Université de Toulon
Département d'Informatique

Architecture des ordinateurs (I22): Ordonnancement des processus, mémoires et systèmes de fichiers

Joseph Razik

dernière mise à jour: 18 mars 2020

Table des matières

Chapitre 1

Système d'exploitation

1.1 Pourquoi un système d'exploitation ?

Le contexte général de l'utilisation d'un ordinateur est le suivant :
D'un côté, un (ou plusieurs) utilisateur(s) (user)
— Attend un résultat de l'ordinateur
— Utilise des outils pour sa requête
— Utilise un (des) programme(s) pour accomplir cette tâche

D'un autre côté, un ordinateur dispose de ressources physiques
— De calcul (CPU)
— De mémoire (RAM)
— De stockage (Disque Dur, CD, clés)
— D'interface (clavier, écran, imprimante ...)

Plusieurs question peuvent alors se poser :
— Comment l'utilisateur peut tirer profit des ressources à sa disposition pour obtenir son résultat ?
— Comment les ressources peuvent "communiquer" entre elles pour arriver au résultat ?
— Comment organiser les ressources pour arriver au résultat demandé de manière efficace ?
— Comment partager ces ressources entre plusieurs utilisateurs simultanés ?
— Comment ne pas tout recommencer si un élément change (modification du CPU, changement de programme, passage à une autre plate-forme ...)

La réponse à cette question est le système d'exploitation. En effet, ce système est un programme qui va généralement être exécuté en premier sur la machine, et c'est ce programme qui va contrôler l'utilisation correcte des ressources matérielles de la machine. Par ailleurs, le système d'exploitation va aussi s'assurer que l'exécution du programme de l'utilisateur ne perturbe pas le système complet.

Ainsi pour résumer, un système d'exploitation, ou OS (Operating System),
— sert d'interface entre le matériel et les applications,
— gère le partage des ressources matérielles entre les applications et les utilisateurs,
— gère les échanges entre les applications,
— gère l'indépendance des applications et leur isolation.

Voici quelques exemples de systèmes d'exploitation plus ou moins connus :
— Unix, Linux, BSD, Mac OS X, ...
— VMS, Symbian, Android, ...
— DOS, Windows (3.11, 95, XP, CE, ...),
— Beaucoup d'autres, souvent spécialisés et propriétaires.

1.2 Rôles d'un système d'exploitation

Un système d'exploitation a pour rôle principal de gérer les ressources de l'ordinateur et de proposer des services à la fois pour l'utilisateur que pour les matériels et les programmes. Ces services vont également permettre l'administration et la gestion de ces ressources et services. Les principaux services indispensables recouvrent par exemple :

- Un service de gestion des utilisateurs,
- Un service de gestion des fichiers,
- Un service de gestion des tâches qui s'exécutent,
- Un service de gestion de l'information.

1.2.1 Gestion des utilisateurs

Un système d'exploitation peut gérer l'identification d'un ou plusieurs utilisateurs. On caractérise ces systèmes par les termes suivants :

- Mono-utilisateur : un seul utilisateur peut utiliser le système en même temps,
- Multi-utilisateurs : plusieurs utilisateurs peuvent utiliser le système en même temps.

Les premiers système était mono-utilisateur : une seule personne pouvait utiliser l'ordinateur à la fois, quand s'était son tour, et il n'y avait pas de personnalisation de l'environnement ou des données.

Cependant, rapidement les systèmes multi-utilisateurs ont permis à plusieurs personnes d'utiliser un même ordinateur en même temps ou de pouvoir séparer les espaces de travail de ceux-ci. Un tel système d'exploitation multi-utilisateur doit proposer les services suivants :

- Service de connexion/déconnexion
  - Vérification de validation de l'accès (login/passwd)
  - Initialisation de l'environnement et des ressources des utilisateurs
- Service d'administration des utilisateurs
  - Pour les utilisateurs : gestion de leurs ressources, modification de paramètres (passwd,...)
  - Pour l'administrateur : gestion globale des ressources, ajout d'utilisateurs, politique d'allocation des ressources, sauvegarde, sécurité, etc...

1.2.2 Gestion des fichiers

Un système d'exploitation a généralement recourt à des périphériques externes pour lire ou écrire des données qui y sont ou y seront stockées. Ces supports physiques sont en général découpés en zones de longueur fixe (blocs). Le système d'exploitation doit donc intégrer un système de gestion des fichiers qui va gérer :

- L'espace physique pour le stockage,
  - Le système assure l'allocation et la restitution des blocs (liste des blocs libres) pour gérer l'espace libre,
  - Gère l'allocation des blocs physiques sur le support,
- L'accès physique aux fichiers,
- Les liens entre la structuration logique et l'allocation physique,
  - Gère les répertoires et mémorise la structure de données du système de gestion de fichiers.
- Les protections associées aux fichiers, habituellement sous forme de :
  - listes de droits d'accès (droits `rwx` sous unix),
  - liste de contrôle d'accès (Access Control List - ACL).

Une fois que le système d'exploitation sait retrouver un fichier et les droits associés, la manipulation des données de ceux-ci implique plusieurs mécanismes, c'est ce qu'on appelle le transferts logique entre "système de fichier" et "processus utilisateurs". Il y a principalement nécessité des deux mécanismes suivants :

- Un mécanisme de gestion des méthodes d'accès
  - Fonctions spécifiques à la structure de fichiers (open, close, read, write, ...)
  - Organisations traditionnellement implantées :
    - Les fichiers séquentiels : accès exclusivement séquentiel ;

    - Les fichiers séquentiels indexés : accès direct via une table d'index et des clés ;
    - Les fichiers directs : accès direct par le numéro de l'enregistrement.
- Un mécanisme de gestion des tampons
    - Gère les tampons nécessaires aux entrées/sorties physiques,
    - Résident dans une zone mémoire du système pour rendre les entrées/sorties performantes,
    - Optimisation par heuristiques pour l'association tampons-blocs physiques.

1.2.3 Gestion des tâches

La principale différence entre ordinateur muni d'un processeur et un circuit électronique dédié vient du fait que le processeur est en quelques sortes « configurable » pour exécuter tous type de tâches alors qu'un circuit dédié ne peut faire que la tâche pour laquelle il a été conçu.

Le rôle du système d'exploitation est aussi de gérer ces différentes « configuration » (les programmes utilisateurs) afin qu'elles puissent se réaliser correctement et si possible de manière « efficace ». Cette efficacité va dépendre du point de vue de l'utilisateur : plus de débit de calcul ou plus de partage entre les tâches.

On peut distinguer ainsi un système d'exploitation selon sa gestion des processus à exécuter. On trouve les systèmes :

- Mono-tâche : une seule tâche peut être exécutée en même temps. Les autres tâches doivent attendre la fin de la tâche courante pour pouvoir commencer.
- Multi-tâches : plusieurs tâches peuvent s'exécuter en parallèle, sans bloquer le lancement d'une nouvelle tâche supplémentaire.

Une autre distinction existe :

- Systèmes mono-programmés
    - Premiers systèmes d'exploitation développés,
    - Un seul processus réside en mémoire et utilise toutes les ressources,
    - Ordinateur utilisé séquentiellement par les différents processus,
- Systèmes multi-programmés
    - Plusieurs processus résident en mémoire,
    - Partagent l'unité centrale,
    - Cèdent leur tour durant les entrées/sorties

La distinction entre mono-tâche et mono-programmée est (très) finie : dans les deux cas, un seul programme s'exécutera jusqu'à sa terminaison mais la différence est que dans un système mono-programmé, un seul processus ne peut être chargé en mémoire.

Un système multi-programmé est en fait un système multi-tâche dont la politique d'exécution des processus est celle de relâcher l'unité centrale au moment des attentes d'entrée/sortie.

La figure 1.1 illustre le fonctionnement d'un système mono-programmé : 3 tâches sont demandées, certaines pendant que d'autres s'exécutent et entrent donc en concurrence. Il apparaît clairement qu'un seul processus s'exécute à la fois et que le suivant ne commencera que quand le précédent aura terminé.

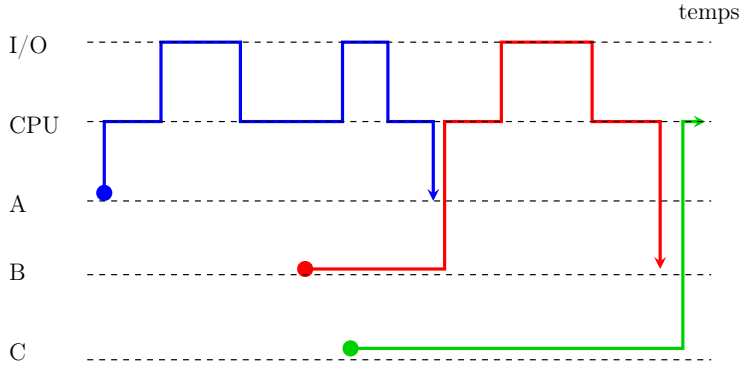


FIGURE 1.1 – Exemple d'exécution dans un système mono-programmé.

La figure 1.2 illustre le fonctionnement d'un système multi-programmé dans les mêmes conditions que précédemment. La différence avec le système mono-programmé est visible dans le fait qu'un processus concurrent peut démarrer son exécution avant la fin des processus précédents, utilisant les temps d'attente sur entrée/sortie du processus actif.

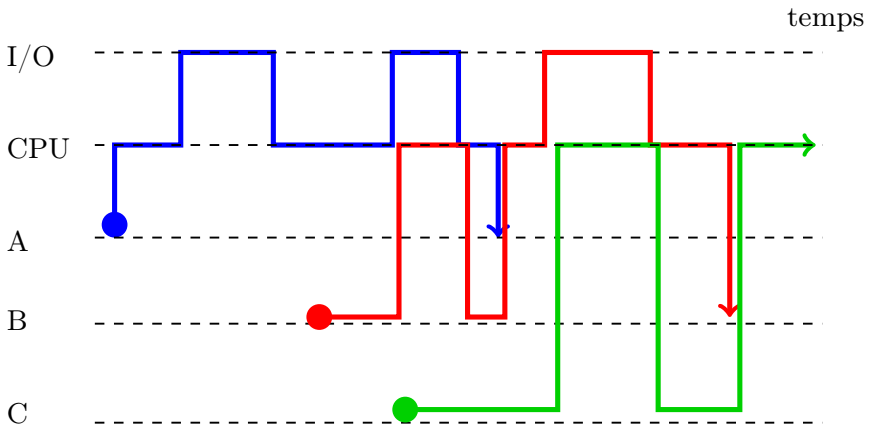


FIGURE 1.2 – Exemple d'exécution dans un système multi-programmé.

Le tableau suivant (tab. 1.1) donne les statistiques d'utilisation d'une unité centrale pour ces deux systèmes. L'unité centrale est ainsi utilisée de manière plus efficace dans le cas d'un système multi-programmé quand le cas d'un système mono-programmé.

Cependant on note un augmentation importante du pourcentage utilisé par le système. En effet, celui-ci passe de 10% à 35%. Ce temps consommé par le système se décompose en 25% pour les utilisateurs, 10% pour l'auto-gestion du système. Le système devant gérer les changements de processus sur le processeur, cela demande des ressources supplémentaires. Malgré cela, le système reste plus efficace et ne passe plus que 5% du temps à attendre au lieu des 50% initiaux. Il y a ainsi une optimisation de l'utilisation des ressources coûteuses de l'ordinateur.

Temps unité centrale	monoprogrammé	multiprogrammé
En attente	50%	5%
Utilisé par le système	10%	35%
Consommé par les utilisateurs	40%	60%

TABLE 1.1 – Comparaison des taux d'utilisation de l'UC dans des systèmes mono- et multi-programmés.

Toutefois, ce système n'est pas équitable et seul les temps morts du processus actif sont laissés aux autres processus concurrents. Un autre type de système permet d'améliorer cette équitabilité, c'est le système à temps partagé qui sera au coeur du prochain chapitre.

1.2.4 Gestion de l'information

La gestion de l'information est une tâche principale d'un système d'exploitation. Ceci recouvre la transmission des données depuis le disque externe par exemple jusqu'au processeur. Le système d'exploitation gère ainsi l'organisation de la mémoire (savoir retrouver les données), la protection des données en mémoire, la communication et la synchronisation des processus. Un chapitre sera également consacré à la cette gestion de l'information du transit depuis la mémoire centrale jusqu'au processeur.

1.3 Types de système d'exploitation

Il existe plusieurs moyen de qualifier les systèmes d'exploitation selon leurs caractéristiques et leur usage. On distingue par exemple trois familles de systèmes d'exploitation selon leurs fonctionnalités :

— Systèmes temps réel
Caractérisés par la prise en compte des contraintes temporelle de l'environnement,
— Systèmes de développement
Systèmes à temps partagé interactifs,
— Systèmes pour l'exploitation
Utilisé sur des « *main-frame* », traitement par lots (traitement de masse). Objectif : optimisation de l'utilisation coûteuse des ressources matériels.

Dans chacune de ces familles, on pourra de nouveau distinguer les caractéristiques des systèmes d'exploitations selon leur gestion des utilisateurs ou des tâches.

1.4 Deux exemples de systèmes d'exploitation

1.4.1 Unix

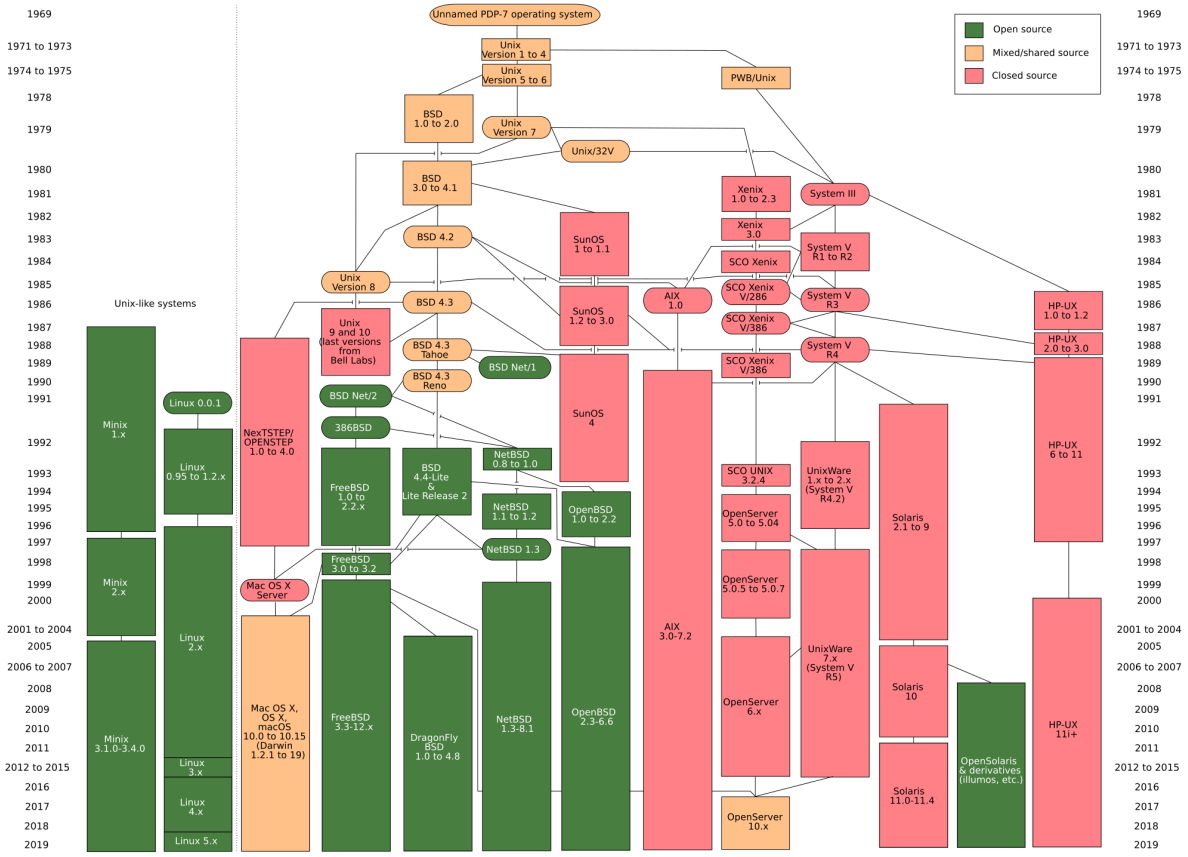


FIGURE 1.3 – Arbre *généalogique* Unix/Linux (image Wikipédia)

— Écrit à l'origine en assembleur,
— Le langage C a été créé pour Unix et Unix a été réécrit en C pour plus de portabilité,
— A l'origine de la norme Posix,
— Permet la communication entre machine,
— Système multi-utilisateurs,
— Système multi-tâches,
— Système de développement : beaucoup d'outils disponibles de base,
— Dispose d'interpréteurs de commandes puissants (shell),
— ...

1.4.2 Windows

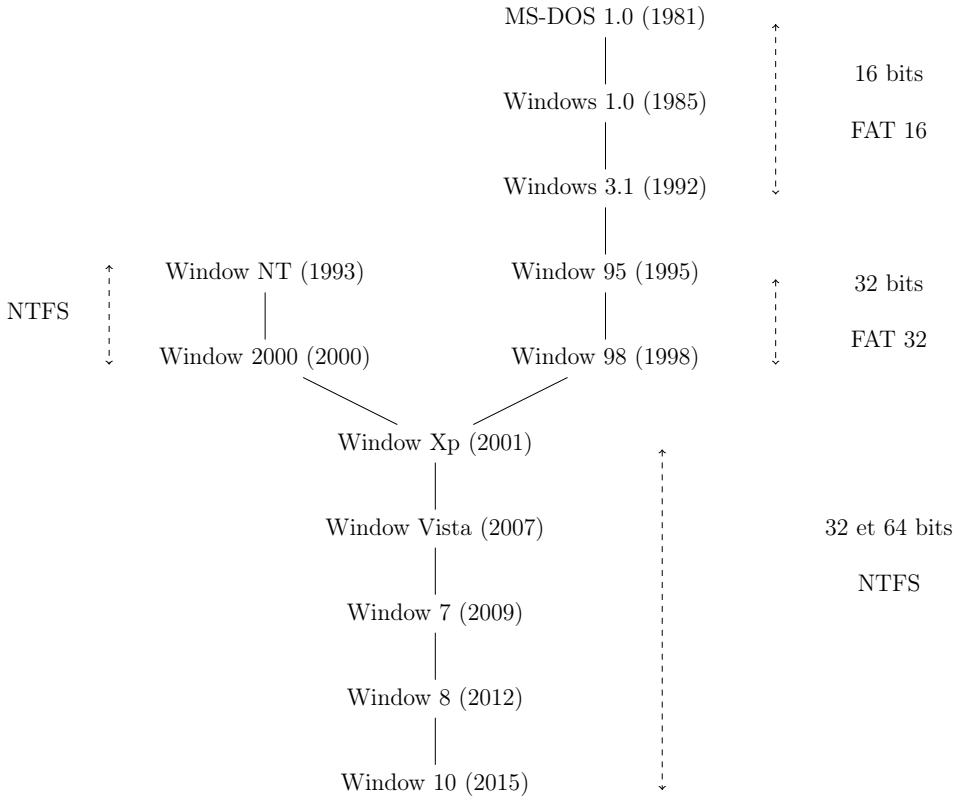


FIGURE 1.4 – Évolution des versions de Windows

Chapitre 2

Allocation du processeur central

Les stratégies d'allocation de l'unité centrale ont en général pour but de trouver un équilibre entre la durée d'utilisation des ressources par les processus et le temps qu'ils mettent pour s'exécuter.

