Faculté
des Sciences
UMONS

Systèmes d'Exploitation

S. Dynerowicz

Bachelier en Sciences Informatiques

Année académique 2022-2023

Table des matières

Préface

Le système d'exploitation constitue la fondation logicielle, souvent oubliée, sur laquelle repose tout système d'information. Celui-ci prend en charge la gestion de nombreuses considérations bas-niveau, tant logicielles que matérielles, afin d'affranchir le programmeur de ces dernières et lui permettre de se focaliser sur la résolution du problème qui l'intéresse.

Bien qu'il soit requis dans n'importe quelle situation où une application doit s'exécuter sur un système constitué de composantes matérielles, les problématiques de gestion des ressources auxquelles le développeur d'un système d'exploitation doit faire face se retrouvent de manière . Les stratégies adaptées pour les solutionner dépendent souvent des hypothèses particulières ou des conditions de fonctionnement de système considéré. L'exploration de ces problématiques ainsi que l'étude de ces stratégies constitue l'objet du présent document.

Ce syllabus reprend les notes du cours de *Systèmes d'Exploitation* de deuxième année du bachelier en Sciences Informatiques de l'Université de Mons.

Seweryn Dynerowicz.

Août 2022.

Chapitre 0 Introduction

Vue d'ensemble

Dans ce chapitre, nous allons couvrir la vue d'ensemble des considérations qui nous intéresserons de manière plus détaillée par la suite. L'objectif est de présenter à un haut niveau les services fournis par le système d'exploitation aux programmes ainsi que les ressources du système et leurs considérations de gestion. Le lecteur ne doit pas s'inquiéter s'il se sent perdu car nous aurons l'occasion d'explorer plus en détail et d'articuler ces différents aspects par la suite.

La taille d'un système d'exploitation moderne dépasse de loin la capacité d'un développeur individuel. À titre indicatif, le code source du noyau Linux [1] tourne aux alentours des 32 millions de lignes de code (commentaires inclus). Cette base de code, gérée par de nombreux développeurs à travers le monde, est formée de l'implémentation des mécanismes qui nous intéresseront plus en détail et ne permet pas à lui seul d'exploiter un système physique concret. À cela vient s'ajouter le code d'une distribution construite autour, pouvant contribuer quelques dizaines voire centaines de millions de lignes de code.

La quantité de modèles et de versions de périphériques (*e.g.* GPU, carte réseau) ajoute une autre difficulté au travail du développeur du système d'exploitation. La nécessité d'abstraire les détails de fonctionnement de dizaines de milliers de périphériques en offrant une interface de programmation adaptée et uniforme pour faciliter la vie du programmeur se pose rapidement. À cela vient s'ajouter, l'obligation de gérer les éventuelles anomalies [2] de certaines versions de périphériques de manière transparente pour le programmeur.

Si l'implémentation d'un algorithme constitue déjà une tâche difficile en soi [3], cette tâche serait encore plus difficile si le programmeur devait prendre en charge de manière explicite dans son code la gestion des ressources, les communications avec les différents périphériques impliqués dans sa tâche.

La Figure 1 donne une idée de l'ampleur et de la complexité du noyau Linux. Les différentes composantes matérielles sur lesquelles reposent les différentes composantes logicielles sont reprises sur la ligne inférieure . Chaque colonne regroupe sous une fonctionnalité spécifique les composantes logicielles ainsi que les sous-systèmes qui en relèvent. Chaque ligne reprend les composantes logicielles à un certain niveau de profondeur ; au sommet ce qui est proche de l'interface haut-niveau exposée au programmeur, au fond toutes les interfaces bas-niveau disponibles pour les développeurs du système d'exploitation. Quelques-unes de ces composantes sont mises en évidence et expliquées ci-dessous.

Interrupts core : durant l'exécution du code d'un programme il est possible que survienne un évènement qui nécessite un traitement dédié avant de pouvoir continuer l'exécution du programme. Afin de réaliser la gestion de cet évènement de facon transparente pour le programme (*cfr.* Chapitre 1), son contexte d'exécution doit être préservé pour la durée du traitement dédié. Ce contexte peut être consulté par le code du système d'exploitation pour identifier la nature exacte de l'évènement.

Virtual memory : sur un système moderne 64 *bits*, il est naturel de disposer d'une quantité de mémoire physique inférieure à la quantité de mémoire théoriquement adressable (*i.e.* 16 exbibytes). Afin de permettre à un programme de pouvoir fonctionner correctement malgré cette réalité, un découplage permet de séparer la mémoire en terme de laquelle un programme raisonne (dite virtuelle) de la mémoire physique dont il dispose effectivement à un moment donné (*cfr.* Chapitre 2).

1. Version 5.19-rc1
2. Le matériel peut tout autant présenter des *bugs* dans leur implémentation physique que le code d'un programme.
3. Surtout si la conception d'un algorithme se fait de manière conjointe à son implémentation dans un langage de programmation

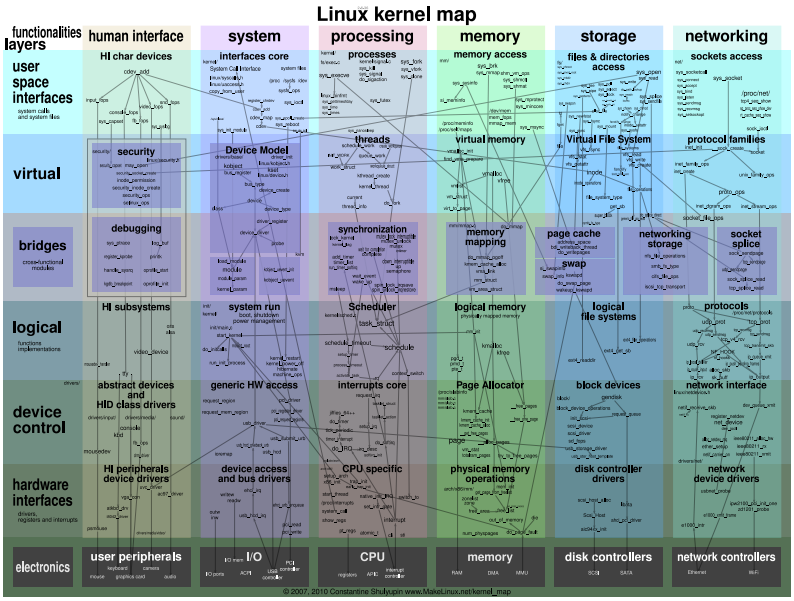


FIGURE 1 – Carte du noyau Linux. Source : `http://www.makelinux.net/kernel_map/LKM.pdf`

Scheduler : dans la mesure où le processeur est partagé entre tous les programmes qui existent à un moment donné [4], le système d'exploitation est confronté au problème de répartir le temps CPU entre ceux-ci. La stratégie utilisée pour sélectionner le programme qui va recevoir le processeur est utilisée par l'ordonnanceur, composante centrale de la gestion des processus (*cfr.* Chapitre 3)

Synchronization : lorsque plusieurs entités d'exécution manipulent des ressources partagées, des incohérences dans les calculs et traitements peuvent apparaître. Ce type de problèmes peuvent être particulièrement difficiles à résoudre et de nombreuses techniques se reposent sur le matériel ou sur le système d'exploitation pour s'en prémunir (*cfr.* Chapitre 4).

Logical file systems : l'espace de stockage disponible sur les différents périphériques (*e.g.* HDD, SSD) doit être organisé pour pouvoir être facilement utilisé par les programmes. Le rôle d'un système de fichier est de maintenir la cohérence des données et de permettre de retrouver le contenu des différents fichiers qui sont stockés dans l'espace de stockage pour un chemin donné (*cfr.* Chapitre 5).

Abstractions de programmation

Un système d'exploitation peut être perçu comme exposant une interface de programmation système aux programmes qui tournent sur une machine. Le *listing* à la Figure 2 présente un programme exploitant une partie de l'interface de programmation sur un système GNU/Linux. Nous aurons l'occasion de découvrir plus en détail ces fonctions systèmes ultérieurement.

L'appel système `open(..)` (*cfr.* ligne 9) permet de demander l'ouverture d'un fichier au système d'exploitation. La valeur de retour de cette fonction, appelée *file descriptor*, représente l'identifiant du fichier ouvert [5]. Ce *file descriptor* est utilisé lors de l'appel système `read(..)` (*cfr.* ligne 17) qui demande au système d'exploitation de lire un certain nombre d'octets depuis le fichier et de les placer à une certaine adresse dans la mémoire. La valeur de retour de cette fonction décrit le nombre d'octets qui ont été effectivement lus depuis le fichier et peut être utilisée dans la condition d'arrêt d'un traitement. Une fois que les traitements sur un fichier sont conclus, l'appel système `close(..)` doit être utilisé pour signaler au système d'exploitation que le fichier en question n'a plus d'utilité pour nous et peut être clôturé.

L'appel système `malloc(..)` (*cfr.* ligne 12) permet de demander au système d'exploitation une allocation dynamique de mémoire. La valeur de retour de cette fonction représente l'adresse à laquelle commence la zone de mémoire de la taille demandée. Une fois le traitement conclu, il est important de libérer la mémoire par le biais de l'appel système `free(..)`.

4. Le gestionnaire de tâches sous Windows et Mac ou la commande `top` sous Linux permettent d'en afficher la liste
5. Pour autant que l'ouverture ait pû être réalisée.

```c
#include <fcntl.h>   // open()
#include <unistd.h>  // read(), write(), close()
#include <stdlib.h>  // malloc(), free()
#include <minimum.h> // minimum()

#define TAILLE 8192

int main(int argc, char* argv[]) {
  int fichier = open("/home/sdy/tableau.bin", O_RDONLY);
  if (fichier == -1) { perror("open"); return EXIT_FAILURE; }

  int *tableau = malloc(TAILLE);
  if (tableau == NULL) {
    perror("malloc"); close(fichier); return EXIT_FAILURE;
  }

  int octets_lus = read(fichier, tableau, TAILLE);

  if (octets_lus >= 4) {
    printf("Minimum trouvé : %d\n", minimum(tableau, octets_lus / 4));
  } else {
    printf("Un tableau vide n'admet pas de minimum.\n")
  }

  // Libération des ressources
  free(tableau); close(fichier);

  return EXIT_SUCCESS;
}
```

FIGURE 2 – Exemple de programme en C.

Dans la mesure où ces appels systèmes traduisent des demandes qui sont faites au système d'exploitation, il est possible que celui-ci détermine l'échec d'une certaine demande. Pour couvrir ces cas de figure, la valeur de retour d'un appel système est utilisée pour indiquer au programme appelant de l'échec de cette tentative. En guise d'illustration, on peut considérer les cas suivants.

— Un *file descriptor* valant -1 indique que l'appel `open(..)` a échoué (*cfr.* ligne 10). Par exemple, si aucun fichier n'existe associé au chemin passé en paramètre ou bien qu'un tel fichier existe mais que le programme ne dispose pas des permissions de le lire.
— Une adresse égale à `NULL` signale que l'allocation n'a pas pû être faite (*cfr.* ligne $13-15$). Par exemple, s'il existe une limite maximale pour la quantité de mémoire détenue par un programme et que celui-ci dépasserait cette limite si l'allocation lui était accordée.

Gestionnaire de ressources

Le système d'exploitation a la responsabilité de gérer les ressources de la machine derrière l'interface de programmation. Le fait de donner toutes les responsabilités de gestion des ressources au système d'exploitation permet au programmeur de s'affranchir des problématiques associées et de se concentrer sur le problème algorithmique qu'il cherche à résoudre. La liste de questions suivante reprend quelques considérations de gestion de ces ressources. Nous aurons l'occasion, dans les chapitres suivants, d'étudier plus en détail ces problématiques.

— Quelles portions de la mémoire sont allouées ou disponibles pour satisfaire la demande d'allocation dynamique ?
— Comment s'assurer qu'aucun autre programme n'accède à la mémoire obtenue à la ligne 12 ?
— Sur quel périphérique de stockage se trouve le fichier `/home/sdy/tableau.bin` ?
— Y est-il stocké d'un seul tenant ou est-il découpé en plusieurs fragments éparpillés ?
— Comment communiquer avec le contrôleur de ce périphérique pour récupérer les fragments du fichier ?
— Le programme a-t-il le droit d'accéder à ce fichier suivant les modalités qui l'intéressent ?

De manière plus générale, le système d'exploitation est en charge de répartir les ressources matérielles de la machine entre les différents programmes qui en ont besoin. Cette répartition peut viser à garantir une équité entre les programmes

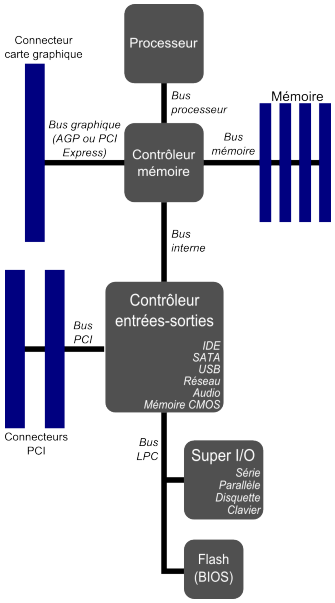


FIGURE 3 – Organisation des composantes matérielles.

ou bien une adéquation avec leurs exigences au vu de leurs besoins ou de leur niveau de priorité. Différentes stratégies peuvent être implémentées dans le systéme d'exploitation à cet effet. Nous nous intéresserons à trois ressources principales dans notre étude des concepts des systèmes d'exploitation.

Processeur

Le processeur constitue l'entité en charge d'exécuter des instructions et, de ce fait, de réaliser des traitements. Un processeur est caractérise par son jeu d'instruction (*i.e. Instruction Set Architecture*) qui décrit l'ensemble des instructions exécutables ainsi que leur format d'encodage (*i.e.* opcode et opérandes). Ces instructions peuvent être classifiées selon leur type : arithmétique (*e.g.* `add`, `div`), logique (*e.g.* `and`), accès mémoire (*e.g.* `lw`, `sw`) ou instructions système. Un processeur moderne est basé sur un structure de *pipeline* qui permet d'occuper différentes parties de celui-ci par différentes instructions.

D'autre part, une architecture processeur incorpore diverses considérations additionnelles. Le processeur peut fournir un jeu d'instruction *RISC* ou *CISC* selon la complexité des opérations réalisées par ses instructions. Parmi les responsabilité de gestion, nous nous intéressons à deux d'entre elles qui sont prises en charge par le système d'exploitation avec l'aide du processeur.

Exécution de programme

La première responsabilité de gestion consiste à surveiller l'exécution des instructions d'un programme lorsqu'il utilise le processeur. Étant donné que tant le système d'exploitation et les programmes doivent se partager le processeur pour effectuer des traitements, il est nécessaire de suspendre l'exécution du programme en cours lorsque survient un évènement particulier.

Le processeur peut réagir au **mauvais comportement** du programme en effectuant un déroutement vers une fonction du système d'exploitation qui prendra en charge son traitement. Dans ce contexte, un mauvais comportement peut être, par exemple, une division par zéro, un accès mémoire vers une adresse considérée comme invalide ou encore une tentative d'exécuter une instruction privilégiée.

Le processeur peut aussi réagir à une **anomalie** lors de l'exécution de l'instruction courante qui n'est pas fatale à la bonne continuation de son exécution. Néanmoins, une telle anomalie nécessite un traitement pour corriger les circonstances qui l'ont provoquée pour ensuite retenter son exécution.

Le processeur peut également réagir à un **appel système** par le programme pour brancher vers la fonction correspondante du système d 'exploitation. Lorsqu'un programme a besoin d'obtenir de la mémoire additionnelle ou d'effectuer une écriture vers un fichier, le système d'exploitation doit prendre le relais pour donner réponse à cette demande.

Un dernier cas qui n'est pas directement lié à l'exécution du programme mais qui néanmoins l'affecte est la survenance d'une **interruption** provenant d'un périphérique externe au processeur. Dans ce cas de figure, il est nécessaire d'effectuer un traitement propre à la raison liée à cette interruption. Par exemple, une interruption du clavier nécessite de réaliser un échange avec le contrôleur du clavier pour déterminer la touche qui est concernée et d'identifier le programme qui en est le destinataire.

Ordonnancement de programmes

La seconde responsabilité de gestion qui incombe au système d'exploitation est la répartition du temps processeur entre les différents programmes qui existent sur le système. Étant donné que les programmes doivent se partager le processeur, qu'il soit *monocore* ou *multi-core* pour pouvoir progresser dans leur code, il est nécessaire de répartir les cycles d'exécution entre ceux-ci.

La problématique de l'**ordonnancement** consiste à trouver un critère de sélection du prochain programme qui va recevoir le processeur lorsque celui-ci devient libre. Selon le type de système et le profil de la charge de travail, différentes stratégies d'ordonnancement peuvent être utilisées. Afin d'évaluer la qualité d'une implémentation particulière d'ordonnanceur, différentes métriques peuvent être appliquées. Par exemple, on peut chercher à minimiser le temps moyen entre début et fin d'un programme ou à respecter l'adéquation entre les besoins d'un programme le temps qu'il reçoit pendant un intervalle donné.

Mémoire

La mémoire constitue l'espace dans lequel réside les données de travail des programmes. Cet espace est typiquement organisé suivant une hiérarchie dont chaque niveau existe physiquement sur une composante spécifique du système. Dans l'ordre, nous discutions brièvement de ces différents niveaux suivant la distance qui les séparent des registres.

Les mémoires **caches** (*e.g.* type $L1$, $L2$) sont utilisées pour servir d'intermédiaire entre les registres du processeur et la mémoire principale. Cet espace est utilisé pour stocker de manière temporaire les données qui sont fréquemment accédées par le processeur. La notion de cache repose sur le principe de localité qui stipule que lorsqu'un morceau de code accède à un emplacement dans la mémoire principale, en général les accès suivants se feront à des addresses qui sont proches de ce dernièr accès. Ce principe est applicable tant pour les données (*e.g.* accès dans un tableau) que pour le code (*e.g.* instructions du programme).

La mémoire **principale** (*e.g.* RAM, GPU) est utilisée pour stocker les données de travail (*e.g.* tableaux, variables, *stack*) d'un programme lorsqu'il n'y a pas de registres disponibles. La mémoire **secondaire** (*e.g.* HDD, SSD) peut être utilisée comme espace temporaire (*e.g. swap*) s'il n'y a pas de place en mémoire principale pour stocker toutes les données sur lesquelles travaille un programme. Pour pouvoir accéder à la mémoire, un système d'adressage est utilisé pour marquer chaque case utilisable. Pour un système d'adressage 32 bits, il est possible d'adresser 4 GiB de données (*e.g.* `0x00000000` à `0xffffffff`).

En pratique, les besoins d'un programme peuvent évoluer au fur et à mesure qu'il progresse dans l'exécution de son code. De ce fait, il n'est pas nécessaire de lui allouer l'intégralité de la mémoire dont il a besoin. Cette approche, appelée *allocation dynamique*, permet de gérer de manière plus flexible la quantité de mémoire disponible.

D'autre part, les besoins d'un programme peuvent dépasser la quantité de mémoire disponible. Par exemple, pour un système avec adressage en 32 bits muni de 2 GiB de mémoire principale, il sera impossible d'avoir l'intégralité de l'espace adressable présent en mémoire principale. Une solution à ce problème, appelée *mémoire virtuelle*, consiste à charger/décharger les portions de données qui ne sont pas activement utilisée pour autant que les accès mémoire puissent être catégorisés ; selon qu'ils sont demandés vers des données présentes ou non.

Protection de la mémoire

La première responsabilité de gestion consiste à contrôler les accès mémoire afin de garantir qu'un programme n'accède qu'à la mémoire qui lui est attachée par le système d'exploitation. Étant donné que tant le système d'exploitation et les programmes doivent se partager la mémoire pour stocker leurs données, il est nécessaire de garantir que chacun reste dans les limites son espace mémoire. Ceci permet d'éviter qu'un programme ne puisse lire des données (*e.g.* clefs de chiffrement) ou les modifier.

Dans la mesure où le processeur est la composante qui réalise les accès mémoire en exécutant des instructions (*e.g.* `lw`, `sw`), celui-ci est en charge d'effectuer des vérifications pour s'assurer qu'un accès est autorisé. Ces vérifications sont faites sur base de descriptions des espaces mémoire accessibles à chaque programme du système qui sont maintenues à jour par le système d'exploitation.

Une adresse **valide** représente une adresse qui se trouve dans l'espace accessible au programme en cours d'exécution. Lorsque celui-ci tente d'accéder à une telle adresse (*e.g.* `lw`, `sw`), l'accès est accordé.

Une adresse **indisponible** représente une adresse qui se trouve dans l'espace accessible mais qui correspond à une donnée qui n'est pas présente dans la mémoire principale. Une tentative d'accéder à une adresse indisponible nécessite de charger la donnée en question depuis la mémoire secondaire vers la mémoire principale.

Une adresse **invalide** représente une adresse qui se trouve hors de l'espace accessible au programme en cours d'exécution. La conséquence typique d'une tentative d'accéder à une adresse invalide est de mettre un terme au programme.

Répartition de la mémoire

La seconde responsabilité de gestion est la répartition de la mémoire principale disponible entre les différents programmes qui existent sur le système. Dans ce contexte, la problématique comporte la représentation et le suivi des portions de mémoire suivant qu'elles sont libres ou non et l'identification d'une portion adéquate pour satisfaire à une demande d'allocation dynamique.

Stockage

Le stockage est utilisé pour préserver les données produites par un programme au-delà de son exécution et du fonctionnement de la machine. Lorsqu'un programme se termine, l'ensemble de la mémoire qu'il occupait se retrouve libérée et potentiellement allouée à un autre programme. La notion de fichier permet de représenter un ensemble de données dont la pérennité doit être assurée. Une difficulté importante réside au niveau de la grande diversité de supports physiques existants. Chacun de ces supports sont adaptés à certains scénarios d'utilisation et des modes d'accès qui découlent de leur conception physique.

Le disque dur (*i.e.* HDD), composé d'un ensemble de plateaux et organisé en cylindres, têtes et secteurs est un support couramment utilisé pour le stockage de données. Relativement bon marché par rapport à son espace, il offre des temps d'accès relativement lent.

Le disque à état solide (*i.e.* SSD), composé des circuits intégrés et organisé en blocs et en pages est un support couramment utilisé pour le stockage des programmes et du système d'exploitation lui-même. Son coût plus élevé qu'un HDD s'explique par des temps d'accès plus rapides.

La bande magnétique, support de grande capacité et fiable, est utilisé typiquement dans des solutions de *backups*. Dans ce type de scénario, il est important d'avoir confiance que la copie de sauvegarde sera exploitable dans l'éventualité d'un *crash* conséquent sur un système en production entraînant la perte des données.

Au-delà de la gestion des accès vers ces périphériques, il est également nécessaire de mettre en place une abstraction pour y accéder depuis un programme. L'arborescence de fichiers et de répertoires dans laquelle un chemin peut être utilisé permet de représenter le contenu de l'espace de stockage. Outre le contenu effectif des fichiers, des méta-données à propos de ces fichiers sont également présentes ; taille en octets, dates (création, modification). Parmi ces méta-données se trouve également les informations liées aux permissions (*i.e.* qui peut faire quoi avec ce fichier) qui sont utilisées par le système d'exploitation pour contrôler l'accès au contenu.

La fonction première d'un système de fichiers est de représenter l'arborescence ainsi que toutes les informations attachées aux fichiers et répertoires sur les supports de stockage disponibles. Il est possible d'implémenter également des mécanismes afin de garantir la cohérence des écritures ou de réduire la possibilité de corruption de données.

Typologie de systèmes

Bien que ces trois types de ressources appararaissent dans bon nombre de systèmes, la nature de ces derniers joue un rôle important dans la conception du système d'exploitation. Nous parlons brièvement dans cette section de quelques types majeurs de système et des particularités importantes qui les caractérisent.

Un **mainframe** a pour fonction d'exécuter des *jobs* de façon régulière. Ces *jobs* sont constitué de traitements qui sollicitent intensivement des entrées-sorties vis-à-vis de périphériques de stockage. À cet effet, ils sont bien mieux équipé pour supporter un volume de transferts conséquents ; centaines ou millier de disques, de l'ordre des TiB de mémoire principale. Par exemple, un département de ressources humaines peut utiliser un tel système pour calculer à la fin de chaque mois les fiches de paye de tous les employés d'un organisation. De plus, de tels systèmes offre une gestion de *timesharing* qui permettent à plusieurs utilisateurs de faire tourner leurs *jobs* sur un même système. Le **mainframe** pourrait ainsi être partagé entre un département de comptabilité (qui doit générer sa comptabilité consolidée chaque année) et un département de facturation (qui doit émettre les factures pour les commandes honorées dans la journée). Les performances d'un tel système peuvent être mesurées sur base du nombre de *jobs* réalisés par heure.

Un **serveur** va mettre à disposition un certain service pour des utilisateurs connectés à distance. Par exemple, un serveur d'impression peut être en charge de contrôler un groupe d'imprimantes et de dispatcher les demandes d'impressions et de se charger de la facturation des utilisateurs sur base de leur utilisation du service.

Un **personal computer** représente le cas où un utilisateur humain interagit avec le système au travers d'une interface graphique. Cet utilisateur a la possibilité de faire tourner plusieurs programmes en même temps et de passer de l'un à l'autre sans constater de délai trop important. L'interactivité jouant un rôle central, il est important dans la conception du système d'en tenir compte pour offrir une expérience utilisateur de bonne qualité. Les performances d'un tel système peuvent être mesurées sur base de la latence (*i.e.* délai) entre l'action

de l'utilisateur et la réaction du système.

Un système **mobile** (*e.g.* tablette, *smartphone*) va présenter une similarité au cas précédent tout en étant soumis à des contraintes plus fortes sur les ressources disponibles. La puissance du processeur, la quantité de mémoire principale sont beaucoup plus faibles et de plus un tel système tourne sur une batterie. Ceci joue un rôle important en matière d'économie d'énergie ; il n'est pas acceptable qu'un programme tourne en continu sur le processeur car ceci drainera la batterie très rapidement.

Un système **embarqué** va consister en un logiciel prédeterminé pour exploiter un appareil ou une machine (*e.g.* TV, voiture) et ne prévoit pas de flexibilité (*i.e.* installation d'applications additionnelles). La matériel étant parfaitement fixer, le système embarqué permet d'utiliser l'appareil auquel il est rattaché.

Un système de **senseurs** est utilisé pour surveiller différents paramètres environnementaux d'une zone géographique donnée (*e.g.* bâtiment, parc, forêt) et de fournir ces données de mesures à un central qui en réalise l'aggrégation. Il est ainsi possible de mesurer en continu la température, l'humidité, la pression atmosphérique ou encore de détecter la présence de fumée. Un tel senseur embarque typiquement une unité de calcul de faible puissance (*i.e.* microcontrolleur), une quantité limité de mémoire (principale et secondaire), une ou plusieurs composantes qui permettent de mesurer un paramètre et une composante permettant de transférer l'information via un réseau de communication. Le tout étant alimenté par une batterie, il est désirable d'optimiser la gestion du matériel et la logique de l'application qui tourne dessus pour réduire au maximum la consommation électrique et retarder la nécessité d'intervenir pour remplacer la batterie.

Un système **temps-réel** prend en compte le temps qui s'écoule dans le monde réel et dont le bon fonctionnement dépend du respect des **échéances** associées aux traitements. De ce fait, la question de choisir quel tâche doit s'exécuter en priorité joue un rôle déterminant dans le bon fonctionnement d'un tel système. De même, la gestion correcte de la mesure du temps joue également un rôle crucial pour s'assurer que les traitements ne sont pas faussés par rapport à l'environnement dans lequel il fonctionne.

Architecture

Un système d'exploitation comporte les mêmes éléments qu'un programme classique ; son code décrit les traitements qu'il est susceptible de réaliser, ses données qui reprend les différentes informations décrivant l'état actuel du système ainsi que sa *stack* sur laquelle il peut stocker les données de travail temporaires. Une différence importante est que ce code doit contenir des instructions qui permettent d'altérer la manière de fonctionner du processeur (*i.e.* instructions systèmes). Ces instructions requièrent un niveau de privilège particulier pour pouvoir être exécutée et le processeur alterne entre les niveaux de privilèges au cours de son fonctionnement. Le noyau d'un système d'exploitation contient toute la gestion des ressources que nous aurons l'occasion d'étudier.

Dans un noyau **monolithique**, l'intégralité du code se trouve dans un fichier exécutable unique, chargé d'un seul tenant dans la mémoire au démarrage de la machine par le *bootloader*. Cette architecture est adaptée pour un système dont la configuration matérielle est déterminée. L'intégralité du code s'exécute au niveau de privilège le plus élevé. Ceci a pour conséquence qu'une erreur de programmation peut amener le code du système d'exploitation à exécuter n'importe quoi tout en autorisant les instructions privilégiées.

Dans un noyau **modulaire**, le code est séparé en une base et un ensemble de modules. Bien que la base soit toujours présente en mémoire quand le système fonctionne, il n'en va pas de même pour les modules qui peuvent être chargé lorsqu'ils sont requis ou déchargés quand ils ne sont plus utilisés. Par exemple, si un câble réseau est connecté sur un carte réseau, le système peut réagir en chargant le module qui permet de gérer les échanges par cette interface. Une fois le câble déconnecté, le module en question peut être déchargé de la mémoire. L'empreinte mémoire du noyau peut donc varier au cours du temps, selon l'activité qui est faite sur la machine. Le noyau Linux est un exemple d'une telle architecture.

Dans un noyau de type **microkernel**, la limite où se situe le niveau de privilège est raffinée. Dans la mesure où une partie de la logique d'un noyau consiste à parcourir et maintenir à jour des structures de données (*e.g.* tableaux, listes), il n'est pas nécessaire pour le noyau d'exécuter ce code au niveau de privilège le plus élevé. Les différentes composantes du système peuvent tourner à un niveau de privilège réduit pour éviter de compromettre l'intégrité du système lorsqu'une d'entre elles provoque une erreur. Un serveur de ré-incarnation peut se charger, au besoin, de terminer proprement une composante ayant commis une erreur fatale et de redémarrer une instance fraîche, ce qui permet d'accroître la robustesse et la fiabilité du système au prix d'une perte de performance. Le

système MINIX3 est un exemple d'un tel type d'architecture.

Dans un noyau de type **exokernel**, la gestion des ressources est laissée entièrement aux programmes qui tournent sur la machine. La seule responsabilité prise en charge par un tel noyau réside dans la protection et l'isolation de ces différents programmes. Un exemple illustratif d'une telle architecture peut être trouvé dans les notions de *machines virtuelles*.

Chapitre 1 Évènements

1.1 Composition et exécution de programme

À un haut niveau, un programme est composé de trois composantes principales. Le **code machine** qui est constitué des instructions représentant la logique des traitements que le programme doit réaliser. Un **espace de données** qui contient toutes les données manipulées par le programme. Cette partie contient autant les variables statiques du programme (*e.g.* variables globales, tableaux) mais est également utilisée pour les variables dynamiques (*i.e.* obtenues via un appel `malloc`). Enfin, la *stack* héberge les données de travail courantes du programme (*e.g.* variables locales) lorsqu'elles ne se trouvent pas dans des registres. D'autre part, les informations d'appels de fonctions imbriqués sont également préservées sur le *stack* pour permettre de correctement gérer les retours d'appels imbriqués correspondants.

Ces trois composantes sont liées logiquement au sein du programme par le biais d'adresses qui sont utilisées pour représenter la logique du programme ; branchements de branches *then* ou *else*, appels et retours de fonction, accès aux cases d'un tableaux ou encore manipulations de la *stack*. Nous étudierons plus en détail la nature et la signification de ces adresses dans le chapitre 2.

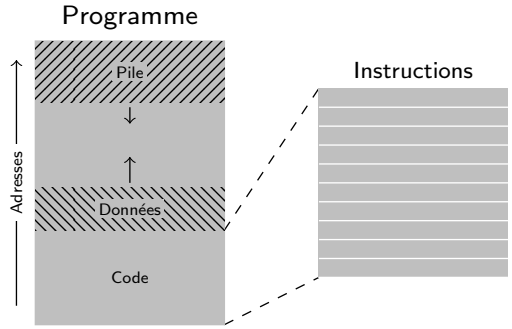


FIGURE 1.1 – Composition d'un programme

Pour pouvoir s'exécuter, ces trois composantes doivent être placées dans la mémoire principale pour que le processeur puisse y accéder. Dans son fonctionnement basique, un processeur travaille suivant une approche *fetch-decode-execute*. Dans un premier temps, le processeur va récupérer l'instruction courante depuis la mémoire principale en vue de son éxécution. Dans un second temps, le processeur va décoder cette instruction pour déterminer ce qu'il est censé faire comme traitement. Dans un troisième temps, le processeur va effectivement l'exécuter en mettant à jour le contenu des registres et/ou de la mémoire principale selon l'instruction qui doit être réalisée.

À tout moment, l'indicateur d'instruction courante (*a.k.a. instruction pointer*, *program counter*) contient l'adresse de l'instruction courante. Cet indicateur pointe vers une instruction dans le code où l'exécution du programme est arrivée. Suite à l'exécution d'une instruction, l'indicateur est mis à jour ; soit par incrémentation constante (*i.e.* séquence d'instruction), soit par assignation d'une valeur particulière (*i.e.* branchement).

Au fur et à mesure que l'exécution d'un programme progresse et que le processeur exécute ses instructions, l'état de ce programme évolue. Lorsque le programme arrive à une certaine instruction, son état est déterminé par quatres éléments ; les valeurs de tous les registres utilisés par le programme, le contenu de la *stack* (y compris le pointeur vers son sommet), le contenu de l'espace de données utilisé par le programme et enfin l'indicateur d'instruction courante. Si une sauvegarde de ces quatres éléments est réalisée, cette copie représentera un instantané de l'état auquel le programme est arrivé. Pour autant que cet état soit préservé, le programme pourra continuer son exécution là où il était arrivé.

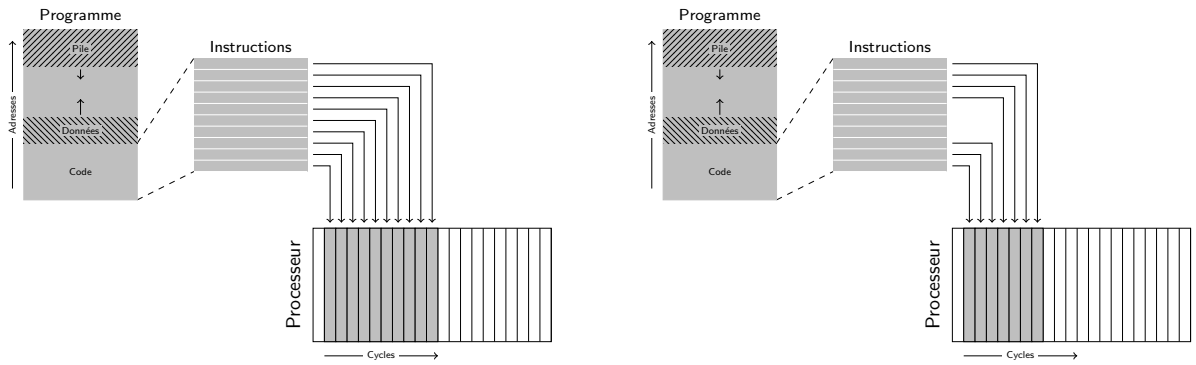


FIGURE 1.2 – Éxécution d'un programme. *À gauche* : séquence d'intsructions. *À droite* : branchement suite à un test.

1.2 Concept de base — Évènement

Lorsqu'un programme est en train d'être exécuté par le processeur, deux questions intéressantes se posent. Premièrement, que faire si l'instruction courante provoque une erreur ? La machine doit-elle être redémarrée ou bien est-il possible de gérer cette anomalie d'une manière propre qui limitera les dégâts au programme qui est en tort ? Deuxièmement, que faire si un périphérique externe fait l'objet d'une activité (*e.g.* frappe au clavier) ? Dans la mesure où le processeur est la seule entité utilisable pour réaliser des traitements, est-il possible de suspendre temporairement le programme en cours d'exécution pour pouvoir le continuer après avoir traité l'activité ? Ces deux situations peuvent être résolues par l'introduction de la notion d'évènements et de leur gestion.

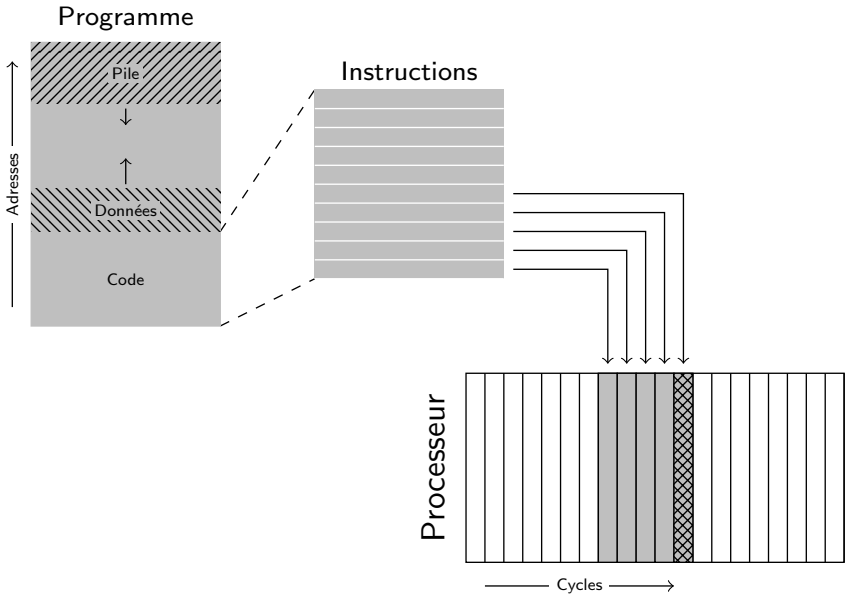


FIGURE 1.3 – Occurrence d'un évènement pendant le $5^{\text{ème}}$ cycle.

Un évènement est un mécanisme implémenté dans le processeur qui lui permet de réaliser un branchement vers un morceau de code précis en réaction à l'occurrence d'un évènement survenu pendant le cycle d'exécution de l'instruction courante. Chaque architecture de processeur prévoit une liste des évènements qui sont susceptibles de survenir dans le cadre de son fonctionnement normal. Il incombe au développeur du système d'exploitation d'implémenter les fonctions de gestion correspondant à chacun de ces évènements possibles. L'ensemble de ces fonctions de gestion forment ce que l'on appelle le **gestionnaire des évènements**.

Les **exceptions** sont des évènements ayant une origine **interne** au processeur. Elles résultent de l'exécution de l'instruction courante et indique l'existence d'une situation anormale. On parle d'évènements **synchrones** car ceux-ci sont provoquées à des moments bien précis dans le cycle d'exécution d'une instruction. D'autre part, ceux-ci sont **prévisibles** étant donnés qu'ils ne peuvent survenir qu'à des instants déterminés. Nous listons ci-dessous quelques exemples d'exceptions.

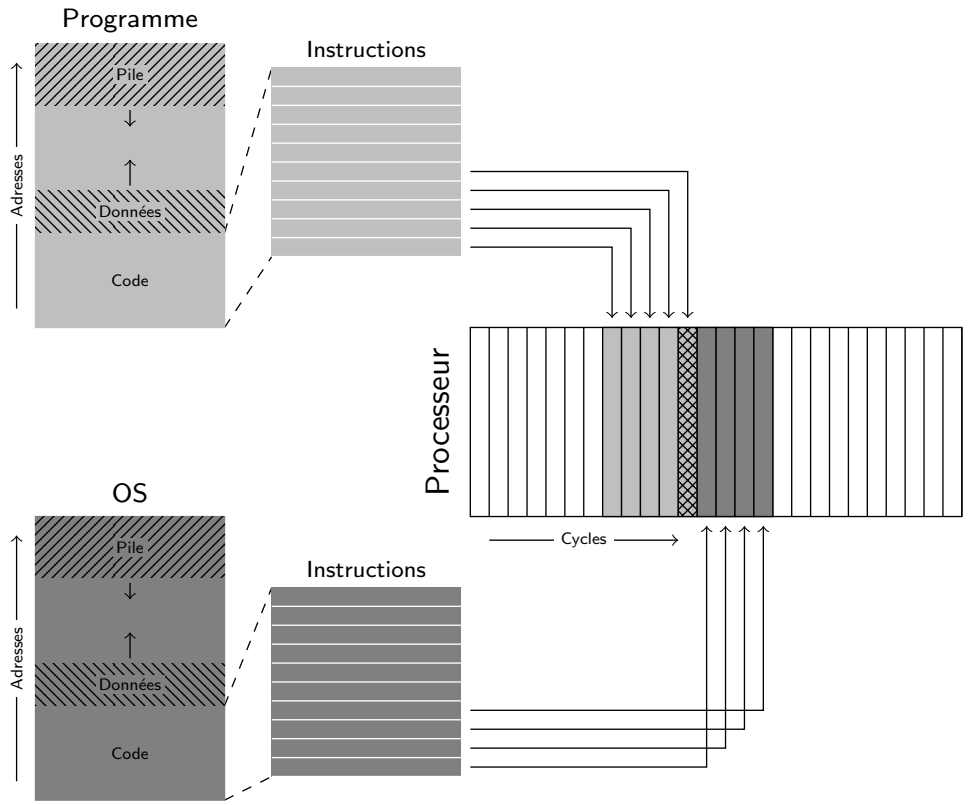


FIGURE 1.4 – Déroutement vers le gestionnaire des évènements suite à l'évènement survenu pendant le $5^{\text{ème}}$ cycle d'exécution.

Code opératoire invalide. Lorsqu'une instruction est en train d'être décodée, il est possible que son *opcode* qui identifie l'opération qui lui correspond, ne soit pas reconnue par le processeur. Ceci peut survenir si le code contient une instruction corrompue ou bien si une instruction d'une version plus avancée d'une architecture est utilisée sur un processeur plus ancien qui ne la supporte pas.

Division par zéro. Lorsqu'une instruction de division entière est réalisée par le processeur, il est nécessaire que la seconde opérande (*i.e.* le dénominateur) soit une valeur non-nulle pour que le résultat de l'opération soit bien défini. Certains types prévoient la gestion de cette anomalie ; par exemple le type `float` du langage C implémente la norme *IEEE-754* qui prévoit une valeur spéciale représentant $\pm\infty$ en guise de résultat.

Instruction privilégiée. Un jeu d'instruction doit permettre au système d'exploitation de configurer le fonctionnement du processeur. À cet effet, des instructions *i.e.* privilégiées sont disponibles pour pouvoir modifier certains aspects que nous rencontrerons au fur et à mesure. Pour éviter qu'un programme quelconque exécute ces instructions, le processeur dispose d'un **niveau de privilège** courant qui est utilisé pour déterminer quelles instructions peuvent être exécutées. Si le processeur est amené à tenter d'exécuter une instruction privilégiée à un niveau de privilège insuffisant, celui-ci peut refuser de l'exécuter en déclenchant une exception.

Instruction non-chargée en mémoire. La mémoire principale étant disponible en quantité limitée, plusieurs techniques existent pour permettre à un programme d'exister partiellement en mémoire. Si le processeur atteint une partie du code qui n'est pas chargée en mémoire, le système d'exploitation peut prendre la main pour corriger cette situation et permettre au programme de continuer une fois cette partie chargée en mémoire.

Les **interruptions** sont des évènements ayant une origine **externe** au processeur. Elles sont provoquées par l'arrivée d'un signal depuis un périphérique. On parle d'évènements **asynchrones** car ceux-ci sont générés par un périphérique à des moments **imprévisibles** par rapport au déroulement de l'exécution des cycles du processeur.

Frappe de clavier. Lorsqu'un utilisateur appuye sur un touche de son clavier, cela a pour effet de fermer un circuit électronique. Ceci pousse le contrôleur du clavier à générer une interruption vers le processeur pour que le système d'exploitation gère celle-ci.

Fichier prêt pour lecture. Une fois qu'un contrôleur de disque a terminé de lire un groupe d'octets sur le(s) disque(s) qu'il gère, il génère une interruption vers le processeur pour la suite des traitements. Le groupe d'octets doit être transféré depuis le **buffer** du contrôleur vers la mémoire principale, à l'endroit où le programme s'attend à trouver ces données.

Réception d'un paquet réseau. Une fois qu'un paquet arrivant du réseau est complètement reçu sur la carte réseau, celle-ci génère une interruption pour que le système d'exploitation réalise la copie depuis le **buffer** de cette carte vers la mémoire principale, à l'endroit où le programme s'attend à trouver ces données.

Erreur du matériel. Si une anomalie est détectée par une composante matérielle (*e.g.* température excessive, niveau de batterie faible), celle-ci peut générer une interruption pour que le système d'exploitation réalise les traitements appropriés.

Signal d'horloge. À côté de l'horloge qui cadence l'exécution des cycles du processeur, il existe une horloge **temps-réel** qui a pour objectif de permettre le suivi du temps qui passe. Cette horloge oscille à une fréquence de 32 kHz et génère une interruption à raison de 32.768 fois par seconde qui a pour conséquence d'incrémenter un compteur du nombre de *ticks* observés depuis que le système d'exploitation a commencé à tourner.

1.3 Conséquences possibles pour le programme

Dans cette section, nous allons nous pencher sur la manière dont le système d'exploitation doit prendre en charge la gestion d'un évènement du point de vue logiciel. Selon l'évènement survenu pendant un cycle, la décision doit être prise concernant le programme qui se retrouve suspendu. Dans le cas **récupérable**, il est possible de continuer l'exécution du programme après le traitement de l'évènement survenu. Dans le cas **irrécupérable**, la faute commise par le programme ne peut être corrigée et il est nécessaire de terminer celui-ci.

1.3.1 Évènement récupérable

Lorsqu'un évènement récupérable survient, il est nécessaire de garantir que l'exécution continuera avec l'état du processeur inchangé. Le code de gestion de l'évènement s'exécutant sur le même processeur que le programme occupait jusque là, le contexte d'exécution de ce dernier doit être préservé pour que les valeurs de tous les registres et l'état de la *stack* reviennent à l'état où ils se trouvaient au moment où l'évènement est survenu.

Comme pour la gestion d'un appel de fonction, une structure de *stack* est utilisée pour sauvegarder (resp. restaurer) le contexte intégralement par des opérations d'empilements (resp. dépilements) entre les registres et cette *stack*. Dans la mesure où le processeur réalise le branchement implicite vers le code de gestion de l'évènement, le programme suspendu n'a pas l'occasion de sauvegarder ses registres utiles. De ce fait, il incombe entièrement au gestionnaire des évènements de réaliser la sauvegarde de ces registres. Selon la complexité du code qui constitue la gestion de l'évènement en question, il est possible de savoir quels registres seront utilisés et de ce fait ne sauvegarder que ceux qui sont pertinents (*e.g.* registres `$r_`, `$s_`, `$t_`). Il est également important de préserver certains registres d'états qui peuvent contenir des informations cruciales (*e.g.* résultat du dernier test) pour garantir la bonne continuation du programme.

