Linux

Programmation système et réseau

Joëlle Delacroix
Maître de conférences en informatique,
Chef de département pour le DUT informatique,
Cnam de Paris

4e édition

DUNOD

Cet ouvrage est une nouvelle présentation de la quatrième édition
parue initialement dans la collection Sciences Sup en 2012.

Illustration de couverture :
Fun penguins © ArchMen - Fotolia.com

5 rue Laromiguière, 75005 Paris
www.dunod.com

ISBN 978-2-10-074855-6

TABLE DES MATIÈRES

AVANT-PROPOS

Cet ouvrage présente les principes fondamentaux des systèmes d'exploitation, illustrés sous le système Linux. Chaque chapitre est ainsi composé d'une partie théorique qui présente les concepts importants liés à la fonction du système d'exploitation étudiée, puis d'une partie applicative basée sur Linux et qui décrit de manière simplifiée[1] l'implémentation faite de ces concepts au sein du noyau Linux, ainsi que les primitives systèmes qui leur sont attachés. Des exemples de programmation illustrent l'emploi de ces primitives. Des exercices corrigés clôturent chaque chapitre et des énoncés de programmation sont suggérés.

Cet ouvrage s'articule autour de 9 chapitres qui décrivent l'ensemble des fonctions d'un système d'exploitation multiprogrammé tel que Linux :

- le chapitre 1 introduit les notions de base sur le fonctionnement d'un noyau tel que Linux ;
- le chapitre 2 traite des notions de processus et d'ordonnancement ;
- les chapitres 3 et 4 présentent le système de gestion de fichiers et le mécanisme des entrées-sorties ;
- le chapitre 5 a trait à la gestion de la mémoire centrale, notamment à la pagination et à la mémoire virtuelle ;
- les chapitres 6 à 9 présentent divers outils de communication et de synchronisation tels que les sockets, les tubes, les files de messages, les signaux, les régions de mémoires partagées et les sémaphores ;
- le chapitre 10 présente des notions relatives aux systèmes temps réel et multiprocesseurs et notamment aux systèmes de type Linux.

1. Bien que le code du noyau Linux soit disponible, nous avons choisi de ne pas décrire trop en détails les choix d'implémentation du noyau Linux. Les lecteurs désireux de pénétrer avec précision les arcanes du code du noyau peuvent se reporter à l'ouvrage suivant : Daniel P. Bovet, Marco Cesati, le noyau Linux, O'Reilly.

1 INTRODUCTION AU SYSTÈME LINUX

PLAN

OBJECTIFS

- ➤ Ce chapitre constitue une introduction au système d'exploitation Linux et plus généralement aux systèmes d'exploitation multiprogrammés, c'est-à-dire au système d'exploitation admettant plusieurs programmes exécutables en mémoire centrale.
- ➤ Après avoir défini ce qu'est un système d'exploitation et présenté sa structure générale, nous effectuons une présentation rapide du système Linux et de ses principales propriétés.
- ➤ Nous terminons ce chapitre en expliquant les notions de base sur lesquelles le fonctionnement d'un système d'exploitation s'appuie.

1.1 LE SYSTÈME D'EXPLOITATION : PRÉSENTATION GÉNÉRALE

1.1.1 Définition

Le *système d'exploitation* est un ensemble de programmes qui réalise l'interface entre le matériel de l'ordinateur et les utilisateurs (figure 1.1). Il a deux objectifs principaux :

- prise en charge de la gestion de plus en plus complexe des ressources et partage de celles-ci ;
- construction au-dessus du matériel d'une machine virtuelle plus facile d'emploi et plus conviviale.

La machine physique et ses différents composants, s'ils offrent des mécanismes permettant de faciliter leur partage entre différents programmes, ne sont malgré tout pas conçus pour supporter et gérer d'eux-mêmes ce partage. C'est là le premier rôle du système d'exploitation dans un environnement multiprogrammé que de gérer le partage de la machine physique et des ressources matérielles entre les différents

programmes. Cette gestion doit assurer l'équité d'accès aux ressources matérielles et assurer également que les accès des programmes à ces ressources s'effectuent correctement, c'est-à-dire que les opérations réalisées par les programmes sont licites pour la cohérence des ressources : on parle alors de *protection des ressources*.

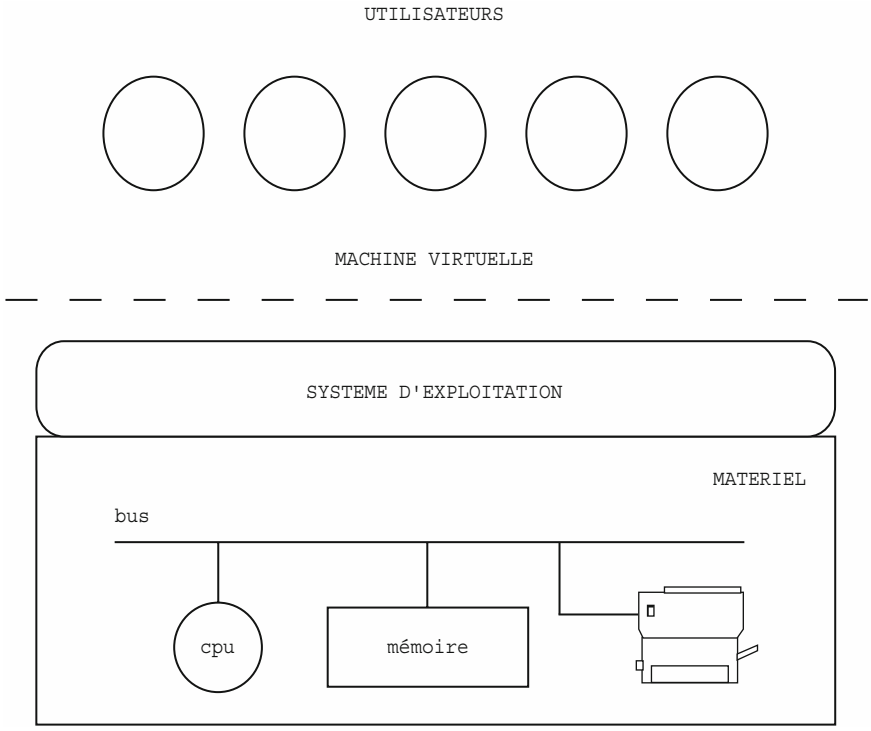

Figure 1.1 – **Place du système d'exploitation**

Le partage des ressources va concerner principalement le processeur, la mémoire centrale et les périphériques d'entrées-sorties. Plus précisément, les questions suivantes vont devoir être résolues :

- Dans le cadre du partage du processeur : parmi tous les programmes chargés en mémoire centrale, lequel doit s'exécuter ?
- Dans le cadre du partage de la mémoire centrale : comment allouer la mémoire centrale aux différents programmes ? Comment disposer d'une quantité suffisante de mémoire pour y placer tous les programmes nécessaires à un bon taux d'utilisation du processeur ? Comment assurer la protection entre ces différents programmes utilisateurs ? Par protection, on entend ici veiller à ce qu'un programme donné n'accède pas à une plage mémoire allouée à un autre programme.
- Dans le cadre du partage des périphériques : dans quel ordre traiter les requêtes d'entrées-sorties pour optimiser les transferts ?

Faciliter l'accès à la machine physique constitue le second rôle du système d'exploitation. Par le biais d'une interface de haut niveau, composée d'un ensemble de primitives attachées à des fonctionnalités qui gèrent elles-mêmes les caractéristiques matérielles sous-jacentes et offrent un service à l'utilisateur, le système d'exploitation construit au-dessus de la machine physique, une machine virtuelle plus simple d'emploi et plus conviviale. Ainsi, pour réaliser une opération d'entrées-sorties, l'utilisateur fait appel à une même primitive `ECRIRE(données)` quel que soit le périphérique concerné. C'est la primitive `ECRIRE` et la fonction de gestion des entrées-sorties du système d'exploitation à laquelle cette primitive est rattachée qui feront la liaison avec les caractéristiques matérielles.

Comme son nom le suggère, le système d'exploitation a en charge l'exploitation de la machine pour en faciliter l'accès, le partage et pour l'optimiser.

1.1.2 Structure générale

Le système d'exploitation réalise donc une couche logicielle placée entre la machine matérielle et les applications. Le système d'exploitation peut être découpé en plusieurs grandes fonctionnalités présentées sur la figure 1.2. Dans une première approche, ces fonctionnalités qui seront étudiées plus en détail dans les chapitres suivants de l'ouvrage sont :

- La fonctionnalité de gestion du processeur : le système doit gérer l'allocation du processeur aux différents programmes pouvant s'exécuter. Cette allocation se fait par le biais d'un *algorithme d'ordonnancement* qui planifie l'exécution des programmes. Dans ce cadre, une exécution de programme est appelée *processus* (*cf.* chapitre 2).
- La fonctionnalité de gestion des objets externes : la mémoire centrale est une mémoire volatile. Aussi, toutes les données devant être conservées au-delà de l'arrêt de la machine, doivent être stockées sur une mémoire de masse non volatile (disque dur, disquette, cédérom…). La gestion de l'allocation des mémoires de masse ainsi que l'accès aux données stockées s'appuient sur la notion de *fichiers* et de *système de gestion de fichiers* (SGF) (*cf.* chapitre 3).
- La fonctionnalité de gestion des entrées-sorties : le système doit gérer l'accès aux périphériques, c'est-à-dire faire la liaison entre les appels de haut niveau des programmes utilisateurs (exemple `getchar()`) et les opérations de bas niveau de l'unité d'échange responsable du périphérique (unité d'échange clavier) : c'est le pilote d'entrées-sorties (*driver*) qui assure cette correspondance (*cf.* chapitre 4).
- La fonctionnalité de gestion de la mémoire : le système doit gérer l'allocation de la mémoire centrale entre les différents programmes pouvant s'exécuter, c'est-à-dire qu'il doit trouver une place libre suffisante en mémoire centrale pour que le chargeur puisse y placer un programme à exécuter, en s'appuyant sur les mécanismes matériels sous-jacents de segmentation et de pagination. Comme la mémoire physique est souvent trop petite pour contenir la totalité des programmes, la gestion de la mémoire se fait selon le principe de la *mémoire*

virtuelle : à un instant donné, seules sont chargées en mémoire centrale, les parties de code et données utiles à l'exécution (*cf.* chapitre 5).

- La fonctionnalité de gestion de la concurrence : comme plusieurs programmes coexistent en mémoire centrale, ceux-ci peuvent vouloir communiquer pour échanger des données. Par ailleurs, il faut synchroniser l'accès aux données partagées afin de maintenir leur cohérence. Le système offre des *outils de communication et de synchronisation* entre processus (*cf.* chapitre 6, 7 et 8).
- La fonctionnalité de gestion de la *protection* : le système doit fournir des mécanismes garantissant que ses ressources (processeur, mémoire, fichiers) ne peuvent être utilisées que par les programmes auxquels les droits nécessaires ont été accordés. Il faut notamment protéger le système et la machine des programmes utilisateurs (mode d'exécution utilisateur et superviseur).
- La fonctionnalité d'accès au réseau : des exécutions de programmes placées sur des machines physiques distinctes doivent pouvoir échanger des données. Le système d'exploitation fournit des outils permettant aux applications distantes de dialoguer à travers une couche de protocoles réseau telle que TCP/IP (*cf.* chapitre 9).

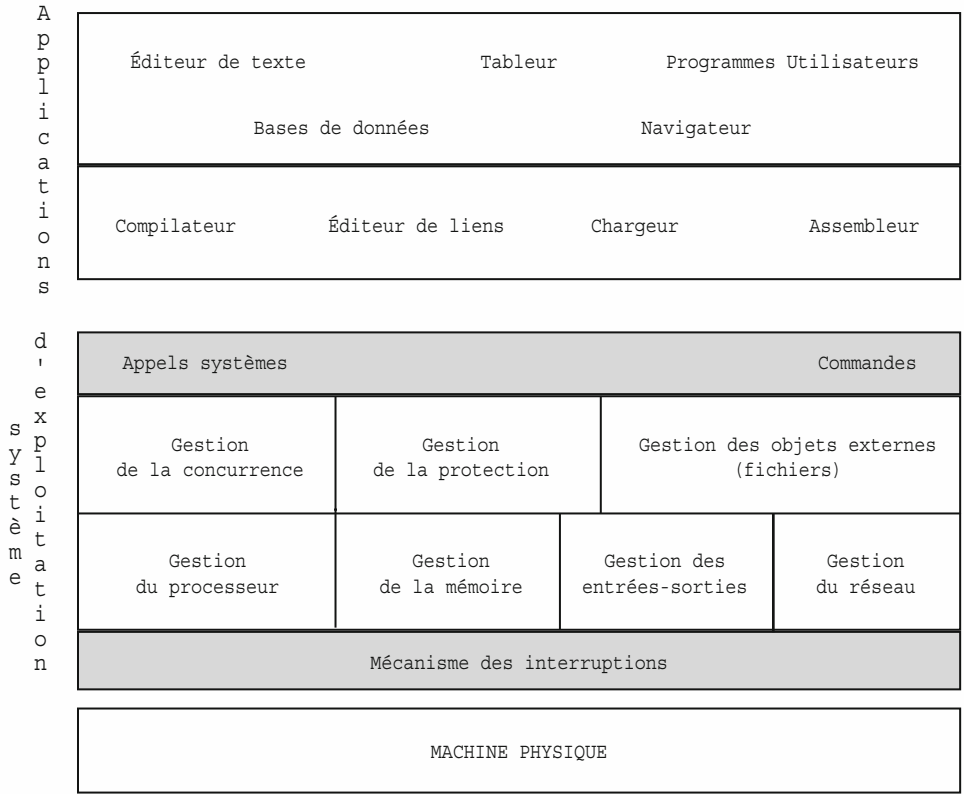

Figure 1.2 – Fonctionnalités du système d'exploitation

Les fonctionnalités du système d'exploitation utilisent les mécanismes offerts par le matériel de la machine physique pour réaliser leurs opérations. Notamment, le système d'exploitation s'interface avec la couche matérielle, par le biais du mécanisme des interruptions, qui lui permet de prendre connaissance des événements survenant sur la machine matérielle.

Par ailleurs, le système d'exploitation s'interface avec les applications du niveau utilisateur par le biais des fonctions prédéfinies que chacune de ses fonctionnalités offre. Ces fonctions que l'on qualifie de routines systèmes constituent les points d'entrées des fonctionnalités du système d'exploitation et sont appelables depuis les applications de niveau utilisateur. Ces appels peuvent se faire à deux niveaux :

- dans le code d'un programme utilisateur à l'aide d'un *appel système*, qui n'est autre qu'une forme d'appel de procédure amenant à l'exécution d'une routine système ;
- depuis le prompt de l'interpréteur de commandes à l'aide d'une *commande*. L'*interpréteur de commandes* est un outil de niveau utilisateur qui accepte les commandes de l'utilisateur, les analyse et lance l'exécution de la routine système associée.

1.1.3 Types de systèmes d'exploitation

Les systèmes d'exploitation multiprogrammés peuvent être classés selon différents types qui dépendent des buts et des services offerts par les systèmes. Trois grandes classes de systèmes peuvent être définies :

- les systèmes à traitements par lots ;
- les systèmes multi-utilisateurs interactifs ;
- les systèmes temps réels.

a) Systèmes à traitements par lots

Les *systèmes à traitement par lots* ou *systèmes batch* constituent en quelque sorte les ancêtres de tous les systèmes d'exploitation. Ils sont nés de l'introduction sur les toutes premières machines de deux programmes permettant une exploitation plus rapide et plus rentable du processeur, en vue d'automatiser les tâches de préparation des travaux à exécuter. Ces deux programmes sont d'une part le chargeur dont le rôle a été initialement de charger automatiquement les programmes dans la mémoire centrale de la machine depuis les cartes perforées ou le dérouleur de bandes et d'autre part, le moniteur d'enchaînement de traitements, dont le rôle a été de permettre l'enchaînement automatique des travaux soumis en lieu et place de l'opérateur de la machine.

Le principe du traitement par lots s'appuie sur la composition de lots de travaux ayant des caractéristiques ou des besoins communs, la formation de ces lots visant à réduire les temps d'attente du processeur en faisant exécuter les uns à la suite des autres ou ensemble, des travaux nécessitant les mêmes ressources.

Par ailleurs, dans un système à traitements par lots, la caractéristique principale est qu'il n'y a pas d'interaction possible entre l'utilisateur et la machine durant

l'exécution du programme soumis. Le programme est soumis avec ses données d'entrées et l'utilisateur récupère les résultats de l'exécution ultérieurement, soit dans un fichier, soit sous forme d'une impression. Là encore, ce mode de fonctionnement est directement influencé par le mode de travail des premiers temps de l'informatique : le programmeur soumettait son programme sous forme de cartes perforées avec ses données à l'opérateur de la machine, qui faisait exécuter le programme et rendait les résultats de l'exécution ensuite au programmeur.

L'objectif poursuivi dans les systèmes à traitements pas lots est de maximiser l'utilisation du processeur et le débit des travaux, c'est-à-dire le nombre de travaux traités sur une tranche de temps.

b) Systèmes interactifs

La particularité d'un système d'exploitation interactif est qu'au contraire des systèmes précédents, l'utilisateur de la machine peut interagir avec l'exécution de son programme. Typiquement, l'utilisateur lance l'exécution de son travail et attend derrière le clavier et l'écran, le résultat de celle-ci. S'il s'aperçoit que l'exécution n'est pas conforme à son espérance, il peut immédiatement agir pour arrêter celle-ci et analyser les raisons de l'échec.

Puisque l'utilisateur attend derrière son clavier et son écran et que par nature, l'utilisateur de la machine est un être impatient, le but principal poursuivi par les systèmes interactifs va être d'offrir pour chaque exécution le plus petit temps de réponse possible. Le temps de réponse d'une exécution est le temps écoulé entre le moment où l'exécution est demandée par l'utilisateur et le moment où l'exécution est achevée. Pour parvenir à ce but, la plupart des systèmes interactifs travaillent en temps partagé.

Un système en temps partagé permet aux différents utilisateurs de partager l'ordinateur simultanément, tout en ayant par ailleurs la sensation d'être seul à utiliser la machine. Ce principe repose notamment sur un partage de l'utilisation du processeur par les différents programmes des différents utilisateurs. Chaque programme occupe à tour de rôle le processeur pour un court laps de temps (*le quantum*) et les exécutions se succèdent suffisamment rapidement sur le processeur pour que l'utilisateur ait l'impression que son travail dispose seul du processeur.

c) Systèmes temps réel

Les systèmes temps réel (figure 1.3) sont des systèmes liés au contrôle de procédé pour lesquels la caractéristique primordiale est que les exécutions de programmes sont soumises à des contraintes temporelles, c'est-à-dire qu'une exécution de programme est notamment qualifiée par une date butoir de fin d'exécution, appelée *échéance*, au-delà de laquelle les résultats de l'exécution ne sont plus valides. Des exemples de système temps réel sont par exemple le pilotage automatique d'un train tel que EOLE ou encore par le dispositif de surveillance d'un réacteur de centrale nucléaire. Le procédé désigne ici soit le train EOLE, soit le réacteur nucléaire. Dans le cas même de EOLE, le système temps réel sera qualifié de système embarqué, puisqu'il est installé à bord du train. Dans de nombreux cas par ailleurs, le système

temps réel est également qualifié de système réactif, car ce système reçoit des informations du procédé auquel il se doit de réagir dans les temps impartis. Dans le cas de EOLE, par exemple, le système recevra des informations relatives à l'approche d'une station et devra entreprendre des actions de freinage pour être à même de s'arrêter à temps.

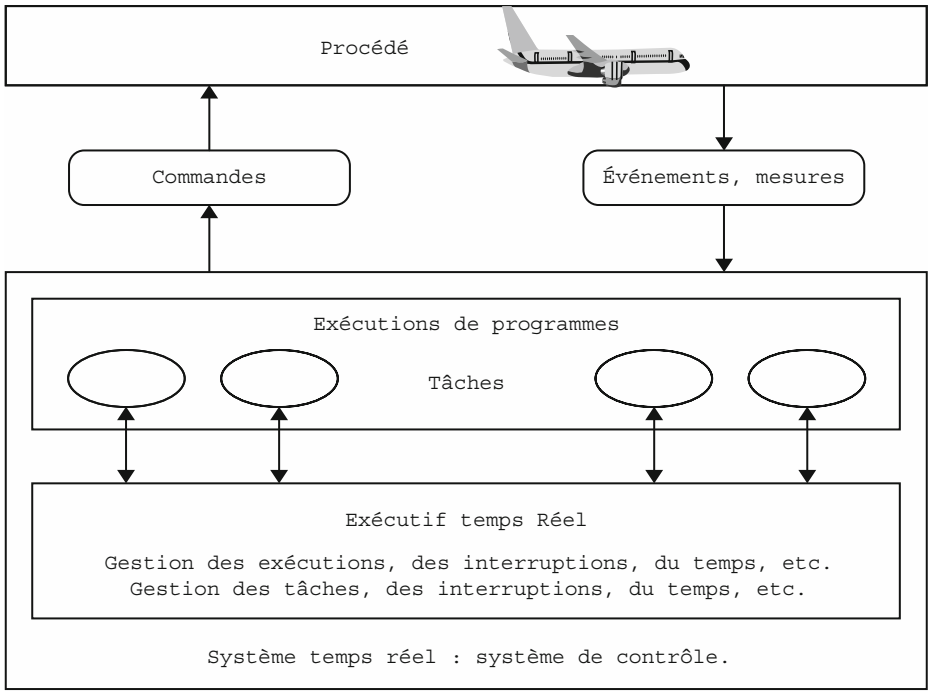

Figure 1.3 – Système temps réel

Il ne s'agit pas de rendre le résultat le plus vite possible, mais simplement à temps. L'échelle du temps relative à la contrainte temporelle varie d'une application à l'autre : elle peut être par exemple de l'ordre de la microseconde dans des applications de contrôle radars, mais peut être de l'ordre de l'heure pour une application de contrôle chimique. Par contre, il est souvent primordial de respecter la contrainte temporelle, sous peine de graves défaillances, pouvant mettre en péril le procédé lui-même et son environnement. Tout retard de réaction vis-à-vis d'une situation anormale au sein du réacteur nucléaire, peut évidemment mener à une situation catastrophique !

On distingue deux types de contraintes temporelles :

- les contraintes temporelles strictes (*temps réel strict ou dur*) : les fautes temporelles (non respect d'une contrainte temporelle) sont intolérables pour la validité du système. Elles mettent en péril le système temps réel lui-même voire son environnement. Par exemple, dans un système de lancement de missiles, si le lancement du missile est retardé, la cible risque d'être ratée ;

- les contraintes temporelles relatives (*temps réel relatif ou mou*) : quelques fautes temporelles peuvent être supportées sans remettre en cause la validité du système. Il s'ensuit généralement une baisse de performance du système. Par exemple, dans un système audio, si un paquet de données est perdu, la qualité du son sera dégradée, mais il restera audible.

Pour être en mesure de respecter les contraintes temporelles associées aux exécutions de programmes, le système temps réel doit offrir un certain nombre de méca-nismes particuliers, dont le but est de réduire au maximum tout indéterminisme au niveau des durées des exécutions et de garantir par là même que les contraintes temporelles seront respectées. Par exemple, le processeur sera de préférence alloué à l'exécution la plus urgente et les choix de conception du système lui-même seront tels que les ressources physiques comme la mémoire ou encore les périphériques seront gérés avec des méthodes simples, limitant les fluctuations et les attentes indéterminées.

Le système temps réel est souvent qualifié d'*exécutif temps réel*.

Le chapitre 10 présente plus en détail ce type de systèmes.

1.2 Le système Linux

1.2.1 Présentation générale

Le système d'exploitation Linux est un système multiprogrammé, compatible avec la norme pour les systèmes d'exploitation POSIX 1003.1, appartenant à la grande famille des systèmes de type Unix. C'est un système de type interactif qui présente également des aspects compatibles avec la problématique du temps réel faiblement contraint.

Le système Linux est né en 1991 du travail de développement de Linus Torvalds. Initialement créé pour s'exécuter sur des plates-formes matérielles de type IBM/PC, Intel 386, le système est à présent disponible sur des architectures matérielles très diverses telles que SPARC, Alpha, IBMSystem/390, Motorola, etc.

Une des caractéristiques principales de Linux est le libre accès à son code source puisque celui-ci a été déposé sous *Licence Publique GNU*. Le code source peut ainsi être téléchargé, étudié et modifié par toute personne désireuse de le faire.

Le noyau Linux vise à être compatible avec la norme pour les systèmes d'exploitation ouverts POSIX de l'IEEE. Ainsi, la plupart des programmes Unix existants peuvent être exécutés sur un système Linux sans ou avec fort peu de modifications.

1.2.2 Structure

Le système Linux est structuré comme un noyau monolithique qui peut être découpé en quatre grandes fonctions :

- une fonctionnalité de gestion des processus qui administre les exécutions de programmes, le *processus* étant l'image dynamique de l'exécution d'un programme (*cf.* chapitre 2) ;

- une fonctionnalité de gestion de la mémoire ;
- une fonctionnalité de gestion des fichiers de haut niveau, le VFS (*Virtual File System*) qui s'interface avec des gestionnaires de fichiers plus spécifiques de type Unix, DOS, etc., lesquels s'interfacent eux-mêmes avec les contrôleurs de disques, disquettes, CD-Rom, etc. ;
- une fonctionnalité de gestion du réseau qui s'interface avec le gestionnaire de protocole puis le contrôleur de la carte réseau.

L'architecture du système Linux peut être ajustée autour du noyau grâce au concept de *modules chargeables*. Un module chargeable correspond à une partie de noyau qui n'est pas indispensable à sa bonne marche et qui peut être ajouté ou ôté en fonction des besoins. Typiquement, un module correspond à un gestionnaire de périphérique, par exemple le pilote d'une carte réseau, un système de gestion de fichiers particulier, plus généralement toute fonction de haut niveau dont la présence n'est pas vitale pour le bon fonctionnement du noyau. Cette notion de modules permet de moduler la taille du noyau final.

Un module, une fois compilé, peut être ajouté au noyau Linux soit de manière statique, soit de manière dynamique. L'ajout statique s'effectue une fois pour toutes à la compilation du noyau. L'ajout dynamique s'effectue au cours de la vie du système. Il suffit pour cela d'avoir configuré le noyau de manière à ce qu'il accepte de tels ajouts. Deux commandes permettent alors de charger un module (`insmod « nom_module »`) et de décharger un module (`rmmod « nom_module »`).

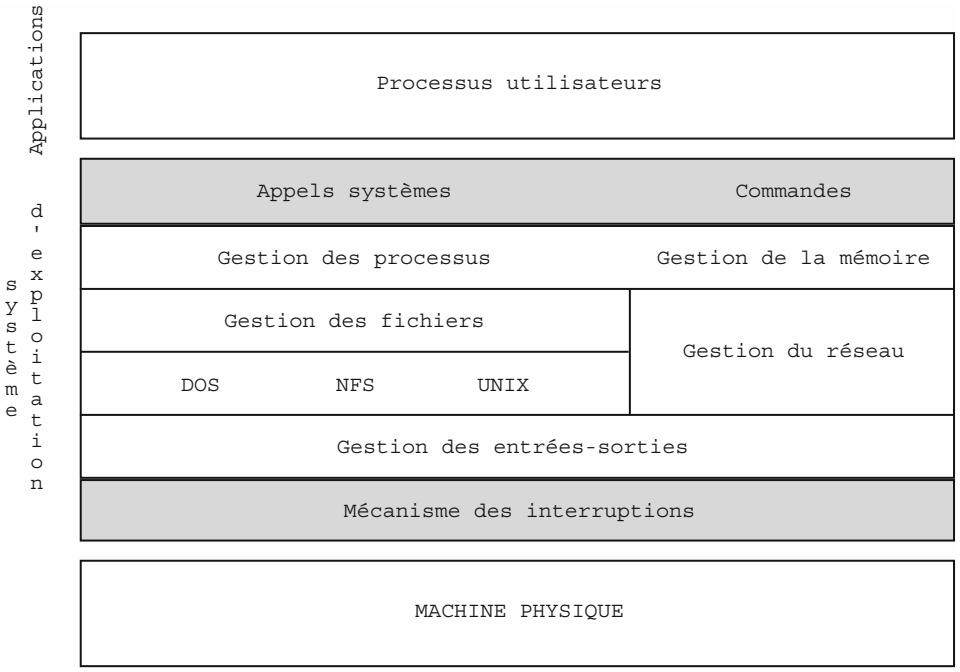

Figure 1.4 – Structure du noyau Linux

1.3 Notions fondamentales

Comme nous l'avons déjà évoqué précédemment, le système d'exploitation s'interface avec les applications du niveau utilisateur par le biais des fonctions prédéfinies qualifiées de routines systèmes et qui constituent les points d'entrées des fonctionnalités du système. À ce titre, le noyau Linux ne doit pas être appréhendé comme étant un processus, mais *un gestionnaire de processus*, qui offrent des services à ceux-ci. Ces services peuvent être appelés de deux façons par les processus utilisateurs, soit par le biais d'un *appel système*, soit par le biais d'une *commande du langage de commandes*.

L'exécution des routines systèmes s'effectue sous un mode privilégié, appelé *mode superviseur* ou *mode maître*.

Le noyau Linux comporte cependant quelques processus privilégiés appelés *threads noyaux* qui sont créés à l'initialisation du système et existent tout au long de la vie de celui-ci. Ces threads s'exécutent exclusivement en mode privilégié et réalisent des tâches spécifiques de gestion de la machine telles que libérer de l'espace en mémoire centrale ou bien encore vider les caches du système.

Nous abordons paragraphe 1.3.1 la définition de notions générales concernant les systèmes d'exploitation et illustrons plus particulièrement celles-ci en étudiant leur implémentation au sein du système Linux, assis une architecture de type Intel x86.

1.3.1 Modes d'exécutions et commutations de contexte

a) Modes d'exécutions

Un processus utilisateur s'exécute par défaut selon un mode qualifié de *mode esclave* ou *mode utilisateur* (*User Mode*) : ce mode d'exécution est un mode pour lequel les actions pouvant être entreprises par le programme sont volontairement restreintes afin de protéger la machine des actions parfois malencontreuses du programmeur. Notamment, le jeu d'instructions utilisables par le programme en mode utilisateur est réduit, et spécialement les instructions permettant la manipulation des interruptions est interdite.

Le système d'exploitation, quant à lui, s'exécute dans un mode privilégié encore appelé *mode superviseur* (*Kernel Mode*), pour lequel aucune restriction de droits n'existe.

Le codage du mode esclave et du mode maître est réalisé au niveau du processeur, dans le registre d'état de celui-ci.

Le noyau Linux est un noyau non préemptible, c'est-à-dire qu'un processus utilisateur passé en mode superviseur pour exécuter une fonction du noyau ne peut pas être arbitrairement suspendu et remplacé par un autre processus. Ainsi, en mode superviseur, un processus libère toujours de lui-même le processeur.

Puisqu'un processus exécute soit son propre code en mode utilisateur, soit le code du système en mode superviseur, deux piles d'exécution lui ont été associées dans le système Linux : une pile utilisateur utilisée en mode utilisateur et une pile noyau utilisée en mode superviseur.

Modes d'exécutions du processeur de la famille Intel 80386

Les niveaux de privilège sont implémentés en attribuant une valeur de 0 à 3 aux objets reconnus par le processeur (segments, tables, etc.), le niveau 0 correspondant au niveau le plus privilégié et le niveau 3 au niveau le moins privilégié. Ces niveaux de privilège affectés aux objets définissent des règles d'accès aux objets ainsi qu'un ensemble d'instructions privilégiées ne pouvant être exécutées que lorsque le niveau courant du processeur est égal à 0. Le système Linux n'utilise que deux de ces niveaux : le niveau 0 correspond au niveau superviseur tandis que le niveau 3 correspond au niveau utilisateur. Les niveaux 1 et 2 ne sont pas pris en compte.

Les séquences d'exécutions réalisées en mode noyau sont appelées *chemins de contrôle du noyau*.

b) Commutations de contexte

Lorsqu'un processus utilisateur demande l'exécution d'une routine du système d'exploitation par le biais d'un appel système, ce processus quitte son mode courant d'exécution (le mode esclave) pour passer en mode d'exécution du système, soit le mode superviseur. Ce passage du mode utilisateur au mode superviseur constitue une *commutation de contexte* : elle s'accompagne d'une opération de *sauvegarde du contexte utilisateur*, c'est-à-dire principalement de la valeur des registres du processeur (compteur ordinal, registre d'état), sur la pile noyau. Un contexte noyau est chargé constitué d'une valeur de compteur ordinal correspondant à l'adresse de la fonction à exécuter dans le noyau, et d'un registre d'état en mode superviseur. Lorsque l'exécution de la fonction système est achevée, le processus repasse du mode superviseur au mode utilisateur. Il y a de nouveau une opération de commutation de contexte avec *restauration du contexte utilisateur* sauvegardé lors de l'appel système, ce qui permet de reprendre l'exécution du programme utilisateur juste après l'appel (figure 1.5).

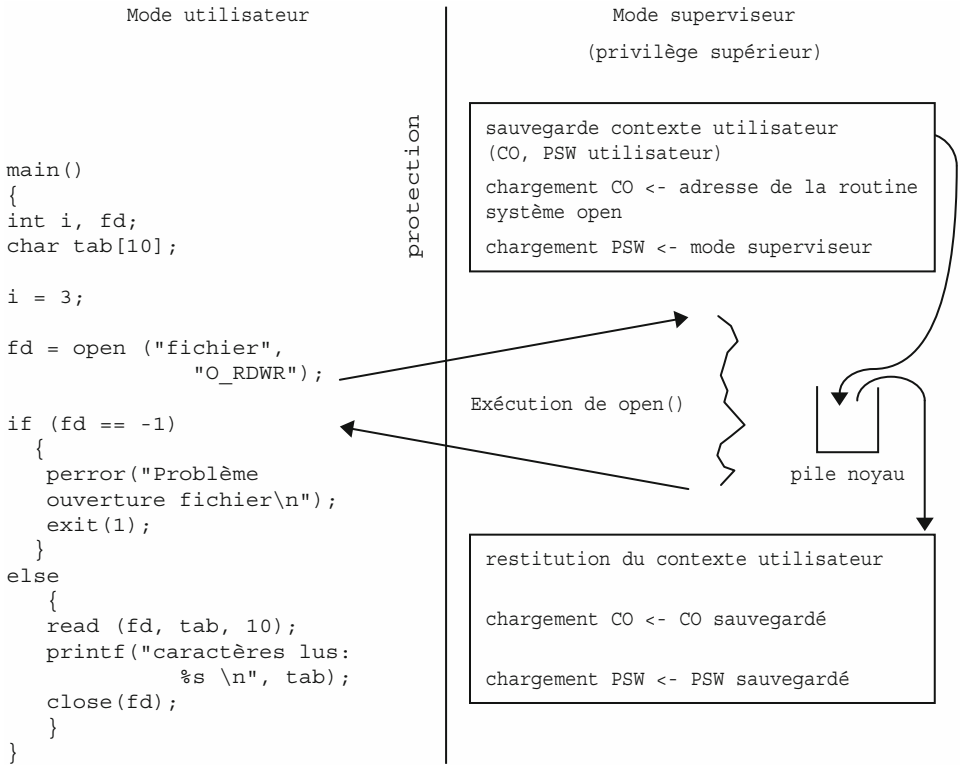

Figure 1.5 – Commutations de contexte

Trois causes majeures provoquent le passage du mode utilisateur au mode superviseur (figure 1.6) :

- le fait que le processus utilisateur appelle une fonction du système. C'est une demande explicite de passage en mode superviseur ;
- l'exécution par le processus utilisateur d'une opération illicite (division par 0, instruction machine interdite, violation mémoire…) : c'est la *trappe* ou l'*exception*. L'exécution du processus utilisateur est alors arrêtée ;
- la prise en compte d'une interruption par le matériel (IRQ *Interrupt Request*) et le système d'exploitation. Le processus utilisateur est alors stoppé et l'exécution de la routine d'interruption associée à l'interruption survenue est exécutée en mode superviseur.

Trappes et appels systèmes sont parfois qualifiés d'*interruptions logicielles ou synchrones* pour les opposer aux interruptions matérielles (IRQ) levées par les périphériques du processeur de manière totalement asynchrone par rapport à l'activité du processeur.

Par ailleurs, l'appel système constitue en réalité une trappe particulière qui n'aboutit pas à la terminaison du processus mais permet à celui-ci de basculer en mode superviseur pour aller exécuter une fonction du système.

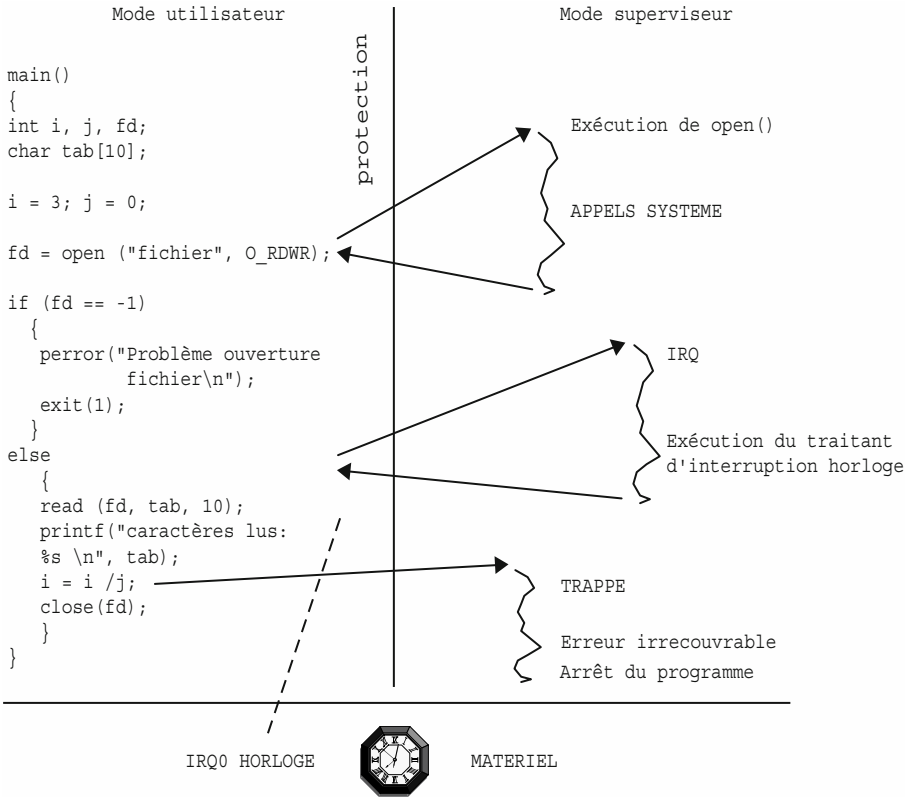

Figure 1.6 – Trois causes de commutations de contexte

Une autre trappe n'aboutissant pas à la terminaison du processus est la trappe levée en cas de défaut de page. Cette trappe initialise plutôt une opération d'entrées-sorties afin de charger la page manquante en mémoire centrale (*cf.* chapitre 5).

Par ailleurs, il faut noter que le noyau Linux étant un *noyau réentrant*, le traitement d'une interruption en mode noyau peut être interrompue au profit du traitement d'une autre interruption survenue entre-temps. De ce fait, les exécutions au sein du noyau peuvent être imbriquées les unes par rapport aux autres comme l'illustre la figure 1.7. Aussi, le noyau maintient-il une valeur de niveau d'imbrication, qui indique la profondeur d'imbrication courante dans les chemins de contrôle du noyau. Sur la figure 1.7, le niveau d'imbrication est de 2 lors du premier passage en mode noyau du fait de la prise en compte de l'interruption horloge durant l'exécution de l'appel système `open()`. Pour plus de détail voir § 1.3.2.

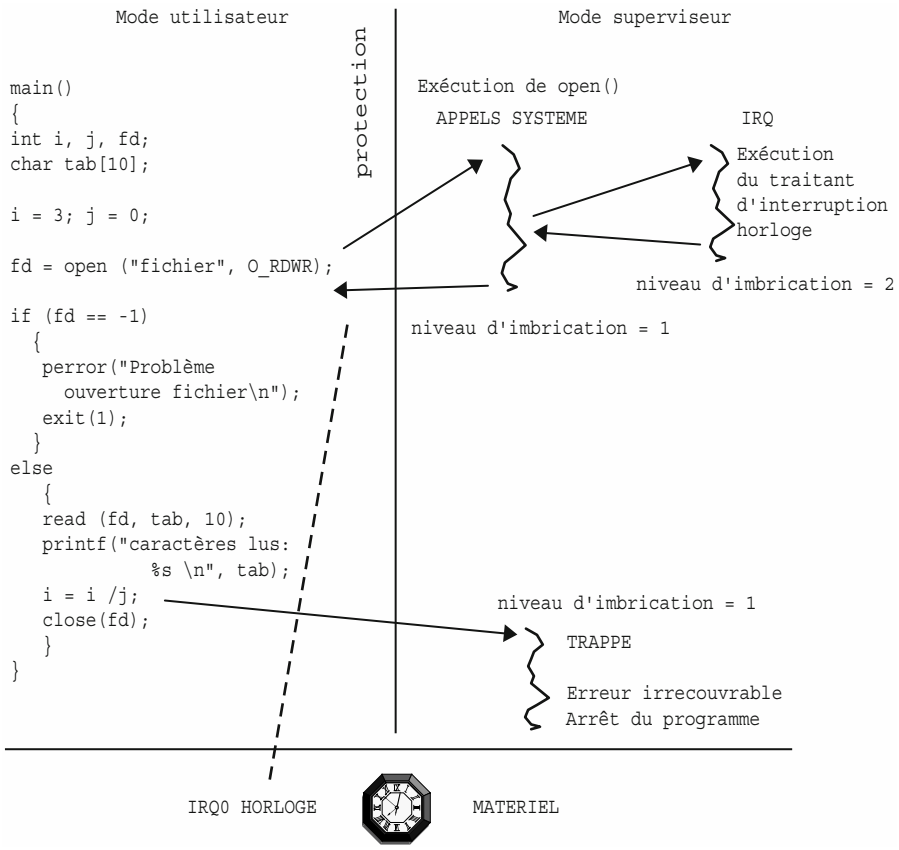

Figure 1.7 – Imbrication des traitements d'interruptions

1.3.2 Gestion des interruptions matérielles et logicielles

Sous le système Linux, chaque interruption, qu'elle soit matérielle ou logicielle, est identifiée par un entier de 8 bits appelé *vecteur d'interruption*, dont la valeur varie de 0 à 255 :

- les valeurs 0 à 31 correspondent aux interruptions non masquables et aux exceptions ;
- les valeurs 32 à 47 sont affectées aux interruptions masquables levées par les périphériques (IRQ) ;
- les valeurs 48 à 255 peuvent être utilisées pour identifier d'autres types de trappes que celles admises par le processeur et correspondant aux valeurs 0 à 31. Seule l'entrée 128 (0x80) est utilisée pour implanter les appels systèmes.

Ce numéro permet d'adresser une table comportant 256 entrées, appelée *table des vecteurs d'interruptions* (`idt_table`), placée en mémoire centrale au moment

du boot de la machine. Chaque entrée de la table contient l'adresse en mémoire centrale du gestionnaire chargé de la prise en compte de l'interruption.

Ainsi, l'unité de contrôle du processeur avant de commencer l'exécution d'une nouvelle instruction machine, vérifie si une interruption ne lui a pas été délivrée. Si tel est le cas, alors les étapes suivantes sont mises en œuvre :

- l'interruption i a été délivrée au processeur. L'unité de contrôle accède à l'entrée i de la table des vecteurs d'interruptions dont l'adresse est conservée dans un registre du processeur et récupère l'adresse en mémoire centrale du gestionnaire de l'interruption levée ;
- l'unité de contrôle vérifie que l'interruption a été émise par une source autorisée ;
- l'unité de contrôle effectue un changement de niveau d'exécution (passage en mode superviseur) si cela est nécessaire et commute de pile d'exécution. En effet, les exécutions des gestionnaires d'interruptions pouvant être imbriquées comme nous l'avons mentionné précédemment et comme nous le verrons au paragraphe 1.3.2, la prise en compte d'une interruption par l'unité de contrôle peut être faite alors que le mode d'exécution du processeur est déjà le mode superviseur (niveau d'imbrication supérieur à 1) ;
- l'unité de contrôle sauvegarde dans la pile noyau le contenu du registre d'état et du compteur ordinal ;
- l'unité de contrôle charge le compteur ordinal avec l'adresse du gestionnaire d'interruption.

Le gestionnaire d'interruption s'exécute. Cette exécution achevée, l'unité de contrôle restaure le contexte sauvegardé au moment de la prise en compte de l'interruption, c'est-à-dire :

- l'unité de contrôle restaure les registres d'état et compteur ordinal à partir des valeurs sauvegardées dans la pile noyau ;
- l'unité de contrôle commute de pile d'exécution pour revenir à la pile utilisateur si le niveau d'imbrication des exécutions du noyau est égal à 1.

Nous allons à présent examiner un peu plus en détail le fonctionnement des différents gestionnaires, pour les interruptions matérielles, pour les exceptions, enfin pour les appels systèmes.

a) Prise en compte d'une interruption matérielle

Contexte matériel

Une interruption matérielle est un signal permettant à un dispositif externe au processeur (un périphérique d'entrées-sorties notamment) d'interrompre le travail courant du processeur afin qu'il aille réaliser un traitement particulier lié à la cause de l'interruption, appelé *routine ou traitant d'interruption*.

Un processeur de la famille 80386 admet 15 niveaux d'interruptions matérielles de IRQ0 à IRQ15, ces 15 niveaux d'interruptions étant gérés par un circuit spécialisé, le contrôleur d'interruptions (figure 1.8).

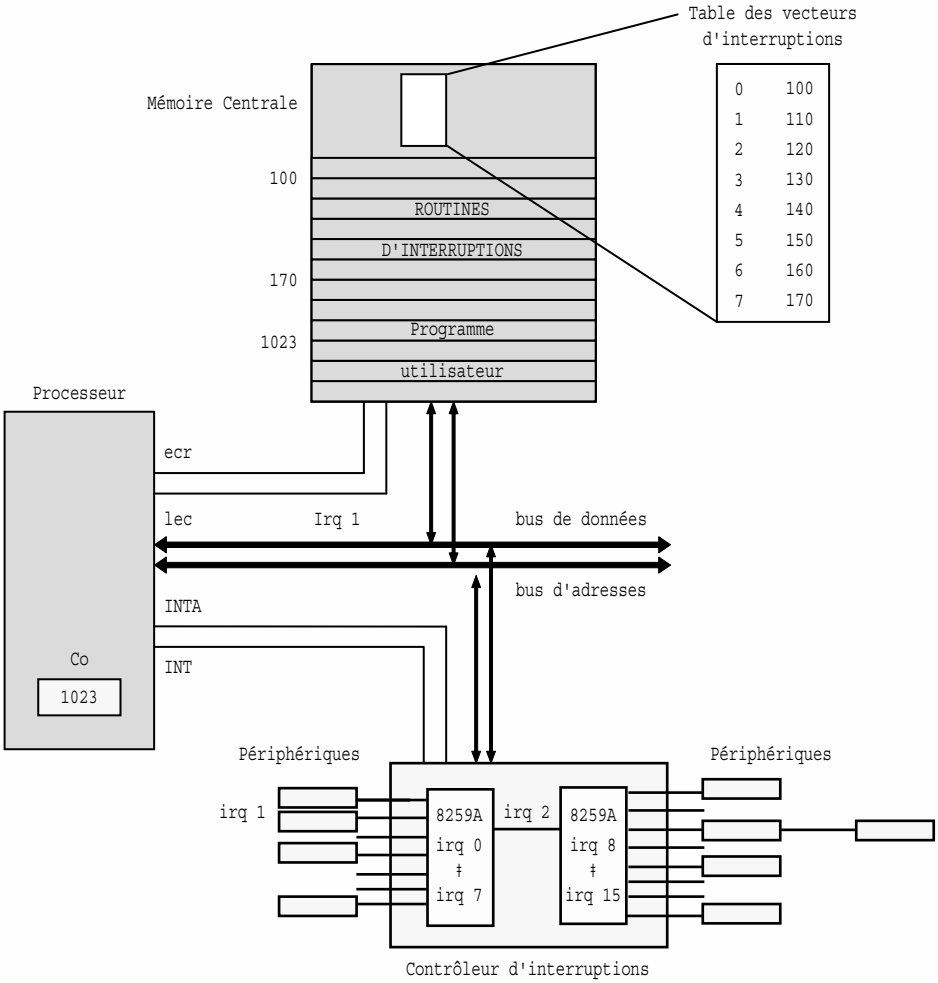

Figure 1.8 – Contrôleur d'interruptions

Physiquement, ce contrôleur d'interruptions est réalisé par deux circuits 8259A, connectés en cascade par l'intermédiaire de la broche IRQ2 du premier circuit. Les périphériques sont branchés à chaque niveau d'interruptions. Comme il n'existe que 15 niveaux d'interruptions, il peut arriver que plusieurs périphériques différents soient attachés à la même interruption. Dans ce cas, le contrôleur d'interruptions scrute chaque périphérique de la ligne d'interruption levée pour connaître le périphérique initiateur du signal.

Le contrôleur d'interruptions mémorise les requêtes d'interruptions levées par les périphériques et effectue un arbitrage entre celles-ci pour les délivrer sur la broche INT du processeur selon un ordre de priorité qui est tel que le niveau 0 est le plus prioritaire et le niveau 15 est le moins prioritaire.

Le protocole suivant est établi entre le contrôleur d'interruptions et le processeur pour la prise en compte d'une interruption :

- le contrôleur émet un signal sur la broche INT du processeur ;
- le processeur acquitte ce signal en envoyant un signal INTA ;
- le contrôleur place le numéro de l'interruption levée sur le bus de données ;
- le processeur (unité de contrôle) utilise ce numéro pour indexer la table des vecteurs d'interruptions et lancer l'exécution du gestionnaire de l'interruption matérielle selon la logique décrite précédemment.

Une interruption survenant à n'importe quel moment, il peut être souhaitable d'interdire sa prise en compte par le processeur, notamment lorsque celui-ci est en train d'exécuter du code noyau critique. Pour cela, les interruptions peuvent être *masquées* grâce à une instruction machine spécifique (instruction `cli`). Dans ce cas, les interruptions sont ignorées par le processeur jusqu'à ce que celui-ci *démasque* les interruptions par le biais d'une seconde instruction machine (instruction `sti`).

Prise en charge de l'interruption par le gestionnaire d'interruption matérielle

Le gestionnaire de l'interruption matérielle n(fonction `IRQn_interrupt()`) – n est le point d'entrée dans la table des vecteurs d'interruptions obtenu en translatant de 32 le numéro de l'interruption matérielle[1] – commence par effectuer une sauvegarde complémentaire des registres généraux sur la pile noyau. À l'issue de cette sauvegarde, le gestionnaire de l'interruption invoque le traitement même de l'interruption (fonction `do_IRQ()`). Comme l'interruption, si elle est partagée, peut avoir été émise par plusieurs périphériques différents, le traitant d'interruption `do_IRQ()` contient plusieurs routines de services (*ISR interrupt service routines*) spécifiques de chacun des périphériques concernés. Ces routines sont exécutées les unes à la suite des autres de façon à déterminer quel est le périphérique initiateur de l'interruption et l'ISR concernée est alors totalement exécutée (figure 1.9).

Les actions exécutées par une routine de service sont réparties selon 3 catégories :

- les actions critiques qui doivent être exécutées immédiatement, interruptions masquées ;
- les actions non critiques qui peuvent être exécutées rapidement, interruptions démasquées ;
- les actions pouvant être reportées. Ce sont des actions dont la durée d'exécution est longue et dont la réalisation n'est pas critique pour le fonctionnement du noyau. Pour cette raison, le système Linux choisit de reporter leur exécution en dehors de la routine de service, dans des fonctions appelées les « parties basses » (*bottom halves*) qui seront exécutées plus tard, juste avant de revenir en mode utilisateur. L'ISR se contente d'activer ces parties basses pour demander leur exécution ultérieure (fonction `mark_bh`).

1. Les interruptions numérotées de 0 à 15 au niveau du contrôleur d'interruptions correspondent aux entrées 32 à 47 de la table des vecteurs d'interruptions.

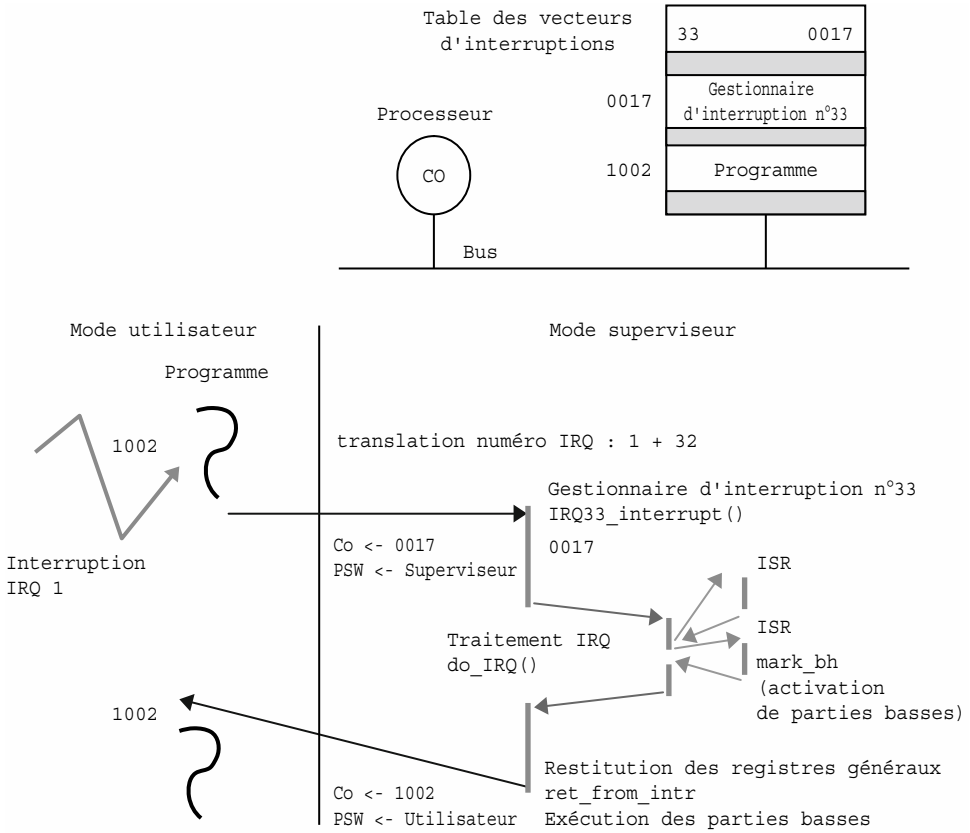

Figure 1.9 – Gestionnaire d'interruptions

Une fois les routines de services exécutées, le contexte sauvegardé au moment de la prise en compte de l'interruption est restitué, d'une part par le gestionnaire de l'interruption matérielle (restitution des registres généraux sauvegardés lors de son appel), d'autre part par l'unité de contrôle du processeur (restitution du registre d'état et du compteur ordinal). Si le niveau d'imbrication du noyau est égal à 1, alors le mode d'exécution bascule au niveau utilisateur et le programme utilisateur reprend son exécution. Avant ce retour en mode utilisateur, les parties basses activées par les routines de services sont exécutées.

Un exemple : le traitement de l'interruption horloge

Le dispositif d'horloge de la machine est lié à l'IRQ0 du contrôleur d'interruptions. Elle correspond à l'entrée 32 de la table des vecteurs d'interruptions pour le système Linux. L'interruption horloge est levée 100 fois par seconde, le temps étant mesuré en « tick » d'une valeur de 20 ms.

Les actions devant être réalisées lors de l'occurrence d'une interruption horloge sont les suivantes :

- mise à jour du temps écoulé depuis le démarrage du système ;
- mise à jour de la date et de l'heure ;
- mise à jour des statistiques d'utilisation des ressources ;
- calcul du temps durant lequel le processus s'est exécuté ;
- gestion des timers du système.

Le gestionnaire d'interruption horloge (fonction `timer_interrupt`) invoque la routine de service `do_timer()`. Cette fonction de service ne réalise que la première action de la liste ci-dessus. Les autres actions sont réalisées dans le cadre d'une fonction partie basse appelée `TIMER_BH` et exécutée avant le retour en mode utilisateur.

À titre indicatif, nous donnons un code simplifié des deux fonctions `do_timer()` et `timer_bh()` issu du fichier `/kernel/linux.c`.

```
void timer_bh(void)
{
  update_times(); /* mise à jour de l'heure système, des statistiques
d'utilisation des ressources et du temps d'exécution du processus courant
*/
  run_timer_list(); /* gestion des timers */
}

void do_timer(struct pt_regs *regs)
{
    (*(unsigned long *)&jiffies)++; /* incrémentation du temps écoulé
depuis le démarrage de la machine, la variable jiffies cumulant le nombre
de ticks écoulés */
     mark_bh(TIMER_BH); /* activation de la partie basse TIMER_BH
correspondant à l'exécution de la function timer_bh() avant le passage en
mode utilisateur */
}
```

b) Gestion d'une exception (trappe)

L'occurrence d'une trappe est levée lorsque le processeur rencontre une situation anormale en cours d'exécution, par exemple une division par 0 (trappe 0 sous Linux), un débordement (trappe 4 sous Linux), l'utilisation d'une instruction interdite (trappe 6 sous Linux) ou encore un accès mémoire interdit (trappe 10 sous Linux).

Le traitement d'une trappe est très comparable à celui des interruptions, évoqué au paragraphe précédent. La différence essentielle réside dans le fait que l'occurrence d'une trappe est synchrone avec l'exécution du programme.

Aussi, de la même manière que pour les interruptions, une trappe est caractérisée par un numéro et un gestionnaire de trappe lui est associé dont l'adresse en mémoire est rangée dans la table des vecteurs d'interruptions.

Le traitement associé à la trappe consiste à envoyer un signal au processus fautif (fonction `do_handler_name()` où `handler_name` est le nom de l'exception). La prise en compte de ce signal provoque par défaut l'arrêt du programme en cours d'exécution, mais le processus a la possibilité de spécifier un traitement particulier qui

remplace alors le traitement par défaut (*cf.* chapitre 5 sur les signaux). Dans le cas par défaut, une image mémoire de l'exécution arrêtée est créée qui peut être utilisée par le programmeur pour comprendre le problème survenu (figure 1.10).

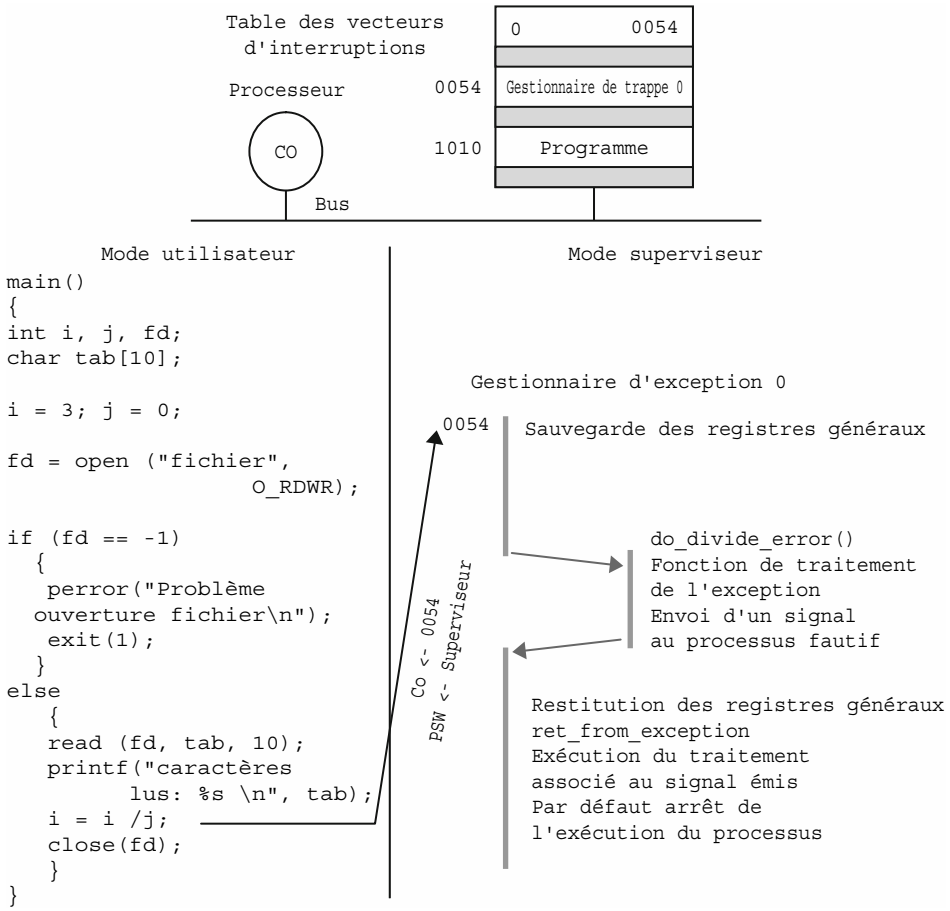

Figure 1.10 – Gestionnaire de trappes

Une trappe particulière est celle liée à la gestion des défauts de pages en mémoire centrale (trappe 14 sous Linux). Son occurrence n'entraîne pas la terminaison du processus associé, mais permet le chargement à la demande des pages constituant l'espace d'adressage du processus.

c) Exécution d'une fonction du système

L'appel à une fonction du système peut être réalisé soit depuis un programme utilisateur par le biais d'un appel de procédure qualifié dans ce cas d'*appel système*, soit par le biais d'une commande. La commande est prise en compte par un programme particulier, l'*interpréteur de commandes*, qui à son tour invoque la fonction système correspondante.

L'invocation d'une fonction système se traduit par la levée d'une interruption logicielle particulière, entraînant un changement de mode d'exécution et le branchement à l'adresse de la fonction.

Plus précisément, l'ensemble des appels système disponibles est regroupé dans une ou plusieurs bibliothèques, avec lesquelles l'éditeur de liens fait la liaison. Chaque fonction de la bibliothèque encore appelée *routine d'enveloppe* comprend une instruction permettant le changement de mode d'exécution, ainsi que des instructions assurant le passage des paramètres depuis le mode utilisateur vers le mode superviseur. Souvent, des registres généraux prédéfinis du processeur servent à ce transfert de paramètres. Enfin, la fonction lève une trappe en passant au système un numéro spécifique à la routine appelée. Le système cherche dans une table, l'adresse de la routine identifiée par le numéro et se branche sur son code. À la fin de l'exécution de la routine, la fonction de la bibliothèque retourne les résultats de l'exécution *via* les registres réservés à cet effet, puis restaure le contexte du programme utilisateur.

Sous le système Linux, la trappe levée par la routine d'enveloppe pour l'exécution d'une fonction du système porte le numéro 128. Les paramètres dont le nombre ne doit pas être supérieur à 5, sont passés *via* les registres ebx, ecx, edx, esi et edi du processeur x86. Le numéro de la routine système concernée est passé *via* le registre eax. Ce numéro sert d'index pour le gestionnaire des appels système dans une table des appels système, la table `sys_call_table`, comportant 256 entrées. Chaque entrée de la table contient l'adresse de la routine système correspondant à l'appel système demandé. Le gestionnaire d'appel système renvoie *via* le registre eax un code retour qui coïncide à une situation d'erreur si sa valeur est comprise entre – 1 et – 125. Dans ce cas, cette valeur est placée dans une variable globale `errno` et le registre eax est rempli avec la valeur – 1, ce qui indique pour le processus utilisateur une situation d'échec dans la réalisation de l'appel système. La cause de l'erreur peut être alors consultée dans la variable `errno` (figure 1.11) à l'aide d'un ensemble de fonctions prédéfinies :

```
#include <stdio.h>
#include <errno.h>
void perror(const char *s);
char *strerror (int errnum);
```

La fonction `perror()` affiche un message constitué de deux parties, d'une part la chaîne spécifiée par le paramètre `s`, d'autre part le message d'erreur correspondant à la valeur de la variable `errno`.

La fonction `strerror()` retourne le message d'erreur correspondant à la valeur de l'erreur spécifié par l'argument `errnum`.

L'ensemble des valeurs admises pour la variable `errno` est spécifié dans le fichier `errno.h`.

Ainsi le message d'erreur affiché par le code de la figure 1.11 est « Problème ouverture fichier : No such file or directory ».

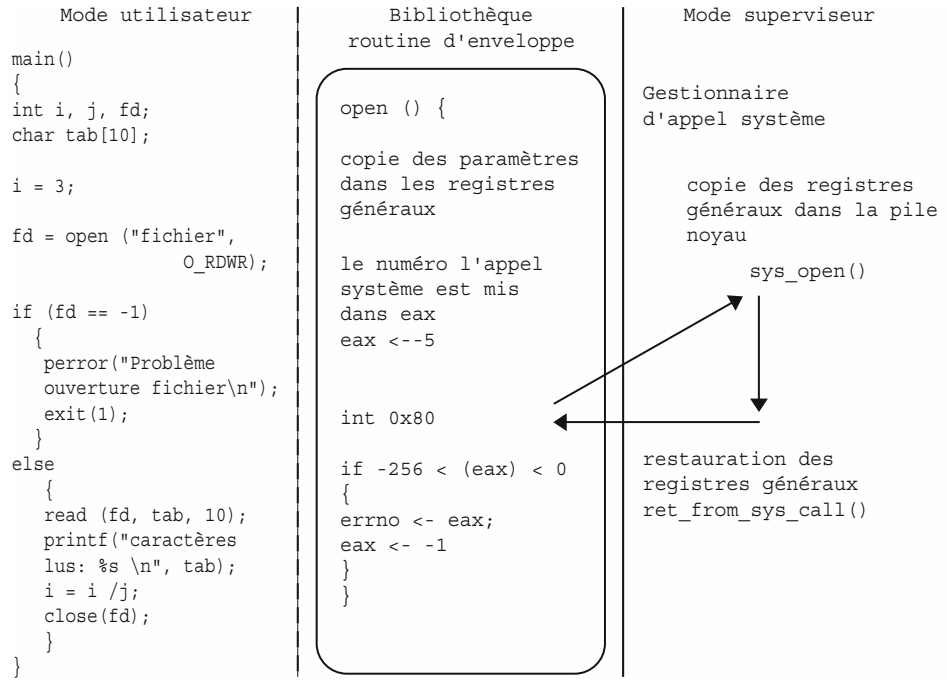

Figure 1.11 **Réalisation d'un appel système**

d) Imbrication de la prise en compte des interruptions

Le noyau Linux est un *noyau réentrant*, c'est-à-dire qu'à un instant *t* donné, plusieurs exécutions en mode noyau peuvent exister, une seule étant active. En effet, le noyau autorise un gestionnaire d'interruption à interrompre l'exécution d'un autre gestionnaire d'interruption. L'exécution du gestionnaire est suspendue, son contexte sauvegardé sur la pile noyau et l'exécution est reprise lorsque l'exécution du gestionnaire survenue entre-temps est achevée. De ce fait, les exécutions au sein du noyau peuvent être arbitrairement imbriquées.

Le noyau maintient une valeur de niveau d'imbrication, qui indique la profondeur d'imbrication courante dans les exécutions du noyau. Le retour en mode utilisateur n'est effectué que lorsque ce niveau d'imbrication est à 1. Il est précédé de l'exécution des parties basses activées par l'ensemble des routines de services exécutées. Plus précisément (figure 1.12) :

- un gestionnaire d'interruption peut interrompre un gestionnaire de trappe ou un autre gestionnaire d'interruption ;
- par contre, un gestionnaire de trappe ne peut jamais interrompre un gestionnaire d'interruption.

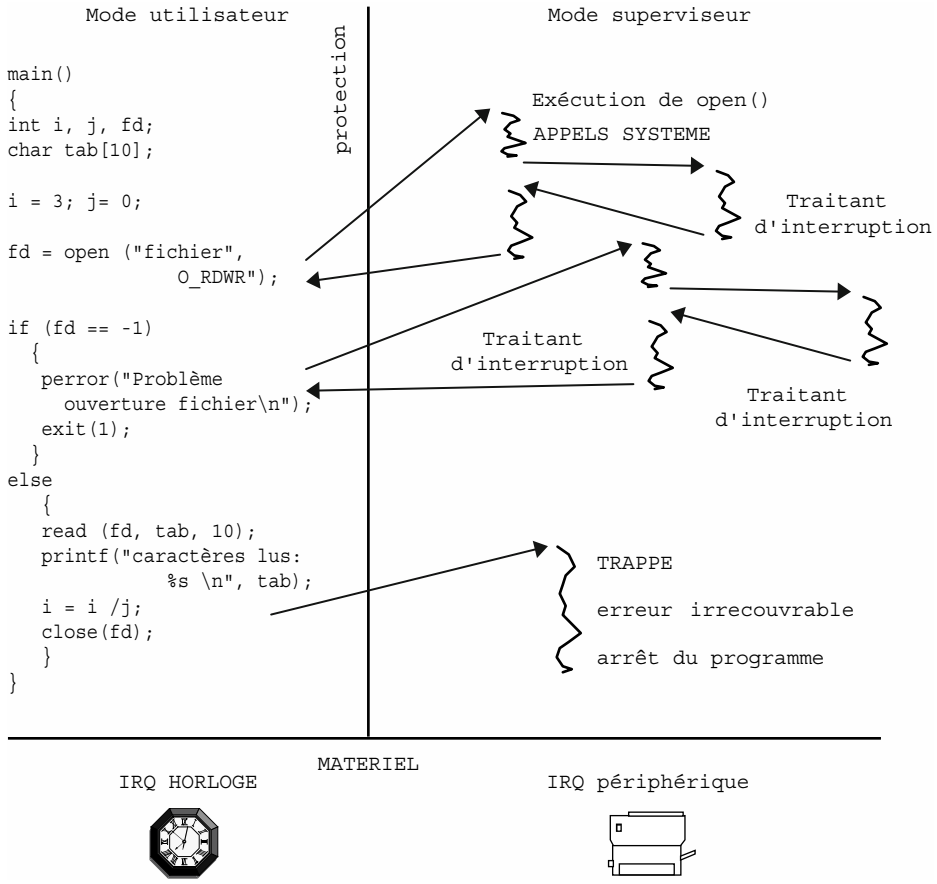

Figure 1.12 – Imbrication des chemins d'exécution au sein du noyau

Exercices

1.1 Qu'avez-vous retenu ?

Pour chacune des questions suivantes, choisissez la ou les réponses qui vous semblent exactes.

Question 1 – Quel est le rôle du système d'exploitation ?

- ☐ a. compiler un programme et construire un exécutable.
- ☐ b. gérer les accès disque.
- ☐ c. partager la machine physique entre les différents programmes et bâtir une machine virtuelle plus accessible à l'utilisateur.

Question 2 – Un système est dit réactif :

- ☐ a. car il doit offrir le plus petit temps de réponse possible à l'utilisateur.
- ☐ b. car il doit réagir à des événements délivrés par des périphériques externes.
- ☐ c. car il est rapide.

Question 3 – Une interruption est un signal dont le rôle est de :

- ☐ a. permettre de séquencer l'exécution des programmes.
- ☐ b. obliger le processeur à interrompre son traitement en cours pour traiter un événement survenu sur la machine.

Question 4 – Une commutation de contexte intervient :

- ☐ a. à chaque changement de mode d'exécution.
- ☐ b. sur l'occurrence d'une interruption.
- ☐ c. lorsque le programme commet une erreur grave.

Question 5 – Le système Linux est un gestionnaire de processus, ce qui veut dire que :

- ☐ a. le système offre des services aux processus utilisateurs sous la forme de fonctions que les processus utilisateurs exécutent eux-mêmes.
- ☐ b. le système est lui-même composé d'un processus qui crée les processus utilisateurs.

Question 6 – Une trappe est qualifiée d'interruption synchrone car :

- ☐ a. elle est synchronisée avec l'horloge du processeur.
- ☐ b. elle survient en liaison avec l'exécution du programme qui la lève.
- ☐ c. elle est levée par un périphérique matériel.

Question 7 – Le mode superviseur :

- ☐ a. est le mode d'exécution d'un programme utilisateur.
- ☐ b. est codé au niveau du registre d'état PSW du processeur.
- ☐ c. est un planificateur de tâches.
- ☐ d. est le mode d'exécution d'un programme système.

1.2 Contrôleur d'interruptions matérielles

On considère une machine admettant 8 niveaux d'interruptions matérielles numérotées de 0 à 7, le niveau d'interruptions 0 étant le plus prioritaire et le niveau 7 le moins prioritaire. Le processeur dispose de deux broches, une broche INT sur laquelle lui parvient le signal d'interruption, une broche INTA avec laquelle il acquitte les interruptions. Les 8 niveaux d'interruptions sont gérés par un contrôleur d'interruptions qui possède deux registres de 8 bits chacun, lui permettant de gérer les requêtes d'interruptions et leurs priorités. Le premier registre appelé registre ISR mémorise les interruptions en cours de traitement en positionnant à 1 le bit correspondant aux interruptions en cours de traitement. Le second registre IRR mémorise les interruptions reçues par le contrôleur en positionnant à 1 le bit correspondant à l'interruption reçue, cette mémorisation durant jusqu'à ce que l'interruption ait été traitée par le processeur.

Les priorités sur les interruptions fonctionnent de la manière suivante : lorsque le processeur traite une interruption de niveau *i* (le bit *i* du registre ISR est alors à 1), toutes les interruptions de priorité inférieure ou égale à *i* parvenant au contrôleur sont mémorisées dans le registre IRR, mais elles ne sont pas délivrées au processeur. Elles seront délivrées selon leur ordre de priorité, ultérieurement, dès que le traitement de l'interruption *i* aura été achevé. Une interruption de priorité supérieure au niveau *i* est au contraire délivrée au processeur et interrompt le traitement de l'interruption *i* en cours qui est repris ultérieurement, une fois le traitement de l'interruption plus prioritaire achevé.

1) Considérez l'état courant du contrôleur et du processeur donné sur la figure 1.13. Que se passe-t-il ? Que va faire le processeur ?

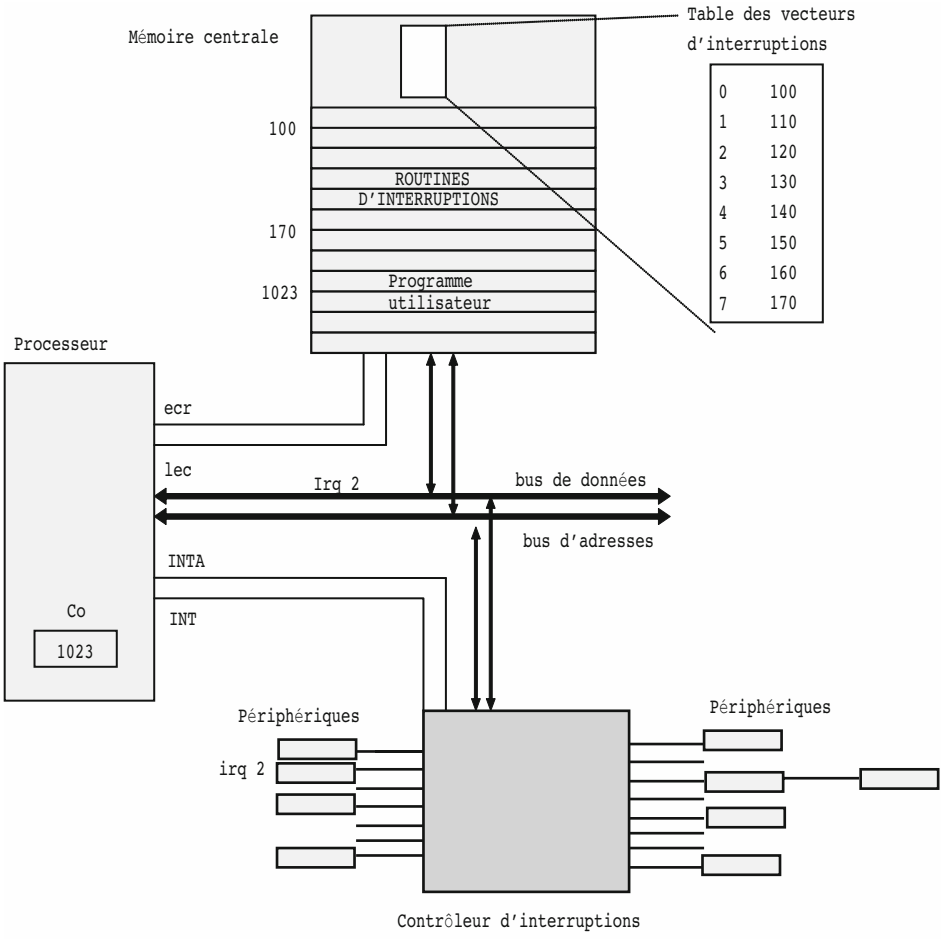

Figure 1.13 – État courant du contrôleur et du processeur

2) En partant de l'état précédent, décrivez la suite des opérations tant au niveau du contrôleur (évolution des registres ISR et IRR) que du processeur si l'on considère que surviennent simultanément au contrôleur les interruptions 1 et 5.

Solutions

1.1 Qu'avez-vous retenu ?

Question 1 – Quel est le rôle du système d'exploitation ?

- ☐ c. partager la machine physique entre les différents programmes et bâtir une machine virtuelle plus accessible à l'utilisateur.

Question 2 – Un système est dit réactif :

- ☐ b. car il doit réagir à des événements délivrés par des périphériques externes.

Question 3 – Une interruption est un signal dont le rôle est de :

- ☐ b. obliger le processeur à interrompre son traitement en cours pour traiter un événement survenu sur la machine.

Question 4 – Une commutation de contexte intervient :

- ☐ a. à chaque changement de mode d'exécution.

Question 5 – Le système Linux est un gestionnaire de processus, ce qui veut dire que :

- ☐ a. le système offre des services aux processus utilisateurs sous la forme de fonctions que les processus utilisateurs exécutent eux-mêmes.

Question 6 – Une trappe est qualifiée d'interruption synchrone car :

- ☐ b. elle survient en liaison avec l'exécution du programme qui la lève.

Question 7 – Le mode superviseur :

- ☐ b. est codé au niveau du registre d'état PSW du processeur.
- ☐ d. est le mode d'exécution d'un programme système.

1.2 Contrôleur d'interruptions matérielles

1) Le processeur va lancer l'exécution de la routine d'interruption liée à IRQ2, à savoir la routine d'adresse 120.

2) Les interruptions 1 et 5 arrivent donc le registre IRR prend la valeur 00100010.

L'interruption 1 est la plus prioritaire donc elle doit être traitée. ISR prend la valeur 00000110 pour mémoriser que les interruptions 1 et 2 sont en traitement au niveau du processeur. Le traitement de l'IRQ2 est interrompu au profit du traitement de l'IRQ1 au niveau du processeur.

À la fin de l'IRQ1, le registre IRR prend la valeur 00100000 et le registre ISR : devient 00000100. L'IRQ 2 est traitée au niveau du processeur. Lorsqu'elle s'achève, l'IRQ 5 en attente au niveau du contrôleur est délivrée au processeur. Les registres IRR et ISR sont respectivement : IRR 00000000 ; ISR 00100000.

PROCESSUS, THREADS ET ORDONNANCEMENT

2

PLAN

OBJECTIFS

➤ Ce chapitre présente les notions de processus lourds, de processus légers et d'ordonnancement au sein du système Linux. Le *processus* correspond à l'image d'un programme qui s'exécute tandis que l'*ordonnancement* correspond au problème de l'allocation du processeur et donc du partage du processeur entre différents processus.

2.1 PROCESSUS LINUX

2.1.1 Rappels sur la notion de processus

a) Définitions

La chaîne de production de programme composée des étapes de compilation et édition des liens transforme un programme écrit dans un langage de haut niveau en un programme dit exécutable, écrit en langage machine. Ce programme exécutable est stocké sur le disque. Afin de pouvoir être exécuté, le programme est placé en mémoire centrale par l'outil chargeur.

Imaginons que le programme à exécuter est placé en mémoire centrale à partir de l'emplacement d'adresse $0A10_{16}$. Le processeur commence l'exécution du programme : la première instruction de celui-ci est chargée dans le registre instruction[1] (RI) et le compteur ordinal (CO) contient l'adresse de la prochaine instruction à exécuter soit $0A14_{16}$. Lorsque l'instruction courante a été exécutée, le processeur charge dans le registre RI l'instruction pointée par le CO, soit par exemple l'instruction `add Imm R1 5` et le compteur ordinal prend la valeur $0A18_{16}$. L'exécution de l'instruction `add Imm R1 5` modifie le contenu du registre d'état PSW (*Program Status Word*)

1. Une instruction est supposée avoir une longueur de 4 octets.

puisque c'est une instruction arithmétique[1] : les indicateurs d'état tels que les drapeaux de signe, de nullité, etc. sont mis à jour. Ainsi, à chaque étape d'exécution du programme, le contenu des registres du processeur évolue. De même le contenu de la mémoire centrale peut être modifié par des opérations d'écriture ou de lecture dans la pile (instructions `push`, `pop`) ou encore des opérations de modification des données (instruction `store`).

On appelle *processus* l'image de l'état du processeur et de la mémoire au cours de l'exécution d'un programme. Le programme est statique et le processus représente la dynamique de son exécution.

Plus précisément, les définitions suivantes permettent de caractériser ce qu'est un processus :

- un processus est un programme en cours d'exécution auquel est associé un environnement processeur (le compteur ordinal CO, le registre d'état PSW, le registre sommet de pile RSP, les registres généraux) et un environnement mémoire (zone de code, de données et de pile) appelés *contexte du processus* ;
- un processus est l'instance dynamique d'un programme et incarne le fil d'exécution de celui-ci dans un *espace d'adressage protégé*, c'est-à-dire sur une plage mémoire dont l'accès lui est réservé.

b) Les outils de la chaîne de production de programme

Les outils de la chaîne de production de programmes sont notamment le compilateur et l'éditeur de liens.

Le compilateur

La *compilation* constitue la première étape de la chaîne de production de programmes. Elle permet la traduction d'un programme dit *programme source* écrit le plus souvent en langage de haut niveau vers un programme dit *programme objet* qui est soit directement le langage machine, soit le langage d'assemblage. Cette étape est précédée par un appel à un outil préprocesseur. Le préprocesseur est un outil exécuté en amont de la compilation. Il effectue des modifications textuelles sur le fichier source à partir de *directives*. Les différentes directives au préprocesseur, introduites par le caractère #, permettent :

- l'incorporation de fichiers source (`#include`) ;
- la définition de constantes symboliques et de macros (`#define`) ;
- la compilation conditionnelle (`#if`, `#ifdef`,...).

La compilation d'un programme écrit en langage C s'obtient en tapant la commande `cc -o prog.c` et produit un fichier objet de nom `prog.o`.

L'éditeur de liens

L'édition des liens constitue la deuxième étape du processus de production de programmes. Elle permet la construction du programme exécutable final en résolvant

1. Cette instruction réalise l'addition du contenu du registre R1 avec la valeur immédiate 5.

les liens vers les bibliothèques ou entre différents modules objets construits à l'aide de *compilations séparées*.

La commande `ld - o fich_exe module1.o module2.o -lm` permet la construction du programme exécutable `fich_exe` à partir des modules objets `module1.o module2.o` et de la bibliothèque mathématiques (`-lm`).

gcc

La commande `gcc` (GNU C Compiler) est un outil qui permet d'enchaîner les différentes phases de la production de programmes : préprocesseur, compilation et éditions des liens.

```
gcc - o fich_exe module1.co module2.c -lm
```

c) États d'un processus

Lors de son exécution, un processus est caractérisé par un *état* (figure 2.1) :

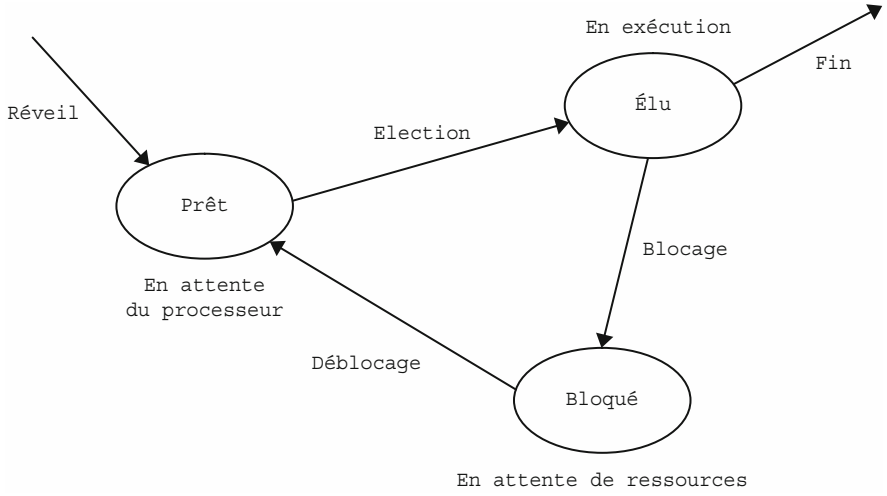

Figure 2.1 – Diagramme d'états d'un processus

- Lorsque le processus obtient le processeur et s'exécute, il est dans l'état *élu*. L'état élu est l'état d'exécution du processus.
- Lors de cette exécution, le processus peut demander à accéder à une ressource, par exemple il demande à lire des données depuis le disque. Le processus ne peut pas poursuivre son exécution tant qu'il n'a pas obtenu la ressource. Selon notre exemple, le processus doit attendre la fin de l'opération d'entrées-sorties disque pour disposer des données sur lesquelles il pourra effectuer les calculs suivants de son code. Le processus quitte alors le processeur et passe dans l'état *bloqué*. L'état bloqué est l'état d'attente d'une ressource autre que le processeur.
- Lorsque le processus a enfin obtenu la ressource qu'il attendait, celui-ci peut potentiellement reprendre son exécution. Cependant, dans le cadre de systèmes

multiprogrammés, il y a plusieurs programmes en mémoire centrale et donc plusieurs processus. Lorsque le processus est passé dans l'état bloqué, le processeur a été alloué à un autre processus. Le processeur n'est donc pas forcément libre. Le processus passe alors dans l'état prêt. L'état *prêt* est l'état d'attente du processeur.

Le passage de l'état prêt vers l'état élu constitue l'*opération d'élection*. Le passage de l'état élu vers l'état bloqué est l'*opération de blocage*. Le passage de l'état bloqué vers l'état prêt est l'*opération de déblocage*.

Un processus est toujours créé dans l'état prêt. Un processus se termine toujours à partir de l'état élu (sauf anomalie).

d) Bloc de contrôle du processus

Le chargeur a pour rôle de monter en mémoire centrale le code et les données du programme à exécuter. En même temps que ce chargement, le système d'exploitation crée une structure de description du processus associé au programme exécutable : c'est le *PCB* ou *Process Control Block* (bloc de contrôle du processus). Le bloc de contrôle d'un processus contient les informations suivantes (figure 2.2) :

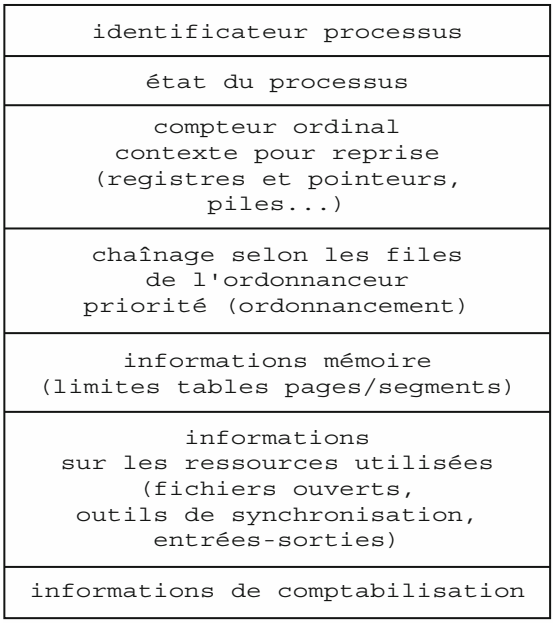

Figure 2.2 – Bloc de contrôle de processus

- un identificateur unique du processus (un entier) ;
- l'état courant du processus (élu, prêt, bloqué) ;
- le contexte processeur du processus : la valeur du compteur ordinal CO, la valeur des autres registres du processeur ;

- le contexte mémoire : ce sont des informations mémoire qui permettent de trouver le code et les données du processus en mémoire centrale ;
- des informations diverses de comptabilisation pour les statistiques sur les performances du système ;
- des informations liées à l'ordonnancement du processus ;
- des informations sur les ressources utilisées par le processus, tels que les fichiers ouverts, les outils de synchronisation utilisés, etc.

Le PCB permet la sauvegarde et la restauration du contexte mémoire et du contexte processeur lors des opérations de commutations de contexte.

2.1.2 Processus Linux

La structure d'un système tel que Linux repose sur le fait que tout processus Linux peut créer un autre processus. Le processus créateur est qualifié de *père* tandis que le processus créé est qualifié de *fils*.

Nous décrivons dans le paragraphe suivant la structure d'un processus Linux ainsi que les opérations permettant de créer et détruire un tel processus, en nous plaçant toujours sur une architecture matérielle de type x86.

a) Structure, états et attributs d'un processus Linux

Structure

Chaque processus Linux est représenté par un bloc de contrôle (structure `task_struct` définie dans le fichier `linux/sched.h`) alloué dynamiquement par le système au moment de la création du processus. Le bloc de contrôle contient tous les attributs permettant de qualifier et de gérer le processus. L'ensemble des blocs de contrôle est repéré depuis une table des processus (structure `task[0..NRTASK]` définie dans le fichier `kernel/sched.c`) qui comporte au plus 512 entrées, chaque entrée contenant un pointeur vers un bloc de contrôle. Par ailleurs, l'ensemble des blocs de contrôle est également inclus dans une liste doublement chaînée. Le premier élément du tableau `task`, qui représente également la tête de la liste doublement chaînée, correspond toujours au premier processus créé par le système au moment du boot. Ce processus ne s'exécute que lorsqu'aucun autre processus n'est prêt.

Le descripteur de processus est conservé dans le segment de données du noyau, conjointement avec la pile d'exécution du processus en mode noyau. En effet, chaque processus Linux dispose de deux piles d'exécution, l'une pour l'exécution en mode utilisateur, l'autre pour l'exécution en mode noyau.

Les informations contenues dans le bloc de contrôle concernent :

- l'état du processus ainsi que les informations d'ordonnancement (quanta d'exécutions, priorités et chaînage dans les files d'ordonnancement) ;
- l'identifiant du processus encore appelé PID, l'identifiant de l'utilisateur auquel appartient le processus (UID), l'identifiant du groupe de processus auquel le processus appartient (GID) ;

- l'identifiant du père du processus (PPID) (c'est-à-dire le processus à partir duquel le processus a été créé), l'identifiant du dernier fils créé, l'identifiant du processus frère cadet le plus jeune et l'identifiant du processus frère aîné le plus jeune ;
- les outils de communication utilisés par le processus tels que les sémaphores et les signaux ;
- les fichiers ouverts par le processus ;
- le terminal attaché au processus ;
- le répertoire courant du processus ;
- les signaux reçus par le processus ;
- le contexte mémoire du processus ;
- la partie du contexte processeur du processus sauvegardé dans le segment d'état de tâche (*TSS Task Segment Segment*), le reste étant placé dans la pile noyau du processus ;
- les temps d'exécutions du processus en mode utilisateur et en mode superviseur ainsi que ceux de ses fils.

États d'un processus Linux

Un processus Linux évolue entre deux modes au cours de son exécution : le mode utilisateur qui est le mode classique d'exécution et le mode noyau qui est le mode dans lequel se trouve un processus élu passé en mode superviseur, un processus prêt ou un processus bloqué (endormi). Plus précisément, un processus Linux peut passer par les états suivants (figure 2.3) :

- `TASK_RUNNING` : le processus est dans l'état prêt ou dans l'état élu ;

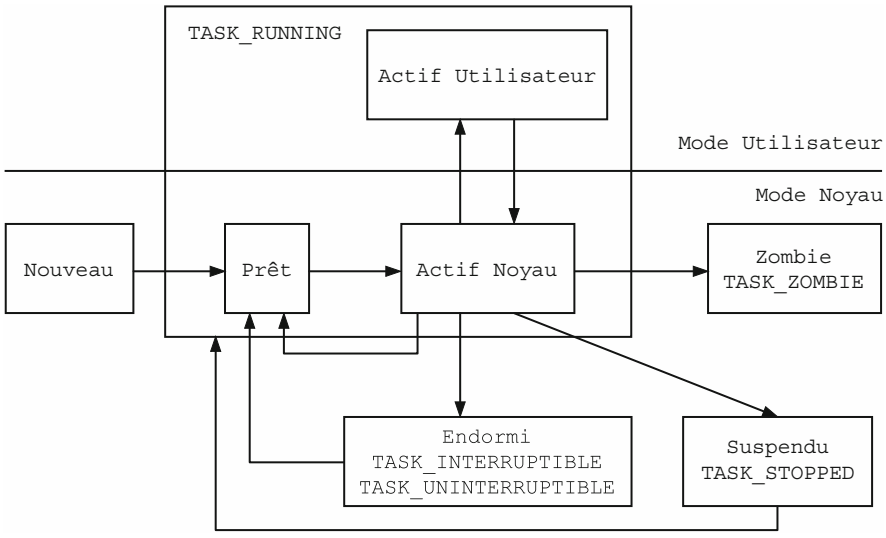

Figure 2.3 – Diagramme d'états d'un processus Linux

- `TASK_INTERRUPTIBLE` et `TASK_UNINTERRUPTIBLE` : le processus est dans l'état bloqué. Dans le second état, le processus peut être réveillé par la réception d'un signal (*cf.* chapitre 6) ;
- `TASK_ZOMBIE` : le processus a terminé son exécution mais son bloc de contrôle existe toujours car son père n'a pas pris en compte sa terminaison ;
- `TASK_STOPPED` : le processus a été suspendu par l'utilisateur suite à la réception d'un signal tel que `SIGSTOP`, `SIGTSTP`, `SIGTTIN` ou `SIGTTOU` (*cf.* chapitre 6).

Principaux attributs

Chaque processus Linux est caractérisé par un numéro unique appelé PID (entier non signé de 32 bits) qui lui est attribué par le système au moment de sa création. Chaque nouveau processus reçoit comme valeur de PID, la dernière valeur attribuée lors de l'opération de création précédente augmentée de 1. Par ailleurs, afin de maintenir une certaine compatibilité avec les systèmes Unix traditionnels fonctionnant sur 16 bits, la plus grande valeur de PID admissible est 32 767. Le 32 768^{e} processus créé reçoit le plus petit numéro de PID libéré par un processus mort entre-temps.

Le processus est également caractérisé par l'identifiant de l'utilisateur ayant provoqué sa création, l'UID. Par ailleurs, chaque processus Linux pouvant créer lui-même un autre processus, chaque processus est également caractérisé par l'identifiant du processus qui l'a créé (son père), appelé PPID.

Enfin, chaque processus Linux appartient à un groupe de processus, identifié par un numéro qui est le PID du processus créateur du groupe, encore appelé processus *leader*. L'identifiant du groupe, le GID, auquel le processus appartient fait partie de ses attributs.

UID et GID sont utilisés pour définir les droits d'accès du processus vis-à-vis des ressources de la machine.

Les primitives suivantes permettent à un processus respectivement de connaître la valeur de son PID, du PID de son père, l'identifiant de l'utilisateur qui l'a créé et le numéro du groupe auquel il appartient.

```
#include <unistd.h>
pid_t getpid(void);    retourne le PID du processus appelant.
pid_t getppid(void);   retourne le PPID du processus appelant.
uid_t getuid(void);    retourne l'UID du processus appelant.
gid_t getgid(void);    retourne le GID du processus appelant.
```

b) Création d'un processus Linux

Sous le système Linux, tout processus peut créer un nouveau processus qui est une exacte copie de lui-même. Le processus créateur est appelé « le père » et le processus créé est appelé « le fils ». Il découle de cette possibilité la formation d'une arborescence de processus, reliés entre eux par des liens de filiation.

La primitive `fork()` permet la création dynamique d'un nouveau processus qui s'exécute de manière concurrente avec le processus qui l'a créé. Tout processus Linux hormis le processus 0 est créé à l'aide de cette primitive. Le processus créateur

(le père) par un appel à la primitive `fork()` crée un processus fils qui est une copie exacte de lui-même (code et données). Le prototype de la fonction est le suivant :

```
#include <unistd.h>
pid_t fork (void);
```

Lors de l'exécution de la primitive `fork()`, si les ressources noyaux sont disponibles, le système effectue les opérations suivantes (primitive `do_fork` du noyau) :

- le système alloue un bloc de contrôle et une pile noyau pour le nouveau processus et trouve une entrée libre dans la table des processus pour ce nouveau bloc de contrôle ;
- le système copie les informations contenues dans le bloc de contrôle du père dans celui du fils sauf les informations concernant le PID, le PPID et les chaînages vers les fils et les frères ;
- le système alloue un PID unique au nouveau processus ;
- le système associe au processus fils le contexte mémoire du processus parent c'est-à-dire le code, les données, et la pile. Les parties données et pile ne seront dupliquées pour le processus fils que lorsque celui-ci ou son père chercheront à les modifier sur une opération d'écriture. Cette technique est appelée technique du « *copy on write* » ; elle permet en différant la duplication du contexte mémoire du processus père de diminuer le coût de l'opération de création d'un nouveau processus (*cf.* chapitre 4) ;
- l'état du nouveau processus est mis à la valeur `TASK_RUNNING` ;
- le système retourne au processus père le PID du processus créé et au nouveau processus (le fils) la valeur 0.

Lors de cette création le processus fils hérite de tous les attributs de son père sauf :

- l'identificateur de son père ;
- son propre identificateur ;
- les temps d'exécution du nouveau processus sont nuls.

À l'issue de l'exécution de la primitive `fork()`, chaque processus, le père et le fils, reprend son exécution après le `fork()`. Le code et les données étant strictement identiques, il est nécessaire de disposer d'un mécanisme pour différencier le comportement des deux processus après le `fork()`. On utilise pour cela le code retour du `fork` qui est différent chez le fils (toujours 0) et le père (PID du fils créé).

Ainsi, dans le programme suivant, le processus n° 12 222 est le processus père. Le processus n° 12 224 est le processus fils créé par le processus n° 12 222 à l'issue de l'appel à la primitive `fork()`. Une fois la primitive `fork()` exécutée par le père, les deux processus (le père et le fils) reprennent leur exécution de manière concurrente. Le processus fils a pour retour de la primitive `fork()` la valeur 0. Il va donc exécuter la partie de l'alternative pour laquelle `(if pid == 0)` est vrai. Le processus père au contraire a pour retour de la primitive `fork()` la valeur du PID du processus créé c'est-à-dire une valeur positive. Il va donc exécuter l'autre partie de l'alternative. Les traces générées par cette exécution sont par exemple[1] :

1. L'ordre des traces dépend de l'ordonnancement réalisé par le système et de l'ordre d'exécution des deux processus père et fils.

```
je suis le fils; mon pid est 12 224
je suis le père; mon pid est 12 222
pid de mon fils, 12 244
pid de mon père, 12 222


/*****************************************************************/
/*                    Création d'un processus fils               */
/*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
main ()
{
pid_t ret;
ret = fork();
if (ret == 0)
{
       printf ("je suis le fils; mon pid est %d\n", getpid());
       printf ("pid de mon père, %d\n", getppid());
}
else
{
       printf ("je suis le père; mon pid est %d\n", getpid());
       printf ("pid de mon fils, %d\n", ret);
}
}
```

En cas d'échec la primitive `fork()` renvoie la valeur – 1 et la variable `errno` est positionnée avec les valeurs suivantes :

- `EAGAIN`, le nombre maximal de processus pour l'utilisateur courant ou pour le système a été atteint ;
- `ENOMEM`, le noyau n'a pas pu allouer assez de mémoire pour créer un nouveau processus.

c) Terminaison d'un processus et synchronisation avec son père

Terminaison d'un processus

Un processus termine normalement son exécution en achevant l'exécution du code qui lui est associé. Cette terminaison s'effectue par le biais d'un appel à la primitive `exit (status)` où `status` est un code retour compris entre 0 et 255 qui est transmis au père par le processus défunt. Par convention, une valeur de retour égale à 0 caractérise une terminaison normale du processus et une valeur supérieure à 0 code une fin anormale. Le prototype de la fonction `exit()` est le suivant :

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

Le processus peut également terminer son exécution de manière anormale, suite à une erreur ayant provoqué par exemple une trappe, et la terminaison du processus est alors forcée par le système d'exploitation. Dans ce cas, le système d'exploitation positionne un code d'erreur spécifique de l'erreur rencontrée qui est également retourné au processus père.

Lors de la terminaison d'un processus, le système désalloue les ressources encore possédées par le processus mais ne détruit pas le bloc de contrôle de celui-ci, passe l'état du processus à la valeur `TASK_ZOMBIE` puis avertit le processus père de la terminaison de son fils, en lui envoyant le signal `SIGCHLD` et en lui passant notamment la valeur `status` ou le code d'erreur positionné (fonction `do_exit()` du noyau).

Un processus fils défunt reste zombie jusqu'à ce que son père ait pris connaissance de sa mort. Un processus fils orphelin, suite au décès de son père (le processus père s'est terminé avant son fils) est toujours adopté par le processus numéro 1 (`Init`) et non par son grand-père.

Synchronisation avec le père

Un processus père peut se mettre en attente de la mort de l'un de ses fils, par le biais des primitives `wait()` et `waitpid()`. Lorsque le processus père prend connaissance de la mort de l'un de ses fils, il cumule les temps d'exécution en mode utilisateur et en mode superviseur de ce fils avec ceux des fils précédemment décédés, puis il détruit le bloc de contrôle du processus fils. Le prototype de la fonction `wait()` est le suivant :

```
#include <sys/wait.h>
pid_t wait (int *status);
```

L'exécution du processus père est suspendue jusqu'à ce qu'un processus fils se termine. Si un processus fils est déjà dans l'état zombie au moment de l'appel, la fonction retourne immédiatement le résultat, à savoir le PID du fils terminé et le code retour de celui-ci dans la variable `status`. Le prototype de la fonction `waitpid()` est le suivant :

```
#include <sys/wait.h>
pid_t waitpid (pid_t pid, int *status, int options);
```

Le principe de la primitive est le même que précédemment, mais elle permet en plus de spécifier dans la variable `pid`, l'identité du processus fils attendu. Si ce processus fils est déjà dans l'état zombie au moment de l'appel, la fonction retourne immédiatement le résultat, à savoir le PID du fils terminé et le code retour de celui-ci dans la variable `status`. Plus précisément, le contenu de la variable `pid` est interprété comme suit :

- si `pid` est une valeur strictement positive, alors le système suspend le processus père jusqu'à ce qu'un processus fils dont la valeur du PID est égal à `pid` se termine ;
- si `pid` est nul, alors le système suspend le processus père jusqu'à la mort de n'importe quel fils appartenant au même groupe que le père ;

- si `pid` est égal à – 1, alors le système suspend le processus père jusqu'à la mort de n'importe lequel de ses fils ;
- si `pid` est une valeur strictement inférieure à – 1 alors le système suspend le processus père jusqu'à la mort de n'importe lequel de ses fils, dont le numéro de groupe est égal à – `pid`.

L'interprétation du contenu de la variable `status` se fait à l'aide des macros `WIFEEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, `WIFSTOPPED` définies dans le fichier `<sys/wait.h>`. Ainsi :

- `WIFEEXISTED(status)` est vrai si le processus fils s'est terminé par un appel à la primitive `exit()` ;
- `WEXITSTATUS(status)` permet de récupérer le code passé par le fils au moment de sa terminaison ;
- `WIFSIGNALED(status)` permet de savoir que le fils s'est terminé à cause d'un signal ;
- `WIFSTOPPED(status)` indique que le fils est stoppé temporairement.

Le champ `options` peut notamment prendre la valeur `WNOHANG` qui provoque un retour immédiat de la primitive si aucun fils ne s'est encore terminé.

La primitive renvoie la valeur – 1 en cas d'échec. La variable `errno` peut prendre alors les valeurs suivantes :

- `ECHILD`, le processus spécifié par `pid` n'existe pas ;
- `EFAULT`, la variable `status` contient une adresse invalide ;
- `EINTR`, l'appel système a été interrompu par la réception d'un signal.

Un exemple

```
/*********************************************************************/
/*                 Terminaison du processus fils                     */
/*               et affichage de sa valeur de retour                 */
/*********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
main ()
{
pid_t ret,fils_mort;
int status;
ret = fork();
if (ret == 0)
{
       printf(" je suis le fils; mon pid est %d\n", getpid());
       printf ("pid de mon père, %d\n", getppid());
       exit(0);}
else
{
       printf ("je suis le père; mon pid est %d\n", getpid());
```

```
        printf ("pid de mon fils, %d\n", ret);
        fils_mort = wait(&status);
        printf ("je suis le père; le pid de mon fils mort est %d\n",
fils_mort);
        if (WIFEXITED(status))
             printf ("je suis le père; le code retour de mon fils est %d\
n", WEXITSTATUS(status));
}
}
```

Les traces d'exécution de ce programme sont par exemple :

```
je suis le père; mon pid est 4243
je suis le fils; mon pid est 4244
pid de mon père, 4243
pid de mon fils, 4244
je suis le père; le pid de mon fils mort est 4244
je suis le père; le code retour de mon fils est 0
```

d) Les primitives de recouvrement

Interface d'un programme exécutable

La forme générale d'un programme principal en langage C est la suivante :

```
int main (int argc, char *argv[], char **arge[]);
```

Cette interface permet au programme exécutable de récupérer un ensemble de paramètres dans un environnement particulier. Ainsi :

- `argc` est le nombre total de paramètres transmis au programme. C'est le nombre total de composants de la commande lancée ;
- `argv` est un tableau comprenant les différents paramètres passés au programme exécutable. Le premier élément de ce tableau `argv[0]` est toujours le nom de la commande elle-même ;
- `arge` est une liste de pointeurs permettant d'accéder à l'environnement d'exécution du processus. Chaque pointeur est une chaîne de caractères de la forme `nom_variable = valeur`. Nous revenons sur les variables d'environnement dans le paragraphe exercices de ce chapitre.