L'objectif est de satisfaire les demandes d'exécution des programmes en respectant certaines contraintes parmi les suivantes :

— garantir un temps d'attente fini et une durée minimale d'allocation,
— respecter certaines priorités entre les processus,
— garantir l'allocation avant une date fixée,
— arrêter un processus dépassant une durée donnée.

2.1 Stratégies d'allocation

La figure 2.1 montre les principales étapes de l'allocation d'un ou plusieurs processeurs à des processus. Les processus circulent selon le schéma suivant :

1. entrée des processus demandeurs,
2. choix d'un processus en attente,
3. choix d'un processeur à allouer ;
4. interruption de service.

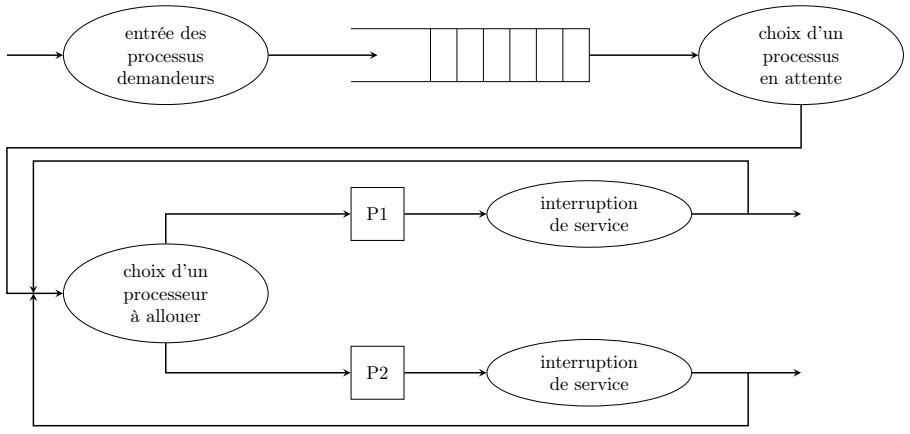


FIGURE 2.1 – Etapes d'allocation d'un processeur à un processus.

Dans la suite nous allons détailler les stratégies utilisées à l'étape 2 : choix d'un processus en attente.

2.1.1 Stratégies sans recyclage des demandes

L'allocation de l'unité centrale est effectuée pour une demande complète, il n'y a pas de préemption (suspension de l'exécution au profit d'un autre processus). La stratégie FIFO est simple et *a priori*

efficace :

— FIFO impératif : on mélange les petits et les gros demandeurs, ce qui défavorise les petits ;
— FIFO indicatif : la file est triée sur un critère, on fait alors des insertions en milieu de file. l'efficacité est raisonnable, mais il y a pénalisation des gros demandeurs ;
— FIFO à priorités : plusieurs files sont gérées en FIFO. Les files sont classées par priorité, on ne choisit un demandeur dans une file que si les files de priorité supérieure sont vides (figure 2.2).

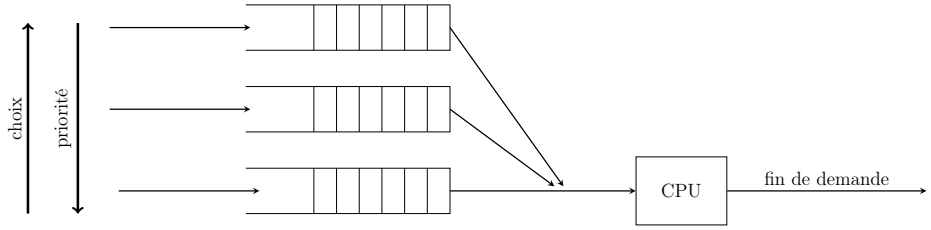


FIGURE 2.2 – Stratégie sans recyclage des demandes, FIFO à priorités.

2.1.2 Stratégies avec recyclage

Les stratégies avec recyclage effectuent l'allocation de l'unité centrale par tranche de temps appelée quantum, les demandes ne sont pas servies globalement, mais par fraction.

Le choix de la durée du quantum est fondamental :

— un quantum très court favorise les traitements interactifs,
— un quantum long augmente le débit global du système.

2.1.2.1 La stratégie du tourniquet

La stratégie du tourniquet (Round Robin) effectue l'allocation de l'unité centrale par quantum (figure 2.3). Le processeur est alloué successivement à chaque processus ; si au bout du quantum alloué, le processus ne s'est pas terminé, il est interrompu et placé en queue de la file des demandeurs.

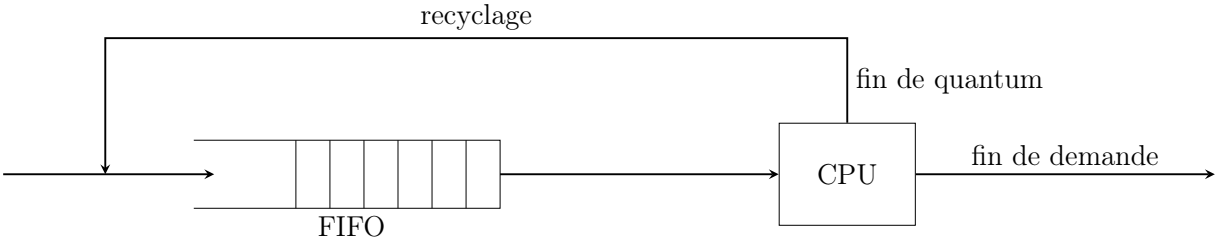


FIGURE 2.3 – Stratégie avec recyclage par tourniquet.

2.1.2.2 La stratégie du tourniquet à deux niveaux

La stratégie du tourniquet à deux niveaux : par une étude de la charge sur un système de temps partagé (IBM CP/CMS), on a constaté deux catégories de demandes :

— les demandes courtes à caractère fortement interactif : on leur alloue un quantum de 20 ms,
— les demandes longues, correspondant aux travaux différés : on leur alloue un quantum de 100 ms (figure 2.4).

Le recyclage provoque :

— une baisse de la priorité réalisée par un changement de file,
— une allocation plus longue.

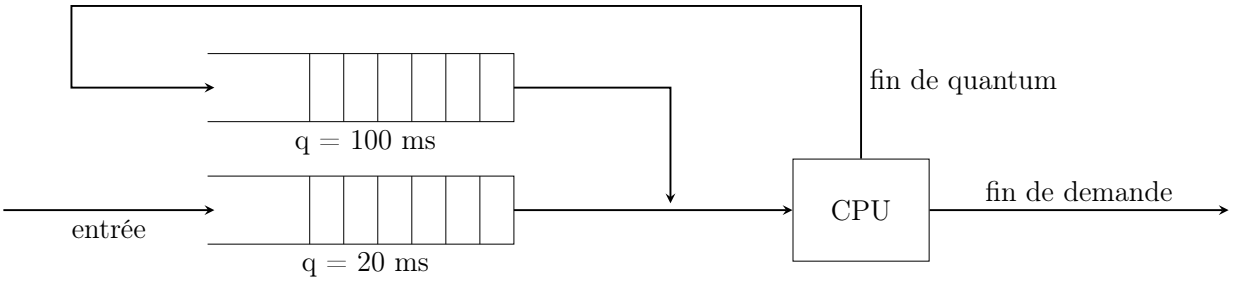


FIGURE 2.4 – Stratégie par recyclage tourniquet multi-niveau.

2.1.2.3 Unix

Dans le système Unix, la stratégie fait appel à un tourniquet multiniveau à quantum unique (figure 2.5) :

- à chaque processus prêt est affectée une priorité flottante. Cette priorité reflète la consommation et la demande en ressources du processus,
- à chaque niveau de priorité correspond une file desservie selon une politique FIFO,
- le tourniquet s'applique sur chaque file,
- la durée du quantum est de 100 ms. Cette durée a été déterminée empiriquement et correspond à la durée maximale ne dégradant pas les travaux interactifs.

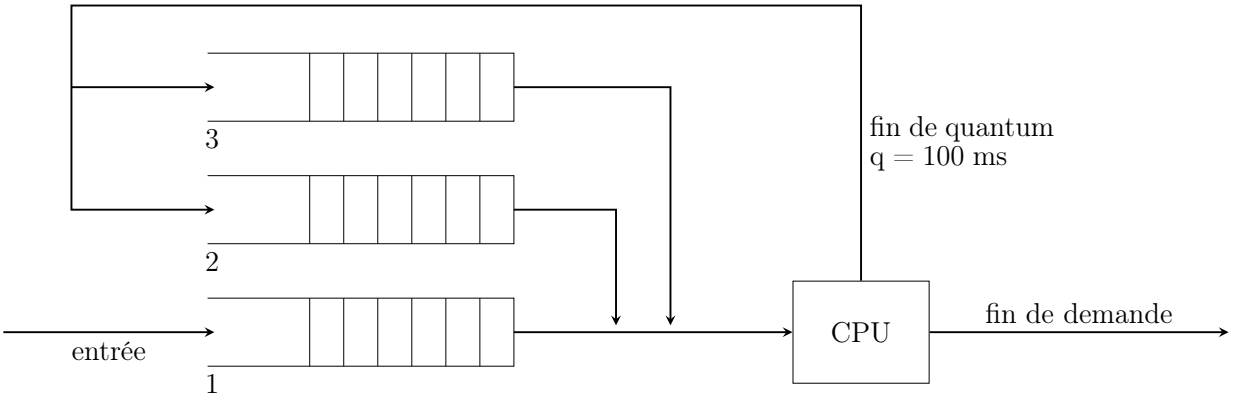
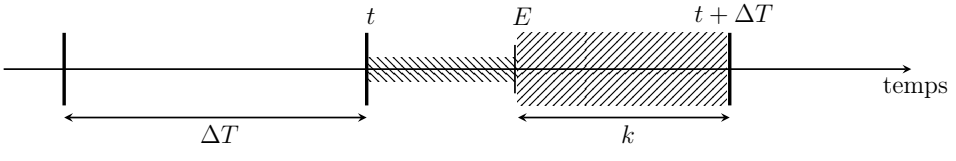


FIGURE 2.5 – Stratégie par recyclage tourniquet multi-niveau Unix BSD.

2.1.3 Stratégies avec échéance : Deadline

Elles sont essentiellement utilisées dans les systèmes temps réel pour le contrôle de processus industriels. L'exemple ci-dessous, de correction d'un satellite, explique l'utilisation de ce type de stratégie.

- Les coordonnées du satellite sont envoyées au calculateur à intervalles de temps réguliers ΔT.
- Le traitement d'une position du satellite dure k secondes et doit être impérativement achevé avant l'arrivée des coordonnées de la position suivante.
- Lors de l'arrivée des coordonnées d'une position X à l'instant t, on détermine l'échéance E comme la date $t + \Delta T - k$.
- Si le traitement des coordonnées de cette position X n'est pas commencé au temps E, le traitement n'est pas ordonnancé et ne sera pas effectué.



2.2 Allocation de l'unité centrale dans le système Unix BSD

Ce paragraphe présente en détail la gestion des processus dans la version HP-UX du système Unix BSD 4.3

2.2.1 Les états d'un processus

L'ordonnanceur, qui gère l'ordre d'affectation du processeur à tâches, manipule des processus. Ces processus sont une instance d'un programme chargé en mémoire pour exécuter une tâche. L'ordonnanceur a le pouvoir d'activer, suspendre ou stoper un processus.

Ainsi, au cours de leur vie, les processus peuvent se trouver dans différents états ; le graphe de la figure 2.6 les présente avec les transitions possibles. Les processus se trouvant dans l'état *Runnable* sont regroupés dans les files d'attente gérées par l'ordonnanceur.

Les principaux états sont :

— *Idle* : le processus en cours de création (embryonnaire), et ne peut s'activer ;
— *Runnable* ou *Prêt* : le processus attend son tour pour utiliser le processeur ;
— *Running* ou *Actif* : le processus utilise le processeur, c'est son tour ;
— *Zombie* : le processus est en cours de destruction, il ne peut plus s'activer.

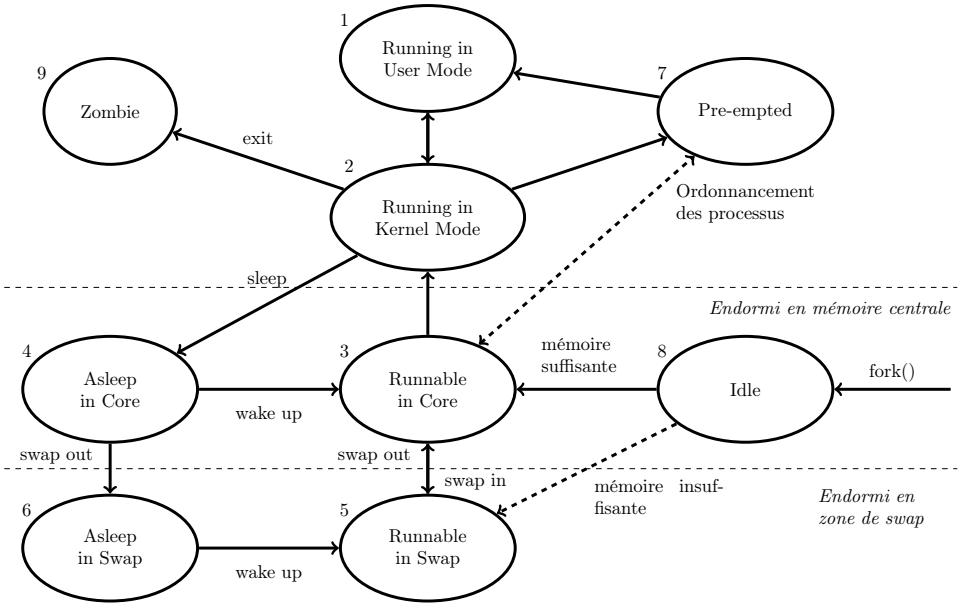


FIGURE 2.6 – Diagramme des états et transitions possibles pour un processus.

Remarque : Dans la figure 2.6, l'état *Running* se compose de deux sous-états ; *Running_ user_ mode* et *Running_ kernel_ mode*.

Tout lancement, soit initial, soit après un changement de contexte d'un processus, passe obligatoirement par le sous-état *Running_ kernel_ mode*.

2.2.2 L'ordonnancement des processus

2.2.2.1 Principes

L'unité d'allocation du temps unité centrale est le quantum et la stratégie utilisée est le tourniquet multi-niveaux géré en FIFO.

Le critère du choix pour l'ordonnancement est la priorité flottante attribuée à chaque processus. Le calcul de cette priorité respecte la règle suivante (fig. 2.7) :

— la priorité augmente avant le service
— la priorité diminue fortement après le service

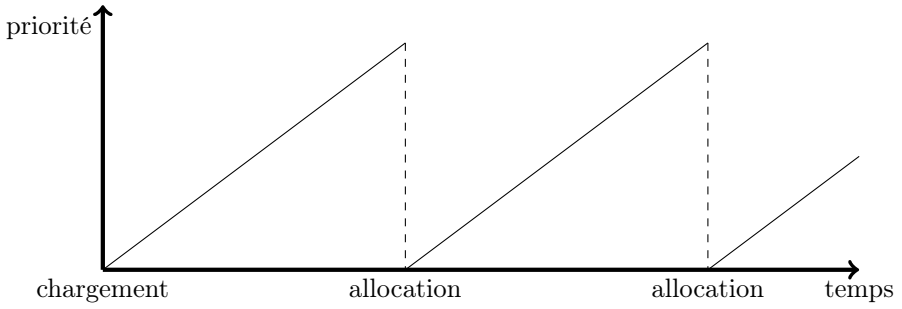


FIGURE 2.7 – Évolution de la priorité d'un processus au cours du temps.

2.2.2.2 Structuration de l'ordonnanceur

Deux familles de fonctions noyau assurent la gestion des processus :
- les fonctions explicitement appelées par les processus pour leur synchronisation et leur communication.
- les fonctions appelées par les activités liées au traitement des interruptions horloge. Cet ensemble de fonctions constitue l'ordonnanceur :
  - `schedcpu` est activée toutes les secondes et recalcule la priorité flottante des processus selon la formule 2.1 p. 16
  - `roundrobin` est activée 10 fois par seconde et assure le temps partagé en allouant à chaque processus un quantum de 100 ms et en faisant circuler les processus dans les files
  - `hardclock` est activée 100 fois par seconde et met à jour la variable p_cpu comptabilisant la consommation en temps unité centrale du processus
  - `setpri` est activée toutes les 40 ms et recalcule la priorité flottante des processus selon la formule 2.2 p. 16
  - `switch` recherche le processus le plus prioritaire et opère le changement de contexte pour son activation.

2.2.2.3 Organisation des files d'attente dans le système HP-UX

Les files d'attente du système HP-UX regroupent l'ensemble des processus prêts ou en attente. A chaque niveau de priorité flottante, depuis 0 jusqu'à 255, correspond une file d'attente.

Les processus en attente sont rangés dans les files correspondant à leur priorité de réveil. Cette priorité est supérieur à celle des processus utilisateurs prêts car au réveil le processus exécute en général une fonction système prioritaire. Cette priorité d'attente est totalement différente de la priorité des processus utilisateurs prêts gérés par l'ordonnanceur.

Les files d'attente sont structurées selon la figure 2.8.

1. Priorité 0–127 : priorités des processus temps réel, ces priorités sont fixes, une file est associée à chaque priorité et les processus sur la file sont gérés en tourniquet **sans** temps partagé.
2. Priorité 128–177 : priorités des processus en attente sur évènement dans l'état *Asleep*. La priorité est fixe durant toute l'attente et dépend de la nature de l'attende :
   - priorité 128–152 : priorités des processus insensibles aux signaux : ils ne seront réveillés que par l'arrivée de l'évènement attendu,
   - priorité 153–177 : priorités des processus sensibles aux signaux : ils sont réveillés par l'évènement attendu ou par un signal quelconque.
3. Priorités 178–255 : priorités des processus utilisateurs s'exécutant en temps partagé. Cette priorité est recalculée toutes les secondes à partir de la priorité de base (PUSER=178) et du temps cpu consommé.

   L'ensemble de ces files constitue la *run queue*, file des processus prêts, scrutée par l'ordonnanceur pour le choix du processus à activer

   La préemption est effectuée au profit d'un processus plus prioritaire :
   - à la sortie du mode *kernel*, si un processus plus prioritaire a été réveillé,

— en fin de quantum, lors de la ré-actualisation des priorités flottantes.

Remarque : Lorsqu'un processus n'a pas consommé tout son quantum, par exemple par suite d'une mise en attende de fin d'entrée/sortie, le temps restant est attribué au nouveau processus activé. Ainsi, ce processus ne reçoit pas un quantum entier pour son exécution.

