**RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE**
**MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR**
**ET DE LA RECHERCHE SCIENTIFIQUE**
**UNIVERSITÉ IBN KHALDOUN-TIARET**

**FACULTÉ DES MATHÉMATIQUES**
**ET DE L'INFORMATIQUE**

# Systèmes d'Exploitation 2 : de la Théorie à la Pratique

**Polycopié pédagogique**
**destiné aux étudiants de la :**

## $3^{eme}$ Année Licence

Spécialité : Systèmes Informatiques

**Réalisé par :**

Dr. **Boudjemaa BOUDAA**

**Expertisé par :**

Pr. **Mostefa BALARBI** - Université de Tiaret.

Dr. **Mohammed SALEM** - Université de Mascara.

**2021-2022**

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

**BOUDAA Boudjemaa**
Maître de conférences

**Département Informatique**
**Faculté des Mathématiques et Informatique**
**Université Ibn Khaldoun de Tiaret**


**Polycopié pédagogique**

# Systèmes d'Exploitation 2: de la Théorie à la Pratique


**07/12/2021**

## Avant-propos

Ce polycopié de cours est le fruit de plus de 10 ans d'expérience dans l'enseignement du module « Systèmes d'exploitation 2 » en faveur les étudiants de la 3eme année Licence Informatique (spécialité Systèmes Informatiques) au niveau du département informatique de l'université de Tiaret. Il contient huit (08) chapitres répartis en deux parties. La première détaille les notions et les concepts sous-jacentes des systèmes d'exploitation en présentant les différentes techniques et stratégies utilisées pour la communication et synchronisation des processus/threads notamment pour les modèles des problèmes classiques tel que : lecteur/rédacteur et producteur/consommateur. Apres présenter le système Linux, la deuxième partie met en pratique ces techniques par la programmation des threads et les différents mécanismes d'exclusion mutuelle en langage C/Java sur des machines Linux.

Apres citer les objectifs visés, chaque chapitre commence par une introduction avant d'aborder les thèmes y afférent. Et il se termine par une série de questions et les références bibliographiques utilisée. Des exercices pratiques supplémentaires avec leurs solutions font l'objet du dernier chapitre dans ce polycopié.

Ce document a été élaboré en se basant sur des célèbres références dans le domaine des systèmes d'exploitation, tel que les livres de Crocus, Krakowiak et Tanenbaum, et d'autres sources intéressantes et disponibles sur Internet.

Le présent polycopié pourrait servir un grand public de lecteurs :

- En premier lieu, les étudiants en Licence informatique vont y trouver les connaissances théoriques et les bonnes pratiques nécessaires pour apprendre la discipline des systèmes d'exploitation 2.
- Egalement, les enseignants peuvent en bénéficier pour préparer soit des cours ou des fiches de travaux pratiques (TP) à travers les exercices et les solutions proposés.
- Enfin, ce polycopié s'adresse à toute personne intéressée par la gestion des processus, threads et les mécanismes d'exclusion mutuelle.

Bonne lecture.

**Tiaret, le 07/12/2021**

# Sommaire


**CHAPITRE 1: CONCEPTS DE BASE ..... 8**

1. INTRODUCTION ..... 8
2. NOTION DE SYSTEME D'EXPLOITATION ..... 8
3. PROCESSUS ET MULTI-THREADING ..... 9
   - *3.1 Contexte d'exécution d'un programme ..... 9*
   - *3.2 Relation entre Programme et processus ..... 9*
     - 3.2.1 Exemple illustratif ..... 10
   - *3.3 Processus ..... 10*
     - 3.3.1 Différentes actions possibles d'un processus ..... 10
     - 3.3.2 Différents états d'un processus ..... 11
     - 3.3.3 Que peut faire un processus ? ..... 11
   - *3.4 Le multithreading ..... 11*
     - 3.4.1 Qu'est-ce qu'un thread ..... 11
     - 3.4.2 Différences et similitudes entre threads et processus ..... 12
4. NOTION DE RESSOURCE ..... 12
   - *4.1 Ressource partageable à un point d'accès ..... 12*
   - *4.2 Ressource partageable à n points d'accès ..... 13*
5. RELATION ENTRE LES PROCESSUS ..... 13
   - *5.1 Exemple de relation de concurrence (compétition) ..... 13*
   - *5.2 Exemple de relation de coopération ..... 13*
6. QUESTIONS ..... 14
7. REFERENCES ..... 14

**CHAPITRE 2: SYNCHRONISATION DES PROCESSUS ..... 16**

1. INTRODUCTION ..... 16
2. DEFINITION DE LA SYNCHRONISATION ..... 16
3. NOTION DE SECTION CRITIQUE ..... 16
   - *3.1 Définitions ..... 16*
   - *3.2 Exemple ..... 17*
4. MODELES DE PROBLEMES CLASSIQUES DE SYNCHRONISATION ..... 17
   - *4.1 Modèle du Lecteur-Rédacteur ..... 17*
   - *4.2 Modèle du Producteur-Consommateur ..... 18*
   - *4.3 Modèle du Diner des Philosophes ..... 18*
   - *4.4 Modèle du Coiffeur Endormi ..... 19*
5. MECANISMES DE L'EXCLUSION MUTUELLE ..... 20
   - *5.1 L'exclusion mutuelle avec attente active ..... 20*
     - 5.1.1 Masquage des interruptions ..... 20
     - 5.1.2 Variables de verrouillage ..... 21
     - 5.1.3 Variables d'alternance ..... 23
   - *5.2 L'exclusion mutuelle avec attente passive ..... 25*
     - 5.2.1 Sémaphores ..... 25
     - 5.2.2 Moniteurs ..... 28
     - 5.2.3 Régions critiques ..... 29
     - 5.2.4 Expressions de chemins ..... 32
   - *5.3 Synchronisation des threads en Java ..... 33*
     - 5.3.1 Création de threads avec Java ..... 33
     - 5.3.2 Méthodes de gestion des threads ..... 33

5.3.3 Synchronisation et Exclusion Mutuelle ..... 34
5.3.4 Cas d'étude : Producteur-Consommateur en Java ..... 35
5.3.5 Ordonnancement et priorité ..... 37
6. QUESTIONS ..... 37
7. REFERENCES ..... 38

**CHAPITRE 3: COMMUNICATION INTERPROCESSUS ..... 40**

1. INTRODUCTION ..... 40
2. COMMUNICATION INTERPROCESSUS ..... 40
3. STRATEGIES DE LA COMMUNICATION INTERPROCESSUS ..... 41
*3.1. Partage de variables ..... 41*
3.1.1 Modèle producteur-consommateur ..... 41
*3.2. Boites aux lettres ..... 42*
*3.3. Echange de messages (modèle du client/ serveur) ..... 43*
4. QUESTIONS ..... 44
5. REFERENCES ..... 44

**CHAPITRE 4: L'INTERBLOCAGE ..... 46**

1. INTRODUCTION ..... 46
2. DEFINITION DE L'INTERBLOCAGE ..... 46
*2.1 Apparition d'interblocages ..... 47*
*2.2 Modélisation des interblocages ..... 47*
3. GRAPHE D'ALLOCATION DES RESSOURCES ..... 48
4. METHODES DE GESTION DES INTERBLOCAGES ..... 50
*4.1 Prévention des interblocages ..... 51*
4.1.1 Exclusion mutuelle ..... 51
4.1.2 Détention et attente ..... 51
4.1.3 Pas de réquisition ..... 51
4.1.4 Attente circulaire ..... 51
*4.2 Evitement des interblocages ..... 52*
4.2.1 Principe de fonctionnement ..... 52
4.2.2 Etat sain ..... 53
4.2.3 Algorithme du Banquier ..... 54
*4.3 Détection et guérison des interblocages ..... 57*
4.3.1 Détection de l'interblocage ..... 57
4.3.2 Guérison de l'interblocage ..... 60
*4.4 Ignorance de l'interblocage : « Politique de l'autruche » ..... 60*
5. QUESTIONS ..... 60
6. REFERENCES ..... 61

**CHAPITRE 5: GENERALITES SUR LINUX ..... 64**

1. INTRODUCTION ..... 64
2. UNIX ..... 64
*2.1. Historique ..... 64*
*2.2. Caractéristiques d'Unix ..... 64*
3. LINUX ..... 65
*3.1. Historique ..... 65*
*3.2. Logiciel libre ..... 65*
4. DISTRIBUTIONS DE LINUX ..... 65
*4.1. Exemples des distributions ..... 66*
*4.2. Domaines d'utilisation de Linux ..... 66*
5. ACCES A LA MACHINE SOUS LINUX ..... 66
*5.1. Position du noyau Linux dans une distribution ..... 66*

*5.2. Le Shell* ..... *67*
6. QUESTIONS ..... 68
7. REFERENCES ..... 68

**CHAPITRE 6: PRISE EN MAIN AVEC LINUX ..... 70**

1. INTRODUCTION ..... 70
2. ARBORESCENCE DU SYSTEME DE FICHIERS ..... 70
3. PRINCIPALES COMMANDES DE BASE ..... 71
4. LIENS DE FICHIERS ..... 75
*4.1. Liens symboliques* ..... *75*
*4.2. Liens physiques* ..... *75*
*4.3. Comment créer un lien* ..... *76*
5. QUESTIONS ..... 76
6. REFERENCES ..... 76

**CHAPITRE 7: TRAVAUX PRATIQUES ..... 78**

1. FICHE TP N° 01 « LINUX : RAPPEL SUR LES COMMANDES DE BASE ET DROITS D'ACCES » ..... 78
2. FICHE TP N° 02 « PROGRAMMATION EN LANGAGE C SOUS LINUX » ..... 80
3. FICHE TP N° 03 « MANIPULATION DES PROCESSUS ET THREADS EN LANGAGE C » ..... 82
4. FICHE TP N° 04 « THREADS ET SEMAPHORES AVEC API POSIX » ..... 85
5. FICHE TP N° 05 « GESTION DES THREADS EN LANGAGE JAVA » ..... 88
6. FICHE TP N° 06 « SYNCHRONISATION DES THREADS AVEC LES MONITEURS EN JAVA » ..... 91
7. FICHE TP N° 07 « ETUDE DE CAS : LES THREADS POUR GERER UN RESTAURANT » ..... 92

**CHAPITRE 8: EXERCICES PRATIQUES CORRIGES ..... 95**

1. ENONCES DES EXERCICES ..... 95
2. CORRIGES DES EXERCICES ..... 97

# 1ere Partie: la Théorié

Chapitre I

# Concepts de Base

## Objectifs

- Comprendre le  role du système d'exploitation au sein d'un machine
- prendre connaissance de tous les concpets de base relatives aux systèmes d'exploitation.

## Themes couverts

- Rappels sur la notion de SE
- Notions de programme, processus, thread et ressource partagée

# Chapitre 1: Concepts de Base

## 1. Introduction

Un système d'exploitation (*Operating System* ou OS) est un ensemble de programmes spécialisés qui permet l'utilisation des ressources matérielles d'un ou plusieurs ordinateurs. Il assure le démarrage (*Boot*) de l'ordinateur et l'exécution des logiciels applicatifs. Il remplit deux fonctions majeures : d'une part, la gestion des ressources matérielles (la mémoire, le processeur et les périphériques), en répartissant leur utilisation entre les différents logiciels ; d'autre part, la fourniture de services aux applications, en offrant une interface de plus haut niveau (comme une machine virtuelle) que celle de la machine physique. Ce chapitre présente les principaux notions et concepts de base liés aux systèmes d'exploitation.

## 2. Notion de système d'exploitation

Un ordinateur est composé de matériel (hardware) et de logiciel (software). Cet ensemble est à la disposition de un ou plusieurs utilisateurs. Il est donc nécessaire que quelque chose dans l'ordinateur permette la communication entre l'homme et la machine. Cette entité doit assurer une grande souplesse dans l'interface et doit permettre d'accéder à toutes les fonctionnalités de la machine. Cette entité douée d'une certaine intelligence de communication se dénomme " la machine virtuelle " (voir Figure 1.1). Elle est la réunion du matériel et du système d'exploitation (SE).

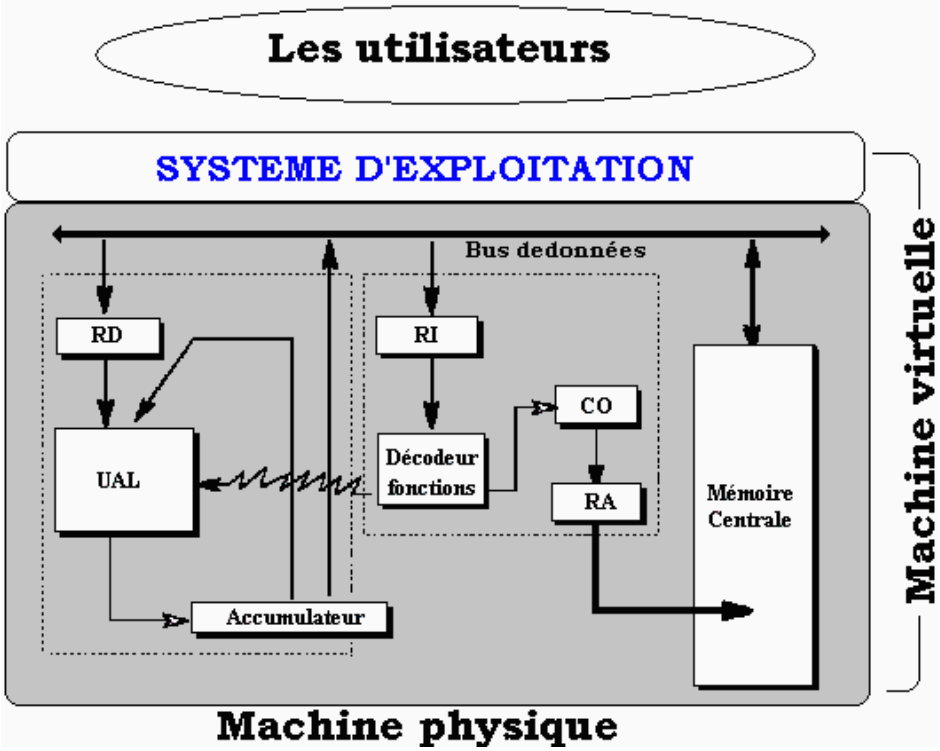



**Figure 1.1 :** Structure de la Machine virtuelle

Le système d'exploitation d'un ordinateur est chargé d'assurer les fonctionnalités de communication et d'interface avec l'utilisateur. Un SE est un logiciel dont l'objectif est la gestion de toutes les ressources de l'ordinateur :

- mémoires,
- fichiers,
- périphériques,
- entrée-sortie,
- interruptions, synchronisation...

Un système d'exploitation n'est pas un logiciel unique mais plutôt une famille de logiciels. Une partie de ces logiciels réside en mémoire centrale (nommée **résident** ou **superviseur**), le reste est stocké en mémoire auxiliaire (disques durs par exemple).

Afin d'assurer une bonne liaison entre les divers logiciels de cette famille, la cohérence de le SE est généralement organisée à travers des tables d'interfaces architecturées en couches de programmation (niveaux abstraits de liaison). La principale tâche du superviseur est de gérer le contrôle des échanges d'informations entre les diverses couches du SE.

## 3. Processus et multi-threading

### 3.1 Contexte d'exécution d'un programme

Lorsqu'un programme qui a été traduit en instructions machines s'exécute, le processeur central lui fournit toutes ses ressources (registres internes, place en mémoire centrale, données, code,…), nous nommerons cet ensemble de ressources mises à disposition d'un programme son contexte d'exécution.

### 3.2 Relation entre Programme et processus

Nous appelons en première analyse, processus l'image en mémoire centrale d'un programme s'exécutant avec son contexte d'exécution. Le processus est donc une abstraction synthétique d'un programme en cours d'exécution et d'une partie de l'état du processeur et de la mémoire.

Lorsque l'on fait exécuter plusieurs programmes "en même temps", nous savons qu'en fait la simultanéité n'est pas réelle (voir Figure 1.2). Le processeur passe cycliquement une partie de son temps (quelques millisecondes) à exécuter séquentiellement une tranche d'instructions de chacun des programmes selon une logique qui lui est propre, donnant ainsi l'illusion que tous les programmes sont traités en même temps parce que la durée de l'exécution d'une tranche d'instruction est plus rapide que notre attention consciente.

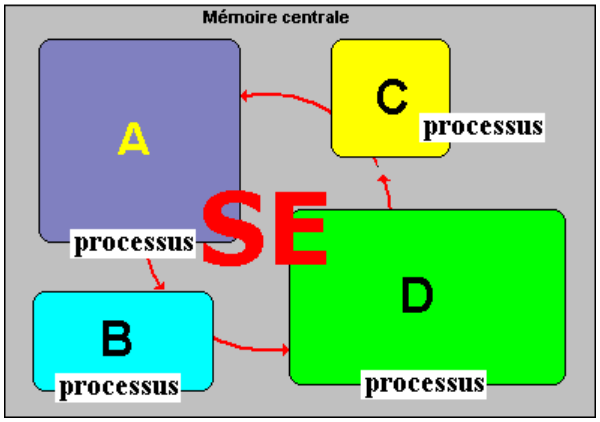



**Figure 1.2 :** Exemple sur la gestion de 4 processus par le système d'exploitation

### *3.2.1 Exemple illustratif*

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

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

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

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

De même que l'informaticien aura un seul exemplaire de la recette pour les deux processus, de même dans une situation analogue en machine, le programme lui-même pourra se trouver en un seul exemplaire en mémoire, pourvu que d'une part le code ne se modifie pas lui-même, d'autre part qu'il permette d'accéder à des zones de données différentes suivant que les instructions sont exécutées pour le compte de l'un ou l'autre des processus. On dit alors que le programme est *réentrant*.

## 3.3 Processus

Nous donnons la définition précise de processus proposée par **A.Tanenbaum**, spécialiste des systèmes d'exploitation : « **c'est un programme qui s'exécute et qui possède son propre espace mémoire, ses registres, ses piles, ses variables et son propre processeur virtuel (simulé en multi-programmation par la commutation entre processus effectuée par le processeur unique) »**.

Un processus a donc une vie propre et une existence éphémère, contrairement au programme qui lui est physiquement présent sur le disque dur. Durant sa vie, un processus peut agir de différentes manières possibles, il peut se trouver dans différents états, enfin il peut travailler avec d'autres processus présent en même temps que lui.

### *3.3.1 Différentes actions possibles d'un processus*

- Un processus est **créé**.
- Un processus est **détruit**.
- Un processus **s'exécute** (il a le contrôle du processeur central et exécute les actions du programme dont il est l'image en mémoire centrale).
- Un processus est **bloqué** (il est en attente d'une information).
- Un processus est **passif** (il n'a plus le contrôle du processeur central).

### *3.3.2 Différents états d'un processus*

- **Etat actif** : le processus contrôle le processeur central et s'exécute.
- **Etat passif** : le processus est temporairement suspendu et mis en attente, le processeur central travaille alors avec un autre processus.
- **Etat bloqué** : le processus est suspendu toutefois le processeur central ne peut pas le réactiver tant que l'information attendue par le processus ne lui est pas parvenue.

### *3.3.3 Que peut faire un processus ?*

- Il peut créer d'autre processus
- Il travaille et communique avec d'autres processus (notion de synchronisation et de messages entre processus)
- Il peut posséder une ressource à titre exclusif ou bien la partager avec d'autre processus.

C'est le rôle du SE que d'assurer la gestion complète de la création, de la destruction, des transitions d'états d'un processus. C'est toujours au SE d'allouer un espace mémoire utile au travail de chaque processus. C'est encore le SE qui assure la synchronisation et la messagerie inter-processus.

Le système d'exploitation implémente cette gestion des processus à travers une table des processus qui contient une entrée par processus créé par le système sous forme d'un bloc de contrôle du processus (PCB ou **P**rocess **C**ontrol **B**lock). Le PCB contient lui-même toutes les informations de contexte du processus, plus des informations sur l'état du processus.

Lorsque la politique de gestion du SE prévoit que le processus $P_k$ est réactivable (c'est au tour de $P_k$ de s'exécuter), le SE va consulter le PCB de $P_k$ dans la table des processus et restaure ou non l'activation de $P_k$ selon son état (par exemple si $P_k$ est bloqué, le système ne l'active pas).

Afin de séparer les tâches d'un processus, il a été mis en place la notion de processus léger (ou thread).

## 3.4 Le multithreading

Nous pouvons voir le multithreading comme un changement de facteur d'échelle dans le fonctionnement de la multi-programmation.

En fait, chaque processus peut lui-même fonctionner comme le système d'exploitation en lançant des sous-tâches internes au processus et par là même reproduire le fonctionnement de la multi-programmation. Ces sous-tâches sont nommées "flux d'exécution" "processus légers" ou **Threads.** La majorité des systèmes d'exploitation (Windows, Unix, MacOs,...) supportent le **Multithreading.**

### *3.4.1 Qu'est-ce qu'un thread*

Un thread ou fil (d'exécution) ou tâche (autres appellations connues : processus léger, fil d'instruction, processus allégé, exétron, …) 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'exécution.

Un processus travaille et gère, pendant le quantum de temps qui lui est alloué, des ressources et exécute des actions sur et avec ces ressources. Un thread constitue la partie exécution d'un processus alliée à un minimum de variables qui sont propres au thread.

Les threads situés dans un même processus partagent les mêmes variables générales de données et les autres ressources allouées au processus englobant. Un thread possède en propre un contexte d'exécution (registres du processeur, code, données).

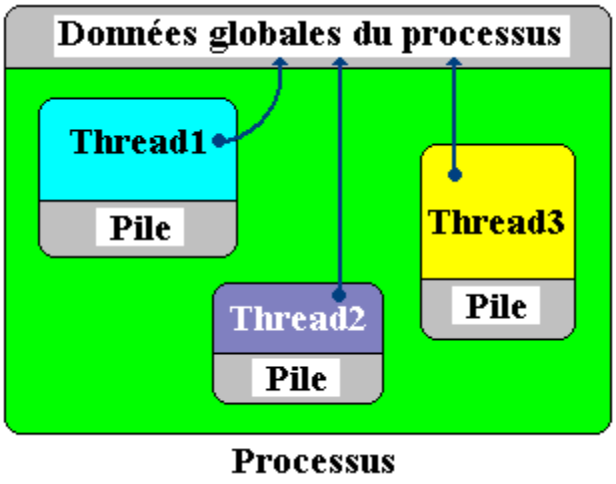



**Figure 1.3 :** Un processus avec trois threads.

Le processus applique au niveau local une multi-programmation interne qui est nommée le multithreading. La différence fondamentale entre la multi-programmation et nommée le multithreading se situe dans l'indépendance qui existe entre les processus, alors que les threads sont liés à minima par le fait qu'ils partagent les même données globales (celles du processus qui les contient).

Les langages de programmation récents comme C++, Delphi, Java et C# disposent chacun de classes permettant d'écrire et d'utiliser des threads.

### *3.4.2 Différences et similitudes entre threads et processus*

- La communication entre les threads est **plus rapide** que la communication entre les processus.
- Les Threads possèdent les mêmes états que les processus.
- Deux processus peuvent travailler sur une même donnée (un tableau par exemple) en lecture et en écriture, dans une situation de **concurrence** dans laquelle le résultat final de l'exécution dépend de l'ordre dans lequel les lectures et écritures ont lieu, il en est de même pour les threads.

## 4. Notion de ressource

Un système informatique possède un nombre fini de ressources qui doivent être distribuées sur les processus concurrents. Les ressources sont des entités nécessaires à l'exécution d'un processus. Ces ressources peuvent être physiques (processeur, mémoire imprimante, etc...) ou logique (données, programmes). On distingue 02 catégories de ressources

### 4.1 Ressource partageable à un point d'accès

Une ressource est dite partageable à **un seul point d'accès (ou ressource critique)**, si elle peut être attribuée à un seul processus à la fois. Par exemple : Processeur, Imprimante.

### 4.2 Ressource partageable à n points d'accès

Une ressource est dite partageable à un n points d'accès, si elle peut être attribuée à plusieurs (n) processus à la fois. Par exemple : Bloc mémoire, etc…

## 5. Relation entre les processus

Ces relations peuvent être soit des relations de concurrence (compétition), soit des relations de coopération. Le premier type de relation se présente lorsque les processus mis en jeu partagent des ressources physiques ou logiques. Par contre, le second type de relation intervient lorsque les processus participent à un traitement global. En d'autres termes, ces relations permettent de synchroniser et faire communiquer les processus composant une application parallèle.

