الجمهورية الجزائرية الديمقراطية الشعبية

République Algérienne Démocratique et Populaire

Ministère de l'Enseignement supérieur et de la recherche scientifique

Université de 8 Mai 1945 – Guelma

Faculté des Mathématiques, d'Informatique et des Sciences de la matière

-Département d'Informatique-

Cours système d'exploitation 2

Troisième année licence " ISIL"

-Cours et exercices corrigés-

Support de cours réalisé par : Dr. KOUAHLA Zineddine

Année 2022

2022

Cours système d'exploitation 2

KOUAHLA Zineddine

Université 8 Mai 1945 Guelma

12/02/2022

Table des Matières

Chapitre I : Rappels sur la notion de SE

Chapitre II : Synchronisation de processus

Table des Matières

Chapitre III : La communication interprocessus

Liste des Figures

Table des Matières

1. SYLLABUS

COURS SYSTÈME D'EXPLOITATION II

Semestre : 6 Parcours SI

Unité d'enseignement fondamentale : UEF1

Matière : Système d'exploitation 2

Crédits : 5 Coefficient : 3

Objectifs de l'enseignement :

Une étude approfondie du système Unix est recommandée pendant les séances de TD et de TP. La programmation des threads et des mécanismes de l'exclusion mutuelle se fera en C sous Unix.

Les modèles producteurs/consommateur, lecteur/rédacteurs et des philosophes avec plusieurs variantes seront étudiés de façon théorique (développement d'algorithmes en pseudo-langage) en TD puis implémentés en C sous Unix durant les séances de TP.

Connaissances préalables recommandées : système d'exploitation 1.

Contenu de la matière :

Chapitre 1 :

- Rappels sur la notion de SE.

- Notions de programme, processus, thread et ressource partagée.

Chapitre 2 : Synchronisation de processus.

- Problème de l'accès concurrent à des ressources et sections critiques (Problème de l'exclusion mutuelle)

- Outils de synchronisation :

- Événements, Verrous
- Sémaphores
- Moniteurs
- Régions critiques.

Chapitre 3 : La communication interprocessus

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

Chapitre 4 : L'inter blocage

- Modèles
- Prévention
- Evitement
- Détection/ Guérison

Mode d'évaluation : Examen (60%), contrôle continu (40%)

1. Introduction générale

Ce document se présente sous la forme d'un polycopié qui reprend le contenu de la matière tel que spécifié dans le programme et le syllabus du cours. Il est destiné principalement aux étudiants du troisième année licence informatique souhaitant approfondir ses connaissances dans le domaine des systèmes d'exploitation.

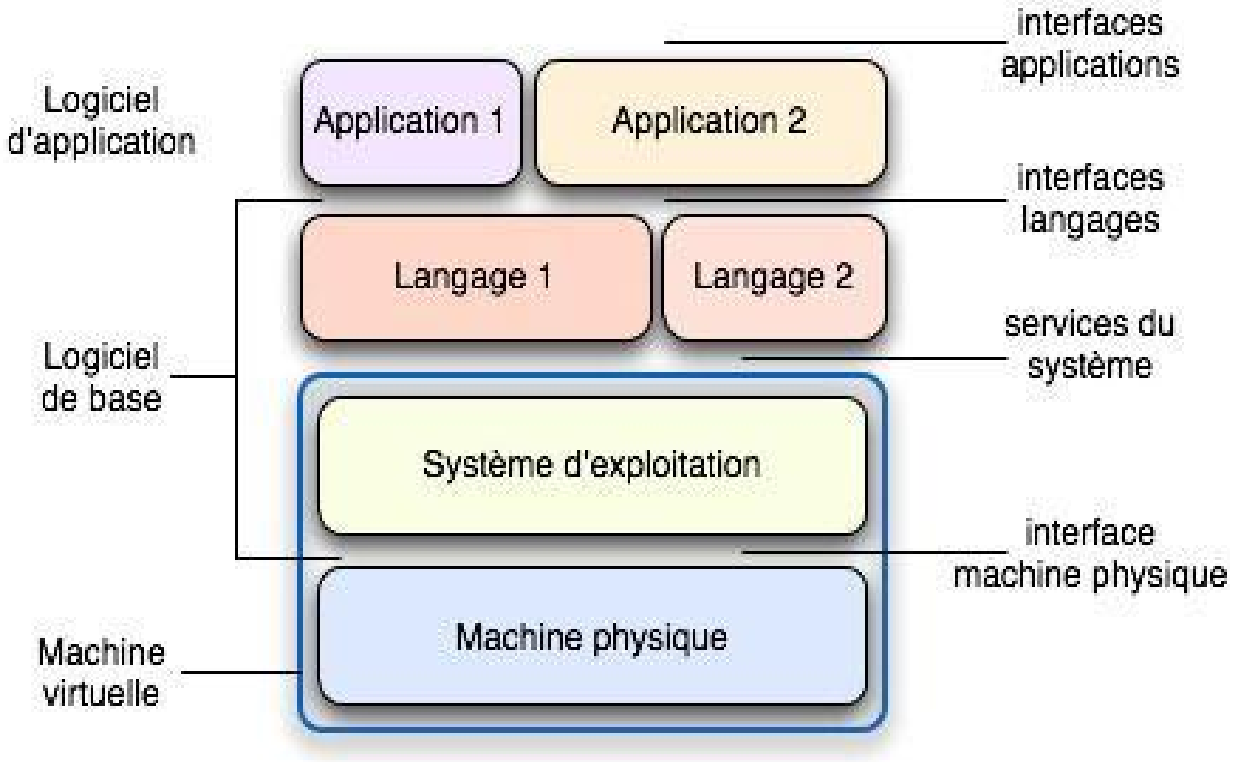
L'objectif principal de ce tutoriel est de comprendre le fonctionnement système, matériel et logiciel d'un ordinateur dans son contexte de travail. Et aussi de fournir aux étudiants des techniques avancées pour gérer les ressources et éviter les problèmes qui peuvent survenir (section critique, exclusion mutuelle, synchronisation, communication inter-processus et inter-blocage).

Ce document est structuré en 4 chapitres. Le chapitre 1 rappelle les notions de base des systèmes d'exploitation. Le chapitre 2 aborde les techniques de gestion des accès concurrents aux ressources critiques et présente quelques algorithmes de gestion des sections critiques. Le chapitre 3 se concentre sur les mécanismes de communication et de coopération interprocessus. Enfin, le chapitre 4 est consacré aux problèmes d'interblocage.

Chapitre 1 : Rappels sur les notions des Systèmes d'Exploitation

2. Qu'est-ce qu'un système d'exploitation ?

Un système d'exploitation est un programme qui contrôle l'exécution de tous les autres programmes (applications). Un système d'exploitation joue le rôle d'intermédiaire entre l'utilisateur (ou les utilisateurs) et l'ordinateur. Les principaux



objectifs de ce programme sont :

Figure 1 Qu'est-ce qu'un système d'exploitation

- ✔ **Commodité**, Le système doit répondre de façon prévisible à des conditions d'erreurs, même celles causées par une panne matérielle.
- ✔ **Efficacité**, Le système doit se protéger et protéger ses utilisateurs contre des attaques délibérées ou accidentelles menées par des programmes d'utilisateurs.

- ✔ **Extensibilité,** il existe différents niveaux de compatibilité : compatibilité source : l'application doit être recompilée. Compatibilité binaire : exécution directe (très difficile entre des processeurs différents).

3. Historique

Les systèmes d'exploitation, comme le matériel informatique, ont subi une série de changements révolutionnaires appelés générations. Dans le cas du matériel informatique, les générations ont été marquées par des avancées majeures dans les composants, des tubes à vide (première génération) aux transistors (deuxième génération), en passant par les circuits intégrés (troisième génération) et les circuits intégrés à grande et très grande échelle (quatrième génération). Les générations successives de matériel informatique se sont toutes accompagnées d'une réduction spectaculaire des coûts, de la taille, des émissions de chaleur et de la consommation d'énergie, ainsi que d'une augmentation spectaculaire de la vitesse et de la capacité de stockage.

1) Les premiers ordinateurs étaient conçus pour exécuter une série de tâches simples, comme une calculatrice. Les caractéristiques de base des systèmes d'exploitation ont été développées dans les années 1950, comme les fonctions de moniteur résident qui pouvaient exécuter automatiquement différents programmes les uns après les autres pour accélérer le traitement.
2) Dans les années 40, les premiers systèmes électroniques numériques n'avaient pas de système d'exploitation. Les systèmes électroniques de cette époque

étaient programmés sur des rangées de commutateurs mécaniques ou par des fils de liaison sur des cartes de connexion.

Il s'agissait de systèmes à usage spécifique qui, par exemple, généraient des tableaux balistiques pour l'armée ou contrôlaient l'impression des chèques de paie à partir de données figurant sur des cartes perforées.

Après l'invention des ordinateurs programmables à usage général, des langages machine (consistant en des chaînes de chiffres binaires 0 et 1 sur des bandes de papier perforé) ont été introduits pour accélérer le processus de programmation.

3) SE/360 a été utilisé sur la plupart des ordinateurs centraux IBM à partir de 1966, y compris les ordinateurs utilisés par le programme Apollo.
4) Au début des années 1950, un ordinateur ne pouvait exécuter qu'un seul programme à la fois. Chaque utilisateur avait l'usage exclusif de l'ordinateur pour une période limitée et arrivait à une heure précise avec son programme et ses données sur des cartes de papier perforé ou des bandes perforées.

   Le programme était chargé dans la machine, et la machine était mise au travail jusqu'à ce que le programme soit terminé ou qu'il tombe en panne. Les programmes pouvaient généralement être débogués via un panneau frontal à l'aide d'interrupteurs à bascule et de voyants. On dit qu'Alan Turing était passé

maître dans ce domaine sur la première machine Manchester Mark 1, et qu'il déduisait déjà la conception primitive d'un système d'exploitation des principes de la machine universelle de Turing.

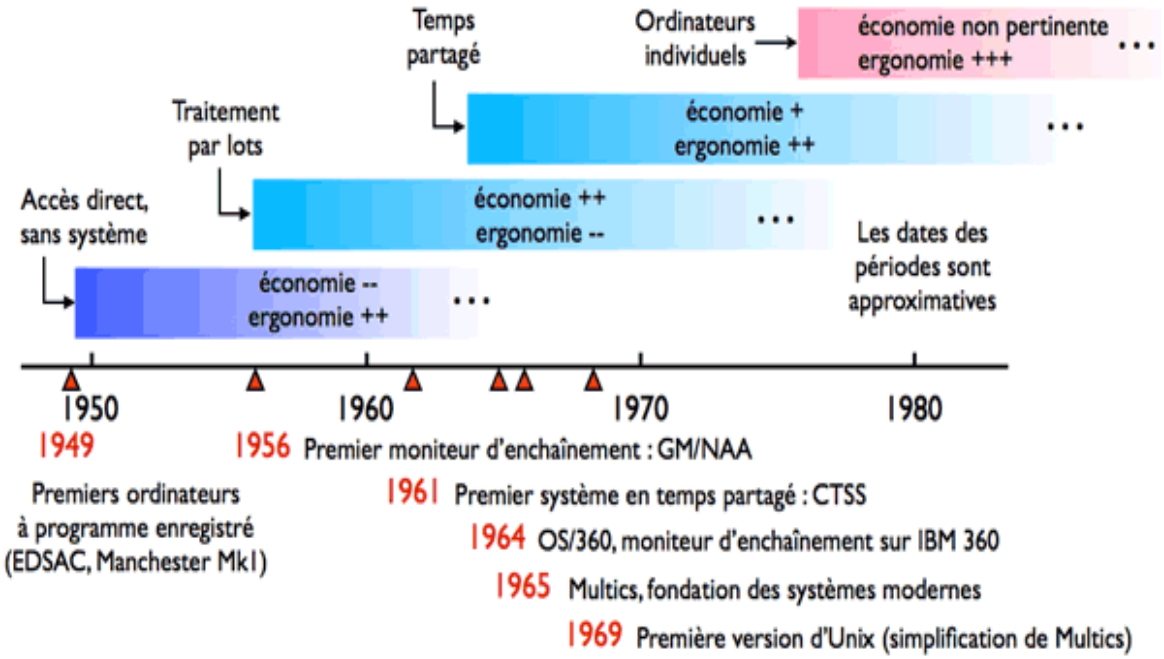


Figure 2 Chronologie simplifiée des premiers systèmes

5) Plus tard, les machines ont été livrées avec des bibliothèques de programmes, qui étaient liées au programme de l'utilisateur pour l'aider dans des opérations telles que l'entrée et la sortie et la compilation (génération du code machine à partir du code symbolique lisible par l'homme).
6) À l'université de Cambridge, en Angleterre, la file d'attente des tâches était à un moment donné une corde à linge à laquelle étaient suspendues des bandes avec des pinces à linge de différentes couleurs pour indiquer la priorité des tâches.

7) Une amélioration a été apportée par le superviseur de l'Atlas. Introduit avec le Manchester Atlas en 1962, il est considéré par beaucoup comme le premier système d'exploitation moderne reconnaissable.

4. Définition d'un système d'exploitation

Un système d'exploitation est un programme (logiciel) qui joue un rôle d'interface entre le matériel et l'utilisateur.

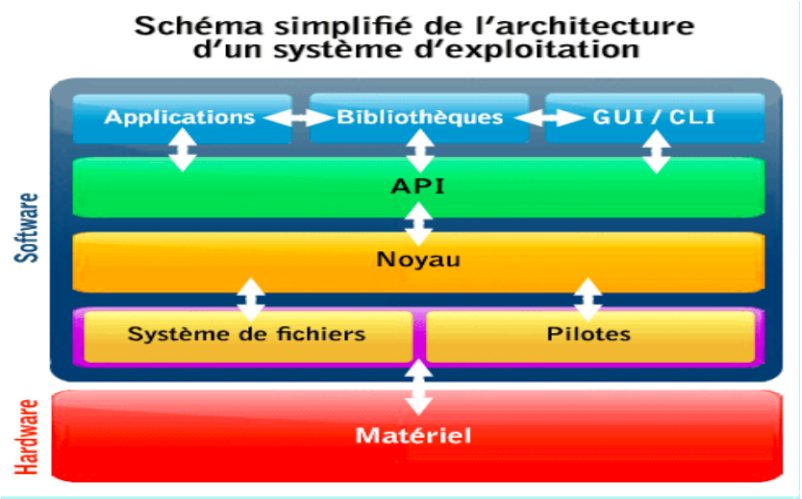


Figure 5 Les composants d'un système d'exploitation

Un système d'exploitation est un **logiciel** qui gère le matériel informatique et les ressources logicielles, et qui fournit des services communs aux programmes informatiques.

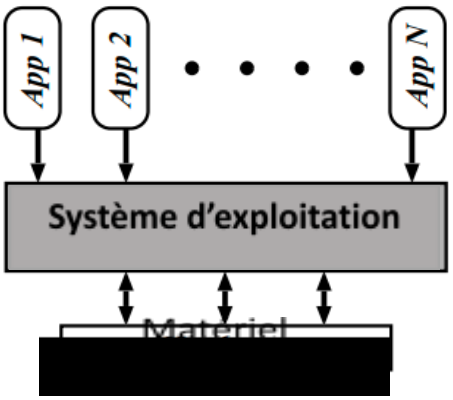


Figure 3 Position du système d'exploitation

Le rôle principal d'un système d'exploitation, c'est **le partage des ressources**. Il se fait entre les programmes appelés plus justement les processus. Ce rôle de policier du système d'exploitation permet d'éviter les conflits d'utilisation de la mémoire, des périphériques d'entrées/sorties, des interfaces réseau, etc.

- ✔ **Exemple 1**

On peut facilement imaginer ce qui arriverait si trois programmes essayaient d'imprimer en même temps sans qu'un certain ordre ne soit respecté. Ce travail d'alternance assuré par le système d'exploitation permet de mettre de l'ordre dans un chaos potentiel.

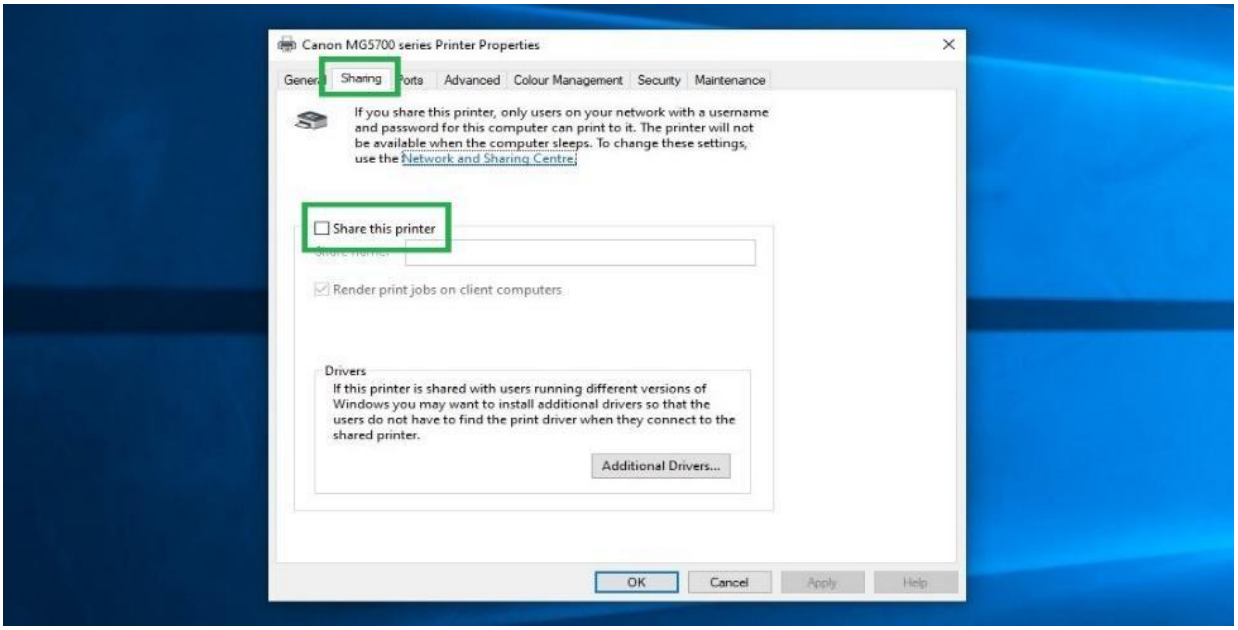


Figure 4 Exemple partage Imprimante

✔ **Exemple 2**

De plus, lorsque l'ordinateur est utilisé par plusieurs usagers (presque tout le temps), le partage de la mémoire et surtout sa protection demeure une priorité absolue. En tout temps, un bon système d'exploitation connaît l'utilisateur d'une ressource, ses droits d'accès, son niveau de priorité.

