Finalement, les mainframes ont déployé de complexes systèmes hiérarchiques de gestion de fichiers, dont celui de MULTICS qui en fut sans doute le point culminant.

Les premiers systèmes de fichiers pour mini-ordinateurs ne supportaient également qu'un unique répertoire ; il en était de même sur les premiers micro-ordinateurs dont CP/M était le système d'exploitation dominant.

La mémoire virtuelle

La mémoire virtuelle (c'est-à-dire la possibilité de faire exécuter des programmes nécessitant une taille mémoire supérieure à celle de la mémoire physique de la machine) a connu un développement similaire. Elle est apparue tout d'abord sur les mainframes, puis sur les minis, sur les micros, etc. Nous étudions en détail cette technique au chapitre 3. MULTICS fut le premier système qui introduisit la mémoire virtuelle qu'on trouve aujourd'hui sur la plupart des systèmes UNIX et Windows.

De cette analyse, on constate que les innovations réalisées dans un certain contexte deviennent obsolètes lorsque le contexte a changé (ce fut le cas du langage d'assemblage, de la monoprogrammation, du répertoire unique, etc.) et qu'on les retrouve souvent dans des contextes différents une dizaine d'années plus tard. C'est pourquoi dans cet ouvrage nous présentons une idée, un concept ou un algorithme, qui peuvent quelquefois sembler datés à l'époque des PC modernes, alors qu'on les retrouve sur les systèmes embarqués et les smart cards.

1.6 Les appels système

Nous avons vu qu'un système d'exploitation a deux grandes fonctions : offrir des abstractions aux programmes utilisateur et gérer les ressources de l'ordinateur. L'interaction avec les programmes utilisateur constitue l'activité majeure du système d'exploitation, comme la création, l'écriture, la lecture et la destruction des fichiers. Quant à la gestion des ressources, transparente à l'utilisateur, elle est automatique.

L'interface entre programmes utilisateur et système d'exploitation traite des abstractions. Elle est définie par l'ensemble des appels systèmes fournis par le système d'exploitation. Pour comprendre vraiment ce que fait un système d'exploitation, il faut donc examiner soigneusement cette interface. Les appels varient d'un système à l'autre, mais les concepts sous-jacents sont souvent assez proches.

Il faut souvent faire un choix entre (1) des généralités (« les systèmes d'exploitation ont des appels système pour lire un fichier ») et (2) un système spécifique (« UNIX définit l'appel système `read()` pour lire un fichier, avec trois paramètres : un pour spécifier le fichier, un pour indiquer où doivent être placées les données lues et un pour le nombre d'octets à lire »).

Nous avons opté pour la seconde approche. C'est un plus gros travail, mais cela donne une idée plus précise de la réalité. Même si ce qui va être décrit fait explicitement référence à POSIX (ISO 9945-1) et donc à UNIX, SystemV, BSD, Linux, MINIX 3, etc., la plupart des systèmes d'exploitation modernes possèdent des appels

système équivalents. Comme la façon dont l'appel système est effectivement réalisé est extrêmement dépendante de la plate-forme, et qu'il est souvent écrit en assembleur, une procédure de bibliothèque est fournie afin d'effectuer l'appel système depuis un programme C (et souvent aussi depuis d'autres langages).

Il est important de se rappeler qu'un ordinateur monoprocesseur ne peut exécuter qu'une instruction à la fois. Si un processus utilisateur a besoin d'un service système, par exemple la lecture de données dans un fichier, il doit exécuter un déroutement pour transférer le contrôle au système d'exploitation. Ce dernier détermine ce que veut le processus à l'aide des paramètres de l'appel. Il effectue alors l'appel et rend le contrôle au processus appelant à l'instruction suivant l'appel système. Un appel système est en ce sens analogue à un appel procédural classique, à ceci près qu'il est effectué en mode noyau.

Pour rendre les choses plus claires, prenons le cas de `read` et de ses trois paramètres. Comme tous les appels système, il est invoqué depuis un programme C par une fonction de même nom :

```
cpt = read(df, tampon, nbOctets);
```

L'appel système et la fonction associée renvoient le nombre d'octets effectivement lus, récupéré ici dans `cpt`. Cette valeur est normalement égale à `nbOctets`, mais peut être inférieure, par exemple si la fin du fichier est atteinte avant le nombre indiqué d'octets.

Si l'appel système ne peut être mené à bien (valeur incorrecte d'un paramètre ou erreur disque), la fonction renvoie –1 et le code (entier) de l'erreur est placé dans une variable globale, `errno`. On doit toujours vérifier le succès d'un appel système dans un programme.