Une question intéressante qui se pose concerne la *stack* qui est utilisée par le gestionnaire d'évènements. Si la *stack* du programme suspendu est utilisée, le gestionnaire va venir travailler dessus et placer toutes sortes d'informations sur celles-ci. Lorsqu'il aura terminé ses traitements et que le programme continuera, celui-ci aura accès à ses informations étant donné que le dépilement depuis une *stack* n'efface pas les données qui s'y trouvent. Pour éviter tout problème, certaines architectures de processeur prévoyent un mécanisme de basculement de *stack* qui accompagne le branchement vers le gestionnaire d'évènements.

Enfin, lorsque la continuation est réalisée, la question se pose de savoir à quelle instruction continuer. Dans le cas d'une **reprise**, il est nécessaire de retenter l'exécution de l'instruction qui occupait le cycle du processeur pendant lequel l'évènement est survenu. Dans le cas d'une **poursuite**, il est possible de passer à l'instruction suivante.

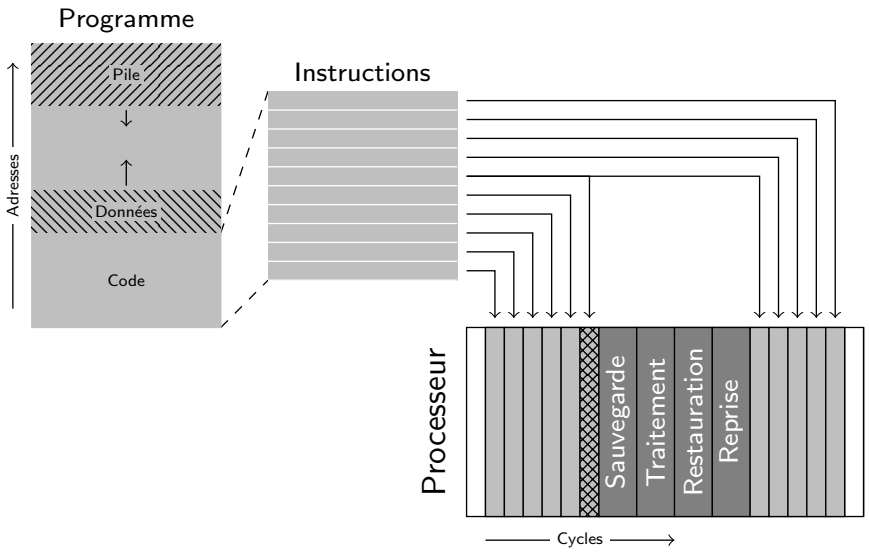


FIGURE 1.5 – Gestion d'un évènement récupérable avec reprise ; sauvegarde du contexte, traitement effectif, restauration du contexte et reprise.

Nous verrons dans le Chapitre 2 qu'il est possible pour un programme d'exister partiellement en mémoire physique avec certaines de ses parties résident en mémoire secondaire. Pendant son exécution par le processeur, un programme peut rencontrer un bloc d'instruction qui n'est pas chargé en mémoire ce qui donnera lieu à un **évènement récupérable avec reprise**. Il est possible de continuer l'exécution de ce programme pour autant que le bloc d'instruction soit chargé en mémoire physique depuis la mémoire secondaire et de reprendre l'exécution à l'instruction que le processeur a tenté d'exécuter. Pour déterminer quel bloc de code est manquant, le système d'exploitation peut consulter l'indicateur d'instruction corrélée à l'évènement.

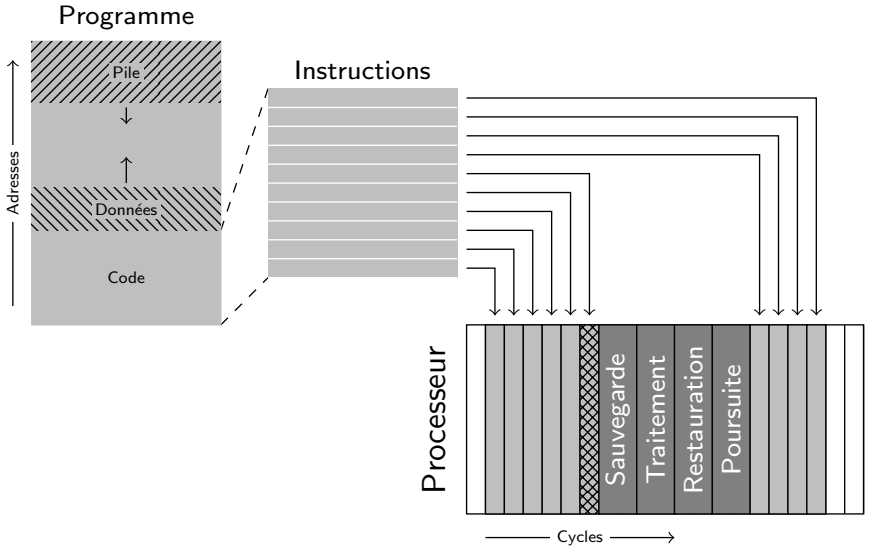


FIGURE 1.6 – Gestion d'un évènement récupérable avec poursuite ; sauvegarde du contexte, traitement effectif, restauration du contexte et poursuite.

De manière générale, les interruptions constituent toutes des évènements récupérables avec poursuite, dans la mesure où leur survenance n'affecte pas le bon déroulement de l'instruction exécutée par le processeur pendant le cycle d'exécution pendant lequel elles surviennent. Du côté des exceptions, une instruction d'appel système (*e.g.* `syscall`, `int`) tombe dans la catégorie des évènements récupérables avec poursuite. Comme pour un appel de fonction en assembleur (*e.g.* `jal`, `call`) après la conclusion du code appelé, le retour doit se faire à l'instruction suivante l'appel.

Lorsque survient une frappe au clavier, une interruption est générée. Le système d'exploitation peut récupérer le code de la touche en lisant sur un port d'entrée/sortie correspondant à un registre du contrôleur du clavier. Ce code doit être traduit en un caractère ou une touche particulière et transféré vers le programme qui en est le destinataire. Dans un système muni d'une interface graphique, la fenêtre active correspond au programme qui est sensé recevoir cette touche.

1.3.2 Évènement irrécupérable

Lorsqu'un évènement irrécupérable survient, l'exécution du programme ne continuera plus. Dans ce cas de figure, il est néanmoins pertinent de réaliser une sauvegarde complète du contexte d'exécution. En effet, le programmeur qui a implémenté ce programme peut utiliser ces informations pour le *débugger* afin de déterminer où se trouve l'erreur dans son implémentation.

Au cours de son exécution, un programme acquiert diverses ressources ; mémoire physique par allocation dynamique (*cfr.* Chapitre 2), descripteurs de fichiers par ouverture (*cfr.* Chapitre 5), *sockets* de communication ouvertes ainsi que toutes les informations de suivi de l'exécution (*cfr.* Chapitre 3). Le programme en question devant être terminé lorsque survient un évènement irrécupérable, toutes ses ressources peuvent être libérées.

Un traitement minimal consiste en l'affichage d'une notification à l'utilisateur qui décrit le problème rencontré avant de proprement clôturer l'exécution. Le contexte sauvegardé peut être écrit dans un fichier (*cfr. coredump*) ou expédié par *mail* vers un serveur de collecte de rapports de *crash*.

Enfin, l'abandon du programme en cours signifie que le processeur se retrouve libéré, rendant possible la continuation d'un autre programme (*cfr.* Chapitre 3). Dans ce cas, le système d'exploitation réalise un **context switch** pour changer le contexte qui sera restauré dans le cadre de la gestion de cet évènement irrécupérable.

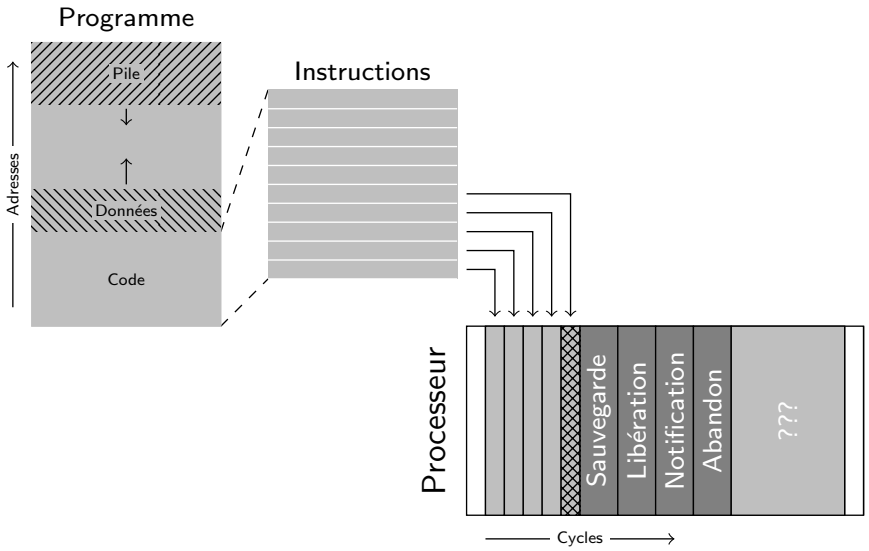


FIGURE 1.7 – Gestion d'un évènement irrécupérable ; sauvegarde du contexte, libération des ressources, notification et abandon.

1.4 Mécanismes de déroutement

Dans cette section, nous couvrons les considérations liées au déroutement à proprement parler que le processeur réalise. Chaque architecture de processeur fournit sa spécification et son implémentation du mécanisme de déroutement ainsi que des considérations associées. Deux techniques possibles sont le déroutement avec registre de cause et la vectorisation. La technique utilisée impacte l'organisation et les responsabilités du code du gestionnaire d'évènements.

1.4.1 Registre de cause

Le registre de cause est utilisé pour encoder les évènements survenus lors du cycle d'exécution courant. Le processeur met à jour pour refléter l'occurrence d'exceptions et/ou d'interruptions. Au terme du cycle d'exécution, le processeur consulte le contenu de ce registre et si au moins un bit d'interruption est positionnée à 1 ou un code d'exception est encodé, le processeur réalise un déroutement vers une adresse précise.

Le code du gestionnaire d'évènements doit lire le contenu de ce registre pour déterminer les traitements qui doivent être réalisés. Il est nécessaire de tester les *bits* des interruptions en attente et de traiter toutes celles pour lesquels le *bit* est égal à 1. D'autre part, le code d'exception stipule quelle anomalie doit être traitée concernant le programme qui s'exécutait. Cette technique permet au système d'exploitation de déterminer entièrement l'ordre de priorité de traitement des différents évènements.

Au démarrage du système, il est important de placer la fonction principale du gestionnaire d'évènements à l'adresse vers laquelle le processeur réalise le déroutement en réaction à un évènement.

Afin de permettre la continuation du programme, l'adresses de l'indicateur d'instruction courante est sauvegardé dans un registre dédié (*e.g. Exception Program Counter* en MIPS32).

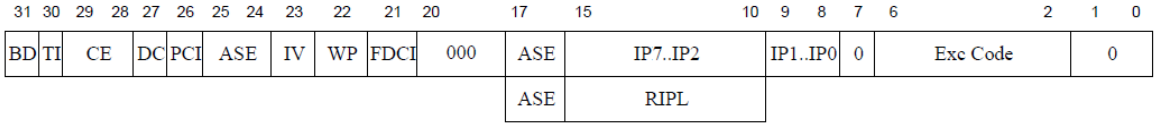


FIGURE 1.8 – Registre de cause en MIPS32. Les *bits* `8` à `15` (*i.e.* `IP0 .. IP7`) encodent les interruptions en attente. Les *bits* `2` à `6` (*i.e.* `Exc Code`) un code d'exception survenue (*e.g.* `4` pour une erreur d'adressage dans un `load` ou un `fetch`, `8` pour un appel système. Source : *MIPS32 Privileged Resource Architecture, Chapter 9, Section 36.*.

1.4.2 Vectorisation

Dans le cadre de la vectorisation, chaque type d'évènement est associé à un numéro unique (*e.g. interrupt vector*). Ce numéro identifie une entrée dans une table de descripteurs (*e.g. interrupt descriptor table*) qui référence la fonction spécifique de traitement à invoquer lorsque ce type d'évènement survient (*a.k.a. interrupt service routine*, *interrupt handler*). Au terme d'un cycle d'exécution, le processeur regarde l'ensemble des numéros d'évènement survenus et invoque, dans un ordre bien déterminé, les fonctions de gestion des évènements correspondantes. Le code d'une fonction de gestion d'évènement doit simplement se focaliser sur l'évènement auquel elle correspond. L'ordre d'invocation étant déterminé pour les exceptions par le processeur.

Au démarrage de la machine, le système d'exploitation doit initialiser le contenu de la table des descripteurs et référencer celle-ci au niveau du processeur en utilisant une instruction privilégiée (*e.g.* `lidt` en `x86`). La Figure 1.9 présente la manière dont le déroutement vers la fonction de gestion est réalisé. Le numéro d'évènement (*i.e. interrupt vector*) sert d'indice dans la table des descripteurs (*IDT*) pour extraire le descripteur correspondant (*i.e. interrupt gate*). Ce descripteur contient l'adresse de la fonction de gestion des évènements vers laquelle brancher (*i.e. offset*). La portion inférieure concerne la segmentation et sera couverte dans le Chapitre 2.

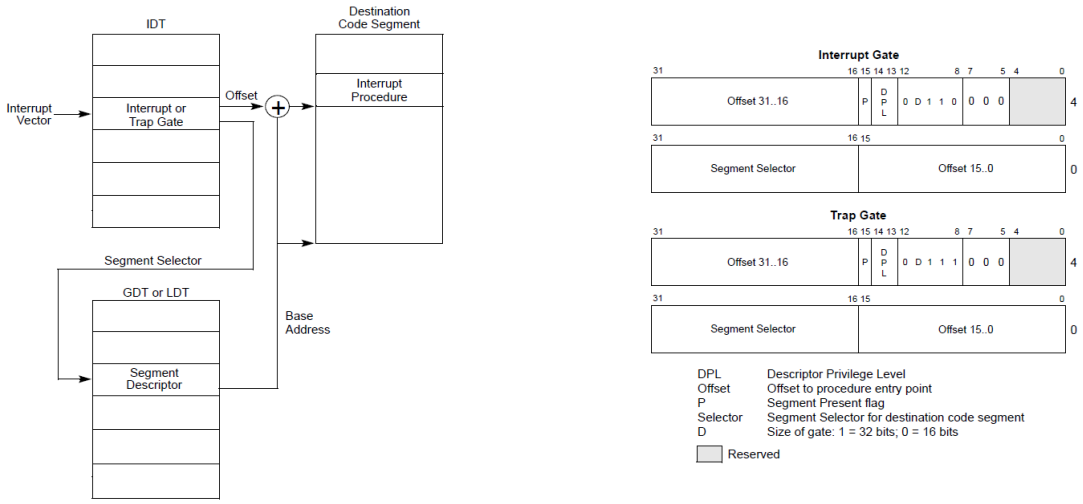


FIGURE 1.9 – Fonctionnement du déroutement sous `x86` avec description de la structure binaire des entrées de la table. Source : *Intel 64 and IA-32 Architectures Software Developer's Manual, Vol. 3, Ch. 6.*

1.4.3 Niveau de privilège

Le jeu d'instruction inclut toutes les instructions qui sont reconnaissables par un processeur implémentant l'architecture qui lui correspond. À côté des instructions arithmétiques et logiques classiques se trouvent des instructions systèmes qui sont utilisées pour modifier la configuration du processeur. Ces instructions devraient en principe uniquement être exécutées par le système d'exploitation. Un niveau de privilège est utilisé par le processeur pour contrôler quelles instructions peuvent être exécutées à quel moment. La spécification d'un processeur décrit la dynamique suivant laquelle ce niveau de privilège change et à quel niveau il est utilisé pour des vérifications. La Figure 1.10 présente l'alternance de niveau de privilège qui accompagne typiquement l'exécution d'instructions par un processeur.

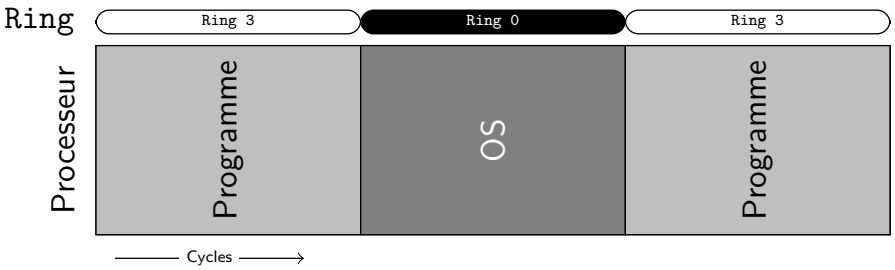


FIGURE 1.10 – Alternance du niveau de privilège suivant l'occupation du processeur. Lorsqu'un évènement survient, le déroutement vers le système d'exploitation s'accompagne d'une élévation du niveau de privilège. Au terme de son traitement, la continuation du programme s'accompagne d'une diminution du niveau de privilège.

1.4.4 Basculement de stack

Lorsqu'un évènement survient, le processeur va dérouter vers une fonction du système d'exploitation qui va commencer à s'exécuter. Cette fonction correspond à un point d'entrée dans le code du système d'exploitation qui est susceptible d'invoquer d'autres fonctions jusqu'à la conclusion des traitements pour l'évènement survenu. De ce fait, tout le code qui va être exécuté manipulera le contenu des registres du processeur pour ses besoins et utilisera un espace de *stack* pour stocker les valeurs qui ne sont pas activement utilisées. Une question se pose quant à la *stack* particulière à utiliser ; la *stack* du programme suspendu ou bien une *stack* dédiée du système d'exploitation.

Lorsqu'un déroutement survient, le processeur peut basculer la *stack* courante, en prenant bien soin de sauvegarder les informations qui permettent au système d'exploitation d'explorer le contenu de la *stack* du programme suspendu (*e.g. stack pointer*). Par exemple, si un programme réalise un appel système, il est susceptible de passer des arguments au système d'exploitation (*e.g.* chemin vers un fichier pour un appel `open`). Ces arguments pouvant être passés via la *stack*, le système d'exploitation doit pouvoir les localiser dans la mémoire principale pour les consulter. De façon complémentaire, au terme du traitement d'un évènement, un basculement inverse doit être

réalisé, la *stack* courante redevient la *stack* du programme, à l'endroit où se situe son pointeur.

La Figure 1.11 illustre la manière dont peut se dérouler ce changement de *stack*. Au moment où survient l'évènement, le pointeur de *stack* a pour valeur $\texttt{sp}_{(\texttt{pgm})}$. Après le déroutement, lorsque le code du gestionnaire d'évènements commence à s'exécuter, le pointeur de *stack* vaut $\texttt{sp}_{(\texttt{os})}$ et le contenu de la *stack* dédiée consiste en une sauvegarde du pointeur de *stack* avant le déroutement (*i.e.* le sommet de la *stack* du programme), l'indicateur d'instruction courante où le programme se trouvait au moment de l'évènement (*i.e.* $\texttt{ip}_{(\texttt{pgm})}$) et un éventuel code d'erreur qui décrit de manière plus fine la nature de l'évènement qui s'est produit. Ces informations pourront être utilisées pour revenir au programme dans le cadre de la continuation.

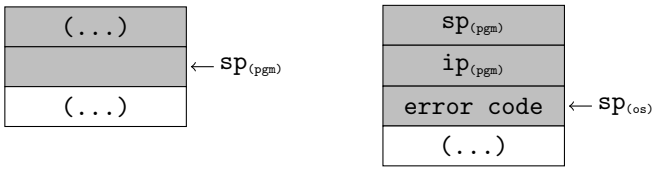


FIGURE 1.11 – Basculement de *stack* dans le cadre d'un déroutement.

1.5 Multiplicité des occurrences

Il est possible que plusieurs évènements se produisent **pendant un même cycle d'exécution** du processeur. De ce fait, il est nécessaire de déterminer selon quel cas traiter cet agrégat (reprise, poursuite ou abandon) ainsi que les traitements qui devront être réalisés. La Figure 1.12 reprend les règles d'agrégation pour deux évènements qui impliquent les règles d'agrégation pour plus de deux évènements.

	P	R	A
P	P	R	A
R	R	R	A
A	A	A	A

FIGURE 1.12 – Règles d'agrégation des cas. Une reprise l'emporte sur une poursuite. Un abandon l'emporte sur une reprise.

Lorsque les deux évènements tombent sous le même cas de figure (resp. deux poursuites, deux reprises ou deux abandons), la conséquence pour le programme est aisément dérivée ; une poursuite, une reprise ou un abandon, respectivement. Dans le cas où les deux évènements sont récupérables mais des deux types différents (une poursuite et une reprise), la nécessité de retenter l'exécution de l'instruction corrélée à l'évènement l'emporte sur la possibilité de passer à l'instruction suivante. Les traitements associés à ces deux évènements doivent être réalisés dans tous les cas. Cependant, si l'évènement récupérable avec poursuite est une interruption, il est envisageable de réaliser son traitement en priorité sur celui de l'évènement avec reprise.

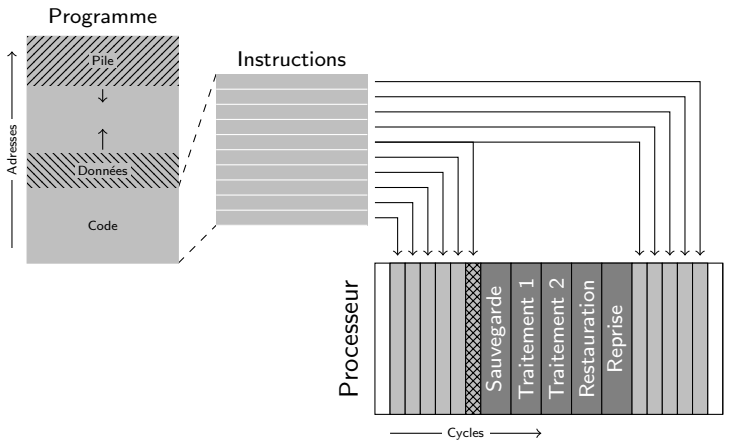


FIGURE 1.13 – Occurrence de deux évènements récupérables ; poursuite et reprise donnent lieu à une reprise.

Dans le cas où l'un des évènements est récupérable avec une poursuite et l'autre est irrécupérable avec abandon, la conséquence pour le programme est l'abandon dont le traitement est réalisé dans tous les cas. Dans

un processeur muni d'une architecture de *pipeline*, l'évènement avec poursuite peut être une exception, auquel cas il n'est pas nécessaire de réaliser son traitement associé étant donné que le programme en question ne continuera plus. En revanche, s'il s'agit d'une interruption, le traitement associé doit être réalisé en toute circonstance.

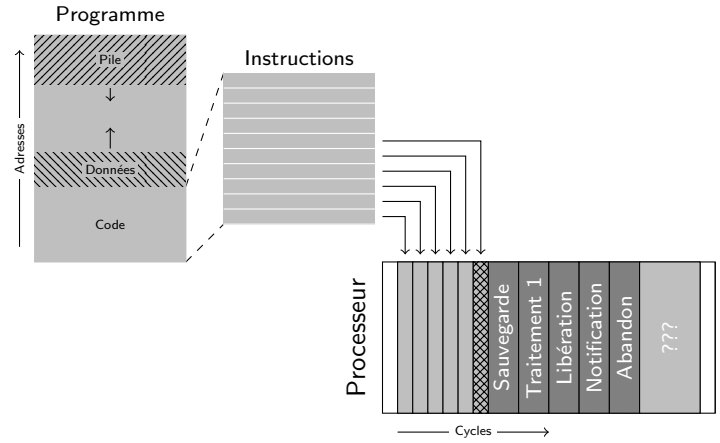


FIGURE 1.14 – Occurrence d'un évènement récupérable (reprise ou poursuite) et un irrécupérable qui donnent lieu à un abandon.

Dans le cas où l'un des évènements est récupérable avec une reprise et l'autre est irrécupérable avec abandon, la conséquence pour le programme est l'abandon dont le traitement est réalisé dans tous les cas. L'évènement avec reprise ne peut être qu'une exception, ce qui requiert un processeur avec une archicture de *pipeline*, auquel cas il n'est pas nécessaire de réaliser son traitement associé étant donné que le programme en question ne continuera plus.

1.6 Gestion ré-entrante des évènements

Un évènement est susceptible de survenir pendant n'importe quel cycle d'exécution du processeur. De ce fait, la possibilité existe qu'un évènement survienne **pendant** la gestion d'un évènement, soit sous la forme d'une exception ou bien d'une interruption. Dans ce cas de figure, il est nécessaire d'adapter le gestionnaire des évènements pour garantir que son imbrication puisse se faire de façon correcte. Dans la mesure où un espace de *stack* est utilisé pour sauvegarder le contexte du programme suspendu, cet espace peut également être utilisé pour sauvegarder le contexte de l'instance du gestionnaire d'évènements qui se retrouve suspendue par l'occurrence d'un nouvel évènement avant la conclusion de ses traitements.

On parle d'un gestionnaire **ré-entrant** lorsque les évènements qui surviennent pendant certaines phases de son exécution provoquent un nouveau déroutement. Les considérations de traitement pour un tel gestionnaire ne sont pas sans rappeler la gestion des appels d'une fonction récursive. Pour garantir que les contextes imbriqués soient sauvegardés de manière cohérente, les interruptions peuvent être désactivées au niveau du processeur pendant les phases critiques ; sauvegarde du contexte, restauration du contexte et continuation. Cette désactivation a pour effet de mettre en attente les interruptions qui surviendrait pendant ces phases. Étant donné que des instances du gestionnaire d'évènements peuvent se retrouver imbriqués, il est important de préserver l'adresse de l'instruction corrélée à chaque niveau.

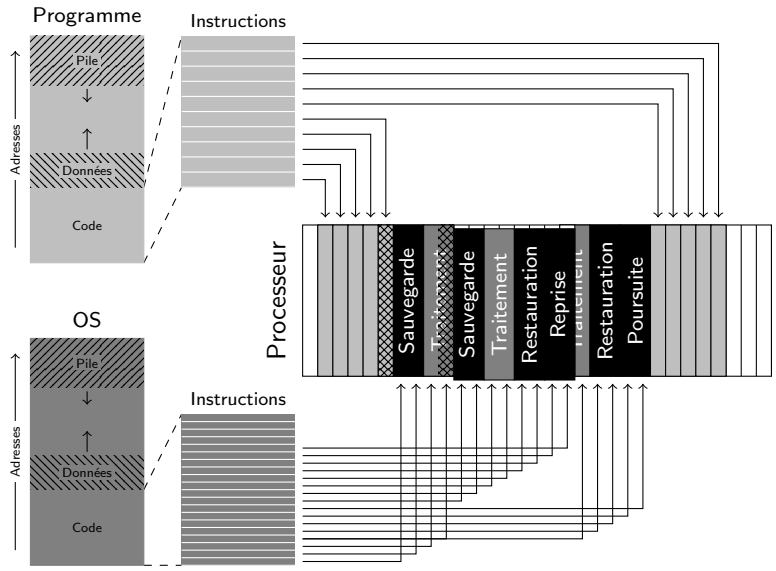


FIGURE 1.15 – Gestionnaire d'évènements ré-entrant. Les phases en noir représentent celles pendant lesquelles les interruptions sont désactivées. La première interruption est supposée être plus prioritaire que la seconde interruption.

1.7 Gestion préemptive des évènements

Dans un système typique, il existe une forme de priorité qui est désirée sur les traitements des différentes interruptions qui peuvent survenir. Un gestionnaire ré-entrant est nécessaire pour pouvoir respecter cet ordre de priorité des traitements mais ne suffit pas à lui seul. Le déroutement **inconditionnel** (*i.e.* provoqué dès qu'un évènement survient) signifie que l'occurrence d'une interruption de basse priorité (*e.g.* réception de données sur la carte réseau) pendant le traitement d'une interruption de haute priorité (*e.g.* frappe au clavier) provoquerait un déroutement. Cette situation, appelée une *inversion de priorité*, correspond au fait que le traitement de basse priorité est exécuté complètement avant le traitement de haute priorité.

La solution à ce problème consiste à introduire une granularité dans l'activation et la désactivation des interruptions. Par contraste avec l'utilisation d'un *bit* unique, comme celui utilisé pour les phases critiques dans la gestion ré-entrante, un **masque d'interruptions** est utilisé pour contrôler quelles interruptions sont actives (resp. inactives). Ce masque comporte autant de *bits* qu'il existe de types d'interruptions et son contenu est combiné avec les *bits* correspondant aux types d'interruptions en attente au terme du cycle d'exécution courant.

Lorsque le gestionnaire des évènements commence son traitement d'une interruption, il positionne les *bits* du masque d'interruption de manière à autoriser un déroutement uniquement par les interruptions plus prioritaires. Au terme de son traitement, l'état du masque est restauré à son état précédent le déroutement.

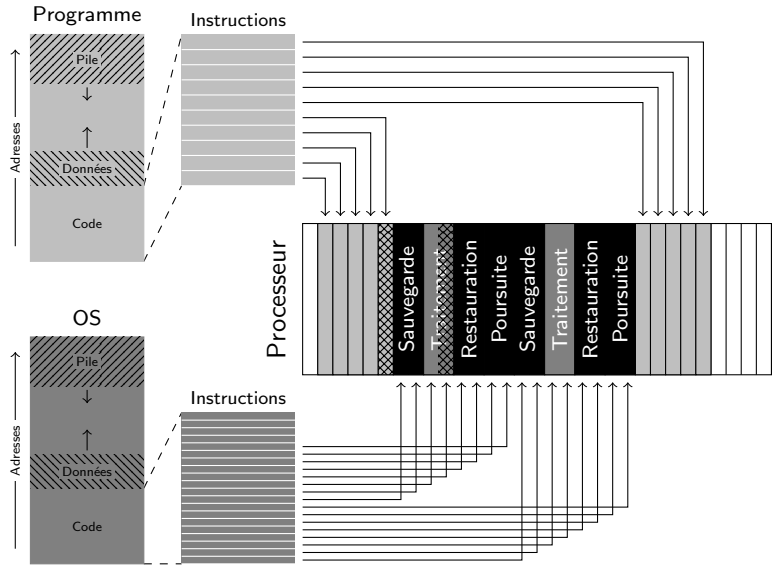


FIGURE 1.16 – Gestionnaire d'évènements préemptif. Les phases en noir représentent celles pendant lesquelles les interruptions sont désactivées.

Chapitre 2 Mémoire

À un haut niveau, un programme est composé de trois composantes essentielles. Le **code machine** qui est constitué des instructions représentant la logique des traitements que le programme doit réaliser. Un **espace de données** qui contient toutes les données manipulées par le programme au cours de son exécution. Cette partie contient autant les variables statiques du programme (*e.g.* variables globales, tableaux) mais est également utilisée pour les variables dynamiques (*i.e.* obtenues via un appel `malloc`). Enfin, la *stack* héberge les données de travail courantes du programme (*e.g.* variables locales) lorsqu'elles ne se trouvent pas dans des registres. D'autre part, les informations d'appels de fonctions imbriqués sont également préservées sur le *stack* pour permettre de correctement gérer les retours d'appels imbriqués correspondants.

Ces trois composantes sont liées logiquement au sein du programme par le biais d'adresses qui sont utilisées pour représenter la logique du programme ; branchements de branches *then* ou *else*, appels et retours de fonction, accès aux cases d'un tableaux ou encore manipulations de la *stack*. En assembleur, ces liaisons sont exprimées par le biais de libellés qui permettent d'accroître la lisibilité du code. Cependant, une fois traduit en code machine (*i.e.* en mots binaires), ces libellés sont remplacés par la place à des adresses qui sont utilisées par le processeur pour localiser le contenu dans la mémoire principale.

La Figure 2.1 présente un exemple de programme écrit en C qui est constitué d'une seule fonction. Cette fonction (*i.e.* `minimum`) identifie le minimum d'un tableau qui existe sous forme de variable globale. L'approche consiste à considérer le premier élément du tableau comme étant le minimum (*cfr.* ligne 5) pour ensuite parcourir les 7 éléments restants et de comparer successivement chacun d'entre eux avec le minimum courant (*cfr.* lignes 6-8). Le code équivalent en MIPS à droite présente le fait qu'il existe un label `TABL` qui correspond aux mots binaires encodant les éléments du tableau ainsi que plusieurs labels pour structurer la logique du programme (*i.e.* `MINI`, `LOOP`, `INCR`).

```
1  int tableau[] = {46, 42, 18, 3, 50, 16, 48, 6};
2  const int N = sizeof(tableau) / sizeof(int);
3
4  int minimum() {
5    int min = tableau[0];
6    for(int index = 1; index < N; index++)
7      if (tableau[index] < min)
8          min = tableau[index];
9    return min;
10 }
```

```
1       .data
2  TABL: .word 46, 42, 18, 3, 50, 16, 48, 6
3       .text
4  MINI: la $t0, TABL            # $t0 = tableau
5       li $t1, 7                # $t1 = iteration count
6       lw $v0, 0($t0)           # $v0 = tableau[0]
7       j INCR
8  LOOP: lw $t2, 0($t0)          # $t2 = tableau[i]
9       slt $a1, $t2, $v0        # test whether $t2 < $v0
10      beq $a1, $zero, INCR     # if not -> goto INCR
11      move $v0, $t2            # else $v0 = $t2
12 INCR: addiu $t0, $t0, 4       # $t0 = &(tableau[i+1])
13      addiu $t1, $t1, -1       # decrease $t1
14      bne $t1, $zero, LOOP     # if ($t1 != 0) -> LOOP
15      jr $ra                   # else end.
```

FIGURE 2.1 – Exemple de recherche du minimum d'un tableau d'entiers en C et en MIPS.

Le registre `$t0` est utilisé pour stocker l'adresse de l'élément courant du tableau, le registre `$t1` stocke le nombre d'itérations de la boucle restantes et le registre `$v0` est utilisé pour garder le minimum trouvé jusqu'au point où le programme est arrivé. Le tableau est inscrit directement dans ce code (*cfr.* label `TABL`). Le programme commence au label `MINI` en initialisant les registres `$t0, $t1 et $v0` à leurs valeurs de départ avant de passer à la portion finale de la boucle (*cfr.* label `INCR`). Dans cette portion, le programme avance l'adresse de l'élément courant (*cfr.* ligne 12), décrémente le nombre d'itérations (*cfr.* ligne 13). Sur base du nombre d'itérations restantes, le programme décide soit de retourner au début de corps de la boucle ou bien de terminer (*cfr.* lignes 14-15). Le corps de la boucle à proprement parler (*i.e.* `LOOP`) consiste à charger la valeur de l'élément courant (*cfr.* ligne 8), le comparer au minimum connu à ce stade (*cfr.* ligne 9) et selon le résultat du test, soit passer à la fin de la boucle (*cfr.* ligne 10) ou bien à mettre à jour le minimum connu (*cfr.* ligne 11).

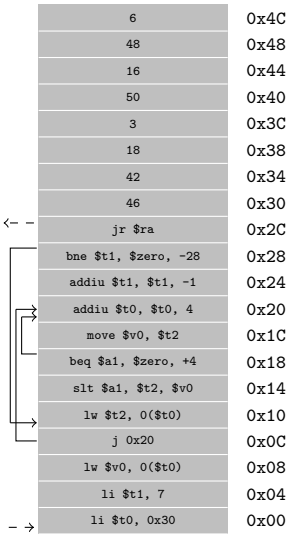


FIGURE 2.2 – Numérotation d'un programme. L'adresse `0x00` est utilisée pour la première instruction et les instructions successives reçoivent les adresses successives. Le tableau correspond aux adresses consécutives à partir de l'adresse `0x30`, juste après la dernière instruction.

Pour pouvoir être exécuté par un processeur, ce programme doit être transformé en un ensemble de mots binaires pour représenter les instructions assembleur qui le constitue (*i.e.* code machine) ainsi que les éléments de son tableau (*i.e.* espace de données). La Figure 2.2 présente une manière de numéroter les pièces de ce programme. Chaque case grise correspond à une pièce qui est représentée par un mot de 32 bits. Dans ce contexte, les noms de fonctions, de variables et labels utilisés n'ont aucun sens et doivent être remplacés par des adresses qui seront inscrites dans le code machine. Pour numéroter toutes les pièces du programme (*i.e.* instructions et données), une adresse de départ est choisie, ici `0x00`, et les pièces consécutives se voient attribuées les adresses consécutives au départ de cette valeur. Par souci de place, nous nous limitons à un adressage en 8 bits représentable par deux digits hexadécimaux. En pratique, cet adressage signifie que nous pourrions numéroter un maximum de 256 octets individuels dans notre programme. Par conséquent, le label `TABL` utilisé dans la première instruction (*cfr.* ligne `4`, adresse `0x00`) doit être remplacé par l'adresse absolue à laquelle commence le tableau (*i.e.* adresse `0x30`). D'autre part, les labels `INCR` et `LOOP` dans les instructions de branchement conditionnelles (*cfr.* lignes `10` et `14`) sont remplacés par des adresses relatives (*i.e.* `+4`, `-28`). Enfin, le label `INCR` du branchement inconditionnel (*cfr.* ligne `7`) est remplacé par l'adresse absolue à laquelle se trouve l'instruction en question (*i.e.* adresse `0x20`).

La première problématique qui va nous intéresser est que lorsqu'un programme est exécuté, il doit résider dans la mémoire principale pour pouvoir faire l'objet d'accès mémoires (*i.e. fetch*, `lw`, `sw`) par le processeur. Cependant, il est possible que les adresses qui ont été choisies lors de la numérotation pour produire l'objet de la Figure 2 soient occupées par d'autres programmes ou par le système d'exploitation. Pour pouvoir exécuter notre programme, celui-ci va devoir être placé à un endroit de la mémoire qui ne correspond pas à la numérotation qui en a été faite. Afin de garantir une exécution correcte, un système de projection des adresses du programme vers les adresses physiques devra être mis en place. D'autre part, lorsqu'un programme est exécuté par le processeur, il est important du surveiller les accès mémoire qu'il réalise pour éviter qu'un programme n'accède à de la mémoire qui ne lui appartient pas.

La deuxième problématique qui nous intéressera porte sur la gestion de la mémoire disponible. La mémoire physique étant une ressources en quantité limitée, le système d'exploitation doit savoir à tout moment quelles parties sont libres et quelles parties sont allouées à un programme particulier. Cette représentation est utilisée pour trouver rapidement un espace de mémoire de taille suffisante lorsque les besoins d'un programme le requiert.

Ces deux problématiques sont intimement liées et le mécanisme de projection et de protection choisi à des conséquences sur la façon dont la gestion de la mémoire disponible sera faite. Dans les deux cas, les solutions apportées doivent présenter un niveau d'efficacité et de performance acceptable pour ne pas introduire une trop grande perte de performance dans le déroulement de l'exécution des différents programmes qui tournent sur le système.

2.1 Projection et protection

Nous avons vu comment un programme peut être perçu comme étant un ensemble de pièces (*i.e.* instructions, données) qui sont liées entre elles par le biais d'adresses. Il est intéressant de dire quelques mots à propos de deux types d'adresses ; **absolues** et **relatives**.

Les adresses **absolues** représentent un endroit bien précis dans la mémoire principale ; elles identifient une instruction (*e.g.* début d'une fonction) ou une donnée spécifique (*e.g.* variable, case d'un tableau, sommet de *stack*). Un programme qui s'exécutent va utiliser la mémoire principale pour stocker toutes les données qu'il n'a pas la possibilité de garder dans un des registres du processeur. Ces adresses désignent un endroit précis dans l'espace d'adressage du programme. Sur un système utilisant un adressage 32 bits linéaire, une adresse absolue est comprise entre `0x00000000` et `0xffffffff` pour couvrir un total de 4 GiB de mémoire.

Les adresses **relatives** sont utilisées pour représenter un décalage par rapport à un endroit précis dans la mémoire principale. Celles-ci sont utilisées par certaines instructions (*e.g.* `beq`, `bne`) pour représenter le nombre d'instructions qui doivent être passées pour avancer ou reculer par rapport à l'indicateur d'instruction courante. De ce fait, une adresse relative spécifique peut produire une adresse absolue différente selon la valeur de l'adresse de base qui lui est ajoutée (*e.g.* `$pc`, `$sp`).

Le choix des adresses qui sont utilisées pour numéroter un programme est fait par le compilateur du langage dans lequel ce programme est écrit. Cependant, ce choix *à priori* peut entrer en conflit avec la réalité de l'occupation de la mémoire physique au moment où le programme est exécuté. Le fait de choisir une adresse *à priori* pour stocker un tableau de données pour un programme n'offre aucune garantie que cette sera libre au moment d'exécuter le programme. En effet, sur un système moderne, il existe à tout moment de nombreux programmes qui tournent sur un système (*cfr.* Chapitre 3) qui peuvent occuper différents endroits de la mémoire physique. Il est donc possible que les adresses qui ont été supposées lors de la construction de l'objet présenté à la Figure 2.2 soient occupées par un autre programme. Nous allons voir comment il est possible de transformer toutes les adresses en terme desquelles un programme est numéroté pour garantir une exécution correcte de celui-ci, quelque soit la manière dont il réside en mémoire physique.

2.1.1 Adressage absolu

Nous allons commencer notre étude de la problématique de projection et de protection en considérant le cas de figure le plus direct ; l'adressage absolu. Dans un tel système, les adresses qui ont été choisies au moment de la compilation seront supposées être celles auxquelles les pièces du programme résident dans la mémoire physique.

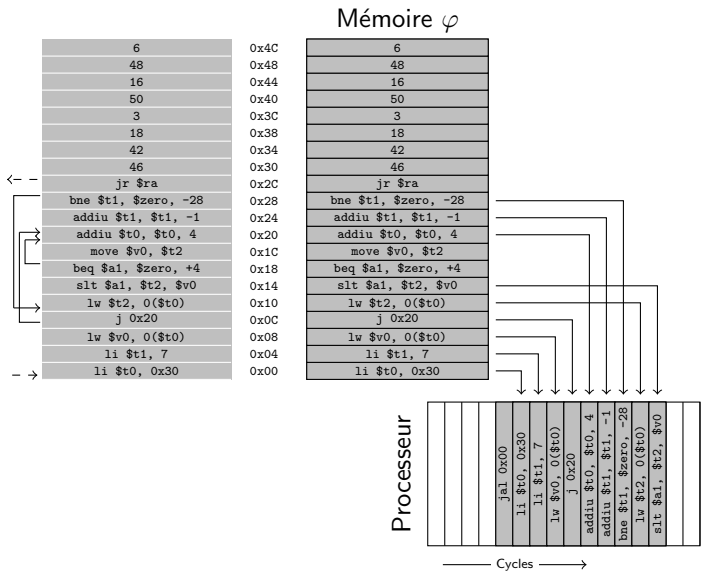


FIGURE 2.3 – Exécution avec adressage absolu.

Dans l'adressage absolu, les pièces constitutives du programme sont copiées précisément aux adresses qui ont été choisies au moment de la compilation. Au moment de démarrer ce programme, le système d'exploitation va utiliser les informations du fichier qui le contient pour savoir où copier chaque instruction et chaque morceau de données dans la mémoire physique. Une fois cette procédure accomplie, le système d'exploitation pourra démarrer l'exécution du programme en branchant vers sa première instruction (*i.e.* adresses `0x00`). À partir de là, le processeur va adopter son comportement normal de progression dans le code ; *fetch-decode-execute* de l'instruction courante et mise à jour de l'indicateur d'instruction courante. À chaque accès mémoire (*e.g. fetch*, `lw`), le processeur utilise l'adresse que l'instruction exige ; soit encodée dans l'instruction, soit présente dans un registre. Le programme fonctionne correctement étant donné que l'hypothèse qui a été faite (*i.e.* la résidence en mémoire physique coïncide parfaitement avec la numérotation *à priori*.

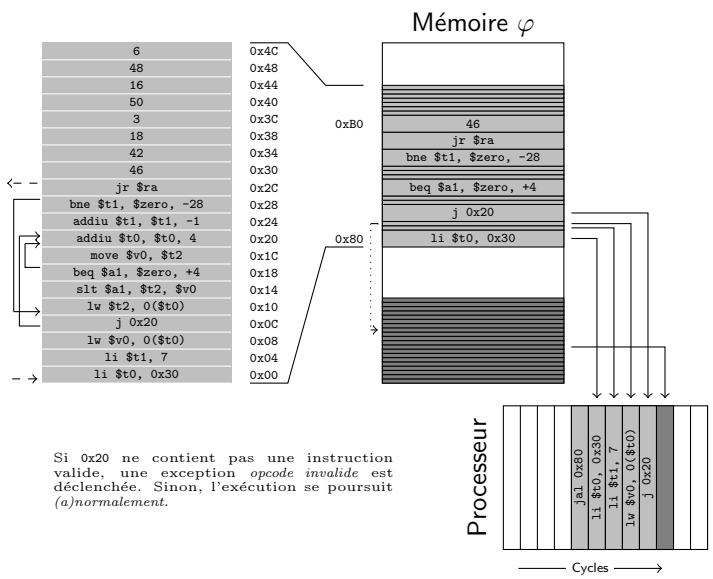


FIGURE 2.4 – Exécution *anormale* avec adressage absolu.

Supposons que les adresses physiques au départ de `0x00` ne soient pas libres ; un autre programme ou le système d'exploitation dispose de ses propres pièces à cette région en gris foncé. Il serait tentant de charger le programme à un autre endroit de la mémoire physique qui est disponible pour pouvoir l'exécuter. Imaginons que le programme soit chargé au départ de l'adresse `0x80` dans la mémoire physique. Au moment de démarrer ce programme, le système d'exploitation va désormais copier chaque instruction et chaque morceau de donnée au départ de cette adresse, violant ainsi l'hypothèse de la numérotation *à priori* qui a été faite. Notons que le tableau qui est supposé résider au départ de l'adresse `0x30`, commence à l'adresse `0xB0` et que la première instruction de la portion d'incrémentation de la boucle qui est supposée résider à l'adresse `0x20` se trouve réellement à l'adresse `0xA0`. Cependant, cette réalité n'est prise en compte nulle part, que ce soit au niveau du code chargé ou du processeur qui va l'exécuter tel quel.

Lorsque le processeur va tenter d'exécuter le premier `lw` pour assigner le premier élément du tableau au minimum courant, l'adresse `0x30` va être utilisée, ce qui correspond à *quelque chose* qui se trouve dans la région en gris foncé dans la mémoire physique. Plus intéressant encore, lorsque le processeur est amené à exécuter le branchement inconditionnel vers `0x20`, celui-ci va se faire un plaisir de le réaliser. À partir de ce point, si à l'adresse `0x20` en mémoire se trouve un mot binaire ne représentant pas une instruction valide, une exception de type *opcode* invalide sera levée par le processeur, donnant lieu à un déroutement vers le système d'exploitation (*cfr.* Chapitre 1). Si par contre, une instruction valide se trouve à l'adresse `0x20`, le processeur va continuer l'exécution de façon *anormale*. Les deux adresses absolues `0x20` et `0x30` sont valides dans la logique du programme mais invalides au moment de l'éxécution, de par le fait que le programme réside à l'adresse `0x80`. Par contraste, les adresses relatives ne sont pas affectées par ce décalage dans la résidence du programme.