5. Les composants d'un système d'exploitation

Le système d'exploitation est composé d'un ensemble d'applications permettant de gérer les interactions avec le matériel. Parmi cet ensemble de logiciels, on distingue généralement les éléments suivants :

- ✔ **Le noyau** (kernel) représentant les fonctions fondamentales du système d'exploitation telles que la gestion de la mémoire, des processus, des fichiers, des entrées-sorties principales et des fonctionnalités de communication.
- ✔ **L'interpréteur de commande** (**shell** - coquille par opposition au noyau) permettant la communication avec le système d'exploitation par l'intermédiaire d'un langage de commandes, afin de permettre à l'utilisateur de piloter les périphériques en ignorant tous des caractéristiques du matériel qu'il utilise, de la gestion des adresses physiques, etc.
    - **Exemple** de commandes :
        - *ls* : lister les répertoires et les fichiers du répertoire courant
        - *mv* x y : changer le nom du fichier/répertoire « x » en « y »
- ✔ **Le système de fichiers** (file system) permet d'enregistrer les fichiers dans une arborescence.

Les fonctionnalités du système d'exploitation

Le système d'exploitation offre une suite de services généraux facilitant la création et l'utilisation de logiciels applicatifs. Les services offerts sont en rapport avec l'utilisation des ressources de l'ordinateur.

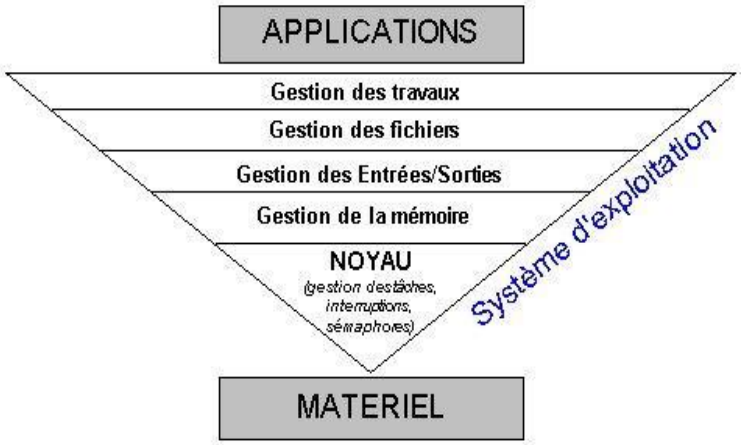


Figure 6 Les fonctionnalités du système d'exploitation

1. ***Gestion des processeurs***. Gérer l'allocation des processeurs entre les différents programmes grâce à un algorithme d'ordonnancement.

   Gestion des processus. Gérer l'exécution des processus en leur affectant les ressources nécessaires à leur bon fonctionnement.
2. ***Gestion des mémoires***. Gérer l'espace mémoire alloué à chaque processus.
3. ***Gestion des périphériques***. Contrôler l'accès des processus aux ressources matérielles par l'intermédiaire des pilotes.
4. ***Gestion des fichiers*.** Gérer la lecture et l'écriture dans le système de fichiers et les droits d'accès aux fichiers par les utilisateurs et les applications.

6. Générations des systèmes d'exploitation

Dans cette section, nous présentons les types de systèmes d'exploitation. Les systèmes d'exploitation informatiques modernes peuvent être classés en six groupes, qui se distinguent par la nature de l'interaction qui a lieu entre l'utilisateur de l'ordinateur et son programme pendant son traitement. Ces groupes sont appelés systèmes d'exploitation par lots, à temps partagé et en temps réel.

1. **Mono utilisateur/Mono tâche**, c'est le cas le plus simple : Un seul utilisateur à la fois et une seule tâche à la fois. Les systèmes d'exploitation des premiers micro-ordinateurs ne dépassaient pas ce niveau de complexité.
2. **Multitâches,** il assure le partager le temps du processeur entre plusieurs programmes. Le passage de l'exécution d'un programme à un autre peut être initié :
   i. Par les programmes eux-mêmes (coopératif)
   ii. Par le S.E (préemptif)
3. **Multiutilisateurs (temps partagé),** Plusieurs utilisateurs peuvent utiliser simultanément une même machine pour des applications similaires ou différentes. Et chaque utilisateur à l'impression d'être le seul à utiliser l'ordinateur.

4. **Multiprocesseurs,** un processeur central (maître) peut coordonner une série de tâches sur plusieurs autres processeurs (esclaves). Organisation à l'extérieur d'une série de tâches sur plusieurs processeurs (systèmes répartis).
5. **Temps réels,** servent pour le pilotage et le contrôle des déroulements externes (exemple central électrique), doivent garantir des temps de réactions données pour des signaux extérieurs urgents.
6. **Distribués,** doivent permettre l'exécution d'un seul programme sur plusieurs machines. Distribuer les tâches et les remettre ensemble. Pour gros calculs, exemple inversion de grandes matrices.

7. Notions de base

Dans cette section, nous allons introduire les notions les plus importantes qui sont communes aux systèmes d'exploitation.

7.1. Processus

Un processus (en anglais, *process*), en informatique, est un programme en cours d'exécution par un ordinateur. Auquel on associe :

- **Un Contexte du processeur (CPU) : l'ensemble des registres par exemple (CO : compteur ordinal, registre d'état. PSW : Processor Status Word)**

- **Un Contexte de la Mémoire Centrale : segments de code, segments de données, ...**

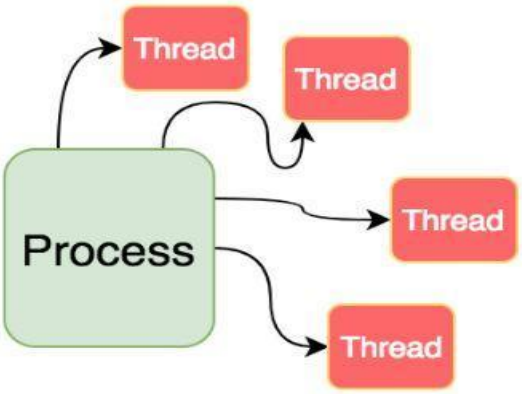


Figure 7 Un processus dans le système d'exploitation

7.2. Les états d'un processus

Un processus est un programme actif. On peut également dire qu'il s'agit d'un programme en cours d'exécution. Il est plus que le code de programme, car il comprend le compteur de programme, la pile de processus, les registres, le code de programme, etc. Par rapport à cela, le code du programme n'est qu'une section de texte. Un processus passe par différents états au cours de son exécution. Ces états peuvent être différents selon les systèmes d'exploitation.

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

1. **Elu** : (en cours d'exécution) : si le processus est en cours d'exécution
2. **Bloqué** : attente qu'un événement se produit ou une ressource pour pouvoir continuer
3. **Prêt** : si le processus dispose de toutes les ressources nécessaires à son exécution, à l'exception du processeur

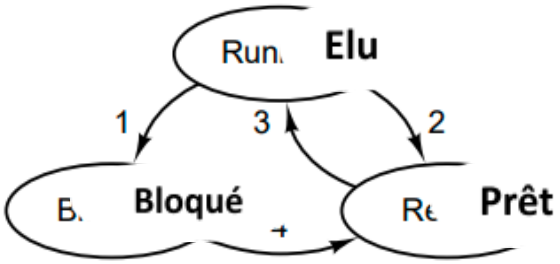
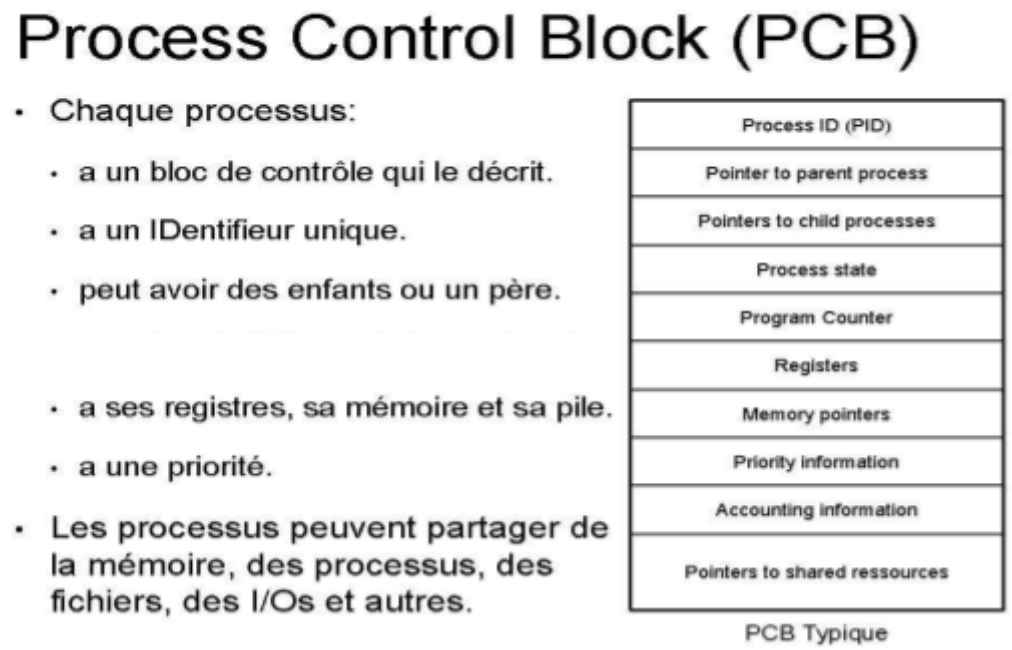


Figure 8 Les états d'un processus

7.3. Représentation interne des processus

Au moment du chargement d'un programme exécutable, le SE lui crée une structure représentant le processus associé. Cette structure devrait contenir toutes les informations nécessaires permettant l'évolution dynamique du processus au sein du SE.

Pour manipuler tous les processus, le SE détient d'une table de processus dont chaque entrée contient un pointeur vers un PCB (***Process Control Block***) d'un processus.



PCB Typique

✔ **Pointeur** - Il s'agit d'un pointeur de pile qui doit être sauvegardé lorsque le processus passe d'un état à un autre pour conserver la position actuelle du processus.

✔ **État du processus** - Il stocke l'état respectif du processus.

✔ **Numéro de processus** - Chaque processus se voit attribuer un identifiant unique connu sous le nom d'ID de processus ou **PID** qui stocke l'identifiant du processus.

✔ **Compteur de programme** - Il stocke le compteur qui contient l'adresse de l'instruction suivante qui doit être exécutée pour le processus.

✔ **Registre** - Il s'agit des registres du processeur qui comprennent : l'accumulateur, la base, les registres et les registres d'usage général.

- ✔ **Limites de mémoire** - Ce champ contient les informations sur le système de gestion du mémoire utilisé par le système d'exploitation. Cela peut inclure les tables de pages, les tables de segments, etc.
- ✔ **Liste des fichiers ouverts** - Cette information comprend la liste des fichiers ouverts pour un processus.

7.4. Tache (thread)

Dans un système d'exploitation, un processus est une tâche ou un programme qui peut être exécuté par l'ordinateur. Pensez à l'application MS Word, qui est un processus exécuté sur un ordinateur. Mais une application peut faire plus d'une chose à la fois, ce qui signifie qu'un processus donné dans un système d'exploitation peut avoir un ou plusieurs threads. Les *threads* représentent le traitement réel du code.

Un processus possède ses propres registres système et son propre pile mémoire, ce qui l'aide à exécuter des threads. Les threads sont parfois appelés processus légers. Le graphique ci-dessous montre un processus avec un seul thread à l'intérieur :

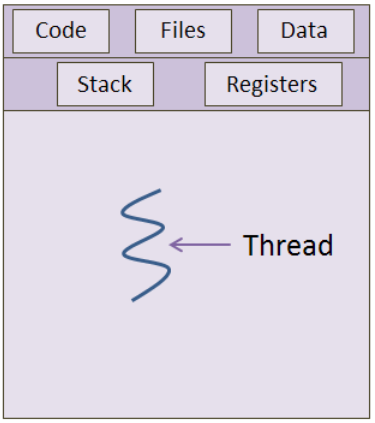


Figure 9 Représentation graphique d'un thread

7.5. Différences entre processus et thread

Les processus et les threads fonctionnent ensemble, mais ils présentent de nombreuses différences entre eux. Généralement, les processus sont assez lourds (comme MS Word), tandis que les threads sont plus légers (comme l'option de sauvegarde en arrière-plan). Le tableau ci-dessous met en évidence certaines des différences entre les deux.

Processus	**Threads**
Lors de la commutation d'un processus, les ressources du système d'exploitation sont requises	Aucune ressource du système d'exploitation n'est nécessaire pour le changement de fil.
Si un processus est bloqué, les autres processus en attente dans la file d'attente sont également bloqués.	Si un thread est bloqué, un autre thread dans le même processus peut toujours s'exécuter.
Chaque processus utilise le même code et possède sa propre mémoire.	Tous les threads peuvent partager des fichiers et des processus enfants.
Une application comportant plusieurs processus utilisera davantage de ressources système.	Les processus utilisant plusieurs threads utilisent moins de ressources système.
Les processus nécessitent plus de temps pour être terminés.	Les fils nécessitent moins de temps pour être terminés.
Les processus ont des segments de données et de code indépendants.	Un thread partage le segment de données, le segment de code, les fichiers, etc. avec ses pairs.

Tous les différents processus sont traités séparément par le système d'exploitation.	Tous les threads pairs du niveau utilisateur sont traités comme une seule tâche par le système d'exploitation.
Les processus individuels sont indépendants les uns des autres.	Les threads font partie d'un processus et sont donc dépendants.

Tableau 1 Différences entre processus et thread

Lorsqu'un processus présente un code un petit peu long ou compliqué, il peut être décortiqué en un ensemble d'unités de traitements élémentaires ayant une cohérence logique dont la fonction est bien déterminée. Cette unité est appelée une tâche.

- ✔ Un thread est une subdivision d'un processus
- ✔ Les différents threads d'un processus partagent l'espace adressable et les ressources d'un processus
- ✔ Les threads sont des processus légers exécutés « à l'intérieur » d'un processus
- ✔ L'exécution des threads est concurrente
- ✔ Il existe toujours au moins un thread : le thread principal

✔ La durée de vie d'un thread ne peut pas dépasser celle du processus qui l'a crée

7.6. Multi-threading

Une application multithread est un logiciel qui dès sa conception a été partagé en différentes sous-applications appelées threads.

A l'inverse de ce qui passe dans le cadre multitâche où chaque processus dispose d'une partie distincte de la mémoire pour s'exécuter indépendamment des autres. Les threads issus de même application partagent un même espace mémoire.

- ✔ Un thread pour interagir avec l'utilisateur ;
- ✔ Un thread pour reformater en arrière-plan ;
- ✔ Un thread pour sauvegarder périodiquement le document ;
- ✔ Un thread la vérification automatique de l'orthographe et de la grammaire.

8. Ressource partager

Une ressource désigne toute entité dont a besoin un processus pour s'exécuter.

- ✔ Ressource matérielle (processeur, périphérique)
- ✔ Ressource logicielle (fichier, variable).

Une ressource est caractérisée :

- ✔ par un état : libre / occupée
- ✔ par son nombre de points d'accès (nombre de processus pouvant l'utiliser en même temps)

Une ressource critique à un seul point d'accès.

9. Exercices

1. Les services et fonctions fournis par un système d'exploitation peuvent être divisés en deux catégories principales. Décrivez brièvement ces deux catégories et expliquez en quoi elles diffèrent.

   Réponse :

   - Une catégorie de services fournis par un système d'exploitation consiste à assurer la protection entre les différents processus qui s'exécutent simultanément dans le système. Les processus ne sont autorisés à accéder qu'aux emplacements de mémoire associés à leurs espaces d'adressage. De même, les processus ne sont pas autorisés à corrompre les fichiers associés à d'autres utilisateurs. Un processus n'est pas non plus autorisé à accéder directement aux périphériques sans l'intervention du système d'exploitation.
   - La deuxième catégorie de services fournis par un système d'exploitation consiste à offrir de nouvelles fonctionnalités qui ne sont pas prises en charge directement par le matériel sous-jacent. La mémoire virtuelle et les systèmes de fichiers sont deux exemples de nouveaux services fournis par un système d'exploitation.

2. Citez cinq services fournis par un système d'exploitation qui sont conçus pour faciliter l'utilisation du système informatique et rendre l'utilisation du système

informatique plus pratique pour les utilisateurs. Dans quels cas, il serait impossible pour les programmes de niveau utilisateur de fournir ces services ? Expliquez.

Réponse :

Exécution du programme. Le système d'exploitation charge le contenu (ou les sections) d'un fichier en mémoire et commence son exécution. Un programme de niveau utilisateur ne pouvait pas être sûr d'allouer correctement le temps CPU.

- **Opérations d'entrée/sortie.** Les disques, les bandes, les lignes série et les autres périphériques doivent être communiqué à un niveau très bas. L'utilisateur doit seulement spécifier le périphérique et l'opération à effectuer sur celui-ci, tandis que le système convertit cette demande en commandes spécifiques au périphérique ou au contrôleur. Les programmes de niveau utilisateur ne peuvent pas être sûrs de n'accéder qu'aux dispositifs auxquels ils doivent avoir accès et de n'accéder qu'aux dispositifs auxquels ils doivent avoir accès. Auxquels ils devraient avoir accès et qu'ils n'y accèdent que lorsqu'ils sont inutilisés.
- **Manipulation du système de fichiers.** Il y a de nombreux détails dans la création, la suppression, l'allocation et le nom des fichiers, suppression, l'allocation et le nommage des fichiers que les utilisateurs ne devraient

pas avoir à effectuer. Les blocs d'espace disque sont utilisés par les fichiers et doivent être suivis. La suppression d'un fichier nécessite la suppression des informations relatives au fichier de nom et la libération des blocs alloués. Libérer les blocs alloués. Les protections doivent également être vérifiées pour garantir un accès correct aux fichiers. Les programmes utilisateurs ne pouvaient ni garantir, ne pouvaient ni garantir le respect des méthodes de protection, ni être sûrs d'allouer uniquement les blocs libres et désallouer les blocs lors de la suppression du fichier.

