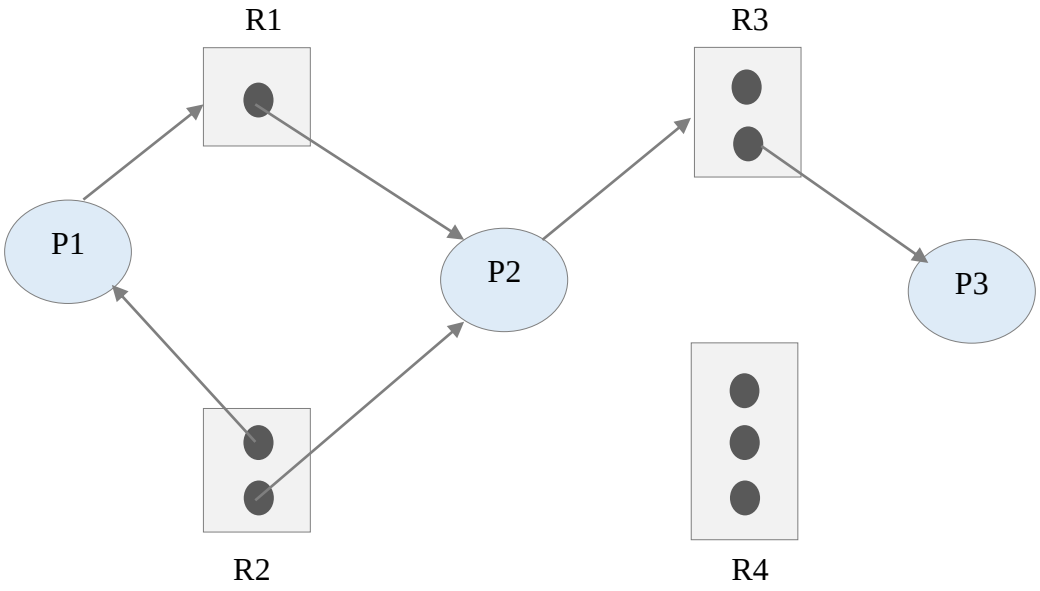
- Ensemble des processus P = {P1, P2, P3}
- Ensemble des ressources R = {R1, R2, R3, R4}
- Ensemble des arcs A = {P1→R1, P2→R3, R1→P2, R2→P2, R2→P1, R3→P3}

Le nombre d'instances par ressources est donné par ce tableau :

Type de ressources	Nombre d'instances
R1	1
R2	2

R3	2
R4	3

Voici le graphe d'allocation des ressources associé à ce système :



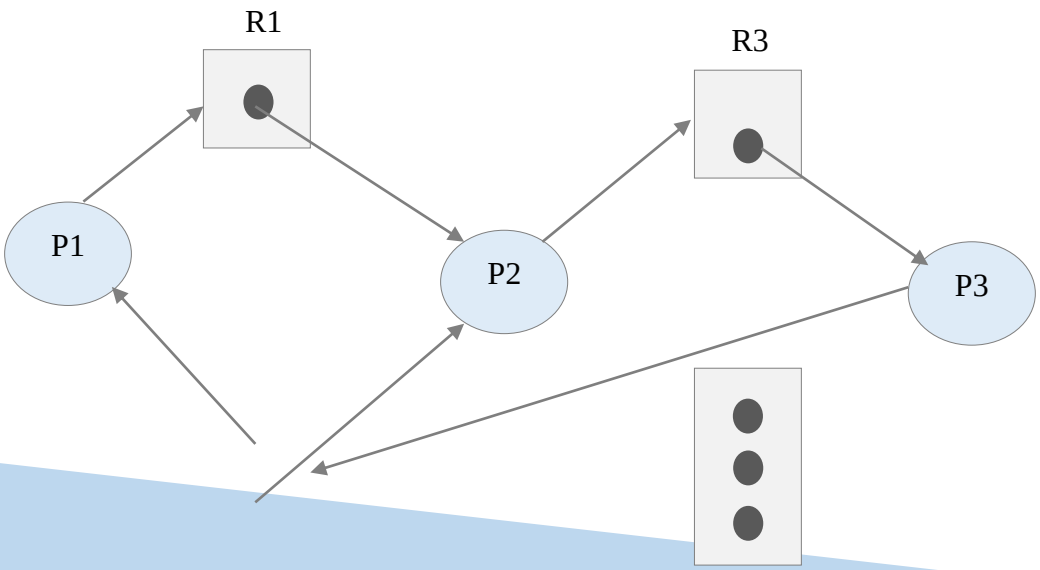
Graphe d'allocation des ressources.

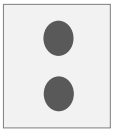
On peut faire la lecture suivante sur ce graphe :

- Le processus P1 détient une instance de la ressource R2 et demande une instance de la ressource R1
- Le processus P2 détient une instance de R1 et attend une instance de R3
- Le processus P3 détient une instance de la ressource R3

Si le graphe d'allocation contient un *circuit*, alors il *peut* exister une situation d'interblocage.

Considérons les exemples suivants :

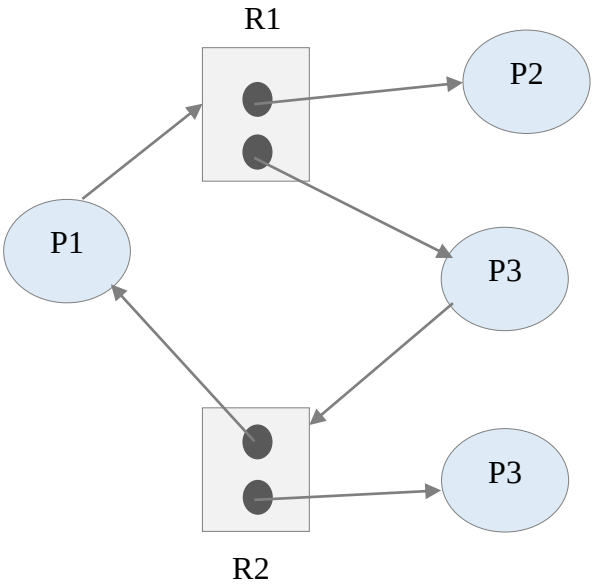




R2

R4

Graphe d'allocation des ressources avec circuit et interblocage



Graphe d'allocation des ressources avec circuit sans interblocage

2.10.6 Les différentes méthodes de traitement des interblocages

Il y a 4 méthodes de traitement des situations d'interblocage : les politiques de détection-guérison, les politiques de prévention, les politiques d'évitements et la politique de "l'autruche".

2.10.6.1 La politique de l'autruche

Elle est très simple, elle consiste à nier l'existence des interblocages et donc de ne rien prévoir pour les traiter. Simplement la machine est redémarrée lorsque trop de processus sont en interblocage. Les trois premières stratégies évoquées (prévention, évitement détection/guérison) sont des politiques qui coûtent excessivement chères à mettre en œuvre. Aussi, comme la fréquence des interblocages dans un système est relativement faible, la politique de l'autruche qui paraît dans un premier abord très "curieuse" se justifie souvent.

2.10.6.2 Les politiques de prévention

Dans les politiques de prévention, on ajoute des contraintes sur l'allocation des ressources afin de faire en sorte qu'au moins une des 4 conditions nécessaires à l'interblocage ne sera jamais vérifiée. Les deux seules conditions nécessaires sur lesquelles il est possible d'agir véritablement sont la condition d'occupation et d'attente ainsi que la condition d'attente circulaire.

- ***Exclusion mutuelle*** :

Un processus ne doit jamais attendre une ressource partageable (exemple : il faut autoriser autant de processus que possible pour la lecture d'un fichier). Cependant, il n'est pas possible de prévenir les interblocages en niant la condition de l'exclusion mutuelle : certaines ressources sont non partageables (exemple : il n'est pas possible de partager un fichier entre plusieurs rédacteurs).

- ***Occupation et attente*** :

Pour s'assurer que la condition d'occupation et d'attente ne se produit jamais dans le système, on doit garantir qu'à chaque fois qu'un processus demande une ressource, il n'en détient aucune autre. Autrement dit, un processus ne doit demander des ressources supplémentaires qu'après avoir libéré les ressources qu'il occupe déjà.

Par exemple, imaginons un processus qui copie des données d'une unité de bandes vers un fichier du disque, trie le fichier puis imprime le résultat. Si l'on doit demander toutes les ressources au début du processus, celui-ci doit dès le départ requérir l'unité de bande, le fichier du disque et l'imprimante. Ainsi, il gardera l'imprimante pendant toute l'exécution, même s'il n'en a besoin qu'à la fin. L'autre façon de faire, serait d'affecter au processus uniquement l'unité de bande et le fichier sur disque, au début de son exécution. Il copie de l'unité de bande vers le disque et les libère ensuite tous les deux. Le processus doit à nouveau demander le fichier sur disque et l'imprimante.

Après avoir copié le fichier sur l'imprimante, il libère ces deux ressources et se termine.

- ***Pas de réquisition*** :

Pour garantir que cette condition ne soit pas vérifiée, on peut utiliser le protocole suivant : Si un processus détenant certaines ressources en demande une autre qui ne peut pas lui être

immédiatement allouée, toutes les ressources actuellement alloues à ce processus sont réquisitionnées. C'est à dire que ses ressources sont implicitement libérées.

Les ressources réquisitionnées sont ajoutées à la liste des ressources pour lesquelles le processus attend. Le processus démarrera seulement quand il pourra regagner ses anciennes ressources, ainsi que les nouvelles qu'il requiert.

Ce protocole présente malheureusement au moins deux inconvénients :

- Lenteur dans l'utilisation des ressources puisqu'on peut allouer plusieurs ressources et ne pas les utiliser pendant longtemps.
- La famine est possible puisqu'un processus peut être retardé indéfiniment parce que l'une des ressources qu'il demande est occupée.

✦ ***Attente circulaire*** **:**

On peut garantir que la condition de l'attente circulaire ne se vérifie jamais, en imposant un ordre total sur les types de ressources et on forçant chaque processus à demander les ressources dans un ordre croissant d'énumération. Par exemple l'unité de bande doit toujours être demandée avant le disque et le disque doit lui-même être toujours demandé avant l'imprimante.

Ainsi, on numérote les types de ressources du système, comme suit :

- O(Unité de bandes)=1

- O(Unité de disque)=2

- O(Imprimante)=3

- O(Lecteur CD-ROM)=4

- O(Lecteur Disquette)=5

- … etc.

Pour prévenir l'interblocage, on peut imaginer le protocole suivant : Un processus ne peut acquérir une ressource d'ordre i que s'il a déjà obtenu toutes les ressources nécessaires d'ordre j (j<i). Par exemple, un processus désirant utiliser l'unité de bandes et l'imprimante, doit d'abord demander l'unité de bande ensuite l'imprimante. On peut démontrer que de cette façon, il n'y a aucun risque d'interblocage.