Dans un système avec adressage absolu, le programme doit absolument résider en mémoire physique à l'endroit où il a été supposé résider au moment de la compilation. Si certaines de ces adresses ne sont pas libres au moment de l'exécution, celle-ci ne peut avoir lieu au risque de compromettre l'intégrité de l'éxécution du programme. D'autre part, même si le programme réside dans la mémoire physique là où sa numérotation le suppose, rien ne l'empêche d'accéder à n'importe quel endroit de la mémoire physique, y compris dans des régions qui appartiennent à un autre programme ou au système d'exploitation. Si le programme contient une erreur de programmation qui l'amène à tenter un accès au-delà des limites de son tableau, le processeur effectuera cet accès.

2.1.2 Adressage relatif

Une première solution pour pouvoir restaurer l'intégrité de l'exécution du programme serait d'appliquer une correction aux adresses qu'il utilise. Dans notre programme illustratif, la seule différence qui existe entre les adresses choisies à priori et les adresses réellement occupées se situe dans le décalage qui existe entre l'adresse de base supposée (*i.e.* `0x00`) et l'adresse de base réelle (*i.e.* `0x80`). Si ce décalage pouvait être utilisé pour corriger toutes les adresses du programme, l'exécution pourrait se faire de la même manière quelque soit l'endroit où commence le programme en mémoire physique. Nous allons couvrir deux approches possibles pour réaliser cette correction.

Adressage relatif statique

La première approche consiste à modifier le code machine du programme à exécuter. Au moment de la compilation, une numérotation peut être choisie comme précédemment. Au moment de charger ce programme en mémoire physique, le système d'exploitation peut choisir un endroit où le placer (*e.g.* `0x80`). Cette adresse de base qui a été choisie peut être utilisée pour appliquer une *translation* sur toutes les adresses qui figurent dans le programme afin de les transformer en l'adresse réelle qui leur correspond. La Figure 2.5 présente le programme d'origine à gauche. Les deux adresses absolues qui posent problème ; `0x20` et `0x30`, peuvent être corrigées en ajoutant le décalage qui existe entre l'adresse de base supposée et l'adresse de résidence physique pour produire

les adresses corrigées ; `0xA0` et `0xB0`. Lorsque le système d'exploitation réalise la copie du code de ce programme vers la mémoire physique, il peut modifier à la volée les instructions qui contiennent les adresses absolues en les remplaçant par les adresses corrigées. Le programme corrigé est celui qui représente la même logique que le programme d'origine et qui pourra être copié dans la mémoire physique à partir de l'adresse `0x80`.

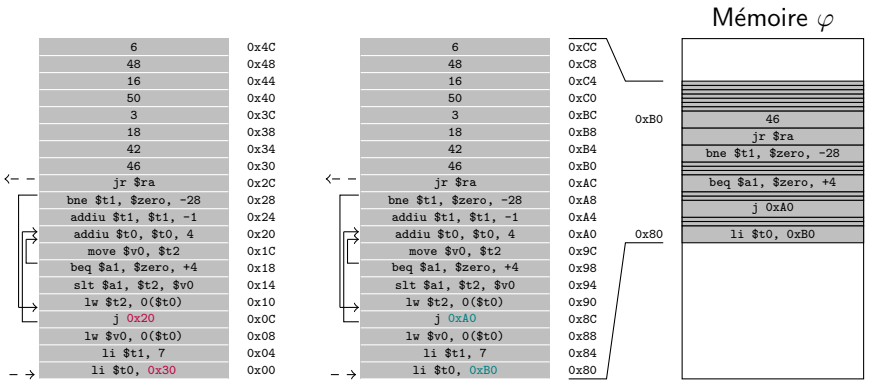


FIGURE 2.5 – Correction des adresses lors du chargement. Les adresses absolues dans le programme d'origine (à gauche) sont modifiées sur base de l'adresse de résidence choisie (au centre) pour produire le programme corrigé qui sera effectivement copié en mémoire physique.

Les corrections apportées dans le code du programme nécessite que toutes les adresses absolues puissent être identifiées sans ambiguité dans son contenu. Pour permettre ceci, un fichier exécutable va pouvoir contenir une table des endroits précis dans le code où apparaissent des adresses absolues. Au moment du chargement, le système d'exploitation va consulter cette table pour savoir quelles instructions doivent être modifiées en conséquence.

L'approche statique de l'adressage relatif permet de corriger toutes les adresses utilisées par le programme pour tenir compte du décalage qui existe entre les adresses *logiques* du programme et les adresses physiques où il réside effectivement. Cependant, cette correction ne peut être effectuée qu'au chargement initial du programme. Une fois que le programme a démarré son exécution et qu'il a progressé dans son code, il n'est plus possible de le déplacer. En effet, bien que ses instructions puissent être remodifié si l'on souhaite déplacer le programme à un autre endroit de la mémoire physique, les adresses calculées qui peuvent être stockées dans la *stack* ne pourront pas être corrigées pour refléter ce nouveau déplacement. D'autre part, cette approche n'empêche pas les accès mémoire hors de la région attribuée au programme. Si le programme contient une erreur de programmation qui l'amène à accéder au-delà des limites de son tableau de données, le processeur effectuera ces accès mémoire. L'adressage relatif statique offre une solution correcte pour la projection des adresses *logiques* du programme vers les adresses physiques. Cependant, il n'offre aucune protection contre les accès vers des adresses qui n'appartient pas au programme.

Adressage relatif dynamique

La seconde approche consiste à adjoindre au processeur une unité en charge de réaliser la translation lors de chaque accès mémoire. Cependant, comme le processeur est sollicité pour réaliser ces translations, il est possible de lui donner une autre responsabilité ; la protection contre les accès mémoire *invalides*. Une distinction apparaît entre les adresses *logiques* qui sont demandées par le programme et les adresses *physiques* vers lesquelles les accès sont réellement effectués.

Pour permettre au processeur de prendre en charge ces responsabilités, deux registres spéciaux doivent lui être ajoutés ; un registre de **base** et un registre de **limite**. Le premier est utilisé pour stocker l'adresse où commence le programme en cours d'exécution en mémoire physique afin de permettre au processeur d'effectuer la translation. Le second registre est utilisé pour stocker la taille en octets qu'occupe ce programme pour permettre au processeur de vérifier que l'accès mémoire reste à l'intérieur de la région de mémoire physique qui lui est allouée.

Lors de chaque accès mémoire, le processeur compare l'adresse *logique* à la **limite**. Si l'adresse *logique* est strictement plus petite, l'accès est considéré comme valide et l'adresse de **base** est ajoutée pour produire l'adresse physique où l'accès doit avoir lieu. Dans le cas contraire, une exception de type *accès mémoire invalide* est générée (*cfr.* Chapitre 1) et le processeur branche vers le système d'exploitation pour sa gestion.

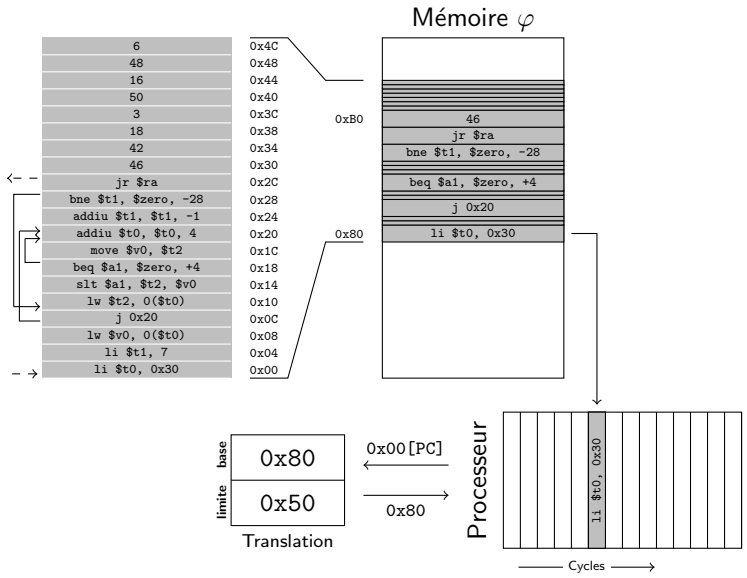


FIGURE 2.6 – Traduction du `$pc` pour réaliser un *fetch*.

Lorsque le système d'exploitation charge le programme, son adresse de résidence en mémoire physique est sauvegardée dans une structure de données associée à ce programme (*cfr.* Chapitre 3) de même que sa limite. Chaque programme a de ce fait une adresse de début en mémoire physique qui lui est propre ainsi qu'une taille en octets qui est déterminée par l'ensemble de son code, son espace de données et son espace de *stack*. Avant que le programme ne commence à s'exécuter, l'adresse où il commence en mémoire physique (*i.e.* `0x80`) est chargée dans le registre de **base** tandis que sa taille en octets (*i.e.* `0x50`) est chargée dans le registre de **limite**. Lorsque le programme commence à s'exécuter à l'adresse `0x00`, celle-ci se trouvant en-dessous de la **limite**, le processeur réalise la translation qui produit l'adresse physique `0x80` où celui-ci réalisera l'accès mémoire.

De manière similaire, lorsque le processeur doit réaliser un accès mémoire pour lire le contenu d'une case du tableau et la charger dans un registre, la même procédure est appliquée. Supposons que le programme arrive à l'instruction `lw` pour laquelle l'adresse logique utilisée sera `0x30` (*i.e.* la première case du tableau). L'adresse en question étant inférieure à la **limite**, l'adresse de **base** lui sera ajoutée pour produire l'adresse physique `0xB0` à laquelle le processeur réalisera l'accès. La progression dans le code va générer les accès vers les adresses logiques consécutives qui correspondent aux cases successives du tableau, le processeur réalisant une opération de translation similaire dans tous les cas.

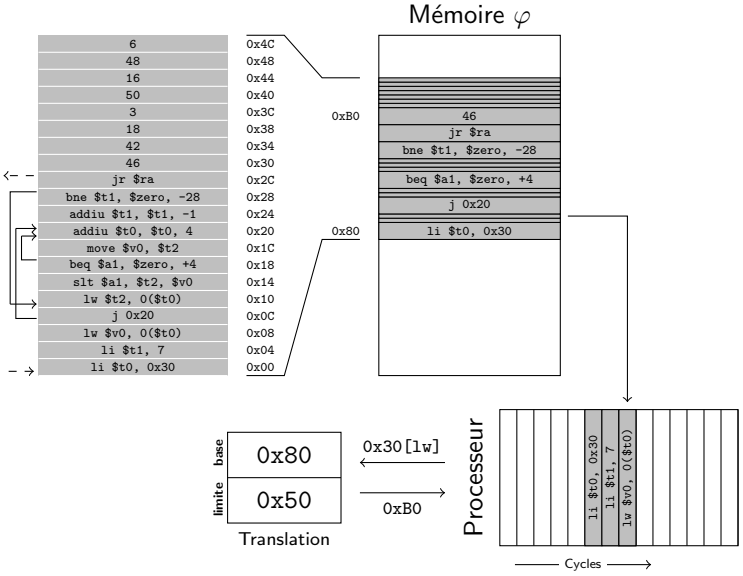
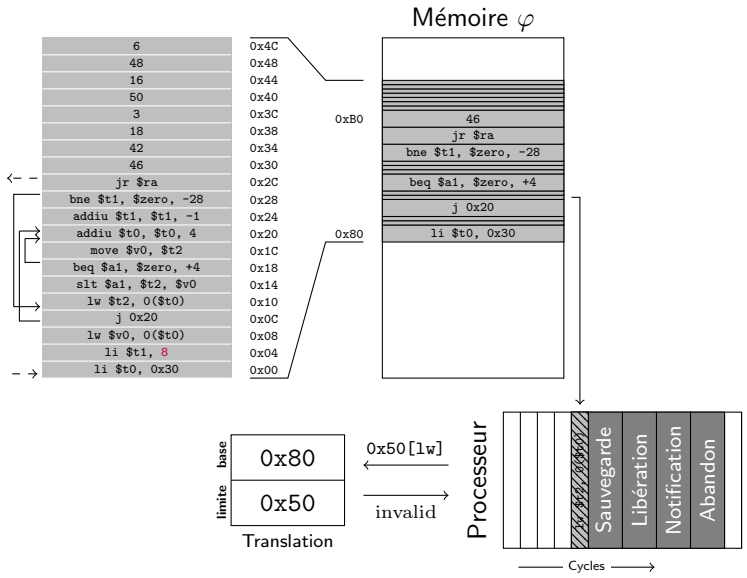


FIGURE 2.7 – Traduction de l'adresse `0x30` lors d'un `lw`.

L'adjonction des deux registres spéciaux et d'une unité de translation permet au processeur de contrôler tous les accès qui sont tentés par le programme dans le cadre de son exécution. Le registre de **limite** permet de vérifier que l'accès se fait à l'intérieur de la région allouée au programme, tandis que le registre de **base** permet de réaliser la projection vers l'endroit où cette région se trouve en mémoire physique.

Une fois que les valeurs associées au programme qui va occuper le processeur sont chargées dans ces registres, tous les accès mémoire qui doivent être réalisés par le processeur (*e.g. fetch*, `lw`) vont nécessiter une opération de test de l'adresse logique (*i.e.* comparaison avec la **limite**) et une opération de translation (*i.e.* addition de la **base**). Bien que ces deux opérations soient parmi les plus rapides qu'un processeur puisse réaliser, celles-ci vont néanmoins rallonger quelque peu la durée de tous les accès mémoire et de ce fait réduire la vitesse d'exécution du programme. Grâce à l'adressage relatif dynamique, un programme peut exister à n'importe quel endroit de la mémoire physique. D'autre part, si le programme est déplacé à un autre endroit de la mémoire physique après qu'il ait commencé son exécution, il suffira de modifier les valeurs de **base** et de **limite** qui lui correspondent. Toutes les adresses *logiques* qu'il produira seront correctement translatées vers les nouvelles adresses physiques, y compris celles qui ont été calculées et sont stockées dans la *stack*.



Afin de mettre en évidence la manière dont la protection va être assurée, introduisons une erreur de programmation dans notre programme. Au lieu de réaliser 7 itérations de la boucle principale, imaginons que notre programme en réalise 8 [1]. En pratique, le programme va être amené à tenter un accès à l'adresse logique `0x50`. Lorsque le processeur arrive à ce point, cette adresse n'étant pas strictement plus petite que la **limite**, celui-ci déclenchera une exception (*i.e. accès mémoire invalide*) qui donnera lieu à un déroutement et traitement par le système d'exploitation. La conséquence pour le programme sera son abandon et donc sa terminaison.

FIGURE 2.8 – Traduction de l'adresse `0x50` lors d'un `lw`.

2.1.3 Segmentation

Dans un système avec segmentation, il existe une multiplicité d'espaces d'adressages disjoints pour numéroter les pièces d'un programme donné. Chacun de ces espaces d'adressage correspondent à un **segment** du programme qui va contenir un sous-ensembles de ses pièces. De la sorte, il existera un ou plusieurs segments de code, de données et de *stack*.

Pour pouvoir déterminer à quel endroit un segment particulier réside, un **descripteur de segment** lui est associé pour encoder son adresse de **base** ainsi que sa **limite**. Il est également possible d'encoder le type du segment correspondant à un descripteur (*i.e.* code ou données) et de lui associer un niveau de privilège requis pour y accéder. Dans le cas d'un segment de données, il est possible de spécifier la direction de croissance de celui-ci (*i.e.* **expansion**) pour couvrir des données classiques (croissance vers le haut) ou d'une *stack* (croissance vers le bas). D'autre part, il est possible également d'attacher une permission d'écriture à un segment de données (*i.e.* **modifiable**). De la sorte, une opération d'écriture dans un segment (*e.g.* `sw`) est susceptible de provoquer une exception (*i.e.* tentative de modifier un segment non-modifiable). Enfin, un champ **modifié** permet de savoir si un segment à fait l'objet d'une modification récemment. Ce champ peut être mis à jour par le processeur lorsque celui-ci réalise une opération d'écriture dans ce segment. Un dernier champ particulièrement intéressant est le bit de **présence** qui permet de contrôler si un segment est actuellement chargé dans la mémoire physique ou non. En pratique, l'ensemble de tous les segments d'un programme ne doivent pas absolument être chargés en mémoire physique pour permettre à ce programme de s'exécuter. Si celui-ci est actuellement dans un segment de code qui contient un algorithme complet de triage qui va l'occuper pendant un temps conséquent, les autres segments de code qui ne sont pas activement utilisés n'ont pas besoin d'être chargé dans la mémoire physique. Nous reviendrons sur une illustration ultérieurement dans cette section.

Dans un système segmenté, les adresses logiques sont formées de deux composantes ; le numéro de segment et le décalage à l'intérieur de ce segment (*i.e. offset*). Pour réaliser la translation d'une telle adresse logique, le processeur doit consulter le descripteur de segment correspondant au numéro de segment, vérifier que le décalage est en dessous de la **limite** de ce segment et lui ajouter l'adresse de **base** de ce segment pour produire l'adresse physique.

Pour décrire complètement la façon dont un programme réside en mémoire physique, une **table des segments** lui est associée qui reprend l'ensemble des descripteurs de ses segments. Cette table, qui existe en mémoire physique, est utilisée lors de chaque accès mémoire pour réaliser la translation de l'adresse logique en une adresse physique. De manière naïve, cette approche nécessiterait de réaliser un accès en mémoire physique (pour consulter le descripteur de segment) lors de chaque accès mémoire. Pour limiter l'impact du coût associé au temps d'accès en mémoire, des registres de **sélecteurs de segment** sont ajoutés au processeur (*e.g.* `cs`, `ds` en x86). En chargeant un numéro de segment dans un de ces registres, le processeur va au passage récupérer le descripteur de segment correspondant et le mettre en cache. De la sorte, toutes les opérations de translations utiliseront le contenu dans la cache au lieu de consulter à chaque fois le descripteur en mémoire physique.

1. Pour produire un tel comportement, il suffit de remplacer `<` par `<=` à la ligne 6 du code C de la Figure 2.1

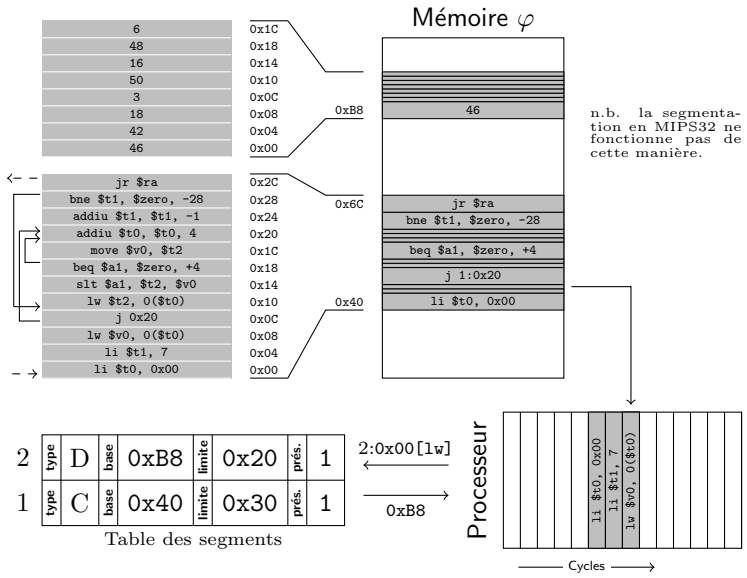


FIGURE 2.9 – Traduction de l'adresse logique `2:0x00` lors d'un `lw`.

Supposons que le programme est constitué de deux segments ; un segment de code (numéro 1) qui contient l'algorithme de recherche du minimum, un segment de données (numéro 2) qui contient le tableau à parcourir. Le choix est fait par le système d'exploitation de charger ces deux segments respectivement aux adresses physiques `0x40` et `0xB8`. Afin de correctement refléter cette réalité, la table des segments va encoder le fait que l'adresse de **base** du segment 1 est `0x40` et sa **limite** est `0x30` tandis que l'adresse de **base** du segment 2 est `0xB8` et sa **limite** est `0x20`. Les bits de **présence** de ces deux segments sont tous les deux à `1`. Dans cette situation, lorsque le processeur doit réaliser un `lw` de la première case du tableau (*i.e.* `0x00` du segment 2), le décalage est inférieur à la limite et la translation produit l'adresse physique `0xB8`.

Présence de segment

La multiplicité des segments dont est composé un programme couplé au bit de présence rend possible l'existence partielle d'un programme en mémoire physique. Avec la technique de l'adressage relatif dynamique, un programme devait exister d'un seul tenant à un endroit arbitraire de la mémoire physique. Avec la segmentation, la flexibilité dans la gestion de la mémoire physique est accrue de par le fait que le système d'exploitation est en mesure de décider de décharger un certain segment pour faire de la place en mémoire physique en offrant la possibilité au processeur de détecter l'absence d'un segment grâce au bit de **présence**.

Lorsque le système d'exploitation prend la décision de retirer un segment de la mémoire physique, il suffit qu'il mette à jour le bit de **présence** du descripteur correspondant à `0`. Si le processeur est amené à tenter un accès dans un segment dont le bit de **présence** est à `0`, une exception est déclenchée (*cfr.* Chapitre 1 ; instruction/donnée non-chargée en mémoire) qui peut être traitée par le système d'exploitation. Le traitement consistera à trouver un endroit dans la mémoire physique pour y charger le segment non-présent dans lequel le processeur a tenté d'accéder. Le segment en question devra être copié depuis la mémoire secondaire où il se trouve vers la mémoire principale. De plus, le système d'exploitation devra mettre à jour l'adresse de **base** pour refléter l'endroit où il réside désormais et positionner le bit de **présence** à `1`. Une fois ces opérations réalisées, une reprise pourra être faite pour que le processeur retente l'opération dans le segment qui est désormais présent en mémoire physique.

Supposons maintenant que le programme est constitué de deux segments de code ; le segment de code 1 contient la fonction principale (*i.e.* `main`) appelant la fonction `minimum` qui se trouve dans le segment de code 2 tandis que le segment de données 3 contient le tableau à parcourir. Supposons également que le segment 2 a résidé à l'adresse physique `0x40` mais qu'il a été déchargé pour faire de la place pour un segment, représenté en gris foncé, qui appartient à un autre programme . La réalite de l'occupation de la mémoire physique est que seuls les segments 1 et 3 sont chargés et pas le segment 2 (notez les valeurs des bits de **présence**). Lorsque le processeur est amené à brancher vers la fonction de `minimum` (*i.e.* `2:0x00`), le bit de **présence** du segment 2 étant à `0`, le processeur déclenchera une exception (segment non-présent) et réalisera un déroutement vers le système d'exploitation.

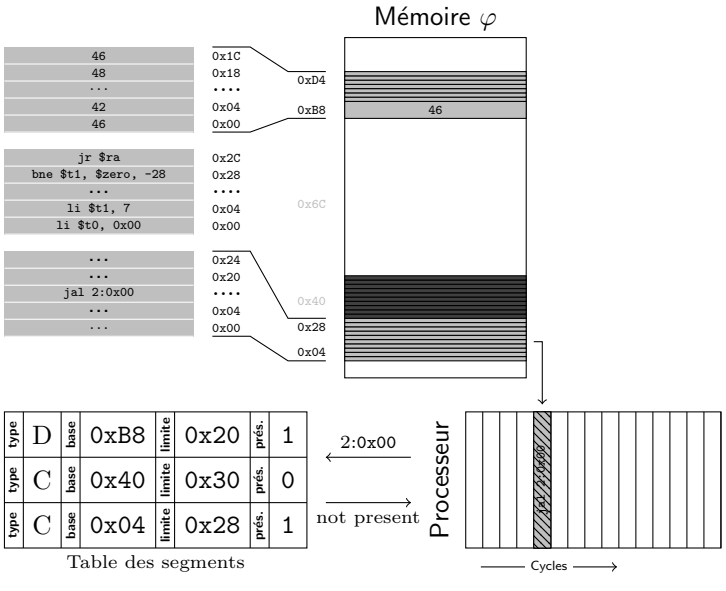


FIGURE 2.10 – Segment de code 2 non-présent.

Après avoir sauvegardé le contexte du programme, le système d'exploitation pourra trouver un emplacement en mémoire physique pour le segment dont le processeur a besoin. Supposons que le système d'exploitation prend la décision de charger le segment 2 à l'adresse physique `0x60`. L'adresse de **base** du descripteur de segment 2 devra être mise à jour pour contenir cette valeur de `0x60` et le bit de **présence** devra être positionné à `1`. À ce stade, le contexte sauvegardé peut être restauré et une reprise peut être faite pour que le processeur retente le branchement vers l'adresse `2:0x00` qui peut désormais se faire correctement. La translation produisant l'adresse physique `0x60` sur base du contenu du descripteur de segment.

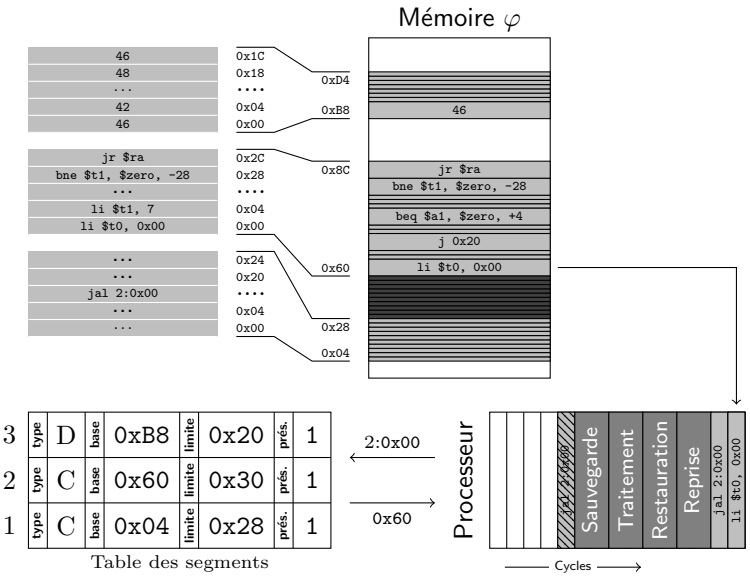


FIGURE 2.11 – Segment de code 2 chargé.

Partage de segment

Dans un système avec segmentation, il est possible de séparer les régions de mémoire physique qui sont accessibles aux différents programmes en contrôlant le contenu de leurs tables des segments. Cependant, il est possible de permettre à différents programmes de se partager une région de mémoire physique pour qu'ils puissent tous y accéder.

Une application pratique de cette technique se situe au niveau des fonctions de librairies partagées. Il est fréquent sur un système que les différents programmes utilisent des fonctions qui sont disponibles pour tous les développeurs (*e.g.* `printf`, `open`, `malloc`). En pratique, le code de ces fonctions est le même pour tous les programmes et si chaque programme devait stocker ce même code dans la mémoire physique qui lui appartient, ceci représenterait un gaspillage de mémoire physique.

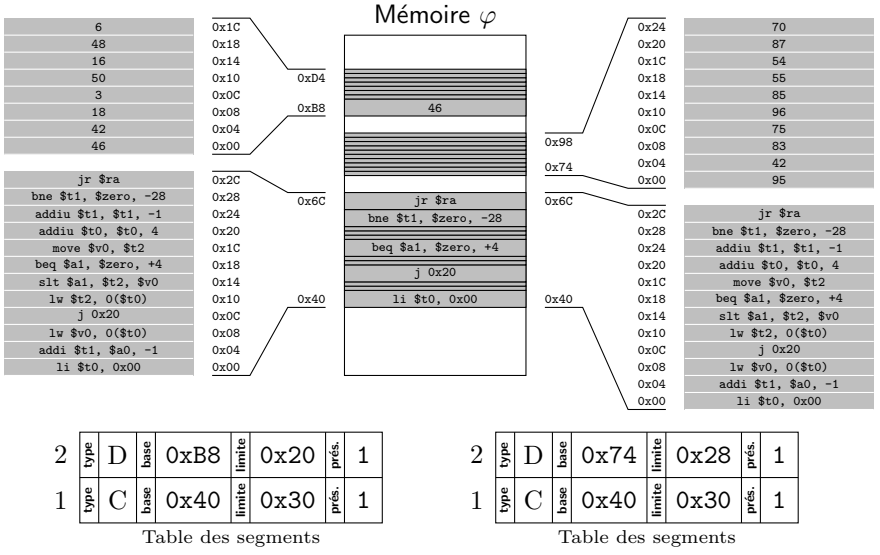


FIGURE 2.12 – Partage de segments

Supposons deux programmes de notre système qui doivent identifier le minimum d'un tableau qui leur est spécifique. Chaque programme va être organisé en deux segments ; un segment de code qui contient la fonction `minimum` et un segment de données qui contient le tableau à parcourir. Cependant, les descripteurs de segment de code dans les deux tables des segments peuvent pointer vers la même adresse de **base**. Lorsque le programme de gauche (resp. droite) est en train de s'exécuter, la table des segments utilisées par le processeur est celle de gauche (resp. droite) qui lui permet uniquement d'accéder au code de la fonction `minimum` et à son tableau.

2.1.4 Pagination

Un système de pagination offre le maximum de souplesse dans la manière dont un programme peut résider en mémoire physique. Deux espaces d'adressage linéaires sont considérés dans le cadre de la pagination ; l'espace *virtuel*, qui est propre à un programme particulier, et l'espace *physique* qui est propre à la machine considérée. La taille du premier dépend de l'architecture processeur (*e.g.* 32 ou 64 bits) et la taille du second dépend du contrôleur mémoire et de la quantité de mémoire vive disponible.

La pagination repose sur un découpage uniforme de ces deux espaces sur base d'une taille fixe (*e.g.* 4 KiB, 4 MiB). L'espace *virtuel* est découpé en unités appelées des **pages** tandis que l'espace *physique* est découpé en unités appelées des **cadres**. La taille fixe a pour conséquence qu'un cadre peut contenir exactement une page. Le choix de la taille introduit une séparation dans chaque adresse virtuelle qui peut être accédée entre la portion qui encode le numéro de la page et la portion qui encode le décalage à l'intérieur de cette page.

Pour une taille de découpage de 4 KiB, étant donné une adresse de 32 bits, les 20 bits de poids le plus fort représentent le numéro d'une page (soit 1.048.576 pages possibles) et les 12 bits de poids faibles encodent un décalage l'intérieur de cette page. Pour une taille de découpage de 4 MiB, étant donné une adresse de 32 bits, les 10 bits de poids le plus fort représentent le numéro d'une page (soit 1024 pages possibles) et les 22 bits de poids faibles encodent un décalage l'intérieur de cette page.

Afin de pouvoir localiser l'endroit où une page réside en mémoire physique, une **table des pages** est utilisée pour encoder dans quel cadre se trouve une certaine page. Cette table des pages, qui est **propre à chaque programme**, contient autant d'entrées qu'il y'a de pages dans l'espace virtuel (*i.e.* **exhaustive**) et réside elle-même en mémoire physique. Avant de donner le processeur à un programme particulier, le système d'exploitation doit charger la table des pages à utiliser dans la *Memory Management Unit* qui est l'unité en charge de l'opération de traduction qui permet de produire l'adresse physique à laquelle l'accès doit réellement être effectué. La taille en octets qu'occupe une table des pages dépend directement du nombre de pages et du nombre d'octets qu'une entrée de la table occupe.

$$\text{Taille} = \frac{\text{Espace virtuel (octets)}}{\text{Taille de page (octets)}} \times \text{Taille d'une entrée (octets)} \tag{2.1}$$

Chaque entrée de la table des pages contient un **numéro de cadre** qui identifie le cadre occupé par la page en question. La validité de ce champ dépend de la valeur d'un bit de **présence**. Un bit de **présence** positionné à `0` signifie que la page en question n'est pas chargée dans un cadre, tandis qu'un `1` dénote le fait que la page occupe bien le cadre identifié. Outre ces deux informations, une entrée encode également les permissions qui décrivent les usages autorisés pour cette page (*i.e.* `read`, `write`, `execute`). Enfin, deux champs permettent d'encoder le fait que la page a fait l'objet d'un accès récemment (*i.e.* **référencée**) et si cet accès était une écriture (*i.e.* **modifiée**).

Supposons un espace d'adressage virtuel de 8 bits avec des tailles de pages de 16 octets. Dans ce système d'adressage, les 4 bits de poids fort encodent un numéro de page (pour un maximum de 16 pages) tandis que les 4 bits de poids faible dénote le décalage à l'intérieur de la page. Le code et les données du programme de recherche du minimum sont organisées dans cet espace virtuel, représenté en gris clair du côté gauche. Chaque groupe de 4 mots binaires consécutifs représente une page complète. Par souci de concision, nous nous limitons à représenter les cinqs premières pages de l'espace virtuel étant donné que les onzes pages restantes ne sont pas utilisées. La table des pages encode la correspondance qui est présentée entre l'espace virtuel et l'espace physique. Les cinqs premières pages occupent cinqs cadres de l'espace physique. Certains cadres sont occupés par des pages d'autres programmes ou du système d'exploitation.

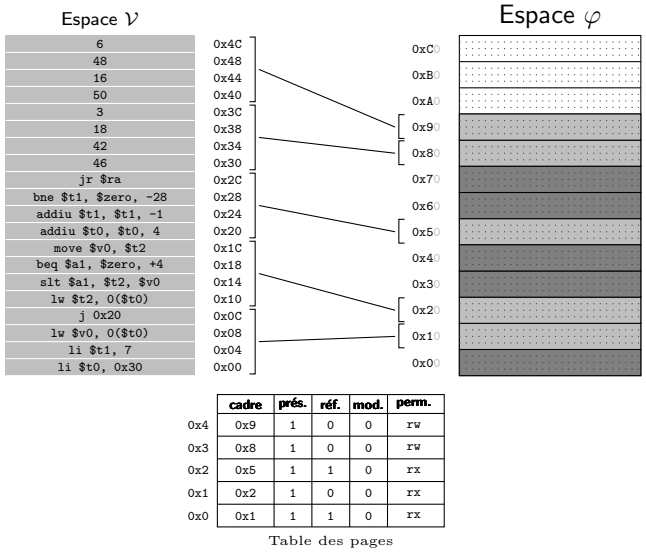


FIGURE 2.13 – Correspondance entre espaces virtuel et physique.

À chaque accès mémoire, le processeur sépare l'adresse virtuelle en deux parties, suivant la taille de découpage qui a été choisie. La portion qui encode le numéro de page est utilisée comme indice dans la table des pages pour en extraire l'entrée correspondante. Si le bit de **présence** est à `1`, le numéro de cadre encodé dans l'entrée est concaténé avec la portion de l'adresse virtuelle qui encode le décalage dans la page pour produire l'adresse physique.

Si le processeur doit réaliser un accès mémoire vers l'adresse virtuelle `0x34`, le numéro de page (*i.e.* `0x3`) est utilisé pour accéder à la quatrième entrée de la table des pages. Le bit de **présence** étant positionné à `1`, le processeur peut utiliser le numéro de cadre de cette entrée pour produire l'adresse physique `0x84`.

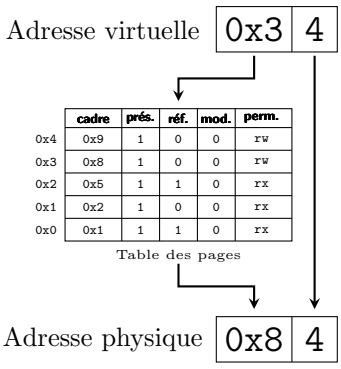


FIGURE 2.14 – Traduction de l'adresse virtuelle `0x34`.

Mémoire virtuelle

Le bit de **présence** rend possible l'exécution d'un programme quelque soit la quantité de mémoire physique disponible pour le stocker. De ce fait, un programme peut exister partiellement en mémoire physique et le système d'exploitation peut se contenter de garder uniquement les pages qui sont requises pour son exécution (*e.g.* la page de code courante, la page de données courante). Le fait qu'une page occupe ou non un cadre est encodé dans la table des pages via le bit de **présence**. Un système d'exploitation dispose d'un espace dédié dans la mémoire secondaire, par exemple sous la forme d'une partition de *swap*, pour héberger sur disque les pages qui ne peuvent pas résider en mémoire physique.

Lorsque le processeur doit accéder à une adresse virtuelle dans une page dont le bit de **présence** est positionné à `0`, cet accès ne peut avoir lieu car la page en question ne se trouve pas dans l'espace physique. Une exception de type **défaut de page** se produit et le processeur réalise un deroutement vers le système d'exploitation qui va devoir gérer cette situation. L'objectif premier va consister à trouver un cadre de l'espace physique pour y charger la page manquante. S'il existe au moins un cadre libre, le système d'exploitation peut en choisir un au hasard pour recevoir la page. Dans le cas où tous les cadres sont occupés, il est nécessaire de réaliser un remplacement de page ; évincer une page d'un des cadres occupés pour la remplacer par la page manquante. Comme le processeur détecte le défaut de page sur base des informations contenues dans la table des pages, il est également nécessaire de mettre à jour celles-ci. En effet, lors d'une éviction, deux entrées doivent être mises à jour ; celle de la page qui se fait évincer du cadre choisi (dont le bit de **présence** doit basculer à `0`) et celle de la page qui se trouve chargée dans le cadre choisi (dont le bit de **présence** doit basculer à `1`). D'autre part, le numéro de cadre de la page qui est chargée doit être mis à jour pour refléter son emplacement dans l'espace physique. À ce stade, le système d'exploitation peut réaliser une reprise du programme pour que le processeur retente d'exécuter l'instruction qui avait généré le défaut de page précédemment.

Supposons que notre programme de recherche du minimum ait déjà parcouru les 4 premières cases de son tableau et qu'il a de ce fait couvert la première page qui le contient (*i.e.* numéro `0x3`). Supposons également que la seconde page (*i.e.* numéro `0x4`) occupée par le reste du tableau n'est pas présente dans l'espace physique. Lorsque le processeur doit accéder à l'adresse virtuelle `0x40`, le bit de **présence** étant positionné à `0`, une exception de type défaut de page est déclenchée. Par conséquent, le processeur va réaliser un déroutement vers le système d'exploitation.

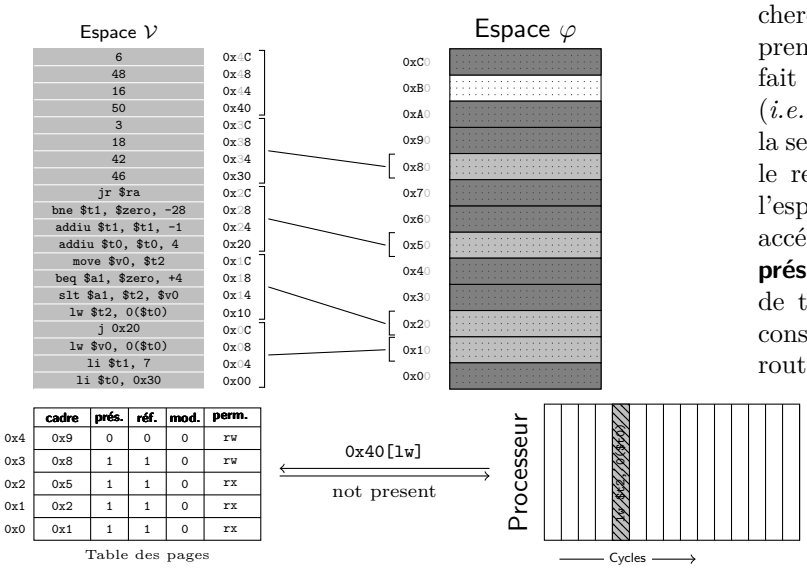


FIGURE 2.15 – Défaut de page (numéro `0x4`).

Dans le cadre de la gestion de ce défaut de page, le système d'exploitation va devoir trouver un cadre libre. Supposons que le cadre `0xB` soit actuellement libre. Le système d'exploitation peut dès lors copier la page requise depuis la mémoire secondaire vers le cadre dans l'espace physique. Pour que le processeur puisse réaliser la traduction, l'entrée correspondante (indice `0x4`) dans la table des pages devra être mise à jour. Le bit de **présence** sera positionné à `1` et le numéro de **cadre** sera positionné à `0xB` pour refléter la réalité de l'espace physique. Une fois la reprise accomplie, le processeur retentera l'accès mémoire vers l'adresse virtuelle `0x40` qui pourra être correctement traduite en l'adresse physique `0xB0` sur base de l'information de la table des pages.

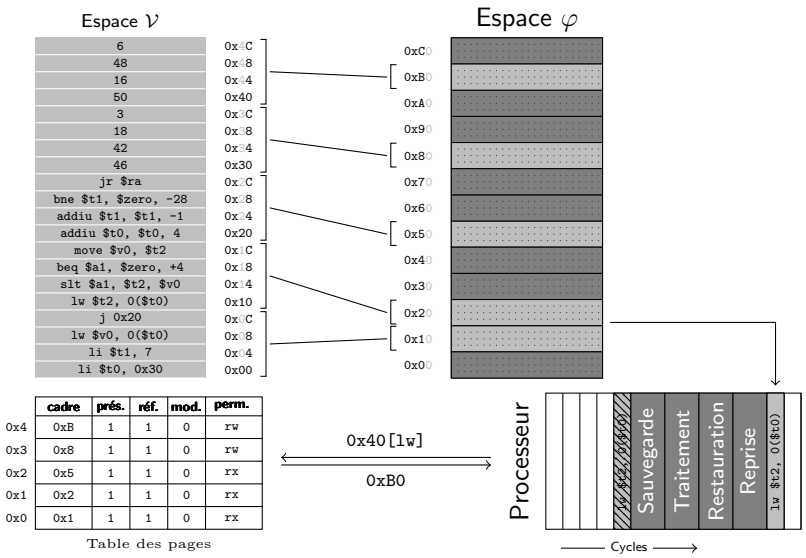


FIGURE 2.16 – Traduction de l'adresse virtuelle `0x50`.

Éviction de page

L'éviction de page s'avère nécessaire lorsque deux conditions sont réunies. Premièrement, le programme en cours d'exécution tente de réaliser un accès dans une page qui n'est pas chargée (*i.e.* bit de **présence** à `0`). Cette situation cause une exception de type *défaut de page*. Deuxièmement, l'ensemble de tous les cadres de l'espace physique sont actuellement occupés par une page.

La procédure de remplacement de page va nécessiter de sélectionner une page à un évincer de façon à libérer le cadre qu'elle occupe. Ensuite, il va être nécessaire de copier la page manquante depuis la mémoire secondaire vers la mémoire physique. D'autre part, la page qui est évincée va potentiellement devoir être sauvegardée si elle a fait l'objet de modification pendant le temps qu'elle a passée dans son cadre. Le bit **modifiée** de l'entrée permet au processeur de marquer les pages lorsqu'il réalise des opérations d'écriture dedans (*e.g.* `sw`). Afin d'éviter de prendre du temps pour réaliser cette écriture, il est préférable d'évincer une page qui n'a pas été modifiée car la copie d'origine dans la mémoire secondaire est identique à celle en mémoire physique. Enifn, les entrées deux pages qui sont affectées par l'éviction ; celle qui perd un cadre et celle qui gagne un cadre, doivent être mises à jour en conséquence pour refléter la nouvelle réalité de l'occupation de la mémoire physique. Notons que l'éviction d'une page de l'espace physique introduit potentiellement un défaut de page futur, si la page qui a été évincée fait l'objet d'une tentative d'accès ultérieurement.

Au vu du coût en terme de temps pour gérer un défaut de page (*i.e.* une copie du disque vers la mémoire physique, une copie potentielle de la mémoire physique vers le disque, deux accès mémoire pour mettre à jour les deux entrées), il est désirable de poser un choix dans la sélection de la page à évincer de manière à minimiser le nombre de défauts de page qui sont produits. Cette ligne directrice nous permet d'imaginer un algorithme qui serait optimal dans le sens où il réduire au strict minimum le nombre de défauts de page qui serait requis pour conclure un ensemble de processus donné.

En considérant l'ensemble de toutes les pages actuellement chargées dans des cadres, cet algorithme optimal procéderait en annotant chacune de ces pages du nombre d'instructions qui devront s'exécuter avant que le programme à qui elle appartient n'y fasse un accès. En choisissant d'évincer la page qui est annoté du nombre le plus grand, nous retarderions le plus loin possible dans le futur le prochain défaut de page. Malheureusement, cet algorithme optimal nécessiterait de pouvoir établir l'ordre des accès dans les différentes pages des différents processus, ce qui nécessiterait de faire tourner ces processus. De nombreuses stratégies pratiques existent pour ne pas devoir prédire le futur mais plutôt d'essayer d'obtenir une décision relativement bonne sur base du passé récent.

La première stratégie consiste à disposer d'une structure de données qui permet de savoir quel temps s'est écoulé depuis le dernier accès dans chacune des pages chargées en mémoire physique (*i.e. Least Recently Used*). Cette stratégie représente une bonne approximation de l'algorithme optimal si les processus impliqués respectent un principe de localité. Pour autant que les programmes sous-jacents réalisent des accès mémoire à proximité de leurs accès mémoire récents, le *LRU* produira de bons résultats en terme de minimisation du nombre de défauts de page. La difficulté principale est que son implémentation peut s'avérer difficile. En effet, il est nécessaire de disposer d'une structure de données qui peut être mise à jour par le processeur et qui devra adjoindre à chaque

entrée de la table des pages un nombre suffisant de bits pour encoder le temps depuis le dernier accès.

La seconde stratégie consiste à simplifier le *LRU* pour se contenter de savoir si une page a fait l'objet d'un accès dans un passé récent (*i.e. Not Recently Used*). Dans cette stratégie, au lieu de représenter le délai depuis le dernier accès, un bit unique est utilisé et mis à jour par le processeur pour marquer le fait qu'un accès (lecture ou écriture) a été fait dans une page. Dans la description de la table des pages que nous avons fait, ceci correspond aux bits **référencée** et **modifiée**. Lorsqu'une page doit être évincée, le système d'exploitation va choisir en priorité parmi les pages qui n'ont pas été référencées. Si toutes les pages ont été référencées dans un passé récent, on choisira en priorité parmi les pages qui n'ont pas été modifiées récemment. Dans cette stratégie, le système d'exploitation doit passer régulièrement sur toutes les pages qui sont chargées pour remettre à `0` leurs bits **référencée** et **modifiée**. La difficulté d'implémentation est grandement réduite et cette approche offre une bonne approximation du *LRU*.

La troisième stratégie consiste à choisir systématiquement la page la plus anciennement chargée dans l'espace physique (*i.e. First-In-First-Out*). Une structure de type *FIFO* peut être utilisée pour suivre à la trace l'ordre dans lequel les différentes pages du système ont été chargées dans l'espace physique. Lorsque la décision doit être prise d'évincer une page, c'est celle qui se trouve à la tête de cette file qui est retenue. Bien que son implémentation soit simple, le risque principale est que cette stratégie peut évincer la page la plus ancienne qui se trouve être la plus utilisée de façon régulière.

