éduSCOL
Informer et accompagner les professionnels de l'éducation

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

MINISTÈRE DE L'ÉDUCATION NATIONALE ET DE LA JEUNESSE

VOIE GÉNÉRALE

2DE | 1RE | TLE

Numérique et Sciences informatiques

ENSEIGNEMENT
SPÉCIALITÉ

# SYSTÈMES DE TYPE UNIX : STRUCTURES DE DONNÉES ET ALGORITHMES

- Histoire de l'informatique
- Représentation des données
- Traitement des données
- Interactions entre l'homme et la machine sur le Web
- Architectures matérielles et systèmes d'exploitation
- Langages et programmation
- Algorithmique

## 1. Description rapide et enjeux

L'utilisateur de base d'un ordinateur (ou d'un téléphone, ou d'une console de jeux, etc.) est en général conscient de l'existence de deux types d'entités sur son équipement :

- des fichiers qui sont des données statiques (ex : photo, fichier pdf, film), souvent modifiables grâce à des logiciels, mais pas toujours (ex : jeu sur une cartouche de console de jeux) ;
- des processus qui sont des applications en train de tourner, donc des données dynamiques : leur état est modifié d'un instant à l'autre, sans nécessiter l'intervention de l'utilisateur.

Dans les notes qui suivent nous introduisons des problématiques de gestion de ces types de données et certaines solutions qui existent ou ont pu exister.

### 1.1. Qu'est-ce-qu'un système d'exploitation ?

La réponse à cette question a un léger côté subjectif, mais globalement voici ce qu'on attend d'un système d'exploitation :

- faire l'interface entre le matériel et l'utilisateur (et par "utilisateur" on entend aussi bien l'utilisateur humain que des applications non système) ; exemple :
  - communication de l'utilisateur vers le matériel : on appuie sur une touche du clavier et on s'attend à ce que le caractère concerné apparaisse à l'écran ;
  - communication du matériel vers l'utilisateur : si un fichier qu'on essaye d'atteindre n'existe pas, on a un message d'erreur.
- isoler le code utilisateur du matériel, permettant ainsi par exemple
  - de changer le matériel de façon transparente (ou presque) pour l'utilisateur (il faudra parfois installer des modules ou drivers supplémentaires, mais une fois cela fait, on continue à fonctionner comme précédemment) ;
  - d'assurer l'intégrité des données : par exemple le système assure qu'un utilisateur autre que `root` ne peut accéder à des données pour lesquelles il n'a pas les droits d'accès ; mais il assure aussi qu'aucun utilisateur ne puisse détruire la