- **Les communications.** Le passage des messages entre les systèmes nécessite que les messages soient transformés en paquets d'information, envoyés au contrôleur de réseau, transmis sur un support de communication et réassemblés par le système de destination. L'ordre des paquets et la correction des données doivent avoir lieu. Encore une fois, les programmes utilisateurs peuvent ne pas coordonner l'accès au périphérique réseau, ou bien, ils peuvent recevoir des paquets destinés à d'autres processus.
- **La détection des erreurs.** La détection des erreurs se fait à la fois au niveau matériel et logiciel. Au niveau matériel, tous les transferts de données doivent être inspectés pour s'assurer que les données n'ont pas été corrompues en transit. Toutes les données sur les supports

doivent être vérifiées pour s'assurer qu'elles n'ont pas changé depuis qu'elles ont été écrites sur le support. Au niveau du logiciel, la cohérence des données des supports logiciels, il faut vérifier la cohérence des données sur les supports. Blocs de stockage alloués et non alloués, correspondent-ils au nombre total sur le périphérique. Les erreurs sont souvent indépendantes du processus (par exemple, la corruption de données sur un disque), donc il doit y avoir un programme global (le système d'exploitation) qui gère tous les types d'erreurs. De plus, en faisant traiter les erreurs par le système d'exploitation, les processus n'ont pas besoin de contenir du code pour corriger toutes les erreurs possibles sur un système

3. C’est quoi un microkernel ?

   Réponse :

   En informatique, un micro-noyau (fréquemment abrégé en *µ-noyau*) est la quantité quasi minimale de logiciel qui peut fournir les mécanismes nécessaires à la mise en œuvre d'un système d'exploitation (SE). Ces mécanismes comprennent la gestion de l'espace d'adressage de bas niveau, la gestion des threads et la communication interprocessus (IPC).

4. Quel est le principal avantage de l'approche *microkernel* dans la conception des systèmes ? Comment les programmes utilisateurs et les services système

interagissent-ils dans une architecture microkernel ? Quels sont les inconvénients de l'utilisation de l'approche microkernel ?

Réponse :

Les avantages sont généralement les suivants :

(a) l'ajout d'un nouveau service ne nécessite pas de modifier le noyau,

(b) il est plus sûr, car davantage d'opérations sont effectuées en mode utilisateur qu'en mode noyau,

(c) une conception et une fonctionnalité du noyau plus simples se traduisent généralement par un système d'exploitation plus fiable. Les programmes utilisateur et les services système interagissent dans une architecture *microkernel* utilisent des mécanismes de communication inter-processus tels que la messagerie. Ces messages sont véhiculés par le système d'exploitation. Le principal inconvénient de l'architecture microkernel est la surcharge associée à la communication inter-processus et l'utilisation fréquente des fonctions de messagerie du système d'exploitation afin de permettre au processus utilisateur et au service système d'interagir entre eux.

5. Quel est le principal avantage pour un concepteur de système d'exploitation d'utiliser une architecture de machine virtuelle ? Quel est le principal avantage pour un utilisateur ?

Réponse :

Le système est facile à déboguer et les problèmes de sécurité sont faciles à résoudre. Les machines virtuelles constituent également une bonne plate-forme pour la recherche sur les systèmes d'exploitation, puisque de nombreux systèmes d'exploitation différents peuvent fonctionner sur un système physique.

6. Le système d'exploitation expérimental Synthesis possède un assembleur intégré au noyau. Pour optimiser les performances des appels système, le noyau assemble les routines dans l'espace du noyau afin de minimiser le chemin que l'appel système doit emprunter à travers le noyau. Cette approche est l'antithèse de l'approche en couches, dans laquelle le chemin à travers le noyau est étendu pour faciliter la construction du système d'exploitation. Discutez des avantages et des inconvénients de l'approche Synthèse pour la conception du noyau et l'optimisation des performances du système.

   Réponse :

   Synthesis est impressionnant en raison des performances qu'il permet d'atteindre grâce à la compilation à la volée. Malheureusement, il est difficile de déboguer les problèmes au sein du noyau en raison de la fluidité du code. De plus, cette compilation est spécifique au système, ce qui rend Synthesis difficile à porter (un nouveau compilateur doit être écrit pour chaque architecture).

Chapitre 2 : Synchronisation de processus.

- Problème de l'accès concurrent à des ressources et sections critiques

  (Problème de l'exclusion mutuelle)

- Outils de synchronisation :
  - Événements, Verrous
  - Sémaphores
  - Moniteurs
  - Régions critiques.

1. Introduction

Lorsque deux ou plusieurs processus doivent partager un objet, un mécanisme d'arbitrage est nécessaire pour qu'ils n'essaient pas de l'utiliser en même temps. L'objet particulier partagé n'a pas un grand impact sur le choix de tels mécanismes.

Le problème de l'évitement des conditions de course peut également être formulé de manière abstraite. Une partie du temps, un processus est occupé à faire des calculs internes et d'autres choses qui ne conduisent pas à des conditions de course. Cependant, il arrive qu'un processus accède à la mémoire partagée ou à des fichiers, ou qu'il fasse d'autres choses critiques qui peuvent conduire à des situations de concurrence. La partie du programme où l'on accède à la mémoire partagée est appelée la Section Critique (SC). Si nous pouvions faire en sorte que deux processus ne se trouvent jamais dans leur section critique en même temps, nous pourrions éviter les situations de concurrence.

Bien que cette exigence permette d'éviter les conditions de course, elle n'est pas suffisante pour que les processus parallèles coopèrent correctement et efficacement en utilisant des données partagées. Quatre conditions doivent être remplies pour obtenir une bonne solution :

- Deux processus ne peuvent pas se trouver simultanément à l'intérieur de leurs sections critiques.

- Aucune hypothèse ne peut être faite sur les vitesses ou le nombre de processeurs.
- Aucun processus fonctionnant en dehors de sa SC ne peut bloquer d'autres processus.
- Aucun processus ne devrait avoir à attendre éternellement pour entrer dans son SC.

✔ **Exemple** : Considérez les exemples suivants :

- Deux processus partageant une imprimante doivent l'utiliser à tour de rôle ; s'ils tentent de l'utiliser simultanément, la sortie des deux processus peut être mélangée dans un fouillis arbitraire qui ne sera probablement d'aucune utilité.
- Deux processus tentant de mettre à jour le même compte bancaire doivent se succéder ; si chaque processus lit le solde actuel à partir d'une base de données, le met à jour, puis le réécrit, l'une des mises à jour sera perdue. Voir en détaille ici

```
procédureCredDeb_Compte
(Numéro_Compte, opération, somme)
début--lecture dans le fichier Fichier_Compte
-- du solde du compte Numéro_Compte
Lire (Fichier_Compte, Numéro_Compte, solde_compte);
        si (opération = débiter)
                alors solde_compte := solde_compte - somme;
                sinonsolde_compte := solde_compte + somme;
        fsi
-- écriture dans le fichier Fichier_Compte --
du nouveau solde du compte Numero_Compte
Ecrire (Fichier_Compte, Numéro_Compte, solde_compte);
fin
```

```
procédureDonner_Solde
(Numéro_Compte, var Solde)
début-- lecture dans le fichier Fichier_Compte
-- du solde du compte Numéro_Compte
Lire (Fichier_Compte, Numéro_Compte, solde_compte);
Solde := solde_compte;
fin
```

Les deux exemples ci-dessus peuvent être résolus s'il existe un moyen pour chaque processus d'exclure l'autre de l'utilisation de l'objet partagé pendant les sections critiques du code. Ainsi, le problème général est décrit comme le problème de l'exclusion mutuelle. Les utilisateurs doivent faire face à un certain nombre de nouveaux problèmes sur les systèmes qui permettent de construire des programmes à partir de plusieurs processus simultanés. Certains de ces problèmes ont été largement étudiés et un certain nombre de solutions sont connues.

Bien que la plupart des systèmes d'exploitation ne prennent en charge que quelques mécanismes de base pour aider à résoudre les problèmes de programmation simultanée, de nombreux langages de programmation expérimentaux ont incorporé de tels mécanismes. Certaines de ces langues ont dépassé le stade expérimental et pourraient devenir largement utilisées ; parmi ceux-ci, le **shell** Unix et le langage de programmation **Ada** sont particulièrement importants.

La notion de **synchronisation** des processus est indispensable pour un système d'exploitation efficace et robuste. Sur la base de la synchronisation, les processus sont classés dans l'un des deux types suivants :

- **Processus indépendant** : l'exécution d'un processus n'affecte pas l'exécution des autres processus.
- **Processus coopératif** : L'exécution d'un processus affecte l'exécution d'autres processus.

Le problème de synchronisation des processus se pose également dans le cas du processus coopératif parce que les ressources sont partagées dans les processus coopératifs.

2. Condition de concurrence

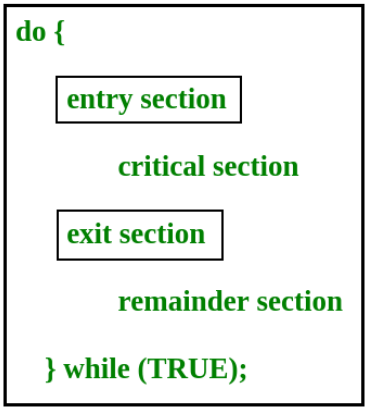
Lorsque plusieurs processus exécutent le même code ou accèdent à la même mémoire ou à toute variable partagée dans cette condition, il est possible que la sortie ou la valeur de la variable partagée soit erronée, de sorte que tous les processus faisant la course pour dire que ma sortie est correcte, cette condition connue comme une condition de course. Plusieurs processus accèdent et traitent les manipulations sur les mêmes données simultanément, puis le résultat dépendent de l'ordre particulier dans lequel l'accès a lieu.

Une condition de concurrence est une situation qui peut se produire à l'intérieur d'une section critique. Cela se produit lorsque le résultat de l'exécution de plusieurs threads dans la section critique diffère selon l'ordre dans lequel les threads s'exécutent.

Les conditions de concurrence dans les sections critiques peuvent être évitées si la section critique est traitée comme une instruction atomique. En outre, une synchronisation correcte des threads à l'aide de verrous ou de variables atomiques peut empêcher les conditions de concurrence.

2.1. Problème de section critique

La section critique est un segment de code accessible par un seul processus à la fois. La section critique contient des variables partagées qui doivent être synchronisées pour maintenir la cohérence des variables de données.



Dans la section d'entrée, le processus demande une entrée dans la **section critique.**

Toute solution au problème de la section critique doit satisfaire trois exigences :

- **Exclusion mutuelle** : si un processus s'exécute dans sa section critique, aucun autre processus n'est autorisé à s'exécuter dans la section critique.
- **Progression** : si aucun processus ne s'exécute dans la section critique et que d'autres processus attendent en dehors de la section critique, seuls les processus qui ne s'exécutent pas dans leur section restante peuvent participer à la décision de celui qui entrera ensuite dans la section critique, et la sélection ne peut pas être reportée indéfiniment.
- **Attente limitée** : une limite doit exister sur le nombre de fois où d'autres processus sont autorisés à entrer dans leurs sections critiques après qu'un

processus a fait une demande pour entrer dans sa section critique et avant que cette demande ne soit accordée.

2.2. Solution de Peterson

La solution de Peterson est une solution logicielle classique au problème de la section critique. Dans la solution de Peterson, nous avons deux variables partagées :

- ***boolean flag[i]*** : Initialisé à FALSE, initialement aucun processus n'est intéressé à entrer dans la SC ;
- ***int turn*** **:** Le processus dont le tour est d'entrer dans la section critique.

```
do {

    flag[i] = TRUE ;
    turn = j ;
    while (flag[j]  &&  turn == j) ;

        critial section

    flag[i] = FALSE ;

        remainder section

} while (TRUE) ;
```

La solution de Peterson préserve les trois conditions :

- L'exclusion mutuelle est assurée, car un seul processus peut accéder à la section critique à tout moment.
- La progression est également assurée, parce qu'un processus en dehors de la section critique n'empêche pas les autres processus d'entrer dans la section critique.

- L'attente limitée est préservée comme chaque processus à une chance équitable.

Inconvénients de la solution de Peterson

- Cela implique une attente active
- Il est limité à deux processus.

2.3. L'algorithme de Dekker

Cela a été réalisé pour la première fois par TJ Dekker et publié (par Dijkstra) en 1965. L'algorithme de Dekker utilise l'attente occupée et ne fonctionne que pour deux processus. L'idée de base est que les processus enregistrent leurs besoins à entrer dans une section critique (dans des variables booléennes) et qu'ils se relaient (à l'aide d'une variable appelée "tour") lorsque les deux ont besoin d'une entrée en même temps. La solution de Dekker est illustrée ci-dessus.

```
Var  process1, process2 :(interieur, exterieur);
        tour  : 1..2;
  process1:= exterieur;
  process2:= exterieur;
```

Processus P1

```
    ----------
process1:= interieur;
tour:= 2;
test: tantque process2 = interieur et tour = 2 faire
        aller à test
       Fintantque
 <section critique>
process1:= exterieur;
  ----------
```

Processus P2

```
    ----------
process2 := interieur;
tour := 1;
test: tantque process1 = interieur et tour = 1 faire
            allerà test
      Fintantque
<section critique>;
process2:=exterieur;
  ----------
```

La solution de **Dekkers** au problème d'exclusion mutuelle nécessite que chacun des processus en conflit ait un identifiant de processus unique appelé "moi" qui est transmis aux opérations d'attente et de signal.

Il suppose que si deux processus tentent d'écrire deux valeurs différentes dans le même emplacement mémoire en même temps, l'une ou l'autre valeur sera stockée, et non un mélange des deux. C'est ce qu'on appelle l'hypothèse de mise à jour atomique.

3. Synchronisation matérielle : Déverrouillage et verrouillage, Test et Set, Swap

Les problèmes de synchronisation des processus se produisent lorsque deux processus s'exécutant simultanément partagent les mêmes données ou la même variable. La valeur de cette variable peut ne pas être mise à jour correctement avant d'être utilisée par un deuxième processus. Une telle situation est connue sous le nom de "Race Around Condition". Il existe des solutions logicielles et matérielles à ce problème. Dans cet article, nous allons parler de la solution matérielle la plus efficace pour résoudre les problèmes de synchronisation des processus et de son implémentation.

Il existe trois algorithmes dans l'approche matérielle de la résolution du problème de synchronisation des processus :

- ☐ **Test and Set**
- ☐ **Swap**
- ☐ **Déverrouiller et verrouiller**

Les instructions matérielles dans de nombreux systèmes d'exploitation aident à résoudre efficacement les problèmes de sections critiques

4. Solution matérielle « testandset »

TestAndSet est une solution matérielle au problème de synchronisation. Dans TestAndSet, nous avons une variable de verrouillage partagé qui peut prendre l'une des deux valeurs, 0 ou 1.

0	Déverrouille r
1	Verrouiller

Avant d'entrer dans la section critique, un processus s'enquiert du verrou. S'il est verrouillé, il continue d'attendre jusqu'à ce qu'il se libère et s'il n'est pas verrouillé, il prend le verrou et exécute la section critique.

Dans *TestAndSet*, l'exclusion mutuelle et la progression sont préservées, mais l'attente limitée ne peut pas être préservée.

```
TAS(C)
    begin
       < verrouiller l'accès à C >;
        lire C
       if  C = 0 then  begin
                          C := 1
                          co := co + 2   % co = compteur  ordinal %   (1)
                       end               % ou compteur d'instructions %
             else co := co + 1           %          (2)                %
       endif
         < libérer l'accès à C >
    end
```

Par exemple, la programmation de l'accès concurrent à un compte bancaire peut être :

```
Processus Debit                    Processus Credit
        .                                  .
        .                                  .
        .                                  .
Test: TAS(C);                      Test: TAS(C);
      Aller a Test;                      aller a Test;
      a1: Rla := Compte ;                b1: Rlb := Compte ;
      a2: Rla := Rla + C ;               b1: Rlb := Rlb - D ;
      a3: Compte := Rla ;                b3: Compte := Rlb ;
      C := 0;                            C := 0;
```

▯ Limites

- **Attente active** (« busy waiting ») Verrou actif (« spin lock »)
- **Inversion de priorités :** un processus prioritaire bloqué derrière un processus standard

5. Swap :

L'algorithme swap ressemble beaucoup à l'algorithme *TestAndSet*. Au lieu de définir directement le verrou à *true* dans la fonction swap, la clé est définie à *true* et ensuite échangée avec le verrou. Ainsi, lorsqu'un processus se trouve dans la section critique, aucun autre processus ne peut y entrer, car la valeur de *lock* est vraie. L'exclusion mutuelle est assurée. De nouveau, en dehors de la section critique, *lock* est changé en *false*, donc tout processus qui le trouve ne peut pas entrer dans la section critique. La progression est assurée. Cependant, l'attente limitée n'est pas garantie pour la même raison.

```
// Shared variable lock initialized to false

// and individual key initialized to false;

boolean lock;

Individual key;

void swap(boolean &a, boolean &b){

    boolean temp = a;

    a = b;

    b = temp;

}

while (1){

    key = true;

    while(key)

         swap(lock,key);

    critical section

    lock = false;

    remainder section

}
```

6. Unlock and Lock

L'algorithme Unlock and Lock utilise TestAndSet pour réguler la valeur de lock mais il ajoute une autre valeur, waiting[i], pour chaque processus qui vérifie si un processus a été en attente ou non. Une file d'attente prête est maintenue en ce qui concerne le

processus dans la section critique. Tous les processus qui arrivent ensuite sont ajoutés à la file d'attente prête en fonction de leur numéro de processus, pas nécessairement de manière séquentielle. Une fois que le ième processus sort de la section critique, il ne met pas le verrou à faux pour que n'importe quel processus puisse accéder à la section critique maintenant, ce qui était le problème avec les algorithmes précédents. Au lieu de cela, il vérifie s'il y a un processus en attente dans la file d'attente. La file d'attente est considérée comme circulaire. j est considéré comme le processus suivant dans la file d'attente et la boucle while vérifie du jème processus au dernier processus et à nouveau de 0 au (i-1)ème processus s'il y a un processus en attente pour accéder à la section critique. S'il n'y a pas de processus en attente, la valeur du verrou est changée en false et tout processus suivant peut accéder à la section critique. S'il y en a un, alors la valeur d'attente de ce processus est changée en false, de sorte que la première boucle while devient false et qu'il peut entrer dans la section critique. Cela garantit une attente bornée. Le problème de la synchronisation des processus peut donc être résolu par cet algorithme.

```
// Shared variable lock initialized to false
// and individual key initialized to false

boolean lock;
Individual key;
Individual waiting[i];
```

```
while(1){
    waiting[i] = true;
    key = true;
    while(waiting[i] && key)
        key = TestAndSet(lock);
    critical section
    j = (i+1) % n;
    while(j != i && !waiting[j])
         j = (j+1) % n;
    if(j == i)
         lock = false;
    else
         waiting[j] = false;
    remainder section}
```

7. Sémaphores

Le sémaphore a été proposé par Dijkstra en 1965, une technique très importante pour gérer des processus concurrents en utilisant une simple valeur entière, connue sous le nom de sémaphore. Le sémaphore est simplement une variable entière partagée entre les threads. Cette variable est utilisée pour résoudre le problème de la section critique et pour réaliser la synchronisation des processus dans l'environnement multitraitement.

