Chapitre 5

Les systèmes de fichiers

5.1 Étude générale des systèmes de fichiers

5.1.1 Intérêt des fichiers

Toutes les applications informatiques doivent enregistrer et retrouver des informations. En effet, un processus en cours d'exécution peut enregistrer une quantité limitée d'informations dans son espace d'adressage ; la capacité de stockage est limitée par la taille de la mémoire vive ; cette taille peut convenir pour certaines applications, mais elle est beaucoup trop petite pour d'autres.

Le stockage des informations en mémoire vive possède un deuxième inconvénient : ces informations sont perdues lorsque le processus se termine, à cause de la technologie employée certes mais c'est comme ça.

Un troisième inconvénient est lié à l'accès simultané à ces informations. Un répertoire téléphonique stocké dans l'espace d'adressage d'un processus ne peut être examiné que par ce seul processus de telle sorte qu'on ne peut rechercher qu'un seul numéro de téléphone à la fois. Pour résoudre ce problème, il faut rendre l'information indépendante d'un processus donné.

Trois caractères sont donc requis pour stocker des informations à long terme :

1. Il faut pouvoir stocker des informations de très grand taille.
2. Les informations ne doivent pas disparaître lorsque le processus les utilisant se termine.
3. Plusieurs processus doivent pouvoir accéder simultanément aux informations.

La solution à tous ces problèmes consiste à stocker les informations dans des **fichiers** (*file* en anglais), sur des disques ou sur d'autres supports. Les processus peuvent alors les lire ou en écrire de nouvelles. Les informations stockées dans des fichiers doivent être *permanentes*, c'est-à-dire non affectées par la création ou la fin d'un processus. Un fichier ne doit disparaître que lorsque son propriétaire le supprime explicitement.

Les fichiers sont gérés par le système d'exploitation. La façon dont ils sont structurés, nommés, utilisés, protégés et implémentés sont des points majeurs de la conception du système d'exploitation. La partie du système d'exploitation gérant les fichiers est appelée **système de fichiers** (en anglais *file system*).

L'utilisateur attache la plus grande importance à l'**interface d'un système de fichiers**, c'est-à-dire à la manière de nommer et de protéger les fichiers, aux opérations permises sur eux, etc. Il est moins important pour lui de connaître le nombre de secteurs d'un bloc logique ou de savoir si on utilise des listes chaînées ou des tables de bits pour mémoriser les emplacements libres. Ces points sont, en revanche, fondamentaux pour le concepteur du système de fichiers.

5.1.2 Interface utilisateur

5.1.2.1 Accès aux fichiers

Syntaxe des noms de fichiers

Les fichiers ressortent d'un mécanisme abstrait. Ils permettent d'écrire des informations sur le disque et de les lire ultérieurement. Ceci doit être fait de manière à masquer à l'utilisateur le fonctionnement et l'emplacement de stockage des informations : il ne doit pas avoir à choisir tel ou tel secteur, par exemple. *La gestion et la syntaxe des noms des objets sont les parties les plus importantes d'un mécanisme abstrait.* Le processus qui crée un fichier lui attribue un nom. Lorsque le processus se termine, le fichier existe toujours et un autre processus peut y accéder au moyen de ce nom.

Les règles syntaxiques de formation des noms de fichiers varient d'un système d'exploitation à un autre, mais tous autorisent les noms de fichiers constitués de chaînes de un à huit caractères non accentués. Ainsi 'pierre' et 'agnes' sont des noms de fichiers valides.

Les chiffres et des caractères spéciaux sont quelquefois autorisés. Ainsi '2', 'urgent !' et 'Fig.2-14' peuvent être des noms valides dans certains systèmes d'exploitation.

Certains systèmes de fichiers font la différence entre les lettres majuscules et les minuscules correspondantes alors que d'autres ne le font pas. UNIX fait partie de la première catégorie et MS-DOS de la seconde. Les noms suivants désignent donc des fichiers distincts sur un système UNIX : 'barbara', 'Barbara' et 'BARBARA'. Sur MS-DOS, ils désignent le même fichier.

De nombreux systèmes d'exploitation gèrent des noms en deux parties, les deux parties étant séparées par un point, comme dans 'prog.c'. La partie qui suit le point est alors appelée **extension** et donne en général une indication sur la nature du fichier. Sous MS-DOS, par exemple, les noms de fichiers comportent 1 à 8 caractères suivis éventuellement d'une extension de 1 à 3 caractères. Sous UNIX, la taille de l'extension éventuelle est libre, le fichier pouvant même avoir plus d'une extension comme dans 'prog.c.Z'.

Dans certains cas, les extensions sont simplement des conventions et ne sont pas contrôlées. Un fichier 'fichier.txt' est vraisemblablement un fichier texte, mais ce nom est destiné davantage au propriétaire du fichier qu'au système d'exploitation. En revanche, un compilateur C peut imposer l'extension '.c' à tous les fichiers à compiler.

Fichiers d'octets

Les fichiers peuvent être structurés de différentes manières. Un fichier peut être une simple suite d'octets. Dans ce cas, le système d'exploitation ne connaît pas et ne s'occupe pas du contenu du fichier. Il ne gère que des octets. La signification est donnée par les programmes utilisateurs. Les fichiers UNIX et MS-DOS, tout au moins depuis 2.0, sont structurés ainsi.

Cette approche offre la plus grande souplesse. Les programmes utilisateurs peuvent écrire ce qu'ils souhaitent dans les fichiers et les nommer à leur convenance. Le système d'exploitation ne fournit pas d'aide et n'impose pas de restriction. Ce dernier point est très important pour les utilisateurs souhaitant effectuer des opérations peu courantes.

Types de fichiers

De nombreux systèmes d'exploitation possèdent différents types de fichiers. UNIX et MS-DOS, par exemple, ont des *fichiers ordinaires*, des *catalogues*, ou *répertoires*, et des *fichiers spéciaux*. La connaissance de ces types est surtout utile au système d'exploitation : les **fichiers ordinaires** contiennent les informations des utilisateurs ; les **catalogues** (en anglais *directories*), ou **répertoires** (en anglais *folders*), sont des fichiers système servant à la structure du système de fichiers ; les **fichiers spéciaux** permettent de modéliser les périphériques d'entrées-sorties comme des fichiers.

Les fichiers ordinaires sont en général des *fichiers texte* ou des *fichiers binaires*. La différence concerne leur comportement à l'égard d'un éditeur de texte. Les **fichiers texte** contiennent des lignes de texte. Dans certains systèmes, chaque ligne est terminée par le caractère (ASCII) « retour chariot ». Dans d'autres, le caractère « passage à la ligne » est utilisé. Parfois les deux sont nécessaires. La longueur d'une ligne n'est pas fixe. Le grand avantage des fichiers texte est qu'ils peuvent être affichés et imprimés sans modification et qu'ils peuvent être édités au moyen d'un éditeur de texte standard.

Les autres fichiers sont, par définition, des **fichiers binaires**, ce qui signifie tout simplement que ce ne sont pas des fichiers texte. Leur affichage à l'écran ou leur impression donne une suite incompréhensible de signes. Ces fichiers ont en général une structure interne, dépendant de l'application qui les a engendrés. Tous les systèmes d'exploitation doivent reconnaître au moins un type de fichiers binaires, leurs propres fichiers exécutables.

Méthodes d'accès aux fichiers

Il existe deux façons d'accéder aux fichiers.

Accès séquentiel.- Les premiers systèmes d'exploitation n'offraient qu'un seul type d'accès aux fichiers : l'**accès séquentiel**. Dans ce cas, un processus peut lire tous les octets d'un fichier dans l'ordre à partir du début du fichier. Les fichiers séquentiels peuvent cependant être « rembobinés » (on garde l'image d'une bande magnétique) et donc être lus autant de fois que nécessaire. Cet accès séquentiel est bien adapté lorsque le support de stockage est une bande magnétique mais peu adapté lorsqu'il s'agit d'un disque.

Accès direct.- L'accès des disques a autorisé la lecture d'octets dans le désordre. Les fichiers dont les octets peuvent être lus dans un ordre quelconque sont appelés **fichiers à accès direct** (en anglais *random access files*).

Les fichiers à accès direct sont indispensables à de nombreuses applications, par exemple les systèmes de gestion de bases de données : si un client d'une compagnie aérienne veut réserver une place sur un vol particulier, le programme de réservation doit pouvoir accéder aux enregistrements de ce vol sans avoir à parcourir les enregistrements de milliers d'autres vols.

Deux méthodes permettent de spécifier la position de départ de la lecture, par exemple. Dans la première, chaque opération de lecteure indique la position dans le fichier à laquelle la lecture doit débuter. Dans la deuxième, une opération spéciale, `SEEK`, permet de se positionner à un endroit donné. À la suite de ce positionnement, la lecture peut débuter à partir de cette nouvelle position courante.

Certains anciens systèmes d'exploitation de grands ordinateurs classaient les fichiers lors de leur création dans deux catégories : séquentiels ou à accès direct. Ceci permettait au système d'exploitation d'utiliser des méthodes de stockage différentes pour chacune de ces catégories. Les systèmes d'exploitation modernes n'effectuent plus cette distinction. Tous leurs fichiers sont automatiquement à accès direct.