### 5.1 Exemple de relation de concurrence (compétition)

Considérons un système de réservation de places dans une salle de spectacle, un avion, un train, ...etc. Il faut gérer un compteur des places disponibles. Les transactions issues des différents points de réservation peuvent conduire à la modification de cette variable via la procédure de test et réservation :

```
int npDispo                 // variable partagée (nombre de places disponibles)
bool reserver (int p) {                          /* p : nombre de places à réserver */
                                if ( p > npDispo) return false;          /* test */
                                nDispo = npDispo - p;   /* décrémentation */
                                return true;
                            }
```

Supposons que l'on exécute les deux réservations suivantes en parallèle :

{npDispo = 12} { reserver(10) || reserver(5)} {npDisp = -3 alors il y a un surbooking ! }

### 5.2 Exemple de relation de coopération

Considérons une application de comptage de véhicules sur une route. Cette application se compose de deux activités cycliques :

– d'une part, un captage des événements « passage » de véhicule;
– d'autre part, un affichage périodique (toutes les10 mns, toutes les1/2 heures,...) du nombre de véhicules passés dans la période fixée.

Ce système parallèle se décrit sous la forme de deux processus partageant une variable globale compteur:

int cpt = 0 ; // variable partagée (compteur)

```
process Capteur()
        {
            while (true)
            {
attendre(passage);
              cpt++ ;
            }
        }
```

```
process Consignateur()
        {
            while (true)
            { attendre(delai) ;
              printf("nbre passés = %i",cpt) ;
              cpt = 0 ;
            }
        }
```

## 6. Questions

**a)** Par des exemples, donner d'autres définitions des concepts suivantes : Programme, Processus, Thread, Ressource partagée?
**b)** Proposer une solution d'un problème de la vie réelle en utilisant le multithreading.
**c)** A l'aide des programmes écrits en langage java ou C, exprimer les relations qui peuvent exister entre les processus.

## 7. Références


- Tanenbaum, « Modern operating systems », third edition, Pearson, 2014
- Tanenbaum, « Systèmes d'exploitation », Dunod, 1994.
- Crocus, « Systèmes d'exploitation des ordinateurs », Dunod,1993.
- Sacha Krakowiak, « Principes des systèmes d'exploitation des ordinateurs », Dunod, 1993
- https://rmdiscala.developpez.com/cours/LesChapitres.html/Cours1/Chap1.6.htm
- http://www-int.impmc.upmc.fr/impmc/Enseignement/ye/informatique/systemes/chap1/

Chapitre II

# Synchronisation des Processus

## Objectifs

- Comprendre les problèmes de la section critiques
- Synchronisation de la gestion des processus par les mechanismes d'exclusion mutuelle (solutions de l'attente active et passive)

## Themes couverts

- Problème d'accès concurrent à des ressources et sections critiques
  - Problème de l'exclusion mutuelle
- Outils de synchronisation :
  - Evénements, Verrous
  - Sémaphores
  - Moniteurs
  - Régions critiques
  - Expressions de chemins

# Chapitre 2: Synchronisation des Processus

## 1. Introduction

La multiprogrammation permet l'exécution des applications parallèles. Une application parallèle est définit comme un ensemble de processus séquentiels qui s'exécutent en parallèle (parallélisme réel cas des machines multiprocesseurs ou pseudo parallélisme dans le cas des machines monoprocesseur). Chaque processus de cet ensemble réalise une tâche bien précise de l'application en question, et il est lié aux autres processus par des relations. Néanmoins, ces processus en relation doivent d'être synchronisés pour une meilleure gestion des traitements exécutés. Dans cet objectif, le présent chapitre aborde ce problème de la nécessité de synchronisation en présentant toutes les solutions possibles.

## 2. Définition de la synchronisation

La synchronisation est l'activité qui consiste à construire des mécanismes permettant à un processus actif de changer son état (Actif/Bloqué) ou de changer l'état d'un autre processus (Bloqué/Actif). La synchronisation de processus cherche par exemple à empêcher des programmes d'exécuter la même partie de code en même temps, ou au contraire forcer l'exécution de deux parties de code en même temps. Dans la première hypothèse, le premier processus bloque l'accès à la section critique avant d'y entrer et libère l'accès après en être sorti. Ce mécanisme peut être implémenté de multiples manières comme nous allons voir ci-après.

## 3. Notion de Section critique

Chaque processus veut exécuter une section critique (séquence d'instructions manipulant une ressource critique). En effet, cette section critique contient des objets (variables) partagés entre les différents processus qui utilisent cette ressource critique. Si la gestion de ces objets n'est pas faite d'une manière judicieuse, elle peut entraîner facilement des incohérences.

### 3.1 Définitions

- Une ressource est dite **ressource critique** lorsque des accès concurrents à cette ressource peuvent résulter un état incohérent.
- Une **section critique** est une section de programme manipulant une ressource critique.
- Une section de programme est dite **atomique** lorsqu'elle ne peut pas être interrompue par un autre processus manipulant les mêmes ressources critiques.
- Un mécanisme d'**exclusion mutuelle** sert à assurer l'atomicité des sections critiques relatives à une ressource critique (en anglais : mutual exclusion, ou **mutex**)

### 3.2 Exemple

Deux processus P1 et P2 veulent mettre à jour une variable commune **Réserve** (valeur initiale égale à 17). Les 02 processus auront la structure suivante :

| *Processus P1* | *Processus P2* |
|---|---|
| *Debut* | *Debut* |
| ..... | ..... |
| *Reserve := Reserve +3* | *Reserve := Reserve +2* |
| ..... | ..... |
| *Fin* | *Fin* |

Pour simplifier, on suppose que les 02 processus s'exécutent sur deux processeurs différents : (UC1, Acc1), (UC2, Acc2) et les vitesses des 02 processus sont inconnues.

L'instruction d'affectation se traduit en assembleur par les instructions suivantes :

| | |
|---|---|
| *1- Load Acc1, Reserve* | *a- Load Acc2, Reserve* |
| *2- ADD 3* | *b- ADD 2* |
| *3- Store Reserve* | *c- Store Reserve* |

L'exécution de ces actions dans l'ordre **1a 2b 3c** produit un résultat : Reserve = 19 au lieu de 22.

## 4. Modèles de Problèmes Classiques de Synchronisation

### 4.1 Modèle du Lecteur-Rédacteur

Dans ce problème, il s'agit d'un ensemble de processus qui partagent une structure de données tel qu'un fichier. Ce fichier peut être à tout instant accédé par des processus dits lecteurs qui ne font que la consultation de celui-ci, ou par des processus dits rédacteurs qui eux par contre modifient son contenu. La cohérence du fichier sera menacée si ce dernier n'est pas manipulé avec précaution selon un protocole bien déterminé. Plusieurs variantes de protocole pour ce schéma ont été développées, la plus simple est celle qui favorise la catégorie des lecteurs et dont la spécification est la suivante :

- Plusieurs lecteurs peuvent lire en parallèle dans le fichier.
- Un rédacteur doit avoir l'accès exclusif au fichier.
- Lorsque le fichier est libre un lecteur et un rédacteur ont la même priorité.
- Si le fichier est déjà accédé en lecture toute nouvelle demande de lecture sera immédiatement honorée, même si des rédacteurs sont en attente.

La structure des processus lecteurs et rédacteurs est la suivante:

**Processus Lecteur_i**
*Debut*
.....
*(* Demander l'autorisation de lire *)*
DemanderLecture
*LIRE*
*(* Le processus a fini de lire : le signal*)*
*TerminerLecture*
.....
*Fin*

**Processus Rédacteur_i**
*Debut*
.....
*(* Demander l'autorisation d'écrire *)*
*DemanderEcriture*
*ECRIRE*
*(* Le processus a fini d'écrire : le signal*)*
*TerminerEcriture*
.....
*Fin*

La solution complète de ce problème réside dans l'écriture des 04 procédures ci-dessous:

- **DemanderLecture :** Invoquée par un processus lecteur avant que celui-ci commence à lire dans le fichier. Elle autorise le processus appelant à lire si les conditions de lecture sont vérifiées. Dans le cas contraire, le processus appelant sera mis en attente jusqu'à ce que les conditions autorisant celui-ci à lire soient vérifiées.
- **DemanderEcriture** : Invoquée par un rédacteur avant de commencer à écrire. Elle autorise le processus à écrire si les conditions d'écriture sont satisfaites. Dans le cas contraire, il se bloque jusqu'à ce que la demande d'écriture soit acceptée.
- **TerminerLecture :** Invoquée par un processus lecteur dès qu'il a fini de lire. Le dernier lecteur ayant exécuté cette procédure passe le contrôle du fichier à un processus rédacteur en attente.
- **TerminerEcriture :** Invoquée par un rédacteur pour signaler qu'il a fini d'écrire et passe le contrôle du fichier à un ou plusieurs autres processus.

### 4.2 Modèle du Producteur-Consommateur

Ce schéma traduit un modèle de communication couramment rencontré dans le traitement parallèle. Dans ce schéma un processus : le producteur produit des messages (objets) et les dépose dans un tampon (boite aux lettres) pour être retirés et traités par un autre processus dit le consommateur. Le protocole qui implémente ce modèle doit prendre en compte les contraintes suivantes :

- Le tampon est composé de N cases, chacune d'elle peut contenir un message.
- Aucune hypothèse n'est faite sur les vitesses des 02 processus.
- Les messages ne doivent pas être perdus, si le tampon contient N messages non retirés, on ne peut y déposer de message supplémentaire ;
- Un message donné n'est retiré qu'une seule fois après avoir être déposé.
- Une opération impossible (dépôt dans un tampon plein ou retrait depuis un tampon vide) bloque le processus qui tente de l'exécuter.

La structure des deux processus est la suivante :

```
Processus Producteur
Debut
Cycle
    Produire(message)
   Deposer(message)
FinCycle
Fin
```

```
Processus Consommateur
Debut
Cycle
   Retirer(message)
   Consommer(message)
FinCycle
Fin
```

Les procédures Déposer et Retirer doivent implémenter les contraintes mentionnées ci-dessus.

### 4.3 Modèle du Diner des Philosophes

En 1965, Dijkstra a posé et résolu un problème de synchronisation qu'il a appelé le problème du **diner des philosophes**. C'est un problème classique d'accès concurrent qui a servi de référence pour évaluer l'efficacité des différents modèles de primitives de synchronisation proposées au fil du temps.
Cinq philosophes sont assis autour d'une table ronde, et ont chacun devant eux une assiette pleine de spaghetti tellement glissants qu'il faut deux fourchettes pour les manger. Une fourchette est disposée entre deux assiettes consécutives (voir Figure 2.1).

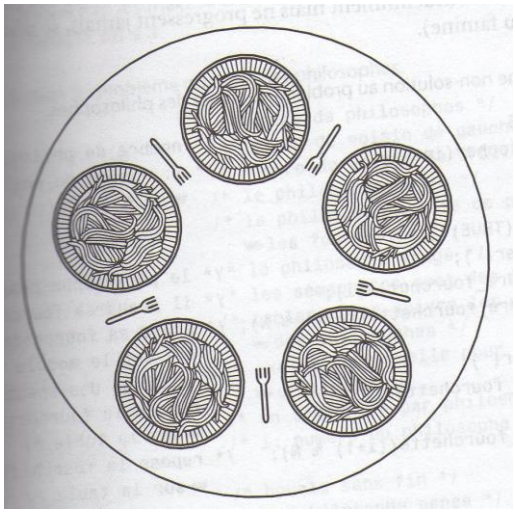


**Figure 2.1** : Le Diner des philosophes

On peut résumer les contraintes de ce problème comme suit :

- Un philosophe passe son temps à penser et à manger.
- Lorsqu'un philosophe a faim, il tente de s'emparer des deux fourchettes qui sont immédiatement à sa gauche et à sa droite, l'une après l'autre, l'ordre n'important pas.
- S'il obtient les deux fourchettes, il mange pendant un certain temps, puis repose les fourchettes et se remet à penser.
- Le problème est que chaque philosophe doit se livrer à ses activités sans jamais être bloqué de façon irrémédiable.

### 4.4 Modèle du Coiffeur Endormi

Une illustration classique du problème de la synchronisation est celui du salon de coiffure. Dans ce dernier, il y a un coiffeur C, un fauteuil F dans lequel se met le client pour être coiffé et N sièges pour attendre.

- S'il n'a pas de clients, le coiffeur C dort dans le fauteuil F.
- Quand un client arrive et que le coiffeur C dort, il le réveille, C se lève. Le client s'assied dans F et se fait coiffer.
- Si un client arrive pendant que le coiffeur travaille :
- si un des N sièges est libre, il s'assied et attend, sinon il ressort. (voir Figure 2.2).

**Figure 2.2** : Le Coiffeur endormi

## 5. Mécanismes de l'exclusion mutuelle

Pour résoudre ces types de problèmes, il faut que les processus qui partagent un objet commun s'excluent mutuellement pour l'usage de cet objet. L'exclusion mutuelle signifie que deux processus ne peuvent être en même temps en section critique. D'une manière concrète il faut que l'exécution de la section critique soit contrôlée par un protocole formé de deux parties (relatives à l'acquisition et à la libération de la section critique **<SC>**). Dans ce cas, chaque processus aura une structure de la forme :

```
Processus P
Debut
.....                                   (* Partie non conflictuelle *)
Entrer_Section_Critique                 (* Acquisition de la SC*)
<Section_Critique>      (*Partie conflictuelle : Accès en exclusion mutuelle à la ressource*)
Quitter_Section_Critique                (*  libération de la SC *)
.....                                   (* Partie non conflictuelle *)
Fin
```

Les mécanismes assurant l'exclusion mutuelle sont de deux types :

1) **Exclusion mutuelle avec attente active** : la procédure entrer_Section_Critique est une boucle dont la condition est un test qui porte sur des variables indiquant la présence ou non d'un processus en Section critique
2) **Exclusion mutuelle avec attente passive *:*** le processus passe dans l'état endormi et libère le processeur, et ne sera réveillé que lorsqu'il sera autorisé à entrer en section critique tout en utilisant les mécanismes tels que : sémaphores, moniteurs et échange de message.

### 5.1 L'exclusion mutuelle avec attente active

Dans cette section, nous examinons diverses propositions permettant d'obtenir une exclusion mutuelle de façon que, pendant qu'un processus est occupé à l'actualisation de la mémoire partagée dans sa section critique, aucun autre processus ne puisse entrer dans sa section critique pour y semer le désordre.

#### *5.1.1 Masquage des interruptions*

Le principe de ce mécanisme est de rendre non observable l'état de la machine durant l'exécution de la section critique (rendre indivisible la SC) en utilisant le Masquage/Démasquage d'interruption. Le masquage d'interruption permet au processeur d'exécuter le code de la section critique jusqu'à sa fin sans être interrompu par un autre processus (par exemple : l'interruption horloge qui permet d'interrompre un processus lorsqu'il a épuisé son temps quantum, serait ignorée). Par contre, le démasquage permet de remettre le processeur dans un état autorisant la prise en compte des interruptions.

**Principe de la solution**

- avant d'entrer en section critique le processus masque les interruptions
- en sortie de section critique le processus démasque les interruptions

```
processus P
  tant que (vrai) faire
        masquer_les_IT;                  // entrée en section critique
        Section_Critique;
       démasquer_les_IT;                 // sortie de section critique
       hors_section_critique;
fait;
```

**Inconvénients de la solution**

- Solution inadaptée dans le cas multiprocesseur : cette solution n'est pas efficace pour les systèmes multiprocesseurs, puisque le masquage des interruptions d'un processeur n'entraîne pas le masquage des autres processeurs qui peuvent toujours entrer en section critique.
- Blocage lorsqu'un processus entre en SC, tous les autres processus sont bloqués, même si la section ne concerne qu'un seul autre processus.
- Solution dangereuse en mode utilisateur : si dans un processus, le programmeur a oublié de démasquer les interruptions, c'est la fin du système.

**Conclusion**

Technique utile dans le noyau, mais pas au niveau utilisateur

### *5.1.2 Variables de verrouillage*

Un verrou est une variable booléenne (ou binaire) partagée qui indique la présence d'un processus en Section Critique.

**Principe de la solution**

- si verrou = faux alors la section critique est libre
- si verrou = vrai alors la section critique est occupée

```
TAS (verrou : booleen ) : booleen          (* Opération indivisible *)
Debut
      Si verrou alors TAS :=vrai
      Sinon   {
                  verrou :=vrai ;
                  TAS :=faux
                  }
    Fsi
Fin
```

L'exclusion mutuelle n'est assurée que si le test et le positionnement du **verrou** est indivisible (sinon le verrou constitue une section critique). Pour cela il faut disposer d'une instruction matérielle (Test And Set ou Swap) qui teste et modifie le contenu d'un mot en mémoire centrale de façon indivisible.

- **Test And Set** qui teste et modifie de manière atomique une variable.
- **Swap** qui échange de manière atomique deux variables.

- **Test And Set**

Test And Set (TAS) est une instruction disponible sur certaines machines. Elle permet la consultation et la mise à jour d'une variable de façon indivisible. Le code qui implémente cette instruction est de la forme :

```
processus P
Debut
tant que (verrou) faire ;   fait                    // attente active
        verrou :=vrai;                              // entrée en section
critique
<Section_Critique>;
verrou :=faux ;                                     // sortie de section
critique
  Fin;
```

La fonction TAS permet la résolution du problème de l'exclusion mutuelle de la manière suivante :

```
booleen verrou := faux                              // variable partagée initialisée
processus P
      Debut
                    tant que (TAS (verrou));        // attente active
                    < Section_Cretique >
                    verrou := faux;
      Fin
```

- **SWAP**

Réalisation d'un verrou avec swap atomique

```
procédure swap (a,b : booleen)                      (* Opération indivisible *)
temp : booleen ;
Debut
          temp=a;
          a=b;
          b= temp;
Fin
```

Le verrou avec swap permet la résolution de l'exclusion mutuelle :

```
booléen verrou :=faux;                              //variable partagée initialisée
processus P
      Debut
                                                    //entrée en section critique
                booleen cle := vrai ;
                tant que (cle) faire
                                swap(cle,verrou);
                                fait;               //attente active
                < section critique>;
                                                    // sortie de section critique
                verrou=faux;
      Fin
```

**Inconvénients de la solution**

- il peut y avoir famine de processus si un processus ne sorte pas au bout d'un temps fini de sa section critique.
- Interblocage possible si deux processus modifient la même variable en même temps.

### *5.1.3 Variables d'alternance*

**Principe de la solution**

- Chaque processus annonce sa candidature à l'autre processus pour assurer l'alternance.
- En cas de candidatures simultanées, le conflit est réglé en donnant la priorité à un processus.
- Pour une solution équitable la priorité doit être variable.
- Les solutions reposent sur l'aspect logiciel et sont présentées pour 2 processus.

- **Algorithme de Dekker (1965)**

```
//déclaration et initialisation des variables communes
booleen C[2] :={faux,faux}; //candidature
entier T:=0; //priorité
```

| ***Processus P0*** | ***Processus P1*** |
|---|---|
| tant que vrai faire<br>//entrée en SC<br>C[0]:=vrai;<br>//P1 est en SC ou demande à y entrer<br>tant que (C[1]) faire<br>//P1 est prioritaire<br>tant que(T=1) faire C[0]:=faux;<br>fait;<br>C[0]:=vrai; //T:=0 P0 recandidate<br>fait;<br><Section_critique>;<br>//sortie de SC<br>C[0]:=faux;<br>T:=1;<br>Hors SC;<br>fait; | tant que vrai faire<br>//entrée en SC<br>C[1]:=vrai;<br>//P0 est en SC ou demande à y entrer<br>tant que (C[0]) faire<br>//P0 est prioritaire<br>tant que(T=0) faire<br>C[1]:=faux;fait;<br>C[1]:=vrai;// T:=1 P1<br>recandidate<br>fait;<br><Section_critique>;<br>//sortie de SC<br>C[1]:=faux;<br>T:=0;<br>Hors SC;<br>fait; |

- **Algorithme de Peterson (1981)**

```
//déclaration et initialisation des variables communes
booleen C[2] :={faux,faux}; //candidature
entier T:=0; //priorité
```

| ***Processus P0*** | ***Processus P1*** |
|---|---|
| tant que vrai faire<br>//entrée en SC<br>C[0] :=vrai;T :=1;<br>//attente en cas de conflit<br>tant que (C[1] and T=1) ;<br><Section_critique>;<br>//sortie de SC<br>C[0] :=faux;<br>Hors SC;<br>fait; | tant que vrai faire<br>//entrée en SC<br>C[1] :=vrai;T :=0;<br>//attente en cas de conflit<br>tant que (C[0] and T=0) ;<br><Section_critique>;<br>//sortie de SC<br>C[1] :=faux;<br>Hors SC;<br>fait; |

**N.B :** La généralisation est difficile pour l'algorithme de Dekker et possible pour l'algorithme de Peterson.

- **Algorithme du Boulanger (Lamport 1974)**

L'algorithme du boulanger est une solution logicielle pour *N* processus consiste à :

- chaque client obtient un numéro
- ça lui permet de négocier la seule baguette restante avant le client suivant
- les clients sont servis par ordre de leur numéro

```
shared boollean choix[N] = false,
shared int numero[N]:=0;
Begin
int i = 0 // pour le processus 0, initialisation à x pour le processus x
int j;
    choix[i] = true;
    numero[i] = Max(numero) + 1;
    choix[i] = false;
    for ( j=0;j< N; j++ )
        {
            While (choix[j] ) attente;
            // tout le monde a son numero
            while (numero[j]!=0 && ((numero[j])<(numero[i]) || (numero[j]==numero[i] && j<i))) attente;
          // c'est mon tour !
       }
    // section critique
    numero[i] = 0; // je rends mon ticket
    // section non critique
Fin
                                Notation : (a,b)<(c,d) ssi (a<c) || ( (a==c) && (b<d) )
```

**Inconvénients de l'attente active**

- Mauvaise utilisation du processeur car l'attente active monopolise le processeur et donc dégrade la performance globale de la machine.
- Ralentissement des accès mémoires (congestion du bus par des accès répétés aux variables de synchronisation)
- Difficulté de conception et complexité des algorithmes

**Cas d'utilisation**

- Sections critiques de courte durée

**Conclusion**

- Réalisation des mécanismes sans attente active (attente passive)

### 5.2 L'exclusion mutuelle avec attente passive

Un processus ne pouvant entrer en section critique, passe dans un état endormi, et sera réveillé lorsqu'il pourra y entrer.

#### *5.2.1 Sémaphores*

Ce mécanisme a été inventé par Dijkstra en 1965. Formellement, un sémaphore **s** est constitué d'un entier **e(s)** pouvant prendre des valeurs positives, négatives ou nulles ; et une file d'attente **f(s)**. Toute opération sur un sémaphore est indivisible. La création d'un sémaphore se fait par déclaration qui doit spécifier la valeur initiale **e0(s)** de e(s).

Conceptuellement, un sémaphore peut être vu comme un distributeur de tickets, initialement dispose d'un certain nombre de tickets, éventuellement 0 ticket. Un processus utilisateur demande un ticket en invoquant une opération appelée **P** (Proberen en Hollandais (Puis-je ou tester)) : si au moins un ticket est disponible, le processus appelant le prend et continue son exécution. Dans le cas contraire, la demande est enregistrée dans une file d'attente et le processus est bloqué dans l'attente d'une condition (disponibilité d'un ticket pour lui).Une deuxième opération possible sur un sémaphore **V** (Verhogen (Vas-y ou incrémenter)) qui permet après avoir été invoquée par un processus :

- Rendre disponible un ticket dans le distributeur.
- Rendre actif un des processus de la file d'attente associée au sémaphore si celle-ci n'est pas vide. Dans le cas contraire, le ticket est conservé dans le distributeur et donc le prochain demandeur verra sa demande immédiatement honorée.