Un sémaphore est un mécanisme de signalisation, et un thread qui attend un sémaphore peut être signalé par un autre thread. Ceci est différent d'un *mutex* car le *mutex* ne peut être signalé que par le thread qui a appelé la fonction d'attente.

- Un sémaphore utilise deux opérations atomiques, attendre et signaler pour la synchronisation des processus.
- Un sémaphore est une variable entière, accessible uniquement via deux opérations :
    - ***wait()* dénomé P()**
    - ***signal() dénomé V()***

Les sémaphores peuvent être utilisés pour résoudre le problème d'exclusion mutuelle au niveau de l'utilisateur ; par exemple, considérez les fragments de code illustrés en bas.

```
var shared: ...      { the shared variable needing mutual
    mutex: semaphore { a shared semaphore; initial count
       .
       .
       .

repeat { a typical process using shared }
       .
       .     { code outside the critical section;
       .       \dreferences to shared are not allowed }\u

       wait( mutex ) { start of critical section };
           .
           .     { code inside the critical section;
           .       \dreferences to shared are allowed }\u

       signal( mutex ) { end of critical section };
       .
       .     { more code outside the critical section }
       .

until ...
```

Ce code utilise un sémaphore pour contrôler l'accès à une variable partagée, mais le sémaphore pourrait tout aussi bien être utilisé pour contrôler l'accès à un fichier ou un périphérique partagé.

7.1. Type de sémaphores

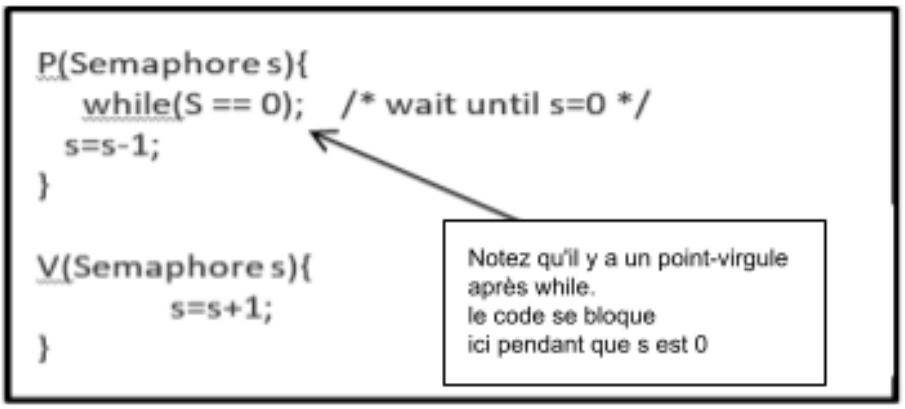
Il existe deux types de sémaphores : les sémaphores binaires et les sémaphores de comptage.

- ***Sémaphores binaires*** : ils ne peuvent être que 0 ou 1. Ils sont également connus sous le nom de verrous mutex, car les verrous peuvent fournir une exclusion mutuelle. Tous les processus peuvent partager le même sémaphore mutex initialisé à 1. Ensuite, un processus doit attendre que le verrou devienne 0. Ensuite, le processus peut rendre le sémaphore mutex 1 et démarrer sa section critique. Lorsqu'il termine sa section critique, il peut réinitialiser la valeur du sémaphore mutex à 0 et un autre processus peut entrer dans sa section critique.
- ***Sémaphores de comptage*** : ils peuvent avoir n'importe quelle valeur et ne sont pas limités à un certain domaine. Ils peuvent être utilisés pour contrôler l'accès à une ressource dont le nombre d'accès simultanés est limité. Le sémaphore peut être initialisé au nombre d'instances de la ressource. Chaque fois qu'un processus veut utiliser cette ressource, il vérifie si le nombre d'instances restantes est supérieur à zéro, c'est-à-dire si le processus a une instance disponible. Ensuite, le processus peut entrer dans sa section critique, diminuant ainsi la valeur du

sémaphore de comptage de 1. Une fois le processus terminé avec l'utilisation de l'instance de la ressource, il peut quitter la section critique, ajoutant ainsi 1 au nombre d'instances disponibles de la ressource.

Tout d'abord, regardez deux opérations qui peuvent être utilisées pour accéder et modifier la valeur de la variable sémaphore.

Un sémaphore simple ou binaire est un sémaphore qui n'a que deux valeurs, zéro et un ; lorsqu'elle est utilisée pour résoudre le problème d'exclusion mutuelle, la valeur zéro indique qu'un processus à l'usage exclusif de la ressource associée, et la valeur un indique que la ressource est libre. Alternativement, les états du sémaphore sont parfois nommés "réclamés" et "libres" ; l'opération d'attente réclame le sémaphore, l'opération de signal le libère. Les sémaphores binaires qui sont implémentés à l'aide de boucles d'attente actives sont de temps en temps appelés verrous tournants, car un processus qui attend l'entrée d'une section critique passe son temps à tourner autour d'une boucle d'interrogation très serrée.

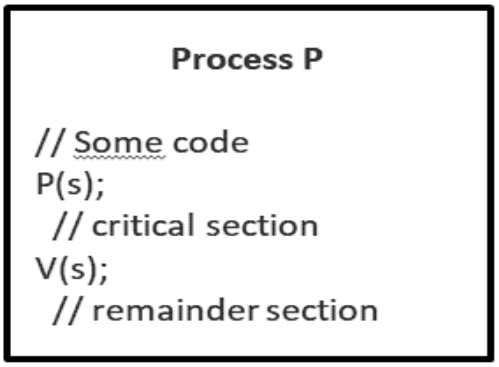


Quelques points concernant le fonctionnement *P* et *V :*

L'opération *P* est également appelée opération d'attente, de veille ou d'arrêt, et l'opération *V* est également appelée opération de signal, de réveil ou d'activation.

Les deux opérations sont atomiques et le(s) sémaphore(s) est toujours initialisé à un. Ici, atomique signifie que la variable sur laquelle la lecture, la modification et la mise à jour se produisent en même temps/moment sans préemption, c'est-à-dire qu'entre la lecture, la modification et la mise à jour, aucune autre opération n'est effectuée qui puisse modifier la variable.

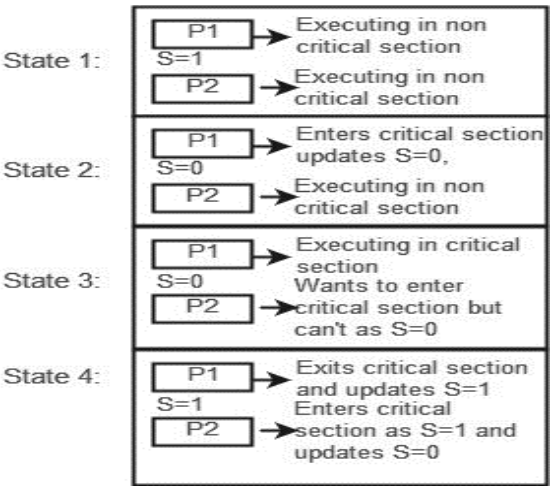
Une section critique est entourée par les deux opérations pour mettre en œuvre la synchronisation des processus. Voir l'image ci-dessous. La section critique du processus P se situe entre les opérations P et V.



Voyons maintenant comment il met en œuvre l'exclusion mutuelle. Soit deux processus P1 et P2 et un sémaphore s est initialisé à 1. Maintenant, si P1 entre dans sa section critique, la valeur du sémaphore s devient 0. Maintenant, si P2 veut entrer

dans sa section critique, il attendra jusqu'à ce que s > 0, cela ne peut se produire que lorsque P1 termine sa section critique et appelle l'opération V sur le sémaphore s.

De cette façon, l'exclusion mutuelle est réalisée. Regardez l'image ci-dessous pour plus de détails sur le sémaphore binaire.



7.2. Implémentation des sémaphores binaires

```
struct semaphore {
                    enum value(0, 1);
                    // q contains all Process Control Blocks (PCBs)
```

```
                // corresponding to processes got blocked
                // while performing down operation.
                Queue<process> q;
} P(semaphore s)
{
                if (s.value == 1) {
                        s.value = 0;
                }
                else {
                        // add the process to the waiting queue
                        q.push(P)
                        sleep();
                }
}
V(Semaphore s)
{
                if (s.q is empty) {
                        s.value = 1;
                }
                else {
                        // select a process from waiting queue
                        Process p=q.pop();
                        wakeup(p);
}
```

```
}
```

La description ci-dessus concerne un sémaphore binaire qui ne peut prendre que deux valeurs 0 et 1 et assurer une exclusion mutuelle. Il existe un autre type de sémaphore appelé sémaphore de comptage qui peut prendre des valeurs supérieures à un.

Supposons maintenant qu'il existe une ressource dont le nombre d'instances est de 4. Maintenant, nous initialisons S = 4 et le reste est le même que pour le sémaphore binaire. Chaque fois que le processus veut cette ressource, il appelle P ou attend la fonction et quand c'est fait, il appelle V ou la fonction de signal. Si la valeur de S devient nulle, un processus doit attendre que S devienne positif. Par exemple, supposons qu'il y ait 4 processus P1, P2, P3, P4, et qu'ils appellent tous l'opération d'attente sur S (initialisé avec 4). Si un autre processus P5 veut la ressource, il doit attendre que l'un des quatre processus appelle la fonction signal et que la valeur du sémaphore devienne positive.

7.3. Limite

L'une des plus grandes limitations du sémaphore est l'inversion de priorité et aussi Interblocage, supposons qu'un processus essaie de réveiller un autre processus qui n'est pas en état de veille. Par conséquent, un interblocage peut bloquer

indéfiniment. Le système d'exploitation doit garder une trace de tous les appels à attendre et à signaler le sémaphore.

Le principal problème avec les sémaphores est qu'ils nécessitent une attente active. Si un processus se trouve dans la section critique, les autres processus essayant d'entrer dans la section critique attendront jusqu'à ce que la section critique ne soit occupée par aucun processus.

- Chaque fois qu'un processus attend, il vérifie en permanence la valeur du sémaphore (regardez cette ligne pendant que (s==0); en fonctionnement P) et gaspille le cycle CPU.
- Il existe également un risque de "spinlock" car les processus continuent de tourner en attendant le verrouillage.

Pour éviter cela, une autre implémentation est fournie ci-dessous.

- Implémentation du sémaphore de comptage :

```
struct Semaphore {
                int value;
                // q contains all Process Control Blocks(PCBs)
                // corresponding to processes got blocked
                // while performing down operation.
                Queue<process> q;

} P(Semaphore s)
{
                s.value = s.value - 1;
                if (s.value < 0) {
                        // add process to queue
                        // here p is a process which is currently
executing
                        q.push(p);
                        block();
                }
                else
                        return;
}
```

```
V(Semaphore s)
{
                    s.value = s.value + 1;
                    if (s.value >= 0) {

                            // remove process p from queue
                            Process p=q.pop();
                            wakeup(p);
                    }
                    else
                            return;
}
```

Dans cette implémentation, chaque fois que le processus attend, il est ajouté à une file d'attente de processus associés à ce sémaphore. Cela se fait via l'appel système block() sur ce processus. Lorsqu'un processus est terminé, il appelle la fonction signal et un processus de la file d'attente est repris. Il utilise l'appel système wakeup().

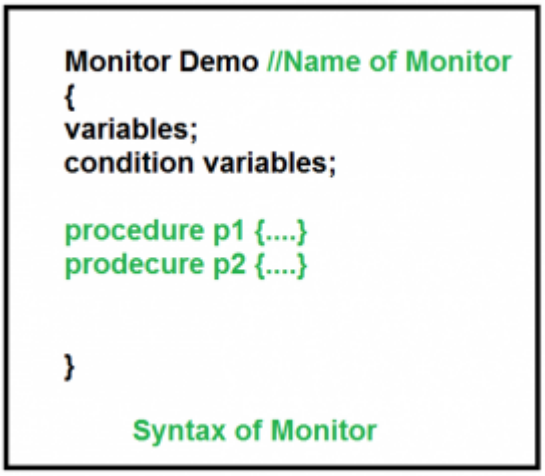
8. Utilisation du Moniteur dans la synchronisation de processus

Le moniteur est l'un des moyens de réaliser la synchronisation des processus. Le moniteur est pris en charge par des langages de programmation pour réaliser une

exclusion mutuelle entre les processus. Par exemple, les méthodes Java Synchronized. Java fournit les constructions wait() et notify().

1. Il s'agit d'un ensemble de variables de condition et de procédures combinées dans un type spécial de module ou de package.
2. Les processus exécutés en dehors du moniteur ne peuvent pas accéder à la variable interne du moniteur, mais peuvent appeler des procédures du moniteur.
3. Un seul processus à la fois peut exécuter du code à l'intérieur des moniteurs.

8.1. Syntaxe :



Deux opérations différentes sont effectuées sur les variables de condition du moniteur.

- ✔ Wait
- ✔ Signal

```
If (x block queue empty)
 // Ignore signal
else
 // Resume a process from block queue.
```

8.2. Opération

x.wait() Le processus effectuant une opération d'attente sur n'importe quelle variable de condition est suspendu. Les processus suspendus sont placés dans la file d'attente des blocs de cette variable de condition.

Chaque variable de condition à sa file d'attente de blocs unique.

x.signal() : lorsqu'un processus effectue une opération de signal sur une variable de condition, l'un des processus bloqués a une chance.

- **Avantages de Moniteur :** Les moniteurs ont l'avantage de rendre la programmation parallèle plus facile et moins sujette aux erreurs que l'utilisation de techniques telles que le sémaphore.
- **Inconvénients de Moniteur:** Les moniteurs doivent être implémentés dans le cadre du langage de programmation. Le compilateur doit générer du code pour eux. Cela donne au compilateur la charge supplémentaire de savoir

quelles fonctionnalités du système d'exploitation sont disponibles pour contrôler l'accès aux sections critiques dans les processus simultanés. Certains langages prenant en charge les moniteurs sont Java, C #, Visual Basic, Ada et Euclid simultané.

9. Exercices

1. Décrivez les différences entre la planification à court, moyen et long terme.

   Réponse :

   a. **Court terme (ordonnanceur CPU)** - sélectionne parmi les tâches en mémoire celles qui sont prêtes à être exécutées et leur alloue le CPU.

   b. **Moyen terme** utilisé en particulier avec les systèmes de partage du temps comme niveau d'ordonnancement intermédiaire. Un système de permutation est mis en œuvre pour retirer de la mémoire les programmes partiellement exécutés et les réintégrer ultérieurement pour qu'ils continuent là où ils se sont arrêtés.

   c. **Long terme (job scheduler)** détermine quels travaux sont amenés en mémoire pour être traités. La principale différence réside dans la fréquence de leur exécution. Le court terme doit sélectionner un nouveau processus assez souvent. Le long terme est utilisé beaucoup moins généralement, car il s'occupe de placer les travaux dans le système et peut attendre un certain temps qu'un travail se termine avant d'en admettre un autre.

2. Considérez le mécanisme RPC (Remote Procedure Call).

   - Décrivez les circonstances indésirables qui pourraient résulter de la non-application de la sémantique "au plus une fois" ou "exactement une fois".

- Décrivez les utilisations possibles d'un mécanisme qui n'offrirait aucune de ces garanties.

Réponse :

- Si un mécanisme RPC ne peut pas supporter la sémantique "au plus une fois" ou "au moins une fois", alors le serveur RPC ne peut pas garantir qu'une procédure distante ne sera pas invoquée plusieurs fois.
- Imaginons qu'une procédure distante retire de l'argent d'un compte bancaire sur un système ne supportant pas cette sémantique.
- Il est possible qu'une seule invocation de la procédure distante entraîne de multiples retraits sur le serveur.
- Pour qu'un système supporte l'une ou l'autre de ces sémantiques, il faut généralement que le serveur maintienne une certaine forme d'état du client, comme l'horodatage décrit dans le texte.
- Si un système n'était pas en mesure de prendre en charge l'une ou l'autre de ces sémantiques, il ne pourrait fournir en toute sécurité que des procédures à distance qui ne modifient pas les données ou ne fournissent pas de résultats sensibles au temps.

Si l'on prend l'exemple de notre compte bancaire, nous avons certainement besoin de la sémantique "au plus une fois" ou "au moins une fois" pour effectuer un retrait (ou un dépôt !). Cependant, une demande de renseignements sur le solde du compte ou sur d'autres informations relatives au compte, comme le nom, l'adresse, etc. ne nécessite pas cette sémantique.

3. Donnez deux exemples de programmation dans lesquels le multithreading n'offre pas de meilleures performances qu'une solution monofilaire

   Réponse :

   a. Tout type de programme séquentiel n'est pas un bon candidat pour le multithreading. Un exemple de ceci est un programme qui calcule une déclaration d'impôts individuelle.

   b. Un autre exemple est un programme "shell" tel que le shell C ou le shell Korn. Un tel programme doit surveiller de près son propre espace de travail, comme les fichiers ouverts, les variables d'environnement et le répertoire de travail actuel.

4. Quelle est la signification de l'expression "attente active" ? Quels autres types d'attente existe-t-il dans un système d'exploitation ? Peut-on éviter complètement l'attente active ? Expliquez votre réponse.

   Réponse :

   L'attente active signifie qu'un processus attend qu'une condition soit satisfaite dans une boucle serrée sans abandonner le processeur. Alternativement, un processus peut attendre en abandonnant le processeur, et se bloquer sur une condition et attendre d'être réveillé à un moment approprié dans le futur. L'attente peut être évitée, mais elle entraîne les frais généraux associés à la mise en veille d'un processus et à la nécessité de le réveiller lorsque l'état approprié du programme est atteint.

5. Expliquez pourquoi les spinlocks ne sont pas appropriés pour les systèmes monoprocesseurs, mais sont souvent utilisés dans les systèmes multiprocesseurs.