Attributs des fichiers

Tout fichier possède un nom et des données. De plus, tous les systèmes d'exploitation associent des informations complémentaires à chaque fichier, par exemple la date et l'heure de création du fichier ainsi que sa taille. On appelle **attributs** du fichier ces informations complémentaires. La liste des attributs varie considérablement d'un système à l'autre.

La table suivante donne quelques attributs possibles :

Champ	Signification
Protection	Spécifie qui peut accéder au fichier et de quelle façon
Mot de passe	Mot de passe requis pour accéder au fichier
Créateur	Personne ayant créé le fichier
Propriétaire	Propriétaire actuel
Lecture seule	Indicateur : 0 pour lecture-écriture, 1 pour lecture seule
Fichier caché	Indicateur : 0 pour fichier normal, 1 pour ne pas afficher dans les listes
Fichier système	Indicateur : 0 pour fichier normal, 1 pour fichier système
À archiver	Indicateur : 0 si la version a déjà été archivée, 1 si à archiver
Texte/binaire	Indicateur : 0 pour fichier texte, 1 pour fichier binaire
Accès direct	Indicateur : 0 pour accès séquentiel, 1 pour accès direct
Fichier temporaire	Indicateur : 0 pour fichier normal, 1 pour suppression du fichier lorsque le processus se termine
Verrouillage	Indicateur : 0 pour fichier non verrouillé, 1 pour fichier verrouillé
Date de création	Date et heure de création du fichier
Date du dernier accès	Date et heure du dernier accès au fichier
Date modification	Date et heure de la dernière modification
Taille actuelle	Nombre d'octets du fichier
Taille maximale	Taille maximale autorisée pour le fichier

Les quatre premiers attributs sont liés à la protection du fichier et spécifient les personnes pouvant y accéder. Certains systèmes demandent à l'utilisateur d'entrer un mot de passe avant de pouvoir accéder à un fichier. Dans ce cas, le mot de passe (crypté) est un champ du fichier.

Les **indicateurs** sont des bits ou des petits champs qui contrôlent ou autorisent certaines propriétés. Les **fichiers cachés**, par exemple, n'apparaissent pas dans le listage de tous les fichiers. L'**indicateur d'archivage** est un bit qui indique si le fichier a été modifié depuis la dernière sauvegarde du disque ; si c'est le cas, le fichier sera sauvegardé lors de la prochaine sauvegarde et le bit remis à zéro. L'**indicateur temporaire** autorise la suppression automatique du fichier lorsque le processus qui l'a créé se termine.

Les différentes dates mémorisent les date et heure de création, de dernier accès et de dernière modification. Elles sont utiles dans de nombreux cas. Par exemple, un fichier source modifié après la création de l'objet correspondant doit être recompilé.

La *taille actuelle* indique la taille actuelle du fichier.

Opérations sur les fichiers

Les fichiers permettent de stocker des informations et de les rechercher ultérieurement. Les systèmes d'exploitation fournissent les moyens pour réaliser le stockage et la recherche. Les principaux appels système relatifs aux fichiers sont les suivants :

— 1. `CREATE`. Le fichier est créé sans données. Cet appel a pour but d'indiquer la création du fichier et de fixer un certain nombre de paramètres.

— 2. `DELETE`. Le fichier devenu inutile est supprimé pour libérer de l'espace sur le disque. Il existe toujours un appel système à cette fin.

— 3. `OPEN`. Un fichier doit être ouvert avant qu'un processus puisse l'utiliser. L'appel `OPEN` permet au système de charger en mémoire les attributs et la liste des adresses du fichier sur le disque afin d'accélérer les accès ultérieurs.

— 4. `CLOSE`. Lorsqu'il n'y a plus d'accès au fichier, les attributs et la liste des adresses du fichier ne sont plus nécessaires en mémoire vive. Le fichier doit être fermé pour libérer de l'espace dans les tables internes. De nombreux systèmes incitent les utilisateurs à fermer les fichiers en imposant un nombre maximal de fichiers ouverts par processus.

— 5. `READ`. Des données sont lues à partir du fichier. En général, les octets sont lus à partir de la position en cours. L'appelant doit spécifier le nombre d'octets demandés ainsi que l'emplacement d'une mémoire tampon de réception.

— 6. `WRITE`. Des données sont écrites dans le fichier à partir de la position en cours. Si la position en cours est située à la fin du fichier, la taille du fichier augmente. Si la position courante est à l'intérieur du fichier, les anciennes données sont remplacées et définitivement perdues.

— 7. `APPEND`. Cet appel est une version restreinte de `WRITE`, ajoutant des données à la fin du fichier. Les systèmes d'exploitation n'ayant qu'un nombre restreint d'appels système ne disposent pas en général de l'appel `APPEND`. Il existe souvent plusieurs manières pour effectuer une opération de sorte que ces systèmes permettent néanmoins d'effectuer un ajout en fin de fichier.

— 8. `SEEK`. Pour les fichiers à accès direct, il faut indiquer la position des données à lire ou à écrire. L'appel système `SEEK`, souvent utilisé, modifie la position en cours dans le fichier. À la suite de cet appel, les données sont lues ou écrites à partir de la nouvelle position en cours.

— 9. `GET ATTRIBUTES`. Les processus doivent souvent lire les attributs des fichiers pour effectuer certaines opérations. Il leur faut donc pouvoir accéder aux attributs.

— 10. `SET ATTRIBUTES`. Quelques attributs peuvent être modifiés par les utilisateurs et peuvent être renseignés après la création du fichier. Cet appel système réalise cette opération. Les informations relatives à la protection constituent un exemple évident d'attributs modifiables (par les personnes permises). La plupart des indicateurs le sont également.

— 11. `RENAME`. Il est fréquent de vouloir modifier le nom d'un fichier. Cet appel système réalise cette opération. Il n'est pas vraiment indispensable puisqu'un fichier peut toujours être copié dans un fichier qui porte le nouveau nom et l'ancien fichier supprimé, ce qui suffit en général.

5.1.2.2 Accès aux répertoires

Le système mémorise les noms, attributs et adresses des fichiers dans des *répertoires*. Ce sont eux-mêmes, dans beaucoup de systèmes d'exploitation, des fichiers.

Un répertoire contient un certain nombre d'entrées, une par fichier. Chaque entrée contient, par exemple, le nom du fichier, ses attributs et les adresses sur le disque à lesquelles sont stockées les données. Une entrée peut, dans un autre système d'exploitation, contenir le nom du fichier et un pointeur sur une structure contenant les attributs et les adresses sur le disque.

Lorsque l'ouverture d'un fichier est demandé, le système d'exploitation recherche le nom du fichier à ouvrir dans le répertoire. Il extrait alors les attributs et les adresses sur le disque, soit directement à partir de l'entrée du répertoire, soit à partir de la structure de données sur laquelle pointe l'entrée du répertoire. Toutes les références ultérieures au fichier utilisent ces informations alors présentes en mémoire.

Répertoire unique.- Le nombre de répertoires varie d'un système à un autre. La méthode la plus simple, du point de vue de la conception, consiste à garder la trace de tous les fichiers des utilisateurs dans un seul répertoire. Cependant, s'il y a plusieurs utilisateurs et s'ils choisissent des noms de fichiers identiques, le système devient très vite inutilisable. Ce modèle n'est utilisé que par les systèmes d'exploitation très simples de certains micro-ordinateurs ; c'est le cas de la première version de MS-DOS.

Un répertoire par utilisateur.- On peut améliorer le modèle du répertoire unique en attribuant un répertoire à chaque utilisateur, sur un système d'exploitation multi-utilisateurs. Cette possibilité élimine les conflits concernant les noms entre les utilisateurs mais elle n'est pas très adaptée lorsque les utilisateurs possèdent beaucoup de fichiers. Par ailleurs, les utilisateurs souhaitent souvent regrouper leurs fichiers de manière logique.

Arborescence.- Une organisation hiérarchique constituée d'une arborescence de répertoires permet à chaque utilisateur d'avoir autant de répertoires qu'il le souhaite et de regrouper les fichiers logiquement.

5.1.3 Mise en oeuvre des systèmes de fichiers

Examinons maintenant le système de fichiers du point de vue du concepteur. Les utilisateurs se préoccupent des noms des fichiers, des opérations permettant de les manipuler, de l'arborescence des fichiers, etc. Les concepteurs portent davantage leur attention sur l'organisation de l'espace du disque et sur la manière dont les fichiers et les catalogues sont sauvegardés. Ils recherchent un fonctionnement efficace et fiable.

5.1.3.1 Stockage des fichiers

Le principe fondamental de stockage d'un fichier est de mémoriser l'adresse des blocs constituant le fichier. Différentes méthodes sont utilisées pour cela.

Allocation contiguë.- La méthode d'allocation la plus simple consiste à stocker chaque fichier dans une suite de blocs consécutifs. Un fichier de 50 kiO, par exemple, occupera 50 blocs consécutifs sur un disque dont la taille des blocs est 1 kiO.