Exemple

On lance l'exécution d'un programme calcul avec trois paramètres :

```
sh > calcul 3 4
```

On a :

- `argc` = 3,
- `argv[0]` = "calcul", `argv[1]` = "3" et `argv[2]` = "4",
- `arge` contient la liste des variables d'environnement avec leur valeur telles qu'elles ont été positionnées pour l'interpréteur de commandes (shell `sh`), père du processus calcul.

Les primitives exec

Les primitives de recouvrement constituent un ensemble de primitives (famille Exec) permettant à un processus de charger en mémoire, un nouveau code exécutable. L'exécution d'une des primitives de la famille Exec entraîne l'écrasement du code hérité au moment de la création (primitive `fork()`) par le code exécutable spécifié en paramètre de la primitive. Des données peuvent être passées au nouveau code exécutable *via* les arguments de la primitive Exec qui les récupère dans le tableau `argv[]`. Il est également possible de modifier l'environnement que le processus a hérité de son père.

La famille Exec est constituée de 6 primitives dont les prototypes sont donnés ci-après :

- `int execl (const char *ref, const char *arg, …, NULL)`: `ref` est le chemin d'un exécutable à partir du répertoire courant, `const char *arg, …, NULL` est la liste des arguments.
- `int execlp (const char *ref, const char arg, …, NULL)`: `ref` est le chemin d'un exécutable à partir de la variable d'environnement `PATH, const char *arg, …, NULL` est la liste des arguments.
- `int execle (const char *ref, const char *arg, …, const char *envp[])`: `ref` est le chemin d'un exécutable à partir du répertoire courant, `const char *arg, …, NULL` est la liste des arguments, `const char *envp[]` est un tableau spécifiant les variables d'environnement sous la forme nom_var = valeur.
- `int execv (const char *ref, const char *arg[])`: `ref` est le chemin d'un exécutable à partir du répertoire courant, `const char *arg []` est un tableau contenant les arguments.
- `int execvp (const char *ref, const char *arg[])`: `ref` est le chemin d'un exécutable à partir de la variable d'environnement `PATH, const char *arg []` est un tableau contenant les arguments.
- `int execve (const char *ref, const char *arg[], …, const char *envp[])`: `ref` est le chemin d'un exécutable à partir du répertoire courant, `const char *arg []` est un tableau contenant les arguments, `const char *envp[]` est un tableau spécifiant les variables d'environnement sous la forme nom_var = valeur.

Dans l'exemple suivant, le processus fils écrase le code hérité de son père par celui de la commande `ls`, avec comme paramètres les chaînes `-l` et `/`. La commande exécutée par le fils est donc `ls -l /` qui liste l'ensemble des fichiers du répertoire racine.

```
/*****************************************************************/
/*   Écrasement du code hérité pour exécuter la commande ls -l /  */
/*****************************************************************/
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

main()
```

```
{
pid_t pid;

pid = fork();
if (pid == -1)
      printf ("erreur création de processus");
else
if (pid == 0)
{
     printf ("je suis le fils, mon pid est %d\n", getpid());
     printf("Le pid de mon père est %d\n", getppid());
     execlp("ls","ls", "-l", "/", NULL);
 }
else
{
     printf ("je suis le père, mon pid est %d\n", getpid());
     printf("Le pid de mon fils est %d\n", pid);
     wait ();
}
```

Nous donnons ci-après un deuxième exemple pour lequel le processus fils créé effectue un appel à une primitive `exec()` en demandant le recouvrement de son code par celui d'un programme `calcul`, qui attend deux arguments entiers à additionner. On fera attention ici au fait que les arguments passés *via* les primitives `exec()` sont des chaînes de caractères. Aussi dans le processus fils, avant l'appel à la primitive `exec()`, il est nécessaire de convertir les entiers `i` et `j` en chaînes de caractères. Dans le programme `calcul.c`, on notera que de manière inverse, les paramètres `i` et `j` récupérés dans les arguments `argv[1]` et `argv[2]` sont convertis de chaînes de caractères vers entiers.

```
/*********************************************************************/
/*             Écrasement de code et conversion de types             */
/*********************************************************************/
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

main()
{
pid_t pid;
int i, j;
char conv1[10], conv2[10];

pid = fork();
printf("Donnez les valeurs de i et j");
scanf ("%d, %d", i, j);
if (pid == 0)
{
  sprintf(conv1, "%d", i); /* conversion de entier vers caractères */
```

```
  sprintf(conv2, "%d", j); /* conversion de entier vers caractères */
  execl("/home/delacroix/calcul", "calcul", conv1, conv2, NULL);
 }
else
{
  wait ();
}

Calcul.c
main(argc,argv)
{
int somme;
if (argc <> 3) {printf("erreur"); exit();}
/* atoi: conversion caractère à entier */
somme = atoi(argv[1]) + atoi(argv[2]);
exit();
}
```

Les primitives `exec()` renvoie la valeur – 1 en cas d'échec. La variable `errno` peut prendre alors les valeurs suivantes :

- `E2BIG`, la liste des arguments ou des variables d'environnement est trop importante ;
- `EACCES`, le processus n'a pas le droit d'exécuter le fichier spécifié par `ref` ;
- `EFAULT`, `ref` est un chemin invalide ;
- `ENAMETOOLONG`, le nom de fichier spécifié par `ref` est trop long ;
- `ENOENT`, le nom de fichier spécifié par `ref` n'existe pas ;
- `ENOEXE`, le fichier spécifié par `ref` n'est pas un fichier exécutable ;
- `ENOMEM`, il n'y a pas suffisamment de mémoire pour exécuter le programme ;
- `ENOTDIR`, l'un des composants du chemin `ref` n'est pas un répertoire.

2.2 Processus léger (thread)

2.2.1 Notion de processus léger

Le *processus léger* (*thread*) constitue une extension du modèle traditionnel de processus, appelé en opposition processus lourd. Un processus classique est constitué d'un espace d'adressage avec un seul fil d'exécution, ce fil d'exécution étant représenté par une valeur de compteur ordinal et une pile d'exécution.

Dans ce contexte, l'extension du modèle précédent consiste à admettre plusieurs fils d'exécution indépendants dans un même espace d'adressage, chacun de ces fils d'exécution encore appelés *processus légers*, *threads* ou *activités* étant caractérisés par une valeur de compteur ordinal propre et une pile d'exécution privée. L'entité contenant les différents fils d'exécutions est appelée *processus* ou bien *acteur* (figure 2.4).

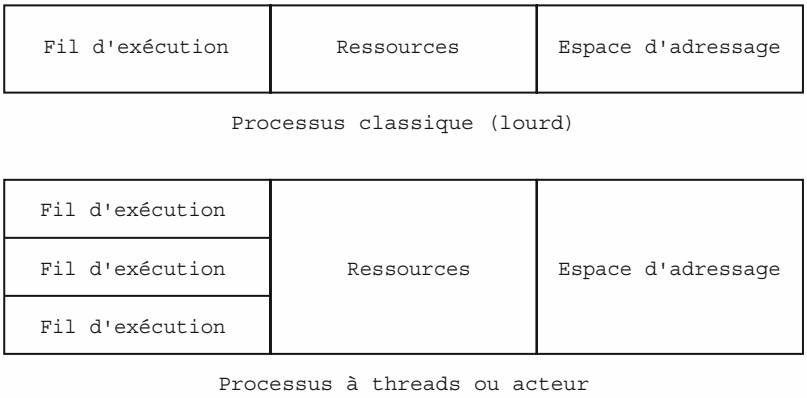

Figure 2.4 – Processus lourd et processus léger

Le gros avantage lié à la notion de processus léger est un allègement des opérations de commutations de contexte : en effet lorsque le processeur commute d'un fil d'exécution à un autre fil et que ces deux fils appartiennent au même espace d'adressage, alors la commutation est très allégée puisqu'elle consiste seulement à changer de pile et de valeur de compteur ordinal, le contexte mémoire restant le même.

De même, l'opération de création d'un nouveau fil d'exécution est sensiblement allégée puisqu'elle ne nécessite plus la duplication complète de l'espace d'adressage du processus père.

A contrario, comme les processus légers au sein d'un même processus partage le même espace d'adressage, il s'ensuit des problèmes de partage de ressources plus importants.

Les processus légers peuvent être implémentés à deux niveaux différents, soit au niveau utilisateur, soit au niveau noyau.

a) Implémentation au niveau utilisateur

Lorsque l'implémentation des processus légers est réalisée au niveau utilisateur, le noyau ignore ceux-ci et ordonnance des processus classiques composés d'un seul fil d'exécution. Une bibliothèque système gère l'interface avec le noyau en prenant en charge la gestion des threads et en cachant ceux-ci au noyau. Cet exécutif est donc responsable de commuter les threads au sein d'un même processus, lorsque ceux-ci en font explicitement la demande (figure 2.5).

L'avantage de cette approche est qu'elle allège les commutations entre threads d'un même processus puisque celles-ci s'effectuent au niveau utilisateur sans passage par le mode noyau. On évite ainsi une commutation de contexte lourde.

Un inconvénient majeur est qu'au sein d'un même processus, un thread peut monopoliser le processeur pour lui seul. Par ailleurs, un thread bloqué au sein d'un processus suite à une demande de ressource ne pouvant être immédiatement satisfaite, bloque l'ensemble des threads du processus.

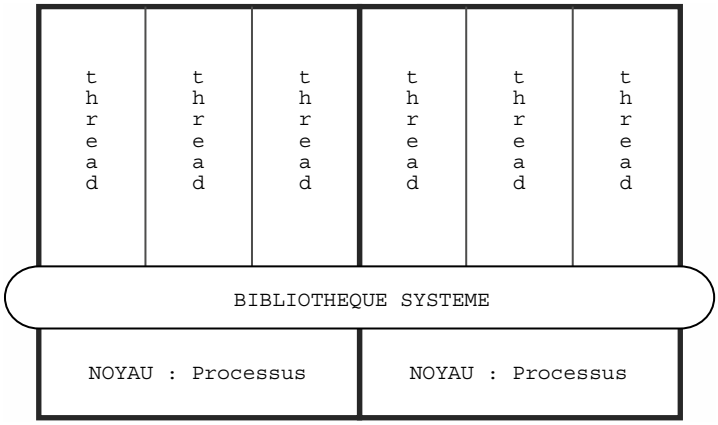

Figure 2.5 – **Processus léger de niveau utilisateur**

b) Implémentation au niveau noyau

Lorsque l'implémentation des threads est effectuée au niveau du noyau, alors ce dernier connaît l'existence des threads au sein d'un processus et il attribue le processeur à chacun des threads de manière indépendante. Chaque descripteur de processus contient alors également une table des threads qui le composent avec pour chacun d'eux la sauvegarde de son contexte processeur (figure 2.6).

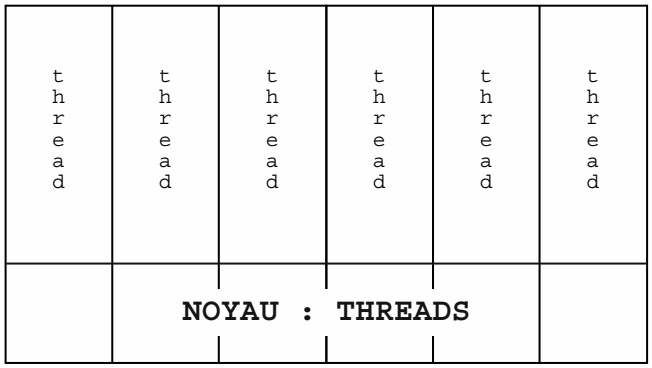

Figure 2.6 **Processus léger de niveau noyau**

L'avantage de cette approche est d'éviter le blocage de tout un processus à partir du moment où l'un de ses threads est bloqué. Par contre, chaque commutation de contexte étant gérée par le noyau, celles-ci sont un peu plus coûteuses.

2.2.2 Primitives de gestion des threads

Les primitives permettant la gestion des threads sont très semblables à celles permettant la gestion des processus. Elles sont définies dans une librairie nommée

LinuxThreads créée par Xavier Leroy et sont conformes à la norme Posix.1c. Cette librairie étant indépendante de la librairie GlibC habituelle, il faut inclure l'en-tête `<pthread.h>` dans les fichiers sources et ajouter l'option `-lpthread` lors de l'appel à l'éditeur des liens pour spécifier l'utilisation de la librairie contenant les fonctions liées aux threads.

Chaque thread est identifié de manière unique au sein d'une application par un type opaque `pthread_t`, qui peut être implémenté selon les systèmes, soit sous la forme d'un entier long non signé, soit sous la forme d'un type structuré. Ce type joue un rôle analogue au type `pid_t` des processus. La primitive `pthread_self()` permet à chaque thread de connaître son propre identifiant.

a) Création d'un thread

La primitive `pthread_create()` permet la création, au sein d'un processus, d'un nouveau fil d'exécution, identifié par l'identificateur `*thread` et attaché à l'exécution de la routine `(*start_routine)`. Le fil d'exécution démarre son exécution au début de la fonction spécifiée et disparaît à la fin de l'exécution de celle-ci. Dans le prototype suivant, `*attr` correspond aux attributs associés au thread si l'on souhaite modifier les attributs standards affectés au moment de la création (cet argument est le plus souvent mis à NULL pour hériter des attributs standards), et `*arg` correspond à un argument passé au thread pour l'exécution de la fonction `(*start_routine)`.

La primitive renvoie la valeur 0 en cas de succès et sinon une valeur négative correspondant à l'erreur survenue.

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

b) Terminaison d'un thread

La primitive `pthread_exit(void *ret)` met fin au thread qui l'exécute. Elle retourne le paramètre `ret` qui peut être récupéré par un autre thread effectuant pour sa part un appel à la primitive `pthread_join (pthread_t thread, void **retour)` où `thread` correspond à l'identifiant du thread attendu et `retour` correspond à la valeur `ret` retournée lors de la terminaison.

L'exécution du processus qui effectue un appel à la fonction `int pthread_join (pthread_t thread, void **ret)` est suspendue jusqu'à ce que le thread attendu se soit achevé. Le paramètre `*ret` contient la valeur passée par le thread au moment de l'exécution de la primitive `pthread_exit()`.

c) Attributs d'un thread

Chaque thread est doté des attributs suivants :

- l'adresse de départ et la taille de la pile qui lui est associée ;
- la politique d'ordonnancement qui lui est associée (*cf.* paragraphe 2.4) ;
- la priorité qui lui est associée ;
- son attachement ou son détachement. Un thread détaché se termine immédiatement sans pouvoir être pris en compte par un `pthread_join()`.

Lors de la création d'un nouveau thread par un appel à la primitive `pthread_create()`, les attributs de ce thread sont fixés par l'intermédiaire de l'argument `pthread_attr_t *attr`. Si les attributs par défaut sont suffisants, ce paramètre est mis à NULL. Sinon, il faut au préalable initialiser une structure de type `pthread_attr_t`, en invoquant la primitive `pthread_attr_init()`, puis modifier chacun des attributs en utilisant les fonctions `pthread_attr_getXXX()` et `pthread_attr_setXXX()` qui permettent respectivement de consulter et modifier l'attribut XXX. L'étude et l'utilisation de ces fonctions font l'objet d'un exercice au paragraphe 2.5.

Exemples

```
/*********************************************************************/
/*      Création de threads exécutant la fonction f_thread()      */
/*********************************************************************/
#include <stdio.h>
#include <pthread.h>

pthread_t pthread_id[3];

void f_thread(int i) {
printf ("je suis le %d-eme pthread d'identité %d.%d\n", i, getpid(),
pthread_self());
}

main()
{
int i;
for (i = 0; i < 3; i++)
   pthread_create(pthread_id[i], pthread_attr_default, f_thread, i);
printf ("je suis le thread initial %d.%d\n", getpid(), pthread_self());
pthread_join();
}
```

Le programme suivant est un deuxième exemple visant à illustrer le partage de l'espace d'adressage entre les threads d'un même processus. Ici, le thread principal et le thread fils qui exécute la fonction `addition()` se partagent une même variable *i*, qu'ils incrémentent chacun de leur côté. Les traces d'exécution sont par exemple :

```
hello, thread principal 1000
hello, thread fils 1010
hello, thread principal 3010
hello, thread fils 3030

/*********************************************************************/
/*     Partage d'une variable entre threads d'un même processus   */
/*********************************************************************/
#include <stdio.h>
#include <pthread.h>

int i;
```

```
void addition()
{
i = i + 10;
printf ("hello, thread fils %d\n", i);
i = i + 20;
printf ("hello, thread fils %d\n", i);
}

main()
{ pthread_t num_thread;

i = 0;
if (pthread_create(&num_thread, NULL, (void *(*)())addition, NULL) == -1)
     perror ("pb pthread_create\n");
i = i + 1000;
printf ("hello, thread principal %d\n", i);
i = i + 2000;
printf ("hello, thread principal %d\n", i);
pthread_join(num_thread, NULL);
}
```

Le programme suivant est analogue au précédent, mais utilise seulement les processus lourds. Le processus père crée un processus fils et chacun des processus incrémente également une variable *i* des mêmes valeurs que précédemment. Les traces d'exécution montrent que cette fois, les deux processus père et fils manipulent deux variables *i* différentes. Le fils a hérité de la variable *i* du père, mais il possède sa propre instance de cette variable qui est différente de celle du père. Les traces d'exécutions sont par exemple :

```
hello, fils 10
hello, fils 30
hello, père 1000
hello, père 3000
```

```
/*********************************************************************/
/*           Héritage de variables entre processus lourds          */
/*********************************************************************/
#include <stdio.h>

int i;

main()
{
int pid;

i = 0;
pid = fork();
if (pid == 0)
{
i = i + 10;
printf ("hello,  fils %d\n", i);
```

```
i = i + 20;
printf ("hello, fils %d\n", i);
}
else
{
i = i + 1000;
printf ("hello, père %d\n", i);
i = i + 2000;
printf ("hello, père %d\n", i);
wait();
}
}
```

2.2.3 Implémentation sous Linux

L'implémentation des threads du système Linux est réalisée au niveau du noyau. La création d'un thread est réalisée par la routine `clone()` qui admet les paramètres :

- `fn` spécifiant la fonction à exécuter par le nouveau fil d'exécution ;
- `arg` pointant sur une liste de paramètres à passer à la fonction `fn` ;
- `flags` un ensemble de drapeaux précisant les ressources à partager entre le fil d'exécution père et le fil d'exécution fils, notamment le contexte mémoire, les fichiers ouverts, les gestionnaires de signaux, etc. ;
- `child_stack` précisant la valeur du pointeur de pile utilisateur pour le fil d'exécution fils.

La fonction `clone()` réalise elle-même un appel à la fonction `do_fork()` du noyau qui effectue au sein du même espace d'adressage une duplication partielle du fil d'exécution père en fonction des différents drapeaux positionnés.

2.3 Démarrage du système Linux

Tout le système Linux repose sur le concept de threads et sur celui arborescent de processus père et de processus fils. Le processus fils est créé par duplication du processus père puis il recouvre le code hérité du processus père par un code exécutable spécifique.

L'ancêtre de tous les processus du système Linux est un thread noyau, appelé processus 0, créé par la fonction `start_kernel()`. Ce thread noyau qui occupe la première entrée de la table des processus du système, initialise les structures de données du noyau, autorise les interruptions, puis il crée un second thread noyau, appelé processus 1 ou encore processus `init`. Le thread 0 s'endort alors pour n'être réveillé et choisi pour s'exécuter que lorsqu'aucun autre processus de la machine n'est prêt.

Le thread `init` crée à son tour 4 autres threads noyau chargés de gérer le cache disque (threads `kupdate` et `bdflush` ; *cf.* chapitre 3) et l'activité de swappage (threads `kswapd` et `kpiod` ; *cf.* chapitre 5). Enfin, le thread `init` invoque la primitive `execve` pour exécuter le programme `init` et devient de ce fait un processus à part entière.

Le processus `init` va créer à son tour un certain nombre de processus : d'une part les *processus démons* responsable de fonctions système telles que la surveillance du réseau (`inetd`), la gestion de l'imprimante (`lpd`)…, d'autre part des processus `getty` chargés de la surveillance des terminaux.

Lorsqu'un utilisateur vient pour utiliser la machine, il se logue sur un terminal. Le processus `getty` crée alors un nouveau processus, le processus `login` chargé de lire le login de l'utilisateur, son mot de passe et de vérifier dans le fichier des mots de passe `/etc/passwd` que l'utilisateur est autorisé à se connecter à la machine. Si l'utilisateur est autorisé à se connecter à la machine, alors le processus `login` va créer à son tour un nouveau processus, le processus `shell`, c'est-à-dire l'interpréteur de commande pour le nouvel utilisateur. Cet interpréteur de commandes va prendre en charge les commandes tapées par l'utilisateur et pour chacune d'elles il va créer un nouveau processus chargé d'exécuter la commande. Ce processus existe le temps de l'exécution du programme ou de la commande (figure 2.7).

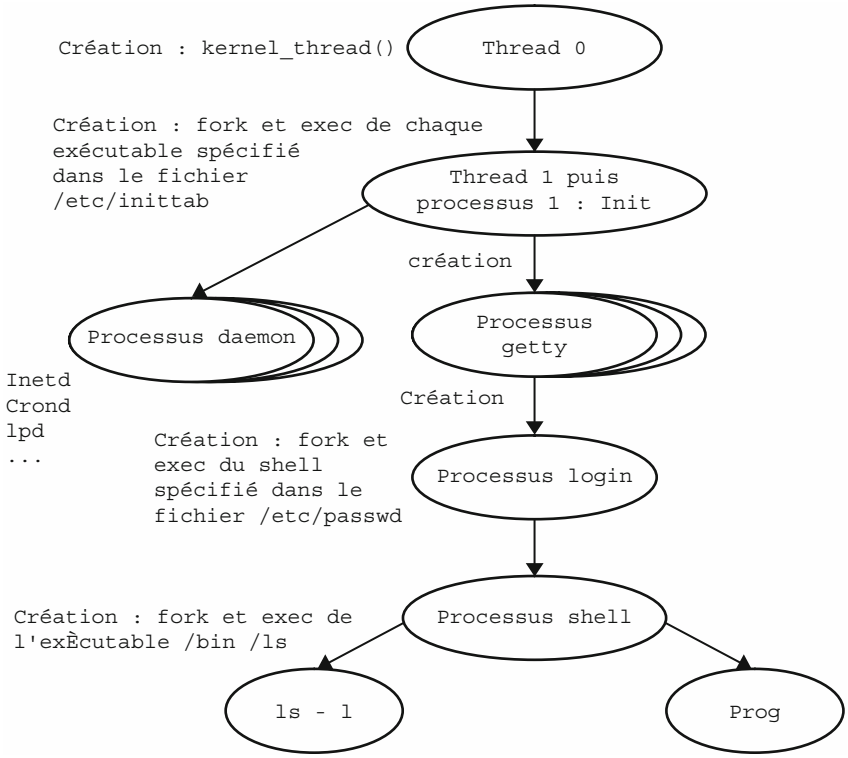

Figure 2.7 – La chaîne de processus Linux

La commande `pstree` permet de visualiser à un instant donné l'arborescence de processus existants sur une machine. Ainsi, sur les traces suivantes, l'arborescence montre que le processus `init` est le père de tous les autres processus. La commande `pstree` est le processus fils de l'interpréteur de commande appelé `bash`, qui est lui-

même le fils du processus `login`. Ce processus `login` est le fils du processus `telnetd`, créé par le démon `inetd` de surveillance du réseau pour accepter une connexion à distance (*cf.* chapitre 9).

```
delacroix@lionne> pstree
init-+-atd
     |-automount
     |-cron
     |-httpd---3*[httpd]
     |-in.identd---in.identd---5*[in.identd]
     |-inetd---in.telnetd---login---bash---pstree
     |-kdm-+-X
     |     `-kdm
     |-keyserv
     |-kflushd
     |-klogd
     |-kpiod
     |-kswapd
     |-kupdate
     |-lpd
     |-master-+-pickup
     |        `-qmgr
     |-md_thread
     |-6*[mingetty]
     |-nscd---nscd---5*[nscd]
     |-portmap
     |-snmpd
     |-syslogd
     |-xconsole
     `-xntpd
```

2.4 Ordonnancement

2.4.1 Le rôle de l'ordonnancement

La fonction d'ordonnancement gère le partage du processeur entre les différents processus en attente pour s'exécuter, c'est-à-dire entre les différents processus qui sont dans l'état prêt. C'est la *politique d'ordonnancement* appliquée par le système d'exploitation qui détermine quel processus obtient le processeur parmi tous ceux qui sont prêts. L'attribution du processeur à un processus prêt constitue l'*opération d'élection*. Le processus passe dans l'état élu.

La politique d'ordonnancement mise en œuvre par le système d'exploitation peut ou non autoriser l'opération de *réquisition* du processeur. Par opération de réquisition, nous entendons le fait que le processeur puisse être retiré au processus élu alors que celui-ci dispose de toutes les ressources nécessaires à la poursuite de son exécution. Cette réquisition porte le nom de *préemption*. Un processus préempté quitte l'état élu pour passer dans l'état prêt (figure 2.8).

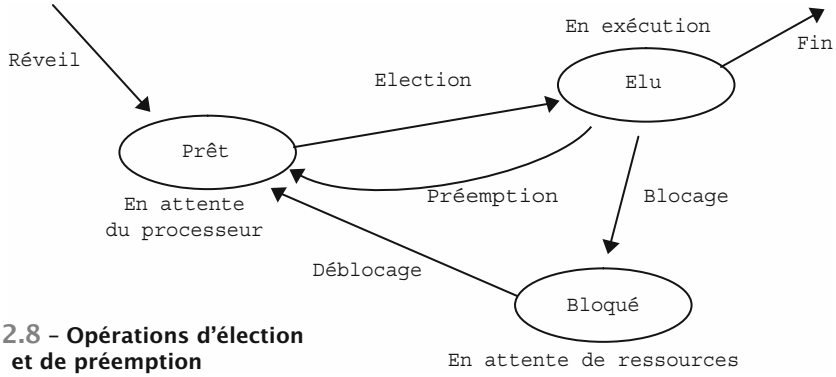

Figure 2.8 – Opérations d'élection et de préemption

Ainsi selon si l'opération de réquisition est autorisée ou non, l'ordonnancement est qualifié d'*ordonnancement préemptif* ou *non préemptif* :

- si l'ordonnancement est non préemptif, la transition de l'état élu vers l'état prêt est interdite : un processus quitte le processeur s'il a terminé son exécution ou s'il se bloque ;
- si l'ordonnancement est préemptif, la transition de l'état élu vers l'état prêt est autorisée : un processus quitte le processeur s'il a terminé son exécution, s'il se bloque ou si le processeur est réquisitionné.

Les processus prêts et les processus bloqués sont gérés dans deux files d'attentes distinctes qui chaînent leur PCB. Le module ordonnanceur (*scheduler*) trie la file des processus prêts de telle sorte que le prochain processus à élire soit toujours en tête de file. Le tri s'appuie sur un critère donné spécifié par la politique d'ordonnancement, comme par exemple l'âge du processus ou une priorité affectée au processus.

Dans un contexte multiprocesseur, le répartiteur (*dispatcher*) alloue un processeur parmi les processeurs libres à la tête de file de la liste des processus prêts et réalise donc l'opération d'élection. Dans un contexte monoprocesseur le rôle du répartiteur est très réduit puisqu'il n'a pas à choisir le processeur à allouer : il est alors souvent intégré dans l'ordonnanceur (figure 2.9).

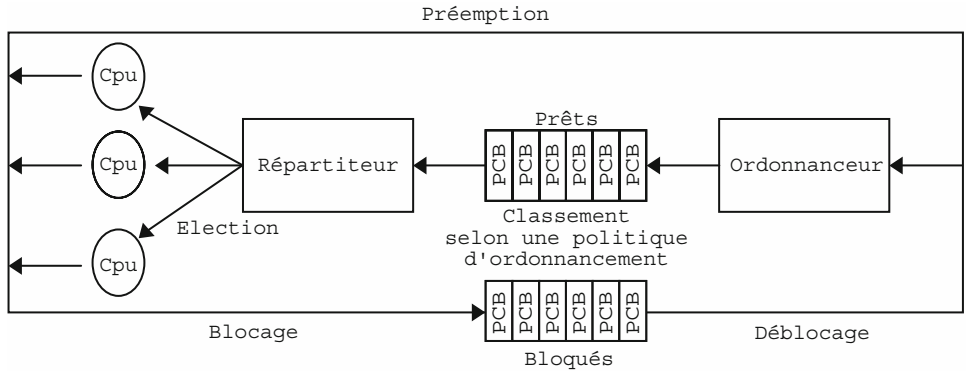

Figure 2.9 – Ordonnanceur et répartiteur

2.4.2 Les principaux algorithmes d'ordonnancement

Nous présentons à présent les politiques d'ordonnancement les plus courantes. Pour chacune des politiques évoquées, nous donnons un exemple sous forme d'un diagramme de Gantt. Un diagramme de Gantt permet de représenter l'assignation dans le temps du processeur aux processus. Le temps est représenté horizontalement, de manière croissante de la gauche vers la droite, à partir d'une origine arbitraire fixée à 0. Le temps est représenté par unité et pour chacune d'elles, figure le nom du processus affecté au processeur.

a) Politique Premier Arrivé, Premier Servi

La première politique est la politique du « Premier Arrivé, Premier Servi ». Les processus sont élus selon l'ordre dans lequel ils arrivent dans le système. Il n'y a pas de réquisition, c'est-à-dire qu'un processus élu s'exécute soit jusqu'à ce qu'il soit terminé, soit jusqu'à ce qu'il se bloque de lui-même (figure 2.10).

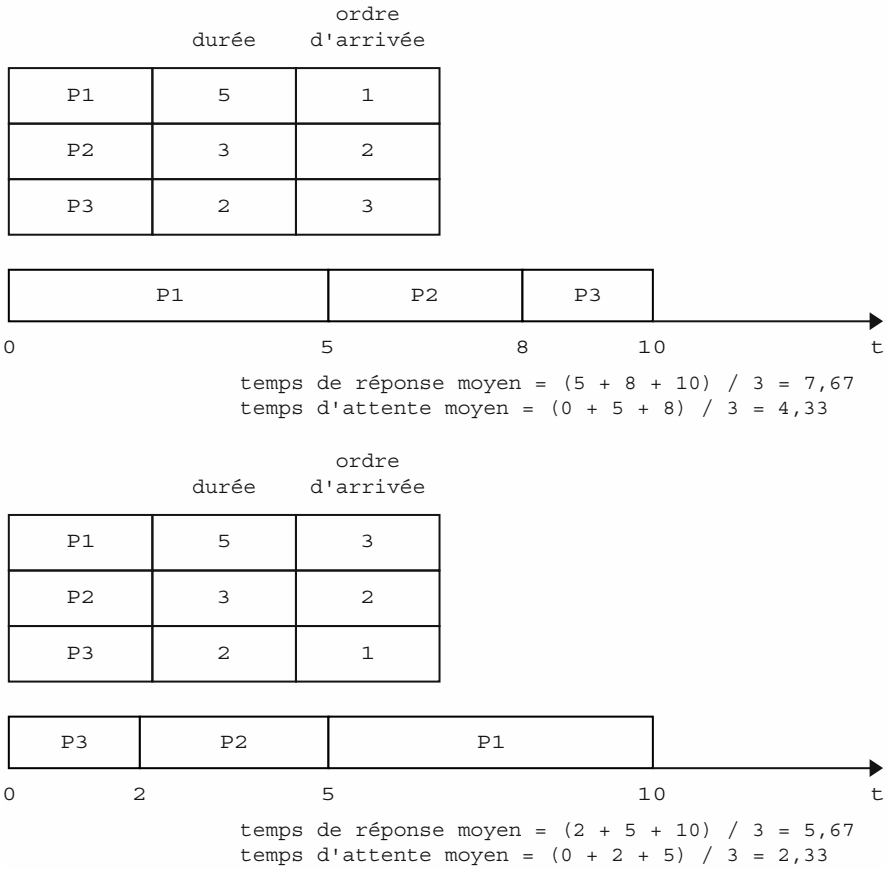

Figure 2.10 – Politique « Premier Arrivé, Premier Servi »

L'avantage de cette politique est sa simplicité. Son inconvénient majeur réside dans le fait que les processus de petit temps d'exécution sont pénalisés en terme de temps de réponse par les processus de grand temps d'exécution qui se trouvent avant eux dans la file d'attente des processus prêts.

b) Politique par priorité

Une autre politique très courante est la politique par priorité constante. Dans cette politique, chaque processus possède une priorité. La priorité est un nombre positif qui donne un rang à chaque processus dans l'ensemble des processus présents dans la machine. À un instant donné, le processus élu est le processus prêt de plus forte priorité. Selon les systèmes d'exploitation, les numéros de priorités les plus bas représentent les valeurs de priorité les plus élevées ou *vice versa*.

Cette politique se décline en deux versions selon si la réquisition est autorisée ou non. Si la réquisition est admise, alors le processus couramment élu est préempté dès qu'un processus plus prioritaire, donc possédant une priorité plus forte devient prêt (figure 2.11).

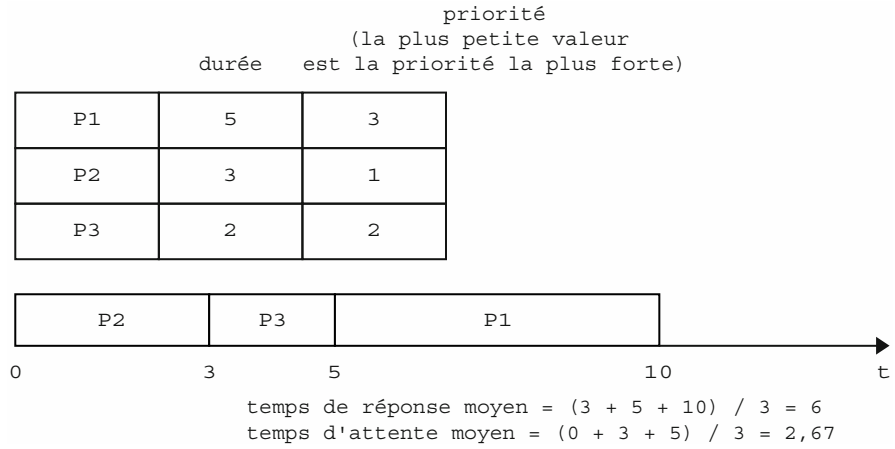

Figure 2.11 – Politique par priorité

Un inconvénient de cette politique est le risque de *famine* pour les processus de petite priorité si il y a de nombreux processus de haute priorité. Par famine, on entend le fait que les processus de plus forte priorité monopolisent le processeur au détriment des processus de plus faible priorité qui de ce fait ne peuvent pas s'exécuter (ils ont « faim » du processeur qu'ils n'obtiennent pas). On dit aussi qu'il y a *coalition* des processus de forte priorité contre les processus de faible priorité. Une solution à ce problème, appelé *extinction de priorité*, consiste à faire baisser la priorité des processus ayant obtenu le processeur.

c) Politique du tourniquet (round robin)

La politique par tourniquet est la politique mise en œuvre dans les systèmes dits en temps partagé. Dans cette politique, le temps est effectivement découpé en tranches

nommées quantums de temps. La valeur du quantum peut varier selon les systèmes entre 10 à 100 ms.

Lorsqu'un processus est élu, il s'exécute au plus durant un quantum de temps. Si le processus n'a pas terminé son exécution à l'issue du quantum de temps, il est préempté et il réintègre la file des processus prêts mais en fin de file. Le processus en tête de la file des processus prêts est alors à son tour élu pour une durée égale à un quantum de temps maximum (figure 2.12).

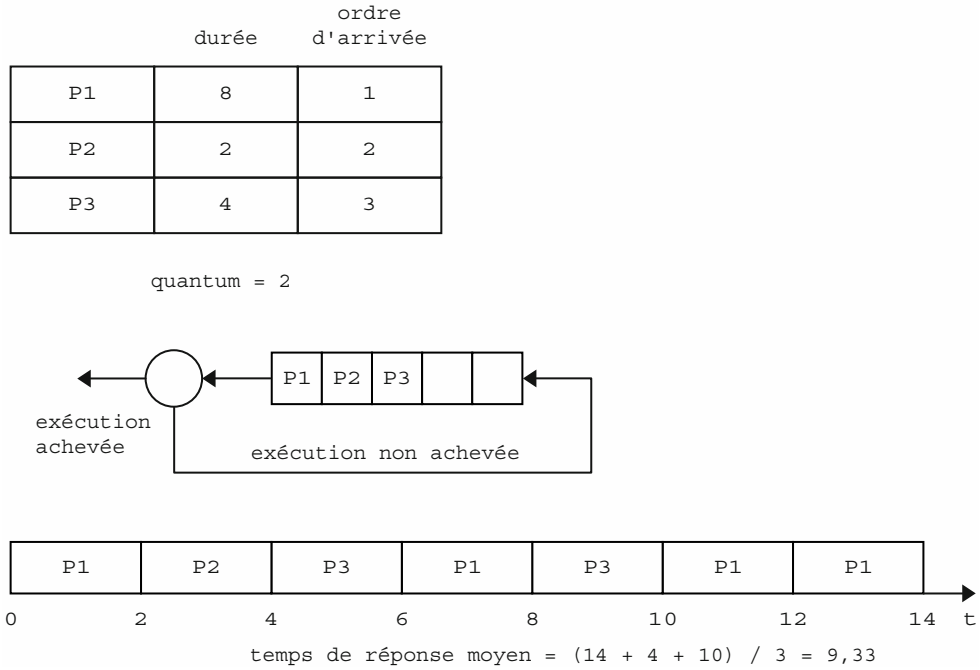

Figure 2.12 – Politique du tourniquet

La valeur du quantum constitue un facteur important de performance de la politique, dans la mesure où elle influe directement sur le nombre de commutations de contexte entre processus. En effet, plus la valeur du quantum sera faible, plus le nombre de commutation de contexte entre les processus à ordonnancer sera important et pourra de ce fait devenir non négligeable.

2.4.3 La fonction d'ordonnancement sous Linux

a) Politiques d'ordonnancement mises en œuvre

Le système Linux met en œuvre trois politiques d'ordonnancement différentes, conformément à la norme POSIX 1003.4. Les deux premières politiques `SCHED_FIFO` et `SCHED_RR` sont destinées aux processus dits « temps réel » et la troisième politique `SCHED_OTHER` est destinée aux processus classiques. Les processus dits temps réel, c'est-à-dire gérés par les politiques d'ordonnancement `SCHED_FIFO` et `SCHED_RR` sont

toujours plus prioritaires que les processus classiques. La politique d'ordonnancement appliquée à un processus est codée dans le champ `policy` de son bloc de contrôle.

L'ordonnancement réalisé par le noyau est découpé en périodes. Au début de chaque période, le système calcule les quantums de temps attribués à chaque processus, c'est-à-dire le nombre de ticks horloge durant lequel le processus peut s'exécuter. Une période prend fin lorsque l'ensemble des processus initialisés sur cette période a achevé son quantum.

Ordonnancement des processus temps réel

Les processus temps réel sont qualifiés par une priorité fixe dont la valeur évolue entre 1 et 99 (paramètre `rt_priority` du bloc de contrôle du processus) et sont ordonnancés soit selon la politique `SCHED_FIFO`, soit selon la politique `SCHED_RR`.

La politique `SCHED_FIFO` est une politique préemptive qui offre un service de type « premier arrivé, premier servi » entre processus de même priorité. À un instant t, le processus `SCHED_FIFO` de plus haute priorité le plus âgé est élu. Ce processus poursuit son exécution, soit jusqu'à ce qu'il se termine, soit jusqu'à ce qu'il soit préempté par un processus temps réel plus prioritaire devenu prêt. Dans ce dernier cas, le processus préempté réintègre la tête de file correspondant à son niveau de priorité.

La politique `SCHED_RR` est une politique du tourniquet à quantum de temps entre processus de même priorité. Dans ce cas, le processus élu est encore le processus `SCHED_RR` de plus forte priorité, mais ce processus ne peut poursuivre son exécution au-delà du quantum de temps qui lui a été attribué. Le quantum de temps (paramètre `counter` du bloc de contrôle du processus) correspond à un certain nombre de ticks horloge. Une fois ce quantum expiré, le processus élu est préempté et il réintègre la queue de la file correspondant à son niveau de priorité. Le processeur est alors alloué à un autre processus temps réel, qui est le processus temps réel le plus prioritaire.

Ordonnancement des processus classiques

Les processus classiques sont qualifiés par une priorité dynamique qui varie en fonction de l'utilisation faite par le processus des ressources de la machine et notamment du processeur. Cette priorité dynamique représente le quantum alloué au processus, c'est-à-dire le nombre de ticks horloge dont il peut disposer pour s'exécuter. Ainsi, un processus élu voit sa priorité initiale décroître en fonction du temps processeur dont il a disposé. Ainsi, il devient moins prioritaire que les processus qui ne se sont pas encore exécutés. Ce principe, appelé *extinction de priorité*, évite les problèmes de famine des processus de petite priorité.

La politique `SCHED_OTHER` mise en œuvre ici correspond à la politique d'ordonnancement des systèmes UNIX classiques. Plus précisément, la priorité dynamique d'un processus est égale à la somme entre un quantum de base (paramètre `priority` du bloc de contrôle du processus) et un nombre de ticks horloge restants au processus sur la période d'ordonnancement courante (paramètre `counter` du bloc de contrôle du processus). Un processus fils hérite à sa création du quantum de base de son père et de la moitié du nombre de ticks horloge restants de son père.

Le processus possède donc une priorité égale à la somme entre les champs `priority` et `counter` de son bloc de contrôle, et s'exécute au plus durant un quantum de temps égal à la valeur de son champ `counter`.

b) Primitives liées à la fonction d'ordonnancement

Plusieurs primitives sont offertes à l'utilisateur pour paramétrer l'ordonnancement et les attributs liés à l'ordonnancement de ses processus.

Modification ou récupération des paramètres d'ordonnancement

Les primitives `sched_setscheduler()`, `sched_getscheduler()`, `sched_setparam()` et `sched_getparam()` permettent respectivement de modifier ou de connaître la politique d'ordonnancement ou les attributs d'ordonnancement d'un processus. Parmi, les attributs d'ordonnancement, seule la priorité fixe d'un processus peut être modifiée par l'utilisateur. Les prototypes de ces fonctions déclarées dans le fichier `<sched.h>` sont :

- `int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param)` : modification de la politique d'ordonnancement `policy` du processus identifié par `pid`. Le paramètre `policy` prend une des trois valeurs `SCHED_FIFO`, `SCHED_RR` et `SCHED_OTHER`. Cette modification ne peut être réalisée qu'avec des droits équivalents à ceux du super-utilisateur. Si `pid` est nul, le processus courant est concerné.
- `int sched_getscheduler(pid_t pid)` : récupération de la politique d'ordonnancement associée au processus identifié par `pid`. Si `pid` est nul, le processus courant est concerné.
- `int sched_setparam(pid_t pid, const struct sched_param *param)` : modification des paramètres d'ordonnancement du processus identifié par `pid`. La structure `param` contient un seul champ, correspondant à la priorité statique du processus. Si `pid` est nul, le processus courant est concerné.
- `int sched_getparam(pid_t pid, const struct sched_param *param)` : récupération des paramètres d'ordonnancement du processus identifié par `pid`. Si `pid` est nul, le processus courant est concerné.

En cas d'échec, ces primitives renvoient la valeur – 1. La variable `errno` peut alors prendre les valeurs suivantes :

- `EFAULT`, `param` contient une adresse invalide ;
- `EINVAL`, `param` ou `policy` contiennent une valeur invalide ;
- `EPERM`, le processus n'a pas les droits nécessaires pour modifier ces paramètres d'ordonnancement ;
- `ESRCH`, le processus spécifié par `pid` n'existe pas.

Priorités et quantum

Les primitives `sched_get_priority_min()`, `sched_get_priority_max()` et `sched_rr_get_interval()` permettent respectivement de connaître les valeurs des priorités statiques minimale et maximale associées à une politique d'ordonnancement ainsi

que la valeur du quantum de temps associé à un processus dans le cadre d'un ordonnancement SCHED_RR. Les prototypes de ces fonctions déclarées dans le fichier <sys/wait.h> sont :

```
int sched_get_priority_min(int policy);
int sched_get_priority_max(int policy);
int sched_rr_get_interval(pid_t pid, struct timespec *interval);
```

La structure timespec qui permet de retourner la valeur du quantum est composée de deux membres de type entier long. Le premier membre tv_sec définit un nombre de secondes tandis que le second membre tv_nsec définit un nombre de nanosecondes.

En cas d'échec, ces primitives renvoient la valeur – 1. La variable errno peut alors prendre les valeurs suivantes :

- EFAULT, interval contient une adresse invalide ;
- ESRCH, le processus spécifié par pid n'existe pas.

Enfin, les primitives nice(), setpriority() et getpriority() permettent aux processus de modifier ou connaître leur priorité ou la priorité d'un groupe de processus. Dans tous les cas, seul un processus privilégié peut augmenter sa priorité. Plus précisément, la primitive nice(int inc) permet de baisser la priorité de base d'un processus de la valeur inc. Les primitives setpriority() et getpriority() permettent de baisser ou de connaître la priorité d'un processus, d'un groupe de processus ou de l'ensemble des processus d'un utilisateur. Leurs prototypes sont :

```
#include <unistd.h>
int nice(int inc);
#include <sys/wait.h>
int setpriority(int which, int who, int prio);
int getpriority(int which, int who);
```

Le paramètre which peut prendre trois valeurs différentes, qui impliquent une interprétation différente du paramètre who :

- si which = PRIO_PROCESS, alors who définit le PID du processus concerné ;
- si which = PRIO_PGRP, alors who définit le PGID du groupe du processus concerné ;
- si which = PRI_USER, alors who définit l'identifiant de l'utilisateur dont tous les processus sont concernés.

En cas d'échec, ces primitives renvoient la valeur – 1. La variable errno peut alors prendre les valeurs suivantes :

- EINVAL, which contient une valeur invalide ;
- EACCES, le processus n'a pas les droits nécessaires pour augmenter la priorité d'autres processus ;
- ESRCH, aucun processus ne correspond à la combinaison spécifiée par les paramètres which et who.

c) Un exemple

Nous donnons l'exemple suivant qui utilise les primitives liées à l'ordonnancement des processus. Dans cet exemple, le processus père affiche les valeurs de priorités minimum et maximum pour les politiques SCHED_FIFO et SCHED_RR, ainsi que la valeur du quantum pour la politique SCHED_RR. Le processus fils quant à lui affiche sa politique d'ordonnancement ainsi que sa priorité statique, puis il change de politique d'ordonnancement et de priorité. Afin que ce changement de politique puisse être réalisé, le programme suivant est exécuté par le super-utilisateur. Les traces d'exécution sont :

```
je suis le fils, ma priorité d'ordonnancement est: 0
je suis le fils, ma politique d'ordonnancement est SCHED_OTHER
je suis le fils, ma priorité d'ordonnancement est: 10
je suis le fils, ma nouvelle politique d'ordonnancement est SCHED_FIFO
Voici les priorités min et max de la politique SCHED_FIFO, 1, 99
Voici les priorités min et max de la politique SCHED_RR, 1, 99
Voici les la valeur du quantum de la politique SCHED_RR, 0 seconde,
150000000 nanosecondes
```

```
/*********************************************************************/
/*       Utilisation des primitives liées à l'ordonnancement      */
/*********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>
#include <sys/wait.h>
#include <errno.h>

main(){
pid_t ret,pid;
int politique, status, priorite;
struct timespec quantum;
struct sched_param param;

ret = fork();
if (ret == 0)
{
pid = getpid();
printf("je suis le fils, ma priorité d'ordonnancement est: %d\n",
                                  getpriority(PRIO_PROCESS, pid));
politique = sched_getscheduler (pid);
if (politique == SCHED_RR)
    printf("je suis le fils, ma politique d'ordonnancement est
                                                    SCHED_RR\n");
if (politique == SCHED_FIFO)
    printf("je suis le fils, ma politique d'ordonnancement est
                                                  SCHED_FIFO\n");
```

```
if (politique == SCHED_OTHER)
    printf("je suis le fils, ma politique d'ordonnancement est
                                                SCHED_OTHER\n");
param.sched_priority = 10;
setpriority(PRIO_PROCESS, pid, 10);
if (sched_setscheduler(pid, SCHED_FIFO, &param) == -1)
    perror ("pb setsceduler");
priorite= getpriority(PRIO_PROCESS, pid);
printf("je suis le fils, ma priorité d'ordonnancement est: %d\n",
                                                        priorite);
politique = sched_getscheduler (pid);
if (politique == SCHED_RR)
    printf("je suis le fils, ma nouvelle politique d'ordonnancement
                                                  est SCHED_RR\n");
if (politique == SCHED_FIFO)
    printf("je suis le fils, ma nouvelle politique d'ordonnancement
                                                 est SCHED_FIFO\n");
if (politique == SCHED_OTHER)
    printf("je suis le fils, ma nouvelle politique d'ordonnancement
                                                 est SCHED_OTHER\n");

}
else
{
printf ("Voici les priorités min et max de la politique SCHED_FIFO,
                  %d, %d\n",
             sched_get_priority_min(SCHED_FIFO),
             sched_get_priority_max (SCHED_FIFO));
printf ("Voici les priorités min et max de la politique SCHED_RR,
             %d, %d\n",
             sched_get_priority_min (SCHED_RR),
             sched_get_priority_max (SCHED_RR));
sched_rr_get_interval(getpid(),&quantum);
printf ("Voici les la valeur du quantum de la politique SCHED_RR, %d
      seconde, %d nanosecondes\n",quantum.tv_sec, quantum.tv_nsec);
wait(&status);
}
}
```

d) Implémentation

Les files d'ordonnancement

Le système Linux maintient deux types de files entre les blocs de processus :

- la file des processus actifs (*runqueue*) relie entre eux tous les blocs de contrôle des processus qui sont dans l'état `TASK_RUNNING` ;
- les files d'attente (*wait queues*) de processus relient entre eux tous les processus bloqués (états `TASK_INTERRUPTIBLE` et `TASK_UNINTERRUPTIBLE`) sur un même événement. Il y a ainsi autant de files d'attentes que d'événements sur lesquels les processus peuvent se bloquer.

Les paramètres des processus liés à l'ordonnancement

Le bloc de contrôle de processus contient les champs suivants liés à l'ordonnancement (ces champs ont été pour la plupart évoqués précédemment) :

- `need_resched` est un paramètre positionné par le noyau pour signaler d'un réordon-nancement est nécessaire avant de revenir en mode utilisateur ;
- `policy` spécifie la classe d'ordonnancement `SCHED_FIFO`, `SCHED_RR` ou `SCHED_OTHER` ;
- `rt_priority` spécifie la priorité statique d'un processus temps réel ;
- `priority` spécifie le quantum de base (priorité de base) d'un processus ;
- `counter` spécifie le nombre de ticks horloge restant au processus avant la fin de son quantum.

Le paramètre `counter` est régulièrement décrémenté lors de l'exécution de la partie basse `timer_bh`, activée par l'interruption horloge (*cf.* chapitre 1). Lorsque ce paramètre devient égal à 0, alors le processus courant a épuisé son quantum et un réordonnancement doit avoir lieu pour élire un nouveau processus. Le paramètre `need_resched` est positionné à vrai dans `timer_bh` et l'ordonnanceur est appelé au moment du retour vers le mode utilisateur.

La fonction d'ordonnancement schedule()

La fonction d'ordonnancement est implémentée par la fonction `schedule()` (fichier `kernel/sched.c`). La fonction `schelule()` est invoquée :

- lorsque le processus courant passe dans l'état bloqué car il a demandé une ressource non disponible ;
- lorsque le processus courant a épuisé son quantum de temps ;
- lorsqu'un processus temps réel plus prioritaire que le processus courant a été réveillé.

La fonction `schedule()` est exécutée par le processus courant lorsque celui-ci s'apprête à quitter le mode noyau pour repasser en mode utilisateur. Elle se décompose en trois parties :

- exécution des parties basses ayant été activées en mode noyau (*cf.* chapitre 1) ;
- gestion du processus courant qui va être préempté : si le processus courant est passé dans l'état bloqué, il est retiré de la file des processus qui sont dans l'état `TASK_RUNNING` et est placé dans la file d'attente correspondant à l'événement attendu. Si le processus est ordonnancé selon la politique `SCHED_RR` et que son quantum de temps est épuisé, le processus courant est placé en fin de file `TASK_RUNNING` avec un nouveau quantum ;
- détermination du processus prêt de plus haute priorité par appel à la fonction `goodness` détaillée par la suite ;
- commutation de contexte pour ordonnancer le processus choisi à l'étape précédente.

La fonction `goodness()` est chargée de réaliser le choix du prochain processus à élire. Pour cela, elle retourne à la fonction `schedule()` une valeur entière telle que :

- un processus temps réel reçoit la valeur de sa priorité statique augmentée de 1 000 ;
- un processus normal reçoit la valeur correspondant au nombre de cycles horloge durant lequel il doit encore s'exécuter (valeur contenue dans le champ `counter`), soit une valeur inférieure à 1 000.

L'ordonnanceur parcourt l'ensemble de la liste des processus prêts et pour chacun d'eux appelle la fonction `goodness()`. Le processus retenu est le processus pour lequel la valeur retournée par la fonction `goodness()` est la plus grande.

Nous donnons ici un code simplifié de la fonction `goodness()` ainsi que la boucle de parcours des processus prêts réalisée dans la fonction `schedule()`.

```
static inline int goodness (struct task_struct * prev, struct task_struct
*p, int this_cpu)
{
int weight;
if (p->policy != SCHED_OTHER) {
      weight = 1000 + p->rt_priority;
      goto out;
}
weight = p->counter;
if (!weight)
          goto out;

out:
    return weight;
}
/* appel de la fonction goodness dans la fonction schedule() */
while (p != &init_task) {
if (can_schedule(p)) {
int weight = goodness(prev, p, this_cpu);
if (weight > c)
c = weight, next = p;
}
p = p->next_run; }
```

La commutation de contexte

La commutation de contexte permettant l'élection du processus choisi est effectuée par la macro `switch_to()`. Cette fonction sauvegarde le contexte du processus courant, commute l'exécution du noyau sur la pile noyau du processus à élire puis restaure le contexte du processus à élire.

Exercices

2.1 Qu'avez-vous retenu ?

Pour chacune des questions suivantes, choisissez la ou les réponses qui vous semblent exactes.

Question 1 – Un processus est :

- ☐ a. un programme exécutable.
- ☐ b. une instance d'un programme exécutable.
- ☐ c. un contexte processeur.

Question 2 – Un processus Linux zombie est un processus :

- ☐ a. qui a perdu son père.
- ☐ b. qui a terminé son exécution en erreur.
- ☐ c. qui a terminé son exécution et attend la prise en compte de cette fin par son père.

Question 3 – Soient trois processus A, B, C soumis dans cet ordre, de priorité respective 4, 2, 9, avec le plus petit chiffre codant la priorité la plus forte. Quelles propositions sont exactes ?

- ☐ a. avec une politique FIFO, l'ordre d'exécution est C, B puis A.
- ☐ b. avec une politique par priorité fixe, l'ordre d'exécution est B, A, puis C.

Question 4 – Le processus A de priorité 7 s'exécute. Le processus B de priorité 5 se réveille. Le plus petit chiffre code la priorité la plus forte. Quelles sont les propositions justes ?

- ☐ a. B interrompt l'exécution de A car B est plus prioritaire et l'ordonnancement est préemptif.
- ☐ b. A continue son exécution car il est plus prioritaire et l'ordonnancement est préemptif.
- ☐ c. A continue son exécution car l'ordonnancement est non préemptif.
- ☐ d. B interrompt l'exécution de A car B est plus prioritaire et l'ordonnancement est non préemptif.

Question 5 – Le processus Linux crée un autre processus en utilisant la primitive :

- ☐ a. `fork()`
- ☐ b. `exit()`
- ☐ c. `wait()`
- ☐ d. `execl()`

Question 6 – Le processus Linux recouvre le code hérité lors de sa création par un autre code en utilisant la primitive :

- ☐ a. `fork()`

- ☐ b. `exit()`
- ☐ c. `wait()`
- ☐ d. `execl()`

Question 7 – Un processus Linux orphelin est un processus :

- ☐ a. qui a perdu son père.
- ☐ b. qui a terminé son exécution en erreur.
- ☐ c. qui a terminé son exécution et attend la prise en compte de cette fin par son père.

2.2 Processus Linux

Soit le programme `essai.c` suivant :

```
#include <stdio.h>

main()
{
int pid;
pid = fork();
if (pid == 0)
     {for(;;)
      printf ("je suis le fils\n");
      }
else
     {for(;;)
      printf("je suis le père\n");
     }
}
```

1) On compile ce programme pour générer un exécutable appelé `essai` dont on lance l'exécution. La commande `ps -l` permettant d'afficher les caractéristiques de l'ensemble des processus de l'utilisateur donne les éléments suivants :

```
F   S UID PID PPID C PRI NI ADDR SZ WCHAN TTY   TIME     MD
100 S  0  467 1    0 60  0 - 256 read_c   tty5  00:00:00 mingetty
100 S  0  468 1    0 60  0 - 256 read_c   tty6  00:00:00 mingetty
100 S  0  576 570  0 60  0 - 498 read_c   pts/0 00:00:00 cat
100 S  0  580 578  0 70  0 - 576 wait4    pts/1 00:00:00 bash
100 S  0  581 579  0 60  0 - 77  wait4    pts/2 00:00:00 bash
000 S  0  592 581  2 61  0 - 253 down_f   pts/2 00:00:01 essai
040 S  0  593 592  2 61  0 - 253 write_   pts/2 00:00:01 essai
100 R  0  599 580  0 73  0 - 652 -        pts/1 00:00:00 ps
```

Les champs S, PID, PPID et CMD codent respectivement l'état du processus (S pour *Stopped*, R pour *Running*), la valeur du PID et du PPID pour le processus et le nom du programme exécuté. On tape la commande `kill -9 593` qui entraîne la terminaison du processus dont le pid 593 est spécifié en argument. L'exécution de la commande `ps -l` donne à présent le résultat suivant. Que pouvez-vous dire à ce sujet ?

```
F   S UID PID PPID C PRI NI ADDR SZ WCHAN TTY   TIME     CMD
100 S  0  467 1    0 60  0 - 256 read_c   tty5  00:00:00 mingetty
100 S  0  468 1    0 60  0 - 256 read_c   tty6  00:00:00 mingetty
100 S  0  576 570  0 60  0 - 498 read_c   pts/0 00:00:00 cat
100 S  0  580 578  0 70  0 - 576 wait4    pts/1 00:00:00 bash
100 S  0  581 579  0 60  0 - 77  wait4    pts/2 00:00:00 bash
000 S  0  592 581  2 60  0 - 253 write_   pts/2 00:00:03 essai
444 Z  0  593 592  2 60  0 - 0   do_exi   pts/2 00:00:03 essai <zombie>
100 R  0  601 580  0 73  0 - 651 -        pts/1 00:00:00 ps
```

2) On lance de nouveau l'exécution du programme `essai`. La commande `ps -l` permettant d'afficher les caractéristiques de l'ensemble des processus de l'utilisateur donne les éléments suivants :

```
F   S UID PID PPID C PRI NI ADDR SZ WCHAN TTY   TIME     CMD
100 S  0  467 1    0 60  0 - 256 read_c   tty5  00:00:00 mingetty
100 S  0  468 1    0 60  0 - 256 read_c   tty6  00:00:00 mingetty
100 S  0  576 570  0 60  0 - 498 read_c   pts/0 00:00:00 cat
100 S  0  580 578  0 65  0 - 576 wait4    pts/1 00:00:00 bash
100 S  0  581 579  0 60  0 - 577 wait4    pts/2 00:00:00 bash
000 S  0  604 581  3 60  0 - 253 write_   pts/2 00:00:00 essai
040 S  0  605 604  2 60  0 - 253 down_f   pts/2 00:00:00 essai
100 R  0  606 580  0 76  0 - 649 -        pts/1 00:00:00 ps
```

On tape la commande `kill -9 604` qui entraîne la terminaison du processus dont le pid 604 est spécifié en argument. L'exécution de la commande `ps -l` donne à présent le résultat suivant. Que pouvez-vous dire à ce sujet ?

```
F   S UID PID PPID C PRI NI ADDR SZ WCHAN TTY   TIME     CMD
100 S  0  467 1    0  60 0 - 256 read_c   tty5  00:00:00 mingetty
100 S  0  468 1    0  60 0 - 256 read_c   tty6  00:00:00 mingetty
100 S  0  576 570  0  60 0 - 498 read_c   pts/0 00:00:00 cat
100 S  0  580 578  0  65 0 - 576 wait4    pts/1 00:00:00 bash
100 S  0  581 579  0  60 0 - 577 wait4    pts/2 00:00:00 bash
100 S  0  581 579  0  60 0 - 577 read_c   pts/2 00:00:00 bash
040 S  0  605 1    2  60 0 - 253 write_   pts/2 00:00:02 essai
100 R  0  608 580  0  73 0 - 649 -        pts/1 00:00:00 ps
```

3) Le programme `essai.c` est modifié :

```
#include <stdio.h>

main()
{
int pid;
pid = fork();
if (pid == 0)
     {for(;;)
      printf ("je suis le fils\n");
      }
else
     {
      printf("je suis le père\n");
      wait();
     }
}
```

On recompile ce programme pour générer un nouvel exécutable appelé `essai` dont on lance l'exécution. La commande `ps -l` permettant d'afficher les caractéristiques de l'ensemble des processus de l'utilisateur donne les éléments suivants :

```
F   S UID PID PPID C PRI NI ADDR SZ WCHAN TTY   TIME     CMD
100 S  0  467 1    0  60 0 - 256 read_c   tty5  00:00:00 mingetty
100 S  0  468 1    0  60 0 - 256 read_c   tty6  00:00:00 mingetty
100 S  0  576 570  0  60 0 - 498 read_c   pts/0 00:00:00 cat
100 S  0  580 578  0  70 0 - 577 wait4    pts/1 00:00:00 bash
100 S  0  581 579  0  60 0 - 577 wait4    pts/2 00:00:00 bash
000 S  0  627 581  0  60 0 - 253 wait4    pts/2 00:00:00 essai
040 S  0  628 627  3  61 0 - 253 write_   pts/2 00:00:00 essai
100 R  0  629 580  0  72 0 - 649 -        pts/1 00:00:00 ps
```

On tape la commande `kill -9 628` qui entraîne la terminaison du processus dont le pid 628 est spécifié en argument. L'exécution de la commande `ps -l` donne à présent le résultat suivant. Que pouvez-vous dire à ce sujet ?

```
F   S UID PID PPID C PRI NI ADDR SZ WCHAN TTY   TIME     CMD
100 S  0  467 1    0 60  0 - 256 read_c   tty5  00:00:00 mingetty
100 S  0  468 1    0 60  0 - 256 read_c   tty6  00:00:00 mingetty
100 S  0  576 570  0 60  0 - 498 read_c   pts/0 00:00:00 cat
100 S  0  580 578  0 70  0 - 577 wait4    pts/1 00:00:00 bash
100 S  0  581 579  0 60  0 - 577 wait4    pts/2 00:00:00 bash
100 R  0  31  580  0 73  0 - 648 -        pts/1 00:00:00 ps
```

2.3 Ordonnancement

Pour chacune des questions suivantes, vous choisissez **la ou les** bonnes réponses en justifiant votre choix, soit par un calcul, soit par une explication.

On considère quatre processus P1, P2, P3 et P4, soumis dans cet ordre, dont les caractéristiques sont les suivantes :

	Temps d'exécution	Priorité (plus petite valeur = plus grande priorité)
P1	8 unités	4
P2	8 unités	3
P3	10 unités	1
P4	4 unités	2

1) Avec un ordonnancement FIFO, les temps de réponse des quatre processus P1, P2, P3 et P4 sont respectivement :
   - ◇ 8, 16, 26, 30
   - ◇ 8, 18, 28, 32
   - ◇ 4, 14, 22, 30

2) Avec un ordonnancement par priorité, les temps de réponse des quatre processus P1, P2, P3 et P4 sont respectivement :
   - 8, 16, 26, 30
   - 10, 14, 22, 30
   - 8, 16, 20, 30
3) Avec un ordonnancement par quantum de temps, $Q = 2$, ordre initial FIFO, les temps de réponse des quatre processus P1, P2, P3 et P4 sont respectivement :
   - 16, 24, 26, 30
   - 24, 26, 30, 16
   - 24, 30,18,12

2.4 Ordonnancement

On considère un système composé de 3 processus Linux P1, P2 et P3 appartenant à la classe d'ordonnancement SCHED_RR. L'ordonnancement est effectué selon une méthode du tourniquet associée à une priorité. Ainsi les processus P1 et P2 sont de même priorité PRIO1 tandis que le processus P3 est de priorité PRIO2, PRIO2 < PRIO1. Le quantum de temps Q est égal à 10 ms.

Par ailleurs chaque processus peut effectuer des entrées-sorties avec un disque. Le disque est une ressource non requérable et l'ordre de service des requêtes disque est du type premier arrivé, premier servi (FIFO). Les profils des trois processus sont les suivants :

Processus P1	Processus P2	Processus P3
20 ms calcul 5 ms entrées-sorties 20 ms calcul 10 ms entrées-sorties 10 ms calcul	30 ms calcul 15 ms entrées-sorties 10 ms calcul 5 ms entrées-sorties 10 ms calcul	30 ms calcul 5 ms entrées-sorties 5 ms calcul

Représentez sur un chronogramme d'exécution où vous ferez apparaître les états prêt, élu (`task_running`) et bloqué (`task_interruptible`), l'exécution des trois processus et les changements d'états intervenant. Pour chacun des trois processus, donnez son temps de réponse.

2.5 Ordonnancement

On considère un système monoprocesseur de type Linux dans lequel les processus partagent un disque comme seul ressource autre que le processeur. Cette ressource n'est accessible qu'en accès exclusif et non requérable, c'est-à-dire qu'une commande disque lancée pour le compte d'un processus se termine normalement avant de pouvoir en lancer une autre. Un processus peut être en exécution, en attente d'entrées-sorties, en entrées-sorties ou en attente du processeur. Les demandes d'entrées-sorties sont gérées à l'ancienneté.

Dans ce système, on considère 4 processus P1, P2, P3, P4 pour lesquels on sait que :

- P1 et P2 sont des processus appartenant à la classe `SCHED_FIFO`. Dans cette classe, le processeur est donné au processus de plus haute priorité. Ce processus peut être préempté par un processus de la même classe ayant une priorité supérieure ;
- P3 et P4 sont des processus appartenant à la classe `SCHED_RR`. Dans cette classe, le processeur est donné au processus de plus haute priorité pour un quantum de temps égal à 10 ms. La politique appliquée est celle du tourniquet.

Les processus de la classe `SCHED_FIFO` sont toujours plus prioritaires que les processus de la classe `SCHED_RR`. Les priorités des processus sont égales à 50 pour le processus P1, 49 pour le processus P2, 49 pour le processus P3 et 49 pour le processus P4. La plus grande valeur correspond à la priorité la plus forte.

Les quatre processus ont le comportement suivant :

```
P1    Calcul pendant 40 ms
      Lecture disque pendant 50 ms
      Calcul pendant 30 ms
      Lecture disque pendant 40 ms
      Calcul pendant 10 ms
P2    Calcul pendant 30 ms
      Lecture disque pendant 80 ms
      Calcul pendant 70 ms
      Lecture disque pendant 20 ms
      Calcul pendant 10 ms
P3    Calcul pendant 40 ms
      Lecture disque pendant 40 ms
      Calcul pendant 10 ms
P4    Calcul pendant 100 ms
```

Établissez le chronogramme d'exécution des quatre processus en figurant les états prêt, élu, en attente d'entrées-sorties et en entrées-sorties.

2.6 Ordonnancement

On considère quatre processus P1, P2, P3 et P4 dont les caractéristiques sont les suivantes :

	Temps d'exécution	Priorité (plus petite valeur = plus grande priorité)
P1	6 unités	3
P2	4 unités	2
P3	14 unités	4
P4	2 unités	1

1) Les quatre processus sont présents à l'instant $t = 0$ dans la file des processus prêts dans l'ordre donné par la liste (P1 est la tête de liste). L'ordonnancement est en FIFO. Donnez l'ordre d'exécution des processus et le temps de réponse de chaque processus.
2) Les quatre processus sont présents à l'instant $t = 0$ dans la file des processus prêts dans l'ordre donné par les priorités. L'ordonnancement est par priorité. Donnez l'ordre d'exécution des processus et le temps de réponse de chaque processus.
3) Les quatre processus sont présents à l'instant $t = 0$ dans la file des processus prêts dans l'ordre donné par la liste (P1 est la tête de liste). L'ordonnancement est en tourniquet avec un quantum Q égal à deux unités. Donnez l'ordre d'exécution des processus et le temps de réponse de chaque processus.
4) Les quatre processus parviennent à présent à des instants différents dans la file d'attente des processus prêts. Ainsi :
   - Date arrivée de P1 = 0 ;
   - Date arrivée de P2 = 3 ;
   - Date arrivée de P3 = 2 ;
   - Date arrivée de P4 = 5.

   L'ordonnancement se fait selon un mode priorité préemptif. Représentez l'exécution des quatre processus et donnez leur temps de réponse.

2.7 Processus shell

On souhaite écrire un pseudo-code correspondant à l'algorithme suivi par le *shell*. Les étapes (simplifiées) du *shell* sont les suivantes :

- le *shell* lit une ligne de commande sur son entrée standard et l'interprète selon un ensemble de règles fixées (on ne s'intéresse pas au détail de cette analyse) ;
- le *shell* gère deux types de commandes :
  - les commandes en premier plan qui correspondent à des processus pouvant lire et écrire sur le terminal ;
  - les commandes en arrière-plan qui sont « détachées » du terminal, c'est-à-dire qui ne peuvent plus lire et écrire sur celui-ci.

Lorsqu'il traite une commande en premier plan, le *shell* attend la fin de celle-ci avant de prendre en compte une nouvelle commande. Au contraire lorsque le *shell* traite une commande en second plan, il n'attend pas la fin de celle-ci pour prendre en compte une nouvelle commande.

Écrivez la boucle d'exécution du *shell*.

2.8 Primitives `fork`, `exec`, `wait`

Soient les programmes suivants (*a*,*b*). Pour chacun d'eux, expliquez combien de processus sont créés et quels sont les affichages réalisés.

`calcul` est un exécutable qui additionne les deux entiers transmis en paramètres et affiche le résultat.

```
 (a)
Pour i de 1 à 2
{
  ret = fork();
  if (ret == 0) afficher (i) ;
  else afficher(ret);
}

(b)
Pour i de 1 à 2
{
  ret = fork();
  if (ret == 0) execl("/home/calcul", "calcul", “5”, “7”, NULL);
}
wait();
```

2.9 Processus zombie et orphelin

Écrivez un programme qui engendre deux fils, dont l'un deviendra zombie, et l'autre deviendra orphelin.

Des idées pour vous exercer à la programmation

Programme 1 – Un processus offre deux services aux utilisateurs : un premier service effectuant l'addition entre deux nombres entiers x et y et un second service effectuant la multiplication entre deux nombres flottants f et g.

Chaque service est rendu par un processus fils distincts *Fils1* et *Fils2*. Les étapes sont les suivantes :

– le père demande le type d'opération à effectuer (multiplication ou addition) et les deux valeurs sur lesquelles réaliser l'opération ;

– le père crée un fils pour réaliser ce service ;

– selon le type d'opération, le fils écrase le code de son père, soit par le code exécutable du service *Fils1*, soit par le code exécutable du service *Fils2*. Ce code exécutable effectue l'opération demandée, affiche le résultat puis se termine. Le fils renvoie une valeur de status nulle à son père si le service a été correctement rendu et sinon une valeur positive ;

– le père attend son fils et affiche la valeur renvoyée par celui-ci.

Programme 2 – Même énoncé mais en utilisant des threads pour rendre les deux services *Fils1* et *Fils2*.

Solutions

2.1 Qu'avez-vous retenu ?

Question 1 – Un processus est :

- ☐ b. une instance d'un programme exécutable.

Question 2 – Un processus Linux zombie est un processus :

- ☐ c. qui a terminé son exécution et attend la prise en compte de cette fin par son père.

Question 3 – Soient trois processus A, B, C soumis dans cet ordre, de priorité respective 4, 2, 9, avec le plus petit chiffre codant la priorité la plus forte. Quelles propositions sont exactes ?

- ☐ b. avec une politique par priorité fixe, l'ordre d'exécution est B, A, puis C.

Question 4 – Le processus A de priorité 7 s'exécute. Le processus B de priorité 5 se réveille. Le plus petit chiffre code la priorité la plus forte. Quelles sont les propositions justes ?

- ☐ a. B interrompt l'exécution de A car B est plus prioritaire et l'ordonnancement est préemptif.
- ☐ c. A continue son exécution car l'ordonnancement est non préemptif.

Question 5 – Le processus Linux crée un autre processus en utilisant la primitive :

- ☐ a. `fork()`

Question 6 – Le processus Linux recouvre le code hérité lors de sa création par un autre code en utilisant la primitive :

- ☐ d. `execl()`

Question 7 – Un processus Linux orphelin est un processus :

- ☐ a. qui a perdu son père.

2.2 Processus Linux

1) Les traces de la commande `ps` nous donne les informations suivantes sur les filiations entre processus : le processus interpréteur de commandes `bash` de pid 581 est le père du processus `essai` de pid 592 qui est lui-même le père du processus `essai` de pid 593. La commande `kill -9 593` tue donc le processus `essai` fils. Celui-ci devient zombie comme nous le montre la seconde exécution de la commande `ps` car son père n'effectue pas d'appel à la primitive `wait()`.
2) Les traces de la commande `ps` nous donne les informations suivantes sur les filiations entre processus : le processus interpréteur de commandes `bash` de pid 581 est le père du processus `essai` de pid 604 qui est lui-même le père du processus `essai` de pid 605. La commande `kill -9 604` tue donc le processus `essai` père. Le processus fils orphelin est maintenant adopté par le processus `init` (pid 1) comme le montre la seconde exécution de la commande `ps` où l'on voit que le PPID du processus `essai` fils est maintenant égal à 1.

3) Le code exécutable a été modifié de telle sorte qu'à présent, le processus `essai` père effectue un appel à la primitive `wait()`. La commande `kill -9 628` tue le processus `essai` fils. Tous les processus `essai` disparaissent à présent comme le montre la seconde exécution de la commande `ps`.

2.3 Ordonnancement

1) Les processus passent sur le processeur selon leur ordre de soumission soit P1, P2, P3 et P4. Les temps de réponse corrects sont donc ceux de l'item 1.
2) Les processus passent sur le processeur selon leur ordre de priorité croissante soit P3, P4, P2 et P1. Les temps de réponse corrects sont donc ceux de l'item 2.
3) Les processus passent sur le processeur selon leur ordre de soumission soit P1, P2, P3 et P4 et au plus pour un quantum de temps à chaque fois. La séquence d'exécution est donc P1 P2 P3 P4 P1 P2 P3 P4 (fin P4) P1 P2 P3 P1 (fin P1) P2 (fin P2) P3 P3 (fin P3). Les temps de réponse corrects sont donc ceux de l'item 2.

2.4 Ordonnancement

La solution est donnée par la figure 2.13.

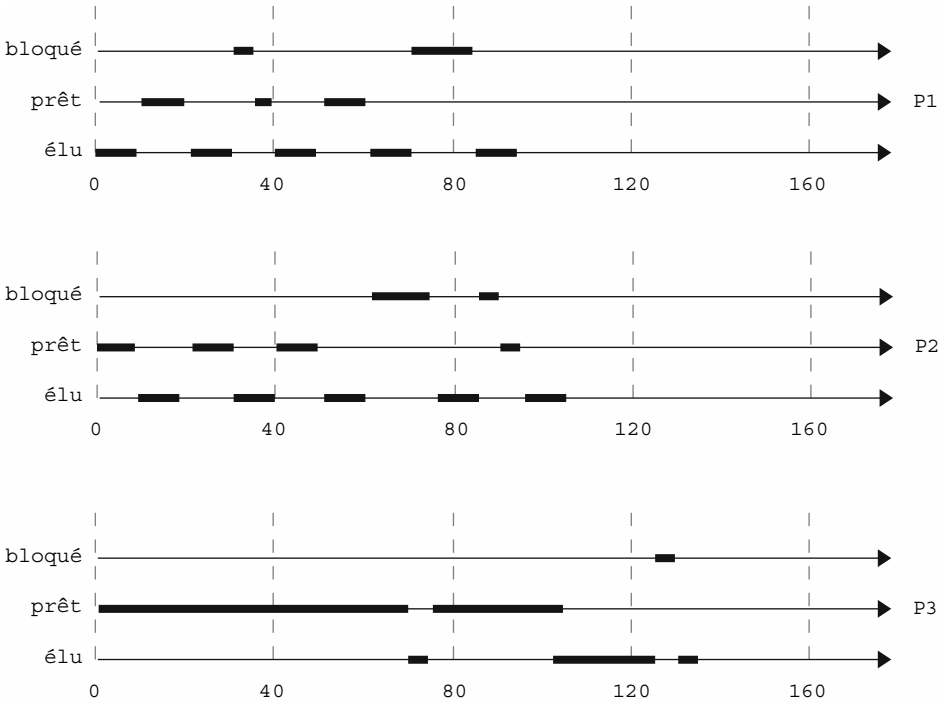

Figure 2.13 – **Ordonnancement Linux**

Le temps de réponse du processus P1 est égal à 95 ms.
Le temps de réponse du processus P2 est égal à 105 ms.

Le temps de réponse du processus P3 est égal à 140 ms.

2.5 Ordonnancement

La solution est donnée par la figure 2.14.

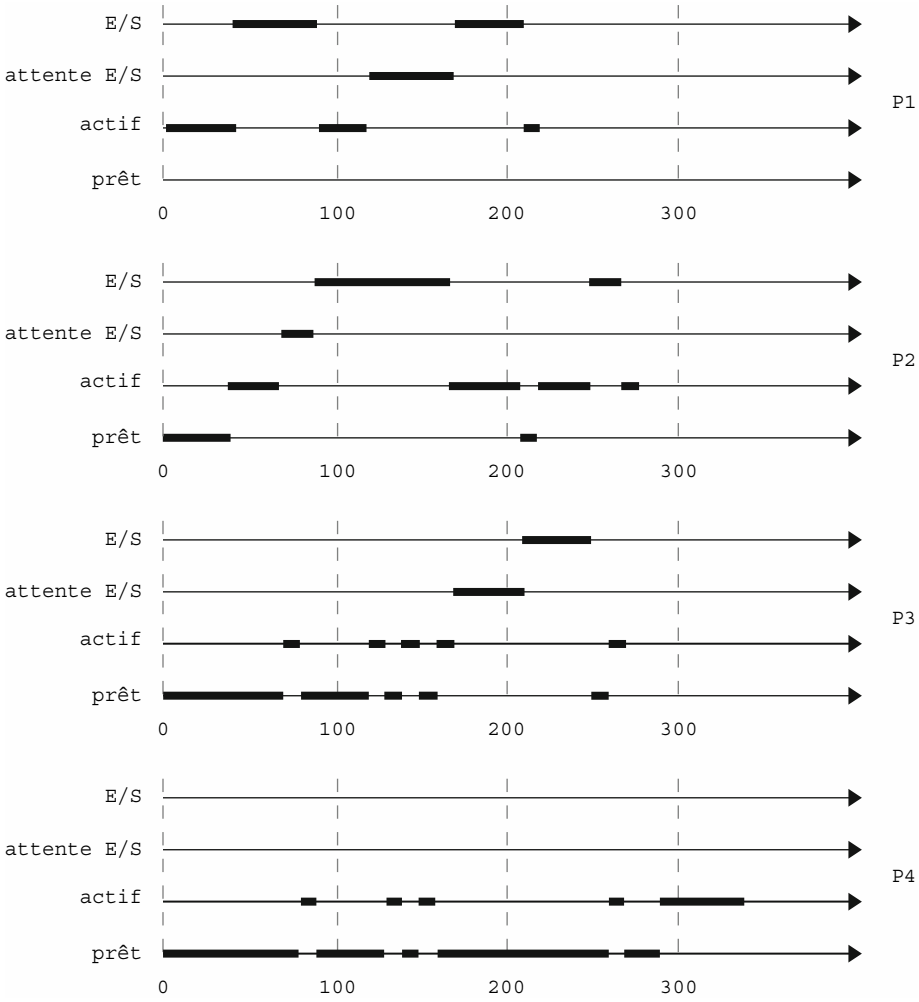

Figure 2.14 – **Ordonnancement Linux**

2.6 Ordonnancement

1) L'ordre de service est P1, P2, P3, P4. Le temps de réponse de P1 est égal à 6 unités, celui de P2 est égal à 10 unités, celui de P3 est égal à 24 unités, celui de P4 est égal à 26 unités.
2) L'ordre de service est P4, P2, P1, P3. Le temps de réponse de P1 est égal à 12 unités, celui de P2 est égal à 6 unités, celui de P3 est égal à 26 unités, celui de P4 est égal à 2 unités.

3) L'ordre de service est P1, P2, P3, P4, P1, P2, P3, P1, P3, P3, P3, P3, P3. Le temps de réponse de P1 est égal à 16 unités, celui de P2 est égal à 12 unités, celui de P3 est égal à 26 unités, celui de P4 est égal à 8 unités.
4) L'ordre de service est P1 (3 unités de temps), P2 (2 unités de temps), P4 (2 unités de temps), P2 (2 unités de temps), P1 (3 unités de temps), P3 (14 unités de temps). Le temps de réponse de P1 est égal à 12 unités, celui de P2 est égal à (9 – 3) unités, celui de P3 est égal à (26 – 2) unités, celui de P4 est égal à (7 – 5) unités.

2.7 Processus shell

```
for(;;)
{
lire une ligne de commande sur l'entrée standard;
analyser la commande lue pour la décomposer en commande, arg1, …, argn;
if (la ligne de commande contient &)
   ARRIERE_PLAN = TRUE;
else {
   ARRIERE_PLAN = FALSE;
pid = fork();   /* le shell crée un fils pour exécuter la
                   commande */
if (pid == 0)
   execlp ("commande", "arg1", .."argn");
else {
   if (ARRIERE_PLAN == FALSE)
       wait(); /* le père attend son fils si celui-ci n'est
                  pas en arrière-plan */
      }
      }
}
```

2.8 Primitives `fork`, `exec`, `wait`

```
(a)
Pour i de 1 à 2
{
  ret = fork();
  if (ret == 0) afficher (i) ;
  else afficher(ret);
}
```

Le processus père P1 crée un fils P2 qui affiche la valeur $i = 1$ et le père P1 affiche le pid du fils créé. Ensuite, chacun des deux processus P1 et P2, au second tour de boucle, créé à son tour un fils P3 et P4 qui affichent tous les deux $i = 2$. P1 et P2 affichent les pid de P3 et P4.

```
(b)
Pour i de 1 à 2
{
  ret = fork();
  if (ret == 0) execl("/home/calcul", "calcul", "5", "7", NULL);
}
wait();
```

Le processus père P1 crée un fils P2 qui recouvre son code avec le code de la fonction « calcul » et affiche la valeur 12. Au second tour de boucle, le processus père P1 crée un second fils P3 qui de nouveau recouvre son code avec le code de la fonction « calcul » et affiche la valeur 12.

2.9 Processus zombie et orphelin

```
#include <stdio.h>

main()
{
int pid;
pid = fork();
if (pid == 0)
      printf ("je suis le fils zombi\n");
else
     { pid = fork();
      if (pid == 0)
     { for(;;)
      printf ("je suis le fils orphelin\n");
      }
     }
}
```

Système de gestion de fichiers

3

PLAN

OBJECTIFS

- Ce chapitre présente les notions relatives au système de gestion de fichiers, qui a pour but la conservation des données en dehors de la mémoire centrale volatile.
- Après avoir exposé les notions importantes liées au système de gestion de fichiers, nous présentons le système de gestion de fichiers de Linux Ext2.
- Puis nous nous intéressons au système de gestion de fichiers virtuel supporté par Linux, qui permet à ce système de s'interfacer avec d'autres types de fichiers tels que ceux des systèmes DOS, Mac, IBM, etc.

3.1 Notions générales

La mémoire centrale de l'ordinateur est une mémoire volatile, c'est-à-dire que son contenu s'efface lorsque l'alimentation électrique de l'ordinateur est interrompue. Cependant, les programmes et les données stockés dans la mémoire centrale et en premier lieu le code et les données même du système d'exploitation, ont besoin d'être conservés au-delà de cette éventuelle coupure : c'est le rôle rempli par le *système de gestion de fichiers* qui assure la conservation des données sur un support de masse non volatile, tel que le disque dur, la disquette, le CD-ROM ou encore la bande magnétique. Le système d'exploitation offre à l'utilisateur une unité de stockage indépendante des propriétés physiques des supports de conservation : le fichier. Ce concept de fichier recouvre deux niveaux. D'une part, le *fichier logique* représente l'ensemble des données incluses dans le fichier telles qu'elles sont vues par l'utilisateur. D'autre part, le *fichier physique* représente le fichier tel qu'il est alloué physiquement sur le support de masse. Le système d'exploitation gère ces deux niveaux de fichiers et assure notamment la correspondance entre eux, en utilisant une structure de répertoire.

3.1.1 Le fichier logique

Le *fichier logique* correspond à la vue que l'utilisateur de la machine a de la conservation de ses données. Plus précisément, le fichier logique est :

- d'une part, un type de données standard défini dans les langages de programmation, sur lequel un certain nombre d'opérations spécifiques peuvent être réalisées. Ces opérations sont les opérations de création, ouverture, fermeture et destruction de fichier. Dans le programme, le fichier est identifié par un nom. Les opérations de création ou d'ouverture du fichier logique effectuent la liaison du fichier logique avec le fichier physique correspondant sur le support de masse. Au contraire, les opérations de fermeture et de destruction rompent cette liaison ;
- d'autre part, un ensemble d'enregistrements ou articles. Un enregistrement est un type de données regroupant des données de type divers liées entre elles par une certaine sémantique inhérente au programme qui les manipule. L'enregistrement constitue pour le programme une unité logique de traitement. Les enregistrements du fichier logique sont accessibles par des opérations spécifiques de lecture ou d'écriture que l'on appelle les fonctions d'accès.

Les modes d'accès les plus courants sont l'*accès séquentiel*, l'*accès indexé* et l'*accès direct*. Ils définissent la sémantique des opérations de lecture et d'écriture d'enregistrements au niveau du fichier. Selon le mode d'accès associé à un fichier, celui-ci peut être accessible en lecture seule, en écriture seule ou en lecture et écriture simultanée :

- le mode d'accès séquentiel traite les enregistrements d'un fichier dans l'ordre où ils se trouvent dans ce fichier, c'est-à-dire les uns à la suite des autres. Dans ce contexte, une opération de lecture d'un enregistrement délivre l'enregistrement courant et se positionne sur le suivant. L'opération d'écriture d'un nouvel enregistrement place obligatoirement cet enregistrement en fin de fichier. Ce mode d'accès est très simple et il découle naturellement du modèle de la bande magnétique. Avec ce mode, un fichier est soit accessible en lecture seule, soit en écriture seule ;
- le mode d'accès indexé encore appelé accès aléatoire permet d'accéder directement à un enregistrement quelle que soit sa position dans le fichier. Un champ commun à tous les enregistrements sert de clé d'accès. Une structure d'accès ajoutée aux données du fichier permet de retrouver un enregistrement en fonction de la valeur du champ d'accès. Dans la fonction de lecture, la valeur du champ d'accès recherché est spécifiée et l'opération de lecture délivre directement l'enregistrement pour lequel la clé correspond. La fonction d'écriture place le nouvel enregistrement en fin de fichier et met à jour la structure de données permettant de le retrouver ultérieurement. Avec ce mode, un fichier est soit accessible en lecture seule, soit en écriture seule, soit tout à la fois en lecture et en écriture ;

- dans le mode d'accès direct encore appelé accès relatif, l'accès à un enregistrement se fait en spécifiant sa position relative par rapport au début du fichier. Ce mode d'accès constitue un cas particulier de l'accès aléatoire pour lequel la clé d'accès est la position de l'enregistrement dans le fichier.

3.1.2 Le fichier physique

Le fichier physique correspond à l'entité allouée sur le support permanent et contient physiquement les enregistrements définis dans le fichier logique. Nous prendrons ici comme support de mémoire secondaire le disque dur.

a) Structure du disque dur

Un disque est constitué d'un ensemble de *plateaux* (jusqu'à 20), empilés verticalement sur un même axe, formant ainsi ce que l'on appelle une pile de disques. Chaque plateau est composé d'une ou deux *faces*, divisées en *pistes* qui sont des cercles concentriques. Selon les modèles, le nombre de pistes par face varie de 10 à plus de 1 000. Chaque piste est elle-même divisée en *secteurs* (de 4 à 32 secteurs par piste), le secteur constituant la plus petite unité de lecture/écriture sur le disque. La taille d'un secteur varie généralement entre 32 à 4 096 octets, avec une valeur courante de 512 octets. On parle également de cylindre, celui-ci étant constitué de l'ensemble des pistes de même rayon sur l'ensemble des plateaux. Une adresse sur le disque est donc de la forme (face, piste ou cylindre, secteur). L'opération consistant à créer à partir d'un disque vierge, l'ensemble des pistes et des secteurs s'appelle l'*opération de formatage*.

L'accès à une face d'un disque s'effectue par l'intermédiaire d'un bras, supportant une tête de lecture/écriture. Le bras avance ou recule pour se positionner sur la piste à lire/écrire tandis que le disque tourne.

La plus petite unité accessible physiquement sur le disque est donc le secteur. Pour optimiser les opérations de lecture et écriture, les secteurs sont regroupés en bloc. Un bloc est constitué de plusieurs secteurs et constitue la plus petite unité d'échange entre le disque et la mémoire centrale. Sa taille dépend du périphérique d'entrée-sortie et est fixée par le matériel. La figure 3.1 illustre la structure d'un disque.

b) Méthodes d'allocation de la mémoire secondaire

Les enregistrements composant le fichier logique doivent être écrits dans les secteurs composant les blocs du disque, pour former ainsi le fichier physique correspondant au fichier logique.

Le fichier physique est donc constitué d'un ensemble de blocs physiques qui doivent être alloués au fichier. Différentes méthodes d'allocation de la mémoire secondaire ont été définies. Ce sont principalement :

- la méthode de l'allocation contiguë ;
- la méthode de l'allocation par zones ;

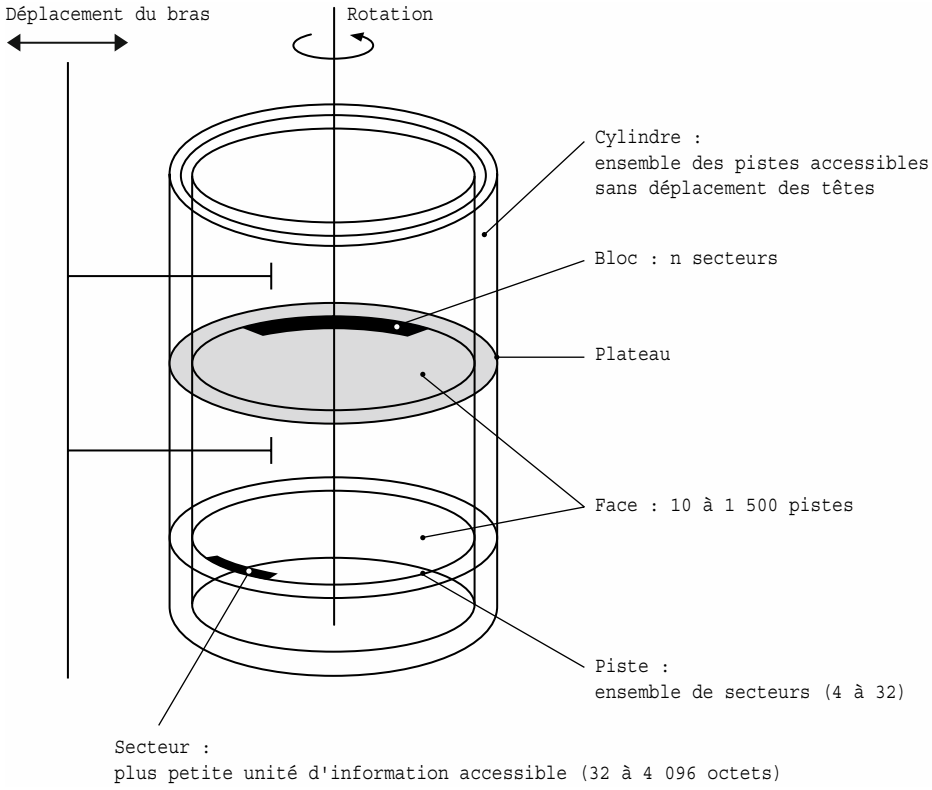

Figure 3.1 – Structure du disque dur

- la méthode de l'allocation par blocs chaînés ;
- la méthode de l'allocation indexée.

Par ailleurs, pour pouvoir allouer des blocs aux fichiers, il faut connaître à tout moment l'ensemble des blocs libres et donc gérer l'espace libre sur le disque.

Allocation contiguë

Cette allocation exige qu'un même fichier occupe un ensemble de blocs physiques contigus. Cette méthode a l'avantage de la simplicité mais elle demande à ce que lors de la création d'un fichier la taille finale de celui-ci soit évaluée afin qu'un espace suffisant puisse être réservé. Il en découle souvent une sous-utilisation de l'espace disque.

L'allocation d'un fichier requiert de trouver un espace libre sur le disque suffisamment grand c'est-à-dire dont le nombre de blocs est au moins égal au nombre de blocs du fichier. Deux méthodes principales permettent de choisir un trou suffisant parmi tous ceux existants :

- la méthode *First Fit* choisit le premier trou qui convient c'est-à-dire dont la taille est au moins égale à celle du fichier ;

- la méthode *Best Fit* choisit le trou dont la taille est la plus proche de celle du fichier, c'est-à-dire le trou dont la taille est au moins égale à celle du fichier et qui génère le plus petit résidu.

Ainsi, sur la figure 3.2, l'allocation du fichier `fich_5` d'une taille évaluée maximale à 4 blocs attribuera les blocs libres 4, 5, 6 et 7 dans le cas d'une stratégie de type *First Fit*, et les blocs libres 13, 14, 15 et 16 dans le cas d'une stratégie *Best Fit*.

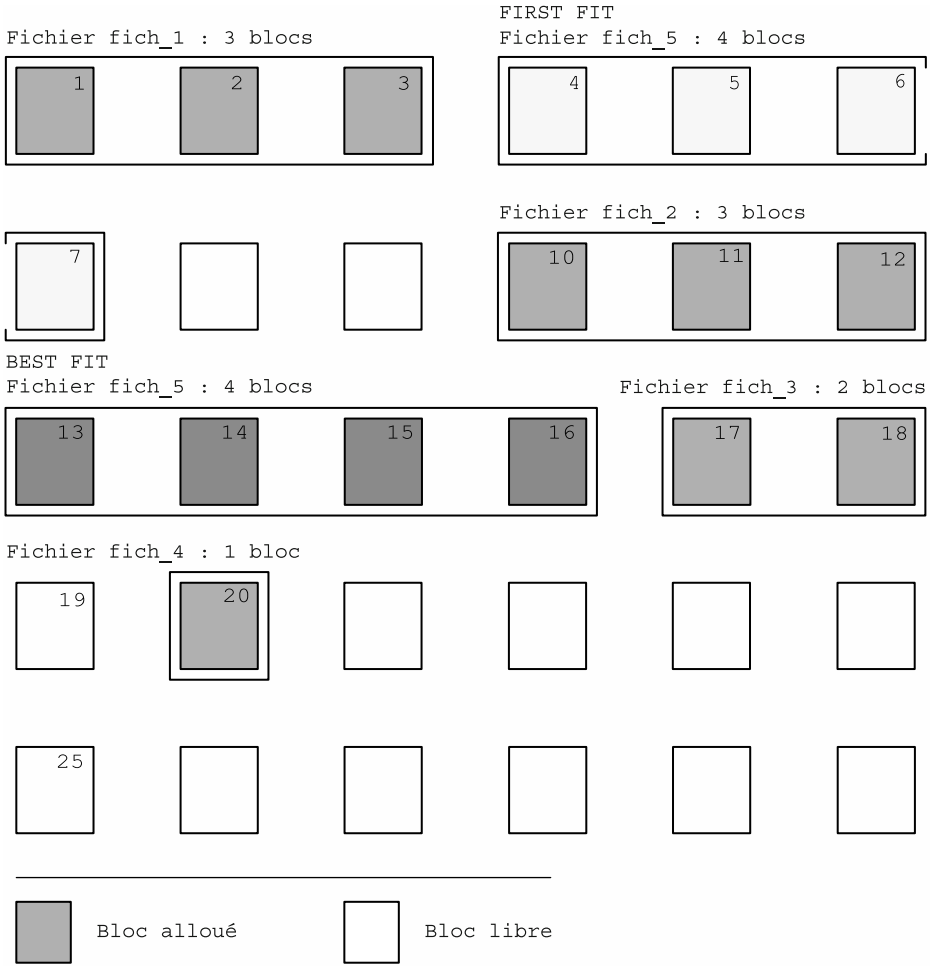

Figure 3.2 – Allocation contiguë

Il découle de cette méthode d'allocation des problèmes de *fragmentation* de l'espace disque qui se traduisent par une difficulté croissante pour trouver des espaces libres suffisants à une nouvelle allocation. Il faut alors avoir recours à une *opération de compactage* pour regrouper les espaces libres dispersés et insuffisants en un seul

espace libre exploitable. Cette opération est évidemment très coûteuse puisqu'elle nécessite le déplacement des blocs du disque, c'est-à-dire la lecture de chaque bloc et sa réécriture.

Une dernière difficulté est celle de l'extension d'un fichier si les blocs disques voisins ne sont pas libres. Ainsi, sur la figure 3.2, l'utilisateur souhaite étendre le fichier fich_1 avec un nouveau bloc. Cependant, dans le cadre d'une allocation de type *First Fit*, le bloc voisin au dernier bloc de `fich_1` est alloué au fichier `fich_5`. Une solution peut être alors de déplacer le fichier déjà existant dans une zone libre plus importante autorisant de ce fait son extension, mais cette solution est excessivement coûteuse. La solution plus généralement utilisée dans ce cas est de générer une erreur.

Cette méthode d'allocation supporte les modes d'accès séquentiel et direct. Par ailleurs, elle offre de bonnes performances pour l'accès aux différents blocs d'un même fichier, car leur contiguïté minimise le nombre de repositionnements nécessaires des têtes de lecture/écriture. Cependant, elle aboutit à une mauvaise utilisation de l'espace disque qui fait qu'elle tend de plus en plus à être abandonnée.

Allocation par zones

La méthode d'allocation par zones constitue une variante de l'allocation contiguë, qui autorise un fichier à être constitué de plusieurs zones physiques distinctes. Une zone doit être allouée dans un ensemble de blocs contigus mais les différentes zones composant le fichier peuvent être dispersées sur le support de masse. Il en résulte une extension plus facile d'un fichier. Lors de la création du fichier, la première zone appelée *zone primaire* est allouée et les zones suivantes appelées *zones secondaires* sont allouées au fur et à mesure de l'extension du fichier. La taille de la zone primaire et celle des zones secondaires sont généralement définies au moment de la création du fichier. Le nombre de zones secondaires autorisées pour un fichier est souvent limité. La figure 3.3 illustre cette méthode d'allocation.

La méthode d'allocation par zones diminue les performances d'accès aux blocs d'un même fichier puisque le passage d'une zone à un autre non contiguë peut nécessiter un repositionnement important du bras. Elle minimise par ailleurs les problèmes de fragmentation externe sans cependant les résoudre complètement.

Allocation par blocs chaînés

Dans cette méthode, un fichier est constitué comme une liste chaînée de blocs physiques, qui peuvent être dispersés n'importe où sur le support de masse. Chaque bloc contient donc l'adresse du bloc suivant dans le fichier.

Étendre un fichier est alors simple : il suffit d'allouer un nouveau bloc physique puis de le chaîner au dernier bloc physique du fichier. Ainsi, il n'est plus nécessaire de connaître à la création du fichier, la taille maximale de celui-ci. Par ailleurs, les problèmes de fragmentation externe disparaissent.

Cette méthode d'allocation souffre malheureusement de deux inconvénients. Le premier est que le seul mode d'accès utilisable est le mode d'accès séquentiel. En effet, accéder à un enregistrement donné du fichier nécessite de parcourir la liste

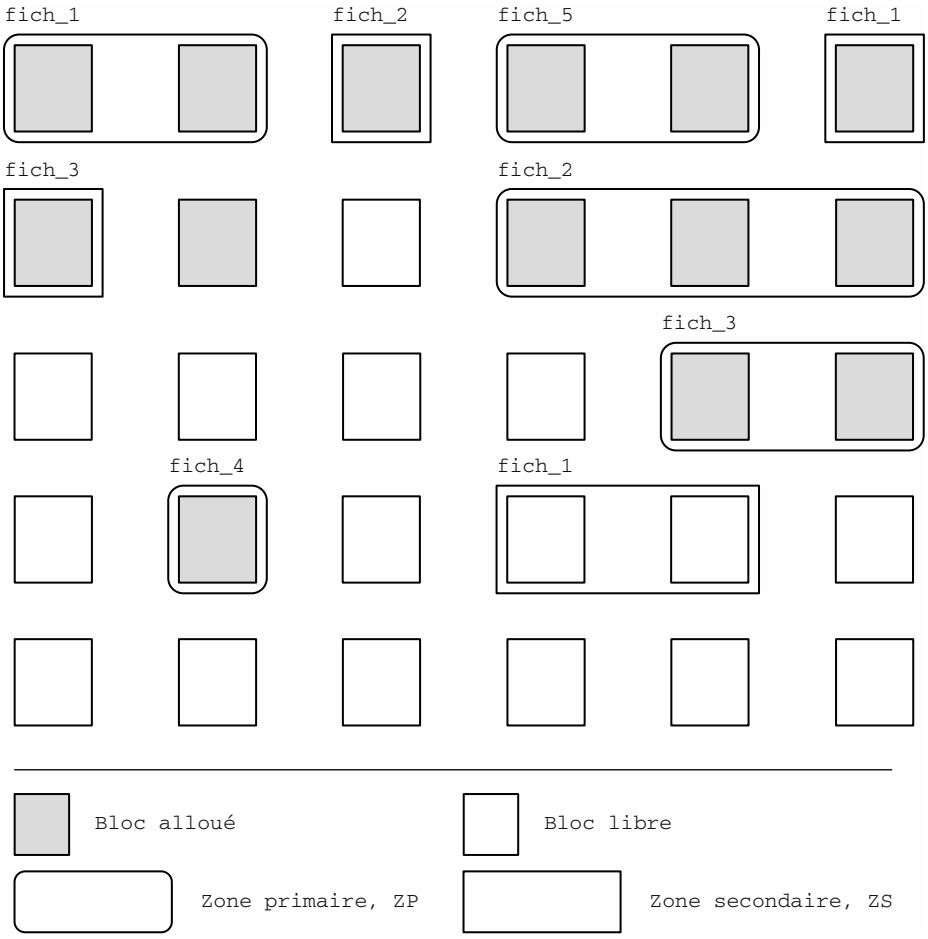

Figure 3.3 – Allocation par zones

chaînée des blocs constituant ce fichier, donc de lire les uns à la suite des autres chacun des blocs du fichier pour connaître l'adresse du suivant. Le second inconvénient est la place occupée dans chaque bloc par le chaînage de la liste, fonction de la taille des adresses physiques des blocs. La figure 3.4 illustre cette méthode d'allocation.

Allocation indexée

L'allocation indexée vise à supprimer les deux inconvénients de la méthode d'allocation chaînée. Dans cette méthode, toutes les adresses des blocs physiques constituant un fichier sont rangées dans une table appelée *index*, elle-même contenue dans un bloc du disque. À la création d'un fichier, l'index est créé avec toutes ses entrées initialisées à vide et celles-ci sont mises à jour au fur et à mesure de l'allocation des blocs au fichier. Ce regroupement de toutes les adresses des blocs constituant un fichier dans une même table permet de réaliser des accès directs à chacun de ces

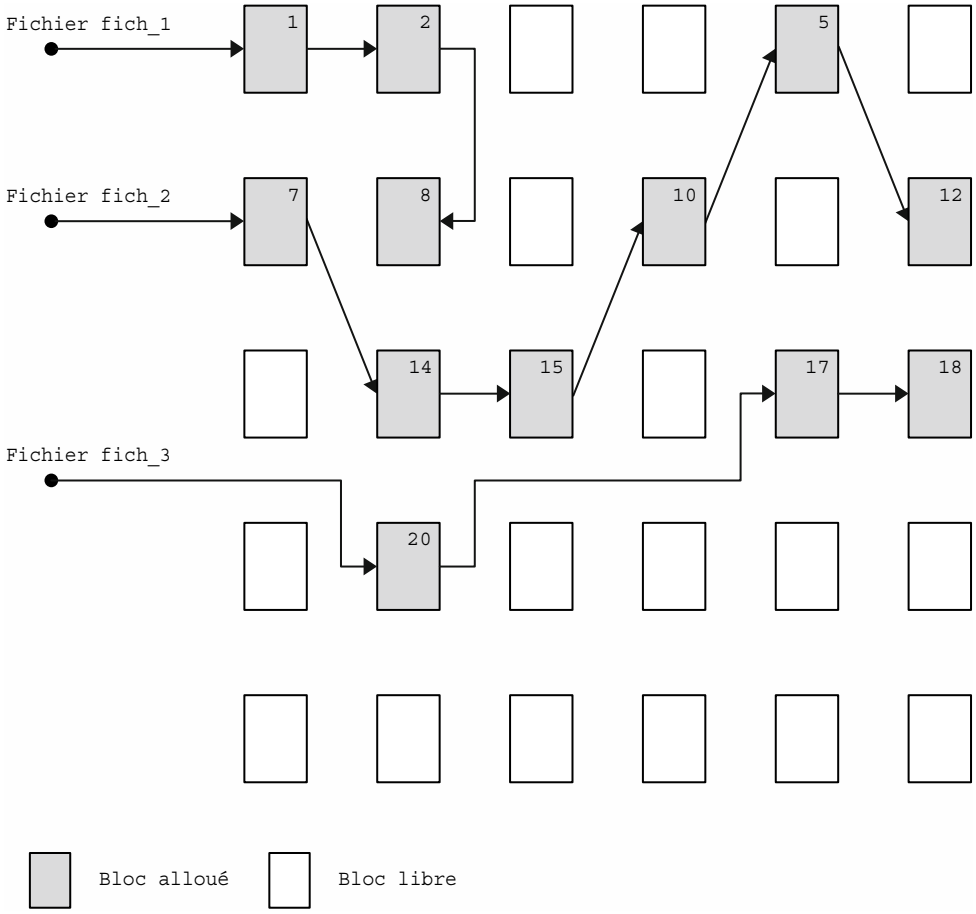

Figure 3.4 – **Allocation par blocs chaînés**

blocs. Par ailleurs, les blocs alloués au fichier contiennent à présent exclusivement des données du fichier. La figure 3.5 illustre cette méthode d'allocation.