***Algorithme de la primitive P(s)*** *(* Opération indivisible *)*

```
Debut
     e(s):= e(s) - 1     // retrait d'une autorisation (ticket)
    Si ( e(s) < 0 ))   alors
                          Bloquer le processus r ayant effectué la
requête
                          Mettre le processus r dans une file
d'attente f(s)
    Finsi
Fin
```

***Algorithme de la primitive V(s)*** *(* Opération indivisible *)*

```
Debut
        e(s):= e(s) + 1      // ajout d'une autorisation (ticket)
        Si ( e(s) <= 0 )   alors
                                Sortir d'un processus q de la file
d'attente f(s).
                              Réveiller le processus q.
        Finsi
Fin
```

## Quelques variantes de sémaphores

### a) Sémaphore d'exclusion mutuelle

Un sémaphore d'exclusion mutuelle appelé sémaphore mutex (pour Exclusion Mutuelle) est un cas particulier des sémaphores à compte. Ce sémaphore initialisé à **1** autorise un seul processus actif à franchir **P(mutex)**.

### b) Sémaphore privé

Un sémaphore **Sp** est privé à un processus p si seul p peut exécuter les opérations **P(Sp)** et **V(Sp)**. Les autres processus ne peuvent agir sur Sp que par V(Sp). La valeur initiale de Sp est égale à **0**.

## Usage du mécanisme

La réalisation de l'exclusion mutuelle à l'aide des sémaphores se fait selon le schéma suivant :

```
Contexte commun
mutex: sémaphore; e0(mutex) :=1        // 1 seule autorisation
processus P
      tant que (vrai) faire
                              P(mutex);//entrée en section critique
                              <Section critique>;
                              V(mutex);//sortie de section critique
     fait;
```

## Caractéristiques du mécanisme

Ce mécanisme présente les caractéristiques suivantes :

- Avec ce mécanisme le processus désirant exécuter sa section critique quitte le processeur si les conditions ne sont pas satisfaites.
- Ce mécanisme ne présente aucun risque concernant la perte des interruptions
- L'usage de mécanisme nécessite une intention particulière de la part du programmeur, car un mauvais appel de P ou de V peut entraîner des conséquences néfastes.

## Résolution du problème du Producteur-Consommateur avec les sémaphores

### a) Description du problème

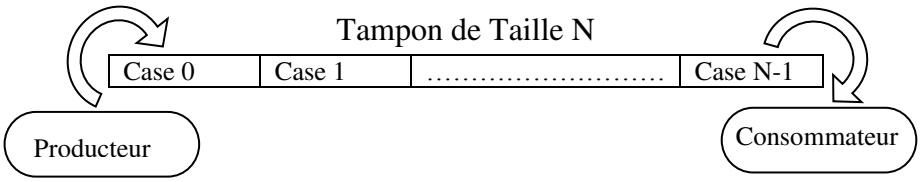



**Figure 2.3** : Schéma du Producteur-Consommateur

- **Les deux processus coopèrent en partageant un même tampon :**

  - Le producteur produit des objets (messages) qu'il dépose dans le tampon
  - Le consommateur retire des messages du tampon pour les consommer

- **Conflits :**

  - Le producteur veut déposer un objet alors que le tampon est déjà plein
  - Le consommateur veut retirer un objet du tampon alors que celui-ci est vide
  - Le producteur et le consommateur ne doivent pas accéder simultanément au tampon

### b) Résolution du problème

| semaphore mutex : = 1; /* contrôle l'accès à la SC */<br>semaphore vide : = N; /* nb d'emplacements vides dans le tampon */<br>semaphore plein : = 0; /* nb d'emplacements pleins */ | |
|---|---|
| **Producteur** | **Consommateur** |
| *Debut*<br>*Cycle // boucle sans fin*<br>*Produire(message);// génère un message*<br>P(vide); //décrémente les emplacements vides<br>P(mutex); // entre en SC<br>/* palce le message dans le tampon*/<br>*Deposer(message)*<br>V(mutex);// quitte la SC<br>V(plein);//incrémente les emplacements pleins<br>Fin<br>Fin | Debut<br>Cycle *// boucle sans fin*<br>P(plein);//décrémente les emplacements pleins<br>P(mutex);// entre en SC<br>/*prélève un message dans le tampon*/<br>*Retirer(message);*<br>V(mutex);//quitte la SC<br>V(vide);// incrémente les emplacements vides<br>*Consommer(message)*;// exploiter le message<br>Fin<br>Fin |

**Inconvénients des sémaphores**

Le sémaphore est un mécanisme simple et puissant, mais "dangereux" dans un emploi non contrôlé où plusieurs inconvénients peuvent surgir :

1. **Interblocage** : L'implémentation d'un sémaphore avec une fille d'attente peut conduire à une situation dans laquelle deux processus ou plus attendent indéfiniment un événement qui ne peut être produite que par l'un des processus en attente. L'événement en question est l'exécution d'une opération V. lorsqu'on atteint un tel état, ces processus sont dits en situation d'**interblocage**.

**Exemple :** Soient 2 processus P0 et P1 qui se partagent 2 ressources critiques. On associe 2 sémaphores mutex1 et mutex2 (initialisés à 1) à ces ressources.

| **processus P0** | **processus P1** |
|---|---|
| …….<br>P(mutex1);<br>P(mutex2);<br>…. …<br>V(mutex1);<br>V(mutex2);<br>……. | ……..<br>P(mutex2);<br>P(mutex1);<br>……..<br>V(mutex2);<br>V(mutex1);<br>…… |

Supposez que :

- P0 exécute P(mutex1),
- Puis, P1 exécute P(mutex2)
- P0 pour exécuter P(mutex2) doit attendre jusqu'à ce que P1 exécute V(mutex2)
- De même manière similaire, P1 pour exécuter P(mutex1) doit attendre jusqu'à ce que P0 exécute V(mutex1)
- Alors on dit que : P0 et P1 sont dans une situation d'**interblocage**

2. **Famine :** un autre problème lié à l'interblocage est la famine ou l'attente indéfinie qui est une situation dans laquelle les processus attendent indéfiniment à l'intérieur d'un sémaphore. Celle-ci peut arriver si nous ajoutons et retirons des processus de la liste associer à un sémaphore avec un ordre LIFO(Last In First Out).

### *5.2.2 Moniteurs*

Bien que les sémaphores proposent un mécanisme général pour contrôler l'accès aux sections critiques, leur utilisation ne garantit pas que cet accès est mutuellement exclusif ou que les interblocages sont évités. Les erreurs de programmation, telles que l'omission d'une opération P avant que le processus entre en Section critique, peuvent conduire à un mauvais fonctionnement du programme. Même, la détection et la correction de ces erreurs sont complexes et difficiles.

Pour éviter ces erreurs de programmation, Hoare(1974) et Brinch Hansen(1975) ont développé un nouveau concept appelé **moniteur** pour faciliter l'écriture des programmes corrects. Un moniteur est une structure de variables et de procédures pouvant être paramétrée et partagée par plusieurs processus. Ces variables et procédures de synchronisation sont encapsulées dans le module « Moniteur » selon la structure suivante :

```
type m : moniteur
  Déclaration des variables locales;
  Déclaration et corps des procédures du moniteur;   // accessibles seulement en exclusion mutuelle
  Initialisation ;
Fin moniteur ;
```

**Usage du mécanisme**

- Seul un processus peut être actif à un moment donné à l'intérieur du moniteur.
- La demande d'entrée dans un moniteur (ou d'exécution d'une primitive du moniteur) sera bloquante tant qu'il y aura un processus actif à l'intérieur du moniteur
  **=> L'accès à un moniteur construit donc implicitement une exclusion mutuelle**
- Lorsqu'un processus actif au sein d'un moniteur ne peut progresser dans son travail (une certaine condition est fausse), il libère l'accès au moniteur avant de se bloquer.
- Lorsque des variables internes du moniteur ont changé, le moniteur doit pouvoir « réveiller » un processus bloqué.
- Synchronisation se fait par les deux primitives :

  – **wait** : qui libère l'accès au moniteur, puis bloque le processus appelant sur une condition.
  – **signal** : qui réveille sur une condition un des processus en fille d'attente à l'intérieur du moniteur (un processus qui a exécuté précédemment un wait sur la même condition). Si cette file d'attente est vide elle ne fait absolument rien.

**Syntaxe**:

```
cond.Wait; /*cond est la variable condition déclarée comme variable locale */
cond.Signal;
```

Une variable condition est une variable qui est définie à l'aide du type condition et qui n'a pas de valeur (contrairement à un sémaphore) mais elle est implémentée à l'aide d'une file d'attente.

**cond.Wait :** bloque toujours le processus appelant
**cond.Signal :** réveille un processus bloqué dans la file d'attente associée à cond.

**Un processus réveillé par Signal continue son exécution à l'instruction qui suit le Wait qui l'a bloqué.**

**Résolution du problème du Producteur-Consommateur avec les moniteurs**

```
Type ProducteurConsommateur : Moniteur
/*variables locales*/
Var Compte : entier ; Plein, Vide : condition ;
/* procédures  accessibles aux programmes utilisateurs*/
Procedure Deposer(message M) ;
Début
si Compte=N alors Plein.Wait ;
dépôt(M);
Compte :=Compte+1;
si Compte=1 alors Vide.Signal;
Fin
Procedure Retirer(message M) ;
Début
si Compte=0 alors Vide.Wait ;
retrait(M);
Compte :=Compte-1;
si Compte=N-1 alors Plein.Signal;
Fin
/*Initialisations*/
Compte := 0;
Fin moniteur ;
```

| Processus Producteur | Processus Consommateur |
|---|---|
| message M;<br>Debut<br>tant que vrai faire<br>Produire(M);<br><br>ProducteurConsommateur.Deposer(M)<br>Fait<br>Fin | message M;<br>Début<br>tant que vrai faire<br><br>ProducteurConsommateur.Retirer(M);<br>Consommer(M);<br>Fait<br>Fin |

### *5.2.3 Régions critiques*

Le concept de régions critiques, identique dans sa sémantique à celui des sections critiques, a été émis dans le souci de fournir au programmeur un outil capable de lui apporter une aide lors de la spécification de la synchronisation. Cette aide réside dans la garantie que toute variable déclarée explicitement partageable sera utilisée en exclusion mutuelle. Autrement dit, le mécanisme d'exécution des régions

critiques veillerait à ce que ces variables soient manipulées correctement. Ce mécanisme, intégrable au compilateur, avertirait donc le programmeur de toute utilisation d'un objet déclaré partageable en dehors d'une région critique.

**Definition**

Une région critique est donc une structure linguistique qui définit une séquence d'instructions à exécuter comme section critique dans laquelle sont manipulées des variables déclarées explicitement partageables. Les variables partagées sont explicitement placées dans des groupes appelés ressources.

Chaque variable partagée peut être au plus dans une ressource. Par exemple, une ressource R contenant les variables V1, V2, ...., Vn est déclarée par: Ressource R: V1,V2, ..., Vn. Les variables de R peuvent être accessibles seulement à partir des formes structurales de régions critiques désignant ces variables.

**Structure des régions critiques (syntaxe et sémantique):**

Pour déclarer une variable partagée V du type T on utilise la notation suivante:

***Var*** V : ***shared*** T qui définit une variable V du type T;

**par exemple:** *var* compte **: *shared real*.**

Il est possible de considérer T comme une structure regroupant un ensemble de variables partagées distinctes (identiques relativement au partage) comme l'illustre la notation suivante:

*Type* **direction** = *record*

**attente, passant, en_tete: integer**

*end*

*Var* **pont : *shared*** *record*

**gauchedroit, droitgauche : direction;**

*end*

On définit une région critique à l'aide de deux syntaxes différentes, à savoir:

***1. Région critique inconditionnelle***

***Region*** V ***do*** Sc

dont la sémantique exprime que la séquence d'instructions Sc dans laquelle est (ou sont) manipulée(s) la (ou les) variable(s) partagée(s) V, constitue une section critique.

La séquence d'instructions Sc contient des variables déclarées partageables et éventuellement des variables locales au processus exécutant Sc. Plusieurs régions critiques peuvent se rapporter à une même variable partagée; leurs exécutions s'excluent mutuellement. Autrement dit, si plusieurs processus tentent de les exécuter simultanément, un seul d'entre eux est autorisé à le faire (évolue normalement); les autres processus seront automatiquement bloqués. Lorsque le processus actif termine l'exécution de la séquence critique sc, un des processus en attente est systématiquement réactivé.

## 2. ***Région critique conditionnelle***

a) ***Region*** V ***when*** cond ***do*** sc

Signifie: l'exécution de la région critique sc est conditionnée par la valeur de la condition cond. Cond représente une expression booléenne composée d'éléments de V et éventuellement de variables locales et constantes. Le scénario d'exécution de cette primitive par un processus appelant P est comme suit:

P accède à la région critique sc puis évalue cond, si cond est vraie, alors P exécute sc puis quitte sc.

sinon (cond est fausse) P sort de la région critique et se bloque jusqu'à ce qu'il soit réveillé lorsqu'un autre processus Q sort de sa propre région critique associée à V. P reprend alors l'évaluation de cond à son début.

b) ***Region*** V ***do*** sc ***await*** cond ...

Cette instruction combine les deux formes précédentes. Le processus appelant exécute donc sc de façon inconditionnelle puis, pour poursuivre, se met en attente (se bloque) jusqu'à ce que la condition cond devienne vraie. Il est possible que sc soit une opération nulle auquel cas, l'évolution du processus appelant dépendra de la valeur de cond (attente si cond est fausse, poursuite en séquence sinon).

***Remarque***: Des régions critiques peuvent s'imbriquer à la manière des boucles DO des langages de programmation de haut niveau, toutefois une attention particulière peut leur être accordée afin d'éviter les interblocages éventuels. Par exemple, soient deux processus concurrents P et Q se partageant deux variables v1 et v2 tels que leurs codes soient:

**P**: ***Region*** v1 ***do Region*** v2 ***do*** sc1;
**Q**: ***Region*** v2 ***do Region*** v1 ***do*** sc2;

Dans pareil cas, il est évident de constater que si P et Q entrent simultanément dans leur propre region critique, un interblocage est inévitable. Il est possible de charger le compilateur qui gère les régions critiques de détecter des situations d'interblocage et d'en informer le programmeur pour agir en conséquence. Une action possible, et à titre indicatif, serait d'imposer un ordre d'utilisation des variables partagées.

Les programmes contenant des éléments de synchronisation spécifiés en termes de régions critiques sont plus lisibles. En effet, il est possible d'utiliser une approche axiomatique, basée sur la notion d'invariants, pour prouver la validité de l'exécution de la séquence d'instructions Sc. Ainsi, on peut associer un invariant Iv à l'état de chaque ressource V, et un prédicat P. Le prédicat P ayant la valeur vraie à l'initialisation de V doit également avoir la valeur vraie à la fin de l'exécution de Sc.

Les régions critiques ont été implantées dans le langage Edison [Brin 1981] conçu spécialement pour des systèmes multiprocesseurs. Des variantes ont été également adaptées pour des environnements distribués.

***Application***: Simuler un sémaphore à l'aide de régions critiques.

***var*** sem: ***shared integer*** (sem ≥0);
**P**(sem): ***region*** sem ***do*** sem := sem - 1 ***await*** sem ≥□0 ... (1) % incorrect %

Bien que la forme (1) précédente, traduisant textuellement la primitive P originelle, semblerait «correcte» indépendamment de V(sem), elle est à rejeter puisqu'elle ne répond pas à une association correcte avec la primitive de réveil V ci-après:

***V***(sem)***: region*** sem ***do*** sem ***:=*** sem + 1***;***

En effet, l'exécution de V simulée précédemment a pour objectif de libérer un processus Q éventuellement en attente. D'après le principe des régions critiques, Q réveillé, doit évaluer sa condition d'entrer en section critique. Si celle-ci est vraie, il exécute effectivement sa section et poursuit en séquence; dans le cas contraire (condition fausse), il entre de nouveau dans la file d'attente.

Quand plusieurs processus se trouvent bloqués par P, impliquant sem < -1, l'exécution de V, à la suite de laquelle sem est incrémenté (sem := sem + 1) provoquerait des réévaluations des conditions de franchissement des points de synchronisation ineffectives, bien que logiquement, un franchissement au moins, devrait avoir lieu.

Par exemple, soient trois processus concurrents, P1, P2, P3, désireux d'entrer dans leur section critique respective (sem initialisé à 1), admettant que P1 soit le premier à exécuter la forme (1) précédente, sem devient = 0 et P1 accède à sa section critique (condition vraie). Si P2 exécute la forme (1) pendant que P1 est encore en section critique, sem devient = -1 et P2 se bloque (condition fausse). Si P3 exécute également (1), il se bloque car sem = -2. Lorsque P1 quitte sa section critique, il doit exécuter V(sem) qui incrémente sem (sem devient égal à - 1). Le mécanisme d'exécution des régions critiques va réveiller P2 qui évalue sa condition de franchissement (sem ≥ 0) et trouve sem = -1 (condition fausse), P2 libère sa section critique et se bloque une nouvelle fois. P3 fait la même chose que P2, trouve sem = -1, et se bloque également. On voit bien que la section critique est libre et aucun des processus ne peut y accéder. P1, lui-même, ou tout autre nouveau processus concurrent ne peut accéder à sa section critique, il se produit donc un blocage. Ce problème est dû au fait que la décrémentation de sem s'est faite avant le test de la condition de franchissement. La solution correcte est donc la suivante:

***Region*** sem ***when*** sem > 0 ***do*** sem := sem - 1 ...

### *5.2.4 Expressions de chemins*

La technique d'expression des chemins consiste à trouver des chemins d'enchainement des exécutions des procédures d'un traitement (processus). Une expression de chemin est pour laquelle il est possible de trouver un graphe tel que le même nom de procédure ne figure pas sur plus d'un arc issu du même nœud est appelé chemin simple. Un chemin simple peut être mis en œuvre par génération d'opérations P et V sur sémaphores.

***Application*** *:*

Soit un tampon pouvant contenir un message unique. Des processus garnissent ou vident ce tampon dans un ordre imprévisible.

```
class TamponUnique;
var mess : Message;
path déposer; retirer end
procedure déposer(m:Message);
begin
mess := m
end; {de déposer}
procedure retirer:Message;
begin
retirer := mess
end; {de retirer}
begin
mess := nil
end; {de TamponUnique}
```

## 5.3 Synchronisation des threads en Java

Un programme est souvent conçu comme une suite (séquentielle) d'instructions. Néanmoins, il est possible de concevoir des programmes où plusieurs taches se déroulent simultanément, en parallèle; ces différents taches portent le nom de ***threads*** et ont dit alors que l'application est ***multithread*** (programme concurrent). Pour obtenir des applications multithread, en lalgage java, on démare, en génaral avec une méthode **main**, un premier thread qui peut en lancer d'autres sans s'interrompre; les nouveaux threads peuvent à leur tour démarrer d'autres threads. Un thread est quelquefois aussi appelé ***processus léger*** ou ***file d'exécution*** (flow de contrôle).

Les threads sont comme des processus, s'exécutent en parallèle, mais dans le même espace d'adressage en partageant les mêmes données.

**Exemple**: concurrence de traitement lors de chargements d'images avec les navigateurs Web.

Lorsque l'application est multithread, il faut souvent synchroniser les accès aux donnés partagées de façon à conserver leur cohérence. Par ailleurs, les différents threads doivent pouvoir se coordonner, s'attendre, s'alterner, être interrompus définitivement, être interrompus provisoirement et alors éventuellement être repris, etc.

### *5.3.1 Création de threads avec Java*

Threads gérés dans différents langages. Gestion lourde avec C (API POSIX) et simplifiée avec Java.
En java, les threads sont des instances des classes dérivées (héritées) de la **classe *Thread***

- La classe Thread crée des threads généraux, sa méthode **run** ne fait rien.
- La méthode **run** indique à un thread les instructions à exécuter
- La méthode **run** doit être publique, ne prendre aucun argument, ne renvoyer aucune valeur et ne lever aucune exception.

Deux techniques pour fournir une méthode run à un thread, à savoir :

1) hériter la classe Thread (java.lang.Thread) et redéfinir la méthode run.
2) Implémenter l'interface Runnable (java.lang.Runnable) et définir la méthode run de cette classe

Un thread commence par exécuter la méthode run de l'objet "cible" qui a été passé au thread.

**Exemples:** `Test1Thread.java et TestThread2.java` (voir TP n°5 dans la 2eme partie de ce document)

### *5.3.2 Méthodes de gestion des threads*

La figure 2.4 montre les différentes méthodes de gestion des threads, à savoir :

**Attente et réveille : wait() , notify() , notifyAll() et sleep()**

Ce sont des méthodes de la classe ***Object*** Appelées dans des blocs de code synchronisés

- ***wait()*** **:**
  - le thread renonce à son verrou et s'endort
  - il est en attente la possession du verrou

- ***notify()*** **:**
  - Chaque appel à *notify()* ne réveille qu'un seul thread en attente par wait() sur le même objet.
  - S'il y a plusieurs threads en attente, Java choisi le premier en file FIFO.

- notifyAll() réveille tous les threads en attente par wait() sur le même objet.
- On utilise souvent notifyAll() plutôt que notify().

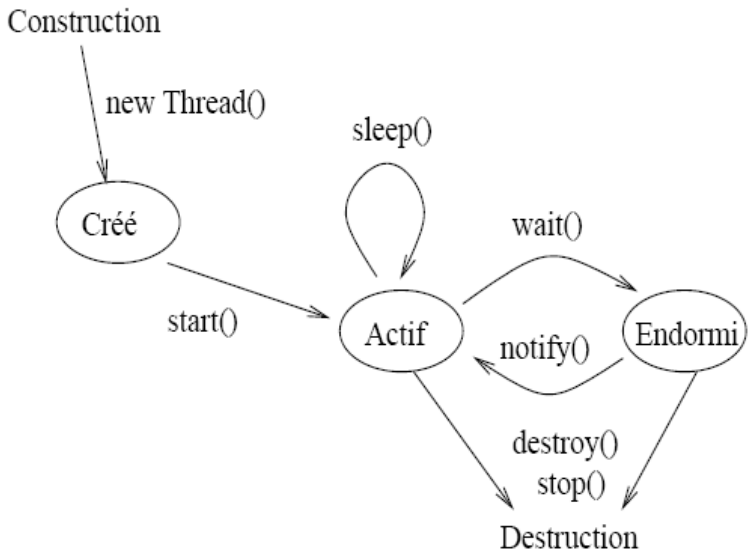



**Figure 2.4** : Méthodes de gestion des threads

Un thread devient inactif si un des événements suivants survient:

- La méthode *sleep(n)* : le thread est mis en sommeil ***n*** milisecondes
- La méthode *wait()* : attendre qu'une condition soit satisfaite.
- Attendre un verrou pour exécuter une méthode synchronisée
- Attendre que des opérations entrées/sorties soient terminées

Le réveil d'un thread inactif dépend de la cause:

- Thread endormi par un appel de *sleep* : réveillé automatiquement lorsque le nombre de milisecondes est écoulé
- Thread endormi par un appel de *wait* : réveillé lorsqu'il y a un autre objet qui signale par un appel *notify()* ou *notifyAll()*
- Thread en attente d'un verrou : continuera son exécution lorsqu'il aura acquis le verrou
- Thread en attente des E/S : réveillé lorsque ces E/S soient complètes

### *5.3.3 Synchronisation et Exclusion Mutuelle*

Lorsque plusieurs threads travaillent sur des mêmes ressources, il est parfois utile ou nécessaire de limiter le parallélisme en assurant qu'un certain objet ne subisse pas en même temps plusieurs séquences d'actions concurrentes.

En bref, les threads partagent même espace d'adressage, alors, problème de synchronisation.

Pour gérer la synchronisation entre les threads, Java fournit des structures basées sur le concept de moniteurs:

- Moniteur = verrou, attaché à une ressource à laquelle peuvent accéder plusieurs threads, mais un seul thread à la fois

- Le mot clé **synchronized** spécifie l'endroit (la ressource) où un thread doit acquérir le verrou avant de poursuivre son exécution.

**Exemple** : Dans `TestThread3.java` (voir TP n°6 dans la 2eme partie de ce document), chaque thread affiche des mots caractère par caractère. Les mots sont mélangés. On souhaite conserver le parallélisme du traitement en garantissant que chaque mot entier ne soit pas coupé.

**Construction d'une classe moniteur**

On définit une classe "moniteur" contenant une méthode "**synchronized**" qui va permettre l'écriture d'un mot dans son entier: lorsque cette méthode est appelée dans un thread, elle bloque le déroulement des autres threads jusqu'à ce qu'elle soit arrivée à la fin de son traitement.
**Attention** : pour que ce mécanisme fonctionne, il faut que les threads partagent le même moniteur. Ici, cela se traduit par la définition, dans la classe des threads, d'un champ "static" pour contenir le moniteur sur lequel on invoquera les méthodes synchronisées.