Cette méthode a deux avantages importants. Premièrement, elle est simple à mettre en œuvre puisqu'il suffit de mémoriser un nombre, l'adresse du premier bloc, pour localiser le fichier. Deuxièmement les performances sont excellentes puisque tout le fichier peut être lu en une seule opération. Aucune autre méthode d'allocation ne peut l'égaler.

Malheureusement, l'allocation contiguë présente également deux inconvénients importants. Premièrement, elle ne peut être mise en œuvre que si la taille maximum du fichier est connue au moment de la création de celui-ci. Sans cette information, le système d'exploitation ne peut pas déterminer l'espace à réserver sur le disque. Dans les systèmes où les fichiers doivent être écrits en une seule opération, elle peut néanmoins être avantageusement utilisée.

Le deuxième inconvénient est la *fragmentation* du disque découlant de cette politique d'allocation. Elle gaspille de l'espace sur le disque. Le compactage du disque peut y remédier mais il est en général coûteux. Il peut cependant être réalisé la nuit lorsque le système n'est pas chargé.

Allocation par liste chaînée.- La deuxième méthode consiste à sauvegarder les blocs des fichiers dans une liste chaînée. Le premier mot de chaque bloc, par exemple, est un pointeur sur le bloc suivant ou l'indication qu'il s'agit du dernier bloc du fichier. Le reste du bloc contient les données.

Contrairement à l'allocation contiguë, tous les blocs peuvent être utilisés. Il n'y a pas d'espace perdu en raison d'une fragmentation du disque. L'entrée du répertoire stocke simplement l'adresse du premier bloc. On trouve les autres blocs à partir de celui-ci.

Il existe également des inconvénients. Si la lecture séquentielle d'un fichier est simple, l'accès direct est extrêmement lent. Par ailleurs, le pointeur sur le bloc suivant occupant quelques octets, l'espace réservé aux données dans chaque bloc n'est plus une puissance de deux. Ceci est moins efficace car de nombreux programmes lisent et écrivent des blocs dont la taille est une puissance de deux.

Allocation par liste chaînée indexée.- Les inconvénients de l'allocation au moyen d'une liste chaînée peuvent être éliminés en faisant passer le pointeur de chaque bloc à une table ou en index en mémoire.

Cette méthode libère intégralement l'espace du bloc pour les données. Elle facilite également les accès directs : la liste doit toujours être parcourue pour trouver un déplacement donné dans le fichier, mais elle réside entièrement en mémoire et peut être parcourue sans accéder au disque. Comme pour la méthode précédente, l'entrée du répertoire contient un seul entier (le numéro du premier bloc) permettant de retrouver tous es autres blocs, quelle que soit la taille du fichier.

MS-DOS utilise cette méthode.

Le principal inconvénient de cette méthode vient du fait que la table doit être entièrement en mémoire en permanence. Un grand disque de 500 000 blocs requiert 500 000 entrées dans la table, occupant chacune au minimum 3 octets. Pour accélérer la recherche, la taille des entrées devrait être de 4 octets. La table occupera donc 1,5 MiO si le système est optimisé pour l'espace et 2 MiO s'il est optimisé pour l'occupation mémoire.

Nœuds d'information.- La quatrième méthode de mémorisation des blocs de chaque fichier consiste à associer à chaque fichier une petite table, appelée **nœud d'information** (ou *i-node*). Cette table contient sur le disque les attributs et les adresses concernant les blocs du fichier.

Les premières adresses disque sont contenues dans le nœud d'information de sorte que les informations des petits fichiers y sont entièrement contenues lorsqu'il est chargé en mémoire à l'ouverture du fichier. Pour les fichiers plus importants, une des adresses du nœud d'information est celle d'un bloc du disque appelé **bloc d'indirection simple**. Ce bloc contient des adresses disque additionnelles. Si cela ne suffit pas encore, une autre adresse du nœud d'information, appelée **bloc d'indirection double**, contient l'adresse d'un bloc contenant une liste de blocs à simple indirection. Chaque bloc d'indirection simple pointe sur quelques centaines de blocs de données. Il existe également des blocs d'indirection triple.

UNIX utilise cette méthode.

5.1.3.2 Mise en oeuvre des répertoires

Il faut ouvrir un fichier avant de pouvoir le lire. Quand on ouvre un fichier, le système d'exploitation utilise le chemin d'accès donné par l'utilisateur pour localiser l'entrée dans le répertoire. Cette entrée fournit les informations nécessaires pour retrouver les blocs sur le disque. En fonction du système, ces informations peuvent être les adresses disque de tout le fichier (cas d'une allocation contiguë), le numéro du premier bloc (cas des deux méthodes d'allocation par liste chaînée) ou le numéro du nœud d'information. Dans tous les cas, la fonction principale du système est d'établir la correspondance entre les chemins d'accès et les informations requises pour trouver les données.

L'emplacement de stockage des attributs est lié au point précédent. Beaucoup de systèmes les stockent directement dans les entrées des catalogues. Les systèmes utilisant des nœuds d'information peuvent les placer dans ces derniers.

Les répertoires de CP/M.- Commençons par un système simple, en l'occurrence CP/M (voir [G-D-86]). Dans ce système, il n'y a qu'un seul répertoire, que le système de fichiers doit donc parcourir pour rechercher un fichier donné. L'entrée des fichiers dans ce répertoire unique contient les numéros des blocs des fichiers ainsi que tous les attributs. Si le fichier doit utiliser plus de blocs que l'entrée ne peut en contenir, le système de fichiers lui alloue une entrée supplémentaire.

Les champs sont les suivants. Le champ *code utilisateur* contient l'identité du propriétaire du fichier. On n'examine pendant une recherche que les entrées qui appartiennent à l'utilisateur connecté. Les deux champs suivants fournissent le *nom* et le *type* du fichier (ce dernier étant devenu *extension* dans le système MS-DOS). Le champ suivant, *extension*, est nécessaire pour les fichiers de plus de 16 blocs, parce qu'ils occupent plus d'une entrée dans le répertoire. Il permet de déterminer l'ordre des différentes entrées. Le champ *entrées utilisées* indique le nombre d'entrées utilisées parmi les 16 entrées de blocs potentielles. Les 16 derniers champs contiennent les numéros des blocs du fichier. Le dernier bloc pouvant ne pas être utilisé complètemet, le système ne peut pas connaître la taille exacte d'un fichier à l'octet près ; il mémorise donc les tailles en nombre de blocs et non en octets.

Les répertoires de MS-DOS.- Le système d'exploitation MS-DOS, tout au moins à partir de MS-DOS 2, possède une arborescence de répertoires. La figure (Tanenbaum, p. 186) montre la structure d'une entrée dans un catalogue MS-DOS. Elle est constituée de 32 octets et contient notamment le nom et le numéro du premier bloc du fichier. Ce numéro sert d'index dans une table. En parcourant la chaîne, on peut trouver tous les blocs d'un fichier.

Les répertoires de MS-DOS peuvent contenir d'autres répertoires, ce qui donne un système de fichiers hiérarchiques.

Les répertoires d'UNIX.- La structure d'un répertoire d'UNIX est simple, comme le montre la figure (Tanenbaum, p. 186). Chaque entrée contient un nom de fichier et le numéro de son nœud d'information. Toutes les informations concernant le fichier (son type, sa taille, les dates de dernière modification et de dernière consultation, l'identité de son propriétaire, les blocs qu'il occupe sur le disque) sont contenues dans le nœud d'information.

5.1.3.3 L'organisation de l'espace disque

Les fichiers étant habituellement sauvegardés sur des disques, l'organisation de l'espace disque est primordiale pour les concepteurs de systèmes de fichiers.

Espace contigu ou blocs.- Il existe deux stratégies pour stocker un fichier de n octets : on alloue n octets consécutifs sur le disque ou on divise le fichier en plusieurs blocs (pas nécessairement contigus).

Si on sauvegarde un fichier sur un nombre contigu d'octets, on doit le déplacer chaque fois que sa taille augmente (ce qui arrive fréquemment). La plupart des systèmes de fichiers préfèrent donc stocker les fichiers dans plusieurs blocs de taille fixe pas nécessairement adjacents.

Taille des blocs.- Ce choix étant fait, quelle est alors la taille optimale d'un bloc ? On peut s'inspirer de l'organisation physique du disque en donnant aux blocs la taille d'un secteur, d'une piste ou d'un cylindre. Pour les systèmes paginés, on peut penser aussi à la taille des pages.

Une unité d'allocation de grande taille, telle que le cylindre, signifie que tout fichier, même s'il ne fait qu'un octet, occupe un cylindre entier. Si, en revanche, l'unité d'allocation est de petite taille, chaque fichier sera constitué de nombreux blocs. Or, la lecture d'un bloc requiert du temps (délais de recherche et de rotation du disque) ; la lecture d'un fichier constitué de plusieurs blocs sera donc lente.

Le compromis habituellement adopté consiste à prendre des blocs de 512 octets, 1 kiO ou 2 kiO. Si on prend des blocs de 1 kiO sur un disque dont les secteurs font 512 octets, le système de fichiers lit et écrit deux secteurs consécutifs en les considérant comme un ensemble unique et indivisible.

