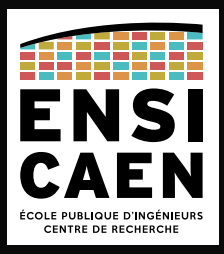


Systèmes d'exploitation

Éléments de programmation système sous Unix

Alain Lebret

2024

Table des matières

Avant-propos 1

Présentation de l'enseignement 3

1 Introduction aux systèmes d'exploitation 5
- 1.1 Généralités sur les systèmes d'exploitation . . . 5
- 1.2 Particularités du système Unix . . . 9
- 1.3 Rôle de la programmation système . . . 12
- 1.4 Défis en programmation système . . . 14

2 Processus 15
- 2.1 Qu'est-ce qu'un processus ? . . . 15
- 2.2 États d'un processus . . . 15
- 2.3 Parallélisme . . . 16
- 2.4 Réalisation des processus . . . 17
- 2.5 Ordonnancement des processus . . . 18
- 2.6 Stratégies d'ordonnancement . . . 19
- 2.7 Blocage des processus . . . 21
- 2.8 Processus sous Unix . . . 22
- 2.9 Environnement d'un processus . . . 23
- 2.10 Vie et mort des processus . . . 24
- 2.11 Création des processus sous Unix . . . 25
- 2.12 Hiérarchie des processus sous Unix . . . 29
- 2.13 Quelques interactions entre processus sous Unix . . . 29
- 2.14 Attention aux zombies . . . 31
- 2.15 Processus orphelin . . . 32
- 2.16 Exécution d'un programme spécifié . . . 33
- 2.17 Lien entre fork(), exec(), wait() et exit() . . . 35
- 2.18 Exercices . . . 37

3 Les threads 39
- 3.1 Introduction . . . 39
- 3.2 Espace mémoire d'un processus . . . 39
- 3.3 Représentation alternative de l'espace d'un processus . . . 40
- 3.4 Espace mémoire d'un processus multithreads . . . 40
- 3.5 Vue logique des threads . . . 41
- 3.6 Exécution concurrente des threads . . . 42
- 3.7 Comparaison avec les processus . . . 42
- 3.8 Threads POSIX (bibliothèque *Pthread*) . . . 43
- 3.9 Programme utilisant la bibliothèque *Pthread* . . . 44
- 3.10 Utilisation des attributs d'un thread . . . 45
- 3.11 Exécution du programme utilisant des threads . . . 49

3.12 Exemple d'application . . . 49
3.13 Partage de variables dans un programme multithreads . . . 52
3.14 Accès à la pile d'un autre thread . . . 53
3.15 Projection des variables en mémoire . . . 53
3.16 Analyse des variables partagées . . . 53
3.17 Fonctions sûres et re-entrantes . . . 54
3.18 Exercices . . . 56

4 Fichiers et flux d'entrées-sorties 59
4.1 Introduction . . . 59
4.2 À propos du dossier courant . . . 63
4.3 Règles de recherche des fichiers . . . 63
4.4 Utilisations courantes des fichiers . . . 64
4.5 Utilisation des fichiers depuis l'interface de commande . . . 67
4.6 Structure des fichiers sous Unix . . . 68
4.7 Structure de donnée i-noeud . . . 69
4.8 Structure des dossiers sous Unix . . . 70
4.9 Protection des fichiers sous Unix . . . 71
4.10 Appels système d'accès aux fichiers . . . 72
4.11 Fichiers et flux d'entrées/sorties . . . 79
4.12 Manipuler les flux d'entrées/sorties (commandes) . . . 80
4.13 Redirection et copie de descripteurs . . . 80
4.14 Exercices . . . 81

5 Gestion de la mémoire virtuelle 83
5.1 Introduction . . . 83
5.2 Hiérarchie des mémoires . . . 83
5.3 Motivations pour une mémoire virtuelle . . . 84
5.4 Mémoire virtuelle . . . 84
5.5 Structuration des espaces d'adressage . . . 85
5.6 Obtenir la taille des pages sur un système . . . 85
5.7 Mémoire virtuelle paginée . . . 86
5.8 Principe de la conversion d'adresse . . . 86
5.9 Segmentation et mémoire virtuelle d'un processus . . . 87
5.10 Visualisation des segments sous Unix . . . 88
5.11 Nature des segments mémoires . . . 91
5.12 Couplage entre mémoire virtuelle et fichiers . . . 91
5.13 Partage de segments entre mémoires virtuelles . . . 92
5.14 Précisions sur les appels `fork()` et `exec()` . . . 94
5.15 Allocation dynamique de mémoire . . . 95
5.16 Exercices . . . 99

6 Communication interprocessus 101
6.1 Introduction . . . 101
6.2 Communication par signaux . . . 101
6.3 Communication par tube . . . 116
6.4 Communication par mémoire partagée . . . 122
6.5 Communication par file de messages . . . 130
6.6 Communication par sockets . . . 135

6.7 Exercices . . . 145

7 Synchronisation des processus 149
7.1 Introduction . . . 149
7.2 Compétition entre processus . . . 149
7.3 Verrouillage de fichiers . . . 155
7.4 Interblocage . . . 157
7.5 Coopération entre processus . . . 162
7.6 Conclusion . . . 166
7.7 Exercices . . . 167

8 Études de cas 171
8.1 Introduction . . . 171
8.2 Étude de cas n°1 : cybersécurité . . . 173
8.3 Étude de cas n°2 : pipeline CI/CD . . . 177
8.4 Étude de cas n°3 : monétique . . . 182
8.5 Étude de cas n°4 : traitement d'images . . . 185
8.6 Étude de cas n°5 : intelligence artificielle . . . 187
8.7 Étude de cas n°6 : simulation numérique . . . 191
8.8 Étude de cas n°7 : développement de jeux vidéo . . . 194
8.9 Étude de cas n°8 : gestion d'un système de voix sur IP (VoIP) . . . 198
8.10 Étude de cas n°9 : blockchain et minage de cryptomonnaies . . . 200
8.11 Étude de cas n°10 : analyse de séquences génomiques en bioinformatique . . . 205
8.12 Étude de cas n°11 : gestion d'une grille de traitement . . . 210
8.13 Étude de cas n°12 : surveillance des performances d'une grille de traitement . . . 215
8.14 Conclusion . . . 225

Éléments de réponse aux exercices 227
Chapitre sur les processus . . . 227
Chapitre sur les threads . . . 230
Chapitre sur les fichiers . . . 232
Chapitre sur la mémoire virtuelle . . . 234
Chapitre sur la communication interprocessus . . . 235
Chapitre sur la synchronisation . . . 240

Annexes 243
Annexe A : appeler `return` ou `exit()` ? . . . 243
Annexe B : Programmation système sous Ms-Windows . . . 245
Annexe C : Interfacer Java et C . . . 255
Annexe D : Algorithmes d'ordonnancement . . . 261

Glossaire 271

Index 279

Bibliographie 283

Table des figures

1.1 Structure d'un système d'exploitation de type Unix . . . 10

2.1 États d'un processus . . . 15
2.2 Exécutions séquentielle, parallèle et pseudo-parallèle. . . . 17
2.3 Ordonnancement des processus . . . 18
2.4 Table des blocs de contrôle des processus. . . . 19
2.5 Ordonnancement et contexte. . . . 19
2.6 Blocage d'un processus . . . 22
2.7 Terminaison d'un processus . . . 25
2.8 Hiérarchie des processus . . . 29
2.9 Association `fork()` et `exec()`. . . . 36

3.1 Espace mémoire d'un processus . . . 40
3.2 Autre représentation de l'espace mémoire d'un processus . . . 40
3.3 Représentation de l'espace mémoire d'un processus multithreads . . . 41
3.4 Vue logique de *threads* . . . 41
3.5 Exemple d'exécutions concurrente et séquentielle . . . 42
3.6 Exécution du programme utilisant des *threads*. . . . 49
3.7 Multiplication matricielle monothread. . . . 50
3.8 Multiplication matricielle avec 4 threads. . . . 51
3.9 Comparaison des temps de calcul monothread / 4 threads . . . 52
3.10 Variables accessibles entre threads . . . 53
3.11 Projection des variables en mémoire . . . 53
3.12 Ensemble des fonctions re-entrantes. . . . 56

4.1 Désignation et différents points d'accès aux fichiers. . . . 59
4.2 Arborescence des fichiers sous Unix . . . 62
4.3 Blocs de fichier sur un disque . . . 68
4.4 i-noeud d'un fichier. . . . 69
4.5 Dossiers et table des i-noeuds. . . . 70
4.6 Exemples de déplacement du pointeur de fichier‘. . . . 75
4.7 Exemples de lecture dans un fichier. . . . 76
4.8 Exemples d'écriture dans un fichier. . . . 77
4.9 Entrée/sorties d'un processus et table des descripteurs. . . . 79
4.10 Exemple de redirection vers un fichier. . . . 80

5.1 Hiérarchie des mémoires [5]. . . . 83
5.2 Rapports de temps d'accès selon [5]. . . . 84
5.3 Mémoire virtuelle et table des pages. . . . 86
5.4 Conversion d'adresse avec MMU et TLB. . . . 86
5.5 Segments mémoire d'un processus. . . . 87
5.6 Association segment / fichier (swap). . . . 91

5.7 Permissions des segments après un appel à `execve()`. 95
5.8 Exemple d'allocations dynamiques sur le tas. 96
5.9 Exemple de tas avec des pointeurs de début des zones libres. 96
5.10 Exemple de tas avec des pointeurs de début et fin des zones libres. 97
5.11 Exemple de groupes. 97

6.1 Fonctionnement d'un signal. 102
6.2 États d'un travail. 104
6.3 Établissement d'un tube anonyme du parent vers l'enfant. 117
6.4 Droits d'accès d'un tube nommé. 120
6.5 Établissement d'une communication unidirectionnelle par tube nommé. 120
6.6 Utilisation d'une mémoire partagée entre deux processus. 122
6.7 Mise en oeuvre d'une mémoire partagée entre deux processus. 125
6.8 Mise en oeuvre d'une file de messages entre deux processus. 132
6.9 Couches TCP/IP et UDP/IP. 135
6.10 Big endian ou little endian. 137
6.11 Client/serveur en mode itératif. 141
6.12 Client/serveur en mode concurrent. 142
6.13 Emetteur/récepteur en mode non connecté. 143

7.1 Interblocage lors de l'accès à deux fichiers. 159
7.2 Résolution de l'interblocage lors de l'accès à deux fichiers. 160
7.3 Exemple de graphe d'allocation des ressources. 161

8.1 Exemple d'une *blockchain* avec trois blocs. 201
2 Architecture de Ms-Windows 2000 (source : Wikipedia -- auteur Elisardojm sous licence CC BY-SA 3.0). 245
3 Étapes de mise en oeuvre de JNI. 255
4 Classes JNI. 256
5 Transformation des méthodes en fonctions. 256

Avant-propos

Dans ce document, vous trouverez trois types de paragraphes particuliers permettant de mettre en évidence des points importants, des remarques et des astuces.

Un point important.

Une remarque.

Une astuce.

D'autre part, des exemples de code qui illustrent le cours sont disponibles à l'adresse suivante :

https://github.com/alainlebret/os

Présentation de l'enseignement

Ce cours explore les mécanismes fondamentaux de la programmation système sur les systèmes d'exploitation de type Unix, tant du point de vue de l'utilisateur que du développeur.

Destiné principalement à de futurs développeurs, ce cours inclut :

- Un format pédagogique comprenant 10 heures de cours théoriques et 16 heures de travaux pratiques réparties en 8 séances.
- Une évaluation à parts égales entre la note d'examen et celle de travaux pratiques.

Prérequis

- Connaissances en architecture des ordinateurs.
- Bases de l'algorithmique.
- Programmation en langage C.

Pourquoi étudier Unix ?

- Unix est une famille de systèmes d'exploitation avec de nombreuses déclinaisons.
- Il supporte les environnements multi-tâches et multi-utilisateurs.
- Son impact historique inclut l'implémentation du concept de temps partagé.
- Il repose sur des concepts de base simples et une interface historiquement épurée.
- Il est souvent associé à une tradition de code ouvert, avec des versions libres comme BSD et Linux.

Pourquoi ne pas étudier Ms-Windows ?

- En raison des contraintes de temps.
- Bien que substantielles, les différences avec Unix restent comparables en termes de concepts de base.
- Un complément sur les appels système Ms-Windows est inclus en annexe afin d'élargir la perspective des apprenants. Le lecteur qui voudrait aller plus loin dans la programmation système sous Ms-Windows peut consulter l'ouvrage de Hart [15].

1 Introduction aux systèmes d'exploitation

1.1 Généralités sur les systèmes d'exploitation

1.1.1 Qu'est-ce qu'un système d'exploitation ?

Un système d'exploitation, ou *operating system* en anglais, est un ensemble de programmes essentiels servant d'intermédiaire entre le matériel informatique, typiquement composé d'un ou plusieurs processeurs, et les applications logicielles mises en oeuvre par les utilisateurs. Selon Silberschatz, un système d'exploitation est avant tout conçu pour rendre l'utilisation du matériel plus conviviale et efficace [32].

Par exemple, en ce qui concerne la gestion des fichiers, le système d'exploitation autorise les applications à accéder aux données sans nécessiter une compréhension approfondie des détails physiques du stockage, tels que le type de disque dur ou les systèmes de fichiers sous-jacents. En simplifiant ces interactions complexes, le système d'exploitation fournit une couche d'abstraction qui facilite le développement et l'exécution des applications tout en optimisant l'utilisation des ressources matérielles.

1.1.2 Les rôles d'un système d'exploitation

Un système d'exploitation remplit plusieurs rôles essentiels, parmi lesquels :

- **Gestion des ressources** : Le système d'exploitation est responsable de l'allocation des ressources matérielles et logicielles, telles que l'accès au processeur, à la mémoire et aux périphériques, ainsi qu'aux diverses applications qui en font la demande. Il organise le partage de ces ressources dans le but d'éviter les conflits et d'optimiser leur utilisation. Par exemple, en partageant le processeur entre plusieurs tâches à l'aide d'un ordonnancement efficace, il met en oeuvre des algorithmes comme l'ordonnancement circulaire ou par priorité lorsqu'il s'agit de décider quel processus doit s'exécuter à un moment donné.
- **Adaptation d'interface** : Le système d'exploitation simplifie et rend plus intuitive l'interaction avec le matériel. Plutôt que d'exiger des développeurs qu'ils gèrent explicitement la mémoire ou les cycles du processeur, il offre des abstractions comme les fichiers pour le stockage ou les processus pour l'exécution des programmes. Cela masque la complexité et les contraintes physiques du système [32 - chap. 11 et 12,35 - chap. 4,4 - chap. 10,5 - chap. 1]. Par exemple, le système de fichiers permet de manipuler des données sans se soucier des détails de leur stockage physique.
- **Sécurité et contrôle d'accès** : Le système protège les données et les ressources matérielles contre les accès non autorisés et les menaces de sécurité. Cela inclut la gestion des utilisateurs, des permissions et la surveillance des activités pour détecter d'éventuelles anomalies [32,35].
- **Gestion des informations** : Le système organise les données à l'aide de systèmes de fichiers, afin de faciliter leur stockage et leur récupération de manière efficiente. Cette gestion comprend la création, la lecture, l'écriture et la suppression de fichiers et de dossiers, rendant les données accessibles et manipulables de manière ordonnée.
- **Gestion du temps** : Elle implique la gestion de l'horloge système et des temporisateurs (*timers*), permettant de synchroniser les opérations et de garantir le respect des délais imposés par les processus. Cette gestion est essentielle pour coordonner les différentes activités du système et assurer une performance optimale.

- **Gestion des erreurs** : Le système d'exploitation détecte, analyse et réagit aux erreurs matérielles et logicielles dans le but de maintenir la stabilité et la fiabilité du système. Cela peut inclure la journalisation des erreurs et la mise en place de mécanismes de récupération [32 - chap. 33,35].

En résumé, les rôles d'un système d'exploitation sont variés et interconnectés. Ils visent tous à optimiser l'utilisation des ressources matérielles et logicielles, à assurer la sécurité et la stabilité du système, et à fournir une interface utilisateur efficace et intuitive. Ces rôles constituent le fondement sur lequel repose le fonctionnement d'un système informatique moderne.

1.1.3 Où trouve-t-on des systèmes d'exploitation ?

Les systèmes d'exploitation sont omniprésents, intégrés dans une vaste gamme de dispositifs, allant des appareils mobiles et autres dispositifs électroniques aux superordinateurs. En 2024, plus de 85 % des serveurs Web dans le monde fonctionnent sur des variantes d'Unix/Linux, soulignant leur prédominance dans les infrastructures critiques [40].

Unix et ses nombreuses variantes, telles que BSD, Linux ou Mac OS X, sont reconnus pour leur stabilité et robustesse. Ces systèmes sont couramment utilisés sur les serveurs et les stations de travail. Parallèlement, Ms-Windows est largement adopté dans les environnements personnels et professionnels grâce à sa large compatibilité avec les logiciels commerciaux, notamment les jeux.

Sur les téléphones et tablettes, Unix est optimisé pour des interfaces utilisateur mobiles et la gestion de ressources limitées[1]. Bien que moins répandu que ses concurrents Unix, Windows 10 Mobile trouve encore des applications dans divers appareils mobiles.

Pour les dispositifs IoT et embarqués, Unix (notamment « Linux embarqué »), ainsi que d'autres systèmes temps réel comme FreeRTOS et QNX, sont fréquemment utilisés. Dans le domaine de l'informatique en nuage (*cloud computing*), Unix/Linux domine également grâce à sa robustesse, sa flexibilité, et son soutien étendu pour les conteneurs et les microservices [32 - chap. 1].

Bien que le système d'exploitation Ms-Windows demeure majoritaire, les tendances montrent une adoption croissante de Linux (ou de Mac OS X) sur les postes de travail des développeurs et dans les environnements d'entreprise en raison de leur sécurité et de leurs performances [32 - chap. 1].

1.1.4 Noyau d'un système d'exploitation

Le **noyau** (*kernel*) constitue le coeur du système d'exploitation. Il regroupe les programmes fondamentaux qui soutiennent les fonctionnalités de base du système, indépendamment des spécificités matérielles de la plateforme. Le noyau travaille en étroite collaboration avec les **gestionnaires d'équipements matériels**, également appelés **pilotes** (*drivers* en anglais), qui assurent la communication entre le noyau et les périphériques d'entrée-sortie. Par exemple, un pilote de périphérique gère la communication entre le noyau et l'interface réseau, facilitant ainsi l'interconnexion de la machine avec d'autres systèmes.

Les noyaux peuvent être classés en deux grandes catégories : **monolithique** et **micronoyau**. Les noyaux monolithiques, tels que celui de Linux, intègrent une large gamme de fonctionnalités directement dans le noyau principal, permettant une gestion centralisée et souvent plus performante des ressources et des processus. En revanche, les micronoyaux, comme celui de Minix, visent à minimiser les fonctions du noyau afin d'augmenter la modularité et la fiabilité. Cette architecture répartit les fonctions du noyau en plusieurs services indépendants qui facilitent la maintenance et la sécurité du système [35].

[1] Android est basé sur Linux, quant à iOS, il est basé sur une version de BSD.

Mac OS X et Windows NT sont des exemples de systèmes d'exploitation qui utilisent une architecture de noyau hybride. Mac OS X intègre un micronoyau appelé Mach et développé à l'origine à l'Université Carnegie Mellon, et le combine avec des composants issus de la couche utilisateur du système Unix BSD [32 - ann. D,33]. Le micronoyau Mach facilite une gestion efficace de la mémoire et des processus dans Mac OS X, tandis que les composants BSD fournissent une stabilité et une richesse fonctionnelle accrues. De son côté, Windows NT utilise un micronoyau conçu par Microsoft et qui s'inspire de différents micronoyaux, entres autres Mach.

Ainsi, le noyau joue un rôle central et critique dans le fonctionnement d'un système d'exploitation, car il orchestre la gestion des ressources, la communication entre les composants matériels et logiciels, et la sécurité du système. Ses différentes architectures reflètent les choix et priorités des concepteurs de systèmes d'exploitation, chaque approche présentant ses propres avantages et inconvénients en termes de performance, de modularité et de fiabilité.

1.1.5 Modes superviseur et utilisateur

Un processeur dispose de deux modes de fonctionnement distincts, accessibles à l'aide d'un « bit de mode » : le mode **superviseur** et le mode **utilisateur**.

Le mode superviseur, également connu sous le nom de « mode noyau », est employé par le noyau du système d'exploitation. Ce mode permet l'exécution de toutes les instructions, y compris celles nécessitant une interaction directe avec le matériel. Il offre un contrôle total sur le système, permettant ainsi une gestion optimale des ressources et garantissant la protection contre les erreurs et les défaillances. En mode superviseur, le noyau peut effectuer des opérations essentielles telles que la gestion de la mémoire, des périphériques, et le contrôle des entrées/sorties, assurant ainsi le bon fonctionnement et la sécurité globale du système.

En revanche, le mode utilisateur est destiné aux programmes et utilitaires exécutés par les utilisateurs. Ce mode restreint l'accès à certaines instructions critiques afin de protéger le système contre les intrusions et les erreurs potentielles. Les applications fonctionnent dans un environnement contrôlé, où leur accès aux ressources matérielles et aux opérations critiques est strictement régulé. Cette limitation empêche toute modification ou perturbation directe du matériel et des ressources système. Ces restrictions sont cruciales puisqu'il s'agit de maintenir l'intégrité du système et des données, en réduisant les risques de dommages accidentels ou intentionnels.

L'utilisation conjointe des deux modes est essentielle de manière à garantir une exécution sécurisée et stable des programmes.

1.1.6 Appels système et primitives

Les **appels système** et les **primitives** du noyau sont des éléments fondamentaux des systèmes d'exploitation. Ils permettent aux programmes en mode utilisateur de solliciter les services du noyau, incluant la gestion des fichiers, la création de processus, et la communication interprocessus.

Un appel système est un mécanisme qui autorise les programmes en mode utilisateur à demander des services au noyau. Ce mécanisme utilise une interruption logicielle pour faciliter la transition du mode utilisateur au mode noyau et se déroule en plusieurs étapes :

1. La transition du mode utilisateur au mode superviseur.
2. La récupération et la validation des paramètres de l'appel.
3. L'exécution de la fonction demandée par l'appel système.
4. La récupération des valeurs de retour.
5. Le retour au programme appelant, avec une réversion au mode utilisateur.

Par exemple, lorsqu'un programme souhaite lire un fichier sur Unix, il utilise l'appel système `read()`, permettant l'accès aux données sur le disque sans avoir à gérer les détails de bas niveau.

Une primitive est une fonction interne au noyau qui réalise une tâche critique de bas niveau. Contrairement aux appels système, les primitives ne sont pas directement accessibles aux programmes utilisateurs; elles sont utilisées par le noyau pour implémenter des fonctionnalités système complexes. Les primitives s'occupent souvent d'opérations telles que la manipulation des structures de données du noyau ou des interactions matérielles spécifiques.

Par exemple, l'appel système `fork()` disponible sur toute distribution d'Unix permet de créer un nouveau processus en dupliquant le processus appelant. Cet appel système repose sur plusieurs primitives du noyau, telles que `mm_copy()` dans le cas spécifique du noyau Linux. Cette primitive est responsable de la copie de zones mémoires en interagissant directement avec la mémoire physique. Cette interaction directe avec le matériel, qui est gérée par les primitives, assure l'efficacité et la robustesse des opérations du noyau.

La distinction entre appels système et primitives est donc importante pour comprendre l'architecture et le fonctionnement des systèmes d'exploitation. Les appels système fournissent une interface sécurisée et contrôlée pour les programmes utilisateurs, tandis que les primitives, opérant en coulisses, réalisent des tâches critiques et spécifiques nécessaires au bon fonctionnement du système. Ensemble, ces mécanismes permettent une gestion efficace et sécurisée des ressources et des opérations système.

1.1.7 Interfaces d'un système d'exploitation

Les systèmes d'exploitation proposent deux types d'interfaces principales qui permettent aux utilisateurs d'interagir avec le système de manière efficace et intuitive.

L'**interface utilisateur** (ou *interface de commande*) est conçue pour être directement utilisable par les humains. Elle peut se présenter sous une forme textuelle, comme les interpréteurs de commandes qui s'exécutent dans un terminal, et où les utilisateurs interagissent avec le système en entrant des commandes. Alternativement, elle peut être graphique, offrant des interactions à l'aide de fenêtres, d'icônes et de menus, de manière à faciliter l'accès aux utilisateurs moins familiers avec les commandes textuelles.

L'**interface de programmation** (API, ou *application programming interface*), en revanche, est destinée à être utilisée par les programmes qui s'exécutent sur le système. Cette interface est constituée d'un ensemble d'appels système qui permettent aux applications de solliciter des services du noyau, ainsi que de fonctions de plus haut niveau. Ces services peuvent inclure la gestion des fichiers, des processus, de la mémoire, et bien d'autres encore.

Prenons l'exemple de la copie d'un fichier pour illustrer l'utilisation de ces interfaces. Sous Unix, en utilisant l'interface de commande textuelle, cela nécessite d'entrer la commande suivante dans un terminal :

```
$ cp fichier_origine fichier_destination
```

Cette commande déclenche une série d'opérations en arrière-plan, où le système d'exploitation, par l'intermédiaire du noyau, gère les détails complexes de la lecture et de l'écriture des données entre les fichiers source et destination.

En revanche, en utilisant l'interface de programmation, il est nécessaire d'écrire un programme dans un langage tel que le C. Ce programme doit recourir à des appels système tels que `read()` pour lire les données du fichier source et `write()` pour écrire ces données dans le fichier de destination (ou encore passer par des fonctions de plus haut niveau telles que `fread()` et `fwrite()` qui emploient ces appels système). En voici un exemple :

```
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
```

```
#define BLKSIZE 1024

void copy_file(int from_fd, int to_fd) {
    char buf[BLKSIZE];
    ssize_t bytesread, byteswritten;
    char *bp;

    while ((bytesread = read(from_fd, buf, BLKSIZE)) > 0) {
        bp = buf;
        while ((byteswritten = write(to_fd, bp, bytesread)) > 0) {
            if (byteswritten <= 0) {
                if (errno != EINTR)
                    break;
            } else {
                bp += byteswritten;
                bytesread -= byteswritten;
            }
        }
        if (byteswritten < 0)
            break;
    }
}
```

Bien que ce niveau d'interaction exige une compréhension plus approfondie de la programmation et des systèmes d'exploitation, il permet un contrôle plus fin et une optimisation potentielle des performances et des ressources.

La distinction entre ces interfaces montre la flexibilité des systèmes d'exploitation modernes. Les interfaces utilisateur offrent une accessibilité et une simplicité d'utilisation, tandis que les API permettent une interaction détaillée et précise avec les fonctionnalités du noyau, adaptées aux besoins des développeurs de logiciels et des applications complexes.

1.2 Particularités du système Unix

1.2.1 Bref historique d'Unix

Unix est un système d'exploitation multitâches et multi-utilisateurs développé dans les années 1960 et 1970 aux laboratoires Bell[2] par Ken Thompson, Dennis Ritchie, et d'autres collaborateurs. Il a exercé une influence profonde sur les systèmes d'exploitation modernes.

Quelques dates marquantes

En 1961, le **CTSS** (*compatible time-sharing system*) développé au MIT devient le premier système d'exploitation à partage de temps et marque une étape notable dans l'évolution des systèmes multitâches et multi-utilisateurs. Puis, en 1964, Multics (*multiplexed information and computing service*), développé conjointement par le MIT et les laboratoires Bell, établit les fondations pour les futurs systèmes d'exploitation. En 1969, Ken Thompson et Dennis Ritchie, ayant travaillé sur le projet Multics, créent Unics (*uniplexed information and computing service*), qui sera plus tard renommé Unix.

[2] Les laboratoires Bell, fondés en 1925 dans le New Jersey aux États-Unis, ont été rachetés par Nokia en 2016.

La réécriture d'Unix en langage C en 1973 accroit sa portabilité et ouvre la voie à son adoption sur différentes plateformes. En 1979, la publication de la version 7 d'Unix marque un tournant important car elle introduit le concept de tube, qui va permettre aux processus de communiquer plus efficacement en facilitant la construction de pipelines de commandes.

Dans les années 1980, les systèmes BSD (*Berkeley software distribution*) émergent et proposent de nouvelles fonctionnalités telles que les *sockets* réseau ce qui améliore la flexibilité des systèmes Unix. En 1983 AT & T publie Unix System V, une version majeure qui introduit la notion de système de fichiers virtuel (VFS), ainsi que d'autres améliorations. La même année, Richard Stallman lance le projet GNU qui vise à créer un système d'exploitation libre basé sur Unix.

En 1991, Linus Torvalds publie la première version du noyau Linux, un système d'exploitation *open-source* basé sur le système pédagogique Minix créé durant les années 1980 par Andrew S. Tanenbaum à l'université Vrije aux Pays-Bas. Enfin, en 1997, Apple adopte un noyau Unix pour son système d'exploitation Mac OS X, basé sur BSD et le micro-noyau Mach.

Le lecteur intéressé par l'histoire des systèmes d'exploitation peuvent consulter l'article succinct de Krakowiak [20] ou au premier chapitre de l'ouvrage de Tanenbaum [35].

1.2.2 Architecture générale d'Unix

Unix se caractérise par une architecture modulaire et une conception à la fois simple et efficace. Ses principales composantes incluent le noyau, l'interpréteur de commandes, le système de fichiers, ainsi que divers utilitaires et applications. La figure 1.1 illustre la structure en couche du système.

Parmi les exemples d'interpréteurs de commandes couramment utilisés, on trouve bash, zsh, et csh.

Le système de fichiers organise et stocke les données sous forme d'une arborescence hiérarchique de dossiers et de fichiers. Les points de montage permettent d'intégrer différents systèmes de fichiers dans un espace de noms unique. En outre, la gestion des permissions et des liens assure la sécurité et l'organisation des données.

Enfin, Unix comprend une grande variété d'utilitaires et d'applications qui sont fournis pour accomplir diverses tâches, allant de la gestion des fichiers à l'édition de texte. Parmi ces outils, on peut citer `ls`, `cp`, `mv`, `grep`, `awk`, et `sed`. Les utilisateurs ont également la possibilité d'ajouter des applications et des outils personnalisés pour répondre à leurs besoins spécifiques.

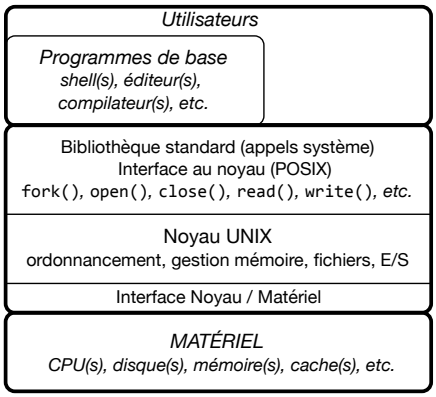


Fig. 1.1 : Structure d'un système d'exploitation de type Unix

1.2.3 Principes fondamentaux d'Unix

Unix repose sur une philosophie de conception simple et efficace, où presque tout est représenté comme un fichier. Cette approche uniforme simplifie la manipulation et l'interaction avec les composants du système. Par exemple, accéder à un périphérique par l'intermédiaire de son pilote s'effectue comme s'il s'agissait d'un fichier, facilitant ainsi grandement la programmation système.

Unix privilégie également l'utilisation de petits programmes spécialisés qui accomplissent une seule tâche, mais la réalisent de manière optimale. Ces programmes peuvent être combinés pour réaliser des tâches plus complexes, grâce à l'utilisation de **pipelines** et de **redirections** de manière à diriger la sortie de l'un vers l'entrée de l'autre, grâce aux opérateurs « | », « > », et « < ».

Par exemple, pour trouver tous les fichiers « .txt » dans un dossier, compter le nombre de lignes de chaque fichier, puis trier les résultats, on peut utiliser la commande suivante :

```
$ find . -name "*.txt" | xargs wc -l | sort -n
```

Ce pipeline utilise les commandes `find` pour rechercher les fichiers, `xargs` pour passer les fichiers trouvés à `wc` qui compte les lignes, et `sort` pour trier les résultats. Cette approche modulaire montre comment des programmes spécialisés peuvent être chaînés pour réaliser des tâches complexes [31 – chap. 6].

Enfin, Unix encourage l'utilisation de standards ouverts. La réécriture en langage C a permis une portabilité accrue du système sur diverses architectures matérielles, renforçant ainsi son adoption et son évolution constante.

1.2.4 Interopérabilité d'Unix

Depuis 1988, Unix adhère à la norme IEEE 1003, connue sous le nom de POSIX (*portable operating system interface*), et qui définit les appels système que toute implémentation doit fournir. Si l'interface est standardisée, son implémentation ne l'est pas, laissant une certaine flexibilité aux développeurs.

La norme POSIX a évolué au fil du temps avec les améliorations apportées au système. En particulier, les étapes notables incluent :

- POSIX.1 (1988) : couvre les processus, les signaux et les tubes.
- POSIX.1b (1993) : introduit des améliorations telles que l'ordonnancement, les sémaphores et la mémoire partagée.
- POSIX.1c (1995) : inclue la prise en charge des processus légers, également connus sous le nom de *threads*.

Les standards POSIX continuent d'évoluer pour répondre aux besoins changeants des systèmes modernes. Les mises à jour récentes incluent le support pour les systèmes multicoeurs, une sécurité accrue et des interfaces réseau améliorées. Ces améliorations assurent que les applications développées sur une plateforme Unix peuvent être portées vers une autre avec peu ou pas de modifications, renforçant ainsi l'interopérabilité et la portabilité des applications Unix à travers diverses distributions et versions.

Cependant, Stevens et Rago soulignent la difficulté qu'ont la plupart des implémentations Unix à respecter totalement la norme [34].

Défis actuels et solutions

L'un des principaux défis en matière d'interopérabilité est la gestion des variations entre les diverses distributions Unix/Linux. Cette diversité peut compliquer le déploiement et la gestion des applications à grande échelle.

Pour surmonter ces obstacles, des outils tels que Docker et Kubernetes ont été développés. Ces technologies permettent d'abstraire les différences entre les environnements, facilitant ainsi le déploiement cohérent et reproductible des applications sur différentes plateformes.

Docker, par exemple, utilise des conteneurs pour encapsuler les applications avec toutes leurs dépendances, assurant ainsi qu'elles fonctionnent de la même manière quel que soit l'environnement sous-jacent. Kubernetes, quant à lui, orchestre ces conteneurs, gérant leur déploiement, leur mise à l'échelle et leur maintenance, ce qui permet une gestion plus efficace et automatisée des applications conteneurisées [32 – chap. 18].

Parallèlement, des initiatives comme la « Linux Standard Base » (LSB) visent à harmoniser les aspects fondamentaux des systèmes Linux. En définissant des standards communs, LSB cherche à améliorer l'interopérabilité entre les différentes distributions Linux, réduisant ainsi les frictions pour les développeurs et les utilisateurs finaux. Ces efforts de standardisation contribuent à créer un écosystème Linux plus cohérent qui facilite le déploiement de solutions à travers diverses distributions [39].

1.3 Rôle de la programmation système

La programmation système sous Unix est d'une importance capitale. Elle permet en effet une gestion fine des ressources matérielles et logicielles, et offre aux développeurs un contrôle précis sur l'utilisation et l'allocation de ces ressources. Cette capacité repose sur une compréhension approfondie de son fonctionnement interne, permettant de concevoir des systèmes plus rapides, efficaces et adaptés aux besoins spécifiques des applications.

En outre, la programmation système facilite l'automatisation des tâches répétitives et la création d'outils personnalisés qui augmentent la productivité tout en réduisant le risque d'erreurs humaines. Une maîtrise des concepts fondamentaux de la programmation système permet aux développeurs de concevoir des applications qui requièrent des performances élevées, une fiabilité accrue et une interaction directe avec le matériel.

Les mécanismes abordés dans ce cours permettront de créer des applications modulaires et robustes, capables de s'adapter aux exigences évolutives des environnements informatiques modernes. Cette expertise est essentielle pour les développeurs qui souhaitent exploiter pleinement les capacités offertes par les systèmes Unix.

1.3.1 Tendances actuelles et futures

La montée en puissance des microservices et des conteneurs a transformé la gestion des ressources et des processus dans les systèmes modernes. Ces approches innovantes offrent des avantages considérables en matière de modularité, de déploiement, et d'adaptabilité à des charges de travail variables. Les microservices permettent de décomposer les applications en composants indépendants, ce qui facilite le développement, la maintenance et le déploiement. Les conteneurs, quant à eux, offrent un environnement léger et portable pour exécuter les applications, assurant une cohérence entre les différents environnements de développement, de test et de production.

Néanmoins, pour tirer pleinement parti de ces technologies contemporaines, une compréhension approfondie des bases de la programmation système demeure indispensable. Les développeurs doivent maîtriser les concepts de bas niveau pour optimiser les performances et garantir la sécurité des applications dans des environnements de plus en plus complexes. La connaissance des mécanismes sous-jacents, tels que la gestion de la mémoire, les appels système, et la synchronisation des processus, permet de concevoir des systèmes plus robustes et efficaces.

Les défis liés à la mise à l'échelle et à la résilience des systèmes distribués exigent une expertise en programmation système pour résoudre des problèmes tels que la tolérance aux pannes et la gestion des ressources à grande échelle. Les tendances actuelles montrent que cette expertise reste cruciale pour l'innovation et l'efficacité dans le développement de solutions technologiques avancées.

1.3.2 La documentation avant tout

Avant de nous aventurer dans les profondeurs de la programmation système, soulignons l'importance de la documentation sous Unix. Les manuels et les guides d'utilisation constituent en effet des ressources essentielles pour les développeurs et les administrateurs système, facilitant le développement et la maintenance des systèmes Unix [31 – chap. 5].

La commande **man** sous Unix permet d'afficher les pages de manuel, fournissant une documentation exhaustive sur diverses sections. Les trois premières sections sont particulièrement utiles :

- `man 1 <commande>` : documentation des commandes de base Unix (section 1).
- `man 2 <appel système>` : documentation des appels système (section 2).
- `man 3 <fonction>` : documentation de la bibliothèque C standard (section 3).

Il est important de préciser la section lors de l'utilisation de la commande `man`, car les sections sont parcourues séquentiellement jusqu'à trouver la première occurrence de l'élément recherché. Cette recherche séquentielle peut entraîner des confusions, notamment lorsque des éléments partagent le même nom dans différentes sections. Par exemple, la commande `man printf` affichera la documentation de la commande Unix associée, et non celle de la fonction de la bibliothèque standard du langage C. Pour éviter toute confusion, le numéro de section est généralement indiqué entre parenthèses à côté de l'élément, ce qui permet de vérifier et, le cas échéant, de corriger le résultat.

Voici un exemple d'affichage pour l'appel « `man 1 cp` » (ou plus simplement « `man cp` ») :

```
CP(1)                   Commandes                   CP(1)

NOM
     cp - Copier des fichiers et des répertoires

SYNOPSIS
     cp [OPTION]... [-T] SOURCE CIBLE
     cp [OPTION] ... SOURCE ... RÉPERTOIRE
     cp [OPTION] ... -t RÉPERTOIRE SOURCE ...

DESCRIPTION
     Copier la SOURCE vers la CIBLE, ou plusieurs SOURCEs vers le RÉPERTOIRE.
```

En résumé, la documentation sous Unix, accessible à l'aide de la commande `man`, est indispensable pour comprendre et utiliser efficacement les commandes, les appels système, et les fonctions de la bibliothèque standard.

1.3.3 Aperçu des langages utilisés en programmation système

Divers langages de programmation sont utilisés en programmation système, chacun offrant des avantages et des inconvénients spécifiques.

Le langage C est largement adopté en raison de ses hautes performances et de son accès bas-niveau au matériel, ce qui le rend idéal pour écrire les noyaux Unix ainsi que de nombreux utilitaires système [19,38]. Il permet une manipulation directe de la mémoire, conférant ainsi un contrôle précis sur les ressources matérielles. Toutefois, cette puissance implique une gestion manuelle de la mémoire, ce qui peut introduire des erreurs difficiles à déboguer.

Le langage C++ ajoute à C le support pour la programmation orientée objet et dispose de bibliothèques riches tout en maintenant des performances élevées. Cependant, la complexité accrue du langage et les défis associés à la

gestion de la mémoire peuvent constituer des obstacles pour les développeurs, et nécessite une maîtrise avancée pour éviter les pièges courants.

Les scripts *shell* sont particulièrement appréciés pour leur simplicité dans l'automatisation des tâches et leur intégration native avec l'environnement Unix [31 – chap. 24 à 36,6,13]. Ils permettent de créer rapidement des scripts pour automatiser des processus répétitifs. Cependant, ils sont moins performants pour des tâches complexes et leur syntaxe varie d'un interpréteur de commandes à un autre, ce qui peut limiter leur portabilité.

Python et Perl sont souvent utilisés pour leur syntaxe simple et leur large support de bibliothèques, rendant ces langages idéaux pour le prototypage rapide et les tâches d'administration système. Toutefois, ils sont généralement moins performants que C ou C++, et la gestion automatique de la mémoire peut entraîner des inefficacités dans certaines situations.

Le langage assembleur permet un contrôle total sur le matériel, et offre ainsi la possibilité d'optimiser les performances. Cependant, la complexité liée aux différentes variantes du langage et la difficulté de maintenance du code en font un choix moins courant. Dans le noyau Linux par exemple, le code assembleur représente environ 2 % du code total selon l'outil SLOCCount [41].

Ada se distingue par sa fiabilité et sa robustesse, ce qui le rend particulièrement adapté aux systèmes embarqués et critiques, tels que ceux utilisés dans l'aérospatiale, la défense et les systèmes ferroviaires. Son utilisation est cependant moins répandue, et il présente une courbe d'apprentissage plutôt élevée en raison de sa syntaxe stricte et de ses concepts avancés.

Enfin, des langages plus modernes tels que Rust, Go et D offrent des avantages spécifiques : Rust se distingue par sa sécurité mémoire, Go par ses performances élevées et sa simplicité syntaxique, et D combine la puissance du C++ avec une syntaxe plus moderne. Bien que leurs communautés et écosystèmes soient encore en développement, ces langages gagnent en popularité et pourraient jouer un rôle important dans l'avenir de la programmation système.

1.4 Défis en programmation système

Les défis émergents en programmation système englobent la sécurité dans les environnements multicoeurs et la gestion des performances dans les architectures distribuées. Pour faire face à ces enjeux, des solutions novatrices, telles que des techniques de sécurité avancées et des optimisations spécifiques aux architectures multicoeurs, sont en cours de développement.

Un aspect important de la sécurité moderne est l'utilisation des enclaves sécurisées. Ces enclaves permettent l'exécution de code sensible dans un environnement isolé, assurant ainsi la protection des données contre les attaques, même si le reste du système est compromis. Cette isolation fournit un niveau de sécurité supplémentaire, indispensable dans un contexte où les menaces deviennent de plus en plus sophistiquées.

Pour ce qui est des performances dans les environnements multicoeurs, des techniques comme le partitionnement des coeurs pour des tâches spécifiques ont été mises en place. Ce partitionnement permet d'affecter des coeurs dédiés à certaines tâches, optimisant ainsi l'utilisation des ressources et améliorant les performances globales du système. En répartissant les charges de travail de manière plus efficace, ces optimisations exploitent pleinement les capacités offertes par les architectures multicoeurs.

Ces solutions, bien que prometteuses, nécessitent une compréhension approfondie des mécanismes de bas niveau pour être mises en oeuvre efficacement. La maîtrise de ces techniques avancées est essentielle pour relever les défis posés par les systèmes modernes et garantir la sécurité ainsi que les performances de leurs applications.

2 Processus

2.1 Qu'est-ce qu'un processus ?

Un **processus** est une instance active d'un programme en cours d'exécution sur un processeur. Silberschatz le décrit comme une entité ayant un commencement, un déroulement continu et une fin. Chaque instant de son exécution est marqué par un **état** spécifique [32 – chap. 3]. Par exemple, exécuter la commande `ls` dans un terminal Unix engendre un processus. Ce processus commence avec sa création, exécute les instructions du programme, et se termine une fois que les résultats sont affichés à l'utilisateur.

La structure d'un processus comprend plusieurs segments de mémoire essentiels : le segment de code où réside le code exécutable, les segments de données qui stockent les variables initialisées et non initialisées, et le segment de pile utilisé pour gérer les appels de fonctions et les variables locales [34]. De plus, un espace de mémoire appelé « tas » est alloué pour les besoins d'allocations dynamiques de mémoire pendant l'exécution du processus.

Contrairement à un *thread*, une unité d'exécution plus légère partageant le même espace mémoire avec d'autres *threads* au sein d'un même processus, un processus possède son propre espace mémoire (voir le chapitre 3).

2.2 États d'un processus

La figure 2.1 montre les différents états que peut prendre un processus [32 – chap. 3].

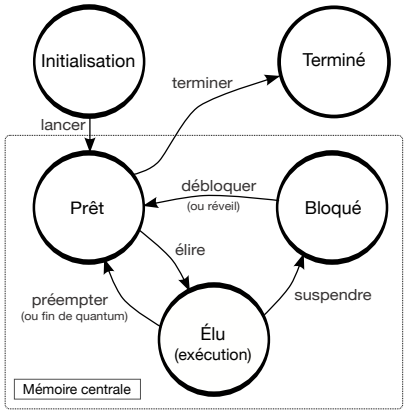


Fig. 2.1 : États d'un processus

Ces états suivent le cycle de vie typique d'un processus :

Lorsqu'un processus est créé, il entre dans l'état **initialisé** (*new*). Dans cet état, il attend que toutes les ressources nécessaires lui soient allouées. Une fois ces ressources allouées, le processus passe à l'état **prêt** (*ready* ou *runnable*). À ce stade, le processus est chargé en mémoire centrale et attend qu'un processeur ou un coeur soit disponible. Le processus prêt avec la plus haute priorité, ou selon la stratégie d'ordonnancement choisie, sera alors **élu** (*running*), signifiant qu'il s'exécute activement sur un processeur.

Si le processus nécessite une opération d'entrée-sortie ou attend un événement, il entre dans l'état **bloqué** (*blocked*). Dans cet état, le processus libère le processeur, permettant à d'autres processus de s'exécuter. Une fois l'opération d'entrée-sortie complétée ou l'événement attendu survenu, le processus retourne à l'état prêt. Enfin, après avoir terminé son exécution, le processus passe à l'état **terminé** (*terminated*), où il libère toutes ses ressources et attend que le système d'exploitation nettoie ses informations résiduelles.

Afin d'illustrer ces transitions d'état, considérons le pseudo-code suivant :

```
if processus créé then
    état = initialisé
if ressources allouées then
    état = prêt
if processeur disponible then
    état = élu
if entrée/sortie requise then
    état = bloqué
if entrée/sortie complétée then
    état = prêt
if exécution terminée then
    état = terminé
```

La terminologie des états peut légèrement varier d'un auteur à l'autre et d'un système à l'autre. Par exemple, sous Linux, les états d'un processus peuvent inclure : *running* (R) pour un processus en cours d'exécution, *sleeping* (S) pour un processus en attente d'un événement extérieur, *stopped* (T) pour un processus temporairement arrêté par un signal, et *zombie* (Z) pour un processus terminé, mais non encore nettoyé [4 – chap. 11].

2.3 Parallélisme

Dans le contexte des systèmes d'exploitation, le **parallélisme** et le **pseudo-parallélisme** décrivent la manière dont les processus s'exécutent simultanément. Par exemple, deux processus parallèles, nommés *p1* et *p2*, peuvent représenter l'exécution de deux programmes séquentiels distincts, désignés respectivement par *P1* et *P2*.

La figure 2.2 illustre l'exécution de deux processus de manière séquentielle, pseudo-parallèle sur un processeur, et parallèle sur deux processeurs (ou un processeur multicoeurs).

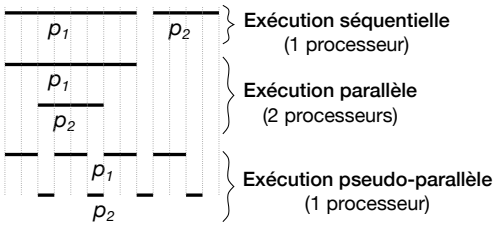


Fig. 2.2 : Exécutions séquentielle, parallèle et pseudo-parallèle.

Exemples d'applications

- **Traitement vidéo** : Une application peut encoder des vidéos plus rapidement en répartissant le travail entre plusieurs coeurs de processeur.
- **Serveur Web** : Utilise le pseudo-parallélisme afin de gérer plusieurs requêtes clients simultanément sur un seul coeur de processeur grâce au partage de temps.

Les techniques modernes, telles que le *multi-threading* matériel ou le *simultaneous multi-threading* (SMT), permettent aux processeurs, notamment les processeurs superscalaires, de gérer plusieurs *threads* en parallèle. Cela améliore l'efficacité du pseudo-parallélisme même sur des processeurs à un seul coeur, en maximisant l'utilisation des unités d'exécution disponibles [16,32,35].

2.4 Réalisation des processus

Un processus peut être envisagé comme l'association d'une **mémoire virtuelle** et d'un **processeur virtuel**. Ces ressources sont fournies par le système d'exploitation, qui alloue les ressources physiques de la machine. Le processeur est alloué par **multiplexage**, attribuant le processeur aux processus pendant des tranches de temps déterminées. Ce mécanisme sera détaillé dans la section suivante. La mémoire virtuelle sera introduite de manière approfondie au chapitre 5.

La **mémoire virtuelle** constitue une abstraction qui permet une gestion plus efficace de la **mémoire physique**. Cette dernière se réfère à la RAM installée sur la machine, tandis que la mémoire virtuelle correspond à un espace d'adressage créé par le système d'exploitation pour chaque processus.

Un multiplexage efficace est essentiel dans les environnements multicoeurs où chaque coeur peut exécuter un processus différent simultanément, optimisant l'utilisation des ressources du système et améliorant les performances globales. Par exemple, un système doté de quatre coeurs peut théoriquement quadrupler le nombre de processus ou de *threads* exécutés en parallèle par rapport à un système monocoeur, réduisant ainsi de manière significative les temps d'attente pour l'exécution des processus. Ce principe sera illustré plus en détail lors de la séance de travaux pratiques dédiée aux *threads*.

2.5 Ordonnancement des processus

Le processeur est alloué aux processus prêts à s'exécuter par tranches de temps successives, appelées *quanta*. La durée d'un *quantum* est typiquement de l'ordre de 1 à 100 millisecondes, mais elle peut être ajustée dynamiquement par le système d'exploitation en fonction des charges de travail. Sur les processeurs modernes, fonctionnant souvent à des fréquences de plusieurs GHz, un quantum permet l'exécution de milliards d'instructions.

La commutation entre les processus, également appelée **commutation de contexte** (*context switching*), est déclenchée par des interruptions d'horloge, comme le représente la figure 2.3. Cette commutation est gérée par un composant du noyau appelé **ordonnanceur** (*scheduler*), qui utilise des algorithmes avancés pour optimiser l'utilisation du processeur.

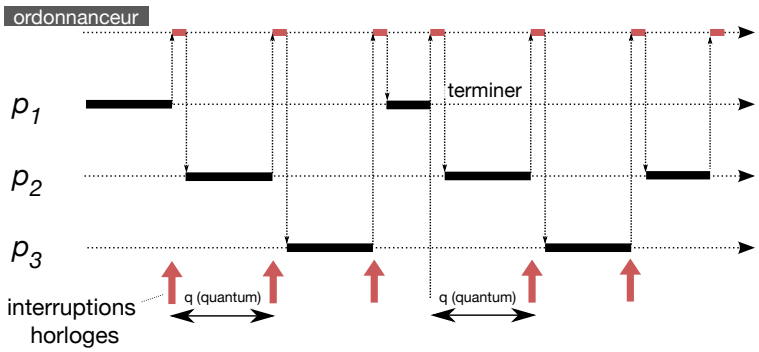


Fig. 2.3 : Ordonnancement des processus

Contexte des processus

Le **contexte** d'un processus est l'ensemble des paramètres qu'il est nécessaire de mémoriser lorsque le processus passe de l'état « élu » à l'état « prêt » ou à l'état « bloqué ». Parmi ces paramètres, on trouve :

- son numéro d'identification unique attribué par le système à sa création (**PID** *process identifier*) ;
- le numéro d'identification du processus parent qui lui a donné naissance (**PPID**) ;
- l'état des registres (compteur ordinal, registre d'état, registres généraux) ;
- registres décrivant les espaces virtuel et physique d'implantation ;
- pointeurs de piles ;
- ressources auxquelles il accède (ex. : fichiers ouverts), valeur d'horloge, etc.

Ce contexte est mémorisé dans une structure appelée **bloc de contrôle du processus** ou **PCB** (*process control block* en anglais) qui est mise à jour lors d'une commutation de processus. Le système d'exploitation conserve l'ensemble des PCB de chaque processus dans une table des processus comme l'indique la figure 2.4. La taille de cette table est fixée par le système[1]. Le **PCB** est décomposé en deux sous-structures appelées :

- **zone proc** définie dans l'en-tête `sys/procfs.h` reste en permanence en mémoire ;
- **zone u** définie dans l'en-tête `sys/user.h` et peut être *swappée*[2] lorsque le processus n'est pas dans l'état "élu".

[1] Une table des PCB pleine ne permet plus au système de lancer de nouveaux processus, ce qui peut « geler » le système. Une *bombe fork* consiste à créer un grand nombre de processus jusqu'à ce que le nombre maximal de processus soit atteint.
[2] Stockée temporairement sur le disque (voir le chapitre sur la mémoire).

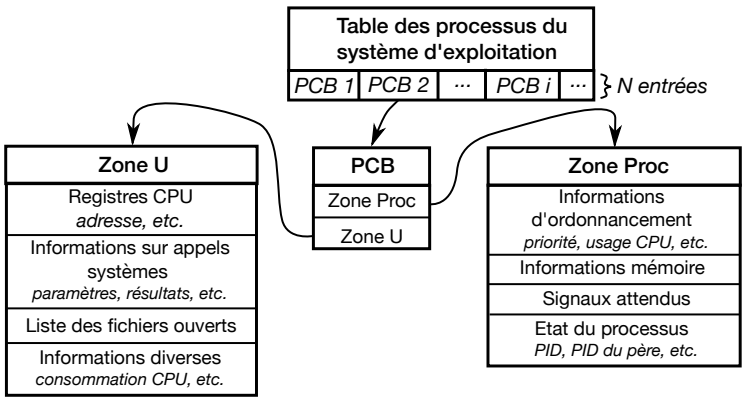


Fig. 2.4 : Table des blocs de contrôle des processus.

Commutation de contexte

L'ordonnanceur gère les files d'attente des processus prêts et bloqués comme le montre la figure 2.5.

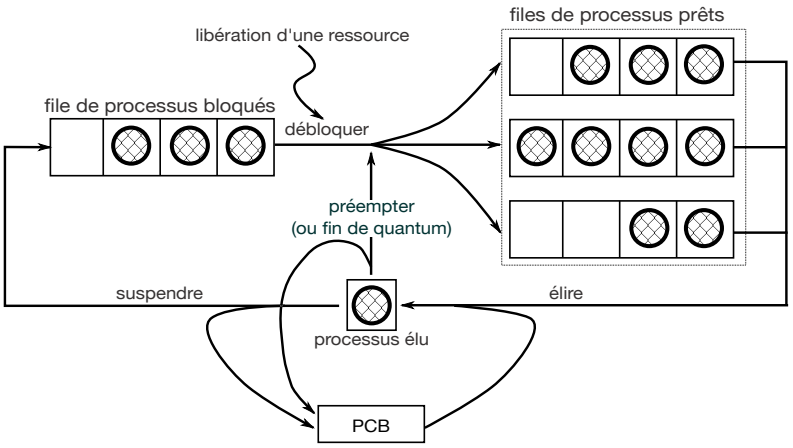


Fig. 2.5 : Ordonnancement et contexte.

Le mécanisme d'allocation des processus nécessite diverses opérations. Il est d'abord nécessaire de déterminer le prochain processus élu suivant la stratégie adoptée, qui peut être basée sur l'ordre d'arrivée, les priorités, ou les durées (fixes ou dynamiques, avec ou sans préemption), en utilisant des **files d'attente** (FIFO). Une fois ce processus déterminé, l'allocation implique la sauvegarde du contexte du processus élu actuel, incluant son PID, son PPID, les états du compteur ordinal et des registres d'état ainsi que des registres décrivant les espaces virtuel et physique d'implantation, les pointeurs de piles, et les différentes ressources auxquelles il accédait. Ensuite, le contexte du nouveau processus élu est restauré, ce qui comprend la restauration des structures de données définissant la mémoire virtuelle ainsi que le chargement des registres, puis du mot d'état du processeur, ce qui lance l'exécution du nouveau processus élu.

2.6 Stratégies d'ordonnancement

L'ordonnancement des processus est une tâche essentielle pour une gestion efficace et équitable du processeur. Les stratégies d'ordonnancement se divisent en deux grandes catégories : sans préemption et avec préemption.

Stratégies sans préemption

Les stratégies sans préemption n'interrompent pas un processus en cours d'exécution, mais attendent qu'il termine ou qu'il soit bloqué. Les stratégies de ce type les plus fréquemment rencontrées sont les suivantes :

1. **FCFS** (*first come first served*) Cette stratégie exécute les processus en fonction de leur ordre d'arrivée. Simple à mettre en oeuvre, elle peut conduire à une performance médiocre si un processus long précède des processus courts.
2. **SJF** (*shortest job first*) Cette stratégie sélectionne le processus ayant la plus courte durée estimé quant à son exécution. Elle est optimale pour minimiser le temps d'attente, mais il est difficile de prédire avec précision les durées.
3. Ordonnancement par priorité Dans cette stratégie, le processeur est attribué au processus qui a la plus haute priorité. C'est une stratégie efficace pour les processus critiques, mais susceptible de causer de la famine pour ceux de faible priorité.

Stratégies avec préemption

Contrairement aux précédentes, ces stratégies peuvent interrompre un processus pour en exécuter un autre, améliorant la réactivité et l'équité. Les stratégies les plus couremment rencontrées sont les suivantes :

1. **SRTF** (*shortest remaining time first*) C'est une variante préemptive de la stratégie SJF. Elle réduit encore plus le temps d'attente moyen, mais requiert la connaissance du temps d'exécution restant.
2. Ordonnancement circulaire ou *round robin* (**RR**) Cette stratégie consiste à assigner un *quantum* de temps fixe à chaque processus. Elle est particulièrement équitable et améliore le temps de réponse moyen, mais sa performance dépend de la valeur du *quantum* (un *quantum* trop court entraîne une surcharge de commutations de contexte, tandis qu'un *quantum* trop long risque de réduire le bénéfice de la préemption).
3. Ordonnancement à priorité avec préemption Cette variante interrompt un processus si un autre de plus haute priorité est prêt à s'exécuter, garantissant que les tâches critiques prennent la priorité.

Stratégies avancées

Les stratégies suivantes sont des stratégies hybrides ou avancées qui utilisent plusieurs files d'attente pour une gestion plus fine.

1. **MLQS** (*multilevel queue scheduling*) : Cette stratégie répartit les processus dans différentes files selon leurs caractéristiques (priorité, type de tâche, etc.), chaque file ayant son propre algorithme d'ordonnancement.
2. **MLFQS** (*multilevel feedback queue scheduling*) : Cette stratégie permet aux processus de se déplacer entre les files selon leur comportement et leur temps d'exécution. Bien que fortement adaptative, elle reste très complexe à paramétrer de manière optimale (nombre de files, critères de promotion/dégradation).
3. **CFS** (*completely fair scheduler*) : Stratégie utilisée par défaut sur Linux depuis 2007, elle répartit équitablement le temps processeur entre tous les processus.

Comparaison et application des stratégies

Le tableau suivant résume les stratégies d'ordonnancement précédentes.

Stratégie	Principe	Avantages	Inconvénients
FCFS	Ordre d'arrivée	Simple à implémenter	Risque de blocage par des processus longs, mauvaise performance en termes de temps de réponse moyen
SJF	Durée estimée de l'exécution	Minimise le temps d'attente moyen	Difficile à prédire, risque de famine pour les processus longs
À priorité	Priorité des processus	Priorise les processus critiques ou urgents	Risque de famine pour les processus de faible priorité, nécessite une gestion des priorités
SRTF	Temps d'exécution le plus court restant	Réduit encore plus le temps d'attente moyen	Complexe à implémenter, nécessite le temps restant
RR	Tranches de temps fixes	Équitable, améliore le temps de réponse moyen	Performance dépendante de la durée du quantum
MLQS	Plusieurs files d'attente	Flexibilité pour gérer différents types de processus	Complexité accrue de gestion des files
MLFQS	Déplacement entre files d'attente	Adaptatif, gère bien les processus longs	Difficile à paramétrer
CFS	Partage équitable du temps processeur	Équité garantie, bonne performance globale	Complexe à implémenter et comprendre

Le choix de la stratégie d'ordonnancement dépend fortement du contexte d'utilisation et des objectifs de performance. Par exemple, pour des systèmes opérant sur des ordinateurs de bureau, des algorithmes avec préemption comme l'ordonnancement circulaire ou l'ordonnancement CFS sont préférés de manière à garantir des temps de réponse rapides. À l'inverse, dans le cas d'un serveur Web, l'algorithme SRTF est privilégié pour que les requêtes courtes soient servies rapidement, améliorant ainsi le temps de réponse global du serveur.

Sur de nombreux systèmes embarqués, des stratégies à priorité stricte sont utilisées afin de garantir les délais critiques. Cependant, dans les systèmes temps réel où il est crucial que chaque tâche reçoive une attention équitable dans un délai prévisible, un ordonnancement circulaire peut être préféré. Enfin, sur des systèmes tels que des superordinateurs, des stratégies telles que FCFS ou SJF sont employées pour optimiser l'utilisation des ressources.

En conclusion, il n'existe pas de solution unique pour l'ordonnancement des processus. Les systèmes d'exploitation modernes implémentent souvent plusieurs algorithmes et les combinent pour répondre aux besoins variés des utilisateurs et des applications. Pour approfondir le sujet, le lecteur se reportera à l'ouvrage de Silberschatz [32] dans le cas des systèmes d'exploitation courants et à Liu pour les systèmes temps réels [23].

2.7 Blocage des processus

Un processus qui passe dans l'état « bloqué » suite à une suspension (accès à une entrée/sortie ou appel à `sleep()`, `wait()`, `pause()`, etc.) doit libérer le processeur. Ce dernier pourra lui être réalloué à la fin de sa période de blo-

cage, souvent indiquée par une interruption signalant la fin de l'opération d'entrée/sortie ou un autre événement de reprise, comme le montre la figure 2.6.

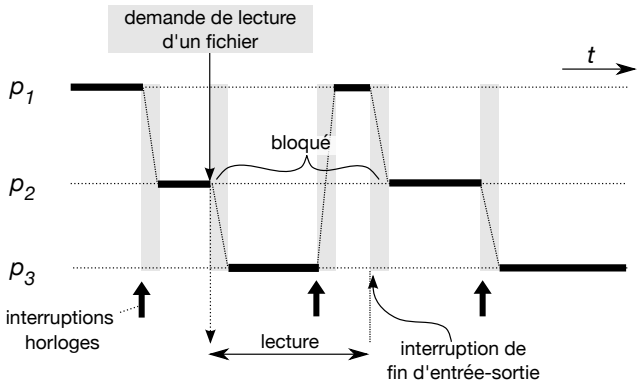


Fig. 2.6 : Blocage d'un processus

Exemple de code

```c
#include <unistd.h> /* for sleep() */
#include <stdio.h>  /* for printf() */
#include <stdlib.h> /* for EXIT_SUCCESS */

int main(void) {
    printf("The process is running...\n");
    sleep(5);  /* The process is blocked for 5 sec. */
    printf("The process has been woke up after 5 seconds.\n");

    return EXIT_SUCCESS;
}
```

Lors de l'exécution de ce programme, le processus affiche un message indiquant qu'il est en cours d'exécution, se place en état de blocage pendant cinq secondes, puis affiche un message indiquant son réveil avant de se terminer.

Le blocage d'un processus est une méthode courante pour gérer les opérations d'entrée/sortie, car ces opérations sont généralement beaucoup plus lentes que les opérations du processeur. En bloquant un processus pendant qu'il attend une opération d'entrée/sortie, le système d'exploitation peut allouer le processeur à d'autres processus, augmentant ainsi l'efficacité globale du système.

Lorsque l'événement attendu survient, une interruption matérielle ou logicielle signale au système d'exploitation que le processus bloqué peut être réveillé. Le processus est alors replacé dans la file d'attente des processus prêts à s'exécuter, et il recevra de nouveau du temps processeur lorsque son tour viendra, en fonction de la stratégie d'ordonnancement utilisée.

2.8 Processus sous Unix

Lorsqu'un processus est créé, le système d'exploitation lui attribue un identifiant unique appelé **PID** (*process identifier*). Ce PID permet d'identifier et de gérer le processus tout au long de son cycle de vie.

Les commandes Unix comme `ps`, `top` et `htop` permettent de lister les processus en cours d'exécution et d'afficher leurs PIDs, ainsi que d'autres informations pertinentes sur l'état et les ressources utilisées par chaque processus.

Par exemple, `ps` affiche la liste des processus actuels avec leurs PIDs, tandis que `top` et `htop` fournissent une vue dynamique de l'activité du processeur et des processus actifs.