   Réponse :

   Les spinlocks ne sont pas appropriés pour les systèmes monoprocesseurs, car la condition qui permettrait à un processus de sortir du spinlock ne peut être obtenue qu'en exécutant un processus différent. Si le processus ne cède pas le processeur, les autres processus n'ont pas la possibilité de définir la condition de programme requise pour que le premier processus puisse progresser. Dans un système multiprocesseur, d'autres processus s'exécutent sur d'autres processeurs et modifient ainsi l'état du programme afin de libérer le premier processus du spinlock.

6. Le programme suivant permet-il de résoudre le problème d'accès à la section critique ?

```
Tour=0
while (1)
{ // attente active
        while (tour !=0) ;
        Section_critique() ;
        tour = 1 ;
        section_noncritique() ;
        ...
}
```

```
// Processus P2
while (1)
{ // attente active
        while (tour !=1) ;
        Section_critique() ;
        tour = 0 ;
        Section_noncritique() ;
        ...
}
```

Réponse : On peut vérifier assez facilement que deux processus ne peuvent entrer en section critique en même temps, toutefois le problème n'est pas vraiment résolu car il est possible qu'un des deux processus ait plus souvent besoin d'entrer en section critique que l'autre; l'algorithme lui fera attendre son tour bien que la section critique ne soit pas utilisée. Un processus peut être bloqué par un processus qui n'est pas en section critique.

P1 lit la valeur de tour qui vaut 0 et entre dans sa section critique. Il est suspendu et P2 est exécuté. P2 teste la valeur de tour qui est toujours égale à 0. Il entre donc dans une boucle en attendant que tour prenne la valeur 1. Il est suspendu et P1 est élu de nouveau. P1 quitte sa section critique, met tour à 1 et entame sa section non critique.

Il est suspendu et P2 est exécuté. P2 exécute rapidement sa section critique, tour = 0 et sa section non critique. Il teste tour qui vaut 0. Il attend que tour prenne la valeur 1.

7. Le programme suivant permet-il de résoudre le problème d'accès à la section critique ?

- 1 :

Occupé est un booleen := false

Entrer_sc()

{

While (occupé)

Occupé=TRUE

}

Sortir_sc()

{

Occupé= FALSE

}

Non il y'a problème, si tous les deux arrivent en même temps, ils trouvent que c'est faux ils peuvent entrer en même temps à la section critique, et donc il y'a un problème comme vu au cours.

- 2 :

Int tour=0 ;

```
Bool je_veux_entrer [2]

Entrer_sc () {

        Je_veux_entrer[i] = TRUE

        Tour = i

        Attendre (je_veux_entrer[1 – i] == FALSE et tour == 1 – i )

                }

Sortir_sc () {

        Je_veux_entrer[i]=FALSE

            }
```

Réponse :

Cette méthode marche pour deux processus, mais elle est mauvaise, elle occupe le processus à 100% pendant le quantum de temps du processus.

8. Décrivez comment l'instruction Swap() peut être utilisée pour fournir une exclusion mutuelle qui satisfait à l'exigence d'attente limitée.

Réponse :

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = Swap(&lock, &key);

    waiting[i] = FALSE;

    /* critical section */

    j = (i+1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;
    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;

    /* remainder section */
} while (TRUE);
```

9. Les serveurs peuvent être conçus pour limiter le nombre de connexions ouvertes. Par exemple, un serveur peut souhaiter n'avoir que *N* connexions de socket à un moment donné. Dès que *N* connexions sont établies, le serveur n'accepte pas d'autres connexions entrantes jusqu'à ce qu'une connexion existante soit libérée. Expliquez comment les sémaphores peuvent être utilisés par un serveur pour limiter le nombre de connexions simultanées.

   Réponse :

   Un sémaphore est initialisé au nombre de connexions socket ouvertes autorisées. Lorsqu'une connexion est acceptée, la méthode **P** (acquire()) est appelée, lorsqu'une connexion est libérée, la méthode **V**(release()) est appelée. Si le système atteint le nombre de connexions socket autorisées, les appels ultérieurs à acquire() se bloqueront jusqu'à ce qu'une connexion existante soit terminée et que la méthode release soit invoquée.

10. Montrez comment implémenter les opérations de sémaphore wait() et signal() dans des environnements multiprocesseurs en utilisant l'instruction TestAndSet(). La solution doit présenter un minimum d'attente occupée.

Réponse :

Voici le pseudo-code pour implémenter les opérations :

```
int guard = 0;
int semaphore_value = 0;

wait()
{
    while (TestAndSet(&guard) == 1);
    if (semaphore_value == 0) {
       atomically add process to a queue of processes
       waiting for the semaphore and set guard to 0;
    } else {
       semaphore_value--;
       guard = 0;
    }
}


signal()
{
    while (TestAndSet(&guard) == 1);
    if (semaphore_value == 0 &&
            there is a process on the wait queue)
        wake up the first process in the queue
        of waiting processes
    else
       semaphore_value++;
    guard = 0;
}
```

11. Démontrez que les moniteurs et les sémaphores sont équivalents dans la mesure où ils peuvent être utilisés pour implémenter les mêmes types de problèmes de synchronisation.

    Réponse :

    Un sémaphore peut être implémenté en utilisant le code moniteur suivant :

```
monitor semaphore {
    int value = 0;
    condition c;

    semaphore_increment() {
       value++;
       c.signal();
    }

    semaphore_decrement() {
       while (value == 0)
       c.wait();
       value--;
    }
}
```

12. Écrivez un moniteur qui implémente un réveil permettant à un programme appelant de se retarder pendant un nombre spécifié d'unités de temps (ticks). Vous pouvez supposer l'existence d'une horloge matérielle réelle qui invoque une procédure tick dans votre moniteur à intervalles réguliers.

    Réponse :

```
monitor alarm {
    condition c;

    void delay(int ticks) {
       int begin_time = read_clock();
       while (read_clock() < begin_time + ticks)
       c.wait();
    }

    void tick() {
       c.broadcast();
    }
}
```

13. Soient trois processus concurrents P1, P2 et P3 qui partagent les variables *n* et out.

    Pour contrôler les accès aux variables partagées, un programmeur propose les codes suivants :

    Semaphore mutex1 = 1 ;

    Semaphore mutex2 = 1 ;

    Code du processus p1 :

    P(mutex1) ;

    P(mutex2) ;

    out=out+1 ;

    n=n-1 ;

V(mutex2) ;

V(mutex1) ;

Code du processus p2 :

P(mutex2) ;

out=out-1 ;

V(mutex2) ;

Code du processus p3 :

P(mutex1) ;

n=n+1 ;

V(mutex1) ;

Cette proposition est-elle correcte ? Sinon, indiquer parmi les 4 conditions requises pour réaliser une exclusion mutuelle correcte, celles qui ne sont pas satisfaites ? Proposer une solution correcte.

Réponse :

Non, car si P2 est en section critique et P1 a exécuté P(mutex1) alors P1 est bloqué et empêche P3 d'entrer en section critique.

Conditions non vérifiées : Un processus en dehors de sa section critique bloque un autre processus.

Processus P1 P(mutex1) ;

n=n-1 ;

V(mutex1) ;

P(mutex2) ;

out = out +1 ;

V(mutex2) ;

14. On veut effectuer en parallèle le produit de deux matrices A et B d'ordre n (nxn). Pour ce faire, on crée m (m<n) processus légers (threads). Chaque processus léger se charge de calculer quelques lignes de la matrice résultat R :

Pour j = 0 à n-1 R[i,j] = ¦ k=0,n-1 A[i,k]*B[k,j] ;

Donner, sous forme de commentaires (en utilisant les sémaphores et les opérations P et V), le code des processus légers : CalculLignes ( ). Préciser les sémaphores utilisés et les variables partagées.

Réponse :

On va utiliser un vecteur T de n booléens. T[i] est 0 si le calcul de la ligne i n'est pas encore entamée. T[i] est égal à 1 sinon. Ici, on n'a pas besoin de sémaphores pour les accès aux matrices (en lecture uniquement). Par contre, on a besoin d'un sémaphore binaire mutex pour contrôler les accès au vecteur T. Initialement, tous les éléments de T sont nuls.

fonction CalculLignes ( )

{

pour i = 0 à n-1 pas 1 faire

P(mutex) si (T[i]==0)

```
        {
        T[i] = 1 ;
        V(mutex) ;
        Pour j = 1 à n pas 1
          faire
                Pour k=1 à n pas 1
              faire R[i,j] += A[i,k] * B[k,j]
          fait
        fait
        } else V(mutex) ;
fait
}
```

Chapitre 3 : La communication interprocessus

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

1. Introduction

Les processus ont souvent besoin de communiquer avec d'autres processus. Par exemple, dans un pipeline shell, la sortie du premier processus doit être transmise au deuxième processus, et ainsi de suite. Il est donc nécessaire que les processus communiquent entre eux, de préférence d'une manière bien structurée et sans utiliser d'interruptions.

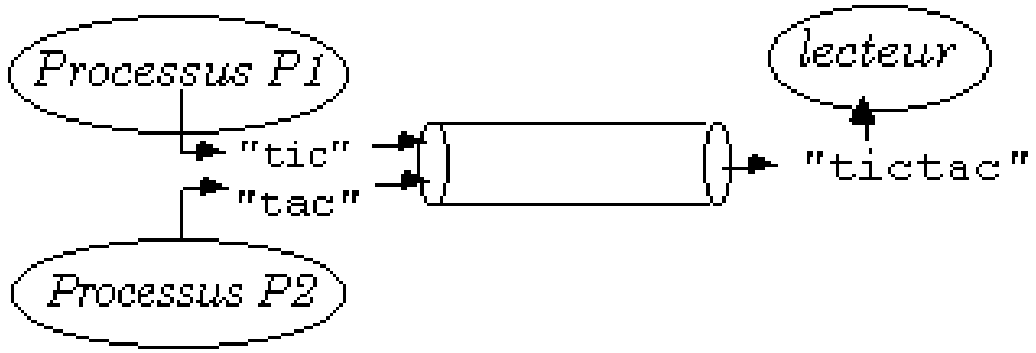


Figure 9 Communication des processus

Cette communication entre processus sous le contrôle du système d'exploitation est appelée *communication interprocessus ou simplement IPC (Inter processus communication).*

Dans certains systèmes d'exploitation, les processus qui travaillent ensemble partagent fréquemment une zone de stockage commune que chacun peut lire et écrire.

Pour voir comment l'IPC fonctionne en pratique, considérons un exemple simple, mais courant, un *spooler d'impression*. Lorsqu'un processus veut imprimer un fichier, il entre le nom du fichier dans un *répertoire* spécial du *spooler*. Un autre processus, le *démon d'impression*, vérifie périodiquement s'il y a des fichiers à imprimer, et s'il y en a, il les envoie à l'imprimante et supprime leurs noms du répertoire.

Des processus actifs simultanément sont dits exécutés en parallèle. Ils peuvent opérer de façon tout à fait indépendante, mais une interaction entre processus parallèles est souvent nécessaire. L'origine de cette nécessité d'interaction peut être :

- La coopération à un but commun, par exemple
    - Le calcul parallèle sur une machine à processeurs multiples,
    - La coopération entre processus se trouvant sur des machines différentes dans le contexte d'applications distribuées,
    - L'organisation d'un programme en tâches parallèles, notamment dans le cas d'un système de contrôle ;
- Le partage de ressources, par exemple un système de fichiers sur disque.

2. Communication inter-processus (ipc)

Un processus peut être de deux types :

- Processus indépendant
- Processus de coopération

Un processus indépendant n'est pas affecté par l'exécution d'autres processus, tandis qu'un processus coopérant peut être affecté par d'autres processus en cours d'exécution. Bien que l'on puisse penser que ces processus, qui s'exécutent indépendamment, s'exécuteront très efficacement, en réalité, il existe de nombreuses situations où la nature coopérative peut être utilisée pour augmenter la vitesse de calcul, la commodité et la modularité.

La communication inter-processus (IPC) est un mécanisme qui permet aux processus de communiquer entre eux et de synchroniser leurs actions. La communication entre ces processus peut être vue comme une méthode de coopération entre eux. Les processus peuvent communiquer entre eux via :

1. La mémoire partagée
2. La transmission de messages

La figure ci-dessous montre une structure de base de communication entre processus via *la méthode de la mémoire partagée* et via *la méthode de transmission de messages*. Un système d'exploitation peut implémenter les deux méthodes de communication.

Tout d'abord, nous discuterons des méthodes de communication de la mémoire partagée, puis de la transmission des messages. La communication entre processus utilisant la mémoire partagée nécessite que les processus partagent une variable, et cela dépend entièrement de la façon dont le programmeur l'implémentera. Une manière de communiquer utilisant la mémoire partagée peut être imaginée comme ceci : supposons que *process1* et *process2* s'exécutent simultanément, et qu'ils partagent certaines ressources ou utilisent des informations d'un autre processus.

Process1 génère des informations sur certains calculs ou ressources utilisés et les conserve sous forme d'enregistrement dans la mémoire partagée. Lorsque *process2* a besoin d'utiliser les informations partagées, il vérifiera l'enregistrement stocké dans la mémoire partagée et prendra note des informations générées par *process1* et agira en conséquence.

Dans ce qui suit, nous examinons un exemple de communication entre processus utilisant la méthode de la mémoire partagée.

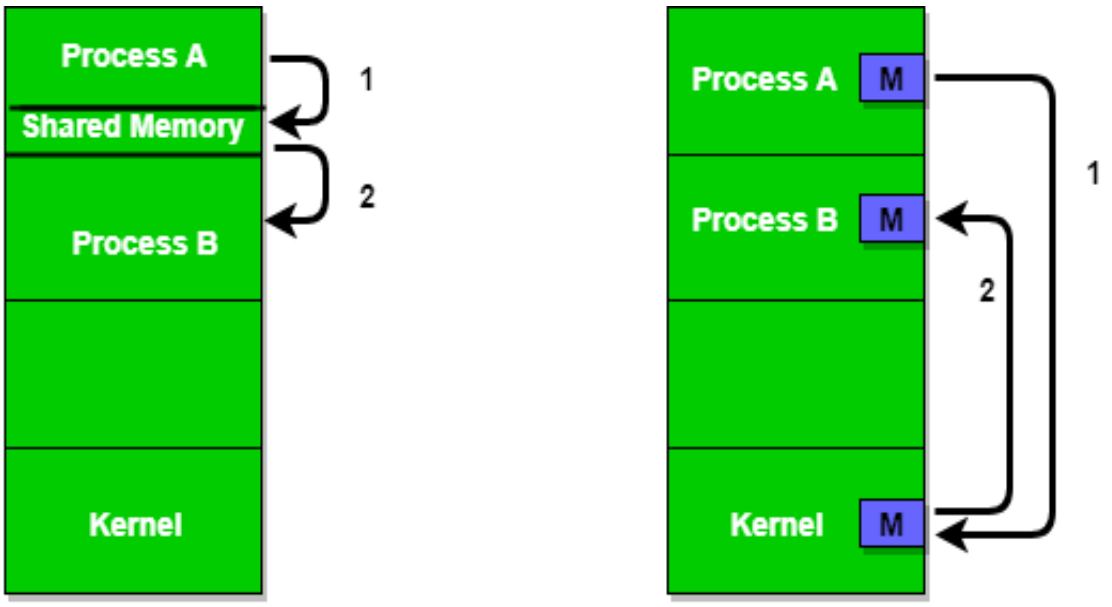


Figure 10 Mémoire partagée et passage de messages

2.1. Méthode de mémoire partagée

La communication interprocessus par le biais de la mémoire partagée est un concept dans lequel deux ou plusieurs processus peuvent accéder à la mémoire commune. Et la communication se fait via cette mémoire partagée, où les modifications apportées par un processus peuvent être vues par un autre processus.

Le problème avec les pipes, fifo et les files d'attente de messages - est que pour que deux processus échangent des informations. L'information doit passer par le noyau.

Le serveur lit le fichier d'entrée.

Le serveur écrit ces données dans un message en utilisant soit un pipe, un fifo ou une file de messages.

Le client lit les données depuis le canal IPC, ce qui nécessite de copier les données depuis le tampon IPC du noyau vers le tampon du client.

Enfin, les données sont copiées à partir du tampon du client.

Au total, quatre copies de données sont nécessaires (2 en lecture et 2 en écriture). La mémoire partagée fournit donc un moyen de permettre à deux ou plusieurs processus de partager un segment de mémoire. Avec la mémoire partagée, les données ne sont copiées que deux fois - du fichier d'entrée vers la mémoire partagée et de la mémoire partagée vers le fichier de sortie.

ftok() *: est utilisé pour générer une clé unique.*

shmget() *: int shmget(key_t,size_tsize,intshmflg) ; en cas de réussite, shmget() retourne un identifiant pour le segment de mémoire partagée.*

shmat() *: Avant de pouvoir utiliser un segment de mémoire partagée, vous devez vous attacher*
*à l'aide de shmat(). void *shmat(int shmid ,void *shmaddr ,int shmflg) ;*
shmid est l'identifiant de la mémoire partagée. shmaddr spécifie l'adresse spécifique à utiliser mais nous devrions la mettre à zéro et le système d'exploitation choisira automatiquement le segment de mémoire partagée.
mais nous devrions le mettre à zéro et le système d'exploitation choisira

automatiquement l'adresse.

***shmdt()** : Lorsque vous en avez fini avec le segment de mémoire partagée, votre programme devrait*

*se détacher de celui-ci en utilisant shmdt(). int shmdt(void *shmaddr) ;*

***shmctl()** : quand vous vous détachez de la mémoire partagée, elle n'est pas détruite. Donc, pour détruire*

shmctl() est utilisé. shmctl(int shmid,IPC_RMID,NULL) ;

Ex : Problème Producteur-Consommateur

Il existe deux processus : Producteur et Consommateur. Le producteur produit certains articles et le consommateur consomme cet article. Les deux processus partagent un espace commun ou un emplacement mémoire appelé tampon où l'article produit par le Producteur est stocké et à partir duquel le Consommateur consomme l'article si nécessaire.

Il existe deux versions de ce problème :

- La première est connue sous le nom de problème de tampon illimité dans lequel le producteur peut continuer à produire des éléments et il n'y a pas de limite à la taille du tampon,
- La seconde est connue sous le nom de problème de tampon borné dans que le Producteur peut produire jusqu'à un certain nombre d'articles avant de commencer à attendre que le Consommateur le consomme.

Nous aborderons le problème du tampon borné. Premièrement, le producteur et le consommateur partageront une mémoire commune, puis le producteur commencera à produire des articles. Si l'article produit total est égal à la taille du tampon, le producteur attendra de le faire consommer par le consommateur. De même, le consommateur vérifiera au préalable la disponibilité de l'article. Si aucun article n'est disponible, le Consommateur attendra que le Producteur le produise. S'il y a des articles disponibles, le consommateur les consommera. Le pseudo-code à démontrer est fourni ci-dessous :

- **Données partagées entre les deux processus en C**

Code de processus du producteur

```
#define buff_max 25
#define mod %
```

```
struct item{
// membre différent des données produites ou des données consommées
}
 // Un tableau est nécessaire pour contenir les éléments. C'est le
    lieu partagé auquel les deux processus auront accès.
// item shared_buff [ buff_max ];

// Deux variables qui garderont la trace des indices des articles
   produits par le producteur et le consommateur, L'indice libre
   pointe vers le prochain indice libre. L'indice complet pointe
   vers le premier indice complet
int free_index = 0;
int full_index = 0;
```

Code de processus du producteur en C

```
item nextProduced;
  while(1){
     // vérifier s'il n'y a pas d'espace pour la production.
     // Si oui, continuez à attendre.
     while((free_index+1) mod buff_max == full_index);
       shared_buff[free_index] = nextProduced;
       free_index = (free_index + 1) mod buff_max;
  }
```

Code de processus consommateur en C

```
item nextConsumed;

  while(1){

    // vérifier s'il y a un article disponible pour la consommation.