Mémorisation des blocs libres.- Dès qu'on a choisi la taille des blocs, on doit trouver un moyen de mémoriser les blocs libres. Les deux méthodes les plus répandues sont représentées sur la figure ci-dessous (Tanenbaum, p. 192).

La première consiste à utiliser une liste chaînée de blocs du disque, chaque bloc contenant des numéros de blocs libres. Si les blocs font 1 kiO et les numéros de blocs 16 bits, chaque bloc de la liste contient les numéros de 511 blocs libres. Un disque de 20 MiO a donc besoin d'une liste de 40 blocs au plus pour contenir tous les numéros de blocs libres (20 kiO). Les blocs libres sont souvent utilisés pour mémoriser la liste des blocs libres.

La deuxième technique de gestion des espaces libres a recours à une table de bits. Un disque de n blocs requiert une table de n bits. Les blocs libres sont représentés par des 1 dans la table et les blocs occupés par des 0 (ou vice versa). La table d'un disque de 20 MiO fait 20 kilobits et n'utilise que 3 blocs. Il n'est pas étonnant que la table de bits requière moins de place que la liste chaînée puisqu'elle n'utilise qu'un bit par bloc, alors que la liste chaînée en utilise 16. Ce n'est que si le disque est pratiquement plein que la liste chaînée occupe moins de blocs que la table de bits.

Si la taille de la mémoire principale est suffisante pour contenir entièrement la table de bits, cette méthode de stockage est préférable. Si, en revanche, on ne dispose que d'un seul bloc en mémoire principale pour mémoriser les blocs libres et si le disque est presque plein, la liste chaînée peut s'avérer meilleure. En effet, si on n'a en mémoire qu'un seul bloc de la table de bits, il peut arriver que ce bloc ne contienne aucun bloc libre. Il faut alors lire sur le disque le reste de la table de bits. En revanche, quand on charge un bloc de la liste chaînée, on peut allouer 511 blocs avant d'avoir à accéder à nouveau au disque.

5.1.3.4 La fiabilité du système de fichiers

La destruction d'un système de fichiers est souvent plus grave que la destruction de l'ordinateur. Un ordinateur endommagé par le feu, la foudre ou par une tasse de café renversée sur le clavier peut être remplacé. Les ordinateurs personnels peu coûteux peuvent même être remplacés en quelques heures : il suffit de se rendre chez le plus proche revendeur et d'en acheter un nouveau.

Si le système de fichiers d'un ordinateur est détruit à cause d'une panne matérielle ou d'une erreur logicielle, la restauration des informations est une opération difficile, longue et, dans bien des cas, impossible. Les conséquences peuvent être catastrophiques si les documents, les fichiers, les enregistrements ou les données sont importants. Si le système de fichiers ne peut pas protéger les équipements et les supports contre une destruction physique, il peut, en revanche, protéger les informations. Nous allons donc examiner quelques méthodes de protection des systèmes de fichiers.

La gestion des blocs endommagés.- Les disques ont souvent des blocs endommagés. Les disquettes sont en général exemptes de défauts lorsqu'elles quittent l'usine, mais elles peuvent présenter des signes d'usure par la suite. Les disques durs ont souvent des blocs défectueux d'origine : il serait en effet trop coûteux de les fabriquer sans défauts. En fait la plupart des fabricants fournissent avec chaque disque la liste des blocs endommagés que leurs tests ont révélés.

Il existe deux solutions au problème des blocs endommagés : l'une est matérielle et l'autre logicielle. La solution matérielle consiste à réserver une piste du disque à la liste des blocs endommagés. Le contrôleur lit cette liste au moment de son initialisation et remplace les blocs endommagés par un bloc de cette liste tout en enregistrant la correspondance dans la liste. Toute requête à un bloc endommagé accéde en fait au bloc de remplacement.

La solution logicielle implique que l'utilisateur ou le système de fichiers construise avec précaution un fichier contenant tous les blocs endommagés. Cette technique retire donc les blocs endommagés de la liste des blocs libres, évitant ainsi leur utilisation future. Il n'y a pas de problème tant qu'on ne cherche pas à accéder au fichier des blocs endommagés. Il faut éviter de lire ce fichier au cours des sauvegardes du disque.

Les sauvegardes.- Même si la méthode de traitement des blocs endommagés est performante, il faut régulièrement effectuer des copies de sauvegarde des fichiers.

Les systèmes de fichiers d'une disquette peuvent être sauvegardés en dupliquant simplement la disquette. Les systèmes de fichiers des petits disques durs peuvent être sauvegardés en copiant (en anglais *to dump*, vider) tout le disque sur une bande magnétique.

La sauvegarde des disques durs de forte capacité est malaisée et requiert beaucoup de temps. Une stratégie facile à mettre en œuvre consiste à livrer deux disques avec tous les ordinateurs. Ces disques sont divisés en deux moitiés : données et sauvegarde. Chaque nuit, les données du disque 0 sont copiées dans la zone de sauvegarde du disque 1 et vice versa. De cette façon, même si un disque est complètement détruit, on ne perd pas d'informations. Malheureusement, cette technique fait perdre la moitié de l'espace du disque.

La **sauvegarde incrémentale** est une alternative à la **sauvegarde quotidienne** globale de tout le système de fichiers que nous venons de décrire. Sous sa forme la plus simple, la sauvegarde incrémentale est équivalente à une **sauvegarde globale** périodique, toutes les semaines ou tous les mois, doublée d'une sauvegarde quotidienne des seuls fichiers modifiés depuis la dernière sauvegarde globale. Une meilleure méthode consiste à sauvegarder uniquement les fichiers modifiés depuis la dernière sauvegarde.

Pour mettre en œuvre cette méthode, il faut conserver sur le disque la liste des dates de sauvegarde des fichiers. Le programme de sauvegarde examine alors chaque fichier du disque. Si

le fichier a été modifié depuis la dernière sauvegarde, il le sauvegarde en mettant à jour la date de sauvegarde. Si le cycle est mensuel, cette méthode requiert 31 bandes pour les sauvegardes quotidiennes et des bandes pour la sauvegarde globale mensuelle.

MS-DOS fournit une aide à la mise en œuvre des sauvegardes. Il associe à chaque fichier un bit appelé *bit d'archivage*. Au moment de la sauvegarde du système de fichiers, les bits d'archivage de tous les fichiers sont mis à zéro. Par la suite, à chaque fois qu'un fichier est modifié, le système d'exploitation positionne automatiquement son bit d'archivage. Lors de la sauvegarde suivante, le programme de sauvegarde teste tous les bits d'archivage et ne sauvegarde que les fichiers dont le bit est positionné. Puis il remet tous les bits d'archivage à zéro.

La cohérence du système de fichiers.- La cohérence du système de fichiers est un élément important de la fiabilité. De nombreux systèmes de fichiers lisent des blocs, les modifient et les réécrivent ensuite. Si le système tombe en panne avant la réécriture des blocs, le système de fichiers peut se retrouver dans un état incohérent. C'est d'autant plus critique lorsque les blocs sont des blocs de nœuds d'information, de répertoires ou des blocs qui contiennent la liste des blocs libres.

La plupart des ordinateurs ont un programme utilitaire qui vérifie la cohérence du système de fichiers. Ce programme peut être exécuté à chaque démarrage du système, surtout à la suite d'un arrêt forcé.

Protection contre les erreurs de l'utilisateur.- Nous avons vu dans les paragraphes précédents les moyens de protéger les données des utilisateurs contre les pannes du système. Il faut aussi quelquefois protéger l'utilisateur contre ses propres erreurs. Si, au lieu de taper :

```
rm *.o
```

pour supprimer les fichiers se terminant par `.o` (fichiers objets générés par le compilateur), il tape par erreur :

```
rm * .o
```

`rm` supprimera tous les fichiers du répertoire courant, puis indiquera qu'il n'arrive pas à trouver `.o`. Dans MS-DOS et quelques autres systèmes, on positionne simplement un bit dans le répertoire ou dans le nœud d'information pour indiquer qu'un fichier est détruit. On ne placera les blocs du fichier dans la liste des blocs libres que lorsqu'on en aura besoin. Ainsi, si l'utilisateur s'aperçoit immédiatement de son erreur, il peut exécuter un programme utilitaire spécial qui restaure les fichiers détruits.

5.2 Cas de MS-DOS

Le système de fichiers de MS-DOS 1.0 est calqué sur celui de CP/M, ne disposant que d'un seul répertoire et utilisant des *blocs de contrôle de fichiers* (FCB) pour effectuer les entrées-sorties. Avec MS-DOS 2.0 apparaît un système de fichiers comportant des *descripteurs de fichiers* et des appels système inspirés d'UNIX. Les anciens appels de style CP/M tombent alors en désuétude et finissent par être considérés comme totalement obsolètes.

5.2.1 Organisation physique de l'espace disque

Nous avons vu antérieurement, à propos de la programmation avec le BIOS, comment un disque est vu physiquement. L'un des rôles du système d'exploitation est de pouvoir accéder au contenu du disque de façon un peu plus conviviale. Nous allons voir maintenant comment fait le système d'exploitation pour cela.