Cependant cet algorithme est inefficace, il peut conduire à une monopolisation inutile d'une ressource (un processus peut détenir une ressource alors qu'il n'en a pas besoin).

2.10.6.3 Les politiques d'évitement

2.10.6.3.1 Fonctionnement

Les interblocages peuvent être évités si certaines informations sont connues à l'avance lors des allocations de ressources. La troisième catégorie de solutions est celle des politiques d'évitement : ici, à chaque demande d'allocation de ressource faite par un processus, le système déroule un algorithme appelé algorithme de sureté qui regarde si cette allocation peut mener le système en interblocage. Le système ne répond favorablement qu'aux requêtes qui mènent à des états « sûrs » ou « sains ».

Un état est dit sûr ou sain s'il existe une suite d'états ultérieurs qui permet à tous les processus d'obtenir toutes leurs ressources et de se terminer. De façon générale :

> Un système est dans un état sain s'il existe une séquence saine. Une séquence de processus <P1, P2, .., PN> est une séquence saine pour l'état d'allocation courant si, pour chaque Pi, les requêtes de ressources de Pi peuvent être satisfaites par les ressources couramment disponibles, plus les ressources détenues par tous les Pj, avec j<i.

En effet, dans cette situation, si les ressources demandées par Pi ne sont pas immédiatement disponibles, Pi peut attendre jusqu'à la terminaison de tous les Pj. Une fois qu'ils ont fini, Pi peut obtenir les ressources nécessaires, achever sa tâche et rendre les ressources allouées. Quand Pi termine, Pi+1 peut obtenir les ressources manquantes, et ainsi de suite. Si une telle séquence n'existe pas, on dit que le système est dans état malsain, c'est à dire qu'il y a un risque d'apparition d'interblocage.

Pour être capable de décider si l'état suivant sera sûr ou non sûr, le système a besoin de connaître à tout moment le nombre et le type de ressources existantes, disponibles et demandées.

Exemple :

Un système possède 12 unités de bandes magnétiques et 3 processus P0, P1 et P2. On suppose que les besoins des trois processus en ressources sont: 10 pour P0, 4 pour P1, 9 pour P2.

D'autre part, on suppose qu'à l'instant t0 l'état d'allocation des ressources par les processus est le suivant : 5 pour P0, 2 pour P1 et 2 pour P2. Le nombre d'unités de bandes libres est donc égal à 3.

Processus	Besoins maximaux	Besoins actuellement satisfaits	Besoins restant à satisfaire
P0	10	5	5
P1	4	2	2
P2	9	2	7

Nombre de ressources (unités de bandes) disponibles : 03.

A l'instant t0, le système est dans un état sain. En effet, la séquence <P1, P0, P2> est saine, puisque :

- On peut allouer immédiatement au processus P1 toutes ses ressources manquantes et à la fin de son exécution il rend toutes les ressources qu'il détient. Dans ce cas le système disposerait de 5 unités de bandes (les 2 que disposait P1 avant + les 3 ressources libres)
- Le processus P0 peut donc obtenir toutes ses ressources manquantes et les rendre ensuite. Le système disposerait alors de 10 unités de bandes
- Enfin, le processus P2 peut obtenir ses unités de bandes et les rendre, le système disposerait donc de 12 unités de bandes.

Il est possible de passer d'un état sain à un état malsain. Supposons qu'à l'instant t1 le processus P2 demande et qu'on lui accorde 1 unité de bandes de plus. L'état du système serait alors le suivant :

Processus	Besoins maximaux	Besoins actuellement satisfaits	Besoins restant à satisfaire
P0	10	5	5
P1	4	2	2
P2	9	3	6

Nombre de ressources (unités de bandes) disponibles : 02.

Le système n'est plus dans un état sain. En effet, on ne peut allouer toutes ses unités de bandes qu'au processus P1. Quand il les rendra, le système ne disposera que de 04 unités de bandes. Comme le processus P0 peut demander 5 unités et le processus P2 7 unités, le système n'aura pas suffisamment de ressources pour les satisfaire. Ou pourra alors avoir un cas d'interblocage, puisque P0 et P2 seront retardés indéfiniment. Pour éviter cette situation, il ne fallait pas accorder à P1 la ressource qu'il a demandée ; il fallait le faire attendre.

2.10.6.3.2 Algorithme du banquier

L' **algorithme du banquier** mis au point par Edsger Dijkstra en 1965 est un algorithme pour éviter les problèmes interblocages et gérer l'allocation des ressources.

Le nom de l'algorithme a été choisi parce que cet algorithme pourrait s'appliquer dans un système bancaire pour s'assurer que la banque ne prête jamais son argent disponible de telle sorte qu'elle ne puisse plus satisfaire tous ses clients.

Quand un nouveau processus entre dans le système, il doit déclarer le nombre maximal d'instances de chaque type de ressources dont il aura besoin. Ce nombre ne doit pas excéder le nombre total de ressources du système. Au cours de son exécution, quand un processus demande un ensemble de ressources, l'algorithme vérifie si cela gardera toujours le système dans un état sain. Dans l'affirmative la demande est accordée, dans la négative la demande est retardée.

Soient *m* le nombre de types de ressources du système, et *n* le nombre de processus. Pour fonctionner, l'algorithme maintient plusieurs structures de données :

- **Available** : C'est un Vecteur (matrice ligne) de longueur m indiquant le nombre de ressources disponibles de chaque type. Ainsi, si Available[j]=k, cela veut dire que le type de ressources Rj possède k instances disponibles.

- **Max** : C'est une matrice nxm définissant la demande maximale de chaque processus. Ainsi, Si Max[i, j]=k, cela veut dire que le processus Pi peut demander au plus k instances du type de ressources Rj.

- **Allocation** : C'est une matrice nxm définissant le nombre de ressources de chaque type de ressources actuellement alloué à chaque processus. Ainsi si Allocation[i, j]=k, cela veut dire que l'on a alloué au processus Pi k instances du type de ressources Rj.

- **Need** : C'est une matrice nxm indiquant les ressources restant à satisfaire à chaque processus. Ainsi, si Need[i, j]=k , cela veut dire que le processus Pi peut avoir besoin de k instances au plus du type de ressources Rj pour achever sa tâche.

- **Request** : C'est une matrice nxm indiquant les ressources supplémentaires que les processus viennent de demander. Ainsi, si Request[i, j]=k, cela veut dire que le processus Pi vient de demander k instances supplémentaires du type de ressources Rj.

De ce qui précède, on peut remarquer que :

1. Ces structures de données peuvent varier dans le temps, en taille et en valeur.

2. La matrice *Need* peut être calculée à partir des matrices *Max* et Allocation :

Notations : Pour des raisons pratiques, on utilise les notations suivantes :

- $Allocation_i$: Vecteur désignant les ressources actuellement allouées au processus Pi.

- $Need_i$: Vecteur désignant les ressources supplémentaires que le processus Pi peut encore demander.

- $Request_i$: Vecteur désignant les ressources supplémentaires que le processus Pi vient de demander.

- Pour deux vecteurs X et Y : X<=Y si X[i] <=Y[i] pour chaque i allant de 1 à n.

L'algorithme du Banquier peut être scindé en deux parties complémentaires : Un algorithme de requête de ressources et un algorithme de vérification si l'état du système est sain.

Nous donnons maintenant le code de chacune des deux parties.

Algorithme de requête de ressources
Début

/* Cet algorithme est appelé à chaque fois qu'un processus fait une demande de ressources

Etape 1 : Si $Request_i$ <=$Need_i$ Alors
Aller à l'étape 2
Sinon
erreur : le processus a excédé ses besoins maximaux
Fin Si

Etape 2 : Si $Request_i$ <= Available Alors
Aller à l'étape 3
Sinon
Attendre : les ressources ne sont pas disponibles.
Fin Si

Etape 3 : Sauvegarder l'état du système (les matrices Available, Allocation et Need).
Allouer les ressources demandées par le processus Pi en modifiant l'état du système de la manière suivante :

Available = Available - $Request_i$
$Allocation_i$ = $Allocation_i$+ $Request_i$
$Need_i$ = $Need_i$ – $Request_i$

Si **Verification_Etat_Sain**=Vrai Alors
L'allocation est validée
Sinon
L'allocation est annulée ; Restaurer l'ancien Etat du système
Fin Si

Fin

Cet algorithme peut être résumé comme suit :

- Si un processus fait une demande de ressources :

1. Vérifier que la demande n'excède pas ses besoins ($Request_i$ <=$Need_i$)
2. Si $Request_i$ <=$Need_i$ alors vérifier que la demande ne dépasse pas les ressources disponibles ($Request_i$ <= Available)
3. Si $Request_i$ <= Available) :
   - On retire aux ressources disponibles celles que vient de demander le processus (Available = Available - $Request_i$)
   - On ajoute les ressources que vient de demander le processus aux ressources allouées aux processus ($Allocation_i$ = $Allocation_i$+ $Request_i$)
   - On retire des besoins du processus les ressources qu'il vient de demander ($Need_i$ = $Need_i$ – $Request_i$)
4. On vérifier grâce à l'algorithme ci-dessous si le nouvel état est sain. Si tel est le cas, on valide la demande, dans le cas contraire, on annule la demande.

L'algorithme suivant est une fonction qui renvoie la valeur *Vrai* si le système est dans un état sain, *Faux* sinon.

Algorithme Verification_Etat_Sain
Début
Work : Tableau[m] de Entier ;
Finish : Tableau[n] de Logique ;

Etape 1 : Work =Available
Finish = Faux ;

Etape 2 : Trouver i tel que : Finish[i]=faux et $Need_i$ <=Work
Si un tel i n'existe pas aller à l'étape 4.

Etape 3 : Work =Work + $Allocation_i$

Finish[i] =Vrai
Aller à l'étape 2

Etape 4 : Si Finish = Vrai (pour tout i) Alors
Verification_Etat_Sain =Vrai
Sinon
Verification_Etat_Sain =Faux
Fin Si

Fin

Remarque :

- Work est initialisé avec les nombres de ressources disponibles
- Finish = Faux veut dire on met toutes les valeurs du vecteur Finish à Faux
- Cet algorithme peut demander mxn^2 opérations pour décider si un état est sain.