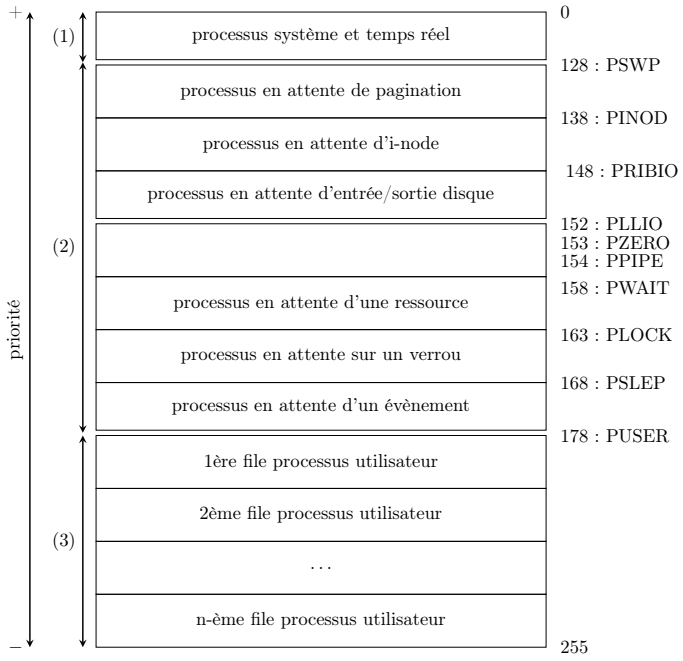


FIGURE 2.8 – Description des files d'attentes de la stratégie tourniquet sous HP-UX.

2.2.3 Algorithme simplifié de l'ordonnanceur

Pour tous les processus appartenant à la *runqueue* :

1. choisir le processus résidant en mémoire centrale ayant la plus forte priorité,
2. ôter le processus de la *runqueue* et effectuer un changement de contexte pour l'exécuter,
3. si aucun processus n'est éligible, mise de la machine dans l'état *idle*.

2.2.4 Principe du calcul de la priorité

— Pour calculer la priorité flottante des processus prêts, l'ordonnanceur prend en compte deux paramètres :
  — le temps CPU précédemment consommé par le processus et comptabilisé dans la variable *p_ cpu*,
  — le temps d'attente passé dans la *runqueue*.
— La priorité des processus en attente est fonction :
  — des caractéristiques de l'évènement ou de la ressource attendu,
  — des ressources bloquées par le processus en attente.
— La priorité de réactivation des processus après une attente est calculée en utilisant la durée de l'attente comptabilisée dans la variable *p_ slptime*.

2.2.5 Dans le système Unix BSD

2.2.5.1 Règles de calcul de la priorité flottante

L'estimation de la charge utilise la variable *load* qui mémorise la longueur de la file d'attente des processus prêts.

La variable *p_ nice* permet à l'utilisateur de moduler sa priorité.

La variable p_cpu du processus actif est incrémentée à chaque interruption horloge, appelée tick (toutes les 10 ms).

Toutes les secondes, la variable p_cpu est ré-ajustée en utilisant une fonction de filtrage prenant en compte un estimateur de la charge du système.

$$p\_cpu = \Big(\frac{2\ load}{2\ load + 1}\Big) p\_cpu + p\_nice \tag{2.1}$$

La priorité flottante est recalculée toutes les 40 ms (après 4 ticks d'horloge) selon la formule suivante :

$$p\_usrpri = PUSER + \frac{p\_cpu}{4} + 2\ p\_nice \tag{2.2}$$

Cette formule provoque une baisse linéaire de la priorité selon le temps unité centrale consommé.

Lors de la réactivation d'un processus utilisateur en attente dans l'état *endormi*, l'ordonnanceur calcule la valeur de la variable p_cpu à l'aide de la formule suivante :

$$p\_cpu = p\_cpu\ \Big(\frac{2\ load}{2\ load + 1}\Big)^{p\_slptime} \tag{2.3}$$

Le système comptabilise le temps d'attente dans la variable $p\_slptime$ qui est réinitialisée à zéro lors de la mise en attente.

Le priorité flottante est directement liée au temps unité centrale précédemment consommé. La fonction de filtrage a pour but d'estomper le passé lointain de la consommation en temps unité centrale. L'exemple ci-après permet d'illustrer l'effet de ce filtre.

Exemple :

— Un seul processus est actif et consomme toute l'unité centrale : la variable *load* vaut 1,

— Ce processus accumule T_i ticks à la fréquence de l'horloge pendant la durée i,

— Toutes les secondes, le filtre est appliqué avec la formule suivante :

$$p\_cpu = 0.66\ p\_cpu + p\_nice \tag{2.4}$$

— On suppose que p_nice vaut 0.

Simulation de l'évolution de la priorité :

$$\begin{aligned}
p_cpu &= 0.66\ T_0 \\
p_cpu &= 0.66\ (T_1 + 0.66\ T_0) = 0.66\ T_1 + 0.44\ T_0 \\
p_cpu &= 0.66\ T_2 + 0.44\ T_1 + 0.30\ T_0 \\
p_cpu &= 0.66\ T_3 + \ldots + 0.20\ T_0 \\
p_cpu &= 0.66\ T_4 + \ldots + 0.13\ T_0
\end{aligned}$$

Après cinq application du filtre (après 5 secondes), près de 90% du temps unité centrale consommé lors du premier quantum a été "*oublié*".

2.2.5.2 Implantation de la run queue

L'ensemble des processus dans l'état *Runnable* constituent la file des processus prêts : la run queue. Dans le système Unix BSD, l'implantation du tourniquet multi-niveau est réalisée ainsi :

— les niveaux de priorité flottante sont regroupés par 4,

— une liste chaînée des processus est associée à chaque groupe de priorité flottante,

— la table des têtes et queues de listes des files est implantée sous forme d'un tableau appelé *qs*. Une seconde table de booléens, appelée *whichqs*, est associée à la table *qs* pour indiquer l'occupation de chaque file (figure 2.9).

En mode noyau, le processus ne peut être en préemption uniquement en sortie de ce mode. De même, si il fait une demande de commutation pendant son passage kernel, elle sera réalisée à la fin du mode kernel (positionne une variable globale).

De même si une interruption de préemption arrive pendant l'activité en mode kernel, elle ne sera prise en compte qu'à la fin (positionne une demande de déroutement AST).

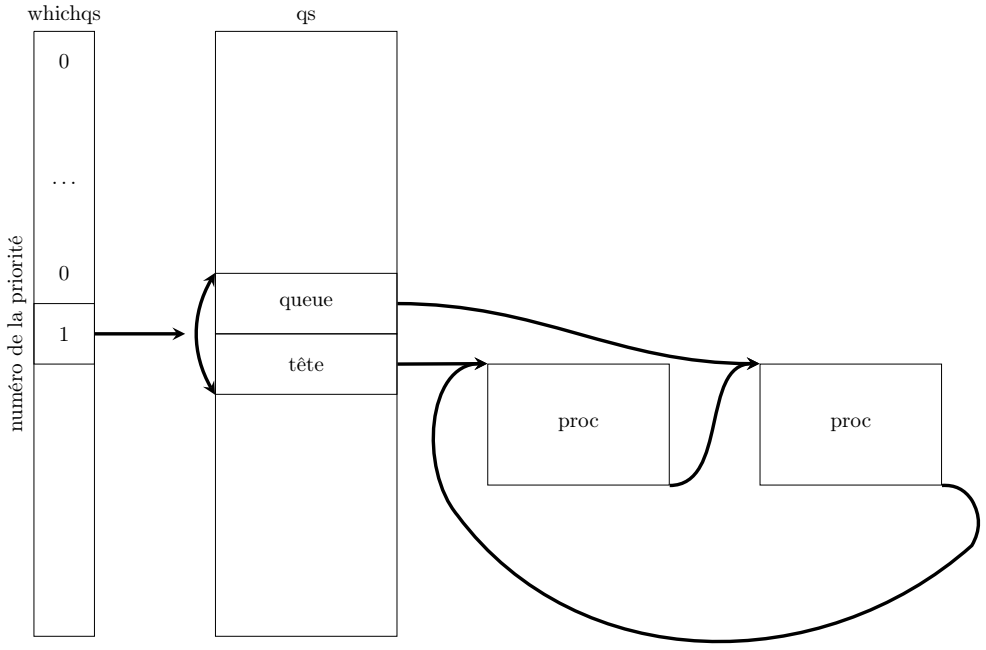


FIGURE 2.9 – Organisation de la *run queue*.

2.2.6 Priorité flottante dans Unix System V

A la fin de chaque quantum T_i on recalcule la consommation de temps CPU dans la variable CPU_usage selon la formule suivante :

$$CPU\_usage(T_i) = \frac{CPU\_usage(T_{i-1}) + nb(T_i)}{2}$$

où $nb(T_i)$ représente le nombre de ticks CPU accumulés pendant la durée T_i.

Un tick est comptabilisé à un processus si l'interruption horloge arrive lorsque le processus s'exécute. Le calcul de la nouvelle priorité se fait selon cette formule :

$$Priorité(T_i) = P\_USER + \frac{CPU\_usage(T_i)}{2}$$

2.2.6.1 Exemple

Soit le contexte suivant :
— 3 processus A, B et C dans la même file de priorité 178,
— un quantum dure 100 ticks d'horloge,
— le recalcul de la priorité se fait en fin de quantum,
— les processus ne font pas d'appels noyau,
— aucun autre processus ne se trouve dans l'état *Runnable*.

2.3 Les structures de données décrivant les processus

Deux tables décrivent les processus : la table `proc`, décrite par la structure `proc.h` et la table `user` décrite par la structure `user.h`.

Les structures `proc` et `user` d'un processus sont au même indice dans les deux tables.

La table proc est une table dimensionnée à nproc entrées, elle possède une entrée par processus. La structure proc résident en mémoire et contient essentiellement :
— Le chaînage vers les autres entrées de la table proc pour matérialiser les files d'attente :
  — La file des processus de même priorité,

	Processus A			Processus B			Processus C		
Quanta	Priorité	CPU_usage	ticks	Priorité	CPU_usage	ticks	Priorité	CPU_usage	ticks
T_1	**exécution 178**	0 (P_USER)	1 . **100**	178 (P_USER)	0	0 . .	178 (P_USER)	0	0 . .
T_2	203	50 ($= \frac{0+100}{2}$)	0 . .	**exécution 178**	0	1 . **100**	178	0	0 . .
T_3	190	25 ($= \frac{50+0}{2}$)	0 . .	203	50	0 . .	**exécution 178**	0	1 . **100**
T_4	**exécution 184**	12 ($= \frac{25+0}{2}$)	1 . **100**	190	25	0 . .	203	50	0 . .
T_5	206	56 ($= \frac{12+100}{2}$)	0 . .	**exécution 184**	12	1 . **100**	190	25	0 . .

— La liste des processus partageant le même texte exécutable,
— La liste des processus prêts (run queue).

— Les identificateurs associés au processus,
— Les variables associées à la gestion des priorités,
— Les variables associées à la gestion du temps,
— Les variables associées à la gestion des signaux,
— Les liens vers d'autres structures du système.

La table user possède une entrée par processus présent en mémoire. Elle accompagne les processus durant leurs va-et-vient entre la mémoire centrale et la mémoire secondaire. La structure user contient essentiellement :

— Les pointeurs vers les zones de sauvegarde et l'entrée proc du processus,
— Le nom du processus,
— Les identificateurs associés au processus,
— Les informations liées à la gestion mémoire,
— Les informations liées à la gestion des signaux,
— Les descripteurs de fichiers ouverts,
— Les noms des chemins (`pathname`) du répertoire de travail (`working directory`) et du répertoire personnel (`home directory`).

Le schéma d'organisation des tables est présenté figure 2.10.

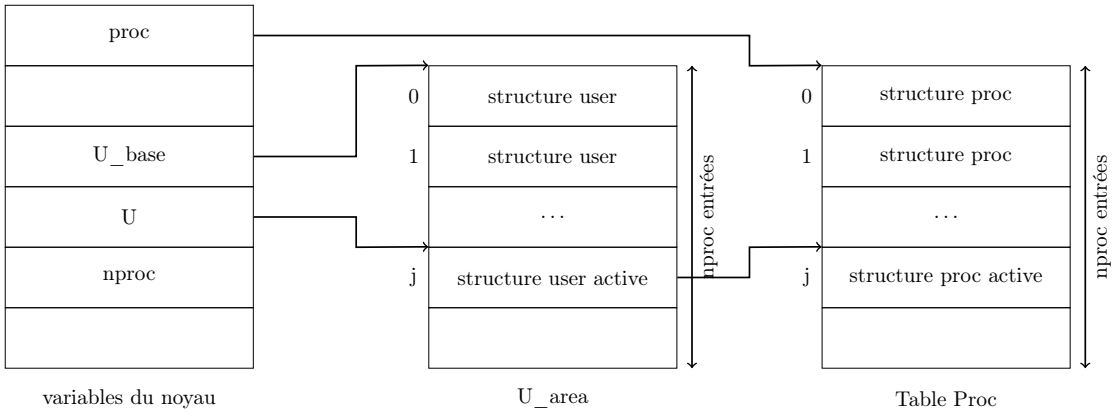


FIGURE 2.10 – Organisation des tables proc et user

2.4 Filiation des processus

Le mécanisme de création des processus implique l'héritage des attributs. La connaissance de l'arbre de filiation (fig. 2.11) des processus est fondamentale pour le système.

La première structure proc de la table proc correspond au processus ancêtre de tous les autres : le processus `Init`. Chaque structure proc possède les pointeurs nécessaire à la description de la filiation :

— Le pointeur sur le fils le plus récemment créé,
— Le pointeur sur le père,
— Le pointeur sur le cadet (frère plus jeune),
— Le pointeur sur l'aîné (frère plus vieux, mais pas le plus vieux).

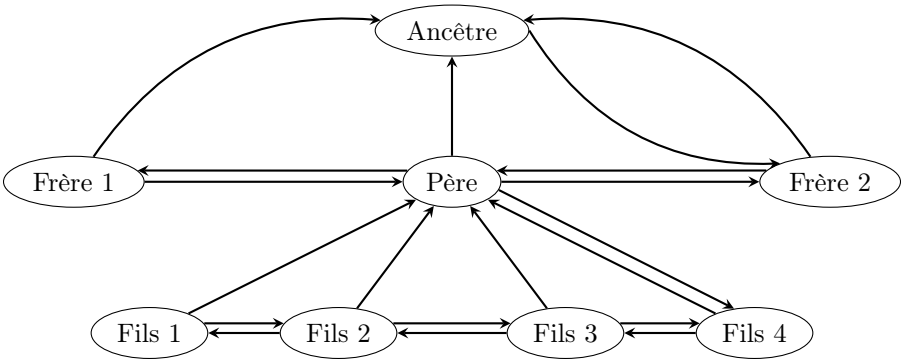


FIGURE 2.11 – Exemple de filiation de processus

Chapitre 3

Allocation de la mémoire centrale

3.1 Les mémoires

L'espace d'adressage d'un processeur est l'ensemble des objets auxquels les instructions des programmes qui s'y exécutent peuvent accéder. La taille de l'espace est fonction de la longueur du champs adresse de l'instruction.

Deux stratégies différentes peuvent être mises en place pour gérer l'espace d'adressage d'un processeur :

— Utiliser un espace d'adressage réel correspondant à la mémoire physique implantée. La mémoire est alors organisée linéairement et est gérée comme un vecteur d'emplacements. Dans ce cas, l'unité centrale manipule des adresses physiques qu'elle envoie directement à la mémoire.

— Utiliser un espace d'adressage virtuel correspondant à une hiérarchie de mémoire de tailles et de temps d'accès différents, par exemple la mémoire centrale et le disque. Dans ce cas, l'unité centrale manipule des adresses virtuelles qu'elle envoie à une unité de traduction d'adresse appelée MMU [1] qui traduit les adresses virtuelles en adresses réelles qu'elle émet vers la mémoire physique.

3.1.1 Mémoire classique *versus* mémoire associative

3.1.1.1 Mémoire *classique*

Une mémoire est la réalisation matérielle d'une fonction qui associe une valeur à une adresse. Par exemple, une mémoire de type 2D peut être vue comme un vecteur. Les composantes du vecteur sont appelées mots. L'indice de chaque mot constitue son adresse.

3.1.1.2 Mémoire associative

Une mémoire associative [2] fonctionne différemment. Elle implante une fonction que l'on peut voir comme l'inverse de celle d'une mémoire classique : elle retourne un prédit d'existence binaire d'une valeur d'interrogation. Le prédicat d'existence sera affecté ainsi :

— Vrai, si la valeur existe en mémoire ;

— Faux, si elle n'existe pas.

La valeur est recherchée dans l'ensemble des mots de la mémoire associative en la comparant avec chaque mot. Chacune de ces comparaisons définit un prédicat binaire d'existence, (d'égalité ou de contenance) : si un mot contient la valeur alors son prédicat est 1.

Remarque : Ce mécanisme élémentaire peut être complété par un mécanisme d'adressage : le vecteur composé par l'ensemble des prédicats est utilisé comme adresse dans une mémoire classique.

3.1.1.3 Exemple d'utilisation des mémoires associatives

On désire stocker sous forme de tableau les valeurs d'une fonction discontinue $y = f(x)$, c'est-à-dire une fonction présentant de nombreux trous dans le domaine de définition de la variable x. On peut

1. Memory Management Unit
2. Content Adressable Memory

implanter ce tableau de deux façons :

— Implantation avec une mémoire classique
La valeur de la variable x est prise comme adresse dans une mémoire bornée par les extrema du domaine de définition. A un mot d'indice x correspond la valeur $f(x)$.
Cette implantation est simple mais peu efficace puisqu'elle engendre un gâchis de mémoire important à cause des nombreux trous dans le domaine de définition de la variable x.

— Implantation avec une mémoire associative, complétée par une mémoire classique
La mémoire associative ne contient que les valeurs pour lesquelles la variable x est définie, la mémoire classique contient les valeurs correspondantes de la fonction $f(x)$. Le lien entre les mémoires deux mémoires est effectué par un adressage utilisant le prédicat d'existence : le rang du mot dans la mémoire associative ayant son prédicat positionné à vrai est utilisé comme adresse pour la mémoire classique.

Remarque : La mise en tableau des valeurs d'une fonction est souvent utilisée pour accélérer les calculs d'une fonction fréquemment employée. Par exemple, on défini généralement une table de la fonction cosinus dans les programmes géométriques intensifs (moteurs de rendu).

3.1.1.4 Implantation des mémoires associatives

La technologie utilisée pour la réalisation des mémoires associatives dépend de leur taille :

— Pour des mémoires de petite taille réalisant la recherche sur un champs réduit, la réalisation est purement électronique avec une comparateur par mot mémoire ;

— Pour des mémoires de taille intermédiaire, la recherche est effectuée par des algorithmes microprogrammés ;

— Pour de grandes mémoires, les techniques de rangement calculé utilisant les fonctions de hashage sont utilisées.

3.2 La mémoire physique

La mémoire physique est un élément clé dans la composition d'un ordinateur : elle contient les programmes, leurs données initiales, les variables et toutes sortes de données manipulées constamment par le processeur. Aussi, elle est sollicitée en permanence et sa réalisation physique aura un impact important sur son efficacité et au final sur les performances du système entier.

Un autre aspect à prendre en compte est aussi le prix. En effet, plus la technologie mise en place sera performante et plus son prix sera lui aussi élevé.

Ainsi, plusieurs organisations et techniques ont été proposées pour faire un compromis entre les performances, la réalisation technique possible et le coût de réalisation.