Un problème est inhérent cependant à la taille de la table d'index qui est conditionnée par celle d'un bloc physique et par le nombre de blocs existants sur le disque. Si le bloc est grand, le nombre d'entrées utilisées dans le bloc d'index peut être faible ce qui conduit à un problème de fragmentation interne et à une perte de place, notamment en ce qui concerne l'allocation des fichiers de petite taille. Au contraire, le nombre d'entrées disponibles dans le bloc d'index plus se révéler être insuffisant vis-à-vis du nombre de blocs requis pour le fichier.

Dans ce dernier cas, une solution est d'utiliser un index multiniveaux. Le premier bloc d'index ne contient pas directement des adresses de blocs de données, mais il contient des adresses de blocs d'index, qui eux contiennent les adresses des blocs de données du fichier.

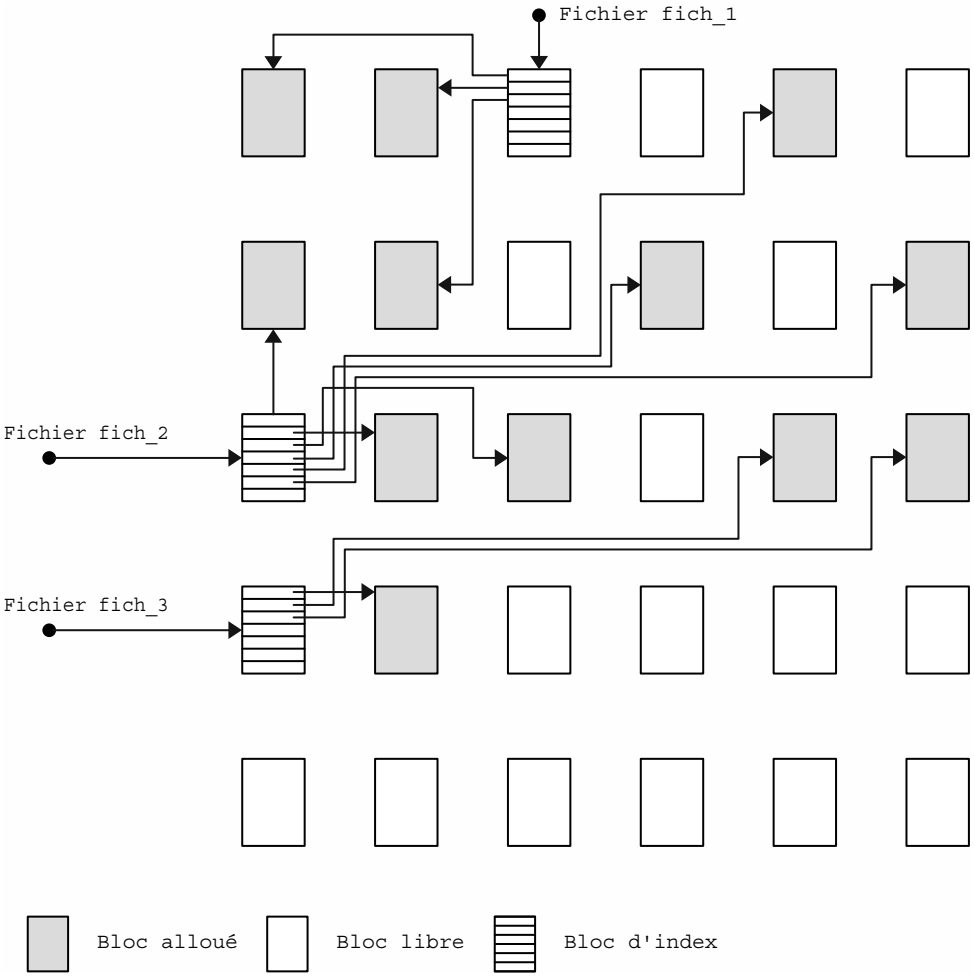

Figure 3.5 – **Méthode d'allocation indexée**

Ainsi, pour lire un bloc de données d'un fichier, le système doit d'abord lire le premier bloc d'index, puis un second bloc d'index avant de lire le bloc de données lui-même. Il s'ensuit une évidente perte de performances qui s'accroît encore si ce schéma est étendu à 3 ou 4 niveaux (figure 3.6).

Gestion de l'espace libre

Le système maintient une liste d'espace libre, qui mémorise tous les blocs disque libres, c'est-à-dire les blocs non alloués à un fichier.

Lors de la création ou de l'extension d'un fichier, le système recherche dans la liste d'espace libre la quantité requise d'espace et l'alloue au fichier. L'espace alloué est supprimé de la liste. Lors de la destruction d'un fichier, l'espace libéré est intégré à la liste d'espace libre.

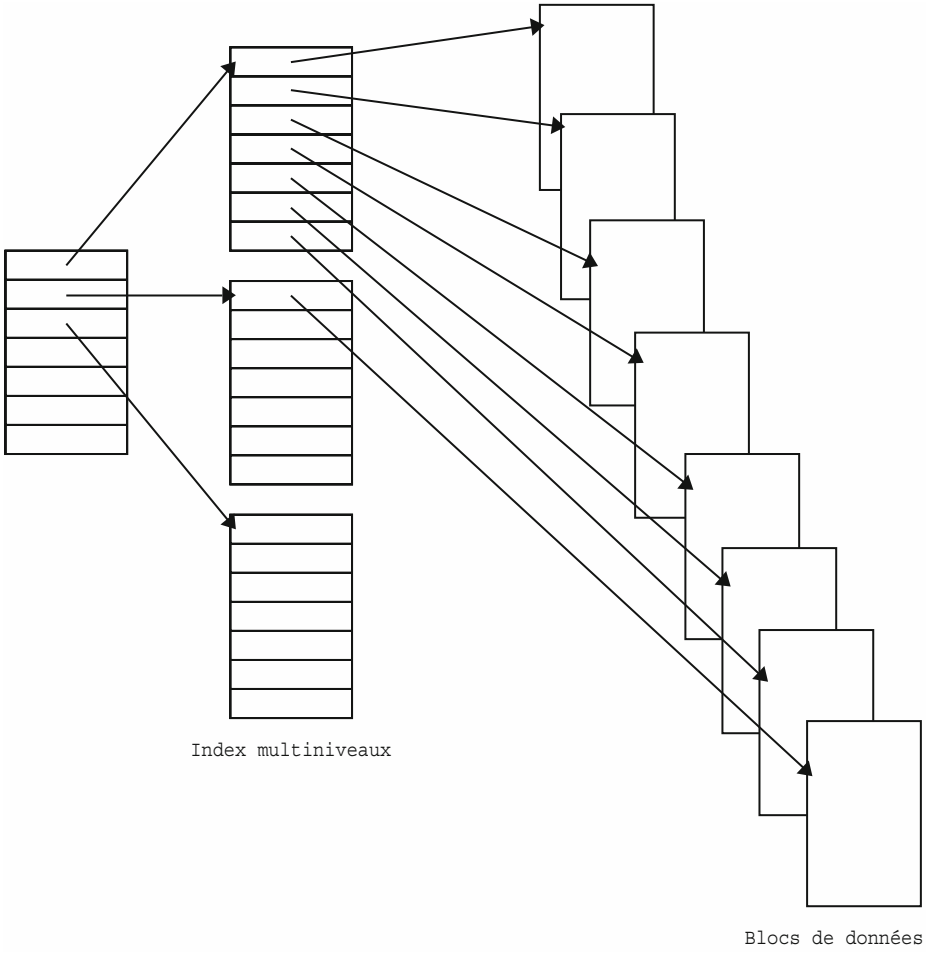

Figure 3.6 – **Index multiniveaux**

Il existe différentes représentations possibles de l'espace libre, les principales sont :

- représentation de l'espace libre sous forme d'un vecteur de bits ;
- représentation de l'espace libre sous forme d'une liste chaînée des blocs libres.

Gestion de l'espace libre par un vecteur de bits

Dans cette méthode, l'espace libre sur le disque est représenté par un vecteur binaire dans lequel chaque bloc est figuré par un bit. La longueur de la chaîne binaire est donc égale au nombre de blocs existants sur le disque. Dans cette chaîne, un bit à 0 indique que le bloc correspondant est libre. Au contraire, un bit à 1 indique que le bloc correspondant est alloué. Ainsi, le vecteur de bits correspondant à l'état du disque de la figure 3.4 est 11001011010101101101000000000.

Gestion de l'espace libre par liste chaînée

Dans cette méthode, l'espace libre sur le disque est représenté par une liste chaînée de l'ensemble des blocs libres du disque. Avec cette méthode, la recherche sur disque de n blocs consécutifs peut nécessiter le parcours d'une grande partie de la liste chaînée. Une variante de la méthode consiste à indiquer pour chaque premier bloc d'une zone libre, le nombre de blocs libres constituant la zone, puis l'adresse du premier bloc de la zone libre suivante. La figure 3.7 illustre cette méthode de représentation de l'espace libre.

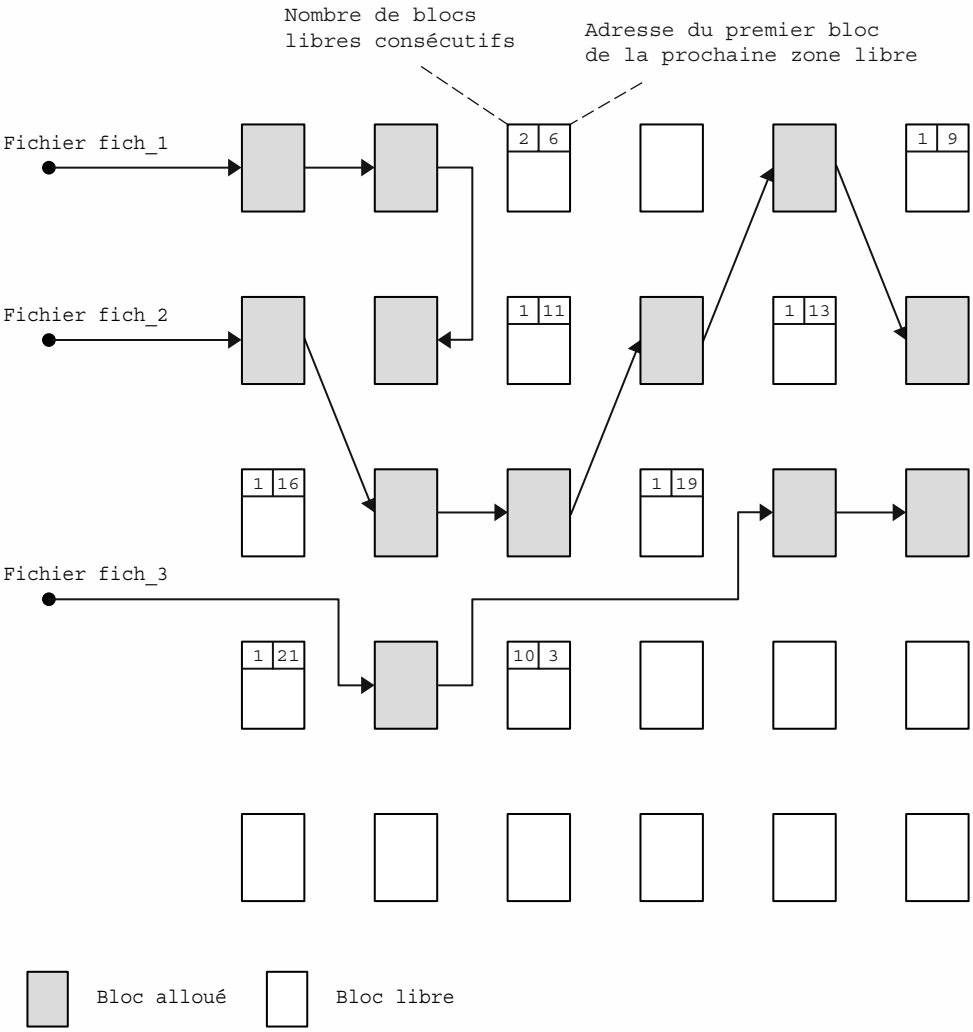

Figure 3.7 – **Gestion de l'espace libre par liste chaînée**

3.1.3 Correspondance fichier logique – fichier physique

a) Notion de répertoire

Le système de gestion de fichiers effectue la correspondance entre fichiers logiques et fichiers physiques par le biais d'une table appelée *répertoire* qui contient des informations de gestion des fichiers, dont notamment, pour chaque fichier existant sur le disque, le nom logique du fichier et son adresse physique sur le disque. Plus précisément, une entrée de répertoire concernant un fichier donné, contient généralement les informations suivantes :

- le nom logique du fichier ;
- le type du fichier si les fichiers sont typés. Les principaux types de fichiers sont les fichiers texte, les fichiers objet, les fichiers exécutables, les fichiers de traitement par lots contenant des commandes pour l'interpréteur de commandes, les fichiers binaires correspondant par exemple à des images ou à des impressions. Le type d'un fichier est souvent codé dans son nom logique, à l'aide d'une extension séparée du nom proprement dit du fichier par un point. Ainsi, les extensions .exe pour les fichiers exécutables, .o pour les fichiers objets, .doc pour les fichiers texte issus de l'application Word, .gif ou .jpeg pour des fichiers binaires de type images ;
- l'adresse physique du fichier, c'est-à-dire une information permettant d'accéder aux blocs physiques alloués au fichier. Cette information dépend de la méthode d'allocation mise en œuvre sur le disque. Ainsi, dans le cas d'une allocation contiguë ou dans le cas d'une allocation par blocs chaînés, l'information mémorise l'adresse du premier bloc alloué au fichier. Dans le cas d'une allocation par zones, cette information est une structure contenant l'adresse de la zone primaire et les adresses des zones secondaires allouées au fichier. Enfin, dans le cas d'une allocation indexée, cette information est constituée par l'adresse du bloc d'index ;
- la taille en octets ou en blocs du fichier ;
- la date de création du fichier ;
- le nom du propriétaire du fichier ;
- les protections appliquées au fichier, à savoir si le fichier est accessible en lecture, écriture ou en exécution et éventuellement par quels utilisateurs.

Le système de gestion de fichiers offre des primitives permettant de manipuler les répertoires ; ce sont les opérations permettant de lister le contenu d'un répertoire, de créer un répertoire ou de détruire un répertoire.

Ainsi sous le système Unix, la commande `mkdir nom_rep` crée un répertoire. La commande `rmdir nom_rep` détruit le répertoire. Enfin, la commande `ls -l nom-rep` permet d'afficher l'ensemble des fichiers connus du répertoire avec leurs caractéristiques.

Les différentes structures de répertoires existantes se distinguent par le nombre de niveaux qu'elles présentent. Les répertoires à un niveau groupent tous les fichiers d'un support de masse dans une même table. S'ils sont simples, les répertoires à un niveau posent des difficultés quand le nombre de fichiers augmente et lorsque

plusieurs utilisateurs différents stockent leurs fichiers sur un même support de masse car tous les noms de fichiers doivent être différents. On préfère alors les répertoires à deux niveaux où chaque utilisateur possède un répertoire propre, appelé *répertoire de travail*. La structure à deux niveaux se généralise facilement dans une structure à *n* niveaux pour laquelle chaque utilisateur hiérarchise son propre répertoire en autant de sous-répertoires qu'il le désire. Cette structure en arbre est composée d'un répertoire initial appelé la racine, souvent symbolisée par « / », d'un ensemble de nœuds constitué par l'ensemble des sous-répertoires et d'un ensemble de feuilles qui sont les fichiers eux-mêmes.

Dans cette structure à *n* niveaux, le nom complet d'un fichier encore appelé *pathname* est constitué de son nom précédé du chemin dans la structure de répertoires depuis la racine. Ce chemin est composé des noms de chacun des répertoires à traverser depuis la racine, séparés par un caractère de séparation (par exemple / sous Linux) Ainsi, sur la figure 3.8, le nom complet du fichier F1 de l'utilisateur Arthur est /Arthur/pg/F1.

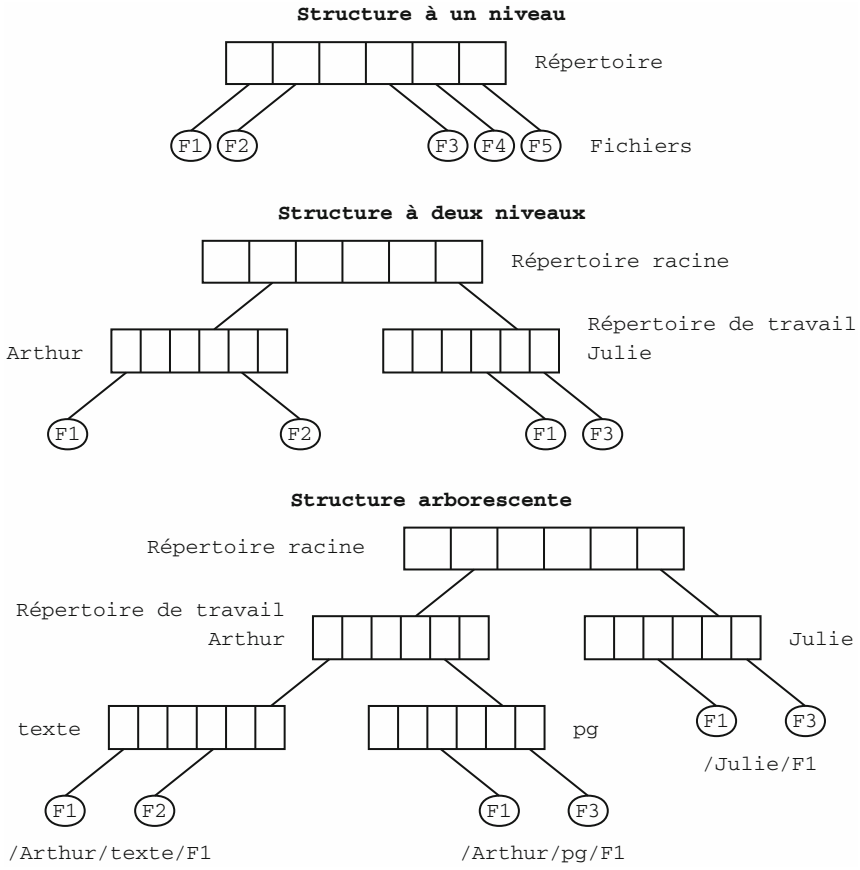

Figure 3.8 – Structures de répertoire

b) Notion de volumes ou partitions

Le système de gestion de fichiers d'un ordinateur peut comporter des milliers de fichiers répartis sur plusieurs giga-octets de disque. Gérer ces milliers de fichiers dans un seul ensemble peut se révéler difficile. Une solution couramment mise en œuvre est alors de diviser l'ensemble du système de gestion de fichiers en morceaux indépendants appelés *volumes* ou *partitions*. Chaque partition constitue alors un disque virtuel, auquel est associé un répertoire qui référence l'ensemble des fichiers présents sur la partition. Le disque virtuel ainsi créé peut physiquement correspondre à une seule partie de disque physique, ou bien regrouper plusieurs parties de disque, placées sur un même disque ou des disques différents (figure 3.9).

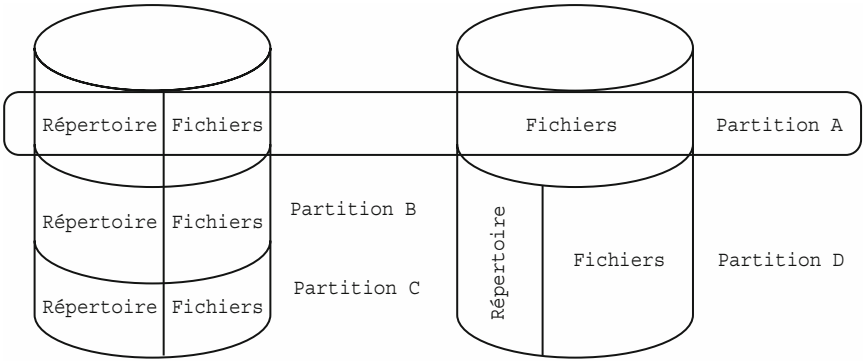

Figure 3.9 – Partitions

Chaque partition est repérée par un nom appelé *label*. Pour pouvoir être accessible, la partition doit être connectée à l'arborescence de fichiers de la machine, en un point d'ancrage qui correspond à un répertoire. Rendre accessible le contenu d'une partition à l'utilisateur de l'ordinateur en liant une partition à un répertoire constitue l'*opération de montage de la partition*. Cette opération est souvent accomplie à l'initialisation du système.

3.2 Le système de gestion de fichiers de Linux

Les premières versions du système Linux étaient basées sur le système d'exploitation Minix et supportaient comme seul système de gestion de fichiers, le système de gestion de fichiers de Minix. Ce système de gestion de fichiers comportait trois défauts importants :

- les adresses des blocs étant sur 16 bits, la taille maximale du système de gestion de fichiers est limitée à 64 Ko ;
- les répertoires comportent seulement 16 entrées ;
- un nom de fichier ne peut pas excéder 14 caractères.

Deux développements ont vu le jour pour répondre à ces problèmes :

- un nouveau système de gestion de fichiers propre à Linux nommé dans sa première version Extfs (*Extended File System*), puis dans sa seconde version actuelle Ext2fs (*The second Extended File System*). Il fait l'objet de ce paragraphe ;
- une couche logicielle appelée *Virtual File System* (VFS) permettant à Linux de supporter d'autres systèmes de gestion de fichiers que le sien. Nous l'étudions au paragraphe 3.3.

3.2.1 Structure d'un fichier dans Ext2

Un fichier physique Linux est identifié par un nom et toutes les informations le concernant sont stockées dans un descripteur appelé *inode* ou i-nœud. Par ailleurs, le fichier n'a pas de structure logique et est simplement constitué comme une suite d'octets. Les blocs sont alloués au fichier selon une organisation indexée. Chaque bloc est identifié par un numéro logique qui correspond à son index dans la partition du disque et est codé sur 4 octets. La taille d'un bloc devant être une puissance de 2 multiple de la taille d'un secteur (généralement 512 octets), elle varie selon les systèmes entre 512, 1 024, 2 048 ou 4 096 octets.

a) L'inode

L'inode du fichier est une structure stockée sur le disque, allouée à la création du fichier, et repérée par un numéro. Une inode contient les informations suivantes :

- le nom du fichier ;
- le type du fichier et les droits d'accès associés ;
- les identificateurs du propriétaire du fichier ainsi que du groupe ;
- diverses heures telles que l'heure de dernier accès au fichier, l'heure de dernière modification du contenu du fichier, l'heure de dernière modification de l'inode, l'heure de suppression du fichier ;
- le nombre de liens vers d'autres fichiers ;
- la taille du fichier en octets ;
- le nombre de blocs de données alloués au fichier ;
- deux listes de contrôle d'accès respectivement pour le fichier et pour le répertoire ;
- une table des adresses des blocs de données.

Type de fichiers Linux

Le système de gestion de fichiers Ext2 reconnaît 5 types de fichiers différents :

- les *fichiers réguliers ou normaux* (*regular files*) sont les fichiers les plus courants. Ils sont destinés à recevoir les données des utilisateurs quel que soit le type de ces données. Ces fichiers sont sans organisation et ne constituent qu'une suite d'octets caractérisée par sa longueur ;
- les *répertoires* ;
- les *liens symboliques* constituent des pointeurs vers un autre fichier et toute action effectuée sur un lien équivaut à une action sur le fichier pointé ;

- les *fichiers de périphériques* sont des fichiers liés à un pilote d'entrées-sorties au sein du noyau. Une opération de lecture ou d'écriture réalisée sur le fichier équivaut à une action sur le périphérique. On distingue à ce niveau les fichiers spéciaux en mode bloc tels que les disques pour lesquels les échanges s'effectuent bloc à bloc et les fichiers spéciaux en mode caractère tels que les terminaux pour lesquels les échanges s'effectuent caractère à caractère. Ces fichiers spéciaux ainsi que le mécanisme des entrées-sorties font l'objet du chapitre suivant ;
- les *tubes nommés* et les *sockets* sont des outils de communication entre processus que nous étudions respectivement au chapitre 6 et au chapitre 8.

Les liens symboliques, les tubes nommés, les sockets et les fichiers de périphériques sont des fichiers dont la structure physique se résume à l'allocation d'une inode. Aucun bloc de données ne leur est associé.

Droits d'accès aux fichiers Linux

Les droits d'accès à un fichier définissent pour un ensemble de trois entités, les actions pouvant être réalisées par les membres de ces entités. Ces entités sont :

- le propriétaire du fichier ;
- le groupe auquel le propriétaire appartient ;
- les autres qui englobent tous les utilisateurs n'appartenant pas au groupe.

Trois types de permissions sont définis sur un fichier. Elles peuvent être notées de deux façons, soit par une lettre, soit par un chiffre en octal :

- la permission en lecture (r ou valeur 3_8) ;
- la permission en écriture (w ou valeur 2_8) ;
- la permission en exécution (x ou valeur 1_8).

Par ailleurs, 3 autres bits ont une signification particulière :

- le bit *setuid*. Un exécutable pour lequel ce bit est positionné s'exécute avec l'identité du propriétaire du fichier et non avec celle de l'utilisateur qui a demandé l'exécution ;
- le bit *setgid*. Un exécutable pour lequel ce bit est positionné s'exécute avec l'identité du groupe du propriétaire du fichier et non avec celle de l'utilisateur qui a demandé l'exécution ;
- le bit *sticky*. Les fichiers appartenant à un répertoire pour lequel ce bit est positionné ne peuvent être supprimés que par leurs propriétaires.

Visualisation des informations liées aux fichiers

La commande `ls -la` permet de visualiser à l'écran les fichiers appartenant à l'utilisateur avec un ensemble d'informations groupées dans des champs :

- le premier champ code le type du fichier et les droits associés au fichier. Le type est codé par une lettre qui prend la valeur suivante :
  - `d` pour un répertoire (*directory*) ;
  - `p` pour un tube (*pipe*) ;

  - `l` pour un lien symbolique ;
  - `b` pour un périphérique bloc ;
  - `c` pour un périphérique caractère ;
  - `-` pour un fichier ordinaire.
- les droits d'accès sont codés selon un ensemble de 3 triplets correspondant à chacune des permissions possibles pour chacune des trois entités citées au paragraphe précédent. Ainsi `rwxr--r-x` indique que l'utilisateur peut lire, écrire et exécuter le fichier (premier groupe `rwx`), le groupe peut seulement lire le fichier (deuxième groupe `r--`) tandis que les autres peuvent lire et exécuter le fichier (troisième groupe `r-x`).
- le deuxième champ indique le nombre de liens pour le fichier ;
- le troisième champ indique le nom de l'utilisateur propriétaire du fichier ;
- le quatrième champ indique le nom du groupe ;
- le cinquième champ indique le nombre d'octets composant le fichier ;
- le sixième champ indique la date de dernière modification du fichier ;
- le dernier champ indique le nom du fichier.

La commande `chmod arg1 arg2` permet au propriétaire d'un fichier de modifier les droits d'accès à ce fichier :

- le premier argument `arg1` a la forme « pourqui action droit » et chacun des champs peut prendre l'une des valeurs suivantes :
  - `u`, `g`, `o` ou `a` pour le champ « pourqui » selon si l'on souhaite désigner le propriétaire (*user*), le groupe (*group*), les autres (*others*) ou tout le monde (*all*) ;
  - `+`, `-` pour le champ « action » selon si l'on souhaite ajouter un nouveau droit ou le retirer ;
  - `r`, `w`, `x` pour le champ « droit » selon le type de permission concerné.
- le deuxième argument `arg2` donne le nom du fichier concerné.

```
lmi20: # ls -la
-rw-r--r--   1 delacroi  ensinf    523 Mar 25 19:28 exemple3.c
-rw-r--r--   1 delacroi  ensinf    591 Mar 25 19:24 exemple3.txt
-rw-r--r--   1 delacroi  ensinf    590 Mar 25 19:24 exemple3.txt~
drwxr-xr-x   2 delacroi  ensinf    4096 Apr  5 19:27 exercices
-rw-r--r--   1 delacroi  ensinf    129816 Dec 27  2001 minirtl.pdf
-rw-r--r--   2 delacroi  ensinf    57 Mar 25 19:29 trace
-rw-r--r--   1 delacroi  ensinf    971 Mar 21 09:56 trace3
lmi20: # chmod a+x exemple3.c
lmi20: # ls -la
-rw-r-xr-x   1 delacroi  ensinf    523 Mar 25 19:28 exemple3.c
-rw-r--r--   1 delacroi  ensinf    591 Mar 25 19:24 exemple3.txt
```

Liens physiques et liens symboliques

Le système de gestion de fichiers Ext2 manipule deux types de liens, les *liens physiques* et les *liens symboliques*.

Le lien physique correspond à l'association d'un nom à un fichier et donc à une inode. Un même fichier peut recevoir plusieurs noms différents, dans un même répertoire ou dans des répertoires différents, placés sur une même partition. Créer un lien physique sur un fichier équivaut à créer une nouvelle entrée dans un répertoire pour mémoriser l'association « nom_de_fichier, inode ». Un lien physique ne peut pas être créé sur un répertoire.

Le lien symbolique correspond à la création d'un fichier de type particulier (l), qui contient comme donnée un pointeur vers un autre fichier.

La commande `ln nomfichier1 nomfichier2` crée un lien physique sur le fichier `nomfichier1`, en lui attribuant un nouveau nom `nomfichier2`.

La commande `ln -s nomfichier1 nomfichier2` crée un lien symbolique sur le fichier `nomfichier1`, en créant le fichier `nomfichier2` qui contient comme donnée le chemin d'accès au fichier `nomfichier1`.

```
lmi20: # ls -la
-rw-r--r--   1 delacroi   delacroi  836 Mar 25 17:53 exemple1.c
lmi20: # ln exemple1.c exemplebis
lmi20: # ls -la
-rw-r--r--   2 delacroi   delacroi  836 Mar 25 17:53 exemple1.c
-rw-r--r--   2 delacroi   delacroi  836 Mar 25 17:53 exemplebis
lmi20: # ln -s exemple1.c versexemple1.c
lmi20: # ls -la
-rw-r--r--   2 delacroi   delacroi  836 Mar 25 17:53 exemple1.c
-rw-r--r--   2 delacroi   delacroi  836 Mar 25 17:53 exemplebis
lrwxrwxrwx   1 delacroi   delacroi  10 Apr  7 09:50 versexemple1.c ->
exemple1.c
```

b) Allocation des blocs de données

L'inode du fichier contient un tableau de `EXT2_N_BLOCKS` entrées, contenant chacune un numéro de blocs logiques. Par défaut la valeur de `EXT2_N_BLOCKS` est égale à 15.

L'organisation de cette table suit l'organisation indexée que nous avons présentée au paragraphe « allocation indexée » dans la section 3.1.2. Plus précisément (figure 3.10) :

- les 12 premières entrées du tableau contiennent un numéro de bloc logique correspondant à un bloc de données du fichier ;
- la treizième entrée sert de premier niveau d'index. Elle contient un numéro de bloc logique contenant lui-même des numéros de blocs logiques qui correspondent à des blocs de données ;
- la quatorzième entrée sert de deuxième niveau d'index. Elle contient un numéro de bloc logique contenant lui-même des numéros de blocs logiques qui contiennent à leur tour les numéros de blocs logiques correspondant à des blocs de données ;

- la quinzième entrée introduit un niveau d'indirection supplémentaire.

Ainsi si T est la taille d'un bloc, alors la taille maximale d'un fichier en nombre de blocs est égale à $12 + (T/4) + (T/4)^2 + (T/4)^3$ blocs.

Lors de la création d'un fichier, aucun bloc de données ne lui est alloué. Les blocs sont attribués au fichier au fur et à mesure de son extension, en commençant par allouer les 12 premières entrées de la table, puis en emplissant ensuite le premier niveau d'indirection, puis le second et enfin le troisième si besoin est.

Cependant pour des raisons d'efficacité et de façon à réduire la fragmentation du fichier, le système préalloue des blocs de données. Plus précisément, lorsque le système alloue un nouveau bloc pour le fichier, il préalloue en même temps jusqu'à 8 blocs adjacents. Ces blocs préalloués sont libérés lorsque le fichier est fermé, s'ils n'ont pas été utilisés.

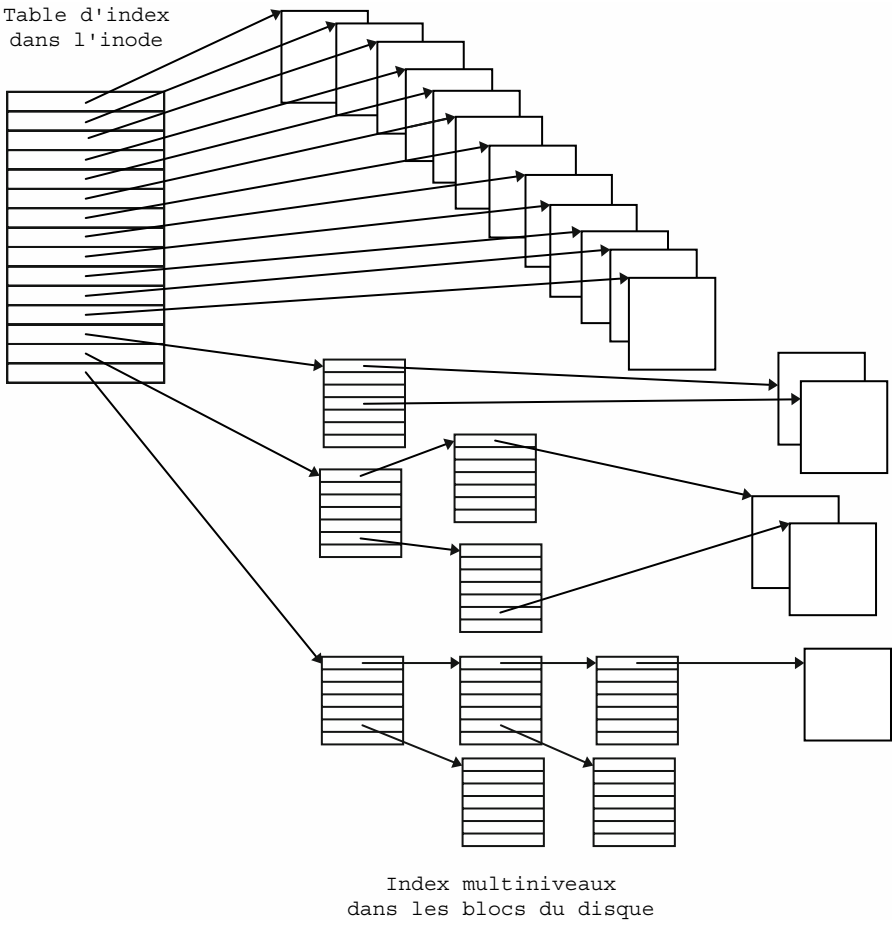

Figure 3.10 – Adressage des blocs constituant un fichier

3.2.2 Structure d'un répertoire

Le système de gestion de fichiers Ext2 est organisé selon une forme arborescente, avec un répertoire de partition organisé sur *n* niveaux. La racine de l'arborescence est représentée par le répertoire racine symbolisé par le caractère « / » et chacun des nœuds de l'arbre est lui-même un répertoire (figure 3.11). Chaque répertoire contient deux entrées particulières, « . » pour désigner le répertoire lui-même et « .. » pour désigner le répertoire parent.

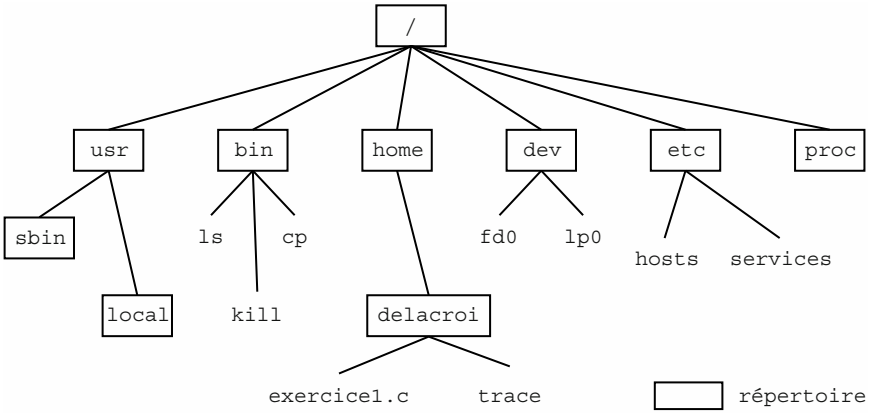

Figure 3.11 – Arborescence des fichiers Ext2

Un répertoire est implémenté comme un type spécial de fichier, pour lequel les blocs de données contiennent les noms de fichiers appartenant au répertoire ainsi que les numéros d'inodes correspondant. Plus précisément une entrée de répertoire contient les informations suivantes pour chaque fichier :

- le numéro d'inode ;
- la longueur de l'entrée du répertoire ;
- la longueur du nom de fichier (maximum 255 caractères) ;
- le type du fichier ;
- le nom du fichier.

La figure 3.12 donne un exemple de répertoire Ext2. Sur cet exemple, certaines entrées de nom de fichier sont complétées par des caractères nuls « \0 » car le système impose que la longueur d'une entrée de répertoire soit toujours un multiple de 4.

Un répertoire étant considéré comme un fichier, il reçoit les mêmes attributs en terme de droits d'accès. Cependant, leur interprétation est un peu différente. Ainsi :

- la permission « r » autorise le listage du contenu du répertoire ;
- la permission « w » autorise la création d'un répertoire et également la suppression du contenu du répertoire ;
- la permission « x » autorise la traversée du répertoire.

Inode	Longueur de l'entrée	Longueur du nom de fichier	Type de fichier	Nom
23	12	1	2 (répertoire)	.\0\0\0
24	12	2	2 (répertoire)	..\0\0\0
68	20	10	1 (fichier régulier)	exercice.C\0\0
36	16	5	2 (répertoire)	livre\0\0\0
12	16	4	5 (tube nommé)	tube\0\0\0\0

Figure 3.12 – **Répertoire Ext2**

3.2.3 Structure d'une partition

Une partition Ext2 est composée tout d'abord d'un bloc d'amorçage utilisé lors du démarrage du système puis d'un ensemble de groupe de blocs, chaque groupe contenant des blocs de données et des inodes enregistrés dans des pistes adjacentes (figure 3.13). Chaque groupe de blocs contient à son tour les informations suivantes :

- une copie du *super-bloc* du système de gestion de fichier ;
- une copie des *descripteurs de groupe de blocs* ;
- un vecteur de bits pour gérer les blocs libres du groupe ;
- un groupe d'inodes souvent appelé table des inodes. Le numéro d'une inode est égal à son rang dans la table ;
- un vecteur de bits pour gérer les inodes libres ;
- des blocs de données.

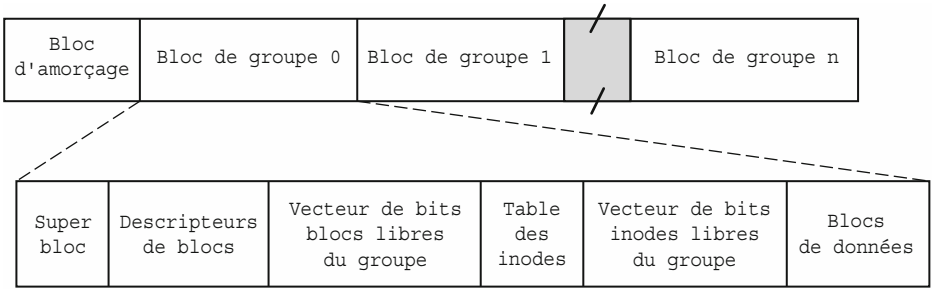

Figure 3.13 – **Partition Ext2**

a) Super-bloc du système de gestion de fichiers

Le super-bloc contient des informations générales sur la partition telles que :

- le nom de la partition ;
- l'heure de la dernière opération de montage, le nombre d'opérations de montage réalisées sur la partition ;
- la taille de la partition en bloc ;
- le nombre total d'inodes dans la partition ;
- la taille d'un bloc dans la partition ;
- le nombre de blocs libres et d'inodes libres dans la partition ;
- le nombre de blocs et d'inodes par groupe ;
- l'heure du dernier contrôle de cohérence et l'intervalle de temps entre chaque contrôle de cohérence.

Ce super-bloc est dupliqué dans tous les groupes de blocs. Le système utilise couramment la copie placée dans le groupe de blocs 0. La mise à jour des copies placées dans les autres groupes de blocs a lieu lors des contrôles de cohérence du système de gestion de fichiers (commande `/sbin/e2fsck`). Si une corruption est constatée sur le super-bloc du groupe 0, alors les copies redondantes sont utilisées pour ramener la partition à un état cohérent.

b) Descripteur de groupe de blocs

Chaque groupe de blocs est associé à un descripteur qui contient les informations suivantes :

- les numéros des blocs contenant les vecteurs de bits gérant la liste des blocs libres et la liste des inodes libres ;
- le nombre de blocs libres et le nombre d'inodes libres dans le groupe ;
- le nombre de répertoires dans le groupe ;
- le numéro du premier bloc contenant les inodes libres.

D'une façon similaire au super-bloc, les descripteurs des groupes de blocs sont dupliqués dans tous les groupes de blocs. Les mises à jour sont effectuées lors des contrôles de cohérence de la partition.

Afin de réduire la fragmentation, le système Linux tente d'allouer un nouveau bloc à un fichier dans le groupe de blocs contenant le dernier bloc alloué pour ce même fichier. Sinon, il cherche un bloc libre dans le groupe de blocs contenant l'inode du fichier.

c) Vecteurs de bits

La liste des blocs libres et la liste des inodes libres sont gérées par l'intermédiaire de deux vecteurs de bits. Un bit égal à 0 signifie que le bloc de donnée ou l'inode correspondant est libre et la valeur 1 signifie au contraire que le bloc de donnée ou l'inode correspondant est occupé. Chacun des vecteurs de bits est entièrement compris dans un seul bloc du disque.

3.3 VFS : LE SYSTÈME DE GESTION DE FICHIERS VIRTUEL

3.3.1 Présentation

Le système Linux supporte d'autres systèmes de gestion de fichiers en plus du système natif Ext2, liés à d'autres systèmes d'exploitation, tels que Minix, MS/DOS, NTFS, les systèmes de gestion de fichiers des variantes d'Unix (System V et BSD) ou encore des systèmes de fichiers propriétaires comme HPFS (OS/2 d'IBM), HFS (Macintosh Apple)…

Afin que l'accès à ces différents systèmes de gestion de fichiers soit transparent et uniforme pour l'utilisateur, une couche logicielle appelée VFS (*Virtual File System*) est insérée dans le noyau Linux. Cette couche logicielle offre à l'utilisateur un modèle de fichier commun à tous ces systèmes de gestion de fichiers, ainsi qu'un ensemble de primitives permettant de manipuler ce modèle commun.

Ces primitives internes au VFS exécutent les parties communes aux différents appels systèmes des différents systèmes de gestion de fichiers, puis elles appellent la fonction spécifique liée au système de gestion de fichiers concerné. Grossièrement, une telle fonction a donc la structure suivante :

```
Vérification des paramètres;
Conversion du nom de fichier en numéro de périphérique et numéro d'inode;
Vérification des permissions;
Appel à la fonction spécifique au système de gestion de fichiers concerné.
```

En plus de cette première fonctionnalité, le VFS gère deux systèmes de cache :

- un cache de noms qui conserve les conversions les plus récentes des noms de fichiers en numéro de périphérique et numéro d'inode ;
- un cache de tampons disque appelé *buffer cache*, qui contient un ensemble de blocs de données lus depuis le disque.

La figure 3.14 présente la place du VFS dans le noyau Linux.

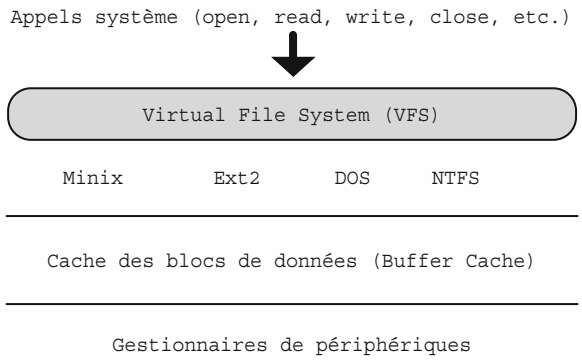

Figure 3.14 – *Virtual File System*

3.3.2 Structure et fonctionnement du VFS

a) Les structures du VFS

Les structures du VFS créent un modèle de fichier pour tout système de gestion de fichiers, qui est très proche du modèle de fichier natif de Linux, Ext2. Ce choix permet à Linux de travailler sur son propre système de gestion de fichiers avec un minimum de surcharge.

La structure du VFS est organisée autour d'objets auxquels sont associés des traitements. Les objets sont (figure 3.15) :

- l'objet système de gestion de fichiers (*super-bloc*) ;
- l'objet inode ;
- l'objet fichier (*file*) ;
- l'objet nom de fichier (*dentry*).

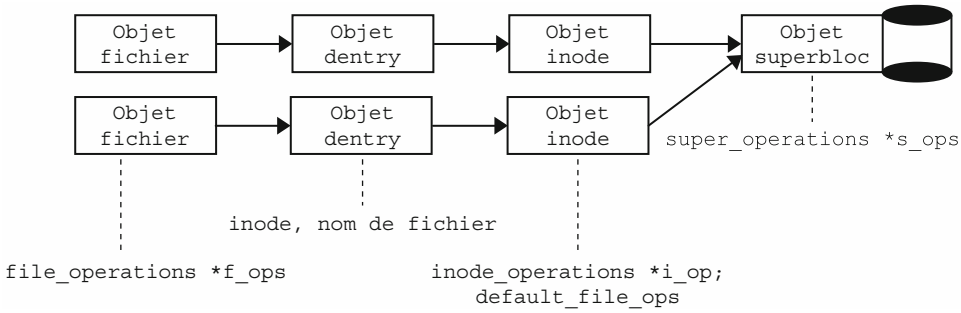

Figure 3.15 – Objets du *Virtual File System*

Objet système de gestion de fichiers

Chaque système de gestion de fichiers accessible par le noyau est représenté par une structure super-bloc (`struct super_block` définie dans `<linux/fs.h>`) qui contient des informations sur les caractéristiques du système de gestion de fichiers telles que la taille des blocs, les droits d'accès, la date de dernière modification, le point de montage sur l'arborescence de fichiers. Cette structure offre un ensemble d'opérations permettant de manipuler le système de gestion de fichiers associé (champ `struct super_operations *s_ops`) notamment lire ou écrire une inode, écrire le superbloc et obtenir des informations sur le système de gestion de fichiers. L'ensemble des structures `super_block` est regroupé dans une liste doublement chaînée.

Enregistrement d'un système de gestion de fichiers et montage d'un système

Pour qu'un système de gestion de fichiers puisse être accessible au noyau Linux, deux opérations doivent être réalisées :

- le système de gestion de fichiers doit être enregistré auprès du noyau. Cet enregistrement s'effectue soit à l'initialisation du système si le support lié au système

de gestion de fichiers a été compilé, soit au moment du chargement d'un module utilisant le système de gestion de fichiers. La fonction `register_filesystem()` effectue cet enregistrement et retourne une structure descriptive du type de système de gestion de fichiers enregistré (structure `file_system_type` définie dans `<linux/fs.h>`) ;

- le système de gestion de fichiers doit être monté sur l'arborescence de fichiers courante en appelant la commande `mount`. Le noyau parcourt la liste des structures `file_system_type` pour trouver la structure descriptive du système de gestion de fichiers à monter, puis exécute la méthode `read_super()` de cette structure qui retourne un super-bloc pour ce système de gestion de fichiers.

Objet inode

Chaque fichier dans le VFS est décrit par une inode (`structure struct inode` définie dans `<linux/fs.h>`). Cette structure contient des informations sur le fichier associé à l'inode de même nature que celles trouvées dans une inode de type Ext2. Elle contient par ailleurs des informations spécifiques liées au type de système de gestion de fichiers ainsi que des méthodes permettant de manipuler la structure (`struct inode_operations *i_op`). Notamment, lors du chargement d'une inode, le champ `default_file_ops` de la structure `i_op` est initialisée avec les opérations sur fichier spécifiques au fichier décrit par l'inode.

L'ensemble des inodes est géré de deux façons différentes :

- une liste circulaire doublement chaînée regroupe l'ensemble des inodes ;
- pour la recherche rapide d'une inode particulière, l'accès s'effectue par le biais d'une table de hachage, avec pour clé de recherche, l'identifiant du super-bloc et le numéro de l'inode dans ce super-bloc.

Objet fichier

Chaque fichier ouvert est décrit par une structure `file` (définie dans `<linux/fs.h>`). Cette structure contient un ensemble d'informations utiles pour la réalisation des opérations de lecture et d'écriture sur le fichier, telles que le type d'accès autorisé, la position courante dans le fichier, le nombre de processus ayant ouvert le fichier, etc.

Comme les structures précédentes, la structure `file` inclut un ensemble de méthodes (`struct file_operations *f_op`) permettant de manipuler la structure, qui sont les opérations pour la lecture, l'écriture, l'ouverture ou la fermeture d'un fichier, etc. Ces opérations sont initialisées à l'ouverture du fichier depuis le champ `default_file_ops` de la structure `i_op` de l'inode correspondant au fichier.

Objet nom de fichier

Les objets nom de fichier ou *dentry* sont créés pour mémoriser les entrées de répertoire lues lors de la recherche d'un fichier. Ainsi la recherche du fichier `/home/delacroi/exemple.c` aboutit à la création de trois objets dentry, un pour `home`, le suivant pour `delacroi`, le dernier pour `exemple.c`, chaque objet dentry effectuant le lien entre le nom et le numéro d'inode associé. Ces objets sont par ailleurs gérés par le cache des noms que nous étudions au paragraphe 3.3.2.

Liaison du processus avec les objets du VFS

Les fichiers couramment ouverts par un processus sont mémorisés dans une table, appelée table des fichiers ouverts, pointée depuis le bloc de contrôle du processus (champ `files`). Chacune des entrées de cette table (table `fd`), nommée *descripteur de fichier*, pointe vers un objet fichier (figure 3.16). Chaque objet fichier pointe à son tour vers un objet dentry, qui effectue la liaison entre le nom du fichier et son inode. Chaque objet dentry pointe donc vers l'objet inode du fichier, cet objet inode pointant à son tour vers l'objet super-bloc décrivant le système de gestion de fichiers auquel appartient le fichier.

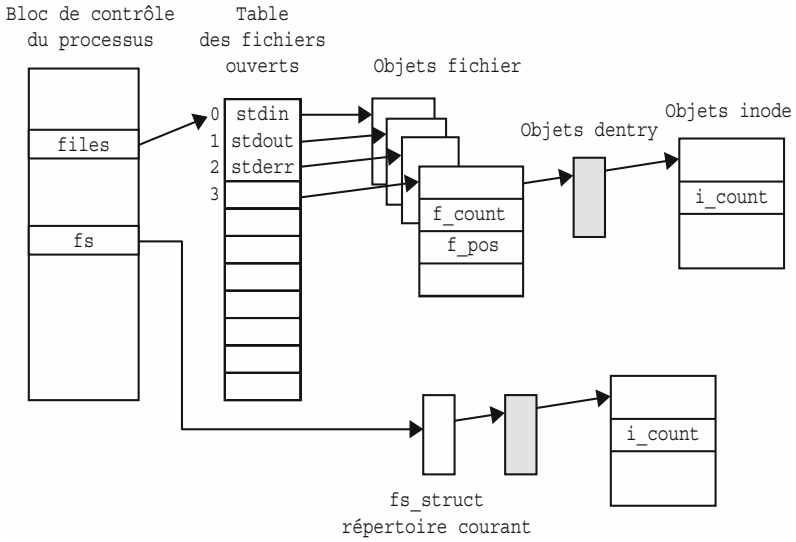

Figure 3.16 – Table des fichiers ouverts et relations entre les objets du VFS

Par ailleurs, comme le montre également la figure 3.16, le champ `fs` du bloc de contrôle du processus pointe sur un objet fichier, qui correspond au *répertoire courant* du processus, c'est-à-dire le répertoire à partir duquel l'exécution du processus a été lancée.

Nous avons vu au chapitre 2 qu'un processus fils hérite des fichiers ouverts par son père. Lors de la création du nouveau processus, le noyau duplique la table des fichiers ouverts du père pour en attribuer la copie au fils. Le père et le fils pointent donc sur les mêmes objets fichiers. Un champ `f_count` placé dans chaque objet fichier permet au VFS de connaître le nombre de références au même objet fichier. Il est incrémenté à chaque fois qu'une nouvelle référence est faite sur le fichier. L'objet fichier est libéré par le noyau lorsque le compteur d'utilisation de l'objet devient égal à 0. Chaque objet fichier maintient également dans le champ `f_pos`, le déplacement courant dans le fichier encore appelé *pointeur du fichier*. Ainsi, sur la figure 3.17, le processus fils a hérité des fichiers ouverts par son père, dont notamment

le fichier pointé par le descripteur numéro 3. Le champ `f_count` de l'objet fichier a été incrémenté d'une unité et est devenu égal à 2. Le père ayant lu 5 éléments dans le fichier avant de créer son fils, le pointeur de fichier a une valeur égale à 5. Le processus père et le processus fils se partage ce pointeur et chacun des deux processus commence ses accès au fichier à partir de cette position courante.

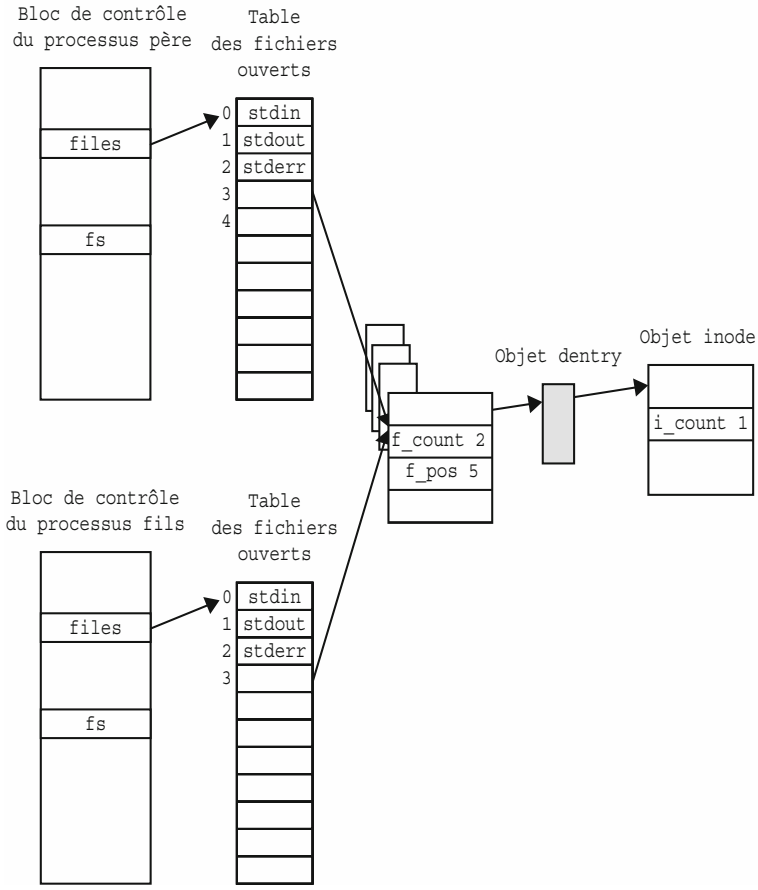

Figure 3.17 – Tables des fichiers ouverts et héritage entre processus

Enfin, chaque inode maintient à son tour dans le champ `i_count`, un compteur d'utilisation qui référence le nombre d'objets fichiers qui lui sont liés. Ainsi, sur la figure 3.18, les deux processus père et fils de la figure précédente ont chacun ouvert le même fichier, accessible pour le processus père depuis son descripteur de fichier numéro 4 et accessible pour le processus fils depuis son descripteur de fichier numéro 5. Deux objets fichiers distincts ont été alloués qui pointent sur le même objet inode. Le compteur de cet objet inode a donc la valeur 2. Le fichier est physiquement fermé par le noyau lorsque le champ `i_count` de l'inode devient nul.

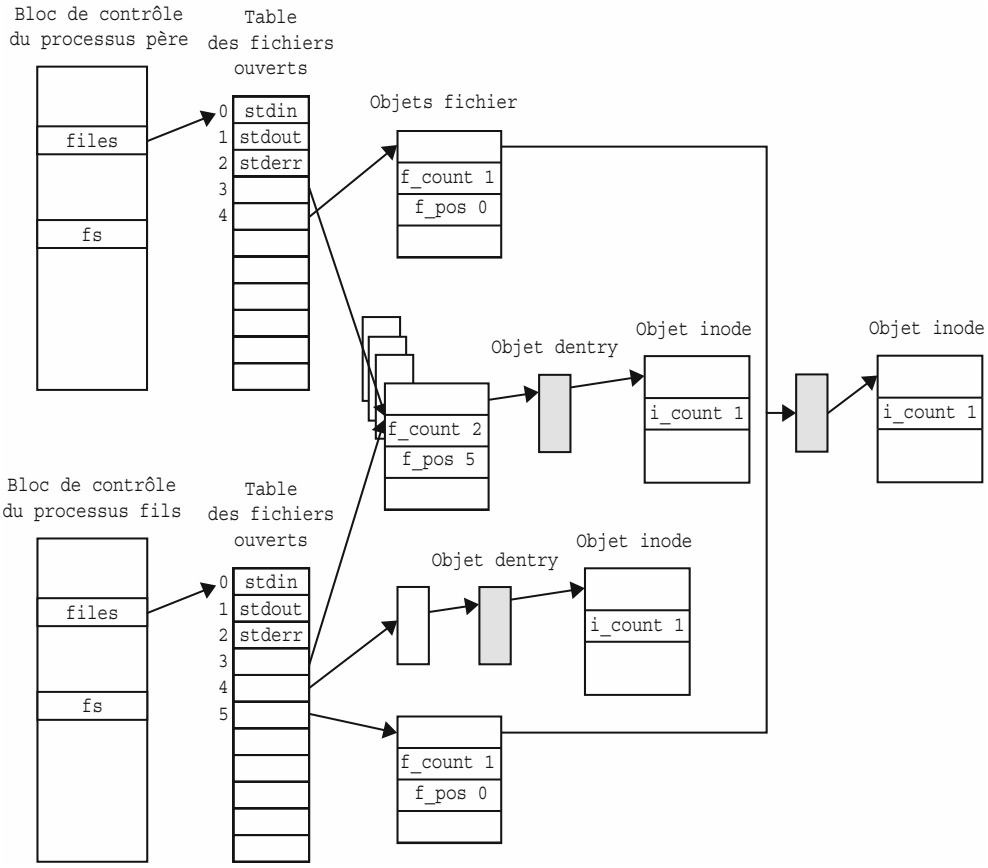

Figure 3.18 – Ouverture d'un même fichier par deux processus

b) Principe d'une opération générique : l'ouverture d'un fichier

Examinons à présent les actions effectuées par le VFS lors de l'ouverture d'un fichier (primitive `desc = open(nom_fichier, mode_ouverture)`) :

- la routine d'enveloppe correspondant à la fonction `open()` lève une trappe, bascule le processus appelant en mode superviseur, puis appelle la routine système `sys_open()` après avoir passé les paramètres d'exécution (nom du fichier et mode d'ouverture) ;
- une nouvelle entrée est allouée dans le tableau `fd` et un nouvel objet fichier pointé par cette entrée est créé ;
- le nom de fichier est converti en numéro de périphérique et inode et l'objet dentry correspondant est associé au fichier ;
- l'inode associée au fichier est obtenue (méthode `look_up` de l'objet inode représentant le répertoire contenant le fichier) ;

- les opérations concernant le fichier (champ `*f_ops` de l'objet fichier) sont initialisées depuis le champ `default_file_ops` de la structure `i_op` de l'inode correspondant au fichier ;
- la méthode `open` de l'objet fichier est appelée pour réaliser l'ouverture physique du fichier en fonction du type du système de gestion de fichiers ;
- l'index de l'entrée de la table `fd` pointant sur l'objet fichier est retourné à l'utilisateur. Elle constitue le descripteur `desc` du fichier.

Dans la table `fd`, les entrées 0, 1 et 2 sont réservées et désignent respectivement l'entrée standard (*stdin*) représentée généralement par le clavier, la sortie standard (*stdout*) et la sortie d'erreur (*stderr*) représentées généralement par l'écran.

c) Fonctionnement des caches

Le VFS maintient deux caches, l'un pour conserver les dentry les plus récemment utilisés, l'autre pour conserver les blocs disque les plus récemment utilisés.

Le cache des dentry (dcache)

Lorsque le VFS doit lire un fichier depuis le support de masse, il doit au préalable convertir le nom logique du fichier vers son équivalent physique, à savoir le numéro du périphérique contenant le fichier et le numéro de l'inode descriptive du fichier. Prenons par exemple la recherche du fichier dont le chemin est `delacroi/exercice/exemple1.c` pour le compte de l'utilisateur delacroi. Le traitement de ce chemin d'accès suit les étapes suivantes :

- le chemin ne commençant pas par « / », la recherche s'effectue depuis le répertoire courant du processus, dont l'inode est repérée depuis le bloc de contrôle du processus. Si le chemin du fichier avait commencé par « / », alors le chemin absolu aurait été analysé à partir de l'inode de la racine du système de gestion de fichiers qui est également conservée en mémoire. En utilisant l'inode décrivant le répertoire courant, le VFS recherche dans les blocs de données de ce fichier répertoire l'entrée concernant « `delacroi` ». L'entrée trouvée, le VFS connaît le numéro d'inode associé à « `delacroi` ». Un objet dentry est créé pour ce couple ;
- l'inode décrivant le répertoire « `delacroi` » est lue depuis le disque et l'entrée concernant « `exercice` » est recherchée dans les blocs de données de ce fichier répertoire. L'entrée trouvée, le VFS connaît le numéro d'inode associé à « exercice ». Un objet dentry est créé pour ce couple ;
- l'inode décrivant le répertoire « exercice » est lue depuis le disque et l'entrée concernant « `exemple1.c` » est recherchée dans les blocs de données de ce fichier répertoire. L'entrée trouvée, le VFS connaît le numéro d'inode associé à « `exemple1.c` ». Un objet dentry est créé pour ce couple.

Cet exemple montre que la recherche d'une inode associée à un fichier est un traitement long et coûteux. Aussi, le VFS maintient-il un cache de tous les objets dentry créés au cours de cette recherche. Ainsi, lorsque le VFS cherche le numéro d'inode associé à un nom de répertoire, il regarde tout d'abord dans le cache si un objet dentry concernant ce nom existe. Si oui, il récupère le numéro d'inode associé sans

lecture disque. Sinon, il lit les blocs de données associés au répertoire depuis le disque jusqu'à trouver l'entrée recherchée.

Le cache des dentry comporte un ensemble d'objets dentry utilisés par le noyau, libres ou non utilisés. Les dentry non utilisés sont des objets relâchés par le noyau mais contenant encore des informations valides qui peuvent être consultées. Cependant un dentry non utilisé peut être récupéré pour y placer de nouvelles informations s'il n'existe plus de dentry libres.

Lorsque le VFS recherche un numéro d'inode associé à un nom, il accède à l'ensemble des dentry utilisés ou non utilisés du cache. Cette consultation s'effectue par le biais d'une table de hachage, qui calcule sa clé à partir de l'objet dentry immédiatement supérieur dans le chemin d'accès et du nom recherché. Si l'objet recherché n'est pas présent, alors le VFS doit créer un nouvel objet dentry. Pour cela, le VFS prend un objet libre s'il en existe un et sinon, récupère l'objet non utilisé le moins récemment utilisé.

Pour terminer sur ce sujet, notons que le cache dentry agit également en tant que contrôleur pour un cache des inodes. En effet chacune des inodes référencées par un objet dentry est placée en mémoire centrale et y reste jusqu'à ce que l'objet dentry associé soit récupéré pour une autre association nom de fichier, inode.

Le cache des blocs disque (buffer cache)

Le VFS maintient également un cache contenant les blocs disque les plus récemment accédés. Lors de la lecture d'un bloc, celui-ci est placé dans une zone mémoire appelée *tampon (buffer)*. Ainsi, lorsque le VFS cherche à accéder à un bloc du disque, il consulte d'abord l'ensemble des tampons présents dans le cache. Si le tampon correspondant est présent, alors l'opération de lecture ou d'écriture est réalisée dans le cache. Sinon, le bloc est lu depuis le disque et placé dans un tampon libre s'il en existe un. Sinon, le VFS récupère le tampon le moins récemment utilisé pour y placer le nouveau bloc dont il a besoin.

Nous avons mentionné un peu plus haut dans ce texte, que l'opération de lecture ou d'écriture était réalisée dans le tampon s'il contenait le bloc requis. Aussi, dans le cas d'une écriture, le bloc est modifié en mémoire centrale sans être recopié immédiatement sur le disque. L'écriture du bloc modifié sur le disque n'est effectuée que lorsque le tampon contenant ce bloc est libéré pour y placer un nouveau bloc. Il s'ensuit qu'un arrêt intempestif du système peut entraîner une perte de données pour tous les tampons dont le contenu modifié n'a pas été recopié sur le disque.

Les threads noyau *bdflush* et *kupdate* que nous avons mentionnés au chapitre 2 sont responsables de la gestion des tampons dans le buffer cache et de leur écriture sur le disque si leur contenu a été modifié :

- *bdflush* est réveillé lorsque trop de tampons modifiés existent dans le cache ou lorsque la taille du cache devint insuffisante par rapport au nombre de tampons requis ;
- *kupdate* effectue une sauvegarde régulière des tampons modifiés.

Par ailleurs, une application utilisateur dispose de trois primitives pour forcer la sauvegarde des tampons modifiés la concernant :

- `sync()` sauvegarde tous les tampons modifiés sur le disque ;
- `fsync()` permet à un processus de sauvegarder sur le disque tous les blocs appartenant à un fichier qu'il a ouvert ;
- `fdatasync()` réalise la même opération que `fsync()` sans sauvegarder l'inode du fichier.

3.4 Primitives du VFS

Les primitives offertes pour l'utilisateur par le VFS concernent d'une part la manipulation des fichiers, d'autre part la manipulation des répertoires. Un troisième groupe de primitives permet de manipuler les liens symboliques. Enfin, un quatrième groupe de primitives concerne les partitions.

3.4.1 Opérations sur les fichiers

a) Ouverture d'un fichier

L'ouverture d'un fichier s'effectue par un appel à la primitive `open()` dont le prototype est donné :

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int open(const char *ref, int mode_ouv, mode_t  mode);
```

Le premier paramètre `*ref` spécifie le chemin du fichier à ouvrir dans l'arborescence du système de gestion de fichiers.

Le paramètre `mode_ouv` est une combinaison de constantes permettant de spécifier les options sur le mode d'ouverture. Ces constantes peuvent être combinées entre elles à l'aide de l'opération de OU binaire « | ». Ce sont :

- `O_RDONLY`, `O_WRONLY`, `O_RDWR` pour demander une ouverture respectivement en lecture seule, écriture seule ou lecture et écriture ;
- `O_CREAT` pour demander la création du fichier s'il n'existe pas. Cette constante peut être combinée avec la constante `O_EXCL` auquel cas la demande de création échoue si un fichier de même nom existe déjà ;
- `O_TRUNC` pour demander la suppression du contenu du fichier si celui-ci existe déjà ;
- `O_APPEND` pour demander l'ouverture du fichier en mode ajout, toutes les écritures ayant lieu à la fin du fichier ;
- `O_SYNC` pour demander la gestion du fichier en mode synchrone, ce qui implique que toute écriture réalisée sur le fichier est immédiatement recopiée sur le disque.

Le troisième paramètre de l'appel n'est positionné que si la création du fichier a été demandée, c'est-à-dire si l'option `O_CREAT` a été utilisée. Elle permet de spécifier les droits d'accès associés au fichier. Les valeurs admises sont :

- `S_ISUID`, `S_ISGID`, `S_ISVTX`, activation respective des bits setuid, setgid, sticky ;
- `S_IRUSR`, `S_IWUSR`, `S_IXUSR`, `S_IRWXU`, activation respective des droits en lecture, écriture, exécution et lecture/écriture/exécution pour le propriétaire du fichier ;
- `S_IRGRP`, `S_IWGRP`, `S_IXGRP`, `S_IRWXG`, activation respective des droits en lecture, écriture, exécution et lecture/écriture/exécution pour le groupe du propriétaire du fichier ;
- `S_IROTH`, `S_IWOTH`, `S_IXOTH`, `S_IRWXO`, activation respective des droits en lecture, écriture, exécution et lecture/écriture/exécution pour les autres.

Comme nous l'avons évoqué précédemment, l'ouverture de fichier entraîne la création d'une nouvelle entrée dans la table des fichiers ouverts du processus, le chargement de l'inode du fichier depuis le disque ou depuis le cache des inodes, enfin l'allocation d'un nouvel objet fichier. La position courante dans le fichier est remise à nulle. Le champ `i_count` de l'inode est incrémenté de 1.

L'index de l'entrée dans la table des fichiers ouverts, constituant le descripteur de fichier, est retourné comme résultat de la primitive `open()`. En cas d'échec la primitive retourne la valeur – 1 et la variable `errno` prend l'une des valeurs suivantes :

- `EACESS`, l'accès n'est pas possible ;
- `EEXIST`, le fichier existe et les options `O_CREAT` et `O_EXCL` ont été spécifiées ;
- `EFAULT`, le chemin spécifié n'est pas valide ;
- `EISDIR`, le chemin spécifié est un répertoire ;
- `ENAMETOOLONG`, le chemin spécifié comprend un nom trop long ;
- `ENOENT`, le fichier n'existe pas et l'option `O_CREAT` n'est pas positionnée ;
- `ENOTDIR`, l'un des composants du chemin d'accès n'est pas un répertoire ;
- `ETXTBSY`, le fichier est un exécutable en cours d'exécution ;
- `ELOOP`, un cycle de liens symboliques est détecté ;
- `EMFILE`, le nombre maximal de fichiers ouverts pour le processus est atteint ;
- `ENOMEM`, la mémoire est insuffisante ;
- `ENOSPC`, le système de fichiers est saturé.

b) Fermeture d'un fichier

La fermeture du fichier par un processus s'effectue par un appel à la primitive `close()` dont le prototype est :

```
#include <unistd.h>
int close(int desc);
```

Le paramètre `desc` correspond au descripteur de fichier obtenu lors de l'ouverture du fichier. L'entrée dans la table des fichiers ouverts du processus est libérée et le champ `f_count` de l'objet fichier est décrémenté d'une unité ainsi que le champ `i_count` de l'inode associée si l'objet fichier disparaît.

c) Lecture et écriture dans un fichier

Le système Linux considère les fichiers comme une suite d'octets non typés et sans aucune structure. C'est donc l'application qui par la définition de types structurés applique une structure au fichier.

La lecture dans un fichier s'effectue à l'aide de la primitive `read()` dont le prototype est :

```
#include <unistd.h>
ssize_t read(int desc, void *ptr_buf, size_t nb_octets );
```

Cette opération de lecture lit dans le fichier correspondant au descripteur `desc`, dans le buffer de réception pointé par `ptr_buf`, un nombre de `nb_octets` octets depuis l'octet où est positionné le pointeur du fichier. Le pointeur du fichier est incrémenté du nombre d'octets lus.

La primitive renvoie le nombre de caractères effectivement lus (0 si la fin de fichier est atteinte) et la valeur – 1 en cas d'échec. La variable `errno` prend l'une des valeurs suivantes :

- `EBADF`, le descripteur est invalide ;
- `EFAULT, buf` n'est pas une adresse valide ;
- `EINTR`, l'appel système a été interrompu par la réception d'un signal ;
- `EINVAL`, le fichier pointé par `fd` ne peut pas être lu ;
- `EIO`, une erreur d'entrées-sorties est survenue ;
- `EISDIR`, le chemin spécifié est un répertoire.

L'écriture dans un fichier s'effectue à l'aide de la primitive `write()` dont le prototype est :

```
#include <unistd.h>
ssize_t write(int desc, void *ptr_buf, size_t nb_octets );
```

Cette opération d'écriture place dans le fichier de descripteur `desc`, le contenu du buffer pointé par `ptr_buf`, un nombre d'octets correspondant à `nb_octets`.

Le pointeur du fichier est incrémenté du nombre d'octets écrits.

La primitive renvoie le nombre de caractères effectivement écrits et la valeur – 1 en cas d'échec. La variable `errno` prend l'une des valeurs suivantes :

- `EBADF`, le descripteur est invalide ;
- `EFAULT, buf` n'est pas une adresse valide ;
- `EINTR`, l'appel système a été interrompu par la réception d'un signal ;
- `EINVAL`, le fichier pointé par `fd` ne peut pas être lu ;
- `EIO`, une erreur d'entrées-sorties est survenue ;
- `EISPIPE`, le descripteur référence un tube sans lecteur (*cf.* chapitre 7) ;
- `ENOSPC`, le système de fichiers est saturé ;
- `EISDIR`, le chemin spécifié est un répertoire.

d) Se positionner dans un fichier

Les lectures et les écritures sur un fichier s'effectuent de façon séquentielle. Il est cependant possible de modifier la position courante du pointeur de fichier à l'aide de la primitive `lseek()` dont le prototype est :

```
#include <unistd.h>
off_t lseek (int desc, off_t dep, int option);
```

Le pointeur du fichier désigné par `desc` est modifié d'une valeur égale à `dep` octets selon une base spécifiée dans le champ `option`. Ce champ `option` peut prendre trois valeurs :

- `SEEK_SET`, le positionnement est effectué par rapport au début du fichier ;
- `SEEK_CUR`, le positionnement est effectué par rapport à la position courante ;
- `SEEK_END`, le positionnement est effectué par rapport à la fin du fichier.

La primitive retourne la position courante en octets par rapport au début du fichier à la suite du positionnement, et la valeur – 1 en cas d'échec. La variable `errno` prend l'une des valeurs suivantes :

- `EBADF`, le descripteur est invalide ;
- `EINVAL, option` n'a pas une valeur admise ;
- `EISPIPE`, le descripteur référence un tube sans lecteur (*cf.* chapitre 7).

e) Un exemple

L'exemple suivant crée un fichier nommé `/home/delacroix/fichnotes` avec des droits en lecture et écriture pour le propriétaire du fichier. Dans ce fichier, 4 couples – nom d'élèves, note – sont enregistrés. Puis le pointeur du fichier est positionné au début du fichier et l'ensemble des éléments précédemment écrits est relu.

```
/*****************************************************************/
/*               Exemple de manipulation d'un fichier:           */
/*                 création, positionnement, fermeture           */
/*****************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

main()
{
struct eleve {
              char nom[10];
              int note;
             };
int fd, i, ret;
struct eleve un_eleve;

fd = open ("/home/delacroix/fichnotes",O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
```

```
if (fd == -1)
     perror ("prob open");
i = 0;
while (i<4)
{
  printf ("Donnez le nom de l'élève \n");
  scanf ("%s", un_eleve.nom);
  printf ("Donnez la note de l'élève \n");
  scanf ("%d", &un_eleve.note);
  write (fd, &un_eleve, sizeof(un_eleve));
  i = i + 1;
}
ret = lseek(fd,0,SEEK_SET);
if (ret==-1)
     perror ("prob lseek");
printf ("la nouvelle position est %d\n", ret);
i = 0;
while(i<4)
{
  read (fd, &un_eleve, sizeof(un_eleve));
  printf ("le nom et la note de l'élève sont %s, %d\n", un_eleve.nom,
un_eleve.note);
  i = i + 1;
}
close(fd);
}
```

f) Manipuler les attributs des fichiers

Accès aux attributs des fichiers

Les attributs associés aux fichiers peuvent être récupérés par un appel aux primitives `stat()` et `fstat()`, dont nous donnons les prototypes :

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *ref, struct stat *infos);
int fstat(const int desc, struct stat *infos);
```

Chacune de ces deux fonctions retourne les attributs associés à un fichier, soit désigné par son nom, soit par son descripteur, dans la structure `struct stat infos`.

Le type `struct stat` est défini dans le fichier `<sys/stat.h>`. Nous détaillons cette structure :

```
struct stat
{
  dev_t   st_dev;        /*identification disque logique*/
  ino_t   st_ino;        /*inode du fichier sur disque*/
  mode_t  st_mode;       /*type du fichier et droits accès
                           utilisateur*/
  nlink_t st_nlink;      /*nombre de liens physiques*/
  uid_t   st_uid;        /*propriétaire du fichier*/
```

```
  gid_t   st_gid;        /*groupe propriétaire du fichier*/
  off_t   st_size;       /*taille du fichier en octets*/
  time_t  st_atime;      /*date dernier accès*/
  time_t  st_mtime;      /*date dernière modification*/
  time_t  st_ctime;      /*date dernière modification du nœud*/
  uint_t  st_blksize;    /*taille d'un bloc dans le fichier*/
  int  st_blocks;        /*blocs alloués pour le fichier*/
};
```

Le type de fichier codé dans le champ `mode_t` peut être obtenu en utilisant l'une des macros suivantes :

- `S_ISBLK (infos->st_mode)`, renvoie vrai si le fichier est un fichier spécial en mode bloc ;
- `S_ISCHR (infos->st_mode)`, renvoie vrai si le fichier est un fichier spécial en mode caractère ;
- `S_ISDIR (infos->st_mode)`, renvoie vrai si le fichier est un répertoire ;
- `S_ISFIFO (infos->st_mode)`, renvoie vrai si le fichier est un tube nommé ;
- `S_ISLNK (infos->st_mode)`, renvoie vrai si le fichier est un lien symbolique ;
- `S_ISREG (infos->st_mode)`, renvoie vrai si le fichier est un fichier régulier ;
- `S_ISREG (infos->st_mode)`, renvoie vrai si le fichier est une socket.

Création et destruction de liens

La création d'un nouveau lien physique `ref2` pour le fichier `ref1` s'effectue par un appel à la primitive `link()` :

```
#include <unistd.h>
int link(const char *ref1, const char *ref2);
```

La commande `ln ref1 ref2` exécute la même action depuis l'interpréteur de commande.

La suppression d'un lien physique `ref` s'effectue par un appel à la primitive `unlink()` :

```
#include <unistd.h>
int unlink(const char *ref);
```

S'il s'agit du dernier lien référençant le fichier, celui-ci est supprimé. La commande `rm ref` exécute la même action depuis l'interpréteur de commande.

Modification du nom du fichier

La modification d'un nom de fichier s'effectue par un appel à la primitive `rename()` :

```
#include <unistd.h>
int rename(const char *ref_anc, const char *ref_nv)
```

Le fichier `ref_anc` est renommé `ref_nv`. La commande `mv ref_anc ref_nv` exécute la même action depuis l'interpréteur de commande.

Modification et test des droits d'accès d'un fichier

Les droits d'accès à un fichier sont positionnés lors de la création de ce fichier (primitive `open()`). Ils peuvent être modifiés par la suite en invoquant les primitives `chmod()` et `fchmod()` :

```
#include <sys/stat.h>
int chmod(const char *ref, mode_t mode);
int fchmod(int desc, mode_t mode);
```

La primitive `chmod()` prend comme premier paramètre le nom du fichier `ref` tandis que la primitive `fchmod()` prend comme premier paramètre le descripteur du fichier `desc` déjà ouvert. Le second paramètre `mode` permet dans les deux cas de spécifier les droits à modifier et il prend les mêmes valeurs que le troisième paramètre de la primitive `open()`.

La commande `chmod mode nom_fich`, vue au paragraphe 3.2.1, exécute la même action depuis l'interpréteur de commande.

Par ailleurs, un processus peut tester les droits d'accès qu'il possède vis-à-vis d'un fichier par un appel à la primitive `access()` :

```
#include <unistd.h>
int access(char *ref, int acces);
```

Le premier paramètre `ref` correspond au nom du fichier. Le paramètre `acces` est une constante qui représente les droits à tester et peut prendre les valeurs suivantes :

- `F_OK` : test d'existence du fichier ;
- `R_OK` : test d'accès en lecture au fichier ;
- `W_OK` : test d'accès en écriture au fichier ;
- `X_OK` : test d'accès en exécution au fichier.

La primitive retourne la valeur 0 si l'accès est possible.

Modification du propriétaire d'un fichier

Le propriétaire et le groupe propriétaire d'un fichier sont fixés lors de la création de ce fichier, en fonction de l'identité du processus demandant la création. Ces deux informations peuvent être modifiées par la suite en faisant appel aux primitives `chown()` ou `fchown()` :

```
#include <unistd.h>
int chown(const char *ref, uid_t id_util, gid_t id_grp);
int fchown(int desc, uid_t id_util, gid_t id_grp);
```

La primitive `chown()` prend comme premier paramètre le nom du fichier `ref` tandis que la primitive `fchown()` prend comme premier paramètre le descripteur du fichier `desc` déjà ouvert. Le second paramètre `id_util` permet dans les deux cas de spécifier l'identité du nouveau propriétaire tandis que le troisième paramètre `id_grp` permet de spécifier l'identité du groupe du nouveau propriétaire. Chacun de ces deux derniers paramètres peut être omis et remplacé par la valeur – 1.

Seul un processus s'exécutant avec les droits du super-utilisateur peut modifier l'identité du propriétaire d'un fichier.

La commande `chown nv_prop nom_fich` effectue la modification de l'identité du propriétaire du fichier `nom_fich` depuis l'interpréteur de commande (cette identité devient `nv_prop`).

La commande `chgrp nv_grp nom_fich` effectue la modification du groupe du propriétaire du fichier `nom_fich` depuis l'interpréteur de commande (ce groupe devient `nv_grp`).

3.4.2 Opérations sur les répertoires

a) Répertoire courant et répertoire racine

Chaque processus possède un répertoire courant qui est le répertoire à partir duquel son exécution a été lancée. La résolution des noms de fichiers relatifs, c'est-à-dire ne commençant pas par « / », s'effectue pour ce processus à partir de ce répertoire courant.

Un processus possède également un répertoire racine qui correspond en principe à la racine de l'arborescence de fichiers. Ce répertoire racine peut cependant être modifié par un processus possédant les droits du super-utilisateur par un appel à la primitive `chroot()` :

```
#include <unistd.h>
int chroot(const char *ref);
```

Le paramètre `ref` spécifie le nom du nouveau répertoire racine. En cas de succès (retour 0), le processus est restreint à l'accès des seuls fichiers placés dans la sous-arborescence dépendant de son nouveau répertoire racine.

Un processus peut modifier son répertoire courant en invoquant les primitives `chdir()` et `fchdir()` :

```
#include <unistd.h>
int chdir(const char *ref);
int fchdir(int desc);
```

La primitive `chdir()` prend comme premier paramètre le nom du nouveau répertoire courant `ref` tandis que la primitive `fchdir()` prend comme premier paramètre le descripteur du répertoire `desc` déjà ouvert par un appel à la primitive `open()`.

Par ailleurs, un processus peut connaître le nom de son répertoire courant en appelant la primitive `getcwd()` :

```
#include <unistd.h>
char *getcwd(char *buf, size_t taille);
```

Le nom absolu du répertoire courant du processus appelant est retourné dans le tampon `buf` dont la taille est spécifiée dans le paramètre `taille`. En cas de succès, la primitive renvoie un pointeur sur le tampon `buf` et sinon la valeur NULL.

b) Création d'un répertoire

Un répertoire est créé par un appel à la primitive `mkdir()` dont le prototype est :

```
#include <sys/types.h>
#include <fnctl.h>
#include <unistd.h>
int mkdir (const char *ref, mode_t mode);
```

Le premier paramètre `ref` spécifie le nom du répertoire à créer. Le champ `mode` spécifie les droits d'accès associé à ce répertoire et prend les mêmes valeurs que le troisième paramètre de la primitive `open()`.

En cas de succès, la primitive renvoie le descripteur associé au répertoire. En cas d'échec la primitive retourne la valeur – 1 et la variable `errno` prend l'une des valeurs suivantes :

- `EACESS`, l'accès n'est pas possible ;
- `EEXIST`, un fichier de même nom existe déjà ;
- `EFAULT`, le chemin spécifié n'est pas valide ;
- `ENAMETOOLONG`, le chemin spécifié comprend un nom trop long ;
- `ENOTDIR`, l'un des composants du chemin d'accès n'est pas un répertoire ;
- `ELOOP`, un cycle de liens symboliques est détecté ;
- `EMFILE`, le nombre maximal de fichiers ouverts pour le processus est atteint ;
- `ENOMEM`, la mémoire est insuffisante ;
- `EROFS`, le système de gestion de fichiers est en lecture seule ;
- `ENOSPC`, le système de fichiers est saturé.

c) Destruction d'un répertoire

Un répertoire est détruit par un appel à la primitive `rmdir()` dont le prototype est :

```
#include <unistd.h>
int rmdir (const char *ref);
```

Le premier paramètre `ref` spécifie le nom du répertoire à détruire.

En cas d'échec la primitive retourne la valeur – 1 et la variable `errno` prend l'une des valeurs suivantes :

- `EACESS`, l'accès n'est pas possible ;
- `EBUSY`, le répertoire à détruire est un répertoire courant de processus ou un répertoire racine ;
- `EFAULT`, le chemin spécifié n'est pas valide ;
- `ENAMETOOLONG`, le chemin spécifié comprend un nom trop long ;
- `ENOENT`, le répertoire désigné n'existe pas ;
- `ELOOP`, un cycle de liens symboliques est détecté ;
- `ENOTEMPTY`, le répertoire à détruire n'est pas vide ;
- `ENOMEM`, la mémoire est insuffisante ;
- `EROFS`, le système de gestion de fichiers est en lecture seule.

d) Exploration d'un répertoire

Trois primitives permettent aux processus d'accéder aux entrées d'un répertoire. Ce sont les appels systèmes `opendir()`, `readdir()` et `closedir()` qui permettent respectivement d'ouvrir un répertoire, de lire une entrée du répertoire et de fermer le répertoire. Les prototypes de ces trois fonctions sont :

```
#include <sys/types.h>
#include <dirent.h>
#include <unistd.h>
DIR *opendir (const char *ref);
struct dirent *readdir (DIR *rep);
int closedir (DIR *rep);
```

Le type `DIR` défini dans le fichier `<dirent.h>` représente un descripteur de répertoire ouvert. La structure `struct dirent` correspond à une entrée de répertoire. Elle comprend les champs suivants :

- `long d_ino`, le numéro d'inode de l'entrée ;
- `unsigned short d_reclen`, la taille de la structure retournée ;
- `char [] d_name`, le nom de fichier de l'entrée.

La primitive `opendir()` ouvre le répertoire `ref` en lecture et retourne un descripteur pour ce répertoire ouvert.

La primitive `readdir()` lit une entrée du répertoire désigné par le descripteur `rep`. Cette entrée est retournée dans une structure de type `struct dirent`.

La primitive `closedir()` ferme le répertoire désigné par le descripteur `rep`.

e) Un exemple

Dans cet exemple, le répertoire courant du processus est ouvert et chacune des entrées de celui-ci est lue et les informations associées (numéro d'inode, nom du fichier) sont affichées.

```
/*********************************************************************/
/*              Exemple de manipulation d'un répertoire              */
/*********************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <dirent.h>

main()
{
char nom[20];
struct dirent *entree;
DIR *fd;

getcwd(nom,20);
```

```
printf ("Mon répertoire courant est %s\n", nom);

fd = opendir (nom);
entree = readdir(fd);
while(entree != NULL)
{
 printf ("le numéro d'inode de l'entrée est %d et le nom de fichier
correspondant est %s\n", entree->d_ino, entree->d_name);
 entree = readdir(fd);
}

closedir(fd);
}
```

Une partie des traces d'exécutions est :

```
Mon répertoire courant est /root
le numéro d'inode de l'entrée est 1073953 et le nom de fichier
correspondant est .
le numéro d'inode de l'entrée est 2 et le nom de fichier correspondant
est ..
le numéro d'inode de l'entrée est 1073954 et le nom de fichier
correspondant est .exrc
le numéro d'inode de l'entrée est 1073955 et le nom de fichier
correspondant est .xinitrc
le numéro d'inode de l'entrée est 357987 et le nom de fichier
correspondant est bin
le numéro d'inode de l'entrée est 98213 et le nom de fichier
correspondant est .k4de
le numéro d'inode de l'entrée 1076353 et le nom de fichier correspondant
est .bash_history
```

3.4.3 Opérations sur les liens symboliques

Le lien symbolique correspond à la création d'un fichier de type particulier (l), qui contient comme donnée un pointeur vers un autre fichier.

Les primitives `symlink()` et `readlink()` permettent respectivement de créer un lien symbolique et de lire le nom du fichier sur lequel il pointe. Les prototypes de ces 2 fonctions sont :

```
#include <unistd.h>
int symlink (const char *ancref, const char *nvref);
int readlink (const char *ref, char *buf, size_t taille);
```

La primitive `symlink()` crée un lien symbolique nommé `nvref`, qui pointe sur le fichier `ancref`. La commande `ln -s ancref nvref` réalise la même action depuis l'interpréteur de commandes.

La primitive `readlink()` retourne dans le tampon `buf` dont la taille est spécifiée dans le paramètre `taille`, le nom de fichier pointé par le lien `ref`. Elle rend par ailleurs le nombre de caractères placés dans `buf`.

3.4.4 Opérations sur les partitions

Les primitives `mount()` et `umount()` permettent de monter ou démonter une partition sur l'arborescence de fichiers de façon à la rendre accessible ou inaccessible. Les prototypes de ces primitives sont :

```
#include <sys/mount.h>
#include <linux/fs.h>
int mount (const char *fichierspecial, const char *dir, const char *sgf,
unsigned long flag, const void *data);
int umount (const char *fichierspecial);
int umount (const char *dir);
```

La primitive `mount ()` monte, sur le répertoire `dir`, le système de gestion de fichiers présent sur le périphérique dont le nom est donné dans le paramètre `fichierspecial`. Le paramètre `sgf` indique le type du système de gestion de fichiers et peut prendre les valeurs « minix », « ext2 », « proc », etc. Le paramètre `flag` spécifie les options de montage sous forme de constantes qui peuvent être combinées par l'opération de OU binaire « | ». Les valeurs admises pour ces constantes sont :

- `MS_RONLY`, les fichiers montés sont accessibles en lecture seule ;
- `MS_NOSUID`, les bits `setuid` et `setgid` ne sont pas utilisés ;
- `MS_NODEV`, les fichiers spéciaux ne sont pas accessibles ;
- `MS_NOEXEC`, les programmes ne peuvent pas être exécutés ;
- `MS_SYNCHRONOUS`, les écritures sont synchrones.

Le paramètre `data` spécifie d'autres options dépendantes du type de système de gestion de fichiers.

La primitive `umount()` démonte un système de gestion de fichiers. Elle prend comme paramètre soit le nom du système de gestion de fichiers `(fichierspecial)`, soit le nom du point de montage `(dir)`.

Ces deux primitives ne peuvent être exécutées que par le super-utilisateur.

3.5 Le système de fichiers /proc

Le système de gestion de fichiers `/proc`, supporté par le VFS, est un système de gestion de fichiers particulier qui ne contient pas de données stockées sur le disque. Son rôle est de fournir à l'utilisateur un ensemble d'informations sur le noyau et les processus, à travers une interface de fichiers virtuels accessibles par l'interface du VFS.

La figure 3.19 illustre le contenu du répertoire `/proc`. On y trouve un ensemble de fichiers qui sont essentiellement de trois types :

- des fichiers d'informations sur le noyau et la machine tels que `cmdline` (arguments passés au noyau lors du démarrage), `cpuinfo` (description du ou des processeurs de la machine), `devices` (liste des gestionnaires de périphériques), `filesystems` (liste des systèmes de gestion de fichiers supportés par le noyau), `interrupts` (liste des interruptions matérielles utilisées par les gestionnaires de

périphériques), `meminfo` (état de la mémoire centrale), modules (liste des modules chargés dans le noyau), etc. ;

- des répertoires tels que `net` (fichiers concernant les protocoles réseau), `scsi` (fichiers concernant les gestionnaires de périphériques scsi), etc. ;
- un répertoire par processus actifs sur la machine, dont le nom est égal au PID du processus. Ce répertoire comprend un ensemble de fichiers tels que `cmdline` (liste des arguments du processus), `cwd` (lien sur le répertoire courant du processus), `environ` (les variables d'environnements du processus), `exe` (lien sur le fichier exécutable du processus), `fd` (répertoire contenant des liens sur les fichiers ouverts par le processus), `maps` (liste des zones mémoire contenues dans l'espace d'adressage du processus), `mem` (contenu de l'espace d'adressage du processus), `root` (lien sur le répertoire racine du processus), `stat`, `statm` et `status` (état du processus).

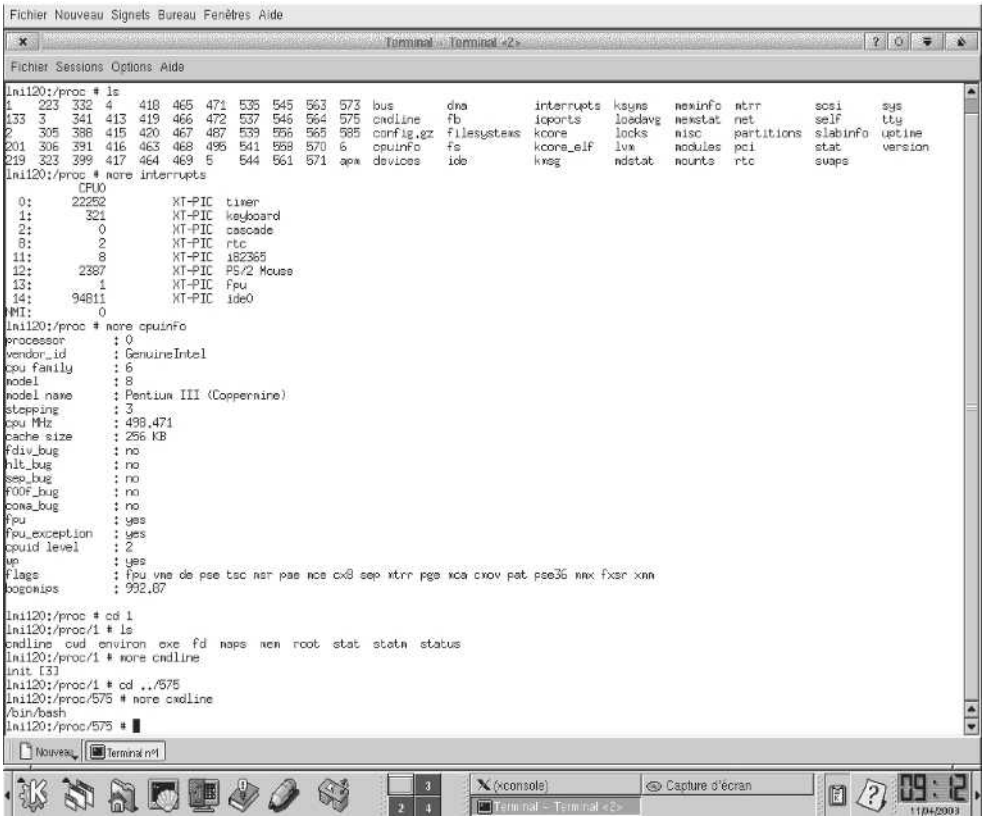

Figure 3.19 – Le système de fichiers /proc

Exercices

3.1 Qu'avez-vous retenu ?

Pour chacune des questions suivantes, choisissez la ou les réponses qui vous semblent exactes.

Question 1 – Quel est le rôle du VFS ?

- ☐ a. permettre l'accès à des données distantes.
- ☐ b. gérer les accès disque.
- ☐ c. uniformiser l'accès à de multiples systèmes de gestion de fichiers.

Question 2 – Une inode :

- ☐ a. est un descripteur de processus.
- ☐ b. est un descripteur de partition.
- ☐ c. est un descripteur de fichier.

Question 3 – Le répertoire est une structure qui permet :

- ☐ a. de sauvegarder le contexte d'un processus.
- ☐ b. de localiser un fichier au sein d'une partition.

Question 4 – Un lien physique :

- ☐ a. est un fichier qui a pour donnée un pointeur vers un autre fichier.
- ☐ b. est une association d'un nom à un fichier.
- ☐ c. est une méthode d'accès.

Question 5 – Un lien symbolique :

- ☐ a. est un fichier qui a pour donnée un pointeur vers un autre fichier.
- ☐ b. est une association d'un nom à un fichier.
- ☐ c. est une méthode d'accès.

Question 6 – Le fichier « toto » est associé aux droits `rwxr-xr--`. Quelles propositions énoncées sont exactes ?

- ☐ a. le propriétaire peut lire, écrire et exécuter le fichier.
- ☐ b. le propriétaire peut lire le fichier et le groupe peut exécuter le fichier.
- ☐ c. les autres peuvent exécuter le fichier.

Question 7 – Une partition est :

- ☐ a. créée lors du formatage physique et correspond au plus petit élément accessible sur le disque dur.
- ☐ b. un sous-espace du support de masse dans lequel un système de gestion de fichiers peut être installé.

Question 8 – Un fils :

- ☐ a. hérite des fichiers ouverts par son père et partage avec celui-ci le pointeur de fichier.
- ☐ b. hérite des fichiers ouverts par son père et obtient un nouveau pointeur de fichier.
- ☐ c. n'hérite pas des fichiers ouverts par son père.

Question 9 – Le *buffer* cache :

- ☐ a. est un mécanisme de cache du processeur.
- ☐ b. est un mécanisme de cache système qui mémorise les pages mémoires les plus récemment accédées.
- ☐ c. est un mécanisme de cache qui mémorise les blocs disques les plus récemment accédés.

3.2 Arborescence et droits

On dispose d'une arborescence de fichiers Linux décrite comme sur la figure 3.20.

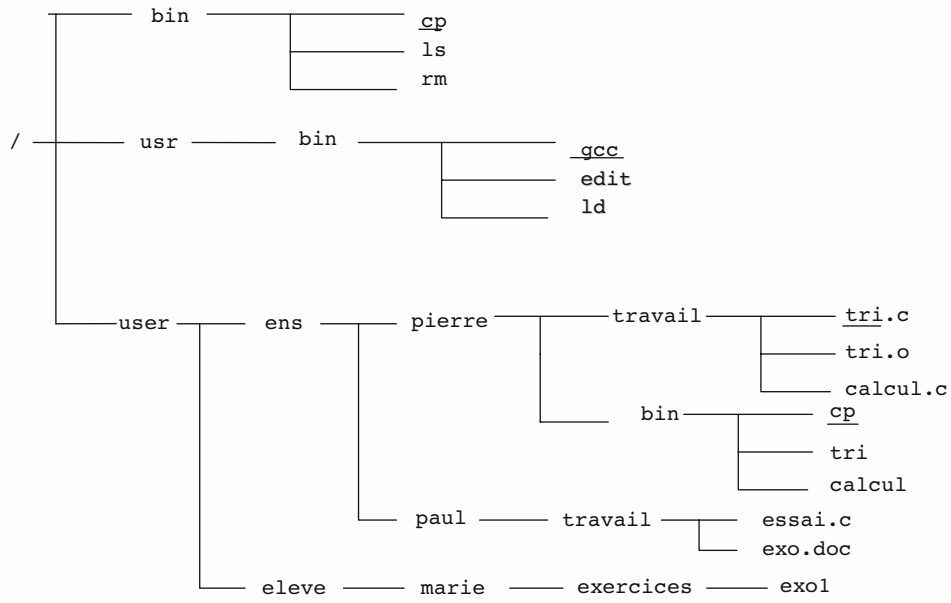

Figure 3.20

1) Donnez le chemin d'accès complet pour les 4 fichiers apparaissant soulignés. On utilisera le séparateur « / ».
2) L'utilisateur Pierre se trouve positionné dans son répertoire courant « `user/ens/pierre` ». Donnez le chemin absolu puis le chemin relatif correspondant au fichier `calcul.c`.

3) Les utilisateurs Pierre et Paul appartiennent tous les deux au groupe des enseignants (`ens`) tandis que l'utilisateur Marie appartient au groupe des élèves (`eleve`). Les droits associés au fichier `calcul.c` de l'utilisateur Pierre sont les suivants :

```
-rw- rw- r--
```

Paul peut-il modifier le fichier ? Marie peut-elle modifier le fichier ?

3.3 Fichiers Ext2

À un instant *t*, un fichier Mesure est un fichier Ext2 de 458 blocs de 1 024 octets. Une adresse de bloc occupe 4 octets. Faites un schéma pour représenter la structure du fichier. Combien d'accès nécessite la lecture séquentielle du 421e bloc par un processus sachant que la lecture se fait par bloc de 1 024 octets et que le système utilise un mécanisme de cache ?

3.4 Fichiers Ext2

Un processus lit séquentiellement un fichier Ext2 de 8 Mo, à raison de 256 octets à la fois. On suppose que les blocs disque sont de 1 024 octets et qu'un numéro de bloc occupe 4 octets. Par ailleurs, le temps d'accès moyen au disque est de 50 ms.

1) Le système ne gère pas de mécanisme de cache au niveau du disque, c'est-à-dire que chaque demande de lecture équivaut à une opération d'entrées-sorties. Donnez le nombre total d'accès disque nécessaire et le temps d'attente en entrées-sorties.
2) Le système gère un mécanisme de cache au niveau du disque, qui mémorise en mémoire centrale les 100 blocs disques les plus récemment accédés. Donnez le nombre total d'accès disque nécessaire et le temps d'attente en entrées-sorties.

3.5 Fichiers Ext2

Un fichier Unix occupe 10 254 blocs de données. Ces blocs sont de 1 024 octets. Les adresses de blocs sont sur 4 octets. Choisissez la réponse qui vous paraît juste, parmi celles qui vous sont proposées.

1) Les 12 premières entrées de la table de l'*i*-nœud contiennent les entrées de 12 blocs de données ? Oui/Non.
2) Un bloc d'adresse contient 64 adresses ? Oui/Non.
3) Combien de niveaux d'indirection sont utilisés pour stocker les données de ce fichier ? 0 niveau/1 niveau/2 niveaux/3 niveaux/4 niveaux.
4) Combien de blocs d'adresses (BA), tous niveaux d'indirection confondus ont été alloués ? 0 BA/1 BA/42 BA/53 BA/75 BA/128 BA/10 254 BA.

3.6 Fichier Linux

Soit un fichier Linux de 64 Mo. Les blocs disque dont de 1 Ko. Un numéro de bloc occupe 2 octets. Le temps d'un d'accès disque est de 10 ms.

1) Le nombre de blocs de données du fichier est 65 536, 64 ou 1 684 ?
2) Le nombre de blocs d'index du fichier est 127, 129, 32 ou 2 ?
3) Le temps d'accès pour lire les 3 000 premiers blocs de données est 30 070 ms, 30 050 ms, 30,07 µs ou 3 s.

3.7 Héritage père – fils

1) Écrivez un programme dans lequel un processus ouvre un fichier en lecture, ce fichier contenant la phrase « bonjour les petits amis ». Puis le processus crée un processus fils. Chacun des deux processus lit alors le contenu du fichier à raison de 4 caractères à la fois et affiche sur la sortie standard les 4 caractères lus. Le fils doit-il ouvrir le fichier à son tour ? Que montrent les traces d'exécution que vous obtenez ?
2) Le processus fils recouvre à présent le code hérité de son père par le contenu de l'exécutable `lire_fichier`. On souhaite obtenir exactement le même comportement que pour la question 1. Écrivez le programme correspondant au processus père et celui du programme `lire_fichier.c`.

Des idées pour vous exercer à la programmation

Programme 1 – Un processus manipule un fichier dans lequel sont stockés des couples de données – nom de personne, âge. Le processus ouvre le fichier et lit l'ensemble des données présentes dans le fichier. Puis il trie ces données par ordre croissant d'âge et réécrit les données triées dans le fichier.

Programme 2 – Même principe mais à présent le processus réécrit les données triées dans un nouveau fichier placé dans un sous-répertoire créé par le processus.

Solutions

3.1 Qu'avez-vous retenu ?

Question 1 – Quel est le rôle du VFS ?

- ☐ c. uniformiser l'accès à de multiples systèmes de gestion de fichiers.

Question 2 – Une inode :

- ☐ c. est un descripteur de fichier.

Question 3 – Le répertoire est une structure qui permet :

- ☐ b. de localiser un fichier au sein d'une partition.

Question 4 – Un lien physique :

- ☐ b. est une association d'un nom à un fichier.