    // Si ce n'est pas le cas, continuez à attendre.

        while((free_index == full_index);

            nextConsumed = shared_buff[full_index];

            full_index = (full_index + 1) mod buff_max;
  }
```

Dans le code ci-dessus, le Producteur recommencera à produire lorsque le mod buff max (free_index+1) sera gratuit, car s'il n'est pas gratuit, cela implique qu'il y a encore des objets qui peuvent être consommés par le Consommateur, donc il n'y a pas besoin produire plus. De même, si l'index libre et l'index complet pointent vers le même index, cela implique qu'il n'y a pas d'éléments à consommer.

2.2. Méthode de transmission de messagerie

Maintenant, nous allons commencer notre discussion sur la communication entre les processus via la transmission de messages. Dans cette méthode, les processus

communiquent entre eux sans utiliser aucun type de mémoire partagée. Si deux processus p_1 et p_2 veulent communiquer entre eux, ils procèdent comme suit :

- Établir un lien de communication (si un lien existe déjà, pas besoin de l'établir à nouveau.)
- Commencez à échanger des messages en utilisant des primitives de base. Nous avons besoin d'au moins deux primitives :
    - **envoyer** (message, destination) ou **envoyer** (message)
    - **recevoir** (message, hôte) ou **recevoir** (message)

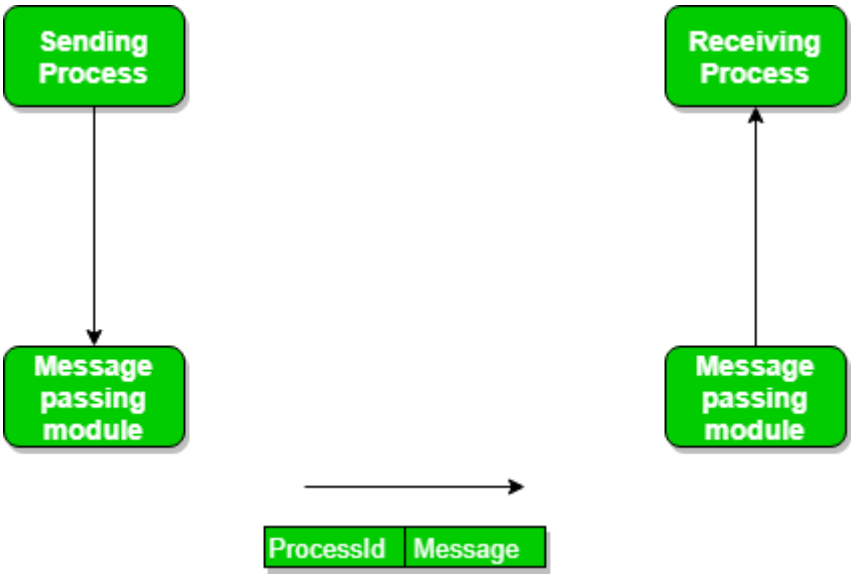


Figure 11 Méthode de transmission de messagerie

La figure 11, ci-dessous, illustre le mécanisme de communication par échange de messages.

- La taille du message peut être de taille fixe ou de taille variable. S'il est de taille fixe, c'est facile pour un concepteur d'OS, mais compliqué pour un programmeur, et s'il est de taille variable, alors c'est facile pour un programmeur, mais compliqué pour le concepteur d'OS. Un message standard peut avoir deux parties : un **en-tête et un corps.**
- La **partie en-tête** est utilisée pour stocker le type de message, l'identifiant de destination, l'identifiant de source, la longueur du message et les informations de contrôle. Les informations de contrôle contiennent des informations telles que ce qu'il faut faire en cas de manque d'espace tampon, le numéro de séquence, la priorité. Généralement, le message est envoyé en utilisant le style FIFO.

2.3. Message passant par le lien de communication

Lien de communication direct et indirect

Maintenant, nous allons commencer notre discussion sur les méthodes de mise en œuvre des liens de communication. Lors de la mise en œuvre du lien, certaines questions doivent être prises en compte, telles que :

- Comment les liens sont-ils établis ?
- Un lien peut-il être associé à plus de deux processus ?
- Combien de liens peut-il y avoir entre chaque paire de processus communicants ?
- Qu'elle est la capacité d'un lien ? La taille d'un message que le lien peut accepter est-elle fixe ou variable ?

- Un lien est-il unidirectionnel ou bidirectionnel ?

Un lien a une certaine potentialité qui détermine le nombre de messages qui peuvent y résider temporairement, pour lesquels chaque lien est associé à une file d'attente qui peut être de possibilité nulle, de possibilité limitée ou de possibilité illimitée. Au maximum de sa possibilité nulle, l'expéditeur attend que le destinataire informe l'expéditeur qu'il a reçu le message. Dans les cas de disposition non nulle, un processus ne sait pas si un message a été reçu ou non après l'opération d'envoi. Pour cela, l'expéditeur doit communiquer explicitement avec le destinataire. La mise en œuvre du lien dépend de la situation, il peut s'agir soit d'un lien de communication direct, soit d'un lien de communication indirect.

- **Liens de communication directe** sont mis en œuvre lorsque les processus utilisent un identifiant de processus spécifique pour la communication, mais il est difficile d'identifier l'expéditeur à l'avance.

  Exemple : le serveur d'impression.
- **La communication indirecte** se fait via une boîte aux lettres partagée (port), qui consiste en une file d'attente de messages. L'expéditeur garde le message dans la boîte aux lettres et le destinataire le récupère.

2.4. Message passant par l'échange des messages.

Transmission de messages synchrone et asynchrone :

Un processus bloqué est un processus qui attend un événement, tel qu'une ressource devenant disponible ou l'achèvement d'une opération d'E/S.

L'IPC est possible entre les processus sur le même ordinateur ainsi que sur les processus exécutés sur différents ordinateurs, c'est-à-dire dans un système en réseau/distribué.

Dans les deux cas, le processus peut ou non être bloqué lors de l'envoi d'un message ou de la tentative de réception d'un message, de sorte que la transmission du message peut être bloquante ou non bloquante. Le blocage est considéré comme synchrone et le blocage de l'envoi signifie que l'expéditeur sera bloqué jusqu'à ce que le message soit reçu par le destinataire. De même, le blocage de la réception bloque le récepteur jusqu'à ce qu'un message soit disponible. Non bloquant est considéré l'envoi asynchrone et non bloquant permet à l'expéditeur d'envoyer le message et de continuer.

Aussi, la réception non bloquante permet au destinataire de recevoir un message valide ou nul. Après une analyse minutieuse, nous pouvons conclure que pour un expéditeur, il est plus naturel d'être non bloquant après le passage du message, car il peut être nécessaire d'envoyer le message à différents processus. Cependant, l'expéditeur attend un accusé de réception de la part du destinataire en cas d'échec de l'envoi. De même, il est plus naturel pour un récepteur de bloquer après avoir émis la réception parce que les informations du message reçu peuvent être utilisées pour une exécution ultérieure. Dans le même temps, si l'envoi du message continu

d'échouer, le destinataire devra attendre indéfiniment. C'est pourquoi nous considérons également l'autre possibilité de transmission de messages.

Il existe essentiellement trois combinaisons préférées :

- Blocage de l'envoi et blocage de la réception
- Envoi non bloquant et réception non bloquante
- Envoi non bloquant et réception bloquante (principalement utilisé)

Dans le passage de message direct, le processus qui veut communiquer doit nommer explicitement le destinataire ou l'expéditeur de la communication.

Par exemple, envoyer (p_1, message) signifie envoyer le message à p_1.

De même, recevoir (p_2, message) signifie recevoir le message de p_2.

Dans cette méthode de communication, la liaison de communication est établie automatiquement, qui peut être unidirectionnelle ou bidirectionnelle, mais une liaison peut être utilisée entre une paire d'expéditeur et de destinataire, et une paire d'expéditeur et de destinataire ne doit pas posséder plus d'une paire de liens. La symétrie et l'asymétrie entre l'envoi et la réception peuvent également être mises en œuvre, c'est-à-dire que les deux processus se nommeront pour l'envoi et la réception des messages, ou que seul l'expéditeur nommera le destinataire pour l'envoi du message et il n'est pas nécessaire que le destinataire nomme l'expéditeur pour réception du message.

Le problème avec cette méthode de communication est que si le nom d'un processus change, cette méthode ne fonctionnera pas.

Dans le passage de message indirect, les processus utilisent des boîtes aux lettres (aussi appelées ports) pour envoyer et recevoir des messages. Chaque boîte aux lettres à un identifiant unique et les processus ne peuvent communiquer que s'ils partagent une boîte aux lettres. Le lien établi uniquement si les processus partagent une boîte aux lettres commune et qu'un seul lien peut être associé à plusieurs processus. Chaque couple de processus peut partager plusieurs liens de communication et ces liens peuvent être unidirectionnels ou bidirectionnels.

Supposons que deux processus souhaitent communiquer via la transmission indirecte de messages, les opérations requises sont : créer une boîte aux lettres, utiliser cette boîte aux lettres pour envoyer et recevoir des messages, puis détruire la boîte aux lettres. Les primitives standard utilisées sont : send(A, message) qui signifie envoyer le message à la boîte aux lettres A.

La primitive de réception du message fonctionne également de la même manière, par exemple Receive (A, message).

Il y a un problème avec cette implémentation de boîte aux lettres. Supposons qu'il y ait plus de deux processus partageant la même boîte aux lettres et supposons que le processus p1 envoie un message à la boîte aux lettres, quel processus sera le destinataire ? Cela peut être résolu soit en imposant que seuls deux processus puissent partager une seule boîte aux lettres, soit en imposant qu'un seul processus,

soit autorisé à exécuter la réception à un moment donné ou en sélectionnant n'importe quel processus au hasard et en informant l'expéditeur du destinataire.

Une boîte aux lettres peut être rendue privée à une seule paire expéditeur/destinataire et peut ainsi être partagée entre plusieurs paires expéditeur/destinataire. Le Port est une implémentation d'une telle boîte aux lettres qui peut avoir plusieurs expéditeurs et un seul destinataire. Il est utilisé dans les applications client/serveur (dans ce cas, le serveur est le récepteur). Le port appartient au processus récepteur et est créé par le système d'exploitation à la demande du processus récepteur et peut être détruit à la demande du même processeur récepteur lorsque le récepteur se termine lui-même, se faire en sorte qu'un seul processus soit autorisé à exécuter la réception peut être fait en utilisant le concept d'exclusion mutuelle. La boîte aux lettres *mutex* est créée et partagée par n processus. L'expéditeur est non bloquant et envoie le message. Le premier processus qui exécute la réception entrera dans la section critique et tous les autres processus seront bloquants et attendront.

Maintenant, discutons du problème Producteur-Consommateur en utilisant le concept de transmission de message.

Le producteur place des articles (à l'intérieur des messages) dans la boîte aux lettres et le consommateur peut consommer un article lorsqu'au moins un message est présent dans la boîte aux lettres. Le code est donné ci-dessous :

Code Producteur en C

```
void Producer(void){

int item;

Message m;

 while(1){

   receive(Consumer, &m);

   item = produce();

   build_message(&m , item ) ;

   send(Consumer, &m);

      }

   }
```

Code de la consommation en C

```
void Consumer(void){

      int item;

      Message m;

      while(1){

          receive(Producer, &m);

         item = extracted_item();

         send(Producer, &m);

         consume_item(item);

      }

   }
```

2.5. Exemples de systèmes IPC

- Posix : utilise la méthode de la mémoire partagée.
- Mach : utilise le passage de messages
- Windows XP : utilise la transmission de messages à l'aide d'appels procéduraux locaux

2.6. Communication en architecture client/serveur :

La communication client/serveur implique deux composants, à savoir un client et un serveur. Il s'agit généralement de plusieurs clients en communication avec un seul serveur. Les clients envoient des demandes au serveur et le serveur répond aux demandes des clients.

Il existe trois méthodes principales de communication client/serveur. Elles sont présentées comme suit .

Il existe différents mécanismes :

- **Pipe**

Il s'agit de méthodes de communication interprocessus qui contiennent deux points d'extrémité. Les données sont introduites à une extrémité du tuyau par un processus et consommées à l'autre extrémité par l'autre processus.

Les deux différents types de tuyaux sont les tuyaux ordinaires et les tuyaux nommés. Les tuyaux ordinaires ne permettent qu'une communication unidirectionnelle. Pour une communication bidirectionnelle, deux tuyaux sont nécessaires. Les pipes

ordinaires ont une relation parent-enfant entre les processus, car les pipes ne peuvent être accédées que par les processus qui les ont créés ou hérités.

Les tubes nommés sont plus puissants que les tubes ordinaires et permettent une communication bidirectionnelle. Ces pipes existent même après la fin des processus qui les utilisent. Ils doivent être explicitement supprimés lorsqu'ils ne sont plus nécessaires.

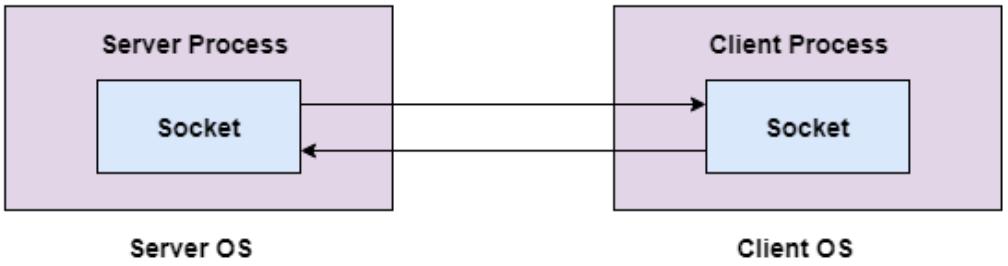
Un diagramme illustrant les pipes est donné ci-dessous.

- **Socket**

Les sockets facilitent la communication entre deux processus sur la même machine ou sur des machines différentes. Elles sont utilisées dans un cadre client/serveur et se composent de l'adresse IP et du numéro de port. De nombreux protocoles d'application utilisent les sockets pour la connexion et le transfert de données entre un client et un serveur.

La communication par sockets est de très bas niveau, car les sockets ne font que transférer un flux d'octets non structuré entre les processus. La structure du flux d'octets est imposée par les applications client et serveur.

Voici un diagramme qui illustre les sockets

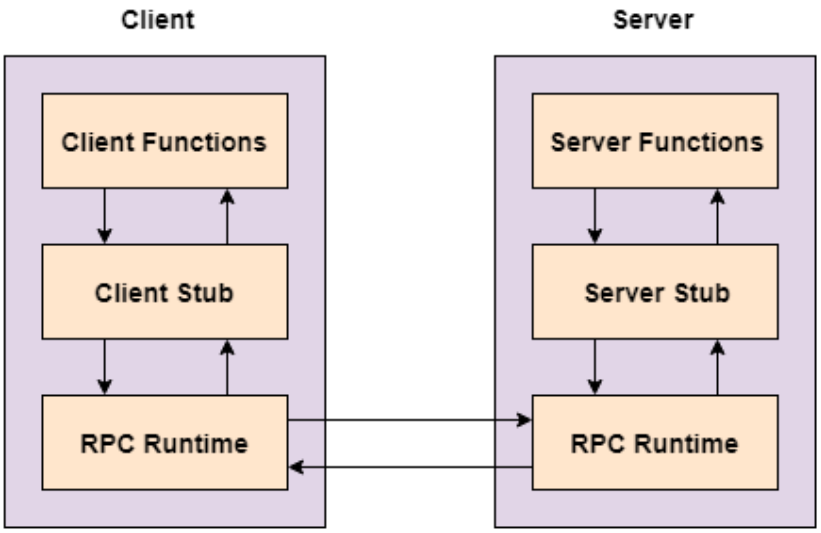


- **Appels procéduraux à distance (RPC)**

Il s'agit de techniques de communication interprocessus qui sont utilisées pour les applications client-serveur. Un appel de procédure à distance est également connu comme un appel de sous-routine ou un appel de fonction.

Un client a une demande que le RPC traduit et envoie au serveur. Cette demande peut être une procédure ou un appel de fonction vers un serveur distant. Lorsque le serveur reçoit la demande, il renvoie la réponse requise au client.

Voici un diagramme qui illustre les appels de procédure à distance



Les trois méthodes ci-dessus seront discutées dans des articles ultérieurs, car elles sont toutes assez conceptuelles et méritent leurs propres articles séparés.

3. Exercices :

1. Expliquez brièvement le principe, et donnez les avantages et les inconvénients des méthodes de communications suivantes : mémoire partagée, file, pipe et socket.

Réponse

Méthode	Principe	Avantages	Inconvénients
Mémoire partagée	La mémoire partagée permet à deux processus ou plus d'accéder à la même zone mémoire comme s'ils avaient leurs pointeurs dirigés vers le même espace mémoire. Lorsqu'un processus modifie la mémoire, tous les autres processus voient la modification	La mémoire partagée est la forme de communication interprocessus la plus rapide car tous les processus partagent la même mémoire. Elle évite également les copies de données inutiles	Le principe de l'exclusion mutuelle, nécessaire en cas d'accès concurrent, n'est pas garanti avec ce mode communicatio n.
File	Une file premier entré, premier sorti (first-in, first-out, FIFO) est un tube qui dispose d'un nom dans le système de fichiers. Tout processus peut ouvrir ou fermer la file FIFO.	Les processus communicants peuvent ne pas avoir de lien de parenté.	Le principe de l'exclusion mutuelle, nécessaire en cas d'accès concurrent, n'est pas

			garanti avec ce mode communication.
Pipe	Un tube est un dispositif de communication qui permet une communication à sens unique. Les données écrites sur l'« extrémité d'écriture » du tube sont lues depuis l'« extrémité de lecture ». Les tubes sont des dispositifs séquentiels; les données sont toujours lues dans l'ordre où elles ont été écrites. Typiquement, un tube est utilisé pour la communication entre deux threads d'un même processus ou entre processus père et fils	La synchronisation des processus en entrée et en sortie du tube est assurée par le dispositif lui-même.	Exigent que les processus soient liés par un lien de parenté (père-fils). Sens unidirectionnel.
Socket	Un socket est un dispositif de communication bidirectionnelle	Possibilité de choisir le protocole, type de	Plus difficile de gérer plusieurs

	pouvant être utilisé pour communiquer avec un autre processus sur la même machine ou avec un processus s'exécutant sur d'autres machines	connexion et style de datagramme	processus communicants en même temps

2. Écrire un programme C permettant à un processus père de récupérer le code renvoyé par un processus fils dans la fonction exit.

Réponse :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include<sys/wait.h>//pour wait
#include<errno.h> /* permet de récupérer les codes d'erreur */
pid_t pid_fils
int main(void)
{
int status;
switch (pid_fils=fork())
 {
case -1 : perror("Problème dans fork()\n");
```

```
exit(errno); /* retour du code d'erreur */
break;
case 0 : puts("Je suis le fils");
puts("Je retourne le code 3");
exit(3);
default : puts("Je suis le père");
puts("Je récupère le code de retour");
wait(&status);
printf("code de sortie du fils %d : %d\n",
pid_fils, WEXITSTATUS(status));
break;
}
return 0;
}
```

3. Soit le code suivant :

```c
#include <signal.h>
#include <stdio.h>

int var;

void hand(int n)
{
        printf("Signal reçu : %d\n",n);
        if (n==SIGINT) signal(SIGINT,SIG_DFL);
        else signal (SIGQUIT, SIG_IGN);
}

int main ()
{
        signal(SIGINT,hand);
        signal(SIGQUIT,hand);
        while(1);
}
```

Expliquez le comportement du programme lorsque vous appuyez sur Ctrl+C.