5.2.1.1 Rappels sur la description physique

Nous avons déjà vu l'organisation des disques à propos de la programmation avec le BIOS. Rappelons-la !

Modèle.- La partie importante d'une **disquette** est un disque, en fait une couronne circulaire, qui a deux **faces**. Un **disque dur** est un ensemble de disques de même axe, ayant chacun deux faces.

Piste.- Chaque face contient un certain nombre de **pistes**, cercles du même axe que le disque. Ces pistes sont d'une certaine façon imaginaires puisqu'elles ne sont pas matérialisées. Les pistes sont numérotées à partir du numéro 0, correspondant à la piste la plus externe.

Secteur.- Chaque piste est divisée en un certain nombre de **secteurs**, correspondant à un angle donné. Cet angle est $2\pi/n$, où n est un entier naturel non nul ; une piste a une longueur d'autant plus courte qu'elle est plus proche du centre. Là encore un secteur n'est pas matérialisé. Un secteur contient un nombre fixé d'octets, le plus souvent 512. C'est la plus petite unité d'information accessible physiquement en une opération sur le disque.

Cylindre.- Un **cylindre** est l'ensemble des pistes de même numéro. Un cylindre de disquette contient deux pistes et sur un disque dur $2p$ pistes, où p est le nombre de disques.

5.2.1.2 Unité d'allocation

Notion.- Nous avons vu que la plus petite unité d'information accessible physiquement en une seule opération est le secteur. On peut accéder à un secteur grâce au BIOS. Par contre, pour des raisons diverses, le système d'exploitation ne permet d'accéder en une seule opération qu'à un certain nombre de secteurs à la fois, nombre dépendant de la nature du disque.

Une **unité d'allocation** (en anglais *cluster*) est un groupe de secteurs que le système d'exploitation considère comme la plus petite unité d'information que l'on peut lire ou écrire à la fois sur le disque.

Conséquence.- Un fichier de 100 octets, suffisamment petit pour tenir sur un secteur, occupera 2 kiO sur un disque dont les unités d'allocation sont composés de quatre secteurs. C'est pour cette raison qu'il faut éviter les trop petits fichiers.

5.2.2 Lecture du contenu d'un disque avec `debug`

On peut, avec `debug`, lire le contenu d'un disque en commençant par en reporter le contenu en mémoire centrale.

Syntaxe.- Par exemple pour charger les 32 premiers secteurs de la disquette A à l'emplacement habituel 100h, on fait :

```
-L 100 0 0 20
```

La syntaxe se comprend bien sur cet exemple :

- `L` est la commande de `debug` pour charger un fichier (évidemment pour l'anglais *load*), comme nous l'avons déjà vu, ou des secteurs d'un disque, comme nous le voyons, en mémoire centrale ;

- le premier paramètre correspond à l'adresse de la mémoire vive, en hexadécimal, à laquelle on veut charger le contenu spécifié ; il s'agit ici de `CS:100`, ce qui est de toute façon l'adresse par défaut ;

- le deuxième paramètre est le numéro du disque, avec 0 pour A, 1 pour B, 2 pour C...

- le troisième paramètre est le numéro du premier secteur (relatif) à charger, en hexadécimal ; ici on veut lire à partir du secteur 0, c'est-à-dire au tout début du disque ;

- le dernier paramètre est le nombre de secteurs consécutifs à charger, en hexadécimal.

Bien entendu on affiche ensuite le contenu chargé grâce à la commande `dump` de `debug`.

Exemple.-

5.2.3 Lecture et écriture des secteurs : fonctions `25h` et `26h`

Les instructions de plus bas niveau concernant un disque sont la lecture et l'écriture d'un secteur. Au niveau du BIOS, on le désigne **physiquement**, c'est-à-dire en spécifiant la piste, la face et le secteur et en se servant de l'interruption `13h`.

Au niveau du système d'exploitation, on numérote linéairement les secteurs et on les désigne **logiquement**, c'est-à-dire en spécifiant le **numéro logique** du secteur, en se servant des interruptions `25h` et `26h`, remplacées plus tard par la fonction `44h`, de l'interruption `21h`.

5.2.3.1 Numérotation linéaire des secteurs

Un secteur est entièrement déterminé par son numéro de cylindre (commençant à 0), son numéro de face (commençant à 0) et son numéro de secteur (commençant à 1). Le système d'exploitation, pour pouvoir parler de secteurs consécutifs, numérote les secteurs linéairement en commençant par le numéro 0, correspondant au secteur 1 de la face 0 de la piste 0. On parle alors de **numérotation relative** ou de **numérotation logique**.

5.2.3.2 Description des interruptions

Cas général.- L'interruption `25h` permet de lire un secteur et l'interruption `26h` d'en écrire un. Le code est le suivant :

```
mov al, disque      ; 0 pour A, 1 pour B,...
mov bx, tampon      ; adresse du tampon contenant le secteur
mov cx, nombre      ; nombre de secteurs a lire ou a ecrire
mov dx, secteur     ; numero relatif du premier secteur
int 25h             ; pour lire, 26h pour ecrire
jc erreurs          ; traitement des erreurs
popf                ; redonne la valeur du registre des indicateurs
```

Ces deux interruptions ont une caractéristique inhabituelle : elles ne désempilent pas la valeur du registre des indicateurs, pourtant sauvegardée en début d'interruption. Ceci explique la dernière ligne de code.

En cas d'erreur, le bit CF du registre des indicateurs est positionné, d'où l'avant-dernière ligne de code, permettant d'en tenir compte. En cas d'erreur, le registre `AL` contient les informations suivantes :

Code	Erreur
10000000	Le disque ne répond pas
01000000	L'opération de recherche a échoué
00001000	Mauvais CRC lu sur le disque
00000100	Secteur non trouvé
00000011	Essai d'écriture sur un disque protégé en écriture
00000010	Autre erreur

Remarque importante.- Rappelons qu'il faut bien faire attention en écrivant sur un secteur car, sur un système d'exploitation npn protégé tel que MS-DOS, on peut écrire sur n'importe quel secteur, y compris les secteurs indispensables au système d'exploitation, et donc détruire des informations qui peuvent rendre le disque inutilisable pour le système.

Cas d'un disque de grande capacité.- Le fait de désigner un secteur par un seule registre, `DX`, implique que le disque ait une capacité inférieure à 32 MiO. À partir de MS-DOS 4.0 on peut cependant accéder à des disques de capacité supérieure. Le registre `DX` n'est alors plus utilisé, la valeur de `CX` doit être `FFFFh` et `DS:BX` doit pointer sur un bloc de 10 octets ayant la structure suivante :

```
00h-03h numéro de secteur sur 32 bits
04h-05h nombre de secteurs à lire ou à écrire
06h-07h décalage de l'adresse du tampon
08h-09h segment de l'adresse du tampon
```

Mais ceci ne fonctionne pas dans le cas d'un disque de capacité supérieure à 2 GiO.

5.2.3.3 Exemples

Exemple 1.- Écrivons un programme permettant de lire le premier secteur du répertoire de la disquette se trouvant dans le lecteur A et d'afficher son contenu à l'écran sur 16 lignes de 32 caractères.

```
TITLE lit_sect.asm
; lit le premier secteur du repertoire du disque A
      .model small
      .stack 256
      .data
tampon db 512 dup(?)
message db 'erreur de lecture', '$'
      .code
debut:
      mov ax, @data
      mov ds, ax

      mov al, 0             ; disque A
      mov bx, offset tampon ; pour entreposer
      mov cx, 1             ; un secteur
      mov dx, 19            ; le premier secteur du repertoire
      int 25h               ; lecture du secteur
      jc erreurs            ; en cas d'erreur
      popf
      jmp affiche

erreurs:
      mov dx, offset message
      mov ah, 9h
      int 21h               ; affiche erreur de lecture
      jmp fin

affiche:
      mov ah, 2h      ; affichage d'un caractere
      mov cl, 16      ; 16 lignes
nouvelle:
      mov dl, 13      ; passage a la ligne
      int 21h
```

```
      mov dl, 10
      int 21h
      mov ch, 32      ; 32 caracteres
ligne:
      mov dl, [bx]
      int 21h
      inc bx          ; caractere suivant
      dec ch
      cmp ch, 0
      jne ligne
      dec cl
      cmp cl, 0
      jne nouvelle
fin:
      mov ax, 4c00h
      int 21h
      end debut
```

En appliquant ce programme, par exemple, à la première disquette de la mise à jour de MS-DOS 6.0, on reconnaît les noms des fichiers, le reste apparaissant sous la forme de symboles graphiques.