Je sais qu'il y a des personnes qui ne comprennent vraiment rien dans cet algorithme, osez l'avouer. Mais, ne vous affolez pas, pour les non-initiés aux algorithmes, voici une manière très simple de résumer cet algorithme ou de vérifier qu'un état est sain (sûr) :

1. Trouver dans la matrice Request une ligne L ($Request_i$) dont les ressources demandées sont toutes inférieures ou égales à celles de disponible (Available) c'est-à-dire $Request_i \leq$ Available. S'il n'existe pas L vérifiant cette condition, il y a interblocage.
2. Supposer que le processus associé à L ou $Request_i$ obtient les ressources et se termine. Supprimer sa ligne et actualiser Available (actualiser Available révient à récupérer les ressources que détient le processus concerné et à les ajouter à Available).
3. Répéter 1 et 2 jusqu'à ce que tous les processus soient terminés (l'état initial était donc sûr) ou jusqu'à un interblocage (l'état initial n'était pas sûr)

❖ Exemple :

Un système possède 5 processus (P0, P1, P2, P3, P4) et 3 types de ressources (A, B, C). Le type de ressources A possède 10 instances, le type de ressources B possède 5 instances et le type de ressources C possède 7 instances. A l'instant T0, l'état des ressources du système est décrit par les matrices *Allocation, Max* et *Available* suivantes :

Matrice : Allocation

	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

Matrice : Max

	A	B	C
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

Matrice : Available

A	B	B
3	3	2

Le contenu de la matrice Need peut être déduit par calcul, Need=Max – Allocation

Matrice : Need

	A	B	C
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

On peut vérifier que le système est dans un état sain, car la **séquence <P1, P3, P4, P2, P0> est saine**.
En effet, on peut satisfaire aux besoins restant de P1, puis à la fin de son exécution récupérer toutes les ressources qu'il détient, puis satisfaire aux besoins restant de P3. A la fin de l'exécution de P3, on récupère toutes les ressources que P3 détient, puis on satisfaire respectivement aux besoins de P4, P2 et P0.

Supposons qu'à l'instant T1 le processus P1 demande une instance supplémentaire du type de ressources A et deux instances du type de ressources C. Nous avons alors : Request =(1, 0, 2).

Afin de décider si la requête peut être immédiatement accordée, on doit d'abord vérifier que

Request$_1$<=Available, ce qui est vrai. On enregistre l'état du système (le contenu des matrices), puis on supposera que la requête a été satisfaite et on arrive à l'état suivant :

Matrice : Allocation

	A	B	C
P0	0	1	0
P1	3	0	2
P2	3	0	2
P3	2	1	1
P4	0	0	2

Matrice : Need

	A	B	C
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1

Matrice : Available

A	B	B
2	3	0

On doit alors déterminer si le nouvel état est sain en appliquant l'algorithme de vérification de l'état sain, ce qui est vrai (la séquence <P1, P3, P4, P0, P2> est saine). On peut donc accorder la demande de ressource faite par P1.

A l'instant T2, une requête (3, 3, 0) arrive du processus P4. Cette requête est immédiatement rejetée parce que les ressources ne sont pas disponibles.

A l'instant T3, une requête (0, 2, 0) arrive du processus P0. Cette requête ne sera pas accordée parce que l'état qu'on obtiendra est malsain, c'est-à-dire aboutit à un interblocage.

❖ **Critique de l'algorithme du Banquier :**

Bien qu'il a l'avantage d'éviter les interblocages, l'algorithme du banquier a néanmoins quelques inconvénients :

- *Coûteux* : L'algorithme est en effet très coûteux en temps d'exécution et en mémoire pour le système Puisqu'il faut maintenir plusieurs matrices, et déclencher à chaque demande de ressource, l'algorithme de vérification de l'état sain qui demande m x n^2 opération. (m est le nombre de types de ressources et n est le nombre de processus).

- *Théorique* : L'algorithme exige que chaque processus déclare à l'avance les ressources qu'il doit utiliser, en type et en nombre. Cette contrainte est difficile à réaliser dans la pratique.

- *Pessimiste* : L'algorithme peut retarder une demande de ressources dès qu'il y a risque d'interblocage (mais en réalité l'interblocage peut ne pas se produire).

2.10.6.4 Les politiques de détection-guérison

Si un système n'emploie pas d'algorithme pour prévenir les interblocages, il peut se produire une situation d'interblocage. Dans ce cas le système doit fournir :

- Un algorithme qui examine l'état du système pour déterminer s'il s'est produit un interblocage.
- Un algorithme pour guérir (corriger) l'interblocage.

La politique de guérison ou détection/guérison des interblocages autorise les interblocages à se produire, elle les détecte et les résout. Pour cette politique, le système maintient un graphe représentant l'allocation des ressources et les attentes des processus.

Le système met à jour le graphe à chaque nouvelle allocation de ressources ou demande d'allocation de ressources. Régulièrement le système parcourt le graphe à la recherche de cycle.

2.10.6.4.1 Détection d'interblocage

Pour détecter un interblocage dans un système disposant de plus d'une instance pour chaque type de ressources, on peut utiliser l'algorithme de détection suivant, qui s'inspire de l'**algorithme du banquier**.

Cet algorithme est lancé périodiquement. Il utilise les structures de données suivantes :

- **Available** : C'est un Vecteur de longueur m indiquant le nombre de ressources disponibles de chaque type. Ainsi, si Available[j]=k, cela veut dire que le type de ressources Rj possède k instances disponibles.
- **Allocation** : C'est une matrice nxm définissant le nombre de ressources de chaque type de ressources actuellement alloué à chaque processus. Ainsi si Allocation[i, j]=k, cela veut dire que l'on a alooué au processus Pi k instances du type de ressources Rj.
- **Request** : C'est une matrice nxm indiquant les ressources supplémentaires que les processus viennent de demander. Ainsi, si Request[i, j]=k, cela veut dire que le processus Pi vient de demander k instances supplémentaires du type de ressources Rj.


```
Algorithme Détection d'interblocage
Début
    Work : Tableau[m] de Entier ;
    Finish : Tableau[n] de Logique ;

    Etape 1 : Work = Available
        Pour i=1 jusqu'à N  Faire
            Si Allocation_i <> 0  Alors    (* voir detail A *)
                Finish[i] =Faux
            Sinon
                Finish[i] =Vrai
            Finsi
        Fin Pour

    Etape 2 : Trouver un indice i tel que :
        Finish[i]=Faux et Request_i <= Work   (* voir detail B *)
        Si un tel i n'existe pas aller à l'étape 4.

    Etape 3 : Work = Work - Allocation
        Finish[i] =Vrai
        Aller à l'étape 2

    Etape 4 : Si Finish[i]=Faux  (pour un certain i)  Alors
                Alors Le système est dans un état d'interblocage
              Sinon
```

```
                Le système n'est pas dans un état d'interblocage
            Fin
Fin
```


- ***detail A :*** $Allocation_i$ <> 0 veut dire que s'il existe un j tel que Allocation[i,j] <> 0
- ***detail B :*** $Request_i$ <= Work veut dire que si pour tout j Request[i,j] <= Work[i,j]
- Work est initialisé avec les nombres de ressources disponibles

❖ **Exemple 1** :

L'état d'allocation des ressources d'un système est donné par le contenu des trois matrices suivantes : *Available*, *Allocation* et *Request*.

Matrice : Allocation

	A	B	C	D
P0	1	0	1	0
P1	2	0	0	1
P2	0	1	2	0

Matrice : Request

	A	B	C	D
P0	2	0	0	1
P1	1	0	1	0
P2	2	1	0	0

Matrice : Available

A	B	C	D
5	2	3	1

En appliquant l'algorithme de détection, on trouvera que les contenus successif des matrices Work et Finish sont :

- Initialisation des matrices Work et Finish (étape 1) :

Work

5	2	3	1

Finish

Faux	Faux	Faux

- **Itération 1**
  - Recherche d'un processus satisfaisant à la condition de l'étape 2 : on choisit **P0** (on peut choisir P1 ou P2), puis on passe à l'étape 3
  - Apres l'exécution de l'étape 3 on a les matrices ci-dessous

Work Finish

4	2	2	1

Vrai	Faux	Faux

- puis on repart à l'étape 2

- **Itération 2**

  - Recherche d'un processus satisfaisant à la condition de l'étape 2 : on choisit **P1,** puis on passe à l'étape 3
  - Apres l'exécution de l'étape 3 on a les matrices ci-dessous

Work

2	2	2	0

Finish

Vrai	Vrai	Faux

- puis on repart à l'étape 2

- **Itération 3**

  - Recherche d'un processus satisfaisant à la condition de l'étape 2 : on choisit **P2,** puis on passe à l'étape 3
  - Apres l'exécution de l'étape 3 on a les matrices ci-dessous

Work

2	1	0	0

Finish

Vrai	Vrai	Vrai

- puis on repart à l'étape 2

- **Itération 4**

  - Recherche d'un processus satisfaisant à la condition de l'étape 2 : Aucun processus ne satisfait à cette condition**,** alors on passe à l'étape 4

- **Etape 4 :** Toutes les valeurs du vecteur Finish sont à Vrai, donc le système n'est pas en situation d'interblocage.

Supposons maintenant que le processus P2 fait une demande supplémentaire d'une instance de la ressource D. Les données du problème deviennent donc :

Matrice : Allocation

	A	B	C	D

P0	1	0	1	0
P1	2	0	0	1
P2	0	1	2	0

Matrice : Request

	A	B	C	D
P0	2	0	0	1
P1	1	0	1	0
P2	2	1	0	1

Matrice : Available

A	B	C	D
5	2	3	1

Une exécution de l'algorithme de détection fera apparaître les contenus suivants :

- Initialisation des matrices Work et Finish (étape 1) :

Work

5	2	3	1

Finish

Faux	Faux	Faux

Indice choisi i=0

- **Itération 1 :**
  - Recherche d'un processus satisfaisant à la condition de l'étape 2 : on choisit **P0** puis on passe à l'étape 3
  - Apres l'exécution de l'étape 3 on a les matrices ci-dessous

Work

4	2	2	1

Finish

Vrai	Faux	Faux

- puis on repart à l'étape 2

- **Itération 2 :**
  - Recherche d'un processus satisfaisant à la condition de l'étape 2 : on choisit **P1** puis on passe à l'étape 3
  - Apres l'exécution de l'étape 3 on a les matrices ci-dessous

Work

2	2	2	0

Finish

Vrai	Vrai	Faux

- puis on repart à l'étape 2

- **Itération 3 :**

  - Recherche d'un processus satisfaisant à la condition de l'étape 2 : Aucun processus ne satisfait à cette condition**,** alors on passe à l'étape 4

- **Etape 4** : Toutes les valeurs du vecteur Finish ne sont pas à Vrai, donc le système est en situation d'interblocage.

❖ **Exemple 2 :**

Soit un système à 3 types de ressources (A, B, C) de valeurs maximale respective (7, 2, 6).