3.2.1 Manipulations sur l'organisation des adresses

L'objectif de ces manipulations est de privilégier soit l'extensibilité par l'ajout de modules mémoire à une mémoire existante, soit les performances par la parallélisation des accès à la mémoire.

3.2.1.1 Découpage en modules

Un module est un bloc de mémoire contenant un nombre fixe d'emplacements situés à des adresses contiguës. Les modules sont désignés par leur numéro ou rang et sont placés dans l'espace d'adressage de façon à assurer la continuité des adresses.

Le champs adresse pour une mémoire organisée en module est découpée en deux parties :

— Les poids forts de l'adresse désignent le numéro du module ;

— Les poids faibles de l'adresse permettent la sélection d'un mot dans le module.

La figure 3.1 illustre ce type d'organisation pour une mémoire ayant un champ "numéro de module" codé sur n bits et un champs "déplacement dans le module" codé sur k bits. La mémoire est découpée en 2^n modules numérotés de 0 à $2^n - 1$, la taille d'un module est de 2^k mots et le déplacement dans un module est compris entre 0 et $2^k - 1$. Ainsi :

— Le module numéro 0 contiendra les mots d'adresses : $0, 1, 2, 3, \ldots, 2^k - 1$

— Le module numéro 1 contiendra les mots d'adresses : $2^k, 2^k+1, 2^k+2, 2^k+3, \ldots, 2\times 2^k-1$
— Le module numéro 2 contiendra les mots d'adresses : $2\times 2^k, 2\times 2^k+1, 2\times 2^k+3, \ldots, 3\times 2^k-1$
— Le module numéro $n-1$ contiendra les mots d'adresses : $(2^n-1)\times 2^k, (2^n-1)\times 2^k+1, (2^n-1)\times 2^k+2, (2^n-1)\times 2^k+3, \ldots, 2^n\times 2^k-1$

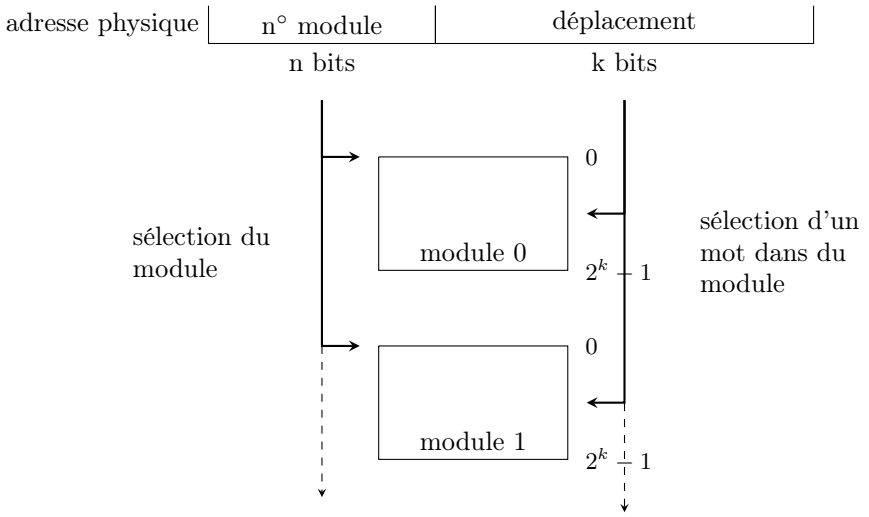


FIGURE 3.1 – Exemple de découpage d'une mémoire en modules.

Avantage : L'extension est aisée puisque la mémoire est découpée en modules indépendants dont les adresses dans l'espace d'adressage sont contiguës.

Inconvénient : Aucun parallélisme n'est possible pour accélérer les accès en utilisant la propriété de localité car le voisinage proche d'une adresse se trouve dans le même module.

3.2.1.2 Découpage en bancs entrelacés

Un banc est un bloc de mémoire contenant un nombre fixe d'emplacements situés à des adresses contiguës modulo le nombre de bancs.

Le champ adresse pour une mémoire organisée en bancs entrelacés [3] est découpé en deux parties :
— Les poids forts de l'adresse permettent la sélection d'un mot dans le banc ;
— Les poids faibles de l'adresse désignent le numéro du banc.

L'exemple de la figure 3.2 illustre ce type d'organisation pour une mémoire de 64 Kmots organisée en quatre bancs entrelacés. Le champ "numéro de banc" est codé sur 2 bits et le champ "déplacement dans le banc" est codé sur 14 bits. Ainsi :
— Le banc numéro 0 contiendra les mots d'adresse $0, 4, 8, \ldots, 65532$
— Le banc numéro 1 contiendra les mots d'adresse $1, 5, 9, \ldots, 645533$
— Le banc numéro 2 contiendra les mots d'adresse $2, 6, 10, \ldots, 645534$
— Le banc numéro 3 contiendra les mots d'adresse $3, 7, 11, \ldots, 645535$

Avantage : Ce découpage permet d'exploiter la propriété de localité pour paralléliser les accès : deux adresses consécutives dans l'espace d'adressage du processeur sont situées dans deux bancs différents qui peuvent être accédés simultanément.

Inconvénient : La mémoire n'est pas extensible.

3. Interleaved banks

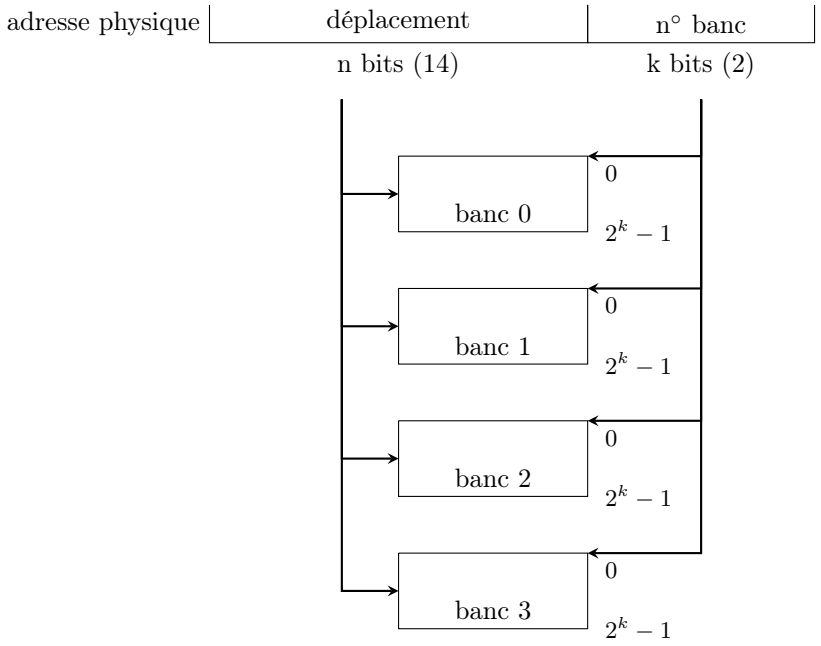


FIGURE 3.2 – Exemple d'un découpage d'une mémoire en 4 bancs entrelacés.

3.2.1.3 Compromis

Le compromis fréquemment mis en oeuvre consiste à découper la mémoire en module, chaque module contenant des bancs entrelacés.

La figure 3.3 présente cette solution utilisée pour bénéficier du découpage en bancs entrelacés sans être pénalisé par l'impossibilité d'étendre la mémoire.

L'adressage s'effectue d'abord par sélection du module, puis du banc et enfin du mot.

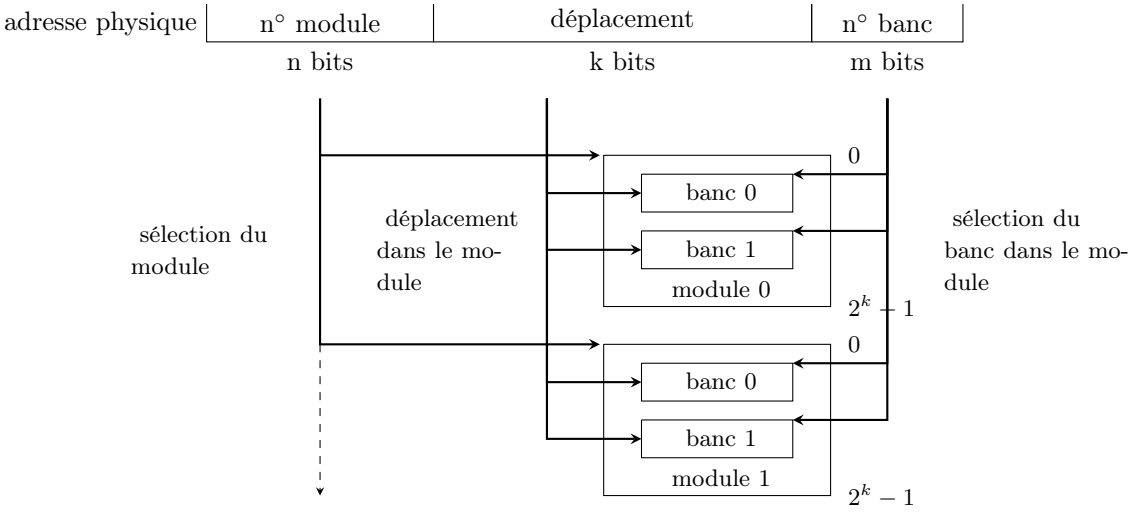


FIGURE 3.3 – Compromis de découpage de la mémoire : bancs entrelacés dans des modules.

3.2.2 Manipulation sur les longueurs des mots

La longueur des mots stockés dans un emplacement mémoire ne sont pas forcément de la même longueur qu'un mot manipulé par le processeur. En effet, le processeur peut manipuler au cours de la même exécution d'un programme aussi des entiers codés sur un octet que des nombres à virgule flottante

codés sur 8 octets. Or, les deux types de nombre seront stockés dans la même mémoire. C'est pourquoi on distingue 2 types de mots mémoires : les mots physiques et les mots logiques.

Définitions :

— **Un mot physique** est le contenu de l'emplacement mémoire désigné par une adresse physique. Sa longueur est fixe et dépend des choix de réalisation du composant.

— **Un mot logique** est la quantité d'information manipulée par les instructions. Sa longueur peut être variable et va dépendre du type de donnée manipulée.

Comme indiqué précédemmant, la taille du mot physique ne correspond pas forcément à celle du mot logique. Comme on ne définit pas différentes mémoires ayant chacunes des mots physiques de longueur différentes, il faut trouver une autre solution qui nécessitera cette fois encore des compromis entre performances, complexité de réalisation et coût.

3.2.2.1 Parallélisation des accès

Dans ce type d'organisation, un mot physique contient plusieurs mots logiques. Utilisant la propriété de localité, cette organisation permet d'anticiper les accès mais elle augmente fortement le coût de la mémoire et de sa logique d'accès.

La figure 3.4 présente le principe de l'adressage avec un découpage de l'adresse en deux champs :

— Les poids forts contiennent l'adresse physique ;

— Les poids faibles contiennent un pointeur sur le mot logique manipulé.

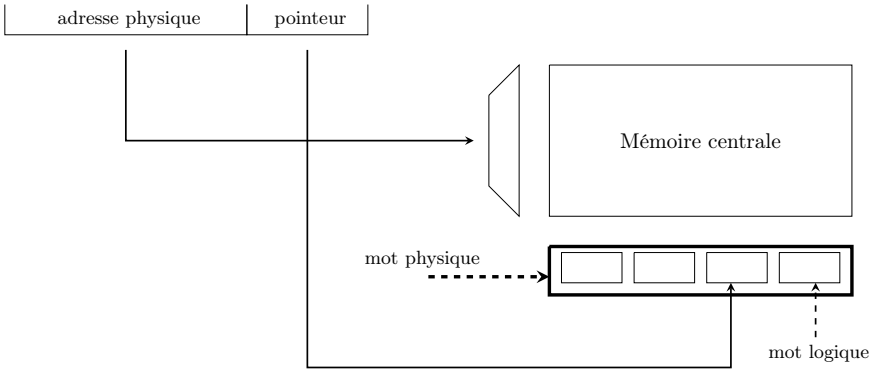


FIGURE 3.4 – Parallélisation des accès aux mots logiques.

Dans l'exemple présenté, un accès à la mémoire fournit quatre mots logiques. Pour des raisons liées au codage binaire des adresses, le nombre de mots logiques contenus dans un mot physique est une puissance de 2.

3.2.2.2 Sérialisation des accès

Dans le but de diminuer le coût de réalisation de la mémoire, on prend l'option inverse : un mot mémoire physique ne contient qu'une fraction d'un mot logique.

Plusieurs accès en série à la mémoire centrale, pour lire plusieurs mots physiques contigus, permettent de recomposer un mot logique. Cette organisation permet d'obtenir des mémoires et des processeurs économiques au prix d'un ralentissement des accès : il faut plusieurs cycles pour obtenir le mot logique. Le processeur 16 bits Intel 8088 accède à la mémoire selon ce principe avec un bus de données sur 8 bits.

La figure 3.5 décrit le principe de l'adressage avec un découpage de l'adresse en deux champs :

— Les poids forts contiennent l'adresse physique ;

— Les poids faibles contiennent un compteur sur la fraction du mot logique manipulé.

Dans l'exemple présenté, deux accès à la mémoire sont nécessaires pour reconstruire un mot logique.

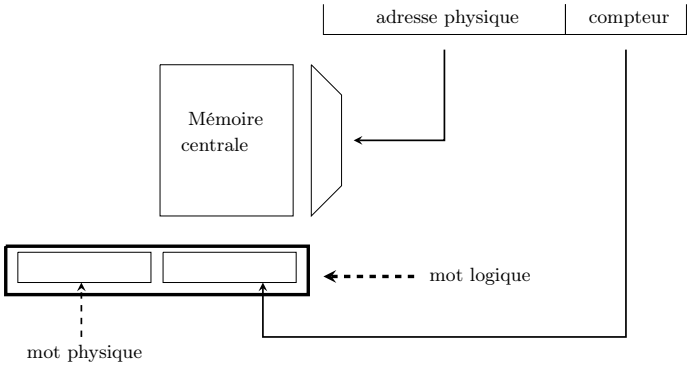


FIGURE 3.5 – Sérialisation des accès aux mots logiques.

3.3 Organisation d'une mémoire hiérarchisée

Dans la réalisation d'un ordinateur il y a compétition entre les propriétés et besoins du matériel et des logiciels, et leur coût. En effet, plus les technologies utilisées font les mémoires rapides et plus elles sont chères. En ramenant ce rapport au niveau de l'octet, plus la technologie est rapide et plus le prix de l'octet est élevé. Ceci explique que cette technologie sera de faible capacité.

Généralement, l'écart entre les vitesses des mémoires selon les technologies utilisées est discontinu et important. Par exemple les mémoires caches sont bien moins rapides que les registres, les mémoires centrales sont bien moins rapides que les caches, les disques sont bien moins rapides que les mémoires centrales, etc.

Ces empilements de technologies de capacité et vitesse différentes ont fait émerger une approche de l'organisation globale de la mémoire sous forme de hiérarchie.

3.3.1 Niveaux de mémoires

Une mémoire hiérarchisée est constituée par un ensemble de mémoires de tailles et de temps d'accès différents. Cette hiérarchie de mémoire est quasi transparente aux programmes s'exécutant sur l'unité centrale. Son fonctionnement est essentiellement fondé sur la théorie de la localité. En général, une hiérarchie de mémoire comprend les niveaux suivants :

— Le niveau le plus haut est composé des registres internes de l'unité centrale ; ce niveau est visible et manipulable par les programmes ;
— Le niveau immédiatement inférieur est appelé anté-mémoire ou cache. Ce niveau est caché aux programmes utilisateurs et n'est pas visible explicitement. Un cache est une petite mémoire très rapide par rapport à la mémoire centrale contenant une projection d'une partie des informations de celle-ci. Ce point pose le problème de son adressage qui sera en général associatif ;
— Le niveau intermédiaire est constitué par la mémoire centrale aussi appelée mémoire réelle. Ce niveau est toujours visible et est explicitement adressé par les programmes ;
— Le dernier niveau est constitué par la mémoire secondaire qui réside habituellement sur disque. Ce niveau n'est visible que du système d'exploitation qui assure sa gestion et son adressage.

La figure 3.6 illustre le concept de hiérarchie de mémoires. Chaque niveau de mémoire ne contient qu'une projection d'une partie des informations du niveau inférieur, il fonctionne donc comme un cache pour le niveau inférieur.

3.3.2 Taille de l'information gérée à chaque niveau

Du fait du prix croissant de l'octet à l'approche de l'unité centrale, la taille de chaque niveau de mémoire est de l'ordre de :

— Registres : quelques octects (8 par exemple en 64 bits),
— Cache : quelques milliers d'octets (de la dizaine à la centaine),
— Mémoire centrale : quelques milliards d'octets (4 à 8 Gio actuellement),
— Mémoire secondaire : quelques milliers de milliards d'octets (4 Tio actuellement).

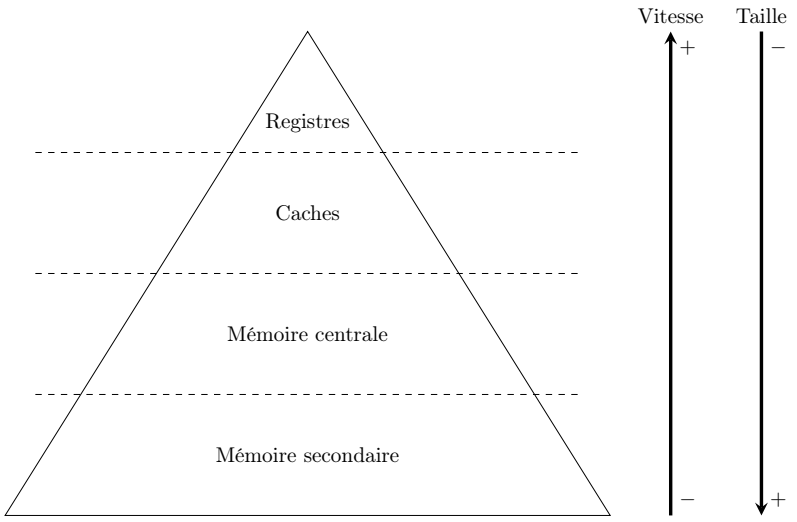


FIGURE 3.6 – Hiérarchie de mémoire.

Il n'est pas rare d'avoir plus de niveaux que ceux mentionnés. Par exemple, le cache d'un processeur Intel core i7 possède lui-même une hiérarchie de cache (cf. figure 3.7).