Pour contrecarrer le risque de la stratégie du *FIFO* d'évincer la page la plus utilisée, la stratégie de la *seconde chance* peut être utilisée. Cette stratégie incorpore le bit de **référencée** lors de la sélection de la page à évincer. De la sorte, le choix se reportera sur la page la plus anciennement chargée qui n'a pas fait l'objet d'un accès dans le passé récent. Dans cette approche, le système d'exploitation doit régulièrement passer sur l'ensemble des pages chargées en mémoire physique pour remettre à `0` le bit **référencée**. Du point de vue pratique, cette stratégie est implémentée à l'aide d'une simple liste chaînée des pages ; la tête correspondant à la page la plus anciennement chargée tandis que la fin de la liste correspond à la page la plus récemment chargée. Le parcours de la liste se fait jusqu'à rencontrer une page pour laquelle le bit **référencée** est à `0`. Tant qu'une page référencée est rencontrée, ce bit est remis à `0`. Au terme du premier parcours, si aucune page non-référencée n'a été rencontrée, la tête de la liste est choisie pour éviction.

Une variante possible est la stratégie de l'*horloge* qui offre un niveau similaire de performance en se reposant sur une liste circulaire au lieu d'une liste simplement chaînée.

Considérations avancées

2.2 Gestion de la mémoire physique

Le système d'exploitation doit gérer l'espace de mémoire physique disponible sur une machine et rendre disponibles les régions allouées aux différents programmes à travers un mécanisme de projection/protection tels ceux étudiés au début de ce chapitre. Nous allons considérer la mémoire physique de manière simplifiée comme étant un ensemble d'octets individuellement adressables.

Un aspect important consiste à savoir, à tout moment, quelles portions de la mémoire physique sont allouées à quels programmes du système. Sur base d'une représentation de l'occupation de la mémoire physique, le système d'exploitation peut parcourir cette représentation pour permettre d'allouer de la mémoire de façon dynamique, en réponse à une demande provenant d'un programme particulier. Un paramètre particulier de l'allocation dynamique se situe au niveau du mode d'allocation considéré. Il est possible de réaliser l'allocation par blocs qui existent d'un seul tenant dans la mémoire physique (*cfr.* adressages relatifs, segmentation) qui peuvent avoir une taille fixe ou flexible. Une autre possibilité est de réaliser une allocation avec un découpage de l'espace occupé (*cfr.* pagination). Le reste de cette section va couvrir deux structures de données qui peuvent être utilisées pour représenter l'état d'occupation de la mémoire physique qui sont adaptées aux différents système de projection/projection que nous avons étudiés.

D'autre part, lorsque le préemption entre programmes est autorisée, si un programme a besoin de mémoire physique (*e.g.* présence de segment, mémoire virtuelle), il est possible de prendre de la mémoire physique à un programme pour la donner à autre. Pour permettre ce genre de comportement, il est nécessaire de disposer d'une structure de données qui décrit les régions de mémoire physique (plages ou cadres) qui sont allouées à un programme particulier. Nous reviendrons sur ces considérations dans le chapitre suivant.

2.2.1 Bitmaps

Une structure de type *bitmap* est un tableau de *bits*. Dans un tel système, la mémoire physique est découpée en unités d'allocation de taille constante (*cfr.* cadres dans la pagination), chaque *bit* représentant l'état d'une unité d'allocation de la mémoire physique. Lorsqu'un *bit* est positionné à la valeur `0` (resp. libre), l'unité correspondante est libre (resp. allouée). Pour une quantité de mémoire physique et une taille d'unité d'allocation particulière, le nombre d'unités disponibles ainsi que la taille du *bitmap* sont données par

$$\text{Nombre d'unités} = \frac{\text{Mémoire physique (octets)}}{\text{Taille d'une unité (octets)}}$$

$$\text{Taille du } \textit{bitmap} = \frac{\text{Nombre d'unités}}{8}$$

La figure 2.17 présente quelques chiffres correspondant à différentes tailles d'unité d'allocation pour une mémoire physique donnée. Ces valeurs permettent de mettre en évidence que la taille du *bitmap* augmente au plus la taille d'unité diminue. Pour que le système d'exploitation puisse offrir la possibilité d'allouer de petites quantités de mémoire, il est nécessaire de disposer d'un *bitmap* plus grand pour représenter ce nombre plus important d'unités.

Taille d'unité	Nombre d'unités	Taille bitmap
1 B	1.073.741.824	131.217.728 B = 128 MiB
1 KiB = 1024 B	1.048.576	131.072 B = 128 KiB
1 MiB = 1.048.576 B	1024	128 B

FIGURE 2.17 – Tailles de *bitmaps* pour une mémoire physique de 1 GiB (1.073.741.824 octets).

D'autre part, si une taille d'unité plus grande est choisie (*e.g.* 1 MiB) mais que les programmes du système ont tendance à avoir besoin de moins dans le cadre de leurs allocations (*e.g.* 512 KiB), cela aura pour conséquence d'augmenter la fragmentation interne (ici 50%). Dans ce cas particulier, les programmes recevront plus de mémoire qu'ils n'ont réellement besoin étant donné que les unités d'allocation ne peuvent pas être découpées dans la représentation choisie.

Considérons une mémoire physique de 512 octets et choisissons une taille d'unité d'allocation de 4 octets. Il en découle que 128 unités d'allocation sont disponibles dans cette mémoire, ce qui signifie un *bitmap* encodé sur 16 octets. Imaginons que l'état d'occupation de la mémoire physique est celle donnée à la Figure 2.23. Dans cette figure, les adresses présentées à droite sont regroupées en blocs de quatres unités d'allocation, ce qui est représenté par le digit hexadécimal de poids le plus faible coloré en gris. Le contenu de la mémoire physique consiste en une portion occupée par le système d'exploitation (y compris le *bitmap*) ainsi qu'un programme unique. Le programme occupe les unités d'allocation numérotées `0x0C0` jusque `0x10C` tandis que le système d'exploitation réside dans les unités de `0x000` jusque `0x07C`. Le *bitmap* ci-dessous encode l'occupation actuelle de la mémoire physique.

00000000 00000000 00000000 00000000	0x0C
00000000 00000000 00000000 00001111	0x08
11111111 11111111 00000000 00000000	0x04
11111111 11111111 11111111 11111111	0x00

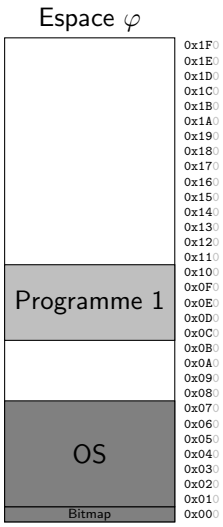


FIGURE 2.18 – Occupation de la mémoire physique

Pour une unité d'allocation précise, l'indice de son *bit* peut être obtenu en divisant son adresse physique par la taille d'unité d'allocation (*i.e.* 4). Pour le système d'exploitation, l'unité d'allocation `0x000` correspond au *bit* 0, tandis que l'unité d'allocation `0x07C` correspond au *bit* 31 ; les *bits* 0 à 3 correspondant aux unités occupées par le *bitmap*. Pour le programme, l'unité d'allocation `0x0C0` correspond au *bit* 47 tandis que l'unité `0x10C` correspond au *bit* 67. De ce fait, les *bits* 0 à 31 (resp. 47 à 67) sont positionnées à `1` pour représenter la région de mémoire physique occupée par le système d'exploitation (resp. le programme).

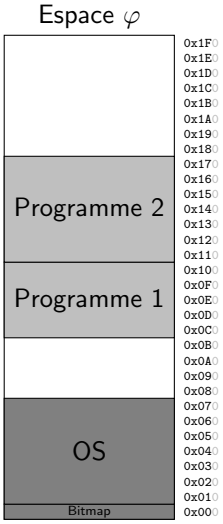


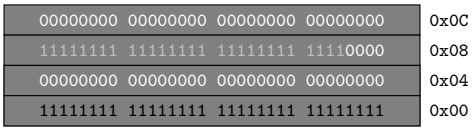
FIGURE 2.19 – Occupation de la mémoire physique après allocation

Supposons qu'un nouveau programme doit être démarré et que celui-ci nécessite de disposer de 112 octets consécutifs de mémoire physique. Ce besoin en mémoire peut être couvert en trouvant 28 unités d'allocation consécutives qui sont disponibles, c'est-à-dire 28 *bits* positionnés à `0` dans le *bitmap*. La région `0x080` à `0x0B0` ne comporte que 16 unités ce qui est insuffisant. En revanche, à partir de `0x110`, il existe une région de mémoire libre et le système d'exploitation peut choisir les unités d'allocation de `0x110` à `0x17C`, ce qui correspond aux *bits* 68 à 95. Pour refléter l'allocation, ces *bits* doivent être positionnés à la valeur `1` pour que ces unités ne puissent pas être allouées pour une autre demande de mémoire. Le nouvel état du *bitmap* sera celui ci-dessous

00000000 00000000 00000000 00000000	0x0C
11111111 11111111 11111111 11111111	0x08
11111111 11111111 00000000 00000000	0x04
11111111 11111111 11111111 11111111	0x00

La région de mémoire physique peut être rendue disponible au programme à travers le mécanisme de projection/protection utilisé. Par exemple, pour de l'adressage relatif dynamique, la valeur de **`base`** est `0x110` et la valeur **`limite`** est de `0x70`. Ces deux valeurs devront être utilisées pour les registres correspondant lorsque le nouveau programme s'exécutera sur le processeur.

Supposons que le programme 1 arrive au terme de son fonctionnement et fait l'objet d'une terminaison normale. En terme de mémoire physique, il est important que le système d'exploitation marque celle detenue par ce programme comme étant désormais libre. En pratique, le système d'exploitation dispose d'une structure de données distincte (*i.e.* descripteur de mémoire) qui décrit quelles unités de mémoire physique sont allouées à un certain programme. En consultant cette structure, le système d'exploitation trouvera que la région de `0x0C0` à `0x10C` est occupée par le programme qui se termine. Les *bits* correspondant dans le *bitmap* sont aux indices 48 et 67 respectivement. Les valeurs de tous les *bits* allant de l'indice 48 à 67 doivent être positionnés à `0` pour dénoter le fait que ces unités d'allocation sont désormais disponibles et peuvent être allouées aux demandes de mémoire à venir ou aux demandes de mémoire en attente.



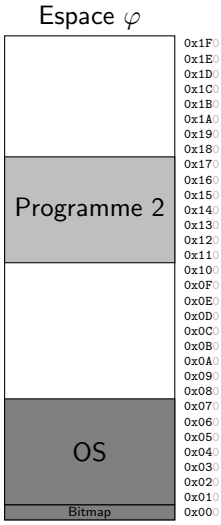


FIGURE 2.20 – Occupation de la mémoire physique après libération

2.2.2 Listes chaînées

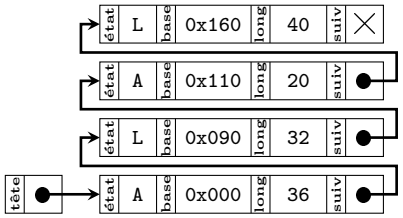
Il est possible de représenter l'occupation de l'espace de mémoire physique à l'aide d'une liste chaînée dont les éléments capture l'information essentielle. Dans un tel cas, chaque nœud de la liste encode une *plage* d'adresse particulière. Un élément contient quatres champs ; l'**état** indiquant si la plage en question est libre ou allouée, la **base** qui represente la première adresse, la **longueur** qui dénote le nombre d'unités couvertes et enfin un pointeur du **suivant** dans la liste.

Afin de déterminer la taille requise pour encoder la liste chaînée complète, il est nécessaire de déterminer combien de *bits* sont requis pour représenter un élément et d'identifier la longueur maximale que la liste peut prendre. La quantité de mémoire physique détermine le nombre d'adresses physiques possibles et le logarithme en base 2 de ce nombre donne la taille d'une adresse physique. La taille des unités d'allocation considérée va déterminer la valeur maximale que le champ **longueur** peut prendre et de ce fait, le nombre de *bits* requis pour encoder cette valeur est le suivant

$$1 + 2 \times \lceil \log_2 (\text{Mémoire physique}) \rceil + \left\lceil \log_2 \left(\frac{\text{Mémoire physique}}{\text{Taille d'allocation}} \right) \right\rceil$$

Pour identifier la taille maximale que la liste peut prendre, il suffit de remarquer qu'elle contiendra exclusivement des plages ayant une **longueur** de 1 avec une alternance de l'**état** entre libre et allouée. Ces deux données prises ensemble permettent de dédier un espace de taille suffisante pour stocker la plus longue liste possible et garantir que le système d'exploitation pourra, en toutes circonstances, disposer d'une représentation complète de l'état d'occupation de la mémoire physique.

Considérons une mémoire physique de 512 octets et choisissons une taille d'unité d'allocation de 4 octets. Il en découle que 128 unités d'allocation sont disponibles dans cette mémoire. Imaginons que l'état d'occupation de la mémoire physique est celle donnée à la Figure 2.21. Dans cette figure, les adresses à droite sont regroupées en blocs de quatres unités d'allocation, ce qui est représenté par le digit hexadécimal de poids le plus faible coloré en gris. Le contenu de la mémoire physique consiste en une portion occupée par le système d'exploitation (y compris l'espace alloué à la liste chaînéé) ainsi qu'un programme. Le programme occupe les unités d'allocation numérotées `0x110` à raison de 20 unités d'allocation tandis que le système d'exploitation réside dans les unités au départ de `0x000` et dispose de 36 unités d'allocations. La liste chaînée ci-dessous encode l'occupation actuelle de la mémoire physique.



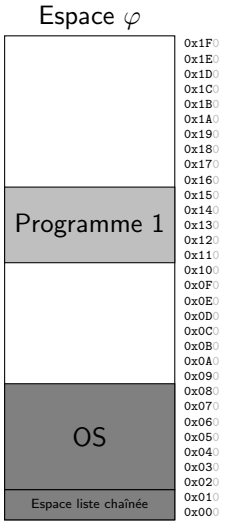


FIGURE 2.21 – Occupation de la mémoire physique

Supposons qu'un nouveau programme doit être démarré et que celui-ci nécessite de disposer de 112 octets consécutifs de mémoire physique. Ce besoin en mémoire peut être couvert en trouvant 28 unités d'allocation consécutives qui sont disponibles. Cet objectif se traduit par la recherche d'un élément de la liste chaînée ayant pour **état** libre et une valeur de **longueur** supérieure ou égale à 28 unités d'allocation. La région `0x090` à `0x100` comportant 32 unités, celle-ci constitue un candidate adapté. Pour refléter l'allocation, la plage est séparée en deux nouvelles plages ; une plage allouée ayant `0x090` pour **base** avec une **longueur** de 28 ainsi qu'une plage libre ayant pour base **0x100** avec une **longueur** de 4 pour représenter les unités résiduelles de la plage d'origine après l'allocation. Le nouvel état de la liste chaîne sera celui ci-dessous

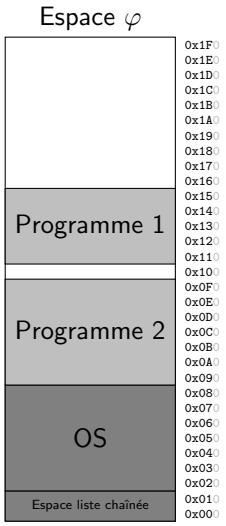
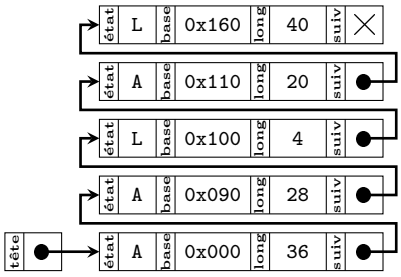
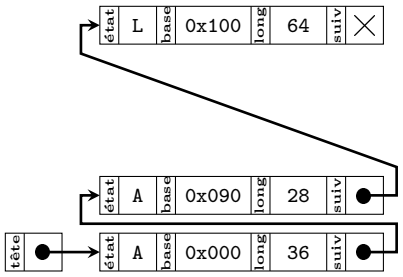


FIGURE 2.22 – Occupation de la mémoire physique après allocation



La région de mémoire physique peut être rendue disponible au programme à travers le mécanisme de projection/protection utilisé. Par exemple, pour de l'adressage relatif dynamique, la valeur de **`base`** est `0x090` et une valeur **`limite`** de `0x70`. Ces deux valeurs devront être utilisées pour les registres correspondant lorsque le nouveau programme s'exécutera sur le processeur.

Supposons que le programme 1 arrive au terme de son fonctionnement et fait l'objet d'une terminaison normale. En terme de mémoire physique, il est important que le système d'exploitation marque celle detenue par ce programme comme étant désormais libre. En pratique, le système d'exploitation dispose d'une structure de données distincte (*i.e.* descripteur de mémoire) qui décrit quelles régions de mémoire physique sont allouées à un certain programme. En consultant cette structure, le système d'exploitation trouvera que la région de `0x110` à `0x15C` est occupée par le programme qui se termine. L'**état** de l'élément correspondant à cette plage dans la liste peut être marqué comme étant libre. Afin de limiter la taille de la liste chaînée, les trois plages libres consécutives à la fin de la liste peuvent être fusionnées en une seule plage libre ayant pour **base** celle de la première (*i.e.* `0x100`) et pour **longueur** la somme de leurs nombres d'unités couvertes (*i.e.* 64 unités au total).



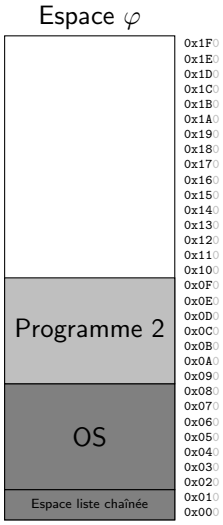


FIGURE 2.23 – Occupation de la mémoire physique après libération

Chapitre 3 Processus

Nous avons considéré jusqu'à présent les problématiques de gestion du processeur et de la mémoire vis-à-vis d'un programme. En pratique, un système d'exploitation moderne doit gérer un ensemble de programmes qui existent à un moment donné, se partageant ces deux ressources. Dans ce chapitre, nous allons étudier les questions liées à la gestion des processus qui se repose sur les concepts vus dans les deux chapitres précédents.

3.1 Constitution

Un **programme** peut être perçu comme étant un objet inerte. Celui-ci est typiquement stocké dans un fichier résidant en mémoire secondaire et est constitué de code machine (incluant les appels systèmes) qui décrit la logique des traitements, un espace de données qui contient toutes les variables globales ainsi qu'une portion de mémoire dynamique et enfin un espace utilisé pour contenir la *stack* qui héberge toutes les variables locales des fonctions et les valeurs qui ne peuvent pas être toutes contenues dans des registres.

Par contraste, un **processus** représente la mise en activité d'un programme. Celui-ci va résider en mémoire, complètement ou partiellement selon les capabilités du mécanisme de projection/protection utilisé. Il va également occuper le processeur lorsque le code de son programme sera effectivement en cours d'exécution. Pendent ces périodes d'exécution, le contenu des registres ainsi que de la *stack* déterminera l'état du programme auquel le processus est arrivé. Outre ces informations techniques pour l'exécution, le système d'exploitation va également devoir réaliser le suivi de l'état d'allocation des ressources à chaque processus (*e.g.* segments et/ou pages utilisées, régions de mémoire physique occupées, fichiers ouverts, *sockets* actives, connections établies).

Dans le cadre de la programmation système, un programme va pouvoir solliciter certains services fournis par le système d'exploitation à travers des **appels systèmes**. Ces fonctions spéciales permettent de directement adresser une demande d'un service particulier ; ouverture ou lecture d'un fichier, allocation de mémoire additionnelle, établissement d'une connection distante. Un appel système prend la forme d'un appel de fonction typique ; un nom de fonction et un ensemble de paramètres qui décrivent la demande. Le *listing* de la figure 3.1 présente un programme qui a pour objectif d'ouvrir et de lire le contenu d'un fichier pour trouver l'entier minimum.

Les lignes $9-10$ prennent en charge l'ouverture du fichier avec l'appel système de la ligne 9 qui stipule le chemin vers le fichier à ouvrir ainsi que les modalités (*i.e.* `O_RDONLY`) d'utilisation de ce fichier. Une demande de service pouvant échouer, la valeur de retour de cette appel peut être testée pour établir si l'ouverture à réussi ou non. Les lignes $12-15$ représente la gestion de l'allocation d'un espace pour acceuillir le contenu du fichier. L'appel système de la ligne 12 indique la quantité d'octets (*i.e.* 8192) qui est demandée au système d'exploitation. La ligne 17 représente la demande de lecture du contenu du fichier vers l'espace de mémoire obtenu avec une taille maximale à récupérer.

Les lignes $19-23$ représente le traitement à proprement parler, à savoir la recherche du minimum dans le tableau en mémoire principale avec l'affichage de la valeur trouvée si le tableau n'est pas vide. Enfin, la libération des ressources se fait à la ligne 26 et la terminaison à la ligne 28.

La différence principale entre un simple appel de fonction et un appel système réside dans l'instruction qui provoque l'appel. Tandis qu'un branchement avec lien (*e.g.* `jal`, `call`) est utilisé pour l'appel de fonction, une instruction d'interruption logicielle (*e.g.* `int`)) est utilisée pour déclencher un évènement dont la fonction attachée est celle qui gère le service demandé.

```c
#include <fcntl.h>   // open()
#include <unistd.h>  // read(), write(), close()
#include <stdlib.h>  // malloc(), free()
#include <minimum.h> // minimum()

#define TAILLE 8192

int main(int argc, char* argv[]) {
  int fichier = open("/home/sdy/infob231/tableau.bin", O_RDONLY);
  if (fichier == -1) { perror("open"); return EXIT_FAILURE; }

  int *tableau = malloc(TAILLE);
  if (tableau == NULL) {
    perror("malloc"); close(fichier); return EXIT_FAILURE;
  }

  int octets_lus = read(fichier, tableau, TAILLE);

  if (octets_lus >= 4) {
    printf("Minimum trouvé : %d\n", minimum(tableau, octets_lus / 4));
  } else {
    printf("Un tableau vide n'admet pas de minimum.\n")
  }

  // Libération des ressources
  free(tableau); close(fichier);

  return EXIT_SUCCESS;
}
```

FIGURE 3.1 – Illustration de programmation de système.

Un processus va devoir alterner entre l'exécution du code de son programme et l'exécution du code du système d'exploitation lorsqu'un service est sollicité. La figure 3.2 illustre la manière dont le processeur est occupé par l'exécution du processus depuis sa création jusqu'à sa terminaison. Les portions en gris clair représentent les tranches de temps CPU utilisé pour exécuter le code du programme tandis que les portions en noir sont les tranches de temps CPU utilisé pour exécuter le code du système d'exploitation en réaction aux appels systèmes déclenchés par le processus (dénotés par un hachurage).

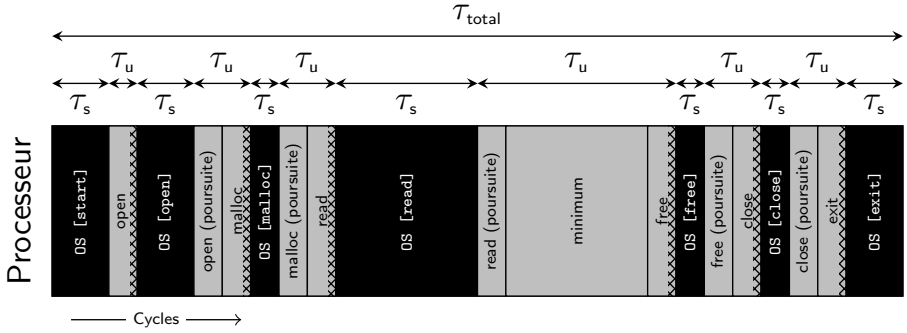


FIGURE 3.2 – Temps d'exécution total d'un processus.

Il en ressort que le temps total d'exécution, compris entre l'instant de création et l'instant de terminaison, est supérieur au temps passé à exécuter le code du programme. De plus, lorsque plusieurs processus existent, ceux-ci peuvent se retrouver à s'exécuter en alternance sur le processeur pour leur permettre de progresser dans leurs traitements respectifs ; on parle dès lors d'un *interleaving* de processus.

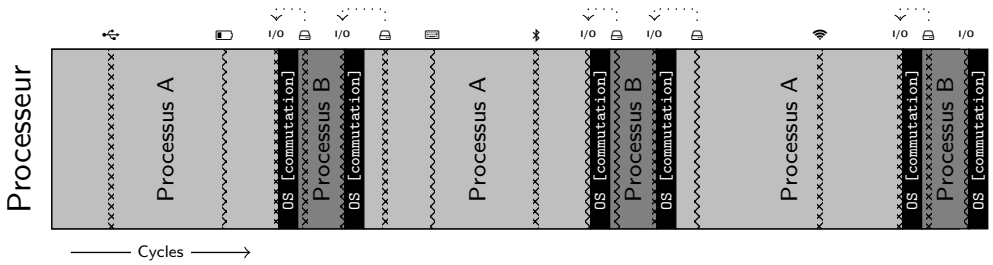


FIGURE 3.3 – *Interleaving* d'un processus *CPU-bound* (A) et *IO-bound* (B).

Il existe une distinction importante entre deux types de processus sur base de leur utilisation des ressources du système ; *CPU-bound* et *IO-bound*. Un processus *CPU-bound* utilise le processeur de manière intensive lorsqu'il le reçoit et ne va que rarement demander une opération nécessitant une entrée-sortie (*e.g.* lecture de fichier, envoi d'un paquet de données). Un processus *IO-bound* utilise peu le processeur une fois qu'il le reçoit et sollicite rapidement un service qui donne lieu à une opération d'entrée-sortie. La figure 3.3 présente un processus de chaque type qui se partage le processeur. Chaque évènement (exception ou interruption) est représenté par un hachurage avec un symbole le décrivant au sommet. Un processus subit une **commutation** lorsqu'il demande une entrée-sortie (I/O) car il en résulte la perte du processeur et l'impossibilité de continuer ses traitements jusqu'à ce que l'entrée-sortie soit conclue. En pratique, l'occupation du processeur est parsemée de supsensions temporaires pour traiter les différents évènements qui peuvent survenir à tout moment.

3.1.1 Table des processus

Parmi les différentes structures de données que le système d'exploitation utilise, la **table des processus** regroupe toutes les informations relatives aux différents processus qui existent à un moment donné. Un identifiant (*a.k.a.* `pid`, *process identifier*) est utilisé pour identifier de manière unique chaque processus. Un **état** permet de représenter dans quelle phase possible de son existence un processus se trouve. Les valeurs possibles d'état sont **exécution**, **prêt**, **bloqué** et **zombie**. Le **contexte** d'exécution, constitué des valeurs des différents registres ainsi que du contenu de la *stack*, doit également être préservé lorsque le programme est **prêt** ou **bloqué**. Ceci inclut également l'indicateur d'instruction courante (*e.g.* `$pc`, `cs:eip`) pour pouvoir continuer l'exécution à l'endroit où elle s'est arrêtée.

Dans le chapitre 2, nous avons étudié deux structures de données liées à la projection de la mémoire ; la **table des segments** et la **table des pages**. Ces deux structures de données se retrouvent dans un **descripteur de mémoire** qui décrit l'état d'utilisation de la mémoire principale ainsi que les régions de mémoire physique qui sont allouées au processus.

Enfin, divers **descripteurs de ressource** sont stockées pour savoir quelles sont les ressources détenues par un processus ainsi que leur état. Par exemple, un processus possède un ensemble de **descripteurs de fichiers ouverts** qui représente les fichiers que ce dernier a ouvert jusqu'à un certain moment (*e.g.* `/home/sdy/data/tableau.bin`) ainsi que le décalage dans ce fichier où la lecture est arrivée.

3.1.2 Cycle de vie

L'**état** joue un rôle important pour capturer l'évolution de la dynamique des processus au cours du temps. Un processus peut se trouver à tout moment dans un seul état possible qui reflète la situation dans laquelle il se trouve. Au cours de son existence, un processus va traverser ces différents états suivants des règles et des restrictions bien précises que nous allons étudier dans cette section.

L'état d'**exécution** représente la situation où le processus est activement exécuté par le processeur (ou par un de ses *cores*). Cet état peut être scindé en deux sous-états, selon que le processeur est occupé à exécuter le code du programme (**mode utilisateur**) ou bien le code du système d'exploitation (**mode système**). Le nombre de processus qui peuvent se trouver dans cet état est limité par le nombre de processeurs/*cores*. Dans le reste de cette section, nous considérons le cas d'un seul processeur *monocore*. L'état **prêt** permet de représenter le cas où un processus peut s'exécuter pour autant qu'un processeur (ou un de ses *cores*) soit libre. En d'autres termes ; rien n'empêche le processus de progresser dans la logique de ses traitements excepté le fait que le processeur est occupé par autre chose. L'état **bloqué** représente la situation où le processus ne peut pas logiquement poursuivre ses traitements tant qu'une certaine condition n'est pas vérifiée (*cfr.* déblocage).

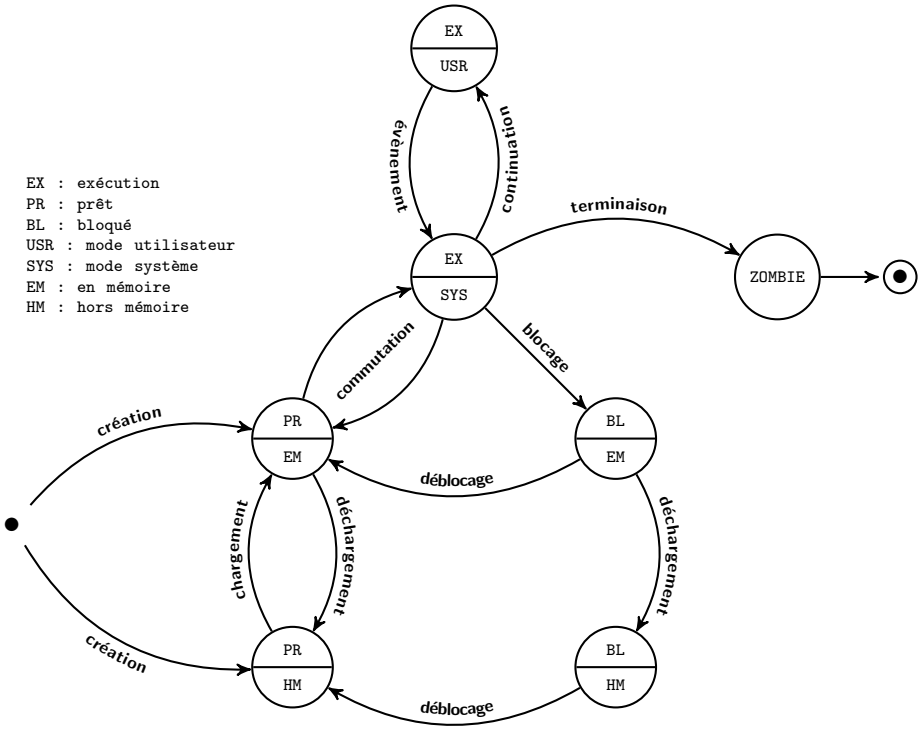
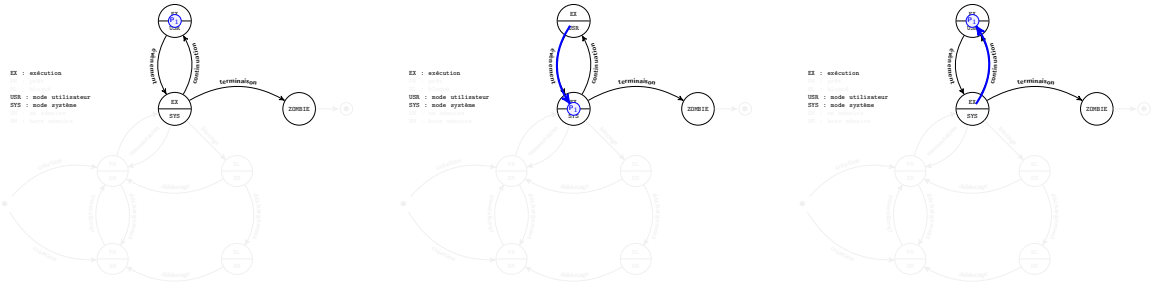


FIGURE 3.4 – Cycle de vie des processus

Gestion d'évènements

Considérons un processus P_1 dont le programme est en train de s'exécuter (*cfr.* Figure 3.5a). Lorsqu'un **évènement** récupérable survient (*e.g.* défaut de page, frappe au clavier), le processeur effectue un déroutement vers le code du système d'exploitation (*cfr.* Figure 3.5b). À ce stade, la sauvegarde du contexte est réalisée suivie du traitement effectif associé à l'évènement (*cfr.* Chapitre 1). Une fois le traitement effectif conclu, le système d'exploitation peut réaliser la restauration du contexte sauvegardé avant la **continuation** (*i.e.* reprise, poursuite). Dans le cas d'un **évènement** irrécupérable, la **continuation** est remplacée par une **terminaison**.



(a) Exécution en cours (b) Occurrence d'un évènement (c) Continuation du programme

FIGURE 3.5 – Transitions d'état dans la gestion d'un évènement

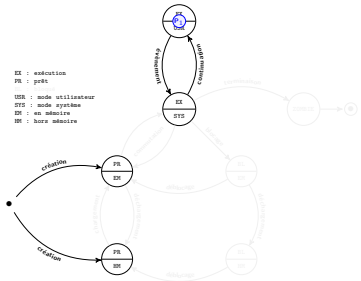
Création de processus

Lors de la création, un processus (appelé *parent*) effectue un appel système pour demander d'engendrer un nouveau processus (appelé *enfant*). Cette relation de filiation introduit une structure d'arborescence des processus. Cette arborescence sous-entend l'existence d'un *premier* processus qui est engendré par le système d'exploitation une fois que l'initialisation de la machine est conclu. Ce processus spécial (`init` ; `pid = 0` ou `1`) a pour responsabilité de créer tous les autres processus de démarrage du système.

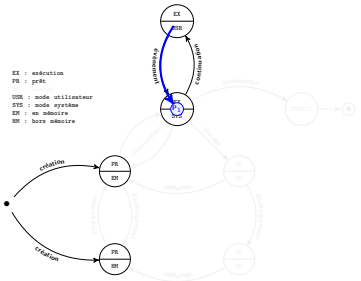
En pratique, la création d'un processus requiert de lui trouver un `pid` parmi ceux disponibles [1] (*i.e.* non-attribués). Le descripteur du processus *enfant* doit être initialisé avec la préparation d'un contexte initial (*i.e.* registres et *stack*). Une allocation de mémoire doit être faite pour recevoir les informations nécessaires de ce

1. Si tous les `pid` sont attribués, la création échoue.

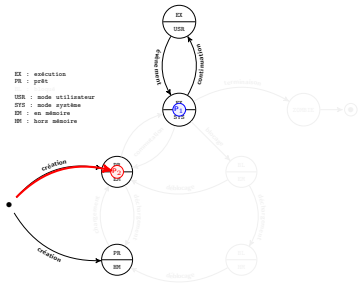
processus (totale ou partielle selon les besoins/disponibilités). Dans le cadre de la création d'un processus, l'espace d'adressage du processus *enfant* est une copie exacte de celui du processus *parent*. De ce fait, la technique du *Copy-on-Write* peut être utilisée afin d'éviter de réaliser une copie complète immédiatement. Les structures de données de projection/protection doivent être initialisées (*e.g.* table des segments, table des pages) pour refléter l'endroit où le programme est chargé en mémoire.



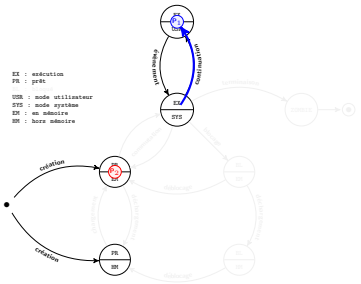
(a) Exécution en cours



(b) Appel système



(c) Initialisation du processus *enfant*



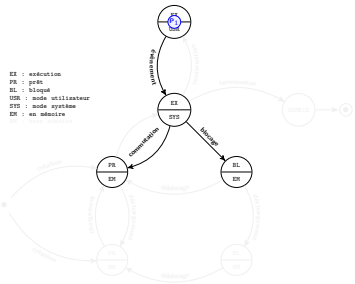
(d) Continuation du processus *parent*

FIGURE 3.6 – Transitions d'état pour la création d'un processus

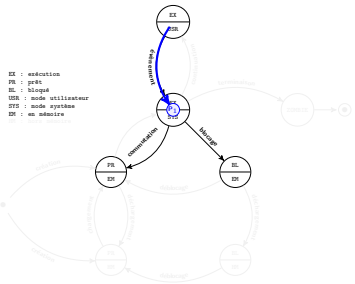
Considérons un processus P_1 dont le programme est en train de s'exécuter (*cfr.* Figure 3.6a). L'appel système de création de processus (*e.g.* `fork()`) prend la forme d'une interruption logicielle, c'est-à-dire un **évènement**, qui provoque un déroutement (*cfr.* Figure 3.6b). Le traitement effectif consiste à réaliser tous les traitements décrits plus haut pour engendrer le processus P_2 qui se retrouve à l'état **prêt**. À ce stade, la seule chose qui empêche P_2 de s'exécuter est le fait que P_1 occupe le processeur. La **continuation** du processus P_1 est effectuée et celui-ci continue son exécution au-delà de l'appel de création (*i.e.* poursuite).

Suspension de processus

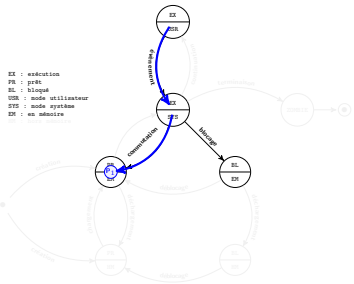
L'occurrence d'un **évènement** est susceptible de provoquer la suspension du processus en cours d'**exécution**. Cette suspension se traduit par la perte du processeur par le processus en cours ce qui permet à un autre processus de le recevoir. Le processus qui perd le processeur est susceptible de le recevoir ultérieurement pour continuer son exécution, ce qui exclut la **terminaison** de celui-ci. Il est possible de distinguer trois cas de suspension selon la transition que subit le processus en cours d'**exécution**.



(a) Exécution en cours



(b) Appel système `sched_yield()`

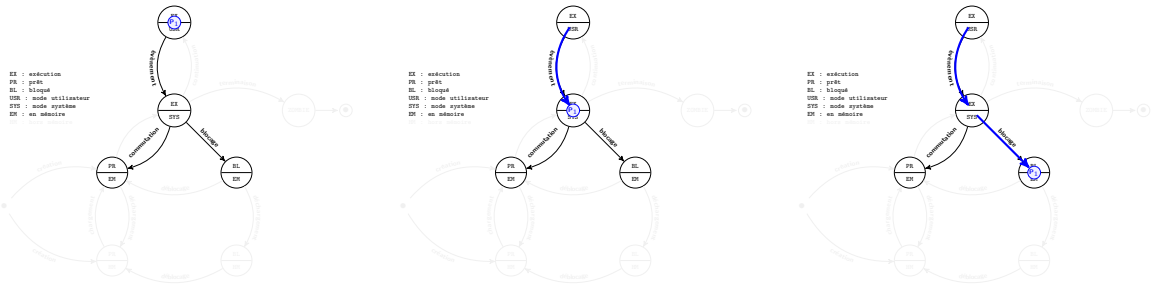


(c) Commutation du processus en cours

FIGURE 3.7 – Transitions d'état pour la suspension simple

La suspension *simple* où le processus perd le processeur et se retrouve à l'état **prêt**. Un processus peut provoquer de lui-même cette suspension en décidant d'abandonner le processeur à l'aide d'un appel système dédié à

cet effet (*e.g.* `sched_yield()`). Une autre possibilité est l'expiration du *quantum* de temps alloué au processus en cours (*cfr. Round-Robin*).



(a) Exécution en cours (b) Appel système `sched_yield()` (c) Blocage du processus en cours

FIGURE 3.8 – Transitions d'état pour la suspension bloquante

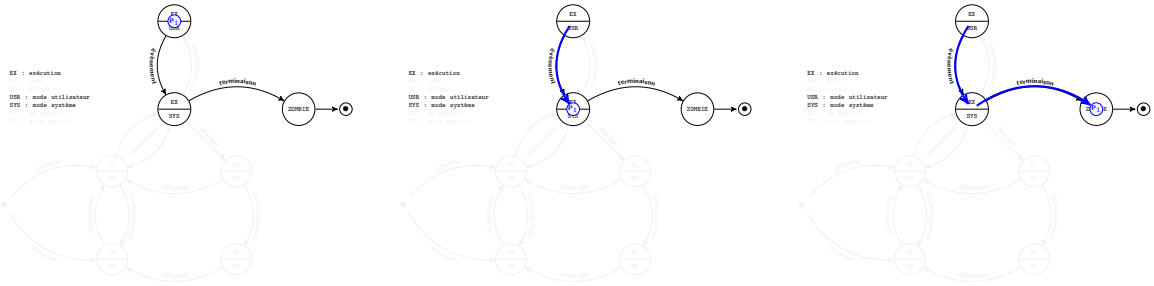
La suspension *bloquante* se produit suite à une demande du processus en cours qui ne peut pas être satisfaite immédiatement, ce qui nécessite de le faire transiter vers l'état **bloqué**. Si le processus en cours sollicite un service qui donne lieu à une entrée-sortie, l'opération en question prendra du temps. L'ouverture ou la lecture (*e.g.* `open`, `read`) depuis un fichier nécessite qu'un périphérique de stockage (*e.g.* HDD, SSD) soit interrogé, ce qui peut prendre un temps plus ou moins conséquent avant que les méta-données du fichier ou son contenu soit effectivement récupéré et présent en mémoire physique. L'établissement d'une connection (*e.g.* `connect`) nécessite l'envoi d'un paquet de données à travers un réseau et la réception de la réponse de la machine vers laquelle la connexion est tentée, une opération qui prendra du temps. Pendant que cette opération progresse, le processus qui est en à l'origine ne peut pas logiquement continuer ses traitements et doit être mis dans un état où il n'est pas candidat pour recevoir le processeur. Dans le cadre de la concurrence (*cfr.* Chapitre 4), une opération d'acquisition d'un *mutex* peut donner lieu à un **blocage** si le *mutex* en question est fermé. Enfin, un processus peut demander à être mis en sommeil pendant une durée précise (*e.g.* `sleep`) ce qui l'empêche de s'exécuter jusqu'à l'expiration de sa période de sommeil.

Enfin, la suspension *non nécessairement bloquante* dans laquelle l'opération demandée peut, selon les circonstances, provoquer une suspension *bloquante* ou une **continuation**. Si le processus en cours réalise une demande d'allocation (*e.g.* `malloc`) et qu'il existe dans la mémoire physique suffisament d'espace libre pour y donner satisfaction, le système d'exploitation peut acter cette allocation (*cfr.* Chapitre 2) avant de continuer le programme du processus en cours. Dans le cas de figure où il n'existe pas d'espace suffisant et que la préemption de mémoire n'est pas possible (*e.g.* remplacement de page interdit), le processus en cours devra être **bloqué** jusqu'à ce que de la mémoire soit libérée par un autre processus.

Terminaison

Le cycle de vie d'un processus est composé de sa création, son existence et sa terminaison. Trois cas possibles peuvent être distingués comme raison d'une terminaison. Un processus peut demander volontairement sa terminaison, par exemple lorsqu'il arrive au terme de ses traitements ou a rencontré une erreur dans le cadre de ceux-ci. À cet effet, un appel système (*e.g.* `exit()`) existe qui permet au processus de fournir une valeur de retour au processus *parent* pour décrire le statut de son exécution (*i.e.* réussi, code d'erreur). Un processus est également susceptible de subir une terminaison en raison d'une exception irrécupérable (*cfr.* Chapitre 1) qui nécessite d'abandonner son exécution. Enfin, un processus peut forcer la terminaison d'un autre processus en lui adressant un signal de terminaison (*e.g.* `kill -SIGKILL`).

Lorsqu'un processus se termine, quelle qu'en soit la raison, l'ensemble de ses ressources allouées doivent être libérées. La mémoire physique qu'il occupait doit être rendue disponible et éventuellement donnée à d'autres processus (*n.b.* **bloqués**) qui en ont besoin. Les ressources ouvertes (*e.g.* fichiers, *sockets*) doivent être proprement clôturés. Pour un fichier, lorsqu'un processus demande une opération d'écriture vers un fichier, le système d'exploitation peut décider de continuer celui-ci directement (pour autant que les données à écrire soient sauvegardées quelque part). Ces opérations d'écritures temporisées doivent être effictivement appliquées sur le support de stockage où le fichier à écrire réside. Pour une connexion réseau (*e.g.* `TCP`), il existe un protocole précis pour gérer la fermeture d'un connexion. Lorsque le processus se termine, il est important de proprement fermer les connexions ouvertes qu'il détenait.



(a) Exécution en cours (b) Appel système `exit()` (c) Libération des ressources

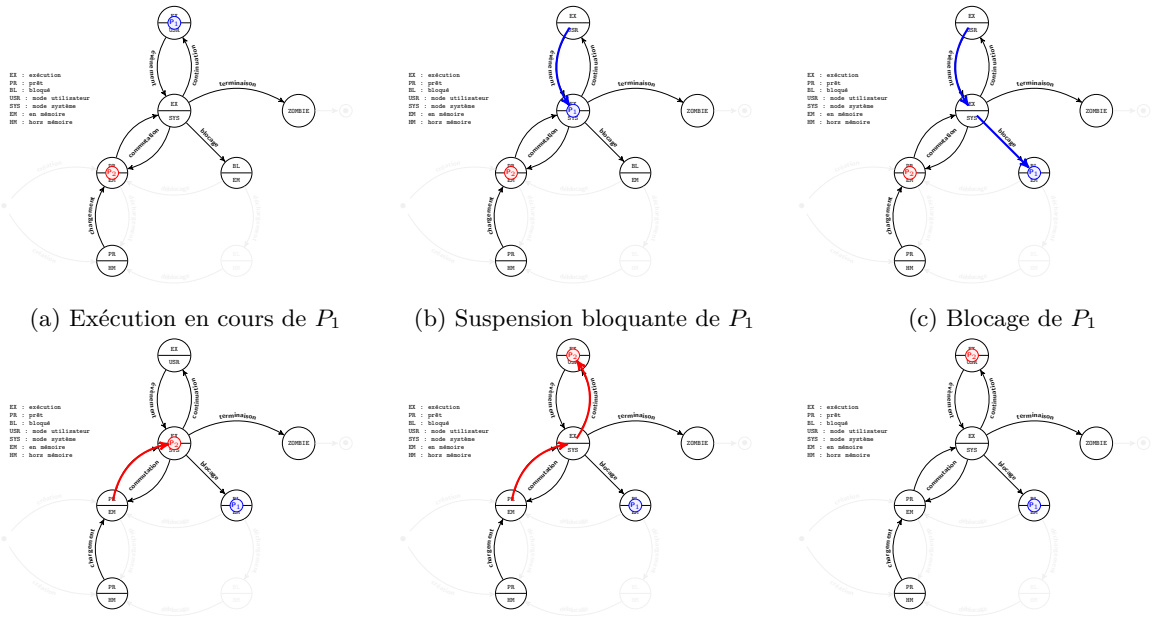
FIGURE 3.9 – Transitions d'état pour la terminaison

Lorsque le nettoyage des ressources détenues par un processus en accompli, celui-ci existe uniquement sous la forme d'un descripteur de processus auquel est associé un `pid` et qui stocke la valeur de retour de ce dernier. Sous cette forme, le processus est passé vers l'état **zombi** est persistera dans cet état jusqu'à ce qu'un processus lise sa valeur de retour. Si le processus *parent* ne se pré-occupe pas de lire cette valeur de retour avant de se terminer lui-même (*n.b.* **zombi** orphelin), le processus racine s'en chargera, ce qui permettre de définitivement libérer le descripteur du processus **zombi** ainsi que son `pid`.