**Exemple**: `TestThread4.java` (voir TP n°6 dans la 2eme partie de ce document)

Les appels aux méthodes synchronisées sont appelés ***sections critiques***. Il faut les utiliser avec prudence:

- elles suppriment le caractère concurrent du traitement d'où une dégradation des performances ;
- elles peuvent conduire à des situations d'interblocage.

### *5.3.4 Cas d'étude : Producteur-Consommateur en Java*

- Une mémoire "tampon" est gérée par un producteur d'objets et un consommateur. Cette mémoire ne peut contenir qu'un objet.
- On synchronise l'accès à cette mémoire (production et consommation non simultanées).
- Si le producteur doit déposer un objet alors qu'il en existe déjà un dans la mémoire tampon, il attend ... blocage !
- Même type de blocage pour un consommateur qui attend le dépôt d'un objet.

La `solution` de ce problème passe par la mise en œuvre d'une procédure d'exclusion mutuelle :
- L'attente du Producteur ou du Consommation se fait au moyen de la méthode `wait()` sur l'objet(susceptible de déclencher une exception `InterruptedException`).
- Un processus sort de l'attente lorsqu'un autre processus exécute la méthode `notify()` (qui relance ***un*** processus en attente) ou la méthode `notifyAll()` (qui relance ***tous*** les processus en attente)

**Exemple :** `ProdConsTest.java`

**ProdConsTest.java**

```
import java.io.*;
import java.lang.*;

class MoniteurProdCons {
            String tampon;
            boolean estVide = true;
```

```
            synchronized void prod(String m)
                {
                if (!estVide) {
                    System.out.println("Producteur attend");
                    try { wait(); }
                    catch (InterruptedException e) {}
                              }
                    System.out.println("Produit : "+ m);
                    tampon = m; estVide=false; notify();
                }
            synchronized void cons()
                {
                if (estVide) {
                    System.out.println("Consommateur attend");
                    try { wait(); }
                    catch (InterruptedException e) {}
                              }
                    System.out.println("Consomme : "+ tampon);
                    estVide=true; notify();
                              }
}
class Producteur extends Thread {
                MoniteurProdCons tampon ;
                public Producteur (MoniteurProdCons t)
                {
                  tampon = t;
                }
                public void run() {
                       tampon.prod("message1");
                       tampon.prod("message2");
                       try { sleep(100); }
                       catch(InterruptedException e) {}
                       tampon.prod("message3");
                                  }
}
class Consommateur extends Thread {
               MoniteurProdCons tampon;
               public Consommateur(MoniteurProdCons t)
               {
               tampon = t;
               }
               public void run() {
                      tampon.cons();
                      tampon.cons();
                      tampon.cons();
                                 }
}
public class ProdConsTest {
       public static void main(String argv[]) {
```

```
        MoniteurProdCons tampon =
        new MoniteurProdCons();
        new Producteur(tampon).start();
        new Consommateur(tampon).start();
                              }
}
```

### *5.3.5 Ordonnancement et priorité*

- Théoriquement : threads s'exécutent en parallèle, pratiquement sur un simple CPU: 1 seul thread à un moment
- Ordonnancement : ordre d'exécution de plusieurs threads sur un simple CPU
- Chaque thread possède une priorité propre, modifiable par *setPriority()*
- Threads de priorité élevée devancent ceux de priorités inférieures
- Même priorité, exécution en tranches de temps

## 6. Questions

### a) Etude de cas 1 : problème du coiffeur endormi

Il s'agit de synchroniser les activités du problème de coiffeur endormi et de ses clients avec des sémaphores (voir la section 4.4 dans ce chapitre). Apres l'initialisation des sémaphores, la solution complète de ce problème est donnée comme suit:

```
const N =5 ;// chaises en attente de clients
entier Attent :=0;//clients en attente
semaphore CFR, CLT, mutex;
Init (CFR, 0);
Init (CLT, 0);
Init (mutex, 1);
```

| **Programme coiffeur** | **Programme client** |
|---|---|
| Coiffeur(){<br>while (1){<br>P(CLT);<br>P(mutex);<br>Attend = Attend -1;<br>V(CFR);<br>V(mutex);<br>Coiffer();<br>}<br>} | Client() {<br>P(mutex);<br>if(Attend < N) {<br>Attend:= Attend + 1;<br>V (CLT);<br>V (mutex);<br>P (CFR);<br><br>SeFaireCoifferEtSortir();<br>}<br>else {<br>V(mutex);<br>Sortir();<br>}<br>} |

1. Détailler le fonctionnement du coiffeur et de ses clients tels qu'ils sont représentés par les deux procédures Coiffeur et Client.
2. Quel est le rôle de chacun des sémaphores CFR, CLT et mutex ?

### b) Etude de cas 2 : problème du carrefour (des feux de circulation)

La circulation au carrefour de deux voies est réglée par des signaux lumineux (feu vert/rouge). Quand le feu est vert pour une voie, les voitures qui y circulent peuvent traverser le carrefour ; quand le feu est rouge, elles doivent attendre. On suppose que les voitures traversent le carrefour en ligne droite et que le carrefour peut contenir une ou plusieurs voitures à la fois. On impose les conditions suivantes :

- toute voiture se présentant au carrefour le franchit en un temps fini,
- les feux de chaque voie passent alternativement du vert au rouge, chaque couleur étant maintenue pendant un temps fini.
- A un instant donné, le carrefour ne doit contenir que des voitures d'une même voie.

Les arrivées sur les deux voies sont réparties de façon quelconque. Le fonctionnement de ce système peut être modélisé par un ensemble de processus parallèles :

- un processus qui exécute la procédure ***Changement*** de commande des feux;
- un processus P est associé à chaque voiture;
- la traversée du carrefour par une voiture qui circule sur la voie i (i = 1, 2) correspond à l'exécution d'une procédure ***Traverseei*** par le processus associé à cette voiture.

**Remarque:** Le processus P doit attendre que les voitures engagées dans le carrefour en soient sorties avant d'ouvrir le passage sur l'autre voie.

1. On demande d'écrire le programme ***Changement*** ainsi que les procédures ***Traversee1*** et ***Traversee2*** dans les deux cas suivants :
   - Cas 1 : le carrefour peut contenir une voiture au plus à la fois
   - Cas 2 : le carrefour peut contenir k voitures au plus à la fois

## 7. Références


- Tanenbaum, « Modern operating systems », third edition, Pearson, 2014
- Tanenbaum, « Systèmes d'exploitation », Dunod, 1994.
- Crocus, « Systèmes d'exploitation des ordinateurs », Dunod,1993.
- Sacha Krakowiak, « Principes des systèmes d'exploitation des ordinateurs », Dunod, 1993
- http://www-int.impmc.upmc.fr/impmc/Enseignement/ye/informatique/systemes/chap3/index.html

Chapitre III

# Communication Interprocessus

## Objectifs

- Comprendre les différentes stratégies de communication entre les processus.

## Themes couverts

- Partage de variables (modèles : producteur/consommateur, lecteurs/rédacteurs)
- Boites aux lettres
- Echange de messages (modèle du client/ serveur)

# Chapitre 3: Communication Interprocessus

## 1. Introduction

Lorsque des processus ont besoin d'échanger des informations, ils peuvent le faire par l'intermédiaire d'une zone de mémoire commune. Nous en avons déjà vu un exemple lors de la présentation des lecteurs et des rédacteurs dans le chapitre précédent. Cette communication présente l'inconvénient d'être peu structurée, et nécessite l'utilisation de l'exclusion mutuelle pour accéder à cette zone commune. Par ailleurs elle pose des problèmes de désignation des données de cette zone qui doit être dans l'espace de chaque processus qui désire communiquer avec les autres. Ce chapitre a pour objectif de présenter les différentes stratégies de communication entre les processus.

## 2. Communication interprocessus

Les processus peuvent communiquer par l'intermédiaire de fichiers, des emplacements de mémoire partagées et des messages. Les fichiers constituent le mécanisme le plus fréquemment utilisé pour le partage d'information. Les informations écrites dans un fichier par un processus peuvent être lues par un autre processus. On peut concevoir aussi des processus 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. Ce mécanisme de la mémoire partagée est utilisé pour prendre en charge les **threads**. Un thread est un flot d'exécution (chemin d'exécution) dans le code du processus, doté d'un compteur ordinal qui effectue le suivi des instructions à exécuter ; des registres qui détiennent ses variables de travail en cours et d'une pile qui contient l'historique de l'exécution lui sont propres. Le terme **multithreading** est également employé pour décrire la situation dans laquelle plusieurs threads sont présents dans le même processus.

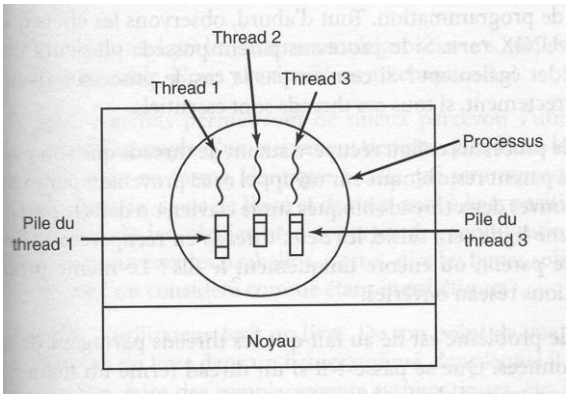



**Figure 3.1** : Chaque thread à sa propre pile

La figure 3.1 représente un processus unique avec trois threads de contrôle. Les trois threads se partagent le même espace d'adressage et s'exécutent quasiment en parallèle comme s'il s'agissait de processus distincts (à l'exception du fait que l'espace d'adressage est commun)

Un thread qui interagit avec une autre au sein du même processus n'**utilise pas** le système d'exploitation et sa gestion est plus légère qu'un processus (un thread est un processus léger).

L'envoi des messages peut également être utilisé pour communiquer les informations entre processus. La communication par message entre processus suppose la définition d'un protocole précis d'échange des messages. De nombreux paramètres interviennent en effet dans une telle communication. Nous considérons ici le type de protocole le plus élémentaire comportant seulement deux opérations émettre et recevoir permettant à deux processus de communiquer selon un protocole de type point à point. Même dans ce cas, de nombreuses variantes existent :

- la synchronisation entre émetteur et récepteur : il s'agit de préciser à quel moment une opération d'émission se termine pour l'émetteur. Si l'émetteur est bloqué jusqu'à ce que le message envoyé ait été reçu et acquitté par le récepteur, on parle de communication **synchrone par rendez-vous**. Sinon, on peut considérer qu'il s'agit d'une communication de type **asynchrone** c.-à-d. l'opération d'émission se termine sans attendre d'acquittement de la part du récepteur. Dans ce type de protocole, la connaissance de l'état du récepteur par l'émetteur est donc beaucoup plus flou et la programmation en "asynchrone" est donc souvent plus délicate bien qu'elle possède l'avantage d'offrir un plus fort parallélisme potentiel entre émetteur et récepteur.

- La communication entre deux processus nécessite un moyen de désignation direct ou indirect du récepteur. C'est pourquoi, la communication entre processus nécessite le plus souvent une phase préalable de connexion.

- La communication peut être occasionnelle ou établie pour une période donnée, **fiable ou non fiable**. On parle de datagramme dans le cas occasionnel et non fiable, par exemple le protocole UDP (User Datagram Protocol), ou de circuit virtuel dans le cas fiable, par exemple TCP (Transmission Control Protocol).

## 3. Stratégies de la communication interprocessus

La communication est un outil important qui permet de faire évoluer l'ensemble des processus dans un système. La communication interprocessus (interprocess communication, IPC) consiste à transférer des données entre les processus. Par exemple, un navigateur Internet peut demander une page à un serveur, qui envoie alors les données HTML. Il existe plusieurs stratégies pour réaliser cette communication, notamment:

- Variables partagées (modèles: producteur/consommateur ; lecteurs/rédacteurs)
- Boites aux lettres
- Échange de messages (modèles: producteur/consommateur, client/serveur,

### 3.1.Partage de variables

Un modèle de communication entre processus avec partage de zone commune (tampon) est le modèle producteur-consommateur. Ce modèle a été décrit en détail dans le chapitre 2 sur la synchronisation.

#### *3.1.1 Modèle producteur-consommateur*

On peut définir un mécanisme général de communication entre processus, où l'un est l'émetteur de l'information (on lui donne le nom de producteur), et l'autre est le récepteur de l'information (on lui donne le nom de consommateur). Il n'est souvent pas nécessaire de faire attendre le producteur jusqu'à ce que le consommateur ait reçu l'information. Pour cela il suffit de disposer d'un tampon entre les deux processus pour assurer la mémorisation des informations produites non encore consommées. Nous appellerons message le contenu d'une telle information. Le tampon peut recevoir un nombre limité de ces messages, noté N. Les deux processus doivent se synchroniser entre eux de façon à respecter

certaines contraintes de bon fonctionnement. La figure 13.2 donne un schéma de cette synchronisation.

```
                        initialisation
                         n_plein.niveau := 0; // variable partagé
                         n_vide.niveau := N; // variable partagé
producteur                                    consommateur
down(n_vide);                                 down(n_plein);
dépôt dans le tampon;                   retrait du tampon;
up(n_plein);                                  up(n_vide);
```

**« Le modèle producteur-consommateur par le partage de variables»**

Le producteur ne peut déposer un message dans le tampon s'il n'y a plus de place libre. Le nombre de places libres dans le tampon peut être symbolisé par un nombre correspondant de jetons disponibles; ces jetons peuvent être conservés par un sémaphore n_vide.
Le consommateur ne peut retirer un message depuis le tampon s'il n'y en a pas. Le nombre de messages déposés peut être également symbolisé par un nombre correspondant de jetons disponibles; ces jetons peuvent être conservés par un sémaphore n_plein.
Le consommateur ne doit pas retirer un message que le producteur est en train de déposer. Cette dernière contrainte est implicitement obtenue, dans le schéma de la figure, si les messages sont retirés dans l'ordre où ils ont été mis.

Informellement, on peut dire que le producteur prend d'abord un jeton de place libre, dépose son message et rend un jeton de message disponible. De son côté, le consommateur prend un jeton de message disponible, retire le message et rend un jeton de place libre.
Pour obliger les processus à respecter la règle du jeu, les systèmes proposent souvent des opérations de dépôt et de retrait qui assurent elles-mêmes la synchronisation entre les processus.

De la même manière cette solution de communication peut être appliquée sur le modèle lecteur-rédacteur.

### 3.2.Boites aux lettres

Nous appellerons boîte aux lettres un objet système qui implante le schéma producteur-consommateur défini plus haut, en conservant le découpage en messages tels que les a déposés le producteur. Les messages sont parfois typés, comme dans iRMX86, où les messages comportent un en-tête qui précise, en particulier, sa longueur et son type sous la forme d'un code défini par le programmeur.
Le problème de la désignation d'une boîte aux lettres par les processus est résolu de différentes façons. Il s'agit, en effet, de permettre à des processus dont les espaces mémoire sont disjoints de pouvoir désigner une même boîte aux lettres. Une solution comme celle proposée par Unix pour les tubes, peut être retenue, qui consiste à mettre cette désignation dans le contexte des processus au moment de leur création. On préfère, en général, une solution semblable à une édition de liens, qui consiste à donner un nom symbolique aux boîtes aux lettres, sous forme d'une chaîne de caractères. Le système maintient une table de ces noms symboliques associés aux boîtes aux lettres qui ont déjà été créées et non encore détruites. Lorsqu'un processus demande l'accès à une telle boîte aux lettres, le système consulte cette table, et retourne au demandeur le descripteur de la boîte aux lettres (c'est, par exemple, son indice dans le table). Les opérations suivantes se feront en utilisant ce descripteur. Dans certains cas, cette demande d'accès entraîne sa création implicite si elle n'existe pas, alors que d'autres systèmes séparent l'opération de création explicite de la demande d'accès.
Les opérations sur une boîte aux lettres sont l'envoi et la réception de message. L'envoi peut être avec ou sans blocage du demandeur lorsque la boîte aux lettres est pleine. La réception peut être également avec ou sans blocage lorsque la boîte aux lettres est vide. De plus, certains systèmes permettent de filtrer les messages reçus, c'est-à-dire, qu'il est possible de préciser le type des messages que l'on accepte de

recevoir.
Les boîtes aux lettres ont souvent une capacité limitée de messages, comme nous l'avons indiqué dans le schéma général. Parfois cette capacité est nulle, l'envoi de message ne pouvant avoir lieu que lorsque le receveur est prêt à recevoir le message. Constatons que le schéma précédent doit être modifié; nous en laissons le soin au lecteur. On dit alors qu'il y a communication synchrone entre le producteur et le consommateur. L'intérêt essentiel est que cette fois, après le retour de la primitive d'envoi, le producteur a la garantie que le consommateur a effectivement reçu le message, alors que dans le cas général, le producteur ne peut préjuger du moment où cette réception aura lieu. En contrepartie, les processus supportent une contrainte de synchronisation plus forte.

### 3.3.Echange de messages (modèle du client/ serveur)

Certains ont estimé que les sémaphores sont de trop bas niveau et les moniteurs descriptibles dans un nombre trop restreint de langages. Ils ont proposé un mode de communication interprocessus qui repose sur deux primitives qui sont des appels système (à la différence des moniteurs) :

- send (destination , &message)
- receive (source , &message)

Généralement, pour éviter les problèmes dans les réseaux, le récepteur acquitte le message reçu (envoi d'un accusé de réception (acknowledgement)). L'émetteur envoie à nouveau son message s'il ne reçoit pas d'acquittement. Le récepteur, s'il reçoit deux messages identiques, ne tient pas compte du second et en tire la conclusion que l'acquittement s'est perdu.

Dans le contexte d'un mode client-serveur, le message reçu contient le nom et les paramètres d'une procédure à lancer. Le processus appelant se bloque jusqu'à la fin de l'exécution de la procédure et le message en retour contient la liste des résultats de la procédure. On parle d'appel de procédure à distance.

On peut proposer une solution au problème producteur-consommateur par échange de messages avec les hypothèses suivantes :

- les messages ont tous la même taille
- les messages envoyés et pas encore reçus sont stockés par le SE dans une mémoire tampon
- le nombre maximal de messages est N
- chaque fois que le producteur veut délivrer un objet au consommateur, il prend un message vide, le remplit et l'envoie. Ainsi, le nombre total de messages dans le système reste constant dans le temps
- si le producteur travaille plus vite que le consommateur, tous les messages seront pleins et le producteur se bloquera dans l'attente d'un message vide ; si le consommateur travaille plus vite que le producteur, tous les messages seront vides et le consommateur se bloquera en attendant que le producteur en remplisse un.

**Résolution du problème du Producteur-Consommateur par l'échange de messages**

```
Producteur
   objet : entier
   m : message                                          /* tampon des messages */
   Faire toujours
        produire_objet (&objet)                  /* produire un nouvel objet dans le tampon */
        receive (consommateur , &m)              /* attendre l'arrivée d'un message vide */
        construire_message (&m , objet)          /* construire un message à envoyer */
        send(consommateur,&m)                    /* envoyer le message au consommateur */
   Fait
```

```
Consommateur
      Objet, i : entier
      m : message
      pour (i = 0 ; i < N ; i++)   send(producteur , &m)        /* envoyer N messages vides */
      Faire toujours
         receive (producteur , &m)                  /* récupérer un message contenant un objet */
         extraire_objet (&m , &objet)               /* extraire l'objet du message */
         send (producteur , &m)                     /* renvoyer une réponse vide */
         consommer_objet (objet)                    /* consommer l'objet */
     Fait
```

Comme une autre solution basé sur ce principe d'échange de messages, on peut imaginer la solution de boîte aux lettres (mailbox) de capacité N messages avec un producteur se bloquant si la boîte est pleine et un consommateur se bloquant si la boîte est vide.

**Inconvénients de l'échange de messages**

1) **Authentification :** comment le client peut-il savoir à coup sûr qu'il communique avec le véritable serveur de fichier, et non avec un imposteur.
2) **Performance :** le fait de copier des messages d'un processus à un autre est toujours plus lent que d'effectuer une opération avec sémaphores ou d'entrer dans un moniteur.

## 4. Questions

1. Traiter un cas d'études sur la communication interprocessus en utilisant les différentes stratégies?
2. Faire une comparaison (avantages et inconvénients) entre les trois stratégies de communications vues en cours?

## 5. Références


- Tanenbaum, « Modern operating systems », third edition, Pearson, 2014
- Tanenbaum, « Systèmes d'exploitation », Dunod, 1994.
- Crocus, « Systèmes d'exploitation des ordinateurs », Dunod,1993.
- Sacha Krakowiak, « Principes des systèmes d'exploitation des ordinateurs », Dunod, 1993
- http://www-int.impmc.upmc.fr/impmc/Enseignement/ye/informatique/systemes/chap2/index.html

Chapitre IV

# L'interblocage

## Objectifs

- Savoir comment prévenir et éviter de tomber dans les situations d'interblocage.
- Comprendre comment faire la détection et la guérison en cas d'interblocage entre les processus.

## Themes couverts

- Modèles
- Prévention
- Evitement
- Détection/ Guérison

# Chapitre 4: L'interblocage

## 1. Introduction

Dans des conditions normales de fonctionnement, un processus ne peut utiliser une ressource qu'en suivant la séquence suivante : Requête – Utilisation – Libération. Les interblocages surviennent lorsque des processus essaient de réserver une ressource qui ne sera jamais libérée. La plupart des systèmes d'exploitation gèrent les interblocages à l'aide de l'Ostrich algorithm (le gain ne valant l'effort). Dans ce chapitre, nous allons exhiber les différentes méthodes de traitements de l'interblocage.

## 2. Définition de l'interblocage

Un ensemble de processus est dans une situation d'interblocage (**Deadlock** en anglais) si chaque processus de l'ensemble attend un événement qui ne peut être produit que par un autre processus de l'ensemble. Situation dans laquelle deux actions ou plus sont en compétition et où chacune attend que l'autre finisse, ce qui n'arrive jamais.

***Exemple d'interblocage***:

Supposons un système possédant les ressources **1** et **2**, ayant les processus **A** et **B** en exécution. Une façon de provoquer un interblocage est la suivante :

1. Le processus A réserve la ressource 1;
2. Le processus B réserve la ressource 2;
3. Le processus A demande la ressource 2, et tombe en attente;
4. Le processus B demande la ressource 1, et tombe en attente;
5. Interblocage !

***Définition de la famine***

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

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

***L'utilisation de ressources se fait en trois étapes:***

- sollicitation de la ressource;
- utilisation de la ressource;
- libération de la ressource.

***Deux types de ressources:***

1. **Ressource non-retirable** : ressource qui ne peut être retirée sans dommage au processus l'ayant réservée. Ex. imprimante, fichier, sémaphore, verrou d'une base de données.
2. **Ressource retirable** : ressource qui peut être retirée sans dommage au processus l'ayant réservée (pose moins de problèmes puisque ces ressources peuvent être retirées au besoin au profit d'autres processus). Ex. Processeur et mémoire mais pas à n'importe quel moment, carte réseau, carte graphique.

## 2.1 Apparition d'interblocages

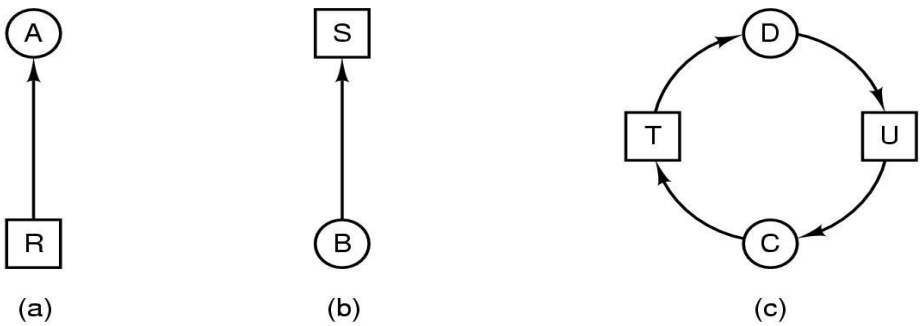
Les interblocages sont évidemment indésirables. Dans un interblocage, les processus ne terminent jamais leur exécution et les ressources du système sont immobilisées, empêchant ainsi d'autres travaux de commencer.
Une situation d'interblocage peut survenir si les quatre conditions suivantes se produisent simultanément (Habermann):

1. ***Exclusion mutuelle***: chaque ressource est soit attribuée à un seul processus, soit disponible.
2. ***Détention et Attente*** : les processus ayant déjà obtenu des ressources peuvent en demander de nouvelles.
3. ***Pas de réquisition*** : les ressources déjà obtenues ne peuvent être retirées de force à un processus.
4. ***Attente circulaire*** : il doit y avoir un cycle d'au moins deux processus chacun attendant une ressource détenue par un autre processus.