On considère le tableau suivant :

Processus	Allocations			Requêtes		
	A	B	C	A	B	C
P0	0	1	0	0	0	0
P1	2	0	0	2	0	2
P2	3	0	3	0	0	0
P3	2	1	1	1	0	0
P4	0	0	2	0	0	2

La matrice Available est : (0, 0, 0).

On aimerait avoir une suite de processus pouvant permettre l'exécution de tous les processus jusqu'à leur terminaison.

En considérant la suite (P0, P2, P3, P1, P4) on a :

Finish = (faux,faux,faux,faux,faux) Work=(0,0,0)

Choix de P0 : F= (vrai,faux,faux,faux,faux) Work=(0,1,0)

Choix de P2: F= (vrai,faux,vrai,faux,faux) Work=(3,1,3)

Choix de P3 : F= (vrai,faux,vrai,vrai,faux) Work=(5,2,4)

Choix de P1 : F= (vrai,vrai,vrai,vrai,faux) Work=(7,2,4)

Choix de P4 : F= (vrai,vrai,vrai,vrai,vrai) Work=(7,2,6)

Conclusion : La suite (P0, P2, P3, P1, P4) est une suite saine, car elle n'aboutit pas à un interblocage.

Supposons que P2 demande une ressource de type C. on a :

Processus	Allocations			Requêtes		
	A	B	C	A	B	C
P0	0	1	0	0	0	0
P1	2	0	0	2	0	2
P2	3	0	3	0	0	1
P3	2	1	1	1	0	0
P4	0	0	2	0	0	2

Choix de P0 : T= (0 1 0) : on ne peut satisfaire aucun autre processus.

P1, P2, P3, P4 sont en interblocage.

2.10.6.4.2 Guérison des interblocages

Il existe plusieurs solutions pour corriger un interblocage si le système détecte qu'il en existe un.

- *Correction manuelle* : Le système alerte l'opérateur qu'il s'est produit un interblocage, et l'invite à le traiter manuellement (en relançant le système par exemple).

- *Terminaison de processus* : On peut éliminer un interblocage en arrêtant un ou plusieurs processus. On peut choisir d'arrêter tous les processus, ou bien de les arrêter un à un jusqu'à éliminer l'interblocage.

- *Réquisition de ressources* : Pour éliminer l'interblocage, en procédant à la réquisition d'une ou plusieurs ressources, en les enlevant à un processus et en les donnant à un autre jusqu'à ce que l'interblocage soit éliminé.

2.10.6.4.3 Difficultés

La politique de guérison peut présenter plusieurs difficultés :

1. Sa mise en œuvre est coûteuse : Il faut maintenir le graphe d'allocation, régulièrement parcourir le graphe à la recherche de cycles et enfin remédier à l'interblocage par destruction de processus.

2. Une autre difficulté tient à la période de parcours du graphe : si cette période est petite, le graphe est parcouru souvent et consomme ainsi les ressources du système inutilement car la probabilité d'un interblocage est faible. Si la période de parcours est grande, le graphe sera parcouru moins souvent et la probabilité de trouver un interblocage sera plus forte. Mais, le nombre de processus impliqués dans un l'interblocage risque d'être d'autant plus grand que la période de parcours du graphe est grande.

3. Par ailleurs le choix des processus à avorter pour remédier à un interblocage n'est pas forcément facile. Une solution est de systématiquement détruire tous les processus impliqués dans l'interblocage mais on peut essayer de raffiner cette solution en choisissant les processus à avorter : se pose ici le problème du choix qui va conduire à éliminer l'interblocage en minimisant le nombre de processus avortés ou le coût pour le système de ces avortements. Ainsi dans le cas de la figure ci-dessus « **Exemple d'attente circulaire** », on pourra choisir d'avorter P2 plutôt que P1 car P1 détient déjà deux ressources sur trois.

2.11 Synchronisation des processus

2.11.1 Introduction

Sur une plateforme multiprogrammée, les processus ont généralement besoin de communiquer pour compléter leurs tâches.

- Un processus est dit indépendant, s'il n'affecte pas les autres processus ou ne peut pas être affecté par eux. Un processus qui ne partage pas de données avec d'autres processus est indépendant.
- Un processus est dit coopératif s'il peut affecter les autres processus en cours d'exécution ou être affecté par eux. Un processus qui partage des données avec d'autres processus est un processus coopératif.
- Les données partagées par les processus coopératifs se trouvent en mémoire principale ou en mémoire secondaire dans un fichier. Il y a alors risque d'incohérence des données.

- L'exécution d'un processus peut être affectée par l'exécution des autres processus ou il peut affecter lui-même leurs exécutions
- La communication interprocessus est assurée généralement via des données partagées qui peuvent se trouver dans la mémoire principale ou dans un fichier.

Les accès concurrents (simultanés) à des données partagées peuvent conduire à des incohérences dans les résultats obtenus.

Quatre conditions doivent être vérifiées pour assurer une bonne synchronisation des processus:

- **Exclusion Mutuelle**: Deux processus ne doivent pas se trouver simultanément dans leurs sections critiques.
- Progression : Aucun processus à l'extérieur de sa section critique ne doit bloquer les autres processus.
- Attente bornée : Aucun processus ne doit attendre indéfiniment pour entrer dans sa section critique.
- Aucune hypothèse : Il ne faut pas faire d'hypothèse quant à la vitesse ou le nombre de processeurs

2.11.2 Exclusion mutuelle

Les processus disposent chacun d'un espace d'adressage protégé inaccessible aux autres processus. Pour communiquer et s'échanger des données, les processus peuvent utiliser des outils de communications offerts par le système. La manipulation de ces outils de communication doit se faire dans le respect de règles de synchronisation qui vont assurer que les données échangées entre les processus restent cohérentes et ne sont pas perdues. Un premier problème de synchronisation est celui de l'accès par un ensemble de processus à une ressource critique, c'est-à-dire une ressource à un seul point d'accès donc utilisable par un seul processus à la fois.

❖ **Un exemple simple pour définir le problème**

Nous allons mettre en lumière le problème qui peut se poser sur un exemple simple : on considère donc un petit programme de réservation de place (dans un avion, un train, …).

Réservation :

```
Si nb_place > 0 alors
        Réserver une place
        nb_place = nb_place - 1
fsi
```

Ce programme réservation peut être exécuté par plusieurs processus à la fois (autrement dit, le programme est réentrant). La variable nb_place, qui représente le nombre de place restant dans l'avion par exemple, est ici une variable d'état du système (de réservation) On considère l'exécution de deux processus *Client_1* et *Client_2* :

- *Client_1* est commuté par l' ordonnanceur juste après avoir testé la valeur de la variable nb_place (nb_place = 1).

- *Client_2* s'exécute à son tour, teste nb_place qu'il trouve également égale à 1 et donc effectue une réservation en décrémentant d'une unité la variable nb_place. Nb_Place devient égale à 0. Comme le processus *Client_2* a terminé son exécution, *Client_1* reprend la main.

- Comme *Client_1* avait trouvé la variable nb_place comme étant égale à 1 juste avant d'être commuté, il continue son exécution en décrémentant à son tour nb_place. De ce fait, nb_place devient égale à –1 ce qui est incohérent !!! Une même place a été allouée à deux clients différents !

La variable nb_place doit être accédée par un seul processus à la fois pour rester cohérente : ici en l'occurrence le processus *Client_1* qui a commencé la réservation en premier. Nb_Place est donc une ressource critique.

La propriété d'exclusion mutuelle assure qu'une ressource critique ne peut jamais être utilisée par plus de un processus à la fois.

Pour garantir l'exclusion mutuelle, il faut donc entourer l'utilisation de la variable nb_place d'un prélude et d'un postlude. Le prélude prend la forme d'une "protection"qui empêche un processus de manipuler la variable nb_place si un autre processus le fait déjà. Ainsi le processus *Client_2* est mis en attente dès qu'il cherche à accéder à la variable nb_place déjà possédée par le processus *Client_1*. Le postlude prend la forme d'une "fin de protection" qui libère la ressource nb_place et la rend accessible au processus *Client_2*.

2.11.3 Section critique

L'exclusion mutuelle ne se limite pas à empêcher le partage de ressources, mais aussi à éviter que deux processus effectuent simultanément des actions qui provoqueraient une situation conflictuelle. L'ensemble des instructions consécutives qui doivent être exécutées dans un processus en exclusion mutuelle constituent une **section critique**. Il peut y avoir plusieurs sections critiques dans le même processus. La section critique d'un programme n'a de sens que s'il existe (ou peut exister) au moins une autre section critique correspondant aux mêmes ressources dans au moins un autre processus. Il peut aussi y avoir plusieurs sections critiques dans un programme, correspondant chacun à l'accès à des ressources critiques différentes.

Quelques règles doivent être données pour une utilisation correcte des sections critiques :

- une section critique doit être la plus courte possible. Elle perturbe en effet le quasi-parallélisme et pourrait priver les autres processus de la ressource processeur.
- un processus **ne peut rester indéfiniment dans une section critique**. Cela implique que, si un programme boucle dans une section critique, le système puisse non seulement l'interrompre au bout d'un temps raisonnable, mais encore débloquer les autre processus en attente de la même ressource.
- **il ne faut faire aucune hypothèse sur les vitesses d'exécution** relative des différentes instructions,
- un processus hors de sa section critique ne doit pas pouvoir empêcher un autre d'accéder à la ressource critique associée à cette section.

❖ **Définition de ressource critique**

Une ressource critique est une ressource accessible par un seul processus à la fois

❖ **Réalisation d'une section critique à l'aide des interruptions matérielles**

Nous rappelons que le mécanisme sous-jacent au réordonnancement des processus peut être la survenue d'une interruption horloge. Aussi une solution pour réaliser l'exclusion mutuelle est de masquer les interruptions dans le prélude et de les démasquer dans le postlude. Ainsi les interruptions sont masquées dès qu'un processus accède à la ressource nb_place et aucun événement susceptible d'activer un autre processus ne peut être pris en compte. Cependant, cette solution est moyennement satisfaisante car elle empêche l'exécution de tous les processus y compris ceux ne désirant pas accéder à la ressource critique. De plus, le masquage et le démasquage des interruptions sont des opérations réservées au mode superviseur et ne sont donc pas accessibles pour les processus utilisateurs.

Une autre solution est d'utiliser un outil de synchronisation offert par le système : les sémaphores.

2.11.4 Les sémaphores

2.11.4.1 Introduction

Un **sémaphore** est une variable (ou un type de donnée abstrait) et constitue la méthode utilisée couramment pour restreindre l'accès à des ressources partagées (par exemple un espace de stockage) et synchroniser les processus dans un environnement de programmation concurrente. Le sémaphore a été inventé par Edsger Dijkstra et utilisé pour la première fois dans le système d'exploitation THE Operating system.