4. Soit le code suivant :

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

int main(int argc, char *argv[])
{
  int filedes[2];
  int PID;
  pipe(filedes);
  if ( (PID=fork()) > 0)
  {
        write(filedes[1],"Message du pere a son fils...q",strlen("Message du pere a son fils...q"));
        printf("Le père a écrit\n");
        while(getchar());
  }

  else if (PID==0)
  {
        char car=0;
        char tab[200];
        int i=0;
        printf("Le fils essaie de lire le message\n");
        do
        {
                if ( read(filedes[0],&car,1)==1 )
                {
                        tab[i]=car;i++;
                        printf("Caractere lu : %c\n",car);
                }
        }while(car!='q');
        tab[i]=0;printf("Message : %s\n",tab);
  }

  else
  {
        printf("Erreur lors du fork()\n");
  }

  return EXIT_SUCCESS;
}
```

- Que réalise l'appel système **pipe**() sur la variable **filedes** ?

- À quoi sert le caractère 'q' dans le message envoyé par le père ?
- Modifiez votre programme pour que le père puisse répondre au fils.

4. Proposer un algorithme d'un serveur de calcul inspiré des calculatrices (on se limitera au support des 4 opérations de base, +, -, ×, /). On rappelle que ce type de calculatrice utilise une pile pour effectuer les calculs.

Le client se connecte au serveur puis envoie des commandes.

Une commande est : — soit un nombre ; — soit un opérateur.

Le protocole (pour chaque envoi de commande) est le suivant :

1. le client envoie la commande terminée par un caractère de retour à la ligne ;

2. le serveur traite la commande : — nombre : le nombre est empilé, — opérateur : les opérandes de l'opérateur sont dépilés, puis le résultat de l'application de l'opérateur à ces opérandes est empilé.

3. le contenu du niveau 1 de la pile est retourné au client.

À la fermeture du client, le serveur repasse en mode acceptation de client.

On suppose pour l'implémentation que les opérations de base sur les piles sont disponibles dans une bibliothèque.

Réponse :

- Le client envoie son chiffre ou sa commande et reçoit un message de réponse de la part du serveur tant qu'il ne souhaite pas s'arrêter (envoi de la commande f).
- Le serveur fait une boucle infinie autour de l'acceptation d'une connexion et du traitement des messages qu'il reçoit.
- Ainsi, lorsqu'un client se déconnecte, c'est le suivant qui sera servi par le serveur.
- On supposera que le client prend en argument sur la ligne de commande l'adresse IP et le port du serveur.
- Le serveur prend simplement en argument le port sur lequel il doit être attaché.
- Des codes possibles sont les suivants, on présente tout d'abord le code d'un client puis celui du serveur.
- Les tests d'échec des primitives ne sont pas faits, ils sont à rajouter dans tout travail sérieux.

Partie Client

```c
int main(int argc, char *argv[]) {
  int  sclient;                       /* Descripteur de la socket client  */
  char commande[BUFFER_SIZE] = {0},   /* Commande a envoyer */
       message[BUFFER_SIZE] = {0};    /* Message reçu */
  struct sockaddr_in saddr = {0};     /* Adresse du serveur */

  /* 1. On crée la socket du client. */
  sclient = socket(AF_INET,SOCK_STREAM,0);

  /* 4.1. On prépare l'adresse du serveur. */
  saddr.sin_family = AF_INET;
  saddr.sin_port = htons(atoi(argv[2]));
  saddr.sin_addr.s_addr = inet_addr(argv[1]);

  /* 4.2. On demande une connexion au serveur, tant qu'on y arrive pas. */
  while (connect(sclient,(struct sockaddr *)&saddr,sizeof(saddr)) == -1) ;

  /* 6. On communique. */
  while(commande[0] != 'f') {
    printf("Entrez un nombre, un opérateur (+ ou -) ou f pour sortir.\n");
    scanf(" %1023s",commande);                  /* Lecture commande clavier */
    write(sclient,commande,strlen(commande)+1); /* Envoi de la commande */
    read(sclient,message,BUFFER_SIZE-1);        /* Réception du message */
    printf("Message reçu : %s\n",message);      /* Affichage */
  }
  /* 7 et 8. On termine et libère les ressources. */
  shutdown(sclient,SHUT_RDWR);
  close(sclient);
  return 0;
}
```

Partie Serveur

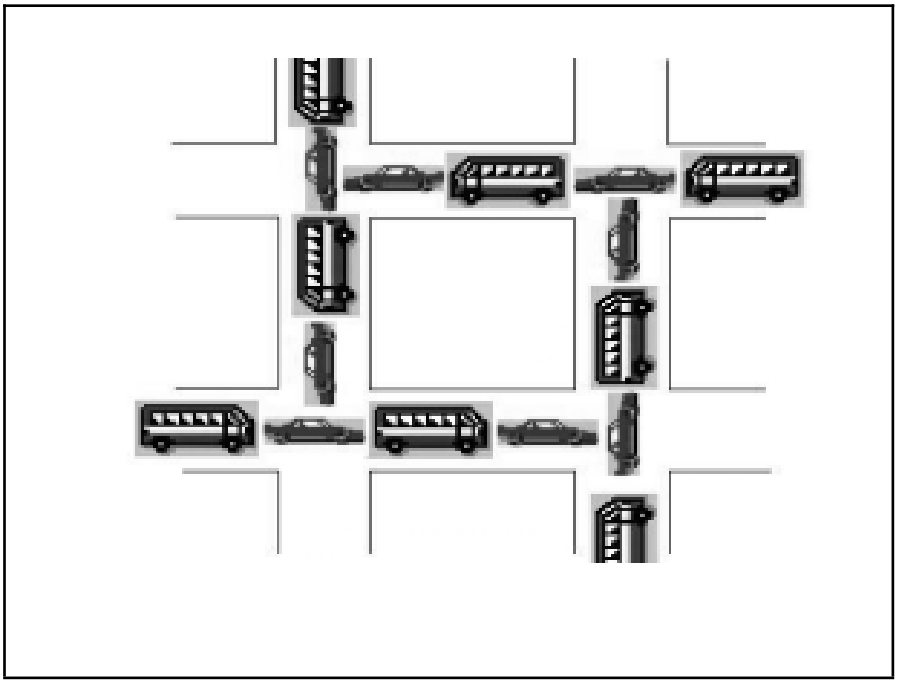
```
int main(int argc, char *argv[]) {
  int  secoute,                    /* Descripteur socket d'écoute */
       sservice;                   /* Descripteur socket de service */
  socklen_t caddrlen;              /* Longueur de l'adresse du client  */
  float resultat;                  /* Résultat pour les calculs */
  char message[BUFFER_SIZE] = {0}; /* Message reçu ou a envoyer */
  struct sockaddr_in saddr = {0},  /* Mon adresse serveur */
                     caddr = {0};  /* Adresse du client */
  pile p;                          /* La pile */

  /* 1. On crée la socket d'écoute. */
  secoute = socket(AF_INET,SOCK_STREAM,0);
  /* 2.1 On prépare l'adresse du serveur. */
  saddr.sin_family = AF_INET;
  saddr.sin_addr.s_addr = htonl(INADDR_ANY);
  saddr.sin_port = htons(atoi(argv[1]));
  /* 2.2. On attache la socket a l'adresse du serveur. */
  bind(secoute,(struct sockaddr *)&saddr,sizeof(saddr));
  /* 3. On attend les demandes de connexion. */
  listen(secoute,5);