Question 5 – Un lien symbolique :

- ☐ a. est un fichier qui a pour donnée un pointeur vers un autre fichier.

Question 6 – Le fichier « toto » est associé aux droits `rwxr-xr--`. Quelles propositions énoncées sont exactes ?

- ☐ a. le propriétaire peut lire, écrire et exécuter le fichier.
- ☐ b. le propriétaire peut lire le fichier et le groupe peut exécuter le fichier.

Question 7 – Une partition est :

- ☐ b. un sous-espace du support de masse dans lequel un système de gestion de fichiers peut être installé.

Question 8 – Un fils :

- ☐ b. hérite des fichiers ouverts par son père et obtient un nouveau pointeur de fichier.

Question 9 – Le *buffer* cache :

- ☐ c. est un mécanisme de cache qui mémorise les blocs disques les plus récemment accédés.

3.2 Arborescence et droits

1) `/usr/bin/gcc ; /bin/cp ; /user/ens/pierre/travail/tri.c ; /user/ens/pierre/bin/cp`
2) Le chemin absolu est : `/user/ens/pierre/travail/calcul.c`.
   Le chemin relatif est `travail/calcul.c`.
3) Paul peut modifier le fichier puisque le groupe des enseignants a des droits en lecture et écriture sur le fichier. Par contre, Marie ne peut pas ; elle dispose seulement de droits en lecture.

3.3 Fichiers Ext2

Le schéma du fichier est donné par la figure 3.21.

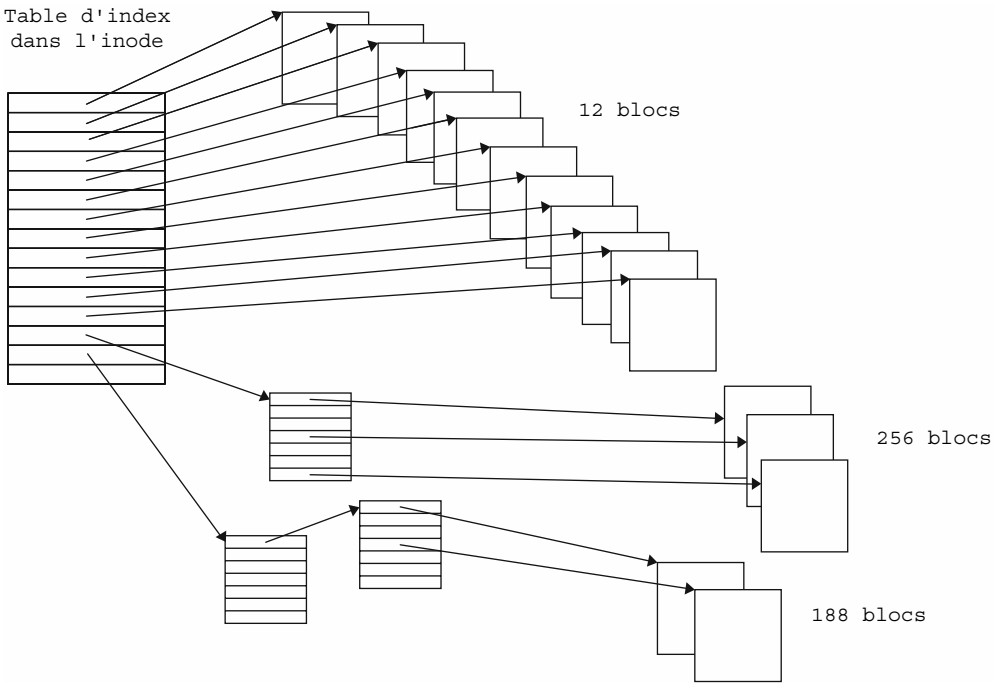

Figure 3.21 – Structure du fichier

Pour lire séquentiellement le 421e bloc du fichier, il faut 12 + (1 + 256) + (2 + 153) accès soit 424 accès disque.

3.4 Fichiers Ext2

1) Lecture des 12 premiers blocs : 4 × 12 accès disque.
   Lecture des 256 blocs suivants (niveau d'indirection 1) : on a deux accès disque par lecture donc 8 × 256 accès disque.
   Lecture des 7 924 blocs restants (niveau d'indirection 2) : on a trois accès disque par lecture donc 12 × 7 924 accès disque.
   Soit un total de 97 184 accès disque et une durée moyenne de lecture égale à 4 859 s.
2) On a un accès disque par blocs de données lors de la lecture des 256 premiers octets du bloc. On a un total de 32 blocs d'adresse à lire, soit un nombre d'accès disque égal à 8 192 + 32 = 9 223 et un temps moyen de lecture égal à seulement 461 s !!!

3.5 Fichiers Ext2

1) Oui.
2) Non : 1 024/4 = 256.
3) 2 niveaux.
4) 42 BA (1 BA en première indirection et 41 BA en seconde indirection).

3.6 Fichier Linux

Soit un fichier Linux de 64 Mo. Les blocs disque dont de 1 Ko. Un numéro de bloc occupe 2 octets. Le temps d'un d'accès disque est de 10 ms.

1) Le nombre de blocs de données du fichier est $64 \times 2^{20} / 1\,024 = 65\,536$.
2) Le nombre de blocs d'index du fichier est :
   - Un bloc d'index comporte 512 entrées.
   - Les entrées directes et le premier niveau d'indirection permettent d'allouer 524 blocs de données. Il reste donc 65 536 – 524 = 65 012 blocs de données.
   - 65 012 / 512 = 126,9.
   - Il y a donc au total 129 blocs d'index.
3) Le temps d'accès pour lire les 3 000 premiers blocs de données est (129 + 65 536) × 10 = 30 070 ms = 30,07 µs.

3.7 Héritage père – fils

1) Le processus fils hérite du descripteur de fichier obtenu par son père, il n'a donc pas à ouvrir lui-même le fichier. Par contre, du fait, de cet héritage, le père et le fils partagent le pointeur de position dans le fichier, d'où les traces suivantes par exemple :

```
valeur fp à l'ouverture, 3
hello père; bonj
hello fils; our
hello père; les
hello père; peti
hello fils; ts a
```

```
hello fils; mis
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int i;
main()
{
int pid, fp,j;
char ch[4];

fp = open("fichier", O_RDONLY);
printf("valeur fp à l'ouverture, %d\n", fp);
pid = fork();
if (pid==0)
{
    for(j=0;j<3;j++)
    {
     read(fp,ch,4);
     printf("hello fils; %s\n", ch);
    }
close(fp);
}
else
{
    for(j=0;j<3;j++)
    {
     read(fp,ch,4);
     printf("hello père; %s\n", ch);
    }
wait();
close(fp);
}
}
```

2) Programmes processus père et `lire_fichier.c`

```
/*****************************************************************/
/*                          Processus père                       */
/*****************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
main()
{
int pid, fp,j;
char ch[4],conv[11];

fp = open("fichier", O_RDONLY);
```

```
printf("valeur fp a l'ouverture, %d\n", fp);
pid = fork();
if (pid == 0)
{
  sprintf (conv, "%d", fp);
  execlp("/root/lire_fichier","lire_fichier",conv, NULL);
  perror ("pb exec");
}
else
{
  j = 0;
  while (j<3)
  {
   read(fp,ch,4);
   printf("hello père; %s\n", ch);
   j = j + 1;
  }
  wait();
  close(fp);
}
}
/*****************************************************************/
/*            Programme lire_fichier.c: processus fils           */
/*****************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main(argc, argv)
char **argv;
{
int fp, i;
char ch[4];
fp = atoi(argv[1]);
printf("valeur fp fils, %d\n", fp);

i = 0;
while (i < 3)
{
  printf("%d", i);
  read(fp,ch,4);
  printf("hello fils; %s\n", ch);
  i = i +1;
}
close(fp);
exit();
}
```

GESTION DES ENTRÉES-SORTIES

4

PLAN

OBJECTIFS

- Dans ce chapitre, nous présentons tout d'abord les principes généraux relatifs à la gestion des entrées-sorties par le matériel et par le système d'exploitation.
- Puis nous décrivons le sous-système d'entrées-sorties de Linux basé sur la notion de fichiers spéciaux.

4.1 PRINCIPES GÉNÉRAUX

Le processeur de l'ordinateur dialogue avec l'extérieur par le biais des périphériques d'entrées-sorties dont les principaux sont par exemple le clavier, la souris, l'écran, l'imprimante, le disque dur, le scanner, le lecteur DVD, les enceintes audio, etc. La multitude des périphériques est grande. De plus, les caractéristiques techniques de ces différents périphériques sont très diverses et la manière de réaliser une opération d'entrées-sorties peut être en conséquence très différente. La gestion des périphériques s'effectue à deux niveaux :

- au niveau du matériel avec le dispositif de l'*unité d'échange* ou contrôleur d'entrées-sorties ;
- au niveau du système d'exploitation, avec le *pilote* d'entrées-sorties.

4.1.1 L'unité d'échange

Au niveau matériel, le processeur s'interface avec le périphérique par le biais de l'*unité d'échange* ou *contrôleur d'entrées-sorties*.

L'unité d'échange offre une interface standard d'entrées-sorties au processeur dans le sens où elle convertit son dialogue avec le processeur dans les signaux spécifiques à l'interface du périphérique, qu'elle est seule à connaître (figure 4.1).

L'unité d'échange adapte ainsi les caractéristiques électriques et logiques des signaux émis ou reçus par les périphériques à celles des signaux véhiculés par les bus.

L'unité d'échange est constituée par un ensemble de registres adressables par le processeur. Leur nombre et leur gestion dépendent des spécificités particulières de l'unité d'échange. Fonctionnellement on trouve :

- un *registre d'état* qui permet de connaître l'état du périphérique piloté par cette unité d'échange. Chaque état est caractérisé par une valeur numérique disponible dans le registre d'état. Par exemple une imprimante qui n'a plus de papier émet un signal vers le contrôleur qui peut alors charger le registre d'état avec la valeur numérique correspondant à cet état. Cette information maintenant placée dans le registre d'état est disponible pour le processeur. Une autre information importante positionnée dans le registre d'état indique si le périphérique attaché à l'unité d'échange est prêt ou non à effectuer un nouveau transfert de donnée ;
- un *registre de commandes* dans lequel le processeur vient décrire l'opération d'entrées-sorties à réaliser ;
- un *registre de données*. C'est au travers de ce registre que se font les échanges de données entre la mémoire et l'unité d'échange.

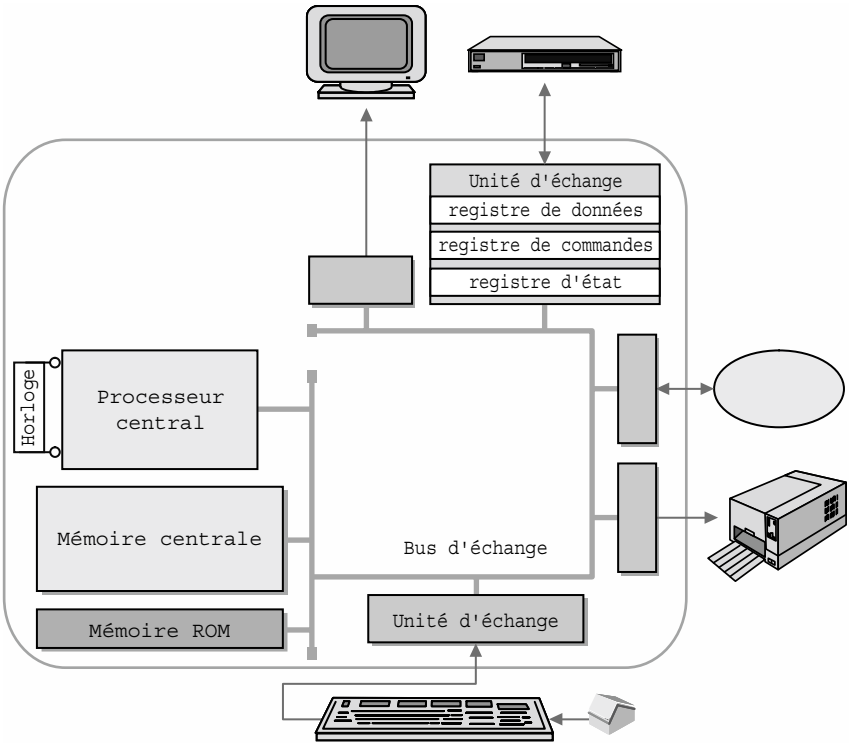

Figure 4.1 – Unité d'échange

4.1.2 Le pilote et les modes d'entrées-sorties

a) Notion de pilote

Au niveau du système d'exploitation, la réalisation des opérations d'entrées-sorties est prise en charge par un programme particulier, appelé *pilote d'entrées-sorties* ou *driver*. Ce pilote maintient des structures de données qui décrivent l'état des unités d'échange associées et un ensemble de fonctions pour agir sur les périphériques :

- `open` : cette fonction permet d'ouvrir un périphérique c'est-à-dire qu'elle initialise le périphérique et les structures de données associées ;
- `close` : cette fonction rompt la liaison avec le périphérique ;
- `read`, `write` : ces fonctions permettent d'effectuer des lectures ou des écritures de données sur ou à partir des périphériques ;
- `schedule` : cette fonction permet d'ordonnancer les requêtes d'entrées-sorties, de façon à optimiser les temps de réalisation des opérations ;
- `interrupt` : cette fonction constitue le gestionnaire d'interruption associé au périphérique et est exécutée lorsque celui-ci lève une interruption à destination du processeur.

b) Protocoles d'entrées-sorties

Le pilote peut dialoguer avec l'unité d'échange selon trois protocoles différents :

- le mode par scrutation ou attente active ;
- le mode avec interruption ;
- le mode avec DMA (*Direct Memory Access*).

Le protocole avec scrutation ou attente active

Dans ce protocole, le pilote scrute en permanence l'unité d'échange pour savoir si elle prête à recevoir ou à délivrer une nouvelle donnée. Cette scrutation s'effectue par une lecture régulière du contenu du registre d'état de l'unité d'échange.

Considérons par exemple que le pilote doit réaliser l'impression de `nb_caractères` caractères. L'algorithme déroulé est alors le suivant :

```
programme pilote:
{
lire le registre d'état;
while (nb_caractères > 0) {
while (périphérique_occupé)
    lire le registre d'état;
écrire un caractère dans le registre de données;
nb_caractères = nb_caractères - 1; }
}
```

Le processeur durant l'opération d'entrées-sorties est totalement occupé par l'exécution du pilote. On peut noter que le travail effectué n'est pas très productif puisqu'il comporte une grande part d'attente active (lecture du registre d'état tant que le périphérique est occupé).

Le protocole avec interruption

Avec ce protocole, l'unité d'échange utilise le mécanisme des interruptions pour signaler qu'elle est prête. Un gestionnaire d'interruption `interrupt()` est associé au pilote du périphérique. À la réception d'une interruption le programme en cours d'exécution est arrêté au profit de ce gestionnaire. L'algorithme précédent devient à présent :

```
programme pilote:
{
lire le registre d'état;
while (périphérique_occupé)
    lire le registre d'état;
écrire le premier caractère dans le registre de données;
nb_caractères = nb_caractères - 1;
enable_it_imprimante;
}

gestionnaire interrupt():
{
disable_it_imprimante;
if (nb_caractères > 0) {
    écrire le caractère suivant dans le registre de données;
    nb_caractères = nb_caractères - 1; }
enable_it_imprimante;
}
```

Ce mode de prise en compte de l'opération d'entrées-sorties libère le processeur sauf pendant les périodes où le processeur est attribué au programme de gestion de l'interruption et au pilote.

Le mode avec DMA

Le dispositif DMA (*Direct Memory Access*) est un composant matériel permettant d'effectuer des échanges entre mémoire centrale et unité d'échange sans utilisation du processeur central. Le DMA comprend :

- un registre d'adresse qui reçoit l'adresse du premier caractère à transférer ;
- un registre de comptage qui reçoit le nombre de caractères à transférer ;
- un registre de commande qui reçoit le type d'opération à effectuer (lecture ou écriture) ;
- une zone tampon permettant le stockage de données ;
- un composant actif, de type processeur, qui exécute un transfert sans utilisation du processeur central.

Le programme pilote effectue simplement les opérations d'initialisation du DMA : chargement des registres puis lancement du processeur périphérique. À partir de ce moment le transfert s'effectue sans utilisation du processeur central qui est alors libre d'effectuer un autre travail. À la fin du transfert, une interruption est émise qui interrompt le programme en cours d'exécution au profit du programme de gestion de l'interruption. L'algorithme pour la réalisation de l'impression devient maintenant :

```
programme pilote:
{
registre_adresse_DMA = adresse du premier caractère en mémoire centrale;
registre_comptage_DMA = nb_caractères;
registre_commande_DMA = écriture;
enable_it_imprimante;
}

gestionnaire interrupt():
{
disable_it_imprimante;
vérifier que le transfert s'est bien passé;
}
```

c) Ordonnancement des requêtes

Le pilote peut comporter une fonction `schedule` dont le rôle est d'ordonnancer les requêtes à destination du périphérique qu'il contrôle. Ceci est notamment vrai pour les requêtes à destination du disque dur. L'accès à un secteur du disque se décompose en deux parties :

- le *temps de positionnement* du bras correspond au temps nécessaire pour que le bras portant la tête de lecture vienne se positionner sur la piste contenant le secteur ;
- le *temps de latence* correspond au temps nécessaire pour qu'une fois la tête placée sur la bonne piste, le secteur passe sous la tête de lecture. Ce temps de latence dépend de la vitesse de rotation du disque.

Le temps de positionnement est la partie la plus pénalisante pour la réalisation d'une opération d'entrées-sorties sur le disque. Afin de réduire cette pénalité induite par les mouvements du bras, le pilote ordonnance les requêtes de la file d'attente du contrôleur de manière à réduire ces mouvements. Les principaux algorithmes d'ordonnancement sont :

- FCFS (*First Come, First Served*) ;
- SSTF (*Shortest Seek Time First*) ;
- SCAN (ascenseur ou balayage).

Nous les présentons dans les paragraphes suivants avec un exemple appliqué à la file de requêtes disque suivante, où chaque nombre représente un numéro de piste à atteindre : 50, 110, 25, 105, 12, 100, 40, 45, 10, 80, 88. Le bras est initialement sur la piste 90. Le disque comporte 150 pistes par plateau.

Algorithme FCFS

Avec cet algorithme, les requêtes disque sont servies selon leur ordre de soumission. Cette politique, si elle est simple à mettre en œuvre, ne minimise pas les mouvements du bras. Sur la file de requêtes définie dans l'exemple, le déplacement total du bras est égal à 624 pistes. La figure 4.2 illustre l'ordre de service des requêtes et le déplacement du bras qui s'ensuit.

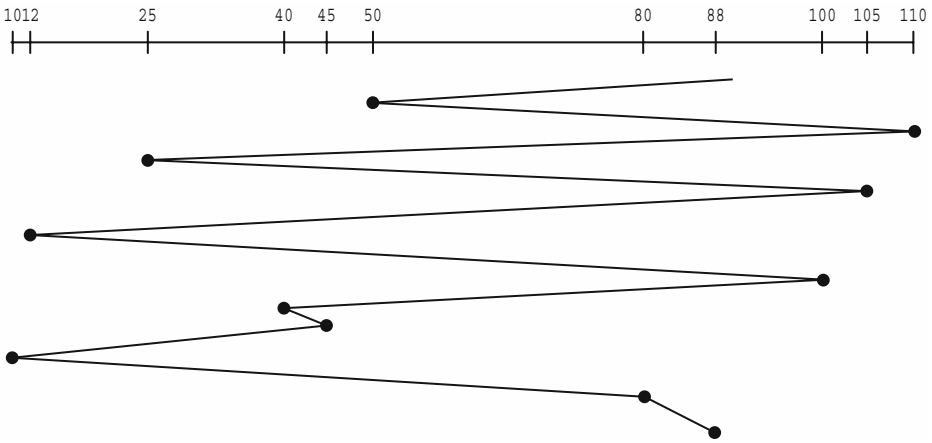

Figure 4.2 – **Algorithme FCFS**

Algorithme SSTF

Avec cet algorithme, la prochaine requête disque servie est celle dont la position est la plus proche de la position courante, donc celle induisant le moins de déplacement du bras. Le problème de cette politique est le risque de famine pour des requêtes excentrées par rapport au point de service courant. Sur la file de requêtes définie dans l'exemple, l'ordre de service est 88, 80, 100, 105, 110, 50, 45, 40, 25, 12, et 10 et le déplacement total du bras est égal à 140 pistes. La figure 4.3 illustre l'ordre de service des requêtes et le déplacement du bras qui s'ensuit.

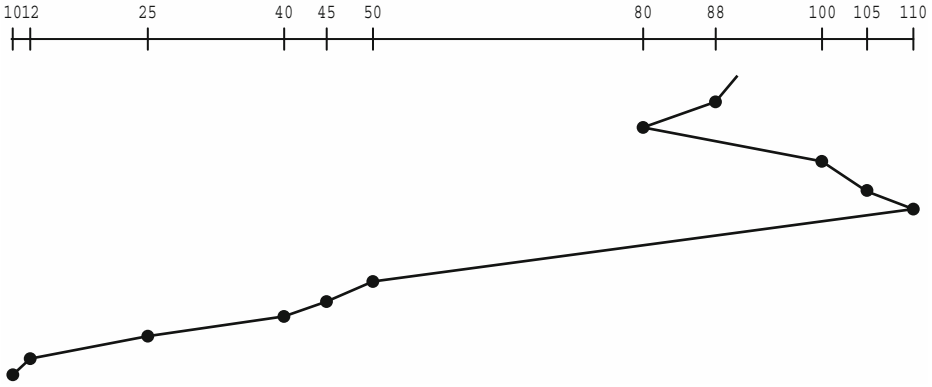

Figure 4.3 – **Algorithme SSTF**

Algorithme SCAN

C'est un algorithme de type balayage. La tête de lecture/écriture démarre à une extrémité du disque et parcourt celui-ci jusqu'à l'autre extrémité, en servant les requêtes au fur et à mesure de chaque piste rencontrée. Quand la tête de lecture/écriture est

parvenue à l'autre extrémité du disque, son mouvement s'inverse et le service continue en sens opposé. Sur la file de requêtes définie dans l'exemple, l'ordre de service est 100, 105, 110, 88, 80, 50, 45, 40, 25, 12, et 10 si l'on suppose un parcours initialement ascendant des pistes et le déplacement total du bras est égal à 200 pistes. La figure 4.4 illustre l'ordre de service des requêtes et le déplacement du bras qui s'ensuit.

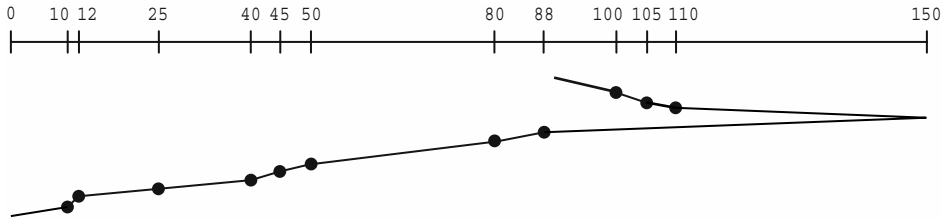

Figure 4.4 – **Algorithme SCAN**

Une variante de cette politique est la politique C-SCAN. Le principe de cet algorithme est le même que le précédent. La seule différence consiste dans le mouvement du bras lorsqu'il arrive à l'autre extrémité du disque. Au lieu de repartir en sens inverse, le bras va ici se repositionner sur l'extrémité initiale de son parcours. Ainsi, le parcours des pistes se fait toujours dans le même sens. Cette politique offre pour chaque requête disque un temps de traitement plus uniforme que la politique précédente. Sur la file de requêtes définie dans l'exemple, l'ordre de service est 100, 105, 110, 10, 12, 25, 40, 45, 50, 80 et 88 et le déplacement total du bras est égal à 260 pistes (figure 4.5).

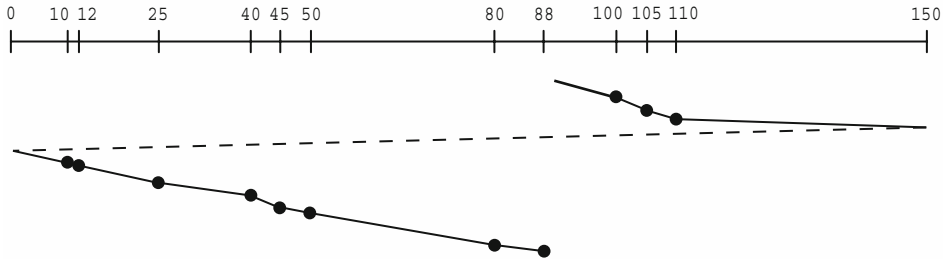

Figure 4.5 – **Algorithme C-SCAN**

4.2 Entrées-sorties Linux

4.2.1 Fichiers spéciaux

Linux gère les entrées-sorties au travers des fichiers spéciaux que nous avons déjà évoqués au chapitre 3. Ainsi un même appel système de type `write` peut être utilisé pour écrire dans un fichier régulier ou pour lancer une impression, en écrivant sur le fichier spécial `/dev/lp0`, attaché au pilote de l'imprimante. Les fichiers spéciaux sont de deux types :

- les *fichiers de type bloc* (b) correspondent aux périphériques structurés en bloc pour lesquels l'accès direct est possible. L'accès à ces périphériques s'effectue au travers du cache de tampons (*buffer cache*) présenté au chapitre 3. Les périphériques disques et CD-ROM appartiennent à cette catégorie ;
- les *fichiers de type caractères* (c) correspondent aux périphériques sans structure, sur lesquels les données sont accessibles de façon séquentielle, octet par octet. Les périphériques comme l'imprimante, le terminal, le scanner, la carte son ou encore le joystick font partie de cette catégorie.

Chaque fichier spécial est identifié par 3 attributs :

- son type (c ou b) ;
- un numéro appelé *numéro majeur*, compris entre 1 et 255, qui identifie le pilote gérant le périphérique ;
- un numéro appelé *numéro mineur*, qui identifie pour un pilote, l'un des périphériques physiques qui lui est assigné.

Ainsi, les ports série sont gérés par un pilote identifié par le numéro majeur 5, chaque port série étant lui-même identifié par un numéro mineur allant de 1 à 3 (ports COM1, COM2, COM3).

À l'instar des systèmes de gestion de fichiers, les pilotes de périphériques s'enregistrent auprès du noyau, soit à la compilation de celui-ci, soit lors du chargement d'un module. Le noyau maintient deux tables de 256 entrées chacune, `blkdevs` et `chrdevs`, qui contiennent respectivement des descripteurs des périphériques blocs ou caractères qui lui sont connus. Chaque entrée donne le nom du pilote et les opérations liées au fichier spécial associé.

L'enregistrement des pilotes s'effectue au moyen des deux opérations suivantes:

```
register_chrdev (unsigned int major, const char *name, struct
file_operations *fops);
register_blkdev (unsigned int major, const char *name, struct
file_operations *fops);
```

Ainsi, un pilote pour un périphérique de type imprimante effectue l'appel suivant pour s'enregistrer auprès du noyau :

```
register_chrdev (LP_MAJOR, "lp", &lp_fops);
```

`LP_MAJOR` correspond au numéro majeur affecté à la classe des périphériques de type `lp` tandis que `lp_ops` est un pointeur sur une table des opérations associées au

périphérique. Lorsqu'un utilisateur effectue une opération `write` sur le fichier `/dev/lp0` par exemple, le VFS accède à l'entrée de la table `chrdev` concernant les périphériques de type `lp` et il remplace l'opération générique `write` par l'opération accessible *via* le pointeur `lp_ops` qui agit directement sur le périphérique.

À l'inverse, un pilote se désenregistre du noyau en faisant un appel à la fonction `unregister_blkdev()` ou `unregister_chrdev()` selon son type.

Un périphérique est ouvert par le noyau par un appel à la fonction `blkdev_open()` ou `chrdev_open()`. Ces deux fonctions obtiennent les opérations sur fichiers associées au périphérique puis ouvrent le périphérique. Si le périphérique est géré par interruption, le numéro de l'IRQ liée à celui-ci est récupéré et le gestionnaire d'interruption associé est installé. Un périphérique est fermé par la fonction `release()`.

La plupart des fichiers spéciaux sont placés dans le répertoire `/dev`. Nous donnons ici un extrait du résultat de la commande `ls -l /dev`. Les cinquième et sixième colonnes correspondent respectivement au numéro majeur identifiant le pilote et au numéro mineur identifiant le périphérique lié au pilote.

```
crw-rw-rw-   1 root   audio   35,   0 Aug  6  2000 midi0
crw-rw-rw-   1 root   audio   14,   2 Aug  6  2000 midi00
crw-rw----   1 root   video   81,   0 Aug  6  2000 video0
crw-rw----   1 root   video   81,   1 Aug  6  2000 video1
crw-rw-r--   1 root   root    10,  32 Aug  6  2000 usbmouse
crw-rw-r--   1 root   root    10,  32 Aug  6  2000 usbmouse0
crw-r--r--   1 root   root   180,  48 Aug  6  2000 usbscanner
brw-rw----   1 root   disk    94,   1 Aug  6  2000 dasda1
brw-rw----   1 root   disk    94,   4 Aug  6  2000 dasdb
brw-rw----   1 root   disk    94,   5 Aug  6  2000 dasdb1
brw-rw----   1 root   disk    94,   8 Aug  6  2000 dasdc
brw-rw----   1 root   disk     1,   0 Aug  6  2000 ram0
brw-rw----   1 root   disk     1,   1 Aug  6  2000 ram1
crw-rw----   1 root   lp       6,   0 Aug  6  2000 lp0
crw-rw----   1 root   lp       6,   1 Aug  6  2000 lp1
```

4.2.2 Appels systèmes

a) Création d'un fichier spécial

La primitive `mknod()` permet de créer un fichier spécial. Son prototype est :

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/sysmacros.h>
#include <fcntl.h>
#include <unistd.h>
int mknod (cons char *ref, mode_t mode, dev_t dev);
```

Le paramètre `ref` spécifie le nom du fichier à créer. Le paramètre `mode` donne les permissions et le type de fichier à créer. Ce type est représenté par l'une des constantes suivantes :

- `S_IFREG` : fichier régulier ;

- `S_IFCHR` : fichier spécial de type caractère ;
- `S_IFBLK` : fichier spécial de type bloc ;
- `S_IFIFO` : tube nommé (*cf.* chapitre 7).

Le paramètre `dev` correspond à l'identificateur du périphérique, c'est-à-dire le couple formé de son numéro majeur et de son numéro mineur. Cet identificateur est un entier de 16 bits dont les 8 bits de poids fort correspondent au numéro majeur et les 8 bits de poids faible au numéro mineur. Les macros suivantes permettent de manipuler cet entier :

- `major` : extrait de l'identificateur le numéro majeur du périphérique ;
- `minor` : extrait de l'identificateur le numéro mineur du périphérique ;
- `makedev` : retourne l'identificateur de périphérique correspondant à un numéro majeur et à un numéro mineur.

b) Opération de contrôle sur un périphérique

La primitive `ioctl()` permet de modifier les paramètres d'un périphérique. Son prototype est :

```
#include <sys/ioctl.h>
int ioctl (int fd, int cmd, char *arg);
```

L'opération désignée par `cmd` avec l'argument `arg` est réalisée sur le périphérique désigné par le descripteur `fd`.

Le type des opérations de contrôle étant fonction des périphériques, celles-ci sont très diverses. Nous ne les détaillerons pas dans cet ouvrage.

Le type d'opérations que permet de réaliser la primitive `ioctl()` est par exemple de manipuler et modifier au niveau d'un terminal, la structure définissant les caractères de contrôle associés à ce terminal.

c) Multiplexage des entrées-sorties

La primitive `select()` permet à un processus de se mettre en attente sur un ensemble de descripteurs, qui peuvent correspondre à un ensemble de périphériques. Le processus s'endort et est réveillé, soit dès qu'au moins un descripteur de l'ensemble est prêt, soit lorsque la temporisation éventuellement associée au `select()` est écoulée. Le prototype de cette fonction est :

```
#include <sys/types.h>
int select (int nb_desc, fd_set *ens_lire, fd_set *ens_ecrire, fd_set
*ens_exceptions, struct_timeval *delai);
```

Trois ensembles de descripteurs apparaissent dans le prototype de la fonction :

- l'ensemble `ens_lire` correspond à des descripteurs sur lesquels le processus attend des données à lire ;
- l'ensemble `ens_ecrire` correspond à des descripteurs sur lesquels le processus attend des données à écrire ;

- l'ensemble `ens_exceptions` correspond à des descripteurs sur lesquels le processus attend l'arrivée de conditions exceptionnelles. Cet ensemble est rarement utilisé sauf dans le cadre de la communication sur le réseau, mettant en œuvre l'envoi de données urgentes dans le cas du protocole TCP (*cf.* chapitre 9).

Le dernier paramètre de la fonction correspond à la définition d'un temps d'attente maximal. Ce paramètre est à NULL si aucune temporisation n'est définie. Sinon, il prend la forme d'une structure de type `timeval` :

```
struct timeval {
               time_t tv_sec;  /* secondes */
               time_t tv_usec;  /* microsecondes */
                };
```

Enfin, le premier paramètre de la fonction correspond au numéro du plus grand descripteur apparaissant dans les ensembles auquel est ajouté 1.

La primitive `select()` renvoie le nombre de descripteurs prêts et modifie les ensembles de descripteurs pour ne laisser dans ceux-ci que les descripteurs effectivement prêts. L'utilisation de la primitive `select()` suit les étapes suivantes :

- construction des ensembles de descripteurs `ens_r`. Cette construction se fait à l'aide des macros `FD_ZERO(fd_set ens_r)` qui met l'ensemble à vide, `FD_SET(int element, fd_set ens_r)` qui ajoute le descripteur `element` à l'ensemble `ens_r`, et `FD_CLR (int element, fd_set ens_r)` qui supprime le descripteur `element` de l'ensemble `ens_r` ;
- le processus se met en attente en invoquant la primitive `SELECT (ens_r, ….)` ;
- le processus sort de son attente. Si le retour de la primitive n'est pas nul, chacun des ensembles initiaux contient maintenant les descripteurs qui sont prêts à être utilisés. On teste chacun des descripteurs initiaux par le biais de la macro `FD_ISSET(int element, fd_set ens_r)` qui renvoie VRAI si le descripteur `element` est dans l'ensemble des descripteurs.

Exemple d'utilisation de la primitive select()

L'exemple suivant utilise la primitive `select()` pour endormir un processus en attente d'un caractère sur l'entrée standard. Une temporisation de 10 secondes est armée pour limiter cette attente.

```
/*********************************************************************/
/*          Exemple d'utilisation de la primitive select()           */
/*********************************************************************/
#include <sys/time.h>
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

main()
{
fd_set readset;
struct timeval duree;
```

```
char c;
int fd, retour;

fd = 0;
FD_ZERO(&readset);
FD_SET (fd, &readset);

duree.tv_sec = 10;
duree.tv_usec = 0;

retour = select (1, &readset, NULL, NULL, &duree);

if (retour == 0)
    printf ("Fin de temporisation, pas de caractères reçus !\n");
else
   if (FD_ISSET(fd, &readset))
   {
    read (fd, &c, sizeof(c));
    printf ("caractère lu: %c\n", c);
   }
}
```

4.2.3 Exemples

a) Périphérique de type bloc (disque dur)

Les opérations `write()` et `read()` associées au fichier spécial correspondant à un périphérique bloc tel qu'un disque dur sont réalisées par les fonctions `block_write()` et `block_read()` du buffer cache. Ces deux fonctions font à leur tour appel à une fonction de plus bas niveau `ll_rw_block()`, qui réalise l'entrée-sortie proprement dite. Cette fonction crée une requête d'entrées-sorties disque qui est insérée dans la liste des requêtes existantes de façon à ordonnancer les requêtes selon un algorithme de parcours SCAN.

b) Périphérique de type caractère (imprimante)

L'imprimante est un périphérique qui peut être géré soit selon un mode de scrutation, soit selon le mode par interruption.

La fonction `lp_char_polled()` réalise l'écriture d'un caractère sur l'imprimante selon un mode de scrutation. Cette fonction effectue tout d'abord une attente active pour attendre que l'imprimante soit prête, puis elle écrit un caractère dans le port du périphérique. Elle se remet alors en attente active pour attendre que l'imprimante devienne de nouveau prête.

La fonction `lp_char_interrupt()` réalise l'écriture d'un caractère sur l'imprimante selon un mode par interruption. Cette fonction effectue tout d'abord une attente active pour attendre que l'imprimante soit prête, puis elle écrit un caractère dans le port du périphérique. Lorsque le caractère a effectivement été écrit, une interruption est levée et le gestionnaire d'interruption `lp_interrupt()` est exécuté. Ce gestionnaire

détermine quelle imprimante a levé l'interruption, puis il réveille les éventuels processus en attente sur l'arrivée de l'interruption.

L'opération `write()` associée au fichier spécial caractère est réalisée par l'une des deux fonctions que nous venons d'évoquer.

Exercices

4.1 Qu'avez-vous retenu ?

Pour chacune des questions suivantes, choisissez la ou les réponses qui vous semblent exactes.

Question 1 – Quel est le rôle du pilote d'entrées-sorties ?

- ☐ a. permettre l'accès à des données distantes.
- ☐ b. gérer le périphérique auquel il est associé.
- ☐ c. ordonnancer les processus.

Question 2 – Un fichier spécial en mode caractères :

- ☐ a. est un périphérique structuré en bloc de caractère.
- ☐ b. est un périphérique qui délivre séquentiellement des caractères.
- ☐ c. est un descripteur de fichier.

Question 3 – On considère un disque de 50 pistes. Les requêtes disque à servir à l'instant *t* sont les suivantes : 20, 10, 23, 35, 42. Le bras est sur la position 25. Quelle proposition est exacte ?

- ☐ a. selon un ordre de service FCFS, le déplacement du bras est égal à 47 pistes.
- ☐ b. selon un ordre de service SSTF, le déplacement du bras est égal à 35 pistes.

4.2 Ordonnancement du disque

Soit un disque formaté en 300 pistes numérotées de 0 à 299. La liste des requêtes (numéro de piste cherchée) à servir donnée selon l'ordre d'arrivée est la suivante : 100, 53, 199, 227, 41, 15, 7, 54, 53, 0. Le bras est positionné au départ sur la piste 100. Donnez l'ordre des services pour FCFS et SSTF.

4.3 Ordonnancement du disque

On considère un disque composé de 300 pistes numérotées de 0 à 299. Le bras est couramment positionné sur la piste 50. La liste des requêtes (numéro de piste cherchée) à servir donnée selon l'ordre d'arrivée est la suivante : 62, 200, 150, 60, 12, 120, 250, 45, 10, 100.

Donnez l'ordre de service des requêtes et le déplacement de bras total en résultant dans le cas d'un service FCFS, d'un service SSTF et d'un service SCAN sens initial montant.

Solutions

4.1 Qu'avez-vous retenu ?

Question 1 – Quel est le rôle du pilote d'entrées-sorties ?

☐ b. gérer le périphérique auquel il est associé.

Question 2 – Un fichier spécial en mode caractères :

☐ b. est un périphérique qui délivre séquentiellement des caractères.

Question 3 – On considère un disque de 50 pistes. Les requêtes disque à servir à l'instant *t* sont les suivantes : 20, 10, 23, 35, 42. Le bras est sur la position 25. Quelle proposition est exacte ?

☐ a. selon un ordre de service FCFS, le déplacement du bras est égal à 47 pistes.

4.2 Ordonnancement du disque

Pour FCFS, l'ordre des services est : 100, 53, 199, 227, 41, 15, 7, 54, 53, 0.
Pour SSTF, l'ordre des services est : 100, 54, 53, 53, 41, 15, 7, 0, 199, 227.

4.3 Ordonnancement du disque

- FCFS
  - ordre de service : 62, 200, 150, 60, 12, 120, 250, 45, 10, 100
  - déplacement du bras :
    12 + 138 + 50 + 90 + 48 + 108 + 130 + 205 + 35 + 90 = 906
- SSTF
  - ordre de service : 45, 60, 62, 100, 120, 150, 200, 250, 12, 10
  - déplacement du bras :
    5 + 15 + 2 + 38 + 20 + 30 + 50 + 50 + 238 + 2 = 450
- SCAN montant
  - ordre de service : 60, 62, 100, 120, 150, 200, 250, 10, 12, 45
  - déplacement du bras :
    10 + 2 + 38 + 20 + 30 + 50 + 50 + 49 + 299 + 10 + 2 + 38 = 598.

GESTION DE LA MÉMOIRE CENTRALE

5

PLAN

OBJECTIFS

- Dans ce chapitre, nous présentons tout d'abord les principes généraux liés à la pagination et à la mémoire virtuelle.
- Puis nous décrivons les notions d'espace d'adressage pour un processus Linux et l'implémentation des concepts de pagination et de mémoire virtuelle dans le système Linux.

5.1 LES MÉCANISMES DE PAGINATION ET DE MÉMOIRE VIRTUELLE

5.1.1 Rappels sur la mémoire physique

La mémoire physique (figure 5.1) est constituée d'un ensemble de *mots mémoire* contigus désignés chacun par une *adresse physique*. Un mot mémoire est lui-même formé par un ou plusieurs octets (8 bits). On trouve ainsi couramment des mots mémoire d'une taille de 16 bits (2 octets), 32 bits (4 octets), 64 bits (8 octets) voire 128 bits (8 octets).

Physiquement, la mémoire centrale est composée par de la mémoire type DRAM (*Dynamic Random Access Memory*). C'est une mémoire volatile, accessible en lecture et écriture et permettant d'effectuer des accès directs aux données qu'elle contient.

Le processeur accède aux mots de la mémoire centrale par le biais de deux registres, le registre adresse RAD et le registre donnée RDO. Le registre RAD contient l'adresse du mot à lire ou l'adresse du mot où écrire tandis que le registre RDO contient soit la donnée à écrire, soit la donnée lue.

Les lignes d'adresses, de données et de commandes du bus processeur-mémoire font la liaison entre les registres processeur et la mémoire.

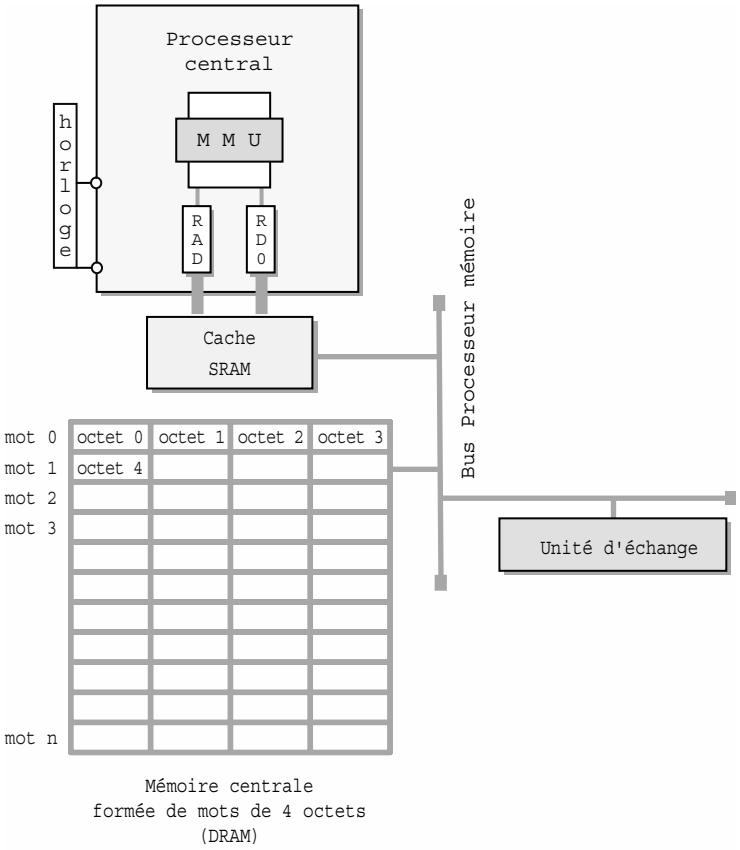

Figure 5.1 – Structure de la mémoire physique

La vitesse de délivrance d'une information par la mémoire centrale étant grande en regard de la fréquence du processeur, un niveau de mémoire supplémentaire appelé mémoire cache, est inséré au niveau du processeur. Cette mémoire vive, composée physiquement de mémoire de type SRAM (*Static Random Access Memory*) est beaucoup plus rapide que la mémoire de type DRAM. Elle est cependant d'une technologie beaucoup plus coûteuse, aussi la mémoire cache est-elle d'une capacité beaucoup plus petite que la capacité de la mémoire centrale. Cette mémoire SRAM est utilisée comme cache de la mémoire centrale et contient les mots mémoire les plus récemment accédés par le processeur.

Aussi, lorsque le processeur demande à lire un mot depuis la mémoire centrale, il cherche tout d'abord ce mot dans le cache. Si il est présent, il y a succès et aucun accès à la mémoire centrale. Sinon, il y a défaut, et le mot ainsi que ces voisins sont chargés dans le cache.

Cette méthode de chargement s'appuie sur les *principes de localité* :

- la *localité temporelle* indique que si un mot est accédé à un instant t, la probabilité qu'il soit de nouveau accédé aux instants $t + 1$, $t + 2$, etc., est grande. D'où le chargement du mot dans le cache ;
- la *localité spatiale* indique que si un mot est accédé à un instant t, la probabilité que les mots d'adresse voisine soient à leur tour accédés aux instants $t + 1$, $t + 2$, etc., est grande. D'où le chargement des mots voisins dans le cache.

Lorsque le processeur souhaite réaliser l'écriture d'un mot en mémoire centrale, l'écriture est réalisée dans le cache si le mot y est trouvé. La mise à jour du même mot en mémoire centrale s'effectue selon l'une des deux politiques suivantes :

- *write-through*, le mot est immédiatement écrit en mémoire centrale ;
- *write-back*, le mot est écrit en mémoire centrale lorsque l'entrée du cache contenant le mot modifié est réquisitionnée pour y placer un nouveau mot.

5.1.2 Espace d'adressage d'un processus

a) Notion d'espace d'adressage

L'*espace d'adressage* d'un processus est constitué de l'ensemble des adresses auxquelles le processus a accès au cours de son exécution. Cet espace d'adressage est au moins constitué de trois parties qui sont le code et les données du processus ainsi que sa pile d'exécution.

L'espace d'adressage d'un processus constitue une notion totalement indépendante de la mémoire physique sous-jacente. Cet espace d'adressage est représenté par l'ensemble des adresses qui le constitue, dont la forme dépend de la structuration de cet espace. Cet ensemble d'adresses est appelé *espace d'adresses logiques ou virtuelles*.

Par exemple, si l'espace d'adressage du processus est linéaire et formé d'un seul ensemble d'adresses contiguës, alors l'adresse logique d'un mot dans cet espace est simplement constitué par le déplacement de ce mot à l'intérieur de l'espace.

Si l'espace est découpé en morceaux indépendants de taille fixe ou variable, numérotés consécutivement, alors l'adresse logique d'un mot dans cet espace est un couple constitué d'une part du numéro du morceau dans lequel se trouve le mot, d'autre part du déplacement du mot à l'intérieur de ce morceau (figure 5.2).

b) Adresse logique et adresse physique

Pour pouvoir exécuter un programme, celui-ci doit être chargé dans la mémoire physique, ce qui implique de mapper l'espace d'adressage du processus sur l'espace physique de la mémoire centrale. L'ensemble des adresses physiques réellement occupées par le programme exécutable suite à son chargement en mémoire centrale est appelé *espace d'adresses physiques*.

Pour que le processeur puisse accéder à la mémoire centrale, il est nécessaire de *convertir* les adresses logiques générées par le processeur en adresses physiques, placées sur le bus et présentées au dispositif de sélection de la mémoire physique.

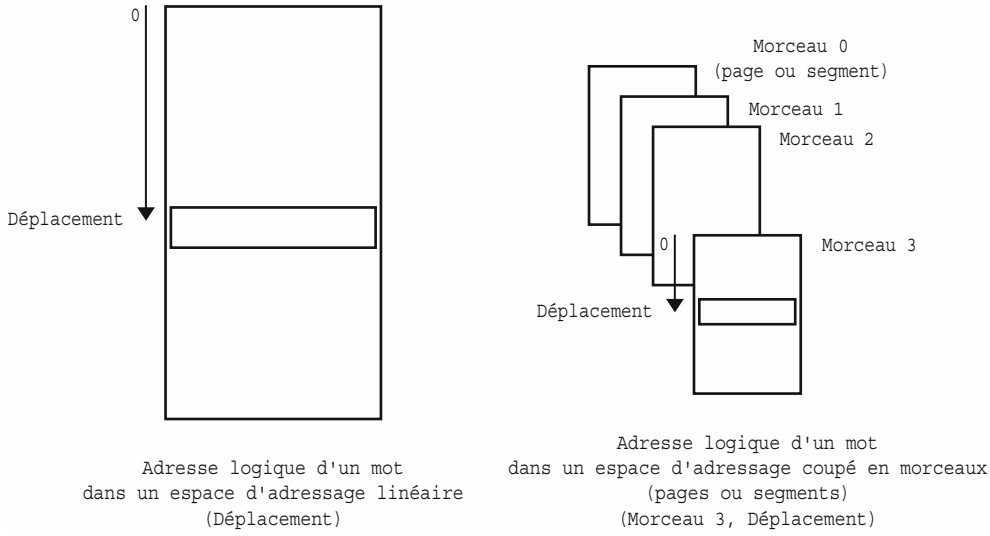

Figure 5.2 **Espace d'adressage d'un processus**

Cette conversion est réalisée par un dispositif matériel, la MMU (*Memory Management Unit*).

Ainsi sur l'exemple de la figure 5.3, la MMU contient un registre appelé registre de translation. Ce registre de translation est toujours chargé avec l'adresse d'implantation du programme exécutable correspondant au processus couramment actif (dans l'exemple 2 048). Lorsque le processeur veut accéder à un mot de l'espace d'adressage du processus désigné par l'adresse logique adr_log (1 024), alors le dispositif de MMU ajoute à l'adresse logique adr_log (1 024) la valeur présente dans le registre de translation (2 048), de façon à générer l'adresse physique correspondante (3 072).

5.1.3 Pagination de la mémoire centrale

a) Principe de la pagination

Dans le mécanisme de pagination, l'espace d'adressage du programme est découpé en morceaux linéaires de même taille appelés *pages*. L'espace de la mémoire physique est lui-même découpé en morceaux linéaires de même taille appelés *cases* ou *cadres de page*. La taille d'une case est égale à la taille d'une page. Cette taille est définie par le matériel, comme étant une puissance de 2, variant selon les systèmes d'exploitation entre 512 octets et 8 192 octets.

Dans ce contexte, charger un programme en mémoire centrale consiste à placer les pages dans n'importe quelle case disponible. Pour connaître à tout moment quelles sont les cases libres en mémoire centrale à un instant *t*, le système maintient une table appelée *table des cadres de pages* ou *table des cases* qui indique pour chaque case de la mémoire physique, si la case est libre ou occupée, et si elle est occupée, quelle page et quel processus la possèdent.

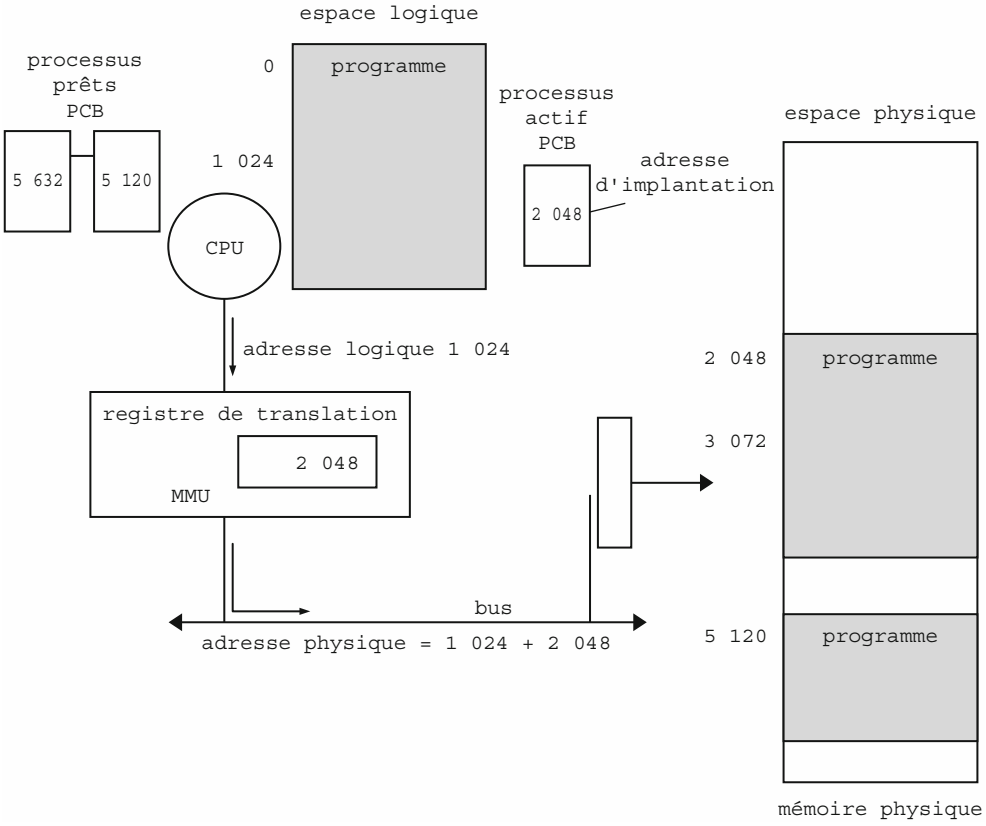

Figure 5.3 – Espace d'adresses logiques et espace d'adresses physiques

La figure 5.4 donne un exemple d'application de ce mécanisme d'allocation pour deux processus P1 et P2 dont les espaces d'adressage sont respectivement égaux à 16 Ko et 7 Ko. Les pages et les cases ont une taille de 4 Ko.

b) Adresse logique et table des pages

Conversion adresse logique – adresse physique

L'espace d'adressage du processus étant découpé en pages, les adresses générées dans cet espace d'adressage sont des *adresses paginées*, c'est-à-dire qu'un octet est repéré par son emplacement relativement au début de la page à laquelle il appartient. L'adresse d'un octet est donc formée par le couple <numéro de page p à laquelle appartient l'octet, déplacement d relativement au début de la page p>. Pour une adresse logique de m bits, en considérant des pages de 2^n octets, les $m - n$ premiers bits correspondent au numéro de page p et les n bits restants au déplacement d dans la page.

Les octets dans la mémoire physique ne peuvent être adressés au niveau matériel que par leur adresse physique. Pour toute opération concernant la mémoire, il faut

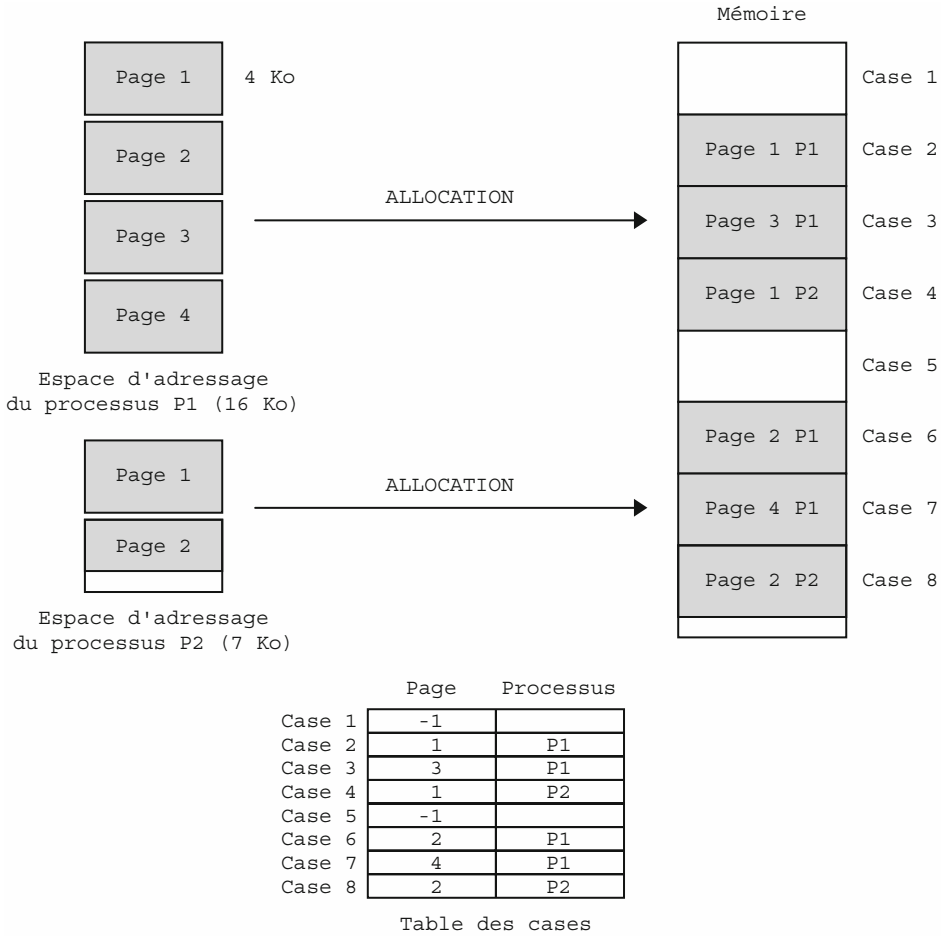

Figure 5.4 – Principe de la pagination

donc convertir l'adresse paginée générée au niveau du processeur en une adresse physique équivalente. L'adresse physique d'un octet s'obtient à partir de son adresse logique en remplaçant le numéro de page *p* par l'adresse physique d'implantation de la case contenant la page *p* et en ajoutant à cette adresse physique d'implantation, le déplacement *d* de l'octet dans la page. C'est la MMU qui effectue cette conversion.

Pour toute page, il faut donc connaître dans quelle case de la mémoire centrale celle-ci a été placée. Cette correspondance s'effectue grâce à une structure particulière appelée la *table de pages*.

La table des pages

Dans une première approche, la table des pages est une table contenant autant d'entrées que de pages dans l'espace d'adressage d'un processus. Chaque processus a sa propre table des pages. Chaque entrée de la table est un couple <numéro de page,

numéro de case physique dans laquelle la page est chargée> ou <numéro de page, adresse d'implantation de case physique dans laquelle la page est chargée>.

Dans l'exemple de la figure 5.5, le processus P1 a 4 pages dans son espace d'adressage, donc la table des pages a 4 entrées. Chaque entrée établit l'équivalence numéro de page, numéro de case dans laquelle la page est chargée relativement au schéma de la mémoire centrale. Le processus P2 quant à lui a un espace d'adressage composé de 2 pages, donc sa table des pages a 2 entrées. Par ailleurs, l'espace d'adressage du processus P2 étant de 7 Ko, la page 2 du processus a une taille égale à 3 Ko. Il s'ensuit un phénomène de *fragmentation interne* en mémoire physique au niveau de la case 8 dans laquelle la page 2 est chargée, c'est-à-dire une perte de place au sein de la case qui n'est pas totalement occupée.

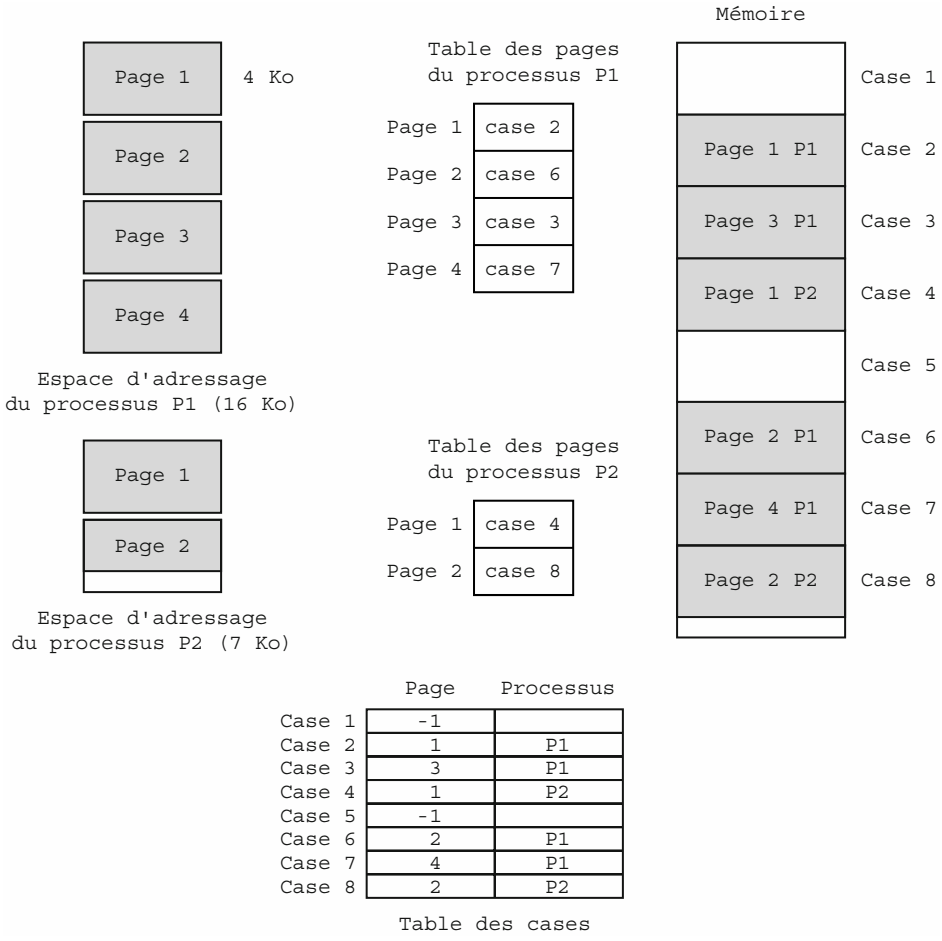

Figure 5.5 – Table des pages

Puisque chaque processus dispose de sa propre table des pages, chaque opération de commutation de contexte se traduit également par un changement de table des pages, de manière à ce que la table active corresponde à celle du processus élu.

La table des pages est généralement une structure logicielle placée en mémoire centrale. L'adresse en mémoire centrale de la table des pages du processus actif est placée dans un registre de la MMU, le PTBR (*page-table base register*). Chaque processus sauvegarde dans son PCB la valeur de PTBR correspondant à sa table.

Accéder à un emplacement mémoire à partir d'une adresse paginée <*p*, *d*> nécessite deux accès à la mémoire (figure 5.6) :

- un premier accès permet de lire l'entrée de la table des pages correspondant à la page *p* cherchée et délivre une adresse physique *c* de case dans la mémoire centrale[1] ;
- un second accès est nécessaire à la lecture ou l'écriture de l'octet recherché à l'adresse *c* + *d*.

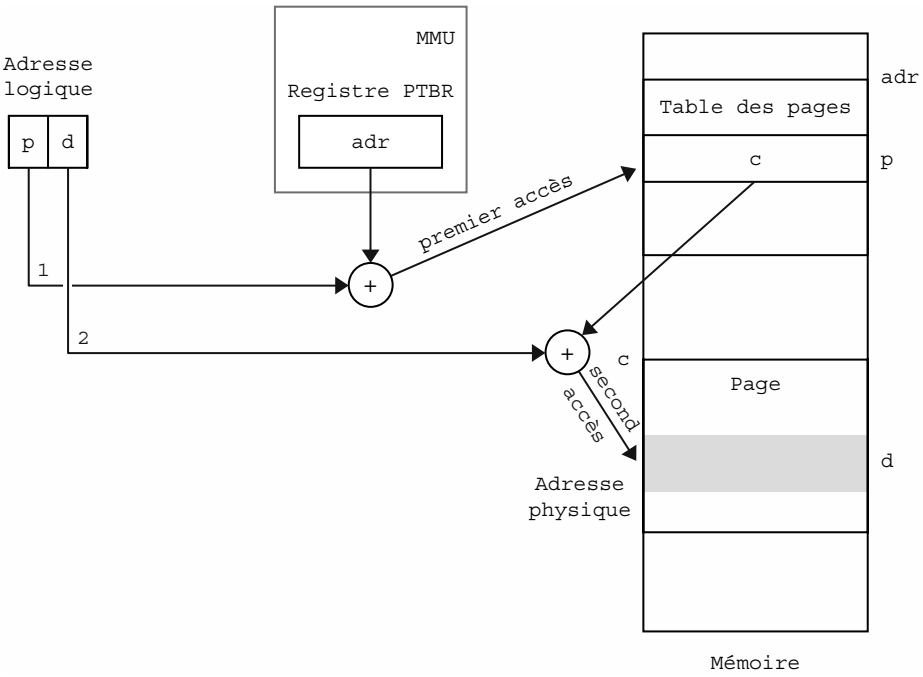

Figure 5.6 – Traduction d'une adresse paginée en adresse physique

1. Si l'entrée de la table des pages contient le numéro *nc* de la case dans laquelle la page est chargée, l'adresse d'implantation *c* est obtenue en multipliant *nc* par la taille en octet d'une case.

Pour accélérer les accès à la mémoire centrale et compenser le coût lié à la pagination, un cache associatif (*TLB look-aside buffers*) est placé en amont de la mémoire centrale. Ce cache associatif contient les couples <numéro de page *p*, adresse d'implantation de la case contenant *p*> les plus récemment accédés. Lorsque la MMU doit effectuer une conversion d'adresse paginée, elle cherche tout d'abord dans le cache si la correspondance <numéro de page *p*, adresse d'implantation de la case contenant *p*> n'est pas dans le cache. Si non, elle accède à la table des pages en mémoire centrale et place le nouveau couple référencé dans le cache. Si oui, elle effectue directement la conversion : un seul accès mémoire est alors nécessaire pour accéder à l'octet recherché (figure 5.7).

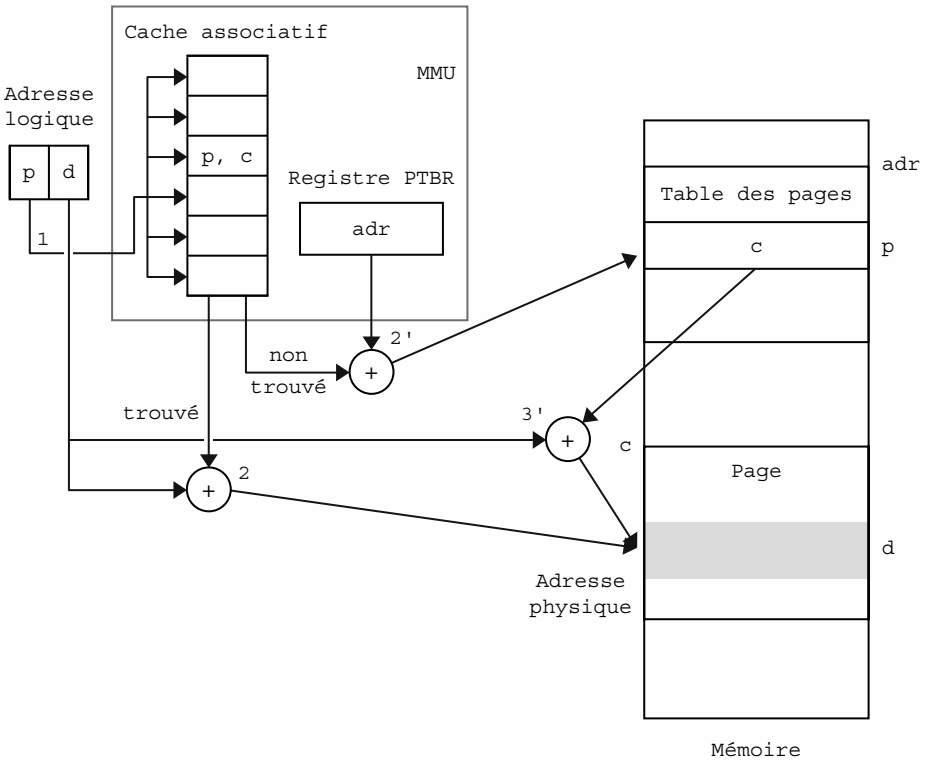

Figure 5.7 – Traduction d'une adresse paginée en adresse physique avec ajout d'un cache associatif

c) Protection de l'espace d'adressage des processus

Des bits de protection sont associés à chaque page de l'espace d'adressage du processus et permettent ainsi de définir le type d'accès autorisés à la page. Ces bits de protection sont mémorisés pour chaque page, dans la table des pages du processus. Classiquement, 3 bits sont utilisés pour définir respectivement l'autorisation d'accès en lecture (`r`), écriture (`w`) et exécution (`x`).

Lors d'un accès à une page, la cohérence du type d'accès avec les droits associés à la page est vérifiée et une trappe est levée par le système d'exploitation si le type d'accès réalisé est interdit.

Par ailleurs, chaque processus ne peut avoir accès qu'à sa propre table des pages et donc à ses propres pages. De ce fait, chaque espace d'adressage est protégé vis-à-vis des accès des autres processus. Malgré tout, il peut être souhaitable que des pages soient accessibles par plusieurs processus différents. C'est par exemple souhaitable pour éviter la duplication en mémoire centrale du code des bibliothèques ou la duplication de code exécutable réentrant comme le code de l'éditeur de texte utilisé couramment par les utilisateurs de la machine. Pour que deux processus puissent partager un ensemble de pages, il faut que chacun des deux processus référence cet ensemble de pages dans sa table des pages respective. Ainsi, sur la figure 5.8, les processus P1 et P2 référencent tous les deux la même page 1, située dans la case 2 de la mémoire centrale.

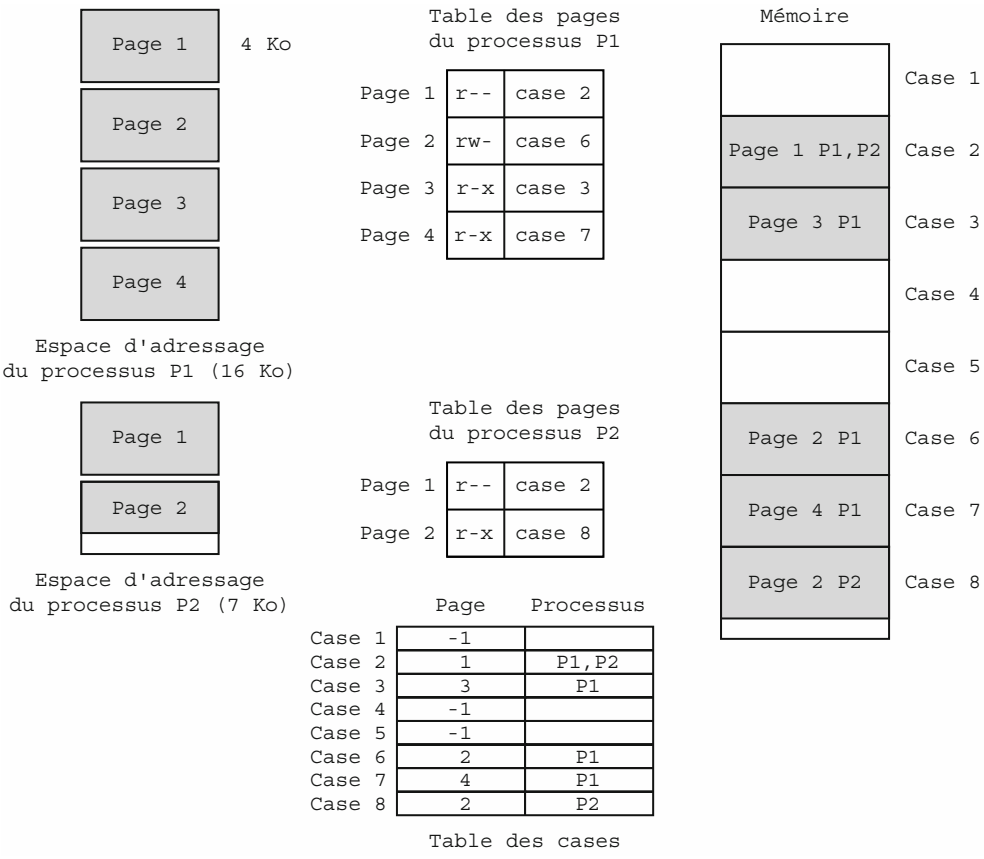

Figure 5.8 – Entrées de la table des pages avec les bits de protection et partage des pages entre processus

d) Pagination multiniveaux

L'espace d'adresses logiques supporté par les systèmes d'exploitation actuels est très grand, de l'ordre de 2^{32} à 2^{64} octets. Dans de telles conditions, la table des pages d'un processus peut devenir également de très grande taille et comporter jusqu'à un million d'entrées. Il n'est dès lors plus envisageable de charger de manière contiguë la table des pages d'un processus. Une solution consiste alors à paginer la table des pages elle-même.

Dans ce contexte, l'adresse paginée devient un triplet <*hp*, *p*, *d*>, où *hp* désigne l'entrée d'une hypertable des pages, dont chaque entrée correspond à une page contenant une partie de la table des pages du processus. *p* désigne une entrée de cette partie de la table des pages qui permet d'accéder à la page *p* de l'espace d'adressage du processus et *d* un déplacement dans la page *p*.

L'hypertable des pages est elle-même placée en mémoire centrale et son adresse d'implantation en mémoire centrale est repérée par un registre matériel de la MMU. La conversion de l'adresse paginée <*hp*, *p*, *d*> en adresse physique et l'accès à l'octet mémoire désigné comporte maintenant une étape supplémentaire (figure 5.9) :

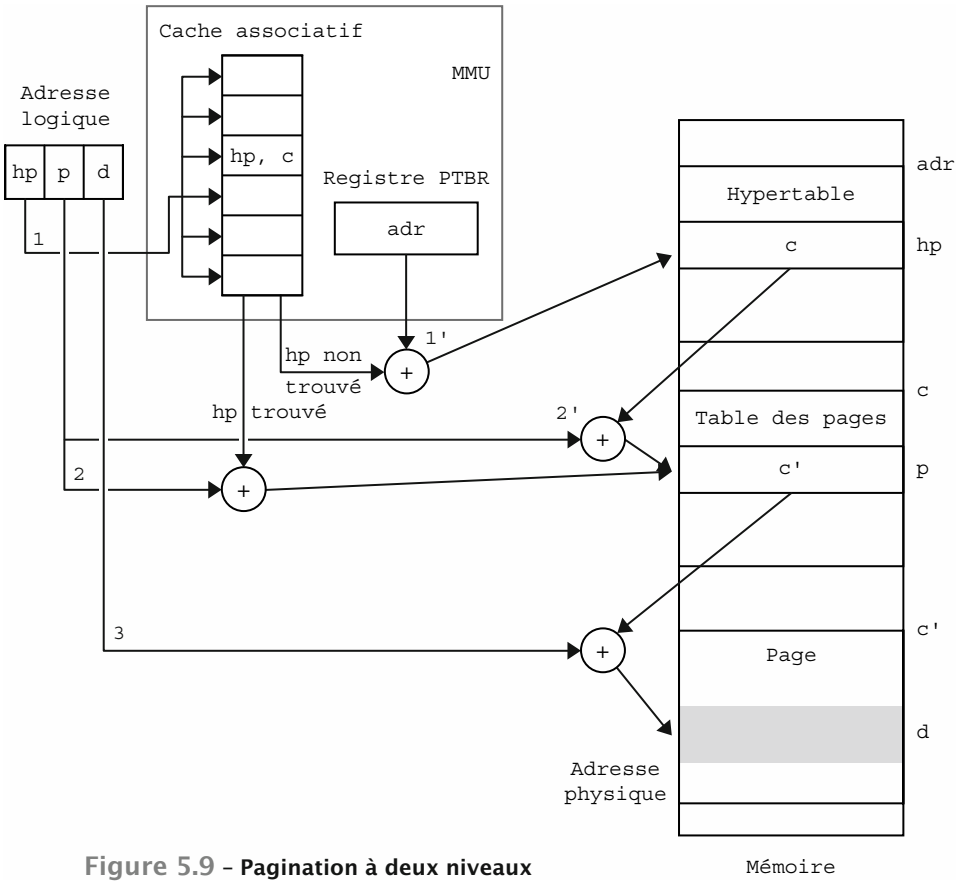

Figure 5.9 – Pagination à deux niveaux

- un premier accès permet de lire l'entrée de l'hypertable des pages correspondant à la page *hp* cherchée et délivre une adresse physique *c* de case dans la mémoire centrale, qui contient une partie de la table des pages du processus ;
- un second accès à la case *c* permet de lire l'entrée *p* de la table des pages et de récupérer l'adresse physique de la case *c'* contenant la page *p* ;
- un troisième accès est nécessaire à la lecture ou l'écriture de l'octet recherché à l'adresse *c'* + *d*.

Pour accélérer la conversion de l'adresse logique vers l'adresse physique, un cache associatif contient les couples <*hp*, *p*> les plus récemment accédés.

Ce schéma se généralise à un nombre de niveaux supérieur à 2.

5.1.4 Principe de la mémoire virtuelle

a) Principe de la mémoire virtuelle

La multiprogrammation implique de charger plusieurs programmes en mémoire centrale de manière à obtenir un bon taux d'utilisation du processeur. Supposons comme sur la figure 5.10 que l'exécution des processus P1, P2, P3 soit nécessaire pour obtenir ce taux d'utilisation du processeur satisfaisant. On peut remarquer qu'une fois les pages des processus P1 et P2 chargées dans la mémoire, il ne reste pas assez de cases libres pour charger la totalité des pages du programme 3. Ce dernier ne peut donc pas être chargé.

Lorsque l'on regarde l'exécution d'un processus, on s'aperçoit qu'à un instant donné le processus n'accède qu'à une partie de son espace d'adressage, par exemple la page de code couramment exécutée par le processeur et la page de données correspondante. Les autres pages de l'espace d'adressage ne sont pas accédées et sont donc inutiles en mémoire centrale. Une solution pour pouvoir charger plus de processus dans la mémoire centrale est donc de ne charger pour chaque processus que les pages couramment utilisées. Ainsi, sur la figure 5.11, seules les pages 1 et 3 du processus P1 sont chargées ainsi que la page 1 du processus P2 et les pages 1 et 2 du processus P3.

Par ailleurs, on remarquera sur cette même figure, que l'espace d'adressage du processus P4 est plus grand que la mémoire physique disponible pour les programmes utilisateurs. Le *principe de la mémoire virtuelle*, qui consiste donc à ne charger à un instant donné en mémoire centrale que la partie couramment utile de l'espace d'adressage des processus, permet de résoudre ce problème et autorise donc la constitution de programmes dont la taille n'est plus limitée par celle de la mémoire physique.

Puisque les pages d'un espace d'adressage de processus ne sont pas toutes chargées en mémoire centrale, il faut que le processeur puisse détecter leur éventuelle absence au moment où s'effectue la conversion de l'adresse logique vers l'adresse physique. Chaque entrée de la table des pages comporte alors un champ supplémentaire, le bit validation *V*, qui est à 1 si la page est effectivement présente en mémoire centrale, 0 sinon.

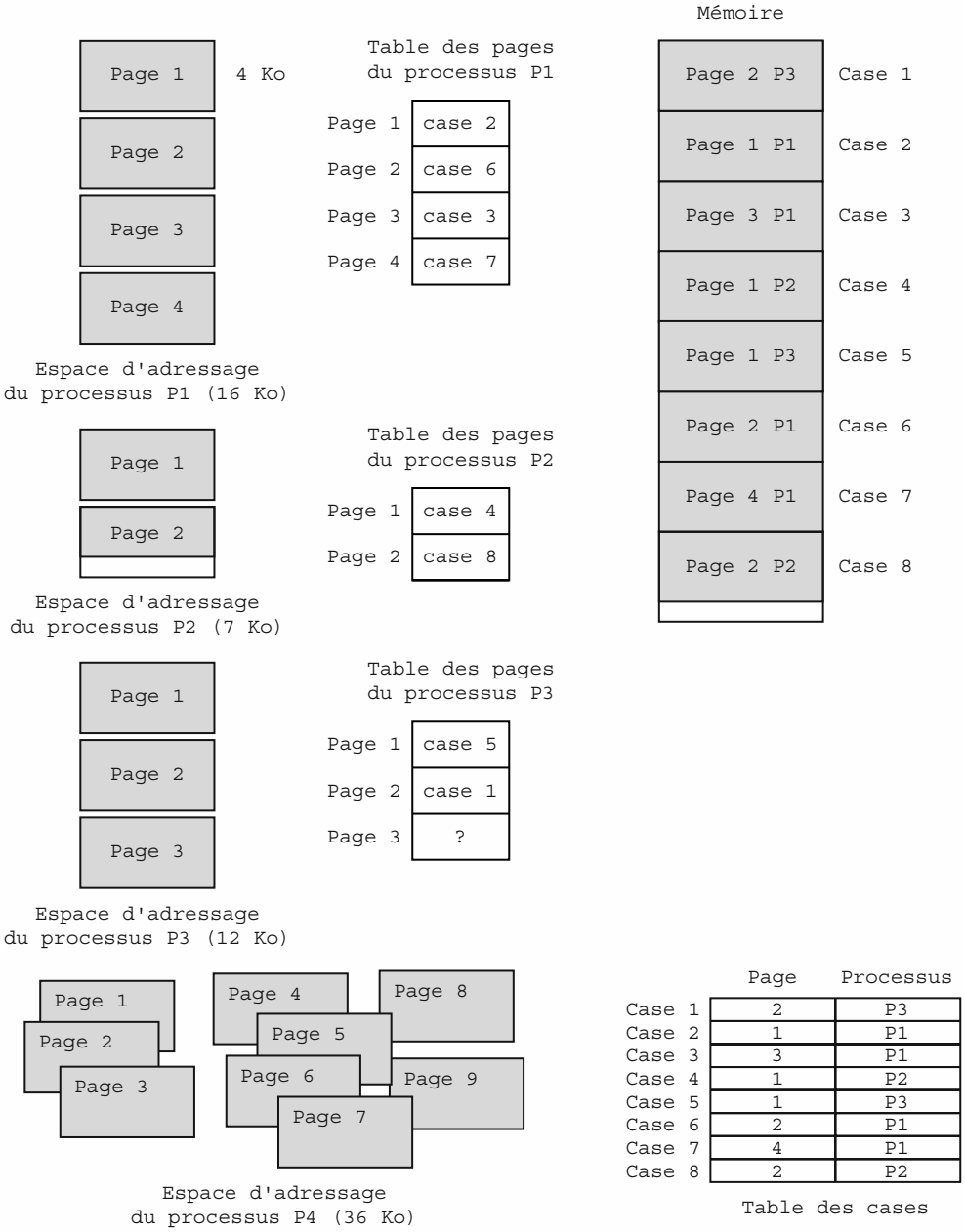

Figure 5.10 – Insuffisance de la mémoire physique

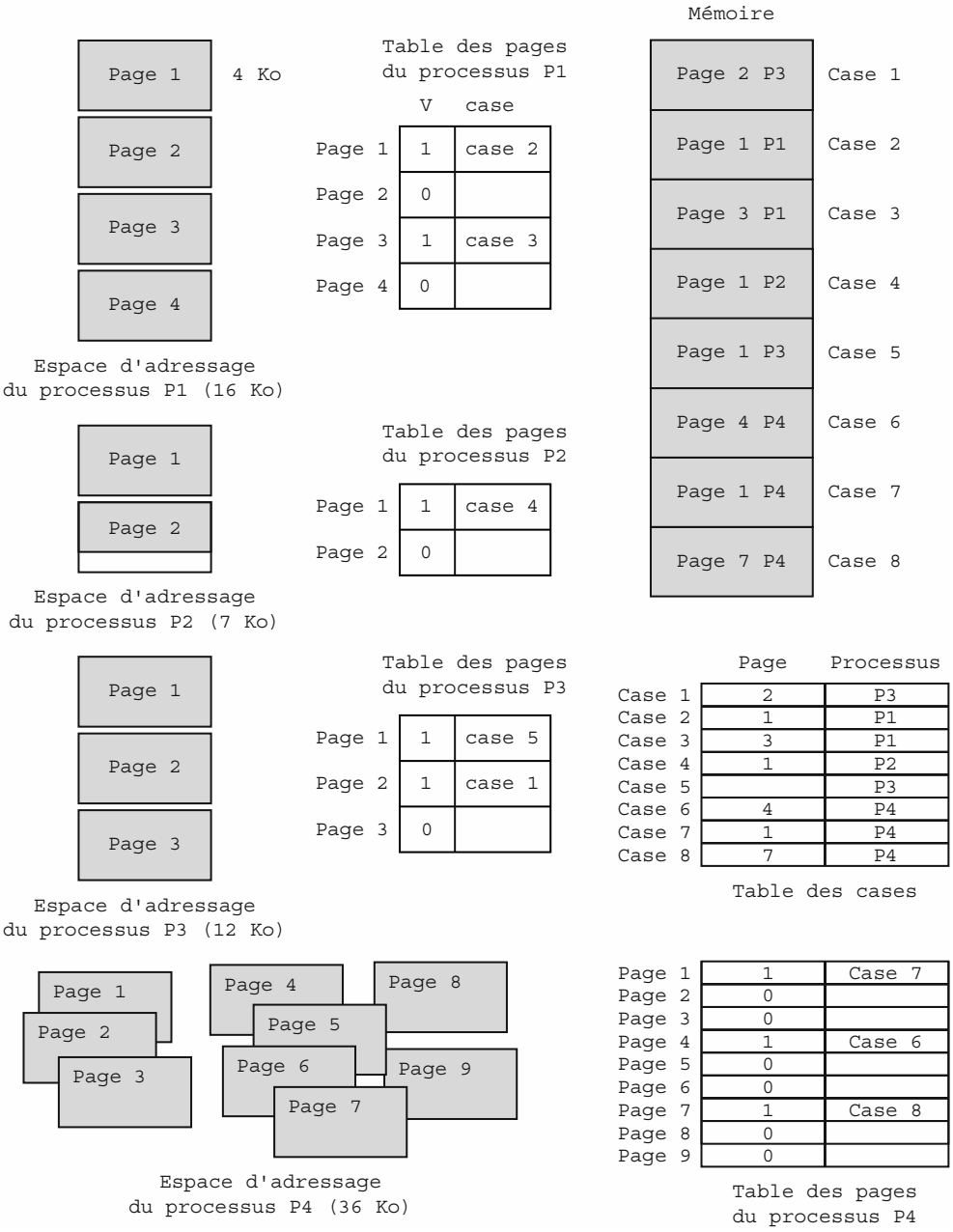

Figure 5.11 – Mémoire virtuelle

La figure 5.11 montre les valeurs des bits de validation pour les tables des pages des 4 processus P1, P2, P3 et P4, en tenant compte des chargements de leurs pages en mémoire centrale. Ainsi pour le processus P1, la page 1 est chargée dans la case 2, le bit de validation est à 1. La page 3 est chargée dans la case 3, le bit de validation est à 1. Par contre les pages 2 et 4 ne sont pas présentes en mémoire centrale et donc le bit de validation est à 0 : dans ce cas, le champ numéro de case n'a pas de signification.

Le principe de la mémoire virtuelle est couramment implémenté avec la pagination à la demande, c'est-à-dire que les pages des processus ne sont chargées en mémoire centrale que lorsque le processeur demande à y accéder.

b) Le défaut de page

Que se passe-t-il à présent lorsqu'un processus tente d'accéder à une page de son espace d'adressage qui n'est pas en mémoire centrale ? La MMU accède à la table des pages pour effectuer la conversion de l'adresse logique vers l'adresse physique et teste la valeur du bit de validation. Si la valeur du bit *V* est à 0, cela signifie que la page n'est pas chargée dans une case et donc la conversion ne peut pas être réalisée. Une trappe appelée *défaut de page* est levée qui suspend l'exécution du processus puis initialise une opération d'entrées-sorties afin de charger la page manquante en mémoire centrale dans une case libre. Les pages constituant l'espace d'adressage du processus sont stockées dans une zone particulière du disque communément appelée *zone de swap*. L'adresse de chaque page sur le disque est mémorisée dans l'entrée correspondante de la table des pages. Ainsi, sur un processeur de type Pentium, l'exception 14 correspond au défaut de pages..

Avec ce principe de pagination à la demande, le mécanisme de conversion d'une adresse logique <*p*, *d*> vers l'adresse physique correspondante devient (figure 5.12) :

1. accès à l'entrée *p* de la table des pages du processus actif et test de la valeur du bit de validation *V* ;
2. si la valeur du bit *V* est à 0, alors il y a défaut de page. Une opération d'entrées-sorties est lancée pour charger la page dans la mémoire centrale (l'adresse de la page sur le disque est stockée dans la table des pages) ;
3. la page est placée dans une case libre, trouvée par l'intermédiaire de la table des cases ;
4. la table des pages du processus est mise à jour c'est-à-dire que le bit de validation *V* passe à 1 et le champ numéro de case est renseigné avec l'adresse d'implantation de la case abritant maintenant la page *p* ;
5. la conversion de l'adresse logique vers l'adresse physique est reprise selon le principe vu au paragraphe 5.1.3.

c) Le remplacement de pages

Lors d'un défaut de page, la page manquante est chargée dans une case libre. Cependant, la totalité des cases de la mémoire centrale peut être occupée. Il faut donc libérer une case de la mémoire physique pour y placer la nouvelle page.

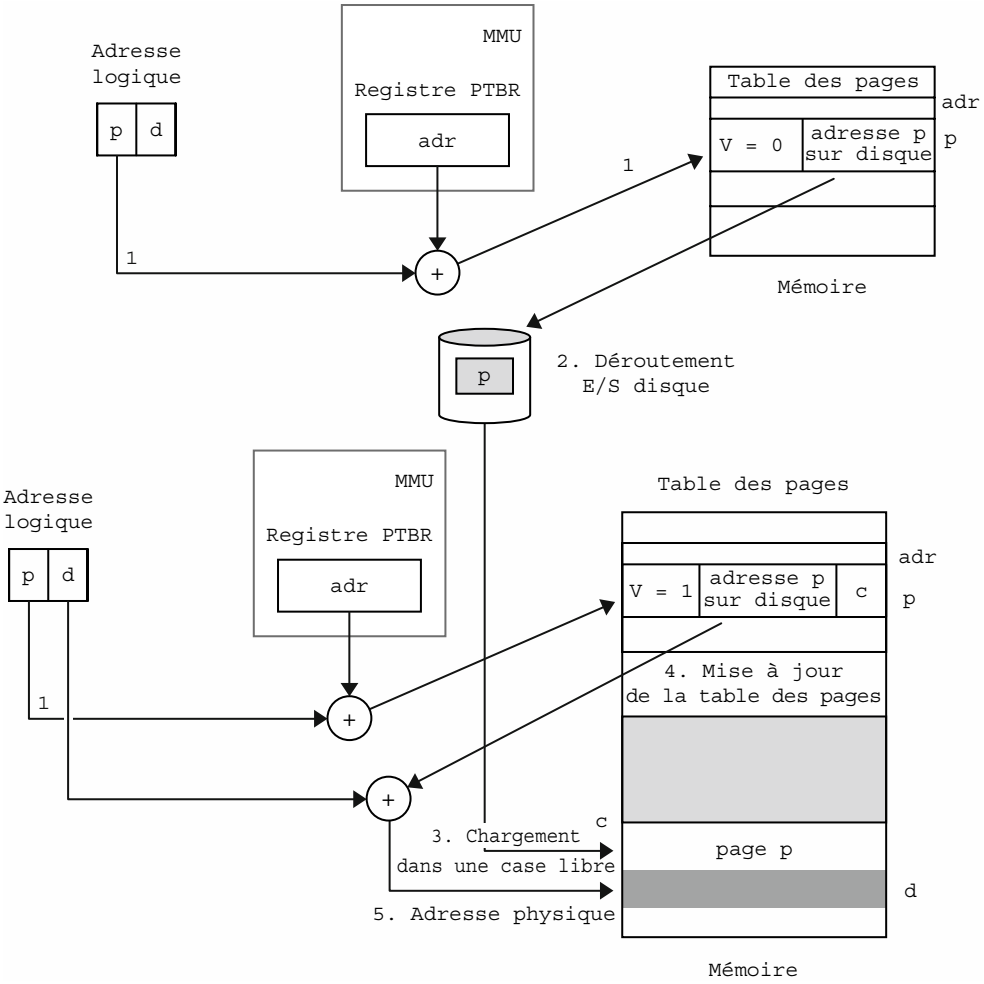

Figure 5.12 – Défaut de page

Lors de la libération d'une case, la page victime contenue dans cette case doit être sauvegardée sur le support de masse si elle a été modifiée lors de son séjour en mémoire centrale. Un bit *M* de modification mis à 1 si la page est modifiée est ajouté à chaque entrée de la table des pages afin d'être à même de savoir si la page doit être réécrite sur le disque avant d'être écrasée par la nouvelle page (figure 5.13).

Le choix de la page victime lors du traitement d'un défaut de page pour un processus peut se faire soit localement à ce processus, soit globalement sur l'ensemble des processus. Dans le premier cas, la page victime doit forcément appartenir à l'espace d'adressage du processus en défaut de page. Dans le second cas, la page victime peut appartenir à l'espace d'adressage de n'importe lequel des processus présents en mémoire centrale. Ce dernier cas est le plus souvent mis en œuvre car il donne de meilleurs résultats quant à l'utilisation de la mémoire centrale.

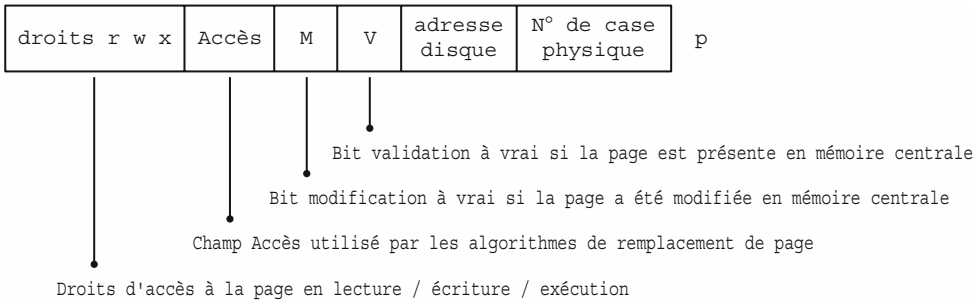

Figure 5.13 – Format d'une entrée de la table des pages

Le mécanisme de conversion d'une adresse logique <*p*, *d*> vers l'adresse physique correspondante devient maintenant :

1. accès à l'entrée *p* de la table des pages du processus actif et test de la valeur du bit de validation *V* ;
2. si la valeur du bit *V* est à 0, alors il y a défaut de page ;
3. une case libre est recherchée. Si il n'y a pas de cases libres, une page victime est choisie. Si la valeur du bit *M* pour cette page est à 1, alors une opération d'entrées-sorties est lancée pour sauvegarder la page victime ;
4. une opération d'entrées-sorties est lancée pour charger la page dans la mémoire centrale dans la case libre ou libérée ;
5. la table des pages du processus est mise à jour c'est-à-dire que le bit de validation *V* passe à 1 et le champ numéro de case est renseigné avec l'adresse d'implantation de la case abritant maintenant la page *p* ;
6. la conversion de l'adresse logique vers l'adresse physique est reprise selon le principe vu au paragraphe 5.1.3.

Le choix optimal pour la page victime consiste à libérer une case en déchargeant de la mémoire centrale une page qui ne servira plus du tout, de façon à ne pas provoquer de défaut de page ultérieur sur cette même page. Évidemment, il n'est pas possible de savoir si telle ou telle page d'un processus est encore utile ou pas.

Plusieurs algorithmes, appelés *algorithmes de remplacement de pages*, ont été définis qui tendent plus ou moins vers l'objectif optimal. Ce sont les stratégies :

- FIFO (*First In, First Out*) ;
- LRU (*Least Recently Used*) ;
- Algorithme de la seconde chance.

L'évaluation de ces différentes stratégies s'effectue en comptant sur une même suite de références à des pages, le nombre de défaut de pages provoqués. Une telle suite de références à un même ensemble de pages est appelée *chaîne de références*. Ainsi dans les paragraphes suivants, nous donnons un exemple de chaque stratégie sur la chaîne de références 8, 1, 2, 3, 1, 4, 1, 5, 3, 4, 1, 4, 3 où chaque chiffre *i* correspond à un accès à la page *i.* Dans chaque exemple, nous supposons une mémoire

centrale composée de trois cases initialement vides. La lettre *D* signale l'occurrence d'un défaut de page.

Remplacement FIFO

Avec cet algorithme, c'est la page la plus anciennement chargée qui est remplacée. Une implémentation de cet algorithme peut être réalisée en conservant dans la table des pages, la date de chargement de chaque page. La page choisie est alors celle pour laquelle la date de chargement est la plus ancienne.

La mise en œuvre de cet algorithme est simple, mais ses performances ne sont pas toujours bonnes. En effet, l'âge d'une page n'est pas le reflet de son utilité. La figure 5.14 donne un exemple de mise en œuvre de cet algorithme.

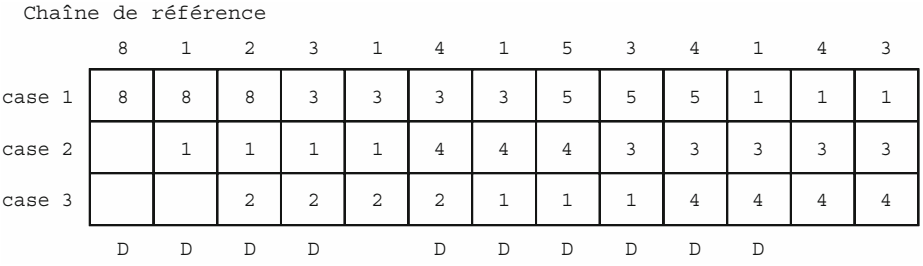

Figure 5.14 – Remplacement FIFO

Remplacement LRU

Avec cet algorithme, c'est la page la moins récemment utilisée qui est remplacée. Cette stratégie utilise le principe de la localité temporelle selon lequel les pages récemment utilisées sont celles qui seront référencées dans le futur. La page la moins récemment utilisée est donc jugée comme étant celle devenue la plus inutile.

Une implémentation de cet algorithme peut être réalisée en conservant dans la table des pages, la date de dernier accès de chaque page. La page choisie est alors celle pour laquelle la date d'accès est la plus ancienne. La figure 5.15 donne un exemple de mise en œuvre de cet algorithme.

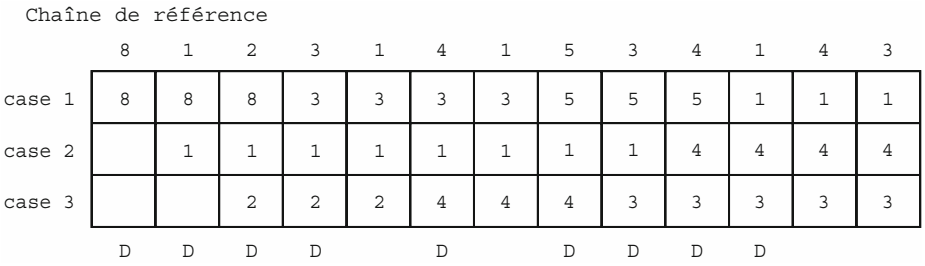

Figure 5.15 – Remplacement LRU

L'algorithme LRU est l'un des algorithmes les plus utilisés car il est considéré comme très bon. Cependant sa mise en œuvre est coûteuse et nécessite le recours à des compteurs matériels spécifiques pour la délivrance des dates d'accès aux pages.

Algorithme de la seconde chance

Un bit de référence est associé à chaque page dans la table des pages des processus. Ce bit de référence est mis à 1 à chaque référence à la page. Le principe de l'algorithme de la seconde chance est le suivant :

- la page la plus anciennement chargée est sélectionnée comme étant a priori la page victime ;
- si la valeur du bit de référence est à 0, alors la page est effectivement remplacée ;
- si la valeur du bit de référence est au contraire à 1, alors une seconde chance est donnée à la page. Son bit de référence est remis à 0, et la page la plus anciennement chargée suivante est sélectionnée.

Cet algorithme constitue une bonne approximation de l'algorithme LRU. Il est beaucoup moins coûteux à implémenter car de nombreux matériels offrent directement un bit de référence associé à chaque case de la mémoire centrale.

5.2 La gestion de la mémoire centrale sous Linux

Le système Linux étant destiné à s'exécuter sur de nombreuses plates formes matérielles, le modèle de gestion mémoire mis en œuvre a été défini indépendamment des architectures matérielles visées. L'implémentation du modèle s'effectue ensuite au cas par cas, en fonction du support matériel offert par la machine. Nous décrivons dans les paragraphes qui suivent le modèle tel qu'il a été défini indépendamment des architectures.

5.2.1 Espace d'adressage d'un processus Linux

a) Notion de régions

L'espace d'adressage d'un processus Linux est composé de cinq parties principales : le code, les données initialisées, les données non initialisées, le tas[1] et la pile. Il comporte en plus une zone contenant les arguments du programme exécutable et les variables d'environnements de celui-ci. Avec un processeur de type x86, 3 Go sont affectés à l'espace d'adressage d'un processus.

Cet espace d'adressage est découpé en régions. Une *région* est une zone contiguë de l'espace d'adressage qui est traitée comme un objet pouvant être partagé et protégé. L'espace d'adressage du processus va donc comporter une région contenant le code,

1. Le tas est une zone mémoire spécifique réservée pour satisfaire des demandes d'allocations dynamiques.

une région contenant les données initialisées, une autre contenant les données non initialisées et enfin une région contenant le tas et une autre pour la pile. Par ailleurs, l'espace d'adressage comporte également des régions allouées pour contenir les données des bibliothèques partagées utilisées par le processus. Enfin, tout processus peut ajouter de nouvelles régions à son espace d'adressage dans lesquelles il peut projeter le contenu d'un fichier. Cette *opération de projection* permet alors au processus d'accéder aux données du fichier par le biais d'opération de lecture et d'écriture en mémoire centrale plutôt que le biais des opérations d'entrées-sorties.

Un descripteur de l'espace d'adressage du processus est accessible depuis le bloc de contrôle du processus (champ `struct mm_struct *mm;`). Ce descripteur contient les informations suivantes :

- la liste des régions composant l'espace d'adressage ;
- le nombre de processus partageant ce descripteur, ce nombre pouvant être supérieur à 1 dans le cas de processus légers évoluant au sein du même espace d'adressage ;
- les adresses de début et de fin des sections de code, données non initialisées, données initialisées, tas et pile ;
- les adresses de début et de fin des sections contenant les arguments du programme exécutable ;
- les adresses de début et de fin des sections contenant les variables d'environnements du programme exécutable ;
- l'adresse de la table globale des pages du processus ;
- la taille en octets de l'espace d'adressage du processus.

La liste des régions est une liste chaînée simple de descripteurs de régions ordonnés par ordre croissant des adresses mémoire. Chaque descripteur (structure `struct vm_area_struct` définie dans le fichier `<linux/mm.h>`) contient les attributs associés à une région (figure 5.16). Ce sont :

- les adresses de début et de fin de la région. La taille d'une région doit être un multiple de 4 096 ;
- les propriétés associées à la région. Plusieurs types de propriétés sont définies ; les principales sont :
  - `VM_READ`, la région est accessible en lecture ;
  - `VM_WRITE`, la région est accessible en écriture ;
  - `VM_EXEC`, la région est accessible en exécution ;
  - `VM_GROWSDOWN`, `VM_GROWSUP`, la région peut être étendue soit vers le bas, soit vers le haut ;
  - `VM_LOCKED`, la région est verrouillée et ne peut pas être retirée de mémoire centrale ;
  - `VM_SHARED`, la région est partagée par plusieurs processus ;
  - `VM_SHM`, la région est une région de mémoire partagée (IPC, *cf.* chapitre 7).

Un accès non conforme à une région provoque la levée d'une trappe et l'envoi au processus fautif du signal `SIGSEGV` (*cf.* chapitre 6) qui entraîne la terminaison du processus fautif.

- l'objet associé à la région ;
- les opérations `vm_ops` qui sont associées.

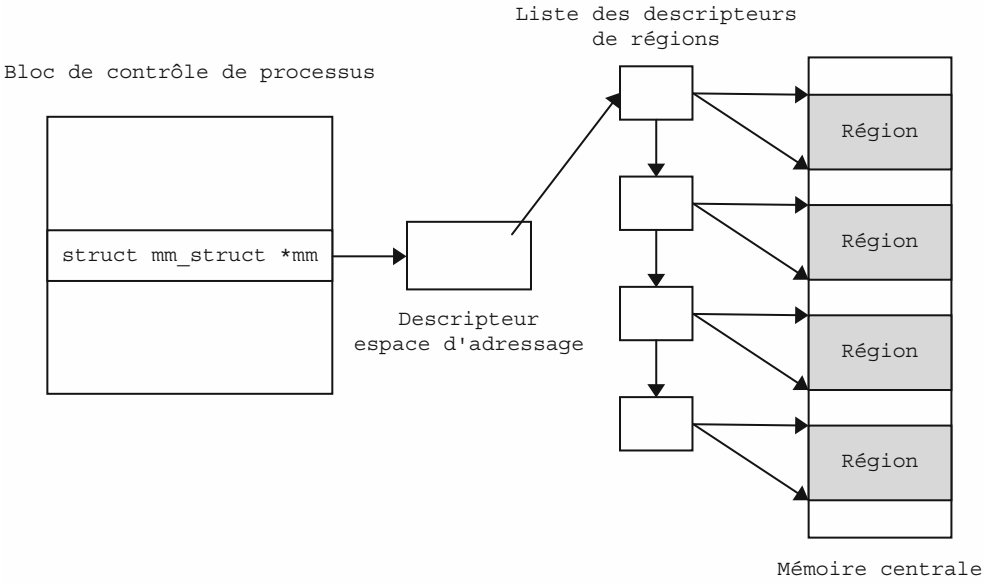

Figure 5.16 – Descripteurs de régions d'un processus

Un processus peut compter jusqu'à `MAX_MAP_COUNT` régions dans son espace d'adressage, la valeur courante pour cette constante étant égale à 65 536. Lorsqu'un processus possède un grand nombre de régions mémoire, l'utilisation de la structure de liste chaînée simple pour la recherche, l'insertion ou la destruction d'une région se révèle peu efficace. Dans ce cas, les descripteurs des régions sont enregistrés dans une structure de données de type arbre AVL (*AVL trees*)[1].

Les régions mémoire contenues dans l'espace d'adressage d'un processus peuvent être visualisées en affichant le contenu du fichier `maps`, placé dans le répertoire du processus, à l'intérieur du répertoire `/proc`. Nous donnons ici le contenu de ce fichier pour un processus de pid égal à 498 dont l'exécutable associé porte le nom `/root/exemple`. Les deux premiers champs de chaque ligne indiquent les adresses de début et de fin de chacun des régions. Le champ suivant montre les droits d'accès associés à la région. Les derniers champs donnent des informations relatives à l'objet associé à la région, c'est-à-dire le déplacement du début de la région par rapport à l'objet, le numéro de périphérique (numéros majeur et mineur) contenant l'objet et le numéro d'inode de l'objet.

1. Pour des précisions sur ces structures, se reporter à C. Carrez, *Structures de données en Java, C++ et Ada 95* (Masson, 1997) ou M.C. Gaudel, M. Soria, C. Froidevaux, *Types de données et algorithmes* (McGrawHill, 1990)