En plus des commandes, des appels système comme `getpid()` et `getppid()` sont disponibles pour récupérer le PID du processus en cours d'exécution et le PID de son processus parent, respectivement. Voici les prototypes de ces appels système :

```
#include <unistd.h>
#include <sys/types.h> /* for pid_t */

pid_t getpid(void);
pid_t getppid(void);
```

Ces outils et appels système permettent aux utilisateurs et aux développeurs de surveiller et de contrôler les processus de manière efficace, assurant ainsi une gestion optimale des ressources et la stabilité du système.

2.9 Environnement d'un processus

Un processus sous Unix dispose d'un ensemble de variables qui constituent son environnement. Ces variables jouent un rôle important en facilitant l'utilisation du système et en personnalisant différents aspects de son fonctionnement.

D'une part, elles simplifient la tâche de l'utilisateur en évitant la redéfinition constante du contexte du processus, comme le nom de l'utilisateur, celui de la machine, ou encore le terminal par défaut. D'autre part, elles permettent de personnaliser des éléments tels que le chemin de recherche des fichiers (`PATH`), le dossier de l'utilisateur (`HOME`), ou l'interpréteur de commandes utilisé (`SHELL`).

Certaines variables d'environnement sont prédéfinies par le système, mais l'utilisateur peut les modifier ou en créer de nouvelles de manière à répondre à des besoins spécifiques.

À retenir suivant l'interpréteur de commande

Commande tcsh	**Commande bash**	**Action**
`setenv`	`printenv`	affiche l'environnement courant
`setenv VAR <val>`	`export VAR=<val>`	attribue `<val>` à la variable `VAR`
`echo $VAR`	`echo $VAR`	affiche la valeur courante de `VAR`

Par exemple sous Bash, la commande « `export PATH=$PATH:"/home/mon_identifiant"` » définit les dossiers de recherche des commandes et applications. On peut aussi modifier les variables d'environnement par programme.

Scripts de modification d'environnement

Le script suivant crée et affiche une nouvelle variable d'environnement :

```
#!/bin/bash
# A bash script to define a new environment variable
export MY_VAR="My value"
echo "The value of MY_VAR is: $MY_VAR"
```

Le script ci-dessous modifie la variable d'environnement `PATH` :

```
#!/bin/bash
# A script to add a new folder to PATH
export PATH=$PATH:/home/my_new_folder/bin
echo "PATH is now: $PATH"
```

En C, il est aussi possible de manipuler les variables d'environnement à l'aide des fonctions `setenv()` qui permet de définir une variable d'environnement, et `getenv()` qui permet de récupérer sa valeur. Ce mécanisme est utile si un processus doit adapter son comportement en fonction de l'environnement dans lequel il s'exécute.

2.10 Vie et mort des processus

Un processus Unix a généralement un début et une fin bien définis.

Début d'un processus

Le processus peut être initié de plusieurs façons. Depuis l'interpréteur de commandes, chaque commande exécutée crée un nouveau processus. Par exemple, en entrant `prog1` suivi d'un appui sur la touche Enter, un processus est créé pour exécuter le programme `prog1`. Il est également possible de créer des processus pour exécuter des commandes en (pseudo-)parallèle, comme dans « `prog1 & prog2 &` » qui lance deux processus pour exécuter respectivement `prog1` et `prog2`, ou encore « `prog1 & prog1 &` » qui crée deux instances (pseudo-)parallèles de `prog1`.

Au niveau des appels système, la création d'un processus se fait à l'aide de l'appel `fork()`. Cet appel crée un nouveau processus en dupliquant le processus appelant.

Fin d'un processus

La fin d'un processus peut se produire de différentes manières. Un processus peut s'auto-détruire en appelant `exit()` ou encore `return` dans la fonction `main()`. Le prototype de la fonction `exit()` est le suivant :

```
#include <stdlib.h>
void exit(int status);
```

Le paramètre `status` est un code de fin retourné au système. Par convention, une valeur de `0` indique une fin normale, et l'en-tête « `stdlib.h` » propose la macro `EXIT_SUCCESS` pour une meilleure lisibilité. Toute autre valeur signale une erreur, et la macro `EXIT_FAILURE`, qui vaut 1, peut être utilisée pour indiquer une fin anormale[3].

Un processus peut également être détruit par un autre processus. Cela peut se faire à l'aide de l'appel système `kill()` (voir figure 2.7) :

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

[3] Vous pouvez consulter l'en-tête « `errno.h` » pour plus d'informations sur les codes d'erreur retournés.

De même, la commande `kill` depuis l'interpréteur de commandes, ou encore une combinaison de touches comme Ctrl + C, permet de terminer un processus. Il est même possible d'utiliser la commande `xkill` pour fermer une fenêtre graphique.

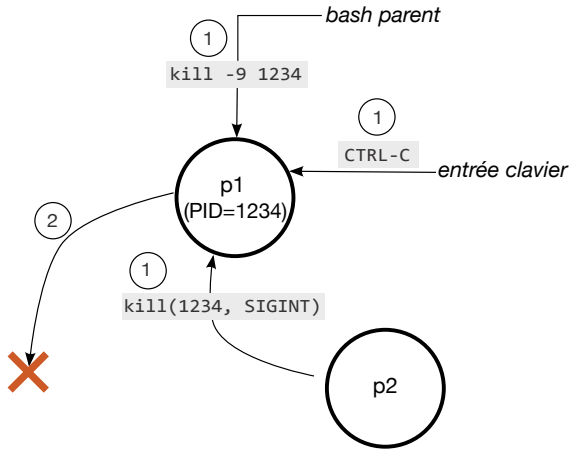


Fig. 2.7 : Terminaison d'un processus

2.11 Création des processus sous Unix

L'appel système `fork()` permet de créer un processus à partir d'un autre. En réalité il ne s'agit pas d'une création *ex nihilo*, mais plutôt du clonage en mémoire du processus courant. Le prototype de `fork()` est déclaré dans l'entête « `unistd.h` » et son type de retour est un entier.

```
#include <unistd.h>
#include <sys/types.h> /* for pid_t */
pid_t fork(void);
```

Lorsque `fork()` réussit, il crée un « processus enfant ». En cas d'échec, l'appel retourne -1, ce qui peut se produire, par exemple, si la table des blocs de contrôle des processus est pleine. Les processus parent et enfant se distinguent par le résultat retourné par `fork()`. Une valeur retournée de 0 indique qu'il s'agit du processus enfant, tandis qu'une valeur strictement positive identifie le processus parent, cette valeur étant le PID du processus enfant.

2.11.1 Un programme, mais deux processus

Le programme suivant va créer deux processus : un processus parent, puis un processus enfant. Les deux vont ensuite afficher leurs PIDs respectifs.

```
#include <unistd.h>    /* for fork() */
#include <stdio.h>     /* for printf() and perror() */
#include <stdlib.h>    /* for EXIT_SUCCESS and exit() */
#include <sys/types.h> /* for pid_t */
#include <sys/wait.h>  /* for wait() */

void handle_fatal_error_and_exit(const char *msg) {
    perror(msg);
    exit(EXIT_FAILURE);
}
```

```
void manage_parent(pid_t child_pid) {
    printf("Parent process (PID %d)\n", getpid());
    printf("My child's PID is %d\n", child_pid);
    printf("Instructions of parent process...\n");
    wait(NULL); /* We will see later why this instruction is important! */
}

void manage_child() {
    printf("Child process (PID %d)\n", getpid());
    printf("My parent's PID is %d\n", getppid());
    printf("Instructions of child process...\n");
}

int main(void) {
    pid_t pid;

    pid = fork();
    if (pid < 0) {
        handle_fatal_error_and_exit("Error using fork().\n");
    }
    if (pid > 0) {
        manage_parent(pid);
    } else {
        manage_child();
    }

    return EXIT_SUCCESS;
}
```

La sortie de ce programme est la suivante :

```
Parent process (PID 9014)
My child's PID is 9015
Instructions of parent process...
Child process (PID 9015)
My parent's PID is 9014
Instructions of child process...
```

2.11.2 Deux mémoires virtuelles et deux jeux de données distincts

Dans le programme qui suit, nous ajoutons un paramètre aux fonctions de gestion de nos processus parent et enfant. La variable `own_variable` est déclarée et définie dans la fonction principale, puis elle est passée par l'intermédiaire d'un pointeur aux deux fonctions qui vont la modifier.

```
void handle_fatal_error_and_exit(const char *msg) {
    perror(msg);
    exit(EXIT_FAILURE);
}

void manage_parent(int32_t *parameter) {
    printf("Parent process (PID %d)\n", getpid());
    printf("Parent modifies its own variable...\n");
```

```
    *parameter = 10;
    wait(NULL);
}

void manage_child(int32_t *parameter) {
    printf("Child process (PID %d)\n", getpid());
    printf("Child modifies its own variable...\n");
    *parameter = 20;
}

int main(void) {
    pid_t pid;
    int own_variable = 0;

    pid = fork();
    if (pid < 0) {
        handle_fatal_error_and_exit("Error using fork().\n");
    }
    if (pid > 0) {
        manage_parent(&own_variable);
    } else {
        manage_child(&own_variable);
    }
    printf("\nPID %d has its own variable equals to: %d\n", getpid(), own_variable);

    return EXIT_SUCCESS;
}
```

La sortie de ce programme est la suivante :

```
Parent process (PID 2848)
Parent modifies its own variable...
Child process (PID 2849)
Child modifies its own variable...

PID 2849 has its own variable equals to: 20
PID 2848 has its own variable equals to: 10
```

Chacun des processus a donc modifié le contenu de sa « propre variable ». Le segment de données initial du processus parent qui la contient ayant été cloné, il est dissocié de celui du processus enfant[4].

Il est également important de noter que les instructions situées après le bloc « `if-else` » sont exécutées à la fois par le processus parent et par le processus enfant. Cela illustre que bien que les processus partagent le même code source, mais que leurs exécutions et leurs environnements sont distincts une fois `fork()` appelé.

2.11.3 Un programme, deux processus et une mise *en sommeil*

Dans le programme suivant, nous utilisons la fonction `sleep()` déclarée dans l'en-tête « `unistd.h` », qui permet de mettre un processus dans l'état « bloqué » pendant un certain nombre de secondes. Pour un ajustement plus fin du temps de blocage, il existe la fonction `usleep()`, également déclarée dans « `unistd.h` », qui permet un blocage de l'ordre de la microseconde. De plus, l'appel système `nanosleep()`, déclaré dans l'en-tête « `time.h` », permet des blocages de l'ordre de la nanoseconde en utilisant un temporisateur.

[4] Nous verrons plus loin que le mécanisme de clonage est plus complexe et qu'en réalité celui-ci n'est pas forcément immédiat.

```c
#define DURATION 5

void handle_fatal_error_and_exit(const char *msg) {
    perror(msg);
    exit(EXIT_FAILURE);
}

void manage_parent() {
    printf("Parent process (PID %d)\n", getpid());
    printf("Parent will be blocked during %d seconds...\n", DURATION);
    sleep(DURATION);
    printf("Parent has finished to sleep.\n");
    wait(NULL);
}

void manage_child() {
    printf("Child process (PID %d)\n", getpid());
    printf("Child will be blocked during %d seconds...\n", DURATION);
    sleep(DURATION);
    printf("Child has finished to sleep.\n");
}

int main(void) {
    pid_t pid;

    pid = fork();
    if (pid < 0) {
        handle_fatal_error_and_exit("Error using fork().\n");
    }
    if (pid > 0) {
        manage_parent();
    } else {
        manage_child();
    }

    return EXIT_SUCCESS;
}
```

La sortie de ce programme est la suivante :

```
Parent process (PID 2886)
Parent will be blocked during 5 seconds...
Child process (PID 2887)
Child will be blocked during 5 seconds...
# TIC TAC... TIC TAC... TIC TAC...
Child has finished to sleep.
Parent has finished to sleep.
```

Dans cet exemple, les processus parent et enfant utilisent la fonction `sleep()` pour se mettre en sommeil pendant une durée spécifiée par la macro `DURATION`, définie ici à 5 secondes. Chaque processus affiche un message avant et après la période de sommeil, montrant ainsi qu'ils entrent dans l'état « bloqué » et en sortent indépendamment l'un de l'autre.

Ce programme illustre comment un processus peut être temporairement suspendu en utilisant des fonctions de

mise en sommeil. Cette technique est couramment utilisée pour simuler des délais ou pour attendre la disponibilité de certaines ressources sans consommer inutilement du temps processeur.

2.12 Hiérarchie des processus sous Unix

Il existe un processus "primitif" créé à l'origine du système et dont le PID est **1**. Ce processus s'appelle `init` sous Linux et `launchd` sous Mac OS X. Ce processus correspond à la racine de la structure arborescente des processus comme le montre la figure 2.8.

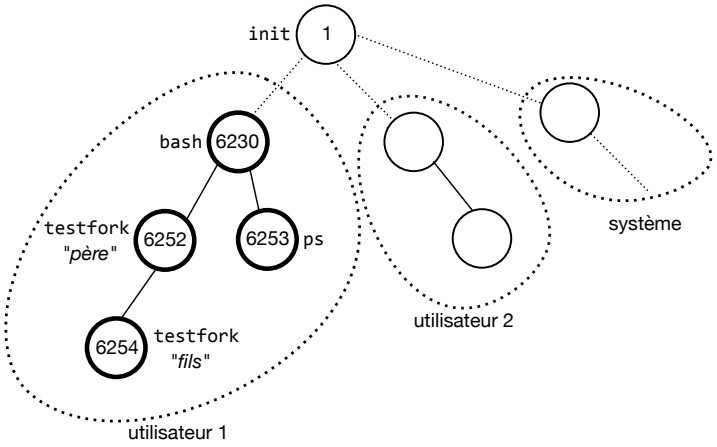


Fig. 2.8 : Hiérarchie des processus

Nous avons déjà rencontré les appels système `getpid()` et `getppid()`. Pour compléter ces informations, il est utile de mentionner `getuid()` et `getgid()`, qui permettent respectivement d'obtenir l'identifiant de l'utilisateur ayant lancé le processus et celui du groupe auquel il appartient. Leurs prototypes sont les suivants :

```
#include <sys/types.h> /* for uid_t and gid_t */
#include <unistd.h>
uid_t getuid(void);
gid_t getgid(void);
```

2.13 Quelques interactions entre processus sous Unix

Envoyer un signal à un autre processus

Nous reviendrons en détail sur la notion de signal dans la section 6.2. Sachez que la commande Unix `kill` ou l'appel système `kill()` ne permettent pas uniquement de « terminer » un processus, mais plus généralement d'envoyer divers signaux d'un processus à un autre.

Faire attendre un processus

Nous avons déjà rencontré les fonctions `sleep()` et `usleep()`, ainsi que l'appel système `nanosleep()`, qui permettent de bloquer un processus pendant un temps déterminé. Il est aussi possible de bloquer un processus de manière asynchrone jusqu'à la réception d'un signal à l'aide de la fonction `pause()`, dont le prototype est déclaré dans l'en-tête « `unistd.h` », ou encore de l'appel système `sigsuspend()`, dont le prototype se trouve dans « `signal.h` ».

Synchronisation entre un processus parent et ses enfants

Un processus enfant termine en général son exécution par un appel à la fonction `exit()` ou à `return` à la fin de la fonction `main()`. Dans les exemples de code précédents, vous aurez remarqué qu'à un moment donné, le processus parent fait appel à l'appel système `wait()`, déclaré dans l'en-tête « `sys/wait.h` ». Cet appel est bloquant et permet au processus parent d'attendre la fin de l'exécution d'un processus enfant. Tant qu'un processus parent n'a pas pris connaissance de la terminaison de son enfant, ce dernier reste, vis-à-vis du système d'exploitation, dans un état dit de **zombie**. Un processus *zombie* ne s'exécute plus, mais il consomme encore des ressources, notamment en occupant une place dans la table des blocs de contrôle des processus.

L'appel système `wait()` n'est pas le seul qui permet à un processus parent d'attendre la fin de ses processus enfants. Dans le cas où plusieurs processus enfants coexistent, il peut être utile pour le processus parent d'utiliser l'appel système `waitpid()`, qui prend comme paramètre le PID spécifique d'un des processus enfants. Les prototypes de ces deux appels système sont les suivants :

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

pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

Ces appels permettent au processus parent de synchroniser sa terminaison avec celle de ses processus enfants, assurant une gestion appropriée des ressources et évitant l'accumulation de processus *zombies* dans le système.

Exemple d'un processus et ses *n* enfants

L'exemple suivant montre un processus parent qui crée *n* processus enfants et attend leur terminaison à l'aide de `waitpid()`.

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

#define NUM_PROCESSES 8

void manage_child() {
    printf("[%d] Executing...\n", getpid());
    sleep(2);
    printf("[%d] Finished.\n");
    exit(EXIT_SUCCESS);
}

int main(void) {
```

```
    pid_t pids[NUM_PROCESSES];
    for (int i = 0; i < NUM_PROCESSES; i++) {
        if ((pids[i] = fork()) == 0) {
            manage_child()
        }
    }
    for (int i = 0; i < NUM_PROCESSES; i++) {
        waitpid(pids[i], NULL, 0);
    }
    printf("All processes completed.\n");
    return EXIT_SUCCESS;
}
```

2.14 Attention aux zombies

L'exemple ci-dessous met en oeuvre un processus parent qui omet de faire appel à `wait()` (ou `waitpid()`).

```
int main(void) {
    if (fork() != 0) {
        printf("Parent process (PID %d)\n", getpid());
        while (1) { /* infinite loop without waiting for child's ending */
            sleep(1);
        }
    } else {
        printf("Child process (PID %d)\n", getpid());
        sleep(10); /* sleep for 10 secondes */
        printf("End of child\n");
        exit(EXIT_SUCCESS);
    }
}
```

La sortie de ce programme sur un système Linux donne :

```
$ ./testfork
Parent process (PID 6885)
Child process (PID 6886)
End of child
[1]+ Stopped      ./testfork

$ ps
PID    TTY        TIME CMD
6826 pts/1    00:00:00 bash
6885 pts/1    00:00:02 testfork
6886 pts/1    00:00:00 testfork <defunct>
6895 pts/1    00:00:00 ps
```

La commande `ps` consulte la table des PCB et y trouve deux processus portant le nom ”testfork”. L'indication ”`<defunct>`” à côté du second indique que celui-ci est dans un état de *zombie*. Bien que le processus enfant ne s'exécute plus en mémoire, il en reste une « trace » dans la table des PCB.

Dans le code suivant, le processus parent fait appel à `wait()` afin d'attendre la fin de son enfant, évitant ainsi le passage à l'état *zombie*. Les macros `WIFEXITED` et `WEXITSTATUS`, définies dans l'en-tête « `wait.h` », retournent

respectivement une valeur non nulle si le processus enfant s'est terminé par un appel à `exit()` et la valeur passée à `exit()`.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void) {
    if (fork() != 0) {
        int status;
        pid_t child;

        printf("Parent process (PID %d) is waiting...\n", getpid());
        child = wait(&status);
        if (WIFEXITED(status)) {
            printf("Parent process (PID %d): Child (PID %d) has ended with code %d)\n",
                                getpid(), child, WEXITSTATUS(status));
        }
    } else {
        printf("Child process (PID %d)\n", getpid());
        sleep(5);
        printf("End of child\n");
    }
    exit(EXIT_SUCCESS);
}
```

Cette fois-ci, la sortie du programme est la suivante :

```
$ ./testfork
Parent process (PID 7158) is waiting...
Child process (PID 7159)
End of child
Parent process (PID 7158): Child (PID 7159) has ended with code 0)
$ ps
PID    TTY        TIME CMD
4781 pts/0    00:00:00 bash
7166 pts/0    00:00:00 ps
```

2.15 Processus orphelin

Un processus enfant est dit **orphelin** lorsque son processus parent se termine avant lui. Dans ce cas, le processus enfant est alors rattaché au processus primitif de PID **1**.

```
#include <unistd.h>    /* for fork(), sleep() */
#include <stdio.h>     /* for printf() */
#include <stdlib.h>    /* for EXIT_SUCCESS, exit() */
#include <sys/types.h> /* for pid_t */

void manage_parent() {
    printf("Parent process (PID %d). Dying...\n", getpid());
}
```

```
void manage_child() {
    printf("Child process (PID %d)\n", getpid());
    printf("Child blocked during 20 seconds...\n");
    sleep(20);
    printf("Child has finished to sleep.\n");
    printf("My parent PID is %d. Now I am an orphan.\n", getppid());
}

int main(void) {
    pid_t pid;

    pid = fork();
    if (pid < 0) {
        handle_fatal_error_and_exit("Error using fork().\n");
    }
    if (pid > 0) {
        manage_parent();
    } else {
        manage_child();
    }

    return EXIT_SUCCESS;
}
```

La sortie du programme est la suivante :

```
$ ./test_fork
Parent process (PID 3063). Dying...
Child process (PID 3064)
Child blocked during 20 seconds...
Child has finished to sleep.
My parent PID is 1. Now, I am an orphan.
```

Les processus **démons** (*daemons* en anglais) sont des processus orphelins qui s'exécutent en arrière-plan, sans être liés à une session utilisateur spécifique. Ils sont souvent utilisés lorsqu'il s'agit d'effectuer des tâches récurrentes ou continues, telles que la maintenance du système, la gestion des services réseau ou encore la surveillance des ressources du système.
Les démons sont caractérisés par leur capacité à s'exécuter même si aucun utilisateur ne les a activés explicitement. Ils peuvent être arrêtés, mais ils seront généralement relancés automatiquement lorsqu'un événement précis se produira (par exemple, une connexion réseau).

2.16 Exécution d'un programme spécifié

La famille d'appels système `exec()` permet de remplacer l'image d'un processus par un autre programme. Les appels système de cette famille viennent « couvrir » la mémoire virtuelle du processus par celle du nouveau programme. Ce mécanisme porte le nom d'*overlay* en anglais. Cette méthode est la plupart du temps utilisée immédiatement après la création d'un processus enfant afin de remplacer l'image du processus enfant. Les diffé-

rentes variantes d'`exec()` existent selon le mode de passage des paramètres (tableau, liste, passage de variables d'environnement).

Exemple 1

```c
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    if (fork() == 0) { /* Child process */
        execl("/bin/ls", "ls", "-al", (char *)0);
    } else { /* Parent process */
        wait(NULL);
    }

    return EXIT_SUCCESS;
}
```

Le processus enfant exécute la commande « `/bin/ls -al` » depuis le dossier courant. Le processus parent, quant à lui, se met en attente de la fin du processus enfant.

Exemple 2

```c
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    pid_t pid;

    if (fork() == 0) {
        int pid = getpid();
        printf("%d > ", pid);
        fflush(stdout);
        if (pid % 10 == 3) { /* PID that ends with 3 */
            execl("/bin/date", "date", "+%d/%m/%y", 0);
        } else {
            execl("/bin/date", "date", (char *)0);
        } else {
            wait(NULL);
        }
    }

    return EXIT_SUCCESS;
}
```

Le processus enfant exécute :

```
82102 > Mer 26 mar 2018 11:13:39 CET
82103 > 26/03/2018
82104 > Mer 26 mar 2018 11:13:39 CET
```

Et le processus parent attend la fin de l'enfant.

2.17 Lien entre fork(), exec(), wait() et exit()

La figure 2.9 montre les relations qu'il y a entre les appels système `fork()`, `exec()` et `wait()`, et la fonction `exit()`.

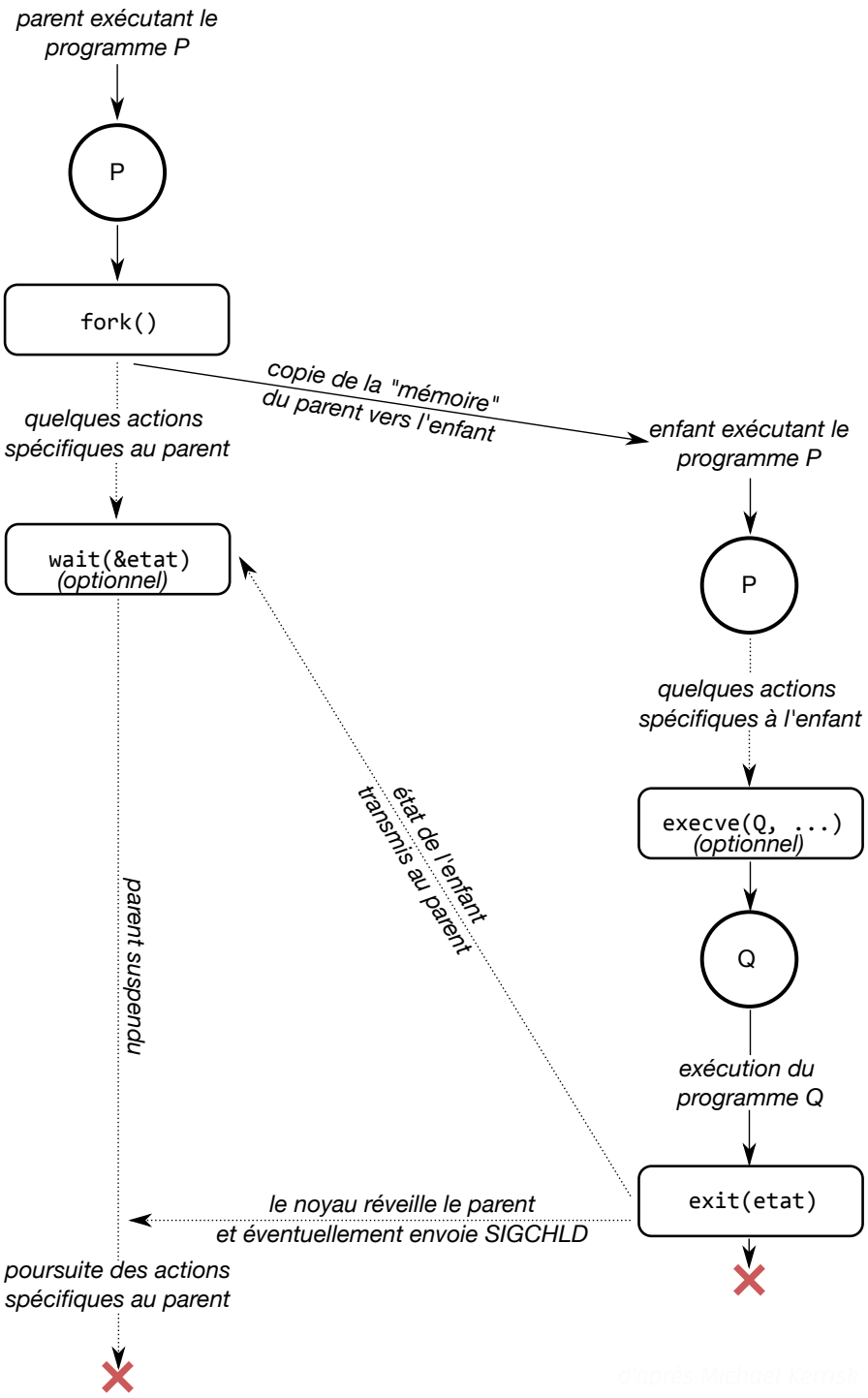


Fig. 2.9 : Association `fork()` et `exec()`.

Dans son article « A fork() in the road », Andrew Baumann critique la combinaison traditionnelle de `fork()` et `exec()` pour la création de processus sous Unix [1]. Il soutient que cette approche, bien adaptée aux machines et programmes des années 1970 (peu de mémoire et programmes plus simples), est devenue obsolète et pose de nombreux problèmes pour les développeurs d'aujourd'hui. En particulier, l'appel système `fork()` est considéré comme une mauvaise abstraction qui complique l'implémentation des systèmes d'exploitation et n'est plus pertinente aujourd'hui du fait que la duplication de l'espace d'adressage dans son entier est souvent inutile et coûteuse en matière de performance. Il suggère de déprécier `fork()` comme appel système et préconise d'utiliser les appels système suivants qui corrigent ses défauts :

```
#include <spawn.h>

int posix_spawn(pid_t *pid, const char *path, const posix_spawn_file_actions_t
 ↪  *file_actions, const posix_spawnattr_t *attrp, char *const argv[], char *const
 ↪  envp[]);
int posix_spawnp(pid_t *pid, const char *file, const posix_spawn_file_actions_t
 ↪  *file_actions, const posix_spawnattr_t *attrp, char *const argv[], char *const
 ↪  envp[]);
```

2.18 Exercices

Pour mieux comprendre les concepts abordés dans ce chapitre, vous pouvez réaliser les exercices suivants dont les solutions se trouvent en fin de polycopié.

Exercice 1 : Implémentation de l'ordonnancement circulaire

Écrivez un programme en C qui simule l'ordonnancement circulaire (*Round Robin*) pour un ensemble de processus avec des temps d'arrivée et des temps d'exécution donnés. Affichez les temps d'attente et les temps de retour pour chaque processus.

Exercice 2 : Gestion des processus zombies

Modifiez le programme fourni dans la section « Attention aux Zombies » pour éviter que le processus enfant ne devienne un *zombie*. Utilisez l'appel système `wait()` ou `waitpid()` pour assurer que le processus parent attend correctement la terminaison de son enfant.

Exercice 3 : Hiérarchie des processus

Écrivez un programme en C qui crée un processus parent et plusieurs processus enfants. Chaque processus enfant doit créer à son tour plusieurs processus enfants, formant ainsi une hiérarchie de processus. Affichez la hiérarchie des processus en utilisant les appels système `getpid()` et `getppid()`.

Exercice 4 : Variables d'environnement

Écrivez un programme en C qui modifie les variables d'environnement d'un processus enfant avant d'exécuter un nouveau programme avec `exec()`. Utilisez les fonctions `setenv()` et `getenv()` pour gérer les variables d'environnement.

3 Les threads

3.1 Introduction

Les *threads*, ou « fils d'exécution », constituent une technique fondamentale pour l'exécution simultanée de plusieurs segments de code au sein d'un même processus. Ce mécanisme permet d'optimiser les performances et la réactivité des systèmes multicoeurs en partageant efficacement un espace mémoire commun.

Les *threads* s'avèrent particulièrement intéressants dans les applications qui nécessitent l'exécution parallèle de diverses tâches, car ils améliorent considérablement la réactivité et l'efficacité du programme. Par exemple, dans les applications de traitement d'images, différents *threads* peuvent être alloués pour traiter simultanément plusieurs parties d'une image, accélérant ainsi le processus global de traitement.

Afin d'appréhender pleinement l'importance des *threads*, il est indispensable de comparer leur fonctionnement à celui des processus. Ce chapitre explore les concepts fondamentaux des *threads*, met en lumière leurs différences avec les processus, et fournit des conseils pratiques sur leur intégration et leur utilisation optimale dans vos programmes.

3.2 Espace mémoire d'un processus

Un processus dispose d'un espace mémoire structuré en plusieurs **segments** distincts, chacun ayant une fonction spécifique.

- Le **segment de code** contient le code exécutable du programme. Il est souvent en lecture seule.
- Le **segment de données** est dédié au stockage des données globales et statiques. Il est subdivisé en deux parties : la partie initialisée (où les variables globales et statiques initialisées sont stockées) et la partie non initialisée, également connue sous le nom de **BSS** (*block started by symbol*).
- Le **segment de pile** (*stack*) gère les appels de fonctions, les variables locales et les adresses de retour. Il suit une structure LIFO (*last in first out*) et sa taille peut varier dynamiquement pendant l'exécution du programme.
- Le **segment de tas** (*heap*) est utilisé pour l'allocation dynamique de la mémoire. Les fonctions telles que `malloc()` et `free()` en C permettent de gérer la mémoire dans ce segment et offre une grande flexibilité pour des structures de données dont la taille peut varier.

La figure 3.1 montre comment ces segments de mémoire sont utilisés au sein d'un processus unique.

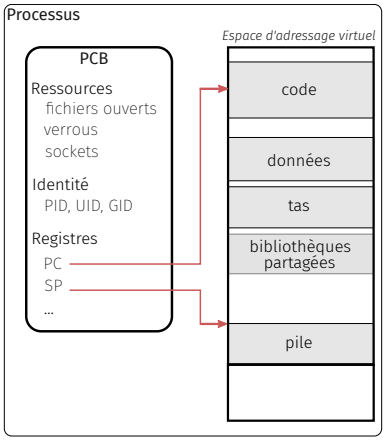


Fig. 3.1 : Espace mémoire d'un processus

3.3 Représentation alternative de l'espace d'un processus

La figure 3.2 offre une perspective différente sur la disposition de l'espace mémoire d'un processus, en prenant en considération l'existence d'un *thread* particulier : le **thread principal**.

Dans cette représentation, il est essentiel de comprendre que même un processus simple met en oeuvre un *thread* pour l'exécution de son code. Ce *thread* partage l'espace mémoire du processus avec les segments classiques que sont le segment de code, le segment de données, la pile et le tas.

L'ajout de *threads* supplémentaires au sein d'un processus modifie quelque peu cette organisation. Chaque *thread* possède sa propre pile, mais tous partagent les segments de code, de données et le tas. Cela permet une exécution concurrente et optimisée de différentes parties du programme, tout en maintenant une cohérence et une intégrité des données partagées.

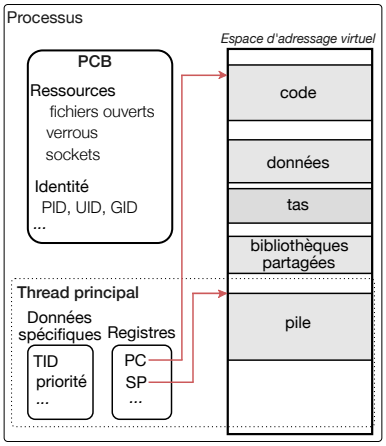


Fig. 3.2 : Autre représentation de l'espace mémoire d'un processus

3.4 Espace mémoire d'un processus multithreads

Dans un processus multithreads, chaque *thread* dispose de sa propre pile tout en partageant les segments de code, de données et le tas, comme le montre la figure 3.3. Cette configuration facilite un échange rapide et efficace des données entre les *threads*, tout en maintenant une isolation nécessaire afin d'éviter les conflits.

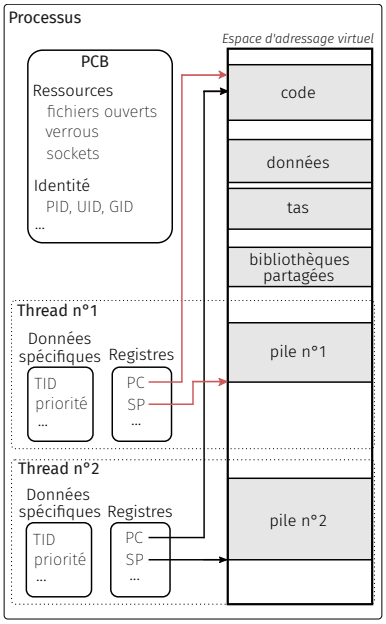


Fig. 3.3 : Représentation de l'espace mémoire d'un processus multithreads

Le partage d'un même tas par tous les *threads* simplifie le partage de données dynamiques. Cependant, cette configuration introduit des risques de situations de compétition où plusieurs *threads* peuvent tenter de modifier simultanément la même donnée, conduisant à des incohérences et des comportements imprévisibles.

Afin de gérer ces risques, l'utilisation de mécanismes de synchronisation tels que les verrous, les mutex ou les sémaphores est indispensable. Ces outils permettent de garantir l'intégrité des données partagées en contrôlant l'accès concurrent aux ressources critiques. Nous examinerons ces mécanismes en détail dans le chapitre 7.

3.5 Vue logique des threads

La figure 3.4 offre une illustration de la vue logique des *threads* au sein d'un processus comparativement à la structure hiérarchique des processus.

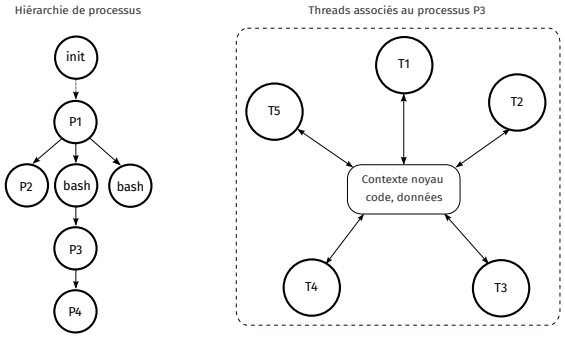


Fig. 3.4 : Vue logique de *threads*

3.6 Exécution concurrente des threads

Deux *threads* sont dits **concurrents** si leurs déroulements logiques se **couvrent** dans le temps. À l'inverse, si leurs exécutions ne se chevauchent pas, ils sont dits **séquentiels**.

Exemple

Pour illustrer cela la figure 3.5 montre différents scénarios d'exécution. Dans cette figure, les *threads* `A` et `B` ainsi que `A` et `C` sont concurrents, car leurs exécutions se chevauchent dans le temps. En revanche, les *threads* `B` et `C` sont séquentiels, car ils s'exécutent l'un après l'autre sans chevauchement.

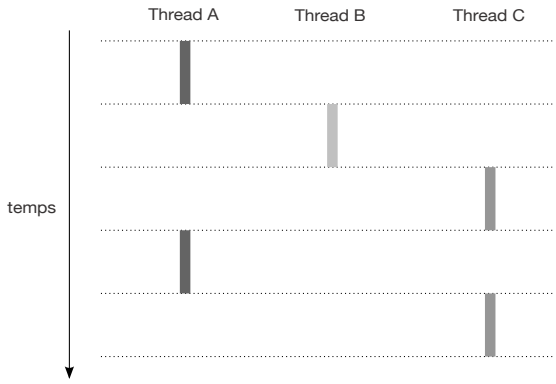


Fig. 3.5 : Exemple d'exécutions concurrente et séquentielle

3.7 Comparaison avec les processus

Les *threads* offrent des avantages significatifs par rapport aux processus, notamment une consommation réduite de ressources et une communication facilitée entre les *threads* qui exécutent des tâches liées. Selon Silberschatz ou encore Tanenbaum, ces caractéristiques rendent les *threads* particulièrement efficaces dans de nombreux scénarios d'application [32,35].

Sur certaines architectures et systèmes d'exploitation, les *threads* nécessitent environ la moitié des cycles processeur par rapport aux processus, ce qui les rend plus adaptés pour des opérations légères et rapides. Cette réduction de la charge processeur est primordiale pour les applications nécessitant une haute performance et une faible latence [32].

Critères de choix entre threads et processus

Les *threads* sont particulièrement bien adaptés aux tâches légères qui nécessitent une communication rapide et fréquente entre différentes parties du programme. Leur capacité à partager l'espace mémoire d'un processus facilite cette communication sans nécessiter des mécanismes complexes.

En revanche, les processus sont préférables pour des tâches qui nécessitent une isolation et une sécurité accrues. Les processus étant indépendants les uns des autres, une défaillance dans un processus n'affecte pas les autres, ce qui est essentiel pour les applications où la robustesse et la sécurité sont primordiales.

Utilisation pratique

- *Threads* : une application typique utilisant des *threads* concerne certaines opérations de traitement d'une image où différentes parties de celle-ci peuvent être traitées simultanément par différents *threads*, augmentant ainsi la performance globale du traitement.
- Processus : un exemple classique de l'utilisation de processus se trouve dans les services que peut offrir un serveur Web pour lequel chaque requête de client est gérée par un processus distinct. Cela permet d'isoler les requêtes, garantissant ainsi que des erreurs ou des défaillances dans une requête n'affectent pas les autres, renforçant ainsi la sécurité et la fiabilité du serveur.

3.8 Threads POSIX (bibliothèque *Pthread*)

La bibliothèque *Pthread* offre une interface standardisée et portable permettant de gérer les *threads* à travers une soixantaine de fonctions de création, de terminaison, et de synchronisation, comme le décrit Stevens [34]. Cette bibliothèque est utile pour le développement de programmes multithreads portables et efficaces

Pour inclure cette bibliothèque dans un code portable, voici un exemple typique :

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

#ifdef __WIN32__
#include <windows.h>
#endif
#ifdef __linux__
#include <unistd.h>
#endif
```

Pour compiler un projet utilisant *Pthread*, utilisez l'option « `-pthread` », qui active à la fois les macros de compilation et les options de lien nécessaires. Cette option assure que le compilateur et l'éditeur de liens intègrent correctement les fonctionnalités de la bibliothèque Pthread :

```
gcc prog.c -o prog -pthread
```

Créer un thread

La fonction `pthread_create()` est utilisée pour instancier un *thread*. Son premier paramètre est un pointeur vers le type `pthread_t` (un entier long non-signé) qui récupère l'identifiant du *thread* (son TID). Le deuxième paramètre permet de configurer les attributs du *thread* (pouvant être `NULL` pour utiliser les attributs par défaut). Le troisième paramètre est un pointeur vers la fonction que le *thread* exécutera. Le dernier paramètre est un pointeur vers les arguments potentiellement passés à cette fonction.

```
pthread_create(pthread_t *thread,
               const pthread_attr_t *attr,
               void *(*start_routine)(void *),
               void *arg);
```

Terminer un thread

La fonction `pthread_exit()` permet à un *thread* de se terminer proprement. Par ailleurs, l'appel à `pthread_cancel()` depuis un autre *thread* permet d'« annuler » un *thread*, bien que cette opération puisse laisser des ressources verrouillées. Il est prudent d'utiliser `pthread_setcancelstate()` afin de configurer l'état d'annulation du *thread* (consulter [4] pour une description plus détaillée).

Il est également possible d'utiliser la fonction `exit()`, qui entraîne la terminaison de tous les *threads*, y compris le *thread* principal.

Attendre des threads

La fonction `pthread_join()` permet à un *thread* parent d'attendre la fin d'un *thread* enfant. Elle est l'équivalent des appels système `wait()` et `waitpid()` pour les processus.

Dans le cas des *threads*, nous verrons dans la [section sec :threads-attributes] qu'un *thread* peut ne pas être « attendu ».

Déterminer l'identifiant du thread courant

La fonction `pthread_self()` permet de récupérer l'identifiant attribué par le système au *thread*. C'est l'équivalent de l'appel système `getpid()` pour les processus.

Synchroniser l'accès aux variables partagées

Afin de gérer efficacement la concurrence entre les *threads* et prévenir les problèmes d'accès aux variables partagées, diverses stratégies et mécanismes de synchronisation peuvent être utilisés comme nous le verrons dans le chapitre 7 sur la synchronisation des processus.

Les mutex (`pthread_mutex_t`) fournissent un moyen de verrouiller les ressources pour un seul *thread* à la fois à l'aide de fonctions adéquates (`pthread_mutex_init()`, `pthread_mutex_[un]lock()`).

De même, les variables conditionnelles (`pthread_cond_t`) permettent aux *threads* de se mettre en pause ou de se réveiller en fonction de certaines conditions (voir `pthread_cond_init()`, `pthread_cond_[timed]wait()`).

Enfin, les sémaphores peuvent également être utilisés pour contrôler l'accès à un nombre limité de ressources.

3.9 Programme utilisant la bibliothèque *Pthread*

Le programme suivant illustre l'utilisation de la bibliothèque *Pthread* pour créer et gérer des *threads*. Le code montre comment créer un *thread* secondaire, obtenir l'identifiant du *thread* principal, et attendre la fin du *thread* secondaire.

```
/* thread routine */
void *todo(void *vargp) {
    printf("Hello from the thread No %ld\n", pthread_self());
    return NULL;
}

int main(void) {
```

```
    pthread_t tid1, tid0;

    pthread_create(&tid1, NULL, &todo, NULL);  /* Creation of a secondary thread */
    tid0 = pthread_self();     /* Get TID of the main thread */
    printf("I am the main thread (No %ld). I launched the thread No %ld\n", tid0, tid1);
    pthread_join(tid1, NULL);  /* Wait for the end of the secondary thread */

    return EXIT_SUCCESS;
}
```

La sortie du programme est la suivante :

```
I am the main thread (No 120362432). I launched the thread No 245100544
Hello from the thread No 245100544
```

Dans ce programme, la fonction `pthread_create()` est utilisée pour lancer un *thread* secondaire qui exécute la routine *todo()*. Le *thread* principal récupère son propre identifiant en appelant `pthread_self()` et affiche un message indiquant qu'il a lancé le *thread* secondaire. Ensuite, la fonction `pthread_join()` est appelée afin d'attendre la terminaison du *thread* secondaire, garantissant que le *thread* principal ne termine pas avant.

3.10 Utilisation des attributs d'un thread

Les attributs des *threads* (voir la structure `pthread_attr_t`) permettent de configurer divers paramètres pour les *threads* avant leur création.

Détachement

La fonction `pthread_attr_setdetachstate()` permet de définir si un *thread* est détachable ou non. Si un *thread* est détachable, alors son parent n'a pas l'obligation d'attendre sa fin. Par défaut, un *thread* n'est pas détachable (valeur `PTHREAD_CREATE_JOINABLE`).

Différence entre `pthread_exit()` et `return EXIT_SUCCESS` (ou `exit()`)
À la fin de la fonction principale, un appel à `pthread_exit()` termine le *thread* principal, mais pas forcément les autres *threads* s'ils n'ont pas été attendus à l'aide de `pthread_join()`. Le processus reste donc actif tant que les *threads* secondaires ne sont pas terminés. L'avantage est que les *threads* secondaires ont le temps d'exécuter l'ensemble de leur code de nettoyage ou de fin avant que le processus ne se termine véritablement.
A contrario, si aucun appel à `pthread_join()` n'a été fait avant l'appel à `return` (ou à `exit()`), alors le processus se termine sans que les *threads* secondaires ne soient forcément terminés. Ils sont donc stoppés brutalement.
Dans le cas où des *threads* détachés sont utilisés, il est **préférable** de faire appel à `pthread_exit()`.

Taille de la pile

La fonction `pthread_attr_setstacksize()` permet de définir la taille de la pile pour le(s) *thread(s)*. La taille est typiquement de 2 Mio par défaut sur les systèmes Linux.

Exemple d'utilisation :

```
size_t stacksize = 1024 * 1024; /* 1 Mb */
pthread_attr_setstacksize(&attr, stacksize);
```

Stratégie d'ordonnancement

De la même manière que pour les processus, il est possible de définir pour les *threads* une stratégie d'ordonnancement. La fonction `pthread_attr_setschedpolicy()` permet de définir la stratégie à utiliser, par exemple FCFS (`SCHED_FIFO`) ou encore un ordonnancement circulaire (`SCHED_RR`). Par défaut, la stratégie est à priorité dynamique basée sur le partage équitable du processeur (valeur `SCHED_OTHER`).

Exemple d'utilisation :

```
int policy = SCHED_FIFO;
pthread_attr_setschedpolicy(&attr, policy);
```

Paramètres de la stratégie d'ordonnancement

Suivant la stratégie d'ordonnancement choisie, il est possible de définir certains paramètres tels que la priorité à l'aide de la fonction `pthread_attr_setschedparam()`. Par défaut la priorité est à 0.

Exemple d'utilisation :

```
struct sched_param param;
param.sched_priority = 10;
pthread_attr_setschedparam(&attr, &param);
```

Héritage des attributs du parent

La fonction `pthread_attr_setinheritsched()` permet de spécifier si le *thread* hérite des attributs d'ordonnancement de son *thread* parent ou s'il utilise les siens. Par défaut, le *thread* hérite des attributs de son *thread* parent (`PTHREAD_INHERIT_SCHED`).

Exemple d'utilisation :

```
pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
```

Protection de la pile

La fonction `pthread_attr_setguardsize()` définit la taille de la zone de protection de la pile qui est utilisée pour détecter les dépassements de pile. Cette valeur est typiquement de 4 Kio sur les systèmes Linux.

Exemple d'utilisation :

```
size_t guardsize = 4096; /* 4 Ko */
pthread_attr_setguardsize(&attr, guardsize);
```

Portée du thread

La fonction `pthread_attr_setscope()` permet de déterminer si l'ordonnancement du *thread* est de la responsabilité du processus parent ou du système d'exploitation lui-même. Par défaut, ce rôle est attribué à l'ordonnanceur du système d'exploitation (valeur `PTHREAD_SCOPE_SYSTEM`).

Exemple d'utilisation :

```
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
```

Voici un exemple de configuration d'attributs de *thread* permettant de définir la taille de la pile et spécifier que les *threads* sont « détachés » (c'est-à-dire qu'une fois terminés, ils n'ont pas à être attendus par le *thread* principal) :

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

void *todo(void *arg) {
    printf("Thread %ld started\n", pthread_self());
    sleep(1);
    printf("Thread %ld finished\n", pthread_self());

    return NULL;
}

int main(void) {
    pthread_t thread;
    pthread_attr_t attr;
    size_t stacksize;
    struct sched_param param;

    /* Initialize the thread's attributes */
    pthread_attr_init(&attr);

    /* Set the size of the stack for threads (1 Mib) */
    stacksize = 1024 * 1024;
    pthread_attr_setstacksize(&attr, stacksize);

    /* Set the thread as detached */
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

    /* Set the scheduler policy to around-robin */
    pthread_attr_setschedpolicy(&attr, SCHED_RR);

    /* Set priority for the nscheduler policy */
    param.sched_priority = 10;
    pthread_attr_setschedparam(&attr, &param);

    /* Create the thread with the previous attributes */
    if (pthread_create(&thread, &attr, todo, NULL) != 0) {
        perror("pthread_create");
        return EXIT_FAILURE;
    }

    /* Destroy attributes */
    pthread_attr_destroy(&attr);

    /* The main thread is waiting for detached threads to end */
    sleep(2);

    pthread_exit(NULL);
```

```
}
```

Linux utilise le concept de **tâche** pour désigner à la fois les processus et les *threads*. Ainsi, dans le noyau Linux il n'y a pas de distinction au niveau de l'ordonnanceur entre les *threads* et les processus : tous sont des tâches avec différents attributs de partage de ressources.
D'ailleurs sous Linux, la gestion des *threads* au niveau du système est réalisée principalement à l'aide de l'appel système `clone()`, qui est plus flexible que `pthread_create()` fourni par la bibliothèque *Pthread* ou même `fork()`.
`clone()` permet de spécifier précisément quels aspects de l'environnement d'exécution du processus appelant peuvent être partagés avec le nouveau *thread*. Cela inclut les espaces d'adressage, le système de fichiers, et les signaux, parmi d'autres options. Linux traite les *threads* créés par `clone()` de la même manière que les processus, en leur attribuant un identifiant de processus unique, mais avec la capacité de partager des ressources [26].
Attention cependant, cet appel système est propre à Linux !

Voici comment nous pourrions réécrire le tout premier programme qui utilisait `fork()` dans la section 2.11.1, mais en utilisant cette fois-ci `clone()` :

```
#include <stdio.h>
#include <sched.h>   /* header to use clone() */
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

#define STACK_SIZE (1024 * 1024) /* stack size for the cloned child */

void handle_fatal_error_and_exit(char *msg){
    perror(msg);
    exit(EXIT_FAILURE);
}

int manage_child(void *arg) {
    printf("Child process (PID %" PRId32 ")\n", getpid());
    printf("My parent's PID is %" PRId32 "\n", getppid());
    printf("Instructions of child process...\n");
    exit(EXIT_SUCCESS);
}

int main(void) {
    char *stack = (char *)malloc(STACK_SIZE);
    if (stack == NULL) {
        handle_error_and_exit("Failed to allocate memory for stack\n");
    }

    /*
     * The flags CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_PARENT
     * can be adjusted based on what you need to share between parent and child
     */
    pid_t pid = clone(manage_child, stack + STACK_SIZE,
                      CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_PARENT,
```

```
                    NULL);
    if (pid < 0) {
        handle_error_and_exit("Error using clone().\n");
    }

    printf("Parent process (PID %" PRId32 ")\n", getpid());
    printf("My child's PID is %" PRId32 "\n", child_pid);
    printf("Instructions of parent process...\n");
    waitpid(pid, NULL, 0); /* Wait specifically for the cloned child */
    free(stack); /* Clean up allocated stack */

    return EXIT_SUCCESS;
}
```

Dans cet exemple, `clone()` est utilisé pour créer un nouveau processus qui partage certaines ressources avec le processus parent.

3.11 Exécution du programme utilisant des threads

La figure 3.6 montre l'exemple d'un fil principal qui a créé un *thread* secondaire et attend sa fin.

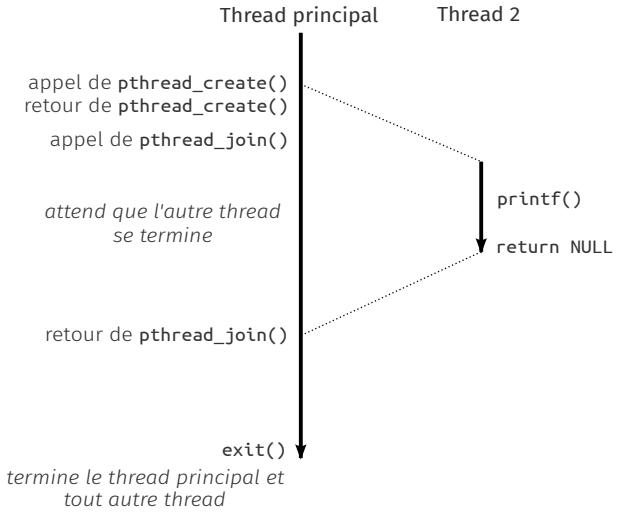


Fig. 3.6 : Exécution du programme utilisant des *threads*.

3.12 Exemple d'application

La multiplication de matrices est une opération fondamentale qui est nécessaire dans de nombreux domaines de l'informatique et de l'ingénierie, comme par exemple le traitement d'images, les réseaux de neurones profonds ou encore la cryptographie.

Par exemple, en traitement d'images, les matrices sont utilisées pour représenter les pixels d'une image. La multiplication de matrices permet de réaliser des transformations géométriques, telles que la rotation, la translation et la mise à l'échelle. De la même façon, les réseaux de neurones profonds utilisent intensivement la multiplication matricielle pour le traitement des données et l'apprentissage des modèles. Les poids et les biais des neurones sont stockés sous forme de matrices, et l'activation des neurones à chaque couche du réseau est obtenue par des

multiplications matricielles. Enfin, en cryptographie, les matrices sont utilisées pour chiffrer et déchiffrer des messages. Un exemple classique est celui du chiffre de Hill qui utilise la multiplication de matrices pour transformer des blocs de texte en texte chiffré [17].

La multiplication de matrices consiste à prendre le produit scalaire des lignes de la première matrice avec les colonnes de la seconde matrice.

Si A est une matrice $M \times N$ et B une matrice $N \times P$, alors leur produit R sera une matrice $M \times P$ définie par :

$$R_{ij} = \sum_{k=1}^{N} A_{ik} \cdot B_{kj}$$

Cette multiplication pour des matrices de grande taille est très coûteuse en ressources.

L'exemple qui suit va illustrer l'utilisation des *threads* pour une multiplication de matrices, puis montrer l'efficacité accrue des opérations en mode multithreads lorsque les matrices sont de tailles conséquentes.

3.12.1 Multiplication de matrices sans threads

Considérons deux matrices carrées A et B de taille $N \times N$ pour simplifier.

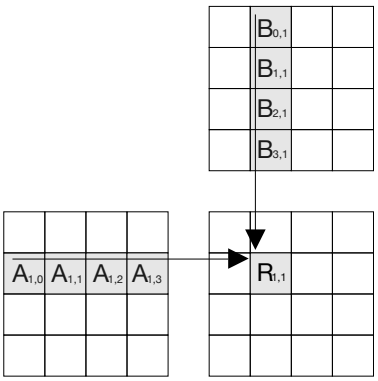


Fig. 3.7 : Multiplication matricielle monothread.

3.12.2 Multiplication de matrices : répartition threads et données

Si nous utilisons quatre *threads* afin de paralléliser notre multiplication des matrices A et B, nous pouvons répartir le travail entre ces *threads*. Pour cela nous divisons la matrice résultante R en blocs de taille équivalente, tels que des colonnes, des lignes ou encore des quadrants comme sur la figure 3.8, puis nous assignons chacun des blocs à un *thread* distinct.

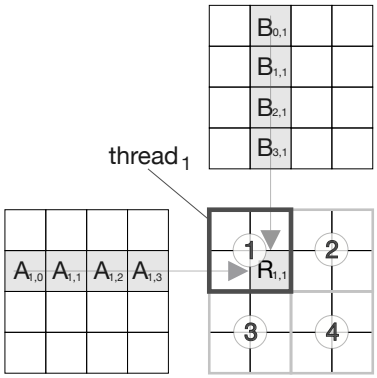


Fig. 3.8 : Multiplication matricielle avec 4 threads.

3.12.3 Multiplication de matrices : choix des structures

Nous devons ensuite transmettre à nos *threads* les différentes matrices. Pour cela, nous définissons les structures de données suivantes :

```
typedef struct {
  int rows;           /*<! The number of rows of the matrix */
  int columns;        /*<! The number of columns of the matrix */
  double **matrix;    /*<! The matrix values */
} matrix_t;

typedef struct {
    int quadrant;     /*<! The quadrant on which a single thread will operate */
    matrix_t *m1;     /*<! The first input matrix */
    matrix_t *m2;     /*<! The second input matrix */
    matrix_t *answer; /*<! The result matrix */
} args_t;
```

La structure de données `args_t` permet de passer les matrices d'entrée et de sortie à la fonction réalisée par les *threads*, ainsi que le quadrant sur lequel le *thread* va limiter ses calculs (voir la mise en pratique en salle machine).

```
/* Multi thread functions */
void multi_thread(matrix_t *m1, matrix_t *m2);
void *product_matrix_thread(void *args);

/* Utils matrix functions */
void print_matrix(matrix_t *m);
/* ... */
```

3.12.4 Multiplication de matrices : comparaison

Afin de comparer les performances entre une implémentation monothread et une implémentation multithreads, nous mesurons le temps d'exécution des deux versions pour des matrices de taille significative comme le montre la figure 3.9.

1 *thread* / 4 *threads*

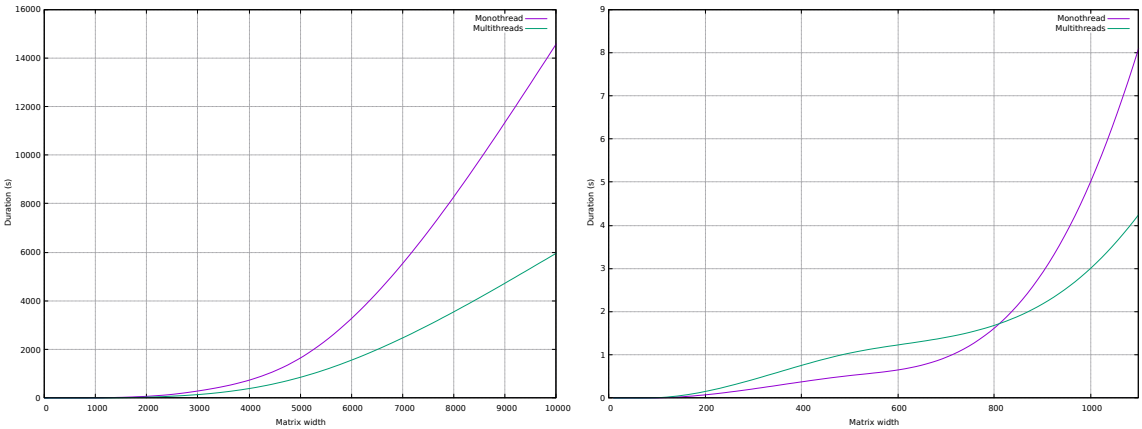


Fig. 3.9 : Comparaison des temps de calcul monothread / 4 threads

Ces résultats démontrent l'amélioration des performances lorsqu'on utilise des *threads* pour paralléliser des calculs conséquents. Notons toutefois que pour des tâches plus simples, ces derniers peuvent ralentir l'application.

3.13 Partage de variables dans un programme multithreads

Quelles sont les variables partagées dans un programme multithreads ? Afin de répondre à cette question, il est nécessaire de répondre aux questions suivantes :

- Quel modèle mémoire pour les *threads* ?
- Comment les variables sont-elles "projetées" dans les instances mémoire ?
- Combien de *threads* les référencent ?

Théoriquement, chaque *thread* :

- s'exécute dans le contexte d'environnement d'un processus ;
- possède son propre contexte d'environnement (son identifiant, pile, pointeur de pile, compteur ordinal, etc.) ;
- partage les éléments restants du contexte du processus :
  - code, données, tas et segments de bibliothèques partagées ;
  - fichiers ouverts, etc.

Mais dans la pratique, seules les valeurs des registres sont séparées et protégées. Par contre, n'importe quel *thread* peut lire et écrire dans la pile des autres, ce qui peut entraîner des problèmes.

3.14 Accès à la pile d'un autre thread

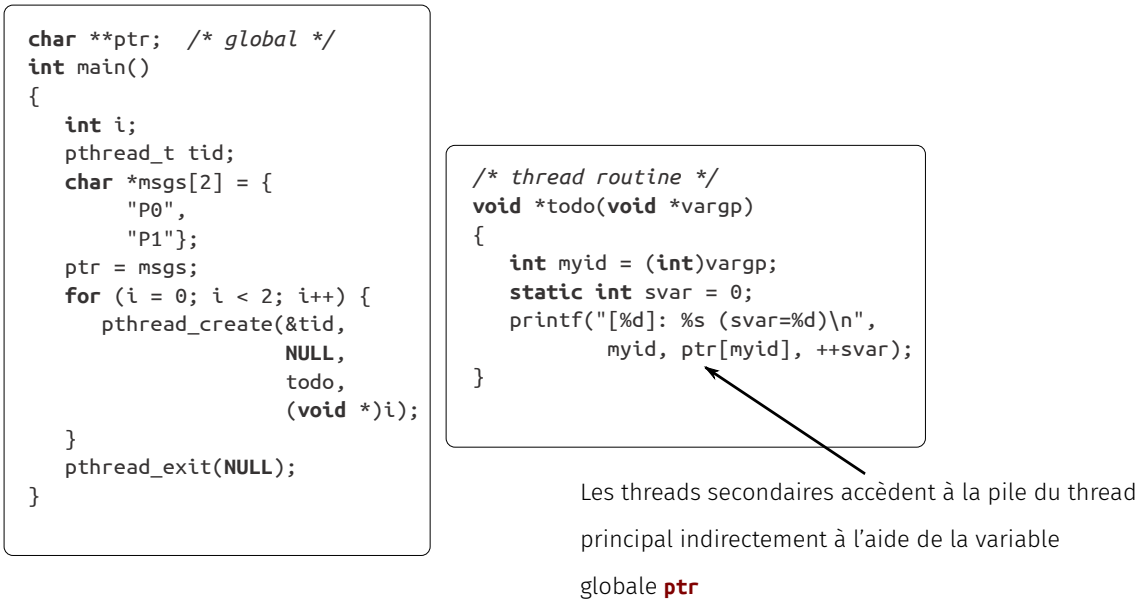


Fig. 3.10 : Variables accessibles entre threads

3.15 Projection des variables en mémoire

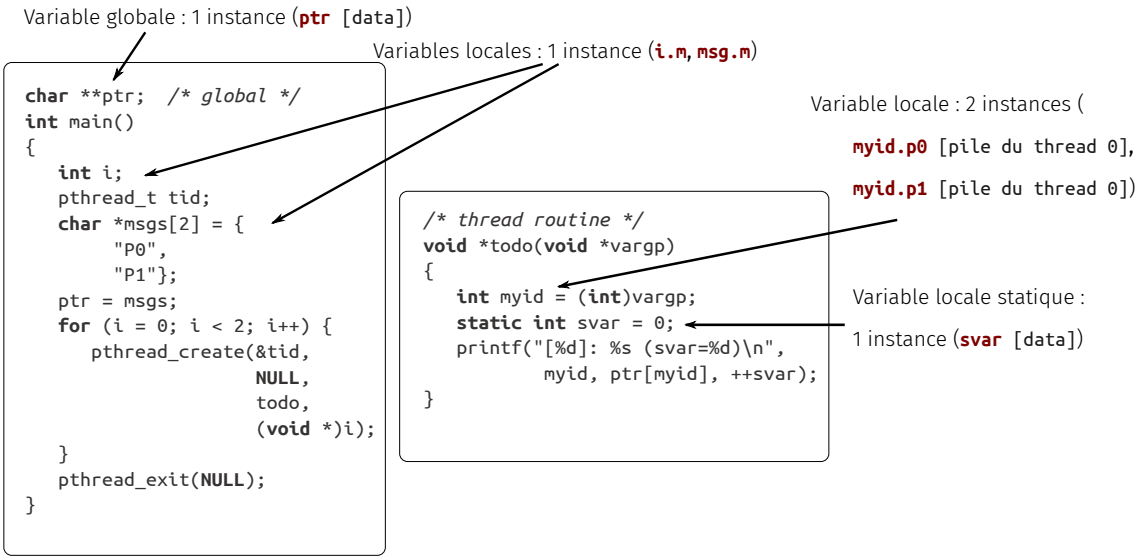


Fig. 3.11 : Projection des variables en mémoire

3.16 Analyse des variables partagées

Variables ou instances référencées par les threads

Variables / instances	*thread* principal	*thread* 0	*thread* 1
ptr	oui	oui	oui
svar	non	oui	oui
i.m	oui	non	non
msgs.m	oui	oui	oui
myid.p0	non	oui	non
myid.p1	non	non	oui

3.17 Fonctions sûres et re-entrantes

3.17.1 Fonction sûre (***thread-safe***)

Les fonctions appelées depuis un *thread* doivent être **sûres** (*thread-safe*). Pour comprendre cela, il est utile de considérer les quatre classes de fonctions qui ne sont pas sûres.