Les sémaphores fournissent la solution la plus courante pour le fameux problème du « dîner des philosophes », bien qu'ils ne permettent pas d'éviter tous les interblocages (ou *deadlocks*). Pour pouvoir exister sous forme logicielle, ils nécessitent une implémentation matérielle (au niveau du microprocesseur), permettant de *tester et modifier* la variable protégée au cours d'un cycle insécable. En effet, dans un contexte de multiprogrammation, on ne peut prendre le risque de voir la variable modifiée par un autre processus juste après que le processus courant vient de la tester et avant qu'il ne la modifie.

Un sémaphore est une variable entière partagée. Sa valeur est positive ou nulle et elle est uniquement manipulable à l'aide de deux opérations wait(s) et signal(s), où s'est un identificateur désignant le sémaphore.

- wait(s) décrémente s si s > 0 ; si non, le processus exécutant l'opération wait(s) est mis en attente.
- signal(s) incrémente s. L'exécution de signal(s) peut avoir pour effet (pas nécessairement immédiat) qu'un processus en attente sur le sémaphore s reprend son exécution.

2.11.4.2 Quelques variantes de sémaphore

Pour adapter le concept des sémaphores à certains problèmes spécifiques et, en particulier, pour répondre aux exigences du temps réel, plusieurs variantes ont été proposées :

- **les sémaphores privés**

Un sémaphore est ***privé à un processus*** dit propriétaire, si ***seul*** ce processus ***peut effectuer une opération P*** sur ce sémaphore. Il ne peut donc y avoir ***qu'au plus un processus en attente*** derrière un sémaphore privé, le processus propriétaire, et il n'est pas nécessaire alors d'avoir une file d'attente associée à un sémaphore privé. ***Tout processus***, y compris le propriétaire, ***peut***, par contre, ***exécuter*** une opération ***V sur le sémaphore*** privé.

- **les sémaphores binaires**,

Un sémaphore binaire est une variante du mécanisme de sémaphore général. Le ***compteur*** associé au sémaphore est simplement ***remplacé par un booléen***, qui ne peut donc prendre que les valeurs Vrai ou Faux ou bien 0 ou 1. Ce type de sémaphore permet de gérer la mise en attente de processus, mais pas de compter.

- **l'opération P immédiate**

L'opération P immédiate permet à un processus de demander des tickets, en lui évitant d'être bloqué si la demande ne peut pas être honorée : il sera seulement averti de l'échec de sa demande.

- **l'opération P temporisée**

Grâce à l'opération P temporisée, un processus ***précise le temps maximum*** pendant lequel il ***accepte d'attendre*** que sa requête soit satisfaite : ce délai écoulé, le processus est débloqué et prévenu de l'échec.

- **l'opération P avec priorité**.

L'opération P avec priorité ***range les demandes*** dans la file d'attente ***en fonction d'une priorité***, ce qui permet à chaque fois de servir en premier le processus réalisant le traitement le plus urgent.

2.11.4.3 L'implémentation des sémaphores

Les sémaphores sont implémentés dans le noyau du système.

- Les valeurs des sémaphores se trouvent dans une table conservée dans la mémoire du noyau. Un sémaphore est caractérisé par son numéro qui correspond à une position dans cette table.

- Il y a des appels systèmes pour créer ou libérer un sémaphore, ainsi que pour exécuter les opérations wait et signal. Ces Opérations sont exécutées en mode superviseur et donc de façon indivisible (les interruptions sont interdites en mode superviseur).

2.11.4.4 Opérations

Un sémaphore Sem est une structure système composée d'une file d'attente L de processus et d'un compteur K, appelé niveau du sémaphore et contenant initialement une valeur Val. Cette structure ne peut être manipulée que par trois opérations P(Sem), V(Sem) et Init(Sem, Val). Une propriété importante de ces opérations est qu'elles sont indivisibles c'est-à-dire que l'exécution de ces opérations ne peut pas être interrompue. Un outil sémaphore peut être assimilé à un distributeur de jetons; l'acquisition d'un jeton donnant le droit à un processus de poursuivre son exécution

P et V du néerlandais *Proberen* et *Verhogen* signifient "tester" et "incrémenter". La valeur initiale d'un sémaphore est le nombre d'unités de ressource (exemple : mémoires, imprimantes...) ; elle est décrémentée à chaque fois qu'un processus exécute l'opération P. Si elle est positive, elle représente donc le nombre de ressources libres, sinon si elle est négative sa valeur absolue correspond au nombre de processus en attente.

- Les opérations P et V doivent être indivisibles, ce qui signifie que les différentes opérations ne peuvent pas être exécutées plusieurs fois de manière concurrente. Un processus qui désire exécuter une opération qui est déjà en cours d'exécution par un autre processus doit attendre que le premier termine.

- Dans les livres en anglais, les opérations V et P sont quelquefois appelées respectivement *up* et *down*. En conception logicielle, elles sont appelées *signal* et *wait* ou *release* et *take*. De façon informelle, un bon moyen de se rappeler laquelle fait quoi est le procédé mnémotechnique les associant à *Puis-je ?* ou *Prendre* et *Vas-y !* ou *Vendre*.

- Pour éviter l'attente, un sémaphore peut avoir une file de processus associée (généralement une file du type FIFO). Si un processus exécute l'opération P sur un sémaphore qui a la valeur zéro, le processus est ajouté à la file du sémaphore. Quand un autre processus incrémente le sémaphore en exécutant l'opération V, et qu'il y a des processus dans la file, l'un d'eux est retiré de la file et reprend la suite de son exécution.

2.11.4.4.1 L'opération Init

L'opération Init a pour but d'initialiser le sémaphore, c'est-à-dire qu'elle met à vide la file d'attente L et initialise avec la valeur Val le compteur K : on définit ainsi le nombre de jetons initiaux dans le sémaphore. Cette opération ne doit être utilisée qu'une seule et unique fois.


```
Init (Sem, Val)
 début
      Sem. K := Val;
      Sem. L := vide;
 fin
```

2.11.4.4.2 L'opération P

L'opération `P(Sem)` ou `wait (sem)` met en attente le processus courant jusqu'à ce qu'une ressource soit disponible, ressource qui sera immédiatement allouée au processus courant. `P(Sem)` "attribue un jeton" au processus appelant si il en reste au moins un et sinon bloque le processus dans `Sem.L`. L'opération P est donc une opération éventuellement bloquante pour le processus élu qui l'effectue. Dans le cas du blocage, il y aura réordonnancement. Concrètement, le compteur K du sémaphore est décrémenté d'une unité. Si la valeur du compteur devient négative, le processus est bloqué.

```
P (Sem)
 début
        Sem.K := Sem.K – 1;
        Si Sem.K < 0 alors
                ajouter ce processus à Sem.L
                bloquer ce processus
        fsi
 fin
```

2.11.4.4.3 L'opération V

L'opération `V(Sem)` ou `signal(Sem)` est l'opération inverse; elle rend simplement une ressource disponible à nouveau après que le processus a terminé de l'utiliser.

L'opération V(Sem) a pour but de "rendre un jeton" au sémaphore. De plus, si il y a au moins un processus bloqué dans la file d'attente L du sémaphore, un processus est réveillé. La gestion des réveils s'effectue généralement en mode FIFO (on réveille le processus le plus

anciennement endormi). L'opération V est une opération qui n'est jamais bloquante pour le processus appelant.


```
V (Sem)
Début
        Sem.K := Sem.K + 1;
        Si Sem.K <= 0  alors
                sortir un processus de Sem.L
                réveiller ce processus
        fsi
 fin
```

- **Signification de la valeur du compteur K**

- Si Sem.K > 0, alors Sem.K est le nombre d'opérations P(Sem) passantes
- Si Sem.K <= 0, alors valeur_absolue(Sem.K) est le nombre de processus bloqués dans Sem.L

2.11.4.5 Réalisation d'une section critique à l'aide des sémaphores

La réalisation d'une section critique à l'aide de l'outil sémaphore s'effectue en utilisant un sémaphore MUTEX, dont le compteur K est initialisé à 1. Le prélude de la section critique correspond à une opération P(MUTEX). Le postlude de la section critique correspond à une opération V(MUTEX).

```
INIT (MUTEX, 1);

P(MUTEX);

Section critique
V(MUTEX)
```

❖ **Sémaphore mutex initialisé à 1**

Prog1	Prog2
P(mutex)	P(mutex)
x=lire (cpte)	x=lire (cpte)
x = x + 10	x = x - 100
écrire (cpte=x)	écrire (cpte=x)

V(mutex) V(mutex)

Le **fonctionnement de la section critique avec l'exemple précédent:**

- *Client_1* effectue en premier la demande de réservation : le P(Mutex) est passant et le "jeton" est alloué au processus *Client_1*. Juste après le test de la valeur de Nb_Place, *Client_1* perd donc la main ;
- *Client_2* est élu mais le P(Mutex) est bloquant : il n'y a plus de jeton disponible dans le compteur du sémaphore. Comme *Client_2* est bloqué, *Client_1* reprend la main. Lorsqu'il a achevé sa réservation, *Client_1* relâche le jeton par un V(Mutex) : *Client_2* est alors réveillé et le jeton lui est attribué.

2.11.4.6 Utilisation

Les sémaphores sont toujours utilisés dans les langages de programmation qui n'implémentent pas intrinsèquement d'autres formes de synchronisation. Ils sont le mécanisme primitif de synchronisation de beaucoup de systèmes d'exploitation. La tendance dans le développement des langages de programmation est de s'orienter vers des formes plus structurées de synchronisation comme les moniteurs. Outre les problèmes d'interblocages qu'ils peuvent provoquer, les sémaphores ne protègent pas les programmeurs de l'erreur courante qui consiste à bloquer par un processus un sémaphore qui est déjà bloqué par ce même processus, et d'oublier de libérer un sémaphore qui a été bloqué. Hoare, Hansen, Andrews, Wirth, et même Dijkstra ont jugé le sémaphore obsolète.

2.11.4.7 Exemples

- **Sémaphores bloquants**

Outre les sémaphores à compteur interne, il existe également les sémaphores bloquants. Un sémaphore bloquant est un sémaphore qui est initialisé avec la valeur 0. Ceci a pour effet de bloquer n'importe quel *thread* qui effectue P(S) tant qu'un autre *thread* n'aura pas fait un V(S). Ce type d'utilisation est très utile lorsque l'on a besoin de contrôler l'ordre d'exécution entre *threads*. Cette utilisation des sémaphores permet de réaliser des barrières de synchronisation.

- **Exclusion mutuelle**

Il existe également le sémaphore binaire qui est une exclusion mutuelle (alias mutex). Il est toujours initialisé avec la valeur 1.