```
> more /proc/498/maps
08048000-08049000 r-xp 0000000000000000 03:06 1076411 /root/exemple
08049000-0804a000 rw-p 0000000000000000 03:06 1076411 /root/exemple
40000000-40013000 r-xp 0000000000000000 03:06 1008910 /lib/ld-2.1.3.so
40013000-40014000 rw-p 0000000000012000 03:06 1008910 /lib/ld-2.1.3.so
40014000-40016000 rw-p 0000000000000000 00:00 0
4001a000-4001b000 rw-p 0000000000000000 00:00 0
4001b000-400f6000 r-xp 0000000000000000 03:06 1008915 /lib/libc.so.6
400f6000-400fb000 rw-p 00000000000da000 03:06 1008915 /lib/libc.so.6
400fb000-400fe000 rw-p 0000000000000000 00:00 0
bfffe000-c0000000 rwxp ffffffffffffe000 00:00 0
```

Les régions apparaissant correspondent, deux par deux :

- au code et aux données initialisées du processus ;
- au code et aux données initialisées du chargeur de programme `ld-2.1.3.so` ;
- aux tas du processus et du chargeur de programme ;
- au code et aux données initialisées de la librairie C `libc.so.6` ;
- au tas de la librairie C et à la pile du processus.

b) Appels système liés à la création et à la modification des régions

Le système Linux manipule l'espace d'adressage des processus et les régions associées dans les cas suivants :

- création d'un nouveau processus *via* l'appel système `fork()`. L'espace d'adressage du processus père est dupliqué pour créer le processus fils par le mécanisme de copie sur écriture (*cf.* paragraphe 5.5.2) qui entraîne l'allocation de nouvelles régions ;
- recouvrement du code et des données d'un processus par un nouveau code et de nouvelles données suite à l'exécution d'un appel système de la famille `exec()`. Les anciennes régions héritées au moment du `fork()` sont libérées et de nouvelles régions sont allouées pour contenir le nouveau code et les nouvelles données du processus ;
- projection d'un fichier en mémoire centrale. Le système alloue alors une nouvelle région pour contenir le fichier ;
- création d'une région de mémoire partagée, c'est-à-dire d'une zone de mémoire contenant des données pouvant être partagées avec d'autres processus (*cf.* chapitre 7). Le système alloue alors une nouvelle région pour implémenter cet outil de communication ;
- allocation de mémoire dynamique au processus dans la région mémoire correspondant au tas. Le noyau accroît alors la taille de la région.

Les appels système `fork()` et `exec()` ont été présentés au chapitre 2 de cet ouvrage. Les appels système relatifs aux régions de mémoire partagée sont présentés au chapitre 7. Nous nous intéressons donc ici uniquement aux appels système liés à la projection des fichiers en mémoire et à l'allocation de mémoire dynamique.

Projection d'un fichier en mémoire centrale

Description des primitives

Les primitives `mmap()` et `munmap()` permettent respectivement de projeter un fichier en mémoire centrale et de mettre fin à cette projection. Les prototypes de ces fonctions sont :

```
#include <unistd.h>
#include <sys/mman.h>
void *mmap (void *debut, size_t longueur, int prot, int flags, int desc,
off_t offset);
int munmap (void *debut, size_t longueur);
```

L'appel système `mmap()` projette le fichier désigné par le descripteur `desc` à compter de l'octet dont la position est définie par `offset` dans une zone mémoire dont l'adresse de départ est spécifiée par `debut`. Le paramètre `longueur` indique le nombre d'octets à projeter en mémoire. Il faut noter que l'adresse `debut` est donnée à titre indicatif, le système pouvant utiliser une tout autre adresse pour allouer la région. L'adresse effective d'allocation de la région est retournée par la primitive.

Le paramètre `prot` spécifie le type de protection à appliquer à la région. Il peut prendre l'une des quatre valeurs suivantes :

- `PROT_NONE`, la région est marquée comme étant inaccessible ;
- `PROT_READ`, la région est accessible en lecture ;
- `PROT_WRITE`, la région est accessible en écriture ;
- `PROT_EXEC`, la région est accessible en exécution.

Le paramètre `flag` permet de spécifier certaines propriétés affectées à la région. Ce paramètre peut prendre l'une des valeurs suivantes :

- `MAP_FIXED`, l'adresse d'allocation de la région doit absolument être celle spécifiée dans le paramètre `debut` ;
- `MAP_SHARED`, la projection est partagée avec tous les autres processus ayant eux-mêmes projeté le fichier. Une modification effectuée dans le contenu de la projection est immédiatement visible par tous les autres processus partageant la région ;
- `MAP_PRIVATE`, la projection est privée. Elle n'est pas partagée avec tous les autres processus ayant eux-mêmes projeté le fichier. Une modification effectuée dans le contenu de la projection n'est pas visible par tous les autres processus ayant projeté le fichier ;
- `MAP_ANONYMOUS`, la projection ne concerne aucun fichier. Une région vide est créée ;
- `MAP_DENYWRITE`, une tentative d'accès en écriture retourne l'erreur `ETXTBSY` ;
- `MAP_LOCKED`, la région créée est verrouillée en mémoire.

L'appel système `munmap()` supprime la projection mémoire dont l'adresse de début est spécifiée par le paramètre `debut` et dont la taille en octets est égale à `longueur`.

Exemple

```
/*********************************************************************/
/*          Programme qui projette un fichier en mémoire             */
/*                          et lit son contenu                       */
/*              le fichier contient 4 enregistrements                */
/*                       de type correspondant                       */
/*********************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

main()
{
struct correspondant {
              char nom[10];
              char telephone[10];
             };
int fd, i;
struct correspondant *un_correspondant;
char *adresse, nom [10];

fd = open ("/root/annuaire",O_RDWR);
if (fd == -1)
     perror ("prob open");
adresse = (char*) mmap(NULL,4*sizeof(struct correspondant),
PROT_READ,MAP_SHARED, fd, (off_t)0);
close(fd);

/*les données du fichier sont accédées à la manière d'un tableau*/
/*dont le premier élément est situé au premier octet de la région*/
i = 0;
while (i < 4 *sizeof(struct correspondant))
{
putchar (adresse[i]);
i = i + 1;
}
}
```

Allocation de mémoire dynamique

Description des primitives

Les fonctions de la famille `malloc()` permettent d'allouer ou de libérer de la mémoire dynamique. Leurs prototypes sont :

```
#include <stdlib.h>
void *malloc (size_t taille);
void *calloc (size_t nombre, size_t taille);
void *reaalloc (void *adresse, size_t taille);
void free (void *adresse);
```

La fonction `malloc()` alloue une zone mémoire de `taille` octets. Elle retourne en cas de succès un pointeur sur la zone allouée et la valeur `NULL` sinon.

La fonction `calloc()` alloue un tableau de nombre zones mémoire de `taille` octets. Elle retourne en cas de succès un pointeur sur la zone allouée et la valeur `NULL` sinon.

La fonction `realloc()` permet de modifier la taille d'une zone préalablement allouée par un appel à la fonction `malloc()`. Le paramètre `adresse` correspond à l'adresse de la zone mémoire retournée par la fonction `malloc()` et `taille` spécifie la nouvelle taille de la zone. La fonction retourne en cas de succès un pointeur sur la nouvelle zone allouée et la valeur `NULL` sinon.

La fonction `free()` libère une zone mémoire dont l'adresse est donnée par le paramètre `adresse`.

Ces fonctions font appel à la primitive `brk()` qui permet de modifier la taille de la section de données d'un processus. Le prototype de cette primitive est :

```
#include <unistd.h>
int brk (void *fin_section_donnees);
```

Le paramètre `fin_section_donnees` spécifie l'adresse de la nouvelle fin de la section des données du processus. Elle doit être supérieure à l'adresse de fin de section de code et inférieure à l'adresse de début de pile.

Exemple

```
/*********************************************************************/
/*      Programme qui crée un fichier et stocke des données          */
/*             de type correspondant dans le fichier                 */
/*********************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

main()
{
struct correspondant {
               char nom[10];
               char telephone[10];
              };
int fd, i;
struct correspondant *un_correspondant;

fd = open ("/root/annuaire",O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
if (fd == -1)
     perror ("prob open");
i = 0;
while (i<4)
{
```

Exemple

```
un_correspondant = (struct correspondant*) malloc (sizeof(struct
correspondant));
    printf("Donnez un nom:\n");
    scanf ("%s", un_correspondant->nom);
    printf("Donnez un numéro de téléphone:\n");
    scanf ("%s", un_correspondant->telephone);
    write (fd, un_correspondant, sizeof(struct correspondant));
    i = i + 1;
}
close(fd);
}
```

5.2.2 Mise en œuvre de la pagination

a) Pages et table des pages

La mémoire centrale est divisée en cases et l'ensemble des régions composant l'espace d'adressage d'un processus sont elles-mêmes divisées en pages. La taille d'une page ou d'une case est fixée par la constante `PAGE_SIZE` dans le fichier `asm/page.h`. Cette valeur est égale à 4 Ko pour un processeur de la famille x86.

Afin de pouvoir gérer l'espace d'adressage du processus sans que la table des pages occupe trop de place en mémoire centrale[1], Linux utilise une table des pages définie sur trois niveaux (figure 5.17) et une adresse paginée est un quadruplet <*gp*, *ip*, *p*, *d*> pour lequel :

- *gp* repère une entrée d'une première table appelée table globale. Chaque entrée de la table globale contient l'adresse d'une case contenant une table intermédiaire ;
- *ip* repère une entrée dans la table intermédiaire désignée *via gp.* Chaque entrée de la table intermédiaire contient l'adresse d'une case contenant une table des pages ;
- *p* repère une entrée dans la table des pages désignée par *ip.* Chaque entrée de la table des pages contient l'adresse d'une case contenant la page *p* de l'espace d'adressage du processus ;
- *d* est le déplacement dans la page *p* de l'espace d'adressage du processus.

Un registre de la MMU maintient l'adresse d'implantation en mémoire centrale de la table globale du processus courant (registre de contrôle cr3 sous un processeur x86).

Une entrée de table a le format suivant :

- le champ `present` indique si la page ou la table des pages référencée par l'entrée est présente en mémoire centrale (1 vrai, 0 faux) ;
- le champ `accédé` (*accessed*) est positionné à chaque fois que la page correspondante est accédée ;

1. Les processeurs de l'architecture Intel x86 peuvent adresser 4 Go. La taille des pages est de 4 Ko et une entrée de la table des pages occupe 4 octets. Dans ces conditions, la table des pages peut contenir 1 048 576 entrées ce qui représente une occupation mémoire de 4 Mo.

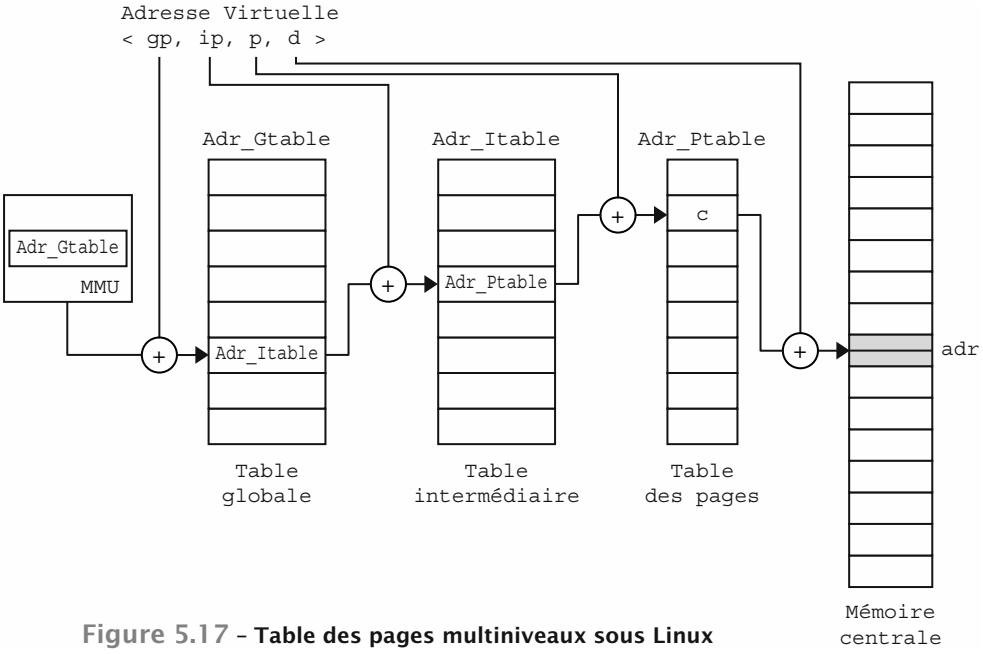

Figure 5.17 – Table des pages multiniveaux sous Linux

- le champ `modifié` (*dirty*) indique si la page présente en mémoire centrale a été modifiée. Ce champ n'est utilisé que pour la table des pages elle-même ;
- le champ `lecture/écriture` (*read/write*) contient les droits d'accès associés à la page ou à la table des pages ;
- le champ `utilisateur/superviseur` (*user/supervisor*) définit le niveau de privilège nécessaire pour accéder à la page ou à la table des pages ;
- les champs `PCD` et `PWT` indiquent respectivement si le cache du processeur est activé ou non et quelle stratégie entre *write-back* et *write-through*, doit être mise en œuvre. Par défaut, le cache est toujours activé et la stratégie mise en œuvre est de type *write-through* ;
- l'adresse physique d'une case.

Linux maintient par ailleurs un descripteur d'état pour chacune des cases de la mémoire centrale (structure `page` définie dans le fichier `<linux/mm.h>`). Ce descripteur, placé dans une liste doublement chaînée, contient notamment les champs suivants :

- les adresses de la case libre précédente et suivante ;
- l'adresse du descripteur de l'inode correspondant au contenu de la case ;
- le nombre de références à la case, c'est-à-dire le nombre de processus se partageant la page placée dans la case (champ `count`) ;
- les drapeaux d'état de la page (page verrouillée, page accédée, page dont le contenu est à jour, etc.) ;
- un booléen indique si la page a été modifiée ou non (champ `dirty`) ;
- un compteur utilisé dans le cadre du remplacement de pages (champ `age`).

b) Protection et partage des pages

Protection

Chaque page est qualifiée par des droits d'accès qui lui sont associés. Ces droits correspondent aux droits d'accès associés à la région à laquelle la page appartient. Lorsqu'un processus effectue un accès à une page, le processeur vérifie si l'opération demandée par le processus est compatible avec les droits associés à la page. Si tel n'est pas le cas, une trappe est levée (trappe 13) et le signal `SIGSEGV` est délivré au processus (*cf.* chapitre 6). L'exécution du processus est achevée par le noyau.

Copie sur écriture

Le noyau Linux permet à plusieurs processus de partager un ensemble de pages. Notamment, ce partage de pages est mis en œuvre par le noyau dans les deux cas suivants :

- les processus exécutent le même code. Les pages contenant le code exécutable étant seulement accédées en lecture, le noyau ne duplique évidemment pas ces pages ;
- lorsqu'un processus crée un fils, son espace d'adressage ainsi que ses tables des pages doivent être dupliqués pour que le fils puisse en hériter. Les pages contenant le code exécutable sont mises en partage entre les deux processus. Par ailleurs, les pages restantes ne sont pas immédiatement dupliquées. Elles sont simplement dotées de droits d'accès en lecture seule et sont mises en partage entre les deux processus. Le champ `count` attaché à chaque case est initialisé avec le nombre de processus se partageant celle-ci (2 dans ce cas). Lorsque l'un des processus tente d'écrire dans l'une de ces pages, le noyau lève une trappe puisque le type d'accès n'est pas conforme avec les droits associés à la page. Le gestionnaire de trappe (fonction `do_sw_page()`) duplique alors la page pour le processus effectuant l'accès en écriture en marquant cette page comme étant accessible en écriture. Le champ `count` attaché à la case initiale est décrémenté de une unité. La page initiale reste seulement accessible en lecture. Elle devient accessible en écriture lors de l'accès en écriture du dernier processus partageant la page (champ `count` à 1). Cette technique est appelée *copie sur écriture* (*copy on write*) ; elle permet au noyau d'économiser de la place en mémoire centrale (les pages non modifiées ne sont jamais dupliquées) et de réduire le coût de l'opération `fork()`.

5.2.3 Mise en œuvre de la mémoire virtuelle

a) Défaut de pages

Les pages de l'espace d'adressage d'un processus sont chargées à la demande. Lorsqu'un processus cherche à accéder à une page de son espace d'adressage qui n'est pas présente en mémoire centrale (le champ `present` dans la table des pages est à faux), le noyau lève la trappe 14 qui entraîne l'exécution du gestionnaire `do_page_fault()`. Deux cas sont distingués :

- la page n'a jamais été chargée par le processus, ce qui correspond à une entrée de la table des pages emplie de 0. Si un fichier est projeté dans la page, alors le noyau invoque la fonction `do_no_page()` liée à la région, qui charge la page manquante. Si aucun fichier n'est projeté dans la page (paramètre `MAP_ANONYMOUS` au moment de la projection dans la région mémoire), le noyau adresse une page particulière appelée *page zéro* remplie de valeurs nulles et applique sur celle-ci le principe de copie à l'écriture. La page zéro est créée statiquement à l'initialisation du noyau et placée dans la sixième case de la mémoire centrale. Cette page n'est donc réellement dupliquée dans l'espace d'adressage du processus que lorsque celui-ci effectue un accès en écriture sur celle-ci ;
- la page a déjà été chargée par le processus mais elle a été retirée de la mémoire centrale et sauvegardée sur le disque, dans la zone de *swap*. L'entrée n'est pas emplie de 0 mais contient l'adresse de la page dans la zone de *swap*. Le noyau effectue une opération d'entrées-sorties en utilisant cette adresse pour ramener la page en mémoire centrale.

b) Gestion des cases libres

Le noyau gère l'ensemble des cases libres de la mémoire centrale par l'intermédiaire d'une table `free_area` comportant 10 entrées. Chacune des entrées comprend deux éléments :

- une liste doublement chaînée de blocs de pages libres, l'entrée *i* gérant des blocs de 2^i pages ;
- un tableau binaire nommé *bitmap*, permettant de connaître les blocs de pages alloués. Un bit du tableau gère un ensemble de deux blocs de pages. Si le bit est à 0, alors les deux blocs de pages correspondant sont tous les deux libres ou tous les deux utilisés. Si le bit vaut 1, alors au moins l'un des deux blocs est occupé.

Le noyau gère ainsi 10 listes de blocs de pages dont la taille est soit de 1 page, 2 pages, 4, 8, 16, 32, 64, 128, 256 et 512 pages (figure 5.18). Il effectue des demandes d'allocation pour des blocs de pages contiguës de taille supérieure à 1, notamment pour satisfaire les requêtes mémoires de composants ne connaissant pas la pagination telles que le DMA et certains pilotes d'entrées-sorties.

Lorsque le noyau traite une demande d'allocation pour un nombre de *i* pages consécutives, il prend dans la liste *i* de la table `free_area` le premier bloc libre. Si aucun bloc n'a été trouvé dans cette liste, le noyau explore les listes de taille supérieure *j* ($j > i$). Dans ce dernier cas, i cases sont allouées dans le bloc de *j* cases et les $j - i$ cases restantes sont replacées dans les listes de taille inférieure.

Lorsque le noyau libère un bloc de *i* pages libres, il replace ce bloc dans une des listes de la table `free_area`, en tentant de fusionner ce bloc avec ses voisins si ceux-ci sont de même taille et si ils sont libres également. Ainsi, un bloc de 2 pages libres ayant un voisin comportant également 2 pages libres, forme un groupe de 4 pages libres. Ce groupe de 4 pages libres est inséré dans la liste de la troisième entrée de la table `free_area`, à moins qu'il ne comporte lui aussi un voisin de 4 pages libres, auquel cas il forme un groupe de 8 pages libres.

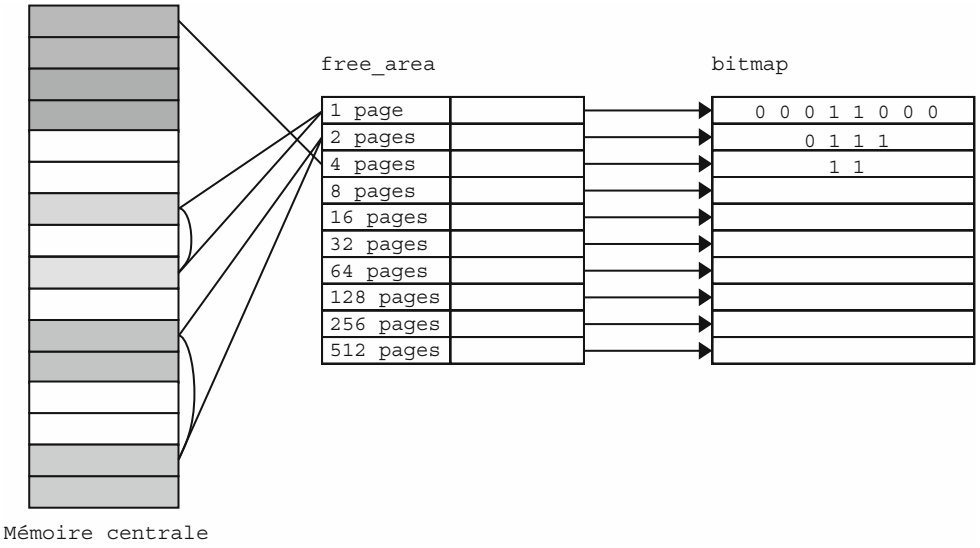

Figure 5.18 – Table des blocs de pages libres `free_area`

L'algorithme se poursuit ainsi jusqu'à former au maximum un groupe de 512 pages libres.

Ainsi, la figure 5.19 illustre la reconfiguration de la structure `free_area` suite à la libération de la case apparaissant en noir. Un bloc de 4 cases libres est créé.

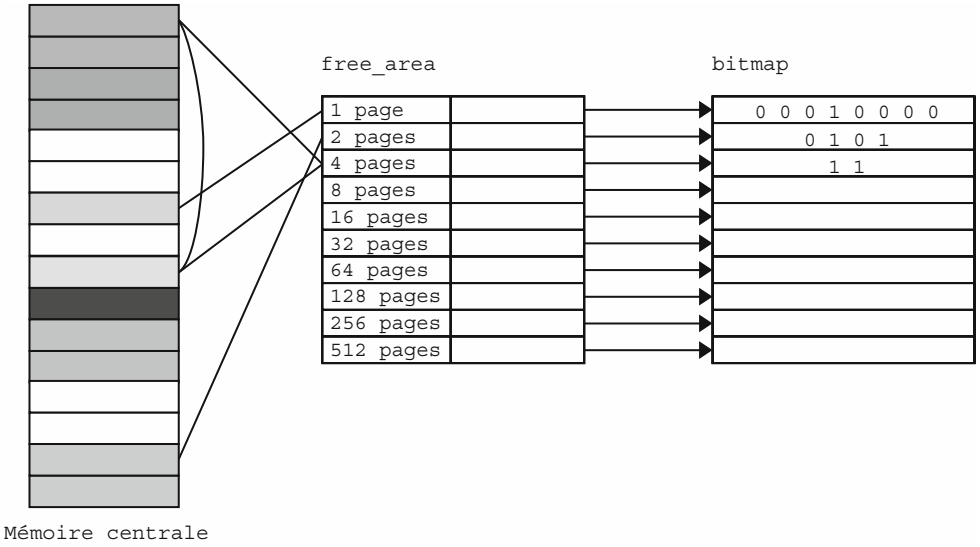

Figure 5.19 – Libération de pages et reconfiguration de `free_area`

c) Remplacement et vidage de pages

Lorsque le noyau a besoin de place en mémoire centrale, il décharge des pages présentes dans les cases de la mémoire centrale. Ces pages sont placées dans une zone disque, appelée *zone de swap*.

Remplacement de pages

Le thread noyau `kswapd` est responsable de la sélection de pages à retirer de la mémoire centrale. Ce thread, encore connu sous le nom de dérobeur de pages, est activé toutes les 10 secondes. Il s'exécute et récupère des pages en mémoire centrale si le nombre de cases libres est tombé en dessous du seuil `freepages.high`.

Le champ `accédé` de la table des pages et le champ `âge` du descripteur de case sont utilisés par le thread `kswapd` pour choisir des pages victimes.

Une page est victime du dérobeur de pages si elle a atteint un âge donné (paramètre système) sans être référencée. Plus précisément :

- à chaque fois qu'une page est référencée, l'âge de la page devient égal à 0 et le bit `accédé` est mis à 1 ;
- à chacun de ses passages, le dérobeur de pages met à 0 le bit `accédé` si il est à 1 puis incrémente l'âge de la page.

Une page est victime si son bit `accédé` est à 0 et si elle a atteint l'âge limite. La page victime est écrite dans la zone de swap. Une adresse dans le périphérique de swap lui est alors affectée qui est composée du numéro du périphérique et de la position de la page dans la zone de swap. Cette adresse est sauvegardée dans l'entrée de la table des pages correspondant à la page. Ainsi, lors d'un défaut de page, le noyau utilise cette adresse pour trouver la page dans la zone de swap et la lire pour la ramener en mémoire centrale.

Verrouillage des pages en mémoire centrale

Un processus privilégié peut choisir de verrouiller ses pages en mémoire centrale afin qu'elles ne puissent pas être victimes du thread noyau `kswapd`.

Les primitives de la famille `mlock()` permettent d'effectuer ce verrouillage. Les prototypes de ces fonctions sont :

```
#include <sys/mman.h>
int mlock (const void *adresse, size_t longueur);
int munlock (const void *adresse, size_t longueur);
int mlockall (int flags);
int munlockall (void);
```

La primitive `mlock()` permet de verrouiller la zone mémoire dont l'adresse de début est `adresse` et la taille en octets est égale à `longueur`.

La primitive `munlock()` permet de déverrouiller la zone mémoire dont l'adresse de début est `adresse` et la taille en octets est égale à `longueur`.

La primitive `mlockall()` permet de verrouiller toutes les pages appartenant à l'espace d'adressage du processus. Le paramètre `flags` détermine des options par rapport à ce verrouillage :

- `MCL_CURRENT`, toutes les pages du processus chargées en mémoire centrale sont verrouillées ;
- `MCL_FUTURE`, toutes les prochaines pages du processus chargées en mémoire centrale, seront verrouillées.

La primitive `munlockall()` permet de déverrouiller toutes les pages appartenant à l'espace d'adressage du processus.

Gestion de la zone de swap

Un *périphérique de swap* est un périphérique en mode bloc, une partition ou encore un fichier régulier. Il est constitué comme étant une suite d'emplacements de pages. Le noyau Linux peut utiliser plusieurs périphériques de *swap* en même temps. En effet, chaque zone de swap est qualifiée par une priorité. Lorsque le noyau doit sauvegarder une page dans une zone de *swap*, il explore les zones actives par ordre de priorité décroissante et sélectionne la première zone dans laquelle il reste de la place disponible.

Un périphérique de *swap* est initialisé à l'aide de la commande `mkswap()`. Cette commande crée un répertoire au début du périphérique. Ce répertoire a une taille égale à celle d'une page mémoire et abrite un tableau de bits, chaque bit correspondant à une page de l'espace de *swap* et indiquant si la page est utilisable ou pas. Ainsi une valeur de bit égale à 1 indique une page occupée tandis qu'une valeur de bit égale à 0 indique une page libre.

Un périphérique de *swap* nommé `ref` est rendu actif par un appel à la primitive `swapon (const char *ref, int parametre)` où `parametre` définit la priorité affectée au périphérique. Le noyau crée un descripteur pour chacune des zones de *swap* actives, qui maintient des informations d'états sur cette zone de swap telles que la priorité qui lui est affectée, le nombre de pages total qu'elle comprend, le nombre de pages disponibles, etc.

La désactivation d'un périphérique de *swap* nommé `ref` est obtenue par un appel à la primitive `swapoff (const char * ref)`.

Exercices

5.1 Qu'avez-vous retenu ?

Pour chacune des questions suivantes, choisissez la ou les réponses qui vous semblent exactes.

Question 1 – Un processus dispose d'une table des pages à trois entrées. Deux pages de son espace d'adressage, les pages 1 et 3 sont chargées en mémoire centrale dans les cases 4 et 8. Il accède à l'adresse logique `<page 3, déplacement 10>`. Que se passe-t-il ?

- ☐ a. Il se produit un défaut de page.
- ☐ b. L'adresse physique générée est `<case 5, déplacement 10>`.
- ☐ c. L'adresse physique générée est `<case 8, déplacement 10>`.
- ☐ d. Il y a remplacement de pages.

Question 2 – Le même processus génère maintenant l'adresse logique `<page 2, déplacement 24>`. Que se passe-t-il ?

- ☐ a. Il se produit un défaut de page.
- ☐ b. L'adresse physique générée est `<case 5, déplacement 10>`.
- ☐ c. L'adresse physique générée est `<case 8, déplacement 10>`.
- ☐ d. Il y a remplacement de pages.

Question 3 – Une mémoire centrale comporte 3 cases libres initialement vides. Un processus effectue les accès suivants à ses pages : 1, 2, 3, 1, 4. Lors de l'accès à la page 4 :

- ☐ a. avec une stratégie FIFO, la page 1 est remplacée.
- ☐ b. avec une stratégie FIFO, la page 3 est remplacée.
- ☐ c. avec une stratégie LRU, la page 1 est remplacée.
- ☐ d. avec une stratégie LRU, la page 2 est remplacée.

Question 4 – Le mécanisme de copie à l'écriture signifie que :

- ☐ a. les pages modifiées d'un processus sont tout de suite recopiées en zone de *swap*.
- ☐ b. les pages partagées entre un processus fils et un processus père ne sont dupliquées dans l'espace d'adressage du fils que lors d'un accès en écriture.
- ☐ c. le noyau sauvegarde les pages avant qu'elles ne soient modifiées en mémoire centrale.

Question 5 – Une zone de *swap* :

- ☐ a. est un périphérique bloc organisé comme une suite de pages.
- ☐ b. est une mémoire tampon avec un périphérique bloc.
- ☐ c. est une zone en mémoire centrale contenant la table des pages des processus.

Question 6 – Il se produit un défaut de page :

- ☐ a. lorsque le *thread* noyau `kswapd` libère des cases de la mémoire centrale.
- ☐ b. lorsqu'un processus accède à une page de son espace d'adressage qui n'est pas en mémoire centrale.
- ☐ c. lorsqu'un processus accède à une page qui ne fait pas partie de son espace d'adressage.

Question 6 – Un processus effectue les accès suivants aux pages de son espace d'adressage : P1, P2, P4, P5, P6, P5, P4. La mémoire centrale comporte 3 cases. Lors d'un nouvel accès à la page P1 :

- ☐ a. La page P4 est remplacée avec un algorithme de remplacement de page FIFO.
- ☐ b. La page P2 est remplacée avec un algorithme de remplacement de page FIFO.
- ☐ c. La page P6 est remplacée avec un algorithme de remplacement de page LRU.
- ☐ d. La page P5 est remplacée avec un algorithme de remplacement de page LRU.

5.2 Pagination

On considère une machine pour laquelle la mémoire est gérée par le mécanisme de pagination à la demande. La mémoire physique est découpée en un ensemble de 20 cases numérotées de 1 à 20. La taille d'une page est de 1 Ko. Dans ce contexte, soit trois processus P1, P2 et P3 dont les espaces d'adressages sont respectivement composés de 4 pages, 6 pages et 5 pages (les pages sont elles aussi numérotées à partir de 1).

À un instant *t*, la configuration de la mémoire centrale est la suivante :

- pour le processus P1, seules ses pages 1 et 4 sont chargées, respectivement dans les cases 1 et 6 ;
- pour le processus P2, seules ses pages 1, 2 et 5 sont chargées, respectivement dans les cases 10, 7 et 15 ;
- pour le processus P3, seules ses pages 5, 1, 3 et 4 sont chargées, respectivement dans les cases 8, 9, 14 et 11.

Questions :

1) Le processus P2 accède à l'adresse paginée `<page 5, déplacement 12>` :

- ◇ Il y a défaut de page.
- ◇ Il accède à l'adresse physique `<case 15, dep 12>`.
- ◇ Il accède à l'adresse physique `<case 8, dep 12>`.
- ◇ Il accède à l'adresse physique `14 348`.
- ◇ Il accède à l'adresse physique `15 372`.
- ◇ Il accède à l'adresse physique `68`.

2) Le processus P3 accède à l'adresse paginée <page 2, déplacement 14> :
   - Il y a défaut de page.
   - Il accède à l'adresse physique <case 7, dep 14>.
   - Il accède à l'adresse physique 6 158.
   - Il accède à l'adresse physique 7 182.

3) Le processus P1 accède à l'adresse linéaire 3 328 :
   - Il y a défaut de page.
   - Il accède à l'adresse physique <case 6, dep 256>.
   - Il accède à l'adresse paginée <page 4, dep 256>.
   - Il accède à l'adresse physique 5 376.

5.3 Remplacement de pages

Soit la liste des pages virtuelles référencées aux instants t = 1, 2, …, 11 : 4 5 6 8 4 9 6 12 4 6 10.

La mémoire centrale est composée de 4 cases initialement vides. Représentez l'évolution de la mémoire centrale au fur et à mesure des accès pour chacune des deux politiques de remplacement de pages FIFO et LRU. Notez les défauts de pages éventuels.

5.4 Programmation

On considère un fichier contenant un ensemble de données concernant des stocks de 4 produits différents disponibles. Chaque produit est enregistré dans le fichier sous la forme d'un numéro codé sur 5 caractères auquel est associé le nombre d'exemplaires disponibles de ce produit codé sur 2 caractères.

Écrivez un programme dans lequel un processus projette ce fichier en mémoire. Puis il lit le contenu de ce fichier, et pour chaque couple <numéro de produit, nombre d'exemplaires du produit>, il insère un élément composé de deux champs <identifiant_produit et nombre_exemplaire>, dans une liste chaînée simple.

5.5 Mémoire paginée

On considère une mémoire paginée pour laquelle les cases en mémoire centrale sont de 1 Ko. La mémoire centrale compte au total pour l'espace utilisateur 20 cases numérotées de 1 à 20. Dans ce contexte, on considère trois processus A, B et C. Le processus A a un espace d'adressage composé de 6 pages P1, P2, P3, P4, P5 et P6. Le processus B a un espace d'adressage composé de 4 pages, P1 à P4. Le processus C a un espace d'adressage composé de 2 pages, P1 et P2. Pour le processus A, seules les pages P1, P5, P6 sont chargées en mémoire centrale respectivement dans les cases 2, 4, 1. Pour le processus B, seule la page P1 est chargée en mémoire centrale dans la case 5. Pour le processus C, seule la page P2 est chargée en mémoire centrale dans la case 12.

1) Représentez sur un dessin les structures allouées pour ce type d'allocation mémoire et la mémoire centrale correspondant à l'allocation décrite.

2) Les trois processus A, B et C sont décrits par un bloc de contrôle qui contient entre autres les informations suivantes :
   - Pour le processus A, compteur ordinal CO = (page P5, déplacement 16), adresse table des pages = ADRP1 (adresse table des pages A) ;
   - Pour le processus B, compteur ordinal CO = (page P2, déplacement 512), adresse table des pages = ADRPB ;
   - Pour le processus C, compteur ordinal CO = (page P1, déplacement 32), adresse table des pages = ADRPC.

Le compteur ordinal CO contient l'adresse de l'instruction à exécuter.

- Le processus A devient actif. Décrivez le processus de conversion d'adresse pour l'instruction exécutée à sa reprise. Quelle valeur contient le registre PTBR ? Quelle adresse physique correspond à l'adresse virtuelle de l'instruction exécutée ?
- Maintenant le processus A est préempté et le processus B est élu. Décrivez succinctement l'opération de commutation de contexte qui a lieu notamment en donnant les nouvelles valeurs des registres CO et PTBR. Que se passe-t-il lorsque le processus B reprend son exécution ?
- Chaque entrée de table des pages contient un champ de bits permettant de spécifier les droits d'accès associés à une page. Ce champ est composé de trois bits `x`, `r`, `w` avec la signification suivante :
  – `x` : 0 pas de droit en exécution sur la page, 1 droit en exécution accordé.
  – `r` : 0 pas de droit en lecture sur la page, 1 droit en lecture accordé.
  – `w` : 0 pas de droit en écriture sur la page, 1 droit en écriture accordé.
  Ce champ « droit » à la valeur 010 pour la page P1 du processus A. Le processus A exécute l'instruction `STORE R1` (page P1, déplacement 128) qui effectue l'écriture du contenu du registre processeur R1 à l'adresse (page P1, déplacement 128). Que se passe-t-il ?

5.6 Mémoire paginée et ordonnancement

1) On considère le schéma suivant (figure 5.20) qui représente à l'instant t les structures utilisées par le système pour gérer l'allocation de la mémoire centrale à trois processus PA, PB, PC. Les cases mémoires ont une taille de 512 octets. Le premier octet de l'espace linéaire de chaque processus ou de chaque page a une adresse égale à 0.

   Pour chacune des adresses linéaires suivantes dans l'espace d'adressage des processus, donnez l'adresse paginée correspondante, puis l'adresse physique. Une nouvelle page est chargée dans la première case libre de la mémoire centrale.
   - 1054 pour le processus A
   - 500 pour le processus C
   - 534 pour le processus B.

2) Dans un système doté d'une pagination à la demande (une page est chargée au moment où le processus y accède pour la première fois), on considère trois pro-

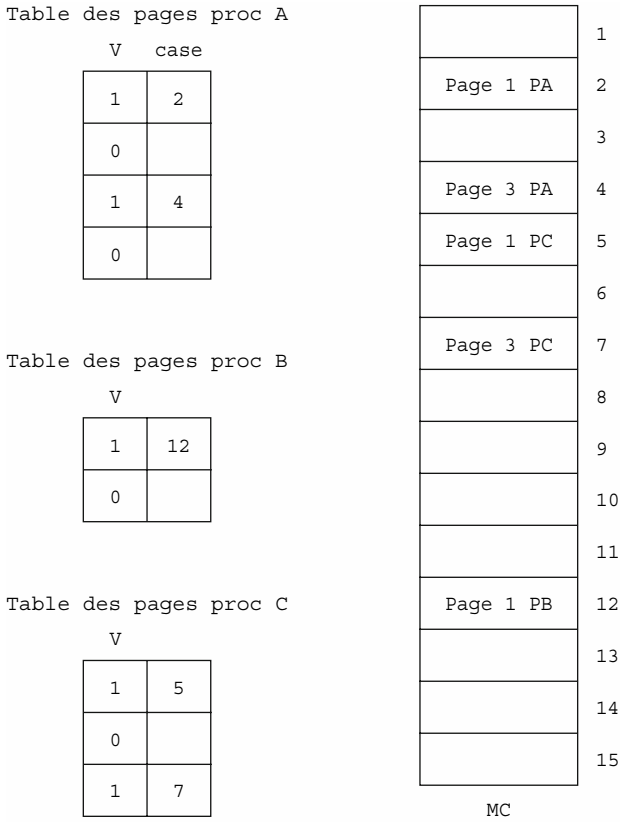

Figure 5.20 – Allocation de la mémoire centrale

cessus P1, P2 et P3. Lors de la création d'un processus, aucune page de l'espace d'adressage de ce processus n'est chargée en mémoire centrale. Lors d'un défaut de page, le système initialise un transfert disque de la page vers une case de la mémoire centrale, qui dure 20 ms. L'allocation du disque se fait selon un mode FIFO non préemptive. On supposera que le nombre de cases de la mémoire centrale est important et en conséquence, on ne se préoccupera pas des problèmes de remplacement de pages.

L'allocation du processeur principal se fait selon un mode de priorité préemptive. À ce propos, on a priorité (P2) > priorité (P3) > priorité (P1). Les trois processus sont tous les trois prêts à s'exécuter à l'instant 0.

Le comportement des trois processus vous est donné ci-après. Établissez les chronogrammes d'exécution des trois processus P1, P2, P3 en faisant apparaître les états prêt, bloqué, et élu.

Processus P1	Processus P2	Processus P3
calcul utilisant la page 1 durant 70 ms calcul utilisant la page 1 et 2 durant 10 ms calcul utilisant la page 1 durant 10 ms fin de P1	calcul utilisant la page 3 durant 20 ms calcul utilisant la page 1 et 2 durant 30 ms calcul utilisant la page 4 durant 10 ms calcul utilisant la page 5 durant 30 ms fin de P2	calcul utilisant la page 1 durant 30 ms calcul utilisant la page 4 durant 10 ms fin de P3

Solutions

5.1 Qu'avez-vous retenu ?

Question 1 – Un processus dispose d'une table des pages a trois entrées. Deux pages de son espace d'adressage, les pages 1 et 3 sont chargées en mémoire centrale dans les cases 4 et 8. Il accède à l'adresse logique `<page 3, déplacement 10>`. Que se passe-t-il ?

- ☐ c. L'adresse physique générée est `<case 8, déplacement 10>`.

Question 2 – Le même processus génère maintenant l'adresse logique `<page 2, déplacement 24>`. Que se passe-t-il ?

- ☐ a. Il se produit un défaut de page.

Question 3 – Une mémoire centrale comporte 3 cases libres initialement vides. Un processus effectue les accès suivants à ses pages : 1, 2, 3, 1, 4. Lors de l'accès à la page 4 :

- ☐ a. avec une stratégie FIFO, la page 1 est remplacée.
- ☐ d. avec une stratégie LRU, la page 2 est remplacée.

Question 4 – Le mécanisme de copie à l'écriture signifie que :

- ☐ b. les pages partagées entre un processus fils et un processus père ne sont dupliquées dans l'espace d'adressage du fils que lors d'un accès en écriture.

Question 5 – Une zone de *swap* :

- ☐ a. est un périphérique bloc organisé comme une suite de pages.

Question 6 – Il se produit un défaut de page :

- ☐ b. lorsqu'un processus accède à une page de son espace d'adressage qui n'est pas en mémoire centrale.

Question 7 – Un processus effectue les accès suivants aux pages de son espace d'adressage : P1, P2, P4, P5, P6, P5, P4. La mémoire centrale comporte 3 cases. Lors d'un nouvel accès à la page P1 :

- ☐ a. La page P4 est remplacée avec un algorithme de remplacement de page FIFO.
- ☐ c. La page P6 est remplacée avec un algorithme de remplacement de page LRU.

5.2 Pagination

1) Le processus P2 accède à l'adresse paginée <page 5, déplacement 12>.
   Il accède à l'adresse physique <case 15, dep 12>.
   Il accède à l'adresse physique 15 372 = (14 × 1 024) + 12.
2) Le processus P3 accède à l'adresse paginée <page 2, déplacement 14>.
   Il y a défaut de page ; la page 2 du processus P3 n'étant pas en mémoire centrale.
3) Le processus P1 accède à l'adresse linéaire 3 328.
   Il accède à l'adresse paginée <page 4, dep 256> car 3 328 = (3 × 1 024) + 256.
   Il accède à l'adresse physique 5 376 = (5 × 1 024) + 256. La page 4 du processus P1 se trouve dans la case 6.

5.3 Remplacement de pages

- FIFO

accès	4	5	6	8	4	9	6	12	4	6	10
case 1	4	4	4	4	4	**9**	9	9	9	9	10
case 2		5	5	5	5	5	5	**12**	**12**	12	12
case 3			6	6	6	6	6	6	**4**	**4**	4
case 4				8	8	8	8	8	8	**6**	**6**
défaut	D	D	D	D		D		D	D	D	D

- LRU

accès	4	5	6	8	4	9	6	12	4	6	10
case 1	4	4	4	4	4	4	4	4	4	4	4
case 2		5	5	5	5	**9**	9	9	9	9	**10**
case 3			6	6	6	6	6	6	6	**6**	6
case 4				8	8	8	8	**12**	**12**	12	12
défaut	D	D	D	D		D		D			D

5.4 Programmation

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
main()
{
struct produit {
               char identifiant[5];
               char nombre_exemplaire[2];
               struct produit *suivant;
              };
int fd, i, ret, j;
struct stat st;
struct produit *un_produit, *courant;
char *adresse;
/* obtention de la taille du fichier */
stat ("/home/delacroi/produit", &st);
fd = open ("/home/delacroi/produit",O_RDONLY);
if (fd == -1)
     perror ("prob open");
adresse = (char*) mmap(NULL,st.st_size, PROT_READ,MAP_SHARED, fd,
(off_t)0);
close(fd);
courant = NULL;
i = 0;
while (i < 4 * 7)
{
un_produit = (struct produit*) malloc (sizeof(struct produit));
j = 0;
while (j < 5) {
un_produit->identifiant[j] = adresse[i];
i = i + 1;
j = j + 1;
}
j = 0;
while (j < 2) {
un_produit->nombre_exemplaire[j] = adresse[i];
i = i + 1;
j = j + 1;
}
if (courant == NULL) courant = un_produit;
else
    {
    un_produit->suivant = NULL;
    courant->suivant = un_produit;
```

```
    courant = un_produit;
    }
}
}
```

5.5 Mémoire paginée

1) La figure 5.21 donne les structures allouées.

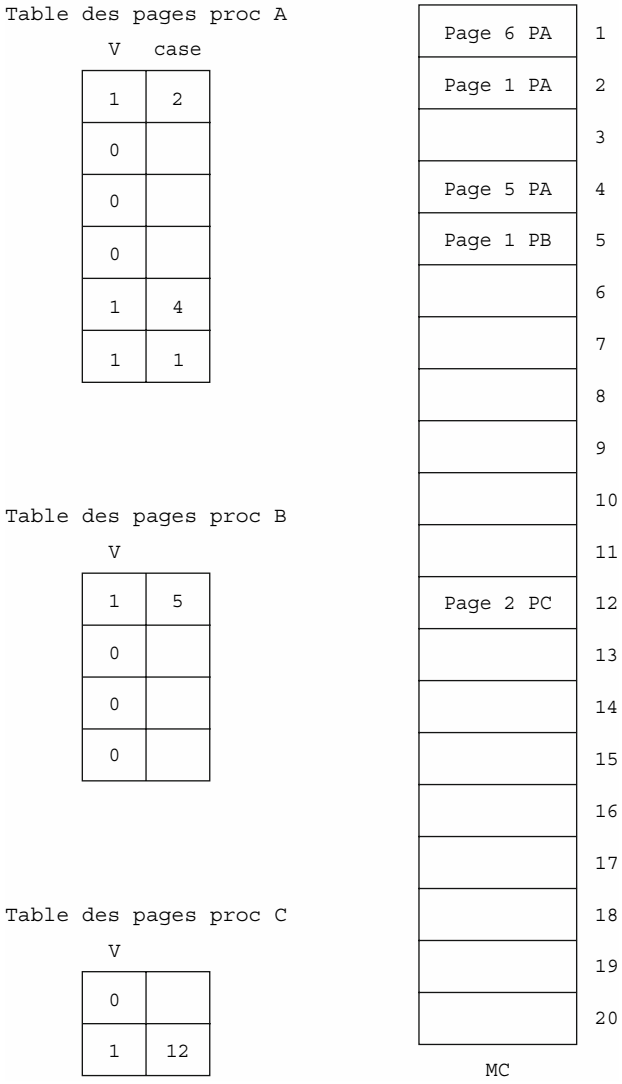

Figure 5.21 – Allocation de la mémoire centrale

2)

- L'instruction de reprise est celle dont l'adresse correspond au CO = (page P5, déplacement 16). La valeur de PTBR est égale à ADRPA.
  La conversion de l'adresse s'effectue selon le processus suivant :
  1) accès à la table *via* le PTBR ;
  2) accès à l'entrée 5 de la table des pages du processus A et test du bit *V* ;
  3) le bit *V* étant à 1, la page est présente en mémoire centrale ; l'adresse physique correspondante est (case 4, déplacement 16) soit 3 Ko + 16 = 3 098.
- Les opérations suivantes sont réalisées lors de la préemption du processus A par le processus B :
  1) sauvegarde dans le bloc de contrôle du processus A de sa valeur de CO et PTBR ;
  2) restauration du contexte du processus B. Le registre CO est chargé avec l'adresse paginée (page P2, déplacement 512) et le registre PTBR prend la valeur ADRPB.
  Lorsque le processus B reprend soin exécution :
  1) accès à la table *via* le PTBR ;
  2) accès à l'entrée 2 de la table des pages du processus B et test du bit *V* ;
  3) le bit *V* étant à 0, la page est absente en mémoire centrale : il y a défaut de page.
- Les droits 010 correspondent à un droit en lecture seule pour la page P1 du processus A. L'instruction STORE étant une opération d'écriture en mémoire centrale, une trappe est levée. L'exécution du processus A est déroutée en mode système et arrêtée (envoi du signal SIGSEGV au processus fautif).

5.6 Mémoire paginée et ordonnancement

1) 1 054 pour le processus A : l'adresse paginée correspondante est (page 3, déplacement 30). L'adresse physique correspondante est (case 4, déplacement 30).
500 pour le processus C : l'adresse paginée correspondante est (page 1, déplacement 500). L'adresse physique correspondante est (case 5, déplacement 500).
534 pour le processus B : l'adresse paginée correspondante est (page 2, déplacement 22). L'adresse physique correspondante est (case 1, déplacement 22). Il y a défaut de page et la page 2 est chargée dans la case 1.

2) La figure 5.22 donne le chronogramme d'exécution des trois processus.

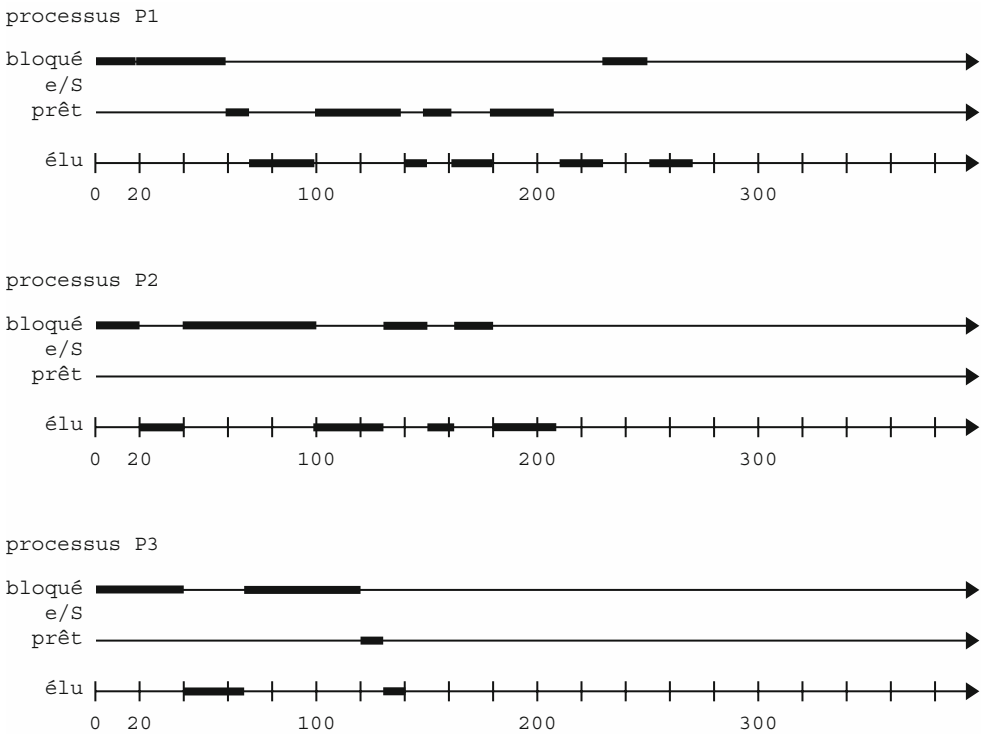

Figure 5.22 – Chronogramme d'exécution

GESTION DES SIGNAUX

6

PLAN

OBJECTIFS

- Ce chapitre présente le fonctionnement et la programmation des signaux sous Linux. Après avoir décrit l'ensemble des signaux admis par Linux, nous présentons la façon dont le noyau Linux traite ces signaux.
- Puis nous nous intéressons à l'interface de programmation associée aux signaux classiques, puis aux signaux temps réel.

6.1 PRÉSENTATION GÉNÉRALE

6.1.1 Définition

Un *signal* est un message envoyé par le noyau à un processus ou à un groupe de processus pour indiquer l'occurrence d'un événement survenu au niveau du système. Ce message ne comporte pas d'informations propres si ce n'est le nom du signal lui-même qui est significatif de l'événement rencontré par le noyau. La prise en compte du signal par le processus oblige celui-ci à exécuter une fonction de gestion du signal appelée *handler de signal*.

Ainsi avons-nous déjà rencontré au chapitre 2, le signal `SIGCHLD` qui permet au noyau d'avertir un processus père de la mort de son fils.

Au chapitre précédent, nous avons évoqué le signal `SIGSEGV` qui est associé à la trappe levée en cas de violation mémoire et provoque la terminaison du processus fautif.

Le mode de fonctionnement des signaux présentant quelques analogies avec le traitement des interruptions, ceux-ci sont souvent qualifiés d'interruptions logicielles, mais comme nous le verrons dans la suite de ce chapitre, les traitements de ces deux types d'interruptions sont réellement différents.

Les signaux sont par contre associés à la gestion des trappes par le noyau, car c'est grâce à cet outil, que le noyau signale au processus l'occurrence de la faute rencontrée.

6.1.2 Listes des signaux

Le noyau Linux 2.2 admet 64 signaux différents. Chaque signal est identifié par un numéro et est décrit par un nom préfixé par la constante `SIG`. Seul le signal 0 ne porte pas de nom. Les signaux 1 à 31 correspondent aux signaux classiques tandis que les numéros 32 à 63 correspondent aux signaux temps réel que nous étudierons à la fin de ce chapitre.

Le tableau 6.1 donne la liste des signaux classiques en précisant l'événement auquel ils sont attachés. Les signaux non définis par la norme POSIX sont distingués.

6.1.3 Champs du PCB associés aux signaux

Le descripteur de processus contient plusieurs champs lui permettant de gérer les signaux :

- le champ `signal` est une variable de type `sigset_t` qui stocke les signaux envoyés au processus. Cette structure est constituée de deux entiers de 32 bits, chaque bit représentant un signal. Une valeur à 0 indique que le signal correspondant n'a pas été reçu tandis qu'une valeur à 1 indique que le signal a été reçu ;
- le champ `blocked` est une variable de type `sigset_t` qui stocke les signaux bloqués c'est-à-dire les signaux dont la prise en compte est retardée ;
- le champ `sigpending` est un drapeau indiquant si il existe au moins un signal non bloqué en attente ;
- le champ `gsig` est un pointeur vers une structure de type `signal_struct` qui contient notamment pour chaque signal, la définition de l'action qui lui est associée.

6.2 Aspects du traitement des signaux par le noyau

Le traitement des signaux par le noyau recouvre plusieurs mécanismes que nous allons étudier :

- l'envoi d'un signal à un processus ;
- la prise en compte du signal reçu par le processus et l'action qui en résulte ;
- les interactions de certains appels systèmes avec les signaux.

6.2.1 Envoi d'un signal

Un processus peut recevoir un signal de deux façons différentes :

- un signal lui est envoyé par un autre processus par l'intermédiaire de la primitive `kill()` que nous détaillerons dans la partie programmation des signaux. Par exemple, le shell envoie le signal `SIGKILL` si la commande `kill -9 pid` est frappée ;

Tableau 6.1 – Signaux classiques

Numéro de signal	Nom du signal	Événement associé
1	SIGHUP	Terminaison du processus leader de la session
2	SIGINT	Interruption du clavier (frappe de « CTRL C »)
3	SIGQUIT	Caractère « quit » frappé depuis le clavier (frappe de « CTRL \ »)
4	SIGILL	Instruction illégale
5	SIGTRAP	Point d'arrêt pour le débogage (non POSIX)
6	SIGIOT/ SIGABRT	Terminaison anormale
7	SIGBUS	Erreur de bus
8	SIGFPE	Erreur mathématique virgule flottante
9	SIGKILL	Terminaison forcée du processus
10	SIGUSR1	Signal à disposition de l'utilisateur
11	SIGSEGV	Référence mémoire invalide
12	SIGUSR2	Signal à disposition de l'utilisateur
13	SIGPIPE	Écriture dans un tube sans lecteur
14	SIGALRM	Horloge temps réel, fin de temporisation
15	SIGTERM	Terminaison du processus
16	SIGSTKFLT	Erreur de pile du coprocesseur
17	SIGCHLD	Processus fils terminé
18	SIGCONT	Reprise de l'exécution d'un processus stoppé
19	SIGSTOP	Stoppe l'exécution d'un processus
20	SIGTSTP	Caractère « susp » frappé depuis le clavier (frappe de « CTRL Z »)
21	SIGTTIN	Lecture par un processus en arrière-plan
22	SIGTTOU	Écriture par un processus en arrière-plan
23	SIGURG	Données urgentes sur une socket
24	SIGXCPU	Limite de temps processeur dépassé
25	SIGXFSZ	Limite de taille de fichier dépassé
26	SIGVTALRM	Alarme virtuelle (non POSIX)
27	SIGPROF	Alarme du profileur (non POSIX)
28	SIGWINCH	Fenêtre redimensionnée (non POSIX)
29	SIGIO	Arrivée de caractères à lire (non POSIX)
30	SIGPOLL	Équivalent à SIGIO (non POSIX)
31	SIGPWR	Chute d'alimentation (non POSIX)
32	SIGUNUSED	Non utilisé

- l'exécution du processus a levé une trappe et le gestionnaire d'exception associé positionne un signal pour signaler l'erreur détectée. Par exemple, l'occurrence d'une division par zéro amène le gestionnaire d'exception `divide_error()` à positionner le signal `SIGFPE`.

Lors de l'envoi d'un signal, le noyau exécute la routine du noyau `seng_sig_info()` qui positionne tout simplement à 1 le bit correspondant au signal reçu dans le champ `signal` du processus destinataire. La fonction se termine immédiatement dans les deux cas suivant :

- le numéro du signal émis est 0. Ce numéro n'étant pas un numéro de signal valable, le noyau retourne immédiatement en positionnant une erreur ;
- le processus destinataire est dans l'état zombie.

Le signal ainsi délivré mais pas encore pris en compte par le processus destinataire est qualifié de *signal pendant*.

Il est important de remarquer que le mécanisme de mémorisation de la réception du signal permet effectivement de mémoriser qu'un signal a été reçu mais en aucun cas, ne permet de mémoriser combien de signaux d'un même type ont été reçus.

6.2.2 Prise en compte d'un signal

La prise en compte d'un signal par un processus s'effectue lorsque celui-ci s'apprête à quitter le mode noyau pour repasser en mode utilisateur. Le signal est alors *délivré* au processus.

Cette prise en compte est réalisée par la routine du noyau `do_signal()` qui traite chacun des signaux pendants du processus. Trois types d'actions peuvent être réalisés :

- ignorer le signal ;
- exécuter l'action par défaut ;
- exécuter une fonction spécifique installée par le programmeur.

Ces actions sont stockées pour chaque signal dans le champ `gsig.sa_handler` du bloc de contrôle du processus. Ce champ prend la valeur `SIG_IGN` dans le premier cas, la valeur `SIG_DFL` dans le second cas et enfin contient l'adresse de la fonction spécifique à exécuter dans le troisième cas.

La fonction `do_signal()` ne traite pas les signaux bloqués par le processus.

a) Ignorer le signal

Lorsque le signal est ignoré, aucune action n'est entreprise au moment de sa prise en compte. Une exception existe cependant concernant le signal `SIGCHLD`. Lorsque ce signal est ignoré, le noyau force le processus à consulter les informations concernant ses fils zombies de manière à ce que leur prise en compte soit réalisée et que les blocs de contrôle associés soient détruits.

À titre indicatif, nous donnons le code correspondant dans la fonction `do_signal()`. La fonction `sys_wait4(-1, NULL, WNOHANG, NULL)` prend en compte n'importe quel

processus zombie du processus courant et retourne le pid du processus zombie pris en compte. L'option WNOHANG permet de ne pas rester bloqué lorsque tous les zombies ont été éliminés.

```
if (ka->sa.sa_handler == SIG_IGN) {
if (signr != SIGCHLD)
      continue;
     /* Check for SIGCHLD: it's special.  */
      while (sys_wait4(-1, NULL, WNOHANG, NULL) > 0)
            /* nothing */;
            continue;
```

b) Exécuter l'action par défaut

Le noyau associe à chaque signal une action par défaut à réaliser lors de la prise en compte de celui-ci. Ces actions par défaut sont de cinq natures différentes :

- abandon du processus ;
- abandon du processus et création d'un fichier core contenant son contexte d'exécution. Ce fichier core est exploitable par l'outil débogueur ;
- le signal est ignoré ;
- le processus est stoppé (passage dans l'état TASK_STOPPED) ;
- le processus reprend son exécution c'est-à-dire qu'il revient dans l'état TASK_RUNNING si il était stoppé.

Le tableau 6.2 indique l'action par défaut associée à chaque signal.

Tableau 6.2 – Actions par défaut

Actions par défaut	Nom du signal
Abandon	SIGHUP, SIGINT, SIGBUS, SIGKILL, SIGUSR1, SIGUSR2, SIGPIPE, SIGALRM, SIGTERM, SIGSTKFLT, SIGXCOU, SIGXFSZ, SIGVTALRM, SIGPROF, SIGIO, SIGPOLL, SIGPWR, SIGNUSED
Abandon avec ficher core	SIGQUIT, SIGILL, SIGTRAP, SIGABRT, SIGIOT, SIGFPE, SIGSEGV
Signal ignoré	SIGCHLD, SIGURG, SIGWINCH
Processus stoppé	SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU
Processus redémarré	SIGCONT

c) Exécuter une fonction spécifique

Tout processus peut installer un traitement spécifique pour chacun des signaux, hormis le signal SIGKILL. Ce traitement spécifique remplace alors le traitement par défaut défini par le noyau. Ce traitement spécifique peut être de deux natures différentes :

- il demande à ignorer le signal et prend alors la valeur SIG_IGN ;
- il définit une action particulière définie dans une fonction C attachée au code de l'utilisateur. C'est le *handler* ou gestionnaire du signal.

La prise en compte d'un gestionnaire de signal défini par le processus demande au noyau de manipuler le contexte de reprise de ce dernier. En effet, le processus doit revenir en mode utilisateur, non pas pour continuer son code là où il a été dérouté en mode noyau, mais pour aller exécuter la fonction associée au signal. Le processus reprendra son exécution à son point de déroutement vers le mode noyau, seulement après avoir achevé l'exécution du handler.

La figure 6.1 illustre les différentes étapes de la prise en compte du handler et de son exécution :

1. Le processus utilisateur passe en mode noyau par exemple suite à la réception de l'interruption horloge, après avoir exécuté l'instruction `x = x + 2`. Le contexte de reprise en mode utilisateur est sauvegardé sur la pile noyau ;
2. le processus exécute en mode noyau le gestionnaire de la routine d'interruption ;

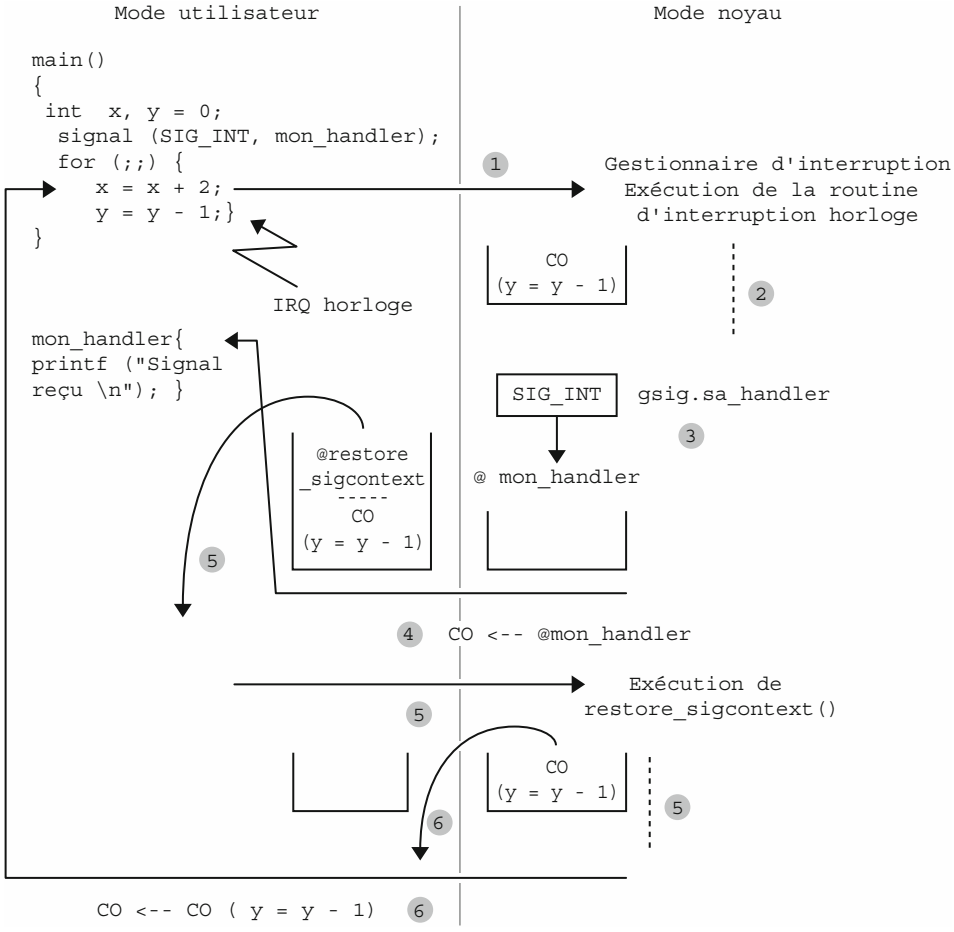

Figure 6.1 – Exécution d'un handler attaché au signal

3. le processus s'apprête à repasser en mode utilisateur et consulte l'ensemble des signaux qui sont pendants (fonction `do_signal()`). Le signal `SIGINT` est pendant et la fonction C `mon_handler()` lui est associée ;
4. le noyau manipule la pile utilisateur en y copiant d'une part le contenu de la pile noyau, d'autre part en y plaçant les paramètres d'appel d'une routine du système spécifique `restore_sigcontext()`. Le compteur ordinal est forcé avec l'adresse du handler ;
5. Le handler s'exécute. À la fin de cette exécution, le contexte de la pile utilisateur entraîne l'exécution par le processus de l'appel système `restore_sigcontext()`. Cette fonction restaure la pile noyau avec le contexte sauvegardé dans la pile utilisateur au point 4 ;
6. Le processus achève l'exécution de l'appel système `restore_sigcontext()`. Le contexte placé dans la pile noyau est restauré et le processus reprend son exécution juste après l'instruction `x = x + 2`.

6.2.3 Signaux et appels systèmes

Lorsqu'un processus exécute un appel système qui se révèle bloquant, le processus est placé par le noyau dans l'état `TASK_INTERRUPTIBLE` ou `TASK_UNINTERRUPTIBLE`.

Un processus placé dans l'état `TASK_INTERRUPTIBLE` est réveillé par le système lorsqu'il reçoit un signal. Le système place le processus dans l'état `TASK_RUNNING` sans terminer l'appel système et positionne la variable `errno` avec un code retour de l'appel système égal à `EINTR` pour signaler cette situation.

6.2.4 Signaux et héritage

Un processus fils n'hérite pas des signaux pendants de son père. En cas de recouvrement du code hérité du père, les gestionnaires par défaut sont réinstallés pour tous les signaux du fils.

6.3 Programmation des signaux

6.3.1 Envoyer un signal

Un processus envoie un signal à un autre processus en appelant la primitive `kill()` dont le prototype est :

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

L'interprétation du résultat de la fonction diffère en fonction de la valeur présente dans le champ `pid` :

- si `pid` est strictement positif, le signal `num_sig` est délivré au processus identifié par le pid `pid` ;

- si pid est nul, le signal num_sig est délivré à tous les processus du groupe auquel appartient le processus identifié par le pid pid ;
- si pid est négatif (en dehors de la valeur – 1), le signal est envoyé à tous les processus du groupe dont le PGID est égal à la valeur absolue de pid ;
- si pid est égal à – 1, le signal est envoyé à tous les processus du système sauf le processus 1 et le processus appelant.

En cas d'échec, la variable errno est positionnée avec l'une des valeurs suivantes :

- EPERM, le processus ne possède pas de droits suffisants pour envoyer le signal ;
- ESRCH, aucun processus ne correspond au processus désigné par l'argument pid ;
- EINVAL, num_sig n'est pas un numéro de signal valable.

La commande kill -numero_de_signal pid_de_processus constitue l'interface du même appel système pour l'interpréteur de commandes shell.

6.3.2 Bloquer les signaux

a) Manipuler un ensemble de signaux

Un ensemble de signaux est un ensemble du type prédéfini sigset_t. Ce type opaque ne peut être manipulé que par les primitives qui lui sont associées. Ce sont :

- int sigemptyset (sigset_t *ens_signaux) permet de vider l'ensemble de signaux ens_signaux ;
- int sigfillset (sigset_t *ens_signaux) permet de remplir l'ensemble de signaux ens_signaux avec tous les signaux du système ;
- int sigaddset (sigset_t *ens_signaux, int num_sig) permet d'ajouter le signal num_sig à l'ensemble de signaux ens_signaux ;
- int sigdelset (sigset_t *ens_signaux, int num_sig) permet de retrancher le signal num_sig à l'ensemble de signaux ens_signaux ;
- int sigismember (sigset_t *ens_signaux, int num_sig) permet de savoir si le signal num_sig appartient à l'ensemble de signaux ens_signaux.

Les quatre premières primitives renvoient 0 en cas de succès et – 1 sinon. La dernière primitive renvoie 1 si le signal sig_num est présent dans l'ensemble, 0 sinon et – 1 en cas d'erreur.

b) Bloquer les signaux

La primitive sigprocmask() permet à un processus de bloquer ou débloquer un ensemble de signaux hormis les signaux SIGKILL et SIGCONT. Cette primitive manipule le champ blocked dans le bloc de contrôle du processus. Le prototype de cette fonction est :

```
#include <signal.h>
int sigprocmask (int action, const sigset_t ens_signaux, sigset_t
*ancien);
```

Le paramètre `action` permet d'indiquer la nature de l'opération voulue. Ainsi :

- si `action = SIG_BLOCK`, l'ensemble des signaux bloqués est égal aux signaux déjà bloqués auxquels s'ajoutent les signaux placés dans l'ensemble `ens_signaux` ;
- si `action = SIG_UNBLOCK`, les signaux placés dans l'ensemble `ens_signaux` sont retirés de l'ensemble des signaux bloqués ;
- si `action = SIG_SETMASK`, l'ensemble des signaux bloqués devint égal à l'ensemble de signaux `ens_signaux`.

La primitive renvoie 0 en cas de succès et – 1 sinon. La variable `errno` prend l'une des deux valeurs suivantes :

- `EFAULT`, l'un des pointeurs n'est pas valide ;
- `EINVAL`, `action` n'est pas une méthode valide.

Par ailleurs, la primitive `sigpending()` permet de connaître l'ensemble des signaux pendants qui sont bloqués par le processus. Le prototype de la fonction est :

```
int sigpending (sigset_t *ens);
```

Enfin, la primitive `sigsuspend()` permet de façon atomique de modifier le masque des signaux et de se bloquer en attente. Une fois un signal non bloqué délivré, la primitive se termine en restaurant le masque antérieur.

```
int sigsuspend (const sigset_t *ens);
```

c) Un exemple d'utilisation

Dans l'exemple suivant, le processus bloque le signal `SIG_INT` durant son travail qui consiste en la réalisation d'un calcul numérique. À la fin de ce travail, le signal est débloqué. Comme le caractère « CTRL C » a été frappé au moins une fois au clavier durant l'exécution du calcul numérique, le signal `SIGINT` est pendant. Il est délivré suite à son déblocage et entraîne la terminaison du processus. Les traces d'exécution sont donc :

```
je vais faire mon travail tranquillement
SIGINT pendant
```

Si le caractère « CTRL C » n'a pas été frappé au clavier durant l'exécution du calcul numérique, les traces sont par contre :

```
je vais faire mon travail tranquillement
signaux débloques
```

```
/*********************************************************************/
/*                 Exemple de blocage de signaux                     */
/*********************************************************************/
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>

main()
{
```

```
char chaine[10];
int i,j;
sigset_t ens, ens1;

sigemptyset (&ens);
sigaddset (&ens, SIGINT);
sigprocmask(SIG_SETMASK, &ens, 0);

printf("je vais faire mon travail tranquillement\n");
i = 0;
while (i < 100000000) {
       j = i * 2;
       i = i + 1;
      }
sigpending (&ens1);
if (sigismember(&ens1, SIGINT))
   printf ("SIGINT pendant\n");
sigemptyset (&ens);
sigprocmask(SIG_SETMASK, &ens, 0);
printf("signaux débloqués\n");
}
```

6.3.3 Attacher un handler à un signal

Deux primitives différentes permettent d'attacher un gestionnaire de traitement à un signal. Ce sont :

- d'une part, l'appel système `signal()`. L'emploi de cette primitive est simple mais son utilisation peut poser des problèmes de compatibilité avec les différents systèmes Unix ;
- d'autre part, l'appel système `sigaction()`. D'un emploi plus complexe, elle présente un comportement fiable, normalisé par Posix.

a) La primitive signal

Description de la primitive

Le prototype de la primitive `signal()` est :

```
#include <signal.h>
void (*signal(int num_sig, void (*func)(int)))(int);
```

Le *handler* défini par `*func` est attaché au signal `sig`, désigné par le nom qui lui est associé. L'argument `func` peut pendre trois valeurs distinctes :

- `func = SIG_DFL`, alors l'action par défaut est attachée au signal `num_sig` ;
- `func = SIG_IGN`, alors le signal `num_sig` (qui ne peut pas être le signal `SIGKILL`) est ignoré ;
- `func` est un pointeur vers une fonction handler définie dans le code utilisateur. La prise en compte du signal `num_sig` par le processus entraîne l'exécution de cette

fonction. La fonction `func` accepte un argument de type entier qui est égal au numéro du signal provoquant son exécution. Elle ne renvoie aucune valeur.

En cas d'échec, la fonction `signal()` renvoie la valeur `SIG_ERR` définie dans le fichier `<signal.h>`. La variable `errno` est positionnée avec l'une des valeurs suivantes :

- `EFAULT`, le pointeur vers le gestionnaire de signal n'est pas valide ;
- `EINVAL, num_sig` n'est pas un numéro de signal valable ou il y a tentative d'installer un gestionnaire de signal pour les signaux `SIGKILL` et `SIGSTOP` ou d'ignorer les signaux `SIGKILL` et `SIGSTOP`.

Un exemple d'utilisation

Le programme suivant réalise une écriture sans fin en mémoire centrale, provoquant une violation mémoire. Lorsque celle-ci se produit, le signal `SIGSEGV` est levé et l'exécution du *handler* associé est réalisée. Ce *handler* affiche la valeur *k* ayant provoqué la violation :

```
reçu signal 11 pour k=910
```

```
/*********************************************************************/
/*         Exemple d'utilisation de la primitive signal()            */
/*********************************************************************/

#include <signal.h>
void segv();
int k,l;
char buf[1];

main()
{
/* mise en place traitement signal "violation"  */
if(signal(SIGSEGV,segv)==SIG_ERR)
   {
    perror("signal");
    exit(1);
     }

/* boucle devant mener à une violation de la segmentation mémoire */
for(k=0;;++k)
         l=buf[k];
}

/* procédure de traitement de la réception du signal */
void segv(sig)
int sig;
{
printf("reçu signal %d pour k=%x\n",sig,k);
exit(0);
}
```

b) La primitive sigaction

Description de la primitive

Le prototype de la primitive `sigaction()` est le suivant :

```
#include <signal.h>
int sigaction (int num_sig, const struct sigaction *nouv_action, const
struct sigaction *anc_action);
```

La structure `sigaction` a la forme suivante :

```
struct sigaction {
                  void (*sa_handler))();
                  sigset_t sa_mask;
                  int sa_flags;
                   };
```

Le handler `nouv_action->sa_handler` est installé comme gestionnaire du signal `num_sig`. Si ce gestionnaire n'est ni `SIG_DFL`, ni `SIG_IGN`, alors les signaux définis dans l'ensemble `nouv_action->sa_mask` ainsi que le signal `num_sig` sont bloqués durant l'exécution du handler. L'ancienne action associée au signal est sauvegardée dans la structure `anc_action`, sauf si ce paramètre est mis à NULL.

L'ensemble de signaux `sa_mask` définit donc les signaux à bloquer durant l'exécution du handler. Cet ensemble est construit et manipulé à l'aide des fonctions que nous avons détaillées au paragraphe « manipuler un ensemble de signaux ».

Le champ `sa_flags` définit plusieurs options permettant de configurer le comportement du handler. Une seule valeur, `SA_NOCLDSTOP`, est définie par POSIX et l'utilisation des autres valeurs nuit sensiblement à la portabilité des applications. Nous ne donnons donc ici que le rôle de la valeur `SA_NOCLDSTOP`, qui concerne le gestionnaire attaché au signal `SIGCHLD`. Cette option permet de ne pas déclencher l'envoi du signal `SIGCHLD` au processus père lorsque l'un de ces fils meurt.

En cas d'échec, la primitive `sigaction()` positionne la variable `errno` avec l'une des valeurs suivantes :

- `EINVAL`, `num_sig` n'est pas un numéro de signal valable ou il y a tentative d'installer un gestionnaire de signal pour les signaux `SIGKILL` et `SIGSTOP` ou d'ignorer les signaux `SIGKILL` et `SIGSTOP`.

Un exemple d'utilisation

Nous reprenons l'exemple précédent en remplaçant l'utilisation de la primitive `signal()` par celle de la primitive `sigaction()`. Par ailleurs le processus effectuant la violation mémoire crée un fils, qui effectue pour sa part un calcul numérique. Ce fils est tué par son père dans le gestionnaire de signal levé par la violation mémoire.

```
/*****************************************************************/
/*        Exemple d'utilisation de la primitive sigaction()      */
/*****************************************************************/

#include <signal.h>
void segv();
int k,l, pid;
```

```
char buf[1];

main()
{
struct sigaction action;
int i,j;
/* mise en place traitement signal "violation"  */
sigemptyset(&action.sa_mask);
action.sa_flags = 0;
action.sa_handler = segv;
sigaction(SIGSEGV, &action, 0);

pid = fork();
if (pid == 0)
{
printf("je suis le fils\n");
i = 0;
while (i < 10000) {
       j = i * 2;
       i = i + 1;
      }
exit(0);
}
else

/* boucle devant mener à une violation de la segmentation mémoire */
for(k=0;;++k)
         l=buf[k];

}

/* procédure de traitement de la réception du signal */
void segv(sig)
int sig;
{
printf("reçu signal %d pour k=%x\n",sig,k);
kill (SIGKILL, pid);
exit();
}
```

6.3.4 Traiter les appels systèmes interrompus

Un processus placé dans l'état `TASK_INTERRUPTIBLE` suite à un appel système bloquant est réveillé par le système lorsqu'il reçoit un signal. Le système place le processus dans l'état `TASK_RUNNING` sans terminer l'appel système et positionne la variable `errno` avec un code retour de l'appel système égal à `EINTR` pour signaler cette situation. Deux solutions sont possibles pour remédier à ce problème :

- relancer « manuellement » l'exécution de l'appel système lorsque celui-ci échoue avec un code retour égal à `EINTR`. Ceci donne le code suivant :

```
        do {
        nb_lus = read(descripteur, buffer, nb_à_lire) }
        while  ((nb_lus == -1) && (errno == EINTR));
```

- utiliser la fonction `int siginterrupt (int sig_num, int interrupt)`. Cette fonction est appelée après l'installation du gestionnaire de signal associé au signal `sig_num`. Si l'indicateur `interrupt` est nul, alors l'appel système interrompu par le signal `sig_num` est relancé automatiquement. Si l'indicateur est non nul, l'appel système échoue en positionnant `errno` à la valeur `EINTR`.

6.3.5 Attendre un signal

La primitive `pause()` permet à un processus de se mettre en attente de la délivrance d'un signal. Son prototype est :

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

Cette primitive ne permet pas de spécifier ni l'attente d'un signal particulier, ni de savoir quel signal a réveillé le processus. Elle renvoie toujours la valeur – 1 en positionnant `errno` à la valeur `EINTR`.

Dans l'exemple suivant, un processus s'endort jusqu'à recevoir le signal `SIGINT` (frappe de « CTRL C » au clavier). Le processus se réveille, exécute le handler associé au signal puis commence un traitement consistant à écrire une chaîne lue au clavier dans un fichier.

```
/*********************************************************************/
/*         Exemple d'utilisation de la primitive pause()             */
/*********************************************************************/

#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>

void fin_attente();
main()
{
char chaine[10];
int fp;
/* processus arme le signal et attend le signal */
if(signal(SIGINT,fin_attente)==SIG_ERR)
   {
    perror("signal");
    exit(1);
    }
/* attente */
pause();
printf("je vais faire mon travail \n");
printf("Donnez moi la chaîne attendue \n");
scanf ("%s", chaine);
```

```
fp = open("fichier", O_WRONLY);
write (fp, chaine, strlen(chaine));
close(fp);
exit(0);
}

void fin_attente()
{
printf("signal reçu, arrêt de la pause\n");
}
```

6.3.6 Armer une temporisation

La primitive `alarm()` permet à un processus d'armer une temporisation. À l'issue de cette temporisation le signal `SIGALRM` est délivré au processus. Le comportement par défaut est d'arrêter l'exécution du processus. Le prototype de la fonction est :

```
#include <unistd.h>
unsigned int alarm (unsigned int nb_sec);
```

Une temporisation d'une durée égale à `nb_sec` secondes est armée. L'opération `alarm(0)` annule une temporisation précédemment armée.

Une utilisation courante de la fonction `alarm()` consiste à armer une temporisation avant la réalisation d'un appel système bloquant. Le délai écoulé, la levée du signal `SIGALRM` fait échouer l'appel système bloquant qui retourne un code erreur égal à `EINTR`.

À titre d'exemple, nous reprenons le programme illustrant le fonctionnement de la fonction `pause()`. Une alarme de 5 secondes est à présent armée avant de demander la lecture au clavier d'une chaîne de caractères. Si la chaîne n'a pas pu être lue au bout de ces 5 secondes, le signal `SIGALRM` est délivré au processus qui se termine en fermant le descripteur du fichier ouvert.

Les traces d'exécution sont, suite de la frappe de la séquence « CTRL C » au clavier :

```
signal reçu, arrêt de la pause
je vais faire mon travail
Donnez moi la chaîne attendue
trop tard, je n'attends plus
```

```
/*********************************************************************/
/*          Exemple d'utilisation de la primitive alarm()            */
/*********************************************************************/

#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>

void fin_attente();
int fp;
```

```
main()
{
char chaine[10];

/* processus arme le signal et attend le signal */
if(signal(SIGINT,fin_attente)==SIG_ERR)
   {
    perror("signal");
    exit(1);
    }
if (signal(SIGALRM, fin_attente)==SIG_ERR)
  {
    perror("signal");
    exit(1);
    }
/* attente */
pause();
printf("je vais faire mon travail \n");
fp = open("fichier", O_WRONLY);
alarm(5);
printf("Donnez moi la chaîne attendue \n");
scanf ("%s", chaine);
alarm(0);
write (fp, chaine, strlen(chaine));
close(fp);
exit(0);
}

void fin_attente(int num_sig)
{
if (num_sig == SIGINT)
   printf("signal reçu, arrêt de la pause\n");
else
{
  printf("trop tard, je n'attends plus");
  close(fp);
  exit(1);
}
}
```

6.4 Signaux temps réel

6.4.1 Présentation générale

En plus des signaux classiques, Linux offre des signaux qualifiés de signaux temps réel qui respectent la norme POSIX.1b. Ces signaux temps réel sont au nombre de 32. Contrairement aux signaux classiques, aucun nom ne leur est associé et les signaux temps réels sont désignés par un numéro allant de 32 à 63. Il est préférable,

plutôt que d'utiliser directement les numéros, de désigner les signaux par rapport à leur rang effectif entre la valeur minimale `SIGRTMIN` et la valeur maximale `SIGRTMAX`. Les caractéristiques principales de ces signaux temps réel sont :

- empilement des occurrences de chaque signal. Comme nous l'avons vu précédemment, la réception d'un signal classique par un processus est gérée à l'aide d'un seul bit par signal qui est mis à 1 si le signal a été reçu et à 0 sinon. Cette manière de faire ne permet évidemment pas de garder la mémoire de plusieurs occurrences successives d'un même signal. Aussi, un signal reçu 2 fois ne sera délivré qu'une seule fois au processus. Linux associe à chaque signal temps réel une file d'attente qui lui permet au contraire de mémoriser l'ensemble de toutes les occurrences. Chaque occurrence présente dans la file d'attente donne lieu à une délivrance spécifique. Le système permet d'empiler jusqu'à 1 024 signaux ;
- lorsque plusieurs signaux temps réel doivent être délivrés à un processus, le noyau délivre toujours les signaux temps réel de plus petit numéro avant les signaux temps réel de plus grand numéro. Ceci permet d'attribuer un ordre de priorité entre les signaux temps réel ;
- le signal temps réel, contrairement au signal classique, peut transporter avec lui une petite quantité d'informations, délivrées au handler qui lui est attaché.

6.4.2 Envoyer un signal temps réel

L'envoi d'un signal temps réel peut être réalisé avec la fonction `kill()`, mais cette manière de procéder ne permet pas de délivrer d'informations complémentaires au numéro du signal reçu.

Pour pouvoir délivrer des informations complémentaires au signal, il faut utiliser la primitive `sigqueue()` dont le prototype est :

```
int sigqueue (pid_t pid, int num_sig, const union sigval info);
```

L'argument `pid` désigne le processus concerné et l'argument `num_sig` identifie le signal envoyé. Le troisième argument `info` contient l'information complémentaire associée au signal. C'est une valeur de type `union sigval` qui peut prendre deux formes :

- un entier `int` si le membre `sival_int` de l'union est utilisé ;
- un pointeur `void*` si le membre `sival_ptr` de l'union est utilisé.

6.4.3 Attacher un gestionnaire à un signal temps réel

Du fait de cette transmission d'informations complémentaires associées au signal temps réel, l'attachement et la définition d'un gestionnaire de signal s'effectuent différemment que dans le cadre d'un signal classique. Le gestionnaire de signal pour un signal temps réel prend la forme suivante :

```
void gestionnaire (int num_sig, struct siginfo *info, void *rien);
```

La structure de type `struct siginfo` contient notamment les champs suivants, conformes à la norme POSIX[1]:

- `int si_signo`, le numéro du signal ;
- `int si_code` est une information dépendant du signal. Pour les signaux temps réel, cette information indique l'origine du signal ;
- `int si_value.sival_int` correspond au champ `sival_int` dans l'appel système `sigqueue()` ;
- `void *si_value.sival_ptr` correspond au champ `sival_ptr` dans l'appel système `sigqueue()`.

Le premier paramètre correspond au numéro du signal reçu. Le troisième paramètre n'est pas normalisé par le standard POSIX et n'est pas utilisé.

L'attachement de ce gestionnaire de signal à un signal s'effectue en utilisant la primitive `sigaction()` décrite au paragraphe 6.3.3, mais le gestionnaire n'est pas renseigné dans le champ `sa_handler` de la structure mais dans le champ `sa_sigaction` de celle-ci. En effet, comme le montre le détail de la structure qui suit, le premier champ de celle-ci est en fait construit comme une union du champ `sa_handler` et du champ `sa_sigaction`.

```
struct sigaction {
 union {
   sighandler_t _sa_handler;
   void (*_sa_sigaction)(int, struct siginfo *, void *);
 } _u;
 sigset_t sa_mask;
 unsigned long sa_flags;
};
```

Par ailleurs, le champ `sa_flags` doit prendre la valeur `SA_SIGINFO`.

Dans l'exemple suivant, un processus père crée un fils. Ce processus fils attache un gestionnaire de signal au signal temps réel `SIGRTMIN` + 1. Le père prend un nombre depuis le clavier et envoie ce nombre à son fils en même temps que le signal temps réel `SIGRTMIN` + 1. Le gestionnaire de signal attaché au fils affiche le numéro du signal reçu ainsi que la valeur transmise par le père. Les traces d'exécution sont par exemple :

```
Je vais faire mon travail
Donnez moi l'entier attendu
44
signal temps reel reçu, 33, 33
entier reçu 44
/*****************************************************************/
/*          Exemple d'utilisation des signaux temps réel         */
/*****************************************************************/
```

1. La structure comporte d'autres champs qui ne sont pas conformes à la norme; aussi nous ne les décrirons pas.

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

void gestion_tempsreel();
main()
{
union sigval valeur;
struct sigaction actionTR;
int pid;

pid = fork();
if (pid == 0)
{
   /* attachement du gestionnaire au signal temps reel */
  actionTR.sa_sigaction = gestion_tempsreel;
  sigemptyset(&actionTR.sa_mask);
  actionTR.sa_flags = SA_SIGINFO;
  sigaction(SIGRTMIN+1, &actionTR, NULL);
  pause();
  exit(0);}
else
{
  printf("Je vais faire mon travail \n");
  printf("Donnez moi l'entier attendu \n");
  scanf ("%d", &valeur.sival_int);
  printf ("%d", valeur.sival_int);
  sigqueue(pid,SIGRTMIN+1,valeur);
  wait();
  exit(0);
}
}

void gestion_tempsreel(int numero, struct siginfo *info, void *rien)
{
printf ("signal temps reel reçu, %d, %d\n", numero, info ->si_signo);
printf ("entier reçu %d\n", info->si_value.sival_int);
}
```

6.4.4 Exécution du gestionnaire de signal

L'exécution d'un gestionnaire de signal telle qu'elle est décrite au paragraphe 6.2.2 nécessite plusieurs commutations de contexte, une pour aller exécuter le gestionnaire dans le code utilisateur puis une seconde pour revenir dans le noyau et achever la routine système prenant en compte les signaux.

Dans le cadre du temps réel, ces commutations de contexte peuvent être pénalisantes en terme de temps.

Le système Linux offre deux primitives permettant à un processus de simplement attendre un signal temps réel, sans que le noyau active le gestionnaire du signal. Ce sont les deux primitives `sigwaitinfo()` et `sigtimedwait()` dont nous donnons les prototypes :

```
int sigwaitinfo(const sigset *signaux, struct siginfo *info);
int sigtimedwait(const sigset *signaux, struct siginfo *info, const struct
timespec *delai);
```

L'ensemble `const sigset *signaux` correspond à l'ensemble des signaux attendus par le processus. Cet ensemble est construit à l'aide des primitives que nous avons vues au paragraphe 6.3.2. La structure `struct siginfo *info` retourne les informations énumérées au paragraphe précédent.

Lorsqu'un processus effectue un appel à la primitive `sigwaitinfo()`, il reste bloqué jusqu'à ce qu'au moins un signal de l'ensemble lui ait été délivré. La primitive se termine simplement en retournant le numéro du signal reçu et les informations associées éventuellement. Le processus peut alors invoquer le gestionnaire de signal associé au signal reçu comme un simple appel de procédure, ce qui est beaucoup moins coûteux en terme de commutations de contexte.

La fonction `sigtimedwait()` constitue tout simplement une version temporisée de la fonction `sigwaitinfo()`. La structure `struct timespec` a la forme suivante :

```
struct timespec {
               long tv_sec;  /* secondes */
               long tv_nsec;  / nanosecondes */
                };
```

Nous reprenons l'exemple précédent. Cette fois, le fils se met en attente du signal temps réel `SIGRTMIN + 1` à l'aide de la primitive `sigwaitinfo()`. Puis il appelle le gestionnaire de signal comme un simple appel de fonction.

```
/*****************************************************************/
/*    Invocation du handler comme un simple appel de procedure    */
/*****************************************************************/

#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>

void gestion_tempsreel();
main()
{
union sigval valeur;
struct siginfo information;
int pid;
sigset_t ensemble;

pid = fork();
if (pid == 0)
{
```

```
  sigemptyset(&ensemble);
  sigaddset(&ensemble, SIGRTMIN+1);
  sigwaitinfo(&ensemble, &information);
  gestion_tempsreel(SIGRTMIN+1, &information, NULL);
  exit(0);}
else
{
  printf("je vais faire mon travail \n");
  printf("Donnez moi l'entier attendu \n");
  scanf ("%d", &valeur.sival_int);
  printf ("%d", valeur.sival_int);
  sigqueue(pid,SIGRTMIN+1,valeur);
  wait();
  exit(0);
}
}

void gestion_tempsreel(int numero, struct siginfo *info, void *rien)
{
printf ("signal temps reel reçu, %d, %d\n", numero, info ->si_signo);
printf ("entier reçu %d\n", info->si_value.sival_int);
}
```

Exercices

6.1 Qu'avez-vous retenu ?

Pour chacune des questions suivantes, choisissez la ou les réponses qui vous semblent exactes.

Question 1 – Un signal :

- ☐ a. est un événement envoyé au processeur par un dispositif matériel externe.
- ☐ b. est un outil pour délivrer l'occurrence d'un événement à un processus.
- ☐ c. est un outil qui permet à des processus de partager des données.

Question 2 – Dans les propositions énoncées, lesquelles sont vraies ?

- ☐ a. le signal temps réel est délivré immédiatement.
- ☐ b. les occurrences de signaux temps réel sont empilées et chaque occurrence est délivrée.
- ☐ c. le *handler* du signal temps réel est associé à une date de fin d'exécution butoir.

Question 3 – Un signal pendant est :

- ☐ a. un signal délivré au processus.
- ☐ b. un signal délivré au processus et bloqué.
- ☐ c. un signal délivré au processus en attente de prise en compte.

Question 4 – Un signal bloqué est :

- ☐ a. un signal ignoré.
- ☐ b. un signal qui n'est pas délivré au processus.
- ☐ c. un signal délivré au processus mais dont la prise en compte est retardée.

Question 5 – Ignorer un signal signifie :

- ☐ a. pour tous les signaux, ne rien faire.
- ☐ b. pour tous les signaux sauf le signal `SIGCHLD`, ne rien faire.

Question 6 – Un processus père qui ignore le signal `SIGCHLD` :

- ☐ a. ne prend pas en compte les morts de ses fils.
- ☐ b. est obligé par le noyau à prendre en compte les morts de ses fils en dehors d'un appel système à une primitive `wait()`.

6.2 Retour du mode noyau

En utilisant ce que vous avez appris au chapitre 2 et dans ce chapitre, donnez les actions importantes réalisées par le noyau lorsqu'il s'apprête à replacer un processus en mode utilisateur.

6.3 Programmation des signaux

Écrivez un programme dans lequel un processus ouvre un fichier sur lequel il travaille (par exemple il écrit 10 fois la chaîne « `abcdefghijgl` » dans le fichier). En cas de réception du signal `SIGINT`, le processus ferme le fichier avant de se terminer.

6.4 Programmation des signaux

Écrivez un programme dans lequel un processus crée un fils puis se met en attente de la fin de son fils. Le fils exécute un code qui boucle. Au bout de 10 secondes, le fils n'étant pas achevé, le père tue son fils.

6.5 Empilement des occurrences de signaux temps réel

Le programme donné au paragraphe 6.4.3 est modifié comme suit :

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

void fin_attente();
void gestion_tempsreel();
main()
{
union sigval valeur;
struct sigaction actionTR;
int pid, i;

pid = fork();
if (pid == 0)
{
/* attachement du gestionnaire au signal temps reel */
```

```
actionTR.sa_sigaction = gestion_tempsreel;
sigemptyset(&actionTR.sa_mask);
actionTR.sa_flags = SA_SIGINFO;
sigaction(SIGRTMIN+1, &actionTR, NULL);
pause();
exit(0);}
else
{

/* attente */

printf("je vais faire mon travail \n");
printf("Donnez moi l'entier attendu \n");
scanf ("%d", &valeur.sival_int);
printf ("%d", valeur.sival_int);
i = 0;
while (i < 10) {
sigqueue(pid,SIGRTMIN+1,valeur);
i = i + 1;}
wait();
exit(0);
}
}

void gestion_tempsreel(int numero, struct siginfo *info, void *rien)
{
printf ("signal temps réel reçu, %d, %d\n", numero, info ->si_signo);
printf ("entier reçu %d\n", info->si_value.sival_int);
}
```

Quelles traces génèrent l'exécution de ce programme. Pourquoi ?

Des idées pour vous exercer à la programmation

Programme 1 – Soit l'application suivante : un processus lit périodiquement une valeur de température qui lui est fournie au clavier. Il enregistre cette valeur dans un fichier puis la transmet à un autre processus en accompagnement d'un signal temps réel si elle est inférieure à la valeur 15 °C ou si elle excède la valeur 20 °C. Le processus récepteur du signal temps réel affiche un message d'alarme indiquant une température trop basse ou une température trop haute.

Programme 2 – Soit l'application suivante : un processus crée deux processus fils qui exécute chacun un travail (par exemple le premier fils exécute indéfiniment l'addition de deux nombres fournis par l'utilisateur et le second fils transforme indéfiniment en minuscules une chaîne de caractères fournie par l'utilisateur). Lorsque le processus père reçoit le signal `SIGINT`, il tue chacun de ses fils avant de se terminer lui-même. Les processus fils affichent toutes les valeurs calculées ou les conversions de chaînes effectuées depuis le début de leur exécution avant de se terminer.

Solutions

6.1 Qu'avez-vous retenu ?

Question 1 – Un signal :

- ☐ b. est un outil pour délivrer l'occurrence d'un événement à un processus.

Question 2 – Dans les propositions énoncées, lesquelles sont vraies ?

- ☐ b. les occurrences de signaux temps réel sont empilées et chaque occurrence est délivrée.

Question 3 – Un signal pendant est :

- ☐ c. un signal délivré au processus en attente de prise en compte.

Question 4 – Un signal bloqué est :

- ☐ c. un signal délivré au processus mais dont la prise en compte est retardée.

Question 5 – Ignorer un signal signifie :

- ☐ b. pour tous les signaux sauf le signal `SIGCHLD`, ne rien faire.

Question 6 – Un processus père qui ignore le signal `SIGCHLD` :

- ☐ b. est obligé par le noyau à prendre en compte les morts de ses fils en dehors d'un appel système à une primitive `wait()`.

6.2 Retour du mode noyau

Les actions suivantes sont entreprises par le noyau lorsqu'il s'apprête à basculer un processus en mode utilisateur :

- exécution des parties basses activées par les routines d'interruptions exécutées (*cf.* chapitre 2) ;
- si le drapeau demandant un réordonnancement est positionné, alors appel à l'ordonnanceur qui choisit un nouveau processus et effectue la commutation de contexte pour le rendre actif ;
- exécution des gestionnaires de signaux attachés aux signaux pendants non bloqués pour le processus actif ;
- basculement en mode utilisateur et reprise du code du processus.

6.3 Programmation des signaux

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

int desc;

main()
{
int i;
```

```
extern onintr();    /* Handler */
desc = open("toto", O_RDWR, 0);
signal (SIGINT, onintr);
i = 0;
while (i < 10)
{
  write (desc, "abcdefghijgl", 12);
  i = i + 1;
}
printf("fin process normal\n");
close (desc);
}

onintr ()
{
printf("Handler onintr\n");
close(desc);
exit (1);
}
```

6.4 Programmation des signaux

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

pid_t pid;

main()
{
int status;
extern onalarm();    /* Handler */

pid = fork();
if (pid == -1)
   printf ("erreur création de processus");
else
if (pid == 0)
{
  for(;;)
       printf("je boucle !!!! \n");
}
else
{
  signal (SIGALRM, onalarm);
  alarm(5);
  wait (&status);
}
}
```

```
onalarm ()
{
printf("Handler onalarm\n");
kill (pid, SIGKILL);
exit (1);
}
```

6.5 Empilement des occurrences de signaux temps réel

Le processus père envoie 10 fois de suite le signal temps réel `SIGRTMIN` + 1 (33) à son fils. Les occurrences de signaux temps réel étant mémorisées à la différence de ce qui est fait pour les signaux classiques, le fils exécute donc 10 fois le gestionnaire de signal `gestion_tempsreel()` et affiche 10 fois le numéro du signal reçu et la valeur de l'entier reçu.

7 COMMUNICATION ENTRE PROCESSUS

PLAN

OBJECTIFS

- Dans un système multiprogrammé, chaque processus dispose d'un espace d'adressage propre et indépendant, protégé par rapport aux autres processus. Malgré tout, les processus peuvent avoir besoin de communiquer entre eux pour échanger des données. Cet échange de données peut se faire par le biais d'une zone de mémoire partagée, par le biais d'un fichier ou encore en utilisant les outils de communication offerts par le système d'exploitation.
- Le système Linux offre différents outils de communication aux processus utilisateurs. Ce sont principalement les *tubes anonymes* encore appelés *pipe*, les *tubes nommés*, les *files de messages* ou *MSQ* et encore la *mémoire partagée*. Les deux premiers outils appartiennent au système de gestion de fichiers tandis que les deux autres font partie de la famille des IPC (*Inter Processus Communication*).
- Ce chapitre les présente.

7.1 LA COMMUNICATION PAR TUBES

Le système Linux propose deux types d'outils « tubes » : les *tubes anonymes* et les *tubes nommés*.

Un tube est un tuyau dans lequel un processus peut écrire des données qu'un autre processus peut lire. La communication dans le tube est unidirectionnelle et une fois le sens d'utilisation du tube choisi, celui-ci ne peut plus être changé. En d'autres termes un processus lecteur du tube ne peut devenir écrivain dans ce tube et vice-versa.

Les tubes sont gérés par le système au niveau du système de gestion de fichiers et correspondent à un fichier au sein de celui-ci. Lors de la création d'un tube, deux descripteurs sont créés, permettant respectivement de lire et écrire dans le tube.

Les données dans le tube sont gérées en flots d'octets, sans préservation de la structure des messages déposés dans le tube, selon une politique de type « Premier entré, Premier servi », c'est-à-dire que le processus lecteur reçoit les données les plus anciennement écrites. Par ailleurs, les lectures sont destructives c'est-à-dire que les données lues par un processus disparaissent du tube.

Le tube a une capacité finie qui est celle du tampon qui lui est alloué. Cette capacité est définie par la constante `PIPE_BUF` dans le fichier `<limits.h>`. Un tube peut donc être plein et amener de ce fait les processus écrivains à s'endormir en attendant de pouvoir réaliser leur écriture.

7.1.1 Les tubes anonymes

Le tube anonyme est géré par le système au niveau du système de gestion de fichiers et correspond à un fichier au sein de celui-ci, mais un fichier sans nom. Du fait de cette absence de nom, le tube ne peut être manipulé que par les processus ayant connaissance des deux descripteurs en lecture et en écriture qui lui sont associés. Ce sont donc le processus créateur du tube et tous les descendants de celui-ci créés après la création du tube et qui prennent connaissance des descripteurs du tube par héritage des données de leur père.

a) Création d'un tube anonyme

Un tube anonyme est créé par la primitive `pipe()` dont le prototype est :

```
#include <unistd.h>
int pipe (int desc[2]);
```

La primitive retourne deux descripteurs placés dans le tableau `desc`. `desc[0]` correspond au descripteur utilisé pour la lecture dans le tube tandis que `desc[1]` correspond au descripteur pour l'écriture dans le tube.

Les deux descripteurs sont alloués dans la table des fichiers ouverts du processus et pointent respectivement sur un objet fichier en lecture et un objet fichier en écriture. Le tube est représenté au sein du système par un objet inode auquel n'est associé aucun bloc de données, les données transitant dans le tube étant placées dans un tampon alloué dans une case de la mémoire centrale.

Tout processus ayant connaissance du descripteur `desc[0]` peut lire depuis le tube. De même tout processus ayant connaissance du descripteur `desc[1]` peut écrire dans le tube. (figure 7.1).

En cas d'échec la primitive `pipe()` renvoie la valeur 0 et la variable `errno` est positionnée avec les valeurs suivantes :

- `EFAULT`, le tableau `desc` passé en paramètre n'est pas valide ;
- `EMFILE`, le nombre maximal de fichiers ouverts par le processus a été atteint ;
- `ENFILE`, le nombre maximal de fichiers ouverts par le système a été atteint.