  while (1) {
    /* 5. On accepte une connexion. */
    caddrlen = sizeof(caddr);       /* Initialisation de caddrlen */
    sservice = accept(secoute,(struct sockaddr *)&caddr,&caddrlen);
    /* 6. On communique. */
    init_pile(&p);                  /* On initialise la pile */
    while (message[0] != 'f') {
      memset(message,'\0',BUFFER_SIZE);  /* Mise a 0 du message */
      read(sservice,message,BUFFER_SIZE);
      switch(message[0]) {         /* Traitement du message (très basique !) */
        case '+' : resultat = depiler(&p)+depiler(&p); /* Cas opérateur + */
                   empiler(&p,resultat);
                   sprintf(message,"%f",resultat);
                   write(sservice,&message,strlen(message)+1);
                   break;
        case '-' : resultat = depiler(&p)-depiler(&p); /* Cas opérateur - */
                   empiler(&p,resultat);
                   sprintf(message,"%f",resultat);
                   write(sservice,&message,strlen(message)+1);
                   break;
        case 'f' : break;                              /* Cas 'f' : fin */
        default  : empiler(&p,atof(message));          /* Cas nombre */
                   write(sservice,"Empilement OK",14);
      }
    }
    /* 7 et 8. Terminaison et libération des ressources. */
    shutdown(sservice,SHUT_RDWR);
    close(sservice);
  }
  close(secoute);
```

Chapitre 4 : L'inter blocage

- Modèles
- Prévention
- Évitement
- Détection/ Guérison



1. Introduction

Un système d’interblocage est un système utilisé pour maintenir un état de fonctionnement sûr des machines, qui détecte les conditions anormales ou les séquences incorrectes et interrompt toute action ultérieure ou lance une action corrective.

Exemple :

Un système de verrouillage de vanne est un ensemble de verrouillage à clé piégée qui verrouille la vanne dans une ou deux positions (ouverte et/ou fermée) avec une clé piégée dans l'ensemble de verrouillage et une clé libre.

2. Définition de l’interblocage

Un ensemble de processus est en interblocage, si chaque processus attend un événement que seul un autre processus de l’ensemble peut provoquer.

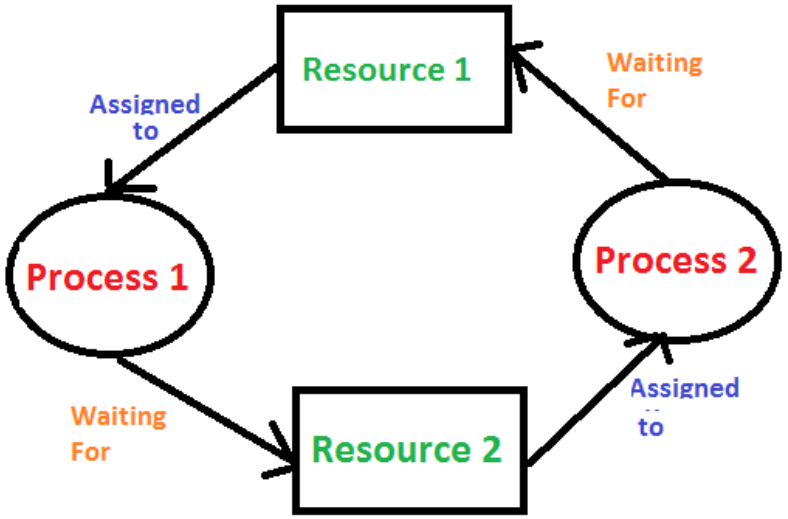


Figure 13 Schéma d'interblockage

Le blocage est une situation où un ensemble de processus est bloqué parce que chaque processus détient une ressource et attend une autre ressource acquise par un autre processus.

Prenons un exemple, lorsque deux trains se rapprochent sur la même voie et qu'il n'y a qu'une seule voie, aucun des trains ne peut se déplacer une fois qu'ils sont l'un devant l'autre.

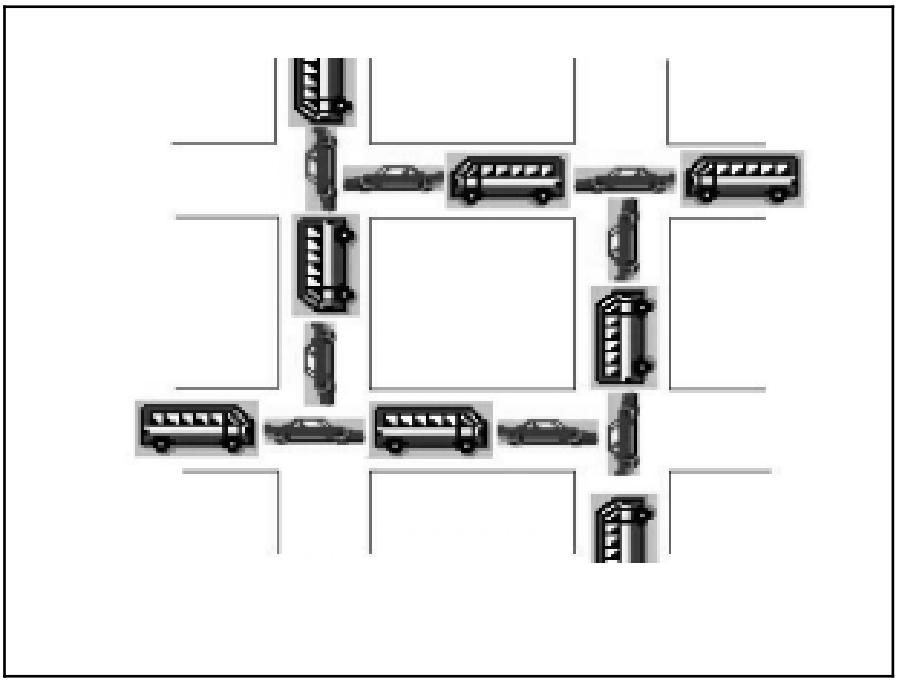


Figure 12 Problème de circulation routière

Une situation similaire se produit dans les systèmes d'exploitation lorsqu'il y a deux processus ou plus qui détiennent certaines ressources et attendent des ressources détenues par d'autres. Par exemple, dans le diagramme ci-dessus, le processus 1 contient la ressource 1 et attend la ressource 2 qui est acquise par le processus 2, et le processus 2 attend la ressource 1

3. Caractérisation de l’interblocage

Une situation d’interblocage peut survenir si les quatre conditions suivantes se produisent simultanément (conditions nécessaires, mais non suffisantes) :

1. **Exclusion mutuelle :** Le système a des ressources non partageables (un seul processus à la fois peut s’en servir). Comme exemple : Processeur, une zone de mémoire, un périphérique, mais aussi un sémaphore, un moniteur, une section critique.

2. **Occupation et attente (hold and wait) :** Un processus détient au moins une ressource non partageable et qui attend d'acquérir des ressources supplémentaires détenus par d'autres processus.
3. **Pas de réquisition (préemption) :** Un processus qui a saisi une ressource non partageable, la garde jusqu'à ce qu'il aura complété sa tâche.
4. **Attente circulaire :** Il y a un cycle de processus tel que chaque processus pour compléter doit utiliser une ressource non partageable qui est utilisée par le suivant, et que le suivant gardera jusqu'à sa terminaison

4. Méthodes de traitement de l'interblocage

Trois méthodes principales sont possibles pour traiter le problème de l'interblocage :

1. Employer un protocole pour assurer que le système ne se trouvera jamais dans une situation d'interblocage (prévention).
2. Permettre que le système se trouve en situation d'interblocage et le corriger ensuite.
3. Ignorer complètement le problème d'interblocage et supposer que le système ne se trouvera jamais dans de telles situations.

Dans ce qui suit, ces trois techniques seront abordées.

5. Prévenir les interblocages

Pour prévenir un interblocage, il faut éviter la réalisation d'au moins une des conditions citées ci-dessous.

- **Exclusion mutuelle**

Réduire le plus possible l'utilisation des ressources partagées et sections critiques. En général, ceci n'est pas possible puisque certaines ressources sont intrinsèquement non partageables.

- **Occupation et attente**

Un processus qui demande de nouvelles ressources ne devrait pas en retenir d'autres (les demander toutes ensemble). Une situation de famine se caractérise par l'attente d'une ressource non satisfaite pendant un certain temps.

- **Pas de réquisition**

Si un processus qui demande d'autres ressources ne peut pas les avoir, il doit être suspendu, ses ressources doivent être rendues disponibles. Cela peut s'appliquer aux ressources dont l'état peut être facilement sauvegardé et restauré (registre UC ou zones mémoires)

- **Attente circulaire**

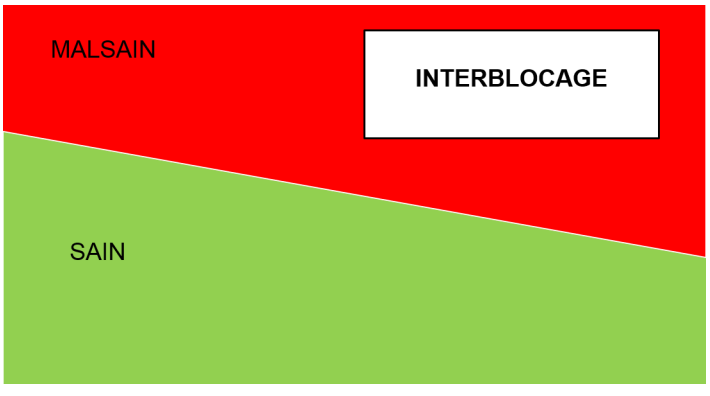
Imposer un ordre total sur les ressources. Un processus doit demander les ressources dans cet ordre (comme exemple : tout processus doit toujours demander une imprimante avant de demander une unité de disque)

L'utilisation d'une autre méthode, à savoir éviter les interblocages en demandant des informations supplémentaires sur la façon dont les ressources vont être requises. Cette méthode est présentée dans la section suivante.

6. Éviter les interblocages

Pour éviter un éventuel futur interblocage, il faut savoir quelles sont les ressources disponibles, les ressources allouées à chaque processus et les futures requêtes de chaque processus. Sur la base de ces informations, un examen dynamique est effectué sur l'état d'allocation des ressources afin d'assurer qu'il ne puisse jamais exister une condition d'attente circulaire. Ainsi, Il faut éviter d'accepter toute requête qui fait passer un système d'un état sain vers un état malsain.

Puisque l'ensemble des états d'interblocage est un sous ensemble des états malsains (voir figure ci-dessous). Un état est dit sain s'il existe à partir de cet état une séquence saine.



Sinon, l'état est dit malsain. Si à partir d'un état donné, l'ensemble des processus du système peuvent terminer normalement dans un ordre donné leur exécution, alors cette succession de terminaisons est dite séquence saine.

Pour traiter le problème d'interblocage d'une façon effective, nous allons définir, dans la section suivante, la notion de graphe d'allocation de ressources.

7. Graphe d'allocation de ressources

○ Définitions du graphe

- G = (V, E) où V : ensemble de sommets et E : ensemble d'arrêtes.
- V est partitionné en deux sous-ensembles :
    - $P = P_1, P_2, \ldots, P_n$, l'ensemble de tous les processus.
    - $R = R_1, R_2, \ldots, R_m$, l'ensemble de tous les types de ressources.
- E contient :
    - Arrête requête : arête dirigée de P_i vers R_k (P_i a besoin d'un exemplaire de R_k)
    - Arrête affectation : arête dirigée de R_i vers P_k (P_i détient un exemplaire de R_k)
    - Arrête de réclamation : arête dirigée de P_i vers R_k (Pi peut demander dans le futur un exemplaire de R_k)

Voir ce schéma explicite.

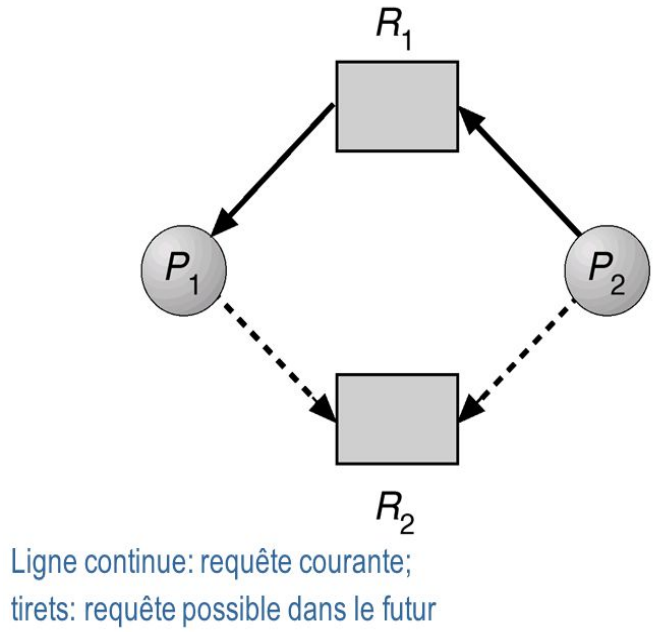
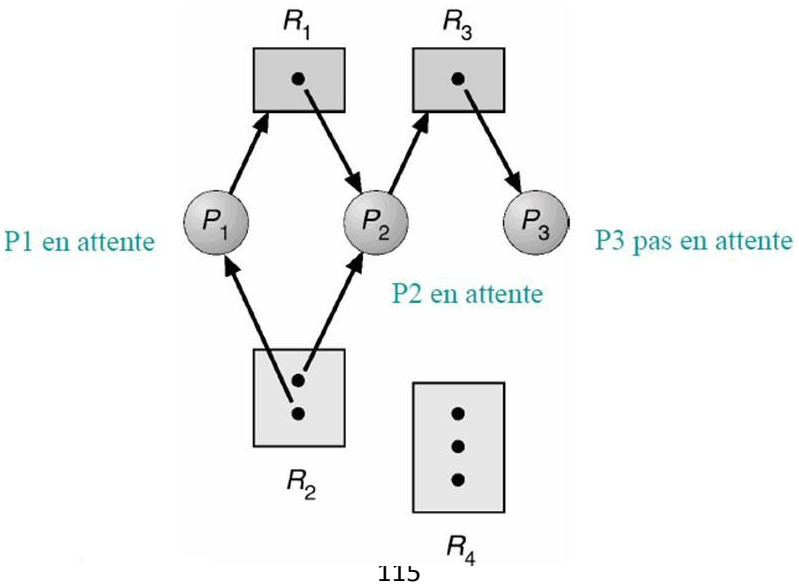


Figure 13 Schéma explicite d'un graphe d'allocation de ressources

○ Algorithme pour éviter l'interblocage

Il s'agit de construire un graphe d'allocation de ressources et voir s'il y a une manière dont tous les processus peuvent terminer. Dans le cas d'une ressource par type, l'algorithme cherche des circuits dans le graphe (algorithme d'ordre n_2 , avec n =



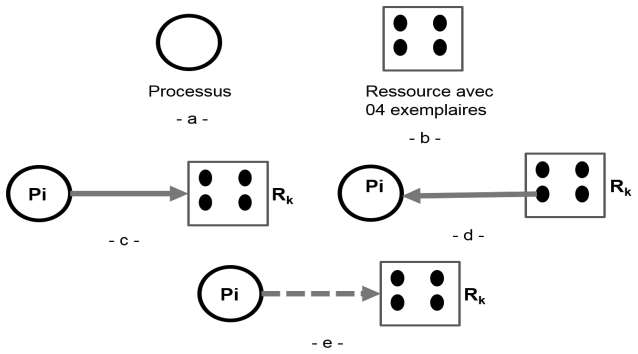
nombre de processus). Par contre, cet algorithme n'est pas applicable dans le cas de plusieurs ressources par type.

○ Algorithme du banquier

Cet algorithme pourrait s'appliquer dans un système bancaire pour s'assurer que la banque ne prête jamais son liquide disponible de telle sorte qu'elle ne puisse plus satisfaire tous ses clients.

Il est applicable à un système d'allocation de ressources avec possibilité de plusieurs instances pour chaque type de ressource. En entrant dans le système, tout processus doit déclarer le nombre maximal d'instances de chaque type de ressource dont il a besoin (il ne doit pas excéder le nombre total de ressources).

Quand un processus requiert un ensemble de ressources, le système doit déterminer si cette allocation laissera le système dans un état sain.



✔ Trouver un processus P_i non marqué dont la rangée i de R est inférieure à A

✔ Si un tel processus n'existe pas, alors l'état est non sûr (il y a interblocage). L'algorithme se termine.

✔ Sinon, ajouter la rangée i de C à A, et marquer le processus.

✔ Si tous les processus sont marqués, alors l'état est sûr et l'algorithme se termine, sinon aller à l'étape une.

L'algorithme du banquier permet bien d'éviter les interblocages, mais il est peu utilisé en pratique, car on ne connaît pas toujours à l'avance les besoins en ressources des processus.

8. Détecter et corriger les interblocages

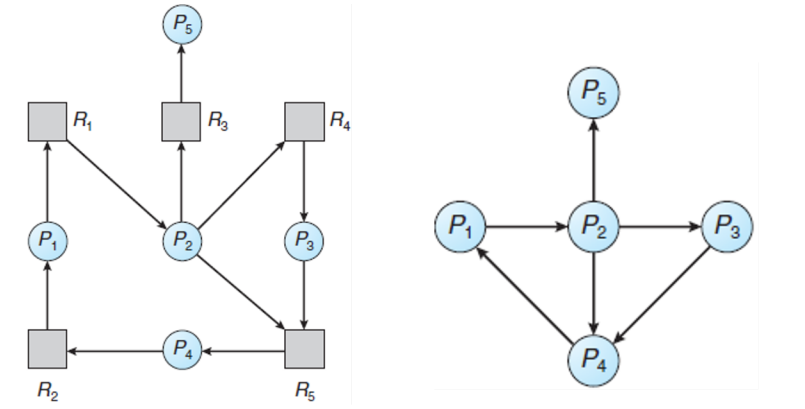
Si un système n'emploie pas d'algorithme pour prévenir ou pour éviter les interblocages, il doit, en cas de situation d'interblocage, détecter cet interblocage ensuite le corriger.

○ Détecter les interblocages

Instance unique pour chaque type de ressource

Dans ce cas, le système doit assurer la maintenance du graphe d'attente et appeler périodiquement un algorithme qui cherche un circuit dans ce graphe.

Le graphe d'attente est obtenu à partir du graphe d'allocation de ressources en éliminant les nœuds représentant les ressources. Ainsi, le graphe d'attente



Graphe d'allocation de ressources Graphe d'attente correspondant

représente un sous graphe du graphe d'allocation de ressources.

Figure 17 Exemple d'un graphe d'attente

9. Corriger les interblocages

Deux alternatives sont possibles pour corriger un interblocage détecté :

- ✔ Manuellement en informant l'opérateur.
- ✔ Automatiquement par le système.

Dans le cas d'une correction automatique, le système peut procéder par l'une des deux possibilités suivantes :

- Terminaison de processus : Dans ce cas, il est possible de :
  - o Avorter tous les processus en interblocage, ce qui va rompre le circuit d'interblocage, mais à un cout très élevé. En effet, tous les calculs partiels

effectués par ces processus seront perdus et recalculés plus tard.

- Avorter un processus a` la fois jusqu'à ce que le circuit d'interblocage soit éliminé. Ceci provoque une surcharge considérable puisque après chaque avortement, on doit appeler l'algorithme de détection de circuit.

- Réquisition de ressources : Enlever des ressources à un processus et les affecter à un autre pour défaire le circuit d'interblocage. Cette réquisition nous oblige à réfléchir sur :
  - La sélection d'une victime en précisant quelles ressources seront réquisitionnées et à partir de quels processus. On peut prendre comme facteur de coût le nombre de ressources détenues et le temps d'exécution.
  - Le retour en arrière puisqu'une perte de ressources implique forcément un redémarrage du processus à partir d'un état sain (état initial).
  - Comment garantir que les ressources ne seront pas réquisitionnées a` partir du même processus, en d'autres termes comment éviter la famine.

10. Conclusion

Une situation d'interblocage se produit si un ou plusieurs processus attendent indéfiniment un évènement qui ne peut être provoqué que par l'un des processus en attente. Le concepteur du système d'exploitation peut adopter l'une des approches suivantes vis à vis du problème d'interblocage :

- Prévenir ou éviter
- Détecter et corriger

- Ignorer

Aucune de ces approches n'est adéquate à elle seule pour tous les types de problème d'allocation de ressources dans les SE. Ainsi, ces approches peuvent être combinées, en permettant d'utiliser séparément une approche optimale pour chaque classe de ressources dans le SE.

11. Exercices

1. Considérons un système avec 5 processus P0 à P4 et 3 ressources de type A, B, C. La ressource de type A a 10 instances, B a 5 instances et le type C a 7 instances. Supposons qu'au moment t0, l'instantané suivant du système a été pris :

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Quel sera le contenu de la matrice des besoins ?

Réponse

Besoin [i, j] = Max [i, j] - Allocation [i, j].

Ainsi, le contenu de la matrice des besoins est :

Process	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

2. Considérons un système avec 4 types de ressources : R1 (3 unités), R2 (2 unités), R3 (3 unités), R4 (2 unités). Une politique d'allocation des ressources non préemptive est utilisée. À n'importe quelle instance donnée, une demande n'est pas prise en compte si elle ne peut pas être complètement satisfaite. Trois processus P1, P2, P3 demandent les sources comme suit s'ils sont exécutés indépendamment.

Processus P1 :

t=0 : demande 2 unités de R2

t=1 : demande 1 unité de R3

t=3 : demande 2 unités de R1

t=5 : libère 1 unité de R2

et 1 unité de R1.

t=7 : libère 1 unité de R3

t=8 : demande 2 unités de R4

t=10 : Termine

le processus P2 :

t=0 : demande 2 unités de R3

t=2 : demande 1 unité de R4

t=4 : demande 1 unité de R1

t=6 : libère 1 unité de R3

t=8 : Termine

Processus P3 :

t=0 : demande 1 unité de R4

t=2 : demande 2 unités de R1

t=5 : libère 2 unités de R1

t=7 : demande 1 unité de R2

t=8 : demande 1 unité de R3

t=9 : Termine

Lequel des énoncés suivants est VRAI si les trois processus s'exécutent simultanément à partir du temps t=0 ?

Réponse : Tous les processus se termineront sans aucune impasse

3. Considérons l'impasse de trafic décrite dans la figure.
   - Montrez que les quatre conditions nécessaires à l'existence d'une impasse sont effectivement réunies dans cet exemple.
   - Énoncez une règle simple pour éviter les blocages dans ce système.

Réponse :

- Les quatre conditions nécessaires à une impasse sont (1) l'exclusion mutuelle, (2) le maintien et l'attente, (3) l'absence de préemption et (4) l'attente circulaire. La condition d'exclusion mutuelle est remplie, car une seule voiture peut occuper un espace sur la chaussée. Il y a maintien et attente lorsqu'une voiture conserve sa place sur la chaussée en attendant de pouvoir avancer sur la chaussée. Une voiture ne peut pas être retirée (c'est-à-dire préemptée) de sa position sur la chaussée. Enfin, il y a bien une attente circulaire, car chaque voiture attend qu'une autre voiture avance. La condition d'attente circulaire est également facilement observable sur le graphique.
- Une règle simple qui permettrait d'éviter ce blocage de la circulation est qu'une voiture ne peut pas s'engager dans une intersection s'il est clair qu'elle ne sera pas en mesure de la dégager immédiatement.

4. Considérez la situation d'impasse qui pourrait se produire dans le problème des philosophes à table lorsque les philosophes obtiennent les baguettes une par une. Discutez de la manière dont les quatre conditions nécessaires à une impasse sont effectivement réunies dans ce cas. Discutez comment les blocages pourraient être évités en éliminant l'une des quatre conditions.

Réponse :

Une impasse est possible parce que les quatre conditions nécessaires sont réunies de la manière suivante : 1) l'exclusion mutuelle est requise pour les baguettes, 2) les philosophes ont tendance à conserver la baguette qu'ils ont en main en attendant l'autre baguette, 3) il n'y a pas de préemption des baguettes dans le sens où une baguette attribuée à un philosophe ne peut pas être retirée par la force, et 4) il y a une possibilité d'attente circulaire.

Les blocages pourraient être évités en surmontant ces conditions de la manière suivante : 1) autoriser le partage simultané des baguettes, 2) demander aux philosophes de renoncer à la première baguette s'ils ne parviennent pas à obtenir l'autre baguette, 3) autoriser le retrait forcé des baguettes si un philosophe en possède une depuis longtemps, et 4) imposer une numérotation des baguettes et toujours obtenir la baguette portant le numéro le plus bas avant d'obtenir celle portant le numéro le plus haut.

5. Expliquez la différence entre la fragmentation interne et externe.

Réponse :

La fragmentation interne est la zone d'une région ou d'une page qui n'est pas utilisée par le travail qui occupe cette région ou cette page. Cet espace n'est pas disponible pour le système jusqu'à ce que le travail soit terminé et que la page ou la région, soit libérée.

Références

1. Systèmes d'exploitation » par Andrew Tanenbaum – cet ouvrage est régulièrement réédité. Référence incontournable (pour les cours de système d'exploitation de première, deuxième et ...)
2. Architecture et technologie des ordinateurs » par Paolo Zannella, Yves Ligier – DUNOD 2002 (ISBN 2-10-003801-X)
3. Principes des Systèmes d'Exploitation, Pages Bleues, les Manuels de l'Etudiant, 2007
4. Systèmes d'exploitation, Collection Synthex, Pearson Education, 2006
5. Système d'Exploitation, Mécanismes de Base", OPU, 2005
6. Operating System Concepts", 7th Edition, John Wiley & Sons Editions, 2005, 921 p
7. Systèmes d'Exploitation : Systèmes Centralisés – Systèmes Distribués", 3ème édition, Editons DUNOD, Prentice Hall, 2001
8. Principes des Systèmes d'Exploitation, traduit par M. Gatumel, 4ème édition, Editions Addison-Wesly France, SA, 1994
9. Architecture des Systèmes d'Exploitation, Edition Hermès, 1990
10. Principes des Systèmes d'Exploitation des Ordinateurs, Editions DUNOD, 1987
11. Architecture de l'Ordinateur, Editions Pearson Education, 2006
12. Collection Synthex, Pearson Education, 2006

13. LINUX, Programmation Système et Réseau, Cours et Exercices Corrigés, Editions DUNOD
14. Conception du système UNIX", Editions Masson, 1989

Sites Internet –

1. Site du cours à l'École Polytechnique de Montréal – http://www.cours.polymtl.ca/inf3600/
2. Sites pour les développeurs Linux – http ://developer.kde.org
3. Site du serveur Web Apache pour Linux/Unix – http ://www.apache.org
4. Sites des distributions et des noyaux Linux – http ://www.redhat.com
5. Sites des bureaux et des gestionnaires de fenêtres Linux– http ://www.gnome.org