Exemple 2.- Écrivons un programme permettant de lire le premier secteur d'un disque dur, volume C d'une capacité inférieure à 2 GiO, et d'afficher son contenu à l'écran sur 16 lignes de 32 caractères.

```
TITLE sect_dd.asm
; lit le premier secteur du disque C
      .model small
      .stack 256
      .data
message db 'erreur de lecture', '$'
tampon db 512 dup(?)
parmBlock label byte
sectorNumber dd 0
sectorCount dw 1
bufferOfs dw offset tampon
bufferSeg dw @data
      .code
debut:
      mov ax, @data
      mov ds, ax

      mov al, 0h              ; disque C
      mov bx, offset parmBlock
      mov cx, 0FFFFh
      int 25h                 ; lecture du secteur
      jc erreurs              ; en cas d'erreur
      popf
      jmp affiche
```

```
erreurs:
      mov dx, offset message
      mov ah, 9h
      int 21h                  ; affiche erreur de lecture
      jmp fin
affiche:
      mov ah, 2h               ; affichage d'un caractere
      mov cl, 16               ; 16 lignes
nouvelle:
      mov dl, 13
      int 21h
      mov dl, 10
      int 21h
      mov ch, 32               ; 32 caracteres
ligne:
      mov dl, [bx]
      int 21h
      inc bx                   ; caractere suivant
      dec ch
      cmp ch, 0
      jne ligne
      dec cl
      cmp cl, 0
      jne nouvelle
fin:
      mov ax, 4c00h
      int 21h
      end debut
```

5.2.4 Organisation logique d'un disque

L'un des rôles du système d'exploitation est de pouvoir accéder au contenu d'un disque de façon un peu plus conviviale qu'en utilisant les secteurs.

5.2.4.1 Organisation générale

Zone système et zone des données.- Un certain nombre de secteurs sont réservés au système d'exploitation pour conserver de l'information sur la façon dont le disque est organisé. Il s'agit des premiers secteurs, leur nombre variant suivant la nature du disque. On parle de la **zone système** (en anglais *system area*), les secteurs restants constituant la **zone des données** (en anglais *data area*).

Organisation de la zone système.- La zone système est elle-même, sous MS-DOS, constituée de trois sous-zones :

- Le **secteur d'amorçage** (en anglais *boot record*) contient les informations nécessaires au chargement des fichiers système s'ils sont présents. Il s'agit toujours du premier secteur, celui de numéro logique 0.

- La **table d'allocation des fichiers** (en anglais *file allocation table* ou FAT) permet de situer physiquement les fichiers repérés en MS-DOS par un nom. Cette table occupent les secteurs suivant immédiatement le secteur d'amorçage, leur nombre dépendant de la nature du disque.

- Le **répertoire** (en anglais *directory*) contient des informations sur les fichiers, les sous-répertoires étant considérés comme des fichiers.

Organisation de la zone des données.- La zone des données est formée des **fichiers système**, présents uniquement dans le cas d'un disque système, et des **fichiers utilisateurs**. Les fichiers système sont, dans le cas de MS-DOS, les fichiers cachés IO.SYS et MSDOS.SYS ; ils permettent de démarrer l'ordinateur avec le système d'exploitation situé sur le disque.

Résumé.- L'organisation logique d'un disque est donc la suivante :

Exemple.- Pour donner une idée, une disquette 3,5" de capacité 1,44 MiO a des unités d'allocation composées d'un seul secteur, a pour secteur d'amorçage celui de numéro relatif 0, une FAT s'étendant sur les secteurs 1 à 18 et un répertoire sur les secteurs 19 à 32.

5.2.5 Écriture et lecture des fichiers : fonctions `3Ch`, `3Dh`, `40h` et `3Eh`

L'écriture et la lecture d'un fichier se fait au niveau du système d'exploitation MS-DOS grâce à des fonctions de l'interruption `21h`. Dans les premières versions du DOS, l'accès se faisait grâce à des **blocs de contrôle de fichiers** (en anglais FCB pour *File Control Block*), ce qui ne permettait pas l'utilisation de sous-répertoires. Cette méthode est toujours implémentée, compatibilité ascendante oblige. Dans les versions suivantes, la notion de **descripteur** (en anglais *handle*) a été empruntée au système d'exploitation UNIX.

Dans le cas des fichiers avec descripteur, la création se fait à l'aide de la fonction `3Ch`, l'écriture d'un certain nombre d'octets avec la fonction `40h` et la fermeture avec la fonction `3Eh`.

5.2.5.1 Les deux façons de désigner un fichier

Nous supposons que le lecteur connaît un peu la manipulation des fichiers, pour l'avoir rencontrée lors de l'initiation à la programmation illustrée par un langage évolué.

Le *nom physique* d'un fichier (nom proprement dit ainsi que son chemin) est indiqué par ce qui est appelé une chaîne de caractères ASCIIZ. Le *nom logique* est le descripteur, que l'on peut voir comme un entier.

Nom physique.- Une `chaîne de caractère ASCIIZ` est une chaîne de caractères de moins de 128 caractères, sur l'alphabet des caractères permis pour désigner le nom complet d'un fichier en MS-DOS, et se terminant par le caractère nul. On aura, par exemple :

```
NOMFICH DB 'a:\data\essai.txt', 0
```

Le séparateur de chemin pouvant être '\' ou '/'.

Nom logique.- Le nom logique est un **descripteur**, donnée tenant dans un répertoire, donc d'au plus seize bits. Il est traditionnel de l'interpréter comme un entier, avec des numéros 00, 01, 02, 03, 04, 05...

Les cinq premiers descripteurs correspondent aux périphériques standard : 00 = entrée (clavier), 01 = affichage (moniteur), 02 = affichage des erreurs (moniteur), 03 = périphérique auxiliaire, 04 = imprimante.

Pour les autres fichiers, on doit demander au système d'exploitation de leur affecter un descripteur (valable pour la session en cours, plus exactement jusqu'à sa fermeture), grâce à la fonction `3Ch` (cas d'une création) ou `3Dh` (cas d'une simple ouverture). En général le premier fichier à être ouvert reçoit le descripteur 05, le second le descripteur 06...

Le PSP, en-tête d'un programme, contient une **table des descripteurs de fichiers** (en anglais *file handle table*) limitée à 20 numéros (d'où une limite pour le nombre de fichiers pouvant être ouverts simultanément dans un même programme) ; on peut accroître cette limite si besoin est (grâce à la fonction `67h`).

5.2.5.2 L'octet des attributs d'un fichier

Les bits 0 à 5 de l'**octet des attributs** d'un fichier spécifient les propriétés suivantes (un bit à 1 indique que le fichier possède la propriété en question) :

+ `01h` **lecture seule**, c'est-à-dire fichier pouvant seulement être lu ; tout essai d'écriture donne lieu à une erreur,

+ `02h` **fichier caché**, c'est-à-dire ne devant pas apparaître dans le répertoire (ou sous-répertoire) à l'écran lors de l'utilisation de la commande `dir`,

+ `04h` **fichier système**, n'apparaît pas non plus à l'écran lors de l'utilisation de la commande `dir`,

+ `08h` **nom du volume**, celui-ci occupant l'emplacement du nom du fichier et de son extension,

+ `10h` sous-répertoire,

+ `20h` fichier d'archive.

5.2.5.3 Création d'un fichier : fonction 3Ch

Syntaxe.- Pour créer un nouveau fichier, ou remplacer (en anglais *overwrite*) un ancien fichier de même nom physique, on utilise la fonction `3Ch` de l'interruption `21h`. Les paramètres en sont l'octet des attributs du fichier, contenu dans le registre `CX`, et l'adresse du nom physique, contenue dans le registre `DX`.

Dans le cas d'une opération valide, le système crée une entrée dans le répertoire avec l'attribut correspondant, met l'indicateur CF à zéro et renvoie le descripteur du fichier dans le registre `AX`. Si une erreur est survenue, CF est mis à 1 et un code d'erreur est renvoyé dans le registre `AX` :

```
03 : chemin non trouvé
04 : trop de fichiers ouverts
05 : accès non permis
```

Dans le dernier cas, ceci signifie soit que le répertoire est plein, soit que l'attribut de lecture seule est positionné. Remarquons une erreur de programmation fréquente : il faut absolument vérifier CF avant de rechercher la nature de l'erreur, sinon on risque de se retrouver avec une erreur qui n'en est pas une, le premier numéro attribué par le système, placé dans `AX`, étant 05.

Exemple.- Créons un fichier, vide, sur la disquette A :

```
TITLE creation.asm
; cree un fichier sur le disque A
      .model small
      .stack 256
      .data
message db 'erreur de creation', '$'
nomfich db 'a:\essai.txt', 0
numero  dw ?                        ; descripteur du fichier
      .code
debut:
      mov ax, @data
      mov ds, ax

      mov ah, 3Ch                   ; creation d'un fichier
      mov cx, 0                     ; attribut normal
      mov dx, offset nomfich        ; nom physique
      int 21h                       ; creation
      jc erreurs                    ; en cas d'erreur
      mov numero, ax                ; sauve le handle
      jmp fin
erreurs:
      mov dx, offset message
      mov ah, 9h
      int 21h                       ; affiche erreur de creation
fin:
      mov ax, 4c00h
      int 21h
      end debut
```

5.2.5.4 Ouverture d'un fichier : fonction 3D

Syntaxe.- Pour ouvrir un fichier, on utilise la fonction `3Dh` de l'interruption `21h`. Le registre `AL` doit contenir le **mode d'accès** :

```
0 : lecture seule
1 : écriture seule
2 : lecture-écriture.
```

Le registre `DX` doit contenir l'adresse du nom physique. Le fichier doit avoir été créé auparavant.