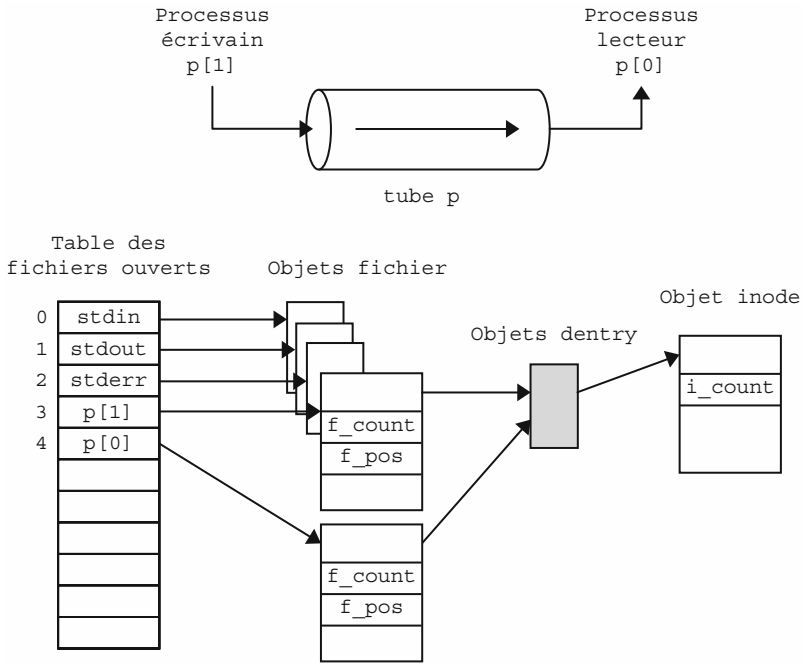

Figure 7.1 – Communication par tube anonyme

b) Fermeture d'un tube anonyme

Un tube anonyme est considéré comme étant fermé lorsque tous les descripteurs en lecture et en écriture existants sur ce tube sont fermés. Un processus ferme un descripteur de tube `fd` en utilisant la primitive `close()` :

```
int close(int fd);
```

À un instant donné, le nombre de descripteurs ouverts en lecture détermine le nombre de lecteurs existants pour le tube. De même, le nombre de descripteurs ouverts en écriture détermine le nombre d'écrivains existants pour le tube. Un descripteur fermé ne permet plus d'accéder au tube et ne peut pas être régénéré.

c) Lecture dans un tube anonyme

La lecture dans un tube anonyme s'effectue par le biais de la primitive `read()` dont le prototype est :

```
int read(int desc[0], char *buf, int nb);
```

La primitive permet la lecture de `nb` caractères depuis le tube `desc`, qui sont placés dans le tampon `buf`. Elle retourne en résultat le nombre de caractères réellement lus. L'opération de lecture répond à la sémantique suivante :

- si le tube n'est pas vide et contient `taille` caractères, la primitive extrait du tube `min(taille, nb)` caractères qui sont lus et placés à l'adresse `buf` ;

- si le tube est vide et que le nombre d'écrivains est non nul, la lecture est bloquante. Le processus est mis en sommeil jusqu'à ce que le tube ne soit plus vide ;
- si le tube est vide et que le nombre d'écrivains est nul, la fin de fichier est atteinte. Le nombre de caractères rendu est nul.

L'opération de lecture sur le tube peut être rendue non bloquante en émettant un appel à la fonction `fcntl(desc[0], F_SETFL, O_NONBLOCK)`. Dans ce cas, le retour est immédiat dans le cas où le tube est vide.

d) Écriture dans un tube anonyme

L'écriture dans un tube anonyme s'effectue par le biais de la primitive `write()` dont le prototype est :

```
int write(int desc[1], char *buf, int nb);
```

La primitive permet l'écriture de `nb` caractères placés dans le tampon `buf` dans le tube `desc`. Elle retourne en résultat le nombre de caractères réellement écrits. L'opération d'écriture répond à la sémantique suivante :

- si le nombre de lecteurs dans le tube est nul, alors une erreur est générée et le signal `SIGPIPE` est délivré au processus écrivain, et le processus se termine. L'interpréteur de commandes shell affiche par défaut le message « `Broken pipe` » ;
- si le nombre de lecteurs dans le tube est non nul, l'opération d'écriture est bloquante jusqu'à ce que les `nb` caractères aient effectivement été écrits dans le tube. Dans le cas où le nombre `nb` de caractères à écrire est inférieur à la constante `PIPE_BUF` (4 096 octets), l'écriture des `nb` caractères est atomique, c'est-à-dire que les `nb` caractères sont écrits les uns à la suite des autres dans le tube. Si le nombre de caractères est supérieur à `PIPE_BUF`, la chaîne de caractères à écrire peut au contraire être arbitrairement découpée par le système.

  De même que la lecture, l'écriture sur le tube peut être rendue non bloquante.

e) Exemples d'utilisation

Un exemple de communication unidirectionnelle

Nous donnons un premier exemple de communication entre processus par l'intermédiaire d'un tube. Dans cet exemple, le processus fils écrit à destination de son père la chaîne de caractères « bonjour ». Les étapes du programme sont les suivantes :

1. Le processus père ouvre un tube en utilisant la fonction `pipe()`.
2. Le processus père crée un fils en utilisant la fonction `fork()`.
3. Les descripteurs en lecture et écriture du tube sont utilisables par les 2 processus. Chacun des deux processus ferme le descripteur qui lui est inutile : ainsi, le processus père ferme le descripteur en écriture et le processus fils ferme le descripteur en lecture.
4. Le fils envoie un message à son père.
5. Le père lit le message.

```
/*********************************************************************/
/*   Communication unidirectionnelle entre un père et son fils    */
/*********************************************************************/
main () {
int pip[2], status;
pid_t retour;
char chaine[7];


pipe(pip);
retour = fork();
if (retour == 0)
{ /* le fils écrit dans le tube */
        close (pip[0]); /* pas de lecture sur le tube */
        write (pip[1], "bonjour", 7);
        close (pip[1]); exit(0);
}
else
{ /* le père lit le tube */
        close (pip[1]); /* pas d'écriture sur le tube */
        read(pip[0], chaine, 7);
        close (pip[0]);
        wait(&status);
}
```

Un exemple de communication bidirectionnelle

L'exemple suivant illustre une communication bidirectionnelle. Comme le tube est un outil de communication unidirectionnelle, la réalisation d'une communication bidirectionnelle nécessite l'utilisation de deux tubes, chaque tube étant utilisé dans le sens inverse de l'autre.

```
/*********************************************************************/
/*    Communication bidirectionnelle entre un père et son fils    */
/*********************************************************************/
#include <stdio.h>

int pip1[2];        /*  descripteurs pipe 1  */
int pip2[2];        /*  descripteurs pipe 2  */
int status;

main()
{
int idfils;
char rep[7], mesg[5];

/* ouverture tubes */
if(pipe(pip1))
  {
  perror("pipe 1");
  exit(1);
  }
```

```
if(pipe(pip2))
  {
  perror("pipe 2");
  exit(2);
  }
/* création processus */
if((idfils=fork())==-1)
  {
  perror("fork");
  exit(3);
  }
if(idfils) {
         /*le premier tube sert dans le sens père vers fils
           il est fermé en lecture */
        close(pip1[0]);
        /*le second tube sert dans le sens fils vers père
           il est fermé en écriture*/
        close(pip2[1]);
        /* on envoie un message au fils par le tube 1*/
        if(write(pip1[1],"hello",5)!=5)
          {
            fprintf(stderr,"père: erreur en écriture\n");
            exit(4);
          }
         /* on attend la réponse du fils par le tube 2 */
         if(read(pip2[0],rep,7)!=7)
            {
             fprintf(stderr,"fils: erreur lecture\n");
             exit(5);
            }
         printf("message du fils: %s\n",rep);
         wait(&status);
else   {
         /*fermeture du tube 1 en écriture */
         close(pip1[1]);
         /* fermeture du tube 2 en lecture */
         close(pip2[0]);
         /* attente d'un message du père */
         if(read(pip1[0],mesg,5)!=5)
         {
             fprintf(stderr,"fils: erreur lecture\n");
             exit(6);
          }
         printf("la chaine reçue par le fils est: %s\n",mesg);
         /* envoi d'un message au père */
         if(write(pip2[1],"bonjour",7)!=7)
         {
            fprintf(stderr,"fils: erreur ecriture\n");
            exit(7);
         }
         exit(0)
       }
}
```

Les tubes et le shell : primitive dup

Les tubes de communication peuvent être également utilisés au niveau de l'interpréteur de commandes shell pour transmettre le résultat d'une commande à une autre commande qui interprète ces données comme des données d'entrées. Au niveau du langage de commande shell, le tube de communication est symbolisé par le caractère « | ».

Par exemple, la commande `ls -l | wc -l` transmet les résultats de la commande `ls -l` à la commande `wc -l`. Comme la commande `ls -l` délivre les caractéristiques des fichiers du répertoire courant, à raison d'une ligne par fichier, et que la commande `wc -l` compte les lignes de l'entrée qui lui est fournie, le résultat de la commande `ls -l | wc -l` consiste dans le nombre de fichiers présents dans le répertoire courant.

Dans ce nouvel exemple, nous souhaitons écrire le programme composé de deux processus qui correspond à l'exécution de cette commande `ls -l | wc -l`. Nous choisissons de faire exécuter la commande `ls -l` par le processus fils et la commande `wc -l` par le processus père.

La difficulté du problème réside dans la nécessaire redirection de la sortie de la commande `ls -l` et de l'entrée de la commande `wc -l`, respectivement sur l'entrée du tube et sur la sortie de celui-ci. En effet, pour la commande `ls -l`, la destination naturelle des informations est la sortie standard `stdout`. La commande `wc -l` prend naturellement ses données depuis l'entrée standard `stdin`.

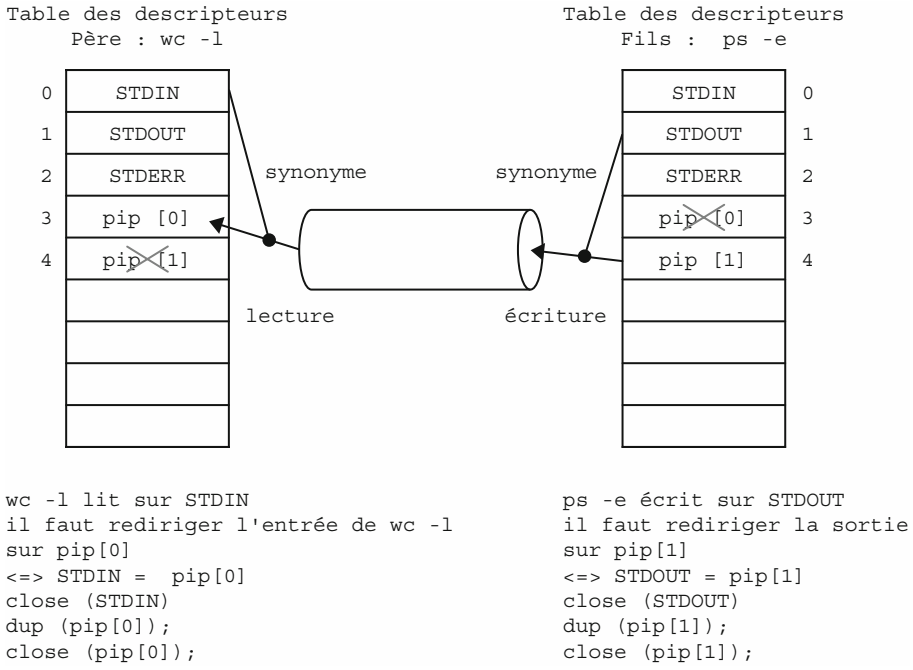

Figure 7.2 – Redirection des descripteurs standards *via* la primitive `dup()`

La primitive `int dup(int desc)` associe le plus petit descripteur disponible du processus appelant à la même entrée dans la table des fichiers ouverts que le descripteur `desc`. Pour rediriger un descripteur standard sur un descripteur de tube, il faut donc (figure 7.2) :

- fermer le descripteur standard ;
- faire un appel à la primitive `dup()` avec comme paramètre le descripteur du tube concerné. Le descripteur standard est associé au descripteur du tube et devient synonyme de celui-ci ;
- fermer le descripteur du tube. Seul reste le descripteur standard rendu synonyme.

```
/*********************************************************************/
/*                     Commande ls -l | wc -l                        */
/*********************************************************************/
#include <stdio.h>
#include <unistd.h>

main()
{
int p[2];

/*  ouverture d'un tube */
if(pipe(p))
  {
  perror("pipe");
  exit();
  }
switch (fork()) {
case -1: /* erreur */
        perror (" pb fork ");
        exit();
case 0: /* le processus fils exécute la commande ls -l */
      /* la sortie standard du processus est redirigée sur le tube */
        close (STDOUT_FILENO);
        (void)dup(p[1]); /* p[1] sortie standard du processus */
          close (p[1]);
         close (p[0]); /* le processus ne lit pas dans le tube */
         execlp ("ls", "ls", "-l", NULL);
default: /* le processus père exécute la commande wc -l */
       /* l'entrée standard du processus est redirigée sur le tube */
         close (STDIN_FILENO);
         (void)dup(p[0]); /* p[1] sortie standard du processus */
          close (p[0]);
          close (p[1]);
          execlp ("wc", "wc", "-l", NULL);
          }
}
```

7.1.2 Les tubes nommés

Les *tubes nommés* sont également gérés par le système de gestion de fichiers, et correspondent au sein de celui-ci à un fichier avec un nom. De ce fait, ils sont accessibles par n'importe quel processus connaissant ce nom et disposant des droits d'accès au tube. Le tube nommé permet donc à des processus sans liens de parenté de communiquer selon un mode flots d'octets.

Les tubes nommés apparaissent lors de l'exécution d'une commande `ls -l` et sont caractérisés par le type p. Ce sont des fichiers constitués d'une inode à laquelle n'est associé aucun bloc de données. Tout comme pour les tubes anonymes, les données contenues dans les tubes sont placées dans un tampon, constitué par une seule case de la mémoire centrale.

a) Création d'un tube nommé

Un tube nommé est créé par l'intermédiaire de la fonction `mkfifo()` dont le prototype est :

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo (const char *nom, mode_t mode);
```

Le paramètre `nom` correspond au chemin d'accès dans l'arborescence de fichiers pour le tube. Le paramètre `mode` correspond aux droits d'accès associés au tube, construits de la même manière que pour tout autre fichier (*cf.* chapitre 3). La primitive renvoie 0 en cas de succès et – 1 dans le cas contraire.

L'appel à la primitive `mkfifo()` est équivalent à l'appel suivant utilisant la primitive `mknod()` vue au chapitre 3 :

```
int mknod (nom, mode|S_FIFO, 0);
```

b) Ouverture d'un tube nommé

L'ouverture d'un tube nommé par un processus s'effectue en utilisant la primitive `open()` déjà rencontrée au chapitre 3 :

```
int open (const char  *nom, int mode_ouverture);
```

Le processus effectuant l'ouverture doit posséder les droits correspondants sur le tube. La primitive renvoie un descripteur correspond au mode d'ouverture spécifié (lecture seule, écriture seule, lecture/écriture).

Par défaut, la primitive `open()` appliquée au tube nommé est bloquante. Ainsi, la demande d'ouverture en lecture est bloquante tant qu'il n'existe pas d'écrivain sur le tube. D'une manière similaire, la demande d'ouverture en écriture est bloquante tant qu'il n'existe pas de lecteur sur le tube. Ce mécanisme permet à deux processus de se synchroniser et d'établir un rendez-vous en un point particulier de leur exécution.

c) Lecture et écriture sur un tube nommé

La lecture et l'écriture sur le tube nommé s'effectuent en utilisant les primitives `read()` et `write()` vues précédemment.

d) Fermeture et destruction d'un tube nommé

La *fermeture* d'un tube nommé s'effectue en utilisant la primitive `close()`. La *destruction* d'un tube nommé s'effectue en utilisant la primitive `unlink()`.

e) Exemples d'utilisation des tubes nommés

Communication unidirectionnelle

Ce premier exemple illustre une communication unidirectionnelle entre deux processus. Un premier processus crée un tube nommé appelé `fictub`, puis écrit la chaîne "0123456789" dans ce tube. L'autre processus lit la chaîne et l'affiche. À l'issue de l'exécution de ces deux processus, la commande `ls -l` montre le tube nommé `fictub`.

```
/*********************************************************************/
/*             Processus écrivain sur le tube nommé                  */
/*********************************************************************/

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

main ()
{
mode_t mode;
int tub;
mode = S_IRUSR | S_IWUSR;

/*création du tube nommé */
mkfifo ("fictub", mode);

/* ouverture du tube */
tub = open ("fictub", O_WRONLY);

/* écriture dans le tube */
write (tub, "0123456789", 10);

/* fermeture du tube */
close(tub);
}
/*********************************************************************/
/*             Processus lecteur sur le tube nommé                   */
/*********************************************************************/
#include <stdio.h>
#include <fcntl.h>
```

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

main ()
{
char zone[11];
int tub;

/* ouverture du tube */
tub = open ("fictub", O_RDONLY);

/* lecture dans le tube */
read (tub, zone, 10);
zone[10] = 0;

printf ("processus lecteur du tube fictub: j'ai lu %s", zone);

/* fermeture du tube */
close(tub);
}
```

Communication bidirectionnelle

Ce second exemple illustre une communication bidirectionnelle à travers deux tubes nommés `tube1` et `tube2`. Un processus client envoie deux nombres entiers à un processus serveur au travers du premier tube `tube1`. Le processus serveur effectue l'addition de ces deux nombres et renvoie le résultat au processus client à travers le tube `tube2`. Le processus serveur a lui-même créé les deux tubes nommés.

Les données véhiculées dans les tubes étant de type chaîne de caractères, les entiers doivent être convertis dans ce format au moment de l'envoi et reconvertis dans l'autre sens à la réception.

```
/*********************************************************************/
/*                        Processus client                           */
/*********************************************************************/

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

main ()
{
int tub1, tub2;
int nb1, nb2, res;
char ent[2];

/* ouverture du tube tube1 en écriture  */
tub1 = open ("tube1", O_WRONLY);
```

```
/* ouverture du tube tube2 en lecture */
tub2 = open ("tube2", O_RDONLY);

printf ("Donnez moi deux nombres à additionner:");
scanf ("%d %d", &nb1, &nb2);

sprintf (ent, "%d", nb1);
/* écriture dans le tube */
write (tub1, ent, 2);

sprintf (ent, "%d", nb2);
/* écriture dans le tube */
write (tub1, ent, 2);

/* lecture du résultat */
read (tub2,ent,2);
res = atoi(ent);

printf ("le résultat reçu est: %d", res);

/* fermeture et destruction des tubes */
close(tub1);
close(tub2);
unlink(tub1);
unlink(tub2);
}
/*********************************************************************/
/*                        Processus serveur                          */
/*********************************************************************/
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

main ()
{
char ent[2];
int tub1, tub2;
int nb1, nb2,res;
mode_t mode;

mode = S_IRUSR | S_IWUSR;

/*création de deux tubes nommés */
mkfifo ("tube1", mode);
mkfifo ("tube2", mode);


/* ouverture du tube 1 en lecture */
tub1 = open ("tube1", O_RDONLY);
```

```
/*ouverture du tube 2 en ecriture */
tub2 = open ("tube2", O_WRONLY);


/* lecture dans le tube */
read (tub1, ent, 2);
nb1 = atoi(ent);
read (tub1, ent, 2);
nb2 = atoi(ent);


res = nb1 + nb2;
sprintf (ent, "%d", res);

write (tub2,ent,2);

/* fermeture du tube */
close(tub1);
close(tub2);
}
```

7.2 Les IPC : files de messages, mémoire partagée

7.2.1 Caractéristiques générales

Les IPC (*Inter Process Communication*) forment un groupe de trois outils de communication indépendants des tubes anonymes ou nommés, dans le sens où ils n'appartiennent pas au système de gestion de fichiers. Ces trois outils sont :

- les files de messages ou MSQ (*Messages Queues*) ;
- les régions de mémoire partagée ;
- les sémaphores que nous étudions au chapitre 8.

Ces trois outils sont gérés dans des tables du système, une par outils.

Un outil IPC est identifié de manière unique par un identifiant externe appelé la clé (qui a le même rôle que le chemin d'accès d'un fichier) et par un identifiant interne (qui joue le rôle de descripteur). Un outil IPC est accessible à tout processus connaissant l'identifiant interne de cet outil. La connaissance de cet identifiant s'obtient par héritage ou par une demande explicite au système au cours de laquelle le processus fournit l'identifiant externe de l'outil IPC.

La clé est une valeur numérique de type `key_t`. Les processus désirant utiliser un même outil IPC pour communiquer doivent se mettre d'accord sur la valeur de la clé référençant l'outil. Ceci peut être fait de deux manières :

- la valeur de la clé est figée dans le code de chacun des processus ;
- la valeur de la clé est calculée par le système à partir d'une référence commune à tous les processus. Cette référence est composée de deux parties, un nom de fichier

et un entier. Le calcul de la valeur de la clé à partir cette référence est effectuée par la fonction `ftok()`, dont le prototype est :

```
#include <sys/ipc.h>
key_t ftok (const char *ref, int numero);
```

La commande `ipcs` permet de lister l'ensemble des outils IPC existants à un moment donné dans le système. Les informations fournies par cette commande sont notamment pour chaque outil IPC :

- le type (*q* pour messages queues, *s* pour sémaphores, *m* pour les régions de mémoire partagée) ;
- l'identification interne ;
- la valeur de la clé ;
- les droits d'accès définis ;
- le propriétaire ;
- le groupe propriétaire.

```
delacroix@vesuve:~> ipcs
------ Segments de mémoire partagée --------
touche      shmid  propriétaire perms     octets   nattch   statut
0x00000000 1015808    root      644        118784     3      dest
0x00000000 1114113    root      644        118784     2      dest
0x00000000 1146882    root      644        110592     2      dest
0x00000000 1179651    root      644        110592     2      dest
0x00000000 1212420    root      644        151552     4      dest
0x00000000 1245189    root      644        151552     4      dest

------ Tables de sémaphores --------
touche    semid   propriétaire perms      nsems
0x0000000c 0       delacroix     600        1

------ Files d'attente de messages --------
touche     msqid    propriétaire perms    octets utilisés messages
0x0000013a  0        delacroix    750      0            0
```

7.2.2 Les files de messages

Le noyau Linux gère au maximum `MSGMNI` files de messages (128 par défaut), pouvant contenir des messages dont la taille maximale est de 4 056 octets.

a) Accès à une file de message

L'accès à une file de message s'effectue par l'intermédiaire de la primitive `msgget()`. Cette primitive permet :

- la création d'une nouvelle file de messages ;
- l'accès à une file de messages déjà existante.

Le prototype de la fonction est :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t cle, int option);
```

Le paramètre `cle` correspond à l'identification externe de la file de messages. Le paramètres `options` est une combinaison des constantes `IPC_CREAT`, `IPC_EXCL` et de droits d'accès définis comme dans le cadre des fichiers. La fonction renvoie l'identifiant interne de la file de messages en cas de succès et la valeur – 1 sinon. La variable `errno` peut alors prendre les valeurs suivantes :

- `EACCESS`, le processus ne dispose pas des droits appropriés ;
- `EEXIST`, une ressource IPC de même numéro de clé que celle dont la création est demandée existe déjà ;
- `EIDRM`, la ressource IPC a été détruite ;
- `ENOENT`, aucune ressource IPC identifiée par la clé fournie n'existe pas et la création n'est pas demandée ;
- `ENOMEM`, la table des IPC est saturée ;
- `ENFILE`, le nombre maximal de ressources IPC est dépassé.

Création d'une file de messages

La création d'une file de messages est demandée en positionnant les constantes `IPC_CREAT` et `IPC_EXCL`. Une nouvelle file est alors créée avec les droits d'accès définis dans le paramètre `option`. Le processus propriétaire de la file est le processus créateur tandis que le groupe propriétaire de la file est le groupe du processus créateur. Si ces deux constantes sont positionnées et qu'une file d'identifiant externe `cle` existe déjà, alors une erreur est générée. Si seule la constante `IPC_CREAT` est positionnée et qu'une file d'identifiant externe égal à `cle` existe déjà, alors l'accès à cette file est retourné au processus.

Ainsi l'exécution de `msgget (cle, IPC_CREAT | IPC_EXCL | 0660)` crée une nouvelle file avec des droits en lecture et écriture pour le processus propriétaire de la file et pour les processus du groupe.

Accès à une file déjà existante

Un processus désirant accéder à une file déjà existante effectue un appel à la primitive `msgget()` en positionnant à 0 le paramètre `option`.

Le cas particulier clé = IPC_PRIVATE

Un processus peut demander l'accès à une file de messages en positionnant le paramètre `cle` à la valeur `IPC_PRIVATE`. Dans ce cas, la file créée est seulement accessible par ce processus et ses descendants.

b) Envoi et réception de message

La communication au travers d'une file de messages peut être bidirectionnelle, c'est-à-dire qu'un processus consommateur de messages dans la file peut devenir producteur de messages pour cette même file. La communication mise en œuvre est une communication de type boite aux lettres, préservant les structures des messages.

Chaque message comporte les données en elles-mêmes ainsi qu'un type qui permet de faire du multiplexage dans la file de messages et de désigner le destinataire d'un message.

Format des messages

Un message est toujours composé de deux parties :

- la première partie constitue le type du message. C'est un entier long positif ;
- la seconde partie est composée des données proprement dites.

Toutes les données composant le message doivent être contiguës en mémoire centrale. De ce fait, le type pointeur est interdit.

Un exemple de structure de messages est par exemple :

```
struct message {
               long mtype;
               int n1;
               char[4];
               float fl1; };
```

Envoi d'un message

L'envoi d'un message dans une file de messages s'effectue par le biais de la primitive `msgsnd()`.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd (int idint, const void *msg, int longueur, int option);
```

Le paramètre `idint` correspond à l'identifiant interne de la file. Le paramètre `*msg` est l'adresse du message en mémoire centrale tandis que le paramètre `longueur` correspond à la taille des données seules dans le message `msg` envoyé. Par défaut, la primitive `msgsnd()` est bloquante c'est-à-dire qu'un processus est suspendu lors d'un dépôt de messages si la file est pleine. En positionnant le paramètre `option` à la valeur `IPC_NOWAIT`, la primitive de dépôt devient non bloquante. La primitive renvoie 0 en cas de succès, – 1 sinon. La variable `errno` peut prendre alors les valeurs suivantes :

- `EINVAL`, la file n'existe pas ou le type spécifié dans le message n'est pas valide ;
- `EACESS`, le processus n'a pas les droits d'écriture sur la ressource IPC ;
- `EAGAIN`, la file est pleine et l'option `IPC_NOWAIT` n'a pas été positionnée ;
- `EFAULT`, l'adresse du message à envoyer n'est pas valide ;
- `EIDRM`, la file a été détruite ;

- EINTR, le processus a reçu un signal et l'appel système a échoué ;
- ENOMEM, la mémoire manque pour pouvoir copier l'objet.

Réception d'un message

Un processus désirant prélever un message depuis une file de messages utilise la primitive msgrcv().

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv (int idint, const void *msg, int longueur, long letype, int
option);
```

Le paramètre idint correspond à l'identifiant interne de la file. Le paramètre *msg est l'adresse d'une zone en mémoire centrale pour recevoir le message tandis que le paramètre longueur correspond à la taille des données seules dans le message msg envoyé.

Le paramètre letype permet de désigner un message à extraire, en fonction du champ type de celui-ci. Plus précisément :

- si letype est strictement positif, alors le message le plus ancien dont le type est égal à letype est extrait de la file ;
- si letype est nul, alors le message le plus ancien est extrait de la file. La file est alors gérée en FIFO ;
- si letype est négatif, alors le message le plus ancien dont le type est le plus petit inférieur ou égal à |letype| est extrait de la file. Ce mécanisme instaure des priorités entre les messages.

Par défaut, la primitive msgrcv() est bloquante c'est-à-dire qu'un processus est suspendu lors d'un retrait de messages si la file ne contient pas de messages correspondant au type attendu. En positionnant le paramètre option à la valeur IPC_NOWAIT, la primitive de retrait devient non bloquante. Le paramètre option peut également prendre la valeur MSG_EXCEPT. Dans ce cas, un message de n'importe quel type sauf celui spécifié dans letype est prélevé. Enfin si le paramètre option prend la valeur MSG_ERROR, alors un message trop long sera tronqué sans qu'une erreur soit levée.

La primitive renvoie la longueur du message prélevé en cas de succès, – 1 sinon. La variable errno peut prendre alors les valeurs suivantes :

- EINVAL, la file n'existe pas ou le type spécifié dans le message n'est pas valide ou la taille du message spécifié est supérieure à la limite autorisée (MSGMAX) ;
- EACESS, le processus n'a pas les droits de lecture sur la ressource IPC ;
- E2BIG, la taille du message est plus grande que celle spécifiée lors de l'appel et l'option MSG_NOERROR n'a pas été indiquée ;
- EFAULT, l'adresse de la zone pour réceptionner le message n'est pas valide ;
- EIDRM, la file a été détruite alors que le processus attendait un message ;
- EINTR, le processus a reçu un signal et l'appel système a échoué ;

- ENOMSG, l'option IPC_NOWAIT a été positionnée et aucun message n'a été trouvé dans la file.

c) Destruction d'une file de message

La destruction d'une file de messages s'effectue en utilisant la primitive msgctl() dont le paramètre operation est positionné à la valeur IPC_RMID. La valeur renvoyée est 0 en cas de succès et – 1 sinon.

```
msgctl (int idint, IPC_RMID, NULL);
```

La suppression d'une file de messages peut également être réalisée depuis le prompt du shell par la commande ipcrm - q identifiant ou ipcrm -Q cle.

d) Un exemple d'utilisation

Nous donnons ici un exemple de communication clients-serveur par le biais d'une file de messages. Comme pour l'exemple concernant les tubes nommés, le serveur effectue l'addition de deux nombres entiers envoyés par un client et renvoie le résultat. La communication s'effectue à présent par l'intermédiaire d'une file de messages unique de clé égale à 314. Cette file de message contient donc tout à la fois les requêtes en provenance des clients et les réponses du serveur.

Dans cette application, le serveur doit uniquement consommer les requêtes depuis la file de messages. Les clients de leur côté doivent uniquement lire la réponse qui les concerne.

Pour parvenir à ce schéma, on utilise le champ type dans la structure des messages véhiculés par la file de messages. Ainsi, le serveur désigne le client auquel s'adresse la réponse en remplissant le champ type de la réponse avec la valeur du pid du client. Ce pid est indiqué par le client au niveau de sa requête. Le client quant à lui désigne le serveur comme destinataire de la requête en initialisant le champ type de sa requête avec une valeur positive qui ne peut pas correspondre à un pid de processus utilisateur, par exemple la valeur 1 (cette valeur 1 correspond obligatoirement au processus init).

Ainsi une requête et une réponse ont le format suivant :

```
struct requete {
                long letype; /* prend la valeur 1 */
                int nb1; /* premier membre de l'addition */
                int nb2; /* second membre de l'addition */
                pid_t mon_pid;  /*le client indique ici son pid */
                };
struct reponse {
                long letype; /* prend la valeur du pid du client */
                int res; /* le résultat de l'addition */
                };

/*****************************************************************/
/*                 Processus serveur: crée la MSQ                */
/*    et additionne les deux nombres reçus dans chaque message    */
/*****************************************************************/
```

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define CLE 314

struct requete {
                long letype;
                int nb1;
                int nb2;
                pid_t mon_pid;
               };
struct reponse {
                long letype;
                int res;
                };

main()
{
int msqid,l;
struct requete la_requete;
struct reponse la_reponse;
/*   allocation  MSQ */

if((msqid=msgget((key_t)CLE,0750|IPC_CREAT|IPC_EXCL))==-1)
  {
   perror("msgget");
   exit(1);
  }

while (1)
{
/* lecture d'une requête */
if((l=msgrcv(msqid,&la_requete,sizeof(struct requete)-4,1,0))==-1)
  {
   perror("msgrcv");
   exit(2);
  }
la_reponse.res = la_requete.nb1 + la_requete.nb2;
la_reponse.letype=la_requete.mon_pid;
/* type associé au message; le pid du client  */
if(msgsnd(msqid,&la_reponse,sizeof(struct reponse) - 4,0)==-1)
  {
   perror("msgsnd");
   exit(2);
  }
}
exit(0);
}
```

```
/*********************************************************************/
/*      Processus client: envoi de deux nombres à additionner     */
/*********************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define CLE 314

struct requete {
                long letype;
                int nb1;
                int nb2;
                pid_t mon_pid;
               };
struct reponse {
                long letype;
                int res;
                };
main()
{
int msqid, l, nb1, nb2;
struct requete la_requete;
struct reponse la_reponse;

/* récupération du msqid */
if((msqid=msgget((key_t)CLE,0))<0)
  {
   perror("msgget");
   exit(1);
  }

/* préparation de la requête et envoi */
printf ("Donnez moi deux nombres à additionner:");
scanf ("%d %d", &nb1, &nb2);

la_requete.letype = 1;
la_requete.nb1 = nb1;
la_requete.nb2 = nb2;
la_requete.mon_pid = getpid();

if(msgsnd(msqid,&la_requete,sizeof(struct requete)-4,0)==-1)
  {
   perror("msgsnd");
   exit(2);
  }

/* réception de la réponse */
if((l=msgrcv(msqid,&la_reponse,sizeof(struct reponse)-4,
getpid(),0)==-1))
```

```
  {
   perror("msgrcv");
   exit(2);
  }

printf ("le resultat reçu est: %d", la_reponse.res);
exit(0);
}
```

7.2.3 Les régions de mémoire partagée

Par défaut, l'espace d'adressage de chaque processus est privé, c'est-à-dire que cet espace mémoire ne peut pas être accédé par un autre processus. Les *régions de mémoire partagée* (*Shared Memory*) constituent une extension de l'espace d'adressage d'un processus, extension qui elle peut être partagée avec d'autres processus. Pour pouvoir utiliser une région de mémoire partagée, un processus doit d'abord créer cette région, puis doit l'attacher à son propre espace d'adressage. Une région de mémoire partagée est accessible par tout processus ayant connaissance de son identification interne et ayant les droits nécessaires pour lire ou écrire dans la région. Ce partage d'une zone mémoire par plusieurs processus implique de synchroniser les accès aux données présentes dans la région, afin que la cohérence de celles-ci soit respectée. Cette synchronisation des accès est souvent réalisée à l'aide du troisième outil IPC, les sémaphores, que nous étudions au chapitre 8, après avoir décrit les schémas classiques de synchronisation.

a) Création ou accès à une région de mémoire partagée

L'accès à ou la création d'une région de mémoire partagée s'effectue par l'intermédiaire de la primitive `shmget()`. Le prototype de la fonction est :

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t cle, int taille, int option);
```

Le paramètre `cle` correspond à l'identification externe de la région de mémoire. Le paramètre `taille` spécifie la taille de la région de mémoire partagée que l'on désire. Le paramètre `options` est une combinaison des constantes `IPC_CREAT`, `IPC_EXCL` et de droits d'accès. Leur emploi suit exactement la logique décrite précédemment dans le cas des files de messages. La fonction renvoie l'identifiant interne de la région de mémoire partagée en cas de succès et la valeur – 1 sinon. La variable `errno` peut alors prendre les valeurs suivantes :

- `EACCESS`, le processus ne dispose pas des droits appropriés ;
- `EEXIST`, une ressource IPC de même numéro de clé que celle dont la création est demandée existe déjà ;
- `EIDRM`, la ressource IPC a été détruite ;
- `EINVAL`, les paramètres ne sont pas valides ;
- `ENOMEM`, la mémoire est insuffisante ;

- ENOENT, il n'existe pas de région de mémoire partagée correspondant à la clé spécifiée ;
- EFAULT, les paramètres sont incorrects ;
- ENOSPC, tous les identificateurs de mémoire partagée sont utilisés.

b) Attachement d'une région de mémoire partagée

L'attachement d'une région de mémoire partagée s'effectue par l'intermédiaire de la primitive shmat(). Le prototype de la fonction est :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmadd, int option);
```

Le paramètre shmid correspond à l'identifiant interne de la région de mémoire. Le paramètre *shmadd spécifie l'adresse de la région de mémoire partagée. Deux cas sont possibles :

- si shmadd = 0, alors la région est attachée à la première adresse disponible déterminée par le système ;
- si shmadd est non nul et si option = SHM_RND alors la région est attachée à l'adresse shmadd modulo SHMLBA (constante définie dans le fichier /sys/shm.h) ;
- si shmadd est non nul et si option n'est pas égal à SHM_RND alors la région est attachée à l'adresse shmadd.

Le paramètre option définit également les droits d'accès à la région. Par défaut, la région est attachée à l'espace d'adressage du processus avec des droits en lecture et en écriture. Cependant si le paramètre option prend la valeur SHM_RONLY, alors le processus acquiert des droits en lecture seule sur la région de mémoire.

La primitive renvoie l'adresse de la région de mémoire partagée en cas de succès. En cas d'erreur, la primitive renvoie la valeur – 1 et la variable errno peut alors prendre les valeurs suivantes :

- EACCESS, le processus ne dispose pas des droits appropriés ;
- EIDRM, la ressource IPC a été détruite ;
- EINVAL, la clé ou l'adresse sont invalides ;
- ENOMEM, la mémoire est insuffisante.

Un processus fils hérite à sa création de toutes les régions de mémoire partagée attachées à l'espace d'adressage de son père.

c) Détachement d'une région de mémoire partagée

Le détachement d'une région de mémoire partagée s'effectue par l'intermédiaire de la primitive shmdt().

Le prototype de la fonction est :

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

```
#include <sys/shm.h>
void *shmdt(const void *shmadd);
```

La région de mémoire détachée devint inaccessible pour le processus appelant. La terminaison d'un processus entraîne le détachement de toutes les régions qu'il avait préalablement attachées à son espace d'adressage.

d) Destruction d'une région de mémoire partagée

La destruction d'une région de mémoire partagée s'effectue en utilisant la primitive `shmctl()` dont le paramètre `operation` est positionné à la valeur `IPC_RMID`. La valeur renvoyée est 0 en cas de succès et – 1 sinon.

```
shmctl (int shmid, IPC_RMID, NULL);
```

La suppression d'une région de mémoire partagée peut également être réalisée depuis le prompt du shell par la commande `ipcrm - q identifiant` ou `ipcrm -Q cle`.

e) Un exemple d'utilisation

Nous donnons ici un exemple très simple d'utilisation des régions de mémoire partagée. Dans cet exemple, un premier processus crée une région de mémoire partagée de clé 256 puis écrit un message dans la région. Un second processus relit ce message puis détruit la région de mémoire.

```
/*********************************************************************/
/*           Processus créateur de la région et écrivain             */
/*********************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define CLE         256

main()
{
int shmid;
char *mem;

/* création de la région de mémoire partagée avec la clé CLE */
if((shmid=shmget((key_t)CLE,1000,0750 |IPC_CREAT | IPC_EXCL)==-1)
{
   perror("shmget");
   exit(1);
}

/* attachement */
if((mem=shmat(shmid,NULL,0))==(char *)-1)
{
```

```
   perror("shmat");
   exit(2);
}

/* écriture dans la région */
strcpy(mem,"voici une écriture dans la region");
exit(0);
}


/*****************************************************************/
/*          Processus destructeur de la région et lecteur        */
/*****************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define CLE         256

main()
{
/* récupération de la région de mémoire */
if((shmid=shmget((key_t)CLE,0,0))<0)
{
   perror("shmget");
   exit(1);
}

/* attachement */
if(mem=shmat(shmid,NULL,0))==(char *)-1)
{
   perror("shmat");
   exit(2);
}

/* lecture dans la région */
printf("lu: %s\n",mem);

/* détachement du processus */
if(shmdt(mem))
{
   perror("shmdt");
   exit(3);
}

/* destruction de la région */
shmctl(shmid,IPC_RMID,NULL);

exit(0)
}
```

Exercices

7.1 Qu'avez-vous retenu ?

Pour chacune des questions suivantes, choisissez la ou les réponses qui vous semblent exactes.

Question 1 – Un tube anonyme :

- ☐ a. permet une communication bidirectionnelle entre n'importe quel processus.
- ☐ b. permet une communication unidirectionnelle entre un père et son fils.
- ☐ c. permet une communication unidirectionnelle entre un père et ses descendants créés après le tube.

Question 2 – Une file de messages :

- ☐ a. préserve la structure des messages.
- ☐ b. permet une communication bidirectionnelle.
- ☐ c. ne peut être utilisée qu'entre un père et ses fils.

Question 3 – Une région de mémoire partagée :

- ☐ a. est une région mémoire accessible à n'importe quel processus.
- ☐ b. est une région mémoire partageable entre processus qui connaissent sa clé et l'ont attachée à leur espace d'adressage.

Question 4 – Les outils IPC :

- ☐ a. sont gérés avec les tubes, par le système de gestion de fichiers.
- ☐ b. sont gérés à part dans des tables du système et sont repérés par une clé unique.

Question 5 – La lecture dans un tube :

- ☐ a. permet de faire des accès directs de n'importe quelle donnée contenue dans le tube.
- ☐ b. est en mode FIFO.
- ☐ c. est destructive.

Question 6 – Le signal SIGPIPE :

- ☐ a. est levé par le système lorsqu'un processus commet un accès mémoire illégal.
- ☐ b. est levé par le système lorsqu'un processus écrit dans un tube nommé sans lecteur.
- ☐ c. est levé par le système lorsqu'un processus écrit dans un tube anonyme sans lecteur.

Question 7 – Dans une file de message, le type d'un message permet :

- ☐ a. de spécifier la nature des données présentes dans le message.
- ☐ b. de désigner le consommateur du message.

7.2 Tubes anonymes

Écrivez un programme répondant à l'énoncé suivant.

Un processus père crée deux fils. Le père communique avec son premier fils par le biais d'un tube *pip*, dans lequel il envoie 3 caractères, lus un à un au clavier grâce à la primitive `read (char c) ;`

Le premier fils obtient du père un ensemble de 3 caractères, qu'il lit depuis le tube pip. Il effectue ensuite un appel à la primitive `execl`, pour recouvrir son code avec l'exécutable de nom MAJ, dont le chemin dans l'arborescence de fichiers est `/home/élèves/MAJ`, avec comme paramètres les trois caractères lus depuis le tube.

Le second fils arme une alarme de 15 ms, et lorsque celle-ci tombe, il tue le premier fils.

7.3 Tubes anonymes

On considère une application composée de trois processus PA, PB et PC. Le processus PA lit toutes les *P* unités de temps un ensemble de mesures (5 entiers) depuis une série de capteurs et les transmet au processus PB en utilisant un outil de communication. Le processus PB écrit cet ensemble de mesures dans un fichier appelé Mesure sur le disque puis il imprime celles-ci sur une imprimante. Le processus PC, réveillé toutes les P' unités de temps, lit l'ensemble des mesures contenues dans le fichier Mesure, effectue un traitement sur ces mesures et imprime le résultat.

Dans cet exercice on considère que la lecture des grandeurs s'effectue en tout 10 000 fois.

Le processus PB est le fils du processus PA. Le code exécutable du processus PB est contenu dans le fichier `/home/durant/application/pb.exe`. Les deux processus communiquent à l'aide d'un tube anonyme. Complétez le squelette des processus PA et PB suivants.

```
#define P      200
main()
{
/* déclarations de variables */
pid_t pid;
int i; int grandeurs [5];
/* tableau de 5 entiers qui contient les grandeurs lues par PA */

 déclarations à compléter


i = 0;
/* mise en place des moyens de communication */
/*et création des processus */

/*                    à compléter                          */

if (à compléter)
{
```

```
/*                    à compléter                          */

}
else {

/*                    à compléter                          */

while (i < 10000) /* il y a encore des mesures à faire */
{
/* lire les grandeurs */
Lecture_capteurs (grandeurs);
/* écrire vers le fils */

/*                    à compléter                          */

i = i  + 1;
sleep (P); /* dormir durant P unités de temps */
}

/*                    à compléter                          */

}
}
```

7.4 Files de messages

Soit une application clients-serveurs dans laquelle deux serveurs *Serveur_1* et *Serveur_2* dialoguent avec les clients par l'intermédiaire d'une file de messages. Le serveur *Serveur_1* effectue l'addition de deux nombres entiers envoyés par le client et renvoie le résultat de l'opération tandis que le serveur *Serveur_2* effectue la multiplication des trois nombres flottants envoyés par le client et renvoie le résultat de l'opération au client. Chacun des serveurs ne doit prélever dans la file de messages que les requêtes qui le concerne. Un client doit évidemment recevoir le résultat correspondant à sa requête.

Écrivez les programmes C correspondant à cette application.

7.5 Tubes anonymes et tubes nommés

On remplace la file de messages utilisée dans l'exercice 3 par des tubes anonymes.

1) Comment faut-il modifier l'architecture logicielle de l'exercice précédent ? Quelles difficultés lèvent cette modification de l'architecture ?
2) En utilisant la primitive `select()` décrite dans le chapitre 4, écrivez le code C correspondant à la solution, pour un seul serveur et deux clients.

7.6 Tubes anonymes et signaux

Soit le code C suivant. Que va-t-il se passer ?

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

```
 void attention_tube(){
  printf("signal SIGPIPE recu\n");
  }

main()
{
int pip[2], pid;
int nb_ecrit;

signal (SIGPIPE, attention_tube);

/*  ouverture d'un pipe */
if(pipe(pip))
   { perror("pipe");
    exit(1);}

pid = fork();
if (pid == 0)
{
close(pip[0]);
close(pip[1]);
printf("Je suis le fils\n");
exit(9);
}
else
{
close(pip[0]);
for(;;) {
if ((nb_ecrit = write(pip[1], "ABC", 3)) == -1)
{
   perror ("pb write");
   exit();
}
}
}
}
```

7.7 Tubes nommés

On remplace la file de messages utilisée dans l'exercice 3 par des tubes nommés. Écrivez les programmes correspondants.

Des idées pour vous exercer à la programmation

Soit l'architecture clients-serveurs suivante :

– Des clients C_i envoient des messages à deux serveurs S1 et S2 à travers une MSQ, cette MSQ ayant été au préalable créée par S1. Les messages sont typés et composés soit d'un opérande x, soit de deux opérandes x et y, soit d'un mot en majuscule.

– Le serveur S1 prélève les messages composés de 1 ou de 2 opérandes. Pour chaque message prélevé dans la MSQ, le serveur S1 crée un processus P_i. Si le message comporte un seul opérande x, le processus P_i calcule x^3, puis renvoie le résultat au client avant de se tuer. Si le message comporte deux opérandes x et y, le processus P_i calcule $x^2 + y^2$, puis renvoie le résultat au client avant de se tuer. Si il y a plusieurs messages en attente, le serveur S1 crée autant de processus P_i que de messages afin de traiter les messages en parallèle. Le processus S1 et les processus P_i communiquent par tubes.

– Le serveur S2 prélève les messages composés d'un mot. Pour chaque message prélevé dans la MSQ, le serveur S2 crée un processus P_i. Le processus P_i transforme le mot pour le mettre en minuscules puis renvoie le résultat au client avant de se tuer. Si il y a plusieurs messages en attente, le serveur S2 crée autant de processus P_i que de messages afin de traiter les messages en parallèle. Le processus S2 et les processus P_i communiquent par tubes.

– Les processus P_i renvoient les résultats aux processus C_i à travers la MSQ. Les processus C_i affichent alors ce résultat.

Écrivez les codes C correspondants.

Solutions

7.1 Qu'avez-vous retenu ?

Question 1 – Un tube anonyme :

- ☐ b. permet une communication unidirectionnelle entre un père et son fils.
- ☐ c. permet une communication unidirectionnelle entre un père et ses descendants créés après le tube.

Question 2 – Une file de messages :

- ☐ a. préserve la structure des messages.
- ☐ b. permet une communication bidirectionnelle.

Question 3 – Une région de mémoire partagée :

- ☐ b. est une région mémoire partageable entre processus qui connaissent sa clé et l'ont attachée à leur espace d'adressage.

Question 4 – Les outils IPC :

- ☐ b. sont gérés à part dans des tables du système et sont repérés par une clé unique.

Question 5 – La lecture dans un tube :

- ☐ b. est en mode FIFO.
- ☐ c. est destructive.

Question 6 – Le signal SIGPIPE :

- ☐ c. est levé par le système lorsqu'un processus écrit dans un tube anonyme sans lecteur.

Question 7 – Dans une file de message, le type d'un message permet :

- ☐ b. de désigner le consommateur du message.

7.2 Tubes anonymes

```
pid_t pid1, pid2;
main()
{
/* déclarations de variables */
int tube[2];
char c1, c2, c3;

/* mise en place des moyens de communication et création des processus */
pipe(tube);
pid2 = fork();
if (pid1 == 0)
{
close (tube[1]);
read (tube[0], c1, 1);
read (tube[0], c2, 1);
read (tube[0], c3, 1);
execl("/home/élèves/MAJ", "MAJ", c1, c2, c3, NULL);
}
else {
close tube[0];
printf ("Donnez moi trois caractères minuscules:");
scanf("%s %s %s",c1, c2, c3);
write (tube[1], c1, 1);
write (tube[1], c2, 1);
write (tube[1], c3, 1);
pid2 = fork();
if (pid2 == 0)
{
  signal (SIGALRM, onalarm);
  alarm(15);
  pause();
}
else
wait();
}
}

onalarm ()
{
printf("Handler onalarm\n");
kill (pid1, SIGKILL);
```

```
exit (1);
}
}
```

7.3 Tubes anonymes

```
#define P      200
main()
{
/* déclarations de variables */
pid_t pid;
int i; int grandeurs [5]; /* tableau de 5 entiers qui contient les
grandeurs lues par PA */
int tube[2];
char conv[2], char gd1[2], char gd2[2], char gd3[2], char gd4[2], char
gd5[2],

i = 0;
/* mise en place des moyens de communication et création des processus */
pipe(tube);
pid = fork();
if (pid == 0)
{
close (tube[1];
sprintf(conv, "%d", tube[0]);
execl("/home/Durant/application/pb.exe", "pb.exe", conv, NULL);
}
else {
close tube[0];
while (i < 10000) /* il y a encore des mesures à faire */
{
/* lire les grandeurs */
Lecture_capteurs (grandeurs);
/* écrire vers le fils */
sprintf(gd1, "%d", grandeur[0]);
sprintf(gd2, "%d", grandeur[1]);
sprintf(gd3, "%d", grandeur[2]);
sprintf(gd4, "%d", grandeur[3]);
sprintf(gd5, "%d", grandeur[4]);
write (tube[1], gd1, 2);
write (tube[1], gd2, 2);
write (tube[1], gd3, 2);
write (tube[1], gd4, 2);
write (tube[1], gd5, 2);
i = i  + 1;
sleep (P); /* dormir durant P unités de temps*/
}
wait()
}
}
```

7.4 Files de messages

```
/*****************************************************************/
/*    Processus client: envoi de deux entiers à additionner     */
/*            ou de trois flottants à multiplier                */
/*****************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct requete {
                long letype;
                int nb1;
                int nb2;
                pid_t mon_pid;
               };
struct reponse {
                long letype;
                int res;
                };
struct requetem {
                long letype;
                float nb1;
                float nb2;
                float nb3;
                pid_t mon_pid;
               };
struct reponsem {
                long letype;
                float res;
                };
main()
{
int msqid, l, nb1, nb2, choix;
key_t cle;
float fb1, fb2, fb3;
struct requete la_requete_add;
struct reponse la_reponse_add;
struct requetem la_requete_mul;
struct reponsem la_reponse_mul;

/* récupération du msqid */
cle = ftok("referencecle",0);
if((msqid=msgget(cle)) < 0)
  {
   perror("msgget");
   exit(1);
  }
```

```
printf ("Quel service desirez-vous? Addition (1), Multiplication (2)?\
n");
scanf ("%d",&choix);
if (choix == 1) {
  /* préparation de la requête et envoi */
  printf ("Donnez moi deux nombres à additionner:");
  scanf ("%d %d", &nb1, &nb2);

  la_requete_add.letype = 1;
  la_requete_add.nb1 = nb1;
  la_requete_add.nb2 = nb2;
  la_requete_add.mon_pid = getpid();

if(msgsnd(msqid,&la_requete_add,sizeof(struct requete)-4,0)==-1)
  {
   perror("msgsnd");
   exit(2);
  }

/* réception de la réponse */
if((l=msgrcv(msqid,&la_reponse_add,sizeof(struct reponse)-
4,getpid(),0)==-1))
  {
   perror("msgrcv");
   exit(3);
  }
printf ("le resultat reçu est: %d", la_reponse_add.res);
 }
 else
 {
  /* préparation de la requête et envoi */
  printf ("Donnez moi trois nombres à multiplier:");
  scanf ("%f %f %f", &fb1, &fb2, &fb3);
  la_requete_mul.letype = 2;
  la_requete_mul.nb1 = fb1;
  la_requete_mul.nb2 = fb2;
  la_requete_mul.nb3 = fb3;
  la_requete_mul.mon_pid = getpid();
if(msgsnd(msqid,&la_requete_mul,sizeof(struct requetem)-4,0)==-1)
  {
   perror("msgsnd");
   exit(2);
  }
/* réception de la réponse */
if((l=msgrcv(msqid,&la_reponse_mul,sizeof(struct reponsem)-
4,getpid(),0)==-1))
  {
   perror("msgrcv");
   exit(3);
  }
```

```
printf ("le resultat reçu est: %f", la_reponse_mul.res);
}
}

/*****************************************************************/
/*               Processus serveur: crée la MSQ                  */
/*   et additionne les deux nombres reçus dans chaque message    */
/*****************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>


struct requete {
                long letype;
                int nb1;
                int nb2;
                pid_t mon_pid;
               };
struct reponse {
                long letype;
                int res;
                };

main()
{
int msqid,l;
key_t cle;
struct requete la_requete;
struct reponse la_reponse;
/*   allocation  MSQ */
cle = ftok("referencecle",0);
if((msqid=msgget(cle,0750|IPC_CREAT|IPC_EXCL))==-1)
  {
   perror("msgget");
   exit(1);
  }

while (1)
{
/* lecture d'une requête */
if((l=msgrcv(msqid,&la_requete,sizeof(struct requete)-4,1,0))==-1)
  {
   perror("msgrcv");
   exit(2);
  }
la_reponse.res = la_requete.nb1 + la_requete.nb2;
la_reponse.letype=la_requete.mon_pid;
/* type associé au message; le pid du client  */
```

```
if(msgsnd(msqid,&la_reponse,sizeof(struct reponse) - 4,0)==-1)
  {
   perror("msgsnd");
   exit(2);
  }
}
exit(0);
}


/*****************************************************************/
/*       Processus serveur: multiplie les trois nombres          */
/*                    reçus dans chaque message                  */
/*****************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>


struct requetem {
                 long letype;
                 float nb1;
                 float nb2;
                 float nb3;
                 pid_t mon_pid;
                };
struct reponsem {
                 long letype;
                 float res;
                 };

main()
{
int msqid,l;
key_t cle;
struct requetem la_requete;
struct reponsem la_reponse;
/*  allocation  MSQ */
cle = ftok("referencecle",0);
if((msqid=msgget(cle,0))==-1)
  {
   perror("msgget");
   exit(1);
  }
while (1)
{
/* lecture d'une requête */
```

```
if((l=msgrcv(msqid,&la_requete,sizeof(struct requetem)-4,2,0))==-1)
  {
   perror("msgrcv");
   exit(2);
  }
la_reponse.res = la_requete.nb1 * la_requete.nb2 * la_requete.nb3;
printf("resultat %f, %d", la_reponse.res, la_reponse.res);
la_reponse.letype=la_requete.mon_pid;
/* type associé au message; le pid du client  */
if(msgsnd(msqid,&la_reponse,sizeof(struct reponsem) - 4,0)==-1)
  {
   perror("msgsnd");
   exit(2);
  }
}
exit(0);
}
```

7.5 Tubes anonymes

1) Il faut un couple de tubes par paire de communiquants, c'est-à-dire entre le père et chacun de ses fils. La difficulté vient du fait que les opérations de lecture pour les tubes sont bloquantes. Le père attend des données sur plusieurs tubes en même temps sans savoir a priori quel tube sera prêt en premier.

2)

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

main()
{
int tub11[2], tub21[2], tub12[2], tub22[2];
int pid1, pid2, nb1, nb2, retour, res;
char ent1[2], ent2[2];
fd_set ens;

pipe (tub11); /*du fils vers le père */
pipe (tub21); /* du père vers le fils */

pid1 = fork(); /* création du premier fils */
if (pid1 == 0)
{
  close (tub11[0]); close (tub21[1]);
  while(1){
  printf ("fils 1 Donnez moi deux nombres à additionner:");
  scanf ("%d %d", &nb1, &nb2);
  sprintf (ent1, "%d", nb1);
  sprintf (ent2, "%d", nb2);
  write (tub11[1], ent1, 2);  write (tub11[1], ent2, 2);
  read (tub21[0], ent2, 2);
```

```
  printf ("le resultat est %d\n", atoi(ent2)); }
  close (tub11[1]); close (tub21[0]);
  exit(0);
}
else
    pipe (tub12); /* du fils vers le père */
    pipe (tub22); /* du père vers le fils */

    pid2 = fork(); /* création du deuxième fils */
    if (pid2 == 0)
    {
     close (tub12[0]); close (tub22[1]);
     while(1) {
     printf ("fils 2 Donnez moi deux nombres à additionner:");
     scanf ("%d %d", &nb1, &nb2);
     sprintf (ent1, "%d", nb1);
     sprintf (ent2, "%d", nb2);
     write (tub12[1], ent1, 2);
     write (tub12[1], ent2, 2);
     read (tub22[0], ent1, 2);
     printf ("le resultat est %d\n", atoi(ent1)); }
     close (tub12[1]); close (tub22[0]);
     exit(0);
    }
    else
    {
     close (tub12[1]); close (tub22[0]);
     close (tub11[1]); close (tub21[0]);
     while (1){
     FD_ZERO(&ens);  /* utilisation de select() */
     FD_SET(tub12[0], &ens);
     FD_SET(tub11[0], &ens);
     retour = select (tub12[0] + 1, &ens, NULL, NULL, NULL);
     if (FD_ISSET(tub12[0], &ens))
        {
          read (tub12[0], ent1, 2);
          read (tub12[0], ent2, 2);
          res = atoi(ent1) + atoi (ent2);
          sprintf (ent1, "%d", res);
          write (tub22[1], ent1, 2);
        }
     if (FD_ISSET(tub11[0], &ens))
        {
          read (tub11[0] , ent1, 2);
          read (tub11[0], ent2, 2);
          res = atoi(ent1) + atoi (ent2);
          sprintf (ent1, "%d", res);
          write (tub21[1], ent1, 2);
        }
      }
```

```
        close (tub12[0]); close (tub22[1];
        close (tub11[0]); close (tub21[1];
        wait();
    }
```

7.6 Tubes anonymes et signaux

Le processus fils et le processus père ferment le descripteur en lecture du tube. Lorsque le père effectue son écriture dans le tube, il n'y a plus de lecteurs sur celui-ci. Le système lève donc une trappe et envoie le signal `SIGPIPE` au processus père écrivain. Le *handler* associé à ce signal (`attention_tube`) est donc exécuté.

7.7 Tubes nommés

La solution de l'exercice est donnée par l'exemple du paragraphe 7.1.2, communication bidirectionnelle. Le processus client est un peu différent et dispose de deux paires de tubes. Selon le service demandé, il écrit dans l'un des deux tubes de chaque paire et attend la réponse sur l'autre tube de la paire. Chaque serveur est conforme au processus serveur de ce paragraphe (le deuxième serveur est identique à ceci près qu'il multiplie trois nombres flottants).

8 Synchronisation entre processus – Interblocage

PLAN

OBJECTIFS

- Ce chapitre présente les grands schémas de synchronisation entre processus que sont l'exclusion mutuelle, les lecteurs-rédacteurs ainsi que les producteurs-consommateurs et il explique la mise en œuvre de ces schémas à l'aide de l'outil sémaphore. L'implémentation faite sous Linux de cet outil est détaillée.
- Nous terminons en abordant le problème de l'interblocage, puis en présentant succinctement la façon dont la synchronisation est assurée entre les chemins de contrôle du noyau.

8.1 Les grands schémas de synchronisation

Dans un système multiprocessus, l'ordonnanceur alloue le processeur à chaque processus selon un algorithme d'ordonnancement : la politique choisie conditionne l'ordre d'exécution des processus et très souvent, les exécutions des processus s'entrelacent les unes avec les autres. Chaque processus dispose d'un espace d'adressage propre et indépendant, protégé par rapport aux autres processus. Malgré tout, les processus peuvent avoir besoin de communiquer entre eux pour échanger des données. Comme nous l'avons vu au chapitre précédent, cet échange de données peut se faire par le biais d'une zone de mémoire partagée, par le biais d'un fichier ou encore en utilisant les outils de communication offerts par le système d'exploitation. Dans tous ces cas, les processus ne sont plus indépendants et ils effectuent des accès concurrents aux ressources logicielles que sont la mémoire partagée, le fichier ou encore les outils de communication.

Plus généralement, une *ressource* désigne toute entité dont a besoin un processus pour s'exécuter. La ressource peut être matérielle comme le processeur ou un périphérique ou elle peut être logicielle comme une variable.

Une ressource est caractérisée par un état qui définit si la ressource est libre ou occupée et par son *nombre de points d'accès*, c'est-à-dire le nombre de processus pouvant l'utiliser en même temps. Notamment, on distinguera la notion de *ressource critique* qui correspond à une ressource ne pouvant être utilisée que par un seul processus à la fois. Un processeur par exemple correspond à une ressource critique ; en effet, il n'est pas possible d'allouer celui-ci à deux processus simultanément. Une imprimante est un autre exemple d'une telle ressource.

L'utilisation d'une ressource par un processus s'effectue en trois étapes. La première étape correspond à l'étape d'*allocation* de la ressource au processus qui consiste à donner la ressource au processus qui la demande. Une fois que le processus a pu obtenir la ressource, il utilise la ressource durant un certain temps (deuxième étape) puis il rend la ressource : c'est la dernière étape de *restitution* de la ressource. Les phases d'allocation et de restitution d'une ressource doivent assurer que la ressource est utilisée conformément à son nombre de points d'accès. Par ailleurs, l'étape d'allocation de la ressource peut se révéler bloquante pour le processus qui l'effectue si tous les points d'accès de la ressource demandée sont occupés. Au contraire, l'étape de restitution d'une ressource par un processus peut entraîner le déblocage d'un autre processus en attente d'accès à cette ressource.

Plusieurs schémas de synchronisation entre processus ont été définis afin de garantir une bonne utilisation des ressources par les processus et d'une manière plus générale une communication entre processus cohérente et sans perte de données. Ces schémas sont :

- l'exclusion mutuelle ;
- l'allocation de ressources ;
- les producteurs-consommateurs ;
- les lecteurs-rédacteurs.

8.1.1 L'exclusion mutuelle

a) Un exemple simple

Nous allons mettre en lumière ce premier problème de synchronisation à l'aide d'un exemple simple. Pour cela, considérons donc le petit programme de réservation de place (dans un avion, un train, …) suivant :

```
Reservation {
if (nbplace > 0)
         /*Réserver une place */
          nbplace = nbplace - 1;
}
```

Le programme `Reservation` peut être exécuté par plusieurs processus à la fois (autrement dit, le programme est réentrant). La variable `nbplace`, qui représente le

nombre de place restant dans l'avion par exemple, est ici une variable d'état du système (de réservation).

On considère l'exécution de deux processus Client_1 et Client_2, alors que `nbplace` est égale à 1. Client_1 s'exécute en premier, et entame une réservation. Maintenant, l'exécution de Client_1 est commutée par l'ordonnanceur juste après le test de la valeur de la variable `nbplace` (`nbplace` = 1). Client_2 s'exécute à son tour, teste `nbplace` qu'il trouve également égale à 1 et donc effectue une réservation en décrémentant de une unité la variable `nbplace`. `nbplace` devient égale à 0. Comme le processus Client_2 a terminé son exécution, Client_1 reprend la main. Comme Client_1 avait trouvé la variable `nbplace` comme étant égale à 1 juste avant d'être commuté, il continue son exécution en décrémentant à son tour `nbplace`. De ce fait, `nbplace` devient égale à – 1 ce qui est incohérent ! Une même place a été allouée à deux clients différents !

b) Ressource critique, section critique et exclusion mutuelle

La variable `nbplace` doit être accédée par un seul processus à la fois pour que sa valeur reste cohérente : ici en l'occurrence le processus Client_1 qui a commencé la réservation en premier doit terminer son opération avant que le processus Client_2 puisse commencer à s'exécuter. Ainsi, Client_2 trouvera la variable `nbplace` égale à 0 lors du test et n'effectuera pas de réservation. `nbplace` constitue donc une *ressource critique*.

Le code d'utilisation d'une ressource critique est appelé *section critique*. La section critique doit au moins offrir la propriété essentielle de l'*exclusion mutuelle* qui garantit que la ressource critique manipulée durant la section critique ne peut effectivement être utilisée que par un seul processus à la fois. Pour ce faire, la section critique est précédée par un prélude et suivie d'un postlude qui assurent cette propriété d'exclusion mutuelle. Plus précisément, la section critique doit offrir les trois propriétés suivantes :

- la propriété d'exclusion mutuelle ;
- la propriété d'attente bornée : un processus demandant l'accès à la section critique doit obtenir satisfaction au bout d'un temps borné ;
- la propriété de bon déroulement : lorsque la section critique est vide et qu'un ou plusieurs processus sont en attente pour entrer dans celle-ci, le choix du processus entrant finalement en section critique ne relève que des processus en attente. Ce choix doit se faire dans un laps de temps borné. Par ailleurs, un processus s'exécutant en dehors de la section critique ne peut pas bloquer l'accès à celle-ci.

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

c) Réalisation matérielle d'une section critique

Utilisation des opérations de masquage/démasquage des interruptions matérielles

Nous rappelons que le mécanisme sous-jacent à l'ordonnancement des processus peut être la survenue d'une interruption horloge. Aussi une solution pour réaliser l'exclusion mutuelle est de masquer les interruptions dans le prélude et de les démasquer dans le postlude. Ainsi les interruptions sont masquées dès qu'un processus accède à la ressource critique et aucun événement susceptible d'activer un autre processus ne peut être pris en compte.

```
Reservation{
disable_interrupt; /*prélude de section critique */
if (nbplace > 0)
          nbplace = nbplace - 1;
enable_interrupt; /* postlude de section critique */
}
```

Cependant, cette solution est moyennement satisfaisante car elle empêche l'exécution de tous les processus y compris ceux ne désirant pas accéder à la ressource critique. De plus, le masquage et le démasquage des interruptions sont des opérations réservées au mode superviseur et ne sont donc pas accessibles pour les processus utilisateurs. Ce mode de réalisation d'une section critique est donc uniquement utilisé dans des parties de code sensibles du système d'exploitation, comme par exemple, la manipulation des files d'attente de l'ordonnanceur ou la protection des tampons du buffer cache.

Test and Set

Le *Test and Set* est une instruction matérielle dont l'exécution est atomique, c'est-à-dire qu'elle s'exécute en une seule fois sans pouvoir être interrompue. La logique du *Test and Set* est la suivante :

```
#define VRAI    1
#define FAUX    0
int function TS (int *verrou)
{
disable_interrupt;
int test;
     test = *verrou;
     *verrou = VRAI;
     return(test);
enable_interrupt;
}
```

À l'aide de cette instruction, une exclusion mutuelle est réalisée de la manière suivante avec la variable globale `cadenas` initialisée à faux.

```
/* prélude */
while TS(cadenas);
section critique
```

```
/* postlude */
cadenas = FAUX;
```

Le premier processus désirant entrer en section critique trouve la variable `cadenas` égale à faux et la positionne à vrai par le biais de la fonction *Test and Set*. Un second processus souhaitant alors lui-même entrer en section critique reste à attendre à l'entrée de la section critique puisque l'appel à la fonction *Test and Set* lui envoie la valeur vrai. Il reste ainsi en attente active jusqu'à ce que le premier processus libère la section critique et repositionne la variable `cadenas` à faux.

Cette solution ne garantit pas l'équité d'accès à la section critique.

Swap

L'instruction *swap* est une seconde instruction matérielle permettant de réaliser une exclusion mutuelle. Cette instruction inverse atomiquement le contenu de deux variables.

```
int function swap (int *a, int *b)
{
disable_interrupt;
int temp;
     temp = *b;
     *b = *a;
     *a = temp;
enable_interrupt;
}
```

L'exclusion mutuelle est réalisée de la manière suivante en utilisant une variable globale `verrou` initialisée à faux :

```
#define VRAI    1
#define FAUX    0

/* prélude */
cle = VRAI;
while (cle == VRAI)
   swap (verrou, cle);
section critique
/* postlude */
verrou = FAUX;
```

Le premier processus désirant entrer en section critique positionne la variable `cle` à vrai puis rentre dans la boucle et exécute l'instruction `swap(verrou, cle)`. La variable `cle` devient donc égale à faux tandis que `verrou` devient égale à vrai. Un second processus souhaitant alors lui-même entrer en section critique positionne à son tour sa variable `cle` à vrai et exécute l'instruction `swap(verrou, cle)`. Cette fois, la variable `verrou` étant à vrai, la variable `cle` reste également égale à vrai. De ce fait, le second processus entre en attente active dans la boucle `while`. Il sort de cette boucle, seulement lorsque le premier processus quitte la section critique et remet la variable `verrou` à faux.

d) Réalisation logicielle d'une section critique

La solution suivante, appelée algorithme de Peterson, est une solution entièrement logicielle ne requérant aucune aide ni du matériel, ni du système d'exploitation pour réaliser l'exclusion mutuelle entre deux processus *i* et *j*. La structure du processus *i* est la suivante :

```
/* prelude */
while{
drapeau[i] = VRAI; tour = j;
while (drapeau(j) & tour == i);
section critique
/* postlude */
drapeau(i) = FAUX;
}
```

La variable `drapeau` permet à chaque processus d'indiquer sa volonté d'entrer en section critique en positionnant celle-ci à vrai. La variable `tour` indique quel processus a le droit d'entrer en section critique. Ce sont deux variables globales.

On peut vérifier facilement que les trois propriétés que doit posséder une section critique sont bien réalisées :

- **exclusion mutuelle :** si les deux processus veulent entrer en section critique, alors `drapeau(i) = drapeau (j) = vrai`. Seul le processus pour lequel `tour` contient sa valeur peut entrer en section critique. Si l'un des processus est déjà en section critique (par exemple *i*) et que l'autre processus (*j*) veut y entrer à son tour, alors la variable `tour` vaut forcément *i* et le processus *j* est bloqué ;
- **attente bornée :** si l'un des deux processus (*i* par exemple) veut entrer en section critique et se trouve mis en attente, alors cela veut dire que la section critique est occupée par l'autre processus (*j*). Lorsque le processus *j* sort de la section critique, il positionne son drapeau à faux. Si ce même processus revient immédiatement demander l'entrée de la section critique, il met son drapeau à vrai mais également la variable `tour` à *i*. Comme le processus *i* en attente dans la boucle ne modifie pas la variable `tour`, le processus i entre en section critique, après une attente qui au maximum est égale à une entrée du processus *j* en section critique ;
- **bon déroulement :** un processus en dehors de la section critique et ne désirant pas y pénétrer positionne son drapeau à faux. De ce fait, il ne peut pas bloquer l'entrée de la section critique. Si deux processus veulent entrer en même temps en section critique, tous deux positionnent leur drapeau à vrai et chacun commute le tour pour le donner à l'autre. Ainsi pour au moins l'un des deux processus, la condition du `while` est fausse et ce processus entre en section critique.

e) L'outil sémaphore

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

Présentation de l'outil sémaphore

Une sémaphore `sem` est une structure système composée d'une file d'attente *L* de processus et d'un compteur *K*, appelé niveau du sémaphore, et contenant initialement une valeur val. Cette structure ne peut être manipulée que par trois opérations `P(sem)`, `V(sem)` et `Init(sem, val)`. Ces trois opérations sont des opérations indivisibles, c'est-à-dire que l'exécution de ces opérations s'effectue interruptions masquées et ne peut donc pas être interrompue. Un outil sémaphore peut être assimilé à un distributeur de jetons dont le nombre initial est fixé par l'opération `Init()`. L'acquisition d'un jeton par un processus donne le droit à ce processus de poursuivre son exécution. Sinon, le processus est bloqué.

L'opération `Init (sem, val)`

L'opération `Init(sem,val)` a pour but d'initialiser le sémaphore, c'est-à-dire qu'elle met à vide la file d'attente *L* et initialise avec la valeur *val* le compteur *K*. On définit ainsi le nombre de jetons initiaux dans le sémaphore.

```
function Init(semaphore sem, int val)
{
disable_interrupt;
sem.K = val;
sem.L:= NULL;
enable_interrupt;
}
```

L'opération `P(sem)`

L'opération `P(sem)` attribue un jeton au processus appelant si il en reste au moins un et sinon bloque le processus dans la file `sem.L`. L'opération P est donc une opération éventuellement bloquante pour le processus élu qui l'effectue. Dans le cas du blocage, il y a réordonnancement et un nouveau processus prêt est élu. Concrètement, le compteur *K* du sémaphore est décrémenté de une unité. Si la valeur du compteur devient négative, le processus est bloqué et intègre une file d'attente de l'ordonnanceur.

```
function P(semaphore sem)
{
disable_interrupt;
sem.K = sem.K - 1;
if (sem.K < 0)
{
          L.suivant = processus_courant;
          processus_courant.state = bloque;
          reordonnancement = vrai;
}
enable_interrupt;
}
```

L'opération `V(sem)`

L'opération `V(sem)` a pour but de rendre un jeton au sémaphore. De plus, si il y a au moins un processus bloqué dans la file d'attente *L* du sémaphore, un processus est réveillé. La gestion des réveils s'effectue généralement en mode FIFO, c'est-à-dire que c'est le processus le plus anciennement endormi qui est réveillé. L'opération *V* est une opération qui n'est jamais bloquante pour le processus qui l'effectue.

```
function V(semaphore sem)
{
disable_interrupt;
sem.K = sem.K + 1;
if (sem.K <= 0) /* il y a au moins un processus dans la file */
{
         processus_reveille = L.tete;
         processus_reveille.state = pret;
         reordonnancement = vrai;
}
enable_interrupt;
}
```

Signification de la valeur du compteur K du sémaphore sem

La valeur du compteur *K* doit être interprétée comme suit :

- tant que la valeur du compteur *K* et positive ou nulle, elle représente le nombre d'opérations `P(sem)` encore passantes ;
- dès que la valeur du compteur *K* devient inférieure à 0, sa valeur absolue donne le nombre de processus présents dans la file d'attente *L* du sémaphore `sem`.

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

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

```
Init (Mutex, 1);
Reservation {
P(Mutex); /* prélude de section critique */
if (nbplace > 0)
          nbplace = nbplace - 1;
V(Mutex); /* postlude de section critique */
}
```

3.1.2 Le schéma de l'allocation de ressources

a) Présentation du schéma

Le schéma d'allocation de *N* exemplaires de ressources critiques à un ensemble de processus est une généralisation du schéma précédent, dans lequel on considère à présent que l'on a *N* sections critiques. Le sémaphore `Res` d'allocation de ressources

est initialisé au nombre d'exemplaires de ressources initialement disponibles, soit *N*, c'est-à-dire que l'on fait correspondre un jeton par exemplaire de ressources. Pour un processus, acquérir un exemplaire de ressource est alors synonyme d'acquérir le jeton correspondant, soit une opération `P(Res)`. Rendre la ressource est synonyme de rendre le jeton associé, soit une opération `V(Res)`.

```
Init (Res, N)
Allocation: P(Res);
      Utilisation de la ressource Res
Restitution: V(Res);
```

b) Interblocage

Considérons à présent la situation suivante pour laquelle deux processus P1 et P2 utilisent tout deux 2 ressources critiques R1 et R2 selon le code suivant. Les deux ressources R1 et R2 sont gérées par le biais de deux sémaphores SR1 et SR2 initialisés à 1.

```
Processus P1                        Processus P2
{                                   {
P(SR1);                             P(SR2);
P(SR2);                             P(SR1);
Utilisation de R1 et R2;            Utilisation de R1 et R2;
V(SR2);                             V(SR1);
V(SR1);                             V(SR2);
}                                   }
```

Initialement, les deux ressources R1 et R2 sont libres. Le processus P1 commence son exécution, demande à accéder à la ressource R1 en faisant l'opération `P(SR1)` et obtient R1 puisque la ressource est libre. Maintenant, le processus P1 est commuté par l'ordonnanceur au bénéfice du processus P2. Le processus P2 commence donc à s'exécuter, demande à accéder à la ressource R2 en faisant l'opération `P(SR2)` et obtient R2 puisque la ressource est libre. Le processus P2 poursuit son exécution et demande à présent à accéder à la ressource R1 en effectuant l'opération `P(SR1)`. Cette fois, le processus P2 est stoppé puisque R1 a été allouée au processus P1. Le processus P2 passe donc dans l'état bloqué; l'ordonnanceur est invoqué et celui-ci réélit le processus P1. Le processus P1 reprend son exécution et demande à présent à accéder à la ressource R2 en effectuant l'opération `P(SR2)`. Le processus P1 est également stoppé puisque R2 a été allouée au processus P2.

La situation dans laquelle les processus P1 et P2 se trouvent maintenant est donc la suivante : le processus P1 possède la ressource R1 et attend la ressource R2 détenue par le processus P2. Pour que le processus P1 puisse reprendre son exécution, il faut que le processus P2 libère la ressource R2 en effectuant l'opération `V(SR2)`. Mais le processus P2 ne peut lui-même pas poursuivre son exécution puisqu'il est bloqué en attente de la ressource R1 possédée par le processus P1. On voit donc clairement que les deux processus P1 et P2 sont bloqués dans l'attente l'un de l'autre et que rien ne peut les sortir de cette attente.

Une telle situation est qualifiée d'interblocage. Elle correspond à une situation où au moins deux processus, possédant déjà un ensemble de ressources, sont en attente infinie les uns des autres pour l'acquisition d'autres ressources. Une telle situation, lorsqu'elle se produit, ne peut être résolue que par la destruction des processus impliqués dans l'interblocage.

Une façon d'éviter l'occurrence d'une telle situation est de veiller à ce que les processus utilisant les mêmes ressources demandent celles-ci dans le même ordre. Ici, donc les opérations `P(SR1)` et `P(SR2)` pour le processus P2 doivent être inversées.

Reprenons maintenant l'exemple de communication bidirectionnelle programmée au chapitre 7 à l'aide de tubes nommés. Dans cet exemple, un processus serveur communique avec un processus client au travers de deux tubes nommés `tube1` et `tube2`. Le code initial est :

```
CLIENT
/* ouverture du tube tube1 en ecriture */
tub1 = open ("tube1", O_WRONLY);
/* ouverture du tube tube2 en lecture */
tub2 = open ("tube2", O_RDONLY);
```

```
SERVEUR
/* ouverture du tube 1 en lecture */
tub1 = open ("tube1", O_RDONLY);
/*ouverture du tube 2 en ecriture */
tub2 = open ("tube2", O_WRONLY);
```

Les primitives d'ouverture de tube nommé sont bloquantes. Ici, le processus client est bloqué sur l'opération d'ouverture en écriture du tube `tube1` jusqu'à ce que le serveur ouvre lui-même le tube `tube1` en lecture. De même le processus client est ensuite bloqué sur l'opération d'ouverture en lecture du tube `tube2` jusqu'à ce que le serveur ouvre lui-même le tube `tube2` en écriture.

Imaginons à présent que nous inversions les deux opérations d'ouverture de tube du côté du serveur. Nous écrivons donc :

```
SERVEUR
/*ouverture du tube 2 en ecriture */
tub2 = open ("tube2", O_WRONLY);
/* ouverture du tube 1 en lecture */
tub1 = open ("tube1", O_RDONLY);
```

Les deux processus sont maintenant en interblocage.

Ce problème de l'interblocage et les solutions qui y sont apportées sont traités plus en détail au paragraphe 8.4

8.1.3 Le schéma lecteurs-rédacteurs

Dans ce problème, on considère la situation où un fichier ou une zone de mémoire commune est accédée simultanément en lecture et en écriture. Il y a donc deux types de processus, d'une part des processus lecteurs et d'autre part des processus rédacteurs.

Dans ce schéma, deux objectifs doivent être atteints. D'un côté, le contenu du fichier ou de la zone de mémoire commune doit évidemment rester cohérent donc les écritures ne doivent pas avoir lieu en même temps. Par ailleurs, on souhaite que les lecteurs lisent toujours une information stable, c'est-à-dire que les lecteurs ne doivent pas lire une information en cours de modification.

Pour parvenir à réaliser ces deux objectifs, il faut, sur la zone de mémoire commune ou sur le fichier partagé :

- soit une seule écriture en cours ;
- soit une ou plusieurs lectures en cours, les lectures simultanées ne gênant pas puisqu'une lecture ne modifie pas le contenu de l'information.

a) Le rédacteur

Un processus rédacteur exclut donc les autres rédacteurs ainsi que tous les lecteurs. Autrement dit, un rédacteur accède toujours seul au fichier ou à la zone de mémoire commune, c'est-à-dire qu'il effectue des accès en exclusion mutuelle des autres rédacteurs et des lecteurs.

L'accès du processus rédacteur peut donc être géré selon le schéma d'exclusion mutuelle vu précédemment, en utilisant un sémaphore appelé ici `Acces` et initialisé à 1.

```
Init (Acces, 1);
Rédacteur {
/* M'assurer que l'accès au fichier est libre */ P(Acces);
                                          accès en écriture
/* Libérer l'accès au fichier*/                V(Acces);
}
```

b) Le lecteur

Un lecteur exclut les rédacteurs mais pas les autres lecteurs. Seul le premier lecteur doit donc s'assurer qu'il n'y a pas d'accès en écriture en cours. Par ailleurs, seul le dernier lecteur doit réveiller un éventuel rédacteur.

Il faut en conséquence compter le nombre de lecteurs qui accèdent au fichier ou à la zone de mémoire partagée. On utilise pour cela une variable `NL`, initialisée à 0. Cette variable `NL` est accédée en concurrence par tous les lecteurs qui vont soit incrémenter cette variable (un lecteur de plus), soit la décrémenter (un lecteur de moins). Pour que le contenu de la variable reste cohérent, il faut que `NL` soit accédée en exclusion mutuelle. L'accès à la variable est donc gardé par un sémaphore `Mutex` initialisé à 1.

```
Init (Mutex,1)
Lecteur{
/* accès à NL pour compter un lecteur de plus */
{
P(Mutex);
NL:= NL + 1;
if(NL == 1)
```

```
/* je suis le premier lecteur; je me place en exclusion mutuelle d'un
rédacteur éventuel */
         P(Acces);
V(Mutex);
accès en lecture
P(Mutex);
/* accès à NL pour compter un lecteur de moins */
NL = NL - 1;
if (NL == 0)
/* je suis le dernier lecteur; je réveille d'un rédacteur éventuel en
attente */
         V(Acces);
V(Mutex);
}
```

La solution que nous donnons ici autorise la *coalition* des lecteurs contre les rédacteurs, c'est-à-dire que les lecteurs peuvent monopoliser l'accès à la ressource fichier ou à la zone de mémoire partagée et laisser les rédacteurs en attente.

c) Utilisation de ce schéma au niveau du système d'exploitation

Le système d'exploitation Linux offre deux types de verrous permettant à des processus de synchroniser leurs accès à des fichiers :

- les verrous partagés ou verrous en lecture peuvent être détenus par plusieurs processus à la fois. Ils sont acquis par les processus lecteurs souhaitant se protéger d'un éventuel rédacteur ;
- les verrous exclusifs ou verrous en écriture ne peuvent être détenus que par un seul processus à la fois. Ils sont acquis par des processus rédacteurs souhaitant se protéger des lecteurs et d'éventuels autres rédacteurs.

La primitive `flock()` permet ainsi à un processus d'acquérir un verrou sur un fichier entier. Le prototype de cette fonction est :

```
#include <sys/file.h>
int flock (int fd, int operation);
```

Le paramètre `operation` définit l'opération à réaliser sur le fichier désigné par le descripteur `fd`. Ce paramètre peut prendre les valeurs suivantes :

- `LOCK_SH`, demande de verrou partagé ;
- `LOCK_EK`, demande de verrou exclusif ;
- `LOCK_UN`, libération du verrou exclusif ou partagé.

Par ailleurs, la primitive `lockf()` permet ainsi à un processus d'acquérir un verrou sur une partie de fichier. Le prototype de cette fonction est :

```
#include <fcntl.h>
#incluse <unistd.h>
int lockf (int fd, int operation, off_t taille);
```

Le paramètre `operation` définit l'opération à réaliser sur le fichier désigné par le descripteur `fd`. Seule une partie du fichier est concernée par le verrouillage, la paramètre `taille` représente la taille de cette partie depuis la position courante dans le fichier. Le paramètre `operation` peut prendre les valeurs suivantes :

- `F_ULOCK`, déverrouillage de la section ;
- `F_TEST`, test de la présence d'un verrou et retour de l'erreur `EACCESS` si un verrou est positionné ;
- `F_TLOCK`, verrouillage de la section spécifiée et retour de l'erreur `EACESS` si le verrou ne peut pas être immédiatement posé ;
- `F_LOCK`, verrouillage de la section spécifiée avec endormissement éventuel du processus poseur de verrou pour attendre l'acquisition effective du verrou.

8.1.4 Le schéma producteurs-consommateurs

a) Présentation du problème 1 producteur/1 consommateur

On considère maintenant deux processus communiquant par un tampon de *N* cases. D'un côté, un processus producteur produit des messages qu'il dépose dans la case du tampon pointée par l'index de dépôt *i.* De l'autre côté, un processus consommateur prélève les messages déposés par le processus producteur dans la case pointée par l'index de retrait *j*. Le tampon est géré selon un mode FIFO circulaire en consommation et en production, c'est-à-dire que :

- le producteur dépose les messages depuis la case 0 jusqu'à la case *N*–1, puis revient à la case 0 ;
- le consommateur prélève les messages depuis la case 0 jusqu'à la case *N*–1, puis revient à la case 0.

Le processus producteur et le processus consommateur sont totalement indépendants dans le sens où ils s'exécutent chacun à leur propre vitesse. Dans ce cas, pour qu'aucun message ne soit perdu, les trois règles suivantes doivent être respectées :

- le producteur ne doit pas produire si le tampon est plein ;
- le consommateur ne doit pas faire de retrait si le tampon est vide ;
- le producteur et le consommateur ne doivent jamais travailler dans une même case.

b) Une solution au problème

Dans ce problème, deux types de ressources distinctes peuvent être mises en avant :

- le producteur consomme des cases vides et fournit des cases pleines ;
- le consommateur consomme des cases pleines et fournit des cases vides.

Il y a donc des ressources cases vides et des ressources cases pleines. Le problème peut maintenant se rapprocher d'un problème d'allocation de ressources critiques tel que nous l'avons vu précédemment.

On associe donc un sémaphore à chacune des ressources identifiées, initialisé au nombre de cases respectivement vides ou pleines initialement disponibles (*N* et 0). Soient donc les deux sémaphores `Vide` initialisé à *N* et `Plein` initialisé à 0.

Le producteur s'alloue une case vide par une opération `P(Vide)`, remplit cette case vide et de ce fait génère une case pleine qu'il signale par une opération `V(Plein)`. Cette opération réveille éventuellement le consommateur en attente d'une case pleine.

```
Producteur
{
int i = 0;
P(Vide);
tampon(i)= message;
i = i + 1 mod N;
V(Plein);
}
```

Le consommateur s'alloue une case pleine par une opération `P(Plein)`, vide cette case pleine et de ce fait génère une case vide qu'il signale par une opération `V(Vide)`. Cette opération réveille éventuellement le producteur en attente d'une case vide.

```
Consommateur
{
int j = 0;
P(Plein);
message = tampon(j);
j: = j + 1 mod N;
V(Vide);
}
```

c) Extension du problème à n producteurs et n consommateurs

Nous étendons le problème précédent en considérant à présent un ensemble de n producteurs et de n consommateurs qui s'échangent des messages *via* un même tampon de données de *N* cases. Comme précédemment, un processus producteur produit des messages qu'il dépose dans la case du tampon pointée par l'index de dépôt *i*. De l'autre côté, un processus consommateur prélève les messages déposés par le processus producteur dans la case pointée par l'index de retrait *j*.

Nous voyons maintenant que l'extension du problème à *n* producteurs et *n* consommateurs ajoute un problème de concurrence d'une part pour l'accès à l'index de dépôt i entre tous les producteurs, d'autre part pour l'accès à l'index de retrait *j* entre tous les consommateurs.

Ainsi, l'index *i* de dépôt doit être accédé en exclusion mutuelle par tous les producteurs tandis que les consommateurs doivent accéder à l'index *j* de retrait également en exclusion mutuelle.

La solution précédente s'enrichit donc de deux sections critiques gardant l'accès à chacun des index *i* et *j*, sections critiques réalisées par le biais de deux sémaphores `Mutexi` et `Mutexj`, initialisés tout deux à la valeur 1.

```
Déclarations globales:
                int i, j = 0;
                semaphore Mutexi, Mutexj, Vide, Plein;
  Initialisations:
     Init(Mutexi, 1); Init(Mutexj, 1); Init(Vide, N); Init(Plein, 0);
```

Producteur	Consommateur
{	{
P(Vide);	P(Plein);
P(Mutexi);	P(Mutexj);
Tampon(i)= message;	Message = Tampon(j);
i: = i + 1 mod N;	j: = j + 1 mod N;
V(Mutexi);	V(Mutexj);
V(Plein);	V(Vide);
}	}

d) Utilisation de ce schéma au niveau du système d'exploitation

Les systèmes d'exploitation offrent directement des outils de communication mettant en œuvre au niveau des primitives d'accès à ces outils, le schéma producteurs-consommateurs.

Le système Linux par exemple offre deux outils de ce type que nous avons étudiés au chapitre 7 :

- d'une part, les tubes qui permettent une communication en mode flux d'octets entre processus ;
- d'autre part, les files de messages (*Messages queues* (*MSQ*)) qui permettent à n'importe quel processus connaissant l'identifiant de la file de communiquer avec d'autres processus, en mode paquet.

8.2 Utilisation des sémaphores sous Linux

Les sémaphores appartiennent à la famille des IPC. Leur implémentation sous Linux permet de réaliser les deux opérations *P* et *V*, non pas seulement sur un seul sémaphore, mais sur un ensemble de sémaphores défini dans un tableau, et ce de manière atomique, c'est-à-dire que le processus ne peut poursuivre son exécution que si l'ensemble des opérations *P* ou *V* demandées a pu être réalisé. Sinon, le processus s'endort. Cette solution permet d'éviter les situations d'interblocage telles que nous les avons décrites au paragraphe 8.1.2.

Tout comme les autres IPC vus au chapitre précédent, un ensemble de sémaphores est identifié au sein du système Linux par une clef.

Par ailleurs l'implémentation Linux des sémaphores ajoute une troisième opération par rapport aux opérations *P* et *V*, l'opération ATT qui permet à un processus d'attendre que la valeur d'un sémaphore soit devenue nulle.

Chaque sémaphore d'un ensemble de sémaphores est représenté par une structure `semaphore` définie comme suit dans le fichier `<asm/semaphore.h>` :

```
struct semaphore {
     atomic_t count;               /* compteur du sémaphore */
     int sleepers;                 /* nombre de processus endormis
                                      sur le sémaphore */
      wait_queue_head_t wait; }    /* file d'attente des processus
                                      endormis sur le sémaphore */
};
```

8.2.1 Création et recherche d'un ensemble de sémaphores

La primitive `semget()` permet à un processus soit de créer un nouvel ensemble de sémaphores, soit de récupérer l'identificateur d'un groupe de sémaphores déjà existant. Le prototype de la fonction est :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t cle, int nsems, int semflg);
```

Les paramètres `cle` et `semflg` ont la même signification que pour les opérations équivalentes portant sur les files de messages et les segments de mémoire partagée. Le paramètre `nsems` indique le nombre de sémaphores à créer dans le groupe. La fonction retourne un identificateur sur le groupe créé en cas de succès, et la valeur – 1 sinon. La variable `ERRNO` peut alors prendre les valeurs suivantes :

- `EACCESS`, le processus ne dispose pas des droits appropriés ;
- `EEXIST`, une ressource sémaphore de même numéro de clé que celle dont la création est demandée existe déjà ;
- `EIDRM`, l'ensemble de sémaphores a été détruit ;
- `ENOENT`, l'ensemble de sémaphores identifié par la clé fournie n'existe pas et la création n'est pas demandée ;
- `ENOMEM`, l'espace mémoire est saturé ;
- `ENOSPC`, le nombre maximal de ressources sémaphore ou d'ensembles de sémaphores est dépassé.

8.2.2 Opérations sur les sémaphores

a) Opérations *P*, *V* et *ATT*

Trois opérations *P*, *V* et *ATT* peuvent être réalisées sur un sémaphore.

La structure `sembuf` permet de décrire une opération à réaliser sur un sémaphore du tableau. Dans cette structure, le champ `sem_op` permet de coder le type d'opération souhaitée de la manière suivante :

- si `sem_op` est négatif, l'opération à réaliser est une opération *P* ;
- si `sem_op` est positif, l'opération à réaliser est une opération *V* ;

- si sem_op est nul, l'opération à réaliser est une opération *ATT*.

```
struct sembuf {
    unsigned short sem_num;/* numéro du semaphore dans le tableau */
    short sem_op; /* opération à réaliser sur le semaphore */
    short sem_flg; /* options */
};
```

Le champ sem_flg de la structure permet de définir deux options :

- IPC_NOWAIT, la réalisation de l'opération n'est pas bloquante, mais si elle n'a pas pu être réalisée, l'erreur EAGAIN est positionnée dans la variable errno ;
- SEM_UNDO, le noyau mémorise l'opération inverse de celle demandée, de manière à pouvoir annuler celle-ci si le processus se termine en erreur. Ainsi, à la fin du processus, toutes les opérations mémorisées sont automatiquement réalisées.

La primitive semop() permet de réaliser l'une des trois opérations décrites par une structure sembuf. Le premier paramètre de la fonction identifie le groupe de sémaphores concernés. Le second paramètre est une liste de structures sembuf décrivant l'ensemble des opérations *P*, *V* ou *ATT* à réaliser. Enfin, le dernier argument donne le nombre total d'opérations à effectuer. Toutes les opérations sont réalisées de manière atomique sauf si l'option IPC_NOWAIT a été positionnée pour une des opérations. Dans ce cas, aucune opération n'est réalisée si l'opération avec l'option positionnée ne peut pas être réalisée.

```
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/sem.h>
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

L'appel système retourne la valeur 0 en cas de succès. Dans le cas contraire, la valeur – 1 est retournée et la variable errno prend l'une des valeurs suivantes :

- EACCESS, le processus ne dispose pas des droits appropriés ;
- EEBIG, l'argument nsops est plus grand que le nombre maximal d'opérations autorisées par le système ;
- EIDRM, le sémaphore a été détruit ;
- EAGAIN, l'option IPC_NOWAIT est positionnée et l'opération n'a pas pu être réalisée immédiatement ;
- EFAULT, l'adresse spécifiée dans sops est invalide ;
- EFBIG, le numéro de sémaphore est incorrect pour une des opérations ;
- EINTR, le processus a reçu un signal alors qu'il attendait l'accès à un sémaphore ;
- EINVAL, l'ensemble de sémaphores demandé n'existe pas ou le nombre d'opérations à réaliser est négatif ou nul ;
- ENOMEM, l'option SEM_UNDO est positionnée mais l'espace mémoire est insuffisant pour créer la structure mémorisant l'opération inverse ;
- ERANGE, la valeur ajoutée au compteur du sémaphore fait dépasser la valeur maximale autorisée par le système.

b) Opération d'initialisation d'un sémaphore

L'initialisation du compteur à une valeur `arg` d'un sémaphore de numéro `semnum` dans l'ensemble identifié par l'identificateur `semid` s'effectue en utilisant la primitive `semctl()` avec le paramètre `cmd` fixé à la valeur `SETVAL`.

```
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/sem.h>
int semctl (int semid, int semnum, int cmd, union semun arg);
```

Ainsi l'opération `semctl (semid, 0, SETVAL, 3)` initialise à la valeur 3 le sémaphore 0 dans l'ensemble désigné par l'identifiant `semid`.

c) Destruction d'un sémaphore

La destruction d'un ensemble de sémaphores identifié par l'identificateur `semid` s'effectue en utilisant la primitive `semctl` avec le paramètre `cmd` fixé à la valeur `IPC_RMID` :

```
                semctl (semid, 0, IPC_RMID, 0);
```

8.2.3 Un exemple

Pour cet exemple, nous reprenons le programme du chapitre 2 illustrant l'utilisation des primitives associées à la gestion des threads. Comme nous l'avons expliqué au chapitre 2, les *threads* d'un même processus se partagent l'espace d'adressage de ce processus, c'est-à-dire le code exécutable mais aussi les données. Dans ce programme, le thread principal et le thread fils se partagent la variable i qu'ils incrémentent chacun de leur côté.

Nous avons modifié le programme du chapitre 2 de façon à ce que les incrémentations effectuées par le père d'une part et par le fils d'autre part, se fassent en exclusion mutuelle les unes des autres, c'est-à-dire que nous interdisons les exécutions pour lesquelles les incrémentations du père et du fils s'entrelacent. Ainsi, les traces d'exécution données pour ce programme au chapitre 2 ne peuvent plus se produire. Les seules traces d'exécution autorisées sont :

```
hello, thread principal 1000
hello, thread principal 3000
hello, thread fils 3010
hello, thread fils 3030
```

ou

```
hello, thread fils 10
hello, thread fils 30
hello, thread principal 1030
hello, thread principal 3030
/*********************************************************************/
/*                   Utilisation des sémaphores                      */
/*********************************************************************/
```

```
#include <stdio.h>
#include <pthread.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int i;
int semid;
struct sembuf operation;

void addition()
{
/* opération P */
operation.sem_num = 0;
operation.sem_op = -1;
operation.sem_flg = 0;
semop (semid, &operation, 1);
i = i + 10;
printf ("hello, thread fils %d\n", i);
i = i + 20;
printf ("hello, thread fils %d\n", i);
/* opération V */
operation.sem_num = 0;
operation.sem_op =  1;
operation.sem_flg = 0;
semop (semid, &operation, 1);
}

main()
{ pthread_t num_thread;

i = 0;
/* création d'un sémaphore initialisé à la valeur 1 */
if ((semid = semget (12, 1, IPC_CREAT|IPC_EXCL|0600)) == -1)
        perror ("pb semget");
if ((semctl (semid, 0, SETVAL, 1)) == -1)
        perror ("pb semctl");

if (pthread_create(&num_thread, NULL, (void *(*)())addition, NULL) == -1)
      perror ("pb fork\n");
/* opération P */
operation.sem_num = 0;
operation.sem_op = -1;
operation.sem_flg = 0;
semop (semid, &operation, 1);
i = i + 1000;
printf ("hello, thread principal %d\n", i);
i = i + 2000;
printf ("hello, thread principal %d\n", i);
/* opération V */
```

```
operation.sem_num = 0;
operation.sem_op =  1;
operation.sem_flg = 0;
semop (semid, &operation, 1);
pthread_join(num_thread, NULL);
semctl (semid, 0, IPC_RMID, 0)
}
```

8.3 Mutex et variables conditions

Les *mutex* et *variables conditions* sont deux outils de synchronisation permettant de réaliser l'exclusion mutuelle entre *threads.*

8.3.1 Mutex

Un *mutex* est une variable de type `pthread_mutext_t` servant de verrou pour protéger l'accès à des zones de codes ou de données particulières. Ce verrou peut prendre deux états, disponible ou verrouillé, et il ne peut être acquis que par un seul *thread* à la fois. Un *thread* demandant à verrouiller un mutex déjà acquis par un autre *thread* est mis en attente.

a) Initialisation d'un mutex

L'initialisation d'un mutex s'effectue à l'aide de la constante `PTHREAD_MUTEX_INITIALIZER` :

```
int pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
```

b) Verrouillage d'un mutex

Le verrouillage d'un mutex s'effectue en appelant la primitive `pthread_mutex_lock()` dont le prototype est :

```
int pthread_mutex_lock(pthread_mutex_t *mut);
```

Si le mutex est libre, alors il est attribué au *thread* appelant. Sinon, le *thread* appelant est mis en attente jusqu'à la libération du mutex.

La primitive `pthread_mutex_trylock()` effectue également le verrouillage d'un mutex. Cependant elle échoue avec l'erreur `EBUSY` lorsque le mutex est déjà verrouillé plutôt que de bloquer le *thread* appelant.

```
int pthread_mutex_trylock(pthread_mutex_t *mut);
```

c) Libération d'un mutex

La libération d'un mutex s'effectue en appelant la primitive `pthread_mutex_unlock()` dont le prototype est :

```
int pthread_mutex_unlock(pthread_mutex_t *mut);
```

d) Destruction d'un mutex

La destruction d'un mutex s'effectue à l'aide de la primitive `pthread_mutex_destroy()` dont le prototype est :

```
int pthread_mutex_destroy(pthread_mutex_t *mut);
```

e) Un exemple

Nous reprenons l'exemple du paragraphe 8.2.3 en remplaçant les sémaphores d'exclusion mutuelle par des mutex.