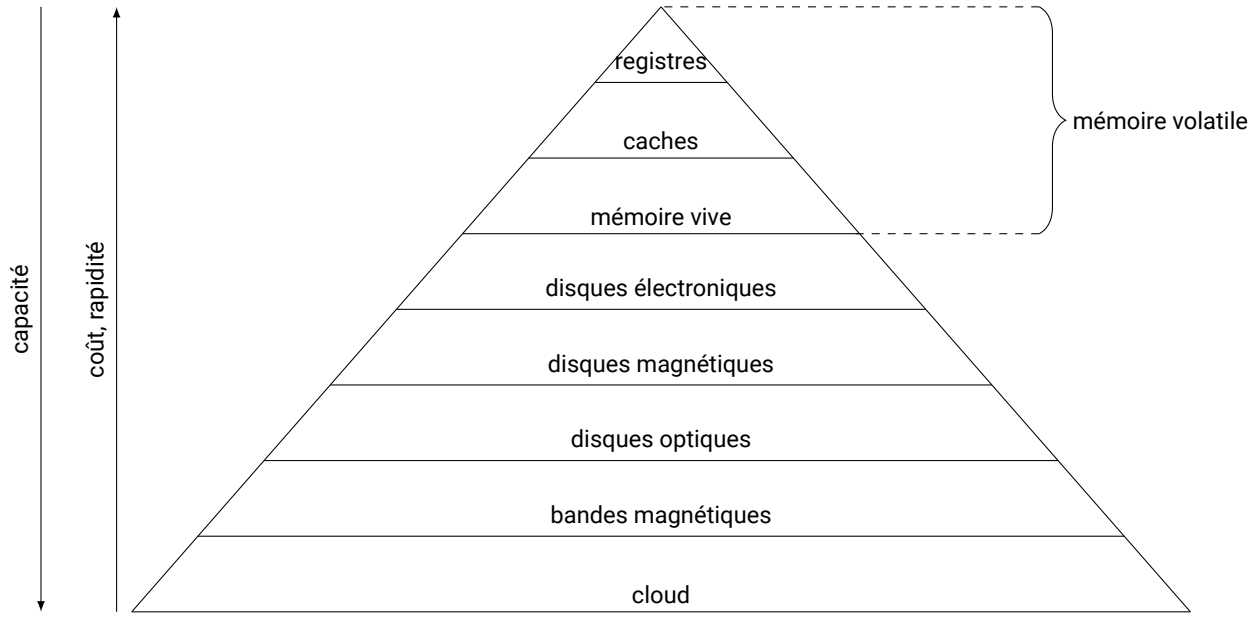


FIGURE 1 – Les divers types de mémoires d'un ordinateur : à l'arrêt de la machine, le contenu de la mémoire volatile disparaît, le contenu des autres mémoires reste.

cohérence de l'ensemble des données (par exemple les informations qui permettent de reconstituer le SGF à chaque redémarrage) ;

- ▷ faire en sorte que tous les programmes puissent s'exécuter de façon équitable (un programme qui demande l'utilisation d'une ressource y aura accès en un temps "raisonnable").

Dans ces notes on s'intéresse essentiellement à certains mécanismes mis en œuvre par les systèmes de type `UNIX` pour atteindre ces buts. Notre source principale est [1]. Les systèmes modernes sont beaucoup plus compliqués que ce que nous décrirons (par exemple dans ces notes ne seront pas évoqués les systèmes multi-processeurs ou les problématiques liées au *cloud*), mais les questions posées ici sont toujours d'actualité.

On peut schématiser le fonctionnement d'un ordinateur à l'aide de couches qui communiquent entre elles de la façon suivante :

utilisateur — *appels systèmes* ↓ / *signaux* ↑

noyau

*interface matériel* ↓ / *interruptions* ↑ — matériel

Si un processus tente d'accéder au matériel, alors qu'il n'en a pas l'autorisation, le noyau tue ce processus (c'est le fameux *segmentation fault* que peuvent rencontre les programmeurs C).

## 1.2. Ressources et mémoire

Pour comprendre tous les mécanismes déployés par un système d'exploitation, il faut avoir en tête le schéma concernant les divers types de mémoire sur un ordinateur, présenté en figure 1.

**registres** emplacement mémoire interne à un processeur, rapide d'accès mais petits (quelques dizaines d'octets en général)

**caches** également sur le processeur, sert à stocker temporairement des instructions ou des données et à accélérer les traitements en limitant les accès à la mémoire vive (accéder au cache : 5 à 10 fois plus rapide qu'accéder à la ram)

**mémoire vive/ram** mémoire volatile, assez rapide d'accès (taille : de l'ordre du Go)

Le noyau a intérêt à avoir sous la main, c'est-à-dire dans la mémoire la plus rapide d'accès (au plus haut dans le schéma de la figure 1), les données dont il a besoin. De façon générale, quand le noyau a besoin d'une ressource (un fichier ou un exécutable par exemple), il la rapatrie dans la mémoire volatile. Il travaille sur cette ressource et, si besoin, met à jour la mémoire permanente par

la suite. C'est ce qui explique que si un ordinateur s'éteind d'un coup et qu'on n'a pas sauvegardé le contenu d'un traitement de texte par exemple, on ne retrouve pas tout ce qui a été écrit au moment du redémarrage.
Malheureusement, plus la mémoire est rapide, plus elle est chère et donc plus sa capacité est réduite. Il est donc impossible de mettre les contenus de tous les fichiers par exemple dans la mémoire volatile ; une partie du travail du système d'exploitation est de gérer le contenu de la mémoire volatile, mais aussi de faire en sorte que les applications ignorent où se trouvent les données qu'elles traitent.
Ce qui fait la vitesse d'un ordinateur ce n'est pas tant que la rapidité du CPU que la taille du cache : le CPU peut être aussi rapide qu'on veut, s'il n'y a pas de cache la machine sera lente. Pourquoi alors ne pas mettre les moyens et faire un cache beaucoup plus grand ? Ce serait au bout d'un moment préjudiciable au niveau rapidité puisque plus une ressource est grande, plus c'est compliqué de trouver des données à l'intérieur.