- **Lecteurs-rédacteurs**

Un problème classique pouvant être résolu à l'aide des sémaphores est le problème des lecteurs/rédacteurs. Ce problème traite de l'accès concurrent en lecture et en écriture à une ressource. Plusieurs processus légers (*thread*) peuvent lire en même temps la ressource, mais il ne peut y avoir qu'un et un seul thread en écriture.

- **Producteurs-consommateurs**

Lorsque des processus légers souhaitent communiquer entre eux, ils peuvent le faire par l'intermédiaire d'une file. Il faut définir le comportement à avoir lorsqu'un thread souhaite lire depuis la file lorsque celle-ci est vide et lorsqu'un thread souhaite écrire dans la file mais que celle-ci est pleine.

2.11.4.8 Quelques problèmes

Les sémaphores servant notamment à effectuer de la synchronisation, ils peuvent conduire à des situations indésirables, par exemple :

- **Interblocage**

Un **interblocage** (ou **étreinte fatale**, ***deadlock*** en anglais) est un phénomène qui peut survenir en programmation concurrente. L'interblocage se produit lorsque deux processus concurrents s'attendent mutuellement. Les processus bloqués dans cet état le sont définitivement, il s'agit donc d'une situation catastrophique

- **Inversion de priorité**

L'**inversion de priorité** est un phénomène qui peut se produire en programmation concurrente. Il s'agit d'une situation dans laquelle un processus de haute priorité ne peut pas avoir accès au processeur car il est utilisé par un processus de plus faible priorité.

2.11.4.9 Conclusions sur les sémaphores

- Les sémaphores sont un excellent mécanisme pour gérer la mise en attente de processus.
- Lorsqu'il faut gérer la mise en attente en combinaison avec la manipulation d'un compteur, les sémaphores sont très bien adaptés.
- L'utilisation des sémaphores binaires indique quelles difficultés apparaissent lorsque l'on combine l'attente implémentée par un sémaphore avec la manipulation d'une autre structure de donnée.
- Il serait intéressant d'avoir une solution systématique pour gérer ce type de problème.

2.11.5 Inversion de priorité

Une ressource critique est une ressource à un seul point d'accès, c'est-à-dire accessible par un seul processus à la fois. L'utilisation s'effectue en exclusion mutuelle. L'allocation et la désallocation d'une ressource critique peuvent être gérées à l'aide d'un sémaphore R initialisé à 1. L'allocation se traduit alors par un P(R) et la restitution par un V(R).

L'inversion de priorité est la situation pour laquelle une tâche de priorité intermédiaire (T3) s'exécute à la place d'une tâche de forte priorité (T1) parce que la tâche de forte priorité (T1) est en attente d'une ressource acquise par une tâche de plus faible priorité (T2). A priori, on ne peut pas borner le temps d'attente de la tâche de haute priorité qui risque ainsi de dépasser son échéance car l'exécution de T3 n'était pas prévisible.

2.11.5.1 Solutions mises en œuvre

Les solutions mises en œuvre ne cherchent pas à éviter le phénomène d'inversion de priorité, mais permettent seulement de borner le temps d'attente des tâches sur l'accès aux ressources. Ces bornes B, peuvent ensuite être ajoutées au temps d'exécution des tâches et ainsi être intégrées dans les tests d'acceptation des configurations. Il existe deux protocoles principaux :

- le protocole de l'héritage de priorité
- le protocole de la priorité plafonnée

Ces protocoles sont mis en œuvre de façon standard dans certains exécutifs temps réel dans les opérations P et V des sémaphores.

2.11.5.1.1 Le protocole de l'héritage de priorité

Principe : une tâche T détentrice d'une ressource R hérite de la priorité des tâches T' plus prioritaires mises en attente sur cette ressource R, jusqu'à ce qu'elle libère la ressource R. Ainsi T est ordonnancée au plus vite pour libérer le plus rapidement possible la ressource R. En effet, les tâches de priorité intermédiaire ne peuvent plus s'exécuter puisque leur priorité devient inférieure à celle de la tâche T.

Inconvénient : ce protocole ne prévient pas les interblocages.

2.11.5.1.2 Le protocole de la priorité plafonnée

Principe : Chaque ressource R possède une priorité qui est celle de la tâche de plus haute priorité pouvant demander son accès. Le principe est ensuite similaire au précédent : une tâche T détentrice d'une ressource R hérite de la priorité des tâches T' plus prioritaires mises en attente

sur cette ressource R, jusqu'à ce qu'elle libère la ressource R. Ainsi T est ordonnancée au plus vite pour libérer le plus rapidement possible la ressource R Cependant, la tâche T ne peut obtenir la ressource R que si la priorité de R est strictement supérieure à celles de toutes les ressources déjà possédées par les autres tâches T". Par ce biais, on prévient l'interblocage.

2.11.5.1.3 Evitement de l'interblocage

Nous approfondissons ici la manière dont les protocoles que nous venons de voir ce comportent vis-à-vis du problème de l'interblocage. Nous considérons deux tâches T1 et T2 telles que T1 utilise tout d'abord la ressource R1 puis la ressource R2 tandis que T2 utilise d'abord la ressource R2 depuis la ressource R1. En appliquant la protocole de l'héritage de priorité : ici la tâche T2 s'exécute et obtient la ressource R2, puis T1 s'exécute et obtient la ressource R1.T1 est ensuite bloquée lorsqu'elle demande à accéder à la ressource R2 et T2 hérite de la priorité de T1. T2 poursuit son exécution, demande à accéder à R1 et se bloque à son tour. Rien n'empêche les deux tâches de tomber en interblocage.

2.12 Communication inter processus

2.12.1 Introduction

Les communications entre processus regroupent un ensemble de mécanismes permettant à des processus concurrents (ou distants) de communiquer. Ces mécanismes peuvent être classés en trois catégories :

- Les outils permettant aux processus de s'échanger des données
- Les outils permettant de synchroniser les processus, notamment pour gérer le principe de section critique (partage de ressource de la machine)
- Les outils offrant directement les caractéristiques des deux premiers (échanger des données et synchroniser des processus)

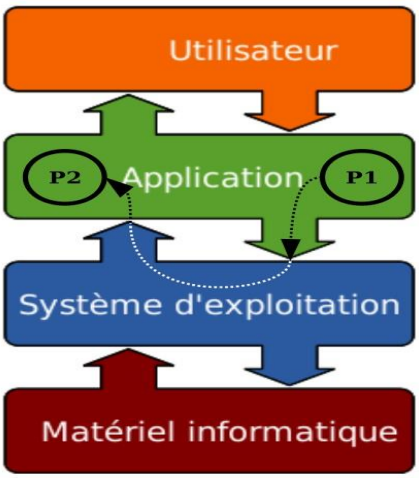


Figure 10 : Communication inter-processus

Pourquoi interagir avec les processus ?

- Les processus interagissent entre eux et doivent donc pouvoir communiquer entre eux
- Le système doit pouvoir avertir les processus en cas de défaillance d'un composant du système
- L'utilisateur doit pouvoir gérer les processus (arrêt, suspension, etc.)

Il existe de multiples moyens de réaliser une communication interprocessus :

- Par **fichiers** (~ tout système)
- **Signaux** (Unix/Linux/MacOS, pas vraiment sous Windows)
- **Sockets** (~ tout système)
- **Les tubes** ou **pipes** :
  - ***Les tubes anonymes***
  - ***Les tubes nommés***
- **Les IPC** (Inter Process Comunication) :
  - ***File d'attente de message*** (~ tout système)
  - ***Mémoire partagée*** (tout système POSIX, Windows)
  - ***Sémaphores*** (tout système POSIX, Windows)
- **Passage de message** : MPI, RMI, CORBA …

- Etc.

2.12.2 L'échange de donnée

❖ Les **fichiers** peuvent être utilisés pour échanger des informations entre deux, ou plusieurs processus. Et par extension de la notion de fichiers, les **bases de données** peuvent aussi être utilisées pour échanger des informations entre deux, ou plusieurs processus.

❖ **La mémoire** (principale) d'un système peut aussi être utilisée pour des échanges de données. Suivant le type de processus, les outils utilisés ne sont pas les mêmes :

- Dans le cas des processus lourds, l'espace mémoire du processus n'est pas partagé. On utilise alors des mécanismes de mémoire partagée, comme les segments de mémoire partagée pour Unix.
- Dans le cas des processus légers (threads) l'espace mémoire est partagée, la mémoire peut donc être utilisée directement.

Quelle que soit la méthode utilisée pour partager les données, ce type de communication pose le problème des sections critiques.

2.12.3 Synchronisation

Les mécanismes de synchronisation sont utilisés pour résoudre les problèmes de sections critiques et plus généralement pour bloquer et débloquer des processus suivant certaines conditions :

- Les verrous permettent de bloquer tout ou une partie d'un fichier
- Les sémaphores (et les mutex) sont un mécanisme plus général, ils ne sont pas associés à un type particulier de ressource et permettent de limiter l'accès concurrent à une section critique à un certain nombre de processus
- Les signaux (ou les évènements) permettent aux processus de communiquer entre eux : réveiller, arrêter ou avertir un processus d'un événement.

L'utilisation des mécanismes de synchronisation est difficile et peut entraîner des problèmes d'interblocage (tous les processus sont bloqués).

2.12.4 Échange de données et synchronisation

Ces outils regroupent les possibilités des deux autres et sont souvent plus simples d'utilisation.

- L'idée est de communiquer en utilisant le principe des files (notion de boite aux lettres), les processus voulant envoyer des informations (messages) les placent dans la file ; ceux voulant les recevoir les récupèrent dans cette même le. Les opérations d'écriture et de lecture dans la file sont bloquantes et permettent donc la synchronisation.

- Ce principe est utilisé par :

  - les files d'attente de message (message queue) sous Unix,
  - les sockets Unix ou Internet,
  - les tubes (nommés ou non),
  - la transmission de messages (Message Passing) (DCOM, CORBA, SOAP, ...)

2.12.5 IPC Système V / POSIX

Les IPC, ou Inter Process Communication, apparus avec SystemV d'UNIX et repris dans linux, ne sont pas des fichiers mais des éléments mémoires gérés par le noyau, donc avec de meilleures performances de gestion.

Les IPC (Système V ou POSIX) recouvrent 3 mécanismes de communication entre processus:

- Les files de messages (message queue), dans lesquelles un processus peut glisser des données ou en extraire. Les messages étant typés, il est possible de les lire dans un ordre différent de celui d'insertion, bien que par défaut la lecture se fasse suivant le principe de la file d'attente.
- Les segments de mémoire partagée (shared memory), qui sont accessibles simultanément par deux processus ou plus, avec éventuellement des restrictions telles que la lecture seule.
- Les sémaphores, qui permettent de synchroniser l'accès à des ressources partagées.