Commutation

Lorsque le processus en cours d'**exécution** perd le processeur, un processus **prêt** peut le recevoir pour progresser dans ses traitements. La **commutation** peut être réalisée en revisitant le canevas de gestion des évènements du Chapitre 1. En autorisant le traitement effectif à *changer* le contexte à restaurer, il est possible de remplacer le processus en cours d'exécution pour réaliser ce que l'on appelle un **context switch**.

Lorsqu'une **commutation** doit être effectuée, le contexte du processus en cours (P_1) est sauvegardé. Le traitement effectif consiste à effectuer un *ordonnancement*, c'est-à-dire sélectionner le processus qui va recevoir le processeur (P_2). La restauration du contexte de P_2 a pour conséquence sa **continuation**, le processus P_1 pouvant transiter soit vers l'état **prêt**, **bloqué** ou **zombi**, selon la raison pour laquelle il a perdu le processeur.



(a) Exécution en cours de P_1 (b) Suspension bloquante de P_1 (c) Blocage de P_1

(d) **Context switch** vers P_2 (e) Continuation de P_2 (f) Exécution en cours de P_2

FIGURE 3.10 – Transitions d'état pour la commutation

Le type de **continuation** (reprise ou poursuite) dont fait l'objet P_2 dépend des circonstances qui l'avait amené à perdre le processeur précédemment. Par exemple, si P_2 avait perdu le processeur en raison de l'expiration de son *quantum* de temps (*cfr. Round-Robin*), l'interruption sur laquelle repose cette expiration donne lieu à une poursuite de P_2. Si P_2 avait perdu le processeur en raison d'un défaut de page pour lequel aucune page ne pouvait être remplacée, il se sera retrouvé dans l'état **bloqué** jusqu'à la libération d'un cadre. Dans ce cas de

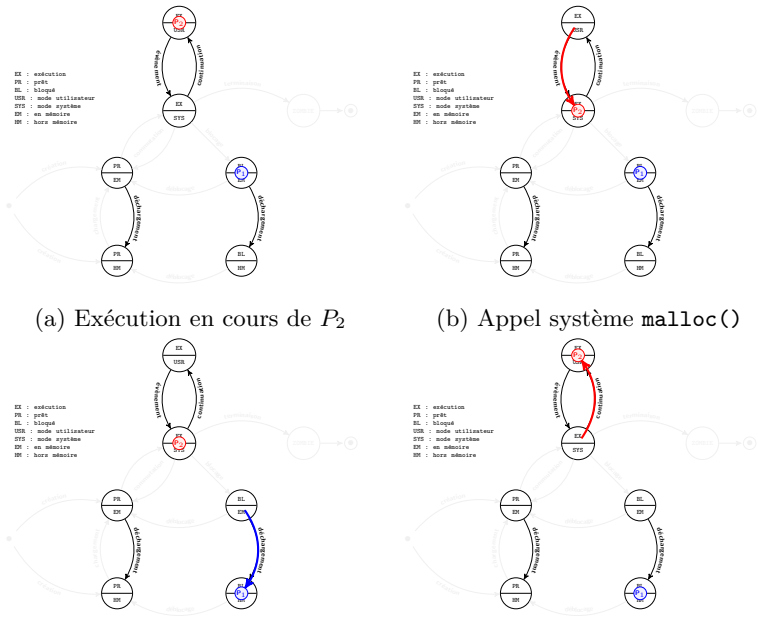
figure, lorsque P_2 reçoit le processeur, une reprise est nécessaire pour retenter l'opération d'accès mémoire ayant provoqué son défaut de page.

Le traitement d'une **commutation** présente un coût dissimulé. Suite à un **context switch**, il est nécessaire d'invalider et de purger le contenu des différentes mémoires caches du processeur (*e.g.* L1, L2, TLB) ce qui signifie que le processus qui reçoit le processeur ne pourra bénéficier de l'effet positif des caches qu'après un certain temps. Les premières tentatives d'accès mémoire (*e.g. fetch*, `lw`, `sw`) vont nécessiter d'aller chercher le contenu dans la mémoire principale. Pour un système avec pagination, les premières opérations de traduction d'adresses virtuelles se feront vis-à-vis d'entrées de la table des pages qui ne se trouvent pas dans la TLB et qui nécessiteront de réaliser des accès en mémoire principale pour pouvoir les effectuer.

Déchargement

Le **déchargement** d'un processus peut survenir lorsque deux conditions sont réunies. Premièrement, le processus en cours d'**exécution** présente un besoin de mémoire physique (*e.g.* défaut de page, `malloc`). Deuxièmement, l'état d'occupation de la mémoire physique ne présente pas de mémoire libre en suffisance pour donner satisfaction à cette demande.

Dans un système qui permet la préemption de mémoire, c'est-à-dire la possibilité de prendre de la mémoire physique à un processus pour la donner à un autre, un processus qui n'est pas en cours d'**exécution** peut perdre de la mémoire physique en faveur de celui qui occupe le processeur. Si un processus doit exister entièrement en mémoire physique (*e.g.* adressages absolu ou relatifs), lorsqu'un processus perd de la mémoire, l'intégralité de sa mémoire physique est perdue. Si un processus peut exister partiellement en mémoire physique (*e.g.* segmentation, pagination), il peut perdre une partie de sa mémoire physique.



(a) Exécution en cours de P_2 (b) Appel système `malloc()`

(c) Déchargement de P_1 (d) Continuation de P_2

FIGURE 3.11 – Transitions d'état pour le déchargement

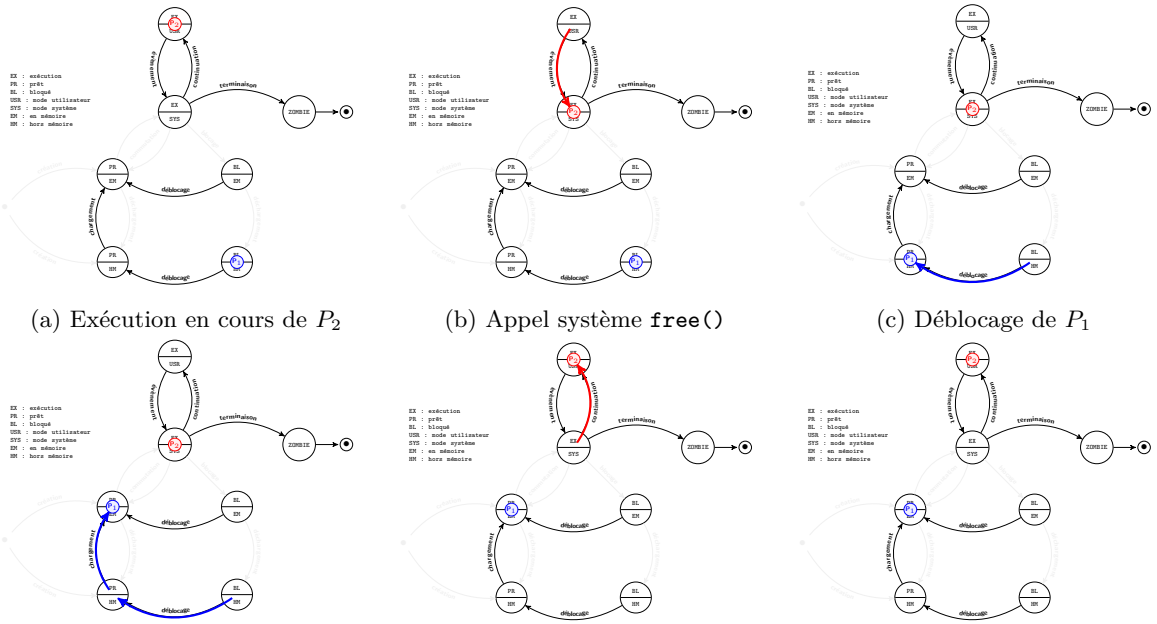
Considérons un processus P_2 en cours d'**exécution** et un processus P_1 actuellement bloqué pour une raison quelconque (Figure 3.11a). Lorsque le processus P_2 réalise une demande d'allocation mémoire (*i.e.* `malloc()`), l'interruption logicielle correspondante provoque un déroutement vers le système d'exploitation (Figure 3.11b). Dans le cadre de la gestion de la mémoire disponible (*cfr.* Chapitre 2), supposons qu'il s'avère nécessaire de prendre tous les cadres appartenant au processus P_1 pour les donner à P_2. Cette préemption sur la mémoire de P_1 s'accompagne d'un déchargement complet de celui-ci de la mémoire physique (*cfr.* Figure 3.11d). En supposant que tous les cadres ayant été transférés de P_1 à P_2, ce dernier peut faire l'objet d'une **continuation** avec poursuite dans le code du programme correspondant.

Déblocage

La transition d'un processus vers l'état **bloqué** sous-entend la possibilité ultérieure pour ce processus de revenir vers l'état **prêt**. Ce **déblocage** est rendu possible par la survenance d'un **évènement** particulier qui rend possible la **continuation** du processus **bloqué**.

Le processus peut être **bloqué** en attente de la fin d'une entrée-sortie (*e.g.* `read`). Lorsque le périphérique en charge de cette entrée-sortie la termine effectivement, celui-ci génère une interruption vers le processeur qui donne lieu à un déroutement. Il incombe dès lors au système d'exploitation de déterminer le processus qui est concerné par cette interruption. Ce processus peut dès lors faire l'objet d'un **déblocage** pour pouvoir éventuellement recevoir le processeur lorsque celui-ci est libéré.

Le processus peut être **bloqué** en attente de mémoire physique (*e.g.* `malloc`, défaut de page). L'évènement débloquant sera la libération par le processus en cours d'**exécution** d'une partie de sa mémoire (*e.g.* `free`) ou bien de l'intégralité de celle-ci suite à une **terminaison** (*e.g.* `exit`). Comme il est possible que plusieurs processus soient **bloqués** en attente de mémoire, la question se pose de savoir à quel processus allouer la mémoire physique qui vient d'être libérée.



(a) Exécution en cours de P_2 (b) Appel système `free()` (c) Déblocage de P_1

(d) Chargement de P_1 (e) Continuation de P_2 (f) Exécution en cours de P_2

FIGURE 3.12 – Transitions d'état pour le déblocage

Considérons un processus P_2 qui est en cours d'**exécution** ainsi qu'un processus P_1 qui a subi un **blocage** en raison d'une demande d'allocation mémoire à un moment où il n'y avait pas suffisament de cadres libres (Figure 3.12a). Supposons également qu'entre-temps, le processus P_1 a perdu tous ses cadres et qu'il a donc subi un **déchargement** complet de sa mémoire physique. Lorsque le processus P_2 effectue une demande de libération à travers un appel système `free()` (*cfr.* Figure 3.12b), l'interruption logicielle amène à un déroutement vers le système d'exploitation. Supposons qu'une partie des cadres libérés est donnée au processus P_1 qui en avait besoin ; celui-ci fera l'objet d'un **déblocage** (*cfr.* Figure 3.12c) et d'un **chargement** (*cfr.* Figure 3.12d) lui permettant de recevoir le processeur lorsque celui-ci sera libéré. Entre temps, le processus P_2 fera l'objet de la **continuation** de son exécution à travers une poursuite (Figure 3.12e).

3.2 Ordonnancement

La problématique de l'ordonnancement dans le cadre de la gestion des processus est celle de la répartition du temps processeur entre les processus du systèmes qui sont **prêts**. Plusieurs circonstances possibles dans l'évolution des processus sont appropriées pour solliciter l'ordonnanceur. Lorsqu'un processus engendre un autre processus (*cfr.* création), qui du parent ou de l'enfant devrait reprendre le processeur ? Lorsque le processus en cours quitte l'état **exécution** pour devenir **prêt**,**bloqué** ou **zombie** (*cfr.* suspension, terminaison), quel processus devrait transiter vers l'état **exécution** ?

D'autre part, si un processus passe de l'état **bloqué** à l'état **prêt** (*cfr.* déblocage), est-il approprié de remplacer le processus en cours par celui-ci ? Si le processus devenu **prêt** est plus important que celui en cours, un telle préemption s'avère légitime.

Dans cette section, nous allons étudier différentes stratégies d'ordonnancement pour trois types de systèmes qui présentent des spécificités et des hypothèses propres.

3.2.1 Systèmes batch

Un système de type batch est orienté vers l'exécution d'un nombre conséquent de tâches de manière régulière (*i.e.* journalière, hedmomadaire, mensuelle). Chaque tâche représente un traitement relativement simple qui doit être appliqué sur une quantité importante de données. Dans un tel système, des contraintes sont imposées sur la nature des traitements que les tâches peuvent effectuer. Il existe une garantie de terminaison sur les tâches (*i.e.* pas de tâche infinie) et il est possible d'estimer de manière raisonnablement fiable le temps que nécessitera une tâche donnée. Enfin, les tâches peuvent engendrer des tâches qui réaliseront des sous-traitements.

Plusieurs propriétés sont désirables concernant la stratégie d'ordonnancement utilisée. Une équité entre les tâches devrait être garantie ; celles-ci devraient reçevoir une portion des ressources du système en adéquation avec leurs besoins. Dans ce contexte, il est désirable de distribuer les ressources de façon à produire une utilisation optimale du système (*e.g.* limiter la sous-utilisation d'une ressource). Enfin, il n'est pas acceptable de voir certaines tâches **prêtes** ne jamais s'exécuter malgré qu'elles puissent logiquement le faire (*i.e.* famine).

Afin d'évaluer les performances d'une stratégie d'ordonnancement, plusieurs métriques sont possibles. Le *throughput* représente le nombre de tâches par unité de temps (*e.g.* minute, heure). Une autre métrique, le *turnaround*, représente le temps moyen entre la création et la terminaison (*cfr.* temps d'exécution). Une bonne stratégie devrait conjointement maximiser le *throughput* et minimiser le *turnaround*.

First-Come-First-Served (FCFS)

La première stratégie consiste à attribuer le processeur dans l'ordre d'arrivée à l'état **prêt**. Une structure de file est associée à cet état et à chaque fois qu'une tâche arrive dans cet état, son identifiant est ajouté à la fin de cette file. Lorsqu'un ordonnancement doit avoir lieu, la tâche dont l'identifiant est à l'avant de la file est choisie pour recevoir le processeur.

Cette stratégie est équitable entre les tâches et évite les situations de famine étant donné que n'importe quelle tâche **prêt** finira par arriver à l'avant de la file une fois que toutes celles qui la précede auront reçu le processeur et terminé leurs traitements. Cependant, cette stratégie ne garantie pas une utilisation optimale des ressources étant donné que si une tâche *CPU-bound* se trouve avant des tâches *IO-bound*, les opérations d'entrées-sorties ne pourront pas être déclenchées étant donné que la tâche *CPU-bound* recevra le processeur en premier.

Il est possible d'illustrer le fait que le FCFS ne minimise pas le *turnaround* en considérant deux tâches ; T_1 avec une durée de 10 unités de temps, T_2 avec une durée de 4 unités de temps. Si la tâche T_1 est **prête** se trouve avant la tâche T_2 dans la file du FCFS, l'ordonnancement produit sera $T_1; T_2$. Si leurs instants de création coïncident sur l'instant 0, la tâche T_1 se terminera à l'instant 10 et la tâche T_2 à l'instant 14, ce qui donne un *turnaround* de 12. Cependant, pour l'ordonnancement $T_2; T_1$, la tâche T_2 se terminerait à l'instant 4 et la tâche T_1 à l'instant 14, ce qui produit un *turnaround* de 9.

Shortest Job First (SJF)

Une deuxième stratégie repose sur le fait que la durée des tâches pour un système batch peut être estimée de manière raisonnablement fiable. La structure de file associée à l'état **prêt** peut être ordonnée sur base de la durée des tâches ; la plus courte se trouvera toujours au début, la plus longue à la fin. Lorsqu'une tâche arrive à l'état **prêt**, elle doit être insérée dans la file juste avant la première tâche qui présente une durée plus longue.

Lorsqu'un ordonnancement est sollicité, la tâche dont l'identifiant se trouve à l'avant de la file est choisie pour recevoir le processeur.

L'utilisation de cette stratégie permet de minimiser le *turnaround* comme l'illustration pour FCFS permet de le voir. En revanche, elle ouvre la possibilité de voir une situation de famine apparaître si des tâches de courtes durées arrivent continuellement à l'état **prêt**. De ce fait, une inéquité est introduite vis-à-vis des tâches plus longues.

Shortest Remaining Task First (SRTF)

Une troisième stratégie repose sur la durée estimée des tâches en considérant le temps processeur déjà utilisé. La structure de file associée à l'état **prêt** peut être ordonnée sur base de la durée *restante* des tâches ; celle qui a encore besoin du moins de temps se trouvera toujours au début, celle qui a encore besoin du plus de temps à la fin. Lorsqu'une tâche arrive à l'état **prêt**, elle doit être insérée dans la file juste avant la première tâche qui présente une durée restante plus longue. Lorsqu'un ordonnancement est sollicité, la tâche dont l'identifiant se trouve à l'avant de la file est choisie pour recevoir le processeur. D'autre part, si une tâche plus courte que la tâche en cours arrive à l'état **prêt**, la préemption permet de lui donner le processeur en priorité.

L'utilisation de cette stratégie permet de minimiser le *turnaround* suivant le même argument que le SJF. En revanche, elle ouvre la possibilité de voir une situation de famine apparaître si des tâches de courtes durées arrivent continuellement à l'état **prêt**. De ce fait, une inéquité est introduite vis-à-vis des tâches plus longues.

3.2.2 Systèmes interactifs

Un système interactif est centré autour d'un utilisateur humain qui interagit avec le système pour exploiter ses ressources. Il est possible d'avoir plusieurs utilisateurs qui sont tous connectés et qui disposent de plusieurs processus chacun. Les utilisateurs ont le droit de créer des nouveaux processus selon leurs besoins. La logique des processus n'est pas soumise à de fortes contraintes ; un processus peut tourner indéfiniment et il est difficile de prédire de manière fiable combien de temps prendra l'exécution d'un processus donné.

Plusieurs propriétés sont désirables pour l'ordonnancement d'un tel système. Une équité entre les utilisateurs et leurs processus devrait être respectée ; chaque utilisateur et chacun de ses processus devrait reçevoir une portion des ressources en adéquation avec ses besoins. Il est également désirable d'utiliser toutes les ressources du système de manière optimale et d'éviter la sous-utilisation d'une ressource particulière dans la mesure du possible. Enfin, il n'est pas acceptable d'avoir un processus **prêt** qui ne reçoit jamais le processeur pour pouvoir s'exécuter (*i.e.* famine).

Afin d'évaluer les performances d'une stratégie d'ordonnancement pour un système interactif, plusieurs métriques peuvent être considérées. La *latence* représente le temps qui s'écoule entre une action de l'utilisateur et la réaction du système. Par exemple, lorsque l'utilisateur clique sur un bouton (action), l'application ciblée doit mettre à jour le contenu affiché à l'écran pour présenter une menu particulière (réaction). La *proportionalité* représente l'adéquation vis-à-vis des attentes de l'utilisateur. Par exemple, lors de l'exécution d'une requête SQL, l'utilisateur s'attend à recevoir les résultats endéans un certain temps. Une bonne stratégie d'ordonnancement visera à maximiser la latence et à maximiser la proportionalité.

Round-Robin

Une première stratégie consiste à découper le temps processeur en *slots* individuels ayant une durée fixe, appelée le *quantum* de temps, et à distribuer les *slots* consécutifs entre les processus qui sont **prêts**. Il est possible d'implémenter cette stratégie à l'aide d'une liste cyclique associée à l'état **prêt**. Lorsqu'un processus arrive à l'état **prêt**, son identifiant est inséré dans la liste à la fin. Lorsqu'un ordonnancement doit avoir lieu, le processus dont l'identifiant se trouve en tête de liste est choisi pour recevoir le processeur. À ce moment, le système d'exploitation configure un *timer* pour générer une interruption lorsque le *quantum* de temps expire. Lorsque cette interruption survient, le processus **prêt** en tête de liste est sélectionné et remplace le processus en cours qui est envoyé à l'état **prêt** et à la fin de la liste.

L'utilisation du *Round-Robin* garantit une répartition équitable du processeur entre les différents processus. En effet, une fois qu'un processus arrive à l'état **prêt**, il devra au plus attendre autant de *quanta* qu'il y'a de processus avant lui dans la liste avant de recevoir le processeur.

Le choix de la valeur du *quantum* joue un rôle important dans le délai avant qu'un processus **prêt** qui doit traiter une action d'un utilisateur ne puisse s'exécuter. La Figure 3.13 illustre le cas où 4 processus sont

ordonnancés suivant une stratégie de *Round-Robin* avec un *quantum* de 6 ms. Si une frappe au clavier à destination du processus D survient au début du premier *slot* alloué à A, il faudra attendre 18 ms avant que D ne reçoive le processeur pour traiter l'action de l'utilisateur.

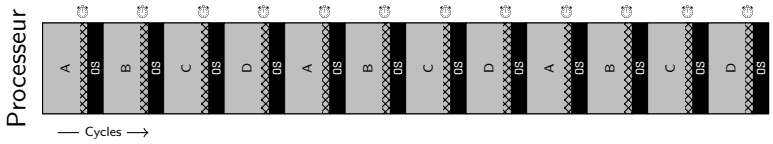


FIGURE 3.13 – Round-Robin pour 4 processus avec un *quantum* de 6 ms.

Si le *quantum* de temps choisi est de 24 ms (*cfr.* Figure 3.14) et que la frappe au clavier survient au début du premier *slot* alloué à A, il faudra attendre 72 ms avant que D ne reçoive le processeur et puisse traiter cette action de l'utilisateur.

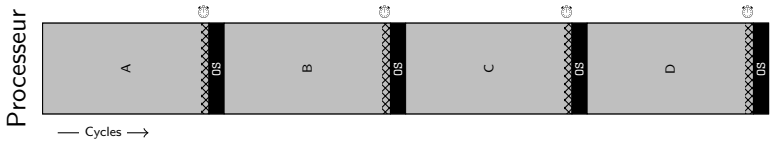


FIGURE 3.14 – Round-Robin pour 4 processus avec un *quantum* de 24 ms.

Un autre aspect à prendre en compte dans le choix du *quantum* de temps est lié au temps que prend la **commutation** entre processus. Si le temps nécessaire pour changer le processus en cours est de 2 ms, un *quantum* de 6 ms a pour conséquence de 25% du temps du processeur est passé à effectuer des commutations. En revanche, si le *quantum* est de 24 ms, cette occupation du processeur tombe à environ 8%.

Priority Scheduling

Une deuxième stratégie repose sur l'attribution d'un niveau de priorité à chaque processus pour refléter l'importance relative entre les différents processus du système. Cette priorité peut être une valeur statique, fixée par l'utilisateur ou au démarrage d'un processus ou bien constituer un attribut dynamique que l'ordonnanceur maintient à jour au cours du temps. Ce caractère dynamique peut être utilisé pour catégoriser les processus comme étant *CPU-bound* ou *IO-bound* et favoriser les seconds dans la réception du processeur afin d'accroître l'utilisation des ressources.

Le niveau de priorité va jouer un rôle double. Premièrement, il permet de déterminer la proportion du temps CPU auquel a droit un certain processus (*e.g.* 4 *quanta* pour haute priorité, 1 *quantum* pour basse priorité). Deuxièmement, la préemption devient possible sur base de la priorité relative entre le processus en cours et un processus qui arrive à l'état **prêt**.

Il est également possible d'utiliser des *classes* de priorité (*e.g. timesharing*, *realtime*) et d'appliquer des stratégies d'ordonnancement différentes selon la classe considérée. Par exemple, des processus *realtime* sont soumis à des exigences beaucoup plus fortes en matière de *timing* et doivent être traités d'une manière différente en matière d'ordonnancement.

Guaranteed Scheduling

La troisième stratégie se fixe pour objectif la répartition au *pro-rata* du nombre de processus. L'ordonnanceur calcule en continu un **délai garanti** pour chaque processus qui existe suivant la formule suivante ;

$$\text{Délai garanti} \;=\; \frac{\text{Temps depuis sa création}}{\text{Nombre de processus}}$$

L'ordonnanceur doit dès lors réaliser un suivi du temps d'exécution reçu par chaque processus et maintenir à jour le rapport entre le temps reçu et le délai garanti.

$$\rho \;=\; \frac{\text{Temps reçu}}{\text{Délai garanti}}$$

Lorsqu'un ordonnancement doit être effectué, le processus ayant la valeur minimale du rapport suivant est choisi pour recevoir le processeur. Le temps accordé sera calculé pour ramener ce rapport au dessus de 1.

Fair-Share Scheduling

Une considération transversale à la stratégie d'ordonnancement est que les utilisateurs doivent également être considérés dans la sélection d'un processus afin d'éviter d'en favoriser certains. Considérons un utilisateur 1 qui possède un processus (A) et un utilisateur 2 qui possède deux processus (B, C). Tous les processus sont toujours **prêts** et apparaissent dans la liste dans l'ordre $A; B; C$. La Figure 3.15 présente l'ordonnancement produit par le *Round-Robin* appliqués à ces trois processus. Étant donné que le *Round-Robin* ne tient pas compte du propriétaire d'un processus, l'utilisateur 1 se retrouvera à recevoir 33% du temps CPU alors que l'utilisateur 2 en reçoit 66%. Les utilisateurs ayant plus de processus sont plus favorisés en matière d'utilisation du processeur.

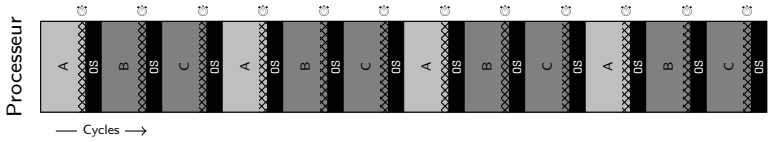


FIGURE 3.15 – Équité entre processus avec inéquité entre utilisateurs. Les processus de l'utilisateur 1 sont colorés en gris clair, ceux de l'utilisateur 2 en gris foncé.

Une solution simple à ce problème consiste à utiliser un *Round-Robin* à deux niveaux. Les utilisateurs sont organisés suivant une liste cyclique et les processus sont regroupés en une liste cyclique par utilisateur. Lorsqu'un ordonnancement doit avoir lieu, l'utilisateur suivant dans la liste cyclique des utilisateurs est sélectionné et le processus suivant dans la liste de ses processus est choisi. Il en résulte l'ordonnancement présenté à la Figure 3.16 qui permet de voir que l'équité est restauré entre les utilisateurs qui reçoivent chacun 50% du temps CPU. Ceci se fait au sacrifice de l'équité entre les processus ; les processus B et C reçoivent désormais 25% du temps CPU chacun contre 33% dans la figure précédente.

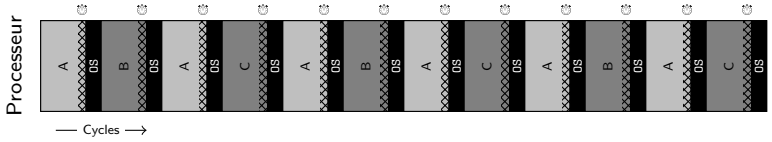


FIGURE 3.16 – Équité entre utilisateurs. Les processus de l'utilisateur 1 sont colorés en gris clair, ceux de l'utilisateur 2 en gris foncé.

3.2.3 Systèmes temps réel

Dans un tel système, le temps qui passe dans le monde réel devient un paramètre déterminant le caractère correct du comportement. Ce type de système intervient dans des situations nécessitant un contrôle précis où il ne suffit pas de réaliser un calcul correct mais également de réagir endéans un temps bien déterminé.

Par exemple, un système de défense anti-missile [2] doit continuellement scanner son périmètre de défense pour déterminer tous les objets volants qui s'y trouvent. Sur base de ces données, il doit calculer la trajectoire d'un objet hostile pour l'intercepter et réagir pour réaliser l'interception. Pour ce faire, le système doit déterminer un point sur cette trajectoire qui sera atteint par l'objet hostile en même temps que les balles qu'il va devoir tirer. Si le système prend ensuite un peu trop de temps avant de tirer, il manquera la cible.

Dans un système temps-réel, l'ensemble des évènements qui peuvent se produire sont connus (*e.g.* mesures de senseurs, positionnement d'un actuateurs) et un traitement bien défini est associé à chaque évènement. Ces traitements doivent avoir une durée qui est prévisible avec une précision élevée. Les traitements sont soumis à des échéances strictes (*i.e. hard deadlines*) qui doivent être respectées ou des échéances flexibles (*i.e. soft deadlines*) qui peuvent être non-respectées. L'ensemble des évènements est caractérisé par leur régularité ; les évènements périodiques surviennent à intervalle régulier (*e.g. blip* radar) tandis que les évènements apériodiques peuvent survenir de façon imprévue (*e.g.* alarme). Pour un évènement périodique, le temps nécessaire pour effectuer son traitement est dénotée par C_i et l'intervalle de temps entre deux occurrences de cet évènement (*i.e.* période) est dénoté par P_i. Pour un ensemble d'évènements périodiques, un système temps-réel est dit *ordonnançable* seulement si la condition suivante est vérifiée ;

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leqslant 1$$

2. *e.g.* C-RAM System (`https://www.youtube.com/watch?v=MMFzlwzFgKw`)

3.3 Multi-threading

Nous avons vu à travers le début de ce chapitre qu'un processus représente la mise en activité d'un programme. Au cours de son existence, un processus va acquérir un ensemble de ressources (*e.g.* mémoire, fichiers ouverts) qu'il manipulera dans le cadre de ses traitements. De plus, le processus va suivre un *fil d'exécution* à travers le code du programme, dont l'état à tout moment sera capture par les valeurs des différents registres ainsi que du contenu de la *stack*. Un processus constitue une entité d'exécution indépendante des autres processus qui peut progresser à son propre rythme. Certaines ressources peuvent être partagées entre plusieurs processus, comme les descripteurs de fichiers ouverts auxquels n'importe quel processus parent et enfant ont accès. De même, certaines ressources sont détenues exclusivement par un seul processus ; l'espace d'adressage d'un processus étant propre à celui-ci, bien que la possibilité existe de partager certaines régions de mémoire.

Ce modèle simplifié pose problème si l'on considère qu'une application moderne est composée de nombreuses activités indépendantes les unes des autres. Par exemple, un éditeur de texte doit prendre en charge la gestion de l'interface graphique pour dessiner le contenu de son interface à l'écran. D'autre part, il doit s'occuper de gérer l'édition et la sauvegarde des fichiers qu'il manipule. Un autre exemple serait un serveur web qui est en charge de répondre à des requêtes provenant d'Internet et pouvant gérer connexions de plusieurs utilisateurs différents en parallèle.

Bien que la notion de processus permettent de séparer différentes entités d'exécution, celle-ci présente plusieurs limitations. Par exemple, sur une machine multi-processeurs et/ou *multi-cores*, un processus simple n'est pas en mesure d'exécuter plusieurs traitements en parallèle. D'autre part, si un traitement particulier d'un processus provoque un **blocage**, le processus en question n'est pas en mesure de passer à un autre traitement dont il a la responsabilité pendant ce temps. Enfin, les opérations de **création** et de **commutation** présentent un certain coût direct (*cfr.* ordonnancement) et indirect (*cfr.* invalidation des *caches*, changement des tables de projection). La notion de *threads* (*a.k.a.* processus léger) permet d'adresser ces limitations en introduisant un découpage des traitements qui, au sein d'un processus, peuvent être effectué de manière indépendante les uns des autres.

3.3.1 Utilitaire de triage

Considérons un utilitaire de triage qui prend en paramètre une liste de noms de fichiers à la ligne de commande pour trier le contenu de ces différents fichiers et ré-ecrire ce contenu trié sur l'original. Le *listing* d'un tel utilitaire est présenté à la Figure 3.17. Les lignes $13-14$ prennent en charge l'ouverture du fichier à trier pour l'itération courante. La ligne 16 gère la lecture du contenu du fichier (avec maximum `TAILLE` octets lus). Le triage est invoqué à la ligne 18 ; une fonction de triage implémentant le *bubble-sort* est supposée exister. Enfin, la ré-écriture du contenu trié est effectuée aux lignes $20-21$ avant la clôture du fichier d'origine à la ligne 23.

```
1  #include <stdio.h>      // perror()
2  #include <fcntl.h>      // open()
3  #include <unistd.h>     // read(), write(), close()
4  #include <stdlib.h>     // EXIT_FAILURE, EXIT_SUCCESS
5  #include <bubbleSort.h> // sort()
6
7  #define TAILLE 2048
8
9  int tableau[TAILLE];
10
11 int main(int argc, char* argv[]) {
12   for(int f = 1; f < argc; f++) {
13     int fd = open(argv[f], O_RDWR);
14     if (fd == -1) { perror("open"); return EXIT_FAILURE; }
15
16     int octets_lus = read(fd, tableau, TAILLE);
17
18     sort(tableau, octets_lus / 4);
19
20     lseek(fd, 0L, 0);
21     write(fd, tableau, octets_lus);
22
23     close(fd);
24   }
25
26   return EXIT_SUCCESS;
27 }
```

FIGURE 3.17 – Utilitaire de triage *mono-thread*.

Cette implémentation présente un fonctionnement purement séquentiel où chaque itération de la boucle de la ligne 12 est exécuté à la suite de l'itération précédente (*cfr.* Figure 3.18). Lorsque le système d'exploitation décide de commuter vers ce processus, le processeur commencera à exécuter le code du programme qui commence sur la boucle `for`. La première phase de l'itération consistera à gérer l'ouverture de fichier qui donnera lieu à un appel système. Un déroutement se fera vers le système d'exploitation (représenté en gris foncé) qui s'exécutera jusqu'à la conclusion de l'opération d'ouverture. Au terme de ce traitement, la poursuite du processus se fera avec la valeur de retour disponible pour gérer l'éventuel échec de l'ouverture. Le processus entrera ensuite dans la phase de gestion de la lecture qui donnera lieu à un autre apple système et un nouveau déroutement vers le système d'exploitation. Une fois la lecture conclue par le système d'exploitation, la poursuite du processus pourra se faire qui permettra au processus d'aborder la phase de triage à proprement parler. Une fois le tableau trié, une opération d'écriture vers le fichier d'origine pourra être demandée à travers un nouvel appel système. Au terme de l'écriture, la poursuite permettra au processus de conclure l'itération courante et de passer à la suivante.

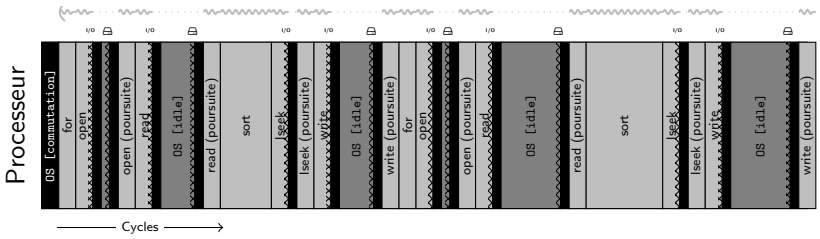


FIGURE 3.18 – Exécution de deux itérations de l'utilitaire de triage.

Au vu des traitements que cet utilitaire réalise, il doit apparaître évident que le traitement d'un fichier peut se faire indépendemment du traitement d'un autre fichier. Cette idée simple permet de découper la logique du programme *mono-thread* en identifiant les activités impliquées (initialisation, ouverture, tri, écriture) et en restructurant la logique de l'utilitaire en plusieurs *threads*.

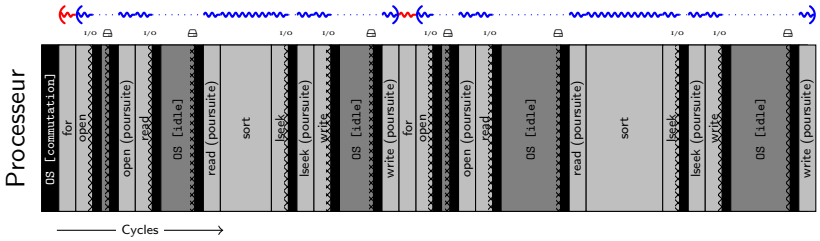


FIGURE 3.19 – Identification des traitements indépendants de l'utilitaire de triage. Le *thread dispatcher* est représenté en rouge et les *threads* de triage en bleu.

La ré-organisation des traitements en une version *multi-threads* est présentée à la Figure 3.20. Le traitement d'un *thread* de tri est placé dans une fonction distincte `sortFile()` qui prend en paramètre un nom de fichier. Cette fonction représente le fil d'exécution bleu. Par contraste, le *thread dispatcher* est le *thread* de la fonction `main()` qui se charge de créer les différents *threads* de tri (lignes $38-39$) avant de se mettre en attente de leurs terminaisons (lignes $40-41$). Étant donné que les *threads* de tri peuvent s'exécuter indépendemment les uns des autres, il est nécessaire qu'ils disposent chacun d'un espace de mémoire qui leur est dédié pour recevoir le contenu du fichier qu'il doivent trier. Cet espace est obtenu par une allocation dynamique de mémoire (ligne 12) mais pourrait tout à fait prendre la forme d'un espace de mémoire statique.

3.3.2 Gestion des commutations

Les coûts associés aux *threads* sont réduits par rapport à des processus. La **création** d'un processus nécessite d'initialiser un descripteur de processus, une table des segments, une table des pages, d'effectuer une allocation de mémoire pour recevoir les données du processus. Par contraste, tous les *threads* d'un processus ont accès à toutes les ressources détenues par le processus duquel ils relèvent. De ce fait, la **création** d'un *thread* nécessite simplement d'effectuer une allocation de mémoire pour héberger la *stack* du *thread* engendré. D'autre part, la **commutation** entre processus nécessite de changer la table des segments et/ou la table des pages courante et de purger les caches du processeur. Par contraste, la **commutation** entre *threads* d'un même processus nécessite simplement de changer la *stack* qui serait restaurée dans le cadre de la **continuation**.

```c
#include <stdio.h>        // perror()
#include <fcntl.h>        // open()
#include <unistd.h>       // read(), write(), close()
#include <stdlib.h>       // EXIT_FAILURE, EXIT_SUCCESS
#include <pthread.h>      // pthread_create(), pthread_exit()
#include "bubbleSort.h" // sort()

#define TAILLE 2048
#define FILEMAX 16

void* sortFile(void* filename) {
  int *tableau = malloc(TAILLE);
  if (tableau == NULL) {
    perror("malloc");
    pthread_exit(NULL);
  }

  int fd = open((char*) filename, O_RDWR);
  if (fd == -1) {
    perror("open");
    free(tableau);
    pthread_exit(NULL);
  }

  int octets_lus = read(fd, tableau, TAILLE);
  sort(tableau, octets_lus / 4);

  lseek(fd, 0L, 0);
  write(fd, tableau, octets_lus);

  close(fd);
  free(tableau);
  pthread_exit(NULL);
}

int main(int argc, char* argv[]) {
  pthread_t threads[FILEMAX];
  for(int f = 1; f < argc && f <= FILEMAX; f++)
    pthread_create(&threads[f], NULL, sortFile, (void*) argv[f]);
  for(int f = 1; f < argc && f <= FILEMAX; f++)
    pthread_join(threads[f], NULL);
  return EXIT_SUCCESS;
}
```

FIGURE 3.20 – Utilitaire de triage *multi-thread*.

Il est possible d'implémenter la gestion des *threads* à deux niveaux ; *user-level* ou *kernel-level*. Lorsque l'implémentation est faite en *user-level*, le système d'exploitation n'a pas conscience de l'existence des *threads* au sein du processus. Une ordonnanceur propre au processus peut être utilisé pour répartir le temps processeur reçu par le processus entre ses différents *threads*. Cependant, il est difficile de permettre la préemption entre les *threads* et le **blocage** d'un *thread* (qui demanderait une ouverture de fichie) a pour conséquence le **blocage** du processus (et donc de tous ses *threads*).

Lorsque l'implémentation de la gestion des *threads* est faite en *kernel-level*, la table des processus devient essentiellement une table des *threads*. La stratégie d'ordonnancement utilisée par le système d'exploitation est appliquée à l'ensemble des *threads* du système. Lorsqu'un ordonnancement est sollicité, le système d'exploitation peut choisir en priorité de donner le processeur à un autre *thread* du même processus pour bénéficier du coût de **commutation** réduit associés aux *threads* mais néanmoins de réaliser parfois une **commutation** vers un *thread* d'un autre processus pour éviter des situations de famine. Dans une telle implémentation, le **blocage** d'un *thread* n'a pas pour conséquence le **blocage** du processus auquel il appartient.

Chapitre 4 Concurrence

4.1 Problématique

La notion de processus et la notion de *threads* rend possible la parallélisation de certains traitements. Comme nous allons le voir au cours de ce chapitre, ces nouvelles possibilités amènent également leur lot de problèmes qui doivent être adressés pour pouvoir accélérer la vitesse des calculs tout en garantissant qu'ils sont réalisés correctement.

4.1.1 Mécanismes de communication

Il existe plusieurs manières de faire communiquer différentes entités d'exécution indépendantes (*e.g.* processus, *threads*. Les problématiques de *race condition* qui nous intéresserons apparaîtront lorsque des **accès concurrents sur un objet partagé** sont utilisés. On entend par accès concurrents les situations où les séquences d'instructions qui touchent à un certain objet (*i.e.* lecture, écriture) peuvent être réalisées suivant n'importe quel entrelacement de ces séquences. Plusieurs *threads* d'un même processus ont tous accès à un espace d'adressage commun. En pratique, ces différents *threads* peuvent accéder à n'importe quelle variable globale (*e.g.* scalaire, tableau), région de mémoire, fichier ouvert qui est appartient au processus duquel ils relèvent. Plusieurs processus ont accès exclusivement à leur espace d'adressage respectifs. Cependant, il est possible à ces espaces de recouvrir des régions de mémoire physique qu'ils ont en commun (*e.g.* `mmap`, `shmget`). De même, tous les fichiers ouverts par un processus avant qu'il ne réalise un appel `fork()` sont partagés avec le processus enfant résultant.

Un autre mécanisme de communication, appelé *message passing* repose sur le transfert de messages entre les entités d'exécution considérées. Dans le cas local, plusieurs processus ou *threads* tournant sur le même système d'exploitation peuvent se passer des messages contenant, par exemple, des demandes de traitements et les réponses à ces demandes. Le contenu échangé devant être copié depuis l'espace d'adressage de l'émetteur du message vers l'espace d'adressage du récepteur, ceci pose problème si le volume de données échangés devient important. Dans le cas non-local, les deux entités d'exécution résident sur deux machines distantes, connectées par le biais d'un réseau de communication (*e.g.* Internet).

Dans le reste de ce chapitre, nous nous focaliserons uniquement sur la première catégorie de mécanismes de communication.

4.1.2 Race condition

Nous allons utiliser le programme de la Figure 4.1a pour illustrer la problématique de *race condition* ainsi que les tentatives de la solutionner. Ce programme est structuré en deux *threads* qui incrémentent de 1000 unités chacun une variable globale (*i.e.* `total`) avant d'en afficher la valeur (*cfr.* ligne 22). Lorsqu'un *thread* touche à cette variable partagée (ligne 12), les instructions exécutées qui implémentent cette opération sont celles des lignes $8-10$ de la Figure 4.1b. Le canevas important à garder à l'esprit en présence de programmation concurrente est celui de *read-modify-write*. Lorsque ce canevas est présent dans du code exécuté par différents *threads* ou processus sur un objet partagé, le risque existe qu'une *race condition* survienne. Dans l'exemple qui nous intéresse, le canevas est présent à travers la séquence des trois instructions ; `lw $t2, 0($t0))`, `addiu $t2, 1`, `sw $t2, 0($t0)` qui respectivement récupère la valeur actuelle de `total` depuis son endroit en mémoire principale, incrémente cette valeur stockée dans le registre `$t2` et enfin écrit la valeur incrémentée à l'endroit où se trouve la variable `total`. Un **interleaving** représente un entrelacement possible des instructions. Par exemple, si l'on considère la séquence `lw-addiu-sw` exécutée par deux *threads* distincts, deux *interleavings* possibles sont ;

```
lw ; addiu ; sw ; lw ; addiu ; sw
lw ; lw ; addiu ; addiu ; sw ; sw
```

L'exécution complète suivant le premier *interleaving* produira une incrémentation de deux unités de la variable `total` tandis que l'exécution du second produira une incrémentation d'une unité. Nous illustrons ce problème de manière plus détaillée aux Figures 4.2a et 4.2b.