Si l'opération réussit, l'indicateur CF est mis à 0 et un descripteur de fichier est placé dans le registre `AX`. Sinon CF est mis à 1 et un code d'erreur est placé dans le registre `AX` :

```
02 : fichier non trouvé
03 : chemin non trouvé
04 : trop de fichiers ouverts
05 : accès refusé
12 : mode d'accès non valide.
```

Index de fichier.- Le système associe un **index** (en anglais *file pointer*) à chaque fichier ouvert. Il s'agit d'un entier naturel, commençant à 0, indiquant l'octet à écrire ou à lire. Toute opération d'écriture ou de lecture incrémente l'index du nombre d'octets adéquat.

Lorsqu'on crée un fichier, l'index est positionné à 0, ce qui fait que l'on détruit l'ancien fichier si on crée un nouveau fichier de même nom.

5.2.5.5 Fermeture d'un fichier : fonction 3Eh

Lorsqu'on a terminé d'écrire ou de lire un fichier, il faut le fermer. Ceci permet d'écrire les données qui sont encore dans un tampon (en mémoire centrale) et de mettre à jour la FAT et le répertoire avec la date et la taille du fichier.

Syntaxe.- On utilise pour cela la fonction `3Eh` de l'interruption `21h`, en plaçant le descripteur du fichier dans le registre `BX`.

En cas d'erreur, l'indicateur CF est positionné et le registre `AX` contient l'erreur :

```
06 : numéro de fichier non valide.
```

5.2.5.6 Écriture dans un fichier : fonction 40h

Syntaxe.- Pour écrire dans un fichier (ouvert, soit parce qu'il a été créé et pas refermé depuis, soit ouvert en écriture), on utilise la fonction `40h` de l'interruption `21h`. Le registre `BX` doit contenir le descripteur du fichier, le registre `CX` le nombre d'octets à écrire et le registre `DX` l'adresse du tampon à transférer.

Une opération d'écriture réussie écrit la chaîne de caractères sur le disque, incrémente l'index du fichier de la valeur correspondante, positionne CF à 0 et place le nombre d'octets réellement écrits dans le registre `AX`. Cette dernière information permet de savoir en particulier si le disque est plein, incident non considéré comme une erreur. Si l'opération ne réussit pas (à part l'incident qui vient juste d'être évoqué), l'indicateur CF est positionné à 1 et un code d'erreur est placé dans le registre `AX` :

```
05 : accès refusé
06 : descripteur (handle) non valide.
```

Exemple.- Créons un fichier sur la disquette A et plaçons-y un texte d'essai :

```
TITLE ecrire.asm
; ecrit sur un fichier sur le disque A
      .model small
      .stack 256
      .data
message  db 'erreur de creation', '$'
message2 db 'erreur en ecriture', '$'
nomfich  db 'a:\essai.txt', 0
donnee   db 'essai de texte ecrit sur un fichier'
numero   dw ?                      ; descripteur du fichier
      .code
debut:
      mov ax, @data
      mov ds, ax

      mov ah, 3Ch                  ; creation d'un fichier
      mov cx, 0                    ; attribut normal
      mov dx, offset nomfich       ; nom physique
      int 21h                      ; creation
      jc erreurs                   ; en cas d'erreur
      mov numero, ax               ; sauve le numero
      mov ah, 40h                  ; ecriture
      mov bx, numero               ; numero du fichier
      mov cx, 35                   ; nombre d'octets a ecrire
      mov dx, offset donnee        ; donnee a ecrire
      int 21h                      ; ecrire
      jc erreurs2                  ; en cas d'erreur d'ecriture
      mov ah, 3Eh
      int 21h                      ; fermeture
      jmp fin
erreurs:
      mov dx, offset message
      mov ah, 9h
      int 21h                      ; affiche erreur de creation
erreurs2:
      mov dx, offset message2
      mov ah, 9h
      int 21h
fin:
      mov ax, 4c00h
      int 21h
      end debut
```

5.2.5.7 Lecture dans un fichier : fonction 3F

Pour lire séquentiellement dans un fichier, il faut ouvrir ce fichier en lecture, lire les données voulues, puis le fermer.

Syntaxe.- Pour lire (séquentiellement) des données dans un fichier, on utilise la fonction `3Fh` de l'interruption `21h`. Le registre `BX` doit contenir le numéro du fichier, obtenu lors de l'ouverture, le registre `CX` le nombre d'octets à lire et le registre `DX` l'adresse mémoire à laquelle transférer les données.

Si l'opération réussit, l'index du fichier est incrémenté du nombre d'octets adéquat, l'indicateur CF est positionné à 0 et le nombre d'octets réellement lus est placé dans le registre `AX`. Si on essaie de lire au delà de la fin du fichier, 0 est placé dans le registre `AX` ; il s'agit d'une mise en garde et non d'une erreur. En cas d'erreur, l'indicateur CF est positionné à 1 et un code d'erreur est placé dans le registre `AX` :

```
05 : accès refusé
06 : numéro de fichier non valide.
```

Exemple.- Écrivons un programme permettant de récupérer le début du texte placé dans notre fichier ci-dessus :

```
TITLE lire.asm
; lit sur un fichier sur le disque A
      .model small
      .stack 256
      .data
message  db 'erreur en ouverture', '$'
message2 db 'erreur en lecture', '$'
nomfich  db 'a:\essai.txt', 0
donnee   db 40 dup(?)
numero   dw ?                    ; descripteur du fichier
      .code
debut:
      mov ax, @data
      mov ds, ax

      mov ah, 3Dh                ; ouverture d'un fichier
      mov al, 0                  ; lecture seulement
      mov dx, offset nomfich     ; nom physique
      int 21h                    ; ouverture
      jc erreurs                 ; en cas d'erreur
      mov numero, ax             ; sauve le numero
      mov ah, 3Fh                ; lecture
      mov bx, numero             ; numero du fichier
      mov cx, 30                 ; nombre d'octets a lire
      mov dx, offset donnee      ; sauvegarde de la donnee
      int 21h                    ; lire
      jc erreurs2                ; en cas d'erreur de lecture
      mov ah, 3Eh
      int 21h                    ; fermeture
      mov donnee[31], '$'        ; pour afficher facilement
```

```
      mov ah, 9h
      int 21h                       ; affichage de la donnee lue
      jmp fin
erreurs:
      mov dx, offset message
      mov ah, 9h
      int 21h                       ; affiche erreur en ouverture
erreurs2:
      mov dx, offset message2
      mov ah, 9h
      int 21h                       ; affiche erreur de lecture
fin:
      mov ax, 4c00h
      int 21h
      end debut
```

5.2.6 Accès direct : fonction 42h

L'**accès direct** (*random access* en anglais) signifie que l'on peut accéder directement à un élément du fichier, sans faire dérouler tous les éléments qui le précèdent. En MS-DOS on peut accéder à tout fichier soit séquentiellement, soit directement ; bien entendu, dans le cas d'un lecteur de cassette par exemple, l'accès direct est émulé.

Pour un accès direct dans un fichier, il faut qu'il soit créé, l'ouvrir (le plus souvent en lecture-écriture), placer l'index du fichier devant l'élément à lire ou à écrire, autant de fois que nécessaire, puis le fermer. Nous avons déjà vu comment créer un fichier, comment l'ouvrir, comment y lire, comment y écrire et comment le fermer ; il ne reste donc plus qu'à voir comment déplacer l'index.

Déplacement de l'index.- Lors de l'ouverture d'un fichier, l'index est égal à 0. Pour en changer la valeur, on se sert de la fonction `42h` de l'interruption `21h`. Le registre `AL` doit contenir la méthode d'accès :

```
00 : depuis le début du fichier
01 : depuis la position en cours de l'index
02 : depuis la fin du fichier.
```

Le registre `BX` doit contenir le descripteur du fichier, les registres `CX` et `DX` la valeur de déplacement de l'index conformément au format `CX:DX`.