```
/*********************************************************************/
/*                      Utilisation des mutex                        */
/*********************************************************************/
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>>

int i;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;


void addition()
{
/* verrouillage du mutex */
pthread_mutex_lock (&mutex);
i = i + 10;
printf ("hello, thread fils %d\n", i);
i = i + 20;
printf ("hello, thread fils %d\n", i);
/* libération du mutex */
pthread_mutex_unlock (&mutex);
}

main()
{ pthread_t num_thread;
i = 0;
if (pthread_create(&num_thread, NULL, (void *(*)())addition, NULL) == -1)
perror ("pb fork\n");
/* verrouillage du mutex */
pthread_mutex_lock (&mutex);
i = i + 1000;
printf ("hello, thread principal %d\n", i);
i = i + 2000;
printf ("hello, thread principal %d\n", i);
/* libération du mutex */
pthread_mutex_unlock (&mutex);
pthread_join(num_thread, NULL);
pthread_mutex_destroy (&mutex);
}
```

8.3.2 Variables conditions

Les variables conditions sont des outils de type `pthread_cond_t` permettant à un processus de se mettre en attente d'un événement provenant d'un autre thread, comme par exemple la libération d'un mutex.

La variable condition est associée à deux opérations, l'une permettant l'attente d'une condition, l'autre permettant de signaler que la condition est remplie. La variable condition est toujours associée à un mutex qui la protège des accès concurrents. Son utilisation suit le protocole suivant :

Thread en attente de la condition	Thread signalant la condition
1. Initialisation de la condition et du mutex associé ; 2. Blocage du mutex et mise en attente sur la condition ; 3. Libération du mutex.	1. Travail jusqu'à réaliser la condition attendue ; 2. Blocage du mutex associé à la condition et signalisation de la condition remplie ; 3. Libération du mutex.

Il faut noter que la primitive effectuant la mise en attente libère atomiquement le mutex associé à la condition, ce qui permet au thread signalant la condition de l'acquérir à son tour (étape 2). Le verrou est de nouveau acquis par le thread en attente, une fois la condition attendue signalée. Plus précisément, l'étape 2 pour le processus attendant la condition peut être précisée comme suit :

- blocage du mutex ;
- mise en attente sur la condition et déblocage du mutex ;
- condition signalée, réveille du thread et blocage du mutex, une fois celui-ci libéré par le thread signalant la condition (étape 3).

a) Initialisation d'une variable condition

L'initialisation d'une variable condition s'effectue à l'aide de la constante `PTHREAD_COND_INITIALIZER` :

```
pthread_mutex_t condition = PTHREAD_COND_INITIALIZER;
```

b) Mise en attente sur une condition

La primitive `pthread_cond_wait()` permet à un thread de se mettre en attente sur une condition. Le prototype de cette primitive est :

```
pthread_cond_wait(pthread_cond_t *condition, pthread_mutex_t *mutex);
```

c) Signalisation d'une condition

La primitive `pthread_cond_signal()` permet à un thread de signaler une condition remplie. Le prototype de cette primitive est :

```
pthread_cond_signal(pthread_cond_t *condition);
```

d) Destruction d'une variable condition

La destruction d'une variable condition s'effectue à l'aide de la primitive `pthread_cond_destroy()` dont le prototype est :

```
int pthread_cond_destroy(pthread_cond_t *condition);
```

e) Un exemple

Nous reprenons l'exemple précédent du *thread* principal et du *thread* fils incrémentant chacun une variable commune *i.* Nous souhaitons à présent que le *thread* principal incrémente toujours la variable *i* avant le *thread* fils. Les seules traces d'exécution autorisées sont maintenant :

```
hello, thread principal 1000
hello, thread principal 3000
hello, thread fils 3010
hello, thread fils 3030
```

Le *thread* fils et le *thread* principal se synchronisent à l'aide d'une variable condition. Le *thread* fils se met en attente de la condition qui est signalée par le thread principal une fois qu'il a terminé d'incrémenter la variable *i.*

```
/*********************************************************************/
/*              Utilisation des variables condition                  */
/*********************************************************************/

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

int i;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condition = PTHREAD_COND_INITIALIZER;

void addition()
{
/* attente de la condition */
pthread_mutex_lock (&mutex);
pthread_cond_wait (&condition, &mutex);
pthread_mutex_unlock (&mutex);
i = i + 10;
printf ("hello, thread fils %d\n", i);
i = i + 20;
printf ("hello, thread fils %d\n", i);
}

main()
{ pthread_t num_thread;
i = 0;
if (pthread_create(&num_thread, NULL, (void *(*)())addition, NULL) == -1)
      perror ("pb fork\n");
/* verrouillage du mutex */
```

```
pthread_mutex_lock (&mutex);
i = i + 1000;
printf ("hello, thread principal %d\n", i);
i = i + 2000;
printf ("hello, thread principal %d\n", i);
/* signalisation de la condition et libération du mutex */
pthread_cond_signal (&condition);
pthread_mutex_unlock (&mutex);
pthread_join(num_thread, NULL);
pthread_mutex_destroy (&mutex);
pthread_cond_destroy (&condition);
}
```

8.4 INTERBLOCAGE

8.4.1 Les conditions nécessaires à l'obtention d'un interblocage

Nous avons déjà évoqué au paragraphe 8.1.2. ce qu'était une situation d'interblocage pour un ensemble de processus. Ainsi, un ensemble de *n* processus est dit en situation d'interblocage lorsque l'ensemble de ces *n* processus attend chacun une ressource déjà possédée par un autre processus de l'ensemble. Dans une telle situation aucun processus ne peut poursuivre son exécution. L'attente des processus est infinie.

Quatre conditions doivent être vérifiées simultanément dans le système pour qu'un interblocage puisse se produire :

- exclusion mutuelle : une ressource au moins doit se trouver dans un mode non partageable ;
- occupation et attente : un processus au moins occupant une ressource attend d'acquérir des ressources supplémentaires détenues par d'autres processus. Les processus demandent les ressources au fur et à mesure de leur exécution ;
- pas de réquisition : les ressources sont libérées sur seule volonté des processus les détenant ;
- attente circulaire : il existe un cycle d'attente entre au moins deux processus. Les processus impliqués dans ce cycle sont en interblocage.

La figure 8.1 donne un exemple d'attente circulaire entre deux processus P1 et P2. Les deux processus P1 et P2 utilisent les trois mêmes ressources : un lecteur de bandes magnétiques, un disque dur et une imprimante. Le processus P1 demande la bande magnétique puis l'imprimante et enfin le disque. Le processus P2 demande le disque puis l'imprimante et enfin la bande magnétique. Sur le schéma de la figure 8.1, le processus P1 s'est exécuté et a obtenu la bande magnétique ainsi que l'imprimante. Le processus P2 lui a obtenu le disque et il demande maintenant à obtenir l'imprimante. L'imprimante a déjà été allouée au processus P1, donc le processus P2 est bloqué. Le processus P1 ne peut pas poursuivre son exécution car il est en attente du disque qui a déjà été alloué au processus P2. On a donc une attente circulaire entre P1 et P2 : en effet le processus P2 attend l'imprimante détenue par le processus P1 tandis que le processus P1 attend le disque détenu par le processus P2.

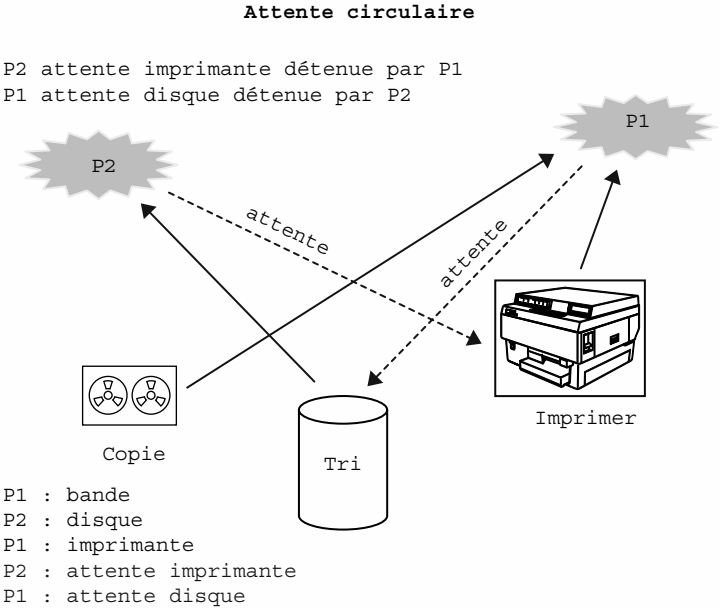

Figure 8.1 – Exemple d'interblocage

8.4.2 Les différentes méthodes de traitement des interblocages

Il existe quatre méthodes de traitement des situations d'interblocage : les politiques de guérison, les politiques de prévention, les politiques d'évitements et la politique de « l'autruche ».

a) Les politiques de guérison

Une première politique est celle de détection/guérison des interblocages. Dans cette politique on autorise les interblocages à se produire, on les détecte puis on les résout.

Détection

Pour cette politique, le système maintient un graphe représentant l'allocation des ressources et les attentes des processus. Dans ce graphe dont un exemple est donné sur la figure 8.2, on distingue deux types de sommets : les processus figurés par un rond et les ressources figurées par un rectangle, chaque point dans le rectangle représentant un exemplaire de la ressource. Une flèche depuis un rectangle vers un rond indique que la ressource a été allouée au processus (arc d'affectation). A contrario une flèche depuis un rond vers un rectangle indique que le processus attend la ressource (arc de requête). Ainsi sur le graphe de la figure 8.2, le processus P1 attend la ressource R1 qui est allouée au processus P2. Le processus P2 attend les ressources R3 et R2. La ressource R3 est allouée au processus P3 qui lui-même attend la ressource R2, allouée au processus P1.

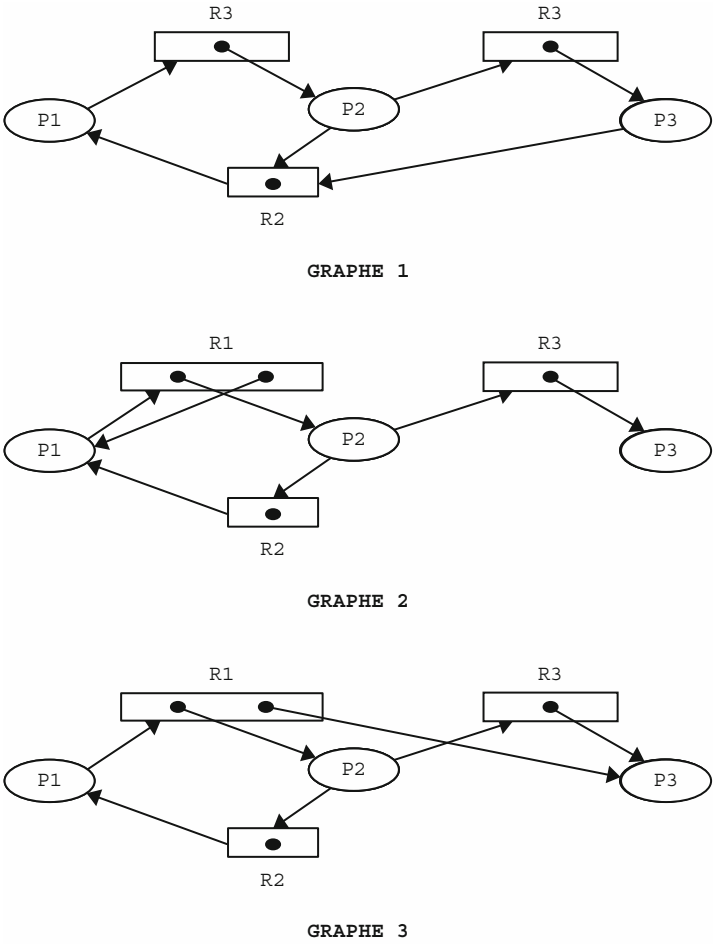

Figure 8.2 – **Graphe d'allocation de ressources**

Dans un tel graphe d'allocation de ressources, l'absence de cycle garantit l'absence d'interblocage.

Au contraire, dans un graphe pour lequel chaque ressource existe en un seul exemplaire, alors la présence d'un cycle implique la présence d'un interblocage et tous les processus participant au cycle sont interbloqués. Dans un graphe pour lequel des ressources disposent de plusieurs instances, la présence d'un cycle peut correspondre à un interblocage, mais cela n'implique pas forcément qu'un interblocage s'est réellement produit.

Ainsi dans la figure 8.2, le graphe 1 présente deux cycles. Un premier cycle existe entre le processus P1, le processus P2, la ressource R1 et la ressource R2. Un second cycle existe qui englobe le processus P1, le processus P2, le processus P3 ainsi que les ressources R1, R3 et R2. Comme chaque ressource n'est présente qu'en un seul exemplaire, ces cycles traduisent que les processus P1, P2 et P3 sont interbloqués.

Le graphe 2 présente une allocation de ressources avec la ressource R1 présente en plusieurs exemplaires. Il existe bel et bien un cycle qui correspond à un interblocage. Par contre, le graphe 3 présente un cycle mais il n'y a pas interblocage.

Le système met à jour le graphe à chaque nouvelle allocation de ressources ou demande d'allocation de ressources.

Guérison

Régulièrement le système parcourt le graphe à la recherche de cycles. Si un cycle est découvert et que celui-ci correspond bien à une situation d'interblocage, alors ce cycle est cassé soit en avortant les processus en interblocage appartenant au cycle, soit en réquisitionnant les ressources allouées à certains processus participant à l'interblocage pour les redistribuer aux autres processus. Ainsi sur l'exemple de la figure 8.2, l'interblocage est cassé en avortant le processus P2 et en réallouant la ressource R1 au processus P1. L'interblocage peut être aussi éliminé en retirant la ressource R1 au processus P2 pour la donner au processus P1 (la différence ici est que le processus P2 n'est pas détruit).

Bien fondé de la politique

Ce type de politique présente plusieurs difficultés. Sa mise en œuvre est coûteuse. Il faut maintenir le graphe d'allocation, régulièrement parcourir le graphe à la recherche de cycles et enfin remédier à l'interblocage par destruction de processus ou réquisition des ressources. Une autre difficulté tient à la période de parcours du graphe : si cette période est petite, le graphe est parcouru souvent et consomme ainsi les ressources du système inutilement car la probabilité d'un interblocage est faible. Si la période de parcours est grande, le graphe sera parcouru moins souvent et la probabilité de trouver un interblocage sera plus forte. Mais, le nombre de processus impliqués dans un l'interblocage risque d'être d'autant plus grand que la période de parcours du graphe est grande. Par ailleurs le choix des processus à avorter pour remédier à un interblocage n'est pas forcément facile. Une solution est de systématiquement détruire tous les processus impliqués dans l'interblocage mais on peut essayer de raffiner cette solution en choisissant les processus à avorter : se pose ici le problème du choix qui va conduire à éliminer l'interblocage en minimisant le nombre de processus avortés ou le coût pour le système de ces avortements. Ainsi dans le cas de la figure 8.1, on pourra choisir d'avorter P2 plutôt que P1 car P1 détient déjà deux ressources sur trois. Le critère de choix peut également se porter sur la priorité du processus, le temps d'exécution déjà effectué ou restant à effectuer. La solution consistant à rompre l'interblocage par réquisition des ressources pose également un problème, celui de replacer la ressource réquisitionnée et le processus privé de sa ressource dans un état cohérent.

b) Les politiques de prévention

Dans les politiques de prévention, des contraintes sont ajoutées sur l'allocation des ressources afin de faire en sorte qu'au moins une des 4 conditions nécessaires à l'interblocage ne soit jamais vérifiée. Les deux seules conditions sur lesquelles il est

possible d'agir sont la condition d'occupation et d'attente ainsi que la condition d'attente circulaire.

Pour la condition d'occupation et d'attente, on interdit à un processus de demander les ressources au fur et à mesure de ses besoins. Ainsi, un processus ne peut démarrer son exécution que lorsque toutes les ressources ont pu lui être allouées. La deuxième condition d'occupation et attente ne peut donc jamais se produire. Cependant l'utilisation résultante des ressources est mauvaise puisqu'un processus dispose des ressources plus longtemps qu'il ne les utilise. Cette méthode correspond à ce qui est appliqué au niveau des sémaphores Linux, où l'ensemble des opérations demandées sur un groupe de sémaphores sont toutes exécutées atomiquement.

Pour la condition d'attente circulaire, une solution est d'imposer un ordre total sur l'ordre d'allocations des différentes ressources du système : ainsi par exemple l'unité de bande doit toujours être demandée avant le disque et le disque doit lui-même être toujours demandé avant l'imprimante. Plus formellement, une fonction $F : R \rightarrow \mathbb{N}$ est définie où $R = \{R_1, R_2, \ldots, R_n\}$ est l'ensemble des classes de ressources du système et $\mathbb{N}$ l'ensemble des nombres naturels. Un processus possédant déjà des ressources de type R_i ne peut accéder à des ressources de type R_j que si $F(R_j) > F(R_i)$. Cette solution est celle que nous avons appliquée au paragraphe 8.1.2, en mettant les opérations `P()` d'allocation de ressources dans le même ordre.

c) Les politiques d'évitement

La troisième catégorie de solutions est celle des politiques d'évitement : ici, à chaque demande d'allocation de ressource faite par un processus, le système déroule un algorithme appelé *algorithme de sûreté* qui regarde si cette allocation *peut* mener le système en interblocage. L'algorithme de sûreté utilise des informations fournies par les processus notamment pour chaque processus, le nombre maximum de ressources requises. Si tel est le cas, l'allocation est retardée.

L'algorithme de sûreté manipule un état d'allocation des ressources à tout instant t, composé de l'ensemble des ressources allouées à cet instant, de l'ensemble des ressources disponibles et pour chaque processus, des besoins maximum en ressources. À partir de ces informations, pour chaque nouvelle allocation demandée, l'algorithme de sûreté examine l'état d'allocation résultant pour déterminer si celui-ci est *sain* ou *malsain* :

- un état est sain si le système peut allouer des ressources à chaque processus (jusqu'à satisfaire sa demande maximum) dans un certain ordre et encore éviter l'interblocage ;
- un état sain n'est pas un état d'interblocage ;
- un état d'interblocage n'est pas un état sain ;
- un état sain ne conduit pas à l'interblocage ;
- un état malsain peut conduire à l'interblocage.

Le système à chaque nouvelle demande d'allocation tente de construire une séquence d'exécution qui inclut l'ensemble des processus. Cette séquence d'exécution simule en quelque sorte l'exécution de chaque processus en lui allouant le

nombre maximum de ressources encore demandées. Si une telle séquence peut être construire alors l'état résultant de la nouvelle allocation est sain et celle-ci est accordée. Sinon, elle est refusée.

Exemple d'état sain et d'état malsain

Considérons trois processus P1, P2 et P3 et un ensemble de ressources composé de 12 imprimantes. Le tableau 8.1 donne l'état d'allocation courant :

Tableau 8.1 – État d'allocation courant

Processus	Besoins maximaux	Ressources allouées (t)	Besoins (besoins maximaux – ressources allouées)
P1	10	5	5
P2	4	2	2
P3	9	2	7

Le nombre de ressources disponibles est égal à 3.

La séquence d'exécution <P2, P1, P3> est saine :

- satisfaction de P2, ressources disponibles = 1 ;
- restitution des ressources par P2, ressources disponibles = 5 ;
- satisfaction de P1, ressources disponibles = 0 ;
- restitution des ressources par P1, ressources disponibles = 10 ;
- satisfaction de P3, ressources disponibles = 3 ;
- restitution des ressources par P3, ressources disponibles = 12.

Considérons à présent l'état d'allocation suivant donné dans le tableau 8.2, résultant de l'allocation au processus P3 d'un exemplaire de ressource imprimante :

Tableau 8.2 – État d'allocation courant

Processus	Besoins maximaux	Ressources allouées (t)	Besoins (besoins maximaux – ressources allouées)
P1	10	5	5
P2	4	2	2
P3	9	3	6

Le nombre de ressources disponibles est égal à 2.

L'état devient malsain et aucune séquence d'exécution incluant les trois processus ne peut être construite. Ici, seul P2 peut être satisfait :

- satisfaction de P2, ressources disponibles = 0 ;
- restitution des ressources par P2, ressources disponibles = 4.

Maintenant, ni P1, ni P2 ne peuvent être satisfaits.

C'est une vision pessimiste qui prédomine car l'allocation est interdite dès que la possibilité de l'interblocage est détectée. Mais cela ne veut pas dire que cet interblocage aura réellement lieu.

Un exemple de la mise en œuvre de cette politique est l'algorithme appelé *algorithme du banquier.*

d) La politique de l'autruche

Une dernière solution, très simple, est de nier l'existence des interblocages et donc de ne rien prévoir pour les traiter. Simplement la machine est redémarrée lorsque trop de processus sont en interblocage.

Les trois premières stratégies évoquées (prévention, évitement détection/guérison) sont des politiques qui coûtent excessivement chères à mettre en œuvre. Aussi, comme la fréquence des interblocages dans un système est relativement faible, la politique de l'autruche se justifie souvent.

8.5 Synchronisation dans le noyau Linux

Les chemins de contrôle du noyau manipulent les structures de données de celui-ci. Aussi, lorsqu'un chemin est suspendu au profit d'un autre, ce dernier ne doit pas accéder aux structures manipulées par le chemin suspendu à moins que celui-ci les ait laissées dans un état cohérent.

Le noyau dispose de trois méthodes pour assurer la cohérence de ses structures :

- le noyau est non préemptible ;
- le noyau peut masquer les interruptions ;
- le noyau dispose de sémaphores.

8.5.1 Le noyau est non préemptible

Un processus s'exécutant en mode noyau ne peut pas être arbitrairement suspendu au profit d'un autre processus et il ne quitte donc le processeur que si il se bloque lui-même. Lorsqu'un processus se bloque en mode noyau, il s'assure qu'il laisse les structures de données qu'il a manipulées dans un état cohérent. Par ailleurs, lorsqu'il reprend son exécution, il vérifie les valeurs des données qu'il a modifiées avant sa suspension afin de connaître si elles ont été altérées.

Comme le noyau est non préemptible, les appels système non bloquants sont réalisés de manière atomique par rapport aux autres chemins de contrôle démarrés dans le noyau. Cette exécution atomique permet aux chemins de contrôle d'accéder sans vérification aux structures de données qui ne sont pas, par ailleurs, manipulées par les gestionnaires d'interruption et d'exception.

8.5.2 Masquage des interruptions

Le noyau peut se placer en section critique, en masquant et démasquant les interruptions, comme nous l'avons évoqué au paragraphe 8.1.1. Ce masquage ne doit être

fait que sur des portions de code non bloquantes, sinon la machine pourrait être gelée.

Le masquage des interruptions s'effectue en sauvegardant tout d'abord le contenu du registre d'état du processeur (`macro save_flags(old)`) qui copie le registre d'état dans une variable locale `old`), puis en désactivant les interruptions à l'aide de l'instruction machine `cli()`. Le démasquage des interruptions opère de façon inverse, en restaurant tout d'abord le registre d'état (macro `restore_flags(old)`) et en réactivant les interruptions par un appel à l'instruction machine `sti`.

Ce masquage et démasquage des interruptions est par exemple utilisé lors de la manipulation des files d'attente de l'ordonnanceur.

8.5.3 Sémaphores

Le noyau Linux peut également utiliser des outils sémaphores pour se placer en section critique. Un sémaphore du noyau est une structure composée d'une file d'attente de processus et d'un compteur. Deux opérations permettent de le manipuler :

- l'opération `down()` décrémente le compteur du sémaphore et si celui-ci devient négatif, bloque le processus appelant ;
- l'opération `up()` incrémente le compteur du sémaphore et si le compteur devient supérieur ou égal à 0, réveille les processus en attente sur le sémaphore.

L'accès aux descripteurs de régions ou encore l'accès aux inodes s'effectue en acquérant le sémaphore gardant l'accès à chacune de ces structures.

8.5.4 Interblocage

Afin d'éviter les opérations d'interblocage lors des accès aux différentes structures du noyau, le noyau Linux impose un ordre total sur les demandes d'allocations de ressources. Cet ordre est fonction des adresses en mémoire centrale des sémaphores gardant l'accès aux structures ; plus précisément les requêtes sur les sémaphores sont émises selon l'ordre croissant des adresses.

Exercices

8.1 Qu'avez-vous retenu ?

Pour chacune des questions suivantes, choisissez la ou les réponses qui vous semblent exactes.

Question 1 – Une opération `P(sem)` est :

- ☐ a. une opération qui initialise le sémaphore `sem`.
- ☐ b. une opération éventuellement bloquante pour le processus appelant.
- ☐ c. une opération qui réveille un processus bloqué sur le sémaphore `sem`.

Question 2 – Une opération `V(sem)` est :

- ☐ a. une opération qui initialise le sémaphore `sem`.
- ☐ b. une opération éventuellement bloquante pour le processus appelant.
- ☐ c. une opération qui réveille un éventuel processus bloqué sur le sémaphore `sem`.

Question 3 – Une ressource critique est une ressource :

- ☐ a. qui n'existe qu'en un seul exemplaire dans le système.
- ☐ b. qui doit être accédée en exclusion mutuelle, c'est-à-dire seulement par des processus lecteurs.
- ☐ c. qui doit être accédée en exclusion mutuelle, c'est-à-dire par un seul processus à la fois.

Question 4 – Deux processus sont en interblocage :

- ☐ a. s'ils s'attendent l'un l'autre pour acquérir une ressource prise par l'autre processus.
- ☐ b. si l'un des deux processus ne peut jamais disposer de l'ensemble des ressources qui lui sont nécessaires pour s'exécuter.

Question 5 – Un état malsain est :

- ☐ a. un état qui conduit à l'interblocage.
- ☐ b. un état d'interblocage.
- ☐ c. un état qui peut conduire à l'interblocage.

Question 6 – Un code atomique ou indivisible est :

- ☐ a. un code dont l'exécution ne peut pas être interrompue par un signal.
- ☐ b. un code qui s'exécute interruptions masquées.
- ☐ c. un code dont l'exécution peut être préemptée.
- ☐ d. un code dont l'exécution ne peut pas être préemptée.

Question 7 – L'exclusion mutuelle :

- ☐ a. garantit que les processus ne peuvent pas être en interblocage.
- ☐ b. régit l'accès à une ressource critique.

8.2 Ressources critiques, interblocage

On considère trois processus P1, P2, P3 dont les caractéristiques sont les suivantes :

	Date d'arrivée	Temps d'exécution	Priorité (plus petite valeur = plus grande priorité)
P1	0	5 unités	4
P2	2	4 unités	1
P3	3	4 unités	2

Les trois processus sont ordonnancés selon une politique de priorités préemptives.

On suppose à présent que les processus P1 et P2 utilisent une même ressource critique R1. P3 ne fait que du calcul.

Les étapes des deux processus P1 et P2 sont les suivantes :

P1	P2
Calcul durant 1 unité Prendre (R1) Faire calcul en utilisant R1 durant 2 unités Rendre (R1) Calcul durant 2 unités	Calcul durant 1 unité Prendre (R1) Faire calcul en utilisant R1 durant 2 unités Rendre (R1) Calcul durant 1 unité

1) R1 est une ressource critique. Traduisez les opérations Prendre(R1) et Rendre(R1) à l'aide d'un sémaphore.
2) Construisez le chronogramme d'exécution des trois processus en tenant compte du partage de la ressource R1 entre P1 et P2. Les opérations Prendre(R1) et Rendre(R1) ne comptent pas de d'unités de temps si elles sont passantes (on considère qu'elles sont instantanées). Que se passe-t-il ? Est-ce que P2 s'exécute effectivement comme étant le processus le plus prioritaire ? Pourquoi ?
3) On suppose à présent que les processus P1 et P2 utilisent deux ressources critiques R1 et R2.

   Les étapes des deux processus sont les suivantes :

P1	P2
Calcul durant 1 unité Prendre (R2) Utiliser R2 durant 1 unité Prendre (R1) Utiliser R2 et R1 durant 1 unité Rendre (R1) Rendre (R2) Calcul durant 2 unités	Calcul durant 1 unité Prendre (R1) Prendre (R2) Utiliser R1 et R2 durant 2 unités Rendre (R1) Rendre (R2) Calcul durant 1 unité

   Que peut-il se passer ? Quelle solution simple peut être mise en œuvre pour éviter la situation constatée.

8.3 Producteurs-Consommateurs

Soit un système composé de trois processus cycliques Acquisition, Exécution et Impression, et de deux tampons Requête et Avis, respectivement composés de *M* et *N* cases. Les deux tampons sont gérés de manière circulaire.

Le processus Acquisition enregistre chacune des requêtes de travail qui lui sont soumises par des clients et les place dans le tampon Requête à destination du processus Exécution.

Le processus Exécution exécute chaque requête de travail prélevée depuis le tampon Requête et transmet ensuite au processus Impression un ordre d'impression de résultats déposé dans le tampon Avis.

Le processus Impression prélève les ordres d'impression déposés dans le tampon Avis et exécute ceux-ci.

1) Programmez la synchronisation des trois processus à l'aide des sémaphores et des variables nécessaires à la gestion des tampons.
2) On étend le système à trois processus Acquisition, trois processus Exécution et trois processus Impression. Complétez la synchronisation précédente pour que celle-ci demeure correcte.

8.4 Sémaphores et mémoire partagée

Reprenez l'exemple du paragraphe 8.2.3, en remplaçant les *threads* par un processus père et son fils. Le comportement doit être le même quant à l'accès à la variable *i*.

8.5 Producteur-Consommateur

Programmez un schéma producteur-consommateur entre un processus père et son fils. Le processus père est le consommateur tandis que le processus fils est le producteur. Le fils écrit dans chacune des cases d'un tampon de 10 cases une valeur *i*, incrémentée de une unité à chaque fois. Le processus consommateur lit chacune des cases du tampon et affiche la valeur lue.

8.6 Lecteurs-Rédacteur

Programmez un schéma lecteurs-rédacteur entre un processus père et ses deux fils pour l'accès à une variable commune. Le processus père est le rédacteur tandis que les processus fils sont des lecteurs. Le père modifie la variable commune en la multipliant par 10. Les processus lecteur lisent la valeur de la variable commune puis l'affiche.

8.7 États sains et états malsains

1) On considère quatre processus P1, P2, P3 et P4 et un ensemble de 8 ressources identiques. Le tableau 8.3 donne l'état d'allocation courant.

Tableau 8.3 – État d'allocation courant

Processus	Besoins maximaux	Ressources allouées (t)	Besoins (besoins maximaux – ressources allouées)
P1	5	2	3
P2	3	1	2
P3	4	2	2
P4	2	1	1

L'état courant est-il un état sain ?

2) On considère trois processus P1, P2, P3 et trois ensembles de ressources distinctes R1, R2 et R3. Les ressources R1 sont présentes en 50 exemples, les ressources R2 sont présentes en 200 exemplaires et les ressources R3 sont présentes en 50 exemplaires. Le tableau 8.4 donne l'état d'allocation courant.

Tableau 8.4 – État d'allocation courant

Processus	Besoins maximaux (R1, R2, R3)	Ressources allouées (*t*) (R1, R2, R3)	Besoins (besoins maximaux – ressources allouées) (R1, R2, R3)
P1	10,100,30	5,30,10	5,70,20
P2	30,150,40	5,10,10	25,140,30
P3	20,60,10	10,50,5	10,10,5

L'état courant est-il un état sain ?

8.8 Exclusion mutuelle

On considère que lorsqu'un processus réalise une opération d'écriture avec le disque, il l'effectue en appelant une fonction du système `ECRIRE_DISQUE (tampon, nb_octets)`, qui écrit les `nb_octets` de données placées dans le tableau `tampon`. Cette fonction système fait appel à une fonction bas niveau du pilote disque `Ecrire_blocdisque` qui effectue l'écriture physique par bloc de 512 octets.

Le code simplifié de la fonction du système `ECRIRE_DISQUE` est le suivant :

```
Fonction ECRIRE_DISQUE (tampon, nb_octets)
char tampon[] ;
int nb_octets
{
i = 0 ;
Tant que (nb_octets > 0)
Faire
  -- l'écriture physique s'effectue par blocs de 512 octets
  Ecrire_blocdisque (tampon[i, i + 512], 512) ;  -- on écrit physiquement
512 octets
  nb_octets = nb_octets - 512 ;  -- on décrémente le nombre total d'octets
à écrire
  i = i + 512 ; -- on avance l'index dans le tableau des octets à écrire
Fait
Return ;
}
```

L'écriture complète des `nb_octets` de données placées dans le tableau `tampon`, réalisée par la fonction système `ECRIRE_DISQUE` constitue une section critique.

1) Qu'est-ce que cela veut dire ?

2) On dispose de deux primitives de synchronisation suivantes agissant sur un objet `Verrou` :
   - `Objet Verrou (état libre, état occupé)` ; `Verrou` est initialement dans l'état libre.
   - `Verrouiller (Verrou)` : si `Verrou` est dans l'état libre alors le mettre à l'état occupé. Si `Verrou` est dans l'état occupé, alors bloquer le processus effectuant l'opération `Verrouiller`.
   - `Deverrouiller (Verrou)` : si un processus est bloqué sur `Verrou`, le réveiller. Sinon, mettre `Verrou` à l'état libre.

   Utilisez ces deux primitives et l'objet `Verrou` pour apporter la propriété d'exclusion mutuelle au code de la fonction système `ECRIRE_DISQUE`.
3) Étant donné que la fonction `ECRIRE_DISQUE` est une fonction du système, celle-ci s'exécute donc en mode superviseur. Quelles opérations aurait-on pu utiliser dans ce cadre pour apporter la propriété d'exclusion mutuelle à la fonction `ECRIRE_DISQUE` ?

Solutions

8.1 Qu'avez-vous retenu ?

Question 1 – Une opération `P(sem)` est :

- ☐ b. une opération éventuellement bloquante pour le processus appelant.

Question 2 – Une opération `V(sem)` est :

- ☐ c. une opération qui réveille un éventuel processus bloqué sur le sémaphore `sem`.

Question 3 – Une ressource critique est une ressource :

- ☐ c. qui doit être accédée en exclusion mutuelle, c'est-à-dire par un seul processus à la fois.

Question 4 – Deux processus sont en interblocage :

- ☐ a. s'ils s'attendent l'un l'autre pour acquérir une ressource prise par l'autre processus.

Question 5 – Un état malsain est :

- ☐ c. un état qui peut conduire à l'interblocage.

Question 6 – Un code atomique ou indivisible est :

- ☐ b. un code qui s'exécute interruptions masquées.
- ☐ d. un code dont l'exécution ne peut pas être préemptée.

Question 7 – L'exclusion mutuelle :

- ☐ b. régit l'accès à une ressource critique.

8.2 Ressources critiques, Interblocage

1) Soit le sémaphore Mutex, initialisé à 1. Alors P (Mutex) équivaut à Prendre (R1) et V (Mutex) équivaut à Rendre (R1).

2) Le chronogramme est donné par la figure 8.3.

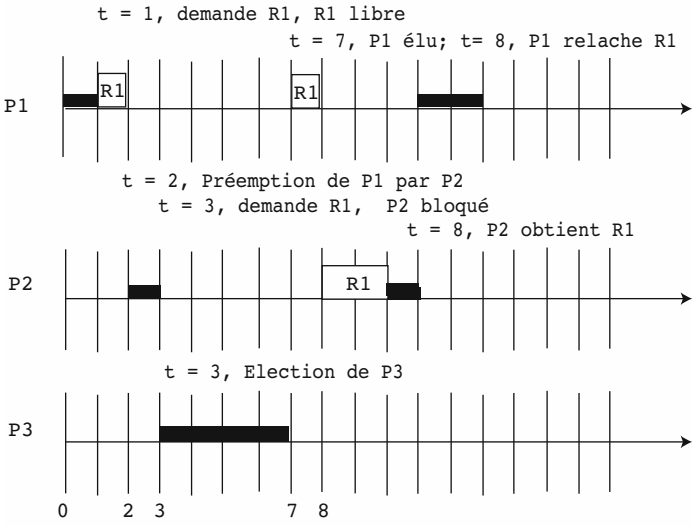

Figure 8.3

Non, P2 ne s'exécute pas comme étant le processus le plus prioritaire, car il est bloqué en attente de la ressource R1 possédée par le processus P1.

3) Il peut se produire un interblocage. Par exemple P1 obtient R2 puis P1 est préempté. P2 s'exécute et obtient R1. Les deux processus sont maintenant en attente l'un de l'autre pour obtenir les ressources qui leur manquent. Une solution simple est d'ordonner complètement l'accès aux ressources R1 et R2, par exemple d'abord R1 puis R2 et d'obliger les deux processus P1 et P2 à demander ces ressources dans cet ordre. Ainsi P1 devient :

P1
Calcul durant 1 unité Prendre (R2) Utiliser R2 durant 1 unité Prendre (R1) Utiliser R2 et R1 durant 1 unité Rendre (R1) Rendre (R2) Calcul durant 2 unités

8.3 Producteurs-Consommateurs

1) On identifie un schéma producteur-consommateur sur chacun des tampons. On utilise pour chacun de ces schémas un couple de sémaphores :
   - `mvide` initialisé à m et `mplein` initialisé à 0 (tampon requête) ;
   - `nvide` initialisé à n et `nplein` initialisé à 0 (tampon avis).

```
déclarations globales:
requête: tampon (0..m-1) de messages;
avis: tampon (0..n-1) de messages;
int m, n;
mvide, nvide, mplein, nplein: sémaphores;
Init(mvide, m); Init (nvide, n); Init(mplein, 0); Init(nplein, 0);
```

Acquisition	Exécution	Impression

mess: message;
int i = 0;
{
for(;;) {
enregistrer_travail
(mess);
P(mvide);
requete(i) = mess;
i = i + 1 mod m;
V(mplein);
 	
}
}
	mess, res: message;
	int j,k = 0;
	{
	for(;;)
	P(mplein);
	
	mess = requete(j);
	j = j + 1 mod m;
	V(mvide);
	exécuter_travail(mess, res);
	P(nvide);
	avis(k) = res;
	k = k + 1 mod n;
	V(nplein);
	}
	}
		mess: message;
		int l = 0;
		{
		for(;;)
		P(nplein);
		
		mess = avis(l);
		l = l + 1 mod n;
		V(nvide);
		imprimer_resultat(mess);
		
		}
		}

2) Il faut maintenant gérer les accès concurrents aux tampons avis et requête. En effet :
   - ◇ les différents processus Acquisition se partagent l'index *i* ;
   - ◇ les différents processus Exécution se partagent l'index *j* et *k* ;
   - ◇ les différents processus Impression se partagent l'index *k*.

   Les variables *i*, *j*, *k*, *l* sont maintenant globales et les accès à ces variables doivent se faire en exclusion mutuelle. On ajoute donc quatre sémaphores d'exclusion mutuelle initialisés à 1 (un sémaphore par index).

```
déclarations globales:
requête: tampon (0..m - 1) de messages;
avis: tampon (0..n - 1) de messages;
int m, n;
int i, j, k, l;
mvide, nvide, mplein, nplein, muti, mutj, mutk, mutl: sémaphores;

{
Init(mvide, m); Init (nvide, n); Init(mplein, 0); Init(nplein, 0);
Init(muti, 1); Init(mutj, 1); Init(mutk, 1); Init(mutl, 1);
i = j= k = l = 0; }
```

Acquisition	Exécution	Impression

mess: message;
{
for(;;)
enregistrer_travail
(mess);
P(mvide);
P(muti);
requete(i) = mess;
i = i + 1 mod m;
V(muti);
V(mplein);
 	
}
}
	mess, res: message;
	{
	for(;;)
	P(mplein);
	
	P(mutj)
	mess = requete(j);
	j = j + 1 mod m;
	V(mutj);
	V(mvide);
	exécuter_travail(mess, res);
	P(nvide);
	P(mutk);
	avis(k) = res;
	k = k + 1 mod n;
	V(mutk);
	V(nplein);
	}
	}
		mess: message;
		{
		for(;;)
		P(mutl);
		
		P(nplein);
		mess = avis(l);
		l = l + 1 mod n;
		V(nvide);
		V(mutl);
		imprimer_resultat
		(mess);
		}
		}

8.4 Sémaphores et mémoire partagée

La variable *i* est placée dans une région de mémoire partagée entre le père et le fils.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

main() {

int i, pid;
int semid;
struct sembuf operation;
int shmid;
int *mem;

if ((shmid = shmget(45, 1000, 0750|IPC_CREAT|IPC_EXCL)) ==  -1)
    perror ("pb création mémoire partagée");

if ((mem = shmat(shmid,NULL, 0))== NULL)
    perror ("shmat");

if ((semid = semget (12, 1, IPC_CREAT|IPC_EXCL|0600)) == -1)
     perror("pb semget");
```

```
if ((semctl(semid, 0, SETVAL, 1)) == -1)
     perror("pb semctl");

pid = fork();
if (pid == 0)
{
  operation.sem_num = 0;
  operation.sem_op = -1;
  operation.sem_flg = 0;
  semop(semid,&operation, 1);
  *mem = *mem + 10;
  printf("la valeur de i est %d\n", *mem);
   *mem = *mem + 20;
  printf("la valeur de i est %d\n", *mem);
  operation.sem_num = 0;
  operation.sem_op =  1;
  operation.sem_flg = 0;
  semop(semid,&operation, 1);
  exit();
  }
else
  {
  operation.sem_num = 0;
  operation.sem_op = -1;
  operation.sem_flg = 0;
  semop(semid,&operation, 1);
  *mem = *mem + 1000;
  printf("la valeur de i est %d\n", *mem);
   *mem = *mem + 2000;
  printf("la valeur de i est %d\n", *mem);
  operation.sem_num = 0;
  operation.sem_op =  1;
  operation.sem_flg = 0;
  semop(semid,&operation, 1);
  wait();
  shmdt(mem);
  shmctl(shmid,IPC_RMID,NULL);
  semctl(semid, 0, IPC_RMID, 0);
  }
}
```

8.5 Producteur-Consommateur

Le tampon de 10 cases est matérialisé par une région de mémoire partagée.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
```

```
main() {

int i, pid;
int semid;
struct sembuf operation;
int shmid;
int *mem, *j, *k;

/* allocation région de mémoire partagée */
if ((shmid = shmget(45, 1000, 0750|IPC_CREAT|IPC_EXCL)) ==  -1)
    perror ("pb création mémoire partagée");

if ((mem = shmat(shmid,NULL, 0))== NULL)
    perror ("shmat");

/* création de deux sémaphores */
if ((semid = semget (12, 2, IPC_CREAT|IPC_EXCL|0600)) == -1)
     perror("pb semget");

/* sémaphore PLEIN initialisé à 10 */
if ((semctl(semid, 0, SETVAL, 10)) == -1)
     perror("pb semctl");

/* sémaphore VIDE initialisé à 0 */
if ((semctl(semid, 1, SETVAL, 0)) == -1)
     perror("pb semctl");

pid = fork();
if (pid == 0)
{
  /* le fils est producteur : */
  /*produit 10 messages contenant la valeur i */
  i = 0; j = mem;
  while (i < 10) {
  /* P(PLEIN) */
  operation.sem_num = 0;
  operation.sem_op = -1;
  operation.sem_flg = 0;
  semop(semid,&operation, 1);
  *j = i;
   printf("producteur:la valeur de i est %d\n", *j);
  j++;
  i++;
  /* V(VIDE) */
  operation.sem_num = 1;
  operation.sem_op =  1;
  operation.sem_flg = 0;
  semop(semid,&operation, 1);
  }
```

```
  exit();
  }
else
  {
  /* le père est consommateur : */
  /* lit 10 messages contenant la valeur i */
  i = 0; k = mem;
  while (i < 10) {
  /* P(VIDE) */
  operation.sem_num = 1;
  operation.sem_op = -1;
  operation.sem_flg = 0;
  semop(semid,&operation, 1);
   printf("consommateur:la valeur de i est %d\n", *k);
   k++;
   i++;
  /* V(PLEIN) */
  operation.sem_num = 0;
  operation.sem_op =  1;
  operation.sem_flg = 0;
  semop(semid,&operation, 1);
  }
  wait();
  shmdt(mem);
  shmctl(shmid,IPC_RMID,NULL);
  semctl(semid, 0, IPC_RMID, 0);
  }
}
```

8.6 Lecteurs-Rédacteur

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

main() {

int i, pid;
int semid;
struct sembuf operation;
int shmid;
int *mem, *NL;


/* allocation région de mémoire partagée */
if ((shmid = shmget(45, 1000, 0750|IPC_CREAT|IPC_EXCL)) ==  -1)
    perror ("pb création mémoire partagée");
if ((mem = shmat(shmid,NULL, 0))== NULL)
    perror ("shmat");
```

```
/* création de deux sémaphores */
if ((semid = semget (12, 2, IPC_CREAT|IPC_EXCL|0600)) == -1)
     perror("pb semget");

/* sémaphore MUTEX initialisé à 1 */
if ((semctl(semid, 0, SETVAL, 1)) == -1)
     perror("pb semctl");

/* sémaphore ACCES initialisé à 1 */
if ((semctl(semid, 1, SETVAL, 1)) == -1)
     perror("pb semctl");

NL = mem+1;
pid = fork();
if (pid == 0)
{
  /* le fils 1 est lecteur: lit la variable i */
  {
  i = 1;
  while (i < 10)
  {
  /* P(MUTEX) */
  operation.sem_num = 0;
  operation.sem_op = -1;
  operation.sem_flg = 0;
  semop(semid,&operation, 1);
  *NL = *NL + 1;
  if (*NL == 1) {
  /* P(ACCES) */
     operation.sem_num = 1;
     operation.sem_op = -1;
     operation.sem_flg = 0;
     semop(semid,&operation, 1); }
  /* V(MUTEX) */
  operation.sem_num = 0;
  operation.sem_op = 1;
  operation.sem_flg = 0;
  semop(semid,&operation, 1);
  printf("lecteur1:la valeur de la variable est %d\n", *mem);
  /* P(MUTEX) */
  operation.sem_num = 0;
  operation.sem_op = -1;
  operation.sem_flg = 0;
  semop(semid,&operation, 1);
  *NL = *NL - 1;
  if (*NL == 0) {
  /* V(ACCES) */
     operation.sem_num = 1;
     operation.sem_op =  1;
```

```
      operation.sem_flg = 0;
      semop(semid,&operation, 1); }
  /* V(MUTEX) */
  operation.sem_num = 0;
  operation.sem_op = 1;
  operation.sem_flg = 0;
  semop(semid,&operation, 1);
  i = i + 1;
}
  }
  exit();
  }
else
   {
   pid = fork();
   if (pid == 0)
  /* le fils 2 est lecteur: lit la variable i */
  {
  i = 1;
  while (i < 10)
  {
  /* P(MUTEX) */
  operation.sem_num = 0;
  operation.sem_op = -1;
  operation.sem_flg = 0;
  semop(semid,&operation, 1);
  *NL = *NL + 1;
  if (*NL == 1) {
  /* P(ACCES) */
      operation.sem_num = 1;
      operation.sem_op = -1;
      operation.sem_flg = 0;
      semop(semid,&operation, 1); }
  /* V(MUTEX) */
  operation.sem_num = 0;
  operation.sem_op = 1;
  operation.sem_flg = 0;
  semop(semid,&operation, 1);
  printf("lecteur2:la valeur de la variable est %d\n", *mem);
  /* P(MUTEX) */
  operation.sem_num = 0;
  operation.sem_op = -1;
  operation.sem_flg = 0;
  semop(semid,&operation, 1);
  *NL = *NL - 1;
  if (*NL == 0) {
  /* V(ACCES) */
      operation.sem_num = 1;
      operation.sem_op =  1;
```

```
   operation.sem_flg = 0;
   semop(semid,&operation, 1); }
/* V(MUTEX) */
operation.sem_num = 0;
operation.sem_op = 1;
operation.sem_flg = 0;
semop(semid,&operation, 1);
i = i + 1;
}
exit();
}

else
{
/* le père est rédacteur: modifie la valeur i */
i = 1;
while (i  < 10){
 /* P(ACCES) */
operation.sem_num = 0;
operation.sem_op = -1;
operation.sem_flg = 0;
semop(semid,&operation, 1);
*mem = i * 10;
 printf("rédacteur:la valeur de i est %d\n", *mem);
 /* V(ACCES) */
operation.sem_num = 0;
operation.sem_op = 1;
operation.sem_flg = 0;
semop(semid,&operation, 1);
i = i + 1;
}
}
}
wait(); wait();
shmdt(mem);
shmctl(shmid,IPC_RMID,NULL);
semctl(semid, 0, IPC_RMID, 0);
}
```

8.7 États sains et états malsains

1) L'état courant est un état sain; en effet il existe plusieurs suites saines de processus comme par exemple <P3, P4, P1, P2> ou encore <P4, P1, P3, P2> ou encore <P2, P3, P4, P1>.
2) L'état courant est un état sain; en effet il existe au moins une suite saine qui est <P1, P2, P3>.

8.8 Exclusion mutuelle

1) Une seule écriture de `nb_octets` de données placées dans le tableau `tampon` est réalisée à la fois.

2)

```
Fonction ECRIRE_DISQUE (tampon, nb_octets)
char tampon[] ;
int nb_octets;
objet verrou := libre;
{
Verrouiller (verrou)
i = 0 ;
Tant que (nb_octets > 0)
Faire
-- l'écriture physique s'effectue par blocs de 512 octets
   Ecrire_blocdisque (tampon[i, i + 512], 512) ;  -- on écrit
physiquement 512 octets
   nb_octets = nb_octets - 512 ;  -- on décrémente le nombre total
d'octets à écrire
   i = i + 512 ;  -- on avance l'index dans le tableau des octets à écrire
Fait
Deverrouiller (verrou)
Return ;
}
```

3) La section critique peut être réalisée en masquant et démasquant les interruptions.

PROGRAMMATION RÉSEAU

9

PLAN

OBJECTIFS

- ➤ Ce chapitre est consacré à l'étude de la programmation socket qui permet de faire communiquer des processus placés sur des machines distantes, en utilisant les protocoles réseau UDP/IP et TCP/IP.
- ➤ Nous commençons par décrire les grandes notions liées au modèle client-serveur dans le cadre du réseau, puis nous effectuons des rappels sur les propriétés du protocole réseau TCP/IP dans le cadre de l'interconnexion de réseaux internet. Nous décrivons ensuite l'interface de programmation socket.
- ➤ Nous terminons ce chapitre en présentant les notions relatives à l'appel de procédure à distance (RPC, *Remote Procedure Call*).

9.1 L'INTERCONNEXION DE RÉSEAUX

9.1.1 Le modèle client-serveur

Le modèle de communication client-serveur est un modèle de communication très largement répandu. D'un côté, un programme serveur ouvre un service et attend des requêtes provenant de clients. Pour chacune des requêtes reçues, le serveur effectue un traitement puis renvoie une réponse. De l'autre côté, un programme client émet des requêtes, puis attend la réponse du serveur.

Le client et le serveur étant placés sur des machines différentes, la communication entre ces deux entités s'effectue par messages, incluant notamment les adresses sur le réseau des deux parties.

Il est possible de caractériser un serveur selon deux critères :

- la manière dont il traite les requêtes provenant des clients. On distingue à ce niveau le serveur itératif du serveur parallèle ;
- la manière dont il réagit vis-à-vis des pannes. On distingue à ce niveau le serveur sans état et le serveur à états.

Un *serveur itératif* (figure 9.1) est un serveur qui ne peut traiter qu'une seule requête client à la fois. Il est composé d'un processus unique qui effectue les opérations de réception de requêtes, de traitement de requêtes et d'émission des réponses. Une file d'attente permet de conserver les requêtes en attente de service.

Un *serveur parallèle* (figure 9.1) est au contraire un serveur qui peut traiter plusieurs requêtes client à la fois. Il est composé d'un processus père qui effectue les opérations de réception de requêtes et pour chaque requête reçue, d'un processus fils qui effectue le traitement de la requête et l'émission de la réponse.

Le serveur parallèle permet évidemment d'offrir des temps de réponse meilleurs pour le traitement des requêtes clients. Il est cependant plus coûteux en ressources système notamment parce qu'il demande la création d'un nouveau processus pour chaque nouvelle requête. Cette difficulté peut être levée en remplaçant les processus lourds par des threads.

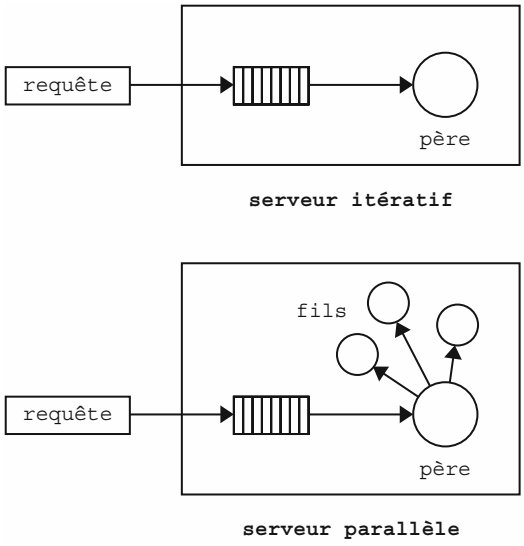

Figure 9.1 – **Serveur itératif et serveur parallèle**

Un *serveur sans état* est un serveur qui ne conserve aucune information sur les requêtes en cours de traitements. C'est donc un serveur sans mémoire qui en cas de panne, ne sait pas reprendre le traitement des requêtes là où il a été arrêté.

Un *serveur à état* est un serveur qui conserve dans un journal placé sur un support non volatil tel qu'un disque dur, des informations sur les requêtes en cours de traitements. C'est donc un serveur à mémoire qui en cas de panne, peut restaurer une partie de son état au moment de la panne et reprendre des requêtes en cours.

9.1.2 Les architectures clients-serveurs

Le modèle clients-serveurs peut être décliné selon différentes architectures dont les plus courantes sont l'architecture à deux niveaux (*2-tier*) et l'architecture à trois niveaux (*3-tier*) (figure 9.2).

Dans l'architecture à deux niveaux, le serveur fournit lui-même et complètement le service aux clients sans faire appel à une autre application.

Dans l'architecture à trois niveaux, le serveur fait appel à un second serveur pour remplir le service demandé par un client. Le premier serveur est appelé *serveur d'application* ou encore *middleware*. Le second serveur qui est très souvent un serveur de bases de données est appelé *serveur secondaire*.

Ce schéma se généralise très aisément dans une architecture à *n* niveaux.

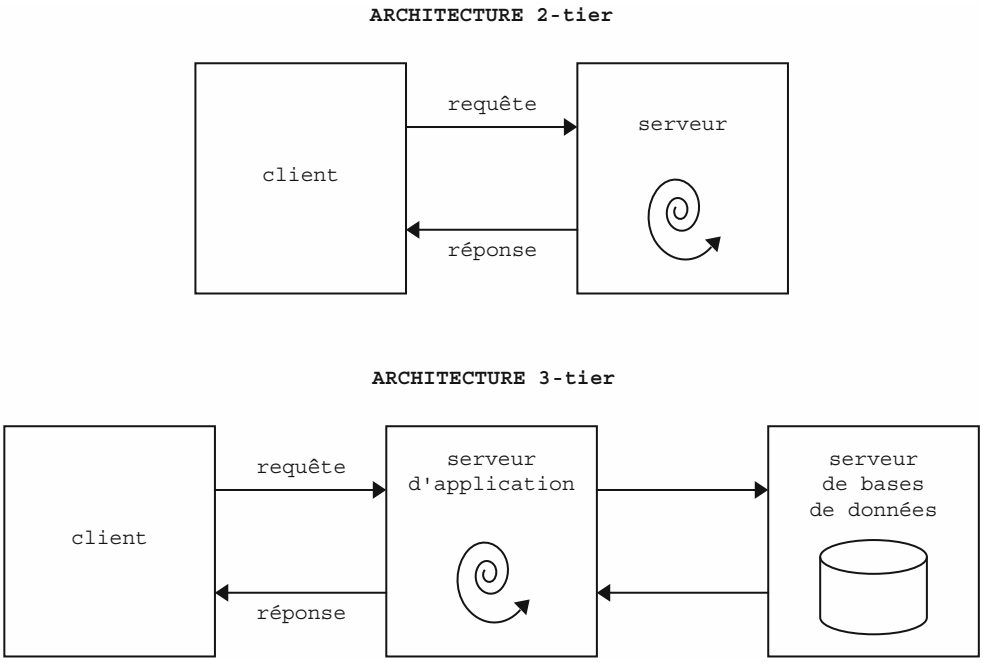

Figure 9.2 – **Architectures clients-serveurs**

9.1.3 L'interconnexion de réseaux

Le but de l'interconnexion de réseaux est d'interconnecter entre eux des réseaux hétérogènes, en créant un réseau virtuel qui masque complètement les caractéristiques des réseaux physiques empruntés. Une machine M1 dialoguant avec une machine M2 passe ainsi par différents réseaux physiques sans en avoir conscience. L'interconnexion des différents réseaux s'effectue au travers de machines charnières appelées *passerelles* (figure 9.3).

Afin de bâtir ce réseau virtuel, il est nécessaire d'une part de définir un plan d'adressage des machines au sein de cette interconnexion, d'autre part de définir un protocole de communication permettant un dialogue de bout en bout entre deux machines physiquement connectées à des réseaux locaux différents.

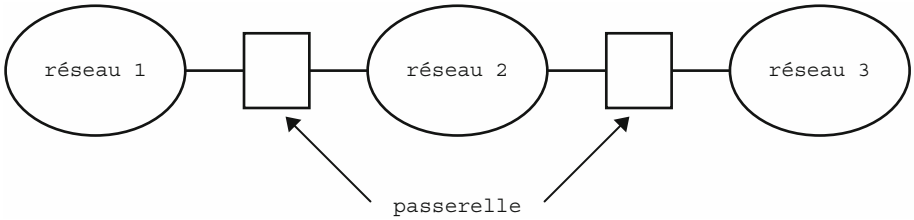

Figure 9.3 – Interconnexion de réseaux

a) L'adressage IP

Adresse logique

Chaque machine dispose au sein de l'interconnexion de réseaux d'une adresse logique encore appelée *adresse IP*. Cette adresse est composée de deux parties. La première partie de l'adresse désigne un réseau au sein de l'interconnexion tandis que la seconde partie de l'adresse désigne une machine au sein de ce réseau. La partie réseau de l'adresse est obtenue auprès d'un organisme particulier, le NIC (*Network Information Control Center*), auquel toute demande de raccordement à l'interconnexion de réseau doit être adressée. La seconde partie de l'adresse est elle gérée de manière autonome.

L'adresse IP est un entier de 32 bits, au sein duquel l'adresse du réseau occupe soit 8 bits, soit 16 bits, soit 24 bits, définissant ainsi trois classes d'adresses appelées classe A, classe B et classe C. Les réseaux de classe A correspondent à des réseaux de grande taille pouvant comporter au plus 2^{24} machines. Au contraire, les réseaux de classe C correspondant à de petits réseaux pouvant comporter au plus 2^8 machines. Enfin, les réseaux de classe B peuvent comporter au plus 2^{16} machines (figure 9.4).

L'adresse IP d'une machine est souvent donnée sous une forme dite pointée constituée par la valeur en base 10 de chacun des octets constituant l'entier séparés par un « . ». Ainsi l'adresse IP 10000000 10100100 00010100 00000110 est donnée sous la forme 128.256.20.6.

L'adresse IP étant essentiellement l'adresse d'une machine au sein d'un réseau donné, il s'ensuit qu'une machine telle qu'une passerelle connectée à plusieurs réseaux différents aura plusieurs adresses IP, une pour chaque réseau auquel elle est attachée. La machine est alors dite *multidomiciliée* (figure 9.5).

Par ailleurs, l'adresse IP étant une adresse purement logique, celle-ci doit être convertie vers l'adresse physique correspondante au sein du réseau auquel la machine appartient, par exemple l'adresse éthernet de la machine pour un réseau de type Ethernet. Cette conversion de l'adresse IP vers l'adresse physique s'appelle *la résolution d'adresses* (figure 9.7).

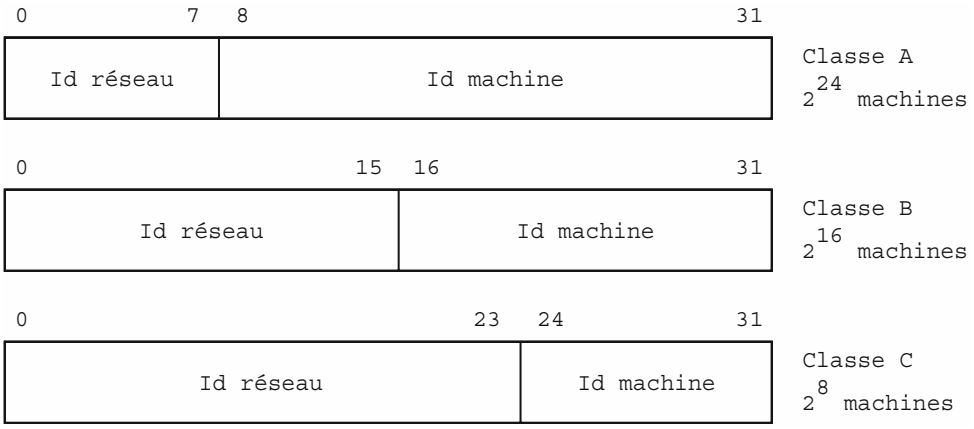

Figure 9.4 – Classes d'adresses IP

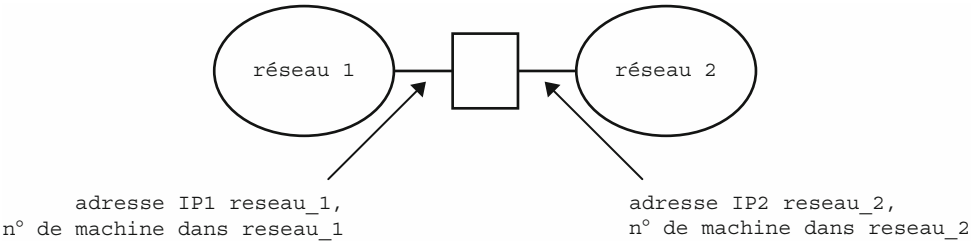

Figure 9.5 – Machine multidomiciliée

Nom logique

À chaque adresse IP, correspond un *nom logique* ou *nom symbolique* formé de caractères alphanumériques. La composition du nom symbolique repose sur la notion de *domaine*. Le domaine correspond à un groupe géographique (fr pour France, jp pour Japon) ou institutionnel (com, mil, edu) qui constitue un espace de noms géré de manière autonome. L'ensemble des domaines est organisé selon une structure hiérarchique qui rend gérable l'immense espace de noms de l'interconnexion de réseaux. Ainsi chaque domaine (hormis la racine) est lui-même inclus dans un domaine et chaque domaine est un espace autonome. Le nom symbolique d'une machine est alors formé de son nom propre suivi de l'ensemble des domaines auxquels elle appartient, chaque élément étant séparés par un point.

Ainsi la machine lionne.cnam.fr correspond à la machine de nom lionne dans le domaine cnam, appartenant lui-même au domaine fr (figure 9.6).

Le nom symbolique est utilisé pour désigner une machine parce qu'étant composé de caractères alphanumériques, il est plus aisément manipulable par un être humain. Ce nom symbolique ne correspond à rien pour le réseau virtuel et doit être

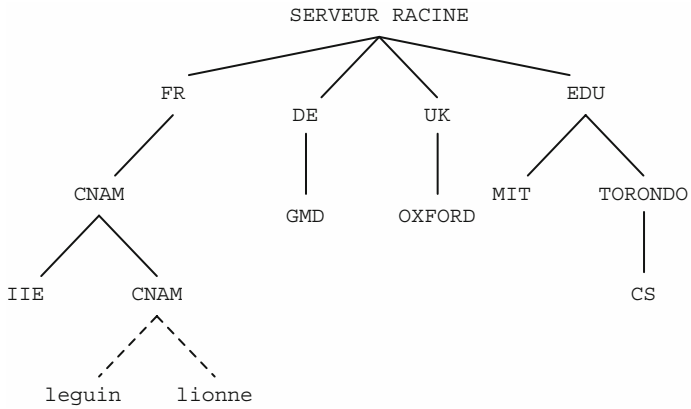

Figure 9.6 – Structure hiérarchique de l'espace des noms

converti en l'adresse IP correspondante. Cette conversion, appelée *résolution de noms*, s'effectue grâce à des serveurs de noms placés dans chaque domaine, qui connaissent la correspondance nom symbolique – adresse IP des machines d'un domaine (figure 9.7). Plus précisément, lors de la demande de raccordement d'un réseau local à l'interconnexion internet, deux machines, futurs serveurs de noms du domaine, doivent être désignées auprès du NIC.

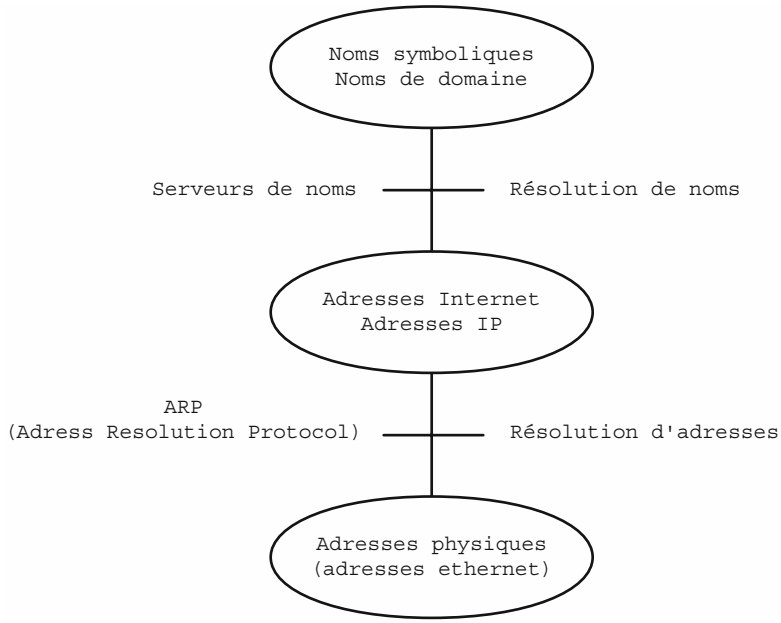

Figure 9.7 – Résolution de noms et résolution d'adresses

b) Le protocole TCP/IP

Le protocole de communication utilisé au sein de l'interconnexion de réseau internet est le protocole TCP/IP. Ce protocole né en 1980 est issu d'un projet de recherche DARPA (*Defense Advanced Research Project Agency*) du DoD (*Department of Defense*) aux États-Unis.

La pile de protocoles TCP/IP est structurée sur 4 couches (figure 9.8) :

- le niveau physique s'occupe de la gestion du médium physique ;
- le niveau IP correspond à la couche de niveau réseau ;
- le niveau UDP et TCP correspond à la couche transport ;
- enfin, le dernier niveau correspond au niveau application.

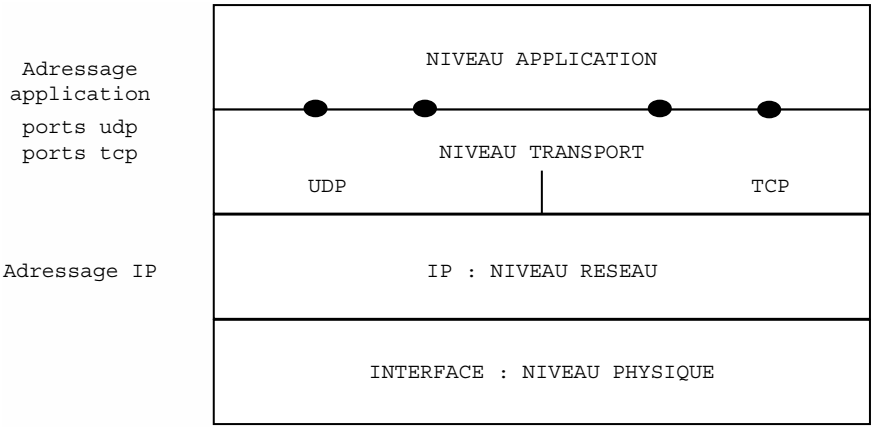

Figure 9.8 – Pile de protocoles TCP/IP

La couche réseau

La couche IP correspond au niveau réseau. Elle permet de faire un adressage de machine à machine par le biais des adresses IP de chacune des machines. La communication au niveau de la couche IP est une communication en mode minimal, c'est-à-dire en mode non connecté et sans fiabilité : il n'y a pas de garantie de bon transfert des datagrammes IP, lesquels sont acheminés indépendamment les uns des autres. La remise peut être faite dans le désordre, il peut y avoir perte de datagrammes ainsi que des duplications.

La couche transport

La couche supérieure UDP/TCP correspond au niveau transport. À ce niveau, l'adressage s'effectue d'applications à applications par le biais de la notion de *port*. Un port est un entier de 16 bits qui permet de désigner une application à travers le réseau, soit au travers du protocole UDP, soit au travers du protocole TCP.

Le protocole UDP est un protocole transport non fiable basé sur IP. Les datagrammes UDP sont acheminés indépendamment les uns des autres, il s'ensuit qu'il peut y avoir perte de datagrammes, duplication de datagrammes et la remise n'est pas forcément effectuée selon l'ordre d'émission.

Le protocole TCP, au contraire est un protocole de transport fiable orienté connexion, également basé sur IP. Avant l'échange des données, une connexion est établie entre les deux entités communicantes. Cette connexion est fiable et assure par un mécanisme d'acquittements et de contrôle de flux qu'il ne peut pas y avoir ni perte, ni duplication de messages. La connexion est rompue lorsque les deux entités communicantes ont achevé leurs échanges de données.

La communication TCP est orientée flux d'octets. Les octets émis sont remis à destination selon leur ordre d'émission. Il existe cependant un mécanisme de *données urgentes* permettant de délivrer des octets de manière prioritaire, sans respecter l'ordre d'émission initial.

La couche application

La couche application regroupe les différentes applications ou services dont le fonctionnement est basé sur une communication à travers le réseau. Chacune de ces applications est identifiée par un port, qui est un port UDP si l'application communique *via* le protocole UDP, sinon un port TCP.

Pour chacun des protocoles, un certain nombre de ports (valeurs 1 à 1023) sont réservés à des applications réseau standards. Ainsi :

- le port TCP 7 est réservé à l'application echo ;
- le port TCP 21 est réservé à l'application ftp ;
- le port UDP 513 est réservé à l'application who.

L'ensemble des valeurs des ports affectées à des applications peut être consulté dans le fichier `/etc/services`.

Les applications standards

De nombreuses applications sont disponibles au niveau de la couche application. Parmi celles-ci, on distingue :

- les applications standard disponibles sur toute implémentation du système. Ce sont les applications de transfert de fichiers (ftp), de courrier électronique (smtp) et encore de terminal virtuel (telnet) ;
- les applications courantes, potentiellement disponibles, telles que le protocole lié à l'appel de procédure à distance (XDR-RPC) ;
- les applications issues du monde Unix, telles que les commandes à distance rlogin, rsh, etc. (*remote commandes*).

Le super-démon `Inetd`

L'exécution de chacune des applications citées ci-dessus est associée à un processus démon. Chacun de ces démons est lui-même lancé par un démon de niveau supérieur,

appelé le super-démon `inetd`, qui a en charge la surveillance de l'ensemble des ports UDP et TCP de la machine.

Lors du boot de la machine, le super-démon `inetd` utilise le fichier `/etc/inetd.conf` pour connaître l'ensemble des ports sur lesquels il doit se mettre en écoute. Ce fichier comporte une ligne par service qui donne notamment les informations concernant le nom du service, le protocole qui lui est associé, le chemin de l'exécutable concerné avec d'éventuels paramètres.

Lorsqu'une requête parvient sur un port donné (par exemple le port TCP 21), le super-démon crée un processus fils qui va exécuter le code du service associé au port (ici le service ftp), créant ainsi un démon `ftpd`, qui disparaît sitôt le service rendu.

9.2 Programmation réseau

La programmation réseau sous Linux s'appuie sur l'interface de programmation *sockets*. Cette interface offre un ensemble de fonctions qui permettent l'accès aux couches de protocoles TCP/IP relativement aisément. Plus précisément, une socket représente une interface de programmation placée entre la couche application et la couche transport de la pile de protocoles TCP/IP, mais également un point de communication existant sur cette même pile. Ce point de communication est notamment repéré par une adresse composée de l'adresse IP de la machine sur lequel ce point est ouvert et d'un numéro de port UDP ou TCP en fonction du protocole transport choisi pour réaliser le transfert de données de l'application liée au port (figure 9.9).

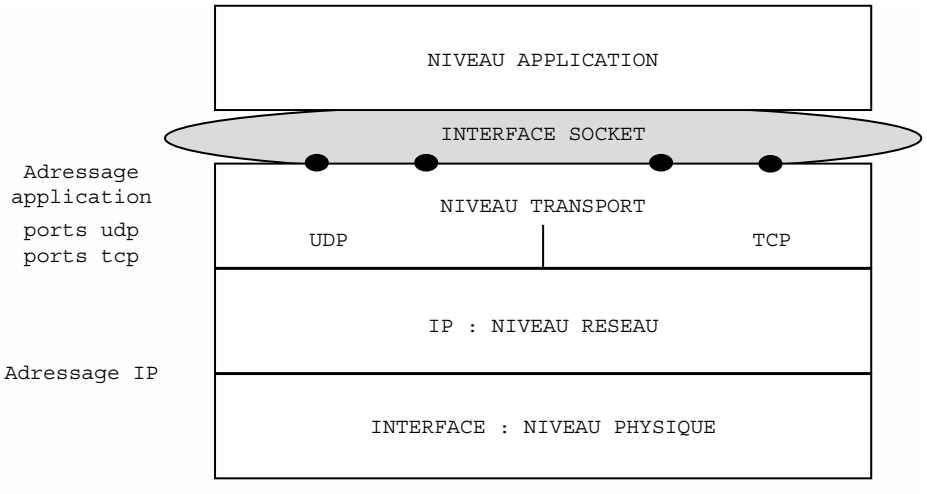

Figure 9.9 – Interface socket

9.2.1 Les utilitaires pour la programmation socket

a) Primitives de conversion pour la représentation des entiers

Une socket est donc associée à une adresse composée d'une part d'une adresse IP, d'autre part d'un numéro de port UDP ou TCP. Comme nous l'avons précisé précédemment, une adresse IP est un entier de 32 bits, donc de quatre octets tandis que le port est un entier de 16 bits donc deux octets.

Les machines admettent deux formats de représentation pour les entiers longs (4 octets) et les entiers courts (2 octets).

Dans le format appelé format Petit Boutiste (*Big Endian*), les octets composant un mot de 16 bits ou de 32 bits sont numérotés de la gauche vers la droite, c'est-à-dire que l'octet de poids faible reçoit le plus gros numéro.

Dans le format appelé format Gros Boutiste (*Little Endian*), les octets composant un mot de 16 bits ou de 32 bits sont numérotés de la droite vers la gauche, c'est-à-dire que l'octet de poids faible reçoit le plus petit numéro.

Imaginons à présent une machine travaillant au format Gros Boutiste, qui envoie l'entier court $(01110000\ 00001110)_2$ sur le réseau soit l'octet 0 $(01110000)_2$ et l'octet 1 $(00001110)_2$. Si la machine réceptrice travaille au format Petit Boutiste, elle interprète l'entier reçu comme étant $(00001110\ 01110000)_2$ c'est-à-dire qu'elle inverse les octets composant le nombre !

Pour remédier à ce problème, un format de communication pour les entiers courts et les entiers longs a été instauré sur le réseau. Ce format, appelé format Réseau, en opposition aux formats locaux et propres des machines, correspond à un format Gros Boutiste.

Toute application émettant des adresses IP ou des numéros de ports sur le réseau doit au préalable convertir ceux-ci vers le format réseau.

Ainsi une machine en émission convertira toujours les adresses IP et les numéros de ports de son format local vers le format réseau tandis qu'une machine en réception convertira toujours les adresses IP et les numéros de port du format réseau vers le format local.

Quatre primitives sont fournies pour permettre la conversion des entiers courts et des entiers longs du format local vers le format réseau et vice versa. Nous donnons leurs prototypes :

```
Fonctions de conversion d'entiers du format local au format réseau
#include <netinet/in.h>
/* conversion d'un entier court valentier du format local au format
réseau */
u_short htons (u_short valentier);
/* conversion d'un entier long valentier du format local au format réseau
*/
u_long htonl (u_long valentier);
/* conversion d'un entier court valentier du format réseau au format
local */
u_short ntohs (u_short valentier);
```

```
/* conversion d'un entier long valentier du format réseau au format local */
u_long ntohl (u_long valentier);
```

b) Correspondance entre le nom symbolique et l'adresse IP d'une machine

Le fichier `/etc/hosts` local à la machine ainsi que l'interrogation des serveurs de noms permet de faire la correspondance entre le nom symbolique d'une machine et son adresse IP (résolution de noms). Ce fichier contient une ligne par machine et donne pour chaque machine son adresse IP, son nom symbolique principal, ses alias et éventuellement un commentaire.

/etc/hosts

Correspondance adresse IP et nom symbolique

adresse IP	nom symbolique	alias	commentaire
163.173.128.6	asimov.cnam.fr	cnam	iris

La primitive `gethostbyname(nom)` permet d'obtenir les informations relatives à une entrée du fichier `/etc/hosts` en spécifiant comme paramètre de recherche le nom symbolique de la machine. Cette fonction retourne une structure de type `hostent` dont la composition est donnée ci-après. La ligne `#define h_addr h_addr_list[0]` permet de connaître la première adresse IP de la machine.

```
struct hostent {
               char *h_name;        /* nom officiel de la machine */
               char **h_aliases;    /* liste des alias */
               int h_addrtype;      /* type de l'adresse */
               int h_length;        /* longueur de l'adresse */
               char **h_addr_list;  /* liste des adresses */
#define h_addr  h_addr_list[0]/* premier élément de la liste */};

/* obtenir l'entrée /etc/host de la machine de nom «nom» */

#include <netdb.h>
struct hostent *gethostbyname (char *nom);
```

En cas d'erreur, la fonction positionne non pas directement la variable `errno`, mais une autre variable globale `h_errno`, déclarée dans le fichier `<netdb.h>`. Cette variable peut prendre les valeurs suivantes :

- `NETDB_SUCCES`, aucune erreur ;
- `HOST_NOT_FOUND`, l'hôte n'a pas été trouvé ;
- `TRY_AGAIN`, un problème temporaire affecte le serveur de noms. Réitérer la demande ;
- `NO_RECOVERY`, une erreur critique est intervenue dans le processus de résolution de noms ;
- `NO_ADRESS`, l'hôte, connu du serveur de noms, n'a pas d'adresse valide.

9.2.2 L'interface socket

L'interface de programmation socket est entièrement compatible avec l'interface du système de gestion de fichiers puisque, comme nous l'avons vu au chapitre 3, une socket constitue un type de fichier pour Ext2. Ainsi, lors de la création d'une socket, un descripteur de socket est alloué dans la table des fichiers ouverts du processus créateur. Ce descripteur pointe sur une inode à laquelle n'est associé aucun bloc de données. Aussi, la socket répond aux critères d'héritage père fils qui s'appliquent aux descripteurs de fichiers.

a) Création d'une socket

La création d'une socket s'effectue par un appel à la primitive `socket()` dont le prototype est :

```
#include <sys/socket.h>
int socket (int domaine, int type, int protocole);
```

Le premier paramètre `domaine` définit le domaine auquel la socket appartient. En effet, une socket peut appartenir soit au domaine `AF_UNIX` et être alors un outil de communication purement local à une machine, soit appartenir au domaine `AF_INET` et être alors un outil de communication accessible *via* l'interconnexion de réseaux. Dans le cadre de ce chapitre, une socket sera toujours du type `AF_INET`.

Le second paramètre `type` définit le type de la socket et par là même les propriétés de la communication sur laquelle elle se base. Les trois principaux types sont :

- le type `SOCK_DGRAM` qui correspond à une communication en mode datagramme, non connectée ;
- le type `SOCK_STREAM` qui correspond à une communication en mode connectée, orientée flux d'octets ;
- le type `SOCK_RAW` qui permet d'accéder à des protocoles de plus bas niveau, de type IP. Seul le super-utilisateur peut utiliser ce type.

Le troisième paramètre `protocole` définit le protocole de communication utilisé par la socket. Ce paramètre appliqué à TCP/IP prend essentiellement les valeurs `IPPROTO_UDP` pour designer le protocole UDP ou `IPPROTO_TCP` pour désigner le protocole TCP. Le champ peut être mis à zéro ce qui laisse alors le système choisir le bon protocole en fonction du type spécifié. Les différents protocoles admis dans ce champ sont définis dans le fichier `/etc/protocols`.

La primitive de création `socket()` retourne un entier correspondant au descripteur de la socket créée. À l'issue de l'opération de création, la socket tout comme un fichier est utilisable par le processus créateur et les processus créés par ce processus créateur après la création de la socket.

En cas d'échec la primitive renvoie la valeur – 1 et positionne la variable `errno` avec l'une des valeurs suivantes :

- `EINVAL`, le domaine est invalide ;

- EPROTONOSUPPORT, le type est incohérent avec le protocole ou le domaine ;
- EACCES, les droits sont insuffisants pour créer la socket.

b) Attachement d'une adresse application à une socket

Pour que des processus extérieurs à la famille du processus créateur puissent accéder à la socket il faut maintenant attacher une adresse réseau à cette socket. La primitive permettant d'attacher une adresse à une socket est la primitive `bind()` dont le prototype est :

```
int bind (int sock, struct sockaddr_in *p_adresse, int lg);
```

Le premier paramètre `sock` correspond au descripteur de socket créé par l'appel `socket()`. Le second paramètre `*p_adresse` est un pointeur sur une structure de type `sockaddr_in`. Cette structure que nous allons détailler permet de stocker une adresse d'application c'est-à-dire notamment une adresse IP et un numéro de port. Le dernier paramètre `lg` correspond à la taille de la structure d'adresse en octets.

En cas d'échec la primitive renvoie la valeur – 1 et positionne la variable `errno` avec l'une des valeurs suivantes :

- EINVAL, la socket a déjà une adresse ou elle est déjà connectée, ou le paramètre `lg` est incorrect ;
- EBADF, ENOTSOCK, le descripteur de socket est invalide ;
- EADDRINUSE, l'adresse est déjà utilisée ;
- EACCES, les droits sont insuffisants pour utiliser l'adresse indiquée.

Construction de l'adresse attachée à une socket

La structure `sockaddr_in`

La structure de type `sockaddr_in` est composée de 4 champs qui permettent de définir l'adresse d'application associée à une socket :

```
struct sockaddr_in {
                    short sin_family; /* AF_INET */
                    ushort sin_port; /* le numéro de port */
                    struct in_adrr sin_addr; /* l'adresse IP */
                    char sin_zero[8]; /* huit 0 */
                    };
```

Le premier champ `sin_family` correspond au type de l'adresse associée à la socket. Dans le cas présent, la valeur stockée dans ce champ sera toujours `AF_INET`. Le deuxième champ de la structure `sin_port` permet de stocker le numéro de port UDP ou TCP sur lequel l'application utilisant la socket est en écoute. Le troisième champ `sin_addr` correspond à une adresse IP de machine. Enfin, le quatrième champ `sin_zero` est constitué de 8 zéros.

Une adresse IP est elle-même définie par une structure de type `in_addr` constituée d'un seul champ de type entier long.

```
struct in_addr {u_long s_addr;};
```

Remplissage du champ adresse IP de la structure `sockaddr_in nom`

Le champ `sin_addr` contient l'adresse IP associée à la socket. Ce champ peut être complété de deux manières différentes :

- la socket correspond à une adresse d'application locale. Le champ prend la valeur `INADDR_ANY`. Cette constante symbolique représente une des adresses de la machine locale ;

```
nom.sin_addr.s_addr = INADDR_ANY;
```

- la socket correspond à une adresse d'application distante. Le champ est rempli en faisant appel à la fonction `gethostbyname()`, qui permet de connaître l'adresse IP du serveur distant, connu sous son nom symbolique, par exemple `dirac.cnam.fr`. La primitive `bcopy (void *adresse_depart, void *adresse_dest, size_t nombre)` effectue une copie de `nombre` octets situés à l'adresse `adresse_depart` vers l'adresse `adresse_dest`.

```
h = gethostbyname("dirac.cnam.fr");
/* copie d'octets à octets */
bcopy(h->h_addr,&nom.sin_addr,h->h_length);
```

Remplissage du champ numéro de port de la structure `sockaddr_in nom`

Le champ `sin_port` de la structure `sockaddr_in` peut être mis à 0. Dans ce cas, le système choisit lui-même un numéro de port libre, au moment de l'opération d'attachement de la socket à une adresse d'application.

La primitive `getsockname()` permet alors de connaître les caractéristiques de l'adresse d'application attachée à la socket. Au retour de l'appel, la structure `sockaddr_in` dont l'adresse est `*p_adresse` contient l'adresse d'application attachée à la socket `sock`. `lg` est la taille de la structure `p_adresse`.

```
int getsockname (int sock, struct sockaddr_in *p_adresse, int *lg);
```

c) Programmation d'une ouverture d'un point de communication socket

Nous donnons un code correspondant aux actions de création d'une socket de type `SOCK_DGRAM`, attachée localement au port 11203. Dans ce code, la fonction `bzero (char *adr, int longueur)` permet de réinitialiser à 0 la zone mémoire débutant à l'adresse `adr` et dont la taille est de `longueur` octets.

```
/*****************************************************************/
/*              Création et attachement d'une socket             */
/*****************************************************************/
#define PORT 11203
main()
{
int sock;
struct sock_addr_in nom;
int longueur;
```

```
/* Création de la socket */
sock = socket(AF_INET,SOCK_DGRAM,0);

/* préparation de l'adresse */
/* la zone mémoire pour l'adresse est mise à 0 */
bzero(&nom,sizeof(nom));
nom.sin_family = AF_INET;
nom.sin_port = htons(PORT); /* passage au format réseau */
nom.sin_addr.s_addr = INADDR_ANY; /* n'importe quelle adresse IP de la
machine locale */

/* attachement de la socket */
bind(sock, &nom, sizeof(nom));

printf ("descripteur de la socket  %d", sock);
/* passage au format local */
printf ("valeur du numéro de port  %d", ntohs(nom.sin_port));
}
```

d) Fermeture d'une socket

Une socket est fermée par un appel à la primitive `close()`. La socket devient inutilisable. Cependant, dans le cas d'une communication en mode connecté, la socket est réellement fermée lorsque toutes les données émises ont bien été transmises au destinataire et que celui-ci a acquitté leur bonne réception. Aussi, après un appel à la primitive `close()`, la socket peut continuer à exister et ce jusqu'à la fin de la transmission des dernières données émises.

Dans ce cas, une nouvelle exécution du processus venant de fermer la socket, fait échouer l'appel système `bind()` avec comme erreur `EADDRINUSE` (*Socket already in Used*).

e) Programmation d'une communication en mode UDP

La communication en mode UDP est un mode de communication non fiable, sans connexion. Une fois, les points de communication créés sur la pile de protocole, le client et le serveur s'échangent directement les données sous forme de datagrammes UDP. Chaque envoi doit contenir l'adresse du destinataire.

Création des points de communication

Chaque entité communicante, le client et le serveur, commence par créer une socket de type `SOCK_DGRAM`, puis chacun attache à cette socket son adresse locale, formée du numéro de port de l'application (client ou serveur) et de l'adresse IP des machines client ou serveur.

Échange de données en mode UDP

L'échange de données s'effectue sous forme de datagrammes. La structure du message est préservée à la réception, c'est-à-dire qu'une opération de réception permet toujours de récupérer l'intégralité d'un datagramme émis sur une opération d'envoi.

La primitive `sendto()` permet l'envoi d'un datagramme. Son prototype est le suivant :

```
#include <sys/types.h>
#include <sys/socket.h>
int sendto(int sock, char *msg, int lg, int option, struct sockaddr_in
*p_dest, int lgdest);
```

Le datagramme placé à l'adresse mémoire `*msg` est émis sur la socket d'émission `sock`, à destination de l'application dont l'adresse est décrite dans la structure `*p_dest`. Le paramètre `lg` correspond à la taille en octets du message à transmettre. Le paramètre `lgdest` correspond à la taille en octets de la structure d'adresse. Le paramètre `option` a toujours la valeur 0. La primitive renvoie le nombre d'octets effectivement émis en cas de succès et sinon la valeur – 1.

La primitive `recvfrom()` permet la réception d'un datagramme. Son prototype est le suivant :

```
#include <sys/types.h>
#include <sys/socket.h>
int recvfrom(int sock, char *msg, int lg, int option, struct sockaddr_in
*p_exp, int *lgexp);
```

Un datagramme dont le contenu est placé dans la zone mémoire d'adresse `*msg` et de taille `lg` octets est reçu sur la socket de réception `sock`. L'adresse de l'application émettrice du datagramme est placée dans la structure d'adresse `*p_exp` et de taille `*lgexp` octets. La primitive renvoie le nombre d'octets effectivement reçus en cas de succès et la valeur – 1 sinon.

L'exécution de cette primitive est bloquante, c'est-à-dire qu'un processus effectuant un appel à la primitive `recvfrom()` est bloqué jusqu'à avoir effectivement reçu un datagramme.

Par défaut, la lecture sur la socket est destructrice. Cependant, si le paramètre option est positionné à la valeur `MSG_PEEK`, alors le message est lu sans être extrait de la socket.

En cas d'échec, ces deux primitives positionnent la variable `errno` avec les valeurs suivantes :

- `EBADF, ENOTSOCK, sock` n'est pas un descripteur valide ;
- `ECONNREFUSED`, la communication est refusée par l'hôte distant ;
- `ENOTCONN`, la socket est associée à un mode connecté ;
- `EAGAIN`, un time-out a été levé sur l'opération de réception alors que toutes les données n'ont pas été reçues ;
- `EINTR`, l'appel système a été interrompu par un signal ;
- `EFAULT`, la zone de réception/émission n'est pas valide ;
- `ENOMEM`, il n'y pas assez de mémoire ;
- `ENOBUFS`, l'émission des données est stoppée temporairement suite à une congestion du réseau ;
- `EINVAL`, un des arguments n'est pas valide.

Un exemple de programmation en mode UDP

Nous donnons un exemple de client-serveur en mode UDP. Le serveur maintient une table à 10 entrées contenant les numéros de téléphone de musiciens célèbres. Le client interroge le serveur pour connaître le numéro de téléphone d'un éventuel abonné. Le serveur recherche cet abonné dans sa table : s'il l'y trouve, il renvoie le numéro de téléphone qui lui est associé et sinon un message d'erreur.

```
/*********************************************************************/
/*                            Client UDP                             */
/*********************************************************************/
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#define PORTS 6260   /* port du serveur */
#define PORTC 6259   /* port du client */

main()

{
struct hostent *h;
struct sockaddr_in sin;
char abonne[30], telephone[10];
int sock;


if((sock = socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP)) < 0)
{
perror("socket");
exit(1);
}
/* préparation de l'adresse */
/* la zone mémoire pour l'adresse est mise à 0 */
bzero(&sin,sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_port = htons(PORTC); /* passage au format réseau */
sin.sin_addr.s_addr = INADDR_ANY; /* n'importe quelle adresse IP de la
machine locale */
if(bind(sock, &sin, sizeof(sin)) < 0)
{
perror("bind");
exit(2);
}

if(!(h=gethostbyname("dirac.cnam.fr")))
{
perror();
exit(3);
```

```
}
bzero(&sin,sizeof(sin));
sin.sin_family = AF_INET;
bcopy(h->h_addr,&sin.sin_addr,h->h_length);
sin.sin_port = htons(PORTS);

printf ("Donnez un nom d'abonné \n");
fgets (abonne, 30, stdin);
sendto(sock, abonne, strlen(abonne), 0,&sin, sizeof(sin));
lg = sizeof(sin);
recvfrom(sock, telephone, strlen(telephone), 0,&sin, &lg);
if (strcmp(telephone, "erreur !! ") == 0)
     printf("désolé, l'abonné %s n'existe pas\n", abonne_);
else
     printf("l'abonné %s a pour numero de téléphone %s\n", abonne,
telephone);

close (sock);
}
/*********************************************************************/
/*                              Serveur UDP                          */
/*********************************************************************/
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#define PORTS 6260   /* port du serveur */
struct sockaddr_in sin;
struct sockaddr_in adresse_client;
struct abonne
{
     char nom[30];
     char telephone [10];
} annuaire [] = {
"handel", "0125468956",
"dowland", "0156897844",
"byrd", "0689786321",
"purcell", "0123568974",
"vivaldi", "0569875586",
"wagner", "0123568974",
"mozart", "0213457896",
"beethoven","0563897541",
"weber", "0156897447",
"monteverdi", "0125255699"
};

main(argc,argv)
```

```
char **argv;
{
int namelen, sock;
int pid, status;

if((sock = socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP)) < 0)
{
perror("socket");
exit(1);
}
/* préparation de l'adresse */
/* la zone mémoire pour l'adresse est mise à 0 */
bzero(&sin,sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_port = htons(PORTS); /* passage au format réseau */
sin.sin_addr.s_addr = INADDR_ANY; /* n'importe quelle adresse IP de la
machine locale */
if(bind(sock, &sin, sizeof(sin)) < 0)
{
perror("bind");
exit(2);
}

service(sock);
}
}

/* fonction qui remplit le service et cherche dans l'annuaire le
correspondant demandé */
service(nsock)
int nsock;
{
int n, i;
char line[30];

lg = sizeof(adresse_client);
n= recvfrom (sock, line, 30, 0, &adresse_client, &lg_adresse);
i = 0;
while (i < 10)
{
if (strcmp(line,annuaire[i].nom) == 0)
{
       sendto(sock, annuaire[i].telephone, 10, 0, &adresse_client,
         lg_adresse);
       return;
}
i = i + 1;
}
sendto(sock, "erreur !! ", 10, 0, &adresse_client, lg_adresse);
}
```

f) Programmation d'une communication en mode TCP

La communication en mode TCP est un mode de communication fiable, orientée connexion. Une fois, les points de communication créés sur la pile de protocole, le client et le serveur doivent établir une connexion avant de pouvoir s'échanger des données sous forme d'un flux d'octets TCP. L'échange des adresses entre émetteur et récepteur s'effectue au moment de l'établissement de la connexion. Lorsque l'échange est terminé, la connexion est fermée.

Établissement d'une connexion TCP/IP

Chaque entité communicante, le client et le serveur, commence par créer une socket de type `SOCK_STREAM`, puis chacun attache à cette socket son adresse locale, formée du numéro de port de l'application (client ou serveur) et de l'adresse IP des machines client ou serveur.

Le serveur, ensuite, se prépare en vue de recevoir des connexions de la part de clients. Pour cela, il fait d'abord appel à la primitive `listen()` pour dimensionner sa file de connexions pendantes sur sa socket initiale que nous appellerons la socket d'écoute.

```
#include <sys/socket.h>
int listen (int sock, int nb);
```

Dans le prototype de la fonction, `nb` correspond au nombre maximal de connexions pendantes (en attente de service) sur la socket d'écoute `sock`. La primitive retourne 0 en cas de succès et – 1 sinon. La variable `errno` prend alors une des valeurs suivantes :

- `EBADF, ENOTSOCK, sock` n'est pas un descripteur valide ;
- `EOPNOTSUPP`, le type de la socket ne supporte pas l'opération `listen()`.

Le serveur se met alors en attente de connexion par un appel à la primitive `accept()`. Cette primitive est bloquante, c'est-à-dire que le serveur reste bloqué sur cette primitive jusqu'à ce qu'un client établisse effectivement une connexion.

```
#include <sys/types.h>
#include <sys/socket.h>
int accept (int sock, struct sockaddr_in *p_adr, int *p_lgadr);
```

Dans le prototype de la fonction, `sock` correspond à la socket d'écoute. L'adresse du client connecté est récupérée dans la structure de type `sockaddr_in` placée à l'adresse `*p_adr` et dont la taille est de `*p_lgadr` octets.

De son côté, le client demande l'établissement d'une connexion en faisant appel à la primitive `connect()`. De la même manière, la primitive `connect()` est bloquante pour le client, c'est-à-dire que le client est bloqué jusqu'à ce que le serveur accepte la connexion.

```
#include <sys/types.h>
#include <sys/socket.h>
int connect (int sock, struct sockaddr_in *p_adr,int lgadr);
```

Dans le prototype de la fonction, `sock` correspond à la socket du client. L'adresse du serveur vers lequel s'effectue la connexion est placée dans la structure de type `sockaddr_in` dont l'adresse est `*p_adr` et la taille est de `lgadr` octets.

Au moment de l'établissement de la connexion, le serveur crée une nouvelle socket que l'on appelle la socket de service. Le descripteur de cette nouvelle socket est retourné par la primitive `accept()`. Cette socket de service est la socket sur laquelle s'effectuent ensuite tous les échanges de données entre les deux entités connectées.

En cas d'échec, les primitives `accept()` et `connect()` renvoient la valeur – 1. La variable `errno` est positionnée avec les valeurs suivantes :

- `EBADF`, le descripteur de socket est invalide ;
- `ENOTSOCK`, le descripteur référence un fichier et pas une socket ;
- `EFAULT`, la zone d'adresse n'est pas valide ;
- `EPERM`, la connexion est interdite par les machines pare-feu ;
- `ENOBUFS`, `ENOMEM`, la mémoire est insuffisante ;
- `EISCONN`, la socket est déjà connectée ;
- `ECONNREFUSED`, il n'y a personne pour accepter la connexion ;
- `ENETUNREACH`, le réseau est inatteignable ;
- `EOPNOTSUPP`, la socket désignée n'est pas de type connectée.

Échange de données en mode TCP

L'échange de données s'effectue sous forme de flux d'octets. Il faut prendre garde au fait que la structure des messages n'est pas conservée. Le découpage en messages identifiables correspondant aux différents envois n'est pas préservé sur la socket destinataire. Ainsi :

- une opération de lecture peut provenir de différentes opérations d'écriture ;
- une opération d'écriture d'une chaîne longue peut provoquer son découpage, les différents fragments étant accessibles sur la socket destinataire.

La primitive `write()` permet l'envoi à travers le point de communication `sock` des `lg` octets constituant le message placé à l'adresse `*msg`.

```
int write (int sock, char *msg, int lg);
```

La primitive `read()` permet la lecture à travers le point de communication `sock` de `lg` octets réceptionnés à l'adresse `*msg`.

```
int read (int sock, char *msg, int lg);
```

Un exemple de programmation TCP : réalisation d'un serveur itératif

Nous reprenons comme exemple l'application d'annuaire téléphonique précédemment programmé sous la forme d'une communication en mode UDP.

Le serveur réalisé est ici un serveur de type itératif composé d'un seul processus qui accepte une nouvelle connexion de la part d'un client, traite la requête de ce client et lui répond, avant de venir accepter une éventuelle nouvelle demande de connexion d'un autre client.

Le serveur utilise la fonction `lireligne()` pour lire le message envoyé par un client, c'est-à-dire le nom de l'abonné recherché. Cette fonction lit caractère à caractère les données parvenues sur la socket de service, jusqu'à trouver le caractère « \n », assurant ainsi de retourner effectivement les données correspondant à un envoi de client. Cette façon d'opérer est nécessaire du fait de la non-préservation de la structure des messages lors d'une communication en mode TCP et de la longueur variable des messages envoyés par les clients.

```
/*********************************************************************/
/*                          Client TCP                               */
/*********************************************************************/
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#define PORTS 6260   /* port du serveur */
#define PORTC 6259   /* port du client */

main()

{
struct hostent *h;
struct sockaddr_in sin;
char abonne[30], telephone[10];
int sock;


if((sock = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP)) < 0)
{
perror("socket");
exit(1);
}
/* préparation de l'adresse */
/* la zone mémoire pour l'adresse est mise à 0 */
bzero(&sin,sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_port = htons(PORTC); /* passage au format réseau */
sin.sin_addr.s_addr = INADDR_ANY; /* n'importe quelle adresse IP de la
machine locale */
if(bind(sock, &sin, sizeof(sin)) < 0)
{
perror("bind");
exit(2);
}

if(!(h=gethostbyname("dirac.cnam.fr")))
{
perror();
```

```
exit(3);
}
bzero(&sin,sizeof(sin));
sin.sin_family = AF_INET;
bcopy(h->h_addr,&sin.sin_addr,h->h_length);
sin.sin_port = htons(PORTS);

if(connect(sock,&sin,sizeof(sin)) < 0)
{
perror("connect");
exit();
}

printf ("Donnez un nom d'abonné \n");
fgets (abonne, 30, stdin);
write(sock, abonne, strlen(abonne));
read(sock, telephone,10);
if (strcmp(telephone, "erreur !! ") == 0)
     printf("désolé, l'abonné %s n'existe pas\n", abonne);
else
     printf("l'abonné %s a pour numero de téléphone %s\n", abonne,
                          telephone);

close (sock);
}
/*********************************************************************/
/*                      Serveur TCP itératif                         */
/*********************************************************************/
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#define PORTS 6260   /* port du serveur */
struct sockaddr_in sin;
struct sockaddr_in adresse_client;
struct abonne
{
      char nom[30];
      char telephone [10];
} annuaire [] = {
        "handel", "0125468956",
        "dowland", "0156897844",
        "byrd", "0689786321",
        "purcell", "0123568974",
"vivaldi", "0569875586",
"wagner", "0123568974",
"mozart", "0213457896",
```

```
"beethoven","0563897541",
"weber", "0156897447",
"monteverdi", "0125255699"
};

main(argc,argv)
char **argv;
{
int namelen, newsock, sock;
int pid, status;

if((sock = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP)) < 0)
{
perror("socket");
exit(1);
}
/* préparation de l'adresse */
/* la zone mémoire pour l'adresse est mise à 0 */
bzero(&sin,sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_port = htons(PORTS); /* passage au format réseau */
sin.sin_addr.s_addr = INADDR_ANY; /* n'importe quelle adresse IP de la
machine locale */
if(bind(sock, &sin, sizeof(sin)) < 0)
{
perror("bind");
exit(2);
}

if(listen(sock,5) < 0)
{
perror("listen");
exit(4);
}
for (;;)
{
if((newsock = accept(sock,&sin,&namelen)) < 0)
{
perror("accept");
exit(5);
}
service(newsock);
}
}

/* fonction qui remplit le service et cherche dans l'annuaire le
correspondant demandé */
service(nsock)
int nsock;
{
int n, i;
```

```
char line[30];

n = lireligne(nsock,line,30);
if (n < 0) {
perror ("Pb lecture de ligne");
return;
}
i =0;
while (i < 10)
{

if (strcmp(line,annuaire[i].nom) == 0)
{
       write(nsock, annuaire[i].telephone, 10);
       close(nsock);
       return;
}
i = i + 1;
}
write(nsock, "erreur !! ", 10);
close(nsock);
}

/* fonction qui lit une ligne sur la socket de réception */
int lireligne(fd,pt,maxlong)
int fd;
char *pt;
int maxlong;

{
int n, rr;
char c;


for (n = 1; n < maxlong; n++)
{
rr = read(fd, &c, 1);
if (c == '\n')
       break;
*pt++ = c;
}
*pt = '\0';
return(n);
}
```

Un exemple de programmation TCP : réalisation d'un serveur parallèle

Le serveur donné en exemple dans le paragraphe précédent est un serveur itératif. Une fois que le serveur a accepté une connexion de la part d'un client, les clients suivants sont mis en attente jusqu'à ce que le serveur ait fini de traiter le premier client et revienne sur l'`accept()` pour accepter une nouvelle connexion. Durant tout

le traitement d'un client connecté, le serveur travaille sur la socket de service et la socket d'écoute n'est pas utilisée.

Il est plus performant de créer un serveur parallèle dont l'architecture logicielle est la suivante :

- un processus père accepte les connexions sur la socket d'écoute et crée une socket de service pour la connexion acceptée ;
- pour chaque connexion acceptée, le processus père crée un processus fils. Ce fils hérite naturellement de la socket d'écoute et de la socket de service ouverte par son père ;
- le processus fils traite la requête du client à travers la socket de service. La socket d'écoute lui est inutile, donc il la ferme ;
- le père est libéré du service du client réalisé par son fils. Il ferme la socket de service et retourne immédiatement accepter de nouvelles connexions sur la socket d'écoute.

Dans ce cas de figure, on a donc autant de fils créés que de connexions acceptées par le père.

Nous donnons le code correspondant pour le même exemple que précédemment. Seul le processus serveur est modifié.

```
/*********************************************************************/
/*                      Serveur TCP parallèle                        */
/*********************************************************************/
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#define PORTS 6260   /* port du serveur */
struct sockaddr_in sin;
struct sockaddr_in adresse_client;
struct abonne
{
      char nom[30];
      char telephone [10];
} annuaire [] = {
        "handel", "0125468956",
        "dowland", "0156897844",
        "byrd", "0689786321",
        "purcell", "0123568974",
"vivaldi", "0569875586",
"wagner", "0123568974",
"mozart", "0213457896",
"beethoven","0563897541",
"weber", "0156897447",
"monteverdi", "0125255699"
```

```
};

main(argc,argv)
char **argv;
{
int namelen, newsock, sock;
int pid, status;

if((sock = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP)) < 0)
{
perror("socket");
exit(1);
}
/* préparation de l'adresse */
/* la zone mémoire pour l'adresse est mise à 0 */
bzero(&sin,sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_port = htons(PORTS); /* passage au format réseau */
sin.sin_addr.s_addr = INADDR_ANY; /* n'importe quelle adresse IP de la
machine locale */
if(bind(sock, &sin, sizeof(sin)) < 0)
{
perror("bind");
exit(2);
}
if(listen(sock,5) < 0)
{
perror("listen");
exit(4);
}
for (;;)
{
if((newsock = accept(sock,&sin,&namelen)) < 0)
{
perror("accept");
exit(5);
}
pid = fork();
if (pid == -1)
      printf ("erreur creation de processus");
else
if (pid == 0)
      service(newsock);
}
}
```

Traitement des fils zombies

On remarque ici que le processus père n'effectue pas d'appel à la primitive `wait()` pour récupérer la mort de ses fils. En conséquence, ceux-ci vont rester dans l'état zombie.

Pour éviter une saturation de la table des processus, on ajoute la ligne suivante au code du serveur ce qui permet au processus père d'ignorer le signal mort du fils et de détruire automatiquement les processus zombies (*cf.* chapitre 6) :

```
signal (SIGCHLD, SIG_IGN);
```

Cet appel à la fonction `signal()` doit être ajouté avant la boucle d'acceptation des connexions.

Envoi de données urgentes

Le protocole TCP assure que les octets d'un message sont délivrés au processus récepteur dans l'ordre de leur émission. Ainsi, il est impossible que des octets « doublent » d'autres octets au moment de la réception.

Dans certains cas, il peut cependant être nécessaire de délivrer des données urgentes à l'application destinatrice, plus prioritaires que les données précédemment émises et devant être remises au plus tôt.

Pour pouvoir réaliser cela, le protocole TCP offre un mécanisme de données urgentes. Les données émises en étant qualifiées de données urgentes, sont remises aussitôt au destinataire, sans tenir compte des éventuelles données précédemment reçues et en attente de délivrance à l'application destinataire. L'arrivée de données urgentes entraîne la délivrance au processus récepteur du signal `SIGURG`. On notera qu'un seul octet de données urgentes est autorisé à chaque envoi.

La programmation d'un envoi et d'une réception de données urgentes s'effectue de la manière suivante :

- l'envoi et la réception de la donnée urgente `durg` s'effectuent à l'aide de deux primitives `send()` et `recv()`. Dans ces deux primitives, l'option `MSG_OOB` qualifie le caractère envoyé `durg` comme donnée urgente ;

```
send (int sock, char durg, 1, MSG_OOB)
recv (int sock, char durg, 1, MSG_OOB)
```

- le processus récepteur des données urgentes traite les données urgentes reçues par l'intermédiaire d'un handler associé au signal `SIGURG`. Il doit auparavant s'être rendu propriétaire de la socket. Pour cela, on utilise la primitive `ioctl()` déjà rencontrée au chapitre 4.

```
pid = getpid();
ioctl(sock, SIOSCPGRP, &pid);
```

Dans l'exemple suivant, le client envoie une donnée urgente au serveur lorsqu'il prend en compte le signal `SIGINT` (frappe par l'utilisateur de la séquence « CTRL C »). La réception de cette donnée urgente entraîne la terminaison du serveur.

```
/*****************************************************************/
/*                          Client OOB                           */
/*****************************************************************/

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

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <signal.h>
#define PORTS 6260   /* port du serveur */
#define PORTC 6259   /* port du client */

/* handler interruption: envoi données urgentes */
void handler_interruption()
{
send(sock, "A", 1, MSG_OOB);
printf("Handler interruption, données urgentes\n");
exit();
}
main()
{
struct hostent *h;
struct sockaddr_in sin;
char abonne[30], telephone[10];
int sock;


if((sock = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP)) < 0)
{
perror("socket");
exit(1);
}
/* préparation de l'adresse */
/* la zone mémoire pour l'adresse est mise à 0 */
bzero(&sin,sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_port = htons(PORTC); /* passage au format réseau */
sin.sin_addr.s_addr = INADDR_ANY; /* n'importe quelle adresse IP de la
machine locale */
if(bind(sock, &sin, sizeof(sin)) < 0)
{
perror("bind");
exit(2);
}
if(!(h=gethostbyname("dirac.cnam.fr")))
{
perror();
exit(3);
}
bzero(&sin,sizeof(sin));
sin.sin_family = AF_INET;
bcopy(h->h_addr,&sin.sin_addr,h->h_length);
sin.sin_port = htons(PORTS);

signal(SIGINT, handler_interruption);
if(connect(sock,&sin,sizeof(sin)) < 0)
```

```
{
perror("connect");
exit();
}
printf ("Donnez un nom d'abonné \n");
fgets (abonne, 100, stdin);
write(sock, abonne, strlen(abonne));
read(sock, telephone,10);
if (strcmp(telephone, "erreur !! ") == 0)
     printf("désolé, l'abonne %s n'existe pas\n", abonne);
else
     printf("l'abonné %s a pour numéro de téléphone %s\n", abonne,
                          telephone);
close (sock);
}
/*********************************************************************/
/*                            Serveur OOB                            */
/*********************************************************************/
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <signal.h>
#define PORTS 6260   /* port du serveur */
struct sockaddr_in sin;
struct sockaddr_in adresse_client;
struct abonne
{
      char nom[30];
      char telephone [10];
} annuaire [] = {
        "handel", "0125468956",
        "dowland", "0156897844",
        "byrd", "0689786321",
        "purcell", "0123568974",
"vivaldi", "0569875586",
"wagner", "0123568974",
"mozart", "0213457896",
"beethoven","0563897541",
"weber", "0156897447",
"monteverdi", "0125255699"
};

/* handler du signal SIGURG: fin du serveur */

void handler_urgent()
```

```
{
char car;

recv(nsock, &car, 1, MSG_OOB);
printf("Handler SIGURG, donnees urgentes\n");
exit();
}

main(argc,argv)
char **argv;
{
int namelen, newsock, sock;
int pid, status;

if((sock = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP)) < 0)
{
perror("socket");
exit(1);
}
/* préparation de l'adresse */
/* la zone mémoire pour l'adresse est mise à 0 */
bzero(&sin,sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_port = htons(PORTS); /* passage au format réseau */
sin.sin_addr.s_addr = INADDR_ANY; /* n'importe quelle adresse IP de la
machine locale */
if(bind(sock, &sin, sizeof(sin)) < 0)
{
perror("bind");
exit(2);
}
if(listen(sock,5) < 0)
{
perror("listen");
exit(4);
}

signal(SIGURG, handler_urgent);
pid =getpid();
ioctl(sock, SIOCSPGRP, &pid);

for (;;)
{
if((newsock = accept(sock,&sin,&namelen)) < 0)
{
perror("accept");
exit(5);
}
service(newsock);
}
}
```

9.3 Appel de procédure à distance

Le modèle de communication présenté jusqu'à présent est un modèle de communication de type requête/réponse dans lequel un client émet une requête à un serveur qui traite cette requête et renvoie une réponse (figure 9.10)

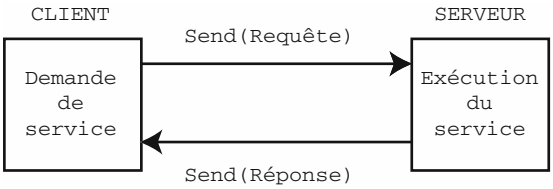

Figure 9.10 – **Modèle requête/réponse**

Le modèle de communication d'appel de procédure à distance (RPC *Remote Procedure Call*) est un mécanisme général de communication entre processus dans les systèmes répartis. Le service rendu par le serveur prend alors la forme d'une procédure que le client peut faire exécuter à distance par le serveur. Il permet à un processus client de réaliser, au cours de son exécution, l'appel d'une procédure `Proc` avec ses paramètres et valeurs de retour éventuels, dont l'exécution a lieu sur une autre machine, avec un résultat globalement identique à celui d'une exécution de la procédure `Proc` en local. L'ensemble des services mettant en place ce mode de communication s'appuie sur l'interface de programmation des sockets et affranchit le programmeur de l'utilisation des primitives de communication de niveau réseau. La figure 9.11 illustre la place du RPC dans la pile de protocoles et le principe de l'appel au travers du réseau.