Classes de fonctions non sûres

La première classe regroupe les fonctions qui échouent à protéger les variables partagées. Par exemple, un programme de comptage où plusieurs *threads* incrémentent une même variable sans synchronisation. La correction passe par l'utilisation de sémaphores ou de mutex, bien que cela puisse ralentir le programme.

La deuxième classe regroupe les fonctions qui comptent sur un état persistant à travers des invocations multiples. Considérons la fonction `rand()` ci-dessous. Elle utilise une variable statique pour maintenir son état à travers les appels, rendant ainsi la fonction non sûre en environnement multithreads. La correction implique de réécrire la fonction pour encapsuler l'état dans une structure passée en argument.

```
/* rand - return pseudo-random integer on 0..32767 */
int rand(void) {
    static unsigned int next = 1;
    next = next * 1103515245 + 12345;
    return (unsigned int)(next / 65536) % 32768;
}

/* srand - set seed for rand() */
void srand(unsigned int seed) {
    next = seed;
}
```

La troisième classe de fonction non-sûres regroupe les fonctions qui retournent un pointeur sur une variable statique : la fonction `gethostbyname()` en est un exemple. La version corrigée `gethostbyname_ts()` alloue de la mémoire dynamiquement pour éviter de partager la variable statique `h`.

```
struct hostent *gethostbyname(char* name) {
    static struct hostent h;
    /*
     * contact the DNS and fill in h
     */
```

```
    return &h;
}

struct hostent *gethostbyname_ts(char *name) {
    struct hostent *q = malloc(sizeof(struct hostent));

    P(&mutex); /* lock */
    struct hostent *p = gethostbyname(name);
    *q = *p;   /* copy */
    V(&mutex); /* unlock */

    return q;
}
```

La quatrième classe de fonctions non-sûres regroupe les fonctions qui elles-même font appel à des fonctions non sûres des classes précédente. La solution consiste à modifier la fonction pour qu'elle n'appelle que des fonctions sûres.

3.17.2 Bibliothèques de fonctions sûres

La plupart des fonctions de la bibliothèque C standard sont sûres, par exemple : `malloc()`, `free()`, `printf()`, `scanf()`. Les appels système Unix sont quant à eux tous **sûrs**.

Appels de bibliothèques

Certaines fonctions non sûres ont des versions re-entrantes qui sont adaptées à l'utilisation dans des environnements multithreads :

Fonction non sûre	Classe	Version re-entrante
asctime()	3	asctime_r()
ctime()	3	ctime_r()
gethostbyaddr()	3	gethostbyaddr_r()
gethostbyname()	3	gethostbyname_r()
inet_ntoa()	3	--
localtime()	3	localtime_r()
rand()	2	rand_r()

3.17.3 Fonction re-entrante

Une fonction est dite **re-entrante** si elle n'accède à aucune variable partagée lorsqu'elle est appelée par plusieurs *threads*. Les fonctions re-entrantes forment un sous-ensemble des fonctions sûres, comme l'illustre la figure 3.12.

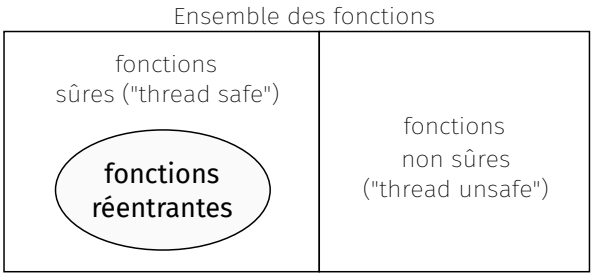


Fig. 3.12 : Ensemble des fonctions re-entrantes.

Exemple de fonction non re-entrante

La fonction `write_hexa()` suivante n'est pas re-entrante. Elle utilise un tableau statique `buffer`. À chaque appel de la fonction, la même adresse mémoire sera donc utilisée, ce qui peut causer des comportements anormaux si un autre *thread* ou une autre fonction utilise `buffer` concomittament. De plus, `sprintf()` n'alloue aucune mémoire lors de son opération et se contente de re-écrire directement sur le tampon. Si plusieurs *threads* appellent `write_hexa()` en même temps, ils risquent les uns les autres de saboter leurs modifications respectives. Le résultat peut être différent d'une exécution à l'autre.

```
/* The content of buffer is undetermined when simultaneous calls. */
const char *write_hexa(int i) {
    static char buffer[0x10];
    sprintf(buffer, "0x%.8x", i);
    return(buffer);
}
```

La fonction `write_hexa_r()` alloue dynamiquement de la mémoire pour éviter les conflits.

```
char *write_hexa_r(int i) {
    char *buffer = malloc(17); /* 17 bytes for "0x%.8x" format string */
    sprintf(buffer, "0x%.8x", i);
    char *result = buffer; /* store the result in a separate variable */
    buffer = NULL; /* set the buffer to NULL to avoid accidental use */
    return result;
}
```

Synchronisation et sécurité

Les fonctions **sûres** (*thread-safe*) et **re-entrantes** sont donc essentielles lorsqu'il s'agit d'éviter les conflits et garantir la cohérence des données entre les *threads*. Elles doivent être couplées avec des pratiques de programmation qui évitent l'utilisation de ressources partagées non comme nous le verrons dans le chapitre 7.

3.18 Exercices

Afin de mieux comprendre les concepts abordés dans ce chapitre, vous pouvez réaliser les exercices suivants dont des éléments de réponse se trouvent à la fin de ce document.

Exercice 1 : Création de threads simples

Écrivez un programme en C qui crée deux *threads* qui vont réaliser la fonction `todo()`. Chaque *thread* doit afficher des messages indiquant qu'il commence puis qu'il termine son exécution. Utilisez la fonction `pthread_create()` pour créer les *threads* depuis le fil d'exécution principal et la fonction `pthread_join()` pour attendre leur terminaison.

Exercice 2 : Création et identification des threads

Écrivez un programme en C qui crée trois *threads*. Chaque *thread* doit afficher son identifiant (TID) et un message indiquant qu'il est en cours d'exécution.

Exercice 3 : Utilisation de threads détachés

Écrivez un programme C qui crée un *thread* détaché. Le *thread* doit afficher un message indiquant qu'il commence puis qu'il termine son exécution.

4 Fichiers et flux d'entrées-sorties

4.1 Introduction

Ce chapitre traite des fichiers et des flux d'entrées-sorties, éléments essentiels des systèmes d'exploitation, notamment Unix. Nous aborderons les systèmes de gestion de fichiers, l'organisation hiérarchique des fichiers, et les opérations courantes comme la création, la lecture, l'écriture et la suppression de fichiers.

4.1.1 Généralités et définitions

Un **fichier** est un ensemble de blocs stockés sur un support de stockage tel qu'un disque dur, un CD ou une clé USB. Les fichiers sont gérés par un **système de gestion de fichiers**, qui organise, nomme, conserve, protège et rend accessibles les fichiers. Ce système repose sur une structure appelée **système de fichiers**, qui définit comment les fichiers sont stockés, organisés et accessibles sur le support.

La figure 4.1 montre les différents moyens d'accéder à des fichiers :

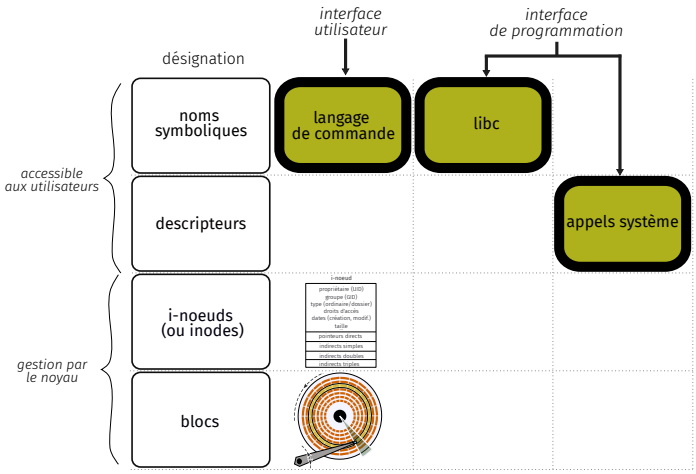


Fig. 4.1 : Désignation et différents points d'accès aux fichiers.

4.1.2 Exemples de systèmes de fichiers

Voici un tableau récapitulatif des systèmes de fichiers couramment utilisés :

Nom	Système	Journalisation	Compatibilité	Taille max. volume	Taille max. fichier
FAT32	Ms-Windows	Non	Très élevée	16 To	4 Gio

Nom	Système	Journalisation	Compatibilité	Taille max. volume	Taille max. fichier
NTFS	Ms-Windows	Oui	Moyenne	256 Tio	16 Tio
exFAT	Ms-Windows	Non	Élevée	16 Eio	128 Pio
ext3, ext4	Linux	Oui	Moyenne	32 Tio / 1 Eio	2 Tio / 256 Tio
Btrfs	Linux	Oui	Moyenne	16 Eio	16 Eio
HFS+	Mac OS X	Oui	Faible	8 **Eo**	8 Eio
APFS	Mac OS X	Oui	Moyenne	--	8 Eio
JFS, JFS2	AIX	Oui	Faible	Pas de limite définie	4 Pio
ZFS	Solaris	Oui	Moyenne	256 trillions de Eio	16 Eio

La journalisation permet de suivre les modifications et d'aider à la récupération après une panne. Les systèmes de fichiers qui supportent la journalisation enregistrent des journaux de transactions garantissant l'intégrité des données.

Rappel :

- 1 Kio (Kibioctet) = 2^{10} octets
- 1 Mio (Mibioctet) = 2^{20} octets
- 1 Gio (Gibioctet) = 2^{30} octets
- 1 Tio (Tébioctet) = 2^{40} octets
- 1 Pio (Pébioctet) = 2^{50} octets
- 1 Eio (Exbioctet) = 2^{60} octets
- 1 Zio (Zébioctet) = 2^{70} octets
- 1 Yio (Yobioctet) = 2^{80} octets

4.1.2.1 FAT32 {-}

FAT32 est une version avancée du système de fichiers FAT (*file allocation table*), principalement utilisée sur Ms-Windows. Il prend en charge des partitions allant jusqu'à 16 Tio et est compatible avec la plupart des systèmes d'exploitation. Cependant, il a des limitations, notamment une taille maximale de fichier de 4 Gio et l'absence de gestion des permissions de fichiers.

4.1.2.2 NTFS {-}

NTFS (*new technology file system*) est un système de fichiers propriétaire développé par Microsoft pour Ms-Windows. Il offre des fonctionnalités avancées comme la journalisation, la sécurité basée sur les permissions, la compression de fichiers et la prise en charge des fichiers de grande taille. NTFS est le système de fichiers par défaut pour les versions modernes de Ms-Windows.

4.1.2.3 exFAT {-}

Extended file allocation table (exFAT) est conçu pour combler les lacunes de FAT32 en prenant en charge des fichiers de plus de 4 Gio. Optimisé pour les clés USB et les cartes SD, il offre une meilleure compatibilité entre différents systèmes d'exploitation.

4.1.2.4 ext, ext2, ext3 et ext4 {-}

La famille des systèmes de fichiers « ext » (*extended file system*) est la première conçue spécifiquement pour Linux.

- **ext** : Première version, maintenant obsolète.
- **ext2** : Meilleures performances et gestion de l'"espace disque, sans journalisation.
- **ext3** : Introduit la journalisation pour une meilleure fiabilité.
- **ext4** : Améliorations en termes de performance, gestion de l'"espace et fiabilité.

4.1.2.5 Btrfs {-}

Btrfs (*B-tree file system*) est conçu pour Linux et offre des fonctionnalités avancées telles que la compression des données, la déduplication, les points de restauration (*snapshots*) et une gestion dynamique de l'espace.

4.1.2.6 HFS+ {-}

HFS+ (*hierarchical file system plus*) est utilisé par Mac OS X avant l'introduction de APFS. Il prend en charge de grandes tailles de fichiers et des fonctionnalités avancées comme la gestion des métadonnées et la journalisation.

4.1.2.7 JFS, JFS2 {-}

Développés par IBM pour AIX, JFS et JFS2 (*journaled file system*) offrent fiabilité et performances grâce à la journalisation, permettant une récupération rapide après une panne.

4.1.2.8 UFS {-}

UFS (*Unix file system*), est un système de fichiers traditionnel utilisé sur de nombreux systèmes Unix, y compris Solaris. Il est connu pour sa stabilité et sa simplicité.

4.1.2.9 ZFS {-}

ZFS (*zettabyte file system*), développé par Sun Microsystems (maintenant Oracle), est un système de fichiers et gestionnaire de volumes logiques réputé pour son intégrité des données, son évolutivité, ses points de restauration et sa gestion avancée des volumes.

4.1.3 Désignation des fichiers : organisation hiérarchique

Les fichiers sont organisés hiérarchiquement. Voici les éléments essentiels :

- Les noms forment une structure arborescente.
- Les **dossiers** ou répertoires (*folders* ou *directories*) constituent les noeuds intermédiaires dans l'arborescence. Sous Unix, ils sont considérés comme des fichiers spéciaux.
- Les **fichiers simples** sont les noeuds terminaux.
- Le nom **absolu** est le chemin complet d'un fichier depuis la racine « `/` ».
- Le nom **relatif** est le chemin d'un fichier par rapport au dossier courant.

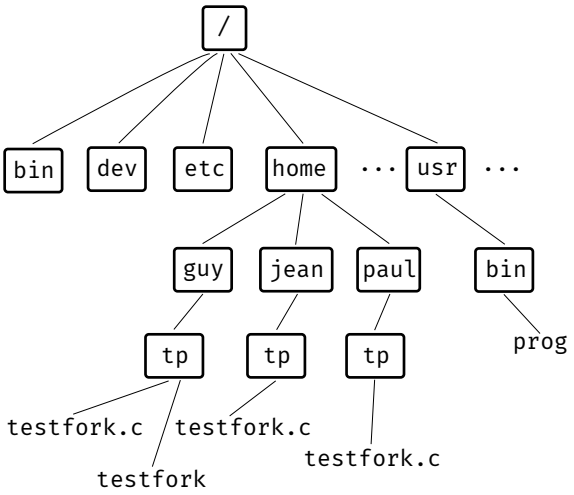


Fig. 4.2 : Arborescence des fichiers sous Unix

Liens

Les liens permettent de créer des raccourcis vers d'autres fichiers, facilitant ainsi l'organisation et l'accès aux données sans duplication. Il existe deux types de liens :

Les **liens symboliques** créent des raccourcis vers d'autres fichiers, facilitant ainsi l'organisation et l'accès aux données sans duplication. Contrairement aux liens symboliques, les **liens physiques** (ou liens durs) pointent directement vers les données sur le disque et ne deviennent pas invalides si le fichier cible est supprimé, tant qu'il reste au moins un lien physique pointant vers les données.

Pour illustrer cela, si le dossier courant est « `/home/guy/` », la commande suivante crée un lien symbolique :

```
$ ln -s /usr/bin/prog monprog
```

Dans ce cas, « `monprog` » dans le dossier courant désigne maintenant le fichier « `/usr/bin/prog` ».

En revanche, un lien physique est créé sans l'option « `-s` » :

```
$ ln /usr/bin/prog monprog
```

Ainsi, « `monprog` » et « `/usr/bin/prog` » pointent tous deux directement vers les mêmes données sur le disque, et la suppression de l'un des fichiers ne rend pas l'autre invalide.

4.2 À propos du dossier courant

Chaque utilisateur possède un dossier de base par défaut, souvent situé dans « `/home/nom_utilisateur` », par exemple « `/home/paul` ». Un raccourci pour désigner ce dossier est « `~paul` ».

Changer de dossier courant se fait au moyen de la commande `cd`. Voici quelques commandes utiles :

Pour changer de dossier :

```
$ cd /chemin/vers/nouveau_dossier
```

Pour retourner au dossier de base :

```
$ cd
```

Pour afficher le chemin absolu du dossier courant :

```
$ pwd
```

Pour lister le contenu du dossier courant avec détails :

```
$ ls -l
```

Pour lister tous les fichiers, y compris les fichiers cachés :

```
$ ls -a
```

4.3 Règles de recherche des fichiers

Pour exécuter un programme (fichier exécutable `x`), il suffit d'entrer son nom dans le terminal. Le système le trouve en suivant une règle de recherche qui utilise une suite de dossiers spécifiée dans la variable d'environnement `PATH`. Cette variable contient une liste de dossiers séparés par le caractère deux-points (« `:` »). Sous Ms-Windows, les dossiers dans `PATH` sont séparés par des points-virgules (« `;` »).

Pour afficher le contenu de la variable `PATH`, utilisez la commande suivante :

```
$ echo $PATH
/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/opt
```

Par exemple, si vous avez écrit votre propre compilateur `gcc` et que vous souhaitez l'exécuter depuis le dossier courant, il y a deux solutions :

1. Changez la règle de recherche en mettant le dossier courant en tête :

   ```
   $ export PATH=.:\$PATH
   ```

2. Exécuter le programme en précisant le chemin relatif :

   ```
   $ ./mynewgcc-project/bin/gcc ...
   ```

Pour les programmes usuels (commandes du système, etc.), vous pouvez utiliser la commande `which` pour connaître le chemin absolu du fichier qui sera exécuté par défaut. Par exemple :

```
$ which gcc
/usr/bin/gcc
```

Cette commande `which` est particulièrement utile lorsqu'il s'agit de vérifier quel fichier exécutable sera utilisé lorsque vous tapez une commande, surtout si vous avez plusieurs versions d'un programme installées à différents endroits.

4.4 Utilisations courantes des fichiers

Sous Unix, le contenu d'un fichier est simplement une suite d'octets. L'interprétation du contenu dépend de l'utilisation. Voici quelques exemples courants.

Programmes exécutables (« binaires »)

Les programmes exécutables contiennent du code machine que le système d'exploitation peut charger en mémoire et exécuter. Il existe deux types principaux de programmes exécutables sous Unix :

- Les commandes du système : Généralement situées dans des dossiers comme « `/bin` », « `/usr/bin` », « `/sbin` ». Elles fournissent des fonctionnalités de base comme `ls`, `cp`, `mv`, `grep`.
- Programmes créés par un utilisateur : Généralement compilés à partir de code source écrit dans des langages de programmation comme le C. Par exemple, pour compiler un programme C :

```
$ gcc -Wall -Wextra -pedantic prog.c -o prog
```

Cette commande produit un fichier exécutable nommé `prog` que l'on exécutera en entrant la commande :

```
$ ./prog
```

Fichiers de données (« textes » ou « binaires »)

Les fichiers de données peuvent être textuels ou binaires.

Les fichiers texte contiennent des données lisibles par l'homme, comme des documents, des scripts, des fichiers de configuration, etc. Ils peuvent être créés et modifiés à l'aide d'éditeurs de texte tels que Vim.

Les fichiers binaires contiennent des données non textuelles, comme des images, des vidéos, des fichiers audio, des programmes compilés, etc. Par exemple, une image JPEG ou une vidéo MP4 sont des fichiers binaires.

Exemples

Pour créer un fichier texte, vous pouvez utiliser la commande suivante :

```
$ echo "Bonjour les Ensicaenais !" > hello.txt
```

Cela crée le fichier texte « `hello.txt` » contenant la chaîne « Bonjour les Ensicaenais ! ».

Pour afficher le contenu d'un fichier texte, utilisez :

```
$ cat hello.txt
```

Pour visualiser une image en utilisant le logiciel *ImageMagick*, la commande suivante peut être utilisée :

```
$ display image.png
```

Ces exemples montrent comment manipuler différents types de fichiers sous Unix, qu'ils soient textuels ou binaires.

Fichiers temporaires servant pour la communication

Les fichiers temporaires sont souvent utilisés pour stocker des données de manière transitoire ou pour permettre la communication entre différents processus. Ils doivent être supprimés après usage pour éviter d'occuper inutilement de l'espace disque.

Les fichiers temporaires classiques peuvent être créés dans des dossiers temporaires comme « `/tmp` » ou « `/var/tmp` ». Ils sont utilisés par les applications pour stocker des données intermédiaires ou des résultats de calculs.

Les tubes nommés (FIFO) sont des fichiers spéciaux utilisés pour la communication interprocessus. Les données écrites dans un tube nommé par un processus peuvent être lues par un autre processus.

Exemples

Pour créer un fichier temporaire, utilisez la commande suivante :

```
$ tempfile=$(mktemp /tmp/tempfile.XXXXXX)
$ echo "Some temporary data" > $tempfile
```

Cette commande crée un fichier temporaire avec un nom unique et y écrit des données temporaires.

Pour supprimer un fichier temporaire, utilisez :

```
$ rm $tempfile
```

Pour créer un tube nommé, utilisez la commande suivante :

```
$ mkfifo mypipe
```

Pour écrire et lire depuis un tube nommé, utilisez :

```
$ echo "Data to transmit" > mypipe &
$ cat mypipe
```

Ces commandes permettent de créer un tube nommé, d'y écrire des données avec la commande `echo` et de les lire avec la commande `cat`. Comme nous le verrons plus en détail dans la section 6.3, les tubes nommés facilitent la communication entre processus en fournissant un moyen simple de transmettre des données de manière séquentielle.

Fichiers journaux (logs)

Les fichiers journaux sont utilisés pour enregistrer des événements ou les messages d'une application ou du système d'exploitation. Ils sont essentiels pour le diagnostic des problèmes et l'analyse des performances.

Les fichiers journaux du système enregistrent des messages système et des événements. Par exemple, les fichiers journaux sous « `/var/log/` » contiennent des informations sur les activités du système, des services, des erreurs, etc.

Les fichiers journaux d'application sont utilisés par des applications spécifiques afin d'enregistrer des messages d'erreur, des informations de débogage, des traces d'exécution, etc.

Exemples

Pour afficher les dernières lignes d'un fichier journal du système, utilisez :

```
$ tail -f /var/log/syslog
```

Cette commande affiche les dernières lignes du fichier « `/var/log/syslog` » et met à jour l'affichage en temps réel au fur et à mesure que de nouvelles lignes sont ajoutées.

Pour rechercher une erreur spécifique dans un fichier journal, utilisez :

```
$ grep "ERROR" /var/log/application.log
```

Cette commande recherche toutes les occurrences du mot « `ERROR` » dans le fichier « `/var/log/application.log` », ce qui permet de cibler rapidement les messages d'erreur dans les journaux.

Fichiers de configuration

Les fichiers de configuration contiennent des paramètres et des options qui contrôlent le fonctionnement des logiciels et du système d'exploitation. Ils sont souvent situés dans des dossiers comme « `/etc` » pour les configurations système globales ou dans des dossiers utilisateur pour des configurations spécifiques à un utilisateur, tels que « `~/.config` ».

Exemples

Pour modifier le fichier de configuration du serveur Web Apache, utilisez la commande suivante :

```
$ sudo vim /etc/apache2/apache2.conf
```

Cette commande ouvre le fichier de configuration d'Apache avec les privilèges du superutilisateur, permettant de modifier les paramètres du serveur Web.

Pour modifier le fichier de configuration Bash d'un utilisateur, utilisez :

```
$ vim ~/.bashrc
```

Cette commande ouvre le fichier « `.bashrc` » dans l'éditeur de texte Vim, permettant de personnaliser les paramètres et les alias pour le *shell* Bash de l'utilisateur.

Les fichiers de configuration sont essentiels pour ajuster les comportements des applications et du système, offrant une grande flexibilité et une personnalisation pour les utilisateurs et les administrateurs système.

Fichiers de sauvegarde (***backups***)

Les fichiers de sauvegarde sont utilisés pour conserver des copies de données importantes afin de prévenir leur perte en cas de défaillance du système, d'erreurs humaines ou d'attaques malveillantes. Ils peuvent être stockés localement ou à distance.

Exemples

Pour créer une archive au format « tar » afin de sauvegarder un dossier, utilisez la commande suivante :

```
$ tar -cvzf backup.tar.gz /chemin/vers/le/dossier
```

Cette commande crée un fichier d'archive compressé nommé « `backup.tar.gz` » qui contient tous les fichiers et sous-dossiers du dossier spécifié.

Pour restaurer l'archive, utilisez :

```
$ tar -xvzf backup.tar.gz
```

Cette commande extrait le contenu de l'archive « `backup.tar.gz` » dans le dossier courant, recréant ainsi la structure des dossiers et fichiers préalablement sauvegardée.

Les sauvegardes régulières sont essentielles pour assurer la sécurité des données et permettre une récupération rapide en cas de problème.

Fichiers spéciaux

Les fichiers spéciaux sont utilisés pour représenter des périphériques ou des ressources système. Ils sont souvent situés dans le dossier « `/dev` » et incluent des fichiers de périphériques de blocs tels que les disques durs et des périphériques de caractères tels que les terminaux.

Exemples

Pour afficher le contenu d'un fichier de périphérique de caractères (terminal), utilisez la commande suivante :

```
$ cat /dev/tty
```

Cette commande affiche les entrées du terminal actuel.

Pour écrire dans un fichier de périphérique de blocs (par exemple, pour créer une image disque), utilisez :

```
$ dd if=/dev/zero of=disk_image.img bs=1M count=1024
```

Cette commande crée un fichier d'image disque de 1 Gio rempli de zéros. Le fichier « `disk_image.img` » peut ensuite être utilisé comme image de disque ou encore monté comme un système de fichiers.

Les fichiers spéciaux sont essentiels pour l'interaction avec le matériel et les ressources système, permettant aux utilisateurs et aux programmes d'accéder aux périphériques de manière uniforme et contrôlée.

4.5 Utilisation des fichiers depuis l'interface de commande

Les fichiers peuvent être manipulés directement depuis l'interface de commande Unix. Voici quelques opérations courantes :

4.5.1 Créer un fichier

Les fichiers sont le plus souvent créés par les applications, non directement dans l'interface de commande. Par exemple, les éditeurs de texte et les compilateurs créent des fichiers automatiquement lors de leur utilisation. Cependant, il est également possible de créer explicitement un fichier en utilisant des commandes spécifiques, ou en appelant la commande « `touch <nom_du_fichier>` » qui crée un fichier vide de 0 octet.

4.5.2 Détruire un fichier

Pour supprimer un fichier, utilisez la commande « `rm <nom_du_fichier>` ». Il est recommandé d'utiliser l'option « `-i` » pour demander une confirmation avant chaque suppression.

4.5.3 Créer un dossier

Pour créer un dossier, utilisez la commande « `mkdir <nom_du_dossier>` ». Le dossier sera initialement vide.

4.5.4 Détruire un dossier

Pour supprimer un dossier, utilisez la commande « `rmdir <nom_du_dossier>` ». Notez que le dossier doit être vide pour être supprimé.

4.5.5 Conventions dans les noms de fichiers

Certaines conventions sont utilisées pour désigner des fichiers ou des groupes de fichiers spécifiques :

- « * » désigne n'importe quelle chaîne de caractères. Par exemple, « `rm *.o` » détruit tous les fichiers dont le nom se termine par « .o », et « `ls *.c` » affiche la liste de tous les fichiers dont le nom se termine par « .c ».
- « ? » désigne n'importe quel caractère unique. Par exemple, « `ls test*.?` » affiche la liste de tous les fichiers dont le nom commence par « test » et dont l'extension contient un unique caractère.

Ces commandes et conventions permettent une gestion efficace et flexible des fichiers directement depuis l'interface de commande Unix.

4.6 Structure des fichiers sous Unix

Les fichiers sont organisés en **blocs** qui peuvent être dispersés à travers le support de stockage, comme l'illustre la figure 4.3.

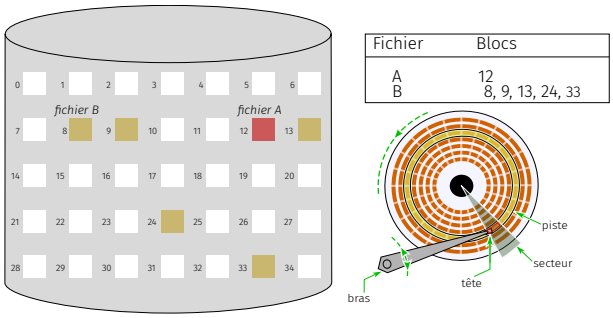


Fig. 4.3 : Blocs de fichier sur un disque

La taille d'un bloc sur un système Unix (typiquement 4 Kio) est récupérable à l'aide de la fonction `statvfs()`. Voici un exemple de récupération de la taille des blocs :

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/statvfs.h>

int main(void) {
    struct statvfs stat;
    int size

    size = statvfs("/", &stat);
```

```
    printf("File system block size: %lu bytes \n", (stat.f_bsize));

    return EXIT_SUCCESS;
}
```

Dans cet exemple, la fonction `statvfs()` est utilisée pour obtenir des informations sur le système de fichiers, y compris la taille des blocs. La structure `statvfs` contient plusieurs champs pertinents, mais ici nous nous concentrons sur le champ `f_bsize` qui renseigne sur la taille d'un bloc en octets. Si `statvfs()` échoue, le programme affiche un message d'erreur et se termine avec un échec.

4.7 Structure de donnée i-noeud

Les métadonnées des fichiers sont gérées par des structures appelées **i-noeuds** (ou **inodes**), qui contiennent des informations sur les fichiers, mais pas leurs noms.

Les noms de fichiers sont gérés dans les dossiers qui pointent vers ces i-noeuds. Cette séparation aide à optimiser les opérations comme renommer ou déplacer des fichiers sans altérer les données.

Les i-noeuds sont contenus dans une table globale maintenue par le système de gestion de fichiers de la même manière que la table des fichiers ouverts. Les pointeurs directs et indirects contiennent les adresses des blocs sur le support de stockage, comme le montre la figure 4.4.

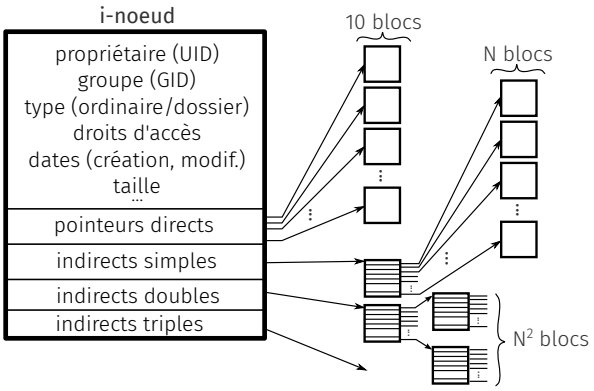


Fig. 4.4 : i-noeud d'un fichier.

L'avantage de cette structure est que les petits fichiers, plus nombreux, sont représentés de manière efficace.

Depuis un terminal Unix, il est possible d'obtenir des informations sur l'i-noeud d'un fichier à l'aide des commandes `ls -li` ou `stat`.

Exemples :

```
$ ls -li zero_sum_game.c
40664348 -rw-r--r--+ 1 jsaigne prof 1675 sept. 20 2018 zero_sum_game.c
```

ou

```
$ stat zero_sum_game.c
Fichier : 'zero_sum_game.c'
Taille: 1675   Blocs: 8      Blocs d'E/S : 16384  fichier
Périphérique : 2bh/43d  Inoeud : 40664348    Liens : 1
Accès : (0644/-rw-r--r--) UID : (200021/ jsaigne) GID : (2000/ prof)
```

```
Accès : 2018-09-20 10:17:18.352303900 +0200
Modif. : 2018-09-20 10:16:23.676303900 +0200
Changt : 2018-09-20 10:16:23.676303900 +0200
Créé : -
```

Ces commandes fournissent des informations détaillées sur l'i-noeud, telles que la taille du fichier, le nombre de blocs utilisés, le périphérique sur lequel se trouve le fichier, l'identifiant de l'i-noeud, le nombre de liens, ainsi que les informations d'accès et de modification.

4.8 Structure des dossiers sous Unix

Chaque entrée d'un dossier (fichier ou sous-dossier) pointe vers l'i-noeud correspondant comme le montre la figure 4.5.

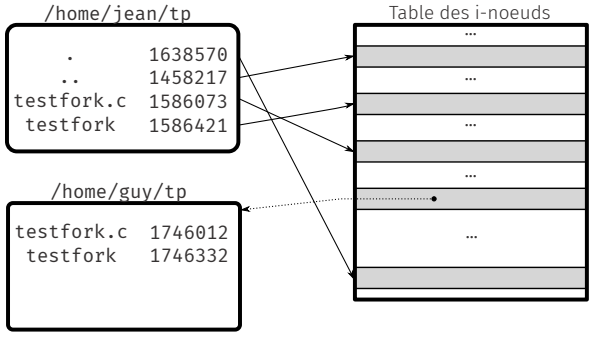


Fig. 4.5 : Dossiers et table des i-noeuds.

La commande `stat` permet aussi de récupérer l'i-noeud d'un dossier, par exemple le dossier personnel :

```
$ stat ~
Fichier : '/home/prof/jsaigne'
Taille : 0    Blocs : 0    Blocs d'E/S : 16384  répertoire
Périphérique : 2bh/43d  Inoeud : 40634743    Liens : 39
Accès : (0710/drwx--x---) UID : (200021/ jsaigne) GID : (2000/ prof)
Accès : 2018-05-15 03:18:06.060263000 +0200
Modif. : 2018-01-22 16:25:46.687307000 +0100
Changt : 2018-05-15 00:30:03.552263000 +0200
Créé : -
```

Cette commande affiche des informations détaillées sur le dossier, y compris la taille et le nombre de blocs (à 0), le périphérique, l'i-noeud, le nombre de liens, ainsi que les permissions et les horodatages d'accès, de modification et de changement.

Les i-noeuds et dossiers sont conservés en cache afin d'améliorer les performances du système de fichiers. Cela permet des accès plus rapides et une meilleure gestion des ressources.

4.9 Protection des fichiers sous Unix

Unix utilise un système de permissions pour sécuriser les fichiers. Les permissions sont divisées en lecture, écriture et exécution, appliquées à l'utilisateur, au groupe et aux autres. Un aspect important de la sécurité est l'utilisation de `setuid` ou `setgid`, qui permet à un exécutable de fonctionner avec les privilèges de son propriétaire ou de son groupe, respectivement. Toutefois, cela peut potentiellement exposer le système à des risques de sécurité.

Fichiers ordinaires

Les permissions sur les fichiers ordinaires sont divisées en trois catégories :

- Lecture (`r`) : Le fichier peut être lu.
- Écriture (`w`) : Le fichier peut être modifié.
- Exécution (`x`) : Le fichier peut être exécuté.

Ces permissions sont définies pour trois classes d'utilisateurs :

- utilisateur propriétaire (`u`) ;
- groupe propriétaire (`g`) ;
- tous les autres (`o`).

Les permissions peuvent être visualisées avec la commande « `ls -l` », et modifiées avec `chmod` (la commande ou l'appel système correspondant).

Exemple :

Pour ajouter la permission d'exécution pour le propriétaire :

```
$ chmod u+x fichier.txt
```

Pour définir des permissions spécifiques (`rw- r-- ---`) :

```
$ chmod 640 fichier.txt
```

Dossiers

Pour les dossiers, les permissions `r`, `w`, et `x` ont des significations légèrement différentes :

- Lecture (`r`) : Lister le contenu du dossier.
- Écriture (`w`) : Ajouter, supprimer, ou renommer des fichiers dans le dossier.
- Exécution (`x`) : Accéder aux fichiers et sous-dossiers (traverser le dossier).

Les permissions peuvent également être visualisées avec la commande « `ls -ld` » et modifiées avec `chmod` (la commande ou l'appel système correspondant).

Exemple :

Pour ajouter la permission d'exécution pour le groupe :

```
$ chmod g+x mon_dossier
```

Pour définir des permissions spécifiques (`rwx r-x ---`) :

```
chmod 750 mon_dossier
```

Commandes Unix associées

En plus de `chmod`, on peut utiliser les commandes Unix pour gérer la propriété des fichiers et dossiers :

- `chown` : changement du propriétaire du fichier ou dossier. Exemples :
  - Changer le propriétaire d'un fichier :
    ```
    $ chown nouvel_utilisateur fichier.txt
    ```
  - Changer le propriétaire et le groupe d'un fichier :
    ```
    $ chown nouvel_utilisateur:nouveau_groupe fichier.txt
    ```
- `chgrp` : changement du groupe du fichier ou dossier. Exemples :
  - Changer le groupe d'un fichier :
    ```
    $ chgrp nouveau_groupe fichier.txt
    ```

Mécanisme de délégation

- Problème : comment partager un programme dont l'exécution nécessite des droits d'accès que n'ont pas les utilisateurs courants ?
- Solution : un utilisateur reçoit temporairement les droits du propriétaire ou du groupe.
- Attention : ce mécanisme court-circuite le système de protection d'Unix ! Un utilisateur quelconque peut alors utiliser un programme avec les droits du superutilisateur.

Les permissions et les mécanismes de sécurité d'Unix permettent une gestion fine des accès aux fichiers et aux dossiers, assurant ainsi la protection des données et des systèmes contre les accès non autorisés et les abus.

4.10 Appels système d'accès aux fichiers

Au niveau de l'interface de programmation, un fichier est représenté par un **descripteur** entier (une entrée dans la table des fichiers ouverts).

Création ou ouverture

Avant d'utiliser un fichier, il faut le créer s'il n'existe pas ou l'ouvrir s'il existe déjà. Cette opération est réalisée par l'appel système `open()`, dont les prototypes sont les suivants sur Linux[1] :

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

int open(const char *pathname, int flags, mode_t mode);
int open(const char *pathname, int flags);
```

[1] Sur Mac OS X, un seul prototype est utilisé à la fois pour la création et l'ouverture : `int open(const char *path, int oflag, ...)` déclaré aussi dans « `fcntl.h` ».

Exemple de création de fichier

```
fd = open("/home/paul/tp/testfork.c", O_CREAT | O_RDWR,
          S_IRUSR | S_IWUSR | S_IRGRP | S_IR_OTH);
```

Dans cet exemple, le fichier est créé avec des droits en lecture et écriture pour le propriétaire du fichier, et en lecture seule pour les membres du groupe ainsi que les autres. Le descripteur de fichier alloué par le système est stocké dans la variable `fd` et correspond à la première entrée libre dans la table des fichiers ouverts (nous parlerons plus loin de la « table des descripteurs »). Si l'opération réussit, la valeur retournée est un nombre strictement supérieur à **2**; sinon la valeur retournée est -1.

Le paramètre `flags` permet de spécifier les modes d'accès au fichier. Les valeurs possibles sont des combinaisons des constantes symboliques définies dans l'entête « `fcntl.h` » :

`flags`	Description
`O_RDONLY`	Ouverture en lecture seule
`O_WRONLY`	Ouverture en écriture seule
`O_RDWR`	Ouverture en lecture et écriture
`O_CREAT`	Création du fichier s'il n'existe pas
`O_EXCL`	Erreur si `O_CREAT` est également spécifié et le fichier existe
`O_NOCTTY`	Ne pas assigner le terminal de ce fichier
`O_TRUNC`	Tronquer le fichier à une longueur de `0`
`O_APPEND`	Ajouter des données à la fin du fichier
`O_NONBLOCK`	Ouverture en mode non-bloquant
`O_DSYNC`	Les mises à jour d'I/O sont synchrones
`O_SYNC`	Les mises à jour d'I/O sont synchrones
`O_RSYNC`	Les lectures attendent les mises à jour en attente
`O_CLOEXEC`	Ferme le descripteur lors de l'exécution d'un autre programme

Le paramètre `mode` quant à lui, permet de définir les droits d'accès. Afin d'offrir une grande flexibilité dans la gestion de ces droits sur différents systèmes Unix, la norme POSIX a introduit les constantes symboliques suivantes :

Constante	Description
`S_IRWXU`	Le propriétaire a les droits en lecture, écriture et exécution
`S_IRUSR`	Le propriétaire a le droit en lecture
`S_IWUSR`	Le propriétaire a le droit en écriture
`S_IXUSR`	Le propriétaire a le droit en exécution
`S_IRWXG`	Le groupe a les droits en lecture, écriture et exécution
`S_IRGRP`	Le groupe a le droit en lecture
`S_IWGRP`	Le groupe a le droit en écriture
`S_IXGRP`	Le groupe a le droit en exécution

Constante	Description
S_IRWXO	Tous les autres ont les droits en lecture, écriture et exécution
S_IROTH	Tous les autres ont le droit en lecture
S_IWOTH	Tous les autres ont le droit en écriture
S_IXOTH	Tous les autres ont le droit en exécution

Exemple d'ouverture d'un fichier pour le lire

```
fd = open("/home/paul/tp/testfork.c", O_RDONLY);
```

Dans cet exemple, le fichier est ouvert en lecture seule.

Le paramètre `flags` permet de spécifier les modes d'accès au fichier. Les valeurs possibles sont : `O_RDWR` pour ouvrir le fichier en lecture et écriture, `O_WRONLY` pour l'ouvrir en écriture seule, et `O_RDONLY` pour l'ouvrir en lecture seule.

L'utilisation judicieuse des différents modes et droits d'accès permet de garantir une gestion fine et sécurisée des fichiers, assurant que seules les opérations autorisées peuvent être effectuées selon les besoins spécifiques de l'application.

Fermeture d'un fichier

Lorsqu'on a fini d'utiliser un fichier, il est essentiel de le fermer pour libérer une des entrées dans la table des fichiers ouverts. C'est le rôle de l'appel système `close()`, dont le prototype est le suivant :

```
#include <unistd.h>

int close(int fd);
```

Exemple d'utilisation :

```
if (close(fd) == -1) {
    perror("Error to close file.");
    exit(EXIT_FAILURE);
}
```

Une fois fermé, le descripteur `fd` n'est plus utilisable et pourra être réalloué par le système pour un autre fichier. La fermeture appropriée des fichiers permet de libérer les ressources et d'éviter les fuites de descripteurs de fichiers, ce qui est nécessaire afin de maintenir la stabilité et la performance du système.

Déplacement dans un fichier

L'ouverture d'un fichier crée un pointeur courant (position dans le fichier), initialisé à 0, et déplacé :

- indirectement, par les appels de lecture `read()` et d'écriture `write()`;
- directement, par appel de `lseek()` qui déplace le pointeur courant.

L'appel système `lseek()` a pour prototype :

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

Le paramètre `whence` caractérise la manière dont le déplacement d'« `offset` » octets va se faire dans le fichier. `whence` peut prendre entre autres les valeurs :

- `SEEK_CUR` pour un déplacement relatif par rapport au pointeur courant ;
- `SEEK_SET` pour un déplacement absolu par rapport au début du fichier ;
- `SEEK_END` pour un déplacement en fin de fichier.

Comme le montre l'exemple de la figure 4.6, le pointeur peut être placé au-delà de la fin du fichier.

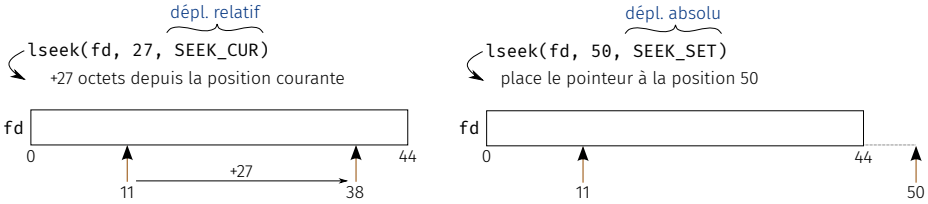


Fig. 4.6 : Exemples de déplacement du pointeur de fichier‘.

Exemple d'utilisation :

Déplacement relatif par rapport au pointeur courant :

```
off_t new_pos = lseek(fd, 100, SEEK_CUR);
```

Déplacement absolu par rapport au début du fichier :

```
off_t new_pos = lseek(fd, 50, SEEK_SET);
```

Déplacement par rapport à la fin du fichier :

```
off_t new_pos = lseek(fd, -10, SEEK_END);
```

Ces déplacements permettent de lire ou décrire à des positions spécifiques dans le fichier, offrant une grande flexibilité pour la gestion des fichiers. Utiliser `lseek()` de manière appropriée est donc essentiel pour manipuler les fichiers efficacement dans les programmes Unix.

Lecture dans un fichier

L'appel système `read()` permet de lire `nbyte` octets dans un fichier et de les placer dans un tampon. Son prototype est le suivant :

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t nbyte);
```

Si l'appel système réussit, alors la valeur de retour correspond au nombre d'octets lus, et s'il échoue alors la valeur retournée est -1.

La figure 4.7 présente deux cas de figure en fonction de la position du pointeur de fichier au moment de la lecture.

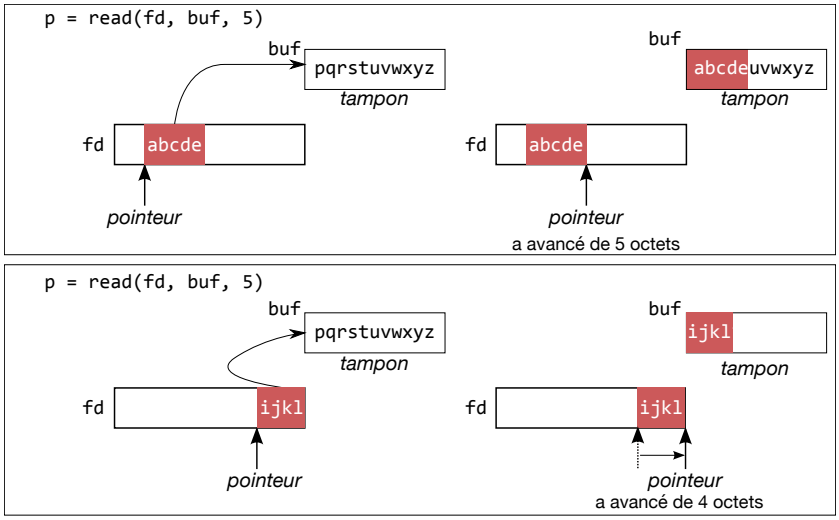


Fig. 4.7 : Exemples de lecture dans un fichier.

Exemple d'utilisation :

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

int main(void) {
    int fd;
    char buffer[100];
    ssize_t bytes_read;

    /* Open a file to read it only */
    fd = open("example.txt", O_RDONLY);
    if (fd == -1) {
        perror("Error to open file.");
        exit(EXIT_FAILURE);
    }

    /* Read the content of the file */
    bytes_read = read(fd, buffer, sizeof(buffer) - 1);
    if (bytes_read == -1) {
        perror("Error to read file.");
        close(fd);
        exit(EXIT_FAILURE);
    }

    /* Add '\0' at the end of the buffer */
    buffer[bytes_read] = '\0';
    printf("Content of the file:\n%s\n", buffer);

    /* Close file */
    if (close(fd) == -1) {
        perror("Error to close file.");
        exit(EXIT_FAILURE);
```

```
    }

    return EXIT_SUCCESS;
}
```

Dans cet exemple, le fichier « example.txt » est ouvert en lecture seule. Le contenu du fichier est lu et stocké dans le tampon `buffer`, qui est ensuite affiché. Le programme vérifie également les erreurs possibles lors de l'ouverture et de la lecture du fichier.

Écriture dans un fichier

L'appel système `write()` permet d'écrire les `nbyte` octets d'un tampon dans un fichier. Son prototype est le suivant :

```
#include <unistd.h>
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

Si l'appel système réussit, alors la valeur de retour correspond au nombre d'octets écrits, et s'il échoue alors la valeur retournée est -1.

La figure 4.8 présente deux cas de figure en fonction de la position du pointeur de fichier au moment de l'écriture.

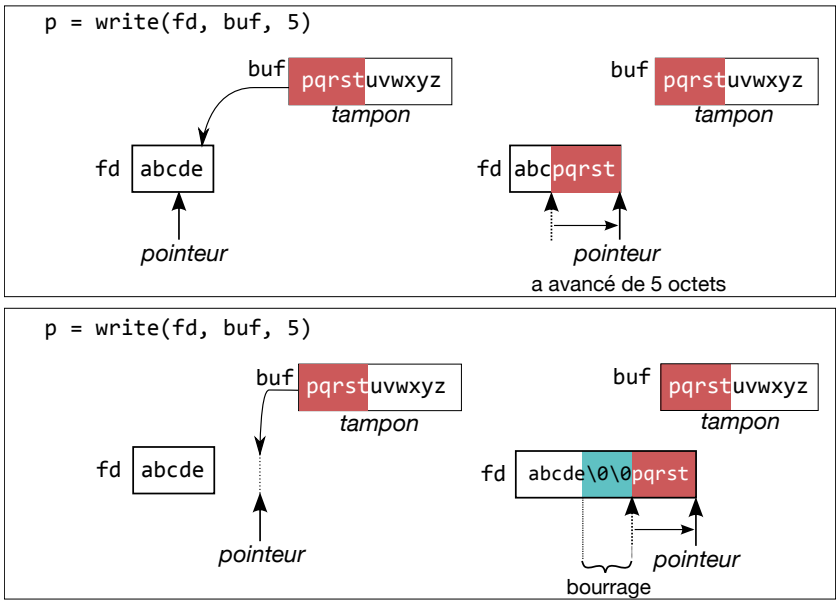


Fig. 4.8 : Exemples d'écriture dans un fichier.

Exemple d'utilisation :

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int fd;
    ssize_t bytes_written;
```

```
    const char *text = "Hello les Ensicaennais!\n";

    fd = open("example.txt", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
    if (fd < 0) {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }

    /* Write the content of the text buffer */
    bytes_written = write(fd, text, sizeof(text) - 1);
    if (bytes_written < 0) {
        perror("Error writing to file");
        close(fd);
        exit(EXIT_FAILURE);
    }

    printf("Wrote %zd bytes to the file.\n", bytes_written);

    close(fd);
    return EXIT_SUCCESS;
}
```

Dans cet exemple, le fichier « example.txt » est ouvert en mode écriture. Le texte « `Hello les Ensicaennais!\n` » est écrit dans le fichier. Le programme vérifie également les erreurs possibles lors de l'ouverture et de l'écriture du fichier.

4.10.1 Bibliothèque standard du C

Les appels système fournis par le noyau (`open()`, `close()`, `lseek()`, `read()` et `write()`) sont de bas niveau. Leur utilisation peut se révéler délicate (gestion des erreurs, lectures tronquées, etc.). On préférera si possible les fonctions de la bibliothèque C dite "standards" qui encapsulent la gestion des erreurs et les complexités des appels système. Elles offrent des fonctionnalités plus avancées et sont généralement plus faciles à utiliser, avec une gestion automatique des tampons et une interface plus conviviale.

- pour les fichiers : `fopen()`, `fread()`, `fwrite()`, `fscanf()`, `fprintf()`, `fflush()`, `fseek()`, `fclose()`, etc.
- pour les chaînes de caractères : `sprintf()`, `sscanf()`, etc.

Exemple d'utilisation des fonctions de la bibliothèque standard du C :

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

int main(void) {
    FILE *file;
    char buffer[100];

    file = fopen("example.txt", "r");
    if (file == NULL) {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }

    if (fgets(buffer, sizeof(buffer), file) != NULL) {
```

```
        printf("Read from file: %s\n", buffer);
    } else {
        perror("Error reading file");
    }

    if (fclose(file) != 0) {
        perror("Error closing file");
        exit(EXIT_FAILURE);
    }

    return EXIT_SUCCESS;
}
```

Dans cet exemple, la fonction `fopen()` est utilisée pour ouvrir un fichier en mode lecture, `fgets()` pour lire une ligne de texte à partir du fichier, et `fclose()` pour fermer le fichier. La gestion des erreurs est simplifiée grâce aux fonctions de la bibliothèque standard du C, rendant le code plus lisible et plus facile à maintenir.

Il est possible à partir du pointeur sur la structure `FILE` de récupérer le descripteur du fichier en utilisant la fonction POSIX `fileno()` dont le prototype est le suivant :

```
#include <stdio.h>
int fileno(FILE *file);
```

4.11 Fichiers et flux d'entrées/sorties

Il est habituel de dire que sous Unix, tout est fichier. Il existe effectivement un lien étroit entre fichiers et entrées/-sorties. Ainsi, Les périphériques d'entrées/sorties sont représentés par des fichiers particuliers présents dans le dossier « `/dev` ».

Les **flux d'entrées-sorties** représentent des canaux de communication entre le processus et son environnement extérieur, pouvant être redirigés vers des fichiers ou d'autres périphériques. Ces flux sont essentiels pour le traitement des données entrantes et sortantes dans les programmes Unix. Ainsi, par défaut, le flux d'entrée standard (`stdin`) est associé au clavier, et les flux de sortie (`stdout`et `stderr`) le sont à l'écran.

Par convention, ces flux sont associés aux descripteurs **`0`**, **`1`** et **`2`**, associés aux constantes symboliques POSIX `STDIN_FILENO`, `STDOUT_FILENO` et `STDERR_FILENO` déclarées dans « `unistd.h` ».

La figure 4.9 illustre la **table des descripeurs** d'un processus telle qu'elle est associée à la table des fichiers ouverts, lui même en lien avec la table des i-noeuds. Cette configuration va faciliter la redirection des entrées/sorties de tout processus comme nous allons le voir plus loin.

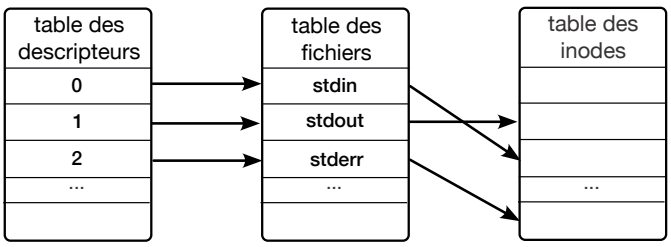


Fig. 4.9 : Entrée/sorties d'un processus et table des descripteurs.

4.12 Manipuler les flux d'entrées/sorties (commandes)

Depuis l'interface de commande, on réoriente les flux standards au moyen des opérateurs < et >.

Quelques exemples :

- `cat fic` écrit le contenu de `fic` sur la sortie standard.
- `cat fic > fic1` copie le contenu de `fic` dans `fic1`. `fic1` est créé s'il n'existe pas.
- `cat /dev/null > fic` crée le fichier vide `fic` s'il n'existe pas, et en supprime le contenu sinon.

Ces opérations permettent de rediriger facilement les flux d'entrée et de sortie pour manipuler les fichiers directement depuis le terminal. Nous allons voir dans la section suivante, que deux appels système nous permettent de réaliser ces redirections.

4.13 Redirection et copie de descripteurs

L'échange des descripteurs dans la table des descripteurs est réalisé à l'aide des appels systèmes `dup()` et `dup2()` dont les prototypes sont les suivants :

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

L'appel système `dup(oldfd)` recopie le descripteur `oldfd` dans le premier emplacement disponible de la table. Cela permet de dupliquer un descripteur existant en utilisant la première entrée libre dans la table des descripteurs.

L'appel système `dup2(oldfd, newfd)` recopie le descripteur `oldfd` dans le descripteur `newfd`. Si `newfd` est déjà ouvert, il est d'abord fermé. Cette opération est utile pour rediriger les flux d'entrées/sorties.

La figure 4.10 présente un exemple de redirection telle qu'elle serait réalisée par le morceau de code suivant :

```
fd = open("temp.txt", ...);
dup2(fd, STDOUT_FILENO); /* Copy fd in the descriptor for stdout */
printf("Bonjour\n");     /* The string is redirected to temp.txt */
close(fd)
```

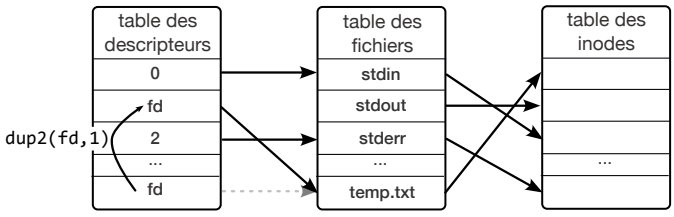


Fig. 4.10 : Exemple de redirection vers un fichier.

La table des descripteurs d'un processus parent est transmise à l'ensemble de ses processus enfants (de même pour les redirections effectuées).

4.14 Exercices

Exercice 1 : Taille maximale d'un fichier

Un système de fichiers Unix permet de définir des blocs de taille maximale de 512 octets. L'adresse d'un bloc est composée de 4 octets et un i-noeud contient 10 pointeurs directs de blocs, un pointeur indirect simple et un pointeur indirect double.

Quelle est alors la taille maximale d'un fichier (1 Kio = 2^{10} octets) ?

Étapes :

1. Calculez le nombre total d'octets pouvant être adressés par les pointeurs directs.
2. Calculez le nombre total d'octets pouvant être adressés par le pointeur indirect simple.
3. Calculez le nombre total d'octets pouvant être adressés par le pointeur indirect double.
4. Additionnez ces valeurs pour obtenir la taille maximale du fichier.

Exercice 2 : Des blocs et des blocs

Combien de blocs sont nécessaires pour stocker un fichier de 2 128 Kio sur le système de fichiers précédent? Estimer l'espace perdu sur le disque par ce fichier.

Exercice 3 : Ping et Pong 1

Soit le code suivant :

```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
int main(void) {
  int fd1, fd2;

  fd1 = open("ping.txt", O_RDONLY, 0);
  close(fd1);
  fd2 = open("pong.txt", O_RDONLY, 0);
  printf("fd2 = %d\n", fd2);

  return EXIT_SUCCESS;
}
```

1. Qu'affiche-t-il lorsque `ping.txt` et `pong.txt` existent tous les deux?
2. *Idem* si `ping.txt` n'existe pas?
3. Et si `pong.txt` n'existe pas?

Exercice 4 : Ping et Pong 2

Cet exercice fait intervenir un tube anonyme. Nous n'avons pas encore clairement décrit comment créer un mécanisme de ce type, mais sachez déjà que celui-ci possède une entrée et une sortie, et nécessite donc deux descripteurs que l'on place dans un tableau.

Voici le code associé à cet exercice :

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main(void) {
  int fd1, fd2;
  int dp[2]; /* array to store input and output descriptors of the pipe */

  fd1 = open("ping.txt", O_RDONLY, 0);
  pipe(dp); /* We create an anonymous pipe... */
  fd2 = open("pong.txt", O_RDONLY, 0);
  printf("fd2 = %d\n", fd2);

  return EXIT_SUCCESS;
}
```

1. Qu'affiche-t-il lorsque `ping.txt` et `pong.txt` existent tous les deux ?
2. *Idem* si `ping.txt` n'existe pas ?
3. Et si `pong.txt` n'existe pas ?

Exercice 5 : Manipulation des flux d'entrées/sorties

Écrivez un programme en C qui redirige la sortie standard vers un fichier et la sortie d'erreur vers un autre fichier.

Exercice 6 : Création d'un lien symbolique

Écrivez un script Bash qui crée un lien symbolique vers un fichier et vérifie que le lien fonctionne correctement.

5 Gestion de la mémoire virtuelle

5.1 Introduction

La gestion de la mémoire virtuelle est fondamentale dans les systèmes d'exploitation modernes. Elle permet l'exécution de processus sans nécessiter leur chargement complet en mémoire, optimisant ainsi les ressources et améliorant les performances globales. Grâce à cette technique, les systèmes peuvent exécuter des programmes de grande taille en utilisant efficacement l'espace mémoire disponible et en offrant une abstraction de la mémoire physique sous-jacente.

5.2 Hiérarchie des mémoires

Les différentes catégories de mémoire d'un système informatique sont hiérarchisées selon leur taille, leur temps d'accès et leur coût par unité de stockage. En haut de la hiérarchie se trouvent les registres, suivis du cache, de la mémoire principale, et enfin de la mémoire secondaire. Les registres sont extrêmement rapides, mais très limités en taille, tandis que la mémoire secondaire offre une grande capacité de stockage, mais avec des temps d'accès beaucoup plus longs. La figure 5.1 inspirée de Bryant et O'Hallaron illustre cette hiérarchie [5].

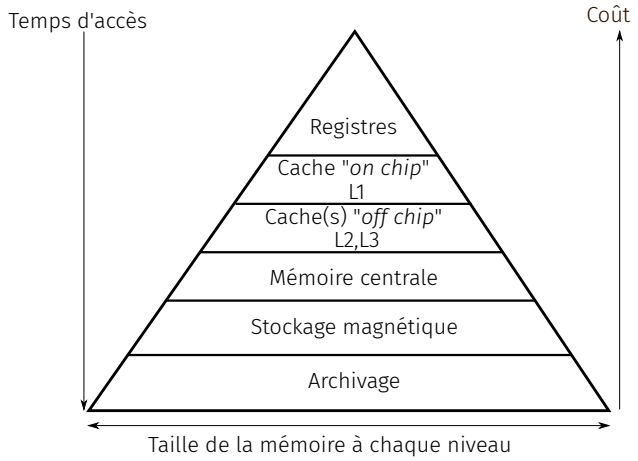


Fig. 5.1 : Hiérarchie des mémoires [5].

Le tableau suivant tiré de Bryant et O'Hallaron fournit une comparaison des tailles, temps d'accès, coûts et unités de transfert pour les principaux types de mémoire [5].

	Registre	Cache	Mémoire	Mémoire secondaire
taille	32/64 bits	32 Kio -- 4 Mio	16 Go	3 To
temps d'accès (ns)	1	1,3	20	3000000

	Registre	Cache	Mémoire	Mémoire secondaire
$/Mo	--	25	0.02	0.03
unité de transfert	8 o	32 / 64 o	4 Kio	--

Les opérations de manipulation des données s'effectuent principalement en mémoire centrale, et le cache joue un rôle déterminant en accélérant les accès. Par ailleurs, l'essentiel des données réside sur le disque. En conséquence, la performance globale du système repose en grande partie sur l'efficacité des transferts entre la mémoire centrale et le disque.

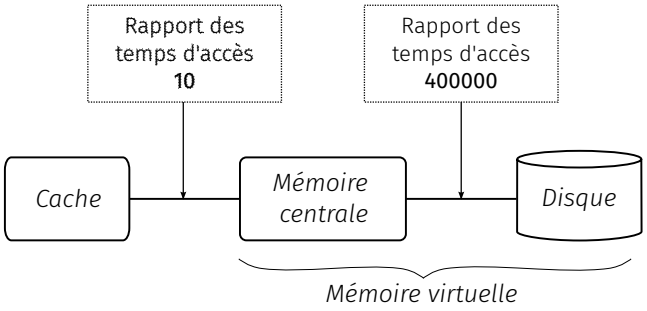


Fig. 5.2 : Rapports de temps d'accès selon [5].

5.3 Motivations pour une mémoire virtuelle

Exiger que la totalité d'un processus soit en mémoire centrale pour son exécution présente plusieurs inconvénients. Premièrement, cela conduit souvent à une sous-utilisation de la mémoire, car un processus n'utilise jamais toutes ses instructions et données simultanément. Deuxièmement, cela empêche l'exécution de processus dont la taille dépasserait celle de la mémoire centrale. La mémoire virtuelle permet de surmonter ces limitations en utilisant le disque dur afin d'y stocker les parties inactives d'un processus, optimisant ainsi l'utilisation des ressources et permettant l'exécution de processus de plus grande taille.

5.4 Mémoire virtuelle

La mémoire virtuelle est donc un mécanisme qui utilise le disque dur pour permettre l'exécution de processus sans que ceux-ci soient entièrement présents en mémoire centrale. Elle distingue les adresses **virtuelles** (ou logiques), utilisées par les processus, des adresses **physiques** (ou réelles) dans la mémoire centrale. La gestion de la mémoire permet le **partage** de la mémoire physique entre les processus et le **calcul des adresses** de manière à transformer une adresse virtuelle en adresse physique et *vice versa*.

Toutefois, la taille de la mémoire virtuelle ne peut pas dépasser la capacité d'adressage du processeur. Par exemple, un système avec un processeur 32 bits peut avoir une mémoire virtuelle de 4 Gio même s'il ne dispose que de 1 Gio de RAM. Pour un processeur 64 bits, la mémoire virtuelle monte à 16 Eio, soit 16×2^{60} octets.

5.5 Structuration des espaces d'adressage

L'espace d'adressage virtuel d'un processus représente l'ensemble des adresses qu'il peut référencer. Cet espace est structuré en blocs, qui peuvent être de tailles fixes (**pages**) ou de tailles variables (**segments**). Les deux organisations peuvent être combinées, comme dans le cas d'un segment formé de plusieurs pages. L'espace d'adressage physique, quant à lui, est structuré en **cadres** de même taille que les pages.

Structuration par pages

Dans une organisation paginée, l'espace d'adressage virtuel est divisé en pages de taille fixe, typiquement 4 Kio, bien que d'autres tailles soient possibles. Chaque page virtuelle est associée à un cadre physique de même taille. Cette approche permet une gestion efficace de la mémoire en réduisant la fragmentation et en facilitant le transfert de pages entre la mémoire centrale et le disque.

Structuration par segments

Dans une organisation segmentée, l'espace d'adressage est divisé en segments de tailles variables, chaque segment correspondant à une région logique du programme, telle que le code, les données, ou la pile. Les segments permettent de refléter plus précisément la structure logique d'un programme, mais peuvent entraîner une fragmentation externe et une gestion plus complexe de la mémoire.

Combinaison de pages et de segments

La combinaison des deux approches permet de tirer parti des avantages des deux systèmes. Un segment peut être divisé en plusieurs pages, permettant une gestion fine de la mémoire tout en respectant la logique segmentée du programme. Cette méthode hybride est couramment utilisée dans les systèmes d'exploitation modernes pour optimiser la gestion de la mémoire et améliorer les performances.