## 2.2 Modélisation des interblocages

On modélise les interblocages à l'aide de graphes orientés conçus de la façon suivante:

- Les processus sont représentés pas des cercles.
- Les ressources sont représentées par des carrés.
- Une flèche qui va d'un carré à un cercle indique que la ressource est déjà attribuée au processus (figure **a**).
- Une flèche d'un cercle vers un carré indique que le processus est bloqué en attente de cette ressource (figure **b**).
- Les interblocages sont représentés dans ces graphiques par la présence d'un cycle (figure **c**)

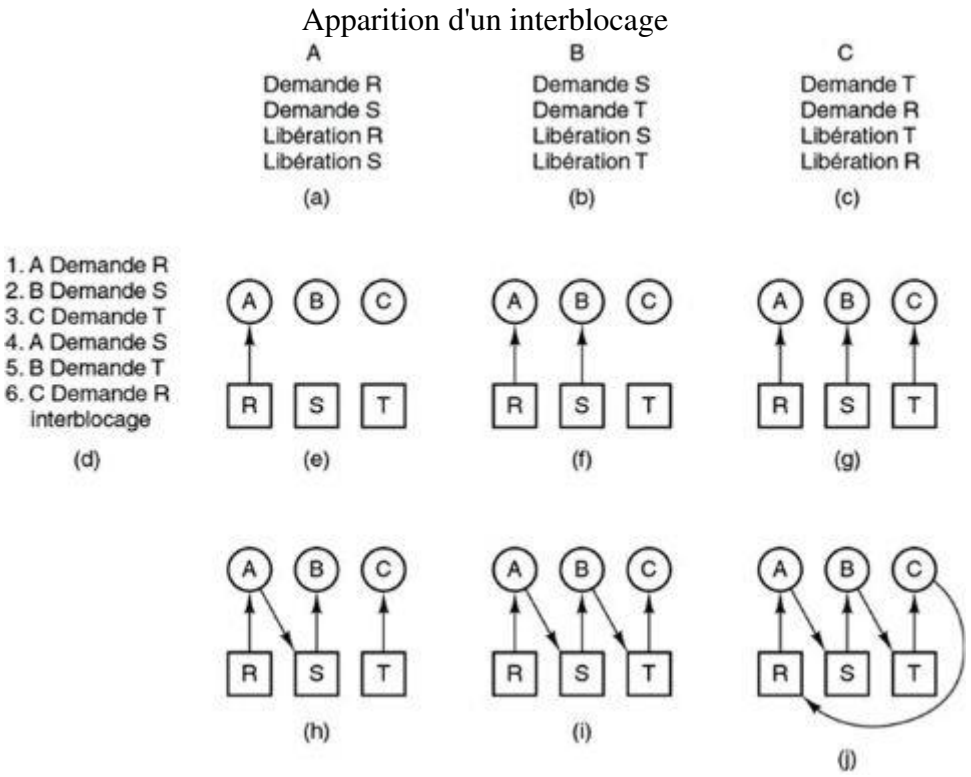



(a) (b) (c)

(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**.

***Exemple de graphe***

Supposons que les trois processus **A**, **B** et **C** effectuent les demandes de ressources illustrées aux figures **a)**, **b)** et **c)**. Il est possible qu'il y ait interblocage si l'ordonnanceur permet à ces processus de réserver leurs ressources comme indiqué en **d)**. En tel cas, la situation évolue de l'image **e)** à l'image **j)**.




## 3. Graphe d'allocation des ressources

La modélisation précédente des interblocages peut être généralisée sous forme d'un graphe d'allocation des ressources d'un système. Ce graphe est composé de N nœuds et de A arcs.

L'ensemble des noeuds 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 (exemplaire) 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 une 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.

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 supprimé.

**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 :

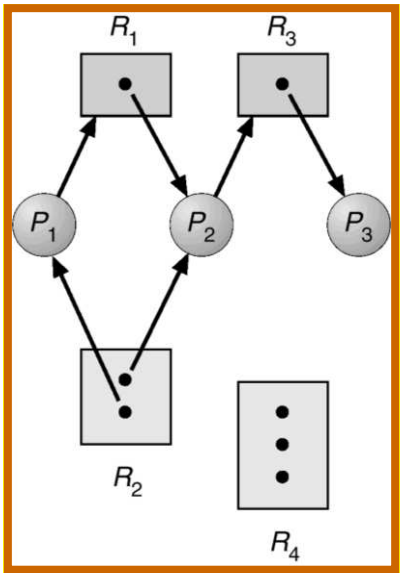



**Figure 4.1 :** Graphe d'allocation des ressources.

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 de R2 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 :

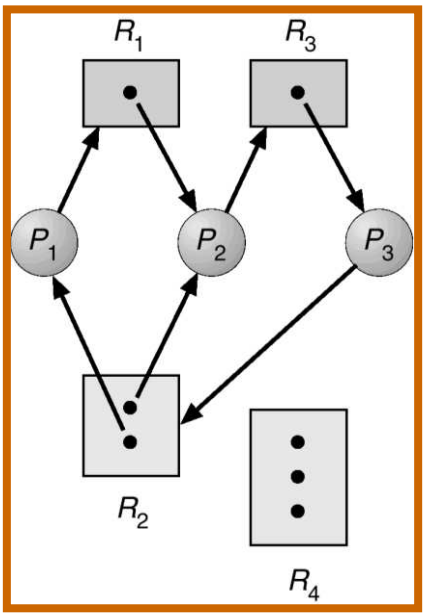



**Figure 4.2 :** Graphe d'allocation des ressources avec circuit et interblocage

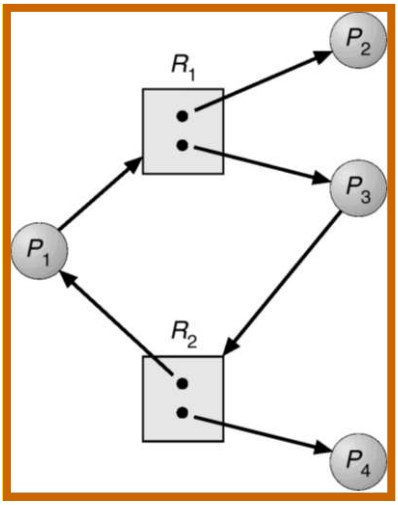



**Figure 4.3 :** Graphe d'allocation des ressources avec circuit sans interblocage

## 4. Méthodes de gestion des interblocages

Il existe essentiellement quatre méthodes pour traiter le problème de l'interblocage : *Prévention, Evitement*, *Détection-Guérison*, *ignorance de l'interblocage.*

- ***Prévention*** : On élimine complètement le risque d'interblocage en faisant en sorte que l'une de ses quatre conditions d'apparition ne soit pas vérifiée.
- ***Evitement*** : On évite le risque d'interblocage en veillant à que le système évolue uniquement entre les états sains.
- ***Détection-Guérison*** : On attend que l'interblocage arrive, on le détecte puis on applique une méthode de guérison.
- ***Ignorance de l'interblocage*** : « Politique de l'autruche »

## 4.1 Prévention des interblocages

Les méthodes de prévention des interblocages s'attaquent aux quatre conditions de l'interblocage en essayant de les éliminer, empêchant ainsi tout interblocage.

### *4.1.1 Exclusion mutuelle*

Le fait de ne pas réserver exclusivement les ressources permet d'éliminer cette condition. 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).

### *4.1.2 Détention et attente*

Pour s'assurer que la condition de détention et d'attente ne se produit jamais dans le système, on doit garantir qu'à chaque fois qu'un processus qui requiert 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 magnétiques 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 imprimé le résultat du fichier sur l'imprimante, il libère ces deux ressources et se termine.

### *4.1.3 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 c, toutes les ressources actuellement allouées 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.

D'autre part, si un processus demande quelques ressources, nous devons d'abord vérifier qu'elles sont disponibles. Si c'est le cas, nous les allouons. Sinon, nous vérifions si elles sont allouées à d'autres processus qui attendent des ressources supplémentaires. Dans ce cas, nous préemptons les ressources désirées du processus en attente et nous allouons au processus demandeur. Si les ressources ne sont pas disponibles, ni détenus par un processus en attente, le processus demandeur doit attendre.

Ce protocole présente un inconvénient majeur:

- ✓ La famine est possible puisqu'un processus peut être retardé indéfiniment parce que l'une des ressources qu'il demande est occupée.

### *4.1.4 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. Un processus demandant une ressource d'ordre inférieure à l'une qu'elle détient se la verra alors refusée. Par exemple, on numérote les types de ressources du système, comme suit :

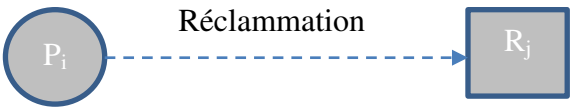
- O(Unité de bandes)=1
- O(Unité de disque)=2
- O(Imprimante)=3
- O(Lecteur CD-ROM)=4
- O(Lecteur Disquette)=5
- … etc.

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).

## 4.2 Evitement des interblocages

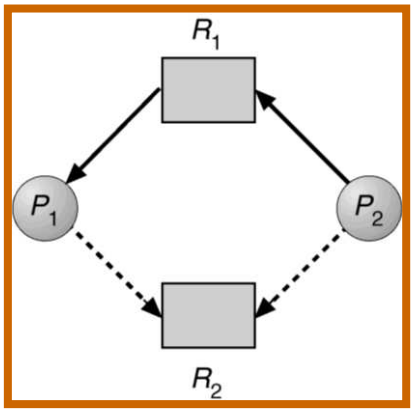
### *4.2.1 Principe de fonctionnement*

Si on dispose d'un système d'allocation de ressources avec une seule instance pour chaque type de ressource, on peut utiliser une variante du graphe d'allocation de ressources pour la prévention des interblocages.
En plus des arcs de requêtes et d'affectation, on introduit un nouveau type d'arc appelé *arc de réclamation*. Un arc de réclamation Pi→Rj indique que le processus Pi peut demander la ressource Rj dans le futur (il est représenté en pointillé).



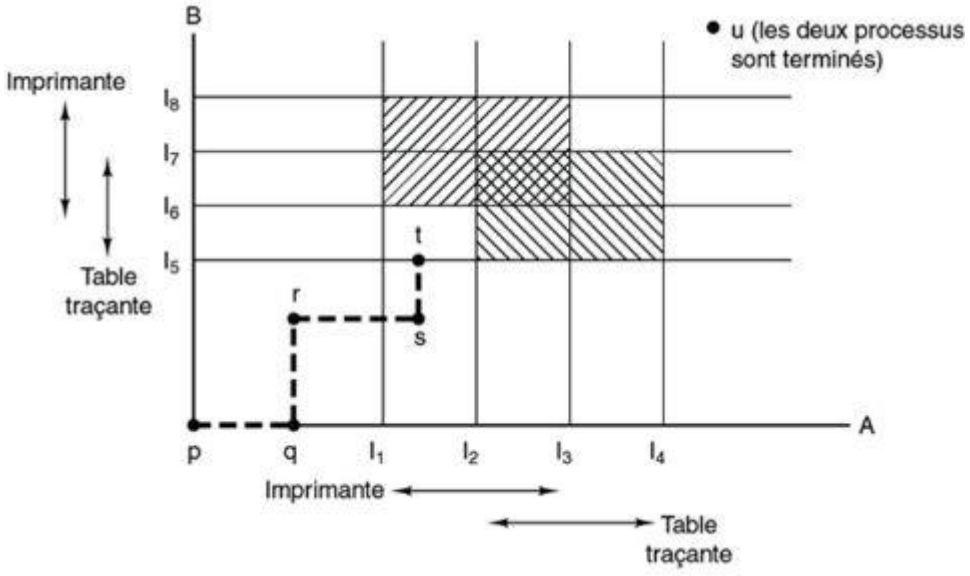

Quand le processus Pi demande réellement la ressource Rj, l'arc de réclamation Pi→Rj est transformé en arc de requête. De même, lorsqu'une ressource est libérée par Pi, l'arc d'affectation Rj→Pi est reconverti en arc de réclamation Pi→Rj.
On peut éviter que l'interblocage arrive en empêchant Pi qui demande la ressource Rj, n'obtienne pas cette ressource si la transformation de l'arc de requête Pi→Rj en arc d'affectation provoque un circuit dans le graphe.
S'il n'existe pas de circuit, l'allocation de la ressource laissera le système dans un état *sain*. Si l'on trouve un circuit, l'allocation de la ressource laissera le système dans un état *malsain*. Le processus Pi devra donc attendre pour que sa requête soit satisfaite.

**Exemple 1** : Soit le graphe d'allocation suivant :



Supposons que P2 demande R2. Bien que R2 soit libre, on ne peut l'allouer à P2, puisque cette action créerait un circuit dans le graphe.

**Exemple 2 :** Soit le schéma de l'allocation des ressources suivante :




- L'axe horizontal représente les instructions exécutées par le processus **A**.
- L'axe vertical représente les instructions exécutées par le processus **B**.
- Au point $\mathbf{I_1}$ le processus **A** demande l'imprimante.
- Au point $\mathbf{I_2}$ le processus **A** demande la table traçante.
- Au point $\mathbf{I_3}$ le processus **A** libère l'imprimante.
- Au point $\mathbf{I_4}$ le processus **A** libère la table traçante.
- Au point $\mathbf{I_5}$ le processus **B** demande table traçante.
- Au point $\mathbf{I_6}$ le processus **B** demande l'imprimante.
- Au point $\mathbf{I_7}$ le processus **B** libère table traçante.
- Au point $\mathbf{I_8}$ le processus **B** libère l'imprimante.
- Les zones grises sont interdites.
- À partir du point **t**, il faut exécuter jusqu'à $\mathbf{I_4}$ sinon il y aura interblocage.

### *4.2.2 Etat sain*

Un état est dit sain (sûr) s'il existe un ordonnancement qui permet à chaque processus de s'exécuter jusqu'au bout, même si chacun demande son maximum de ressources. Un état qui ne répond pas à ce critère est dit non sain (malsain ou non sûr). Pour ces algorithmes, il faut que le nombre maximal de ressources pouvant être demandées par chaque processus soit connu.

***Définition d'un état sain :***

Un système est dans un état sain s'il existe une séquence saine (sûre). 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 un état malsain, c'est à dire qu'il y a un risque d'apparition d'interblocage.

**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 l'on peut allouer immédiatement au processus P1 toutes ses ressources manquantes et les rendre ensuite. Dans ce cas le système disposerait de 5 unités de bandes, le processus P0 peut donc obtenir toutes ses ressources manquantes et les rendre ensuite. Le système disposerait alors de 10 unités de bandes et 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 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 6 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.

### *4.2.3 Algorithme du Banquier*

L'algorithme du banquier (Habermann et Dijkstra) est un algorithme d'évitement des interblocages qui s'applique dans le cas général où chaque type de ressources possède plusieurs instances. 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 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 **n x m** 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 **n x m** 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 **n x m** 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 **n x m** 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** :

$$\mathbf{Need[i, j]=Max[i, j] - Allocation[i, j]}$$

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

- **$\text{Allocation}_i$**:vecteur désignant les ressources actuellement allouées au processus Pi.
- **$\text{Need}_i$** : vecteur désignant les ressources supplémentaires que le processus Pi peut encore demander.
- **$\text{Request}_i$** : vecteur désignant les ressources supplémentaires que le processus Pi vient de demander.
- **X<=Y** : X et Y sont des vecteurs 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 demande d'allocation des ressources et un algorithme de vérification si l'état du système est sain.
Nous donnons maintenant le code de chacune des deux parties.

### Algorithme d'allocation des ressources

/* Cet algorithme est appelé à chaque fois qu'un processus fait une demande de ressources
Début
**Etape 1**: Si $\text{Request}_i <= \text{Need}_i$
Alors Aller à l'étape 2
Sinon erreur : le processus a excédé ses besoins maximaux
Finsi
**Etape 2**: Si $\text{Request}_i <= \text{Available}$
Alors Aller à l'étape 3
Sinon Attendre : les ressources ne sont pas disponibles.
Finsi

**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
Finsi

Fin.

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[i] :=Faux; (pour tout i : o .. n-1)

**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 : o .. n-1)
Alors Verification_Etat_Sain :=Vrai
Sinon Verification_Etat_Sain :=Faux
Finsi

Fin.

Remarque: Cet algorithme peut demander **$mxn^2$** opérations pour décider si un état est sain.

**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 | 4 | 3 |
| P1 | 1 | 2 | 2 |
| P2 | 6 | 0 | 0 |
| P3 | 0 | 1 | 1 |
| P4 | 4 | 3 | 1 |

***Matrice : Available***

| A | B | C |
|---|---|---|
| 3 | 2 | 2 |

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

Le système est dans un état sain puisque la séquence <P1, P3, P4, P2, P0> est saine.

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 : $Request_1$=(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 | C |
|---|---|---|
| 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 obtenu est malsain.

**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 $mxn^2$ opérations. (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).

## 4.3 Détection et guérison des interblocages

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.

### *4.3.1 Détection de l'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 alloué au processus Pi k instances du type de ressources Rj.
- **Request** : C'est une matrice nxm indiquant les ressources supplémentaires que les processus viennet 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 Finish[i] :=Faux
Sinon Finish[i]:=Vrai
Finsi
Fait ;

**Etape 2** : Trouver un indice i tel que : Finish[i]=Faux et $Request_i$<=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[i]=Faux (pour un certain i)
Alors Le système est dans un état d'interblocage
Sinon Le système n'est pas dans un état d'interblocage
Finsi
Fin.

**Exemple** : 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 :

**Itération : 1**

Work

| 5 | 2 | 3 | 1 |
|---|---|---|---|

Finish

| Faux | Faux | Faux |
|---|---|---|

Indice choisi i= 0 ;

**Itération : 2**

Work

| 6 | 2 | 4 | 1 |
|---|---|---|---|

Finish

| Vrai | Faux | Faux |
|---|---|---|

Indice choisi i= 1 ;

**Itération : 3**

Work

| 8 | 2 | 4 | 2 |
|---|---|---|---|

Finish

| Vrai | Vrai | Faux |
|---|---|---|

Indice choisi i= 2 ;

**Itération : 4**

Work

| 8 | 3 | 6 | 2 |
|---|---|---|---|

Finish

| Faux | Faux | Faux |
|---|---|---|

Le vecteur Finish a la valeur Vrai, donc le système n'est pas en situation d'interblocage.

Supposons maintenant que le processus P0 fait une demande supplémentaire de 3 instances de la ressource B, et P2 demande en plus 4 instances de la ressource C. 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 | 3 | 0 | 1 |
| P1 | 2 | 0 | 1 | 0 |
| P2 | 2 | 1 | 4 | 0 |

***Matrice : Available***

| A | B | C | D |
|---|---|---|---|
| 5 | 2 | 3 | 1 |

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

Itération : 1

Work

| 5 | 2 | 3 | 1 |
|---|---|---|---|

Finish

| Faux | Faux | Faux |
|---|---|---|

Indice choisi i= 0 ;

Itération : 2

Work

| 7 | 2 | 3 | 2 |
|---|---|---|---|

Finish

| Faux | Vrai | Faux |
|---|---|---|

Indice choisi i= 1 ;

Le déroulement de l'algorithme s'arrête avec le vecteur Finish contenant deux valeurs fausses: P0 et P2 sont en interblocage.

### *Critique de l'algorithme de détection :*

L'algorithme de détection des interblocages est très coûteux s'il est exécuté après chaque demande de ressources. En effet, il sera aussi coûteux que l'algorithme d'évitement du Banquier. L'idée donc c'est de le lancer périodiquement, mais comment choisir cette période ?.

### *4.3.2 Guérison de l'interblocage*

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.

### 4.4 Ignorance de l'interblocage : « Politique de l'autruche »

Tous les mécanismes dédiés au traitement des interblocages présentent des inconvénients. La vérité est qu'il n'existe aucun moyen efficace de traiter les interblocages. Pour la plupart des systèmes d'exploitation, l'apparition d'interblocages est fort rare. Par conséquent, dans de nombreuses situations, le problème des interblocages est ignoré, à l'instar d'une autruche qui s'enfonce la tête dans le sable en espérant que le problème disparaisse. **C'est-à-dire, que l'algorithme de l'autruche consiste à accepter les interblocages et n'avoir aucun plan pour les régler.**

- Windows et UNIX adoptent cette stratégie
- Raisonnable ssi:
  - Les interblocages surviennent rarement
  - Le coût de prévention contre les interblocages est élevé

**Exemple:**

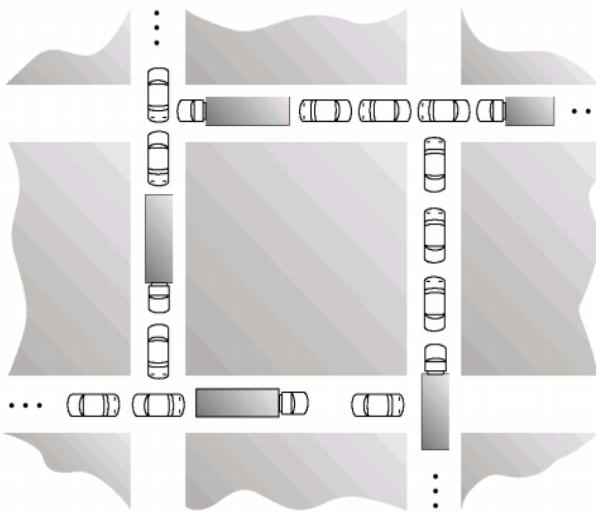
Supposons que la taille de la table de processus dans UNIX est égale à 100

- 10 processus s'exécutent, chacun doit créer 12 processus enfants.
- La table est pleine dès que chaque processus a fini la création de son 9éme enfant.
- Par la suite échec du *fork* ( la méthode fork() est pour créer des processus fils sous l'Unix/Linux).
- Solution: pour résoudre l'interblocage, chaque processus retente après un temps aléatoire.

## 5. Questions

**Exercice 1 :**

L'embouteillage (gridlock) est un problème de circulation (voir la figure) dans lequel aucun mouvement n'est possible dans le trafic. Démontrer que les quatre conditions de l'interblocage sont vérifiées.



**Exercice 2:**

| Processus | Allocation | | Max | | Available | |
|---|---|---|---|---|---|---|
| | R1 | R2 | R1 | R2 | R1 | R2 |
| P1 | 1 | 2 | 4 | 2 | 1 | 1 |
| P2 | 0 | 1 | 1 | 2 | | |
| P3 | 1 | 0 | 1 | 3 | | |
| P4 | 2 | 0 | 3 | 2 | | |

L'état du système décrit dans le tableau est-il sain ? (si oui, justifier par une séquence saine).

**Exercice 3:**

Considérons le tableau d'utilisation des ressources suivant :

| Processus | Allocation | | | Max | | | Request | | | Available | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | R1 | R2 | R3 | R1 | R2 | R3 | R1 | R2 | R3 | R1 | R2 | R3 |
| P1 | 2 | 0 | 0 | 2 | 0 | 1 | 1 | 0 | 0 | 0 | 2 | 0 |
| P2 | 1 | 2 | 0 | 2 | 5 | 2 | 0 | 0 | 1 | | | |
| P3 | 0 | 1 | 1 | 1 | 4 | 2 | 0 | 0 | 0 | | | |
| P4 | 0 | 0 | 1 | 2 | 0 | 1 | 0 | 0 | 1 | | | |

1) Dessiner le graphe d'allocation des ressources.
2) Dessiner le graphe réduit d'allocation des ressources.
3) A partir de ce graphe réduit, le système est-il en interblocage ?
4) Le système est-il sain? justifier
5) Utilisant l'algorithme de détection, montrer s'il y a un interblocage ou non.