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

#define NB_THREADS 2
pthread_t threads[NB_THREADS];

unsigned int total = 0;

void* ajouter(void* argument) {
  for(int i = 0; i < 1000; i++)
    total = total + 1;
  pthread_exit(NULL);
}

int main(int argc, char* argv[]) {
  for(unsigned int t = 0; t < NB_THREADS; t++)
    pthread_create(&threads[t], NULL, ajouter, NULL);
  for(unsigned int t = 0; t < NB_THREADS; t++)
    pthread_join(threads[t], NULL);

  printf("Total : %u\n", total);

  return EXIT_SUCCESS;
}
```

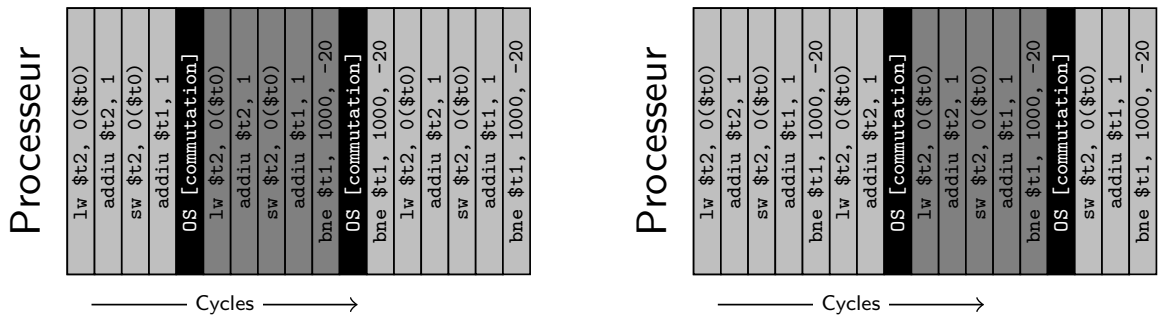
(a) Implémentation en langage `C`

```
  .data
TOTAL: .word 0
  .text
AJOUTER:
    la $t0, TOTAL
    li $t1, 0
BOUCLE:
    lw $t2, 0($t0)
    addiu $t2, 1
    sw $t2, 0($t0)
    addiu $t1, 1
    bne $t1, 1000, BOUCLE
    lw $v0, EXIT_SUCCESS
    jal pthread_exit
```

(b) Code `MIPS32` de la fonction `ajouter`

FIGURE 4.1 – Incrémentation par deux *threads*.

Considérons les deux *threads* exécutant respecivement deux itérations et une itération de la boucle de leur fonction principale. Supposons que la valeur de la variable `total` au début de ce scénario soit de 0. Au terme de l'exécution complète de ces trois itérations, la valeur de la variable `total` devrait être 3. La Figure 4.2a présente la situation où l'*interleaving* ne donne pas lieu à une *race condition* et la valeur obtenue de `total` est bien 3. Bien que le premier *thread* subisse une **commutation** pendant son exécution, celle-ci ne vient pas scindée la séquence de code manipulant la variable partagée. Par contraste, la Figure 4.2b présente le cas où l'*interleaving* produit une *race condition* et un résultat erroné. La seconde itération se fait de manière incomplète ; le premier *thread* a l'occasion de récupérer la valeur de `total` qui est de 1 à ce moment et incrémenter cette valeur dans le registre `$t2` (*i.e.* valeur de 2). Cependant, la **commutation** survient et provoque la perte du processeur en faveur du deuxième *thread*, la valeur de `$t2` étant sauvegardée dans le cadre de la sauvegarde du contexte. Le deuxième *thread* accompli la séquence complète d'incrémentation de la valeur de `total` qui était toujours de 1 à ce stade pour écrire une valeur de 2 à l'endroit où se trouve la variable `total`. Une nouvelle **commutation** survient, provoquant la perte du processeur et la restauration du contexte du premier *thread* (y compris du registre `$t2`). Celui-ci effectue dès lors l'opération d'écriture (*i.e.* `sw`) pour inscrire sa valeur de 2 à l'endroit où se trouve la variable `total`.



(a) Exécution ininterrompue des séquences `lw-addiu-sw`. (b) Exécution interrompue d'une séquence `lw-addiu-sw`.

Cette illustration met en évidence une solution directe au problème ; empêcher que la séquence de code qui manipule la variable partagée puisse être scindée pour permettre à l'autre *thread* d'entamer cette même séquence ; l'objectif serait de garantir l'atomicité de l'exécution de cette séquence.

4.2 Section critique

Une section critique représente une portion du code manipulant un objet partagé. Dans notre illustration précédente, la ligne consistant à l'incrémentation de la variable `total` constituait une section critique. Il est possible d'avoir une section critique couvrant plusieurs morceaux de code disjoints ; par exemple une fonction qui ajoute un élément dans une structure de données et une autre fonction qui retire un élément de cette structure de données relèvent ensemble de la même section critique. L'objectif de cette approche consiste à mettre en place une règlementation des accès en section critique afin d'offrir trois garanties importantes ; l'exclusion mutuelle, l'absence d'attente injustifiée et l'absence d'attente infinie. La garantie d'**exclusion mutuelle** constitue la propriété de *correctness* de cette approche (*i.e.* quand un *thread* **peut** entrer). Celle-ci prévoit que deux *threads* ne peuvent pas se trouver en même temps dans la section critique. Le fait qu'un *thread* soit en section critique doit empêcher l'entrée par n'importe quel autre *thread* dans cette section critique. La garantie d'**absence d'attente injustifiée** constitue une propriété complémentaire (*i.e.* quand un *thread* **doit** entrer) qui affirme qu'un *thread* ne peut attendre à l'entrée de la section critique que si celle-ci contient déjà un *thread*. Enfin, comme les *threads* d'une application peuvent continuellement arriver à l'entrée de la section critique, il est important de garantir que chacun de ces *threads* finiront par entrer et traverser la section critique.

Afin qu'une solution de règlementation des accès soit valide, il est important de ne faire aucune hypothèse concernant la vitesse ou le nombre de processeurs utilisés. La solution de section critique doit respecter les trois garanties pour tous les *interleavings* possibles sans porter de jugement sur la probabilité qu'un *interleaving* particulier se produise. Si un *interleaving* problématique peut se produire, il finira par se produire. La seule hypothèse que nous ferons concernant le processeur est que celui-ci ne travaille pas suivant une exécution *out-of-order* où l'ordre d'exécution effectif des instructions ne respecte pas l'ordre exprimé dans le code produit.

4.2.1 Solutions avec attente active

La première catégorie de solutions qui vont nous intéresser font intervenir une **attente active** où l'attente à l'entrée de la section critique prend la forme d'une boucle qui teste continuellement l'état d'une ou de plusieurs variables. Bien que ce type d'attente ne met pas à mal les garanties désirables de section critique, il représente un gaspillage de temps processeur lorsque le temps d'attente est long.

Variable verrou

Une première solution consiste à associer un `verrou` à l'objet partagé qui subit les accès concurrents pouvant prendre deux valeurs possibles ; `OUVERT` ou `FERMÉ`. La signification attendue de cette variable est que lorsque le `verrou` est `OUVERT`, cela signifie qu'il n'y a aucun *thread* dans la section critique. Lorsque le `verrou` est `FERMÉ`, il y'a actuellement un *thread* dans la section critique.

L'implémentation de cette approche est présentée dans la Figure 4.3. Lorsqu'un *thread* arrive à l'entrée de la section critique, il va devoir attendre *tant que* le `verrou` est `FERMÉ` (*cfr.* ligne 17). Si le `verrou` est `OUVERT`, le *thread* basculera celui-ci à `FERMÉ` (*cfr.* ligne 18) et se trouvera dès lors lui-même en section critique (*cfr.* ligne 21). Une fois qu'un *thread* sort de la section critique, il basculera le `verrou` à `OUVERT` (*cfr.* ligne 24).

```
11 unsigned int verrou = OUVERT;
12 unsigned int total = 0;
13
14 void* ajouter(int identifiant) {
15   for(int i = 0; i < 1000; i++) {
16     // Entrée en section critique
17     while(verrou == FERMÉ) {}
18     verrou = FERMÉ;
19
20     // Section critique
21     total = total + 1;
22
23     // Sortie de section critique
24     verrou = OUVERT;
25   }
26   pthread_exit(NULL);
27 }
```

FIGURE 4.3 – Solution avec `verrou` pour l'incrémentation.

L'**exclusion mutuelle** n'est pas garantie par cette solution. La variable `verrou` est un objet partagé qui est manipulé par les *threads* suivant le canevas *read* (*i.e.* test du `verrou`) et *write* (*i.e.* fermeture du `verrou`). Supposons que le `verrou` soit `OUVERT`. Imaginons un *thread* effectuant le test de la ligne 17 qui invalide la condition de boucle mais qu'avant de pouvoir exécuter la ligne 18 ce *thread* perd le processeur. Si le *thread* qui le remplace réalise également le test de la ligne 17, le `verrou` est toujours `OUVERT` bien que le premier *thread* soit déjà en route vers la section critique. À ce stade, rien n'empêche les deux *threads* d'entrer en section critique en même temps et de potentiellement produire un *interleaving* incorrect des instructions de la ligne 21.

Alternance stricte avec tour

Une deuxième solution consiste à associer un `tour` à l'objet partagé qui pourra prendre comme valeur l'identifiant du *thread* qui peut entrer en section critique. Pour cette solution, nous supposons que chaque *thread* dispose d'un identifiant unique compris entre `0` et `NB_THREADS-1`.

L'implémentation de cette approche est présentée dans la Figure 4.4. Lorsqu'un *thread* arrive à l'entrée de la section critique, il va devoir attendre *tant que* ce n'est pas son `tour` (*cfr.* ligne 14). Si c'est son `tour`, il pourra entrer dans la section critique (*cfr.* ligne 17). Une fois qu'un *thread* sort de la section critique, il passe le `tour` au suivant (*cfr.* ligne 20).

```
8  unsigned int tour = 0;
9  unsigned int total = 0;
10
11 void* ajouter(int identifiant) {
12   for(int i = 0; i < 1000; i++) {
13     // Entrée en section critique
14     while(tour != identifiant) {}
15
16     // Section critique
17     total = total + 1;
18
19     // Sortie de section critique
20     tour = (tour + 1) % NB_THREADS;
21   }
22   pthread_exit(NULL);
23 }
```

FIGURE 4.4 – Solution avec `tour` pour l'incrémentation.

L'**exclusion mutuelle** est garantie par cette solution. En effet, lorsque plusieurs *threads* arrivent à la ligne 14, la variable de `tour` ne permettra au plus qu'à un seul d'entre eux de passer la boucle d'attente active ; la condition étant invalidée uniquement pour le *thread* dont l'identifiant est dans la variable `tour`. Ce *thread* sera celui qui traverse la section critique et qui met à jour la variable `tour` en y mettant l'identifiant suivant.

L'**absence d'attente injustifiée** n'est pas garantie par cette solution. Le fait d'alterner strictement entre les *threads* à pour conséquence que chaque *thread* ne pourra exécuter qu'une itération de la boucle avant de bloquer sur l'attente active. Supposons un *thread* 0 qui attende à l'entrée de la section critique (*cfr.* ligne 14) et un *thread* 1 qui sort de celle-ci. Le *thread* 1 passe le `tour` à 0 juste avant de perdre le processeur en faveur du *thread* 0. Pour rendre plus évident le fait que le *thread* 1 n'est pas à l'entrée de la section critique, imaginons qu'il affiche un *feedback* (*i.e.* appel `printf`) juste après la ligne 20. Le *thread* 0 reçoit le processeur et la condition de la boucle (*cfr.* ligne 14) est invalidée lui permettant d'entrer en section critique. Celui-ci la traverse et sort en basculant le `tour` à 1 (*cfr.* ligne 20). Imaginons que le *thread* 1 continue son exécution, affiche le *feedback* et revient au début de l'itération suivante de la boucle `for` (*cfr.* ligne 12). À ce stade, il arrive à l'entrée de la section critique (*cfr.* ligne 14) et doit attendre étant donné que le `tour` est de 0. Le *thread* 0 ne se trouve pas dans la section critique, ce qui correspond à une attente injustifiée du *thread* 1 à cause du *thread* 0.

L'**absence d'attente infinie** est garantie par cette solution. En effet, l'évolution cyclique de l'identifiant dans la variable `tour` garantit que chaque *thread* va à un moment reçevoir le droit d'entrer en section critique. Pour autant qu'un *thread* ne se termine pas prématurément et qu'il met à jour correctement la variable `tour` à la sortie de la section critique.

Alternance avec phase

Une troisième solution consiste à suivre la `phase` dans laquelle chaque *thread* se trouve ; `CRITIQUE` ou `AILLEURS`. Lorsque la `phase` d'un *thread* est `CRITIQUE`, cela signifie que le *thread* en question est soit dans la section critique soit à l'entrée de celle-ci. Lorsque sa `phase` est `AILLEURS`, il se trouve n'importe où dans son code excepté dans la section critique ou à l'entrée de celle-ci. Nous nous limitons au cas de deux *threads*.

L'implémentation de cette approche est présentée dans la Figure 4.5. Lorsqu'un *thread* arrive à l'entrée de la section critique, il bascule sa `phase` à `CRITIQUE` (*cfr.* ligne 19) et va attendre *tant que* la `phase` de l'autre *thread* est `CRITIQUE` (*cfr.* ligne 20). Si la `phase` de l'autre *thread* n'est pas `CRITIQUE`, le *thread* peut entrer en section critique (*cfr.* ligne 23). Lorsqu'un *thread* sort de la section critique, il bascule sa `phase` à `AILLEURS`.

```
11  unsigned int phase[NB_THREADS] = { AILLEURS, AILLEURS };
12  unsigned int total = 0;
13
14  // Correct seulement si NB_THREADS = 2
15  void* ajouter(int identifiant) {
16    unsigned int autre = 1 - identifiant;
17    for(int i = 0; i < 1000; i++) {
18      // Entrée en section critique
19      phase[identifiant] = CRITIQUE;
20      while(phase[autre] == CRITIQUE) {}
21
22      // Section critique
23      total = total + 1;
24
25      // Sortie de section critique
26      phase[identifiant] = AILLEURS;
27    }
28    pthread_exit(NULL);
```

FIGURE 4.5 – Solution avec `phase` pour l'incrémentation.

L'**exclusion mutuelle** est garantie par cette solution. Si un *thread* est en section critique, sa `phase` empêche n'importe quel autre *thread* d'y entrer également.

L'**absence d'attente injustifiée** est garantie par cette solution. En effet, la seule raison pour laquelle un *thread* peut être forcé à attendre à l'entrée de la section critique est qu'un autre *thread* se trouve déjà dedans.

L'**absence d'attente infinie** n'est pas garantie par cette solution. Si deux *threads* arrivent à l'entrée de la section, ils basculent leurs valeurs de `phase` respective à `CRITIQUE`. Cependant, les deux *threads* se retrouvent dès lors coincés mutuellement dans la boucle d'attente active en attendant que la phase de l'`autre` passe à `AILLEURS`. Cependant, la seule manière pour que cela se produise nécessiterait que l'un des *threads* traverse la section critique.

Solution de Peterson

Les solutions d'alternance (`phase`) et d'alternance stricte (`tour`) présentent une forme de complémentarité vis-à-vis des trois garanties qu'une solution de section critique doit offrir. En combinant les deux approches, les variables de `phase` pourrait être utilisée pour éviter les attentes injustifiées et la variable de `tour` pour départager les *threads* qui seraient en attente active à `CRITIQUE`. Nous nous limitons au cas de deux *threads*.

L'implémentation de cette approche est présentée à la Figure 4.6. Lorsqu'un *thread* arrive à l'entrée de la section critique, il va basculer sa `phase` à `CRITIQUE` et passer le `tour` à l'identifiant de l'`autre` *thread*. Une fois ces opérations conclues, le *thread* restera en attente active *tant que* le `phase` de l'`autre` est à `CRITIQUE` et que c'est le `tour` de l'`autre`. Si l'une de ces conditions est fausse (*i.e.* soit l'`autre` n'est pas à `CRITIQUE` ou que ce n'est pas son `tour`), le *thread* pourra entrer en section critique (*cfr.* ligne 18). Lorsqu'un *thread* sort de la section critique, il se contente de basculer sa `phase` à `AILLEURS`.

```
8  // Correct seulement si NB_THREADS = 2
9  void* ajouter(int identifiant) {
10   unsigned int autre = 1 - identifiant;
11   for(int i = 0; i < 1000; i++) {
12     // Entrée en section critique
13     phase[identifiant] = CRITIQUE;
14     tour = autre;
15     while(phase[autre] == CRITIQUE && tour == autre) {}
16
17     // Section critique
18     total = total + 1;
19
20     // Sortie de section critique
21     phase[identifiant] = AILLEURS;
22   }
23   pthread_exit(NULL);
24 }
```

FIGURE 4.6 – Solution avec `phase` et `tour` pour l'incrémentation.

L'**exclusion mutuelle** est garantie par cette approche. En effet, quelque soit l'*interleaving* des instructions des lignes 13 et 14, lorsque les deux *threads* sont dans l'attente active, la variable `tour` détermine lequel des deux peut entrer en section critique. L'**absence d'attente injustifiée** est garantie par cette approche. Si un seul *thread* arrive à l'entrée de la section critique il basculera le `tour` à l'identifiant de l'`autre` *thread*. Cependant, si cet `autre` *thread* est actuellement `AILLEURS`, le *thread* pourra entrer en section critique, la première des deux conditions de la boucle d'attente active étant fausse. L'**absence d'attente infinie** est garantie par cette approche. Cette garantie peut être démontrée par l'explication utilisée pour la garantie d'**exclusion mutuelle**.

Bien que la solution de Peterson soit *logiquement* correcte, celle-ci ne fonctionne pas si le processeur repose sur une architecture d'exécution *out-of-order*.

Instructions spéciales

Les solutions présentée jusqu'ici repose sur l'utilisation d'instructions standards d'un processeur et une gestion prudente de variables partagées pour synchroniser les accès en section critique. Cependant, les processeurs modernes offrent des facilités permettant d'implémenter des manipulations de variables partagées ou des variables d'accès en section critique de manière atomique.

Une solution consiste à désactiver les interruptions pour la durée de la section critique. En utilisant l'instruction de désactivation des interruptions à l'entrée et l'instruction d'activation des interruptions à la sortie, il est aisé de garantir que la section critique sera exécutée d'un seul tenant. Pour autant que la section critique ne déclenche pas d'exception, l'atomicité de son exécution sera garantie. Cependant, cette solution est risquée au niveau des processus et de *threads* car un oubli de ré-activer les interruptions dans le code du programme aurait pour conséquence le blocage complet de la machine. En effet, les interruptions du timer (*cfr. Round-Robin*), de frappes au clavier ne seraient plus traitées et le système ne pourrait donc plus réagir aux actions de l'utilisateur.

Une autre solution repose sur l'utilisation d'instructions spéciales de manipulations des registres et de la mémoire. En MIPS, l'instruction `tsl` (*cfr.* Figure 4.7a) permet en une seule instruction de charger le contenu depuis une adresse dans un registre et d'écrire une valeur de 1 à cette même adresse. En x86, l'instruction `xchg` (*cfr.* Figure 4.7b) permet d'échanger en une seule instruction le contenu d'un registre avec le contenu d'un emplacement en mémoire.

```
1 ENTREE: tsl reg, VERROU # reg := VERROU; VERROU := 1
2         cmp reg, 0
3         jne ENTREE
4 # Section
5 # Critique
6 SORTIE: mov VERROU, 0
```

(a) Solution par `verrou` avec l'instruction `tsl`.

```
1 ENTREE: mov  reg, 1
2         xchg reg, VERROU # swap reg <-> VERROU
3         cmp  reg, 0
4         jne ENTREE
5 # Section Critique
6 SORTIE: mov VERROU, 0
```

(b) Solution par `verrou` avec l'instruction `xchg`.

Il est possible de corriger l'implémentation de la solution par `verrou` en utilisant une instruction spéciale du processeur. L'implémentation corrigée est présentée à la Figure 4.8. La seule différence se situe au niveau de l'entrée en section critique pour laquelle une variable intermédiaire (*i.e.* `etat`) est utilisée pour stocker la valeur du `verrou` avant l'échange. La syntaxe de la ligne 18 (*i.e.* `__asm__`) est utilisée pour coller une instruction assembleur ponctuelle dans le code en spécifiant que le registre utilisé pour la variable `etat` et l'adresse où réside la variable `verrou` doivent être utilisée dans l'encodage de cette instruction. De la sorte, le corps de la boucle (*cfr.* ligne 18) échange le contenu de la mémoire avec le contenu du registre et le *thread* reste en attente active *tant que* la valeur de la mémoire *avant* cet échange (*cfr.* ligne 19) était `FERMÉ`. Dès que la variable `verrou` en mémoire est basculée à `OUVERT` (par la sortie d'un autre *thread*), le *thread* peut entrer en section critique.

```
8  #define OUVERT 0
9  #define FERMÉ  1
10
11 unsigned int verrou = OUVERT;
12 unsigned int total = 0;
13
14 void* ajouter(int identifiant) {
15   for(int i = 0; i < 1000; i++) {
16     // Entrée en section critique
17     unsigned int etat = FERMÉ;
18     do { __asm__("xchg %0, %1" : "+q" (etat), "+m" (verrou)); }
19     while (etat == FERMÉ);
20
21     // Section critique
22     total = total + 1;
23
24     // Sortie de section critique
25     verrou = OUVERT;
26   }
27   pthread_exit(NULL);
28 }
```

FIGURE 4.8 – Solution avec `verrou` et instruction `xchg`.

4.2.2 Solutions avec blocage

La seconde catégorie de solutions repose sur l'utilisation d'un **blocage** à l'entrée de la section critique lorsqu'il n'y a plus de place et un **déblocage** lorsqu'une place se libère par la sortie d'un autre *thread*. Outre la possibilité d'implémenter la gestion d'une section critique, nous allons également nous pencher sur les possibilité de gérer des problèmes de synchronisation entre *threads*.

Mutex

La notion de *mutex*, qui est similaire au concept de verrou de la catégorie précédente, permet de mettre en place une section critique autour d'un objet partagé. En pratique, derrière un *mutex* est dissimulé une variable entière `m` qui peut prendre deux valeurs possibles ; $0 \leqslant m \leqslant 1$. La valeur `0` peut être interprétée comme représentant l'état `FERMÉ` tandis que la valeur `1` représente l'état `OUVERT`. De plus, une liste d'attente des *threads* qui sont `BLOQUÉS` en attente est également adjointe au *mutex*. Pour pouvoir utiliser le *mutex*, deux opérations atomiques sont disponibles ; `lock(m)` et `unlock(m)`.

Lorsqu'un *thread* arrive à l'entrée de la section critique, il doit utiliser la fonction `lock(m)` pour tenter d'obtenir l'accès exclusif. Si la valeur de `m` est `0` (*n.b.* `FERMÉ`) au moment de l'appel, le *thread* est placé dans l'état `BLOQUÉ` et est ajouté à la liste d'attente du *mutex*. Si la valeur de `m` est `1` (*n.b.* `OUVERT`) au moment de l'appel, la valeur de `m` est passée à `0` (*n.b.* `FERMÉ`) et l'appel se termine sans **blocage**. Le *thread* appelant continue son exécution, ce qui lui permet d'avancer en section critique.

Lorsqu'un *thread* arrive à la sortie de la section critique, il doit utiliser la fonction `unlock(m)` pour relacher l'accès exclusif qu'il détient. Si la liste d'attente est vide, la valeur de `m` est passée à `1` (*n.b.* `OUVERT`) pour restituer l'accès exclusif. Si la liste d'attente n'est pas vide, un *thread* est sélectionné et subit un **déblocage** (*i.e.* transition à l'état `PRÊT`) ce qui lui permettra de recevoir le processeur et d'avancer en section critique.

Sémaphore

Il est possible de relaxer la contrainte de la borne supérieure du *mutex* pour définir le concept de sémaphore. L'utilité d'un tel objet réside dans la possibilité de synchroniser des *threads* au lieu de simplement les exclure

mutuellement d'une section critique. En pratique, derrière un sémaphore est dissimulée une variable entière `s` qui peut prendre n'importe quelle valeur positive ou nulle ; $0 \leqslant$ `s`. Le sémaphore peut être perçu comme un compteur qui suit à la trace le nombre de jetons d'accès disponibles pour l'objet partagé qu'il protège. Une liste des *threads* qui sont **`BLOQUÉS`** en attente d'une place est adjointe au sémaphore. Pour pouvoir l'utiliser, deux opérations atomiques sont disponibles ; `proberen(s)` et `verhogen(s)`. Selon les implémentations, la fonction `proberen(s)` est parfois aussi appelée `down(s)` ou `wait(s)` et la fonction `verhogen(s)` est parfois appelée `up(s)` ou `signal(s)`.

Lorsqu'un *thread* arrive à l'entrée de la région protégée par le sémaphore, il doit utiliser la fonction `proberen(s)` afin de tenter d'obtenir un jeton d'accès. Si la valeur de `s` est `0` au moment de l'appel, le *thread* appelant est placé dans l'état **`BLOQUÉ`** et est ajouté à la liste d'attente du sémaphore. Si la valeur de `s` est strictement supérieure à `0` au moment de l'appel, la valeur est décrémentée d'une unité et l'appel se termine sans **blocage**. Le *thread* appelant peu continuer son exécution, ce qui lui permet d'avancer dans la région de code manipulant l'objet partagé.

Lorsqu'un *thread* arrive à la sortie de la région protégée, il doit utiliser la fonction `verhogen(s)` pour restituer son jeton d'accès. Si la liste d'attente est vide, la valeur de `s` est incrémentée d'une unité pour refléter le fait que le jeton est replacé sur le *tas* que le sémaphore représente. Si la liste d'attente n'est pas vide, un *thread* est sélection et subit un **déblocage** (*i.e.* transition à l'état **`PRÊT`**) qui lui permettra de recevoir le processeur et d'avancer en section critique.

4.3 Problèmes classiques

Afin d'illustrer l'utilisation des solutions de section critique, nous allons considérer deux problèmes classiques ; le problème du producteur-consommateurs et le dîner des philosophes. Bien que nous aborderons ces problèmes sous un angle plus théorique, ceux-ci représentent un canevas qui revient régulièrement dans l'implémentation d'application nécessitant des accès concurrents et de la synchronisation entre *threads*.

4.3.1 Producteur-consommateur

Dans la version qui va nous intéresser, le problème du producteur-consommateur fait intervenir deux *threads* ; un producteur qui fabrique des unités d'un produit continuellement et un consommateur qui obtient ces produits et les consomme continuellement. Les cadences de production et de consommation sont imprévisibles ; la production et la consommation d'une unité sont impossible à prévoir et sujette à des variations d'un moment à un autre. Afin de pouvoir échanger les unités du produit, un *buffer* partagé est implémenté sous la forme d'une structure de données non-spécifiée. Cet espace permet également d'accumuler les unités en attendant que le consommateur en ait besoin. Lorsque l'espace est vide (*i.e.* aucune unité disponible), le consommateur doit attendre jusqu'à ce que le producteur produise une unité additionnelle. La Figure 4.9 présente le canevas de base qui met en place les deux *threads*.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include "produit.h"
5
6  produit* buffer;
7
8  void* production(void* inutile) { ... }
9  void* consommation(void* inutile) { ... }
10
11 int main(int argc, char* argv[]) {
12   pthread_t producteur, consommateur;
13
14   pthread_create(&producteur, NULL, production, NULL);
15   pthread_create(&consommateur, NULL, consommation, NULL);
16
17   pthread_join(producteur, NULL);
18   pthread_join(consommateur, NULL);
19
20   return EXIT_SUCCESS;
21 }
```

FIGURE 4.9 – Canevas de base

Le *thread* principale du programme va engendrer les deux *threads* (*i.e.* producteur et consommateur, *cfr.* lignes $14-15$) avant de se mettre en attente de leur terminaison (*cfr.* lignes $17-18$). En pratique, la production et la consommation étant continuelle, ces deux *threads* ne se termineront jamais.

Cas du buffer non-borné

Nous allons d'abord nous pencher sur le cas d'un espace infini pour le *buffer*. Nous ne devrons pas nous pré-occuper d'une condition de *buffer* plein ou de mémoire indisponible.

Dans notre problème, le *buffer* constitue un objet partagé sur lequel les deux *threads* sont susceptibles de réaliser des accès concurrents. En effet, quelque soit l'implémentation de la structure de données sous-jacente, les opérations d'insertion et d'extraction vont tant devoir lire que modifier cette structure de données. De ce fait, il est important de protéger le *buffer* contre les accès concurrents à l'aide d'un *mutex*.

D'autre part, lorsque le *buffer* est vide, la consommation doit bloquer jusqu'à ce que le producteur insère une unité dans celui-ci. Si nous utilisions directement une condition sur la longueur du *buffer*, nous serions forcé de mettre en place une boucle d'attente active (*i.e. tant que* le *buffer* est vide). Il est possible d'intégrer cette notion d'attente en utilisant un sémaphore qui capture le nombre d'unités disponibles dans le *buffer*. L'implémentation résultante pour la production et la consommation est donnée à la Figure 4.10.

```
7  #include "mutex.h"     // lock(..), unlock(..)
8  #include "semaphore.h" // proberen(..), verhogen(..)
9
10 mutex acces = OUVERT;      // Accès au buffer
11 semaphore disponibles = 0; // Nombre de produits disponibles
```

```
13 void* production(void* inutile) {
14   while(1) {
15     produit nouveau = produire();
16     lock(acces);
17     inserer(buffer, nouveau);
18     unlock(acces);
19     verhogen(disponibles);
20   }
21 }
```

```
23 void* consommation(void* inutile) {
24   while(1) {
25     proberen(disponibles);
26     lock(acces);
27     produit suivant = extraire(buffer);
28     unlock(acces);
29     consommer(suivant);
30   }
31 }
```

FIGURE 4.10 – Implémentation pour un *buffer* de capacité infinie.

Nous supposons disposer d'une implémentation de *mutex* et de sémaphore (*cfr.* lignes $7-8$). Initialement, le *mutex* protégeant l'accès au *buffer* est `OUVERT` et le nombre de produits `disponibles` est de `0`.

Du côté de la production, une boucle infinie capture le caractère *continuel* (*cfr.* ligne 14) qui consiste à produire une nouvelle unité (*cfr.* ligne 15), insérer cette nouvelle unité dans le *buffer* sous la protection du *mutex* (*cfr.* lignes $16-18$) et enfin incrémenter le nombre de produits `disponibles`.

Du côté de la consommation, une boucle infinie capture le caractère *continuel* (*cfr.* ligne 24) qui consiste à attendre qu'il y'ait au moins une unité dans le *buffer* (*cfr.* ligne 25), extraire une unité sous la protection du *mutex* (*cfr.* ligne $26-28$) et enfin consommer l'unité en question (*cfr.* ligne 29). Quelque soient les cadences respectives de production et de consommation (*i.e.* temps pour `produire()` et `consommer(..)`, le sémaphore permettra de synchroniser les deux *threads* pour gérer le cas du *buffer* vide et le *mutex* se limitera aux opérations qui touchent au *buffer*.

Cas du buffer borné

Nous allons maintenant nous pencher sur le cas d'un espace de taille finie disponible pour le *buffer*. De ce fait, nous devrons gérer la condition du *buffer* plein.

Dans cette variante du problème, le *buffer* constitue toujours un objet partagé sur lequel des accès concurrents peut être réalisés. La protection à l'aide d'un *mutex* est toujours requise. D'autre part, le cas du *buffer* vide est toujours présent et la consommation est bloquée tant que le *buffer* est vide lorsque le consommateur a besoin d'un produit.

La nouvelle contrainte qui est introduite est que lorsque le *buffer* est plein, le producteur devrait bloquer. En constatant que cette contrainte ressemble fortement à celle imposée sur le consommateur dans le cas du *buffer* vide, il est facile de la représenter à l'aide d'un autre sémaphore qui représente le nombre d'`emplacements` libres dans le *buffer* pour recevoir des unités produites. L'implémentation adaptée pour la production et la consommation est donnée à la Figure 4.11.

```
7  #include "mutex.h"     // lock(..), unlock(..)
8  #include "semaphore.h" // proberen(..), verhogen(..)
9
10 mutex acces = OUVERT;         // Accès au buffer
11 semaphore disponibles = 0;    // Nombre de produits disponibles
12 sempahore emplacements = 10; // Nombre d'emplacements libres
```

```
14 void* production(void* inutile) {
15   while(1) {
16     produit nouveau = produire();
17     proberen(emplacements);
18     lock(acces);
19     inserer(buffer, nouveau);
20     unlock(acces);
21     verhogen(disponibles);
22   }
23 }
```

```
25 void* consommation(void* inutile) {
26   while(1) {
27     proberen(disponibles);
28     lock(acces);
29     produit suivant = extraire(buffer);
30     unlock(acces);
31     verhogen(emplacements);
32     consommer(suivant);
33   }
34 }
```

FIGURE 4.11 – Implémentation pour un *buffer* de capacité bornée.

Nous supposons, comme précedemment, disposer d'une implémentation de *mutex* et de sémaphore (*cfr.* lignes $7-8$). Le *mutex* d'accès au *buffer* est initialement `OUVERT`, le nombre d'unités `disponibles` est de `0` et le nombre d'`emplacements` libres est de `10`. La valeur initiale utilisée pour ce second sémaphore va déterminer la capacité du *buffer* car il représentera le nombre maximal d'unités qui pourront être stockées dans le *buffer* à n'importe quel moment.

Du côté de la production, l'insertion de la nouvelle unité produite dans le *buffer* est désormais précédée d'une vérification qu'il y a au moins une place dans le *buffer* (*cfr.* ligne 17). Le reste de la logique du producteur est inchangée.

Du côté de la consommation, l'extraction d'une unité du *buffer* est désormais suivie d'une incrémentation du nombre d'`emplacements` libres dans le *buffer* (*cfr.* ligne 31).

4.3.2 Dîner des philosophes

Le dîner des philosophes va nous permettre d'étudier un problème de synchronisation particulier ; lorsque les ressources communes doivent être réparties entre plusieurs *threads*. Ce problème va nous permettre de mettre en évidence une problématique à laquelle il est possible de se heurter ; le *deadlock*, où plusieurs *threads* indépendants attendent mutuellement qu'un autre parmi eux relâche une de ces ressources pour réaliser ses traitements.

Dans ce contexte, cinq philosophes sont assis à une table sur laquelle se trouve cinq baguettes et un plat au centre. Un philosophe passe son temps à alterner entre *penser* et *manger*. Afin de pouvoir *manger* un philosophe doit détenir deux baguettes ; celle à sa *droite* et celle à sa *gauche*. Cette contrainte à pour conséquence que deux voisins ne peuvent pas *manger* en même temps, étant donné qu'ils ont une baguette en commun qui ne peut être détenue que par l'un ou par l'autre.

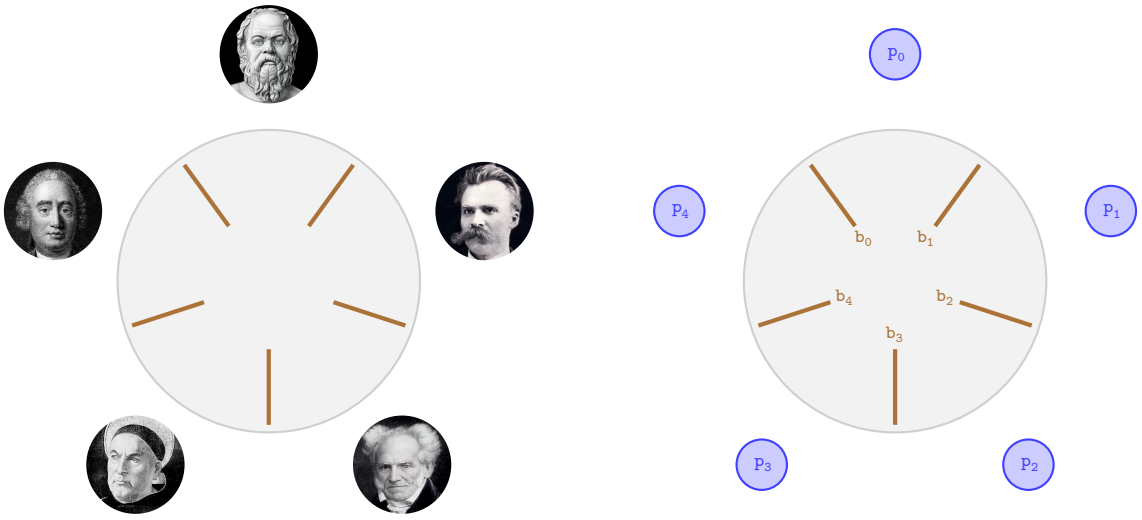


FIGURE 4.12 – Agencement de la table du dîner des philosophes et numérotation des *threads* et des baguettes.

Le canevas de base que nous utiliserons pour nos tentatives de solutions successives est donné dans la Figure 4.14. Nous supposons qu'il existe quelque part des fonctions `penser()` et `manger()` qui peuvent être utilisées dans la fonction `philosopher()` que les *threads* exécutent. Le *thread* principal engendre les cinq *threads* philosophes (*cfr.* lignes 13 − 14) avant de se mettre en attente de leur terminaison (*cfr.* lignes 17 − 18). Nous définissons également deux macros symboliques aux lignes 6 − 7 qui permettent de dériver l'identifiant des baguettes à la droite et à la gauche du philosophe $\texttt{p}_\texttt{i}$. L'implémentation sera concentrée derrière la fonction `philosopher()` où nous intégrerons les tentatives successives de résolution.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include "philosophe.h" // penser(), manger()
5
6  #define DROITE(i) ((  i  ) % NB_PHILOSOPHES)
7  #define GAUCHE(i) ((i + 1) % NB_PHILOSOPHES)
8
9  #define NB_PHILOSOPHES 5
10 pthread_t philosophes[NB_PHILOSOPHES];
11
12 void* philosopher(int moi) { ... }
13
14 int main(int argc, char* argv[]) {
15   // Création des threads philosophes
16   for(unsigned int p = 0; p < NB_PHILOSOPHES; p++)
17     pthread_create(&philosophes[p], NULL, philosopher, p);
18
19   // Attente des threads philosophes
20   for(unsigned int p = 0; p < NB_PHILOSOPHES; p++)
21     pthread_join(philosophes[p], NULL);
22
23   return EXIT_SUCCESS;
24 }
```

FIGURE 4.13 – Canevas de base pour le dîner des philosophes.

Première version

En guise de première tentative, nous allons utiliser un simple tableau pour représenter l'état des `baguettes`. Une `baguette` peut être soit `LIBRE` lorsqu'elle est sur la table, soit `PRISE` lorsqu'elle est détenue par un philosophe.

```
14 #define LIBRE 0
15 #define PRISE 1
16 int baguettes[NB_PHILOSOPHES];
```

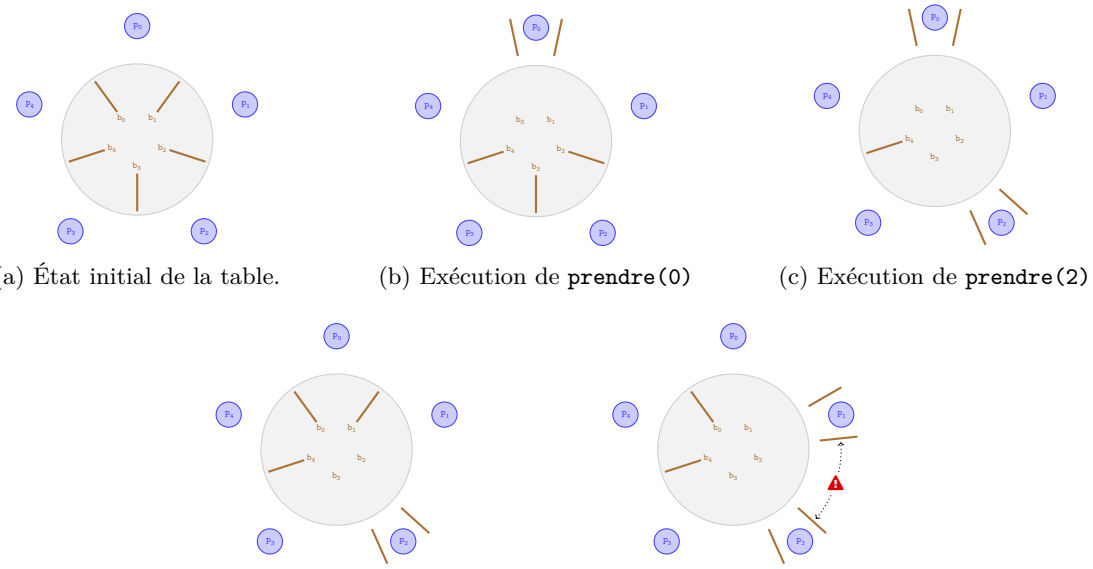
```
18 void prendre(int moi) {
19   baguettes[DROITE(moi)] = PRISE;
20   baguettes[GAUCHE(moi)] = PRISE;
21 }
```

```
23 void deposer(int moi) {
24   baguettes[DROITE(moi)] = LIBRE;
25   baguettes[GAUCHE(moi)] = LIBRE;
26 }
```

```
28 void* philosopher(int moi) {
29   while(1) {
30     penser();
31     prendre(moi);
32     manger();
33     deposer(moi);
34   }
35 }
```

FIGURE 4.14 – Représentation des baguettes à l'aide d'un tableau.

Le comportement continuel d'un philosophe est capturé dans une boucle infinie (*cfr.* lignes 29−34) qui consiste à commencer par `penser()` (*cfr.* ligne 30) pendant un certain temps qui est déterminé par l'implémentation de cette fonction. Le philosophe entre ensuite dans la période où il va manger ; il prend les baguettes qui se trouvent de part et d'autre de sa place (*cfr.* ligne 31) et commence à `manger()` (*cfr.* ligne 32). Une fois qu'il a fini de `manger()`, le philosophe doit `déposer()` ses baguettes avant de revenir au début du corps de la boucle infinie. La prise des `baguettes` consiste à simplement marquer leur état comme étant `PRISE` et leur dépôt consiste à les marquer comme étant `LIBRE`.



(a) État initial de la table. (b) Exécution de `prendre(0)` (c) Exécution de `prendre(2)`

(d) Exécution de `deposer(0)` (e) Exécution de `prendre(1)`

FIGURE 4.15 – Évolution de l'exécution de la solution 1.

Cette première implémentation présente un problème assez évident ; l'absence de vérification qu'une baguette est déjà prise pour garantir un accès exclusif dessus. Initialement, toutes les baguettes sont posées sur la table et les cinq philosophes sont en train de `penser()` (*cfr.* Figure 4.15a). Supposons que le philosophe p_0 termine de `penser()` et prend les baguettes qui le concerne (*cfr.* Figure 4.15b) et commence à `manger()`. À ce stade, le philosophe p_2 termine également de `penser()` et prend ses baguettes (*cfr.* Figure 4.15c) et commence lui-même à `manger()`. Supposons que le philosophe p_0 termine de `manger()` et dépose ses baguettes avant de recommencer à `penser()` (*cfr.* Figure 4.15d). À ce stade, si le philosophe p_1 termine de `penser()` il pourra prendre ses baguettes et commencer à `manger()` (*cfr.* Figure 4.15e), introduisant une situation où deux voisins (philosophes p_1 et p_2) sont en train de `manger()` en même temps.

Deuxième version

Afin d'éviter que deux voisins ne mangent en même temps, nous pouvons introduire un *mutex* sur la `table` pour s'assurer qu'un seul philosophe puissent prendre ses baguettes. Lorsqu'un philosophe terminera de `penser()`, il devra obtenir le *mutex* sur la table pour pouvoir prendre ses baguettes.