Les IPC permettent de faire communiquer deux processus d'une manière asynchrone, l'inverse des tubes.

2.12.5.1 Les files de messages

Les files de messages sont des listes chaînées gérées par le noyau dans lesquelles un processus peut déposer des données (messages) ou en extraire. Elle correspond au concept de boîte aux lettres.

- Un message est une structure comportant un nombre entier (le type du message) et une suite d'octets de longueur arbitraire, représentant les données proprement dites.
- Les messages étant typés, il est possible de les lire dans un ordre différent de celui d'insertion. Le processus récepteur peut choisir de se mettre en attente soit sur le premier message disponible, soit sur le premier message d'un type donné.

Les files de messages autorisent des mécanismes de multiplexage, c'est-à-dire la possibilité d'envoyer des messages à plusieurs processus dans la même file. Ce que ne permet pas, par exemple, un socket simple.

Les messages forment un mode de communication privilégié entre les processus. Ils sont utilisés dans le système pédagogique Minix de Tanenbaum. Ils sont au cœur de Mach qu'on présente comme un successeur possible d'Unix et qui a inspiré Windows NT pour certaines parties. Par ailleurs, ils s'adaptent très bien à une architecture répartie et leur mode de fonctionnement est voisin des échanges de données sur un réseau.

Les communications de messages se font à travers deux opérations fondamentales : `envoie(message)` et `reçois(message)`. (`send` et `receive`). Les messages sont de tailles variables ou fixes. Les opérations d'envoi et de réception peuvent être soit directes entre les processus, soit indirectes par l'intermédiaire d'une boîte aux lettres.

Les communications directes doivent identifier le processus, par un n° et une machine, par exemple. On aura alors : `envoie(Proc, Message)` et `reçois(Proc, Message)`. Dans ce cas la communication est établie entre deux processus uniquement, par un lien relativement rigide et bidirectionnel. On peut rendre les liaisons plus souples en laissant vide l'identité du processus dans la fonction `reçois`.

Les communications peuvent être indirectes grâce à l'utilisation d'une boîte aux lettres (un « port » dans la terminologie des réseaux). Les liens peuvent alors unir plus de deux processus du moment qu'ils partagent la même boîte aux lettres. On devra néanmoins résoudre un certain nombre de problèmes qui peuvent se poser, par exemple, si deux processus essayent de recevoir simultanément le contenu d'une même boîte.

Les communications se font de manière synchrone ou asynchrone. Le synchronisme peut se représenter par la capacité d'un tampon de réception. Si le tampon n'a pas de capacité, l'émetteur doit attendre que le récepteur lise le message pour pouvoir continuer. Les deux processus se synchronisent sur ce transfert et on parle alors d'un « rendez-vous ». Deux processus asynchrones : P et Q, peuvent aussi communiquer de cette manière en mettant en œuvre un mécanisme d'acquittement :

P	Q
envoie(Q, message) **reçois(Q, message)**	reçois(P, message) envoie(P, acquittement)

- Les avantages principaux de la file de message (par rapport aux tubes et aux tubes nommés) sont :

    - un processus peut émettre des messages même si aucun processus n'est prêt à les recevoir.
    - les messages déposés sont conservés, même après la mort de l'émetteur, jusqu'à leur consommation ou la destruction de la file.

- Le principal inconvénient de ce mécanisme est la limite de la taille des messages (8192 octets sous Linux) ainsi que celle de la file.

2.12.5.2 Segment de mémoire partagée

Les processus peuvent avoir besoin de partager de l'information. On peut concevoir des applications qui communiquent à travers un segment de mémoire partagée. Le principe est le même que pour un échange d'informations entre deux processus par un fichier. Dans le cas d'une zone de mémoire partagée, on devra déclarer une zone commune par une fonction spécifique, car la zone mémoire d'un processus est protégée.

Le système Unix fournit à cet effet un ensemble de routines permettant de créer et de gérer un segment de mémoire partagée (shared memory).

- Un segment de mémoire partagée est identité de manière externe par un nom qui permet tout processus possédant les droits, d'accéder ce segment. Lors de la création ou de l'accès un segment mémoire, un numéro interne est fourni par le système.
- Parmi les mécanismes de communication entre processus, l'utilisation de segments de mémoire partagée est la technique la plus rapide, car il n'y a pas de copie des données transmises. Ce procédé de communication est donc parfaitement adapté au partage de gros volumes de données entre processus distincts.

Un soucis des mécanismes de communication classique (fichiers, tubes, etc.), lorsqu'on échange de grande quantités de données est qu'ils nécessitent le transfert des données depuis l'espace d'adressage du processus émetteur vers l'espace noyau, puis de l'espace noyau vers l'espace d'adressage du processus destinataire, voir même par le système de fichier. Perte de temps. L'un des avantages des segments de mémoire partagés est que les processus vont partager des pages mémoires directement par l'intermédiaire de leur espace d'adressage. Gain de temps. Cependant, l'utilisation de ces espaces partagés peut entraîner des effets de bords si les processus ne se synchronisent pas (signal, sémaphore, etc.). D'autre part, un segment a une existence indépendante des processus.

Le système Unix fournit les primitives permettant de partager la mémoire. Windows NT aussi sous le nom de fichiers mappés en mémoire. Ces mécanismes, bien que très rapides, présentent l'inconvénient d'être difficilement adaptables aux réseaux. Pour les

communications locales, la vitesse est sans doute semblable à celle de la communication par un fichier à cause de la mémoire cache. Lorsqu'il a besoin de partager un espace mémoire, le programmeur préférera utiliser des fils d'exécution.

- Si deux processus écrivent et lisent "en même temps" dans une même zone de mémoire partagée, il ne sera pas possible de savoir ce qui est réellement pris en compte. Le segment de mémoire partagée est considéré comme une **section critique**.

- D'où l'obligation d'utiliser des sémaphores pour ne donner l'accès cette zone qu'un processus à la fois.

2.12.6 Les tubes

2.12.6.1 Définition

Les tubes (pipe) sont un mécanisme de communication entre processus résidants sur une même machine. On peut distinguer 2 catégories :

- les tubes anonymes (volatiles) : ils concernent des processus issus de la même application
- les tubes nommés (persistants) : ils concernent des processus totalement indépendants

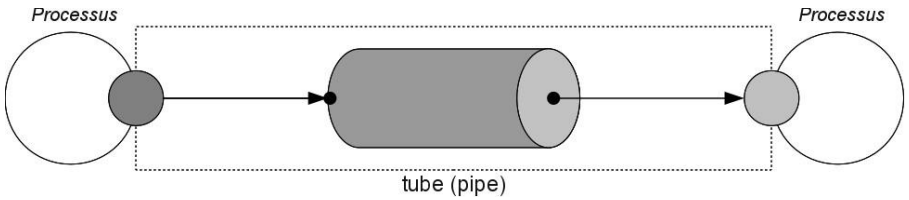
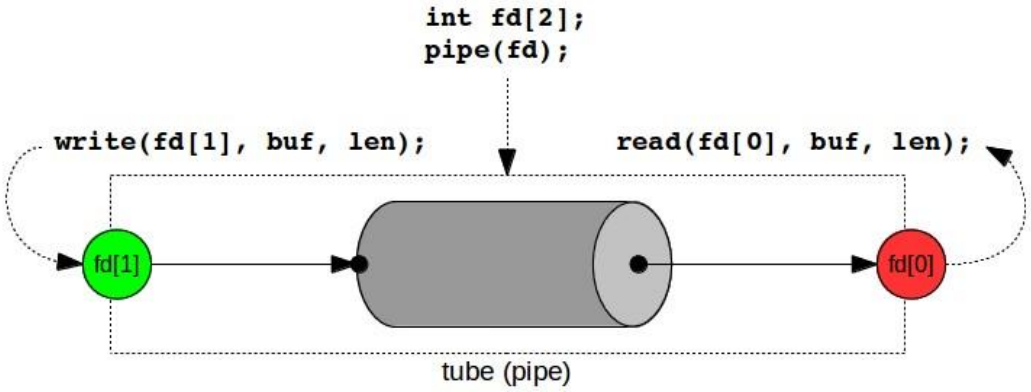


Figure 11 : tube

2.12.6.2 Principe

Un tube peut être vu comme un tuyau dans lequel un processus, à une extrémité, produit des informations, tandis qu'à l'autre extrémité un autre processus les consomme.



Un tube est créé par l'appel système pipe (int descripteur[2]) qui remplit le tableau descripteur avec les descripteurs de chacune des extrémités du tube (l'entrée en lecture seule et la sortie en écriture seule).

Le système Unix offre des primitives qui permettent de synchroniser facilement des processus lecteurs et écrivains dans un tampon sans passer par des sémaphores. L'appel de la fonction du noyau `pipe()` crée un tampon de données, un tube, dans lequel deux processus pourront venir respectivement lire et écrire. La fonction fournit, d'autre part, deux descripteurs, analogues aux descripteurs de fichiers, qui serviront de référence pour les opérations de lecture et d'écriture. Le système réalise la synchronisation de ces opérations de manière interne.

Les deux processus communicants doivent partager les mêmes descripteurs obtenus par `pipe()`. Il est donc commode qu'ils aient un ancêtre commun car les descripteurs de fichiers sont hérités. Dans l'exemple suivant, deux processus, un père et un fils communiquent par l'intermédiaire d'un tube. Le père écrit les lettres de l'alphabet que lit le fils.

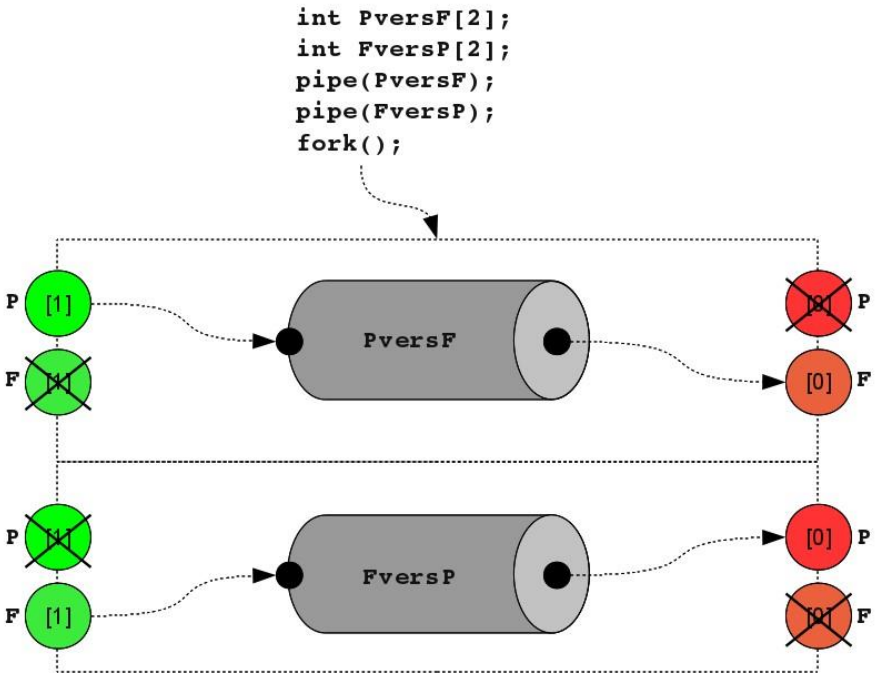
2.12.6.3 Caractéristiques

Les caractéristiques d'un tube :

- intra-machine
- communication unidirectionnelle
- communication synchrone
- communication en mode flux (stream)
- remplissage et vidage en mode FIFO
- limité en taille

2.12.6.4 Communication bidirectionnelle

Les tubes étant des systèmes de communication unidirectionnels, il faut créer deux tubes et les employer dans le sens opposé si l'on souhaite réaliser une communication bidirectionnelle entre 2 processus.