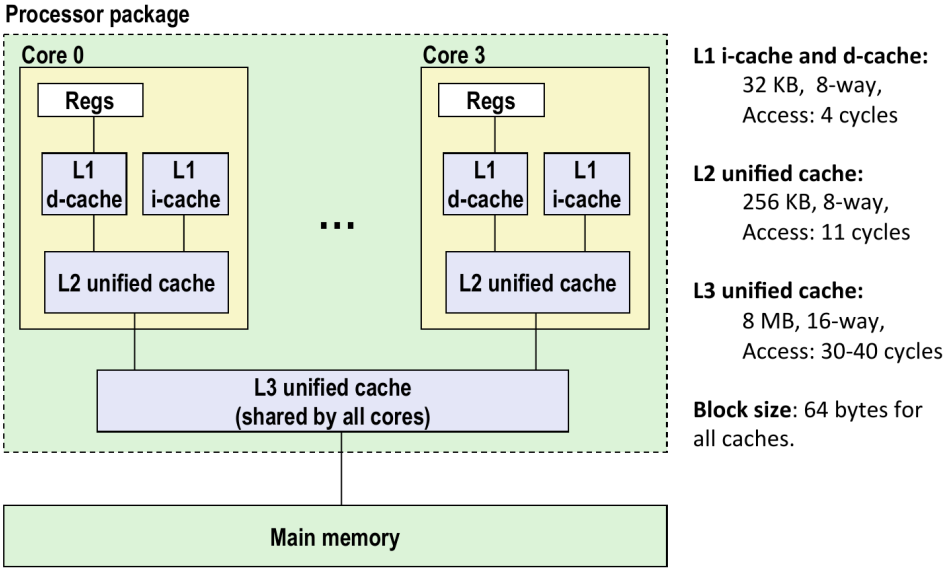


FIGURE 3.7 – Architecture hiérarchique du cache de l'intel core i7.

3.4 Les anté-mémoires ou mémoires caches

Une mémoire cache est une petite mémoire très rapide par rapport à la mémoire centrale et qui ne contient qu'une projection d'une partie des informations de la mémoire centrale. Son rôle est d'accélérer l'accès à des données de la mémoire centrale par l'utilisation d'un mécanisme fondé sur la théorie de la localité.

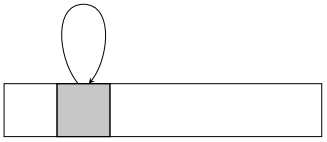
Le cache n'est pas visible par l'unité centrale, il est caché. L'unité centrale ne peut adresser que la mémoire centrale. Le cache va en quelques sortes intercepter les accès à la mémoire centrale et répondre à sa place plus rapidement si la donnée se trouve dans le cache.

3.4.1 Théorie de la localité

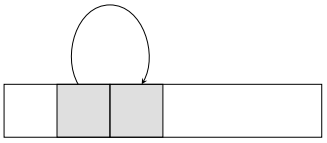
La hiérarchisation de mémoire en mémoires de plus en plus petites et plus rapides est basée sur le principe de la théorie de la localité. En effet, les caches ne contenant qu'une projection de la mémoire supérieure, le choix des données à projeter et conserver dans le cache est important. Le principe de la théorie de la localité est que les programmes tendent à utiliser des données ou des instructions ayant des adresses proches ou égales à celles récemment utilisées.

Il est possible de définir deux types de localité :

— Localité temporelle : les objets référencés récemment ont plus de chance d'être référencés de nouveau dans un futur proche.



— Localité spatiale : les objets ayant des adresses voisines ont plus de chance d'être référencés dans un futur proche.



3.4.1.1 Exemple

Soit le code en langage C suivant qui effectue la somme des éléments d'un tableau unidimensionnel.

```
sum = 0;
for (i = 0; i < n; i++)
    sum = sum + a[i];
return sum;
```

Ce code contient plusieurs exemples de localité temporelle et spatiale :

— Sur les données :
  — Temporelle : la variable `sum` est référencée à chaque itération,
  — Spatiale : le tableau `a[]` est linéairement parcouru élément par élément.
— Sur les instructions :
  — Temporelle : répétition des instructions de la boucle,
  — Spatiale : les instructions sont appelées en séquence.

Le parcours d'un tableau multidimensionnel doit être réfléchi pour essayer de tirer partie du principe de localité. Par exemple, en C, les données d'un tableau bi-dimensionnel (une matrice) sont stockées en mémoire ligne par ligne (*row major mode*). Le tableau peut être vue comme uni- ou multidimensionnel :

	...	...	...	...
a[0]	0	1	2	3
a[4]	4	5	6	7
a[8]	8	9	10	11
	...	...	...	...

	...	...	...	...
a[0][0]	0	1	2	3
a[1][0]	4	5	6	7
a[2][0]	8	9	10	11
	...	...	...	...

Ainsi, pour prendre en compte le principe de localité, pour effectuer la somme des éléments d'une matrice, il est préférable de savoir comment les données sont stockées en mémoire et modifier le code en conséquence. Par exemple on préférera le code suivant :

```
sum = 0;
for (i = 0; i < 3; i++)
    for (j = 0; j < 4; j++)
        sum = sum + a[i][j];
return sum;
```

plutôt que le code suivant :

```
sum = 0;
for (j = 0; j < 4; j++)
    for (i = 0; i < 3; i++)
        sum = sum + a[i][j];
return sum;
```

Le résultat sera strictement identique mais ce dernièr code va parcourir les valeurs selon les colonnes et donc effectuer un saut de la taille d'une ligne entre chaque valeur (cf. figure 3.8). Ainsi, vis-à-vis de l'utilisation du cache le second code ne sera pas optimal et l'exécution finale du programme sera ralentie.

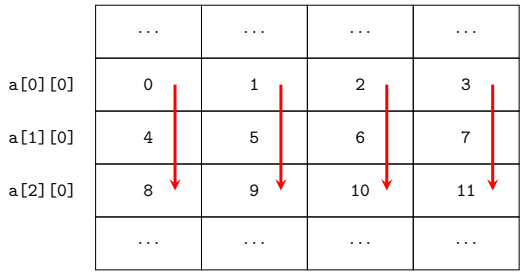


FIGURE 3.8 – Parcours des éléments d'un tableau en colonne d'abord.

Un parcours en ligne comme dans le premier cas va simplement passer à la valeur suivante et sera donc plus optimal pour l'exécution et l'utilisation du cache (cf. figure 3.9).

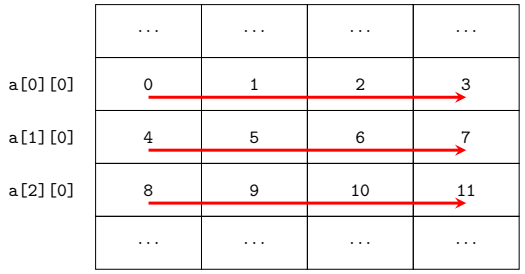


FIGURE 3.9 – Parcours des éléments d'un tableau en ligne d'abord.

3.4.2 Temps d'accès apparent

Dans une hiérarchie de mémoires composée d'un cache et de la mémoire centrale, le temps d'accès à la mémoire centrale est modifié par la présence du cache. Il est calculé en tenant compte de la probabilité de présence des informations dans le cache. Le temps d'accès est alors appelé *temps d'accès apparent*, il est calculé avec la formule suivante :

$$T_{AP} = \tau_{PC} \times T_{AC} + (1 - \tau_{PC}) \times T_{AMC} \tag{3.1}$$

où

T_{AP} est le temps d'accès apparent,

τ_{PC} est le taux de présence en cache,

T_{AC} est le temps d'accès au cache,

T_{AMC} est le temps d'accès à la mémoire centrale,

$1 - \tau_{PC}$ est le taux d'échec ou le taux d'absence des informations dans le cache.

L'élément d'information rangé dans le cache est le bloc. Un bloc correspond à un mot physique du cache et peut contenir plusieurs mots logiques du processeur.

Les adresses utilisées par les instructions ne références pas le cache, son accès est transparent, il est effectué, soit par indexation, soit associativement sur un champs préfixe reprenant tout ou partie de l'adresse mémoire. Cette particularité dans le mécanisme d'adressage permet de distinguer plusieurs types de caches différents.

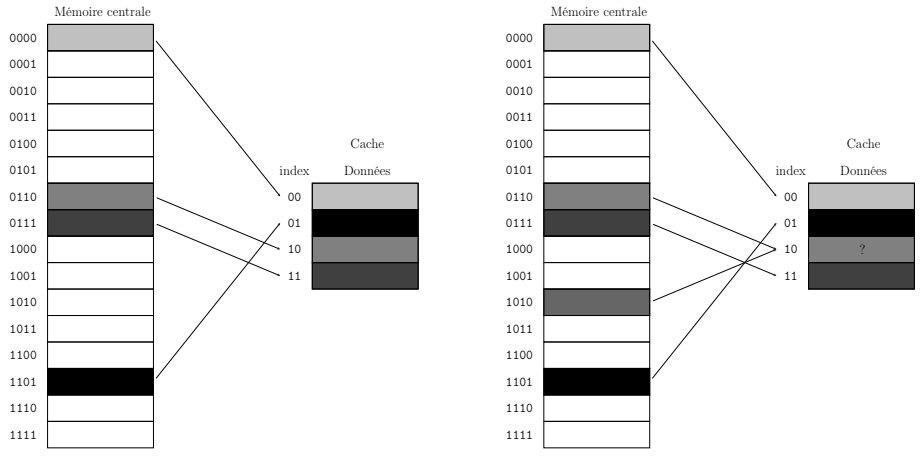
3.4.3 Typologie des caches

Le cache est une mémoire de taille inférieure à la mémoire centrale et contient une partie de celle-ci. Bien qu'elle ne soit pas directement adressable, chaque donnée dans le cache possède une adresse (relative à ce cache). La première problématique de l'utilisation d'un cache est de savoir comment placer les données dans le cache. La figure 3.10a illustre la placement de 4 données de la mémoire centrale dans le cache. La taille du cache est de 4 mots mémoires (blocs) et la taille de la mémoire est de 16 mots mémoires.

L'idée intuitive du placement illustré dans la figure 3.10a consiste à prendre comme adresse dans le cache les bits de poids faible de l'adresse mémoire. L'adresse de placement dans le cache s'appelle l'index. Le cache étant de taille 4, l'index nécessite 2 bits. La mémoire étant de taille 16, son adresse fait 4 bits de longueur. Les architectures de cache fonctionnant sur ce principe s'appellent les caches *direct-mapped*.

Pour retrouver une donnée, le contrôleur de cache va extraire de l'adresse envoyée par le processeur l'index de l'adresse dans le cache et renvoyer la donnée de l'entrée correspondante.

Supposons qu'une autre adresse soit demandée puis chargée dans le cache. Cette nouvelle adresse générera le même index qu'une donnée déjà présente, comme illustrée dans la figure 3.10b (les deux derniers bits sont identiques). La seconde problématique est de déterminer la valeur que va retourner le cache à la demande d'une des deux adresses par le processus. Comment le cache va gérer cette confusion possible entre les deux données et les deux adresses ?



(a) Par indexation à partir de l'adresse.

(b) Problème d'adresses synonymes (même index).

FIGURE 3.10 – Adressage direct des données dans un cache.

La méthode employée pour désambiguïser ces problèmes d'adresses synonymes est d'introduire une information supplémentaire qui permettra de distinguer les adresses générant le même index. Cette information est une préfixe (*tag*) et à chaque entrée du cache est associé un préfixe extrait de l'adresse dont la donnée se trouve effectivement dans le cache (voir figure 3.11).

Ainsi, à partir de l'adresse mémoire d'une donnée, deux éléments en sont extraits :
— l'index de placement dans le cache à partir des bits de poids faibles de l'adresse
— le préfixe (*tag*) de discrimination à partir du reste de l'adresse.

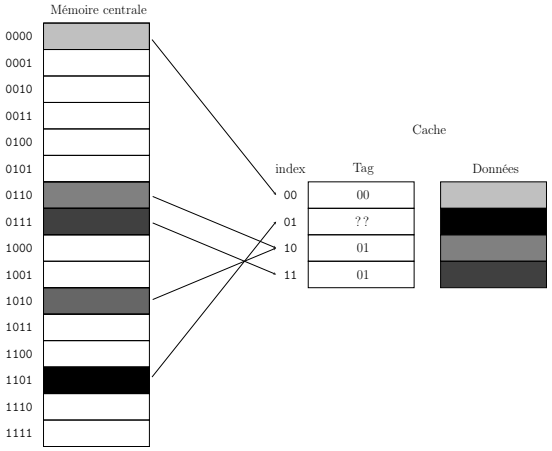


FIGURE 3.11 – Adressage index-préfixe dans un cache.

Cependant, stocker un unique mot mémoire dans le cache ne prend pas en compte la localité spatiale. En effet, un mot mémoire (un octet par exemple) utilisé très souvent (localité temporelle) et placé dans le cache a de forte probabilité d'être directement lu dans le cache au prochain accès. Mais pour un tableau d'octets par exemple, une seule donnée n'est utilisée et présente à la fois dans le cache. Au prochain élément accéder, il faudra le charger dans le cache à partir de la mémoire centrale, puis le suivant etc, et ainsi le cache n'apporte aucune accélération.

Pour palier ce problème, l'architecture du cache peut être modifiée pour définir un mot physique du cache comme plusieurs mots logiques, *i.e.* un mot du cache contient plusieurs mots mémoires contigus. Un mot mémoire est appelé *bloc*, et ainsi un mot physique du cache peut contenir plusieurs blocs. Ceci va avoir un impact sur l'extraction de l'index et du préfixe car les bits de poids faibles vont maintenant correspondre à la sélection du bloc dans le mot du cache (qu'on appelle ligne de bloc *block line* ou ligne du cache). L'index sera extrait à partir des bits suivants, puis le préfixe à partir des bits de l'adresse restants.

3.4.3.1 Les caches *direct-mapped*

Les caches *direct-mapped* sont la formalisation des principes présentés dans la section précédente. Le mécanisme d'accès à ce type de cache repose sur le principe qu'un bloc d'information situé en mémoire centrale à une adresse x ne peut être placé dans le cache qu'à un et un seul emplacement (*slot*) fonction de x.

Cet emplacement est désigné par un index calculé à partir de l'adresse du bloc en mémoire centrale. La fonction de calcul de l'index réalise la projection des blocs situés en mémoire centrale dans les emplacements du cache, cette projection est non injective. En effet, deux blocs situés à des adresses différentes en mémoire centrale peuvent être projetés dans le même emplacement du cache. Dans ce cas, ces blocs sont appelés des blocs synonymes (cf. figure 3.10b).

La partie du cache réservée aux adresses est composée du champs préfixe (*tag*) qui permet de distinguer les blocs synonymes.

La figure 3.12 explicite un exemple simple de fonctionnement de l'accès à ce type de cache.

Le cache est adressé par un index codé sur 2 bits variant de 0X00 à 0X11 (le cache possède $4 = 2^2$ entrées). Un *block line* contient 2 octets, il y a donc 2 blocs par ligne de cache. Pour choisir le bloc 1 seul bit suffit. L'adressage se faisant sur 4 bits, on associe le restant des bits de poids fort au préfixe. Le préfixe est donc codé sur 1 bits et permet de désambiguïser les adresses synonymes.

L'accès est effectué selon le schéma suivant :

1. L'index obtenu par extraction de la chaîne de bits numéro 2 à 1 de l'adresse désigne un emplacement unique dans le cache.

2. Le préfixe de discrimination des blocs synonymes correspond au bit de poids fort de l'adresse (bit numéro 3).
3. Le contrôleur du cache compare le préfixe de l'emplacement trouvé avec la partie préfixe de l'adresse en mémoire centrale du bloc cherché.
   — S'il y a égalité alors on renvoie le bloc de données du cache vers l'unité centrale (*Cache Hit*).
   — En cas de non égalité, le contrôleur du cache génère un signal de défaut de cache (*Cache Miss*) et l'unité centrale accède à la mémoire centrale pour y lire l'information. La mise à jour du cache est ensuite effectuée.

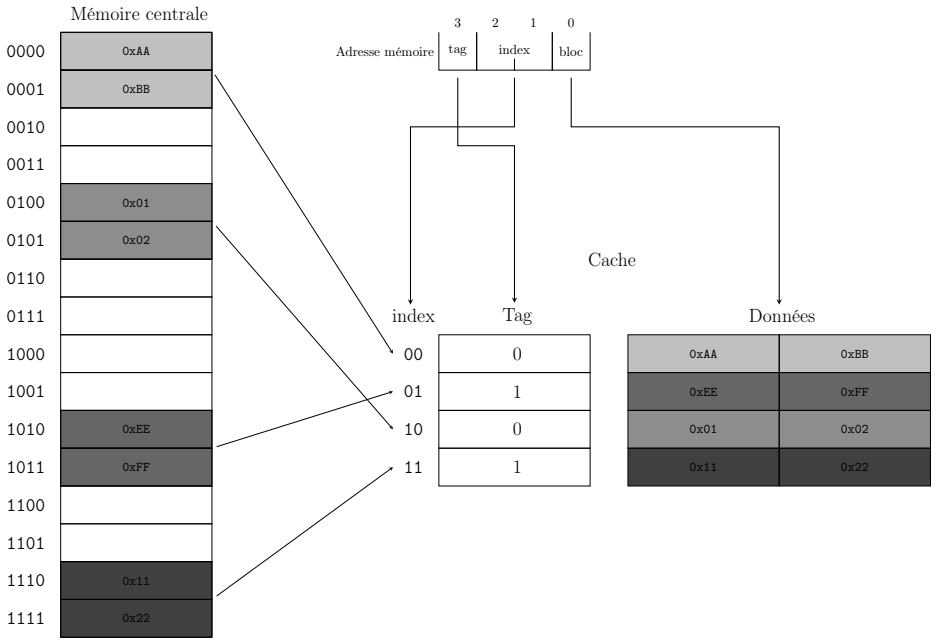


FIGURE 3.12 – Schéma d'un cache *direct-mapped*.

3.4.3.2 Les caches *full-associative*

Les caches *direct-mapped* sont facile à mettre en oeuvre car l'emplacement d'une donnée dans le cache est fonction directe de l'adresse. Cependant, si un programme utilise alternativement deux objets dont les adresses génère le même index, le cache sera constamment en train de mettre à jour son contenu pour cette entrée et donc ne jouera plus son rôle.

Les caches *full-associative* permettent de s'abstraire de cette fonction de projection et place les données indépendamment de leur adresse, comme les mémoires associatives. Dans ce type de cache, il y a mémorisation des blocs d'information et de leurs adresses complètes en mémoire centrale. Il n'y a plus d'index et l'adresse complète devient le préfixe (*tag*) (en supposant que ligne de cache ne fasse qu'un bloc). Un bloc de données et son préfixe associé peuvent être rangés dans un emplacement quelconque du cache.

Lorsque le processeur demande l'accès à une donnée, le contrôleur du cache compare l'adresse de l'information requise avec les préfixes stockés. S'il y a correspondance, il envoie le bloc de données associé au processeur.

Il y a deux façons de voir l'implantation d'une telle mémoire associative :
— Soit comme une pile de préfixes : les préfixes sont stockés dans la pile selon une politique de gestion particulière (FIFO, autres) (cf. figure 3.13).
— Soit comme un ensemble de mini-caches, de taille d'un bloc, accédés en parallèle (cf. figure 3.14).

La figure 3.13 présente un exemple de ce type d'organisation. Le cache a une capacité de 512 emplacements, chacun contenant un préfixe de 22 bits et un bloc de données de 16 bits. Le préfixe contient l'adresse complète du bloc de données en mémoire centrale. Ainsi, le mot situé à l'adresse 0xFFFFFC est référencé dans le cache par le préfixe 0x3FFFFF (le bit de sélection du bloc n'est pas pris en compte).

Les poids faibles de l'adresse permettent au processeur de sélectionner le mot désigné dans la ligne de blocs récupérée.