Lorsque l'opération a réussi, l'indicateur CF est positionné à 0 et la nouvelle valeur de l'index est placée dans `DX:AX`. Sinon l'indicateur est positionné à 1 et le registre `AX` contient un code d'erreur :

```
01 : code d'accès non valide
06 : numéro de fichier non valide.
```

Exemple.- Écrivons un programme permettant de récupérer les caractères 10 à 19 du texte placé dans notre fichier ci-dessus :

```
TITLE direct.asm
; lecture a acces direct sur un fichier sur le disque A
      .model small
      .stack 256
      .data
message  db 'erreur en ouverture', '$'
message2 db 'erreur en lecture', '$'
message3 db 'erreur de positionnement', '$'
nomfich  db 'a:\essai.txt', 0
donnee   db 11 dup(?)
numero   dw ?                    ; descripteur du fichier
      .code
debut:
      mov ax, @data
      mov ds, ax

      mov ah, 3Dh                ; ouverture d'un fichier
      mov al, 0                  ; lecture seulement
      mov dx, offset nomfich     ; nom physique
      int 21h                    ; ouverture
      jc erreurs                 ; en cas d'erreur d'ouverture
      mov numero, ax             ; sauve le numero
```

```
      mov ah, 42h                ; deplacement de l'index
      mov al, 0                  ; depuis le debut du fichier
      mov bx, numero
      mov cx, 0
      mov dx, 10                 ; d'une valeur de 10
      int 21h
      jc erreurs3                ; erreur de deplacement

      mov ah, 3Fh                ; lecture
      mov bx, numero             ; numero du fichier
      mov cx, 10                 ; nombre d'octets a lire
      mov dx, offset donnee      ; sauvegarde de la donnee
      int 21h                    ; lire
      jc erreurs2                ; en cas d'erreur de lecture

      mov ah, 3Eh
      int 21h                    ; fermeture
      mov donnee[11], '$'        ; pour afficher facilement
      mov ah, 9h
      int 21h                    ; affichage de la donnee lue
      jmp fin

erreurs:
      mov dx, offset message
      mov ah, 9h
      int 21h                    ; affiche erreur en ouverture
erreurs2:
      mov dx, offset message2
      mov ah, 9h
      int 21h                    ; affiche erreur de lecture
erreurs3:
      mov dx, offset message3
      mov ah, 9h
      int 21h                    ; erreur de deplacement
fin:
      mov ax, 4c00h
      int 21h
      end debut
```

Applications.- On peut utiliser l'accès direct pour ajouter de nouvelles valeurs à un fichier existant : on l'ouvre en écriture et on place l'index à la fin avec le mode d'accès 2 et un déplacement de 0, puis on écrit. On obtient de la même façon la taille du fichier, en octets, qui est placé dans `DX:CX`.

Remarque.- Nous avons considéré le fichier comme un fichier d'octets. Bien entendu, en pratique, on veut le considérer comme un fichier d'éléments. Du moment que tous les éléments *ont la même taille*, on peut facilement émuler un accès direct, en déplaçant l'index d'un nombre entier de fois la taille d'un élément.

5.2.7 Gestion des sous-répertoires : fonctions 39h et 3Ah

Nous venons de voir comment créer, lire et écrire sur un fichier. Voyons maintenant comment gérer un ensemble de fichiers, c'est-à-dire les placer dans des sous-répertoires, détruire ceux-ci et les déplacer.

5.2.7.1 Création d'un sous-répertoire : fonction 39h

Description.- Pour créer un sous-répertoire, on utilise la fonction 39h de l'interruption 21h. Le registre DX doit contenir l'adresse d'une chaîne de caractères ASCIIZ spécifiant le disque et le chemin.

Si l'opération a réussi, l'indicateur CF est positionné à 0. Sinon il est positionné à 1 et le registre AX reçoit un code d'erreur :

```
03 : chemin non trouvé
05 : accès non permis.
```

On a donc l'analogue de la commande MKDIR du DOS, à part que le chemin complet doit être indiqué.

Exemple.- Écrivons un programme permettant de créer un sous-répertoire de nom « data » sur le disque A :

```
TITLE makedir.asm
; cree un sous-repertoire sur le disque A
      .model small
      .stack 256
      .data
message  db 'erreur de creation de repertoire', '$'
nomrep  db 'a:\data', 0
      .code
debut:
      mov ax, @data
      mov ds, ax

      mov ah, 39h                ; creation d'un
      mov dx, offset nomrep      ; sous-repertoire
      int 21h
      jc erreurs                 ; en cas d'erreur de creation
      jmp fin

erreurs:
      mov dx, offset message
      mov ah, 9h
      int 21h                    ; affiche erreur de creation

fin:
      mov ax, 4c00h
      int 21h
      end debut
```

5.2.7.2 Destruction d'un sous-répertoire : fonction 3Ah

En MS-DOS on ne peut détruire un sous-répertoire que s'il est vide, c'est-à-dire s'il ne contient aucun fichier ou sous-répertoire, et que s'il ne s'agit pas du répertoire actif, c'est-à-dire celui à partir duquel on lance l'opération.

Syntaxe.- Pour détruire un sous-répertoire, on utilise la fonction `3Ah` de l'interruption `21h`. Le registre `DX` doit contenir l'adresse d'une chaîne de caractères ASCIIZ spécifiant le disque et le chemin.

Si l'opération a réussi, l'indicateur CF est positionné à 0. Sinon il est positionné à 1 et le registre `AX` reçoit un code d'erreur :

```
03 : chemin non trouvé
05 : accès non permis
10h : essai de destruction du repertoire racine.
```

On a donc l'analogue de la commande `RMDIR` du DOS, à part que le chemin complet doit être indiqué.

Exemple.- Écrivons un programme permettant de détruire le sous-répertoire de nom « data » sur le disque A :

```
TITLE rmovdir.asm
; detruit un sous-repertoire sur le disque A
      .model small
      .stack 256
      .data
message  db 'erreur de destruction de repertoire', '$'
nomrep  db 'a:\data', 0
      .code
debut:
      mov ax, @data
      mov ds, ax

      mov ah, 3Ah                ; destruction d'un
      mov dx, offset nomrep      ; sous-repertoire
      int 21h
      jc erreurs                 ; en cas d'erreur de
                                 ; destruction
      jmp fin

erreurs:
      mov dx, offset message
      mov ah, 9h
      int 21h                    ; affiche erreur

fin:
      mov ax, 4c00h
      int 21h
      end debut
```

5.2.8 Fonctions 41h et 56h de destruction et de déplacement des fichiers

Nous avons vu comment créer, lire et écrire sur un fichier. Voyons maintenant comment gérer un ensemble de fichiers, c'est-à-dire les détruire et les déplacer.

5.2.8.1 Fonction 41h de destruction d'un fichier

Syntaxe.- Pour détruire un fichier, qui n'est pas en lecture seule, on utilise la fonction `41h` de l'interruption `21h`. Le registre `DX` doit contenir l'adresse d'une chaîne de caractères ASCIIZ spécifiant le disque, le chemin et le nom du fichier.

Si l'opération a réussi, l'indicateur CF est positionné à 0, indique dans le répertoire que le fichier est écrasé et marque les secteurs associés de la FAT comme étant à nouveau disponible. En cas d'erreur, l'indicateur CF est positionné à 1 et le registre `AX` reçoit un code d'erreur :

```
02 : fichier non trouvé
03 : chemin non trouvé
05 : accès non permis.
```

On a donc l'analogue des commandes `erase` et `delete` du DOS, à part que le chemin complet doit être indiqué.

Exemple.- Écrivons un programme permettant de détruire le fichier de nom « essai.txt » sur le disque A :

```
TITLE effacer.asm
; detruit un fichier sur le disque A
      .model small
      .stack 256
      .data
message  db 'erreur de destruction de fichier', '$'
nomfich  db 'a:\essai.txt', 0
      .code
debut:
      mov ax, @data
      mov ds, ax

      mov ah, 41h                ; destruction d'un
      mov dx, offset nomfich     ; fichier
      int 21h
      jc erreurs                 ; en cas d'erreur de
                                 ; destruction
      jmp fin

erreurs:
      mov dx, offset message
      mov ah, 9h
      int 21h                    ; affiche erreur

fin:
      mov ax, 4c00h
      int 21h
      end debut
```

5.2.8.2 Fonction 56h de renommage et de déplacement d'un fichier ou d'un sous-répertoire

Syntaxe.- Pour renommer (ou déplacer) un fichier, ou un sous-répertoire, on utilise la fonction `56h` de l'interruption `21h`. Les registres `DS:DX` doivent contenir l'adresse d'une chaîne de caractères ASCIIZ spécifiant le disque, le chemin et l'ancien nom du fichier tandis que les registres `ES:DI` doivent contenir l'adresse d'une chaîne de caractères ASCIIZ spécifiant le disque, le chemin et le nouveau nom du fichier.

Si l'opération a réussi, l'indicateur CF est positionné à 0. Sinon CF est positionné à 1 et le registre `AX` reçoit un code d'erreur :

```
02 : fichier non trouvé
03 : chemin non trouvé
05 : accès non permis
11h : pas sur le même disque.
```

On a donc l'analogue des commandes `rename` et `move` du DOS, à part que le chemin complet doit être indiqué.

Exemple.- Écrivons un programme permettant de changer le nom du fichier « essai.txt » en « texte.txt » sur le disque A :

```
TITLE deplacer.asm
; change le nom d'un fichier sur le disque A
      .model small
      .stack 256
      .data
message    db 'erreur de deplacement de fichier', '$'
anciennom  db 'a:\essai.txt', 0
nouveaunom db 'a:\texte.txt', 0
      .code
debut:
      mov ax, @data
      mov ds, ax
      mov es, ax                  ; pour le nouveau nom

      mov ah, 56h                 ; deplacer un
      mov dx, offset anciennom    ; fichier
      mov di, offset nouveaunom
      int 21h
      jc erreurs                  ; en cas d'erreur de
                                  ; deplacement
      jmp fin

erreurs:
      mov dx, offset message
      mov ah, 9h
      int 21h                     ; affiche erreur

fin:
      mov ax, 4c00h
      int 21h
```

```
end debut
```

5.3 Bibliographie

[G-D-86] Golden, Donald & Pechura, Michael, *The structure of microcomputer file systems*, **Communications of the ACM**, Volume 29, Issue 3, March 1986, pp. 222–230.