5.6 Obtenir la taille des pages sur un système

La fonction `getpagesize()` permet de connaître la taille des pages du système. Son prototype est le suivant :

```
#include <unistd.h>
int getpagesize(void);
```

L'exemple ci-dessous montre comment utiliser `getpagesize()` de manière à afficher la taille des pages du système.

```
#include <stdio.h>   /* printf() */
#include <stdlib.h>  /* exit() */
#include <unistd.h>  /* fork() and getpagesize() */

int main(void) {
    printf("The page size on this system is: %d bytes\n", getpagesize());
    return EXIT_SUCCESS;
}
```

5.7 Mémoire virtuelle paginée

Chaque processus dispose d'une mémoire virtuelle **indépendante**, constituée d'un ensemble de pages qui représentent son **espace adressable**. La conversion des adresses virtuelles en adresses physiques est réalisée en utilisant une **table des pages**. Cette table contient les numéros des cadres et est indexée par les numéros de pages comme l'illustre la figure 5.3.

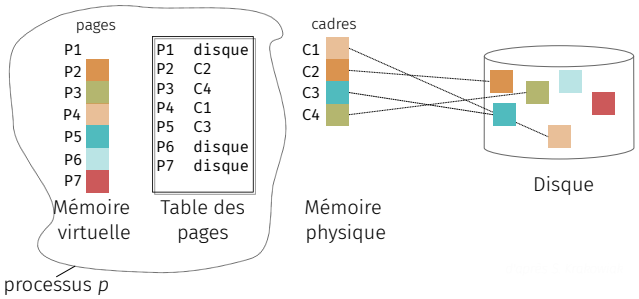


Fig. 5.3 : Mémoire virtuelle et table des pages.

Une page virtuelle dont le contenu est en mémoire physique est immédiatement accessible. Si son contenu est sur disque, il doit d'abord être transféré en mémoire physique pour être accessible. Il faut alors lui trouver un cadre libre ; s'il n'y en a aucun, il faut en libérer un en copiant son contenu sur le disque s'il a changé.

Le système d'exploitation garantit que toute page virtuelle utilisée par le processeur sera transférée en temps utile du disque en mémoire physique, et que ce transfert sera transparent pour les utilisateurs.

5.8 Principe de la conversion d'adresse

La conversion des adresses virtuelles en adresses physiques est réalisée par l'**unité de gestion de mémoire** (*memory management unit* ou **MMU**) appelée parfois **PMMU** (*paged MMU*)[1]. La MMU vérifie si l'adresse virtuelle correspond à une adresse de la mémoire physique en consultant la table des pages. Si c'est le cas, elle transmet l'adresse réelle sur le bus de la mémoire ; sinon, il y a un **défaut de page**. Un défaut de page provoque une interruption dont le rôle est de rapatrier la page manquante depuis le disque. La conversion est accélérée grâce à une mémoire associative appelée **TLB** (*translation lookaside buffer*) comme le montre la figure 5.4.

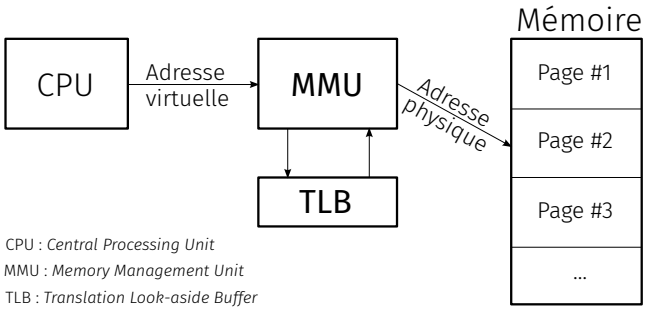


Fig. 5.4 : Conversion d'adresse avec MMU et TLB.

[1] La MMU est le composant qui contrôle les accès qu'un processeur fait à la mémoire de l'ordinateur (se référer au cours d'architecture des ordinateurs en première année).

Le TLB stocke les correspondances récentes entre les adresses virtuelles et physiques. Ce mécanisme accélère la traduction des adresses lors des accès à la mémoire, car il permet d'éviter une recherche coûteuse dans la table des pages à chaque accès. Le TLB est couramment organisé avec une stratégie de mise en cache de type LRU (*least recently used*) ou FIFO, afin de conserver les entrées les plus fréquemment utilisées. Des études montrent que le TLB peut réduire le temps d'accès mémoire de 20 à 25 % pour des scénarios d'utilisation typiques, ce qui est non négligeable pour les performances globales du système [16 -- annexe B].

Lorsque la MMU reçoit une adresse virtuelle, elle contrôle d'abord si le numéro de la page virtuelle est présent dans le TLB. Si c'est le cas et si le mode d'accès est conforme aux bits de protection, le numéro de cadre est récupéré directement du TLB. Sinon, il y a un défaut de protection ou un **défaut de page**. En cas de défaut de page, la MMU accède à la table des pages. Si le bit de présence est à 1, la MMU remplace une des entrées du TLB par celle qu'elle a trouvée. Sinon, elle provoque un défaut de page ou une erreur de segmentation si l'entrée est incorrecte. Lors d'un défaut de page, le processus demandeur est bloqué jusqu'à ce que la page soit chargée en mémoire, après quoi la table des pages est mise à jour et le processus est réveillé.

On pourrait penser que plus il y a de mémoire centrale et moins il y a moins de défauts de pages. Il s'agit d'une intuition trompeuse. Cette anomalie a été mise en évidence par Bélády et ses co-auteurs dans le cas d'une stratégie de remplacement de type FIFO [2].

5.9 Segmentation et mémoire virtuelle d'un processus

La mémoire virtuelle d'un processus se décompose en espaces d'adressage appelés **segments mémoires**, de tailles variables et pas forcément contiguës. Les segments sont associés à des ensembles de pages. Sous Unix, un processus possède plusieurs segments : un segment contenant les instructions (*text segment*), un segment contenant les données initialisées (*data segment*), un segment contenant les données statiques non initialisées (*BSS segment*), un segment avec la pile d'exécution (*call stack*), ainsi qu'un segment appelé tas (*heap*) si des allocations dynamiques sont réalisées. Chaque segment est associé à des droits d'accès (`rwx`) et il peut être privé (`p`) ou partagé (`s`). La figure 5.5 illustre cette décomposition de l'espace mémoire d'un processus en segments.

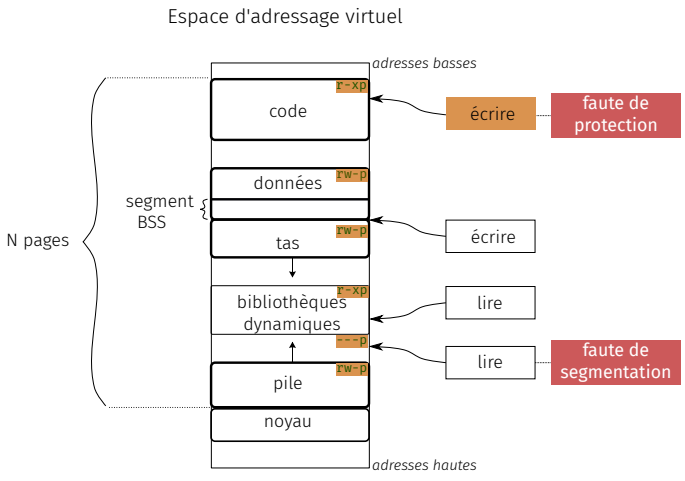


Fig. 5.5 : Segments mémoire d'un processus.

5.10 Visualisation des segments sous Unix

Pour chaque processus créé, le système crée dans le dossier « `/proc` » un sous-dossier dont le nom est le PID du processus. Dans ce dossier, on trouve notamment le fichier « `maps` » qui spécifie la cartographie mémoire du processus. Cette cartographie montre comment la mémoire virtuelle du processus est structurée, y compris les différents segments dans cette mémoire.

Considérons l'exemple du programme suivant :

```
#include <stdio.h>     /* printf() */
#include <unistd.h>    /* getpid() */
#include <time.h>      /* time() */

int main(void) {
    time_t end = time(NULL) + 240; /* about 4 min */

    printf("Process with PID %d\n", getpid());
    while (1) {
        if (time(NULL) >= end) {
            break;
        }
    }

    return EXIT_SUCCESS;
}
```

Alors sa cartographie mémoire aura la configuration suivante :

```
$ cat /proc/4155/maps
address           perms offset   time  inode     pathname
00400000-00401000 r-xp 00000000 08:04 2886969 /home/prof/jsaigne/mem_02
00600000-00601000 r--p 00000000 08:04 2886969 /home/prof/jsaigne/mem_02
00601000-00602000 rw-p 00001000 08:04 2886969 /home/prof/jsaigne/mem_02
7f88d2ac1000-7f88d2c3b000 r-xp 00000000 08:03 262191 /lib/libc-2.11.1.so
...
7f88d2e3e000-7f88d2e3f000 rw-p 0017d000 08:03 262191 /lib/libc-2.11.1.so
7f88d2e3f000-7f88d2e44000 rw-p 00000000 00:00 0
7f88d30c7000-7f88d30e7000 r-xp 00000000 08:03 262166 /lib/ld-2.11.1.so
7f88d32af000-7f88d32b2000 rw-p 00000000 00:00 0
7f88d32e4000-7f88d32e6000 rw-p 00000000 00:00 0
7f88d32e6000-7f88d32e7000 r--p 0001f000 08:03 262166 /lib/ld-2.11.1.so
7f88d32e7000-7f88d32e8000 rw-p 00020000 08:03 262166 /lib/ld-2.11.1.so
7f88d32e8000-7f88d32e9000 rw-p 00000000 00:00 0
7fff03f3e000-7fff03f53000 rw-p 00000000 00:00 0      [stack]
```

Visualisation du segment BSS

La commande Unix « `size` » renseigne sur la taille des différents segments d'un programme :

```
$ size mem_02
   text   data    bss    dec    hex   filename
   1472    576      8   2056    808   ./mem_02
```

Déclarons une variable statique dans la fonction principale précédente :

- `static int temperatures[10];`

Alors on peut constater l'augmentation du segment BSS :

```
$ size mem_02
   text   data    bss    dec    hex   filename
   1472    576     72   2120    848   memory_02_2
```

Initialisons cette variable statique :

- `static int temperatures[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};`

Le segment BSS retrouve sa taille initiale, et la donnée est transférée dans le segment de données « data » :

```
$ size mem_02
   text   data    bss    dec    hex   filename
   1472    632      8   2112    840   memory_02_3
```

Visualisation des bibliothèques partagées

Le programme suivant utilise la fonction mathématique « `sqrt()` » nécessitant le chargement de la bibliothèque mathématique lors de son exécution (en effet la version dynamique de la bibliothèque est privilégiée lors de la compilation).

```c
#include <stdio.h>     /* printf() */
#include <stdlib.h>    /* random() */
#include <unistd.h>    /* getpid() */
#include <sys/types.h> /* pid_t */
#include <math.h>      /* sqrt() -- link with -lm */

int main(void) {
    long value;
    double square_root;

    printf("Process with PID %d\n", getpid());
    while (1) {
        value = random();
        square_root = sqrt(value);
    }

    return EXIT_SUCCESS; /* unreachable code */
}
```

Supposons que le programme précédent se voit allouer le PID 4155 par le système d'exploitation lors de son exécution, alors la cartographie mémoire du processus aura par exemple la configuration suivante :

```
$ cat /proc/4155/maps
address           perms  offset   time inode     pathname
00400000-00401000 r-xp 00000000 08:04 2886969 /home/prof/jsaigne/mem_03
00600000-00601000 r--p 00000000 08:04 2886969 /home/prof/jsaigne/mem_03
00601000-00602000 rw-p 00001000 08:04 2886969 /home/prof/jsaigne/mem_03
7f88d2ac1000-7f88d2c3b000 r-xp 00000000 08:03 262191 /lib/libc-2.11.1.so
...
7f88d2e3e000-7f88d2e3f000 rw-p 0017d000 08:03 262191 /lib/libc-2.11.1.so
7f88d2e3f000-7f88d2e44000 rw-p 00000000 00:00 0
```

```
7f88d2e44000-7f88d2ec6000 r-xp 00000000 08:03 262240 /lib/libm-2.11.1.so
...
7f88d30c6000-7f88d30c7000 rw-p 00082000 08:03 262240 /lib/libm-2.11.1.so
7f88d30c7000-7f88d30e7000 r-xp 00000000 08:03 262166 /lib/ld-2.11.1.so
7f88d32af000-7f88d32b2000 rw-p 00000000 00:00 0
7f88d32e4000-7f88d32e6000 rw-p 00000000 00:00 0
7f88d32e6000-7f88d32e7000 r--p 0001f000 08:03 262166 /lib/ld-2.11.1.so
7f88d32e7000-7f88d32e8000 rw-p 00020000 08:03 262166 /lib/ld-2.11.1.so
7f88d32e8000-7f88d32e9000 rw-p 00000000 00:00 0
7fff03f3e000-7fff03f53000 rw-p 00000000 00:00 0      [stack]
```

Visualisation du tas

Considérons à présent le programme suivant qui met en oeuvre une allocation dynamique :

```
#include <stdio.h>     /* printf() */
#include <stdlib.h>    /* random() */

int main(void) {
    int *pointer;
    int value;

    pointer = (int *) malloc(sizeof(int));

    while (1) {
        value = random();
        *pointer = value;
    }

    return EXIT_SUCCESS; /* unreachable code */
}
```

Sa cartographie mémoire fait alors apparaître un segment de tas :

```
$ cat /proc/4155/maps
address           perms  offset   time inode     pathname
00400000-00401000 r-xp 00000000 08:04 2886969 /home/prof/jsaigne/mem_03
00600000-00601000 r--p 00000000 08:04 2886969 /home/prof/jsaigne/mem_03
00601000-00602000 rw-p 00001000 08:04 2886969 /home/prof/jsaigne/mem_03
011a5000-011c6000 rw-p 00000000 00:00 0       [heap]
7f88d2ac1000-7f88d2c3b000 r-xp 00000000 08:03 262191 /lib/libc-2.11.1.so
7f88d2e3e000-7f88d2e3f000 rw-p 0017d000 08:03 262191 /lib/libc-2.11.1.so
7f88d2e3f000-7f88d2e44000 rw-p 00000000 00:00 0
7f88d2e44000-7f88d2ec6000 r-xp 00000000 08:03 262240 /lib/libm-2.11.1.so
7f88d30c6000-7f88d30c7000 rw-p 00082000 08:03 262240 /lib/libm-2.11.1.so
7f88d30c7000-7f88d30e7000 r-xp 00000000 08:03 262166 /lib/ld-2.11.1.so
7f88d32af000-7f88d32b2000 rw-p 00000000 00:00 0
7f88d32e4000-7f88d32e6000 rw-p 00000000 00:00 0
7f88d32e6000-7f88d32e7000 r--p 0001f000 08:03 262166 /lib/ld-2.11.1.so
7f88d32e7000-7f88d32e8000 rw-p 00020000 08:03 262166 /lib/ld-2.11.1.so
7f88d32e8000-7f88d32e9000 rw-p 00000000 00:00 0
7fff03f3e000-7fff03f53000 rw-p 00000000 00:00 0      [stack]
```

Avantages et inconvénients de la segmentation

La segmentation facilite la gestion de la mémoire en permettant une protection et un partage plus granulaires. Cependant, elle peut aussi conduire à une fragmentation externe où l'espace mémoire est gaspillé entre les segments alloués.

La gestion de la mémoire a également un impact significatif sur la sécurité des systèmes. Des techniques comme la distribution aléatoire de l'espace d'adressage (*address space layout randomization* ou ASLR) utilisent la mémoire virtuelle de manière à prévenir les attaques par débordement de tampon en rendant aléatoire l'agencement des espaces d'adressage des processus (typiquement la pile et le tas). En cela, elles rendent beaucoup plus difficile pour de potentiels attaquants la localisation des données, et augmentent donc la sécurité des systèmes [25].

5.11 Nature des segments mémoires

Un segment mémoire peut être associé à un fichier ou à une région contiguë d'un fichier, comme le montre la figure 5.6. Dans ce cas, le segment a une image sur le disque, mais il n'y a véritablement transfert que lors d'une demande de page associée à celui-ci.

Alternativement, un segment peut être associé à un fichier « fictif » rempli de zéros. Au premier accès à une page, celle-ci est mise à zéro sans transfert depuis le disque. En cas de modification, son contenu est alors copié dans un fichier commun de réserve appelé **swap**.

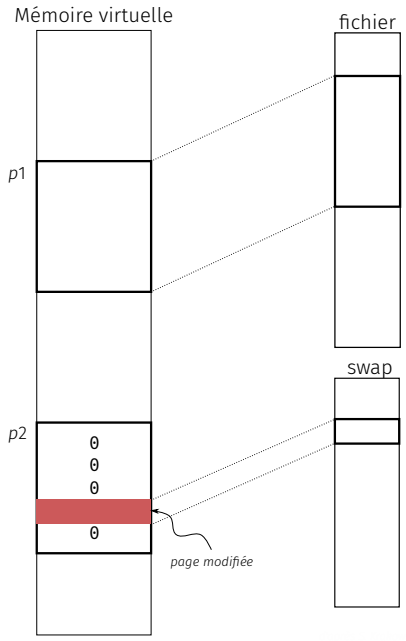


Fig. 5.6 : Association segment / fichier (swap).

5.12 Couplage entre mémoire virtuelle et fichiers

Le couplage entre mémoire virtuelle et fichiers est réalisé par l'appel système `mmap()`, qui associe une zone de mémoire virtuelle à une zone de fichier. On utilise le terme **projection** ou encore **mappage**. Ce mécanisme permet

un accès direct au contenu du fichier sans réaliser explicitement une entrée-sortie (particulièrement utilisé par les serveurs Web). Le découplage (ou **démappage**), quant à lui, sera réalisé à l'aide de l'appel système `munmap()`.

Les prototypes de ces deux appels système sont les suivants :

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

Les paramètres de `mmap()` et `munmap()` sont les suivants :

- `addr` : pointeur sur la zone de la mémoire virtuelle à associer ;
- `length` : longueur de la zone de fichier qui sera couplée ;
- `prot` : protection (`PROT_READ`, `PROT_WRITE`) ;
- `flags` : type de couplage (`MAP_PRIVATE`, `MAP_SHARED`) ;
- `fd` : descripteur du fichier associé ;
- `offset` : déplacement depuis le début de fichier (la valeur doit être un **multiple** de la taille d'une page).

Exemple d'appel à `mmap()`

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>

int main(void) {
    int fd = open("example.dat", O_RDONLY);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    char *data = mmap(NULL, getpagesize(), PROT_READ, MAP_PRIVATE, fd, 0);

    if (data == MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }
    printf("First byte: %c\n", data[0]);
    munmap(data, getpagesize());
    close(fd);

    return EXIT_SUCCESS;
}
```

5.13 Partage de segments entre mémoires virtuelles

Un segment, souvent une zone de fichier, peut être couplé à plusieurs mémoires virtuelles. Il existe deux modes de couplage : **partagé** ou **privé**.

Un segment partagé existe en un seul exemplaire en mémoire physique et peut être associé à plusieurs processus. Toute modification par un processus devient visible à tous les autres, puis elle est reportée sur le disque. Ce mécanisme est utile pour les applications qui nécessitent un accès concurrent aux mêmes données, comme les bases de données partagées ou les segments de code partagés entre différents processus lorsqu'il s'agit d'économiser la mémoire.

En revanche, un segment privé se comporte comme un segment partagé tant qu'il n'est modifié par aucun processus. Si une page du segment privé est modifiée, une nouvelle page est allouée en mémoire physique pour la modification, qui devient alors invisible aux autres processus. Chaque nouvelle modification entraîne une allocation similaire. Cette approche, connue sous le nom de *copy-on-write*, est utilisée pour optimiser les performances et la gestion de la mémoire, notamment lors de la création de processus par `fork()`.

Exemple d'utilisation de segments partagés et privés

Supposons que deux processus projettent le même fichier en utilisant `mmap()` avec des options de partage différentes. Voici un exemple de code illustrant cette situation :

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main(void) {
    int fd = open("shared_file.dat", O_RDWR | O_CREAT, 0666);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    /* Initialize file with some data */
    if (write(fd, "Hello les Ensicaenais !", 23) != 23) {
        perror("write");
        close(fd);
        exit(EXIT_FAILURE);
    }

    /* Map the file as shared */
    char *shared_mem = mmap(NULL, getpagesize(), PROT_READ | PROT_WRITE,
                            MAP_SHARED, fd, 0);
    if (shared_mem == MAP_FAILED) {
        perror("mmap");
        close(fd);
        exit(EXIT_FAILURE);
    }

    /* Map the file as private */
    char *private_mem = mmap(NULL, getpagesize(), PROT_READ | PROT_WRITE,
                             MAP_PRIVATE, fd, 0);
    if (private_mem == MAP_FAILED) {
        perror("mmap");
```

```
        munmap(shared_mem, getpagesize());
        close(fd);
        exit(EXIT_FAILURE);
    }

    /* Modify the shared memory */
    strcpy(shared_mem, "Shared Hello");

    /* Modify the private memory */
    strcpy(private_mem, "Private Hello");

    /* Print the contents of the mapped memory */
    printf("Shared memory: %s\n", shared_mem);
    printf("Private memory: %s\n", private_mem);

    /* Cleanup */
    if (munmap(shared_mem, getpagesize()) == -1) {
        perror("munmap shared");
    }
    if (munmap(private_mem, getpagesize()) == -1) {
        perror("munmap private");
    }
    close(fd);

    return EXIT_SUCCESS;
}
```

Dans cet exemple, les modifications apportées à `shared_mem` sont visibles par tous les processus projetant le même fichier en mode partagé. En revanche, les modifications apportées à `private_mem` sont locales et ne sont pas visibles par les autres processus.

5.14 Précisions sur les appels `fork()` et `exec()`

L'appel système `fork()` crée un processus dont la mémoire virtuelle est initialement une copie conforme de celle du processus parent. En réalité, il n'y a pas de recopie physique du contenu de la mémoire, mais seulement des descripteurs de segments et de la table des pages. Les deux mémoires doivent évoluer indépendamment tout en acceptant le partage de parties communes non modifiées. Chaque segment des deux mémoires virtuelles est d'abord placé en mode *privé*, ce qui autorise par la suite une évolution indépendante à partir d'un état initial commun. Dans la pratique, `fork()` est souvent suivi d'un appel à `exec()`, qui modifie la mémoire virtuelle du processus enfant en y associant un nouveau fichier exécutable.

Quant à l'appel système `execve("prog", argv, env)`, il charge en mémoire virtuelle le programme contenu dans le fichier spécifié par `prog`, avec les arguments `argv` et l'environnement `env`, puis lance son exécution. Le fonctionnement interne des appels système de type `exec` supprime les descripteurs de segments existants, crée des descripteurs pour les segments *text*, *data*, *BSS* et pile, et associe les bibliothèques dynamiques nécessaires. Le compteur ordinal est alors initialisé au point d'entrée du programme exécutable (pointe vers `main()` dans un programme C).

La figure 5.7 montre les permissions des segments après un appel à `execve()`.

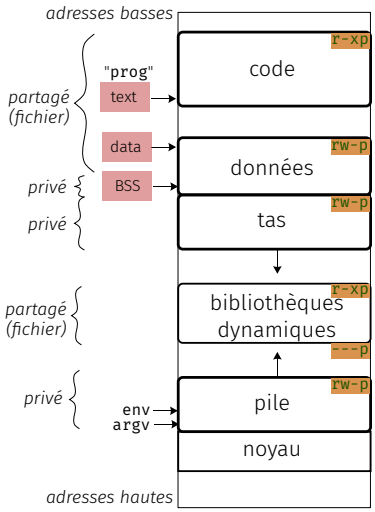


Fig. 5.7 : Permissions des segments après un appel à `execve()`.

5.15 Allocation dynamique de mémoire

Introduction

L'allocation *dynamique* de mémoire est réalisée **lors de l'exécution**, par opposition à l'allocation *statique* où une zone fixée est réservée **avant l'exécution**. Elle est nécessaire lorsque les besoins en mémoire sont inconnus au moment de la compilation, comme dans le cas des structures de données dynamiques ou des fonctions récursives. L'allocation dynamique utilise une pile pour les structures dont la durée de vie suit une règle LIFO (*last in, first out*), et un tas pour les structures dont la durée de vie est indéterminée.

Importance de l'allocation dynamique de mémoire

L'allocation de mémoire a une influence significative sur la performance d'une application. Beaucoup d'applications ont des besoins importants en mémoire, et une mauvaise gestion peut être pénalisante en raison des différences de temps d'accès entre la mémoire principale et secondaire, même si la mémoire virtuelle occulte cette distinction. L'allocation dynamique pose des problèmes délicats et est à l'origine d'erreurs difficiles à détecter, comme les fuites de mémoire. Certains langages comme Java gèrent automatiquement l'allocation de mémoire, tandis que des langages comme Rust renforcent les vérifications à la compilation afin de limiter les erreurs.

Allocation dynamique de mémoire

Sous Unix, les fonctions d'allocation de mémoire sont `malloc()` et `free()` de la bibliothèque C standard. `malloc()` alloue un bloc de mémoire virtuelle de taille au moins égale à `size` octets dans le tas[2]. En cas d'erreur, `malloc()` renvoie `NULL` et affecte une valeur au code d'erreur `errno`. La mémoire allouée n'est pas initialisée (la fonction `calloc()` permet de pallier à ce manque). `free()` libère le bloc alloué par `malloc()`, et son effet est indéterminé pour une autre valeur de pointeur qui lui serait passé en argument.

Les prototypes de `malloc()` et `free()` sont les suivants :

[2] Le terme "sur le tas" n'est pas vraiment adapté à l'allocation dynamique étant donné que le tas n'est pas une pile et souffre de fragmentation comme nous le verrons plus loin.

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
```

Exemple d'allocation dynamique de mémoire

La figure 5.8 présente un exemple d'allocations/désallocations dynamiques sur le tas. Comme on peut le constater, l'organisation du tas peut laisser apparaître des trous dans le segment.

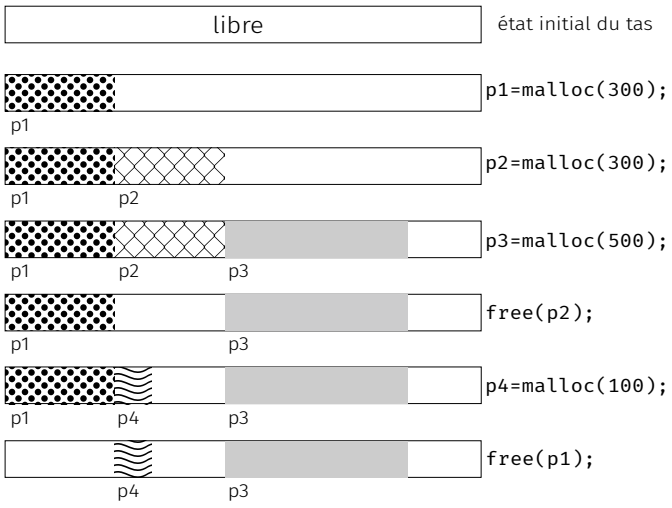


Fig. 5.8 : Exemple d'allocations dynamiques sur le tas.

Algorithmes d'allocation dynamique de mémoire

Méthode avec un pointeur de début de zone libre

La représentation des zones libres se fait souvent sous forme de liste chaînée avec des pointeurs sur les zones libres comme le montre la figure 5.9. Pour satisfaire une demande, on parcourt la liste et on choisit soit la première zone libre assez grande (*first fit*), soit la zone dont la taille est la plus proche de la demande (*best fit*). La variante *next fit* commence la recherche à partir de la position courante du pointeur de parcours.

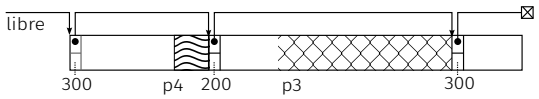


Fig. 5.9 : Exemple de tas avec des pointeurs de début des zones libres.

First fit est donc rapide, mais l'espace libre risque d'être mal utilisé avec la perte de grands segments. La méthode *next fit* permet d'éviter les petites zones libres en début de liste. Quant à la méthode *best fit*, elle permet le meilleur ajustement, mais elle est plus lente et risque d'introduire de nombreuses zones de petite taille (on parle d'émiettement).

Méthode avec des pointeurs de début et de fin de zone libre

Une autre représentation des zones libres utilise des marqueurs de début et de fin. Chaque zone, qu'elle soit libre ou allouée, comporte deux marqueurs identiques contenant la taille de la zone et un bit d'occupation (0 : libre, 1 : occupé). Cette méthode facilite la réunion de deux zones libres adjacentes lors d'une libération : ce sont celles qui ont 2 marqueurs voisins avec leur bit d'occupation à 0. La méthode utilise les mêmes stratégies de recherche (*first fit*, *next fit*, *best fit*).

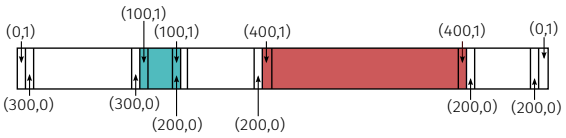


Fig. 5.10 : Exemple de tas avec des pointeurs de début et fin des zones libres.

Méthode avec des groupes de tailles

Il est fréquent que les demandes d'allocation portent sur un nombre restreint de tailles différentes. On organise alors la mémoire en plusieurs groupes de zones d'une taille donnée. Un groupe spécial peut être prévu pour les demandes hors des tailles standards, ou les demandes peuvent être arrondies à la taille supérieure la plus proche. Cette méthode permet une gestion efficace de l'espace libre.

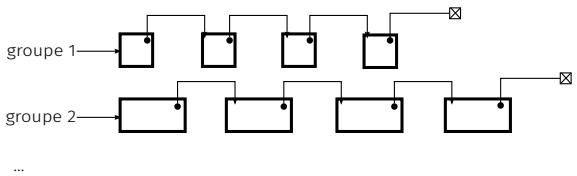


Fig. 5.11 : Exemple de groupes.

Critères d'allocation dynamique de mémoire

Une bonne gestion de la mémoire requiert des performances élevées pour l'allocation et la libération (si possible en temps constant), une bonne utilisation de l'espace disponible pour réduire la fragmentation, et une sécurité permettant de vérifier la libération des zones allouées et d'éviter la référence à des espaces non alloués. Les fonctions `malloc()` et `free()` ne satisfont pas toujours ces critères de sécurité.

Pièges de l'allocation dynamique de mémoire

Voici quelques pièges d'allocation dynamique mis en avant par Bryant et ses co-auteurs [5].

Mauvais déréférencement de pointeurs

```
int main(void) {
    int val;

    scanf("%d", val);
    /* ... */
}
```

Correction :

```
int main(void) {
    int val;

    scanf("%d", &val);
    /* ... */
}
```

Mémoire non initialisée

```
/* Calculate matrix x vec */
#define N ...

int *matvec(int **matrix, int *vec) {
    int *y = (int *)malloc(N * sizeof(int));
    int i, j;

    for (j = 0; j < N; j++)
        for (i = 0; i < N; i++)
            y[i] += matrix[i] [j] * vec[j];
    return y;
}
```

Correction :

```
/* Calculate matrix x vec */
#define N ...

int *matvec(int **matrix, int *vec) {
    int *y = (int *)malloc(N * sizeof(int));
    int i, j;

    memset(y, 0, N * sizeof(int)); /* initialization */
    for (j = 0; j < N; j++)
        for (i = 0; i < N; i++)
            y[i] += matrix[i] [j] * vec[j];
    return y;
}
```

Débordement de tampon

```
int main(void) {
    char buf[64];

    gets(buf);
    /* ... */
}
```

Correction :

```
int main(void) {
    char buf[64];
```

```
    if (fgets(buf, sizeof(buf), stdin) != NULL) {
        /* ... */
    }
}
```

Arithmétique des pointeurs

```
int *search(int *p, int val) {
    while (*p && *p != val)
        p += sizeof(int);
    return matrix;
}
```

Correction :

```
int *search(int *p, int val) {
    while (*p && *p != val)
        p++;
    return p;
}
```

Fuite de mémoire

```
void fuite(int n) {
    int *x = (int *) malloc(n * sizeof(int));
    return;
}
```

Correction :

```
void fuite(int n) {
    int *x = (int *) malloc(n * sizeof(int));
    free(x);
    return;
}
```

5.16 Exercices

Exercice 1 : Cadres et pages

Un système d'exploitation fournit aux processus un espace d'adressage virtuel de 2^{32} octets. L'ordinateur dispose quant à lui de 2^{18} octets de mémoire RAM. La gestion mémoire est paginée avec des pages de taille 4 Kio. On considérera que les tailles de pages et de cadres (cases) sont identiques. En combien de cadres de 4 Kio la mémoire physique peut-elle être découpée ?

Exercice 2 : Ordonnancement des pages

Un processus utilise 5 pages virtuelles dans l'ordre suivant : P1, P2, P3, P4, P1, P2, P5, P1, P2, P3, P4, P5 avec une politique de remplacement FIFO. On considère de plus que les tailles de pages et de cadres sont identiques.

Quel est le nombre de défauts de pages pour une mémoire physique composée de : 3 cadres ? 4 cadres ?

6 Communication interprocessus

6.1 Introduction

La communication interprocessus (*inter-process communication* ou IPC) est essentielle dans les systèmes modernes car elle permet aux processus de partager des informations et de coordonner leurs actions. Cela est particulièrement important dans les systèmes distribués et les architectures multiprocesseurs, où les processus peuvent s'exécuter sur des machines différentes ou des coeurs de processeur distincts.

Cette communication peut être effectuée au moyen de différents mécanismes qui ont chacun leurs propres avantages et inconvénients. Parmis ceux-ci, nous détaillerons :

- les **signaux** qui sont utilisés pour la notification d'événements, mais peuvent être délicats à gérer correctement ;
- les **tubes** qui sont utilisés pour la communication unidirectionnelle entre processus ;
- les **files de messages** qui fournissent des files d'attente de messages avec des priorités ;
- les **sockets** qui permettent la communication bidirectionnelle entre processus sur des machines différentes, et avec un bon support pour les protocoles réseau ;
- la **mémoire partagée** qui offre une communication très rapide pour le partage de grandes quantités de données, mais nécessite une gestion rigoureuse de la synchronisation.

6.2 Communication par signaux

Un **signal** est une notification envoyée à un processus pour l'informer d'un événement système ou utilisateur. Les signaux peuvent être utilisés pour une multitude de raisons, allant de l'indication d'une opération illégale (comme une division par zéro) à la notification qu'un utilisateur a interrompu le processus.

Un signal peut être émis par un utilisateur, par un processus ou encore par le noyau du système d'exploitation lui-même. Les signaux se comportent de manière comparable aux interruptions : le processus destinataire réagit en exécutant un gestionnaire de signaux, appelé **traitant** (*handler* en anglais). La différence réside dans le fait qu'une interruption s'adresse à un processeur, tandis qu'un signal s'adresse à un processus.

Les signaux sont un mécanisme de bas niveau qui doit être manipulé avec précaution pour éviter des problèmes comme la perte de signaux. Ils sont néanmoins utiles pour contrôler les interactions utilisateur via le terminal ou pour traiter des événements périodiques. En programmation concurrente, ils permettent de gérer des états comme la terminaison ou la suspension des processus enfants.

Exemples :

- Division par zéro, *overflow* ou instruction interdite sont retransmis par le noyau.
- Signaux envoyés depuis le clavier par l'utilisateur : séquences Ctrl + Z, Ctrl + C, etc.
- Signaux émis par la commande `kill` depuis l'interface de commande ou à l'aide de l'appel système `kill()` depuis un processus en cours.

6.2.1 Fonctionnement d'un signal

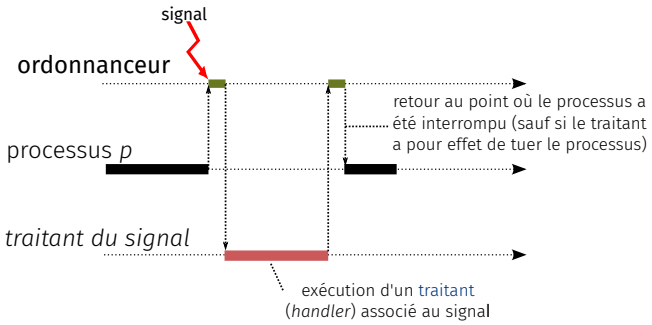


Fig. 6.1 : Fonctionnement d'un signal.

Un signal est identifié par un **nom symbolique** (un entier) et est associé à un **traitant par défaut**. Lorsque le traitant est vide, le signal peut être **ignoré**. Le traitant par défaut d'un signal peut être modifié, excepté pour deux signaux particuliers. Enfin un signal peut être **bloqué** (puis **débloqué**), mais quoi qu'il en soit, n'est **pas mémorisé**.

6.2.2 Quelques signaux POSIX

Signal	Signification	Comportement
`SIGINT`	Interruption depuis le clavier Ctrl + C	Terminaison
`SIGQUIT`	Demande "Quitter" depuis le clavier Ctrl + \	Terminaison et *core dump*
`SIGILL`	Instruction illégale	Terminaison et *core dump*
`SIGABRT`	Signal d'arrêt depuis `abort()`	Terminaison et *core dump*
`SIGFPE`	Erreur en virgule flottante	Terminaison et *core dump*
`SIGKILL`	Signal de terminaison	Terminaison
`SIGSEGV`	Violation de protection mémoire	Terminaison et *core dump*
`SIGPIPE`	Écriture dans un tube sans lecteur	Terminaison
`SIGALRM`	Temporisation `alarm()` écoulée (envoyé par noyau)	Ignoré
`SIGTERM`	Signal de fin	Terminaison
`SIGUSR1`	Signal utilisateur 1	Terminaison
`SIGUSR2`	Signal utilisateur 2	Terminaison
`SIGCHLD`	Terminaison d'un fils	Ignoré
`SIGCONT`	Continuation d'un processus stoppé	Ignoré
`SIGSTOP`	Arrêt du processus	Suspension
`SIGTSTP`	Suspension invoquée depuis tty Ctrl + Z	Suspension
`SIGTTIN`	Lecture sur tty en arrière-plan	Suspension
`SIGTTOU`	Écriture sur tty en arrière-plan	Suspension

Les signaux **`SIGKILL`** et **`SIGSTOP`** ne peuvent pas être bloqués ou ignorés, et leur traitant ne peut pas être modifié.

Tous ces noms symboliques sont définis dans l'entête « `signal.h` » qu'il faudra inclure.

Il est important de comprendre les nuances entre les différents types de signaux et leur comportement par défaut de manière à éviter les erreurs courantes dans la gestion des signaux, notamment la perte de signaux et les conditions de course.

6.2.3 États d'un signal

Un signal est envoyé à un processus destinataire et reçu par ce processus. Tant qu'il n'a pas été pris en compte par le destinataire, le signal est dit **en attente** ou **pendant** (*pending*). Une fois pris en compte (exécution du traitant), le signal est dit **traité**.

Qu'est-ce qui pourrait empêcher qu'un signal soit immédiatement traité dès qu'il est reçu ?

- S'il est **bloqué** ou **masqué** (retardé) par le destinataire (dès qu'il est débloqué, il est alors **délivré**).
- Si un signal du même type est en cours de traitement.

- Les signaux sont traités dans l'ordre croissant de leurs numéros (consulter l'entête « `signal.h` »).
- Il ne peut exister qu'un seul signal en attente pour un type donné (un bit par signal pour indiquer les signaux de ce type qui sont en attente).

L'importance de la gestion correcte des signaux bloqués et en attente réside dans la nécessité de préserver l'intégrité du traitement des données dans les applications multithreads. Un signal bloqué voit son traitement retardé jusqu'à ce que le processus soit prêt à le gérer, évitant ainsi les interruptions potentiellement dangereuses pendant les sections critiques du code.

6.2.4 Structures internes associées aux signaux

Chaque processus possède la table suivante dans sa zone **Proc**. Cette table est structurée comme suit :

N° signal	pendant	bloqué	pointeur traitant	masque temporaire
1	0/1	0/1	`void (*)(int)`	1 0/1 ⋯ 0/1
2	0/1	0/1	`void (*)(int)`	0/1 1 ⋯ 0/1
…	…	…	…	…
`NSIG-1`	0/1	0/1	`void (*)(int)`	0/1 0/1 ⋯ 1

Lorsqu'un signal de type `i` est reçu, l'indicateur `pendant[i]` est mis à 1. Si le signal n'est pas bloqué (`bloque[i] == 0`), le système exécute le traitant associé. Pendant l'exécution du traitant, un masque temporaire est mis en place et le signal `i` est automatiquement bloqué pour éviter qu'il ne soit traité de nouveau. Après l'exécution du traitant, l'indicateur `pendant[i]` est remis à 0.

Un signal `i` est perdu s'il arrive alors que `pendant[i]` est déjà à 1, indiquant que le signal précédent de même type n'a pas encore été traité.

6.2.5 Terminaux, sessions et groupes

Le fonctionnement de certains signaux est lié aux **sessions** et aux **groupes**.

Une session est associée à un utilisateur du système au moyen d'un interpréteur de commandes. Le processus qui exécute l'interpréteur est appelé « processus-chef » de la session (*leader*). Une session peut avoir plusieurs groupes de processus correspondant à divers **travaux** en cours (*jobs*).

Il n'existe au plus qu'un groupe interactif (dit de « premier plan ») avec lequel l'utilisateur interagit au moyen de l'interpréteur. Par contre, il peut exister plusieurs groupes d'« arrière-plan » qui s'exécutent en tâche de fond (par exemple les processus lancés avec l'esperluette **&**). Seuls les processus du groupe interactif peuvent lire sur le terminal. D'autre part, les signaux `SIGINT` (Ctrl + C) et `SIGTSTP` (Ctrl + Z) s'adressent au groupe interactif et non à ceux d'arrière-plan.

6.2.6 États d'un travail

Un **travail** est un processus (ou groupe de processus) lancé par une commande depuis l'interpréteur de commandes. Seul le travail de premier plan peut recevoir des signaux du clavier. Les autres sont manipulés par des commandes.

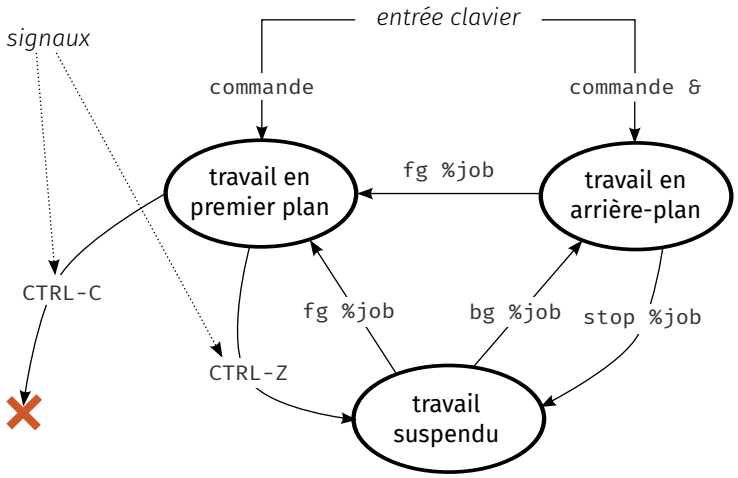


Fig. 6.2 : États d'un travail.

Exemple :

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

int main(void) {
    printf("processus %d, groupe %d\n", getpid(), getpgrp());
    while (1) {
        sleep(2);
    }
```

```
    return EXIT_SUCCESS;
}
```

```
$ ./loop & ./loop & ps
[1] 4906
[2] 4907
processus 4906, groupe 4906
processus 4907, groupe 4907
  PID TTY          TIME CMD
 4785 pts/1    00:00:00 bash
 4906 pts/1    00:00:00 loop
 4907 pts/1    00:00:00 loop
 4908 pts/1    00:00:00 ps

$ fg
./loop
^Z  ==> Frappe de <CTRL-Z>
[2]+  Stopped                 ./loop
$ jobs
[1]-  Running                 ./loop &
[2]+  Stopped                 ./loop

$ bg
[2]+ ./loop &

$ fg
./loop
^C ==> Frappe de <CTRL-C>

$ ps
  PID TTY          TIME CMD
 4785 pts/1    00:00:00 bash
 4906 pts/1    00:04:57 loop
 4928 pts/1    00:00:00 ps
^C ==> Frappe de <CTRL-C>

$ ps
  PID TTY          TIME CMD
 4785 pts/1    00:00:00 bash
 4906 pts/1    00:05:08 loop
 4929 pts/1    00:00:00 ps
```

Dans cet exemple, deux instances du programme `loop` sont lancées en arrière-plan. La commande `fg` fait passer l'un des processus au premier plan, où il peut recevoir des signaux du clavier comme Ctrl + Z pour le suspendre ou Ctrl + C pour le terminer. La commande `jobs` affiche l'état des travaux en cours, tandis que `bg` renvoie un processus suspendu en arrière-plan.

6.2.7 Envoi d'un signal

L'appel système **`kill()`** permet l'envoi de signaux entre processus. Son prototype est le suivant :

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

```
int kill(pid_t pid, int sig);
```

Le signal de numéro `sig` est envoyé par le processus émetteur *p* :

- au processus de numéro `pid` si `pid > 0`;
- à tous les processus du même groupe que *p* si `pid = 0`;
- à tous les processus si `pid = -1`.

Restriction : un processus n'est autorisé à envoyer un signal qu'aux processus de même UID, sauf s'il a les droits du superutilisateur.

Pour des raisons de sécurité, le processus primitif (PID = 1) ne peut pas recevoir de signaux, car sa mort entraînerait celle de tous les autres.

6.2.8 Gestion des signaux

Un processus peut gérer les signaux de plusieurs façons :

1. Ignorer le signal en définissant le gestionnaire de signal à `SIG_IGN`.
2. Utiliser l'action par défaut associée au signal à l'aide du gestionnaire par défaut.
3. Définir un gestionnaire personnalisé à l'aide d'une fonction qui sera exécutée lorsque le signal est reçu.

La gestion des signaux est configurée à l'aide de l'appel système `sigaction()` qui permet de définir des comportements complexes et de spécifier quels sigaux doivent être bloqués pendant l'exécution du gestionnaire.

On utilise la structure `sigaction` suivante déclarée dans `signal.h` :

```
struct sigaction {
    union {
        void (*sa_handler)(int); /*<! pointer to the first handler function */
        void (*sa_sigaction)(int, siginfo_t *, void *); /*<! pointer to the second handler function */
    } sa_handler_union;
    sigset_t sa_mask;            /*<! signals to block */
    int sa_flags;                /*<! options */
    void (*sa_restorer)(void);   /*<! deprecated -- DO NOT USE */
};
```

et l'appel système du même nom dont le prototype est le suivant :

```
int sigaction(int sig, const struct sigaction *newaction, struct sigaction *oldaction);
```

L'appel système `signal()` que l'on trouve encore de temps à autre dans la littérature ancienne ou sur le Web, et déclaré dans « `signal.h` » ne respecte pas la norme POSIX. Il faut lui préférer l'appel `sigaction()` !

Exemple 1a : Sortie d'une boucle infinie

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

```c
#include <unistd.h>

void signal_handler(int sig) {
    printf("Signal SIGINT received!\n");
    printf("Now, I terminate properly\n");
    exit(EXIT_SUCCESS);
}

int main(void) {
    struct sigaction action;

    action.sa_handler = signal_handler;
    sigemptyset(&action.sa_mask);
    sigaction(SIGINT, &action, NULL);

    while (1) {
        sleep(1);  /* Simulate doing some stuff infinitely */
    }
    printf("This line will never be printed.\n");

    return EXIT_SUCCESS;
}
```

Exemple 1b : Variante de sortie d'une boucle infinie

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

volatile int quit = 0;

void signal_handler(int sig) {
    printf("Doing some stuff to terminate properly!\n");
    quit = 1;
}

int main(void) {
    struct sigaction action;

    action.sa_handler = signal_handler;
    sigemptyset(&action.sa_mask);
    sigaction(SIGINT, &action, NULL);

    while (!quit) {
        sleep(1);  /* Simulate doing some stuff infinitely */
    }

    printf("Now, I terminate properly\n");

    return EXIT_SUCCESS;
}
```

Exemple 2 : utilisation de la temporisation

La fonction **`alarm()`** provoque l'envoi du signal `SIGALRM` par le noyau du système après `nbSec` secondes. Ce mécanisme est annulé avec `nbSec` = 0.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void timeout_handler(int sig) {
    printf("Too late!\n");
    exit(EXIT_FAILURE);
}

int main(void) {
    struct sigaction action;
    int response;
    int remaining;

    action.sa_handler = timeout_handler;
    sigemptyset(&action.sa_mask);
    sigaction(SIGALRM, &action, NULL);

    printf("Enter a number within 5 seconds: ");
    alarm(5);  /* Set alarm for 5 seconds */
    scanf("%d", &response);
    remaining = alarm(0);  /* Cancel the alarm */

    printf("Received in %d seconds.\n", 5 - remaining);

    return EXIT_SUCCESS;
}
```

Exemple 3 : horloge

L'exemple suivant présente une horloge simple permettant d'afficher l'heure indéfiniment. Il utilise la fonction `alarm()` une première fois afin de lancer l'horloge, puis la rappelle depuis le traitant pour réamorcer le mécanisme.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

int h = 0, m = 0, s = 0;

void tick(int sig) {
    s++;
    if (s == 60) {
        s = 0;
        m++;
        if (m == 60) {
```

```
            m = 0;
            h++;
            if (h == 24) h = 0;
        }
    }
    alarm(1); /* Reset the alarm for next second */
    printf("%02d:%02d:%02d\n", h, m, s);
}

int main(void) {
    struct sigaction action;

    action.sa_handler = tick;
    sigemptyset(&action.sa_mask);
    sigaction(SIGALRM, &action, NULL);

    alarm(1); /* Start the timer */
    while (1) {
        pause();  /* Wait for signals */
    }

    return EXIT_SUCCESS;
}
```

Exemple 4 : synchronisation parent-enfant

Lorsqu'un processus se termine ou est suspendu, le système envoie automatiquement un signal `SIGCHLD` à son processus parent. Ce signal est ignoré par défaut.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/wait.h>

void cleanup_child(int sig) {
    int status;

    printf("Child process terminated, cleaning up\n");
    wait(&status);  /* Clean up the child process */
}

int main(void) {
    struct sigaction action;

    sigemptyset(&action.sa_mask);
    action.sa_handler = cleanup_child;
    sigaction(SIGCHLD, &action, NULL);

    if (fork() == 0) {
        printf("Child process: %d\n", getpid());
        exit(EXIT_SUCCESS);
    } else {
```

```
        while (1) {
            printf("Parent process working\n");
            sleep(1);
        }
    }

    return EXIT_SUCCESS;
}
```

Exemple 5 : renseigner sur le processus émetteur du signal

Le champ `sa_flags` permet de changer le type du traitant exécuté à la réception d'un signal. Dans ce cas-là le prototype va recevoir des informations sur le processus qui a envoyé le signal. Ces informations sont récupérées par l'intermédiaire d'un pointeur sur la structure `siginfo_t`. `siginfo_t` est définie dans l'entête « `sys/signal.h` » :

```
typedef struct siginfo_t {
    int si_signo;      /* signal number */
    int si_errno;      /* errno value */
    int si_code;       /* signal code */
    pid_t si_pid;      /* sending process's PID */
    uid_t si_uid;      /* sending process's real UID */
    int si_status;     /* exit value or signal */
    clock_t si_utime;  /* user time consumed */
    clock_t si_stime;  /* system time consumed */
    sigval_t si_value; /* signal payload value */
    int si_int;        /* POSIX.1b signal */
    void *si_ptr;      /* POSIX.1b signal */
    void *si_addr;     /* memory location that caused fault */
    int si_band;       /* band event */
    int si_fd;         /* file descriptor */
};
```

Pour des compléments d'information sur ce mécanisme, reportez-vous à [24] ou encore [4].

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

void signal_info_handler(int signal, siginfo_t *siginfo, void *context) {
    printf("Received signal from PID: %ld, UID: %ld\n", (long)siginfo->si_pid,
    ↪ (long)siginfo->si_uid);
}

int main(void) {
    struct sigaction action;

    sigemptyset(&action.sa_mask);
    action.sa_flags = SA_SIGINFO;
    action.sa_sigaction = signal_info_handler;
    sigaction(SIGTERM, &action, NULL);
```

```
    while (1) {
        sleep(10);  /* Wait 10 seconds */
    }

    return EXIT_SUCCESS;
}
```

Exemple 6 : créer un processus démon

Comme nous l'avons entraperçu dans le chapitre 2, les démons sont des processus qui s'exécutent en arrière-plan sans interaction directe avec l'utilisateur. Un démon se détache du terminal de contrôle et continue de fonctionner même après la fermeture de celui-ci.

Pour transformer un processus en démon, il faut suivre plusieurs étapes :

1. Créer un processus enfant et terminer le processus parent afin de détacher l'enfant du terminal de contrôle (le processus enfant devient orphelin).
2. Créer une nouvelle session et devenir le *leader* de la session pour se détacher complètement du terminal de contrôle.
3. Changer le dossier de travail pour éviter de bloquer un point de montage.
4. Fermer les descripteurs de fichier standard (`stdin`, `stdout`, `stderr`).

Voici un exemple de transformation d'un processus en démon :

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

volatile sig_atomic_t keep_running = 1;

void handle_signal(int sig) {
    if (sig == SIGTERM || sig == SIGINT) {
        keep_running = 0;
    }
}

void daemonize() {
    pid_t pid = fork();
    if (pid < 0) {
        exit(EXIT_FAILURE);
    }
    if (pid > 0) {
        exit(EXIT_SUCCESS);
    }

    if (setsid() < 0) {
        exit(EXIT_FAILURE);
    }
```

```
    struct sigaction sa;
    sa.sa_handler = SIG_IGN;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;

    sigaction(SIGCHLD, &sa, NULL);
    sigaction(SIGHUP, &sa, NULL);

    pid = fork();
    if (pid < 0) {
        exit(EXIT_FAILURE);
    }
    if (pid > 0) {
        exit(EXIT_SUCCESS);
    }

    umask(0);

    if (chdir("/") < 0) {
        exit(EXIT_FAILURE);
    }

    close(STDIN_FILENO);
    close(STDOUT_FILENO);
    close(STDERR_FILENO);
}

void log_message(const char *message) {
    FILE *logfile = fopen("/var/log/mydaemon.log", "a");
    if (!logfile) {
        return;
    }
    fprintf(logfile, "%s\n", message);
    fclose(logfile);
}

int main(void) {
    struct sigaction action;

    action.sa_handler = handle_signal;
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;
    sigaction(SIGTERM, &action, NULL);
    sigaction(SIGINT, &action, NULL);

    daemonize();

    while (keep_running) {
        log_message("Daemon is running...");
        sleep(10); /* Simulate what the daemon is doing */
    }

    log_message("Daemon is stopping.");
```

```
    return EXIT_SUCCESS;
}
```

- Le premier appel à `fork()` crée un processus enfant, et l'appel système `setsid()` crée une nouvelle session qui permet au démon de se détacher du terminal de contrôle.
- Les signaux `SIGCHLD` et `SIGHUP` sont ignorés. Quant aux signaux `SIGTERM` et `SIGINT`, ils sont capturés pour permettre un arrêt propre du démon.
- Le deuxième appel à `fork()` garantit que le démon ne pourra jamais réacquérir un terminal de contrôle.
- `umask(0)` garantit que les permissions de fichier ne seront pas restreintes et `chdir("/")` change le dossier de travail pour éviter de bloquer un point de montage.
- Les descripteurs de fichier standard sont fermés pour se détacher du terminal de contrôle.

Ce modèle de démon peut être adapté pour différentes tâches de surveillance ou de gestion de ressources système.

6.2.9 Gestion avancée des signaux

Prévention des situations de compétition

Dans un environnement multi-processus ou multithreads, la gestion des signaux nécessite une attention particulière de manière à éviter les situations de compétition (voir le chapitre 7) et s'assurer que ces signaux ne soient pas perdus. Une situation de compétition peut survenir lorsque plusieurs processus ou *threads* tentent de gérer un signal simultanément, ou lorsqu'un processus modifie des structures de données partagées pendant la gestion d'un signal.

Afin de minimiser ce risque, il est nécessaire d'utiliser des mécanismes de synchronisation appropriés et de suivre les meilleures pratiques de programmation concurrente, notamment :

- Masquer les signaux dans les *threads* critiques : Les *threads* qui accèdent à des ressources partagées doivent masquer les signaux pendant ces opérations. Cela peut être accompli en utilisant `sigprocmask()` dans le cas des processus ou `pthread_sigmask()` dans le cas des *threads* afin de bloquer et débloquer des signaux dans ces sections critiques.
- Utiliser des variables de type `sig_atomic_t` : Lorsque des variables sont partagées entre un gestionnaire de signal et le reste du programme, elles doivent être déclarées comme `volatile sig_atomic_t`. Cette déclaration assure que les accès à la variable sont atomiques, c'est-à-dire non interrompus par l'arrivée de signaux.
- Éviter les fonctions non re-entrantes dans les gestionnaires de signaux : Les gestionnaires de signaux doivent éviter d'appeler des fonctions qui ne sont pas re-entrantes (*thread-safe*), telles que `malloc()`, `free()`, et que la plupart des fonctions de la bibliothèque standard C. Il faut alors utilisez des alternatives re-entrantes ou les écrire soi-même.

Pour illustrer la gestion sécurisée des signaux dans un contexte multithreads, considérons l'exemple suivant où un signal est utilisé pour notifier les *threads* d'une condition particulière (par exemple, l'arrêt du programme) :

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

volatile sig_atomic_t stop_requested = 0;

void signal_handler(int sig) {
```

```
    stop_requested = 1;
}

void *todo(void *arg) {
    /* The SIGTERM signal is blocked in this thread */
    sigset_t set;

    sigemptyset(&set);
    sigaddset(&set, SIGTERM);
    pthread_sigmask(SIG_BLOCK, &set, NULL);

    while (!stop_requested) {
        /* Doing the job of this thread */
    }

    /* The signal is unblocked */
    pthread_sigmask(SIG_UNBLOCK, &set, NULL);
    printf("Thread stopping\n");

    return NULL;
}

int main(void) {
    struct sigaction sa;

    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = signal_handler;
    sigaction(SIGTERM, &sa, NULL);

    pthread_t thread_id;
    pthread_create(&thread_id, NULL, todo, NULL);
    pthread_join(thread_id, NULL);

    return EXIT_SUCCESS;
}
```

Le code précédent configure un gestionnaire de signaux pour `SIGTERM` qui met à jour une variable de type `sig_atomic_t`. Les *threads* du programme bloquent le signal pendant leur exécution et vérifient périodiquement si `stop_requested` a été défini, ce qui permet une terminaison contrôlée du *thread*.

Gestion d'un signal pendant une opération d'entrée/sortie bloquante

Une opération d'entrée/sortie bloquante peut significativement impacter la réactivité d'un programme, en particulier dans des environnements où les processus doivent rester réactifs aux événements système ou aux interactions avec l'utilisateur. Utiliser des signaux pour gérer ou interrompre des opérations d'entrée/sortie bloquantes peut alors aider à améliorer la gestion des ressources et la réactivité des applications. Voici deux exemples pour l'illustrer.

Exemple d'interruption d'une opération de lecture bloquante Dans cet exemple, un signal `SIGALRM` est configuré pour être déclenché après 5 secondes. Si le signal est déclenché pendant l'opération de lecture bloquante, le

gestionnaire de signal positionne à 1 la variable `timeout_occurred`. Après l'opération de lecture, le programme vérifie cette variable pour déterminer si l'opération a été interrompue par le signal.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <fcntl.h>

volatile sig_atomic_t timeout_occurred = 0;

void handle_alarm(int sig) {
    timeout_occurred = 1;
}

int main(void) {
    int fd = open("a_slow_device", O_RDONLY);

    if (fd < 0) {
        perror("Failed to open device.");
        exit(EXIT_FAILURE);
    }

    struct sigaction sa;
    sa.sa_handler = handle_alarm;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    sigaction(SIGALRM, &sa, NULL);

    alarm(5);  /* Set an alarm to go off in 5 seconds */

    char buffer[100];
    int bytes_read = read(fd, buffer, sizeof(buffer));

    alarm(0);  /* Cancel any pending alarm */

    if (timeout_occurred) {
        fprintf(stderr, "Operation timed out\n");
    } else {
        printf("Read %d bytes: %s\n", bytes_read, buffer);
    }

    close(fd);

    return EXIT_SUCCESS;
}
```

Exemple d'utilisation de `SA_RESTART` L'option `SA_RESTART` permet de faire en sorte que des appels système interrompus par des signaux soient automatiquement relancés. Dans l'exemple qui suit, `SA_RESTART` permet de relancer l'appel à `fgets()` si celui-ci a été interrompu par le signal `SIGINT`. Ce mécanisme simplifie la gestion des erreurs et le flux de contrôle en évitant de traiter explicitement l'interruption.

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

void handler(int sig) {
    printf("Signal %d caught\n", sig);
}

int main(void) {
    struct sigaction sa;
    char buf[100];

    sa.sa_handler = handler;
    sa.sa_flags = SA_RESTART;  /* System calls are restartable */
    sigemptyset(&sa.sa_mask);
    sigaction(SIGINT, &sa, NULL);

    printf("Please type something or press Ctrl-C.\n");
    if (fgets(buf, sizeof(buf), stdin) == NULL) {
        perror("fgets failed.");
    } else {
        printf("You typed: %s", buf);
    }

    return EXIT_SUCCESS;
}
```

6.2.10 Autres appels système pour la gestion des signaux

Outre l'appel système `sigaction()` permettant de définir la réaction d'un processus à un signal, on trouve aussi les appels système :

- `sigreturn(&context)` : retourne un signal;
- `sigsuspend(sigmask)` : remplace le masque de signaux et suspend le processus.

Pour un approfondissement quant à l'utilisation des signaux sur Linux, reportez-vous aux chapitres 6, 7 et 8 de l'ouvrage de C. Blaess [4].

6.3 Communication par tube

Les tubes sont des mécanismes de communication **unidirectionnels** permettant à deux processus de communiquer.

Les systèmes Unix fournissent deux types distincts de **tubes** : les tubes **anonymes** et les tubes **nommés** :

- Les tubes **anonymes** permettent la communication entre un processus et ses enfants, mais ils ne figurent pas dans le système de fichiers.
- Les tubes **nommés** permettent la communication entre des processus quelconques et ils apparaissent dans le système de fichiers.

Ces tubes assurent une communication efficace, bien que leur nature unidirectionnelle et leur utilisation limitée à la même hiérarchie de processus pour les tubes anonymes restreignent quelque peu leur application.

6.3.1 Communication par tube anonyme

Un tube anonyme est exclusivement utilisable entre un processus et ses descendants, ou entre descendants d'un même processus. L'appel système `pipe()` permet de créer un tube anonyme, dont les extrémités, entrée et sortie, sont associées à des descripteurs de fichier attribués par le système (généralement les deux premiers descripteurs disponibles dans la table des descripteurs).

Le prototype de `pipe()` est le suivant :

```
#include <unistd.h>

int pipe(int fd[2]);
```

En cas de succès, l'appel système crée un tube et met à jour le tableau `fd` de descripteurs. `fd[0]` correspond au descripteur de lecture du tube et `fd[1]` à celui d'écriture. L'appel retourne 0 en cas de réussite et -1 en cas d'erreur. La figure 6.3 illustre comment une connexion unidirectionnelle est établie du processus parent vers le processus enfant.

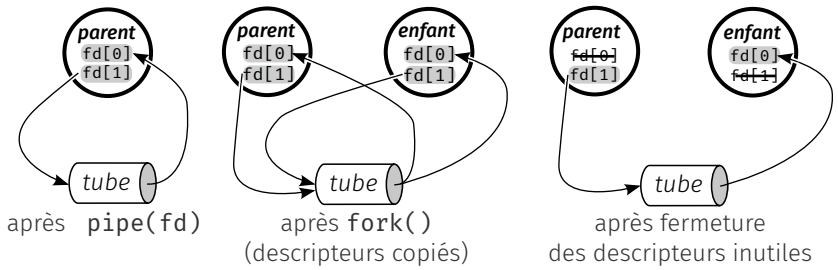


Fig. 6.3 : Établissement d'un tube anonyme du parent vers l'enfant.

Exemple 1 : programmation d'un tube parent vers son enfant

Le programme suivant lit des caractères depuis l'entrée standard et transmet les lettres et les chiffres à un processus enfant. Le processus enfant compte les lettres et chiffres reçus et affiche les résultats à la fin. Le processus parent attend la terminaison de l'enfant avant de s'arrêter.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/wait.h>

#define BYTE_SIZE 1
#define KEYBOARD  0
#define INPUT     1
#define OUTPUT    0

void handle_fatal_error(const char *msg) {
    perror(msg);
    exit(EXIT_FAILURE);
}
```

```
void manage_parent(int anonymous_pipe[]) {
    char byte;

    printf("Parent process (PID %d)\n", getpid());
    close(anonymous_pipe[OUTPUT]);

    while (read(KEYBOARD, &byte, BYTE_SIZE) > 0) {
        if (isalnum(byte)) {
            write(anonymous_pipe[INPUT], &byte, 1);
        }
    }
    close(anonymous_pipe[INPUT]);

    wait(NULL);
    printf("Parent: has received child termination.\n");
}

void manage_child(int anonymous_pipe[]) {
    char byte;
    int letters = 0;
    int digits = 0;

    printf("Child process (PID %d)\n", getpid());
    printf("Enter Ctrl-D to end.\n");
    close(anonymous_pipe[INPUT]);

    while (read(anonymous_pipe[OUTPUT], &byte, BYTE_SIZE) > 0) {
        if (isdigit(byte)) {
            digits++;
        } else {
            letters++;
        }
    }
    close(anonymous_pipe[OUTPUT]);
    printf("\n%d digits received\n", digits);
    printf("%d letters received\n", letters);
}

int main(void) {
    pid_t pid;
    int anonymous_pipe[2]; /* pipe descriptors */

    if (pipe(anonymous_pipe) == -1) {
        handle_fatal_error("Error creating pipe.\n");
    }

    pid = fork();
    if (pid < 0) {
        handle_fatal_error("Error using fork().\n");
    }
    if (pid > 0) {
        manage_parent(anonymous_pipe);
```

```
    } else {
        manage_child(anonymous_pipe);
    }

    return EXIT_SUCCESS;
}
```

Exemple 2 : commande réalisant `cat test.c | grep fork`

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

int main(int argc, char **argv) {
    int anonymous_pipe[2];
    pid_t pid;

    char *cat_args[] = {"cat", "test.c", NULL};
    char *grep_args[] = {"grep", "fork", NULL};

    pipe(anonymous_pipe);
    pid = fork();

    if (pid == 0) { /* The child process manages grep */
        /* Replace standard input with the read end of the pipe */
        dup2(anonymous_pipe[OUTPUT], KEYBOARD);
        close(anonymous_pipe[INPUT]);
        execvp("grep", grep_args);
    } else { /* The parent process manages cat */
        /* Replace standard output with the write end of the pipe */
        dup2(anonymous_pipe[INPUT], STDOUT_FILENO);
        close(anonymous_pipe[OUTPUT]);
        execvp("cat", cat_args);
    }

    return EXIT_SUCCESS;
}
```

L'utilisation des tubes anonymes est particulièrement adaptée aux modèles de communication parent-enfant dans les architectures multiprocessus, où les processus doivent échanger des données de manière sécurisée sans exposer ces données aux autres processus du système. Cette méthode garantit que seuls le processus parent et ses enfants directs peuvent accéder aux informations transmises, préservant ainsi la confidentialité et l'intégrité des données échangées.

6.3.2 Communication par tube nommé (FIFO)

Pour permettre à deux processus quelconques de communiquer, on utilise des tubes nommés (FIFO) qui possèdent un nom symbolique. Pour créer un tube nommé, on peut soit utiliser la commande Unix `mkfifo`, soit faire un appel à la fonction `mkfifo()` dont le prototype est le suivant :

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

int mkfifo(char *pathname, mode_t mode);
```

Le paramètre `mode` peut être spécifié avec les mêmes constantes que celles rencontrées dans la section 4.10 pour l'appel système `open()`. Le tube nommé sera alors visible sur le système de fichier et repérable avec l'indicateur « `p` » (*pipe*) comme l'illustre la [figure **?**].