## 6. Références


- Tanenbaum, « Modern operating systems », third edition, Pearson, 2014
- Tanenbaum, « Systèmes d'exploitation », Dunod, 1994.
- Crocus, « Systèmes d'exploitation des ordinateurs », Dunod,1993.
- Sacha Krakowiak, « Principes des systèmes d'exploitation des ordinateurs », Dunod, 1993
- Cours du Pr. Mourad LOUKAM, https://www.loukam.net/index_Courses.html
- http://www.uqac.ca/pguerin/8INF341/Cours9_Interblocage.html

# 2eme Partie: la Pratique

Chapitre V

# Généralités sur Linux

## Objectifs

- Apprendre l'histoire de Linux
- Quelles sont les caractéristiques et fonctions principales de Linux

## Themes couverts

- Unix et ses caractéristiques
- Linux et son histoire
- Distributions de linux
- Acces a la machine sous linux
- Shell

# Chapitre 5: Généralités sur Linux

## 1. Introduction

Un système d'exploitation est un logiciel qui gère directement les composants physiques du système ainsi que ses ressources, telles que le processeur, la mémoire et le stockage. Il représente l'interface entre les applications et le matériel. Parmi plusieurs systèmes d'exploitation, vient Linux avec une large utilisation à travers le monde.
Linux ou GNU/Linux est une famille de systèmes d'exploitation open source de type Unix fondé sur le noyau Linux, créé en 1991 par Linus Torvalds. De nombreuses distributions Linux ont depuis vu le jour et constituent un important vecteur de popularisation du mouvement du logiciel libre.

## 2. Unix

Linux est un système d'exploitation qui est un ensemble de programmes permettant l'utilisation de l'ordinateur et la gestion de ses ressources (processeurs, mémoires, disques, périphériques, communication inter-processus et inter-machines, …etc.). Pour présenter Linux nous allons commencer par donner une brève histoire de son ancêtre UNIX.

### 2.1.Historique

- Unix est né aux laboratoires Bell en 1969, Développé par **Ken Thompson et Dennis Ritchie** (le premier à avoir développé le langage C).
- En 1973, Unix a pu être réécrit, presque entièrement, en C (pour faciliter la Portabilité)
- Code source vendu à un prix bas aux sociétés.
- Plusieurs sociétés ( IBM , Sun ,…) se sont intéressées au système et elles ont repris son développement pour avoir leur propre version (Solaris:Sun , AIX:IBM, HP-UX: HP, FreeBSD:Université de Berkely ,…)

### 2.2.Caractéristiques d'Unix

- Basé sur le principe tout est fichier.
- multi-tâches en temps partagé
- multi-utilisateurs
- Utilisation du **Shell** comme interpréteur de commandes
- La configuration du système est stockée sous forme de texte
- Disponibilité sur un large gamme d'architecture matérielle (du PC jusqu'au Super calculateur massivement parallèle)
- C'est le système le plus utilisé (Dans les universités, les centres de recherches, les serveurs d'Internet, …)
- Unix produit commercial (système payant)

**Remarque** : L'objectif des chercheurs dans le temps était de rendre UNIX accessible sans frais.

# 3. Linux

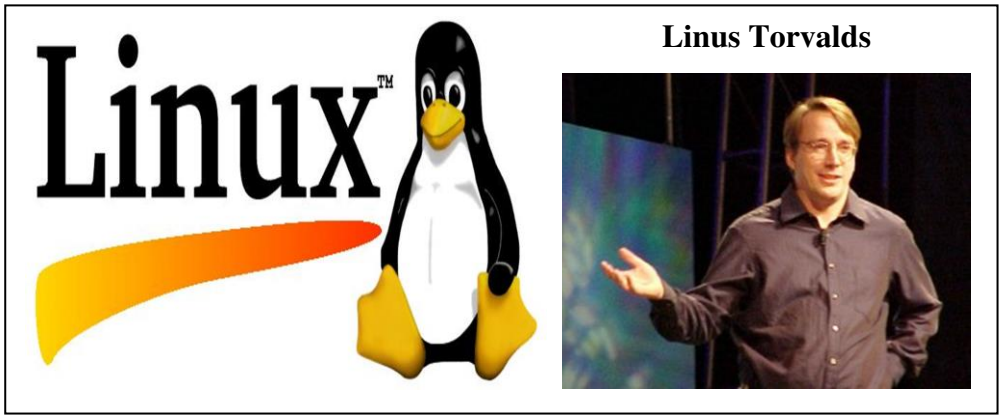
Linux est une version librement diffusable de l'Unix.

## 3.1.Historique

- En 1991, En Finlande un autre noyau était disponible : c'est « Linux » par **Linus Torvalds (jeune étudiant à l'université d'Helsinki, né le 28/12/1969)**
- Des milliers de personnes participent à son développement
- Code source est disponible sur Internet (logiciel libre et gratuit)

**Remarque :** Le nom LINUX peut signifier, au choix :

- LINUs uniX
- Linux Is Not UniX




## 3.2.Logiciel libre

| Logiciel libre = Logiciel Open Source |
|---|

Un logiciel libre est un logiciel dont la licence prévoit:

- La disponibilité des codes sources.
- La possibilité de modifier, améliorer, adapter le logiciel sous réserve que ces modifications soient rendues accessibles à toute personne intéressée.
- La possibilité de copier et de diffuser le logiciel sous réserve que les termes de la licence ne soient pas modifiés.

**Exemples de logiciels libres**

Linux, OpenOffice , Apache, Sendmail, MySQL, PostgreSQL, gcc, PHP, ...

# 4. Distributions de Linux

Le noyau (ou kernel) développer par Linus Torvalds est l'élément essentiel de toutes les **distributions Linux** existantes.

Chaque distribution essaye d'offrir de la valeur ajoutée sous la forme des outils d'**installation** et d'**administration**.

| **Une distribution Linux** = noyau + outils d'installation + outils d'administration + un ensemble de logiciels d'application |
|---|

### 4.1.Exemples des distributions

- Red Hat Linux (USA www.redhat.com) [05 CD]
- Mandrake Linux (France www.linux-mandrake.com)
- SuSE Linux (Allemagne www.suse.com)
- Debian (Internet www.debian.org) [23 CD d'installation]
- Fedora:(www.fedora.org) [ version libre de Red Hat ]
- Ubuntu(www.ubuntu.com) [version de debian]
- KNOPPIX, slackware, Mandriva…

### 4.2.Domaines d'utilisation de Linux

- Station de travail : Multimédia et bureautique (openoffice, koffice,...)
- Réseaux et Internet : serveur Web (Apache), messagerie (sendmail), Explorateur (FireFox de Mozila)….
- Développement : C/C++, Delphi, Java, PHP,…
- SGBD ( Oracle, Informix, MySQL, PostgreSQL,…)
- Recherche scientifique
- …

## 5. Accès à la Machine sous Linux

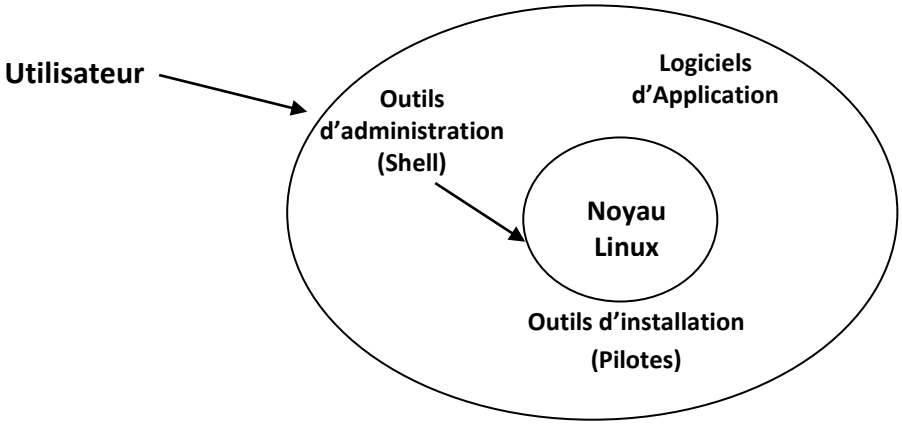
Le système linux étant un système multi-utilisateur, pour y a accéder on doit entrer un nom et mot de passe utilisateur.

a) **Système Multi-Utilisateur :**
- Utilisateur simple : droits restreints et chaque utilisateur possède un répertoire de travail.
- Super Utilisateur ou Administrateur (root) : tous les droits.

b) **Organisation d'utilisateurs en groupes:** chaque utilisateur doit appartenir à un groupe

c) **Connexion au système :** login/passaword requis et la connexion se fait de deux mode différents :
- Mode texte ou console (exécution des commandes Shell)
- Mode graphique (à l'aide du serveur de graphisme X-Window)

### 5.1.Position du noyau Linux dans une distribution



- Noyau Linux : ensemble de programmes pour gérer la machine et ses ressources
- Shell : Interpréteur de commandes
- Pilotes: Drivers des périphériques (disques durs, carte graphique, carte réseau, imprimantes,…)
- Logiciels d'applications : différents logiciels
  - X-Window : pour fournir un environnement de graphisme
  - KDE et Gnome : pour gérer l'interface graphique (fenêtres,…)
  - …

### 5.2.Le Shell

C'est l'interface entre les utilisateurs et le système. Le Shell est un interpréteur qui exécute les commandes une à une après traduction de l'instruction (écrit en langage évolué) en langage interne (langage machine).

**Versions du Shell :** sh, ksh, bash,chs,…

**La distribution Red Hat:**

a) **Connexion en mode graphique :** voir interface graphique
b) **Connexion en mode texte :** Si les login/password sont valides alors un message de la forme suivante sera affiché:

**[user@machine ~ ]$**

- user: le nom de l'utilisateur connecté
- machine : le nom de la machine
- ~ : caractère spécial désigne le répertoire de travail de l'utilisateur connecté
- Le signe $ : indique que vous êtes en un simple utilisateur
- Le signe # : indique que vous êtes en mode super-user.

**L'invité de commandes (utilisation du Shell)**

**Syntaxe d'une commande :**

**Commande [option] [paramètres]**

**Option** : non nécessaires pour exécuter la commande
**Paramètres** : arguments nécessaires pour exécuter la commande

- Les trois champs sont séparés par des espaces
- Plusieurs commandes sur la même ligne : séparer par ' ;'

**Quelques commandes :**

- **date** : Afficher la date et l'heure
- **cal** : afficher un calendrier
- **uname** : afficher le nom et les caractéristiques du système.
- **passwd** : modifier son mot de passe
- **man** nom-commande :

- manuel pour afficher une page d'aide (forme de commande, option,...)
- exemple : **man man** : plus d'information sur l'utilisation du man

**Remarque** : **commande --help** ou **commande -h** : une option commune à la majorité des commandes pour afficher l'aide (exp : ls --help, man -h)

- **adduser** user1 **:** ajoute l'utilisateur user1
- **passwd** user1 **:** pour donner ou changer le mot de passe user1
- **addgroup** grp1 **:** ajoute le groupe grp1
- **adduser** user1 grp1 : ajoute l'utilisateur user1 au groupe grp1
- **groupadd** grp2 : crée un groupe grp2
- **chgrp** grp2 user1 : le groupe de user1 est maintenant grp2
- **userdel** : pour supprimer un utilisateur ;
- **groupdel** : pour supprimer un utilisateur ;
- **adduser –ingroup** grp1 user1 : crée un utilisateur user1 dont le groupe principal est grp1

(consulter les fichiers **: /etc/passwd, etc/group, /etc/shadow**)

- **su** user1 **:** bascule vers l'utilisateur user1
- **su - :** bascule vert le super-utilisateur (administrateur)
- **ifconfig eth0 192.168.10.10 :** configure la carte reseau ethernet eth0 et lui affecte l'adresse indiquée.
- **mount –t vfat /dev/hdc1 /mnt/hd1** : montage de la partition 1 du disk 3 sur le répertoire /mnt/hd1, si le point de mount est déclare dans /etc/fstab, l'utilisateur peut le monter, s'il lui est permis. Autrement il faut être (**unmount** pour démonter)

**Exemple** : pour lire un CDROM

- Consulter le fichier fstab (par **cat /etc/fstab) pour savoir si le point de montage (**dans notre cas, c'est /dev/hdc)
- Créer un répertoire : **mkdir tt** (tt répertoire pour contenir le point de montage)
- Monter par **: mount /dev/hdc tt**
- Accéder au répertoire tt pour utiliser votre CDROM

**N.B :** il est déconseillé de se loger en superutilisateur (root). Si à un moment donné nous avons besoin de lancer une commande d'administrateur, il suffit d'utiliser « su –» (switch user), qui permet de passer momentanément en root.

## 6. Questions

a) Quelle est la différence entre UNIX et LINUX ?
b) Qu'est-ce que le BASH et la différence avec le DOS?
c) Que sont les démons « daemons » ?
d) Comment passer d'un environnement de bureau à un autre, comme passer de KDE à Gnome ?
e) Comment monter un flash disk ?

## 7. Références


- Michel Divay, « Unix, Linux et les systèmes d'exploitation : cours et exercices corrigés », 2004.
- Nicolas Pons, « Linux - Principes de base de l'utilisation du système », Editions ENI, 6° édition, 2018.
- A.Belahcene et S.Khelifati, « Administration et réseau sous Linux », polycopié de cours.
- https://openclassrooms.com/fr/courses/43538-reprenez-le-controle-a-laide-de-linux

Chapitre VI

# Prise en main avec Linux

## Objectifs

- Apprendre administrer une machine Linux via les commandes de base

## Themes couverts

- Arborescence du systeme de fichiers sous Linux
- Principales commandes de base
- Liens de Fichiers

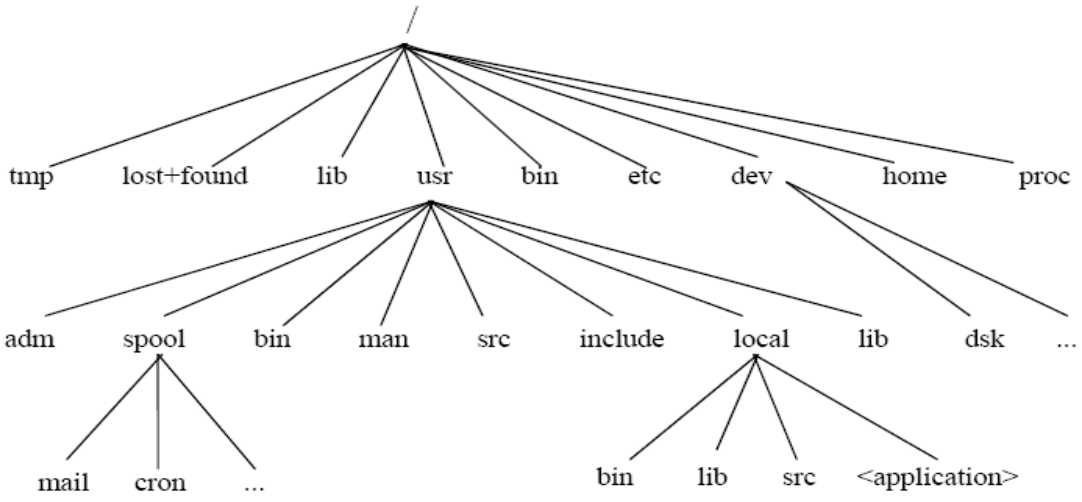
# Chapitre 6: Prise en main avec Linux

## 1. Introduction

Linux est un système d'exploitation que l'on peut utiliser en ligne de commandes à partir d'un terminal. Dans ce chapitre, nous décrivons l'utilisation des principales commandes Shell Linux. Il s'agit de l'essentiel des commandes pour administrer une machine Linux dans toutes les distributions.

## 2. Arborescence du système de fichiers

La structure hiérarchique des fichiers et répertoires d'une machine Linux est comme représentée dans la figure suivante :




- La racine est dénoté par `/`
- Les chemins sont séparés par `/`
- Les noms des objets sont des séquences de maximum 255 caractères sensibles à la casse.
- Il est préférable de ne pas utiliser les caractères : ?, *, &. `, ´,“,<,>
- Il convient de se limiter à: - A à Z , a à z, 0 à 9, le caractère de soulignement ` _`, le tiret `-` et le point `.`
- Le point ( . ) comme premier caractère d'un nom signifie un fichier caché.
- Éviter des noms contenant des caractères accentués ou des blancs.
- Le point ( . ) ne sépare pas forcement le nom de son extension
  (Exemples : rapport.ps.gz, postgresql-2.3.6-src.tar.Z, README)

**Les principaux répertoires**

- /boot : contient les fichiers nécessaires au démarrage du système.
- /bin : contient les commandes de base (binaire)
- /sbin : ontient les commandes du super utilisateur (administrateur)
- /usr/bin, /usr/sbin,…: contient des fichiers binaires
- /etc : contient les fichiers de configuration du système et des services
- /home : contient les répertoires personnels des utilisateurs simples
- /root : contient le répertoire personnel de l'administrateur.
- /usr : contient le reste des programmes du système et les logiciels.
- /dev: contient les noms périphériques.
- /mnt/ , /media/ : contient les points de montage d`autres systèmes de fichiers.
- /proc : est un répertoire virtuel, n'a pas d'existence sur disque, contient les informations sur les processus. Faire par exemple cat /proc/meminfo, pour des informations sur la mémoire.

## 3. Principales Commandes de Base

Dans ce qui suit, nous allons présenter les principales commandes pour l'utilisation d'une machine Linux.

**La commande man**

**syntaxe** : man [options] nom_de_commande
**description** : Elle permet d'afficher un manuel en ligne sur la commande spécifiée. Cet affichage s'effectue en mode console. La commande **q** permet de quitter le manuel.
**options** : -h : affiche l'aide concernant la commande **man** et donc permet notamment de savoir que **q** permet de quitter.
-w or --path : affiche le lieu où est stocké le fichier contenant le manuel de la commande.

**La commande ls**

**syntaxe** : ls [options] [chemin_d'un_repertoire]
**description** : Affiche l'ensemble des fichiers passés en argument puis la liste des fichiers contenus dans les répertoires passés en argument. Par défaut, elle affiche le répertoire courant ".".
**options** : -l : permet d'obtenir des informations détaillées sur chaque fichier listé (date, taille, droits, etc.).
-a : permet d'afficher tous les fichiers contenus dans un répertoire, y compris les fichiers commençant par un point(les fichiers cachés).
-R : permet d'afficher récursivement le contenu des sous-répertoires.
--color , --color=yes: permet d'afficher les fichiers en couleur selon leur type.
--color=no : annule l'affichage en couleur des fichiers.
**exemples : ls . :** lister le répertoire courant
ls .. : lister le répertoire parent
ls -lrt /etc : Donne la liste des fichiers et répertoires de /etc avec les détails (option -l) trie par temps (-t) en ordre inverse (-r)
ls -l > f1.txt: le symbole « > » redirige la sortie de « ls » vers le fichier f1.txt avec écrasement. Si on utilise « > > » il y aura ajout à la fin (append).

**La commande cd**

**syntaxe** : cd [chemin]

**description** : Elle permet de se déplacer dans l'arborescence du système de fichier.
Le chemin peut-être absolu ou relatif.
**exemples** : cd .. : permet de remonter d'un cran dans l'arborescence.
cd ou cd ~ : permet de se placer directement à la racine de votre répertoire de travail.
cd - : pour retourner au répertoire précédent
cd /usr/local : déplacement selon un chemin absolu.
cd bin : déplacement selon un chemin relatif.
**remarques** : - chemin absolu : chemin d'accès complet à partir la racine (/)
Par exemple : /usr/local/seminaires/apprentissage/text1
- chemin relatif : spécifier le chemin d'accès relativement au répertoire courant
Par exemple : seminaires/apprentissage/text1 représente le nom relatif de text1 par rapport au répertoire courant /usr/local

**La commande mkdir**

**syntaxe** : mkdir [options] repertoire
**description** : Elle permet de créer un nouveau répertoire.
**options**: -p, --parents : s'assure que chaque répertoire spécifié existe et crée les répertoires parents manquants.
**exemples** : mkdir -p Informatique/MuPAD : permet de créer le répertoire Informatique et le sous répertoire MuPAD.

**La commande pwd**

**syntaxe :** pwd
**description :** Elle permet de connaître le répertoire courant.

**La commande rmdir**

**syntaxe :** rmdir [options] repertoires
**description :** Elle permet de supprimer un ou plusieurs répertoires vides.
**options :** -p, --parents : efface les répertoires parents s'ils deviennent vides.

**La commande rm**

**syntaxe** : rm [options] nom
**description** : Elle efface chaque fichier spécifié et par défaut n'efface par les répertoires.
**options** : -f, --force : efface les fichiers en ignorant ceux qui n'existent pas et en ne demande de confirmation à l'utilisateur.
-i, --interactive: demande à l'utilisateur de confirmer chaque suppression.
-r, -R, --recursive : supprime récursivement les contenus des répertoires et le répertoire lui-même.

**La commande cp**

**syntaxe** : cp [options] fichier1 fichier2
cp [options] fichier repertoire
**description** : Elle sert à copier des fichiers et éventuellement des répertoires depuis un endroit précis vers une destination précise ou un répertoire.
**options** : -i : interroge l'utilisateur avant d'écraser les fichiers existants.
-R : copie récursivement les répertoires et gère correctement les fichiers spéciaux.
-f : force l'effacement des fichiers cibles existants.
-p : conserve le propriétaire, le groupe, les permissions d'accès, et les horodatages du fichier original.

**La commande mv**

**syntaxe** : mv [options] source destination
mv [options] source repertoire
**description** : Elle sert à déplacer ou renommer les fichiers. Si le dernier argument est un nom de répertoire alors tous les fichiers sources seront déplacés, en conservant leur nom, vers ce répertoire sinon il déplacera le premier pour remplacer le second.
**options** : -i : interroge l'utilisateur avant d'écraser les fichiers existants.
-f, --force : écrase les fichiers de destination existants sans demander de confirmation à l'utilisateur.
-u, --update : ne pas déplacer un fichier régulier qui écraserait un fichier destination existant ayant une date de modification plus récente.

**La commande touch**

**syntaxe :** touch nom_fichier
**description :** Elle permet de créer un fichier.
**exemple :** touch linux.txt
ls
linux.txt

**La commande file**

**syntaxe :** file nom_fichier
**description :** Elle permet de connaître le type d'un fichier. L'extension des fichiers n'est pas toujours utilisée.

**La commande cat**

**syntaxe :** cat fichier1 fichier2...
**description :** défiler à l'écran le contenu d'un ou plusieurs fichiers.
**remarques** : - **<ctrl-c> :** Mettre fin au défilement
- **<ctrl s> :** Interrompre le défilement avec possibilité de reprise
- **<ctrl q> :** Reprendre le défilement

**exemple** : **cat /etc/passwd | grep -v bash:** Le symbole « | », pipe permet de récupérer la sortie d'une commande pour l'envoyer en entrée dans la suivante. Ici le contenu du fichier passwd est récupéré puis filtré avec **grep** pour garder les lignes ne contenant pas le mot « bash ».
**cat /etc/profile > ~/confenv** : pour copier le contenu du fichier profile dans le fichier confenv de son répertoire HOME.
**Ls *.txt | xargs grep –l ce_mot :** rechercher les fichiers ayant l'extension txt qui contiennent le texte ce_mot
**cat *.txt | grep ce_mot** : rechercher les lignes contenant ce_mot dans tous les fichiers ayant l'extension txt.

**La commande more**

**syntaxe** : more fichier
**description :** défiler le contenu une page d'écran à la fois.
**remarques : -** taper la barre d'espacement : Défiler la page suivante
- taper <entree> : Défiler une ligne supplémentaire la touche
- taper la lettre q : Quitter la commande **more**

**Les commandes less, head et tail**

**Less :** similaire à la commande more.

**head :** afficher les dix premières lignes d'un fichier
**tail :** les dix dernières.
**exemple** : tail -15 file.txt: Donne les 15 (les 10, si rien n'est indiqué) dernières lignes du file.txt, la commande **head** est pareille pour le début du fichier.