La création de ces deux tubes est sous la responsabilité du processus père. Ensuite, la primive fork() réalisant une copie des données dans le processus fils, il suffit pour le père et le fils de fermer les extrémités de tubes qu'ils n'utilisent pas.

2.12.6.5 Les tubes nommés

Un tube nommé est simplement un nœud dans le système de fichiers. Le concept de tube a été étendu pour disposer d'un nom dans ce dernier. Ce moyen de communication, disposant d'une représentation dans le système de fichier, peut être utilisé par des processus indépendants.

- La création d'un tube nommé se faitl'aide de la fonction mkfifo().
- La suppression d'un tube nommé s'effectue avec unlink().
- Une fois le nœud créé, on peut l'ouvrir avec open() avec les restrictions dues au mode d'accès.
- Si le tube est ouvert en lecture, open() est bloquante tant qu'un autre processus ne l'a pas ouvert en écriture. Symétriquement, une ouverture en écriture est bloquante jusqu' à ce que le tube soit ouvert en lecture.
- Il est possible d'utiliser la fonction fdopen() qui retourne un flux associé au descripteur passé en paramètre. On peut ensuite utiliser les fonctions fread(), fwrite(), fscanf() ou fprintf() ...

2.12.7 Les signaux

2.12.7.1 Définition

Un signal peut être imaginé comme une sorte d'impulsion qui oblige le processus cible à prendre immédiatement une mesure spécifique.

Le rôle des signaux est de permettre aux processus de communiquer. Ils peuvent par exemple :

- réveiller un processus
- arrêter un processus
- avertir un processus d'un événement

2.12.7.2 Envoi et réception d'un signal

A la réception d'un signal, le processus peut :

- l'ignorer
- déclencher l'action prévue par défaut sur le système (souvent la mort du processus)
- déclencher un traitement spécial appelé handler (gestionnaire du signal)

Sous Linux, la primitive signal() permet d'associer un traitement à la réception d'un signal d'interruption. Une autre primitive, sigaction(), permet de faire la même chose et présente l'avantage de définir précisément le comportement désiré et de ne pas poser de problème de compatibilité.

L'attente d'un signal démontre tout l'intérêt du multitâche. En effet, au lieu de consommer des ressources CPU inutilement en testant la présence d'un événement dans une boucle while, on place le processus en sommeil (sleep) et le processeur est mis alors à la disposition d'autres tâches.

2.12.8 Les sockets

Une architecture *client/serveur* implique un ensemble d'au moins deux processus qui dialoguent :

- le *client* qui demande un *service*,
- le *serveur* qui fournit le *service*.

Le dialogue peut se faire de différentes façons (par des sockets, des pipes, à travers une liaison série...).

Berkeley Sockets Interface ou simplement **sockets**, est un ensemble normalisé de fonctions de communication lancé par l'université de Berkeley au début des années 1980 pour leur Berkeley Software Distribution (abr. *BSD*). 30 ans après son lancement, cette interface de programmation est proposée dans quasiment tous les langages de programmation populaires (Java, C#, C++, …).

La notion sur laquelle est construite cette interface sont les *sockets* (en français « interfaces de connexion ») par lesquelles une application peut se brancher à un réseau et communiquer ainsi avec une autre application branchée depuis un autre ordinateur.

2.12.8.1 Fonctionnalité

Un *socket* représente une *prise* par laquelle une application peut envoyer et recevoir des données. Cette prise permet à l'application de se brancher sur un réseau et communiquer avec d'autres applications qui y sont branchées. Les informations écrites sur une prise depuis une machine sont lues sur la prise d'une autre machine, et inversement. Il existe différents modèles de prises, en fonction des protocoles réseau; le plus fréquent est les *socket* TCP/IP. La première interface de programmation (anglais *API* pour *application programming interface*) mettant en œuvre les *sockets* a été développée par l'université de Berkeley pour leur Unix, dans les années 1980.

La fonction `socket` de cette API sert à créer un certain type de prise. Le type de prise sera choisi en fonction de la technologie de communication à utiliser (par exemple TCP/IP). L'API permet à un logiciel serveur de servir plusieurs clients simultanément. Sur les systèmes d'exploitation Unix le programme serveur utilisera la fonction `fork` pour chaque demande d'un client[4].

Une connexion est établie entre le client et le serveur en vue de permettre la communication. La fonction `connect` permet à un client de demander la connexion à un serveur, et la fonction `accept` permet à un serveur d'accepter cette connexion. Le programme serveur utilisera préalablement la fonction `listen` pour informer le logiciel sous-jacent qu'il est prêt à recevoir des connexions. Une fonction `close` permet de terminer la connexion. Lorsqu'un des deux interlocuteurs termine la connexion, l'autre est immédiatement avisé.

Une fois la connexion établie, les fonctions `send` et `recv` servent respectivement à envoyer et à recevoir des informations. Une fonction auxiliaire `gethostbyname` permet d'obtenir l'adresse IP d'une machine en interrogeant le DNS, adresse qui sera utilisée par d'autres fonctions de l'API.

Chaque socket possède un *type* et un ou plusieurs processus qui lui sont associés. Il est également caractérisé par le *domaine de communication* dans lequel il se trouve. Ce dernier est une abstraction qui permet de regrouper les processus ayant des propriétés communes et

communiquant par l'intermédiaire de sockets. Normalement, un socket ne peut échanger des données qu'avec un socket se trouvant dans le même domaine de communication.

La communication inter-processus de 4.3BSD supportait trois domaines de communication :

- le *domaine Unix* dans lequel deux processus se trouvant sur la même station Unix uniquement peuvent communiquer;
- le *domaine Internet* pour les processus utilisant le protocole TCP/IP pour communiquer entre eux ;
- le *domaine NS* pour les processus échangeant des données en utilisant le protocole standard de Xerox.

2.12.8.2 Principes de base

- le *serveur* attend la connexion d'un *client* pour lui fournir un *service*.
- un *client* se connecte à un *serveur*, reçoit un *service*, puis se déconnecte du *serveur*.
- le *serveur* peut servir simultanément (ou séquentiellement) plusieurs *clients*.
- la *connexion* établit un *canal* (d'échange) entre les deux processus.
- la *déconnexion* supprime ce *canal*.
- les échanges entre les deux processus respectent des règles qui forment un *protocole*.
- ce *protocole* définit une *session* entre deux processus.
- un processus a une *adresse* (réseau) (adresse IP sous TCP/IP).
- un service a un *nom* (numéro de port sous TCP/IP).

2.12.8.3 Les types de sockets

Les différents types de sockets dépendent de quelques propriétés visibles par le programmeur. Rien n'empêche deux sockets de types différents de communiquer entre eux si le protocole utilisé le supporte — même si les processus sont supposés communiquer uniquement par des sockets de même type.

Il existe généralement quatre types de sockets :

- Un socket *stream* permet une communication bidirectionnelle, sûre, séquencée et un flux de données sans duplication pouvant entraîner une fragmentation des paquets transmis. Dans le domaine Internet, il s'agit du protocole TCP.
- Un socket *datagram* permet une communication bidirectionnelle qui n'est pas séquencée, pas sûre, et peut éventuellement entraîner une duplication des données. Un processus utilisant ce type de socket peut donc recevoir les données dans un ordre différent de l'ordre de départ. Dans le domaine Internet, il s'agit du protocole UDP.
- Un socket *raw* permet d'accéder au contenu brut des paquets de données. Les *sockets raw* ne sont pas destinés aux utilisateurs courants — seul l'utilisateur root peut y avoir accès sur la plupart des systèmes UNIX® — et sont utilisés par exemple pour analyser le trafic d'un réseau.
- Un socket *sequenced packet*, qui ressemble à un socket *stream* sauf qu'il n'utilise pas de fragmentations de paquets.

2.12.8.4 Les sockets internet

Les sockets du domaine *Internet* sont utilisés pour la communication bidirectionnelle entre des processus qui sont sur des machines différentes d'un réseau IP. Ainsi, un socket du domaine *Internet* désigne un des nœuds d'un flux de données à travers un réseau IP tel qu'Internet.

Dans ce type de socket, une *adresse socket* est la combinaison d'une adresse IP et d'un port (lequel est associé à un processus) formant une seule et unique entité.

Un socket Internet se caractérise par la combinaison des éléments suivants :

- Adresse du socket local : adresse IP locale et numéro de port.
- Adresse du socket distant : uniquement pour les sockets TCP déjà établis.
- Protocole : un protocole de la couche transport (TCP, UDP).

2.12.9 L'API Win32 de Microsoft

L'API Win32 de Microsoft propose différents moyens pour faire communiquer plusieurs processus ou threads.

L'API Win32 de Microsoft propose différents moyens pour coordonner l'exécution de plusieurs threads. Les mécanismes fournis sont :

- les exclusions mutuelles
- les sections critiques
- les sémaphores

La technique officielle préconisée par Microsoft pour partager de la mémoire est d'utiliser les différentes fonctions pour accéder un fichier mappé (FileMapping).

- Un des processus ou threads doit créer en espace de mémoire partagée avec un chier mappé en utilisant CreateFileMapping().
- Les autres processus ou threads peuvent accéder à la mémoire partagée en l'ouvrant avec la fonction OpenFileMapping().
- Pour avoir un pointeur utilisable pour accéder à une zone de la mémoire, il faut utiliser MapViewOfFile(). Après utilisation, il faut appeler UnmapViewOfFile() pour détacher le handle.
- Pour libérer la mémoire, il suffit d'appeler CloseHandle() sur le handle obtenu.

L'accès simultané à la mémoire partagée doit être sécurisé par l'utilisation d'un mutex par exemple.