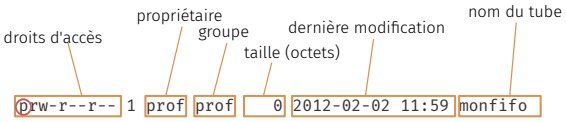


Fig. 6.4 : Droits d'accès d'un tube nommé.

Un tube nommé doit être ouvert en écriture par le processus à l'entrée du tube et en lecture par l'autre. Chaque processus reste bloqué tant que l'autre ne l'a pas ouvert. La figure 6.5 illustre la réalisation d'une communication entre deux processus à l'aide d'un tube nommé.

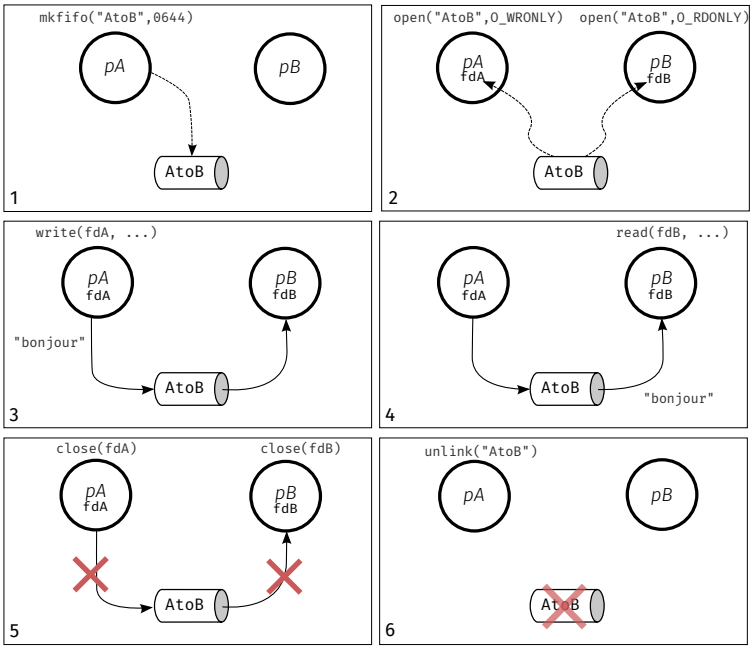


Fig. 6.5 : Établissement d'une communication unidirectionnelle par tube nommé.

Exemple d'utilisation d'un tube nommé

Cet exemple illustre comment deux processus peuvent échanger des messages à l'aide d'un tube nommé. Un processus écrit un message dans le tube et un autre lit ce message.

La création du tube nommé peut se faire à l'aide de la commande Unix `mkfifo` ou à l'aide du programme suivant :

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
```

```c
#include <sys/stat.h>

int main(void) {
    const char *fifo_name = "/tmp/my_fifo";
    mode_t mode = S_IRUSR | S_IWUSR;

    if (mkfifo(fifo_name, mode) == -1) {
        perror("mkfifo");
        exit(EXIT_FAILURE);
    }
    printf("FIFO created at %s\n", fifo_name);

    return EXIT_SUCCESS;
}
```

Processus d'écriture :

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

int main(void) {
    const char *fifo_name = "/tmp/my_fifo";
    int fd;
    char *message = "Hello from sender process!";

    fd = open(fifo_name, O_WRONLY);
    if (fd == -1) {
        perror("open for writing");
        exit(EXIT_FAILURE);
    }

    write(fd, message, sizeof(message));
    close(fd);

    return EXIT_SUCCESS;
}
```

Processus de lecture :

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

int main(void) {
    const char *fifo_name = "/tmp/my_fifo";
    int fd;
    char buffer[100];

    fd = open(fifo_name, O_RDONLY);
    if (fd == -1) {
        perror("open for reading");
        exit(EXIT_FAILURE);
```

```
    }

    read(fd, buffer, sizeof(buffer));
    printf("Received: %s\n", buffer);
    close(fd);

    return EXIT_SUCCESS;
}
```

Exécutez le programme de création du tube nommé, puis dans deux autres terminaux, lancez les programme de lecture et d'écriture.

Les tubes nommés, contrairement aux tubes anonymes, permettent une communication même entre des processus qui n'ont pas de relation parent-enfant directe, offrant ainsi une plus grande flexibilité pour des applications où différents programmes doivent interagir sur un même système. De plus, ils sont persistants jusqu'à leur suppression explicite, ce qui peut être avantageux pour des sessions de communication prolongées ou récurrentes entre processus.

6.4 Communication par mémoire partagée

Une **mémoire partagée** permet à plusieurs processus de partager une zone mémoire, facilitant ainsi l'échange de données sans nécessiter de copie entre les espaces utilisateur et noyau, ce qui est le cas pour les fichiers et les tubes. Cette approche est particulièrement efficace pour partager de gros volumes de données. Le système d'exploitation détermine la localisation des projections dans la mémoire virtuelle de chaque processus accédant au segment de mémoire partagée, comme cela est illustré sur figure 6.6.

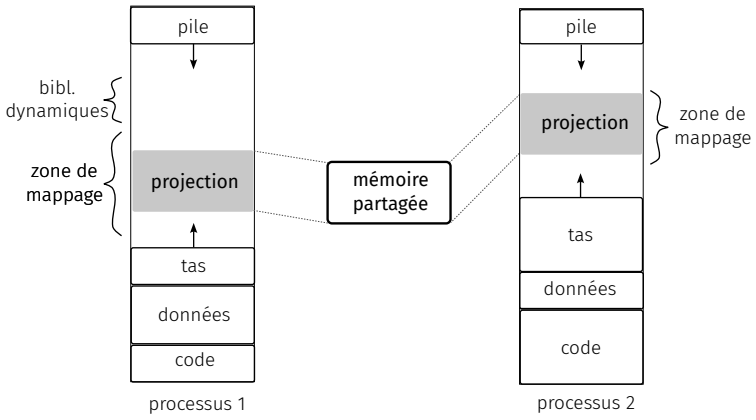


Fig. 6.6 : Utilisation d'une mémoire partagée entre deux processus.

Les segments de mémoire partagée ont une existence indépendante des processus qui les utilisent, ce qui signifie qu'ils persistent même après la terminaison des processus, à moins qu'ils ne soient explicitement détruits.

Un processus peut attacher un segment de mémoire partagée à son espace d'adressage et y accéder directement à l'aide de l'adresse du segment. Cependant, il est nécessaire de protéger ces segments avec des mécanismes de synchronisation tels que des sémaphores pour éviter les accès concurrents non contrôlés.

Les segments de mémoire partagée possèdent également des attributs de sécurité, incluant l'identifiant du propriétaire et les droits d'accès, similaires à ceux des fichiers. Unix offre deux principales implémentations pour la mémoire partagée : System V et POSIX. Ce cours se concentre uniquement sur l'implémentation POSIX.

6.4.1 À propos des segments de mémoire partagée POSIX

Un segment de mémoire partagée POSIX est désigné comme un fichier, mais il n'apparaît pas toujours dans l'espace des noms de fichiers comme c'est le cas sur Mac OS X. Le nom donné à un segment de mémoire partagée commence par le caractère « **/** », par exemple « **/spacebattle** », et sous Linux, le segment apparait dans le dossier « /dev ».

```
$ ls -l /dev/shm
-rw------- 1 jsaigne prof 0 mars   1 13:26 spacebattle
```

6.4.2 Ouvrir et fermer un segment de mémoire partagée

L'appel système `shm_open()` permet de créer et ouvrir un nouveau segment de mémoire partagée, ou simplement d'ouvrir un segment qui existe déjà. Cet appel retourne le descripteur du segment de mémoire partagée, qui pourra ensuite être utilisé pour d'autres opérations sur ce segment. En revanche, l'appel système `close()` ferme le descripteur de segment de mémoire partagée, mais cela ne supprime pas le segment lui-même du système de fichiers. Pour le supprimer réellement, il est nécessaire d'utiliser l'appel système `shm_unlink()`.

Voici les prototypes de ces appels système :

```
#include <unistd.h>    /* close */
#include <sys/mman.h>  /* shm_open() and shm_unlink() */
#include <sys/stat.h>  /* Modes */
#include <fcntl.h>     /* O_xxx */

int shm_open(const char *pathname, int flags, mode_t mode);
int close(int shmd);
int shm_unlink(const char *pathname);
```

Les paramètres `flags` et `mode` utilisés dans l'appel à `shm_open()` sont identiques à ceux utilisés dans l'appel système `open()`, comme décrit dans la section 4.10 du chapitre sur les fichiers et les flux.

6.4.3 Dimensionner un segment de mémoire partagée

Le segment de mémoire partagée créé à l'issue de l'appel à **`shm_open()`** a une **taille nulle**. C'est la fonction **`ftruncate()`** qui permet de tronquer ou de redéfinir la taille de tout fichier, et notamment celle d'un segment de mémoire partagée qui est ensuite réalisé par le processus qui crée le segment.

```
#include <unistd.h>    /* ftruncate */
#include <sys/types.h> /* type off_t */

int ftruncate(int fd , off_t length);
```

Exemple :

```
$ ls -l /dev/shm # avant appel à ftruncate()
-rw------- 1 jsaigne prof 0 mars 1 13:26 spacebattle
$
$ ls -l /dev/shm # après appel à ftruncate()
-rw------- 1 jsaigne prof 4294967300 mars 1 13:29 spacebattle
```

6.4.4 Mapper / démapper un segment de mémoire partagée

L'appel système **`mmap()`** que nous avons rencontré dans le chapitre 5 permet de mapper le segment de mémoire partagée à l'espace virtuel des adresses du processus appelant. Quant à **`munmap()`**, il permet l'opération inverse.

```
#include <sys/types.h> /* type off_t */
#include <sys/mman.h>  /* mmap */

void *mmap(void addr, /* adresse du début de page à mapper */
    size_t length,    /* taille du mappage */
    int prot,         /* accès aux pages */
    int flags,        /* type de segment */
    int fd,           /* descripteur du segment */
    off_t offset      /* offset du mappage */
);

void munmap(int fd,    /* descripteur du segment */
            size_t len /* taille du mappage */
);
```

Valeur `prot`	Description
`PROT_READ`	données accessibles en lecture
`PROT_WRITE`	données accessibles en écriture
`PROT_EXEC`	données exécutables
`PROT_NONE`	données non accessibles

Valeur `flags`	Description
`MAP_SHARED`	tout changement est partagé
`MAP_PRIVATE`	les changements sont privés
`MAP_FIXED`	`addr` devient l'adresse pour la valeur retournée

6.4.5 Étapes de mise en oeuvre

La figure 6.7 illustre la mise en oeuvre d'une communication entre deux processus à l'aide d'un segment de mémoire partagée.

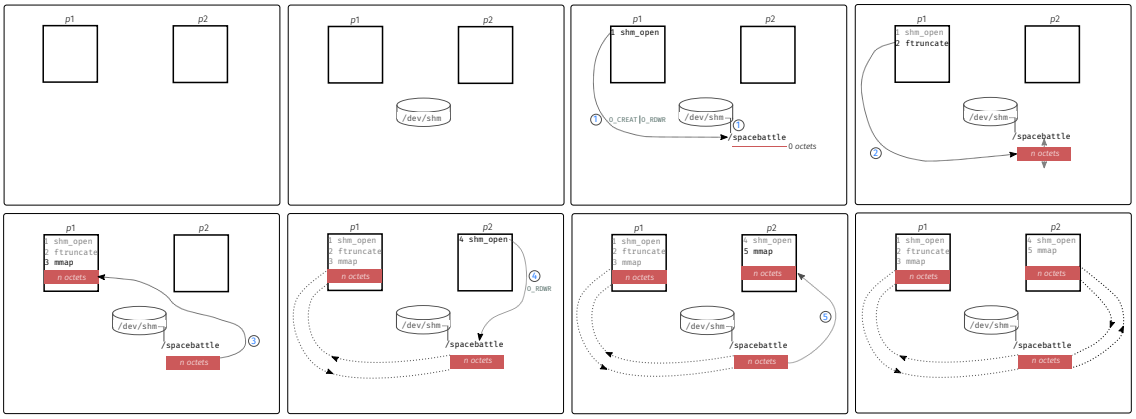


Fig. 6.7 : Mise en oeuvre d'une mémoire partagée entre deux processus.

Exemple n°1

L'exemple suivant montre comment mettre en oeuvre un segment de mémoire partagé (à compiler avec l'option **`-lrt`** sous Linux et sans sous Mac OS X).

```
#define SIZE 1024

struct game_board {
    int ships;
    int space_grid[SIZE][SIZE][SIZE];
};

struct game_board *gb;

int main(void) {
    int shmd = shm_open("/spacebattle", O_CREAT | O_RDWR,
                        S_IRUSR | S_IWUSR | S_IRGRP | S_IR_OTH);
    if (shmd == -1) { /* Error handling */ }
    if (ftruncate(shmd, sizeof(struct game_board)) == -1) { /* Error handling */ }
    gb = mmap(NULL,              /* address to be mapped */
       sizeof(struct game_board), /* size of the mapping */
       PROT_READ | PROT_WRITE,    /* access permissions */
       MAP_SHARED,                /* type of segment */
       shmd,                      /* segment descriptor */
       0                          /* offset of the mapping */
    );

    /* ... perform operations on the shared memory */

    munmap(gb, sizeof(struct game_board));
    close(shmd);
    shm_unlink("/spacebattle");

    return EXIT_SUCCESS;
}
```

Exemple n°2

Cet exemple illustre comment un processus parent et son enfant peuvent communiquer au travers d'un segment de mémoire partagée (à compiler avec l'option **`-lrt`** sous Linux et sans sous Mac OS X).

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <time.h>

#define SHM_SIZE 100

int main(void) {
    int fd, i;
    int *ptr;
    pid_t pid;

    srand(time(NULL));

    /* Create and open a shared memory segment */
    fd = shm_open("/blabla", O_RDWR | O_CREAT,
                  S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    if (fd == -1) {
        perror("shm_open");
        exit(EXIT_FAILURE);
    }

    /* Set the size of the shared memory segment */
    if (ftruncate(fd, SHM_SIZE) == -1) {
        perror("ftruncate");
        close(fd);
        shm_unlink("/blabla");
        exit(EXIT_FAILURE);
    }

    /* Map the shared memory segment into the address space */
    ptr = (int *)mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE,
                      MAP_SHARED, fd, 0);
    if (ptr == MAP_FAILED) {
        perror("mmap");
        close(fd);
        shm_unlink("/blabla");
        exit(EXIT_FAILURE);
    }

    /* Fork a new process */
    pid = fork();
    if (pid == -1) {
        perror("fork");
```

```
        munmap(ptr, SHM_SIZE);
        close(fd);
        shm_unlink("/blabla");
        exit(EXIT_FAILURE);
    }

    if (pid > 0) {
        /* Parent process: write random values to the shared memory */
        for (i = 0; i < SHM_SIZE; i++) {
            ptr[i] = rand() % SHM_SIZE;
            printf("%d ", ptr[i]);
        }
        printf("\n");
        wait(NULL); /* Wait for the child process to finish */
    } else {
        /* Child process: read and print the values from the shared memory */
        for (i = 0; i < SHM_SIZE; i++) {
            printf("%d ", ptr[i]);
        }
        printf("\n");
    }

    /* Unmap the shared memory and close the file descriptor */
    munmap(ptr, SHM_SIZE);
    close(fd);
    if (pid > 0) {
        shm_unlink("/blabla"); /* Remove the shared memory segment */
    }

    return EXIT_SUCCESS;
}
```

Exemple n°3

Cet exemple montre le partage de tout ou partie d'une structure plus complexe entre un processus parent et son enfant, en utilisant des octets de bourrage de manière à aligner les structures sur des limites de pages mémoire (à compiler avec l'option **`-lrt`** sous Linux et sans sous Mac OS X).

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <string.h>
#include <time.h>

#define PAGESIZE 4096

typedef struct {
    float x;
    float y;
```

```
    float z;
} vector_t;

typedef struct {
    unsigned char red;
    unsigned char green;
    unsigned char blue;
} color_t;

typedef struct {
    int id;
    char name[20];
    int age;
    char padding[PAGESIZE - 2 * sizeof(int) - 20 * sizeof(char)];
} s1_t;

typedef struct {
    vector_t vec;
    color_t col;
    char padding[PAGESIZE - sizeof(vector_t) - sizeof(color_t)];
} s2_t;

typedef struct {
    s1_t a_s1;
    s2_t a_s2;
} s1_and_s2_t;

int main(void) {
    int shm_fd;
    pid_t pid;
    s1_t *ptr1;
    s2_t *ptr2;

    srand(time(NULL));

    /* Create and open a shared memory segment */
    shm_fd = shm_open("/pipeautique3", O_CREAT | O_RDWR,
                      S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    if (shm_fd == -1) {
        perror("shm_open");
        exit(EXIT_FAILURE);
    }

    /* Set the size of the shared memory segment to accommodate the structures */
    if (ftruncate(shm_fd, sizeof(s1_and_s2_t)) == -1) {
        perror("ftruncate");
        close(shm_fd);
        shm_unlink("/pipeautique3");
        exit(EXIT_FAILURE);
    }

    /* Map the first structure (s1_t) into the address space of the process */
    ptr1 = (s1_t *)mmap(NULL, sizeof(s1_t), PROT_READ | PROT_WRITE,
```

```
       MAP_SHARED, shm_fd, 0);
    if (ptr1 == MAP_FAILED) {
        perror("mmap ptr1");
        close(shm_fd);
        shm_unlink("/pipeautique3");
        exit(EXIT_FAILURE);
    }

    /* Map the second structure (s2_t) into the address space of the process */
    ptr2 = (s2_t *)mmap(NULL, sizeof(s2_t), PROT_READ | PROT_WRITE,
                        MAP_SHARED, shm_fd, sizeof(s1_t));
    if (ptr2 == MAP_FAILED) {
        perror("mmap ptr2");
        munmap(ptr1, sizeof(s1_t));
        close(shm_fd);
        shm_unlink("/pipeautique3");
        exit(EXIT_FAILURE);
    }

    printf("> %p\n", (void *)ptr1);
    printf("> %p\n", (void *)ptr2);

    /* Fork a new process */
    pid = fork();

    if (pid < 0) {
        perror("fork");
        munmap(ptr1, sizeof(s1_t));
        munmap(ptr2, sizeof(s2_t));
        close(shm_fd);
        shm_unlink("/pipeautique3");
        exit(EXIT_FAILURE);
    }

    if (pid > 0) { /* Parent process */
        /* Write to the shared memory */
        strcpy(ptr1->name, "Monkeypox");
        ptr2->col.red = 112;
        wait(NULL); /* Wait for the child process to finish */
    } else { /* Child process */
        /* Read from the shared memory */
        printf("%s\n", ptr1->name);
        printf("%d\n", ptr2->col.red);
    }

    /* Unmap the shared memory and close the file descriptor */
    munmap(ptr1, sizeof(s1_t));
    munmap(ptr2, sizeof(s2_t));
    close(shm_fd);

    if (pid > 0) {
        shm_unlink("/pipeautique3"); /* Remove the shared memory segment */
    }
```

```
    return EXIT_SUCCESS;
}
```

Les structures utilisées dans l'exemple, `s1_t` et `s2_t`, incluent des octets de bourrage (*padding*) pour s'assurer que chaque structure occupe une page mémoire complète (`PAGESIZE`). Cette technique est utile pour aligner les structures sur des limites de pages mémoire, facilitant ainsi une gestion et un accès efficaces à ces structures.

La structure `s1_and_s2_t` quant à elle, combine les deux structures `s1_t` et `s2_t`, permettant ainsi un partage structuré et aligné des données entre les processus. Le mappage des structures dans l'espace d'adressage virtuel des processus utilise `mmap()` afin d'associer les deux sous-parties du segment de mémoire partagée à deux adresses spécifiques, permettant un accès sélectif à chacune des structures.

Les utilisateurs d'un système Mac OS X peuvent être déçus de ne pas avoir accès aux segments de mémoire partagés, puisque le dossier "`/dev/shm`" leur est inaccessible. Il y a malgré tout une solution pour lister les segments de mémoires partagées. Cette solution fait appel à la commande Unix `lsof`. Voici une fonction permettant de lister les segments sur un système Mac OS X :

```
/**
 * @brief Retrieves the names of shared memory segments on macOS.
 *
 * @param shm_names Array to store the shared memory segment names.
 * @param shm_count Pointer to the count of shared memory segments.
 */
void get_shared_memory_names(char **shm_names, int *shm_count) {
    char buffer[256];
    FILE *fp = popen("lsof -u $(whoami) | grep PSXSHM | cut -w -f 7", "r");
    if (fp == NULL) {
        perror("popen");
        exit(EXIT_FAILURE);
    }

    while (fgets(buffer, sizeof(buffer), fp) != NULL) {
        buffer[strcspn(buffer, "\n")] = 0;  /* We remove the newline character */
        char *shm_name = strdup(buffer);
        if (shm_name != NULL) {
            shm_names[*shm_count] = shm_name;
            (*shm_count)++;
        }
    }

    pclose(fp);
}
```

6.5 Communication par file de messages

6.5.1 Introduction

Une file de messages (`mqueue` pour *memory queue*) propose un mécanisme qui permet l'échange de messages entre processus de manière asynchrone. Bien qu'elle soit désignée comme un fichier, une file de messages n'apparaît

pas toujours dans l'espace des noms de fichiers. Comme pour les segments de mémoire partagée, le nom d'une file de messages commence par la caractère « / », par exemple : **/news**. Les files de messages gèrent la priorité des messages, ce qui permet de traiter certains messages avant d'autres selon leur importance.

L'édition des liens sous Linux nécessite l'utilisation de l'option « -lrt », mais pas sous Mac OS X.

> Deux implémentations coexistent : les files de messages POSIX et les files de messages System V. Ces dernières ne seront pas abordées dans ce cours.
> Mac OS X, bien que respectant POSIX, n'implémente pas les files de messages. Une émulation pour Mac OS X est toutefois proposée par Stanislav Pankevich. Cette implémantation est disponible à l'adresse : https://github.com/stanislaw/posix-macos-addons.

6.5.2 Créer et ouvrir une file de messages

La fonction mq_open() permet de créer et d'ouvrir une file de messages. Unix fournit deux versions de mq_open(), une version permettant la création et l'ouverture, et une version pour l'ouverture d'une file existante. Les prototypes sont les suivants :

```
#include <mqueue.h>

mqd_t mq_open(const char *pathname, int flags, mode_t mode, struct mq_attr *attr);
mqd_t mq_open(const char *pathname, int flags);
```

Les paramètres flags et mode utilisés dans l'appel à mq_open() sont identiques à ceux utilisés dans l'appel système open(), comme décrit dans la section 4.10 du chapitre sur les fichiers et les flux.

La structure mq_attr quant à elle, permet de configurer la file de messages :

```
struct mq_attr {
  long mq_flags;   /* indicateur pour la file (0 ou O_NONBLOCK) */
  long mq_maxmsg;  /* nombre max. de messages dans la file */
  long mq_msgsize; /* taille maximale des messages */
  long mq_curmsgs; /* nombre de messages actuellement dans la file */
};
```

Il est possible de récupérer les valeurs utilisées pour la configuration de la file à l'aide de la fonction mq_getattr() et de modifier cette configuration avec la fonction mq_setattr().

6.5.3 Fermer et détruire une file de messages

Pour fermer une file de messages, on utilise la fonction mq_close(), et pour la détruire, on utilise mq_unlink() :

```
int mq_close(mqd_t mqd);
int mq_unlink(const char *pathname);
```

6.5.4 Envoyer et recevoir des messages

Les fonction mq_send() et mq_receive() permettent respectivement d'envoyer et de recevoir des messages dans une file de messages :

```
#include <mqueue.h>

int mq_send(mqd_t desc,     /* descripteur de la file */
      const char *msg_ptr,  /* message à envoyer */
      size_t msg_len,       /* longueur du message */
      unsigned int msg_prio /* priorité du message */
);

ssize_t mq_receive(mqd_t desc,  /* descripteur de la file */
         char *msg_ptr,          /* message à recevoir */
         size_t msg_len,        /* longueur du message */
         unsigned int *msg_prio /* priorité du message */
);
```

Si les ressources ne sont pas disponibles, `mq_send()` et `mq_receive()` échouent et placent `errno` à `EAGAIN`

6.5.5 Étapes de mise en oeuvre d'une file de messages

La figure 6.8 illustre la mise en oeuvre d'une communication entre deux processus à l'aide d'une file de messages.

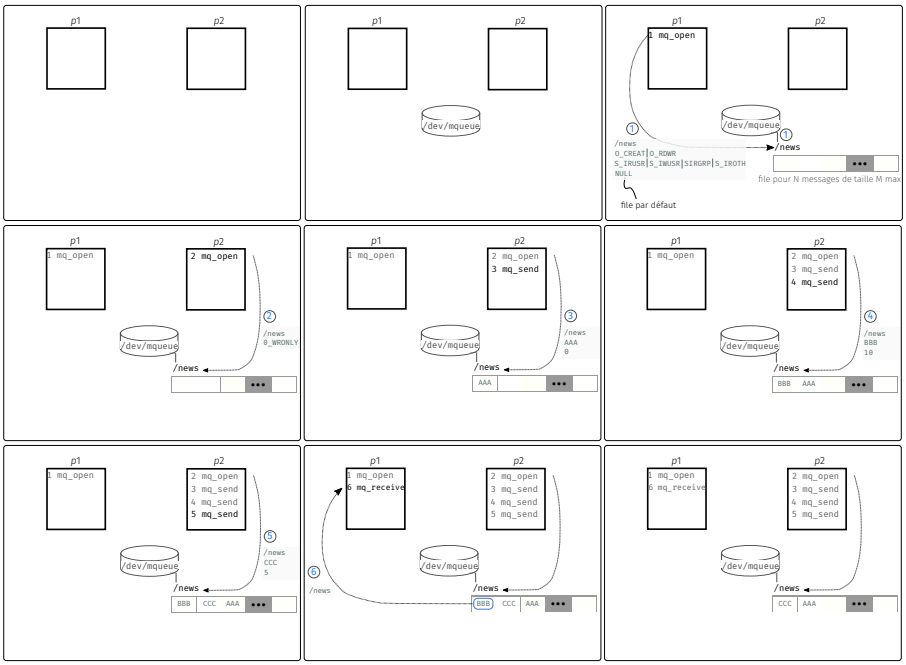


Fig. 6.8 : Mise en oeuvre d'une file de messages entre deux processus.

Exemple d'émetteur

L'exemple suivant montre comment un processus émetteur peut envoyer des messages sur une file de messages. Le programme prend trois arguments : le nom de la file de messages, la priorité du message et le message à envoyer.

```
#include <fcntl.h>
#include <mqueue.h>
#include <stdio.h>
```

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

int main(int argc, char * argv[]) {
    mqd_t mq;
    int priority;

    if (argc != 4) {
        fprintf(stderr, "Syntax: %s queue priority message\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* Open the message queue */
    mq = mq_open(argv[1], O_WRONLY | O_CREAT,
                 S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH, NULL);
    if (mq == (mqd_t)-1) {
        perror("mq_open");
        exit(EXIT_FAILURE);
    }

    /* Parse the priority */
    if (sscanf(argv[2], "%d", &priority) != 1) {
        fprintf(stderr, "Invalid priority\n");
        exit(EXIT_FAILURE);
    }

    /* Send the message */
    if (mq_send(mq, argv[3], strlen(argv[3]), priority) == -1) {
        perror("mq_send");
        exit(EXIT_FAILURE);
    }

    /* Close the message queue */
    mq_close(mq);

    return EXIT_SUCCESS;
}
```

Le programme émetteur après avoir vérifié que le nombre d'arguments est correct, ouvre ou crée une file de messages à l'aide de `mq_open()` suivant que la file existe déjà ou pas. La priorité du message est ensuite analysée à partir des arguments de la ligne de commande et le message est envoyé sur la file à l'aide de `mq_send()`. Enfin, la file est fermée.

Exemple de récepteur

L'exemple suivant montre comment un processus récepteur peut lire des messages depuis une file de messages, par exemple celle créée précédemment. Le programme prend un unique argument : le nom de la file de messages.

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

```c
int main(int argc, char * argv[]) {
    int n, priority;
    mqd_t mq;
    struct mq_attr attr;
    char *buffer = NULL;

    if (argc != 2) {
        fprintf(stderr, "Syntax: %s queue\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* Open the message queue */
    mq = mq_open(argv[1], O_RDONLY);
    if (mq == (mqd_t)-1) {
        perror("mq_open");
        exit(EXIT_FAILURE);
    }

    /* Get the attributes of the message queue */
    if (mq_getattr(mq, &attr) == -1) {
        perror("mq_getattr");
        mq_close(mq);
        exit(EXIT_FAILURE);
    }

    /* Allocate a buffer to receive the message */
    buffer = malloc(attr.mq_msgsize);
    if (buffer == NULL) {
        perror("malloc");
        mq_close(mq);
        exit(EXIT_FAILURE);
    }

    /* Receive the message */
    n = mq_receive(mq, buffer, attr.mq_msgsize, &priority);
    if (n == -1) {
        perror("mq_receive");
        free(buffer);
        mq_close(mq);
        exit(EXIT_FAILURE);
    }

    /* Print the received message */
    fprintf(stdout, "[%d] %s\n", priority, buffer);

    /* Clean up */
    free(buffer);
    mq_close(mq);

    return EXIT_SUCCESS;
}
```

Le récepteur ouvre la file de messages supposément existante, puis il récupère les attributs de celle-ci, notamment la taille maximale des messages, à l'aide de la fonction `mq_getattr()`. Un tampon est alloué dynamiquement afin de stocker le message. Le message est alors lu à partir de la file de messages à l'aide de la fonction `mq_receive()`, et son contenu ainsi que sa priorité sont affichés. En fin de programme, les ressources allouées sont libérées et la file de messages est fermée.

6.6 Communication par sockets

6.6.1 Sockets dans les couches TCP/IP

Les **sockets** fournissent des mécanismes de communication qui permettent à plusieurs processus, qu'ils soient sur la même machine ou sur des machines différentes, d'échanger des données. Un *socket* représente l'extrémité d'une connexion, qu'il s'agisse du côté émetteur ou récepteur, et il est associé à un **port de communication**. Une fois la connexion établie entre deux processus, ceux-ci peuvent échanger des données en utilisant les mêmes appels système que pour les opérations classiques d'entrée/sortie, à savoir `read()` et `write()`.

Les *sockets* fournissent une interface d'accès aux protocoles de transport TCP et UDP :

- **TCP** (« mode connecté ») : une liaison est établie au préalable entre deux systèmes afin de permettre l'échange de flots d'octets. Ce mode assure la fiabilité et l'ordre des données transmises.
- **UDP** (« mode non connecté ») : aucune liaison n'est établie, les messages sont échangés individuellement. Ce mode privilégie la rapidité et l'efficacité au détriment de la fiabilité.

La figure 6.9 illustre les différentes couches TCP/IP et UDP/IP.

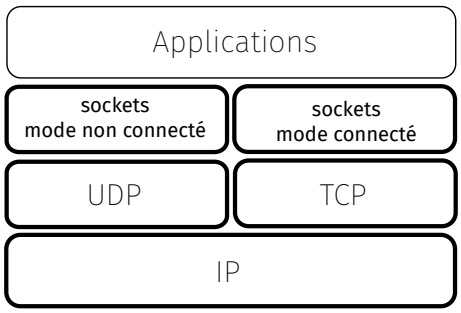


Fig. 6.9 : Couches TCP/IP et UDP/IP.

6.6.2 Créer une socket

L'appel système `socket()` permet de créer un *socket* « anonyme » et retourne un descripteur qui servira aux opérations de lecture et d'écriture. Son prototype est le suivant :

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

int socket(int domain, int type, int protocol);
```

- `domain` spécifie le domaine de communication (`AF_` pour *address family*) :
  - `AF_INET` : utilisé pour la communication IPv4, il permet des connexions entre machines sur un réseau local ou sur Internet.

  - `AF_INET6` : similaire à `AF_INET`, mais pour IPv6, il offre une portée plus grande et une gestion améliorée des adresses IP.
  - `AF_UNIX` ou `AF_LOCAL` : utilisé pour la communication locale entre processus sur une même machine, permettant des performances élevées en évitant le passage par le réseau et l'encapsulation des entêtes. Ce domaine est souvent privilégié par les démons système ou les services d'arrière-plan qui nécessitent un partage rapide de données.

- `type` indique le type de socket : `SOCK_STREAM` (TCP), `SOCK_DGRAM` (UDP), `SOCK_RAW` (utile pour créer un renifleur par exemple), etc.
- `protocol` spécifie le protocole à utiliser avec le *socket*, généralement déterminé par le domaine. On utilise souvent la valeur `0` pour laisser le système choisir le protocole approprié en fonction du type et du domaine (`PF_UNIX`, `PF_INET`, `PF_INET6`, etc.).

L'appel système `socket()` permet ainsi de créer une interface flexible lorsqu'il s'agt d'établir des communications entre processus. En choisissant correctement le domaine, le type et le protocole, il est possible de configurer le *socket* pour diverses situations, que ce soit pour des communications locales rapides ou des connexions réseau robustes et fiables. La création d'un *socket* est la première étape dans la mise en place d'une communication interprocessus qui utilise les protocoles de la suite TCP/IP, posant ainsi les bases pour les étapes suivantes telles que la liaison à une adresse, l'écoute des connexions entrantes, et l'établissement des connexions pour l'échange de données.

6.6.3 Affectation d'un nom à un socket

Afin qu'un processus puisse désigner un *socket*, il est nécessaire de lui associer un nom à l'aide d'une adresse contenue dans une structure `sockaddr` (adresse générique) ou `sockaddr_in` (adresse Internet).

```
#include <sys/socket.h>

struct sockaddr {        /* sockets génériques */
  sa_family_t sa_family; /* famille de protocoles */
  char sa_data[14];      /* données d adressage */
};

struct sockaddr_in {         /* sockets Internet */
  sa_family_t sin_family;    /* toujours AF_INET */
  unsigned short sin_port;   /* numéro de port */
  struct in_addr sin_addr;   /* adresse IP, ordre réseau */
  unsigned char sin_zero[8]; /* bourrage */
};
```

Il existe aussi la structure **`sockaddr_un`** pour les adresses Unix, mais nous ne l'utiliserons pas dans ce cours.

L'appel système **`bind()`** permet de fournir un nom au *socket*. Son prototype est le suivant :

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

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

`bind()` affecte l'adresse spécifiée dans `addr` au *socket* référencé par le descripteur de fichier `sockfd`. Le paramètre `addrlen` indique la taille, en octets, de la structure d'adresse pointée par `addr`.

6.6.4 Big Endian ou Little Endian

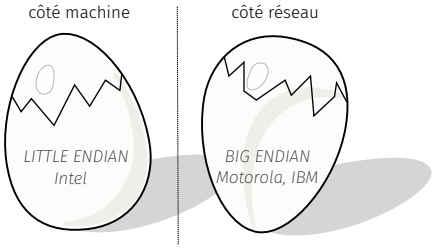


Fig. 6.10 : Big endian ou little endian.

Pour cette étape de codage/décodage, nous devons utiliser les fonctions suivantes :

```
#include <arpa/inet.h>

uint16_t htons(uint16_t hostshort);
uint16_t ntohs(uint16_t netshort);
uint32_t htonl(uint32_t hostlong);
uint32_t ntohl(uint32_t netlong);
```

Exemple d'affectation de nom à une socket

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>

/**
 * A simple test that creates a stream socket and gives it a
 * name. The host address is set to INADDR_ANY and the OS
 * replaces that with the machines actual address.
 */
int main(void) {
    int sd, port = 5432;
    struct sockaddr_in name;

    /* Create the socket */
    sd = socket(PF_INET, SOCK_STREAM, 0);
    if (sd < 0) {
        fprintf(stderr, "socket() failed\n");
        exit(EXIT_FAILURE);
    }
    /* Give a name to the socket */
    name.sin_family = AF_INET;
    name.sin_port = htons(port);
    name.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
    if (bind(sd, (struct sockaddr *)&name, sizeof(name)) < 0) {
        fprintf(stderr, "bind() failed\n");
        exit(EXIT_FAILURE);
    }

    return EXIT_SUCCESS;
}
```

Autre exemple d'affectation de nom à un socket

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

/**
 * A test that creates a stream socket and gives it a name.
 */
int main(int argc, char **argv) {
    int sd, port = 6543;
    struct sockaddr_in name;
    struct hostent *hostinfo;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s hostname\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* Create the socket */
    sd = socket(PF_INET, SOCK_STREAM, 0);
    if (sd < 0) {
        fprintf(stderr, "socket() failed\n");
        exit(EXIT_FAILURE);
    }

    /* Give a name to the socket */
    name.sin_family = AF_INET;
    name.sin_port = htons(port);
    hostinfo = gethostbyname(argv[1]);

    if (hostinfo == NULL) {
        fprintf(stderr, "Unknown host %s.\n", argv[1]);
        exit(EXIT_FAILURE);
    }
    name.sin_addr = *(struct in_addr *) hostinfo->h_addr;

    if (bind(sd, (struct sockaddr *) &name, sizeof(name)) < 0) {
        fprintf(stderr, "bind() failed\n"); *
        exit(EXIT_FAILURE);
    }
```

```
    return EXIT_SUCCESS;
}
```

Les communications par *sockets* peuvent être optimisées en utilisant un multiplexage à l'aide des appels système `select()` ou `poll()`. Ces derniers permettent de gérer plusieurs connexions simultanément sans bloquer l'exécution. Par exemple, un serveur peut utiliser `select()` afin de surveiller plusieurs *sockets* en même temps et réagir lorsqu'une connexion devient prête à être lue ou écrite, améliorant ainsi la réactivité et l'efficacité du serveur.
Il est à noter que `select()` et `poll()` ne sont pas limités à une utilisation avec des *sockets*. Ces appels système permettent également de gérer efficacement toutes les entrées/sorties multiples sans bloquer le programme, et en attendant que l'un des descripteurs soit prêt pour une opération.

6.6.5 Mode connecté

Sockets côté serveur

Un serveur fournit un service à des clients : il doit donc attendre une requête, puis il doit la traiter. Les fonctions d'attente et de traitement sont séparées, afin de permettre au serveur d'accepter de nouvelles demandes pendant qu'il traite les requêtes en cours.

Un serveur en mode connecté utilise deux *sockets* :

- Le **socket serveur** qui est associé à un port connu, comme par exemple le port 80 s'il s'agit d'un serveur Web.
- Le **socket de communication** qui est associé au même numéro de port que le *socket* serveur, mais avec un descripteur différent.
- Lors d'une demande de connexion par un client sur le *socket* serveur, le serveur crée un *socket* de communication pour établir la liaison bidirectionnelle entre lui et le client.

Étapes de création de sockets du côté serveur

Étape n°1 : créer un socket

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

serversocket = socket(AF_INET, SOCK_STREAM, 0);
```

Étape n°2 : associer un nom et un port au socket Pour cela on crée une variable `serveraddr` de type `sockaddr_in` de manière à spécifier le port d'écoute.

```
serveraddr.sin_port = port;
...
bind(serversocket, serveraddr, sizeof(serveraddr));
```

Étape n°3 : indiquer qu'il s'agit d'un socket serveur

```
#define QUEUE_SIZE 10

listen(serversocket, QUEUE_SIZE) ;
```

Un *socket* serveur est en attente de demandes de connexion -- si une demande arrive pendant qu'une autre est en cours de traitement elle est placée dans une file d'attente (10 éléments ici), et si une demande arrive alors que la file est pleine elle est rejetée, mais pourra être réitérée ultérieurement.

Étape n°4a : se mettre en attente des demandes client

```
comsocket = accept(serversocket, &clientaddr, &addrlen);
```

L'appel système `accept()` est bloquant.

Étape n°4b : après acceptation d'une demande

- `comsocket` contient le descripteur du *socket* de communication créé ;
- `clientaddr` et `addrlen` sont respectivement l'adresse et la taille de la structure `sockaddr_in` du client dont la demande de connexion a été acceptée.

Étapes de création de sockets du côté client

On suppose connus l'adresse et le port du serveur.

Étape n°1 : créer un socket

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

clientsocket = socket(AF_INET, SOCK_STREAM, 0);
```

L'opération est identique à la création d'un *socket* serveur.

Étape n°2 : établir une connexion entre le socket et le serveur

```
struct sockaddr_in serveraddr;
/* ... */
/* remplir serveraddr avec adresse et port du serveur */
connect(clientsocket, &serveraddr, sizeof(serveraddr));
/* retourne 0 si succès, 1 si échec */
```

Le numéro de port associé au *socket* client est alloué par le système -- client et serveur peuvent maintenant dialoguer sur la connexion.

Résumé des fonctions pour les sockets client/serveur

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

/* créer un socket client ou serveur et retourner son descripteur */
int socket(int domain, int type, int protocol);
```

```
/* associer un socket à une structure de description */
int bind(int sockfd, struct sockaddr *addr, int addrlen);

/* déclarer un socket serveur */
int listen(int sockfd, int maxqueuesize);

/* placer un socket serveur en attente de connexions */
int accept(int sockfd, struct sockaddr *addr, int *addrlen);

/* envoyer une demande de connexion à un serveur -
 * serveur et numéro de port sont spécifiés dans addr */
int connect(int sockfd, struct sockaddr *addr, int addrlen);
```

Échanges sur une connexion entre sockets

- Une fois la connexion établie, le client et le serveur disposent chacun d'un descripteur vers l'extrémité correspondante de la connexion.
- Ce descripteur est analogue à un descripteur de fichier : on peut l'utiliser pour les opérations `read()` et `write()`, puis on le ferme par un appel à `close()`[1].
- On leur préférera `recv()` et `send()`[2].

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

ssize_t recv(int sockfd, void *buf, size_t len, int flags);
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

Client/serveur en mode itératif

Les étapes précédentes permettent de réaliser un serveur en mode itératif où un seul client est servi à la fois :

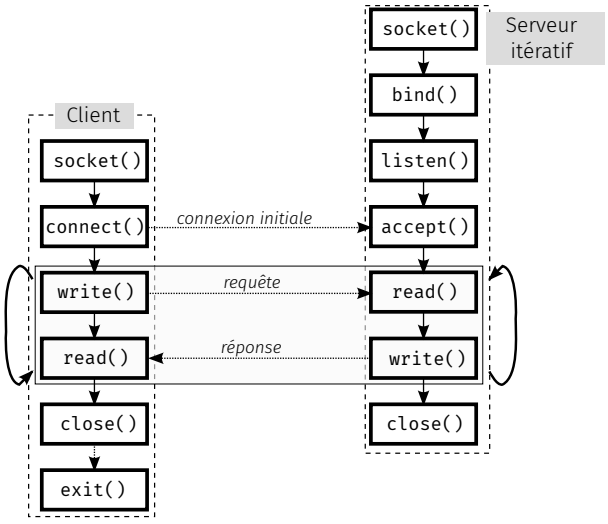


Fig. 6.11 : Client/serveur en mode itératif.

[1] Attention, `lseek()` qui permet de positionner un pointeur de lecture/écriture dans un fichier, n'a aucun sens dans le cas des *sockets* !
[2] L'appel système `send()` peut n'envoyer qu'une partie des données et il est donc nécessaire de vérifier sa valeur de retour qui doit être différente de -1.

Client/serveur en mode concurrent

Pour réaliser un serveur en mode concurrent, une solution consiste pour le programme principal (le *veilleur*) à créer un processus enfant (l'*exécutant*) pour chaque demande de connexion, le *veilleur* ne réalisant que la boucle d'attente sur les demandes de connexion :

```
...
s_veilleur = socket(/* ... */);
bind(/* ... */);
listen(s_veilleur, /* ... */);
while (1) {
    s_executant = accept(s_veilleur, /* ... */);
    if (fork() == 0) {
        close(s_veilleur);
        executer_service(s_executant, /* ... */);
        close(s_executant);
    } else {
        close(s_executant);
    }
}
...
```

D'autres solutions existent comme l'utilisation de *threads*.

Les étapes précédentes permettent de réaliser un serveur en mode concurrent où plusieurs clients peuvent être servis en même temps :

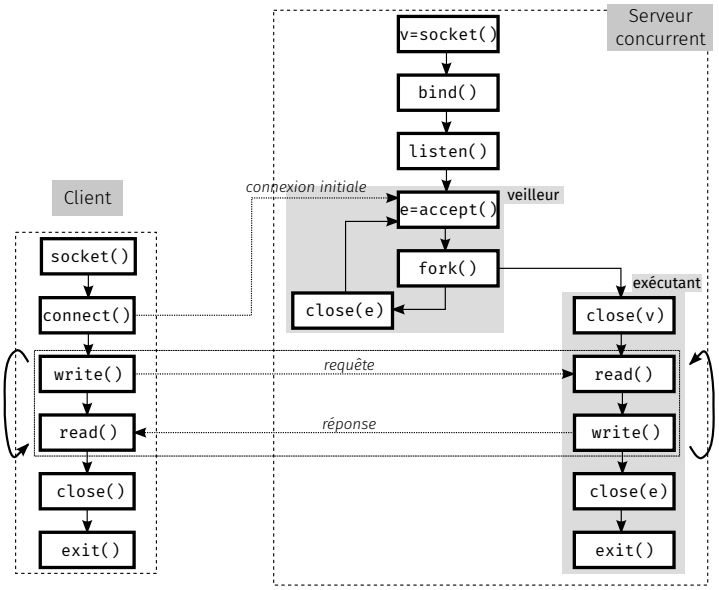


Fig. 6.12 : Client/serveur en mode concurrent.

6.6.6 Mode non connecté (UDP)

Appelé **mode datagramme**, les échanges au travers de *sockets* en mode non connecté utilisent, outre `socket()` et `bind()`, les fonctions `sendto()` et `recvfrom()` pour envoyer et recevoir des messages de 2 Kio maximum.

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

ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t
 ↪  *addrlen);
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *dest_addr,
 ↪  socklen_t addrlen);
```

Échange de messages en mode non connecté

Les appels système précédents permettent de réaliser un échange de messages entre deux systèmes distants :

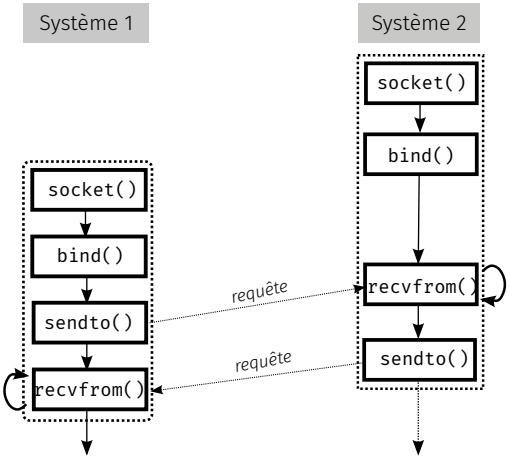


Fig. 6.13 : Emetteur/récepteur en mode non connecté.

Exemple d'un récepteur de datagrammes

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#define PORT     4950
#define MAXBUFLEN  100

int main(int argc, char *argv[]) {
    int sockfd, addr_len, numbytes;
    struct sockaddr_in address, remoteaddr;
    char buf[MAXBUFLEN];

    if ((sockfd = socket(AF_INET,SOCK_DGRAM,0)) == -1) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    address.sin_family = AF_INET;
```

```
    address.sin_port = htons(PORT);
    address.sin_addr.s_addr = INADDR_ANY; /* A local IP addr. */
    bzero(&(address.sin_zero), 8);        /* Zero for everything else. */

    if (bind(sockfd,(struct sockaddr *)&address,sizeof(struct sockaddr)) == -1) {
        perror("bind");
        exit(EXIT_FAILURE);
    }

    addr_len = sizeof(struct sockaddr);
    if ((numbytes = recvfrom(sockfd, buf, MAXBUFLEN, 0,
                  (struct sockaddr *)&remoteaddress,
                  &addr_len)) == -1) {
        perror("recvfrom");
        exit(EXIT_FAILURE);
    }

    printf("Got packet from %s\n", (char *)inet_ntoa(their_addr.sin_addr));
    printf("The packet's size is %d bytes\n", numbytes);
    buf[numbytes] = \0;
    printf("Packet content: %s\n",buf);

    close(sockfd);

    return EXIT_SUCCESS;
}
```

Exemple d'un émetteur de datagrammes

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#define PORT 4950

int main(int argc, char *argv[]) {
    int sockfd, numbytes;
    struct sockaddr_in remoteaddress;
    struct hostent *he;

    if (argc != 3) {
        fprintf(stderr, "Usage: sender host message\n");
        exit(EXIT_FAILURE);
    }

    if ((he = gethostbyname(argv[1])) == NULL) { /* récupère infos hote */
        perror("gethostbyname");
```

```
        exit(EXIT_FAILURE);
    }

    if ((sockfd = socket(AF_INET,SOCK_DGRAM,0)) == -1) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    remoteaddress.sin_family = AF_INET;
    remoteaddress.sin_port   = htons(PORT);
    remoteaddress.sin_addr   = *((struct in_addr *)he->h_addr);
    bzero(&(remoteaddress.sin_zero), 8);

    if ((numbytes = sendto(sockfd, argv[2], strlen(argv[2]), 0,
                (struct sockaddr *)&remoteaddress,
                sizeof(struct sockaddr))) == -1) {
        perror("recvfrom");
        exit(EXIT_FAILURE);
    }
    printf("Send %d bytes to %s\n", numbytes, (char *)inet_ntoa(remoteaddress.sin_addr));

    close(sockfd);

    return EXIT_SUCCESS;
}
```

Le lecteur qui souhaite approfondir cette section peut consulter l'ouvrage de Tanenbaum (2021) pour une vision globale des réseaux et ceux de Blaess et Kerrish pour une approche plus technique [[36] ; Blaess2019 ; [26]].

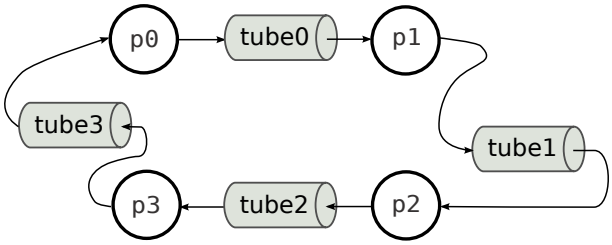
6.7 Exercices

Exercice 1

Écrire un programme en C qui crée un processus enfant. Le processus parent doit envoyer des signaux `SIGUSR1` et `SIGUSR2` à l'enfant, qui doit les capturer et afficher des messages appropriés.

Exercice 2

Soient N processus ayant une filiation entre eux. Ces processus communiquent au moyen de tubes de communication anonymes. Chaque processus partage deux tubes (un avec le processus en aval et un autre avec le processus en amont). Par exemple pour $N = 4$, les processus communiquent selon le schéma suivant :



Complétez le code de la fonction `main()` ci-dessous de manière à implémenter l'architecture de communication des N processus créés. L'entrée standard et la sortie standard de chaque processus p_i sont redirigées vers les tubes appropriés. Par exemple, pour le processus p_0, l'entrée et la sortie standards deviennent respectivement les tubes *tube3* et *tube0*.

```
#include /* what we need ... */

#define N 4

void proc(int pid);

int main(void) {
    int i;
    /** Begin of A **/
    /** End of A **/

    for (i = 0; i < N; i++) {
        if (fork() == 0) {
            /** Begin of B **/
            /** End of B **/
            proc(i);
            exit(EXIT_SUCCESS);
        }
    }
    return EXIT_SUCCESS;
}
```

Exercice 3

Créer un serveur et son client qui utilisent des *sockets* et le domaine `AF_INET` afin de communiquer. Le client envoie "Hello server" au serveur qui doit répondre "Welcome dear client".

Exercice 4

Créer deux processus qui communiquent à l'aide d'un segment de mémoire partagée. Le processus parent écrit le message "Bonjour du parent !" dans la mémoire partagée, et le processus enfant lit ce message, l'affiche, puis écrit "Bonjour de l'enfant !" que le parent lit et affiche ensuite.

Exercice 5

Écrire un programme en C qui utilise les files de messages POSIX pour la communication entre un processus parent et un processus enfant. Le processus parent doit envoyer des messages de différentes priorités à l'enfant, et l'enfant doit les recevoir et les afficher en ordre de priorité.

7 Synchronisation des processus

7.1 Introduction

Dans un système multitâches, plusieurs processus peuvent s'exécuter simultanément, partageant les mêmes ressources matérielles et logicielles. Sans une synchronisation appropriée, les processus peuvent entrer en conflit lorsqu'ils accèdent aux ressources partagées, ce qui peut entraîner des résultats imprévisibles et des erreurs difficiles à diagnostiquer. Par exemple, des systèmes temps réel critiques comme ceux utilisés dans l'aviation ou les soins médicaux nécessitent une synchronisation rigoureuse pour éviter les comportements indésirables.

La synchronisation va permettre de :

- assurer l'intégrité des données en évitant les accès concurrents indésirables ;
- coordonner les actions entre les processus pour qu'ils puissent coopérer efficacement ;
- Prévenir les situations de compétition et les interblocages susceptibles de paralyser le système.

Types d'interactions : compétition et coopération

Les processus peuvent interagir de deux manières principales :

- **Compétition** : Lorsqu'un processus accède à une ressource que d'autres processus souhaitent également utiliser, il y a compétition. Par exemple, plusieurs processus peuvent vouloir écrire dans un même fichier. La synchronisation est alors nécessaire afin de garantir que les accès soient gérés correctement et éviter les conflits. Un exemple classique est celui de plusieurs processus qui écrivent dans un fichier journal.
- **Coopération** : Les processus coopèrent lorsqu'ils travaillent ensemble pour accomplir une tâche commune. Par exemple, un processus peut produire des données que d'autres processus consomment. La synchronisation assure que les processus coopèrent efficacement, en respectant l'ordre des opérations et en évitant les interférences. Un exemple courant est le modèle producteur/consommateur, où un processus produit des données placées dans un tampon et un autre les consomme.

Dans la section qui suit nous aborderont la compétition entre processus, en commençant par le problème de l'exclusion mutuelle et en présentant des exemples concrets.

7.2 Compétition entre processus

Problème de l'exclusion mutuelle

L'**exclusion mutuelle** est une condition nécessaire lorsqu'il s'agit d'éviter les conflits quand plusieurs processus accèdent à une ressource partagée. Lorsqu'un processus entre dans une section critique pour accéder à une ressource, il doit s'assurer qu'aucun autre processus n'est en train de faire la même chose.

Exemple concret

Prenons l'exemple de deux processus accédant à un compte bancaire partagé. Sans mécanisme d'exclusion mutuelle, les opérations de mise à jour du solde peuvent s'entrelacer de manière imprévue, conduisant à des résultats incorrects.

```
unsigned int account_balance = 1000;

void withdraw_money(unsigned int amount) {
    unsigned int account_balance_temp = account_balance;
    account_balance_temp -= amount;
    account_balance = account_balance_temp;
}

void deposit_money(unsigned int amount) {
    unsigned int account_balance_temp = account_balance;
    account_balance_temp += amount;
    account_balance = account_balance_temp;
}
```

Dans cet exemple, si deux processus tentent de modifier le solde simultanément, ils peuvent entraîner une incohérence des données. Par exemple, si le processus p_1 retire 200 unités et le processus p_2 en retire 100 en même temps, le solde final pourrait ne pas refléter correctement ces deux opérations. Ces deux processus sont en situation de compétition (on parlera de *race condition* en anglais).

Mécanismes et solutions d'exclusion mutuelle

Une **section critique** est une partie du code où un processus accède à une ressource partagée. Pour éviter les conflits, toutes les opérations dans une section critique doivent être exécutées de manière **atomique**, c'est-à-dire sans interruption. Une **action atomique** est une opération qui est effectuée entièrement ou pas du tout, sans être interrompue.

Les sections critiques doivent être courtes de manière à minimiser le temps pendant lequel les autres processus sont bloqués. Voici un exemple simple en C :

```
void critical_section() {
    /* Enter critical section */

    /*
     * Working on the shared resource...
     */

    /* Get out of critical section */
}
```

Verrou

Un **verrou** est un mécanisme utilisé pour assurer qu'un seul processus à la fois peut entrer dans une section critique. L'exemple suivant montre comment mettre en oeuvre un verrou qui utilise un **mutex** de la bibliothèque *Pthread* afin de garantir l'exclusion mutuelle du solde dans les fonctions `withdraw_money()` et `deposit_money()` précédentes.

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
unsigned int account_balance = 1000;

void withdraw_money(unsigned int amount) {
    pthread_mutex_lock(&mutex);
    unsigned int account_balance_temp = account_balance;
    account_balance_temp -= amount;
    account_balance = account_balance_temp;
    pthread_mutex_unlock(&mutex);
}

void deposit_money(unsigned int amount) {
    pthread_mutex_lock(&mutex);
    unsigned int account_balance_temp = account_balance;
    account_balance_temp += amount;
    account_balance = account_balance_temp;
    pthread_mutex_unlock(&mutex);
}
```

Les fonctions `pthread_mutex_lock()` et `pthread_mutex_unlock()` déclaréees dans "`pthread.h`" réalisent respectivement la prise d'un verrou et sa restitution. Les verrous ont l'avantage d'être simples à implémenter, mais ils peuvent entraîner des blocages si un processus oublie de libérer le verrou.

Sémaphore

Un sémaphore est un mécanisme proposé par **Dijkstra** pour gérer l'accès concurrent [10]. Il généralise la notion de verrou. Un sémaphore peut être utilisé pour gérer l'exclusion mutuelle ainsi que la synchronisation entre processus.

Il existe deux types principaux de sémaphores :

- le **sémaphore binaire** qui fonctionne comme un verrou ;
- le **sémaphore de comptage** qui permettent un accès multiple à une ressource.

L'entête "`semaphore.h`" définit le type `sem_t` ainsi que les fonctions `sem_open()` pour la création ou l'ouverture en sémaphore binaire ou de comptage, et `sem_wait()` et `sem_post()` qui réalisent respectivement la prise du sémaphore et sa restitution. L'exemple suivant reprend l'exemple du solde bancaire.

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

sem_t *sem;
unsigned int account_balance = 1000;

void withdraw_money(unsigned int amount) {
    sem_wait(sem);
    account_balance -= amount;
    sem_post(sem);
}

void deposit_money(unsigned int amount) {
```

```
    sem_wait(sem);
    account_balance += amount;
    sem_post(sem);
}

int main(void) {
    if ((sem = sem_open("account_balance", O_CREAT | O_EXCL, S_IRUSR | S_IWUSR | S_IRGRP, 1)) ==
     ↪  SEM_FAILED) {
        perror("sem_open");
        exit(EXIT_FAILURE);
    }
    pthread_t t1, t2;

    pthread_create(&t1, NULL, (void *)withdraw_money, (void *)200);
    pthread_create(&t2, NULL, (void *)deposit_money, (void *)300);

    pthread_join(t1, NULL); /* Wait for both threads to complete */
    pthread_join(t2, NULL);

    printf("Final account balance: %u\n", account_balance);
    sem_unlink("account_balance");

    return EXIT_SUCCESS;
}
```

Les sémaphores sont plus flexibles que les verrous, mais leur utilisation incorrecte peut malgré tout entraîner des blocages ou des fuites de sémaphore.

Moniteur

Un **moniteur** est une structure de synchronisation qui regroupe des procédures et des données partagées. Un moniteur garantit que les procédures sont exécutées en exclusion mutuelle et il utilise pour cela des variables de condition pour gérer la synchronisation. Un moniteur encapsule verrou et variable de condition, ce qui simplifie son utilisation et réduit le risque d'erreurs.

Voici à nouveau l'exemple du solde bancaire protégé cette fois-ci par un moniteur qui utilise la bibliothèque *Pthread* :

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

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond   = PTHREAD_COND_INITIALIZER;
unsigned int account_balance = 1000;

void enter_critical_section() {
    pthread_mutex_lock(&mutex);
}

void get_out_critical_section() {
    pthread_mutex_unlock(&mutex);
}
```

```
void wait_for_condition() {
    pthread_cond_wait(&cond, &mutex);
}

void signal_condition() {
    pthread_cond_signal(&cond);
}

void withdraw_money(unsigned int amount) {
    enter_critical_section();
    while (account_balance < amount) {
        wait_for_condition();
    }
    account_balance -= amount;
    get_out_critical_section();
}

void deposit_money(unsigned int amount) {
    enter_critical_section();
    account_balance += amount;
    signal_condition();
    get_out_critical_section();
}
```

Les moniteurs simplifient la gestion des sections critiques et des conditions, mais leur implémentation peut être complexe dans certains langages.

Algorithmes et techniques

Attente active L'**attente** active consiste à faire tourner un processus en boucle en attendant qu'une condition soit remplie. Cette méthode est simple mais peut être inefficace, car elle consomme inutilement des cycles processeur. Elle est parfois utilisée dans des environnements où les sections critiques sont très courtes.

Appels système fournis par le noyau Le noyau fournit des appels système pour gérer l'exclusion mutuelle et la synchronisation (sémaphores, moniteurs, etc.). Ces appels sont généralement atomiques et assurent une gestion correcte des ressources partagées. Les appels système `sem_wait()` et `sem_post()` en sont des exemples.

Algorithme de Peterson (à titre éducatif) L'algorithme de Peterson est un algorithme classique pour gérer l'exclusion mutuelle. Il fonctionne bien pour deux processus, mais est limité dans les environnements modernes.

```
bool flag[2];
int turn;

void enter_region(int process) {
    int other = 1 - process;
    flag[process] = true;
    turn = other;
    while (flag[other] && turn == other) {}
}
```

```
void leave_region(int process) {
    flag[process] = false;
}
```

Algorithme de la boulangerie (*Lamport's Bakery algorithm*) Cet algorithme proposé par Leslie Lamport est conçu pour gérer l'exclusion mutuelle dans des environnements avec plus de deux processus [21]. Il utilise un système de tickets pour gérer l'ordre d'accès aux ressources.

```
#define N 10
bool choosing[N];
int number[N];

void lock(int i) {
    choosing[i] = true;
    number[i] = max(number) + 1;
    choosing[i] = false;

    for (int j = 0; j < N; j++) {
        while (choosing[j]) {}
        while (number[j] != 0 && (number[j] < number[i] || (number[j] == number[i] && j < i))) {}
    }
}

void unlock(int i) {
    number[i] = 0;
}
```

Verrouillage par test et modification (*test-and-set*) Le verrouillage par **test et modification** est une technique atomique où un processus teste et modifie une variable en une seule opération. Cette technique est efficace pour l'exclusion mutuelle dans des environnements multiprocesseurs.

```
bool lock = false;

bool test_and_set(bool *target) {
    bool rv = *target;
    *target = true;
    return rv;
}

void enter_critical_section() {
    while (test_and_set(&lock));
}

void leave_critical_section() {
    lock = false;
}
```

Le verrouillage par test et modification peut provoquer une attente active, mais il est généralement efficace pour les sections critiques courtes.