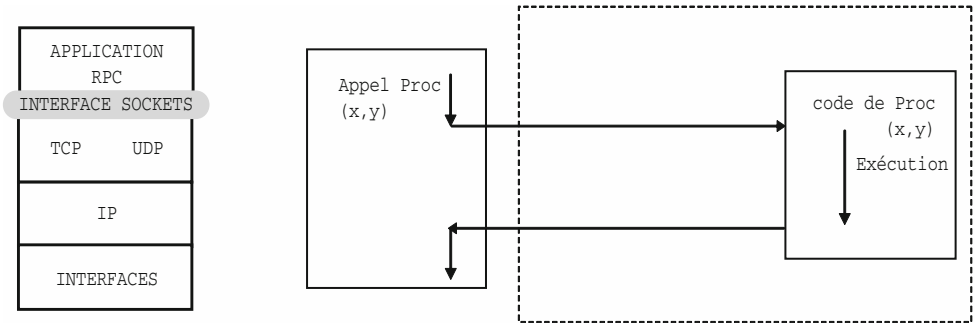

Figure 9.11 – **Principe du RPC**

9.3.1 Mise en œuvre de l'appel de procédure à distance

La mise en œuvre du modèle d'appel de procédure à distance repose sur la notion de souche – souche client et souche serveur – (*Stub*). Une souche est un programme dont le rôle est de rendre transparent l'appel distant, en offrant au client et au serveur une interface d'appel identique à celle d'un appel local.

Ainsi, lors d'un appel de la procédure `Proc(x,y)` réalisé par le client (figure 9.12) :

1. la souche client intercepte l'appel de procédure local et le transforme en requête émise sur le réseau à destination du serveur. Les paramètres de l'appel de procédure *x* et *y* sont « assemblés » dans le message requête (phase d'assemblage ou « *marshalling* ») ;
2. le noyau transmet la requête sur le réseau ;
3. la souche serveur réceptionne la requête, « désassemble » les paramètres et appelle effectivement la procédure correspondante chez le serveur ;
4. la procédure locale s'exécute et transmet les résultats à la souche serveur ;
5. la souche serveur assemble les résultats dans le message réponse ;
6. le noyau transmet le message réponse ;
7. la souche client réceptionne le message réponse et « désassemble » les résultats. Elle les transmet au client en réalisant un retour d'appel de procédure local habituel.

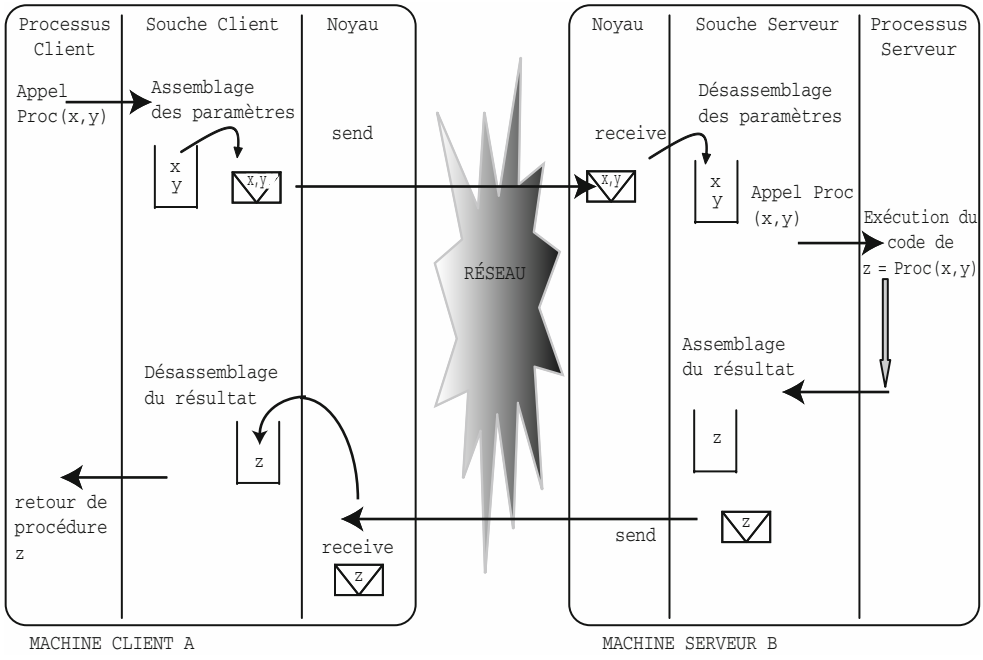

Figure 9.12 – Étapes d'un appel de procédure à distance

9.3.2 Les difficultés

La mise en œuvre de la transparence au niveau de l'appel de procédure à distance qui assimile pour le client l'effet de l'appel distant à celui d'un appel local se heurte à plusieurs difficultés :

- le mode de passage des paramètres ;
- les pannes de client, de serveur et les pertes de messages ;
- la localisation par le client du service appelé.

a) Les passages de paramètres

Les modes de passage classique des paramètres lors d'un appel de procédure sont les modes par valeur et les modes par référence.

Le mode par valeur consiste à copier sur la pile de l'appelant, la valeur du paramètre à transmettre. Cela ne pose pas de difficulté lors d'un appel de procédure à distance où la souche client se contente de copier la valeur du paramètre dans le message requête lors de la phase d'assemblage des paramètres.

Le passage par référence consiste à copier sur la pile de l'appelant l'adresse en mémoire centrale du paramètre à transmettre. Ce mode de transmission de paramètre est donc totalement inutilisable dans le cas d'un appel de procédure à distance puisque l'adresse mémoire transmise perd toute sa signification sur la machine distante (par exemple sur la figure 9.13, l'adresse 1000 sur le serveur correspond à l'adresse d'une instruction).

Deux solutions sont envisageables :

1. interdire le passage par référence mais cela introduit une distinction importante entre l'appel de procédure locale et l'appel de procédure à distance ;
2. simuler le passage par référence en le remplaçant par un passage de type valeur/référence. Dans ce cas, la souche client copie dans le message requête non pas l'adresse du paramètre (adresse 1000) mais sa valeur (50). La souche serveur, à réception de la requête, alloue le paramètre en mémoire centrale côté serveur (adresse 2000) et réalise l'appel de procédure local avec passage par référence. Le résultat est transmis au client selon le même procédé : le contenu de la case mémoire coté serveur (adresse 2000) est copié dans le message réponse. À réception par la souche client, ce résultat est copié dans la case mémoire du paramètre côté client (adresse 1000) et le retour de procédure est exécuté. Ce mécanisme entraîne de nombreuses copies et ne peut s'employer que pour un petit nombre de paramètres.

b) Les pannes

Les pannes à considérer sont d'une part les pertes de messages requête ou réponse, l'arrêt du fonctionnement du processus client ou du processus serveur, par exemple suite à une panne des machines hôtes.

La détection de ces pannes s'effectue classiquement par l'utilisation de délais de garde. Ainsi un client arme un délai de garde lors de l'émission d'une requête vers le serveur. L'expiration du délai de garde, sans réponse du serveur, peut alors être due soit à une perte du message requête, soit à une perte du message réponse, soit à l'arrêt du fonctionnement du serveur.

> On dit qu'un service RPC est idempotent si le résultat de son exécution est identique quel que soit le nombre de fois (1 à *n*) où s'exécute ce service. Par exemple, réaliser la copie des 1 024 premiers octets d'un fichier est un service idempotent. Par contre débiter un compte bancaire d'une somme *X* n'est pas un service idempotent.

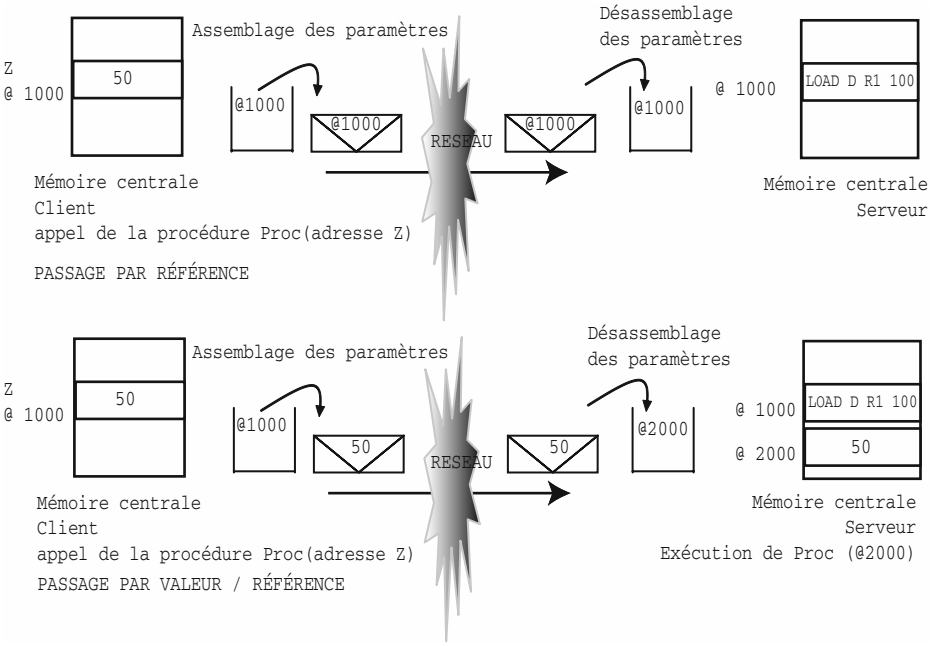

Figure 9.13 – Passage de paramètres par référence et par valeur/référence

Lors de l'expiration du délai de garde, le client, en l'absence de réponse de la part du serveur, réémet sa requête.

- Si l'absence de réponse du serveur est due à la perte du message requête précédent, alors le service est exécuté de nouveau et la panne résolue ;
- Si l'absence de réponse du serveur est due à une perte du message de réponse, alors le service a déjà été exécuté et la réémission de la requête va entraîner une seconde exécution de ce service. Cela peut poser un problème si le service n'est pas idempotent. Pour résoudre ce problème, un numéro de séquence peut être inséré dans le message requête ; le serveur n'exécutant pas de nouveau une requête avec un numéro de séquence déjà traité ;
- Si l'absence de réponse du serveur provient d'une panne de serveur, alors le service a pu être exécuté (panne du serveur après exécution du service et avant l'envoi de la réponse) ou le service n'a pas pu être exécuté (panne du service avant exécution ou pendant l'exécution du service). Comme précédemment, la réémission de la requête de la part du client va entraîner une nouvelle exécution du service et éventuellement poser un problème si le service n'est pas idempotent. Plusieurs sémantiques d'appel de procédure à distance sont alors définies :
  - Au plus une fois : au redémarrage du serveur, le service n'est pas exécuté de nouveau ;
  - Au moins une fois : au redémarrage du serveur, le service est exécuté.

Un dernier cas de panne à considérer est celui des pannes de clients, c'est-à-dire qu'un client envoie une requête et tombe en panne avant de recevoir la réponse du

serveur. Une exécution est donc active chez le serveur dont plus aucun client n'attend le résultat. Une telle exécution, appelée *orphelin*, monopolise des ressources et du temps processeur. Une solution à ce problème peut être d'affecter à chaque exécution de service un quantum Q d'exécution. À l'expiration du délai imparti, le serveur interroge le client pour obtenir un nouveau quantum et s'assure ainsi qu'il est toujours présent.

c) La localisation des services

Lorsqu'un client souhaite effectuer un appel de procédure à distance, il doit connaître la localisation de ce service, par exemple l'adresse IP de la machine hébergeant le service et le numéro de port sur lequel il est en écoute.

Cette localisation peut être réalisée grâce à un serveur de désignation. L'objet de ce serveur est de connaître les localisations et spécifications de chacun des services RPC. Ce sont les souches client et serveur qui dialoguent avec le serveur de désignation.

Ainsi, comme illustré par la figure 9.14 :

- Lorsqu'un service RPC est créé, la souche serveur enregistre ce service auprès du serveur de désignation, en indiquant le nom du service, sa spécification et son adresse réseau (couple adresse IP, numéro de port) ;
- Lorsqu'un client souhaite envoyer une requête à un service RPC, la souche client interroge le serveur de localisation pour connaître sa localisation et pouvoir émettre le message de requête.

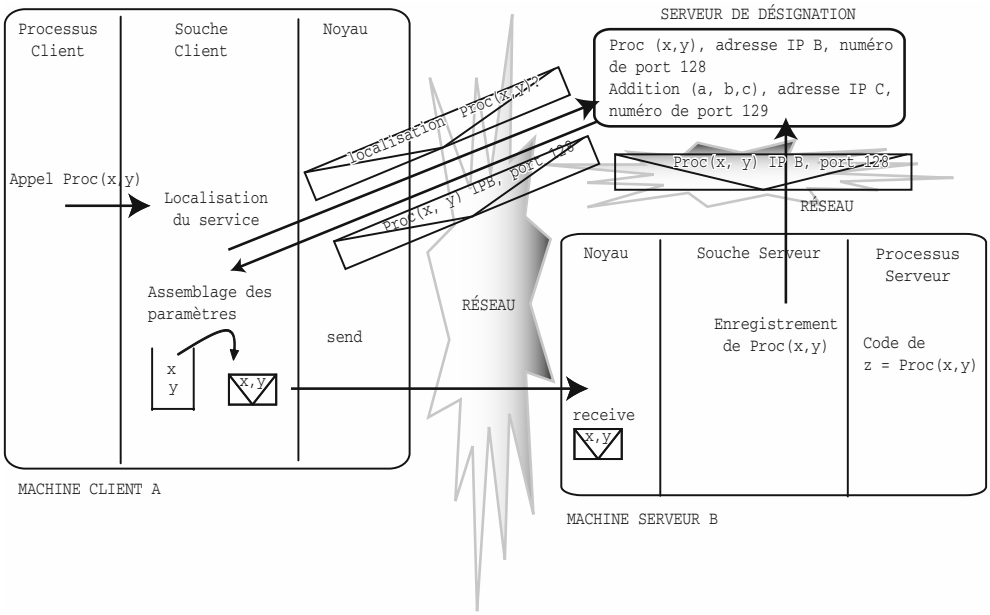

Figure 9.14 – Serveur de désignation pour la localisation des services RPC

Exercices

9.1 Qu'avez-vous retenu ?

Pour chacune des questions suivantes, choisissez la ou les réponses qui vous semblent exactes.

Question 1 – Une adresse de niveau application :

- ☐ a. est un couple composé d'une adresse IP et d'un numéro de port.
- ☐ b. est composée par une adresse IP.
- ☐ c. est matérialisée par une socket.

Question 2 – Une machine multidomiciliée :

- ☐ a. est une passerelle entre plusieurs réseaux.
- ☐ b. est une machine qui possède plusieurs adresses IP distinctes.
- ☐ c. est une machine dupliquée sur plusieurs sites géographiques distincts.

Question 3 – Une socket :

- ☐ a. est une interface de programmation placée au-dessus des couches transport de la pile de protocoles.
- ☐ b. est un outil de communication compatible avec le VFS.
- ☐ c. matérialise une adresse application.

Question 4 – Le protocole de communication TCP :

- ☐ a. est un protocole de niveau réseau qui offre une communication fiable.
- ☐ b. est un protocole de niveau transport orienté connexion qui offre une communication fiable.

Question 5 – Un serveur parallèle :

- ☐ a. est un serveur qui conserve des informations sur les requêtes en cours de traitement.
- ☐ b. est un serveur qui traite une seule requête à la fois.
- ☐ c. est un serveur qui associe un fil d'exécution nouveau à chaque nouvelle requête à traiter.

Question 6 – Le protocole de communication UDP :

- ☐ a. est un protocole de niveau réseau qui offre une communication fiable.
- ☐ b. est un protocole de niveau transport orienté connexion qui offre une communication fiable.
- ☐ c. est un protocole de niveau transport orienté datagramme qui offre une communication non fiable.

Question 7 – La primitive « bind » :

- ☐ a. permet de créer une socket.
- ☐ b. permet d'attacher une adresse d'application à une socket existante.
- ☐ c. permet d'envoyer des datagrammes.

Question 8 – La primitive « socket » :

- ☐ a. permet de créer une socket.
- ☐ b. permet d'attacher une adresse d'application à une socket existante.
- ☐ c. permet d'envoyer des datagrammes.

Question 9 – Une « souche » :

- ☐ a. est un programme dont le rôle est de rendre transparent l'appel distant de procédure.
- ☐ b. est un point d'accès au réseau associé à une adresse d'application.

9.2 Communication répartie

On considère une architecture répartie client-serveur. Les clients lisent au clavier des caractères et ils les envoient ensuite à un serveur d'impression de type parallèle. On vous donne ci-après les pseudo-codes des processus :

Client	Serveur (père)	Serveur (fils)
Debut Répéter Lire_clavier(caractère1) ; Lire_clavier(caractère2) ; **Etablir_Connexion_Serveur ;** Envoyer_requete(caractère1, caractère2) ; Déconnexion_Serveur ; Fin répéter Fin	Début Répéter **Etablir_Connexion_Client ;** Créer_Fils ; Fin_Répéter ; Fin	Début Recevoir_requête(caractère1, caractère2) ; Imprimer(caractère1) ; Imprimer(caractère2) ; Fin

1) Tous les fils du serveur parallèle se partagent une même imprimante. Pour une même requête client, les deux caractères la composant doivent être forcément imprimés l'un à la suite de l'autre.
   Exemple : soient deux requêtes client1 (caractères A et B) et client2 (caractères C et D), l'impression doit être ABCD ou CDAB mais pas ACBD.
   Modifiez le code côté serveur pour que cela fonctionne, en utilisant l'outil sémaphore. Quel schéma de synchronisation mettez-vous en œuvre ? Vous donnez la solution en pseudo-code.
2) En utilisant les primitives de communication Linux adéquates, traduisez les primitives de communication indiquées en gras dans les pseudo-codes des clients et des serveurs. La communication est en mode TCP. Le serveur a pour nom symbolique serveur_impression.cnam.fr. Il est en écoute sur le port 1863.

9.3 Le jeu des 7 erreurs

Il y a au moins 7 erreurs dans la programmation socket de cette application client-serveur en mode connecté (ci-après) : trouvez-les et corrigez-les.

```
/* programme serveur.c */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
main()
{
int newsock, sock;
int pid, lg;
struct sockaddr_in sin;
char message[20];

sock = soc-
ket(AF_INET,SOCK_STREAM,IPPROTO_TCP)
;

/* Préparation de l'adresse du
serveur: famille internet, adresse IP
(INADDR_ANY désigne l'adresse IP de
la machine locale, ici
dirac.cnam.fr), n°de port du serveur
*/
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = ntohs(10256);

bind(sock, &sin, sizeof(sin));

for (;;) {
lg = sizeof(sin);
newsock = accept(sock,&sin,&lg));
pid = fork();
if (pid == -1)
   printf ("erreur création de
processus");
else
   if (pid == 0)
   {
    read(sock, message, 15);
    printf ("%s", message);
    close(sock);
    exit();
   }
}
}
```

```
/* programme client.c */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

struct sockaddr_in nom = {AF_INET};

main()
{
struct hostent *h;
struct sockaddr_in sin;
int sock;

if (sock = soc-
ket(AF_INET,SOCK_DGRAM,IPPROTO_TCP);

bind(sock, &nom, sizeof(nom));

/* on récupère les informations
concernant la machine serveur:
notamment l'adresse IP (champ h-
>h_addr de taille ,h->h_length
octets */
h=gethostbyname("dirac.cnam.fr");

/* Préparation de l'adresse du
serveur sin famille internet, copie
de l'adresse IP du serveur, n°de
port du serveur */
sin.sin_family = AF_INET;
bcopy(h->h_addr,&sin.sin_addr,h-
>h_length);
sin.sin_port = htons(10265);

connect(sock,&sin,sizeof(sin));

sendto(sock, "bonjour serveur", 15);

close (sock);
}
```

Figure 9.15

9.4 Le super-démon `Inetd`

Le super-démon `Inetd` surveille l'ensemble des ports UDP et TCP affectés aux applications réseaux standards définies dans le fichier `/etc/inetd.conf`. Lorsqu'une requête provient à l'un de ces ports, le super-démon `Inetd` crée un processus pour servir la requête selon le protocole de communication adéquat. Ce processus meurt une fois le service rempli.

Donnez un pseudo-code correspondant aux grandes étapes du fonctionnement du super-démon `Inetd`.

9.5 Serveur de scrutation

On considère une application composée d'un processus serveur placé sur une machine physique différente des processus clients. Les services rendus par le serveur sont :

- d'une part, un service qui réalise l'addition de deux nombres entiers envoyés par le client et renvoie le résultat ;
- d'autre part, un service qui réalise la multiplication de trois nombres flottants envoyés par le client et renvoie le résultat.

Les opérations d'addition sont effectuées en simultanée par un service de type parallèle. Par contre, les multiplications s'effectuent les unes après les autres par un service de type itératif. Le service en mode parallèle fonctionne en mode TCP tandis que le service en mode itératif fonctionne sous UDP. Une socket d'écoute est dédiée à chacun de ces services qui sont donc appelables de manière indépendante par les clients.

Réalisez la programmation de cette application.

Des idées pour vous exercer à la programmation

Programme 1 – Reprenez le serveur parallèle donné en exemple dans le cours et réalisez-le à l'aide de threads.

Programme 2 – Reprenez l'énoncé de l'exercice 7 du chapitre 7 en remplaçant la file de messages par des sockets.

Solutions

9.1 Qu'avez-vous retenu ?

Question 1 – Une adresse de niveau application :

- ☐ a. est un couple composé d'une adresse IP et d'un numéro de port.
- ☐ c. est matérialisée par une socket.

Question 2 – Une machine multidomiciliée :

- ☐ a. est une passerelle entre plusieurs réseaux.
- ☐ b. est une machine qui possède plusieurs adresses IP distinctes.

Question 3 – Une socket :

- ☐ a. est une interface de programmation placée au-dessus des couches transport de la pile de protocoles.
- ☐ b. est un outil de communication compatible avec le VFS.
- ☐ c. matérialise une adresse application.

Question 4 – Le protocole de communication TCP :

- ☐ b. est un protocole de niveau transport orienté connexion qui offre une communication fiable.

Question 5 – Un serveur parallèle :

- ☐ c. est un serveur qui associe un fil d'exécution nouveau à chaque nouvelle requête à traiter.

Question 6 – Le protocole de communication UDP :

- ☐ c. est un protocole de niveau transport orienté datagramme qui offre une communication non fiable.

Question 7 – La primitive « bind » :

- ☐ b. permet d'attacher une adresse d'application à une socket existante.

Question 8 – La primitive « socket » :

- ☐ a. permet de créer une socket.

Question 9 – Une « souche » :

- ☐ a. est un programme dont le rôle est de rendre transparent l'appel distant de procédure.

9.2 Communication répartie

1) Le schéma qui est mis en œuvre est de type exclusion mutuelle. Un seul processus à la fois effectue une impression.

Soit `MUTEX` un sémaphore, `INIT(MUTEX, 1)`. Le code du serveur fils devient :

```
Début
Recevoir_requête(caractère1, caractère2) ;
P(MUTEX) ;
Imprimer(caractère1) ;
```

```
Imprimer(caractère2) ;
V(MUTEX) ;
Fin
```

2) Client : `Etablir_Connexion_Serveur ;`

```
#define PORTS 1863
sock = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP) ;
bzero(&sin,sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_port = 0 ;
sin.sin_addr.s_addr = INADDR_ANY;
bind(sock, &sin, sizeof(sin));
h=gethostbyname("impression.cnam.fr");
bzero(&sin,sizeof(sin));
sin.sin_family = AF_INET;
bcopy(h->h_addr,&sin.sin_addr,h->h_length);
sin.sin_port = htons(PORTS);
connect(sock,&sin,sizeof(sin)) ;
```

```
Serveur : Etablir_Connexion_Client ;
```

```
#define PORTS 1863
sock = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
bzero(&sin,sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_port = htons(PORTS);
sin.sin_addr.s_addr = INADDR_ANY;
bind(sock, &sin, sizeof(sin)) ;
listen(sock,5);
namelen = sizeof(sin);
newsock = accept(sock,&sin,&namelen);
```

9.3 Le jeu des 7 erreurs

Les 7 erreurs suivantes peuvent être relevées :

- le client crée une socket de type `SOCK_DGRAM` au lieu d'une socket de type `SOCK_STREAM` ;
- le numéro de port utilisé par le processus client lors de la construction de l'adresse est erroné (10 265 au lieu de 10 256) ;
- le processus client emploie la primitive `sendto()` à la place de la primitive `read()` ;
- le processus fils côté serveur effectue le service en utilisant la socket d'écoute à la place de la socket de service ;
- le processus père côté serveur ne ferme pas la socket de service ;
- le processus père côté serveur n'attend pas la mort de ses fils et il n'a pas mis en place de mécanisme pour détruire ses processus fils zombies (`signal(SIGCHLD, SIG_IGN)`) ;

- le processus serveur n'utilise pas la bonne primitive de conversion lorsqu'il remplit le champ numéro de port de son adresse locale (primitive `ntohs()` à la place de `htons()`).

9.4 Le super-démon `Inetd`

L'organigramme de la figure 9.16 donne la solution.

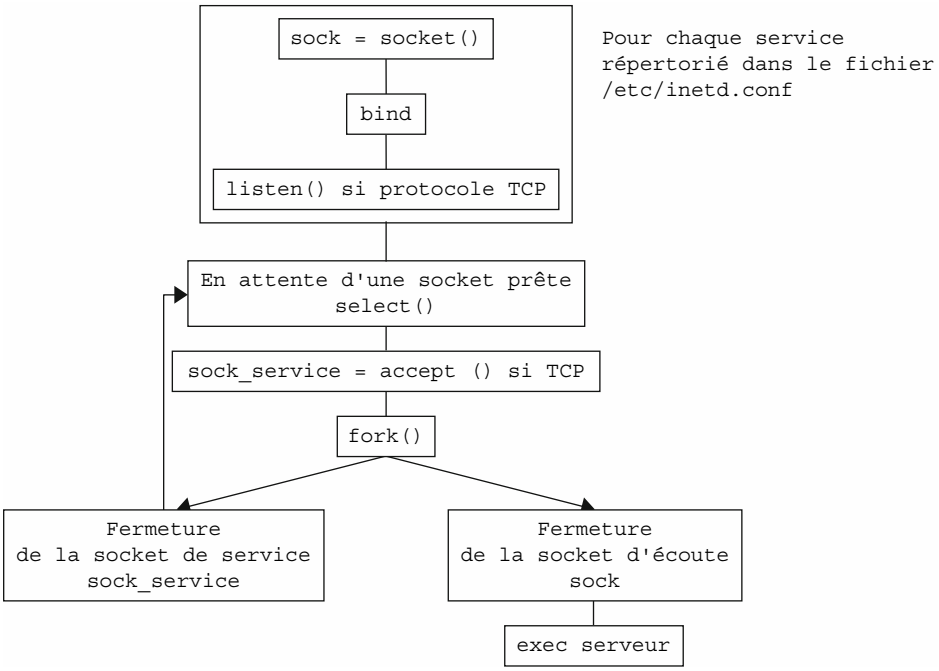

Figure 9.16 – Traitements effectués par le super-démon `Inetd`

9.5 Serveur de scrutation

Le serveur applicatif doit être en attente de requêtes sur deux sockets à la fois, à l'aide d'une part de la primitive `accept()`, d'autre part de la primitive `recvfrom()`, qui sont toutes deux des opérations bloquantes. Ainsi, le serveur qui ne connaît pas l'ordre d'arrivée des requêtes clients peut se mettre en attente de requêtes d'addition, se bloquer sur la primitive `accept()` et se trouver ainsi dans l'incapacité de répondre à des requêtes de multiplication parvenant à l'autre socket. Pour pouvoir gérer les attentes sur les deux ports et éviter ce type de blocage, il faut utiliser la primitive `select()` que nous avons déjà rencontrée au chapitre 3.

Nous donnons le code du serveur correspondant. Dans ce code, le serveur définit à l'aide des macros `FD_ZERO` et `FD_SET`, l'ensemble `read_fd` des descripteurs sur lequel il doit se mettre en attente. Une temporisation de 10 ms est définie.

```
/*****************************************************************/
/*                      Serveur de scrutation                    */
/*****************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/time.h>
#include <signal.h>
#define PORT1 6259
#define PORT2 6260
#define TRUE  1

 struct operplus {
            int nb1;
            int nb2;
       };
 struct opermult {
              float nb1;
              float nb2;
              float nb3;
      };

main ()
{
int lg, lg_adresse, port, sock1, sock2, nsock1, nsock2,d, retour, pid,
resplus;
struct sockaddr_in adr, nom, adresse_client;
/* adresse de la socket distante */
struct operplus operandeplus;
struct opermult operandemult;
float resmult;
fd_set readf;       /* variable pour select */
struct timeval to;  /* timeout pour select */

if ((sock1 = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    perror ("Création impossible de socket");
/* Preparation de l'adresse */
port = htons(PORT1);
bzero ((char *)&nom, sizeof(struct sockaddr_in));
nom.sin_port = port;
nom.sin_addr.s_addr = INADDR_ANY;
nom.sin_family = AF_INET;
if (bind(sock1, (struct sockaddr *)&nom, sizeof(struct sockaddr)) == -1)
    perror ("Nommage impossible de socket");*/

port = htons(PORT2);
if ((sock2 = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
    perror ("Creation impossible de socket");
```

```
printf ("%d", sock2);

/* Préparation de l'adresse */
bzero ((char *)&nom, sizeof(struct sockaddr_in));
nom.sin_port = port;
nom.sin_addr.s_addr = INADDR_ANY;
nom.sin_family = AF_INET;
if (bind(sock2, (struct sockaddr *)&nom, sizeof(struct sockaddr_in)) == -
1)
    perror ("Nommage impossible de socket");

/* Création de la file des connexions pendantes */
listen(sock1,5);

/* Boucle d'acceptation d'une connexion */
/signal(SIGCHLD, SIG_IGN);*

while (TRUE) { /* Attente de question sur les sockets */
FD_ZERO(&readf);
FD_SET(sock1, &readf);
FD_SET(sock2, &readf);
to.tv_sec = 10;

retour = select(sock2 + 1, &readf, NULL, NULL, &to);

if (retour == 0) {
   fprintf (stderr, "retour sur time out\n");
   return;
   }

if (FD_ISSET(sock1, &readf)) {
   lg = sizeof(adr);
   nsock1 = accept (sock1,(struct sockaddr *)&adr,&lg);
   if (nsock1 == -1)
        perror ("Pb connexion serveur");
   pid = fork();
    if (pid == 0)
   {
     close(sock1);
     read(nsock1,  &operandeplus, sizeof(struct operplus));
     resplus = operandeplus.nb1 + operandeplus.nb2;
     write (nsock1, &resplus, sizeof(int));
     close(nsock1);
   }
   else
     close (nsock1);
   }
```

```
if (FD_ISSET(sock2, &readf)) {
   lg = sizeof(adresse_client);
   recvfrom(sock2,&operandemult, sizeof(struct opermult), 0,(struct
sockaddr *)&adresse_client,&lg_adresse);
   resmult = operandemult.nb1 * operandemult.nb2 * operandemult.nb3;
   sendto (sock2, &resmult, sizeof(float), 0, (struct sockaddr
*)&adresse_client, lg_adresse);
}
}
}

/*****************************************************************/
/*               Client1 du serveur de scrutation                */
/*                et client du serveur parallèle                 */
/*****************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#define PORT 6259
#define TRUE  1

struct operplus {
  int nb1;
  int nb2;
  };

main ()
{
int  sock, nb1, nb2, resplus, port;
struct sockaddr_in adr, nom;
struct hostent *hp;
struct operplus operandeplus;

printf ("Donnez moi deux nombres à additionner:");
scanf ("%d", &operandeplus.nb1);
scanf ("%d", &operandeplus.nb2);

/* Création de la socket */
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    perror ("Creation impossible de socket");

/* Préparation de l'adresse */
bzero ((char *)&nom, sizeof(struct sockaddr_in));
port = htons(PORT);
nom.sin_port = 0;
nom.sin_addr.s_addr = INADDR_ANY;
nom.sin_family = AF_INET;
```

```
if (bind(sock, (struct sockaddr *)&nom, sizeof(struct sockaddr_in)) == -
1)
    perror ("Nommage impossible de socket");

/* Préparation de l'adresse de la socket destinataire */
if ((hp = gethostbyname("vesuve.cnam.fr")) == NULL)  {
    perror ("Nom de machine irrecuperable");
    return;
}
bzero ((char *)&adr,sizeof(adr));
adr.sin_port = htons(PORT);
adr.sin_family = AF_INET;
bcopy (hp->h_addr, &adr.sin_addr, hp->h_length);
connect (sock, (struct sockaddr *)&adr, sizeof(adr));
write (sock, &operandeplus, sizeof(struct operplus));
read(sock, &resplus, sizeof(int));
printf ("le résultat reçu est: %d", resplus);
close(sock);
}

/*****************************************************************/
/*               Client2 du serveur de scrutation                */
/*                et client du serveur itératif                  */
/*****************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#define PORT 6259
#define TRUE  1

struct opermult {
    float nb1;
    float nb2;
    float nb3;
};

main ()
{
int  sock, nb1, nb2, resmult, port, lg;
struct sockaddr_in adr, nom;
struct hostent *hp;
struct opermult operandemult;

printf ("Donnez moi trois nombres à multiplier:");
scanf ("%f", &operandemult.nb1);
scanf ("%f", &operandemult.nb2);
scanf ("%f", &operandemult.nb3);
```

```
/* Création de la socket */
if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
    perror ("Creation impossible de socket");

/* Préparation de l'adresse */
bzero ((char *)&nom, sizeof(struct sockaddr_in));
port = htons(PORT);
nom.sin_port = 0;
nom.sin_addr.s_addr = INADDR_ANY;
nom.sin_family = AF_INET;
if (bind(sock, (struct sockaddr *)&nom, sizeof(struct sockaddr_in)) == -
1)
    perror ("Nommage impossible de socket");
/* Préparation de l'adresse de la socket destinataire */
if ((hp = gethostbyname("vesuve.cnam.fr")) == NULL)  {
    perror ("Nom de machine irrecuperable");
    return; }
bzero ((char *)&adr,sizeof(adr));
adr.sin_port = htons(PORT);
adr.sin_family = AF_INET;
bcopy (hp->h_addr, &adr.sin_addr, hp->h_length);

sendto (sock, &operandemult, sizeof(struct opermult), 0, (struct sockaddr
*)&adr, sizeof(struct sockaddr_in));
lg =sizeof(struct sockaddr_in);
recvfrom(sock, &resmult, sizeof(float), 0,(struct sockaddr *)&adr, &lg);
printf ("le résultat reçu est: %f", resmult);
close(sock);
}
```

SYSTÈMES LINUX AVANCÉS

10

Plan

PLAN

OBJECTIFS

- Ce chapitre est consacré à la présentation des systèmes Linux temps réel puis des systèmes Linux multiprocesseurs. Nous commençons par définir les particularités d'un tel système.
- Nous présentons la notion d'exécutif temps réel et les services qui lui sont attachés. Plus spécifiquement, nous détaillons la fonction de l'exécutif responsable de l'ordonnancement de tâches. Nous finissons par la présentation d'une version temps réel de Linux, le système RTLinux.

10.1 SYSTÈMES LINUX TEMPS RÉEL

Nous présentons la notion d'exécutif temps réel et les services qui lui sont attachés. Plus spécifiquement, nous détaillons la fonction de l'exécutif responsable de l'ordonnancement de tâches. Nous finissons par la présentation d'une version temps réel de Linux, le système RTLinux.

10.1.1 Applications temps réel

a) Définition d'une application temps réel

Une application temps réel est une application pour laquelle le temps est la principale contrainte à respecter. Une application qui fournit un résultat juste mais hors délai ne peut pas être validée. Il ne s'agit pas par contre de rendre un résultat le plus vite possible, mais simplement à temps. L'échelle du temps relative au type de l'application varie d'une application à l'autre : elle est de l'ordre de la microseconde dans les radars, et peut être de l'ordre de l'heure pour une réaction chimique.

Ces applications temps réel sont aussi qualifiées d'applications réactives car elles sont le plus souvent interfacées avec un procédé qu'elles pilotent.

Ainsi, une application temps réel peut être vue comme en deux parties qui interagissent entre elles (figure 10.1.) :

- d'un côté, un procédé, en général industriel, muni de capteurs, envoie des mesures ou déclenche des événements avec une occurrence périodique ou aléatoire. Un tel procédé peut être un moteur, un train, un réacteur, un robot, etc. ;
- de l'autre côté, un *système informatique souvent qualifié d'exécutif temps réel,* connecté au procédé dont il doit de manière dynamique commander et contrôler le comportement. Pour être temps réel, il doit réagir aux événements du procédé avant un délai fixé ou à une date donnée. C'est le système informatique qui doit permettre à l'application de respecter ses contraintes temporelles.

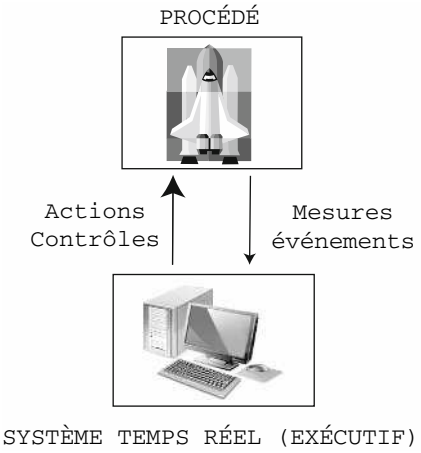

Figure 10.1 – Applications temps réel interfacée au procédé

b) Contraintes temporelles

Une faute temporelle se produit lorsqu'une contrainte temporelle liée à l'application temps réel n'est pas respectée.

Deux types de contraintes temporelles peuvent être définis :

- les **contraintes temporelles strictes** : les fautes temporelles sont intolérables pour la validité du système. L'occurrence d'une faute temporelle entraîne des dommages sur le procédé et son environnement ;
- les **contraintes temporelles relatives** : quelques fautes temporelles peuvent être supportées sans remettre en cause la validité du système. Il s'ensuit généralement une simple baisse de performances.

c) Un exemple d'application temps réel

Le système de perception d'un robot mobile permet au robot mobile d'appréhender les modifications de son environnement et d'adapter son comportement en conséquence. Le système de perception doit gérer des traitements événementiels liés à la découverte de l'environnement inconnu et des traitements périodiques liés à l'analyse de l'environnement. Il doit gérer un grand nombre de capteurs tels qu'un capteur

ultrasonore, un capteur radar, un capteur laser, une caméra infrarouge. Les informations qui en sont extraites doivent être filtrées en temps contraint de manière à prélever des informations telles que la présence d'obstacles, la position du robot, la modélisation de l'environnement, etc., qui vont influer sur la génération de la trajectoire du robot. Un retard dans l'analyse de ces informations, dû par exemple à l'arrivée asynchrone de nombreux événements, peut amener le robot à percuter un obstacle et causer un dommage important à celui-ci.

10.1.2 Exécutifs temps réel

L'application temps réel est le plus souvent conçue comme une application multiprogrammée, c'est-à-dire composée de plusieurs processus qui interagissent entre eux. Le rôle de l'exécutif temps réel est donc notamment de gérer l'exécution de ces processus.

a) Tâches temps réel

Dans le cadre des systèmes temps réel, un processus, image dynamique de l'exécution d'un programme tel que nous l'avons étudié dans cet ouvrage, est plutôt appelé tâche temps réel.

Les tâches composant une application temps réel peuvent être, en fonction de leurs caractéristiques, répertoriées sous différents types. Deux critères interviennent principalement :

- la façon dont la tâche survient dans la vie du système temps réel. On distingue les tâches périodiques, qui s'exécutent à intervalles réguliers (typiquement des mesures ou commandes de systèmes asservis échantillonnés) et les tâches apériodiques qui surviennent aléatoirement (typiquement une alarme) ;
- le degré de flexibilité de la tâche vis-à-vis de l'application temps réel. On distingue les tâches à contraintes strictes (tâches critiques), qui ne doivent jamais violer la contrainte temporelle qui leur est associée sous peine de mettre le procédé en péril et les tâches à contraintes relatives (tâches essentielles), qui elles, peuvent de temps à autre violer cette contrainte sans que le système en souffre.

L'application temps réel peut donc être composée de quatre types de tâches différentes :

- des tâches périodiques à contraintes strictes ;
- des tâches apériodiques à contraintes strictes ;
- des tâches périodiques à contraintes relatives ;
- des tâches apériodiques à contraintes relatives.

Modélisation des tâches périodiques

Couramment, une tâche périodique *Tp* est représentée avec les paramètres suivants, illustrés sur la figure 10.2 :

- s_0, est la date de première soumission de la tâche dans le système ;
- *C* est le temps d'exécution de la tâche ;

- *R* est le délai maximum dont dispose la tâche pour s'exécuter à partir du moment où elle a été soumise. *R* est qualifié de délai critique et représente donc la contrainte temporelle associée à l'exécution. Il définit l'échéance de l'exécution courante, date de fin d'exécution au plus tard encore appelée date d'échéance *d*, avec $d = s + R$;
- *P* est la période de la tâche, c'est-à-dire l'intervalle séparant deux soumissions de l'exécution. Ainsi la soumission *k* ou occurrence *k* de la tâche *Tp* survient à la date $s_k = s_{k-1} + P$.

Dans le cas particulier où $R = P$, la tâche temps réel est qualifiée de tâche à échéance sur requête. La contrainte temporelle associée à la tâche se résume alors à ce qu'une occurrence d'exécution soit achevée au plus tard au réveil de la suivante.

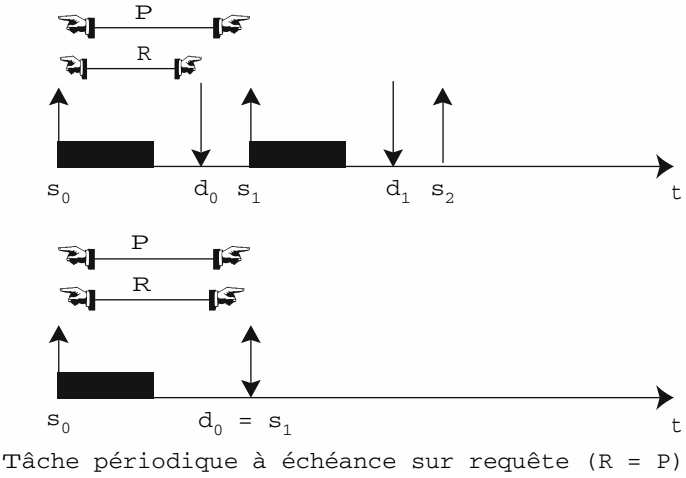

Figure 10.2 – **Paramètres des tâches périodiques**

Modélisation des tâches apériodiques

Couramment, une tâche apériodique *Tap* est représentée avec les paramètres suivants, illustrés sur la figure 10.3 :

r, est la date de soumission de la tâche dans le système ;

C est le temps d'exécution de la tâche ;

R est le délai maximum dont dispose la tâche pour s'exécuter à partir du moment où elle a été soumise. *R* est qualifié de délai critique et représente donc la contrainte temporelle associée à l'exécution. Il définit l'échéance de l'exécution courante, date de fin d'exécution au plus tard *d*, avec $d = r + R$.

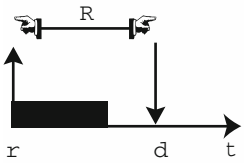

Figure 10.3 – **Paramètres des tâches apériodiques**

b) Services et propriétés de l'exécutif temps réel

Un exécutif temps réel est un système d'exploitation dédié au support de l'exécution d'une application temps réel. Il présente :

- un service de gestion des tâches qui assure l'ordonnancement sur le processeur et également toutes les opérations de création, activation, suspension, terminaison des tâches. Ce service doit notamment offrir les mécanismes permettant la définition de tâches de nature périodique ou apériodique ainsi que des contraintes temporelles qui leur sont associées. Il doit aussi offrir un service d'ordonnancement capable de bâtir une planification d'exécution des tâches valide et certifiable ;
- un service de gestion des interruptions, de gestion des exceptions et de reprise d'erreurs ;
- un service de gestion des entrées-sorties, prenant en charge les différents types d'interfaces que peut offrir le procédé temps réel ;
- un service d'outils de communication entre les tâches et de synchronisation pour l'accès aux ressources. Les outils de communication offerts sont classiquement des modes de communication par boîtes aux lettres et plus généralement selon un mode client-serveur. Les outils de synchronisation sont notamment les sémaphores ;
- un service de gestion du temps dont la précision soit compatible avec les exigences temporelles des applications temps réel. Ce service doit offrir les outils permettant de définir des intervalles de temps correspondant aux périodes des tâches ou de dater des événements. Il doit offrir une haute résolution temporelle et la gestion de temporisateurs.

Notamment l'exécutif temps réel doit permettre et garantir l'exécution des tâches de l'application temps réel dans le respect des contraintes temporelles qui leur sont associées. L'exécutif temps réel doit alors présenter la propriété d'être déterministe, c'est-à-dire que certaines opérations du système telles que la durée d'une commutation de contexte, la durée de prise en compte d'une interruption, etc. doivent pouvoir être bornées. Ainsi, des temps d'exécution maximum C peuvent être associés aux tâches et des tests d'acceptabilité permettent hors ligne, c'est-à-dire avant le démarrage de l'application, de déterminer si l'application, ordonnancée selon un certain algorithme d'ordonnancement, respecte ses contraintes temporelles.

10.1.3 Le service d'ordonnancement temps réel

Nous allons successivement présenter les algorithmes d'ordonnancement pour le temps réel les plus courants pour des tâches périodiques. Pour chacun de ces algorithmes, un test d'acceptabilité est défini.

a) Propriétés

Un ensemble de tâches est dit *acceptable* pour un algorithme d'ordonnancement si les tâches de l'ensemble s'ordonnancent avec cet algorithme dans le respect de leur contrainte de temps. Un *test d'acceptabilité* sur les paramètres temporels des tâches

permet pour chaque algorithme d'ordonnancement de déterminer si un ensemble de tâches est acceptable avec cet algorithme.

Soit un ensemble de tâches périodiques Tp_i ($s_{i0} = 0$, C_i, R_i, P_i). La séquence d'ordonnancement de ces tâches est cyclique et se répète sur l'intervalle [0 , PPCM(P_i)]. Cet intervalle est appelé intervalle d'étude *I*.

b) Algorithmes d'ordonnancement

Rate Monotonic

Une priorité fixe est associée à chaque tâche périodique. Plus la période de la tâche est petite, plus sa priorité est forte. L'ordonnancement est préemptif et à tout moment, la tâche de plus forte priorité (donc de plus petite période) est exécutée.

Dans le cas de tâches à échéance sur requête, le test d'acceptabilité pour un ensemble de *n* tâches est (condition suffisante) :

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{1/n} - 1)$$

Exemple

Soit l'ensemble de tâches périodiques à échéance sur requête composé des trois tâches *Tp1*($s_0 = 0$, $C = 3$, $P = 20$), *Tp2*($s_0 = 0$, $C = 2$, $P = 5$) et *Tp3*($s_0 = 0$, $C = 2$, $P = 10$).

Le test d'acceptabilité donne $(3/20 + 2/5 + 2/10) \leqslant 0{,}779$ ce qui est vrai. La figure 10.4 donne le chronogramme d'exécution de ces trois tâches sur l'intervalle d'étude correspondant soit [0 , 20]. Les trois tâches par ordre de priorité décroissante sont *Tp2*, *Tp3* et *Tp1*. Sur ce chronogramme, une flèche « pointe en l'air » symbolise un réveil. Une flèche « pointe en bas » symbolise une date d'échéance. Ainsi la tâche *Tp1* a une périodicité de 20 ; elle se réveille donc au temps $t = 0$ et $t = 20$. La tâche *Tp2* a une périodicité de 5 ; elle se réveille donc aux instants $t = 0$, 5, 10, 15 et 20. La tâche *Tp3* a une périodicité de 10 ; elle se réveille donc aux instants $t = 0$, 10, et 20. Comme chacune de ces tâches est une tâche à échéance sur requête, les dates de réveil correspondent également aux dates d'échéance.

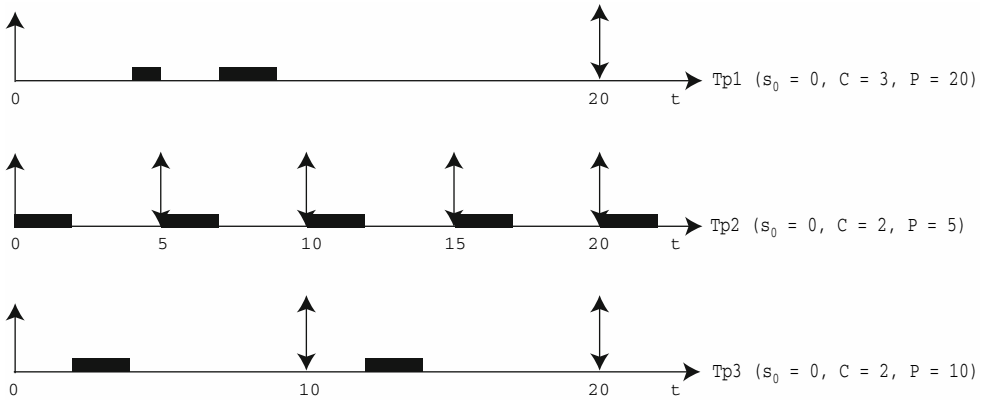

Figure 10.4 – Algorithme Rate Monotonic

Inverse Deadline

Une priorité fixe est associée à chaque tâche. Plus de délai critique de la tâche est petit, plus sa priorité est forte. L'ordonnancement est préemptif et à tout moment, la tâche de plus forte priorité (donc de plus petit délai critique) est exécutée.

Le test d'acceptabilité pour un ensemble de n tâches est (condition suffisante) :

$$\sum_{i=1}^{n} \frac{C_i}{R_i} \leq n(2^{1/n} - 1)$$

Exemple

Soit l'ensemble de tâches périodiques composé des trois tâches *Tp1*($s_0 = 0$, $C = 3$, $R = 15$, $P = 20$), *Tp2*($s_0 = 0$, $C = 1$, $R = 4$, $P = 5$) et *Tp3*($s_0 = 0$, $C = 2$, $R = 9$, $P = 10$).

Le test d'acceptabilité donne $(3/15 + 1/4 + 2/9) = 0{,}779$, ce qui est vrai. La figure 10.5 donne le chronogramme d'exécution de ces trois tâches sur l'intervalle d'étude correspondant soit [0 , 20]. Les trois tâches par ordre de priorité décroissante sont *Tp2*, *Tp3* et *Tp1*. Comme précédemment, sur ce chronogramme, une flèche « pointe en l'air » symbolise un réveil et une flèche « pointe en bas » symbolise une date d'échéance.

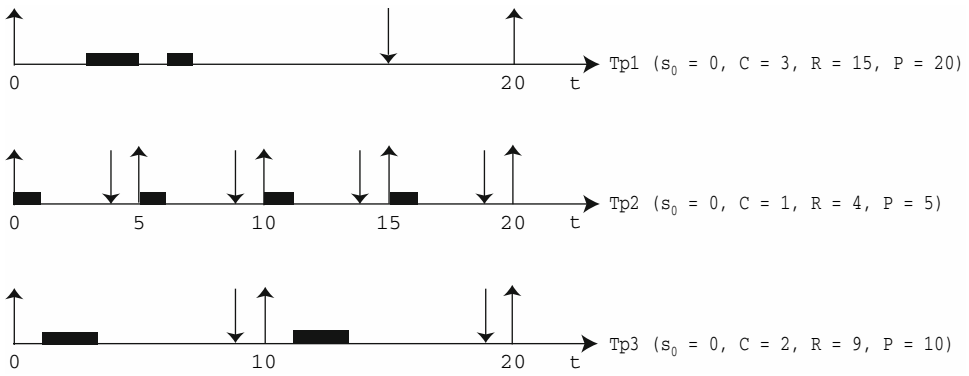

Figure 10.5 – Algorithme Inverse Deadline

Earliest Deadline

Une priorité dynamique est associée à chaque tâche. À l'instant t, la tâche la plus prioritaire est celle dont l'échéance est la plus proche. L'ordonnancement est préemptif et à tout moment, la tâche la plus urgente est exécutée.

Le test d'acceptabilité pour un ensemble de n tâches est (condition suffisante) :

$$\sum_{i=1}^{n} \frac{C_i}{R_i} \leq 1$$

Exemple

Soit l'ensemble de tâches périodiques composé des trois tâches *Tp1*($s_0 = 0$, $C = 3$, $R = 8$, $P = 20$), *Tp2*($s_0 = 0$, $C = 1$, $R = 4$, $P = 5$) et *Tp3*($s_0 = 0$, $C = 2$, $R = 7$, $P = 10$).

Le test d'acceptabilité donne (3/8 + 1/4 + 2/7) ⩽ 1, ce qui est vrai. La figure 10.6 donne le chronogramme d'exécution de ces trois tâches sur l'intervalle d'étude correspondant soit [0 , 20].

À l'instant $t = 0$, les tâches réveillées sont Http, *Tp2* et *Tp3*. La tâche de plus proche échéance d est la tâche *Tp2* ($d = 4$). Elle est élue et s'exécute. Lorsqu'elle a achevé son exécution la tâche *Tp3* est maintenant celle de plus proche échéance et elle est élue à son tour. Puis la tâche Tp1 commence son exécution. À l'instant $t = 5$, la tâche *Tp2* qui a une périodicité de 5 se réveille de nouveau. Mais l'échéance de cette seconde occurrence de *Tp2* est plus tardive ($d = 9$) que celle de l'occurrence de *Tp1* en cours d'exécution ($d = 8$). Tp1 poursuit donc son exécution. Ainsi la première occurrence de *Tp2* était plus prioritaire que la première occurrence de *Tp1*. Mais la seconde occurrence de *Tp2* est moins prioritaire que la première occurrence de *Tp1*. Ceci est dû au fait que les priorités sont dynamiques.

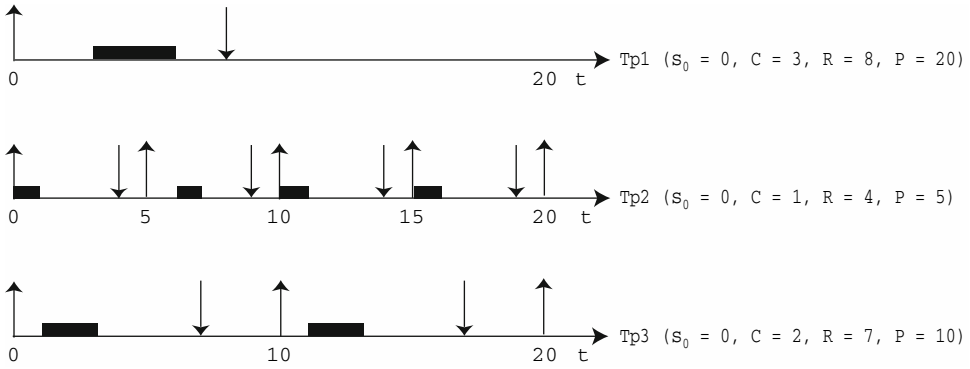

Figure 10.6 – Algorithme Earliest Deadline

La figure 10.7 donne le chronogramme d'exécution pour le même ensemble de tâches mais en appliquant cette fois l'algorithme Inverse Deadline. Dans ce cas, les priorités des tâches sont fixes : ainsi *Tp2* est plus prioritaire que *Tp3* qui elle-même est plus prioritaire de *Tp1*. Ici, chaque occurrence de *Tp2* est toujours prioritaire qu'une occurrence de *Tp1*.

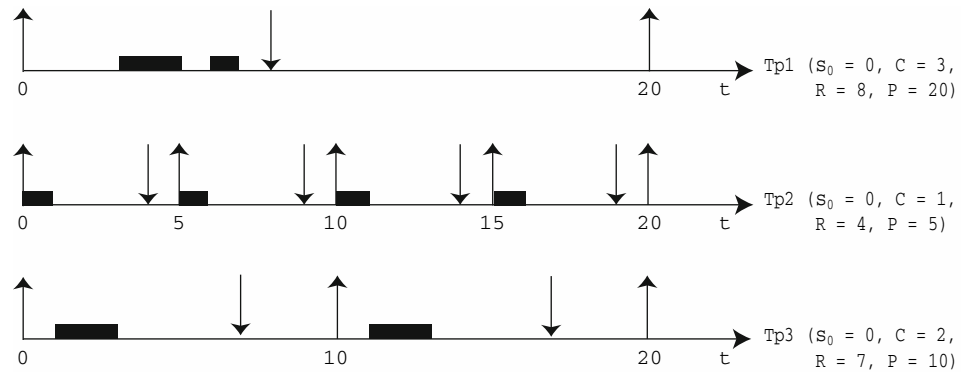

Figure 10.7 – Algorithme Inverse Deadline

c) Ordonnancement de tâches temps réel avec contraintes de ressources

L'inversion de priorité

L'inversion de priorité est la situation pour laquelle une tâche de priorité intermédiaire s'exécute à la place d'une tâche de forte priorité parce que la tâche de forte priorité est en attente d'une ressource acquise par une tâche de plus faible priorité.

La figure 10.8 illustre ce phénomène. On considère les trois tâches périodiques *Tp1*($s_0 = 0$, $C = 3$, $P = 20$), *Tp2*($s_0 = 0$, $C = 2$, $P = 5$), *Tp3*($s_0 = 0$, $C = 2$, $P = 6$). Ces trois tâches périodiques sont ordonnancées selon l'algorithme Rate Monotonic qui attribue la plus forte priorité à la tâche de plus faible période. Le classement des tâches selon l'ordre décroissant des priorités est donc *Tp2*, *Tp3* et *Tp1*. Par ailleurs *Tp2* et *Tp1* utilisent au cours de leur exécution une même ressource critique *R*.

À l'instant $t = 0$, *Tp2* qui est la tâche la plus prioritaire, s'exécute. Elle demande l'accès à la ressource *R*, l'obtient car *R* est libre. À l'instant $t = 2$, à la fin de son exécution, elle libère la ressource. *Tp3* s'exécute. À l'instant $t = 4$, *Tp1* démarre son exécution et demande à obtenir la ressource *R*. *R* est libre et lui est attribué. À l'instant $t = 5$, la tâche *Tp2* se réveille de nouveau (deuxième occurrence) et préempte *Tp1*. *Tp2* demande à accéder à la ressource *R* mais cette fois la tâche est bloquée car la ressource *R* a été attribuée à *Tp1* qui reprend son exécution. À l'instant $t = 6$, la deuxième occurrence de *Tp3* se réveille et préempte *Tp1*. Il y a alors inversion de priorité puisque *Tp3* est de priorité plus faible que *Tp2* et s'exécute à sa place car *Tp2* est bloquée en attente de la ressource *R* détenue par *Tp1*.

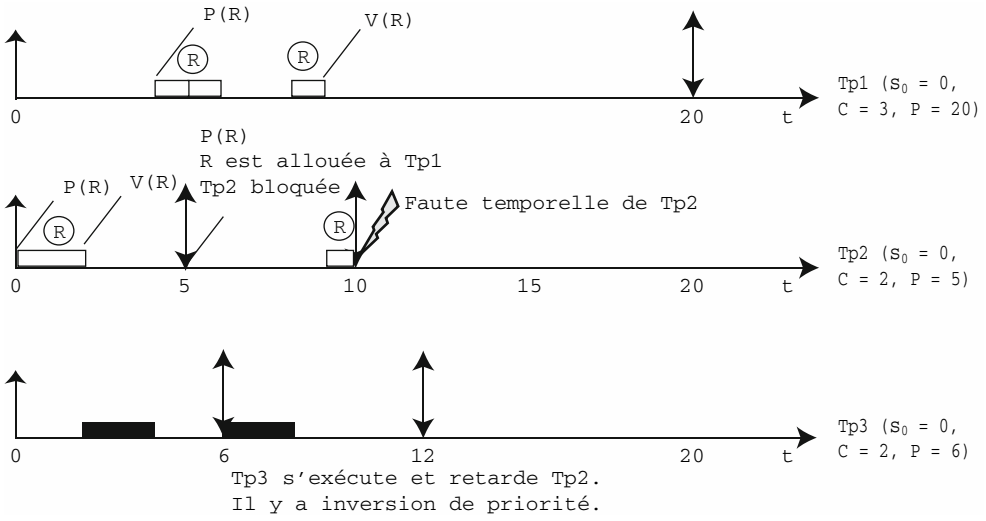

Figure 10.8 – Inversion de priorité

On note sur l'exemple qu'à cause de l'exécution de cette tâche de priorité intermédiaire *Tp3*, la tâche *Tp2* commet une faute temporelle.

Le protocole de la priorité plafonnée

Le protocole de la priorité plafonnée vise à résoudre le problème de l'inversion de priorité.

Il fonctionne selon le principe suivant :

- chaque ressource *R* possède une priorité qui est celle de la tâche de plus haute priorité pouvant demander son utilisation ;
- une tâche *T* hérite de la priorité de cette ressource *R* lorsqu'elle l'obtient et la garde jusqu'à ce qu'elle la libère ;
- cependant, la tâche *T* ne peut obtenir la ressource *R* que si la priorité de *R* est strictement supérieure à celles de toutes les ressources déjà possédées par les autres tâches *T*. Par ce biais, on prévient l'interblocage.

Ainsi sur l'exemple précédent, la ressource *R* a pour priorité, la priorité de la tâche *Tp2*. À l'instant 4, lorsque la tâche *Tp1* acquiert la ressource *R*, elle poursuit son exécution avec la priorité de cette ressource. Ainsi, à l'instant $t = 6$, la tâche Tp3 est de plus faible priorité et ne peut s'exécuter (figure 10.9).

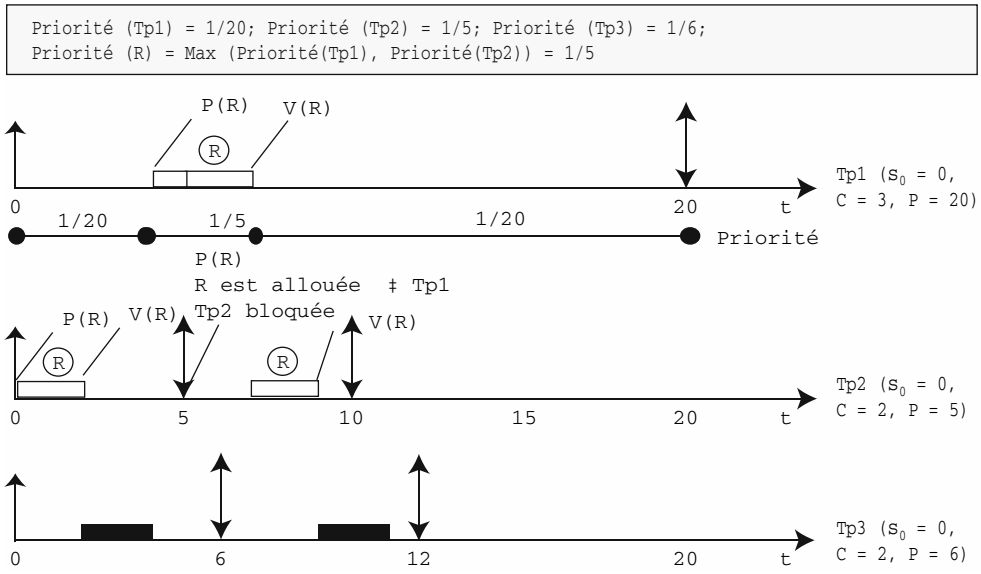

Figure 10.9 – Protocole de la priorité plafonnée

10.1.4 Les systèmes Linux temps réel

a) Limites du système Linux pour le temps réel

Le système Linux est un système d'exploitation qui se révèle inadapté aux exigences temporelles d'une application temps réel. Plus précisément, cinq points sont problématiques :

- l'ordonnancement des processus est un ordonnancement de type temps partagé, visant à offrir à chaque processus une équité d'accès au processeur. Les politiques `SCHED_RR` et `SCHED_FIFO` qualifiées de temps réel n'offrent aucune garantie de respect des contraintes temporelles ;
- le noyau est non préemptif, c'est-à-dire que lorsqu'un processus s'exécute en mode noyau, ce processus ne peut être préempté, même si un processus plus prioritaire a besoin du processeur. Dans un contexte temps réel, ce mode de fonctionnement n'est pas souhaitable puisque l'exécution d'un processus plus prioritaire est retardée ;
- les sections critiques du noyau sont réalisées en masquant les interruptions. Ainsi, la prise en compte d'une interruption destinée à un processus temps réel peut être retardée jusqu'à ce qu'une section critique du noyau soit libérée. Par ailleurs, les exécutions des routines d'interruption ne sont pas ordonnancées conjointement avec les processus, mais sont toujours prioritaires. Dans un système temps réel, il est préférable d'exécuter les routines d'interruption comme des tâches à part entière, c'est-à-dire qualifiées avec une priorité ;
- la fréquence de l'horloge système est de l'ordre de 100 Hz. Les systèmes temps réel sont par définition soumis à des contraintes de temps d'une extrême précision. Une horloge à faible fréquence risque de retarder le déclenchement d'une tâche.

b) Systèmes Linux pour le temps réel

Modifier Linux pour l'adapter au temps réel

Il existe à l'heure actuelle plusieurs techniques pour corriger le noyau Linux et l'adapter aux problématiques du temps réel :

- modifier l'exécution du système Linux afin de rendre le noyau préemptif. De cette façon la prise en compte des événements touchant les tâches les plus prioritaires peut être plus rapide. La difficulté résultante est que le noyau devient réentrant et que les fonctions le composant subissent des appels multiples et concurrents ;
- verrouiller en mémoire centrale les pages de l'espace d'adressage des tâches temps réel afin d'éviter les défauts de pages. De la sorte, les temps d'accès à l'espace d'adressage reste de l'ordre de la nanoseconde ;
- améliorer la résolution temporelle de l'horloge du système ;
- utiliser un micronoyau temps réel cohabitant avec le noyau Linux. Ce micronoyau exécute les tâches temps réel. Le noyau Linux est considéré par ce micronoyau comme la tâche de plus faible priorité. Elle n'est exécutée que lorsqu'aucune autre tâche temps réel n'est prête.

Un exemple de noyau Linux temps réel : RTLINUX

RTLinux a été conçu en ajoutant des propriétés temps réel au système d'exploitation standard Linux. Cette démarche répond à deux objectifs apparemment contradictoires : bénéficier de services temps réel et bénéficier, sans les développer à nouveau,

des services fournis par les systèmes classiques (outils de développement complexes, interfaces utilisateurs graphiques, services réseaux, etc.). La solution adoptée pour RTLinux consiste à séparer la partie temps réel et la partie non-temps réel du système, c'est-à-dire que RTLinux est conçu comme un système dans lequel un noyau temps réel coexiste avec un système non-temps réel. Le noyau temps réel est prioritaire par rapport au système non-temps réel.

Le système RTLinux est composé de trois parties (figure 10.10) :

- une couche logicielle, appelée **émulateur**, fait l'interface avec le matériel et stocke les interruptions destinées à Linux lorsque celui-ci les a bloquées. Il transmet les interruptions à Linux sous forme d'interruptions logicielles ;
- un noyau Linux, dit **noyau de base**, ne voit plus les interruptions qu'à travers les interruptions logicielles que lui transmet l'émulateur. Le noyau Linux a donc été légèrement modifié : toute la partie qui gérait les interruptions matérielles a été remplacée par des appels à l'émulateur ;
- entre ces deux couches, un **noyau temps réel** RTLinux partage le processeur avec le noyau de base Linux. Ce noyau temps réel peut, par son indépendance vis-à-vis du noyau de base Linux, être petit, rapide et déterministe, et supporter même les contraintes temps réel strictes.

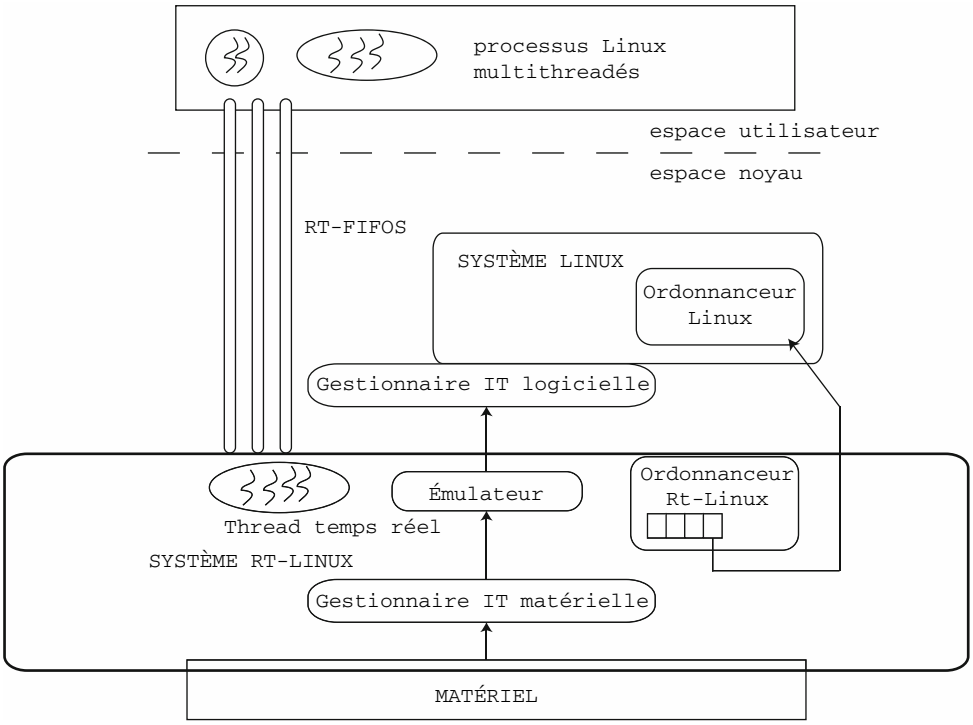

Figure 10.10 – **Architecture de RTLinux**

Les principes de fonctionnement du noyau RTLinux sont les suivants :

- un émulateur d'interruptions est intercalé entre le noyau RTLinux et le système Linux qui fait correspondre à chaque interruption matérielle visible par RTLinux seul, une interruption logicielle, visible par Linux seul. Ainsi le noyau RTLinux intercepte les interruptions matérielles, les traite pour son compte puis, *via* l'émulateur, simule pour le système Linux le masquage ou le démasquage des interruptions en agissant sur les interruptions logicielles correspondantes ;
- l'horloge est modifiée afin de se comporter selon le modèle des horloges à intervalles programmables (mode *one shot timer* et mode *interrupt-on–terminal-count* du circuit d'horloge Intel 8354). Plutôt que d'interrompre le processeur périodiquement, l'horloge est programmée de façon à interrompre le processeur seulement lorsqu'il y a un événement à traiter. Cet événement peut être par exemple le prochain réveil d'une tâche temps réel périodique ;
- la partie temps réel de l'application est constituée d'une seule tâche multithread. qui s'exécute en mode noyau et ne peut être swappée. Ainsi les coûts de commutation entre *threads* sont réduits et l'accès direct à la machine matérielle, est rendu possible, en évitant le coût induit par les appels système. *A contrario*, toute erreur de programmation peut avoir des conséquences graves pour le fonctionnement du noyau et les primitives disponibles pour la programmation sont des primitives de bas niveau, d'une manipulation souvent moins aisée. Les *threads* constituant l'application temps réel sont soit des threads périodiques, soit des threads apériodiques. L'ordonnanceur mis en œuvre par le noyau RTLinux pour ordonnancer les threads temps réel est un ordonnanceur préemptif à priorité fixe comportant 256 niveaux de priorité ;
- un outil de communication non bloquant, les RT-FIFOS, permet la communication des *threads* temps réel avec les processus Linux classiques.

10.2 Systèmes Linux pour les architectures multiprocesseurs

Les architectures parallèles ou multiprocesseurs sont des architectures composées de plusieurs unités de calcul, aptes à exécuter des processus ou des threads en même temps.

Malgré les performances sans cesse croissantes des processeurs, ces architectures sont nécessaires pour l'exécution d'applications requérant de gros volumes de traitements, sur de nombreuses données, telles que les applications manipulant des modèles dans le domaine de la météorologie, la simulation numérique, synthèses et constructions d'images.

Dans ce type d'applications, l'utilisation de plusieurs processeurs permet d'exécuter des taches ou des calculs en parallèle et de réduire ainsi les temps de calcul.

10.2.1 Classification des architectures multiprocesseurs

Les architectures multiprocesseurs peuvent être classées selon leurs caractéristiques. Une taxonomie courante est celle établie au milieu des années 1960, par

Michael J. Flynn[1], qui se base notamment sur le degré de réplication de la partie opérative. Une autre classification est celle établie selon le degré de réplication de la partie mémorisation.

a) Taxonomie de Flynn

Ainsi en fonction des flots d'instructions et de données, les architectures sont réparties en quatre groupes :

- **SISD** (*Single Instruction on Single Data*) : un seul processeur exécute un seul flot d'instruction sur des données résidant dans une seule mémoire. En fait, il n'y a aucun parallélisme.
- **SIMD** (*Single Instruction on Multiple Data*) : plusieurs unités de calcul exécutent simultanément une même instruction sur des données différentes et produisent ainsi plusieurs résultats.
- **MISD** (*Multiple Instructions on Single Data*) : des traitements différents sont appliqués sur des données identiques.
- **MIMD** (*Multiple Instructions on Multiple Data*) : plusieurs unités de calcul exécutent simultanément des traitements différents sur des données différentes. Ce modèle correspond à l'exécution de programmes différents sur des processeurs distincts.

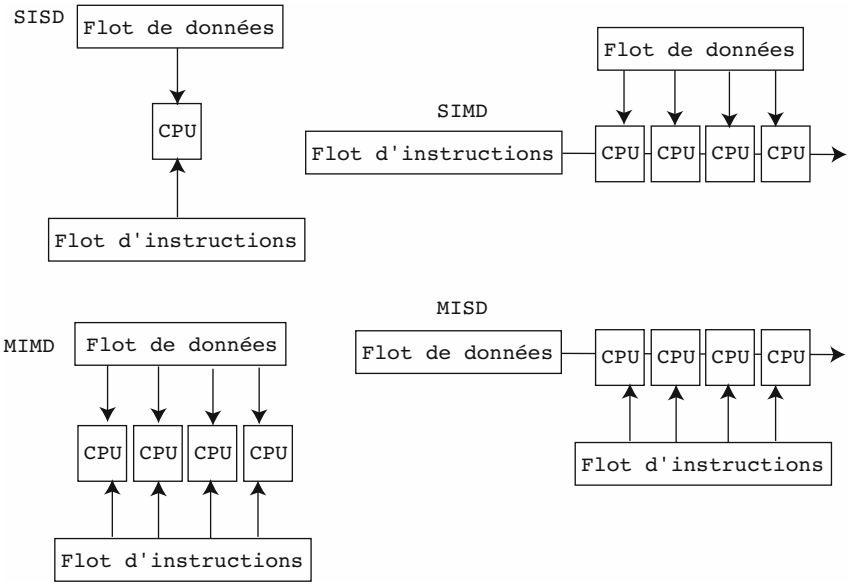

Figure 10.11 – Classification de Flynn

1. Michael J. Flynn, né en mai 1934 à New York, professeur émérite à l'université de Stanford.

b) Taxonomie selon la mémoire

Les machines MIMD peuvent être de deux types différents :

- Les processeurs accèdent à une seule mémoire qui est partagée. Dans ce type d'architecture, le nombre de processeurs est limité. L'interconnexion processeurs-mémoire centrale s'effectue à l'aide d'un bus. La relation entre les processeurs et la mémoire est symétrique, on parle d'architectures SMP (*Shared Memory Multiprocessor*).
- Les processeurs disposent chacun d'une mémoire, et chaque processeur peut accéder à n'importe quelle mémoire s'il dispose des droits nécessaires. De ce fait, la mémoire est distribuée. Le nombre de processeurs peut être plus important. La communication s'effectue grâce à un réseau d'interconnexion offrant une large bande passante.

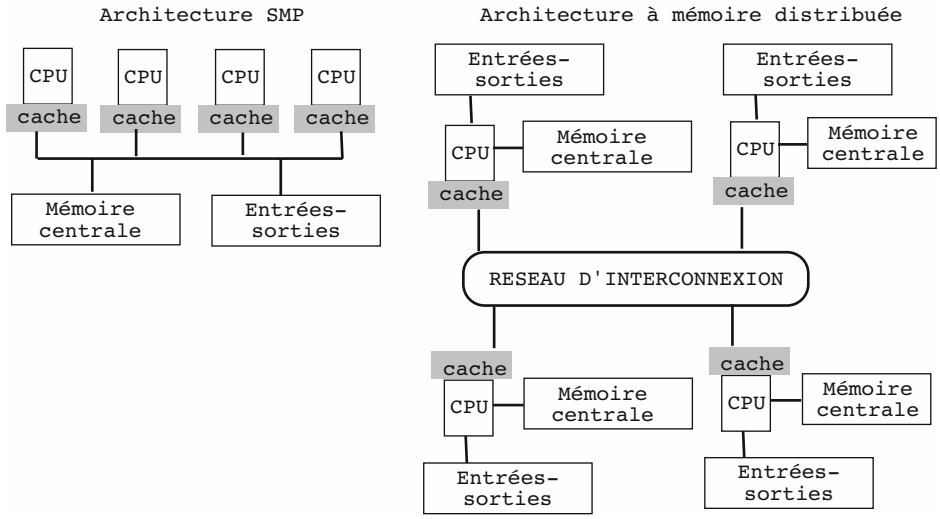

Figure 10.12 – Architecture mémoire

c) Loi de Amdahl

Gene Myron Amdahl[1] a énoncé une loi exprimant le gain de performance qu'on peut attendre d'un ordinateur en améliorant une composante de sa performance. Cette loi s'applique notamment au gain attendu lorsque l'on multiplie le nombre de processeurs de la machine.

Cette loi est de la forme suivante. Elle indique le gain de temps *S* que va apporter un système multiprocesseur en fonction :

- du nombre de processeurs *N ;*

1. Gene Myron Amdahl, né le 16 novembre 1922.

- de la proportion d'activité non parallélisable *F*.

$$S = \frac{1}{\frac{F + (1 - F)}{N}}$$

Sur la figure 10.13, la courbe du haut représente le cas idéal. L'ajout d'un processeur multiplie d'autant la vitesse d'exécution Malheureusement, l'ensemble de l'architecture de la machine ne peut pas être parallélisé. Par ailleurs, la gestion des différents processeurs induit un surcoût. Aussi, le gain réel *S* est moins important. Ainsi un ensemble de 10 processeurs permet un gain réel *S* égal à seulement 5 processeurs.

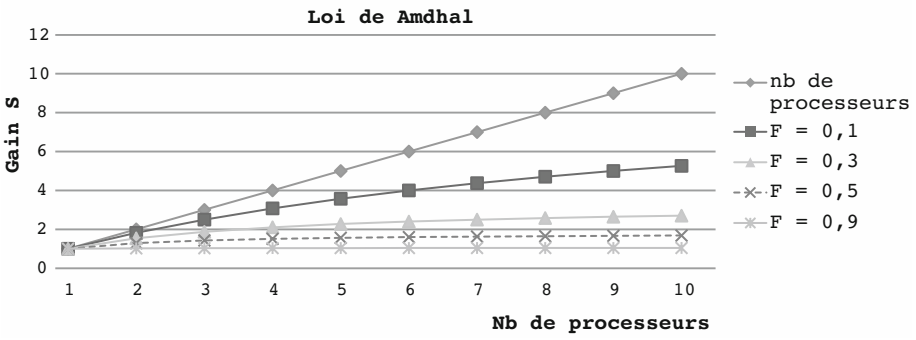

Figure 10.13 – **Loi de Amdahl**

10.2.2 Architectures SMP Linux

Le noyau Linux supporte les architectures multiprocesseurs de type SMP depuis la version 2.2 du noyau. Des modifications ont été apportées notamment sur les points suivants :

- Initialisation du noyau ;
- Ordonnancement des exécutions ;
- Synchronisation au sein du noyau ;
- Gestion des interruptions.

a) Initialisation du noyau

L'un des processeurs est choisi par le BIOS pour être le processeur de *boot* (BSP) ; les autres processeurs appelés processeurs d'applications (AP) sont initialement désactivés.

Le processeur de *boot* exécute la fonction d'initialisation du noyau (`start_kernel()`), puis il active les autres processeurs *via* la fonction `smp_init()` : chaque processeur d'applications reçoit sa pile, initialise les mécanismes de prise en compte des exceptions

et des interruptions. Une tâche « `cpu_idle` » est créée pour chaque processeur, qui sont ensuite aptes à exécuter leurs propres processus.

b) Ordonnancement des exécutions

Lorsqu'un système est composé de plusieurs processeurs, une politique d'équilibrage de charge (*load balancing*) peut être mise en œuvre afin de répartir les différentes exécutions entre les processeurs, de sorte qu'il n'y ait pas de processeurs en situation de surcharge et d'autres en situation de sous charge. Cette répartition est faite à l'insu des utilisateurs par un module classiquement appelé « répartiteur de charge », et vise à améliorer les taux de réponse et l'utilisation des ressources.

Ce mécanisme de répartition de charge pose cependant un problème quant à l'utilisation des caches du processeur. En effet, chaque processeur dispose d'un petite mémoire rapide qui contient les données et instructions dont il a besoin dans l'instant présent et dans un futur proche. L'intérêt de ce cache est d'éviter les accès à la mémoire centrale qui sont des opérations toujours pénalisantes pour le processeur. Pour un processus, le cache du processeur est considéré comme « chaud » s'il contient des données et instructions utiles à ce processus. Sinon, il est considéré comme « froid ». Lorsqu'un processus démarre son exécution, le cache est froid ; il devient chaud au fur et à mesure des accès du processus à son espace d'adressage. Malheureusement, lorsqu'un processus est déplacé d'un processeur à un autre, il perd le bénéfice du cache du processeur sur lequel il s'exécutait, et doit reprendre son exécution avec un cache « froid ».

Ordonnancement Linux multiprocesseurs

Chaque processeur dispose de deux files pour gérer l'ordonnancement des processus : une file des processus actifs (*active runqueue*) et une file des processus expirés (*expired runqueue*). Chaque file dispose de 140 niveaux de priorités, les 100 premiers niveaux réservés à des processus « temps réels » et les 40 restants à des processus utilisateurs. La file des processus actifs est servie avec une politique FIFO. Chaque processus s'exécute au plus pour un quantum d'exécution ; lorsque celui-ci est expiré, le processus est déplacé dans la file des processus expirés. Son quantum et sa priorité sont recalculés. Lorsqu'il n'y a plus de processus dans la file des processus actifs, les deux files sont échangées, la file des processus expirés devenant la nouvelle file active.

Le processus élu sur un processeur est le processus de plus forte priorité. L'ordonnancement est préemptif. Lorsqu'un processus est préempté, il est réinséré dans la file des processus actifs, à son niveau de priorité.

Lorsqu'un processus a consommé son quantum, il est déplacé dans la file des processus expirés. Sa priorité est recalculée, afin d'éviter les phénomènes de famine. Ce recalcul de priorité vise à pénaliser les processus utilisant beaucoup le processeur, pour notamment permettre aux processus endormis en attente d'entrées-sorties d'accéder également au processeur.

Par ailleurs, le système met en œuvre un mécanisme de répartition de charge. Lorsqu'un processus est créé, il est affecté à un processeur. Cependant, toutes les

200 ms, le répartiteur de charge évalue la charge des processeurs. S'il détecte que des processeurs sont sous-utilisés, il déplace des processus venant d'un processeur plus chargé.

c) Synchronisation au sein du noyau

Dans un système monoprocesseur, les situations de concurrence au sein du noyau sont limitées et résolues grâce à la désactivation des interruptions ou grâce à l'utilisation de sémaphores du noyau.

Les situations de concurrence sont plus nombreuses dans le cadre d'un système multiprocesseur puisque plusieurs exécutions ont lieu en parallèle et que plusieurs d'entre elles peuvent être en train d'exécuter du code du noyau.

Pour protéger les structures de données du noyau de ces accès concurrents, le noyau utilise des verrous tournants (*spin locks*).

Verrous tournants en exclusion mutuelle

Un verrou tournant est un outil de synchronisation à deux états, verrouillé et déverrouillé, qui permet de réaliser des exclusions mutuelles.

Trois opérations atomiques peuvent être appliquées au verrou :

- la macro `SPIN_LOCK_UNLOCKED` initialise un verrou à l'état déverrouillé ;
- la macro `spin_lock` est appelée par un processus désirant acquérir un verrou. Si le verrou est dans l'état déverrouillé, le processus l'acquiert et le verrou passe dans l'état verrouillé. Par contre si le verrou est dans l'état verrouillé, le processus entre dans une phase d'attente active jusqu'à trouver le verrou disponible ;
- la macro `spin_unlock` libère un verrou précédemment acquis. Le verrou revient dans l'état déverrouillé.

Verrous tournants en lecture-écriture

Le noyau Linux définit également des verrous tournants en lecture-écriture. Ce type de verrou autorise des lectures en simultanée, en exclusion des écritures. Les écritures s'exercent en exclusion les unes des autres.

Trois opérations atomiques peuvent être appliquées au verrou :

- la macro `RW_LOCK_UNLOCKED` initialise un verrou à l'état déverrouillé ;
- la macro `read_lock` est appelée par un processus désirant acquérir un verrou en lecture. Si le verrou n'est pas verrouillé en écriture, le processus l'acquiert et le verrou passe dans l'état verrouillé en lecture. Par contre si le verrou est dans l'état verrouillé en écriture, le processus entre dans une phase d'attente active jusqu'à trouver le verrou disponible ;
- la macro `read_unlock` libère un verrou précédemment acquis en lecture ;
- la macro `write_lock` est appelée par un processus désirant acquérir un verrou en écriture. Si le verrou n'est pas verrouillé en écriture ou en lecture, le processus l'acquiert et le verrou passe dans l'état verrouillé en écriture. Par contre si le verrou est dans l'état verrouillé en écriture ou en lecture, le processus entre dans une phase d'attente active jusqu'à trouver le verrou disponible ;

- la macro `write_unlock` libère un verrou précédemment acquis en écriture.

Utilisation des verrous au sein du noyau

La synchronisation au sein du noyau Linux est gérée par plusieurs niveaux de verrous :

- Un verrou tournant global de noyau procure un accès exclusif à l'entièreté du noyau. Par le biais de ce verrou, un seul processus à la fois est autorisé à parcourir le code du noyau. Ce verrou global doit notamment être acquis par les processus désirant utiliser le système de gestion de fichiers, les structures de données liées aux IPC et à la gestion du réseau.
- Des verrous tournants en lecture-écriture associés à des structures de données spécifiques telles que les files d'attentes, les files d'ordonnancement, la manipulation des inodes. La définition de ces différents verrous augmente les performances du système en autorisant l'exécution de plusieurs chemins de contrôle du noyau en parallèle, à partir du moment où ceux-ci manipulent des structures de données protégées par des verrous différents.

d) Gestion des interruptions

Afin d'exploiter pleinement le parallélisme sur une architecture de type SMP, il est important de pouvoir délivrer les interruptions à tout processeur.

Le composant Intel E/S APIC (*I/O Advanced Programmable Interrupt Controller*) a été introduit pour gérer les interruptions dans les environnements multiprocesseurs. Il est composé de trois parties (figure 10.14) :

- l'APIC local est interne à chaque processeur ; il gère notamment les interruptions délivrées à chaque processeur ;
- l'APIC E/S est général ; il dispose d'une table de redirections qui permet de spécifier pour chaque interruption, sa priorité, le processeur destinataire de l'interruption et la façon de le sélectionner ;
- le bus APIC dédié permet de faire communiquer les APIC locaux avec l'APIC E/S.

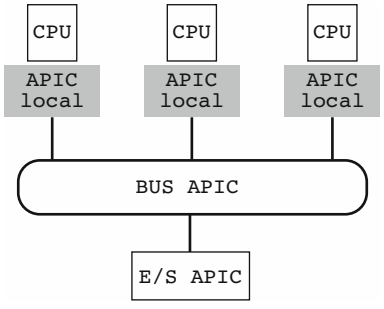

Figure 10.14 – Système APIC

L'APIC E/S peut distribuer les requêtes d'interruptions entre les différents APIC locaux de deux façons :

- selon un mode fixe : dans ce cas une interruption est délivrée aux processeurs spécifiés dans la table de redirection ;
- selon un mode dynamique : dans ce cas une interruption est délivrée au processeur qui exécute le processus de plus basse priorité.

Parmi les processeurs recevant une interruption relayée par l'E/S APIC, un seul doit effectivement la traiter. Chaque descripteur d'interruption définie par le noyau contient un champ `IRQ_INPROGRESS` protégé par un verrou tournant, qui est à vrai si le gestionnaire d'interruption est en exécution sur un processeur. Ainsi, le premier processeur qui accède en exclusion des autres processeurs au descripteur met à vrai le champ `IRQ_PROGRESS` et démarre le traitant d'interruption. Les autres processeurs trouvant ensuite le champ déjà positionné à vrai, ne font rien au regard de l'interruption reçue.

Exercices

10.1 Qu'avez-vous retenu ?

Pour chacune des questions suivantes, choisissez la ou les réponses qui vous semblent exactes.

Question 1 – Une application temps réel :

- ☐ a. est une application devant fournir le plus petit temps de réponse possible.
- ☐ b. est une application soumise à des échéances de temps d'exécution.
- ☐ c. est une application en temps partagé.

Question 2 – Soit l'ensemble de tâches périodiques composé des trois tâches *Tp1*($s_0 = 0$, $C = 3$, $R = 8$, $P = 15$), *Tp2*($s_0 = 0$, $C = 1$, $R = 4$, $P = 10$) et *Tp3*($s_0 = 0$, $C = 2$, $R = 2$, $P = 5$).

- ☐ a. Pour l'algorithme *Rate Monotonic*, la tâche la plus prioritaire est la tâche *Tp2*.
- ☐ b. Pour l'algorithme *Rate Monotonic*, la tâche la plus prioritaire est la tâche *Tp3*.
- ☐ c. Pour l'algorithme *Inverse Deadline*, la tâche la plus prioritaire est la tâche Tp3.
- ☐ d. Pour l'algorithme *Inverse Deadline*, la tâche la plus prioritaire est la tâche Tp1.

Question 3 – Une architecture SMP :

- ☐ a. est une architecture monoprocesseur.
- ☐ b. est une architecture multiprocesseur à mémoire distribuée.
- ☐ c. est une architecture multiprocesseur à mémoire commune.

Question 4 – Un verrou tournant :

- ☐ a. est un outil de synchronisation par attente activée.
- ☐ b. est un mécanisme de prise en compte des interruptions.

Question 5 – L'équilibrage de charge :

- ☐ a. permet de répartir les exécutions de processus sur différents processeurs.
- ☐ b. permet de garantir le respect de contraintes temporelles.
- ☐ c. est la situation pour laquelle une tâche de priorité intermédiaire s'exécute à la place d'une tâche de forte priorité parce que la tâche de forte priorité est en attente d'une ressource acquise par une tâche de plus faible priorité.

Question 6 – L'inversion de priorité :

- ☐ a. permet de répartir les exécutions de processus sur différents processeurs.
- ☐ b. permet de garantir le respect de contraintes temporelles.
- ☐ c. est la situation pour laquelle une tâche de priorité intermédiaire s'exécute à la place d'une tâche de forte priorité parce que la tâche de forte priorité est en attente d'une ressource acquise par une tâche de plus faible priorité.

10.2 Algorithmes d'ordonnancement temps réel

On considère les trois tâches Tp1 ($s_0 = 0$, $C = 1$, $R = 3$, $P = 3$), Tp2($s_0 = 0$, $C = 1$, $R = 4$, $P = 4$), Tp3 ($s_0 = 0$, $C = 2$, $R = 6$, $P = 6$).

Décrivez graphiquement les séquences obtenues dans le cas des ordonnancements *Rate Monotonic*, et *Earliest Deadline*.

10.3 Algorithmes d'ordonnancement temps réel

On considère les deux tâches Tp1 ($s_0 = 0$, $C = 3$, $R = 7$, $P = 10$) et Tp2 ($s_0 = 0$, $C = 3$, $R = 4$, $P = 5$).

Décrivez graphiquement les séquences obtenues dans le cas des ordonnancements *Inverse Deadline* et *Earliest Deadline*. Notez les éventuelles fautes temporelles.

10.4 Inversion de priorité

On considère trois processus P1, P2, P3 dont les caractéristiques sont les suivantes :

	Date d'arrivée	Temps d'exécution	Priorité (plus petite valeur = plus grande priorité)
P1	0	5 unités	4
P2	2	4 unités	1
P3	3	**4** unités	2

Les trois processus sont ordonnancés selon une politique de priorités préemptives.

On suppose à présent que les processus P1 et P2 utilisent une même ressource critique R1, protégée par un sémaphore *Sem1* initialisé à 1. P3 lui ne fait que du calcul.

Les étapes des deux processus P1 et P2 sont les suivantes :

P1	P2
Calcul durant 1 unité *P(Sem1)* Faire calcul en utilisant *R1* durant 2 unités *V(Sem1)* Calcul durant 2 unités	Calcul durant 1 unité *P(Sem1)* Faire calcul en utilisant *R1* durant 2 unités *V(Sem1)* Calcul durant 1 unité

1) Construisez le chronogramme d'exécution des trois processus en tenant compte du partage de la ressource R1 entre P1 et P2. Les opérations *P(Sem1)* et *V(Sem1)* ne comptent pas de d'unités de temps si elles sont passantes (on considère qu'elles sont instantanées). Que se passe-t-il ? Est-ce que P2 s'exécute effectivement comme étant le processus le plus prioritaire ?
2) Construisez le chronogramme d'exécution des trois processus en appliquant un protocole de priorité plafonnée.

Solutions

10.1 Qu'avez-vous retenu ?

Question 1 – Une application temps réel :

- ☐ b. est une application soumise à des échéances de temps d'exécution.

Question 2 – Soit l'ensemble de tâches périodiques composé des trois tâches *Tp1*($s_0 = 0$, $C = 3$, $R = 8$, $P = 15$), *Tp2*($s_0 = 0$, $C = 1$, $R = 4$, $P = 10$) et *Tp3*($s_0 = 0$, $C = 2$, $R = 2$, $P = 5$).

- ☐ b. Pour l'algorithme *Rate Monotonic*, la tâche la plus prioritaire est la tâche *Tp3*.
- ☐ c. Pour l'algorithme *Inverse Deadline*, la tâche la plus prioritaire est la tâche Tp3.

Question 3 – Une architecture SMP :

- ☐ c. est une architecture multiprocesseur à mémoire commune.

Question 4 – Un verrou tournant :

- ☐ a. est un outil de synchronisation par attente activé.

Question 5 – L'équilibrage de charge :

- ☐ a. permet de répartir les exécutions de processus sur différents processeurs.

Question 6 – L'inversion de priorité :

- ☐ b. permet de garantir le respect de contraintes temporelles.
- ☐ c. est la situation pour laquelle une tâche de priorité intermédiaire s'exécute à la place d'une tâche de forte priorité parce que la tâche de forte priorité est en attente d'une ressource acquise par une tâche de plus faible priorité.

10.2 Algorithmes d'ordonnancement temps réel

Les chronogrammes d'exécution sont donnés par la figure 10.15. La période d'étude est [0 , 12]. On peut noter que dans le cas de l'algorithme *Earliest Deadline*, les dernières occurrences de chacune des trois tâches ont la même urgence. La figure 10.15 donne une solution, toute permutation entre les trois dernières exécutions étant correcte.

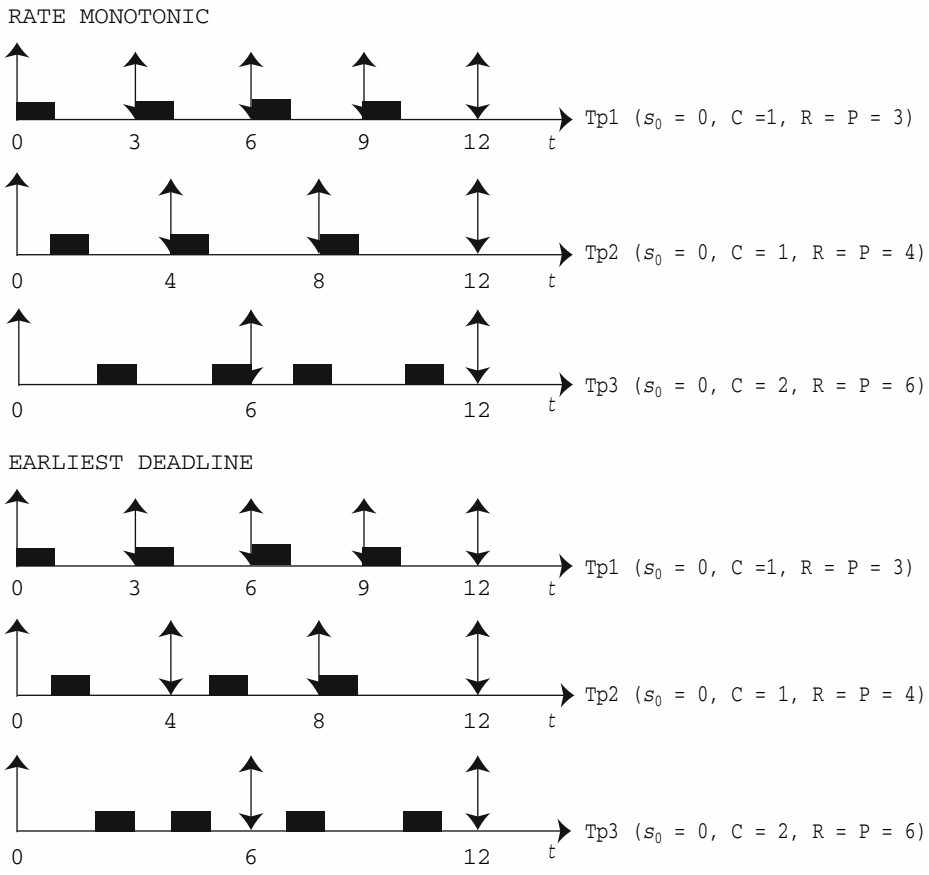

Figure 10.15 – Chronogramme d'exécution

10.3 Algorithmes d'ordonnancement temps réel

Les chronogrammes d'exécution sont donnés par la figure 10.16.
La période d'étude est [0 , 10].

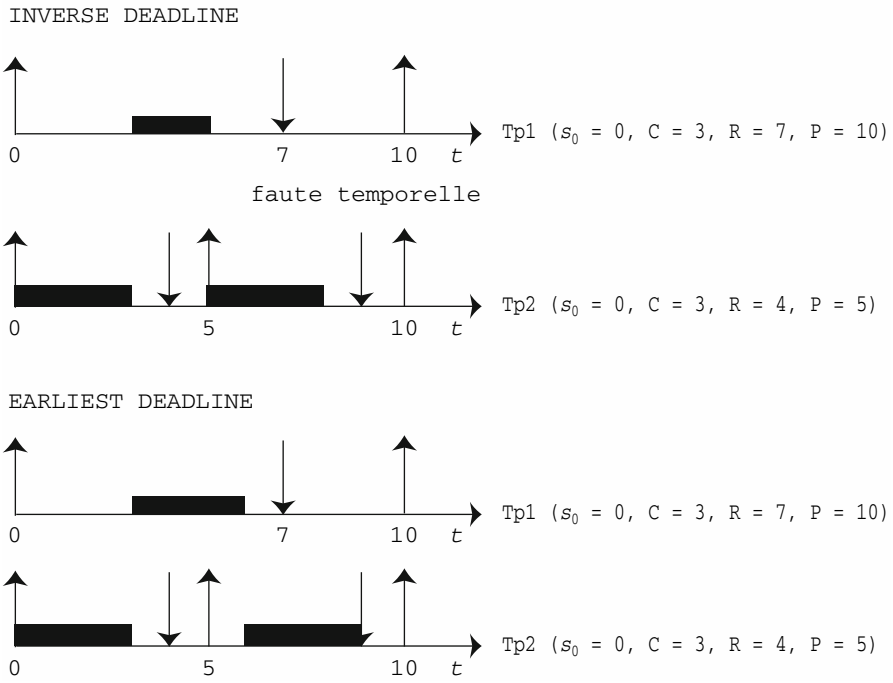

Figure 10.16 – **Chronogramme d'exécution**

10.4 Inversion de priorité

1) Le chronogramme est donné par la figure 10.17.

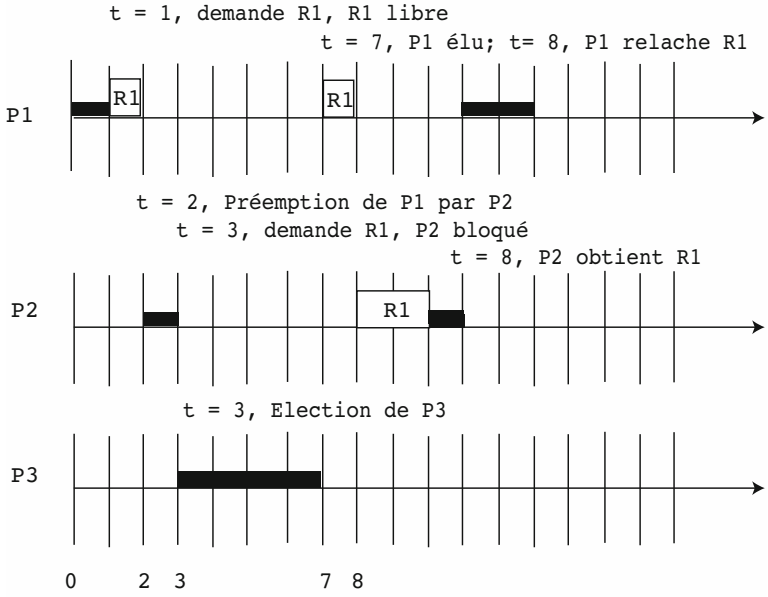

Figure 10.17

Non, P2 ne s'exécute pas comme étant le processus le plus prioritaire, car il est bloqué en attente de la ressource R1 possédée par le processus P1.
Il y a inversion de priorité.

2) Le chronogramme est donné par la figure 10.18. La ressource R1 a pour priorité la valeur 1.

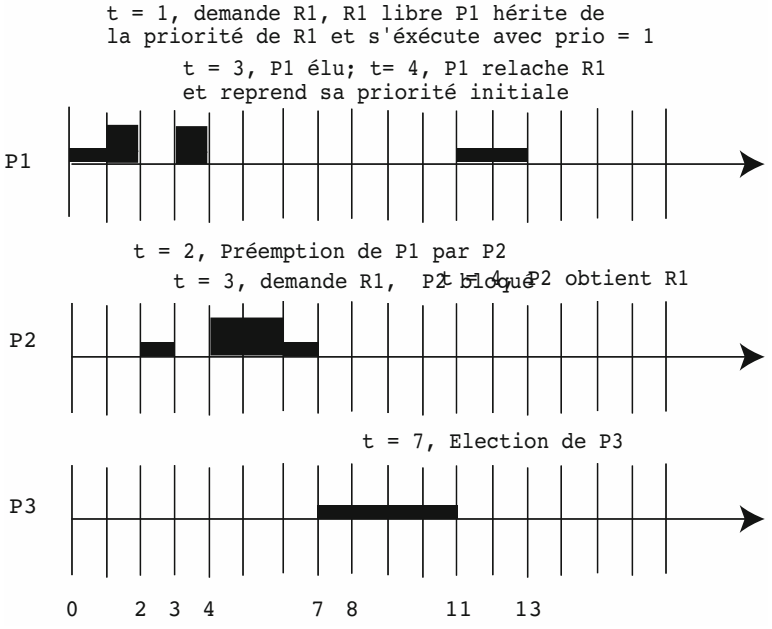

Figure 10.18

INDEX

Q