**Les commandes nl et wc**

**nl** : permet la numérotation des ligne d'un fichier donné
**wc** : permet le calcul du nombre de ligne, de mot et de caractère contenu dans un fichier.
**exemples**: - nl /etc/passwd
- wc /etc/mail/sendmail.mc

**Commandes pour les process (processus : programme en exécution)**

**ps aux** **:** liste tous les process qui tournent sur la machine.
**kill -9 2345:** tue le process dont le numéro est 2345, (s'il vous appartient).
**top** : donne l'utilisation des ressources mémoire et CPU, en réel.
**free** : affiche les informations sur la mémoire

**Commandes pour éditer les fichiers**

**nano** nom_fichier
**vi** nom_fichier
**emacs** nom_fichier ( voir aussi **gedit** )

**Commandes pour modifier les droits d'accès**

**chmod** **:** change les droits des répertoires et fichiers (chmod xxx)
**chown** **:** change le propriétaire et le groupe
**exemples : chmod o+w lmd:** permet de changer les permissions, ici par exemple on ajoute « w », permission d'écriture aux autres (other), g-x : enlève la permission d'exécution au groupe. ou bien avec des valeurs par exemple **chmod 756 lmd veut dire rwxr-xrw-** pour lmd**.**
**chown lmd :mongroup f1.txt:** change le propriétaire et le groupe du fichier f1.txt deviennent lmd et mongroup.

**Autres commandes utiles**

**echo ''message''**: renvoie l'écho message sur écran.
**echo $HOME**: donne le contenu de la variable d'environnement HOME (attention au $, qui doit précéder toute variable shell).
**echo $PATH** : indique les chemins déclarés pour accéder aux librairies et binaires.
**locate**: localise un fichier dans la base de données des fichiers
**find –name nom_fich** : recherche le fichier nom_fich
**gzip fich.txt**: compresse le fichier fich.txt et sera fich.txt.gz. Pour l'inverse, utiliser **gunzip**.
**tar -zvf dir.tgz ladir:** tare et compresse les fichiers de la diretory « ladir » sous le nom dir.tgz, avec x : pour extraire et t pour lister ou bien tester
**su**: switch user, on peut aussi utiliser « sudo ».
**df**: disk free, donne la taille des partitions montées, option -h : human readable (lecture appropriée)
**du /home**: disk used, donne la utilisée par le répertoire /home. On peut aussi utiliser l'option –h
**tty**: donne le terminal en cours d'utilisation. Le premier est /dev/pts/0, ensuite /dev/pts/1, …etc , on peut faire sortir le résultat d'une commande dans un autre terminal. Par exemple si on est dans terminal 1, alors ls > /dev/pts/2 sort le résultat dans le deuxième terminal (console).
**who** : indique qui est connecté et sur quelle console (tty)
**tree -d -L 2 / > tree.txt**: permet de sortir l'arborescence du système et de stocker le résultat dans

le fichier tree.txt
**diff f1 f2**: donne la différence entre les fichiers f1 et f2. diff3 : donne la différence entre 3 fichiers.
**Le fichier caché « .bash_history »**: Contient l'historique des commandes exécutées, périodiquement mis à jour. les éléments les plus anciens sont supprimés. La même information peut être obtenue avec la commande **history.**
**ps2pdf**: Convertit le fichier du format ps en format pdf. **pdf2ps** fait l'inverse.
**convert f1.gif f2.jpg** : convertit le fichier image de format gif en image jpeg. Les extensions déterminent le type de conversion à faire. Cet outil fait partie du logiciel imagemagik.

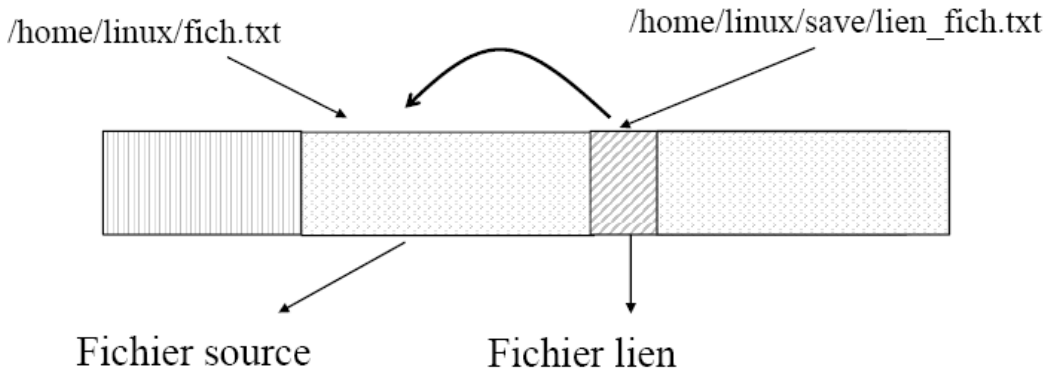
## 4. Liens de Fichiers

Les liens sont utiles pour faire apparaître un même fichier dans plusieurs endroits, même avec des noms différents. il existe deux types de liens, à savoir :

a) Lien symbolique
b) Lien physique

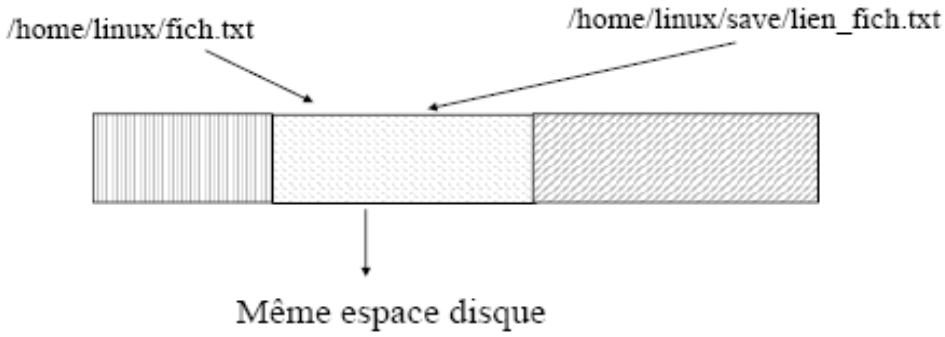
### 4.1. Liens symboliques

- Fait référence à un fichier dans un répertoire.
- Si suppression du fichier source alors le lien sera considéré comme "cassé".
- Utile dans le cas des fichiers binaires (commandes)




### 4.2. Liens physiques

- Associe deux ou plusieurs fichiers à un même espace disque
- Les deux fichiers restant indépendants.
- Le fichier sera supprimé seulement si tous ces liens sont supprimés
- Utile dans le cas des fichiers de données




### 4.3. Comment créer un lien

A l'aide de la commande « ln »

**La commande ln**

**Description :** c'est la commande de création des liens
**Syntaxe** : **ln** [option] source lien
- Cas des liens physiques : ln linux.txt linux1.txt /*linux1.txt est un lien physique*/
- Cas des liens symboliques : ln -s linux.txt linux1.txt /*linux1.txt est un lien symbolique*/
  ln -s /home/linux/linux.txt rep/linux1.txt

## 5. Questions

a) Donner le nombre de répertoires ou de fichiers, ou liens de la directory /usr/lib, lister les fichiers.
b) Reprendre la question précédente et donner la taille totale des fichiers.
c) Dans quelle partition vous êtes, combien y en a t-il dans votre système, quelle est la taille de votre installation ?
d) Quelle est la taille de la mémoire étendue utilisée de votre machine.
e) Quelle est votre chemin (PATH), compléter le avec le répertoire courant, faire l'export vers les autres terminaux?, quel est votre répertoire courant ? Rendre votre répertoire accessible uniquement a votre groupe, pas de lecture, ni d'écriture.
f) Prendre votre nouveau PATH, donner un message à chaque connexion (nouveau shell)

## 6. Références


- Michel Divay, « Unix, Linux et les systèmes d'exploitation : cours et exercices corrigés », 2004.
- Nicolas Pons, « Linux - Principes de base de l'utilisation du système », Editions ENI, 6° édition, 2018.
- A.Belahcene et S.Khelifati, « Administration et réseau sous Linux », polycopié de cours.
- https://openclassrooms.com/fr/courses/43538-reprenez-le-controle-a-laide-de-linux
- https://www.malekal.com/liste-des-commandes-linux/

Chapitre VII

# Travaux Pratiques

## Objectifs

- Pratiquer les notions theoriques de la 1ere partie par des programmes écrits en langage C et Java.

## Themes couverts

- Fiche TP N° 01 « Linux : Rappel sur les commandes de base et droits d'accès »
- Fiche TP N° 02 « Programmation en langage C sous Linux »
- Fiche TP N° 03 « Manipulation des Processus et Threads en langage C »
- Fiche TP N° 04 « Threads et Sémaphores avec API POSIX »
- Fiche TP N° 05 « Gestion des Threads en langage Java »
- Fiche TP N° 06 « Synchronisation des Threads avec les Moniteurs en Java »
- Fiche TP N° 07 « Etude de cas : Les Threads pour Gérer un Restaurant »

# Chapitre 7: Travaux pratiques

Dans ce qui chapitre, nous allons présenter une série de travaux pratiques pour mettre en œuvre les notions et les concepts de base étudiés dans la première partie de ce document. Ces travaux pratiques sont à exécuter sur une machine Linux et lors de ces séances de TP, les étudiants font se familiariser avec le système d'exploitation Linux et inspecter méthodiquement les divers outils nécessaires à la programmation.

---

## 1. Fiche TP N° 01 « Linux : Rappel sur les commandes de base et droits d'accès »

**Objectifs:**

L'intérêt de ce TP est de vous apprendre à être autonome. Il est assez normal que des erreurs surviennent quand vous essayez de répondre aux questions. Chercher à comprendre vous-même d'où vient l'erreur, tout en sachant décider quand il faut appeler à l'aide !

**Remarque:**

- `man` pour afficher le manuel de l'utilisation d'une commande.
- dans la fenêtre Terminal, la flèche vers le haut permet d'obtenir les commandes précédentes sans avoir à les retaper. Ctrl-A permet de déplacer le curseur au début de la ligne de commande, Ctrl-E à la fin de la commande.
- Shift+PgUp (PgDown) permet de remonter ou descendre dans le terminal (simulation de l'ascenseur) valable aussi en mode texte.
- Je vous conseille de consulter les sites web suivants qui sont très intéressants pour apprendre Linux, Java, C++ et d'autres:

https://www.developpez.com/

https://www.fun-mooc.fr/

https://openclassrooms.com/fr/dashboard

**Exercice 1:** Entrainement sur les commandes de base de Linux

1. Commencez par créer trois fichiers à l'aide de la commande `touch` dans votre `home directory`. Donnez–leur les noms `file1`, `file2` et `file3`.
2. Créez trois répertoires dans le répertoire `tp1` (répertoire à créer) et nommez-les respectivement `rep1`, `rep2` et `rep3` (utiliser la commande `mkdir`).
3. Copiez le fichier `file1` dans les répertoires `rep1` et `rep2` (par la commande `cp`). Copiez également `file2` dans les répertoires `rep2` et `rep3` mais renommez-le `file4`.
4. Déplacez (par `mv`) le fichier `file3` dans le répertoire `rep3`.

5. Vérifiez à l'aide de la commande `ls` que les répertoires contiennent bien ce qui suit :

   `rep1 : file1, rep2 : file1 et file4, rep3 : file3 et file4.`

6. Copiez `rep3` dans `rep2` (utiliser `cp -r` pour copier les répertoires) et déplacez `rep2` dans `rep1`.
7. Supprimez les deux fichiers restés à la base du répertoire `tp1` à l'aide de la commande `rm`. Supprimez le répertoire `rep3` (avec `rm -r`)
8. Vérifiez (`ls`) qu'il ne vous reste plus que `rep1` dans le répertoire `tp1`.

**Exercice 2:** Entrainement sur les droits d'accès de Linux

1. Déplacez-vous, si ce n'est pas déjà fait, à la base de votre `home directory` et faites un `ls -l`. vous pouvez constater que vous (`user`) possédez tous les droits sur `rep1 (rwx)`.
2. Enlevez-vous les droits de lecture sur rep1 en utilisant `chmod u-r rep1`.faites un listing (`ls -l`) pour vérifier que tout a bien fonctionné.
3. Déplacez-vous dans rep1. Remarquez bien que vous ne possédiez plus les droits de lecture sur ce répertoire, vous pouvez quand même vous y rendre.
4. Faites un listing du répertoire. Normalement vous devriez obtenir le message d'erreur (`ls : . Permission denied` ) signifiant que vous n'avez pas le droit de lire le répertoire courant(.).
5. Essayez de créer un fichier `file5` dans `rep1`. Ceci devrait fonctionner sans problème, car vous avez seulement enlevé les droits de lecture, mais pas ceux d'écriture.
6. Retournez à présent dans le répertoire parent (`cd ..`) et changez les droits de rep1 comme suit :

   `drw-r-xr-x`

7. Tentez de vous rendre dans rep1. Ça devrait se solder par un échec (`rep1 : Permission denied`), le droit d'exécution pour un répertoire correspond au droit de le traverser. Essayez de faire un listing de `rep1`. Que pouvez-vous en conclure ?
8. Reprenez tous les droits sur rep1 et déplacez-vous dans ce répertoire. A partir de là, tentez d'exécuter le fichier (tapez `./file1`). Qu'observez-vous ?
9. Ajoutez le droit d'exécution pour ce fichier et exécutez-le à nouveau. Cette fois, vous ne devriez pas obtenir de message d'erreur. Rien ne va se passer, évidemment, car le fichier est vide. Il arrive couramment qu'on ne parvienne pas à lancer un programme, parce que celui-ci ne possède pas les bons droits. Souvenez-vous en si ça vous arrive !

**Remarque** : pour certains caractères spéciaux, utiliser **Windows (démarrer)+R ensuite charmap**.

## 2. Fiche TP N° 02 « Programmation en langage C sous Linux »

**Remarques:**

- Les commandes Linux doivent être écrites en lettres minuscules
- **`exit`** pour quitter le shell actuel
- **`id`** pour afficher le numéro de l'utilisateur courant ainsi que les groupes auxquels ce dernier appartient.
- Pendant cette séance de TP, on va utiliser l'éditeur de texte **gedit** pour écrire nos programmes (vous pouvez utiliser d'autres éditeurs tels que : vi, emacs et nano,...). Pour nano les raccourcis les plus importants sont affichés en bas de sa fenêtre.
- Compiler le programme avec le compilateur **gcc** du projet GNU (pour les programmes C et C++) en tapant la commande : **`gcc -o nom_du_programme nom_du_programme.c`**
- Exécuter le programme par : **`./nom_du_programme`**

**Exercice 1:**

1) Compiler et exécuter le programme suivant:

```
# include <stdio.h>
main()
{
int NOMBRE, SOMME, COMPTEUR; SOMME=0;COMPTEUR=0;
While (COMPTEUR<4)
   {Printf ("Entrer un nombre entier:");
   Scanf (%i, &NOMBRE); SOMME+=NOMBRE;COMPTEUR++;}
   printf ("la somme est: %i \n", SOMME);
   Return 0;
}
```

2) Qu'est-ce qu'il fait ce programme:

**Exercice 2:**

1) Corriger le programme ci-dessous et faire son exécution.

```
# include <stdio.h>
int main(void)
}
int i, som, nbm
double moy
int t[20]
for (i=0; i<20, i++);
{
Printf ("donnez la note numéro %d:"; i+1)
Scan ("%d", &t[i]);
}
for(i=0, som=0; i<20; i++) som += t[i];
moy = som / 20
print ("\n\n moyenne de la classe : %f\n", moy);
for (i=0, nbm=0; i<20, i++ )
if (t[i] > moy ) nbm++
```

```
printf (%d élèves ont plus de cette moyenne, nbm);
return 0
{
```

2) Qu'elle la fonction principale de ce programme.

**Exercice 3:**

Soit les deux entiers a et b tel que a=-1972 et b=2021

1) Ecrire un programme en C qui permet de calculer et afficher : a-b, a+b, a*b, a/b et a%b en format décimal.
2) Que va-t-il se passer lors de l'exécution du programme précédent si a et b sont des variables réels (afficher le résultat).

**Exercice 4:**

1) Compiler et exécuter le programme suivant:

```
# include <stdio.h>
main()
{
int T[20];
int N,I;
long SOM;
printf ("donner la dimension du tableau T:");
do
scanf ("%d", &N);
while ((N<0) || (N>20));
for (I=0; I<N; I++)
{
printf ("Elément %d",I);
scanf ("%d", &T[I]);
}
printf ("Tableau donné:\n");
for (I=0;I<n;I++)
printf ("%d", &T[I]);
printf ("\n");
for (SOM=0;I=0;I<N;i++)
SOM+=T[I];
Printf ("la somme d'éléments est: %ld\n,SOM);
}
```

**Exercice 5:**

1) Compiler et exécuter le programme suivant:

```
# include <stdio.h>
main()
int A[10][20], B[20][30];
int N,M;/*dimensions des matrices*/
int I,J;/*indices courants*/
/*saisie des données*/
printf ("Nombre de lignes de A:");
scanf ("%d",&N);
printf ("Nombre de colonnes de A:");
scanf ("%d",&M);
```

```
for (I=0,i<N;I++)
for (J=0;J<M;J++)
{printf ("Element [%d][%d]:", I,J);
Scanf("%d",&A[i][j]);}
/*affichage des matrices*/
printf("Matrice donnée A:\n");
for (I=0,i<N;I++)
for (J=0;J<M;J++)
printf ("%7d", A[i][j]);
printf ("\n");
/*affectation de la matrice transposée à B*/
for (I=0,i<N;I++)
for (J=0;J<M;J++)
B[J][I]=A[I][J];
/*Edition des resultants*/
Printf("Matrice transposée de A:\n");
for (I=0,i<M;I++)
{for (J=0;J<N;J++)
printf ("%7d", B[I][J]);
printf("\n");}
return 0;
}
```

2) Exploiter les résultats de ce programme en modifiant les paramètres internes et déduire d'autres applications si possibles.

---

## 3. Fiche TP N° 03 « Manipulation des Processus et Threads en langage C »

**Remarques:**

- Les primitives de manipulation des processus sous Unix/Linux sont :

> **signal()** : gestion des signaux
> **fork()** : création d'un processus
> **pause()** : mise en sommeil sur l'arrivée d'un signal
> **wait()** : mise en sommeil sur la terminaison d'un fils
> **sleep()** : mise en sommeil sur une durée déterminée (argument)
> **kill()** : envoi de signal à un processus
> **exit()** : terminaison d'un processus
>
> **( pour plus d'informations, voir le man de chaque primitive)**

- L'appel `fork()` duplique un processus et le système crée alors une copie complète du processus, avec un PID différent. L'un des deux processus est fils de l'autre.
- `fork()` peut échouer par manque de mémoire ou si l'utilisateur a déjà créé trop de processus; dans ce cas, aucun fils n'est créé et `fork()` retourne -1
- Lors de la création d'un nouveau thread dans un processus, il obtient sa propre pile (et de ce fait ses propres variables locales) mais partage avec son créateur les variables globales.
- Pour compiler un programme en langage C, on utilise soit **gcc** (logiciel libre dans la cadre du projet GNU/GPL) ou bien **cc** (compilateur c/c++)

- **Remarque** : pour certains caractères spéciaux, utiliser **Windows(démarrer)+R ensuite charmap**.

**ps [-e][-l] :** Affiche la liste des processus

- L'option -e : permet d'afficher les processus de tous les utilisateurs.
- L'option -l : permet d'obtenir plus d'informations dont les plus importantes sont:

(**UID**: identité du propriétaire du processus; – **PID**: numéro du processus; – **PPID**: PID du père du processus; – **NI**: priorité (nice); – **S**: état du processus(**R** si actif, **S** si bloqué, **Z** si terminé).

1- En utilisant l'éditeur **gedit**, écrire les programmes suivants.

2- Compiler et exécuter ces programmes

3- Expliquer le déroulement de ces programmes

**Exercice 1** : gestion des processus par : `fork(), pid()` **`et`** `ppid()`

**cprocessus.c**

```
/* Exemple utilisation primitive fork() sous Linux */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main(void) {
    int pid; /* PID du processus fils */
    printf(("Bonjour … ") ;
    pid = fork();
    switch (pid) {
    case -1:
        printf("Erreur: echec du fork()\n");
        exit(1);
        break;
    case 0:
        /* PROCESSUS FILS */
        printf("je suis le processus fils : PID=%d , mon pere est :
        PPID=%d\n", getpid(), getppid());
        exit(0); /* fin du processus fils */
        break;
    default:
        /* PROCESSUS PERE */
        printf("Ici le pere: le fils a un pid=%d\n", pid );
        wait(0); /* attente de la fin du fils */
        printf("Fin du pere.\n");
    }

}
```

**Exercice 2** : gestion des threads à l'aide de l'API POSIX (**pthread.h**)
Ce programme crée un autre thread qui va montrer qu'il partage des variables avec le thread original, et permettre au "petit nouveau" de renvoyer un résultat à l'original.

**cthread.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <string.h>
void *fonction_de_thread(void *arg);
char message[] = "Hello World";
int main() {
int res;
pthread_t un_thread;
void *resultat_de_thread;
res=pthread_create(&un_thread, NULL, fonction_de_thread,
(void*)message);
if (res != 0)
{
perror("Echec de la creation du thread ");
exit(EXIT_FAILURE);
}
printf("En attente de terminaison du thread ...\n");
res=pthread_join (un_thread, &resultat_de_thread);
if (res != 0)
{
perror("Echec de l'ajout de thread ");
exit(EXIT_FAILURE);}
printf("Retour du thread, il a renvoye %s\n",(char
*)resultat_de_thread);
printf("Voici a present le message %s\n", message);
exit(EXIT_SUCCESS);
}
void *fonction_de_thread (void *arg)
{
printf("la fonction_de_thread est en cours d'execution. L'argument
était : %s\n", (char *) arg);
sleep (3);
strcpy(message, "Salut !");
pthread_exit("Merci pour le temps processeur");}
```

Fonction exécutée par le thread créé

Paramètre passé à la fonction

Identifiant du thread attendu

adresse de l'objet renvoyé au thread appelant

**N.B:**

- pour compiler ce programme il faut établir un lien avec la bibliothèque des threads :

```
gcc - o cthread cthread.c -lpthread
```

- **pthread_create:** crée un nouveau thread (comme le "fork" pour un processus)
- **pthread_join:** fait attendre la fin d'un thread (comme "wait" pour les processus)
- **pthread_exit :** termine un thread (comme "exit" pour les processus)

## 4. Fiche TP N° 04 « Threads et Sémaphores avec API POSIX »

**Remarques :**

**Commandes pour les processus (processus : programme en exécution)**

**ps [-e][-l] :** Affiche la liste des processus

- L'option -e : permet d'afficher les processus de tous les utilisateurs.
- L'option -l : permet d'obtenir plus d'informations dont les plus importantes sont:

(**UID**: identité du propriétaire du processus; – **PID**: numéro du processus; – **PPID**: PID du père du processus;
– **NI**: priorité (nice); – **S**: état du processus(**R** si actif, **S** si bloqué, **Z** si terminé).

1) Pour compiler et effectuer l'édition de liens vous devez utiliser la ligne suivante :

```
gcc -o votre_programme votre_programme.c –lpthread
```
ou bien
```
gcc votre_programme.c -lpthread -o votre_programme
```

2) Utilisation de man pour toute aide sur l'utilisation des fonctions ci-après.

### Partie 1: Les threads LINUX (processus de poids légers)

Les threads de LINUX sont gérés à la fois par le système et par une librairie au niveau utilisateur. Voici quelques fonctions standards de la l'API POSIX (API : **A**pplication **P**rogramming **I**nteface et POSIX : **P**ortable **O**perating **S**ystem **I**nterface, dont le **X** exprime l'héritage UNIX de l'API)

```
pthread_create( thread, attribut, routine, argument )
```

Création d'un thread. Le nouveau flot d'exécution démarre en se branchant à la routine spécifiée. Cette routine reçoit l'argument prévu.

```
pthread_exit( résultat )
```

Suicide d'un thread.

```
pthread_join( thread, résultat )
```

Attendre la terminaison d'un autre thread.

```
pthread_kill( thread, nu_du_signal )
```

Envoyer un signal (UNIX) à un thread. C'est un moyen dur pour tuer un thread.