La section suivante abordera le verrouillage de fichier, ses opérations, des exemples sous Unix, et les problèmes liés à ce type de verrouillage.

7.3 Verrouillage de fichiers

Le verrouillage de fichiers est une technique utilisée pour contrôler l'accès concurrent à des fichiers et éviter les conflits et la corruption des données. Il permet de garantir que lorsqu'un processus accède à un fichier, aucun autre ne peut modifier le fichier simultanément.

Opérations de verrouillage

Le verrouillage de fichiers sous Unix peut être réalisé à l'aide de deux fonctions de la bibliothèque standard : `fcntl()` et `lockf()`.

Utilisation de `fcntl()`

La fonction `fcntl()` offre une flexibilité considérable. Voici un exemple d'utilisation :

```
#include <fcntl.h>
#include <unistd.h>

int lock_file(int fd) {
    struct flock lock;

    lock.l_type   = F_WRLCK;  /* Lock writing */
    lock.l_whence = SEEK_SET;
    lock.l_start  = 0;
    lock.l_len    = 0;        /* Lock the entire file */

    return fcntl(fd, F_SETLK, &lock);
}

int unlock_file(int fd) {
    struct flock lock;

    lock.l_type   = F_UNLCK;  /* Unlock */
    lock.l_whence = SEEK_SET;
    lock.l_start  = 0;
    lock.l_len    = 0;        /* Unlock the entire file */

    return fcntl(fd, F_SETLK, &lock);
}
```

Dans cet exemple, `lock_file()` verrouille le fichier pour une écriture exclusive, tandis que `unlock_file()` le déverrouille. La structure `flock` spécifie le type de verrou, la position de départ et la longueur du verrou.

Utilisation de `lockf()`

La fonction `lockf()` fournit une interface plus simple pour le verrouillage de fichiers, mais elle offre moins de flexibilité que la précédente. Voici un exemple d'utilisation :

```
#include <unistd.h>
#include <fcntl.h>
```

```
int lock_file(int fd) {
    return lockf(fd, F_LOCK, 0);  /* Lock the entire file */
}

int unlock_file(int fd) {
    return lockf(fd, F_ULOCK, 0); /* Unlock the entire file */
}
```

`lockf()` fournit des opérations de verrouillage et de déverrouillage simples. Dans cet exemple, `lock_file()` verrouille le fichier, et `unlock_file()` le déverrouille.

Exemples de verrouillage de fichiers sous Unix

Voici un exemple complet illustrant l'utilisation des verrous de fichiers avec `fcntl()` :

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

int main(void) {
    int fd = open("example.txt", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR | S_IRGRP);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    if (lock_file(fd) == -1) {
        perror("lock_file");
        close(fd);
        exit(EXIT_FAILURE);
    }

    printf("File locked. Press Enter to unlock and exit...\n");
    getchar();

    if (unlock_file(fd) == -1) {
        perror("unlock_file");
    }

    close(fd);
    return EXIT_SUCCESS;
}

int lock_file(int fd) {
    struct flock lock;

    lock.l_type   = F_WRLCK;
    lock.l_whence = SEEK_SET;
    lock.l_start  = 0;
    lock.l_len    = 0;

    return fcntl(fd, F_SETLK, &lock);
```

```
}

int unlock_file(int fd) {
    struct flock lock;

    lock.l_type   = F_UNLCK;
    lock.l_whence = SEEK_SET;
    lock.l_start  = 0;
    lock.l_len    = 0;

    return fcntl(fd, F_SETLK, &lock);
}
```

Dans cet exemple, le fichier "example.txt" est ouvert et verrouillé pour écriture. Le programme attend ensuite que l'utilisateur appuie sur Enter avant de déverrouiller et de fermer le fichier.

Problèmes liés au verrouillage de fichiers

Bien que le verrouillage de fichiers soit essentiel pour éviter la corruption des données, il présente également plusieurs défis et problèmes :

1. **Blocage** : un processus qui verrouille un fichier et ne le déverrouille pas correctement peut entraîner le blocage d'autres processus qui en attendent l'accès.
2. **Famine** : un processus peut être indéfiniment retardé s'il ne parvient pas à obtenir le verrou sur un fichier.
3. **Granularité des verrous** : le choix de verrouiller la totalité d'un fichier entier ou seulement une partie de celui-ci peut avoir un impact significatif sur la performance et la concurrence. *A contrario*, verrouiller de petites sous-parties peut améliorer la concurrence, mais augmente la complexité de la gestion des verrous.
4. **Portabilité** : les mécanismes de verrouillage de fichiers peuvent varier d'un système d'exploitation à l'autre, ce qui peut poser des problèmes de portabilité pour les applications qui doivent fonctionner sur plusieurs plateformes.

Afin d'éviter ces problèmes, il est nécessaire de planifier soigneusement l'utilisation des verrous et de s'assurer que ceux-ci sont toujours libérés correctement, même en cas d'erreur.

La section suivante aborde le problème de l'**interblocage**, ses conditions, des exemples classiques, ainsi que des stratégies afin de l'éviter.

7.4 Interblocage

Un **interblocage** (ou *deadlock*) est une situation où un ensemble de processus est bloqué parce que chaque processus attend qu'un autre libère une ressource. Les interblocages peuvent paralyser jusqu'à la totalité d'un système, rendant obligatoire leur prévention et leur gestion, aussi bien lors de la conception des systèmes d'exploitation que d'applications concurrentes.

Conditions nécessaires pour l'interblocage

Pour qu'un interblocage se produise, quatre conditions doivent être remplies. Ces conditions sont connues sous le nom de **conditions de Coffman** [7,32 -- chap. 7] :

1. **Exclusion mutuelle** : au moins une ressource doit être en mode non partageable. Autrement dit, un seul processus peut utiliser la ressource à un moment donné.
2. **Maintien et attente** : un processus détient au moins une ressource et attend d'obtenir des ressources supplémentaires détenues par d'autres processus.
3. **Non-préemption** : les ressources ne peuvent pas être préemptées ; elles ne peuvent être libérées que par le processus qui les détient.
4. **Attente circulaire** : il existe un ensemble de processus $\{P_1, P_2, \cdots, P_n\}$ tel que P_1 attend une ressource détenue par P_2, P_2 attend une ressource détenue par P_3, $\cdots$, et P_n attend une ressource détenue par P_1.

Exemples classiques

Exemple n°1 : le dîner des philosophes

Le problème du dîner des philosophes est un exemple classique d'interblocage. Cinq philosophes sont assis autour d'une table circulaire. Entre chaque paire de philosophes se trouve une seule baguette. Pour manger, un philosophe a besoin des deux baguettes adjacentes. Un interblocage se produit si chaque philosophe prend la baguette à sa droite et attend la baguette à sa gauche.

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

#define N 5

pthread_mutex_t chopsticks[N];

void think(int id) {
    printf("Philosopher No%d is thinking.\n", id);
    sleep(rand() % 3);
}

void eat(int id) {
    printf("Philosopher No%d is eating.\n", id);
    sleep(rand() % 2);
}

void *philosopher(void* num) {
    int id = *(int*)num;

    while (1) {
        think(id);

        /* Take the chopsticks */
        pthread_mutex_lock(&chopsticks[id]);
        pthread_mutex_lock(&chopsticks[(id + 1) % N]);

        eat(id);

        /* Put the chopsticks */
        pthread_mutex_unlock(&chopsticks[id]);
        pthread_mutex_unlock(&chopsticks[(id + 1) % N]);
```

```
    }
    return NULL;
}

int main(void) {
    pthread_t thread_id[N];
    int ids[N];

    for (int i = 0; i < N; i++) {
        pthread_mutex_init(&chopsticks[i], NULL);
    }

    for (int i = 0; i < N; i++) {
        ids[i] = i;
        pthread_create(&thread_id[i], NULL, philosopher, &ids[i]);
    }

    for (int i = 0; i < N; i++) {
        pthread_join(thread_id[i], NULL);
    }

    for (int i = 0; i < N; i++) {
        pthread_mutex_destroy(&chopsticks[i]);
    }

    return EXIT_SUCCESS;
}
```

Exemple n°2 : système de verrouillage de fichiers

Imaginons deux processus, p_1 et p_2, qui nécessitent chacun l'accès à deux fichiers f_1 et f_2, pour effectuer leurs opérations. Si p_1 verrouille le fichier f_1 et attend que f_2 soit disponible, tandis que p_2 verrouille le fichier f_2 et attend que f_1 soit disponible, les deux processus se retrouveront en situation d'interblocage. C'est ce qu'illustre la figure 7.1.

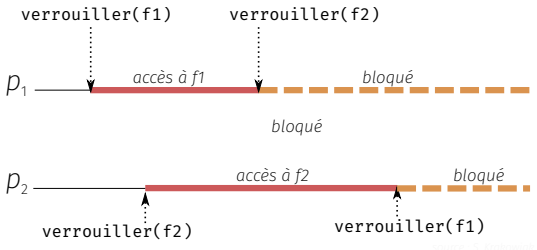


Fig. 7.1 : Interblocage lors de l'accès à deux fichiers.

Stratégies pour éviter l'interblocage

Prévention

La prévention des interblocages consiste à structurer le système de manière à ce que l'une des quatre conditions nécessaires à l'interblocage ne puisse jamais se produire. Voici les quatre techniques les plus courantes [32 -- chap.

7] :

1. **Exclusion mutuelle** : rendre certaines ressources partagées, mais cela n'est pas toujours possible ;
2. **Maintien et attente** : exiger que les processus demandent toutes les ressources dont ils auront besoin en une seule fois, et ne les gardent pas lorsqu'elles ne sont pas nécessaires.
3. **Non-préemption** : permettre la préemption des ressources --- si un processus accaparant certaines ressources fait une demande qui ne peut pas être immédiatement satisfaite, il doit libérer ses ressources actuelles et les redemander toutes ensemble.
4. **Attente circulaire** : imposer une numérotation ordonnée des ressources et exiger que chaque processus demande les ressources dans un ordre croissant de numéros comme le réalisent p_1 et p_2 sur la figure 7.2 afin de résoudre l'interblocage d'accès aux fichiers.

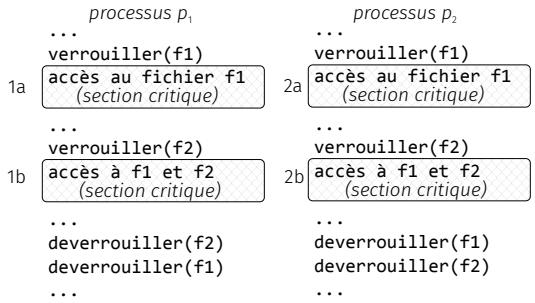


Fig. 7.2 : Résolution de l'interblocage lors de l'accès à deux fichiers.

Détection

La détection des interblocages implique de surveiller les états du système de manière à déterminer si un interblocage est en cours. Un algorithme de détection d'interblocage utilise une matrice d'allocation des ressources et une matrice des demandes de manière à identifier les éventuels cycles dans un graphe d'allocation des ressources. Trouver les cycles dans un graphe orienté peut se faire par exemple à l'aide de l'algoritme de Tarjan [37].

Si un interblocage est détecté, le système peut :

- terminer un ou plusieurs processus afin de briser le cycle ;
- préempter certaines des ressources pour les processus impliqués.

Graphe d'allocation des ressources Ce graphe est utilisé pour modéliser et analyser l'état des ressources et des processus, et détecter puis prévenir les interblocages [32 -- chap. 7]. Le graphe est constitué de deux types de noeuds et deux types d'arêtes :

- Les noeuds sont soit des processus représentés par des cercles, soit des ressources représentées par des carrés. Chaque type de ressource peut avoir plusieurs instances, représentées par des points à l'intérieur du carré.
- Quant aux arêtes, on distingue :
  - les arêtes de demande dirigées du noeud d'un processus à un noeud de ressource pour indiquer que ce processus a fait une demande pour cette ressource ;
  - les arêtes d'allocation dirigées du noeud d'une ressource au noeud d'un processus afin d'indiquer que la ressource a été allouée à ce processus.

Matrices d'allocation des ressources et des demandes Les arêtes d'allocation de ressources à chacun des processus permettent de définir la matrice d'allocation des ressources (les ressources en colonne et les processus en ligne).

De même, la matrice des demandes est réalisée à partir des arêtes de demande de chaque processus pour chacune des ressources.

Exemple Supposons un système avec trois types de ressources (R_1, R_2, R_3) et trois processus (p_1, p_2, p_3) qui accèdent à ces ressources.

Les matrices d'allocation et de demande sont respectivement :

	R_1	R_2	R_3
p_1	1	0	1
p_2	1	1	0
p_3	0	0	1

et

	R_1	R_2	R_3
p_1	0	1	0
p_2	0	0	1
p_3	1	0	0

Sur le graphe d'allocation des ressources de la figure 7.3, on peut observer les cycles suivants qui entrainent des interblocages :

- $p_1 \to R_2 \to p_2 \to R_1 \to p_1$
- $p_3 \to R_1 \to p_1 \to R_2 \to p_2 \to R_3 \to p_3$

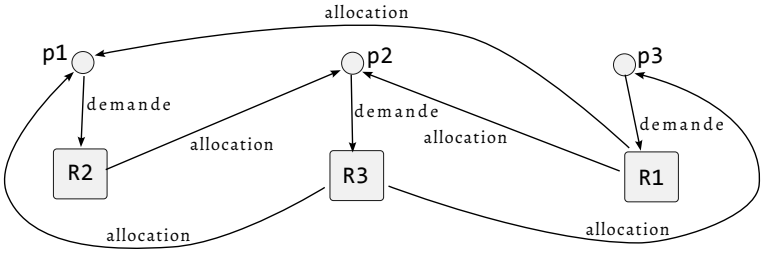


Fig. 7.3 : Exemple de graphe d'allocation des ressources.

Récupération

La récupération d'un interblocage implique des actions pour libérer les ressources et permettre aux processus de continuer. On peut réaliser au choix :

- terminaison de processus : terminer un ou plusieurs processus pour libérer leurs ressources.
- préemption des ressources : forcer certains processus à libérer des ressources, qui seront ensuite attribuées aux autres processus.

Dans l'exemple du graphe d'allocation des ressources de la figure 7.3, l'administrateur du système pourra choisir de terminer un processus afin de briser les cycles. Par exemple, en terminant le processus p_1 qui apparaît dans les deux cycles, il est possible de libérer les ressources R_1 et R_3, et donc de débloquer les processus p_2 et p_3.

La section suivante aborde la coopération entre processus, y compris le modèle producteur/consommateur, des exemples de coopération comme le lecteur et l'imprimeur, et la synchronisation par rendez-vous.

7.5 Coopération entre processus

Modèle producteur/consommateur

Le modèle **producteur/consommateur** est un paradigme classique de la programmation concurrente. Dans ce modèle, des processus producteurs génèrent des données et les placent dans un tampon, tandis que des processus consommateurs récupèrent ces données du tampon afin de les traiter. La synchronisation est nécessaire pour assurer que les producteurs ne remplissent pas le tampon lorsqu'il est plein et que les consommateurs ne le vident pas lorsqu'il est vide.

Exemple de modèle producteur/consommateur avec sémaphores

Voici un exemple en C utilisant des sémaphores pour synchroniser un producteur et un consommateur partageant un tampon de taille fixe :

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

#define BUFFER_SIZE 10

char buffer[BUFFER_SIZE];
int in  = 0;
int out = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;

void *producer(void* arg) {
    char item;

    while (1) {
        item = 'A' + (rand() % 26);
        pthread_mutex_lock(&mutex);
        while ((in + 1) % BUFFER_SIZE == out) {
            pthread_cond_wait(&not_full, &mutex);
        }
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
        printf("Produced: %c\n", item);
        pthread_cond_signal(&not_empty);
        pthread_mutex_unlock(&mutex);
        sleep(rand() % 2);
```

```
    }
}

void *consumer(void* arg) {
    char item;

    while (1) {
        pthread_mutex_lock(&mutex);
        while (in == out) {
            pthread_cond_wait(&not_empty, &mutex);
        }
        item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        printf("Consumed: %c\n", item);
        pthread_cond_signal(&not_full);
        pthread_mutex_unlock(&mutex);
        sleep(rand() % 2);
    }
}

int main(void) {
    pthread_t prod_tid, cons_tid;

    pthread_create(&prod_tid, NULL, producer, NULL);
    pthread_create(&cons_tid, NULL, consumer, NULL);

    pthread_join(prod_tid, NULL);
    pthread_join(cons_tid, NULL);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&not_empty);
    pthread_cond_destroy(&not_full);

    return EXIT_SUCCESS;
}
```

Dans cet exemple, les sémaphores `empty` et `full` sont utilisés pour suivre le nombre de places libres ainsi que celles qui sont déjà occupées dans le tampon, tandis que le sémaphore `mutex` assure l'exclusion mutuelle lors de l'accès au tampon.

Exemple de coopération : lecteur/imprimeur

Un exemple courant de coopération entre processus est le modèle **lecteur/imprimeur**. Dans ce modèle, un processus lecteur lit les données d'une source (comme un fichier ou un capteur) et les place dans un tampon, tandis qu'un processus imprimeur récupère les données de ce tampon et les transmet à une imprimante ou à un autre dispositif de sortie.

Voici un exemple en C utilisant des sémaphores pour synchroniser un lecteur et un imprimeur :

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

```c
#define BUFFER_SIZE 10

char buffer[BUFFER_SIZE];
int in  = 0;
int out = 0;

sem_t empty, full, mutex;

void *reader(void* arg) {
    char item;

    while (1) {
        item = 'A' + (rand() % 26);
        sem_wait(&empty);
        sem_wait(&mutex);
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
        printf("Read: %c\n", item);
        sem_post(&mutex);
        sem_post(&full);
        sleep(rand() % 2);
    }
}

void *printer(void* arg) {
    char item;

    while (1) {
        sem_wait(&full);
        sem_wait(&mutex);
        item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        printf("Printed: %c\n", item);
        sem_post(&mutex);
        sem_post(&empty);
        sleep(rand() % 2);
    }
}

int main(void) {
    pthread_t reader_tid, printer_tid;

    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    sem_init(&mutex, 0, 1);

    pthread_create(&reader_tid, NULL, reader, NULL);
    pthread_create(&printer_tid, NULL, printer, NULL);

    pthread_join(reader_tid, NULL);
    pthread_join(printer_tid, NULL);
```

```
    sem_destroy(&empty);
    sem_destroy(&full);
    sem_destroy(&mutex);

    return EXIT_SUCCESS;
}
```

7.5.1 Synchronisation par rendez-vous -

La synchronisation par **rendez-vous** est une technique où deux ou plusieurs processus se synchronisent à un point précis avant de continuer leur exécution. Cette technique est couramment utilisée dans les systèmes temps réel où des actions coordonnées sont nécessaires.

Exemple pratique de rendez-vous

Voici un exemple utilisant une barrière (`pthread_barrier_t`) permettant de synchroniser deux *threads* à un point de rendez-vous :

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

pthread_barrier_t barrier;

void *task1(void* arg) {
    printf("Task 1 part 1\n");
    sleep(1);
    pthread_barrier_wait(&barrier);
    printf("Task 1 part 2\n");
    return NULL;
}

void *task2(void* arg) {
    printf("Task 2 part 1\n");
    sleep(2);
    pthread_barrier_wait(&barrier);
    printf("Task 2 part 2\n");
    return NULL;
}

int main(void) {
    pthread_t tid1, tid2;

    pthread_barrier_init(&barrier, NULL, 2);

    pthread_create(&tid1, NULL, task1, NULL);
    pthread_create(&tid2, NULL, task2, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    pthread_barrier_destroy(&barrier);
```

```
    return EXIT_SUCCESS;
}
```

Dans cet exemple, `task1` et `task2` exécutent chacun une première partie de leur travail, puis ils se synchronisent au point de rendez-vous (la barrière) avant de continuer.

7.6 Conclusion

Récapitulatif des concepts clés

La synchronisation des processus est une composante essentielle de la programmation système sous Unix. Tout au long de ce chapitre, nous avons exploré divers aspects et mécanismes de synchronisation, chacun ayant ses propres avantages et inconvénients.

1. **Importance de la synchronisation** :
    - Garantir l'intégrité des données.
    - Coordonner les actions entre processus pour une coopération efficace.
    - Prévenir les situations de compétition et les interblocages.
2. **Types d'interactions** :
    - **Compétition** : nécessite des mécanismes pour gérer l'exclusion mutuelle.
    - **Coopération** : nécessite des mécanismes pour synchroniser les processus de manière à accomplir une tâche commune.
3. **Compétition entre processus** :
    - **Exclusion mutuelle** : empêcher plusieurs processus d'accéder simultanément à des ressources partagées.
    - **Mécanismes** :
        - Sections critiques et actions atomiques.
        - Verrous et sémaphores.
        - Moniteurs pour une gestion plus simple des sections critiques et des conditions.
    - **Algorithmes et techniques** :
        - Attente active.
        - Appels système fournis par le noyau (sémaphores).
        - Algorithmes pour l'exclusion mutuelle (Peterson ou algorithme de la boulangerie).
        - Verrouillage par test et modification.
4. **Verrouillage de fichiers** :
    - Utilisation de `fcntl()` et `lockf()` pour gérer les accès concurrents aux fichiers.
    - Nécessité d'éviter les blocages et la famine.
5. **Interblocage (*deadlock*)** :
    - Conditions nécessaires pour l'interblocage : exclusion mutuelle, maintien et attente, non-préemption, attente circulaire.
    - Exemples classiques tels que le dîner des philosophes et les verrouillage de fichiers.
    - Stratégies pour éviter l'interblocage : prévention, détection et récupération.
6. **Coopération entre processus** :

- Modèle producteur/consommateur pour une gestion efficace des tâches concurrentes.
- Exemple de coopération : lecteur/imprimeur.
- Synchronisation par rendez-vous pour des actions coordonnées.

Importance de la synchronisation pour la stabilité et la sécurité des systèmes

La synchronisation est nécessaire pour maintenir la stabilité et la sécurité des systèmes multitâches. Elle permet de garantir que les processus peuvent fonctionner de manière harmonieuse, sans interférer les uns avec les autres. Les mécanismes de synchronisation tels que les verrous, les sémaphores, les moniteurs et les algorithmes d'exclusion mutuelle jouent un rôle vital dans la prévention des conditions de compétition, des interblocages et des incohérences des données.

En conclusion, une bonne compréhension et une utilisation correcte de ces techniques sont essentielles pour un développeur. En maîtrisant ces concepts, il est possible de concevoir des applications multiprocessus robustes et fiables, capables de gérer efficacement la concurrence et la coopération.

7.7 Exercices

Exercice 1 : Une synchronisation ratée

Voici deux versions d'un programmes permettant à deux processus (ou deux *threads*) d'incrémenter un compteur partagé sans synchronisation appropriée.

Version utilisant fork()

```
#define NB_ITERS 100000000

unsigned int *cnt; /* shared */

void count();

int main(void) {
    pid_t tid1, tid2;
    int shmd;

    shmd = shm_open("/cnt", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR | S_IRGRP);
    ftruncate(shmd, sizeof(unsigned int));
    cnt = mmap(NULL, sizeof(unsigned int), PROT_READ | PROT_WRITE, MAP_SHARED, shmd, 0);

    tid1 = fork();
    if (tid1 == 0) {
        count();
        exit(EXIT_SUCCESS);
    } else if (tid1 > 0) {
        tid2 = fork();
        if (tid2 == 0) {
            count();
            exit(EXIT_SUCCESS);
        }
```

```
        wait(NULL);
        wait(NULL);
    }

    if (*cnt != (unsigned)NB_ITERS * 2) {
        printf("Error: cnt=%u\n", *cnt);
    } else {
        printf("cnt=%u\n", *cnt);
    }

    munmap(cnt, sizeof(unsigned int));
    close(shmd);
    shm_unlink("/cnt");

    return EXIT_SUCCESS;
}

void count() {
    int i;

    for (i = 0; i < NB_ITERS; i++) {
        (*cnt)++;
    }
}
```

Version utilisant des threads

```
#define NB_ITERS 100000000

unsigned int cnt = 0; /* shared */

void *count(void *arg);

int main(void) {
    pthread_t tid1, tid2;

    pthread_create(&tid1, NULL, count, NULL);
    pthread_create(&tid2, NULL, count, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    if (cnt != (unsigned)NB_ITERS * 2) {
        printf("Error: cnt=%d\n", cnt);
    } else {
        printf("cnt=%d\n", cnt);
    }

    return EXIT_SUCCESS;
}

void *count(void *arg) {
```

```
    int i;

    for (i = 0; i < NB_ITERS; i++) {
        cnt++;
    }
    return NULL;
}
```

Ces deux versions donnent en sortie des résultats incorrects à chaque fois :

```
$ ./cnt-fork
Error: cnt=65486848 # cnt devrait valoir 200 000 000
$ ./cnt-fork
Error: cnt=113868800 # cnt devrait valoir 200 000 000
$
$ ./cnt-thread
Error: cnt=94832671 # cnt devrait valoir 200 000 000
$ ./cnt-thread
Error: cnt=101488317 # cnt devrait valoir 200 000 000
...
```

1. Expliquez les raisons qui ont amené à ces résultats imprévisibles.
2. Utilisez un des mécanismes de synchronisation vus dans ce chapitre (verrous, sémaphores, etc.) pour corriger le code ci-dessus de manière à ce qu'il produise le résultat attendu.

8 Études de cas

8.1 Introduction

Ce chapitre présente une série d'études de cas qui illustrent l'application des concepts abordés dans les chapitres précédents. Chaque étude de cas est conçue pour mettre en oeuvre tout ou partie des mécanismes fondamentaux de la programmation système en C sous Unix dans des contextes reflétant des situations réelles que vous pourrez rencontrer.

Fonctions communes

Les fonctions suivantes sont utilisées dans plusieurs études de cas.

Fonction de journalisation

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

/**
 * @brief Logs a given event to "system.log".
 *
 * @param event_desc The description of the event to be logged.
 * @return true if the event was logged successfully, false otherwise.
 */
bool log_event(const char *event_desc) {
    FILE *logfile = fopen("system.log", "a");
    if (!logfile) {
        perror("Failed to open log file");
        return false;
    }
    time_t now = time(NULL);
    fprintf(logfile, "%s: %s\n", ctime(&now), event_desc);
    fclose(logfile);
    return true;
}
```

Fonctions de gestion d'une mémoire partagée

```c
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
```

```c
#include <errno.h>
#include <stdbool.h>

/**
 * @brief Sets up shared memory.
 *
 * @param shm_fd Pointer to the file descriptor for the shared memory.
 * @param shared_memory Pointer to the shared memory region.
 * @param size The size of the shared memory region.
 * @param name The name of the shared memory object.
 * @return true if the shared memory was set up successfully, false otherwise.
 */
bool setup_shared_memory(int *shm_fd, void **shared_memory, size_t size,
                         const char *name) {
    *shm_fd = shm_open(name, O_CREAT | O_RDWR, S_IRUSR | S_IWUSR | S_IRGRP);
    if (*shm_fd == -1) {
        perror("shm_open");
        return false;
    }
    if (ftruncate(*shm_fd, size) == -1) {
        perror("ftruncate");
        close(*shm_fd);
        return false;
    }
    *shared_memory = mmap(NULL, size, PROT_READ | PROT_WRITE,
                          MAP_SHARED, *shm_fd, 0);
    if (*shared_memory == MAP_FAILED) {
        perror("mmap");
        close(*shm_fd);
        return false;
    }
    return true;
}

/**
 * @brief Cleans up shared memory.
 *
 * @param shm_fd The file descriptor for the shared memory.
 * @param shared_memory Pointer to the shared memory region.
 * @param size The size of the shared memory region.
 * @param name The name of the shared memory object.
 */
void cleanup_shared_memory(int shm_fd, void *shared_memory, size_t size,
                           const char *name) {
    munmap(shared_memory, size);
    close(shm_fd);
    shm_unlink(name);
}
```

8.2 Étude de cas n°1 : cybersécurité

Contexte

Dans le domaine de la cybersécurité, la gestion des permissions et la sécurité des accès aux fichiers sont des éléments importants. Les applications doivent être conçues de manière à empêcher tout accès non autorisé aux données sensibles. Une application intéressante serait la mise en place d'une surveillance des accès aux fichiers afin de détecter et de prévenir toute tentative non autorisée.

Objectif

Pour cette étude de cas, nous devons développer un service qui :

1. Surveille les accès aux fichiers sensibles.
2. Enregistre les tentatives d'accès dans un journal de sécurité.
3. Restreint les permissions d'accès aux fichiers selon des rôles définis.
4. Alerte le propriétaire (ou mieux l'administrateur) en cas d'accès non autorisé.

Méthodologie

Nous allons implémenter deux approches pour cette surveillance :

1. **Interrogation périodique** (*polling*). Cette méthode implique un contrôle des fichiers sensibles à intervalles réguliers pour détecter toute modification.
2. **API *inotify***. Cette méthode offre une surveillance plus réactive en capturant les événements directement depuis le noyau Linux lorsqu'un fichier (ou un dossier) est accédé, modifié, ou ses attributs sont changés.

Implémentation

Approche n°1 : interrogation périodique

L'interrogation périodique est une méthode simple qui consiste à vérifier régulièrement les attributs des fichiers (par exemple, l'heure de la dernière modification) afin de détecter des changements.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <time.h>

#define FILE_PATH     "sensitive_file"
#define POLL_INTERVAL 5 /* in seconds */

bool log_event(const char *event_desc);

/**
 * @brief Sends a notification to the administrator using the Unix "mail"
```

```
 * command.
 *
 * @param message The notification message.
 */
void notify_user(const char *message) {
    char command[256];

    snprintf(command, sizeof(command),
             "echo \"%s\" | mail -s \"Security alert\" admin@ensicaen.fr",
             message);
    system(command);
}

int main(void) {
    const char *filepath = FILE_PATH;
    struct stat file_stat;    /* A stat structure to store file information */
    time_t last_mod_time = 0; /* Initialize the last modification time */

    while (1) { /* Loop indefinitely to monitor the file */
        if (stat(filepath, &file_stat) == -1) {
            perror("stat");
            exit(EXIT_FAILURE);
        }

        if (last_mod_time != file_stat.st_mtime) { /* Check if the file has been modified */
            last_mod_time = file_stat.st_mtime; /* Update the last modification time */

            char event_desc[256];
            snprintf(event_desc, sizeof(event_desc),
                     "File %s accessed or modified at %s",
                     filepath, ctime(&file_stat.st_mtime));
            if (!log_event(event_desc)) {
                fprintf(stderr, "Failed to log event. Exiting.\n");
                return EXIT_FAILURE;
            }
            notify_user(event_desc);
        }

        sleep(POLL_INTERVAL); /* Wait for the poll interval (e.g. 5 seconds) */
    }

    return EXIT_SUCCESS;
}
```

L'interrogation périodique est très simple à mettre en oeuvre, mais elle peut être inefficace car elle consomme des ressources même lorsque rien ne se passe. Pour une utilisation dans des environnements critiques, l'intervalle entre chaque interrogation doit être soigneusement choisi de manière à minimiser la charge tout en détectant les accès en temps utile.

Approche n°2 : utilisation de l'API inotify

L'API *inotify*, propre à Linux, permet de surveiller les événements du système de fichiers en temps réel, tels que l'accès, la modification ou le changement des attributs des fichiers. Contrairement à l'interrogation périodique, cette méthode est réactive et ne consomme des ressources que lorsque des événements se produisent.

Pour utiliser *inotify*, il est nécessaire de suivre les étapes suivantes :

1. Créer une instance de *inotify* en appelant la fonction `inotify_init()`. Cette dernière retourne un descripteur de fichier qui représente l'instance inotify.
2. Ajouter un ou plusieurs fichiers (ou dossiers) à surveiller en appelant la fonction `inotify_add_watch()` de manière à indiquer quels sont les fichiers ou dossiers à surveiller, ainsi que les événements spécifiques à surveiller comme les accès ou les modifications.
3. La fonction `read()` sur le descripteur obtenu en 1 permet de lire les événements lorsqu'ils se produisent.
4. Une fois la surveillance terminée, on fait appel à `inotify_rm_watch()` afin d'enlever les fichiers ou dossiers de la liste de surveillance, puis à `close()` pour fermer le descripteur de fichier *inotify*.

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <sys/inotify.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <syslog.h>

#define EVENT_SIZE    (sizeof(struct inotify_event))
#define EVENT_BUF_LEN (1024 * (EVENT_SIZE + 16))
#define FILE_PATH     "sensitive_file"

bool log_event(const char *event_desc);

/**
 * @brief Sends a notification to the administrator.
 *
 * @param message The notification message.
 */
void notify_user(const char *message) {
    char command[256];
    snprintf(command, sizeof(command),
             "echo \"%s\" | mail -s \"Security alert\" admin@ensicaen.fr",
             message);
    system(command);
}

int main(void) {
    int inotify_fd; /* inotify file descriptor */
    int watch_fd;   /* watch file descriptor */
    char buffer[EVENT_BUF_LEN];

    inotify_fd = inotify_init(); /* Initialize the inotify instance */
    if (inotify_fd < 0) {
        perror("inotify_init");
```

```
        exit(EXIT_FAILURE);
    }

    /* Watch the file for access, modify, and attribute changes */
    watch_fd = inotify_add_watch(inotify_fd, FILE_PATH,
                                 IN_ACCESS | IN_MODIFY | IN_ATTRIB);
    if (watch_fd == -1) {
        perror("inotify_add_watch");
        exit(EXIT_FAILURE);
    }

    while (1) { /* Loop indefinitely to monitor the file for events */
        /* Read inotify events from the file descriptor */
        int length = read(inotify_fd, buffer, EVENT_BUF_LEN);
        if (length < 0) {
            perror("read");
            exit(EXIT_FAILURE);
        }

        int i = 0; /* Initialize a counter for parsing inotify events */
        while (i < length) { /* Parse each inotify event */
            /* Get the current inotify event */
            struct inotify_event *event = (struct inotify_event *) &buffer[i];
            if (event->len) { /* Check if the event has a filename associated with it */
                char event_desc[256]; /* A buffer for the event description */
                if (event->mask & IN_ACCESS) { /* Check if the file was accessed */
                    snprintf(event_desc, sizeof(event_desc),
                             "File %s accessed", FILE_PATH);
                } else if (event->mask & IN_MODIFY) { /* Check if the file was modified */
                    snprintf(event_desc, sizeof(event_desc),
                             "File %s modified", FILE_PATH);
                } else if (event->mask & IN_ATTRIB) { /* Check if the metadata changed */
                    snprintf(event_desc, sizeof(event_desc),
                             "Metadata of file %s changed", FILE_PATH);
                }

                if (!log_event(event_desc)) {
                    fprintf(stderr, "Failed to log event. Exiting.\n");
                }
                notify_user(event_desc);
            }
            i += EVENT_SIZE + event->len; /* Move to the next inotify event */
        }
    }

    inotify_rm_watch(inotify_fd, watch_fd); /* Remove the watch from the file descriptor */
    close(inotify_fd); /* Close the inotify file descriptor */

    return EXIT_SUCCESS;
}
```

inotify est déclenché par des événements spécifiques, ce qui réduit la consommation de ressources par rapport à une interrogation périodique. L'API capture des événements en temps réel, ce qui permet une réponse plus

rapide aux accès non autorisés.

Résultats et discussion

Les deux approches implémentées pour la surveillance des accès aux fichiers offrent des solutions différentes. L'interrogation périodique est simple à mettre en oeuvre et portable, mais elle peut être inefficace en raison de son utilisation intensive des ressources. En revanche, l'utilisation de l'API *inotify* est plus réactive et efficace, mais elle est spécifique aux systèmes Linux.

Perspectives d'amélioration

Voici quelques pistes d'amélioration de cette application :

- Transformer notre service en démon pour que la surveillance soit continue.
- Ajouter une analyse automatique des journaux afin d'y détecter des schémas d'accès suspects.
- Coupler la surveillance avec d'autres outils de sécurité pour une protection renforcée, par exemple AppArmor.

Pour approfondir les concepts de surveillance des accès aux fichiers et de sécurité des systèmes Unix en général, nous conseillons aux lecteurs de consulter les ouvrages de Stevens ou de Garfinkel [11,34]. En ce qui concerne l'API *inotify*, elle est détaillée par Kerrisk et Blaess [26 -- chap. 19,4 -- chap. 29].

8.3 Étude de cas n°2 : pipeline CI/CD

Contexte

Dans le domaine du *DevOps*[1], la gestion efficace d'un canal de validation et de mise en production continues, communément appelé *pipeline CI/CD* (*Continuous Integration/Continuous Deployment*), est nécessaire afin d'assurer des déploiements rapides et fiables. Un pipeline CI/CD automatisé permet d'orchestrer les processus de compilation, de test et de déploiement de manière systématique.

Objectif

Pour cette étude de cas, nous devons développer un service qui :

1. Met en oeuvre une file de messages POSIX pour gérer la file d'attente des tâches du pipeline.
2. Gère les signaux POSIX pour la notification et la synchronisation des étapes du pipeline.
3. Journalise les opérations pour le suivi et un audit ultérieur.

Méthodologie

Le service peut être décomposé en deux parties :

1. Un programme principal qui ajoute les tâches à la file de messages et attend leur achèvement.
2. Un programme collaborateur qui exécute les tâches qu'il reçoit en fonction de leur priorité, puis envoie des notifications une fois qu'elles sont terminées.

[1]Terme issu de la contraction des mots anglais « development » (développement) et « operations » (exploitation), est une pratique technique visant à l'unification du développement logiciel (*dev*) et de l'administration des infrastructures informatiques (*ops*), notamment l'administration système.

Implémentation

Programme principal

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mqueue.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <errno.h>
#include <signal.h>
#include <stdbool.h>

#define QUEUE_NAME "/ci_cd_queue"
#define MAX_SIZE   1024
#define MSG_STOP   "STOP"
#define LOG_FILE   "/var/log/ci_cd.log"

bool log_event(const char *event_desc);

/**
 * @brief Signal handler for received notifications.
 *
 * @param signum The signal number.
 */
void signal_handler(int signum) {
    log_event("Received notification signal");
}

/**
 * @brief Sets up the signal handler for notifications.
 *
 * This function configures the signal handler for SIGUSR1. When the signal
 * is received, the signal_handler function will be invoked.
 */
void setup_signal_handler() {
    struct sigaction sa;

    sa.sa_handler = signal_handler;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    if (sigaction(SIGUSR1, &sa, NULL) == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }
}

/**
 * @brief Creates the task queue for the CI/CD pipeline.
 */
void create_task_queue() {
```

```
    mqd_t mq;
    struct mq_attr attr;

    attr.mq_flags   = 0;
    attr.mq_maxmsg  = 10;
    attr.mq_msgsize = MAX_SIZE;
    attr.mq_curmsgs = 0;

    mq = mq_open(QUEUE_NAME, O_CREAT | O_RDWR,
                 S_IRUSR | S_IWUSR | S_IRGRP, &attr);
    if (mq == -1) {
        perror("mq_open");
        exit(EXIT_FAILURE);
    }
    mq_close(mq);
}

/**
 * @brief Adds a task to the CI/CD pipeline queue.
 *
 * This function sends a task to the POSIX message queue, which the collaborator
 * will then execute.
 *
 * @param task The task to be added to the queue.
 */
void add_task(const char *task) {
    mqd_t mq = mq_open(QUEUE_NAME, O_WRONLY);
    if (mq == -1) {
        perror("mq_open");
        exit(EXIT_FAILURE);
    }
    if (mq_send(mq, task, strlen(task), 0) == -1) {
        perror("mq_send");
    }
    mq_close(mq);
}

int main(void) {
    /* Set up the signal handler for task completion notifications */
    setup_signal_handler();

    /* Create the CI/CD task queue */
    create_task_queue();

    /* Add tasks to the queue */
    add_task("Step 1: Compile the code");
    add_task("Step 2: Run unit tests");
    add_task("Step 3: Deploy to staging environment");
    add_task(MSG_STOP); /* Add a special message to signal the end of tasks */

    pause();  /* Waiting for a completion signal */

    return EXIT_SUCCESS;
```

```
}
```

Collaborateur CI/CD

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mqueue.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>
#include <stdbool.h>

#define QUEUE_NAME "/ci_cd_queue"
#define MAX_SIZE   1024
#define LOG_FILE   "/var/log/ci_cd_worker.log"

bool log_event(const char *event_desc);

/**
 * @brief Executes a specific task in the CI/CD pipeline.
 *
 * This function performs the task described in the message received from the
 * message queue, and logs the results.
 *
 * @param task The task to be executed.
 */
void execute_task(const char *task) {
    log_event(task);
    if (strcmp(task, "Step 1: Compile the code") == 0) {
        system("make all");  /* Example compilation command */
        log_event("Compilation completed");
    } else if (strcmp(task, "Step 2: Run unit tests") == 0) {
        system("./test_suite.sh");  /* Example script for running tests */
        log_event("Unit tests completed");
    } else if (strcmp(task, "Step 3: Deploy to staging environment") == 0) {
        system("./deploy.sh");  /* Example deployment script */
        log_event("Deployment to staging environment completed");
    } else {
        log_event("Unknown task");
    }
    kill(getppid(), SIGUSR1);  /* Notify the manager about the task completion */
}

/**
 * @brief Worker function for handling tasks from the CI/CD pipeline queue.
 *
 * This function reads tasks from the message queue and executes them one by one.
 * It continues running until it encounters the "STOP" message.
 */
void worker() {
    mqd_t mq;
```

```
    char buffer[MAX_SIZE + 1];
    ssize_t bytes_read;

    mq = mq_open(QUEUE_NAME, O_RDONLY);
    if (mq == -1) {
        perror("mq_open");
        exit(EXIT_FAILURE);
    }

    while (1) {
        bytes_read = mq_receive(mq, buffer, MAX_SIZE, NULL);
        if (bytes_read >= 0) {
            buffer[bytes_read] = '\0';
            if (strncmp(buffer, "STOP", strlen("STOP")) == 0) {
                break;
            }
            execute_task(buffer);
        } else {
            perror("mq_receive");
        }
    }

    mq_close(mq);
}

int main(void) {
    pid_t pid = fork();
    if (pid == 0) {
        worker();
    } else if (pid > 0) {
        worker();
    } else {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    return EXIT_SUCCESS;
}
```

Résultats et discussion

Le service CI/CD proposé utilise les files de messages POSIX et les signaux afin d'orchestrer les différentes étapes du pipeline. La séparation des responsabilités entre le programme principal et le collaborateur CI/CD offre une gestion efficace et évolutive des tâches du pipeline.

Perspectives d'amélioration

Pour améliorer ce système, nous pourrions :

- Ajouter une interface graphique pour la gestion et le suivi des tâches du pipeline.
- Intégrer des outils de surveillance afin de suivre l'état et les performances des différentes étapes du pipeline.

- Mettre en place des notifications plus avancées, telles que des alertes SMS ou des notifications à l'aide de plateformes de messagerie instantanée.

Le lecteur qui désire approfondir la mise en oeuvre des pipelines CI/CD consultera l'ouvrage de Humble [18].

8.4 Étude de cas n°3 : monétique

Contexte

Dans le domaine de la monétique, la gestion des transactions bancaires est une tâche critique nécessitant une sécurité, une fiabilité et une performance élevées. Comme nous l'avons constaté au chapitre 7, les transactions doivent être traitées de manière concurrente tout en garantissant l'intégrité des données et la protection contre les accès non autorisés.

Objectif

Pour cette étude de cas, nous allons développer une application qui :

1. Gère plusieurs transactions bancaires simultanément.
2. Assure l'isolation et la sécurité des transactions.
3. Journalise les transactions pour le suivi et un audit ultérieur.
4. Utilise des processus de manière à garantir l'indépendance et l'isolation des transactions.

Méthodologie

Nous allons utiliser les mécanismes suivants :

- Des processus enfants pour la gestion de chaque transaction.
- Des tubes anonymes pour la communication entre le processus parent et ses processus enfants.
- Un journal afin de suivre les transactions.

Implémentation

Dans l'exemple suivant, chaque processus enfant gère une unique transaction, tandis que le processus parent continue de recevoir des demandes pour lancer de nouvelles transactions.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <string.h>
#include <stdbool.h>

#define MAX_TRANSACTIONS 10
#define FRAUD_THRESHOLD  100000

bool log_event(const char *event_desc);
```

```c
/**
 * @brief Simulates a fraud detection mechanism.
 *
 * @param transaction_detail The detail of the transaction to check.
 * @return true if fraud is detected, false otherwise.
 */
bool detect_fraud(const char *transaction_detail) {
    int amount;
    sscanf(transaction_detail, "Transaction %*d: amount %d", &amount);
    return amount > FRAUD_THRESHOLD;
}

/**
 * @brief Processes a transaction by writing detail message to a pipe and
 *        logging the transaction.
 *
 * @param transaction_id The ID of the transaction being processed.
 * @param pipe_fd A pair of file descriptors representing the read and
 *                write ends of the anonymous pipe.
 */
void process_transaction(int transaction_id, int pipe_fd[2]) {
    char transaction_detail[256];

    close(pipe_fd[0]); /* Close read end */
    snprintf(transaction_detail, sizeof(transaction_detail),
             "Transaction %d: amount %d", transaction_id, rand() % 200000);
    if (write(pipe_fd[1], transaction_detail,
              strlen(transaction_detail) + 1) == -1) {
        perror("Failed to write to pipe");
    }
    close(pipe_fd[1]); /* Close write end */

    if (detect_fraud(transaction_detail)) {
        log_event("Fraud detected!");
    }

    if (!log_event(transaction_detail)) {
        fprintf(stderr, "Failed to log transaction %d\n", transaction_id);
    }
    printf("Transaction %d processed successfully\n", transaction_id);
}

int main(void) {
    int transaction_id = 0;
    int pipe_fd[2];

    if (pipe(pipe_fd) == -1) {
        perror("pipe");
        return EXIT_FAILURE;
    }

    while (transaction_id < MAX_TRANSACTIONS) {
        pid_t pid = fork();
```

```
        if (pid == 0) { /* Child process */
            process_transaction(transaction_id, pipe_fd);
            exit(EXIT_SUCCESS);
        } else if (pid > 0) { /* Parent process */
            transaction_id++;
        } else {
            perror("fork");
            return EXIT_FAILURE;
        }
    }

    /* Parent process waits for all child processes to complete */
    for (int i = 0; i < MAX_TRANSACTIONS; i++) {
        wait(NULL);
    }

    /* Read the pipe in parent process */
    char buffer[256];

    close(pipe_fd[1]); /* Close write end */
    while (read(pipe_fd[0], buffer, sizeof(buffer)) > 0) {
        printf("Transaction log: %s\n", buffer);
    }
    close(pipe_fd[0]); /* Close read end */

    return EXIT_SUCCESS;
}
```

Résultats et Discussion

Cette implémentation permet de traiter plusieurs transactions bancaires simultanément en utilisant des processus enfants de manière à garantir l'isolation et la sécurité. Chaque transaction est journalisée pour un suivi et un audit ultérieur, et un mécanisme de détection de fraude simple est mis en place pour détecter les transactions suspectes.

Perspectives d'amélioration

Pour aller plus loin, nous pourrions :

- Intégrer une base de données pour une gestion plus robuste et une persistance des transactions.
- Améliorer le mécanisme de détection de fraude en utilisant des algorithmes d'apprentissage.
- Mettre en place une infrastructure de surveillance des performances et des anomalies en temps réel.

Le lecteur qui désire approfondir l'aspect monétique pourra se reporter aux enseignements de l'école. Concernant les algorithmes associés à la cryptographie, l'ouvrage de référence est celui de B. Schneier [30].

8.5 Étude de cas n°4 : traitement d'images

Contexte

Dans le domaine du traitement des images, la nécessité de traiter de grandes quantités de données rapidement et efficacement est importante. Les applications doivent être capables de manipuler des images en haute résolution, souvent en parallèle, pour des tâches telles que le filtrage, l'amélioration ou l'analyse d'images.

Objectif

Pour cette étude de cas, nous allons développer une application qui :

1. Gère le traitement parallèle des images.
2. Utilise la mémoire virtuelle pour optimiser l'utilisation des ressources.
3. Journalise les opérations de traitement pour le suivi et un audit ultérieur.
4. Assure la communication entre les processus de manière à coordonner les tâches.

Méthodologie

Le traitement de grandes quantités de données nous amène à privilégier l'utilisation d'une mémoire partagée. Chaque tâche de traitement de l'image est associée à un processus enfant.

Implémentation

L'application simule le traitement d'une image en haute résolution en utilisant plusieurs processus. Chaque processus enfant accède à une région spécifique de la mémoire partagée contenant l'image, la modifie (ici, une inversion des pixels), puis journalise l'opération.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <string.h>
#include <errno.h>
#include <stdbool.h>

#define IMG_SHM        "/image_data"
#define NUM_PROCESSES  4
#define IMAGE_SIZE     (1024 * 1024 * 10)

bool log_event(const char *event_desc);
bool setup_shared_memory(int *shm_fd, void **shared_memory,
                         size_t size, const char *name);
void cleanup_shared_memory(int shm_fd, void *shared_memory, size_t size,
                           const char *name);

/**
```

```
 * @brief Processes a part of an image using shared memory and logs the
 *        processing detail.
 *
 * This function handles the processing of a specific part of an image by
 * inverting its pixels. The processed data is written back to the shared memory.
 *
 * @param part_id The ID of the part of the image being processed.
 * @param shared_memory A pointer to the shared memory buffer containing
 *        the image data.
 */
void process_image_part(int part_id, unsigned char *shared_memory) {
    char operation_detail[256];

    printf("Processing part %d of the image\n", part_id);
    for (int i = part_id * (IMAGE_SIZE / NUM_PROCESSES);
         i < (part_id + 1) * (IMAGE_SIZE / NUM_PROCESSES); i++) {
        shared_memory[i] = (unsigned char)(0xff - shared_memory[i]);
    }
    snprintf(operation_detail, sizeof(operation_detail),
             "Processed part %d of the image", part_id);
    if (!log_event(operation_detail)) {
        fprintf(stderr, "Failed to log operation for part %d\n", part_id);
    }
}

int main(void) {
    int shm_fd;
    unsigned char *shared_memory;
    pid_t pids[NUM_PROCESSES];

    if (!setup_shared_memory(&shm_fd, &shared_memory, IMAGE_SIZE, IMG_SHM)) {
        return EXIT_FAILURE;
    }

    for (int i = 0; i < NUM_PROCESSES; i++) {
        pids[i] = fork();
        if (pids[i] == 0) {
            process_image_part(i, shared_memory);
            exit(EXIT_SUCCESS);
        } else if (pids[i] < 0) {
            perror("fork");
            cleanup_shared_memory(shm_fd, shared_memory);
            return EXIT_FAILURE;
        }
    }

    for (int i = 0; i < NUM_PROCESSES; i++) {
        if (wait(NULL) == -1) {
            perror("wait");
        }
    }

    cleanup_shared_memory(shm_fd, shared_memory, IMAGE_SIZE, IMG_SHM);
```

```
    return EXIT_SUCCESS;
}
```

Résultats et discussion

L'utilisation de la mémoire partagée permet d'optimiser l'utilisation des ressources et de gérer le traitement parallèle des images. Chaque processus enfant traite une partie de l'image, ce qui accélère considérablement le traitement par rapport à une exécution séquentielle.

Perspectives d'amélioration

Pour aller plus loin, nous pourrions :

- Implémenter des algorithmes de traitement d'image plus avancés, tels que le filtrage convolutif, la transformée de Fourier, ou la détection de contours.
- Gérer des images de très grandes tailles en divisant l'image en blocs (pages) et en traitant chaque bloc individuellement.
- Intégrer des bibliothèques de traitement d'images comme *OpenCV* afin de bénéficier de fonctions optimisées et d'une plus grande flexibilité [28].

Le lecteur intéressé par les techniques de traitement d'images pourra consulter l'ouvrage de Gonzalez et Wood [14].

8.6 Étude de cas n°5 : intelligence artificielle

Contexte

Dans le domaine de l'intelligence artificielle, la gestion de calculs complexes et conséquents est essentielle. De nombreuses applications nécessitent un traitement parallèle pour des algorithmes d'apprentissage, d'analyse de données ou encore de traitement du langage naturel. La régression linéaire est un outil statistique largement utilisé en intelligence artificielle afin de modéliser les relations entre variables.

Objectif

Pour cette étude de cas, nous allons développer un système qui :

1. Implémente une régression linéaire pour modéliser la relation entre deux variables.
2. Utilise des *threads* pour paralléliser les calculs et optimiser les performances.
3. Assure la synchronisation entre *threads* à l'aide de mutex pour éviter l'apparition de situations de compétition.
4. Journalise les opérations pour le suivi et un audit ultérieur.

Régression linéaire

La régression linéaire est une technique d'analyse statistique qui permet de prédire une variable dépendante (souvent notée y) en fonction d'une ou plusieurs variables indépendantes (notées x). Le modèle de base est une équation linéaire de la forme :

$$y = a \cdot x + b$$

où a est le coefficient directeur (*slope*) et b est l'ordonnée à l'origine (*intercept*). L'objectif est de minimiser l'erreur de prédiction en ajustant les coefficients a et b pour qu'ils s'adaptent le mieux possible aux données observées.

En intelligence artificielle, la régression linéaire est utilisée pour des tâches de prédiction lorsque la relation entre les variables est supposée linéaire.

Méthodologie

Pour améliorer l'efficacité des calculs, nous utilisons des *threads* afin de paralléliser le processus de calcul des coefficients de régression. Chaque *thread* calcule une partie des sommes nécessaires pour déterminer les coefficients a et b. Pour éviter les conflits d'accès aux variables partagées, nous utilisons un mutex.

Chaque étape du calcul est journalisée pour assurer la traçabilité et faciliter le diagnostic en cas de problème.

Implémentation

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

#define NUM_THREADS    4
#define DATA_SIZE    1E9

double data_x[DATA_SIZE];
double data_y[DATA_SIZE];
double sum_x = 0;
double sum_y = 0;
double sum_xy = 0;
double sum_x2 = 0;

pthread_mutex_t mutex;

bool log_event(const char *event_desc);

/**
 * @brief Calculates partial sums of data and logs thread activity.
 *
 * This function calculates partial sums of X, Y, XY, and X^2 data elements
 * for a specific thread and range, then updates global sums using a mutex
 * lock. It also logs the thread's processing activity using
 * log_operation().
 *
 * @param thread_id A pointer to the current thread ID (long).
 */
void *calculate_partial_sums(void *thread_id) {
    long tid = (long)thread_id;
    int start = tid * (DATA_SIZE / NUM_THREADS);
    int end = (tid + 1) * (DATA_SIZE / NUM_THREADS);
```

```
    double local_sum_x = 0;
    double local_sum_y = 0;
    double local_sum_xy = 0;
    double local_sum_x2 = 0;

    for (int i = start; i < end; i++) {
        local_sum_x += data_x[i];
        local_sum_y += data_y[i];
        local_sum_xy += data_x[i] * data_y[i];
        local_sum_x2 += data_x[i] * data_x[i];
    }

    /* Lock the mutex before updating the global sums */
    pthread_mutex_lock(&mutex);
    sum_x += local_sum_x;
    sum_y += local_sum_y;
    sum_xy += local_sum_xy;
    sum_x2 += local_sum_x2;
    pthread_mutex_unlock(&mutex); /* Unlock the mutex */

    char log_entry[256];
    snprintf(log_entry, sizeof(log_entry),
             "Thread %ld processed data range %d to %d", tid, start, end);
    log_event(log_entry);

    pthread_exit(NULL);
}

/**
 * @brief Creates a specified number of threads and initializes them with
 * calculate_partial_sums.
 *
 * @param threads A pointer to an array where the created thread IDs will
 *                be stored.
 * @param num_threads The number of threads to create.
 * @return true if all threads were successfully created, false otherwise.
 */
bool create_threads(pthread_t *threads, int num_threads) {
    for (long t = 0; t < num_threads; t++) {
        int rc = pthread_create(&threads[t], NULL, calculate_partial_sums,
                                (void *)t);
        if (rc) {
            fprintf(stderr,
                    "ERROR; return code from pthread_create() is %d\n", rc);
            return false;
        }
    }
    return true;
}

/**
 * @brief Joins a specified number of threads and waits for their completion.
```

```
 *
 * @param threads A pointer to an array of thread IDs to be joined.
 * @param num_threads The number of threads to join.
 * @return true if all threads were successfully joined, false otherwise.
 */
bool join_threads(pthread_t *threads, int num_threads) {
    for (int t = 0; t < num_threads; t++) {
        int rc = pthread_join(threads[t], NULL);
        if (rc) {
            fprintf(stderr,
                    "ERROR; return code from pthread_join() is %d\n", rc);
            return false;
        }
    }
    return true;
}

int main(void) {
    pthread_t threads[NUM_THREADS];

    /* Initialize the mutex */
    pthread_mutex_init(&mutex, NULL);

    /* Initialize data (for simplicity, random values are used) */
    for (int i = 0; i < DATA_SIZE; i++) {
        data_x[i] = rand() % 100;
        data_y[i] = rand() % 100;
    }

    /* Create threads to calculate partial sums */
    if (!create_threads(threads, NUM_THREADS)) {
        pthread_mutex_destroy(&mutex);
        return EXIT_FAILURE;
    }

    /* Join threads after they finish execution */
    if (!join_threads(threads, NUM_THREADS)) {
        pthread_mutex_destroy(&mutex);
        return EXIT_FAILURE;
    }

    /* Destroy the mutex */
    pthread_mutex_destroy(&mutex);

    /* Calculate regression coefficients */
    double slope = (NUM_THREADS * sum_xy - sum_x * sum_y)
                   / (NUM_THREADS * sum_x2 - sum_x * sum_x);
    double intercept = (sum_y - slope * sum_x) / NUM_THREADS;

    printf("Linear regression model: y = %.2fx + %.2f\n", slope, intercept);

    return EXIT_SUCCESS;
}
```

Résultats et discussion

Cette étude de cas montre comment réaliser le traitement parallèle d'un calcul régression linéaire en utilisant des *threads* et un mutex. L'utilisation de *threads* permet de tirer parti des processeurs multicoeurs, tandis que le mutex assure une synchronisation efficace entre eux.

Perspectives d'amélioration

Pour aller plus loin, nous pourrions :

- Optimiser davantage la gestion des *threads* pour des charges de travail plus importantes.
- Utiliser des bibliothèques de calcul parallèle telles que *OpenMP* pour accroitre les performances.
- Intégrer des techniques d'apprentissage plus avancées pour des applications d'intelligence artificielle plus complexes.

Le lecteur intéressé par les concepts de l'intelligence artificielle en général pourra se reporter à l'ouvrage de référence de Russel et Norvig [Russel2020]. Pour ceux qui souhaiteraient se limiter au problème de la régression et en particulier la régression linéaire, vous pouvez consulter le chapitre 23 du livre de Zaki et Meira [42].

8.7 Étude de cas n°6 : simulation numérique

Contexte

La simulation numérique est essentielle pour modéliser des phénomènes physiques complexes, par exemple la diffusion de la chaleur dans un matériau. Ce type de simulation permet de comprendre comment la chaleur se propage dans un matériau en fonction du temps, ce qui est important dans des domaines tels que la science des matériaux. La diffusion de la chaleur est généralement modélisée par l'équation de la chaleur, une équation aux dérivées partielles qui décrit la distribution de la température dans un milieu en fonction du temps et de l'espace.

Problématique

La diffusion de la chaleur est régie par l'équation de la chaleur, qui, dans sa forme unidimensionnelle, est donnée par :

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$$

où :

- $u(x, t)$ est la température en fonction de la position x et du temps t ;
- α est la diffusivité thermique du matériau considéré.

Objectif

Pour cette étude de cas, nous allons développer une application qui consiste à résoudre numériquement l'équation de la chaleur afin de prédire comment la chaleur se répartit dans un matériau sur une période donnée. La solution numérique implique la discrétisation de l'espace et du temps, ce qui nous permet de calculer la température à des points spécifiques du matériau à des instants précis. Cette simulation :

1. Utilise la mémoire virtuelle pour optimiser l'utilisation des ressources.
2. Utilise des processus pour isoler les différentes parties de la simulation.
3. Journalise les opérations de simulation pour le suivi et un audit ultérieur.

Méthodologie

Discrétisation de l'équation de la chaleur

Pour résoudre numériquement l'équation de la chaleur, nous utilisons une méthode de différences finies. L'équation est discrétisée en espace et en temps, ce qui permet de transformer l'équation aux dérivées partielles en une équation de récurrence :

$$u_i^{n+1} = u_i^n + \alpha \frac{\Delta t}{(\Delta x)^2} \left(u_{i+1}^n - 2u_i^n + u_{i-1}^n \right)$$

où :

- u_i^n : température au point i à l'instant n ;
- Δt : pas de temps ;
- Δx : espacement entre les points de la grille.

Parallélisation avec des processus

Pour optimiser le calcul, la simulation est parallélisée en divisant le matériau en plusieurs segments, chaque processus s'occupant de la simulation sur un segment spécifique. Ces processus communiqueront au travers d'une mémoire partagée afin d'assurer la cohérence des calculs aux extrémités des segments.

Journalisation des opérations

Chaque processus journalise les résultats de la simulation de manière à réaliser une analyse *a posteriori*. Les journaux incluront des informations sur la progression de la simulation, les valeurs calculées aux extrémités et toute autre anomalie détectée.