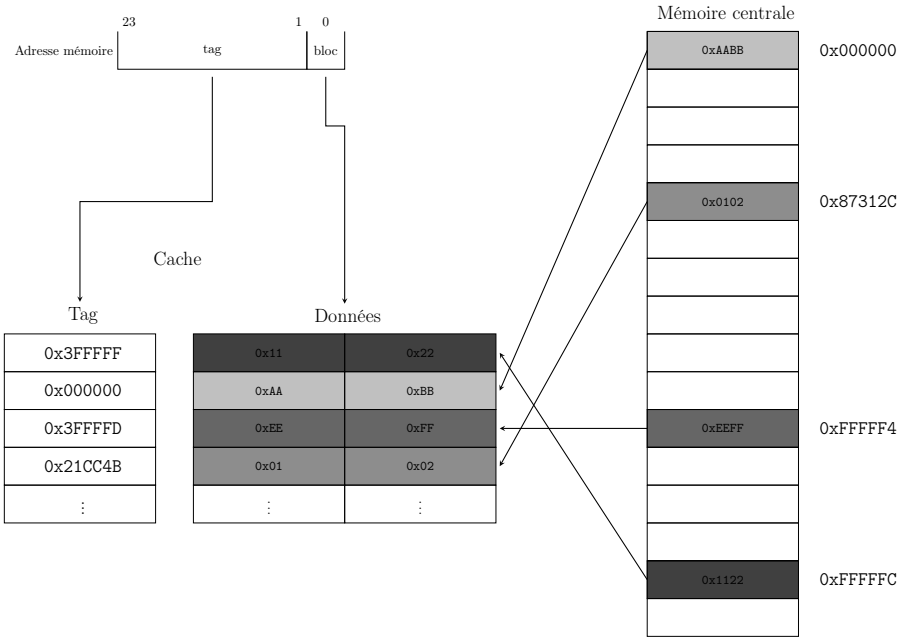


FIGURE 3.13 – Schéma d'un cache *full-associative*.

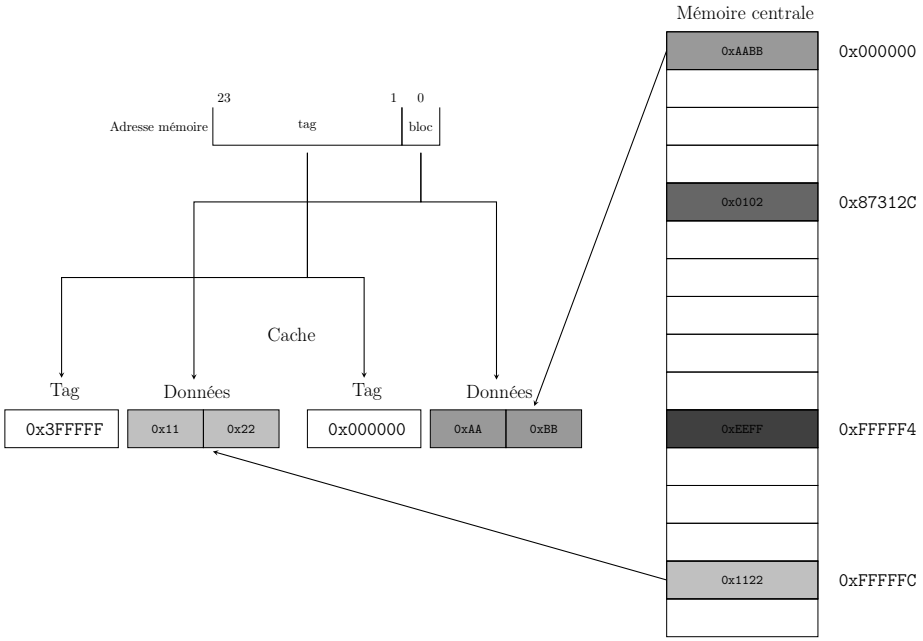


FIGURE 3.14 – Schéma d'un cache *full-associative* (parallèle).

L'avantage des caches full-associatives est le fait ne pas avoir à gérer une position précise des données. Ainsi, si plusieurs adresses auraient été projetées sur le même index, dans le cas de ce type de cache, les données sont dans des emplacement différents et le cache peut ainsi joueur son rôle d'accélérateur. Par contre, la réalisation d'un tel cache est complexe et plus lent.

3.4.3.3 Les caches *set-associative*

L'architecture des caches *set-associative* est un compromis entre les caches *direct-mapped* et les caches *full-associative* : elle garde l'efficacité d'indexation par une fonction de placement et gère les synonymes par l'utilisation de sous-caches parallèles.

Ainsi, à partir de l'adresse en mémoire centrale d'un bloc de données, un index unique est calculé et une recherche dans plusieurs sous-caches est effectuée en parallèle. Chaque sous-cache possède sa propre mémoire de préfixes et sa propre mémoire de données.

La figure 3.15 présente un cache *set-associative* composé de 2 sous-caches (*2 way set-associative cache*).

Dans cet exemple, deux sous-caches fonctionnent en parallèle et sont tous les deux adressés par un index codé sur 13 bits et variant de 0x0000 à 0X1FFF.

L'index obtenu par extraction de la chaîne de bits 14 à 2 de l'adresse désigne un emplacement unique dans le cache.

Le préfixe de discrimination des blocs synonymes correspond aux 9 bits de poids forts de l'adresse.

A un index correspondent deux préfixes, le contrôleur du cache effectue deux comparaisons en parallèle pour déterminer dans quel sous-cache est situé le préfixe issu de l'adresse en mémoire centrale.

Ce type de cache requiert deux comparaisons en parallèle, ce qui complique sa logique mais améliore nettement, par rapport à un cache *direct-mapped*, la probabilité de trouver l'information recherchée.

Ce type de cache est nommé *n-way set-associative cache* avec n représentant le nombre de sous-caches en parallèle.

Les trois exemples de fonctionnement ci-après utilisent le modèle décrit par la figure 3.15.

Exemple 1 :
- Le processeur recherche une information située à l'adresse 0x00FFFC en mémoire centrale.
- Le contrôleur du cache examine le contenu du premier sous-cache à l'index 0x1FFF pour vérifier si le préfixe est égal à 0x001, d'après la figure, c'est le cas.
- En parallèle, le contrôleur a effectué une opération identique sur le second sous-cache et a trouvé la correspondance. Le bloc de données contenant 55556666 est transmis du second sous-cache vers le processeur.

Exemple 2 :
- Le processeur recherche une information située à l'adresse 0x008004 en mémoire centrale.
- L'index calculé est 0x001 et le préfixe est 0x001.
- Ce bloc de données est trouvé dans le premier sous-cache.

Exemple 3 :
- Le processeur recherche une information située à l'adresse 0x010000 en mémoire centrale.
- L'index calculé est 0x000 et le préfixe est 0x002.
- Ce bloc de données n'est pas présent dans le cache. L'unité centrale le récupère depuis la mémoire centrale et une mise à jour du cache est effectuée. Le choix du sous-cache mis à jour sera fait par le contrôleur du cache selon une heuristique, par exemple LRU.

3.4.3.4 Récapitulatif des caractéristiques d'un cache

Il est possible de généraliser les trois types de cache précédent en un seul qui est défini par les paramètres :
- S : le nombre de partitions (*sets*), correspond au nombre d'entrées dans le cache défini sur s bits tels que $S = 2^s$;
- E : le nombre de blocs de mots du cache *cache lines*, qui correspond au nombre de sous-caches interrogés en parallèle pour un même index, défini sur e bits tels $E = 2^e$;
- B : Le nombre de mots mémoires dans un bloc du cache, en général des octets consécutifs, défini sur b bits tels que $B = 2^b$.

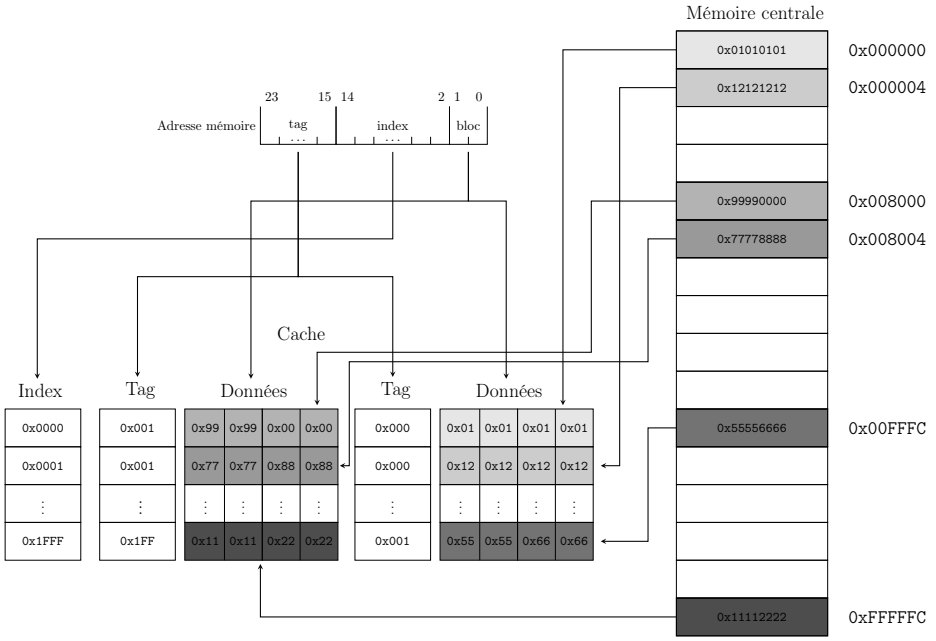


FIGURE 3.15 – Schéma d'un cache *2 way set-associative*.

Une fois ces paramètres définis, la taille du cache en octets est être déterminée par le produit du nombre d'octets par blocs, par lignes et par partitions :

$$C = S \times E \times B = 2^{s+e+b}$$

La figure 3.16 représente cette synthèse du découpage d'un cache selon ces trois paramètres.

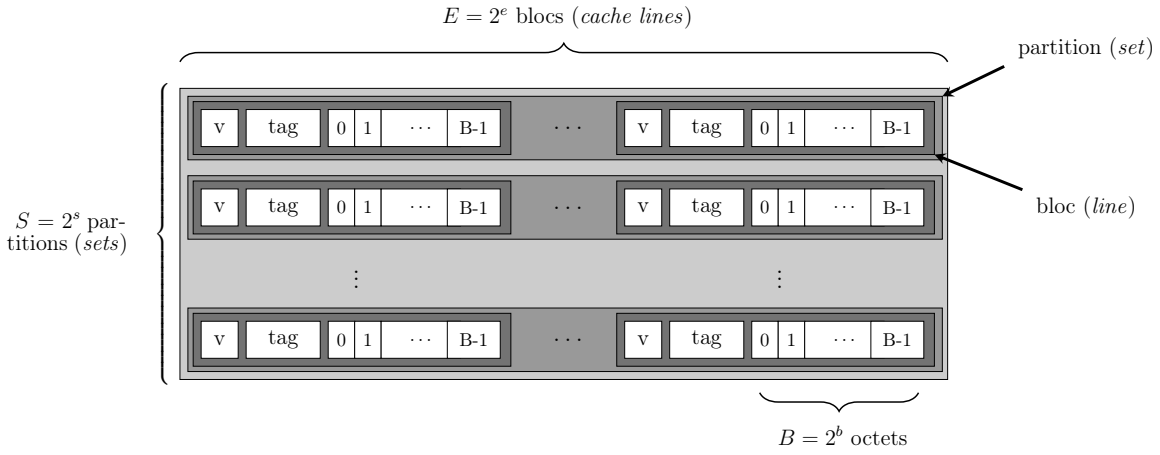


FIGURE 3.16 – Caractéristiques générales d'un cache.

3.4.4 Gestion des caches

3.4.4.1 Adresses utilisées pour le recherche

La recherche dans le cache peut être effectuée :

— Soit à partir de l'adresse réelle : le contrôleur du cache utilise l'adresse traduite par la MMU. Cette solution est en générale retenue pour la simplicité de sa mise en oeuvre.
La figure 3.17 illustre ce mode d'accès. Le fonctionnement est séquentiel, l'adresse virtuelle est traduite puis une recherche dans le cache à l'aide de l'adresse réelle est effectuée.

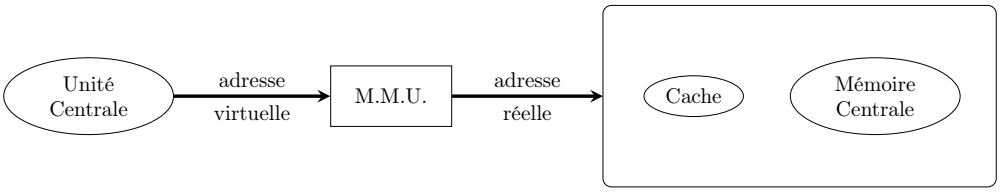


FIGURE 3.17 – Interrogation du cache par des adresses réelles.

— Soit à partir de l'adresse virtuelle. Cette seconde méthode est plus complexe à mettre en oeuvre mais elle améliore les performances des accès. La recherche dans le cache et la traduction des adresses virtuelles se font en parallèle.
La figure 3.18 illustre ce second mode d'accès.
Cette technique nécessite une vérification supplémentaire. En effet, le cache est une projection de la mémoire réelle et non de l'espace virtuel. Il faut donc vérifier la correspondance entre l'objet désigné en mémoire réelle et l'objet accédé dans le cache. La vérification est effectuée en comparant le préfixe du cache avec un préfixe calculé à partir de l'adresse traduite.

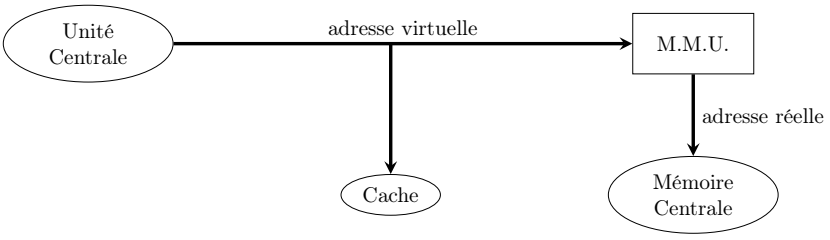


FIGURE 3.18 – Interrogation du cache par des adresses virtuelles.

3.4.4.2 Stratégies de remplacement

Les stratégies de remplacement applicables à un cache sont identiques à celles que l'on retrouve dans la gestion d'information :
— RANDOM : Choix aléatoire de l'information à vider du cache.
— FIFO : L'information entrée la première dans le cache est vidée.
— LRU : L'information qui a été inutilisée durant la plus longue période est vidée du cache.

Remarque : La stratégie LRU est très souvent appliquée car elle est la plus proche de la stratégie optimale.

3.4.4.3 Mise à jour de la mémoire centrale

Lorsque le processeur effectue la modification d'une donnée du cache, plusieurs techniques peuvent être appliquées pour la mise à jour de l'objet correspondant en mémoire centrale :
— Ecriture simultanée : *write through*
Chaque fois que le processeur modifie une donnée du cache, il y a automatiquement écriture de la modification dans la mémoire centrale. Le temps perdu par le processeur pour une écriture dans le cache équivaut alors à celui perdu pour un défaut de cache pour une lecture. Cependant, la cohérence des données est forte.
— Ecriture postée : *posted write*
Cette technique constitue une amélioration de la méthode précédente. L'écriture en mémoire centrale s'effectue de manière asynchrone en utilisant un tampon d'écriture (buffer) contenant l'adresse et la valeur du mot à écrire. La logique d'écriture du tampon se charge de la mise à jour de la mémoire centrale pendant la poursuite de l'exécution du flot d'instructions par l'unité centrale.
Dans le cas où une écriture en mémoire est demandée par l'unité centrale et que le tampon d'écriture est plein, l'unité centrale va attendre son vidage.

— Ecriture retardée : *write back* ou *copy back*
Dans ce mécanisme, à chaque bloc du cache est associé un bit indiquant si la donnée a été modifiée (*Flag Dirty*). A chaque modification d'une valeur dans un bloc, ce bit est positionné à 1 sans mettre à jour la mémoire centrale. Au moment de ranger un autre bloc dans ce même emplacement du cache, le contrôleur du cache teste le *flag Dirty*. S'il est positionné, il effectue une copie en mémoire centrale du bloc modifié avant de charger le nouveau bloc dans le cache.
Cette méthode est plus rapide que le *write through* puisque la recopie n'est pas systématique. Toutefois, à un instant donné, le contenu du cache n'est pas strictement cohérent avec celui de la mémoire centrale.

Les techniques souvent utilisées sont *write through* et *write back*.

3.4.4.4 Problèmes posés par la gestion des caches

Cohérence du cache

La mise à jour de la mémoire centrale par la méthode *write back* pose le problème de la cohérence du cache.

Ces problèmes apparaissent dans un système multiprocesseur fortement couplé où les processeurs se partagent la mémoire centrale, ou dans un système mono-processeur avec des périphériques effectuant des entrées/sorties par DMA (*Direct Memory Access*). Dans ce cas, il faut garantir la cohérence des caches afin que chaque processeur ou organe d'entrée/sortie n'accède pas à des données périmées.

Plusieurs solutions à ce problème de cohérence peuvent être mises en oeuvre :

— Mettre en place une mémoire non cachée partagée entre les processeurs,

— Utiliser la technique appelée *bus snooping*, *bus watch*, *snoopy cache* :
Elle consiste à espionner les bus d'accès à la mémoire centrale pour déceler les signaux d'écriture concernant des adresses présentes dans le cache. Lors de la détection d'une adresse d'un bloc présent dans le cache, le contrôleur du cache met à jour dynamiquement le contenu du bloc correspondant dans le cache.

— Vider le cache, *cache flush* :
Avant de céder le bus à un autre processeur ou organe d'entrée/sortie, on peut vider complètement le cache. Ce vidage implique l'invalidation de toutes les entrées et pour chaque entrée ayant le *Dirty Flag* positionné, l'écriture en mémoire centrale du bloc correspondant.
Cependant, pour retrouver les performances optimales du mécanisme de cache, il faudra attendre que le cache soit de nouveau rempli.

Trashing

Un autre problème posé par l'emploi de caches est celui du trashing. Ce problème se produit dans un système multiprogrammé utilisant un *cache direct-mapped*.

Supposons que plusieurs processus références des adresses mémoires différentes générant le même index pour le cache. Les données correspondant à ces adresses sont donc projetées dans le même emplacement du cache.

De ce fait, lors de changement de contexte successifs, le processeur va successivement charge la donnée correspondant au processus 1, puis la vider du cache pour mettre la donnée correspondant au processus 2, puis la vider du cache pour mettre la donnée correspondant au processus 3, ...

Ce phénomène de défaut de cache répété est appelé *trashing* par analogie avec le phénomène d'écroulement dans un système à mémoire virtuelle paginée.

L'utilisation d'un cache plus grand géré en *set-associative* avec k blocs résout ce problème. Le phénomène de *trashing* se produit d'autant moins que le nombre de partitions est élevé.

3.5 La mémoire virtuelle

3.5.1 Principe

La taille d'un programme, des données et de la pile ne peut pas dépasser l'espace disponible en mémoire centrale.

Pour éliminer cette contrainte, le système d'exploitation ne conserve en mémoire que les parties du programme qui sont en cours d'utilisation et stocke le reste sur disque. Les va-et-vient des fragments de l'espace d'adressage sont effectués dynamiquement lors de l'exécution du processus.

La mémoire virtuelle est particulièrement bien adaptée aux système multiprogrammé. En effet, lorsqu'un processus attend le chargement de l'une de ses parties, l'unité centrale peut être allouée à un autre processus.