```
14 #define LIBRE 0
15 #define PRISE 1
16 int baguettes[NB_PHILOSOPHES];
17 mutex table = OUVERT;
```

```
19 void prendre(int moi) {
20   baguettes[DROITE(moi)] = PRISE;
21   baguettes[GAUCHE(moi)] = PRISE;
22 }
```

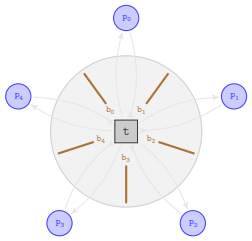
```
24 void deposer(int moi) {
25   baguettes[DROITE(moi)] = LIBRE;
26   baguettes[GAUCHE(moi)] = LIBRE;
27 }
```

```
29 void* philosopher(int moi) {
30   while(1) {
31     penser();
32     lock(table);
33     prendre(moi);
34     manger();
35     deposer(moi);
36     unlock(table);
37   }
38 }
```

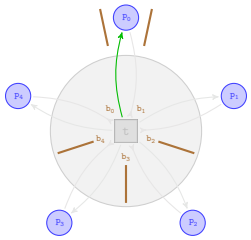
FIGURE 4.16 – Utilisation d'un *mutex* global sur la table.

Pour permettre à cette solution de fonctionner, le *mutex* est initialement `OUVERT` (*cfr.* ligne 17). Le comportement continuel du philosophe est toujours capturé par une boucle infinie (*cfr.* lignes 30 – 37) où la phase de `manger()` (incluant la prise et le dépôt des baguettes) est protégée par le *mutex* de `table`.

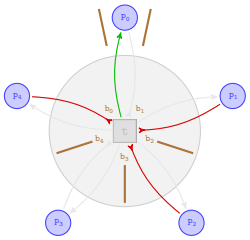
Cette implémentation résout le problème des deux voisins qui mangent en même temps. La Figure 4.17 présente un canevas d'exécution de la solution avec un *mutex* sur la `table`. Initialement, le *mutex* de `table` est `OUVERT` et tous les philosophes sont en train de `manger()`. Imaginons que le philosophe p_0 termine de `penser()` et obtient le *mutex* (qui devient `FERMÉ`) et ses baguettes avant de commencer à `manger()` (*cfr.* Figure 4.17b). À ce stade, imaginons que les trois philosophes p_2, p_1 et p_4 émergent de leurs pensées et tentent d'acquérir le *mutex* de la `table`. Celui-ci étant `FERMÉ`, ces trois philosophes se retrouvent `BLOQUÉS` en attente de ce *mutex* (*cfr.* Figure 4.17c).



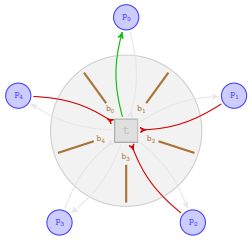
(a) État initial de la table



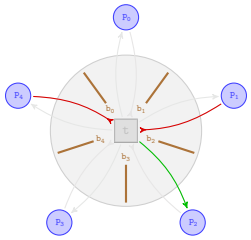
(b) Acquisition des baguettes par p_0



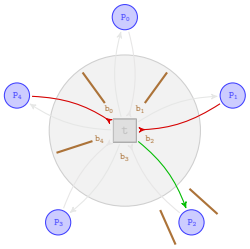
(c) Attente par p_1, p_2 et p_4



(d) Dépôt des baguettes par p_0



(e) Relâchement du *mutex*



(f) Acquisition des baguettes par p_2

FIGURE 4.17 – Évolution de l'exécution de la solution 2.

Une fois que le philosophe p_0 a terminé de `manger()`, il effectuera le dépôt des baguettes (*cfr.* Figure 4.17d). Ce dépôt est suivi du déverrouillage du *mutex* de la `table` (*cfr.* Figure 4.17e) ce qui a comme conséquence le **déblocage** d'un des philosophes ; *e.g.* philosophe p_2 qui est le premier à avoir bloqué sur le *mutex*. Ceci permet au philosophe p_2 de `prendre()` ses baguettes et commencer à `manger()` (*cfr.* Figure 4.17f). Dans cette configuration, le philosophe p_1 (qui souhaite manger) est empêché de le faire en raison du *mutex*. Cependant, cette approche introduit un autre problème ; un non-voisin (*i.e.* philosophe p_4) n'est pas autorisé à `manger()` alors qu'il n'a aucun baguette en commun avec celui qui est en train de `manger()` (*i.e.* philosophe p_2).

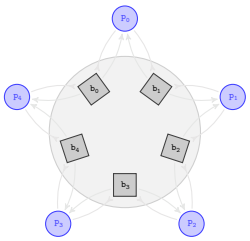
Troisième solution

Pour permettre à deux non-voisins de `manger()` en même temps, une approche consisterait à augmenter la granularité en séparant de *mutex* global en plusieurs *mutex* ; un par baguette et de se reposer sur le **blocage/déblocage**. Le tableau de `baguettes` est modifié pour que chaque case corresponde au *mutex* qui les représente. De plus, les fonctions `prendre()` et `deposer()` sont adaptées pour utiliser les opérations `lock()` et `unlock()` sur les *mutex* pertinents.

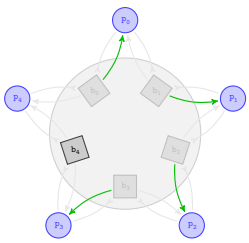
```
14  mutex baguettes[NB_PHILOSOPHES];

16  void prendre(int moi) {
17    lock(baguettes[DROITE(moi)]);
18    lock(baguettes[GAUCHE(moi)]);
19  }

21  void deposer(int moi) {
22    unlock(baguettes[DROITE(moi)]);
23    unlock(baguettes[GAUCHE(moi)]);
24  }

26  void* philosopher(int moi) {
27    while(1) {
28      penser();
29      prendre(moi);
30      manger();
31      deposer(moi);
32    }
33  }
```

FIGURE 4.18 – Utilisation d'un *mutex* par baguette.

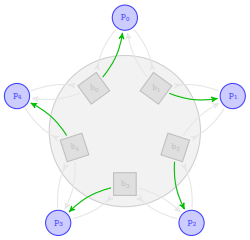
Les *mutex* associés aux `baguettes` sont initialement tous à l'état `OUVERT`. Le comportement continuel est toujours le même (*cfr.* lignes $27-32$) et représenté par une boucle infinie mais le *mutex* global n'est plus utilisé. La prise des `baguettes` consiste à verrouiller les *mutex* correspondant aux deux baguettes qui concernent le philosophe tandis que le dépôt des `baguettes` consiste à déverrouiller ces *mutex*.



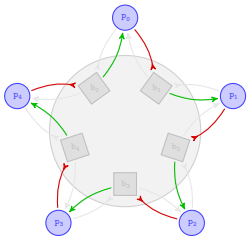
(a) État initial de la table. Cinqs baguettes disponibles.



(b) Acquisition des baguettes de *droite* par $\mathtt{p}_0$, $\mathtt{p}_1$, $\mathtt{p}_2$, $\mathtt{p}_3$



(c) Acquisition de la baguette de *droite* par $\mathtt{p}_4$



(d) Attente des baguettes de *gauche*. Manifestation du *deadlock*.

FIGURE 4.19 – Évolution de l'exécution de la solution 3.

Cette implémentation garantit que deux voisins ne peuvent pas `manger()` en même temps et que deux non-voisins peuvent `manger()` en même temps. Cependant, elle introduit la possibilité d'apparition d'un *deadlock*, illustré à la Figure 4.19.

Au départ de l'état où les cinq baguettes sont déposées sur la table (*cfr.* Figure 4.19a), imaginons que les quatres premiers philosophe terminent de `penser()` et acquiert leurs baguettes de droite (*cfr.* Figure 4.19b). À ce stade, si le dernier philosophe ($\mathtt{p}_4$) en fait de même (*cfr.* Figure 4.19c), le *deadlock* devient inévitable étant donné que lorsque tous les philosophes tenteront d'obtenir leurs baguettes de gauche, ils bloqueront en attente (*cfr.* Figure 4.19d). Dans cet état, le *deadlock* est devenu manifeste et tous les *threads* impliqués sont bloqués mutuellement en attente qu'un autre *thread* relâche une de ses baguettes.

Résolution de *deadlock*

La problématique des *deadlocks* a déjà été étudiée depuis longtemps et les quatres conditions suivantes ont été identifiées comme étant nécessaires pour qu'un *deadlock* survienne.

- $\mathcal{C}_1$ Exclusion mutuelle sur les ressources manipulées par les *threads*. Dans une implémentation exempte de section critique, il est en impossible que les *threads* impliqués soient bloqués en attente d'une ressource.
- $\mathcal{C}_2$ Détention de $\mathcal{R}_1$ par $\mathcal{T}$ et attente de $\mathcal{R}_2$. Pour observer une situation de *deadlock* il est nécessaire qu'un *thread* détienne un accès exclusif sur une certaine ressource et soit bloqué en attente d'un accès exclusif
- $\mathcal{C}_3$ Absence de préemption. Une fois qu'un accès exclusif est accordé à un *thread* il est impossible de lui retirer cet accès exclusif ; seul le *thread* en question peut déverrouiller le *mutex* une fois qu'il n'en a plus besoin
- $\mathcal{C}_4$ Cycle de détention et d'attente. Une fois qu'un ensemble de *threads* et de *ressources* sont enchaînées par la relation de détention/attente de la condition $\mathcal{C}_2$, le *deadlock* est manifeste et les *threads* sont tous bloqués

FIGURE 4.20 – Représentation des conditions $\mathcal{C}_2$ (maillons indivuels) et cycle complet $\mathcal{C}_4$.

Afin de résoudre un problème de *deadlock*, plusieurs stratégies sont applicables que nous allons développer brièvement.

La première approche prend la forme de l'algorithme de l'autruche [1] qui consiste simplement à ignorer le *deadlock*. Cette approche peut sembler consternante mais prend son sens si le dommage de sa survenance est inférieure au coût à déployer pour trouver son origine et le résoudre sans affecter le bon fonctionnement de l'application où il apparaît. Si un serveur d'application présente un *deadlock* une fois tous les cinqs ans et qu'en redémarrant le système aucune opération ou transaction n'est perdue et rend simplement le système indisponible pendant cinq minutes, mettre sa tête dans le sable est tout à fait approprié.

La deuxième approche (détection et résolution) consiste à faire le suivi de l'état d'allocation des différentes ressources entre les *threads* afin de détecter lorsque les *threads* sont bloqués dans un *deadlock*. La représentation graphique utilisée à la Figure 4.19 peut être implémentée pour faire le suivi de l'état d'allocation ; quelles ressources sont détenues/attendues par quels *threads*. À chaque allocation, le système peut vérifier si un cycle de détention/attente existe dans la représentation et appliquer une procédure de résolution. En pratique, pour pouvoir résoudre le *deadlock*, il est nécessaire de choisir un *thread* qui va perdre une ressource qui sera donnée à un *autre* afin de briser le cycle. Cependant, si un *thread* perd l'accès exclusif sur une ressource, il est nécessaire de faire marche arrière dans tous les traitements que ce *thread* à réalisé sur la ressource depuis qu'il détient l'accès exclusif.

Dans l'implémentation basique d'un *mutex* ou sémaphore, la seule condition qui détermine la traversée ou le **blocage** repose sur la valeur actuelle du *mutex* ou du sémaphore. La troisième approche (stratégie d'évitement, *e.g.* Algorithme du Banquier) consiste à modifier cette condition pour inclure l'état d'allocation actuel dans cette décision. Si accorder l'accès exclusif sur une ressource amène l'état d'allocation dans une situation *dangereuse*, le *thread* demandeur devrait être `BLOQUÉ`. Par exemple, dans la Figure 4.19b, si le philosophe $\mathtt{p}_4$ tente d'obtenir la baguette $\mathtt{b}_4$ et qu'il la reçoit, le système tombe dans un état *dangereux* où le *deadlock* est inévitable. En revanche, si le système choisissait dans cette situation de bloquer le philosophe $\mathtt{p}_4$, la baguette resterait disponible pour permettre au philosophe $\mathtt{p}_3$ de l'obtenir quand il la demandera. Le *thread* `BLOQUÉ` sera débloqué lorsqu'il ne sera pas dangereux de le faire.

La quatrième approche (prévention structurelle) consiste à invalider une des quatres conditions $\mathcal{C}_1$-$\mathcal{C}_4$ pour éliminer la possibilité qu'un *deadlock* apparaisse. La condition $\mathcal{C}_1$ ne peut pas systématiquement être invalidée car elle dépend du problème de base qui est résolu ; il faut qu'il existe une solution qui ne nécessite pas d'exclusion mutuelle et de sections critiques. La condition $\mathcal{C}_3$ peut être invalidée plus facilement pour autant que l'acquisition des différentes ressources soit regroupée en une séquence de code dans les *threads* qui précède n'importe quel traitement touchant à ces ressources. Dans les deux sections restantes, nous allons étudier les solutions qui consistent à invalider les deux conditions restantes $\mathcal{C}_4$ et $\mathcal{C}_2$.

Quatrième solution — Invalidation de $\mathcal{C}_4$

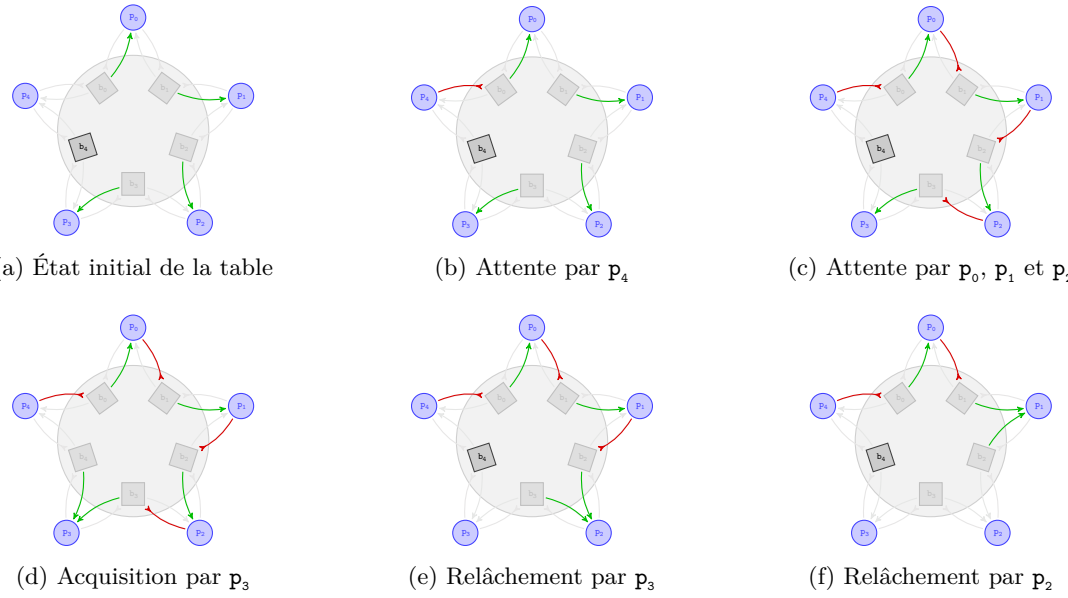
La quatrième solution consiste à empêcher la formation d'un cycle de détention/attente en modifiant l'ordre de prise des baguettes. Un *mutex* est toujours utilisé pour chaque baguette et l'ordre dans la prise des baguettes est inversé pour l'un des philosophes (*i.e.* $\mathtt{p}_4$) qui prendre sa baguette de gauche en premier et ensuite sa baguette droite.

```
14  mutex baguettes[NB_PHILOSOPHES];

16  void prendre(int moi) {
17    lock(baguettes[min(DROITE(moi), GAUCHE(moi))]);
18    lock(baguettes[max(DROITE(moi), GAUCHE(moi))]);
19  }

21  void deposer(int moi) {
22    unlock(baguettes[DROITE(moi)]);
23    unlock(baguettes[GAUCHE(moi)]);
24  }

26  void* philosopher(int moi) {
27    while(1) {
28      penser();
29      prendre(moi);
30      manger();
31      deposer(moi);
32    }
33  }
```

FIGURE 4.21 – Inversion de l'ordre d'acquisition des baguettes

Dans cette implémentation, l'ordre d'acquisition des baguettes est contrôlé en se reposant sur l'identifiant des différentes baguettes ; les philosophes acquiert leurs baguettes en prenant celle portant le plus petit identifiant en premier suivie de celle portant le plus grand identifiant.

1. Algorithme ô-combien puissant qui peut être appliqué à tout problème ☺



(a) État initial de la table (b) Attente par p_4 (c) Attente par p_0, p_1 et p_2

(d) Acquisition par p_3 (e) Relâchement par p_3 (f) Relâchement par p_2

FIGURE 4.22 – Évolution de l'exécution de la solution 4.

Au départ de l'état dangereux de la Figure 4.19b repris à la Figure 4.22a, les quatre premiers philosophes ont déjà obtenus leur baguette portant le plus petit identifiant. Lorsque le philosophe p_4 tente d'obtenir sa baguette ayant le plus petit identifiant (*i.e.* à sa gauche), celui-ci bloquera étant donné qu'elle est détenue par son voisin de gauche (*cfr.* Figure 4.22b). Supposons que les trois premiers philosophes obtiennent leurs baguettes de plus grand identifiant et bloquent tous en raison de la détention de ces baguettes par leurs voisins de gauche (*cfr.* Figure 4.22c). Le philosophe p_3 est libre de prendre sa baguette ayant l'identifiant le plus grand (à sa gauche) car celle-ci est disponible, lui permettant de commencer à `manger()` (*cfr.* Figure 4.22d). Une fois que le philosophe p_3 a terminé de `manger()`, il peut déposer ses deux baguettes, débloquant au passage le philosophe p_2 lui permettant de commencer à `manger()` (*cfr.* Figure 4.22e). À son tour, lorsque le philosophe p_2 dépose ses baguettes, il débloque le philosophe p_1 qui peut commencer à `manger()` (*cfr.* Figure 4.22f).

Cette solution élimine le *deadlock* mais ré-introduit la restriction d'empêcher deux non-voisins de `manger()` en même temps. En effet, à la Figure 4.22d, le philosophe p_3 qui est le seul en train de `manger()` ne devrait pas empêcher le philosophe p_1 de `manger()`. Ce problème apparaît occasionnellement contrairement à la solution du *mutex* global où le problème était systématique.

Cinquième solution — Invalidation de $\mathcal{C}_2$

La cinquième solution consiste à garder la granularité de l'accès exclusif offerte par l'utilisation de cinq *mutex* mais à changer leur signification. Au lieu de représenter une baguette individuelle, chaque *mutex* représente le droit de manger d'un philosophe. De plus, dans cette approche, l'état d'un philosophe est suivi à la trace pour pouvoir établir s'il est approprié de donner le droit de manger à un philosophe qui a faim.

```
27 void prendre(int moi) {
28   lock(table);
29   etats[moi] = AFAIM;
30   tenter(moi);
31   unlock(table);
32   lock(peutManger[moi]);
33 }
```

```
35 void deposer(int moi) {
36   lock(table);
37   etats[moi] = PENSE;
38   tenter(GAUCHE(moi));
39   tenter(DROITE(moi));
40   unlock(table);
41 }
```

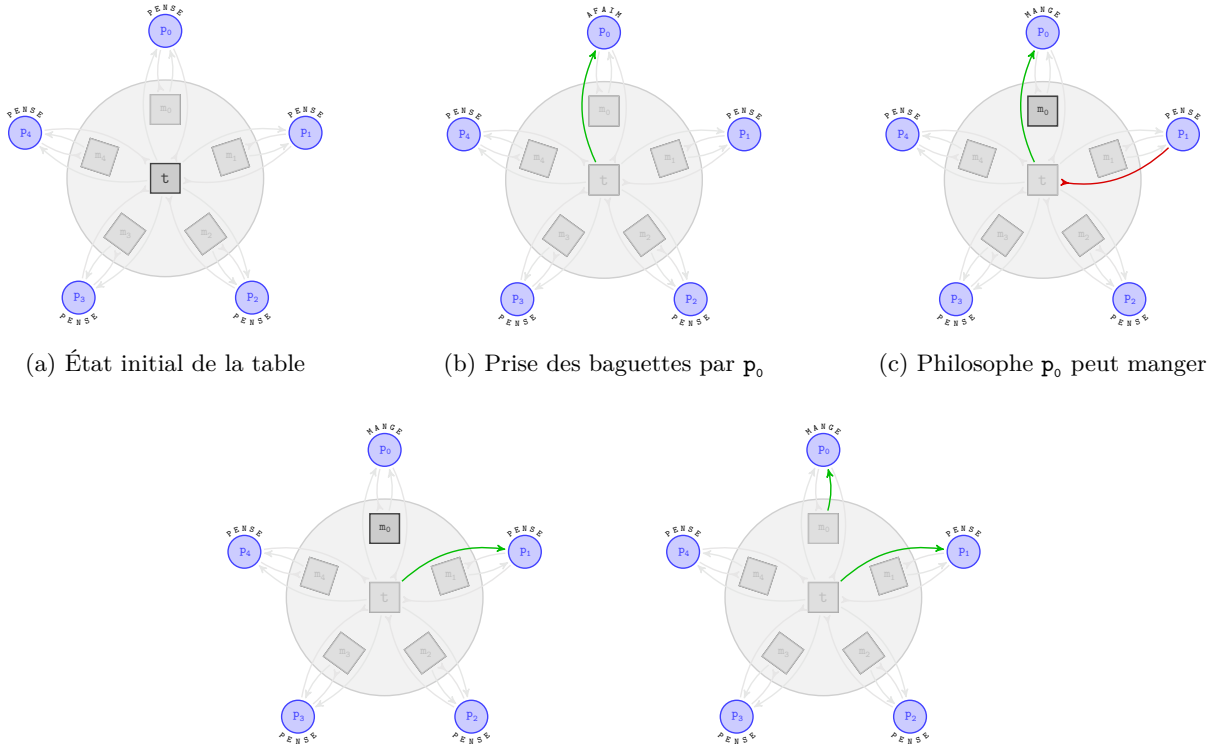
FIGURE 4.23 – Prise et dépôt des baguettes

Lorsqu'un philosophe est en train de `penser()`, son **`état`** reflète ceci en ayant la valeur `PENSE`. Une fois qu'un philosophe termine de `penser()`, il cherchera à `prendre()` ses baguettes. Pour ce faire, il basculera son **`état`** à la valeur `AFAIM` pour indiquer son désir de manger. Après ce changement, le philosophe va `tenter()` d'obtenir le droit de manger qui va dépendre uniquement de l'**`état`** actuel de trois philosophes (lui-même et ses deux voisins). Cette fonction que nous détaillons en dessous repose sur une condition impliquant les **`états`** ainsi qu'une modification d'un **`état`**. De ce fait, cet objet partagé peut faire l'objet d'accès concurrents et doit être protégé par un *mutex* global. Cependant, ce *mutex* sur la `table` est uniquement utilisé pour consulter et éventuellement modifier les **`états`** de manière atomique. Lorsqu'un *thread* a terminé de `manger()`, il doit `deposer()` ses baguettes ce qui consistent à basculer son **`état`** pour indiquer qu'il `PENSE` et ensuite `tenter()` d'accorder le droit à son voisin de gauche et de droite. Cette action va débloquer les voisins qui répondent aux conditions de la fonction `tenter()`.

```
14  enum etat { PENSE = 0, AFAIM = 1, MANGE = 2 };
15
16  mutex table = OUVERT;
17  etat etats[NB_PHILOSOPHES];       // etats[*] = PENSE
18  mutex peutManger[NB_PHILOSOPHES]; // peutManger[*] = FERMÉ
19
20  void tenter(int moi) {
21    if(etats[moi] == AFAIM && etats[GAUCHE(moi)] != MANGE && etats[DROITE(moi)] != MANGE) {
22      etats[moi] = MANGE;
23      unlock(peutManger[moi]);
24    }
25  }
```

FIGURE 4.24 – Partie commune de la solution avec suivi de l'`état` des philosophes.

Initialement le *mutex* de la `table` est `OUVERT`, les *mutex* qui représentent le droit de manger de chaque philosophe sont tous `FERMÉS` et chaque philosophe commence à l'`état` où il `PENSE`. La fonction `tenter()` va vérifier si le droit de manger doit être accordé à une certain philosophe et peut être invoquée par n'importe quel philosophe qui termine de `penser()` ou de `manger()`. Pour obtenir le droit de manger, il est suffit qu'un philosophe soit à l'`état` où il `AFAIM` et que son voisin de droite ne soit pas à l'`état` `MANGE` de même que son voisin de gauche ne soit pas à l'`état` `MANGE`.

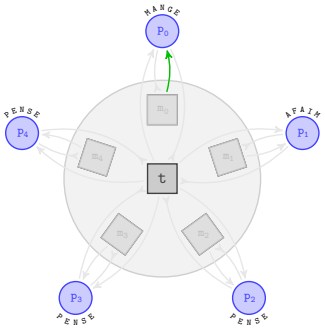


(a) État initial de la table (b) Prise des baguettes par p_0 (c) Philosophe p_0 peut manger

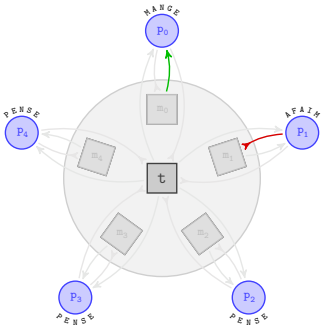
(d) Relâchement du *mutex* de `table` (e) Philosophe p_0 va `manger()`

FIGURE 4.25 – Évolution de l'exécution de la solution avec `états`

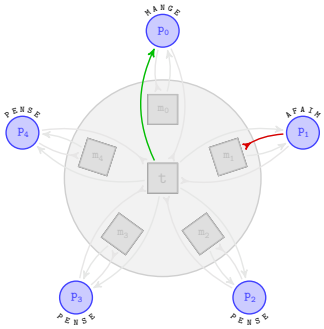
Initialement, tous les philosophes `PENSENT`, le *mutex* global est `OUVERT` et les *mutex* représentant le droit de manger de chaque philosophe est `FERMÉ` (*cfr.* Figure 4.25a). Lorsque le philosophe p_0 termine de `penser()` et commence la prise des baguettes, supposons qu'il a le temps de basculer son `état` à `AFAIM` (*cfr.* Figure 4.25b). Si le philosophe p_1 termine de `penser()`, sa tentative d'obtenir le *mutex* de `table` le bloquera pendant que le p_0 effectuera sa tentative d'obtenir le droit de manger. Étant donné qu'il `AFAIM` et que ses deux voisins ne sont pas à l'`état` `MANGE`, son `état` passera à `MANGE` et son *mutex* sera déverrouillé (*cfr.* Figure 4.25c). À ce stade, en relachant le *mutex* de `table`, le philosophe p_0 débloquera le philosophe p_1 et pourra saisir son droit pour commencer à `manger()` (*cfr.* Figures 4.25d et 4.25e).



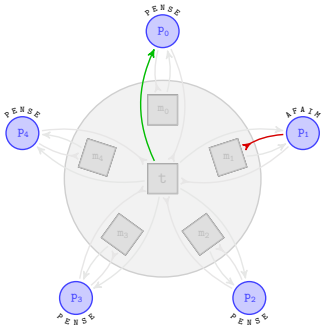
(a) Échec d'obtention par p_1



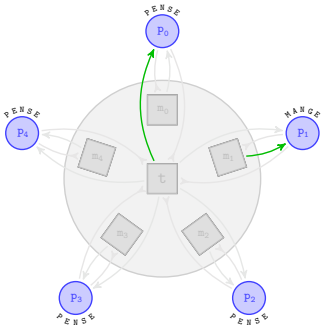
(b) Blocage du philosophe p_1



(c) Dépôt des baguettes par p_0



(d) Transition de l'`état` du philosophe p_0



(e) Tentative d'obtention du droit de manger par p_1

FIGURE 4.26 – Déblocage

Lorsque le philosophe p_1 réalise sa tentative d'obtenir son droit de manger, il n'obtiendra pas le droit de manger car son voisin de droite est à l'`état MANGE` (*cfr.* Figure 4.26a). Lorsque le philosophe p_1 essayera de verrouiller son *mutex* il bloquera à ce stade (*cfr.* Figure 4.26b). Si le philosophe p_0 termine de `manger()` il prendra le *mutex* de la `table` pour réaliser le dépôt de ses baguettes (*cfr.* Figures 4.26c et 4.26d). Étant donné que le philosophe p_0 va tenter d'accorder le droit au philosophe p_1 et que les deux voisins de celui-ci ne seront pas à l'`état MANGE`, son *mutex* sera déverrouillé ce qui aura pour effet de passer l'`état` de p_1 à `MANGE` et de le débloquer (*cfr.* Figure 4.26e). Par contraste, le philosophe p_4 n'étant pas à l'`état AFAIM`, son `état` ne sera pas passé à `MANGE` et son *mutex* restera verrouillé étant donné qu'il n'a pas besoin d'obtenir le droit de manger à ce stade de son fonctionnement.

Chapitre 5 Systèmes de fichiers

5.1 Problématique

Dans ce chapitre, nous allons nous pencher sur la troisième ressource importante que le système d'exploitation doit gérer ; la mémoire secondaire. Ce type de mémoire existe sur une machine moderne sous la forme d'un ensemble périphériques de stockages qui sont utilisés pour entreposer un certain volume de données que les processus du système sont amenés à utiliser.

L'objectif premier d'un système de fichier est d'organiser cet espace de stockage pour permettre deux opérations élémentaires ; lire des données et écrire des données. Un système de fichiers va offrir un ensemble de garanties sur ces données.

Tout d'abord, il garantit la **pérennité** des données au-delà de l'exécution du processus qui les produit ou qui les modifie. Une fois qu'un processus se termine, la mémoire principale qu'il occupait est libérée est potentiellement allouée à d'autres processus. De ce fait, tout le contenu qui était stocké dans cette mémoire est susceptible d'être modifié. Le système de fichiers va être utilisé pour permettre au processus se terminant d'écrire le contenu des données importantes dans un fichier qui pourra être lu lors d'une prochaine exécution.

Ensuite, le système de fichier doit offrir une **protection des accès** sur le contenu d'un fichier pour éviter qu'un processus non-autorisé ne lise ou ne modifie ce contenu. À cet effet, la notion de **permission** permet d'associer à chaque fichier des règles d'accès qui sont vérifiées par le système d'exploitation au moment de la demande d'ouverture (*i.e.* `open()`) du fichier. Ceci ne protège les accès que pour les processus qui tournent sur le système. Si un utilisateur démonte le périphérique de stockage et le monte sous un système d'exploitation qui ignore les permissions, l'utilisateur en question pourra librement accéder aux données. Pour se protéger contre ce type d'accès, il est nécessaire de recourir à un mécanisme plus avancé, comme le chiffrement des données à l'aide d'une *passphrase* secrète (*e.g. LUKS*, *FileVault*).

D'autre part, l'**efficacité** des opérations de lecture et d'écriture d'un fichier nécessitant de réaliser des accès physiques est importante. La manière d'organiser le contenu des fichiers à une conséquence directe sur le temps nécessaire pour déterminer si un fichier existe et, le cas échéant, lire complètement le contenu de ce fichier. Pour être utile, le système de fichiers devrait être implémenté de façon à offrir de bonnes performances pour ces opérations.

Enfin, la **résistance aux pannes** pourrait être accrue par différentes stratégies implémentées dans le système de fichiers. En effet, un périphérique de stockage est un objet physique qui est soumis aux problèmes d'usure et de défaillance que n'importe quel composante peut subir. Par exemple, un bloc ou un secteur d'un périphérique de stockage peut subir une corruption qui rend illisible les octets qui y sont stockés. Un système de fichiers pourrait prévoir des stratégies pour réduire le risque de perte de données, par exemple en répliquant les données importantes à plusieurs endroits d'un périphérique voire sur plusieurs périphériques (*cfr. RAID*).

L'une des difficultés auxquelles le concepteur du système est confronté se situe dans la grande diversité des supports physiques. Chaque type de support physique présente des caractéristiques qui le rendent approprié pour une usage particulier.

Les **bandes magnétiques** offrent une grande fiabilité est sont utilisées typiquement pour des solutions de *backups* sur site. Dans leur fonctionnement physique, une tête de lecture/écriture est placée au dessus de la bande qui peut être défilée dans deux sens différents (*i.e. forward*, *backward*). Elles sont beaucoup plus adaptée pour des accès séquentiels (*i.e.* lecture/écriture en faisant défiler la bande). Pour des accès aléatoires, le temps d'attente avant de disposer des données recherchées augmente rapidement en raison du temps de défilement pour placer la tête au bon endroit.

Les **disques durs** (*i.e.* HDD) sont parmi les types de périphériques de stockage les plus connus. Ceux-ci offrent une grande quantité de stockage à faible coût bien et présentent des vitesse d'accès assez rapides pour des accès tant séquentiels qu'aléatoires. D'autre part, les temps d'accès ne sont pas réguliers et dépendent de la construction physique.

Les **disques à état solide** (*i.e.* SSD) constituent un périphérique de stockage qui plus utilisé pour la vitesse des accès que la quantité de stockage. Ces temps d'accès plus rapides justifient un coût plus élevés. Cependant, les temps d'accès sont beaucoup plus réguliers par rapport à un *HDD*.

Citons également les périphériques qui ont plus comme utilité de stocker des données qui seront exclusivement lues (*i.e. CD*, *DVD*, *BluRay*) ou encore les supports de types cartes mémoires (*i.e.* SDHC, MM2, Flash).