Les appels système sont découpés en étapes. Reprenons l'exemple de `read`. Avant l'appel de la fonction (qui déclenche l'appel système), le programme appelant empile les paramètres sur la pile, comme le montrent les étapes 1 à 3 de la figure 1.17. Les compilateurs C et C++ utilisent un ordre inverse pour l'empilement, pour des raisons historiques relatives à la fonction `printf` et à son premier paramètre (la chaîne de format). Le premier et le troisième paramètre sont appelés par valeur, mais le deuxième est passé par référence : l'adresse du tampon (et non son contenu) est transmise. L'appel effectif à la fonction a alors lieu (étape 4). Cette fonction de bibliothèque, le plus souvent écrite en assembleur, place le numéro de l'appel système à un endroit convenu, par exemple un registre (étape 5). Elle exécute alors une instruction `TRAP` pour basculer en mode noyau et démarrer l'exécution à une adresse fixée du noyau (étape 6). Le code noyau examine le numéro d'appel système, puis donne la main à l'appel système concerné en utilisant une table de pointeurs indicée sur les numéros d'appels système (étape 7). Le code de l'appel s'exécute alors (étape 8). Une fois l'exécution terminée, le contrôle peut être rendu à la fonction de bibliothèque (dans l'espace utilisateur), à l'instruction qui suit le `TRAP` (étape 9). Cette fonction rend alors le contrôle au programme utilisateur comme une procédure ordinaire (étape 10). Ce dernier doit enfin terminer l'opération en nettoyant la pile, comme pour n'importe quelle invocation de procédure (étape 11). En supposant que la pile s'étende vers les adresses basses (comme c'est le cas le plus fréquent), le code doit

incrémenter le pointeur de pile de la quantité nécessaire pour supprimer les paramètres empilés juste avant l'appel à `read`. Le programme peut alors continuer son exécution normalement.

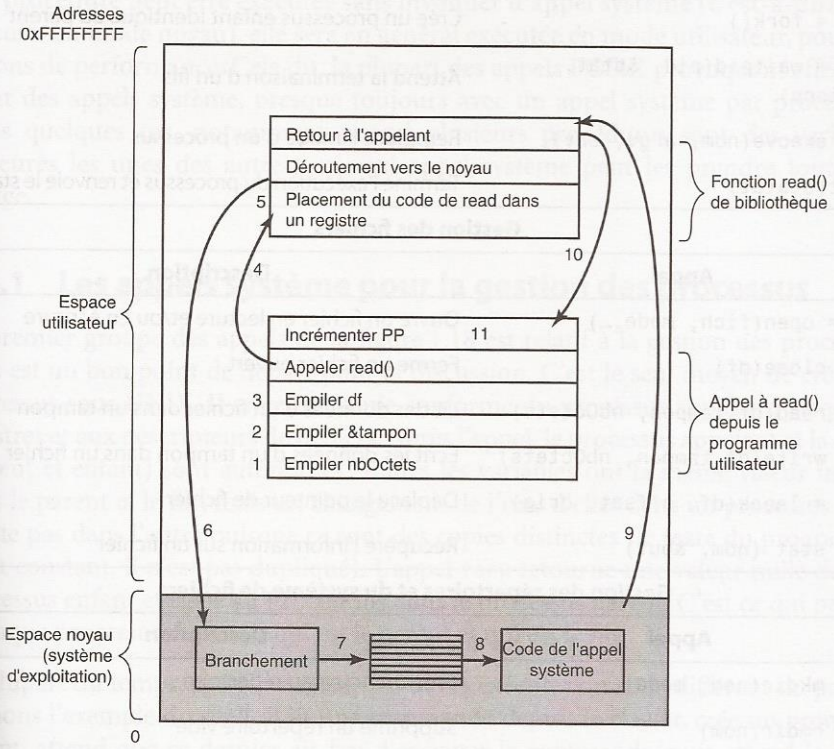

Figure 1.17 • Les 11 étapes de réalisation de l'appel système `read(df, tampon, nbOctets)`.

Dans l'étape 9, l'appel système peut bloquer l'appelant, l'empêchant de continuer. Par exemple, si la lecture se fait depuis le clavier et que rien n'a encore été tapé par l'utilisateur, l'appelant doit être bloqué. Le système va alors chercher si un autre processus peut s'exécuter. Plus tard, quand les données d'entrée seront disponibles, ce processus redeviendra candidat et les étapes 9 à 11 pourront se dérouler.

Dans les sections qui suivent, nous examinons les principaux appels POSIX, plus précisément les fonctions de bibliothèque qui leur correspondent. POSIX possède environ 100 appels de procédures. Les plus importants sont listés à la figure 1.18, et regroupés en quatre catégories. Nous examinerons brièvement chacune d'elles pour savoir son rôle. Dans une large mesure, les services proposés par ces procédures recouvrent la plupart des activités du système d'exploitation puisque la gestion des ressources sur un ordinateur personnel est minimale (du moins si on la compare à ce qu'elle est sur des serveurs multi-utilisateurs). Ces services couvrent la gestion des processus, des fichiers et répertoires, et des entrées/sorties.