3.5.2 Espace virtuel privé ou global

L'espace virtuel peut être implanté selon deux philosophies différentes :

— Chaque processus peut posséder son propre espace virtuel privé. Le principal avantage de ce choix est que les processus sont de facto protégés les uns des autres, chaque processus n'ayant une visibilité que sur son espace privé.

— Tous les processus partagent un espace virtuel global. L'inconvénient de ce choix est qu'il faut prévoir explicitement un mécanisme de protection entre les processus.

En général, le mécanisme implanté correspond à un compromis entre ces eux philosophies. En effet, la solution des espaces virtuels privés pénalise le partage d'objets communs, comme le noyau du système, et la vérification des protections alourdit le mécanisme de traduction dans les architectures à espace virtuel global.

Le choix du type d'espace virtuel global ou privé a une conséquence importante sur les performances de l'architecture :

— Une architecture à espace virtuel privé nécessite que chaque processus possède sa propre fonction de traduction, donc ses propres tables. Un changement de contexte entre deux processus sera donc très coûteux puisqu'il faudra échanger les tables de la fonction de traduction et mettre à jour le mécanisme de traduction accéléré ;

— Une architecture à espace virtuel global évite ce problème, la fonction de traduction est commune à tous les processus.

3.5.3 Segmentation et pagination

Quelques définitions préliminaires des objets manipulés sont nécessaires :

— Segment : une partie, de longueur variable, de l'espace virtuel correspondant à un découpage logique des données (procédure, fichier, variables, code, etc) ;

— Page : partie, de longueur fixe, de l'espace virtuel correspondant à un découpage physique ne tenant pas compte du contenu logique ;

— Case : Partie, de longueur fixe, de la mémoire réelle destinée à contenir une page de l'espace virtuel. Une case est donc une partie physique de la mémoire réelle, inamovible et définie par son adresse. La case est le contenant, la page est le contenu.

Le transfert des parties d'un programme entre le disque et la mémoire centrale peut être effectué selon deux stratégies différentes :

— Le système d'exploitation peut mettre en oeuvre une gestion par segments et transférer des zones de longueur variable correspondant à un découpage logique du programme
  — Segment de code,
  — Segment des données initialisées,
  — Segment des données non initialisées,
  — Segment de pile.

  Ce mécanisme est appelé segmentation : il consiste à charger les segments dans des zones de la mémoire centrale. Comme les segments sont de longueur variable, ce type de gestion rend les entrées/sorties très pénalisantes.

— Le système d'exploitation peut mettre en oeuvre une gestion par pages et manipuler des pages de longueur fixe. Les entrées/sorties sont beaucoup plus efficaces que le cas de la gestion par segments mais la structure logique des programmes est ignorée.

Ce mécanisme est appelé pagination : la pagination consiste à transférer des pages depuis la mémoire secondaire dans des cases de la mémoire centrale.

Remarques :
- Les pages et les cases ont la même taille,
- Les transferts entre le disque et la mémoire centrale se font toujours par page entière.
- Il est également possible de mélanger les deux stratégies en gérant des segments découpés en pages (segmentation paginée).

3.5.4 Allocation de la mémoire centrale

Pour s'exécuter, les instructions d'un programme doivent être chargées en mémoire centrale.

Les instructions exécutées par un processus désignent les objets qu'elles manipulent dans un espace appelé "espace d'adressage". Les adresses émises par le processeur peuvent être de deux types :
- Adresses réelles : ces adresses sont envoyées directement à la mémoire et désignent des emplacements physiques de celle-ci ;
- Adresses virtuelles : ces adresses sont envoyées à une unité de traduction, la MMU [4] qui les traduit en adresses réelles.

Lorsque les instructions contiennent des adresses virtuelles, l'espace d'adressage est appelé espace virtuel. Lorsque le mécanisme de traduction est propre à chaque processus, l'espace virtuel est appelé espace virtuel privé. Réciproquement, lorsque ce mécanisme est commun à l'ensemble des processus, l'espace virtuel est appelé espace virtuel global.

Lors de la traduction d'un programme par le compilateur celui-ci range les objets dans un espace logique. L'éditeur de liens mets à jour les adresses logiques pour les transformer en adresses de l'espace d'adressage, réel ou virtuel.

3.5.5 Liaison espace d'adressage — espace réel

A l'exécution d'un programme, les objets manipulés par les instructions doivent se trouver en mémoire réelle et y être accessibles par les modes d'adressage. Il est nécessaire d'établir la liaison entre l'espace d'adressage d'un processus et l'espace de la mémoire réelle.

Rappels :
- Un module exécutable est translatable s'il peut être chargé à n'importe quelle adresse de la mémoire réelle. Le chargeur assure la translation à l'aide du dictionnaire de translation ;
- Un module exécutable est relogeable si une fois son exécution démarrée il peut encore être déplacé en mémoire.

3.5.5.1 Liaison statique

La chaîne de production de programmes, compilateur et éditeur de liens, produit un code utilisant des adresses réelles. La liaison espace virtuel — espace réel est faite lors de la création du programme exécutable. Ce type de liaison est courante sur les micro-ordinateurs à système mono-tâche.

Le module exécutable généré est non translatable et non relogeable.

3.5.5.2 Liaison effectuée au chargement en mémoire

La chaîne de production de programme produit un code translatable. Le chargeur effectue la translation de l'implantation en mémoire réelle.

Le module doit être translatable, la translation est effectuée par le chargeur, il n'est pas forcément relogeable, sa ré-implantation peut être statique.

4. Memory Management Unit

3.5.5.3 Liaison effectuée à l'exécution de chaque instruction

On effectue la correspondance espace d'adressage virtuel — espace réel à l'exécution de chaque instruction. Le lien adresse virtuelle — adresse réelle contient :
— Une adresse si l'objet désigné est en mémoire à cet instant,
— Nil si ce n'est pas le cas.

Ce mécanisme est très souple mais coûteux, l'exécution des instructions est ralentie par la traduction des adresses.

Conséquences sur le module exécutable :
— Le module est translatable : le chargeur charge le module dans une région de l'espace virtuel, à aucun moment le programme se manipulera des adresses réelles ;
— Le module est relogeable grâce au mécanisme de traduction dynamique des adresses.

3.5.6 Les mécanismes d'adressage

Dans ce paragraphe nous décrivons les différents mécanismes matériels qui réalisent la liaison entre l'espace d'adresse et l'espace réel.

3.5.6.1 Cas de la mémoire uniforme

les registres de base Lors de l'exécution de chaque instruction, ce mécanisme ajoute à chaque adresse le contenu d'un registre spécial, en général inaccessible au programmeur. Cette addition provoque la translation complète de l'espace d'adressage pour le faire coïncider avec la zone de mémoire réelle attribuée au processus.

La mémoire topographique : memory mapping Le principe est analogue à celui des registres de base mais à la place d'une addition on effectue une substitution des poids forts de l'adresse avec le contenu d'un registre pris dans une table.

3.5.6.2 Cas de la mémoire hiérarchisée

Deux types de mécanismes d'adressage peuvent être utilisés dans une mémoire hiérarchisée : le découpage en segments ou le découpage en pages. Chacun de ces découpages est un compromis entre :
— Les performances pour les transferts entre les niveaux de mémoires,
— La protection des informations,
— La facilité de gestion.

Les transferts de page sont plus simples et plus performants que les transferts de segments mais la protection des segments est plus souple et plus facile à gérer.

La segmentation Une adresse virtuelle est composée de deux champs :
— Le numéro du segment : poids forts de l'adresse,
— Le déplacement dans le segment : poids faibles de l'adresse.

Le numéro de segment est utilisé comme indice dans la Table Des Segments (TDS), dont chaque entrée contient un descripteur de segment. En général, un descripteur est constitué des champs suivants :
— Un indicateur de validité pour l'entrée
— L'adresse de base du segment en mémoire réelle,
— La longueur du segment,
— Des informations pour la protection du segment.

L'algorithme de traduction d'une adresse segmentée est schématiquement le suivant (cf. figure 3.19) :

1. Récupération du numéro de segment et accès à la table des segments,
2. Si (indicateur_de_validité = faux) alors Défaut_de_Segment,
3. Si (longueur_du_segment < déplacement_de_l'instruction) alors Erreur_d'adressage,

4. Si (informations_de_protection ! = droits_du_processus) alors Violation_Protection_Mémoire,
5. calcul de l'adresse réelle en additionnant l'adresse de base et le déplacement.

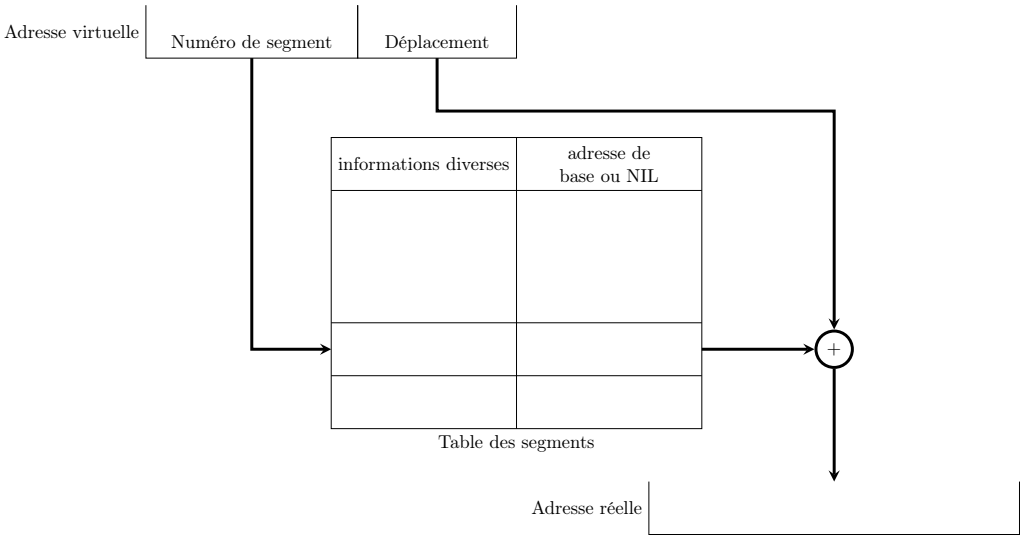


FIGURE 3.19 – Traduction d'une adresse virtuelle segmentée.

La pagination Une adresse virtuelle est composée de deux champs :
— Le numéro de page : poids forts de l'adresse,
— Le déplacement dans la page : poids faibles de l'adresse.

Le numéro de page est utilisé comme indice dans la Table Des Pages (TDP), dont chaque entrée contient un descripteur de page. En général, un descripteur est constitué des champs suivants :
— Un indicateur de validité pour l'entrée
— Le numéro de la case associée à la page,
— Des informations diverses : protection de la page, information de service pour les algorithmes de gestion des transferts de page ...

L'algorithme de traduction d'une adresse paginée est schématiquement le suivant (cf. figure 3.20) :
1. Récupération du numéro de page et accès à la table des pages,
2. Si (indicateur_de_validité = faux) alors Défaut_de_Page,
3. Si (informations_de_protection ! = droits_du_processus) alors Violation_Protection_Mémoire,
4. Mise à jour des informations de service,
5. calcul de l'adresse réelle en substituant le numéro de page par le numéro de case.

L'accès systématique à la Table des Pages rend cette traduction lourde et coûteuse en temps. Des mécanismes d'accélération ont été proposés, le plus récent est la TLB [5], utilisant le même principe que les mémoires caches (3.21). La TLB est une table dont chaque entrée possède les champs suivants :
— Un indicateur de validité de l'entrée,
— Un préfixe de discrimination des synonymes (Tag),
— Un numéro de case,
— Des informations de service.

L'indice d'une entrée est calculée à partir du numéro de page à l'aide d'une fonction de hashage. A l'issue de ce calcul, il faut vérifier si l'entrée sélectionnée correspond à l'adresse virtuelle qu'on souhaite traduire. A cet effet, le préfixe de discrimination des synonymes est comparé avec une partie de l'adresse

5. Translation Lookaside Buffer

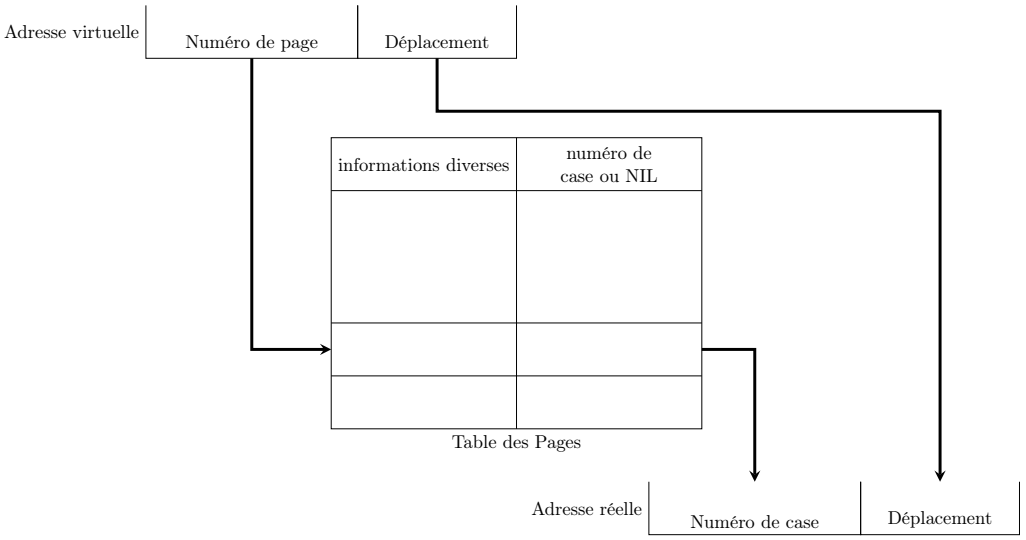


FIGURE 3.20 – Traduction d'une adresse virtuelle paginée.

virtuelle de départ. En cas de discordance, il y a émission du signal Défaut_TLB traité par le système.

La TLB n'étant qu'un mécanisme d'accélération, un Défaut_TLB ne signifie pas que la page n'est pas chargé en mémoire. Le système devra alors parcourir la Table des Pages pour effectuer la traduction.

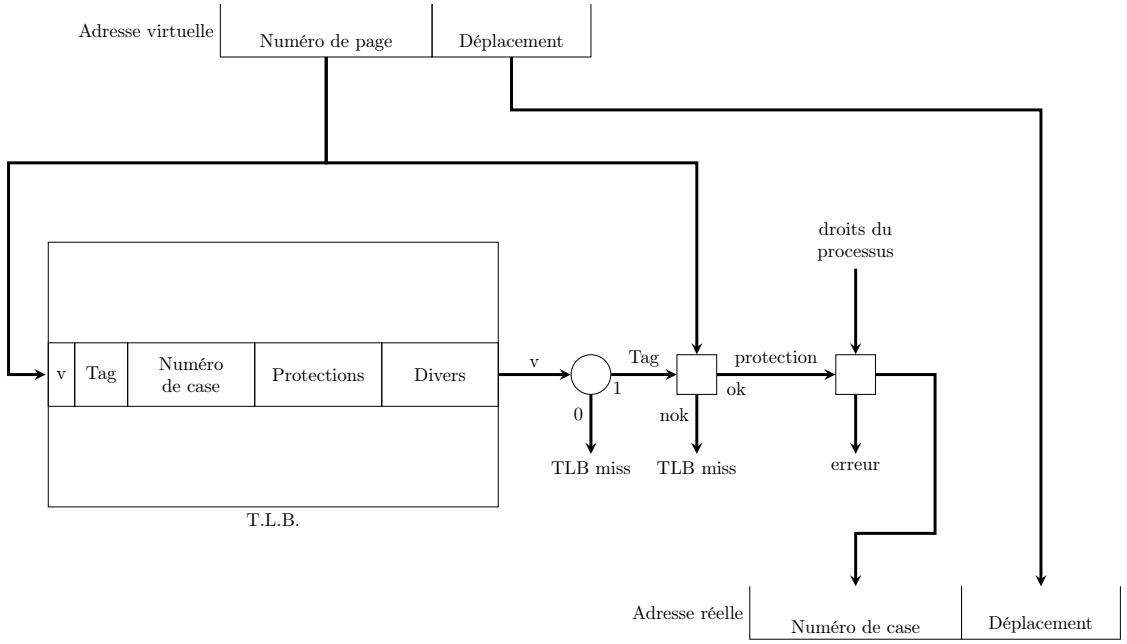


FIGURE 3.21 – Fonctionnement d'une T.L.B.

Des problèmes liés à la taille et à l'accès des tables apparaissent lors de la mise en oeuvre de l'adressage dans une mémoire paginée :

— La taille de la Table de Pages est directement liée à la taille de l'espace virtuel. Sur les processeurs actuels, la taille de l'espace virtuel est de l'ordre de plusieurs Gio, ce qui implique une Table des Pages de plusieurs millions d'entrées, ce qui est beaucoup.

— L'accès aux tables : les tables étant très grandes, dans certains systèmes elles ne sont pas résidentes en mémoire centrale ce qui allonge leur temps d'accès.

Une technique d'accélération consiste à ne conserver dans une petite table que les derniers couples (numéro_page, numéro_case) accédés par un processus. Cette technique met en oeuvre le principe de la localité. L'utilisation d'une mémoire associative, ou d'une TLB, s'avère très efficace.

La pagination à deux niveaux a été introduite pour remédier aux inconvénients de la pagination simple, elle a pour objectifs :
— La réduction de la taille de la table des pages,
— Le partage des morceaux de l'espace virtuel.

Définition : Une hyperpage est un ensemble de pages dont la liaison avec la mémoire réelle est décrite par la même Table des Pages (Fig. 3.22). La Table des Hyperpages contient les pointeurs sur les Tables des Pages.

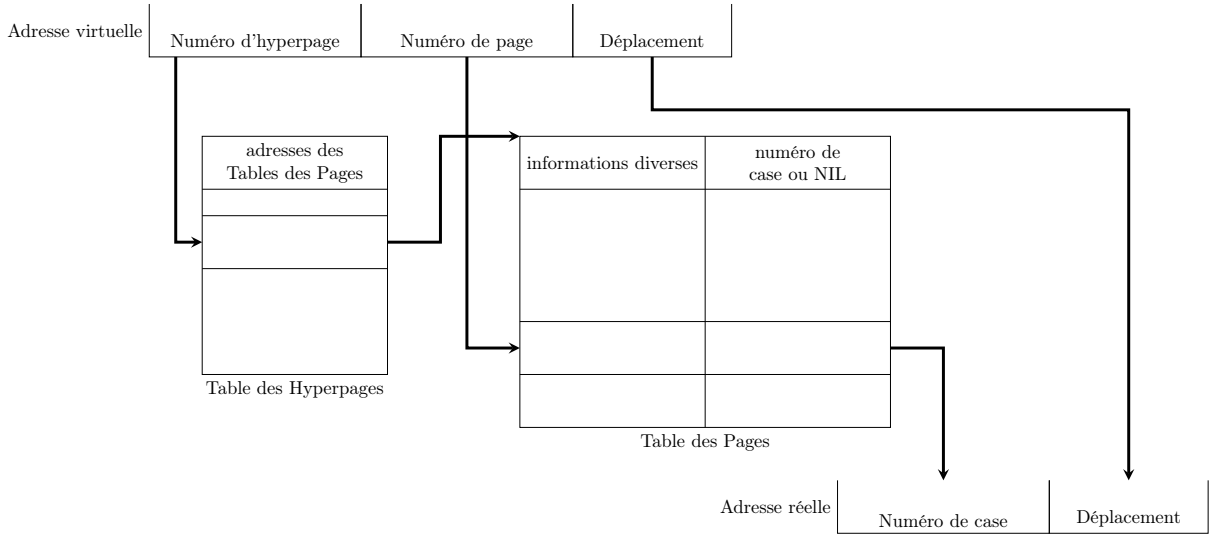


FIGURE 3.22 – Traduction d'une adresse avec une pagination à deux niveaux.