```
1  #include <fcntl.h>   // open()
2  #include <unistd.h>  // read(), write(), close()
3  #include <stdlib.h>  // malloc(), free()
4  #include <minimum.h> // minimum()
5
6  #define TAILLE 8192
7  int tableau[TAILLE];
8
9  int main(int argc, char* argv[]) {
10   int fd = open("/home/sdy/data/tableau.bin", O_RDWR);
11   if (fd == -1) { perror("open"); return EXIT_FAILURE; }
12
13   int octets_lus = read(fd, tableau, TAILLE);
14
15   sort(tableau, octets_lus / 4);
16
17   lseek(fd, 0L, 0);
18   write(fd, tableau, octets_lus);
19
20   close(fichier);
21
22   return EXIT_SUCCESS;
23 }
```

FIGURE 5.1 – Programme de triage d'un fichier.

Le programme présenté à la Figure 5.1 repose sur l'utilisation de fichiers pour réaliser ces traitements. Notamment, le processus demandera l'ouverture d'un fichier particulier (*cfr.* ligne 10) suivi de sa lecture vers un espace de mémoire principale (*cfr.* ligne 13). Une fois le contenu de ce fichier trié (*cfr.* ligne 15), le processus demandera l'écriture du contenu trié sur le fichier d'origine (*cfr.* lignes $17 - 18$).

5.1.1 Arborescence des fichiers

Afin de faciliter la vie de l'utilisateur et du programmeur, le contenu d'un espace de stockage est organisé sous une forme d'arborescence. Les nœuds de cette arborescence peuvent être de deux types ; fichiers et répertoires. Pour pouvoir identifier un nœud particulier, un chemin doit être spécifié (*cfr.* Figure 5.1, ligne 10).

Un nœud de répertoire est utilisé pour regrouper un ensemble de répertoire et/ou de fichiers de manière à préserver un comfort d'exploration de l'arborescence complète. Pour l'essentiel, un nœud de répertoire va encoder la liste de tous les nœud de fichier et de répertoire qu'il contient.

Un nœud de fichier va correspondre à un contenu particulier, organisé suivant une structure interne. Par exemple, un fichier peut être organisé de manière séquentielle (*e.g.* code source, fichier texte brut) ou sous la forme de *records* (*e.g. SQLite3*). D'autre part, le nœud va également être utilisé pour stocker les attributs du fichier (*e.g.* taille, permissions, propriétaire, dates). Enfin, selon la nature du fichier, son type peut différer ; outre les fichiers de données, il est possible de trouver sur un système des fichiers *spéciaux* qui peuvent correspondre directement à une ressource physique du système. Par exemple, le fichier `/dev/sda1` donne accès aux octets bruts de la première partition du premier périphérique de stockage. Il est également possible d'accéder aux informations associées au processus `4272` en consultant les fichiers qui se trouvent dans le répertoire `/proc/4272`.

Selon le type de fichier (*e.g.* `.c`, `.jpg`, `.sqlite3`), le mode d'accès va différer en fonction de l'application. Dans un mode d'accès *séquentiel*, le contenu est typiquement lu dans l'ordre depuis le début jusque la fin (*e.g.* code source, fichier texte). Dans un mode d'accès *direct*, il est possible d'accéder à un endroit précis du fichier pour trouver les données pertinents (*e.g.* fichier exécutable, base de données).

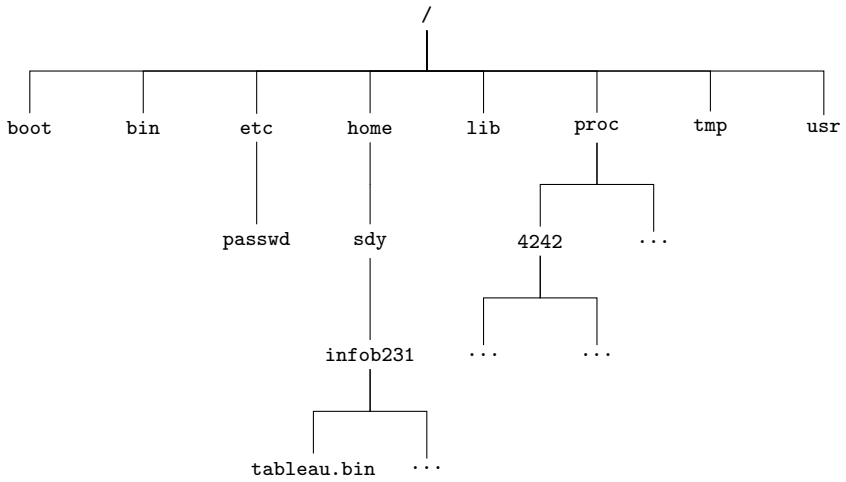


FIGURE 5.2 – Exemple d'arborescence

5.1.2 Supports physiques

Avant de plonger dans la manière d'organiser le contenu des fichiers et répertoires dans la mémoire secondaire, il est utile de se pencher sur la façon dont les supports physiques sont organisés et comment il est possible de les interroger.

Nous allons considérer dans la suite de ce chapitre qu'un périphérique de stockage peut être considéré comme étant un ensemble de **secteurs** (appelé aussi **blocs**) de taille fixe (*e.g.* 512B, 4KiB) qui sont identifiés de manière unique. Chaque bloc contient une en-tête (données d'identification), les données à proprement parler ainsi qu'un code de correction d'erreur. De la sorte, le contrôleur du périphérique est en mesure de chercher parmi l'ensemble des blocs, confirmer avoir trouvé celui qui l'intéresse sur base de son en-tête, vérifier si les données du bloc sont cohérentes avec le code ECC afin de détecter si une éventuelle corruption est présente et la corriger si possible.

Selon la construction physique du périphérique, les secteurs sont agencés physiquement de façon différente. Sur un volume de type bande magnétique, les secteurs sont placés les uns à la suite des autres (*i.e.* une seule dimension).

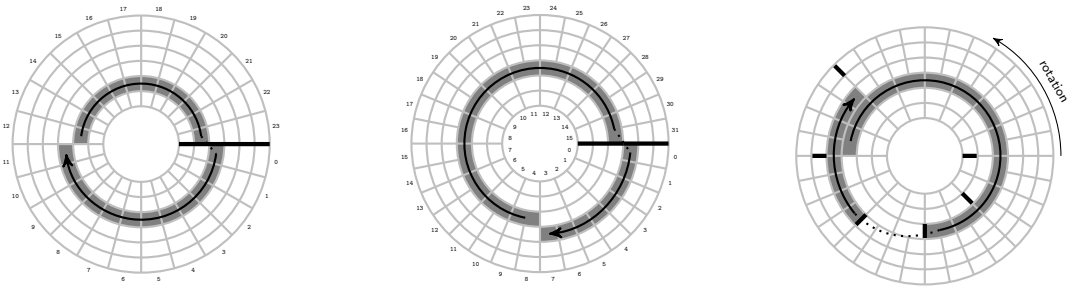
Sur un volume de type disque dur, les secteurs sont agencés selon un identifiant pour le **cylindre**, la **tête** et le **secteur** (*i.e. Cylinder-Head-Sector*) spécifique (*i.e.* trois dimensions). Les secteurs sont agencés en *pistes concentriques* sur chaque face de chaque plateau. Toutes les pistes qui sont équidistantes de l'axe de rotation au centre du disque sur les différents plateaux constituent un cylindre. Enfin, une tête de lecture/écriture est présente au dessus de chaque face pour effectivement lire un secteur particulier. Lorsque le disque est en activité, l'axe centrale est en rotation (*e.g.* 5400, 7200 rpm). Une fois que la tête de lecture sélectionnée est positionnée au dessus de la piste voulue, il est nécessaire d'attendre que le secteur désiré défile sous la tête de lecture. Sur un tel volume, il est également possible d'identifier les secteurs sur base d'un numérotation linéaire des secteurs (*i.e. Logical Block Addressing*).

Considérations géométriques

L'agencement physique des secteurs pour une organisation en cylindres-têtes-secteurs peut se faire de différentes façons. Dans le cas le plus simple consiste à placer les secteurs de manière régulière sur la surface de chaque plateau. Dans ce cas de figure, le numéro de tête permet de sélection une face d'un plateau et cette dernière peut être pensée comme étant un tableau à deux dimensions numérotés par le numéro de cylindre (*i.e.* ligne) et le numéro de secteur (*i.e.* colonne). Par exemple, la Figure 5.3a présente l'agencement de 6 cylindres comportant chacun 24 secteurs. Dans cette numérotation, les numéros de cylindres et de secteurs sont soumis aux restrictions suivantes ; $0 \leqslant \mathbf{C} \leqslant 5$ et $0 \leqslant \mathbf{S} \leqslant 23$.

Les pistes sont systématiquement découpées en un nombre régulier de secteurs. Ceci a pour conséquence que les secteurs sur le cylindre à la périphérie du plateau occupe une plus grande surface qu'un secteur sur le cylindre au centre du plateau. En pratique, il est possible de faire varier la densité de secteurs au fur et à mesure qu'une piste est éloignée du centre. La Figure 5.3b présente un autre agencement avec une densité variable ; 16 secteurs sur les trois centraux cylindres et 32 secteurs sur les trois à la périphérie. Il devient dès lors nécessaire de transformer une adresse régulière en cylindre-tête-secteur pour tenir compte de cette variabilité. De plus, au vu de la vitesse de rotation de l'axe centrale, le temps nécessaire pour faire un tour complet ne dépend change pas d'un cylindre à l'autre. De ce fait, il est possible de lire 16 secteurs sur le cylindre central mais 32 sur le cylindre à la périphérie en un tour complet.

Une dernière considération concerne le temps nécessaire à déplacer la tête de lecture/écriture d'un cylindre au suivant. Étant donné que le plateau continue de tourner pendant ce déplacement latéral, la numérotation peut comporter un biais pour permettre à lire de façon continue une séquence de secteurs qui se trouve sur deux cylindres consécutifs (*cfr.* Figure 5.3c).



(a) Adressage régulier (b) Densité variable de secteurs (c) Biais de numérotation de $\frac{1}{8}$

FIGURE 5.3 – Considérations d'agencement et de numérotation dans un système **CHS**

5.2 Modes d'allocation

Pour stocker un certain contenu sur disque, différents modes d'allocation des secteurs peuvent être utilisées. Dans cette section, nous allons considérer une numérotation simple des secteurs depuis le numéro `0` jusqu'à un numéro maximal qui dépend du nombre de secteur total disponible sur le support physique.

5.2.1 Allocation contïgue

Le premier mode d'allocation consiste à associer le numéro du **premier** secteur où commence le fichier ainsi que le **nombre** de secteurs occupés par celui-ci. Dans un tel mode d'allocation, il est possible de lire entièrement le fichier en une seule passe ; les secteurs étant placés de manière consécutive sur le périphérique considéré. Cependant, il est difficile d'étendre un fichier s'il remplit son dernier secteur et que celui qui suit est attribué à un autre fichier. Dans ce cas de figure, pour pouvoir agrandir le fichier, il est nécessaire soit de déplacer tous ses secteurs à un endroit du périphérique où il puisse être étendu, soit de déplacer l'autre fichier qui occupe le secteur suivant. Ce mode d'allocation est susceptible de présenter de la **fragmentation externe**, où il est possible de trouver des secteurs non alloués pris entre des secteurs qui sont alloués. En ce qui concerne les types d'accès, l'allocation contigüe est appropriée pour des accès séquentiels ainsi que des accès directs. Dans ce second cas, si un processus souhaite accéder dans un fichier à partir d'un point particulier (*i.e.* décalage en octets), le système de fichiers est en mesure de calculer le décalage en secteurs qui doit être ajouté au numéro du premier secteur.

La Figure 5.4 ci-contre présente une allocation de secteurs pour un fichier qui en requiert 8 (*e.g.* 4096 octets pour des secteurs de 512 octets). Imaginons que notre fichier commence au secteur numéro 10. Sur chaque ligne, le numéro représente celui du premier secteur de cette ligne et le décalage au dessus des colonnes représente la valeur à ajouter pour obtenir le numéro du secteur dans cette colonne. Par exemple, le secteur qui se trouve aux "coordonnées" `8` et `+2` a pour numéro `8 + 2 = 10`. De la sorte, les huits secteurs marqués en gris foncé (*i.e.* `10` jusque `17`). Dans le cadre d'une demande de lecture de 4096 octets, le système de fichiers va être utilisé pour demander au contrôleur du périphérique de récupérer ces numéros de secteurs pour ensuite les copier dans l'espace d'adressage du processus, à l'endroit où ce dernier s'attend à trouver le contenu.

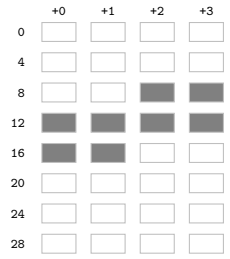


FIGURE 5.4 – Allocation contigüe pour un fichier de 8 secteurs.

5.2.2 Allocation chaînée

Le deuxième mode d'allocation consiste à associer le numéro du **premier** secteur et d'utiliser ce secteur comme la tête d'une liste chaînée des secteurs occupés par le fichier. Cette approche introduit un *overhead* sur les octets disponibles par secteur en raison du stockage d'un numéro du **suivant** dans chaque secteur. De plus, la corruption d'un secteur entraîne la perte non seulement des données du fichier qui s'y trouve mais également du numéro du secteur **suivant**. La fragmentation externe est éliminée étant donné qu'il est possible d'utiliser un secteur libre se trouvant entre deux secteurs occupés en faisant passer une liste chaînée par celui-ci. En matière d'accès, cette implémentation est appropriée pour les accès séquentiels en suivant la chaîne de secteurs. En revanche, pour réaliser un accès direct à un secteur particulier occupé par le fichier, il est nécessaire de récupérer tous les secteurs qui le précedent pour pouvoir identifier son numéro.

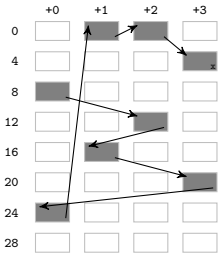


FIGURE 5.5 – Allocation chaînée pour un fichier de 8 secteurs.

La Figure 5.5 présente l'allocation chaînée pour un fichier qui requiert 8 secteurs. Imaginons que ce fichier commence au secteur numéro 8. En suivant les numéros de **suivants**, la séquence des secteurs occupés par le fichier est 14, 17, 23, 24, 1, 2 et 7 qui est le dernier secteur. En pratique, la logique de lecture consistera à utiliser un numéro de secteur **courant** initialisé au **premier** et à appliquer le traitement suivant tant que le numéro **courant** n'a pas la valeur dénotant la fin de la liste (*e.g.* `final`). Le traitement consistera à transférer le secteur **courant** depuis le périphérique vers la mémoire principale, copier les octets utiles du secteur vers la région du le mémoire où se trouve le `tableau` et mettre à jour le **courant** avec la valeur de **courant.suivant**.

5.2.3 Allocation chaînée avec table

Le troisième mode d'allocation constitue une variante du précédent. Un numéro de **premier** secteur est associé à un fichier et une chaîne de secteurs est construite à travers le périphérique de stockage pour stocker l'ensemble du fichier. Cependant, l'information de chaînage est encodée dans une **table d'allocation globale**. En pratique, cette table contient autant d'entrées qu'il existe de secteurs valides. Pour un numéro de secteur donné, la table encode le secteur **suivant** dans la chaîne dont il fait partie. Certaines valeurs spéciales permettent d'encoder le secteur `final` ou si un secteur particulier est `libre`.

Pour que le système de fichier puisse récupérer le contenu de n'importe quel noeud, la table doit être chargée dans la mémoire principale. La taille de la table dépend directement du nombre de secteurs disponibles sur le périphérique de stockage utilisant un tel mode d'allocation. De ce fait, l'allocation chaînée avec table est plus adapté pour des volumes de taille faible. De plus, la corruption d'un secteur qui contient la table d'allocation a pour conséquence possible la perte des informations de chaînage de plusieurs fichiers. Une solution facile pour ce problème est de stocker une copie de la table à un autre endroit du périphérique de stockage qui doit être synchronisée avec l'originale à tout moment. Ce mode d'allocation est adapté pour des accès séquentiels étant donné que la liste des secteurs à récupérer pour lire l'entièreté d'un fichier peut être obtenue en parcourant la **GAT**. Contrairement à l'allocation chaînée, il est possible de déterminer le numéro d'un secteur particulier sur base du décalage dans le fichier en consultant directement la **GAT** sans devoir charger de secteurs depuis le périphérique de stockage ce qui rend les accès directs plus efficaces.

La Figure 5.6 ci-contre représente l'agencement d'un fichier occupant 8 secteurs en utilisant une table d'allocation globale. Le fichier commence au secteur 8 et la chaîne traverse les secteurs 14, 17, 23, 24, 1, 2, 7. Celle-ci peut être reconstruite en consultant la **GAT** sur base d'un numéro de secteur. Lorsque le secteur courant est le numéro 17, l'entrée à cet indice encode le numéro 23. Pour le secteur 7, le marqueur `final` indique que le fichier se termine dans le secteur 7. En pratique, la logique de lecture consistera à utiliser un numéro de secteur **courant** initialisé au **premier** et à appliquer le traitement suivant tant que le numéro **courant** n'a pas la valeur dénotant la fin de la liste (*e.g.* `final`). Le traitement consistera à transférer le secteur **courant** depuis le périphérique vers la mémoire principale directement vers la région du le mémoire où se trouve le `tableau` et mettre à jour le **courant** avec la valeur de **GAT[courant]**.

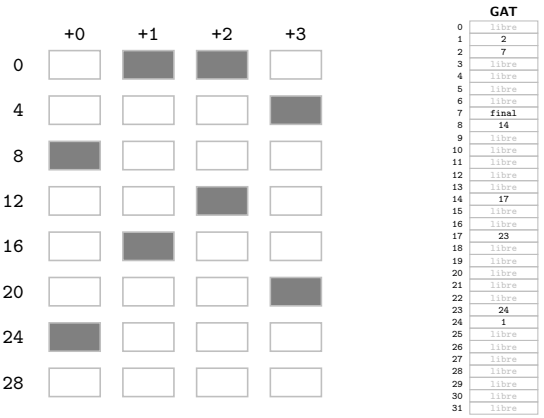


FIGURE 5.6 – Allocation pour un fichier sur 8 secteurs.

5.2.4 Allocation par *i-nodes*

Le quatrième mode d'allocation consiste à regrouper toutes les informations relatives aux secteurs occupés par un fichier dans un **nœud d'index** (*i.e. i-node*). Dans ce mode, à chaque fichier est associé un *index-node* qui regroupe les attributs (*e.g.* propriétaire, permissions, dates de création et de modification). De plus, les numéros de secteurs occupés par le fichier sont directement encodés dans l'*i-node*. Afin de pouvoir accéder au contenu du fichier, son *i-node* doit être localisé sur le périphérique de stockage et chargé dans la mémoire principale. Les numéros des premiers secteurs sont encodés directement dans l'*i-node* (*cfr.* Figure 5.7, tableau `blocs[*]`). De plus, pour ne pas limiter excessivement la taille maximale d'un fichier, une extension est possible à travers l'encodage *indirect* de numéros de secteurs additionnels (*cfr.* Figure 5.7, `blocPointeurs`). Le pointeur de secteurs *indirect* pointe vers un secteur qui encode directement des numéros de secteurs additionnels qui sont occupés par le fichier.

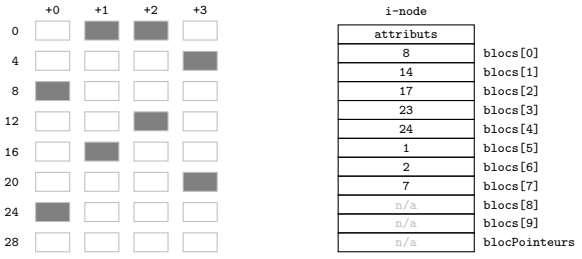


FIGURE 5.7 – Allocation pour un fichier de 8 secteurs.

La Figure 5.7 représente l'occupation de 8 secteurs par un fichier auquel est associé l'*i-node* représenté du côté droit. Dans le cadre de la lecture de ce fichier, la séquence de secteurs qui seront récupérés sera 8, 14, 17, 23, 24, 1, 2 et 7. En pratique, la logique de la lecture consistera à utiliser un indice **courant** initialisé à 0 et à appliquer le traitement suivant tant le numéro de secteur courant est assigné (*i.e.* **inode.blocs[courant] !=** `n/a`) et que l'indice se situe dans les limites du tableau (*i.e.* **courant** < 10). Le traitement consiste à transférer le secteur **inode.blocs[courant]** vers la mémoire principale et avancer l'indice (*i.e.* **courant +=** `1;`). Une fois cette première boucle est arrivée à son terme, s'il est encore nécessaire de récupérer du contenu et que **inode.blocPointeurs !=** `n/a`, le secteur en question est transféré vers la mémoire principale et la boucle précédente est appliquée sur son contenu.

5.2.5 Considérations avancées

L'implémentation d'un système de fichiers peut être améliorée par l'adjonction d'une *cache* (dans la mémoire principale) pour les secteurs fréquemment accédés. De la sorte, lorsqu'un secteur doit être lu, le système de fichiers peut consulter en premier lieu la cache et, le cas échéant, demander au périphérique de stockage de lire le secteur en question. Pour la gestion des écritures, cette *cache* peut être utilisée suivant un mode *write-through*; lors d'une tentative d'écriture, le système de fichiers peut inscrire la nouvelle version du secteur et directement demander son inscription par le périphérique de stockage.

Lors d'une lecture séquentielle d'un fichier, le système de fichier peut prendre les devants et demander au périphérique les secteurs qui suivent ceux qui sont requis au vu de la demande de lecture (*read-ahead*). Par exemple, si un fichier occupe les secteurs de 512 octets; 8, 14, 17, 23, 24, 1, 2, 7 et qu'un processus demande à lire 1024 octets, le système de fichiers doit récupérer les secteurs 8 et 14 mais il peut déjà récupérer les secteurs 17 et 23 en plus.

Il est également possible d'optimiser le placement de secteurs fréquemment utilisés sur le périphérique de stockage afin de réduire le temps nécessaire pour les récupérer. Par exemple, les *i-nodes* pourraient être placés sur les cylindres à la périphérie des plateaux étant donnés qu'il est possible de lire plus de secteurs par rotation à la périphérie. Une considération plus proche du contrôleur du périphérique a trait à l'ordonnancement des opérations de lectures (*e.g.* minimisation des déplacements des têtes de lecture).

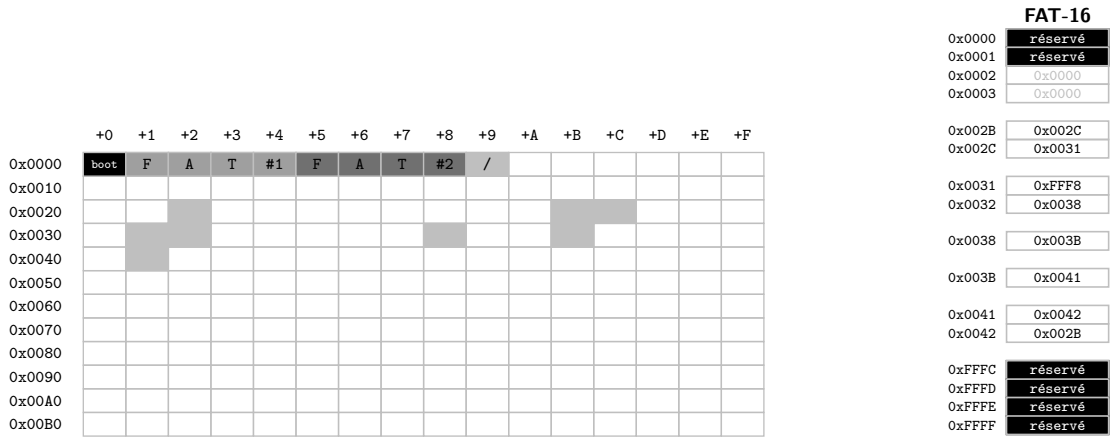
5.3 Illustration — FAT16

Dans une partition formatée en FAT16, les secteurs consécutifs sont regroupés en **clusters** d'une taille fixe comprise entre 4 et 64 secteurs. Ce regroupement permet de limiter la fragmentation externe et d'éviter qu'un fichier ne soit trop éparpillé sur le périphérique de stockage. Chaque cluster, qui constitue une unité élémentaire d'allocation, est identifié en utilisant un nombre codé sur 16 bits (d'où le 16 dans FAT16). Le mode d'allocation chaînée avec table est utilisé ; la *File Allocation Table* étant la structure de données utilisée pour encoder les chaînes de clusters alloués à un fichier. Cependant, la FAT permet également d'encoder des informations additionnelles à propos des clusters en donnant des significations spéciales à certaines valeurs. Ces significations sont spécifiées dans la Table 5.8.

Valeur	Signification
`0x0000`	Cluster libre
`0x0001`	Valeur **réservée**
`0x0002-0xFFEF`	Cluster de données **suivant**
`0xFFF0-0xFFF6`	Valeurs **réservées**
`0xFFF7`	Cluster inutilisable
`0xFFF8-0xFFFF`	Dernier cluster de données

FIGURE 5.8 – Significations des valeurs apparaissant dans la FAT

La valeur spéciale `0x0000` est utilisée pour représenter le fait que le cluster en question est libre et de ce fait disponible si un fichier à besoin d'une extension d'espace de stockage. Lorsqu'une valeur est comprise entre `0x0002-0xFFEF`, celle-ci représente le numéro du cluster **suivant** dans la chaîne. Il est possible qu'un secteur devienne défectueux, auquel cas la valeur spéciale `0xFFF7` doit être utilisée pour marquer le cluster comme étant inutilisable. Les valeurs de `0xFFF8-0xFFFF` peuvent être utilisées pour marquer le fait que le cluster courant est le dernier de la chaîne. Dans toute spécification, certaines valeurs peuvent être réservées, ce qui signifie qu'elles ne devraient pas être utilisées (*e.g.* `0x0001`, `0xFFF0-0xFFF6`).



(a) Organisation d'une partition

(b) Chaînage des *clusters*

FIGURE 5.9 – Illustrations de l'organisation d'un système de fichier en FAT16.

Supposons un taille de cluster de 32 KiB (cluster de 64 secteurs de 512 octets). La Figure 5.9a représente l'organisation d'une partition où haque case représente un cluster. Le premier secteur d'une partition (*i.e.* cluster 0x0000 ; *boot sector*) reprend tous les paramètres de la partition ainsi qu'une portion de code qui permet de démarrer le système d'exploitation qui est stocké sur cette partition au démarrage de la machine. Ensuite, la FAT de la partition est inscrite (*i.e.* clusters `0x0001-0x0004`), qui a une taille maximale de 2^{16} entrées de 2 octets (*i.e.* 16 bits), pour un total de 128 KiB (soit quatres clusters). Pour limiter les dégâts qui serait occasionnée par la perte d'un secteur de ces clusters, une copie est stockée à la suite (*i.e.* clusters `0x0005-0x0008`). Le cluster

`0x0009` est utilisé pour encoder le descripteur de la racine de l'arborescence ; le contenu du répertoire au sommet de celle-ci.

Enfin, tous les clusters restants constitue la **région de données** qui est couverte par tous les numéros valides `0x0002-0xFFEF`, soit $2^{16} - 16$ clusters de 32.768 octets pour un peu moins de 2.048 MiB. Les clusters marqués en gris clair représentent ceux attribués à un fichier commençant au cluster `0x0032` et dont la chaîne peut être reconstruite depuis la FAT ; `0x0038`, `0x003B`, `0x0041`, `0x0042`, `0x002B`, `0x002C` et `0x0031`.

Pour encoder un nœud de répertoire, 32 octets consécutifs sont utilisés pour représenter les informations d'une entrée du répertoire.

Octets	Signification
8	Nom court
3	Extension
1	*Flags*
1	Réservé
5	Date de création
2	Date du dernier accès
4	Attributs étendus
4	Date de dernière modification
2	Numéro du premier cluster
4	Taille du fichier (max. $2^{32} = 4$ GiB)

FIGURE 5.10 – Détails d'une entrée de répertoire

Les 8 premiers octets d'une entrée encode le nom du fichier au format ASCII (*e.g.* `tableau`). Viennent ensuite 3 octets d'extension (*e.g.* `bin`, `exe`, `jpg`). Les différents *flags* associé au noeud de l'entrée peuvent être stocké (*e.g.* lecture seule, sous-répertoire). L'octet suivant est réservé et viennent ensuite les octets encodante la date et l'heure de création (4 octets) ainsi que la date du dernier accès (2 octets). Les attributs étendus permettent d'encoder les informations relatives aux droits d'accès (*e.g.* propriétaire, permissions). La date de dernière modification (4 octets) est suivie du numéro du premier cluster (2 octets) où commence le contenu de ce noeud. Dans le cadre d'une demande d'ouverture de fichier, l'objectif consiste à trouver ce numéro sur base du chemin spécifié. Enfin, la taille du fichier (en octets) est encodée sur les 4 derniers octets de l'entrée.

5.4 Illustration — ext3

Dans une partition ext3, l'espace est organisé en **blocs** de taille fixe (*e.g.* $n = 1024$, $n = 2048$ ou $n = 4096$ octets). Le premier bloc (*i.e. boot block*) est utilisé lors du démarrage de la machine pour localiser, charger et démarrer le système d'exploitation qui est stocké sur la partition. Le reste de la partition est organisé en **groupes** de blocs consécutifs dont la taille dépend directement de la taille choisie pour les blocs. Chaque groupe commence par un *superblock* (1 bloc) qui contient les méta-données du système de fichiers (*e.g.* nombre de blocs, nombre d'*i-nodes*, nombre de blocs par groupe). Après le *superblock*, les descripteurs de groupe (n blocs) sont stockés qui encodent des informations à propos de chaque groupe (*e.g.* nombre de blocs libres, nombre d'*i-nodes* libres). Afin de suivre l'état d'occupation d'un groupe, celui-ci contient deux *bitmaps* ; le premier pour l'état d'allocation des blocs (1 bloc), le second pour les *i-nodes* (1 bloc). À la suite de ces deux *bitmaps* se trouve la table des *i-nodes* (n blocs) suivie des blocs de données (n blocs). Le contenu d'un fichier est confiné autant que possible dans un seul groupe.

Une partition ext3 fonctionne suivant un mode d'allocation par *i-node* qui sont utilisés pour encoder toutes les informations relatives à un nœud de l'arborescence. Un *i-node* est encodé sur 128 octets dont quelques valeurs intéressantes sont mise en évidence à la Table 5.11.

Octets	Signification
2	Type de fichier, permissions (*i.e.* `rwx`)
2	Propriétaire (`uid`)
48	Pointeurs directs
4	Pointeur indirect de niveau 1
4	Pointeur indirect de niveau 2
4	Pointeur indirect de niveau 3

FIGURE 5.11 – Détails d'un *i-node* en ext2

Le type de fichier est utilisé pour décrire plus spécifiquement la nature du nœud représenté (*e.g.* fichier, répertoire, lien symbolique). Le propriétaire encode l'identifiant de l'utilisateur (*i.e. user identifier*) à qui appartient ce nœud particulier. Un total de 12 pointeurs directs sont encodés dans l'*i-node* qui représentent les pointeurs vers les 12 premiers blocs de données qui sont occupés par le nœud. Pour étendre le fichier au-delà, le pointeur indirect de niveau 1 permet de référencer un bloc dans lequel le système de fichier peut encoder des pointeurs directs additionnels. Lorsque ce bloc est rempli, le pointeur indirect de niveau 2 peut être utilisé pour référencer un block qui est utiliser pour encoder des pointeurs indirects de niveau 1. Enfin, le pointeur indirect de niveau 3 référence un bloc qui peut contenir des pointeurs indirects de niveau 2.

En supposant une taille de bloc de 1024 octets, la taille maximale d'un fichier est donnée en calculant le nombre total de blocs de données qui peuvent être encodé dans un *i-node* sous forme de pointeurs directs ou indirects. Un pointeur de bloc étant encodé sur 4 octets, un pointeur indirect de niveau i rend possible l'encodage de $(1.024/4)^i$ pointeurs directs. Ceci nous donne la taille maximale suivante ;

$$\left(12 + \frac{1.024}{4} + \left(\frac{1.024}{4}\right)^2 + \left(\frac{1.024}{4}\right)^3\right) \times 1.024 \approx 16\text{GiB}$$

5.4.1 Journalisation

Afin de garantir la cohérence de l'écriture d'un bloc, ext3 implémente un mécanisme de journalisation. Étant donné qu'un bloc recouvre potentiellement plusieurs secteurs (*e.g.* blocs de 2.048 octets, secteurs de 512), si une panne survient pendant l'écriture d'un bloc, celui-ci peut être inscrit de manière incomplète sur le périphérique de stockage, ce qui introduit une incohérence dans le fichier correspondant.

La solution de la journalisation consiste à utiliser un journal, stocké dans la partition, pour suivre la progression des opérations d'écriture. Lorsqu'un bloc doit être écrit, une entrée est crée dans le journal qui va contenir une copie du bloc ainsi que les méta-données associées. Ensuite, l'écriture du bloc est entamée qui consiste à envoyer les ordres d'écriture au contrôleur du périphérique de stockage ainsi que des méta-données du système de fichiers. Une fois l'écriture de tous les secteurs conclue, l'entrée du journal peut être supprimée. Supposons qu'une panne survienne pendant la création de l'entrée dans le journal. Au redémarrage suivant, le système de fichiers ignorera les entrées incomplètes du journal. Supposons qu'une panne survienne pendant l'opération d'écriture. Dans ce cas, au redémarrage suivant le système de fichier consultera le journal et retentera toutes les opérations d'écriture qui y sont inscrites.

Cette technique augmente le nombre d'écritures sur disque, étant donné que la création d'une entrée du journal requiert d'écrire le bloc dans le journal avant de l'écrire à l'endroit voulu.

Chapitre 6 Entrées/Sorties

6.1 Problématique

Les Chapitres 1 et 3 nous avons vu que le processus en cours d'exécution peut être suspendu temporairement ou bloqué pour permettre le traitement d'un certain évènement. Selon qu'il s'agisse d'une demande de service (*e.g.* `open`, `read`) ou la survenance d'une interruption (*e.g.* frappe au clavier, fin de lecture par le contrôleur d'un disque). De ce fait, le processeur est effectivement réparti entre le système d'exploitation et les processus qui existent. Nous avons vu que les processus peuvent être catégorisés en *CPU-bound* et *IO-bound* selon la fréquence à laquelle ces processus demandent des opérations qui nécessitent des entrées-sorties vis-à-vis de périphériques externes au processeur.

La Figure 6.1 présente une architecture typique de carte mère où le processeur, la mémoire physique et la carte graphique sont tous interconnectés par le contrôleur mémoire. Au-delà, le contrôleur des entrées-sorties fait le lien avec tous les périphériques externes (*e.g.* disques durs, SSD, cartes réseaux) pour permettre au processeur de communiquer avec ceux-ci. En pratique, la connectique utilisée va présenter des vitesses de transfert variables ; SATAIII (600 MB/s), USB 3.0 (SuperSpeed, 625 MB/s) ou encore PCIe 3.0 (985 MB/s pour du 1x).

D'autre part, les vitesses d'accès brutes varient selon la nature du périphérique externe ; HDD de 7.2 kRPM (100 MB/s), SSD (550 MB/s) ou encore WiFi 802.11n (54.25 MB/s) qui va également dépendre de la connectique utilisée.

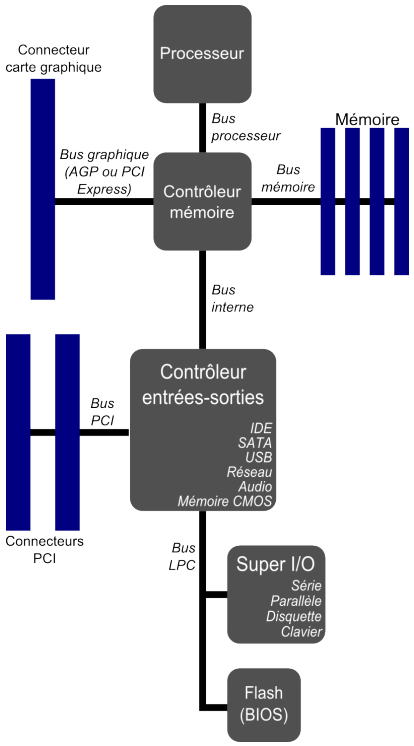


FIGURE 6.1 – Architecture de carte mère (Source : Gribeco@commons.wikimedia.org)

Toutes les demandes d'entrées-sorties résultant des demandes venant des processus doivent être envoyées par le système d'exploitation pour permettre à ces processus demandeurs de progresser dans leurs traitements. Un objectif secondaire pour le système d'exploitation est d'utiliser tous les périphériques externe de façon optimale et de les saturer autant que possible au vu de leurs capacités respectives.

Dans ce chapitre, nous allons nous pencher sur la nature des traitements requis dans le cadre de la gestion des entrées-sorties et comment ces dernières peuvent être traitées par le système d'exploitation.

6.2 Mécanismes de communication

Afin de permettre au système d'exploitation de communiquer avec les périphériques externes (*i.e.* envoi de demandes de lecture/écriture), le processeur doit mettre en place un mécanisme de communication qui peut être utilisé. Nous allons couvrir deux mécanismes bas-niveau qui peuvent être utilisés pour envoyer de ordres aux périphériques externes ou vérifier l'état d'une opération en cours.

6.2.1 Ports *I/O*

La notion de ports d'entrée-sortie repose sur la possibilité d'envoyer ou de recevoir des octets individuels vis-à-vis d'un périphérique externe. Le nombre de ports disponibles est déterminé par le processeur utilisé et chaque port est identifié par un numéro unique. Par convention, certains ports doivent être utilisés pour communiquer avec un périphérique externe pour recevoir une information particulière ou envoyer un ordre spécifique. Fréquemment, les ports *I/O* sont associés aux registres du périphériques et peuvent être utilisés pour récupérer ou modifier la valeur actuelle stockée dans un registre. Pour pouvoir réaliser ces envois/réceptions, des instructions spéciales sont disponibles dans le jeu d'instruction de l'architecture processeur. Les communications à travers les ports *I/O* sont du type *character device*, où le canal de communication représente un flux de caractères, soit une séquence d'octets non adressables sur lequel il est uniquement possible d'envoyer ou de recevoir un octet.

L'instruction `in REG, PORT_ID` peut être utilisée pour recevoir un certain nombre d'octets venant du port `PORT_ID` et de les stocker dans le registre `REG`. L'ordre de placement des octets dans le registre (*i.e. little-endian*, *big-endian*) est en spécifié dans la documentation du processeur. De façon complémentaire, l'instruction `out REG, PORT_ID` est utilisée pour envoyer un certain nombre d'octets stockés dans le registre `REG` à destination du port `PORT_ID`.

```
1  unsigned char caracteres[128] = {
2    0, 0, '1', '2', '3', '4', '5', '6', '7', '8', '9', '0', ')', '-', '\b',
3    '\t', 'a', 'z', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p', '^', '$', '\n',
4    0, 'q', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', 'm', '%', 0, 0, '$',
5    'w', 'x', 'c', 'v', 'b', 'n', ',', ';', ':', '=', 0, '*', 0, ' ',
6    0,0,0,0,0,0,0,0,0,0,0,0,0,0,'-',0,0,0,'+',0,0,0,0,0,0,0,0,0,0,0,
7  };
8
9  void gestion_clavier() {
10   unsigned char scancode = inportb(0x60); // lecture d'un octet sur le port 0x60
11   if (~scancode & 0x80)
12     putchar(caracteres[scancode]); // afficher à l'écran
13 }
```

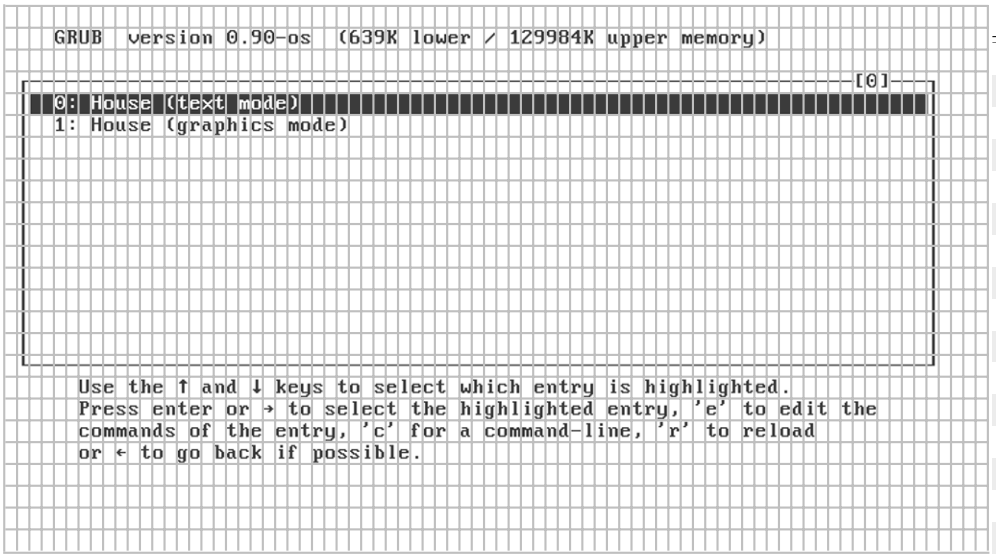
FIGURE 6.2 – Gestion des frappes au clavier avec utilisation d'un port *I/O*.

En guise d'illustration, considérons le cas des frappes au clavier. Lorsque l'utilisateur manipule une touche, une interruption est générée par le contrôleur du clavier (*i.e.* `IRQ1`) qui provoque un déroutement au niveau du processeur (*cfr.* Chapitre 1). À ce stade, un *scancode* est disponible sur le port `0x60` qui encode l'évènement spécifique qui s'est produit. Ce *scancode* est encodé sur 1 octet pour lequel le bit de poids le plus fort dénote si la touche a été enfoncée (*i.e.* bit à `0`) ou relâchée (*i.e.* bit à `1`). Les 7 bits de poids faible encodent un code identifiant la position physique de la touche qui est décrit dans le standard que le clavier implémente (*e.g.* PC-105). Sur base d'une *keymap*, la traduction peut être faite entre le *scancode* et le code ASCII du symbole que la touche est censée générer.

La Figure 6.2 présente le code d'une simple fonction de gestion des frappes au clavier. Celle-ci représente le *traitement effectif* à réaliser lorsqu'une interruption du clavier survient. Après que le contexte du processus courant ait été sauvegardé, cette fonction va exécuter en premier lieu la ligne 10 pour récupérer un octet sur le port `0x60`. La fonction `inportb` dissimule simplement l'instruction `in` et renvoie un `unsigned char` qui représente l'octet reçu. Le test de la ligne 11 réussira uniquement si le bit 7 du *scancode* est positionné à `0` (*i.e.* lorsque la touche est enfonce) et utilisera la fonction `putchar` qui place à la position courante à l'écran le caractère correspondant à travers la *keymap* définie aux lignes $1-7$. En pratique, si l'utilisateur enfonce la touche ayant pour valeur de *scancode* `20`, cette fonction affichera un `t` à la position courante du curseur à l'écran.

6.2.2 *Memory-Mapped I/O*

L'autre approche pour communiquer repose sur un *mapping* de certaines adresses physiques (*cfr.* Chapitre 2) vers une zone de mémoire qui réside effectivement sur le périphérique externe. Cette correspondance est mise en place au niveau du contrôleur mémoire (*cfr.* Figure 6.1) qui se charge de traduire les adresses physiques reçues du processeur pour réaliser l'accès sur le bon périphérique. Sous cette approche, les instructions élémentaires d'accès mémoire (*e.g.* `lw`, `sw`) peuvent être utilisées et une logique d'accès à un tableau peut être appliquées pour manipuler le contenu de la mémoire ciblée.



(a) Affichage en mode texte. Mise en évidence des cases individuelles.

Valeur	Couleur
0x0	Noir
0x1	Bleu
0x2	Vert
0x3	Cyan
0x4	Rouge
0x5	Magenta
0x6	Brun
0x7	Gris clair
0x8	Gris
0x9	Bleu clair
0xA	Vert clair
0xB	Cyan clair
0xC	Rouge clair
0xD	Magenta clair
0xE	Jaune
0xF	Blanc

(b) Code des couleurs

Une carte graphique moderne dispose d'un mode de fonctionnement appelé le mode texte. Lorsque la carte travaille dans ce mode, l'écran est divisé en une console de 25 lignes de 80 colonnes (soit 2000 cases). Par convention, ce tableau de 2000 cases est accessible au départ de l'adresse physique `0x000B8000`. La Figure 6.3a représente un exemple d'écran de démarrage (*i.e. bootloader*) avec la mise en évidence des cases en mode texte. Ces 2000 cases sont encodées sous la forme d'un tableau dont les éléments occupent 16 *bits*. Chaque mot de 16 est divisé en trois parties ; les 8 bits de poids faible (*i.e. bits* $0-7$) contiennent le code ASCII du caractère qui doit être affiché à cet emplacement, les 4 bits de poids supérieur (*i.e. bits* $8-11$) encode la couleur du caractère et les 4 bits de poids le plus fort (*i.e. bits* $12-15$) encode la couleur de fond. La correspondance entre les valeurs de ces deux groupes de 4 *bits* et la couleur qu'il représente est listée à la Figure 6.3b.

```
1  #include "printk.h"
2
3  unsigned short *vga = (unsigned short*) 0x000B8000;
4  unsigned short p = 0;
5
6  void putchar(char c) {
7    vga[p] = 0x0F00 | c;
8    p = (p+1) % (25*80);
9  }
10
11 void printk (const char *str) {
12   int i = 0;
13   while (str[i] != '\0') {
14     putchar(str[i]);
15     i++;
16   }
17 }
```

FIGURE 6.4 – Gestion de l'affichage en mode texte via *MMI/O*.

Pour pouvoir afficher des chaînes de caractères à l'écran, les fonctions `printk()` et `putchar()` dont l'implémentation est donnée à la Figure 6.4. La fonction `printk()` va simplement parcourir l'entiereté de la chaîne de caractères `str` et appeler la fonction `putchar()` sur le caractère courant. Cette fonction va simplement placer, dans la case courante du tableau `vga` (*i.e.* le contenu à l'écran) le caractère passé en paramètre qui sera affiché

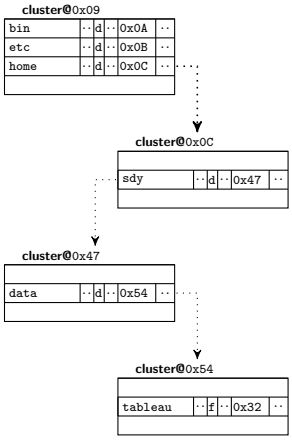
en `blanc` sur fond `noir`. La ligne 7 utilise l'opérateur *bitwise* `|` pour combiner les 8 *bits* du caractère `c` avec les 16 *bits* dont la partie supérieure encode les couleurs (*i.e.* `0x0F`).

6.3 Phases de traitement

Nous allons voir comment est organisée la gestion d'une entrée-sortie et comment les différentes phases qui la constituent peuvent être répartie entre différentes composantes logicielles et matérielles.

```
1  #include <fcntl.h>   // open()
2  #include <unistd.h>  // read(), write(), close()
3  #include <stdlib.h>  // malloc(), free()
4  #include <minimum.h> // minimum()
5
6  #define TAILLE 8192
7  int tableau[TAILLE];
8
9  int main(int argc, char* argv[]) {
10   int fd = open("/home/sdy/data/tableau.bin", O_RDWR);
11   if (fd == -1) { perror("open"); return EXIT_FAILURE; }
12
13   int octets_lus = read(fd, tableau, TAILLE);
14
15   sort(tableau, octets_lus / 4);
16
17   lseek(fd, 0L, 0);
18   write(fd, tableau, octets_lus);
19
20   close(fichier);
21
22   return EXIT_SUCCESS;
23 }
```

(a) Programme de triage



(b) Clusters FAT16

FIGURE 6.5 – Illustration d'un programme et d'un agencement du système de fichiers.

Nous considérons le programme de triage de la Figure 6.5a ainsi que l'organisation des clusters encodant les répertoire concerné par ce programme à la Figure 6.5b. Le cluster `0x09` encode les entrées du répertoire racine du système de fichier et indique que le sous-répertoire `/home` est encodé dans le cluster `0x0C`. La chaîne de répertoire est constitué du sous-répertoire `/home/sdy` (*cfr.* cluster `0x47`), `/home/sdy/data` (*cfr.* cluster `0x54`) qui référence le cluster où commence le fichier `/home/sdy/data/tableau.bin` (*i.e.* cluster `0x32`).

6.3.1 Composition d'une entrée-sortie

Selon le service qui est sollicité par un processus (*e.g.* `open`, `read`, `write`), le système d'exploitation est susceptible de devoir réaliser une ou plusieurs entrées-sorties afin de pouvoir y répondre. Tant que la demande n'a pas été complètement traitée, le processus demandeur devra être **bloqué**. De ce fait, plusieurs opérations d'entrées-sorties peuvent être requise avant de pouvoir procéder à un **déblocage** du processus demandeur.

Par exemple, sur une partition formatée en FAT16, l'objectif de l'appel `open` consiste à identifier le numéro du **premier** cluster où commence le fichier. Pour se faire, il est nécessaire de lire au moins 4 secteurs depuis le disque, chacun de ces lectures représentera une opération d'entrée-sortie. Cet appel est susceptible d'échouer si un nœud du chemin demandé n'existe pas où que les permissions du fichier ne permettent pas son ouverture par le processus demandeur. Si l'ouverture a réussi, un appel `read` va nécessiter de lire autant de secteurs que nécessaire pour couvrir la quantité d'octets demandés.

Dans le cas d'une partition formatée en ext3, l'objectif de l'appel `open` consiste à localiser l'*i-node* associé au fichier et à le charger en mémoire principale. De nouveau, ceci nécessitera de réaliser plusieurs lectures intermédiaires à travers la partition pour localiser celui-ci. Une fois l'ouverture réussi, l'appel à `read` nécessitera la lecture d'autant de blocs (via les pointeurs directs et indirects) que nécessaires pour satisfaire la quantité d'octets demandée. La lecture de ces blocs prendra la forme d'opérations d'entrées-sorties pour transférer les

secteurs sous-jacents.

Cependant, une entrée-sortie peut également être réalisée suite à l'occurrence d'une interruption (*e.g.* réception d'un paquet réseau). Dans ce cas de figure, le système d'exploitation devra identifier quel processus est le destinataire de ce paquet (*e.g. IPv4*, numéro de port *TCP*). À partir de ce point, le système d'exploitation devra réaliser des opérations d'enrées-sorties pour transférer le paquet depuis la mémoire de la carte réseau (*e.g.* par *MMI/O*) vers la mémoire principale du processus destinataire.

La **phase de préparation** d'une entrée-sortie consiste à envoyer les ordres d'opérations vers le contrôleur du périphérique concerné. Par exemple, la lecture du cluster `0x47` pour des clusters de 32 KiB nécessiterait la lecture des secteurs `4544`, `4543`, ..., `4608`. Ces ordres peuvent être tous envoyés directement ou le système d'exploitation peut progressivement demander ces secteurs et uniquement demander le suivant si le secteur récupéré ne contient pas l'information dont il a besoin.

La **phase d'exécution** d'une entrée-sortie consiste à réaliser effectivement l'opération demandée. Par exemple pour une lecture sur disque, ceci nécessite de démarrer la rotation des plateaux si ceux-ci sont à l'arrêt, déplacer le bras sur le bon cylindre, commuter vers la bonne tête de lecture/écriture, attendre éventuellement que le secteur attendu passe sous la tête, lire le secteur et l'inscrire dans un *buffer* interne au contrôleur en attente de récupération et enfin préparer une notification d'accomplissement ou d'erreur (*e.g.* disponible sur un port *I/O*).

La **phase de récupération** d'une entrée-sortie de lecture sur disque consiste à transférer le secteur lu depuis le *buffer* interne du contrôleur vers la mémoire principale.

Ces trois phases apparaissent toujours dans une opération d'entrée-sortie, cependant il est possible de les répartir entre différentes composantes selon les capacités matérielles. Nous allons nous pencher sur trois manière d'implémenter ces trois phases.

6.3.2 *Programmed I/O*

Dans sa forme la plus simple, les entrées-sorties programmées sont intégralement prise en charge par le système d'exploitation. Cependant, la phase d'exécution va différer si le périphérique externe ne dispose pas d'un contrôleur dédié. Dans ce cas, le système d'exploitation devra lui-même réaliser le contrôle pendant la phase d'exécution. Si par contre un contrôleur dédié est attaché au périphérique externe, le système d'exploitation devra réaliser une attente active (*i.e. polling*) pour interroger continuellement le contrôleur pour savoir si l'opération est conclue (*i.e. tant que* le contrôleur est `OCCUPÉ`) en lisant sur un port *I/O* attaché au contrôleur.

Dans une telle approche, le processeur est occupé pendant la phase d'exécution et ne peut pas être utilisé pour réaliser d'autres traitements. La notion de **blocage-déblocage** ne peut pas être utilisée (*cfr.* Chapitre 3) et les ressources sont sous-utilisées. Cette approche est néanmoins pertinente lorsque le temps d'attente active est de courte durée et peut s'avérer plus efficace que les deux autres approches que nous allons voir.

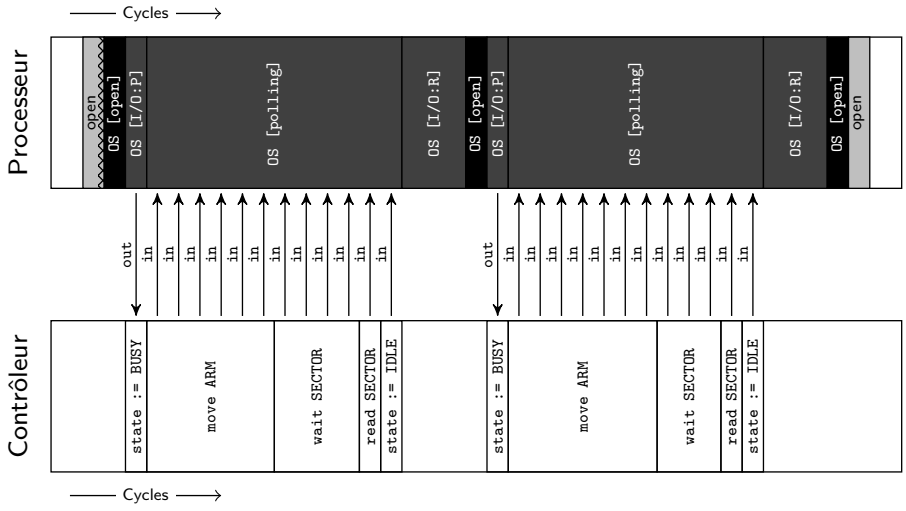


FIGURE 6.6 – Gestion d'une *programmed I/O*

La Figure 6.6 présente le canevas d'exécution de l'appel `open` par notre programme de la Figure 6.5a. Nous supposons que les clusters `0x09` et `0x0C` sont présents dans la cache du système de fichier et ne doivent donc pas être récupérés sur le disque. En pratique, le premier secteurs des clusters `0x47` et `0x54` doivent faire l'objet d'entrées-sorties. Lorsque l'appel `open()` est effectué par le processus, un basculement survient et le système d'exploitation commence à s'exécuter. Dans cette partie (*i.e.* `OS [open]`), le système de fichier va déterminer qu'il doit demander la lecture du premier secteur du cluster `0x47`. La phase de préparation (*i.e.* `OS [I/O:P]`) consiste à envoyer cet ordre en écrivant dans les registres du contrôleur (*i.e.* instruction `out`). À partir de ce point, le contrôleur va basculer à l'état `OCCUPÉ` et commencer à faire son travail (déplacement du bras, attente, lecture du secteur) et rebasculer à l'état `REPOS`. Pendant tout ce temps, le système d'exploitation sera dans une boucle d'attente active (*i.e.* `OS [polling]`) qui testera continuellement l'octet reçu sur le port où peut être lue la valeur de l'état du contrôleur. Au terme de cette attente active, le système d'exploitation transférera le secteur depuis le *buffer* du contrôleur vers la mémoire principale (*i.e.* `OS [I/O:R]`). À ce stade, il sera en mesure de consulter les entrées du répertoire encodées dans ce premier secteur (*i.e.* `OS [open`) pour déterminer qu'il est censé consulter le cluster `0x54`. Une nouvelle entrée-sortie sera réalisée suivant le même canevas pour récupérer le premier secteur du cluster `0x54`. Au terme de cette entrée-sortie, le système d'exploitation trouvera l'entrée correspondant au fichier dans laquelle il pourra identifier le numéro du premier cluster où commence le fichier demandé et la continuation du processus pourra se faire.

6.3.3 *Interrupt-driven I/O*

Une seconde approche nécessite de disposer d'un contrôleur qui peut générer une interruption au terme de la phase d'exécution. Dans ce cas, le système d'exploitation doit uniquement prendre en charge la préparation et la récupération. L'utilisation des interruptions rend possible le **blocage** du processus demandeur jusqu'à la conclusion du traitement de la demande d'ouverture. Cependant, la complexité de gestion est accrue en raison de leur discontinuité. Lors de la survenance d'une interruption il est nécessaire de pouvoir déterminer quel processus est effectivement concerné. De plus, si le processus demande une opération de lecture ou d'écriture, le système d'exploitation doit être prudent de ne pas évincer une des pages correspondant à l'espace de mémoire ciblé (*cfr.* Chapitre 2). Grâce à l'utilisation des interruptions, un recouvrement partiel des latences est possible de par le fait que le processeur peut être utilisé pour exécuter un autre processus pendant que l'entrée-sortie est en cours d'exécution sur le contrôleur. Cette approche est appropriée si la durée de la phase d'exécution dure longtemps.

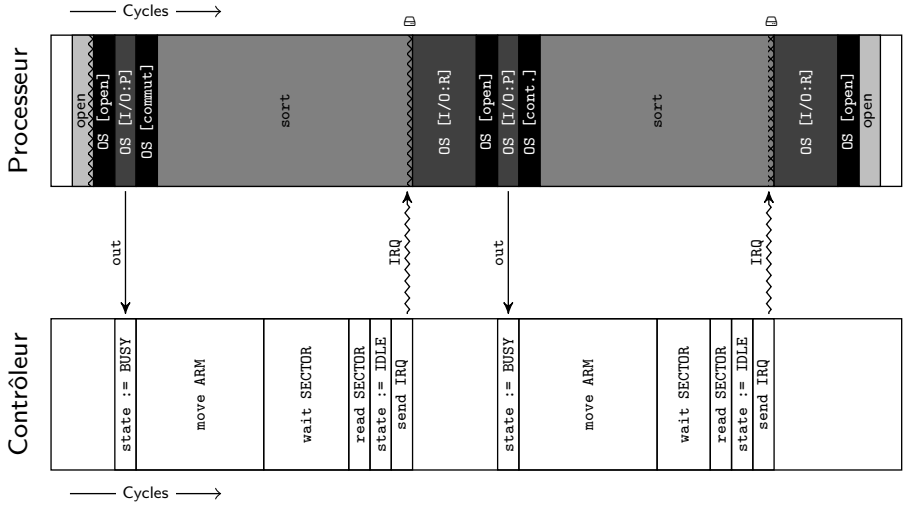


FIGURE 6.7 – Gestion d'une *interrupt-driven I/O*

Une fois la phase de préparation conclue (*i.e.* `OS[I/O:P]`), le système d'exploitation est libre de réaliser un ordonnancement qui va donner le processeur à un processus **prêt** (*e.g.* processus de triage ; `sort`) qui va pouvoir poursuivre son exécution. Une fois que le contrôleur conclut l'exécution et qu'il a récupéré le secteur demandé, celui-ci peut envoyer un signal d'interruption vers le processeur qui provoque un déroutement (*i.e.* `IRQ`). Le système d'exploitation prend dès lors en charge la récupération dans le cadre de cette entrée-sortie (*i.e.* `OS [I/O:R]`) pour consulter le secteur récupéré (premier du cluster `0x47`) et déterminer qu'il est nécessaire de récupérer le premier secteur du cluster `0x54` (*i.e.* `OS [open]`). Une nouvelle entrée-sortie sera préparée vers le contrôleur (*i.e.* `OS[I/O:P]`) et continuer le processus de triage (*i.e.* `sort`). Lorsque le contrôleur aura terminé son travail et généré l'interruption (*i.e.* `IRQ`), le système d'exploitation effectuera la récupération (*i.e.* `OS [I/O:R]`) et identifier le numéro du premier cluster du fichier demandé (*i.e.* `OS [open]`) et continuer le processus demandeur si sa priorité est supérieure à celui de triage.

6.3.4 *DMA-based I/O*

La dernière approche consiste à se reposer sur un contrôleur intermédiaire (*i.e. Direct Memory Access*) qui prendra en charge les trois phases vis-à-vis du contrôleur du périphérique concerné. En pratique, un contrôleur *DMA* dispose de plusieurs canaux pour pouvoir gérer plusieurs entrées-sorties en même temps. Le contrôleur *DMA* recevra l'interruption du contrôleur du périphérique concerné et génerera sa propre interruption vers le processeur une fois qu'il aura conclut la phase de récupération.

Le système d'exploitation devra effectuer une préparation de l'entrée-sortie au niveau du contrôleur *DMA* en spécifiant l'opération que ce dernier doit effectuer. Par exemple, copier le secteur 4544 vers une certaine adresse physique. La discontinuité nécessite de faire le lien entre l'occurrence d'une interruption *DMA* et le processus qui est concerné par cette dernière pour déterminer la suite des opérations.

Cette approche permet un recouvrement complet des latences étant donné que le système d'exploitation est entièrement déchargé des trois phase vis-à-vis du contrôleur concerné. Les ressources sont utilisées de façon optimale grâce à ce déchargement de responsabilités. Cette approche est appropriée si la durée de la phase d'exécution dure longtemps. Cependant, le contrôleur *DMA* n'a aucun connaissance du mécanisme de projection utilisé (*cfr.* Chapitre 2), ce qui signifie que le système d'exploitation doit être prudent de ne pas évincer une des pages correspondant à l'espace de mémoire ciblé (*cfr.* Chapitre 2). Cependant, des contentions peuvent apparaître au niveau du bus mémoire étant donné que tant le processeur que le contrôleur *DMA* peuvent tenter de réaliser un accès en même temps.

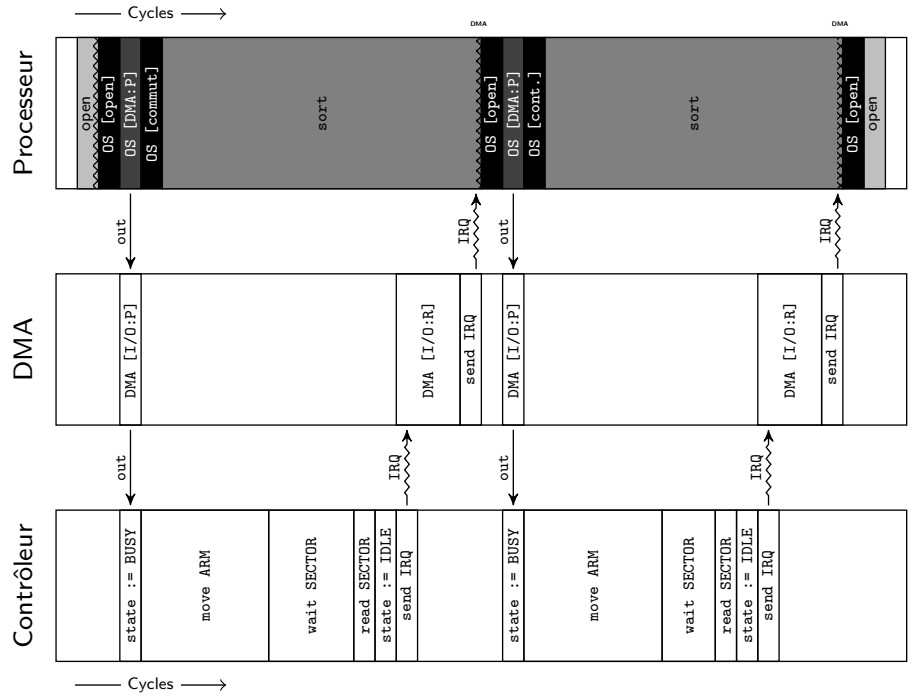


FIGURE 6.8 – Gestion d'une *DMA-based I/O*

Lorsque la demande d'ouverture est effectuée par le processus, le déroutement vers le système d'exploitation (*i.e.* `OS [open]`) va nécessiter de préparation une entrée-sortie vis-à-vis du *DMA*. Dans ce cadre, le système d'exploitation indiquera quel périphérique est concerné, quel secteur doit être lu et à quel endroit dans la mémoire il doit être placé (*i.e.* `OS [DMA:P]`). Une fois cette opération préparée, le système d'exploitation peut réaliser le **blocage** du processus demandeur et une **commutation** vers une processus **prêt** (*e.g.* processus de triage). Une fois que le contrôleur termine son travail, l'interruption est générée vers le *DMA* (*i.e.* `IRQ`) qui peut commencer la récupération (*i.e.* `DMA [I/O:R]`). Lorsque la récupération est conclue, une interruption du *DMA* est envoyée vers le processeur qui provoque un déroutement (*i.e.* `OS [open]`). Le système d'exploitation détermine qu'il est nécessaire de récupérer le premier secteur du cluster `0x54` pour lequel une nouvelle entrée-sortie est préparée au niveau du *DMA*. Au terme de celle-ci, le système d'exploitation déterminera enfin le numéro du premier cluster occupé par le fichier et permettra au processus de continuer.