Implémentation

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <string.h>
#include <errno.h>
#include <stdbool.h>

#define SIM_SHM        "/sim_data"
#define SHM_SIZE      (1024 * 1024 * 1024)
#define NUM_PROCESSES 4
```

```c
#define DX 0.01
#define DT 0.0001
#define ALPHA 0.01
#define TIME_STEPS 1000

bool log_event(const char *event_desc);
bool setup_shared_memory(int *shm_fd, void **shared_memory,
                         size_t size, const char *name);
void cleanup_shared_memory(int shm_fd, void *shared_memory, size_t size,
                           const char *name);

/**
 * @brief Simulates heat diffusion for a specific part of the shared memory.
 *
 * This function simulates heat diffusion by calculating the temperature at
 * each point in the grid over time, using the finite difference method.
 * The calculations are parallelized across multiple processes.
 *
 * @param part_id The ID of the part of the material being simulated.
 * @param shared_memory A pointer to the shared memory buffer where the
 *        simulation results are stored.
 */
void simulate_heat_diffusion(int part_id, double *shared_memory) {
    printf("Simulating heat diffusion part %d\n", part_id);
    int start = part_id * (SHM_SIZE / NUM_PROCESSES);
    int end = (part_id + 1) * (SHM_SIZE / NUM_PROCESSES);

    for (int t = 0; t < TIME_STEPS; t++) {
        for (int i = start; i < end; i++) {
            if (i == 0 || i == SHM_SIZE - 1) {
                continue; /* Boundary conditions */
            }
            shared_memory[i] = shared_memory[i] + ALPHA * DT / (DX * DX) *
                (shared_memory[i+1] - 2 * shared_memory[i] + shared_memory[i-1]);
        }
    }

    char operation_detail[256];
    snprintf(operation_detail, sizeof(operation_detail),
             "Simulated heat diffusion part %d", part_id);
    if (!log_event(operation_detail)) {
        fprintf(stderr, "Failed to log simulation part %d\n", part_id);
    }
}

int main(void) {
    int shm_fd;
    double *shared_memory;
    pid_t pids[NUM_PROCESSES];

    if (!setup_shared_memory(&shm_fd, (void **)&shared_memory, SHM_SIZE, SIM_SHM)) {
        return EXIT_FAILURE;
    }
```

```
    for (int i = 0; i < NUM_PROCESSES; i++) {
        pids[i] = fork();
        if (pids[i] == 0) {
            simulate_heat_diffusion(i, shared_memory);
            exit(EXIT_SUCCESS);
        } else if (pids[i] < 0) {
            perror("fork");
            cleanup_shared_memory(shm_fd, shared_memory, SHM_SIZE, SIM_SHM);
            return EXIT_FAILURE;
        }
    }

    for (int i = 0; i < NUM_PROCESSES; i++) {
        wait(NULL);
    }

    cleanup_shared_memory(shm_fd, shared_memory, SHM_SIZE, SIM_SHM);

    return EXIT_SUCCESS;
}
```

Résultats et discussion

Cette implémentation montre comment diviser un problème de simulation numérique complexe en sous-problèmes plus petits et paralléliser les calculs afin de gagner en efficacité. Chaque processus ne traite qu'une partie du matériau, calculant la diffusion de la chaleur dans la section qui lui est impartie. La mémoire partagée permet de coordonner les calculs aux extrémités des segments, de manière à garantir la cohérence globale de la simulation.

Perspectives d'amélioration

Pour aller plus loin, nous pourrions :

- Optimiser l'algorithme afin de réduire davantage les temps de calcul, par exemple en utilisant des méthodes numériques plus avancées.
- Gérer des matériaux plus complexes, avec des propriétés thermiques hétérogènes ou des géométries non linéaires.
- Intégrer des méthodes de visualisations des résultats afin de mieux comprendre la propagation de la chaleur.

Le lecteur qui désire en savoir plus sur les techniques de simulation numérique pourra se tourner vers l'ouvrage très complet de W. Press [29].

8.8 Étude de cas n°7 : développement de jeux vidéo

Contexte

Le développement de jeux vidéo nécessite la gestion de multiples aspects techniques, tels que le rendu graphique, la gestion des entrées utilisateur, la simulation physique ou encore la synchronisation des éléments du jeu. Les

jeux modernes doivent également gérer des ressources importantes tout en maintenant des performances élevées qui permettent d'offrir aux joueurs des expériences fluides et immersives.

Objectif

Pour cette étude de cas, nous devons développer une application qui :

1. Gère le rendu graphique et les entrées utilisateur.
2. Utilise des *threads* afin de paralléliser les tâches lourdes, comme la simulation physique des éléments du jeu.
3. Assure la synchronisation entre les différents composants du jeu.
4. Optimise l'utilisation des ressources système afin de maintenir des performances élevées tout au long du jeu.

Méthodologie

Nous utilisons des *threads* qui gèrent le rendu graphique, la simulation physique, et les entrées utilisateur, tout en assurant la synchronisation avec un mutex et une variable conditionnelle.

Implémentation

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>

#define NUM_THREADS 3

pthread_mutex_t render_mutex;
pthread_cond_t cond_var;

bool log_event(const char *event_desc);

/**
 * @brief Renders graphics in a separate thread.
 *
 * This function simulates the rendering process, locking the mutex and waiting
 * for a condition signal to start rendering. It logs the rendering activity
 * and simulates rendering time to maintain a 60 FPS rate.
 *
 * @param arg Not used in this implementation.
 */
void *render_graphics(void *arg) {
    while (1) {
        pthread_mutex_lock(&render_mutex);
        pthread_cond_wait(&cond_var, &render_mutex);

        printf("Rendering graphics...\n");
        log_event("Rendering graphics");
```

```
        /* Simulate the rendering time */
        usleep(16000); /* Approx. 60 FPS */

        pthread_mutex_unlock(&render_mutex);
    }
    pthread_exit(NULL);
}

/**
 * @brief Handles user input in a separate thread.
 *
 * This function simulates the handling of user input, locking the mutex and
 * signaling the condition to notify other threads. It logs the input handling
 * and simulates input processing time.
 *
 * @param arg Not used in this implementation.
 */
void *handle_input(void *arg) {
    while (1) {
        printf("Handling user input...\n");
        log_event("Handling user input");
        /* Simulate input handling time */
        usleep(10000);

        pthread_mutex_lock(&render_mutex);
        pthread_cond_signal(&cond_var);
        pthread_mutex_unlock(&render_mutex);

        usleep(50000); /* Sleep to simulate input rate */
    }
    pthread_exit(NULL);
}

/**
 * @brief Simulates physics in a separate thread.
 *
 * This function simulates the physics calculations of the game, locking the
 * mutex and signaling the condition to notify other threads. It logs the
 * physics simulation and simulates calculation time.
 *
 * @param arg Not used in this implementation.
 */
void *simulate_physics(void *arg) {
    while (1) {
        printf("Simulating physics...\n");
        log_event("Simulating physics");
        /* Simulate physics calculation time */
        usleep(20000);

        pthread_mutex_lock(&render_mutex);
        pthread_cond_signal(&cond_var);
        pthread_mutex_unlock(&render_mutex);
```

```
        usleep(50000); /* Sleep to simulate physics update rate */
    }
    pthread_exit(NULL);
}

int main(void) {
    pthread_t threads[NUM_THREADS];
    pthread_mutex_init(&render_mutex, NULL);
    pthread_cond_init(&cond_var, NULL);

    /* Create threads */
    if (pthread_create(&threads[0], NULL, render_graphics, NULL) != 0) {
        perror("Failed to create thread for rendering graphics");
        return EXIT_FAILURE;
    }
    if (pthread_create(&threads[1], NULL, handle_input, NULL) != 0) {
        perror("Failed to create thread for handling input");
        return EXIT_FAILURE;
    }
    if (pthread_create(&threads[2], NULL, simulate_physics, NULL) != 0) {
        perror("Failed to create thread for simulating physics");
        return EXIT_FAILURE;
    }

    /* Join threads */
    for (int t = 0; t < NUM_THREADS; t++) {
        if (pthread_join(threads[t], NULL) != 0) {
            perror("Failed to join thread");
            return EXIT_FAILURE;
        }
    }

    pthread_mutex_destroy(&render_mutex);
    pthread_cond_destroy(&cond_var);

    return EXIT_SUCCESS;
}
```

Résultats et discussion

Cette étude de cas montre comment gérer le rendu graphique, les entrées utilisateur et la simulation physique en utilisant des *threads* pour paralléliser les tâches lourdes. L'utilisation d'un mutex et d'une variable conditionnelle permet de synchroniser efficacement les différentes parties du jeu.

Perspectives d'amélioration

Pour aller plus loin, nous pourrions :

- Intégrer des moteurs de rendu graphique avancés comme *OpenGL* ou *Vulkan* pour améliorer la qualité et les performances graphiques.
- Implémenter des algorithmes de simulation physique plus réalistes et complexes.

- Optimiser l'architecture du jeu pour des plateformes spécifiques comme les consoles de jeux ou les appareils mobiles.

8.9 Étude de cas n°8 : gestion d'un système de voix sur IP (VoIP)

Contexte

Les systèmes de voix sur IP sont largement utilisés dans les communications modernes car ils permettent des appels vocaux et vidéo sur les réseaux IP. La gestion d'un système VoIP implique de maîtriser plusieurs paramètres tels que la gestion des ressources, la synchronisation ou encore la latence.

Objectif

Pour cette étude de cas, nous devons développer un système qui :

1. Gère les flux audio en temps réel pour les appels VoIP.
2. Utilise des processus pour isoler et gérer les flux de manière efficace.
3. Synchronise les processus de manière à minimiser la latence.
4. Journalise les traitements pour le suivi et un audit ultérieur.

Méthodologie

Capture du flux audio

Dans un système VoIP, la capture du flux audio est souvent réalisée à l'aide d'API spécifiques fournies par le système d'exploitation ou encore à l'aide de bibliothèques comme ALSA (*Advanced Linux Sound Architecture*) sous Linux ou CoreAudio sous Mac OS X. Le flux audio est capturé depuis le microphone, encodé en un format compressé, puis transmis sur le réseau IP.

Traitement et transmission du flux audio

Une fois capturé, le flux audio doit être traité et transmis à l'autre extrémité de la communication. Pour cela, des processus séparés peuvent être utilisés pour gérer l'encodage, le décodage, et la transmission des paquets audio. Chaque processus s'occupe d'une tâche spécifique, comme le traitement du flux audio ou la gestion des paquets réseau, et communique avec d'autres processus au travers d'une mémoire partagée ou d'autres mécanismes de communication interprocessus.

Dans notre exemple, nous utilisons des processus distincts pour simuler la manipulation du flux audio. La communication entre ces processus se fait à l'aide d'une mémoire partagée, ce qui permet une synchronisation efficace tout en minimisant la latence.

Implémentation

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
```

```c
#include <sys/mman.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <string.h>
#include <errno.h>
#include <stdbool.h>

#define VOIP_SHM      "/voip_data"
#define SHM_SIZE      4096
#define NUM_PROCESSES 2

bool log_event(const char *event_desc);
bool setup_shared_memory(int *shm_fd, void **shared_memory,
                         size_t size, const char *name);
void cleanup_shared_memory(int shm_fd, void *shared_memory, size_t size,
                           const char *name);

/**
 * @brief Handles an audio stream by inverting its bits.
 *
 * This function simulates the handling of an audio stream in a VoIP system.
 * It processes a segment of the shared memory and inverts the bits, representing
 * a simple manipulation of the audio data.
 *
 * @param stream_id The ID of the audio stream being handled.
 * @param shared_memory A pointer to the shared memory buffer where the stream
 *                      data is located.
 */
void handle_audio_stream(int stream_id, unsigned char *shared_memory) {
    printf("Handling audio stream %d\n", stream_id);
    for (int i = stream_id * (SHM_SIZE / NUM_PROCESSES);
         i < (stream_id + 1) * (SHM_SIZE / NUM_PROCESSES); i++) {
        shared_memory[i] = (unsigned char)(shared_memory[i] ^ 0xff);
    }
    char operation_detail[256];
    snprintf(operation_detail, sizeof(operation_detail),
             "Handled audio stream %d", stream_id);
    if (!log_event(operation_detail)) {
        fprintf(stderr, "Failed to log audio stream %d\n", stream_id);
    }
}

int main(void) {
    int shm_fd;
    char *shared_memory;
    pid_t pids[NUM_PROCESSES];

    if (!setup_shared_memory(&shm_fd, &shared_memory, SHM_SIZE, VOIP_SHM)) {
        return EXIT_FAILURE;
    }

    for (int i = 0; i < NUM_PROCESSES; i++) {
        pids[i] = fork();
```

```
        if (pids[i] == 0) {
            handle_audio_stream(i, shared_memory);
            exit(EXIT_SUCCESS);
        } else if (pids[i] < 0) {
            perror("fork");
            cleanup_shared_memory(shm_fd, shared_memory);
            return EXIT_FAILURE;
        }
    }

    for (int i = 0; i < NUM_PROCESSES; i++) {
        wait(NULL);
    }

    cleanup_shared_memory(shm_fd, shared_memory, SHM_SIZE, VOIP_SHM);

    return EXIT_SUCCESS;
}
```

Résultats et discussion

Cette implémentation montre comment gérer les flux audio en temps réel en utilisant des processus et une mémoire partagée. Chaque processus gère un flux audio distinct, ce qui permet d'isoler et de synchroniser efficacement les flux pour minimiser la latence.

Perspectives d'amélioration

Pour aller plus loin, nous pourrions :

- Intégrer des protocoles VoIP plus avancés qui offrent une meilleure gestion des flux audio et vidéo.
- Optimiser la gestion des ressources pour réduire la latence et améliorer la qualité audio.
- Ajouter des mécanismes de sécurité afin de protéger les flux de communication contre les interceptions et les attaques.

Le lecteur qui en savoir davantage sur les techniques de VoIP se tournera vers le livre de J. Davidson [9].

8.10 Étude de cas n°9 : blockchain et minage de cryptomonnaies

Contexte

La *blockchain* est une technologie de registre distribué qui permet de sécuriser les transactions de manière transparente et décentralisée. Chaque nouvelle transaction est enregistrée dans un bloc, qui est ensuite lié au bloc précédent par un processus cryptographique, formant ainsi une chaîne continue et sécurisée appelée *blockchain*.

Le minage de cryptomonnaies, quant à lui, est le processus par lequel de nouvelles unités de cryptomonnaie sont créées et les transactions sont vérifiées et ajoutées à la *blockchain*. Ce processus nécessite une puissance de calcul significative et repose sur la résolution de problèmes mathématiques complexes pour valider et sécuriser les blocs.

Objectif

Pour cette étude de cas, nous allons développer un système qui :

1. Implémente la structure simplifiée d'une *blockchain*.
2. Effectue le minage de blocs en utilisant des *threads* pour paralléliser les calculs.
3. Assure la sécurité et l'intégrité des données à travers des mécanismes de hachage.
4. Journalise les opérations pour le suivi et un audit ultérieur.

Structure de la *blockchain*

Notre *blockchain* est une structure de données qui enregistre des transactions de manière sécurisée, immuable et décentralisée. Chaque bloc dans notre *blockchain* contient les champs suivants :

1. Index : Le numéro du bloc dans la chaîne.
2. Horodatage : Le moment où le bloc a été créé.
3. Données de transaction : Les transactions enregistrées dans le bloc.
4. Hash du bloc précédent : La valeur de hachage du bloc précédent, qui relie le bloc actuel à la chaîne.
5. Hash : Une valeur de hachage unique pour le bloc, générée à partir des données contenues dans le bloc.

La figure 8.1 montre une représentation simplifiée de cette *blockchain* :

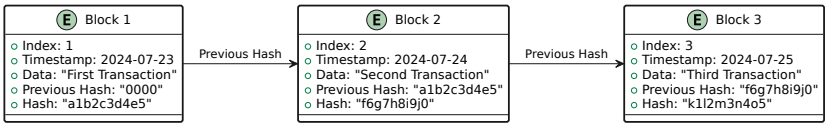


Fig. 8.1 : Exemple d'une *blockchain* avec trois blocs.

Méthodologie

Nous allons implémenter une *blockchain* simplifiée en utilisant les concepts suivants :

1. Chaque bloc contient les informations de base telles que l'index, l'horodatage, les données de transaction, le hash du bloc actuel, et le hash du bloc précédent.
2. Pour assurer la sécurité et l'intégrité des données, nous utilisons l'algorithme de hachage SHA-256 pour générer un hash unique pour chaque bloc.
3. Le processus de minage est effectué en parallèle en utilisant des *threads*, chaque *thread* étant responsable de la création et de la validation d'un bloc.

Implémentation

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <openssl/sha.h>
#include <time.h>
#include <stdbool.h>

#define MAX_BLOCKS  100
```

```c
#define NUM_THREADS 4

/**
 * @brief A struct representing a blockchain block.
 */
typedef struct block {
    int index;
    time_t timestamp;
    char data[256];
    char previous_hash[SHA256_DIGEST_LENGTH * 2 + 1];
    char hash[SHA256_DIGEST_LENGTH * 2 + 1];
} block_t;

block_t blockchain[MAX_BLOCKS];
int block_count = 0;
pthread_mutex_t blockchain_mutex;

bool log_event(const char *event_desc);

/**
 * @brief Generates SHA-256 hash for the input string.
 *
 * This function generates a SHA-256 hash from the given input string
 * and stores the result as a hexadecimal string in the output buffer.
 *
 * @param str The input string to be hashed.
 * @param output_buffer A buffer to store the resulting SHA-256 hash
 *                      as a hexadecimal string.
 */
void sha256(const char *str, char output_buffer[65]) {
    /* A buffer to store the binary (raw) hash output (32 bytes, 256 bits). */
    unsigned char hash[SHA256_DIGEST_LENGTH];
    /* A SHA256 context structure to maintain the state of the hash algorithm during computation. */
    SHA256_CTX sha256;

    /* Initialize the SHA256 context, preparing it for a new hash computation. */
    SHA256_Init(&sha256);
    /*
     * Update the context with the input string data. This step can be repeated
     * if the data is processed in chunks.
     */
    SHA256_Update(&sha256, str, strlen(str));
    /* Finalize the hash computation, producing the final binary hash and storing it in 'hash'. */
    SHA256_Final(hash, &sha256);

    /*
     * Convert the binary hash into a readable hexadecimal string.
     * Each byte (8 bits) is converted into two hexadecimal characters.
     */
    for (int i = 0; i < SHA256_DIGEST_LENGTH; i++) {
        sprintf(output_buffer + (i * 2), "%02x", hash[i]);
    }
    output_buffer[64] = 0; /* Add a NULL terminator */
```

```
}

/**
 * @brief Adds a new block to the blockchain.
 *
 * @param new_block The new block to be added.
 */

void add_block(block_t new_block) {
    pthread_mutex_lock(&blockchain_mutex);
    blockchain[block_count++] = new_block;
    pthread_mutex_unlock(&blockchain_mutex);
}

/**
 * @brief Mines a new block and adds it to the blockchain.
 *
 * This function creates a new block, populates its fields with current
 * information and the previous block's hash, calculates its hash using
 * SHA-256, logs the operation, and then adds the block to the blockchain.
 * It does all this in a separate thread to keep the main program running
 * smoothly.
 *
 * @param arg Not used in this implementation.
 */
void *mine_block(void *arg) {
    block_t new_block;
    char input[512];
    char log_entry[256];

    new_block.index = block_count;
    new_block.timestamp = time(NULL);

    strcpy(new_block.previous_hash, blockchain[block_count - 1].hash);
    snprintf(new_block.data, sizeof(new_block.data), "Block %d data",
             new_block.index);

    snprintf(input, sizeof(input), "%d%ld%s%s", new_block.index,
             new_block.timestamp, new_block.previous_hash, new_block.data);
    sha256(input, new_block.hash);

    snprintf(log_entry, sizeof(log_entry), "Mined block %d with hash %s",
             new_block.index, new_block.hash);
    if (!log_event(log_entry)) {
        fprintf(stderr, "Failed to log mined block %d\n", new_block.index);
    }

    add_block(new_block);
    pthread_exit(NULL);
}

/**
 * @brief Creates a specified number of threads to mine new blocks.
```

```
 *
 * @param threads An array to store the IDs of the created threads.
 * @param num_threads The number of threads to create.
 * @return true on success, false on failure.
 */
bool create_threads(pthread_t *threads, int num_threads) {
    for (int i = 0; i < num_threads; i++) {
        int rc = pthread_create(&threads[i], NULL, mine_block, NULL);
        if (rc) {
            fprintf(stderr,
                    "ERROR; return code from pthread_create() is %d\n", rc);
            return false;
        }
    }
    return true;
}

/**
 * @brief Joins a specified number of threads that were created to mine new
 *        blocks.
 *
 * @param threads An array containing the IDs of the threads to be joined.
 * @param num_threads The number of threads to join.
 * @return true on success, false on failure.
 */
bool join_threads(pthread_t *threads, int num_threads) {
    for (int i = 0; i < num_threads; i++) {
        int rc = pthread_join(threads[i], NULL);
        if (rc) {
            fprintf(stderr,
                    "ERROR; return code from pthread_join() is %d\n", rc);
            return false;
        }
    }
    return true;
}

int main(void) {
    pthread_t threads[NUM_THREADS];
    block_t genesis_block;
    char genesis_input[512];

    pthread_mutex_init(&blockchain_mutex, NULL);

    /* Create the genesis block */
    genesis_block.index = 0;
    genesis_block.timestamp = time(NULL);
    strcpy(genesis_block.previous_hash, "0");
    snprintf(genesis_block.data, sizeof(genesis_block.data), "Genesis Block");
    snprintf(genesis_input, sizeof(genesis_input), "%d%ld%s%s",
             genesis_block.index, genesis_block.timestamp,
             genesis_block.previous_hash, genesis_block.data);
    sha256(genesis_input, genesis_block.hash);
```

```
    blockchain[block_count++] = genesis_block;

    if (!create_threads(threads, NUM_THREADS)) {
        pthread_mutex_destroy(&blockchain_mutex);
        return EXIT_FAILURE;
    }

    if (!join_threads(threads, NUM_THREADS)) {
        pthread_mutex_destroy(&blockchain_mutex);
        return EXIT_FAILURE;
    }

    pthread_mutex_destroy(&blockchain_mutex);

    for (int i = 0; i < block_count; i++) {
        printf("Block %d: %s\n", blockchain[i].index, blockchain[i].hash);
    }

    return EXIT_SUCCESS;
}
```

Résultats et discussion

Cette implémentation montre comment créer une *blockchain* simplifiée et effectuer le minage de blocs en utilisant des *threads*. Chaque *thread* mine un bloc en calculant son hachage et en ajoutant le bloc à la *blockchain*. Le mutex assure la synchronisation des accès aux données partagées.

Perspectives d'amélioration

Pour aller plus loin, nous pourrions :

- Optimiser l'algorithme de minage afin d'en améliorer les performances.
- Permettre des transactions plus complexes et gérer leur mise en place dans un réseau réellement distribué.
- Intégrer des mécanismes de consensus distribués pour sécuriser davantage la *blockchain*.

Le lecteur intéressé par la technologie *blockchain* pourra consulter le tutoriel de GeeksForGeeks.org [12].

8.11 Étude de cas n°10 : analyse de séquences génomiques en bioinformatique

Contexte

La bioinformatique est un domaine qui combine la biologie et l'informatique afin d'analyser et d'interpréter des données biologiques, notamment des séquences génomiques. L'analyse de séquences d'ADN est importante pour comprendre les fonctions biologiques, identifier des gènes, et étudier les variations génétiques entre individus ou espèces.

Objectif

Pour cette étude de cas, nous devons développer un système qui :

1. Aligne deux séquences d'ADN pour identifier les régions similaires.
2. Trouve des motifs spécifiques dans une séquence génomique.

Méthodologie

Afin de tirer parti des capacités multicoeurs, nous utilisons des *threads* pour effectuer les tâches d'analyse en parallèle. L'algorithme de Needleman-Wunsch est utilisé pour l'alignement des séquences [27]. Le lecteur curieux pourra consulter l'ouvrage « Algorithmique du texte » de Maxime Crochemore qui propose les pseudo-codes d'un grand nombre d'algorithmes d'alignement, notamment celui de Needleman-Wunsch [8 -- chap. 7].

Algorithme de Needleman-Wunsch

La complexité en temps pour la comparaison de deux séquences de longueur m et n est $\mathcal{O}(m \cdot n)$. L'exploration de chaque position de chaque séquence pour la détermination éventuelle d'une insertion augmente d'un facteur $m + n$ le temps de calcul. La programmation dynamique permet de limiter cette augmentation pour conserver une complexité en temps en $\mathcal{O}(m \cdot n)$. Elle est basée sur le fait que tous les événements sont possibles et calculables, mais que la plupart sont rejetés en considérant certains critères. Needleman et Wunsch ont introduit les premiers ce type d'approche pour un problème biologique et leur algorithme reste une référence dans le domaine.

L'algorithme remplit une matrice de score S de taille $(m + 1) \times (n + 1)$ en réalisant les étapes suivantes :

1. **Initialisation** : remplissage de la première ligne et de la première colonne de la matrice de score (complexité en temps : $\mathcal{O}(m + n)$).
2. **Remplissage de la matrice** : chaque cellule $S(i, j)$ de la matrice est remplie en calculant le score optimal basé sur les cellules précédentes (diagonale, haut, gauche). La complexité est en $\mathcal{O}(m \cdot n)$.
3. **Calcul du score optimal et parcours arrière** : une fois la matrice remplie, le score optimal et l'alignement des séquences sont déterminés en parcourant la matrice de la dernière cellule à la première (complexité : $\mathcal{O}(m + n)$).

La complexité dominante est celle du remplissage de la matrice, soit $\mathcal{O}(m \cdot n)$.

Pour deux séquences $A = a_1, a_2, \ldots, a_m$ et $B = b_1, b_2, \ldots, b_n$, la matrice de score S est définie comme suit :

1. Initialisation :

   $S(i, 0) = i \cdot d \quad$ pour $i = 0, 1, \ldots, m$

   $S(0, j) = j \cdot d \quad$ pour $j = 0, 1, \ldots, n$

2. Récurrence :

$$S(i, j) = \max \begin{cases} S(i-1, j-1) + s(a_i, b_j) \\ S(i-1, j) + d \\ S(i, j-1) + d \end{cases} \qquad \text{pour } i = 1, 2, \ldots, m \text{ et } j = 1, 2, \ldots, n$$

   où $s(a_i, b_j)$ est le score de substitution (match/mismatch) et d est le score de gap.

3. On détermine enfin l'alignement optimal en suivant les scores maximaux de la dernière cellule à la première.

Implémentation

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

#define MATCH        1
#define MISMATCH    -1
#define GAP         -2
#define NUM_THREADS  4

typedef struct {
   const char *seq1;
   const char *seq2;
   char *aligned_seq1;
   char *aligned_seq2;
   int start;
   int end;
} thread_data_t;

/**
 * @brief Finds the maximum of three integers.
 *
 * This function is used to find the maximum value among three integers,
 * which is a critical operation in the Needleman-Wunsch algorithm.
 *
 * @param a First integer.
 * @param b Second integer.
 * @param c Third integer.
 * @return The maximum value among the three integers.
 */
int max(int a, int b, int c) {
   if (a >= b && a >= c) return a;
   if (b >= a && b >= c) return b;
   return c;
}

/**
 * @brief Implements the Needleman-Wunsch algorithm for sequence alignment.
 *
 * This function computes the optimal alignment of two sequences using
 * dynamic programming, filling a score matrix and then backtracking
 * to produce the aligned sequences.
 *
 * @param seq1 The first sequence.
 * @param seq2 The second sequence.
 * @param aligned_seq1 A buffer to store the aligned version of seq1.
 * @param aligned_seq2 A buffer to store the aligned version of seq2.
 */
void needleman_wunsch(const char *seq1, const char *seq2,
                      char *aligned_seq1, char *aligned_seq2) {
   int len1 = strlen(seq1);
```

```
int len2 = strlen(seq2);
int i, j;
int **score = malloc((len1 + 1) * sizeof(int *));

for (i = 0; i <= len1; i++) {
   score[i] = malloc((len2 + 1) * sizeof(int));
}
for (i = 0; i <= len1; i++) score[i][0] = i * GAP;
for (j = 0; j <= len2; j++) score[0][j] = j * GAP;
for (i = 1; i <= len1; i++) {
   for (j = 1; j <= len2; j++) {
      int match = score[i-1][j-1] + (seq1[i-1] == seq2[j-1]
                  ? MATCH : MISMATCH);
      int delete = score[i-1][j] + GAP;
      int insert = score[i][j-1] + GAP;
      score[i][j] = max(match, delete, insert);
   }
}
int align_len = 0;
i = len1;
j = len2;
while (i > 0 && j > 0) {
   if (score[i][j] == score[i-1][j-1] + (seq1[i-1] == seq2[j-1]
      ? MATCH : MISMATCH)) {
      aligned_seq1[align_len] = seq1[i-1];
      aligned_seq2[align_len] = seq2[j-1];
      i--;
      j--;
   } else if (score[i][j] == score[i-1][j] + GAP) {
      aligned_seq1[align_len] = seq1[i-1];
      aligned_seq2[align_len] = '-';
      i--;
   } else {
      aligned_seq1[align_len] = '-';
      aligned_seq2[align_len] = seq2[j-1];
      j--;
   }
   align_len++;
}
while (i > 0) {
   aligned_seq1[align_len] = seq1[i-1];
   aligned_seq2[align_len] = '-';
   i--;
   align_len++;
}
while (j > 0) {
   aligned_seq1[align_len] = '-';
   aligned_seq2[align_len] = seq2[j-1];
   j--;
   align_len++;
}
for (i = 0; i < align_len / 2; i++) {
   char temp = aligned_seq1[i];
```

```
      aligned_seq1[i] = aligned_seq1[align_len - i - 1];
      aligned_seq1[align_len - i - 1] = temp;
      temp = aligned_seq2[i];
      aligned_seq2[i] = aligned_seq2[align_len - i - 1];
      aligned_seq2[align_len - i - 1] = temp;
   }
   aligned_seq1[align_len] = '\0';
   aligned_seq2[align_len] = '\0';
   for (i = 0; i <= len1; i++) free(score[i]);
   free(score);
}

/**
 * @brief Thread function to align sequences.
 *
 * This function is executed by each thread to perform the Needleman-Wunsch
 * algorithm on a portion of the sequences.
 *
 * @param arg A pointer to the thread data containing the sequences and buffers.
 * @return None
 */
 void *align_sequences(void *arg) {
   thread_data_t *data = (thread_data_t *)arg;
   needleman_wunsch(data->seq1, data->seq2,
                    data->aligned_seq1, data->aligned_seq2);
   pthread_exit(NULL);
}

int main(void) {
   const char *seq1 = "GAGTCCCTGGTGACGTTCAG"; /* monkey ebola */
   const char *seq2 = "GCTGACCCTGAAGTGACGTCACTTC"; /* measles (rougeole) */
   char aligned_seq1[100];
   char aligned_seq2[100];
   pthread_t threads[NUM_THREADS];
   thread_data_t thread_data[NUM_THREADS];

   for (int i = 0; i < NUM_THREADS; i++) {
      thread_data[i].seq1 = seq1;
      thread_data[i].seq2 = seq2;
      thread_data[i].aligned_seq1 = aligned_seq1;
      thread_data[i].aligned_seq2 = aligned_seq2;
      pthread_create(&threads[i], NULL, align_sequences,
                     (void *)&thread_data[i]);
   }
   for (int i = 0; i < NUM_THREADS; i++) {
      pthread_join(threads[i], NULL);
   }
   printf("Aligned sequence 1: %s\n", aligned_seq1);
   printf("Aligned sequence 2: %s\n", aligned_seq2);

   return EXIT_SUCCESS;
}
```

8.11.1 Résultats et discussion

Cette implémentation montre comment utiliser des *threads* pour paralléliser efficacement l'alignement de séquences d'ADN en utilisant l'algorithme de Needleman-Wunsch. Chaque *thread* traite une portion de l'alignement, ce qui permet de réduire le temps de calcul total.

8.11.2 Perspectives d'amélioration

Pour aller plus loin, nous pourrions :

- Optimiser l'algorithme pour des séquences plus longues et complexes.
- Intégrer des méthodes de recherche de motifs.
- Construire un arbre phylogénétique pour étudier les relations évolutives entre différentes séquences.

Le lecteur qui souhaite approfondir les concepts et algorithmes de bioinformatique se reportera à l'ouvrage de G. Deléage [Deleage2015].

8.12 Étude de cas n°11 : gestion d'une grille de traitement

Contexte

Une grille de traitement (ou *cluster* de calcul distribué) est un ensemble d'ordinateurs interconnectés qui travaillent ensemble pour effectuer des tâches de calcul intensif. Chaque ordinateur, ou noeud, dans la grille, contribue avec une partie de ses ressources à résoudre des problèmes complexes qui seraient trop lourds pour une seule machine.

Objectif

Pour cette étude de cas, nous devons développer un système qui :

1. Coordonne les tâches entre plusieurs noeuds de calcul.
2. Utilise des processus et des *threads* afin de paralléliser les tâches.
3. Assure une communication efficace entre les noeuds de la grille.
4. Gère les ressources de manière optimale pour maximiser les performances.
5. Journalise les opérations pour le suivi et un audit ultérieur.

Méthodologie

Notre grille de traitement est constituée d'un noeud maître responsable de la distribution des tâches et de la collecte des résultats, ainsi que de plusieurs noeuds esclaves qui effectuent les calculs et retournent les résultats au noeud maître. Des *sockets* seront utilisés pour la communication entre le noeud maître et les noeuds esclaves.

Implémentation

Noeud maître

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

```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <pthread.h>
#include <stdbool.h>

#define PORT       54321
#define NUM_SLAVES     4
#define TASK_SIZE    256
#define BACKLOG       10  /* Max number of pending connections */

pthread_mutex_t log_mutex;

/**
 * @brief Logs an operation detail to a log file.
 *
 * @param operation_detail The detail of the operation to be logged.
 * @return true on success, false on failure.
 */
bool log_operation(const char *operation_detail) {
    pthread_mutex_lock(&log_mutex);
    FILE *logfile = fopen("/var/log/cluster_master.log", "a");
    if (logfile == NULL) {
        perror("Failed to open log file");
        pthread_mutex_unlock(&log_mutex);
        return false;
    }
    fprintf(logfile, "Operation: %s\n", operation_detail);
    fclose(logfile);
    pthread_mutex_unlock(&log_mutex);
    return true;
}

/**
 * @brief Handles communication with a slave process.
 *
 * @param socket_desc A pointer to an integer describing the socket.
 * @return None
 */
void *handle_slave(void *socket_desc) {
    int new_socket = *(int *)socket_desc;
    char buffer[TASK_SIZE] = {0};
    char *task_result = "Task completed";

    if (read(new_socket, buffer, TASK_SIZE) < 0) {
        perror("Failed to read from socket");
    } else {
        if (!log_operation(buffer)) {
            fprintf(stderr, "Failed to log operation from slave\n");
        }
        printf("Received task from slave: %s\n", buffer);
```

```
        if (send(new_socket, task_result, strlen(task_result), 0) < 0) {
            perror("Failed to send to socket");
        } else {
            printf("Task result sent to slave\n");
        }
    }

    close(new_socket);
    free(socket_desc);
    pthread_exit(NULL);
}

/**
 * @brief Sets up a server socket.
 *
 * This function sets up a server socket, creating and binding it to a
 * specific address and port. It also listens for incoming connections. If
 * any of these operations fail, an error message is printed and false is
 * returned.
 *
 * @param server_fd A pointer to an integer describing the server socket
 *                  file descriptor.
 * @param address A pointer to a sockaddr_in structure describing the
 *                address and port.
 * @return true on success, false on failure.
 */
bool setup_server(int *server_fd, struct sockaddr_in *address) {
    if ((*server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("Socket failed");
        return false;
    }

    address->sin_family = AF_INET;
    address->sin_addr.s_addr = INADDR_ANY;
    address->sin_port = htons(PORT);

    if (bind(*server_fd, (struct sockaddr *)address, sizeof(*address)) < 0) {
        perror("Bind failed");
        close(*server_fd);
        return false;
    }

    if (listen(*server_fd, BACKLOG) < 0) {
        perror("Listen");
        close(*server_fd);
        return false;
    }

    return true;
}

int main(void) {
```

```
    int server_fd
    int new_socket;
    int *new_sock;
    struct sockaddr_in address;
    int addrlen = sizeof(address);

    pthread_mutex_init(&log_mutex, NULL);

    if (!setup_server(&server_fd, &address)) {
        pthread_mutex_destroy(&log_mutex);
        return EXIT_FAILURE;
    }

    while (1) {
        if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
                                 (socklen_t *)&addrlen)) < 0) {
            perror("Accept");
            continue;
        }

        pthread_t slave_thread;
        new_sock = malloc(sizeof(int));
        if (new_sock == NULL) {
            perror("Failed to allocate memory");
            close(new_socket);
            continue;
        }
        *new_sock = new_socket;

        if (pthread_create(&slave_thread, NULL, handle_slave,
                           (void *)new_sock) != 0) {
            perror("Could not create thread");
            free(new_sock);
            close(new_socket);
            continue;
        }

        pthread_detach(slave_thread);
    }

    close(server_fd);
    pthread_mutex_destroy(&log_mutex);

    return EXIT_SUCCESS;
}
```

Noeud esclave

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

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>

#define PORT      54321
#define TASK_SIZE   256

/**
 * @brief Logs an operation detail to a log file.
 *
 * @param operation_detail The detail of the operation to be logged.
 */
void log_operation(const char *operation_detail) {
    FILE *logfile = fopen("/var/log/cluster_slave.log", "a");
    if (logfile == NULL) {
        perror("Failed to open log file");
        exit(EXIT_FAILURE);
    }
    fprintf(logfile, "Operation: %s\n", operation_detail);
    fclose(logfile);
}

/**
 * @brief Performs a dummy task to simulate computation.
 *
 * @param task_id The ID of the task to be performed.
 */
void perform_task(int task_id) {
    char task[TASK_SIZE];
    snprintf(task, sizeof(task), "Task %d: Performing computation", task_id);
    log_operation(task);
    printf("%s\n", task);
    sleep(5); /* Simulate computation */
}

int main(void) {
    int sock = 0, valread;
    struct sockaddr_in serv_addr;
    char *task = "Task data";
    char buffer[TASK_SIZE] = {0};

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Socket creation error");
        exit(EXIT_FAILURE);
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
        perror("Invalid address / Address not supported");
        exit(EXIT_FAILURE);
    }
```

```
    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        perror("Connection Failed");
        exit(EXIT_FAILURE);
    }

    send(sock, task, strlen(task), 0);
    printf("Task data sent\n");
    log_operation("Task data sent to master");

    valread = read(sock, buffer, TASK_SIZE);
    printf("Task result received: %s\n", buffer);
    log_operation(buffer);

    perform_task(1);

    close(sock);

    return EXIT_SUCCESS;
}
```

Résultats et discussion

Cette étude de cas montre comment gérer une grille de traitement en utilisant des processus et des *threads*, ainsi que des mécanismes de communication interprocessus. Le noeud maître coordonne les tâches et collecte les résultats, tandis que les noeuds esclaves effectuent les calculs et retournent les résultats au noeud maître.

Perspectives d'amélioration

Pour aller plus loin, nous pourrions :

- Optimiser les algorithmes de distribution des tâches pour une meilleure répartition des charges.
- Intégrer des systèmes de gestion de grilles plus avancés pour une meilleure gestion des ressources.
- Améliorer la tolérance aux pannes et la sécurité pour un fonctionnement plus robuste et sécurisé.

Le lecteur qui souhaiterait en connaître plus sur les systèmes distribués consultera le chapitre 8 de l'ouvrage d'A. Tanenbaum [35]. Pour un approfondissement sur les grilles de traitement, le lecteur se reportera au livre de F. Berman [3].

8.13 Étude de cas n°12 : surveillance des performances d'une grille de traitement

Contexte

Dans une grille de traitement, il est primordial de surveiller les performances afin de garantir une utilisation optimale des ressources et de détecter les éventuelles anomalies le plus rapidement possible. Cette étude de cas se concentre sur la mise en place d'un système de surveillance des performances des noeuds de calcul d'une grille de traitement.

Objectif

Développer un système qui :

1. Collecte les données de performance des noeuds de la grille.
2. Transmet les données au serveur central à l'aide de *sockets*.
3. Affiche les données en temps réel à l'aide d'une interface graphique.
4. Assure la modularité de l'application de manière à faciliter l'ajout de nouvelles fonctionnalités.

Méthodologie

Notre système de surveillance est constitué de :

1. Un serveur central qui recueille les données de performance des noeuds de la grille et les stocke.
2. Des noeuds de calcul qui surveillent leur performance et envoient ces informations au serveur central.
3. Une interface graphique en Python qui utilise *Tkinter* pour afficher les données en temps réel.

Implémentation

Noeud de calcul

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <pthread.h>
#include <stdbool.h>

#define PORT     54321
#define BUF_SIZE   256
#define HOST     "127.0.0.1"

/**
 * @brief Retrieves system usage data.
 *
 * @param buffer A pointer to a character array where the system usage data
 *               will be stored.
 * @param size The maximum number of characters that can be written to the
 *             buffer.
 * @return true if the operation was successful, false otherwise.
 */
bool get_system_usage(char *buffer, size_t size) {
    FILE *fp;
    char result[BUF_SIZE] = {0};

    /* Simulate system usage data collection */
    snprintf(result, sizeof(result), "CPU: %d%%, Memory: %d%%",
             rand() % 100, rand() % 100);
```

```
    strncpy(buffer, result, size - 1);
    buffer[size - 1] = '\0';
    return true;
}

/**
 * @brief Sends performance data to a server.
 *
 * @param arg A pointer to an integer representing the socket file descriptor.
 */
void *send_performance_data(void *arg) {
    int sock = *(int *)arg;
    char buffer[BUF_SIZE] = {0};

    while (1) {
        if (!get_system_usage(buffer, sizeof(buffer))) {
            fprintf(stderr, "Failed to get system usage\n");
            continue;
        }

        if (send(sock, buffer, strlen(buffer), 0) < 0) {
            perror("Failed to send data to server");
            break;
        }

        sleep(5); /* Send data every 5 seconds */
    }

    close(sock);
    pthread_exit(NULL);
}

int main(void) {
    int sock = 0;
    struct sockaddr_in serv_addr;

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Socket creation error");
        return EXIT_FAILURE;
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    if (inet_pton(AF_INET, HOST, &serv_addr.sin_addr) <= 0) {
        perror("Invalid address / Address not supported");
        return EXIT_FAILURE;
    }

    if (connect(sock, (struct sockaddr *)&serv_addr,
                sizeof(serv_addr)) < 0) {
        perror("Connection Failed");
```

```
        return EXIT_FAILURE;
    }

    pthread_t thread_id;
    if (pthread_create(&thread_id, NULL, send_performance_data,
                       (void *)&sock) != 0) {
        perror("Could not create thread");
        return EXIT_FAILURE;
    }

    pthread_join(thread_id, NULL);

    return EXIT_SUCCESS;
}
```

La version ci-dessous ne considère plus des données aléatoires, mais procède à une récupération des informations pour des noeuds qui tourneraient sur des systèmes Linux ou des systèmes Mac OS X.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <pthread.h>
#include <stdbool.h>
#include <errno.h>

#define PORT     54321
#define BUF_SIZE   256
#define HOST     "127.0.0.1"

#ifdef __linux__
/* Get CPU and memory usage on Linux */
bool get_system_usage(char *buffer, size_t size) {
    FILE *fp;
    char cpu_info[BUF_SIZE] = {0};
    char mem_info[BUF_SIZE] = {0};
    int cpu_usage = 0;
    int mem_total = 0, mem_free = 0, mem_available = 0;

    /* Read CPU usage from /proc/stat */
    fp = fopen("/proc/stat", "r");
    if (!fp) {
        perror("Failed to open /proc/stat");
        return false;
    }
    fgets(cpu_info, sizeof(cpu_info), fp);
    fclose(fp);

    /* Parse CPU usage */
    long user, nice, system, idle, iowait, irq, softirq, steal;
    sscanf(cpu_info, "cpu  %ld %ld %ld %ld %ld %ld %ld %ld",
```

```
           &user, &nice, &system, &idle, &iowait, &irq, &softirq, &steal);
    cpu_usage = 100 - (idle * 100
        / (user + nice + system + idle + iowait + irq + softirq + steal));

    /* Read memory usage from /proc/meminfo */
    fp = fopen("/proc/meminfo", "r");
    if (!fp) {
        perror("Failed to open /proc/meminfo");
        return false;
    }
    while (fgets(mem_info, sizeof(mem_info), fp)) {
        if (sscanf(mem_info, "MemTotal: %d kB", &mem_total) == 1) continue;
        if (sscanf(mem_info, "MemFree: %d kB", &mem_free) == 1) continue;
        if (sscanf(mem_info, "MemAvailable: %d kB", &mem_available) == 1) continue;
    }
    fclose(fp);

    /* Calculate memory usage */
    int mem_used = mem_total - mem_available;

    /* Format the output */
    snprintf(buffer, size, "CPU: %d%%, Memory: %d%%", cpu_usage, (mem_used * 100) / mem_total);
    return true;
}
#elif __APPLE__
#include <sys/sysctl.h>
#include <mach/mach.h>

/* Get CPU and memory usage on Mac OS X */
bool get_system_usage(char *buffer, size_t size) {
    /* Get CPU usage */
    host_cpu_load_info_data_t cpuinfo;
    mach_msg_type_number_t count = HOST_CPU_LOAD_INFO_COUNT;
    kern_return_t kr = host_statistics(mach_host_self(), HOST_CPU_LOAD_INFO, (host_info_t)&cpuinfo,
↪   &count);
    if (kr != KERN_SUCCESS) {
        perror("Failed to get CPU load info");
        return false;
    }

    long totalTicks = 0;
    for (int i = 0; i < CPU_STATE_MAX; i++) {
        totalTicks += cpuinfo.cpu_ticks[i];
    }
    long idleTicks = cpuinfo.cpu_ticks[CPU_STATE_IDLE];
    int cpu_usage = (totalTicks - idleTicks) * 100 / totalTicks;

    /* Get memory usage */
    int64_t physical_memory;
    size_t length = sizeof(physical_memory);
    if (sysctlbyname("hw.memsize", &physical_memory, &length, NULL, 0) != 0) {
        perror("Failed to get total memory size");
        return false;
```

```
    }

    mach_port_t host_port = mach_host_self();
    mach_msg_type_number_t host_size = sizeof(vm_statistics_data_t) / sizeof(integer_t);
    vm_size_t pagesize;
    vm_statistics_data_t vm_stat;

    if (host_page_size(host_port, &pagesize) != KERN_SUCCESS) {
        perror("Failed to get page size");
        return false;
    }

    if (host_statistics(host_port, HOST_VM_INFO, (host_info_t)&vm_stat, &host_size) != KERN_SUCCESS)
     ↪ {
        perror("Failed to get VM statistics");
        return false;
    }

    int64_t mem_free = vm_stat.free_count * pagesize;
    int64_t mem_used = physical_memory - mem_free;
    int mem_usage = (mem_used * 100) / physical_memory;

    /* Format the output */
    snprintf(buffer, size, "CPU: %d%%, Memory: %d%%", cpu_usage, mem_usage);
    return true;
}
#endif

void *send_performance_data(void *arg) {
    int sock = *(int *)arg;
    char buffer[BUF_SIZE] = {0};

    while (1) {
        if (!get_system_usage(buffer, sizeof(buffer))) {
            fprintf(stderr, "Failed to get system usage\n");
            continue;
        }

        if (send(sock, buffer, strlen(buffer), 0) < 0) {
            perror("Failed to send data to server");
            break;
        }

        sleep(5); /* Send data every 5 seconds */
    }

    close(sock);
    pthread_exit(NULL);
}

int main(void) {
    int sock = 0;
    struct sockaddr_in serv_addr;
```

```
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Socket creation error");
        return EXIT_FAILURE;
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    if (inet_pton(AF_INET, HOST, &serv_addr.sin_addr) <= 0) {
        perror("Invalid address / Address not supported");
        return EXIT_FAILURE;
    }

    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        perror("Connection Failed");
        return EXIT_FAILURE;
    }

    pthread_t thread_id;
    if (pthread_create(&thread_id, NULL, send_performance_data,
                       (void *)&sock) != 0) {
        perror("Could not create thread");
        return EXIT_FAILURE;
    }

    pthread_join(thread_id, NULL);

    return EXIT_SUCCESS;
}
```

Serveur central

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <pthread.h>

#define PORT     54321
#define BACKLOG     10
#define BUF_SIZE   256

pthread_mutex_t log_mutex;

/* Logs performance data to a specified log file. */
void log_performance_data(const char *data) {
    pthread_mutex_lock(&log_mutex);
    FILE *logfile = fopen("performance.log", "a");
```

```
    if (logfile == NULL) {
        perror("Failed to open log file");
        pthread_mutex_unlock(&log_mutex);
        return;
    }
    fprintf(logfile, "Data: %s\n", data);
    fclose(logfile);
    pthread_mutex_unlock(&log_mutex);
}

/* Handles incoming performance data from a client node. */
void *handle_client(void *socket_desc) {
    int new_socket = *(int *)socket_desc;
    char buffer[BUF_SIZE] = {0};

    while (1) {
        int valread = read(new_socket, buffer, BUF_SIZE);
        if (valread < 0) {
            perror("Failed to read from socket");
            break;
        }
        buffer[valread] = '\0';
        log_performance_data(buffer);
        printf("Received data: %s\n", buffer);
    }

    close(new_socket);
    free(socket_desc);
    pthread_exit(NULL);
}

/* Sets up the server socket. Returns true on success, false on failure. */
bool setup_server(int *server_fd, struct sockaddr_in *address) {
    if ((*server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("Socket failed");
        return false;
    }

    address->sin_family = AF_INET;
    address->sin_addr.s_addr = INADDR_ANY;
    address->sin_port = htons(PORT);

    if (bind(*server_fd, (struct sockaddr *)address, sizeof(*address)) < 0) {
        perror("Bind failed");
        close(*server_fd);
        return false;
    }

    if (listen(*server_fd, BACKLOG) < 0) {
        perror("Listen");
        close(*server_fd);
        return false;
    }
```

```
    return true;
}

int main(void) {
    int server_fd, new_socket, *new_sock;
    struct sockaddr_in address;
    int addrlen = sizeof(address);

    pthread_mutex_init(&log_mutex, NULL);

    if (!setup_server(&server_fd, &address)) {
        pthread_mutex_destroy(&log_mutex);
        return EXIT_FAILURE;
    }

    while (1) {
        if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
                                 (socklen_t *)&addrlen)) < 0) {
            perror("Accept");
            continue;
        }

        pthread_t client_thread;
        new_sock = malloc(sizeof(int));
        if (new_sock == NULL) {
            perror("Failed to allocate memory");
            close(new_socket);
            continue;
        }
        *new_sock = new_socket;

        if (pthread_create(&client_thread, NULL, handle_client,
                           (void *)new_sock) < 0) {
            perror("Could not create thread");
            free(new_sock);
            close(new_socket);
            continue;
        }

        pthread_detach(client_thread);
    }

    close(server_fd);
    pthread_mutex_destroy(&log_mutex);

    return EXIT_SUCCESS;
}
```

Interface graphique minimale

```
import tkinter as tk
from tkinter import scrolledtext
import socket
import threading

class PerformanceMonitor(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("Cluster Performance Monitor")
        self.geometry("600x400")

        self.log_text = scrolledtext.ScrolledText(self, wrap=tk.WORD, width=60, height=20)
        self.log_text.pack(pady=10)

        self.start_button = tk.Button(self, text="Start Monitoring", command=self.start_monitoring)
        self.start_button.pack(pady=5)

        self.stop_button = tk.Button(self, text="Stop Monitoring", command=self.stop_monitoring,
        ↪  state=tk.DISABLED)
        self.stop_button.pack(pady=5)

        self.running = False
        self.server_thread = None

    def start_monitoring(self):
        self.running = True
        self.server_thread = threading.Thread(target=self.run_server)
        self.server_thread.start()
        self.start_button.config(state=tk.DISABLED)
        self.stop_button.config(state=tk.NORMAL)

    def stop_monitoring(self):
        self.running = False
        if self.server_thread:
            self.server_thread.join()
        self.start_button.config(state=tk.NORMAL)
        self.stop_button.config(state=tk.DISABLED)

    def run_server(self):
        server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server_socket.bind(("0.0.0.0", 54321))
        server_socket.listen(5)

        while self.running:
            client_socket, addr = server_socket.accept()
            threading.Thread(target=self.handle_client, args=(client_socket,)).start()

        server_socket.close()

    def handle_client(self, client_socket):
        while self.running:
```

```
            data = client_socket.recv(1024).decode('utf-8')
            if not data:
                break
            self.log_text.insert(tk.END, data + '\n')
            self.log_text.see(tk.END)

        client_socket.close()

if __name__ == "__main__":
    app = PerformanceMonitor()
    app.mainloop()
```

Résultats et discussion

Cette étude de cas montre comment construire une application modulaire pour surveiller les performances d'une grille de traitement. Chaque noeud de calcul collecte ses propres données de performance et les envoie au serveur central. Les données sont affichées en temps réel à l'aide d'une interface graphique indépendante.

Perspectives d'amélioration

Pour aller plus loin, nous pourrions :

- Optimiser la collecte et la transmission des données pour réduire l'impact sur les performances des noeuds.
- Ajouter des alertes afin de notifier les administrateurs en cas d'anomalies détectées.
- Intégrer des fonctionnalités de visualisation avancées pour une analyse plus approfondie des performances.

8.14 Conclusion

Ce chapitre a mis en évidence l'application pratique d'un certain nombre de concepts de programmation système en langage C sous Unix à travers une série d'études de cas couvrant des domaines variés auxquels vous serez peut-être confrontés. Chaque étude de cas a illustré comment les techniques et les mécanismes vus dans les chapitres précédents peuvent être utilisés pour résoudre des problèmes réels et souvent complexes.

Éléments de réponse aux exercices

Chapitre sur les processus

Exercice n°2.1 : Implémentation de l'ordonnancement circulaire

```
#include <stdio.h>

typedef struct {
    int pid;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int waiting_time;
    int turnaround_time;
} process_t;

void calculate_times(process_t processes[], int n, int quantum) {
    int time = 0;
    int completed = 0;

    while (completed < n) {
        for (int i = 0; i < n; i++) {
            if (processes[i].remaining_time > 0) {
                if (processes[i].remaining_time <= quantum) {
                    time += processes[i].remaining_time;
                    processes[i].remaining_time = 0;
                    processes[i].turnaround_time = time - processes[i].arrival_time;
                    processes[i].waiting_time = processes[i].turnaround_time -
 ↪  processes[i].burst_time;
                    completed++;
                } else {
                    processes[i].remaining_time -= quantum;
                    time += quantum;
                }
            }
        }
    }
}

void print_processes(process_t processes[], int n) {
    printf("PID\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n",
               processes[i].pid, processes[i].arrival_time, processes[i].burst_time,
```

```
                processes[i].waiting_time, processes[i].turnaround_time);
    }
}

int main(void) {
    int n, quantum;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    process_t processes[n];
    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        printf("Enter arrival time and burst time for process %d: ", i + 1);
        scanf("%d %d", &processes[i].arrival_time, &processes[i].burst_time);
        processes[i].remaining_time = processes[i].burst_time;
    }

    printf("Enter the time quantum: ");
    scanf("%d", &quantum);

    calculate_times(processes, n, quantum);
    print_processes(processes, n);

    return EXIT_SUCCESS;
}
```

Exercice 2.2 : Gestion des processus zombies

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void) {
    pid_t pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid != 0) {
        printf("Parent process (PID %d) is waiting...\n", getpid());
        int status;
        pid_t child_pid = wait(&status);
        if (WIFEXITED(status)) {
            printf("Parent process (PID %d): Child (PID %d) has ended (code %d)\n",
                   getpid(), child_pid, WEXITSTATUS(status));
        }
    } else {
        printf("Child process (PID %d)\n", getpid());
        sleep(10); /* sleep for 10 seconds */
```

```
        printf("End of child\n");
    }
    return EXIT_SUCCESS;
}
```

Exercice 2.3 : Hiérarchie des processus

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

void create_children(int level, int max_level) {
    if (level > max_level) return;

    for (int i = 0; i < 2; i++) {
        pid_t pid = fork();
        if (pid == -1) {
            perror("fork");
            exit(EXIT_FAILURE);
        }

        if (pid == 0) { /* child process */
            printf("Process %d created by %d at level %d\n", getpid(), getppid(), level);
            create_children(level + 1, max_level);
            exit(EXIT_SUCCESS);
        }
    }

    /* Wait for all child processes to terminate */
    while (wait(NULL) > 0);
}

int main(void) {
    printf("Parent process %d at level 0\n", getpid());
    create_children(1, 3);

    return EXIT_SUCCESS;
}
```

Exercice 2.4 : Variables d'environnement

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

int main(void) {
    pid_t pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
```

```
    }

    if (pid == 0) { /* child process */
        printf("Child process %d, setting environment variable MY_HELLO\n", getpid());
        setenv("MY_HELLO", "Bonjour les Ensicaenais !", 1);
        execl("/usr/bin/env", "env", NULL);
        perror("execl");
        exit(EXIT_FAILURE);
    } else { /* parent process */
        wait(NULL);
        printf("Parent process %d\n", getpid());
    }

    return EXIT_SUCCESS;
}
```

Chapitre sur les threads

Exercice 3.1 : Création de threads simples

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

void *print_message(void *thread_id) {
    long tid = (long)thread_id;
    printf("Thread No %ld is beginning...\n", tid);
    /* Simulate some work... */
    sleep(1);
    printf("Thread No %ld is terminated.\n", tid);
    pthread_exit(NULL);
}

int main(void) {
    pthread_t threads[2];

    for (long t = 0; t < 2; t++) {
        printf("[main thread]: Going to create thread No %ld\n", t);
        int rc = pthread_create(&threads[t], NULL, print_message, (void *)t);
        if (rc) {
            printf("Error when trying to create thread No %ld; error code: %d\n", t, rc);
            exit(EXIT_FAILURE);
        }
    }

    /* Wait for all threads to terminate */
    for (long t = 0; t < 2; t++) {
        pthread_join(threads[t], NULL);
    }

    printf("[main thread]: all threads are terminated.\n");
```

```
    return EXIT_SUCCESS;
}
```

Exercice 3.2 : Création et identification des threads

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

void *print_thread_id(void *thread_id) {
    printf("Thread No %ld is running\n", pthread_self());
    pthread_exit(NULL);
}

int main(void) {
    pthread_t threads[3];

    for (long t = 0; t < 3; t++) {
        int rc = pthread_create(&threads[t], NULL, print_thread_id, (void *)t);
        if (rc) {
            printf("Error when trying to create thread No %ld; error code: %d\n", t, rc);
            exit(EXIT_FAILURE);
        }
    }

    /* Wait for all threads to terminate */
    for (long t = 0; t < 3; t++) {
        pthread_join(threads[t], NULL);
    }

    printf("[main thread]: all threads are terminated.\n");

    return EXIT_SUCCESS;
}
```

Exercice 3.3 : Utilisation des threads détachés

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

void *print_message(void *thread_id) {
    printf("Detached thread is beginning...\n");
    /* Simulate some work... */
    sleep(1);
    printf("Detached thread has ended.\n");
    pthread_exit(NULL);
}

int main(void) {
```

```
    pthread_t thread;
    pthread_attr_t attr;

    /* Initialize the thread's attributes */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

    printf("[main thread]: Going to create a detached thread.\n");
    int rc = pthread_create(&thread, &attr, print_message, NULL);
    if (rc) {
        printf("Error creating a detached thread: %d\n", rc);
        exit(EXIT_FAILURE);
    }

    /* Destroy previous attributes */
    pthread_attr_destroy(&attr);

    /* Wait for a moment to be sure that the detached thread is running */
    sleep(2);

    printf("[main thread]: The program terminates.\n");

    return EXIT_SUCCESS;
}
```

Chapitre sur les fichiers

Exercice 4.1 : Taille maximale d'un fichier

1 bloc de 512 peut contenir : 512/4 = 128 pointeurs

8.14.1 Taille max. de fichier

$T_{max} = 10 \times 512 + 128 \times 512 + (128 \times 128) \times 512$

$T_{max} = 8\,459\,264$ octets

$T_{max} = 8\,459\,264/2^{10}$ Kio

$T_{max} = 8\,261$ Kio

Réponse **d**

8.14.2 Exercice 4.2 : Des blocs et des blocs {—}

Blocs nécessaires

2 128 Kio = 2 179 072 octets, soient 4 256 blocs de 512 octets

L'ensemble est décomposable en : 10 blocs de 512 + 128 blocs de 512 + 128 × 33 blocs de 512

8.14.2.1 Espace perdu {—}

$2\,233\,344 - 2\,179\,072 = 53$ Kio (54 272 octets)

Soit environ 2.5 % de la taille du fichier

Précision sur la décomposition

- $4\,256 - 10 = 4\,246$ blocs hors pointeurs directs
- $4\,246 - 128 = 4\,118$ blocs hors pointeurs directs et indirects simples

On cherche donc le nombre de pointeurs indirects doubles x tel que : $x \geq 4\,118/128$

Soit $x \geq 32.17$, donc $x = 33$

Exercice 4.4 : Ping et Pong 1

1. `fd2 = 3`
2. `fd2 = 3`
3. `fd2 = -1`

Exercice 4.4 : Ping et Pong 2

1. `fd2 = 6`
2. `fd2 = 5`
3. `fd2 = -1`

Exercice 4.5 : Manipulation des flots d'entrées/sorties

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

int main(void) {
    int fd_out, fd_err;

    /* Open files used for redirection */
    fd_out = open("stdout.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd_out == -1) {
        perror("Error when trying to open stdout.txt");
        exit(EXIT_FAILURE);
    }

    fd_err = open("stderr.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd_err == -1) {
        perror("Error when trying to open stderr.txt");
        close(fd_out);
        exit(EXIT_FAILURE);
    }
```

```
    /* Activate redirections */
    dup2(fd_out, STDOUT_FILENO);
    dup2(fd_err, STDERR_FILENO);

    /* Write some messages to redirected streams */
    printf("This is a message for the standard output.\n");
    fprintf(stderr, "This is a message for the standard error output.\n");

    /* Close files */
    close(fd_out);
    close(fd_err);

    return EXIT_SUCCESS;
}
```

Exercice 4.6 : Création d'un lien symbolique

```
#!/bin/bash

# Create a test file
echo "This is a file to test symbolic links..." > original.txt

# Then add a symbolic link
ln -s original.txt link_to_original.txt

# Verify the symbolic link
if [ -L "link_to_original.txt" ] && [ -e "link_to_original.txt" ]; then
    echo "The symbolic link as been created and it works..."
else
    echo "Error to create a symbolic link."
fi
```

Chapitre sur la mémoire virtuelle

Exercice 5.1 : Cadres et pages

1 Kio = 2^{10} octets

2^{18} correspond en Kio à $2^{18}/2^{10}2^{(18-10)} = 2^8 = 256$ Kio

Pour obtenir des cadres de 4 Kio : 256/4 → 64 cadres

Exercice 5.2 : Ordonnancement des pages

- Pour 3 cadres : 9 défauts de page.
- Pour 4 cadres : 10 défauts de page.

Vérifie l'*anomalie de Bélády*.

Chapitre sur la communication interprocessus

Exercice 6.1

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

void handle_signal(int sig) {
    if (sig == SIGUSR1) {
        printf("Received SIGUSR1\n");
    } else if (sig == SIGUSR2) {
        printf("Received SIGUSR2\n");
    }
}

int main(void) {
    struct sigaction sa;
    pid_t pid;

    // Clear the sigaction structure
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0; // No flags
    sa.sa_handler = handle_signal;

    // Set up sigaction for SIGUSR1 and SIGUSR2
    if (sigaction(SIGUSR1, &sa, NULL) == -1) {
        perror("sigaction");
        return EXIT_FAILURE;
    }
    if (sigaction(SIGUSR2, &sa, NULL) == -1) {
        perror("sigaction");
        return EXIT_FAILURE;
    }

    pid = fork();
    if (pid < 0) {
        perror("Fork failed");
        return EXIT_FAILURE;
    }

    if (pid == 0) { // Child process
        // Keep the child running to listen to signals
        while (1) {
            sleep(1);
        }
    } else { // Parent process
        sleep(2); // Ensure child is ready
        kill(pid, SIGUSR1);
        sleep(1);
        kill(pid, SIGUSR2);
```

```
        sleep(1);
        kill(pid, SIGKILL); // Terminate child
    }

    return 0;
}
```

Exercice 6.2

```
int main(void) {
    int i;
    /** Begin of A **/
    int j;
    int fd[N][2];
    for (i = 0 ; i < N; i++) {
        pipe(fd[i]);
    }
    /** End of A **/

    for (i = 0; i < N; i++) {
        if (fork() == 0) {
            /** Begin of B **/
            dup2(fd[i][1], 1);
            dup2(fd[(N+i-1)%N][0], 0);
            for (j = 0; j < N; j++) {
                close(fd[j][0]);
                close(fd[j][1]);
            }
            /** End of B **/
            proc(i);
            exit(EXIT_SUCCESS);
        }
    }
    return EXIT_SUCCESS;
}
```

Exercice 6.3

Client

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 54321
```

```
int main(int argc, char *argv[]) {
    int sockfd;
    struct sockaddr_in serv_addr;
    struct hostent *server;

    char buffer[256] = "Hello server";
    char response[256];

    if (argc < 2) {
       fprintf(stderr, "Usage: %s <hostname>\n", argv[0]);
       exit(EXIT_FAILURE);
    }
    server = gethostbyname(argv[1]);
    if (server == NULL) {
        fprintf(stderr, "Error, no such host\n");
        exit(EXIT_FAILURE);
    }

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        perror("Error opening socket");
        exit(EXIT_FAILURE);
    }

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr, (char *)&serv_addr.sin_addr.s_addr, server->h_length);
    serv_addr.sin_port = htons(SERVER_PORT);

    if (connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        perror("Error connecting");
        exit(EXIT_FAILURE);
    }

    write(sockfd, buffer, strlen(buffer));
    read(sockfd, response, 255);
    printf("%s\n", response);
    close(sockfd);

    return EXIT_SUCCESS;
}
```

Serveur

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```c
#define SERVER_PORT 54321
#define BUF_SIZE 256

void error(const char *msg) {
    perror(msg);
    exit(EXIT_FAILURE);
}

int main(void) {
    int sockfd, newsockfd, portno;
    socklen_t clilen;
    char buffer[BUF_SIZE];
    struct sockaddr_in serv_addr, cli_addr;
    int n;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
       error("Error opening socket");
    }

    bzero((char *) &serv_addr, sizeof(serv_addr));
    portno = SERVER_PORT;
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(portno);

    if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
              error("Error on binding");

    listen(sockfd, 5);
    clilen = sizeof(cli_addr);
    newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
    if (newsockfd < 0) {
          error("Error on accept");
    }

    bzero(buffer, BUF_SIZE);
    n = read(newsockfd, buffer, BUF_SIZE-1);
    if (n < 0) {
        error("Error reading from socket");
    }
    printf("Here is the message: %s\n", buffer);

    n = write(newsockfd, "Welcome dear client", 20);
    if (n < 0) {
        error("Error writing to socket");
    }

    close(newsockfd);
    close(sockfd);

    return EXIT_SUCCESS;
}
```

Exercice 6.4

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>

#define SHM_NAME "/my_shm"
#define BUF_SIZE 256

int main(void) {
    int shm_fd;
    void *ptr;
    const char *messages[2] = {"Bonjour du parent !", "Bonjour de l'enfant !"};
    pid_t pid;

    // Création ou ouverture du segment de mémoire partagée
    shm_fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);
    if (shm_fd == -1) {
        perror("shm_open");
        exit(EXIT_FAILURE);
    }

    // Dimensionnement du segment
    if (ftruncate(shm_fd, BUF_SIZE) == -1) {
        perror("ftruncate");
        exit(EXIT_FAILURE);
    }

    // Mapping du segment de mémoire partagée
    ptr = mmap(0, BUF_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
    if (ptr == MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }

    pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid > 0) {  // Processus parent
        sprintf(ptr, "%s", messages[0]);  // Écriture du message du parent
        wait(NULL);  // Attente de la fin du processus enfant
        printf("Parent lit : %s\n", (char *)ptr);  // Lecture de la réponse de l'enfant
    } else {  // Processus enfant
        printf("Enfant lit : %s\n", (char *)ptr);  // Lecture du message du parent
        sprintf(ptr, "%s", messages[1]);  // Écriture du message de l'enfant
```

```
    }

    // Nettoyage
    if (pid > 0) {
        shm_unlink(SHM_NAME);
    }
    return EXIT_SUCCESS;
}
```

Chapitre sur la synchronisation

Exercice : Une synchronisation râtée

Analyse de la boucle de comptage (version thread)

La boucle de comptage peut être finement analysée afin de comprendre pourquoi une synchronisation est nécessaire.

```
for (i = 0; i < NB_ITERS; i++) {
    cnt++;
}
```

Code assembleur correspondant

Voici le code assembleur correspondant :

```
.L9:
    movl -4(%ebp),%eax  #
    cmpl $99999999,%eax # Head      (H)
    jle .L12            #
    jmp .L10            #
.L12:
    movl cnt,%eax       # Load      (L)
    leal 1(%eax),%edx   # Update    (U)
    movl %edx,cnt       # Store     (S)
.L11:
    movl -4(%ebp),%eax  #
    leal 1(%eax),%edx   # Terminate (T)
    movl %edx,-4(%ebp)  #
    jmp .L9             #
.L10:
```

Séquences possibles en sortie

Les problèmes surviennent parce que les opérations de chargement (`L`), mise à jour (`U`) et sauvegarde (`S`) ne sont pas atomiques. Lorsque deux *threads* exécutent ces opérations de manière concurrente, l'entrelacement des instructions peut produire des résultats incorrects.