3.5.7 Récapitulatif translation d'adresse

Soit une adresse virtuelle codée sur $m = 14$ bits et une adresse réelle codée sur $n = 12$ bits, m et n ne sont pas forcément égaux car généralement l'espace virtuel est plus grand que l'espace réel. La taille d'une page est de 64 octets et la TLB est un 4-way set-associative cache à 16 entrées.

A partir de l'adresse virtuelle il est alors possible d'extraire les quantités suivantes (cf. figure 3.23) :
— VPO : déplacement dans la page virtuelle (Virtual Page Offset),
— VPN : numéro de page virtuelle (Virtual Page Number),
— PPO : déplacement dans la case physique (Physical Page Offset),
— PPN : numéro de case réelle (Physical Page Number),
— TLBI : indice dans la TLB (TLB index),
— TLBT : préfixe de discrimination dans la TLB (TLB Tag).

Une fois l'adresse réelle obtenue, il est possible d'extraire les informations classiques relatives au cache, sachant dans cet exemple que le cache est de type direct-mapped à 16 blocs et 4 octets par blocs (adressé par l'adresse physique), cf. figure 3.24 :
— CO : numéro d'octet dans le bloc du cache (Cache Offset),
— CI : indice dans le cache (Cache Index),
— CT : préfixe de discrimination dans le cache (Cache Tag).

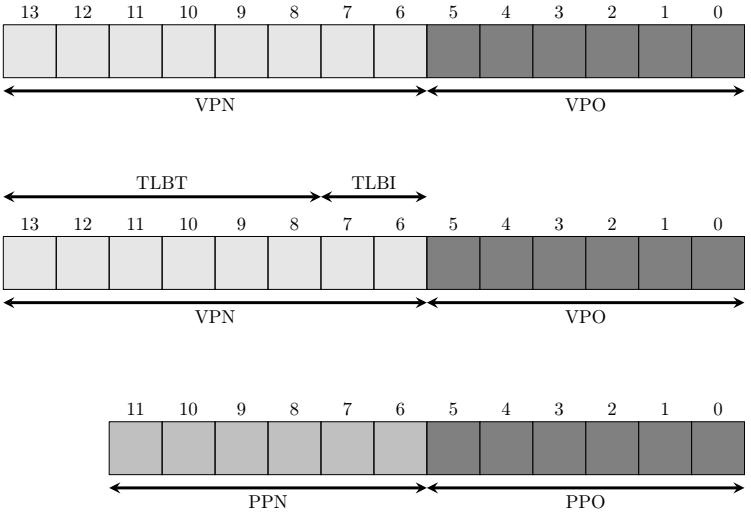


FIGURE 3.23 – Découpage des adresses virtuelle et physique.

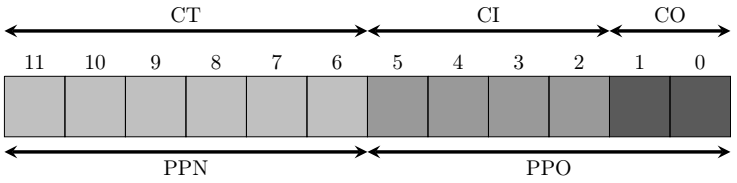


FIGURE 3.24 – Découpage de l'adresse physique pour l'accès au cache.

Chapitre 4

Gestion des fichiers

Un fichier est une collection d'un nombre quelconque d'articles de taille fixe ou variable. En général, le système de gestion des fichiers utilise deux unités :
— Une unité de transfert pour les entrées/sorties physiques,
— Une unité d'allocation pour la mémoire externe.

Dans le système Unix, ces deux unités sont de même taille et sont appelées blocs. La mémoire externe est constituée de l'ensemble des supports contenant les fichiers.

4.1 Gestion de l'espace disque dans le système Unix System V

L'implantation des mécanismes de la gestion des fichiers diffère sensiblement entre la version System V et la version BSD 4.3.

Dans le système Unix, les fichiers sont perçus par l'utilisateur comme une suite non structurée d'octets. Physiquement, un fichier est implanté par un ensemble de blocs. La taille d'un bloc est un paramètre du système, dans cette présentation, nous la fixons à 512 octets.

A chaque fichier est associé un descripteur unique, l'*i-node*, et c'est dans cet *i-node* que sont stockés tous les renseignements concernant le fichier.

4.1.1 Le système de fichiers Unix System V

4.1.1.1 Structure d'un l'i-node

Un i-node est une zone de 64 octets, un bloc disque contient donc exactement 8 i-nodes. La lecture d'un seul bloc suffit pour accéder à n'importe quel i-node car il n'y a jamais de chevauchement d'un i-node sur deux blocs.

L'i-node est structuré en 9 champs :

Nom	Adresse	Taille	Signification
`di_mode`	0	2	Type du fichier et mode de protection
`di_nlink`	2	2	Nombre de liens vers ce fichier
`di_uid`	4	2	Numéro d'utilisateur du propriétaire
`di_gid`	6	2	Numéro du groupe du propriétaire
`di_size`	8	4	Nombre d'octets du fichier
`di_addr`	12	40	Adresses disque des blocs du fichier
`di_atime`	52	4	Date du dernier accès
`di_mtime`	56	4	Date de la dernière modification
`di_ctime`	60	4	Date de création

4.1.1.2 Détail des différents champs d'un i-node

— `di_uid` et `di_gid` : ces deux champs contiennent les identificateurs d'utilisateur et de groupe du propriétaire du fichier. Ces numéros sont déclarés dans le fichier `/etc/passwd` à la création d'un utilisateur.
— `di_atime`, `di_mtime`, `di_ctime` : les dates correspondantes sont mesurées en secondes depuis l'origine fixée au 01/01/1970 à 0h, codée sur 4 octets. Le temps maximum codable est d'environ 138 ans.
— `di_mode` : champs de 16 bits contenant les renseignements concernant le type de fichier et son mode de protection. Il est découpé en 5 zones (cf. figure 4.1).
Dans la notation utilisée, le caractère `*` représente un bit quelconque et les différentes possibilités peuvent être combinées. Par exemple, pour un champs autorisation, la combinaison `1 1 0` indique les possibilités de lecture et d'écriture ainsi que l'interdiction d'exécution.

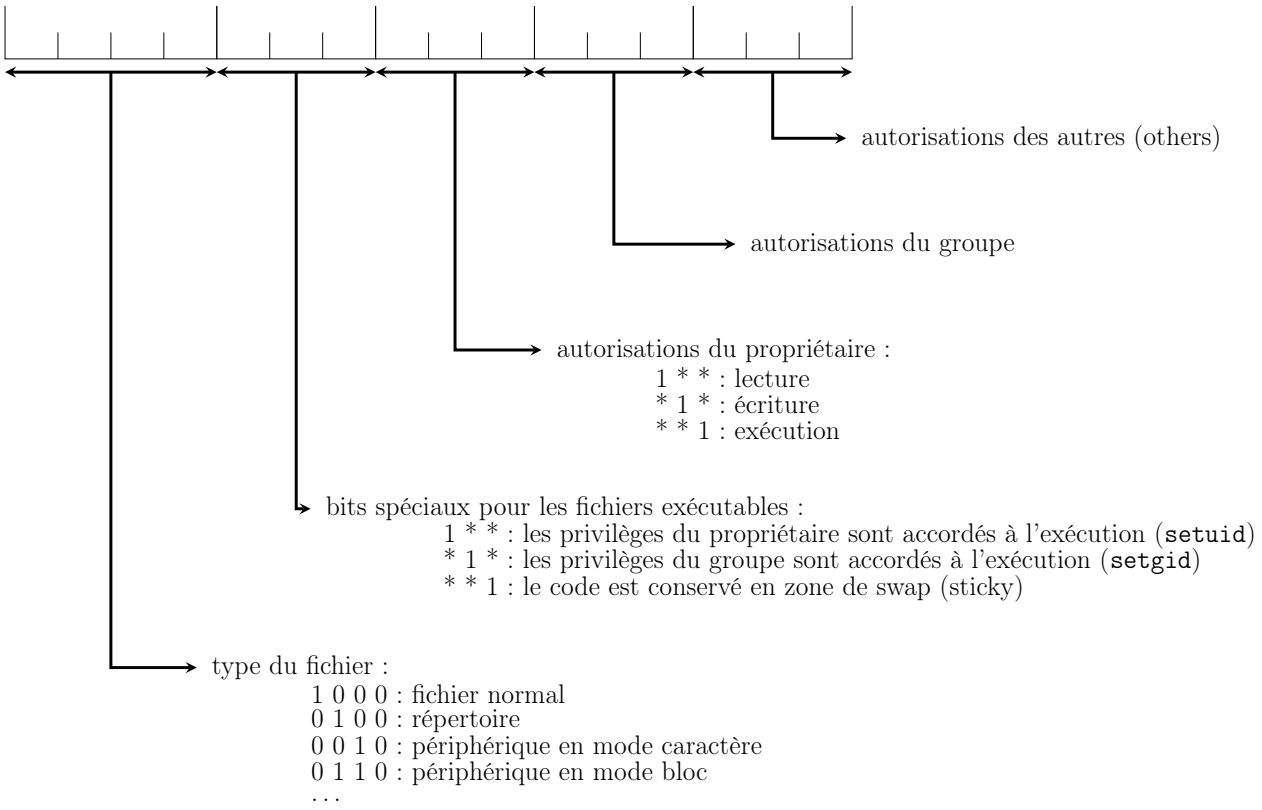


FIGURE 4.1 – Champs mode et droits d'un i-node.

— `di_addr` : champs de 40 octets permet de localiser précisément les blocs du fichier. Il contient 13 adresses disque de 3 octets chacune, le 40ième octet n'est pas utilisé (cf figure 4.2). Les 13 adresses sont utilisées de la manière suivante :
  — Les 10 premières : adresses des 10 premiers blocs du fichier ;
  — 11ième : adresse d'un bloc qui contient les adresses des 128 blocs suivants, mécanisme d'indirection simple ;
  — 12ième : adresse d'un bloc avec une double indirection ;
  — 13ième : adresse d'un bloc avec une triple indirection.

On peut déduire de cette structure que la taille maximum d'un fichier est de 1082201087 octets.

4.1.1.3 Les fichiers répertoires (*directory*)

Dans le système Unix, un répertoire est un fichier comme les autres dans le sens où sa structure sur disque est la même : elle utilise les blocs et un i-node.

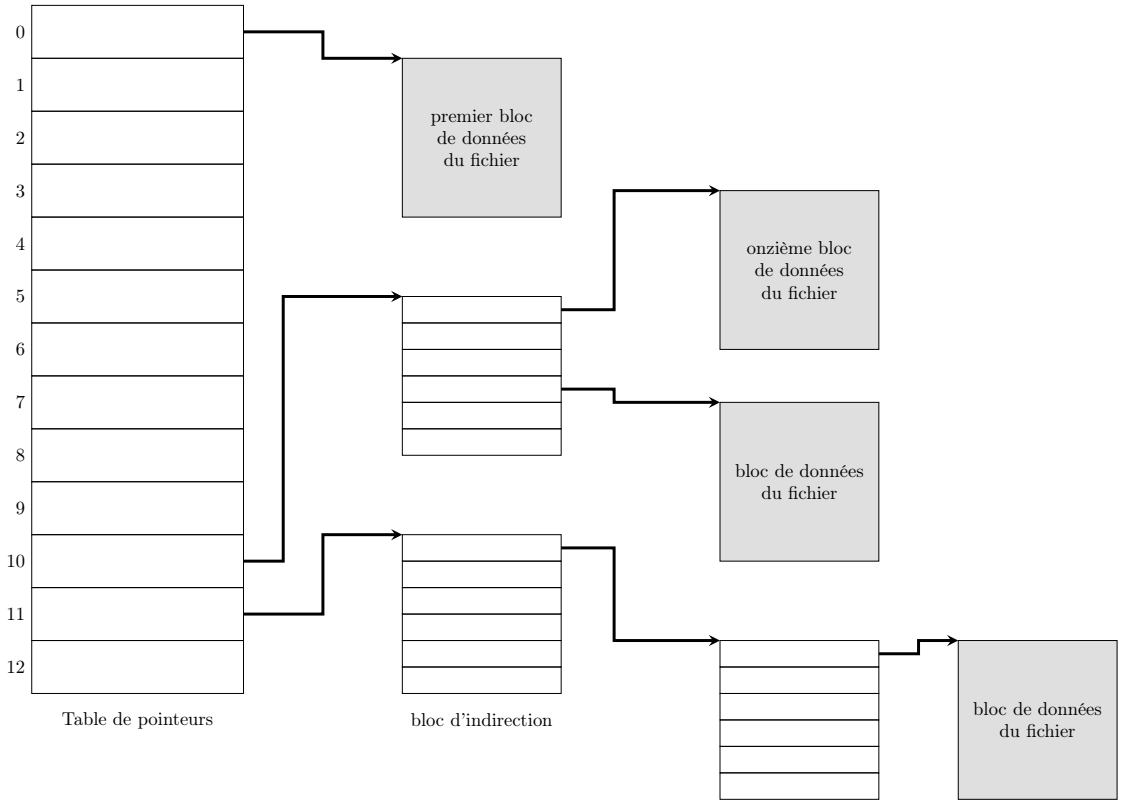


FIGURE 4.2 – Organisation d'un i-node.

Contenu d'un fichier répertoire Un répertoire est une liste d'entrées de 16 octets chacune, la taille d'un fichier répertoire est donc un multiple de 16. Une entrée d'un répertoire désigne un fichier ou un répertoire *fils* et est composée de deux champs :

— `d_ino` : numéro d'i-node du fils (2 octets),
— `d_name` : nom du fils (14 octets).

Le répertoire est donc une table effectuant la correspondance entre le nom choisi par l'utilisateur et un numéro d'i-node choisi par le système.

Un répertoire possède toujours deux entrées qui sont "`.`" et "`..`". L'entrée "`.`" est associée à l'i-node du fichier lui-même et "`..`" à celle de son père.

Si le répertoire est la racine du système de fichier, ses deux entrées "`.`" et "`..`" sont associées au même i-node, l'i-node numéro 2 par convention pour toutes les racines des systèmes de fichiers (*file system*). L'arborescence du système de fichiers sous Unix peut être représenté par le schéma de la figure 4.3.

Particularité d'un répertoire Il existe une contrainte essentielle au niveau des répertoires : aucun utilisateur, y compris le super-utilisateur, ne peut écrire directement dans un fichier répertoire. C'est-à-dire qu'il est impossible d'ouvrir un répertoire en écriture. Il n'existe que deux possibilités pour modifier le contenu d'un tel fichier :

— Utiliser les commandes `rm`, `rmdir`, `mv` ...
— Travailler directement sur le support en ouvrant le fichier spécial associé au disque physique en lecture/écriture.

D'autre part, certains champs de l'i-node d'un répertoire possèdent une sémantique particulière :

— `di_mode` : la signification des autorisations est la suivante :
  — Lecture : le fichier répertoire est accessible en lecture, il est possible d'accéder à la liste de ses entrées ;
  — Ecriture : l'ajout ou la suppression d'une entrée est autorisée, le système accepte d'écrire dans

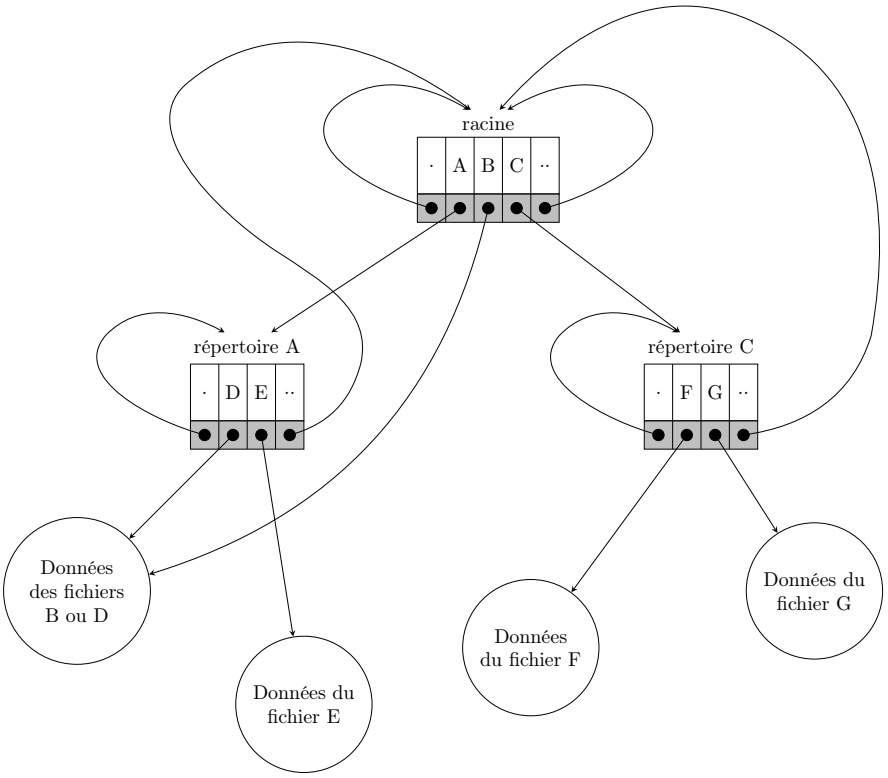


FIGURE 4.3 – Les pointeurs de l'arborescence des fichiers sous Unix.

le fichier répertoire ;
— Exécution : la recherche d'une entrée est autorisée.
— `di_nlink` : pour un fichier `f` d'i-node `if`, `if.di_nlink` indique le nombre de répertoires faisant référence à l'i-node `if` dans leurs entrées et le nombre de liens crées par l'utilisateur.
Pour un répertoire `r` d'i-node `ir`, `ir.di_nlink` indique également le nombre de références faites à `ir`, c'est-à-dire :
— Le nombre de répertoires fils de `r`, car leur entrée `".."` fait référence à `ir` ;
— `r` lui-même à cause de son entrée `"."` ;
— Le père de `r` qui fait nécessairement référence à `r`.

On peut noter que dans le cas d'un répertoire, `di_nlink` correspond au nombre de liens crées par le système.

Mécanisme d'ajout et suppression d'une entrée La suppression d'une entrée dans un répertoire ne libère en aucun cas de l'espace disque, même si cette entrée était la dernière d'un bloc. Le champs `d_ino` de l'entrée est simplement mis à `0`.

A la création d'une nouvelle entrée, le système vérifie s'il n'existe pas une entrée libre et dans ce cas la table du répertoire n'augmente pas en taille. Si aucune entrée n'est libre il ajoute un élément en fin du fichier répertoire.