## 2. Fichiers

Dans l'autre poly sur le systèmes, il est question d'i-nœuds : un i-nœud contient toutes les informations concernant un fichier, sauf son nom. Il existe en fait deux types d'i-nœuds :

- l'i-nœud *sur disque* (dans la partie non volatile de la mémoire) : ces i-nœuds sont rangés dans un tableau sur le disque, le numéro d'i-nœud est en fait le numéro de la case dans ce tableau ; ces i-nœuds sont utiles pour conserver des données ;
- l'i-nœud *en mémoire* (dans la partie volatile de la mémoire) : il contient en gros les mêmes types de renseignements, plus le numéro d'i-nœud associé ; ces i-nœuds sont beaucoup plus rapides d'accès, mais il y en a moins.

Tout le travail se fait en mémoire. Quand l'utilisateur fait une sauvegarde, les modifications sont alors recopiées sur le disque.

Supposons qu'un utilisateur veuille modifier un fichier avec un éditeur de texte : il le désigne par son nom. Le système trouve alors le numéro d'i-nœud associé en parcourant le répertoire qui contient le fichier. Si l'i-nœud ne se trouve pas en mémoire, le système le charge en mémoire, puis vérifie les droits d'accès. Le système transmet alors un descripteur correspondant à une ouverture du fichier.

## 3. Processus

Les systèmes de type UNIX sont multi-tâches : l'utilisateur a l'impression que plusieurs tâches peuvent s'exécuter en même temps. Un processeur ne peut pourtant exécuter qu'un seul processus à la fois, qu'on appelle *processus actif*. L'ensemble de la mémoire associé à un processus (le code qu'il exécute, la pile d'appel, les données, le compteur qui lui permet de savoir où il en est de son exécution, etc.) est appelé son *contexte*. L'ordonnanceur change régulièrement de processus actif en procédant à un changement de contexte. Ainsi le nouveau processus actif n'a pas conscience qu'il a été interrompu et continue là où il en était.
Un des algorithmes classiques d'ordonnancement de processus est appelé *round robin*. On peut imaginer que les processus sont rangés dans une liste chaînée circulaire et à chaque tic d'horloge l'ordonnanceur donne la main au processus suivant dans cette liste. Si le processus se termine avant le tic suivant, il est supprimé de la liste, sinon il y reste.

## 4. Systèmes propriétaires vs. systèmes libres

Il existe divers systèmes d'exploitation et tous n'apportent pas les mêmes solutions aux questions de la section 1.1. Indépendemment des solutions apportées, il existe deux familles de systèmes : les systèmes propriétaires et les systèmes libres. La différence essentielle est que le code d'un logiciel libre (et donc d'un système libre) est public. On peut en général le modifier ou s'en servir pour fabriquer de nouveaux produits (il est quand même prudent de connaître la licence sous laquelle le logiciel a été publié qui peut préciser ces droits et des obligations du type citer les auteurs originaux ou non, pourvoir fabriquer des logiciels propriétaires ou non, etc.). Les logiciels propriétaires sont en général non ouverts, il est donc plus difficile (voire illégal) de les modifier. Les logiciels libres sont souvent maintenus par la communauté, mais peuvent aussi l'être par des entreprises qui les utilisent et qui ont intérêt à ce qu'ils restent efficaces et utilisés par d'autres, ce qui assure l'existence de développeurs susceptibles de participer à leur maintien. Les logiciels propriétaires quant à eux sont essentiellement développés et mis à jour par l'entreprise qui les possède et qui peut décider d'arrêter de les maintenir. Ce sont deux modèles économiques très différents.

# Références

[1] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.