Séquence qui fonctionne Dans cette séquence, les opérations des deux *threads* sont correctement intercalées, ce qui permet d'obtenir le résultat correct :

`i` (*thread*)	instr	`%eax1`	`%eax2`	`cnt`
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
1	S_1	1	-	1
2	H_2	-	-	1
2	L_2	-	1	1
2	U_2	-	2	1
2	S_2	-	2	2
2	T_2	-	2	2
1	T_1	1	-	2

Séquence qui génère une erreur Dans cette séquence par contre, les opérations `L` et `S` des deux *threads* s'entrelacent de manière incorrecte, entraînant une mise à jour incorrecte du compteur partagé :

`i` (*thread*)	instr	`%eax1`	`%eax2`	`cnt`
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
2	H_2	-	-	0
2	L_2	-	0	0
1	S_1	1	-	1
1	T_1	1	-	1
2	U_2	-	1	1
2	S_2	-	1	1
2	T_2	-	1	1

Dans ce cas, la valeur de `cnt` est mise à jour de manière incorrecte parce que les opérations `L`, `U` et `S` ne sont pas atomiques. L'instruction `cnt++` semble simple mais nécessite plusieurs étapes qui peuvent être interrompues par d'autres *threads*, ce qui provoque ces résultats inattendus.

Cet exemple démontre clairement la nécessité de mécanismes de synchronisation pour éviter les accès concurrents indésirables et garantir l'intégrité des données.

Annexes

Annexe A : appeler `return` ou `exit()` ?

Dans un programme écrit en C, il est important de comprendre la distinction entre l'utilisation de l'instruction `return` et l'appel à la fonction `exit()`.

Utilisation de `return`

L'instruction `return` est utilisée pour terminer l'exécution d'une fonction et retourner une valeur à la fonction appelante. C'est l'instruction standard pour sortir d'une fonction, y compris la fonction principale `main()`. Lorsque `return` est appelé dans `main()`, il permet de retourner un code de sortie au système d'exploitation.

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

int main(void) {
    printf("Hello les Ensicaenais !\n");
    return EXIT_SUCCESS;  /* Terminate the process and return 0 */
}
```

Utilisation de `exit()`

La fonction `exit()` termine immédiatement le processus en cours. Contrairement à `return`, qui termine une fonction, `exit()` réalise une terminaison « propre » du processus. En effet, lors de l'appel de `exit()`, les actions suivantes sont mises en oeuvre :

1. Toutes les fonctions enregistrées à l'aide de `atexit()` sont exécutées dans l'ordre inverse de leur enregistrement.
2. Tous les tampons associés aux flux d'entrée-sortie ouverts sont « vidangés », ce qui garantit que que toutes les données en attente d'écriture sont bien écrites.
3. Tous les flux ouverts sont fermés de manière ordonnée.
4. Tous les fichiers temporaires créés avec la fonction `tmpfile()` sont supprimés.

Voici un exemple d'utilisation de `exit()` :

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

void cleanup(void) {
    printf("Performing some cleanup tasks...\n");
}

int main(void) {
```

```
    if (atexit(cleanup) != 0) {
        fprintf(stderr, "Cannot set exit function\n");
        return EXIT_FAILURE;
    }

    printf("Hello les Ensicaenais !\n");
    exit(EXIT_SUCCESS);  /* Terminate the process by executing cleanup first */
}
```

En conclusion, bien que `return` et `exit()` puissent tous deux être utilisés pour terminer un programme, ils diffèrent par leur portée et les actions qu'ils entreprennent. L'instruction `return` est appropriée pour terminer des fonctions et retourner des valeurs à l'appelant, tandis que `exit()` est destiné à une terminaison plus complète et contrôlée du processus en cours.

Annexe B : Programmation système sous Ms-Windows

Exemple de l'architecture de Ms-Windows 2000

La figure ci-dessous illustre l'architecture de Ms-Windows 2000 et détaille comment les différents composants du système interagissent.

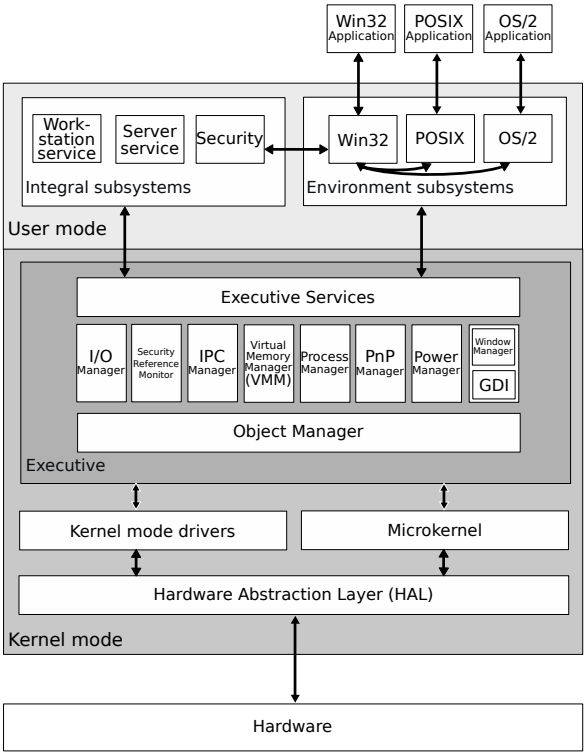


Fig. 2 : Architecture de Ms-Windows 2000 (source : Wikipedia -- auteur Elisardojm sous licence CC BY-SA 3.0).

Pour plus de détails sur les API Ms-Windows, visitez le site officiel de Microsoft.

Ms-Windows et les processus

Sous Ms-Windows, la gestion des processus diffère significativement de celle d'Unix, particulièrement en ce qui concerne la création, la gestion des processus et leurs communications.

La table suivante présente les appels système équivalents concernant la gestion des processus.

UNIX	WIN32/64	Description
`fork`	`CreateProcess`	Crée un nouveau processus
`execve`		Exécution d'un programme
`wait`	`WaitForMultipleObject`	Se met en attente de la fin d'un processus
`waitpid`	`WaitForSingleObject`	Se met en attente de la fin d'un processus
`exit`	`ExitProcess`	Termine l'exécution du processus
`getpid`	`GetCurrentProcess`	Crée un fichier ou ouvre un fichier existant

UNIX	WIN32/64	Description
`sleep`	`Sleep`	Bloque le processus pendant un nombre de secondes/millisecondes

Pour récupérer le PID du processus parent, il est nécessaire d'accéder au champ `th32ParentProcessID` de la structure `PROCESSENTRY32`.

Sous Ms-Windows, il est nécessaire de spécifier explicitement les ressources dont hériteront les processus créés, en particulier les tubes.

L'appel système `CreateProcess()` crée un processus enfant afin d'exécuter un programme. L'équivalent Unix correspond à la combinaison `fork()` + `exec()`. Son prototype est le suivant :

```
#include <windows.h>

BOOL WINAPI CreateProcess(
    _In_opt_    LPCTSTR               lpApplicationName,
    _Inout_opt_ LPTSTR                lpCommandLine,
    _In_opt_    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    _In_opt_    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    _In_        BOOL                  bInheritHandles,
    _In_        DWORD                 dwCreationFlags,
    _In_opt_    LPVOID                lpEnvironment,
    _In_opt_    LPCTSTR               lpCurrentDirectory,
    _In_        LPSTARTUPINFO         lpStartupInfo,
    _Out_       LPPROCESS_INFORMATION lpProcessInformation
);
```

Exemple n°1

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

int main(int argc, char *argv[]) {
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    if (argc != 2) {
        printf("Usage: %s [cmdline]\n", argv[0]);
        return -1;
    }

    if (!CreateProcess(NULL, /* No module name (use command line) */
```

```
        argv[1],            /* Its args */
        NULL,               /* Process handle not inheritable */
        NULL,               /* Thread handle not inheritable */
        FALSE,              /* Do not heritate of descriptors */
        0,                  /* No flags */
        NULL,               /* Uses parent's environment block */
        NULL,               /* Uses parent's starting directory */
        &si,                /* Pointer to STARTUPINFO structure */
        &pi)                /* Pointer to PROCESS_INFORMATION structure */
      ) {
        printf("Echec (%d).\n", GetLastError());
        return -1;
    }

    /* Wait until child process exits */
    WaitForSingleObject(pi.hProcess, INFINITE);
    /* Close process and thread handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);

    return 0;
}
```

Exemple n°2

```
/* child.c */
#include <windows.h>
#include <stdio.h>

int main(void) {
    printf("I am sleeping a while...\n");
    Sleep(5000);
    printf("Done.\n");

    return 0;
}
```

```
/* parent.c */
#include <windows.h>
#include <stdio.h>

int main(void) {
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory(&si, sizeof(si));
    ZeroMemory(&pi, sizeof(pi));

    si.cb = sizeof(si);
    if (!CreateProcess("child.exe", NULL, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi)) {
        return -1;
    }
```

```
    /* Wait until child process exits */
    WaitForSingleObject(pi.hProcess, INFINITE);
    /* Close process and thread handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);

    return 0;
}
```

Ms-Windows et les threads

UNIX	WIN32/64	Description
`pthread_create`	`CreateThread`	Crée un nouveau *thread*
`pthread_exit`	`ExitThread`	Termine un *thread*
`pthread_join`	`WaitForSingleObject`	Se met en attente de la fin d'un *thread* enfant

Ms-Windows et les flots d'entrées/sorties

L'accès aux entrées-sorties sous Ms-Windows peut se faire à l'aide de :

- la bibliothèque standard du C ;
- l'API Ms-Windows.

```
#include <stdio.h>

int main(void)
{
    FILE *in, *out;
    char buf[80];

    in = fopen(/*...*/);
    out = fopen(/*...*/);

    while (buf = fread(/*...*/, in)) {
        fwrite(/*...*/, buf, out);
    }

    fclose(in);
    fclose(out);
}
```

```
#include <windows.h>

int main(void)
 {
    HANDLE in, out ;
    char buf[80];

    in = CreateFile(/*...*/);
    out = CreateFile(/*...*/);

    while (ReadFile(in, buf)) {
        WriteFile(out, buf, /* ... */);
    }

    CloseHandle(in);
    CloseHandle(out);
 }
```

UNIX	WIN32/64	Description
`open`	`CreateFile`	Crée un fichier ou ouvre un fichier existant
`close`	`CloseHandle`	Ferme un fichier
`read`	`ReadFile`	Lit des données depuis un fichier
`write`	`WriteFile`	Écrit des données dans un fichier

UNIX	WIN32/64	Description
`lseek`	`SetFilePointer`	Déplace le pointeur de fichier
`stat`	`GetFileAttributesEx`	Retourne différents attributs du fichier
`dup` / `dup2`	`DuplicateHandle`	Modifie les attributs d'un fichier
`mkdir`	`CreateDirectory`	Crée un nouveau dossier
`rmdir`	`RemoveDirectory`	Supprime un répertoire vide
`link`		Création de liens
`unlink`	`DeleteFile`	Supprime un fichier existant
`mount`		Monte un système de fichiers à un emplacement
`umount`		Démonte un système de fichier
`chdir`	`SetCurrentDirectory`	Change le dossier de travail.
`chmod`		Modifie les bits de comportement (sécurité)

Le tampon d'un flux doit toujours être vidé : `fflush()` → `FlushFileBuffers()`

Ms-Windows et la gestion mémoire

UNIX	WIN32/64	Description
	`CreateFileMapping`	Crée une zone d'échange
	`OpenFileMapping`	Ouvre une zone d'échange
`mmap`	`MapViewOfFile`	Projection d'une zone d'échange vers la mémoire
`munmap`	`UnmapViewOfFile`	Déprojection d'une zone d'échange vers la mémoire
`malloc`	`HeapAlloc` ou `malloc`	Allocation dynamique
`free`	`HeapFree` ou `free`	Libération de mémoire

Ms-Windows et la communication interprocessus

Ms-Windows et les signaux

Les signaux types Unix ne font pas partie du système, par contre il existe des temporisateurs (*timers*) :

- `alarm()` → `SetTimer()` et `alarm(0)` → `KillTimer()`

Ms-Windows et les tubes

UNIX	WIN32/64	Description
`pipe`	`CreatePipe`	Crée un tube anonyme

UNIX	WIN32/64	Description
`mkfifo`	`CreateNamedPipe`	Crée un tube nommé
`open`	`ConnectNamedPipe`	Se connecte à un tube nommé

Les appels systèmes `ReadFile()`, `WriteFile()` et `CloseHandle()` peuvent être utilisés de manière similaire à leur utilisation sous Unix.

```
#include <windows.h>

BOOL WINAPI CreatePipe(
    _Out_    PHANDLE               hReadPipe,
    _Out_    PHANDLE               hWritePipe,
    _In_opt_ LPSECURITY_ATTRIBUTES lpPipeAttributes,
    _In_     DWORD                 nSize
);

HANDLE CreateNamedPipeA(
    LPCSTR                lpName,
    DWORD                 dwOpenMode,
    DWORD                 dwPipeMode,
    DWORD                 nMaxInstances,
    DWORD                 nOutBufferSize,
    DWORD                 nInBufferSize,
    DWORD                 nDefaultTimeOut,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

Exemple d'un processus parent

```
/*
 * parent.c
 * Parent process that communicates with its child using an
 * anonymous pipe.
 */
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

#define BUFSIZE 4096
HANDLE fd[2];

int main(void) {
    /* Create an anonymous pipe */
    SECURITY_ATTRIBUTES saAttr;
    saAttr.nLength = sizeof(SECURITY_ATTRIBUTES);
    saAttr.bInheritHandle = TRUE; /* inherit handles */
    saAttr.lpSecurityDescriptor = NULL;

    if (!CreatePipe(&fd[0], &fd[1], &saAttr, 0)) {
        printf("CreatePipe failed (%d).\n", GetLastError());
        ExitProcess(1);
    } else
```

```
        printf("fd[0]= %d, fd[1]=%d\n", fd[0], fd[1]);

    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* Convert the descriptor fd[0] to char[] */
    char buf[15];
    sprintf(buf, "child.exe %d", fd[0]);

    /* Create a child process */
    if (!CreateProcess("child.exe", buf, NULL, NULL, TRUE, 0, NULL, NULL, &si, &pi)) {
        printf("Error (%d).\n", GetLastError());
        ExitProcess(1);
    }
    CloseHandle(fd[0]);

    DWORD dwWritten;
    CHAR chBuf[BUFSIZE] = "Bonjour du parent à l'enfant\0";

    if (!WriteFile(fd[1], chBuf, strlen(chBuf)+1, &dwWritten, NULL)) {
        printf("WriteFile to pipe failed (%d).\n", GetLastError());
        return -1;
    } else
        printf("WriteFile to pipe succeeded (%d).\n", dwWritten);

    CloseHandle(FD[1]);

    /* Wait for the child to finish and exit. */
    WaitForSingleObject(pi.hProcess, INFINITE);

    /* Close process and thread handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
    ExitProcess(0);
}
```

Exemple du processus enfant

```
/* child.c */
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

int main(int argc, char * argv[]) {
    DWORD dwRead;
    char chBuf[4096];

    printf(""I am the child (PID %d, TID %d)\n", GetCurrentProcessId(), GetCurrentThreadId());
    HANDLE fd0 = (HANDLE) atoi(argv[1]);
    printf("argc = %d, argv[0]=%s, argv[1]=%s, fd0=%d\n", argc, argv[0], argv[1], fd0);
```

```
    if (!ReadFile(fd0,chBuf,4096,&dwRead,NULL) || dwRead==0) {
        printf("ReadFile from pipe failed (%d)\n", GetLastError());
        CloseHandle(fd0);
        ExitProcess(1);
    }
    printf("Message received: %s (size %d)\n", chBuf, dwRead);
    CloseHandle(fd0);

    ExitProcess(0);
}
```

Ms-Windows et la mémoire partagée

UNIX	WIN32/64	Description
shm_open	CreateFileMapping	Crée une zone d'échange
ftruncate	SetEndOfFile + SetFilePointer	Redimensionne la zone d'échange
close	CloseHandle	Libère la zone d'échange

Exemple : processus P1

```
/* shm1.c - process P1 */
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <tchar.h>

#define BUF_SIZE 256
TCHAR szName[] = TEXT("SharedMemory");
TCHAR szMsg[] = TEXT("Message from the first process.");

int main(void) {
    HANDLE hMapFile;
    LPCTSTR pBuf;

    hMapFile = CreateFileMapping(
       INVALID_HANDLE_VALUE, /* use paging file */
       NULL,                 /* default security */
       PAGE_READWRITE,       /* read/write access */
       0,                    /* max. object size */
       BUF_SIZE,             /* buffer size */
       szName);              /* name of the shared memory object */

    if (hMapFile == NULL) {
        printf(TEXT("Could not create file mapping object (%d).\n"), GetLastError());
        return 1;
    }

    pBuf = (LPTSTR) MapViewOfFile(
```

```
        hMapFile,            /* handle to the map object */
        FILE_MAP_ALL_ACCESS, /* read/write permission */
        0, 0, BUF_SIZE);

    if (pBuf == NULL) {
        printf(TEXT("Could not map view of file (%d).\n"), GetLastError());
        CloseHandle(hMapFile);
        return 1;
    }

    CopyMemory((PVOID)pBuf, szMsg, (_tcslen(szMsg) * sizeof(TCHAR)));
    _getch();

    UnmapViewOfFile(pBuf);
    CloseHandle(hMapFile);

    return 0;
}
```

Exemple : processus P2

```
/* shm2.c - process P2 */
#pragma comment(lib, "user32.lib")
#define BUF_SIZE 256
TCHAR szName[] = TEXT("Global\\SharedMemory");

int main(void) {
    HANDLE hMapFile;
    LPCTSTR pBuf;

    hMapFile = OpenFileMapping(
        FILE_MAP_ALL_ACCESS, /* read/write access */
        FALSE,               /* do not inherit the name */
        szName);             /* name of the shared memory object */

    if (hMapFile == NULL) {
        printf(TEXT("Could not open file mapping object (%d).\n"), GetLastError());
        return 1;
    }

    pBuf = (LPTSTR) MapViewOfFile(
        hMapFile,            /* handle to the map object */
        FILE_MAP_ALL_ACCESS, /* read/write permission */
        0, 0, BUF_SIZE);

    if (pBuf == NULL) {
        printf(TEXT("Could not map view of file (%d).\n"), GetLastError());
        CloseHandle(hMapFile);
        return 1;
    }

    MessageBox(NULL, pBuf, TEXT("Process P2"), MB_OK);
    UnmapViewOfFile(pBuf);
    CloseHandle(hMapFile);
```

```
    return 0;
}
```

Ms-Windows et les files de messages

Les files de messages n'existent pas sous Ms-Windows. Les tubes, plus flexibles que sous Unix seront utilisés à leur place.

Ms-Windows et la synchronisation

UNIX	WIN32/64	Description
`lockf`	`LockFile` / `UnlockFile`	Verrouillage/déverrouillage de fichier
`sem_init`	`CreateSemaphore`	Création et initialisation d'un sémaphore
`sem_open`	`OpenSemaphore`	Ouvre un sémaphore
`sem_wait`	`WaitForSingleObject`	Attend l'accès à un sémaphore
`sem_post`	`ReleaseSemaphore`	Redonne l'accès à un sémaphore
`sem_unlink`	`CloseHandle`	Détruit un sémaphore
`pthread_mutex_init`	`CreateMutex`	Création et initialisation d'un mutex
`pthread_mutex_lock`	`WaitForSingleObject`	Attend l'accès à un mutex
`pthread_mutex_unlock`	`ReleaseMutex`	Redonne l'accès à un mutex
`pthread_mutex_destroy`	`CloseHandle`	Détruit un mutex

Annexe C : Interfacer Java et C

L'interface **JNI** (*Java native interface*) propose un mécanisme qui permet de définir certaines méthodes non portables dans des langages tels que le langage C. Pour plus de détails, vous pouvez consulter la documentation officielle à l'adresse suivante : https://docs.oracle.com/javase/8/docs/technotes/guides/jni/ ou encore vous reporter à l'ouvrage de S. Liang [22].

JNI : principe

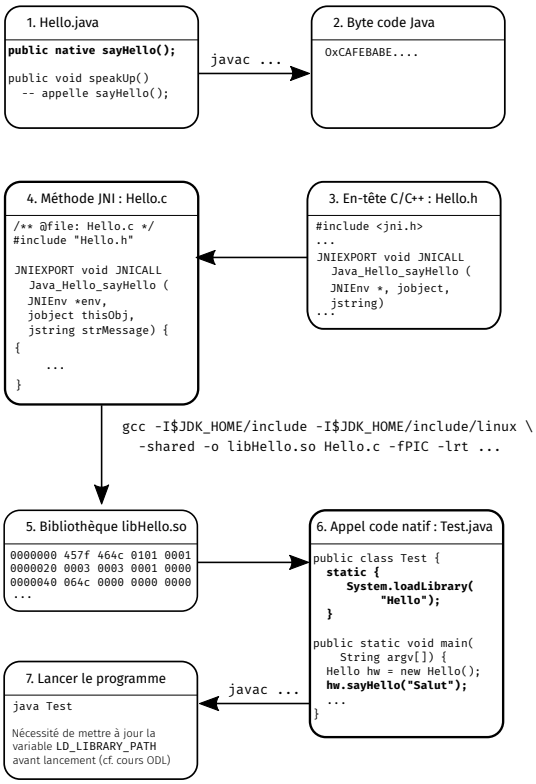


Fig. 3 : Étapes de mise en oeuvre de JNI.

JNI : fichiers d'en-tête

Les entêtes « `jni.h` » et « `jni_md.h` » définissent l'ensemble des options, des types et des structures utilisés par JNI. Ils sont respectivement placés dans les dossiers « `$JAVA_HOME/include/` » et « `$JAVA_HOME/include/[win32 ou linux]/` ». L'entête « `jni_md.h` » est spécifique à la plate-forme et inclus par « `jni.h` ».

Exemple de dossiers

```
/usr/lib/jvm/java-8-oracle/include
/usr/lib/jvm/java-8-oracle/include/linux
```

JNI : types de base et équivalents natifs

Type Java	Type natif	Taille (en bits)
`boolean`	`jboolean`	8 unsigned
`byte`	`jbyte`	8
`char`	`jchar`	16 unsigned
`double`	`jdouble`	64
`int`	`jint`	32
`float`	`jfloat`	32
`long`	`jlong`	64
`short`	`jshort`	16
`void`	`void`	-

JNI : classes Java et équivalents JNI

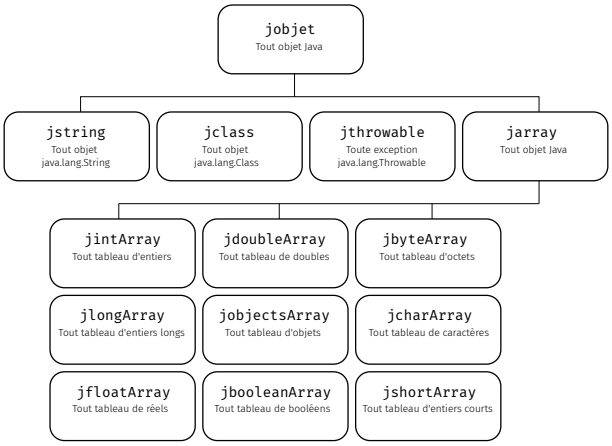


Fig. 4 : Classes JNI.

JNI : déclaration des fonctions

```
package test
class Prompt {
    private native String getLigne(String prompt);
}
```

```
JNIEXPORT jstring JNICALL Java_test_Prompt_getLigne(JNIEnv *,
                                                    jobject,
                                                    jstring);
```

```
Préfixe + "_" + NomPaquetage + "_" + NomClasse + "_" + NomMéthode
```

Fig. 5 : Transformation des méthodes en fonctions.

- `JNIEXPORT` : option C définie dans `jni_md.h` et correspondant à `__declspec(dllexport);`
- `JNICALL` : option C définie dans `jni_md.h` et correspondant à `__stdcall;`
- `JNIEnv` : structure de données définie dans `jni.h` et renseignant sur l'environnement de la machine virtuelle d'appel -- elle permet une communication entre la fonction C et le programme Java.
- `jobject` : structure définie dans `jni.h` et qui référence l'objet ayant appelé la fonction native.

JNI : accéder à des chaînes de caractères

Mauvais emploi de `jstring`

```
JNIEXPORT jstring JNICALL Java_Prompt_getLigne(JNIEnv *env, jobject obj, jstring prompt)  {
    printf("%s", prompt);
    /* ... */
}
```

Emploi correct de `jstring`

```
JNIEXPORT jstring JNICALL Java_Prompt_getLigne(JNIEnv *env, jobject obj, jstring prompt) {
    const char *my_string = (* env)->GetStringUTFChars(env, prompt, 0);
    printf("%s", my_string);

    /* ... */

    /* Free memory after using */
    (* env)->ReleaseStringUTFChars(env, prompt, my_string);

    /* ... */
}
```

JNI : accéder à des tableaux

Mauvaise utilisation d'un tableau d'entiers

```
JNIEXPORT jint JNICALL Java_TabEnt_sommer(JNIEnv *env, jobject obj, jintArray my_array) {
    int sum = 0;

    for (int i = 0; i < 10; i++)  {
        sum += my_array[i];
    }
    /* ... */
}
```

Utilisation correcte du tableau

```
JNIEXPORT jint JNICALL Java_TabEnt_sommer(JNIEnv *env, jobject obj, jintArray my_array) {
     int sum = 0;

    /* 1. Get the size of my_array */
    jsize size = (* env)->GetArrayLength(env, my_array);

    /* 2. Get a pointer to my_array */
    jint *elements = (* env)->GetIntArrayElements(env, my_array, 0);

    /* 3. Operate on elements of my_array */
    for (int i = 0; i < size; i++) {
        sum += elements[i];
    }
```

```
    /* 4. Free memory */
    (* env)->ReleaseIntArrayElements(env, my_array, elements, 0);
}
```

JNI : accéder à des attributs d'objet

Pour accéder aux attributs d'un objet, il est nécessaire de récupérer la signature des attributs. Utilisez la commande suivante :

```
$ javap -s -p NameOfTheClass
```

Cette signature est utilisée lors de l'appel des méthodes `GetFieldID()` et `GetStaticFieldID()` qui permettent de récupérer l'identifiant des attributs. Ensuite, l'accès aux attributs se fait de la manière suivante :

- Lecture : `Get<Type>Field()` ou `GetStatic<Type>Field()`
- Écriture : `Set<Type>Field()` ou `SetStatic<Type>Field()`

Exemple (lancer `javap -s -p AccessFields` pour obtenir une signature)

```
class AccessFields {
    static int si; /* signature "I" */
    String s;      /* signature "Ljava/lang/String;" */
}
```

```
 /* 1. Get the ID of the field */
 fid = (* env)->GetStaticFieldID(env, cls, "si", "I");

 /* 2. Find the corresponding attribute */
 si  = (* env)->GetStaticIntField(env, cls, fid);

 /* 3. Operate on the attribute */
 (* env)->SetStaticIntField(env, cls, fid, 200);

 /* 1. Get the ID of the field */

 fid = (* env)->GetFieldID(env, cls, "s", "Ljava/lang/String;");

 /* 2. Find the corresponding attribute */
 jstr = (* env)->GetObjectField(env, obj, fid);

 /* 3. Operate on the attribute */
 jstr = (* env)->NewStringUTF(env, "123");
 (* env)->SetObjectField(env, obj, fid, jstr);
```

JNI : appeler une méthode

Pour appeler une méthode, utilisez `GetObjectClass()` pour trouver la classe qui définit la méthode. Chaque méthode possède un identifiant obtenu par `GetMethodID()`. L'appel dépend du type de retour, JNI proposant une API pour chaque type (`CallVoidMethod()`, etc.).

Exemple

```
jclass cls = (*env)->GetObjectClass(env, obj);
jmethodID mid=(*env)->GetMethodID(env, cls, "displayMessage", "(I)V");
(*env)->CallVoidMethod(env, obj, mid, parm1);
```

JNI : exemple `Port`

```
public class Port {
    public Port() {}
    public native int initialize(int aPort);
    public native void write(int aValue);
    public native int read();
    public native int close();

    static
    {
      /*
       * Load libPort.so under Unix and Port.dll under Ms-Windows
       */
       System.loadLibrary("Port");
    }
}
```

Production de l'entête

```
javac Port.java
javah -jni Port
```

Fichier `Port.h` généré

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class Port*/

#ifndef _Included_Port
#define _Included_Port
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     Port
 * Method:    initialize
 * Signature: (I)V
 */
JNIEXPORT jint JNICALL Java_Port_initialize(JNIEnv *,jobject,jint);

/*
 * Class:     Port
 * Method:    write
 * Signature: (I)V
```

```
 */
JNIEXPORT void JNICALL Java_Port_write(JNIEnv *,jobject,jint);
...

/*
 * Class:     Port
 * Method:    read
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_Port_read(JNIEnv *,jobject);

/*
 * Class:     Port
 * Method:    close
 * Signature: (I)V
 */
JNIEXPORT jint JNICALL Java_Port_close(JNIEnv *,jobject);

#ifdef __cplusplus
}
#endif
```

Implémentation de `Port.c`

```
#include "Port.h"
#include <sys/io.h> /* for ioperm(), inb(), and outb() */

int port;

JNIEXPORT jint JNICALL Java_Port_initialize(JNIEnv *env, jobject obj, jint aPort) {
    port = ( int )aPort;
    /* ioperm provides access rights to use the port */
    return ( jint )ioperm(port, 1, 1);
}

JNIEXPORT void JNICALL Java_Port_write(JNIEnv *env, jobject obj, jint aValue) {
    outb(( int )aValue, port);
}

JNIEXPORT jint JNICALL Java_Port_read(JNIEnv *env, jobject obj) {
    jint value = ( jint )inb(port);
    return value;
}

JNIEXPORT jint JNICALL Java_Port_close(JNIEnv *env, jobject obj) {
    /* Remove access rights to use the port */
    return ( jint )ioperm(port, 1, 0);
}
```

Compilation

```
gcc -I/usr/lib/jvm/java-8-oracle/include -I/usr/lib/jvm/java-8-oracle/include/linux -O -shared -o
↪  libPort.so Port.c -fPIC
```

Les fichiers produits sont les suivants :

- `Port.c`
- `Port.class`
- `Port.java`
- `Port.o`
- `Port.h`
- `libPort.so`

Il faut penser à exporter la bibliothèque dynamique :

```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

Le fichier Makefile correspondant est le suivant :

```
CC = gcc
JDK = /usr/lib/jvm/java-8-oracle
CPPFLAGS = -I$(JDK)/include -I$(JDK)/include/linux
CFLAGS = -g -O2 -shared -I$(JDK)/include -I$(JDK)/include/linux

.PHONY: all clean

all: port
    @LD_LIBRARY_PATH=.:$$LD_LIBRARY_PATH; export LD_LIBRARY_PATH; \

port: Port.class libPort.so

Port.class: Port.java
    javac Port.java
    javah -jni Port

libPort.so: Port.c Port.h
    ($CC) ($CPPFLAGS) ($CFLAGS) -o libPort.so Port.c -fPIC

clean:
    rm -f *.class *.so *.o
```

Cette annexe vise à fournir une introduction succinte à l'utilisation de l'interface JNI pour interfacer du code Java avec du code C, en détaillant les étapes nécessaires pour déclarer, implémenter et compiler des méthodes natives.

Annexe D : Algorithmes d'ordonnancement

Sans préemption

Algorithme FCFS (*first come first served*)

L'algorithme d'ordonnancement FCFS est l'un des algorithmes les plus simples à réaliser. Les processus sont exécutés dans l'ordre de leur arrivée. Cet algorithme ne prend pas en compte la durée d'exécution des processus,

ce qui peut amener les processus plus courts à rester bloqués derrière les processus plus longs.

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

#define MAX_PROCESSES 100

typedef struct {
    int pid;      /* Process ID */
    int arrival;  /* Arrival time */
    int burst;    /* Burst time */
    int start;    /* Start time */
    int finish;   /* Finish time */
    int wait;     /* Waiting time */
    int turnaround; /* Turnaround time */
} process_t;

void fcfs(process_t proc[], int n) {
    int current_time = 0;

    for (int i = 0; i < n; i++) {
        /* Process start time is the maximum of the current time or arrival time */
        proc[i].start = (current_time > proc[i].arrival) ? current_time : proc[i].arrival;
        proc[i].finish = proc[i].start + proc[i].burst;
        proc[i].wait = proc[i].start - proc[i].arrival;
        proc[i].turnaround = proc[i].finish - proc[i].arrival;
        current_time = proc[i].finish;
    }
}

void print_processes(process_t proc[], int n) {
    printf("PID\tArrival\tBurst\tStart\tFinish\tWait\tTurnaround\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", proc[i].pid, proc[i].arrival, proc[i].burst,
↪   proc[i].start, proc[i].finish, proc[i].wait, proc[i].turnaround);
    }
}

int main(void) {
    int n; /* Number of processes */
    process_t proc[MAX_PROCESSES];

    printf("Enter the number of processes: ");
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        proc[i].pid = i + 1;
        printf("Enter arrival time and burst time for process %d: ", proc[i].pid);
        scanf("%d %d", &proc[i].arrival, &proc[i].burst);
    }

    fcfs(proc, n);
    print_processes(proc, n);
```

```
    return EXIT_SUCCESS;
}
```

Cet algorithme FCFS est un bon point de départ pour comprendre les concepts de base de l'ordonnancement des processus. Il est facile à implémenter et à comprendre, mais il peut entraîner des temps d'attente élevés pour les processus arrivant plus tard, surtout si un processus long est en cours d'exécution.

Algorithme SJF (*shortest job first*)

L'algorithme SJF sélectionne le processus avec le temps le plus court pour l'exécution suivante. Cet algorithme peut réduire le temps d'attente moyen des processus, mais il nécessite une connaissance préalable des durées, ce qui n'est pas toujours possible. L'implémentation ci-dessous suppose que les processus sont triés par ordre croissant de leur durée.

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

#define MAX_PROCESSES 100

typedef struct {
    int pid;      /* Process ID */
    int arrival;  /* Arrival time */
    int burst;    /* Burst time */
    int start;    /* Start time */
    int finish;   /* Finish time */
    int wait;     /* Waiting time */
    int turnaround; /* Turnaround time */
    bool completed; /* Completion status */
} process_t;

void sjf(process_t proc[], int n) {
    int current_time = 0, completed = 0;
    while (completed < n) {
        int shortest = -1;
        for (int i = 0; i < n; i++) {
            if (!proc[i].completed && proc[i].arrival <= current_time) {
                if (shortest == -1 || proc[i].burst < proc[shortest].burst) {
                    shortest = i;
                }
            }
        }

        if (shortest != -1) {
            proc[shortest].start = current_time;
            proc[shortest].finish = proc[shortest].start + proc[shortest].burst;
            proc[shortest].wait = proc[shortest].start - proc[shortest].arrival;
            proc[shortest].turnaround = proc[shortest].finish - proc[shortest].arrival;
            current_time = proc[shortest].finish;
            proc[shortest].completed = true;
            completed++;
        } else {
```

```
            current_time++;
        }
    }
}

void print_processes(process_t proc[], int n) {
    printf("PID\tArrival\tBurst\tStart\tFinish\tWait\tTurnaround\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", proc[i].pid, proc[i].arrival, proc[i].burst,
↪   proc[i].start, proc[i].finish, proc[i].wait, proc[i].turnaround);
    }
}

int main(void) {
    int n; /* Number of processes */
    process_t proc[MAX_PROCESSES];

    printf("Enter the number of processes: ");
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        proc[i].pid = i + 1;
        proc[i].completed = false;
        printf("Enter arrival time and burst time for process %d: ", proc[i].pid);
        scanf("%d %d", &proc[i].arrival, &proc[i].burst);
    }

    sjf(proc, n);
    print_processes(proc, n);

    return EXIT_SUCCESS;
}
```

L'algorithme SJF est efficace pour minimiser le temps d'attente moyen des processus. Cependant, il peut entraîner une « injustice » pour les processus plus longs et souffrir du problème de l'inversion de priorité si les durée des processus ne sont pas bien estimés.

Algorithme d'ordonnancement par priorité

L'algorithme d'ordonnancement par priorité affecte à chaque processus une priorité et le processus avec la priorité la plus élevée est exécuté en premier. Cet algorithme peut être non préemptif (où un processus en cours d'exécution n'est pas interrompu) ou préemptif (où un processus peut être interrompu par un processus de priorité plus élevée).

Voici une implémentation non préemptive de l'algorithme d'ordonnancement par priorité :

```
#include <stdio.h>

#define MAX_PROCESSES 100

typedef struct {
    int pid;      /* Process ID */
    int arrival;  /* Arrival time */
```

```
    int burst;    /* Burst time */
    int priority;  /* Priority */
    int start;    /* Start time */
    int finish;   /* Finish time */
    int wait;     /* Waiting time */
    int turnaround; /* Turnaround time */
    int remaining; /* Remaining time */
} process_t;

void sort_by_priority(process_t proc[], int n) {
    process_t temp;
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (proc[j].priority < proc[i].priority) {
                temp = proc[i];
                proc[i] = proc[j];
                proc[j] = temp;
            }
        }
    }
}

void priority_scheduling(process_t proc[], int n) {
    int current_time = 0;
    int completed = 0;

    while (completed < n) {
        sort_by_priority(proc, n);
        for (int i = 0; i < n; i++) {
            if (proc[i].arrival <= current_time && proc[i].remaining > 0) {
                if (proc[i].remaining == proc[i].burst) {
                    proc[i].start = current_time;
                }

                current_time += proc[i].remaining;
                proc[i].finish = current_time;
                proc[i].turnaround = proc[i].finish - proc[i].arrival;
                proc[i].wait = proc[i].turnaround - proc[i].burst;
                proc[i].remaining = 0;
                completed++;
                break;
            }
        }
    }
}

void print_processes(process_t proc[], int n) {
    printf("PID\tArrival\tBurst\tPriority\tStart\tFinish\tWait\tTurnaround\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t\t%d\t%d\t%d\t%d\n", proc[i].pid, proc[i].arrival, proc[i].burst,
↪   proc[i].priority, proc[i].start, proc[i].finish, proc[i].wait, proc[i].turnaround);
    }
}
```

```
int main() {
    int n; // Number of processes
    process_t proc[MAX_PROCESSES];

    printf("Enter the number of processes: ");
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        proc[i].pid = i + 1;
        printf("Enter arrival time, burst time, and priority for process %d: ", proc[i].pid);
        scanf("%d %d %d", &proc[i].arrival, &proc[i].burst, &proc[i].priority);
        proc[i].remaining = proc[i].burst;
    }

    priority_scheduling(proc, n);
    print_processes(proc, n);

    return 0;
}
```

Dans cet exemple, les processus sont triés par priorité avant chaque exécution. Un processus est sélectionné pour l'exécution en fonction de sa priorité et s'il est arrivé. Les processus sont ensuite exécutés jusqu'à ce qu'ils soient terminés.

L'algorithme d'ordonnancement par priorité peut entraîner une famine si les processus à basse priorité attendent indéfiniment. Pour atténuer ce problème, une approche de priorité vieillissante peut être utilisée où la priorité des processus augmente au fil du temps.

Avec préemption

Algorithme RR (*round robin*)

L'algorithme circulaire est un algorithme d'ordonnancement préemptif où chaque processus bénéficie d'un *quantum* pendant lequel il peut utiliser le processeur. Si un processus n'est pas terminé à la fin de son *quantum*, il est replacé à la fin de la file d'attente. Cet algorithme est simple et équitable, mais la performance dépend du choix du *quantum*.

```
#include <stdio.h>

#define MAX_PROCESSES 100

typedef struct {
    int pid;        /* Process ID */
    int arrival;    /* Arrival time */
    int burst;      /* Burst time */
    int remaining;  /* Remaining time */
    int start;      /* Start time */
    int finish;     /* Finish time */
    int wait;       /* Waiting time */
    int turnaround; /* Turnaround time */
} process_t;
```

```
void round_robin(process_t proc[], int n, int quantum) {
    int current_time = 0;
    int completed = 0;
    int index = 0;

    while (completed < n) {
        if (proc[index].arrival <= current_time && proc[index].remaining > 0) {
            if (proc[index].remaining == proc[index].burst) {
                proc[index].start = current_time;
            }

            if (proc[index].remaining <= quantum) {
                current_time += proc[index].remaining;
                proc[index].finish = current_time;
                proc[index].turnaround = proc[index].finish - proc[index].arrival;
                proc[index].wait = proc[index].turnaround - proc[index].burst;
                proc[index].remaining = 0;
                completed++;
            } else {
                current_time += quantum;
                proc[index].remaining -= quantum;
            }
        }
        index = (index + 1) % n;
    }
}

void print_processes(process_t proc[], int n) {
    printf("PID\tArrival\tBurst\tStart\tFinish\tWait\tTurnaround\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
               proc[i].pid, proc[i].arrival, proc[i].burst,
               proc[i].start, proc[i].finish, proc[i].wait, proc[i].turnaround);
    }
}

int main(void) {
    int n; /* Number of processes */
    int quantum; /* Quantum time */
    process_t proc[MAX_PROCESSES];

    printf("Enter the number of processes: ");
    scanf("%d", &n);

    printf("Enter the quantum time: ");
    scanf("%d", &quantum);

    for (int i = 0; i < n; i++) {
        proc[i].pid = i + 1;
        printf("Enter arrival time and burst time for process %d: ", proc[i].pid);
        scanf("%d %d", &proc[i].arrival, &proc[i].burst);
        proc[i].remaining = proc[i].burst;
    }
```

```
    round_robin(proc, n, quantum);
    print_processes(proc, n);

    return EXIT_SUCCESS;
}
```

L'algorithme RR garantit qu'aucun processus ne monopolise le CPU, ce qui améliore l'équité. Toutefois, la performance dépend grandement du choix du *quantum*. Un *quantum* trop petit peut entraîner un surcoût de commutation contextuelle élevé, tandis qu'un *quantum* trop grand peut dégrader l'équité et se rapprocher du comportement du FCFS.

Algorithme SRTF (***shortest remaining time first***)

L'algorithme SRTF est une extension préemptive de l'algorithme SJF. Dans SRTF, le processus avec le plus court temps restant d'exécution est choisi pour être exécuté en premier. Si un nouveau processus arrive avec un temps d'exécution plus court que le temps restant du processus actuel, le processus en cours est préempté et le nouveau processus est exécuté.

Voici une implémentation de l'algorithme :

```
#include <stdio.h>
#include <limits.h>

#define MAX_PROCESSES 100

typedef struct {
    int pid;        /* Process ID */
    int arrival;    /* Arrival time */
    int burst;      /* Burst time */
    int remaining;  /* Remaining time */
    int start;      /* Start time */
    int finish;     /* Finish time */
    int wait;       /* Waiting time */
    int turnaround; /* Turnaround time */
} process_t;

void srtf_scheduling(process_t proc[], int n) {
    int current_time = 0;
    int completed = 0;
    int shortest = -1;
    int min_remaining = INT_MAX;

    while (completed < n) {
        for (int i = 0; i < n; i++) {
            if (proc[i].arrival <= current_time
                && proc[i].remaining > 0
                && proc[i].remaining < min_remaining) {
                min_remaining = proc[i].remaining;
                shortest = i;
            }
        }
```

```
        if (shortest == -1) {
            current_time++;
            continue;
        }

        if (proc[shortest].remaining == proc[shortest].burst) {
            proc[shortest].start = current_time;
        }

        proc[shortest].remaining--;
        current_time++;

        if (proc[shortest].remaining == 0) {
            proc[shortest].finish = current_time;
            proc[shortest].turnaround = proc[shortest].finish - proc[shortest].arrival;
            proc[shortest].wait = proc[shortest].turnaround - proc[shortest].burst;
            completed++;
            shortest = -1;
            min_remaining = INT_MAX;
        }
    }
}

void print_processes(process_t proc[], int n) {
    printf("PID\tArrival\tBurst\tStart\tFinish\tWait\tTurnaround\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", proc[i].pid, proc[i].arrival, proc[i].burst,
↪   proc[i].start, proc[i].finish, proc[i].wait, proc[i].turnaround);
    }
}

int main(void) {
    int n; /* Number of processes */
    process_t proc[MAX_PROCESSES];

    printf("Enter the number of processes: ");
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        proc[i].pid = i + 1;
        printf("Enter arrival time and burst time for process %d: ", proc[i].pid);
        scanf("%d %d", &proc[i].arrival, &proc[i].burst);
        proc[i].remaining = proc[i].burst;
    }

    srtf_scheduling(proc, n);
    print_processes(proc, n);

    return EXIT_SUCCESS;
}
```

Dans cette implémentation, le processus avec le temps restant le plus court est sélectionné pour l'exécution à chaque instant. Si un processus avec un temps restant plus court arrive, le processus en cours est préempté et le

nouveau processus est exécuté. Cela continue jusqu'à ce que tous les processus soient terminés.

Cet algorithme minimise le temps d'attente moyen et le temps de rotation moyen par rapport aux autres algorithmes sans préemption. Cependant, il peut entraîner une forte fragmentation du temps de processeur si les processus courts continuent à arriver, ce qui peut pénaliser les processus longs.

Glossaire

Adresse physique

Localisation réelle d'une donnée dans la mémoire RAM de l'ordinateur. Les adresses physiques sont utilisées par le matériel pour accéder à la mémoire.

Adresse virtuelle

Adresse générée par les processus d'un système d'exploitation, qui est ensuite traduite en une adresse physique par la mémoire virtuelle pour accéder à la mémoire RAM.

Algorithme de Peterson

Algorithme d'exclusion mutuelle pour deux processus basé sur l'utilisation de deux variables de signalisation et une variable tournante.

Algorithme de la boulangerie

Algorithme d'exclusion mutuelle qui permet à un nombre non borné de processus d'entrer dans leur section critique, tout en garantissant l'exclusion mutuelle.

Appel système

Mécanisme permettant aux programmes en mode utilisateur de demander des services au noyau du système d'exploitation.

Atomicité

Propriété d'une opération qui s'exécute en totalité sans interruption ni interférence, garantissant que toute opération est complète ou pas du tout exécutée.

Attente active

Technique de synchronisation où un processus boucle continuellement en vérifiant une condition plutôt que de se mettre en sommeil.

Big Endian / Little Endian

Formats pour le stockage des octets dans les types de données. *Big Endian* stocke l'octet le plus significatif à l'adresse la plus basse. Quant à *Little Endian* c'est l'octet le moins significatif qui est stocké à l'adresse la plus basse.

Big Endian stocke l'octet de poids fort en premier, tandis que *Little Endian* stocke l'octet de poids faible en premier.

Bloc de contrôle du processus (PCB)

Structure de données contenant les informations nécessaires pour gérer un processus.

Cache

Petite quantité de mémoire plus rapide située sur ou à proximité du processeur, utilisée pour stocker les copies des données et des instructions les plus fréquemment accédées provenant de la mémoire principale.

Cadre (***frame***)

Unité de mémoire dans laquelle les pages sont chargées en mémoire physique. Les cadres sont de taille fixe, identique à celle des pages.

Communication interprocessus (IPC)

Mécanismes permettant aux processus de communiquer et de synchroniser leurs actions sans partager le même espace mémoire.

Commutation de contexte

Mécanisme de sauvegarde de l'état d'un processus en cours d'exécution et de restauration de l'état d'un autre processus.

Conditions de Coffman

Ensemble de quatre conditions qui doivent toutes être présentes pour qu'un interblocage survienne : exclusion mutuelle, maintien et attente, non-préemption et attente circulaire.

Contexte d'un processus

Ensemble des informations nécessaires pour sauvegarder et restaurer l'état d'un processus lors de la commutation de contexte.

Défaut de Page (***page fault***)

Événement déclenché lorsque le système tente d'accéder à une page qui n'est pas présente en mémoire physique, nécessitant le chargement de cette page depuis la mémoire secondaire.

Descripteur de fichier

Entier représentant un fichier ouvert.

Entrée/Sortie

Opérations de communication entre un système informatique et le monde extérieur (humains, autres systèmes) via des périphériques tels que le clavier, l'écran, le disque dur, etc.

États d'un processus

Différentes phases qu'un processus peut traverser, telles que initialisé, prêt, élu, bloqué, et terminé.

Exclusion Mutuelle

Stratégie de synchronisation qui assure qu'une seule tâche ou processus peut accéder à une ressource à la fois.

Famine

Situation dans laquelle un ou plusieurs processus ne peuvent pas progresser car ils n'obtiennent jamais les ressources nécessaires, souvent en raison d'une allocation prioritaire des ressources à d'autres processus.

Fichier

Ensemble de blocs sur une unité de stockage.

File de messages (***message queue***)

Mécanisme IPC permettant de stocker des messages à destination ou en provenance de processus, offrant une communication asynchrone entre eux.

Flux d'entrées-sorties

Canaux de communication entre le processus et son environnement extérieur.

Hiérarchie des processus

Organisation des processus en une structure arborescente où chaque processus a un parent.

i-noeud (***inode***)

Structure contenant des métadonnées sur les fichiers.

Interblocage (*deadlock*)

Situation dans laquelle deux ou plusieurs processus sont bloqués indéfiniment en attendant qu'une ressource détenue par un autre processus soit libérée.

Journalisation

Fonctionnalité qui enregistre les modifications apportées aux fichiers pour permettre une récupération rapide après des pannes.

Mémoire partagée

Mécanisme IPC fournissant un segment de mémoire accessible par plusieurs processus, et utilisé pour échanger de grandes quantités d'information.

Mémoire virtuelle

Espace d'adressage que le système d'exploitation crée pour chaque processus et qui permet une abstraction et une gestion plus efficace de la mémoire physique.

MMU (*memory management unit*)

Composant du système informatique qui gère la conversion des adresses virtuelles en adresses physiques. La MMU utilise des structures telles que des tables de pages pour cette conversion.

Moniteur

Structure de données de haut niveau qui combine des variables partagées, des variables de condition et des procédures pour gérer l'accès concurrent à des ressources.

Mutex

Objet de synchronisation utilisé spécifiquement pour l'exclusion mutuelle dans la programmation multithreads, par opposition à la programmation multiprocessus, où d'autres mécanismes tels que les sémaphores sont préférés.

Ordonnancement des processus

Stratégie utilisée par l'ordonnanceur pour déterminer l'ordre d'exécution des processus.

Ordonnanceur

Composant du système d'exploitation responsable de la gestion de l'exécution des processus en allouant le processeur selon diverses stratégies d'ordonnancement.

Page

Bloc de mémoire de taille fixe qui constitue l'unité de base de la gestion de la mémoire dans les systèmes utilisant la pagination.

Pagination

Technique de gestion de la mémoire où la mémoire est divisée en sections de taille égale, appelées pages, qui sont gérées indépendamment.

Parallélisme

Exécution simultanée de plusieurs processus sur des processeurs ou coeurs différents.

Permission

Droit d'accès à un fichier (lecture, écriture, exécution).

PID (*process identifier*)

Identifiant unique attribué à chaque processus en cours d'exécution.

Planificateur

Synonyme d'ordonnanceur, bien que moins couramment utilisé.

Processus

Exécution d'un programme statique par un processeur. Il est constitué de plusieurs segments de mémoire et est caractérisé par un état à chaque instant de son déroulement.

Processus bloqué

État d'un processus qui attend la fin d'une opération d'entrée/sortie ou d'un événement spécifique.

Processus en attente

État d'un processus qui est prêt à s'exécuter, mais ne l'est pas actuellement en raison de l'ordonnancement. Les états ”en attente” et ”bloqué” sont souvent confondus.

Processus orphelin

Processus dont le parent s'est terminé avant lui, ce qui le rattache au processus primitif (init ou launchd).

Processus zombie

Processus terminé mais non encore nettoyé par le processus parent, laissant une entrée résiduelle dans la table des processus.

Producteur/Consommateur

Modèle de conception où les processus producteurs génèrent des données et les placent dans un tampon, et les processus consommateurs retirent ces données pour les utiliser.

Pseudo-parallélisme

Technique de partage de temps où plusieurs processus semblent s'exécuter en même temps sur un seul processeur par une commutation rapide entre eux. Ce terme se réfère souvent au parallélisme au niveau des *threads* au sein d'un seul coeur de processeur, par opposition au parallélisme réel obtenu sur un système multiprocesseurs.

Pthread

Bibliothèque POSIX multiplateforme pour la gestion des *threads*.

Quantum

Intervalle de temps durant lequel un processus est autorisé à s'exécuter avant qu'une commutation de contexte ne soit effectuée.

Rendez-vous

Méthode de synchronisation où plusieurs processus ou threads se rencontrent à un point de contrôle prédéfini avant de continuer leur exécution.

Section critique

Partie du code où les processus accèdent à des ressources partagées et qui nécessite un contrôle d'accès pour éviter les conflits.

Segmentation

Méthode de gestion de la mémoire où l'espace d'adressage est divisé en segments de taille variable qui peuvent être de tailles différentes, reflétant les besoins logiques des programmes.

Sémaphore

Objet de synchronisation utilisé pour contrôler l'accès à une ressource commune par plusieurs processus dans un environnement de système d'exploitation multitâches. Un sémaphore peuvent être de type compteur (permettant à un nombre limité de processus ou de *threads* d'accéder à une ressource) ou binaire (fonctionnant comme un mutex).

Signal

Mécanisme IPC utilisé pour notifier un processus de la survenue d'un événement particulier.

Socket

Mécanisme IPC qui permet d'échanger des données entre processus, soit au sein d'un même système (*sockets* Unix) soit entre différents systèmes (*sockets* Internet).

Swap

Technique de gestion de la mémoire où les pages de mémoire inactive sont déplacées vers un espace de stockage sur disque dur pour libérer de la mémoire physique pour les autres processus actifs.

Synchronisation

Ensemble de techniques permettant de coordonner l'exécution des processus ou des *threads* pour éviter les conflits d'accès aux ressources partagées.

Système de fichiers

Méthode et structure de données qu'un système d'exploitation utilise pour gérer les fichiers sur un disque ou une partition.

Système de gestion de fichiers (SGF)

Responsable de l'organisation, de la désignation, de la conservation permanente, de la protection et de l'accès aux fichiers.

Table des pages

Structure de données utilisée par la MMU pour maintenir la trace de la correspondance entre les adresses virtuelles et les adresses physiques.

Thread

Unité de base d'utilisation du processeur dans une application, représentant une séquence d'instructions.

TLB (*translation lookaside buffer*)

Cache spécialisé utilisé pour accélérer la traduction des adresses virtuelles en adresses physiques en gardant les résultats des récentes traductions.

Tube (*pipe*)

Mécanisme IPC qui permet la communication unidirectionnelle entre deux processus à l'aide d'un flux de données.

Variables d'environnement

Variables accessibles par les processus pour définir leur environnement d'exécution.

Verrou

Mécanisme qui permet de contrôler l'accès aux ressources en les réservant à un seul processus à la fois.

Index

Allocation dynamique, 95
  free(), 95
  malloc(), 95
Appel système, 7
  close(), 74
  dup(), 80
  dup2(), 80
  exec(), 33, 35, 94
  fork(), 25, 26, 35, 94
  getgid(), 29
  getpid(), 23
  getppid(), 23
  getuid(), 29
  kill(), 24, 29, 105
  lseek(), 74
  mmap(), 91, 124
  munmap(), 91, 124
  nanosleep(), 27, 30
  open(), 72
  read(), 75
  shm_open(), 123
  shm_unlink(), 123
  sigaction(), 106
  sigsuspend(), 30
  wait(), 30, 35
  waitpid(), 30
  write(), 77

Bit de mode, 7

Communication interprocessus
  Files de messages, 130
    Configuration, 131
    Création, 131
    Destruction, 131
    Envoi, 131
    Fermeture, 131
    Mise en oeuvre, 132
    Ouverture, 131
    Réception, 131
  Mémoire partagée, 122
    Création, 123
    Dimensionnement, 123
    Démappage, 124
    Fermeture, 123
    Mappage, 124
    Ouverture, 123
  Signaux, 101
    Envoi, 105
    Gestion, 106
    Gestion avancée, 113
    SA_RESTART, 115
    États, 103
  Sockets, 135
    Mode connecté (TCP), 139
    Mode non connecté (UDP), 142
  Tubes, 116
    Anonymes, 117
    mkfifo(), 119
    Nommés, 119
Conversion d'adresse, 86
  MMU, 86
  TLB, 86

Défaut de page, 87
  Anomalie de Bélády, 87

Fichier, 59
  Configuration, 66
  Données, 64
  Exécutable, 64
  Journal (Log), 65
  Sauvegarde (Backup), 66
  Spécial, 67
  Temporaire, 65
Flux d'entrées-sorties, 79
  Redirection, 80
  Table descripteurs, 79

init, 29

launchd, 29
Lien

Lien physique, 62
Lien symbolique, 62

Mode superviseur, 7
Mode utilisateur, 7
Mémoire, 17, 83
Cadres, 85
Hiérarchie, 83
Mémoire virtuelle et fichiers, 91
Pages, 85
Paginée, 86
Physique, 17, 83
Segments, 85
Taille des pages, 85
Virtuelle, 17, 83–85
Mémoires
Cartographie
Bibliothèques partagées, 89
Segments, 87, 88
Tas, 90

Noyau (Kernel), 6

Ordonnanceur, 18

Primitive, 7
Processus, 15
Bloc de contrôle (PCB), 18
Commutation de contexte, 18, 19
Contexte, 18
Démon, 33, 111
Hiérarchie, 29
Identifiant (PID), 18, 19, 22, 23
Identifiant du parent(PPID), 18, 19, 23
Multiplexage, 17
Ordonnancement, 18, 19
CFS, 20
FCFS, 20
MLFQS, 20
MLQS, 20
Ordonnancement circulaire, 20
Ordonnancement par priorité, 20
SJF, 20
SRTF, 20
Orphelin, 32
Parallélisme, 16
Zombie, 16, 31
États, 15, 16, 21
Protection des fichiers, 71
Délégation, 72
Permissions, 71
Préemption, 19

Segments mémoire
Fichiers, 91
Partage de segments, 92
Segment partagé, 92
Segment privé, 93
Swap, 91
Signal, 24
Structure des fichiers, 68
Blocs, 68
Dossier, 70
i-noeud (inode), 69
Synchronisation, 149
Compétition, 149
Algorithme de la boulangerie, 154
Algorithme de Peterson, 153
Attente active, 153
Interblocage, 157
Moniteur, 152
Sémaphore, 151
Verrou, 150
Verrouillage de fichiers, 155
Verrouillage par test et modification, 154
Coopération, 162
Lecteur-imprimeur, 163
Producteur-consommateur, 162
Rendez-vous, 165
Systèmes de fichiers
APFS, 59
Btrfs, 59, 61
exFAT, 59, 61
ext, 59, 61
FAT32, 59, 60
HFS+, 59, 61
JFS, 59
JFS, JFS2, 61
NTFS, 59, 60
UFS, 59, 61
ZFS, 59, 61

Temps d'attente
nanosleep(), 27
sleep(), 21, 27
usleep(), 27
Thread, 15, 39
Attributs, 45
Détachement, 45

Héritage des attributs, 46
Ordonnancement, 46
Pile, 45
Portée du thread, 46
Protection de la pile, 46
Comparaison avec les processus, 42
Critères de choix, 42
Espace mémoire, 39
Processus multithreads, 40
Représentation alternative, 40
Exécution concurrente, 42
Fonctions re-entrantes, 55
Fonctions sûres, 54
Classes non sûres, 54
Partage de variables, 52
Accès à la pile, 53
Pthread, 43, 44
Attendre des threads, 44
Créer un thread, 43
Identifier un thread, 44
Terminer un thread, 44
Synchronisation, 44
Vue logique, 41

Variable d'environnement, 23

Bibliographie

[1] A. Baumann, J. Appavoo, O. Krieger, T. Roscoe, A fork() in the road, Proceedings of the Workshop on Hot Topics in Operating Systems (2019).

[2] L.A. Belady, R.A. Nelson, G.S. Shedler, An anomaly in space-time characteristics of certain programs running in a paging machine, 12 (1969).

[3] F. Berman, G. Fox, A.J.G. Hey, T. Hey, Grid Computing : Making the Global Infrastructure a Reality, John Wiley & Sons, Inc., USA, 2003.

[4] C. Blaess, Développement système sous Linux, 5 éd., Eyrolles, 2019.

[5] R.E. Bryant, D.R. O'Hallaron, Computer Systems : A Programmer's Perspective, 3 éd., Addison-Wesley, 2016.

[6] R. Clouard, A. Lebret, Outils de développement logiciel, (2024).

[7] E.G. Coffman, M. Elphick, A. Shoshani, System Deadlocks, 3 (1971) 67 à 78.

[8] M. Crochemore, C. Hancart, T. Lecroq, Algorithmique du texte, Vuibert, 2001.

[9] J. Davidson, J. Peters, B. Grace, Voice over IP Fundamentals, 2 éd., Cisco Press, 2006.

[10] E.W. Dijkstra, Cooperating sequential processes, in : The origin of concurrent programming : from semaphores to remote procedure calls, Springer, 2002 : p. 65 à 138.

[11] S. Garfinkel, G. Spafford, A. Schwartz, Practical Unix & Internet Security, 3 éd., O'Reilly Media, Inc., 2003.

[12] GeeksForGeeks.org, Tutoriel Blockchain, (2024).

[13] Gnu.org, Bash Reference Manual, (2024).

[14] R.C. Gonzalez, R.E. Woods, Digital Image Processing, 4 éd., Prentice-Hall, Inc., USA, 2019.

[15] J.M. Hart, Windows System Programming, 4 éd., Addison-Wesley, USA, 2010.

[16] J.L. Hennessy, D.A. Patterson, Computer Architecture : A Quantitative Approach, 5 éd., Morgan Kaufmann, Amsterdam, 2012.

[17] L.S. Hill, Concerning Certain Linear Transformation Apparatus of Cryptography, The American Mathematical Monthly 38 (1931) 135 154.

[18] J. Humble, D. Farley, Continuous Delivery : Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley Professional, 2010.

[19] B.W. Kernighan, D.M. Ritchie, The C Programming Language, 2 éd., Prentice Hall Professional Technical Reference, 1988.

[20] S. Krakowiak, Brève histoire des systèmes d'exploitation, 1024 8 (2016) 67 à 91.

[21] L. Lamport, A new solution of Dijkstra's concurrent programming problem, Commun. ACM 17 (1974) 453 à 455.

[22] S. Liang, Java Native Interface : Programmer's Guide and Reference, Addison-Wesley Professional, USA, 1999.

[23] J.W.S.W. Liu, Real-Time Systems, Prentice Hall PTR, USA, 2000.

[24] R. Love, Linux System Programming : Talking Directly to the Kernel and C Library, 2 éd., O'Reilly Media, 2013.

[25] H. Marco-Gisbert, I. Ripoll, Address Space Layout Randomization Next Generation, Applied Sciences 9 (2019).

[26] Michael Kerrisk, The Linux Programming Interface, No Starch Press, 2010.

[27] S.B. Needleman, C.D. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of two proteins, Journal of Molecular Biology 48 (1970) 443 453.

[28] OpenCV, The OpenCV Reference Manual, (2024).

[29] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical Recipes 3rd Edition : The Art of Scientific Computing, 3 éd., Cambridge University Press, USA, 2007.

[30] B. Schneier, P. Sutherland, Applied Cryptography : Protocols, Algorithms, and Source Code in C, 20e anniversaire, John Wiley & Sons, Inc., USA, 2015.

[31] W.E. Shotts, The Linux command line : a complete introduction, 2 éd., No Starch Press, USA, 2012.

[32] A. Silberschatz, P.B. Galvin, G. Gagne, Operating System Concepts, 10 éd., Wiley, 2018.

[33] A. Singh, Mac OS X Internals : A System Approach, Addison-Wesley Professional, 2006.

[34] W.R. Stevens, S.A. Rago, Advanced Programming in the UNIX Environment, 3 éd., Addison-Wesley Professional, 2013.

[35] A.S. Tanenbaum, H. Bos, Modern Operating Systems, 5 éd., Pearson, 2023.

[36] A.S. Tanenbaum, D.J. Wetherall, Computer Networks, 6 éd., Prentice Hall Press, USA, 2021.

[37] R. Tarjan, Depth-first search and linear graph algorithms, in : 12th Annual Symposium on Switching and Automata Theory (swat 1971), 1971 : p. 114 à 121.

[38] The kernel development community, The Linux Kernel, (2024).

[39] The Linux Foundation, Linux Standard Base, (2024).

[40] W3Techs, Usage Statistics and Market Share of Unix for Websites, (2024).

[41] D.A. Wheeler, SLOCCount, (2024).

[42] M.J. Zaki, W.M. Jr, Data Mining and Analysis : Fundamental Concepts and Algorithms, 2 éd., Cambridge University Press, USA, 2020.