```
sched_yield()
```

Abandonner la CPU pour la donner à un autre thread (ou un autre processus). Attention : il n'existe pas de préemption de la CPU à l'intérieur des threads d'un même processus. En clair, si un thread garde la CPU, les autres threads ne vont pas s'exécuter. Cette routine permet de programmer un partage équitable de la CPU entre threads coopératifs.

**Exercice 1** : gestion des threads à l'aide de l'API POSIX (**pthread.h**)

Un thread lit des caractères au clavier et les passe à un autre thread qui se charge de les afficher. Il faut noter que le thread principal (le père) se charge de la création de ses fils et de l'attente de leur mort. Cette disparition est programmée à l'arrivée du caractère "F".

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

volatile char theChar = '\0';
volatile char afficher = 0;

void* lire (void* name) {
    do {
        while (afficher == 1) ; /* attendre mon tour */
        theChar = getchar();
        afficher = 1; /* donner le tour */
        }
    while (theChar != 'F');
    return NULL;
    }

void* affichage (void* name) {
    int cpt = 0;
    do {
        while (afficher == 0) cpt ++; /* attendre */
        printf("cpt = %d, car = %c\n", cpt, theChar);
        afficher = 0; /* donner le tour */
        }
    while (theChar != 'F');
    return NULL;
    }

int main (void) {
    pthread_t filsA, filsB;

    if (pthread_create(&filsA, NULL, affichage, "AA")) {
        perror("pthread_create");
        exit(EXIT_FAILURE);
        }
    if (pthread_create(&filsB, NULL, lire, "BB")) {
        perror("pthread_create");
        exit(EXIT_FAILURE);
        }

    if (pthread_join(filsA, NULL))
        perror("pthread_join");

    if (pthread_join(filsB, NULL))
        perror("pthread_join");

    printf("Fin du pere\n") ;
    return (EXIT_SUCCESS);
    }
```

## Partie 2: Utilisation des sémaphores pour synchroniser les threads

La librairie de gestion des *threads* offre les fonctions ci-dessous pour créer et utiliser des sémaphores. **Attention** : ces sémaphores sont propres à un processus. Ils permettent de synchroniser plusieurs threads entre eux, mais ils ne peuvent synchroniser plusieurs processus. Pour réaliser cette synchronisation il faut se tourner vers les sémaphores système V basés sur les IPC (*Inter Processus Communication*).

**`int sem_init(sem_t *semaphore, int pshared, unsigned int valeur)`**
Création d'un sémaphore et préparation d'une valeur initiale.

**`int sem_wait(sem_t * semaphore);`**

Opération P sur un sémaphore.

**`int sem_trywait(sem_t * semaphore);`**

Version non bloquante de l'opération P sur un sémaphore.

**`int sem_post(sem_t * semaphore);`**

Opération V sur un sémaphore.

**`int sem_getvalue(sem_t * semaphore, int * sval);`**

Récupérer le compteur d'un sémaphore.

**`int sem_destroy(sem_t * semaphore);`**

Destruction d'un sémaphore.

**Exercice 2** : utilisation des sémaphores à l'aide de l'API POSIX (**pthread.h**)

Cet exercice illustre la mise en œuvre d'une section critique (mutuelle exclusion) permettant d'éviter un mélange des affichages réalisés par les deux threads.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t mutex;

void* affichage (void* name) {
    int i, j;
    for(i = 0; i < 20; i++) {
        sem_wait(&mutex); /* prologue */
        for(j=0; j<5; j++) printf("%s ",(char*)name);
        sched_yield(); /* pour etre sur d'avoir des problemes */
        for(j=0; j<5; j++) printf("%s ",(char*)name);
        printf("\n ");
        sem_post(&mutex); /* epilogue */
    }
    return NULL;
    }
int main (void) {
    pthread_t filsA, filsB;
    sem_init(&mutex, 0, 1);
    if (pthread_create(&filsA, NULL, affichage, "AA")) {
        perror("pthread_create");
        exit(EXIT_FAILURE);
        }
    if (pthread_create(&filsB, NULL, affichage, "BB")) {
        perror("pthread_create");
        exit(EXIT_FAILURE);
        }
    if (pthread_join(filsA, NULL))
        perror("pthread_join");
    if (pthread_join(filsB, NULL))
        perror("pthread_join");
    printf("Fin du pere\n") ;
    return (EXIT_SUCCESS);
    }
```

## 5. Fiche TP N° 05 « Gestion des Threads en langage Java »

**Remarques:**

- Gestion lourde des threads avec C (API POSIX) et simplifiée avec Java.
- En java, les threads sont des instances des classes dérivées (héritées) de la **classe *Thread***
  - La classe Thread crée des threads généraux, sa méthode **run** ne fait rien.
  - La méthode **run** indique à un thread les instructions à exécuter
  - La méthode **run** doit être publique, ne prendre aucun argument, ne renvoyer aucune valeur et ne lever aucune exception.
- Deux techniques pour fournir une méthode run à un thread
  3) hériter la classe Thread (java.lang.Thread) et redéfinir la méthode run.
  4) Implémenter l'interface Runnable (java.lang.Runnable) et définir la méthode run de cette classe
- Un thread commence par exécuter la méthode run de l'objet "cible" qui a été passé au thread.

**Exercice 1** : Technique n°1 : **Hériter la classe Thread**

**Thread1.java**

```
class TPrint extends Thread
{
String txt;
int attente;
public TPrint(String t, int p)
{ txt= t; attente=p;}
public void run (){ for (int  i=0 ; i<8;i++) {
System.out.print (txt+i+" ");
try {
    sleep(attente);
    }
catch (InterruptedException e) {};
}
}
}
public class Thread1 {
static public void main(String args[]){
TPrint a= new TPrint("A", 100); // créer un thread
TPrint b= new TPrint("B", 200); // créer un autre thread
a.start();
b.start();
}
}
```

**N.B:**

Le Résultat des deux exercices devrait être : A0 B0 A1 A2 B1 A3 A4 B2 A5 A6 B3 A7 B4 B5 B6 B7

**Question**: Donner une explication de ce résultat ?

**Exercice 2** : Technique n°2 : **Implémenter l'interface Runnable**

**Thread2.java**

```
import java.io.*;
import java.lang.*;
class TPrint implements Runnable {
String txt;
int attente;
public TPrint (String t, int p)
{ txt= t; attente=p;}
public void run (){ for (int  i=0 ; i<8;i++) {
System.out.print (txt+i+" ");
try {
    Thread.currentThread().sleep(attente);
    }
catch (InterruptedException e) {};
}
}
}
public class Thread2 {
static public void main(String args[]){
TPrint a= new TPrint("A", 100);
TPrint b= new TPrint("B", 200);
new Thread(a).start(); // Créer et lancer un thread
```

**Exercice 3** : Soit 2 threads qui comptent de 1 à 10 et à 15. Commençons donc par créer une sous-classe de la classe Thread, puis créons une classe permettant de lancer les deux threads via la méthode main() :

**LanceCompteurs.java**

```
import java.io.*;
import java.lang.*;

class ThreadCompteur extends Thread {
int no_fin;
// Constructeur
ThreadCompteur(int fin) {
no_fin = fin;
}
// On redéfinit la méthode run()
public void run() {
for (int i=1; i<=no_fin ; i++) {
System.out.println(this.getName()+"==>"+i);
}
}
}
// Classe lançant les threads
class LanceCompteurs {
public static void main (String args[]) {
// On instancie les threads
ThreadCompteur cp1 = new ThreadCompteur(10);
ThreadCompteur cp2 = new ThreadCompteur(15);
```

```
// On démarre les deux threads
cp1.start();
cp2.start();
// On attend qu'ils aient fini de compter
while (cp1.isAlive() || cp2.isAlive()) {
// On bloque le thread 100 ms
try {
Thread.sleep(100);
} catch (InterruptedException e) { return; }
}
}
}
```

**Le résultat devrait être similaire à ce qui suit :**

Thread-0==>1
Thread-1==>1
Thread-0==>2
Thread-1==>2
Thread-0==>3
Thread-1==>3
….
Thread-1==>13
Thread-1==>14
Thread-1==>15

**Question** : Donner une explication de ce résultat ?

**Rappel :**

1) Pour compiler un programme java (par exemple, exemple.java) :

**javac exemple.java**

qui va générer exemple.class (le bytecode de ce programme)

2) Pour l'exécuter :

**java exemple**

## 6. Fiche TP N° 06 « Synchronisation des Threads avec les Moniteurs en Java »

Compiler et exécuter les programmes suivants:

**Exercice 1:** TestThread1.java (Attention: Java est sensible à la casse)

```
import java.io.* ;
import java.lang.* ;

class TPrint extends Thread {
         String txt;
         public TPrint(String t) {
                txt = t;
         }
        public void run() {
               for (int j=0; j<3; j++) {
                    for (int i=0;
i<txt.length();i++) {

System.out.print(txt.charAt(i));
                        try { sleep(100);}
                        catch
(InterruptedException e) {};
                        }  } }  }
```

```
public class TestThread1
{
    static public void main(String
args[])
   {
     TPrint a = new TPrint("bonjour
");
     TPrint b = new TPrint("au revoir
");
     a.start(); b.start(); }   }
```

**// résultat de l'exécution :**
// baoun jroeuvro ibro najuo urre vbooinrj oauur revoir

**Qu'observez-vous?**

**Exercice 2:** TestThread2.java

```
import java.io.* ;
import java.lang.* ;
class MoniteurImpression {
     synchronized public void imprime (String
t) {
          for (int i=0; i<t.length(); i++) {
             System.out.print(t.charAt(i));
             try {
Thread.currentThread().sleep(100);}
             catch (InterruptedException e)
{};
          }
    }
}
class TPrint extends Thread {
String txt;
     static MoniteurImpression mImp =
                  new MoniteurImpression();
     public TPrint(String t) {txt = t;}
     public void run(){for (int j=0; j<3; j++)
                  mImp.imprime(txt);     }}
```

```
public class TestThread2{
 static public void main(String
args[]) {
    TPrint a = new TPrint("bonjour
");
    TPrint b = new TPrint("au revoir
");
    a.start();
    b.start(); } }
```

**// resultat de l'execution :**
// bonjour au revoir bonjour au revoir bonjour au revoir

**Qu'observez-vous?**

## 7. Fiche TP N° 07 « Etude de cas : Les Threads pour Gérer un Restaurant »

Ce TP présente une étude de cas réelle sur la restauration des clients et comment les sémaphores peuvent organiser le travail.

Compiler et exécuter les programmes suivants pour voir les différents résultats attendus de cette synchronisation des threads.

**Programme Client.java :**

```
package PROJECT_SE;


import java.util.Random;
import java.util.concurrent.Semaphore;

public class Client extends Thread {

          String name;
          Semaphore sem;

          public Client(String pName, Semaphore pSem){
             name = pName;
             sem = pSem;
          }

          public void run() {
                try {
                //Nous prenons une réservation de ressource
                sem.acquire();

                Random rand = new Random();

                //Pour avoir une pause conséquente et bien
                //et bien voir le mécanisme du sémaphore.
                long pause = 0;
                while(pause < 9000)
                    pause = rand.nextInt(15000);

                System.out.println(name + " : Je mange au restaurant pendant " +
pause/1000 + " minutes");

                Thread.sleep(pause);

                System.out.println(name + " : Au revoir. Je quitte le restaurant. ");

                //Pour libérer la ressource
                sem.release();
             } catch (InterruptedException e) {
                e.printStackTrace();
             }
           }


}
```

**Programme Restaurant.java :**

```
package PROJECT_SE;


import java.util.Scanner;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

public class Restaurant {
   public static void main(String[] args) {
      //Bon, c'est un restaurant à 5 places...
      //C'est petit, mais en Bretagne, il y en a. ;)
      Semaphore sem = new Semaphore(8);

      ExecutorService execute = Executors.newCachedThreadPool();
      System.out.print("Combient de client arrive au restaurant: ");
      Scanner clavier = new Scanner(System.in);
      int cl = clavier.nextInt();
      int i = 0;
      while(i<cl){
         Client cli = new Client("Client N°" + (i++), sem);
         execute.execute(cli);

         try {
            Thread.sleep(1000);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
      clavier.close();
   }
}
```

Chapitre VIII

# Exercices Pratiques Corrigés

## Objectifs

- Pratiquer la synchronisation des processus et threads avec les Pipes Linux.

## Themes couverts

Enoncés et corrigés de:

- Exercice 1 : Problème de synchronisation
- ***Exercice 2 : Verrou simple par fichier***
- ***Exercice 3 : Synchronisation simple par pipe***
- ***Exercice 4 : Résolution du dysfonctionnement***
- ***Exercice 5 : Synchronisation de threads***

# Chapitre 8: Exercices pratiques corrigés

Ce chapitre propose un supplément d'exercices pratiques sur la synchronisation des processus en examinons d'autres techniques tel que l'utilisation des Pipes.

En général, Le Pipe est une commande sous Linux qui vous permet d'utiliser deux ou plusieurs commandes de sorte que la sortie d'une commande serve d'entrée à la suivante. En bref, la sortie de chaque processus directement en entrée du suivant comme un pipeline. Le symbole « | » désigne un tuyau. Les tuyaux vous aident à mélanger deux ou plusieurs commandes en même temps et à les exécuter consécutivement. Vous pouvez utiliser des commandes puissantes qui peuvent effectuer des tâches complexes en un tournemain.

Comprenons cela avec un exemple. Lorsque vous utilisez la commande 'cat' pour afficher un fichier qui s'étend sur plusieurs pages, l'invite passe rapidement à la dernière page du fichier et vous ne voyez pas le contenu au milieu. Pour éviter cela, vous pouvez diriger la sortie de la commande « cat » vers « less » qui vous montrera une seule longueur de défilement de contenu à la fois.

```
cat filename | less
```

Les étudiants font se familiariser avec l'utilisation des pipes pour la communication et synchronisation des processus/threads. Les corrigés de ces exercices sont à la fin de ce chapitre.

## 1. Enoncés des exercices

***Exercice 1 : Problème de synchronisation***

Ecrire un programme qui affiche à l'écran et dans un fichier, caractère par caractère, la chaîne de caractères passée en 1er argument de la ligne de commande. Insérer l'appel de la fonction usleep entre chaque écriture.

Utiliser :

- la fonction usleep pour introduire une attente
- la fonction write pour écrire dans le fichier.
- la fonction fsync pour vider le buffer après chaque caractère.

Pour obtenir l'aide à propos de cette fonction, taper man nom_fonction.

- Compiler le programme :
  gcc -o exo1.x exo1.c
- Exécuter plusieurs instances du programme en parallèle comme suit :
  ./exo1 coucou & ./exo1 la & ./exo1 voilou &

Observer les résultats. Exécuter de nouveau puis observer.

## ***Exercice 2 : Verrou simple par fichier***

Ecrire un programme qui crée un fichier verrou, attend 5 secondes, affiche un message, ferme et supprime le fichier verrou. Si le verrou est déjà pris, afficher simplement un message d'erreur.

Visualiser l'aide à propos des fonctions open, unlink :

- man 2 open
- man 2 unlink

Exécuter plusieurs instances de ce programme en parallèle pour vérifier son fonctionnement.

## ***Exercice 3 : Synchronisation simple par pipe***

Ecrire un programme qui lance l'exécution de deux processus communiquant par un pipe : le processus A écrit périodiquement un octet quelconque dans le pipe ; le processus B affiche à l'écran un nombre au hasard à chaque fois qu'un octet est lu depuis le pipe. Les deux processus sont ainsi synchronisés via le pipe.

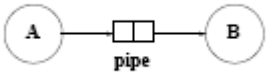



Fig. 1 – A et B communiquent par un pipe

Consulter les aides suivantes si nécessaire :

- man pipe
- man fork
- man perror

## ***Exercice 4 : Résolution du dysfonctionnement***

Résoudre le dysfonctionnement de la synchronisation de l'exercice 1. Utiliser la technique de fichier verrou. Pourquoi la solution de synchronisation par pipe est-elle difficilement envisageable ?

## ***Exercice 5 : Synchronisation de threads***

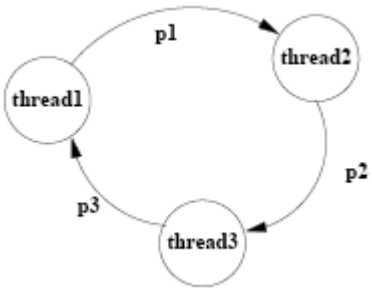



Fig. 2 – Graphe de tâches

Créer un programme qui effectue les opérations suivantes :

- Créer 3 pipes p1, p2, p3
- Créer 3 processus thread1, thread2, thread3 respectant le graphe 2

Implémenter un système de synchronisation par messages afin d'exécuter les tâches dans l'ordre suivant : thread1, thread2, thread3, thread1.

## 2. Corrigés des exercices

***Solution de l'exercice 1***

```
  #include <unistd.h>
  #include <stdio.h>
  int main (int argc, char **argv){
  int i;
  if (argc < 2){
     fprintf(stderr,"Il faut un argument\n");
     return -1;
  }
  int fd = open ("bob.txt", O_RDWR | O_CREAT | O_APPEND, S_IRWXU);
  for (i = 0; i < strlen(argv[1]); ++i)
  {
    write (fd, argv[1] + i, 1);
    fprintf(stdout, "%c ",*(argv[1]+i));
    usleep (1);
    fsync (fd);
  }
  close (fd);
  return 0;
  }
```

***Solution de l'exercice 2***

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>

int main(int argc, char **argv){
  int fd;
  printf("Starting %s with PID %d\n", argv[0], getpid());
  printf("Trying to open the lockfile first...\n");
  fd = open("lockfile", O_CREAT | O_EXCL, 0660);
  if (fd < 0) { /* if open returns a negative number, there was an error */
    if (errno == EEXIST) {
      printf("file is locked, can't do anything\n");
      return -1;
    } else {
      perror("unexpected error");
      return -1;
    }
  }
  printf("Lockfile opened successfully, doing stuff...\n");
 /* insert code here that puts data in another file... */
  sleep(10);
 /* close / remove the lockfile, because now we are done */
  close(fd);
  unlink("lockfile");
 return 0;
}
```

***Solution de l'exercice 3***

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>

void thread1(void* arg) {
  int randnum;
  int* toto = (int*)arg;
  ssize_t ret_write;
  while(1) {
    randnum=rand();
    do {
     ret_write = write(toto[1],&(randnum),sizeof(int));
    }
        while (ret_write == -1 || ret_write == 0);
      sleep(1);

   }
   close(toto[1]);

   }
void thread2(void* arg) {
  int* toto = (int*)arg;
  ssize_t ret_read;
  int read_int;
  while (1) {
  ret_read = read(toto[0], &read_int, sizeof(int));

  if (ret_read == -1) {
     perror("thread2: return value from pipe");
  continue;
  }
  if (ret_read == 0) {
     perror("thread2: return value from pipe");
     continue;
  }
  printf("child 2 : caractere recu: %d\n",read_int);
  if (read_int == '\0')

     break;
  }
  close(toto[0]);
  }
int main(int argc, char * argv[]) {
  int com[2], pid;
//  pthread_t tidThread1, tidThread2;
  if (pipe(com) != 0) {
     perror("Error when creating pipe\n");
     exit(1);
  }
  pid=fork();
  if (pid == 0)
  {
//  child 1
    fprintf(stdout,"child 1 started\n");
    thread1(com);
    exit(0);

  }
```

```
  if (pid <0)
  {
  fprintf(stderr,"Error creating child 1\n");
  exit(1);
  }
  pid=fork();
  if (pid == 0)
  {
//  child 2
    fprintf(stdout,"child 2 started\n");
    thread2(com);
    exit(0);
   }
   if (pid <0)
   {
   fprintf(stderr,"Error creating child 2\n");
   exit(1);
   }
  exit(0);
 }
```

***Solution de l'exercice 4***

```
  #include <stdio.h>
  #include <unistd.h>
  #include <sys/types.h>
  #include <sys/stat.h>
  #include <fcntl.h>
  #include <errno.h>

int main(int argc, char **argv) {
  int fd,fd2;
  int i;
  if (argc < 2)return -1;
  printf("Trying to open the lockfile first...\n");
  do {
      fd = open("lockfile", O_CREAT | O_EXCL, 0660);
      sleep(1);
     } while ((fd < 0) && (errno == EEXIST));
   printf("Lockfile opened successfully, doing stuff...\n");
  fd2 = open ("bob.txt", O_RDWR | O_CREAT | O_APPEND, S_IRWXU);
  for (i = 0; i < strlen(argv[1]); ++i)
  {
    write (fd2, argv[1] + i, 1);
    fsync(fd2);
  }
  close (fd2);
 /* close / remove the lockfile, because now we are done */
  printf("removing the lockfile...\n");
  close(fd);
  unlink("lockfile");
  return 0;
 }
```

***Solution de l'exercice 5***

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

```
  #include <sys/stat.h>
  #include <fcntl.h>
  #include <stdio.h>

void* thread1 (void *arg)
  {
  int *filedes = (int*)arg;
  int  i = 0;
  char buf[16];
  printf ("thread1 running\n");
  if (write (filedes[1], "Coucou", 7) == -1)
  {
    perror ("Error while writing");
    return NULL;
  }
  while (1) {
    ssize_t s = read  (filedes[0], buf + i, 16);
    i += s;
    if (s == 0)
    continue;
    if (s == -1)
    {
       perror ("Error while reading");
       return NULL;
    }
  if (buf[i-1] == '\0')
    break;
  }
  printf ("result : %s\n",buf);
  printf ("thread1 done.\n");
  return NULL;
 }
void*
  thread2 (void *arg)
  {
  int *filedes = (int*) arg;
  char buf[16];
  int  i;
  printf ("thread2 running\n");
  while (1) {
  int j;
  ssize_t s = read  (filedes[0], buf, 16);
  if (s == 0)
  continue;
  if (s == -1)
  {
  perror ("Error while reading");
  return NULL;
  }
  printf ("2 : %d\n", s);
  for (j = 0; j < s; j++)
  if (buf[j] != '\0') buf[j]++;
  if (write (filedes[1], buf, s) == -1)
  {
  perror ("Error while writing");
  return NULL;
  }
  if (buf[s-1] == '\0')
  break;
  printf ("thread2 done.\n");
  }
  return NULL;
  }
void* thread3 (void *arg)
```

```
  {
  int *filedes = arg;
  char buf[16];
  printf ("thread3 running\n");
  while (1) {
    ssize_t s = read  (filedes[0], buf, 16);
    int j;
    if (s == 0)
       continue;
    if (s == -1)
    {
      perror ("Error while reading");
      return NULL;
    }
    printf ("3 : %d\n", s);
    for (j = 0; j < s; j++)
       switch (buf[j]) {
       case 'd' : buf[j] = 'a'; break;
       case 'p' : buf[j] = 'b'; break;
       case 'v' : buf[j] = 'c'; break;
       case 'D' : buf[j] = 'a'; break;
       case '\0': case '\n' : break;
       default :
          buf[j] = '.';
    }
  printf ("bob\n");
  if (write (filedes[1], buf, s) == -1)
  {
     perror ("Error while writing");
    return NULL;
  }
  printf ("bob\n");
  if (buf[s-1] == '\0')
    break;
    printf ("bob\n");
  }
  printf ("thread3 done.\n");
  return NULL;
}

  main()
  {
  int p1[2];
  int p2[2];
  int p3[2];
  int pid;
  int pipe1[2];
  int pipe2[2];
  int pipe3[2];
// creation pipes
  if (pipe (pipe1) == -1)
     perror ("pipe");
  printf ("%d %d\n",pipe1[0], pipe1[1]);
  if (pipe (pipe2) == -1)
     perror ("pipe");
  printf ("%d %d\n",pipe2[0], pipe2[1]);
  if (pipe (pipe3) == -1)
     perror ("pipe");
  printf ("%d %d\n",pipe3[0], pipe3[1]);
  p1[0] = pipe3[0]; p1[1] = pipe1[1];
  p2[0] = pipe1[0]; p2[1] = pipe2[1];
  p3[0] = pipe2[0]; p3[1] = pipe3[1];
// start the threads
    pid=fork();
```

```
    if (pid == 0)
    {
    //  child 1
    fprintf(stdout,"child 1 started\n");
    thread1(p1);
    exit(0);
   }
    if (pid <0)
    {
    fprintf(stderr,"Error creating child 1\n");
    exit(1);
    }
   pid=fork();
   if (pid == 0)
   {
//  child 2
    fprintf(stdout,"child 2 started\n");
    thread2(p2);
    exit(0);
   }
   if (pid <0)
   {
   fprintf(stderr,"Error creating child 2\n");
   exit(1);
   }
   pid=fork();
   if (pid == 0)
   {
//  child 3
    fprintf(stdout,"child 3 started\n");
    thread3(p3);
    exit(0);
   }
   if (pid <0)
   {
   fprintf(stderr,"Error creating child 2\n");
   exit(1);
   }
  printf ("All threads done.\n");
  return 0;

  }
```